Advanced Algorithms – COMS31900

Hashing part three

Cuckoo Hashing

Raphaël Clifford

Slides by Benjamin Sach
Back to the start (again)

- A dynamic dictionary stores *(key, value)*-pairs and supports:
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

Universe $U$ of $u$ keys.

Hash table $T$ of size $m \geq n$.

Collisions are fixed by chaining.

A hash function maps a key $x$ to position $h(x)$

$n$ arbitrary operations arrive online, one at a time.

A set $H$ of hash functions is weakly universal if for any two keys $x, y \in U$ (with $x \neq y$),

$$\Pr (h(x) = h(y)) \leq \frac{1}{m}$$

($h$ is picked uniformly at random from $H$)

Using weakly universal hashing:

For any $n$ operations, the expected run-time is $O(1)$ per operation.
Back to the start (again)

- A dynamic dictionary stores *(key, value)-pairs and supports:

  \[
  \text{add}(\text{key, value}), \text{lookup}(\text{key}) \text{ (which returns value) and delete(\text{key})}
  \]

Universe \( U \) of \( u \) keys.

Hash table \( T \) of size \( m \geq n \).

Collisions are fixed by chaining

A hash function maps a key \( x \) to position \( h(x) \)

\( n \) arbitrary operations arrive online, one at a time.

A set \( H \) of hash functions is weakly universal if for any two keys \( x, y \in U \) (with \( x \neq y \)),

\[
Pr \left( h(x) = h(y) \right) \leq \frac{1}{m}
\]

\((h \text{ is picked uniformly at random from } H)\)

Using weakly universal hashing:

For any \( n \) operations, the expected run-time is \( O(1) \) per operation.

*in fact this result can be generalised*
A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **chaining**.

\(n\) arbitrary operations arrive online, one at a time.

A **hash function** maps a key \(x\) to position \(h(x)\).

A set \(H\) of hash functions is **weakly universal** if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

**Using weakly universal hashing:**

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised} \ldots\)
A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[
\text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and} \quad \text{delete}(key)
\]

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h \text{ is picked uniformly at random from } H)\)

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised} \ldots\)
A dynamic dictionary stores \((key, value)\)-pairs and supports:

- `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

---

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **chaining**

**A hash function** maps a key \(x\) to position \(h(x)\)

\(n\) arbitrary operations arrive online, one at a time.

---

A set \(H\) of hash functions is **weakly universal** if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr (h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\ is\ picked\ uniformly\ at\ random\ from\ H)\)

---

**Using weakly universal hashing:**

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised . . .}\)
A dynamic dictionary stores \((\text{key}, \text{value})\)-pairs and supports:

- \(\text{add}(\text{key}, \text{value})\), \(\text{lookup}(\text{key})\) (which returns \text{value}) and \(\text{delete}(\text{key})\).

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by chaining.

A hash function maps a key \(x\) to position \(h(x)\).

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is \textbf{weakly universal} if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

(\(h\) is picked uniformly at random from \(H\)).

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised} \ldots\)
A dynamic dictionary stores \((key,\ value)\)-pairs and supports:

- \text{add}(\text{key, value})
- \text{lookup}(\text{key}) \text{ (which returns value)}
- \text{delete}(\text{key})

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by 

We require that we can recover any key from its bucket in \(O(s)\) time, where \(s\) is the number of keys in the bucket.

\(n\) arbitrary operations arrive online, one at a time.

A hash function maps a key \(x\) to position \(h(x)\).

A set \(H\) of hash functions is weakly universal if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\ \text{is picked uniformly at random from } H)\)

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised . . .}\)
Back to the start (again)

- **A dynamic dictionary** stores \((key, value)\)-pairs and supports:

  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
  \]

---

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **chaining**.

We require that we can recover any key from its **bucket** in \(O(s)\) time where \(s\) is the number of keys in the **bucket**.

\(n\) arbitrary operations arrive online, one at a time.

---

A set \(H\) of hash functions is **weakly universal** if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h \text{ is picked uniformly at random from } H)\)

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

*in fact this result can be generalised*
Back to the start (again)

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

\[
\text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
\]

---

**Universe** \(U\) of \(u\) keys.

**Hash table** \(T\) of size \(m \geq n\).

Collisions are fixed by [chaining](#) or [bucketing](#).

We require that we can recover any key from its **bucket** in \(O(s)\) time

where \(s\) is the number of keys in the bucket

\(n\) arbitrary operations arrive online, one at a time.

---

A set \(H\) of hash functions is **weakly universal** if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr (h(x) = h(y)) \leq \frac{1}{m}
\]

\((h \text{ is picked uniformly at random from } H)\)

---

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

*in fact this result can be generalised ...*
Back to the start (again)

A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**Universe** \(U\) of \(u\) keys.

**Hash table** \(T\) of size \(m \geq n\).

Collisions are fixed by **chaining**

We require that we can recover any key from its **bucket** in \(O(s)\) time

where \(s\) is the number of keys in the bucket

\(n\) arbitrary operations arrive online, one at a time.

A set \(H\) of hash functions is **weakly universal** if for any two keys \(x, y \in U\) (with \(x \neq y\)),

\[
\Pr(h(x) = h(y)) \leq \frac{1}{m}
\]

\((h\) is picked uniformly at random from \(H\))

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

\(\text{in fact this result can be generalised} \ldots\)
Back to the start (again)

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  - \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.
Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by

- **bucketing**

- Locating the bucket containing a given key takes \(O(1)\) time

We require that we can recover any key from its bucket in \(O(s)\) time where \(s\) is the number of keys in the bucket

\(n\) arbitrary operations arrive online, one at a time.
Back to the start (again)

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and } \text{delete}(key)
  \]

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **bucketing**

Locating the bucket containing a given key takes \(O(1)\) time

We require that we can recover any key from its bucket in \(O(s)\) time where \(s\) is the number of keys in the bucket

\(n\) arbitrary operations arrive online, one at a time.

If our construction has the property that,
for any two keys \(x, y \in U\) (with \(x \neq y\)),
the probability that \(x\) and \(y\) are in the same bucket is \(O\left(\frac{1}{m}\right)\)
Back to the start (again)

- **A dynamic dictionary** stores \((key, value)\)-pairs and supports:

  \[
  \text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and delete}(key)
  \]

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by **bucketing**

We require that we can recover any key from its **bucket** in \(O(s)\) time

\(n\) arbitrary operations arrive online, one at a time.

If our construction has the property that, for any two keys \(x, y \in U\) (with \(x \neq y\)),

the probability that \(x\) and \(y\) are in the same bucket is \(O\left(\frac{1}{m}\right)\)
A dynamic dictionary stores \((\text{key}, \text{value})\)-pairs and supports:

- \(\text{add}(\text{key}, \text{value})\), \(\text{lookup}(\text{key})\) (which returns \text{value}) and \(\text{delete}(\text{key})\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by bucketing.

We require that we can recover any key from its bucket in \(O(s)\) time

where \(s\) is the number of keys in the bucket

Locating the bucket containing a given key takes \(O(1)\) time

\(n\) arbitrary operations arrive online, one at a time.

If our construction has the property that,

for any two keys \(x, y \in U\) (with \(x \neq y\)),

the probability that \(x\) and \(y\) are in the same bucket is \(O\left(\frac{1}{m}\right)\)

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \text{add}(key, value), \text{lookup}(key) (which returns value) and \text{delete}(key)

\begin{tcolorbox}
\textbf{Theorem}

In the Cuckoo hashing scheme:
- Every \text{lookup} and every \text{delete} takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time
\end{tcolorbox}
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
  \]

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

What does *amortised expected \(O(1)\) time* mean?!
Dynamic perfect hashing

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

  add\((key, value)\), lookup\((key)\) (which returns \(value\)) and delete\((key)\)

---

**THEOREM**

In the **Cuckoo hashing** scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes **amortised expected** \(O(1)\) time

What does **amortised expected** \(O(1)\) time mean?!  

*let's build it up...*
Dynamic perfect hashing

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:

  \[\text{add}(key, value), \text{lookup}(key)\] (which returns \text{value}) and \text{delete}(key)

---

**Theorem**

In the **Cuckoo hashing** scheme:

- Every \text{lookup} and every \text{delete} takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An \text{insert} takes \textit{amortised expected} \(O(1)\) time

What does \textit{amortised expected} \(O(1)\) time mean?!

\[O(1)\] worst-case time per operation

means every operation takes constant time
Dynamic perfect hashing

- A **dynamic dictionary** stores \((\text{key}, \text{value})\)-pairs and supports:
  - `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

**Theorem**

In the **Cuckoo hashing** scheme:
- Every `lookup` and every `delete` takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes **amortised expected** \(O(1)\) time

What does **amortised expected** \(O(1)\) time mean?!

*let’s build it up…*

“\(O(1)\) worst-case time per operation”

means every operation takes constant time

“The total worst-case time complexity of performing any \(n\) operations is \(O(n)\)”
Dynamic perfect hashing

- A dynamic dictionary stores \((\text{key}, \text{value})\)-pairs and supports:

  \[
  \text{add(} \text{key}, \text{value}) \text{, lookup(} \text{key}) \quad \text{(which returns } \text{value}) \text{ and delete(} \text{key})
  \]

THEOREM

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

What does \textit{amortised expected } \(O(1)\) time mean?! \textit{let’s build it up...}

\textit{“} \(O(1)\) worst-case time per operation”

means every operation takes constant time

\textit{“The total worst-case time complexity of performing any } n \text{ operations is } O(n)”

this \textbf{does not} imply that every operation takes constant time
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\( \text{add}(key, value), \text{lookup}(key) \) (which returns \(value\)) and \( \text{delete}(key) \)

**Theorem**

In the Cuckoo hashing scheme:

- Every \text{lookup} and every \text{delete} takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes \text{amortised expected} \(O(1)\) time

What does \text{amortised expected} \(O(1)\) time mean?! *let's build it up...*

“\(O(1)\) worst-case time per operation”

means every operation takes constant time

“The total worst-case time complexity of performing any \(n\) operations is \(O(n)\)”

this does not imply that every operation takes constant time

However, it does mean that the \text{amortised worst-case} time complexity of an operation is \(O(1)\)
Dynamic perfect hashing

- A **dynamic dictionary** stores \((key, value)\)-pairs and supports:
  - \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

**Theorem**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes **amortised expected** \(O(1)\) time

What does **amortised expected** \(O(1)\) time mean?! **let’s build it up…**

“\(O(1)\) **expected** time per operation”

means every operation takes constant time **in expectation**

“The total **expected** time complexity of performing any \(n\) operations is \(O(n)\)”

this **does not** imply that every operation takes constant time **in expectation**

However, it **does mean** that the **amortised expected** time complexity of an operation is \(O(1)\)
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value) and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

---

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((\text{key, value})\)-pairs and supports:
  - \(\text{add(key, value)}\), \(\text{lookup(key)}\) (which returns \text{value}) and \(\text{delete(key)}\)

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
A dynamic dictionary stores \((key, value)\)-pairs and supports:

- `add(key, value)`, `lookup(key)` (which returns `value`) and `delete(key)`

---

**THEOREM**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes *amortised expected* \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:

  \[
  \text{add}(key, value), \text{lookup}(key) \quad \text{(which returns value)} \quad \text{and delete}(key)
  \]

---

**Theorem**

In the Cuckoo hashing scheme:

- Every \text{lookup} and every \text{delete} takes \(O(1)\) \text{ worst-case} time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An \text{insert} takes \text{amortised expected} \(O(1)\) \text{ time}

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).
Dynamic perfect hashing

- A dynamic dictionary stores $(key, value)$-pairs and supports:
  - add($key, value$), lookup($key$) (which returns $value$) and delete($key$)

**Theorem**

In the Cuckoo hashing scheme:
- Every lookup and every delete takes $O(1)$ worst-case time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An insert takes amortised expected $O(1)$ time

In Cuckoo hashing there is a single hash table but two hash functions: $h_1$ and $h_2$.

Each key in the table is either stored at position $h_1(x)$ or $h_2(x)$. 
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  - add\((key, value)\), lookup\((key)\) (which returns \(value\)) and delete\((key)\)

**Theorem**

In the Cuckoo hashing scheme:
- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

**Important:** We never store multiple keys at the same position
Dynamic perfect hashing

- A dynamic dictionary stores \((key, value)\)-pairs and supports:
  \[
  \text{add}(key, value), \text{lookup}(key) \text{ (which returns value)} \text{ and delete}(key)
  \]

**Theorem**
In the Cuckoo hashing scheme:
- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An insert takes amortised expected \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

Important: We never store multiple keys at the same position

Therefore, as claimed, lookup takes \(O(1)\) time...
Dynamic perfect hashing

A dynamic dictionary stores \((key, value)\)-pairs and supports:

\[ \text{add}(key, value), \text{lookup}(key) (\text{which returns } value) \text{ and } \text{delete}(key) \]

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored,
- An insert takes \textit{amortised expected } \(O(1)\) time

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

**Important:** We never store multiple keys at the same position

Therefore, as claimed, lookup takes \(O(1)\) time... but how do we do inserts?
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop (and congratulate yourself on a job well done)
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
evict key $y$ and replace it with key $x$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
   \textit{if that position is empty, stop}

Step 2: Let $y$ be the key currently in position $h_1(x)$
   evict key $y$ and replace it with key $x$
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

*where should we put key $y$?*
Inserts in Cuckoo hashing

**Step 1**: Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2**: Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

*where should we put key $y$?*

in the *other* position it’s allowed in
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
   if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
   evict key $y$ and replace it with key $x$

where should we put key $y$?
   in the other position it’s allowed in
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$  
*if that position is empty, stop*

Step 2: Let $y$ be the key currently in position $h_1(x)$  
evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the *other* position $y$ is allowed to be in  
*i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise*
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$

if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

Step 3: Let pos be the other position $y$ is allowed to be in

i.e pos = $h_2(y)$ if $h_1(x) = h_1(y)$ and pos = $h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position pos

if that position is empty, stop
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

**Step 3:** Let $pos$ be the *other* position $y$ is allowed to be in

*i.e pos = $h_2(y)$ if $h_1(x) = h_1(y)$ and pos = $h_1(y)$ otherwise*

**Step 4:** Attempt to put $y$ in position $pos$

*if that position is empty, stop*
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

**Step 3:** Let $\text{pos}$ be the *other* position $y$ is allowed to be in

* i.e $\text{pos} = h_2(y)$ if $h_1(x) = h_1(y)$ and $\text{pos} = h_1(y)$ otherwise

**Step 4:** Attempt to put $y$ in position $\text{pos}$

*if that position is empty, stop*
Inserts in Cuckoo hashing

**Step 1:** Attempt to put $x$ in position $h_1(x)$
if that position is empty, stop

**Step 2:** Let $y$ be the key currently in position $h_1(x)$
evict key $y$ and replace it with key $x$

**Step 3:** Let $pos$ be the other position $y$ is allowed to be in
i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

**Step 4:** Attempt to put $y$ in position $pos$
if that position is empty, stop

**Step 5:** Let $z$ be the key currently in position $pos$
evict key $z$ and replace it with key $y$
**Inserts in Cuckoo hashing**

**Step 1:** Attempt to put $x$ in position $h_1(x)$

*if that position is empty, stop*

**Step 2:** Let $y$ be the key currently in position $h_1(x)$

evict key $y$ and replace it with key $x$

**Step 3:** Let $pos$ be the *other* position $y$ is allowed to be in

*i.e $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise*

**Step 4:** Attempt to put $y$ in position $pos$

*if that position is empty, stop*

**Step 5:** Let $z$ be the key currently in position $pos$

evict key $z$ and replace it with key $y$
Inserts in Cuckoo hashing

Step 1: Attempt to put $x$ in position $h_1(x)$
   if that position is empty, stop

Step 2: Let $y$ be the key currently in position $h_1(x)$
   evict key $y$ and replace it with key $x$

Step 3: Let $pos$ be the other position $y$ is allowed to be in
   i.e. $pos = h_2(y)$ if $h_1(x) = h_1(y)$ and $pos = h_1(y)$ otherwise

Step 4: Attempt to put $y$ in position $pos$
   if that position is empty, stop

Step 5: Let $z$ be the key currently in position $pos$
   evict key $z$ and replace it with key $y$
Inserts in Cuckoo hashing

Step 1: Attempt to put \( x \) in position \( h_1(x) \)
if that position is empty, stop

Step 2: Let \( y \) be the key currently in position \( h_1(x) \)
evict key \( y \) and replace it with key \( x \)

Step 3: Let \( pos \) be the other position \( y \) is allowed to be in
i.e \( pos = h_2(y) \) if \( h_1(x) = h_1(y) \) and \( pos = h_1(y) \) otherwise

Step 4: Attempt to put \( y \) in position \( pos \)
if that position is empty, stop

Step 5: Let \( z \) be the key currently in position \( pos \)
evict key \( z \) and replace it with key \( y \) and so on...
Pseudocode

**add**(*x*):

- `pos ← h₁(*x*)`

  Repeat at most *n* times:

  - If *T*[*pos*] is empty then *T*[*pos*] ← *x*.
  - Otherwise,
    
    
    \[
    \begin{align*}
    y & \leftarrow T[\text{pos}], \\
    T[\text{pos}] & \leftarrow x,
    \end{align*}
    \]

    `pos ←` the other possible location for *y*.

    (i.e. if *y* was evicted from *h₁(*y*)* then *pos* ← *h₂(*y*), otherwise *pos* ← *h₁(*y*)).

    `x ← y`.

  Repeat

- Give up and rehash the whole table.

  *i.e. empty the table, pick two new hash functions and reinsert every key*
Rehashing

If we fail to insert a new key $x$, (i.e. we still have an “evicted” key after moving around keys $n$ times) then we declare the table “rubbish” and rehash.
Rehashing

If we fail to insert a new key \( x \),

\[
\text{(i.e. we still have an “evicted” key after moving around keys } n \text{ times)}
\]

then we declare the table “rubbish” and rehash.

What does rehashing involve?
Rehashing

If we fail to insert a new key \( x \),

(i.e. we still have an “evicted” key after moving around keys \( n \) times)

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \)

at the time of we fail to insert key \( x \).
Rehashing

If we fail to insert a new key $x$,

(i.e. we still have an “evicted” key after moving around keys $n$ times)

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$

at the time of we fail to insert key $x$.

To rehash we:
Rehashing

If we fail to insert a new key \( x \),

\[
\text{(i.e. we still have an “evicted” key after moving around keys } n \text{ times)}
\]

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \)

at the time of we fail to insert key \( x \).

To rehash we:

Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)
Rehashing

If we fail to insert a new key $x$,

\[ (i.e. \text{ we still have an "evicted" key after moving around keys } n \text{ times}) \]
then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$
at the time of we fail to insert key $x$.

To rehash we:

Randomly pick two new hash functions $h_1$ and $h_2$. (More about this in a minute.)

Build a new empty hash table of the same size
Rehashing

If we fail to insert a new key $x$,

\textit{(i.e. we still have an “evicted” key after moving around keys $n$ times)}

then we declare the table “rubbish” and rehash.

\textbf{What does rehashing involve?}

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$

at the time of we fail to insert key $x$.

\textbf{To rehash we:}

Randomly pick two new hash functions $h_1$ and $h_2$. (More about this in a minute.)

Build a \textit{new} empty hash table of the same size

\textit{Reinsert} the keys $x_1, \ldots, x_k$ and then $x$,

\textit{one by one, using the normal add operation.}
Rehashing

If we fail to insert a new key $x$,

\[(i.e. \text{ we still have an “evicted” key after moving around keys } n \text{ times})\]

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$ at the time of we fail to insert key $x$.

To rehash we:

Randomly pick two new hash functions $h_1$ and $h_2$. (More about this in a minute.)

Build a new empty hash table of the same size

Reinsert the keys $x_1, \ldots, x_k$ and then $x$,

\[\text{one by one, using the normal add operation.}\]

If we fail while rehashing...we start from the beginning again
Rehashing

If we fail to insert a new key $x$,

\[ (i.e. \text{we still have an "evicted" key after moving around keys } n \text{ times}) \]

then we declare the table "rubbish" and rehash.

What does rehashing involve?

Suppose that the table contains the $k$ keys $x_1, \ldots, x_k$

at the time of we fail to insert key $x$.

To rehash we:

Randomly pick two new hash functions $h_1$ and $h_2$. (More about this in a minute.)

Build a new empty hash table of the same size

Reinsert the keys $x_1, \ldots, x_k$ and then $x$,

one by one, using the normal add operation.

If we fail while rehashing... we start from the beginning again

This is rather slow... but we will prove that it happens rarely
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (*see the link on unit web page*).
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:

\[ h_1 \text{ and } h_2 \text{ are independent} \]

i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- $h_1$ and $h_2$ are independent
  - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are truly random
  - i.e. each key is independently mapped to each of the $m$ positions in the hash table with probability $\frac{1}{m}$. 
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page).*

We make the following assumptions:

- \( h_1 \) and \( h_2 \) are truly random
  i.e. each key is independently mapped to each of the \( m \) positions in the hash table with probability \( \frac{1}{m} \).

- \( h_1 \) and \( h_2 \) are independent
  i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.

Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:

- $h_1$ and $h_2$ are independent
  - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are truly random
  - i.e. each key is independently mapped to each of the $m$ positions in the hash table with probability $\frac{1}{m}$.

Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

*There are at most $n$ keys in the hash table at any time.*
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- $h_1$ and $h_2$ are truly random
  - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are independent
  - i.e. each key is independently mapped to each of the $m$ positions in the hash table with probability $\frac{1}{m}$.

Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

*There are at most $n$ keys in the hash table at any time.*
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh *(see the link on unit web page)*.

We make the following assumptions:

- \( h_1 \) and \( h_2 \) are truly random
  - i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.

- \( h_1 \) and \( h_2 \) are independent
  - i.e. each key is independently mapped to each of the \( m \) positions in the hash table with probability \( \frac{1}{m} \).

Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time

*There are at most \( n \) keys in the hash table at any time.*
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- $h_1$ and $h_2$ are truly random
  - i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are independent
  - i.e. each key is independently mapped to each of the $m$ positions in the hash table with probability $\frac{1}{m}$.

- Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

There are at most $n$ keys in the hash table at any time.
Assumptions

We will follow the analysis in the paper *Cuckoo hashing for undergraduates*, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- $h_1$ and $h_2$ are independent
  i.e. $h_1(x)$ says nothing about $h_2(x)$, and vice versa.

- $h_1$ and $h_2$ are truly random
  i.e. each key is independently mapped to each of the $m$ positions in the hash table with probability $\frac{1}{m}$.

- Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

*There are at most $n$ keys in the hash table at any time.*
Cuckoo graph

Hash table
(size $m$)
Cuckoo graph

Hash table (size $m$)

The **cuckoo graph**: 
Cuckoo graph

Hash table (size $m$)

The **cuckoo graph**: A vertex for each position of the table.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.
The **cuckoo graph**:  
A vertex for each position of the table.  
For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

Hash table (size $m$) 

$m$ vertices
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 
The **cuckoo graph**:

- A vertex for each position of the table.
- For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

Cuckoo graph

Hash table (size $m$)
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

- $h_1(x_1)$
- $h_2(x_1)$
- $x_1$
- $x_2$
- $x_3$

$\text{m vertices}$
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

---

**Cuckoo graph**

Hash table (size $m$)

$m$ vertices

The key $x_1$ has two hash functions $h_1(x_1)$ and $h_2(x_1)$.

- $h_1(x_1)$ points to $x_1$.
- $h_2(x_1)$ points to $x_4$.

The graph shows the connections between the keys and their hash function positions.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

---

**Diagram**: Hash table (size $m$)

- $x_1$
- $x_2$
- $x_3$
- $x_4$
The cuckoo graph:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.
The **cuckoo graph**: A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

A hash table (size $m$) with $m$ vertices.
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

There is no space for $x_5$...
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

There is no space for $x_5$... so we make space by moving $x_2$ and then $x_3$. 

There is an undirected edge between $h_1(x)$ and $h_2(x)$. 

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

There is no space for $x_5\ldots$ so we make space by moving $x_2$ and then $x_3$. 

A hash table (size $m$) with $m$ vertices:

- $x_2$ connects to $h_1(x_5)$
- $x_3$ connects to $h_2(x_5)$
- $x_1$ connects to $x_2$
- $x_4$ connects to $x_1$
The **cuckoo graph**: 
A vertex for each position of the table.
For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

There is no space for $x_5$... so we make space by moving $x_2$ and then $x_3$. 

**Cuckoo graph**

- Hash table (size $m$)
- $x_4$, $x_3$, $x_2$, $x_1$, $x_5$
- $h_1(x_5)$, $h_2(x_5)$
- $m$ vertices
The **cuckoo graph**:  

A vertex for each position of the table. 

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$. 

There is no space for $x_5$... 

so we make space 

by moving $x_2$ and then $x_3$. 

**Hash table** 
(size $m$) 

$m$ vertices
Cuckoo graph

The **cuckoo graph**:

- A vertex for each position of the table.
- For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

There is no space for $x_5$... so we make space by moving $x_2$ and then $x_3$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.
The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.
The **cuckoo graph**: 

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*
The **cuckoo graph**: 

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key \(x\) there is an undirected edge between \(h_1(x)\) and \(h_2(x)\).

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*

Inserting key \(x_6\) creates a cycle.
Cuckoo graph

The **cuckoo graph**:

- A vertex for each position of the table.
- For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting key $x_6$ creates a cycle.

*Cycles are dangerous...*
The **cuckoo graph**:  

A vertex for each position of the table.  

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.  

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.  

Inserting key $x_6$ creates a cycle.  

*Cycles are dangerous…*
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting key $x_6$ creates a cycle.

*Cycles are dangerous...*

When key $x_7$ is inserted where does it go?
Cuckoo graph

The **cuckoo graph**:  
A vertex for each position of the table.  
For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.  

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.*

Inserting key $x_6$ creates a cycle.

*Cycles are dangerous.*

When key $x_7$ is inserted where does it go?  
*there are 6 keys but only 5 spaces*
Cuckoo graph

The **cuckoo graph**:

- A vertex for each position of the table.
- For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.*

Inserting key $x_6$ creates a cycle.

*Cycles are dangerous...*

When key $x_7$ is inserted where does it go?

*there are 6 keys but only 5 spaces*

The keys would be moved around in an infinite loop but we stop and rehash after $n$ moves...
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.*

Inserting key $x_6$ creates a cycle.

_Cycles are dangerous…_

When key $x_7$ is inserted where does it go?

*there are 6 keys but only 5 spaces*

The keys would be moved around in an infinite loop but we stop and rehash after $n$ moves…

Inserting a key into a cycle **always** causes a rehash
The cuckoo graph:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting a key into a cycle always causes a rehash.
Cuckoo graph

The **cuckoo graph**: 

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

*The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph*

Inserting a key into a cycle **always** causes a rehash.
Cuckoo graph

The **cuckoo graph**:

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting a key into a cycle **always** causes a rehash.

*This is the only way a rehash can happen*
The **cuckoo graph**:  

A vertex for each position of the table.

For each key $x$ there is an undirected edge between $h_1(x)$ and $h_2(x)$.

The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph.

Inserting a key into a cycle **always** causes a rehash. This is the only way a rehash can happen.

We will analyse the probability of either a cycle or a long path occurring in the graph while inserting any $n$ keys.
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$. 

**Lemma**

The table size is $m$ and there are $n$ keys.
LEMMMA

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

What does this say?
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$.

(let $c = 2$ for simplicity)
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

(let $c = 2$ for simplicity)
Paths in the cuckoo graph

**Lemma**

For any positions \( i \) and \( j \), and any constant \( c > 1 \), if \( m \geq 2cn \) then the probability that there exists a shortest path in the cuckoo graph from \( i \) to \( j \) with length \( \ell \geq 1 \), is at most \( \frac{1}{c^\ell \cdot m} \).

**What does this say?**

(let \( c = 2 \) for simplicity)

Probability of a shortest path of length 1 is at most \( \frac{1}{2 \cdot m} \)
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

What does this say?

Probability of a shortest path of length 2 is at most $\frac{1}{4 \cdot m}$

(let $c = 2$ for simplicity)
Lemma

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

What does this say?

(let $c = 2$ for simplicity)

Probability of a shortest path of length $3$ is at most $\frac{1}{8 \cdot m}$
Paths in the cuckoo graph

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$.

What does this say?

Probability of a shortest path of length $4$ is at most $\frac{1}{16 \cdot m}$

(let $c = 2$ for simplicity)
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$.

What does this say?

Probability of a shortest path of length 4 is at most $\frac{1}{16 \cdot m}$

How likely is it that there even is a path?
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$.

(LET $c = 2$ FOR SIMPLICITY)

**What does this say?**

**How likely is it that there even is a path?**
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

*(let $c = 2$ for simplicity)*

**How likely is it that there even is a path?**

If a path exists from $i$ to $j$, there must be a shortest path (from $i$ to $j$)
Paths in the cuckoo graph

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

*What does this say?*

*How likely is it that there even is a path?*

If a path exists from $i$ to $j$, there must be a shortest path (from $i$ to $j$)

Therefore the probability of a path from $i$ to $j$ existing is at most...

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

*(using the union bound over all possible path lengths.)*
**Lemma**

For any positions \(i\) and \(j\), and any constant \(c > 1\), if \(m \geq 2cn\) then the probability that there exists a shortest path in the cuckoo graph from \(i\) to \(j\) with length \(\ell \geq 1\), is at most \(\frac{1}{c^\ell \cdot m}\).

*(let \(c = 2\) for simplicity)*

**What does this say?**

**How likely is it that there even is a path?**

If a path exists from \(i\) to \(j\), there must be a shortest path (from \(i\) to \(j\))

Therefore the probability of a path from \(i\) to \(j\) existing is at most…

\[
\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m} \sum_{\ell=1}^{\infty} \frac{1}{c^\ell}
\]

*(using the union bound over all possible path lengths.)*
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

*(let $c = 2$ for simplicity)*

**What does this say?**

**How likely is it that there even is a path?**

If a path exists from $i$ to $j$, there must be a shortest path (from $i$ to $j$)

Therefore the probability of a path from $i$ to $j$ existing is at most . . .

$$
\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m} \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{1}{m \cdot (c-1)} = \frac{1}{m}
$$

*(using the union bound over all possible path lengths.)*
Paths in the cuckoo graph

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

*(let $c = 2$ for simplicity)*

What does this say?

**How likely is it that there even is a path?**

If a path exists from $i$ to $j$, there must be a shortest path (from $i$ to $j$)

Therefore the probability of a path from $i$ to $j$ existing is at most...

$$
\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m} \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{1}{m \cdot (c-1)} = \frac{1}{m}
$$

*(using the union bound over all possible path lengths.)*

So a path from $i$ to $j$ is rather unlikely to exist
Paths in the cuckoo graph

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^{\ell} \cdot m}$.

*What is the proof?*
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**What is the proof?**

The proof is in the directors cut of the slides (see notes)
For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**What is the proof?**

The proof is in the directors cut of the slides (see notes)

**Can we at least see the pictures?**
**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

**What is the proof?**

The proof is in the directors cut of the slides (see notes)

**Can we at least see the pictures?**

The proof is by induction on the length $\ell$:

**Base case: $\ell = 1$.**

Argue that each key has prob $\frac{2}{m^2}$ to create an edge $(i, j)$

Union bound over all $n$ keys

**Inductive step:**

Pick a third point $k$ to split the path

Union bound over all $k$ then all keys
A dynamic dictionary stores \((key, value)\)-pairs and supports:

- \(\text{add}(key, value)\), \(\text{lookup}(key)\) (which returns \(value\)) and \(\text{delete}(key)\)

Universe \(U\) of \(u\) keys.

Hash table \(T\) of size \(m \geq n\).

Collisions are fixed by bucketing.

Locating the bucket containing a given key takes \(O(1)\) time.

We require that we can recover any key from its bucket in \(O(s)\) time, where \(s\) is the number of keys in the bucket.

\(n\) arbitrary operations arrive online, one at a time.

If our construction has the property that, for any two keys \(x, y \in U\) (with \(x \neq y\)),
the probability that \(x\) and \(y\) are in the same bucket is \(O\left(\frac{1}{m}\right)\)

For any \(n\) operations, the \(\text{expected}\) run-time is \(O(1)\) per operation.
Hash table

We say that two keys $x, y$ are in the same bucket (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

Don’t put all your eggs in one bucket

Table size is $m$ keys $n$
We say that two keys $x, y$ are in the same **bucket** (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x, y$, the probability that they are in the same bucket is at most

$$\sum_{\ell=1}^{\infty} \frac{4}{c^\ell \cdot m} = \frac{4}{m} \cdot \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{4}{m(c - 1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

*(another union bound over all possible path lengths.)*
Don’t put all your eggs in one bucket

Hash table

We say that two keys \( x, y \) are in the same **bucket** (conceptually) iff there is a path between \( h_1(x) \) and \( h_1(y) \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is at most

\[
\sum_{\ell=1}^{\infty} \frac{4}{c^\ell \cdot m} = 4 \cdot \frac{\sum_{\ell=1}^{\infty} 1}{m(c-1)} = O\left(\frac{1}{m}\right)
\]

where \( c > 1 \) is a constant.

(Another union bound over all possible path lengths.)

**Lemma**

For any positions \( i \) and \( j \), and any constant \( c > 1 \), if \( m \geq 2cn \) then the probability that there exists a shortest path in the cuckoo graph from \( i \) to \( j \) with length \( \ell \geq 1 \), is at most \( \frac{1}{c^\ell \cdot m} \).
We say that two keys \( x, y \) are in the same **bucket** (conceptually) iff there is a path between \( h_1(x) \) and \( h_1(y) \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is at most

\[
\sum_{\ell=1}^{\infty} \frac{4}{c^\ell \cdot m} = \frac{4}{m} \cdot \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{4}{m(c - 1)} = O\left(\frac{1}{m}\right)
\]

where \( c > 1 \) is a constant.

*(another union bound over all possible path lengths.)*
We say that two keys $x$, $y$ are in the same **bucket** (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x$, $y$, the probability that they are in the same bucket is at most

$$\sum_{\ell=1}^{\infty} \frac{4}{c^\ell \cdot m} = \frac{4}{m} \cdot \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)$$

where $c > 1$ is a constant.

(Another union bound over all possible path lengths.)

The time for an operation on $x$ is bounded by the number of items in the bucket. *(assuming there are no cycles.)*
Don’t put all your eggs in one bucket

We say that two keys $x, \ y$ are in the same bucket (conceptually) iff there is a path between $h_1(x)$ and $h_1(y)$ in the cuckoo graph.

For two distinct keys $x, \ y$, the probability that they are in the same bucket is at most

$$
\sum_{\ell=1}^{\infty} \frac{4}{c^{\ell} \cdot m} = \frac{4}{m} \cdot \sum_{\ell=1}^{\infty} \frac{1}{c^{\ell}} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)
$$

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

The time for an operation on $x$ is bounded by the number of items in the bucket. (assuming there are no cycles.)

So we have that the expected time per operation is $O(1)$ (assuming that $m \geq 2cn$ and there are no cycles).
We say that two keys \( x, y \) are in the same bucket (conceptually) iff there is a path between \( h_1(x) \) and \( h_1(y) \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is at most

\[
\sum_{\ell=1}^{\infty} \frac{4}{c^\ell \cdot m} = 4 \cdot \sum_{\ell=1}^{\infty} \frac{1}{c^\ell} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)
\]

where \( c > 1 \) is a constant. (another union bound over all possible path lengths.)

The time for an operation on \( x \) is bounded by the number of items in the bucket. (assuming there are no cycles.)

So we have that the expected time per operation is \( O(1) \) (assuming that \( m \geq 2cn \) and there are no cycles).

Further, lookups take \( O(1) \) time in the worst case.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*. However, we would expect there to be cycles every now and then, causing a rehash.
Rehashing

The previous analysis on the expected running time holds when there are no cycles. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting \( n \) keys into the table...
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting $n$ keys into the table...

A cycle is a path from a vertex $i$ back to itself.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting $n$ keys into the table...

A cycle is a path from a vertex $i$ back to itself.

so use previous result with $i = j$...
Rehashing

The previous analysis on the expected running time holds when there are no cycles. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting $n$ keys into the table...

A cycle is a path from a vertex $i$ back to itself.

so use previous result with $i = j$...

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.
Rehashing

The previous analysis on the expected running time holds when there are *no cycles*. However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting $n$ keys into the table...

A cycle is a path from a vertex $i$ back to itself.

so use previous result with $i = j$...

**Lemma**

For any positions $i$ and $j$, and any constant $c > 1$, if $m \geq 2cn$ then the probability that there exists a shortest path in the cuckoo graph from $i$ to $j$ with length $\ell \geq 1$, is at most $\frac{1}{c^\ell \cdot m}$.

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$

*another union bound over all possible path lengths.*)
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$  

(another union bound over all possible path lengths.)
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$  

(another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.$$
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}.$$  
(Another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$  
(Union bound over all $m$ positions in the table.)
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^{\ell} \cdot m} = \frac{1}{m(c - 1)}.$$

(Another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.$$

(Union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs (that there is a rehash) during the $n$ insertions.
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}.$$  

(another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$  

(union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs

(that there is a rehash) during the $n$ insertions.

The probability that there are two rehashes is $\frac{1}{4}$, and so on.
Rehashing

The probability that a position $i$ is involved in a cycle is at most

$$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}.$$  

(Another union bound over all possible path lengths.)

The probability that there is at least one cycle is at most

$$m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}.$$  

(Union bound over all $m$ positions in the table.)

If we set $c = 3$, the probability is at most $\frac{1}{2}$ that a cycle occurs (that there is a rehash) during the $n$ insertions.

The probability that there are two rehashes is $\frac{1}{4}$, and so on.

So the expected number of rehashes during $n$ insertions is at most $\sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i = 1$. 
Rehashing

If the expected time for one rehash is \( O(n) \) then

the expected time for all rehashes is also \( O(n) \)

*(this is because we only expect there to be one rehash).*
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).
Rehashing

If the expected time for one rehash is $O(n)$ then
the expected time for all rehashes is also $O(n)$
*(this is because we only expect there to be one rehash)*.

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is
$O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the **amortised expected** time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph using the at most $n$ keys.
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

*(this is because we only expect there to be one rehash).*

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph

using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)

*(you can do this using breadth-first search)*
Rehashing

If the expected time for one rehash is $O(n)$ then

the expected time for all rehashes is also $O(n)$

(this is because we only expect there to be one rehash).

Therefore the *amortised expected* time for the rehashes over the $n$ insertions is $O(1)$ per insertion (i.e. divide the total cost with $n$).

Why is the expected time per rehash $O(n)$?

First pick a new random $h_1$ and $h_2$ and construct the cuckoo graph using the at most $n$ keys.

Check for a cycle in the graph in $O(n)$ time (and start again if you find one)

(you can do this using breadth-first search)

If there is no cycle, insert all the elements,

this takes $O(n)$ time in expectation (as we have seen).
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic where any two keys $x, y$ are independent.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys $x, y$ are independent

A set $H$ of hash functions is **weakly universal** if for any two distinct keys $x, y \in U$,

$$\Pr(h(x) = h(y)) \leq \frac{1}{m} \quad \text{(where } h \text{ is picked uniformly at random from } H)$$
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic where any two keys \(x, y\) are independent.

A set \(H\) of hash functions is **weakly universal** if for any two distinct keys \(x, y \in U\),

\[
\Pr (h(x) = h(y)) \leq \frac{1}{m}
\]  (where \(h\) is picked uniformly at random from \(H\))

We can define a stronger hash families with \(k\)-wise independence. here the hash values of any choice of \(k\) keys are independent.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys \( x, y \) are independent

A set \( H \) of hash functions is **weakly universal** if for any two distinct keys \( x, y \in U \),

\[
\Pr \left( h(x) = h(y) \right) \leq \frac{1}{m} \quad (\text{where } h \text{ is picked uniformly at random from } H)
\]

We can define a stronger hash families **with \( k \)-wise independence.**

here the hash values of any choice of \( k \) keys are independent.

A set \( H \) of hash functions is **\( k \)-wise independent** if

for any \( k \) distinct keys \( x_1, x_2 \ldots x_k \in U \) and \( k \) values \( v_1, v_2, \ldots v_k \in \{0, 1, 2 \ldots m - 1\} \),

\[
\Pr \left( \bigcap_{i} h(x_i) = v_i \right) = \frac{1}{m^k}
\]

(where \( h \) is picked uniformly at random from \( H \))
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys \( x, y \) are independent

We can define a stronger hash families with \( k \)-wise independence.

here the hash values of any choice of \( k \) keys are independent.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys $x, y$ are independent

We can define a stronger hash families with $k$-wise independence.

here the hash values of any choice of $k$ keys are independent.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic

where any two keys $x, y$ are independent

We can define a stronger hash families with $k$-wise independence.

here the hash values of any choice of $k$ keys are independent.

It is feasible to construct a $(\log n)$-wise independent family of hash functions

such that $h(x)$ can be computed in $O(1)$ time
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic where any two keys \( x, y \) are independent.

We can define a stronger hash families with \( k \)-wise independence.

here the hash values of any choice of \( k \) keys are independent.

It is feasible to construct a \((\log n)\)-wise independent family of hash functions such that \( h(x) \) can be computed in \( O(1) \) time.

By changing the cuckoo hashing algorithm to perform a rehash after \( \log n \) moves it can be shown (via a similar but harder proof) that the results still hold.
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic, where any two keys $x, y$ are independent.

We can define a stronger hash families \textit{with $k$-wise independence}. here the hash values of any choice of $k$ keys are independent.

It is feasible to construct a $(\log n)$-wise independent family of hash functions such that $h(x)$ can be computed in $O(1)$ time.

By changing the cuckoo hashing algorithm to perform a rehash after $\log n$ moves it can be shown (via a similar but harder proof) that the results still hold.

\textbf{THEOREM} In the Cuckoo hashing scheme:

- Every lookup and every delete takes $O(1)$ worst-case time,
- The space is $O(n)$ where $n$ is the number of keys stored
- An insert takes \textit{amortised expected} $O(1)$ time