# An Optimal Bloom Filter Replacement*

Anna Pagh[†] Rasmus Pagh[†] S. Srinivasa Rao[‡]

**Abstract**

This paper considers space-efficient data structures for storing an approximation $S'$ to a set $S$ such that $S \subseteq S'$ and any element not in $S$ belongs to $S'$ with probability at most $\epsilon$. The *Bloom filter* data structure, solving this problem, has found widespread use. Our main result is a new RAM data structure that improves Bloom filters in several ways:

- The time for looking up an element in $S'$ is $O(1)$, *independent of* $\epsilon$.
- The space usage is within a lower order term of the lower bound.
- The data structure uses *explicit* hash function families.
- The data structure supports insertions *and deletions* on $S$ in amortized expected constant time.

The main technical ingredient is a succinct representation of dynamic multisets. We also consider three recent generalizations of Bloom filters.

## 1 Introduction

In many applications where a set $S$ of elements (from a finite universe) is to be stored, it is acceptable to include in the set some *false positives*, i.e., elements that appear to be in $S$ but are not. For example, on the question of whether a remote server contains a desired piece of information, it is acceptable that there is a small probability that we are wrongly led to believe that this is indeed the case, since the only cost would be making an unfruitful request for the item. Storing only an approximation to a set can yield great dividends compared to storing the set explicitly. If we only require that every element not in $S$ is a false positive with probability at most $\epsilon$, the number of bits needed to store the approximation is roughly $n \log(1/\epsilon)^1$, where $n = |S|$ and the logarithm has base 2 [5]. In contrast, the amount of space for storing $S \subseteq \{0,1\}^w$ explicitly is at least $\log \binom{2^w}{n} \geq n(w - \log n)$ bits, which may be much larger if $w$ is large. Here, we consider subsets of the set of $w$-bit machine words on a standard RAM model, since this is the usual framework in which RAM dictionaries are studied.

Let the (random) set $S' \supseteq S$ consist of the elements that are stored (including false positives). We want $S'$ to be chosen such that any element not in $S$ belongs to $S'$ with probability at most $\epsilon$. For ease of exposition, we assume that $\epsilon \leq 1/2$ is an integer power of 2. A Bloom filter [1] is an elegant data structure for selecting and representing a suitable set $S'$. It works by storing, as a bit vector, the set

$$I_S = \{h_i(x) \mid x \in S, \, i = 1, \ldots, k\}$$

where $h_1, \ldots, h_k : \{0,1\}^w \rightarrow [n \log(1/\epsilon) \log e]$ are assumed to be truly random hash functions, and $k = \log(1/\epsilon)$. The set $S'$ consists of those $x \in \{0,1\}^w$ for which $\{h_i(x) \mid i = 1, \ldots, k\} \subseteq I_S$. Looking up a key in $S'$ requires $k = \log(1/\epsilon)$ hash function evaluations and memory lookups. Insertions are handled by setting $k$ bits in the bit vector to 1. Deletions, on the other hand, are not supported: Setting any bit to 0 might exclude from $S'$ some other element in $S$. In [10] the authors instead store the *multiset* defined as $I_S$ above, which makes it possible to support deletions. However, this incurs a $\log \log n$ factor space overhead, as single bits are replaced by small counters.

Bloom filters have found many applications, in theory and practice, in particular in distributed systems. We refer to [2] for an overview of applications. In spite of its strength, the data structure has several weaknesses:

1. **Dependence on $\epsilon$.** The lookup time grows as the false positive rate decreases. For example, to get a false positive rate below 1%, 7 memory accesses are needed. If $S$ is large this will almost surely mean 7 cache misses, since the addresses are random hash function values.

[*] This work was initiated during Dagstuhl seminar No. 04091 on Data Structures, 2004.

[†] IT University of Copenhagen, Rued Langgaards Vej 7 2300 København S, Denmark Email: {annao,pagh}@itu.dk

[‡] Department of Computer Science, University of Waterloo, Ontario, N2L 3G1, Canada Email: ssrao@monod.uwaterloo.ca

[1] This assumes that $n < 2^w/2$ and that $\epsilon$ is not very small. For $\epsilon$ less than about $n/2^w$ we might as well store the exact set $S$.

2. **Suboptimal space usage.** The space usage is a factor $\log e \approx 1.44$ from the information theoretically best possible.

3. **Lack of hash functions.** There is no known way of choosing the hash functions such that the scheme can be shown to work, unless $O(n \log n)$ bits of space is used [14]. The use of cryptographic hash functions has been suggested in e.g. [6]. However, since the range of the hash functions is small, it is computationally easy to find a set $S$ for which any fixed set of (efficiently computable) hash functions behave badly, i.e., yield too large a set $S'$.

4. **No deletions.** Deletions are not supported (unless modified to use asymptotically more space than the minimum possible, as in [10]).

Issues 1 and 2 were considered in [12], where it was observed that increasing the size of the bit vector makes it possible to reduce the number of hash functions. In particular, if $O(n/\epsilon)$ space is used, a single hash function suffices. In distributed applications, the data structure may be compressed to the optimal $n \log(1/\epsilon) + O(n)$ bits before being transferred over the network. As most previous results, this assumes that truly random hash functions are available, and deletions are not supported. Solutions based on space-optimal minimal perfect hashing [11], mentioned e.g. in [2], resolve issues 1–3, but fail to support insertions as well as deletions.

In this paper we revisit a reduction from the approximate membership problem to the exact membership problem. The reduction, first described in [5], corresponds to the extreme case with a single hash function in [12]. Using the reduction with succinct solutions to the exact membership problem, e.g. [16], addresses issues 1–3. To obtain a dynamic solution, resolving all issues above, a succinct dynamic multiset representation of independent interest is developed.

Theorem 1.1. *A dynamic multiset of $n$ elements from $[u]$, supporting lookup, insertion and deletion, can be implemented using $B + o(B) + O(n)$ bits of space, where $B = \log\lceil\binom{u+n}{n}\rceil = n\log(u/n) + \Theta(n)$ is the information theoretic space lower bound. Insertions and deletions can be performed in amortized expected constant time and lookup in worst case constant time. Reporting the number $c$ of occurrences of an element takes time $O(1 + \log c)$.*

The above theorem implies our main result:

Theorem 1.2. *Let a positive integer $n$ and $\epsilon > 0$ be given. On a RAM with word length $w$ we can maintain a data structure for a dynamic multiset $M$ of size at most $n$, with elements from $\{0,1\}^w$, such that:*

- *Inserting in $M$ and deleting from $M$ can be done in amortized expected constant time. The data structure does* not *check that deletions are valid.*
- *Looking up whether a given $x \in \{0,1\}^w$ belongs to $M$ can be done in worst case constant time. If $x \in M$ the answer is always 'yes'. If $x \notin M$ the answer is 'no' with probability at least $1 - \epsilon$.*
- *The space usage is at most $(1 + o(1))\, n \log_2(1/\epsilon) + O(n + w)$ bits.*

The reduction from [5] is described in Section 2. We show how to extend a known result about succinct set representations [19] to succinct multiset representations, by a general reduction described in Section 4. The reduction uses a result on redundant binary counters described in Section 3.

The theorem is mainly a theoretical contribution, and the data structure is probably only competitive with Bloom filters for rather small $\epsilon$. In section 5 we describe an alternative data structure based on [8], for which no provably good choice of hash function is known, but which is likely to improve on Bloom filters in practice even for large $\epsilon$. This data structure allows only insertions, not deletions. Finally, in section 6 we obtain new results on three recent Bloom filter generalizations [6, 9, 17].

## 2 From approximate to exact set membership

Choose $h$ as a random function from a universal class of hash functions [4] mapping $\{0,1\}^w$ to $[n/\epsilon]$. The function can be chosen in constant time and stored in space $O(w)$ bits. For any set $S \subseteq \{0,1\}^w$ of at most $n$ elements, and any $x \in \{0,1\}^w \backslash S$ it holds that

$$\Pr[h(x) \in h(S)] \leq \sum_{y \in S} \Pr[h(x) = h(y)] \leq n/(n/\epsilon) = \epsilon \ .$$

(The first inequality is by the union bound, and the second is by definition of universality.) This was observed in [5], along with the implication that we can solve the approximate membership problem simply by storing the set $h(S)$. If a space optimal set representation is used, the resulting space usage is $n \log(1/\epsilon) + \Theta(n + w)$ bits (see [3]), which is within $\Theta(n + w)$ bits of the best possible.

It is easy to see that if we instead store the *multi*set $h(S)$, we have enough information to maintain $h(S)$ under insertions to and deletions from $S$. (But obviously we can't detect an attempt to delete a false positive.) Using the dynamic multiset representation of Theorem 1.1 we obtain Theorem 1.2.

If we double the size of the range of $h$, it suffices that $h$ be chosen from a 2-universal family. A function from the 2-universal family described in [15] can be stored using $O(\log n + \log w)$ bits, and chosen in expected time $(\log n + \log w)^{O(1)}$. This could be used to reduce the $O(w)$ additive term in the space bound to an optimal $O(\log w)$ term. However, the data structure would then have to use time $(\log n + \log w)^{O(1)}$ on preprocessing.

Another observation is that the reduction can be slightly improved by exploiting that $h(S)$ is expected to be somewhat smaller than $n$, in particular if $\epsilon$ is not too large. In other words: Use a bound tighter than the union bound to estimate the probability of a false positive. In particular, if we choose $h$ from a family of 4-wise independent functions, we may use a range smaller than $\lceil n/\epsilon \rceil$ while preserving the property that the probability of a false positive is at most $\epsilon$. However, even if $h$ is assumed to be a truly random function, this approach still gives a space usage that is $\Theta(n)$ bits from optimal.

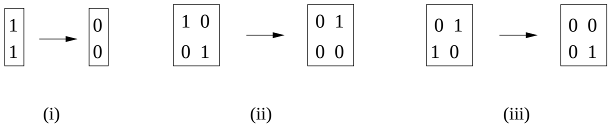
## 3 Counters in the bit probe model

In this section we discuss the problem of how to maintain a counter in almost optimal space while performing updates (incrementing and decrementing the counter) efficiently. The solution will be used in the next section to obtain a succinct dynamic multiset data structure.

More formally, we consider the following problem: Given an integer counter C, maintain it efficiently under the following operations:

- increment(): $C \leftarrow C + 1$,
- decrement(): if $C > 0$ then $C \leftarrow C - 1$, and
- iszero(): return 1 if $C = 0$ and 0 otherwise.

Many efficient solutions to this problem are known, e.g., Clancy and Knuth [7] show how to implement each operation in worst case constant time, using space $O(\log C)$ on a RAM. We consider the problem in a variant of the *bit probe* model, where time is counted as the number of bits accessed to perform an operation, and space as the number of bits used. The special feature of the bit probe model we consider is that it allows a special value $\perp$, in addition to the usual alphabet $\{0, 1\}$. The special value is the initial value of every bit, and we count only bits that are 0 or 1 when talking about space usage. We need counters in such a restricted model since we will be simulating accesses to single bits by lookups and updates in set data structures. In such a simulation, reading and updating a pointer would require too much time. In particular, this means that the solution in [7] is not efficient, as it uses pointers. Our solution uses standard ideas, and is included mainly for completeness.



(i) (ii) (iii)

Figure 1: Transformation rules. The top row contains one or two adjacent bits in $C^+$ and the bottom row contains the corresponding bit(s) in $C^-$.

If we simply maintain the value of $C$ in binary, then a sequence of alternate increment and decrement operations may require $\Omega(\log C)$ time per operation. Note that if we do not have decrements, then one can easily show that each increment takes $O(1)$ amortized time. So, we implement $C$ using two counters $C^+$ and $C^-$, maintaining the invariant $C = C^+ - C^-$. We denote the $i$th least significant bit of a counter $C$ by $C_i$. To perform an increment operation we increment the $C^+$ counter, and to perform a decrement operation we increment the $C^-$ counter. To reduce space usage, we maintain the invariant that all 0-bits having only 0-bits to the left (in more significant positions) are represented by $\perp$ in the data structure. This also gives a way of detecting when the most significant bit of $C^+$ or $C^-$ has been read.

To bound the values of $C^+$ and $C^-$, we also maintain the invariant that for any $i \geq 1$, out of the 4 bits $C^+_i$, $C^+_{i+1}$, $C^-_i$ and $C^-_{i+1}$ at most one is a 1. This invariant is maintained by using the transformation rules shown in Figure 1. Whenever we update a bit, we check to see if any of the transformation rules involving that bit can be applied, and if so we apply it. If we start from the point where $C^+ = C^- = 0$ and make the updates while maintaining the invariant after each update, it is possible to show that at most one transformation rule will be used after each update. However, this is not important for our argument.

The invariant ensures that $C^+ \geq 2C^-$ and hence $C^+ \leq 2C$. Thus the space required by this solution is at most $2\lceil \log C \rceil + 2$ bits. Additionally we can ensure that the space used by the two counters is no larger than $C$. This is done by minimizing the number of bits used if $C < 4$, and will be important in our application. Note that by transformation rule (i) there are only three possible assignments for the two bits, $C^+_i$ and $C^-_i$, at each position in the two counters. We use this fact later to represent the counter efficiently.

One can easily show that updates in this structure take $O(1)$ amortized time, by associating a potential of 1 to each bit that is 1: Each of the transformation rules strictly reduces the number of 1s, and incrementing one of the counters increases the number of 1s by at most one. An iszero query can be answered by simply testing whether $C_1^+$ is represented by $\perp$ in the data structure.

Lemma 3.1. *In the bit probe model with alphabet $\{0, 1, \perp\}$ one can maintain an integer counter $C$ that supports increment() and decrement() in $O(1)$ bit probes and bit updates, amortized, and iszero() in 1 bit probe, using space* $\min(2\lceil\log C\rceil + 2, C)$ *bits.*

## 4 From dynamic sets to dynamic multisets

In this section we consider the problem of storing a dynamic multiset succinctly. While the dynamic set problem has received considerable interest, the same is not true for the dynamic multiset problem, presumably due to the fact that most solutions to the dynamic set problem can easily be extended by associating a counter with each element. However, the space usage of the counters is $n \log n$ bits, which means that the resulting multiset representation cannot be succinct, i.e., cannot use close to the information theoretically optimum space. We note that this optimum is within $O(n)$ bits of the optimum for sets, i.e., $\log\binom{u+n}{n} \leq \log\binom{u}{n} + O(n)$. The only previous succinct representation of a multiset that allows constant time lookups is static [18].

A *static* multiset can be efficiently implemented using a number of set data structures, as follows. Let $U = [u]$ be the universe, let $M$ be the multiset of elements from $U$, and let $n = |M|$ be the size of the multiset. The multiset $M$ can be represented by $\lceil\log(n + 1)\rceil$ sets $S_1, \ldots, S_{\lceil\log n\rceil} \subseteq [2u]$. An element $x \in U$ has two corresponding elements in $[2u]$, $x_0$ and $x_1$. An element $x$ is represented by exactly one of $x_0$ and $x_1$ in the set $S_i$ if the number of occurrences, $c_x$, of $x$ in $M$ is at least $2^{i-1}$. It is represented in $S_i$ by the element $x_d$, where $d$ is the $i$th least significant bit in the binary representation of the number $c_x$. Implementing a multiset in this way, it is enough to perform two lookups in the set $S_1$ to decide if an element is in the multiset. The total number of elements to store is at most equal to the number of elements in the multiset. The time to get the number of occurrences of an element $x$ in the multiset depends on $c_x$. All sets in which $x$ is represented have to be searched to find out the exact number, so $O(1 + \log c_x)$ lookups are necessary.

We want a *dynamic* multiset for the Bloom filter implementation. There are two problems with the above multiset data structure if we want to make it dynamic. First, updates may take $\Omega(\log n)$ time. The second problem is memory management, i.e., how to allocate space for the data structures, which may grow and shrink independently of each other.

The solution to the first problem is to use redundant counters instead of binary counters. The redundant counters described in Section 3 require only amortized $O(1)$ bits updated when increasing or decreasing the counter. On the other hand we need to represent each element by one of three, rather than one of two, elements in each set, and two extra bits are needed for the counter.

Our data structure consists of $\lceil\log n\rceil + 2$ dynamic sets, $S_1, \ldots, S_{\lceil\log n\rceil+2}$, with elements from $[3u]$. An element $x \in U$ has three corresponding elements in $[3u]$, call them $x_0$, $x_1$, and $x_{-1}$. We include exactly one of the corresponding elements in $S_i$ if the redundant counter $C$ for $x$ does not have both $C_i^+$ and $C_i^-$ represented by $\perp$. Otherwise, depending on the values of $C_i^+$ and $C_i^-$ we include either $x_0$, $x_1$, or $x_{-1}$ in $S_i$. (There are only three possibilities because of transformation rule (i).)

Memory management according to Lemma 4.1 below is used to efficiently store the collection of sets. It is possible to maintain all sets under insertions and deletions without using too much extra time or space. The time for the update operations is $O(1)$, expected amortized.

Lemma 4.1. *A collection of dynamic sets $S_1, \ldots S_k \subseteq [u]$ can be maintained under insertions/deletions of elements to/from the individual sets, which take $O(1)$ amortized expected time, while supporting membership queries on any set in $O(1)$ time. If $B_i$ is the information-theoretic minimum space required to store the set $S_i$, for $1 \leq i \leq k$, then the total space occupied by this structure is $s + o(s) + O(\sqrt{sk \log u}) + O(k \log u)$ bits, where $s = \sum_{i=1}^{k} B_i$.*

**Proof sketch.** We maintain each set $S_i$ using the succinct dynamic dictionary structure of [19], which uses $B_i + o(B_i)$ bits and supports insertions and deletions on $S_i$ in expected amortized $O(1)$ time and membership queries in $O(1)$ time. Each of these dynamic dictionaries are stored in 'extendable arrays' (see [19] for details) with record size $w$. We then store these extendable arrays using the structure of Lemma 1 of [19]. The space bound follows, since the total 'nominal size' of all the extendable arrays is $s + o(s)$ bits. ■

We have now shown part of Theorem 1.1, namely that in any of the set data structures, we can do lookup in $O(1)$ time and updates in amortized expected $O(1)$ time, including memory management of the collection of sets. The $O(1 + \log c)$ time to report the number of occurrences follows this and from the above description.

What remains to show Theorem 1.1 is to calculate the total space usage for the multiset data structure. Denote by $n_1, n_2, \ldots$ the number of elements in the sets $S_1, S_2, \ldots$. By Lemma 3.1 the space of a counter will never exceed the value of the counter. Thus, an element occurring in the multiset $c$ times is stored in at most $c$ sets, so $\sum_i n_i \leq n$. Also, since the space of a counter of value $c$ is at most $2\lceil \log c \rceil + 2$, and since only counters with space at least $i$ give rise to elements in $S_i$, it follows that $n_i = O(n/2^{i/2})$. Theorem 1.1 follows by bounding the sum of information theoretical minimum space usage for the sets in the collection:

$$\begin{aligned}\sum_i B_i &\leq \sum_i n_i \log(3ue/n_i) \\ &\leq n\log(u/n) + \sum_i n_i \log(3en/n_i) \\ &= n\log(u/n) + O(\sum_i (n/2^i)\log(3en/(n/2^{i/2})) \\ &= n\log(u/n) + O(\sum_i ni/2^{i/2}) \\ &= n\log(u/n) + O(n)\ .\end{aligned}$$

## 5 A practical variant

The dynamic dictionary structure of Raman and Rao [19] is not efficient in terms of practical performance. To get a more practical variant in the case where we have only insertions and thus only need to store a set, we can replace this dynamic dictionary by a simpler dynamic hashing scheme by Cleary [8], based on linear probing. Using this scheme, a set of size $n$ from the universe $\{1, \ldots, u\}$ can be stored using $(1/\alpha)(n\log(4u/n))$ bits while supporting insertions, deletions and membership queries in expected $O(\frac{1}{(1-\alpha)^2})$ time, for $0 < \alpha < 1$. Note that if $\alpha = 1 - O(\frac{1}{\log(1/\epsilon)})$ then the space usage is $O(n)$ bits from optimal.

The time bound, but not the space bound, uses the assumption that there is free access to a hash function that is a uniformly random permutation of the universe. However, heuristically linear probing works very well even with restricted randomness. For example, if $u+1$ is prime one could use the permutation

$$x \mapsto ax \mod (u+1)$$

where $a$ is a random number in $\{1, \ldots, u\}$. (If $u+1$ is not prime, there exists a prime not much larger than $u+1$ which can be used instead, causing only a small degradation in space usage.)

The main insight behind Cleary's data structure is that, when using as a hash function the last bits of the random permutation on the universe, an uninterrupted sequence of $t$ occupied locations in the hash table can be represented using $t(\log(4u/n))$ bits, such that efficient decoding of the elements residing in these positions is possible. In our application, each cell of the linear probing hash table will be just $\log(1/\epsilon) + 2$ bits (assuming $\epsilon$ is a negative power of 2), and the expected number of sequential bits involved in a lookup or update is $O(\frac{\log(1/\epsilon)}{(1-\alpha)^2})$. For $\alpha = 1 - O(\frac{1}{\log(1/\epsilon)})$ this is $O(\log^3(1/\epsilon))$ bits, which is $O(1)$ machine words unless $\epsilon$ is quite small. Choosing constant $\alpha < 1$, the expected *worst case* number of bits accessed is $O(\log(n)\log(1/\epsilon))$, which is $O(\log(1/\epsilon))$ machine words. Hence, the worst case number of machine words accessed is the same as for Bloom filters. An important point here is that memory accesses are sequential, and hence cache performance will be much better than for Bloom filters.

## 6 On some Bloom filter generalizations

**6.1 Spectral Bloom filters.** Spectral Bloom filters [9] generalize Bloom filters to storing an approximate multiset. The membership query is generalized to a query on the multiplicity of an element. Now, the answer to any multiplicity query is never smaller than the true multiplicity, and greater only with probability $\epsilon$. The space usage is similar to that of a Bloom filter for a set of the same size (adding multiplicities). The construction generalizes Bloom filters, and thus the query time is $\Theta(\log(1/\epsilon))$. Using our data structure it is also possible to answer cardinality queries, and the answer is approximate in the same sense as for Spectral Bloom Filters. The time needed to determine a multiplicity of $k$ is $O(\log k)$. This result is not strictly comparable to that in [9]. Notice, however, that if the dynamic multiset representation of Theorem 1.1 could be replaced with a representation supporting constant time cardinality queries, we could also obtain constant time approximate cardinality queries.

**6.2 Bloomier filters.** Bloomier filters [6] generalize Bloom filters to associate with each element of $S$ some satellite information from $\{1, \ldots, m\}$. Elements not in $S$ are false positives with probability $\epsilon$, and will in that case have associated information equaling that of some element in $S$. The result in [6] is that one can get by with $O(n\log(1/\epsilon) + n\log m)$ bits of space, which is within a constant factor of optimal, in the case where $S$ is a static set. The lookup time is constant, assuming that $\log m = O(w)$. The analysis assumes truly random hash functions. It is shown that without free access to random hash functions, $\Omega(\log w)$ bits of space are needed.

There is a conceptually very simple way of improving the result to use explicit hash functions and have space that is $O(n + \log w)$ bits from optimal. The idea is to store a minimal perfect hash function for $S$, using $O(n + \log w)$ bits [11], and a function $h : \{0,1\}^w \to \{0,1\}^{\lceil \log(2/\epsilon) \rceil}$ from a 2-universal family, using $O(\log n + \log w)$ bits [15]. Then store an array of size $n$, where the entry that is the perfect hash value of $x \in S$ contains:

1. The value $h(x)$, and
2. The information associated with $x$.

Lookup of $x$ simply consists of computing a value of the perfect hash function and checking whether the stored hash function value is $h(x)$. It follows from the definition of 2-universal hashing that any element $y \notin S$ has probability at most $\epsilon$ of having the same hash function value $h(y)$ as the element in $S$ that maps to the same entry of the array.

An efficient data structure for the *dynamic* version of the Bloomier filter problem was recently given in [13].

**6.3 Lossy dictionaries.** Lossy dictionaries, considered in [17], are set representations with false positives *and false negatives.* It was shown in [17] that a lossy dictionary with $\gamma n$ false negatives requires space that is $1 - \gamma$ times that of a lossy dictionary without false negatives (up to an additive $O(n)$ term). Thus, a space optimal lossy dictionary can be obtained by omitting a fraction $\gamma$ of the keys in the set stored by our data structure. This improves upon the static lossy dictionary described in [17].

In the dynamic case we need to "remember" which keys were not stored, since these should never be deleted from the data structure. This can be done by using a strongly universal (i.e., pairwise independent) hash function $\rho : \{0,1\}^w \to \{1, \ldots, 2^w\}$ and omitting keys $x$ where $\rho(x) \leq 2^w\gamma$. The expected number of omitted keys is then exactly $\gamma n$ (assuming $2^w\gamma$ is an integer), and hence the expected space usage is optimal, up to lower order terms.

## 7 Conclusion

We have described a data structure dealing with some of the most important shortcomings of Bloom filters: Time dependence on $\epsilon$, suboptimal space usage, lack of explicit analyzable hash functions, and the inability to do deletions. Additionally, we described a variant that is likely to compete well with Bloom filters in practice. The main technical contribution of the paper is a succinct multiset representation that, when used with a reduction from [5], gives our main result.

An interesting open problem is whether it is possible to obtain the information theoretic lower bound for the approximate membership problem, in a way that facilitates efficient lookups. All present data structures use $\Omega(n)$ bits more than this. Finding an optimal space dynamic multiset representation that supports cardinality queries in constant time is another open problem.

## References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] A. Z. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing*, pages 636–646. ACM Press, 2002.

[3] A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.

[5] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, 1978.

[6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, pages 30–39. ACM Press, 2004.

[7] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report CS-TR-77-606, Stanford University, Department of Computer Science, 1977.

[8] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, C-33(9):828–834, Sept. 1984.

[9] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 241–252. ACM Press, 2003.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[11] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS '01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. Springer-Verlag, 2001.

[12] M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[13] C. W. Mortensen, R. Pagh, and M. Pătraşcu. On dynamic range reporting in one dimension. Manuscript, 2004.

[14] A. Östlin and R. Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC '03)*, pages 622–628. ACM Press, 2003.

[15] R. Pagh. Dispersing Hash Functions. In *Proceedings of the 4th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '00)*, volume 8 of *Proceedings in Informatics*, pages 53–67. Carleton Scientific, 2000.

[16] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[17] R. Pagh and F. F. Rodler. Lossy dictionaries. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 300–311. Springer-Verlag, 2001.

[18] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 233–242. ACM Press, 2002.

[19] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, 2003.