Collision Resolution: Cuckoo Hashing

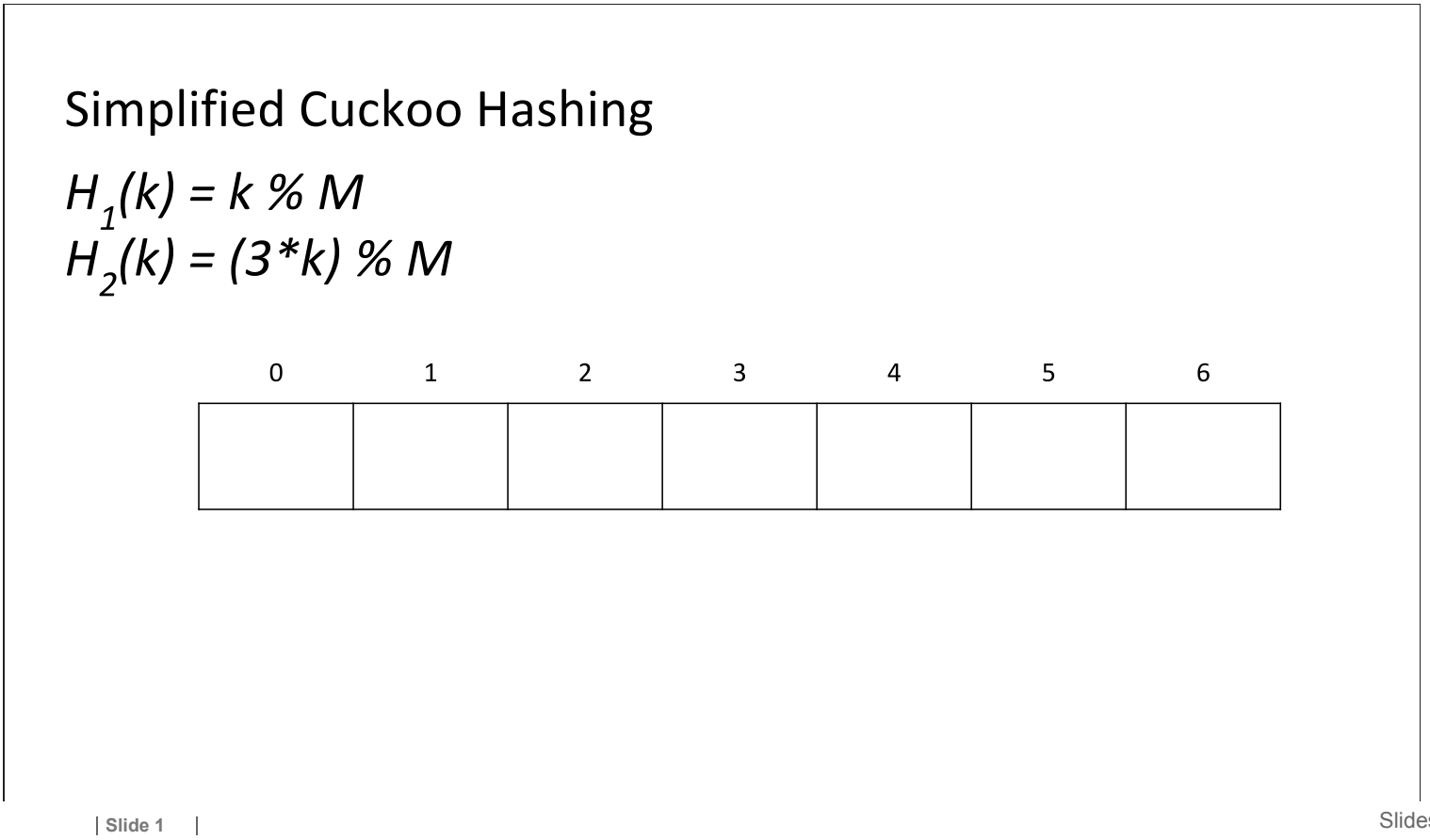
Step 1

The final collision resolution strategy that we will discuss is called **Cuckoo Hashing**. **Cuckoo Hashing**—and its weird name—comes from the concept of actual Cuckoo chicks pushing each other out of their nests in order to have more space to live. In all of the previous open addressing collision resolution strategies we have discussed thus far, if an inserting key collided with a key already in the **Hash Table**, the existing key would remain untouched and the inserting key would take responsibility to find a new place. In **Cuckoo Hashing**, however, an arguably more aggressive and opposite approach takes place: if an inserting key collides with a key already in the **Hash Table**, the inserting key pushes out the key in its way and takes its place. The displaced key then hashes to a new location (ouch...).

Figure: Chestnut-winged Cuckoo (photo by William Lee)

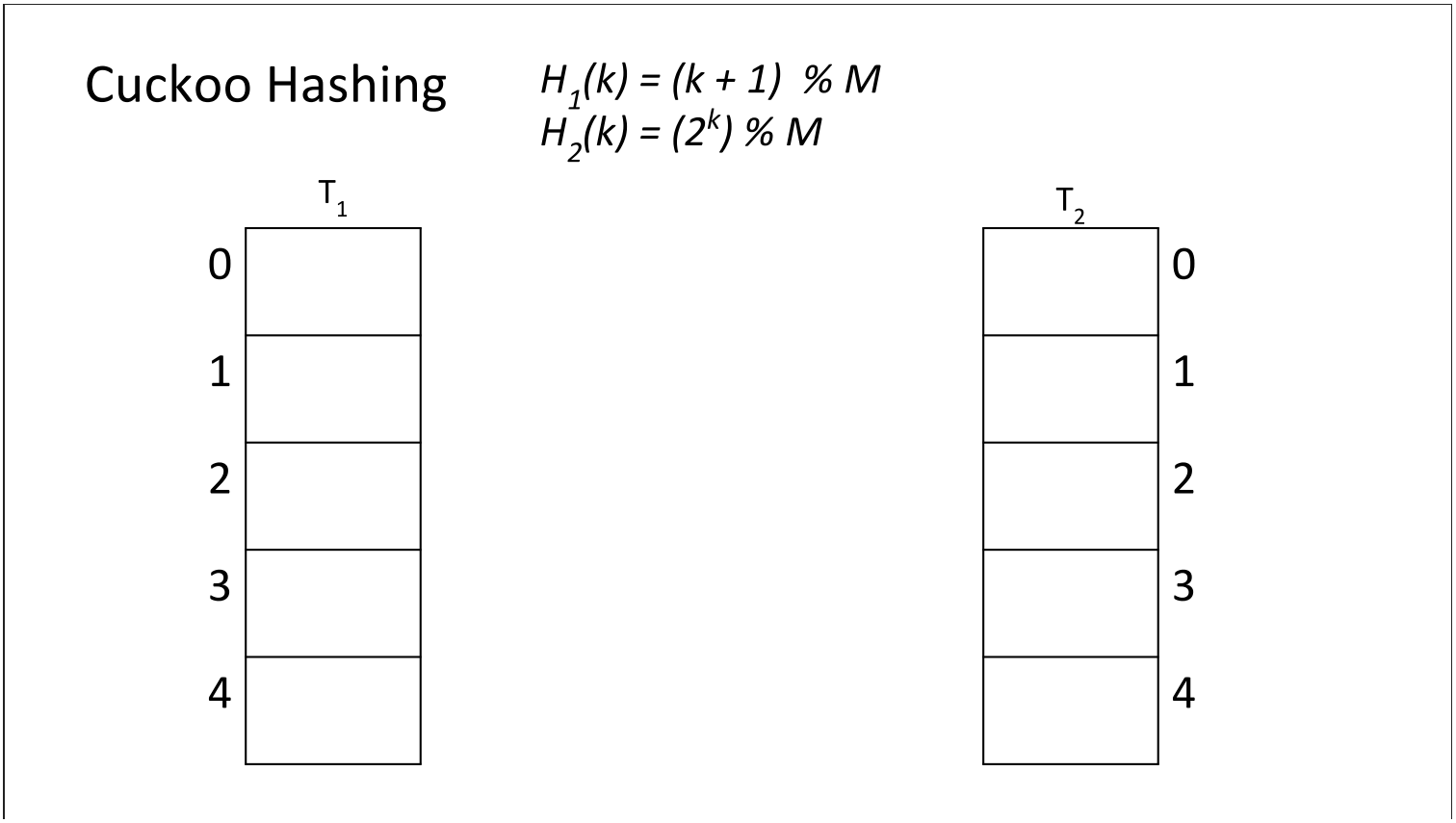
Step 2

More formally, **Cuckoo Hashing** is defined as having two **hash functions**, $H_1(k)$ and $H_2(k)$, both of which return a position in the **Hash Table**. As a result, one key has strictly two different hashing locations, where $H_1(k)$ is the first location that a key always maps to (but doesn't necessarily always stay at). A simplified version of **Cuckoo Hashing** can be seen in the animation below. We will see in the next step that the algorithm traditionally uses two **Hash Tables**, but for now, this simplified version uses just one. Use the arrows in the bottom left to walk through the slides.



Step 3

As mentioned in the previous step, **Cuckoo Hashing** was invented with the use of two **Hash Tables** simultaneously, in order to decrease the probability of collisions in each table, which is how **Cuckoo Hashing** is typically implemented today. Formally, the **hash function** $H_1(k)$ hashes keys *exclusively* to the *first* **Hash Table** T_1, and the **hash function** $H_2(k)$ hashes keys *exclusively* to the *second* **Hash Table** T_2. A key *k* starts by hashing to T_1, and if another arbitrary key *j* collides with key *k* at some point in the future, key *k* then hashes to T_2. **However**, a key can also get kicked out of T_2, in which case it hashes back to T_1 and potentially kicks out another key. We admit that this may sound confusing, so we have provided another visualization below to see how the keys jump around between the two tables. Use the arrows in the bottom left to walk through the slides.



STOP and Think: Is a *new* inserting key guaranteed to always end up in the index returned by the primary **hash function** $H_1(k)$?

Step 4

How do we go about actually implementing this seemingly complex collision resolution strategy? Here is pseudocode to implement **Cuckoo Hashing** when inserting a key *k*, with two tables `t1` and `t2`, both of capacity `M`, with backing arrays `arr1` and `arr2`, respectively, using a **hash function** `H1(k)` and `H2(k)`, respectively:

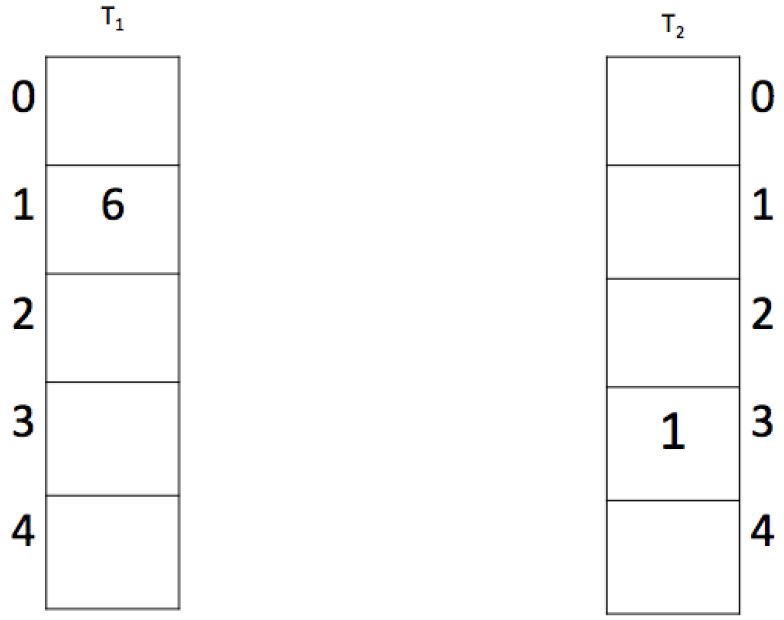
```
insert_CuckooHash(k): // return true upon successful insertion

    index1 = H1(k)
    index2 = H2(k)

    // check for duplicate insertions (not allowed)
    if arr1[index1] == k or arr2[index2] == k:
        return false

    current = k

    // loop for a limited amount of time (we will discuss details in the next step)
    while looping less than MAX times: // MAX is commonly set to 10

        // insert until the slot inserted in is empty
        oldValue = arr1[H1(current)]   // save the value currently in the slot
        arr1[H1(current)] = current    // insert the new key

        if oldValue == NULL:           // if slot was empty, we are done inserting
            return true

        current = oldValue             // time to re-insert what was kicked out

        oldValue = arr2[H2(current)]   // save the value currently in the slot
        arr2[H2(current)] = current    // insert the new key

        if oldValue == NULL:           // if slot was empty, we are done inserting
            return true

        // repeat loop, but with the key displaced from arr2
        current = oldValue

     // loop ended, so insertion failed (need to rehash the table)
     // rehash is commonly done by introducing two new hash functions
     return false
```

STOP and Think: Is **Cuckoo Hashing** considered to be an **Open Addressing** collision resolution strategy?

Step 5

EXERCISE BREAK: Based on the **Hash Tables** and **hash functions** below, write the corresponding index to which each of the keys will hash—using the collision resolution strategy of **Cuckoo Hashing**—*after* key 11 is inserted. Do not worry about *which* **Hash Table** the keys are inserted in: you only need to type the number of the index.

$H_1(k) = k \% M$
$H_2(k) = (3^k) \% M$

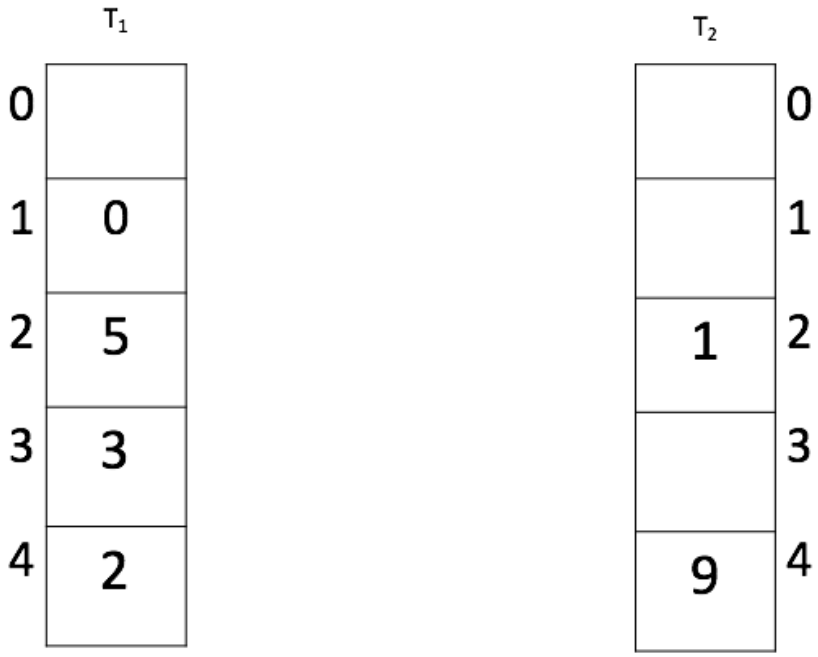


To solve this problem please visit https://stepik.org/lesson/31226/step/5

Step 6

EXERCISE BREAK: Based on the **Hash Tables** and **hash functions** below, write the corresponding index to which each of the keys will hash—using the collision resolution strategy of **Cuckoo Hashing**—*after* key 7 is inserted. Do not worry about *which* **Hash Table** the keys are inserted in: you only need to type the number of the index.

$H_1(k) = (2^k) \% M$
$H_2(k) = (k + 1) \% M$



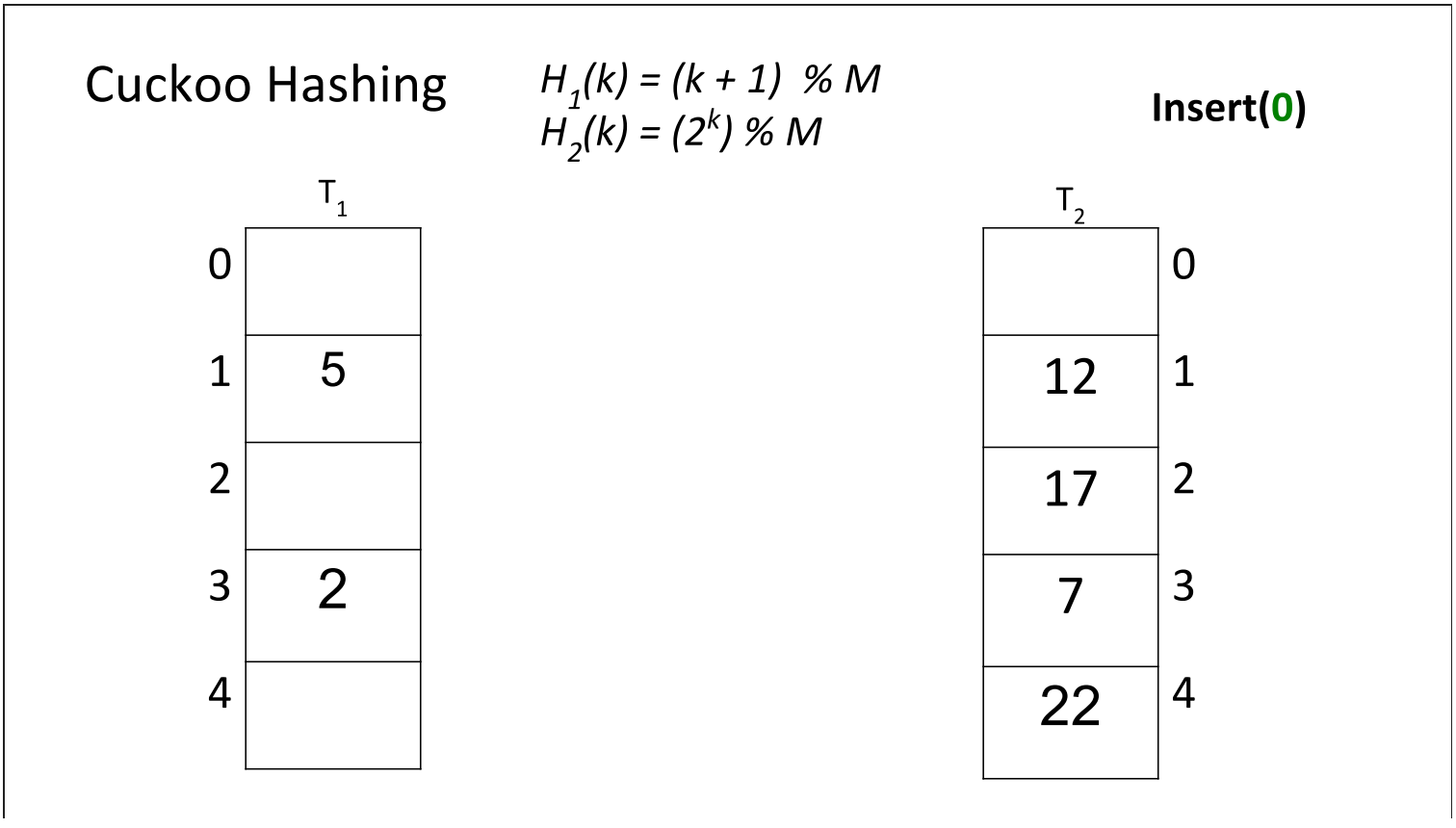
To solve this problem please visit https://stepik.org/lesson/31226/step/6

Step 7

In the previous step, we saw that the bulk of the algorithm ran inside a while-loop that was bounded by a MAX limit. Why do we even have a limit in the first place?

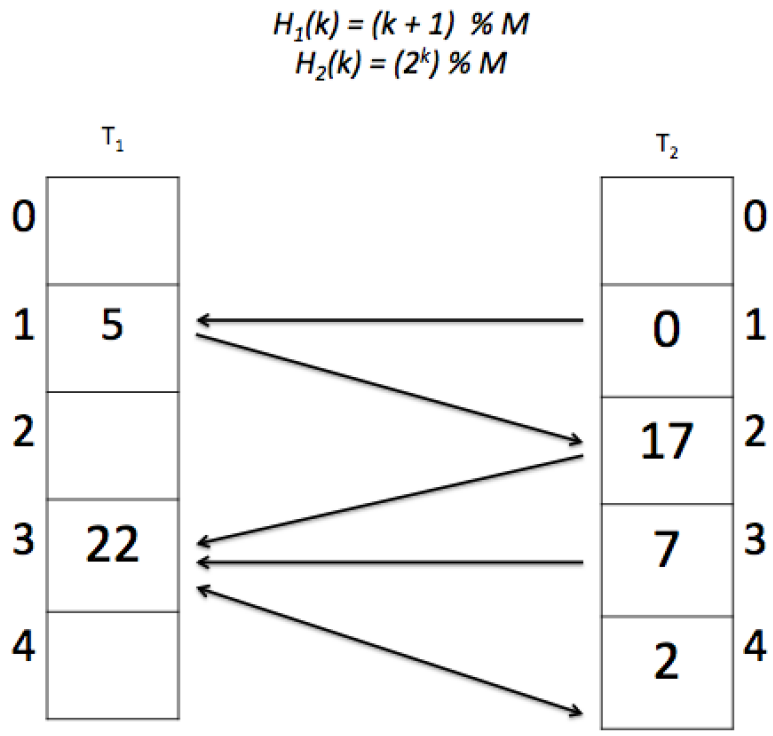
You may have noticed that, in the second visualization, we provided that inserting a key started to take longer because different keys started bouncing back and forth between places. For example, inserting the integer key 17 took 4 iterations of hashing. As both tables begin to fill up, the probability of collisions increases. *However*, unlike the other collision resolution strategies we discussed in which a key *k* could end up trying every single **Hash Table** slot until the entire **Hash Table** was full (e.g. **Linear Probing**, **Double Hashing**, **Random Hashing**) a key *k* in **Cuckoo Hashing** only has **two** different locations that it can map to (index1 = $H_1(k)$ and index2 = $H_2(k)$).

Consequently, if both **Hash Table** locations for *all* keys are full, then the algorithm faces an infinite cycle, as seen in the visualization below (this visualization leaves off from where the previous one ended). Use the arrows in the bottom left to walk through the slides.



Step 8

Specifically, the reason why we were in an infinite cycle in the previous visualization is that there were no empty slots in either T_1 or T_2 in which a key could potentially land, like so:



Consequently, unless **Cuckoo Hashing** has a limit as to how many times it attempts to move a key, the algorithm will never stop. As seen in the pseudocode in the previous step, when a key causes an infinite loop and the insertion fails, we must then resort to rehashing. Rehashing is generally done by introducing two new **hash functions** and reinserting all the elements.

Note: It is important to make sure that the second **hash function** used returns *different* indices for keys that originally hashed to the same index. This is because, if a key collides with another key in the first **Hash Table**, we want to make sure that it will not collide with the same key again in the second **Hash Table**. Otherwise, we risk hitting a cycle the moment we insert two keys that hash to the same first index.

STOP and Think: Let *k* be a key and *M* be the capacity of the **Hash Table**. Why would the pair of **hash functions** $H_1(k) = k \% M$and $H_2(k) = (k + 3) \% M$ not be considered good?

Step 9

By making the sacrifice of only allowing each key to hash to strictly two different locations (thereby potentially causing the cycle in the first place), we end up getting a reward of a ***worst-case constant*** time complexity for two of our major operations! Specifically:

- For the "find" operation: if the key is not in either index1 = $H_1(k)$ or index2 = $H_2(k)$, then it is not in the table; this is a constant time operation
- For the "delete" operation: if the key exists in our table, we know that it is either in index1 = $H_1(k)$ or index2 = $H_2(k)$, and all we have to do is remove the key from its current index; this is a constant time operation

This is unique to **Cuckoo Hashing** because, in all of the other open addressing collision resolution strategies we have discussed thus far, we could only guarantee an **average-case** constant time complexity with respect to those two operations because, in the worst case, we had to traverse the entire **Hash Table**.

For the "insert" operation in **Cuckoo Hashing**, however, we only get an *average* case constant time complexity because, in the worst case, we would have to rehash the entire table, which has an **O(*n*)** time complexity. However, we can prove through amortized cost analysis (which is beyond the scope of this text) that, on *average*, we can finish an insertion before the algorithm is forced to terminate and rehash.

Fun Fact: A lot of proofs about cycles in **Cuckoo Hashing** are solved by converting the keys within the two **Hash Tables** to nodes and their two possible hashing locations to edges to create a graph theory problem!

Note: Cuckoo Hashing is not necessarily restricted to using just two **Hash Tables**; it is not uncommon to use more than two **Hash Tables**, and generally, for *d* **Hash Tables**, each **Hash Table** would have a capacity of $\frac{M}{d}$, where *M* is the calculated capacity of a single **Hash Table** (i.e., the capacity that we would have calculated had we decided to use a different collision resolution strategy that required only one **Hash Table**).

Step 10

EXERCISE BREAK: Which of the following statements about **Cuckoo Hashing** are true? (Select all that apply)

Note: Assume that we are using only two **Hash Tables** in our implementation of **Cuckoo Hashing**

To solve this problem please visit https://stepik.org/lesson/31226/step/10

Step 11

Hopefully by now you are convinced that **Cuckoo Hashing** is by far the most optimized open addressing collision resolution strategy that we have discussed in terms of worst-case time complexities. Specifically, it provides us a guaranteed **worst-case constant time** complexity in the "find" and "remove" operations that the previous strategies were unable to guarantee.

Because of the complex nature of **Cuckoo Hashing**, however, there are many more analyses and proofs that this lesson has left out for the sake of simplicity. If you are curious and have the desire to skim/read the original 26-page paper that describes the capabilities of **Cuckoo Hashing** beyond what we have described in this lesson, then you can find it here. There is also another much shorter and accessible paper that is intended for undergraduates here.

Also, if you want to play around with inserting and deleting elements, you can find an excellent visualization tool here.