Randomized Algorithms

Cuckoo hashing: constant-time lookup via relocation

Alex Gittens
Department of Computer Science, Rensselaer Polytechnic Institute

Fall 2026

From short chains to constant-time lookup

Recall Lecture 6: two-choice chaining puts each key in the less-loaded of two buckets. It gives

$$L_{\max} = \frac{\ln \ln n}{\ln 2} + O(1)$$

with high probability.

From short chains to constant-time lookup

Recall Lecture 6: two-choice chaining puts each key in the less-loaded of two buckets. It gives

$$L_{\max} = \frac{\ln \ln n}{\ln 2} + O(1)$$

with high probability.
That means the worst bucket has length $\Theta(\log \log n)$, so a lookup in that bucket may require $\Theta(\log \log n)$ comparisons.

From short chains to constant-time lookup

Recall Lecture 6: two-choice chaining puts each key in the less-loaded of two buckets. It gives

$$L_{\max} = \frac{\ln \ln n}{\ln 2} + O(1)$$

with high probability.
That means the worst bucket has length $\Theta(\log \log n)$, so a lookup in that bucket may require $\Theta(\log \log n)$ comparisons.

This is already excellent. Today we ask for a stronger kind of guarantee:

Can every lookup take $O(1)$ worst-case time?

From short chains to constant-time lookup

Recall Lecture 6: two-choice chaining puts each key in the less-loaded of two buckets. It gives

$$L_{\max} = \frac{\ln \ln n}{\ln 2} + O(1)$$

with high probability.
That means the worst bucket has length $\Theta(\log \log n)$, so a lookup in that bucket may require $\Theta(\log \log n)$ comparisons.

This is already excellent. Today we ask for a stronger kind of guarantee:

Can every lookup take $O(1)$ worst-case time?

This matters when lookups are read-heavy or latency-sensitive: caches, routers, high-throughput services, and other settings where predictable memory access matters.

PART 1

Why relocation is useful

1

A simple route to worst-case constant lookup

One way to force constant lookup time is to give each key only a constant number of possible cells and allow at most one key per cell.

A simple route to worst-case constant lookup

One way to force constant lookup time is to give each key only a constant number of possible cells and allow at most one key per cell.

We try the smallest nontrivial version. There is one table with m cells. Each key x has two *distinct* candidate cells

$$h_1(x), h_2(x) \in [m], \qquad h_1(x) \neq h_2(x).$$

A simple route to worst-case constant lookup

One way to force constant lookup time is to give each key only a constant number of possible cells and allow at most one key per cell.

We try the smallest nontrivial version. There is one table with m cells. Each key x has two *distinct* candidate cells

$$h_1(x), h_2(x) \in [m], \qquad h_1(x) \neq h_2(x).$$

Lookup checks those two cells. Deletion checks those two cells and clears the one holding x.

A simple route to worst-case constant lookup

One way to force constant lookup time is to give each key only a constant number of possible cells and allow at most one key per cell.

We try the smallest nontrivial version. There is one table with m cells. Each key x has two *distinct* candidate cells

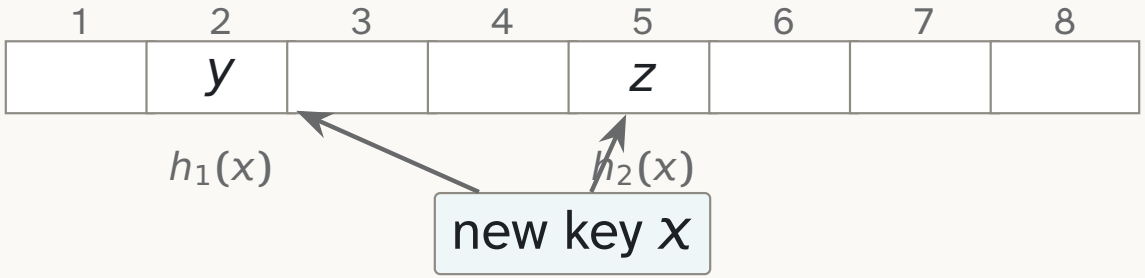
$$h_1(x), h_2(x) \in [m], \qquad h_1(x) \neq h_2(x).$$

Lookup checks those two cells. Deletion checks those two cells and clears the one holding x.

The price is paid during insertion: we may have to move keys that are already stored.

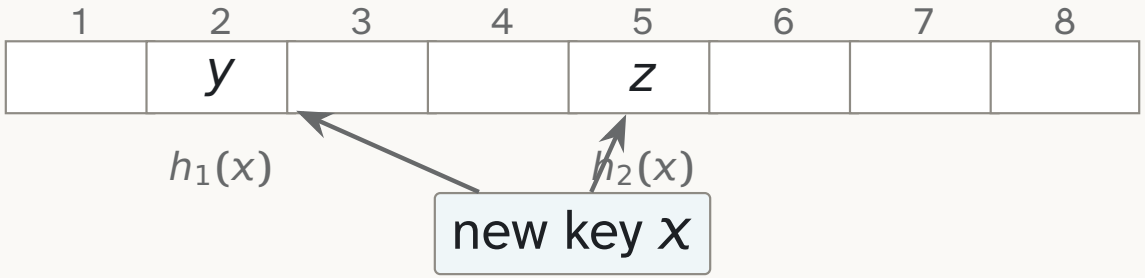
First try: no relocation

The obvious insertion rule is: put x in one of its two candidate cells if one is empty.



First try: no relocation

The obvious insertion rule is: put x in one of its two candidate cells if one is empty.



If both cells are occupied, this naive rule fails. But the resident keys also have second candidate cells. The failure may be a failure of the rule, not a failure of the hash choices.

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.
One candidate. After t successful insertions exactly t cells are occupied, so

$$\Pr\{\text{next key collides}\} = \frac{t}{m}, \qquad \sum_{j<t} \frac{j}{m} \asymp \frac{t^2}{m}.$$

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.
One candidate. After t successful insertions exactly t cells are occupied, so

$$\Pr\{\text{next key collides}\} = \frac{t}{m}, \qquad \sum_{j<t} \frac{j}{m} \asymp \frac{t^2}{m}.$$

Thus collisions occur at constant probability around $t \asymp \sqrt{m}$.

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.
One candidate. After t successful insertions exactly t cells are occupied, so

$$\Pr\{\text{next key collides}\} = \frac{t}{m}, \qquad \sum_{j<t} \frac{j}{m} \asymp \frac{t^2}{m}.$$

Thus collisions occur at constant probability around $t \asymp \sqrt{m}$.

Two distinct candidates, no relocation. After t successful insertions,

$$\Pr\{\text{both choices occupied}\} = \frac{t(t-1)}{m(m-1)} \sim \left(\frac{t}{m}\right)^2,$$

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.
One candidate. After t successful insertions exactly t cells are occupied, so

$$\Pr\{\text{next key collides}\} = \frac{t}{m}, \qquad \sum_{j<t} \frac{j}{m} \asymp \frac{t^2}{m}.$$

Thus collisions occur at constant probability around $t \asymp \sqrt{m}$.

Two distinct candidates, no relocation. After t successful insertions,

$$\Pr\{\text{both choices occupied}\} = \frac{t(t-1)}{m(m-1)} \sim \left(\frac{t}{m}\right)^2,$$

so the expected number of blocked insertions is on the order of t^3/m^2, and blocking appears around $t \asymp m^{2/3}$.

How soon does the naive rule fail?

Compare capacity-one hashing rules in a table with m cells.
One candidate. After t successful insertions exactly t cells are occupied, so

$$\Pr\{\text{next key collides}\} = \frac{t}{m}, \qquad \sum_{j<t} \frac{j}{m} \asymp \frac{t^2}{m}.$$

Thus collisions occur at constant probability around $t \asymp \sqrt{m}$.

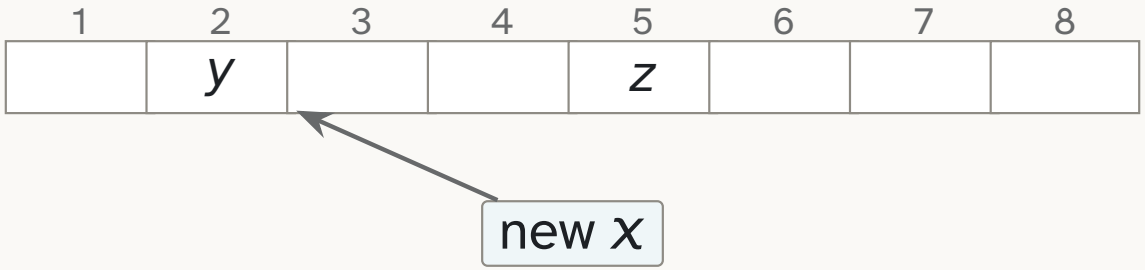
Two distinct candidates, no relocation. After t successful insertions,

$$\Pr\{\text{both choices occupied}\} = \frac{t(t-1)}{m(m-1)} \sim \left(\frac{t}{m}\right)^2,$$

so the expected number of blocked insertions is on the order of t^3/m^2, and blocking appears around $t \asymp m^{2/3}$.
Two choices postpone the obstruction from $\sqrt{m}$ to $m^{2/3}$, but still do not support $\Theta(m)$ keys.

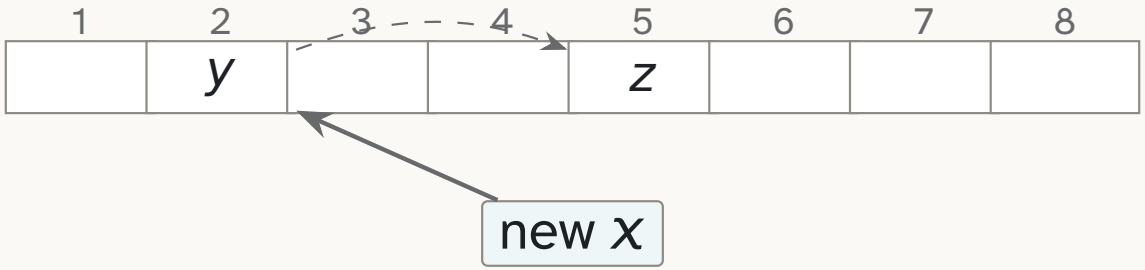
Relocation is the extra idea

A blocked insertion is not necessarily hopeless. If x wants a cell currently occupied by y, we can try moving y to its other candidate cell.



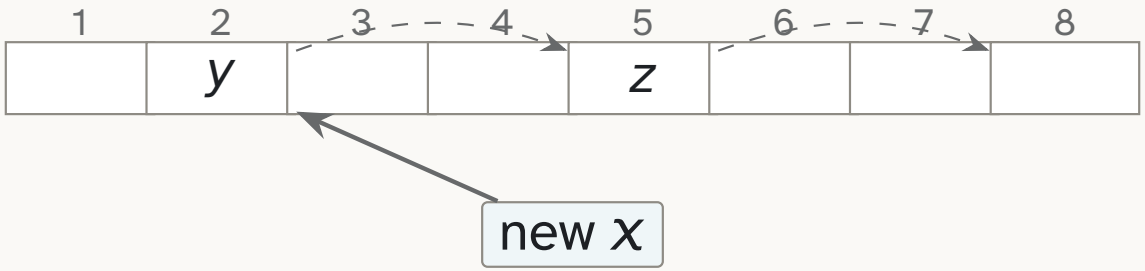
Relocation is the extra idea

A blocked insertion is not necessarily hopeless. If x wants a cell currently occupied by y, we can try moving y to its other candidate cell.



Relocation is the extra idea

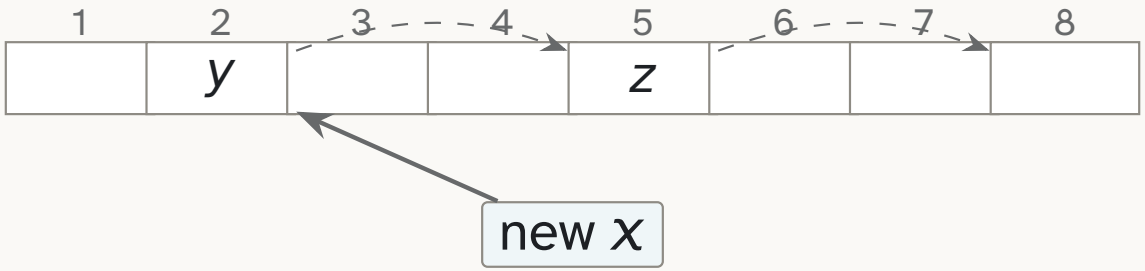
A blocked insertion is not necessarily hopeless. If x wants a cell currently occupied by y, we can try moving y to its other candidate cell.



dashed arrows show the resident key's other candidate cell

Relocation is the extra idea

A blocked insertion is not necessarily hopeless. If x wants a cell currently occupied by y, we can try moving y to its other candidate cell.

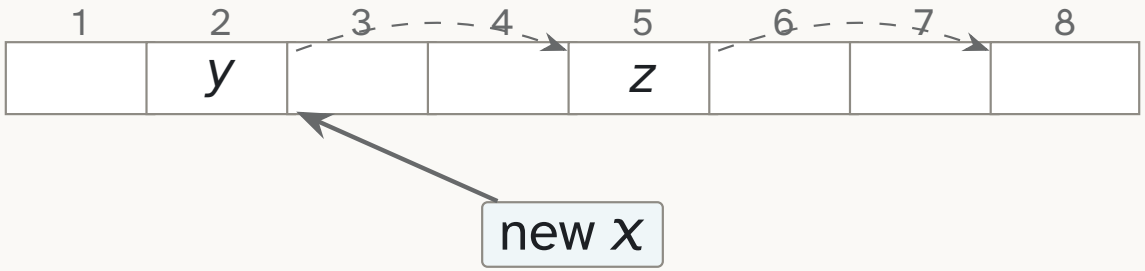


dashed arrows show the resident key's other candidate cell

We allow longer insertion time in order to make worst-case constant lookup possible. The insertion follows a chain of evictions until it finds an empty cell or times out.

Relocation is the extra idea

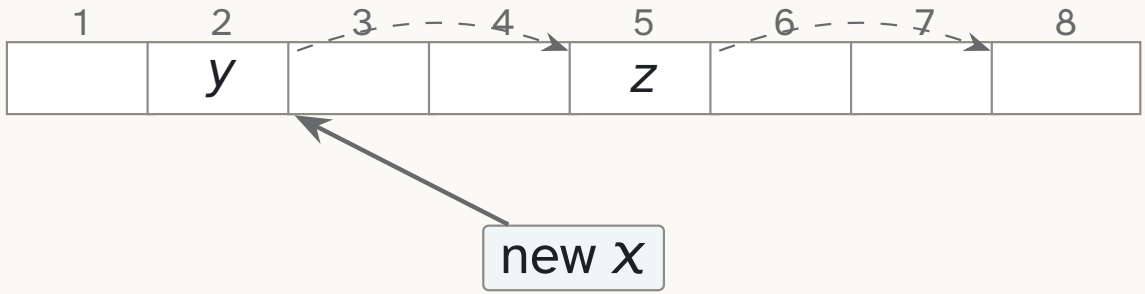
A blocked insertion is not necessarily hopeless. If x wants a cell currently occupied by y, we can try moving y to its other candidate cell.



dashed arrows show the resident key's other candidate cell

We allow longer insertion time in order to make worst-case constant lookup possible. The insertion follows a chain of evictions until it finds an empty cell or times out.
This is cuckoo hashing: an incoming key can kick out a resident key, which must then move elsewhere.

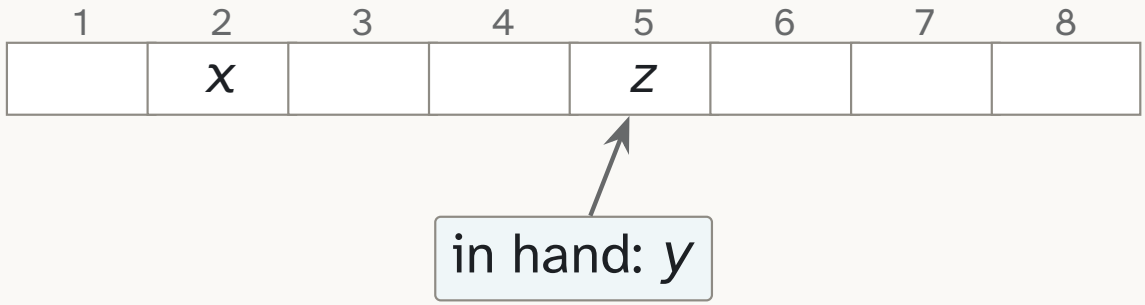
A displacement chain



dashed arrows show the resident key's other candidate cell

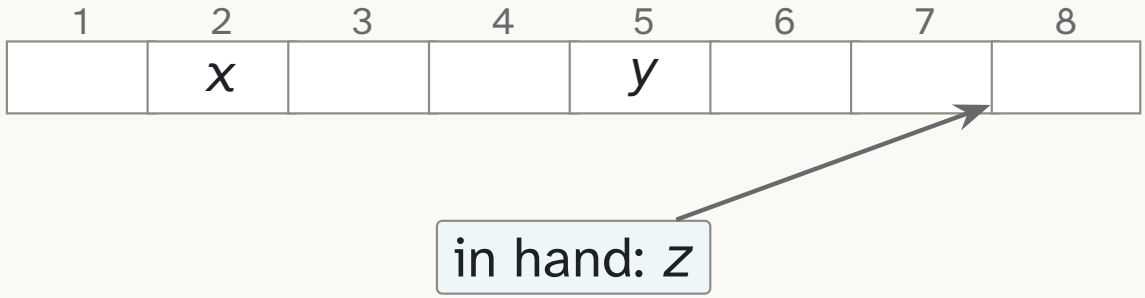
Suppose x tries cell 2, which holds y. The other cell of y is cell 5, which holds z. The other cell of z is empty.

A displacement chain



First displacement. Put x in cell 2 and evict y. Since y just left one of its candidate cells, its next destination is its other candidate cell.

A displacement chain



Second displacement. Put y into cell 5 and evict z, which must move to its own other candidate cell.

A displacement chain



all placed

z lands in an empty cell, so the insertion succeeds after two displacements.

Cuckoo insertion algorithm

Input: a key x, table $A[1{:}m]$, candidate functions h_1, h_2, and displacement cap L.
Output: success, with x stored in the table, or rehash if L displacements are reached.

1. Set $w \leftarrow x$ and choose one of $h_1(x), h_2(x)$ as the current cell c.
2. If c is empty, store w in c and return success.
3. Otherwise swap w with the resident of c. The evicted key is now in hand; count one displacement.
4. If L displacements have occurred, return rehash. Otherwise set c to the evicted key's other candidate cell and return to step 2.

At every point there is exactly one key in hand. Lookup is unaffected by the displacement history: to determine where x is, simply check $h_1(x)$ and $h_2(x)$.

The insertion rule raises three questions

Relocation gives us constant-time lookup, but we now have to understand insertion.

The insertion rule raises three questions

Relocation gives us constant-time lookup, but we now have to understand insertion.

1. **Placement.** Do these hash choices admit any valid placement of all the keys?

The insertion rule raises three questions

Relocation gives us constant-time lookup, but we now have to understand insertion.

1. **Placement.** Do these hash choices admit any valid placement of all the keys?
2. **Insertion time.** If a valid placement exists, how long can the displacement chain be?

The insertion rule raises three questions

Relocation gives us constant-time lookup, but we now have to understand insertion.

1. **Placement.** Do these hash choices admit any valid placement of all the keys?
2. **Insertion time.** If a valid placement exists, how long can the displacement chain be?
3. **When to rebuild.** If the chain runs too long, is placement impossible for these hash functions, or was this insertion merely unusually long?

The insertion rule raises three questions

Relocation gives us constant-time lookup, but we now have to understand insertion.

1. **Placement.** Do these hash choices admit any valid placement of all the keys?
2. **Insertion time.** If a valid placement exists, how long can the displacement chain be?
3. **When to rebuild.** If the chain runs too long, is placement impossible for these hash functions, or was this insertion merely unusually long?

For the placement question, the only relevant data are the two candidate cells assigned to each key.

PART 2

The graph behind the table

2

From a table to a graph

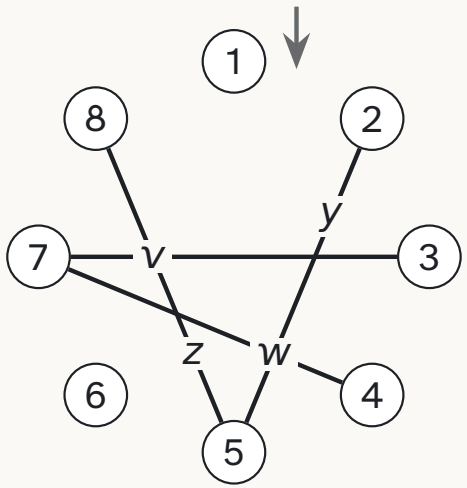
Represent each table cell by a vertex. Represent each key by an edge joining its two candidate cells.

1	2	3	4	5	6	7	8
	y		w	z		v	

From a table to a graph

Represent each table cell by a vertex. Represent each key by an edge joining its two candidate cells.

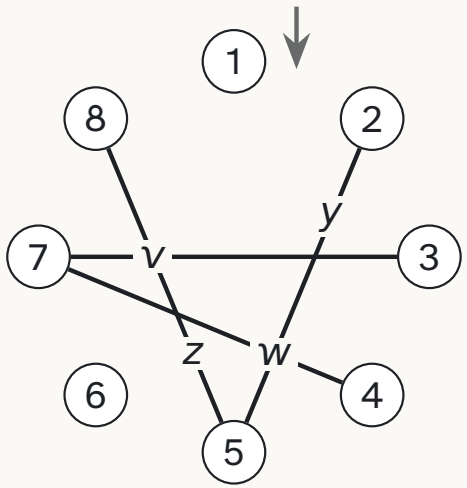
1	2	3	4	5	6	7	8
	y		*w*	*z*		*v*	



From a table to a graph

Represent each table cell by a vertex. Represent each key by an edge joining its two candidate cells.

1	2	3	4	5	6	7	8
	y		*w*	*z*		*v*	



For example, key *y* has candidate cells 2 and 5, so the edge labeled *y* joins vertices 2 and 5. The other labeled edges are read the same way.

The cuckoo graph

Cuckoo graph

Fix the n keys to be hashed and their two candidate cells. The cuckoo graph has one vertex for each of the m table cells and one edge for each key, joining that key's two candidate cells.

Different keys can have the same pair of candidate cells, so parallel edges are allowed. Because the two candidate cells of a key are distinct, there are no loops.

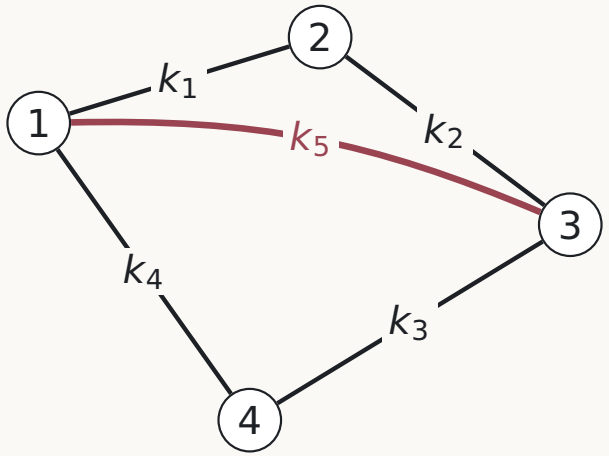
Thus

$$|V(G)| = m, \qquad |E(G)| = n.$$

The graph is determined by the keys and their hash choices; it does not depend on where the keys are currently stored.

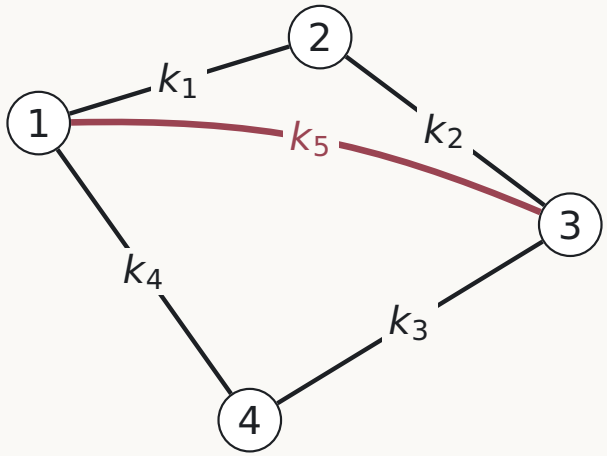
Too many keys for too few cells

Can these five keys all be placed using only the four cells shown?



Too many keys for too few cells

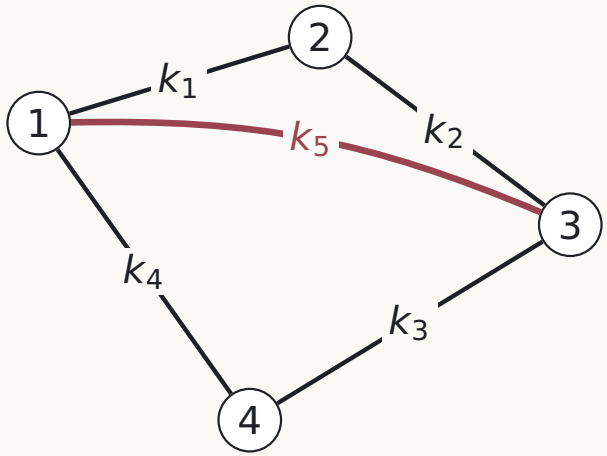
Can these five keys all be placed using only the four cells shown?



No. The five edge-keys have only four vertex-cells available to them, and each cell can hold at most one key.

Too many keys for too few cells

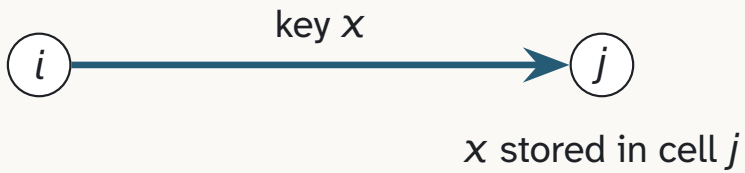
Can these five keys all be placed using only the four cells shown?



No. The five edge-keys have only four vertex-cells available to them, and each cell can hold at most one key.
This obstruction depends only on the connected component containing those five edges.

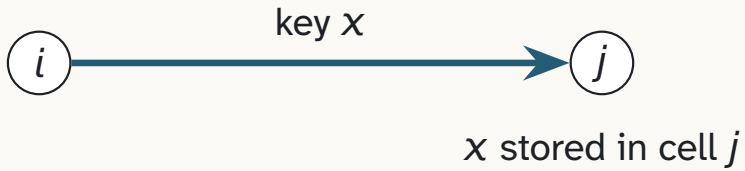
Placement as an orientation

If key x is stored at one of its two candidate cells, orient the edge for x toward that cell:



Placement as an orientation

If key x is stored at one of its two candidate cells, orient the edge for x toward that cell:



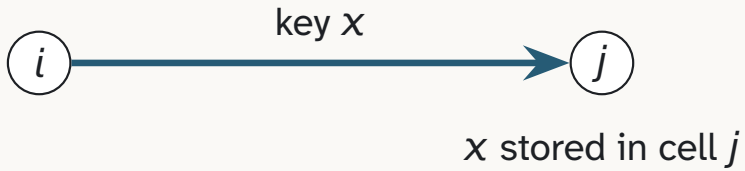
Since each cell stores at most one key, at most one edge may point into any vertex:

$$\deg^-(v) \le 1 \qquad \text{for every vertex } v.$$

Here $\deg^-(v)$ is the indegree of v, the number of oriented edges pointing into it.

Placement as an orientation

If key x is stored at one of its two candidate cells, orient the edge for x toward that cell:



Since each cell stores at most one key, at most one edge may point into any vertex:

$$\deg^-(v) \le 1 \qquad \text{for every vertex } v.$$

Here $\deg^-(v)$ is the indegree of v, the number of oriented edges pointing into it.

A valid placement is exactly an orientation of the cuckoo graph in which every vertex has indegree at most 1.

Can a component store all its keys?

Take one connected component C of the cuckoo graph.

Can a component store all its keys?

Take one connected component C of the cuckoo graph. Its edges $E(C)$ are keys, and its vertices $V(C)$ are the only cells those keys may use.

Can a component store all its keys?

Take one connected component C of the cuckoo graph. Its edges $E(C)$ are keys, and its vertices $V(C)$ are the only cells those keys may use. Each key must occupy one of its endpoints, and no cell may be used twice. By the pigeonhole principle, a placement can exist only if

$$\boxed{|E(C)| \leq |V(C)|.}$$

Can a component store all its keys?

Take one connected component C of the cuckoo graph. Its edges $E(C)$ are keys, and its vertices $V(C)$ are the only cells those keys may use. Each key must occupy one of its endpoints, and no cell may be used twice. By the pigeonhole principle, a placement can exist only if

$$\boxed{|E(C)| \leq |V(C)|.}$$

If $|E(C)| > |V(C)|$, no placement using these hash choices can store all of the keys in C.

Can a component store all its keys?

Take one connected component C of the cuckoo graph. Its edges $E(C)$ are keys, and its vertices $V(C)$ are the only cells those keys may use. Each key must occupy one of its endpoints, and no cell may be used twice. By the pigeonhole principle, a placement can exist only if

$$\boxed{|E(C)| \leq |V(C)|.}$$

If $|E(C)| > |V(C)|$, no placement using these hash choices can store all of the keys in C.
The next theorem says that this counting condition is also sufficient.

The exact placement criterion

Cuckoo placement theorem

For fixed h_1, h_2, all keys can be stored if and only if every connected component C satisfies

$$|E(C)| \leq |V(C)|.$$

Equivalently, every connected component is a tree ($|E| = |V| - 1$) or unicyclic ($|E| = |V|$).

A single cycle is therefore allowed. A double edge is a cycle of length 2. Failure begins only when some connected component has more key-edges than cell-vertices.

Proof of the placement theorem

(⇒) Assume all keys can be stored. Orient every key-edge toward the cell containing that key. Then every vertex has indegree at most 1. Hence for every connected component C,

$$|E(C)| = \sum_{v \in V(C)} \deg^-(v) \le |V(C)|.$$

Proof of the placement theorem

(⇒) Assume all keys can be stored. Orient every key-edge toward the cell containing that key. Then every vertex has indegree at most 1. Hence for every connected component C,

$$|E(C)| = \sum_{v \in V(C)} \deg^-(v) \le |V(C)|.$$

(⇐) Assume $|E(C)| \le |V(C)|$ for every connected component C. Because C is connected, $|E(C)| \ge |V(C)| - 1$. Thus each component is either a tree or unicyclic.

Proof of the placement theorem

(⇒) Assume all keys can be stored. Orient every key-edge toward the cell containing that key. Then every vertex has indegree at most 1. Hence for every connected component C,

$$|E(C)| = \sum_{v \in V(C)} \deg^-(v) \le |V(C)|.$$

(⇐) Assume $|E(C)| \le |V(C)|$ for every connected component C. Because C is connected, $|E(C)| \ge |V(C)| - 1$. Thus each component is either a tree or unicyclic.
If C is a tree, choose a root and orient every edge away from the root. Every non-root vertex then has indegree 1, and the root has indegree 0.

Proof of the placement theorem

(⇒) Assume all keys can be stored. Orient every key-edge toward the cell containing that key. Then every vertex has indegree at most 1. Hence for every connected component C,

$$|E(C)| = \sum_{v \in V(C)} \deg^-(v) \le |V(C)|.$$

(⇐) Assume $|E(C)| \le |V(C)|$ for every connected component C. Because C is connected, $|E(C)| \ge |V(C)| - 1$. Thus each component is either a tree or unicyclic.
If C is a tree, choose a root and orient every edge away from the root. Every non-root vertex then has indegree 1, and the root has indegree 0.
If C is unicyclic, orient its unique cycle cyclically and orient every tree attached to the cycle away from the cycle. Again every vertex has indegree at most 1.

Proof of the placement theorem

(⇒) Assume all keys can be stored. Orient every key-edge toward the cell containing that key. Then every vertex has indegree at most 1. Hence for every connected component C,

$$|E(C)| = \sum_{v \in V(C)} \deg^-(v) \le |V(C)|.$$

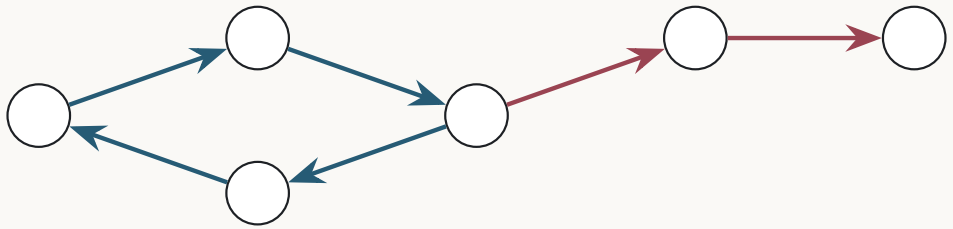
(⇐) Assume $|E(C)| \le |V(C)|$ for every connected component C. Because C is connected, $|E(C)| \ge |V(C)| - 1$. Thus each component is either a tree or unicyclic.
If C is a tree, choose a root and orient every edge away from the root. Every non-root vertex then has indegree 1, and the root has indegree 0.
If C is unicyclic, orient its unique cycle cyclically and orient every tree attached to the cycle away from the cycle. Again every vertex has indegree at most 1.
Thus every component has an orientation with indegree at most 1, so all keys can be stored. □

Insertion in a tree or unicyclic component

The placement theorem is static: it says a placement exists. The insertion algorithm still has to *find* an empty cell by following evictions.

Eviction-walk bound

Let C be the component containing the new key after it is added. If C is a tree or is unicyclic, the eviction walk reaches an empty cell within at most $2|V(C)|$ displacements.



around the unique cycle, then along a tree path

Thus, whenever the component containing x is a tree or unicyclic,

$$T_x \le 2|V(C_x)|.$$

The proof is in the appendix.

Two graph properties imply the guarantees

The placement theorem and the eviction-walk bound reduce the analysis of cuckoo hashing to two properties of the random cuckoo graph.

Two graph properties imply the guarantees

The placement theorem and the eviction-walk bound reduce the analysis of cuckoo hashing to two properties of the random cuckoo graph.

1. **Placement.** If every connected component is a tree or unicyclic, then all n keys can be stored, so every lookup checks only the two candidate cells.

Two graph properties imply the guarantees

The placement theorem and the eviction-walk bound reduce the analysis of cuckoo hashing to two properties of the random cuckoo graph.

1. **Placement.** If every connected component is a tree or unicyclic, then all n keys can be stored, so every lookup checks only the two candidate cells.
2. **Insertion time.** If every connected component has $O(\log n)$ vertices, then
$$T_x \leq 2|V(C_x)| = O(\log n)$$
for every insertion whose component is tree or unicyclic.

Two graph properties imply the guarantees

The placement theorem and the eviction-walk bound reduce the analysis of cuckoo hashing to two properties of the random cuckoo graph.

1. **Placement.** If every connected component is a tree or unicyclic, then all n keys can be stored, so every lookup checks only the two candidate cells.
2. **Insertion time.** If every connected component has $O(\log n)$ vertices, then
$$T_x \leq 2|V(C_x)| = O(\log n)$$
for every insertion whose component is tree or unicyclic.

So we next prove that, with high probability,

there are no components with more edges than vertices,

Two graph properties imply the guarantees

The placement theorem and the eviction-walk bound reduce the analysis of cuckoo hashing to two properties of the random cuckoo graph.

1. **Placement.** If every connected component is a tree or unicyclic, then all n keys can be stored, so every lookup checks only the two candidate cells.
2. **Insertion time.** If every connected component has $O(\log n)$ vertices, then
$$T_x \leq 2|V(C_x)| = O(\log n)$$
for every insertion whose component is tree or unicyclic.

So we next prove that, with high probability,

there are no components with more edges than vertices,

and the largest component has $O(\log n)$ vertices.

PART 3

Why random hashing usually works

3

The load condition

For the random-graph analysis, fix a set of n keys and let each key independently choose two distinct uniformly random candidate cells from the m table cells.
The cuckoo graph then has m vertices and n edges, so its average degree is exactly

$$\frac{2n}{m}.$$

We work below the critical average-degree scale: n is a fixed constant factor below $m/2$.

For the finite-m calculations below, it is convenient to impose the slightly stronger condition

$$\boxed{\frac{2n}{m-1} \le 1-\varepsilon}, \qquad \varepsilon > 0 \text{ fixed.}$$

This differs from $2n/m \le 1-\varepsilon$ only by the harmless $m/(m-1)$ finite-size factor.

How likely is a bad component?

Bad components are rare

Fix $\varepsilon > 0$. Suppose $2n/(m-1) \le 1-\varepsilon$, and each of the n keys chooses two distinct candidate cells independently and uniformly. Then

$$\Pr\{\exists \text{ component } C : \ |E(C)| > |V(C)|\} \le \frac{c_\varepsilon}{m}.$$

Thus a fresh pair of hash functions admits a valid placement with probability at least $1 - c_\varepsilon/m$.

To prove this, we show that every bad component contains a smaller subgraph with a very restricted shape. We then count the possible labeled subgraphs of those shapes and union-bound the probability that any one of them occurs.

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\operatorname{excess}(H) := |E(H)| - |V(H)|.$$

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\mathrm{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\mathrm{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

1. H has no leaf. Deleting a leaf and its incident edge preserves $|E| - |V|$, contradicting minimality.

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\mathrm{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

1. H has no leaf. Deleting a leaf and its incident edge preserves $|E| - |V|$, contradicting minimality.
2. $\mathrm{excess}(H) = 1$. If the excess were at least 2, an edge on a cycle could be deleted while keeping the graph connected and with positive excess.

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\operatorname{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

1. H has no leaf. Deleting a leaf and its incident edge preserves $|E| - |V|$, contradicting minimality.
2. $\operatorname{excess}(H) = 1$. If the excess were at least 2, an edge on a cycle could be deleted while keeping the graph connected and with positive excess.

Therefore every bad component contains a connected subgraph H satisfying

$$|E(H)| = |V(H)| + 1, \qquad \deg_H(v) \geq 2 \text{ for every } v.$$

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\mathrm{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

1. H has no leaf. Deleting a leaf and its incident edge preserves $|E| - |V|$, contradicting minimality.
2. $\mathrm{excess}(H) = 1$. If the excess were at least 2, an edge on a cycle could be deleted while keeping the graph connected and with positive excess.

Therefore every bad component contains a connected subgraph H satisfying

$$|E(H)| = |V(H)| + 1, \qquad \deg_H(v) \geq 2 \text{ for every } v.$$

We call such an H a **minimal bad witness**.

Minimal bad witnesses

Suppose a connected component C is bad: $|E(C)| > |V(C)|$. Among its connected subgraphs with positive excess, choose one H with as few edges as possible, where

$$\mathrm{excess}(H) := |E(H)| - |V(H)|.$$

Minimality forces two properties:

1. H has no leaf. Deleting a leaf and its incident edge preserves $|E| - |V|$, contradicting minimality.
2. $\mathrm{excess}(H) = 1$. If the excess were at least 2, an edge on a cycle could be deleted while keeping the graph connected and with positive excess.

Therefore every bad component contains a connected subgraph H satisfying

$$|E(H)| = |V(H)| + 1, \qquad \deg_H(v) \geq 2 \text{ for every } v.$$

We call such an H a **minimal bad witness**.
Therefore, to rule out bad components, it is enough to bound the probability that any minimal bad witness occurs.

Three witness topologies

For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Three witness topologies

For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Each term $\deg v - 2$ is a nonnegative integer because every vertex has degree at least 2. Therefore exactly one of the following happens:

Three witness topologies

For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Each term $\deg v - 2$ is a nonnegative integer because every vertex has degree at least 2. Therefore exactly one of the following happens:

- one vertex has degree 4 and every other vertex has degree 2;

Three witness topologies

For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Each term $\deg v - 2$ is a nonnegative integer because every vertex has degree at least 2. Therefore exactly one of the following happens:

- one vertex has degree 4 and every other vertex has degree 2;
- two vertices have degree 3 and every other vertex has degree 2.

Three witness topologies

For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Each term $\deg v - 2$ is a nonnegative integer because every vertex has degree at least 2. Therefore exactly one of the following happens:

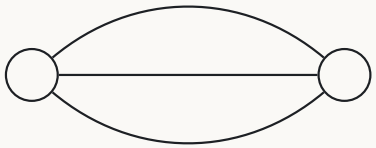
- one vertex has degree 4 and every other vertex has degree 2;
- two vertices have degree 3 and every other vertex has degree 2.

Suppress every maximal path whose internal vertices all have degree 2. This does not change how the cycles are connected; it only replaces such a path by a single schematic edge. The resulting topology is one of:

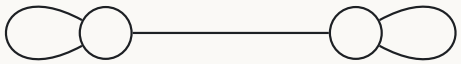
figure-eight



theta



dumbbell



Three witness topologies

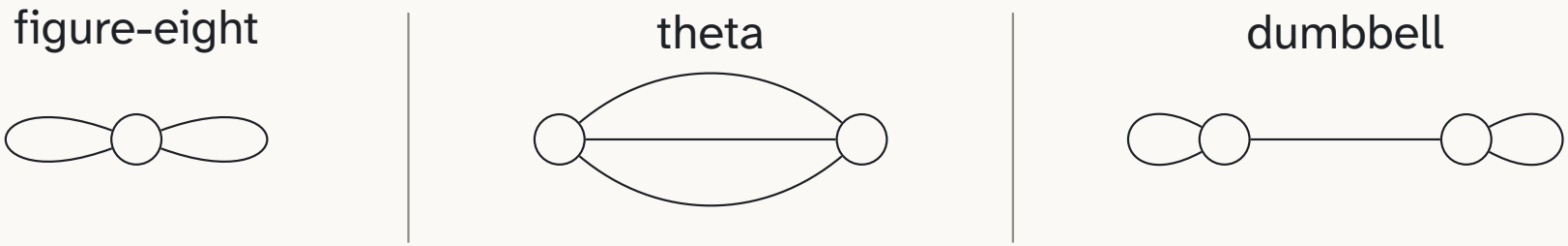
For a minimal bad witness H,

$$\sum_{v \in V(H)} (\deg v - 2) = 2|E(H)| - 2|V(H)| = 2.$$

Each term $\deg v - 2$ is a nonnegative integer because every vertex has degree at least 2. Therefore exactly one of the following happens:

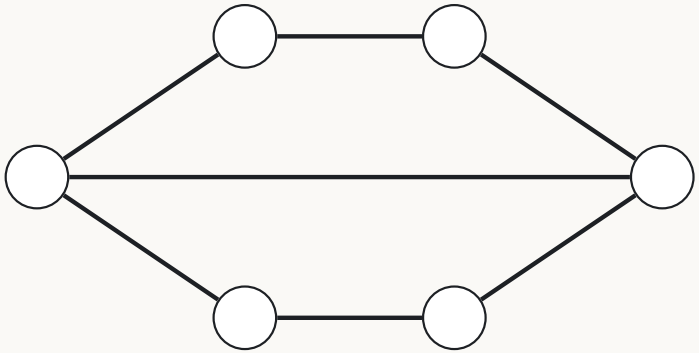
- one vertex has degree 4 and every other vertex has degree 2;
- two vertices have degree 3 and every other vertex has degree 2.

Suppress every maximal path whose internal vertices all have degree 2. This does not change how the cycles are connected; it only replaces such a path by a single schematic edge. The resulting topology is one of:

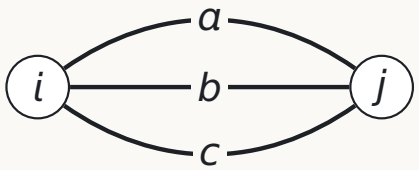


These excess-one connected graphs have two independent cycles. They are often called **bicycles**.

A witness in the cuckoo graph



a theta witness: 6 cells, 7 keys



smallest witness: 2 cells, 3 keys

If a minimal witness has $s := |E(H)|$ edges, then $|E(H)| = |V(H)| + 1$ gives immediately

$$|V(H)| = s - 1.$$

Because the cuckoo graph has no loops, no connected witness with $s = 1$ or 2 can satisfy $|V| = s - 1$. Hence $s \geq 3$. The smallest example is three parallel edges between two cells.

From witness counts to probability

Fix a witness size s. A **candidate witness** specifies

1. one of the three topological forms and its path lengths;
2. the $s - 1$ table cells assigned to its vertices;
3. the s distinct keys assigned to its edges.

Let N_s be the number of candidate witnesses of size s, and let p_s be the probability that one fixed candidate occurs. The union bound gives

$$\Pr\{\exists \text{ bad witness of size } s\} \le N_s p_s.$$

So the proof has two separate tasks: count N_s, then compute p_s.
We first work out the count for the theta shape.

A worked count: theta witnesses

A theta witness has three internally disjoint paths with common endpoints. If their positive lengths are a, b, c, then

$$a + b + c = s.$$

A worked count: theta witnesses

A theta witness has three internally disjoint paths with common endpoints. If their positive lengths are a, b, c, then

$$a + b + c = s.$$

Set $a' = a - 1$, $b' = b - 1$, $c' = c - 1$. Then $a' + b' + c' = s - 3$ with nonnegative variables, so stars and bars gives

$$\binom{(s-3)+2}{2} = \binom{s-1}{2} = O(s^2)$$

possible triples (a, b, c).

A worked count: theta witnesses

A theta witness has three internally disjoint paths with common endpoints. If their positive lengths are a, b, c, then

$$a + b + c = s.$$

Set $a' = a - 1$, $b' = b - 1$, $c' = c - 1$. Then $a' + b' + c' = s - 3$ with nonnegative variables, so stars and bars gives

$$\binom{(s-3)+2}{2} = \binom{s-1}{2} = O(s^2)$$

possible triples (a, b, c).
For each triple, the witness has $s - 1$ vertices and s edge positions, so

$$\text{cell assignments} \le m^{s-1}, \qquad \text{key assignments} \le (n)_s \le n^s.$$

A worked count: theta witnesses

A theta witness has three internally disjoint paths with common endpoints. If their positive lengths are a, b, c, then

$$a + b + c = s.$$

Set $a' = a - 1$, $b' = b - 1$, $c' = c - 1$. Then $a' + b' + c' = s - 3$ with nonnegative variables, so stars and bars gives

$$\binom{(s-3)+2}{2} = \binom{s-1}{2} = O(s^2)$$

possible triples (a, b, c).
For each triple, the witness has $s - 1$ vertices and s edge positions, so

$$\text{cell assignments} \le m^{s-1}, \qquad \text{key assignments} \le (n)_s \le n^s.$$

Therefore the number of candidate theta witnesses is at most $O(s^2)m^{s-1}n^s$.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.
- Theta: three positive path lengths summing to s, so $O(s^2)$ choices.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.
- Theta: three positive path lengths summing to s, so $O(s^2)$ choices.
- Dumbbell: two positive cycle lengths and a nonnegative connecting-path length, so $O(s^2)$ choices.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.
- Theta: three positive path lengths summing to s, so $O(s^2)$ choices.
- Dumbbell: two positive cycle lengths and a nonnegative connecting-path length, so $O(s^2)$ choices.

For any fixed choice of lengths, there are at most m^{s-1} assignments of table cells to witness vertices and at most n^s ordered assignments of distinct keys to witness edges.

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.
- Theta: three positive path lengths summing to s, so $O(s^2)$ choices.
- Dumbbell: two positive cycle lengths and a nonnegative connecting-path length, so $O(s^2)$ choices.

For any fixed choice of lengths, there are at most m^{s-1} assignments of table cells to witness vertices and at most n^s ordered assignments of distinct keys to witness edges.
Thus, for a constant c_1,

$$\#\{\text{candidate witnesses with } s \text{ edges}\} \le c_1 s^2 m^{s-1} n^s.$$

Counting all witness types

The same argument gives the same order of magnitude for all three topological forms.

- Figure-eight: two positive cycle lengths summing to s, so $O(s)$ choices.
- Theta: three positive path lengths summing to s, so $O(s^2)$ choices.
- Dumbbell: two positive cycle lengths and a nonnegative connecting-path length, so $O(s^2)$ choices.

For any fixed choice of lengths, there are at most m^{s-1} assignments of table cells to witness vertices and at most n^s ordered assignments of distinct keys to witness edges.
Thus, for a constant c_1,

$$\#\{\text{candidate witnesses with } s \text{ edges}\} \le c_1 s^2 m^{s-1} n^s.$$

Allowing repeated cell labels in this count only overcounts, which is harmless for an upper bound.

Probability that one candidate witness occurs

Fix one candidate witness: s specified keys assigned to s specified unordered pairs of distinct table cells.

Probability that one candidate witness occurs

Fix one candidate witness: s specified keys assigned to s specified unordered pairs of distinct table cells.
For one specified key, the two distinct candidate cells are a uniformly random unordered pair, so

$$\Pr\{\text{the key realizes its prescribed edge}\} = \frac{1}{\binom{m}{2}} = \frac{2}{m(m-1)}.$$

Probability that one candidate witness occurs

Fix one candidate witness: s specified keys assigned to s specified unordered pairs of distinct table cells.
For one specified key, the two distinct candidate cells are a uniformly random unordered pair, so

$$\Pr\{\text{the key realizes its prescribed edge}\} = \frac{1}{\binom{m}{2}} = \frac{2}{m(m-1)}.$$

The s keys are distinct and choose their pairs independently. Therefore

$$\Pr\{\text{this fixed candidate witness occurs}\} = \left(\frac{2}{m(m-1)}\right)^s.$$

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

$$\Pr\{\exists \text{ bad witness with } s \text{ edges}\} \leq c_1 s^2 m^{s-1} n^s \cdot \left(\frac{2}{m(m-1)}\right)^s$$

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

$$\begin{aligned}\Pr\{\exists \text{ bad witness with } s \text{ edges}\} &\leq c_1 s^2 m^{s-1} n^s \cdot \left(\frac{2}{m(m-1)}\right)^s \\ &\leq \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s\end{aligned}$$

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

$$\begin{aligned}\Pr\{\exists \text{ bad witness with } s \text{ edges}\} &\leq c_1 s^2 m^{s-1} n^s \cdot \left(\frac{2}{m(m-1)}\right)^s \\ &\leq \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s \\ &= \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s.\end{aligned}$$

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

$$\begin{aligned}\Pr\{\exists \text{ bad witness with } s \text{ edges}\} &\leq c_1 s^2 m^{s-1} n^s \cdot \left(\frac{2}{m(m-1)}\right)^s \\ &\leq \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s \\ &= \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s.\end{aligned}$$

This is where the load condition enters. If $2n/(m-1) < 1$, the probability decreases geometrically with the witness size s.

Union bound for witnesses of fixed size

For a fixed $s \geq 3$, combine the count with the probability that one fixed candidate witness occurs:

$$\begin{aligned}\Pr\{\exists \text{ bad witness with } s \text{ edges}\} &\leq c_1 s^2 m^{s-1} n^s \cdot \left(\frac{2}{m(m-1)}\right)^s \\ &\leq \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s \\ &= \frac{c_2}{m} s^2 \left(\frac{2n}{m-1}\right)^s.\end{aligned}$$

This is where the load condition enters. If $2n/(m-1) < 1$, the probability decreases geometrically with the witness size s.
The factor $1/m$ comes from the obstruction itself: a bicycle has s edges but only $s-1$ cell-vertices. That one-edge surplus leaves one uncancelled factor of $1/m$.

Sum over all witness sizes

Let $q = 2n/(m-1) \leq 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \geq 3$, so

Sum over all witness sizes

Let $q = 2n/(m-1) \le 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \ge 3$, so

$$\Pr\{\text{some bad component exists}\} \le \sum_{s \ge 3} \Pr\{\exists \text{ bad witness with } s \text{ edges}\}$$

Sum over all witness sizes

Let $q = 2n/(m-1) \le 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \ge 3$, so

$$\begin{aligned}\Pr\{\text{some bad component exists}\} &\le \sum_{s\ge 3} \Pr\{\exists \text{ bad witness with } s \text{ edges}\} \\ &\le \frac{c_2}{m} \sum_{s\ge 3} s^2 q^s\end{aligned}$$

Sum over all witness sizes

Let $q = 2n/(m-1) \le 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \ge 3$, so

$$\begin{aligned}\Pr\{\text{some bad component exists}\} &\le \sum_{s\ge 3} \Pr\{\exists \text{ bad witness with } s \text{ edges}\} \\ &\le \frac{c_2}{m} \sum_{s\ge 3} s^2 q^s \\ &\le \frac{c_1}{m} \sum_{s\ge 0} s^2 (1-\varepsilon)^s.\end{aligned}$$

Sum over all witness sizes

Let $q = 2n/(m-1) \le 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \ge 3$, so

$$\begin{aligned}\Pr\{\text{some bad component exists}\} &\le \sum_{s\ge 3} \Pr\{\exists \text{ bad witness with } s \text{ edges}\} \\ &\le \frac{c_2}{m} \sum_{s\ge 3} s^2 q^s \\ &\le \frac{c_1}{m} \sum_{s\ge 0} s^2 (1-\varepsilon)^s.\end{aligned}$$

For $0 < q < 1$,

$$\sum_{s\ge 0} s^2 q^s = \frac{q(1+q)}{(1-q)^3} \le \frac{2}{\varepsilon^3} \qquad (q \le 1-\varepsilon).$$

Sum over all witness sizes

Let $q = 2n/(m-1) \le 1-\varepsilon$. A bad cuckoo graph must contain a bicycle witness of some size $s \ge 3$, so

$$\begin{aligned}\Pr\{\text{some bad component exists}\} &\le \sum_{s\ge 3} \Pr\{\exists \text{ bad witness with } s \text{ edges}\} \\ &\le \frac{c_2}{m} \sum_{s\ge 3} s^2 q^s \\ &\le \frac{c_1}{m} \sum_{s\ge 0} s^2 (1-\varepsilon)^s.\end{aligned}$$

For $0 < q < 1$,

$$\sum_{s\ge 0} s^2 q^s = \frac{q(1+q)}{(1-q)^3} \le \frac{2}{\varepsilon^3} \qquad (q \le 1-\varepsilon).$$

Therefore

$$\Pr(BadGraph) \le \frac{2c_2}{\varepsilon^3 m} = O_\varepsilon(1/m).$$

PART 4

Why insertion is fast

4

Next: bound the component sizes

The witness count has controlled placement failure. To control insertion time, we now need a bound on connected-component size.

Next: bound the component sizes

The witness count has controlled placement failure. To control insertion time, we now need a bound on connected-component size. We study one component by BFS. The relevant regime is *subcritical*: each processed vertex produces fewer than one new vertex on average. Then the exploration dies out quickly.

Subcritical BFS has an exponential tail

The homework proves the following sparse-graph bound. We restate it here in the form we need.

Subcritical component bound

Suppose a BFS exploration can be dominated by a branching process whose number of children has mean at most $c < 1$. If $C(v)$ is the component of the starting vertex, then

$$\Pr\{|C(v)| \geq k+1\} \leq \exp\left(-\frac{(1-c)^2}{4}k\right)$$

up to harmless absolute constants.

The proof follows the BFS queue and applies a Chernoff bound to the number of newly discovered vertices. We will verify the hypothesis for the cuckoo graph.

Why Poissonize the cuckoo graph?

For the fixed-n model, define

$$X_{uv} := \#\{x : (h_1(x), h_2(x)) = (u, v)\}, \qquad u \neq v.$$

There are $m(m-1)$ possible ordered pairs.

ordered pair	number of keys
$1 \to 2$	X_{12}
$1 \to 3$	X_{13}
$2 \to 1$	X_{21}
$\vdots$	$\vdots$

Why Poissonize the cuckoo graph?

For the fixed-n model, define

$$X_{uv} := \#\{x : (h_1(x), h_2(x)) = (u, v)\}, \qquad u \neq v.$$

There are $m(m-1)$ possible ordered pairs.

ordered pair	number of keys
$1 \to 2$	X_{12}
$1 \to 3$	X_{13}
$2 \to 1$	X_{21}
$\vdots$	$\vdots$

These counts are dependent: they must satisfy

$$\sum_{u \neq v} X_{uv} = n.$$

Why Poissonize the cuckoo graph?

For the fixed-n model, define

$$X_{uv} := \#\{x : (h_1(x), h_2(x)) = (u, v)\}, \qquad u \neq v.$$

There are $m(m-1)$ possible ordered pairs.

ordered pair	number of keys
$1 \to 2$	X_{12}
$1 \to 3$	X_{13}
$2 \to 1$	X_{21}
$\vdots$	$\vdots$

These counts are dependent: they must satisfy

$$\sum_{u \neq v} X_{uv} = n.$$

That dependence is inconvenient during BFS: learning how many keys used some pairs changes the distribution of the remaining pair counts.

Independent edge counts after Poissonization

Instead of fixing the total number of keys, choose

$$N \sim \mathrm{Poisson}(\alpha m), \qquad \alpha := \frac{1 - \varepsilon/2}{2},$$

and let those N keys choose independent ordered pairs of distinct cells.

Independent edge counts after Poissonization

Instead of fixing the total number of keys, choose

$$N \sim \mathrm{Poisson}(\alpha m), \qquad \alpha := \frac{1 - \varepsilon/2}{2},$$

and let those N keys choose independent ordered pairs of distinct cells. Poisson splitting then gives, simultaneously for all $u \neq v$,

$$X_{uv} \overset{\text{ind}}{\sim} \mathrm{Poisson}\left(\frac{\alpha}{m-1}\right).$$

Independent edge counts after Poissonization

Instead of fixing the total number of keys, choose

$$N \sim \mathrm{Poisson}(\alpha m), \qquad \alpha := \frac{1-\varepsilon/2}{2},$$

and let those N keys choose independent ordered pairs of distinct cells. Poisson splitting then gives, simultaneously for all $u \neq v$,

$$X_{uv} \overset{\text{ind}}{\sim} \mathrm{Poisson}\left(\frac{\alpha}{m-1}\right).$$

Conversely, if we independently generate these $m(m-1)$ Poisson counts, then

$$N = \sum_{u \neq v} X_{uv} \sim \mathrm{Poisson}\left(m(m-1)\frac{\alpha}{m-1}\right) = \mathrm{Poisson}(\alpha m).$$

Independent edge counts after Poissonization

Instead of fixing the total number of keys, choose

$$N \sim \mathrm{Poisson}(\alpha m), \qquad \alpha := \frac{1-\varepsilon/2}{2},$$

and let those N keys choose independent ordered pairs of distinct cells. Poisson splitting then gives, simultaneously for all $u \neq v$,

$$X_{uv} \overset{\text{ind}}{\sim} \mathrm{Poisson}\left(\frac{\alpha}{m-1}\right).$$

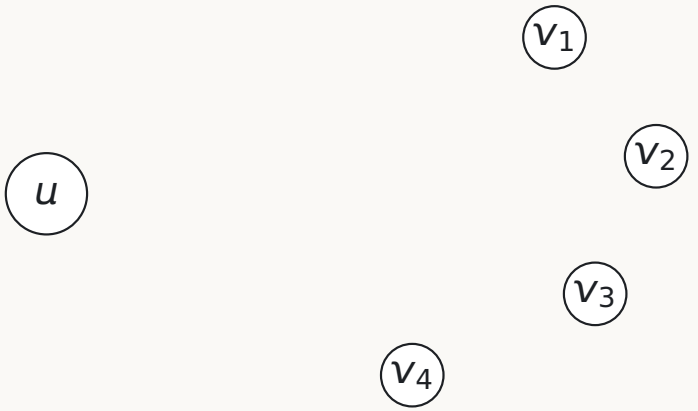
Conversely, if we independently generate these $m(m-1)$ Poisson counts, then

$$N = \sum_{u \neq v} X_{uv} \sim \mathrm{Poisson}\left(m(m-1)\frac{\alpha}{m-1}\right) = \mathrm{Poisson}(\alpha m).$$

The total number of keys is now random, while the edge counts for different ordered pairs are independent.

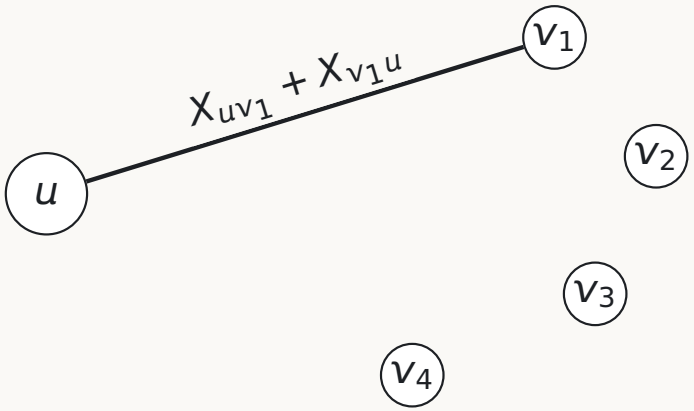
One BFS step in the Poissonized graph

Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



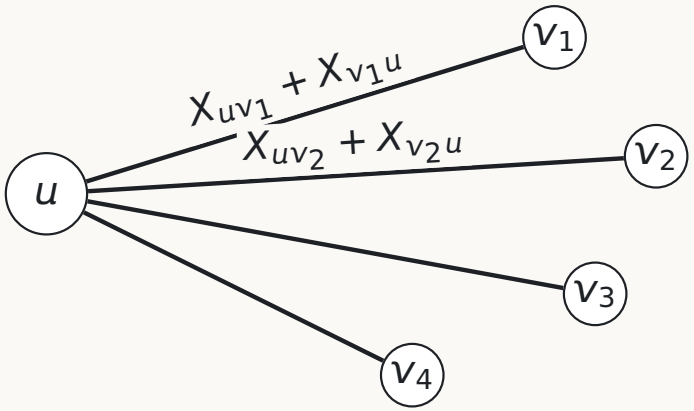
One BFS step in the Poissonized graph

Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



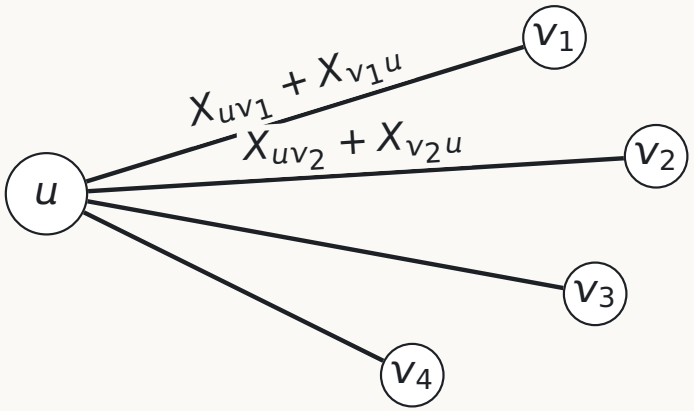
One BFS step in the Poissonized graph

Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



One BFS step in the Poissonized graph

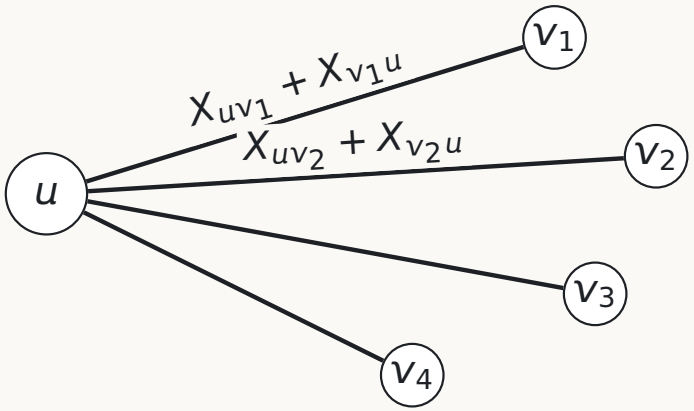
Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



Parallel edges count separately here,
but can discover at most one new vertex.

One BFS step in the Poissonized graph

Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



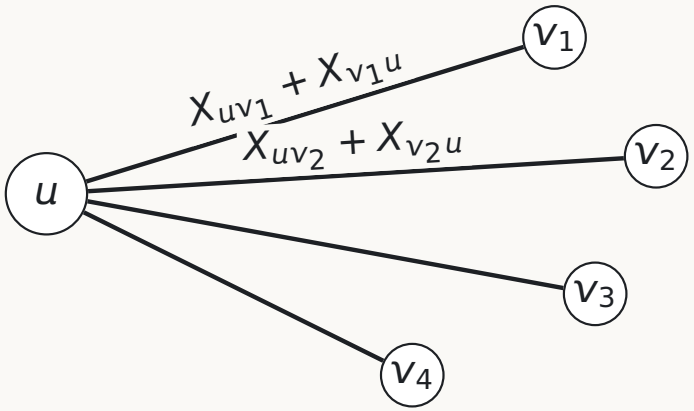
Parallel edges count separately here,
but can discover at most one new vertex.

The total number of incident edge occurrences is Poisson with mean

$$2(m-1)\frac{\alpha}{m-1} = 2\alpha = 1 - \frac{\varepsilon}{2}.$$

One BFS step in the Poissonized graph

Process a vertex u during BFS. For each unseen cell v, keys can connect u and v in either order.



Parallel edges count separately here, but can discover at most one new vertex.

The total number of incident edge occurrences is Poisson with mean

$$2(m-1)\frac{\alpha}{m-1} = 2\alpha = 1 - \frac{\varepsilon}{2}.$$

Therefore the number of *new vertices* discovered in one BFS step is dominated by $\mathrm{Poisson}(1 - \varepsilon/2)$.

The Poissonized component tail

The preceding slide verifies the hypothesis of the subcritical component bound with

$$c = 1 - \frac{\varepsilon}{2}.$$

The Poissonized component tail

The preceding slide verifies the hypothesis of the subcritical component bound with

$$c = 1 - \frac{\varepsilon}{2}.$$

Hence

$$\Pr_{\text{Pois}}\{|C(u)| \geq k+1\} \leq \exp\left(-\frac{(1-c)^2}{4}k\right).$$

The Poissonized component tail

The preceding slide verifies the hypothesis of the subcritical component bound with

$$c = 1 - \frac{\varepsilon}{2}.$$

Hence

$$\Pr_{\text{Pois}}\{|C(u)| \geq k+1\} \leq \exp\left(-\frac{(1-c)^2}{4}k\right).$$

Now substitute $1 - c = \varepsilon/2$:

$$\frac{(1-c)^2}{4} = \frac{(\varepsilon/2)^2}{4} = \frac{\varepsilon^2}{16}.$$

The Poissonized component tail

The preceding slide verifies the hypothesis of the subcritical component bound with

$$c = 1 - \frac{\varepsilon}{2}.$$

Hence

$$\Pr_{\text{Pois}}\{|C(u)| \geq k+1\} \leq \exp\left(-\frac{(1-c)^2}{4}k\right).$$

Now substitute $1 - c = \varepsilon/2$:

$$\frac{(1-c)^2}{4} = \frac{(\varepsilon/2)^2}{4} = \frac{\varepsilon^2}{16}.$$

Therefore

$$\boxed{\Pr_{\text{Pois}}\{|C(u)| \geq k+1\} \leq e^{-\varepsilon^2 k/16}.}$$

Coupling back to exactly n keys

Generate one infinite sequence of independent random key-edges

$$e_1, e_2, e_3, \ldots$$

and independently draw $N \sim \text{Poisson}(\alpha m)$.

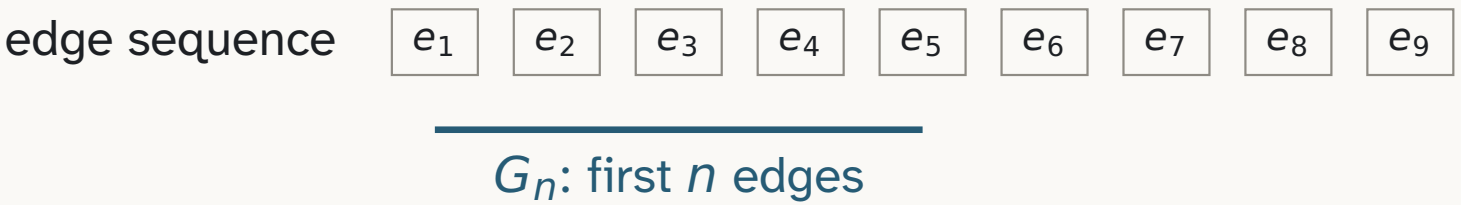
edge sequence | e_1 | e_2 | e_3 | e_4 | e_5 | e_6 | e_7 | e_8 | e_9 |

Coupling back to exactly n keys

Generate one infinite sequence of independent random key-edges

$$e_1, e_2, e_3, \ldots$$

and independently draw $N \sim \text{Poisson}(\alpha m)$.

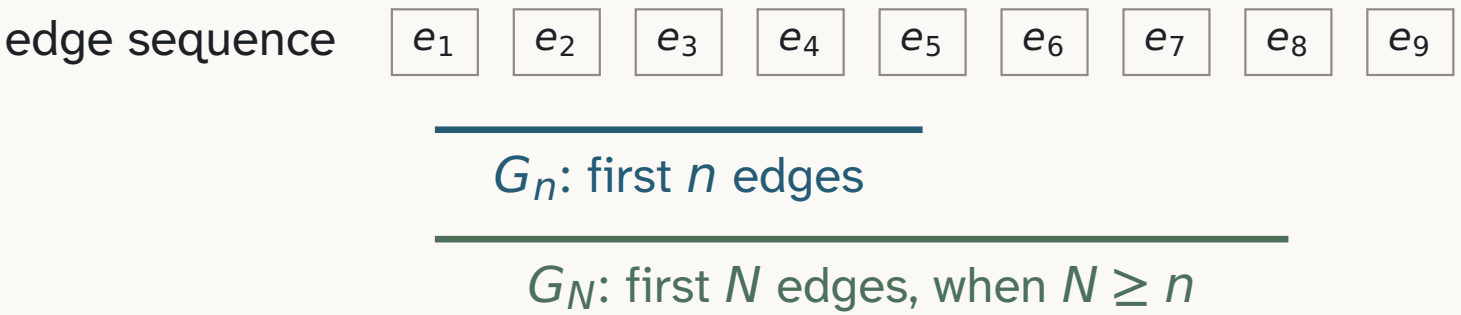


Coupling back to exactly n keys

Generate one infinite sequence of independent random key-edges

$$e_1, e_2, e_3, \ldots$$

and independently draw $N \sim \text{Poisson}(\alpha m)$.

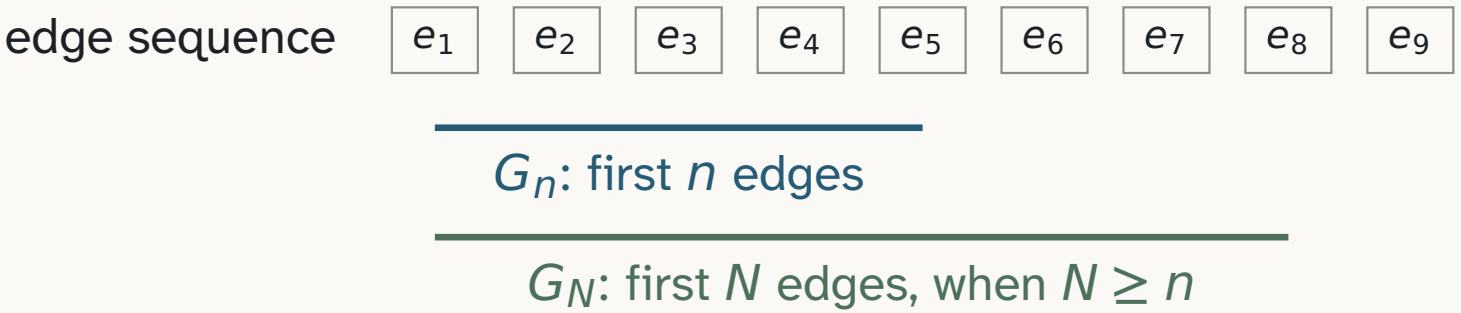


Coupling back to exactly n keys

Generate one infinite sequence of independent random key-edges

$$e_1, e_2, e_3, \ldots$$

and independently draw $N \sim \text{Poisson}(\alpha m)$.



Whenever $N \geq n$, every edge of G_n also appears in G_N, so $G_n \subseteq G_N$. Adding edges can only enlarge connected components.

Why $N \geq n$ with high probability

Here

$$\mu := \mathbb{E}N = \frac{1-\varepsilon/2}{2}m, \qquad n < \frac{1-\varepsilon}{2}m.$$

Why $N \geq n$ with high probability

Here

$$\mu := \mathbb{E}N = \frac{1-\varepsilon/2}{2}m, \qquad n < \frac{1-\varepsilon}{2}m.$$

Thus $n \leq (1-\delta)\mu$ with

$$\delta \geq 1 - \frac{1-\varepsilon}{1-\varepsilon/2} = \frac{\varepsilon}{2-\varepsilon} \geq \frac{\varepsilon}{2}.$$

Why $N \geq n$ with high probability

Here

$$\mu := \mathbb{E}N = \frac{1-\varepsilon/2}{2}m, \qquad n < \frac{1-\varepsilon}{2}m.$$

Thus $n \leq (1-\delta)\mu$ with

$$\delta \geq 1 - \frac{1-\varepsilon}{1-\varepsilon/2} = \frac{\varepsilon}{2-\varepsilon} \geq \frac{\varepsilon}{2}.$$

For $N \sim \mathrm{Poisson}(\mu)$, the standard lower-tail Chernoff bound is

$$\Pr\{N \leq (1-\delta)\mu\} \leq e^{-\mu\delta^2/2}.$$

Why $N \geq n$ with high probability

Here

$$\mu := \mathbb{E}N = \frac{1-\varepsilon/2}{2}m, \qquad n < \frac{1-\varepsilon}{2}m.$$

Thus $n \leq (1-\delta)\mu$ with

$$\delta \geq 1 - \frac{1-\varepsilon}{1-\varepsilon/2} = \frac{\varepsilon}{2-\varepsilon} \geq \frac{\varepsilon}{2}.$$

For $N \sim \mathrm{Poisson}(\mu)$, the standard lower-tail Chernoff bound is

$$\Pr\{N \leq (1-\delta)\mu\} \leq e^{-\mu\delta^2/2}.$$

Since $0 < \varepsilon \leq 1$ gives $\mu \geq m/4$,

$$\boxed{\Pr\{N < n\} \leq e^{-\varepsilon^2 m/32}.}$$

Component tail for the original cuckoo graph

Let A_k be the event $|C(u)| \geq k+1$ in the graph formed by the first n edges. Because N is independent of the edge sequence,

$$\Pr(A_k)\Pr(N \geq n) = \Pr(A_k,\, N \geq n).$$

Component tail for the original cuckoo graph

Let A_k be the event $|C(u)| \geq k+1$ in the graph formed by the first n edges. Because N is independent of the edge sequence,

$$\Pr(A_k)\Pr(N \geq n) = \Pr(A_k,\, N \geq n).$$

On $N \geq n$, the coupling gives $G_n \subseteq G_N$, so

$$\Pr(A_k,\, N \geq n) \leq \Pr_{\text{Pois}}\{|C(u)| \geq k+1\}.$$

Component tail for the original cuckoo graph

Let A_k be the event $|C(u)| \geq k+1$ in the graph formed by the first n edges. Because N is independent of the edge sequence,

$$\Pr(A_k)\Pr(N \geq n) = \Pr(A_k,\, N \geq n).$$

On $N \geq n$, the coupling gives $G_n \subseteq G_N$, so

$$\Pr(A_k,\, N \geq n) \leq \Pr_{\text{Pois}}\{|C(u)| \geq k+1\}.$$

For sufficiently large m, the preceding Chernoff bound gives $\Pr(N \geq n) \geq 1/2$. Therefore

$$\boxed{\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.}$$

Component tail for the original cuckoo graph

Let A_k be the event $|C(u)| \geq k+1$ in the graph formed by the first n edges. Because N is independent of the edge sequence,

$$\Pr(A_k)\Pr(N \geq n) = \Pr(A_k,\, N \geq n).$$

On $N \geq n$, the coupling gives $G_n \subseteq G_N$, so

$$\Pr(A_k,\, N \geq n) \leq \Pr_{\text{Pois}}\{|C(u)| \geq k+1\}.$$

For sufficiently large m, the preceding Chernoff bound gives $\Pr(N \geq n) \geq 1/2$. Therefore

$$\boxed{\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.}$$

For the finitely many smaller values of m, the factor 2 can be replaced by another absolute constant.

The largest component is logarithmic

From the fixed-n tail,

$$\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.$$

The largest component is logarithmic

From the fixed-n tail,

$$\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.$$

Union-bound over the m cells:

$$\Pr\{\max_u |C(u)| \geq k+1\} \leq 2me^{-\varepsilon^2 k/16}.$$

The largest component is logarithmic

From the fixed-n tail,

$$\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.$$

Union-bound over the m cells:

$$\Pr\{\max_u |C(u)| \geq k+1\} \leq 2me^{-\varepsilon^2 k/16}.$$

To make the right-hand side at most m^{-3}, it is enough that

$$k \geq \frac{16}{\varepsilon^2}\big(4\log m + \log 2\big).$$

The largest component is logarithmic

From the fixed-n tail,

$$\Pr\{|C(u)| \geq k+1\} \leq 2e^{-\varepsilon^2 k/16}.$$

Union-bound over the m cells:

$$\Pr\{\max_u |C(u)| \geq k+1\} \leq 2me^{-\varepsilon^2 k/16}.$$

To make the right-hand side at most m^{-3}, it is enough that

$$k \geq \frac{16}{\varepsilon^2}\big(4\log m + \log 2\big).$$

Hence, with probability at least $1 - m^{-3}$,

$$\boxed{\max_u |C(u)| = O(\varepsilon^{-2}\log m).}$$

Two insertion questions

We now have an exponential tail for component size:

$$\Pr\{|C(u)| \geq k\} \leq Ae^{-a\varepsilon^2 k}$$

for absolute constants $A, a > 0$.

Two insertion questions

We now have an exponential tail for component size:

$$\Pr\{|C(u)| \geq k\} \leq Ae^{-a\varepsilon^2 k}$$

for absolute constants $A, a > 0$.
Together with the eviction-walk bound, this answers two algorithmic questions:

Two insertion questions

We now have an exponential tail for component size:

$$\Pr\{|C(u)| \geq k\} \leq Ae^{-a\varepsilon^2 k}$$

for absolute constants $A, a > 0$.
Together with the eviction-walk bound, this answers two algorithmic questions:

1. **Average work.** How many displacements does a typical successful insertion perform?

Two insertion questions

We now have an exponential tail for component size:

$$\Pr\{|C(u)| \geq k\} \leq Ae^{-a\varepsilon^2 k}$$

for absolute constants $A, a > 0$.
Together with the eviction-walk bound, this answers two algorithmic questions:

1. **Average work.** How many displacements does a typical successful insertion perform?
2. **Timeout rule.** How large should the displacement cap be before we give up on the current hash functions and rebuild?

Average insertion work is constant

For a nonnegative integer-valued random variable X,

$$\mathbb{E}X = \sum_{k \geq 1} \Pr\{X \geq k\}.$$

Average insertion work is constant

For a nonnegative integer-valued random variable X,

$$\mathbb{E}X = \sum_{k \geq 1} \Pr\{X \geq k\}.$$

Applying the exponential component tail gives

$$\mathbb{E}|C(u)| \leq A \sum_{k \geq 1} e^{-a\varepsilon^2 k} = O_\varepsilon(1).$$

Average insertion work is constant

For a nonnegative integer-valued random variable X,

$$\mathbb{E}X = \sum_{k \geq 1} \Pr\{X \geq k\}.$$

Applying the exponential component tail gives

$$\mathbb{E}|C(u)| \leq A \sum_{k \geq 1} e^{-a\varepsilon^2 k} = O_\varepsilon(1).$$

A newly inserted key touches at most two pre-existing components, so the component C_x created by adding its edge has the same exponential-tail form up to constants. Thus

$$\mathbb{E}|C_x| = O_\varepsilon(1).$$

Average insertion work is constant

For a nonnegative integer-valued random variable X,

$$\mathbb{E}X = \sum_{k \geq 1} \Pr\{X \geq k\}.$$

Applying the exponential component tail gives

$$\mathbb{E}|C(u)| \leq A \sum_{k \geq 1} e^{-a\varepsilon^2 k} = O_\varepsilon(1).$$

A newly inserted key touches at most two pre-existing components, so the component C_x created by adding its edge has the same exponential-tail form up to constants. Thus

$$\mathbb{E}|C_x| = O_\varepsilon(1).$$

When C_x is tree or unicyclic, the appendix proves $T_x \leq 2|V(C_x)|$. Hence

$$\boxed{\mathbb{E}[T_x;\ C_x \text{ tree or unicyclic}] = O_\varepsilon(1).}$$

Choosing the displacement cap

For a valid component, $T_x \le 2|V(C_x)|$. The component tail therefore gives, for suitable absolute constants $A', a' > 0$,

$$\Pr\{T_x > L,\ C_x \text{ tree or unicyclic}\} \le A' e^{-a'\varepsilon^2 L}.$$

Choosing the displacement cap

For a valid component, $T_x \le 2|V(C_x)|$. The component tail therefore gives, for suitable absolute constants $A', a' > 0$,

$$\Pr\{T_x > L,\ C_x \text{ tree or unicyclic}\} \le A' e^{-a'\varepsilon^2 L}.$$

Suppose we want an unusually long *valid* insertion to trigger a rebuild with probability at most m^{-3}. Solve

$$A' e^{-a'\varepsilon^2 L} \le m^{-3}.$$

Choosing the displacement cap

For a valid component, $T_x \le 2|V(C_x)|$. The component tail therefore gives, for suitable absolute constants $A', a' > 0$,

$$\Pr\{T_x > L,\ C_x \text{ tree or unicyclic}\} \le A' e^{-a'\varepsilon^2 L}.$$

Suppose we want an unusually long *valid* insertion to trigger a rebuild with probability at most m^{-3}. Solve

$$A' e^{-a'\varepsilon^2 L} \le m^{-3}.$$

Taking logs gives the sufficient choice

$$L \ge \frac{3\log m + \log A'}{a'\varepsilon^2} = \Theta(\varepsilon^{-2}\log m).$$

Choosing the displacement cap

For a valid component, $T_x \le 2|V(C_x)|$. The component tail therefore gives, for suitable absolute constants $A', a' > 0$,

$$\Pr\{T_x > L,\ C_x \text{ tree or unicyclic}\} \le A' e^{-a'\varepsilon^2 L}.$$

Suppose we want an unusually long *valid* insertion to trigger a rebuild with probability at most m^{-3}. Solve

$$A' e^{-a'\varepsilon^2 L} \le m^{-3}.$$

Taking logs gives the sufficient choice

$$L \ge \frac{3\log m + \log A'}{a'\varepsilon^2} = \Theta(\varepsilon^{-2}\log m).$$

So a logarithmic displacement cap makes false timeouts very rare while keeping every insertion attempt bounded.

The complete cuckoo-hashing algorithm

State: table $A[1{:}m]$, hash functions h_1, h_2, and cap

$$L = \lceil c_0 \varepsilon^{-2} \log m \rceil$$

for a sufficiently large absolute constant c_0.
Lookup (x)**:** inspect $A[h_1(x)]$ and $A[h_2(x)]$.

The complete cuckoo-hashing algorithm

State: table $A[1{:}m]$, hash functions h_1, h_2, and cap

$$L = \lceil c_0 \varepsilon^{-2} \log m \rceil$$

for a sufficiently large absolute constant c_0.
Lookup (x)**:** inspect $A[h_1(x)]$ and $A[h_2(x)]$.
Insert (x)**:**

1. Set $w \leftarrow x$ and choose one candidate cell of w.
2. If that cell is empty, store w there and stop.
3. Otherwise swap w with the resident key. Move the evicted key to its other candidate cell and continue.
4. If L displacements occur without reaching an empty cell, keep the current in-hand key w, draw fresh h_1, h_2, rebuild from the keys now in the table together with w, and retry.

The complete cuckoo-hashing algorithm

State: table $A[1{:}m]$, hash functions h_1, h_2, and cap

$$L = \lceil c_0 \varepsilon^{-2} \log m \rceil$$

for a sufficiently large absolute constant c_0.
Lookup (x)**:** inspect $A[h_1(x)]$ and $A[h_2(x)]$.
Insert (x)**:**

1. Set $w \leftarrow x$ and choose one candidate cell of w.
2. If that cell is empty, store w there and stop.
3. Otherwise swap w with the resident key. Move the evicted key to its other candidate cell and continue.
4. If L displacements occur without reaching an empty cell, keep the current in-hand key w, draw fresh h_1, h_2, rebuild from the keys now in the table together with w, and retry.

Delete (x)**:** inspect the two candidate cells and clear the one containing x.

Cuckoo-hashing guarantees

Assume fully random distinct candidate cells and at most $(1-\varepsilon)(m-1)/2$ keys in the table.

Operation/event	Guarantee	Reason
Lookup	2 probes, worst case	two choices
Delete	$O(1)$, worst case	two choices
Valid insertion	$O_\varepsilon(1)$ expected	tail + walk lemma
One attempt	$O_\varepsilon(\log m)$ worst case	cap
False timeout	$\leq m^{-3}$	cap choice
Bad hash pair	$O_\varepsilon(1/m)$	witness count
Space	$m \gtrsim 2n/(1-\varepsilon)$	load margin

For an insertion-only run of $\Theta(m)$ keys, one fixed hash pair is bad with probability $O_\varepsilon(1/m)$. Fresh retries therefore do not change the order of the total work, so the expected amortized insertion cost remains $O_\varepsilon(1)$; the retry bookkeeping is in the appendix.
If the table becomes too full, double its size and rehash. If it becomes much emptier after deletions, shrink it using a lower threshold.

Two-choice chaining vs. cuckoo hashing

	Two-choice chaining	Cuckoo hashing
Bucket capacity	unbounded	one key per cell
Worst-case lookup	$\Theta(\log \log n)$ probes w.h.p.	2 probes, always
Insertion	$O(1)$, never moves keys	$O_\varepsilon(1)$ exp. amortized; moves keys
Failure mode	none (buckets grow)	timeout $\Rightarrow$ rebuild
Space	n buckets + list nodes	$> 2n$ cells, over half empty

Cuckoo hashing gets worst-case constant lookup by doing more work during insertion: keys may move, and occasionally the table must be rebuilt.

Cuckoo hashing: the proof in one view

- Cuckoo hashing turns each key into an edge between its two candidate cells.
- A placement exists exactly when every connected component has at most one cycle: $|E(C)| \leq |V(C)|$.
- If a connected component has more edges than vertices, some key cannot be stored. Reducing such a component to a bicycle witness and counting all witnesses gives failure probability $O_\varepsilon(1/m)$ below load one.
- Below average degree one, component exploration is subcritical: large components have exponentially small probability.
- Therefore insertions have constant expected local work, while a logarithmic displacement cap makes timeouts on valid components very unlikely.

Further reading

- R. Pagh and F. F. Rodler, “Cuckoo Hashing,” *Journal of Algorithms* 51 (2004), 122–144.
- L. Devroye and P. Morin, “Cuckoo hashing: Further analysis,” *Information Processing Letters*, 2003.
- M. Mitzenmacher and E. Upfal, *Probability and Computing*, 2nd ed., Cambridge University Press, 2017. Chapter 5 covers Poissonization.

Questions?

PART A

Student appendix: proofs worth reading

A

How to use this appendix

The main lecture leaves four technical points to these slides. They are organized so that each part can be read on its own.

1. **The eviction walk:** why a valid component is found in at most $2|V(C)|$ displacements.
2. **Small components:** how the sparse-graph homework bound applies to the cuckoo graph.
3. **Timeouts and rebuilds:** how the component tail determines $L = \Theta_\varepsilon(\log m)$ and why rebuilding is cheap for insertion-only runs.
4. **Resizing:** why growing and shrinking the table adds only constant amortized work.

All notation needed below is restated where it is used.

1. The eviction walk: setup

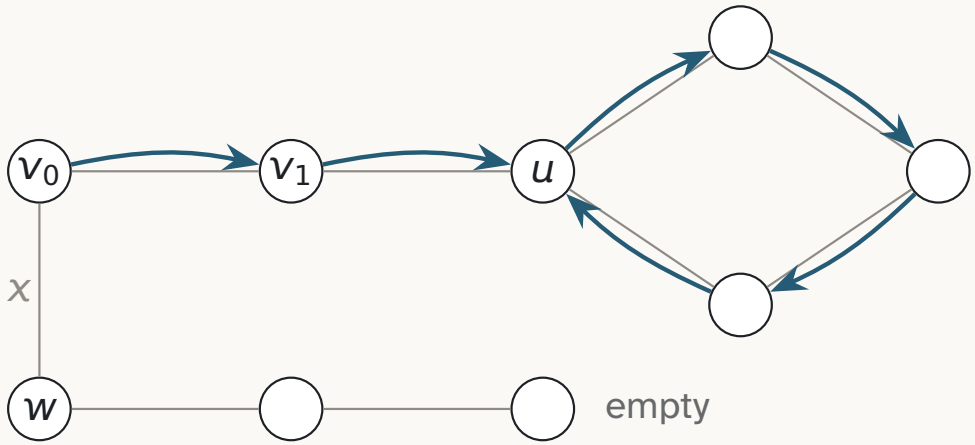
During an insertion, exactly one key is “in hand.” If its chosen cell is occupied, the incoming key takes that cell and the displaced key moves to its other candidate cell. This produces a walk through cells of the cuckoo graph.

Walk lemma

Let C be the component containing the new key x after x is added. If C is a tree or has exactly one cycle, the eviction walk reaches an empty cell after at most $2|V(C)|$ displacements.

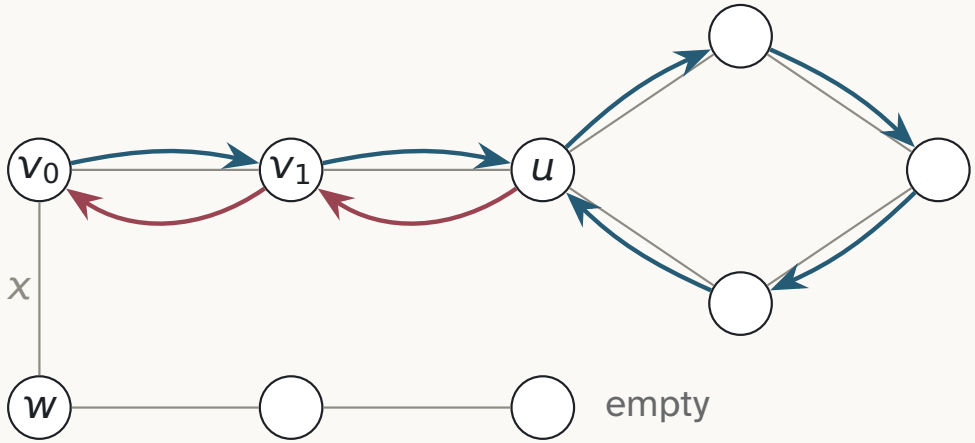
The key local fact is simple: the key evicted from a cell is not the key that just arrived there. Hence the walk cannot immediately traverse the same key-edge backward. Therefore the first repeated cell closes a genuine cycle.

After the walk finds the unique cycle



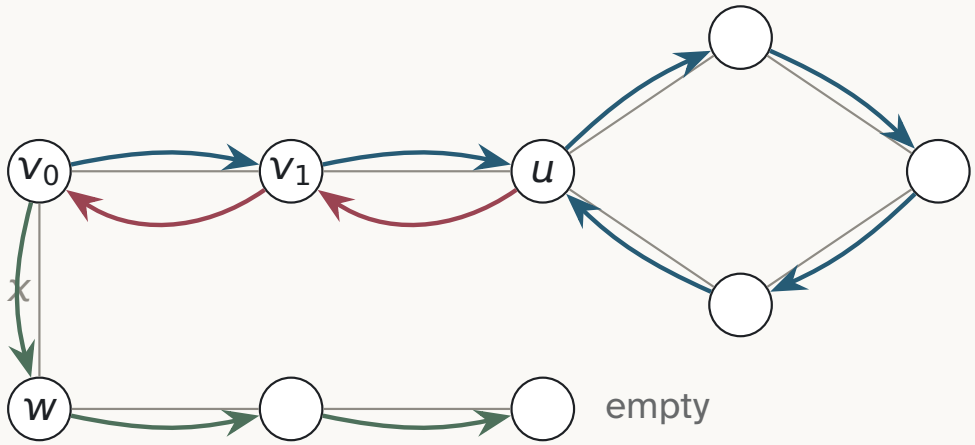
Forward. Starting from v_0, the walk reaches the unique cycle at u, goes around it, and returns to u.

After the walk finds the unique cycle



Retrace. On the second visit to u, the walk evicts the key that arrived on the first visit. The same thing then happens at v_1, v_0, . . .: each key returns to the cell it occupied before this insertion began. Eventually x is evicted from v_0.

After the walk finds the unique cycle



New path. Key x moves to its other candidate cell w. If this part of the walk repeated a cell, it would create a second cycle. Hence it must eventually reach an empty cell.

Proof of the walk lemma

Let $P = (v_0, v_1, \ldots)$ be the cells visited. If no cell repeats, there are at most $|V(C)|$ visits.
Otherwise let $u = v_i$ be the first cell to repeat, with second visit v_{i+c}. Because the walk never immediately reverses an edge, $v_i, \ldots, v_{i+c}$ traces a cycle $\mathcal{C}$.

- **Retracing the incoming path.** On the second visit to u, the evicted key is the key that arrived on the first visit. Inductively, the walk visits $v_{i-1}, \ldots, v_0$, restoring the keys displaced on the way to $\mathcal{C}$, and then carries x to its other cell w.
- **After x moves to w.** If the walk now reaches any previously visited cell, or repeats a cell within this new part of the walk, it closes a cycle different from $\mathcal{C}$. That would give C at least two cycles, contradicting the hypothesis.

The cells in the forward part and the cells in the retracing/new part are each distinct. Each part therefore contains at most $|V(C)|$ visits, so the total is at most $2|V(C)|$. □

2. A sparse-graph result we will reuse

A homework problem proves the following fact for subcritical random graphs. You do not need to have completed that problem in order to use the result here.

Subcritical component bound

Consider $G(N, c/N)$ with $c < 1$. Start BFS from a vertex v, and let $C(v)$ be its connected component. Then

$$\Pr\{|C(v)| \geq k+1\} \leq \exp\left(-\frac{(1-c)^2}{4}k\right)$$

up to harmless absolute constants.

The proof follows the BFS queue. When a vertex is processed, the number of previously unseen neighbors has a binomial/Chernoff tail with mean at most $c < 1$. It is unlikely for the queue to survive for k steps when the average number of new vertices per step is below one.

Why we Poissonize before applying the theorem

In the fixed-n cuckoo graph, a direct conditional calculation is awkward: after BFS has ruled out some cells as endpoints of still-unseen keys, those remaining endpoint choices are not literally uniform over all m cells. Introduce a slightly denser Poissonized model instead. Let

$$\alpha := \frac{1 - \varepsilon/2}{2}, \qquad N \sim \mathrm{Poisson}(\alpha m),$$

and let the N keys choose two distinct uniform candidate cells. Since $n/m \le (1-\varepsilon)/2 < \alpha$, this model has a little more density than the original one while remaining subcritical.
By Poisson splitting, the numbers of keys on the m^2 possible ordered cell pairs are independent $\mathrm{Poisson}(\alpha/m)$ random variables. This restores exactly the independence needed by the homework exploration argument.

Poissonized cuckoo graph: one BFS step

Process one cell w during BFS. Conditional on everything exposed so far, the edge counts from w to still-unexposed cells are independent Poisson variables. Their total has mean at most $\mu := 2\alpha = 1 - \varepsilon/2$.
If Y_t is the number of genuinely new cells discovered at this step, then it is bounded by a Poisson random variable Z_t with mean at most μ.
Therefore

$$\begin{aligned}\mathbb{E}[\, e^{\theta Y_t} \mid \mathcal{F}_{t-1}] &\le \mathbb{E}[\, e^{\theta Z_t} \mid \mathcal{F}_{t-1}] \\ &\le \exp\bigl(\mu(e^{\theta} - 1)\bigr).\end{aligned}$$

The homework theorem applies with $c = \mu = 1 - \varepsilon/2$. Hence, in the Poissonized graph,

$$\Pr\{|C_P(u)| \ge k\} \le Ce^{-c\varepsilon^2 k}$$

for absolute constants $C, c > 0$.

Return to exactly n keys

Couple the two models using an infinite sequence of i.i.d. key-edges $e_1, e_2, \ldots$, independent of N. The fixed-n graph uses the first n edges; the Poissonized graph uses the first N.
The event $A_k = \{|C(u)| \geq k\}$ is increasing: adding edges cannot shrink a component. On $\{N \geq n\}$, the fixed-n graph is a subgraph of the Poissonized graph, so

$$\Pr(A_k \text{ with } n \text{ keys}) \leq \frac{\Pr(A_k \text{ with } N \text{ keys})}{\Pr(N \geq n)}.$$

Because $\mathbb{E}N = \alpha m > n$ by a constant fraction, a Poisson lower-tail Chernoff bound gives $\Pr(N \geq n) \geq 1/2$ for all sufficiently large m. Thus

$$\Pr\{|C(u)| \geq k\} \leq 2Ce^{-c\varepsilon^2 k}.$$

For the finitely many smaller m, absorb the difference into the constant.

Component-size consequences

Write the fixed-n tail as

$$\Pr\{|C(u)| \geq k\} \leq C_\varepsilon e^{-c_\varepsilon k}, \qquad c_\varepsilon = \Omega(\varepsilon^2).$$

A union bound over the m cells gives

$$\Pr\left\{\max_u |C(u)| \geq \frac{3\log m + \log(C_\varepsilon)}{c_\varepsilon}\right\} \leq m^{-2}.$$

The same tail gives constant expected component size:

$$\mathbb{E}|C(u)| = \sum_{k\geq 1} \Pr\{|C(u)| \geq k\} \leq C_\varepsilon \sum_{k\geq 1} e^{-c_\varepsilon k} = O_\varepsilon(1).$$

These are the two component facts used in the insertion analysis.

3. From component size to insertion work

Let K_x be the number of cells in the component containing a newly inserted key x, and let T_x be the number of displacements in the uncapped eviction walk. Before x is added, its two candidate cells lie in at most two components. Therefore the same exponential tail gives

$$\Pr\{K_x \geq k\} \leq C'_\varepsilon e^{-c'_\varepsilon k}, \qquad \mathbb{E}K_x = O_\varepsilon(1)$$

for suitable constants $C'_\varepsilon, c'_\varepsilon > 0$.
If the resulting component has at most one cycle, the walk lemma gives $T_x \leq 2K_x$. Hence

$$\Pr\{T_x > L \text{ and a valid placement exists}\} \leq C'_\varepsilon e^{-c'_\varepsilon L/2}.$$

To make this probability at most m^{-3}, it is enough to choose

$$L_m = \left\lceil \frac{2}{c'_\varepsilon}\left(3\log m + \log C'_\varepsilon\right) \right\rceil = \Theta_\varepsilon(\log m).$$

This is why the implementation uses a logarithmic displacement cap.

Expected work of one capped insertion attempt

Let $\widehat{T}_x := \min\{T_x, L_m\}$ be the number of displacements actually performed. There are two possibilities.

- If a valid placement exists, $\widehat{T}_x \le 2K_x$.
- If no valid placement exists, the cap gives $\widehat{T}_x \le L_m$.

Thus

$$\widehat{T}_x \le 2K_x + L_m \mathbf{1}\{\text{no valid placement}\}.$$

The witness-counting argument from the main lecture gives $\Pr\{\text{no valid placement}\} = O_\varepsilon(1/m)$. Therefore

$$\mathbb{E}\widehat{T}_x \le 2\mathbb{E}K_x + L_m\, O_\varepsilon(1/m) = O_\varepsilon(1).$$

So one capped insertion attempt has constant expected displacement cost, even though its worst-case cost is $L_m = O_\varepsilon(\log m)$.

Why $O(1/m)$ is not paid per insertion

For one insertion,

$$\begin{aligned}\{\text{timeout}\} \subseteq\ & \{\text{no valid placement}\} \\ & \cup \{T_x > L_m \text{ although a valid placement exists}\}.\end{aligned}$$

The first event has probability $O_\varepsilon(1/m)$ and the second at most m^{-3}. It would be wrong, however, to union-bound the $O(1/m)$ term over all $\Theta(m)$ insertions made with one hash pair. If an insertion first creates a component with more edges than vertices, that obstruction remains in every later insertion-only graph. Hence

$$\begin{aligned}&\{\text{some insertion creates an impossible graph}\} \\ &\quad = \{\text{the final graph is impossible}\}.\end{aligned}$$

and the witness-counting theorem bounds this single event by $O_\varepsilon(1/m)$.

One insertion-only run with a fixed hash pair

Fix the capacity m, draw a fresh pair (h_1, h_2), and insert a predetermined sequence of $r \le (1-\varepsilon)(m-1)/2$ keys. Call this one *attempt*.
An attempt can fail for only two reasons:

1. the final cuckoo graph has a component with more edges than vertices, with probability $O_\varepsilon(1/m)$;
2. some insertion has a valid component of size at least $L_m/2$, whose probability is at most $rm^{-3} \le m^{-2}$.

Thus an attempt fails with probability

$$q_m = O_\varepsilon(1/m).$$

Until success or the first timeout, the expected work is

$$O_\varepsilon(r + \log m),$$

because the r local insertion costs have constant expectation and at most one timeout contributes L_m additional work.

Rebuilding after a timeout

Suppose the first attempt times out after the first i keys. The implementation now draws an independent new hash pair, reinserts those same i keys, and then continues with the remaining keys. Relative to the fixed insertion sequence, this is a fresh attempt with independent randomness.
Let Γ_a be the work in attempt a, and let A be the number of attempts. Since attempt a is started only if the previous $a-1$ attempts failed,

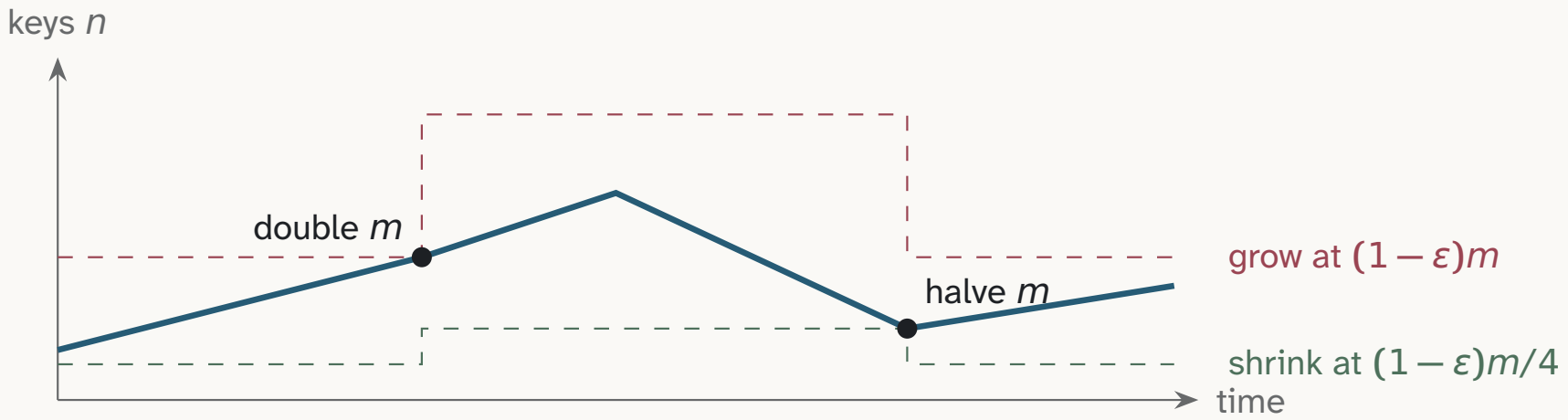
$$\Pr(A \geq a) \leq q_m^{a-1},$$

and the fresh randomness in attempt a is independent of the event $\{A \geq a\}$. Therefore

$$\mathbb{E}\left[\sum_{a=1}^{A} \Gamma_a\right] = \sum_{a \geq 1} \Pr(A \geq a)\, \mathbb{E}\Gamma_a \leq \frac{O_\varepsilon(r + \log m)}{1 - q_m}.$$

For $r = \Theta_\varepsilon(m)$ this is $O_\varepsilon(m)$ total expected work, or $O_\varepsilon(1)$ amortized work per insertion.

4. Resizing: use separate thresholds



- Grow when an insertion would make $n > (1-\varepsilon)m$: double m, draw fresh hashes, and rebuild.
- Shrink only when $n < (1-\varepsilon)m/4$: halve m and rebuild.

Using different thresholds leaves the table far from the opposite threshold after a resize. Consequently $\Theta_\varepsilon(m)$ updates must occur before the next resize, while rebuilding costs $O_\varepsilon(m)$ expected work. Resizing therefore contributes $O_\varepsilon(1)$ amortized work per update.

PART B

Complete technical appendix

B

How this appendix is organized

This appendix records two arguments that are useful for completeness but are not needed in the 110-minute lecture.

1. **An alternative component-size proof.** Instead of invoking the homework result, we prove the exponential component tail directly by Poissonization and Chernoff bounds.
2. **Long insert/delete sequences.** We first analyze a scheduled-rebuild variant, where every hash pair used in an epoch is fresh, and then explain the extra conditioning that appears in the standard implementation.

Each subsection restates the assumptions it needs. The second part is substantially more technical than the student appendix.

I. Alternative proof of the component-size bound

The main lecture obtains the component-size bound from the homework. Here is a separate proof using Poissonization.

Start from a fixed cell u. Since each of the n keys chooses u as one of its two candidate cells with probability at most $2/m$, its expected degree is at most $2n/m \leq 1 - \varepsilon$. If successive BFS generations behaved independently, one would expect

$$1 + \frac{2n}{m} + \left(\frac{2n}{m}\right)^2 + \cdots \leq \frac{1}{\varepsilon}$$

cells in the component.

Two issues remain: conditioning on the BFS history changes the law of the unexplored graph, and an expectation calculation alone does not give the exponential tail needed for a logarithmic timeout. Poissonization restores independence; Chernoff gives the tail.

Poissonization makes edge counts independent

Set $\alpha := (1 - \varepsilon/2)/2$. In the Poissonized model, take

$$N \sim \mathrm{Poisson}(\alpha m)$$

keys, each choosing two distinct uniform candidate cells.
Poisson splitting says that the number of keys on each ordered pair of distinct cells is an independent $\mathrm{Poisson}(\alpha/(m-1))$ random variable. Consequently, after any collection of ordered cell pairs has been exposed, the counts on all remaining pairs are still independent Poisson variables. In particular, the number of as-yet-unexposed edges incident to a processed cell is stochastically dominated by a Poisson random variable with mean

$$2(m-1) \cdot \frac{\alpha}{m-1} = 2\alpha = 1 - \varepsilon/2 < 1.$$

We chose $\alpha > n/m$ so that returning to the fixed-n model will only require a lower-tail bound for N.

Explore the Poissonized component with a queue

Start with queue $[u]$. Repeatedly remove the first cell c, expose the edge counts from c to unexposed cells whose pairs have not yet been exposed, and add any newly discovered cells to the queue.
Let B'_j be the number of edges exposed at expansion j. Conditional on the entire past, B'_j is Poisson with mean at most μ. We may add independent dummy Poisson counts so that

$$B'_j \le B_j, \qquad B_1, B_2, \ldots \overset{\text{i.i.d.}}{\sim} \text{Poisson}(\mu).$$

The number of new cells discovered at step j is at most B'_j, hence at most B_j.
If the component contains at least k cells, the queue is still nonempty after $k-1$ expansions. Therefore

$$\sum_{j=1}^{k-1} B_j \ge k-1.$$

The sum is $\text{Poisson}(\mu(k-1))$, whose mean is strictly below the threshold $k-1$.

Chernoff gives the Poissonized tail

For $Y \sim \mathrm{Poisson}(\lambda)$ and $a \geq \lambda$,

$$\Pr(Y \geq a) \leq e^{-\lambda} \left(\frac{e\lambda}{a} \right)^a .$$

Take $\lambda = \mu r$ and $a = r$, where $r = k - 1$. Then

$$\Pr\{S_P(u) \geq k\} \leq \exp[-rI(\mu)], \qquad I(\mu) := \log \frac{1}{\mu} - (1 - \mu).$$

For $\mu = 1 - \varepsilon/2$, the inequality $-\log(1-t) - t \geq t^2/2$ gives

$$I(\mu) \geq \frac{\varepsilon^2}{8}.$$

Hence

$$\Pr\{S_P(u) \geq k\} \leq e^{-\varepsilon^2(k-1)/8}.$$

Here $S_P(u)$ denotes the component size in the Poissonized graph.

Monotone depoissonization

We now return to exactly n keys. Couple both models by drawing an infinite sequence of i.i.d. key-edges $e_1, e_2, \ldots$, independently of $N \sim \mathrm{Poisson}(\alpha m)$. The fixed-$n$ graph uses the first n edges; the Poissonized graph uses the first N.
Call an event A *increasing* if adding edges cannot make it false. On $\{N \geq n\}$, the fixed-n graph is a subgraph of the Poissonized graph, so

$$\Pr(A_n) = \Pr(A_n \mid N \geq n) \leq \Pr(A_N \mid N \geq n) \leq \frac{\Pr(A_N)}{\Pr(N \geq n)}.$$

The event $\{S(u) \geq k\}$ is increasing. Since $n \leq (1-\varepsilon)(m-1)/2 < \alpha m$, a Poisson lower-tail Chernoff bound gives $\Pr(N \geq n) \geq 1/2$ for all sufficiently large m. Therefore

$$\Pr\{S(u) \geq k\} \leq 2e^{-\varepsilon^2(k-1)/8}.$$

For the finitely many smaller m, enlarge the constant.

Why the monotone coupling is useful

A cruder way to return from a Poissonized model is to condition on $N = n$. Around its mean, $\Pr(N = n) = \Theta(n^{-1/2})$, so this can multiply a tail probability by $O(\sqrt{n})$.
Applying that crude factor to the exponential tail would still prove an $O(\log n)$ largest-component bound, but summing the resulting tail yields only

$$\mathbb{E}S(u) = O\left(\frac{\log n}{\varepsilon^2}\right).$$

It is therefore too weak to prove constant expected insertion work.
The monotone coupling loses only a constant factor:

$$\mathbb{E}S(u) = \sum_{k \geq 1} \Pr\{S(u) \geq k\} \leq 1 + 2\sum_{j \geq 0} e^{-\varepsilon^2 j/8} = O(\varepsilon^{-2}).$$

Thus the same proof gives both the exponential tail and the constant expected component size.

II. Dynamic updates: setup

Fix a capacity m and a fixed-in-advance update sequence $o_1, \ldots, o_T$. Let S_t be the set of keys after operation o_t, and assume

$$|S_t| \le (1-\varepsilon)m \qquad \text{for every } t.$$

Each resize begins a new fixed-capacity period with fresh hash functions, so it is enough to analyze one such period.
Let $L = L_m$ be the displacement cap and set

$$w := \lfloor \varepsilon m/4 \rfloor.$$

If U is the union of the keys present during any w consecutive updates, then

$$|U| \le (1-\varepsilon)m + w \le (1-3\varepsilon/4)m.$$

Write $\varepsilon' := 3\varepsilon/4$. Call a hash pair *good for* U if every component of $G_h(U)$ has at most one cycle and fewer than $L/2$ cells.

What a good hash pair guarantees

Two facts proved earlier are being used here:

1. if every component has at most one cycle, every subset of those keys admits a placement;
2. if the relevant component has fewer than $L/2$ cells, the walk lemma gives fewer than L displacements.

Therefore, if h is good for U, no insertion involving only keys from U can time out.
For a fresh hash pair,

$$\Pr\{h \text{ is not good for } U\} \le q_m := \frac{c_{\varepsilon'} + 1}{m}.$$

Indeed, witness counting contributes at most $c_{\varepsilon'}/m$. For the size condition, the fixed-n tail and a union bound over the m cells give

$$m \cdot 2e^{-\varepsilon'^2(L/2-1)/8} \le \frac{1}{m}$$

for the chosen $L = L_m$ and sufficiently large m.

III. A scheduled-rebuild variant

Split the updates into epochs of w consecutive operations.

1. At the start of every epoch, draw fresh hash functions and rebuild, whether or not the previous epoch had a timeout.
2. Within the epoch, use the ordinary capped insertion rule; after a timeout, draw another fresh pair and rebuild.

Fix one epoch and let U_e be the union of all keys appearing during it. The set U_e is determined by the fixed update sequence, not by the hash functions used in the epoch, and $|U_e| \le (1 - \varepsilon')m$.
Thus every newly drawn hash pair is independent of the fixed set U_e. If it is good for U_e, it cannot time out anywhere in the epoch. This is the reason the scheduled variant is easy to analyze.

Expected cost of one epoch

For each draw j in the epoch, let I_j be the indicator that draw j is started and let W_j be its work. Because draw j is fresh and independent of the previous failures,

$$\Pr(I_j = 1) \le q_m^{j-1}.$$

Conditioned on $I_j = 1$, the new hash pair is fresh. A rebuild plus at most w ordinary updates contains at most m insertions, each with $O_\varepsilon(1)$ expected local work, plus at most one timeout of cost L. Hence

$$\mathbb{E}[\, W_j \mid I_j = 1\,] = O_\varepsilon(m).$$

Therefore,

$$\mathbb{E}[\,\text{epoch work}\,] = \sum_{j \ge 1} \mathbb{E}[I_j W_j] \le O_\varepsilon(m) \sum_{j \ge 1} q_m^{j-1} = \frac{O_\varepsilon(m)}{1 - q_m} = O_\varepsilon(m).$$

Since an epoch contains $w = \Theta(\varepsilon m)$ updates, the expected amortized cost is $O_\varepsilon(1)$ per update.

IV. Standard implementation: conditioning

The standard implementation rebuilds only after a timeout or resize. A hash pair may therefore remain in use for a long time.
At time t, the pair currently in use has been selected by the event that all earlier operations under that pair avoided a timeout. Its distribution need not be the same as that of a fresh pair.
We will use three steps:

1. quantify how much selecting the current pair can increase the expectation of a nonnegative statistic;
2. show that the number of hash pairs tried by time T is only $O_\varepsilon(T/m + 1)$ in expectation;
3. treat reinserted keys and never-before-seen keys separately.

This conditioning issue is absent from the scheduled-rebuild variant because each epoch starts with independent fresh randomness.

Index the hash pairs that are tried

Fix an infinite i.i.d. sequence of fully random candidate-pair functions

$$h^{(1)}, h^{(2)}, \ldots .$$

The implementation starts with $h^{(1)}$ and uses the next pair whenever it rebuilds with fresh hashes.
Let D_t be the index of the pair in use when operation o_t begins, and write

$$h_t := h^{(D_t)}.$$

The event $\{D_t \geq j\}$ says that pair j has been reached by time t. Whether this happens is determined entirely by the fixed update sequence and the earlier pairs $h^{(1)}, \ldots, h^{(j-1)}$. Consequently

$$\{D_t \geq j\} \quad \text{is independent of} \quad h^{(j)}.$$

This elementary independence is the bookkeeping fact used in the next lemma.

Selection lemma

Selection lemma

Let $f \geq 0$ be any function of a hash pair; it may also depend on t and on the fixed update sequence. If h denotes a fresh pair, then

$$\mathbb{E} f(h_t) \leq \mathbb{E}[D_t]\, \mathbb{E} f(h).$$

In particular, for any event B depending on one hash pair,

$$\Pr(h_t \in B) \leq \mathbb{E}[D_t] \Pr(h \in B).$$

Proof. Since h_t is one of $h^{(1)}, \ldots, h^{(D_t)}$ and $f \geq 0$,

$$f(h_t) \leq \sum_{j \geq 1} \mathbf{1}\{D_t \geq j\} f(h^{(j)}).$$

Taking expectations and using the independence above,

$$\mathbb{E} f(h_t) \leq \sum_{j \geq 1} \Pr(D_t \geq j) \mathbb{E} f(h) = \mathbb{E}[D_t] \mathbb{E} f(h). \qquad \square$$

How many hash pairs are tried?

Call draw j *short* if, after it begins, it fails during its rebuild or before the next w operations have completed.

Lifetime bound

$$\mathbb{E}[D_T] \le \frac{T/w + 1}{1 - q_m} = O_\varepsilon\left(\frac{T}{m} + 1\right).$$

Proof. Suppose draw j starts at time τ. Let U contain the keys present at time τ together with every key inserted during the next w operations. Conditional on all earlier draws, both τ and U are fixed, while $h^{(j)}$ is fresh. If $h^{(j)}$ is good for U, draw j survives its rebuild and those w operations. Hence

$$\Pr(j \text{ is short} \mid D_T \ge j) \le q_m.$$

So the expected number of short draws started by time T is at most $q_m \mathbb{E} D_T$. Non-short draws occupy disjoint blocks of at least w operations, so at most $T/w + 1$ such draws can start. Rearranging gives the claim. □

Rebuilds and timeouts

When draw j starts, it rebuilds at most $(1-\varepsilon)m$ keys using the fresh pair $h^{(j)}$. Conditional on the earlier draws, the expected local work of each rebuild insertion is $O_\varepsilon(1)$, and at most one timeout contributes an additional L. Thus

$$\mathbb{E}[\,W_j \mid D_T \geq j\,] \leq C_\varepsilon m + L$$

for the rebuild-and-timeout work W_j of draw j.
Using the same indicator-sum argument as before,

$$\begin{aligned}\mathbb{E}[\,\text{rebuild and timeout work}\,] &= \sum_{j\geq 1} \Pr(D_T \geq j)\,\mathbb{E}[\,W_j \mid D_T \geq j\,] \\ &\leq (C_\varepsilon m + L)\mathbb{E}[\,D_T\,] \\ &= O_\varepsilon(T+m).\end{aligned}$$

No assumption is needed that the pair currently in use is fresh; freshness is used only at the moment a new draw begins.

Ordinary insertions: new keys versus reinsertions

Consider an insertion $o_t = \mathrm{Insert}(x)$ performed under the current pair h_t. Timeout work has already been counted. If the insertion succeeds,

$$\mathrm{cost}_t \le \min\{L, 2K_t\},$$

where K_t is the size of the component containing x after insertion. For a fresh hash pair this cost has constant expectation. Under the selected pair h_t, distinguish two cases:

- **Reinsertion:** x appeared earlier, so its hash values may have influenced which pair survived.
- **New key:** x has never appeared before, so its two hash values under h_t have never been queried and remain independent uniform cells, even conditional on the surviving pair's behavior on earlier keys.

The second case has more fresh randomness and admits a sharper bound.

Reinsertions: a logarithmic bound

For $a \geq 1$, let $B_a = \{2K_t \geq a\}$. The fresh-pair component tail and the selection lemma give

$$\Pr\{2K_t(h_t) \geq a\} \leq \mathbb{E}[D_t]\, C_\varepsilon e^{-c_\varepsilon a}.$$

Hence, using the tail-sum formula for the capped cost,

$$\mathbb{E}\,\mathrm{cost}_t \leq \sum_{a=1}^{L} \min\{1, \mathbb{E}[D_t] C_\varepsilon e^{-c_\varepsilon a}\}.$$

Split the sum at

$$a_0 := \left\lceil c_\varepsilon^{-1} \log(C_\varepsilon \mathbb{E} D_t) \right\rceil_+ .$$

The first a_0 terms contribute at most a_0 and the remaining geometric tail contributes $O_\varepsilon(1)$. Using the lifetime bound,

$$\mathbb{E}\,\mathrm{cost}_t = O_\varepsilon\big(\log(2 + t/m)\big),$$

with the trivial cap $\mathrm{cost}_t \leq L = O_\varepsilon(\log m)$ always available.

New keys: average over their fresh endpoints

Suppose x has never appeared before time t. Let u, v be its two candidate cells. Conditional on everything the algorithm has seen before inserting x, the pair (u, v) is still uniform and independent of the graph on S_{t-1}.
If $S_h(c)$ denotes the size of the component of cell c in $G_h(S_{t-1})$, then

$$\mathrm{cost}_t \le 2\min\{L, S_{h_t}(u)\} + 2\min\{L, S_{h_t}(v)\}.$$

Define

$$\Phi_t(h) := \sum_{\text{cells } c} \min\{L, S_h(c)\}.$$

Averaging over the two fresh endpoints gives

$$\mathbb{E}[\mathrm{cost}_t \mid h_t] \le \frac{2}{m}\Phi_t(h_t).$$

For a fresh pair, the component tail gives $\mathbb{E}\Phi_t(h) = O_\varepsilon(m)$. It remains to control how much selecting h_t can change this global statistic.

Why Φ_t has bounded differences

The function Φ_t depends on the independent endpoint pairs of the keys in S_{t-1}, at most m independent inputs.
Change the endpoint pair of one key. This replaces one graph edge by another.

1. Deleting the old edge can affect only its old connected component, splitting it into at most two components.
2. Adding the new edge can touch at most two components of the intermediate graph and merge them.

Thus all cells whose component size can change lie in a constant number (at most six, using a loose count) of components before or after the change.
A term $\min\{L, S_h(c)\}$ changes only if one of the relevant before/after components has fewer than L cells; otherwise both values equal L. Hence at most $6L$ terms can change, each by at most L, and

$$|\Phi_t(h) - \Phi_t(h')| \le 6L^2.$$

McDiarmid gives concentration of Φ_t

We use the following standard bounded-differences inequality. If F is a function of independent inputs and changing input i changes F by at most c_i, then

$$\Pr\{F - \mathbb{E}F \geq s\} \leq \exp\left(-\frac{2s^2}{\sum_i c_i^2}\right).$$

For Φ_t, there are at most m inputs and each has $c_i \leq 6L^2$. Taking $s = m$ gives

$$\Pr\{\Phi_t(h) \geq \mathbb{E}\Phi_t + m\} \leq \exp\left(-\frac{2m^2}{m(6L^2)^2}\right) = e^{-m/(18L^4)}.$$

This statement is included here so the argument does not depend on having already reached the later lecture on bounded differences.

New keys: constant expected cost

Let

$$B := \{\Phi_t \geq \mathbb{E}\Phi_t + m\}.$$

Since $\Phi_t \leq 2mL$, we have $(2/m)\Phi_t \leq 4L$. Therefore

$$\frac{2}{m}\Phi_t(h_t) \leq \frac{2}{m}(\mathbb{E}\Phi_t + m) + 4L\,\mathbf{1}\{h_t \in B\}.$$

Taking expectations and applying the selection lemma only to the rare event B,

$$\mathbb{E}\,\mathrm{cost}_t \leq O_\varepsilon(1) + 4L\,\mathbb{E}[D_t]e^{-m/(18L^4)}.$$

The second term is at most 1 whenever

$$\mathbb{E}[D_t] \leq \frac{e^{m/(18L^4)}}{4L}.$$

For sufficiently large m, the lifetime bound makes this true throughout

$$T \leq \exp\left(\frac{m}{36L_m^4}\right).$$

For fixed ε, this horizon is superpolynomial in m.

Total expected work through time T

Let R_T be the number of insertions by time T that reinsert a key seen earlier.
For every T, the selection-lemma bound applies to every insertion, so

$$\sum_{t \le T} \mathbb{E}[\text{ordinary insertion cost at } t] \le O_\varepsilon(T \log(2 + T/m)).$$

Adding the $O_\varepsilon(T + m)$ rebuild-and-timeout cost gives

$$\mathbb{E}[\text{total work}] = O_\varepsilon((T + m) \log(2 + T/m)).$$

On the shorter horizon $T \le \exp(m/(36L_m^4))$, never-before-seen keys cost $O_\varepsilon(1)$ each, while each reinsertion costs at most $O_\varepsilon(\log(2 + T/m))$. Hence

$$\mathbb{E}[\text{total work}] = O_\varepsilon(T + m) + O_\varepsilon(R_T \log(2 + T/m)).$$

In particular, if no key is ever reinserted, the total expected work is $O_\varepsilon(T + m)$ throughout this horizon.

Limitations of the bound

The extra logarithm for reinserted keys comes from the factor $\mathbb{E}[D_t]$ in the selection lemma. Once a key has appeared before, its hash values may have influenced which pair is still in use, and the argument above does not show that this conditioning is harmless for that particular key.
The finite horizon in the constant-cost bound for new keys has the same source: the exponentially small concentration failure probability is multiplied by $\mathbb{E}[D_t]$. A forced rebuild at sufficiently long intervals would reset this factor at negligible amortized cost.
These are limitations of the proof above, not of the basic cuckoo-hashing placement theorem. The scheduled-rebuild variant avoids both conditioning issues because every epoch begins with a fresh pair independent of the keys that can appear in that epoch.