Cuckoo Hashing

Rasmus Pagh
Flemming Friche Rodler

BRICS Report Series RS-01-32

ISSN 0909-0878 August 2001

Cuckoo Hashing

Rasmus Pagh[*] and Flemming Friche Rodler
BRICS[†]
Department of Computer Science
University of Aarhus, Denmark
{pagh,ffr}@brics.dk

August, 2001

Abstract

We present a simple and efficient dictionary with worst case constant lookup time, equaling the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. (*Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput., 23(4):738–761, 1994*). The space usage is similar to that of binary search trees, i.e., three words per key on average. The practicality of the scheme is backed by extensive experiments and comparisons with known methods, showing it to be quite competitive also in the average case.

1 Introduction

The *dictionary* data structure is ubiquitous in computer science. A dictionary is used to maintain a set S under insertion and deletion of elements (referred to as *keys*) from a universe U. Membership queries ("$x \in S$?") provide access to the data. In case of a positive answer the dictionary also provides a piece of *satellite data* that was associated with x when it was inserted.

A large literature, surveyed in Section 1.1, is devoted to practical and theoretical aspects of dictionaries. It is common to study the case where keys are bit strings in $U = \{0,1\}^w$ and w is the word length of the computer (for theoretical purposes modeled as a RAM). Section 2 discusses this restriction. It is usually, though not always, clear how to return associated information once membership has been determined. E.g., in all methods discussed in this paper, the associated information of $x \in S$ can be stored together with x in a hash table. Therefore we disregard the time and space used to handle associated information and concentrate on the problem of maintaining S. In the following we let n denote $|S|$.

[*]Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Work initiated while visiting Stanford University.

[†]Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

The most efficient dictionaries, in theory and in practice, are based on hashing techniques. The main performance parameters are of course lookup time, update time, and space. In theory there is no trade-off between these. One can simultaneously achieve constant lookup time, expected amortized constant update time, and space within a constant factor of the information theoretical minimum of $B = \log \binom{|U|}{n}$ bits [5]. In practice, however, the various constant factors are crucial for many applications. In particular, lookup time is a critical parameter. It is well known that the expected time for all operations can be made a within a factor of $(1+\epsilon)$ from optimal (one universal hash function evaluation, one memory lookup) if space $O(n/\epsilon)$ is allowed. Therefore the challenge is to combine speed with a reasonable space usage. In particular, we only consider schemes using $O(n)$ words of space.

The contribution of this paper is a new, simple hashing scheme called *cuckoo hashing*. A description and analysis of the scheme is given in Section 3, showing that it possesses the same theoretical properties as the dynamic dictionary of Dietzfelbinger et al. [10]. That is, it has worst case constant lookup time and amortized expected constant time for updates. A special feature of the lookup procedure is that (disregarding accesses to a small hash function description) there are just two memory accesses, which are *independent* and can be done in parallel if this is supported by the hardware. Our scheme works for space similar to that of binary search trees, i.e., three words per key in S on average.

Using weaker hash functions than those required for our analysis, cuckoo hashing is very simple to implement. Section 4 describes such an implementation, and reports on extensive experiments and comparisons with the most commonly used methods, having no worst case guarantee on lookup time. It seems that an experiment comparing the most commonly used methods on a modern multi-level memory architecture has not previously been described in the literature. Our experiments show cuckoo hashing to be quite competitive, especially when the dictionary is small enough to fit in cache. We thus believe it to be attractive in practice, when a worst case guarantee on lookups is desired.

1.1 Previous Work on Linear Space Dictionaries

Hashing, first described in public literature by Dumey [12], emerged in the 1950s as a space efficient heuristic for fast retrieval of keys in sparse tables. Knuth surveys the most important classical hashing methods in [17, Section 6.4]. These methods also seem to prevail in practice. The most prominent, and the basis for our experiments in Section 4, are Chained Hashing (with separate chaining), Linear Probing and Double Hashing. A more recent scheme called Two-Way Chaining will also be investigated. We detail our implementation in Section 4.

1.1.1 Theoretical Work.

Early theoretical analysis of hashing schemes was typically done under the assumption that hash function values were uniformly random and independent. Precise analyses of the average and expected worst case behaviors of the above-mentioned schemes have been made, see e.g. [14, 17]. We mention just a few facts, disregarding asymptotically vanishing terms.

For Linear Probing the expected number of memory probes for succesful and unsuccessful lookups are $\frac{1}{2}(1+\frac{1}{1-\alpha})$ and $\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$, respectively, where α denotes the fraction of the table occupied by keys, $0 < \alpha < 1$. The *longest* probe sequence is of expected length $\Omega(\log n)$. In Double Hashing the expected cost of successful and unsuccessful lookups are, respectively, $\ln(\frac{1}{1-\alpha})/\alpha$ and $\frac{1}{1-\alpha}$. The longest successful probe sequence is expected to be of length $\Omega(\log n)$, and there is no bound on the length of unsuccessful searches. For Chained Hashing lookups have expected cost $1 + \alpha/2$ and $1 + \alpha^2/2$, respectively, for hash table size n/α. The expected maximum chain length is $\Theta(\log n/\log\log n)$. In terms of number of *probes*, the above implies that Chained Hashing is better than Double Hashing, which is again better than Linear Probing. Note that for these three schemes, an insertion corresponds to an unsuccessful lookup, and that a deletion corresponds to a successful lookup.

Two-Way Chaining is an alternative to Chained Hashing that offers $O(\log\log n)$ expected maximal lookup time. The implementation that we consider represents the lists by arrays of size $O(\log\log n)$. To achieve linear space usage, one must then use a hash table of size $O(n/\log\log n)$, implying that the *average* chain length is $\Omega(\log\log n)$. Another scheme with expected $O(\log\log n)$ time per operation is *Multilevel Adaptive Hashing* [3]. However, lookups can be performed in $O(1)$ worst case time if $O(\log\log n)$ hash function evaluations, memory probes and comparisons are possible in parallel. This is similar to the scheme described in this paper, though we use only two hash function evaluations, memory probes and comparisons.

Though the results seem to agree with practice, the randomness assumptions used for the above analyses are questionable in applications. Carter and Wegman [6] succeeded in removing such assumptions from the analysis of chained hashing, introducing the concept of *universal* hash function families. When implemented with a random function from Carter and Wegman's universal family, chained hashing has constant expected time per dictionary operation (plus an amortized expected constant cost for resizing the table). Constructions of universal hash function families with very efficient evaluation have since appeared [7, 9, 25].

A dictionary with worst case constant lookup time was first obtained by Fredman, Komlós and Szemerédi [13], though it was *static*, i.e., did not support updates. It was later augmented with insertions and deletions in amortized expected constant time by Dietzfelbinger et al. [10]. Dietzfelbinger and Meyer auf der Heide [11] improved the update performance by exhibiting a dictionary in which operations are done in constant time with high probability, namely at least $1 - n^{-c}$, where c is any constant of our choice. A simpler dictionary with the same properties was later developed [8]. When $n = |U|^{1-o(1)}$ a space usage of $O(n)$ words is not within a constant factor of the information theoretical minimum. The dictionary of Brodnik and Munro [5] offers the same performance as [10], using $O(B)$ bits in all cases.

1.1.2 Experimental Work.

Although the above results leave little to improve from a theoretical point of view, large constant factors and complicated implementation hinder direct practical use. For example, in the "dynamic perfect hashing" scheme of [10] the upper bound on space is $35n$ words. The authors of [10] refer to a more practical variant due to Wenzel that uses space comparable to that of binary search trees.

According to [16] the implementation of this variant in the LEDA library [20], described in [26], has average insertion time larger than that of AVL trees for $n \leq 2^{17}$, and more than four times slower than insertions in chained hashing[1]. The experimental results listed in [20, Table 5.2] show a gap of more than a factor of 6 between the update performance of chained hashing and dynamic perfect hashing, and a factor of more than 2 for lookups[2].

Silverstein [24] reports that the space upper bound of the dynamic perfect hashing scheme of [10] is quite pessimistic compared to what can be observed when run on a subset of the DIMACS dictionary tests [19]. He goes on to explore ways of improving space as well as time, improving both the observed time and space by a factor of roughly three. Still, the improved scheme needs 2 to 3 times more space than an implementation of linear probing to achieve similar time per operation. Silverstein also considers versions of the data strucures with packed representations of the hash tables. In this setting the dynamic perfect hashing scheme was more than 50% slower than linear probing, using roughly the same amount of space.

Is seems that recent experimental work on "classical" dictionaries (that do not have worst case constant lookup time) is quite limited. In [16] it is reported that chained hashing is superior to an implementation of dynamic perfect hashing in terms of both memory usage and speed. Judging from leading textbooks on algorithms, Knuth's selection of algorithms is in agreement with current practice for implementation of general purpose dictionaries. In particular, the excellent cache usage of LINEAR PROBING makes it a prime choice on modern architectures.

2 Preliminaries

We assume that keys from U fit exactly in a single machine word, that is, $U = \{0,1\}^w$. A special value $\bot \in U$ is reserved to signal an empty cell in hash tables. For DOUBLE HASHING an additional special value is used to indicate a deleted key.

Our algorithm uses hash functions from a *universal* family.

Definition 1 *A family $\{h_i\}_{i \in I}$, $h_i : U \to R$, is (c,k)-*universal *if, for any k distinct elements $x_1, \ldots, x_k \in U$, any $y_1, \ldots, y_k \in R$, and uniformly random $i \in I$, $\Pr[h_i(x_1) = y_1, \ldots, h_i(x_k) = y_k] \leq c/|R|^k$.*

[1]On a Linux PC with an Intel® Pentium® 120 MHz processor.

[2]On a 300 MHz SUN ULTRA SPARC.

A standard construction of a $(2,k)$-universal family for range $R = \{0,\ldots,r-1\}$ and prime $p > 2^w$ is

$$\{x \mapsto ((\sum_{l=0}^{k-1} a_l x^l) \bmod p) \bmod r \quad | \quad 0 \leq a_0, a_1, \ldots, a_{k-1} < p\} \ . \tag{1}$$

If U is not too large compared to k, there exists a space-efficient $(2,k)$-universal family due to Siegel [23] that has *constant* evaluation time (however, the constant factor of the evaluation time is rather high).

Theorem 1 (Siegel) *There is a constant c such that for, $k = 2^{\Omega(w)}$, there exists a $(2,k)$-universal family, using space and initialization time k^c, that can be evaluated in* constant *time.*

The restriction that keys are single words is not a serious one. Longer keys can be mapped to keys of $O(1)$ words by applying a random function from a $(O(1),2)$-universal family. There is such a family whose functions can be evaluated in time linear in the number of input words [6]. It works by evaluating a function from a $(O(1),2)$-universal family on each word, computing the bitwise exclusive or of the function values. (See [25] for an efficient implementation). Such a function with range $\{0,1\}^{2\log(n)+c}$ will, with probability $1-O(2^c)$, be injective on S. In fact, with constant probability the function is injective on a given *sequence* of $\Omega(2^{c/2}n)$ consecutive sets in a dictionary of initial size n (see [10]). When a collision between two elements of S occurs, everything is rehashed. If a rehash can be done in expected $O(n)$ time, the amortized expected cost of this is $O(2^{-c/2})$ per insertion. In this way we can effectively reduce the universe size to $O(n^2)$, though the full keys still need to be stored to decide membership. For $c = O(\log n)$ the new keys are of length $2\log n + O(1)$. For any $\epsilon > 0$, Theorem 1 then provides a family of constant time evaluable $(2, n^{\Omega(1)})$-universal hash functions, whose functions can be stored using space n^ϵ.

3 Cuckoo Hashing

Cuckoo hashing is a dynamization of a static dictionary described in [21]. The dictionary uses two hash tables, T_1 and T_2, of length r and two hash functions $h_1, h_2 : U \to \{0,\ldots,r-1\}$. Every key $x \in S$ is stored in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2, but never in both. Our lookup function is

```
function lookup($x$)
   return $T_1[h_1(x)] = x \ \vee \ T_2[h_2(x)] = x$
end
```

Two table accesses are in fact optimal among all data structures using linear space, except for special cases, see [21].

Remark: The idea of storing keys in one out of two places given by hash functions previously appeared in [15] in the context of PRAM simulation, and in [2] for TWO-WAY CHAINING, mentioned in Section 1.1.1.

It is shown in [21] that if $r \geq (1+\epsilon)\,n$ for some constant $\epsilon > 0$ (i.e., the tables are to be a bit less than half full), and h_1, h_2 are picked uniformly at random from an $(O(1), O(\log n))$-universal family, the probability that there is no way of arranging the keys of S according to h_1 and h_2 is $O(1/n)$. By the discussion in Section 2 we may assume without loss of generality that there is such a family, with constant evaluation time and negligible space usage. A suitable arrangement was shown in [21] to be computable in linear time by a reduction to 2-SAT.

We now consider a simple dynamization of the above. Deletion is of course simple to perform in constant time, not counting the possible cost of shrinking the tables if they are becoming too sparse. As for insertion, it turns out that the "cuckoo approach", kicking other keys away until every key has its own "nest", works very well. Specifically, if x is to be inserted we first see if cell $h_1(x)$ of T_1 is occupied. If not, we are done. Otherwise we set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "nestless". This key is then inserted in T_2 in the same way, and so forth, see Figure 1(a). As it may happen that this process loops, see Figure 1(b), the number of iterations is bounded by a value "MaxLoop" to be specified in Section 3.1. If this number of iterations is reached everything is rehashed with new hash functions, and we try once again to accommodate the nestless key. Using the notation $x \leftrightarrow y$ to express that the values of variables x and y are swapped, the following code summarizes the insertion procedure.

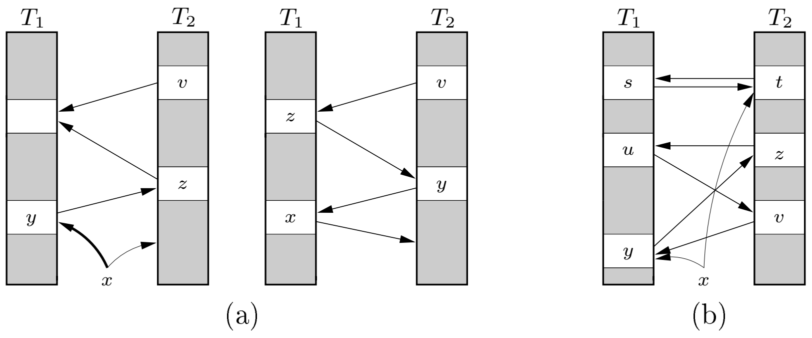


Figure 1: (a) Key x is successfully inserted by moving keys y and z to the other table. (b) Key x cannot be accomodated and a rehash is necessary.

```
procedure insert($x$)
  if lookup($x$) then return
  loop MaxLoop times
    if $T_1[h_1(x)] = \perp$ then { $T_1[h_1(x)] \leftarrow x$; return }
    $x \leftrightarrow T_1[h_1(x)]$
    if $T_2[h_2(x)] = \perp$ then { $T_2[h_2(x)] \leftarrow x$; return }
    $x \leftrightarrow T_2[h_2(x)]$
  end loop
  rehash(); insert($x$)
end
```

The above procedure assumes that the tables remain larger than $(1+\epsilon)\,n$ cells. When no such bound is known, a test must be done to find out when a rehash to larger tables is needed.

The lookup call preceding the insertion in the procedure ensures robustness if the key to be inserted is already in the dictionary. A slightly faster implementation can be obtained if this is known not to occur.

Note that the insertion procedure is biased towards inserting keys in T_1. As will be seen in Section 4 this leads to faster successful lookups, due to more keys being found in T_1. The insertion time is only slightly worse than that of a more symmetric implementation. This effect is even more pronounced if one uses an *asymmetric* scheme where T_1 is larger than T_2. Another variant is to use a single table for both hash functions, but this requires keeping track of the hash function according to which each key is placed. In the following we consider just the basic two table scheme.

3.1 Analysis

Our analysis has two main parts:

- In Section 3.1.1 we consider what happens if one tries arbitrarily long to insert the new key, i.e., for MaxLoop $=\infty$. We show that if the insertion procedure does not terminate, it is not possible to accommodate all the keys of the new set using the present hash functions, and a rehash is necessary. In conjunction with the result from [21], this shows that the insertion procedure loops without limit with probability $O(1/n)$.
- In Section 3.1.2 we turn to the analysis for the case where insertion is possible, showing that the insertion procedure terminates in $O(1)$ iterations, in the expected sense.

This accounts for the claimed time bound, except for the cost of rehashing. A rehash has no failed insertions with probability $1-O(1/n)$. In this case, the expected time per insertion is constant, so the expected time is $O(n)$. As the probability of having to start over with new hash functions is bounded away from 1, the total expected time for a rehash is $O(n)$. This implies that the expected time for insertion is constant if $r \geq (1+\epsilon)(n+1)$. Resizing of tables can be done in amortized expected constant time per update by the usual doubling/halving technique (see e.g. [10]).

3.1.1 The insertion procedure loops

Consider the sequence $x_1, x_2, \ldots$ of nestless keys in the infinite loop. For $i, j \geq 1$ we define $X_{i,j} = \{x_i, \ldots, x_j\}$. Let j be the smallest index such that $x_j \in X_{1,j-1}$, and let l be the minimum index such that $l+1 > j$ and $x_{l+1} \in X_{1,l}$.

We now argue that the first l steps of the insertion proceeds as depicted in Figure 2. The topmost configuration is the one preceeding the insertion of x_1. The configuration just before x_j becomes nestless for the second time is shown in the middle. One step later we have that x_k is now in the previous location of x_{k+1}, for $1 \leq k < j$. Let $i < j$ be the index such that $x_i = x_j$. We now consider

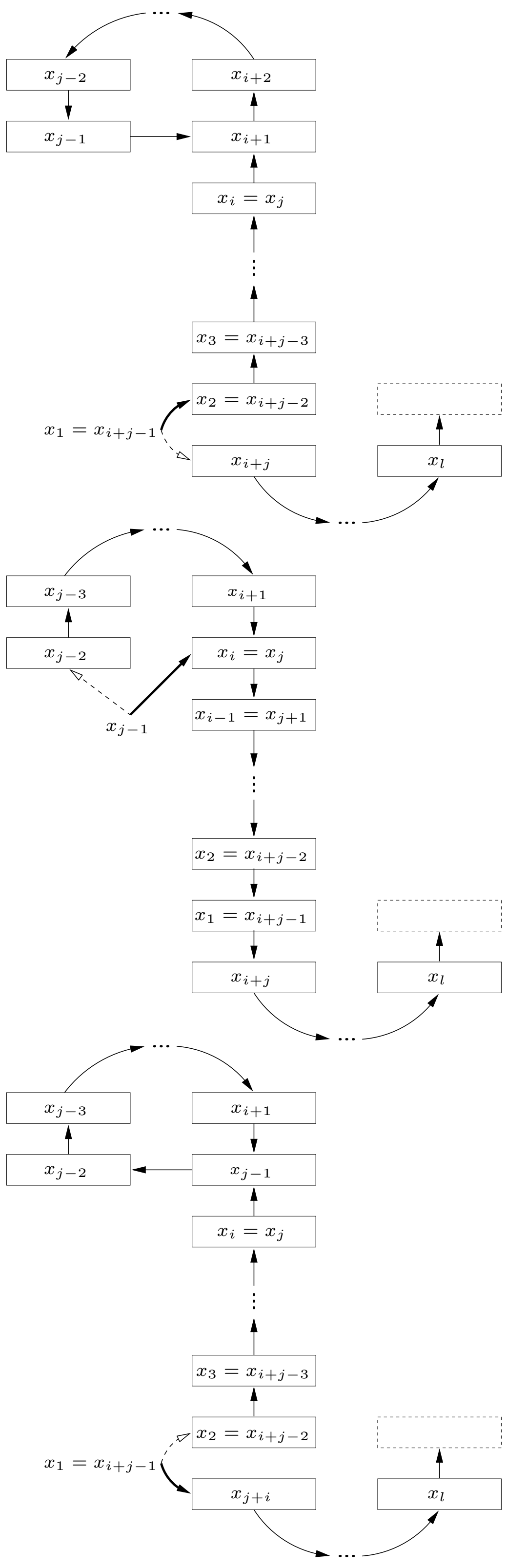


Figure 2: Stages of an insertion of key x_1. Boxes correspond to cells in either of the two tables, and arcs indicate the other possible position of a key. Bold arcs show where the nestless key is to be inserted.

what happens towards the third stage. If $i > 1$ then x_j reclaims its previous location, occupied by x_{i-1}. If $i > 2$ then x_{i-1} subsequently reclaims its previous position, which is occupied by x_{i-2}, and so forth. Thus we have $x_{j+z} = x_{i-z}$ for $z = 0, 1, \ldots, i-1$, and end up with x_1 occurring again as x_{i+j-1}. This is shown in the third stage of the figure. Note that the dotted cell must, by definition of l, be identical to one of the other cells in the figure.

It is now not hard to see that the number of cells is not sufficient to accommodate $X_{i,l}$ for the current choice of hash functions. For a formal proof, we define $s_k = |h_1[X_{1,k}]| + |h_2[X_{1,k}]|$, i.e., s_k is the number of table cells available to $X_{1,k}$. Obviously $s_k \le s_{k-1} + 1$, as every key x_i, $i > 1$, has either $h_1(x_i) = h_1(x_{i-1})$ or $h_2(x_i) = h_2(x_{i-1})$. In fact, $s_{j-1} = s_{j-2} \le j-1$, because the key x_j found in $T_1[h_1(x_{j-1})]$ or $T_2[h_2(x_{j-1})]$ occurred earlier in the sequence. As all of the keys $x_j, \ldots, x_{j+i-1}$ appeared earlier in the sequence, we have $s_{j+i-2} = s_{j-2}$. Similar to before we have $s_l = s_{l-1}$. In conclusion, $|X_{1,l}| = l + 1 - i$ and $s_l = s_{l-1} \le s_{j+i-2} + (l-1) - (j+i-2) = s_{j-2} + l + 1 - j - i < l + 1 - i$.

3.1.2 Successful insertion

Consider a prefix $x_1, x_2, \ldots, x_l$ of the sequence of nestless keys. The crucial fact is that there must be a subsequence of at least $l/3$ keys without repetitions, starting with an occurrence of the key x_1, i.e., the inserted key. If there is no repetion this is obvious. Otherwise the first l steps of the insertion proceeds as in Figure 2. In particular, one of the sequences $x_1, \ldots, x_{j-1}$ and $x_{j+i-1}, \ldots, x_l$ is the desired one of length at least $l/3$.

Suppose that the insertion loop runs for at least t iterations. By the above there is a sequence of distinct keys $b_1, \ldots, b_m$, $m \ge (2t-1)/3$, such that b_1 is the key to be inserted, and such that for some $\beta \in \{0, 1\}$

$$h_{2-\beta}(b_1) = h_{2-\beta}(b_2),\ h_{1+\beta}(b_2) = h_{1+\beta}(b_3),\ h_{2-\beta}(b_3) = h_{2-\beta}(b_4), \ldots \tag{2}$$

Given b_1 there are at most n^{m-1} sequences of m distinct keys. For any such sequence and any $\beta \in \{0, 1\}$, if the hash functions were chosen from a (c, m)-universal family, the probability that (2) holds is bounded by $c\,r^{-(m-1)}$. Thus, the probability that there is *any* sequence of length m satisfying (2) is bounded by $2c\,(n/r)^{m-1} \le 2c\,(1+\epsilon)^{-(2t-1)/3+1}$. Suppose we use a $(c, 6\log_{1+\epsilon} n)$-universal family, for some constant c. Then the probability of more than $3\log_{1+\epsilon} n$ iterations is $O(1/n^2)$. Thus, we can set MaxLoop $= 3\log_{1+\epsilon} n$ with a negligible increase in the probability of a rehash. When there is no rehash the expected number of iterations is at most

$$\begin{aligned} 1 + & \sum_{t=2}^{\infty} 2c\,(1+\epsilon)^{-(2t-1)/3+1} \qquad (3) \\ &= 1 + 2c(1+\epsilon)^{4/3} \sum_{t=0}^{\infty} ((1+\epsilon)^{-2/3})^t \\ &= 1 + O(\tfrac{1}{1-(1+\epsilon)^{-2/3}}) \\ &= O(1 + 1/\epsilon)\ . \end{aligned}$$

4 Experiments

To examine the practicality of Cuckoo Hashing we experimentally compare it to three well known hashing methods, as described in [17, Section 6.4]: Chained Hashing (with separate chaining), Linear Probing and Double Hashing. We also consider Two-Way Chaining [2].

The first three methods all attempt to store a key x at position $h(x)$ in a hash table. They differ in the way collisions are resolved, i.e., what happens when two or more keys hash to the same location.

Chained Hashing. A chained list is used to store all keys hashing to a given location.

Linear Probing. A key is stored in the next empty table entry. Lookup of key x is done by scanning the table beginning at $h(x)$ and ending when either x or an empty table entry is found. When deleting, some keys may have to be moved back in order to fill the hole in the lookup sequence, see [17, Algoritm R] for details.

Double Hashing. Insertion and lookup are similar to Linear Probing, but instead of searching for the next position one step at a time, a second hash function value is used to determine the step size. Deletions are handled by putting a “deleted” marker in the cell of the deleted key. Lookups skip over deleted cells, while insertions overwrite them.

Two-Way Chaining can be described as two instances of Chained Hashing. A key is inserted in one of the two hash tables, namely the one where it hashes to the shortest chain. A cache-friendly implementation, as recently suggested in [4], is to simply make each chained list a short, fixed size array. If a longer list is needed, a rehash must be performed.

4.1 Data Structure Design and Implementation

We consider positive 32 bit signed integer keys and use 0 as $\perp$. The data structures are *robust* in that they correctly handle attempts to insert an element already in the set, and attempts to delete an element not in the set. During rehashes this is known not to occur and slightly faster versions of the insertion procedure is used.

Our focus is on achieving high performance dictionary operations with a reasonable space usage. By the *load factor* of a dictionary we will understand the size of the set relative to the memory used[3]. As seen in [17, Figure 44] the speed of Linear Probing and Double Hashing degrades rapidly for load factors above 1/2. On the other hand, none of the schemes improve much for load factors below 1/4. As Cuckoo Hashing only works when the size of each table is larger than the size of the set, we can only perform a comparison for load factors less than 1/2. To allow for doubling and halving of the table size,

[3]For Chained Hashing, the notion of load factor traditionally disregards the space used for chained lists, but we desire equal load factors to imply equal memory usage.

we allow the load factor to vary between 1/5 and 1/2, focusing especially on the "typical" load factor of 1/3. For Cuckoo Hashing and Two-Way Chaining there is a chance that an insertion may fail, causing a "forced rehash". If the load factor is larger than a certain threshold, somewhat arbitrarily set to 5/12, we use the opportunity to double the table size. By our experiments this only slightly decreases the average load factor.

Apart from Chained Hashing, the schemes considered have in common the fact that they have only been analyzed under randomness assumptions that are currently, or inherently, unpractical to implement ($O(\log n)$-wise independence or n-wise independence). However, experience shows that rather simple and efficient hash function families yield performance close to that predicted under stronger randomness assumptions. We use a function family from [9] with range $\{0,1\}^q$ for positive integer q. For every odd a, $0 < a < 2^w$, the family contains the function $h_a(x) = (ax \bmod 2^w) \text{ div } 2^{w-q}$. Note that evaluation can be done very efficiently by a 32 bit multiplication and a shift. However, this choice of hash function restricts us to consider hash tables whose sizes are powers of two. A random function from the family (chosen using C's `rand` function) appears to work fine with all schemes except Cuckoo Hashing. For Cuckoo Hashing we experimented with various hash functions and found that Cuckoo Hashing was rather sensitive to the choice of hash function. It turned out that the exclusive or of three independently chosen functions from the family of [9] was fast and worked well. We have no good explanation for this phenomenon. For all schemes, various alternative hash families were tried, with a decrease in performance.

All methods have been implemented in C. We have striven to obtain the fastest possible implementation of each scheme. Specific choices made and details differing from the references are:

Chained Hashing. C's `malloc` and `free` functions were found to be a performance bottleneck, so a simple "freelist" memory allocation scheme is used. Half of the allocated memory is used for the hash table, and half for list elements. If the data structure runs out of free list elements, its size is doubled. We store the first element of each linked list directly in the hash table. This often saves one cache miss. It also slightly improves memory utilization, in the expected sense. This is because every non-empty chained list is one element shorter and bacause we expect more than half of the hash table cells to contain a linked list for the load factors considered here.

Double Hashing. To prevent the tables from clogging up with deleted cells, resulting in poor performance for unsuccessful lookups, all keys are rehashed when 2/3 of the hash table is occupied by keys and "deleted" markers. The fraction 2/3 was found to give a good tradeoff between the time for insertion and unsuccessful lookups.

Linear Probing. Our first implementation, like that in [24], employed deletion markers. However, we found that using the deletion method described in [17, Algoritm R] was considerably faster, as far fewer rehashes were needed.

Two-Way Chaining. We allow four keys in each bucket. This is enough to keep the probability of a forced rehash low for hundreds of thousands of keys, by the results in [4]. For larger collections of keys one should allow more keys in each bucket, resulting in general performance degradation.

Cuckoo Hashing. The architecture on which we experimented could not parallelize the two memory accesses in lookups. Therefore we only evaluate the second hash function after the first memory lookup has shown unsuccessful.

Some experiments were done with variants of Cuckoo Hashing. In particular, we considered Asymmetric Cuckoo, in which the first table is twice the size of the second one. This results in more keys residing in the first table, thus giving a slightly better average performance for successful lookups. For example, after a long sequence of alternate insertions and deletions at load factor 1/3, we found that about 76% of the elements resided in the first table of Asymmetric Cuckoo, as opposed to 63% for Cuckoo Hashing. There is no significant slowdown for other operations. We will describe the results for Asymmetric Cuckoo when they differ significantly from those of Cuckoo Hashing.

4.2 Setup

Our experiments were performed on a PC running Linux (kernel version 2.2) with an 800 MHz Intel® Pentium® III processor, and 256 MB of memory (PC100 RAM). The processor has a 16 KB level 1 data cache and a 256 KB level 2 "advanced transfer" cache. As will be seen, our results nicely fit a simple model parameterized by the cost of a cache miss and the expected number of probes to "random" locations. They are thus believed to have significance for other hardware configurations. An advantage of using the Pentium® III processor for timing experiments is its `rdtsc` instruction which can be used to measure time in clock cycles. This gives access to very precise data on the behavior of functions. In our case it also supplies a way of discarding measurements significantly disturbed by interrupts from hardware devices or the process scheduler, as these show up as a small group of timings significantly separated from all other timings. Programs were compiled using the `gcc` compiler version 2.95.2, using optimization flags `-O9 -DCPU=586 -march=i586 -fomit-frame-pointer -finline-functions -fforce-mem -funroll-loops -fno-rtti`. As mentioned earlier, we use a global clock cycle counter to time operations. If the number of clock cycles spent exceeds 5000, and there was no rehash, we conclude that the call was interrupted, and disregard the result (it was empirically observed that no operation ever took between 2000 and 5000 clock cycles). If a rehash is made, we have no way of filtering away time spent in interrupts. However, all tests were made on a machine with no irrelevant user processes, so disturbances should be minimal. On our machine it took 32 clock cycles to call the `rdtsc` instruction. These clock cycles have been subtracted from the results.

4.3 Results

Dictionaries of Stable Size

Our first test was designed to model the situation in which the size of the dictionary is not changing too much. It considers a sequence of mixed operations generated at random. We constructed the test operation sequences from a collection of high quality random bits publicly available on the Internet [18]. The sequences start by insertion of n distinct random keys, followed by $3n$ times four operations: A random unsuccessful lookup, a random successful lookup, a random deletion, and a random insertion. We timed the operations in the "equilibrium", where the number of elements is stable. For load factor 1/3 our results appear in Figure 3, which shows an average over 10 runs. As Linear Probing was consistently faster than Double Hashing, we chose it as the sole open addressing scheme in the plots. Time for forced rehashes was added to the insertion time. Results had a large variance, over the 10 runs, for sets of size 2^{12} to 2^{16}. Outside this range the extreme values deviated from the average by less than about 7%. The large variance sets in when the data structure starts to fill the level 2 cache. We believe it is due to other processes evicting parts of the data structure from cache.

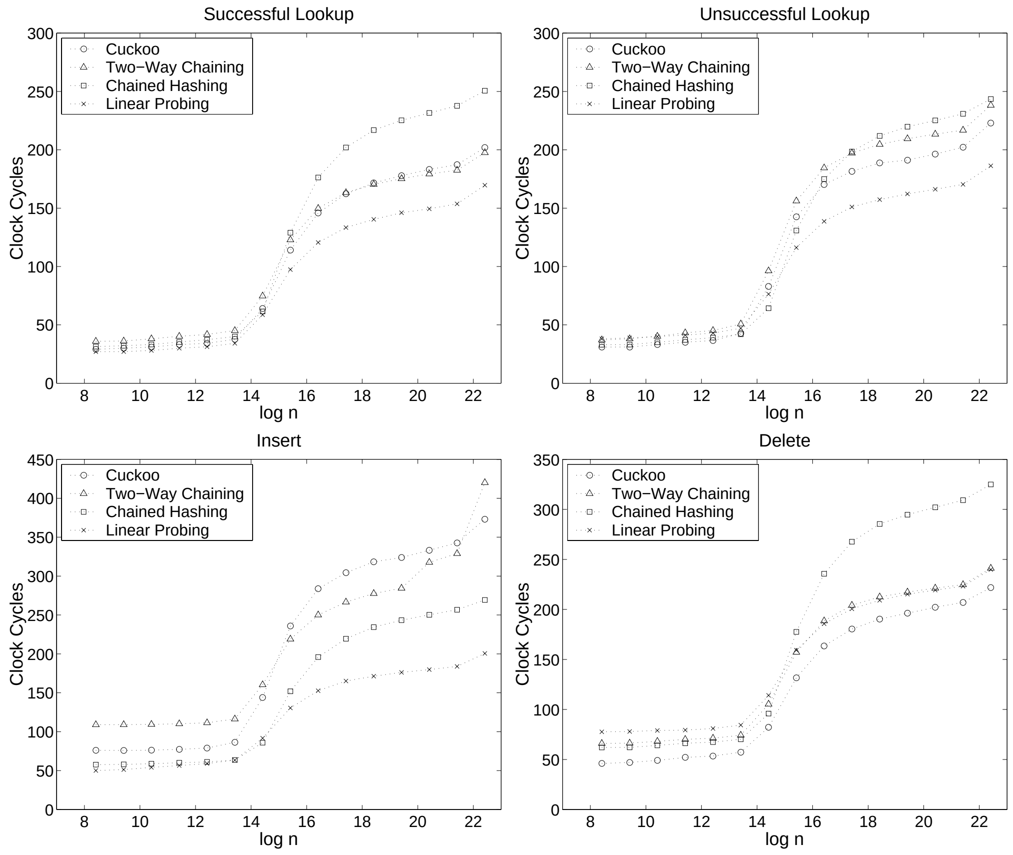


Figure 3: The average time per operation in equilibrium for load factor 1/3.

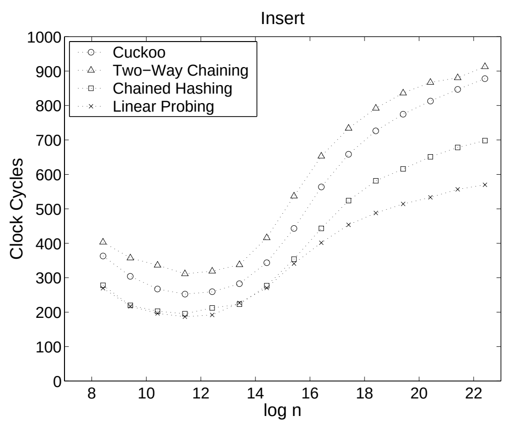
As can be seen, the time for lookups is almost identical for all schemes as long as the entire data structure fits in level 2 cache, i.e., for $n < 2^{16}/3$. After this the average number of random memory accesses (with the probability of a cache miss approaching 1) shows up. This makes linear probing an average case winner, with CUCKOO HASHING and TWO-WAY CHAINING following about 40 clock cycles behind. For insertion the number of random memory accesses again dominates the picture for large sets, while the higher number of in-cache accesses and more computation makes CUCKOO HASHING, and in particular TWO-WAY chaining, relatively slow for small sets. The cost of forced rehashes sets in for TWO-WAY CHAINING for sets of more than a million elements, at which point better results may have been obtained by a larger bucket size. For deletion CHAINED HASHING lags behind for large sets due to random memory accesses when freeing list elements, while the simplicity of CUCKOO HASHING makes it the fastest scheme. We suspect that the slight rise in time for the largest sets in the test is due to saturation of the bus, as the machine runs out of memory and begins swapping.

Growing and Shrinking Dictionaries

The third test concerns the cost of insertions in growing dictionaries and deletions in shrinking dictionaries. This will be different from the above due to the cost of rehashes. Together with Figure 3 this should give a fairly complete picture of the performance of the data structures under general sequences of operations. The first operation sequence inserts n distinct random keys, while the second one deletes them. The plot is shown in Figure 4. For small sets the time per operation seems unstable, and dominated by memory allocation overhead (if minimum table size 2^{10} is used, the curves become monotone). For sets of more than 2^{12} elements the largest deviation from the averages over 10 runs was about 6%. Disregarding the constant minimum amount of memory used by any dictionary, the average load factor during insertions was within 2% of 1/3 for all schemes except CHAINED HASHING whose average load factor was about 0.31. During deletions all schemes had average load factor 0.28. Again the fastest method is LINEAR PROBING, followed by CHAINED HASHING and CUCKOO HASHING. This is largely due to the cost of rehashes.

DIMACS Tests

Access to data in a dictionary is rarely random in practice. In particular, the cache is more helpful than in the above random tests, for example due to repeated lookups of the same key, and quick deletions. As a rule of thumb, the time for such operations will be similar to the time when all of the data structure is in cache. To perform actual tests of the dictionaries on more realistic data, we chose a representative subset of the dictionary tests of the 5th DIMACS implementation challenge [19]. The tests involving string keys were preprocessed by hashing strings to 32 bit integers, as described in Section 2. This preserves, with high probability, the access pattern to keys. For each test we recorded the average time per operation. The minimum and maximum of six runs can be found in Tables 1 and 2, which also lists the average load factor. Linear probing



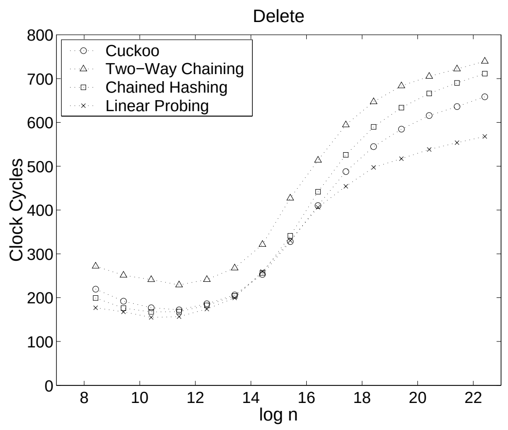


Figure 4: The average time per insertion/deletion in a growing/shrinking dictionary for average load factor $\approx 1/3$.

	Joyce		Eddington	
LINEAR	42 - 45	(.35)	26 - 27	(.40)
DOUBLE	48 - 53	(.35)	32 - 35	(.40)
CHAINED	49 - 52	(.31)	36 - 38	(.28)
A.CUCKOO	47 - 50	(.33)	37 - 39	(.32)
CUCKOO	57 - 63	(.35)	41 - 45	(.40)
TWO-WAY	82 - 84	(.34)	51 - 53	(.40)

Table 1: Average clock cycles per operation and load factors for two DIMACS string tests.

is again the fastest, but mostly just 20-30% faster than the CUCKOO schemes.

The Number of Cache Misses During Insertion

We have seen that the number of random memory accesses (i.e., cache misses) is critical to the performance of hashing schemes. Whereas there is a very precise understanding of the probe behavior of the classic schemes (under suitable randomness assumptions), the analysis of the expected time for insertions in Section 3.1 is rather crude, establishing just a constant upper bound. One reason that our calculation does not give a very tight bound is that we use a

	3.11-Q-1		Smalltalk-2		3.2-Y-1	
LINEAR	99 - 103	(.30)	68 - 72	(.29)	85 - 88	(.32)
DOUBLE	116 - 142	(.30)	77 - 79	(.29)	98 - 102	(.32)
CHAINED	113 - 121	(.30)	78 - 82	(.29)	90 - 93	(.31)
A.CUCKOO	166 - 168	(.29)	87 - 95	(.29)	95 - 96	(.32)
CUCKOO	139 - 143	(.30)	90 - 96	(.29)	104 - 108	(.32)
TWO-WAY	159 - 199	(.30)	111 - 113	(.29)	133 - 138	(.32)

Table 2: Average clock cycles per operation and load factors for three DIMACS integer tests.

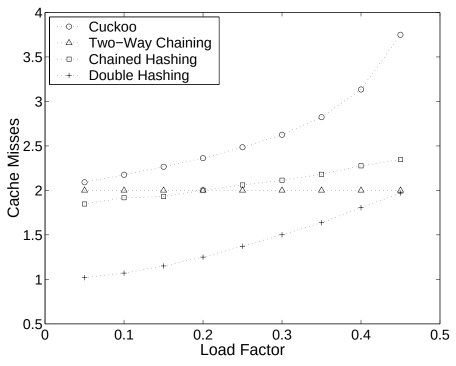


Figure 5: The average number of random memory accesses for insertion.

pessimistic estimate on the number of key moves needed to accommodate a new element in the dictionary. Often a free cell will be found even though it could have been occupied by another key in the dictionary. We also pessimistically assume that a large fraction of key moves will be spent backtracking from an unsuccessful attempt to place the new key in the first table.

Figure 5 shows experimentally determined values for the average number of probes during insertion for various schemes and load factors below 1/2. We disregard reads and writes to locations known to be in cache, and the cost of rehashes. Measurements were made in "equilibrium" after 10^5 insertions and deletions, using tables of size 2^{15} and truly random hash function values. It is believed that this curve is independent of the table size (up to vanishing terms). The curve for LINEAR PROBING does not appear, as the number of non-cached memory accesses depends on cache architecture (length of the cache line), but it is typically very close to 1. The curve for CUCKOO HASHING seems to be $2 + 1/(4 + 8\alpha) \approx 2 + 1/(4\epsilon)$. This is in good correspondance with (3) of the analysis in Section 3.1.2. As noted in Section 3, the insertion algorithm of CUCKOO HASHING is biased towards inserting keys in T_1. If we instead of starting the insertion in T_1 choose the start table at random, the number of cache misses decreases slightly for insertion. This is because the number of free cells in T_1 increases as the load balance becomes even. However, this also means a slight increase in lookup time. Also note that since insertion checks if the element is already inserted CUCKOO HASHING uses at least two cache misses. It should be remarked that the highest load factor for TWO-WAY CHAINING is $O(1/\log\log n)$.

Since lookup is very similar to insertion in CHAINED HASHING, one could think that the number of cache misses would be equal for the two operations. However, in our implementation, obtaining a free cell from the freelist may result in an extra cache miss. This is the reason why the curve for CHAINED HASHING in the figure differs from a similar plot in Knuth [17, Figure 44].

5 Model

In this section we look at a simple model of the time it takes to perform a dictionary operation, and note that our results can be explained in terms of this model. On a modern computer, memory speed is often the bottleneck. Since the operations of the investigated hashing methods mainly perform reads and writes to memory, we will assume that cache misses constitute the dominant part of the time needed to execute a dictionary operation. This leads to the following model of the time per operation.

$$\text{Time} = O + N \cdot R \cdot (1 - C/T) \ , \tag{4}$$

where the parameters of the model are described by

- O – Constant overhead of the operation.
- R – Average number of memory accesses.
- C – Cache size.
- T – Size of the hash tables.
- N – Cost of a non-cache read.

The term $R \cdot (1 - C/T)$ is the expected number of cache misses for the operations with $(1 - C/T)$ being the probability that a random probe into the tables results in a cache miss. Note that the model in not valid when the table size T is smaller than the cache size C. The size C of the cache and the size T of the dictionary are well known. From Figure 5 we can, for the various hashing schemes and for a load factor of 1/3, read the average number R of memory accesses needed for inserting an element. Note that several accesses to consecutive elements in the hash table is counted as one random access, since the other accesses are then in cache. The overhead of an operation, O, and the cost of a cache miss, N, are unknown factors that we will estimate.

Performing experiments, reading and writing to and from memory, we observed that the time for a read or a write to a location known not to be in cache could vary dramatically depending on the state of the cache. For example, when a cache line is to be used for a new read, the time used is considerably higher if the old contents of the cache line has been written to, since the old contents must first be moved to memory. For this reason we expect parameter N to depend slightly on both the particular dictionary methods and the combination of dictionary operations. This means that R and T are the only parameters not dependent on the methods used.

Using the timings from Figure 3 and the average number of cache misses for insert observed in Figure 5, we estimated N and O for the four hashing schemes. As mentioned, we believe the slight rise in time for the largest sets in the tests of Figure 3 to be caused by other non-cache related factors. So since the model is only valid for $T \geq 2^{16}$, the two parameters were estimated for time timings with $2^{16} \leq T \geq 2^{23}$. The results are shown in Table 3. As can be seen from

Method	N	O
CUCKOO	71	142
TWO-WAY	66	157
CHAINED	79	78
LINEAR	88	89
Average	76	-

Table 3: Estimated parameters according to the model for insertion.

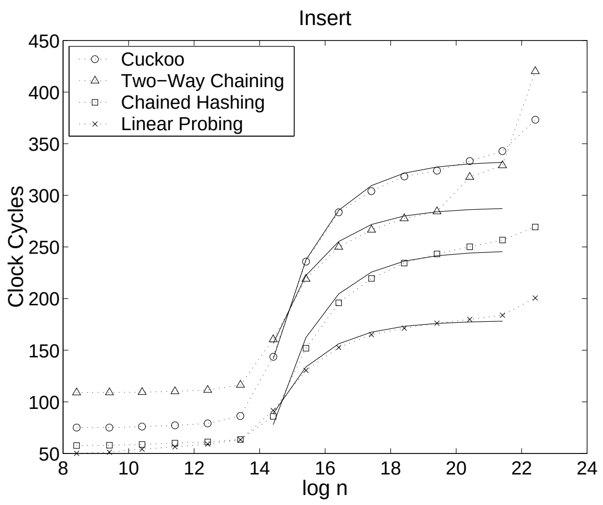


Figure 6: Model versus observed data.

the table, the cost of a cache miss varies slightly from method to method. The largest deviation from the average is about 15%.

To investigate the accuracy of our model we plotted in Figure 6 the estimated curves for insertion together with the observed curves used for estimating the parameters. As can be seen, the simple model explains the observed values quite nicely. The situation for the other operations is similar.

Having said this, we must admit that the values of N and O estimated for the schemes cannot be accounted for. In particular, it is clear that the true behavior of the schemes is more complicated than suggested by the model.

6 Conclusion

We have presented a new dictionary with worst case constant lookup time. It is very simple to implement, and has average case performance comparable to the best previous dictionaries. Earlier schemes with worst case constant lookup time were more complicated to implement and had worse average case performance. Several challenges remain. First of all an explicit practical hash function family that is provably good for the scheme has yet to be found. For example, future advances in explicit expander graph construction could make

Siegel's hash functions practical. Secondly, we lack a precise understanding of why the scheme exhibits low constant factors. In particular, the curve of Figure 5 and the fact that forced rehashes are rare for load factors quite close to 1/2 need to be explained. Another point to investigate is whether using more tables yields practical dictionaries. Experiments in [22] suggest that space utilization could be improved to more than 80%, but it remains to be seen how this would affect insertion performance.

References

[1] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 441–450. IEEE Comput. Soc. Press, Los Alamitos, CA, 1996.

[2] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200 (electronic), 1999.

[3] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 43–53. ACM Press, New York, 2000.

[4] Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. To appear in INFOCOM 2001, 2001.

[5] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640 (electronic), 1999.

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

[7] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS '96)*, pages 569–580. Springer-Verlag, Berlin, 1996.

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

[9] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

[10] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. 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 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, Berlin, 1990.

[12] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

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

[14] Gaston Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley Publishing Co., London, 1984.

[15] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

[16] Jyrki Katajainen and Michael Lykke. Experiments with universal hashing. Technical Report DIKU Report 96/8, University of Copenhagen, 1996.

[17] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., Reading, Mass., second edition, 1998.

[18] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[19] Catherine C. McGeoch. The fifth DIMACS challenge dictionaries. http://cs.amherst.edu/∼ccm/challenge5/dicto/.

[20] Kurt Mehlhorn and Stefan Näher. *LEDA. A platform for combinatorial and geometric computing.* Cambridge University Press, Cambridge, 1999.

[21] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01).* ACM Press, New York, 2001.

[22] Rasmus Pagh and Flemming Friche Rodler. Lossy dictionaries. To appear in Proceedings of ESA 2001, 2001.

[23] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20–25. IEEE Comput. Soc. Press, Los Alamitos, CA, 1989.

[24] Craig Silverstein. A practical perfect hashing algorithm. Manuscript, 1998.

[25] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 496–497. ACM Press, New York, 2000.

[26] M. Wenzel. Wörterbücher für ein beschränktes Universum. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992.