Cuckoo Hashing with Perfect Rehash

Judismar Arpini Junior[1], Vinícius Gusmão Pereira de Sá[1,*]

Abstract

The worst-case time to insert a key in the classic cuckoo hashing scheme is a random variable that may assume arbitrarily big values, owing to the strictly positive probability that an endless sequence of rehashes take place. We propose a cuckoo hashing variant in which the worst-case insertion time is polynomial. To accomplish this, we use two basic ideas. The first is to employ a perfect hashing method on one of the tables whenever a rehash is called for. The second idea is to make it so that the number of underlying hash tables is no longer constant, but rather an appropriate function of the number of keys. The price to pay is a higher lookup time, which is no longer constant, but doubly logarithmic.

Keywords: Hashing, Data structures, Computational complexity, Algorithms
2010 MSC: 00-01, 99-00

1. Introduction

The well-known cuckoo hashing technique offers constant worst-case lookup time and constant amortized insertion time [1]. In general terms, there are $d \geq 2$ hash tables T_j and d hash functions h_j, $1 \leq j \leq d$. To insert a key k_1, the first hash function is applied on k_1, producing a position $p_1 = h_1(k_1)$ in the first table, where k_1 will be inserted. If the position p_1 is not empty in T_1, the existing key (say, k_2) is removed from T_1 and reinserted by following the same steps above, except that we will now use the second table T_2 (and the

[*]Corresponding author
Email address: `vigusmao@dcc.ufrj.br` (Vinícius Gusmão Pereira de Sá)
[1]Universidade Federal do Rio de Janeiro, Brazil

second hash function h_2). This procedure goes on recursively, reinserting in the next table, in circular fashion, the key that was evicted from a table where the target position was occupied, cycling through tables (and hash functions) until an insertion takes place in a position that was *not* previously occupied. To look up a key k, it suffices to read the position given by function $h_j(k)$ on table T_j, for all $j \in \{1, \ldots, d\}$.

It is possible that the insertion algorithm enters a loop and fails to accommodate all keys. When such an event occurs, the standard solution is to perform a *full rehash* on all n existing keys using new hash functions. The new functions, though, may also fail with strictly positive probability. Thus, the worst-case insertion time is *infinite*, i.e. it can execute in arbitrarily large amounts of time. The classic value for d is 2, and, while values greater than 2 have been used with practical savings in memory usage [2], the worst-case insertion time has remained unchanged.

We propose a cuckoo hashing variant in which the worst-case insertion time is no longer infinite. We introduce the *perfect rehashing* technique, ensuring a bounded worst-case behavior by reinserting $o(n)$ keys in a collision-free manner. Preventing infinite worst-case times is not new in the literature, e.g. a Las Vegas algorithm can be converted into a Monte Carlo algorithm to yield finite predictable time [3]. Our Cuckoo Hashing with Perfect Rehash scheme (CHPR) does not use a constant number d of tables, but instead a function $d = \delta(n)$ of the total number n of keys intended to be inserted.[2] By making $\delta(n) = \Theta(\log \log n)$, we guarantee a worst-case $\Theta(\log \log n)$ time for lookups. The insertion takes $O(\log \log n)$ expected time, and $O\left(n^3/(\log \log n)^3\right)$ worst-case time. Such results may be appealing to scenarios where a bounded worst-case insertion time is important, while still requiring a near-optimal performance for lookups.

[2]If it turns out that we store more than the originally intended number of keys, it is always possible to recompute the number of tables, adding more tables if necessary—whereupon a full rehash must take place, much like what is done in standard hash table schemes to keep the load factor under control.

In this text, we let $[u] = \{1, 2, \ldots, u\}$ be the universe of all possible keys, for some integer $u \geq n$. We assume all hash functions are independent and uniform. This assumption is reasonable thanks to universal hash function schemes [4], a technique whose application to cuckoo hashing has been discussed thoroughly [5]. We assume that reading and writing in a table takes $O(1)$ time.

2. A cuckoo scheme with bounded insertion time

Our cuckoo hashing variant employs $\delta(n) = \lfloor \log_2 \log_2 n \rfloor + 3$ tables, each of them associated to a hash function. The size of each table is $\lceil (1+\epsilon)n/\delta(n) \rceil$, for some small $\epsilon > 0$. The rationale for enforcing that $\delta(n) \geq 4$ (for $n \geq 4$) is based on theoretical and empirical results and aims at optimal memory usage [2].

To look up a key k, we still check all its candidate locations (one per table). To insert a key k, we first look it up to avoid duplicates, and then we use a random walk approach. We pick a table T_j uniformly at random, $1 \leq j \leq \delta(n)$, among those having at least one empty slot, and insert k at position $h_j(k)$ in T_j. If another key is found in $h_j(k)$, that pre-existing key is evicted. In this case, we pick another random table and repeat the insert-and-evict procedure until no key is evicted or a predefined limit of iterations is exceeded. In the latter case, a rehash is performed on a single table, namely the one where the latest dislodged key came from. Since the keys to be (re-)inserted are all known, we compute a perfect hash function [6], so the rehash itself is always successful, with no two keys disputing the same spot. We call this operation a *perfect rehash.*

2.1. Perfect Rehash

Whenever the keys to be inserted in a hash table are known beforehand, the perfect hashing method can be applied to avoid collisions [6]. That is precisely the case when rehashing the $r = O(n/\log\log n)$ keys that resided in one of the tables constituting our cuckoo scheme plus the key evicted last (waiting to be reinserted in that same table). We are particularly interested in the *random hypergraphs* perfect hashing method [7], with linear expected time. It so happens that a subprocedure of that method is a *random search* algorithm: an auxiliary hypergraph is randomly generated, repeatedly, until it contains no cycle. Since such a subprocedure has an infinite worst-case time, we impose a maximum number cr^2 of repetitions, for some $c > 0$, so that the maximum time spent running the random hypergraphs method is cr^2 times the $O(r)$ cost of the acyclicity test in each repetition, yielding an $O(r^3)$ time in the worst case. If ever our limit is exceeded, we fallback to a deterministic method based on the following result. Given a set $S \subseteq [u]$ with size r, there exists a prime $q < r^2 \log u$ such that the function $\zeta : x \mapsto x \bmod q$ is perfect (injective) for the set S [8]. Thus, we obtain a perfect hash function by computing q via exhaustive search in $O(rq) = O(r^3 \log u)$ time. It is reasonable to neglect $\log u$ as a constant, so we have an $O(r^3)$ worst-case time for the deterministic phase as well. Consequently, the two phases of the perfect rehash procedure combined run in $O(r^3) = O\left(n^3/(\log\log n)^3\right)$ time in the worst case. The drawback of requiring such a deterministic fallback phase is that the size of the hash table undergoing a perfect rehash may have to grow from $\Theta\left(n/\log\log n\right)$ to $O\left(n^2/(\log\log n)^2\right)$, as possibly required by the deterministic perfect hashing method [8] depending on the value of q. That should not be a problem in many practical cases.[3]

Note that, because of perfect rehashing, the lookup procedure must be modified so we look up a key in a table using its underlying hash function and, if not found, its *underlying perfect hash function* as well. Indeed, we do not want to *replace* the original hash function with the perfect hash function because, while the latter function is perfect for the keys that were *already there* when we last rehashed that table, it is not guaranteed to be universal—which is important for new keys that may still be added. Moreover, because the current universal hash function proved to be unable to accommodate all keys in the table being rehashed, we take the time to obtain a new universal hash function for that

[3]If memory turns out to be an issue, one may want use a different, less simple perfect hashing approach, as in [8], to keep the size of the rehashed table $O\left(n/\log\log n\right)$.

table, anticipating possible attempts of accommodating that same set of keys in that very table (e.g, after some keys have been evicted/deleted and reinserted).

2.1.1. Perfect Rehash Amortized Cost

We now argue that the amortized cost of a (perfect) rehash is negligible in the grand scheme of things, as is the case in traditional hash schemes [1, 2, 9, 10]. Let again $r = O(n/\log\log n)$ denote the number of keys we are performing a perfect rehash on. Note that our perfect rehash procedure may either run entirely via the random hypergraphs method, whose expected time would already be $O(r)$ (see [7]) even if we were not to interrupt it after cr^2 iterations (as explained in Section 2.1), or require a deterministic, cubic-time phase if that cr^2 limit is reached. Let B denote the number of iterations that would be run had we not imposed any limit. Since the expectation $E[B] = O(1)$ (please refer to [7] here again), by employing the Markov inequality we obtain

$$Pr[B > cr^2] \leq \frac{E[B]}{cr^2} = O\left(\frac{1}{r^2}\right),$$

an upper bound to the probability that we require the deterministic fallback during a perfect rehash. The amortized time of the deterministic method (which can be regarded as an average of its execution times taken over all executions of the perfect rehash procedure) is therefore $O(1/r^2) \cdot O(r^3) = O(r)$.

Summing up the contributions of those distinct phases (random hypergraphs and deterministic), we still obtain $O(r) = O(n/\log\log n)$ as the expected time overall for a perfect rehash. Now, since the probability that a rehash is required is $O(1/n^2)$ (see [1, 2]), its amortized time becomes $O(n/\log\log n) \cdot O(1/n^2) = O\left(1/(n\log\log n\right)$, which is dominated by the average $O(\log\log n)$ complexity of the (necessary) lookup that precedes each insertion—hence negligible.

3. CHPR insertion and lookup analysis

For the lookup operation, the average-case time for present keys is

$$\Theta\left(\frac{1}{\log\log n}\sum_{i=1}^{\log\log n} i\right) = \Theta\left(\frac{\log\log n + 1}{2}\right) = \Theta(\log\log n),$$

and the overall worst-case time, as well as the average-case time for absent keys, is clearly linear on the number of tables, hence also $\Theta(\log\log n)$.

For the insertion operation, we must first state that the probability of a rehash is no bigger in our scheme, where a single hash function is replaced when a rehash is performed, than the $O(1/n^2)$ rehash probability of standard cuckoo schemes [1, 2], where *all* hash functions are replaced during a rehash (since all tables are rehashed). However intuitive that may be, we present computational experiments which corroborate that in Section 4. Note also that even a weaker bound such as $O(1/n)$ for the rehash probability would suffice in our analysis.

The expected insertion time in the standard, multiple-table cuckoo hash schemes using a random walk approach is known to be $O(1)$, and it does not increase asymptotically when more tables are employed [10].[4] Because those schemes execute a rehash in expected $\Omega(n)$ time with a $O(1/n^2)$ probability [1], and our perfect rehash solution rehashes in expected $O(n/\log\log n)$ time with the same probability, the expected time for the whole insertion in the CHPR scheme is also $O(1)$, *plus* the cost of the initial lookup (to avoid duplicates), which is no longer $O(1)$ in our scheme. The initial lookup, in the CHPR scheme, runs in $\Theta(\log\log n)$ time on average, therefore yielding an overall $\Theta(\log\log n)$ average-case insertion time.

The worst-case insertion time is $O\left(n^3/(\log\log n)^3\right)$, corresponding to the case where a perfect rehash exhausted all allowed repetitions of the random hypergraphs phase and resorted to the deterministic phase, as seen in Section 2.1.

Table 1 compares our construct with other relevant data structures. The chained hashing is a hash table with linked lists for collision resolution; its analysis, as well as that of balanced binary search trees, can be found in [11].

Note that our variant is outperformed by balanced search trees only in the worst-case insertion time. Comparing to known hashing schemes, it improves the worst-case insertion time of the traditional cuckoo hashing and the worst-case lookup time of the chained hashing.

[4]In practice, it actually decreases marginally.

	Worst case		Average case	
	Lookup	Insertion	Lookup	Insertion
BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
CH	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(1)$
CCH	$O(1)$	∞	$O(1)$	$O(1)$
CHPR	$\Theta(\log \log n)$	$\Theta\left(\frac{n^3}{(\log \log n)^3}\right)$	$\Theta(\log \log n)$	$\Theta(\log \log n)$

Table 1: The time complexity of basic operations for Balanced Search Trees (BST), the traditional Chained Hashing scheme (CH), the Classic Cuckoo Hashing (CCH), and our CHPR.

145 4. Experiments

We designed some experiments to complement the theoretical results summarized in Table 1. Both the Python code and the output data for average-case and worst-case behavior are available in GitHub[5]. We refrained from including the experimental results in the text body owing to size constraints. We made sure to include, though, the output of the experiment that corroborated that the probability that a perfect rehash is called for given that a previous perfect rehash has already been performed remains the same regardless of whether we replace a single hash function or all hash functions during the rehash (see Figure 1). For each value of the number of keys n, the experiment was repeated 200 times in order to obtain the average number of insertions between the first and the second rehash. The structure is first populated with n keys, then the experiment begins and every insertion is preceded by the deletion of a random key so the load factor is unchanged. The load factor was set as 0.5, and the maximum number of iterations before a perfect rehash is triggered was set as $3 \log_2 n$, based on the standard limit of iterations of the cuckoo hashing [1, 9]. For $j \in \{1, \ldots, \delta(n)\}$, our universal hash function h_j was defined as

$$h_j(k) = \big((a_j k^2 + b_j k + c_j) \bmod p\big) \bmod m_j$$

[5]https://github.com/judismar/chpr

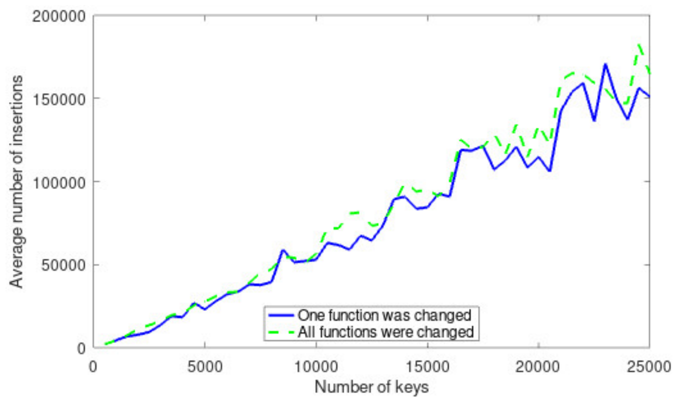

Figure 1: The average number of insertions until the second rehash is called for, after the first rehash has been performed.

for some $a_j, b_j, c_j \in \{1, 2, \ldots, p\}$ chosen independently and uniformly at random, where $p = 10^7 + 19$ is the smallest prime greater than the maximum possible key $u = 10^7$, and m_j is the size of the j-th table. Actually, that corresponds to the universal function class template given in [5].

5. Conclusions

With the perfect rehashing approach, all cuckoo hashing operations become predictable and finite. We showed that our variant improves some aspects of the traditional cuckoo hashing, chained hashing and balanced binary search trees.

While the proposed CHPR scheme cannot be claimed to be an undisputed improvement over those traditional data structures (and it surely has disadvantages as well as advantages when compared to each one of them), it may suit well applications where a highly competitive average-case performance for lookup and insertion operations is desired, without prescinding from a predictable, polynomial bound for their worst-case behavior.

A natural open problem is to improve the $O(n^3/(\log\log n)^3)$ bound for the worst-case insertion time, but that would probably require better perfect hashing methods with finite worst-case performance.

References

[1] R. Pagh, F. F. Rodler, Cuckoo hashing, Journal of Algorithms 51 (2) (2004) 122–144.

[2] D. Fotakis, R. Pagh, P. Sanders, P. Spirakis, Space efficient hash tables with worst case constant access time, Theory of Computing Systems 38 (2005) 229–248.

[3] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis, Cambridge University Press, 2017.

[4] J. L. Carter, M. N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences 18 (2) (1979) 143–154.

[5] M. Dietzfelbinger, U. Schellbach, On risks of using cuckoo hashing with simple universal hash classes, in: Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms, SIAM, 2009, pp. 795–804.

[6] Z. J. Czech, G. Havas, B. S. Majewski, Perfect hashing, Theoretical Computer Science 182 (1–2) (1997) 1–143.

[7] B. S. Majewski, N. C. Wormald, G. Havas, Z. J. Czech, A family of perfect hashing methods, The Computer Journal 39 (6) (1996) 547–554.

[8] M. L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with o(1) worst case access time, Journal of the Association for Computing Machinery 31 (3) (1984) 538–544.

[9] L. Devroye, P. Morin, Cuckoo hashing: Further analysis, Information Processing Letters 86 (4) (2003) 215–219.

[10] A. Frieze, T. Johansson, On the insertion time of random walk cuckoo hashing, Random Structures and Algorithms 54 (4) (2019) 721–729.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2009.