Not All Keys Can be Hashed
in Constant Time
(Preliminary Version)

Joseph Gil \footnote{Dept. of Comp. Sc. Hebrew University, Jerusalem 91904, Israel.}
Friedhelm Meyer auf der Heide \footnote{Research supported in part by the Leibnitz Center for Research in Computer Science.}
Avi Wigderson \footnote{Research supported in part by DFG grants Me 872/1-3, and the Leibnitz 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, communication, parallel and distributed computation, efficient algorithm design and even complexity theory [8,9].

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

A fundamental result of [4] shows that a elements from any universe can be hashed to a linear size table in linear time, allowing for constant search time. It was observed, however, that although average insertion time per element is constant, parallel application of this (and other) algorithms cannot work in constant time. The reason is that while the average is constant, some elements will have to be hashed non-constant number of times.

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

2 The Basic Model

The process of inserting $n$ elements $A = \{a_1, a_2, \ldots, a_n\}$ taken from some universe $U$ into a hash table can be thought of as a process of refining partitions, and is simply depicted by a tree. Originally, all elements reside in a single node (the root). The algorithm chooses a range $m$, picks a function $f : U \rightarrow [m]$ according to some distribution, and partitions $A$ into sets $A_1, A_2, \ldots, A_m$ (some empty) s.t. $x_j \in A_i$ if $f(x_j) = i$. The function $f$ is assigned to the root, and the sets $A_i$ move respectively to $m$ children of the root. The process repeats for each $A_i$ that contains at least two elements. This refining process halts when every leaf contains at most one element from $A$.

A search for an element $b \in U$ in the hash table is easily performed by following a path from the root that is determined by applying the hash functions at internal nodes to the element $b$, and when reach-
In a leaf comparing $i$ to the element residing there (if there is one).

This approach leads to an alternate view of hashing algorithm as an element distinctness proof generator. The input is a set of distinct keys taken from 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.

2.1 Comments

1. 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. The existence of algorithms for the dynamic case that achieve both constant amortized time and minimal worst case remain open.

2. Most common algorithms are stronger than the process we described — they we retire 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 leaves. We shall consider these generalizations later.

3. Yao's cell probe model [10], which is the standard general model for hashing can also be described as a tree in a similar way. Our model differs from it in a way that a decision tree differs from a Turing machine. It allows each cell a limited number of bits, (depending on $U$), but these can encode arbitrary objects and be computed for free. 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, but less general.

4. The two types of strategic decisions made by a specific algorithms are the choice of range (number of children) for the hash function, and the choice of distribution on functions to this range. Simple convexity considerations show that the second choice is optimal (for the relevant method) when choosing a random function, i.e. needing each element of $U$ independently to each possible child with uniform distribution. Hence, analyzing the hashing process is reduced to analyzing a natural process of successively throwing identical balls into boxes until all the balls reside in distinct boxes.

2.2 Resources

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

Space The space required, or hash table size, denoted by $S_H(A)$, is simply the total number of nodes in the tree.

Insertion Time We denote by $T_H(A)$ the total insertion time. This is the total depth of leaves containing an element, i.e. the number of applications of hash functions to elements. Each application counts as one time unit. Later we discuss the number of arithmetic operations.

Parallel insertion time Denoted by $D_H(A)$ is simply the depth of the tree. (Allowing to refine all leaves in parallel, or equivalently, to apply one hash function to each element in parallel.) This parameter has two important meanings for sequential algorithms as well. It captures the number of functions needed to resolve the "worse" pair of elements, and the worse case insert and search time. Search time will not be identical to insertion time in the more general algorithms, so we devote a different notation for it.

Maximum search time This is the largest number of function applications to find out if $x \in U$ is in $A$, using the tree generated by $H$ on $A$. This parameter will be denoted by $P_H(A)$.

Let $P$ be a generic parameter (e.g. $S_T D F$). Calligraphic case letters will denote the expectation of our random variables, with respect to the random choices made by the algorithm $H$. So $E_{\mathcal{H}} \mathcal{P}_H(A) = E_{\mathcal{H}} \mathcal{P}_H(A)$. We denote by

$$P_H(a) = \max_{A \subseteq U} P_H(A).$$
the performance of $H$ on a worst case set $A$, and by

$$P(n) = \min_H P_H(n)$$

the performance of the best algorithm on its worst case set $A$.

2.3 Variants of the Model

Finally, we consider more powerful algorithms than described above, in the following variants. Most hashing algorithms deviate from our basic model by allowing one or both of the following:

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 fall into one or very few boxes. 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. To maintain the meaning of depth (the parameters $D, D'$), we create one single child to $v$, and move all the balls there. We attach the superscript $v$ to measures for this model, e.g. $S^v_H(A)$ and $D^v(n)$ etc. Note that here $F^v$ may be much smaller than $D^v$, since when searching no function application is needed at a node that has only one child.

Chaining Here we allow 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 at $v$. The word chaining is termed here since this variant models hashing techniques where a chain of keys can be stored in hashing array positions. We add the superscript $v$ to the complexity measures. Clearly $F^v$ and $D^v$ are the same again, since even if there is no branching at node $v$, the element we search for should be compared to the one residing at $v$.

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 functions. Superscript $p$ is added in this variant to the complexity measures. This variant can be combined with the two previous ones; if retries are permitted then the algorithm not to use any of the the hash were tried; if exploiting internets made then the algorithm would be at $v$ no matter which (if any) selected for the node. Despite variant does not lead directly algorithm. The major difficulty idle processors to tasks.

3 Results

The most interesting algorithms achieve $S_H(n) = O(n)$, i.e. linear space 1979 paper "Should Tables be Sorted" if one can simultaneously achieve $F(n) = O(1)$. In our basic model:

Theorem 1. If $S_H(n) = O(n)$ then

$$D_H(n) = F_H(n) = \Omega(\log n)$$

However, allowing retries, Yao gave $Y$ achieving $S_Y(n) = O(n)$ and $F_Y$ large enough universes $U$. Fredon Szemerédi [4] closed the gap by an achieving $S_{FSK}(n) = O(n)$ and $F_{FSK}$ for any universe $U$ (and any set their algorithm, we find that while $T_{FSK}(n) = O(n)$, i.e. on the average only a constant number of functions), $D_{FSK}(n) = \Omega(\log n)$, etc. will be hashed $\Omega(\log n)$ times, an bound on hashing the elements in the FSK scheme. A natural question is whether this parameter decrease answer this question by giving:

Theorem 2. If a hashing scheme $O(n)$ size memory then $D(n) = \Omega(\log \log n)$.

With the help of retries, the lower bound theorem 1 can be achieved.

Theorem 3. There is an algorithm retries and linear space $(S_{FSK}(n) = T_{FSK}(n) = O(n)$, as well as $D_{FSK}(n)$ and $F_{FSK}(n) = O(1)$. 

246
This algorithm is a variant to the FKS algorithm. The improvement in $D(n)$ is achieved by using a different, more adaptive memory allocation scheme while doing the retries. This algorithm is optimal with respect to all parameters, even if we count arithmetic operations and limit word size to $O(\log U)$. Moreover, if we restrict the algorithm to the basic model, all the parameters, except for $F(n)$ that will be the same as $D(n)$ will remain optimal.

The general trade-off between space and depth is given by

**Theorem 4** If $S(n) = n^{1+1/\lambda}$ then $D(n) = \Omega(D(n)) = \Omega(\log \lambda)$.

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** Both $S(n) = O(n)$ and $D(n) = O(\log n / \log \log \log n)$ can be achieved simultaneously.

The algorithm behind this theorem uses truly random hash functions (or, equivalently, high degree polynomials), and these bounds hold only in our structured model. We do not know if the above bounds are achievable if we charge for arithmetic operations and limit word size. What we can show is that this means improvement of $\log \log \log n$ is best possible.

**Theorem 6** If $S(n) = O(n)$ then $D(n) = \Omega(\log \log n / \log \log \log n)$.

The general trade-off is given by:

**Theorem 7** If $S(n) = n^{1+1/\lambda}$ then $D(n) = \Omega(\log \lambda / \log \log \lambda)$.

As before, adding the power of retries to this variant of the model cannot improve $D(n)$.

**Theorem 8** Let $m$ be the total memory used by the algorithm, if $m = n^{1+1/\lambda}$ then $D^m(n) = \Omega(D^m(n))$.

"Multiple retries" which are permitted in the parallel variant, perhaps at the cost of memory, do help because they allow folding many iterations into one.

**Theorem 9** If memory usage is restricted then $D^m(n) = \Theta(\log n)$ and $F^m(n) = \Theta(1)$.

The algorithm here is not directed at the $PRAM$, where the time for assigning vectorizers has to be counted.

### 4 Proofs of lower bounds

We view hashing algorithms from an $H^*$ perspective. Each parallel iteration is allowed to separate all keys that were not assigned to previous iteration. Thus successive iterations respond to successive tree levels.

For a certain hashing problem we define $H^*$, the best possible algorithm. To prove the lower bounds we need to prove the expected number of iterations of $H^*$ make the following assumptions:

**Extra memory** Say that the problem's memory usage to a total of $m$. This restriction will be weakened, and we will be allowed to use $m$ mem.

**Partial separation** A mapping from a set of $n$ keys to memory will be called partial if there are two keys in the set that are not separated (or distinct cells). $H^*$ may consider separation as being a total loss. It is extremely unlikely that if all $n$ keys were mapped to the same cell, there would be a complete fault. So failure of $H^*$ needs to be passed to the failure of $H^*$.

**Restricted set size** In the $i^{th}$ iteration the algorithm needs to deal only with nodes of $s_i$ keys. Smaller and bigger sets are completely ignored. The exact value of $s_i$ is specified later.

**Early termination** $H^*$ need not end where there are fewer than $k^1$. As soon as the number of key pairs reaches that bound, $H^*$ can terminate.

---

1 Base $2$ is implicit in all logarithms.
Higher success probability. While analyzing $H^*$ we will assume that failure probability is determined by $s = s_0$, although $s_i$ keys are actually mapped. It will be shown that this may only decrease the failure probability.

The rest of this section is outlined as follows. We first compute $k(s)$, the initial number of sets of size $s$. Next we estimate $k_t = k(s_t)$, the number of those sets in the $t + 1$ iteration as a function of $k_{t-1}(s)$. Then we derive an explicit formula for $k(s)$. Proofs are then completed by picking suitable values of $s$ for the different cases, and obtaining the lower bound from that. This process is done for the basic model and the chaining variant together, and then it is repeated for the parallel variant. We end with a remark explaining why all lower bound proofs are applicable to the retries variants of the model.

4.1 Root node hashing

The root node will be called iteration 0. In it $n$ keys are hashed into $m$ memory cells. The set $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is separated using a random function $f: U \rightarrow [m]$ into subsets $A_1, A_2, \ldots, A_m$. Let $k = k(s)$ be the number of subsets that have exactly $s$ elements in them and let $n = k(s) = k_t(s)/m$.

Lemma 1 Let $\mu = n/m$, if $\mu = \mu(n) < C$ for some constant $C$ and $s = o(n)$ then

$$\lim_{n \rightarrow \infty} E(k(s)) = m e^{-\mu} \frac{\mu^s}{s!}$$

$$\sigma(k(s)) \leq E(k)^{1/2}$$

Proof Let $z_i = z_i(s)$ be the random variable defined by

$$z_i = \begin{cases} 1 & \text{if } |A_i| = s \\ 0 & \text{otherwise} \end{cases}$$

since the mapping done by $f$ is uniform

$$E(z_i) = \text{Prob}(z_i = 1) = \binom{n}{s} \left(\frac{1}{m}\right)^s \left(1 - \frac{1}{m}\right)^{n-s}$$

When $s = o(n)$ this can be approximated by the Poisson distribution

$$\lim_{n \rightarrow \infty} E(z_i) = e^{-\mu} \frac{\mu^s}{s!}$$

The proof of the second part of the result.

Definition 1 Events that occur with probability $\Omega(n^{-\epsilon})$ for some $\epsilon > 0$ are negligible events.

Corollary 2 If $E(k(s)) = \Omega(n^\epsilon)$ then the event $k(s) < E(k(s))/4$ has probability $O(n^{-\epsilon})$ and hence is negligible.

Negligible events will be ignored in the discussion, since even if they could not cut any resource investment, the effect on the algorithm performance measures do not significantly.

4.2 The basic model and the chaining variant

In any iteration $H^*$ must handle $k_t$ keys each using $m$ memory cells. We should allocate memory to cells in order to minimise the number of failures $k_t$. A lemma reveals the memory allocation by $H^*$.

Lemma 3 $H^*$ uses a balanced allocation scheme, that is each of the $k_t$ sets are in $m/k_t$ cells.

Proof If $s_t$ keys are mapped by a uniform random mapping into $m$ cells, then the complexity is $\omega^{1-s_t}$. This probability is zero as $s$ increases, and therefore a memory allocation of the $m$ cells to the $k_t$ keys may be unacceptable if some of them are not utilised. Let $m = m_1 + m_2 + \cdots + m_{k_t}$ and $s = s_1$, the initial size of the memory allocation.

$$E(k_{t+1}) = \sum_{i=1}^{k_t} m_i^{-1}$$

Clearly this is minimised when all

$H^*$ uses a complete failure protocol from $s = s_1$, the initial size of the
and by lemma 3

\[ E(k_{t+1}) = k_t \left( \frac{m}{k_t} \right)^{1-n} \]

The probability that \( k_{t+1} \) will be much smaller than its expected value is estimated by the following lemma

**Lemma 4** \( \Pr \left( k_{t+1} < \frac{E(k_t)}{4} \right) < e^{-E(k_t)/4} \)

**Proof** Note that for any i, the sum of \( k_t \) independent binary random variables, with equal distribution. The lemma is obtained from a application of Chernoff inequality.

If \( k_t = \Omega(\log n) \) and \( t = O(\log \log n) \) then the event \( k_t < E(k_t)/4 \) for \( i \leq t \) is negligible. Since the union of \( O(\log \log n) \) negligible events is also a negligible event we can assume that \( k_t < E(k_t)/4 \) will never happen. For simplicity we permit \( H^* \) have

\[ k_{t+1} = \frac{k_t}{4} \left( \frac{m}{k_t} \right)^{1-n} \]

Defining \( \alpha_t = \frac{k_t}{m} \), we get the above in a simpler form

\[ k_{t+1} = \frac{k_t}{4} \left( \frac{m}{k_t} \right)^{1-n} \]

This representation demonstrates the fact that the fraction of sets of a given size decreases “only” double-exponentially, which gives rise to the double logarithmic lower and upper bounds. The solution of the above recurrence is given by

\[ k_t = \frac{k_0^{2^{-t}}}{4^{2^{-t}}} \]

Again we benefit \( H^* \) by setting

\[ \alpha_t = \left( \frac{k_0}{4^{2^{-t}}} \right)^{1-n} \]

The following lemma is the core of all lower bound proofs

**Lemma 5** \( H^* \) tree has at least

\[ \Omega \left( \frac{\log \left( \frac{\log 2}{\log k_0} - 1 \right)}{\log \alpha} \right) \]

levels.

**Proof** The initial fraction of sets of size \( \alpha \) is

\[ m_0(\alpha) = m \left( \frac{n_0}{4} \right)^{\alpha} = \left( \frac{n_0}{4} \right)^{\alpha} \]

Before \( H^* \) performs \( r \) iterations it will have more than \( \log n \) sets, and it will terminate. The explicit value of \( r \) is

\[ r = \frac{\log \log n - \log \alpha - 1}{\log \alpha} \]

which is (asymptotically) the bound on \( \alpha \).

When \( H^* \) cannot use internal nodes only sets of size exactly \( n \) i.e. \( \alpha_1 \) When usage of intermediate nodes of fixed size \( s \) can no longer be true, the number of iterations is dependent on complete failure to hash a set will occur by 1. Instead define \( s_0 = \frac{s}{s-1} \) and \( s_1 = s \). Note that in this variant \( s_0 \) is larger than the desired lower bound for iterations.

The following lemma will estimate certain pairs of \( s \) and \( m \) we are interested that in all of those cases \( E(k) \) is at least some \( \epsilon > 0 \) and hence the event negligible.

**Lemma 6** The initial \( k = k_0(\alpha) \) node hashing is given by

1. If \( s = 2 \) and \( m = O(n) \) then

\[ E(k) = \Omega(n) \]

2. If \( s = 2 \) and \( m = n^{1+1/\lambda} \) for \( \lambda > 1 \)

\[ E(k) = \Omega(n^{1+1/\lambda}) \]

3. If \( s = \frac{\log 2}{\log \log 2} \) and \( m = O(n) \)

\[ E(k) = \Omega \left( n^{1-\log_2 \log_2 n} \right) \]

4. If \( s = \frac{\log 2}{\log \alpha} \) and \( m = n^{1+1/\lambda} \) then

\[ E(k) = \Omega \left( n^{1+1/\lambda} \right) \]
Applying lemma 5 on the above estimates will yield the proofs for the lower bounds set by theorems 1, 4, 6 and 7.

4.3 The parallel variant

The memory allocation scheme as used by $H^*$ is a little different here. Many hash functions can be applied in parallel to the same set. Let $\omega_{i,1}, \omega_{i,2}, \ldots$ be the cardinalities of ranges of those functions in step $i$. Let $\omega_i = \omega_1 + \omega_2 + \cdots$ be the total memory allocation for a certain set of size $a_i$. The probability that all those hash functions will be a complete failure is

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

This probability is minimized when all the $\omega_{i,j} = 2$ and in this case the failure probability is

$$2^{-\omega_i}.$$ 

Let $m = m_1 + m_2 + \cdots + m_k$ be a memory allocation of the $m$ cells to the $k_i$ sets. The expected value of $k_{i+1}$ is given in the parallel hashing variant.

$$E(k_{i+1}) = \sum_{j=1}^{k_i} m_j / m.$$ 

And again this is minimized when all the $m_j$ are equal. We concluded in

Lemma 7 In the parallel variant all hash functions ever used by $H^*$ are to a range of size 2. In the $k$th iteration $m/2k_i$ functions with a total range of size $m/k_i$ are applied to each of the $k_i$ sets of size 2.

We get that

$$E(k_{i+1}) = k_i 2^{-\omega_i}.$$ 

We set $a = \Theta(\log^2 n)$ (which is big enough for the chaining parallel variant too). Note that lemma 4 holds in this case too. Let $\omega_i = m/k_i$. We will be interested only in values of $\omega_i$ which are much smaller than $\log n$. For those values the probability that $k_{i+1}$ will be significantly small than its expected value is exponentially small. We get that

$$\omega_{i+1} \leq 4 \omega_i 2^{-\omega_i}.$$ 

This variant $H^*$ is required to $\omega_i > \log \log \log n$, but this will require $O(\log^2 n)$ steps. (which is sufficient parallel and the chaining parallel variant).

4.4 The retries variant

To complete this section we assume lemmas 2 and 3. Note that the retry, if a certain application of a hash function was a complete failure in a hash node, then doing a retry is idea of learning to leave with it, it will do again. However, our simplified dichotomous classification of a hash function is not a complete failure in a hash node, and then by doing a retry is idea of learning to leave with it, it will do the next iteration. Given if this internal node was not allowed $H^*$ to ignore it, and if we could do a retry can on the root.

Retries cannot help in the root of the tree 1, even $O(\log n)$ retries is not yield a significantly better value.

The above arguments could not lead to the observation that separation maintained from tree node, can only be done in the memory of the node. The equivalence of any retry to an algorithm without retries here because the $H^*$ could use the $H^*$ in a hash node, then by doing a retry is idea of learning to leave with it, it will do the next iteration. Given if this internal node was not allowed $H^*$ to ignore it, and if we could do a retry can on the root.

5 Proofs of upper bounds

The algorithm presented by [4] is the following way. In the root node is used to map all keys to a table defined as (as mod $p$) mod $a$ with a fixed large prime, and $a$ is in the range $[1, p - 1]$. This function is independent, and with high
satisfy
\[ \sum_{i=1}^{n} 2|A_i|^2 \leq 5n. \]

Thus \(2|A_i|^2\) memory cells can be assigned to the set \(A_i\) without exceeding the linear memory bound. A sequence of similar functions \(h_i(x)\) are then tried on the set \(A_i\), until a complete separation function is found.

This function is defined by
\[ h_i(x) = (a \cdot x \mod p) \mod (2|A_i|^2) \]
where \(p\) is as before and \(a\) is picked again at random from the interval \([1, p-1]\).

The probability that \(h_i(x)\) will not achieve complete separation is \(\Omega(1)\), and since the number of the sets is \(\Omega(n)\), the number of iterations will be \(\Omega(\log n)\).

5.1 The basic model and the retries variant

Before we can improve on that we need the following lemma which is an easy consequence of the basic lemma of [4].

Lemma 8: Let \(h_i(x)\) be a function of the type \((a \cdot x \mod p) \mod (q|A_i|^2)\) that maps the bucket \(A_i\) into the range \([1..q|A_i|^2]\). If \(h_i\) is picked at random from all the functions of its kind then the probability that \(h\) is a one-to-one function is at least \(1 - 1/g\).

Our improvement will be based on a different approach to memory allocation. The idea is to classify all sets according to the log of their size. Each class uses a portion of the memory which is proportional to the initial number of keys in the set. The total memory used by a class in an iteration decreases geometrically. That memory is equally divided to all living sets in the class. The details are given in the following algorithm: As before let \(h(x)\) be the initial count of sets of size exactly \(s\), and let \(h_i(x)\) be the count of "living" sets by iteration \(i\).

Algorithm QuickHash
begin
Find \(h\), an initial PKS function for which \(\sum_{s \leq 5} |A_i|^2 \leq 5n\)
Mark all sets \(A_i\) as generated by \(h\);
Classify all sets having \(s\) elements as having \(s = 2^{\log 27}\) elements.
Loop 1: foreach \(s\) s.t. there are marks
\[ f \leftarrow 0 \]
\[ s_t \leftarrow 16 \]
Loop 2: while there are still marks
\[ k(s) \leftarrow \# \text{marked sets of size } s \]
allocate a memory block of size \(k(s)\) for each set of size \(s\)
Loop 3: repeat
foreach living set \(X\) of size \(2^t\)
Pick a random number \(f\) from range \([1..27]\]
Apply \(h(x) = (a \cdot x \mod p) \mod (q|A_i|^2)\) to all members of \(X\)
If \(k(s)\) gives a one to one mapping, mark \(X\)
\[ f = f \]
mark \(X\)
until \# (marked sets of size \(s\))
\[ g_{t+1} = \frac{g_t}{4} \]
\[ f = f + 1 \]
end
end QuickHash

Time analysis: To analyze the time we modify it in the following way: if loop \(2\) the count of marked sets is required level then all the loop effect.
In this modification every iteration be modeled as a random variable (that was unmarked) sampling. The size of this random variable is at most \(1/3\) at the most.

In analysing loop 2 note that handling size clearly terminates before \(g_t > 1\) straightforwardly to verify that \(g_t\) into every assignment of the type \(g_{t+1} = g_t / 4\). We conclude that loop 2 is \(O(\log \log n)\) times. Incorporating the loop 3, we get that the expected number of loop 3 in loop 2 is \(O(\log \log n)\).

To analyze the outer loop, one account the fact that many copies are executed simultaneously in it. The p
one of the copies of loop2 will be executed more than its expected value can be estimated using Chernoff bounds. The probability that one instance of loop2 will take more than $C$ times its expected value is $o(1/\log n)$, and since there are at most $O(\log n)$ such instances, the probability that the slowest of them will terminate in time longer than $O(\log \log n)$ is $o(1)$.

Note that the initial phase can be done in $O(1)$ expected time too, but it may require (with a low probability) a retry. We do not allow retries in the root node, and if the initial hashing was not satisfactory then the algorithm will fail. This of course cannot change the expected behavior of the algorithm.

**Memory Analysis:** Total memory used is $O(n)$: In each iteration of loop2, $k_s n^2$ cells are allocated, since $k_{t+1}(s) \leq 2k_t(s)/s_t$ and $s_{t+1} = s_t^2/4$ we get that the memory usage in the $t+1$ iteration is at most $1/4$ of that of the $t$th iteration, thus showing that the total memory usage is dominated by the initial memory allocation. Extra memory beyond the FKS requirements is needed in the initial phase, because sets sizes can almost be doubled, when increased from $s'$ to $s$. Another initial memory demand comes from the first phase allocation of $16n^2$ cells to every set instead of $2n^2$. It is easy to see that the initial memory usage is still $O(n)$.

If retries are to be avoided, then our algorithm can be modified to allocate a new memory block in each iteration, in this case memory usage will still be linear, but it could not be predetermined.

### 5.2 The parallel variant

The above algorithm is not applicable directly to PRAM machines, but its ideas can be used to supply an actual PRAM algorithm [5]. In contrast to that the following algorithm seems much harder to implement. In each iteration of it—more and more processors are drafted to the hashing of fewer and fewer keys. Locating those in need and organizing help for them require (probably) time, and the algorithm ignores that.

The algorithm for the parallel variant is almost identical to the above. The main difference is that the $s_t n^2$ memory block allocated to a set of size $s$ in the $t$th iteration is divided further to subblocks of size $2a^2$, into which the pass is done. The failure probability in this case loop2 can be repeated until there is a "live set" of size $a$. The new $g_{t+1}$ is $2^{t+1/2-3}$ which gives rise to the $O(\log \log n)$ of the previous one. Further details of the proof are given in the previous one.

### 5.3 Chaining variant

Here we would like to use the previous in order to bring the hash cost to $O(\log \log n/\log \log \log n)$. Let

$$r = C \frac{\log \log n}{\log \log \log n}$$

for some constant $C$. The algorithm with an FKS-like function with $\sum_{i \leq n} |A_i|^2 \leq 5n$, and then hashing only on sets of size $s > r$.

The goal of this hashing will be so that $A_i$ will be divided into subsets of size $s$ elements, instead of the usual attempt to achieve complete separation.

When all sets are of size $s \leq r$ they are just by the ability of storing 1 element per internal node, adding at most $\log n$ levels.

Let $H$ be a mapping from a fixed memory of size $gs^2$. We say that $H$ is more of the subsets (buckets) it overflows if $ class $H$ is a $g$ probability that $A$ picked is random, is at most $s^{-r}$. This property is linear in random functions, as can be seen from $\log$.

The outline of algorithm QuickHash is too for the set breaking. In the next loop2, loop3 is executed at most $2k_2(s)g^2 \leq 2k_1(s)g^2$. Thus

$$k_{t+1}(s) \leq 2k_{t+1}(s)/s_t^2$$

If we require

$$k_{t+1}(s)g_{t+1} \leq k_1(s)g$$

we will ensure that the total memory is linear. We get $g_{t+1} = g_t^2/4$. Reg for $r$ and solving the recurrence $O(\log \log n/\log \log \log n)$ iterations
\(O(n)\) and all the sets will be broken. As we said random functions are good breakers. However, true random functions are not useful for hashing as they require huge space for representation. If instead \(H\) will be the class of \(r - 1\) degree polynomials then \(H\) is a good breaker and it can be represented efficiently. However in this case each application of a hash function takes \(O(r)\), and hence the total run time will not be reduced below \(O(\log \log n)\) time (although the number of hash function application is still \(O(\log \log n / \log \log \log n)\)).

Recently, Dietzfelbinger and Meyer auf der Heide suggested in [1,2] a class \(DM\) of hash function that is a good breaker, which can be evaluated in \(O(1)\) time and represented in \(O(n)\) memory. Implementation of their function, is not included in our basic model or its variants, since merging of tree nodes is used there. In general, merging of nodes in the hash tree can only increase separation time. Merging in [1] is only used to increase randomness, which is implicitly assumed in our model.

References


