Cuckoo Hashing [1]

Seminar Advanced Algorithms and Data Structures

Lucas Brunner

October 2018

1 Introduction

This is a report on the paper on Cuckoo Hashing by Rasmus Path and Flemming Friche Rodler. Cuckoo Hashing is a dictionary that stands out from other hashing techniques because of its ease of implementation, constant lookup time and expected constant time for insertions. I wrote it as part in a seminar on Advanced Algorithms and Data Structures. Before presenting cuckoo hashing we will go through some definitions and present some other hashing techniques.

2 Definitions

2.1 Dictionary

The method that we are going to discuss today is a dictionary. A dictionary is a data structure denoted as set S of elements, which we identify by keys. The common functions of a dictionary are insertion, deletion and lookups. Let's define with $n := |S|$ the number of elements in our dictionary. We use hash function in a way such that the keys we use, as well as the results, are smaller than the word length of the machine we are using. Because we are only interested in the workings and performance of a dictionary we will omit data associated with the key. This makes almost no difference, because one could easily store a pointer to some data along with the key.

2.2 Universal Family of Hash Functions

Definition 1. A family of $\{h_i\}_{i \in I}$, $h_i : U \to R$, us (c,k)-universal if, for any k distinct elements $x_i, ..., x_k \in U$, any $y_i, ..., y_k \in R$, and uniformly random $i \in I, Pr[h_i(x_i) = y_1, ..., h_i(x_k) = y_k] \leq c/|R|^k$.

2.3 Others

We use a special value $\perp \in U$ to denote an empty cell and for double hashing we use an additional marker for deleted keys. In the implementation we use a 0 instead of $\perp$. These deleted marker are used because the handling of deletion might be a problem.

3 Previous Methods

When we have a hash table, the question is on how one handles collisions. Meaning that the hash function in use, maps two or more values x_i to the same value y_i e.g. $h_i(x_i) = h_i(x_j)$ $i \neq j$. We are going to look at some methods that existed before cuckoo hashing was released. For the sake of simplicity we use simple hash functions in our examples.

3.1 Chained hashing [4]

In chained hashing one determines a bucked based on the hash function and then uses a list to save all the elements ending up in this bucket. How one exactly handles that is implementation specific. The paper uses an implementation where the first element of the list is saved in the hash table. This is beneficiary because one can often saves the time for a cache miss. Cache misses are one of the disadvantages of this method. As one can see in Experiments the time a certain method uses is highly dependent on the amount of cache misses it has. [1, Sect 4.1]

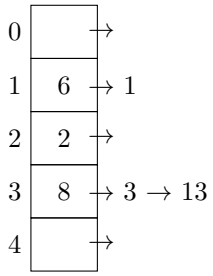

Figure 1: $h(x) = x \bmod 5$

3.2 Linear probing [5]

In linear probing a collision is resolved by putting the conflicting key in the next free cell in the hash table. We use deleted markers, to show, that a cell has been deleted. One of the advantages of linear probing is, that one would not need that many cache misses, because elements might be close by. Of course when having a populated table it might happen, that one has to search for a long time. That is because one has to go to the next free spot, to be certain, tat a value is not in the table. This phenomena is referred to as clustering. In Fig. 2 and Fig. 3 you can find an example of this method.

3.3 Double hashing [3]

As seen with linear probing clustering is a big issue (likely to have clusters of logarithmic length). Double hashing tries to eliminate that trough a more elaborate scheme to handle collisions. For double hashing one uses two random, uniformly and independent hash functions as the name might suggest. The position in the hash table is determined by
$h(i,k) = (h_1(k) + i \cdot h_2(k)) \bmod$ (table-size), where i corresponds to the sequence we check in case of a collision and k is our key. This means, we start with i=0 and only if that bucket is occupied with a value we were not searching for, we compute h(i,k) for i=1....
The implementation in the paper rehashes all keys, when more than 2/3 of the hash table are occupied with $\perp$ (deleted markers) or normal elements.[1, Sect 4.1]

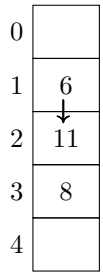

Figure 2: Nr. 11 goes in bucket 3 because bucket 2 is full.
$h(x) = x \bmod 5$

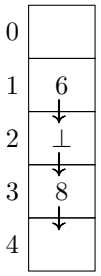

Figure 3: Searching for Nr. 11 tests up to bucket 4.
$h(x) = x \bmod 5$

3.4 Two-way chaining/ 2-choice hashing

2-choice hashing uses two hash functions. Instead of having space for one element per bucket we now have arbitrary many. In our implementation we have four.[1, Sect 4.1] With our two hash functions we compute in which bucket we would put an element. We now look how many elements are in each bucket and put the new element in the one containing less. In a tie we can take either one. As shown in [6] the number of elements in the bucket with the most balls is with high probability lnln$n$/ ln2+O(1), compared to the case in chained hashing of (1+o(1))ln$n$ /lnln$n$. In practice one can assume that four elements per bucket are enough. When searching for an element we of course have to look in both buckets.[1]

4 Cuckoo Hashing

Cuckoo hashing has two hash tables (T_1 and T_2) of length r and two hash functions h_1, h_2: $U \rightarrow 0, ..., r-1$. Every key $x \in S$ is stored in position $h_1(x)$ of T_1 or $h_2(x)$ of T_2 but never in both. Next we are going to show you the tree standard function of a dictionary, lookup, insert and delete. In the following we pick hash functions from a $(O(1), O(log(n)))$ family of hash functions.

4.1 Functions

4.1.1 lookup

As one might image the lookup function is quite simple. There are only two places a value is allowed to be stored. We just check both of them.

Algorithm 1: lookup function

Result: True if key is in dictionary
Input: key x
begin
  return $T_1[h_1(x)] == x \vee T_2[h_2(x)] == x$

4.1.2 delete

deletion is straight forward as well.

Algorithm 2: delete function

Result: deletes key from dictionary
Input: key x
begin
  if $T_1[h_1(x)] == x$ **then**
    $T_1[h_1(x)] = \perp$
  if $rT_2[h_2(x)] == x$ **then**
    $T_2[h_2(x)] = \perp$

4.1.3 insert

As the name of the paper hints the method for inserting elements reminds of the cuckoo bird. When inserting an element to an already occupied location we place it in that location, kicking out the old element. This now nestless element goes into the other table, eventually kicking out the there previously stored element and so forth. As you might see, this could lead to a never ending loop. The probability that happening is bounded to O(1/n). This bound holds for $r \geq (1 + \epsilon)n$ for some constant $\epsilon > 0$ and r being the length of each table. This means that the bound holds for tables that are less than half full.

Algorithm 3: insert function

Result: inserts key into dictionary
Input: key x
begin
  if *lookup(x)* **then**
    return
  for *MaxLoop times* **do**
    if $T_1[h_1(x)] == \perp$ **then**
      $T_1[h_1(x)] = x$; **return**
    swap(x, $T_1[h_1(x)]$);
    if $rT_2[h_2(x)] == \perp$ **then**
      $T_2[h_2(x)] = x$; **return**
    swap(x, $T_2[h_2(x)]$);
  rehash(); insert(x);

As one sees in the code after MaxLoop unsuccessfully attempts we rehash the dictionary with two new hash functions. Because it is assumed, that the tables are less than half full, one also has to do a rehash into larger tables if there are to many elements in the table.

4.2 Example

In the Figures 4 to 8 you can see the state of the two hash tables after the corresponding insertions. As hash function we take $h_1(x) = x \bmod 5$ and $h_2(x) = \lfloor x/5 \rfloor \bmod 5$. In figure 8 we get the problem of a loop, such that we end up in the same configuration as we started. Later you will see how unlikely this case actually is.

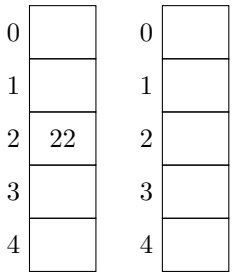

Figure 4: insert(22)

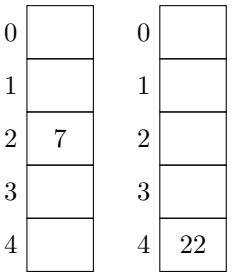

Figure 5: insert(7)

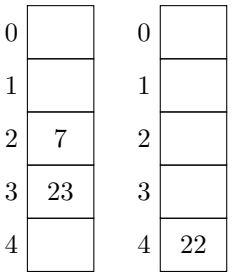

Figure 6: insert(23)

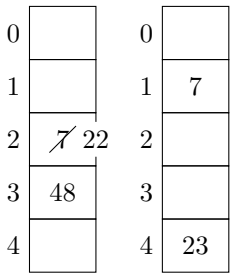

Figure 7: insert(22)

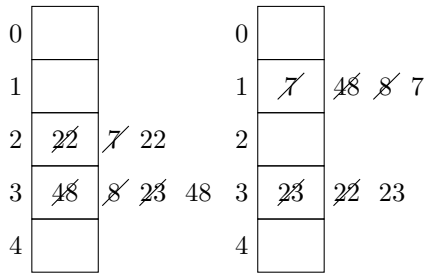

table 1	table 2
$8 \to 48$	$48 \to 7$
$7 \to 22$	$22 \to 23$
$23 \to 8$	$8 \to 48$
$48 \to 23$	$23 \to 22$
$22 \to 7$	$7 \to 8$

Figure 8: insert(8)

4.3 Analysis [1, Sect. 3.1]

Before starting the analysis we should specify, that when we talk about a "nestless" key, we mean a key that is not stored in one of the tables(currently has no nest). Therefore the inserted key and all keys kicked out because of an insertion count as nestless. Note that at any point in time there can be at most one nestless key. In (1) you see the sequence of nestless key for Fig. 8.

$$8, 48, 7, 22, 23, 8, 48, 23, 22, 7, 8, 48, 7, 22, 23, 8, 48, ... \tag{1}$$

To follow the proof it is helpful, to think about how an execution of insert can look like. In Fig. 9 you can find a possible beginning of an insertion. Every arrow in the figure represents the time a key is nestless. It starts at the spot in one table where the key got kicked out and points toward the position in the other table, where it is insert the next, possibly resulting in another arrow. Of course the usual case would be elements switching left and right and not strictly in order as in the illustration. It is drawn in this way so one can see the important parts of an infinite loop. There is always a sequence from a_1 to a_j and a loop that ends with a_j bouncing back resulting in the reversed path we took before. Finally our inserted element a_1 is nestless again and we try inserting it into

the other table. There we get a similar path as the path from a_1 to a_{j+i-1}, with the difference that the tables are switched. Notice, that for an infinite loop the key a_1 would have to be pushed back to the first table.

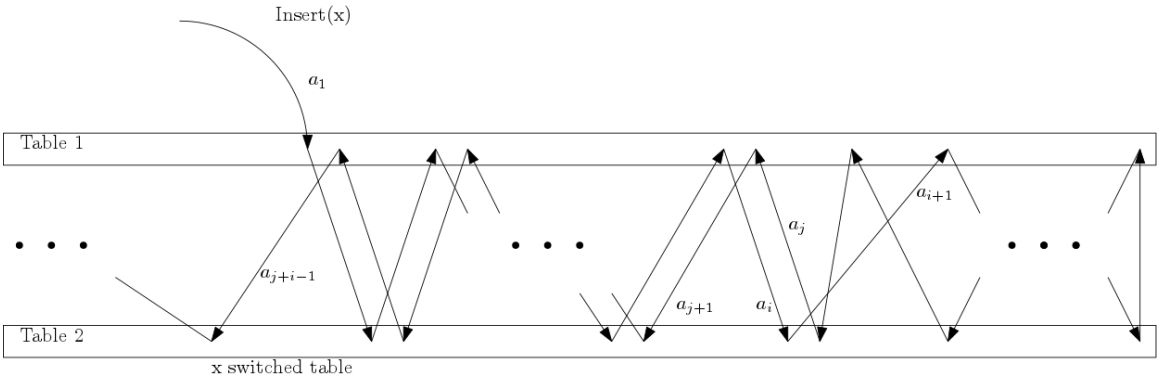

Figure 9: illustration of an insertion

Lemma 4.1. *If the procedure loops for $MaxLoop = \infty$ times, it is not possible to accommodate all the keys of the new set using the present hash functions.*

Proof. Consider the sequence sequence $a_1, a_2, ...$ of nestless keys. For $i, j \geq 1$ we define $A_{i,j} = \{a_i, ..., a_j\}$. Let j be the smallest index such that. $a_j \in A_{1,j-1}$. The key a_j is the first key repeating itself. One might notice that the changes in the table for $1 \leq k < j$ are that a_k is afterwards in the position in which a_{k+1} previously was. That is because we replaced a_{k+1}'s position with a_k. Let $i < j$ be the index such that $a_i = a_j$. This means, that the first occurrence of the key a_j in the nestless sequence was as a_i. One should add, that a_i very well may be the key to be inserted. You can verify that that would be the case in Fig 8. In Fig. 9 we show the case, when $a_i \neq a_1$. Let's consider what happens when $a_i = a_j$ is nestles for the second time. If $i > 1$ then a_j reclaims its previous position occupied by a_{i-1}. If $i > 2$ then a_{i-1} subsequently reclaims its previous position, occupied by a_{i-2} and so on. Therefore we follow, that $a_{j+z} = a_{i-z}$ for $z = 0, ..., i-1$ and as we can verify, we get a_1 again as a_{i+j-1}.
Let's define the number of table cells available to $A_{1,k}$ as $s_k = |h_1[A_{1,k}]| + |h_2[A_1, k]|$. By available cells we mean cells that could potentially be populated by keys from $A_{1,k}$. We therefore do not count other cells/ free spots in the hash table, as we do not have a key 'v' yet for which, $h_1(v)$ or $h_2(v)$, would result in in such a cell. As you can easily see, $A_{1,k}$ has possibly one key more than $A_{1,k-1}$ (a_k can also be a repetition, then they have equally many). We can write this fact as $s_k \leq s_{k-1} + 1$. Additionally it holds, that $s_{j-1} = s_{j-2} \overset{(4)}{\leq} j-1$, because the key a_j appeared earlier in the sequence. Not only a_j, but all the keys $a_j, ..., a_{j+i-1}$ appeared earlier and we get $s_{j+i-2} \overset{(3)}{=} s_{j-2}$. Let j' being the minimum index s.t. $j' > j$ and $a_{j'} \in A_{1,j'-1}$. As before we get $s_{j'-1} \overset{(1)}{=} s_{j'-2}$.

$$\Rightarrow |A_{1,j'-1}| = j' - i \wedge s_{j'-1} \overset{(1)}{=} s_{j'-2} \leq s_{j+i-2} + (j'-2) - (j+i-2) \overset{(3)}{=} s_{j-2} + j' - j - i \overset{(4)}{<} j' - i \quad (2)$$

Therefore we know, we can't accommodate $A_{i,j'-1}$ with the current pair of hash function. □

Theorem 4.2. *The procedure loops without limit with probability* $O(1/n)$.
With Lemma 1 and the work in [2] one can prove it, but this is not in the scope of this report. As an intuition one could say, that because we pick a hash function at random from an $(O(1), O(log(n)))$ family of hash function, that the probability of creating a new circle in the graph is lower than O(1/n). Note that as long there is only one circle we can accommodate all the keys.

Lemma 4.3. *In the insertion of an element v with the prefix* $a_1, a_2, ..., a_l$ *of nestless keys, there exists a subsequence starting from v, s.t. its length is at least* $l/3$ *and it has no repetitions as well.*

Proof. As before we have i and j, $i \leq j$ and j minimal s.t. $a_i = a_j$. As earlier we have $a_{j+z} = a_{i-z}$ for $z = 0, ..., i-1$. There can be no index j' with $j' > j + i - 1$ such that $a_{j'} \in A_{1,j'-1}$ because in Lemma 1 we showed, that in this case not all keys can be accommodated. As $a_i = a_{j+i-1}$ and $i < j$ we have the two sequences $a_1, ..., a_{j-1}$ and $a_{j+i-1}, ..., a_l$ with no repetitions. Note that those two sequences have a gap in between and are not consecutive. Therefore one of them needs to be larger than $l/3$. □

Lemma 4.4. Cuckoo Hashing takes expected $O(1)$ time for an insertion.

Proof. Suppose that the insertion loop runs for at least t iterations. By Lemma 4.3 there is a sequence of distinct keys $b_1, ..., b_m$, $m \geq (2t-1)/3$, such that b_1 is the key to be inserted and such that for $\beta \in 0, 1$

$$h_{2-\beta}(b_1) = h_{2-\beta}(b2), h_{1+\beta}(b_2) = h_{1+\beta}(b3), h_{2-\beta}(b_3) = h_{2-\beta}(b4), ... \tag{3}$$

This means that b_1 in the sequence lays either in the first or the second table. There are at most n^{m-1} sequences with pairwise distinct keys, starting with b_1 possible. When we use a (c,m)-universal family of hash functions, we know that $Pr[(3) holds] \leq cr^{-(m-1)}$ and therefore...

$$Pr[\text{any sequence } b_1, .., b_m \text{ satisfying (3)}] \leq 2 \cdot cr^{-(m-1)} \cdot n^{m-1} = 2c(n/r)^{m-1} \leq 2c(1+\epsilon)^{-(2t-1)/3+1} \tag{4}$$

As an Ansatz let us use a (c,$log_{1+\epsilon}n$)-universal family for some constant c. The probability of having more than $3log_{1+\epsilon}n$ iteration is $O(1/n^2)$. With a small probability for a rehash, we can set $MaxLoop = 3log_{1+\epsilon}n$ and get an expected number of iterations of

$$1 + \sum_{t=2}^{\infty} 2c(1+\epsilon)^{-(2t-1)/3+1} = O(1 + 1/\epsilon) \tag{5}$$

The total expected time for a rehash is $O(n)$ because the probability of having to do one is smaller than 1. Therefore the expected time for an insertion is constant if $r \geq (1+\epsilon)(n+1)$. □

5 Experiments [1, Sect 4.2]

The paper covers three tests done on cuckoo hashing as well as the other hashing techniques presented before. Additionally they use an alternative cuckoo hashing approach. The difference to the normal cuckoo hashing approach is that the first table is double the size of the second one.

The first test starts by inserting a sequence of n distinct random keys, followed by $3n$ times the following operations. These operations are a random unsuccessful lookup, a random successful lookup, a random deletion and a random insertion. The results show that, lookup time is almost

identical for every hashing techniques. That is because in this test, the test case was rather small and the whole data structure fit into cache. Linear probing had the best performance with cuckoo hashing and Two-way chaining lagging behind because they make more computations and also more memory and cache accesses.
In another test were there are just n elements being inserted and deleted afterwards, linear probing kept its winning place.
In the third test with 32bit keys nothing surprising came up. Linear probing is still the best.
The conclusion is, that the number of memory accesses has a huge impact on the performance. Cuckoo hashing is only constant factors slower. Nevertheless it has many use cases because of its constant lookup time.

6 Conclusion

The paper takes seemingly simple idea and shows its potential. In comparison cuckoo hashing is slower than many of its competitors, especially for small dictionaries. It nevertheless has great potential because of its expected constant lookup time. I criticize on the paper the relative non straight forward proof. It is more of a proof outline than an actual proof. Maybe it was intended in this way. Also the handling of double hashing is strange. It gets introduced as a candidate for the experiment only to let it drop out of the graphs, because of its slower performance compared to linear probing and its small presence in the third test. Not a critic but still worth a note is that it would be interesting to see, how they perform on newer hardware with more cache capacity. Research on my part on differences has shown to be inconclusive.

References

[1] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing, 2001.

[2] Rassmus Pagh. *On the Cell Probe Complexity of Membership and Perfect Hashing.* Symposium on Foundation of Computer Science (FOCS 89), 1989.

[3] Wikipedia. Double hashing. [accessed 13-Okt-2018].

[4] Wikipedia. Hash table. [accessed 13-Okt-2018].

[5] Wikipedia. Linear probing. [accessed 13-Okt-2018].

[6] Anna R. Karlins Yossi Azar, Andrei Z. Broder and Eli Upfal. *BALANCED ALLOCATIONS.* SIAM J. COMPUT., 1999.