

# Deterministic Dictionaries\*

Torben Hagerup

*Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt  
D-60054 Frankfurt am Main, Germany  
E-mail: hagerup@informatik.uni-frankfurt.de*

and

Peter Bro Miltersen

*BRICS†, Department of Computer Science, Aarhus University  
Ny Munkegade Bldg. 540, DK-8000 Århus C, Denmark  
E-mail: bromille@daimi.au.dk*

and

Rasmus Pagh

*BRICS\*, Department of Computer Science, Aarhus University  
Ny Munkegade Bldg. 540, DK-8000 Århus C, Denmark  
E-mail: pagh@daimi.au.dk*

It is shown that a static dictionary that offers constant-time access to  $n$  elements with  $w$ -bit keys and occupies  $O(n)$  words of memory can be constructed deterministically in  $O(n \log n)$  time on a unit-cost RAM with word length  $w$  and a standard instruction set including multiplication. Whereas a randomized construction working in linear expected time was known, the running time of the best previous deterministic algorithm was  $\Omega(n^2)$ . Using a standard dynamization technique, the first deterministic dynamic dictionary with constant lookup time and sublinear update time is derived. The new algorithms are weakly nonuniform; i.e., they require access to a fixed number of precomputed constants dependent on  $w$ . The main technical tools employed are unit-cost error-correcting codes, word parallelism, and derandomization using conditional expectations.

*Key Words:* data structures, information retrieval, deterministic dictionaries, perfect hashing, error-correcting codes, word parallelism, derandomization

\*This work has been submitted to Academic Press for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

† Basic Research in Computer Science, Centre of the Danish National Research Foundation.

## 1. INTRODUCTION

*Dictionaries* are among the most fundamental data structures. A dictionary stores a subset  $S$  of a universe  $U$ , offering membership queries of the form “Is  $x \in S$ ?” for  $x \in U$ . It also supports the retrieval of *satellite data* associated with the elements of  $S$ , which are called *keys*. One distinguishes between the *dynamic* case, where the dictionary supports insertion and deletion of keys (with their satellite data), and the *static* case, where  $S$  does not change over time.

Several performance measures are of interest for dictionaries: the amount of space occupied by a dictionary, the time needed to construct or update it, and the time needed to answer a query. In this paper, our primary interest lies in obtaining static dictionaries with optimal query time and minimal space consumption that can be constructed rapidly by deterministic algorithms. By general dynamization results, this also has implications for deterministic dynamic dictionaries.

Our model of computation is the unit-cost *word RAM*. This natural and realistic model of computation has been the object of much recent research, surveyed in [13], which also offers a detailed definition. For a positive integer parameter  $w$ , called the *word length*, the memory cells of a word RAM store  $w$ -bit *words*, variously viewed as integers in  $\{0, \dots, 2^w - 1\}$  or as bit vectors in  $\{0, 1\}^w$ , and standard operations can be carried out on words in constant time. We adopt the *multiplication model*, whose instruction set includes addition, bitwise boolean operations, shifts, and multiplication and measure the space requirements of a word-RAM algorithm in units of  $w$ -bit words. Word-RAM algorithms can be *weakly nonuniform*, that is, access a fixed number of word-size constants that depend (only) on  $w$ . These constants, which we call *native constants*, may be thought of as computed at “compile time”.

The keys to be stored in a dictionary are assumed to be representable in single words, i.e., to come from the universe  $U = \{0, 1\}^w$ . For simplicity, we assume that each piece of satellite data occupies a single word of memory (if necessary, it can be a pointer to more bulky data).

Denoting the number of keys by  $n$ , we see that constant query time and  $O(n)$  space is the best for which one can hope. A seminal result of Fredman et al. [11] states that in the static case, a dictionary with these properties, henceforth referred to as *efficient*, is indeed possible. To achieve fast construction of the dictionary, Fredman et al. augment their RAM model with an additional resource: a source of random bits. In this setting, there is a construction algorithm with expected running time  $O(n)$ . This efficient dictionary and its construction algorithm are known as the *FKS scheme*.

The main result of this paper is an alternative efficient dictionary that can be constructed deterministically in  $O(n \log n)$  time. A standard dynamization technique yields a range of combinations of lookup time and update time. For example, we achieve constant lookup time with update time  $O(n^\epsilon)$ , for arbitrary constant  $\epsilon > 0$ .

### 1.1. Related work

Efficient dictionaries have been known for a long time for certain combinations of  $n$  and  $w$ . For  $n = \Omega(2^w)$ , e.g., a bit vector does the job. Tarjan and Yao [25] showed how to construct efficient static dictionaries when the number of keys and the size of the universe are polynomially related, i.e., when  $w = O(\log n)$ . As already mentioned, Fredman et al. demonstrated how to build efficient static dictionaries for arbitrary word sizes. Besides the randomized construction running in expected time  $O(n)$ , they gave a deterministic one with a running time of  $O(n^3 w)$ . A bottleneck in the deterministic algorithm is the choice of appropriate hash functions. It can be shown that exhaustive search in any *universal* class of hash functions [7] yields suitable functions. A more efficient way of conducting the search was devised by Raman [23], who lowered the deterministic construction time to  $O(n^2 w)$ . For  $w = n^{\Omega(1)}$ , a deterministic construction related to fusion trees [12] achieves  $O(n)$  construction time [13, Corollary 8]. Alon and Naor [1] used small-bias probability spaces to derandomize a variant of the FKS scheme, achieving construction time  $O(nw(\log n)^4)$ . However, their lookup operation requires evaluation of a linear function in time  $\Theta(w/\log n)$ , so the dictionary is not efficient unless  $w = O(\log n)$ . Another variant of the FKS scheme reduces the number of random bits to  $O(\log n + \log w)$ , while achieving  $O(n)$ -time construction with high probability [9].

Allowing randomization, the FKS scheme can be dynamized to support insertions and deletions in amortized expected constant time [10]. Without a source of random bits, the task of simultaneously achieving fast updates and constant query time seems considerably harder, and no solution with nontrivial performance bounds was previously known. Also, it is shown in [10] that approaches similar to the double hashing of the FKS scheme are destined to perform poorly in a deterministic setting. The best result when the update and query times are considered equally important is  $O(\sqrt{\log n / \log \log n})$  time per dictionary operation; it uses the data structure of Beame and Fich [5] with the dynamization result of Andersson and Thorup [4]. A different trade-off, lookup time  $O((\log \log n)^2 / \log \log \log n)$  and update time  $O((\log n)^2)$ , was recently obtained by Pagh [22]. For  $w = n^{\Omega(1)}$ , a standard dynamization of the fusion-tree-like data structure mentioned above provides constant-time lookups with update time  $O(n^\epsilon)$ , for arbitrary fixed  $\epsilon > 0$ .

An unpublished manuscript by Sundar [24] states an amortized lower bound of  $\Omega(\log \log_w n / \log \log \log_w n)$  per operation for a dynamic dictionary in the cell-probe model; this bound, in particular, implies the same lower bound on the word RAM.

Andersson et al. [3] have shown that a unit-cost RAM that allows efficient dictionaries must have an instruction of circuit depth  $\Omega(\log w / \log \log w)$ . Since this matches the circuit depth of multiplication, we see that efficient dictionaries are not possible with weaker instruction sets (in the circuit-depth sense). However, some work has been done on minimizing the query time in weaker models. In a word-RAM model providing only  $AC^0$  instructions, there is a tight bound of  $\Theta(\sqrt{\log n / \log \log n})$  on the query time [3, 14]. The upper bound uses nonstandard instructions. In a restricted word-RAM model that lacks multiplication, the best upper bound is  $\sqrt{\log n (\log \log n)^{1+o(1)}}$ , due to Brodnik et al. [6].

## 1.2. Our contributions

In this paper we sum up results contained in three consecutive conference publications [19, 15, 21]. Our main theorem is the following:

**THEOREM 1.1.** *A static dictionary for  $n$   $w$ -bit keys and their satellite data with constant lookup time and a space consumption of  $O(n)$  memory words can be constructed in  $O(n \log n)$  time on a word RAM with word length  $w$  by a weakly nonuniform deterministic algorithm that uses  $O(n)$  words of memory.*

The static dictionary can be turned into a dynamic one, supporting insertions and deletions, by a standard dynamization result [20, Theorem A].

**THEOREM 1.2.** *Let  $t(n) = O(\sqrt{\log n})$  be a nondecreasing function from  $\mathbb{N}$  to  $\mathbb{N}$  such that  $t(n)$  is computable in time and space  $O(n)$  and  $t(2n) = O(t(n))$ . Then there is a weakly nonuniform deterministic dynamic dictionary that runs on a word RAM with word length  $w$  and, when  $n$  elements are stored, uses  $O(n)$  words of memory and supports lookups in time  $O(t(n))$ , insertions in time  $O(n^{1/t(n)})$ , and deletions in time  $O(\log n)$ .*

The theorem is most interesting when  $t$  grows slowly. In particular, no previous deterministic linear-space dictionary combined lookup time  $O(\log \log n)$  with update time  $o(n)$ .

It should be noted that we make heavy use of weak nonuniformity. Whereas it is common to employ native constants that can be computed in  $O(w)$  or even  $O(\log w)$  time, our data structures depend on native constants that are not known to be computable in  $w^{O(1)}$  time.

## 2. TECHNICAL OVERVIEW

Let  $S \subseteq U$  denote the set of keys to be stored and take  $n = |S|$ . In order to prove Theorem 1.1, we show how to construct a function  $h : U \rightarrow \{0, \dots, m-1\}$ , where  $m = O(n)$ , that is 1-1 on  $S$  and can be stored in constant space and evaluated in constant time. Informally, such a function will be called an *efficient perfect hash function* for  $S$ . The desired efficient static dictionary consists of the description of  $h$  together with a hash table of size  $m$ .

Our approach is to first perform a *universe reduction* by finding a function  $\rho : U \rightarrow \{0, 1\}^r$ , with  $r = O(\log n)$ , that is 1-1 on  $S$  and can be stored in constant space and evaluated in constant time. Then an efficient perfect hash function  $h'$  is found for  $\rho(S) \subseteq \{0, 1\}^r$ . The desired function is  $h = h' \circ \rho$ .

The universe reduction is based on error-correcting codes, whose use in the context of hashing is introduced in Section 3. By applying an error-correcting code  $\psi$ , replacing each element  $x \in U$  by  $\psi(x) \in \{0, 1\}^{O(w)}$ , the Hamming distance (the number of differing bit positions) between any two elements of  $U$  can be made  $\Omega(w)$ . It is then possible to find a set  $D$  of  $O(\log n)$  *distinguishing bit positions* such that for every pair  $\{x, y\}$  of distinct keys in  $S$ ,  $\psi(x)$  and  $\psi(y)$  differ on  $D$ . Exploiting word parallelism, we show how to find such distinguishing bit positions in  $O(n \log n)$  time. Given these, Raman's deterministic selection of perfect hash functions can be used to construct a function  $\rho$  mapping to the desired range  $\{0, 1\}^r$ . We further show that a good error-correcting code can be picked from a universal family of functions from  $U$  to  $\{0, 1\}^{O(w)}$ . Since there are such families whose functions can be evaluated in constant time, we obtain an error-correcting code with the same property. The choice of an appropriate function is a source of weak nonuniformity.

In section 4 we show how to find an efficient perfect hash function for  $\rho(S) \subseteq \{0, 1\}^r$ . We first develop a randomized variant of the efficient perfect hash function of Tarjan and Yao. The construction algorithm is then derandomized using conditional expectations, yielding an  $O(n \log n)$ -time deterministic algorithm. Our algorithm is quite simple compared with the  $O(n(\log n)^5)$ -time algorithm described in [1].

## 3. UNIVERSE REDUCTION

In this section we describe the construction of a universe-reduction function  $\rho : U \rightarrow \{0, 1\}^r$ , where  $r = O(\log n)$ . The prime feature of  $\rho$  is that it is 1-1 on  $S$ . Because of this, it may be used to “translate” a search for  $x \in U$  into a search for  $\rho(x)$  within the smaller universe  $\{0, 1\}^r$ . Since we

are interested in constant-time queries,  $\rho$  should be evaluable in constant time. Constant space will suffice to store the description of  $\rho$ .

### 3.1. Distinguishing bit positions

Let  $\psi : U \rightarrow \{0, 1\}^{4w}$  be an error-correcting code of relative minimum distance  $\delta > 0$ . This means that for every pair  $\{x, y\}$  of distinct elements of  $U$ , the Hamming distance between  $\psi(x)$  and  $\psi(y)$  is at least  $4\delta w$ . We assume that  $\delta \leq 1/2$  (in fact, by the Plotkin bound for error-correcting codes [17, p. 41], this is always the case for  $w > 2$ ). Denote the  $i$ th bit of a bit string  $v$  (counted from the right, say) by  $v_i$ . We have the following:

**LEMMA 3.1.** *For every subset  $S$  of  $U$  of size  $n$ , there is a set  $D \subseteq \{1, \dots, 4w\}$  with  $|D| \leq 2 \log(n) / \log \frac{1}{1-\delta}$  such that for every pair  $\{x, y\}$  of distinct elements of  $S$ ,  $\psi(x)_d \neq \psi(y)_d$  for some  $d \in D$ .*

*Proof.* We will construct a sequence of sets  $D_0 = \emptyset \subseteq D_1 \subseteq \dots \subseteq D_k \subseteq \{1, \dots, 4w\}$  that are increasingly better at distinguishing elements of  $S$ . For a set  $D \subseteq \{1, \dots, 4w\}$ , we split  $S$  into  $2^{|D|}$  disjoint clusters  $C(S, D, 0^{|D|}), \dots, C(S, D, 1^{|D|})$ , one for each possible vector of bit values at the positions given by  $D$ . Define the *badness* of  $D$  as  $B(S, D) = \sum_{v \in \{0,1\}^{|D|}} \binom{|C(S, D, v)|}{2}$ , which is the number of pairs within the clusters. We will determine our sets such that  $|D_i| \leq i$  and  $B(S, D_i) < (1 - \delta)^i n^2 / 2$  for  $i = 0, \dots, k$ , a condition that clearly holds for  $i = 0$  with  $D_0 = \emptyset$ . Assume that  $D_i$  has been found for some  $i$  with  $0 \leq i < k$ . A pair of distinct elements in some cluster  $C(S, D_i, v)$  is also in  $C(S, D_i \cup \{d\}, v')$ , for some  $v'$ , for at most a fraction of  $1 - \delta$  of the possible choices of  $d \in \{1, \dots, 4w\}$ . By an averaging argument, it is possible to choose  $d$  such that  $B(S, D_i \cup \{d\}) \leq (1 - \delta)B(S, D_i)$ , and we let  $D_{i+1} = D_i \cup \{d\}$  for such a  $d$ . Setting  $k = \lceil 2 \log_{1/(1-\delta)} n \rceil$ , we achieve  $B(S, D_k) < 1$ , so we can take  $D_k$  as the desired set of distinguishing bits. ■

The lemma shows that a very simple hash function with polynomial-sized range can be found that it is 1-1 on the error-corrected representations of the keys: Simply use the projection on  $O(\log n)$  suitable bit positions.

We next make the proof of the lemma constructive by giving an algorithm for actually finding a small set  $D$  of distinguishing bit positions. When choosing a position, we need only care about the *nontrivial* clusters, those of size at least 2, since smaller clusters do not contribute to the badness. Maintaining the nontrivial clusters under addition of new bit positions is easy: Simply keep a linked list for each cluster; the lists can be split with respect to the bit value at a new position in a linear pass. Therefore the task of finding distinguishing positions boils down to that of finding a single *good* bit position, one that decreases the badness by a factor of at least  $1 - \delta$ .

LEMMA 3.2. *Let  $S$  be a subset of  $U$  of size  $n$ . Given linked lists of the nontrivial clusters of  $S$  corresponding to distinguishing positions  $D$ , a bit position  $d$  with*

$$B(S, D \cup \{d\}) \leq (1 - \delta) B(S, D)$$

*can be found deterministically in time and space  $O(n)$ .*

*Proof.* We show how to efficiently compute the badness  $B(S, D \cup \{d\})$  for each  $d \in \{1, \dots, 4w\}$ . The bit position  $d$  with the smallest badness must, by the proof of Lemma 3.1, satisfy  $B(S, D \cup \{d\}) \leq (1 - \delta)B(S, D)$ . For each  $d \in \{1, \dots, 4w\}$ , we perform the following steps:

1. For each nontrivial cluster  $C$ , compute  $s_C = \sum_{x \in C} x_d$ , the number of 1s in position  $d$ .
2. For each nontrivial cluster  $C$ , compute  $z_C = \binom{s_C}{2} + \binom{|C| - s_C}{2}$ , the combined badness of the two clusters resulting from  $C$  if  $d$  is included in  $D$ .
3. Compute the total badness  $\sum_C z_C$  over all (nontrivial) clusters  $C$ .

It is an easy matter to execute steps 1–3 in  $O(n)$  time for a single value of  $d$ . In order to execute steps 1–3 for all  $d \in \{1, \dots, 4w\}$  within the same time bound, we resort to word-level parallelism, viewing each string of  $4w$  bits as a bit vector. We first describe the algorithm under the following (unrealistic) assumptions:

A. Prior to the execution, each bit vector representing an element of  $S$  is “stretched” by a sufficiently large factor  $f$  through the introduction of  $f - 1$  zeros to the left of each original bit. This turns each original bit position into a *field* of  $f$  consecutive bit positions.

B. The machine instructions applicable to words can also be applied to vectors of  $4w$  fields and still take constant time.

Under these assumptions, it is trivial to execute steps 1 and 3 in  $O(n)$  time for all  $d \in \{1, \dots, 4w\}$ : The only operation needed is field-wise addition, which can be realized through ordinary word-level addition. The field width  $f$  is assumed to be sufficiently large to prevent overflows between fields. Step 2 needs the following additional operations:

- Replication of a value  $m$ , stored in the rightmost field, to all other fields. This can be done by multiplying  $m$  by the constant  $1_f$  that contains 1 in every field. For the time being, we assume  $1_f$  to be a native constant.
- Field-wise subtraction with a nonnegative result, which can be realized through word-level subtraction.

- Field-wise multiplication. This can be realized through the usual shift-and-add algorithm that successively tests each bit of one factor and, if it is 1, adds an appropriately shifted copy of the other factor to an accumulated sum. We refer to [2, Sect. 3] for a description of the low-level details needed to carry out such steps as field-wise conditional addition based on a comparison with zero, noting only that the constant  $1_f$  comes in handy here as well. The time needed is  $O(f)$ .
- Field-wise division of even integers by 2, which can be realized through a right shift.

Since the field values manipulated by the algorithm are polynomial in  $n$ , a field width  $f$  of  $O(\log n)$  clearly suffices. However, assumptions A and B are not realistic even for this value of  $f$ . In particular, vectors of  $4w$  fields of  $f$  bits each occupy  $\Theta(f)$   $w$ -bit words, and operations such as adding two vectors take  $\Theta(f)$  time. We counter this problem by using a variable field width, storing small numbers in small fields and large numbers, of which there are few, in large fields. We begin by describing how to double and halve the field width.

In order to double the field width of a vector from  $f$  to  $2f$ , we use the constant  $1_{2f}$  to create a mask whose  $f$ -bit fields contain alternately only 0s and only 1s — again, the reader is referred to [2] for programming details. Using this mask, it is easy to separate the odd- and the even-numbered fields, storing each group of fields in a separate vector. This spreads the original vector over twice as many words and (implicitly) changes the field width from  $f$  to  $2f$ . Halving the field width of a vector from  $2f$  to  $f$  can be done very easily by breaking the vector into halves and forming the disjunction of the halves after shifting one half by  $f$  bits. Doubling the field width scrambles the order of the fields, but halving the field width returns the fields to their original order.

In order to carry out step 1, we sum the vectors of each nontrivial cluster  $C$  in a minimum-height binary tree, using a constant field width at the leaves and larger field widths at inner nodes of the tree. The time needed at an inner node of field width  $f$ , including any necessary field doubling for the vectors produced at the children of the node, is  $O(f)$ . Since it is easy to see that a field width of  $O(i)$  suffices for an inner node at height  $i$ , for all  $i \geq 1$ , while there are only  $O(|C|/2^i)$  such nodes, the total time needed is  $O(|C| \sum_{i=1}^{\infty} i/2^i) = O(|C|)$ . Thus step 1 takes  $O(n)$  time.

For each nontrivial cluster  $C$ , since a field width of  $O(\log |C|)$  suffices, the computation of step 2 can be carried out in  $O((\log |C|)^2)$  time. Over all nontrivial clusters, this sums to  $O(n)$  time.

For step 3, we divide the nontrivial clusters into *size groups*: If a cluster contains between  $2^j$  and  $2^{j+1} - 1$  elements, for some integer  $j \geq 1$ , it is put in size group  $j$ . Separately for each value of  $j$ , we then sum the

vectors computed in step 2 for all clusters in size group  $j$  in a minimum-height binary tree. If  $W_j$  is the total size of all clusters in size group  $j$ , the number of nodes at height  $i$  in the tree is  $O(W_j/2^{i+j})$ , for all  $i \geq 1$ , and a field width of  $O(i+j)$  suffices at each such node. As in the analysis of step 1, the summation within size group  $j$  therefore takes  $O(W_j)$  time, which sums to  $O(n)$  over all values of  $j$ . What remains is to add  $O(\log n)$  vectors, one for each size group. Since the maximum field width is  $O(\log n)$ , this can be done in  $O((\log n)^2)$  time. Thus step 3 also takes  $O(n)$  time.

The output of steps 1–3 is a vector of  $4w$  fields, each of which specifies the badness associated with the corresponding bit position. The field width  $f$  is  $O(\log n)$ , for which reason the set of fields containing the minimum badness can be computed in  $O((\log n)^2)$  time by binary search over the range of possible minima. Specifically, we can assume the output of the binary search to be a vector, each field of which contains 1 if the corresponding badness is minimum, and 0 otherwise. We now reduce the field width back to 1 by  $\log f$  halvings, which also restores the original order of the fields. The result is a nonzero  $4w$ -bit vector, each 1 of which indicates a good bit position. To get hold of a single good position, we compute the position of the most significant bit set to 1. This can be done in constant time, employing weak nonuniformity [12, p. 431–432].

A final issue is the dependence on the constants  $1_{2^i}$  for  $i = 1, \dots, \log f_{\max}$ , where  $f_{\max} = O(\log n)$ . If  $w < n$ , we can compute  $1_{2^i}$  in  $O(\log n)$  time by  $O(\log w)$  shift-and-or steps, each of which doubles the number of fields containing a 1. Otherwise we use native constants  $l = \Theta(\sqrt{w})$ , chosen as a power of 2, and  $1_l$  together with the number  $Q$ , composed of segments of  $l$  bits each, the  $i$ th of which is a “piece” of  $1_{2^i}$ . We can easily pick out the  $i$ th segment from  $Q$ . Multiplying the segment with  $1_l$  yields the required constant  $1_{2^i}$ . ■

**THEOREM 3.1.** *Let  $S$  be a subset of  $U$  of size  $n$  and suppose that  $\psi : U \rightarrow \{0, 1\}^{4w}$  is an error-correcting code with minimum relative distance  $\delta$  for some constant  $\delta > 0$ . Then there is a bit vector  $d \in \{0, 1\}^{4w}$  containing  $O(\log n)$  1s for which  $\rho_d : x \mapsto (\psi(x) \text{ AND } d)$  is 1-1 on  $S$ , and such a bit vector can be computed deterministically from  $\{\psi(x) \mid x \in S\}$  in  $O(n \log n)$  time and  $O(n)$  space.*

### 3.2. Unit-cost error-correcting codes

In order to evaluate the function  $\rho_d$  of Theorem 3.1 in constant time, we need an error-correcting code that can be evaluated in constant time. Our construction is based on universal families of hash functions.

DEFINITION 3.1. [26, 18] For  $c > 0$ , a family  $\mathcal{H}$  of functions from  $U$  to  $V$  is  $(c, 2)$ -universal if, for all  $x_1, x_2 \in U$  and all  $y_1, y_2 \in V$ , the probability that  $h(x_1) = y_1$  and  $h(x_2) = y_2$  is at most  $c/|V|^2$  when  $h \in \mathcal{H}$  is chosen uniformly at random.

PROPOSITION 3.1. Let  $\mathcal{H}$  be a  $(2, 2)$ -universal family of functions from  $\{0, 1\}^w$  to  $\{0, 1\}^{4w}$ . For all  $\delta$  with  $0 < \delta \leq \frac{1}{2}$ , a random member of  $\mathcal{H}$  is an error-correcting code of relative minimum distance  $\delta$  with probability at least  $1 - ((\frac{e}{\delta})^{4\delta}/4)^w$ .

*Proof.* Assume first that  $\delta \geq \frac{1}{4w}$ . The number of vectors in  $\{0, 1\}^{4w}$  within Hamming distance  $k \geq 1$  of a fixed vector is

$$\begin{aligned} \sum_{i=0}^k \binom{4w}{i} &\leq \left(\frac{4w}{k}\right)^k \sum_{i=0}^k \binom{4w}{i} \left(\frac{k}{4w}\right)^i \\ &\leq \left(\frac{4w}{k}\right)^k \left(1 + \frac{k}{4w}\right)^{4w} \quad (\text{by the binomial theorem}) \\ &\leq \left(\frac{4w}{k}\right)^k e^k = \left(\frac{4ew}{k}\right)^k. \end{aligned}$$

This means that for all  $x_1, x_2 \in \{0, 1\}^w$  with  $x_1 \neq x_2$  and for a random function  $h \in \mathcal{H}$ , the probability that the Hamming distance between  $h(x_1)$  and  $h(x_2)$  is no larger than  $k$  is at most  $2^{1-4w} \left(\frac{4ew}{k}\right)^k$  (where we used  $(2, 2)$ -universality). The probability that this happens for *any* of the  $\binom{2^w}{2} < 2^{2w}/2$  such pairs is bounded by  $2^{-2w} \left(\frac{4ew}{k}\right)^k$ . Setting  $k = \lfloor 4\delta w \rfloor$ , we see that  $h$  fails to have minimum relative distance  $\delta$  with probability at most  $2^{-2w} \left(\frac{4ew}{\lfloor 4\delta w \rfloor}\right)^{\lfloor 4\delta w \rfloor} \leq 2^{-2w} \left(\frac{4ew}{4\delta w}\right)^{4\delta w} = ((\frac{e}{\delta})^{4\delta}/4)^w$ .

If  $\delta < \frac{1}{4w}$ , the desired property of  $h$  is simply that it should be injective. The probability that this is not the case is bounded by  $2^{-2w} \leq 2^{-2w} \left(\frac{4ew}{4\delta w}\right)^{4\delta w} = ((\frac{e}{\delta})^{4\delta}/4)^w$ . ■

The quantity  $(\frac{e}{\delta})^{4\delta}/4$  converges to  $1/4$  as  $\delta$  approaches 0, so the success probability of Proposition 3.1 is positive for sufficiently small values of  $\delta$  for all  $w$ . Thus we can indeed find an error-correcting code with relative minimum distance  $\delta$  for some constant  $\delta > 0$ . As a concrete example, assume that  $w > 10$  and let  $\mathcal{H}$  be a  $(2, 2)$ -universal family of functions from  $\{0, 1\}^w$  to  $\{0, 1\}^{4w}$ . The proposition shows that more than half the functions in  $\mathcal{H}$  are error-correcting codes with relative minimum distance  $1/10$ .

Many  $(2, 2)$ -universal families are known. Moreover, when the range is  $\{0, 1\}^{O(w)}$ , there are such families whose functions can be stored in constant space and evaluated in constant time. One example is  $\{x \mapsto ((ax + b) \bmod p) \bmod 2^{4w} \mid a, b \in \{0, \dots, p - 1\}\}$ , where  $p$  is a fixed prime between  $2^{4w}$  and  $2^{4w+1}$ . Multiplication of  $O(w)$ -bit numbers can be done using a constant number of single-word multiplications and additions. Also, as noted by Knuth [16, p. 509], forming the remainder modulo a constant  $p$  can be carried out in constant time with multiplications and shifts, so a division instruction is not needed to evaluate the functions in constant time. Another, very appealing, such family is  $\{x \mapsto ((ax + b) \bmod 2^{5w}) \operatorname{div} 2^w \mid a, b \in \{0, \dots, 2^{5w} - 1\}\}$ , which has parameter 1 [8, Theorem 3(b)]. This family can be simplified to  $\{x \mapsto ax \mid a \in \{0, \dots, 2^{5w} - 1\}\}$  without decreasing the relative minimum distance of the corresponding error-correcting code by more than a constant factor. A direct proof of the error-correction property of this family, along with some results more general than those needed here, can be found in [19].

### 3.3. Finishing the construction

We still need to address the issue of mapping injectively to  $O(\log n)$  consecutive bits. We must “gather” the distinguishing bits in an interval of  $O(\log n)$  positions. For every  $D \subseteq \{1, \dots, r\}$ , define  $Z_D = \{x \in \{0, 1\}^r \mid x_i = 1 \Rightarrow i \in D\}$ , the set of  $r$ -bit vectors that have only zeros outside the positions given by  $D$ .

**LEMMA 3.3.** *Let  $D$  be a subset of  $\{1, \dots, 4w\}$  of size  $O(\log n)$ . Then there is a function  $\rho' : \{0, 1\}^{4w} \rightarrow \{0, 1\}^r$ , where  $r = O(\log n)$ , that is 1-1 on  $Z_D$  and can be evaluated in constant time, and a constant-size description of such a function can be computed deterministically in  $o(n)$  time and space.*

*Proof.* Without loss of generality we can assume that  $w = O(\sqrt[4]{n})$ : If this is not the case, begin by using a method of Fredman and Willard [12, p. 428–429] to gather the bits with positions in  $D$  within  $O((\log n)^4)$  consecutive positions by multiplying with a suitable integer  $M_D$ . The procedure for finding  $M_D$  runs in time  $(\log n)^{O(1)}$ .

Partition  $D$  into a constant number of sets  $D_1, \dots, D_k$  of size at most  $\frac{1}{4} \log n$ . Using the algorithm of Raman [23], we can find hash functions  $\rho_1, \dots, \rho_k$  with range  $\{0, 1\}^{\lceil \frac{1}{2} \log n \rceil}$ , perfect for  $Z_{D_1}, \dots, Z_{D_k}$ , respectively, in time and space  $O((\sqrt[4]{n})^2 w) = o(n)$ . Given an argument value, masking out the bit positions not in  $D_i$  and evaluating  $\rho_i$ , for  $i = 1, \dots, k$ , and concatenating the resulting values can be done in constant time. This defines the function  $\rho'$ . ■

Combining Theorem 3.1 and Lemma 3.3, we have the desired result on universe reduction:

LEMMA 3.4. *Let  $S$  be a subset of  $U$  of size  $n$ . Then there is a function from  $U$  to  $\{0, 1\}^r$ , with  $r = O(\log n)$ , that is 1-1 on  $S$  and can be evaluated in constant time, and a constant-size description of such a function can be computed deterministically in time  $O(n \log n)$  and space  $O(n)$ .*

#### 4. UNIVERSES OF POLYNOMIAL SIZE

In this section we develop a variant of the double-displacement scheme of Tarjan and Yao [25], which computes an efficient perfect hash function for  $w = O(\log n)$ . The construction algorithm of Tarjan and Yao is deterministic and has worst-case complexity  $\Theta(n^2)$ . We first show how to achieve expected construction time  $O(n)$  with a randomized algorithm. This algorithm is then derandomized using the method of conditional expectations, which yields a deterministic  $O(n \log n)$ -time algorithm.

##### 4.1. Reduction to universes of quadratic size

Following Tarjan and Yao, we observe that bit vectors of length  $O(\log n)$  can be regarded as constant-length strings over an alphabet of size  $n$ . The trie (with  $n$ -way branching) of such strings permits lookup of elements (and associated values) in constant time. Although each of the  $O(n)$  nodes of the trie uses a table of size  $n$ , only  $O(n)$  entries of all tables contain important information (an element or a pointer). That is, to store all tables in space  $O(n)$ , it suffices to construct an efficient perfect hash function for the set of important entries within the universe of all  $O(n^2)$  table entries. For this reason Tarjan and Yao proceed to study the case  $w \leq 2 \log n + O(1)$ .

##### 4.2. Randomized double displacement

Our aim is to find a function  $h : U \rightarrow \{0, 1\}^r$ , with  $r = \log n + O(1)$ , such that there are no *collisions* under  $h$ , i.e., pairs  $\{x, y\}$  of distinct keys in  $S$  with  $h(x) = h(y)$ . The *displacement* method of Tarjan and Yao can be viewed as a way of taking an “imperfect” function  $f : U \rightarrow \{0, 1\}^r$  and generating a new function with fewer collisions between the keys. The method needs “advice” in the form of a function  $g : U \rightarrow \{0, 1\}^r$  such that  $x \mapsto (f(x), g(x))$  is 1-1 on  $S$ .

The idea of Tarjan and Yao is to use  $f(x)$  as an index into a table of suitably chosen *displacement values*  $a_v \in \{0, 1\}^r$ , with  $v \in \{0, 1\}^r$ . Then the function  $x \mapsto g(x) \oplus a_{f(x)}$ , where  $\oplus$  denotes bitwise exclusive-or, may have far fewer collisions than  $f$ . Also,  $x \mapsto (g(x) \oplus a_{f(x)}, f(x))$  is 1-1 on  $S$ , so the procedure can be repeated. Two repetitions will suffice; hence the

term “double displacement”. Our contribution is an efficient procedure for finding suitable displacement values.

DEFINITION 4.1. For  $q \geq 0$ , a pair of functions  $(f, g)$ , both mapping from  $U$  to  $\{0, 1\}^r$ , is  $q$ -good if  $f$  has at most  $q$  collisions and  $x \mapsto (f(x), g(x))$  is 1-1 on  $S$ .

LEMMA 4.1. Suppose that  $(f, g)$  is  $q$ -good and that  $r \geq \log n + 1$ . Then there exist  $a_v \in \{0, 1\}^r$ , for  $v \in \{0, 1\}^r$ , such that  $(x \mapsto g(x) \oplus a_{f(x)}, f)$  is  $q'$ -good, where  $q' = \min\{n, \lfloor 2^{3-r} q \rfloor n\}$ . On input  $\{(f(x), g(x)) \mid x \in S\}$ , such values  $a_v$  can be computed by a randomized algorithm in expected time  $O(n)$  and space  $O(n)$ .

Let us first see how to use the lemma to obtain an efficient perfect hash function for  $S$ . We will need  $r > \max\{w/2, \log n + 3\}$ . Since  $r \geq w/2$ , it is trivial to find a pair  $(f, g)$  of constant-time-evaluable functions that is  $\binom{n}{2}$ -good (e.g., let  $f(x)$  and  $g(x)$  be the first and the last  $r$  bits of  $x$ , respectively). By Lemma 4.1, we can find  $a_v \in \{0, 1\}^r$ , for  $v \in \{0, 1\}^r$ , such that  $(x \mapsto g(x) \oplus a_{f(x)}, f)$  is  $n$ -good. Applying Lemma 4.1 to the pair  $(x \mapsto g(x) \oplus a_{f(x)}, f)$ , we obtain values  $b_v \in \{0, 1\}^r$ , for  $v \in \{0, 1\}^r$ , such that  $(x \mapsto f(x) \oplus b_{g(x) \oplus a_{f(x)}}, x \mapsto g(x) \oplus a_{f(x)})$  is  $(\lfloor 2^{3-r} n \rfloor n)$ -good. By the choice of  $r$ ,  $\lfloor 2^{3-r} n \rfloor = 0$ , so the function  $x \mapsto f(x) \oplus b_{g(x) \oplus a_{f(x)}}$  is 1-1 on  $S$ . When  $w \leq 2 \log n + O(1)$  we can choose  $r = \log n + O(1)$ , so the hash-function parameters use space  $O(n)$ , and the range  $\{0, 1\}^r$  has size  $O(n)$ . Thus  $x \mapsto f(x) \oplus b_{g(x) \oplus a_{f(x)}}$  is the desired efficient perfect hash function.

Summing up sections 4.1 and 4.2, we have, for  $w = O(\log n)$ , a randomized algorithm constructing an efficient perfect hash function in expected time  $O(n)$  and space  $O(n)$ .

*Proof of lemma 4.1.* For  $v \in \{0, 1\}^r$ , let  $S_v = \{x \in S \mid f(x) = v\}$ . Our algorithm starts by bucket-sorting the  $(f(x), g(x))$ -pairs, for  $x \in S$ , by their first coordinates. It is then easy to compute a permutation  $v_1, \dots, v_{2^r}$  of  $\{0, 1\}^r$  with  $|S_{v_1}| \geq |S_{v_2}| \geq \dots \geq |S_{v_{2^r}}|$ . We now successively compute  $a_{v_1}, \dots, a_{v_{2^r}}$ ; i.e., the sets  $S_v$  are processed in some order of nonincreasing size.

Before the  $j$ th step of the computation, for  $1 \leq j \leq 2^r$ , the algorithm will have determined  $a_{v_1}, \dots, a_{v_{j-1}}$ , and thus also the value of  $g(x) \oplus a_{f(x)}$  for all  $x \in S_{v_{<j}}$ , where  $S_{v_{<j}} = \bigcup_{i=1}^{j-1} S_{v_i}$ . We maintain counts of the values determined so far,  $m_v = |\{x \in S_{v_{<j}} \mid g(x) \oplus a_{f(x)} = v\}|$ , for  $v \in \{0, 1\}^r$ . If  $a_{v_j}$  is picked at random from  $\{0, 1\}^r$ , the expected number of new collisions, i.e., pairs  $(x, y) \in S_{v_{<j}} \times S_{v_j}$  with  $g(x) \oplus a_{f(x)} =$

$g(y) \oplus a_{f(y)} = g(y) \oplus a_{v_j}$ , is  $|S_{v_j}| |S_{v_{<j}}| / 2^r$ . The algorithm aims to introduce at most twice this number of collisions, i.e., to find  $a_{v_j}$  such that  $\sum_{y \in S_{v_j}} m_{g(y) \oplus a_{v_j}} \leq \lfloor 2 |S_{v_j}| |S_{v_{<j}}| / 2^r \rfloor$  (we can round down since the left-hand side is an integer). By Markov's inequality, the expected number of random attempts required to find such an  $a_{v_j}$  is no more than 2. Each attempt takes time  $O(|S_{v_j}|)$ , so the expected running time for all steps is  $O(n)$ .

It remains to be seen that the number of collisions of  $x \mapsto g(x) \oplus a_{f(x)}$  is no larger than  $q'$ . Note that the number of collisions of  $f$  is  $\sum_{v \in \{0,1\}^r} \binom{|S_v|}{2}$  and, by assumption, is at most  $q$ . Let  $j^* = |\{v \in \{0,1\}^r \mid |S_v| > 1\}|$ . The number of collisions of  $x \mapsto g(x) \oplus a_{f(x)}$  is at most

$$\begin{aligned}
& \sum_{j=1}^{2^r} \lfloor 2 |S_{v_j}| |S_{v_{<j}}| / 2^r \rfloor \\
& \leq \sum_{j=1}^{j^*} \lfloor 2^{1-r} |S_{v_j}| \sum_{i=1}^{j-1} |S_{v_i}| \rfloor && \text{(as } 2n/2^r < 1) \\
& \leq \sum_{j=1}^{j^*} \lfloor 2^{1-r} \min\{|S_{v_j}| n, \sum_{i=1}^{j-1} |S_{v_i}|^2\} \rfloor && \text{(as } |S_{v_j}| \leq |S_{v_i}| \text{ for } i < j) \\
& \leq \sum_{j=1}^{j^*} \min\{2^{1-r} |S_{v_j}| n, \lfloor 2^{3-r} \sum_{i=1}^{j-1} \binom{|S_{v_i}|}{2} \rfloor\} && \text{(as } |S_{v_i}| \geq 2 \text{ for } i \leq j^*) \\
& \leq \min\{n, \lfloor 2^{3-r} q \rfloor n\} . \quad \blacksquare
\end{aligned}$$

### 4.3. Derandomizing double displacement

In this section we employ the method of conditional expectations to obtain a deterministic  $O(n \log n)$ -time version of the algorithm of section 4.2. Recall the problem solved in the randomized part of the algorithm: Given a table of values  $m_v$ , for  $v \in \{0,1\}^r$ , and a set  $X \subseteq \{0,1\}^r$ , find  $a \in \{0,1\}^r$  such that  $\sum_{x \in X} m_{x \oplus a} \leq \lfloor 2^{1-r} |X| \sum_{v \in \{0,1\}^r} m_v \rfloor$ .

We show how to find  $a$  deterministically in  $O(|X|r) = O(|X| \log n)$  time. That is, the time for finding a displacement value is  $O(\log n)$  times that expected for the randomized algorithm. To do this we maintain an extension of the table, storing values  $m_u$  for all bit strings  $u$  of length at most  $r$ . For  $k = 0, \dots, r$ , let  $\pi_k(v)$  denote the  $k$ -bit prefix of  $v \in \{0,1\}^r$ , and for  $u \in \{0,1\}^k$  define  $Z_u = \{v \in \{0,1\}^r \mid \pi_k(v) = u\}$  as the set of bit strings of length  $r$  with  $u$  as a prefix. Then the extended table is defined by  $m_u = \sum_{v \in Z_u} m_v$ .

We can think of the extended table as a binary trie whose leaves (indexed by strings of length  $r$ ) contain the original table entries and each of whose internal nodes contains the sum over all leaves in its sub-trie. The extension can be initialized and maintained during  $n$  updates of leaves in time  $O(nr) = O(n \log n)$ .

Starting with  $u_0$ , the empty string, we show how to find a sequence of bit strings  $u_0, \dots, u_r$ , where  $u_k \in \{0, 1\}^k$ , such that the expected value of  $\sum_{x \in X} m_{x \oplus a}$ , when  $a \in Z_{u_k}$  is chosen uniformly at random, is at most  $2^{-r} |X| \sum_{v \in \{0, 1\}^r} m_v$ , for  $k = 0, \dots, r$ . Since  $Z_{u_r} = \{u_r\}$ , we must have  $\sum_{x \in X} m_{x \oplus u_r} \leq \lfloor 2^{-r} |X| \sum_{v \in \{0, 1\}^r} m_v \rfloor$  (rounding down being justified by the integrality of the left-hand side), and we can take  $a = u_r$ .

For  $u_0$  the requirement is clearly met, so for  $1 \leq k < r$  the task is to extend  $u_{k-1}$  to  $u_k$  without increasing the expected value. By linearity of expectation, we can always achieve this by extending  $u_{k-1}$  by either 0 or 1. An appropriate extension can be found by computing the expectations in time  $O(|X|)$ :

**LEMMA 4.2.** *For every  $u \in \{0, 1\}^k$ , where  $0 \leq k \leq r$ , the expectation of  $\sum_{x \in X} m_{x \oplus a}$ , when  $a \in Z_u$  is chosen uniformly at random, is  $2^{k-r} \sum_{x \in X} m_{\pi_k(x) \oplus u}$ .*

*Proof.* For every  $x \in X$  we have  $\sum_{a \in Z_u} m_{x \oplus a} = m_{\pi_k(x) \oplus u}$  by definition, so the expected value of  $m_{x \oplus a}$ , when  $a \in Z_u$  is chosen uniformly at random, is  $2^{k-r} m_{\pi_k(x) \oplus u}$ . The lemma follows by linearity of expectation.  $\blacksquare$

## ACKNOWLEDGMENT

The second author thanks Martin Dietzfelbinger for helpful discussions on hashing and error-correcting codes. The third author thanks Gerth Stølting Brodal, Gudmund Skovbjerg Frandsen, and Theis Rauhe for useful feedback.

## REFERENCES

1. N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.
2. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comput. System Sci.*, 57(1):74–93, 1998.
3. A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC<sup>0</sup> RAMs: Query time  $\Theta(\sqrt{\log n / \log \log n})$  is necessary and sufficient. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 441–450. IEEE Comput. Soc. Press, Los Alamitos, CA, 1996.
4. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 335–342. ACM Press, New York, 2000.
5. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304. ACM Press, New York, 1999.
6. A. Brodnik, P. B. Miltersen, and J. I. Munro. Trans-dichotomous algorithms without multiplication — some upper and lower bounds. In *Proceedings of the 5th International Workshop on Algorithms and Data Structures (WADS '97)*, volume 1272 of *Lecture Notes in Computer Science*, pages 426–439. Springer-Verlag, Berlin, 1997.
7. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.
8. M. Dietzfelbinger. Universal hashing and  $k$ -wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS '96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, Berlin, 1996.
9. M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, Berlin, 1992.
10. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.
11. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.
12. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.
13. T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, Berlin, 1998.
14. T. Hagerup. Simpler and faster dictionaries on the AC<sup>0</sup> RAM. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 79–90. Springer-Verlag, Berlin, 1998.
15. T. Hagerup. Fast deterministic construction of static dictionaries. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pages 414–418. ACM Press, New York, 1999.

16. D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., 1973. Volume 3. Sorting and searching, Addison-Wesley Series in Computer Science and Information Processing.
17. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Publishing Co., Amsterdam, 1977. North-Holland Mathematical Library, Vol. 16.
18. K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, 21:339–374, 1984.
19. P. B. Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 556–563. ACM Press, New York, 1998.
20. M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.*, 12(4):168–173, 1981.
21. R. Pagh. Faster deterministic dictionaries. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 487–493. ACM Press, New York, 2000.
22. R. Pagh. A new trade-off for deterministic dictionaries. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT '00)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000.
23. R. Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proceedings of the 4th Annual European Symposium on Algorithms (ESA '96)*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, Berlin, 1996.
24. R. Sundar. A lower bound on the cell probe complexity of the dictionary problem. Manuscript, 1993.
25. R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.
26. M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *J. Comput. System Sci.*, 22(3):265–279, 1981.