Dynamic Hashing in Real Time[1]

Martin Dietzfelbinger[2]
Friedhelm Meyer auf der Heide[2]

Fachbereich 17 · Mathematik – Informatik
und Heinz-Nixdorf-Institut
Universität-GH-Paderborn,
4790 Paderborn
Germany

Abstract

A dynamic hashing scheme that performs in real time is presented. It uses linear space and needs worst case constant time per instruction. Thus instructions can be given in constant length time intervals. Answers to queries given by the algorithm are always correct, the space bound is always satisfied. The algorithm is of the Monte Carlo type; it fails to meet the real time assumption only with probability $O(n^{-c})$, where n is the number of data items currently stored. The constant c can be chosen arbitrarily large.

Further, a parallel version of this dictionary for a p-processor CRCW PRAM is described, where in constant length time intervals p instructions are given via the p processors. For dictionaries of size $n \geq p^{1+\varepsilon}$, the same performance bounds as for the sequential case are obtained.

The construction is based on a new high performance universal class of hash functions.

[1] A preliminary version of parts of the material in this paper has been presented at ICALP 90

[2] Supported in part by DFG Grant ME 872/1-4

J. Buchmann et al. (eds.), *Informatik*

1 Introduction

A *dictionary* is a data structure that supports three kinds of instructions, namely "Insert x", "Delete x", and "Lookup x". Here the possible *keys* x are taken from some finite *universe* U. This is to be understood as follows: the instruction "Insert x" causes x together with some information associated with x to be stored in the data structure, the instruction "Delete x" causes x to be removed from the data structure; the instruction "Lookup x" returns the information associated with x (or a default message if x is currently not stored). We always assume that the user that provides the instructions waits for one instruction to be processed before entering the following one.

We will be interested in time requirements (for single instructions as well as amortized over a sequence of instructions) and space requirements for implementations of such a data structure.

Classical implementations of dictionaries are dynamic search trees such as AVL-trees, 2-3-trees, etc. (cf. [17]). They need $\Theta(\log n)$ time per instruction for a dictionary of size n.

A dynamic hashing strategy with *constant* time per instruction on the average and linear space was presented in [1]. Here the average is taken over all input sequences. A significant improvement was shown in [4]. Here the expected amortized time per instruction is constant, where the time bound holds for each sequence of instructions; the expected value is taken over all random choices of the randomized algorithm. A scheme with similar features was described in [3]. The main difference is that lookups take worst case constant time in [4] and expected constant time in [3].

Dynamic hashing in real time The main result of the present paper is an implementation for an optimal dictionary that essentially has worst case constant time per instruction and needs linear space. The algorithm performs in real time and is probabilistic of the Monte Carlo type; that means, there is a certain (small) probability that the scheme fails. Note that the probability space underlying the analysis is induced by the random choices the algorithm makes; no assumptions are made concerning the distribution of the keys occurring in the instructions. We formulate the result in more detail.

Theorem 1 *There is a probabilistic (Monte Carlo type) algorithm for implementing a dictionary with the following features. Let $c \geq 1$ be constant.*

(i) *At any time, the space used is proportional to the number of keys currently stored in the dictionary.*

(ii) *Performing one lookup takes constant time in the worst case.*

(iii) *Insertions and deletions take constant time per operation.*

(iv) *The assertions* (i) *and* (ii) *always hold; the probability that* (iii) *is violated during a sequence of $\frac{1}{2}n$ instructions executed in a dictionary of size n is $O(n^{-c})$.*

(The constants in the time bounds and the space bound depend on c.)

High performance hash functions The construction of the dictionary is based on a dynamic hashing strategy that uses a novel type of hash functions. We introduce a new class $\mathcal{R}$ of high performance hash functions that can be evaluated in constant time but share many properties of n-universal hash functions, i. e., hash functions that map n keys independently, uniformly into $\{0,\ldots,n-1\}$. The most striking properties of the new class are the following. For each $S \subseteq U$, $|S| \leq n$, there is $\mathcal{R}_S \subseteq \mathcal{R}$ such that:

(*) Given S, a randomized construction of a random $h \in \mathcal{R}_S$ and the corresponding search table can be done in time $O(n)$ with high probability. The construction of h needs time $O(n^\varepsilon)$, where $0 < \varepsilon \leq 1$ may be arbitrary.

(**) For a random $h \in \mathcal{R}_S$ and a fixed $j \in \{0,\ldots,n-1\}$, $\Pr(|\{x \in S \mid h(x) = j\}| \geq u)$ is exponentially small in u.

(***) $E\Big(\max\Big\{|\{x \in S \mid h(x) = j\}|, 0 \leq j < n\Big\}\Big) = O(\log n / \log\log n)$.

Previous work on high performance hash functions Polynomials of degree up to $d-1$ as hash functions are used e. g. in [8], [4], [5] and many other papers. Such a random polynomial h is only d-wise independent, that means, the random variables $h(x_1),\ldots,h(x_d)$ are independent for any d distinct keys $x_1,\ldots,x_d \in U$, but not for larger sets of keys. If we demand constant evaluation time, thus take d to be constant, we get much weaker properties than mentioned above; for example, if polynomials of degree up to $d-1$ are used, one can only prove $E\Big(\max\Big\{|\{x \in S \mid h(x) = j\}|, 0 \leq j < n\Big\}\Big) = O(n^{1/d})$.

In [19], a construction of a class of n^η-wise independent hash functions is given that can be evaluated in constant time and constructed in time $O(n^\varepsilon)$, where $0 < \eta < \varepsilon < 1$. This class has similar properties as (**) and (***) from above. The constants in the time and space bounds for that class depend exponentially on r, where n^r is the size of the universe.

Parallel dynamic hashing in real time We further present a parallel dictionary implemented on a CRCW PRAM with ARBITRARY write conflict resolution. We assume the following consistency rules: If several processors want to access key x, first the deletions, then the insertions, and finally the lookups are executed. If several processors try to insert the same key x in the same round, an arbitrary one of these instructions is performed.

Theorem 2 *There is a probabilistic (Monte Carlo type) algorithm for implementing a parallel dictionary on a CRCW PRAM with p processors with the following features. Let $c \geq 1$, $\varepsilon > 0$ be arbitrary constants.*

(i) *If n keys are stored in the dictionary, then space $O(\max\{n, p^{1+\varepsilon}\})$ is occupied.*

(ii) *Each instruction needs worst case constant time, i. e., p instructions, one at each processor, can simultaneously be input in constant time intervals.*

(iii) *The algorithm fails to meet* (ii) *with probability at most n^{-c}, during the execution of $\frac{1}{2}n$ instructions in a dictionary of size n.*

Previously known parallel dictionaries are dynamic perfect hashing schemes that guarantee worst case constant lookup time but only amortized constant update time, see [5]. Very recently, a parallel dictionary with comparable features as ours was presented in [9]. It already works in optimal space for a dictionary of size p. On the other hand, it can only be executed with $p/\log^*(p)$ processors, executing p instructions in optimal time $O(\log^*(p))$, whereas our dictionary allows more parallelism: p processors execute p instructions in constant time. The construction in [9] generalizes static parallel hashing schemes as presented in [2] and [15].

Applications of the new dictionary (a) *Distributed dictionaries.* Assume that each one of p processors maintains a dictionary, working independently. It is very useful to have a bound for the time the slowest processor needs to perform n instructions. The best estimate that could be obtained by using schemes available before was $O(n \log p)$ (expected). With our scheme, the probability that all dictionaries execute each instruction in constant time is at least $1 - O(p \cdot n^{-c})$. Sequential dictionaries that operate as evenly as that are indispensable when constructing efficient distributed dictionaries (cf. [6] for details of such techniques).

(b) *"Clocked adversaries".* It is shown in [14] that several schemes for implementing dynamic hashing strategies by use of chaining are susceptible to attacks by adversaries that are able to time the algorithm. Even though the adversary (who chooses the instructions) does not have access to the random numbers the algorithm chooses, he can draw conclusions about the structure of the hash function being used from the time the execution of certain instructions takes, and subsequently choose keys to be inserted that make the algorithm perform badly. Lipton and Naughton ask whether the algorithm of [4] is susceptible to such attacks. This question is still open; but the algorithm described in the present paper is immune to such attacks in a natural way: Normally, each instruction takes constant time, which gives no information to the adversary at all. From time to time the algorithm may crash; but the consequence is that the data structure is built anew, redoing all random choices. This results in a data structure that is independent of all previous events, in particular of the situation that caused the crash.

Structure of paper In Section 2, some basic definitions are given and the technical background for our construction is reviewed. Section 3 contains several probability bounds useful for the analysis of the performance of our hash functions and dictionaries. Section 4 presents the new hash functions; Section 5 the new sequential dictionary, and Section 6 its parallelization.

A more complicated version of the sequential dictionary and a slightly different version of the new hash functions are described in [7].

2 Background: Polynomials as hash functions

Our constructions will be based on the following universal class of hash functions: Let p be prime, $U = \{0, 1, \ldots, p-1\}$ be the *universe*. Consider two parameters: $d \geq 2$, the *degree*, and $s \geq 1$, the *table size*. Define

$$\mathcal{H}_s^d := \{\, h_\alpha \mid \alpha = (\alpha_0, \ldots, \alpha_{d-1}) \in U^d \,\},$$

where for $\alpha = (\alpha_0, \ldots, \alpha_{d-1}) \in U^d$ we let

$$h_\alpha(x) := \Big(\sum_{0 \leq i < d} \alpha_i x^i \bmod p \Big) \bmod s, \qquad \text{for } x \in U.$$

We will have h chosen uniformly at random from $\mathcal{H}_s^d$. All probabilities are with respect to this probability space; no assumptions about the distribution of the input keys are made. We give some basic definitions and recall two useful facts. Assume for the following that some set $S \subseteq U$ with $|S| = n$ is given.

Definition 1 *Let $h : U \to \{0, \ldots, s-1\}$.*

(a) *The jth* bucket *is $B_j^h := \{\, x \in S \mid h(x) = j \,\}$.*

(b) *Its size is $b_j^h := |B_j^h|$.*

(c) *The set of keys colliding with $x \in U$ is $B_x^{\mathrm{coll},h} := \{y \in S \mid h(y) = h(x)\}$.*

(d) *The number of keys colliding with $x \in U$ is $b_x^{\mathrm{coll},h} := |B_x^{\mathrm{coll},h}|$.*

(e) *h is called* l-perfect for S *if $b_i^h \leq l$ for all $i \in \{0, \ldots, s-1\}$.*

(f) *If h is chosen at random from some class, we write B_j for the random set B_j^h and b_j for the random variable b_j^h, analogously for B_x^{coll} and b_x^{coll}.*

Fact 1 ([4]) *Let $n \leq s$. For each d there is a constant c_d (which can be assumed to be smaller than $\frac{1}{2}$) so that for all $S \subseteq U$ with $|S| = n$ and all h randomly chosen from $\mathcal{H}_s^d$ we have*

$$\Pr\big(h \text{ is } (d-1)\text{-perfect for } S\big) \geq 1 - c_d \cdot n \cdot (n/s)^{d-1}.$$

We immediately apply this fact to obtain highly reliable real-time dictionaries in the case where much more space is available than the number of keys to be stored.

Fact 2 *Let $0 < \varepsilon < 1$, let $d \geq 2$, and let n be given. There is a dictionary that uses $d \cdot n$ space and that with probability exceeding $1 - O(n^{\varepsilon-(1-\varepsilon)(d-1)})$ executes a sequence of n^ε instructions in such a way that each one takes constant time (in the worst case).*

Proof: Use a hash table of size $s = n$, each table position comprising d slots for one key each. By Fact 1, a hash function h chosen randomly from $\mathcal{H}_n^d$ maps at most $d-1$ of the n^ϵ keys to each of the table positions, with the claimed probability. Note that if a sequential list of length n^ϵ is maintained in which all keys are recorded that were ever entered in the dictionary, the memory space used can be cleared in $O(n^\epsilon)$ steps after finishing the n^ϵ instructions. ∎

In a different situation, namely where the hash table size s is much smaller than the number n of keys occurring, the following result tells us that with high probability all buckets have size close to the average. (This fact will be used for the construction of the parallel dictionary.)

Fact 3 ([13, Section 4]) *Assume that $S \subseteq U$ with $|S| = n$ is given, and that $s = n^{1-\delta}$ for some fixed δ, $0 < \delta < 1$. Let $h \in \mathcal{H}_s^d$ be randomly chosen. Then*

$$\Pr(b_j \leq 2n^\delta \text{ for } 0 \leq j < n) \geq 1 - \alpha(d) \cdot n^{1-\delta-\delta d/2},$$

where $\alpha(d) = (d/2)^{d+1}$ is constant.

Finally, we sketch the famous construction of Fredman, Komlós, and Szemerédi [8] that provides a *static* dictionary with constant lookup time in the worst case. (The construction of the real-time dictionary given below is based on this prototype.) Assume a set $S \subseteq U$ of n keys is given. A level-1 hash function $h: U \to \{0, \ldots, s-1\}$ splits S into the buckets $B_j^h = \{x \in S \mid h(x) = j\}$, for $0 \leq j < s$. For each of the buckets B_j^h there is a secondary hash table ST_j of size $2|B_j^h|^2$ and a level-2 hash function $h_j: U \to \{0, \ldots, 2|B_j^h|^2 - 1\}$ that is one-to-one on B_j^h. The element $x \in S$ is stored in position $h_j(x)$ of subtable ST_j, where $j = h(x)$. Access to the subtables and to (the programs for) the functions h_j is facilitated by a primary hash table HT with s entries. For $0 \leq j < s$, entry $HT[j]$ of this table contains a pointer to ST_j and a description of h_j. In order to construct (probabilistically) such a structure that takes space $O(n) = O(|S|)$, one chooses functions h and h_j at random from suitable classes $\mathcal{H}_s^2$. (See [8] for details.)

3 Probability bounds

In this section, we first quote a classical theorem that estimates the probability for a sum of independent *bounded* random variables to deviate far from the expected value. (For an overview over variants of this result, see [11].) Next, we prove a theorem in the same spirit dealing with sums of independent random variables that are approximately geometrically distributed, thus *unbounded*. Both estimates will be used in the next section for establishing the main properties of a new class of hash functions to be defined there. For the analysis of the real-time dictionary algorithm we need another type of estimate for the tails of the distribution of variables that depend on many independent random variables, each of which has little influence. Such an estimate is provided by the "martingale tail theorem" quoted at the end of this section.

Theorem 3 ([12]) *Let $X_1, \ldots, X_n$ be independent random variables with values in the interval $[0, z]$ for some $z > 0$. Let $Y := \sum_{1 \le i \le n} X_i$ and $m := E(Y)$. Then for all a, $0 < a < nz - m$, we have*

$$\Pr\big(Y \ge m + a\big) \le \left[\left(\frac{m}{m+a}\right)^{m+a} \left(\frac{nz-m}{nz-m-a}\right)^{nz-m-a}\right]^{1/z} \le \left(\frac{m}{m+a}\right)^{(m+a)/z} \cdot e^{a/z}.$$

(The second inequality holds since $(1 + \frac{x}{\alpha})^{\alpha} \le e^x$ for $\alpha > 0$.) □

We need another technical fact about sums of independent random variables with distribution bounded by a geometrical distribution.

Lemma 1 *Let $m \ge 1$, $w_1, \ldots, w_m > 0$ be arbitrary. Abbreviate $\sum_{i=1}^{m} w_i$ by W and $\max\{w_i \mid 1 \le i \le m\}$ by M. Assume that $X_1, \ldots, X_m$ are independent random variables so that for $1 \le i \le m$ holds*

$$\Pr(X_i > L \cdot w_i) \le 2^{-L}, \quad \text{for } L = 1, 2, 3, \ldots .$$

Then

$$\Pr\Big(\sum_{i=1}^{m} X_i \ge 3 \cdot l \cdot W\Big) \le e^{-(l-1)W/M}, \quad \text{for } l = 1, 2, 3, \ldots .$$

(Note that $E(\sum_{i=1}^{m} X_i) \le 2W$.)

Proof: W. l. o. g. we may assume that X_i is as large as possible while still satisfying the assumption $\Pr(X_i > L \cdot w_i) \le 2^{-L}$, that means, $\Pr(X_i \in \{Lw_i \mid L = 1, 2, 3, \ldots\}) = 1$ and

$$\Pr(X_i = L \cdot w_i) = 2^{-L} \quad \text{for } 1 \le i \le m,\ L = 1, 2, 3, \ldots . \tag{1}$$

Define $Y := \sum_{i=1}^{m} X_i$. It is a well-known trick to use the following inequality for proving estimates for sums of independent random variables. (See, e. g., the proof of the preceding theorem in [12].) For arbitrary $h > 0$ the following holds:

$$\Pr(Y \ge 3l \cdot W) \le E(e^{h(Y - 3lW)}). \tag{2}$$

Thus, in order to prove the lemma, we need only estimate $E(e^{hY})$. By the independence of $X_1, \ldots, X_m$ we have

$$E(e^{hY}) = \prod_{i=1}^{m} E(e^{hX_i}). \tag{3}$$

Clearly, by (1),

$$E(e^{hX_i}) = \sum_{L=1}^{\infty} \Pr(X_i = Lw_i) \cdot e^{hLw_i} = \sum_{L=1}^{\infty} \left(\frac{e^{hw_i}}{2}\right)^{L}.$$

Hence, if h is so small that $e^{hw_i} < 2$, for which it is certainly sufficient if

$$h < M^{-1} \cdot \ln 2, \tag{4}$$

we obtain

$$E(e^{hX_i}) = \frac{e^{hw_i}}{2 - e^{hw_i}}. \tag{5}$$

It is an easy observation that $\frac{t}{2-t} \leq t^3$ for $1 \leq t \leq \frac{1}{2}(1+\sqrt{5})$. Hence, for h so small that $e^{hw_i} < \frac{1}{2}(1+\sqrt{5})$, which is certainly satisfied if

$$h < M^{-1} \cdot \ln(\tfrac{1}{2}(1+\sqrt{5})), \tag{6}$$

we get from (5) that

$$E(e^{hX_i}) < e^{3hw_i}. \tag{7}$$

By substituting (7) into (3) and by the definition of W, we conclude

$$E(e^{hY}) < e^{3hW}; \tag{8}$$

hence, in combination with (2) we get

$$\Pr(Y \geq 3l \cdot W) \leq e^{-(l-1)\cdot 3h\cdot W}. \tag{9}$$

We let $h := (3M)^{-1}$. Then clearly (4) and (6) are satisfied, and (9) reads $\Pr(Y \geq 3lW) \leq e^{-(l-1)W/M}$, which was to be shown. ∎

Remark 1 In [18], mean, variance, and higher moments for sums of independent geometrically distributed random variables are calculated. Starting from those formulas, estimates similar to that in Lemma 1 can be derived. By different methods, in [20] an analogous result is proved for sums of independent, unbounded, but not exactly geometrically distributed, random variables.

The following theorem (a proof of which can be found in [16]) is extremely helpful when we need to estimate the probability that a random variable T deviates far from its mean in case T is a function of many independent variables each of which has only small influence on T. The usefulness of this theorem for analyzing hash functions and parallel dictionary algorithms has been noted before, see, e. g., [2] and [9].

Theorem 4 ([16]) *Let $X_1, \ldots, X_n$ be independent random variables with ranges $\Omega_1, \ldots, \Omega_n$. Further, let $g: \prod_{1\leq i\leq n} \Omega_i \to \mathbb{R}$ be an arbitrary function so that g changes by at most a constant $\vartheta > 0$ in response to a change in a single component, i. e.,*

$$\left| g(\omega_1 \ldots, \omega_{i_0}, \ldots, \omega_n) - g(\omega_1 \ldots, \omega'_{i_0}, \ldots, \omega_n) \right| \leq \vartheta,$$

for $\omega_i \in \Omega_i$, $1 \leq i \leq n$, and $\omega'_{i_0} \in \Omega_{i_0}$ arbitrary. Let $T := g(X_1, \ldots, X_n)$. Then for all $t \geq 0$ we have

$$\Pr\big(T \geq E(T) + t\big) \leq e^{-t^2/(2\vartheta^2 n)}.$$

□

The proof of this theorem is based on a "martingale tail estimate" ("Azuma's inequality"); this is why also arguments based on Theorem 4 will be referred to as "martingale estimates".

4 A new class of hash functions

In this section, we define a new class of hash functions and establish some of the properties of this class. The technical definition is as follows.

Definition 2 *Assume* $r, s \geq 1$ *and* $d \geq 2$.

(a) *For arbitrary* $f: U \to \{0, 1, \ldots, rs-1\}$ *define* $f_1(x) := f(x) \operatorname{div} s$ *and* $f_2(x) := f(x) \bmod s$, *for* $x \in U$. *(*$a \operatorname{div} b$ *means* $\lfloor a/b \rfloor$.*)*

(b) *For* $f: U \to \{0, 1, \ldots, rs-1\}$ *and* $a_0, \ldots, a_{r-1} \in \{0, 1, \ldots, s-1\}$ *let the function* $h =$ $h(f, a_0, \ldots, a_{r-1})$ *be defined by*

$$h(x) := \Big(f_2(x) + a_{f_1(x)}\Big) \bmod s.$$

(c) *The class* $\mathcal{R}(r, s, d)$ *consists of all functions* $h(f, a_0, \ldots, a_{r-1})$ *with* $f \in \mathcal{H}_{rs}^d$ *and* $a_0, \ldots, a_{r-1} \in \{0, \ldots, s-1\}$ *arbitrary.*

Remark 2 (a) Note that f and the pair (f_1, f_2) are practically the same function. It will be convenient to regard the range of (f_1, f_2) as an $r \times s$-array.

(b) If $f \in \mathcal{H}_{rs}^d$ is given as $f(x) = F(x) \bmod rs$, where

$$F(x) = \Big(\sum_{0 \leq l < d} \alpha_l x^l\Big) \bmod p,$$

an efficient way for evaluating $h = h(f, a_0, \ldots, a_{r-1})$ is given by the formula

$$h(x) = \Big(F(x) + a_{(F(x) \operatorname{div} s) \bmod r}\Big) \bmod s.$$

(c) Obviously, a function $h \in \mathcal{R}(r, s, d)$ can be stored in $O(d + r)$ cells, can be generated in $O(d + r)$ steps, and can be evaluated in time $O(d)$.

Before embarking on the rigorous analysis of the class $\mathcal{R}(r, s, d)$, let us (informally) explain the effect of such a function on the keys $x \in U$. First, the function f maps the keys into an $r \times s$-array (x is mapped into the cell in row $f_1(x)$ and column $f_2(x)$ of the array). In a second step, row i is shifted cyclically by an offset a_i, i. e., key x is moved from column $f_2(x)$ to column $(f_2(x) + a_i) \bmod s$. These shifts are independent for the different rows. The hash value $h(x)$ is given by the number of the column to which x is moved by these two steps.

For the following, we assume that a set $S \subseteq U$ is given, with $|S| = n$, and that $rs > n$. The nice behaviour of the class $\mathcal{R}(r, s, d)$ rests essentially on the fact that if $n/(rs)$ is sufficiently small then with high probability f will be $(d-1)$-perfect on S; this property implies that each row i of the array can contribute at most $d-1$ keys to the quantity $b_j = |\{x \in S \mid h(x) = j\}|$.

Definition 3 *For $0 \le i < r$, let $B_i^f := \{x \in S \mid f_1(x) = i\}$ (the elements of S mapped to the ith row of the array).*

Lemma 2 *We have $\Pr(f$ is $(d-1)$-perfect for $S) = 1 - O\left(n^d/(rs)^{d-1}\right)$, if f is chosen at random from $\mathcal{H}_{rs}^d$.*

Proof: This is immediate from Fact 1. ∎

Definition 4 *Let $f: U \to \{0, 1, \ldots, rs-1\}$ be a function that is $(d-1)$-perfect on S. We fix f and consider only the random experiment of choosing $a_0, \ldots, a_{r-1}$. Formally, let*

$$\mathcal{R}_f(r,s) := \left\{h(f, a_0, \ldots, a_{r-1}) \;\middle|\; a_0, \ldots, a_{r-1} \in \{0, \ldots, s-1\}\right\}.$$

We write $\Pr_f(\mathcal{A})$ for $\Pr(\mathcal{A} \mid \mathcal{R}_f(r,s))$ and $E_f(X)$ for $E(X \mid \mathcal{R}_f(r,s))$, for arbitrary events $\mathcal{A}$ and random variables X.

The following theorem lists the most important properties of functions from $\mathcal{R}(r,s,d)$, for the case $s \ge n$. Note that these properties are close to what one would get if h mapped the keys from S uniformly and independently into $\{0, \ldots, s-1\}$, i. e., in the case of uniform hashing (cf. [10]). We just remark that corresponding statements are also true for s smaller than n.

Theorem 5 *For $S \subseteq U$ fixed, $|S| \le n = s$, and h randomly chosen from $\mathcal{R}_f(r,s)$ the following holds.*

(a) $\Pr_f(b_j \ge u) \le \left(\dfrac{e^{u-1}}{u^u}\right)^{1/(d-1)}$, *for* $0 \le j < n$.

(b) $\Pr_f\left(b_x^{\text{coll}} \ge (d-1) + u\right) \le \left(\dfrac{e^{u-1}}{u^u}\right)^{1/(d-1)}$, *for all* $x \in U$.

(c) *If* $u \ge 4(d-1)\ln n / \ln\ln n$ *then* $\Pr_f\left(\max\{b_j \mid 0 \le j < n\} \ge u\right) \le u^{-u/(2(d-1))}$.

(d) $E_f\left(\max\{b_j \mid 0 \le j < n\}\right) = O(\log n / \log\log n)$.

Proof: Assume that $|S| = n = s$. (Otherwise add some dummy elements to S.) Fix some f that is $(d-1)$-perfect on S.

(a) Fix j, $0 \le j < n$. We define random variables X_i that measure the contribution of B_i^f to the bucket B_j, as follows:

$$X_i := \left|\left\{x \in S \;\middle|\; h(x) = j \text{ and } f_1(x) = i\right\}\right|, \quad \text{for } 0 \le i < r.$$

Obviously, we have

$$b_j = \sum_{0 \le i < r} X_i.$$

We observe that, by definition,

$$X_i = \left|\left\{x \in S \;\middle|\; f_1(x) = i \text{ and } f_2(x) = (j - a_i) \bmod n\right\}\right|, \quad \text{for } 0 \le i < r.$$

Now a_i is randomly chosen, hence $(j-a_i) \bmod n$ is a random element of $\{0,\ldots,n-1\}$. Thus, X_i is the number of elements of S in a randomly chosen cell of the ith row of the $r \times n$-array that forms the range of f. This implies the following:

(i) $0 \le X_i \le d-1$, for $0 \le i < r$;

(ii) the X_i, $0 \le i < r$, are independent;

(iii) $E_f(X_i) = \frac{1}{n} \cdot |\{x \in S \mid f_1(x) = i\}| = \frac{1}{n} \cdot |B_i^f|$, for $0 \le i < r$.

From (iii) we get the (obvious) expected value of b_j:

$$E_f(b_j) = E_f\Big(\sum_{0 \le i < r} X_i\Big) = \tfrac{1}{n} \cdot \sum_{0 \le i < r} |B_i^f| = \tfrac{1}{n} \cdot |S| = 1.$$

Applying Hoeffding's Theorem (Theorem 3) immediately yields

$$\Pr_f(b_j \ge u) \le \left(\frac{1}{u}\right)^{u/(d-1)} \cdot e^{(u-1)/(d-1)},$$

which is (a).

(b) This is proved in exactly the same way as (a), excepting that in addition to f also $a_{f_1(x)}$ is considered fixed. At most $d-1$ elements of $B^f_{f_1(x)}$ are mapped to the same cell as x by h; the contribution of the other B_i^f, $i \ne f_1(x)$, to b_x^{coll} is analyzed just as in (a).

(c) This follows easily from (a), as follows. Clearly,

$$\begin{aligned}&\Pr_f\Big(\max\{b_j \mid 0 \le j < n\} \ge u\Big)\\ &\quad \le \min\Big\{1, \sum_{0 \le j < n} \Pr_f(b_j \ge u)\Big\} \le \min\left\{1, n \cdot \left(\frac{e^{u-1}}{u^u}\right)^{1/(d-1)}\right\}.\end{aligned}$$

We must show that no u with $u \ge 4(d-1)\ln n / \ln\ln n$ can satisfy the inequality $n \cdot (e^{u-1}/u^u)^{1/(d-1)} > u^{-u/(2(d-1))}$. But this is clear: the second inequality implies $2(d-1)\ln n + 2(u-1) > u \ln u$, while the first one implies $u \ln u \ge 2(u-1) + \frac{4}{5} \cdot (4(d-1)\ln n / \ln\ln n) \cdot (\ln\ln n - \ln\ln\ln n) > 2(u-1) + \frac{16}{5} \cdot (d-1)\ln n \cdot (1 - \frac{\ln\ln\ln n}{\ln\ln n}) > 2(u-1) + 2(d-1)\ln n$. (We use that $\frac{\ln\ln\ln n}{\ln\ln n} < 1/e \approx 0.368$.)

(d) This is immediate from (c), by using that $E(Z) = \sum_{u \ge 1} \Pr(Z \ge u)$ for integral, nonnegative random variables Z. ∎

The basic properties given in the previous theorem are already very useful if the class $\mathcal{R}(r,s,d)$ is to be used in a simple hashing scheme, e. g., in chained hashing. For our purposes, we have to study the class a little more closely.

Below, we will only consider the case $|S| = r = s = n$.

Remark 3 Note that if it is desirable that the space needed to store h is smaller than n, also $r = n^\delta$ for some $\delta < 1$ is a suitable choice as long as $n/(rs)$ remains sufficiently small. Also, for $n/\log n \leq s \leq n$ the class $\mathcal{R}(r,s,d)$ exhibits a behaviour comparable to that of random functions.

It is our aim to obtain a sharper version of the estimate

$$E\Big(\sum_{0\leq j<n}(b_j)^2\Big) = O(n),$$

which $\mathcal{R}(n,n,d)$ shares with all other universal classes. We are interested in sums of the form $\sum_{j\in J}(b_j)^2$ for only a subset $J \subseteq \{0,\ldots,n-1\}$ (where J may depend on h), and we need to establish that this sum is close to its expectation with high probability. The essential tool for the proof will be the martingale tail estimate (Theorem 4). — We start with some technical preparations.

Definition 5 *Assume $r = s = n$. We abbreviate $\mathcal{R}(n,n,d)$ by $\mathcal{R}(d)$; if the value of d is inessential, we also write simply $\mathcal{R}$.*

Definition 6 *Let $S \subseteq U$ with $|S| = n$.*

(a) *We say that a function $f: U \to \{0,\ldots,n^2-1\}$* distributes *$S$* well *if it is $(d-1)$-perfect on S and*

$$|B_i^f| = |\{x \in S \mid f_1(x) = i\}| \leq n^{1/4}\,, \quad \text{for } 0 \leq i < n.$$

(b) *The subclass $\mathcal{R}_S(d)$ (or $\mathcal{R}_S$) of $\mathcal{R}(d)$ consists of all functions $h = h(f,a_0,\ldots,a_{n-1}) \in \mathcal{R}(d)$ for which f distributes S well.*

If we regard the range of f as an $n \times n$-array, as discussed above, then the subclass $\mathcal{R}_S(d)$ consists of those functions $h = h(f,a_0,\ldots,a_{n-1})$ for which f maps at most $(d-1)$ keys from S into each cell of the array and at most $n^{1/4}$ into each row. We show that $\mathcal{R}_S(d)$ contains almost all functions from $\mathcal{R}(d)$.

Lemma 3 *For arbitrary $c \geq 1$, if the constant d is chosen sufficiently large, the following holds: For arbitrary $S \subseteq U$ with $|S| = n$ we have*

$$\frac{|\mathcal{R}_S(d)|}{|\mathcal{R}(d)|} = 1 - O(n^{-c}).$$

Proof: For $f: U \to \{0,\ldots,n^2-1\}$, consider the function $\hat{f}$ defined by

$$\hat{f}(x) := \left(f_1(x), f_2(x) \bmod \left\lfloor \frac{n^{1/4}}{d-1} \right\rfloor\right), \text{ for } x \in U.$$

This function has range $\{0,\ldots,n-1\} \times \{0,\ldots,\lfloor n^{1/4}/(d-1)\rfloor - 1\}$, which has $\hat{s} \approx n^{5/4}/(d-1)$ elements. It is not hard to check that the class $\{\hat{f} \mid f \in \mathcal{H}_{n^2}^d\}$ has

essentially the same properties as the class $\mathcal{H}_{\hat{s}}^d$ (cf. Fact 1); in particular, for f chosen at random from $\mathcal{H}_{n^2}^d$ we have

$$\begin{aligned}\Pr(\hat{f} \text{ is } (d-1)\text{-perfect on } S) &= 1 - O\left(n \cdot \left(\frac{n \cdot (d-1)}{n^{5/4}}\right)^{d-1}\right) \\ &= 1 - O\left(n^{1-(d-1)/4}\right).\end{aligned}$$

The last expression is $1 - O(n^{-c})$ for d sufficiently large. Finally, observe that clearly if $\hat{f}$ is $(d-1)$-perfect on S then f distributes S well. This finishes the proof of the lemma. ∎

Now we turn to the study of certain sums $\sum_{j \in J}(b_j)^2$.

Definition 7 *For $S' \subseteq S$ arbitrary and $h \in \mathcal{R}$ random we define the random set $J(S')$ as the set of all indices j of buckets B_j that are hit by elements of S', i. e.,*

$$J(S') := \{j \mid 0 \leq j < n \text{ and } \exists x \in S' : h(x) = j\}.$$

Further, we let

$$M_2(S') := \sum_{j \in J(S')} (b_j)^2.$$

Theorem 6 *Let $c \geq 1$ be fixed. Then there is a constant $C > 0$ such that for all $S \subseteq U$ with $|S| = n = s$ and all $S' \subseteq S$ with $|S'| \geq n^{7/8}$ the following holds:*

(a) $E\big(M_2(S')\big) \leq (C-1) \cdot |S'|$

(b) $\Pr\big(M_2(S') \geq C \cdot |S'|\big) = O(n^{-c})$.

Proof: By Lemma 3, it is sufficient to show the following, for some fixed f that distributes S well (for the notation E_f and $\Pr_f$, see the proof of Theorem 5):

(a') $E_f(M_2(S')) \leq (C-1) \cdot |S'|$;

(b') $\Pr_f\big(M_2(S') \geq E_f(M_2(S')) + |S'|\big) = O(n^{-c})$.

Thus, fix such an f. We first prove (a'). For this, recall from Theorem 5 that for arbitrary $x \in S$ the random variable $b_x^{\text{coll}} = |B_x^{\text{coll}}| = |\{y \in S \mid h(x) = h(y)\}|$ satisfies $\Pr_f(b_x^{\text{coll}} \geq u + (d-1)) \leq (e/u)^{u/(d-1)}$, for $u \geq 1$. This implies that for all $v \geq 1$ and all $x \in S$ we have

$$\Pr_f\big((b_x^{\text{coll}})^2 \geq v\big) = \Pr_f\big(b_x^{\text{coll}} \geq \lfloor\sqrt{v}\rfloor\big) \leq \left(\frac{e}{\lfloor\sqrt{v}\rfloor - (d-1)}\right)^{(\lfloor\sqrt{v}\rfloor - (d-1))/(d-1)}.$$

Thus,

$$E_f\left((b_x^{\text{coll}})^2\right) = \sum_{v\geq 1}\sum_{w\geq 1}(2w+1)\cdot\left(\frac{e}{w-(d-1)}\right)^{(w-(d-1))/(d-1)} = O(1).$$

Choose the constant C so that the last sum is bounded by $C-1$. By the obvious inequality

$$M_2(S') \leq \sum_{x\in S'}(b_x^{\text{coll}})^2,$$

we conclude (by linearity of expectation) that

$$E_f(M_2(S')) \leq \sum_{x\in S'} E_f\left((b_x^{\text{coll}})^2\right) \leq (C-1)\cdot|S'|.$$

Thus, (a') is proved. Now we turn to the proof of (b'). For technical reasons (which will become clear below), we need a truncated version of the random variable $M_2(S')$, which is defined as follows. Let

$$\hat{b}_j := \min\{b_j, \log n\}, \text{ for } 0\leq j<n.$$

Note that by Theorem 5(c) we have

$$\Pr_f(b_j \leq \log n \text{ for } 0\leq j<n) = O(n^{-c}).$$

Now, let

$$\hat{M}_2(S') := \sum_{j\in J(S')}(\hat{b}_j)^2.$$

By the above, we have

$$\Pr_f\left(M_2(S') \neq \hat{M}_2(S')\right) = O(n^{-c});$$

further it is clear, by definition, that

$$E_f(M_2(S')) \geq E_f(\hat{M}_2(S')).$$

Thus, in order to prove (b'), it suffices to establish the following:

(b") $\Pr_f\left(\hat{M}_2(S') \geq E_f(\hat{M}_2(S')) + |S'|\right) = O(n^{-c})$.

For fixed f, the hash function $h = h(f, a_0, \ldots, a_{n-1})$, and hence also $\hat{M}_2(S')$ is a function of the independent random variables $a_0, \ldots, a_{n-1}$. In order to apply Theorem 4 we must estimate the effect a change in a single one of the offsets a_i has on $\hat{M}_2(S')$. If a_i is changed, only the hash values $h(x)$ of the up to $n^{1/4}$ many keys x in B_i^f change. This can change the set $J(S')$ (if $h(x)$ changes for some $x\in S'$) and it can change $\hat{b}_j$ for some j that is in $J(S')$ before and after changing a_i. However, it is clear that the total number of $\hat{b}_j$'s that increase (or for which j enters $J(S')$) is bounded by $|B_i^f| \leq n^{1/4}$;

similarly for the total number of $\hat{b}_j$'s that decrease (or for which j leaves $J(S')$). Since, by definition, all $\hat{b}_j$ are in $[0, \log n]$, the total change of the sum $\hat{M}_2(S') = \sum_{j \in J(S')} (\hat{b}_j)^2$ in response to changing one a_i is bounded by $\vartheta := n^{1/4} \cdot (\log n)^2$.
Applying Theorem 4 now yields (using the notation $\exp(y)$ for e^y):

$$\begin{aligned}
&\Pr_f\Big(\hat{M}_2(S') \geq E_f(\hat{M}_2(S') + |S'|)\Big) \\
&\quad \leq \exp\Big(-|S'|^2/(2n \cdot (n^{1/4} \cdot (\log n)^2)^2)\Big) \\
&\quad = \exp\Big(-\Omega(n^{1/4}/(\log n)^4)\Big) \\
&\quad = O(n^{-c}).
\end{aligned}$$

(The second to the last estimate uses the assumption $|S'| \geq n^{7/8}$.) Thus, (b") (and hence also (b)) is proved. ∎

Remark 4 In [7], a slightly different type of hash functions with the same performance is used. The main advantage of the functions introduced in this section is the relatively simple proof of Lemma 3. The analogous argument in [7] is much more complicated.

5 Dynamic hashing in real time

In this section, we construct the dictionary with the features claimed in Theorem 1.

5.1 An auxiliary construction

As a first but crucial step, we present a probabilistic algorithm for a variant of a dictionary with the following properties: it is able to store n keys; lookups are performed right away; "batches" of n^ε update operations are performed in $O(n^\varepsilon)$ time; the algorithm works correctly with high probability. The algorithm differs from a standard dictionary in that the updates contained in a "batch" are performed in an arbitrary order, not in the order prescribed by the input. This auxiliary data structure constitutes the core of our real-time dictionary.

Definition 8 *Let $0 < \varepsilon \leq 1$, and let n be given. A* type-A dictionary *with respect to ε is a data structure with the following properties.*

(a) *The input is provided in two sequences of instructions:*

$(x_1, Op_1), \ldots, (x_n, Op_n)$ *are the insertion and deletion instructions;*

$(x'_1, Lookup), \ldots, (x'_n, Lookup)$ *are the lookup instructions.*

(b) *The operations are performed in* phases $l = 1, 2, \ldots, n^{1-\varepsilon}$*; in the lth phase the update instructions $\{(x_i, Op_i) \mid (l-1)n^\varepsilon < i \leq ln^\varepsilon\}$ and the lookup instructions $\{(x'_i, Lookup) \mid (l-1)n^\varepsilon < i \leq ln^\varepsilon\}$ are performed. All keys occurring in update instructions for phase l are assumed to be distinct.*

(c) *Each phase takes at most time Kn^ε in the worst case, for a constant K.*

(d) *The lookup instructions of phase l are performed at a constant rate, in the order given by the input sequence. For keys x_i' that do not occur in update instructions for phase l the answer is correct, that means, it reflects the state of the dictionary at the end of phase $l-1$.*

(e) *The algorithm uses space $O(n)$.*

Theorem 7 *For arbitrary $\frac{7}{8} < \varepsilon \leq 1$ and $c \geq 1$ there is a probabilistic algorithm for implementing a type-A dictionary with respect to ε. The probability that the algorithm fails to stay within the time bounds of Definition 8(c) is $O(n^{-c})$. The other properties are always satisfied.*

Proof: We describe the algorithm, which is a generalization of the perfect hashing schemes from [8] and [4] (cf. Section 1). Let $S = \{x_1, \ldots, x_n\}$ be the set of keys used in the n update instructions.

Algorithm 1 (Type-A dictionary)

Choose $h \in \mathcal{R} = \mathcal{R}(n, n, d)$ at random (cf. Definitions 2 and 5), for d large enough for the probability estimates below to hold. The function h splits S into buckets B_j, $0 \leq j < n$. The keys $x \in B_j$ are stored in a subtable ST_j. Initially, all ST_j are empty, as well as the header table HT, which is an array with n positions. Then the algorithm proceeds in phases $l = 1, 2, \ldots, n^{1-\varepsilon}$.

Phase l:

Part 1: Perform update instructions

Evaluate $h(x)$ for each $x \in S(l) := \{x_{(l-1)n^\varepsilon}, \ldots, x_{ln^\varepsilon}\}$. Let $J_l := \{j \mid S(l) \cap B_j \neq \emptyset\}$. For all $j \in J_l$, collect $x \in S(l) \cap B_j$ in a sequential list T_j'', construct a list T_j' of those keys from B_j already stored in ST_j, and construct $T_j :=$ concatenation of T_j' and T_j''.

Then, for each $j \in J_l$ in turn do the following: Randomly choose a function h_j from $\mathcal{H}^2_{2|T_j|^2}$ and test whether h_j is one-to-one on T_j. Repeat this until such a function is found. Set up a new subtable ST_j of size $2|T_j|^2$, and enter the keys $x \in T_j$ into this new table according to the hash function h_j, together with the information fields that belong to these keys according to the entry in the old subtable ST_j or to the input sequence. Deletions are effected by adding a tag "deleted" to the entry. When this is done, change the pointer in the header table $HT[j]$ to point to the new subtable ST_j.

Part 2: Perform lookup instructions

The n^ε lookup instructions are performed in an interleaved manner concurrently with the update procedure of Part 1, so that lookup instructions are accepted at constant time intervals. (Clearly, each lookup instruction takes constant time.)

Analysis: Let $c \geq 1$ be given. For d large enough, we have

$$\Pr(h \in \mathcal{R}_S) \geq 1 - O(n^{-c-1}),$$

by Lemma 3. Further, by Theorem 6 there is a constant C so that

$$\Pr\Big(\sum_{j \in J_l} (b_j)^2 \geq C \cdot n^\varepsilon \;\Big|\; h \in \mathcal{R}_S \Big) = O(n^{-c-1}),$$

where $b_j = |B_j|$, $0 \leq j < n$, and the set

$$J_l = J(S(l)) = \Big\{ j \;\Big|\; 0 \leq j < n \text{ and } \exists x \in S(l) \colon h(x) = j \Big\}$$

is as in Definition 7. Finally, we have noted already in the previous section that Theorem 5(c) implies that

$$\Pr\Big(\max\{b_j \mid 0 \leq j < n\} \geq \log n \;\Big|\; h \in \mathcal{R}_S \Big) = O(n^{-c}).$$

Thus, for $h \in \mathcal{R}$ randomly chosen, we have that with probability at least $1 - O(n^{-c})$ both the following two properties are satisfied:

$$\forall l \in \{1, \ldots, n^{1-\varepsilon}\} \colon \sum_{j \in J_l} (b_j)^2 < C \cdot n^\varepsilon, \tag{1}$$

and

$$\forall j \in \{0, \ldots, n-1\} \colon b_j \leq \log n. \tag{2}$$

As we allow the algorithm to fail to work properly with a probability bounded by $O(n^{-c})$, we may assume for the following that properties (1) and (2) are satisfied for the level-1 hash function h we have chosen. Under this assumption, we analyze the time needed for a single fixed phase l. The following obvious fact is crucial:

$$|T_j| \leq b_j, \text{ for all } j \in J_l. \tag{3}$$

(A subtle point to note here is that for this to be true, if T_j is regarded as a list, we need the condition that the keys occurring in update instructions in phase l are all distinct.) Evaluating $h(x)$ for $x \in S(l)$ takes total time $O(|S(l)|) = O(n^\varepsilon)$. Constructing the lists T_j', T_j'', and T_j, for $j \in J_l$, can easily be done in time $O(\sum_{j \in J_l} |T_j|^2)$, which is $O(n^\varepsilon)$, by inequalities (1) and (3). Transferring the data from the old ST_j to the new one and changing pointers can also clearly be done in time $O(\sum_{j \in J_l} |T_j|^2)$. The only point left to consider is the construction of the new level-2 hash functions h_j, $j \in J_l$, by the randomized procedure described above. Clearly, choosing one function h_j from $\mathcal{H}^2_{2|T_j|^2}$ and testing it for injectivity on T_j takes time $O(|T_j|^2)$. The probability of one such trial being successful is at least $\frac{1}{2}$, by Fact 1; hence the probability that more than L trials are needed until success occurs is at most 2^{-L}. If X_j, $j \in J_l$, denotes the time needed for constructing h_j, then X_j is bounded by a geometrically distributed random variable with mean $O(|T_j|^2)$ (which is $O((\log n)^2)$, by inequality (2)). Since

$$\sum_{j \in J_l} E(X_j) = O\Big(\sum_{j \in J_l} |T_j|^2 \Big) = O(n^\varepsilon)$$

and $\max\{E(X_j) \mid j \in J_l\} = O((\log n)^2)$, a straightforward application of Lemma 1 yields that, for some constant C', we have

$$\Pr\Big(\sum_{j\in J_l} X_j \geq C' \cdot n^\varepsilon\Big) = e^{-\Omega(n^\varepsilon/(\log n)^2)} = O(n^{-c-1}).$$

Thus, the update part of phase l is finished within $O(n^\varepsilon)$ steps with probability $1 - O(n^{-c-1})$, hence *all* phases are finished within this time bound with probability $1 - O(n^{-c})$.

As for the space requirements, it is obvious that the number of memory cells needed in phase l (for storing the new subtables) is bounded by $O(\sum_{j\in J_l} |T_j|^2) = O(\sum_{j\in J_l}(b_j)^2)$, which is $O(n^\varepsilon)$ as soon as (1) holds. This shows that in this case the total space needed is $O(n)$.

Performing the lookup operations in an interleaved manner with the update procedure causes no problems at all. It is obvious from the algorithm that the consistency condition stated in Definition 8(d) is satisfied.

This finishes the proof of Theorem 7. ∎

Remark 5 Starting with a clear memory, setting up an empty type-A dictionary takes constant time, if we assume that for choosing the level-1 function h from $\mathcal{R}$ only f is chosen at the beginning and the offset a_i is chosen only when the first key x with $f_1(x) = i$ appears, for each $i \in \{0, \ldots, n-1\}$.

5.2 The real-time dictionary

We describe two versions of the dynamic dictionary: one for the case that n, the number of instructions to be performed, is known in advance, and a fully dynamic one, where no such assumption is made.

First, assume that the number n of instructions is given in advance. The dictionary uses space $O(n)$; it is initially empty.

Definition 9 *A* type-B dictionary *is a data structure with the following properties. Let n arbitrary instructions to be performed. Then*

(a) *Space $O(n)$ is used;*

(b) *$O(1)$ steps are used for each instruction, in the worst case.*

Theorem 8 *Let $c \geq 1$ be arbitrary. There is a probabilistic algorithm for implementing a type-B dictionary. The probability that the algorithm fails to fulfill property (b) from Definition 9 is $O(n^{-c})$; property (a) is always satisfied.*

Proof: We first give an overall description of the parts of the data structure. Let $\frac{7}{8} < \varepsilon < 1$. The data structure consists of three layers:

(α) the *(current) temporary dictionary* D_{new} of capacity n^ε;

(β) the *closed temporary dictionary* D_{old} of capacity n^ε;

(γ) the *background dictionary* D_{back} of capacity n.

The algorithm proceeds in phases $l = 1, 2, \ldots, n^{1-\varepsilon}$. In each phase, n^ε arbitrary instructions are processed. The effect of insert- and delete-instructions during phase l is recorded in D_{new}. The closed dictionary D_{old} is the temporary dictionary of the previous phase $l-1$, in the state it had at the end of this phase. At the beginning of phase l, D_{back} is the complete dictionary in the state of the beginning of phase $l-1$. During phase l, concurrently with processing the n^ε new instructions, the information stored in D_{old} is used to update D_{back}.

Let $S(l)$ denote the keys occurring in phase l (i.e., those keys in instructions $(l-1)n^\varepsilon + 1, \ldots, l \cdot n^\varepsilon$). We now describe the algorithm in detail.

Algorithm 2 (Type-B dictionary)
Preparation: (During this period no instructions are processed.) Set up D_{back} as an empty type-A dictionary of capacity n. Set up empty dictionaries D_{new} and D_{old}, space $O(n)$, choose randomly one $h \in \mathcal{H}_n^d$ that serves as hash function for both D_{new} and D_{old} (cf. Fact 2).
Phase l:

The phase consists of n^ε rounds; in each round one instruction is processed. The actions in a single round are as follows.

1. Read next instruction (x, Op).
   (a) If $Op =$ *Insert*, then enter key x in D_{new}. Overwrite the information associated with x, if x is already present in D_{new}.
   (b) If $Op =$ *Delete*, then enter key x in D_{new} with tag "deleted", or add this tag, if x is already present in D_{new}.
   (c) If $Op =$ *Lookup*, then look for x in $D_{\text{new}}, D_{\text{old}}$, and D_{back}, in this order; return the information associated with x in the first dictionary in which x is present. If x is found with tag "deleted" or not found at all, return "not found".
2. Perform a constant number of steps of the following procedure.
   Transfer D_{old} to D_{back}: Collect the keys in D_{old} in a sequential list. Insert the keys in the list into D_{back} (including tags "deleted", if applicable), as described in Algorithm 1. When finished, make D_{old} inaccessible for lookups, then erase all entries in D_{old}.

This finishes the description of a round. After all n^ε rounds that belong to phase l are finished, the roles of D_{new} and D_{old} are switched. (At the beginning of a phase, D_{new} is empty and D_{old} contains the result of the phase that just ended.)

Time analysis:
"Preparation": Setting up D_{new} and D_{old} clearly takes constant time, if the memory is clear initially. Setting up D_{back} also takes constant time, by Remark 5.
"Process Instructions": Executing an instruction (Part 1 of a round) takes constant time, if the hash function h of D_{new} is $(d-1)$-perfect for $S(l)$. By Fact 2, this is the case for *all* phases l with probability

$$1 - O\left(n^{1-\varepsilon} \cdot n^{\varepsilon} \cdot n^{-(\varepsilon-1)(d-1)}\right) = 1 - O(n^{-c}),$$

for d large enough. Clearing D_{old} takes time $O(n^{\varepsilon})$ by Fact 2. The transfer of elements from D_{old} to D_{back} takes $O(n^{\varepsilon})$ steps *in each phase* l with probability $1 - O(n^{-c})$ (Theorem 7). (Note that for each key stored in D_{old} there is only one entry; thus all keys to be inserted into D_{back} are distinct.) Thus, we must only set the constant large enough that directs how many steps of the "transfer" procedure are executed in each round to make sure that n^{ε} rounds are sufficient to finish the procedure.
Correctness:

We need to check that the answer for a task $(x, \textit{Lookup})$ given by the algorithm is correct, i. e., reflects the last change to the information associated with key x. There are three cases.

Case 1: x occurred last in a delete- or insert-instruction in phase l. Then D_{new} has the correct information on x, and this is found first.

Case 2: x occurred last in a delete- or insert-instruction in phase $l-1$. Then x is not found in D_{new}, but it is found together with the correct information in D_{old}, if the lookup takes place before D_{old} is made inaccessible. After that, the correct information concerning x is found in D_{back}.

Case 3: x occurred neither in phase l non in phase $l-1$. Then D_{back} contains the correct information on x, and it is found there.

This finishes the proof of Theorem 8. ∎

Remark 6 At the end of Algorithm 2, i. e., directly after the last instruction is processed, the data structure looks as follows. D_{back} reflects the state of the dictionary at the beginning of the last phase, D_{old} reflects the updates from the last phase, D_{new} is empty. It is easy to see that from this structure a list of the keys that eventually belong to the dictionary may be computed in $O(n)$ steps (worst case).

We still have to handle the case that the number of instructions to be performed by a dictionary may not be known in advance and/or exceed the number of keys stored in the dictionary at any one time. For these cases, a type-B dictionary is not flexible enough. Thus, we need a means for adjusting the size of the dictionary in use. This can easily be done if during these periods of readjustment no instructions are accepted, but the resulting behaviour of the dictionary would not have the desired real-time feature. We can do better by employing a similar strategy as in Algorithm 2: use a temporary dictionary for recording the changes made to the dictionary; simultaneously transfer the information contained in the previous temporary dictionary to

a background dictionary. (In this approach we combine ideas from [4] and [3].) The construction finally yields the fully dynamic real-time dictionary claimed to exist in Theorem 1.

Algorithm 3 The algorithm again works in phases. The first phase starts when the first instruction is read. It ends as soon as a fixed constant number of keys are in the dictionary. If phase $l-1$ finishes with n keys in the dictionary, phase l consists of processing the next $n/2$ instructions. For each phase l we use a temporary dictionary D_{NEW} to keep track of the instructions of this phase. D_{NEW} is a Type-B dictionary. At the beginning of phase l, D_{NEW} is empty. A further dictionary, D_{OLD}, is the dictionary D_{NEW} of phase $l-1$ at the end of phase $l-1$ (cf. Remark 6). The complete dictionary in the state of the beginning of phase $l-1$ is stored in D_{OLDBACK}. D_{OLDBACK} is a Type-A dictionary with $\varepsilon = 1$. D_{NEWBACK} is of the same type; it is empty at the beginning of phase l. During phase l, the keys from D_{OLDBACK} and D_{OLD} will be inserted in the dictionary D_{NEWBACK}.

Assume that presently n keys are stored in the dictionary. We describe phase l.

Preparation: Set up an empty dictionary D_{NEW} of capacity $\frac{1}{2}n$ and an empty dictionary D_{NEWBACK} of capacity n.

Process Instructions:

This is done in $\frac{1}{2}n$ rounds. In each round one instruction is read and executed.

(a) Insertions are made into D_{NEW}.

(b) Deletions are executed by entering the key with tag "deleted" into D_{NEW}.

(c) Lookups are executed by searching for the key in $D_{\text{NEW}}, D_{\text{OLD}}, D_{\text{OLDBACK}}$, in this order.

Also, in each round, a constant number of steps of the following procedure is performed:

Construct D_{NEWBACK}**:**

Collect the keys from D_{OLD} and D_{OLDBACK} into one sequential list. Eliminate keys marked "deleted" in D_{OLD}. (This leaves a list of those n keys present in the dictionary at the end of the previous phase.) Insert the keys from the list into D_{NEWBACK}. When the construction of D_{NEWBACK} is finished, rename D_{NEWBACK} into D_{OLDBACK} and mark D_{OLD} "empty". Then clear the two old dictionaries.

After all $n/2$ rounds that belong to phase l are finished, D_{NEW} is declared "closed" and the names of D_{NEW} and D_{OLD} are switched.

This finishes the description of phase l.

Analysis:

If we set the constant large enough that directs how many steps of the construction of D_{NEWBACK} are performed in each round, this construction will be finished within $n/2$ rounds, with probability exceeding $1 - O(n^{-c})$, by Theorem 7. Each operation in D_{NEW} will take constant time with probability $1 - O(n^{-c})$, by Theorem 8. Thus

the overall probability that each round takes constant time and the construction of D_{NEWBACK} is finished after $n/2$ rounds is $1 - O(n^{-c})$. It is clear that $O(n)$ space is used. The correctness proof is similar to that in Theorem 8. □

Remark 7 *Recovery from failure.* If in a phase any one of the dictionaries or procedures fails to perform properly, i. e., exceeds its time bounds, we abort the phase. This happens with probability $O(n^{-c})$. The following recovery procedure is invoked in this case: All keys currently stored in the dictionary are collected and inserted into a new background dictionary D_{NEWBACK}. No instructions are accepted during this time. The expected time needed for this reconstruction is $O(n)$. Note that no information stored in the dictionary is lost if such a failure occurs.

6 Parallel dynamic hashing in real time

In this section we present a parallel version of the real-time dictionary, thus proving Theorem 2.

The basic idea of the algorithm is simple: Let M be a p-processor CRCW PRAM with processors $P_1, \ldots, P_p$ and ARBITRARY write-conflict resolution. Let $0 < \varepsilon < 1$ be fixed. A set S of $n \geq p^{1+\varepsilon}$ keys is stored in the shared memory as follows: S is split into $s = n^{1-\varepsilon/2}$ buckets $B_0^H, \ldots, B_{s-1}^H$ by a hash function $H \in \mathcal{H}_s^d$, for sufficiently large d. For each $i \in \{0, \ldots, s-1\}$, the keys from B_i^H are organized as a sequential real time dictionary D_i as presented in the previous section. The following algorithm shows how to perform n instructions, n/p per processor, in a real-time manner in space $O(n)$ if $n \geq p^{1+\varepsilon}$.

Algorithm 4 (Parallel dictionary) Let $(x_1^t, Op_1^t), \ldots, (x_p^t, Op_p^t)), t = 1, \ldots, n/p$ be the sequence of p-tuples of instructions to be executed, where (x_r^t, op_r^t) is known to processor P_r. Let $S = \{x_1, \ldots, x_n\}$ be the set of keys occurring in the n instructions. Let $s := n^{1-\varepsilon/2}$. Set up empty real-time dictionaries D_i, $0 \leq i < s$, each suitable for performing $2n^{\varepsilon/2}$ instructions, of the kind described in Algorithm 2.

- For $t := 1$ to n/p do
  - For all $r \in \{1, \ldots, p\}$ in parallel do
    - P_r computes $i_r := H(x_r^t)$.
    - P_r executes (x_r^t, Op_r^t) in the sequential dictionary D_{i_r}. Different instructions arriving at the same D_i are executed sequentially. If several update instructions (x, Op_r^t), with the same key x arrive at the same D_i simultaneously, only one deletion (if any), then an arbitrary one of the insertions (if any) is executed. (Note that lookups for the same key do not interfere, since concurrent reading is permitted.)

Lemma 4 *For each $c \geq 1$ and $0 < \varepsilon < 1$, there is a $d \geq 2$ such that Algorithm 4 needs constant time for each pass of the outer loop, with probability at least $1 - n^{-c}$.*

Proof: By Facts 3 and 1 and by Theorem 8, each of the following properties hold with probability at least $1 - n^{-c'}$, for arbitrary $c' \geq 1$, if d is sufficiently large.

(i) H splits S into buckets of (roughly equal) size at most $2n^{\varepsilon/2}$.

(ii) For each $t = 1, \ldots, n/p$, the function H is $(d-1)$-perfect on $\{x_1^t, \ldots, x_p^t\}$; in particular, at most $d-1$ different instructions arrive at each D_i during the tth pass through the outer loop.

(iii) For each $i \in \{0, \ldots, s-1\}$, the instructions arriving at D_i are executed in real time.

Thus, if c' is sufficiently large, (i), (ii) for all $t = 1, \ldots, n/p$, and (iii) for $i \in \{0, \ldots, s-1\}$ hold simultaneously with probability at least $1 - O(n^{-c})$. As each pass of the outer loop needs constant time if (i), (ii), (iii) hold, the lemma follows. ■

In order to obtain the general parallel dictionary as described in Theorem 2, we have to show how to handle the situation where the size of the dictionary is not known in advance. This is done in a way very similar to the method used in the last section to get the general sequential dictionary from Type-A and Type-B dictionaries; the details are omitted here.

Bibliography

[1] H. V. Aho and D. Lee. Storing a dynamic sparse table. In *Proc. of the 27th IEEE Ann. Symp. on Foundations of Computer Science*, pages 55–60. IEEE, 1986.

[2] H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *Proc. of the 3rd Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 50–61, 1991.

[3] G. Brassard and S. Kannan. The generation of random permutations on the fly. *Inform. Proc. Letters*, **28**:207–212, 1988.

[4] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. Technical Report 77, Universität-GH-Paderborn, Fachbereich Mathematik/Informatik, Jan. 1991. *Revised Version* of the paper of the same title that appeared in *Proc. of the 29th IEEE Ann. Symp. on Foundations of Computer Science*, pages 524–531, 1988.

[5] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. of the 1989 ACM Symp. on Parallel Algorithms and Architectures*, pages 360–368, 1989. (Revised version to appear in *Information and Computation*).

[6] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 117–127, 1990.

[7] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In M. S. Paterson, editor, *Proceedings of 17th ICALP*, pages 6–19. Springer, 1990. Lecture Notes in Computer Science 443.

[8] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, **31**(3):538–544, July 1984.

[9] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. of the 32nd IEEE Ann. Symp. on Foundations of Computer Science*, 1991.

[10] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. Assoc. Comput. Mach.*, **28**(2):289–304, Apr. 1981.

[11] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Inform. Proc. Letters*, **33**:305–308, 1989/90.

[12] W. Hoeffding. Probability inequalites for sums of bounded random variables. *J. Am. Stat. Ass.*, **58**:13–30, 1963.

[13] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.*, **71**:95–132, 1990.

[14] R. J. Lipton and J. G. Naughton. Clocked adversaries for hashing. Technical Report CS-TR-203-89, Princeton, 1989.

[15] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, pages 307–316, 1991.

[16] C. McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics, 1989*, pages 148–188. Cambridge University Press, 1989. London Math. Soc. Lecture Note Series 141.

[17] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Berlin, 1984.

[18] J. H. Reif and P. G. Spirakis. Random matroids. In *Proc. of the 12th Ann. ACM Symp. on Theory of Computing*, pages 385–397, 1980.

[19] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. of the 30th IEEE Ann. Symp. on Foundations of Computer Science*, pages 20–25, 1989. *Revised Version.*

[20] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, chapter 18, pages 943–971. Elsevier, Amsterdam, 1990.