BRICS

Basic Research in Computer Science

BRICS RS-99-48 R. Pagh: Faster Deterministic Dictionaries

Faster Deterministic Dictionaries

Rasmus Pagh

BRICS Report Series **RS-99-48**

ISSN 0909-0878 **December 1999**

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/99/48/

Faster Deterministic Dictionaries

Rasmus Pagh*

Abstract

We consider static dictionaries over the universe $U = \{0,1\}^w$ on a unit-cost RAM with word size w. Construction of a static dictionary with linear space consumption and constant lookup time can be done in linear expected time by a randomized algorithm. In contrast, the best previous deterministic algorithm for constructing such a dictionary with n elements runs in time $O(n^{1+\epsilon})$ for $\epsilon > 0$. This paper narrows the gap between deterministic and randomized algorithms exponentially, from the factor of n^ϵ to an $O(\log n)$ factor. The algorithm is weakly non-uniform, i.e. requires certain precomputed constants dependent on w. A by-product of the result is a lookup time vs insertion time trade-off for dynamic dictionaries, which is optimal for a certain class of deterministic hashing schemes.

1 Introduction

We consider the amount of time needed to deterministically construct a *static dictionary*, i.e. a data structure for storing any subset S of universe U such that lookups (queries of the form "$x \in S$?") can be carried out efficiently. Static dictionaries show up in many applications and are an important part of a number of data structures. It is therefore of interest to have static dictionaries which are space economical, support fast lookups, and can be constructed efficiently.

Our model of computation is a unit-cost word RAM [9] with a standard instruction set, including multiplication and bit operations. Throughout this paper S will refer to an arbitrary set of n elements from the universe $U = \{0,1\}^w$, where w is the word size (a RAM model where a single unit of data fits into one machine word is often referred to as *trans-dichotomous*). As is usual in this model, we regard the elements of U both as non-negative integers and as bit strings.

It is known that static dictionaries with worst case *constant* lookup time and a space consumption of $O(n)$ words can be constructed in this model. We therefore focus on dictionaries with these properties, henceforth referred to as

*__BRICS__ (Basic Research in Computer Science, Centre of the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Denmark. E-mail: `pagh@brics.dk`

efficient. Until recently, all algorithms for constructing efficient dictionaries in time time better than $O(n^3)$ were randomized, forcing the use of randomness in many applications. The aim of this paper is to show that randomized static dictionaries can be replaced with efficient deterministic ones which have nearly the same construction time.

1.1 Related work.

Tarjan and Yao [15] gave a rigorous basis for understanding some heuristics which had been used for table compression, resulting in the "double displacement" dictionary which is efficient for $w = O(\log n)$. In this case, it was shown how to construct the dictionary in time $O(n^2)$.

A breakthrough was made by Fredman, Komlós and Szemerédi [7], who showed how to use universal hash functions [4] to build an efficient dictionary for any word size. Two construction algorithms were given: A randomized one running in expected time $O(n)$, and a deterministic one with a running time of $O(n^3 w)$. Raman [13] sped up the choice of universal hash functions in the deterministic algorithm, obtaining $O(n^2 w)$ deterministic construction time. Alon and Naor [1] used small bias probability spaces to derandomize a variant of the FKS scheme, achieving construction time $O(nw \log^4 n)$. However, in this variant a lookup requires evaluation of a linear function in time $\Theta(w/\log n)$, so the dictionary is not efficient unless $w = O(\log n)$. Another variant of the FKS scheme reduces the number of random bits to $O(\log n + \log w)$, while achieving $O(n)$ time construction with high probability [5].

For $w = n^{\Omega(1)}$, fusion trees [8] improve the above deterministic bounds to $n^{1+\epsilon}$ for any fixed $\epsilon > 0$, as observed by Andersson [2]. The fusion tree construction algorithm is *weakly non-uniform* in that it requires access to a constant number of precomputed word-size constants depending (only) on w. These constants may be thought of as computed at "compile time".

Miltersen [11] introduced the use of *error-correcting codes* in a novel approach to the construction of perfect hash functions. For any fixed $\epsilon > 0$, this yields an efficient dictionary requiring $O(n^{1+\epsilon})$ construction time, also by a weakly non-uniform algorithm. The lookup time is inversely proportional to ϵ. Extending the result of Miltersen, Hagerup exhibited a trade-off between construction time and lookup time [10]. The algorithm achieves construction time $O(n \log n)$ and lookup time $O(\log \log n)$ simultaneously. For lookup time $o(\log \log n)$ it needs construction time $n\, 2^{(\log n)^{1-o(1)}}$.

1.2 This work.

We show how to construct efficient static dictionaries in deterministic time $O(n \log n)$. We first develop a randomized variant of the Tarjan-Yao dictionary, which is efficient for $w = O(\log n)$. The construction algorithm is derandomized using conditional expectations, resulting in a deterministic $O(n \log n)$ algorithm. The resulting algorithm is quite simple compared with the $O(n \log^5 n)$ time algorithm in [1]. Finally, universe reduction techniques, including error-correcting codes, are applied to extend the result to arbitrary word sizes.

As a consequence of the result we are able to state an improved trade-off between lookup time and insertion time for deterministic dynamic dictionaries. By a result of Dietzfelbinger et al. [6] the trade-off is optimal for a certain class of data structures based on hashing.

What is actually shown in the following is how to construct an efficient perfect hash function for S.

Definition 1 *A function $h : U \to \{0, \ldots, r\}$, where $r = O(n)$, is an* efficient perfect hash function *(for S) if it is 1-1 on S, can be stored using $O(n)$ words of memory and evaluated in constant time.*

Such a function immediately yields an efficient solution to the static dictionary problem.

2 Randomized double displacement

In this section we develop a randomized, but derandomization-friendly, variant of the double displacement perfect hash functions of Tarjan and Yao [15]. The resulting perfect hash function is efficient for $w = O(\log n)$. Following Tarjan and Yao, observe that words of length $O(\log n)$ can be regarded as constant length strings over an alphabet of size n. The trie (with n-way branching) of such strings permits lookup of elements (and associated values) in constant time. Although each of the $O(n)$ nodes of such a trie uses a table of size n, only $O(n)$ entries contain important information (an element or a pointer). That is, to store all tables within $O(n)$ words, we just need to construct a perfect hash function from the $O(n^2)$ table entries to a range of size $O(n)$. Since it is simple to map all tables into a universe of size $O(n^2)$, Tarjan and Yao proceed to look at the case where $w \leq 2 \log n + O(1)$.

Without loss of generality we assume $w \geq 2 \log n + k$, where k is a constant to be determined (If words are shorter, we conceptually pad them with zeros). We split words into parts of $\alpha = \lceil \log(2n) \rceil$ and $\beta = w - \alpha$ bits, denoting the parts of a word x by x' and x'', respectively. The universe may be viewed as a 2^α times 2^β grid, where the column and row numbers of x are given by x' and x''.

The idea of Tarjan and Yao is to perform a "double displacement" rearrangement process, permuting the elements of the grid such that elements of S end up in distinct columns. Then a perfect hash function for S is immediate: map x to the column that x is moved to by the displacement process. Let $\oplus$ denote the operation of bitwise exclusive-or. The double displacement process has two steps:

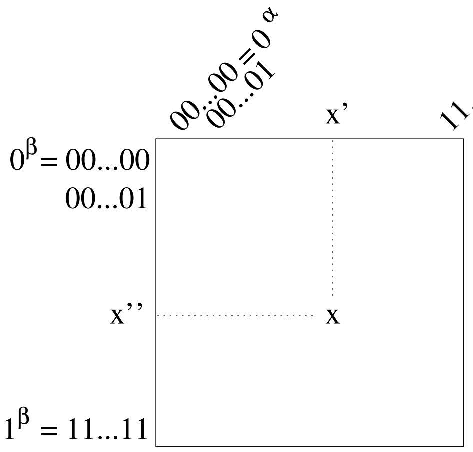

Figure 1: Grid view of U

1. Displace each column by moving element x in column c to the row $x'' \oplus a_c$.

2. Displace each row by moving element x in row r to the column $x' \oplus b_r$.

These steps use "displacement values" $a_{0^\alpha}, \ldots, a_{1^\alpha} \in \{0,1\}^\beta$ and $b_{0^\beta}, \ldots, b_{1^\beta} \in \{0,1\}^\alpha$. The resulting hash function is $x \mapsto x' \oplus b_{x'' \oplus a_{x'}}$.

Tarjan and Yao use another group operator (integer addition) instead of $\oplus$, and use a grid of height $O(n \log\log n)$. But the significant departure from the Tarjan-Yao approach comes with the way the displacement values are chosen. They use a "first-fit" sequential search strategy, whereas we will be choosing the displacement values in a random fashion. The selection process will be *controlled*, in that each random choice is checked, and if necessary re-done, until it is (close to) as "good" as could be expected.

Our measure of goodness when choosing a column displacement value is the number of "row collisions" introduced. More precisely, suppose that displacement values for columns $p_1, \ldots, p_{i-1}$ have been found, and denote by $S'_{p_j} = \{x \in S \mid x' = p_j\}$ the elements in column p_j. The set of *row collisions* introduced by choosing displacement value a_{p_i} is $C_i = \{(x,y) \in S'_{p_j} \times S'_{p_i} \mid j < i,\ x'' \oplus a_{p_j} = y'' \oplus a_{p_i}\}$. Since any pair $(x,y) \in S'_{p_j} \times S'_{p_i}$ has probability $2^{-\beta}$ of colliding when a_{p_i} is chosen uniformly at random from $\{0,1\}^\beta$, the expected number of row collisions is $E[|C_i|] = 2^{-\beta}|S'_{p_i}| \sum_{j<i} |S'_{p_j}|$. Pick any ϵ with $0 < \epsilon < 1$. We have $|C_i| \le (1+\epsilon)E[|C_i|]$ with probability at least $1 - 1/(1+\epsilon)$. Hence, in an expected constant number of random attempts we can find a_{p_i} such that

$$|C_i| \le (1+\epsilon)\, 2^{-\beta}|S'_{p_i}| \sum_{j<i} |S'_{p_j}| \ . \tag{1}$$

Row displacements are chosen according to the same criterion as the column displacements: We allow at most $1+\epsilon$ times the expected number of column collisions. The only new thing is an insistence that row displacements are found in *non-increasing* order of the number of elements in the rows, i.e. in an order $\tilde{p}_1, \ldots, \tilde{p}_{2^\beta}$ such that $|S''_{\tilde{p}_j}| \geq |S''_{\tilde{p}_{j+1}}|$ for $j = 1, \ldots, 2^\beta - 1$, where $S''_{\tilde{p}_j} = \{x \in S \mid x'' \oplus a_{x'} = \tilde{p}_j\}$ denotes the elements in row $\tilde{p}_j$ after the column displacements. Suppose displacement values for rows $\tilde{p}_1, \ldots, \tilde{p}_{i-1}$ have been found, and we want to find displacement value $b_{\tilde{p}_i}$. For rows with at most one element, the expected number of collisions introduced by a random displacement is at most 1/2 (since $2^\alpha \geq 2n$), so we find a displacement value with at most $(1+\epsilon)/2$ collisions (that is, no collisions). When $|S''_{\tilde{p}_i}| > 1$, we can bound the number of new column collisions, denoted by $|\tilde{C}_i|$, as follows:

$$\begin{aligned}
\frac{2^\alpha|\tilde{C}_i|}{1+\epsilon} &\leq |S''_{\tilde{p}_i}| \sum_{j<i} |S''_{\tilde{p}_j}| && \text{by reasoning similar to (1)} \\
&\leq \sum_{j<i} |S''_{\tilde{p}_j}|^2 && \text{since } |S''_{\tilde{p}_i}| \leq |S'_{\tilde{p}_j}| \text{ for } j < i \\
&\leq 4 \sum_{j<i} \binom{|S''_{\tilde{p}_j}|}{2} && \\
&\leq 4 \sum_{i} |C_i| && \text{since } |S''_{\tilde{p}_j}| > 1 \text{ for } j \leq i \\
&\leq 4(1+\epsilon)2^{-\beta}\binom{n}{2} \ . && \text{by (1)}
\end{aligned}$$

Choosing $k \geq 1 + 3\epsilon$, the requirement that $\alpha + \beta \geq 2\log n + k$, implies $|\tilde{C}_i| = 0$, as desired. This means that a double displacement hash function constructed in the way described will indeed be perfect for S.

Figure 2 details an implementation of the scheme described above. The function $\mathsf{DoubleDisplacement}_\epsilon$ sets up two calls to the displacement procedure $\mathsf{RandDispl}_\epsilon$ to find the column and row displacement values. We outline the argument that $\mathsf{RandDispl}_\epsilon$ runs in expected linear time. The order in which to choose the displacement values (array P) can be found in linear time (using bucket-sort). The array T contains at any time the number of elements so far displaced to each column/row, and we also maintain a counter t of the number of displaced elements (the total work for maintaining this information is clearly $O(n)$). This allows computation of the number of collisions introduced when displacing row/column $P[i]$ in time $O(|S[P[i]]|)$. By our analysis, the **repeat** loop needs an expected constant number of iterations before finding a good displacement value, so the total work done in this loop is expected $O(n)$. The work done in the rest of the algorithm is clearly $O(n)$, so we are done.

Lemma 2 *For $2(\log n + \epsilon) + 1 \leq w \leq 2\log n + O(1)$, $\mathsf{DoubleDisplacement}_\epsilon(S)$ computes the $O(n)$ parameters of a double displacement perfect hash function for S. The computation takes expected time $O(n)$ and uses $O(n)$ words of storage.*

```
function RandDispl_ε(S[0], ..., S[b − 1] ⊆ {0,1}^γ)
   Find permutation P such that
      |S[P[j − 1]]| ≥ |S[P[j]]| for 1 ≤ j < b;
   for x ∈ {0,1}^γ do T[x] := 0;
   t := 0;
   for i := 0 to b − 1 do
      repeat
         D[P[i]] :=Random({0,1}^γ);
         c := Σ_{x∈S[P[i]]} T[x ⊕ D[P[i]]];
      until c ≤ (1 + ε)2^{−γ} t|S[P[i]]|;
      for x ∈ S[P[i]] do
         T[x ⊕ D[P[i]]] := T[x ⊕ D[P[i]]] + 1;
      t := t + |S[P[i]]|;
   end;
   return D;
end;

function DoubleDisplacement_ε(S ⊆ {0,1}^w)
   if w < 2 log |S| + 1 + 3ε then return 'Error';
   α := ⌈log(2|S|)⌉; β := w − α;
   Define x′ and x″ to be the first α and
      the last β bits of a word x, respectively;
   for i ∈ {0,1}^α do S′[i] = {x″ | x ∈ S, x′ = i};
   (A[0^α], ..., A[1^α]) :=RandDispl_ε(S′[0^α], ..., S′[1^α]);
   for i ∈ {0,1}^β do S″[i] = {x′ | x ∈ S, x″ ⊕ A[x′] = i};
   (B[0^β], ..., B[1^β]) :=RandDispl_ε (S″[0^α], ..., S″[1^α]);
   return (A, B);
end;
```

Figure 2: Randomized double displacement

```
function Hash(A, B, x′, x″)
   return x′ ⊕ B[x″ ⊕ A[x′]];
end;
```

Figure 3: The double displacement hash function

Corollary 3 *For $w = O(\log n)$, an efficient perfect hash function for S can be constructed by a randomized algorithm in expected time $O(n)$, using $O(n)$ words of storage, and such that the only randomization occurs in two calls of $\mathsf{RandDispl}_\epsilon$.*

3 Derandomizing double displacement

In this section we obtain a deterministic version of $\mathsf{RandDispl}_\epsilon$ running in time $O(n \log n)$. Corollary 3 then implies

Theorem 4 *For $w = O(\log n)$, an efficient perfect hash function for S can be constructed deterministically in time $O(n \log n)$, using $O(n)$ words of storage.*

Let us restate the problem solved in the the randomized part of $\mathsf{RandDispl}_\epsilon$ in a slightly more abstract way: Given a table T indexed by $\{0,1\}^\gamma$, and a set $A \subseteq \{0,1\}^\gamma$, find $d \in \{0,1\}^\gamma$ such that

$$\sum_{x \in A} T[x \oplus d] \le (1+\epsilon) 2^{-\gamma} |A| \sum_{i \in \{0,1\}^\gamma} T[i] \ . \tag{2}$$

The right hand side is $1+\epsilon$ times the expected value of the left hand side for $d \in \{0,1\}^\gamma$ chosen uniformly at random. We show how to find d deterministically in time $O(\gamma|A|)$, such that (2) holds with $\epsilon = 0$. Since $\gamma = \log n + O(1)$ in the algorithm, the time for finding a displacement value is $O(\log n)$ times the expected time in the randomized algorithm.

To find d efficiently we maintain an extension of T indexed by all bit strings of length at most γ. Let $p_\delta(j)$ denote the δ-bit prefix of $j \in \{0,1\}^\gamma$, and for $e \in \{0,1\}^\delta$ define $D_e = \{j \in \{0,1\}^\gamma \mid p_\delta(j) = e\}$, the bit strings of length γ with e as a prefix. The extension of T is defined by

$$T[i] = \sum_{j \in D_i} T[j] \ . \tag{3}$$

We can think of the extension of T as a binary trie whose leaves (indexed by strings of length γ) contain the original table entries, and where an internal node contains the sum over all leaves in its sub-trie. The extension can be initialized and maintained during n updates of the leaves with total work $O(\gamma n)$.

Starting with $e_0 = \varepsilon$, we will show how to find a sequence of bit strings $e_0, \ldots, e_\gamma$, where $e_k \in \{0,1\}^k$, such that the expected value of $\sum_{x \in A} T[x \oplus d]$ when choosing $d \in D_{e_k}$ uniformly at random is at most $2^{-\gamma}|A| \sum_{i \in \{0,1\}^\gamma} T[i]$. Then $d = e_\gamma$ is the displacement value sought. For e_0 the requirement is clearly met, so the key step is to extend e_k to e_{k+1} such that the expected value does not increase. By linearity of expectation we can always extend e_k by 0 or 1 such that this is the case. The right extension can be found because the expectations are easy to compute.

function Displ($S[0], \ldots, S[b-1] \subseteq \{0,1\}^\gamma$)
  Find permutation P such that
    $|S[P[j-1]]| \geq |S[P[j]]|$ for $1 \leq j < b$;
  for $k := 0$ **to** γ, $x \in \{0,1\}^k$ **do** $T[x] := 0$;
  for $i := 0$ **to** $b-1$ **do**
    $e := \varepsilon$;
    $E := 2^{-\gamma}|S[P[i]]|T[\varepsilon]$;
    for $k := 1$ **to** γ **do**
      if $\sum_{x \in S} T[p_k(x) \oplus (e \cdot 0)] \leq E$ **then** $e := e \cdot 0$
      else $e := e \cdot 1$;
    $D[P[i]] := e$;
    for $k := 0$ **to** γ, $x \in S[P[i]]$ **do**
      $T[p_k(x \oplus D[P[i]])] := T[p_k(x \oplus D[P[i]])] + 1$;
  end;
  return D;
end;

Figure 4: Derandomized displacement procedure

Lemma 5 *For any* $e \in \{0,1\}^\delta$, $\delta \leq \gamma$, *the expected value of* $\sum_{x \in A} T[x \oplus d]$, *when choosing* $d \in D_e$ *uniformly at random, is* $2^{\delta-\gamma} \sum_{x \in A} T[p_\delta(x) \oplus e]$.

Proof. For any x we have $\sum_{d \in D_e} T[x \oplus d] = T[p_\delta(x) \oplus e]$ by definition, so the expected value of $T[x \oplus d]$ when choosing $d \in D_e$ uniformly at random is $2^{\delta-\gamma} T[p_\delta(x) \oplus e]$. The lemma follows by linearity of expectation. □

Using the formula of the lemma, the γ bits of d can be found in time $O(\gamma|A|)$, as desired. Figure 4 shows the derandomized displacement procedure.

4 Universe reduction

In this section we show how to perform a *universe reduction* by finding a function $\rho : \{0,1\}^w \to \{0,1\}^r$, $r = O(\log n)$, which is 1-1 on S, can be stored in $O(1)$ words and is evaluable in constant time.

4.1 Previous results.

There are universal classes of hash functions [4] with range $\{0,1\}^{2\log n}$ from which suitable ρ can be found. Indeed, a constant fraction of the functions of such a class will fulfill the requirements, so a random search for a reduction function can be carried out in expected time $O(n)$. However, the best known deterministic search algorithm is that of Raman [13], running in time $O(n^2 w)$. A much faster search algorithm is that of Alon and Naor [1], but the resulting reduction function is not shown to be evaluable in constant time (and probably is not).

A novel approach to deterministic universe reduction, due to Miltersen [11], is the use of error-correcting codes. By applying an error-correcting code e, replacing each element $x \in U$ by $e(x) \in \{0,1\}^{4w}$, the Hamming distance between any two elements can be made $\Omega(w)$. It is then relatively easy to find a set D of $O(\log n)$ *distinguishing bit positions*, such that for any distinct elements $x, y \in S$, $e(x)$ and $e(y)$ differ on D. Using (bitwise) AND, the distinguishing bits can be filtered out. Exploiting word parallelism, Hagerup has shown how to find such distinguishing positions very efficiently [10, Lemma 3]. We summarize these results as follows:

Theorem 6 *(Miltersen, Hagerup) Suppose $e : \{0,1\}^w \to \{0,1\}^{4w}$ is a good error-correcting code. There exists $d \in \{0,1\}^{4w}$ with Hamming weight $O(\log n)$ such that $\rho_d : x \mapsto e(x)$ AND d is 1-1 on S. The bit string d can be computed from $\{e(x) \mid x \in S\}$ in time $O(n \log n)$ by a deterministic algorithm using $O(n)$ words of storage.*

In order to make ρ_d evaluable in constant time using standard instructions, Miltersen shows that a good error-correcting code can be implemented using multiplication: $e(x) = c_w \cdot x$, for suitable $c_w \in \{0,1\}^{3w}$ (the choice of which is a source of weak non-uniformity).

4.2 Gathering bits.

Miltersen does not address the issue of mapping injectively to $O(\log n)$ *consecutive* bits (which is what we would like), since he can work directly on the distinguishing bits where they appear. What we would like is to "gather" the distinguishing bits in an interval of $O(\log n)$ positions, using only standard instructions. For any $D \subseteq \{0, \ldots, r-1\}$, define $S_D = \{x \in \{0,1\}^r \mid x_i = 1 \Rightarrow i \in D\}$, the set of r-bit strings which have zeros outside the positions given by D.

Lemma 7 *Let $d \in \{0,1\}^{4w}$ have Hamming weight $O(\log n)$, and define $D = \{i \mid d_i = 1\}$. There is a function $\rho' : \{0,1\}^{4w} \to \{0,1\}^r$, $r = O(\log n)$ that is 1-1 on S_D and can be evaluated in constant time. The description of ρ' occupies $O(1)$ machine words and can be computed in time $o(n)$ by a deterministic algorithm using $o(n)$ words of storage.*

Proof. Without loss of generality we can assume $w = O(\sqrt[4]{n})$: If this is not the case, use the method of Fredman and Willard to gather the bits with positions in D within $O(\log^4 n)$ consecutive positions by multiplying with a suitable integer c_D [8, p. 428-429]. The algorithm for finding c_D uses the fact that the algorithm of Theorem 6 can be modified to output the bits of d separately. The running time is polylogarithmic.

Partition D into a constant number of sets $D_1, \ldots, D_k$ of size at most $\frac{1}{4}\log n$. Using the algorithm of Raman [13] we can find hash functions $\rho_1, \ldots, \rho_k$ with range $\{0,1\}^{\frac{1}{2}\log n}$, perfect for $S_{D_1}, \ldots, S_{D_k}$, in time $O((\sqrt[4]{n})^2 w) = o(n)$. Masking out the bits of $D_1, \ldots, D_k$, evaluating the hash functions and concatenating the results can be done in constant time. This defines the desired function ρ'. □

5 Results

Combining Theorem 6 and Lemma 7, we have the following result on universe reduction:

Lemma 8 *There is a function $\rho : \{0,1\}^w \to \{0,1\}^r$, $r = O(\log n)$, which is 1-1 on S and can be evaluated in constant time. The description of ρ occupies $O(1)$ machine words and can be computed in time $O(n \log n)$ by a weakly non-uniform, deterministic algorithm using $O(n)$ words of storage.*

Together with Theorem 4 this yields the main result:

Theorem 9 *An efficient perfect hash function for S can be constructed by a weakly non-uniform deterministic algorithm using time $O(n \log n)$ and $O(n)$ words of storage.*

6 Dynamization

Allowing randomization, the FKS static dictionary can be made dynamic, supporting insertions and deletions in amortized expected constant time [6]. Without a source of random bits, the task of simultaneously achieving fast updates and very fast queries seems considerably harder. Miltersen's static dictionary can be dynamized, giving update time $O(n^\epsilon)$ and lookup time $O(1)$. Hagerup uses the transformation result of [12] (Theorem A) to dynamize his static dictionary [10]. This gives a lookup time vs insertion time trade-off. We follow the same path and obtain a dynamic dictionary with an improved trade-off:

Theorem 10 *For any function $l(n) = O(\sqrt{\log n})$, where $l(n)$ is computable in time and space $O(n)$, there is a deterministic dynamic dictionary using $O(n)$ space, supporting lookups in time $O(l(n))$, insertions in time $O(n^{1/l(n)})$ and deletions in time $O(\log^{O(1)} n)$.*

The requirement $l(n) = O(\sqrt{\log n})$ is not essential, but gives a simpler expression for the insertion time. For $l(n) = \Omega(\sqrt{\log n / \log\log n})$ the data structure of Beame and Fich [3] is superior to any approach using static dictionaries as a black box.

It might be argued that insertion time $2^{\Omega(\sqrt{\log n})}$ makes Theorem 10 uninteresting. However, if the frequency of insertions compared to lookups is small, the total time for a sequence of operations may still be good. To study this asymptotically, we assume that for any sequence of n operations, starting with the empty dictionary, at most a fraction $1/f(n)$ of the operations are not lookups. For $f(n) = 2^{\Omega(\sqrt{\log n})}$ we can choose $l(n) = \log n / \log(f(n))$ in Theorem 10, making the total time for n operations $O(n \log n / \log(f(n)))$.

In fact, the trade-off between insertion time and lookup time in Theorem 10 is optimal for a rather general class of data structures based on hashing. Data structures in this class, described in [6], are rooted trees with at most one element in each of $O(n)$ leaves. Each inner node contains a hash function which maps from U to the children of the node. A search is guided by the hash functions in the obvious way. Rebuilding of the data structure is done by finding a new minimal perfect hash function for an entire subtree – this is assumed to take linear time in the size of the subtree. The lower bound of [6, Theorem 4.6] implies the following amortized lower bound for this class of "hash trees":

Theorem 11 *(Dietzfelbinger et al.) Consider sequences of n insertions into an initially empty hash tree, interleaved with arbitrary lookups. If, for some $l(n) = O(\sqrt{\log n})$, the total time for performing such insertions is $O(n^{1+1/l(n)})$, and if $|U| > (2n/l(n))^{l(n)}$, then in the worst case a lookup requires time $\Omega(l(n))$.*

It should be noted that the data structure of Theorem 10 does not fall into the class of hash trees. However, the amortized lower bound of Theorem 11 can be matched with hash trees, using our hash functions as a black box in the upper bound of [6, Theorem 4.6].

The best known randomized dictionaries are hash trees with the special property that the hash functions used when rebuilding are random. Further arguments that the restrictions on hash trees are reasonable can be found in [6]. However, data structures as the one in [3] show that in some cases there are ways to circumvent the lower bound.

The worst case time for deletions in Theorem 10 may be improvable, but in an amortized sense the deletions are free, since insertions dominate the total update time.

7 Conclusion and open problems

We have seen that the gap between deterministic and randomized algorithms for the static dictionary problem is rather small. If we take time $n \log^{O(1)} n$ (and linear space) as the paradigm for what is practical with massive data sets, this is the first construction algorithm for efficient dictionaries, which has an asymptotically practical worst-case behavior (where the worst case is taken over the input and any random choices). The dynamization of the dictionary is optimal for a natural class of "hash tree" data structures, and we argued that it is competitive with randomized dictionaries for skewed distributions on queries and updates.

Whether the gap between deterministic and randomized algorithms can be closed (at least for word lengths not too large) is the obvious open question. One approach, similar to the one used here, is to set the bits in the description of a perfect hash function one by one, each in constant time. Since a perfect hash function can be described in $O(n + \log w)$ bits, this would give a linear time algorithm except for large word lengths.

Another natural question is whether weak non-uniformity is necessary in order to deal with large word lengths. A discussion of weak non-uniformity can be found in [11].

An unpublished manuscript by Sundar [14] claims an amorized lower bound of $\Omega((\log\log n - \log\log w)/\log\log\log n)$ time per operation in a deterministic dynamic dictionary, so there is no hope of matching the expected performance of the best randomized dictionaries deterministically (even in the cell probe model). However, in the case where there are sufficiently more lookups than updates, there still is hope. Perhaps the most interesting possibility suggested by the results of this paper is a deterministic dynamic dictionary with logarithmic update time and very fast (constant time?) lookups. Such a data structure would be an attractive alternative to search trees.

Acknowledgments: The author would like to thank Gerth Stølting Brodal, Gudmund Skovbjerg Frandsen, Peter Bro Miltersen and Theis Rauhe for helpful discussions in connection with this work.

References

[1] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[2] Arne Andersson. Faster deterministic sorting and searching in linear space. In *37th Annual Symposium on Foundations of Computer Science (Burlington, VT, 1996)*, pages 135–141. IEEE Comput. Soc. Press, Los Alamitos, CA, 1996.

[3] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304, New York, 1999. ACM Press.

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.

[5] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246, Berlin, 1992. Springer-Verlag.

[6] 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.

[7] 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.

[8] Michael L. Fredman and Dan E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.

[9] Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98 (Paris, 1998)*, pages 366–398. Springer, Berlin, 1998.

[10] Torben Hagerup. Fast deterministic construction of static dictionaries. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999)*, pages 414–418, New York, 1999. ACM.

[11] Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 556–563, New York, 1998. ACM.

[12] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.*, 12(4):168–173, 1981.

[13] Rajeev Raman. Priority queues: small, monotone and trans-dichotomous. In *Proceedings of the 4th European Symposium on Algorithms (ESA '96)*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, Berlin, 1996.

[14] R. Sundar. A lower bound on the cell probe complexity of the dictionary problem. Manuscript, 1993.

[15] Robert Endre Tarjan and Andrew Chi Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

Recent BRICS Report Series Publications

RS-99-48 Rasmus Pagh. *Faster Deterministic Dictionaries*. December 1999. 14 pp. To appear in *The Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00 Proceedings, 2000.

RS-99-47 Peter Bro Miltersen and Vinodchandran N. Variyam. *Derandomizing Arthur-Merlin Games using Hitting Sets*. December 1999. 21 pp. Appears in Beame, editor, *40th Annual Symposium on Foundations of Computer Science*, FOCS '99 Proceedings, 1999, pages 71–80.

RS-99-46 Peter Bro Miltersen, Vinodchandran N. Variyam, and Osamu Watanabe. *Super-Polynomial Versus Half-Exponential Circuit Size in the Exponential Hierarchy*. December 1999. 14 pp. Appears in Asano, Imai, Lee, Nakano and Tokuyama, editors, *Computing and Combinatorics: 5th Annual International Conference*, COCOON '99 Proceedings, LNCS 1627, 1999, pages 210–220.

RS-99-45 Torben Amtoft. *Partial Evaluation for Designing Efficient Algorithms—A Case Study*. December 1999.

RS-99-44 Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. *Aliasing Models for Mobile Objects*. December 1999. To appear in a special FOOL '99 issue of *Information and Computation*.

RS-99-43 Uwe Nestmann. *What Is a 'Good' Encoding of Guarded Choice?* December 1999. To appear in a special EXPRESS '97 issue of *Information and Computation*. This revised report supersedes the earlier BRICS report RS-97-45.

RS-99-42 Uwe Nestmann and Benjamin C. Pierce. *Decoding Choice Encodings*. December 1999. To appear in *Journal of Information and Computation*.

RS-99-41 Nicky O. Bodentien, Jacob Vestergaard, Jakob Friis, Kåre J. Kristoffersen, and Kim G. Larsen. *Verification of State/Event Systems by Quotienting*. December 1999. 17 pp. Presented at *Nordic Workshop in Programming Theory*, Uppsala, Sweden, October 6–8, 1999.