# Cuckoo Hashing and Cuckoo Filters

Noah Fleming

May 17, 2018

## 1 Preliminaries

A dictionary is an abstract data type that stores a collection of elements, located by their key. It supports operations: insert, delete and lookup. Implementations of dictionaries vary by the ways their keys are stored and retrieved. A hash table is a particular implementation of a dictionary that allows for expected constant-time operations. Indeed, in many cases, hash tables turn out to be on average more efficient than other dictionary implementations, such as search trees, both in terms of space and runtime.

A hash table is comprised of a table $T$ whose entries are called *buckets*. To insert a (key, value) pair, whose key is $x$, into the table, a *hash function* is used to select which bucket to store $x$. That is, a hash function is a map $h : \mathcal{U} \to [T]$ on a universe $\mathcal{U}$ mapping keys to locations in $T$. We will be particularly interested in the following strong family of universal hash functions.

**$k$-Universal Hash Functions:** A family $\mathcal{H}$ of hash functions $h : \mathcal{U} \to [T]$ is $k$-universal if for any $k$ distinct elements $x_1, \dots, x_k \in \mathcal{U}$, and any outcome $y_1, \dots, y_k \in [T]$

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1, \dots, h(x_k) = y_k] \leq 1/|T|^k,$$

where the probability is taken as uniform over $h \in \mathcal{H}$.

Typically, the universe $\mathcal{U}$ to which our keys belong is much larger than the size of our table $T$, and so collisions in the hash table are often unavoidable. Therefore, the hash function attempts to minimize the number of collisions because collisions lead to longer insertion time, or even insertions failing. Unfortunately, as the number of elements in a hash table grows, so does the number of collisions. This is measured by the *load factor* of the hash table

**Load Factor:** The *load factor* of a hash table $T$ with $n$ keys is

$$\text{load factor} := \frac{n}{|T|}.$$

## 2 Cuckoo Hashing

Many variants on the standard hash table have been proposed to handle the issue of collisions. Cuckoo Hashing, introduced by Pagh and Rodler [8], is one such variant, which uses two hash functions $h_1, h_2$ chosen at random from $\mathcal{H}$ and a unique insertion scheme. The benefits of Cuckoo Hashing are substantial:

1. worst-case constant-time lookup and delete,
2. expected constant-time insertion,
3. uses table size only slightly larger than the space needed to keep all elements.

Standard hash tables only allow for at most two of these properties to hold simultaneously.

To motivate Cuckoo Hashing, we will recall the lovely exposition of Mitzenmacher [5]. Suppose that we are hashing $n$ keys to a table with $n$ buckets in the online setting, using a hash function picked from some universal family of hash functions. Then, largest number of collisions to any single bucket is expected to be about $O(\log n / \log \log n)$ by the famous balls-in-bins problem. If instead we are given two hash functions, one mapping only to the first $|T|/2$ buckets in the table, and the other mapping only to the bottom $|T|/2$ buckets, and at each insertion, we are allowed to choose which of $T[h_1(x)]$ or $T[h_2(x)]$ an element $x$ is placed into, then expected largest number of collisions to any bucket drops to only $O(\log \log n)$! This was shown by Azar, Broder, Karlin, and Upfal [1].

It is natural to ask how this compares to the optimal distribution. That is, in the offline setting, where you're given all of the keys upfront, and you're asked to minimize the number of collisions, subject to placing each key according to one of its two hash functions. Pagh et al. [6] showed that the minimum number of collisions drops to constant. Furthermore, they showed that if $|T| \geq (2+\delta)n$ for small $\delta > 0$, then the probability that there exists a placement according to the hash functions such that there is no collisions at all becomes $1 - O(1/n)$. This begs the question of whether there exists an efficient algorithm that can achieve the same performance in the online setting. Cuckoo hashing is the dynamization of this static process.

In Cuckoo Hashing we will maintain the following property:

*for every key $x$, if $x$ is in $T$, then $x = T[h_1(x)]$ or $x = T[h_2(x)]$.*

This property will allow us to perform the lookup operation by only looking at two positions in the table:

**Algorithm 1:** Lookup($x$):

```
1 return T[h1(x)] = x  ∨  T[h2(x)] = x;
```

Of course, when inserting a new key $x$ it could be the case that both $h_1(x)$ and $h_2(x)$ are occupied. To remedy this, we perform a special insertion procedure, which is reminiscent of the behaviour of the Cuckoo bird, and is where the algorithm gets its name.

*Cuckoos are known for laying their eggs in the nests of other birds. When the cuckoo egg hatches, the hatchling pushes the other eggs out of the nest and assumes the role of the unsuspecting mother's child.*

If both $h_1(x)$ and $h_2(x)$ are occupied, the insertion algorithm will displace one of the elements, and inserts $x$ into the newly freed bucket. The displaced element will then need to be inserted, and so we must place it into its other location, possibly displacing another element. The procedure repeats until an empty bucket is found. The following is an implementation of this insertion algorithm; we will denote by $\leftrightarrow$ the operation of swapping the contents of two variables.

**Algorithm 2:** Insert($x$):

```
1  if lookUp(x) then return;
2  if T[h2(x)] = ∅ then T[h2(x)] ← x; return;
3  hold ← x;
4  index ← h1(x);
5  for i ← 0 to Max do
6  |   if T[index] = ∅ then T[index] ← hold; return;
7  |   T[index] ↔ hold;
8  |   if index = h1(hold) then
9  |   |   index ← h2(hold);
10 |   else
11 |   |   index ← h1(hold);
12 |   end
13 end
14 rehash(T); insert(x);
```

This creates a sequence of keys $x_1 = x, x_2 \dots x_k$ such that

$$h_1(x_1) = h_1(x_2), h_2(x_2) = h_2(x_3), h_1(x_3) = h_1(x_4), \dots$$

and such that $x_k$ is either placed in an unoccupied bucket, or $k = Max$. Note that the same hash functions occur between the shared locations, that is $h_i(x_j) = h_i(x_{j+1})$, because we enforce that $h_1$ can only map into the first half of $T$, while $h_2$ can only map into the second half.

To observe the behaviour of the insertion strategy, consider the following example in Figure 1a. The table $T$ contains 6 buckets, and is populated by 5 keys. The directed line leading from each of the occupied buckets indicates the other possible location for each key in the table. That is, $b$ could be stored at location 1 or location 6.

Now, consider inserting a key $x$ into the table. Suppose that $h_1(x) = 2$ and $h_2(x) = 1$. Because they are both occupied, the algorithm will place $x$ in location 2, replacing $a$. It will then move $a$ to 5, $k$ to 3 and $z$ to 4. Thus resulting in the table in Figure 1b.

Unfortunately, the sequence of keys displaced during may become complicated, and keys may be revisited multiple times, causing cycles. Furthermore, there may be no way to place all of the keys into the hash table according to the current hash functions, and therefore

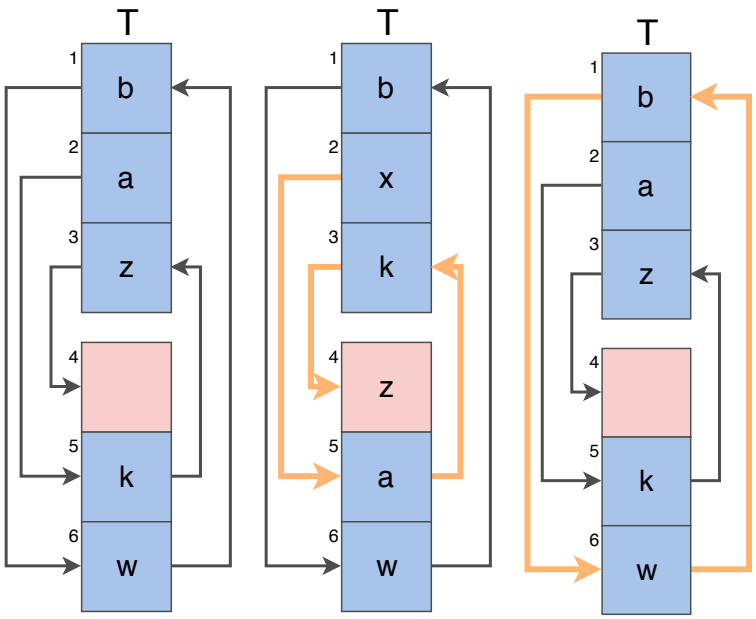



Figure 1: Example of a Cuckoo Hash Table $T$. The head of the directed arrow indicates the alternate location of the element found at the tail. (a) The path of insertion of an element $x$ with $h_1(x) = 2$ and $h_2(x) = 6$. (b) The path of insertion of an element $x$ with $h_1(x) = 1$ and $h_2(x) = 6$.

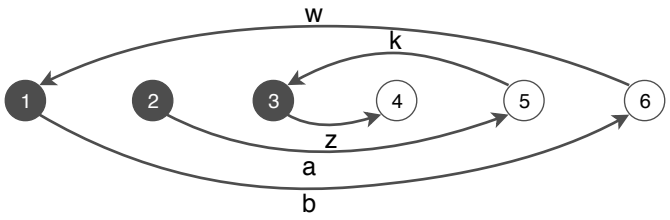



Figure 2: The Cuckoo Graph corresponding to the Cuckoo table in Figure 1a. Black represents that the bucket belongs to the first half of the table, white means that it belongs to the second half.

the table must be rehashed. That is, new hash functions are selected and elements are reinserted. For example, consider inserting $x$ where $h_1(x) = 1$ and $h_2(x) = 6$. This will call the insertion algorithm to loop between the locations

$$1 \to 6 \to 1 \to 6 \to \dots$$

In this case, we say that the sequence of elements, $x, b, w, x, b, w, \dots$ has entered a bad configuration. Because of this, a maximum number of iterations is enforced by the $Max$ variable on the insertion algorithm. Pagh and Rodler [8] show that in order to keep all in operations expected constant-time, $Max$ must be set to $\lceil 6 \log_{1+\delta/2} |T| \rceil$. We will discuss this further in the next section.

The performance deteriorates as the load factor increases. Therefore, an upper bound of $\left(\frac{1}{2+\delta}\right)$ is imposed on the load factor. That is, $|T| \geq (2+\delta)n$. If the table exceeds this load factor, then the size is increased and the table is rehashed.

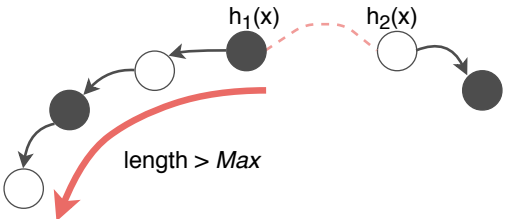
## 2.1 Properties of Cuckoo Hashing

We will now study the properties of Cuckoo Hashing and work our way up to proving that it has expected constant-time operations. For this, it will be useful to associate a directed graph with the elements of a Cuckoo Hash table. The nodes will be buckets in the hash table, while the edges will be labelled by distinct elements in the hash table. If the element $x$ is in bucket $T[h_1(x)]$, then the edge will be directed from $h_1(x)$ to $h_2(x)$. Otherwise, if $x \in T[h_2(x)]$ then the edge will be directed from $h_2(x)$ to $h_1(x)$.

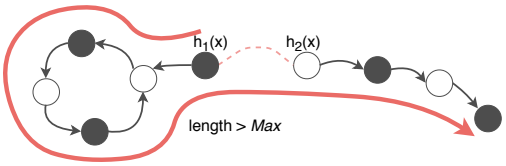
**Cuckoo Graph:** Let $T, h_1, h_2$ be a Cuckoo Hash table, then the associated directed Cuckoo Graph $G = (V, E)$ has $V = [T]$ and $E = \{(i, j) : \exists e \in \mathcal{U}, h_k(e) = i, h_{1-k}(e) = j \text{ for some } k \in \{0, 1\}\}$.

For example, for the previous example, the Cuckoo Graph from the previous example in Figure 1 is given in Figure 2. Observing this Cuckoo graph, we can derive the following properties which will be important later on.

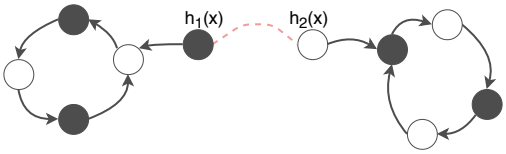
**Proposition 1.** *Any Cuckoo Graph satisfies the following properties:*




(a) Case 3(a) of Proposition1.




(b) Case 3(b) of Proposition1.




(c) Case 3(c) of Proposition1; a bad configuration.

1. *Each vertex has at most 1 outgoing edge.*
2. *Any path can contain at most one directed cycle.*
3. *An insertion of an element $x$ will fail if and only if there is a directed edge leading from $h_2(x)$ and one of the following not mutually exclusive cases holds:*
   (a) *The directed path leading from $h_1(x)$ has length $> Max$ (Figure 3a).*
   (b) *The directed path leading from $h_1(x)$ contains a cycle and the number of buckets required to traverse to return to $x$ plus the length of the directed path leading from $h_2(x)$ exceeds $Max$ (Figure 3b).*
   (c) *both endpoints $h_1(x)$ and $h_2(x)$ have directed paths leading from them containing a cycle (Figure 3c).*

*Proof.* We prove each of the properties as follows:

(1) Each vertex corresponds to a bucket, which can be occupied by at most one element.

(2) This follows immediately from property (1).

(3) 3(a) is clear. For 3(b) and 3(c), observe that traversing a directed path corresponds to displacing each vertex to its alternate bucket, and therefore flipping the direction of that edge. This can be seen in Figure 4. Therefore, when we encounter a cycle, we will traverse around the cycle and then exit it. This means that the only way for the

insertion to become stuck in a loop, revisiting a bucket more than twice, is if both of the outgoing paths from $h_1(x)$ and $h_2(x)$ contain cycles.

For the other direction, briefly if none of the conditions 3(a),3(b) or 3(c) hold, then either there is a path leading from $h_1(x)$ of $k < Max$ edges and $k+1$ vertices with no cycle, and so the final bucket must be free. Otherwise, $h_1(x)$ contains a cycle, and so traversing it will return us to $x$, and we will traverse the path corresponding to $h_2(x)$, which will terminate in an empty bucket by the previous argument.

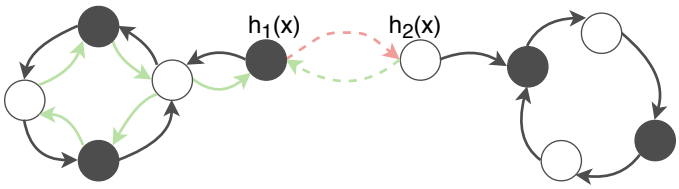



Figure 4: Traversing a bad configuration. The green edges represent the direction of the edges after we have traversed them each once.

□

Recall that Pagh and Rodler set $|T| \geq (2+\delta)n$ for some small constant $\delta > 0$. We will now give some intuition as to why this load factor was imposed. Because our hash function satisfies good independence and uniformity properties, the undirected version of the cuckoo graph should behave similar to a random graph. Random graphs exhibit sharp threshold properties; in particular, it is well-known that the threshold for the appearance of cycles in a $|T|$ node graph occurs at $|T|/2$ edges. Before $|T|/2$ edges, the occurrence of a cycle is unlikely; after $|T|/2$, they occur in abundance. As a cycle (or extremely long path) must exist in order for insertion to fail, this informs our choice of load factor, that it should be approximately $1/2$ in order to avoid this threshold.

We will call the configuration in case 3(c) of Proposition 1 a *bad configuration*. We will say that the sequence of keys $x_1, \ldots, x_k$ generated during the run of the insertion algorithm thus far has encountered a bad configuration if $x_1, \ldots, x_k$ contains at least a single traversal of the two cycles.

We will now turn our attention to showing that insertions are expected constant-time. For this, we will need the following lemma.

**Lemma 1.** *Suppose that the insertion procedure for a key $x$ has not encountered a bad configuration. For any prefix $x = x_1, x_2, \ldots, x_p$ of the keys encountered by the insertion procedure, there exists a consecutive sub-sequence of length at least $p/3$ distinct keys beginning with an occurrence of $x$.*

*Proof.* We will prove this by consider the following exhaustive set of cases.

**Case 1:** If the insertion procedure never returns to a previously visited bucket, then the entire sequence can be taken as the sub-sequence.

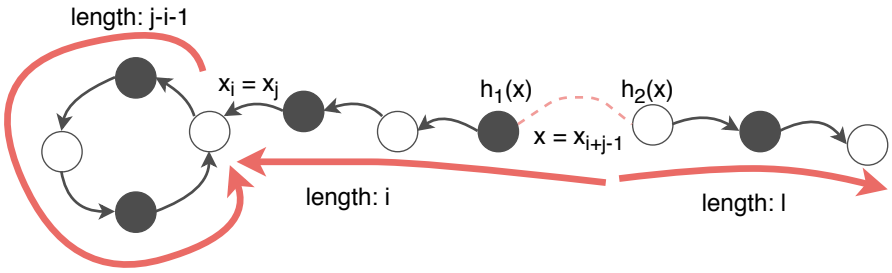



Figure 5: Depiction of the argument for Case (2) of Lemma 1.

**Case 2:** Let $j$ be the minimum value such that there exists $i < j$ with $x_i = x_j$. Thus, $(x_i, x_j)$ will be the key right before the start of the cycle, and the key right after the end of the loop. This can be seen in Figure 5. Furthermore, note that $x = x_{i+j-1}$, and these will be at most two occurrences of $x$ in the sequence. We will argue that either the sequence of distinct keys following the first occurrence of $x$, or the sequence following the second occurrence of $x$ must have length at least $p/3$.

**Case 2a:** If $p < i+j-1$: the first $j-1$ keys $x_1, \ldots, x_{j-1}$ must be distinct because $x_j$ is the first repeated key. Because $p < i + j$, the last element in the sequence must be somewhere before the second occurrence of $x$. The sequence $x_1, \ldots, x_{i+j-1}$ is composed of $j - 1$ distinct elements, followed by $i$ repeated elements, and so $j - 1 \geq (i + j - 1)/2 \geq p/2$, and so we have found such a consecutive sequence.

**Case 2b:** If $p \geq i + j - 1$, then the final element in in the sequence must be after the second occurrence of $x$. I claim that either the sequence $x_1, \ldots, x_{j-1}$ has length $p/3$ or $x_{j-i-1}, \ldots, x_p$ has length at least $p/3$. The sequence can be decomposed as follows: the keys $j - 1$ keys are distinct, followed by $i$ repeated elements, with the last being $x$ at index $i + j - 1$. From then on, we traverse the directed path beginning at $h_2(x)$, on which all keys are distinct; this path has length $p - (i + j - 1)$. Then we have

$$(j - 1) + i + (p - (i + j - 1)) = p,$$

Now, if $p - (i + j - 1) \geq p/3$, then sequence $x_{i+j-1}, \ldots, x_p$ satisfies the properties of the Lemma. Otherwise, if $p - (i + j - 1) < p/3$, then

$$(j - 1) + i > 2p/3,$$

and because $j - 1 \geq i$,

$$j - 1 \geq p/3.$$

□

**Theorem 1.** *Cuckoo Hashing has expected constant-time insertion over random choices for* $h_1, h_2 \in \mathcal{H}$.

*Proof.* Let $n$ be the number of elements in the hash table, and recall that we are maintaining that $n(2+\delta) \leq |T|$, where $|T|$ is the number of buckets in the hash table $T$, and $\delta > 0$ is some small constant, as well, suppose that $h_1, h_2$ are $Max$-universal hash functions. Recall as well that $h_1$ maps into the top $|T|/2$ buckets of $T$ and $h_2$ maps only into the bottom $|T|/2$ buckets.

We will bound the probability that the loop runs for at least $k$ iterations. Consider the following two cases for the insertion algorithm after $k$ steps:

**Case 1:** The sequence of $k$ keys generated has not yet entered a bad configuration.
By Lemma 1, there is a distinct consecutive sub-sequence $b_1 = x, b_2 \ldots, b_{k/3}$ of the keys encountered. Since, by the insertion procedure, neighbouring $b_i$ in the sequence share one of their locations. As well, because we have not encountered a bad configuration, there can be at most two occurrences of $x$ in the sequence of keys thus far. Therefore, we have that if $b_1$ is the first occurrence of $x$, then

$$h_1(b_1) = h_1(b_2), h_2(b_2) = h_2(b_3), h_1(b_3) = h_1(b_4), \ldots \tag{1}$$

or if $b_1$ is the second occurrence of $x$, then

$$h_2(b_1) = h_2(b_2), h_1(b_2) = h_1(b_3), h_2(b_3) = h_2(b_4). \ldots \tag{2}$$

This follows because we only allow $h_1$ to index into the first half of the table and $h_2$ to index into the second half of the table. Now, we want to bound the probability that such a sequence of keys satisfying the sequence (1) is encountered. There are at most $n$ keys in the hash table, and because $b_1 = x$ is fixed, there are at most

$$1 \cdot n(n-1)\ldots(n-k/3+2) \leq n^{k/3-1}$$

choices for the keys $b_1, \ldots, b_{k/3}$. Now, by our assumption that $h_1$ and $h_2$ are sampled from a $Max$-universal family of hash functions, and by marginalizing, are therefore $k/3$-universal, the probability that $h_1$ and $h_2$ map $b_1, \ldots, b_{k/3}$ such that sequence (1) holds is

$$\begin{aligned}
&\Pr[h_1(b_1) = h_1(b_2) \wedge h_2(b_2) = h_2(b_3) \wedge \ldots] \\
&= \Bigg( \sum_{\substack{y_1, y_3, y_5, \ldots \in [1, |T|/2] \\ y_2, y_4, y_6, \ldots \in [|T|/2+1, |T|}} \Pr[h_1(b_1) = y_1 = h_1(b_2), h_1(b_3) = y_3 = h_1(b_4), \ldots] \\
&\quad \cdot \Pr[h_2(b_2) = y_2 = h_2(b_3), h_2(b_4) = y_4 = h_2(b_5), \ldots] \Bigg) \\
&\leq \frac{(|T|/2)^{k/3-1}}{(|T|/2)^{(k/3-1)}(|T|/2)^{(k/3-1)}}, \\
&= \frac{1}{(|T|/2)^{k/3-1}}.
\end{aligned}$$

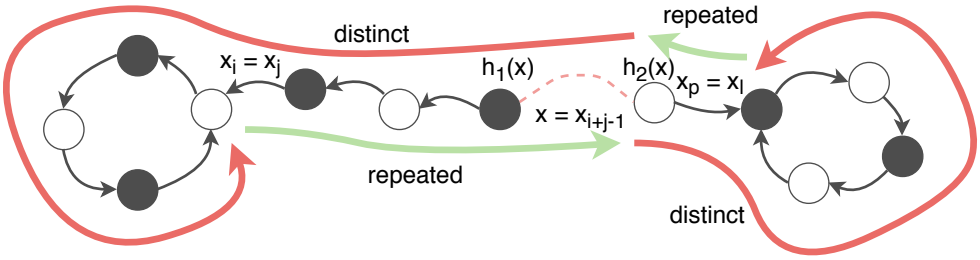



Figure 6: Depiction of the argument for Case (2) of Theorem 1.

Similarly, probability that the sequence 2 holds is the same. Therefore, the probability of the occurrence of a sequence of $k$ keys that has not entered a bad configuration is bounded by

$$\frac{2n^{k/3-1}}{(|T|/2)^{k/3-1}} < \frac{2}{(1+\delta/2)^{(k/3)-1}} \tag{3}$$

Here, we are using the fact that $|T| \geq (2+\delta)n$. Note that this is a very loose bound.

**Case 2:** The sequence of $k$ keys generated encountered a bad configuration.
Let $v$ be the number of distinct keys encountered in the sequence of $k$ keys generated by the insertion algorithm so far. We will bound the probability of entering a bad configuration after $k$ steps, having encountered $v$ distinct keys. The argument for this case can be seen in Figure 6.

Because we have entered a bad configuration, both $h_1(x)$ and $h_2(x)$ must have directed paths leading from them, each containing a cycle. We will pick witnesses for both of these cycles. Let $j$ be the first index such that there exists an $i < j$ and $x_i = x_j$. That is, $x_i$ is the key encountered before traversing the first loop and $x_j$ is the first key after the first loop. Note as well that $x_{i+j-1} = x$. Similarly, let $\ell \geq i+j$ be the first index such that there exists an index $p$ such that $x_\ell = x_p$. That is, $x_p$ is the key encountered before traversing the second loop, and $x_p$ is the first key encountered after the second loop.

I claim that the number of ways to choose $i, j, p, \ell$ is the number of ways to partition $v$ items into 4 groups. To see this, observe that $i, j, p, \ell$ uniquely determines the bad configuration. This can be seen in Figure 6 as follows: on our first traversal through the bad component, the elements $x_1, \ldots, x_{j-1}$ are all distinct, then $i$ elements repeat, following this, $x_{i+j}, \ldots, x_\ell$ are all distinct, and then $p$ repeat, completing a traversal of the bad component. Each following traversal will be identical. Therefore, this is equivalent to partitioning a sequence of $v$ items into 4 groups and so there are at most $\binom{v}{3} \leq v^3$ choices altogether for $i, j, p$ and $\ell$.

Now, there are at most $n^{v-1}$ ways to label the $v$ edges such that the first one is labelled $x$, and $|T|^{v-1}$ ways to label the vertices excluding $x$ (because $x$ is the element being inserted, $h_1(x)$ and $h_2(x)$ are fixed), since we require that the vertices alternate between coming from the first $|T|/2$ buckets of $T$ and the second $|T|/2$ buckets. Since $h_1$ and $h_2$ are $Max$-universal,

the probability that $h_1$ is consistent with the labelled graph is at most $(|T|/2)^{-v}$, and similarly for $h_2$. Since $h_1$ and $h_2$ are chosen independently, the probability that both $h_1$ and $h_2$ are consistent with the labelled graph is at most $(|T|/2)^{-2v}$. Altogether, the probability of case 2 occurring is bounded by summing over all possible values for $v$ and all choices for the labelled graph,

$$\sum_{v=3}^{k} n^{v-1}(|T|/2)^{v-1}v^3(|T|/2)^{-2v} \leq \frac{2}{|T|n}\sum_{v=3}^{\infty} v^3\left(\frac{2n}{|T|}\right)^v < \frac{2}{|T|n}\sum_{v=3}^{\infty} v^3\left(1+\delta/2\right)^{-v} = O(1/n^2).$$

**Putting it all together:** Combining cases 1 and 2 we have that expected number of iterations in the insertion loop is bounded by

$$\begin{aligned}
1+\sum_{k=2}^{Max}\left(\frac{2}{(1+\delta/2)^{(k/3)-1}}+O(1/n^2)\right) &\leq 1+O\left(\frac{Max}{n^2}\right)+2\sum_{k=0}^{\infty}(1+\delta/2)^{-k/3}, \\
&= O\left(1+\frac{1}{1-(1+\delta/2)^{-1/3}}\right), \\
&= O(1).
\end{aligned}$$

**Rehashing:** By case 2, the probability of encountering a bad configuration and, therefore, rehashing is $O(1/n^2)$. Otherwise, for an insertion to fail without entering a bad configuration, it must run for $k = Max$ iterations. By case 1,the probability that it survives $Max$ iterations without inserting an element or entering a bad configuration is at most

$$2(1+\delta/2)^{-Max/3+1}.$$

Setting $Max = \lceil 6\log_{1+\delta/2}|T|\rceil$, we have

$$2(1+\delta/2)^{-\lceil 6\log_{1+\delta/2}|T|\rceil/3+1} = O(1/n^2).$$

Therefore, the probability of a rehash is $O(1/n^2)$.

During a rehash, $n$ insertions are performed, each with a failure probability of $O(1/n^2)$. Therefore, all insertions succeed with probability at least $1-O(1/n)$. Furthermore, each insert takes expected $O(1)$ time, and therefore, the expected cost of rehashing is $O(1/n)$. □

## 2.2 Buckets and Hash Functions Allow For Higher Loads

Since its inception, Cuckoo Hashing has been extended in several ways that improves its performance. One obvious shortcoming of Cuckoo Hashing is its load factor of less than 1/2. Follow up research by Erlingsson, Manasse and McSherry [4] showed that this can easily be improved by either increasing the number of hash functions, or allowing multiple items to be stored in a single bucket, or a combination of the two. Indeed even increasing the number of hash functions used from two to three empirically increases the achievable load factor to 91%. Note that this no longer suffers from the cycle threshold for random graphs. Similarly, allowing the buckets to contain 2 keys instead of 1 increases the load factor to 86%. Combinations of the two increase it further, with the maximum of 99.9% achieved by 4 hash functions and 4 keys per bucket, or 8 keys per bucket and 3 hash functions.

# 3 Cuckoo Filters

We will now discuss an application of Cuckoo Hashing used in fast membership testing. So suppose that you have a large set of elements that changes over time, and you want to be able to do very fast membership testing on this set. That is, determine whether a particular element is in this set. One could achieve this by hashing each of the elements to a hash table, but to avoid collisions, the table must be quite large obtain good accuracy, the size of the keys stored grows with the size of the set. If the set is quite large, this wasted space become problematic.

One way of avoiding these problems is by storing only a single bit indicating whether the item is in the set or not, rather than storing the entire element in the hash table. To check whether an element is in the table, one hashes to the location of that element, and checks whether the bit is 0 or 1. Of course, this yields error, but on the upside, it is only one sided error: if the the bit is 0, then we know for sure that the element is not in the set, but, due to possible collisions in the hash function, if the bit is 1, we only know that the element is *maybe* in the set. To mitigate this error, and to allow for a better load factor in the table, $k$ hash functions can used instead of one. This structure is known as a *Bloom Filter* and supports the following operations:

1. **Insert:** To insert an element, the hash table hashes to all $k$ locations, $h_1(x), \ldots, h_k(x)$, and sets $T[h_1(x)] = 1, \ldots, T[h_k(x)] = 1$,

2. **Lookup:** To lookup an element, the table hashes to all $k$ locations for $x$, and returns True if $T[h_1(x)] = 1, \ldots T[h_k(x)] = 1$.

Furthermore, the false-positive rate $\varepsilon$ of a Bloom filter can be approximately controlled by the number of hash functions $k$. In particular, to achieve a false-positive rate of $\varepsilon$, $k$ should be set to

$$k = \log(1/\varepsilon).$$

Unfortunately, while Bloom filters can add elements, they cannot delete them. Suppose that that two elements $x$ and $y$ shared one of their hash locations, that is $h_1(x) = h_2(y)$. Then, to delete $x$ from the table, we would have to set all of $x$'s hash locations to 0, and in particular, $h_2(y) = 0$, causing a false-negative on the membership of $y$. Several modifications of Bloom filters have been proposed to add deletion, but they either require extra space or increase running time.

The introduction of the Cuckoo Filter shows that one does not need to sacrifice space or performance to support deletion. The authors show that (in practice) the Cuckoo Filter is able to perform the same set of operations as a Bloom Filter, while supporting deletion and using less space.

A *Cuckoo Filter* is a combination of Cuckoo Hashing and a Bloom filter. Rather than storing an entire element, each cell in the hash table will now store only a short identifying string of the element, known as a *fingerprint*.

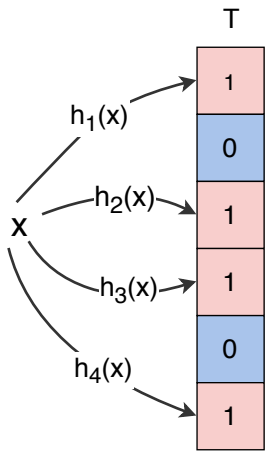



Figure 7: A bloom filter mapping an element $x$ according to the $k$ hash functions $h_1, \ldots, h_k$.

**Fingerprint:** A fingerprint as a hash function $\phi : \mathcal{U} \to \{0,1\}^f$ which maps each element from the universe to its *fingerprint*, a short identifying string on $f$ bits, such that if $x = y$ then $\phi(x) = \phi(y)$ and if $x \neq y$, then $\phi(x) \neq \phi(y)$ with high probability.

Instead of storing the entire element $x$ in the Cuckoo hash table, we will only store $\phi(x)$. When inserting an element into the table, it may sometimes need to move previously inserted elements to their alternate location. However, we don't have access to the original element, only the fingerprint, and so we cannot determine alternate location. We only have the fingerprint, and the hash of the fingerprint. Therefore, the two locations of each element can no longer be chosen independently. For an element $x$, and two hash functions $h_1, h_2$, we define its two locations to be

$$\begin{aligned} &\text{location 1: } h_1(x), \\ &\text{location 2: } h_1(x) \oplus h_2(\phi(x)), \end{aligned}$$

where recall that $\phi(x)$ is the fingerprint of $x$. Observe now that if we find $\phi(x)$ at location $i$ in the table, we can recover the other location of $x$ by computing

$$i \oplus h_2(\phi(x)) = \begin{cases} h_1(x) & \text{if } i = h_1(x) \oplus h_2(\phi(x)), \\ h_1(x) \oplus h_2(\phi(x)) & \text{if } i = h_1(x). \end{cases}$$

A natural question is why bother to use $h_2$, instead of just using $h_1(x) \oplus \phi(x)$ as the other location. One of our objectives is to minimize the number of bits stored, and therefore we should think of $\phi(x)$ as a very short bit string, significantly shorter than $h_2(\phi(x))$. Because of this, if we only used $h_1(x) \oplus \phi(x)$, then the items kicked out of nearby buckets would land near to each-other. For example, if we used 8-bit fingerprints, then the two locations of each item would be at most 256 buckets away from each other.

Unfortunately, the lack of independence between the two locations has several drawbacks. First, if a pair of elements $x$ and $y$ map to the same bucket and share the same fingerprint,

then they also have the same secondary location. Therefore, inserting both $x$ and $y$ would cause the insertion algorithm to fail. Because of this, the buckets of the Cuckoo Filter table are generally allowed to store a multi-set of fingerprints, controlled by a bucket-size parameter $b$. Even so, this implies that there can be at most $2b$ elements that share the same hash location and fingerprint, before the algorithm fails. The second issue is that the theoretical results that hold for Cuckoo Hashing do not apply directly to Cuckoo Filters.

We now describe the operations of a Cuckoo Filters on a table $T$. Lookup is immediate.

**Algorithm 3:** Insert($x$):

```
1  f ← φ(x) ;
2  if T[h1(x)] or T[h1(x) ⊕ h2(f)] contains an empty entry then
3  |   add φ(x) to that bucket ;
4  |   return;
5  i ← h1(x) ;
6  for i ← 0 to Max do
7  |   select a fingerprint e at random from T[i] ;
8  |   swap f with e ;
9  |   i = i ⊕ h2(f) ;
10 |   if T[i] contains an empty entry e then
11 |   |   e ← f ;
12 |   |   return;
13 end
14 Rehash; Insert(x);
```

To delete an element $x$, we check whether either of the locations for $x$ contain the fingerprint $\phi(x)$, and if so we remove it. If there are multiple copies of $\phi(x)$, we remove only one; as they are identical, it does not matter which one is removed.

## 3.1 On the Issue of Space

In experimental results, the authors show that with a false positive rate $\varepsilon < 3\%$ and a bucket size of $b = 4$, Cuckoo Filters can achieve a the same runtime performance, while being more space efficient than standard space-optimized Bloom filters.

Unfortunately, this is not true asymptotically. It is not difficult to show (under some assumptions on the hash function) that $f$, the length of the fingerprints must grow logarithmically with the number of elements to avoid a high likelihood of failure. They prove this with the assumption of uniformly random hash functions, that is, hash functions where each element assigned probability uniform over the range.

**Lemma 2.** *$f = \Omega(\log n/b)$ for uniformly random hash functions.*

*Proof.* Let $x$ be an element and suppose that it has locations $i$ and $i \oplus (\phi(x))$. The probability of $q-1$ elements sharing either of the two buckets is at least $(2/|T|)^{q-1}$. Note that if an element has the same fingerprint as $x$ and shares one of $x$'s buckets, then it must also share

the other. Therefore, the probability of $q$ items sharing the same two buckets is

$$(2/|T| \cdot 1/2^f)^{q-1}.$$

Recall that a Cuckoo Filter fails when $q = 2b+1$ elements are inserted into the same bucket. Consider inserting $n$ random elements into a table $T$, where $|T| = cn$ for some constant $c > 0$. The expected number of groups of $2b + 1$ elements mapping to the same location is lower-bounded by

$$\binom{n}{2b+1}\left(\frac{2}{2^f \cdot m}\right)^{2b} = \Omega\left(\frac{n}{4^{bf}}\right).$$

□

Recall that Bloom Filters required only about $\log(1/\varepsilon)$ space per element. Therefore, for $\varepsilon$ asymptotically less than $n$, Cuckoo Filters will require significantly more space. This should be juxtaposed with the experimental results, which show that this failure rate is negligible for fairly large set sizes.

In a followup paper, Eppstein [3] shows that this lower bound is tight by giving an upper bound of $f = O(\log n/b)$ under the assumption of uniform hash functions. To do this, they analyze a simplified version of Cuckoo Filters, which uses removes the second hash function $h_2$, that is, the two locations for each element are

$$\begin{aligned}&\text{location 1: } h_1(x),\\ &\text{location 2: } h_1(x) \oplus \phi(x),\end{aligned}$$

This is a significant limitation as we mentioned previously, because it drastically limits the range in which the two locations can exist relative to one-another. As well as the aforementioned result, they show that

# 4 Open Problems

The two main problems of interest left open by these works are the following:

1. Extend the analysis of Cuckoo Hashing to allow the hash functions $h_1$ and $h_2$ to both map to the entire table $T$, rather than restrict them to being non-overlapping. This is the form of Cuckoo Hashing used in practice.

2. Extend the theoretical results about Cuckoo Hashing to Cuckoo Filters.

# Acknowledgement

The author would like to acknowledge the informative expositions of Charles Chen [2] and Ramus Pagh [7].

# References


[1] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, September 1999.

[2] Charles Chen. An overview of cuckoo hashing.

[3] David Eppstein. Cuckoo filter: Simplification and analysis. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, pages 8:1–8:12, 2016.

[4] lfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, January 2006.

[5] Michael Mitzenmacher. Cuckoo hashing, theory and practice, 2007.

[6] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In *STOC*, 2001.

[7] Rasmus Pagh. Cuckoo hashing for undergraduates. 2006.

[8] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.