SIAM J. COMPUT.
Vol. 31, No. 2, pp. 353–363

LOW REDUNDANCY IN STATIC DICTIONARIES WITH CONSTANT QUERY TIME*

RASMUS PAGH†

Abstract. A *static dictionary* is a data structure storing subsets of a finite universe U, answering membership queries. We show that on a unit cost RAM with word size $\Theta(\log |U|)$, a static dictionary for n-element sets with constant worst case query time can be obtained using $B+O(\log\log|U|)+o(n)$ bits of storage, where $B = \lceil \log_2 \binom{|U|}{n} \rceil$ is the minimum number of bits needed to represent all n-element subsets of U.

Key words. information retrieval, dictionary, hashing, redundancy, compression

AMS subject classifications. 68P10, 68P20, 68P30

PII. S0097539700369909

1. Introduction.

Consider the problem of storing a subset S of a finite set U, such that membership queries, "$u \in S$?", can be answered in worst case constant time on a unit cost RAM. We are interested only in membership queries, so we assume that $U = \{0, \ldots, m-1\}$. Also, we restrict attention to the case where the RAM has word size $\Theta(\log m)$. In particular, elements of U can be represented within a constant number of machine words, and the usual RAM operations (including multiplication) on numbers of size $m^{O(1)}$ can be done in constant time.

Our goal will be to solve this, the *static dictionary problem*, using little memory, measured in consecutive bits. We express the complexity in terms of $m = |U|$ and $n = |S|$, and often consider the asymptotics when n is a function of m. Since the queries can distinguish any two subsets of U, we need at least $\binom{m}{n}$ different memory configurations, that is, at least $B = \lceil \log \binom{m}{n} \rceil$ bits. (Log is base 2 throughout this paper.) We will focus on the case $n \leq m/2$ and leave the symmetry implications to the reader. Using Stirling's approximation to the factorial function, one can derive the following (where $e = 2.718\ldots$ denotes the base of the natural logarithm):

$$B = n \log(e\, m/n) - \Theta(n^2/m) - O(\log n). \tag{1.1}$$

It should be noted that using space very close to B is only possible if elements of S are stored *implicitly*, since explicitly representing all elements requires $n \log m = B + \Omega(n \log n)$ bits.

Previous work. The static dictionary is a very fundamental data structure and has been much studied. We focus on the development in space consumption for schemes with worst case constant query time. A bit vector is the simplest possible solution to the problem, but the space complexity of m bits is poor compared to B unless $n \approx m/2$. By the late 1970s, known dictionaries with a space complexity of $O(n)$ words (i.e., $O(n \log m)$ bits) either had nonconstant query time or worked only for restricted universe sizes [4, 16, 17].

*Received by the editors April 3, 2000; accepted for publication (in revised form) November 27, 2000; published electronically August 8, 2001. A preliminary version of this paper appeared at ICALP 1999 [12].

http://www.siam.org/journals/sicomp/31-2/36990.html

†BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation), University of Aarhus, Aarhus, Denmark (pagh@brics.dk).

The breakthrough paper of Fredman, Komlós, and Szemerédi [7] described a general constant time hashing scheme, from now on referred to as the FKS dictionary, using $O(n)$ words. A refined solution in the paper uses $B+O(n\log n+\log\log m)$ bits, which is $O(B)$ when $n=m^{1-\Omega(1)}$. Brodnik and Munro [2] constructed the first static dictionary using $O(B)$ bits with no restrictions on m and n. They later improved the bound to $B+O(B/\log\log\log m)$ bits [3].

No nontrivial space lower bound is known in a general model of computation. However, various restrictions on the data structure and the query algorithm have been successfully studied. Yao [17] showed that if words of the data structure must contain elements of S, the number of words necessary for $o(\log n)$ time queries cannot be bounded by a function of n. Fich and Miltersen [6] studied a RAM with standard unit cost arithmetic operations but *without division and bit operations*, and showed that query time $o(\log n)$ requires $\Omega(m/n^\epsilon)$ words of memory for any $\epsilon>0$. Miltersen [11] showed that on a RAM with bit operations but *without multiplication*, one needs m^ϵ words, for some $\epsilon>0$, when $n=m^{o(1)}$.

This paper. In this paper we show that it is possible to achieve space usage very close to the information theoretical minimum of B bits. The additional term of the space complexity, which we will call the *redundancy*, will be $o(n)+O(\log\log m)$ bits. More precisely we show the following.

THEOREM 1.1. *The static dictionary problem with worst case constant query time can be solved using* $B+O(n\,(\log\log n)^2/\log n+\log\log m)$ *bits of storage.*

Theorem 1.1 improves the redundancy of $\Omega(\min(n\log\log m, m/(\log n)^{o(1)}))$ obtained by Brodnik and Munro [3] by a factor of $\Omega(\min(n, \log\log m\,(\log n)^{1-o(1)}))$. For example, when $n=\Omega(m)$ we obtain space $B+B/(\log B)^{1-o(1)}$ as compared to $B+B/(\log B)^{o(1)}$. For $n=\Theta(\log m)$ our space usage is $B+n/(\log n)^{1-o(1)}$ rather than $B+\Omega(n\log n)$.

We will also show how to associate *satellite data* from a finite domain to each element of S, with nearly the same redundancy as above.

Our main observation is that one can save space by "compressing" the hash table part of data structures based on (perfect) hashing, storing in each cell not an element of S but only a *quotient* — information that distinguishes the element from the part of U that hashes to the cell.[1] This technique, referred to as quotienting, is presented in section 2, where a $B+O(n+\log\log m)$ bit dictionary is exhibited. Section 3 outlines how to improve the dependency on n to that of theorem 1.1. The construction uses a dictionary supporting *rank* and *predecessor* queries, described in section 4. Section 5 gives the details of the construction and an analysis of the redundancy. The sizes of the data structures described are not computed explicitly. Rather, indirect means are employed to determine the number of redundant bits. While direct summation and comparison with (1.1) would be possible, it is believed that the proofs given here contain more information about the "nature" of the redundancy.

Without loss of generality, we will assume that n is greater than some sufficiently large constant. This is to avoid worrying about special cases for small values of n.

2. First solution. This section presents a static dictionary with a space consumption of $B+O(n+\log\log m)$ bits. Consider a *minimal perfect hash function* for S, i.e., $h_{\text{perfect}}: U\to\{0,\ldots,n-1\}$ which is 1-1 on S. Defining an n-cell *hash table* T

[1] The term "quotient" is inspired by the use of modulo functions for hashing, in which case the integer quotient is exactly what we want in the cell.

such that $T[i] = s_i$ for the unique $s_i \in S$ with $h_{\text{perfect}}(s_i) = i$, the following program implements membership queries for S:

function lookup(x)
  return ($T[h_{\text{perfect}}(x)] = x$);
end.

A more compact data structure results from the observation that $T[i]$ does not need to contain s_i itself ($\lceil \log m \rceil$ bits), but only enough information to identify s_i within $U_i = \{u \in U \mid h_{\text{perfect}}(u) = i\}$ ($\lceil \log |U_i| \rceil$ bits). We will be slightly less ambitious, though, and not necessarily go for the minimal number of bits in each hash table cell. In particular, to allow efficient indexing we want the number of bits to be the same for each table cell. Note that $\log(m/n)$ bits is a lower bound on the size of a cell, since the average size of the U_i is m/n.

To compute the information needed in the hash table, we define a *quotient function* (for h_{perfect}) as a function $q : U \to \{0, \ldots, r-1\}$, $r \in \mathbf{N}$, which is 1-1 on each set U_i. Given such a function, let $T'[i] = q(T[i])$, and the following program is equivalent to the above:

function lookup'(x)
  return ($T'[h_{\text{perfect}}(x)] = q(x)$);
end.

Thus it suffices to use the hash table T' of $\lceil \log r \rceil$-bit entries. By the above discussion, we ideally have that r is close to m/n.

Although the FKS dictionary [7] is not precisely of the form "minimal perfect hash function + hash table," it is easy to modify it to be of this type. We will thus speak of the FKS minimal perfect hash function, h_{FKS}. It has a quotient function which is evaluable in constant time, and costs no extra space in that its parameters k, p, and a are part of the data structure already:

$$q : u \mapsto (u \operatorname{div} p)\, \lceil p/a \rceil + (k\, u \bmod p) \operatorname{div} a. \tag{2.1}$$

Intuitively, this function gives the information that is thrown away by the modulo applications of the scheme's top-level hash function:

$$h : u \mapsto (k\, u \bmod p) \bmod a, \tag{2.2}$$

where k and a are positive integers and $p > a$ is prime. We will not give a full proof that q is a quotient function of h_{FKS}, since our final result does not depend on this. However, the main part of the proof is a lemma that will be used later, showing that q is a quotient function for h.

LEMMA 2.1. *For $U_i = \{u \in U \mid h(u) = i\}$ where $i \in \{0, \ldots, a-1\}$, q is* 1-1 *on U_i. Further, $q[U] \subseteq \{0, \ldots, r-1\}$, where $r = \lceil m/p \rceil \lceil p/a \rceil$.*

Proof. Let $u_1, u_2 \in U_i$. If $q(u_1) = q(u_2)$ we have that $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ and $(k\, u_1 \bmod p) \operatorname{div} a = (k\, u_2 \bmod p) \operatorname{div} a$. From the latter equation and $h(u_1) = h(u_2)$, it follows that $k\, u_1 \bmod p = k\, u_2 \bmod p$. Since p is prime and $k \neq 0$ this implies $u_1 \bmod p = u_2 \bmod p$. Since also $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ it must be the case that $u_1 = u_2$, so q is indeed 1-1 on U_i. The bound on the range of q is straightforward. □

In the FKS scheme, $p = \Theta(m)$ and $a = n$, so the range of q has size $O(m/n)$ and $\log(m/n) + O(1)$ bits suffice to store each hash table element. The space needed to store h_{FKS}, as described in [7], is not good enough to show the result claimed at the beginning of this section. However, Schmidt and Siegel [15] have shown how to implement (essentially) h_{FKS} using $O(n + \log\log m)$ bits of storage (which is optimal up to a constant factor; see, e.g., [10, Theorem III.2.3.6]). The time needed to evaluate the hash function remains constant. Their top-level hash function is not of the form (2.2), but the composition of two functions of this kind, h_1 and h_2. Call the corresponding quotient functions q_1 and q_2. A quotient function for $h_2 \circ h_1$ is $u \mapsto (q_1(u), q_2(h_1(u)))$, which has a range of size $O(m/n)$. One can thus get a space usage of $n\log(m/n) + O(n)$ bits for the hash table, and $O(n + \log\log m)$ bits for the hash function, so by (1.1) we have the following proposition.

PROPOSITION 2.2. *The static dictionary problem with worst case constant query time can be solved using $B + O(n + \log\log m)$ bits of storage.*

As a by-product we get the following corollary.

COROLLARY 2.3. *When $n > c\log\log m/\log\log\log m$, for a suitable constant $c > 0$, the static dictionary problem with worst case constant query time can be solved using n words of $\lceil \log m \rceil$ bits.*

Proof. The dictionary of Proposition 2.2 uses $n\log m - n\log n + O(n + \log\log m)$ bits. For suitable constants c and N, the $O(n + \log\log m)$ term is less than $n\log n$ when $n > N$. If $n \le N$ we can simply list the elements of S. □

The previously best result of this kind needed $n \ge (\log m)^c$ for some constant $c > 0$ [5]. (An interesting feature of this nonconstructive scheme is that it is *implicit*, i.e., the n words contain a permutation of the elements in S.) The question whether n words suffice in all cases was posed in [6].

3. Overview of final construction. This section describes the ideas which allow us to improve the $O(n)$ term of Proposition 2.2 to $o(n)$. There are two redundancy bottlenecks in the construction of the previous section:

- The Schmidt–Siegel hash function is $\Omega(n)$ bit redundant.
- The hash table is $\Omega(n)$ bit redundant.

The first bottleneck seems inherent to the Schmidt–Siegel scheme: it appears there is no easy way of improving the space usage to $1 + o(1)$ times the minimum, at least if constant evaluation time is to be preserved. The second bottleneck is due to the fact that m/n may not be close to a power of two, and hence the space consumption of $n\lceil \log r \rceil$ bits, where $r \ge m/n$, may be $\Omega(n)$ bits larger than the ideal of $n\log(m/n)$ bits. Our way around these bottlenecks starts with the following observations:

- We need only to solve the dictionary problem for some "large" subset $S_1 \subseteq S$.
- We can look at some universe $U_1 \subseteq U$, where $S_1 \subseteq U_1$ and $|U_1|/|S_1|$ is "close to" a power of 2.

The first observation helps by allowing "less than perfect" hash functions which occupy much less memory. The remaining elements, $S_2 = S \backslash S_1$, can be put in a more redundant dictionary, namely the refined FKS dictionary [7]. The second observation gives a way of minimizing redundancy in the hash table.

We will use a hash function of the form (2.2). The following result from [7] shows that (unless a is not much larger than n) it is possible to choose k such that h is 1-1 on a "large" set S_1.

LEMMA 3.1. *If $u \mapsto u \bmod p$ is 1-1 on S, then for at least half the values of $k \in \{1, \ldots, p-1\}$, there exists a set $S_1 \subseteq S$ of size $|S_1| \ge (1 - O(n/a))\,|S|$, on which h is 1-1.*

Without loss of generality we will assume that $|S_1| = n_1$, where n_1 only depends on n and a, and $n_2 = n - n_1 = O(n/a)$. The hash function h is not immediately useful, since it has a range of size $a \gg n_1$. To obtain a minimal perfect hash function for S_1, we compose with a function $g : \{0, \ldots, a-1\} \to \{0, \ldots, n_1 - 1\} \cup \{\perp\}$ which has $g[h[S_1]] = \{0, \ldots, n_1 - 1\}$. (In particular, it is 1-1 on $h[S_1]$.) The extra value of g allows us to look at $U_1 = \{u \in U \mid g(h(u)) \neq \perp\}$, which clearly contains S_1. We require that $g(v) = \perp$ when $v \notin h[S_1]$, since this makes q a quotient function for $g \circ h$ (restricted to inputs in U_1).

The implementation of the function g has the form of a dictionary for $h[S_1]$ within $\{0, \ldots, a-1\}$, which apart from membership queries answers *rank queries*. (The result of a rank query on input v is $|\{w \in h[S_1] \mid w < v\}|$.) So we may take g as the function that returns $\perp$ if its input v is not in the set, and otherwise returns the rank of v. The details on how to implement the required dictionary are given in section 4.

The dictionary of section 4 will also be our dictionary of choice when $m \leq n \log^3 n$ (the "dense" case). Only when $m > n \log^3 n$ do we use the scheme described in this section. This allows us to choose suitable values of hash function parameters p and a (where $p = O(n^2 \log m)$ and $a = \Theta(n (\log n)^2) \leq m$), such that the range of the quotient function, $r = \lceil m/p \rceil \lceil p/a \rceil$, is close to m/a. The details of this, along with an analysis of the redundancy of the resulting dictionary, can be found in section 5.

4. Dictionaries for dense subsets. In this section we describe a dictionary which has the redundancy stated in Theorem 1.1 when $m = n (\log n)^{O(1)}$. Apart from membership queries, it will support queries on the ranks of elements (the rank of u is $|\{v \in S | v < u\}|$), as well as queries on predecessors (the predecessor of u is $\max\{v \in S \mid v < u\}$).

As a first step, we describe a dictionary which has redundancy dependent on m, namely $O(m \log\log m / \log m)$ bits. The final dictionary uses the first one as a substructure.

4.1. Block compression. The initial idea is to split the universe into blocks $U_i = \{b\,i, \ldots, b\,(i+1) - 1\}$ of size $b = \lceil \frac{1}{2} \log m \rceil$, and store each block in a compressed form. (This is similar to the ideas of range reduction and "a table of small ranges" used in [3].) To simplify things we may assume that b divides m (otherwise consider a universe at most $b - 1$ elements larger, increasing the space usage by $O(b)$ bits). If a block contains j elements from S, the compressed representation is the number j ($\lfloor \log\log m \rfloor$ bits) followed by a number in $\{1, \ldots, \binom{b}{j}\}$ corresponding to the particular subset with i elements ($\lceil \log \binom{b}{j} \rceil$ bits). Extraction of information from a compressed block is easy, since any function of the block representations can be computed by table lookup. (The crucial thing being that, since representations have size at most $\frac{1}{2} \log m + \log\log m$ bits, the number of entries in such a table makes its space consumption negligible compared to $O(m \log\log m / \log m)$ bits.)[2]

Let $n_i = |S \cap U_i|$ and $B_i = \lceil \log \binom{b}{n_i} \rceil$. The overall space consumption of the above encoding is $\sum_i B_i + O(m \log\log m / \log m)$. Let s denote the number of blocks, $s = O(m / \log m)$. A lemma from [2] bounds the above sum by $B + s$.

LEMMA 4.1 (Brodnik–Munro). *Let $m_0, \ldots, m_{s-1}$ and $n_0, \ldots, n_{s-1}$ be nonnegative integers. The following inequality holds:*

$$\sum_{i=0}^{s-1} \lceil \log \tbinom{m_i}{n_i} \rceil < \log \binom{\sum_{i=0}^{s-1} m_i}{\sum_{i=0}^{s-1} n_i} + s.$$

[2] Alternatively, assume that the RAM has instructions to extract the desired information.

Proof. We have $\sum_{i=0}^{s-1}\lceil\log\binom{m_i}{n_i}\rceil < \sum_{i=0}^{s-1}\log\binom{m_i}{n_i} + s \le \log\binom{\sum_{i=0}^{s-1} m_i}{\sum_{i=0}^{s-1} n_i} + s$. The latter inequality follows from the fact that there are at least $\prod_{i=0}^{s-1}\binom{m_i}{n_i}$ ways of picking $\sum_{i=0}^{s-1} n_i$ elements out of $\sum_{i=0}^{s-1} m_i$ elements (namely, by picking n_1 among the m_1 first, n_2 among the m_2 next, etc.). □

We need an efficient mechanism for extracting rank and predecessor information from the compressed representation. In particular we need a way of finding the start of the compressed representation of the ith block. The following result, generalizing a construction in [16], is used.

PROPOSITION 4.2 (Tarjan–Yao). *A sequence of integers $z_1, \ldots, z_k$, where for all $1 < i \le k$ we have $|z_i| = n^{O(1)}$ and $\max(|z_i|, |z_i - z_{i-1}|) = (\log n)^{O(1)}$, can be stored in a data structure allowing constant time random access, using $O(k \log\log n)$ bits of memory.*

Proof. Every $\lceil\log n\rceil$th integer is stored "verbatim," using a total of $O(k)$ bits. All other integers are stored as either an offset relative to the previous of these values, or as an absolute value. (One of these has size $(\log n)^{O(1)}$.) This uses $O(k \log\log n)$ bits in total. □

Placing the compressed blocks consecutively in numerical order, the sequence of pointers to the compressed blocks can be stored by this method. Also, the rank of the first element in each block can be stored like this. Finally, we may store the distance to the predecessor of the first element in each block (from which the predecessor is simple to compute). All of these data structures use $O(m \log\log m/\log m)$ bits. Ranks and predecessors of elements within a block can be found by table lookup, as sketched above. So we have the following proposition.

PROPOSITION 4.3. *A static dictionary with worst case constant query time, supporting rank and predecessor queries, can be stored in $B + O(m \log\log m/\log m)$ bits.*

4.2. Interval compression. The dictionary of section 4.1 has the drawback that the number of compressed blocks, and hence the redundancy, grows almost linearly with m. For $m \le n\,(\log n)^c$, where c is any integer constant, the number of "compressed units" can be reduced to $O(n \log\log n/\log n)$ by instead compressing *intervals* of varying length. We make sure that the compressed representations have length $(1 - \Omega(1))\log n$ (so that information can be extracted by lookup in a table of negligible size) by using intervals of size $O((\log n)^{c+1})$ with at most $\log n/(2c \log\log n)$ elements. We must be able to retrieve the interval number and position within the interval for any element of U in constant time. The block compression scheme of section 4.1 is then trivially modified to work with intervals, and the space for auxiliary data structures becomes $O(n\,(\log\log n)^2/\log n)$ bits.

We now proceed to describe the way in which intervals are formed and represented. Let $d = \lfloor\sqrt{\log n}\rfloor$, and suppose without loss of generality that d^{2c} divides m. (Considering a universe at most d^{2c} elements larger costs $O(d^{2c})$ bits, which is negligible.) Our first step is to partition U into "small blocks" U_i, satisfying $|S \cap U_i| \le \log n/(2c \log\log n)$. These will later be clustered to form the intervals. The main tool is the dictionary of Proposition 4.3, which is used to locate areas with a high concentration of elements from S. More specifically, split U into at most n blocks of size d^{2c} and store the indices of blocks that are *not* small, i.e., contain more than $\log n/(2c \log\log n)$ elements from S. Since at most $2cn \log\log n/\log n$ blocks are not small, the memory for this data structure is $O(n\,(\log\log n)^2/\log n)$ bits. The part of the universe contained in nonsmall blocks has size at most $2cm \log\log n/\log n \le n\,d^{2c-1}$. A rank query can be used to map the elements of nonsmall blocks injectively

and in an order preserving way to a subuniverse of this size. The splitting is repeated recursively on this subuniverse, now with at most n blocks of size d^{2c-1}. Again, the auxiliary data structure uses $O(n\,(\log\log n)^2/\log n)$ bits. At the bottom of the recursion we arrive at a universe of size at most nd, and every block of size d is small. This defines our partition of U into $O(n)$ small blocks, which we number $0, 1, 2, \ldots$ in order of increasing elements. Note that the small block number of any element in U can be computed by a rank query and a predecessor query at each level.

As every small block has size at most $(\log n)^c$, intervals can be formed by up to $\log n$ consecutive small blocks, together containing at most $\log n/(2c\log\log n)$ elements of S. The "greedy" way of choosing such compressible intervals from left to right results in $O(n\log\log n/\log n)$ intervals, as no two adjacent intervals can both contain less than $\log n/(4c\log\log n)$ elements and be shorter than $(\log n)^{c+1}$. To map the $O(n)$ block numbers to interval numbers, we use the dictionary of Proposition 4.3 to store the number of the first small block in each interval, using $O(n\,(\log\log n)^2/\log n)$ bits. A rank query on a small block number then determines the interval number. Finally, the first element of each interval is stored using Proposition 4.2, allowing positions within intervals to be computed, once again using $O(n\,(\log\log n)^2/\log n)$ bits.

THEOREM 4.4. *For $m = n\,(\log n)^{O(1)}$, a static dictionary with worst case constant query time, supporting rank and predecessor queries, can be stored in $B + O(n\,(\log\log n)^2/\log n)$ bits.*

5. Dictionary for sparse subsets. In this section we fill out the remaining details of the construction described in section 3, and provide an analysis of the redundancy obtained. By section 4 we need only consider the case $m > n\,(\log n)^c$ for some constant c. (It will turn out that $c = 3$ suffices.)

5.1. Choice of parameters. We need to specify how hash function parameters a and p are chosen. (A choice of k then follows by Lemma 3.1.) Parameter a will depend on p, but is bounded from above by $A(n)$ and from below by $A(n)/3$, where A is a function we specify later. For now, let us just say that $A(n) = n\,(\log n)^{\Theta(1)}$. (Our construction requires $A(n) = n\,(\log n)^{O(1)}$, and we want $A(n)$ large in order to make $S\backslash S_1$ small.) Parameter p will have size $O(n^2\log m)$, so it can be stored using $O(\log n + \log\log m)$ bits. It is chosen such that $u \mapsto u \bmod p$ is 1-1 on S, and such that $r = \lceil m/p\rceil\,\lceil p/a\rceil$ is not much larger than m/a.

LEMMA 5.1. *For m larger than some constant, there exists a prime p in each of the following ranges, such that $u \mapsto u \bmod p$ is 1-1 on S:*

1. $n^2\ln m \le p \le 3\,n^2\ln m$.
2. $m \le p \le m + m^{2/3}$.

Proof. The existence of a suitable prime between $n^2\ln m$ and $3\,n^2\ln m$ is guaranteed by the prime number theorem (in fact, at least half of the primes in the interval will work). See [7, Lemma 2] for details. By [9] the number of primes between m and $m + m^\theta$ is $\Omega(m^\theta/\log m)$ for any $\theta > 11/20$. Take $\theta = 2/3$ and let p be such a prime; naturally the map is then 1-1. □

A prime in the first range will be our choice for p when $m > A(n)\,n^2\ln m$; otherwise we choose a prime in the second range. In the first case, $r < (m/p+1)(p/a+1) = (1 + a/p + p/m + a/m)\,m/a$. In the second case, $r = \lceil p/a\rceil < (m + m^{2/3})/a + 1 \le (1 + a/m + m^{-1/3})\,m/a$. Since we can assume $m > a\log n$, we have in both cases that $r = (1 + O(1/\log n))\,m/a$. We make r close to a power of 2 by suitable choice of parameter a.

LEMMA 5.2. *For any $x, y \in \mathbf{R}_+$ and $z \in \mathbf{N}$, with $x/z \geq 3$, there exists $z' \in \{z+1, \ldots, 3z\}$, such that $\lceil \log \lceil x/z' \rceil + y \rceil \leq \log(x/z') + y + O(z/x + 1/z)$.*

Proof. Since $x/z \geq 3$, it follows that $\log\lceil x/z \rceil + y$ and $\log\lceil x/3z \rceil + y$ have different integer parts. So there exists z', $z < z' \leq 3z$, such that $\lceil \log \lceil x/z' \rceil + y \rceil \leq \log\lceil x/(z'-1) \rceil + y$. A simple calculation gives $\log\lceil x/(z'-1) \rceil + y = \log(x/(z'-1)) + y + O(z/x) = \log(x/z') + \log(z'/(z'-1)) + y + O(z/x) = \log(x/z') + y + O(z/x + 1/z)$, and the conclusion follows. □

Since $\log r = \log\lceil p/a \rceil + \log\lceil m/p \rceil$ and $p/A(n) \geq 3$ (for n large enough), the lemma gives an a satisfying $A(n)/3 \leq a \leq A(n)$, such that $\lceil \log r \rceil = \log r + O(a/p + 1/a) = \log((1 + O(1/\log n))\, m/a)$.

To conclude, we can choose p and a such that the number of bit patterns in each hash table cell, $2^{\lceil \log r \rceil}$, is $(1 + O(1/\log n))\, m/a$.

5.2. Storing parameters. A slightly technical point remains, concerning the storage of parameters in the data structure. If the universe size m is supposed to be implicitly known, there is no problem storing the parameters using $O(\log n + \log\log m)$ bits (say, using $O(\log\log m)$ bits to specify the number of bits for each parameter). However, if m is considered a parameter unknown to the query algorithm, it is not clear how to deal with, e.g., queries for numbers larger than m, without actually using $O(\log m)$ extra bits to store m. Our solution is to look at a slightly larger universe U', whose size is specified using $O(\log n + \log\log m)$ bits. Using $O(\log n + \log\log m)$ bits we may store $\lceil \log m \rceil$ (by assumption we know the number of bits needed to store this number within an additive constant) and the $\lceil \log n \rceil$ most significant bits of m. This defines $m' = (1 + O(1/n))\, m$, the universe size of U'. We need to estimate the information theoretical minimum of the new problem, $B' = \lceil \binom{m'}{n} \rceil$.

LEMMA 5.3. *For $n < m_1 < m_2$ we have $\log \binom{m_2}{n} - \log \binom{m_1}{n} < n \log(\frac{m_2 - n}{m_1 - n})$.*

Proof. We have $\binom{m_2}{n} / \binom{m_1}{n} = \frac{m_2\,(m_2-1)\ldots(m_2-n+1)}{m_1\,(m_1-1)\ldots(m_1-n+1)} < (\frac{m_2-n}{m_1-n})^n$. □

Thus, since $n \leq m/2$, $B' = B + O(n \log(m'/m)) = B + O(n/\log n)$. So our slight expansion of the universe is done without affecting the redundancy of Theorem 1.1.

5.3. Redundancy analysis. First note that we can assume all parts of the data structure to have size depending only on m and n (that is, not on the particular set stored). Hence, the entire data structure is a bit pattern of size $B + f(n, m)$, for some function f. To show the bound $f(n, m) = O(n\,(\log\log n)^2/\log n + \log\log m)$, we construct a function ϕ, mapping n-element subsets of U to subsets of $\{0,1\}^{B+f(n,m)}$, such that

- $\log|\phi(S)| = O(n\,(\log\log n)^2/\log n + \log\log m)$.
- $\bigcup_S \phi(S) = \{0,1\}^{B+f(n,m)}$.

This implies $B + f(n, m) \leq \log(\sum_S |\phi(S)|) = B + O(n\,(\log\log n)^2/\log n + \log\log m)$, as desired. Recall that the data structure consists of

- hash function parameters and pointers ($b_1 = O(\log n + \log\log m)$ bits);
- a dictionary supporting rank, representing the function g via a set of n_1 elements in $\{0, \ldots, a-1\}$ ($b_2 = \log \binom{a}{n_1} + O(n\,(\log\log n)^2/\log n)$ bits);
- a hash table ($b_3 = n_1 \lceil \log r \rceil$ bits);
- a dictionary representing a set of size n_2 in U ($b_4 = \log \binom{m}{n_2} + O(n_2 \log n_2 + \log\log m)$ bits).

Since the dictionary supporting rank has redundancy $O(n\,(\log\log n)^2/\log n)$, there exists a function ϕ' from the n_1-element subsets of $\{0, \ldots, a-1\}$ to $\{0,1\}^{b_2}$, such that $\cup_{\tilde{S}_1} \phi'(\tilde{S}_1) = \{0,1\}^{b_2}$ and $\log|\phi'(\tilde{S}_1)| = O(n\,(\log\log n)^2/\log n)$. Similarly, there exists a function ϕ'' from the n_2-element subsets of U to $\{0,1\}^{b_4}$, such that $\cup_{\tilde{S}_2} \phi''(\tilde{S}_2) =$

$\{0,1\}^{b_4}$ and $\log|\phi''(\tilde{S}_2)| = O(n_2 \log n_2 + \log\log m)$. Choosing $A(n) = n \log^2 n$ we have $n_2 = O(n/\log^2 n)$, and hence $\log|\phi''(\tilde{S}_2)| = O(n/\log n + \log\log m)$.

Let h_1 denote the hash function $u \mapsto (u \bmod p) \bmod a$. (Any hash function of the form (2.2) would do; we pick this one for simplicity.) By Lemma 5.3 we can assume that p divides m, since either $m \le p \le m+m^{2/3}$, in which case expanding the universe to $\{0,\ldots,p-1\}$ increases the information theoretical minimum by $O(n/m^{1/3})$, or $p = O(m/n)$, in which case the increase by rounding m to the nearest higher multiple of p is $O(1)$.

When p divides m, the number of elements hashed to a cell by h_1 is at least $\lfloor m/a \rfloor$. Hence for any function g, there is a set $T(g)$ of at least $(m/a-1)^{n_1}$ possible bit patterns in the hash table. The total number of bit patterns is $2^{\lceil \log r \rceil\, n_1} = ((1+O(1/\log n))\, m/a)^{n_1}$, so the ratio between this and the $|T(g)|$ patterns used is

$$\left(\tfrac{1+O(1/\log n)}{1-a/m}\right)^{n_1} = (1+O(1/\log n))^{n_1} = 2^{O(n/\log n)}.$$

Thus, there exists a function ϕ_g from $T(g)$ onto $\{0,1\}^{b_3}$, such that $\log|\phi_g(z)| = O(n/\log n)$. For notational convenience we will from now on denote bit patterns in the hash table simply by the corresponding set of universe elements.

We will take $\phi(S)$ as the union of sets $\phi(\tilde{S}_1, \tilde{S}_2)$, over all $\tilde{S}_1, \tilde{S}_2 \subseteq S$ with $|\tilde{S}_1| = n_1$ and $|\tilde{S}_2| = n_2$. If $|h_1[\tilde{S}_1]| \ne n_1$, we set $\phi(\tilde{S}_1, \tilde{S}_2) = \emptyset$. Otherwise $h_1[\tilde{S}_1]$ defines the function g, and we set

$$\phi(\tilde{S}_1, \tilde{S}_2) = \{s_1\, s_2\, s_3\, s_4 \mid s_1 \in \{0,1\}^{b_1},\ s_2 \in \phi'(h_1[\tilde{S}_1]),\ s_3 \in \phi_g(\tilde{S}_1),\ s_4 \in \phi''(\tilde{S}_2)\}\ .$$

By our bounds on the sizes of $\phi'(h_1[\tilde{S}_1])$, $\phi_{h_1,g}(\tilde{S}_1)$, and $\phi''(\tilde{S}_2)$, we conclude that $\log|\phi(\tilde{S}_1, \tilde{S}_2)| = O(n\,(\log\log n)^2/\log n + \log\log m)$. Since $\phi(S)$ is the union of the $2^{O(n/\log n)}$ sets of the form $\phi(\tilde{S}_1, \tilde{S}_2)$, it follows that the requirement on $|\phi(S)|$ holds.

To see that $\bigcup_S \phi(S) = \{0,1\}^{B+f(n,m)}$, take any $x \in \{0,1\}^{B+f(n,m)}$. Let $x = s_1\, s_2\, s_3\, s_4$, where $s_i \in \{0,1\}^{b_i}$. By definition of ϕ' and ϕ'', there is some set $T \subseteq \{0,\ldots,a-1\}$, $|T| = n_1$, such that $s_2 \in \phi'(T)$, and a set $\tilde{S}_2 \subseteq U$, $|\tilde{S}_2| = n_2$, such that $s_4 \in \phi''(S_2)$. The set T corresponds to a function g. From the way we defined ϕ_g, there exists a bit pattern $z \in T(g)$, such that $s_3 \in \phi_g(z)$. Let $\tilde{S}_1$ be the set of size n_1 corresponding to h_1, g, and z. We then have that $x \in \phi(\tilde{S}_1, \tilde{S}_2)$, so if we take $S \supseteq \tilde{S}_1 \cup \tilde{S}_2$, we get $x \in \phi(S)$ as desired.

6. Satellite information. We now discuss how to associate information with dictionary elements. More specifically, we consider the setting where each element of S has an associated piece of satellite information from some set $V = \{0,\ldots,s-1\}$, where $s = m^{O(1)}$. The information theoretical minimum for this problem is $B_s = \lceil \log \binom{m}{n} + n \log s \rceil$.

The quotienting technique generalizes to this setting. We simply extend the quotient function to take an extra parameter from V as follows: $q'(u,v) = q(u) + r\,v$. Note that from $q'(u,v)$ it is easy to compute $q(u)$ and v and that the range of q' has size $r\,s$. With this new quotient function, the remaining parts of the construction for $m > n\,(\log n)^3$ are unchanged.

In the dense range, the rank dictionary can be used to index into a table of V-values, but in general $\Omega(n)$ bits will be wasted in the table since $|V|$ need not be a power of 2. Thus we have the following theorem.

THEOREM 6.1. *The static dictionary problem with satellite data and worst case constant query time can be solved with storage:*

- $B_s + O(n\,(\log\log n)^2/\log n + \log\log m)$ *bits, for* $m > n\,(\log n)^3$.
- $B_s + O(n)$ *bits, otherwise.*

Using the data structure for the sparse case, it is in fact possible to achieve redundancy $o(n)$ when $n = o(m)$. The dictionary of Proposition 2.2 is then used to store S_2, and parameter a is chosen around $\sqrt{nm}$.

7. Construction. We now sketch how to construct the static dictionaries in expected time $O(n + (\log\log m)^{O(1)})$. The dictionaries of section 4 can in fact be constructed in time $O(n)$ when $m = n^{O(1)}$. The construction algorithm is quite straightforward, so we do not describe it here. As for the dictionary described in sections 3 and 5, the hardest part is finding appropriate parameters for the hash function. Once this is done, the dictionary for $h[S_1]$, the hash table, and the dictionary for S_2 can all be constructed in expected time $O(n)$ (see [7] for the latter construction algorithm).

The prime p is found by randomly choosing numbers from the appropriate interval of Lemma 5.1. Each number chosen is checked for primality (using a probabilistic check which uses expected time polylogarithmic in the number checked [1], that is, time $(\log n + \log\log m)^{O(1)}$). When a prime is found, it is checked whether $u \mapsto u \bmod p$ is 1-1 on S (the element distinctness problem on the residues, taking expected $O(n)$ time using universal hashing). The process is repeated until this is the case. Inspecting the proof of Lemma 5.1 it can be seen that the expected number of iterations is $O(1)$, so the expected total time is $O(n + (\log\log m)^{O(1)})$.

Parameter a is simple to compute according to Lemma 5.2, for example by binary search on the interval in which it is wanted.

Parameter k is tentatively chosen at random and checked in time $O(n)$ for the inequality of Lemma 3.1, with some constant c in the big-oh. For sufficiently large c, the expected number of attempts made before finding a suitable k is constant, and thus the expected time for the choice is $O(n)$.

THEOREM 7.1. *The data structure of Theorem* 1.1 *can be constructed in expected time* $O(n + (\log\log m)^{O(1)})$.

8. Conclusion and final remarks. We have seen that for the static dictionary problem it is possible to come very close to using storage at the information theoretic minimum, while retaining constant query time. From a data compression point of view this means that a sequence of bits can be coded in a number of bits close to the first-order entropy, in a way that allows efficient random access to the original bits.

The important ingredient in the solution is the concept of quotienting. Quotienting was recently applied in a space efficient dictionary supporting rank [14]. In general, quotienting can be used to save around $n \log n$ bits in hash tables. Thus, the existence of an efficiently evaluable quotient function is a desirable property for a hash function. For the quotient function to have a small range, it is necessary that the hash function used hashes U quite evenly to the entire range.

Quotienting works equally well in a dynamic setting, where it can be used directly to obtain an $O(B)$ bit scheme, equaling the result of Brodnik and Munro [2]. However, lower bounds on the time for maintaining ranks under insertions and deletions (see [8]) show that our construction involving the dictionary supporting rank will not dynamize well.

It would be interesting to determine the exact redundancy necessary to allow constant time queries. In particular, it is remarkable that no lower bound is known in the *cell probe* model (where only the number of memory cells accessed is considered). As for upper bounds, a less redundant implementation of the function g would

immediately improve the asymptotic redundancy of our scheme. There seems to be no hope of getting rid of the $O(\log \log m)$ term using our basic approach, since any hash function family ensuring that some function is 1-1 on a "large" subset of S has size $\Omega(\log m)$ [13].

Acknowledgments. The author would like to thank Peter Bro Miltersen for encouragement and advice. Thanks also to Jakob Pagter and Theis Rauhe for feedback on a previous version of this paper [12] and to Kunihiko Sadakane for pointing out an error in a draft of this paper.

REFERENCES

[1] L. M. Adleman and M.-D. Huang, *Recognizing primes in random polynomial time*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC '87), ACM Press, New York, 1987, pp. 462–469.

[2] A. Brodnik and J. I. Munro, *Membership in constant time and minimum space*, in Proceedings of the 2nd European Symposium on Algorithms (ESA '94), Lecture Notes in Comput. Sci. 855, Springer-Verlag, Berlin, 1994, pp. 72–81.

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

[4] 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), ACM Press, New York, 1978, pp. 59–65.

[5] A. Fiat and M. Naor, *Implicit $O(1)$ probe search*, SIAM J. Comput., 22 (1993), pp. 1–10.

[6] F. Fich and P. B. Miltersen, *Tables should be sorted (on random access machines)*, in Proceedings of the the 4th Workshop on Algorithms and Data Structures (WADS '95), Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 482–493.

[7] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538–544.

[8] M. L. Fredman and M. E. Saks, *The cell probe complexity of dynamic data structures*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89), ACM Press, New York, 1989, pp. 345–354.

[9] D. R. Heath-Brown and H. Iwaniec, *On the difference between consecutive primes*, Invent. Math., 55 (1979), pp. 49–69.

[10] K. Mehlhorn, *Data Structures and Algorithms.* 1*, Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[11] P. B. Miltersen, *Lower bounds for static dictionaries on RAMs with bit operations but no multiplication*, in Proceedings of the 23rd International Colloquium on Automata, Languages and Programming (ICALP '96), Lecture Notes in Comput. Sci. 1099, Springer-Verlag, Berlin, 1996, pp. 442–453.

[12] R. Pagh, *Low redundancy in static dictionaries with $O(1)$ lookup time*, in Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99), Lecture Notes in Comput. Sci. 1644, Springer-Verlag, Berlin, 1999, pp. 595–604.

[13] R. Pagh, *Dispersing hash functions*, in Proceedings of the 4th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '00), Proceedings in Informatics 8, Carleton Scientific, Waterloo, ON, Canada, 2000, pp. 53–67.

[14] V. Raman and S. S. Rao, *Static dictionaries supporting rank*, in Proceedings of the 10th International Symposium on Algorithms And Computation (ISAAC '99), Lecture Notes in Comput. Sci. 1741, Springer-Verlag, Berlin, 1999, pp. 18–26.

[15] J. P. Schmidt and A. Siegel, *The spatial complexity of oblivious k-probe hash functions*, SIAM J. Comput., 19 (1990), pp. 775–786.

[16] R. E. Tarjan and A. C.-C. Yao, *Storing a sparse table*, Communications of the ACM, 22 (1979), pp. 606–611.

[17] A. C.-C. Yao, *Should tables be sorted?*, J. Assoc. Comput. Mach., 28 (1981), pp. 615–628.