

Low Redundancy in Static Dictionaries with $O(1)$ Worst Case Lookup Time

Rasmus Pagh¹

BRICS², Department of Computer Science, University of Aarhus,
8000 Aarhus C, Denmark
Email: pagh@brics.dk

Abstract. A *static dictionary* is a data structure for storing subsets of a finite universe U , so that membership queries can be answered efficiently. We study this problem in a unit cost RAM model with word size $\Omega(\log |U|)$, and show that for n -element subsets, 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 such subsets. For $|U| = n \log^{O(1)} n$ the dictionary supports constant time rank queries.

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. Since we are interested only in membership queries, we assume that $U = \{0, \dots, m - 1\}$. We restrict the attention to the case where elements of U can be represented within a constant number of machine words. In particular it is assumed that 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 data structure problem using little memory, measured in consecutive bits (A part of the last word may be unused, and the query algorithm must work regardless of the contents of this part). 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). Using Stirling’s approximation to the factorial function, one can get (see [2]):

$$B = n \log \frac{m}{n} + (m - n) \log \frac{m}{m - n} - O \left(\log \frac{n(m - n)}{m} \right). \quad (1)$$

¹ Supported in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT)

² Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

For $n = o(m)$ the dominant term is $n \log \frac{m}{n}$, since $(m - n) \log \frac{m}{m-n} = \Theta(n)$ (see Lemma 8). 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 it has been heavily studied. We will focus on the development in space consumption for worst case constant time lookup schemes. A bit vector is the simplest possible solution to the problem, but the space complexity of m bits is poor compared to B unless $n \approx m/2$. During the 70's, schemes were suggested which obtain a space complexity of $O(n)$ words, that is $O(n \log m)$ bits, for restricted cases (e.g. “dense” or very sparse sets). It was not until the early 80's that Fredman, Komlós and Szemerédi (FKS) [5] found a constant time hashing scheme using $O(n)$ words in the general case. A refined solution in their paper uses $B + O(n \log n + \log \log m)$ bits. Brodnik and Munro [2] construct a static dictionary using $O(B)$ bits for any m and n . In the journal version of this paper [3], they achieve $B + O(\frac{B}{\log \log \log m})$ bits.

No lower bound better than the trivial B bits is known without restrictions on the data structure or the query algorithm (see [4], [8] and [11]).

This Paper

The result of Brodnik and Munro is strengthened, bringing the additional term of the space complexity, which we shall call the *redundancy*, down to $o(n) + O(\log \log m)$ bits. The exact order of the bound, compared with a lower bound on the redundancy of the solution in [3], is given in the table below³.

Range	Brodnik & Munro	This paper
$n < m / \log \log m$	$\min(n \log \log m, \frac{m}{(\log \log m)^{O(\log \log \log m)}})$	$n \frac{\log^2 \log n}{\log n} + \log \log m$
$n \geq m / \log \log m$		$m \frac{\log \log m}{\log m}$

We also show how to associate information from some domain to each element of S (solving the *partial function* problem), with the same redundancy as above in the sparse case, and $O(n)$ bits in the dense case.

The 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 the element itself, but only a *quotient* — information that distinguishes it from the part of U that hashes to this cell. This technique, referred to as *quotienting*, is described in Sect. 2, where a $B + O(n + \log \log m)$ bit scheme is presented.

³ The bounds given are asymmetric in the sense that if S is replaced by $U \setminus S$ we get another bound although the problems obviously have the same difficulty. However, for simplicity we focus on $n \leq m/2$ and leave the symmetry implications to the reader.

For dense subsets another technique is used, building upon the ideas of range reduction and a “table of small ranges” (both used in [3]). This results in a dictionary, treated in Sect. 3, that supports rank queries. The rank query capability is subsequently used in an improved solution for the non-dense case, described in Sect. 4.

2 First solution

This section presents a static dictionary with a space consumption of $B + O(n + \log \log m)$ bits. As mentioned in the overview, the compact representation achieved stems from the observation that each bucket j of a hash table may be resolved with respect to the part of the universe hashing to bucket j , which we denote by A_j . We phrase this in terms of injective functions on the A_j . Consider the lookup procedure of a dictionary using a perfect hash function \mathbf{h} , and a table \mathbf{T} :

```

proc lookup(x)
  return (T[h(x)]=x);
end

```

If \mathbf{q} is a function which is 1-1 on each A_j (we call this a *quotient function*), and we let $\mathbf{T}'[j] := \mathbf{q}(\mathbf{T}[j])$, then the following program is equivalent:

```

proc lookup'(x)
  return (T'[h(x)]=q(x));
end

```

Thus, given a description of \mathbf{q} , it suffices to use the hash table \mathbf{T}' . The gain is that \mathbf{q} may have a range significantly smaller than U (ideally \mathbf{q} would enumerate the elements of each A_j), and thus fewer bits are needed to store the elements of \mathbf{T}' .

The FKS perfect hashing scheme [5] 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:

$$q_{k,p} : u \mapsto (u \operatorname{div} p) \cdot \lceil p/a \rceil + (k \cdot u \operatorname{mod} p) \operatorname{div} a . \quad (2)$$

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

$$h_{k,p} : u \mapsto (k \cdot u \operatorname{mod} p) \operatorname{mod} a \quad (3)$$

where p is prime and k , a positive integers. (So in fact $q_{k,p}$ is 1-1 even on the elements hashing to each bucket in the top level hash table). Since $p = O(m)$ and $a = n$ in the FKS scheme, the range of $q_{k,p}$ has size $O(m/n)$, so $\log \frac{m}{n} + O(1)$ bits suffice to store each hash table element. We prove that $q_{k,p}$ is indeed a quotient function for $h_{k,p}$:

Lemma 1. Let $A_j(k, p) = \{u \in U \mid h_{k,p}(u) = j\}$ be the subset of U hashing to j . For any j , $q_{k,p}$ is 1-1 on $A_j(k, p)$. Furthermore, $q_{k,p}[U] \subseteq \{0, \dots, r-1\}$, where $r = \lceil m/p \rceil \cdot \lceil p/a \rceil$.

Proof. Let $u_1, u_2 \in A_j(k, p)$. If $q_{k,p}(u_1) = q_{k,p}(u_2)$ then $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ and $(k \cdot u_1 \operatorname{mod} p) \operatorname{div} a = (k \cdot u_2 \operatorname{mod} p) \operatorname{div} a$. By the latter equation and since $h_{k,p}(u_1) = h_{k,p}(u_2)$, we have $k \cdot u_1 \operatorname{mod} p = k \cdot u_2 \operatorname{mod} p$. Since p is prime and $k \neq 0$ this implies $u_1 \operatorname{mod} p = u_2 \operatorname{mod} p$. Since also $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ it must be the case that $u_1 = u_2$, so $q_{k,p}$ is indeed 1-1 on $A_j(k, p)$. The bound on the range of $q_{k,p}$ is straightforward. \square

Schmidt and Siegel [9] show how to simulate the FKS hashing scheme in a “minimal” version (i.e. the hash table has size n), using $O(n + \log \log m)$ bits of storage for the hash function, still with constant lookup time. Their top-level hash function is not (3), but the composition of two functions of this kind, h^1 and h^2 (with quotient functions q^1 and q^2). A corresponding quotient function 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 $n \log \frac{m}{n} + O(n)$ bits for the hash table elements, and $O(n + \log \log m)$ bits for the hash function, so using (1) we have:

Proposition 2. *The static dictionary problem with worst case constant lookup time can be solved using $B + O(n + \log \log m)$ bits of storage.*

3 Improvement for Dense Subsets

In this section we describe data structures which are more space efficient for dense subsets than that of the previous section. They will support queries on the ranks of elements (where the rank of u is defined as $\operatorname{rank}(u) = |\{v \in S \mid v \leq u\}|$). Using rank queries it is possible to do membership queries; therefore we will call the data structures presented *static rank dictionaries*.

Two solutions will be given. The first one has redundancy dependent on m , namely $O(\frac{m \log \log m}{\log m})$ bits. The second solution achieves redundancy $O(\frac{n \log^2 \log n}{\log n})$ bits, for $m = \log^{O(1)} n$.

3.1 Block Compression

The initial idea is to split the universe into blocks $U_i = \{b \cdot i, \dots, b \cdot (i+1) - 1\}$ of size $b = \lceil \frac{1}{2} \log m \rceil$, and store each block in a compressed form. If a block contains i elements from S , the compressed representation is the number i followed by a number in $\{1, \dots, \binom{b}{i}\}$ corresponding to the particular subset with i elements. 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 $\frac{m \log \log m}{\log m}$ bits). Alternatively, assume that the RAM has instructions to extract the desired information. 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 $\sum_i B_i + O(\frac{m \log \log m}{\log m})$. Let s denote the number of blocks, $s = O(m/\log m)$. A lemma from [2] bounds the above sum:

Lemma 3. *The following inequality holds: $\sum_{i=0}^{s-1} B_i < B + s$.*

Proof. We have $\sum_{i=0}^{s-1} B_i < \sum_{i=0}^{s-1} \log \binom{b}{n_i} + s \leq B + s$. The latter inequality follows from the fact that $\prod_{i=0}^{s-1} \binom{b}{n_i}$ is the number of sets having n_i elements in block i , which is a subset of all n -subsets in U . \square

We need an efficient mechanism for extracting rank information from the compressed representation. The following result contained in [10] is used:

Proposition 4. *A sequence of integers z_1, \dots, z_k where $|z_i| = n^{O(1)}$ and $|z_{i+1} - z_i| = \log^{O(1)} n$ can be stored in a data structure allowing constant time access, and using $O(k \log \log n)$ bits of memory.*

Proof. Every $\log n$ th integer is stored “verbatim”, using a total of $O(k)$ bits. All other integers are stored as offsets of size $\log^{O(1)} n$ relative to the previous of these values, using $O(k \log \log n)$ bits in total. \square

A sequence of pointers to the compressed blocks can be stored in this way, using $O(\frac{m \log \log m}{\log m})$ bits. Also, the rank of the first element in each block can be stored like this. Ranks of elements within a block can be found by table lookup, as sketched above. So we have:

Proposition 5. *A static rank dictionary with worst case constant query time can be represented using $B + O(\frac{m \log \log m}{\log m})$ bits.*

3.2 Interval Compression

Our first solution to the static rank dictionary problem has the drawback that the number of compressed blocks (and hence the redundancy) grows almost linearly with m . The number of compressed units can be reduced to $O(\frac{n \log \log n}{\log n})$, for $m = n \log^{O(1)} n$, by clustering suitable adjacent blocks together into *intervals* of varying length (bounded by $\log^{O(1)} m$). This will reduce the space needed to store the pointers to compressed units and the rank of the first element in each unit to $O(\frac{n \log^2 \log n}{\log n})$ bits. Since the lengths of intervals are not fixed, we store the starting positions of the intervals, using $O(\frac{n \log^2 \log n}{\log n})$ bits. Observing that Lemma 3 is independent of the sizes of the compressed units, this will show:

Theorem 6. *For $m = n \log^{O(1)} n$, a static rank dictionary with worst case constant query time can be represented using $B + O(\frac{n \log^2 \log n}{\log n})$ bits.*

Let $c \geq 1$ be a constant such that $m \leq \frac{n \log^c n}{2^c}$. We show how to partition U into “small blocks” U_i satisfying $|S \cap U_i| < \frac{\log n}{2^c \log \log n}$. These blocks will be the building stones of the intervals to be compressed. The main tool is the rank

dictionary of Prop. 5, which is used to locate areas with a high concentration of elements from S . More specifically, split U into $n \log \log n$ blocks and store the positions of the blocks which are *not* small, i.e. contain at least $\frac{\log n}{2c \log \log n}$ elements from S . Since at most $\frac{2cn \log \log n}{\log n}$ blocks are not small, the memory for this data structure is $O(\frac{n \log^2 \log n}{\log n})$. The part of the universe corresponding to the non-small blocks has size at most $\frac{n \log^{c-1} n}{2c}$. The splitting into $n \log \log n$ blocks is repeated for this sub-universe, and so on for $c - 1$ steps, until the entire sub-universe has size at most $\frac{n \log n}{2c}$, and hence all $n \log \log n$ blocks must be small. This defines our partition of U into small blocks. Given $u \in U$ it takes time $O(c)$ to use the rank dictionaries to find the associated block number (blocks are numbered “from left to right”, i.e. according to the elements they contain).

Note that every small block has size at most $\log^c n / \log \log n$. We are interested in intervals (of consecutive small blocks) which are *compressible*, that is, can be stored using $\frac{1}{2} \log n + O(\log \log n)$ bits. Intervals of at most $\log n$ consecutive small blocks, containing at most $\frac{\log n}{2c \log \log n}$ elements of S , have this property by (1). The “greedy” way of choosing such compressible intervals from left to right results in $O(\frac{n \log \log n}{\log n})$ intervals (note that for $m = n \log^{\Omega(1)} n$ this is optimal since the entire compressed representation has size $\Omega(n \log \log n)$). To map block numbers into interval numbers we use Prop. 5, and the space usage is once again $O(\frac{n \log^2 \log n}{\log n})$ bits. Having found the intervals to compress, the construction proceeds as that leading to Prop. 5.

4 Improvement for Non-dense Subsets

Section 2 gave a solution to the static dictionary problem with space usage $B + O(n + \log m)$. To achieve redundancy sub-linear in n , we cannot use the hash functions of [9], since the representation is $\Omega(n)$ bit redundant (and it is far from clear whether a minimal, perfect hash function can have $o(n)$ bit redundancy *and* be evaluable in constant time). Also, it must be taken care of that $o(1)$ bit is wasted in each hash table cell, that is, nearly all bit patterns in all cells must be possible independently.

To use less space for storing the hash function, we will not require it to be perfect, but only to be perfect on some sufficiently large subset of S , which we handle first. The rest of S may then be handled by a dictionary that wastes more bits per element.

We will use the hash function (3). The family is indexed by k, p — the range a will depend only on m and n . Parameter p , where $p > a$, will be chosen later. The following result from [5] shows that it is possible to choose k such that $h_{k,p}$ is “almost 1-1 on S ” when hashing to a super-linear size table:

Lemma 7. *If the map $u \mapsto u \bmod p$ is 1-1 on S , there exists k , $0 < k < p$, such that $h_{k,p}$ is 1-1 on a set $S_1 \subseteq S$, where $|S_1| \geq (1 - O(\frac{a}{p}))|S|$.*

Without loss of generality, we will assume S_1 to be maximal, that is, $h_{k,p}[S_1] = h_{k,p}[S]$.

The idea will be to build two dictionaries: One for S_1 of Lemma 7, and one for $S_2 = S \setminus S_1$. Lookup may then be accomplished by querying both dictionaries. The dictionary for S_1 consists of the function $h_{k,p}$ given by Lemma 7, together with an a -element “virtual” hash table ($a = n \log^{O(1)} n$ to be determined). The virtual table contains $n_1 = |S_1|$ non-empty cells; to map their positions into n_1 consecutive memory locations, we need a partial function defined on $h_{k,p}[S]$, mapping bijectively to $\{1, \dots, n_1\}$. The static rank dictionary of Theorem 6 is used for this (two rank queries are used in order to determine if a cell is empty). For a good estimate of the memory used, we show:

Lemma 8. $B = n \log \frac{m}{n} + \frac{n}{\ln 2} - \Theta(n^2/m) - O(\log n)$.

Proof. By (1), it suffices to show the following:

$$(m-n) \log \frac{m}{m-n} = \frac{n}{\ln 2} - \Theta(n^2/m) . \quad (4)$$

We can assume $n = o(m)$. The Taylor series $\ln(1-x) = -\sum_{i>0} x^i/i$ shows $\ln(1-1/x) = -1/x - 1/2x^2 - O(x^{-3})$. Writing $(m-n) \log \frac{m}{m-n} = \frac{n-m}{\ln 2} \ln(1-n/m)$ and plugging in the above with $x = m/n$ gives the result. \square

Thus, the memory for the rank dictionary is $n_1 \log \frac{a}{n_1} + \frac{n_1}{\ln 2} + O(\frac{n \log^2 \log n}{\log n}) = n_1 \log \frac{a}{n} + \frac{n_1}{\ln 2} + O(\frac{n \log^2 \log n}{\log n} + n^2/a)$ bits.

We next show that the memory used for the hash table elements in the S_1 dictionary, $n_1 \lceil \log r \rceil$ bits, where r is the number defined in Lemma 1, can be made close to $n_1 \log \frac{m}{a}$ by suitable choice of p and a .

Lemma 9. *For any A with $3n \leq A = O(n \log n)$, there exists a prime p , $3A \leq p = O(n^2 \ln m)$, and a value of a , $A/3 \leq a \leq A$, such that:*

1. *The map $u \mapsto u \bmod p$ is 1-1 on S .*
2. $n_1 \lceil \log r \rceil = n_1 \log \frac{m}{a} + O(na/m + n^{12/21})$.

Proof. We first show how to make $\lceil \log r \rceil$ close to $\log r$ by suitable choice of a :

Sublemma 10. *For any $x, y \in \mathbf{R}_+$ and $z \in \mathbf{N}$, with $x/z \geq 3$, there exists $z' \in \{z+1, \dots, 3z\}$, 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 $\log \lceil \frac{x}{z} \rceil + y$ and $\log \lceil \frac{x}{3z} \rceil + y$, have different integer parts. So there exists z' , $z < z' \leq 3z$, such that $\lceil \log \lceil \frac{x}{z'} \rceil + y \rceil \leq \log \lceil \frac{x}{z'-1} \rceil + y$. A simple calculation gives $\log \lceil \frac{x}{z'-1} \rceil + y = \log \frac{x}{z'-1} + y + O(z/x) = \log \frac{x}{z'} + \log \frac{z'}{z'-1} + y + O(z/x) = \log \frac{x}{z'} + y + O(z/x + 1/z)$, and the conclusion follows. \square

Since $\log r = \log \lceil p/a \rceil + \log \lceil m/p \rceil$ and $p/A \geq 3$, the sublemma gives (for any p) an a in the correct range such that $\lceil \log r \rceil = \log r + O(a/p + 1/a)$.

Parameter $p = O(n^2 \ln n)$ is chosen such that $u \mapsto u \bmod p$ is 1-1 on S , and such that r is not much larger than m/a .

Sublemma 11. *In both of the following ranges, there exists a prime p , such that $u \mapsto u \bmod p$ is 1-1 on S :*

1. $n^2 \ln m \leq p \leq 3n^2 \ln m$.
2. $m < p < m + m^{12/21}$.

Proof. The existence of a suitable prime between $n^2 \ln m$ and $3n^2 \ln m$ is guaranteed by the prime number theorem (in fact, at least half of the primes in the interval will work). See [5, Lemma 2] for details. By [7] the number of primes between m and $m + m^\theta$ is $\Omega(m^\theta / \log m)$ for any $\theta > 11/20$. Take $\theta = 12/21$ and let p be such a prime; naturally the map is then 1-1. \square

A prime in the first range will be our choice when $m > n^2 A$, otherwise we choose a prime in the second range. For an estimate of $\log r$ in terms of m , n and a , we look at the two cases:

1. $\log r \leq \log\left(\frac{m}{a}\left(1 + \frac{a}{p} + \frac{p}{m}\right)\right) = \log(m/a) + O(a/p + p/m) = \log(m/a) + O(1/n)$.
2. $\log r = \log\lceil p/a \rceil \leq \log\left\lceil \frac{m+m^{12/21}}{a} \right\rceil \leq \log\left(\frac{m}{a}\left(1 + \frac{a}{m} + m^{-9/21}\right)\right) = \log(m/a) + O(a/m + m^{-9/21})$.

This, together with $\lceil \log r \rceil = \log r + O(a/p + 1/a)$, shows that the n_1 hash table entries use $n_1 \log(m/a) + O(na/m + n^{12/21})$ bits. \square

For the choices of k , p and a given by Lemmas 7 and 9, we can now compute the total space consumption for the S_1 dictionary:

- $O(\log n + \log \log m)$ bits for the k , p and a parameters, and for various pointers (the whole data structure has size $< n \log m$ bits).
- $n_1 \log \frac{a}{n} + \frac{n_1}{\ln 2} + O\left(\frac{n \log^2 \log n}{\log n} + n^2/a\right)$ bits for the “virtual table” mapping.
- $n_1 \log \frac{m}{a} + O\left(\frac{na}{m} + n^{12/21}\right)$ bits for the hash table contents.

This adds up to $n_1 \log \frac{m}{n} + \frac{n_1}{\ln 2} + O\left(\frac{n^2}{a} + \frac{na}{m} + \frac{n \log^2 \log n}{\log n} + \log \log m\right)$ bits.

The S_2 dictionary is implemented using the refined FKS dictionary [5] with a space consumption of $n_2 \log \frac{m}{n_2} + O(n_2 \log n + \log \log m) = n_2 \log \frac{m}{n} + O\left(\frac{n^2 \log n}{a} + \log \log m\right)$ bits. Thus, the total space usage of our scheme is:

$$n \log \frac{m}{n} + \frac{n}{\ln 2} + O\left(\frac{n^2 \log n}{a} + \frac{na}{m} + \frac{n \log^2 \log n}{\log n} + \log \log m\right) \text{ bits.} \quad (5)$$

By Lemma 8 this is

$$B + O\left(\frac{n^2 \log n}{a} + \frac{na}{m} + \frac{n \log^2 \log n}{\log n} + \log \log m\right) \text{ bits.} \quad (6)$$

We now get the main theorem:

Theorem 12. *The static dictionary problem with worst case constant lookup time can be solved with storage:*

1. $B + O(n \log^2 \log n / \log n + \log \log m)$ bits, for $n < m / \log \log m$.
2. $B + O(m \frac{\log \log m}{\log m})$ bits, for $n \geq m / \log \log m$.

Proof. In case 1 use the rank dictionary of Theorem 6 when $m \leq n \log^3 n$, and choose $A = \Theta(n \log^2 n)$ in the above construction when $m > n \log^3 n$. In case 2 use the rank dictionary of Prop. 5. \square

We have not associated any information with the elements of our set. The technique presented in this section extends to storing a partial function defined on S , mapping into a finite set V (whose elements are representable within $O(1)$ words). The information theoretical minimum is then $B^V = \lceil \log \binom{m}{n} + n \log |V| \rceil$, and for $m = \Omega(n \log^3 n)$ we get the exact same redundancy as in Theorem 12. The data structure is a simple modification of that described in this section; the size a of the virtual hash table is chosen such that the information packed in a hash table cell (quotient and function value) comes from a domain of size close to a power of 2. In the dense range, the rank dictionary can be used to index into a table of function values, but in general $\Omega(n)$ bits will be wasted in the table since $|V|$ need not be a power of 2.

Theorem 13. *The static partial function problem with worst case constant lookup time can be solved with storage:*

1. $B^V + O(n \log^2 \log n / \log n + \log \log m)$ bits, for $n < m / \log^3 m$.
2. $B^V + O(n)$ bits, for $n \geq m / \log^3 m$.

By using the dictionary of Prop. 2 to store S_2 and choosing a smaller in the data structure of described in this section, it is possible to achieve redundancy $o(n)$ when $n = o(m)$.

5 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 lookup 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 which allows efficient random access to the original bits.

The important ingredient in the solution is the concept of quotienting. Thus, the existence of an efficiently evaluable corresponding quotient function is a good property of a hash function. It is also crucial for the solution that the hash function used hashes U quite evenly to the buckets.

Quotienting works equally well in a dynamic setting, where it can be used directly to obtain an $O(B)$ bit scheme. However, lower bounds on the time for maintaining ranks under insertions and deletions (see [6]) show that our constructions involving the rank dictionary will not dynamize well.

It has not been described how to build the dictionary. It is, however, relatively straightforward to design a randomised algorithm which uses expected $O(n + p(n, m))$ time, where $p(n, m)$ is the expected time required for finding a prime in the range specified by Sublemma 11. By [1] we have that $p(n, m) = (\log n + \log \log m)^{O(1)}$.

It would be interesting to determine the exact redundancy necessary to allow constant time lookup. In particular, it is remarkable that no lower bound is known in a *cell probe* model (where only the number of memory cells accessed is considered). As for upper bounds, a less redundant way of mapping the elements of the virtual table to consecutive memory locations would immediately improve the asymptotic redundancy of our scheme. The idea of finding a replacement for the $h_{k,p}$ hash function, which can hash to a smaller “virtual table” or be 1-1 on a larger subset of S will not bring any improvement, because of a very sharp rise in the memory needed to store a function which performs better than $h_{k,p}$.

Acknowledgements: I would like to thank my supervisor, Peter Bro Miltersen, for encouragement and advice. Thanks also to Jakob Pagter and Theis Rauhe for their help on improving the presentation.

References

- [1] L. Adleman and M. Huang. Recognizing primes in random polynomial time. In Alfred Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 462–469, New York City, NY, May 1987. ACM Press.
- [2] A. Brodnik and J. I. Munro. Membership in constant time and minimum space. *Lecture Notes in Computer Science*, 855:72–81, 1994.
- [3] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640 (electronic), 1999.
- [4] Faith Fich and Peter Bro Miltersen. Tables should be sorted (on random access machines). In *Algorithms and data structures (Kingston, ON, 1995)*, pages 482–493. Springer, Berlin, 1995.
- [5] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.
- [6] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 15–17 May 1989.
- [7] D. R. Heath-Brown and H. Iwaniec. On the difference between consecutive primes. *Invent. Math.*, 55(1):49–69, 1979.
- [8] Peter Bro Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *Automata, languages and programming (Paderborn, 1996)*, pages 442–453. Springer, Berlin, 1996.
- [9] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k -probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.
- [10] Robert Endre Tarjan and Andrew Chi Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.
- [11] Andrew Chi Chih Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.