A cuckoo hashing scheme with bounded insertion time

Judismar Arpini Junior[a], Vinícius Gusmão Pereira de Sá[b]

[a] *Colégio Pedro II*
[b] *Universidade Federal do Rio de Janeiro*

Abstract

The time to insert a key in the classic cuckoo hashing scheme is a random variable that may assume arbitrarily big values, owing to a strictly positive probability that any (finite) long sequence of rehashes takes place. In other words, its worst-case execution time is unbounded. We propose a cuckoo hashing variant in which the worst-case insertion time is polynomially bounded, while retaining competitive time complexities for lookup, deletion and average-case insertion.

Keywords: hashing, data structures, computational complexity, algorithms

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 there is some key k_2 occupying position p_1 in T_1, then k_2 is removed from T_1 and reinserted in table T_2 at position $p_2 = h_2(k_2)$, possibly evicting some other key. Such a procedure goes on recursively, reinserting in the next table, in circular fashion, a key that has just been evicted, 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 positions given by function $h_j(k)$ on table T_j, for all $j \in \{1, \ldots, d\}$.

Unfortunately, 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 a new family of hash functions. The new functions, though, might also fail with strictly positive probability. Thus, the worst-case insertion time of cuckoo hashing is *unbounded*, i.e. the algorithm might run for an arbitrarily long time. While values of d greater than 2—the classic value—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 finite. To accomplish that, we introduce the *perfect rehash* idea, where every rehash operation is executed in collision-free fashion. Our Cuckoo Hashing with Perfect Rehash scheme (CHPR) does not use a constant number d of tables, but rather a function $d = \delta(n)$ of the number n of keys. The worst-case time for lookups is still linear in the number of tables, as well as the average insertion time, both $O(\delta(n))$. The smallest the number of tables, though, the higher the insertion time in the worst case, which we prove to be $O\left((n/\delta(n))^3\right)$. By choosing an appropriate $\delta(n) = o(\log n)$, we can optimize our worst-case insertion time while still guaranteeing that our scheme will be more efficient, with respect to lookups and average-case insertions, than balanced binary search trees or any other known data structure with bounded insertion time (see Section 4). 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.

In this text, all hash functions are considered independent and uniform. This assumption is reasonable thanks to universal hash functions [3], a technique whose application to cuckoo hashing has been discussed thoroughly [4]. We also assume that reading and writing in a given position of an array (the so-called "table") takes $O(1)$ time.

2. A cuckoo scheme with bounded insertion time

In the proposed CHPR scheme, we first choose some function $\delta(n)$, which will determine the number of tables to be used as a function of the intended number n of keys to be stored.[1] The size of each table is $\lceil (1+\epsilon)n/\delta(n) \rceil$, for some small $\epsilon > 0$.

To look up a key k, we check all its candidate locations (one per table), in traditional cuckoo style. To insert a key k, we first look it up to avoid duplicates, and then we use a random walk approach, selecting a table T_j uniformly at random, $1 \leq j \leq \delta(n)$, among those having at least one empty slot, and inserting k at position $h_j(k)$ in T_j. If some other key is found in $h_j(k)$, then that pre-existing key is evicted, in which 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 [5], 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* [5] method can be applied to avoid collisions. That is precisely the case when rehashing the $r = O(n/\delta(n))$ keys that resided in one of the tables constituting our cuckoo scheme plus the last evicted key (waiting to be reinserted in that same table during the rehash). We are particularly interested in the *random hypergraphs* perfect hashing method [6], 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 unbounded 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 an internal 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. Let $[u] = \{1, 2, \ldots, u\}$ be the universe of all possible keys, for some integer u. 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) with respect to the set S [7]. Thus, we can 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 $O(r^3) = O\left((n/\delta(n))^3\right)$ 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 $\Theta(n/\delta(n))$ to $O\left((n/(\delta(n))^2\right)$, as possibly required by the deterministic perfect hashing method [7].[2]

Note that, because of the perfect rehash, the lookup procedure must be modified so we look up a key in a table using its associated 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 thereafter. Moreover, because the current (universal) hash function proved to be unable to accommodate all keys in the table being rehashed, we seize the rehash opportunity to also choose a new (universal) hash function associated to that table, anticipating possible attempts of accommodating that same set of keys in that very table (e.g, if ever those keys happen to be reinserted in that table after having been evicted or deleted.).

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, 8, 9]. Let again $r = O(n/\delta(n))$ 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

[1] If it turns out that our structure must hold 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, without implying any change in the asymptotic complexities.

[2] If memory happens to be an issue, one may want to use a different, albeit less simple perfect hashing approach, as in [7], to keep the size of the rehashed table $O(n/\delta(n))$.

expected time would already be $O(r)$ (see [6]) 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 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 [6] 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 $O(r) = O(n/\delta(n))$ 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 $O(n/\delta(n)) \cdot O(1/n^2) = O(1/(n\delta(n))$, which is dominated by the average $O(\delta(n))$ complexity of the (necessary) lookup that precedes each insertion, hence negligible.

3. CHPR average and worst-case analysis

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

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

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 $\Theta(\delta(n))$. The average and worst-case time complexities of deleting a key are precisely the same as those of the lookup operation, since we delete a key in the CHPR by simply erasing it (i.e., by overwriting it with a null or other tombstone) from wherever it happens to be stored.

For the insertion operation, we must first state that the probability of a rehash is no greater 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 might 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 [9]. Because those schemes execute a rehash in expected $\Omega(n)$ time with a $O(1/n^2)$ probability [1], and our perfect rehash solution executes a rehash in a cheaper expected $O(n/\delta(n))$ time with the same probability, it follows that the expected time for the whole insertion in the CHPR scheme must also be $O(1)$ *plus* the cost of the initial lookup (to avoid duplicates). The cost of a lookup in our scheme, however, is no longer $O(1)$, but $\Theta(\delta(n))$ time on average, therefore yielding an overall $\Theta(\delta(n))$ average-case insertion time.

The worst-case insertion time is $O\left((n/\delta(n))^3\right)$, 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 [10].

When we make $\delta(n) = o(\log n)$, as mentioned in Section 1, note that our variant is outperformed by balanced binary search trees only in the worst-case insertion time. When compared to known hashing schemes, it improves the worst-case insertion time of the traditional cuckoo hashing and the worst-case lookup time of chained hashing, something that is—possibly, depending on the choice of $\delta(n)$—counterbalanced by less efficient average-case times. As usual, practical applications and the priorities they assign to each operation (and whether one is more concerned about average or worst-case behaviors) should dictate the relevant suitability of each solution.

	Lookup		Insertion	
	Average	Worst Case	Average	Worst Case
BBST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
CH	$O(1)$	$O(n)$	$O(1)$	$O(n)$
CCH	$O(1)$	$O(1)$	$O(1)$	∞
CHPR	$O(\delta(n))$	$O(\delta(n))$	$O(\delta(n))$	$O\left((n/\delta(n))^3\right)$

Table 1: Time complexities of basic operations for balanced binary search trees search trees (BBST), the traditional chained hashing (CH), the classic cuckoo hashing (CCH), and our CHPR, where $\delta(n)$ stands for the chosen number of tables (and hash functions).

4. Experiments

All experiments were conducted on a Debian GNU/Linux 11 bullseye (x86-64) 64-bit machine, with a multicore 11th Gen Intel(R) Core(TM) i5-1135G7 CPU with 8 cores of 2.40GHz, and with 7.6 GiB of RAM memory. We used a CHPR implementation with $\delta(n) = \lfloor \log_2 \log_2 n \rfloor + 3$ tables (and hash functions). 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]. The load factor was set as 0.8, and the limit to the number of evictions before a perfect rehash took place was set as $10 \log_2 n$, based on the standard limit of iterations of the classic cuckoo hashing [1, 8]. For $j \in \{1, \ldots, \delta(n)\}$, our universal hash function h_j was defined as

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

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. That corresponds to the universal function class template given in [4]. The code for all experiments is available in GitHub.[3] Figure 1 shows the measured average times of lookup and insertion operations, as well as the worst-case time of the lookup, for a range of values of $x = \log_2 \log_2 n$, where n is the number of stored keys.

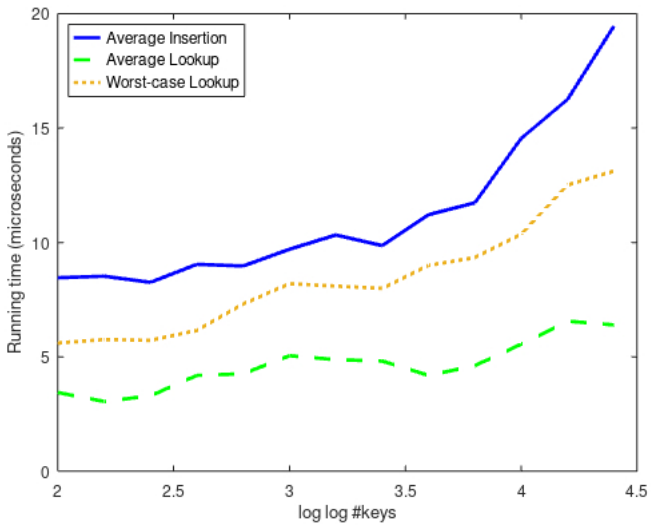

Figure 1: Running times of CHPR insertion and lookup operations as a function of $\log_2 \log_2 n$.

To obtain the average time of the lookup operation, we began by inserting a uniform random sample of size n from the universe of keys $[u] = [10^7]$. We then proceeded to lookup 10,000 keys, selected independently and uniformly at random from that universe, for every value of x from 2 to 4.4 in 0.2 increments. As for the insertion times, we started

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

from empty tables, and sequentially inserted n keys chosen independently and uniformly at random from the universe, dividing the total time by n. The entire procedure was repeated independently 100 times to obtain the averages.

To emulate the worst-case of the lookup, we searched for keys in a set with 5,000,000 absent keys. That was repeated 1,000,000 times to obtain the averages.

We did not try to emulate worst-case scenarios for the insertion operation, something that would likely require an artificial mocking of the employed perfect hashing methods. Our main contribution nevertheless lies in the fact that we can keep average lookup and insertion times (and worst-case lookup time) totally under control with an adjustable $O(\delta(n))$ bound, while guaranteeing that the worst-case time for insertion is *also* (polynomially) bounded, something we have proved analytically.

Finally, we made sure to conduct an experiment to corroborate that the probability that a (perfect) rehash is required in the CHPR after some (perfect) rehash has already been performed remains the same regardless of whether we replace a *single* hash function (as advocated by our CHPR) or all hash functions (in classic cuckoo fashion) during a rehash, as anticipated in Section 3. 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 was first populated with n keys before the experiment began, and every insertion was preceded by the deletion of a random key so the load factor would remain unchanged. The experimental results matched what we expected, as no significant difference was observed throughout (see Figure 2).

5. Conclusions

With the perfect rehash approach, the worst-case of every dictionary operation in our cuckoo hashing variant becomes predictable. Preventing unbounded worst-case times is not new in the literature, e.g. a Las Vegas randomized algorithm can be converted into a Monte Carlo algorithm to yield a finite worst-case behavior [11].

We showed that our variant improves some aspects of the traditional cuckoo hashing, chained hashing and balanced binary search trees. All theoretical time complexities were verified experimentally. 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 well-behaved polynomial bound for insertions.

A natural open problem is to improve the $O\left((n/\delta(n))^3\right)$ bound for the worst-case insertion time, which probably requires more efficient perfect hashing methods (with bounded worst-case performance).

References

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

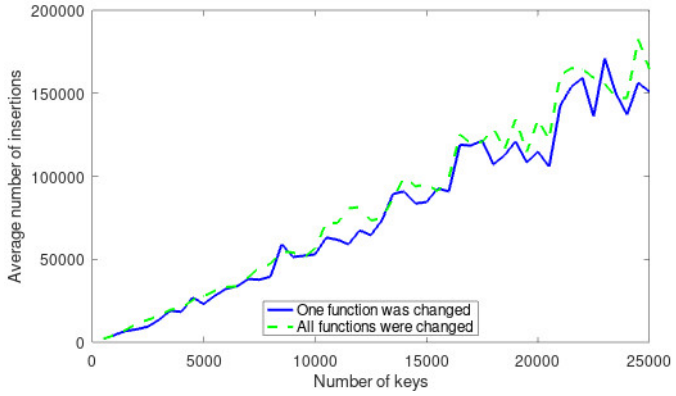

Figure 2: Changing just one of the hash functions versus changing all functions during a rehash: the average number of insertions before the second rehash was called for, after the first rehash has been performed.

[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] J. L. Carter, M. N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences 18 (2) (1979) 143–154.
[4] 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.
[5] Z. J. Czech, G. Havas, B. S. Majewski, Perfect hashing, Theoretical Computer Science 182 (1–2) (1997) 1–143.
[6] 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.
[7] 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.
[8] L. Devroye, P. Morin, Cuckoo hashing: Further analysis, Information Processing Letters 86 (4) (2003) 215–219.
[9] A. Frieze, T. Johansson, On the insertion time of random walk cuckoo hashing, Random Structures and Algorithms 54 (4) (2019) 721–729.
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2009.
[11] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis, Cambridge University Press, 2017.