An Overview of Cuckoo Hashing

Charles Chen

1 Abstract

Cuckoo Hashing is a technique for resolving collisions in hash tables that produces a dictionary with constant-time worst-case lookup and deletion operations as well as amortized constant-time insertion operations. First introduced by Pagh in 2001 [3] as an extension of a previous static dictionary data structure, Cuckoo Hashing was the first such hash table with practically small constant factors [4]. Here, we first outline existing hash table collision policies and go on to analyze the Cuckoo Hashing scheme in detail. Next, we give an overview of (c, k)-universal hash families, and finally, we summarize some selected experimental results comparing the varied collision policies.

2 Introduction

The *dictionary* is a fundamental data structure in computer science, allowing users to store key-value pairs and look up by key the corresponding value in the data structure. The hash table provides one of the most common and one of the fastest implementations of the dictionary data structure, allowing for amortized constant-time lookup, insertion and deletion operations.[1] In its archetypal form, a hash function h computed on the space of keys is used to map each key to a location in a set of bucket locations $\{b_0, \cdots, b_{r-1}\}$. The *key*, as well as, potentially, some auxiliary data (termed the *value*), is stored at the appropriate bucket location upon insertion. Upon deletion, the key with its value is removed from the appropriate bucket, and in lookup, such a key-value pair, if it exists, is retrieved by looking at the bucket location corresponding to the hashed key value. Here, for a hash table T with hash function h and a key x, we denote by $T[x]$ the bucket location b_x.

This general scheme of using hash functions to distribute key-value pairs uniformly across the buckets allows hash-based dictionaries to achieve superior expected asymptotic performance over other dictionary schemes, including, for example, balanced binary tree dictionaries. In search trees, a fixed ordering is determined on the set of possible keys and this is consulted to locate a particular key. In hash tables, hash functions on keys are used to map possible keys to buckets in such a way that it is unlikely that too distinct keys are mapped to the same bucket. However, in the absence of perfect hash functions mapping keys injectively to locations in a set of buckets, collisions where two distinct keys map to a single bucket location must be resolved.

A number of schemes to resolve collisions with different performance characteristics have been introduced. Cuckoo Hashing[2] is a simple scheme for resolving collisions that allows for constant-time worst-case lookup and deletion operations, as well as amortized constant-time

[1] We say that an operation is performed in amortized constant time in the sense that though a particular invocation of an operation may incur a large runtime cost (for instance, in the case of hash tables, tables may need to be resized to accommodate an increased number of elements), averaged over a large number of operations, the average runtime over these invocations is constant.

[2] Cuckoo Hashing is named for the brood parasitism behavior of the cuckoo bird: the cuckoo bird is

insertion operations. Briefly, Cuckoo Hashing maintains two hash tables T_1 and T_2 with two independent hash functions h_1 and h_2, respectively, and a key is stored in exactly one of two locations, with one possible location in each hash table. Lookup of a key x is performed by looking for a matching entry in either the bucket location $T_1[h_1(x)]$ or the bucket location $T_2[h_2(x)]$. Deletion can be performed simply by removing the key from either possible bucket location, but insertion is more complex and is analyzed in detail in Section 5. A detailed description of the scheme is described in Section 4.

In order to have a broader view of the range of hashing schemes and to give a baseline for theoretical and empirical comparisons with Cuckoo Hashing, we first introduce in our next section the details of several previous hashing schemes: chained hashing, linear probing and two way chaining.

3 Previous hashing schemes

For our hash table descriptions here, we use n as the number of items inserted into a hash table, r as the number of buckets in a hash table and define the load factor n/r as a measure of the expected number of buckets.

Chained hashing. In a chained hashing scheme, a linked list is used to store all keys hashing to a given location in the set of buckets. Collisions are therefore resolved by lengthening the list structures in buckets with collisions. Insertion operations can be done in constant time by appending or prepending to the list for the relevant bucket, but lookup and deletion operations may require traversal of the entire list. Performance of the hash table then degrades when these lists grow large, i.e. when the load factor for the hash table is high. By choosing appropriate times to resize the hash table, lookup and deletion operations can be made to perform in amortized constant time, but in particular, unlike Cuckoo Hashing, there is no constant-time worst-case lookup guarantee.

Linear probing. In a linear probing scheme, keys are hashed to form a bucket index and if possible, are stored in that bucket location. On insertion, if the bucket location is already taken by another key, the bucket locations are scanned in order to find a free bucket to store the key. During lookup for a certain key, that key is hashed to provide a bucket location, and consecutive buckets starting with that bucket are scanned until the key is found or an empty bucket is encountered. Certainly, these insertion and lookup schemes do not have reasonable constant-time worst-case runtime bounds, since large clusters of contiguous non-empty buckets may force many buckets to be accessed before a correct bucket is found. Deletion of a key can be performed by marking the bucket containing that key with a special `DELETED` value, which is retained until the table is periodically rebuilt or resized. A more complex deletion scheme examines the cluster of consecutive non-empty buckets containing

known for laying eggs in the nests of other birds, and when these eggs hatch, the cuckoo hatchlings often evict the eggs of the host species. In the context of Cuckoo Hashing, when keys are inserted into buckets of a hash table that have preexisting keys, the preexisting keys are evicted and forced to move to their alternate locations in the other hash table: in some sense, then, this mimics the behavior of the cuckoo bird.

the key to be deleted, empties the bucket containing that key and if necessary, rearranges keys to fill the ”hole” left by the removed key so that lookup operations will succeed.

Linear probing is a type of *open addressing* collision resolution scheme, where a hash collision is resolved by *probing* through a set of alternate locations in an array of buckets. Similar open addressing schemes include *quadratic probing*, where the interval between probes increases quadratically, and *double hashing*, where the distance between probes for a given key is linear, but is given by the value of another hash function on that key.

Two-way chaining. In a two-way chaining scheme, two hash tables with two independent hash functions are used so that each possible key hashes to one bucket location in each hash table. Each bucket holds a list of items that hash to that bucket, as in the case of chained hashing. When a key is inserted, it is inserted into the table whose corresponding bucket location has the list with the fewest number of keys. During lookup and deletion, lists in both tables are traversed to locate a given key. Performance here can be better than in the chained hashing case because the action of choosing the bucket with the fewest number of existing keys serves to, with high probability, “even out” the distribution of keys to buckets. Analysis by Azar and Broder [1] demonstrates that this scheme allows for $O(\log\log n)$ maximal list length and lookup time with high probability, assuming random hash functions. This scheme shares with Cuckoo Hashing its idea of designating possible locations in two hash tables to store a given key.

4 Description of Cuckoo Hashing

Cuckoo hashing is a hash table scheme using two hash tables T_1 and T_2 each with r buckets with independent hash functions h_1 and h_2 each mapping a universe U to bucket locations $\{0, \cdots, r-1\}$. A key x can be stored in exactly one of the locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$. A lookup operation in Cuckoo Hashing examines both locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$ and succeeds if the key x is stored in either location. Formally, then, the following algorithm describes the lookup behavior of Cuckoo Hashing.

```
function lookup(x)
    return $T_1[h_1(x)] = x \;\vee\; T_2[h_2(x)] = x$
end
```

Certainly, this algorithm performs in worst-case constant time. Deletion is also a simple operation, since we may simply remove a key from its containing bucket. The main question for Cuckoo Hashing and the subject of the bulk of our analysis is then, given hash functions h_1 and h_2 and a set of keys to be stored in our hash tables, how likely is it that these keys be placed into buckets so that for each key x, either $T_1[h_1(x)] = x$ or $T_2[h_2(x)] = x$; further, can we demonstrate a scheme to efficiently insert keys to reach such a configuration? Pagh previously demonstrated that for randomly chosen h_1 and h_2, if $r \geq (1+\epsilon)n$ [3], i.e. each of the two hash tables has size at least $1+\epsilon$ times the total number of keys for some ϵ, the probability that the keys cannot be put in such a configuration is $O(1/n)$. More constructively, we demonstrate a simple insertion procedure that allows for amortized constant-time insertion operations.

In our insertion procedure, we place a key in one of its designated places in the two tables, evicting any key already present. We subsequently try to insert the evicted key into its alternative bucket location in the other hash table, potentially evicting another key, and continuing in this way until a key is inserted into an empty bucket, or after a number of iterations, we decide to rehash the table and try again. A simple instance of this procedure is given in Figure 4.

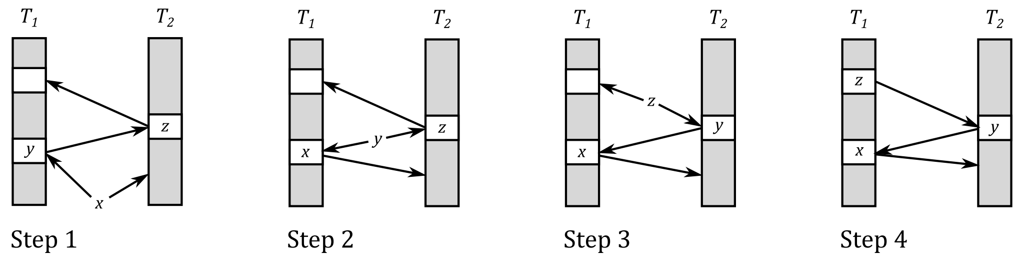

Figure 1: An instance of the insertion procedure for x.

As can be imagined, the general insertion procedure is more complicated, and certain keys may be revisited multiple times. In some instances, there does not exist a placement of all existing keys into the Cuckoo Hashing structure, so a rehash of the table will be required. We will thus choose a threshold of MaxLoop iterations of the insertion loop, after which retry with a rehashed table.

Formally, this is described by the following algorithm, where we make the arbitrary choice to initially attempt insertion into T_1.[3] Here, we use $a \leftrightarrow b$ to denote the operation of swapping the values of a and b, and we use $\perp$ to denote the empty value.

```
procedure insert(x)
    if lookup(x) then return
    loop MaxLoop times
        $x \leftrightarrow T_1[h_1(x)]$
        if $x = \perp$ then return
        $x \leftrightarrow T_2[h_2(x)]$
        if $x = \perp$ then return
    end loop
    rehash(); insert(x);
end
```

We analyze this procedure in the next section and prove that using a value of MaxLoop $= \lceil 3 \log_{1+\epsilon} r \rceil$ will result in an amortized constant-time insertion operation.

[3]Intuitively, choosing to initially insert x into T_1, as opposed to another scheme, of say, inspecting $T_2[h_2(x)]$ if $T_1[h_1(x)]$ is taken by some other key y, has negligible expected cost in terms of iterations in the insertion loop because with random hash functions, the probability that $T_2[h_2(x)]$ is taken is equal to the probability that the position $T_2[h_2(y)]$ of the evicted key y is taken, and furthermore, the number of cell accesses in each case is expected to be equal. Experimentally, Pagh and Rodler [4] have found that with a load factor of 1/3, only 10% of new keys are placed in T_2 when both $T_1[h_1(x)]$ and $T_2[h_2(x)]$ are inspected, so there is little advantage in doing an extra check.

As with many other hashing schemes, performance deteriorates as the load factor r/n increases. We therefore maintain that our hash tables each have size at least $r \geq (1+\epsilon)n$ for some constant ϵ (this invariant will be useful during our runtime analysis in the next section). We do this by doubling the size of the table and rehashing the table whenever $r < (1+\epsilon)n$ after an insertion, and we show in the next section that per insertion, this is an amortized constant-time operation.

5 Runtime analysis of Cuckoo Hashing

In this section, we prove that expected number of iterations in the insertion loop for a given key is constant. We further prove that the expected cost of rehashing a Cuckoo Hashing structure with n elements takes expected time $O(n)$ and use this result to conclude that the Cuckoo Hashing insertion operation, including any rehashing, is an amortized constant-time operation.

5.1 Hash family assumption

In our analysis, because we will consider the probability a set of keys will hash to a certain arrangement of buckets, we need a universality guarantee on the hash family from which h_1 and h_2 are selected. Pagh and Rodler develop the notion of (c,k)-universal hash functions to provide this guarantee. For simplified analysis, we make the assumption that with probability $1 - O(1/n^2)$, our hash functions h_1 and h_2 are both drawn randomly from all mappings $O \to 0, \cdots, r-1$, and with probability $O(1/n^2)$, we make no assumptions about h (here, h could be a pathological mapping). In Section 6, we develop the notion of (c,k)-universality and show that with such a weaker universality assumption, the runtime analysis developed in this section continues to hold.

5.2 Resizing analysis

Here, we show that the amortized cost of resizing the hash table per insertion is $O(1)$. For our analysis, we maintain that our hash tables each have size at least $r \geq (1+\epsilon)n$ for some constant ϵ. We do this by doubling the size of the hash table as necessary whenever $r < (1+\epsilon)n$. We will prove in Section 5.2 that rehashing a table of size n requires $O(n)$ expected time. For a given number n of items inserted into the hash table, we expect $O(\log n)$ resizing and rehashing operations of sizes $n, n/2, \cdots, n/2^{O(\log n)}$ for a total runtime of at most $O(n + n/2 + \cdots + n/2^{O(\log n)}) < O(n + n/2 + n/4 + \cdots) = O(n)$. Amortizing this cost over the n items, the amortized cost of resizing the hash table as necessary per insertion is then $O(1)$.

5.3 Insertion iteration analysis

Here, we show that the expected number of iterations spent in the insertion loop for a certain key is $O(1)$. We first analyze three cases for the progression of insertion operations. Next, we prove a lemma on the series of keys encountered. Finally, we prove an upper bound on the expected number of iterations in the insertion loop.

5.3.1 Insertion overview

Consider the insertion of key x into the Cuckoo Hashing structure along with the subsequent series of evictions and recursive insertions. If the bucket chosen to hold x is taken by some other key y, the key y is evicted to its alternative position in the other hash table, and if that bucket is taken by the key z, that key is evicted to its alternative position and so forth. We call this series of keys $x_1 = x, x_2 = y, x_3 = z, \cdots$ the sequence of nestless keys. This procedure can proceed in three ways:

Case 1. All nestless keys are distinct, implying that the insertion process terminates.

Case 2a. Nestless keys repeat, i.e. the nestless key x_i is evicted for the second time at position x_j (here and in the next case, for some (i, j), $x_i = x_j$, and we assume (i, j) is the lexicographically smallest ordered pair for which this condition holds), but the insertion process terminates when a nestless key moves to an empty bucket.

Case 2b. Nestless keys repeat, but the insertion process does not terminate.

Analysis of Case 1: all nestless keys are distinct. This is the simplest case to describe: here, a series of evictions results in the final nestless key x_p to be evicted to an empty bucket, ending the insertion procedure. An example of this case is given in Figure 5.3.1: in these figures, we represent a key as a labeled circle with arrows to the two buckets, shown as squares, which can hold it. When a key is stored in a bucket, its one emanating arrow points to the alterative bucket in the other hash table that can hold it (here, we omit for simplicity the association of buckets to the two hash tables). When a bucket as a result of our insertion procedure holds a different key than at the start of the procedure, we label the bucket outline in bold.

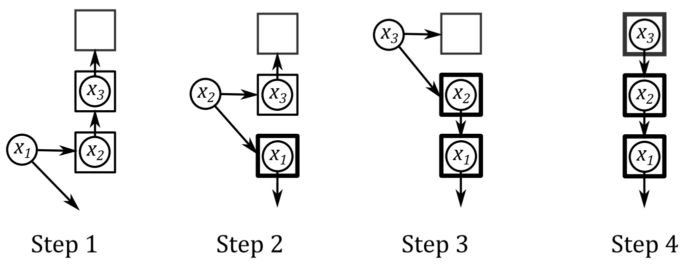

Figure 2: Case 1: the insertion of x_1 evicts x_2, which in turn evicts x_3, which finally moves to an empty bucket.

Analysis of Case 2: Nestless keys $x_1 = x, x_2, \cdots$ repeat. Here, suppose the nestless key x_i is evicted for the second time at position x_j, where $x_i = x_j$ and $i < j$, where we choose (i, j) to be the lexicographically smallest ordered pair for which this condition holds. In this case, after $x_i = x_j$ is evicted for the second time, the insertion procedure retraces its series of evictions so that the keys that are evicted twice, starting with $x_i = x_j$, continuing with $x_{i-1} = x_{j+1}, \cdots$, up to $x_1 = x_{j+i-1}$, return back to their original positions (see steps 9-12 in Figure 5.3.1). Once $x_1 = x_{j+i-1}$ is evicted back to being a nestless key, the insertion procedure continues with nestless keys $x_{j+i}, \cdots, x_l$ $(l \geq j + i - 1)$, where either x_l is moved to an empty bucket (Case 2a) or x_l evicts a previously key x_m (Case 2a). In Case 2a, the insertion procedure terminates, while in Case 2b, a "closed loop of $l - i$ keys (i keys, i.e. $x_i = x_j, x_{i-1} = x_{j+1}, \cdots, x_1 = x_{j+i-1}$, are duplicated) have hashes among $l - i - 1$ buckets (at this point in the insertion, the distinct nestless keys in buckets are exactly $x_2, \cdots, x_{j-1}$ and $x_1 = x_{i+j-1}, \cdots x_{l-1}$ for $(j - 2) + (l - i - j + 1) = l - i - 1$ buckets), meaning a rehash must be performed. We detail an example execution of Case 2a in Figure 5.3.1 (Case 2b is similar).

5.3.2 Insertion loop lemma

We will use the following lemma in our analysis of the expected number of iterations in the insertion loop.

Lemma 5.1. *Suppose at some step p, the insertion procedure for x produces a sequence $x_1 = x, x_2, \cdots, x_p$ of nestless keys, where no closed loop has yet been formed. Then, in $x_1, \cdots, x_p$, there exists a consecutive subsequence of length $l \geq p/3$ $x_q, x_{q+1}, \cdots, x_{q+l-1}$ of distinct nestless keys for which $x_q = x_1 = x$.*

Proof. Suppose all nestless keys $x_1, \cdots, x_p$ are distinct: then, $x_1, \cdots, x_p$ is such a sequence, so the lemma trivially holds. Now, if $p < i + j$, we have that the first $j - 1$ keys $x_1, \cdots, x_{j-1}$ are distinct, so since $j > i$, $j - 1 \geq (i + j - 1)/2 \geq p/2$, so $x_1, \cdots, x_{j-1}$ is the desired sequence. Now, if $p \geq i + j$, consider the sequences of distinct keys $x_1, \cdots, x_{j-1}$ and $x_{i+j-1}, \cdots, x_p$: we claim one of these must be of length at least $p/3$. These two sequences have $j - 1$ and $p - i - j + 2$ keys, respectively. Note that $p = (j - 1) + (i - 1) + (p - i - j + 2)$, and we know $j - 1 > i - 1$. If $j - 1 > p - i - j + 2$, then we know $3(j - 1) > p$, and otherwise, $3(p - i - j + 2) \geq p$. In either case, there is a sequence with properties as above. □

5.3.3 Insertion loop analysis

Here, consider the probability that the insertion procedure for x runs to produce a sequence of $k \leq 2$MaxLoop nestless keys $x_1 = x, x_2, \cdots, x_k$ (where we select some value of MaxLoop later to be sublinear in n; further, note that our insertion algorithm processes at most 2 nestless keys per iteration loop). This can happen under three non-mutually exclusive cases:

1. The hash functions h_1 and h_2 were not chosen randomly. Our assumption from Section 5.1 provides that this happens with probability at most $O(1/n^2)$.

2. The insertion procedure has not yet entered a closed loop, as in Case 2 of Section 5.3.1. We will bound this probability by $2(1 + \epsilon)^{-\frac{k-1}{3}+1}$, where ϵ is as in our table size invariant $r \geq (1 + \epsilon)n$.

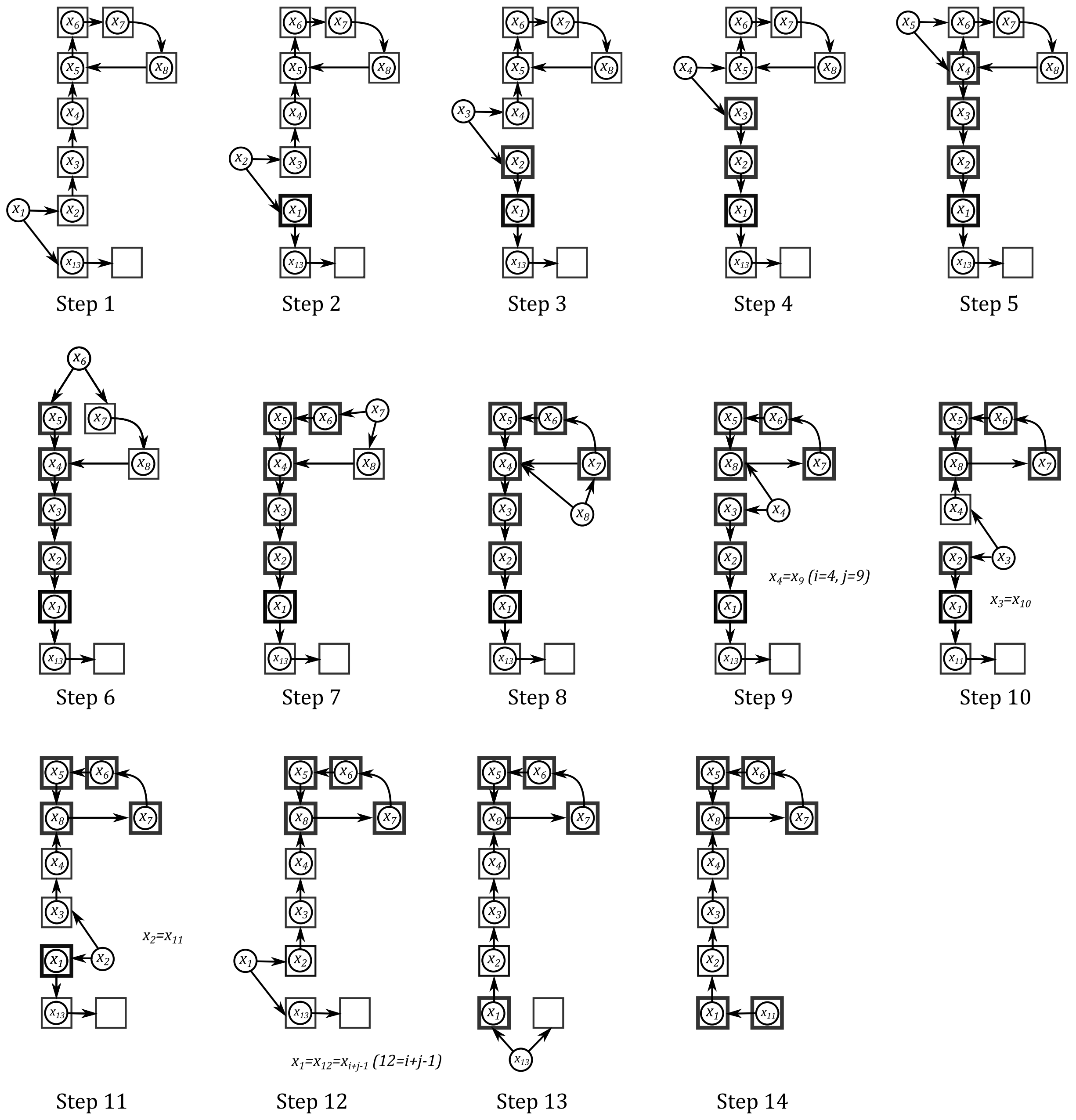

Figure 3: An instance of insertion described in Case 2a.

3. The insertion procedure has entered a closed loop. We will bound this probability by $O(1/n^2)$.

We can then compute the expected number nestless keys produced in the iteration loop as

$$1 + \sum_{k=2}^{2\text{MaxLoop}} \left[2(1+\epsilon)^{-\frac{k}{3}+1} + O(1/n^2)\right] \leq 1 + O(\text{MaxLoop}/n^2) + 2\sum_{k=0}^{\infty}(1+\epsilon)^{-k/3}$$

$$= O(1) + \frac{O(1)}{1-(1+\epsilon)^{-1/3}} = O(1). \tag{5.1}$$

Analysis of case 2 When the insertion procedure has not entered a closed loop, Lemma 5.1 provides that for $v = \lceil k/3 \rceil$, there is a sequence of consecutive distinct nestless keys $b_1 = x, b_2, \cdots, b_v$ among the nestless keys in our insertion procedure. Since consecutive nestless keys in our insertion procedure by definition have hash functions equal, alternating among the two hash functions for each consecutive pair, for either $(\beta_1, \beta_2) = (1, 2)$ or $(\beta_2, \beta_2) = (2, 1)$,

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \cdots. \tag{5.2}$$

We wish to bound the probability this case occurs for each of the two choices of (β_1, β_2). Given the choice of $b_1 = x$, there are less than n^{v-1} possible ways to choose possible sequences of v distinct keys and further, since we assume that h_1 and h_2 are random functions from the key space to $\{0, 1, \cdots, r-1\}$, the equalities in 5.2 hold with probability $r^{-(v-1)}$. Together, this implies that the probability that there exists $b_1, \cdots, b_v$ as above is at most

$$2n^{v-1}r^{-(v-1)} = 2(r/n)^{-(v-1)} < 2(1+\epsilon)^{-\lceil k/3 \rceil+1} \leq 2(1+\epsilon)^{-k/3+1}, \tag{5.3}$$

where we used our invariant $r/n < 1 + \epsilon$. Since the probability of this case is bounded above by the probability that $b_1, \cdots, b_v$ as above exist, this case has probability bounded by 5.3.

Analysis of case 3 Here, we wish to bound the probability that our sequence of nestless keys $x_1, \cdots, x_k$ has entered a closed loop. Let $v \leq k$ be the number of distinct nestless keys. We can choose the keys other than $x_1 = x$ in less than n^{v-1} ways and among the keys, we can assign them into buckets in r^{v-1} ways. In a closed loop, as in Section 5.3.1, we defined x_i and x_j to be such that (i, j) is the lexicographically first ordered pair for which $x_i = x_j$, $i \neq j$, and we defined x_l to be the first nestless key $l \geq i + j$ for which x_l has been encountered previously as $x_l = x_o$ for $1 \leq o \leq i + j - 1$. Here, as a crude approximation, we have at most v^3 possible values for i, j and l. Since each we assume our hash functions h_1 and h_2 are random, the probability for each configuration described above is r^{-2v}, since each of the v nestless keys y must in each configuration have set values of $h_1(y)$ and $h_2(y)$, each of which occurs with probability $1/r$. Putting this together and summing over possible v, we get that the probability of this case is at most

$$\sum_{v=3}^{l} n^{v-1}r^{v-1}v^3r^{-2v} \leq \frac{1}{nr}\sum_{v=3}^{\infty} v^3(n/r)^v < \frac{1}{nr}\sum_{v=3}^{\infty} v^3(1+\epsilon)^{-v} = \frac{1}{nr}O(1) = O(1/n^2), \tag{5.4}$$

where we used our invariant $r/n < 1 + \epsilon$, as well as the fact that we maintain $r = O(n)$.

This concludes our analysis on the number of insertion iterations. Equation 5.1 thus implies that the expected number of iterations in the insertion loop for a given key is $O(1)$.

5.4 Rehashing analysis

Here, we show that for a given insertion, with the appropriate value of MaxLoop, the probability of performing a rehash (i.e. reaching MaxLoop iterations in the insertion algorithm) is $O(1/n^2)$. Next, we show that rehashing succeeds after $O(n)$ operations. We then conclude that the cost of rehashing per insertion is $O(1/n)$, which is bounded by a constant.

In our previous section, we showed that for a number k of nestless keys $x_1, x_2, \cdots, x_k$, the probability that the hash functions were not chosen randomly, along with the probability that the insertion procedure entered a closed loop were both bounded above by $O(1/n^2)$. Further, the probability the procedure had not entered a closed loop was $2(1+\epsilon)^{-\frac{k}{3}+1}$. If we let $k = 2\text{MaxLoop}$ be the total possible number of possible nestless keys, where $\text{MaxLoop} = \lceil 3 \log_{1+\epsilon} r \rceil$, we will then have the probability of k iterations without entering a closed loop be

$$2(1+\epsilon)^{-\frac{k}{3}+1} = O(2(1+\epsilon)^{-2\log_{1+\epsilon} r}) = O(1/n^2).$$

Since we rehash after $k = 2\text{MaxLoop}$ nestless keys (again, note that we obtain two nestless keys per iteration of the insertion), the probability of any given insertion causing a rehash is then $O(1/n^2)$.

Now, note that during a rehashing operation, n insertions are performed, each requiring $O(1)$ time and failing with probability $O(1/n^2)$, so all insertions succeed with probability $1 - O(1/n)$. For sufficiently large n, this probability becomes $1 - \delta$ for some $1/2 > \delta > 0$, so counting any recursive rehashing operations, we succeed in rehashing after $O(n)$ operations.

We thus conclude that the cost of rehashing per insertion is $O(1/n)$, which is bounded by a constant. Thus, the insertion of a key x has amortized runtime $O(1)$.

6 (c, k)-universal hash families

In section 5.1, we made strong assumptions about the family from which the hash functions h_1 and h_2 was drawn. Here, we introduce a somewhat weaker hash family assumption in the concept of (c, k)-universality used by Pagh and Rodler [4].

6.1 Definition

We define a family $\{h_i\}_{i \in I}$ of hash functions mapping a universe U to R be (c, k)-universal if for any k distinct elements $x_1, \cdots, x_k \in U$ and any values $y_1, cdots, y_k \in R$,

$$\Pr_{i \in I}[h_i(x_1) = y_1, \cdots, h_i(x_k) = y_k] \leq c/|R|^k. \tag{6.1}$$

Intuitively, this means that over this hash family, any k elements hash to specific values in R with probability at most c times the probability if h_i were a random function from U to R. In the case of $(c, k) = (1, 2)$, we recover the common notion of a "standard universal hash family. Note that such $(1, 2)$-universal hash families are not sufficient in our analysis because here, we need guarantees on more than 2 hash values at a time. Note further that any (c, k)-universal hash family is also (c, l)-universal for any $2 \leq l \leq k$, since summing along marginals of (6.1) produces the desired result.

6.2 Application to our Cuckoo Hashing analysis

Pagh and Rodler [4] use a construction of Siegels [5] to construct a family of hash functions that when restricted to any set of r^2 keys, is $(1, n^\delta)$-universal for some $\delta > 0$ (in particular,

$n^\delta >$ MaxLoop). In our analysis of Case 1 of the insertion loop analysis of Section 5.3.3, the failure condition becomes that h_1 and h_2 are not $(1, \text{MaxLoop})$-universal with probability $O(1/n^2)$. In Case 2, the $(1, \text{MaxLoop})$-universality condition guarantees that (5.2) continues to hold with probability $r^{-(v-1)}$, and similarly, in Case 3, the condition guarantees that each configuration described occurs with probability r^{-2v}.

6.3 Hash functions in practice

Above, refined our analysis to use a family of hash functions that has been demonstrated to be constructible. However, the above family of hash functions is not used in practice as it does not admit fast computation, so in practice, Cuckoo Hashing uses families of hash functions that have weaker properties than those used in our analysis.

In the experimental work of Pagh and Rodler [4], it was found that unlike other hashing schemes, Cuckoo Hashing was very sensitive to the hash family used to generate h_1 and h_2. Pagh and Rodler considered the family from [2] of hash functions parameterized by odd integers a mapping to $\{0,1\}^q$, $h_a(x) = (ax \bmod 2^w) \text{ div } 2^{w-q}$. When a was selected using Cs `rand` function, performance was degraded. The workaround used by Pagh and Rodler was to select h_1 and h_2 as the XOR of three independently chosen functions from the above family. It is unclear why, theoretically, this scheme works, but the resulting hash functions seemed to perform well with Cuckoo hashing.

7 Comparison to previous hash families

Pagh and Rodler [4] performed experiments measuring the runtime of Cuckoo Hashing with optimized implementations of chained hashing, linear probing and two-way chaining. On a clock cycle basis, Cuckoo Hashing is quite competitive. We have reproduced the results of these experiments in Figure 7, where various operations were performed with a load factor of 1/3. Note that for values of n below 2^{15}, runtimes of the various hash table schemes are quite similar, due to the effect of CPU caching. With values of n above that threshold, it is likely that cache miss effects explain the difference in runtime of the various schemes.

References

[1] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180200, 1999.

[2] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):1951, 1997.

[3] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC 01)*, pages 425–432. ACM Press, 2001.

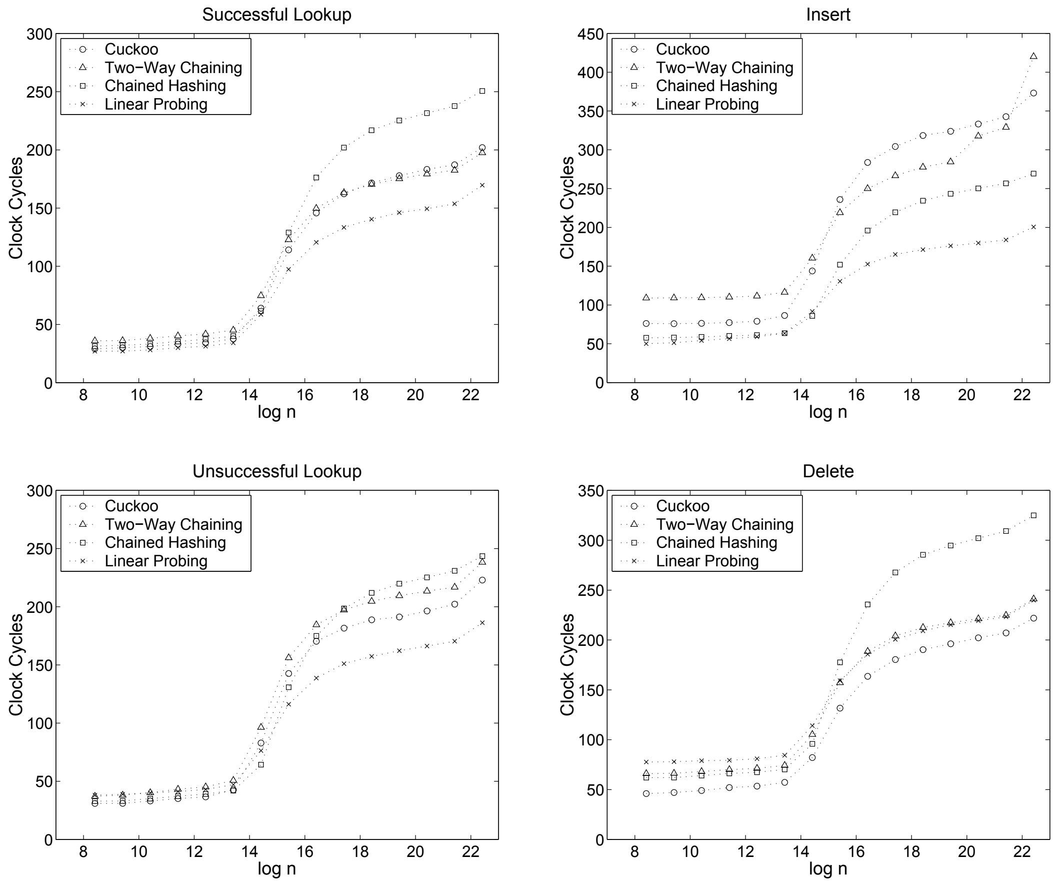

Figure 4: Graphs, reproduced from [4], demonstrating the experimental competitiveness of Cuckoo Hashing compared to two-way chaining, chained hashing and linear probing.

[4] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *Journal of Algorithms*, pages 122–144. Elseiver, 2004.

[5] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. *In Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS 89)*, pages 2025. IEEE Comput. Soc. Press, 1989.