Advanced Algorithms – COMS31900

2013/2014

Lecture 4
Cuckoo hashing

Markus Jalsenius
Previously on COMS31900...

Hashing with chaining

**Observe**

- For any two distinct keys $x$ and $y$, $Pr \left( h(x) = h(y) \right)$ is $O\left( \frac{1}{m} \right)$.
- The time for an operation on key $x$ is bounded by the number of items at position $h(x)$.
- The expected time per operation is $O(1)$ ($m \geq n$).

This is what we showed last week.
Previously on COMS31900... 

Hashing with chaining

Instead of a linked list, throw colliding elements into a *bucket*!

**Observe**

- For any two distinct keys $x$ and $y$, $\Pr (h(x) = h(y))$ is $O\left(\frac{1}{m}\right)$.
- The time for an operation on key $x$ is bounded by the number of items at position $h(x)$.
- The expected time per operation is $O(1)$ ($m \geq n$).

This is what we showed last week.
Previously on COMS31900...

Hashing with chaining

Instead of a linked list, throw colliding elements into a *bucket*!

- If, for distinct $x, y$, the probability they are in the same bucket is $O\left(\frac{1}{m}\right)$,
- the time for an operation on $x$ is bounded by the number of items in its bucket,
- then the expected time per operation is $O(1)$ ($m \geq n$).

---

**Observe**

- For any two distinct keys $x$ and $y$, $\Pr(h(x) = h(y))$ is $O\left(\frac{1}{m}\right)$.
- The time for an operation on key $x$ is bounded by the number of items at position $h(x)$.
- The expected time per operation is $O(1)$ ($m \geq n$).

This is what we showed last week.
Previously on COMS31900...

Hashing with chaining

Instead of a linked list, throw colliding elements into a bucket!

We will describe a rather unusual type of bucket in this lecture.

Observe

- If, for distinct $x, y$, the probability they are in the same bucket is $O\left(\frac{1}{m}\right)$,
- the time for an operation on $x$ is bounded by the number of items in its bucket,
- then the expected time per operation is $O(1)$ ($m \geq n$).

Observe

- For any two distinct keys $x$ and $y$, $\Pr(h(x) = h(y))$ is $O\left(\frac{1}{m}\right)$.
- The time for an operation on key $x$ is bounded by the number of items at position $h(x)$.
- The expected time per operation is $O(1)$ ($m \geq n$).

This is what we showed last week.
Dynamic perfect hashing

We want:

- \( O(1) \) worst case lookup time (like with static perfect hashing).
- No static keys (i.e. we do not know the keys in advance).
- Good expected performance for insertions.
Dynamic perfect hashing

We want:

- \(O(1)\) worst case lookup time (like with static perfect hashing).
- No static keys (i.e. we do not know the keys in advance).
- Good expected performance for insertions.

**Cuckoo hashing** is the answer:

- Two hash functions: \(h_1\) and \(h_2\).
- Key \(x\) is stored at either position \(h_1(x)\) or \(h_2(x)\).
- At most one key per position in the hash table (i.e. no chaining).
Dynamic perfect hashing

We want:

- $O(1)$ worst case lookup time (like with static perfect hashing).
- No static keys (i.e. we do not know the keys in advance).
- Good expected performance for insertions.

**Cuckoo hashing** is the answer:

- **Two** hash functions: $h_1$ and $h_2$.
- Key $x$ is stored at either position $h_1(x)$ or $h_2(x)$.
- At most one key per position in the hash table (i.e. no chaining).
- Looking up a key $x$ always takes $O(1)$ time; check if the key is at either $h_1(x)$ or $h_2(x)$.
- Removing a key is also constant time.
- Adding a key could take more time...
Cuckoo hashing

- When adding a new key $x$, add it to $h_1(x)$ if that position is empty.
Cuckoo hashing

- When adding a new key $x$, add it to $h_1(x)$ if that position is empty.
- If $h_1(x)$ is **not empty**, then there is another key $y$ there already.
Cuckoo hashing

- When adding a new key \( x \), add it to \( h_1(x) \) if that position is empty.
- If \( h_1(x) \) is *not empty*, then there is another key \( y \) there already.
- Replace \( y \) with \( x \) and reinsert \( y \) at its other position (i.e. \( h_1(y) \) or \( h_2(y) \)).
Cuckoo hashing

- When adding a new key $x$, add it to $h_1(x)$ if that position is empty.
- If $h_1(x)$ is *not empty*, then there is another key $y$ there already.
- Replace $y$ with $x$ and reinsert $y$ at its other position (i.e. $h_1(y)$ or $h_2(y)$).
- Repeat by relocating other keys if necessary.
Pseudocode

\textbf{add}(x):

\begin{itemize}
  \item \textbf{pos} $\leftarrow h_1(x)$
  \item \textbf{loop} \textit{n} times:
    \begin{itemize}
      \item If \( T[\text{pos}] \) is empty then \( T[\text{pos}] \leftarrow x \). \textbf{Done!}
      \item Otherwise,
        \begin{itemize}
          \item \( y \leftarrow T[\text{pos}] \),
          \item \( T[\text{pos}] \leftarrow x \),
          \item \( \text{pos} \leftarrow \text{the other possible location for } y. \)
        \end{itemize}
        (i.e. if \( y \) was evicted from \( h_1(y) \) then \( \text{pos} \leftarrow h_2(y) \), otherwise \( \text{pos} \leftarrow h_1(y) \).)
      \item \( x \leftarrow y \).
    \end{itemize}
  \end{itemize}
  \item Repeat (at most \textit{n} times).
  \item Rehash the whole table, then make a new attempt to add \textit{x}.
\end{itemize}
Rehashing

- If we fail to insert a new key $x$ (i.e. we still have a “free” key that has to go back into the table after moving around keys $n$ times) then we declare the table “rubbish” and rehash.
Rehashing

- If we fail to insert a new key \( x \) (i.e. we still have a “free” key that has to go back into the table after moving around keys \( n \) times) then we declare the table “rubbish” and rehash.
- Suppose the table contains the \( k \) keys \( x_1, \ldots, x_k \) at the time of insertion of key \( x \).
- Rehashing means:
  - 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 \( m \).
  - Reinsert the keys \( x_1, \ldots, x_k \).
  - If we fail to insert these \( k \) keys, scrap the hash table and construct a new one with new hash functions and try again.
Rehashing

- If we fail to insert a new key $x$ (i.e. we still have a “free” key that has to go back into the table after moving around keys $n$ times) then we declare the table “rubbish” and rehash.
- Suppose the table contains the $k$ keys $x_1, \ldots, x_k$ at the time of insertion of key $x$.
- Rehashing means:
  - 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 $m$.
  - Reinsert the keys $x_1, \ldots, x_k$.
  - If we fail to insert these $k$ keys, scrap the hash table and construct a new one with new hash functions and try again.
- Now try to insert $x$ again.
- If we fail, rehash and try to insert $x$ again. Repeat until it succeeds.
Assumptions

- In the following we will analyse the running time of this hashing scheme.
- We will follow the analysis presented in the paper *Cuckoo hashing for undergraduates*, 2006, by Pagh (see link on unit web page).
Assumptions

In the following we will analyse the running time of this hashing scheme.

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

We make the following assumptions:

- $h_1$ and $h_2$ are truly random, i.e. a key is mapped to a particular position in the hash table with probability $1/m$.

  **Observe**
  
  True randomness is not feasible, so similarly to weakly universal hashing, we will use a property that, for our purposes, is like true randomness.

- $h_1$ 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 constant time (not necessarily true, but more about this later.)

- At most $n$ keys are stored simultaneously in the hash table.
Cuckoo graph

Hash table
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 $h_2(x)$. 

$m$ 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 $h_2(x)$. 

$m$ 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 $h_2(x)$. 

Hash table

$m$ vertices

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

- $h_2(x_1)$
- $x_1$
- $h_1(x_1)$
- $x_2$
- $x_3$
- $m$ 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 $h_2(x)$.
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 $h_2(x)$.
- Including key $x_6$ causes 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 $h_2(x)$.
- Including key $x_6$ causes a cycle. Cycles are dangerous...
- Inserting the key $x_7$ will cause a rehash, as keys will be moved around in an infinite loop (recall that we stop after $n$ steps).
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 $h_2(x)$.
- Including key $x_6$ causes a cycle. Cycles are dangerous...
- Inserting the key $x_7$ will cause a rehash, as keys will be moved around in an infinite loop (recall that we stop after $n$ steps).
- We will analyse the probability of having a cycle when inserting $n$ keys.

**Observe**
Just because there is a cycle does not necessarily mean there is a problem. The actual problem with collisions arise when there are two cycles, including the edge from key we are trying to insert. Why?
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 in the undirected cuckoo graph there exists a path from $i$ to $j$ of length $\ell \geq 1$, which is a shortest path from $i$ to $j$, is at most $\frac{1}{c^\ell \cdot m}$. 
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 in the undirected cuckoo graph there exists a path from $i$ to $j$ of length $\ell \geq 1$, which is a shortest path from $i$ to $j$, is at most $\frac{1}{c^\ell \cdot m}$.

**Proof**

- **Proof by induction.**
  
- **Base case:** $\ell = 1$.
  
- Let $K$ be the set of keys in the hash table. $|K| \leq n$.
  
- The probability that a key $x$ is mapped to $i$ and $j$, i.e. either $h_1(x) = i$, $h_2(x) = j$ or $h_1(x) = j$, $h_2(x) = i$, is at most $\frac{2}{m^2}$ (recall we have assumed independence between $h_1$ and $h_2$).
**LEMMA**

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

**PROOF**

- **Proof by induction.**
- **Base case:** $\ell = 1$.
- Let $K$ be the set of keys in the hash table. $|K| \leq n$.
- The probability that a key $x$ is mapped to $i$ and $j$, i.e. either $h_1(x) = i$, $h_2(x) = j$ or $h_1(x) = j$, $h_2(x) = i$, is at most $\frac{2}{m^2}$ (recall we have assumed independence between $h_1$ and $h_2$).
- Thus, the probability that there is an edge between $i$ and $j$ is at most (using the union bound)
  \[ \sum_{x \in K} \frac{2}{m^2} \leq \frac{2n}{m^2} \leq \frac{1}{c \cdot m}. \]

  since $m \geq 2cn$. 

  \[ \sum_{x \in K} \frac{2}{m^2} \leq \frac{2n}{m^2} \leq \frac{1}{c \cdot m}. \]
Paths in the cuckoo graph

**Inductive step:** assume lemma is true for lengths 1, 2, \ldots, \ell - 1.


Paths in the cuckoo graph

**Proof continued...**

- **Inductive step:** assume lemma is true for lengths 1, 2, \ldots, \(\ell - 1\).
- If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\) then there must be a position \(k\) such that:

  A. there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\), and
  B. there is an edge from \(k\) to \(j\).
**Inductive step**: assume lemma is true for lengths 1, 2, \ldots, \ell - 1.

If there is a path between i and j of length \ell but *not shorter* than \ell
then there must be a position k such that:

A. there is a shortest path of length \ell - 1 from i to k that does not go through j, and

B. there is an edge from k to j.

By the induction hypothesis,
\[ \Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}. \]

*Observe* The “*not got through j*” can only make the probability smaller.
**Inductive step:** assume lemma is true for lengths 1, 2, \ldots, ℓ − 1.

If there is a path between \( i \) and \( j \) of length \( ℓ \) but *not shorter* than \( ℓ \) then there must be a position \( k \) such that:

- **A** there is a shortest path of length \( ℓ − 1 \) from \( i \) to \( k \) that does not go through \( j \), and
- **B** there is an edge from \( k \) to \( j \).

By the induction hypothesis,
\[
\Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}.
\]

Given that **A** is true, the probability that **B** holds as well is upper bounded by
\[
\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}. \quad \text{(Union bound like on the previous slide over keys in } K).\]

**Observe** The “*not got through \( j \)” can only make the probability smaller.
Inductive step: assume lemma is true for lengths 1, 2, \ldots, \ell - 1.

If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\) then there must be a position \(k\) such that:

A. there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\), and

B. there is an edge from \(k\) to \(j\).

By the induction hypothesis,

\[
\Pr(A) \leq \frac{1}{c^{\ell - 1} \cdot m}.
\]

Given that \(A\) is true, the probability that \(B\) holds as well is upper bounded by

\[
\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}. \quad \text{(Union bound like on the previous slide over keys in \(K\))}
\]

\[
\Pr(A \text{ and } B) = \Pr(A) \cdot \Pr(B \mid A) \leq \frac{1}{c^{\ell - 1} \cdot m} \cdot \frac{1}{c \cdot m} = \frac{1}{c^\ell \cdot m^2}.
\]

**Observe** The “not got through \(j\)” can only make the probability smaller.
**Inductive step:** assume lemma is true for lengths 1, 2, \ldots, \ell - 1.

If there is a path between \(i\) and \(j\) of length \(\ell\) but *not shorter* than \(\ell\) then there must be a position \(k\) such that:

- **A** there is a shortest path of length \(\ell - 1\) from \(i\) to \(k\) that does not go through \(j\), and
- **B** there is an edge from \(k\) to \(j\).

By the induction hypothesis,

\[
\Pr(A) \leq \frac{1}{c^{\ell-1} \cdot m}.
\]

Observe: The “not got through \(j\)” can only make the probability smaller.

Given that **A** is true, the probability that **B** holds as well is upper bounded by

\[
\sum_{x \in K} \frac{2}{m^2} \leq \frac{1}{c \cdot m}.
\]

(Union bound like on the previous slide over keys in \(K\).)

\[
\Pr(A \text{ and } B) = \Pr(A) \cdot \Pr(B \mid A) \leq \frac{1}{c^{\ell-1} \cdot m} \cdot \frac{1}{c \cdot m} = \frac{1}{c^{\ell} \cdot m^2}.
\]

Union bound over all \(k\) gives an upper bound on the probability of a shortest path between \(i\) and \(j\) of length \(\ell\):

\[
m \cdot \frac{1}{c^{\ell} \cdot m^2} = \frac{1}{c^{\ell} \cdot m}.
\]
Back to buckets

Hash table

Two keys $x, y$ are in the same bucket if there is a path between $\{h_1(x), h_2(x)\}$ and $\{h_1(y), h_2(y)\}$ in the cuckoo graph.
Two keys $x, y$ are in the same bucket if there is a path between $\{h_1(x), h_2(x)\}$ and $\{h_1(y), h_2(y)\}$ in the cuckoo graph.

For two distinct keys $x, y$, the probability that they are in the same bucket is therefore upper bounded by

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

where $c > 1$ is a constant.

(Union bound of all possible path lengths. Why factor 4?)
Back to buckets

Hash table

Two keys \( x, y \) are in the same bucket if there is a path between \( \{ h_1(x), h_2(x) \} \) and \( \{ h_1(y), h_2(y) \} \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is therefore upper bounded by

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

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

(Union bound of all possible path lengths. Why factor 4?)

The time for an operation on \( x \) is bounded by the number of items in the bucket. (Assuming there are no cycles.)
Two keys \( x, y \) are in the same bucket if there is a path between \( \{h_1(x), h_2(x)\} \) and \( \{h_1(y), h_2(y)\} \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is therefore upper bounded by

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

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

(Union bound of all possible path lengths. Why factor 4?)

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

Thus, following the analysis from last week, we have that the expected time per operation is \( O(1) \).

We assume that \( m \geq 2cn \).

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

- The previous analysis on the expected running time applies 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?
Rehashing

- The previous analysis on the expected running time applies 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?
- For simplicity, let us assume that there are $n$ keys in the table and we want to insert *another* $n$ keys.
- We assume that the table size $m \geq 2c \cdot 2n = 4cn$, where $c > 1$ is the constant from the previous slides.
Rehashing

- The previous analysis on the expected running time applies 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?

- For simplicity, let us assume that there are \( n \) keys in the table and we want to insert *another* \( n \) keys.

- We assume that the table size \( m \geq 2c \cdot 2n = 4cn \), where \( c > 1 \) is the constant from the previous slides.

- A cycle is a path from a vertex \( i \) back to itself. We can use previous result where \( i = j \).

Recall the previous lemma...
Rehashing

- The previous analysis on the expected running time applies 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?
- For simplicity, let us assume that there are $n$ keys in the table and we want to insert another $n$ keys.
- We assume that the table size $m \geq 2c \cdot 2n = 4cn$, where $c > 1$ is the constant from the previous slides.
- A cycle is a path from a vertex $i$ back to itself.
  - We can use previous result where $i = j$.
  - Recall the previous lemma...

**Lemma**

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

The probability that a position $i$ is involved in a cycle is upper bounded, using the union bound, by

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

- The probability that a position $i$ is involved in a cycle is upper bounded, using the union bound, by
  \[
  \sum_{\ell=1}^{\infty} \frac{1}{c^{\ell} \cdot m} = \frac{1}{m(c - 1)}.
  \]

- The probability that there is at least one cycle is upper bounded, using the union bound over all positions, by
  \[
  m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.
  \]
Rehashing

- The probability that a position $i$ is involved in a cycle is upper bounded, using the union bound, by
  \[
  \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.
  \]

- The probability that there is at least one cycle is upper bounded, using the union bound over all positions, by
  \[
  m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.
  \]

- For $c = 3$, 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 upper bounded, using the union bound, by
  $$\sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c - 1)}.$$

- The probability that there is at least one cycle is upper bounded, using the union bound over all positions, by
  $$m \cdot \frac{1}{m(c - 1)} = \frac{1}{c - 1}.$$

- For $c = 3$, 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 (two independent cycles) is therefore $\frac{1}{4}$, and so on.
Rehashing

- The probability that a position $i$ is involved in a cycle is upper bounded, using the union bound, by
  \[ \sum_{\ell=1}^{\infty} \frac{1}{c^\ell \cdot m} = \frac{1}{m(c-1)}. \]

- The probability that there is at least one cycle is upper bounded, using the union bound over all positions, by
  \[ m \cdot \frac{1}{m(c-1)} = \frac{1}{c-1}. \]

- For $c = 3$, 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 (two independent cycles) is therefore $\frac{1}{4}$, and so on.

- Thus, the expected number of rehashes during $n$ insertions is therefore 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 $O(n)$ (since we expect there to be only one rehash).
- Thus, the *amortised* 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 $O(n)$ (since we expect there to be only one rehash).

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

- To see why the expected time per rehash is $O(n)$:
  - First pick random $h_1$ and $h_2$ and construct the cuckoo graph. Working out whether there is a cycle or not can be done in $O(n)$ time. **How?** The probability of there being a cycle is at most $\frac{1}{2}$. 
Rehashing

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

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

- To see why the expected time per rehash is $O(n)$:
  - First pick random $h_1$ and $h_2$ and construct the cuckoo graph.
    Working out whether there is a cycle or not can be done in $O(n)$ time. How? The probability of there being a cycle is at most $\frac{1}{2}$.
  - If there is no cycle, insert all the elements, which takes $O(n)$ in expectation (as we have seen).
Global rebuilding

We can use a technique called **global rebuilding** to adapt the size of the hash table to the number of keys inserted:

If the number of stored keys drops below a certain level then we may get away with a smaller hash table than we currently have and still have good performance. More precisely, if the number of keys drop below a certain threshold then we *half* the size of the table and rehash.
Global rebuilding

- We can use a technique called **global rebuilding** to adapt the size of the hash table to the number of keys inserted:
  - If the number of stored keys drops below a certain level then we may get away with a smaller hash table than we currently have and still have good performance.
  - More precisely, if the number of keys drop below a certain threshold then we *half* the size of the table and rehash.
  - Similarly, if the number of keys go above a certain threshold then we *double* the size of the table and rehash.
Global rebuilding

- We can use a technique called **global rebuilding** to adapt the size of the hash table to the number of keys inserted:

  - If the number of stored keys drops below a certain level then we may get away with a smaller hash table than we currently have and still have good performance. More precisely, if the number of keys drop below a certain threshold then we *half* the size of the table and rehash.

  - Similarly, if the number of keys go above a certain threshold then we *double* the size of the table and rehash.

- One can show that the amortised cost of rebuilding the hash table is constant time per operation.
A word about the assumptions

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

- Similarly to the property of a weakly universal hash family, where any two keys \( x, y \) are independent, we can define a property called \( k \)-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 seen, this is not realistic.

- Similarly to the property of a weakly universal hash family, where any two keys $x, y$ are independent, we can define a property called $k$-independence. Here the hash values of any choice of $k$ keys are independent.

- With $k = \log n$ it is feasible to construct a family of hash functions that are $k$-independent. It is not obvious though how the value of a hash function can be computed in constant time.
A word about the assumptions

- We have assumed true randomness. As we have seen, this is not realistic.
- Similarly to the property of a weakly universal hash family, where any two keys $x, y$ are independent, we can define a property called $k$-independence. Here the hash values of any choice of $k$ keys are independent.
- With $k = \log n$ it is feasible to construct a family of hash functions that are $k$-independent. It is not obvious though how the value of a hash function can be computed in constant time.
- By changing the cuckoo hashing algorithm to perform a rehash if a new key cannot be inserted after $k = \log n$ steps (instead of $n$ as in the previous slides), we can show that the expected performance is still good when using the $k$-independent family of hash functions.