Basic Research in Computer Science

BRICS RS-00-4 R. Pagh: A New Trade-off for Deterministic Dictionaries

A New Trade-off for Deterministic Dictionaries

Rasmus Pagh

BRICS Report Series RS-00-4
ISSN 0909-0878 February 2000

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/00/4/`

A New Trade-off for Deterministic Dictionaries

Rasmus Pagh
pagh@brics.dk

BRICS*
University of Aarhus
Denmark

Abstract

We consider dictionaries over the universe $U = \{0,1\}^w$ on a unit-cost RAM with word size w and a standard instruction set. We present a linear space deterministic dictionary with membership queries in time $(\log \log n)^{O(1)}$ and updates in time $(\log n)^{O(1)}$, where n is the size of the set stored. This is the first such data structure to simultaneously achieve query time $(\log n)^{o(1)}$ and update time $O(2^{\sqrt{\log n}})$.

1 Introduction

Among the most fundamental data structures is the *dictionary*. A dictionary stores a subset S of a universe U, offering membership queries of the form "$x \in S$?". The result of a membership query is either 'no' or a piece of *satellite data* associated with x. Updates of the set are supported via insertion and deletion of single elements.

Several performance measures are of interest for dictionaries: The amount of space used, the time needed to answer queries, and the time needed to perform updates. The most efficient dictionaries known depend on a source of random bits (are randomized, as opposed to deterministic). However, being randomized means that either: 1. There is a chance that the expected time

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

bounds do not hold, or 2. There is a chance of the data structure returning a wrong answer. In some situations, this may not be acceptable. Even if their use is acceptable, random bits may be expensive or unavailable. Finally, an understanding of the power of randomization is important from a theoretical point of view. All this has led to an interest in derandomization of known randomized algorithms and data structures. Several recent papers consider deterministic dictionaries [4, 11, 12, 13, 14]. However, previous space-efficient dictionaries with very fast lookups (time $(\log n)^{o(1)}$) have had update time much larger than that of, say, binary search trees. Therefore these dictionaries are of interest mainly when insertions are quite rare compared to lookups. Our interest here lies in obtaining space-efficient deterministic dictionaries which combine fast updates (time $(\log n)^{O(1)}$) with very fast lookups.

The model of computation used is a unit-cost *word RAM*, in which each memory register contains a w-bit integer (a *word*). This model of computation, resembling modern computers, has been the object of much recent research. Hagerup's survey [10] contains a more detailed definition. We adopt the *multiplication model* whose instruction set includes addition, bitwise boolean operations, shifts and multiplication. Note that all operations can also be carried out in constant time on arguments spanning a constant number of words. The universe considered is the set of machine words, $U = \{0, 1\}^w$. Such a RAM model, where a single unit of data fits into one machine word, is often referred to as *trans-dichotomous* [9]. For simplicity, we assume that each piece of satellite data occupies a single machine word (this could be a pointer to more bulky data). Throughout this paper, S will refer to an arbitrary set of n elements from U. All bounds will be independent of w, unless explicitly stated. Note that the optimal space consumption of a dictionary is $\Theta(n)$ words.

1.1 Related Work

The seminal result of Fredman, Komlós and Szemerédi [7] is that a *static* dictionary (i.e. without update operations) can have constant query time and linear space consumption. Allowing randomization, the FKS static dictionary can be made dynamic, supporting insertions and deletions in amortized expected constant time [4]. Improving this, Dietzfelbinger and Meyer auf der Heide [5] have constructed a dictionary in which all operations are done in constant time with high probability (i.e. probability at least $1 - n^{-c}$, where c is any constant of our choice). A simpler dictionary with the same properties was later developed [3]. As for randomized dictionaries, this leaves very little to be improved.

Without a source of random bits, the task of simultaneously achieving fast updates and constant query time seems considerably harder. The best deterministic dictionary with constant query time supports updates in time $O(n^\epsilon)$, for constant $\epsilon > 0$ [12]. The query time is $O(1/\epsilon^2)$. In fact, a range of trade-offs between update time an query time is known. For query time $O(q(n))$, where $q(n) = O(\sqrt{\log n})$, insertion time $O(n^{1/q(n)})$ and deletion time $O(\log^{O(1)} n)$ can be achieved [13]. The best known result in the situation where update and query time are considered equally important, is $O(\sqrt{\log n/ \log\log n})$ time per dictionary operation. It is a dynamization of the static data structure of Beame and Fich [2] using the exponential search trees of Andersson and Thorup [1].

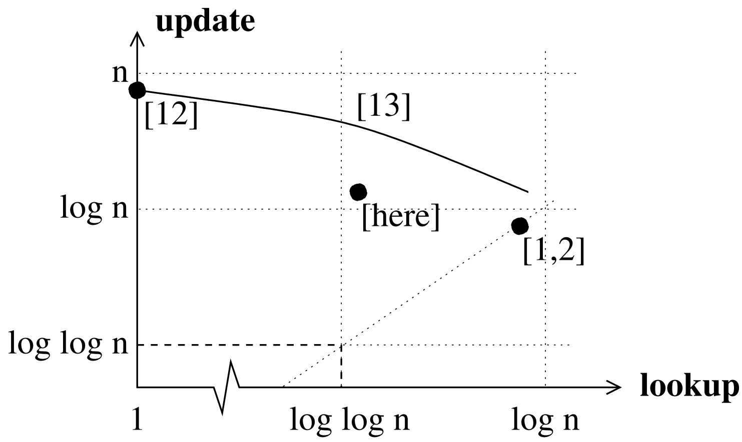

Figure 1: Overview of deterministic dictionaries using linear space.

The Beame-Fich-Andersson-Thorup (BFAT) data structure in fact supports *predecessor queries* of the form "What is the largest element of S not greater than x?". Its time bound improves significantly if the word length is not too large compared to $\log n$. For example, if $w = \log^{O(1)} n$, the time per operation is $O((\log\log n)^2/ \log\log\log n)$. This will be a key component in our construction.

An unpublished manuscript by Sundar [14] states an amortized lower bound of time $\Omega(\frac{\log\log_w n}{\log\log\log_w n})$ per operation for a deterministic dictionary in Yao's cell probe model [16], which in particular implies the same lower bound on the word RAM. Note that for $w = \log^{O(1)} n$, the BFAT data structure has time per operation polynomially related to the lower bound. The challenge therefore seems to be finding ways of dealing with large word length.

1.2 This Work

In this paper we obtain a dictionary with query time $O((\log\log n)^2/\log\log\log n)$ and amortized update time $O(\log^2 n)$ (we sketch how to make the latter bound worst-case). We deal with the problem of large word lengths by devising a dynamic *universe reduction* scheme, which reduces the problem to one within a smaller universe, which is then handled by the BFAT data structure. This is the first deterministic dictionary to simultaneously achieve query time $(\log n)^{o(1)}$ and update time $O(2^{(\log n)^c})$ for a constant $c < 1$. The data structure is *weakly non-uniform* in that it needs access to a fixed number of word-size constants depending (only) on w. These constants may be thought of as computed at "compile time".

In the following we assume that $w \geq \log^5 n$. Smaller word sizes can be handled using the BFAT data structure directly, and standard rebuilding techniques can be used to change from one data structure to the other. Similarly, we assume that n is larger than some fixed, sufficiently large constant, since constant size dictionaries are trivial to handle. We will look at machine words as binary numbers, with the most significant bits on the left and the least significant bits on the right. Bit positions are numbered from right to left, starting with zero.

2 Universe Reduction

Miltersen [12] has shown the utility of error-correcting codes to deterministic universe reduction. A *universe reduction function* $\rho : U \to U'$ translates the dictionary problem from universe U to the reduced universe U' (a search for x becomes a search for $\rho(x)$). The advantage of this is that U' may be smaller and easier to handle. Previous universe reduction functions for the static dictionary problem [7, 12, 13] have been 1-1 on S. In the dynamic case this appears hard to combine with efficient updates, and in our construction the reduction function is $O(\log n)$-1. That is, $O(\log n)$ elements of S may translate into the same element $\rho(x)$. A search among the elements "attached to $\rho(x)$" is then needed to establish whether $x \in S$.

2.1 Error-correcting Codes and Distinguishing Bits

Miltersen's approach plays a key role in our construction, so we review it here. The basic idea is to employ an error-correcting code $e : \{0,1\}^w \rightarrow \{0,1\}^{4w}$ and look at the dictionary problem for the transformed set $\{e(x) \mid x \in S\}$. For this it is possible to find a very simple function which is 1-1 on S, namely a projection onto $O(\log n)$ bit positions.

The code must have relative minimum distance bounded from 0 by a fixed positive constant, that is, there must exist a constant $\alpha > 0$ such that any two distinct codewords $e(x)$ and $e(y)$ have Hamming distance at least $\alpha \cdot 4w$ (the supremum of such constants is called the relative minimum distance of the code). We can look at the transformed set without loss of generality, since Miltersen has shown that such an error-correcting code can be computed in constant time using multiplication: $e(x) = c_w \cdot x$, for suitable $c_w \in \{0,1\}^{3w}$. The choice of c_w is a source of weak non-uniformity. The relative minimum distance for this code is greater than 1/11. In the following, α will denote a constant strictly smaller than the relative minimum distance of the error-correcting code (e.g. $\alpha = 1/11$).

Lemma 1 *(Miltersen) For any $R \subseteq U \times U$ there exists a* discriminating bit position $i \in \{0, \ldots, 4w-1\}$ *such that* $|\{(x,y) \in R \mid x \neq y,\, e(x)_i = e(y)_i\}| \leq (1-\alpha)\,|R|$.

Corollary 2 *(Miltersen) Let T be a set of m elements. There exists a set of* distinguishing bit positions $D \subseteq \{0, \ldots, 4w-1\}$ *with* $|D| < \frac{2}{\alpha} \log m$ *such that for all pairs of distinct elements $x, y \in S$, there is $i \in D$ where $e(x)_i \neq e(y)_i$. The set D can be constructed deterministically in time $O(m \log m)$, given a deterministic $O(m)$ time algorithm for finding a discriminating bit from the equivalence classes of an equivalence relation over T.*

Proof sketch. Elements of D may be found one by one, as discriminating bits of the equivalence relation where $x, y \in T$ are equal iff $e(x)$ and $e(y)$ do not differ on the bit positions already chosen. The number of pairs not distinguished decreases exponentially with the number of bit positions chosen. $\square$

Miltersen's universe reduction function is simply $x \mapsto e(x)$ AND d, where AND denotes bitwise conjunction and d is the incidence vector of D. The reduced universe U' consists of the $4w$-bit vectors which are zero outside the positions given by D.

Two problems remain: 1. We must show how to find discriminating bit positions in time $O(m)$. 2. We want the reduction function to map to $O(\log m)$ *consecutive* bits, that is, to $\{0,1\}^{O(\log m)}$. The first problem was solved by Hagerup [11]. We need the following slight extension of his result to also solve the second problem:

Lemma 3 *(Hagerup) Given a set T of m elements, divided into equivalence classes, a discriminating bit position i can be found in time $O(m)$ by a deterministic, weakly non-uniform algorithm. Further, for any set $I \subseteq \{0, \ldots, 4w-1\}$ of size $O(\log^4 n)$ (given as a bit vector), we can assure that $i \notin I$.*

Proof sketch. It can be shown how to compute $|\{\{x,y\} \subseteq T \mid x \neq y,\, x \equiv y,\, e(x)_i = e(y)_i\}|$ for all $i \in \{0, \ldots, 4w-1\}$ in time $O(m)$. The algorithm employs word-level parallelism, and the result vector spans $O(\log m)$ words, since each number occupies $O(\log m)$ bits. Word-parallel binary search can be used to find the smallest entry. To avoid entries in I, simply overwrite the entries of I with the largest possible integer before finding the minimum. This corresponds to changing the error-correcting code to be constant (i.e. non-discriminating) on the bit positions of I. Since $|I| = O(\log^4 n)$ and the length of codewords is $4w \geq 4\log^5 n$, the relative minimum distance of this modified code is still $> \alpha$ (for n large enough). Hence, this procedure will find a discriminating bit position. $\square$

2.2 Multiple Set Universe Reduction

To accommodate efficient updates, we will not maintain a set of distinguishing bit positions for S itself. Instead, we maintain $k = \lceil \log(n+1) \rceil$ sets of distinguishing bit positions $D_0, \ldots, D_{k-1}$ for subsets $S_0, \ldots, S_{k-1}$ whose (disjoint) union is S. A change of set i and recomputation of D_i, should take time $O(s \log s)$, where s is the size of the new set. Further, the complete set of distinguishing bit positions should be *well separated*, that is, no pair of positions should differ by less than $2c \log^2 n$, where c is a suitably large constant. By the results of Sect. 2.1, this can indeed be achieved, in such a way that $|D_i| = O(\log n)$.

Since the distinguishing bit positions are well separated, we are able to "collect" and order the distinguishing bits within $O(\log^2 n)$ consecutive bit positions, such that the distinguishing bits of S_0 are least significant, and the distinguishing bits of S_{k-1} are most significant. For each empty set S_i we will have $O(\log |S_i|)$ zero-bits. The following lemma makes this precise.

Lemma 4 *Given a list $d_1, \ldots, d_p$ of well separated bit positions, where $p \leq c \log^2 n$, there is a function $f_{\bar{d}} : \{0,1\}^{4w} \mapsto \{0,1\}^p$ such that for any x, $f_{\bar{d}}(x)_i = x_{d_i}$. The function can be evaluated in constant time, and updated under changes of bit positions in constant time.*

Proof. We will show how to "move" bit d_i of $x \in \{0,1\}^{4w}$ to bit $u+i$ of a $u+p$-bit string, where $u \geq \max_i d_i$ (the desired value can then be obtained by shifting the word by u bits). We simply multiply x by $m_{\bar{d}} = \sum_i 2^{u+i-d_i}$ (a method adopted from [8, p. 428-429]). One can think of the multiplication as p shifted versions of x being added. Note that if there are no carries in this addition, we do indeed get the right bits moved to $u+1, \ldots, u+p+1$. However, since the bit positions are well separated, all carries occur either left of the $u+p$th position (which is harmless) or right of position $u-p$ (which can never influence the values at positions greater than u, since there are more than enough zeros in between to swallow all carries). Note that $m_{\bar{d}}$ can be updated in constant time when a bit position changes. $\square$

We are now ready to describe how to update the dynamic universe reduction function under updates. We keep the invariant that $|S_i| \in \{0, 2^i\}$. New elements are inserted in the lowest numbered empty set S_i together with the elements of $S_0, \ldots, S_{i-1}$ (these sets are then empty). Note that the work per element when constructing a new set of distinguishing positions is $O(\log n)$. Since elements are always transferred to higher numbered sets, the total amortized work for an insertion is $O(k \log n) = O(\log^2 n)$. As we will see in the next section, this cost will be dominant in the cost of an insertion in the final dictionary.

The universe reduction function will not be updated during deletions. Rather, deletions are implemented by simply *marking* deleted elements in the dictionary. When more than half of the elements in the dictionary are marked, a new dictionary containing the unmarked elements is constructed. The cost of this is amortized over the deletions, which hence also have cost $O(\log^2 n)$.

3 Using the Predecessor Data Structure

Recall that our universe reduction function, which we will call ρ, computes the concatenation of functions $f_k, \ldots, f_0$ which are 1-1 on $S_k, \ldots, S_0$, respectively. The value $\rho(x)$ after x is inserted in S_i is used as key for accessing x in the BFAT predecessor data structure. Functions $f_0, \ldots, f_{i-1}$ return zero vectors at this time. However, these functions will change in the period until the next update of S_i, and specifically $f_0(x), \ldots, f_{i-1}(x)$ may change. When a search for $\rho(x)$ is conducted, the result will be either the BFAT key for x, or that of a key y later inserted, whose BFAT key agrees with that of x except possibly for some of the values of $f_0, \ldots, f_{i-1}$. In this case we want x to be present in y's associated (sorted) list of elements. That is, for each new key $\rho(y)$ in the BFAT data structure, we want a list of elements which includes $x \in S_i$ iff x and y agree on $f_k, \ldots, f_i$.

A predecessor query on $\rho(y) - 1$ will return the BFAT key which has the longest common prefix with y (if any). By invariant, the associated list of this key contains all the elements needed, apart from y itself, so it is easy to create the list associated with y. The crux is that, since $f_k, \ldots, f_0$ are 1-1, an associated list can contain at most one element from each set.

Example. We go through figure 2. This example has 3, 4 and 5 distinguishing bit positions for S_0, S_1 and S_2, respectively. The keys inserted in the BFAT data structure are annotated with their list of elements. At $t = 4$ the dictionary contains four elements, denoted a, b, c, d, all residing in S_2. At $t = 5$ element e is inserted and put into S_0. The key for e coincides with the key for c on the first five bits, so the associated list contains c and e. A search for the key of c at this time would in fact find 00111 0000 000, so c is not strictly necessary in the new list. However, at $t = 6$ element f enters, and S_1 is filled by e and f. After this, a search for the key of c will find 00111 0010 000, and c can be found in the new list. At $t = 7$ element g is inserted, and its key coincides with both the first five bits of c's key and the first nine bits of e's key, so the associated list becomes ceg.

3.1 Time and Space

A search for x requires computation of $\rho(x)$ in constant time, a predecessor lookup in time $O((\log\log n)^2/\log\log\log n)$ and finally search of an associated list in time $O(\log\log n)$. That is, the total time is $O((\log\log n)^2/\log\log\log n)$.

	t=4	t=5	t=6	t=7
	00010 0000 000 a	00010 0000 100	00010 0001 000	00010 0001 010
	00110 0000 000 b	00110 0000 110	00110 1000 000	00110 1000 000
S_2	00111 0000 000 c	00111 0000 001	00111 1100 000	00111 0101 011
	11011 0000 000 d	11011 0000 110	11011 0010 000	11011 0010 001
S_1			00111 0010 000 **ce**	00111 0010 101
			11111 1101 000 **f**	11111 1101 100
S_0		00111 0000 011 **ce**		10111 0010 011 **ceg**

Figure 2: Universe reduction function values for elements in S during three insertions.

As for insertions, remember that the cost of maintaining the universe reduction function is $O(\log n)$ for each element that is moved to a new set (this happens $O(\log n)$ times per element). We only need to argue that the cost of maintaining the associated lists is no larger. This is not hard to see, since all that is needed is a single predecessor query and insertion of an element in a sorted list of length $O(\log n)$.

The only part of the data structure which is not clearly in linear space is the set of associated lists, where elements may occur $\log n$ times. To see that their total length is $O(n)$, note that there can be no more than $n/2^{i-1}$ lists of length i, since such lists must have been created in connection with insertion of elements in $S_0, \ldots, S_{k+1-i}$.

4 Final Remarks

4.1 Speedups

If the requirement of linear space is abandoned, substituting van Emde Boas trees [15] for the BFAT data structure gives membership queries in time $O(\log \log n)$. The space usage then rises to $n^{O(\log n)}$.

Updates can be sped up slightly, to time $O(\log^2 n / \log \log n)$, by using another strategy, in which there are $\Theta(\log n)$ sets of each size, and only $O(\log n / \log \log n)$ different set sizes.

It can be noted that the predecessor data structure is used in such a way that it essentially answers "longest common prefix" queries on strings of length $k+1$, where the characters are described by the bits corresponding to sets $S_k, \ldots, S_0$, respectively. A plausible way of improving the query time to, say, $O(\log\log n)$ is by designing a faster data structure which can find such longest common prefixes.

4.2 Worst-case Bounds

We gave amortized bounds. The same worst-case bounds follow by standard lazy rebuilding techniques, to be sketched below. Where the amortized insertion algorithm would "build" S_i and empty $S_{i-1}, \ldots, S_0$, the *worst-case insertion* algorithm keeps $S_{i-1}, \ldots, S_0$ in memory and starts building S_i at a pace of $c \log n$ steps per insertion (for some sufficiently large constant c). Only when S_i is completed, we throw out the lower numbered sets.

More precisely, we now have sets $S_{i,j}$ for $0 \leq j < i \leq k$, where $|S_{i,j}| \in \{0, 2^j\}$. The first index signifies that $S_{i,j}$ will next become part of a new set of size 2^i. Consider insertion number $2^b d - 2^a$, where $a < b$ (any positive integer can be written like this for unique integers a, b and d). At this point we start constructing $S_{b,a}$ from the new element and $S_{a,0}, \ldots, S_{a,a-1}$. As the last stage of the construction, we set $S_{a,0} = \cdots = S_{a,a-1} = \emptyset$. Constant c above can be chosen such that this is guaranteed to be finished before any of the sets $S_{i,0}, \ldots, S_{i,i-1}$ are to be reconstructed. The ordering of distinguishing bits is with respect to primarily the first index, secondarily the second index.

Since we need associated element lists of length $\Omega(\log^2 n)$, we cannot afford to use sorted lists as before (updates would become more expensive). Instead, we use persistent balanced search trees [6], which support updates and queries in time $O(\log t)$ for a sequence of trees of size at most t. One technicality is that many instances of the algorithm finding distinguishing bits have to run at the same time and must produce well separated bit positions. However, since positions are chosen one by one, this poses no problem. In addition to what is done in the amortized case, the *worst-case deletion* algorithm inserts two elements of S in a new dictionary. When the transfer of all elements in S is completed, the new dictionary takes the place of the old one. Of course, transferred elements may be deleted before the new dictionary takes over.

5 Conclusion

We have seen a new lookup time vs insertion time trade-off for linear space deterministic dictionaries. This presents progress towards closing the gap between known upper and lower bounds. It also shows that universe reduction techniques have a place not only in the static setting.

The big open question is whether updates in such a dictionary can be accommodated in time $(\log n)^{o(1)}$. For example, time $(\log \log n)^{O(1)}$ would mean that Sundar's lower bound is tight up to a polynomial. For $w = (\log n)^{O(1)}$ this is achieved by the BFAT data structure. Thus, large word length seems to be the main enemy, and new universe reduction schemes with faster updates appear a promising approach.

Acknowledgments: The author would like to thank Rolf Fagerberg and Jakob Pagter for useful feedback.

References

[1] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing (STOC 2000)*, New York, 2000. ACM Press.

[2] 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.

[3] 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.

[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] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In

Automata, languages and programming (Coventry, 1990), pages 6–19. Springer, New York, 1990.

[6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. System Sci.*, 38(1):86–124, 1989. 18th Annual ACM Symposium on Theory of Computing (Berkeley, CA, 1986).

[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] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994. Appeared at the 31st Annual Symposium on Foundations of Computer Science (FOCS '90).

[10] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, pages 366–398. Springer, Berlin, 1998.

[11] 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.

[12] 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.

[13] Rasmus Pagh. Faster deterministic dictionaries. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 487–493, New York, 2000. ACM.

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

[15] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (Berkeley, Calif., 1975)*, pages 75–84, Long Beach, Calif., 1975. IEEE Computer Society.

[16] Andrew Chi Chih Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.

Recent BRICS Report Series Publications

RS-00-4 **Rasmus Pagh.** ***A New Trade-off for Deterministic Dictionaries.*** **February 2000.**

RS-00-3 **Fredrik Larsson, Paul Pettersson, and Wang Yi.** ***On Memory-Block Traversal Problems in Model Checking Timed Systems.*** **January 2000. 15 pp. To appear in** ***Tools and Algorithms for The Construction and Analysis of Systems: 6th International Conference*****, TACAS '00 Proceedings, LNCS, 2000.**

RS-00-2 **Igor Walukiewicz.** ***Local Logics for Traces.*** **January 2000. 30 pp.**

RS-00-1 **Rune B. Lyngsø and Christian N. S. Pedersen.** ***Pseudoknots in RNA Secondary Structures.*** **January 2000. 15 pp. To appear in** ***Fourth Annual International Conference on Computational Molecular Biology*****, RECOMB '00 Proceedings, 2000.**

RS-99-57 **Peter D. Mosses.** ***A Modular SOS for ML Concurrency Primitives.*** **December 1999. 22 pp.**

RS-99-56 **Peter D. Mosses.** ***A Modular SOS for Action Notation.*** **December 1999. 39 pp. Full version of paper appearing in Mosses and Watt, editors,** ***Second International Workshop on Action Semantics*****, AS '99 Proceedings, BRICS Notes Series NS-99-3, 1999, pages 131–142.**

RS-99-55 **Peter D. Mosses.** ***Logical Specification of Operational Semantics.*** **December 1999. 18 pp. Invited paper. Appears in Flum, Rodríguez-Artalejo and Mario, editors,** ***European Association for Computer Science Logic: 13th International Workshop*****, CSL '99 Proceedings, LNCS 1683, 1999, pages 32–49.**

RS-99-54 **Peter D. Mosses.** ***Foundations of Modular SOS.*** **December 1999. 17 pp. Full version of paper appearing in Kutyłowski, Pacholski and Wierzbicki, editors,** ***Mathematical Foundations of Computer Science: 24th International Symposium*****, MFCS '99 Proceedings, LNCS 1672, 1999, pages 70–80.**

RS-99-53 **Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen.** ***Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL.*** **December 1999. 9 pp.**