# Unique Permutation Hashing

Shlomi Dolev* Limor Lahiani* Yinnon Haviv*

May 19, 2009

**Abstract**

We propose a new hash function, the *unique-permutation hash function*, and a performance analysis of its hash computation. Following the notations of [15], the cost of a hash function $h$ is denoted by $C_h(k, N)$ and stands for the expected number of table entries that are checked when inserting the $(k+1)^{st}$ key to 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. According to [15], the cost of a random hash function, denoted by $C_0(k, N) = 1 + k/(N-k+1)$, is a "lower bound" on hashing performance. A lower bound in a sense that if some hash function $h'$ has a cost $C_{h'} < C_0(k, N)$, for some $k$ and $N$, then there exists $k' < k$ such that $C_{h'}(k', N) > C_0(k', N)$. 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 cost of $C_0(k, N)$. 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 with 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 $\alpha$ is the ratio between the number of inserted items and the table size. The unique-permutation hash function has an application for parallel, distributed and multi-core systems that avoid contention as much as possible.

*Department of Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Partially supported by IBM faculty award, NSF grant and the Israeli ministry of defense. Email: {dolev,lahiani,haviv}@cs.bgu.ac.il.

# 1 Introduction

In this paper, we introduce a new hash function, the *unique-permutation hash function*, for implementing the *open addressing* hashing scheme [2]. Our hash function is based on mapping each 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).

Given 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 that case, when inserting an item with key $x$ to the hash table, the first location which is checked is $i_1$. If location $i_1$ is filled, then location $i_2$ is checked and so on until finding an empty location into which the item is inserted. Similarly, the same probe sequence also defines the sequence of table location that are checked when searching for $x$. In that 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).

Given 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 an item $x$, $h(x) = \pi \in \Pi(N)$ and $\pi = i_1, i_2, \ldots, i_N$, then $\pi$ 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 items can be divided into $N!$ classes $[\pi]$, where $[\pi] = \{x : h(x) = \pi\}$.

We show that our unique-permutation hash function is a *random hash function*, (also defined *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 [15], our hash function also has an optimal expected number of location trials while inserting an element. Obviously the *random* property neither holds for linear probing, nor for double hashing, since when probing an filled location, according to these methods, some locations have a higher probability to be tried next, as we will demonstrate. We note that [12] shows that double hashing have *asymptotically* uniform probing properties.

We distinguish *local steps*, which refers to the hash function computation from a hash *table entry access*. We assume that table entry access is much more expensive, in terms of time, than hash computation. One may assume that such access would require interaction/communication with a secondary memory device, while computation of the hash function is done in memory. Still, in case keys are given as integers and obviously, when keys are explicitly given as permutations, we present a simple algorithm that (locally) calculates the next probe number in the probe sequence upon a request. The expected number of local steps done by this algorithm, during insertion, is a function of the load factor (i.e. $\frac{1}{(1-\alpha)^3}$). We also prove that the unique-permutation hash function is random, which according to [15] implies that the expected number of table entry accesses is given in $C_0(k, N) = 1 + k/(N - k + 1)$, 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., [15, 4, 16]. The main open addressing hash methods include *linear probing* and *double hashing*.

Given a hash table of size $N$, with locations denoted by $1, 2, \ldots, N$. In *linear probing* [2], 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 $i_j = (h(x) + (j-1) \cdot c) \mod N$, 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* [2], 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 $h_2(x)$.

The linear probing and the double hashing techniques are very well known and studied, yet there are further hashing schemes.

In *quadratic probing* [2], 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, $j = 0, 1, .., N-1$, $c_1, c_2$ are constants and $c_2 \neq 0$.

In *dynamic perfect hashing* [9] 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 $\mathcal{H}_s = \{h : U \longrightarrow \{1, ..., s\} | h(x) = (kx \bmod p) \bmod s, 1 \leq k \leq p-1\}$, 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 of [9]. 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 $k \in \mathcal{U}$ is stored either in $T_1[h_1(k)]$ or in $T_2[h_2(k)]$.

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

Jeffrey Ullman suggested in [15] 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 is no other open addressing hashing schemes that achieves the optimal bound $C_0(k, N)$ on the cost. In fact, we conjecture the only hash scheme that may achieve this bound is the unique-permutation hash scheme.

**Our work**. Given two integers $N$ and $k$, we propose an algorithm that generates the $k^{th}$ permutation in $\Pi(N)$ (when lexicographically ordered). We compute each number in the permutation only upon a request. For the sake of presentation completeness, we present a simple algorithm that finds the first $j$ numbers in the $k^{th}$ permutation in $O(j^2)$ operations. In fact, it takes $O(1)$ expected number of operations for inserting an item, when $\alpha$ is bounded, say $\alpha < \frac{2}{3}$. We also address the cases in which the number of items inserted is greater or smaller than $N!$. In case the number of items is $c \cdot N!$, where $c$ is a positive integer, our function satisfies the random property. Note that when $c$ is a non integer number that is greater than 1, there are some permutations that are chosen with probability $\frac{\lfloor c \rfloor}{N!}$, while other permutations are chosen with probability $\frac{\lfloor c \rfloor + 1}{N!}$. The ratio between these probabilities is negligible for a large enough $c$. In case the number of items is less than $N!$, we can *start probing the first table locations in a uniform fashion*, using our hash function and continuing in any deterministic fashion (say in a linear probing fashion) that probes the rest of the table locations.

The rest of the paper is organized as follows. In the next section we introduce the unique-permutation hash function and present our runtime complexity analysis. Lastly, we suggest applications, in the domain of sensor networks, and parallel access in a multi-core multi-processing

systems to shared memory, that can benefit the use of our unique-permutation hash function.

## 2 Unique-Permutation Hashing

In this paper we assume a finite set of items $I$ of $N!$ items with unique identifiers (keys) in the range $1, 2, \ldots, N!$. The *unique-permutation hash function* $h_{up}$ maps an item $x \in I$ to a permutation $\pi_i \in \Pi(N)$ in the following way: Let $i = id(x)$, where $i \in \{1, 2, \ldots, N!\}$, the unique identifier of $x$. $h_{up}(x) = \pi_i$, where $\pi_i$ is the $i^{th}$ permutation in $\Pi(N)$, where all permutations are lexicographically ordered. We refer the permutation $\pi_i = \langle i_1, i_2, \ldots, i_N \rangle$, where $h_{up}(x) = \pi_i$ , as the *probe sequence* of $x$. Thus, when $x$ is inserted to the hash table, the first location which is checked is $i_1$. If location $i_1$ is filled, then location $i_2$ is checked and so on until finding an empty location, into which $x$ is inserted.

As there are $N!$ permutations of $1, 2, \ldots, N$, and there are $N!$ unique identifiers, one for each item in $I$, there is exactly one item 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 [2]. Assume a given a hash table of size $N$ and an item $x$, which is independently chosen from a set of items with 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 it. 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 item $x$ is inserted to a given table entry, when no collision occurs, is $\frac{1}{N}$ . A hash function $h$ is *random* if for any item $x$ in the set of items, 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 $1, 2, \ldots, N$ [15]. 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.

### 2.1 Random property of permutation hash

As defined in [15], $S(\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_k})$ is the set of table locations filled as a result of inserting the sequence of items $\langle x_1, x_2, \ldots, x_k \rangle$, where $\pi_{i_j} = h_{up}(x_j)$ for $j = 1, .., k$. Also, $p_\pi$ 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 item.

By the definition of $h_{up}$ it is clear that $h_{up}$ is a random hash function. Each item is chosen uniformly out of $I$ with probability $\frac{1}{N!}$, and each item 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 [15]). 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 an item can be drawn from the set of items $I$, independently and more than once (approximation done for the sake of analysis). In that case, an item $x \in I$, which is inserted to 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 $0 \leq k < N$ and a hash table with filled locations denoted by $A$. 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 item $x_{k+1}$.

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 $j = 1, .., k$ (as defined in [15]). We also define $p_h(loc(x_{k+1} = j)|A)$ to be the probability that item $x_{k+1}$, chosen uniformly at random out of $I$ is inserted at location $j$ when using the hash function $h$ and the filled table locations are denoted by a set $A$ ($|A| = k, 0 \leq k < N$). 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 items $I = \{x_1, \ldots x_6\}$, where, for every $i = 1, \ldots, 6$ $id(x_i) = i$. Also given a sequence of insertions $X = \langle x_{i_1}, x_{i_2}, x_{i_3} \rangle$. Lets denote the $i^{th}$ number in a given permutation $\pi$ 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 item $x_1$ is chosen out of $I$ with probability $\frac{1}{N!} = \frac{1}{6}$. The probability $p(loc(x_1 = 1)|A)$ equals to the probability that the first number in the permutation $\pi_{i_1}$ 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 to 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 to 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 $p(loc(x_2 = 2)|A)$ equals 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 2.1** *Given a hash table of size $N$ with $0 \leq k < N$ filled entries denoted by a set $A \subset \{1, 2, \ldots, N\}$ (clearly, $|A| = k$). Also given a finite set $I$ of $N!$ items with unique identifiers 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 items, 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 ($|A| = k, 0 \leq k < N$), and let $j$ be any empty table location. We calculate the probability $p(loc(x_{k+1} = j)|A)$, for the newly inserted item $x_{k+1}$ and assume that $\pi_{i_{k+1}} = h_{up}(x_{k+1})$. Item $x_{k+1}$ is inserted at location $j$ if one of the following cases holds.
(1) $\pi_{i_{k+1}}[1] = j$ and the probability of that 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 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 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 $\pi_{i_{k+1}}$ fails 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 probabilities that either one of the above cases holds. Note that the cases are distinct.

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

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

**Comparison with double hashing**. In the case of double hashing, the $i^{th}$ probe number of an item $x$ is denoted by $((h_1(x) + (i-1) \cdot h_2(x)) \bmod N)$ for $i = 1..N$. Given $k < N$ probe sequences $\pi_{i_1}, \ldots \pi_{i_k}$, which respectively match the sequence of items $\langle x_1, \ldots, x_k \rangle$ previously inserted to the table. Also, given 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 to a given table location $i_{k+1}$ depends on $A$.

**Lemma 2.2** *Given a hash table of size $N$, a hash function $h_1$, a step function $h_2$ and a randomly chosen item $x$. The probe sequence implied by $h_1, h_2$ and $x$ is $\langle i_1, i_2, \ldots, i_N \rangle$ where $(i_j = (h_1(x) + (j-1)h_2(x)) \bmod N)$ for $j = 1 \ldots N$. There exists $0 \leq k < N$, a set $A = \{i_1, i_2, \ldots, i_k\}$ and a table location $i_{k+1} \notin A$, such that $p(loc(x_{k+1} = i_{k+1})|A) \neq \frac{1}{N-k}$.*

*Proof:* For a given $N$ (assuming $N > 6$), we construct $A = \{i_1, i_2, \ldots, i_k\}$ as follows: First, we pick $k = 2$ and $i_1 = 1$. Next, we choose an interval $1 \leq j \leq N$, such that $2j + 1 \leq N$. Next, we define $i_2 = (i_1 + j)$ and $i_3 = (i_1 + 2j)$. Note that $i_3 \leq N$. Finally, we calculate the probability that a randomly chosen key $x$ is inserted at location $i_3$, namely $Pr_A(loc(x) = i_3)$. Item $x$ 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, we suggest the algorithm presented in Figure 1. The algorithm is composed of two methods. The first, `findKPermutation_init(k)` is called before probing for insertion of the key $k$. 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**. Given an integer $1 \leq k \leq N!$, where $N$ is known, the algorithm `findKPermutation_init(k)` initiates the state which is updated by the algorithm

```
1. state:
2.     k' ∈ K
3.     probe_seq ⊆ {0,...,M−1} // sorted
4.     i ∈ {0,...,N}
5.     M ∈ 1,...N!

6. findKPermutation_init(k)
7.     k' := k
8.     probe_seq := ∅
9.     i = 0
10.    M := N!

11.findKPermutation_probe()
12.    M = M/(N − i)
13.    j := k'/M + 1
14.    k' := k' mod M
15.    next_probe := convert(j, probe_seq)
16.    probe_seq.insert(next_probe)
17.    i := i + 1
18.return next_probe
```

Figure 1: Finding the $k^{th}$ permutation of $1, 2, \ldots, N$

`findKPermutation_probe()`. The latter iterates over the numbers in the $k^{th}$ permutation in $\Pi(N)$ (lexicographically ordered). For example, for $N = 3$ and $k = 3$, after the state is initiated, the first call for `findKPermutation_probe()` returns 2, the second call returns 1 and the third returns 3, as the third permutation in $\Pi(3)$ is $(2, 1, 3)$.

The state of the algorithm presented in Figure 1 is described in lines 2-5. Line 2 declares the variable $k'$ which is used to indicate the part of the $k^{th}$ 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 is used for holding the value $(N - i - 1)!$ in each probe. We assume that $N!$ is known prior to all hash insertions.

The method `findKPermutation_init(k)` initiates the state trivially. The method `findKPermutation_probe()`, presented in Lines 11-18 of Figure 1 is used for obtaining the next probe number in the probe sequence. The algorithm starts by calculating $(N - i - 1)!$ by dividing $M$ by $(N - i)$, as shown in Line 12. Note that $N - i$ divides $M$ in each probe, since $M$ is initiated by $N!$. Line 13 sets $j$ with $(k'/M + 1)$, which indicates the index of the next probe number in the sorted sequence of numbers yet to be probed. Line 14 sets $k'$ with $(k' \mod M)$ to indicate the probed numbers so far. Line 15 calls a sub-method for the $j^{th}$ 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 increments $i$ by one, for the next iteration. The requested probe number *next_probe*, is returned on line 18.

Figure 2 contains the implementation for the convert method. The method finds the $j^{th}$ probe number simply by iterating the sorted set *taken* of already chosen probe numbers, increasing $j$ on each encounter with a chosen number. It is easy to see that the method's time complexity is bounded

by $O(|taken|)$. Moreover, the time complexity of the $i^{th}$ 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.

```
1. /* finds the j'th number not in taken, j ∈ 1, 2, ... N */
2. convert(j, taken)
3.     curr := first(taken)
4.     j' := j
5.     while (curr! = null)and(data(curr) ≤ j')
6.         j' := j' + 1
7.         curr := next(curr)
8.         return j'
```

Figure 2: The `convert` method

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 item is analyzed similarly to the open addressing analysis in [2]. The idealized assumption of that analysis, where each permutation having 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 $\alpha$. 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, assuming the probability of each table location to be filled is at most $\alpha$. 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.

**Lemma 2.3** *When using the unique-permutation hash function for inserting items to a hash table with a load factor of at most $\alpha$, for every table configuration, the expected runtime is $O\left(\left(\frac{1}{1-\alpha}\right)^3\right)$.*

*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 $\alpha$, for every table configuration. Let $c$ be a constant, such that $c \cdot i$ bounds the runtime of the $i^{th}$ call to `findKPermutation_probe()` for a given $k$. Thus, an insertion which succeeds after $j$ probes takes $\sum_{i=1}^{j} c \cdot i = O(j^2)$.

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

Note that for $j > n$ it holds that $Pr[\text{item is inserted in exactly } j \text{ probes}] = 0$. ■

Note that 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 an efficient algorithm for such operations (e.g. the efficient algorithm for division and remainder computation on long numbers in [13]).

# 3 Conclusions

We presented the unique-permutation hash function, designed for $N!$ keys and proved that it satisfies the *random* property. According to [15], the random property implies an optimal number of failed probes when inserting an item to the table. This implies the best worst-case insertion time. In the case $I = \{1, .., (c \cdot N!)\}$, where $c$ is a constant integer, the random property of our hash function $h_{up}$ is not affected. In that case, the number of items in each equivalent set of items which 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 indices. When $\alpha$ is small enough, the entire process takes small number of uniform probing over the table indices.

**Applications.** In the scope of self-stabilization [5, 6], it is important to consider the performance for any possible initial state. In this scope, permutation hashing, unlike double hashing ensures uniform distribution. The reason is that permutation hashing probes each location with the same probability regardless of the current occupied locations.

In the scope of sensor networks, when virtual stationary automata are used for storing information [7, 8, 10], defining a procedure that maps keys or user information to a virtual automata located in a region, is required. The unique-permutation hash will naturally support a self-stabilizing implementation of such a scheme.

Our hash function can be used in the scope of parallel machines that interact through shared memory, when the machines use a shared hash table). Since each machine will access a table entry

in a uniform fashion, the access to the hash table using our hash function will result in the minimal possible contention; see the research direction identified in [11]. This is very important for the current multi-core architecture.

## References


[1] Azar, Y., Broder, A., Z., Karlin, A., R., and Upfal., E, "Balanced Allocations", *SIAM J. Comput.*, 29(1):180-200 (electronic), 1999.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., *Introduction to Algorithms*, $2^{nd}$ edition, MIT Press and McGraw-Hill, 2001.

[3] Celis, P., Larson, P.-A., and Munro, J., "Robin Hood hashing", In *26th IEEE Symposium on the Foundations of Computer Science*, pp. 281288, 1985.

[4] Carter J. L., M. N. Wegman "Universal Classes of Hash Functions", *Journal of Computer and System Sciences* 18 (1979), pp. 143-154.

[5] Dijkstra, E.W., "Self stabilizing systems in spite of distributed control", *Communications of the ACM*, 1974.

[6] Dolev, S., *Self-Stabilization*, MIT Press, 2000.

[7] Dolev, S., Herman, T., and Lahiani, L., "Polygonal Broadcast, Secret Maturity and the Firing Sensors", *Third International Conference on Fun with Algorithms (FUN)*, pp. 41-52, May 2004. Also in *Ad Hoc Networks Journal*, Vol 4, Issue 4, pp 447-486, July 2006.

[8] Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., Nolte, T., "Timed Virtual Stationary Automata for Mobile Networks", *Proc. of the 2005 International Conference on Principles of Distributed Systems, (OPODIS)*, LNCS, 2005. Also in the *43rd Allerton Conference on Communication, Control, and Computing*, September, 2005.

[9] Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., auf der Heide, F. M., Rohnert, H., and Tarjan, R. E., "Dynamic Perfect Hashing: Upper and Lower Bounds", *SIAM Journal on Computing*, 23(4):738761, 1994.

[10] Dolev, S., Lahiani, L., Lynch, N., Nolte, T., "Self-stabilizing Mobile Node Location Management andMessage Routing", *Proc. of the 7th International Symposium on Self-Stabilizing Systems*, October, 2005.

[11] Karp, M. R., Luby, M., Meyer auf der Heide, F., "Efficient PRAM Simulation on a Distributed Memory Machine", *Algorithmica*, 16(4/5) pp. 517-542, 1996.

[12] Lueker, G. S., Molodowitch, M., "More Analysis of Double Hashing", STOC, pp. 354-359, 1988. Journal version *Combinatorica* 1993.

[13] Menezes, A., Oorschot, v. P., Vanstone, S., *Handbook of Applied Cryptography*, CRC Press, Chapter 10, 1997.

[14] Pagh, R., and Rodler, F. F., "Cuckoo Hashing", In *ESA: Annual European Symposium on Algorithms*, vol. 2161 of LNCS, Springer, 2001.

[15] Ullman, J., D., "A note on the efficiency of hashing functions", *Journal of the ACM*, 19(3):569-575, July 1972.

[16] A. C. Yao, "Uniform hashing is optimal", *Journal of the ACM*, 32(3):687-693, 1985.