

Universidade Federal do Rio de Janeiro
Instituto de Matemática
Programa de Pós-Graduação em Informática

Judismar Arpini Júnior

New Multiple-Choice Hashing Schemes with Theoretical and Practical Contributions

Rio de Janeiro

2022

Judismar Arpini Júnior

New Multiple-Choice Hashing Schemes with Theoretical and Practical Contributions

Doctoral Thesis submitted to the Programa de Pós-Graduação em Informática at the Instituto de Matemática and the Instituto Tércio Pacitti of the Universidade Federal do Rio de Janeiro - UFRJ, as part of the necessary requirements for obtaining the degree of Doctor in Computer Science.

Universidade Federal do Rio de Janeiro – UFRJ

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

Programa de Pós-Graduação em Informática

Supervisor: Vinícius Gusmão Pereira de Sá, D.Sc.

Rio de Janeiro

2022

CIP - Catalogação na Publicação

A772n Arpini Júnior, Judismar
New multiple-choice hashing schemes with
theoretical and practical contributions / Judismar
Arpini Júnior. -- Rio de Janeiro, 2022.
78 f.

Orientador: Vinícius Gusmão Pereira de Sá.
Tese (doutorado) - Universidade Federal do Rio
de Janeiro, Instituto Tércio Pacitti de Aplicações e
Pesquisas Computacionais, Programa de Pós-Graduação
em informática, 2022.

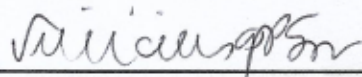
1. Hashing. 2. Data structures. 3. Concurrent
hashing. I. Sá, Vinícius Gusmão Pereira de, orient.
II. Título.

Judismar Arpini Júnior

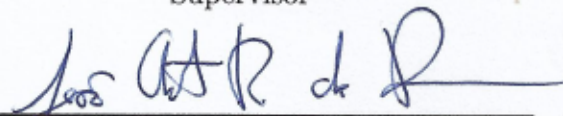
New Multiple-Choice Hashing Schemes with Theoretical and Practical Contributions

Doctoral Thesis submitted to the Programa de Pós-Graduação em Informática at the Instituto de Matemática and the Instituto Tércio Pacitti of the Universidade Federal do Rio de Janeiro - UFRJ, as part of the necessary requirements for obtaining the degree of Doctor in Computer Science.

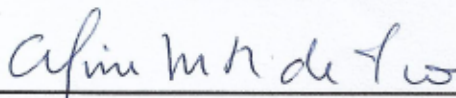
Text approved. Rio de Janeiro, Brazil, 21st of November 2022:



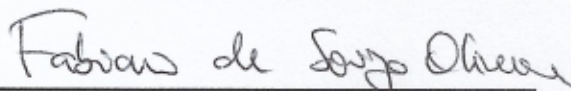
Vinícius Gusmão Pereira de Sá, D.Sc.
Supervisor



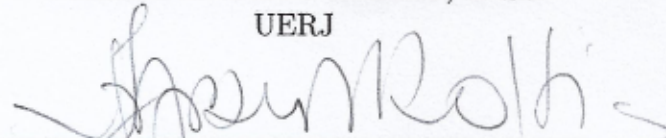
João Antonio Recio da Paixão, D.Sc.
PPGI-UFRJ



**Celina Miraglia Herrera de
Figueiredo, D.Sc.**
PESC-UFRJ



Fabiano de Souza Oliveira, D.Sc.
UERJ



Fábio Protti, D.Sc.
UFF

To my wife, Ariana. To my parents, Judismar and Marlene. To my brother, Rômulo. To my niece, Maya. To my mother-in-law, Cristina.

RESUMO

O tempo de pior caso para inserir uma chave no esquema clássico de hashing do cuco é uma variável aleatória que pode assumir valores arbitrariamente grandes, devido à probabilidade estritamente positiva de que uma sequência de rehashes sem fim aconteça. Nossa primeira contribuição é uma variante do hashing do cuco em que o tempo de pior caso de inserção é polinomial. Para conseguir isso, usamos duas ideias básicas. A primeira é empregar um método de hashing perfeito em uma das tabelas quando um rehash for necessário. A segunda ideia é fazer com que o número de tabelas hash subjacentes não seja mais constante, mas uma função apropriada do número de chaves. O preço a pagar é um tempo de busca maior, que não é mais constante na média, mas duplamente logarítmico.

Nossa segunda contribuição explora hashing de múltipla escolha para criar um esquema de hashing concorrente com tempo de pior caso $O(1)$ para operações de busca, inserção e remoção, um hash map eventualmente consistente que não usa qualquer tipo de sincronização para segurança de threads. Ele é muito bem-sucedido em cenários com um único escritor e múltiplos leitores, superando os populares `ConcurrentHashMap` de Java e o `concurrent_hash_map` do Intel TBB em C++. O preço a pagar é uma probabilidade de que uma inserção possa falhar, a qual pode ser feita pequena o suficiente para atender a aplicações práticas.

Palavras-chave: Hashing; complexidade computacional; estruturas de dados; hashing concorrente; programação multicore.

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 a strictly positive probability that an endless sequence of rehashes take place. Our first contribution is 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 on average, but doubly logarithmic.

Our second contribution exploits multiple-choice hashing to create a concurrent hashing scheme with $O(1)$ worst-case time for lookup, insert and remove operations, an eventually consistent hash map that does not use any kind of synchronization for thread-safety. It is particularly successful in scenarios with a single writer and multiple readers, where it outperforms the popular Java's `ConcurrentHashMap` and C++ Intel's TBB `concurrent_hash_map`. The price to pay is a probability that an insertion may fail, which can be made small enough to suit practical applications.

Keywords: Hashing; Computational complexity; Data structures; Concurrent hashing; Multicore programming.

LIST OF FIGURES

Figure 1 – An example of a hash table with $m = 5$ and with hash function $h(x) = x \bmod 5$	16
Figure 2 – Illustration of the hashing scheme that uses linked lists to resolve collisions.	17
Figure 3 – The graph generated by the mapping step.	21
Figure 4 – The insertion of key x in the cuckoo scheme with two tables.	23
Figure 5 – Infinite loop when inserting key x	24
Figure 6 – A cuckoo tree where the first key of each table is a split key.	31
Figure 7 – Inserting key 40 in the cuckoo tree.	32
Figure 8 – An illustration of the worst case scenario of a cuckoo tree.	33
Figure 9 – Illustration of mutual exclusion, where threads A , B and C run concurrently. The straight horizontal arrows represent time. Thread B waits between times t_1 and t_3 , and thread C waits between times t_2 and t_4	36
Figure 10 – A table that compares blocking and non-blocking progress conditions.	40
Figure 11 – Illustration of quiescent periods, where threads A , B and C run concurrently. The straight horizontal arrows represent time. The thick segments represent method or function calls, and the red rectangles delimit the quiescent periods.	41
Figure 12 – Illustration of a concurrent execution of operations, where threads A , B and C run concurrently. The straight horizontal arrows represent time. The thick black segments represent method or function calls. The vertical red lines represent the linearization points of each call in this particular execution. The calls are numbered after their invocation time.	42
Figure 13 – An instance of the adapted version of Java’s ConcurrentHashMap with $N = 16$ and $L = 8$. Each lock covers $2^{N/L}$ entries.	43
Figure 14 – The TBB concurrent_hash_map.	45
Figure 15 – The hopscotch insertion algorithm looks for a sequence of displacements of keys.	46
Figure 16 – The hopscotch insertion algorithm after the keys were displaced to create an empty entry for key v	47
Figure 17 – The optimistic cuckoo hash table.	47
Figure 18 – A cuckoo path, where \emptyset represents an empty slot.	49
Figure 19 – An illustration of the wait-free hashing scheme.	50
Figure 20 – An example of hashed keys stored in the wait-free hash map.	51

Figure 21 – Some possible setups using MHM: (a) hash map for objects; (b) hash map for primitive types; (c) hash set for objects; (d) hash set for primitive types. K, V, UA and KVOP indicate, respectively, the stored key, the mapped value, the underlying array (table), and the key-value object pair.	57
Figure 22 – The average number of insertions until the second rehash is called for, after the first rehash has been performed.	59
Figure 23 – The average time of an insertion and a lookup of the cuckoo hashing with perfect rehash in function of $\log \log n$, where n is the number of keys in the scheme.	60
Figure 24 – The time of the emulated worst-case insertion and the emulated worst-case lookup of the cuckoo hashing with perfect rehash, in function of the number of keys in the scheme.	60
Figure 25 – 25,000 inserted items.	62
Figure 26 – 100,000 inserted items.	62
Figure 27 – 25,000 inserted items.	62
Figure 28 – 100,000 inserted items.	62
Figure 29 – 25,000 inserted items.	63
Figure 30 – 100,000 inserted items.	63
Figure 31 – A graphical representation of a concurrent execution of a FIFO queue. A and B are threads, and the return value of the dequeue operation is given after the colon.	72
Figure 32 – A graphical representation of a concurrent execution of a FIFO queue. A and B are threads, and the return value of the dequeue operation is given after the colon.	74

LIST OF TABLES

Table 1	– Assignment of integer values in range $[0, 9]$ for each edge.	21
Table 2	– The table of $g(v)$ values, $v \in V$	22
Table 3	– 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.	30
Table 4	– The time complexity of basic operations for Balanced Search Trees (BST), the traditional Chained Hashing scheme (CH), the Classic Cuckoo Hashing (CCH), and the preliminary CHPR.	30

CONTENTS

1	INTRODUCTION	13
1.1	A CUCKOO HASHING VARIANT	14
1.2	A HIGH-PERFORMANCE, THREAD-SAFE HASH TABLE	14
1.3	OBJECTIVES	14
2	HASHING	16
2.1	UNIVERSAL HASH FUNCTIONS	17
2.2	PERFECT HASHING	18
2.2.1	Random Hypergraphs Method	18
2.2.1.1	Mapping Step	18
2.2.1.2	Graph Assignment Step	19
2.2.1.3	An Example	20
2.2.2	Prime Number Method	22
2.3	CUCKOO HASHING	23
2.3.1	Insertion	23
2.3.2	Load Thresholds	25
2.3.3	Time Complexity	25
2.3.3.1	Worst-Case Insertion Time	25
3	CUCKOO HASHING WITH PERFECT REHASH	26
3.1	PERFECT REHASH	27
3.1.1	Modified Lookup	28
3.1.2	Perfect Rehash Amortized Cost	28
3.2	INSERTION AND LOOKUP ANALYSIS	29
3.3	PRELIMINARY WORK: THE FIRST VERSION	30
3.4	THE CUCKOO TREE: AN OPEN PROBLEM	30
4	CONCURRENT HASHING	34
4.1	SHARED DATA AND SYNCHRONIZATION	34
4.1.1	Shared Hash Table Issues	35
4.2	MUTUAL EXCLUSION	35
4.2.1	Mutual Exclusion Overhead	36
4.2.2	Deadlock and Starvation	37
4.3	NON-BLOCKING ALGORITHMS	38
4.3.1	Lock-Freedom and Wait-Freedom	39
4.4	CONSISTENCY	39

4.4.1	Quiescent Consistency	40
4.4.2	Linearizability	41
4.4.3	Eventual Consistency	41
4.5	SOME CONCURRENT HASHING SCHEMES	42
4.5.1	Java ConcurrentHashMap	42
4.5.1.1	The Adapted Version	42
4.5.1.2	The Current Version	44
4.5.2	TBB concurrent_hash_map	45
4.5.3	Hopscotch Hashing	46
4.5.4	Optimistic Cuckoo Hashing	47
4.5.4.1	Tags	48
4.5.4.2	Insertion	48
4.5.4.3	Lookup	48
4.5.5	A Wait-Free Hashing Scheme	49
4.5.5.1	Traversing the Multi-Level Array	50
4.5.5.2	Insertion	51
4.5.5.3	Disadvantages	51
4.5.6	Other Concurrent Hashing Schemes	52
5	MONKEY HASHING	53
5.1	THREAD-SAFETY AND EVENTUAL CONSISTENCY	55
5.2	INSERTION FAILURE PROBABILITY	56
5.2.1	Working with Different Load Factors	57
5.3	RESIZABLE MONKEY HASHING	57
6	EXPERIMENTS	59
6.1	CHPR EXPERIMENTS	59
6.2	MH EXPERIMENTS	61
7	CONCLUSION	64
	BIBLIOGRAPHY	66
	APPENDIX	70
	APPENDIX A – THE MATHEMATICAL DEFINITION OF LINEARIZABILITY	71
A.1	THE FORMAL MODEL OF A CONCURRENT EXECUTION	71
A.2	THE FORMAL DEFINITION OF LINEARIZABILITY	73
A.3	EXAMPLES	74

APPENDIX B – MONKEY HASH SET VARIANT THAT SUPPORTS MULTIPLE WRITER THREADS 76

B.1	THE CAS INSTRUCTION	76
B.2	THE VARIANT	76
B.2.1	The Lookup Algorithm	77
B.2.2	The Insert Algorithm	77
B.2.3	The Remove Algorithm	78

1 INTRODUCTION

Hashing-based data structures have enormous importance in computing, starring in a number of efficient algorithms for a variety of problems (1, 2, 3, 4, 5). The basic intent consists of storing *keys* (unique identifiers), sometimes mapped to *values*, to be retrieved at some later time. Dictionary operations are made efficient by determining the position (*bucket*) in an underlying array (*table*) where the key (and its associated value object, if any) will be stored as a function of the key itself.

Our work is based in multiple-choice hashing, where each key can be placed in one of at least two distinct positions in the hash table. The performance of a hashing scheme can be drastically improved when we allow each key to have alternative storing places (6, 7).

Cuckoo hashing (8) is a variant of multiple-choice hashing, with constant worst-case lookup time and constant amortized insertion time. Its drawback is its worst-case insertion time, which is infinite.

Not too long ago, advances in computer hardware usually meant advances in clock speed, so all existing software would speed up by itself over time. In recent times, manufacturers have been leaning towards architectures with multiple communicating processors, increasing the ability to execute parallel tasks first and foremost, with only modest increases in clock speed. Those systems are called *shared-memory multiprocessors*, or *multicores* (9), where multiple concurrent threads must coordinate their operations on some shared memory.

This text is divided into two parts. In the first part, we create a cuckoo hashing variant aiming at a scenario that prioritizes lookup performance and where a finite worst-case insertion time is required. Our approach is mostly theoretical, comparing our variant with well-known hashing schemes and with balanced search trees. In the second part, we exploit multiple-choice hashing to create a simple, high-performance and eventually consistent hashing scheme, one that does not use locks or any kind of synchronization. Although this novel scheme has interesting theoretical aspects, our motivation is very practical, using experiments to show it consistently outperforms popular solutions such as Java's `ConcurrentHashMap` and C++ Intel's `TBB concurrent_hash_map` in relevant scenarios.

In this text, we let n denote the number of stored keys, and we let $[u] = \{0, 1, \dots, u-1\}$ be the universe of keys, for some integer $u \geq n$. For all intents and purposes, reading and writing in an array takes $O(1)$ time.

1.1 A Cuckoo Hashing Variant

In cuckoo hashing, the worst-case insertion time is *infinite*: the probability that an insertion executes in arbitrarily large amounts of time is strictly positive.

We tackle this boundless insertion time problem by proposing a new version of the *cuckoo hashing with perfect rehash* (CHPR), first presented in (10). We introduce the *perfect rehash*, a technique based on perfect hashing (11) that ensures a bounded worst-case behavior. 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 (12).

Unlike balanced search trees, hashing schemes do not preserve the order of its elements. Nevertheless, when using a balanced search tree instead of a hashing approach, we are not necessarily wanting to traverse the elements in order, but we may be interested in avoiding a linear-time worst-case performance. In that sense, we compare the theoretical times of their basic operations to show the advantages of the CHPR.

The first version of the CHPR in (10) has its advantages when compared to other hashing schemes, but as we shall see, it is outperformed by balanced search trees in every aspect. The version proposed here, however, improves the average-case time of all dictionary operations and the worst-case time of all those operations except for the insertion which remains polynomial nonetheless.

1.2 A High-Performance, Thread-Safe Hash Table

In a scenario where a single writer thread and multiple reader threads share access to a hash table, in the absence of mutual exclusion locks, some problems with non-atomic operations arise (9). Locking a block of code means that threads wait for arbitrary amounts of time, as will be seen in Section 4.2. Such waiting is unacceptable for certain applications and systems, where a high-performance, wait-free solution is of interest (13).

In this context, we exploit multiple-choice hashing to create the *monkey hashing* (MH), a hashing scheme that does not use locks or any kind of synchronization, where every operation runs in $O(1)$ worst-case time. The price to pay is a small probability that an insertion might fail. As we shall see, we can make the upper bound on such probability to be so small that it becomes negligible to all imaginable applications.

1.3 Objectives

Our two main objectives are the following:

- to propose a cuckoo hashing variant that improves the worst-case insertion time of cuckoo hashing, and that also improves some theoretical aspects of traditional data structures;
- to present a novel concurrent hash map that does not require locks or synchronization mechanisms and that outperforms widely used concurrent hash tables in 2 of the top 4 most popular programming languages as of the writing of this thesis (Java and C++) (14).

The general objectives can be decomposed in the following specific objectives:

- to expose our cuckoo hashing variant, the cuckoo hashing with perfect rehash;
- to obtain the time complexity of the insertion and lookup of this variant;
- to compare these complexities with those of balanced search trees, traditional chained hashing, and classic cuckoo hashing;
- to design experiments that corroborate the time complexities;
- to introduce our concurrent hash map, the monkey hashing;
- to obtain an upper bound on its insertion failure probability and argue that it is good enough in practice;
- to run experiments in Java and C++, comparing the monkey hashing with concurrent hashing solutions such as Java's `ConcurrentHashMap`, C++ Intel's `concurrent_hash_map` and the high-performance hash map by Laborde et al.

2 HASHING

A *hash function* is a function $h : S \mapsto M$, which maps elements from a set S of n *keys* into the integer range $M = \{0, 1, \dots, m - 1\}$. The *hash values* from the codomain M are in general associated with a structure to store data, like a list or an array.

Given a key $x \in S$, the hash function returns an address (index), i.e. an integer in $\{0, 1, \dots, m - 1\}$ to store and retrieve x in the storage area, which we call *hash table*. The computation of this function is performed in $O(1)$ time under reasonable assumptions. Figure 1 shows an example of a hashing scheme.

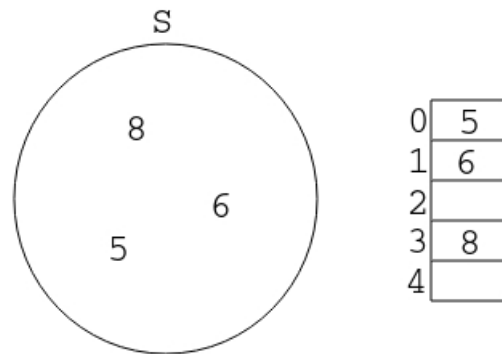


Figure 1 – An example of a hash table with $m = 5$ and with hash function $h(x) = x \bmod 5$.

Although we will be considering keys as positive integers, the domain of the hash function can be, for example, a set of strings. Each symbol of the alphabet is thus associated with a previously defined numerical value, making it possible to compute the index associated with the string. We note that, in order to compute the hash function in $O(1)$ time, the domain must be bounded. In the case of a set of integers, each key has a maximum size bounded by a constant (in general the number of bits used to represent the integer in a programming language). In the case of a set of strings, we must set a maximum size.

There exists the case where two different keys are mapped to the same index of the table. We call this event a *collision*. A common way to resolve collisions is to create a linked list of keys in each entry of the table (15). Thus, to access the data associated with a key, it is necessary to compute its address in the table and to walk through the list, as illustrated in Figure 2.

In the worst case, all keys collide, yielding a list of size n . In the average case, when the hash function is uniform and the load of the table is suitable, looking up and inserting a key takes $O(1)$ time (15).

We define the *load factor* of the hash table as $\alpha = n/m$.

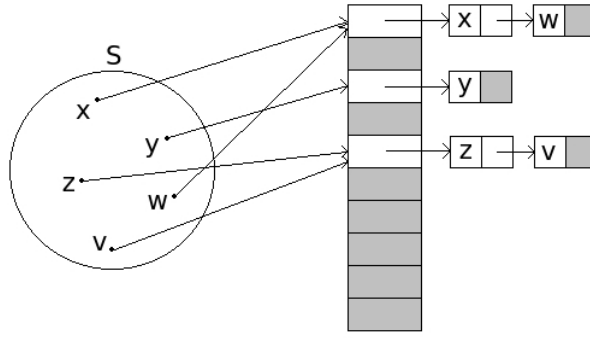


Figure 2 – Illustration of the hashing scheme that uses linked lists to resolve collisions.

Note that the memory allocated for the underlying array is given by a function of the provided number of keys to be inserted and the *intended load factor*. In case the load factor exceeds a settled maximum value, it is anyway necessary to reallocate the keys in a larger underlying array, in a way to lower the load factor — and with it the complexity of basic operations — to the desired level.

2.1 Universal Hash Functions

Hash functions may have high collision rates. We want to choose functions that perform well for any set of keys S . It is convenient to assume the existence of random functions mapping each key independently and uniformly to the range of hash table indexes (and this is done for theoretical purposes). However, implementing this idea would require an exponential number of bits (16).

To obtain hash functions that work well in practice, we use *universal classes of hash functions* (17). A universal class guarantees that, when obtaining a hash function to work with any set of keys S , it is good on average. In fact, such functions work as well as idealized uniform hash functions on average (16).

We are interested in the universal hash function family of polynomials (17). Let p be a prime number such that $p > x$ for every $x \in S$. For a fixed $d \geq 2$, we obtain $a_0, \dots, a_{d-1} \in [p]$ and define our hash function as

$$h(x) = ((a_0 + a_1x + \dots + a_{d-1}x^{d-1}) \bmod p) \bmod m.$$

The prime number may be informed beforehand, but may also be computed efficiently using, for example, Rabin's randomized algorithm for primality (18).

An example of hash function for $S = \{8, 14, 5, 10\}$ and $d = 3$ is $h(x) = ((14x^2 + 7x + 6) \bmod 17)$. Hence, the hash values are $h(8) = 2$, $h(14) = 1$, $h(5) = 0$, $h(10) = 2$. Once the values of m and p are set, a function is obtained by choosing constants a_0, a_1, a_2 . Distinct constants yield distinct hash functions from the same universal family.

2.2 Perfect Hashing

When the set of keys we are storing is known beforehand, we may use a *perfect hashing* method to find a function that does not yield collisions (11). Such injective function is called a *perfect hash function*.

We are interested in two perfect hashing methods: the method of random hypergraphs (19) and a prime number method described in (20).

2.2.1 Random Hypergraphs Method

The random hypergraphs method uses the concept of *generalized hypergraphs*, or *r-graphs*. Every edge of an *r-graph* is a set of r vertices. For every value assigned to r , we have a different method. We thus have a family of random hypergraphs methods.

A 1-graph is equivalent to a simple graph with no edges. A 2-graph is equivalent to a simple graph. For simplicity, we set $r = 2$ and work with simple graphs. We then refer to the method as *random graphs method*. To study the entire family, see (19) and (11).

The algorithm is divided into two steps, where the first is randomized. The running time is thus a random variable. The total expected running time is $O(n)$.

The hash function computed by the algorithm is of the form

$$h(x) = (g(f_1(x)) + g(f_2(x))) \bmod m,$$

where f_1 and f_2 are functions that map the set of keys S into $V = [\nu]$ for some integer $\nu > 0$, and g maps V to M .

2.2.1.1 Mapping Step

The first step of the random graphs method is to map the n keys into an acyclic graph of μ edges and ν vertices. The graph is constructed in an edge oriented fashion, where $\mu = n$ and $\nu = 2.09\mu$ (the choice of constant 2.09 is justified in (19)). Every key is mapped to an edge $\{f_1(x), f_2(x)\}$.

The random graphs method is originally used in an universe of strings. Since the universe we work with in this text is a set of positive integers, we use universal hashing (seen in Section 2.1) to execute the mapping step, as suggested in (19).

The mapping step is a random search algorithm. We obtain the functions f_1 and f_2 uniformly at random from an universal family of hash functions, in order to generate the graph from the keys of S . If the graph is cyclic, we obtain new functions at random from the family.

In order to test if the graph is acyclic, we use a special case of the algorithm described in (19). It runs in $O(\mu + \nu) = O(n)$ time. The graph can be represented by an adjacency list.

Scan all vertices, each vertex only once. If vertex v has degree 1 then remove the incident edge. When edge e is removed, check if the other vertex incident to it now has degree 1. If so, remove the only edge incident to it. Repeat this until it is not possible to remove (i.e. the other vertex incident to the removed edge does not have degree 1). After all vertices are scanned, check if the graph contain edges. If so, the graph failed the acyclicity test.

When the graph is acyclic, the reverse order in which the edges were removed are necessary in the next step. A stack can be used to store each edge in the moment it is removed.

2.2.1.2 Graph Assignment Step

The assignment problem is defined as follows. Let $G = (V, E)$ be a graph with $|E| = \mu$, $|V| = \nu$. Find the function $g : V \mapsto [\mu]$ such that function $h : E \mapsto [\mu]$ defined for each $e \in E$, $e = \{v_1, v_2\}$, as

$$h : e \mapsto (g(v_1) + g(v_2)) \bmod \mu$$

is a bijection. In other words, we must find an assignment of values to vertices such that, for each edge, the sum of the values associated to its vertices *modulo* the number of edges is a unique integer in the range $[\mu]$.

To obtain a solution to this problem, we proceed as follows. Associate with each edge a unique number $h(e) \in [\mu]$ in any order. Consider the edges in reverse order to the order of deletion in the acyclic test; to that end, use the stack built in that step. Define each vertex as yet unassigned. Each edge $\{v_1, v_2\}$, in the time it is considered, has at least one incident unassigned vertex. When two vertices v_1 and v_2 are unassigned, set $g(v_1) \leftarrow 0$; then, set $g(v_2) \leftarrow (h(e) - g(v_1)) \bmod \mu$. If one vertex v_1 is already assigned, set only $g(v_2) \leftarrow (h(e) - g(v_1)) \bmod \mu$.

The proof of correctness of this method is found in (19). Generating a minimal perfect hash function is reduced to the graph assignment problem as follows. As each edge $e = \{v_1, v_2\} \in E$ uniquely corresponds to some key x (such that $f_1(x) = v_1$ and $f_2(x) = v_2$), we set $h(e) = i - 1$ if x is the i th key of S . Then, the values of function g for each $v \in V$ are computed by the assignment step. Function h is a minimal perfect hash function for S .

2.2.1.3 An Example

Consider the set of keys $S = \{12, 67, 98, 3, 55, 112, 90, 84, 9, 44\}$. We want to obtain a perfect hash function that maps the keys to the integer range $[0, 9]$. Since $\nu = 2.09\mu$ and $\mu = n = |S| = 10$, we have $\nu = 20$.

The first step is mapping. We obtain functions f_1 and f_2 from an universal family of hash functions to compute the vertices of each edge. Hence, we have:

$$\begin{aligned} f_1(12) &= 17, f_2(12) = 3; \\ f_1(67) &= 3, f_2(67) = 0; \\ f_1(98) &= 0, f_2(98) = 17. \end{aligned}$$

Key 98 completes the cycle $(17, 3, 0)$, introduced by the edges $\{17, 3\}$, $\{3, 0\}$ and $\{0, 17\}$. Thus, the graph fails the acyclicity test and new hash functions must be obtained. The following generated graph is acyclic, and is shown in Figure 3. The image of f_1 and of f_2 is as follows:

$$\begin{aligned} f_1(12) &= 4, f_2(12) = 6; \\ f_1(67) &= 15, f_2(67) = 0; \\ f_1(98) &= 16, f_2(98) = 8; \\ f_1(3) &= 8, f_2(3) = 15; \\ f_1(55) &= 10, f_2(55) = 1; \\ f_1(112) &= 19, f_2(112) = 11; \\ f_1(90) &= 5, f_2(90) = 8; \\ f_1(84) &= 7, f_2(84) = 9; \\ f_1(9) &= 1, f_2(9) = 17; \\ f_1(44) &= 9, f_2(44) = 18. \end{aligned}$$

The stack of edges generated in the acyclicity test is $(112, 9, 55, 44, 84, 98, 90, 12, 3, 67)$.

In the assignment step we choose distinct integers in range $[0, 9]$ for each edge. Table 1 shows this assignment.

The algorithm starts with edge 112, the edge in the top of the stack $(112, 9, 55, 44, 84, 98, 90, 12, 3, 67)$. Hence, we have

$$g(11) \leftarrow 0,$$

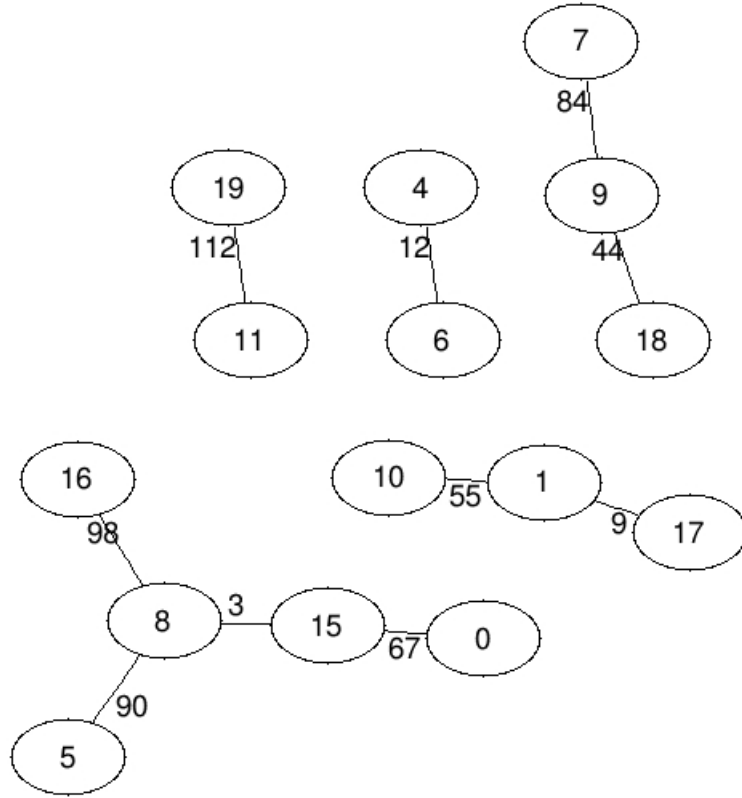


Figure 3 – The graph generated by the mapping step.

Value	0	1	2	3	4	5	6	7	8	9
Edge	12	67	98	3	55	112	90	84	9	44

Table 1 – Assignment of integer values in range $[0, 9]$ for each edge.

$$g(19) \leftarrow (5 - g(11)) \mod 10 = 5.$$

The algorithm unstacks the edge and assigns the values to its vertices. It goes on with edge 9:

$$\begin{aligned}
 g(17) &\leftarrow 0, \\
 g(1) &\leftarrow (8 - g(17)) \mod 10 = 8, \\
 g(10) &\leftarrow (4 - g(1)) \mod 10 = 6, \\
 g(18) &\leftarrow 0, \\
 g(9) &\leftarrow (9 - g(18)) \mod 10 = 9, \\
 g(7) &\leftarrow (7 - g(9)) \mod 10 = 8, \\
 g(16) &\leftarrow 0, \\
 g(8) &\leftarrow (2 - g(16)) \mod 10 = 2, \\
 g(5) &\leftarrow (6 - g(8)) \mod 10 = 4,
 \end{aligned}$$

$$\begin{aligned}
g(4) &\leftarrow 0, \\
g(6) &\leftarrow (0 - g(4)) \mod 10 = 0, \\
g(15) &\leftarrow (3 - g(8)) \mod 10 = 1, \\
g(0) &\leftarrow (1 - g(15)) \mod 10 = 0.
\end{aligned}$$

All $g(v)$ values, for $v \in V$ incident to at least one edge, are now computed. The isolated vertices are not used by the hash function, since they are not associated with any key. Table 2 presents these values.

v	0	1	4	5	6	7	8	9	10	11	15	16	17	18	19
g(v)	0	8	0	4	0	8	2	9	6	0	1	0	0	0	5

Table 2 – The table of $g(v)$ values, $v \in V$.

Now, in order to compute the address of key 112 in the hash table:

$$\begin{aligned}
h(112) &= (g(f_1(112)) + g(f_2(112))) \mod 10 = \\
&= (g(11) + g(19)) \mod 10 = (0 + 5) \mod 10 = 5.
\end{aligned}$$

2.2.2 Prime Number Method

The perfect hashing method we denote as the prime number method is based on the following result. Given a set $S \subseteq [u]$ with size n , there exists a prime $q < n^2 \log u$ such that the function $\zeta : x \mapsto x \mod q$ is perfect (injective) for the set S (20).

We obtain a perfect hash function by computing q via exhaustive search as follows. Set $q = n$ (for lower values of q , there will be collision), and hash every key of S with the hash function $x \mod q$. If a collision occurs, increment q and try again. This goes on until there are no collisions. It is guaranteed this algorithm halts before $q = n^2 \log u$.

As an example, let $u = 11$ and $S = \{5, 6, 9\}$. The algorithm starts with $q = 3$, and key 9 collides with key 6. It then goes on with $q = 4$, and key 9 collides with key 5. When $q = 5$, there are no collisions and ζ is a perfect hash function for S .

Since every iteration takes $O(n)$ time, computing the perfect hash function takes $O(nq) = O(n^3 \log u)$ time. It is reasonable to neglect $\log u$ as a constant, so the prime number method takes $O(n^3)$ time and $O(n^2)$ space in the worst case.

If memory turns out to be an issue, the reader can use this method to build a less simple perfect hashing approach denoted as the FKS scheme, with $O(n)$ space, as seen in (20) and (11).

2.3 Cuckoo Hashing

Cuckoo hashing is a hashing scheme with constant worst-case lookup time and constant amortized insertion time (8). In general terms, it consists of $d \geq 2$ tables with an underlying hash function each.

We define T_i , $1 \leq i \leq d$ as our cuckoo tables and h_i as each of their underlying hash functions. Every key x is stored in $T_k[h_k(x)]$ for $k \in \{1, \dots, d\}$, but never in more than one table. To lookup a key, we simply check $T_k[h_k(x)]$ for all $k \in \{1, \dots, d\}$. The lookup algorithm is found in Figure 2.1.

CuckooLookup (x)

1. **for** $i = 1, 2, \dots, d$
2. **if** $T_i[h_i(x)] = x$
3. **return** true
4. **return** false

Algorithm 2.1 – The cuckoo hashing lookup algorithm.

2.3.1 Insertion

When inserting a new key, the keys dislodge one another until they are all accommodated in the tables. In (8) we find Figure 4, which illustrates the insertion in a cuckoo scheme with two tables.

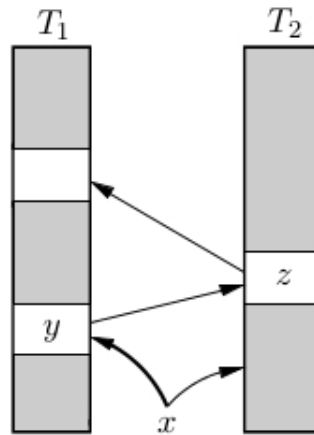
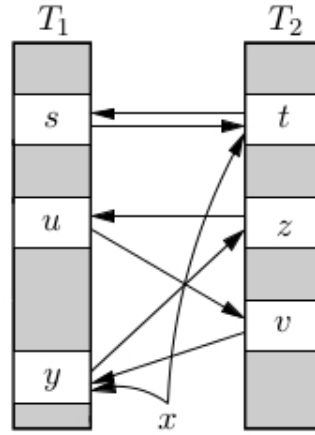


Figure 4 – The insertion of key x in the cuckoo scheme with two tables.

We are interested in a random walk approach. The insertion algorithm first looks up the key to be inserted to avoid duplicates. In the main loop, it picks a random table and insert the current evicted key there, thus evicting another key that was found in the same position. The algorithm ends when no key is evicted.

Figure 5 – Infinite loop when inserting key x .

The insertion may enter in an infinite loop, as illustrated in Figure 5, found in (8). Hence, we set $c \log n$ as the maximum number of iterations, for a suitable chosen c . After that, we declare the insertion a failure and rehash the entire scheme. The $c \log n$ as the maximum number of iterations is a good rehash criterion when the load of the scheme is low enough (8, 21). We then obtain a new hash function for every table and attempt to accommodate all keys, including the evicted one, in this new setting.

The insertion algorithm is found in Figure 2.2, where $\text{random}(d)$ returns a random value in $\{1, \dots, d\}$ and \perp denotes an empty position.

```

CuckooInsertion ( $x$ )
1.  if CuckooLookup( $x$ )
2.    return
3.  repeat  $c \log n$  times
4.     $k \leftarrow \text{random}(d)$ 
5.     $\text{temp} \leftarrow T_k[h_k(x)]$ 
6.     $T_k[h_k(x)] \leftarrow x$ 
7.     $x \leftarrow \text{temp}$ 
8.    if  $x = \perp$ 
9.      return
10. rehash()
11. CuckooInsertion( $x$ )

```

Algorithm 2.2 – The cuckoo hashing insertion algorithm.

It is possible that two or more rehashes take place consecutively since, after a rehash, the new hash functions might fail to accommodate all keys.

2.3.2 Load Thresholds

In order for the insertion algorithm to perform well, the load of the table must be sufficiently below a threshold. As the number of keys approaches em for some constant e , where m is the size of the cuckoo hashing scheme (summing the sizes of all tables), the probability of failure (infinite loop) increases. When above em , the cuckoo hashing will fail with high probability (22).

Different number of tables $d \geq 2$ yields different load thresholds. When $d = 2$, at most half of the cuckoo hashing scheme can be filled in order to successfully accommodate the keys. That is, the load threshold is $e = 1/2$ (23).

As first studied in (24), the memory usage highly improves as we increase d . The load threshold for $d = 3$ is approximately 0.917; for $d = 4$, it is approximately 0.976 and for $d = 5$, it is approximately 0.992 (25). Therefore, we can attain high loads and minimize the waste of space of our cuckoo hashing scheme by choosing a suitable value for d .

2.3.3 Time Complexity

The hash functions are assumed to be independent and uniformly random, and the load of the scheme is assumed to be below the threshold for d .

The worst-case lookup time is trivially $O(1)$. The probability a insertion might fail and a rehash is required is known to be $O(1/n^2)$, so the expected amortized cost of a rehash is $O(1/n)$ (8, 24). The expected time of the random walk insertion algorithm is $O(1)$, as shown in (24, 26).

In certain scenarios, such as in the context of router hardware, where hashing is used for a variety of operations, the amortized performance is not good enough. An alternative is the use of *de-amortizing* techniques, as seen in (27, 28). The idea is to reduce the probability of inserting in $\Omega(\log n)$ time.

2.3.3.1 Worst-Case Insertion Time

In order to study the worst case of the insertion, we focus on the rehash step, after the insertion fails.

When the first rehash occurs, there is a strictly positive probability that the new functions will not accommodate all keys. Then, another rehash is required during the first rehash. This second rehash may also yield failure and another rehash is required, and so on. There is a possibility an endless sequence of rehashes takes place during one insertion; thus, the worst-case time of the insertion algorithm is *infinite*.

The cuckoo hashing scheme that we propose in this text has finite worst-case insertion time — that is, $O(n^3/(\log \log n)^3)$.

3 CUCKOO HASHING WITH PERFECT REHASH

The cuckoo hashing variant proposed in this text seeks to avoid an arbitrary large insertion time, therefore drastically reducing the worst-case insertion time at the cost of a more expensive lookup, which is still very close to constant time in practice. The first version of the cuckoo hashing with perfect rehash was proposed in (10) (see the differences in Section 3.3).

The cuckoo hashing with perfect rehash runs an insertion in $O(n^3/(\log \log n)^3)$ time in the worst case, which means the insertion becomes *controlled*, without the risk of entering in an infinite loop. However, the lookup time is now $O(\log \log n)$ in the worst case.

Our cuckoo hashing variant employs $\delta(n) = \lfloor \log_2 \log_2 n \rfloor + 3$ tables, each of them associated with 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, as discussed in Section 2.3.2.

To look up a key x , we still check all its candidate locations (one per table). The lookup algorithm is found in Figure 3.1.

```

CHPRLookup ( $x$ )
1. for  $i = 1, 2, \dots, \delta(n)$ 
2.   if  $T_i[h_i(x)] = x$ 
3.     return true
4. return false

```

Algorithm 3.1 – The cuckoo hashing with perfect rehash lookup algorithm.

To insert a key x , we first look it up to avoid duplicates, and then we use the 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 x at position $h_j(x)$ in T_j . If another key is found in $h_j(x)$, 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 (11), so the rehash itself is always successful, with no two keys disputing the same spot. We call this operation a *perfect rehash*.

The CHPR insertion algorithm is summarized in Figure 3.2, where $\text{random}(\delta(n))$ returns a value in $\{1, \dots, \delta(n)\}$ and \perp denotes an empty space.

```

CHPRInsertion ( $x$ )
1.  if CHPRLookup( $x$ )
2.    return
3.  repeat  $c \log n$  times
4.     $j \leftarrow \text{random}(\delta(n))$ 
5.     $\text{temp} \leftarrow T_j[h_j(x)]$ 
6.     $T_j[h_j(x)] \leftarrow x$ 
7.     $x \leftarrow \text{temp}$ 
8.    if  $x = \perp$ 
9.      return
10. PerfectRehash( $x, j$ )

```

Algorithm 3.2 – The cuckoo hashing with perfect rehash insertion algorithm.

3.1 Perfect Rehash

Whenever the keys to be inserted in a hash table are known beforehand, a perfect hashing method can be applied to avoid collisions (11). 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 (19), seen in Section 2.2.1, with linear expected time. It so happens that the mapping step is a *random search* algorithm: an auxiliary graph 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 the deterministic method from Section 2.2.2. 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 $O(r^3) = O(n^3/(\log \log n)^3)$ 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 $\Theta(n/\log \log n)$ to $O(n^2/(\log \log n)^2)$, as possibly required by the deterministic perfect hashing method (20) depending on the value of q . As discussed in Section 2.2.2, one may use the prime number method to build the FKS scheme with $O(r)$ space.

3.1.1 Modified Lookup

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 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).

The pseudo-code in Figure 3.3 implements the modified lookup. The perfect hash functions are denoted as h'_i for $i = 1, \dots, \delta(n)$.

```

CHPRModifiedLookup ( $x$ )
1. for  $i = 1, 2, \dots, \delta(n)$ 
2.   if  $T_i[h_i(x)] = x$  or  $T_i[h'_i(x)] = x$ 
3.     return true
4. return false

```

Algorithm 3.3 – The cuckoo hashing with perfect rehash modified lookup algorithm. The underlying perfect hash function of the i -th table is denoted as h'_i .

3.1.2 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 cuckoo hashing (8, 24, 21, 26).

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 $O(r)$ even if we were not to interrupt it after cr^2 iterations (as explained in Section 3.1), 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 (19)), 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 (8, 24)), its amortized time becomes $O(n/\log \log n) \cdot O(1/n^2) = O(1/(n \log \log n))$, which is dominated by the average $O(\log \log n)$ complexity of the (necessary) lookup that precedes each insertion—hence negligible.

3.2 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 (8, 24), 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 6.1. 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 (26). Because those schemes execute a rehash in expected $\Omega(n)$ time with a $O(1/n^2)$ probability (8), 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 $O(n^3/(\log \log n)^3)$, 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 3.1.

Table 3 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 (15).

	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 3 – 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.

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.

3.3 Preliminary Work: The First Version

In my master’s thesis (10) we find preliminary work on our variant. In it, the CHPR employs \sqrt{n} tables and improves the classic cuckoo hashing worst-case insertion time and the chained hashing worst-case lookup time. Table 4 is adapted from (10) to include balanced search trees, which outperforms the CHPR in every aspect. Moreover, a deterministic perfect rehash was performed using a method that required every pair of keys to be relatively prime, thus restricting the keys used in the worst-case insertion.

	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	$\Theta(\sqrt{n})$	$\Theta(n \log n)$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$

Table 4 – The time complexity of basic operations for Balanced Search Trees (BST), the traditional Chained Hashing scheme (CH), the Classic Cuckoo Hashing (CCH), and the preliminary CHPR.

3.4 The Cuckoo Tree: An Open Problem

To reduce the lookup time from $O(\delta(n))$ to $O(\log \delta(n))$, we introduce an improvement we call the *cuckoo tree*, a binary tree where each node has a *split key*, a table and a hash function. The split key comes from the *generalized binary split trees* (29), and is used

to determine the keys that will compose the left and right sub-trees of the current node. If we are interested in a worst-case lookup time of $O(\log \log n)$, then $\delta(n) = \log n$ suffices.

The tables may be allocated as a tree from the start with pre-defined split keys. Another option is to allocate tables on demand, as keys are inserted and evicted. The current evicted key can be used as the split key of the new table, and the tree may be built as a balanced search tree. Operations like rotations, however, will force some keys to be reallocated among tables to preserve the properties of the tree, which affects the performance of the insertion.

The cuckoo tree is illustrated in Figure 6. The split keys are represented at the first position of each table.

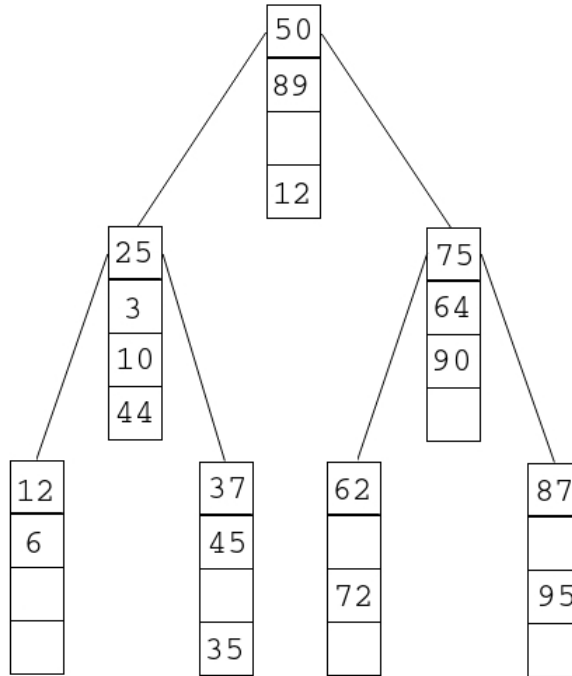


Figure 6 – A cuckoo tree where the first key of each table is a split key.

To lookup a key x , we start by looking it up in the root node with its underlying hash function. If x is not in the computed index, we compare it with the split key: if lower, we proceed to the left sub-tree; else, we proceed to the right sub-tree. This goes on until we find the key or reach a leaf node, in which case the lookup fails. As an example, when looking up key 45 in the cuckoo tree of Figure 6, we first look it up in the root node with its underlying hash function. Then, we compare it with 50 and proceed to the left sub-tree, where it is not found. Next, we compare 45 with 25 and proceed to the right sub-tree, where we find the key.

The insertion starts in the root node. If key y is evicted, we compare it with the split key to know if we proceed to the left sub-tree or to the right sub-tree, and then we continue this insert-and-evict procedure in the current node. When a key from a leaf node

is evicted, the next iteration is in the root node. Figure 7 illustrates the insertion of a key.

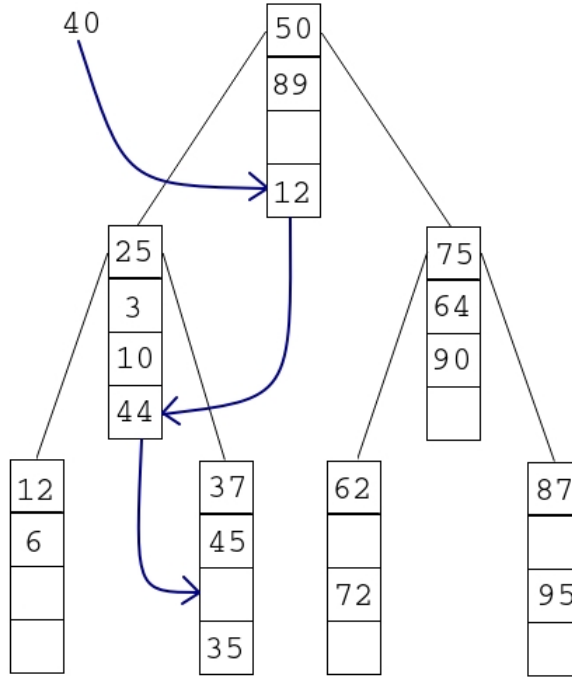


Figure 7 – Inserting key 40 in the cuckoo tree.

In order to preserve the properties of the tree, we can no longer choose an arbitrary table to perform a perfect rehash on: a key in a right (left) sub-tree must be higher (lower) than the parent split key. A simple solution is to pick a table in a walk through the tree, with the evicted key, to perform a perfect rehash on, and resize it if necessary (in case all available tables are full). However, in the worst case, all keys are placed in one path from the root node to some leaf — and hence the cuckoo tree degenerates to our regular CHPR approach without a random-walk insertion algorithm. The worst case scenario of the cuckoo tree is illustrated in Figure 8. Thus, insertions would take $\Theta(n^3/(\log \log n)^3)$ time in the worst case, because the $\log \log n$ tables in the aforementioned path would resize from $\Theta(n/\log n)$ to $\Theta(n/\log n)$ to accommodate all our keys.

It remains an open problem to perform a perfect rehash that does not increase the size of tables of the cuckoo tree, if it is actually possible.

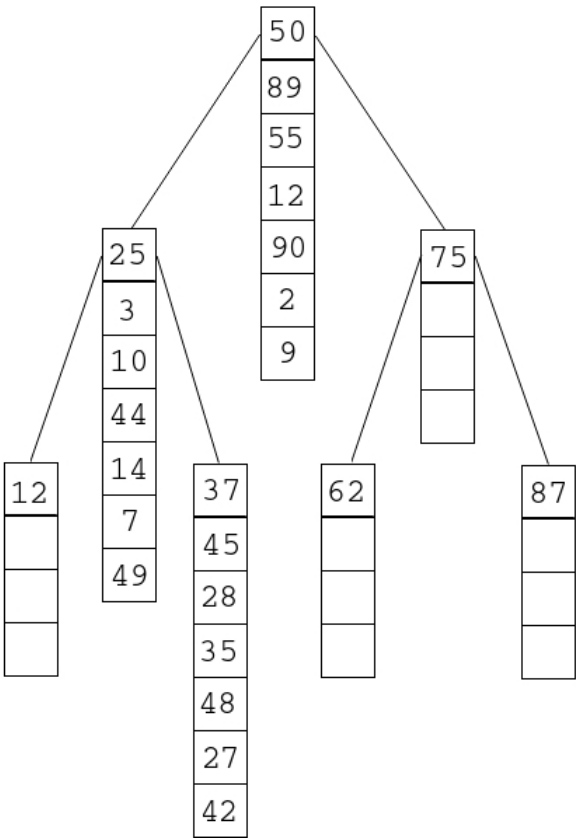


Figure 8 – An illustration of the worst case scenario of a cuckoo tree.

4 CONCURRENT HASHING

The advent of multicore machines brought the design of software where multiple concurrent *threads* manipulate shared objects, variables and data structures. It is fairly common, in parallel computing, that distinct threads share access to some hash table. The traditional way to provide an atomic behavior in software, so that only one thread at a time accesses the shared structure, is by using *mutual exclusion locks*. The mutual exclusion problem is one of the classic coordination problems in multiprocessor programming (9, 30, 31). It was first described and solved by Dijkstra in (32). Of course that does not come for free, as those traditional synchronization methods take a toll on performance.

4.1 Shared Data and Synchronization

To illustrate the problems we encounter in coordinating multiple threads in accessing shared data, we find the following example in (9).

Let us solve the problem of finding all primes in an integer range using multiple threads. Our approach is to assign each thread one integer at a time. When a thread is done testing an integer, it asks for another integer via a function that increments a global shared counter variable and returns its prior value to the caller. We call this variable *counter*, which is shared by all threads. Figure 4.1 shows an implementation of function *ReturnAndIncrement()* in pseudo-code.

```
ReturnAndIncrement ()
1.  temp  $\leftarrow$  counter
2.  counter  $\leftarrow$  temp + 1
3.  return temp
```

Algorithm 4.1 – An implementation of a function that increments a shared counter variable and returns its prior value.

Using the counter variable like this works well for a single thread, but it fails with multiple threads. The variable *temp* is a local variable to each thread, while the counter variable is shared among all threads. If both threads *A* and *B* call the *ReturnAndIncrement()* function at about the same time, they might simultaneously read 1 from the counter, set their local *temp* variables to 1, *counter* to 2, and both return 1. So, concurrent calls to *ReturnAndIncrement()* return the same value, but we expect them to return distinct values.

The problem is that incrementing the counter variable requires two distinct operations on it: reading its value into a temporary variable and writing its new value back to it. It is not an *atomic* (indivisible) hardware step.

4.1.1 Shared Hash Table Issues

In this Section, we examine some problems with non-atomic operations for traditional chained hashing and cuckoo hashing in a scenario with a single writer thread and multiple reader threads.

In traditional chained hashing, a concurrency issue arises when the writer is removing a key. When the reader is iterating through the list of synonyms of the key it is looking up, and computes the address of the next key x , the writer might be removing it from the list. Then, when reading the key from that memory cell, that memory space might be on use by another process; or, while comparing x with the searched key, the pointer to the next item might be nullified so that the operating system can reclaim those memory positions (via the garbage collector of the programming language, if present, or upon an explicit deallocation instruction in the program). Consequently the reader might fail the lookup of a key that is present.

An insertion is consistent in the case of a single writer. Nevertheless, in the optimized version of chained hashing, such as Java's `HashMap`¹, after the number of items in a bucket is above a pre-set threshold, the bucket is reorganized as a balanced search tree (a red-black tree in the case of `HashMap`). Thus, during an insertion in a bucket that is a tree, a lookup of a key from a reader might fail during a rotation (or during a node splitting, in the case such a tree is a B-tree), even if the key is present in the bucket.

In cuckoo hashing, after a key is evicted, it becomes temporary unavailable while it has not been reinserted in a different table. A reader thread may fail a lookup while the key is nowhere to be found. Also, during a lookup, the writer thread might place the current evicted key in a position that was already verified, and then the lookup fails.

4.2 Mutual Exclusion

In order to avoid problems with a non-atomic sequence of operations on shared memory, we transform the sequence into a *critical section*: a block of code that can be executed by only one thread at a time. This property is called *mutual exclusion*, and the standard way to approach it is through the use of *locks* (9).

We say a thread acquires the lock when it executes a `lock()` function call, and releases the lock when it executes an `unlock()` function call. Figure 4.2 shows a thread-safe

¹ The source code is available in <http://hg.openjdk.java.net/jdk8/jdk8/jdk>.

implementation of the *ReturnAndIncrement()* function, now with mutual exclusion. The operations in lines 2 and 3 are in the critical section.

```
ReturnAndIncrementMutualExclusion ()
1. lock()
2. temp  $\leftarrow$  counter
3. counter  $\leftarrow$  temp + 1
4. unlock()
5. return temp
```

Algorithm 4.2 – A thread-safe implementation of a function that increments a shared counter variable and returns its prior value using mutual exclusion.

Figure 9 illustrates the use of critical sections. When thread *B* executes the *lock()* function, it is trying to enter the critical section, and must wait for thread *A* to release the lock. Thread *C* executes the *lock()* function and also waits. After thread *A* releases the lock, thread *B* enters the critical section and thread *C* waits until *B* releases the lock.

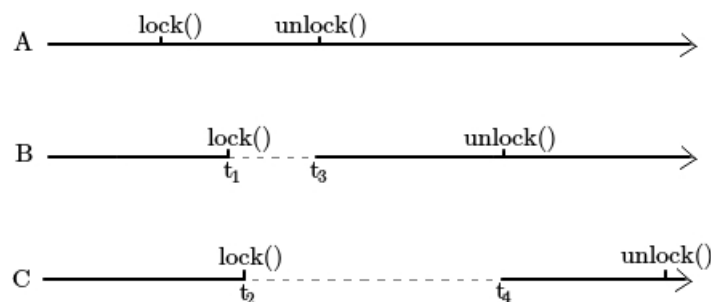


Figure 9 – Illustration of mutual exclusion, where threads *A*, *B* and *C* run concurrently. The straight horizontal arrows represent time. Thread *B* waits between times t_1 and t_3 , and thread *C* waits between times t_2 and t_4 .

We may want to associate different locks with different critical sections. In programming languages, a lock is normally encapsulated as a structure variable or as an object, and the functions or methods are called for individual locks.

The Java `ConcurrentHashMap` (33) is a lock-based hash map. It splits the hash table into segments and uses one lock per segment. Using one lock for the entire structure is an inefficient solution, because only one thread at a time is allowed to access the map. We want to maximize the code that can be executed concurrently.

4.2.1 Mutual Exclusion Overhead

Mutual exclusion synchronization takes time to be executed. For example, the operations of locking and unlocking, when executed many times, produces a relevant time overhead. To illustrate this, we designed a simple experiment in Java².

² The experiment is found in <https://github.com/judismar/sync-overhead>.

In our experiment, a variable was incremented in a loop that was repeated 1,000,000,000 times. We measured the total time it took to execute the loop, which was 65 milliseconds. We repeated this experiment by declaring the line of code that increments the variable as a Java *synchronized* block of code, and it took 27,000 milliseconds to execute the loop. We also used a lock variable to define the line of code that increments the variable as a critical section (locking before and unlocking after), and it took 22,000 milliseconds to execute the loop.

4.2.2 Deadlock and Starvation

When designing algorithms with mutual exclusion, we are mostly interested in progress guarantees. We may want our lock-based algorithms to be deadlock-free and starvation-free (9).

A deadlock occurs when all threads are waiting for locks to be released and neither of them is able to make progress, because the progress of one thread would require another thread to release a lock, while that thread (as well as all the others) is in the same situation.

A simple example is as follows. We have an object or data structure, and threads A and B are going to operate on it at about the same time. Thread A runs an operation that acquires lock 1, thread B runs another operation that acquires lock 2. While running the same operation, thread A eventually acquires lock 2, and thread B, before releasing lock 2, acquires lock 1. This yields a deadlock, because thread A is waiting for thread B to release lock 2 and thread B is waiting for thread A to release lock 1. Hence, both threads fail to progress.

Deadlock-freedom is a property that states that if one thread wants to enter the critical section, it eventually succeeds; if more than one threads want to enter the critical section, then eventually at least one of them succeeds.

Starvation occurs when a thread is unable to gain regular access to a critical section, and hence is unable to make progress. This happens when other threads block the shared data for long periods of time. As a simple example, thread A invokes an operation that locks a shared data for a very long time (say, for one billion instructions), and it invokes this operation very often. Thread B also wants synchronized access to the shared data, but the operation it runs locks the data for only 4 instructions. Hence, thread B will starve as it rarely gains access to the shared data.

The formal definition of starvation-freedom, however, requires that every thread that wants to enter a critical section will eventually succeed. In our example, thread B will eventually gain access to the shared data. This formal definition is important in the design of mutual exclusion solutions. For example, Dijkstra's first solution to mutual exclusion (32) is deadlock-free, but not starvation-free. When multiple threads want to

enter the critical section, one will eventually succeed. However, the thread that succeeds is arbitrary, meaning that some thread could wait forever and never gain access to the critical section. Modern solutions to the mutual exclusion problem are deadlock-free and starvation-free, as seen in (9).

The formal definition is also important in the case where a thread acquires a lock but never releases it, mostly due to a bug in the code. All other threads acquiring the same lock will starve.

4.3 Non-blocking Algorithms

The lock-based implementation in Figure 4.2 is called *blocking*: an unexpected delay by one thread can prevent others from making progress. Blocking implementations are poorly suited for certain systems (13). *Non-blocking* algorithms are also of interest, where threads synchronize their operations without the use of mutual exclusion.

A simple example is as follows. Suppose we are in a scenario with a single writer thread and multiple reader threads, and the writer thread keeps updating the value of two shared variables that are read by the reader threads. One variable could be a student name, where the other variable is the student's grade. When the writer thread is updating the variables, it writes a student name to the first variable and their corresponding grade to the second one. Because we are not using mutual exclusion to update both variables atomically, the reader thread might read a student name and a grade that corresponds to another student — an inconsistency we want to avoid. A simple solution, based on a solution implemented by the *optimistic cuckoo hashing* (34) (seen in Section 4.5.4), is to add a shared variable that works as version counter.

The version counter variable starts with 0. Before the writer thread begins to update the shared variables' values, it increments the shared version counter. When it finishes updating the variables, it increments the shared counter again. Figure 4.3 shows an algorithm that implements this idea.

UpdateValues (a, b)

1. version-counter \leftarrow version-counter + 1
2. $x \leftarrow a$
3. $y \leftarrow b$
4. version-counter \leftarrow version-counter + 1

Algorithm 4.3 – A thread-safe implementation of a function that updates the values of the shared global variables x and y , which respectively stores the student's name and their grade, using a shared global version counter variable.

Before reading the two variables, the reader thread snapshots the value stored in

the shared counter. If the value is odd, the writer thread is updating the variables' values and thus the reader thread may perform other tasks and return later to retry. Otherwise, it proceeds to reading the variables. After it is done, the reader thread snapshots the shared counter again. It then compares its new version with the old version: if they differ, the writer thread performed an update on the variables and the reader thread should retry its reading operation.

This simple synchronization solution guarantees that the reader threads will read a grade that corresponds to the student's name without the use of mutual exclusion.

4.3.1 Lock-Freedom and Wait-Freedom

When designing concurrent algorithms, we may be interested in some progress guarantees. Lock-freedom and wait-freedom are examples of non-blocking progress conditions (9).

A function or method is lock-free if it guarantees that infinitely often *some* call finishes its execution in a finite number of steps. A function or method is wait-free if it guarantees that *every* call finishes its execution in a finite number of steps. This means that an arbitrary delay by one thread does not necessarily prevents other threads from making progress.

An alternative definition of lock-freedom and wait-freedom is found in (35). An object or shared data structure is lock-free if it always guarantees that *some* thread completes an operation within a finite number of steps. An object or shared data structure is wait-free if it guarantees that *each* thread completes an operation within a finite number of steps.

Our solution to the students' grade problem is lock-free: the writer thread updates without any delay, at any time. However, it is not wait-free. A reader thread might attempt to read the shared variables and always read an odd version from the version counter, meaning that the writer thread is updating the variables. Thus, it might retry over and over again.

Figure 10, adapted from (36), compares all the progress conditions we have seen in this text.

4.4 Consistency

In sequential programming, we concern ourselves with the correctness property of *safety* (that is, some “bad thing” never happens during the execution of a program). The correctness of a multicore program is more complex: we are concerned about *liveness*

		non-blocking	blocking
every method makes progress		Wait-free	Starvation-free
maximal vs. minimal			
some method makes progress		Lock-free	Deadlock-free
			blocking vs. non-blocking

Figure 10 – A table that compares blocking and non-blocking progress conditions.

properties (such as lock-freedom and wait-freedom, seen in Section 4.3.1) and with safety (consistency) properties, which we study in this Section.

We study two safety conditions. The weaker condition of *quiescent consistency* and the stronger condition of *linearizability*.

4.4.1 Quiescent Consistency

The notion of quiescent consistency was introduced by (37) and (38).

A method or function call is the interval that starts with an invocation event and ends with a response event. A call is *pending* if the invocation event has occurred, but its response event has not.

An object or data structure is *quiescent* if it has no pending method or function calls. Two principles define the quiescent consistency (9): (1) method or function calls should appear to happen in a one-at-a-time, sequential order; and (2) method or function calls separated by a period of quiescence should appear to take effect in their real-time order.

In other words, quiescent consistency requires that the methods and functions that are called after a quiescent interval take effect after the ones that were called before this interval. When the calls overlap in an interval that is not quiescent, any order of their effects satisfies the quiescent consistency condition. Figure 11 helps illustrating quiescent consistency.

In Figure 11, the method or function call before the first rectangle takes effect before the ones after the first rectangle. The calls between the first and second rectangles may take effect in any order among themselves. The calls after the second rectangle take effect after the calls that are before the second rectangle. The two calls between the second

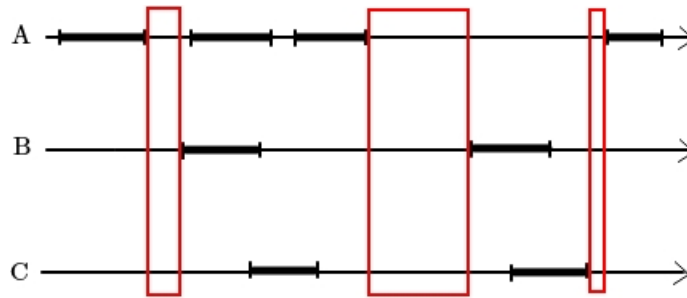


Figure 11 – Illustration of quiescent periods, where threads *A*, *B* and *C* run concurrently. The straight horizontal arrows represent time. The thick segments represent method or function calls, and the red rectangles delimit the quiescent periods.

and third rectangles may take effect in any order among themselves. Finally, the call after the third rectangle is the last one to take effect.

4.4.2 Linearizability

The notion of linearizability was introduced in (39).

Two principles define the property of linearizability (9): (1) method or function calls should appear to happen in a one-at-a-time, sequential order; and (2) each method or function call should appear to take effect instantaneously at some moment between its invocation and response.

The method or function call *appears* to take effect in a *linearization point*. The usual way to show that a concurrent object or data structure is linearizable is to identify, for each method or function, its linearization points. For lock-based implementations, each critical section can serve as a linearization point. For non-blocking implementations, the linearization point is typically a single step where the effect of the method or function call becomes visible to other method or function calls.

For a linearizable object, the method or function calls that overlap in time may take effect in any order. However, if a call ended with its response event, we are sure it already took effect. Figure 12 helps illustrating a linearizable object.

In Figure 12, assuming all calls come from a linearizable object: call 2 takes effect before call 1, which takes effect before call 3 (notice that there is no quiescent period between them). Then, call 4 takes effect, followed by call 5. Call 6 takes effect before call 7.

A formal, mathematical definition of linearizability is given in Appendix A.

4.4.3 Eventual Consistency

Eventual consistency guarantees that, if no additional updates are done to a given data item, all readers will eventually read the latest written data (40). Even though it

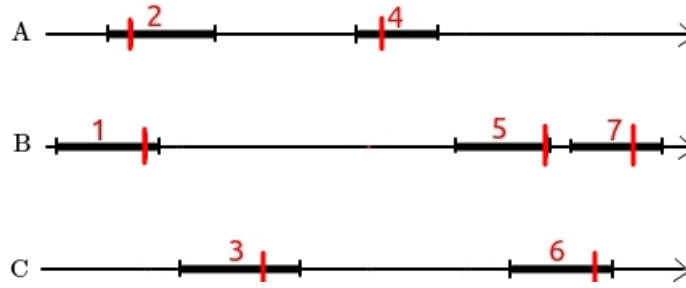


Figure 12 – Illustration of a concurrent execution of operations, where threads *A*, *B* and *C* run concurrently. The straight horizontal arrows represent time. The thick black segments represent method or function calls. The vertical red lines represent the linearization points of each call in this particular execution. The calls are numbered after their invocation time.

provides but a few guarantees, eventual consistency is inherent to many modern solutions, from distributed databases to event stores to the Internet Domain Name System (DNS).

The concept of eventual consistency was originated in the context of distributed computing (41). The theorem known as CAP theorem (42) states that it is impossible to achieve data consistency (linearizability) and system availability in the presence of tolerance to network partition. Thus, the designers of distributed systems sought weaker consistency models. Eventual consistency was a solid alternative to linearizability.

While eventually consistent systems do not guarantee safety, there is evidence that they work well in practice. There are quantitative metrics used to measure consistency, and studies have shown that eventual consistency is often strongly consistent, as seen in (40). Thus, sacrificing safety guarantees for the sake of high performance and simplicity of design is worth it in many cases.

4.5 Some Concurrent Hashing Schemes

4.5.1 Java ConcurrentHashMap

Java's `ConcurrentHashMap` (33) is a hashing scheme that uses lists to resolve collisions. Its first version used a small number of locks and was designed to optimize lookups. For simplicity, we first study an adapted version of the original `ConcurrentHashMap` found in (9). Then, we study the current version (since 2014) of Java's `ConcurrentHashMap` by explaining the most relevant changes and improvements.

4.5.1.1 The Adapted Version

Instead of using a single lock to synchronize the entire hash table, we split the table into independently synchronized pieces. The hash table is initialized with an array of L locks and a table with $N = L$ entries.

Although the array of locks and the hash table are initially of the same capacity, the table will grow when required, but the array of locks will not. When the table is resized, we double the capacity N . Thus, lock i covers table entry j , where $j \bmod L = i$. Figure 13, found in (9), illustrates this hashing scheme.

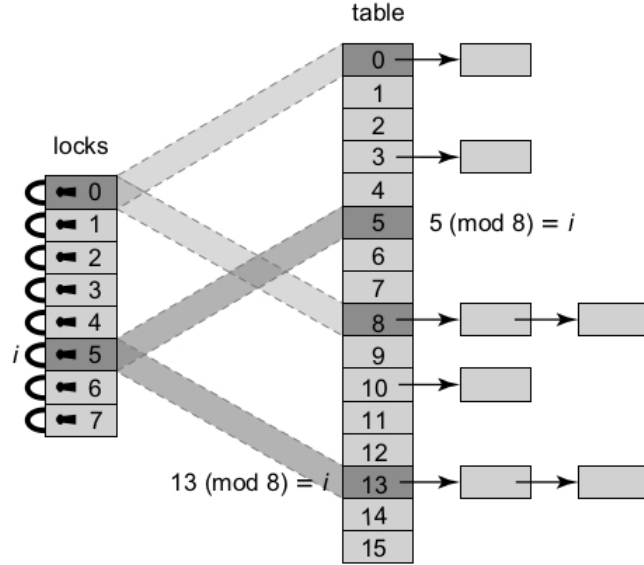


Figure 13 – An instance of the adapted version of Java’s ConcurrentHashMap with $N = 16$ and $L = 8$. Each lock covers $2^{N/L}$ entries.

Figure 4.4 shows the lookup algorithm. We first acquire the lock in the position $h(x) \bmod L$ of the array of locks. Then, we look up the key in the bucket of the position $h(x)$ of the table, and then release the lock.

```
CHMLookup ( $x$ )
1.  locks[ $h(x) \bmod L$ ].lock()
2.  result  $\leftarrow$  table[ $h(x)$ ].listLookup( $x$ )
3.  locks[ $h(x) \bmod L$ ].unlock()
4.  return result
```

Algorithm 4.4 – The lookup algorithm of the adapted version of Java’s ConcurrentHashMap.

The insertion algorithm is found in Figure 4.5. We first acquire the lock in the position $h(x) \bmod L$ of the array of locks. Then, we attempt to insert the key in the bucket of the position $h(x)$ of the table (line 2). The *listInsert* method call returns true if the key is successfully inserted, or false if it was already present. Before releasing the lock, we increment variable *size* (which starts with 0) if the key was successfully inserted (lines 3 and 4). Before returning the result of the insertion, we check if the average bucket size exceeds 4, in which case we double the capacity of the table (lines 6 and 7). Details on the resize procedure is found in (9).


```

CHMInsert ( $x$ )
1.  locks[ $h(x) \bmod L$ ].lock()
2.  result  $\leftarrow$  table[ $h(x)$ ].listInsert( $x$ )
3.  if result
4.    size  $\leftarrow$  size + 1
5.  locks[ $h(x) \bmod L$ ].unlock()
6.  if size/table.length > 4
7.    resize()
8.  return result

```

Algorithm 4.5 – The insertion algorithm of the adapted version of Java’s ConcurrentHashMap.

A *resize* operation halts all other concurrent operations. It acquires every lock in the array of locks in ascending order. A *resize()* call cannot deadlock with another *resize()* call because they both start without holding any locks, and acquire the locks in the same order. When a thread starts to resize the table, it stores the current table capacity. After it is done acquiring every lock, if it then discovers that another thread has updated the table capacity, then it releases the locks and gives up.

4.5.1.2 The Current Version

Java’s ConcurrentHashMap last update was introduced in 2014 and its code can be found in <http://hg.openjdk.java.net/jdk8/jdk8>.

The *initial capacity* and the *intended load factor* can be informed as arguments of the constructor. The table is thus pre-allocated to accommodate a given number of elements. For compatibility with previous versions, you can also inform the *concurrency level*, which is used, in the current version, as an additional hint for internal sizing. In the original version, the expected concurrency level was used to split the table into *segments*, where each segment was synchronized independently (like seen in the adapted version). For example, a concurrency level of 16 would yield 16 independently synchronized segments.

The most relevant change introduced in the current version is the absence of segments. Now, each bucket is synchronized independently using a Java *synchronized* block. This means that the “concurrency level” grows with the table size, and each entry of the hash table is synchronized independently.

The read operations are generally executed without blocking, so they can overlap with write operations.

The most common case of the insertion algorithm is when the bucket is empty. In this case, the first node is allocated without blocking. When the bucket is not empty, the insertion is performed in a Java *synchronized* block, using the first node of the linked list

as the monitor object. This means that concurrent insertions, updates and removals in the same bucket will have to wait.

With the exception of insertions that do not incur rehashes, all other write operations rely on blocking.

4.5.2 TBB concurrent_hash_map

Intel's TBB `concurrent_hash_map` is explained in (43). It is a popular concurrent hash map for C++.

The hash table consists of an array of buckets, where each bucket consists of a list of nodes and a read-write lock (which is explained in the following paragraphs) to control concurrent access by multiple threads. Elements are placed in the bucket located at the table position obtained from the computed hash value. Figure 14, found in (43), illustrates the hash map.

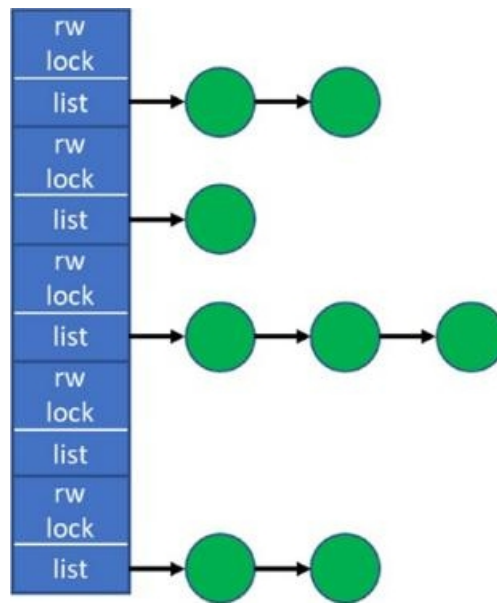


Figure 14 – The TBB `concurrent_hash_map`.

A lookup (the *find* operation) computes the hash value and acquires the read lock of the corresponding bucket. The read lock guarantees that there is no concurrent modifications to the bucket while it is being read by multiple reader threads. Inside the bucket, the lookup is performed by executing a linear search through the list of nodes.

The insert operation first allocates a new node. It then computes the hash value of the key to find the corresponding bucket, and acquires its write lock — which locks both readers and writers. The new node is then linked to the list of nodes and the write lock is released.

4.5.3 Hopscotch Hashing

The hopscotch hashing (44) was created for sequential and concurrent use, with $O(1)$ worst-case lookup time. Like cuckoo hashing, an insertion might fail and a full rehash is required.

We have a single hash function h and a single table. A key x will be found either in position $h(x)$, or in one of the next $H - 1$ positions, where H is a constant that the authors set to 32. Thus, a bucket has fixed size and overlaps with the next $H - 1$ buckets. Each entry includes a *hop-information* bitmap of H bits. It indicates which of the next $H - 1$ entries contains keys that hashed to the current entry's bucket. A lookup can thus be performed by looking at the bitmap to see which entries belong to the bucket, and then checking the constant number of entries.

When inserting a key, the H positions of the bucket might be occupied. The idea is to "move the empty slot towards the desired bucket". The insertion algorithm works as follows, where $h(x) = i$. Starting at i , look for the first empty entry of the table. If the empty entry's index j is within $H - 1$ of i , place x there and return. Otherwise, j is too far from i . To create an empty entry closer to i , find a key y whose hash value lies between i and j , but within $H - 1$ of j , and whose entry lies below j . Displacing y to j creates a new empty slot closer to i . Repeat. If no such key exists, or if the bucket i already contains H keys, resize and rehash the table.

Figures 15 and 16, found in (44), show an example execution of the hopscotch insertion algorithm with $H = 4$. The blank entries are empty, and the others are occupied. In Figure 15, we are inserting key v with hash value 6. We look for the first empty entry, which is entry 13. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is $H - 1 = 3$ entries before, at entry 10. That entry's hop-information bitmap indicates that key w at entry 11 can be displaced to 13, which we do. Entry 11 is still too far from entry 6, so we examine entry 8. The hop-information bitmap indicates that key z at entry 9 can be moved to entry 11. Finally, key x at entry 6 is moved to entry 9. Then, key v can be inserted in entry 6, as shown in Figure 16.

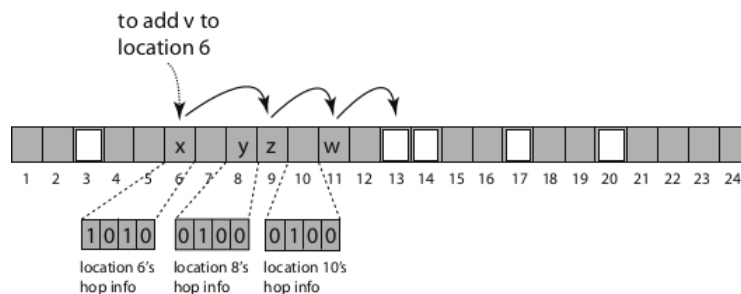


Figure 15 – The hopscotch insertion algorithm looks for a sequence of displacements of keys.

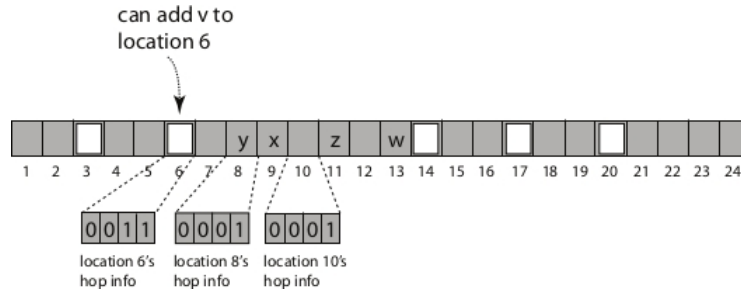


Figure 16 – The hopscotch insertion algorithm after the keys were displaced to create an empty entry for key v .

The concurrent version of the hopscotch algorithm splits the table into segments, where each segment is protected by a lock, in a manner similar to (33). The lookup algorithm is wait-free, it ignores locks, while the insert and remove algorithms acquire locks before modifying the data. The pseudo-code that implements the lookup, insert and remove algorithms is found in (44).

4.5.4 Optimistic Cuckoo Hashing

The optimistic cuckoo hashing (34) maintains high memory efficiency. It is designed for scenarios where lookups dominate, where only one writer is allowed at a time. In other words, it is tailor-built for the very scenario with a single writer and multiple readers that our proposed monkey hashing scheme is especially well-suited for.

This scheme uses two hash functions. The hash table consists of an array of buckets, each having 4 slots. Each slot contains a *tag* and a *pointer* to the key-value object. A null pointer indicates that this slot is not used. The use of a pointer to store the key-value pair outside of the table is to support keys of variable length. Figure 17, adapted from (34), illustrates the hash table.

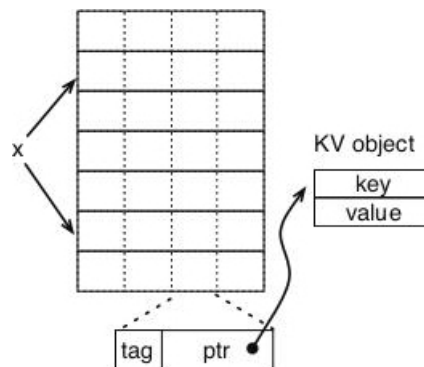


Figure 17 – The optimistic cuckoo hash table.

Each key is mapped to two random buckets, so the lookup algorithm checks all 8 slots. The maximum number of iterations of the insertion algorithm is set to 500. When inserting key x into the table, if either of the two buckets has an empty slot, it is then

inserted in that bucket. If neither bucket has an empty slot, the algorithm selects a random key y from one candidate bucket and displaces it to its alternate location. Displacing key y might also require dislodging another existing key z , so this insert-and-evict procedure might repeat until an empty slot is found, or until the maximum number of iterations is reached — in which case a rehash and a resize are performed.

4.5.4.1 Tags

The tag is a short hash of the keys and is used to implement the lookup and insertion operations. The lookup algorithm first compares the tag, then retrieves the full key only if the tag matches — thus avoiding unnecessary pointer dereferences.

It is possible to have false positives when looking up a key, due to two different keys having the same tag, so the retrieved full key is further verified to ensure it was indeed the correct one. With a 1-byte tag, the probability of tag collision is $1/2^8 = 0.39\%$.

The insertion can operate using tags and never have to retrieve keys. This hashing scheme computes the two candidate buckets for key x by $b_1 = h(x)$ (based on the entire key) and $b_2 = b_1 \oplus h(tag)$ (based on b_1 and tag of x).

Using the same formula, b_1 can be computed from b_2 and tag. This property ensures that to displace a key originally in bucket b (whether it is b_1 or b_2) it is possible to calculate its alternate bucket b' from bucket index b and the tag stored in bucket b by $b' = b \oplus h(tag)$.

4.5.4.2 Insertion

As discussed in Section 4.1.1, when the writer thread is performing an insertion, a reader thread may fail to lookup a key that is present. To eliminate this concurrency issue, the optimistic cuckoo hashing changes the order of the cuckoo hashing insertion. First, it discovers the cuckoo path to an empty slot. Then, it moves the keys backwards along the cuckoo path. Thus, we ensure that there is never a key that is temporary unavailable, a key that would otherwise be evicted out of its slot.

The cuckoo hashing traditional insertion algorithm dislodges keys along the cuckoo path until an empty slot is found. Figure 18, adapted from (34), illustrates a cuckoo path. The optimistic cuckoo hashing first discovers the path. Then, key c is displaced to the empty slot in bucket 3, followed by relocating key b to the original slot of c in bucket 1 and so on, until key x can be inserted in the newly created empty slot in bucket 6.

4.5.4.3 Lookup

To synchronize the insertion and the lookups, we assign a version counter for each key. We update its version when displacing the key during an insertion, and we look for a

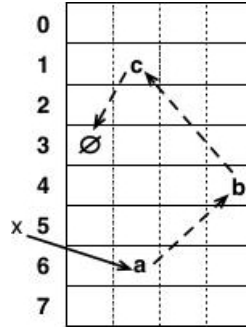


Figure 18 – A cuckoo path, where \emptyset represents an empty slot.

version change during a lookup to detect any concurrent displacement.

We create an array of 8192 counters. Each counter is shared among multiple keys by hashing (e.g. the i -th counter is shared by all keys whose hash value is i). All counters are initialized to 0.

Before displacing a key, an insertion first increments the counter, indicating to the reader threads an on-going update for this key. After the key is moved to its new location, the counter is again incremented to indicate the completion. As a result, the key version is increased by 2 after each displacement.

Before reading the two buckets for a given key, the lookup algorithm first snapshots the version stored in the key's counter: if this version is odd, the writer thread must be working on the same key (or another key sharing the same counter), and thus the reader thread should wait and retry; otherwise it proceeds to the two buckets. After it finishes reading both buckets, it snapshots the counter again and compares its new version with the old version. If they differ, the writer thread must have modified this key, and the reader thread should retry.

More details on the optimistic cuckoo hashing are found in (34).

4.5.5 A Wait-Free Hashing Scheme

The first wait-free hash map was proposed in (45). It is a high-performance, linearizable multi-level array that avoids global resize by allocating new arrays when necessary.

Figure 19, found in (45), shows the structure of the scheme. It is similar to a tree, and each position can hold an array of nodes or a single node.

A position that holds a single node is called a *data node*. It stores the hashed key and the value associated with that key. An *array node* is a position that holds an array of nodes.

Similarly to a tree, this construct keeps a pointer to the root, which is an array

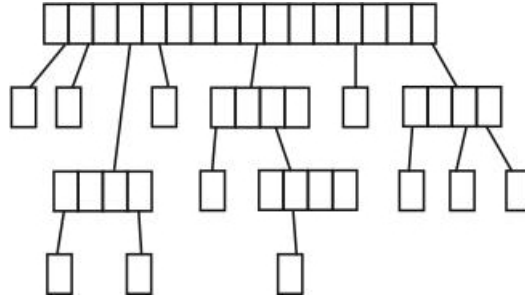


Figure 19 – An illustration of the wait-free hashing scheme.

node of unique length, whereas every other array node has uniform length. The maximum depth of the tree is the maximum number of array nodes that must be followed to reach any node.

The hashed key is expressed as a list of indices, where each index has the same size of the array length (e.g. $A - B - C - D$, where A is the first index, B is the next index and so on; these represent positions on different array nodes).

This hashing scheme requires all lists of indices to be distinct, that is, it lacks the freedom to choose a hash function — like in a direct addressing solution. An example of implementation (found in the authors' code) is the use of strings as keys: a hashed key is a list of integer values, where each value corresponds to a symbol of the string. Thus, because the strings are distinct, the hash values are distinct.

The maximum depth is calculated as the ceiling of the division of the number of bits in the key with the number of bits needed to represent the length of each array. For example, with keys of 32 bits and an array length of 64, we have a maximum depth of 6, because $\lceil 32 / \log_2 64 \rceil = 6$. The time complexity of every operation is bounded by the maximum depth, and are therefore constant. This is only possible because the hash function is, in practice, the key itself.

4.5.5.1 Traversing the Multi-Level Array

When traversing the multi-level array, we start at the right-most index of the list of indices of the hashed key. We then examine that position on the current array node (the root node). If the pointer stores the address of an array node, we continue the traversal procedure in this next node with the next index of the list. This goes on until a data node is reached.

Figure 20, found in (45), illustrates this process. In this example, the array nodes have a length of four, except for the root node which has a larger size than every other array node. The hashed key is expressed as a list of indices, e.g. $A - B - C$, where C is the first index, B is the second index and so on. These indices are positions of array nodes at various depths. Empty data nodes are represented as *null* values.

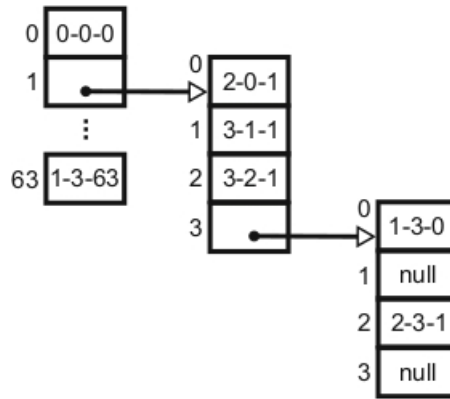


Figure 20 – An example of hashed keys stored in the wait-free hash map.

As an example of lookup, if we need to find key 42 in the hash map shown in Figure 20, we first hash the key. We assume that this operation yields 2-3-1. To find 2-3-1 we first take the right-most index, 1, and go to that position on the root node. It is an array node, so we take the next index, 3, and examine position 3 on this node. This position is also an array node, so we take the next index, 2, and examine that position on this node. That position is a data node, so we compare its hashed key to the hashed key that we are searching for, and thus we successfully finish the lookup.

4.5.5.2 Insertion

The insertion algorithm traverses the hash map until it finds a position that is empty to insert the new element. When it reaches a data node that is not empty, the collision is resolved by allocating a new array node with this data node in it, and the algorithm continues its traversal from the new array node.

The implementing details of every operation are found in (45).

4.5.5.3 Disadvantages

This hashing scheme has some disadvantages that are worth discussing.

Because the hash function cannot be chosen arbitrarily and its behavior is known by the programmer, the malicious adversary may choose keys that will lead all operations to their worst case (bigger multiplicative factors, that is, but still within constant time), where the tree reaches its maximum depth. The number of collisions may also happen to be (intentionally) maximized so that $\Theta(n)$ arrays are allocated, yielding high memory overhead.

This scheme cannot be pre-allocated to accommodate a given number of keys — it would use as much memory as a direct-addressing solution.

The worst-case performance of all basic operations is affected when larger keys are used — the maximum depth of the tree increases with the size of the key (the worst-case

time is still constant, with bigger multiplicative factors).

The authors' code³ has a *segmentation fault* bug that had to be fixed in order to execute the experiments in Section 6.2.

4.5.6 Other Concurrent Hashing Schemes

The hash table in (46) is a high-performance, lock-free scheme that resolves collisions via linked lists. It can perform resizes when needed, to grow and shrink the table size. However, after a resize is performed, the average time complexity of the operations is no longer constant (47).

The work in (47) presents another high-performance, lock-free hash table that maintains the average constant time complexity of operations after a resize.

A lock-based concurrent cuckoo hashing is found in (9). It improves concurrency by trading space; thus, its space overhead is higher than the basic cuckoo hashing (34).

The optimistic cuckoo hashing was later improved by (48) to allow multiple writers, using the concurrent writer approach from the concurrent cuckoo hashing in (9).

The work in (49) presents a hashing scheme that is wait-free when the hash table is not resized. The wait-freedom is violated only when a resize is performed.

³ <https://github.com/AgamAgarwal/wait-free-extensible-hash-map>

5 MONKEY HASHING

Monkey hashing (MH) is a thread-safe hashing scheme that does not use locks or any other kind of synchronization, requiring only that single-word read and write operations are atomic. It consists of a single hash table and $k \geq 1$ hash functions, meaning multiple alternative locations for each key in the same table. Unlike cuckoo hashing, elements will never be evicted from where they first landed, so new keys being inserted must always find an unoccupied spot to call their own.¹

Like the concurrent cuckoo hashing in (34), our hashing scheme is specially tailored for scenarios with a single writer and multiple reader threads, though it certainly works just fine even in single-thread applications.

The monkey hashing can be implemented as a *hash map*, that is, it can store key-value pairs, even though a pair consists of two memory words and cannot be read atomically. Of course it can also be implemented as a *hash set*, where only keys are stored.

It is required that one knows beforehand the maximum number of entries that may ever coexist in the map (i.e., its maximum capacity), since memory is pre-allocated to enforce the intended load factor without the need of rehashes.²

The k hash functions may be obtained from a universal class (17). Such functions work as well as idealized uniform hash functions do on average (16).

To look up some key x , we follow the sequence of positions $h_1(x), h_2(x), \dots, h_{k'}(x)$ in the underlying array T (i.e., the table) until we have either found x or reached the maximum number of hash functions that were actually called for by any insert operation, denoted by k' . Such a *maximum in use* value k' is dynamically kept track of. The pseudo-code of the lookup algorithm is shown in Figure 5.1.

```

Lookup( $x$ )
1. for  $i = 1, 2, \dots, k'$ 
2.   if  $T[h_i(x)] = x$ 
3.     return true
4. return false

```

Algorithm 5.1 – The monkey hashing lookup algorithm.

¹ The name stems from the analogy to a monkey that jumps along branches of a tree until it finds a vacant one to rest. Some cultures even have popular sayings in the likes of “every monkey to their own branch”, meaning “every jack to his trade”.

² In the experiments of Section 6.2, Java’s `ConcurrentHashMap` and Intel’s TBB `concurrent_hash_map` are also pre-allocated to avoid resizes (something they do support), thus ensuring a fair comparison.

If we intend on prioritizing the insertions, in a scenario with fewer reads, we may give up of the k' variable and of all of its control mechanisms. It would not change the performance of the operations that iterate through all entries, and every read and write operations would still perform in constant time.

When a key x is inserted, we insert x in the first position in the hash table that is not occupied along that same sequence $h_1(x), h_2(x), \dots$, up to $h_k(x)$, where k is the aforementioned number of available hash functions, i.e., the *maximum allowed* number of insertion attempts for a given key. The insertion fails if all k positions are occupied. The pseudo-code in Figure 5.2 shows the insertion algorithm, where an empty position is denoted by \perp . Note that we use a mapping A of all positive integers $i \in [1, k']$ onto the number of keys that required i hash functions during its insertion. Such a mapping will also be updated by the remove operation, and is used ultimately to keep track of $k' \leq k$.

```

Insert( $x$ )
1.  if Lookup( $x$ )
2.    return
3.  for  $i = 1, 2, \dots, k$ 
4.    if  $T[h_i(x)] = \perp$ 
5.       $A[i] \leftarrow A[i] + 1$  // one more key requiring  $i$  functions
6.       $k' \leftarrow \max\{k', i\}$  //  $k'$  may have changed
7.       $T[h_i(x)] \leftarrow x$ 
8.    return
9.  fail

```

Algorithm 5.2 – The monkey hashing insertion algorithm.

The insertion algorithm may also be implemented with unlimited k ; hence, the insertion time would become a random variable, as well as the time of every other basic operation (with $\omega(1)$ worst-case time). This would remove the chance of failure, while the operation that iterates through all entries would maintain its efficiency.

To remove a key, we first look it up in the MH structure. If it is found, then, along with removing it from the table, we also make sure to remove its contribution to the mapping A of hash function counts. Figure 5.3 gives the pseudo-code for the remove operation.

Since there are no keys ever being moved and no collision lists to maintain, the threads accessing a MH structure are free to execute all basic operations without any kind of synchronization. Hence, for a constant number k of hash functions, the operations are all wait-free, bounded by $O(k) = O(1)$ steps.

We can iterate through the entries in thread-safe fashion by traversing the underlying array while skipping empty positions. Insertions and deletions done while an iteration is

```

Remove( $x$ )
1.  for  $i = 1, 2, \dots, k'$ 
2.     $p \leftarrow h_i(x)$ 
3.    if  $T[p] = x$ 
4.       $T[p] \leftarrow \perp$ 
5.       $A[i] \leftarrow A[i] - 1$  // one less key requiring  $i$  functions
6.      if  $A[i] = 0$  and  $k' = i$  // must update  $k'$ 
7.        for  $j = i - 1, i - 2, \dots, 1$ 
8.          if  $A[j] > 0$ 
9.             $k' \leftarrow j$ 
10.     return
11.    $k' \leftarrow 0$  // empty table
12. fail // not found

```

Algorithm 5.3 – The monkey hashing removal algorithm.

taking place might not be immediately seen, as discussed in Section 5.1, something usually referred to as eventual consistency (seen in Section 4.4.3), which is inherent to—and all the most tolerable by—most multi-threaded applications.

5.1 Thread-Safety and Eventual Consistency

The writer thread and the reader threads execute their operations concurrently, as if they were isolated in a sequential scenario. This is possible because the monkey hashing does not require *any* kind of synchronization.

An entry with key x that has just been inserted may not be seen immediately thereafter by all reading threads. A reading thread might have started to look it up before it was inserted, and may have just checked position $h_j(x)$. Then, the writer thread inserts the entry in that same position and the reader thread resumes its operation from position $h_{j+1}(x)$ and fails to find the key. After it is first seen, though, never (again) should it fail to be successfully retrieved by each reading thread. Such a behavior—when no more updates are done, all readers will eventually read the latest written data—is commonly referred to as eventual consistency, which we studied in Section 4.4.3.

Deleting a key does not cause any trouble to the lookup operations in the reader threads: it takes a single atomic step to erase the key, be it of a primitive data type or an object reference (setting it to a *null* value). In the latter case, if the reference was already retrieved, and then removed shortly thereafter, that object will still be accessible to the reader, even if the writer replaces that reference with a null or some other reference.

When using MH to implement maps, the value, in each key-value pair, should be written first (and deleted last) to avoid the retrieval of a key without the intended

corresponding value. Moreover, when storing key-value pairs, a mapped value that is changed may not be immediately seen by all reading threads. Indeed, a reader thread might have just retrieved the reference of a value, when the writer thread replaces it with a reference to another value. Hence, the reader thread will actually have accessed the *previous* value. This is an expected, harmless race condition, and the outcome would have been the same had the update operation started after the completion of the read operation. Note, though, that reading partially updated entries must be avoided by enforcing that the data types of both key and value have atomic writes (which is the case, for example, in modern Java, for pointers and wrappers of primitive types).

Note that key-value pairs will probably be implemented as objects allocated in the random-access memory. Each position in the underlying array (i.e., the table) stores a reference to one of those objects (see Figures 21a and 21b). To improve performance, we may want to reuse the referenced object when removing a key-value pair. This avoids the overhead of dynamically freeing and (re-)allocating memory for new entries, as well as reducing the garbage collection burden, when applicable. We can achieve that by logically erasing both the key and the value of the existing entry without actually freeing the key-value object. However, after a reader thread retrieves an entry, the writer thread might overwrite the fields in that reused key-value object pair (after, say, a logical deletion followed by an insertion), and thus the reader thread might end up reading the value associated with the new key. When the key can be inferred from the value object (say, the key is some *user id* and the value is some *User* object, containing, among other data attributes, the user id itself), we can avoid this concurrency issue altogether by checking that the retrieved value does indeed correspond to the informed key. This can be done by providing MH with a function that, given a value object, returns its (unique) key. If the key obtained from the value does not correspond to the search key, then a deletion surely took place in the meantime and a null would be returned instead; otherwise return the retrieved, verified value object. In case such a verification function is not provided, the MH scheme will still work, albeit *not* reusing key-value objects, therefore actually freeing such objects during deletions, and reallocating a new key-value object from scratch at every insertion.

In case only keys are stored in the scheme (i.e., when implementing sets, not maps), keys (or, alternatively, references to key objects) may be written directly to the underlying array (see Figures 21c and 21d).

5.2 Insertion Failure Probability

The insertion method is strictly speaking a randomized algorithm with a probability of failure. We can make such a probability be so small that the MH would work just as

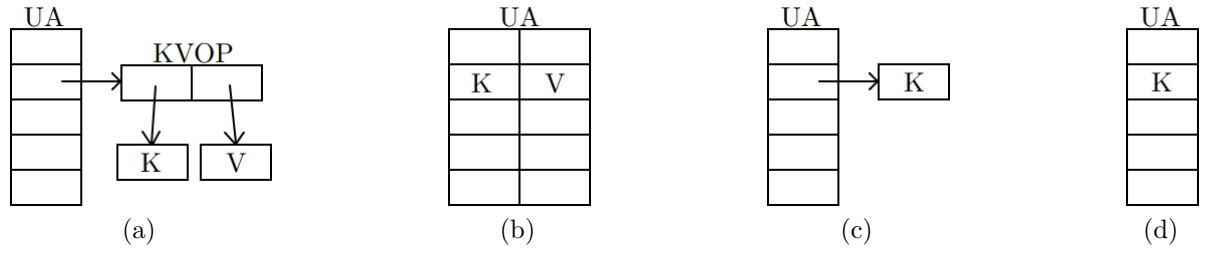


Figure 21 – Some possible setups using MHM: (a) hash map for objects; (b) hash map for primitive types; (c) hash set for objects; (d) hash set for primitive types. K, V, UA and KVOP indicate, respectively, the stored key, the mapped value, the underlying array (table), and the key-value object pair.

reliably as any deterministic algorithm would, in practice.

The upper bound on the probability that an insertion might fail decreases exponentially with the predefined maximum number of hash functions. Say we set 50 as the maximum number of hash functions, as we did in the tests depicted in Section 6.1, and $m = 2n$ as the size of the hash table, where n is the maximum number of entries stored at any given time. Because the load factor never exceeds $1/2$, when inserting key x the probability that position $h_i(x)$ is occupied is at most $1/2$ for all $i \in \{1, \dots, 50\}$. Thus, the probability that the insertion of x fails is at most 2^{-50} . The number of insertions until the first failure occurs is a geometric random variable with expectation at least 2^{50} , which is more than one quadrillion. If we perform one million insertions (and one million deletions, keeping the table at its maximum capacity) every second, non-stop, this means we would expect to see our first failure in 35.7 years.³ If we increase the number of functions very slightly, from 50 to just 52, the expected time before the first failure occurs increases to nearly one and a half century.

5.2.1 Working with Different Load Factors

It is not required that the load factor is set to $1/2$. The failure probability ρ of an insertion is at most α^k , where α is our load factor. For example, with $\alpha = 0.95$ and $k = 90$, we obtain $\rho \approx 0.0099$.

The user of our monkey hashing might want to set the failure probability they are willing to work with, and compute α and k from there.

5.3 Resizable Monkey Hashing

By default, hashing schemes in both sequential and concurrent scenarios are implemented in a way that allows the table to grow and shrink when necessary. If we aim

³ To keep things in perspective, Twitter worldwide usually registers an average of 5,700 new tweets per second, with a registered peak of 143,199 tweets per second in August, 2020 (50).

for a load factor of $1/2$, as soon as the number of items in the table is above $n = m/2$, the size of the table is usually doubled and it is rehashed, because the old hash function (or hash functions, depending on the scheme) would work for a table of size m , not $2m$. When the number of items is below $n = m/4$, we may want to avoid wasting space and hence halve the table size, rehashing all items. Both operations clearly take $O(n)$ time, because all present items will be reorganized.

The monkey hashing can be implemented as a resizable hash table without losing its thread-safety and eventual consistency. When the load factor is above the desired value, instead of resizing our array, we allocate a new one with the double of its original size. We then obtain k new hash functions and begin a process of copying all items from the old array to the new array. When this process is completed, we update the pointer or reference of the hash table to the newly allocated array. Even as items are inserted into and removed from the new hash table, concurrent reader threads that were reading the old array will continue their operation, without losing eventual consistency.

The average case of an insertion (and removal) in such hashing schemes can still be $O(1)$ (in an amortized setting, that is, considering a sequence of operations), as we shall show for our resizable monkey hashing. Our worst-case insertion was considered the one where it fails, but we shall shift the focus to a worst-case insertion time. Such time is clearly $O(n)$, because any resize takes $O(n)$ time. Without loss of generality, we focus on the case of the first resize. Before it takes place, $n = m/2$ insertions of $O(1)$ time are inevitably performed. Under reasonable assumptions, allocating an array of size $2m$ takes $O(m)$ time. Obtaining new hash functions clearly takes $O(1)$ time, and copying every key to the newly allocated array takes m operations of $O(1)$ time. Hence, the amortized cost of an insertion is

$$\frac{O(m) + O(m) + O(1) + O(m)}{m/2} = \frac{O(m)}{m} = O(1).$$

In a similar way, removing an element takes $O(1)$ expected time and $O(n)$ worst-case time in our resizable monkey hashing.

If memory is not a concern, we can always pre-allocate, as advocated previously in this text, and enforce a $O(1)$ worst-case insertion and removal time.

6 EXPERIMENTS

6.1 CHPR Experiments

All experiments in this Section were conducted on a Linux Mint 20.3 64-bit machine, with a multicore Intel(R) Core(TM) i3-2100 CPU with 4 cores of 3.10GHz, and with 3.7 GiB of RAM memory.

Both the Python code and the output data for the experiments in this Section are available in <https://github.com/judismar/chpr>.

We designed an 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 22). 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.

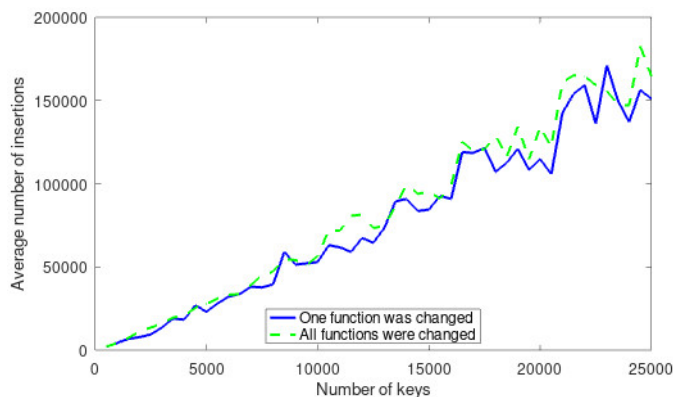


Figure 22 – The average number of insertions until the second rehash is called for, after the first rehash has been performed.

We also designed experiments to complement the theoretical results summarized in Table 3. In, Figure 23, the average time of a lookup and an insertion are in function of $x = \log_2 \log_2 n$, where n is the number of elements in the scheme.

To get the average time of a lookup, we begin by inserting an uniform random sample of size n from the universe of keys $[10^7]$. We then lookup 10,000 random keys, obtained independently and uniformly from the universe, for every value of x . As for the insertion amortized cost, we begin with empty tables. We sequentially insert n keys chosen

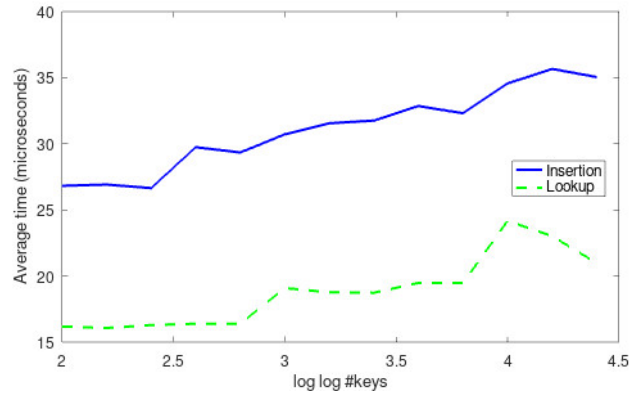


Figure 23 – The average time of an insertion and a lookup of the cuckoo hashing with perfect rehash in function of $\log \log n$, where n is the number of keys in the scheme.

independently and uniformly at random from the universe, and divide the total time by n . This is done independently 100 times to obtain the average of the amortized cost.

Figure 24 presents the results of the worst-case experiment.

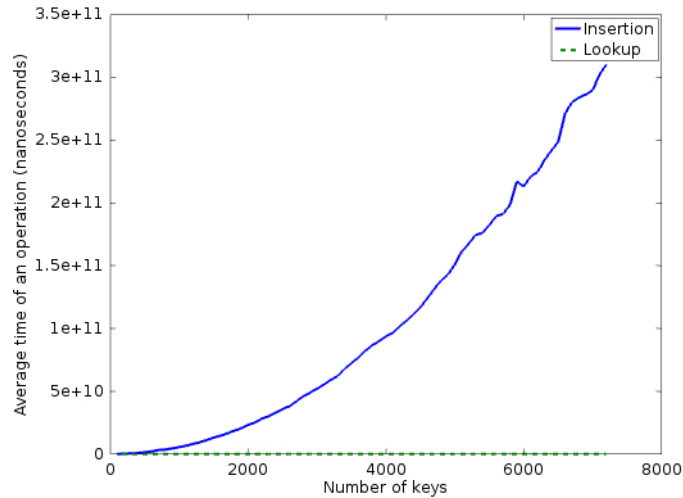


Figure 24 – The time of the emulated worst-case insertion and the emulated worst-case lookup of the cuckoo hashing with perfect rehash, in function of the number of keys in the scheme.

To emulate the worst-case lookup, we simply perform it on a key that is not in the structure. This is done 1000 times to obtain the average time of the worst case emulation. To emulate the worst-case insertion, we modify the acyclic test of our perfect hashing method so that it always fails. We start an arbitrary table with an uniform random sample of $m - 1$ keys. In every iteration, we insert the current evicted key in the same table; hence, two keys evict one another until a perfect rehash is called for. We use a hash function that always returns an arbitrary (occupied) index. This emulated worst-case insertion is done independently 10 times to obtain the average time of the worst case emulation.

For $j \in \{1, \dots, \delta(n)\}$, 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 $a_j, b_j, c_j \in \{1, 2, \dots, p\}$ 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 (51).

6.2 MH Experiments

We have conducted experiments to compare the proposed MH scheme against existing, well-known solutions. All experiments in this Section were conducted on a Ubuntu 20.04.5 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. All the experiment's source code is found in <https://github.com/judismar/monkey-hashing>.

First, we compared a MH implementation in Java against Java's inbuilt thread-safe hash map solution, the `ConcurrentHashMap` (CHM), and against Java's `ConcurrentSkipListMap` (CSLM), a non-hash-based map whose thread-safety is known to be very well implemented, to the extent that it outperforms the CHM in scenarios with a big number of threads and also Java's `TreeMap` (a non-thread-safe red-black tree) in single-threaded scenarios. Monkey hashing significantly outperformed CHM in all multi-thread test scenarios. It also outperformed CSLM when the number of threads was the same or lower as the number of CPU cores. Second, we compared a C++ implementation of MH against Intel's TBB `concurrent_hash_map`. Again, MH performed significantly better.

Figures 25 and 26 show the average running time of a sequence of read and write operations as a function of the number of concurrent threads. Our MH significantly outperforms Java's CHM. It also outperforms the CSLM up to 12 threads in the first experiment, and up to 13 threads in the second experiment. This is due to the fact that we have a single writer thread, because the CSLM's iterator runs *a lot faster* than that of our hash maps. That is why its efficiency seems to be maintained as we increase the number of threads: the writer thread is always the last one to finish.

Our scenario consists of a single writer thread and multiple reader threads. The number of items inserted in the map is 25,000 in the experiment of Figure 25, and 100,000 in the experiment of Figure 26. The target load factor was $1/2$ in all experiments. The maximum capacity and the load factor are provided as constructor arguments to pre-allocate the table size, thus avoiding resizes. This was done to MH and CHM schemes all the same. The number of concurrent threads varied from 2 to 16 (always one writer, several readers).

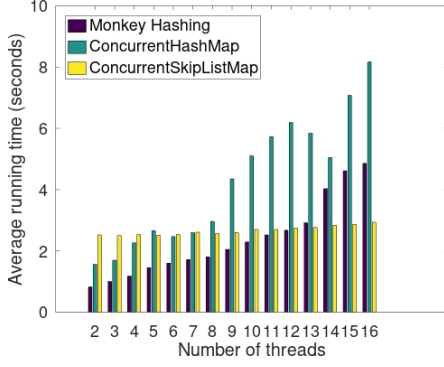


Figure 25 – 25,000 inserted items.

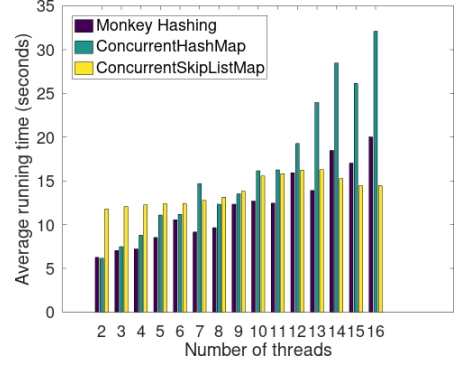


Figure 26 – 100,000 inserted items.

When we increase the number of threads, we also increase the total amount of work, because each reading thread iterates through the entries 2,000 times. The writer thread inserts 25,000 or 100,000 items and overwrites the values of all these items 1,000 times. This sequence of operations is repeated 4 times to obtain the average total running time.

In Figures 27 and 28, we compare the monkey hashing against the Intel’s TBB `concurrent_hash_map` in C++. They show the average running time of a sequence of read and write operations in function of the number of concurrent threads. Our MH significantly outperforms the TBB `concurrent_hash_map` in our experiments.

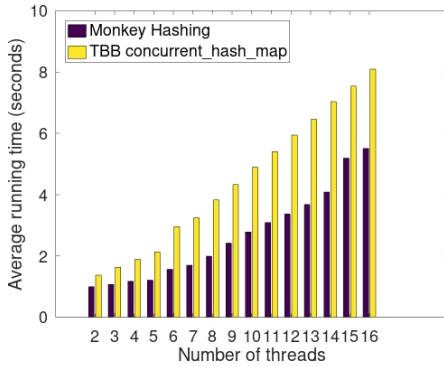


Figure 27 – 25,000 inserted items.

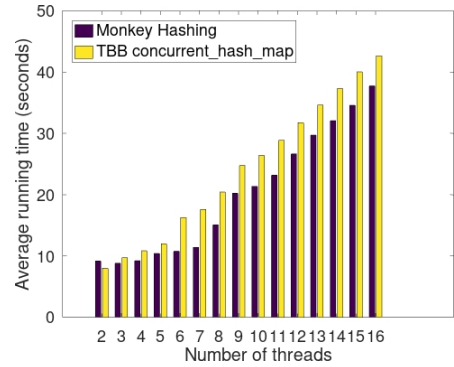


Figure 28 – 100,000 inserted items.

In this scenario of a single writer thread and multiple reader threads, the number of items inserted by the writer thread is 25,000 in the experiment of Figure 27 and 100,000 in the experiment of Figure 28. The intended load factor is 1/2. The maximum capacity and the load factor are provided as constructor arguments of the monkey hashing to pre-allocate the table size. Twice the maximum capacity is provided as the constructor argument of the TBB `concurrent_hash_map` to pre-allocate the table size, thus avoiding resizes. The number of threads running concurrently is varied from 2 to 16.

When we increase the number of threads, we also increase the total amount of work, because each reading thread iterates through the entries 1,000 times. The writer

thread inserts 25,000 or 100,000 items. This sequence of operations is repeated 5 times to obtain the average total running time.

Figures 29 and 30 show the average running time of a sequence of read and write operations in function of the number of concurrent threads in order to compare the monkey hashing with the wait-free hash map in (45) in C++. The monkey hashing ties with the Laborde et al.'s hash map.

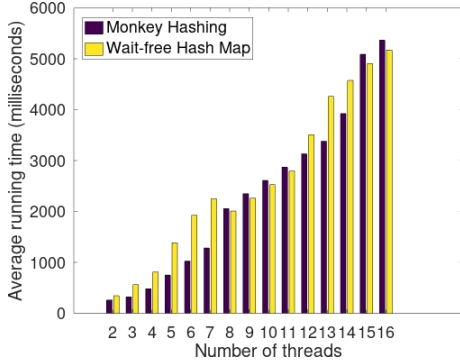


Figure 29 – 25,000 inserted items.

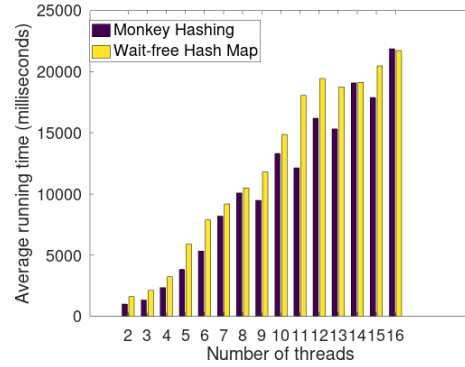


Figure 30 – 100,000 inserted items.

The scenario of the simulation in Figures 29 and 30 consists of a single writer thread and multiple reader threads. In the experiment of Figure 29, the writer thread inserts 25,000 items, and each reader thread performs 25,000 get operations. In the experiment of Figure 30, the writer thread inserts 100,000 items, and each reader thread performs 100,000 get operations. The intended load factor for the monkey hashing is $1/2$, and the maximum capacity and the load factor are provided as constructor arguments to pre-allocate its table size. The number of threads running concurrently is varied from 2 to 16.

When we increase the number of threads, we also increase the total amount of work, because each reading thread runs 25,000 or 100,000 get operations. The writer thread inserts 25,000 or 100,000 items. This sequence of operations is repeated 5 times to obtain the average total running time.

7 CONCLUSION

Our first contribution is a new cuckoo hashing variant, the Cuckoo Hashing With Perfect Rehash (CHPR). 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. We also presented experiments that corroborate the theoretical time of the basic operations of our variant. 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. Another open problem is to design a perfect rehash execution that does not increase the size of the tables of the cuckoo tree, yielding better worst-case insertion time than the time of the regular CHPR. Or, alternatively to prove that it is not possible.

We also proposed a novel thread-safe hashing scheme that is simple to understand and to implement. High concurrent performance is attained by dispensing with locks or other forms of thread synchronization, for the price of a strictly positive probability of failure during an insertion operation, a probability which can be made negligible for all intents and purposes. In the scenario of a single writer and multiple readers, for which our contribution is particularly well-suited, our scheme consistently outperforms well-known, widely used solutions such as Java’s `ConcurrentHashMap` and C++ Intel’s TBB `concurrent_hash_map`, while not presenting the drawbacks of previous works with roughly the same objectives. An open problem for the monkey hashing is allowing multiple writers. The optimistic cuckoo hashing (34), also designed for scenarios with a single writer and multiple readers, allows only one writer at a time by locking the entire structure. It is a solution for scenarios where read operations dominate¹. While this could also be done with the monkey hashing, it would still remain an open problem to augment its design to allow multiple writers *efficiently*. However, such a change would inevitably require some sort of thread synchronization², a performance-hindering inconvenience that is purposefully

¹ Anecdotal evidence suggests that, in most applications, lookup operations dominate in sets (9). Concrete evidences that read operations dominate in key-value workloads was published in (52).

² It looks possible to devise a variant of the monkey hash set (shown in Appendix B), which uses an atomic synchronization instruction and allows multiple writers (while still purposefully ignoring communication between threads).

ignored by our monkey hashing for the sake of performance — because the very nature of its operations permits it —, while still maintaining consistency.

BIBLIOGRAPHY

- 1 BENTO, L. M. d. S.; SÁ, V. G. P. d.; SZWARCFITER, J. L. Some illustrative examples on the use of hash tables. *Pesquisa Operacional*, SciELO Brasil, v. 35, p. 423–437, 2015.
- 2 CHI, L.; ZHU, X. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, ACM, v. 50, n. 1, p. 1–36, 2017.
- 3 THOMA, M. *The 3 Applications of Hash Functions*. <https://levelup.gitconnected.com/the-3-applications-of-hash-functions-fab1a75f4d3d>. Accessed: 2022-04-22.
- 4 GEEKSFORGEEKS. *Applications of Hashing*. <https://www.geeksforgeeks.org/applications-of-hashing/>. Accessed: 2022-04-22.
- 5 KNUTH, D. E. *"The Art of Computer Programming, Vol. 3: Sorting and Searching"*. second. [S.l.]: Addison-Wesley Publishing Co., 1998.
- 6 MITZENMACHER, M. Some open questions related to cuckoo hashing. *Lecture Notes in Computer Sciences*, v. 5757, 2009.
- 7 AZAR, Y. et al. Balanced allocations. *SIAM Journal on Computing*, Society for Industrial and Applied Mathematics, v. 29, n. 1, p. 180, 1999.
- 8 PAGH, R.; RODLER, F. F. Cuckoo hashing. *Journal of Algorithms*, v. 51, n. 2, p. 122–144, 2004.
- 9 HERLIHY, M.; SHAVIT, N. *The Art of Multiprocessor Programming*. [S.l.]: Elsevier, 2012.
- 10 JUNIOR, J. A. *Hashing do Cuco com Realocação Perfeita*. 2017.
- 11 CZECH, Z. J.; HAVAS, G.; MAJEWSKI, B. S. Perfect hashing. *Theoretical Computer Science*, v. 182, n. 1–2, p. 1–143, 1997.
- 12 MITZENMACHER, M.; UPFAL, E. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. [S.l.]: Cambridge University Press, 2017.
- 13 HERLIHY, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 13, n. 1, p. 124–149, 1991.
- 14 STATISTICS; DATA. *The Most Popular Programming Languages - 1965/2022 - New Update*. <https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2022-new-update>. Accessed: 2022-08-09.
- 15 CORMEN, T. H. et al. *Introduction to Algorithms*. 2. ed. [S.l.]: Elsevier, 2002.
- 16 MITZENMACHER, M.; VADHAN, S. P. Why simple hash functions work: exploiting the entropy in a data stream. In: *SODA*. [S.l.: s.n.], 2008. v. 8, p. 746–755.

- 17 CARTER, J. L.; WEGMAN, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences*, v. 18, n. 2, p. 143–154, 1979.
- 18 RABIN, M. O. Probabilistic algorithm for testing primality. *Journal of number theory*, Elsevier, v. 12, n. 1, p. 128–138, 1980.
- 19 MAJEWSKI, B. S. et al. A family of perfect hashing methods. *The Computer Journal*, v. 39, n. 6, p. 547–554, 1996.
- 20 FREDMAN, M. L.; KOMLÓŠ, J.; SZEMERÉDI, E. Storing a sparse table with $o(1)$ worst case access time. *Journal of the Association for Computing Machinery*, v. 31, n. 3, p. 538–544, 1984.
- 21 DEVROYE, L.; MORIN, P. Cuckoo hashing: Further analysis. *Information Processing Letters*, v. 86, n. 4, p. 215–219, 2003.
- 22 FOUNTOULAKIS, N.; PANAGIOTOU, K.; STEGER, A. On the insertion time of cuckoo hashing. *SIAM Journal on Computing*, SIAM, v. 42, n. 6, p. 2156–2181, 2013.
- 23 KUTZELNIGG, R. Bipartite random graphs and cuckoo hashing. In: *Discrete Mathematics and Theoretical Computer Science*. [S.l.: s.n.], 2006. p. 403–406.
- 24 FOTAKIS, D. et al. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, v. 38, p. 229–248, 2005.
- 25 DIETZFELBINGER, M. et al. Tight thresholds for cuckoo hashing via xorsat. In: SPRINGER. *International Colloquium on Automata, Languages, and Programming*. [S.l.], 2010. p. 213–225.
- 26 FRIEZE, A.; JOHANSSON, T. On the insertion time of random walk cuckoo hashing. *Random Structures and Algorithms*, v. 54, n. 4, p. 721–729, 2019.
- 27 KIRSCH, A.; MITZENMACHER, M. Using a queue to de-amortize cuckoo hashing in hardware. In: *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*. [S.l.: s.n.], 2007. v. 75.
- 28 KIRSCH, A.; MITZENMACHER, M.; WIEDER, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, v. 39, n. 4, p. 1543–1561, 2009.
- 29 HUANG, S. H. S.; WONG, C. Generalized binary split trees. *Acta Informatica*, v. 21, p. 113–123, 1984.
- 30 LAMPORT, L. The mutual exclusion problem: part i—a theory of interprocess communication. In: *Concurrency: the Works of Leslie Lamport*. [S.l.: s.n.], 2019. p. 227–245.
- 31 LAMPORT, L. The mutual exclusion problem: part ii—statement and solutions. In: *Concurrency: the Works of Leslie Lamport*. [S.l.: s.n.], 2019. p. 247–276.
- 32 DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM*, ACM, v. 8, n. 9, p. 569, 1965.
- 33 LEA, D. Hash table util.concurrent.concurrenthashmap. *JSR-166, the proposed Java Concurrency Package*, 2003.

- 34 FAN, B.; ANDERSEN, D. G.; KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: *10th USENIX Symposium on Networked Systems Design and Implementation*. [S.l.: s.n.], 2013. p. 371–384.
- 35 ISRAELI, A.; RAPPOPORT, L. Disjoint-access-parallel implementations of strong shared memory primitives. In: *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. [S.l.: s.n.], 1994. p. 151–160.
- 36 HERLIHY, M.; SHAVIT, N. On the nature of progress. In: SPRINGER. *International Conference On Principles Of Distributed Systems*. [S.l.], 2011. p. 313–328.
- 37 ASPNES, J.; HERLIHY, M.; SHAVIT, N. Counting networks. *Journal of the ACM (JACM)*, ACM, v. 41, n. 5, p. 1020–1048, 1994.
- 38 SHAVIT, N.; ZEMACH, A. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 14, n. 4, p. 385–428, 1996.
- 39 HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 12, n. 3, p. 463–492, 1990.
- 40 BAILIS, P.; GHODSI, A. Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety? *ACM Queue*, ACM, v. 11, n. 3, p. 20–32, 2013. ISSN 1542-7730.
- 41 VOGELS, W. Eventually consistent. *Communications of the ACM*, ACM, v. 52, n. 1, p. 40–44, 2009.
- 42 BREWER, E. A. Towards robust distributed systems. In: PORTLAND, OR. *PODC*. [S.l.], 2000. v. 7, n. 10.1145, p. 343477–343502.
- 43 SCARGALL, S. *Programming persistent memory: A comprehensive guide for developers*. [S.l.]: Springer Nature, 2020.
- 44 HERLIHY, M.; SHAVIT, N.; TZAFRIR, M. Hopscotch hashing. In: SPRINGER. *International Symposium on Distributed Computing*. [S.l.], 2008. p. 350–364.
- 45 LABORDE, P.; FELDMAN, S.; DECHEV, D. A wait-free hash map. *International Journal of Parallel Programming*, Springer, v. 45, n. 3, p. 421–448, 2017.
- 46 MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. [S.l.: s.n.], 2002. p. 73–82.
- 47 SHALEV, O.; SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, ACM, v. 53, n. 3, p. 379–405, 2006.
- 48 LI, X. et al. Algorithmic improvements for fast concurrent cuckoo hashing. In: *Proceedings of the Ninth European Conference on Computer Systems*. [S.l.: s.n.], 2014. p. 1–14.
- 49 GAO, H.; GROOTE, J. F.; HESSELINK, W. H. Almost wait-free resizable hashtables. In: IEEE. *18th International Parallel and Distributed Processing Symposium*. [S.l.], 2004. p. 50.

- 50 TWITTER. *New Tweets per second record, and how!*
https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how. Accessed: 2022-08-31.
- 51 DIETZFELBINGER, M.; SCHELLBACH, U. On risks of using cuckoo hashing with simple universal hash classes. In: SIAM. *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. [S.l.], 2009. p. 795–804.
- 52 ATIKOGLU, B. et al. Workload analysis of a large-scale key-value store. In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. [S.l.: s.n.], 2012. p. 53–64.
- 53 VOSS, M.; ASENJO, R.; REINDERS, J. *Pro TBB: C++ parallel programming with threading building blocks*. [S.l.]: Springer, 2019.

Appendix

APPENDIX A – THE MATHEMATICAL DEFINITION OF LINEARIZABILITY

Before presenting the mathematical definition of linearizability, we introduce the mathematical model of a concurrent execution. All this formalization is found in (39) and (9). The text in this Section is based in (39).

A.1 The Formal Model of a Concurrent Execution

A concurrent execution is modeled by a *history*, which is a finite sequence of operation *invocation* and operation *response* events. A *subhistory* of a history H is a subsequence of the events of H . An operation (function or method) invocation is denoted as $x \text{ op}(args) A$, where x is an object or data structure name, op is an operation name, $args$ denotes a sequence of argument values, and A is a thread name. The response to an operation invocation is denoted as $x \text{ term}(res) A$, where $term$ is a termination condition (e.g. *ok*, *fail*), and res is a sequence of results. A response *matches* an invocation if their object (or data structures) names agree and their thread names agree. An invocation is *pending* in a history if no matching response follows the invocation. If H is a history, $complete(H)$ is the maximal subsequence of H consisting only of invocations and matching responses.

A history H is *sequential* if (1) the first event of H is an invocation, and (2) each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by an invocation. A history that is not sequential is *concurrent*.

A *thread subhistory*, $H|T$, of a history H is the subsequence of all events in H whose thread names are T . An *object subhistory* $H|x$ is similarly defined for an object x . Two histories H and H' are *equivalent* if for every thread T , $H|T = H'|T$. A history H is *well-formed* if each thread subhistory $H|T$ of H is sequential.

An *operation* e in a history is a pair consisting of an invocation, $inv(e)$, and the next matching response, $res(e)$. We denote an operation by $[x \text{ inv}/res A]$, where x is an object and A is a thread name. An operation e_0 *lies within* another operation e_1 in H if $inv(e_1)$ precedes $inv(e_0)$ and $res(e_0)$ precedes $res(e_1)$ in H . Object and thread names are omitted when they are clear from the context.

An example of a concurrent execution of a FIFO queue q is seen in Figure 31, adapted from (39).

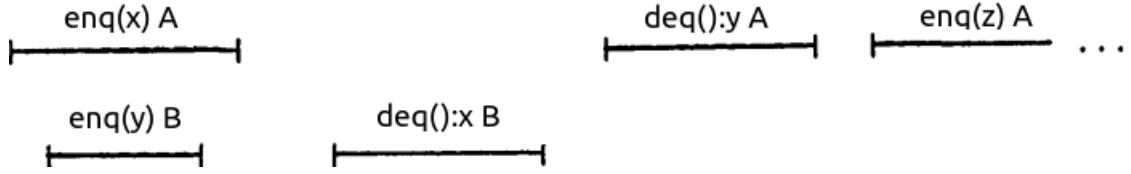


Figure 31 – A graphical representation of a concurrent execution of a FIFO queue. A and B are threads, and the return value of the dequeue operation is given after the colon.

The history H of Figure 31 is the following well-formed history for q , where $enq()$ is its enqueue operation and $deq()$ is its dequeue operation.

```

q enq(x) A
q enq(y) B
q ok() B
q ok() A
q deq() B
q ok(x) B
q deq() A
q ok(y) A
q enq(z) A

```

The first event in H is an invocation of enq with argument x by thread A, and the fourth event is the matching response with termination condition ok and no results. The $[q\ enq(y)/ok() B]$ operation lies within the $[q\ enq(x)/ok() A]$ operation. The subhistory, $complete(H)$, is H with the last (pending) invocation of enq removed. Notice that reordering the first two events yields one of many histories equivalent to H, and that we could have omitted the object name q from H for simplicity.

A *sequential specification* for an object is a set of sequential histories for the object. In order to define a sequential specification, we write axioms that gives the pre-conditions and post-conditions on the operations of the object (we use the notation of (39)). The axiom for the $enq()$ operation is

$$\begin{aligned}
& \{\text{true}\} \\
& enq(x) / ok() \\
& \{q' = ins(q, x)\},
\end{aligned}$$

where $ins(q, x)$ denotes the queue q after the insertion of item x . Notice that the pre-condition is always true, which means that an enqueue can be performed regardless of

the state of our queue. We denote such state as the *value* of the queue. The post-condition is a queue value after the insertion of x .

The axiom for the $deq()$ operation is

$$\begin{aligned} & \{q \neq []\} \\ & deq() / ok(x) \\ & \{q' = rest(q) \wedge x = first(q)\}. \end{aligned}$$

The pre-condition for the $deq()$ operation is that the queue is not empty. The post-condition is a queue value $rest(q)$, that is, the queue q without x , and the return of $first(q)$, which is the item x that was in the head of the queue. In our axiom, the operation is left undefined for empty queues.

A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification for x .

A.2 The Formal Definition of Linearizability

A history H induces an irreflexive partial order $<_H$ on operations:

$$e_0 <_H e_1 \text{ if } res(e_0) \text{ precedes } inv(e_1) \text{ in } H.$$

Informally, $<_H$ captures the real-time precedence ordering of operations in H . Operations unrelated by $<_H$ are said to be *concurrent*. If H is sequential, $<_H$ is a total order.

A history H is *linearizable* if it can be extended (by appending zero or more response events) to some history H' such that:

1. $complete(H')$ is equivalent to some legal sequential history S , and
2. $<_H \subseteq <_S$.

Informally, extending H to H' captures the notion that some pending invocations may have taken effect even though their responses have not yet been returned to the caller (e.g. the pending $enq(z)$ in Figure 31). Restricting attention to $complete(H')$ captures the notion that the remaining pending invocations have not yet had an effect. Item 1 states that threads act as if they were interleaved at the granularity of complete operations. Item 2 states that this apparent sequential interleaving respects the real-time precedence ordering of operations.

We call S a *linearization* of H . For each H , there may be more than one extension H' satisfying items 1 and 2, and for each extension H' , there may be more than one linearization S . A *linearizable object* is one whose concurrent histories are linearizable with respect to some sequential specification.

A.3 Examples

The history H corresponding to the concurrent computation shown in Figure 31 is linearizable. If we append $q\ ok() A$ to it, the resulting history is equivalent to the following sequential history:

```

enq(x) A
ok() A
enq(y) B
ok() B
deq() B
ok(x) B
deq() A
ok(y) A
enq(z) A
ok() A.

```

Figure 32, adapted from (39), shows another concurrent computation with a queue.

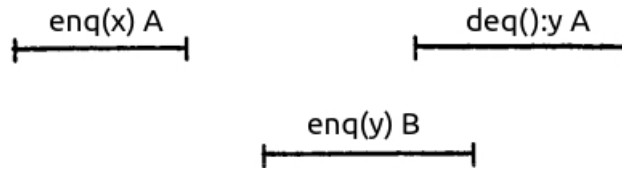


Figure 32 – A graphical representation of a concurrent execution of a FIFO queue. A and B are threads, and the return value of the dequeue operation is given after the colon.

Its corresponding history H_2 is

```

enq(x) A
ok() A
enq(y) B
deq() A
ok() B

```

ok(y) A.

H_2 is not linearizable because the complete $enq(x)$ operation precedes $enq(y)$, but y is dequeued before x .

Notice that linearizability does not rule out histories such as

enq(x) A

deq() B

ok(x) B,

in which an operation “takes effect” before its response event occurs. We can extend this history by appending $q\ ok() A$, which is equivalent to

enq(x) A

ok() A

deq() B

ok(x) B.

APPENDIX B – MONKEY HASH SET VARIANT THAT SUPPORTS MULTIPLE WRITER THREADS

Our monkey hash set variant, which allows multiple writers, requires not only that single-word read and write operations are atomic, but also that the computing system supports the *compare-and-swap* (CAS) synchronization instruction.

B.1 The CAS Instruction

The CAS instruction is the most common technique for implementing non-blocking concurrent algorithms (9). In fact, it is used by Java’s `ConcurrentHashMap` (as seen in the code found in <http://hg.openjdk.java.net/jdk8/jdk8>), by C++ Intel’s TBB library (53) and by the wait-free hash map by Laborde et al. (45), to name a few.

CAS is an atomic instruction that compares the current value in a memory location (memory address) with a given *expected* value and, if they are the same, it changes the contents of the memory location with a given *update* value and returns *true*. If the compared values are not the same, it does not update the contents of the memory location and returns *false*. We denote this operation in our pseudo-code as *CAS*, and it takes a memory address, the expected value and the update value as arguments. For example, consider the call

$$\text{CAS}(\&x, a, b),$$

where $\&x$ denotes the address of the variable x , a is the expected value and b is the update value. If $x = a$, b is thus stored in $\&x$ and the call returns *true*. Otherwise (if $x \neq a$), x is not updated and the call returns *false*. Let $x = 3$, $a = 3$ and $b = 5$. Because the operation is atomic, if another thread is reading x at about the same time that the *CAS* instruction is in process, it will either read 3 (if the reading is done before the *CAS*), or it will read 5 (if *CAS* is done before the reading).

B.2 The Variant

Our monkey hash set variant is very similar to the original monkey hashing, although it is even simpler in terms of algorithms. It consists of a single hash table and

$k \geq 1$ hash functions that may be obtained from a universal class. It is also required that one knows beforehand the table's maximum capacity to pre-allocate memory, in order to enforce the intended load factor without the need of rehashes.

Because it is a set, its basic operations are three: lookup, insert and remove. The operation that iterates through the collection of keys is implemented as in the original MH, and it is also eventual consistent. Our variant is *consistent* if we consider only the three basic operations — and thus eventual consistent, by definition.

This variant allows multiple writer threads. In fact, any thread is allowed to perform read operations and write operations, without getting to a case of inconsistency. This is guaranteed by its eventual consistency, which will be demonstrated.

B.2.1 The Lookup Algorithm

The lookup algorithm of our variant is found in Figure B.1. Notice that we no longer use the number of hash functions that were actually called for during insertions.

```

SetLookup( $x$ )
1. for  $i = 1, 2, \dots, k$ 
2.    $p \leftarrow h_i(x)$ 
3.   if  $T[p] = x$ 
4.     return true
5. return false

```

Algorithm B.1 – The monkey hash set lookup algorithm.

The lookup algorithm is linearizable, hence it is also eventual consistent. The linearization point for successful lookups is line 3: after we read $T[p]$ atomically, concurrent removals and insertions are unable to change the return of *true* — regardless of how much time it takes for the rest of the instruction to finish. For failed lookups, the linearization point is line 5.

B.2.2 The Insert Algorithm

The insert algorithm of our variant is found in Figure B.2.

The insertion algorithm is linearizable. In the case of a successful insertion, its linearization point is line 5: its effect would depend on comparing and setting, which is done atomically by the *CAS* instruction. In the case x is already found within T , its linearization point is line 3 of the lookup algorithm. In the case of a failed insertion, its linearization point is line 7.

```

SetInsert( $x$ )
1. if SetLookup( $x$ )
2.   return
3. for  $i = 1, 2, \dots, k$ 
4.    $p \leftarrow h_i(x)$ 
5.   if CAS(& $T[p]$ ,  $\perp$ ,  $x$ )
6.     return
7. fail

```

Algorithm B.2 – The monkey hash set insertion algorithm.

B.2.3 The Remove Algorithm

The remove algorithm of our variant is found in Figure B.3.

```

SetRemove( $x$ )
1. for  $i = 1, 2, \dots, k$ 
2.    $p \leftarrow h_i(x)$ 
3.   if CAS(& $T[p]$ ,  $x$ ,  $\perp$ )
4.     return
5. return

```

Algorithm B.3 – The monkey hash set removal algorithm.

The removal algorithm is linearizable. In the case of a successful removal, its linearization point is line 3: much like in the insertion algorithm, its effect would depend on comparing and setting, which is done atomically. In the case x is not found, its linearization point is line 5.