Succinct Data Structures for Retrieval and Approximate Membership*

(Extended Abstract)

Martin Dietzfelbinger[1] and Rasmus Pagh[2]

[1] Technische Universität Ilmenau, 98684 Ilmenau, Germany
martin.dietzfelbinger@tu-ilmenau.de
[2] IT University of Copenhagen, 2300 København S, Denmark
pagh@itu.dk

Abstract. The *retrieval problem* is the problem of associating data with keys in a set. Formally, the data structure must store a function $f\colon U \to \{0,1\}^r$ that has specified values on the elements of a given set $S \subseteq U$, $|S| = n$, but may have any value on elements outside S. All known methods (e. g. those based on perfect hash functions), induce a space overhead of $\Theta(n)$ bits over the optimum, regardless of the evaluation time. We show that for any k, query time $O(k)$ can be achieved using space that is within a factor $1 + e^{-k}$ of optimal, asymptotically for large n. The time to construct the data structure is $O(n)$, expected. If we allow logarithmic evaluation time, the additive overhead can be reduced to $O(\log\log n)$ bits whp. A general reduction transfers the results on retrieval into analogous results on *approximate membership*, a problem traditionally addressed using Bloom filters. Thus we obtain space bounds arbitrarily close to the lower bound for this problem as well. The evaluation procedures of our data structures are extremely simple. For the results stated above we assume free access to fully random hash functions. This assumption can be justified using space $o(n)$ to simulate full randomness on a RAM.

1 Introduction

Suppose we want to build a data structure that is able to distinguish between girls' and boys' names, in a collection of n names. Given a string not in the set of names, the data structure may return any answer. It is clear that in the worst case this data structure needs at least n bits, even if it is given access to the list of names. The previously best solution that does not require the set of names to be stored uses more than $1.22n$ bits. Surprisingly, as we will see in this paper, $n + o(n)$ bits is enough, still allowing fast queries. If "global" hash functions, shared among all data structures, are available the space usage drops all the way to $n + O(\log\log n)$ bits whp. This is a rare example of a data structure with

* The main ideas for this paper were conceived while the authors were participating in the 2006 Seminar on Data Structures at IBFI Schloss Dagstuhl, Germany.

non-trivial functionality and a space usage that essentially matches the entropy lower bound.

1.1 Problem definition and motivation

The *dictionary problem* consists of storing a set S of n keys, and r bits of data associated with each key. A *lookup* query for x reports whether or not $x \in S$, and in the positive case reports the data associated with x. We will denote the size of S by n, and assume that keys come from a set U of size $n^{O(1)}$. In this paper, we restrict ourselves to the *static* problem, where S and the associated data are fixed and do not change. We study two relaxations of the static dictionary problem that allow data structures using less space than a full-fledged dictionary:

• The *retrieval problem* differs from the dictionary problem in that the set S does not need to be stored. A retrieval query on $x \in S$ is required to report the data associated with x, while a retrieval query on $x \notin S$ may return any r-bit string.

• The *approximate membership problem* consists of storing a data structure that supports membership queries in the following manner: For a query on $x \in S$ it is reported that $x \in S$. For a query on $x \notin S$ it is reported with probability at least $1 - \varepsilon$ that $x \notin S$, and with probability at most ε that $x \in S$ (a "false positive"). For simplicity we will assume that ε is a negative power of 2.

Our model of computation is a unit cost RAM with a standard instruction set. For simplicity we assume that a key fits in a single machine word, and that associated values are no larger than keys. Some results will assume free access to fully random hash functions, such that any function value can be computed in constant time. (This is explicitly stated in such cases.)

The approximate membership problem has attracted significant interest in recent years due to a number of applications, mainly in distributed systems and database systems, where false positives can be tolerated and space usage is crucial (see [4] for a survey). Often the false positive probability that can be tolerated is relatively large, say, in the range $1\% - 10\%$, which entails that the space usage can be made much smaller than what would be required to store S exactly.

The retrieval problem shows up in situations where the amount of data associated with each key is small, and it is either known that queries will only be asked on keys in S, or where the answers returned for keys not in S do not matter. As an example, suppose that we have ranked the URLs of the World Wide Web on a 2^r step scale, where r is a small integer. Then a retrieval data structure would be able to provide the ranking of a given URL, without having to store the URL itself. The retrieval problem is also the key to obtaining a space-optimal RAM data structure that answers range queries in constant time [1, 18].

1.2 Previous results

Approximate membership. The study of approximate membership was initiated by Bloom [2] who described the *Bloom filter* data structure which provides an elegant, near-optimal solution to the problem. Bloom showed [2,4] that a space usage of $n \log_2(1/\varepsilon) \log_2 e$ bits suffices for a false positive probability of ε. Carter *et al.* [7] showed that $n \log_2(1/\varepsilon)$ bits are required for solving the approximate membership problem when $|U| \gg n$ (see also [13] for details). Thus Bloom filters have space usage within a factor $\log_2 e \approx 1.44$ of the lower bound, which is tight.

Another approach to approximate membership is *perfect hashing.* A function $h \colon U \to [n]$ is a minimal perfect hash function for S if it maps the keys of $S \subseteq U$ bijectively to $[n] = \{0, \ldots, n-1\}$, where $n = |S|$. Hagerup and Tholey [16] showed how to store a minimal perfect hash function h in a data structure of $n \log_2 e + o(n)$ bits such that h can be evaluated on a given input in constant time. This space usage is optimal. Now store an array of n entries where, for each $x \in S$, entry $h(x)$ contains a $\log_2(1/\varepsilon)$-bit hash signature $q(x)$. When looking up a key x, we answer "$x \in S$" if and only if the hash signature at entry $h(x)$ is equal to $q(x)$. The origin of this idea is unknown to us, but it is described e.g. in [4]. The space usage for the resulting data structure differs from the lower bound $n \log_2(1/\varepsilon)$ by the space required for the minimum perfect hash function, and improves upon Bloom filters when $\varepsilon \leq 2^{-4}$ and n is sufficiently large.

Mitzenmacher [19] considered the *encoding* problem where the task is to represent and transmit an approximate set representation (no fast queries required). However, even in this case existing techniques have a space overhead similar to that of the perfect hashing approach.

Retrieval. The retrieval problem has traditionally been addressed through the use of perfect hashing. Using the Hagerup-Tholey data structure yields a space usage of $nr + n \log_2 e + o(n)$ bits with constant query time. Recently, Chazelle *et al.* [8] presented a different approach to the problem. Each key is associated with $k = O(1)$ locations in an array with $O(n)$ entries of r bits. The answer to a retrieval query on x is found by combining the values of entries associated with x, using bit-wise XOR. In place of the XOR operation, any abelian group operation may be used. In fact, this idea was used earlier by Majewski, Wormald, Havas, and Czech [17] and by Seiden and Hirschberg [23] to address the special case of order-preserving minimal perfect hashing. It is not hard to see that these data structure in fact solve the retrieval problem. The main result of [17] is that for $k = 3$ a space usage of around $1.23nr$ bits is possible, and this is the best possible using the construction algorithm of [8,17] (other values of k give worse results). Though this space usage is larger than when using perfect hashing, asymptotically for large n, the simplicity and the lack of lower order terms in the space usage that may dominate for small n makes it interesting from a practical viewpoint. A particular feature is that (like for Bloom filters) all memory lookups are nonadaptive, i.e., the memory addresses can be determined from the query only. This can be exploited by modern CPU architectures that are able to parallelize memory lookups (see e.g. [24]). In fact, Chazelle *et al.* also show how approximate membership can be incorporated into their data structure by extending array entries to $r + \log_2(1/\varepsilon)$ bits. This generalized data structure is called a *Bloomier filter.* Again, the space usage is a constant factor higher, asymptotically, than the solution based on perfect hashing.

Approximate membership by retrieval. We observe that there exists a simple reduction from approximate membership problem to the retrieval problem. Though it is used in the approximate membership data structure based on perfect hashing, we do not believe that it has been stated in this generality before (for a proof, see the full version of this paper [13]).

Observation 1 *Assuming free access to fully random hash functions, any static retrieval data structure can be used to implement an approximate membership data structure having false positive probability 2^{-r}, with no additional cost in space, and $O(1)$ extra time.*

If we drop the assumption of fully random hash functions being provided for free, only a $o(1)$ term has to be added to the false positive probability (for details see [13]).

Parallel work. Immediately after a draft full version of this work appeared ([13], March 26, 2008), we were informed that E. Porat had independently worked on the same problems. His results are described in a report ([22], April 11, 2008). He also uses linear equations, however without restricting the weight of rows. The resulting problems with construction and evaluation time are cicumvented by using a two-level splitting technique similar to one used in [16]. The space usage is asymptotically smaller than in our Theorem 1(a).

1.3 New contributions

Our main contribution shows that the approach of Chazelle *et al.* [8], Majewski *et al.* [17], and Seiden and Hirschberg [23] can be used to achieve space for retrieval that is very close to the lower bound, while retaining efficient evaluation.

Theorem 1. *There exist data structures for the retrieval problem having the following space and time complexity on a unit cost RAM with free access to a fully random hash function ($c > 0$ is any constant)*: (a) *For any fixed $\gamma > 0$, for any sufficiently large n and every r with $1 \leq r \leq c \log n$: space $(1+\gamma)nr$ bits, constant query time $O(1+\log(\frac{1}{\gamma}))$, and expected construction time $O(n)$*; (b) *For any sufficiently large n and every r with $1 \leq r \leq c \log n$: space $nr + O(\log \log n)$ bits whp.*[3]*, query time $O(\log n)$, and expected construction time $O(n^3)$.*

The basic data structure and query evaluation algorithm is the same as in [8]. The new contribution is to analyze a different construction algorithm (suggested in [23]) that is able to achieve a space usage arbitrarily close to the optimum. Our analysis needs tools and theorems from linear algebra, while that of [8] was based on elementary combinatorics ([23] provided only experimental results). To get a data structure that allows expected linear construction time we devise a new variant of the data structure and query evaluation algorithm, retaining

[3] "whp." means with probability $1 - O(\frac{1}{\mathrm{poly}(n)})$.

simplicity and non-adaptivity. (We note that the data structure of [22] has an adaptive evaluation procedure, using many auxiliary tables.)

The papers on Bloom filters, and the work of Chazelle *et al.* [8] all make the assumption of access to fully random hash functions. We state that our data structures can be realized on a RAM, with a small additional cost in space (proof in [13]).

Theorem 2. *In the setting of Theorem 1, for some $\varepsilon > 0$, we can avoid the assumption of fully random hash functions and get data structures with the following space and time complexities:*(a) *Space $(1+\gamma)nr$ bits, query time $O(1+\log(\frac{1}{\gamma}))$, expected construction time $O(n)$, for any constant $\gamma > 0$;* (b) *Space $nr + O(n^{1-\varepsilon})$ bits, query time $O(\log n)$, expected construction time $O(n^{1+\delta})$, for an arbitrary constant $\delta > 0$.*

1.4 Overview of paper

Section 2 describes our basic retrieval data structure and its analysis. This leads to part (a) of Theorem 1, except that the construction time is $O(n^3)$. For lack of space, the approach to the proof of part (b) is only sketched briefly. Section 3 completes the proof of part (a) of Theorem 1 by showing how the construction algorithm can be made to run in linear time. Section 4 describes a close relationship between the space requirements for dictionary implementations based on the balanced allocation paradigm (like k-ary cuckoo hashing [15]) and the space requirements for retrieval structures.

2 Retrieval in constant time and almost optimal space

In this section, we give the basic construction of a data structure for retrieval with constant time lookup operation and $(1+\delta)nr$ space. As a technical basis, we first describe results by Calkin [6].

2.1 Calkin's results

All calculations are over the field $\mathrm{GF}(2) = \mathbb{Z}_2$ with 2 elements. We consider binary matrices $M = (p_{ij})_{1\le i\le n, 0\le j<m}$ with n rows and m columns. If M is such a matrix, then row vector $(p_{i0}, \ldots, p_{i,m-1})$ is called p_i, for $1 \le i \le n$.

Theorem 3 (Calkin [6, Theorem 1.2]). *For every $k > 2$ there is a constant $\beta_k < 1$ such that the following holds: Assume the n rows $p_1, \ldots, p_n$ of a matrix M are chosen at random from the set of binary vectors of length m and weight (number of 1s) exactly k. Then:*
(a) *If $n/m \le \beta < \beta_k$, then $\Pr(M$ has full row rank$) \to 1$ (as $n \to \infty$).*
(b) *If $n/m \ge \beta > \beta_k$, then $\Pr(M$ has full row rank$) \to 0$ (as $n \to \infty$).*
Furthermore, $\beta_k - (1 - (e^{-k}/(\ln 2)) \to 0$ for $k \to \infty$ (exponentially fast in k).

k	3	4	5	6
β_k	0.88949	0.96714	0.98916	0.99622
β_k^{appr}	0.9091	0.9690	0.9893	0.99624
β_k^{-1}	1.1243	1.034	1.011	1.0038

Table 1. Approximate threshold values from Theorem 3, using (1) and (2).

Remark 1. (a) The case $k = 2$ is omitted in this discussion. The threshold value for this case is $\beta_2 = 2$, as is well known from the theory of random graphs. In [17] and [3] this fact is used for constructing perfect hash functions, in a way that implicitly includes the construction of retrieval structures.
(b) A closer look into the proof of Theorem 1.2 in [6] reveals that for each k there is some $\varepsilon = \varepsilon_k > 0$ such that in the situation of Theorem 3(a) we have $\Pr(M \text{ has full row rank}) = 1 - O(n^{-\varepsilon})$. The following values are suitable: $\varepsilon_3 = \frac{2}{7}$, $\varepsilon_4 = \frac{5}{7}$, $\varepsilon_k = 1$ for $k \geq 5$.
(c) According to [6], the threshold value β_k is characterized as follows: Define

$$f(\alpha, \beta) = -\ln 2 - \alpha \ln \alpha - (1 - \alpha) \ln(1 - \alpha) + \beta \ln(1 + (1 - 2\alpha)^k), \qquad (1)$$

for $0 < \alpha < 1$. Let β_k be the minimal β so that $f(\alpha, \beta)$ attains the value 0 for some $\alpha \in (0, \frac{1}{2})$. Using a computer algebra system, it is easy to find approximate values for β_k and β_k^{-1} for small k, see Table 1. Calkin further proves that

$$\beta_k = 1 - \frac{e^{-k}}{\ln 2} - \frac{1}{2 \ln 2}\Big(k^2 - 2k + \frac{2k}{\ln 2} - 1\Big) \cdot e^{-2k} \pm O(k^4) \cdot e^{-3k}, \qquad (2)$$

as $k \to \infty$. It seems that the approximation obtained by omitting the last term in (2) is quite good already for small values of k. (See the row for β_k^{appr} in Table 1.)

2.2 The basic retrieval data structure

Now we are ready to describe a retrieval data structure. Assume $f \colon S \to \{0, 1\}^r$ is given, for a set $S = \{x_1, \ldots, x_n\}$. For a given (fixed) $k \geq 3$ let $1 + \delta > \beta_k^{-1}$ be arbitrary and let $m = (1 + \delta)n$. We can arrange the lookup time to be $O(k)$ and the number of bits in the data structure to be $mr = (1 + \delta)nr$ plus lower order terms.

We assume that we have access to a mapping $A \colon U \to \binom{[m]}{k}$, $x \mapsto A_x$, where $\binom{X}{k} = \{Y \subseteq X \mid |X| = k\}$, so that A is fully random on S. We write $A_x = \{h_1(x), \ldots, h_k(x)\}$ (the order is irrelevant). It must be possible to repeatedly choose a new function A if the need arises. We need to store an index to identify the function A that was actually used. It is not hard to see that using standard hash functions with ranges $[m], [m-1], \ldots, [m-k+1]$, such random sets with exactly k elements can be constructed in time $O(k)$. (For details see [13].)

The construction starts from $= \{x_1, \ldots, x_n\}$ and the bit strings $u_i = f(x_i) \in \{0,1\}^r$, $1 \le i \le n$. We consider the matrix

$$M = (p_{ij})_{1 \le i \le n, 0 \le j < m}, \text{ with } p_{ij} = 1 \text{ if } j \in A_{x_i} \text{ and } p_{ij} = 0 \text{ otherwise.} \tag{3}$$

Theorem 3(a) (with Remark 1(b)) says that M has full row rank with probability $1 - O(n^{-\varepsilon})$ for some $\varepsilon > 0$. Assume n is so large that this happens with probability at least $\frac{3}{4}$. If M does have full row rank, the column space of M is all of $\{0,1\}^n$, hence for all $u \in \{0,1\}^n$ there is some $a \in \{0,1\}^m$ with $M \cdot a = u$. More generally, we arrange the bit strings $u_1, \ldots, u_n \in \{0,1\}^r$ as a column vector $u = (u_1, \ldots, u_n)^\mathsf{T}$. We stretch notation a bit (but in a natural way) so that we can multiply binary matrices with vectors of r-bit strings: multiplication is just bit/vector multiplication and addition is bitwise XOR. It is then easy to see, working with the components of the u_i separately, that there is a (column) vector $a = (a_0, \ldots, a_{m-1})^\mathsf{T}$ with entries in $\{0,1\}^r$ such that $M \cdot a = u$.

We can rephrase this as follows (using $\oplus$ as notation for bitwise XOR): For $a \in (\{0,1\}^r)^m$ and $x \in U$ define

$$h_a(x) = \bigoplus_{j \in A_x} a_j. \tag{4}$$

Then for an arbitrary sequence $(u_1, \ldots, u_n)$ of prescribed values from $\{0,1\}^r$ there is some $a \in (\{0,1\}^r)^m$ with $h_a(x_i) = u_i$, for $1 \le i \le n$. Such a vector $a \in (\{0,1\}^r)^m$, together with an identifier for the function A used in the successful construction, is a data structure for retrieving the value $u_i = f(x_i)$, given x_i. There are k accesses to the data structure, plus the effort to calculate the set A_x from x.

Remark 2. A similar construction (over arbitrary fields GF(q)) was described by Seiden and Hirschberg [23]. However, those authors did not have Calkin's results, and so could not give theoretical bounds on the number m of columns needed. Also, our construction generalizes the approach of [17] and [8], where it was required that M could be transformed into echelon form by permuting rows and columns, which is sufficient, but not necessary, for M to have full row rank. Using these constructions it is not possible to work with $m \le 1.22n$ [17].

Some details of the construction are missing. We describe one of several possible ways to proceed. — From S, we first calculate the sets A_{x_i}, $1 \le i \le n$, in time $O(n)$. Using Gaussian elimination, we can check whether the induced matrix $M = (p_{ij})$ has full row rank. If this is not the case, we start all over with a new mapping $A\colon x \mapsto A_x$. This is repeated until a suitable matrix M is obtained. The expected number of repetitions is $1 + O(n^{-\varepsilon})$. For a matrix M with independent rows Gaussian elimination will also yield a "pseudoinverse" of M, that is, an invertible $n \times n$-matrix C (coding a sequence of elementary row transformations without row exchanges) with the property that in $C \cdot M$ the n unit vectors $e_i^\mathsf{T} = (0, \ldots, 0, 1, 0, \ldots, 0)^\mathsf{T}$ occur as columns:

$$\forall i,\ 1 \le i \le n,\ \exists\ b_i \in [m]\text{: column } b_i \text{ of } C \cdot M \text{ equals } e_i^\mathsf{T}. \tag{5}$$

Given $u = (u_1, \ldots, u_n) \in \{0,1\}^n$ we wish to find $a \in \{0,1\}^m$ such that

$$(C \cdot M) \cdot a = C \cdot u = u' = (u'_1, \ldots, u'_n)^{\mathsf{T}}. \tag{6}$$

Since $C \cdot M$ has the unit vectors in columns $b_1, \ldots, b_n$, we can easily read off a special a that solves (6): Let $a_j = 0$ for $j \notin \{b_1, \ldots, b_n\}$, and let $a_{b_i} = u'_i$ for $1 \le i \le n$. Exactly the same formula works if u, u', and a are vectors of r-bit strings. — We have established the following.

Theorem 4. *Assume that a mapping* $A\colon U \to \binom{[m]}{k}$ *is available that is fully random on* S (*with the option to choose such functions repeatedly and independently*)*. Let* $k > 2$ *be fixed, let* $1 + \delta > \beta_k^{-1}$*, and assume* n *is sufficiently large. Then given* $S = \{x_1, \ldots, x_n\}$ *and a sequence* $(u_1, \ldots, u_n)$ *of prescribed elements in* $\{0,1\}^r$*, we can find a vector* $a = (a_0, \ldots, a_{m-1})$ *with elements in* $\{0,1\}^r$ *such that* $h_a(x_i) = u_i$*, for* $1 \le i \le n$*. The expected construction time is* $O(n^3)$*, and the scratch space needed is* $O(n^2)$*.*

Remark 3. At the first glance, the time complexity of the construction seems to be forbiddingly large. However, using a trick ("split-and-share" described in [11] and in [13]) makes it possible to obtain a data structure with the same functionality and space bounds (up to a $o(n)$ term) in time $O(n^{1+\delta})$ for any given $\delta > 0$. In Section 3 we show how to construct a retrieval structure with essentially the same space requirements in expected linear time.

We briefly give some ideas how Theorem 1(b) can be proved. The basic approach is similar to the above, but we use $k(x)$ hash functions for key x, where $k(x)$, $x \in S$, are independent random variables, each approximately binomially distributed with expectation $\Theta(\log n)$, and a range size $m = n$. Theorem 2(a) in [9] entails that the resulting square matrix will be regular with probability > 0.28. It takes $O(n^3)$ time to test one matrix; trying $O(\log n)$ sets of hash functions will be sufficient whp. to find a set of hash functions that induces a matrix with full rank. Storing the index of this set of functions takes an extra $O(\log \log n)$ bits. The rest of the construction is similar as above. A lookup then requires evaluating $O(\log n)$ hash functions. A splitting trick can be used to reduce the construction time to $O(n^{1+\delta})$, without changing the functionality. (Details in [13].)

3 Retrieval in almost optimal space, with linear construction time

In this section we show how, using a variant of the retrieval data structure described in Section 2.2, we can achieve linear expected construction time and still get arbitrarily close to optimal space. This will prove Theorem 1(a). The reader should be aware that the results in this section hold asymptotically, only for rather large n.

Using the notation of Sections 2.1 and 2.2, we fix some k and some $\delta > 0$ such that $(1 + \delta)\beta_k > 1$. Further, some constant $\varepsilon > 0$ is fixed. We assume that the required fully random hash functions and mappings from keys to sets are at our disposal, and in case the construction fails we can choose a new set of such functions, even repeatedly. In [13] it is explained how this can be justified. Define $b = \frac{1}{2}\sqrt{\log n}$. We assume that ε and δ are so small that $(1+\varepsilon)^2(1+\delta) < 4$, and hence that $b \cdot 2^{(1+\varepsilon)^2(1+\delta)b^2} = o(n/(\log n)^3)$.

Assume $f\colon S \to \{0,1\}^r$ is given, the value $f(x)$ being denoted by u_x. The global setup is as follows: We use one fully random hash function φ to map S into the range $[m_0]$ with $m_0 = n/b$. In this way, m_0 blocks $B_i = \{x \in S \mid \varphi(x) = i\}$, $0 \le i < m_0$, are created, each with expected size b. The construction has two parts: a primary structure and a secondary structure for the "overflow keys" that cannot be accommodated in the primary structure. This is similar to the global structure of a number of well-known dictionary implementations. For the primary structure, we try to apply the construction from Section 2.2 to each of the blocks separately, but only once, with a fixed set of k hash functions. This construction may fail for one of two reasons: (i) the block may be too large — we do not allow more than $b' = (1+\varepsilon)b$ keys in a block if it is to be treated in the primary structure, or (ii) the construction from Section 2.2 fails because the row vectors in the matrix M_i induced by the sets A_x, $x \in B_i$, are not linearly independent.

For the primary structure, we set up a table T with $(1+\delta)(1+\varepsilon)n$ entries, partitioned into m_0 segments of size $(1+\delta)(1+\varepsilon)b = (1+\delta)b'$. Segment number i is associated with block B_i. If the construction from Section 2.2 fails, we set all the bits in segment number i to 0 and use the secondary structure to associate keys in B_i with the correct values. As secondary structure we choose a retrieval structure as in [8, 17], built on the basis of a second set of three hash functions, which are used to associate sets $A'_x \subseteq [1.3n']$ with the keys $x \in S'$, and a table $T'[0..1.3n'-1]$. This uses space $1.3n'r$ bits, where n' is the size of the set S' of keys for which the construction failed (the "*overflow keys*"). Of course, the secondary structure associates a value $f'(x)$ with *any* key $x \in S$. Rather than storing information about which blocks succeed we compensate for the contribution from $f'(x)$ as follows: If the construction succeeds for B_i, we store $(1+\delta)b'$ vectors of length r in segment number i of table T so that $x \in B_i$ is associated with the value $f(x) \oplus f'(x)$. On a query for $x \in U$, calculate $i = \varphi(x)$, then the offset $d_i = (i-1)(1+\delta)b'$ of the segment for block B_i in T', and return

$$\bigoplus_{j \in A_x} T[j + d_i] \;\oplus\; \bigoplus_{j \in A'_x} T'[j].$$

It is clear that for $x \in S$ the result will be $f(x)$: For $x \in S'$ the two terms are $\mathbf{0}$ and $f(x)$, and for $x \notin S'$ the two terms are $f'(x)$ and $f(x) \oplus f'(x)$. Note that the accesses to the tables are nonadaptive: all $k+3$ lookups may be carried out in parallel. In fact, if T and T' are concatenated this can be seen as the same evaluation procedure as in our basic algorithm (4), the difference being that the hash functions were chosen in a different way (e.g., do not all have the same range).

Lemma 1. *The expected number of overflow keys is $o(n)$.*

The proof is a standard application of Chernoff bounds — we refer to [13] for details. The overall space is $(1+\delta)(1+\varepsilon)n(r+1/b)+c|S'|r$ bits (apart from lower order terms). If $\gamma > 0$ is given, we may choose ε and δ (and k) so that this bound is smaller than $(1+\gamma)nr$ for n large enough.

Lemma 2. *The primary structure can be constructed in time $O(n)$.*

Proof: It is clear that linear time is sufficient to find the blocks B_i and identify the blocks that are too large. Now consider a fixed block B_i of size at most $(1+\varepsilon)b$. We must evaluate $|B_i| \cdot k$ hash functions to find the sets A_x, $x \in B_i$, and can piece together the matrix M_i that is induced by these sets in time $O(b)$ (assuming one can establish a word of $O(b)$ 0s in constant time and set a bit in such a word given by its position in constant time). The whole matrix has fewer than $\log n$ bits and fits into a single word. This makes it possible to use precomputed tables to speed up the computations we need. (The number of relevant matrixes is $o(\frac{n}{(\log n)^3})$ so that it is possible to calculate and store pseudoinverses and some matrix-vector products that we need in time and space $o(n)$. The details can be found in the full paper [13].)

Now assume a bit vector $u = (u_1, \ldots, u_{|B_i|})^{\mathsf{T}} \in \{0,1\}^{|B_i|}$, is given. Using C_i and a lookup table we can find $C_i \cdot u$ in constant time. A bit vector $a = (a_j)_{1 \le j \le (1+\delta)b'}$ that solves $M_i \cdot a = u$ can then be found in time $O(b)$. This leads to an overall construction time of $O(n)$ for the whole primary structure.

If the values in the range are bit vectors $f(x) = u_x \in \{0,1\}^r$, $x \in B_i$, a construction in time $O(nr)$ follows trivially. We may improve this time bound by arranging lookup tables that make it possible to multiply C_i even with vectors $U = (u_1, \ldots, u_{|B_i|})$ of bit vectors of length up to $O(\log n)$ in constant time. □

Note that the lookup tables are needed only by the construction algorithm, and not as part of the resulting data structure.

4 Retrieval and dictionaries by balanced allocation

In several recent papers, the following scenario for (statically) storing a set $S \subseteq U$ of keys was studied. A set $S = \{x_1, \ldots, x_n\} \subseteq U$ is to be stored in a table $\mathtt{T}[0..m-1]$ of size $m = (1+\delta)n$ as follows: To each key x we associate a set $A_x \subseteq [m]$ of k possible table positions. Assume there is a mapping $\sigma\colon \{1, \ldots, n\} \to [m]$ that is one-to-one and satisfies $\sigma(i) \in A_{x_i}$, for $1 \le i \le n$. (In this case we say $(A_x, x \in S)$ is *suitable* for S.) Choose one such mapping and store x_i in $\mathtt{T}[\sigma(i)]$. Examples of constructions that follow this scheme are cuckoo hashing [20], k-ary cuckoo hashing [15], blocked cuckoo hashing [12, 21], and perfectly balanced allocation [10]. In [5, 14] threshold densities for blocked cuckoo hashing were determined exactly. These schemes are the most space-efficient dictionary structures known, among schemes that store the keys explicitly in a hash table. For example, k-ary cuckoo hashing [15] works in space $m = (1+\varepsilon_k)n$ with $\varepsilon_k = e^{-\Theta(k)}$. Perfectly balanced allocation [10] works in optimal space $m = n$ with A_x consisting of 2 contiguous segments of $[n]$ of length $O(\log n)$ each.

Here, we point out a close relationship between dictionary structures of this kind and retrieval structures for functions $f\colon S \to R$, whenever the range R is not too small. We will assume that $R = \mathbb{F}$ for a finite field $\mathbb{F}$ with $|\mathbb{F}| \geq n$. (Using a simple splitting trick, this condition can be attenuated to $|\mathbb{F}| \geq n^{\delta}$.) From Section 2.2 we recall equation (3) where the matrix $M = (p_{ij})_{1 \leq i \leq n,\, 0 \leq j < m}$ was defined from the sets A_x, $x \in S$.

Observation. (For arbitrary fields $\mathbb{F}$.) If the 1s in M can be replaced by elements of $\mathbb{F}$ in such a way that the resulting matrix $M' = (p'_{ij})$ has full row rank over $\mathbb{F}$, then $(A_x, x \in S)$ is suitable for S. (*Proof*: If M' has full row rank, it has an $n \times n$ submatrix N with nonzero determinant. By the definition of the determinant there must be a mapping $\sigma\colon \{1, \ldots, n\} \to [m]$ with $\prod p'_{i\sigma(i)} \neq 0$, hence $p_{i\sigma(i)} = 1$ for $1 \leq i \leq n$.)

The observation implies that Calkin's bounds give upper space bounds for dictionary constructions like k-ary cuckoo hashing, which match values observed in experiments in [15]. Surprisingly, for fields that are not too small, the observation also works the other way around: existence of a dictionary implies existence of a retrieval structure.

Theorem 5. *Assume a mapping $x \mapsto A_x$ is given that is suitable for S. Then the following holds: If $g_1, \ldots, g_k\colon S \to \mathbb{F}$ are random, then with probability $\geq 1 - \frac{n}{|\mathbb{F}|}$ $M' = (p'_{ij})_{1 \leq i \leq n 0 \leq j < m}$, where $p'_{ij} = g_\ell(x_i)$ if $j = h_\ell(x_i)$ and $p'_{ij} = 0$ otherwise, has full row rank over $\mathbb{F}$.*

The proof uses the Schwartz-Zippel Theorem; it is given in the full paper [13].

The theorem implies the following: If the mapping $x \mapsto A_x$ is suitable for S, if $|\mathbb{F}| \geq 2n$, and if we have hash functions $g_1, \ldots, g_k\colon U \to \mathbb{F}$ that are random on S, then with probability at least $\frac{1}{2}$ we can build a retrieval structure for a function $f\colon S \to \mathbb{F}$ consisting of a table $T[0..m-1]$ with entries from $\mathbb{F}$ with $f(x) = \sum_{1 \leq \ell \leq k} g_\ell(x) \cdot T[h_\ell(x)]$. If we can switch to new functions $g_1, \ldots, g_k$ if necessary, this construction succeeds in $O(1)$ iterations in the expected case and in $O(\log n)$ iterations whp.

For example, from the dictionary constructions in [12] or [10], resp., we obtain retrieval structures with a table of size $\leq (1 + e^{-k})nr$ and lookup time $O(k)$, or optimal size n and lookup time $O(\log n)$, resp. In both cases for one retrieval operation we need to access only two contiguous segments of the table T, which makes these implementations very cache-friendly.

Acknowledgement. The authors thank Philipp Woelfel for several motivating discussions on the subject.

References

1. S. Alstrup, G. S. Brodal, and T. Rauhe, Optimal static range reporting in one dimension, Proc. 33rd ACM STOC, 2001, pp. 476–482.
2. B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, *Commun. ACM* **13**(7) 1970: 422–426.

3. F. C. Botelho, R. Pagh, and N. Ziviani, Simple and space-efficient minimal perfect hash functions, in: Proc. 10th WADS 2007, Springer LNCS 4619, pp. 139–150.
4. A. Z. Broder and M. Mitzenmacher, Network applications of Bloom filters: a survey, in: Proc. 40th Annual Allerton Conference on Communication, Control, and Computing, pp. 636–646, ACM Press, 2002.
5. J. A. Cain, P. Sanders, N. C. Wormald, The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation, Proc. 18th ACM-SIAM SODA, 2007, pp. 469–476.
6. N. J. Calkin, Dependent sets of constant weight binary vectors, *Combinatorics, Probability and Computing* **6**(3) 1997: 263–271.
7. L. Carter, R. W. Floyd, J. Gill, G. Markowsky, and M. N. Wegman, Exact and approximate membership testers, Proc. 10th ACM STOC, 1978, pp. 59–65.
8. B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, The Bloomier filter: an efficient data structure for static support lookup tables, Proc. 15th ACM-SIAM SODA, 2004, pp. 30–39.
9. C. Cooper, On the rank of random matrices, *Random Struct. Algorithms* **16**(2) 2001: 209–232.
10. A. Czumaj, C. Riley, C. Scheideler, Perfectly Balanced Allocation, in: Proc. RANDOM-APPROX 2003, Springer LNCS 2764, pp. 240–251.
11. M. Dietzfelbinger, Design strategies for minimal perfect hash functions, in: Proc. 4th Int. Symp. on Stochastic Algorithms: Foundations and Applications (SAGA), 2007, Springer LNCS 4665, pp. 2–17.
12. M. Dietzfelbinger and C. Weidling, Balanced allocation and dictionaries with tightly packed constant size bins, *Theoret. Comput. Sci.* **380**(1–2) 2007: 47–68.
13. M. Dietzfelbinger and R. Pagh, Succinct Data Structures for Retrieval and Approximate Membership, Technical Report, arXiv:0803.3693v1 [cs.DS], March 26, 2008.
14. D. Fernholz and V. Ramachandran, The k-orientability thresholds for $G_{n,p}$, Proc. 18th ACM-SIAM SODA, 2007, pp. 459–468.
15. D. Fotakis, R. Pagh, P. Sanders, P. G. Spirakis, Space efficient hash tables with worst case constant access time, *Theory. Comput. Syst.* **38**(2) 2005: 229–248.
16. T. Hagerup, T. Tholey, Efficient minimal perfect hashing in nearly minimal space, in: Proc. 18th STACS 2001, Springer LNCS 2010, pp. 317–326.
17. B. S. Majewski, N. C. Wormald, G. Havas, Z. J. Czech, A family of perfect hashing methods, *Computer J.* **39**(6) 1996: 547–554
18. C. W. Mortensen and R. Pagh, and M. Pătraşcu, On dynamic range reporting in one dimension, Proc. 37th ACM STOC, 2005, pp. 104–111.
19. M. Mitzenmacher, Compressed Bloom filters, *IEEE/ACM Transactions on Networking*, **10**(5):604–612 (2002).
20. R. Pagh and F. F. Rodler, Cuckoo Hashing, *J. Algorithms* **51**:122–144 (2004).
21. R. Panigrahy, Efficient hashing with lookups in two memory accesses, Proc. 16th ACM-SIAM SODA, 2005, pp. 830–839.
22. Ely Porat, An Optimal Bloom Filter Replacement Based on Matrix Solving, Technical Report, arXiv:0804.1845v1 [cs.DS], April 11, 2008.
23. S. S. Seiden and D. S. Hirschberg, Finding succinct ordered minimal perfect hash functions, *Inf. Process. Lett.* **51**(6) 1994: 283–288.
24. M. Zukowski and S. Heman and P. A. Boncz, Architecture-conscious hashing, in: Proc. Int. Workshop on Data Management on New Hardware (DaMoN), Chicago, 2006, Article No. 6 (8 pages).