CS420+500: Advanced Algorithm Design and Analysis

Lectures: April 3rd + 5th, 2017

Prof. Will Evans *Scribe: Yasha Pushak*

In this lecture we:

- Discussed Universal Hash Functions;
- and Cuckoo Hashing.

Handouts (posted on webpage):

- Rasmus Pagh's Cuckoo Hashing for Undergraduates

1 Universal Hash Functions

The expected load of a bucket in a universal hash table

Theorem 1. *For key k,* $\mathbb{E}\left[n_{h(k)}\right] \leq \begin{cases} \alpha + 1 & \text{if } k \in T \\ \alpha & \text{otherwise,} \end{cases}$

where n_i = *the number of items in table T bucket i,*
n = *the total number of items in T,*
m = *the size of T,*
$\alpha = \frac{n}{m}$, *the load factor.*

Proof. Let the random variable $X_{k,l} = \begin{cases} 1 & \text{if } h(k) = h(l) \\ 0 & \text{otherwise.} \end{cases}$

Let the random variable Y_k be the number of keys that hash to the same spot as k (not including k). Then we have

$$Y_k = \sum_{l \neq k, l \in T} X_{k,l},$$

so

$$\mathbb{E}\left[Y_k\right] = \mathbb{E}\left[\sum_{l \neq k, l \in T} X_{k,l}\right],$$

since expectations are linear operations we have

$$= \sum_{l \neq k, l \in T} \mathbb{E}\left[X_{k,l}\right],$$

and since T is built using a universal hash function, we know that the probability of $h(k) = h(l)$ is $\frac{1}{m}$, hence we have

$$= \sum_{l \neq k, l \in T} \frac{1}{m}.$$

All that remains are to consider the two cases, $k \in T$ and $k \notin T$.

If $k \in T$, then $n_{h(k)} = Y_k$ and $\left|\{l \in T \,|\, l \neq k\}\right| = n$.

$$\begin{aligned} \implies \mathbb{E}\left[n_{h(k)}\right] &= \mathbb{E}\left[Y_k\right] \\ &= \frac{n}{m} \\ &= \alpha \end{aligned}$$

.

If $k \notin T$, then $n_{h(k)} = Y_k + 1$ and $\left|\{l \in T \,|\, l \neq k\}\right| = n - 1$

$$\begin{aligned} \implies \mathbb{E}\left[n_{h(k)}\right] &= \mathbb{E}\left[Y_k + 1\right] \\ &= \mathbb{E}\left[Y_k\right] + 1 \\ &= \frac{n-1}{m} + 1 \\ &\leq \frac{n}{m} + 1 \\ &= \alpha + 1 \end{aligned}$$

□

Example of a universal set of hash functions

$$h_{a,b}(k) = ((a \cdot k + b) \texttt{ mod } p) \texttt{ mod } m),$$

where p is a (fixed) prime bigger than any key, and we chose

$$a \in \{1, 2, ..., p-1\} \text{ and } b \in \{0, 1, ..., p-1\}$$

at random to select a hash function from the set.

2 Cuckoo Hashing

Cost of Operations The cost of some of the standard operations for cuckoo hash tables are:

- `find` $\in \mathcal{O}(1)$
- `delete` $\in \mathcal{O}(1)$
- `insert` $\in \mathcal{O}(1)$ (amortized, expected)

Intuition The cuckoo bird is a parasitic bird that lays its eggs in the nest of other birds. To avoid detection, the bird first kicks out one of the original eggs from the nest. Then, when the cuckoo bird egg hatches, the first thing it does it kick all of the other eggs out of the nest.

Cuckoo hash tables use two different hash functions h_1 and h_2. Similar to the cuckoo bird, when a new item is inserted into a cuckoo hash table and there is a collision, the new item kicks out the old item from the table. The old item is then forced to use the alternate hash function and is placed in its secondary position. If there is already an item there, it too is kicked out, and that item attempts to use the alternate hash function (which may be either h_1 or h_2, depending on where the evicted item began).

This means that the table will have no more than one key in each spot.

Item i	$h_1(i)$	$h_2(i)$
0	D	A
1	C	B
2	D	G
4	F	G
5	E	D
6	E	F
7	A	H

Table 1: A sequence of 7 inserts into a cuckoo hash table. Note that there is no item labeled 3.

The `insert` operation

```
insert(k)
    pos = h1(k)
    repeat n times
    {
        b = T[pos]
        T[pos] = k
        if (b == NULL)
            return
        \\ Cuckoo k has kicked out b
        if (pos = h1(b))
            then pos = h2(b)
        else
            pos = h2(b)
        \\ b becomes the new cuckoo
        k = b
    }
    \\ The Cuckoos are cycling, so we need to use
    \\ new hash functions and a larger table
    rehash()
    insert(k)
```

Cuckoo graph for n items Vertices $= \{v_0, v_1, ..., v_{m-1}\}$ and Edges $= \{(h_1(k), h_2(k)) \,|\, k \in T\}$ Each vertex in the graph represents one location in the hash table. The edges in the graph represent elements in the hash table, and they connect the two vertices to which an element k are mapped by the two hash functions h_1 and h_2.

As an example, let's consider the graph with vertices $= \{A, B, C, D, E, F, G, H\}$ with the sequence of seven item inserts in Table 2 (note that there is no item with label 3)). The following sequence of figures, Figures 1 to 16 shows the result of the insertions on the cuckoo graph. We use directed edges to represent the location at which the elements are stored in the hash table.

Figure 1: The empty cuckoo graph, before any elements are inserted.

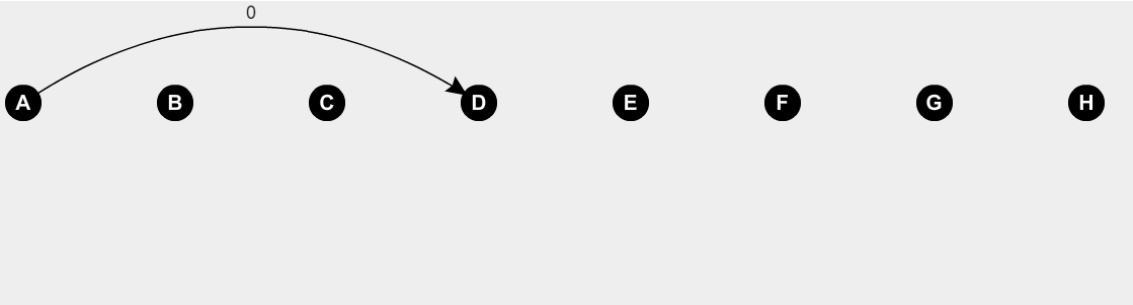

Figure 2: After the element 0 is inserted, it is located at the position of $h_1(0) = D$.

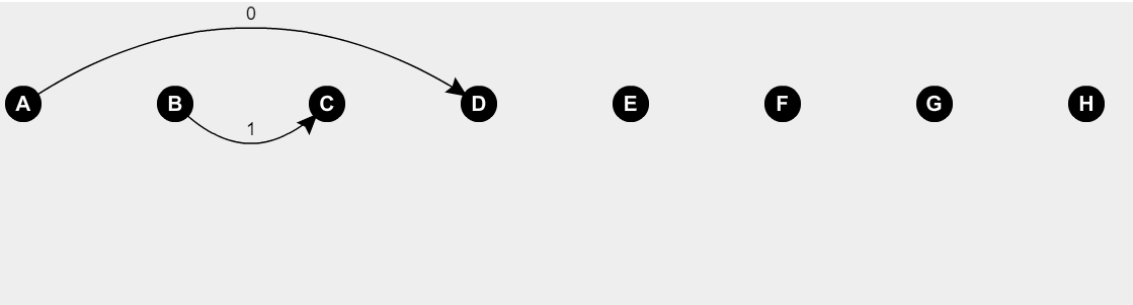

Figure 3: The cuckoo graph after the second element, 1, is inserted.

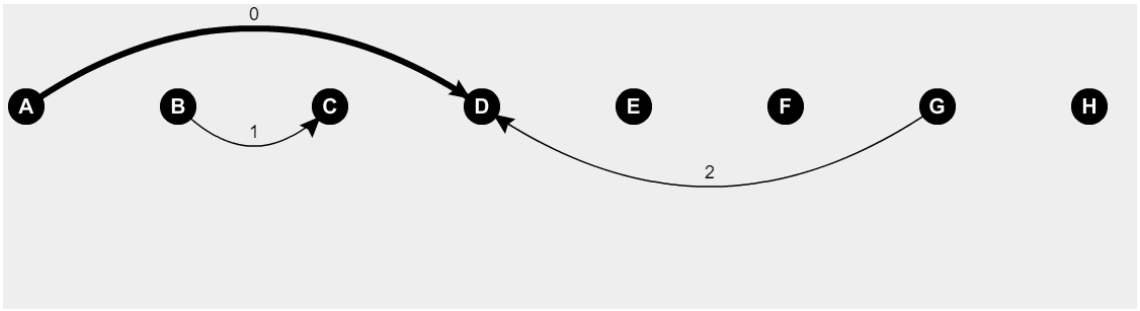

Figure 4: When the third element, 2, is inserted, it causes a collision, with element 0. We highlight this by showing the edge for element 0 in bold. At this point, element 0 is kicked out of its location, and must move to its back-up position, A.

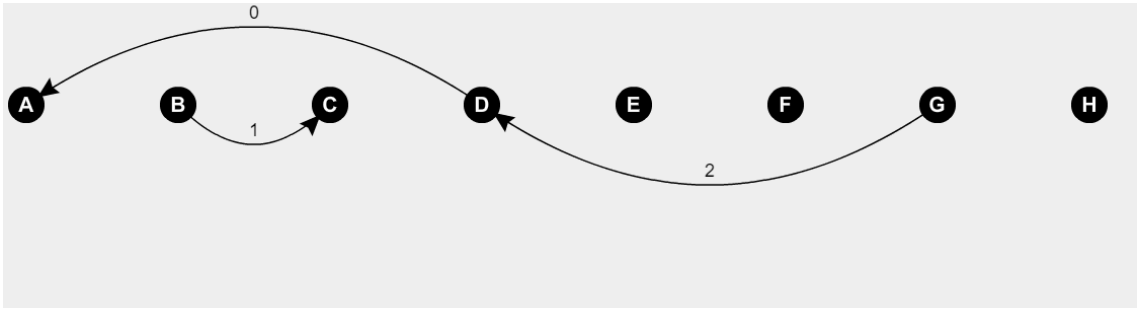

Figure 5: The cuckoo graph after the second insert has been resolved.

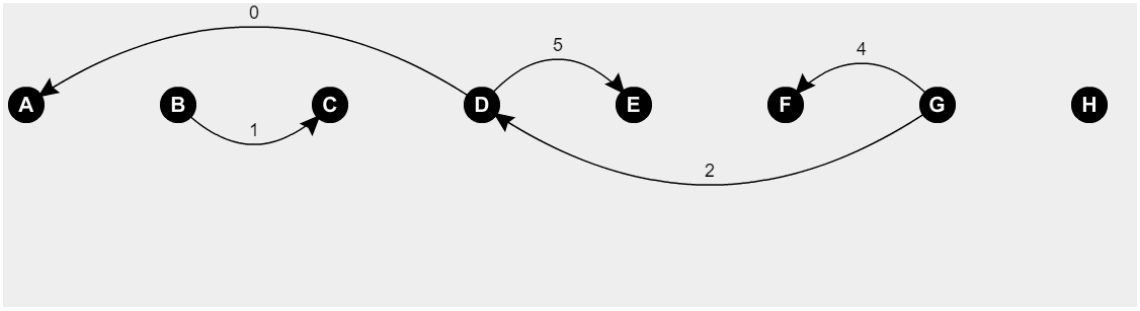

Figure 6: The next two inserts are uneventful, placing elements 4 and 5 into their respective locations. (Note that there is no element 3.)

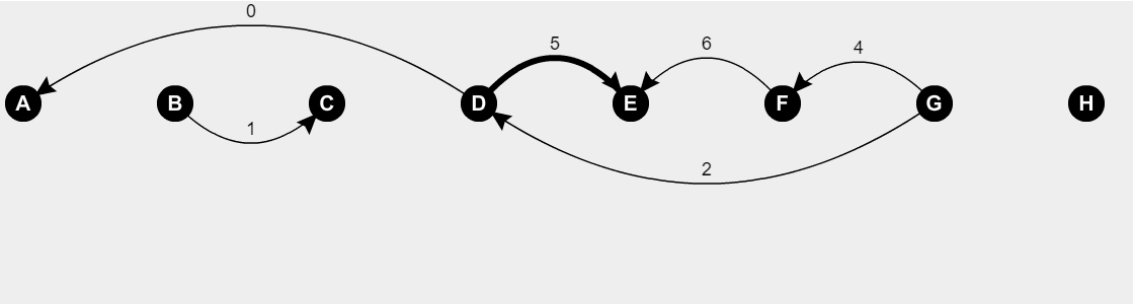

Figure 7: When element 6 is inserted it causes a collision with element 5, which becomes the new cuckoo.

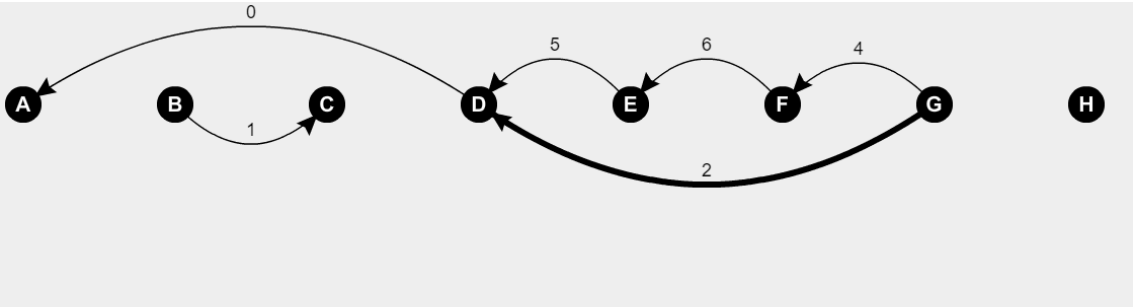

Figure 8: Unfortunately, moving element 5 also causes a collision, which results in element 2 being evicted.

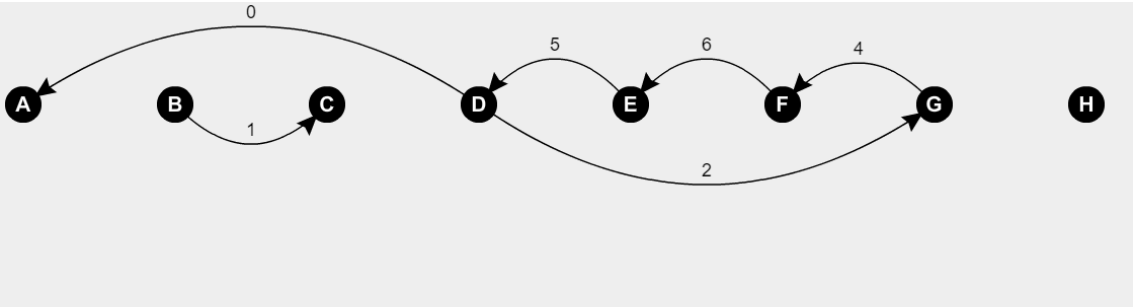

Figure 9: Despite the presence of the cycle, element 2 finds an open position at its back-up location, G. So no further actions are necessary for this insert operation.

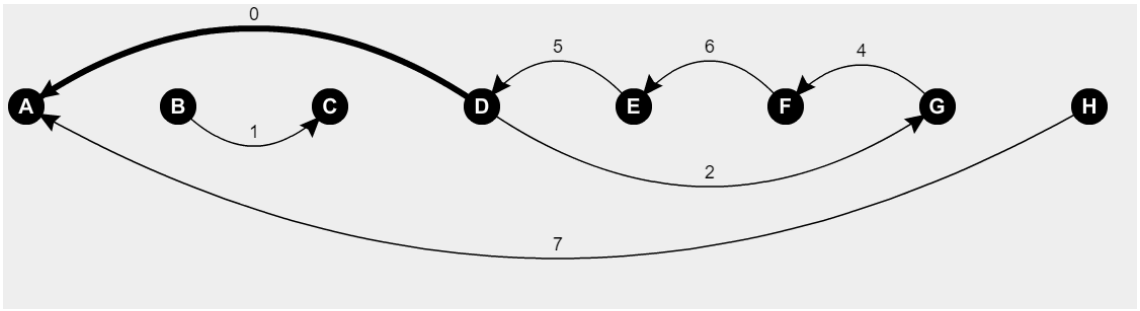

Figure 10: When element 7 is inserted, it results in a collision with element 0, causing element 0 to be evicted and become the new cuckoo. This is the first collision on this insertion operation.

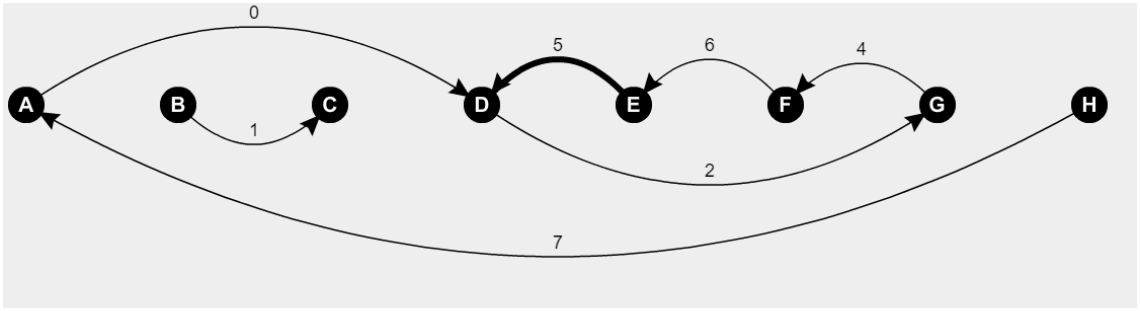

Figure 11: Moving element 0 causes the second collision for this insertion, which evicts element 5.

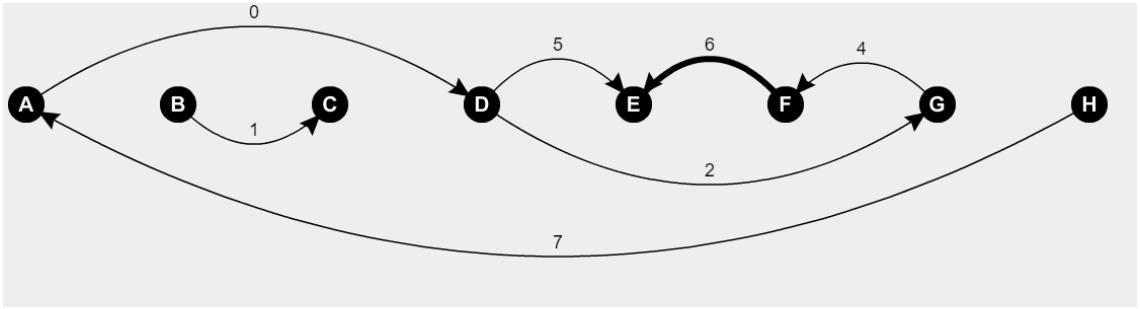

Figure 12: Moving element 5 causes the third collision for this insertion, which evicts element 6.

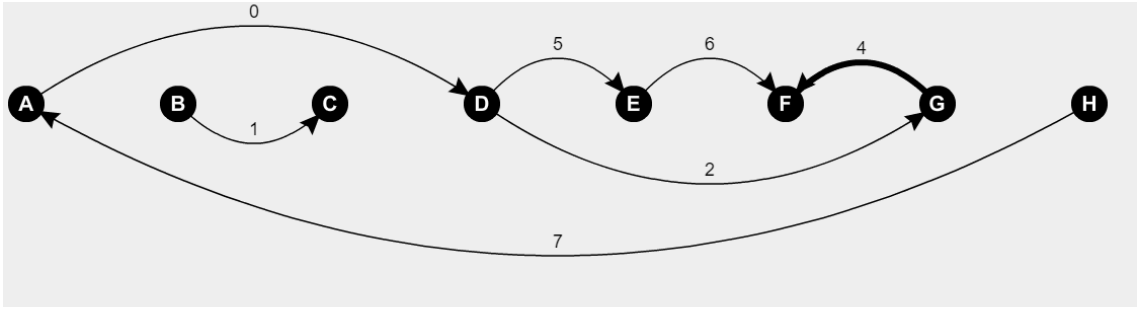

Figure 13: Moving element 6 causes the fourth collision for this insertion, which evicts element 4.

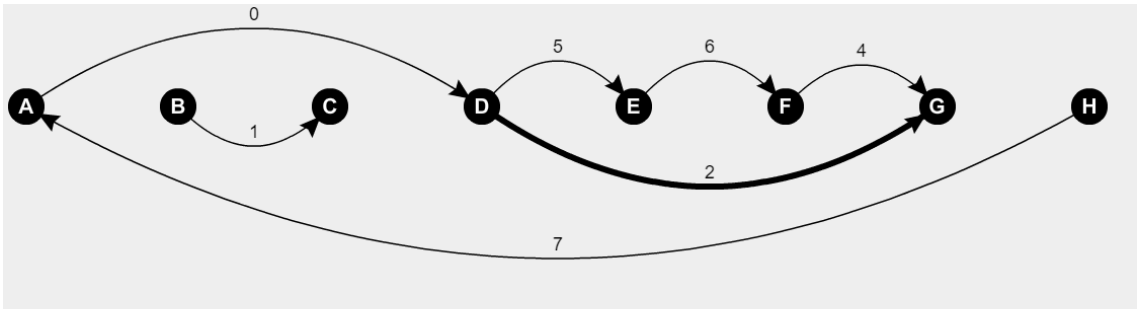

Figure 14: Moving element 4 causes the fifth collision for this insertion, which evicts element 2.

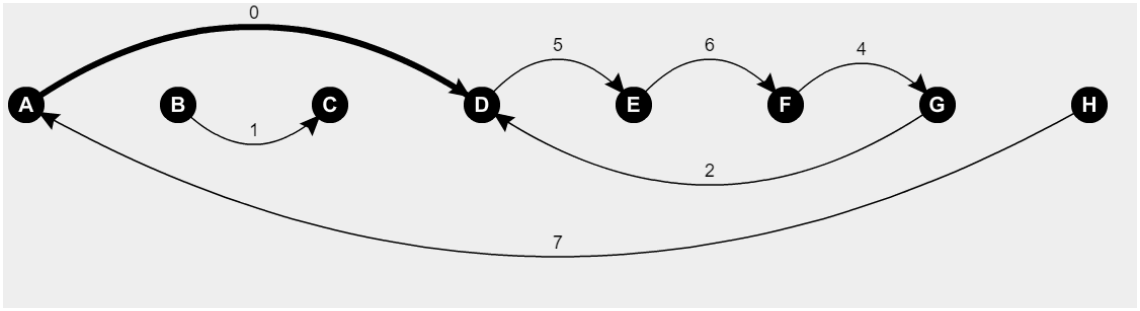

Figure 15: Moving element 2 causes the sixth collision for this insertion, which evicts element 0 (again).

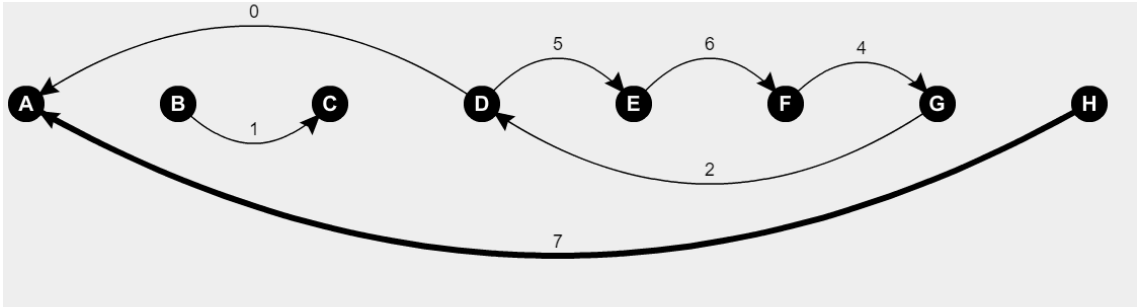

Figure 16: Moving element 0 causes the seventh collision for this insertion, which evicts element 7. However, at this point, even though the insertion could be resolved by simply moving element 7 to its empty, back-up location at H, the algorithm triggers a rehash operation. This is because we have encountered 7 collisions and we only have 7 elements in the table. As a result, we know that there must be a cycle somewhere in the graph, which, if left alone, would likely degrade performance for future insertions.

Cuckoo graph paths and cycles Clearly, an insertion operation will succeed if there is no cycle in the cukcoo graph. Before we analyze cycles, we first consider paths in the cuckoo graph.

Lemma 2. *For some $c > 1$, if $m \geq 2 \cdot c \cdot n$ then the probability that the cuckoo graph has a shortest path of length $l \geq 1$ from i to j is $\leq \frac{1}{m \cdot c^l}$.*

Proof. By induction on l.

Base case: $l = 1$

The edge (i, j) exists in the cuckoo graph with probability $\leq n \cdot \frac{2}{m^2}$. That is, the probability of a single random edge being (i, j) is $\frac{1}{m}$, but since the edge $(j, i) = (i, j)$ we have the factor 2, and since we have n edges, we get $n \cdot \frac{2}{m^2}$. However, since we know that $2 \cdot c \cdot n \leq m$ we have

$$\begin{aligned} n \cdot \frac{2}{m^2} &\leq n \cdot \frac{2}{2 \cdot c \cdot n \cdot m}, \\ &= \cdot \frac{1}{c \cdot m}, \\ &= \cdot \frac{1}{c^1 \cdot m}, \\ &= \cdot \frac{1}{c^l \cdot m}, \end{aligned}$$

as needed.

Inductive step: $l \geq 1$

The shortest path from i to j has length l only if there exists a vertex p such that

1. there exists a path from i to p of length $l - 1$; and
2. there exists the edge (p, j).

By induction, we have that condition 1 occurs with probability $\leq \frac{1}{m \cdot c^{l-1}}$ and condition 2 occurs with probability $\leq \frac{1}{m \cdot c}$. Together, this give a probability $\leq \frac{1}{m^2 \cdot c^l}$, but when you sum over all possible vertices p, we get that the probability is $\leq m \cdot \frac{1}{m^2 \cdot c^l} = \frac{1}{m \cdot c^l}$

Now, the probability that k and k' hash to the same path ("bucket") of the cuckoo graph is the probability of a path from $h_1(k)$ or $h_2(k)$ to $h_1(k')$ or $h_2(k')$, which is

$$\leq 4 \cdot \sum_{l=1}^{\infty} \frac{1}{m \cdot c^l},$$

which converges, via an argument using the geometric series, to

$$= \frac{4}{m} \cdot \frac{1}{c - 1} \in \mathcal{O}(\frac{1}{m}).$$

□

This result means that we have all of the constant time operations in regular hash tables, with high probability.

Rehashing Rehashing means that we must choose new hash functions and rehash all of the keys. In addition, when a cuckoo graph is rehashed, the size of the table m is increased; however, for simplicity in the following we show that a rehash succeeds (i.e., does not introduce a new cycle) with high probability, even when m is held constant.

Probability that a rehash occurs $\leq$ the probability that hashing creates a cuckoo graph with a cycle.

$$\leq \sum_{i=1}^{m} \text{Prob.}[\text{ cycle involing } i]$$

$$= \sum_{i=1}^{m} \text{Prob.[path from } i \text{ to } i]$$

$$= \sum_{i=1}^{m} \sum_{l=1}^{\infty} \frac{1}{m \cdot c^l}$$

$$\leq \frac{1}{2}, \text{ if } c \geq 3$$

This means that the expected number of rehashes is ≤ 2 (and in fact, is actually closer to 1, when m is increased).