The Tree Model for Hashing: Lower and Upper Bounds

by
Joseph Gil
Friedhelm Meyer auf Der Heide and
Avi Wigderson

Technical Report 91-23
December 1991

Department of Computer Science
University of British Columbia
Rm 333 - 6356 Agricultural Road
Vancouver, B.C.
CANADA V6T 1Z2

The Tree Model for Hashing: Lower and Upper Bounds *

Joseph Gil
The University of British Columbia

Friedhelm Meyer auf Der Heide
Universität Paderborn

Avi Wigderson
The Hebrew University of Jerusalem

December 1991

Abstract

We define a new simple general model which captures many natural (sequential and parallel) hashing algorithms. In a game against nature, the algorithm and coin-tosses cause the evolution of a random tree, whose size corresponds to space (hash table size), and two notions of depth correspond to the longest probe sequences for insertion (parallel insertion time) and search of a key, respectively.

We study these parameters of hashing schemes by analyzing the underlying stochastic process, and derive tight lower and upper bounds on the relation between the amount of memory allocated to the hashing execution and the worst case insertion time. In particular, we show that if linear memory is used then not all key insertions to the hash table can be done in constant time. Except for extremely unlikely events, every input set of size n will have members for which $\Omega(\lg\lg n)$ applications of a hash function are required. From a parallel perspective it can be said that n processors need $\Omega(\lg\lg n)$ expected time to hash themselves into $O(n)$ space, although serial algorithms exist that achieve constant amortized time for insertion, as well as constant worst case search time [16].

Three variants of the basic model, which represent common hashing practice, are defined and tight bounds are presented for them as too. The most striking conclusion that can be drawn from the bounds is that, under all combinations of model variants, not all keys may be hashed in constant time.

*This work included in the first author Ph.D. dissertation [21] and appeared in a preliminary form in [25]. Research supported in part by the Leibniz Center for Research in Computer Science

1 Introduction

Hashing is one of the most important concepts in Computer Science. Its applications touch almost every aspect of this field—operating systems, file structure organization [31], communication, parallel and distributed computation, efficient algorithm design and even complexity theory [52, 53]. Nevertheless, the most common use of hashing is for the very fundamental question of efficient storage of sparse tables (see [43] and [36] for a systematic study).

A multitude of models and algorithms for hashing have been suggested and analyzed. However, almost all of them are specific in their assumptions and results. We present a simple new general model that captures many natural (sequential and parallel) hashing algorithms. In a game against nature, the algorithm and coin-tosses cause the evolution of a random tree, whose size corresponds to space (hash table size), and two notions of depth correspond to the longest probe sequences for insertion (parallel insertion time) and search of a key respectively.

A fundamental result of [20] shows that n elements from a universe of any size can be hashed to a linear size table in linear time, allowing for constant search time. It was observed, however, that although the average insertion time per element is constant, parallel application of this algorithm cannot work in constant time. The reason is that while the average is constant, some elements will have to be hashed a non-constant number of times.

Our main results exhibit tight tradeoffs between space and parallel time, in the basic model and three variants which capture standard hashing practice.

2 Review of Hashing Algorithms and Models

The word hashing usually refers to the process of cutting and chopping something until it loses its original shape. In Computer Science, hashing is done by applying a hash function that maps a huge (but finite) universe of all possible keys into a smaller range. In this mapping, a given set of keys will usually loose a lot of its "structure". This, together with the fact that the hash function is in many times a "grinding" of the key representation, explain the usage of the term hashing.

Hashing is traditionally used for memory-efficient implementation of sparse tables and dictionaries; a hash function reduces the universe size to a suitably small range, which is sometimes called the *hash table*, and then manipulation of keys (insert, lookup and delete) can be done in direct access.

Hashing algorithms can be classified by the hash function they use and by the way they deal with collisions, i.e., two distinct keys having the same hashed value. Knuth [36] enumerates several "random-like" hash functions. Random functions were intuitively perceived as the best for hashing [37]. This intuition was proved correct by Ajtai, Komlós and Szemerédi [3].

Random functions, the usage of universal hashing for achieving random-like behaviour and the FKS scheme are discussed in the following section. Except for the FKS scheme there are two main techniques for dealing with collisions: chaining and open-addressing. Let us review these techniques and the models used in the literature for analyzing them.

Chaining Perhaps the most popular collision resolution tool used in practice is *chaining*. The hash table in this case is an array of linear lists, where all keys with the hashed value i are stored

in the ith list. The simplicity and the extendibility of this algorithm makes it useful in diverse applications such as compilers [2, pages 340–341] operating systems [5] and text formatters [38, pages 107-111]. Analysis of the performance of chaining can be found in [43, pages 120–124]. This analysis assumes that the hash function distributes the universe uniformly and that all its elements are equally likely to appear in any place of the input. Alternately, it may be assumed that the hash function is random, i.e., that it is selected at random from all possible functions that map the universe to the table. This random function assumption is used by Knuth [36, pages 514–517] to analyze a certain variant of a technique called *coalesced chaining* in which lists are not completely separate. The advantage of coalesced chaining is that it does not use an external dynamic memory allocation procedure for lists management; all lists records are allocated from hash table entries.

Another variant of coalesced chaining was suggested and analyzed by Knott [33] and independently by Vitter [56]; further analysis of this variant can be found in [11, 12, 46, 47]. Random function are assumed in all of these studies. Chen and Vitter [13] considered the problem of preserving the "random" order of elements in coalesced chains after deletions, and suggested a battery of algorithms for this problem. Not surprisingly, the assumption underlying their research was again that the hash function itself spreads the elements uniformly at random among the lists, any bias is due to list administration.

Other chaining variants were proposed and analyzed as well. (See e.g., [35].)

Open addressing Another popular method for collision handling is within the hash table itself, using *open addressing*. In this method, instead of lists, we define for each key a sequence of table entries, the first of which is the key's hashed value. Key access is done by probing this entries one by one, until the key is found. For keys which are not in the table this probes will end in an empty entry. The probe sequence can be defined in many ways, the two most popular ones being *linear probing* and *double hashing*.

In linear probing, the probe sequence is just a linear scan of the table starting at the position equal to the key's hashed value and wrapping around the table's end. Analysis of linear probing under the assumptions of keys selected at random, or equivalently that the hashing function used is random can be found in [36].

Double hashing is similar to linear probing except that instead of scanning always in increments of 1, other step sizes (which are relatively prime to the table size) are used. The step size is determined by a secondary hash function of the key; hence the name double hashing. Text book (e.g., [1, pages 130-131]) analysis of double hashing is done using a model of *uniform Hashing*, i.e., that the probe sequence of a key is a random permutation.

Ullman [55] suggested a model for studying the the performance of closed hashing schemes. In this model he showed that that there is no closed hashing scheme which in terms of the expected insertion cost will perform better than uniform hashing. Yao [59] complemented this result by showing that the optimality of uniform hashing is true also in terms of the expected retrieval cost.

Guibas and Szemerédi [29] showed that if the primary and the secondary hash functions are random then the performance of double hashing is equivalent (asymptotically for large tables) to that of uniform hashing up to a load factor of about 0.319. Lueker and Molodowitch [40] showed that this equivalence holds for load factors arbitrarily close t

Some other studies of open addressing and of the uniform hashing model include the works of Brent [7], Rivest [50], Gonnet and Munro [27], Gonnet [26], Larson [39], Cellis, Larson and

Munro [9], and Poblete and Munro [48]. Ramakrishna [49] gives an analysis of open addressing hashing under the *random probing* model in which the probes are independent random positions in the hashing table. Linear open addressing was considered by Mendelson and Yechiali [44], Pflug and Kessler [45] (assuming non-uniform key distribution), Pittel [46] and Knott [34].

Perfect hashing Some of the research on hashing was devoted for the the search for perfect hash functions. A hash function is *perfect* for a given set of keys if it is in injective on the set. The function is *minimal perfect* if in addition the size of its range is equal to the set size. The function is *ordered minimal perfect* if in addition it also sorts the input keys. The problem of finding a minimal and almost minimal perfect hash functions for a given set of keys has received much attention. The main parameter studied was the number of bits required for the representation of such function. Note that the corresponding hashing model concentrates only on space complexity. Numerous algorithms have been suggested for constructing minimal, almost minimal, and ordered minimal hash functions (See for example [6, 10, 28, 57].) Some of this algorithms aim at achieving a good hash function construction time and/or evaluation time. Schmidt and Siegel [51] give a short account of the best upper and lower bounds results for this problem.

3 Preliminaries

Definitions Let S be a set of n keys drawn from a finite universe $U = \{0, 1, \ldots, q-1\}$. A *hash function* h is a function mapping the universe to a bounded range, $U \stackrel{h}{\mapsto} [0, \ldots, m-1]$. Such a function partitions S to subsets, also called *buckets*, $S_i^h = h^{-1}(i) \cap S$, of sizes $b_i^h = \left|S_i^h\right|$, $0 \leq i < m$. For convenience, the h superscript will be omitted when it is clear from the context. We call h a *perfect hash function* for S if there are no two keys in S that are mapped by h to the same value. We say that h is *r-perfect for S* if $b_i^h \leq r$ for all $0 \leq i < m$; h is *perfect* if it is 1-perfect.

Perfect hash functions are of considerable importance since they induce a linear storage scheme for S that supports lookup operations (queries of the type "is $x \in S$") at the cost of one function application. It is also important to find relatively small classes of hash functions that contain a perfect hash function for any given input set of a certain size. If the hash class is very large then the representation of a single function from the class cannot be efficient. In the extreme case, there is always a perfect hash function among all m^n possible functions (provided, of course, that $m \geq n$). However, writing down such a function can generally be done only by tabulating all values $x \in U$. This lengthy representation is inadequate for another reason: to evaluate the function value at certain point a table lookup is required, which was the original problem the perfect hash function was introduced to solve.

The following definition encapsulates the most-desired properties of hash functions:

Definition 3.1 *Let h be a hash function. Then h is called a* good lookup function for S *if*

(i) h is perfect for S;
(ii) h uses linear storage (i.e., $m = O(n)$);
(iii) h can be represented in $O(n)$ space;
(iv) h supports quick lookups, i.e., for every $x \in U$, $h(x)$ can be evaluated in $O(1)$ time by a single processor.

The data structure induced by a good lookup function is called a good linear static hash table. *This data structure requires $O(n)$ storage and $O(1)$ time for a lookup query.*

3.1 The FKS Scheme

In their seminal paper [20], Fredman, Komlós and Szemerédi introduced a sequential scheme that generates a good linear static hash table in $O(n)$ expected time for any input set. Their scheme builds a *2-level* hash function: a *level-1* function splits S into subsets whose sizes are distributed in a favorable way. Then, a perfect *level-2* hash function is built for each subset. Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert and Tarjan [16] extended the *2-level* scheme for a dynamically changing input set (a dictionary), in which the amortized cost for insert and delete is constant. Dietzfelbinger and Meyer auf der Heide [19] presented a scheme for implementing a dictionary in worst case constant time per instruction with very high probability. Parallel and network adaptations of the FKS scheme were also done, see [17, 18, 41].

Polynomial Hash Functions The class of d-degree polynomial hash functions is defined by

$$\mathcal{H}_m^d := \left\{ h \;\middle|\; h(x) := \left(\sum_{i=0}^{d} a_i x^i \bmod q\right) \bmod m,\ a_i \in U \right\} .$$

If q, the universe size, is prime, then polynomial hash functions are useful in construction of good lookup functions.[1] Henceforth we assume that q is prime.[2] Polynomial classes are *not* rich enough to contain a good lookup function for every input set. However, as Fredman, Komlós and Szemerédi [20] demonstrated, it is possible to use combinations of linear (i.e., polynomial of degree 1) hash functions to construct good lookup functions.

The main properties of the linear hash functions which enable their construction are given in the following fact.

Fact 3.1 (FKS) *If the input set S is fixed then*

(a) There exists at least one function $h \in \mathcal{H}^1_{|S|}$ such that $\sum_{i=0}^{m-1} |S_i|^2 < 3|S|$.

(b) There exists at least one function $h \in \mathcal{H}^1_{|S|^2}$ which is perfect for S.

Remark A selection of a random function in the class $\mathcal{H}^1$ provides pair-wise element independence, i.e., for any $x_1, x_2 \in U$, $0 \le i_1, i_2 < m$ and h selected at random from $\mathcal{H}^1$

$$\textbf{Prob}\left(h(x_1) = i_1 \wedge h(x_2) = i_2\right) .$$

This attribute underlies the above fact as well as some other properties of $\mathcal{H}^1$ mentioned later.

[1] Applicability of such functions for hashing purposes was first demonstrated by Carter and Wegman [8].

[2] This assumption is not restrictive since there is no need for a large increase in the universe size before this property is attained.

The FKS construction For a given input set S, a *level-1* hash function is selected using Fact 3.1 *(a)*. This function splits S into buckets in a such way that each bucket can be allocated memory quadratic in its size, while keeping the total memory requirements limited to $O(n)$. Fact 3.1 *(b)* is then applied to find a *level-2* perfect hash function for each bucket. The good lookup function for the whole input set is the composition of the appropriate level-2 function and the level-1 function. Representation of the function is carried out by allocating an auxiliary array whose entries correspond to the buckets. Each such entry stores the level-2 hash function associated with a bucket as well as pointer to its memory block.

The most prominent feature of the work of FKS is that their construction works for *any* given set and for *any* universe size. This stands in clear contrast to many other hashing schemes which could guarantee and analyze performance only under the assumption that the hashed keys are chosen uniformly at random from the universe. The assumption that S is fixed is henceforth implied.

FKS continued their investigation noting that although the resulting set representation is efficient, the construction is costly and it may take $O(nq)$ operations. An improvement in the efficiency of the construction algorithm (with the cost of a constant factor increase in memory) was given by using the following facts:

Fact 3.2 (FKS) *If h is chosen uniformly at random from the class $\mathcal{H}^1_{|S|}$ then*

$$\mathbf{E}\Big(\sum_{i=0}^{m-1} |S_i|^2\Big) \;\leq\; 2.5|S| \;,$$

consequently (by Markov's inequality)

$$\mathbf{Prob}\Big(\sum_{i=0}^{m-1} |S_i|^2 \leq 5|S|\Big) \;\geq\; 1/2 \;.$$

Fact 3.3 *If h is chosen uniformly at random from the class $\mathcal{H}^1_m$*

$$\mathbf{Prob}\,(h \textit{ is not perfect for } S) \;\leq\; |S|^2/m \;.$$

The last fact does not appear exactly as presented in [20] but it is easily derivable from their basic lemma.

An Efficient FKS Construction From Fact 3.2 it follows that a level-1 function, with properties similar to those of the level-1 function in the previous construction, can be selected for any given input set in $O(n)$ *expected* time. A bucket S_i is then allocated $2|S_i|^2$ memory cells. Using Fact 3.3, we infer that the probability of a failure of such a level-2 hash function on a fixed bucket is less than $1/2$. The processing time for a function applied to a bucket S_i is $O(|S_i|)$, therefore the expected time before a perfect hash function is found for the bucket is also $O(|S_i|)$. The total expected time for finding a good lookup function for all the buckets is

$$\sum_{i=0}^{m-1} |S_i| = |S| = n \;.$$

Note that if this hashing scheme was performed in rounds, then in every round the number of "active" buckets will drop "only" by a factor of 2 so there will be $\Omega(\lg n)$ rounds. It can be shown that if the memory allocated to a bucket is the same in each round, and if the total memory is linear in the size of the input, then the total number of rounds is $\Omega(\lg n)$.

In our hashing schemes we will be using functions from $\mathcal{H}^1$ (with various ranges) as level-2 functions, and we will be interested in the analysis of the distribution of the number of buckets that fail.

The 2-level hash scheme of FKS and the strength of polynomial hash functions provided ground for further developments in hashing theory. The usability of higher degree polynomials as hash functions has been demonstrated, and it has been observed that such d-degree polynomials grant $(d+1)$-independence, as well as the following properties which are generalizations of similar properties found in linear hash functions:

Fact 3.4 (DKM88) *Let the input set S be fixed, and h be picked at random from $\mathcal{H}_m^d$, where $d \geq 1$, $m \geq n$. Then*

(a) $\mathbf{E}\left(\sum_{i=0}^{m-1}\prod_{j=0}^{d}(|S_i| - j)\right) \leq 2n\left(\frac{2n}{m}\right)^d$,

(b) $\mathbf{Prob}\left(\sum_{i=0}^{m-1} b_i^d \leq c \cdot n\right) \geq \frac{1}{2}$, *for some constant $c > 0$, and*

(c) $\mathbf{Prob}$ (h is not d-perfect for S) $\leq 1 - n\left(\frac{2n}{m}\right)^d$.

Proof. See [16] and [19]. ∎

Corollary 3.1 *Let the input set S be fixed, and let h be picked at random from $\mathcal{H}_m^d$, where $d \geq 1$, and $m \geq n$. Then there is a constant $c > 0$ such that*

$$\mathbf{Prob}\left(\sum_{i=0}^{m-1} |S_i|^d \leq c \cdot n\right) \geq \frac{1}{2}\ .$$

Parallel and network adaptations of the FKS scheme were also done. (See [17, 18].)

3.2 Chernoff Bounds

One of the advantages of the 2-level scheme is that the hashing of different buckets is completely independent, hence exponential Chernoff bounds [14, 4, 30] may be used in the analysis of the probabilistic behavior of the hashing of a batch of buckets.

Fact 3.5 (Chernoff) *Let $x_1, \ldots, x_n$ be independent binary random variables. Let $X = \sum_{i=1}^n x_i$, and let $m = \mathbf{E}(X)$. Then for $0 \leq \epsilon \leq 1$ and $\gamma > 1$*

(a) $\mathbf{Prob}(X \geq (1+\epsilon)m) \leq e^{-\epsilon^2 m/3}$,
(b) $\mathbf{Prob}(X \leq (1-\epsilon)m) \leq e^{-\epsilon^2 m/2}$, *and*
(c) $\mathbf{Prob}(X \geq \gamma m) \leq (\gamma/e)^{-\gamma m}$.

Proof. See [30]. ∎

3.3 Pseudo Random Classes

Recently, Dietzfelbinger and Meyer auf der Heide [19] showed how polynomial hash functions can be combined to create a new class of hash functions such that a function chosen uniformly at random from that class, looks very much like a truly random function.

Definition 3.2 *The class $\mathcal{R}^{d_1,d_2}(k,m)$ of hash functions is the set of all $(k+2)$-tuples $h = \langle f, g, a_1, a_2, \ldots, a_k \rangle$, where $f \in \mathcal{H}_k^{d_1}$ and $g \in \mathcal{H}_m^{d_2}$, and $a_1, a_2, \ldots, a_k \in \{0, \ldots, m-1\}$. The action of $h \in \mathcal{R}^{d_1,d_2}(k,m)$ on $x \in U$ is defined as $h(x) := (g(x) + a_{f(x)}(x)) \bmod m$.*

The following fact summarizes those aspects of their results which we require:

Fact 3.6 ([19]) *There is a class $\mathcal{R} = \mathcal{R}^{d_1,d_2}(k,m)$ of hash functions and a subclass $\mathcal{R}(S) \subseteq \mathcal{R}$ of functions to the range $[0, \ldots, n-1]$ that satisfy the following properties:*

(a) $h \in \mathcal{R}$ can be evaluated in constant time.
(b) n processors can pick a random $h \in \mathcal{R}$ in $O(1)$ time.
(c) If h is picked at random from $\mathcal{R}$ then $\mathbf{Prob}(h \notin \mathcal{R}(S)) < n^{-k}$ for any $k > 0$.
(d) For h randomly chosen from $\mathcal{R}(S)$,

- *h is $\lg n$-perfect with probability $1 - n^{-k}$ for any $k > 0$.*
- *$\mathbf{Prob}(b_i^h \leq r) \geq (e^{r-1}/r^r)^{1/d_1}$.*

4 Our Model

4.1 The Basic Model

The process of inserting a set of n elements taken from some universe U into a hash table can be thought of as a process of refining partitions, and is depicted simply by a tree. Originally, all elements reside in a single node (the root). The algorithm chooses a range m, picks a hash function $h : U \mapsto [m]$ according to some distribution, and partitions S into subsets $S_1, S_2, \ldots, S_m$ (some empty) such that $S_i = h^{-1}(i) \cap S$. The function h is stored at the root, the root has m children and the subsets S_i move to different children of the root. The function h is stored at the root, the root has m children and the subsets S_i move to different children of the root.

The process repeats for each S_i that contains at least two elements. The refining halts when every leaf contains at most one element from S.

A search for an element $x \in U$ in the hash table is easily performed by following the path from the root which is determined by applying the hash functions at internal nodes to the element x, and when reaching a leaf, comparing x to the element residing there if such an element exists.

In particular, our approach leads to an alternate view of a hashing algorithm as an element distinctness proof generator. The input is a set of distinct keys taken from a universe with no order relation defined on it. The output is a proof that all elements are distinct. The proof components are functions from the universe to a bounded range.

4.2 Comments

1. The two types of strategic decisions made by a specific algorithm are the choice of range (i.e., number of children) for the hash function, and the choice of distribution of functions to this range. We *assume* that the function used is a truly random function, i.e., all possible functions are given equal probability. This assumption follows that of many of the previous models for analyzing hashing algorithms, and the general trend in the hashing algorithms design of trying to construct pseudo-random functions. (See also discussion in Section 9.)

   Thus, the hashing process that occurs in a node can be described as the act of independently sending each element of U to each possible child with uniform distribution. Analyzing the full hashing algorithm is reduced to analyzing a natural process of successively throwing identical balls into boxes until all the balls reside in distinct boxes.

2. Most common algorithms are stronger than the process we described—they use retries when the chosen hash function is extremely bad (e.g., all elements were mapped to one cell), and allow storing elements in the internal nodes of the tree as well as in the leaves. These generalizations and others will be considered later by introducing variants to the basic model.

3. We deliberately deal here with the static case, i.e., when all the elements to be inserted are known in advance. This does not restrict the generality of the lower bounds. However, the algorithms presented in this chapter are for the static case only. The question of existence of *sequential* algorithms (which fall into the framework of our basic model or its variants) for the dynamic case that achieve both constant amortized time and minimal worst time are not dealt with here. PRAM algorithms for the static case are described in [22]; the results presented there were the basis of the extremely fast dictionary algorithm of Gil, Matias and Vishkin [24].

4. Yao's cell probe model [58], the standard general model for hashing, can also be described as a tree in a similar way. Our model differs from his in the way that a decision tree differs from a Turing machine. The cell probe model allows each cell a limited number of bits (depending on U), but these can encode arbitrary objects and be computed at no cost. Our cells contain either elements or functions. Functions can only be applied to elements and two elements can only be tested for equality. Our model, being more structured, is cleaner and easier to analyze, though less general.

4.3 Resources

The stochastic process determined by a hashing algorithm Alg given $S \subset U$ of size n, is described by a random tree. The main resources of Alg operating on S are natural parameters of this tree.

Space The space required, or hash table size, denoted by $\textsc{Space}(\mathsf{Alg}, S)$, is simply the total number of nodes in the tree.

Insertion Time We denote by $\textsc{Time}(\mathsf{Alg}, S)$ the total insertion time. This is the sum of depths of all leaves containing an element, i.e., the number of hash function applications to all the elements. Each application counts as one time unit.[3]

[3]If hash functions are taken from $\mathcal{H}^d$ or from $\mathcal{R}^{d_1,d_2}$ for constant d, d_1, d_2 then this is the case for actual algorithms as well.

Parallel Insertion Time We denote the depth of the tree by $\textsc{Depth}(\mathsf{Alg}, S)$. This is the parallel insertion time under the assumption that each processor is assigned one key and this processor alone is responsible for inserting this key. This parameter has two important meanings for sequential algorithms as well. It captures the number of functions needed to resolve the "worst" pair of elements, and the worst case insert and search time.

Search time will not be identical to insertion time in the more general models, so we devote a different notation for it:

Maximum Search Time This is the largest number of function applications needed to find out if $x \in U$ is in S using the tree generated by Alg on S. This parameter will be denoted by $\textsc{Search}(\mathsf{Alg}, S)$. In the basic model this definition coincides with that of $\textsc{Depth}(\mathsf{Alg}, S)$. However, this will no longer be true when the model model is elaborated, although we will still have $\textsc{Search}(\mathsf{Alg}, S) \leq \textsc{Depth}(\mathsf{Alg}, S)$.

Let PARAM be a generic parameter (SPACE, TIME, DEPTH or SEARCH), then $\overline{\textsc{Param}}(\mathsf{Alg}, S)$ will denote the expectation of PARAM with respect to the random choices made by the algorithm Alg, so $\overline{\textsc{Param}}(\mathsf{Alg}, S) = \mathbf{E}(\textsc{Param}(\mathsf{Alg}, S))$. We denote by

$$\overline{\textsc{Param}}(\mathsf{Alg}, n) = \max_{|S|=n} \overline{\textsc{Param}}(\mathsf{Alg}, S)$$

the performance of Alg on a worst case set S of size n, and by

$$\overline{\textsc{Param}}(n) = \min_{\mathsf{Alg}} \overline{\textsc{Param}}(\mathsf{Alg}, n)$$

the performance of the best algorithm on its worst case set S. At times it will be useful to ignore the probabilistic performance, and consider the worst possible performance of Alg (over all possible runs) on the worst case input, which we denote by $\textsc{Param}(\mathsf{Alg}, n)$.

4.4 Variants of the Basic Model

Finally, we consider more powerful algorithms than those permitted by the basic model:

Retries An algorithm may allocate (say) m boxes (children) for n balls residing at a node v, and find that in throwing them randomly they all fell into one or very few cells. This is an unlikely event, that causes a big waste of space. The algorithm is allowed to consider this (or other more likely events) "bad", and try again. We do not charge for space used in v. To maintain the meaning of depth in this variant, we create one single child for v, and move all the balls there.

We attach the subscript r to the resources measures in this model, e.g., $\textsc{Space}_r(\mathsf{Alg}, S)$ and $\overline{\textsc{Depth}_r}(n)$ etc. Note that $\textsc{Search}_r$ may be much smaller than $\textsc{Depth}_r$, since while a search is being performed no function application should be done at a node with only one child.

Chaining This variant allows the algorithm to store elements in internal nodes too. Specifically, when m keys reach a node v, only $m-1$ proceed to v's children, and one of them is stored in v. The term chaining is used since this variant generalizes hashing techniques in which a chain of keys can be stored in hashing array positions.

The subscript c is added to the resources measures. Clearly SEARCH_c and DEPTH_c are the same again, since even if no branching occurs at node v, the element being searched for should be compared to the one residing at v.

We allow a combination of chaining and retries, which is denoted by the double subscript cr. In this combination the algorithm may leave a key in an internal node, even if the function used at this node was discarded. Obviously, such a node cannot be skipped in a search.

Parallel Hashing In this variant of the model, we allow the algorithm to try in parallel several hash functions in a node v, and then pick one of them to create v's children. Space here is counted as the sum of ranges of *all* those functions. Subscript p is added in this variant to the resources measures.

This variant may be combined with the two previous ones; if retries are permitted then the algorithm may choose not to use any of the hash functions that were tried; if exploiting internal nodes is possible then the algorithm can leave one element at v, no matter which hash function is selected for the node.

Despite its name, this variant does not lead directly to a PRAM algorithm. The major difficulty causing this is the assignment of processors that have completed handling their original key, to assist the yet unhashed keys.

One possible hashing variant was deliberately omitted from the above list; we do not permit the merging of nodes in the hash trees. Intuitively, merges *lose* separation information, and omitting merges from a hashing algorithm should only improve its performance. It is easy to verify that the TIME, DEPTH and SEARCH complexity measures can only decrease as a result of eliminating merge operations. The only possible merit of merging is to SPACE. It will be evident from the lower bounds proofs that merging cannot improve an algorithm with respect to all complexity measures defined above. [4]

Most hashing algorithms deviate from our basic model by allowing one or both of the retries or the chaining variants. The parallel variant is mentioned as an alternate hashing idea in [22], and used by Matias and Vishkin [42].

5 Results

The most interesting algorithms are those that achieve $\text{SPACE}(\mathsf{Alg}, n) = O(n)$ i.e., linear space. In his seminal paper "Should Tables be Sorted?" [58], Yao asked if one can simultaneously achieve $\text{SPACE}(n) = O(n)$ and $\overline{\text{SEARCH}}(n) = O(1)$. In our *basic* model, this is impossible.

Theorem 5.1 *If* $\text{SPACE}(\mathsf{Alg}, n) = O(n)$ *then* $\overline{\text{DEPTH}}(\mathsf{Alg}, n) = \overline{\text{SEARCH}}(\mathsf{Alg}, n) = \Omega(\lg \lg n)$.

However, allowing retries, Yao gave an algorithm Y which achieves $\text{SPACE}_r(\mathsf{Y}, n) = O(n)$ and $\text{SEARCH}_r(\mathsf{Y}, n) = O(1)$ for large enough universes. For small universes, $q = n^{O(1)}$, Tarjan and Yao [54] showed how linear storage and constant search time can be maintained. Fredman, Komlós and Szemerédi [20] closed the gap by an algorithm FKS that satisfies $\text{SPACE}_r(\mathsf{FKS}, n) = O(n)$ and $\text{SEARCH}_r(\mathsf{FKS}, n) = O(1)$ for *any* universe size, and any input set. Analyzing their algorithm,

[4] However, it is interesting to note that merging is useful for the construction of good pseudo-random functions, which may be used for implementing hashing algorithms [19].

we find that while insertion time $\overline{\text{TIME}_r}(\text{FKS}, n) = O(n)$, (i.e., on the average we apply only a constant number of functions to each element), $\overline{\text{DEPTH}_r}(\text{FKS}, n) = \Omega(\lg n)$, so some element will be hashed $\Omega(\lg n)$ times, and this is the time required for hashing the elements in parallel using the FKS scheme.[5] A natural question that arises is whether this parameter can decrease to $O(1)$? We answer this question in the following theorem:

Theorem 5.2 *If* $\text{SPACE}(\text{Alg}, n) = O(n)$ *then* $\overline{\text{DEPTH}_r}(n) = \Omega(\lg \lg n)$.

Remark There exists an algorithm, DM, due to Dietzfelbinger and Meyer auf der Heide [19], for managing a dynamic data hash table, that achieves with very high probability constant worse case performance. However, DM does not contradict the stated lower bounds since it does not fit in our basic model nor in any of its variants. In particular, DM pipelines the insertions; processing an inserted element can continue for up to n^ϵ steps after the insertion took place; the algorithm allows keys to be fetched even if they are not "fully" inserted. Still, as the lower bounds indicate, there is no easy way of constructing a fast parallel version of DM. There are always keys for which DM requires as many as n^ϵ function applications.

With the help of retries, the lower bound of Theorem 5.1 can be met:

Theorem 5.3 *There is an algorithm* Shallow *which uses linear space (i.e.,* $\text{SPACE}_r(\text{Shallow}, n) = O(n)$*), that gives*

(i) $\overline{\text{TIME}_r}(\text{Shallow}, n) = O(n)$,
(ii) $\overline{\text{DEPTH}_r}(\text{Shallow}, n) = O(\lg \lg n)$, *and*
(iii) $\text{SEARCH}_r(\text{Shallow}, n) = O(1)$.

This algorithm is a variant of the FKS algorithm. The improvement in $\overline{\text{DEPTH}}(n)$ is accomplished by using a different, more adaptive memory allocation scheme while executing the retries, so that the probability of failure decreases very quickly. This algorithm is optimal with respect to all parameters, even if we count arithmetic operations and limit word size to $O(\lg U)$. Moreover, if we restrict the algorithm to the basic model by eliminating retries, all the parameters, except for SEARCH(Alg) (which will be the same as DEPTH(Alg)) will remain optimal:

Theorem 5.4 *There is an algorithm* Shallow' *for which*

(i) $\overline{\text{SPACE}}(\text{Shallow}', n) = O(n)$,
(ii) $\overline{\text{TIME}}(\text{Shallow}', n) = O(n)$, *and*
(iii) $\overline{\text{DEPTH}}(\text{Shallow}', n) = \overline{\text{SEARCH}}(\text{Shallow}', n) = O(\lg \lg n)$.

A worst case upper bound for SPACE is not possible here, because for any such bound there are (admittedly rare) cases, in which enough failures occur to force an algorithm to overflow this bound.

The general tradeoff between space and depth is given by

Theorem 5.5 *If* $\text{SPACE}(\text{Alg}, n) = n^{1+1/\lambda}$ *then* $\overline{\text{DEPTH}_r}(\text{Alg}, n) = \Omega(\lg \lambda)$.

[5] Indeed the parallel hashing scheme of Matias and Vishkin [41] being based on FKS takes $O(\lg n)$ parallel time.

Can the common practice of using internal nodes for storage help by more than a constant factor? Again, perhaps surprisingly, the answer is positive:

Theorem 5.6 *Both* $\text{SPACE}_c(n) = O(n)$ *and* $\overline{\text{DEPTH}_c}(n) = O(\lg\lg n/\lg\lg\lg n)$ *can be achieved simultaneously.*

The algorithm behind this theorem uses truly random hash functions or equivalently, high degree polynomials. As a more practical alternative, the class $\mathcal{R}$ can be used here too. The next theorem shows that this meager improvement of $\lg\lg\lg n$ is the best possible, and even it cannot coexist with employment of retries to achieve $O(1)$ search time. (As before, adding the power of retries to this variant of the model cannot improve $\overline{\text{DEPTH}}(n)$.)

Theorem 5.7

(a) If $\text{SPACE}_{cr}(\mathsf{Alg}', n) = \text{SPACE}_c(\mathsf{Alg}, n)$ *then*

$$\overline{\text{DEPTH}_{cr}}(\mathsf{Alg}', n) = \Omega\left(\overline{\text{DEPTH}_c}\right)(\mathsf{Alg}, n) \ .$$

(b) If $\text{SPACE}_c(\mathsf{Alg}, n) = O(n)$ *then*

$$\Omega\left(\overline{\text{DEPTH}_c}(\mathsf{Alg}, n)\right) = \Omega\left(\overline{\text{SEARCH}_c}(\mathsf{Alg}, n)\right) = \Omega(\lg\lg n/\lg\lg\lg n) \ .$$

(c) If $\text{SPACE}_{cr}(\mathsf{Alg}', n) = \text{SPACE}_c(\mathsf{Alg}, n) = O(n)$ *and* $\overline{\text{DEPTH}_{cr}}(\mathsf{Alg}, n) = O(\lg\lg n/\lg\lg\lg n)$ *then*

$$\overline{\text{SEARCH}_{cr}}(\mathsf{Alg}', n) = \Omega\left(\overline{\text{DEPTH}_c}(\mathsf{Alg}, n)\right) \ .$$

The general tradeoff is given by:

Theorem 5.8 *If* $\text{SPACE}_c(\mathsf{Alg}, n) = n^{1+1/\lambda}$ *then* $\overline{\text{DEPTH}_c}(\mathsf{Alg}, n) = \Omega(\lg\lambda/\lg\lg\lambda)$.

In a clear contrast to the first two variants, the "Simultaneous Retries" which may be applied in the parallel variant, lead to a significant improvement in DEPTH because they allow folding many iterations into one. Nevertheless, constant time hashing time cannot be achieved in this case too.

Theorem 5.9 *If* $\text{SPACE}_p(\mathsf{Alg}, n) = O(n)$ *then* $\overline{\text{DEPTH}_p}(\mathsf{Alg}, n) = \Theta(\lg^* n)$.

Neither retries nor chaining can further decrease the maximal insertion time of the parallel variant.

Theorem 5.10 *If memory usage is restricted to* $O(n)$ *then*

$$\overline{\text{DEPTH}_{rcp}}(n) = \Omega\left(\overline{\text{DEPTH}_p}(n)\right) \ .$$

6 Proofs of Lower Bounds

We view hashing algorithms from a parallel perspective. Each parallel iteration is an attempt to separate *all* subsets of S that were not previously separated, i.e., subsets that still have two or more keys in them. Thus successive iterations correspond to successive tree levels.

Let Opt be the best possible algorithm for the current setting of the parameters (space and model variant). Our proofs are based upon showing that with a dominant probability, there is a minimal number of iterations Opt has to go through. For simplicity in the analysis, we let Opt make the following assumptions:

Extra Memory Say that the problem restricts the memory usage to a total of m memory cells. This restriction will be weakened for Opt and it will be allowed to use m memory cells in *each* iteration.

Partial Separation A mapping of a set of keys to memory is called a *partial separation* if there exist two keys in the set that are mapped to distinct cells. Opt may consider any partial separation as being a total separation. The extremely unlikely case in which all keys from the set were mapped to the same memory is called a *complete failure*. Only complete failures need to be passed to the next iteration of Opt.

Restricted Set Size In iteration t, Opt has only to deal with (nodes containing) sets of r_t keys. Smaller or larger sets can be completely ignored. The exact value of r_t will be specified later.

Early Termination need not be concerned with the case where there are fewer than $\lg n$ sets of size r_t. As soon as the number of sets drops below that bound, Opt can terminate immediately.

Higher Success Probability While analyzing Opt we will assume that failure probability is determined by $r = r_1$ although r_t keys are actually mapped. It will be shown that this may only decrease the failure probability.

To account for the random nature of the hashing process, the following definition is introduced.

Definition 6.1 *Events that occur with probability smaller than $n^{-\epsilon}$ for some $\epsilon > 0$ are called* negligible *events.* Dominating *events are the complement of negligible events.*

Negligible events will be ignored in the following discussion, since even if they could be treated by Opt without *any* resource investment, the expected value of the performance measures will essentially be the same.

The rest of this section is outlined as follows. Suitable values for r_t will be set, and then we will compute a lower bound on the initial number of sets of size r_1. (Since the algorithm is based on a random process, it may be extremely lucky and break this bound; thus the lower bound statement should be read with "ignoring negligible events" appended to it. Such quantification is implied henceforth.) We next estimate the number of sets of size r_{t+1} in iteration $t+1$ as a function of the number of sets of size r_t in iteration t. Then an *explicit* lower bound for the number of sets of size r_t in iteration t is derived. The lower bound proofs are then completed by computing the minimal number of iterations Opt must undergo before completion. The analysis is done for the basic model and the chaining variant together, and then it is repeated for the parallel variant. We conclude with a remark explaining why all lower bound proofs are applicable to all the retries variants.

6.1 Root Node Hashing

The root node corresponds to iteration number 0. In it n keys are hashed into m memory cells. The set S is separated using a random function $h : U \mapsto [m]$ into subsets $S_1, S_2, \ldots, S_m$. Since h is a random function, the root node is accurately modeled by the the well studied "balls into boxes" or "urn" model. Let $N = N(r)$ be the number of subsets that have exactly r elements in them and let $\eta = \eta(r) = N(r)/m$. The values of r that we will be using are such that $r > 1$ and $r = o(n)$, so the results obtained there can be used as follows:

- For large enough n, $N(r)$ has Poison distribution. We get that there exists n' such that for every $n > n'$

$$\mathbf{E}(N(r)) > \frac{1}{2} m e^{-\alpha} \frac{\alpha^r}{r!} \tag{1}$$

  where $\alpha = n/m$. Without loss of generality assume that $n > n'$ from now on.

- If $\mathbf{E}(N(r)) = \Omega(n^\epsilon)$ then the event $N(r) < \mathbf{E}(N(r))/2$ is negligible. (See [21, Corollary 2.3] for proof.) Consequently, if $\mathbf{E}(N(r)) = \Omega(n^\epsilon)$

$$N(r) \geq \frac{1}{4} m e^{-\alpha} \frac{\alpha^r}{r!} \ , \tag{2}$$

  except in a negligible number of cases.

6.2 The Basic Model and the Chaining Variant

In the basic model, we follow only sets of size 2, i.e., $r_t = 2$ for $t \geq 1$. When the usage of intermediate nodes (chaining) is possible, sets of fixed size r can no longer be tracked since the number of iterations will depend on n, and even complete failure to hash a set will decrease its size by 1. Instead define $r_0 = r = r(n)$ and $r_{t+1} = r_t - 1$. Note that, in this variant, r_0 must be greater than the desired lower bound for the number of iterations. The following fact estimates $\mathbf{E}(N(r))$ for those pairs of r and m we are interested in. Note that in all of those cases $\mathbf{E}(N(r))$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$ and hence the event $N(r) < \mathbf{E}(N(r))/2$ is negligible.

Fact 6.1 *The expected value of $N_0(r)$, the initial (after the root node hashing) number of sets of size r, is given by:*

1. *If $r = 2$ and $m = O(n)$ then*

$$\mathbf{E}(N_0(r)) = \Omega(n) \ .$$

2. *If $r = 2$ and $m = n^{1+1/\lambda}$ for some fixed λ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-1/\lambda-1/n^{1/\lambda}\ln n}\right) = \Omega\left(n^{1-1/\lambda-o(1)}\right) \ .$$

3. *If $r = \lg\lg n/\lg\lg\lg n$ and $m = O(n)$ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-r(\lg r-\lg\alpha)/\lg n)}\right) = \Omega\left(n^{1-o(1/\lg n)}\right) \ .$$

4. *If $r = \lg\lambda/\lg\lg\lambda$ and $m = n^{1+1/\lambda}$ for some fixed λ then*

$$\mathbf{E}(N_0(r)) = \Omega\left(n^{1-(r-1)/\lambda-o(1)}\right) \ ,$$

Proof. Apply Inequality (1). ∎

From the simplifying assumptions it follows that in iteration t, Opt uses m memory cells to deal with N_t sets of r_t keys each. The algorithm should allocate memory to cells in a way that will minimize the number of failures N_{t+1}. The following lemma reveals the memory allocation scheme used by Opt.

Lemma 6.1 Opt *uses a balanced memory allocation scheme; each of the N_t sets is hashed into m/N_t cells.*

Proof. In an iteration t, if a subset (that has r_t keys) is mapped by a random function into m_i memory cells, then the *complete failure* probability is $m_i^{1-r_t}$. This probability is inversely proportional to m_i, therefore a memory allocation is not optimal unless all the m cells are utilized. Let $m = m_1 + m_2 + \cdots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is given by

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} m_i^{1-r_t}$$

and by convexity this is minimized when all m_i are equal. ∎

The complete failure probability, $m_i^{1-r_t}$, increases as r_t decrease. Thus it is permissible to assume that Opt uses a complete failure probability derived from $r = r_1$, the initial size of the sets. We can then write

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} m_i^{1-r}$$

and by Lemma 6.1

$$\mathbf{E}(N_{t+1}) = N_t\left(\frac{m}{N_t}\right)^{1-r} .$$

The probability that N_{t+1} will be much smaller than its expected value is estimated by

Lemma 6.2 *Let N_t be fixed. The event $N_{t+1} < \mathbf{E}(N_{t+1})/4$ is n-negligible if $\mathbf{E}(N_{t+1}) > \lg n$.*

Proof. Note that N_{t+1} is the sum of N_t independent random binary variables. The lemma is obtained from application of the Chernoff inequality (Fact 3.5). ∎

Thus we can assume that $N_t \geq \mathbf{E}(N_t)/4$ simultaneously in all iterations. For simplicity we permit Opt to have

$$N_{t+1} = \frac{N_t}{4}\left(\frac{m}{N_t}\right)^{1-r} .$$

Let $\eta_t = N_t/m$. Then, by dividing the above by m, we have

$$\eta_{t+1} = \eta_t^r/4 .$$

This representation demonstrates the fact that the fraction of sets of a given size decreases "only" double-exponentially, giving rise to the double logarithmic lower and upper bounds. The exact solution of the recurrence is given by

$$\eta_t = \eta_0^{r^t} / 4^{(r^t-1)/(r-1)} \ .$$

For our purposes it is sufficient to write

$$\eta_t \leq \left(\frac{\eta_0}{4}\right)^{r^t}$$

which facilitates an easy counting of the minimal number of iterations:

Lemma 6.3 *If $m \leq n^3$, then the number of levels in* Opt*'s tree is*

$$\Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \ .$$

Proof. Let T be given by

$$T = \frac{1}{\lg r} \lg \frac{\lg \lg n - \lg m}{\lg \eta_0 - 2} \ .$$

Then, for $t < T$,

$$N_t = m\eta_t = m\left(\frac{\eta_0}{4}\right)^{r^t} > m\left(\frac{\eta_0}{4}\right)^{r^T} = \lg n \ .$$

It follows that if Opt executes less than T iterations it will have more than $\lg n$ sets and it cannot terminate. The proof is completed by noting that for $m \leq n^3$

$$T = \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \ .$$

■

Applying this lemma to the estimates in Fact 6.1 will yield the proofs for the lower bounds set by Theorems 5.1, 5.5, Theorem 5.7(b) and Theorem 5.8. In particular,

The basic model Setting $r = 2$ and $m = O(n)$ we have $-\lg \eta_0 = O(1)$ and hence

$$\begin{aligned} \overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\ &= \Omega\left(\lg \frac{\lg n + O(1)}{2 + O(1)}\right) \\ &= \Omega(\lg \lg n) \ . \end{aligned}$$

The basic model Setting $r = 2$ and $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = \lg n/2\lambda + o(1)$ and hence

$$\begin{aligned}
\overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\
&= \Omega\left(\lg \frac{(1+1/\lambda)\lg n}{2 + \lg n/2\lambda + o(1)}\right) \\
&= \Omega\left(\lg \frac{1+1/\lambda}{1/2\lambda + o(1)}\right) \\
&= \Omega(\lg \lambda) \ .
\end{aligned}$$

The retries model Setting $r = \lg n / \lg\lg n$ and $m = O(n)$ we have $-\lg \eta_0 = r \lg r + O(1)$ and hence

$$\begin{aligned}
\overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\
&= \Omega\left(\frac{1}{\lg\lg\lg n} \lg \frac{\lg n + O(1)}{2 + r \lg r + O(1)}\right) \\
&= \Omega\left(\frac{\lg\lg n}{\lg\lg\lg n}\right) \ .
\end{aligned}$$

The retries model Setting $r = \lg \lambda / \lg\lg \lambda$, $m = n^{1+1/\lambda}$, λ fixed, we have $-\lg \eta_0 = -(r - 1)/\lambda \lg n + o(1))$ and hence

$$\begin{aligned}
\overline{\text{DEPTH}}(\text{Opt}, n) &= \Omega\left(\frac{1}{\lg r} \lg \frac{\lg m}{2 - \lg \eta_0}\right) \\
&= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{(1+1/\lambda)\lg n}{2 + (r-1)\lg n/\lambda}\right) \\
&= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{(1+1/\lambda)}{(r-1)/\lambda}\right) \\
&= \Omega\left(\frac{1}{\lg\lg \lambda} \lg \frac{\lambda+1}{r-1}\right) \\
&= \Omega\left(\frac{\lg\lg \lambda}{\lg\lg\lg \lambda}\right) \ .
\end{aligned}$$

6.3 The Parallel Variant

The memory allocation scheme as used by Opt is a slightly different here. Many hash functions can be applied in parallel to the same set. In an iteration t let $m_{i,1}, m_{i,2}, \ldots$ be the cardinalities of ranges of those functions for some subset S_i, and let $m_i = m_{i,1} + m_{i,2} + \cdots$ be the total range used for it. The probability that all those hash functions will be a complete failure is

$$\prod_j m_{i,j}^{1-r_t} \ .$$

This probability is minimized when $m_{i,j} = 2$ for all j. In this case the complete failure probability is

$2^{m_i(1-r_t)/2}$.

Note that if the set size is greater than 2 then a complete separation is not possible if only 2 cells are allocated to a set. This does not pose a problem in our lower bound analysis since we consider any partial separation to be a complete separation.

Let $m = m_1 + m_2 + \cdots + m_{N_t}$ be a memory allocation of the m cells to the N_t sets. The expected value of N_{t+1} is

$$\mathbf{E}(N_{t+1}) = \sum_{i=1}^{N_t} 2^{m_i(1-r_t)} .$$

Once again, this is minimized when all the m_i are equal. Hence we have

Lemma 6.4 *In the parallel variant, all hash functions used by* Opt *are to a range of size 2. In iteration t, $m/2N_t$ functions with a total range of size m/N_t are applied to each one of the N_t sets of size r_t.*

From which we get a recursion formula for N_t

$$\mathbf{E}(N_{t+1}) = N_t 2^{m(1-r_t)/2N_t} .$$

Note that Lemma 6.2 also holds here, so we can write

$$N_{t+1} \geq \frac{N_t}{4} 2^{m(1-r_t)/2N_t}$$

which will take a simpler form using the definition $\nu_t = m/N_t$:

$$\nu_{t+1} = 4\nu_t 2^{(r_t-1)\nu_t/2} \leq 2^{(r_t-1)\nu_t/2+\lg\nu_t+2} .$$

For $r_iteration \geq 4$ we have

$$\nu_{t+1} \leq 2^{r_t\nu_t} .$$

By setting $r_t = 4$, $m = O(n)$, we get that $\nu_0 = O(1)$. The number of iterations T required to decrease the number of subsets below $\lg n$ (i.e., until $\nu_T = n/\lg n$) is $\Omega(\lg^* n)$. This completes the proof of the lower bound part of Theorem 5.9.

The proof of (the chaining variant part of) Theorem 5.10 is conducted in a similar manner to the lower bound proof for the ordinary chaining variant. Let $r = r_1 = \lg^* n$, $r_{t+1} = r_t - 1$, $m = O(n)$. Then by Inequality (2)

$$\nu_0 = O\left((\lg^* n)^{\lg^* n}\right) .$$

We can also write

$$\nu_{t+1} \leq 2^{r\nu_t} \leq 2^{\nu_t^2} .$$

Now, the number of iterations required to achieve $\nu > \lg n$ is at least

$$\frac{\lg^* n - 1 - \lg^* \nu_0}{2} = \Omega(\lg^* n) .$$

6.4 The Retries Variants

To complete the lower bounds analysis we need to discuss the retries variant and provide the proofs for Theorems 5.2 items (a) and (c) of Theorem 5.7. Theorem 5.5 and Theorem 5.10 reference the retries variant as well. These references will be treated in a similar manner.

The retries technique is useful if a certain application of a hash function was not satisfactory according to some criteria. Then, instead of coping with it, the algorithm may try another hash function. However, our simplifications allow Opt a dichotomous classification of poll results. If there was a complete failure in a hash of a specific internal node, then doing a retry is the same as what Opt will do in the next iteration, but with less memory. On the other hand, if this internal node was not a complete failure, we allow Opt to ignore it without any further resource investment. If retry was done on such a node then this may only result in a deeper tree. Retries cannot help in the root node either, as the root node behavior is dominant (Inequality (2)), and even $O(\lg n)$ retries in the root node will not yield a significantly better value for $N_0(r)$.

Thus, the addition of retries to any model combination will not affect the DEPTH lower bounds. Upon further examination of the lower bound proof of the chaining variant, it can be seen that a parallel insertion time of $O(\lg\lg n/\lg\lg\lg n)$ cannot be achieved unless a key is left in $\Omega(\lg\lg n/\lg\lg\lg n)$ nodes, which will nullify the ability of the retries variant to achieve SEARCH $= O(1)$.

The equivalence of an algorithm without retries to an algorithm with retries was possible here because the Opt could use its total memory allowance in each and every iteration. In general, using this technique to transform Alg, an algorithm that uses retries into Alg′, an algorithm that avoids retries, leads to an increase in the memory used by the algorithm by a factor of up to DEPTH(Alg, n).

7 Proofs of Upper Bounds

8 Proofs of Upper Bounds

Any fixed memory allocation scheme is bound to lead to DEPTH $= \Omega(\lg n)$. In order to further decrease the total number of hashing iterations, a more flexible memory allocation is employed. Following the same general scheme used by the idealized algorithm Opt, we try to use almost the same size memory in every iteration. Thus, in every iteration we can increase the memory portion allocated to "stubborn" buckets that managed to survive all previous attempts to hash them.[6] A decreasing geometric series defines the partitioning of the total memory allowance between the iterations. Informally, we can say that although this series decreases quite rapidly, the decrease in the number of active buckets is so much quicker that our algorithm will be in conditions which are very similar to the **Extra Memory** assumption we permitted Opt.

Our basic algorithm is Shallow. This algorithm works for the retries model. Algorithm Shallow' for the basic model is obtained from small changes to Shallow. Algorithm ParShallow for the parallel model, and ParShallow for the chaining model follow.

[6]Those buckets are more unfortunate than stubborn, their persistence is dictated by the laws of probability which predict that there will always be a certain number of failures.

8.1 The Retries Variant

Let us begin with the description of algorithm Shallow which works in the retries variant. The algorithm builds a 2-level hash table as was done by algorithm FKS. Correspondingly the algorithm will have two phases.

Phase I Let S be the input set, $|S| = n$. Algorithm Shallow starts by finding a hash function h which partitions S into buckets $S_1^h, \ldots, S_n^h$ such that

$$\sum_{i=0}^{n} \left|S_i^h\right|^2 < 5n \ . \tag{3}$$

This is done by repeatedly trying functions selected at random from the class $\mathcal{H}_n^1$.

Phase II The algorithm proceeds to find a perfect hash function for all the buckets. This is done by trying memory blocks of rapidly growing size. The block size growth rate is characterized by the sequence β_t:

$$\beta_{t+1} = \beta_t^2/4 \qquad\qquad \beta_1 = 16 \ , \tag{4}$$

or, in an explicit form

$$\beta_t = 2^{2^t - 2} \ . \tag{5}$$

Specifically, Shallow executes simultaneously procedures $\mathsf{Shallow}_1, \ldots \mathsf{Shallow}_{\lceil \lg n \rceil}$. Let j be any. Then, the description of $\mathsf{Shallow}_j$ is as follows. The procedure handles all buckets S_i for which $2^{j-1} < |S_i| \leq 2^j$. Initially, all buckets are active. The procedure executes in stages, each stage constitutes of rounds. In a round of stage t, each of the active buckets is hashed into a memory block of size $\beta_t 2^{2j}$ using a function selected at random from the class $\mathcal{H}^1_{\beta_t 2^{2j}}$. Let Nt, j be the number of keys in active buckets at the beginning of stage t. By the end of each round consider the total number of keys in active buckets for which the selected hash functions were not perfect. We say that a round fails if this number is greater than $2N(t,j)/\beta_t$. If a round fails, then all buckets remains active, and all separations obtained in the round are disregarded (a "retry"). Otherwise, buckets for which a perfect hash function was found become inactive; non-perfect hash functions are disregarded and the procedure moves to stage $t+1$. The procedure terminates when there are no more active buckets. The algorithm terminates when all procedures terminate.

Analysis of $\overline{\text{DEPTH}}$(Shallow, n) It follows from Fact 3.3 that with probability at least $1/2$ a function selected at random from $\mathcal{H}_n^1$ will satisfy Inequality (3). Therefore, the expected number of functions selected until Inequality (3) is attained is $O(1)$. The contribution of this part of the algorithm to the $\overline{\text{DEPTH}}$(Shallow, n) is at most constant.

The number of stages of $\mathsf{Shallow}_j$, $j = 1, \ldots \lceil \lg n \rceil$ can be determined by the following simple consideration. If t is such that $\beta_t \geq 4n$ then by the end of stage t the number of active buckets is less than $1/2$, or in other words $Nt, j = 0$. It follows from Equation (5) that taking $t = O(\lg \lg n)$ is enough to ensure $\beta_t \geq 2n$. Hence, the number of stages of $\mathsf{Shallow}_j$ is $O(\lg \lg n)$.

It follows from Fact 3.3 that a function selected at random from $\mathcal{H}^1_{\beta_t 2^{2j}}$ is perfect for a bucket of size at most 2^j with probability at least $1 - 1/\beta_t$. Hence, the expected number of active keys by the end of a round is at most $N(t,j)/\beta_t$, and by Markov's inequality the probability of any given round to be the last in a stage is at least $1/2$. The expected number of rounds in each stage is therefore at most 2, and the total expected number of rounds is $O(\lg\lg n)$.

Round failures are independent. We can therefore apply Chernoff bounds (Fact 3.5) getting that there is a constant C, such that for any $\gamma > C$, the probability that the total number of rounds will be more than γ times its expected value is $o(1/\lg^{\gamma/C} n)$. Since there are at most $\lceil \lg n \rceil$ procedures executing simultaneously, the expected parallel time until the slowest procedure terminates is also $O(\lg\lg n)$. We therefore have $\overline{\text{DEPTH}}(\text{Shallow}, n) = O(\lg\lg n)$.

Analysis of $\text{SPACE}(\text{Shallow}, n)$ In phase I, Shallow applies a series of retries into a memory of size n. The total memory used by this phase is therefore n. Consider any procedure Shallow_j. In stage t there are at most $N_{t,j}/2^j$ active buckets, each bucket is hashed into memory of size $\beta_t 2^{2j}$. The total memory used in the stage is therefore at most $N_{t,j}\beta_t 2^j$. (Recall that no space is charged for retry nodes.) Since $N_{t+1,j} \leq 2N_{t+1,j}/\beta_t$ and $\beta_{t+1} = \beta_t^2/4$ we get that the bound for memory usage in stage $t+1$ is at most $1/2$ of the bound of memory used in a round of stage t. Thus, the total memory usage is up to a constant factor the same as memory used in stage 1. Using the fact that initially $\mathbf{E}\left(\sum |S_i|^2\right) = O(n)$ and $\beta_1 = O(1)$ we get that even accounting for the possible doubling of set sizes due to rounding, $\text{SPACE}(\text{Shallow}, n) = O(n)$.

Analysis of $\overline{\text{TIME}}(\text{Shallow}, n)$ Note that the expected number of times Shallow accesses any memory cell is constant. Then, $\overline{\text{TIME}}(\text{Shallow}, n) = O(n)$ follows from $\text{SPACE}(\text{Shallow}, n) = O(n)$. Since all functions used in Shallow are polynomials of degree 1, we get that the number of operations is linear even if arithmetic operations are counted.

Analysis of $\text{SEARCH}(\text{Shallow}, n)$ Recall that algorithm Shallow constructs a 2-level hashing scheme for S. The first level consists of the function selected at phase I, which splits S to buckets. The second level consists of the perfect hash functions found in phase II for each of the buckets. We therefore have $\text{SEARCH}(\text{Shallow}, n) = 2$. This completes the proof of Theorem 5.3.

8.2 The Basic Model

Algorithm Shallow' which works in the basic model is derived from Shallow, by replacing retry nodes in Shallow by refining nodes in Shallow'. A description of the modifications follows.

Phase I Algorithm Shallow' starts partitioning S into buckets $S_1, S_2, \ldots, S_n$, by applying a hash function selected at random from $\mathcal{H}^1_n$. It follows from Fact 3.2 that

$$\mathbf{E}\left(\sum_{i=1}^{n} |S_i|^2\right) \leq 2.5|S| \ . \tag{6}$$

Phase II Similarly to Shallow, Shallow' partition into procedures $\mathsf{Shallow'}_1, \ldots \mathsf{Shallow'}_{\lceil \lg n \rceil}$. All buckets S_i, $2^{j-1} < |S_i| \leq 2^j$ are refined by $\mathsf{Shallow'}_j$. However, smaller buckets generated from refining do do not move between procedures. The allocation of memory to buckets needs to be modified since there are greater variations in the size of buckets within a procedure. Instead of allocating a memory block of size 2^{2j} to all buckets associated with $\mathsf{Shallow'}_j$, the allocation for smaller buckets decreases in proportion to their size. Specifically, in stage $t \geq 1$ of $\mathsf{Shallow'}_j$ a bucket of size r is hashed $\beta_t r 2^j$ range using a function selected at random from $\mathcal{H}^1_{\beta_t r 2^j}$. If this function is perfect then the bucket becomes inactive. The probability that this happens is at least $1 - r/(2^j \beta_t) > 1 - 1/\beta_t$. If the function is not perfect then the bucket is broken into smaller sub-buckets, and processing continues (within the same procedure) for all sub-buckets which have two or more keys. Let Nt, j be the number of keys in active buckets at the beginning of stage t. Then, the rounds of a stage t continue until the number of keys in active buckets drops below $2N(t,j)/\beta_t$.

8.2.1 Analysis

Note that the probability that a round will fail (i.e., that it will not be terminate the stage) is again at most $1/2$. The expected number of rounds in each round is therefore a constant. Also, we loose no generality in assuming that rounds are independent. Following the same lines of the analysis of $\overline{\textsc{Depth}}(\mathsf{Shallow}, n)$ we have $\overline{\textsc{Depth}}(\mathsf{Shallow'}, n) = O(\lg \lg n)$, and therefore $\overline{\textsc{Search}}(\mathsf{Shallow'}, n) = O(\lg \lg n)$.

The space used by Shallow' in Phase I is n. For the analysis of space used in Phase II, we apply similar considerations to those used in the analysis of the space used by Shallow. A moment's thought will show that also here space usage decreases geometrically with stage number, and therefore space usage in stage 1 dominates all others. However, in this case we can only get a bound for the expected memory usage. Memory used in stage 1 is at most a constant times $\mathbf{E}\left(\sum_{i=1}^{n} |S_i|^2\right)$. Applying Inequality (6) we get $\overline{\textsc{Space}}(\mathsf{Shallow'}, n) = O(n)$ and consequently $\overline{\textsc{Time}}(\mathsf{Shallow'}, n) = O(n)$. This completes the proof of Theorem 5.4.

8.3 The Parallel Variant

Both Shallow and Shallow' are not applicable directly to PRAMs, but the ideas underlying them can be used to supply an actual PRAM algorithm [22]. In contrast, the following ParShallow algorithm seems much harder to implement. In each successive iteration, more and more processors are drafted to hash fewer and fewer keys. Locating those keys in need of help and organizing help for them apparently requires considerable time, and ParShallow completely ignores that.

The description and the analysis of Shallow are used here as a skeleton. The following differences apply: the sequence β_t starts with $\beta_1 = 4$ and increases at a quicker rate, $\beta_t = 2^{\beta_t - 2}$. In stage t of $\mathsf{ParShallow}_j$ the $\beta_t r 2^j$ size memory block allocated to a bucket of $r \leq 2^j$ keys is further divided into $\beta_t 2^j / 2r$ sub-blocks of $2r^2$ cells each. The parallel hashings are then done into these sub-blocks. The failure probability is thus at most

$$2^{-\beta_t r/2r'} = 2^{-(r/r'-1)\beta_t/2} \cdot 2^{-\beta_t/2} \leq 2^{-2(r/r'-1)} \cdot 2^{-\beta_t/2} < \frac{r'}{r} \cdot 2^{-\beta_t/2} .$$

In a stage, rounds continue to the point when the number of active keys is at most $2 \cdot 2^{-\beta_t/2} N_t(r)$, and hence

$$N_{t+1}(r) \leq 2 \cdot 2^{-\beta_t/2} N_t(r) \ .$$

This last bound, together with the definition of the sequence $\{\beta_t\}$, proves that $\overline{\text{DEPTH}}(\text{ParShallow}) = O(\lg^* n)$. To see that $\overline{\text{SPACE}}(\text{ParShallow}) = \text{SPACE}_r(\text{ParShallow}) = O(n)$ note that the total memory used by a round of iteration $t+1$ is at most $1/2$ of the total memory used by a round of stage t. This completes the proof of the upper bound part of Theorem 5.9.

8.4 The Chaining Variant

The idea behind the reduction of DEPTH to $O(\lg\lg n/\lg\lg\lg n)$ is to replace Phase II of Shallow by two phases:

Phase IIa Use the prototype of algorithm Shallow to continuously break subsets until they are all small enough, instead of the usual attempt to achieve a complete separation. "Small enough" in this context means that the subset size is $\leq R = \lg\lg n/\lg\lg\lg n$.

Phase IIb When all the buckets are this small, then the ability of this model variant to store one element in every internal nodes is applied. No matter which function is used on nodes containing small subsets, at most R levels will be added to the tree.

The main point in which the first phase of Shallow_c differs from Shallow is that it handles only subsets of at least R elements; as soon as a subset is broken into smaller pieces, the algorithm ceases to handle it. Another difference is that hash functions used in an iteration must have a "good breaking" property.

8.4.1 Good Breakers

Definition 8.1 *Let $\mathcal{H}$ be a class of hash functions mapping sets of size r into memory of size βr^2, then $\mathcal{H}$ is a* good breaker *if the probability that h picked at random from $\mathcal{H}$ is not R-perfect is at most β^{1-R}.*

This property is achieved by random functions [21, Chapter 2]. However, true random functions are not useful for hashing as they require huge space for representation, and consequently non-constant evaluation time.

The class $\mathcal{H}^{R-1}$ is a good breaker too, and its members can be represented efficiently. Unfortunately, each application of a hash function takes $O(R)$, which amounts to a total of $O(\lg\lg n)$ run time, (although the number of hash function applications is still $O(\lg\lg n/\lg\lg\lg n)$).

The class $\mathcal{R}$ is probably best suited for a more practical implementation of our algorithm as it offers the advantages of sub-linear representation and constant time evaluation.

8.4.2 Parameters Setting and Analysis

The sequence $\{\beta_t\}$ is defined in algorithm ParShallow by

$$\beta_{t+1} = \beta_t^R/4 \qquad \beta_1 = O(1) \ . \tag{7}$$

As in Shallow, the basic unit of memory allocation is $\beta_t 2^{2j}$ in a stage t of ParShallow$_j$. A subset of size $r \leq 2^j$ belonging to this procedure is hashed using a good breaker into a $\beta_t r 2^j$ size memory block. The probability that the subset will not be broken into small enough subsets is at most β_t^{1-R}. If a stage's rounds carry on until $N(r) \leq 2N_t(r)\beta_t^{1-R}$ then the expected number of rounds in an iteration will be ≤ 2 here too. Thus we have

$$N_{t+1}(r) \leq 2N_t(r)\beta_t^{1-R} \ . \tag{8}$$

Combining the recurrences (7) and (8) we see that memory requirements decrease geometrically:

$$N_{t+1}(r)\beta_{t+1} \leq N_t(r)\beta_t/2 \ .$$

Adding to this the fact that the root node function must have satisfied Inequality (3), we infer that the total memory usage is linear.

Solving the recurrence (7) we have

$$\beta_t = \beta_1^{R^{t-1}} 4^{-(R^t-1)/(R-1)} \ ,$$

from which it follows that the number of iterations is $O(R)$, completing the proof of Theorem 5.6.

9 Conclusions

Hashing models The hashing model proposed leads to an alternate view of hashing algorithms as element distinctness proof generator. In our lower bound analysis we assumed that all the hash functions (the proof components) are completely random. However, it is not difficult to see that if the universe is not too large, say polynomial in the size of the input set, and if non-random functions can be used, then the lower bounds do not hold. (E.g., by an integer sorting algorithm.)

Is the random functions assumption essential? It was shown before [3, 59] that random functions are optimal in certain hashing situations. On the other hand, hashing algorithms that are based on open addressing or on the FKS scheme (such as the one described in [23]) as well as the upper bounds presented here do not assume the existence of random functions. We conjecture that the lower bound results hold for super polynomial sized universe even if arbitrary function are allowed.

Although the model proposed is very general, it does not cover all hashing algorithms (e.g., the one presented in [19]). It may be interesting to define more general models and to derive lower bounds for this models as well.

Reliability of polynomial hash functions Polynomial hash functions were successfully used in the past as a replacement for truly random hash functions. However, only expected value results were known for the measures of performance of polynomial hash functions. Thus, hashing algorithms which use these functions are typically not-solid. In fact, the real time dictionary—an application which requires performance guaranteed with high probability, motivated Dietzfelbinger and Meyer auf der Heide in their invention of the class $\mathcal{R}$. Selecting a function from $\mathcal{R}$ meets the high probability requirement but with a cost of a large number of random bits, and of considerable space for its representation. Can the simpler polynomial hash functions give similar performance? In a work which followed this thesis Dietzfelbinger, Gil, Matias and Pippenger [15] show that a small increase of the polynomial degree improves the reliability of polynomial hash functions.

Parallel Hashing The hashing problem considered in this dissertation was "static", i.e., the input set S is fixed. If the S is changing dynamically, then the problem is known as the *dictionary* problem, in which it is required to maintain a data structure that supports the instructions *insert*, *delete* and *lookup*. The fastest optimal dictionary published prior to this thesis is due to Dietzfelbinger and Meyer auf der Heide [17]. Its time complexity is $O(n^\epsilon)$ for any constant $\epsilon > 0$. The static hashing scheme presented in chapter 7 can be extended to a dynamic hashing scheme with similar complexities [24]. Furthermore, a recent result is that a dictionary can be maintained optimally in $O(\lg^* n)$ time. An important application of parallel hashing and the dictionary algorithm is in the recent work of Karp, Luby, and Friedhelm Meyer auf der Heide on the extremely fast simulation of a PRAM on a complete network [32].

Acknowledgments Fruitful comments made by Faith E. Fich are gratefully acknowledged.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1983.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1977.

[3] M. Ajtai, János Komlós, and Endre Szemerédi. There is no fast single hashing function. *Information Processing Letters*, 7:270–273, 1978.

[4] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian paths and matchings. *J. Comp. Syst. Sci.*, 18:155–193, 1979.

[5] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[6] Francine Berman, Mary E. Bock, Eric Dittert, Michael J. O'Donnel, and Darrell Plank. Collections of functions for perfect hashing. *SIAM Journal on Computing*, 15(2):604–618, 1986.

[7] Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, February 1972.

[8] Larry J. Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[9] Pedro Cellis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 281–288, October 1985.

[10] C. C. Chang and C.H.K Chang. An ordered minimal perfect hashing scheme with single parameter. *Information Processing Letters*, 27(2):79–83, February 1988.

[11] W. C. Chen and Jeffrey S. Vitter. Analysis of early-insertion standard coalesced hashing. *SIAM Journal on Computing*, 12(4):667–676, 1983.

[12] W. C. Chen and Jeffrey S. Vitter. Analysis of new variants of coalesced hashing. *ACM Transactions on Database Systems*, 9(4):616–645, 1984.

[13] W. C. Chen and Jeffrey S. Vitter. Deletion algorithms for coalesced hashing. *The Computer Journal*, 29(5):436–450, 1986.

[14] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–507, 1952.

[15] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable. Submitted for publication, November 1991.

[16] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proc. of the 29th IEEE Annual Symp. on Foundation of Computer Science*, pages 524–531, October 1988. Also, Revised Version: Tech. Report, University of Paderborn, FB 17 Mathematik/Informatik, 1991.

[17] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. An optimal parallel dictionary. In *1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[18] Martin Dietzfelbinger and Friedhelm 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*, 1990.

[19] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hahshing in real time. In *Proc. of 17th International Colloquium on Automata Languages and Programming, Springer LNCS 443*, pages 6–19, 1990.

[20] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[21] Joseph Gil. *Lower Bounds and Algorithms for Hashing and Parallel Processing.* PhD thesis, The Hebrew University of Jerusalem, Givat Ram 91904, Jerusalem, Israel, November 1990.

[22] Joseph Gil and Yossi Matias. Fast and efficient simulations among CRCW models. Manuscript, 1990.

[23] Joseph Gil and Yossi Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, January 1991.

[24] Joseph Gil, Yossi Matias, and Uzi Vishkin. A fast parallel dictionary. manuscript, 1990.

[25] Joseph Gil, Friedhelm Meyer auf der Heide, and Avi Wigderson. Not all keys can be hashed in constant time. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 244–253, 1990.

[26] Gaston H. Gonnet. Expected length of the longest sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[27] Gaston H. Gonnet and J. Ian Munro. Efficient ordering of hash tables. *SIAM Journal on Computing*, 8(3):544–555, August 1979.

[28] M Gori and G. Soda. An algebraic approach to cichelli's perfect hashing. *BIT—Computer Science Numerical Mathematics*, 29(1):2–13, 1989.

[29] Leonidas I. Guibas and Endre Szemerédi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16:226–274, April 1978.

[30] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[31] Anna R. Karlin and Eli Upfal. Parallel hashing—an efficient implementation of shared memory. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 160–168, May 1986.

[32] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. Manuscript, to appear, December 1991.

[33] Gary D. Knott. Direct chaining with coalescing lists. *Journal of Algorithms*, 4(1):7–21, 1984.

[34] Gary D. Knott. Linear open addressing and Peterson's theorem rehashed. *BIT—Computer Science Numerical Mathematics*, 28(2):364–371, 1988.

[35] Gary D. Knott and Pilar de la Torre. Hash table collision resolution with direct chaining. *Journal of Algorithms*, 10(1):20–34, 1989.

[36] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[37] Donald E. Knuth. Computer science and its relationship to mathematics. *Am. Math. Monthly*, 8:323–343, 1974.

[38] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1986.

[39] Per-Åke Larson. Analysis of uniform hashing. *Journal of the ACM*, 30:805–819, 1983.

[40] George S. Lueker and Mariko Molodowitch. More analysis of double hashing. In *Proc. of the 20th Ann. ACM Symp. on Theory of Computing*, 1988.

[41] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. In *Proc. of 17th International Colloquium on Automata Languages and Programming, Springer LNCS 443*, pages 729–743, 1990.

[42] Yossi Matias and Uzi 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. Also in UMIACS-TR-91-65, Inst. for Adv. Comp. Studies, Univ. of Maryland, April 1991.

[43] Kurt Mehlhorn. *Data Structures and Algorithms.* Springer-Verlag, Berlin Heidelberg, 1984.

[44] H. Mendelson and U. Yechiali. A new approach to the analysis of linear hashing. *Journal of the ACM*, 27:474–483, 1980.

[45] Georg Ch. Pflug and Hans W. Kessler. Linear probing with a nonuniform address distribution. *Journal of the ACM*, 34(2):397–410, 1987.

[46] Boris K. Pittel. Linear probing the probable largest search time grows logarithmically with the number of records. *Journal of Algorithms*, 8(2):236–249, 1987.

[47] Boris K. Pittel and Jenn-Hwa Yu. On search times for early-insertion coalesced hashing. *SIAM Journal on Computing*, 17(3):492–503, June 1988.

[48] Patricio V. Poblete and J. Ian Munro. Last-come-first-served hashing. *Journal of Algorithms*, 10(2):228–248, 1989.

[49] M. V. Ramakrishna. Analysis of random probing hashing. *Information Processing Letters*, 31(2):83–90, 1989.

[50] Ronald L. Rivest. Optimal arrangment of keys in a hash table. *Journal of the ACM*, 25(2):200–209, April 1978.

[51] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, 1990.

[52] Michael Sipser. A complexity theoretic approach to randomness. In *Proc. of the 15th Ann. ACM Symp. on Theory of Computing*, pages 330–335, April 1983.

[53] Larry J. Stockmeyer. The complexity of approximate counting. In *Proc. of the 15th Ann. ACM Symp. on Theory of Computing*, pages 118–126, April 1983.

[54] Robert E. Tarjan and Andrew C.C. Yao. Storing a sparse table. *Communications of the ACM*, 21:606–611, November 1979.

[55] Jeffrey D. Ullman. A note on the efficiency of hashing functions. *Journal of the ACM*, 19(3):569–575, July 1972.

[56] Jeffrey S. Vitter. *Analysis of Coalesced Hashing.* PhD thesis, Stanford University, Stanford, CA, August 1982. Technical Report STAN-CS-80-817.

[57] Vincent G. Winters. Minimal perfect hashing in polynomial time. *BIT—Computer Science Numerical Mathematics*, 30(2):235–244, 1990.

[58] Andrew C.C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

[59] Andrew C.C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32(3):687–693, 1985.