Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Unique permutation hashing[☆]

Shlomi Dolev[*], Limor Lahiani, Yinnon Haviv

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel

ARTICLE INFO

Article history:
Received 18 September 2011
Received in revised form 10 November 2012
Accepted 25 December 2012
Communicated by P. Spirakis

ABSTRACT

We propose a new hash function, the *unique permutation hash function*, and a performance analysis of its hash computation. We denote the cost of a hash function h by $C_h(k, N)$, which stands for the expected number of table entries that are checked when inserting the $(k+1)^{st}$ key into a hash table of size N, where k out of N table entries are filled by previous insertions. A hash function maps a key to a permutation of the table locations. A hash function, h, is *simple uniform* if items are equally likely to be hashed to any table location (in the first trial). A hash function, h, is *random* or *strong uniform* if the probability of any permutation to be a probe sequence, when using h, is $\frac{1}{N!}$, where N is the size of the table.

We show that the unique permutation hash function is not only a *simple uniform* hash function but also a *random* hash function, i.e., *strong uniform*, and therefore has the optimal cost. Namely, each probe sequence is equally likely to be chosen when the keys are uniformly chosen. Our hash function ensures that each empty table location has the same probability to be assigned by a uniformly chosen key. We also show that the expected time for computing the unique permutation hash function is $O(1)$ and the expected number of table locations that are checked before an empty location is found during insertion (or search) is also $O(1)$ for constant load factors $\alpha < 1$, where the load factor α is the ratio between the number of inserted items and the table size.

1. Introduction

In this paper, we introduce a new hash function, the *unique permutation hash function*, for implementing the *open addressing* hashing scheme [4]. Our hash function is based on mapping each key (and its auxiliary data item) to a unique permutation. The unique permutation defines the probe sequence, namely, the locations that should be checked when inserting a key into the hash table (or when searching for a key).

Unique permutation hashing in a nutshell. Consider a hash table of size N, a hash function, h, maps a key to a table location in $1, 2, \ldots, N$. The *probe sequence* of a key, defined by the hashing method and h is, in fact, a permutation of $1, 2, \ldots, N$. For a given key x, assume that the probe sequence is $i_1, i_2, \ldots, i_N$. In this case, when inserting a key, x, into the hash table, the first location that is checked is i_1. If location i_1 is filled, then location i_2 is checked and so on until an empty location is found into which the key is inserted. Similarly, the same probe sequence also defines the sequence of table locations that are checked when searching for x. In this case, table locations are checked until x is found or until an empty location is found, indicating that the search has failed (x is not on the table).

[☆] Partially supported by IBM faculty award NSF grant, the Israeli Ministry of Defense (MAFAT), the Israeli Ministry of Science (Russia Israel grant), Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium, Grant from the Technion's Institute for Future Defense Technologies Research named for the Medvedi, Shwartzman and Gensler families, and the Israeli Internet Association. A brief announcement on this work has been presented in SSS 2009. Experiment results appear in Gichuiri (2010) [9] and Suriana (2011) [12].

[*] Corresponding author. Tel.: +972 8 6472718; fax: +972 8 6477650.
E-mail addresses: dolev@cs.bgu.ac.il (S. Dolev), lahiani@cs.bgu.ac.il (L. Lahiani), haviv@cs.bgu.ac.il (Y. Haviv).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.12.047

Consider a hash table of size N with locations denoted by $1, 2, \ldots, N$. Let $\Pi(N) = \{\pi_1, \pi_2, \ldots, \pi_{N!}\}$ be the set of all permutations of $1, 2, \ldots, N$, lexicographically ordered. The *unique permutation hash function* maps a key to a unique probe sequence in $\Pi(N)$. That is, if for a key x, $h(x) = \pi \in \Pi(N)$ and $\pi = i_1, i_2, \ldots, i_N$, then π is the probe sequence used for inserting x as well as for searching for x. As there are $N!$ permutations of the N table locations, the keys can be divided into $N!$ classes $[\pi]$, where $[\pi] = \{x : h(x) = \pi\}$. We next discuss the performance of the unique permutation hash function.

Optimal performance. We show that our unique permutation hash function is a *random hash function*, also defined as *strong uniform hash function*, which ensures that each empty entry has the same probability to be filled with a uniformly chosen key. Therefore, according to [13], our hash function also has an optimal expected number of location trials while inserting an element. The *random* property neither holds for linear probing nor for the widely used double hashing, since when probing a filled location using these methods some locations have a higher probability to be tried next, as we will demonstrate in Section 2. We note that [10] shows that double hashing has *asymptotically* uniform probing properties.

Let α be the ratio between the number of inserted keys and the table size; in the sequel we always assume that α is less than 1. The expected number of local steps done by the algorithm that computes the next probe number during insertion is a function of the load factor, namely, $O(\frac{1}{(1-\alpha)^3})$ steps.

The minimal expected number of locations that are inspected during key insertion (or key search) is analyzed in [13] with relation to random insertion. In this paper, it is stated that when inserting keys into the table at random, it holds that the expected number of locations which must be looked for until an empty one is found is $1 + k/(N - k + 1)$, where k is the number of filled locations and N is the table size. This expression is denoted by $C_0(k, N)$ and is used to measure a hash table efficiency. It refers to a case that the keys are chosen out of an infinite set of keys, I. The value of $C_0(k, N)$ can be improved for certain k and N, yet, it is a lower bound on the hash table performance in the sense that if an insertion algorithm is superior to random insertion for some random k, then it is inferior for some smaller value of k. We show that the unique permutation hash function has the performance of $C_0(k, N)$. Namely, the unique permutation hash function is random, which according to [13], implies that the expected number of table entry accesses is given by $C_0(k, N) = 1 + k/(N - k + 1)$, where k is the number of filled table entries. Note that for a bounded load factor, say $\alpha < \frac{2}{3}$, both the expected number of local steps and the expected number of table entry accesses are constants.

Related work. Research regarding efficient hash functions for implementing the hash table data structure and hash function analysis have been of great interest for a long time, e.g., [13,6,3,5,11,14]. The main open addressing hash methods include *linear probing* and *double hashing*.

Consider a hash table of size N, with locations denoted by $1, 2, \ldots, N$. In *linear probing* [4], given a hash function, h, and a key, x, the probe sequence of x is denoted by *linear_probe*$(x) = i_1, i_2, \ldots, i_N$, where $i_1 = h(x)$ and $i_j = (h(x) + (j - 1) \cdot c) \bmod N$, for $j = 2, \ldots, N$. That is, the interval (modulo N) between two sequential probes is fixed and given in c, where c is a constant number, usually 1.

In *double hashing* [4], given a hash function, h_1, a step function, h_2, and a key, x, the probe sequence of x is denoted by *double_probe*$(x) = i_1, i_2, \ldots, i_N$, where $i_1 = h_1(x)$ and $i_j = (h_1(x) + (j - 1)h_2(x)) \bmod N$, for $j = 2, \ldots, N$. That is, the interval (modulo N) between two sequential probes is fixed and given in $h_2(x)$.

Linear probing and the double hashing techniques are very well known and studied. We next present additional hashing schemes.

In *quadratic probing* [4], the probe sequence is denoted by $quadratic - probe(x) = i_1, i_2, \ldots, i_N$, where $i_j = (h(x) + c_1 \cdot (j - 1) + c_2 \cdot (j - 1)^2) \bmod N + 1$, h is a hash function, $j = 0, 1, \ldots, N - 1$, c_1, c_2 are constants and $c_2 \neq 0$.

In *dynamic perfect hashing* [8], a solution to the dynamic dictionary problem is suggested. Multiple levels of table hierarchies are used for storing the elements. In each level, the hash function used is chosen uniformly at random out of a set $\mathcal{H}_s = \{h : U \longrightarrow \{1, \ldots, s\} | h(x) = (kx \bmod p) \bmod s, 1 \leq k \leq p - 1\}$, where U is the universe of the keys, p is prime, and $p \geq |U|$. The randomized algorithm takes $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions. The deterministic algorithm has an amortized worst-case time complexity of $\Omega(\log n)$. The space complexity is linear in the number of elements currently stored in the table.

The *Cuckoo hashing* method has the same worst-case lookup time and the amortized expected time as dynamic perfect hashing in [8]. Two hash tables, T_1 and T_2, of the same size and two hash functions, h_1 and h_2, are used to store the keys in the universe, $\mathcal{U}$. Each key $x \in \mathcal{U}$ is stored either in $T_1[h_1(x)]$ or in $T_2[h_2(x)]$.

Our hash method does not involve rehashing and thus no amortized runtime analysis is required, but rather expected time analysis.

Jeffrey Ullman suggested in [13] a criterion for analyzing the performance of a hash function. The technique suggested evaluates the success of the hash function by uniformly distributing the keys to the hash table entries. We use this approach in order to analyze the performance of the unique permutation hash function. To the best of our knowledge, there are no other open addressing hashing schemes that achieve the optimal bound, $C_0(k, N)$, on the cost. In fact, we conjecture that the only hash scheme that may achieve this bound is the unique permutation hash scheme.

Recently, (later than [7]), a related scheme in terms of the general approach of using permutations for hashing was suggested in [1]. The main difference between the two approaches is in the goal of using permutations; whereas, in our scheme, permutations are used for determining the probing sequences of the elements, while [1] uses permutations for saving space.

Given two integers, N and x, we propose an algorithm that generates the xth permutation in $\Pi(N)$, where permutations are lexicographically ordered. Producing a permutation in lexicographic order was addressed in the past in a different context. Alon Itai [2] presents an algorithm to produce n consecutive permutations of $\{1, 2, \ldots, m\}$ in $O(n+m)$ local steps. Given a permutation, σ, the algorithm in [2] requires $O(1)$ amortized number of local steps and $O(1)$ additional space to produce n consecutive permutations in lexicographical order. In contrast, we compute each number in the permutation only upon request. For the sake of presentation completeness, we present a simple algorithm which finds the first j numbers in the xth permutation in $O(j^2)$ operations. In fact, it takes $O(1)$ expected number of operations for inserting a key, when α is bounded, say $\alpha < \frac{2}{3}$.

Number and range of keys. We also address the cases in which the value of the keys inserted can be greater than or smaller than $N!$. In case the value of the largest key is $c \cdot N!$, where c is a positive integer (c can be computed as a function of N), our function satisfies the random property. Note that when c is a non integer number greater than 1, there are some permutations that are chosen with a probability $\frac{\lfloor c \rfloor}{N!}$, while other permutations are chosen with a probability $\frac{\lfloor c \rfloor + 1}{N!}$. The ratio between these probabilities is negligible for a large enough c.

In case the range of keys is in (or can be mapped to) a smaller range than 0 to $N!$, we can *start probing the first table locations in a uniform fashion*, using our hash function (mapping the key to an index of a permutation in a shorter permutation domain, $N!/b!$, the largest domain that is equal to or smaller than the domain of the keys) and continuing in any deterministic fashion (say in a double hashing fashion) that probes the rest of the table locations.

Note that the use of the above smaller domain, $N!/b!$ (for a convenient choice of b), can be chosen to support efficient arithmetics when $N!$ is expensive to be used, even in case the value of the largest key is of the order of $N!$ or larger. We elaborate on this in the sequel.

2. Unique permutation hashing

For ease of presentation of our scheme, we assume a finite set of keys, I, of $N!$ possible key values, in the range $1, 2, \ldots, N!$. The *unique permutation hash function* h_{up} maps a key $x \in I$ to a permutation $\pi_i \in \Pi(N)$, $h_{up}(x) = \pi_x$, where π_x is the xth permutation in $\Pi(N)$, where all permutations are lexicographically ordered. We refer to the permutation $\pi_x = \langle i_1, i_2, \ldots, i_N \rangle$, where $h_{up}(x) = \pi_x$, as the *probe sequence* of x. Thus, when x is inserted into the hash table, the first location that is checked is i_1. If location i_1 is filled, then location i_2 is checked and so on until an empty location is found, into which x is inserted.

As there are $N!$ permutations of $1, 2, \ldots, N$, and there are $N!$ unique identifiers, one for each key in I, there is exactly one key in each one of the $N!$ classes, $[\pi]$, where $[\pi] = \{x : h_{up}(x) = \pi\}$.

A definition of a simple uniform hashing property is given in [4]. Assume we are given a hash table of size N and a key, x, which is independently chosen from a set of keys with a probability p_x. A hash function h is *simple uniform* if each table location, $j \in \{1, 2, \ldots, N\}$ has an equal probability that the chosen key x is hashed into that table location. Formally, a hash function, h, is *simple uniform* if $\sum_{x:h(x)=j} p_x = \frac{1}{N}$ for all $j = 1, 2, \ldots, N$. Note that the above definition of simple uniform hash function only refers to the first probe, i.e., the first number in the probe sequence of x. In other words, the probability that key x is inserted into a given table entry, when no collision occurs, is $\frac{1}{N}$. A hash function, h, is *random* if for any key, x, in the set of keys, the probe sequence $\langle i_1, i_2, \ldots, i_N \rangle$, defined by $h(x)$ and the probing method, is equally likely to be any permutation of $1, 2, \ldots, N$ [13]. This property of a hash function is a rather theoretical model for analyzing the performance of the function. Yet, the suggested unique permutation is an actual implementation of a hash function, which satisfies the random property.

Lemma 2.1. *The unique permutation hash function h_{up} satisfies the simple uniform hashing property.*

Proof. Let x be a key, which is randomly chosen out of the set I. As there are $N!$ keys in I, the probability of x to be chosen is $Pr(x) = \frac{1}{N!}$. Also, let $h_{up}(x) = \pi_k$, where $\pi_k = i_1, i_2, \ldots, i_N$. Given $j \in \{1, 2, \ldots, N\}$, the probability that x is inserted into location j in the hash table, when there are no collisions, is equal to the probability that $i_1 = j$. The number of permutations in $\Pi(N)$, for which $i_1 = j$, is $(N-1)!$. It is also the number of keys $x \in I$, for which the first number in $h_{up}(x)$ is j. The probability of such a key to be chosen is $\frac{1}{N!}$. Therefore, $\sum_{x:h(x)=j} Pr(x) = \frac{(N-1)!}{N!} = \frac{1}{N}$. $\square$

2.1. The random property of the permutation hash

As defined in [13], $S(\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k})$ is the set of table locations filled as a result of inserting the sequence of keys $\langle x_1, x_2, \ldots, x_k \rangle$, where $\pi_{i_j} = h_{up}(x_j)$ for $j = 1, \ldots, k$. Additionally, p_π denotes the probability of permutation $\pi \in \Pi(N)$ as computed from the hash function. A hash function, h, is *random* if $p_\pi = 1/N!$ for all permutations in $\Pi(N)$. Namely, each permutation is equally likely to be a probe sequence of a hashed key.

By the definition of h_{up}, it is clear that h_{up} is a random hash function. Each key is chosen uniformly out of I with a probability $\frac{1}{N!}$, and each key is mapped to a unique permutation in $\Pi(N)$ by h_{up}. Therefore, $p_\pi = \frac{1}{N!}$ for every permutation $\pi \in \Pi(N)$. The cost of a hash function, h, denoted by $C_h(k, N)$, is equal to $C_0(k, N) = 1 + k(N - k + 1)$ if h is random (see [13]). Therefore, it holds that $C_{h_{up}}(k, N) = C_0(k, N)$.

We now analyze the probability of a given table location to be assigned with a key, given a set, A, of filled table locations. We assume that a key can be drawn from the set of keys, I, independently and more than once (this is an approximation done for the sake of analysis). In this case, a key $x \in I$, which is inserted into the table and already exists in it, appears twice in the table.

We analyze the following process. Let A be any subset of $\{1, 2, \ldots, N\}$, where $|A| = k$ and $0 \le k < N$ and a hash table with filled locations in the indexes of A. Also given, an empty table location j. We claim that the probability of an element x_{k+1}, chosen uniformly at random out of I, to be inserted at location j is $\frac{1}{N-k}$. Namely, each empty table location is equally likely to be filled with key x_{k+1}.

First, for a hash table of size N and a hash function h, we define $S(\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k})$ to be the set of locations filled by the insertion sequence $X_k = \langle x_1, x_2, \ldots, x_k \rangle$, where $h(x_j) = \pi_{i_j}$ for all $j = 1, \ldots, k$ (as defined in [13]). We also define $p_h(loc(x_{k+1} = j)|A)$ to be the probability that key x_{k+1}, chosen uniformly at random out of I, is inserted at location j, when using the hash function, h. The filled table locations are denoted by a set A ($|A| = k$, $0 \le k < N$). We use the notation $p(loc(x_{k+1} = j)|A)$ when h is known (as in our case, when h is the unique permutation hash function).

For example, given a hash table of size $N = 3$ and a set of keys $I = \{x_1, \ldots, x_6\}$, where, for every $i = 1, \ldots, 6$, $x_i = i$. Also a sequence of insertions $X = \langle x_{i_1}, x_{i_2}, x_{i_3} \rangle$ is given. Denote the ith number in a given permutation π by $\pi(i)$. By definition, $\Pi(3) = \langle (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1) \rangle$. For every $j = 1, \ldots, 6$, let $\pi_{i_j} = h_{up}(x_{i_j})$.

Initially, the hash table is empty, thus $A = \emptyset$ (i.e., locations 1, 2, 3 are empty and $|A| = 0$). The first key, x_1, is chosen out of I with a probability $\frac{1}{N!} = \frac{1}{6}$. The probability $p(loc(x_1 = 1)|A)$ equals the probability that the first number in the permutation π_{i_1} is 1 (denoted by $p_{i_1}[1] = 1$). There are two such permutations in $\Pi(N)$: (1, 2, 3) and (1, 3, 2). Therefore $p(loc(x_1 = 1)|A)$ equals the probability that $\pi_{i_1} = (1, 2, 3)$ or $\pi_{i_1} = (1, 3, 2)$, which is $\frac{2}{6} = \frac{1}{3}$. The same argument holds for locations 2 and 3 and hence it is clear that $p(loc(x_1 = 1)|A) = p(loc(x_1 = 2)|A) = p(loc(x_1 = 3)|A) = \frac{1}{3}$.

Now, assume that x_1 is inserted into the table at location 1; hence $A = 1$ and the empty table locations are 2 and 3. The next inserted element is x_2 and let $\pi_{i_2} = h_{up}(x_2)$. The probability $p(loc(x_2 = 2)|A)$ is equal to the probability that either $\pi_{i_2}[1] = 2$ (there are two such permutations) or $\pi_{i_2}[1] = 1$ and $\pi_{i_2}[2] = 2$ (there is one such permutation). Hence, $p(loc(x_2 = 2)|A) = \frac{2}{6} + \frac{1}{6} = \frac{1}{2}$. The same argument holds when calculating $p(loc(x_2 = 3)|A)$, and hence $p(loc(x_2 = 2)|A) = p(loc(x_2 = 3)|A) = \frac{1}{2}$.

Lemma 2.2. *Consider a hash table of size N with* $0 \le k < N$ *filled entries denoted by a set* $A \subset \{1, 2, \ldots, N\}$ *(clearly,* $|A| = k$*). Also, consider a finite set, I, of N! distinct keys in the range* $\{1, 2, \ldots, N!\}$ *and an insertion sequence* $X_k = \langle x_{i_1}, x_{i_2}, \ldots x_{i_k} \rangle$ *of keys, chosen uniformly out of I such that* $S(h_{up}(x_{i_1}), h_{up}(x_{i_2}), \ldots h_{up}(x_{i_k})) = A$*. We claim that* $p(loc(x_{k+1} = j)|A) = \frac{1}{N-k}$ *for every empty table location j.*

Proof. Let A be the set of filled table locations ($|A| = k$, $0 \le k < N$), and let j be any empty table location. We calculate the probability $p(loc(x_{k+1} = j)|A)$, for the newly inserted key, x_{k+1}, and assume that $\pi_{i_{k+1}} = h_{up}(x_{k+1})$. Key x_{k+1} is inserted at location j if one of the following cases holds.
(1) $\pi_{i_{k+1}}[1] = j$ and the probability of this case is $\frac{(N-1)!}{N!} = \frac{1}{N}$.
(2) $\pi_{i_{k+1}}[1] \in A$ and $\pi_{i_{k+1}}[2] = j$ with a probability $\frac{\binom{k}{1} \cdot 1! \cdot (N-2)!}{N!}$.
(3) $\pi_{i_{k+1}}[1], \pi_{i_{k+1}}[2] \in A$ and $\pi_{i_{k+1}}[3] = j$ with a probability $\frac{\binom{k}{2} \cdot 2! \cdot (N-3)!}{N!}$, and so on....
Generally, case (i) stands for the case in which the first i probes in $\pi_{i_{k+1}}$ fail and probe $i + 1$ succeeds. The probability of case (i), where $i = 0, \ldots, k$, is $\frac{\binom{k}{i} \cdot i! \cdot (N-i-1)!}{N!}$.

Hence, the probability $p(loc(x_{k+1} = i_{k+1})|A)$ is the sum of the probabilities that either one of the above cases holds. Note that the cases are distinct.

$$p(loc(x_{k+1} = j)|A) = \frac{1}{N!} \cdot \sum_{i=0}^{k} \left(\binom{k}{i} \cdot i! \cdot (N - i - 1)! \right) \tag{1}$$

The probability $p(loc(x_{k+1} = j)|A)$ is calculated in the same way for any given set, A, of size $0 \le k < N$, as well as for any given empty index, j. Therefore, any empty location is equally likely to be assigned with the newly inserted key $x_{i_{k+1}}$ and it holds that $p(loc(x_{k+1} = j)|A) = \frac{1}{N-k}$. $\square$

Comparison with double hashing. In the case of double hashing, the ith probe number of a key x is denoted by $((h_1(x) + (i-1) \cdot h_2(x)) \bmod N)$ for $i = 1, \ldots, N$. Consider $k < N$ probe sequences $\pi_{i_1}, \ldots \pi_{i_k}$, which respectively match the sequence of keys $\langle x_1, \ldots, x_k \rangle$ previously inserted to the table. Also, consider a set $A \subseteq \{1, 2, \ldots, N\}$, where $S(\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k}) = A$, namely, the set of filled table locations resulting from the insertion of $x_1, \ldots, x_k$ to the initially empty hash table. The probability that the next randomly chosen key, x_{k+1}, is inserted into a given table location, i_{k+1}, depends on A.

Lemma 2.3. *Consider a hash table of size N, a hash function,* h_1*, a step function,* h_2*, and a randomly chosen key x. The probe sequence implied by* h_1, h_2 *and x is* $\langle i_1, i_2, \ldots, i_N \rangle$ *where* $\big(i_j = (h_1(x) + (j-1)h_2(x)) \bmod N\big)$ *for* $j = 1 \ldots N$*. There exist* $0 \le k < N$*, a set* $A = \{i_1, i_2, \ldots, i_k\}$ *and a table location* $i_{k+1} \notin A$*, such that* $p(loc(x_{k+1} = i_{k+1})|A) \ne \frac{1}{N-k}$*.*

```
1. state:
2.     $x' \in I$
3.     $probe\_seq \subseteq \{1, \dots, N\}$ /* probing order */
4.     $i \in \{0, \dots, N\}$
5.     $M \in 1, \dots N!$

6. findKPermutation_init($x$)
7.     $x' := x$
8.     $probe\_seq := \emptyset$
9.     $i = 0$
10.    $M := N!$

11.findKPermutation_probe()
12.    $x' := x' \bmod M$ /* find the key position within the current bucket range */
13.    $M = M/(N - i)$ /* define the next bucket size */
14.    $j := \lfloor x'/M \rfloor + 1$ /* find the bucket index in which the key resides */
15.    $next\_probe := convert(j, probe\_seq)$ /* convert to index in remaining unprobed entries */
16.    $probe\_seq.insert(next\_probe)$
17.    if $i < N - 1$ /* and hash table $next\_probe$ entry occupied */
18.        $i := i + 1$
19.        $findKPermutation\_probe()$
```

Fig. 1. Finding the xth permutation of $1, 2, \dots, N$.

Proof. For a given N (assuming $N > 6$), we construct $A = \{i_1, i_2, \dots, i_k\}$ as follows. First, we pick $k = 2$ and $i_1 = 1$. Next, we choose an interval, $1 \le j \le N$, such that $2j + 1 \le N$. Next, we define $i_2 = (i_1 + j)$ and $i_3 = (i_1 + 2j)$. Note that $i_3 \le N$. Finally, we calculate the probability that a randomly chosen key, x, is inserted at location i_3, namely $Pr_A(loc(x) = i_3)$. Key x is inserted at location i_3 if any of the following cases hold:

1. $h_1(x) = i_3$
2. $h_1(x) = i_1$ and $h_2(x) = j$
3. $h_1(x) = i_1$ and $h_2(x) = 2j$
4. $h_1(x) = i_2$ and $h_2(x) = j$.

Assuming h_1 and h_2 are simple uniform hash functions, the probability of case (1) is the probability that $h_1(k) = i_3$ and that is $\frac{1}{N}$, assuming h is uniform. The probability of either one of the cases (2, 3, 4) is $\frac{1}{N^2}$. Therefore, the probability that x is inserted at location i_3 is $\frac{3}{N^2} + \frac{1}{N}.\, p(loc(x = 3)|A) = \frac{3}{N^2} + \frac{1}{N}$. Assuming $N > 6$, $\frac{3}{N^2} + \frac{1}{N} \neq \frac{1}{N-2}$. □

Expected constant time table insertion. In order to calculate the table locations for probing using the unique permutation hash function, any algorithm that gradually maps a key to a permutation will fit. We suggest the specific algorithm presented in Fig. 1. The algorithm is composed of two methods. The first, `findKPermutation_init(`x`)`, is called before probing for insertion of the key x. This method is used for initiating the algorithm. The second method, `findKPermutation_probe()`, is called each time the next probe number in the probe sequence is required.

Code description. For simplicity, the code is written for a key x that is in the range 1 to $N!$, each such key belongs to one of the N portions, say the jth portion, of the range 1 to $N!$, which we call the jth bucket. The index of this bucket is the first index in the probe sequence. Once we extract the index of the first bucket, we change the scope to the position of the key x within the first bucket, by partitioning the first bucket (of size $(N-1)!$) to smaller $N-1$ buckets of size $(N-2)!$ each. The index of the smaller-bucket, j', in which the key x resides, implies the next index probed, which is the j'th in the ordered left, not-yet-probed indexes. The next indexes are found in a similar fashion.

Given an integer $1 \le x \le N!$, where N is known, the algorithm `findKPermutation_init(`x`)` initiates the state which is updated by the algorithm `findKPermutation_probe()`. The latter iterates over the numbers in the xth permutation in $\Pi(N)$, where permutations are lexicographically ordered. For example, for $N = 3$ and $x = 3$, after the state is initiated the first call for `findKPermutation_probe()` returns 2, the second call returns 1 and the third returns 3, because the third permutation in $\Pi(3)$ is (2, 1, 3).

The state of the algorithm presented in Fig. 1 is described in lines 2–5. Line 2 declares the variable x', which is, roughly speaking, the key not yet used information that defines the part of the xth permutation that has not been probed yet. Line 3 declares the variable *probe_seq*, which is the probe sequence that has been probed so far (sorted by the order of the probe). Line 4 declares the variable i that keeps track of the number of probes already made. Line 5 declares M, which, roughly speaking, is used for holding the current scope in which the position of the key is examined. We assume that $N!$ is known prior to all hash insertions.

```
1. /* finds the $j$th number not in $taken$, $j \in 1, 2, \ldots, N$ */
2. convert(j, taken)
3.     $curr := first(taken)$
4.     $j' := j$
5.     while $(curr! = null)$ and $(data(curr) \leq j')$
6.         $j' := j' + 1$ /* increment the index to accommodate past probe */
7.         $curr := next(curr)$ /* pass the already existing probe */
8.     return $j'$
```

Fig. 2. The `convert` method.

The method `findKPermutation_init(x)` initiates the state trivially. The method `findKPermutation_probe()`, presented in lines 11–19 of Fig. 1, is used for obtaining the next probe number in the probe sequence. The algorithm starts by calculating $x \bmod N!$ (and thus fits even larger range of keys), as shown in line 12. Note that in line 13, $N - i$ divides M in each probe, that changing the scope in which the position of the key is examined (namely the new bucket position within the last greater bucket). Line 14 sets j with $\lfloor k'/M \rfloor + 1$, which indicates the index of the (bucket, and) next probe number in the sorted sequence of numbers yet to be probed. Line 15 calls a sub-method for the jth number in $\{1, 2, \ldots, N\} \setminus probe\_seq$. The variable *next_probe* is set with this number and inserted to the *probe_seq* in the following line. Line 17 checks whether there is a need to increment i by one (line 18) and then the next index should be computed (line 19).

Fig. 2 contains the implementation for the convert method. The method finds the jth probe number simply by iterating the sorted set *taken* of already chosen probe numbers, increasing j on each encounter with a chosen number. Thus, the method's time complexity is bounded by $O(|taken|)$. Moreover, the time complexity of the ith execution of `findKPermutation_probe()` is $O(i)$, since the insertion of the chosen probe into *probe_seq* also takes $O(i)$ as it is sorted.

When analyzing the expected performance of hash table operations, it is customary to define the complexity in terms of the maximum load factor, $\alpha = \frac{n}{N}$, where n is the maximum number of filled table entries.

The expected number of filled entries that are checked until finding an empty entry for the newly inserted key is analyzed similarly to the open addressing analysis in [4]. The idealized assumption of that analysis, where each permutation has an equal probability to be a probe sequence is, in fact, a verified assumption in our system model. Hence, the expected number of probes while inserting a key is less than or equal to $\frac{1}{1-\alpha}$.

We now turn to analyze the probe sequence computation in terms of α. We claim that the current implementation of the hash function computation provides $O\left(\left(\frac{1}{1-\alpha}\right)^3\right)$ expected time complexity for insertion (or unsuccessful search). That is, we assume that the probability of each table location to be filled is at most α. Our implementation provides $O\left(\left(\frac{1}{1-\alpha}\right)^3\right)$ expected time complexity for insertion (or unsuccessful search) operation with *no* assumptions on the current table configuration. Notice that in terms of the table size, we still obtain average constant time complexity. The following lemma formally states the expected runtime for insertion.

Theorem 2.4. *Using the unique permutation hash function for inserting keys into a hash table with a load factor of at most α, then regardless of the table configuration it holds that the expected runtime is at most $O\left(\left(\frac{1}{1-\alpha}\right)^3\right)$.*

Proof. The use of the unique permutation hash function ensures that the probability of each probe number in the probe sequence to fail is at most α for every table configuration. Let c be a constant, such that $c \cdot i$ bounds the runtime of the ith call to `findKPermutation_probe()` for a given x. Thus, an insertion that succeeds after j probes takes $\sum_{i=1}^{j} c \cdot i = O(j^2)$.

$$\begin{aligned}
E[\text{runtime of insertion}] &= \sum_{j=1}^{n} j^2 \cdot Pr[\text{key is inserted in exactly } j \text{ probes}] \\
&\leq \sum_{j=1}^{n} j^2 \cdot \alpha^{j-1}(1-\alpha) \\
&\leq \sum_{j=1}^{\infty} j^2 \cdot \alpha^{j-1}(1-\alpha) \\
&\leq \sum_{j=1}^{\infty} j^2 \cdot \alpha^{j-1} \\
&= \left(\sum_{j=1}^{\infty} j \cdot \alpha^j\right)' = \left(\frac{1}{(1-\alpha)^2}\right)' = \frac{2}{(1-\alpha)^3} = O\left(\frac{1}{(1-\alpha)^3}\right)
\end{aligned} \tag{2}$$

Note that for $j > N$ it holds that $Pr[\text{key is inserted in exactly } j \text{ probes}] = 0$. $\square$

Computation with large values. The above analysis assumes that modulo and division operations on numbers are executed in constant time. The number of basic ALU operations needed to compute the modulo and division is architecture dependent. In some cases, one may consider using a smaller key range, $N!/b!$, to facilitate the use of efficient ALU operations for computing a sufficient prefix of the probe sequence and continue in a different way, such as double hashing over the leftover entries. Such a reduction in the values handled by the ALU will keep the benefits of the scheme in the first $N - b$ probes (which suffices for reasonable α values) and reduce the computation needed for finding the next index to be probed. Note that arithmetic is always done in the range of possible key values or smaller, as the value of a key encodes a specific permutation, or a prefix of a permutation. Thus, the arithmetic used during the computation of probes is always based on the size of the key range values or less.

Experimental results. Experiments for comparing the performance of the unique permutation hashing with standard hashing were performed and the results are reported in [9,12]. The experiments used Python implementations of the insert procedure of the compared hash tables. The sizes of the tables examined, N, in the experiments were in the range of 50 to 750 and consider range of load factors from 0.1 by steps of 0.1 up to almost 1. Keys were randomly chosen uniformly in the range 0 to $N! - 1$ and the number of probes to insert keys were counted and compared among the different hash table schemes. The conclusions from [9,12] are that the unique permutation hash function performs better than the linear probing, quadratic hashing, and double hashing, with which it is compared. The unique permutation hash function maintains its optimal performance over a wide range of table sizes — both prime and non prime number table sizes. The practical experiments support our theoretical proof that the unique permutation hash function is a random hash function; any not-yet-explored entry has the same probability to be explored during the key insertion/search, which is a property that does not hold for other hash functions, in particular for double hashing. We believe that our technique is a better alternative to double-hashing and other popular hashing techniques when there is a need to optimize the performance at the expense of losing some of the simplicity of the double hashing technique.

3. Conclusions

We presented the unique permutation hash function designed for $N!$ keys and proved that it satisfies the *random* property. According to [13], the random property implies an optimal number of failed probes when inserting a key into the table. This implies the best worst-case insertion time. In the case $I = \{1, \ldots, (c \cdot N!)\}$, where c is a constant integer, the random property of our hash function h_{up} is not affected. In this case, the number of keys in each equivalent set of keys that share the same probing sequence is c. On the other hand, if $|I| < N!$, we may still use a prefix of the permutation according to our algorithm and complete the permutation in some pre-defined deterministic way. In this case, we preserve uniformity while scanning table locations by using the first permutation indexes. When α is small enough, the entire process takes a small number of uniform probing steps over the table indexes.

Acknowledgments

We gratefully thank Patrick Gichuiri and Patricia A. Suriana for simulating the algorithm.

References

[1] Y. Arbitman, M. Naor, G. Segev, Backyard Cuckoo hashing: constant worst case operations with a succinct representation, in: Annual Symposium on Foundations of Computer Science, FOCS, 2010.
[2] I. Alon, Generating permutations and combinations in lexicographical order, Journal of the Brazilian Society 7 (3) (2001) 65–68.
[3] Y. Azar, A.Z. Broder, A.R. Karlin, E. Upfal, Balanced allocations, SIAM Journal on Computing 29 (1) (1999) 180–200 (electronic).
[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press and McGraw-Hill, 2001.
[5] P. Celis, P.-A. Larson, J. Munro, Robin Hood hashing, in: 26th IEEE Symposium on the Foundations of Computer Science, 1985, pp. 281–288.
[6] J.L. Carter, M.N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences 18 (1979) 143–154.
[7] S. Dolev, L. Lahiani, Y. Haviv, Unique permutation hashing, in: Proc. of the 11th International Symposium on Stabilization, Safety and Security, 2009, also in Ph.D. Thesis of Limor Lahiani, April 2008.
[8] M. Dietzfelbinger, A.R. Karlin, K. Mehlhorn, F.M. auf der Heide, H. Rohnert, R.E. Tarjan, Dynamic perfect hashing: upper and lower bounds, SIAM Journal on Computing 23 (4) (1994) 738–761.
[9] P. Gichuiri, Unique permutation hashing, experiment results, Dept. of Computer Science, Ben-Gurion University, Technical Report 10-08, August, 2010.
[10] G.S. Lueker, M. Molodowitch, More analysis of double hashing, in: STOC, 1988, pp. 354–359. Journal version Combinatorica 13 (1) (1993) 83–96.
[11] R. Pagh, F.F. Rodler, Cuckoo hashing, in: ESA: Annual European Symposium on Algorithms, in: LNCS, vol. 2161, Springer, 2001.
[12] P. Suriana, Unique permutation hashing, performance analysis, Dept. of Computer Science, Ben-Gurion University, Technical Report 11-10, August, 2011.
[13] J.D. Ullman, A note on the efficiency of hashing functions, Journal of the ACM 19 (3) (1972) 569–575.
[14] A.C. Yao, Uniform hashing is optimal, Journal of the ACM 32 (3) (1985) 687–693.