Alcom-FT Technical Report Series
ALCOMFT-TR-01-68

Nordic Journal of Computing

A TRADE-OFF FOR WORST-CASE EFFICIENT DICTIONARIES

RASMUS PAGH*
BRICS†, *Department of Computer Science,*
University of Aarhus, DK–8000 Århus C, Denmark
pagh@brics.dk

Abstract. We consider dynamic dictionaries over the universe $U = \{0,1\}^w$ on a unit-cost RAM with word size w and a standard instruction set, and present a linear space deterministic dictionary accommodating 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. Previous solutions either had query time $(\log n)^{\Omega(1)}$ or update time $2^{\omega(\sqrt{\log n})}$ in the worst case.

CR Classification: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Sorting and searching; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

Key words: Deterministic dynamic dictionary, membership, worst-case, trade-off

1. Introduction

The *dictionary* is among the most fundamental data structures. It supports maintenance of a set S under insertion and deletion of elements, called *keys*, from a universe U. Data is accessed through membership queries, "$x \in S$?". In case of a positive answer, the dictionary also returns a piece of *satellite data* that was associated with x when it was inserted.

Dictionaries have innumerable applications in algorithms and data structures. They are also interesting from a foundational point of view, as they formalize the basic concept of information retrieval from a "corpus of knowledge" (an associative memory). The main question of interest is: What are the computational resources, primarily time and space, needed by a dictionary? Our interest lies in how good *worst-case* bounds can be achieved. For the sake of simplicity one usually assumes that keys are bit strings in $U = \{0,1\}^w$, for a positive integer parameter w, and restricts attention to the case where keys of U fit a single memory location. Memory locations are referred to as *words*, and w is called the *word length*. Each piece of satellite data is assumed to be a single word, which could be a pointer to more

*Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
†Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

Received July 2000. Revised October 2000.

bulky data. This means that the best space complexity one can hope for is $O(n)$ words, where n is the size of S. We will only consider data structures using $O(n)$ space. Motivated by applications in which queries are somewhat more frequent than updates, our focus is on providing very fast membership queries, while maintaining fast updates.

In comparison-based models the complexity of dictionary operations is well understood. In particular, queries require $\Omega(\log n)$ comparisons in the worst case, and we can implement all dictionary operations in $O(\log n)$ time on a pointer machine using only comparisons [Adel'son-Vel'skii and Landis 1962]. However, for a model of computation more resembling real-world computers, a unit cost RAM with words of size w, the last decade has seen the development of algorithms "blasting through" comparison-based lower bounds [Fredman and Willard 1993, Andersson 1996, Beame and Fich 1999, Andersson and Thorup 2000], resulting in a dictionary with time $O(\sqrt{\log n / \log\log n})$ for all operations. Other authors have found ways of combining very fast queries with nontrivial update performance. Most notably, Miltersen [1998] achieves constant query time together with update time $O(n^\epsilon)$ for any constant $\epsilon > 0$. It is important to note that all bounds in this paper are *independent of* w, unless explicitly stated otherwise.

All previous schemes have had either query time $(\log n)^{\Omega(1)}$ or update time $2^{\omega(\sqrt{\log n})}$. In this paper we obtain the following trade-off between query time and update time:

THEOREM 1. *There is a deterministic dictionary running on a unit cost RAM with word length w that, when storing n keys of w bits, uses $O(n)$ words of storage, supports insertions and deletions in $O((\log n \log\log n)^2)$ time, and answers membership queries in $O((\log\log n)^2 / \log\log\log n)$ time.*

Ignoring polynomial differences, our dictionary has exponentially faster queries or exponentially faster updates than any previous scheme.

1.1 Model of Computation

The *word RAM* model used is the same as in the earlier work mentioned above. We only briefly describe the model, and refer to the "multiplication model" in [Hagerup 1998] for details.

The RAM operates on words of size w, alternately looked upon as bit strings and integers in $\{0, \ldots, 2^w - 1\}$. It has the following computational operations: bitwise boolean operations, shifts, addition and multiplication. Note that all operations can also be carried out in constant time on arguments spanning a constant number of words.

Word RAM algorithms are allowed to be *weakly nonuniform*, i.e., use a constant number of word-size constants depending only on w. These constants can be thought of as computed at "compile time". In this paper, weak nonuniformity could be replaced by an extended instruction set, with certain natural but nonstandard instructions from uniform NC^1.

1.2 Related Work

As mentioned, the best known worst-case bound holding simultaneously for all dictionary operations is $O(\sqrt{\log n/\log\log n})$. It is achieved by a dynamization of the static data structure of Beame and Fich [1999] using the exponential search trees of Andersson and Thorup [2000]. This data structure, from now on referred to as the BFAT data structure, in fact supports *predecessor queries* of the form "What is the largest key of S not greater than x?". Its time bound improves significantly if the word length is not too large compared to $\log n$. In particular, if $w = (\log n)^{O(1)}$ the bound is $O((\log\log n)^2/\log\log\log n)$. These properties of the BFAT data structure will play a key role in our construction.

An unpublished manuscript by Sundar [1993] states an amortized lower bound of time $\Omega(\frac{\log\log_w n}{\log\log\log_w n})$ per operation in the cell probe model of Yao [1981]. In particular, this implies the same lower bound on the word RAM. Note that for $w = (\log n)^{O(1)}$, the BFAT data structure has time per operation polynomially related to the lower bound.

A seminal result of Fredman *et al.* [1984] is that in the *static* case (with no updates to the data structure), one can achieve *constant* query time. A recent line of research has brought about a dictionary with constant query time, that can be constructed in time $O(n \log n)$ [Miltersen 1998, Hagerup 1999, Pagh 2000]. A standard dynamization then gives, for any "nice" function $q(n) = O(\sqrt{\log n})$, query time $O(q(n))$ and update time $O(n^{1/q(n)})$.

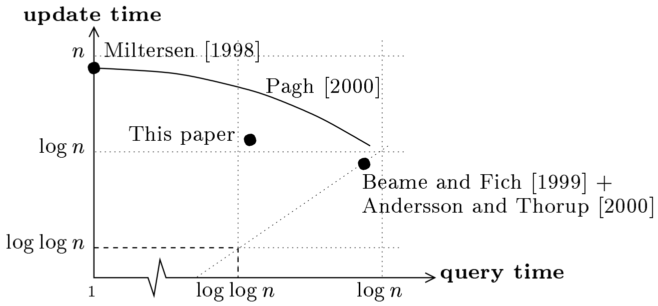

Fig. 1.1: Overview of trade-offs for linear space dictionaries.

If one abandons the requirement of good worst-case performance, new possibilities arise. Most notably, one can consider amortized and randomized (expected) time bounds. Using the universal hash functions of Carter and Wegman [1979] for chained hashing, one obtains expected constant time for all operations. The query time can be made worst-case constant by dynamizing the static dictionary of Fredman *et al.* to allow updates in amortized expected constant time [Dietzfelbinger *et al.* 1994]. Subsequent works have described dictionaries in which every operation is done in constant time with high probability [Dietzfelbinger and Meyer auf der Heide 1990, Dietzfelbinger *et al.* 1992, Willard 2000]. The best result with no dependence on w is, for any constant c, a success probability of $1 - n^{-c}$ [Dietzfelbinger *et al.* 1992].

For a general introduction to dynamic data structures, covering both amortized and worst-case bounds, we refer to the book of Overmars [1983].

1.3 Overview

As mentioned, the BFAT data structure is very fast when the word length is $(\log n)^{O(1)}$. Our strategy is to reduce the dictionary problem to a predecessor problem on words of length $(\log n)^{O(1)}$, solved by the BFAT data structure. A query for x translates into a query for $h(x)$ in the predecessor data structure, where h is an efficiently evaluable *universe reduction function*, mapping U to a (smaller) universe $\{0,1\}^r$. This approach is similar to universe reduction schemes previously employed in the static setting [Miltersen 1998, Pagh 2000]. One difference is that we use $r = \Theta(\log^2 n)$ rather than $r = \Theta(\log n)$. Moreover, in the dynamic setting we need to dynamically update h when new keys are inserted, which means that $h(x)$ may change for keys $x \in S$. This can be handled, however, as a predecessor query for $h(x)$ may still find the BFAT key for x if it is smaller than $h(x)$. In general there may be other keys between the BFAT key of x and $h(x)$. However, we maintain the invariant that if $x \in S$ then x can be found in a sorted list of $O(\log n)$ keys associated with the predecessor of $h(x)$.

After some preliminary observations and definitions, Section 3 introduces the tools needed for the universe reduction. In Section 4 we show how to construct a dictionary with the desired properties, except that the time bound for updates is amortized. Finally, Section 5 describes how to extend the technique of the amortized data structure to make the bound worst-case.

2. Preliminaries

2.1 Simplifications

Without loss of generality, we may disregard deletions and consider only insertions and membership queries. Deletions can be accommodated within the same time bound as insertions, by the standard technique of *marking* deleted keys and periodically *rebuilding* the dictionary. To be able to mark a key, we let the satellite information point to a marker bit and the "real" satellite information. For the rebuilding one maintains two insertion-only dictionaries: An active one catering insertions and membership queries, and an inactive one to which two new unmarked keys from the active dictionary are transferred each time a key is marked (in both dictionaries). When all unmarked keys have been transferred, the inactive dictionary is made active, and we start over with an empty inactive dictionary. This assures that no more than a constant fraction of the keys are marked at any time. So if the insertion-only dictionary uses linear space, the space usage of the above scheme is $O(n)$.

We may also assume that $w \geq \log^5 n$. Word size $O(\log^5 n)$ 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.

2.2 Notation and Definitions

Throughout this paper S refers to a set of n keys from U. When we need to distinguish between values before and after a change of the data structure, we used primed variables to denote the new values. We will look at bit strings also as binary numbers with the most significant bits first. The ith last bit of a string x is denoted by x_i; in particular, $x = x_l x_{l-1} \dots x_1$, where l is the length of x. We say that x is the *incidence string* of $D \subseteq \{1, \dots, l\}$ if $i \in D \Leftrightarrow x_i = 1$, for $1 \leq i \leq l$. The *Hamming weight* of x is the number of positions i where $x_i = 1$. The *Hamming distance* between strings x and y is the number of positions i where $x_i \neq y_i$. For clarity, we will distinguish between keys of our dictionary and keys in predecessor data structures, by referring to the latter as *p-keys*. The set of positive integers is denoted by $\mathbb{N}$.

3. Universe Reduction Tools

Miltersen [1998] has shown the utility of error-correcting codes to deterministic universe reduction. This approach plays a key role in our construction, so we review it here. For further details we refer to [Hagerup *et al.* 2000], which sums up all the results and techniques needed here. The basic idea is to employ an error-correcting code $\psi : \{0,1\}^w \to \{0,1\}^{4w}$ (which is fixed and independent of S) and look at the transformed set $\{\psi(x) \mid x \in S\}$. For this set it is possible to find a very simple function that is 1-1 and has small range, 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 $\delta > 0$ such that any two distinct codewords $\psi(x)$ and $\psi(y)$ have Hamming distance at least $4w\delta$. The infimum 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: $\psi(x) = c_w \cdot x$, for suitable $c_w \in \{0,1\}^{3w}$. The choice of c_w is a source of weak nonuniformity. The relative minimum distance of this code is greater than $\delta = 1/25$.

DEFINITION 1. *For an equivalence relation $\equiv$ over $T \subseteq U$, a position $d \in \{1, \dots, 4w\}$ is* discriminating *if, for $K = \{\{x,y\} \subseteq T \mid x \neq y \wedge x \equiv y\}$, $|\{\{x,y\} \in K \mid \psi(x)_d = \psi(y)_d\}| \leq (1-\delta)\,|K|$.*

For $T \subseteq U$, a set $D \subseteq \{1, \ldots, 4w\}$ is distinguishing *if, for all pairs of distinct keys $x, y \in T$, there exists $d \in D$ where $\psi(x)_d \neq \psi(y)_d$.*

Miltersen's universe reduction function (for a set T) is $x \mapsto \psi(x)$ AND v, where AND denotes bitwise conjunction and v is the incidence string of a distinguishing set for T. A small distinguishing set can be found efficiently by finding discriminating bits for certain equivalence relations:

LEMMA 1. *(Miltersen) Let $T \subseteq U$ be a set of m keys, and suppose there is an algorithm that, given the equivalence classes of an equivalence relation over T, computes a discriminating position in time $O(m)$. Then a distinguishing set for T of size less than $\frac{2}{\delta} \log m$ can be constructed in time $O(m \log m)$.*

PROOF SKETCH. Elements of the distinguishing set may be found one by one, as discriminating positions of the equivalence relation where $x, y \in T$ are equal if and only if $\psi(x)$ and $\psi(y)$ do not differ on the positions already chosen. The number of pairs not distinguished by the first k discriminating positions is at most $(1-\delta)^k \binom{m}{2}$. □

Hagerup [1999] showed how to find a discriminating position in time $O(m)$. We will need a slight extension of his result.

LEMMA 2. *(Hagerup) Given a set $T \subseteq U$ of m keys, divided into equivalence classes of a relation $\equiv$, a discriminating position d can be computed in time $O(m)$. Further, for an auxiliary input string $b \in \{0,1\}^{4w}$ of Hamming weight $o(w)$ and w larger than some constant, we can assure that $b_d = 0$.*

PROOF SKETCH. In [Hagerup 1999] it is shown how to employ word-level parallelism to compute $|\{\{x, y\} \subseteq T \mid x \neq y, x \equiv y, \psi(x)_d = \psi(y)_d\}|$ for all $d \in \{1, \ldots, 4w\}$ in time $O(m)$. The algorithm computes a vector of $O(\log m)$-bit numbers compressed into $O(\log m)$ words.

Word-parallel binary search can be used to find the index of the smallest entry, which will be discriminating. To avoid positions where $b_d = 1$, we replace the corresponding entries of the vector 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 indicated by b. Since b has Hamming weight $o(w)$, the relative minimum distance of this modified code is still greater than α, for n large enough. Hence, this procedure will find a discriminating bit position. □

DEFINITION 2. *A set of positions $\{d_1, \ldots, d_p\} \subseteq \{1, \ldots, 4w\}$ is* well separated *if $|d_i - d_j| \geq 2p$ for all i, j where $1 \leq i < j \leq p$.*

We will keep the distinguishing positions used for the universe reduction well separated. This is done by forming a vector b with 1s in positions within some distance of previously chosen positions, and using this in the algorithm of Lemma 2. Good separation allows us to "collect" distinguishing bits into consecutive positions (in arbitrary order):

LEMMA 3. *For a list $d_1, \ldots, d_p$ of well separated positions, there is a function $f_{\bar{d}} : \{0,1\}^{4w} \to \{0,1\}^p$ that can be stored in a constant number of words and evaluated in constant time, such that for any $x \in \{0,1\}^{4w}$ and $i \in \{1, \ldots, p\}$, we have $f_{\bar{d}}(x)_i = x_{d_i}$. The function description can be updated in constant time when the list is extended (with position d_{p+1}) and under changes of positions, given that the resulting list is well separated.*

PROOF. We will show how to "move" bit d_i of $x \in \{0,1\}^{4w}$ to bit $4w + i$ of a $(4w + p)$-bit string. The desired value can then be obtained by shifting the string by $4w$ bits. We first set all bits outside $\{d_1, \ldots, d_p\}$ to 0 using a bit mask. Then simply multiply x by $M_{\bar{d}} = \sum_{i=1}^{p} 2^{4w+i-d_i}$ (a method adopted from Fredman and Willard [1993, 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 positions $4w+1, \ldots, 4w+p$. However, since the positions are well separated, all carries occur either left of the $4w + p$th position (and this has no impact on the values at positions $4w+1, \ldots, 4w+p$) or right of position $4w-p$ (and this can never influence the values at positions greater than $4w$, 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 position is added or changed. □

4. Dictionary with Amortized Bounds

This section presents the main ideas allowing the universe reduction techniques of Section 3 and a predecessor data structure to be combined in an efficient dynamic (insertion-only) dictionary with *amortized* bounds. Section 5 will extend the ideas to achieve nearly the same bounds in the worst case.

We will start by outlining how the query algorithm is going to work. A membership query for x proceeds by first computing the value $h(x)$ of the universe reduction function h, described in Section 4.1. Then we search for the p-key κ that is predecessor of $h(x)$ (if it does not exist, the search is unsuccessful). Finally, we search for x in a list A_κ associated with κ. The invariant is kept that if $x \in S$, then x is found in this way.

4.1 The Universe Reduction Function

We will not maintain a distinguishing set for S itself, but rather maintain $k = \lceil \log(n+1) \rceil$ distinguishing sets $D_1, \ldots, D_k$ for a partition of S into subsets $S_1, \ldots, S_k$. Let $h_{D_i} : U \to \{0,1\}^{|D_i|}$ denote a function "collecting" the bits in positions of D_i from its error-corrected input, i.e., such that for each $d \in D_i$ there is $j \in \{1, \ldots, |D_i|\}$ where $h_{D_i}(x)_j = \psi(x)_d$. The universe reduction function is:

$$h : x \mapsto h_{D_k}(x) \circ 1 \circ h_{D_{k-1}}(x) \circ 1 \circ \cdots \circ h_{D_1}(x) \circ 1 \qquad (4.1)$$

where $\circ$ denotes concatenation. The basic idea is that set S_i and distinguishing set D_i changes only once every 2^i insertions. This means that during most insertions, function values change only in the least significant bits. Hence, for many keys $x \in S$, a query for the predecessor of $h(x)$ will return the same p-key before and after a change of h. In cases where a new p-key becomes predecessor of $h(x)$, we explicitly put x in the list of keys associated with the p-key. Details follow below.

The following invariants are kept:

(1) $|S_i| \in \{0, 2^i\}$, for $1 \le i \le k$.

(2) $|D_i| = 2i/\delta$, for $1 \le i \le k$.

(3) $|d_1 - d_2| \ge 4k^2/\delta$, for all distinct $d_1, d_2 \in D_1 \cup \cdots \cup D_k$.

The first invariant implies that $|S|$ must be even, but this is no loss of generality since insertions may be processed two at a time. Invariant (3) is feasible because of the lower bound on w. By invariant (2) the size of $D_1 \cup \cdots \cup D_k$ is at most $2k^2/\delta$, so invariant (3) implies that $D_1 \cup \cdots \cup D_k$ is well separated. Lemma 3 then gives that h can be evaluated in constant time and updated in time $O(\sum_{i=1}^{k} 2i/\delta) = O(\log^2 n)$ when $D_1, \ldots, D_k$ changes. Within this time bound we can also compute a string $b \in \{0,1\}^{4w}$ of Hamming weight $O(\log^4 n) = o(w)$, such that all bits of b with distance less than $4\,(k+1)^2/\delta$ to a position in $D_1, \ldots, D_k$ are 1s: Multiply the incidence string of $D_1 \cup \cdots \cup D_k$ by $2^{4\,(k+1)^2/\delta} - 1$ (whose binary expansion is the string $1^{4\,(k+1)^2/\delta}$) to get a string v, and compute the bitwise or of v and v right-shifted $4\,(k+1)^2/\delta$ positions.

We now describe how to update $S_1, \ldots, S_k$ and $D_1, \ldots, D_k$ when new keys are inserted. Let m denote the smallest index such that $S_m = \emptyset$, or let $m = k + 1$ if no subset is empty. When new keys x and y are inserted, we compute a distinguishing set D for $S_{m-1} \cup \cdots \cup S_1 \cup \{x, y\}$. By Section 3 this can be done such that $|D| = 2m/\delta$, adding extra positions if necessary. Also, using the string b we can assure that positions in D have distance at least $4\,(k+1)^2/\delta$ from positions in $D_1, \ldots, D_k$ as well as from each other. We then perform the following updates: $S'_m = S_{m-1} \cup \cdots \cup S_1 \cup \{x, y\}$, $D'_m = D$, and $S'_{m-1} = \cdots = S'_1 = \emptyset$. The invariants are easily seen to remain satisfied.

To see that the amortized time per insertion is $O(\log^2 n)$, note that no key at any time has been part of more than k distinguishing set computations, and that each such computation took time $O(\log n)$ per key.

4.2 Using the Predecessor Data Structure

The predecessor data structure has one p-key for each key in S. For $x \in S_i$ the key is

$$\kappa_x = h_{D_k}(x) \circ 1 \circ h_{D_{k-1}}(x) \circ 1 \circ \cdots \circ h_{D_i}(x) \circ 1 \circ 0^{z_i} \qquad (4.2)$$

where z_i is the number of bits in $h_{D_{i-1}}(x) \circ 1 \circ \cdots \circ h_{D_1}(x) \circ 1$. Note that $\kappa_x \leq h(x) < \mu_x$, where $\mu_x = \kappa_x + 2^{z_i}$. We also observe that, since h_{D_i} is 1-1 on S_i, p-keys belong to unique keys of S, and that κ_x and μ_x are fixed during the time where $x \in S_i$. Associated with κ_x is a sorted list of the following keys, each represented by a pointer to a unique instance:

$$A_{\kappa_x} = \{y \in S \mid \kappa_y \leq \kappa_x < \mu_y\} \ . \tag{4.3}$$

For $y \in S_j$, the condition $\kappa_y \leq \kappa_x < \mu_y$ holds if and only if κ_x has prefix $h_{D_k}(y) \circ 1 \circ \cdots \circ h_{D_j}(y) \circ 1$. Since h_{D_j} is 1-1 on S_j, it follows that for $x \in S_i$, the set A_{κ_x} consists of at most one key from each of $S_k, \ldots, S_i$. Also, A_{κ_x} is constant during the time where $x \in S_i$. To see that our query algorithm is sound, note that for $x \in S$ the search for a predecessor of $h(x)$ returns a p-key κ with $\kappa_x \leq \kappa \leq h(x) < \mu_x$.

When $S_1, \ldots, S_k$ and $D_1, \ldots, D_k$ have changed, we must update the predecessor data structure accordingly. We first delete the old p-keys of keys in $S'_m = S_1 \cup \cdots \cup S_{m-1}$. For each key $x \in S'_m$ we then compute the new p-key κ'_x, and its predecessor is searched for. If κ'_x has no predecessor, then its associated list must contain only x. If there is a predecessor κ'_y, the associated list consists of x plus a subset of $A_{\kappa'_y}$. To see this, recall that no key from $S'_m \backslash \{x\}$ can be in $A_{\kappa'_x}$, and that by invariant $A_{\kappa'_y} = \{v \in S' \mid \kappa'_v \leq \kappa'_y < \mu'_v\}$. Specifically, we have $A_{\kappa'_x} = \{v \in A_{\kappa'_y} \mid \kappa'_x < \mu'_v\} \cup \{x\}$. Thus, in time $O(\log n)$ per key we can create and insert p-keys and associated lists for all $x \in S'_m$. Again, this means that the amortized cost of updating the predecessor data structure is $O(\log^2 n)$.

Apart from the predecessor query, the time used by the query algorithm is $O(\log k) = O(\log \log n)$. Hence, if we use the BFAT predecessor data structure, the time to perform a query is $O((\log \log n)^2 / \log \log \log n)$. The only part of our data structure not immediately seen to be in linear space is the set of associated lists. For $x \in S_i$, the set A_{κ_x} contains at most $k+1-i$ keys, so the total length of all associated lists is $O(\sum_{i=1}^{k} (k+1-i)\, 2^i) = O(n\,(1 + \sum_{i=1}^{k} (k-i)\, 2^{-(k-i)})) = O(n)$.

Proposition 1. *There is a deterministic dictionary that, when storing a set of n keys, uses $O(n)$ words of storage, supports insertions and deletions in amortized time $O(\log^2 n)$, and answers membership queries in time $O((\log \log n)^2 / \log \log \log n)$.*

5. Dictionary with Worst-Case Bounds

Whereas the dictionary described in the previous section efficiently performs any sequence of operations, the worst-case time for a single insertion is $\Omega(n \log n)$ (this happens when we compute a distinguishing set for S_k, which has size $\Omega(n)$). We now describe a similar dictionary that has the worst-case time bounds stated in Theorem 1. Focus is on the aspects different from the amortized case. In particular, the description must be read together with Section 4 to get the full picture.

The worst-case dynamization technique used is essentially that of Overmars and van Leeuwen [1981]. By maintaining the partition of S slightly differently than in Section 4, it becomes possible to start computation of distinguishing sets for future partitions early, such that a little processing before each insertion suffices to have the distinguishing sets ready when the partition changes. Similarly, predecessor data structure updates can be done "in advance", leaving little work to be done regardless of whether an insertion triggered a small or a large change in the partition.

The partition of S now involves $2k$ sets, $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, where $|S_i|, |T_i| \in \{0, 2^i\}$. When two nonempty sets with the same index i arise, computation of a distinguishing set for $S_i \cup T_i$ is initiated, proceeding at a pace of $O(\log n)$ steps per insertion (this is the first part of what will be referred to as the "processing at level i"). At a designated time after this computation has finished, either S_{i+1} or T_{i+1} is replaced by $S_i \cup T_i$.

We use two predecessor data structures containing the p-keys of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, respectively. The tricky part is to update the p-keys in these data structures such that the properties of the amortized scheme are preserved. For example, we have to make sure that old p-keys do not interfere with searches during the time it takes to delete them. In the following we describe only the universe reduction function and predecessor data structure for p-keys of $S_1, \ldots, S_k$, as everything is completely symmetric with respect to switching the roles of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$.

5.1 The Universe Reduction Function

The universe reduction function is again (4.1), and invariants (1), (2) and (3) are still kept. However, updates are performed in a different manner. We describe the way in which $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$ are updated; the corresponding distinguishing sets are implicitly updated accordingly. Keys are inserted in pairs. Just before the pth pair is inserted, from now on referred to as "time step p", we spend $O(\log n\, (\log \log n)^2)$ time on processing at each level i for which S_i and T_i are both nonempty. The processing has two parts: computation of a distinguishing set for $S_i \cup T_i$, followed by insertions and updates of p-keys in one of the predecessor data structures (the latter part is described in Section 5.2). Processing at level i starts at time step $2^i z + 2^{i-1}$, for $z \in \mathbb{N}$, and proceeds for 2^{i-2} time steps (we make sense of half a time step by considering each time step to have two subparts).

At time step $2^i(z+1)-1$, for $z \in \mathbb{N}$, we then perform an update by moving the keys of $S_i \cup T_i$ to level $i+1$, i.e., either S_{i+1} or T_{i+1} is set to $S_i \cup T_i$. Note that the distinguishing set for $S_i \cup T_i$ has been computed at this point (in fact, already at time step $2^i(z+1) - 2^{i-2}$). If z is even, processing is about to begin at level $i+1$, and $S_i \cup T_i$ replaces the empty set at level $i+1$. Otherwise $T_{i+1} \cup S_{i+1}$ is either empty or about to move to level $i+2$, and we arbitrarily replace one of S_{i+1} and T_{i+1} by $S_i \cup T_i$ and the other by $\emptyset$

(this is done at all levels simultaneously). At even time steps, the new pair replaces the empty set at level 1. At odd time steps, we arbitrarily replace one of T_1 and S_1 by the newly inserted pair and the other by $\emptyset$.

An important property of the scheme described above is, that even though processing occurs at k levels simultaneously, distinguishing sets that are going to appear together in the universe reduction function are computed one at a time. Specifically, consider the update at time step $2^i z - 1$, where $z > 1$ is odd. Here, new distinguishing sets for $S_1 \cup T_1, \ldots, S_i \cup T_i$ are about to become part of the universe reduction function. The distinguishing set for $S_j \cup T_j$, $j \leq i$, was computed from time step $2^i z - 2^{j-1}$ to time step $2^i z - 2^{j-2} - 1$, for $1 \leq j \leq i$. Distinguishing sets at level $i+1$ and above were computed well before time step $2^i z - 2^{i-1}$. We may thus use the method of Section 4 to keep invariant (3) satisfied.

5.2 Using the Predecessor Data Structure

The predecessor data structure (for $S_1, \ldots, S_k$) still contains the p-key κ_x of equation (4.2) for each $x \in S_1 \cup \cdots \cup S_k$. Such p-keys are called *active*. Additionally, there will be *previous* p-keys (that used to be active but whose corresponding key is no longer in $S_1 \cup \cdots \cup S_k$) and *future* p-keys (that will become active after an upcoming change of the partition). As a consequence, there may be up to two keys corresponding to each p-key. We consider p-keys with two corresponding keys as *two* p-keys; in particular, a p-key can be both future and active, or both active and previous.

What follows is explained more easily if we switch to the language of strings. Specifically, we will look at a p-key of the form $q_k \circ 1 \circ q_{k-1} \circ 1 \circ \cdots \circ q_i \circ 1 \circ 0^{z_i}$, where $q_j \in \{0,1\}^{2j/\delta}$, as a string consisting of the characters $q_k q_{k-1} \ldots q_i$. We refer to i as the *level* of the p-key. In particular, for $x \in S_i$ the level of κ_x is i. Similarly, the range of the universe reduction function is looked upon as a set of strings.

At any level we maintain the invariant that there cannot be both previous p-keys and future p-keys in the predecessor data structure. Also, future p-keys are distinct and so are previous p-keys. This means that each p-key can have at most two corresponding elements: one for which it is the current p-key, and one for which it is a previous or future p-key.

The list associated with a p-key κ now contains up to $2k$ keys, including:

$$A_\kappa = \{y \in S_1 \cup \cdots \cup S_k \mid \kappa_y \text{ is a prefix of } \kappa\} \ . \tag{5.1}$$

Membership queries are performed by searching the associated list of the predecessor of $h(x)$ (in both predecessor data structures). The predecessor κ of $h(x)$ will be one of the p-keys having the longest common prefix with $h(x)$. Thus, A_κ contains all keys whose p-keys are prefixes of $h(x)$. In particular, if $x \in S_1 \cup \cdots \cup S_k$ the p-key κ_x is a prefix of $h(x)$, so x is found in the associated list.

Now consider the processing initiated at level i at time step $2^i z + 2^{i-1}$, $z \in \mathbb{N}$. As described earlier, the processing spans 2^{i-2} time steps, so the computation of a distinguishing set for $S_i \cup T_i$ is completed before time step $2^i(z+1) - 2^{i-2}$. At time step $2^i(z+1) - 1$ an update will replace a set at level $i+1$ by $S_i \cup T_i$, and hence (active) p-keys at level $i+1$ must be present for $S_i \cup T_i$ (in the predecessor data structure for $S_1, \dots, S_k$ if $S_i \cup T_i$ replaces S_{i+1}, which we may assume by symmetry). A crucial point is that the future p-keys are known as soon as the distinguishing set for $S_i \cup T_i$ has been computed. This is because any changes to distinguishing sets at level $i+2$ and above that will take effect at time step $2^i(z+1) - 1$ were determined before time step $2^i z + 2^{i-1}$.

Part of the processing is to insert the future p-keys at level $i+1$. Future p-keys that will become active at time step $2^i(z+1) - 1$ and are above level $i+1$, were inserted before time step $2^i(z+1) - 2^{i-1}$. For each new p-key $\kappa = q_k q_{k-1} \dots q_{i+1}$ we look up any keys corresponding to prefixes q_k, $q_k q_{k-1}, \dots, q_k q_{k-1} \dots q_{i+1}$, thus finding the up to $2(k-i+1)$ keys that are in A_κ *before and after* time step $2^i(z+1) - 1$. The keys are sorted and put into the list associated with κ, and κ is inserted in the predecessor data structure. In the same way we compute new associated lists for the active p-keys at level $i+1$. The dominant part of the processing time is used for the $O(2^i \log n)$ lookups in the predecessor data structure. Hence, the total time is $O(2^i \log n \, (\log\log n)^2)$, which amounts to $O(\log n \, (\log\log n)^2)$ time per insertion.

At time step $2^i(z+1) - 1$ the active p-keys at level i change status to previous, and the future p-keys at level $i+1$ change status to active. During the following 2^{i-1} time steps the previous p-keys at level i are deleted. This means that there are no previous p-keys at time step $2^i(z+1) + 2^{i-1}$ when processing begins again at level i. The time per insertion for the deletions at level i is negligible.

6. Open problems

An interesting open question is whether both queries and updates in a linear space deterministic dictionary can be accommodated in time $(\log n)^{o(1)}$. For example, a bound of $(\log\log n)^{O(1)}$ would mean that Sundar's lower bound is tight up to a polynomial (not considering upper bounds dependent on w). For $w = (\log n)^{O(1)}$ the bound is achieved by the BFAT data structure. Thus, large word length seems to be the main enemy, and new dynamic universe reduction schemes with faster updates appear a promising approach.

Acknowledgements: The author is grateful to Gerth Stølting Brodal for suggesting the use of a simpler worst-case dynamization technique, and to the anonymous referees for helpful suggestions improving the presentation.

References

Adel'son-Vel'skii, Georgii Maksimovich and Landis, Evgenii Mikhailovich. 1962. An Algorithm for Organization of Information. *Dokl. Akad. Nauk SSSR 146*, 263–266.

ANDERSSON, ARNE. 1996. Faster Deterministic Sorting and Searching in Linear Space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*. IEEE Comput. Soc. Press, Los Alamitos, CA, 135–141.

ANDERSSON, ARNE AND THORUP, MIKKEL. 2000. Tight(er) Worst-Case Bounds on Dynamic Searching and Priority Queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*. ACM Press, New York, 335–342.

BEAME, PAUL AND FICH, FAITH. 1999. Optimal Bounds for the Predecessor Problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*. ACM Press, New York, 295–304.

CARTER, J. LAWRENCE AND WEGMAN, MARK N. 1979. Universal Classes of Hash Functions. *J. Comput. System Sci. 18*, 2, 143–154.

DIETZFELBINGER, MARTIN, GIL, JOSEPH, MATIAS, YOSSI, AND PIPPENGER, NICHOLAS. 1992. 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*. Springer-Verlag, Berlin, 235–246.

DIETZFELBINGER, MARTIN, KARLIN, ANNA, MEHLHORN, KURT, MEYER AUF DER HEIDE, FRIEDHELM, ROHNERT, HANS, AND TARJAN, ROBERT E. 1994. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput. 23*, 4, 738–761.

DIETZFELBINGER, MARTIN AND MEYER AUF DER HEIDE, FRIEDHELM. 1990. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*. Volume 443. Springer-Verlag, Berlin, 6–19.

FREDMAN, MICHAEL L., KOMLÓS, JÁNOS, AND SZEMERÉDI, ENDRE. 1984. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *J. Assoc. Comput. Mach. 31*, 3, 538–544.

FREDMAN, MICHAEL L. AND WILLARD, DAN E. 1993. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. System Sci. 47*, 424–436.

HAGERUP, TORBEN. 1998. Sorting and Searching on the Word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*. Volume 1373 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 366–398.

HAGERUP, TORBEN. 1999. Fast Deterministic Construction of Static Dictionaries. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*. ACM Press, New York, 414–418.

HAGERUP, TORBEN, MILTERSEN, PETER BRO, AND PAGH, RASMUS. 2000. Deterministic Dictionaries. *Submitted for journal publication.*

MILTERSEN, PETER BRO. 1998. Error Correcting Codes, Perfect Hashing Circuits, and Deterministic Dynamic Dictionaries. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*. ACM Press, New York, 556–563.

OVERMARS, MARK H. 1983. The Design of Dynamic Data Structures Volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

OVERMARS, MARK H. AND VAN LEEUWEN, JAN. 1981. Dynamization of Decomposable Searching Problems Yielding Good Worst-Case Bounds. In *Proceedings of the 5th GI-Conference*. Volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 224–233.

PAGH, RASMUS. 2000. Faster Deterministic Dictionaries. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*. ACM Press, New York, 487–493.

SUNDAR, RAJAMANI. 1993. A Lower Bound on the Cell Probe Complexity of the Dictionary Problem.

WILLARD, DAN E. 2000. Examining Computational Geometry, van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree. *SIAM J. Comput. 29*, 3, 1030–1049 (electronic).

YAO, ANDREW CHI-CHIH. 1981. Should Tables be Sorted? *J. Assoc. Comput. Mach. 28*, 3, 615–628.