Maintaining External Memory Efficient Hash Tables
(Extended Abstract)*

Philipp Woelfel

Univ. of Toronto, Dept. of Computer Science, Toronto, ON M5S3G4.
philipp.woelfel@utoronto.ca

Abstract. In typical applications of hashing algorithms the amount of data to be stored is often too large to fit into internal memory. In this case it is desirable to find the data with as few as possible non-consecutive or at least non-oblivious probes into external memory. Extending a static scheme of Pagh [11] we obtain new randomized algorithms for maintaining hash tables, where a hash function can be evaluated in constant time and by probing only one external memory cell or $O(1)$ consecutive external memory cells. We describe a dynamic version of Pagh's hashing scheme achieving 100% table utilization but requiring $(2+\epsilon)\cdot n \log n$ space for the hash function encoding as well as $(3+\epsilon)\cdot n \log n$ space for the auxiliary data structure. Update operations are possible in expected constant amortized time. Then we show how to reduce the space for the hash function encoding and the auxiliary data structure to $O(n \log\log n)$. We achieve 100% utilization in the static version (and thus a minimal perfect hash function) and $1-\epsilon$ utilization in the dynamic case.

1 Introduction

In this paper, we devise randomized algorithms for efficiently maintaining hash tables under circumstances typical for applications dealing with massive data. Consider a set S of n keys from a finite universe U and assume that each key $x \in S$ has some data D_x associated with it. A static dictionary for S supports a query operation which returns the data D_x for a given key x. A dynamic dictionary also support update operations which allow to insert new data into the dictionary or to remove data from it. For many applications it is especially important to be able to retrieve the data D_x as quickly as possible (examples are databases used by web-servers, where a huge amount of queries have to be answered in short time). A typical solution is to maintain a hash function h mapping each key $x \in S$ to an entry of a hash table T. Such a hash function h is called *perfect for* S if it is injective on S. If h has range $[n] := \{0, \ldots, n-1\}$, $n = |S|$, then h is called *minimal perfect*. If h is perfect on S, the data associated with each key in S can be stored in $T[h(x)]$. We call such an implementation of a dictionary a *hash table implementation based on perfect hashing*. An algorithm

* The research was supported by DFG grant Wo 1232/1-1.

dynamically maintaining a perfect hash function is called *stable* if $h(x)$ remains fixed for the duration that x is in S.

The following assumptions are typical for many dictionary and hashing applications: Firstly, efficiency is much more a concern for lookups than for update operations. For example, in database backends of webservers a huge number of queries have to be answered momentarily while updates of the database only rarely occur or sometimes even can be delayed to times of low load.

Secondly, the data set is so massive that even the description of the hash function does not fit into the internal memory. For example, the encoding of a minimal perfect hash function requires at least $\Omega(n) + \log\log|U| - O(\log n)$ bits, assuming that $|U| \geq n^{2+\epsilon}$ [8]. In this case, just in order to evaluate the hash function we have to access external memory. But then usually the number of non-consecutive accesses to external memory dominate the evaluation time of our hash function.

Finally, the data D_x associated with a key x requires much more space than its key x. Therefore it is especially important that a hash table implementation achieves a high utilization, since we have to reserve a fixed amount of space for each table entry (if we want to avoid another level of indirection). Assuming that the hash table is implemented by an extendible array $T[0], T[1], \ldots$, its utilization is given as $|S|/(\max\{h(S)\} + 1)$. In particular, even a small constant utilization seems infeasible, and a utilization as close to 100% as possible should be achieved. A minimal perfect hash function for the set S achieves 100% utilization.

Although general dictionary implementations (not necessarily based on perfect hashing) can be used to maintain minimal perfect hash functions by associating each key $x \in S$ with a unique value from $[n]$, such solutions require another level of indirection.

Previous and Related Work. Throughout this paper we assume that $|U| = n^{O(1)}$. It is well-known how to reduce the size of the universe by choosing a random hash function $\zeta : U \to [n^{O(1)}]$ such that ζ is injective on S with high probability. Moreover, we assume that the size of the internal memory is bounded by n^ϵ, $\epsilon < 1$.

Fredman, Komlós, and Szemerédi [9] were the first to devise an algorithm which constructs a perfect hash function (with $O(n \log n)$ bits) in expected linear time such that the hash function can be evaluated in constant time. The utilization is less than 0.2 in the case that only consecutive probes into external memory are allowed for hash function evaluation. Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan [5] have devised a dynamic version of that scheme with essentially the same parameters, but which also supports updates in expected amortized constant time. Later improvements have either focused on reducing the space requirements or on obtaining a constant update time even with high probability. All schemes which do in fact achieve a constant update time with high probability are mostly of complexity theoretical interest (as opposed to practical). Demaine, Meyer auf der Heide, Pagh, and Patrăscu [1] show an upper space bound of essentially $O(n \log\log(u/n) + n \log n - n \log t)$ for maintaining a perfect hash function with range $[n+t]$ and $O(n \log\log(u/n) + n \cdot r)$

for a dynamic dictionary where the data associated with each key comprises r bits. Update operations are supported with high probability in constant time and the algorithm is stable. For the static case, Hagerup and Tholey [10] hold the space record: They show how to construct a minimal perfect hash function in expected $O(n + \log\log|U|)$ time such that its encoding requires only almost optimal $(1 + o(1))(n \cdot \log e + \log\log|U|)$ space. Multiple non-oblivious probes into external memory are required for lookups in these space efficient dynamic or static schemes.

Dictionary algorithms such as Cuckoo-Hashing [12] and its extensions [7, 6] also allow the retrieval of data with few non-consecutive probes into external memory. Especially space and external memory efficient is the Cuckoo-Hashing variant of Dietzfelbinger and Weidling [6], where two hash functions h_1 and h_2 and two hash tables T_1 and T_2 are used. A table position consists of d consecutive memory cells, and the data D_x is stored in one of the $2 \cdot d$ memory cells from $T_1[h_1(x)]$ and $T_2[h_2(x)]$. For $d \geq 90 \cdot \ln(1/\epsilon)$ a utilization of $1-\epsilon$ can be achieved and clearly the data can be retrieved with only two non-consecutive probes into external memory. Due to the large constant for d, it may be disadvantageous if the data associated with the keys is very large (now the time for finding D_x depends on the size of the data). Moreover, such dictionary solutions do not provide perfect hash functions without an additional level of indirection.

Pagh [11] showed how to construct a minimal perfect hash function in expected linear time which can be very efficiently evaluated with very simple arithmetics (essentially one or two multiplications) and by probing only one word from external memory. The hash function itself can be encoded in $(2 + \epsilon) \cdot n \cdot \log n$ bits. Dietzfelbinger and Hagerup [3] improved Pagh's scheme so that the resulting hash function can be encoded with $(1 + \epsilon) \cdot n \cdot \log n$ bits. Both schemes yield a static dictionary with 100% utilization.

In this paper we devise a dynamic variant of Pagh's scheme. Maintaining 100% utilization and using exactly the same hash functions, we show how to perform updates in expected amortized constant time. I.e., the hash functions can be evaluated very efficiently in constant time and with only one probe into external memory. In addition to the $(2+\epsilon) \cdot n \cdot \log n$ bits for encoding of the hash function we also need an auxiliary data structure comprising $(3 + \epsilon) \cdot n \cdot \log n$ bit. However, this auxiliary data structure is only needed for update operations and not for lookups. For many applications updates occur infrequently, e.g., at night time, so that the auxiliary data structure may be swapped out (or it can be removed and later be rebuild from scratch in expected linear time if needed). We believe that this scheme is quite practical if the main focus is on lookup performance, although the algorithm for updates is not very simple.

In Section 3 we investigate how much the space for the hash function description can be reduced under the constraint that evaluation requires only consecutive probes into external memory. We show that it is possible to reduce the encoding size of the hash functions and the space for the auxiliary data structure to $O(n \log\log n)$ bits. In the dynamic case we obtain a utilization of $1 - \epsilon$, for arbitrary small $\epsilon > 0$. In the static case we still achieve 100% utilization, hence

we even have a minimal perfect hash function. For both implicit versions the corresponding hash functions can be evaluated in constant time and by probing $O(1)$ consecutive words from external memory. (Here $O(1)$ is a very small constant, e.g. 4). The hash functions itself are a little bit more complicated – their evaluation times are dominated by the arithmetics involved for evaluating two polynomials of small constant degree.

Our dynamic hashing algorithms are not stable. For updates we have to assume that the key x can be retrieved from the hash table entry $T[h(x)]$. But the hash function description itself is independent from the table contents.

2 The Displacement Scheme

As explained before, we assume throughout the paper that $U = [n^{O(1)}]$. Moreover, we assume a word-RAM model where every key in U fits into a memory word (i.e., we have a word-size of $\Omega(\log n)$).

Let $\mathcal{H}_a$ be a family of hash functions $h : U \to [a]$. $\mathcal{H}_a$ is *c-universal* if for any $x, x' \in U$, $x \neq x'$ and for randomly chosen $h \in \mathcal{H}_a$, it holds $Prob\big(h(x) = h(x')\big) \leq c/a$. If $\mathcal{H}_a$ is c-universal for some arbitrary constant c, then we call it *approximately universal*. $\mathcal{H}_a$ is *uniform* if $h(x)$ is uniformly distributed over $[a]$ for all $x \in U$.

Examples of efficient 1- and 2-universal and uniform hash families can be found in [2, 4, 13]. For our purposes it suffices to know that most hash functions from c-universal hash families can be evaluated in constant time with a few arithmetic operations (usually dominated by one multiplication and a division) and that they can be encoded in $O(\log |U|)$ or even $\mathrm{O}(\log n + \log\log |U|)$ bits.

Pagh [11] showed how to construct minimal perfect hash functions $h_{g,f,d}$ defined in the following. Let a and b be positive integers and suppose that S is a set of n keys from the universe U. Let $f : U \to [a]$, $g : U \to [b]$ and $d = (d_0, \ldots, d_{b-1}) \in [a]^b$. Then $h_{g,f,d} : U \to [a]$ is defined by $x \mapsto \big(f(x) + d_{g(x)}(x)\big) \bmod a$.

One can visualize the hash function $h_{f,g,d}$ by a $(b \times a)$-matrix M, where the ith row, $i \in [b]$, is associated with the *displacement* value d_i. In order to evaluate h for an element $x \in U$, one first maps x into the matrix element in row $g(x)$ and column $f(x)$. Then the row is rotated cyclically $d_{g(x)}$ steps to the right, where $d_{g(x)}$ is the displacement associated with this row. We call two row displacements d_i, d_j, $i \neq j$, *compatible* (with respect to g, f and S), if for all $x \in g^{-1}(i) \cap S$ and $x' \in g^{-1}(j) \cap S$ it holds $f(x) + d_i \neq f(x') + d_j$. Clearly, $h_{f,g,d}$ is injective on S if and only if all row displacements are pairwise compatible. According to the informal description above, $S \cap g^{-1}(i)$ is the set of elements which are mapped into the ith row of the matrix M. We call $\big|S \cap g^{-1}(i)\big|$ the *weight* of row i. In order to construct the minimal perfect hash functions, Pagh used the following notion.

Definition 1. *Let $S \subseteq U$ and $f : U \to [a]$, $g : U \to [b]$, and $w_i = \big|g^{-1}(i) \cap S\big|$ for $i \in [b]$. The pair (f, g) is* δ-nice *for S if the function $x \mapsto (f(x), g(x))$ is injective on S, and $\sum_{i, w_i > 1} w_i^2 \leq \delta \cdot a$.*

Note that what we call δ-nice was originally denoted as r-good, where $r = \delta \cdot a/n$; we chose a different notion because it is more convenient for our purposes. Pagh has shown that a pair (f, g) being δ-nice for a set S, $\delta < 1$, suffices to find a displacement vector d such that $h_{f,g,d}$ is injective on S. On the other hand, δ-nice hash functions can be found easily using universal hash families.

Lemma 1 (Pagh [11]). *Let H_a be c_f-universal and H_b be c_g-universal. If $2 \cdot c_g \cdot n^2/(a \cdot b) \leq \delta \leq 4/c_f$, then for any n-element set $S \subseteq U$, the probability that a randomly chosen pair $(f, g) \in H_a \times H_b$ is δ-nice for S is more than $\Big(1 - \delta \cdot c_f/4\Big) \cdot \Big(1 - 2 \cdot c_g \cdot n^2/(a \cdot \delta \cdot b)\Big)$.*

For instance, let $a = n$ and $b = (2 + \epsilon_b)n$, $\epsilon_b > 0$. If H_a is 4-universal and H_b is 1-universal, then there exists a $\delta < 1$ such that (f, g) is δ-nice with positive constant probability. We refer the reader to [11] for more details on suitable hash families, or on constructions where $f(x)$ and $g(x)$ can be determined with essentially one multiplication if b is chosen only slightly larger.

Consider again the matrix M as described above. If (f, g) is δ-nice for S, all elements in S are mapped to disjoint matrix elements (for any displacement vector d). Pagh's algorithm finds in expected linear time a vector d such that all displacements are compatible. The row displacements are chosen randomly one after the other in an order with decreasing row weights. This order and the δ-niceness guarantee that a compatible displacement can be found for the next row to be processed. Clearly, such an ordering of the rows cannot be used for a dynamic algorithm. Our idea is the following: If an insertion yields incompatible displacements, then we randomly choose one such displacement anew. This new displacement may now be incompatible with other displacements, but after a constant number of tries, the total number of elements in rows with weight larger than one and with incompatible displacements decreases by a constant factor. Rows with weight are taken care of in the end – for them new compatible displacements can be found easily by keeping track of empty table cells.

Consider a hash table $T[0], \ldots, T[N-1]$ for a set $S \subseteq U$ of at most N elements. We first consider a fixed value of N and later show how to adapt if $|S|$ exceeds N. We store a perfect hash function $h := h_{f,g,d} : U \to [N]$ and an auxiliary data structure. Every element $x \in S$ is stored in $T[h(x)]$. If $|S| = N$, then h is minimal perfect. Since f and g can be stored in $O(\log n)$ bits, one probe into external memory (to retrieve $d_{g(x)}$) suffices for computing $h(x)$.

The functions f and g as well as the displacements can be chosen from the same sets H_a and H_b as in the static case. Fix some $\delta < 1$ such that (f, g) is δ-nice with constant probability. Throughout the description of the algorithm let $w_i = \big|S \cap g^{-1}(i)\big|$, if S and g are clear from the context.

For insertions and deletions we need an auxiliary data structure consisting of b linked lists $L_0, \ldots, L_{b-1}$. The list L_i contains all table positions $h(x)$ for which there is an element $x \in S \cap g^{-1}(i)$ (i.e., the columns j in the matrix to which all elements of the row i are mapped to). We don't need to store row weights w_i since we can compute them by searching through the lists L_i, but we store the sum $W = \sum_{i \in [b]} {w_i}^2$. Finally, we use a data structure for storing all empty table

cells, i.e., the indices j such that $T[j] = \infty$. We use a function `free_pos` which returns in expected constant time the index of an arbitrary empty table cell (if there is one). The implementation of such a data structure comprising $\epsilon \cdot n \cdot \log n$ space is easy – a description can be found in the full version of the paper.

Update Operations. In order to delete an element it suffices to set $T[h(x)]$ to ∞ and to remove x from the list L_i, where $i = g(x)$. This requires $O(w_i)$ time. Since g is chosen from an approximately universal hash family H_b, the expectation of $w_{g(x)}$ is $O\big(n/b\big) = O(1)$.

Assume that $h_{f,g,d}$ is injective on S and that we want to insert a new element $x \notin S$. Let $S' = S \cup \{x\}$. Using the list L_i, it is easy to update the sum of squared row weights to $W' = W - w_i^2 + (w_i + 1)^2$ if $w_i > 0$ and $W' = W$ if $w_i = 0$. Using this list we can also check whether (f, g) is still δ-nice for S'. All this can be done in $O(w_i + 1)$ time. If (f, g) is not δ-nice for S', we have to perform a global rehash, i.e., we have to remove all keys and insert them again with a new randomly chosen pair (f, g).

Now assume that (f, g) is δ-nice for S'. If $T[j]$ is empty for $j = h_{g,f,d}(x)$, we simply store x in $T[j]$ and insert j into the lists L_i. If $T[j]$ is already occupied, then we have to determine new displacement values for some rows. For that we maintain a set Q of *possibly bad* rows such that all rows in $[b] - Q$ are compatible. With each row $i \in Q$ we also store the set $V_i := S' \cap g^{-1}(i)$ of all keys which are hashed to that row. Consequently we remove every element $x \in V_i$ from the hash table at the time we insert it in V_i. Initially, Q contains only row i and we collect the set V_i with the help of the list L_i. Let $w(Q) = \sum_{i \in Q, w_i > 1} w_i$.

We repeat the following procedure until $w(Q) = 0$. First we pick an arbitrary row i in Q where $w_i > 1$ and choose a new displacement d_i'. We now define a condition in which we accept this new displacement.

Definition 2. *Fix a set $S \subseteq U$ and a hash function $h_{f,g,d}$ and let d_i' be a new displacement of row $i \in Q$. The set $J(d_i', i)$ contains the indices of all rows $j \in [b] - Q$ such that row i and row j are not compatible. The displacement d_i' is* acceptable *if $\sum_{j \in J(d_i'), w_j > 1} w_j \;<\; w_i \cdot (1 + \delta)/2$.*

Clearly, we can determine the set $J(d_i', i)$ in time $O(w_i)$. By seeking (at least partly) through the lists L_j with $j \in J(d_i', i)$ we can also check in time $O(w_i)$ whether d_i' is acceptable. Proposition 1 below states that with constant probability a randomly chosen displacement is acceptable. Hence, in expected time $O(w_i)$ we can find an acceptable new displacement d_i'.

Then we change Q to $Q' = J(d_i', i) \cup Q - \{i\}$, create the sets V_j, $j \in J(d_i', i)$, and accordingly remove the elements in V_j from the hash table. Finally we store all elements $x \in V_i$ at their designted places $T[h_{f,g,d'}(x)]$ (since row i is now compatible with all rows not in Q', these table positions are not occupied).

Repeating this procedure eventually leads to a set Q^* with $w(Q^*) = 0$ (see the time analysis below). Hence, Q^* consists only of possibly bad rows with weight 1. For these rows it is easy to find new compatible displacement values using the function `free_pos`. After that, the resulting hash function h_{f,g,d^*} is injective on S' and all elements in S' are stored in their designated table positions.

Time Analysis. We first consider the case that the hash function pair (f, g) is still δ-nice for S' and that no global rehash is necessary. The following proposition shows that we can quickly find an acceptable displacement for each row. The (straight forward) proof has to be omitted due to space restrictions.

Proposition 1. *Let $(f, g) \in H_a \times H_b$ be δ-nice, $\delta < 1$, for some n-element set S. With positive constant probability a randomly chosen displacement is acceptable.*

Hence, the total expected time for finding an acceptable displacement for a row $i \in Q$ is $O(w_i)$. Then in expected time $O(w_i)$ we can decrease the value $w(Q)$ to a value of $w(Q') \leq w(Q) - w_i + w_i \cdot (1+\delta)/2 = w(Q) - \Omega(w_i)$ (using $\delta < 1$). It follows from the linearity of expectation that in order to obtain a set Q^* with $w(Q^*) = 0$ expected time $O\big(w(Q)\big)$ suffices. Now recall that we started with $Q = \{i\}$, where $i = g(x)$ and where x was the element we inserted. Hence, the total expected time until the resulting set Q^* contains only rows with weight 1 is $O(w_i)$. Since we can collect only $O(t)$ rows with weight 1 in time t, the total number of rows in Q^* is also $O(t)$. By the assumption that the operation `free_pos` can be executed in expected constant time, we can redisplace these rows in expected time $O(t)$. To conclude, the total expected time for inserting an element x is $O(w_{g(x)})$. As argued in the section about deletions, $E\big(w_{g(x)}\big) = O(1)$.

We have shown so far that we obtain an expected constant insertion time as long as (f, g) remains δ-nice for the resulting set. By Lemma 1, a simple calculation shows that a randomly chosen set pair (f, g) is with constant probability δ-nice even for the $\lfloor \alpha N \rfloor$ sets obtained from some set S of size N during a sequence of $\lfloor \alpha N \rfloor$ update operations, for some sufficiently small $\alpha > 0$. Therefore, during $\lfloor \alpha \cdot N \rfloor$ update operations we expect only a constant number of global rehashes and thus the expectation of the amortized update time is constant.

A Dynamic Hash Table with 100% Utilization. So far we can insert and delete elements from a hash table of size N, as long as $|S| \leq N$. We now sketch an algorithm which maintains a dynamic hash table T where all n elements in S are stored in the table positions $T[0], \ldots, T[n-1]$ at all times. A complete description will be given in the full version of the paper.

The problem is mainly with deletions. If an arbitrary element is deleted, a "hole" is left behind in the middle of the hash table, say at position i. But in order to store all remaining $n-1$ elements in the table positions $T[0], \ldots, T[n-2]$, we have to move the element x from $T[n-1]$ to some other position. Since it is not clear how to bound the weight of the row $g(x)$ of that element, we don't know how to obtain an expected constant deletion time. The idea is now to ensure that the last $\gamma \cdot n$ entries of the table, $\gamma > 0$, are filled with elements from rows with weight one. Then we can easily choose a new displacement for the corresponding rows, so that the element in $T[n-1]$ moves into the hole $T[i]$.

We now interpret the displacements of the hash function $h_{f,g,d}$ differently: Let $i = g(x)$. Then $h_{f,g,d}(x) = (f(x) + d_i) \bmod a$ if $d_i < a$, and $h_{f,g,d}(x) = d_i$ if $d_i \geq a$. This way, the range of $h_{f,g,d}$ is not limited to $[a]$.

Consider a situation right before a rehash. Let S be the n-element set currently stored and let S_k, $k = 0, 1, 2, \ldots$ be the set obtained after the next k

operations (i.e., $S_0 = S$). We let $a = (1-\gamma)n$ for some sufficiently small $\gamma > 0$, and b, H_a, and H_b as before. Now we have to perform a global rehash also if the size of the set S drops below a.

In order to insert a new element x in a set S of size $n \geq a$, we redisplace rows exactly as before, but using only displacement values $d_i < a$ for rows with weight $w_i > 1$. As before we end up with a set Q^* with $w(Q^*) = 0$, i.e., $w_i = 1$ for all rows $i \in Q^*$. Now for one of the remaining rows in Q^* we choose the displacement value d_i such that the unique element x^* in that row obtains a hash value of n and we store x^* in $T[n]$. All other displacement values for rows with weight 1 can be determined using `free_pos` as before.

The insertion procedure guarantees that displacement values $d_i \geq a$ are only used for rows with weight $w_i = 1$. Hence, as long as $n = |S| > a$, the element x^* stored in $T[n-1]$ belongs to a row i^* with weight $w_{i^*} = 1$. Hence, in order to delete an element x from S, $|S| > a$, we simply change d_{i^*} to a value such that x^* moves into the table cell $T[h_{f,g,d}(x)]$, formerly occupied by x.

Theorem 1. *For any $\epsilon > 0$ a dynamic hash table with 100% utilization can be maintained with constant amortized update time and $(2+\epsilon)n \log n$ space for the hash function encoding and $(3+\epsilon)n \log n$ space for the auxiliary data structure. The hash function can be evaluated in constant time and with only one probe into external memory.*

Corrupted Hash Table Cells. For the following sections we need to consider a variant of the above scheme, which may also be of independent interest. Consider a hash table $T[0], \ldots, T[n+k-1]$ with k *corrupted* cells. If a cell $T[i]$ is corrupted, then none of the keys in S may be stored there, but we assume that we can check in constant time whether a cell $T[i]$ is corrupted or not. Let $I \subseteq \{0, \ldots, n+k-1\}$, $|I| = k$, be the set of indices of corrupted table cells. For $k = o(\sqrt{n})$, we can modify our data structure in such a way that an n-element set S is stored in the hash table T without using any corrupted cells. If a new element is inserted we use the same algorithm as above, except that when we choose a new displacement d_i' for a row i we have to ensure that none of the keys from that row are hashed to a corrupted cell. Thus, for every n-element set S we can maintain a hash function $h := h_{f,g,d}$ which is injective on S and where $h(S) = \{0, \ldots, n+k-1\} - I$, and with the same time- and space-complexity as in Theorem 1.

3 Implicit Hash Functions

We now show how to reduce the space of our hash functions significantly. Recall that we assume a word-size of $\Omega(\log n)$. Similar as in [10] we use one additional hash function $\hat{h}$ in order to split the n-element set S into small groups.

For the following we need two functions $\mu(n) = (\log n)/K$ and $\lambda(n) = n/(\log n)^K$ for some large enough constant K. Let $\hat{h} : U \to [\hat{a}]$, $\hat{a} \in \mathbb{N}$, and let $S \subseteq U$ be an n-element set. We call a group $G_i := S \cap \hat{h}^{-1}(i)$, $i \in [\hat{a}]$, *c-small*

if $|G_i| \leq \log n/(c \cdot \log\log n)$. If G_i is not c-small, then it is *c-large*. The hash function $\hat{h}$ is *c-good* for S, if all groups have a size of at most $\mu(n)$ and if the total number of c-large groups is at most $\lambda(n)$

Let $\hat{b} = \lfloor n^\gamma \rfloor$ and $\hat{a} = \lceil Z \cdot n \cdot \log\log n/\log n \rceil$. We use the polynomial hash families $\mathcal{H}_s^k$ described in [10]. For a prime $p > |U|$ and $a \in [p]^{k+1}$ the hash function $r_a : U \to [s]$ is given as $x \mapsto \left(\sum_{i=0}^{k} a_i \cdot x^i \bmod p\right) \bmod s$. The hash family $\mathcal{H}_s^k$ consists of all hash functions r_a, $a \in [p]^{k+1}$.

Lemma 2. *For any n-element set S, any integer $c > 1$, any $\hat{b} = n^\gamma$, $\gamma > 0$, and any $\hat{a} = \lceil Z \cdot n \cdot \log\log n/\log n \rceil$, $Z > c$, there exist k_a, k_b such that for a randomly chosen pair $(f, g) \in \mathcal{H}_{\hat{a}}^{k_a} \times \mathcal{H}_{\hat{b}}^{k_b}$ and a randomly chosen vector $\hat{d} \in [\hat{a}]^{[\hat{b}]}$ the following is true:*

1. *With probability $1 - o(1)$ a random hash function $\hat{h} = \hat{h}_{\hat{f},\hat{g},\hat{d}}$ is c-good for S.*
2. *For every element $x \in S$, the probability that x is in a c-large group is $2^{-\Omega(\log n/\log\log n)}$.*

The idea of that proof, which we have to omit due to space restrictions, is very similar to a proof in [10], Lemma 3. The main difference is here that our expected group sizes are smaller by a $\log\log n$ factor and we therefore can only achieve that most groups instead of all groups deviate little from their expectation.

The Implicit Data Structure. We now sketch the dynamic scheme which achieves $1 - \epsilon$ utilization, $\epsilon > 0$, but requires only $O(n \cdot \log\log n)$ space for the hash function encoding and the auxiliary data structure. We choose $Z > c > 1$, $Z' = (1 + \alpha)Z$ for some arbitrary small $\alpha > 0$, and $\hat{a} = \lceil Z' \cdot n \cdot \log\log n/\log n \rceil$. As just described we use a hash function $\hat{h} : U \to [\hat{a}]$ in order to split the set S into groups. Consider a subsequence of operations between two global rehashes, i.e., during the time the hash function $\hat{h}$ remains c-good. The hash table T is split up in hash tables $T_0, \ldots, T_{\hat{a}-1}$ as well as one hash table T'. The tables T_i, $i \in [\hat{a}]$, are of size $t = \lfloor \log n/(c \cdot \log\log n) \rfloor$, and T' is of size $a' = O(n/\log n)$ (the constant factor can be chosen arbitrarily). In the following we call a group G_i *clean*, if it has been c-small since $\hat{h}$ was chosen the last time. At the moment a group G_i becomes c-large it is *dirty* and remains so until the next global rehash, even if it becomes c-small again before that. The idea is that all elements from a clean group G_i, $i \in [\hat{a}]$, are stored in the corresponding hash table T_i. For all bad groups the one larger hash table T' will suffice.

If after an insertion the function $\hat{h}$ is not c-good anymore, it has to be chosen anew (which triggers a global rehash). However, it can be shown that with constant probability $\hat{h}$ remains c-good during any sequence of $\lfloor \alpha n \rfloor$ update operations. Therefore we just discuss the update operations under the assumption that no global rehashes occur.

For each element $x \in S$ its group i is determined by $i = \hat{h}(x)$. With each group G_i we keep track of the number of its elements and store a bit indicating whether it is bad or not. If the group is clean, then it is also c-small and we can

use the dynamic scheme as described in the previous section. It is easy to see that if we choose c as a large enough constant, then we can store all displacements and the auxiliary data structure in one word of size $\Omega(\log n)$. Thus, all the information for one group G_i can be stored in $O(1)$ words.

If the group G_i is bad, then we use instead one hash function h_i from an approximately universal and uniform hash family $H_{a'}$, where $a' = \lceil n/\log n \rceil$. An element $x \in G_i$ is now stored in the table position in $T'[h_i(x)]$. As $O(\log n)$ bits suffice for storing h_i, we can store the hash function information for each group in a constant number of words. We also maintain a list L'_i containing pointers to the table positions in T' for all elements in group G_i. This is the auxiliary data structure for a bad group G_i. Since all lists for bad groups contain only $O(n/\log n)$ elements altogether, linear space suffices for all of them.

It is not hard to see that the total space for storing all hash functions and the auxiliary data structures is $O(n \cdot \log\log n)$ and that in order to evaluate the hash function it suffices to read a constant number of consecutive memory cells from a data structure with more than n^ϵ space.

Insertions and Deletions. Between two global rehashes we know that in each clean group G_i there are at most t elements, and thus we can insert and delete just as described in Section 2. We now discuss updates for elements hashed into bad groups by $\hat{h}$.

Let S' be the set of elements in bad groups and assume that a newly inserted element x is mapped by $\hat{h}$ to a bad group G_i. Since $\hat{h}$ is c-good we know that $n' := |S'| \leq \lambda(n) = n/(\log n)^K$. The designated table entry for x is $T'[h_i(x)]$. Since h_i is chosen from an approximately universal and uniform hash family $H_{a'}$, the probability that this table position is already occupied by an element in S' is at most $n'/a' = O\big((\log n)^{1-K}\big)$. If that table position is already occupied we randomly choose h_i from the universal hash family $H_{a'}$ anew. We use the list L'_i to collect all elements from G_i and rehash them again using the new hash function. The probability that one of the $O(\log n)$ elements in group G_i is mapped by h_i to one of the already occupied table cells in T' or that two of the elements in the group collide is at most $n'/a' + |G_i|^2/a' = O\big((\log n)^{1-K}\big)$. Such a rehash requires $|G_i| = O(\log n)$ time if it is successfull, and thus x can be inserted in expected $O(\log n)$ time, given that a rehash is necessary. On the other hand, as we have seen, with probability $1 - O\big((\log n)^{1-K}\big)$ the element x can be inserted without any rehash. Thus, for large enough K, x can be inserted in constant expected time given that it is hashed by $\hat{h}$ to a bad group.

We still have to discuss the transition from clean to dirty groups, though: If we insert a new element x into a clean group G_i, then this group may become dirty. In this case we have to move all elements from T_i to T' using a newly sampled hash function h_i (i.e., we rehash group G_i in expected $O(\log n)$ time). By the bound from part two of Lemma 2 on the probability that element x is in a bad group, the total expected time for inserting x is still constant in this case.

In order to delete an element x from a bad group we may simply set $T'[h'(x)]$ to ∞. Hence, we can delete elements in bad groups in worst-case constant time.

Theorem 2. *For any $\epsilon, \epsilon' > 0$ a dynamic hash table with $1 - \epsilon$ utilization can be maintained with constant amortized update time and $O(n \log\log n)$ space for the hash function encoding and the auxiliary data structure. The hash function can be evaluated in constant time and by probing $O(1)$ consecutive words from external memory (if the internal memory has size $n^{\epsilon'}$).*

Minimal Perfect Hashing with Implicit Hash Functions. We finally sketch an algorithm which constructs a minimal perfect hash function h in expected linear time such that the encoding of h requires only $O(n \log\log n)$ space and that h can be evaluated with only a few consecutive probes into external memory. The idea is again to use a hash function $\hat{h}$ to split the set S into $\hat{a}$ groups, but now we can use the fact that the group sizes do not change.

Let $S \subseteq U$ be a fixed n-element set. We will store all elements from S in a table $T = T[0], \ldots, T[n-1]$. As in the previous section we choose an integer c, a value $Z > c$, and let $\hat{a} = \lceil Z \cdot n \cdot \log\log n \rceil$. By Lemma 2 it is obvious how to find in $O(n)$ expected time a c-good hash function $\hat{h}$. Let $G_i = S \cap \hat{h}^{-1}(i)$, $i \in [\hat{a}]$.

We first process all c-large groups, one after the other. When we process a c-large group i, we create a hash function $h_i : U \to [n]$ mapping all elements in G_i to non-occupied table positions. The hash function h_i is a mapping $x \mapsto \lfloor \log n \rfloor \cdot h_i^*(x)$, where $h_i^* : U \to [\lfloor n / \log n \rfloor]$ is chosen from an approximately universal and uniform hash family $H_{\lfloor n/\log n \rfloor}$. We randomly sample such a h_i^* and then try to store each element $x \in G_i$ in the table position $T[h_i(x)]$. If that table cell is already occupied, we have to sample h_i anew. By arguments similar to those used in the dynamic case, the expected number of tries for each hash function h_i is only constant. Therefore, we can find in $O(n)$ expected time all hash functions h_i for c-large groups such that they map the elements from these groups to disjoint table positions.

Once we have found all hash functions h_i for the c-large groups, some of the table positions in T are occupied, which causes some interference with the c-small groups. That is where the notion of corrupted table cells (see Section 2) comes in handy. From now on we assume that every table cell $T[i]$ is corrupted, if one of the elements from a c-large group is stored there. Since we obtained each hash value $h_i(x)$ for an element x in a c-large group by multiplying a hash value $h_i^*(x)$ with $\lfloor \log n \rfloor$, we know that any $\lfloor \log n \rfloor$-sized interval of table cells, $T[i], \ldots, T[i + \lfloor \log n \rfloor]$, contains at most one corrupted cell.

We now process all c-small groups in increasing order. As in the dynamic case we find a hash function $h_i = h_{f_i,g_i,d_i}$ for each c-small group, mapping the elements of that group to a subtable T_i. We keep track of an offset o_i for each group i, indicating at which position in T the subtable T_i starts. Let a_i be the number of table cells we need for the ith group (this may be one more than the number of elements stored there, in the case that one table cell is corrupted). Then we can construct a hash function $h_i = o_i + h_{f_i,g_i,d_i}$ with the obvious random choices for f_i, g_i and d_i, which maps G_i injectively to the table positions $T[o_i], \ldots, T[o_i + a_i - 1]$ and spares out the corrupted table cell (if there is any). As we have seen in Section 2, h_i can be constructed even dynamically in expected constant time for each insertion and for $o(\sqrt{a_i})$ corrupted table cells.

Thus, we can compute all hash functions h_i, $1 \leq i \leq \hat{a}$, in expected time $O(n)$. The resulting mapping $h : S \to [n]$, $x \mapsto h_{\hat{h}(x)}(x)$, is a bijection. Each hash function h_i can be stored with $O(\log n)$ bits and thus the total space for storing h is $O(\hat{a} \cdot \log n) = O(n \cdot \log \log n)$.

Theorem 3. *For any n-element set $S \subseteq U$ a bijection $h : S \to [n]$ with encoding size $O(n \cdot \log \log n)$ can be constructed in expected time $O(n)$. The hash function can be evaluated in constant time and by probing $O(1)$ consecutive words from external memory (if the internal memory has size n^ϵ, $\epsilon > 0$).*

Acknowledgment

The author is grateful to Martin Dietzfelbinger and Rasmus Pagh for enlightening discussions on the subject of the paper. The anonymous referees provided very helpful comments.

References

1. E. D. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Pătrascu. De dictionariis dynamicis pauco spatio utentibus (lat. on dynamic dictionaries using little space). In *Proc. of the 7th LATIN*, volume 3887 of *LNCS*, pp. 349–361. 2006.
2. M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. of 13th STACS*, volume 1046 of *LNCS*, pp. 569–580. 1996.
3. M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *Proc. of 9th ESA*, number 2161 in LNCS, pp. 109–120. 2001.
4. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. of Alg.*, 25:19–51, 1997.
5. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. on Comp.*, 23:738–761, 1994.
6. M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proc. of 32nd ICALP*, volume 3580 of *LNCS*, pp. 166–178. 2005.
7. D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Comp. Syst.*, 38:229–248, 2005.
8. M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.
9. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. of the ACM*, 31:538–544, 1984.
10. T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. of 18th STACS*, volume 2010 of *LNCS*, pp. 317–326. 2001.
11. R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Proc. of 6th WADS*, volume 1663 of *LNCS*, pp. 49–54. Berlin, 1999.
12. R. Pagh and F. F. Rodler. Cuckoo hashing. *J. of Alg.*, 51:122–144, 2004.
13. P. Woelfel. Efficient strongly universal and optimally universal hashing. In *Proc. of 24th MFCS*, volume 1672 of *LNCS*, pp. 262–272. 1999.