BRICS RS-98-28 R. Pagh: Low Redundancy in Dictionaries with $O(1)$ Worst Case Lookup Time

Basic Research in Computer Science

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

Rasmus Pagh

BRICS Report Series **RS-98-28**

ISSN 0909-0878 **November 1998**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/98/28/

Low redundancy in dictionaries with O(1) worst case lookup time

Rasmus Pagh

November 1998

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. The solution for dense subsets uses $B + O(\frac{|U| \log\log |U|}{\log |U|})$ bits of storage, and supports constant time rank queries. In a dynamic setting, allowing insertions and deletions, our techniques give an $O(B)$ bit space usage.

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 $U = \{0, \ldots, m-1\}$. We restrict the attention to the case where elements of U can be represented within $O(1)$ 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[1]. 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 $B = n \log \frac{m}{n} + (m-n) \log \frac{m}{m-n} - O(\log \frac{n(m-n)}{m})$, see [3]. For $n = o(m)$ the dominant term is $n \log \frac{m}{n}$, since $(m-n) \log \frac{m}{m-n} = \Theta(n)$.

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

[1] A part of the last word may be unused, and the query algorithm must work regardless of the contents of this part

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 [6] found a 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 [3] construct a static dictionary using $O(B)$ bits for any m and n. In the journal version of this paper [2], they achieve $B+O(\frac{B}{\log\log\log m})$ bits, and raise the question whether a more powerful model of computation is needed to further tighten the gap to the information theoretic minimum.

Yao [11] showed that in a restricted model, where words in the data structure must contain elements of S, the number of words necessary for $o(\log n)$ time lookup cannot be bounded by a function of n. Fich and Miltersen [5] showed that on a RAM with standard unit cost arithmetic operations but *without division and bit operations*, $o(\log n)$ time lookup requires $\Omega(m/n^\epsilon)$ words of memory for any $\epsilon > 0$. In [8] a RAM with bit operations but *without multiplication* is considered, and a lower bound of m^ϵ, for some $\epsilon > 0$, is shown when $n = m^{o(1)}$. No lower bound better than the trivial B bits seems to be known without restrictions on the data structure or the query algorithm.

As for dynamic dictionaries, an $O(n)$ word solution giving constant time lookup and constant expected amortized time for deletions and insertions is given in [4]. The space usage is improved in [3], where a solution using $O(B)$ bits (with the same time bounds) is sketched.

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. In particular, for $n = \omega(\log\log m)$ this means a vanishing number of redundant bits per element stored. The exact order of the bound, compared with lower bounds on the redundancy of the solution in [2], is given in the table below.

Range	Brodnik/Munro	This paper
$n < m^{\frac{\log\log m}{\log m}}$	$n \log\log m$	$n\sqrt{\frac{\log\log n}{\log n}} + \log\log m$
$m^{\frac{\log\log m}{\log m}} \le n < m(\frac{\log\log m}{\log m})^{2/3}$	$n \log\log m$	$n\sqrt{n/m} < n\sqrt[3]{\frac{\log\log n}{\log n}}$
$n \ge m(\frac{\log\log m}{\log m})^{2/3}$	$\min(n\log\log m, \frac{m}{(\log\log m)^{O(\log\log\log m)}})$	$m\frac{\log\log m}{\log m} < n\sqrt[3]{\frac{\log\log n}{\log n}}$

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, except for the last case (the "dense" range).

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 section 3, together with the main construction.

For dense subsets another technique is used, building upon the ideas of range reduction and a "table of small ranges" (both used in [2]). This dictionary supports rank queries, a fact which will be used in the construction for the non-dense case. This is treated in section 2.

The first part of section 3 describes a $B + O(n + \log\log m)$ bit scheme which does not depend on section 2, and can be read independently.

Section 4 describes a dynamic dictionary (insertions and deletions supported in expected amortized constant time), which uses $O(B)$ bits.

2 Dense subsets

In this section we describe a data structure for storing sets, which is space efficient for dense subsets (say, $n = \Omega(m \log\log m / \log m)$). The data structure will support queries on the ranks of elements (where the rank of u is defined as $\mathrm{rank}(u) = |\{v \in S | v \leq u\}|$). Using rank queries, it is possible to do membership queries; therefore we will call the data structure presented a *static rank dictionary.*

We split the universe into blocks U_i of size b ($b < \log m$ to be determined). Without loss of generality, assume $m = b \cdot s$ for some $s \in \mathbf{N}$ — otherwise pad U with at most $b-1$ dummy elements, increasing the final space consumption with $O(b)$ bits. Let $U_i = \{b \cdot i \ldots b \cdot (i+1) - 1\}$ denote the ith block. Blocks are grouped into clusters of c blocks each.

The idea will be to "explicitly" store the rank of the first element of each block, and store a compressed representation of the block itself. Extraction of rank information from this compressed form is done by table lookup. The ranks are stored using a 2-level structure (also used by Tarjan and Yao in [10]): The "A" level holds the rank of the first element in each cluster, and the "B" level holds the *offset* in rank for each block within the clusters, see Figure 1. Pointers to the compressed blocks are stored in the same way.

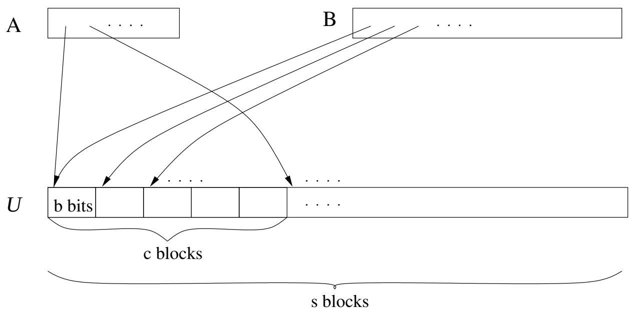

Figure 1: Splitting of U and structure of A and B tables

Let $\mathrm{ptr}(i)$ denote a pointer to the compressed representation of block i and let $\lfloor x \rfloor_y = \lfloor x/y \rfloor \cdot y$ be the largest number less than x which is a multiple of y. The representation consists of the following:

- Table A_r ($\lceil \log m \rceil$ bits/element), where $A_r[i] = \mathrm{rank}(b \cdot c \cdot i)$, $i = 0 \ldots \lceil s/c \rceil - 1$.

- Table B_r ($\lceil \log(c \log m) \rceil$ bits/element), where $B_r[i] = \text{rank}(b \cdot i) - \text{rank}(b \cdot \lfloor i \rfloor_c)$, $i = 0 \ldots s-1$.
- Table A_p ($\lceil \log m \rceil + 1$ bits/element), where $A_p[i] = \text{ptr}(c \cdot i)$, $i = 0 \ldots \lceil s/c \rceil - 1$.
- Table B_p ($\lceil \log(c \log m) \rceil + 1$ bits/element), where $B_p[i] = \text{ptr}(i) - \text{ptr}(\lfloor i \rfloor_c)$, $i = 0 \ldots s - 1$.
- A bit string C containing the ordered compressed representations of blocks. The representation of block i is the number $x = |S \cap U_i|$, $\lceil \log \log m \rceil$ bits, followed by $\lceil \log \binom{b}{x} \rceil$ bits representing $S \cap U_i$.
- Table D ($\lceil \log \log m \rceil$ bits/element), where $D[x, y, z]$ is the rank of the zth element of the set with x elements and compressed representation the first bits of y (if any such set exists, otherwise undefined), $x, z = 0 \ldots b$, $y = \{0, 1\}^b$. Elements are numbered $1 \ldots b$, the 0th element has rank 0 by definition.

A query for the rank of u can now be processed as follows ($C[a \ldots b]$ denotes the bit string starting with $C[a]$ and ending with $C[b-1]$):

1. Calculate the block $i = \lfloor u/b \rfloor$, and cluster $j = \lfloor i/c \rfloor$.
2. Determine the rank of $v = i \cdot b$ as $\text{rank}(v) = A_r[j] + B_r[i]$.
3. Determine the location of the compressed block as $\text{ptr}(i) = A_p[j] + B_p[i]$.
4. Set $x = C[\text{ptr}(i) \ldots \text{ptr}(i) + \lceil \log(c \log m) \rceil]$ and $y = C[\text{ptr}(i) + \lceil \log(c \log m) \rceil \ldots \text{ptr}(i) + \lceil \log(c \log m) \rceil + b]$.
5. **return** $\text{rank}(u) = \text{rank}(v) + D[x, y, u - v]$.

The correctness of the returned result should be immediate.

Analysis

It remains to be seen that the asserted space bounds on the table elements hold. Clearly the elements of tables A_r and A_p can be represented with $\lceil \log m \rceil$ and $\lceil \log m \rceil + 1$ bits, respectively (the latter because C has length less than $2m$). B_r holds non-negative integers bounded by the largest possible difference in rank within a span of c consecutive blocks, that is, the elements are in the range $0 \ldots b \cdot c$, so certainly $\lceil \log(c \log m) \rceil$ bits is sufficient. Similarly, B_p holds non-negative integers bounded by the largest possible difference in position between the compressed representations within a span of c consecutive blocks. And since the compressed representation of a block occupies less than $2b$ bits, we are done.

Now, for the analysis of the representation size, we need the following lemma from [3] on the total size of the compressed blocks:

Lemma 1 *Let $x_i = |S \cap U_i|$ and $B_i = \lceil \log \binom{b}{x_i} \rceil$. Then $\sum_{i=0}^{s-1} B_i < B + s$.*

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

Summing up the representation sizes (leaving out $O(\log m)$ bits for various pointers) we have:

- $O(\frac{m \log m}{bc})$ bits in each of A_r and A_p.
- $\frac{m \log(c \log m)}{b} + O(m/b)$ bits in each of B_r and B_p.
- $\frac{m \log \log m}{b} + O(m/b)$ bits within C for representing set sizes
- Less than $B + \lceil m/b \rceil$ bits in C for representing sets (by Lemma 1).
- $2^b b \lceil \log b \rceil$ bits for D.

Setting $b = c = \alpha \log m$, for some $0 < \alpha < 1$ we get the following:

Theorem 2 *A static rank dictionary with worst case constant query time, can be represented using $B + O(\frac{m \log \log m}{\log m})$ bits.*

By letting α vary by at most a factor of 2, b and c can be set to a power of 2, making multiplication and division a matter of shifting. Navigating the 3-dimensional array D can also be made easy by extending each side of the "cube" to have length a power of two. The reader is invited to verify that this means the above result holds, even if the query algorithm can use only AC^0 instructions.

Construction

We now sketch how to construct the static rank dictionary in $O(n + \frac{m \log \log m}{\log^2 m})$ time. For word length $\Theta(\log m)$ this is proportional to the time needed to read the input and write the representation, and thus optimal.

The construction algorithm will assume that the elements of S are given as a sorted list, $u_1 < \cdots < u_n$. This merely factors out the problem of sorting: Starting with an unordered list of elements of S and the corresponding rank dictionary, one can trivially sort in linear time.

First consider tables A_r and B_r. They may easily be filled out while running through $u_1 \ldots u_n$. However, in the case of B_r it is inefficient to fill in consecutive identical elements one at a time. Using a small table of precomputed words containing repetitions of element-size patterns, and appropriate shift and masking operations, one can fill any part of a word with a repeating element pattern. Thus the cost of filling out B_r can be brought down to the number of words in the representation of B_r plus the number of changes in the offset, n.

For C we use a 2^b-element table T, containing for any possible block the compressed representation and its length. Constructing C is then merely a matter of using each block as an index into the table. As positions of compressed representations get known, A_p and B_p are filled out similarly to A_r and B_r.

The table T may be constructed in several ways. A simple one is to compute each element in turn as a sum of at most b binomial coefficients (from a precomputed table). This approach takes time $O(b2^b)$ which is $o(\frac{m \log\log m}{\log^2 m})$.

D is constructed by running through T, for each $T[i]$ filling in the fields in D concerning the block coded by i. Again, a small precomputed table is used, this time for counting bits. The cost of all this is proportional to the number of entries in D, which is again $o(\frac{m \log\log m}{\log^2 m})$.

We can now state the following:

Theorem 3 *The static rank dictionary of Theorem 2 can be constructed from a sorted list of elements in S, in time $O(n + \frac{m \log\log m}{\log^2 m})$.*

3 Non-dense subsets

This section presents a static dictionary, which is space efficient unless the set S is dense (in which case the dictionary of the previous section is used). 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 `h`, and a table `T`:

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

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

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

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

We still need to argue that `q` need not be too expensive in terms of memory usage or evaluation time. The FKS perfect hashing scheme [6] 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:

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

Intuitively, this function gives the information that is thrown away by the modulo applications of the scheme's top level hash function (so in fact it is 1-1 even on the elements hashing to each bucket in the top level hash table). Since $p = O(m)$, the range of the function is $O(m/n)$, so $\log \frac{m}{n} + O(1)$ bits suffice for each hash table element.

Example This example assumes familiarity with the FKS scheme [6] (and is in fact the example in that paper subjected to quotienting). We look at $U = \{1, \ldots, 30\}$, $S = \{2, 4, 5, 15, 18, 30\}$, and choose $p = 31$, $k = 2$. The elements have the following quotient values:

u	2	4	5	15	18	30
$q_{k,p}(u)$	0	1	1	5	0	4

The quotient values take the place of elements in the data structure. The corresponding elements of S are written in quotes.

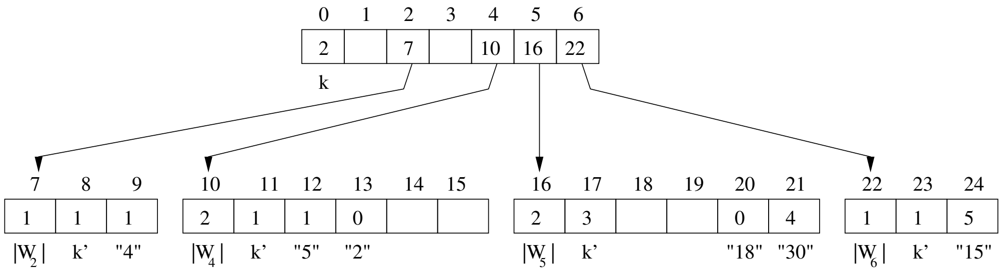

Figure 2: FKS scheme with quotienting

□

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).

One can thus get a space usage of $n \log \frac{m}{n} + O(n)$ bits for the hash table elements, and $O(n + \log\log m)$ for the hash function, that is:

Proposition 4 *The static dictionary problem with worst case constant lookup time can be solved using $B + O(n + \log\log m)$ bits of storage.*

Together with the dictionary of the previous section, this gives our first improvement of the result in [2]. As a corollary, we get a partial answer to an open problem stated in [5]:

Corollary 5 *When $n = \omega(\log\log m / \log\log\log m)$, the static dictionary problem with worst case constant lookup time can be solved using n words of storage (word size $\log m$).*

Proof. The dictionary of Proposition 4 uses $n \log m - n \log n + \Theta(n + \log\log m)$ bits. By assumption $n \log n = \omega(\log\log m)$, so this is less than $n \log m$ bits for $n > N$, where N is some sufficiently large constant. For $n \leq N$ we can simply list the elements of S. □

Refinement

To achieve a 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 constant time evaluable minimal, perfect hash function can have $o(n)$ bit redundancy). Also, it must be taken care of that $o(1)$ bit is wasted in each hash table cell, i.e. 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 use a hash function family from [6]: For any prime p and positive integers k, a, define the function

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

The family is indexed by k, p — parameter a is regarded as "fixed" since it will depend only on m and n. Parameter p, where $p > a$, will be chosen later. The corresponding quotient function family is $q_{k,p}$ defined earlier. We prove that $q_{k,p}$ is indeed appropriate:

Lemma 6 *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, $q_{k,p}[U] \subseteq \{0, \ldots, r-1\}$, where $r = \lceil m/p \rceil \cdot \lceil p/a \rceil$.*

Proof. Let $u_1, u_2 \in A_j(k,p)$ be such that $q_{k,p}(u_1) = q_{k,p}(u_2)$, in particular $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ and $(k \cdot u_1 \mod p) \operatorname{div} a = (k \cdot u_2 \mod p) \operatorname{div} a$. By the latter equation and the assumption on u_1, u_2, we have $k \cdot u_1 \mod p = k \cdot u_2 \mod p$, so since p is prime and $k \neq 0$, $u_1 \mod p = u_2 \mod p$. Since also $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ it must be the case that $u_1 = u_2$. The bound on the range of $q_{k,p}$ is straightforward. □

We shall make use of the following result from [6], which states that that one can get an "almost 1-1 on S" hash function $h_{k,p}$ by hashing to a super-linear size table:

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

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

The idea will be to build two dictionaries: One for S_1 of Lemma 7, and one for $S_2 = S \backslash S_1$. Lookup may then be accomplished by querying both dictionaries.

The dictionary for S_1 consists of the function $h_{k,p}$ of Lemma 7, together with an a-element "virtual" hash table ($a < n \log n$ to be determined). The virtual table contains $n_1 = |S_1|$ non-empty cells; to map these positions into n_1 consecutive memory locations, we need a partial function defined on $h_{k,p}[S]$ and mapping these elements bijectively to $\{1, \ldots, n_1\}$. The static rank dictionary of section 2 is used for this (two rank queries are used in order to determine if a position is used). Figure 3 shows an overview of the construction. By Theorem 2 the rank dictionary uses nearly minimal memory: $n_1 \log \frac{a}{n_1} + (a - n_1) \log \frac{a}{a-n_1} + O(\frac{a \log\log n}{\log n})$ bits. The first term is $n_1 \log \frac{a}{n} + O(n^2/a)$. The second term is less than $\frac{n_1}{\ln 2}$; we show something slightly stronger:

Lemma 8 *The following estimate holds: $(m-n) \log \frac{m}{m-n} = \frac{n}{\ln 2} - \Theta(n^2/m)$.*

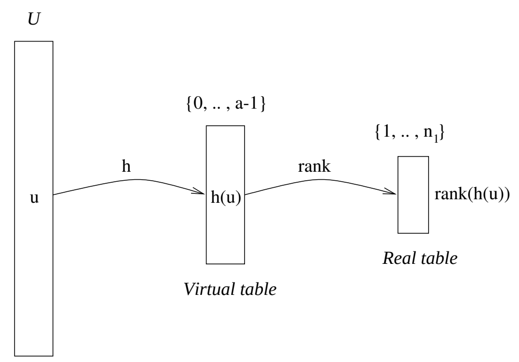

Figure 3: Overview of the dictionary for S_1

Proof. 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. $\square$

It is interesting to note that $h_{k,p}$ and the rank dictionary constitute a perfect hash function for S_1, but use more space[2] than the $O(n + \log\log m)$ bits sufficient the represent such a function. However, as we shall see, the rank dictionary encodes just enough information on S to justify this extra use of space.

We next show that the memory used for the hash table elements in the S_1 dictionary, $n_1\lceil\log r\rceil$ bits, can be made close to $n_1\log\frac{m}{n}$:

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

1. *The map $u \mapsto u \mod p$ 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. The memory used for storing each table element is $\lceil\log r\rceil$. This can be made close to $\log r$:

Claim 10 *For any $x, y \in \mathbf{R}_+$ and $z \in \mathbf{N}$, with $x/z \ge 3$, there exists $z' \in \{z+1, \ldots, 3z\}$, such that $\lceil\log\lceil x/z'\rceil + y\rceil \le \log(x/z') + y + O(z/x + 1/z)$.*

Proof. Since $x/z \ge 3$, 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', $z < z' \le 3z$, such that $\lceil\log\lceil\frac{x}{z'}\rceil + y\rceil \le \log\lceil\frac{x}{z'-1}\rceil + y$. 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 $p/A > 3$, the claim gives (for any p) an a such that $\lceil\log r\rceil = \log r + O(a/p + 1/a)$.

[2] When $a = \omega(n)$, which will be the case

Parameter p is chosen such that $u \mapsto u \mod p$ is 1-1 on S, such that it is not too big (it needs to be stored) and such that r is not much larger than m/a.

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

1. $n^2 \ln m \le p \le 3n^2 \ln m$ *(this will be our choice when $m > n^3 \ln m$)*
2. $m < p < m + m^{12/21}$ *(this will be our choice when $m \le n^3 \ln m$)*

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 [6, 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. $\square$

For an estimate of $\log r$ in terms of m, n and a, we look at the cases for p in Claim 11:

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

This, together with Claim 10, gives that the n_1 hash table entries use $n_1 \log(m/a) + O(na/m + n^{12/21})$ bits. $\square$

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 representation has size $< n \log m$ bits).
- $n_1 \log \frac{a}{n} + \frac{n_1}{\ln 2} + O(\frac{a \log\log n}{\log n} + n^2/a)$ bits for the "virtual table" mapping.
- $n_1 \log \frac{m}{a} + O(\frac{na}{m} + n^{12/21})$ bits for the hash table contents.

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

We now look at the space of the dictionary for S_2. First note that since $S_2 \subseteq \cup_{j \in h_{k,p}[S_1]} A(k,j)$, the rank dictionary offers a constant time computable map, which is 1-1 on S_2, namely $\rho_2 : u \mapsto r \cdot \mathrm{rank}(h_{k,p}(u)) - q_{k,p}(u)$, where the rank is with respect to $h_{k,p}[S_1]$ (i.e. the set stored in the rank dictionary). A dictionary for $\rho_2[S_2]$ is constructed with respect to the universe $U_2 = \rho_2[U]$ (a query for u will be "converted", in constant time, into a query for $\rho_2(u)$). We then only have to deal with a universe of size $|U_2| = O(mn/a)$. The S_2 dictionary may be built using the dictionary of Proposition 4, without wasting too many bits: The space usage is $n_2 \log \frac{|U_2|}{n_2} + \frac{n_2}{\ln 2} + O(n_2 + \log\log m) = n_2 \log \frac{m}{n} + \frac{n_2}{\ln 2} + O(n^2/a + \log\log m)$. Thus, the total space usage of our scheme is $n\log\frac{m}{n}+\frac{n}{\ln 2}+O(\frac{n^2}{a}+\frac{na}{m}+\frac{a\log\log n}{\log n}+\log\log m)$ bits. Using the estimate in Lemma 8 this is

$$B+O(\frac{n^2}{a}+\frac{na}{m}+\frac{a\log\log n}{\log n}+\log\log m)\text{ bits}$$

We now get the main theorem:

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

1. $B+O(n\sqrt{\log\log n/\log n}+\log\log m)$ *bits, for* $n<m^{\frac{\log\log m}{\log m}}$.
2. $B+O(n\sqrt{n/m})$ *bits, for* $m^{\frac{\log\log m}{\log m}}\leq n<m(\frac{\log\log m}{\log m})^{2/3}$.
3. $B+O(m\frac{\log\log m}{\log m})$ *bits, for* $n\geq m(\frac{\log\log m}{\log m})^{2/3}$.

Proof. In case 1. choose $a=\Theta(n\sqrt{\log n/\log\log n})$. In case 2. choose $a=\Theta(\sqrt{mn})$ in the above construction. In case 3. we use the dictionary of the previous section. □

We have not associated any information with the elements of our set. In the non-dense case, it is possible to store a partial function defined on S, mapping into a finite set V, with the exact same redundancy as in Theorem 12 (this time the information theoretical minimum is $B^V=B+n\log|V|$). The data structure is a simple modification of the above; the value of a 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.

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

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

For $n=\Theta(m)$ the rank dictionary gives a $o(n)$ bit redundant solution when $|V|$ is a power of 2, but it seems hard to avoid wasting $\Omega(1)$ bit for each function value for general $|V|$.

3.1 Construction

We now sketch how to construct the static dictionary described above, in expected time $O(n+(\log\log m)^{O(1)})$. The last part of the expression comes from the time needed to find the prime p of the hash function, but is so small that it can be ignored unless m is very large compared to n.

First note that the hardest part is finding the parameters p and k of the hash function, and building the dictionary for S_2:

- Parameter a is simple to compute according to Claim 10, for example by binary search on the interval in which a is wanted.
- Theorem 3 implies that the rank dictionary for $h_{k,p}[S]$ with respect to $\{0,\ldots,a-1\}$ can be constructed in time $O(n)$ (The initial sorting can be done in linear time using Radix-sort).
- Filling in quotient values in the hash table is clearly possible in constant time per element once $h_{k,p}$, $q_{k,p}$ and the rank dictionary are available.

Parameter p is found by randomly choosing numbers from the interval given in Claim 11. Each such number chosen is checked for primality (using a probabilistic check which uses expected time poly-logarithmic in the number checked [1], that is, time $(\log n + \log\log m)^{O(1)}$). When a prime is found, it is checked that the map $u \mapsto u \mod p$ is 1-1 on S (time $O(n)$ using Radix-sort on the function values). The following sharpening of the statement of Claim 11 implies that all of this takes expected time $O(n + (\log\log m)^{O(1)})$:

Claim 11'

1. When choosing a random number in $\{x,\ldots,x+y\}$, where $x^{12/21} \leq y \leq 2x$, it is prime with probability $\Omega(1/\log x)$.
2. When choosing a random prime $q \in \{n^2 \ln m, \ldots, 3n^2 \ln m\}$, the map $u \mapsto u \mod q$ is 1-1 on S with probability at least 1/2.

Proof. Same as Claim 11. □

Parameter k is chosen at random and checked for the inequality of Lemma 7 (in time $O(n)$). For a large enough constant in the big-oh of Lemma 7, the expected number of attempts made before finding a suitable k is constant, and thus the expected time for the choice is $O(n)$.

The dictionary for S_2 (based on [9]) can be built in expected time $O(|S_2|)$. This is not described in [9], but the only real change compared to the expected $O(n)$ time construction in the FKS scheme is how to choose second-level hash functions: At all times maintain a linked list of all buckets for which no resolving hash function has been found. In each of $\log n$ rounds, a hash function resolving buckets containing at least half the remaining elements of S_2 is found. When randomly selecting hash functions, the expected number of unsuccessful attempts each round is constant, and the cost of an attempt is proportional to the number of remaining elements. So the expected time for choosing all hash functions is linear.

Thus we have:

Theorem 14 *The data structure of Theorem 12 can be constructed in expected time $O(n + (\log\log m)^{O(1)})$.*

4 Dynamic version

In this section we outline how to apply quotienting to building a space efficient dynamic dictionary, supporting insertion and deletion of elements. The dictionary uses $O(B)$ bits, matching the bound achieved in [3] (more precisely, the data structure at all times resides within a contiguous memory segment of $O(B)$ bits). For simplicity, the model will not be a RAM, but a *cell probe* model, i.e. we only count the number of memory accesses (the result in [3] holds in a weaker RAM model). In particular, tiny data structures residing within a single word can trivially be handled. We require that the query and update routines are "memoryless", i.e. know nothing about the set stored when they are started.

It has been known for some time how to implement dynamic dictionaries using $O(n)$ words [4]. This is $O(n \log m)$ bits, and since $B = n \log \frac{m}{n} + \Theta(n)$, this is $O(B + n \log n)$ bits. That is, the result in [4] yields:

Theorem 15 *There exists a dynamic dictionary requiring $O(B+n \log n)$ bits, which supports worst-case constant time lookup and amortized expected constant time insertion and deletion.*

Note that the space consumption is $O(B)$ for $n = O(m^{1-\epsilon})$, for constant $\epsilon > 0$, hence we can concentrate on $n = \Omega(m^{1-\epsilon})$. (But in fact, all we will be using is that a pointer to a bit in the representation can be stored in $O(\log n)$ bits). On the other hand, as already noticed, for very small universes the dynamic dictionary problem is efficiently solvable:

Lemma 16 *In a cell probe model with word size at least m, the dynamic dictionary problem is solvable in space $O(B)$.*

The approach is to use a hash function to split the universe into parts, and handle each "small universe" by a separate dynamic dictionary. The small universes are $U_i = h_{k,p}^{-1}\{i\}$, $i = 0, \ldots, a-1$, where $a = \Theta(n/\log n)$. (The number of universes is chosen such that pointers to all dictionaries can be stored in $O(n)$ bits). We denote the corresponding subsets by $S_i = S \cap U_i$. The quotient function $q_{k,p}$ is used for injectively mapping elements of U_i to elements in the range $\{0, \ldots, O(m/a)\}$ (the dictionary handling U_i will work on these values).

The history of the dynamic dictionary proceeds in phases. Each phase takes expected $O(n)$ time, and the number of insertions and deletions in a phase starting with a dictionary of n elements is $n/2$. At the beginning of each phase, the entire dictionary is rebuilt (in expected $O(n)$ time). Each small dictionary is given twice as much space as it uses. If it runs out of space, it is moved and given twice as much space in the upper, unused memory area. This guarantees that the total space used at any time is $O(1)$ times the space occupied by the dictionaries.

If we handle each universe by a dictionary of the kind given by Theorem 15, the total space consumption is $O(B + \sum_i |S_i| \log |S_i|)$ bits (using Lemma 1). We now investigate when this can be made $O(B)$. Let $\tilde{S}$ be the set of all elements which are in the dictionary during some phase, starting with n elements; we have the following:

Lemma 17 *Let p be a prime such that $m < p < 2m$. Consider the sets $\tilde{S}_i = \tilde{S} \cap h_{k,p}^{-1}\{i\}$. For at least half the choices of $k \in \{1, \ldots, p-1\}$ we have: $\sum_i |\tilde{S}_i| \log |\tilde{S}_i| = O(n \log \log n)$.*

Proof. By the results in [6], for at least half the choices of k we have $\sum_i |\tilde{S}_i|^2 = O(|\tilde{S}|^2/a) = O(n \log n)$. This means that the number of $\tilde{S}_i$ of size more than $\log n$ is $O(n/\log n)$, so these sets contribute only $O(n)$ to the sum. $\square$

Therefore, with probability at least 1/2 the choice of hash function is "good". With probability at most 1/2, the inequality of Lemma 17 fails to hold at some time during the phase. In this case a new hash function is chosen, and everything is rebuilt. The expected time for a phase is therefore $O(n)$. We have arrived at:

Proposition 18 *There exists a dynamic dictionary requiring $O(B + n \log\log n)$ bits, which supports worst-case constant time lookup and amortized expected constant time insertion and deletion.*

This is $O(B)$ bits when $n = O(m/\log^\epsilon m)$, for constant $\epsilon > 0$.

In order to deal with the case $n = \Omega(m/\log^\epsilon m)$, each small universe is split into $\log n/\log\log n$ "tiny universes" using another hash function. Again, all hash functions and pointers to the dictionaries of the tiny universes use $O(n)$ bits, since the entire data structure for the small universe has size $\log^{O(1)} n$. Now the size of each tiny universe is $m \log\log n/n$, which is $\log^\epsilon m \log\log n$ and hence less than one word. So Lemma 16 can be used to handle the tiny dictionaries. The analysis of this is similar to that leading to Proposition 18, but now on two levels (each small dictionary has phases, etc.). We do not go into details, but just state

Theorem 19 *There exists a dynamic dictionary requiring $O(B)$ bits, which supports worst-case constant time lookup and amortized expected constant time insertion and deletion in a cell probe model.*

5 Conclusion

We have seen that for the static dictionary problem it is possible to come very close to the information theoretic minimum, while retaining constant lookup time. 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.

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, without the restrictions mentioned in the introduction. A lower bound in a *cell probe* model (where only the number of memory cells accessed is considered) would be interesting. 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 $h_{k,p}$.

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 almost minimum space. *To appear in SIAM Journal on Computing.*

[3] A. Brodnik and J. I. Munro. Membership in constant time and minimum space. *Lecture Notes in Computer Science*, 855:72–81, 1994.

[4] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[5] 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.

[6] 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.

[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.

Recent BRICS Report Series Publications

RS-98-28 **Rasmus Pagh.** ***Low Redundancy in Dictionaries with $O(1)$ Worst Case Lookup Time.*** **November 1998. 15 pp.**

RS-98-27 **Jan Camenisch and Markus Michels.** ***A Group Signature Scheme Based on an RSA-Variant.*** **November 1998. 18 pp. Preliminary version appeared in Ohta and Pei, editors, *Advances in Cryptology: 4th ASIACRYPT Conference on the Theory and Applications of Cryptologic Techniques*, ASIACRYPT '98 Proceedings, LNCS 1514, 1998, pages 160–174.**

RS-98-26 **Paola Quaglia and David Walker.** ***On Encoding $p\pi$ in $m\pi$.*** **October 1998. 27 pp. Full version of paper to appear in *Foundations of Software Technology and Theoretical Computer Science: 18th Conference*, FCT&TCS '98 Proceedings, LNCS, 1998.**

RS-98-25 **Devdatt P. Dubhashi.** ***Talagrand's Inequality in Hereditary Settings.*** **October 1998. 22 pp.**

RS-98-24 **Devdatt P. Dubhashi.** ***Talagrand's Inequality and Locality in Distributed Computing.*** **October 1998. 14 pp.**

RS-98-23 **Devdatt P. Dubhashi.** ***Martingales and Locality in Distributed Computing.*** **October 1998. 19 pp.**

RS-98-22 **Gian Luca Cattani, John Power, and Glynn Winskel.** ***A Categorical Axiomatics for Bisimulation.*** **September 1998. ii+21 pp. Appears in Sangiorgi and de Simone, editors, *Concurrency Theory: 9th International Conference*, CONCUR '98 Proceedings, LNCS 1466, 1998, pages 581–596.**

RS-98-21 **John Power, Gian Luca Cattani, and Glynn Winskel.** ***A Representation Result for Free Cocompletions.*** **September 1998. 16 pp.**

RS-98-20 **Søren Riis and Meera Sitharam.** ***Uniformly Generated Submodules of Permutation Modules.*** **September 1998. 35 pp.**

RS-98-19 **Søren Riis and Meera Sitharam.** ***Generating Hard Tautologies Using Predicate Logic and the Symmetric Group.*** **September 1998. 13 pp.**

RS-98-18 **Ulrich Kohlenbach.** ***Things that can and things that can't be done in PRA.*** **September 1998. 24 pp.**