Advanced Algorithms – COMS31900

Hashing part three

Cuckoo Hashing

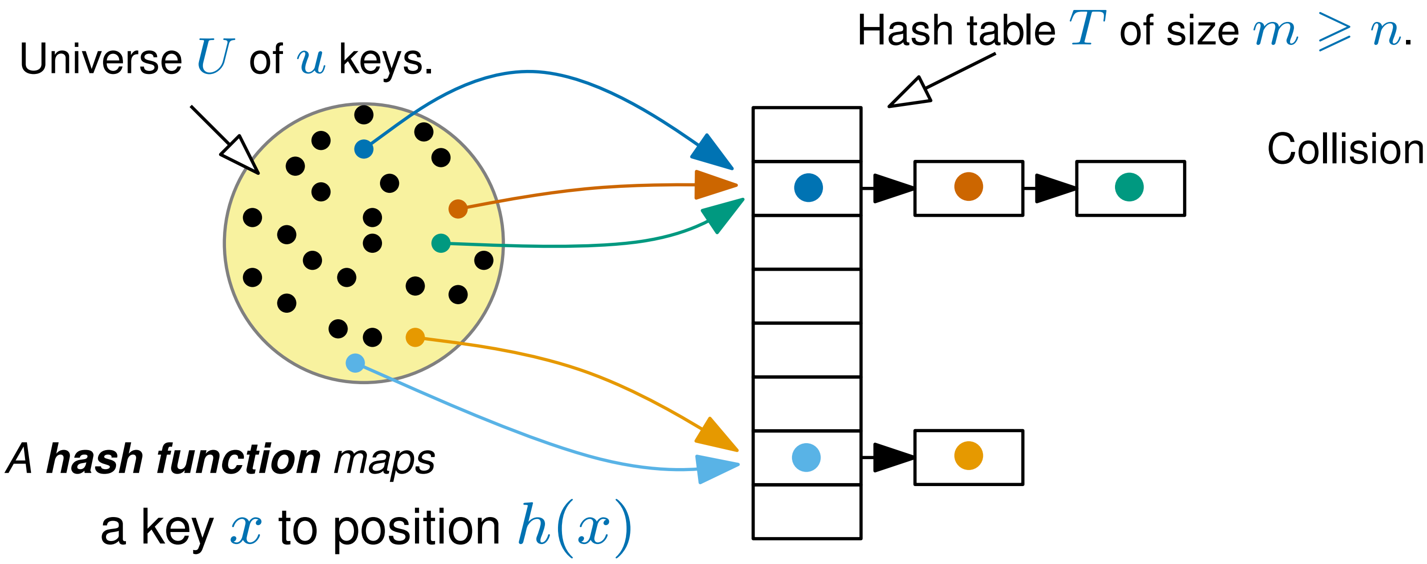
Raphaël Clifford

Slides by Benjamin Sach

Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)



Universe U of u keys.

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

Collisions are fixed by **chaining**

*A **hash function** maps*

a key x to position $h(x)$

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

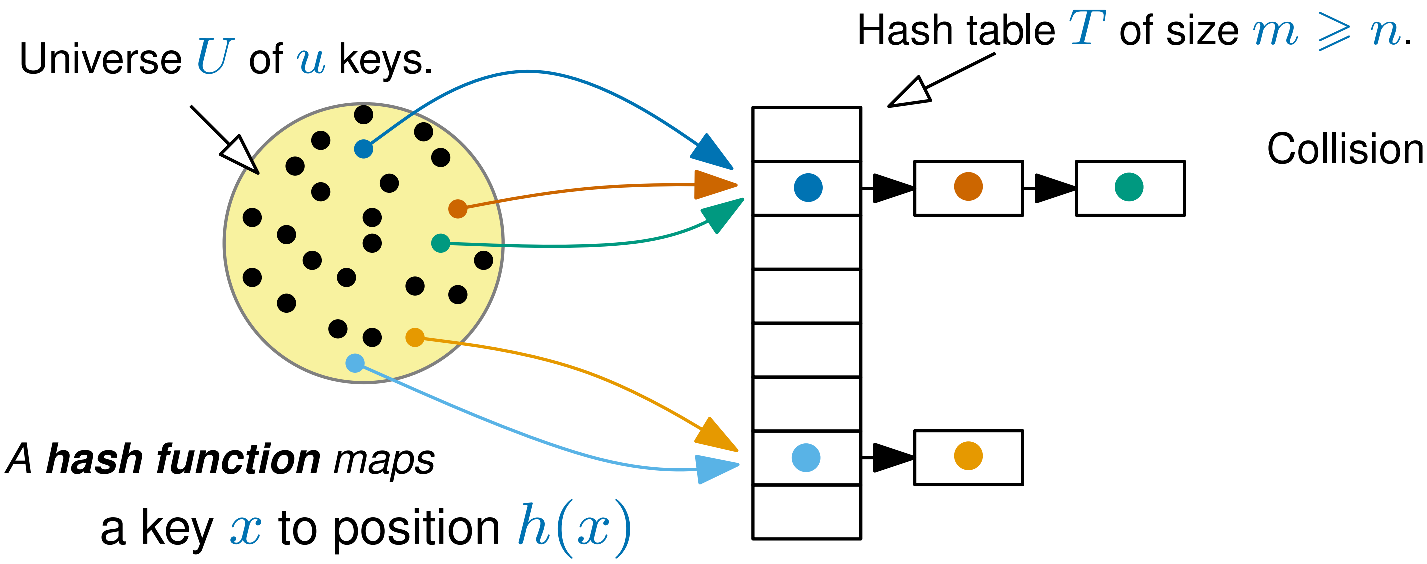
Using weakly universal hashing:

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

Back to the start (again)

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

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


Universe U of u keys.

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

Collisions are fixed by **chaining**

*A **hash function** maps*

a key x to position $h(x)$

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

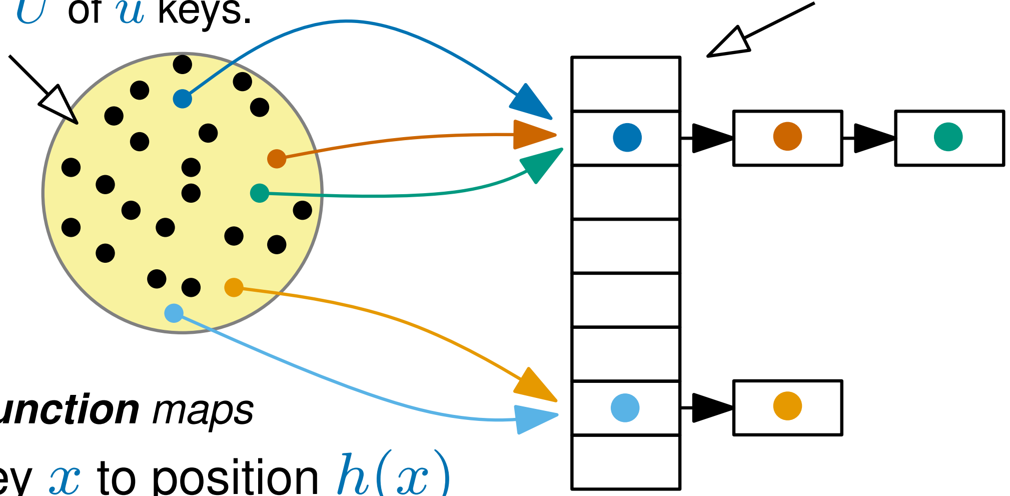
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)



Universe U of u keys.

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

Collisions are fixed by ~~**chaining**~~ **bucketing**

*A **hash function** maps*

a key x to position $h(x)$

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

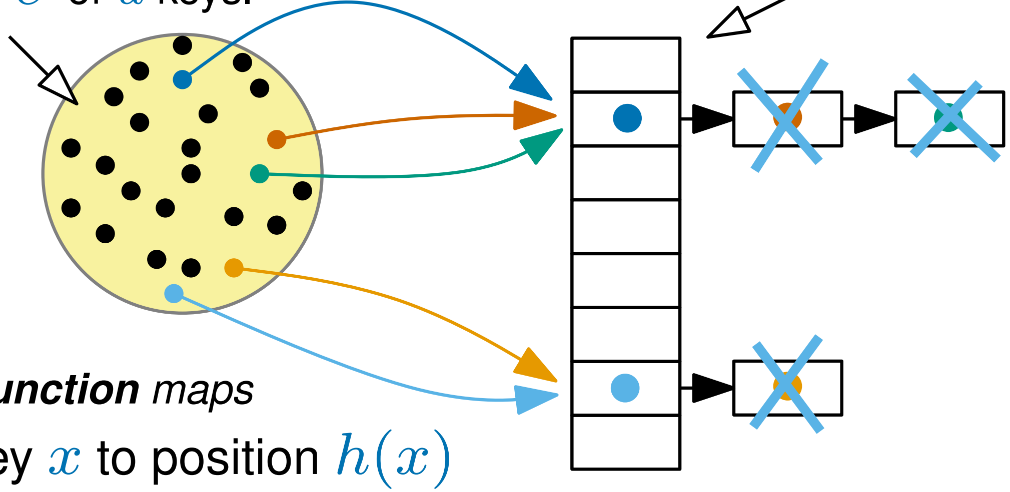
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)



Universe U of u keys.

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

*A **hash function** maps*

a key x to position $h(x)$

Collisions are fixed by ~~**chaining**~~

bucketing

n arbitrary operations arrive online, one at a time.

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

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

in fact this result can be generalised . . .

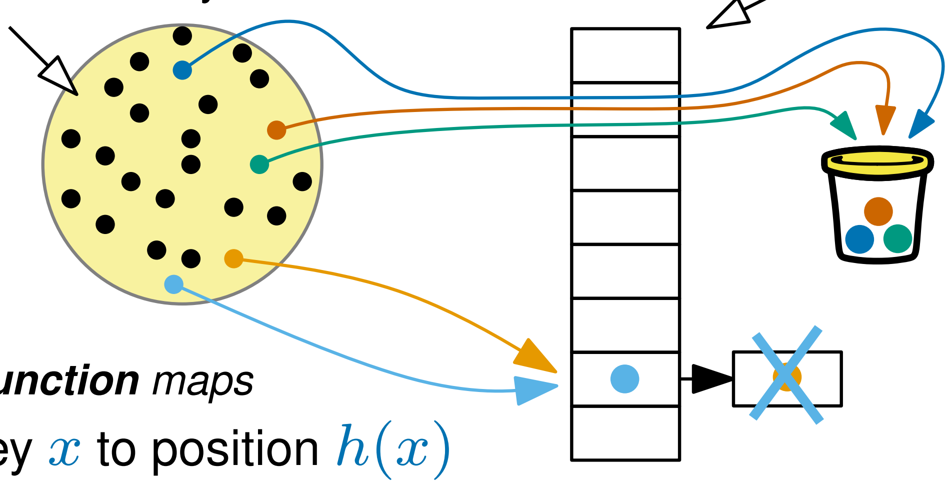
Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

Universe U of u keys.

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



Collisions are fixed by ~~**chaining**~~ **bucketing**

A ***hash function*** *maps*

a key x to position $h(x)$

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

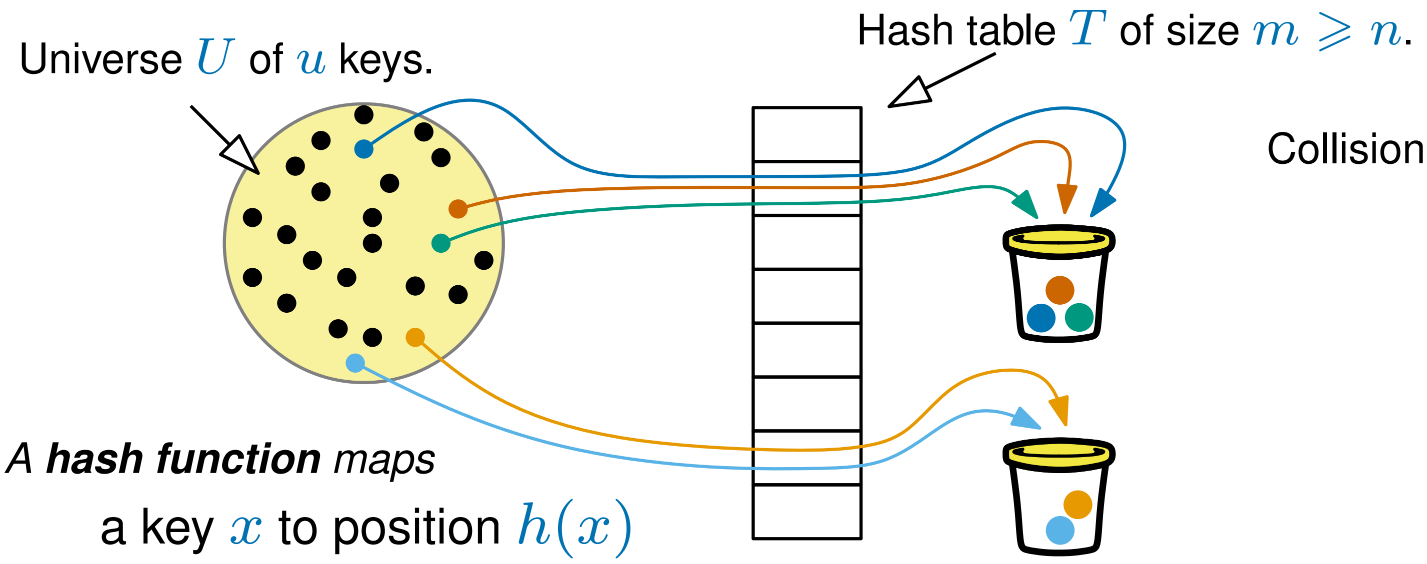
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)



*A **hash function** maps*

a key x to position $h(x)$

Collisions are fixed by ~~**chaining**~~ **bucketing**

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

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

in fact this result can be generalised . . .

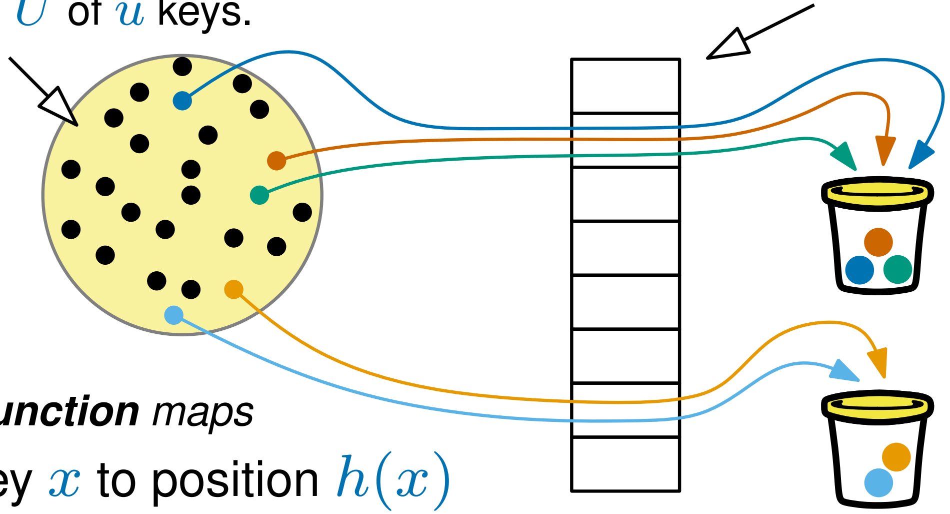
Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

Universe U of u keys.

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



Collisions are fixed by ~~chaining~~ **bucketing**

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the **bucket**

*A **hash function** maps*

a key x to position $h(x)$

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

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

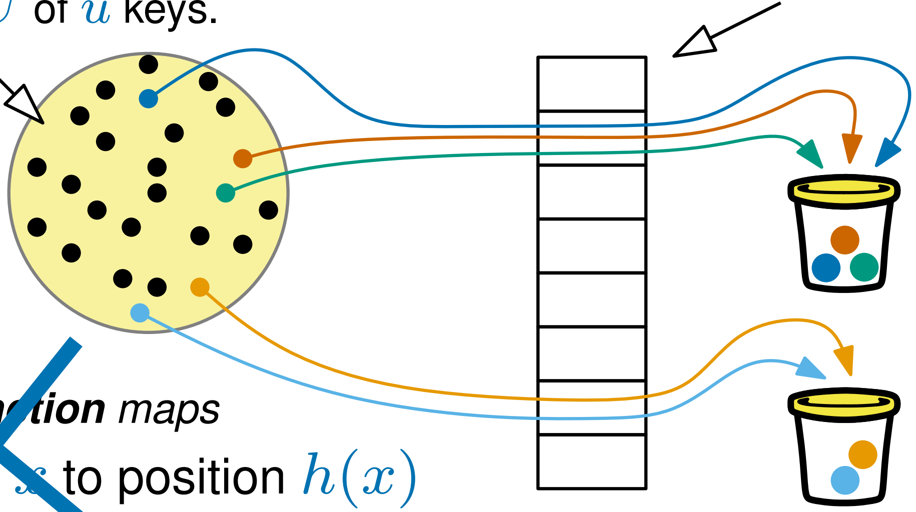
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)



Universe U of u keys.

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

Collisions are fixed by ~~**chaining**~~ **bucketing**

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the **bucket**

~~*A **hash function** maps*~~

~~a key x to position $h(x)$~~

n arbitrary operations arrive online, one at a time.

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

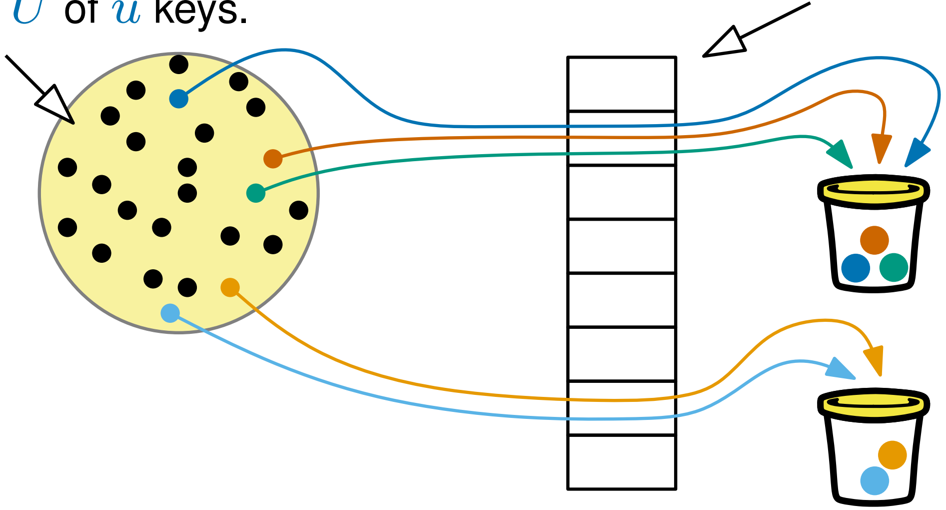
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$



Universe U of u keys.

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

Collisions are fixed by ~~**chaining**~~ **bucketing**

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

n arbitrary operations arrive online, one at a time.

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

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

in fact this result can be generalised . . .

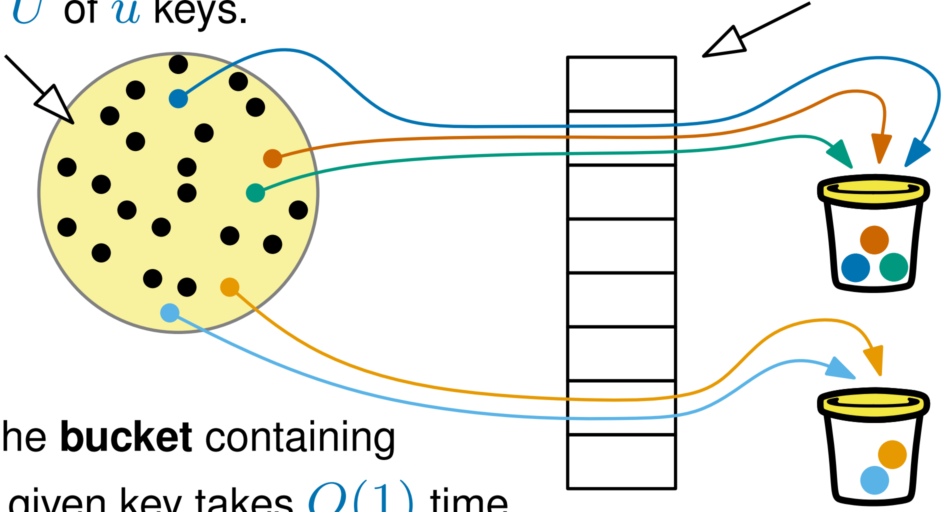
Back to the start (again)

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

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

Universe U of u keys.

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



Collisions are fixed by **~~chaining~~**

bucketing

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

Locating the **bucket** containing

a given key takes $O(1)$ time

n arbitrary operations arrive online, one at a time.

A set H of hash functions is **weakly universal** if for any

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

$$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$$

(h is picked uniformly at random from H)

Using weakly universal hashing:

For *any* n operations, the *expected*

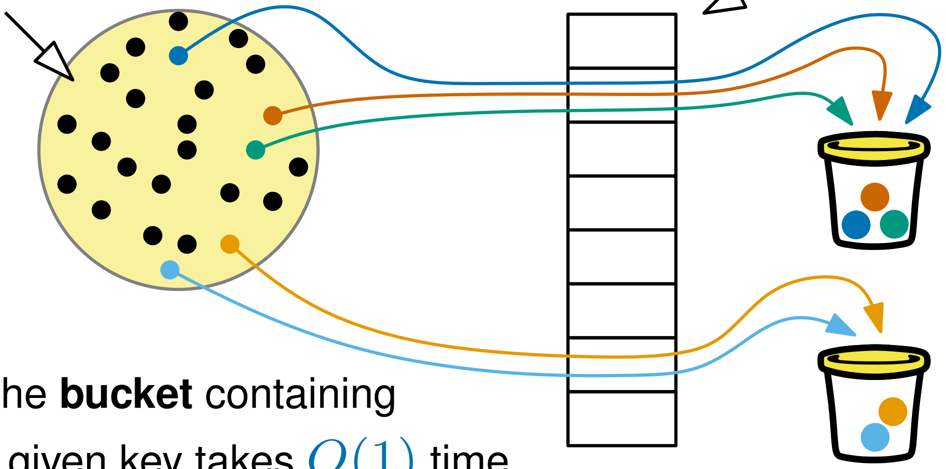
run-time is $O(1)$ per operation.

in fact this result can be generalised . . .

Back to the start (again)

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

  add$($*key*, *value*$)$, lookup$($*key*$)$ (which returns *value*) and delete$($*key*$)$



Universe U of u keys.

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

Collisions are fixed by ~~**chaining**~~ **bucketing**

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

where s is the number of keys in the ***bucket***

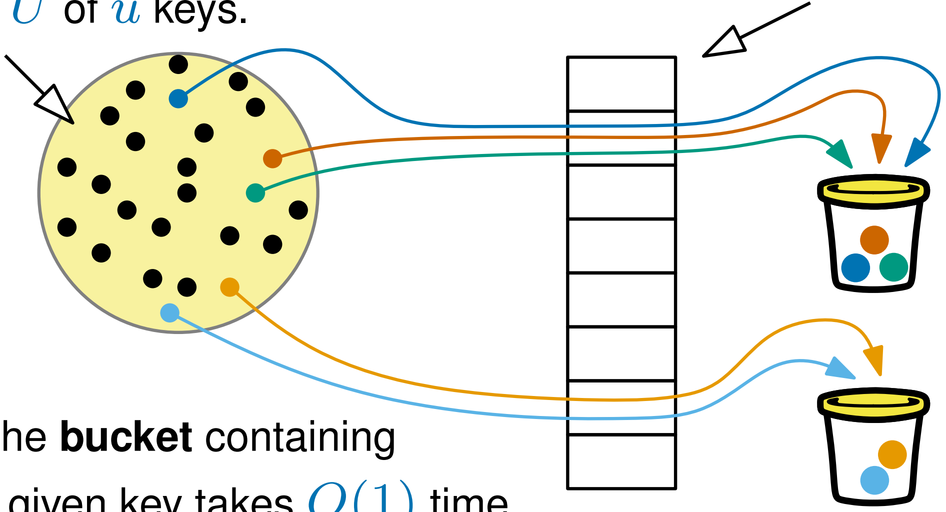
Locating the **bucket** containing a given key takes $O(1)$ time

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

Back to the start (again)

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$



Universe U of u keys.

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

Collisions are fixed by **~~chaining~~**

bucketing

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

Locating the **bucket** containing

a given key takes $O(1)$ time

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

If our construction has the property that,

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

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

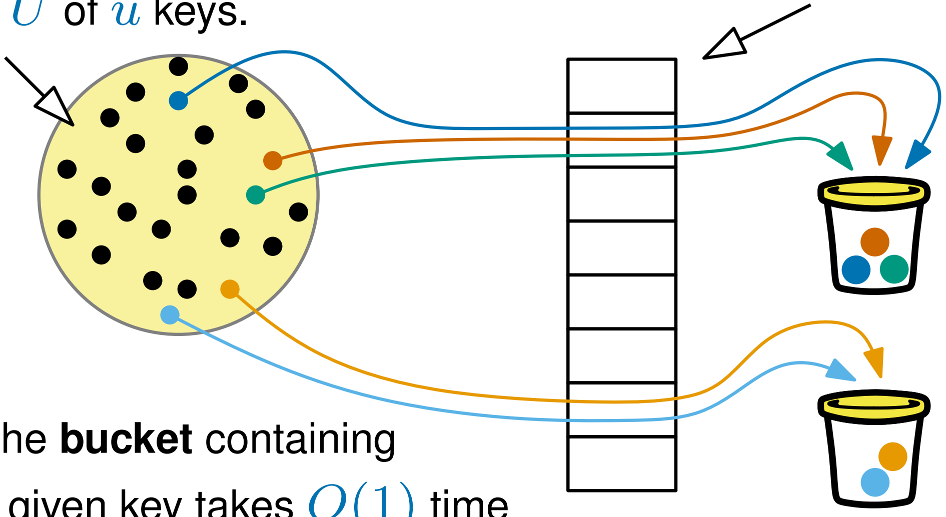
Back to the start (again)

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$

Universe U of u keys.

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



Collisions are fixed by ~~**chaining**~~ **bucketing**

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

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

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

If our construction has the property that,

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

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

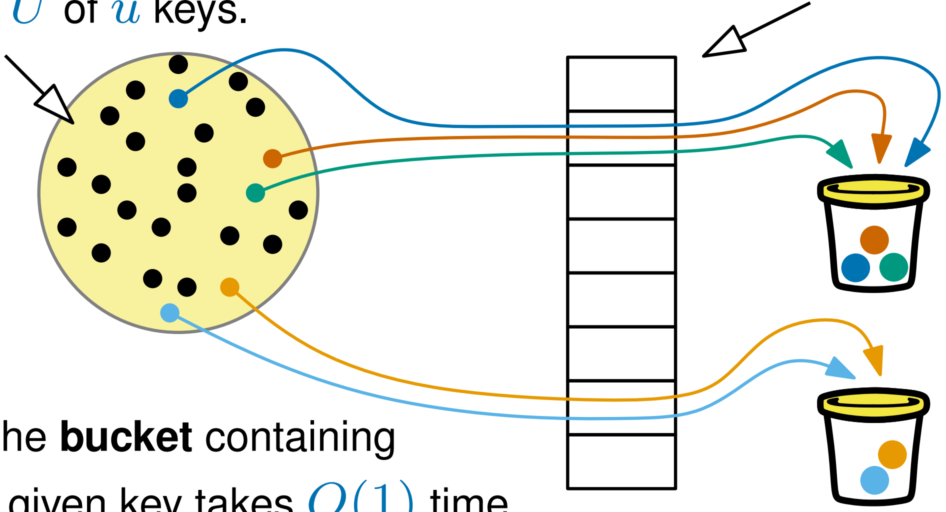
Back to the start (again)

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$

Universe U of u keys.

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



Collisions are fixed by ~~**chaining**~~ **bucketing**

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

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

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

If our construction has the property that,

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

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

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

Dynamic perfect hashing

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

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

THEOREM

In the **Cuckoo hashing** scheme:

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

Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

Dynamic perfect hashing

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

add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

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

means every operation takes constant time

Dynamic perfect hashing

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

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

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

"$O(1)$ worst-case time per operation"

means every operation takes constant time

"The total worst-case time complexity of performing any n operations is $O(n)$"

Dynamic perfect hashing

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

"$O(1)$ worst-case time per operation"

means every operation takes constant time

"The total worst-case time complexity of performing any n operations is $O(n)$"

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

Dynamic perfect hashing

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

  add$(\mathit{key}, \mathit{value})$, lookup$(\mathit{key})$ (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

"$O(1)$ worst-case time per operation"

means every operation takes constant time

"The total worst-case time complexity of performing any n operations is $O(n)$"

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

However, it **does mean** that the *amortised worst-case* time complexity of an operation is $O(1)$

Dynamic perfect hashing

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

  add$(\textit{key}, \textit{value})$, lookup$(\textit{key})$ (which returns *value*) and delete$(\textit{key})$

THEOREM

In the **Cuckoo hashing** scheme:

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

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

let's build it up. . .

"$O(1)$ **expected** time per operation"

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

"The total **expected** time complexity of performing any n operations is $O(n)$"

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

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

Dynamic perfect hashing

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

  add$(\mathit{key}, \mathit{value})$, lookup$(\mathit{key})$ (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

THEOREM

In the **Cuckoo hashing** scheme:

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

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

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

Dynamic perfect hashing

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

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

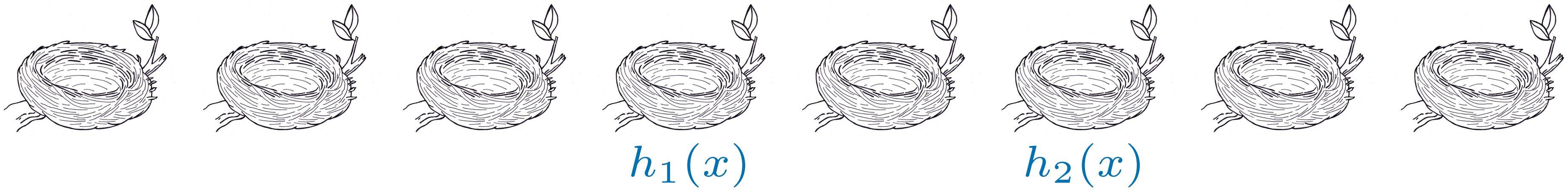
THEOREM

In the **Cuckoo hashing** scheme:

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

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

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



Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

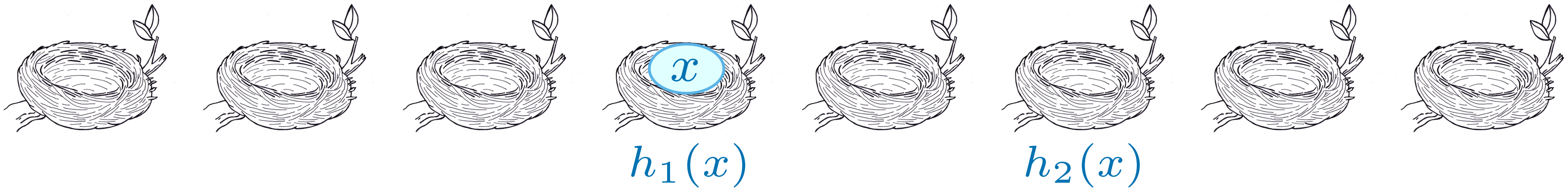
THEOREM

In the **Cuckoo hashing** scheme:

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

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

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



Dynamic perfect hashing

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

  add$(\mathit{key}, \mathit{value})$, lookup$(\mathit{key})$ (which returns *value*) and delete(key)

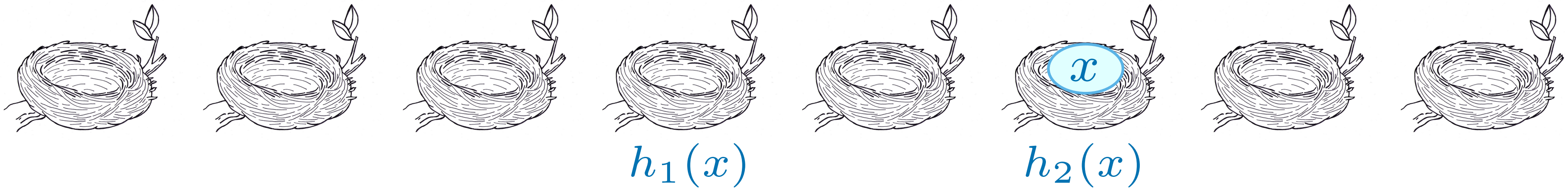
THEOREM

In the **Cuckoo hashing** scheme:

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

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

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



Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

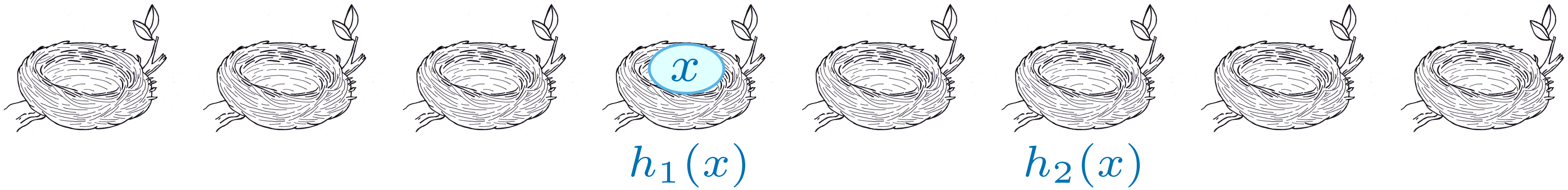
THEOREM

In the **Cuckoo hashing** scheme:

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

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

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



Dynamic perfect hashing

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

  add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

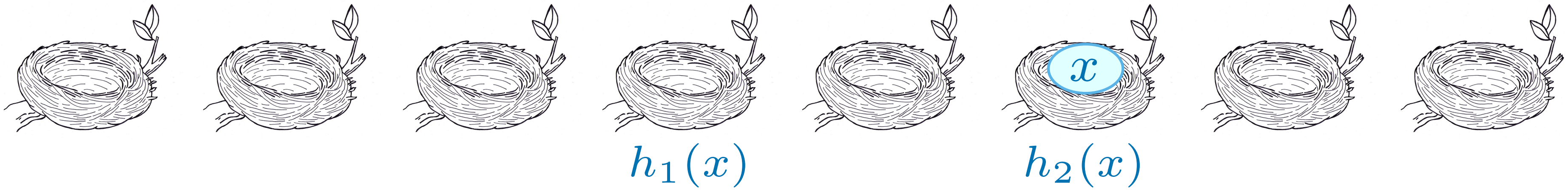
THEOREM

In the **Cuckoo hashing** scheme:

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

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

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



Dynamic perfect hashing

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

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

THEOREM

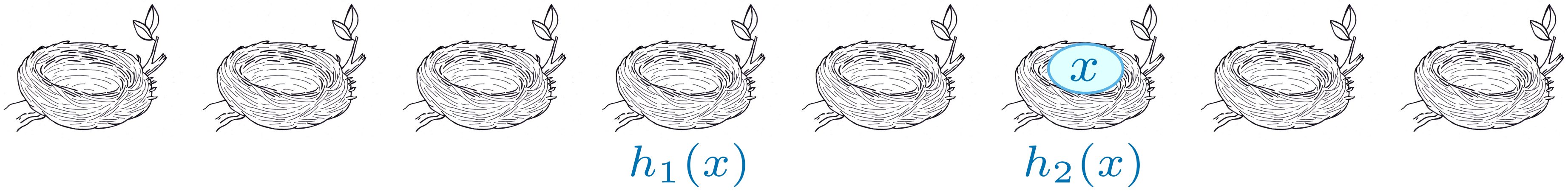
In the **Cuckoo hashing** scheme:

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

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

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

Important: We never store multiple keys at the same position



Dynamic perfect hashing

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

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

THEOREM

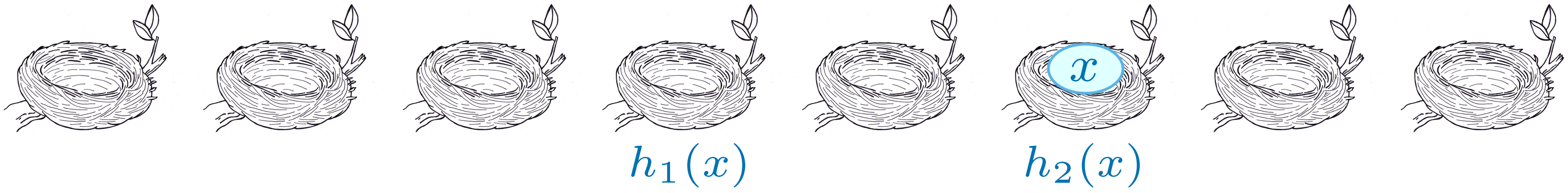
In the **Cuckoo hashing** scheme:

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

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

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

Important: We never store multiple keys at the same position



Therefore, as claimed, lookup *takes* $O(1)$ *time. . .*

Dynamic perfect hashing

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

add$(key, value)$, lookup(key) (which returns *value*) and delete(key)

THEOREM

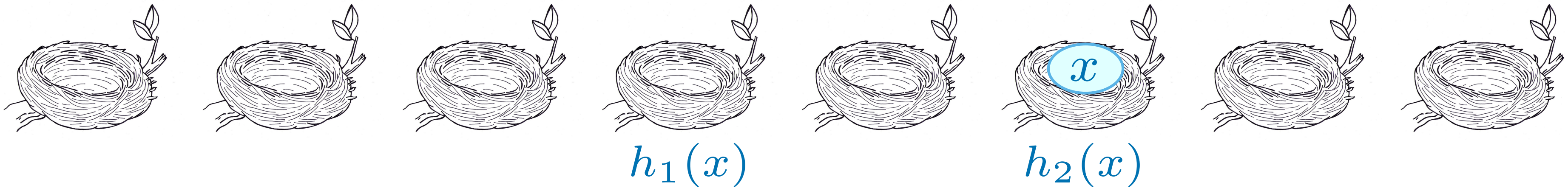
In the **Cuckoo hashing** scheme:

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

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

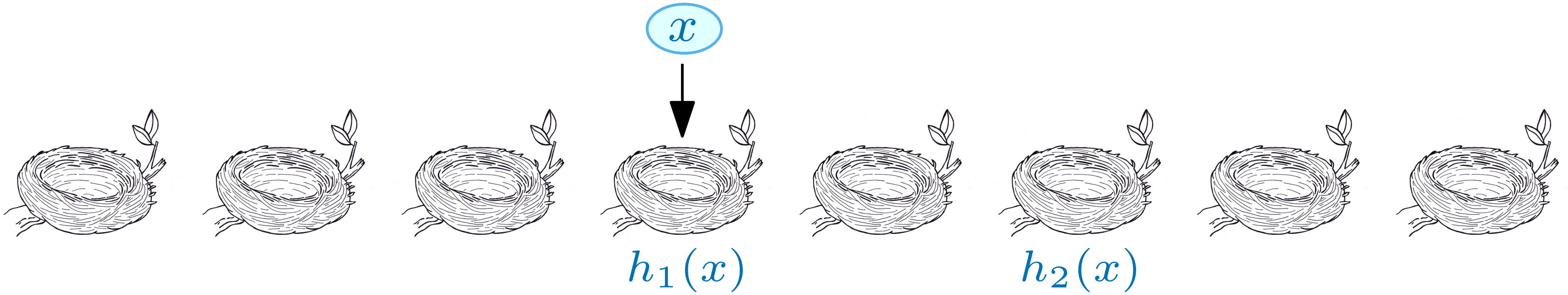
Each key in the table is either stored at position $h_1(x)$ or $h_2(x)$.

Important: We never store multiple keys at the same position



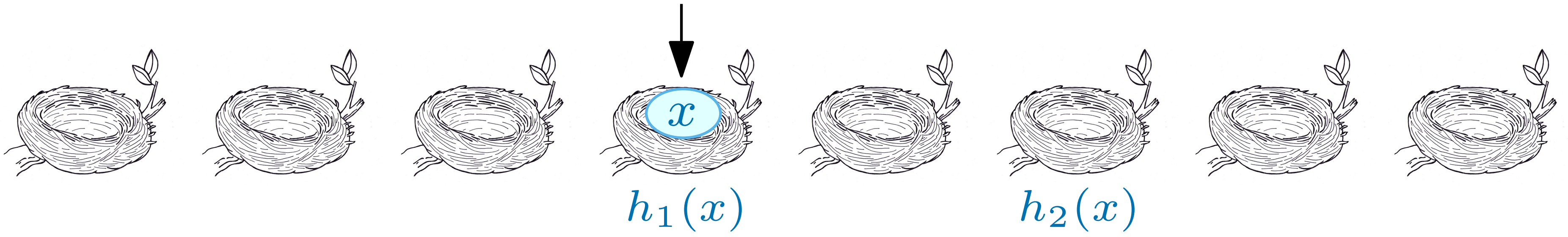
Therefore, as claimed, lookup *takes* $O(1)$ *time… but how do we do inserts?*

Inserts in Cuckoo hashing



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

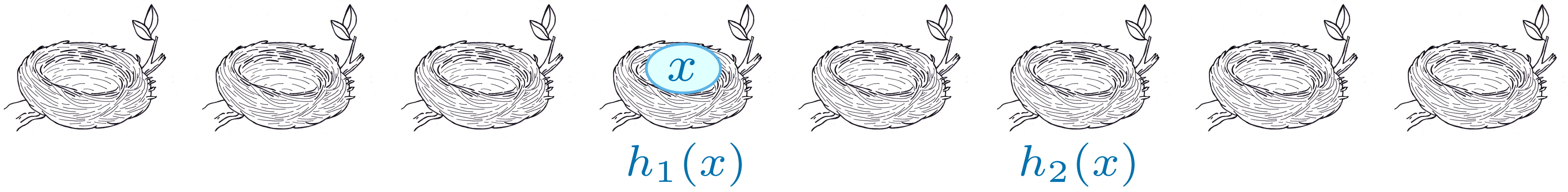
Inserts in Cuckoo hashing



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

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

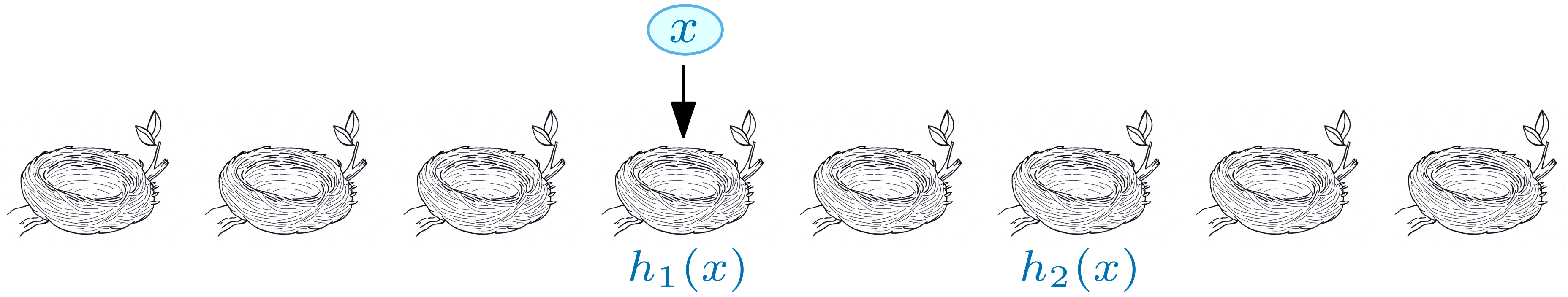
Inserts in Cuckoo hashing



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

if that position is empty, stop

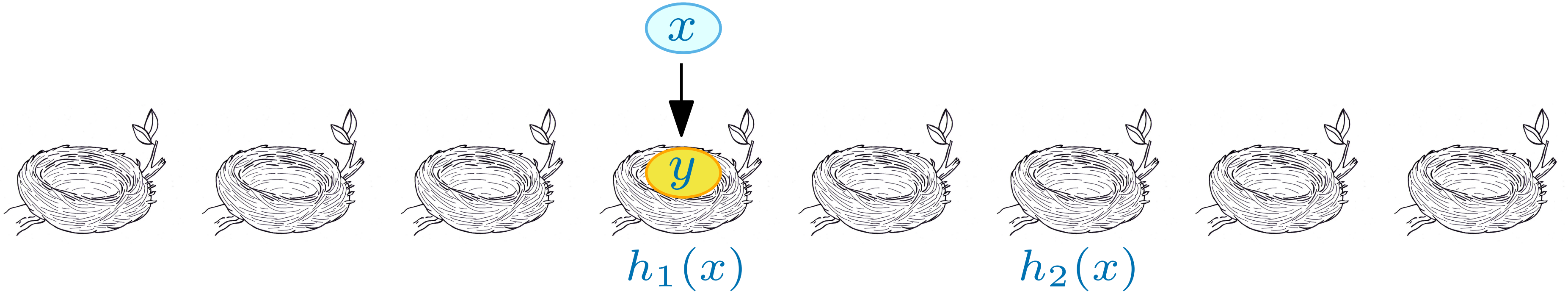
Inserts in Cuckoo hashing



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

if that position is empty, stop

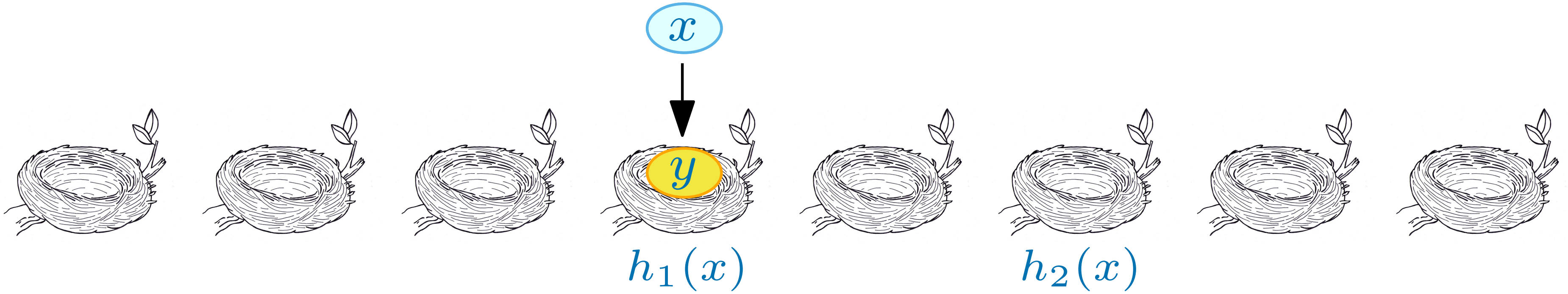
Inserts in Cuckoo hashing



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

if that position is empty, stop

Inserts in Cuckoo hashing

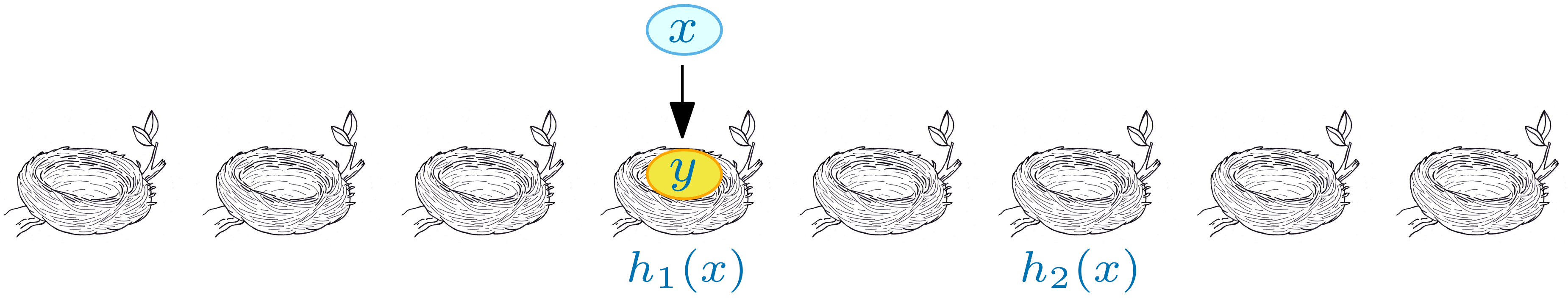


Step 1: Attempt to put x in position $h_1(x)$

if that position is empty, stop

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

Inserts in Cuckoo hashing



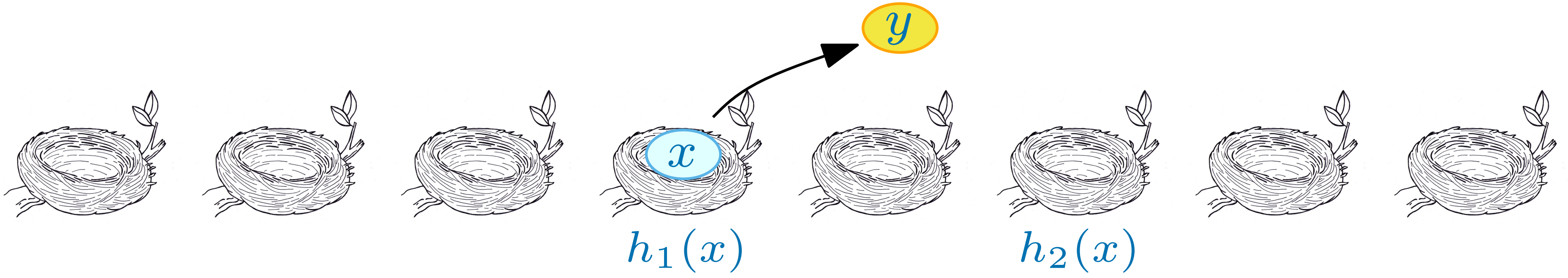
Step 1: Attempt to put x in position $h_1(x)$

if that position is empty, stop

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

evict key y and replace it with key x

Inserts in Cuckoo hashing



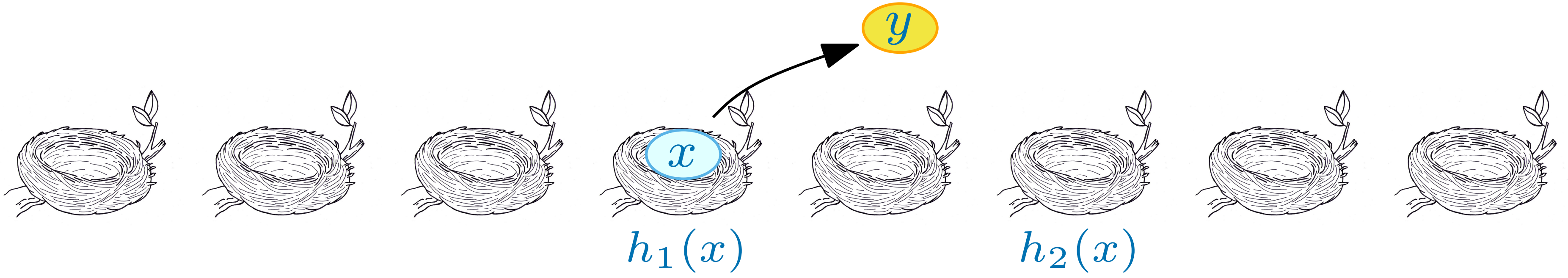
Step 1: Attempt to put x in position $h_1(x)$

if that position is empty, stop

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

evict key y and replace it with key x

Inserts in Cuckoo hashing



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

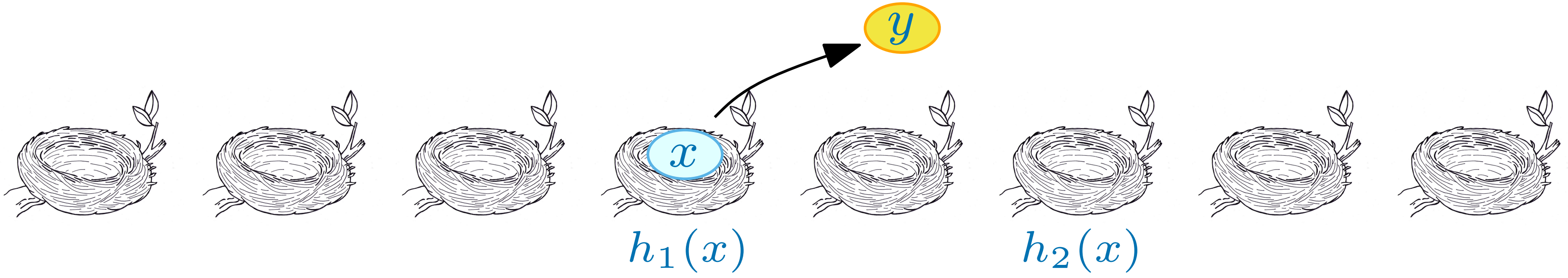
if that position is empty, stop

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

evict key y and replace it with key x

where should we put key y*?*

Inserts in Cuckoo hashing



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

if that position is empty, stop

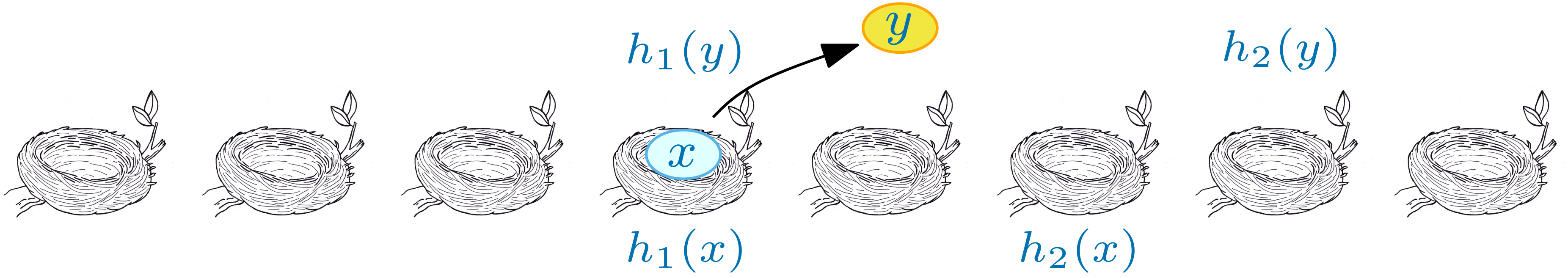
Step 2: Let y be the key currently in position $h_1(x)$

evict key y and replace it with key x

where should we put key y?

in the *other* position it's allowed in

Inserts in Cuckoo hashing



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

if that position is empty, stop

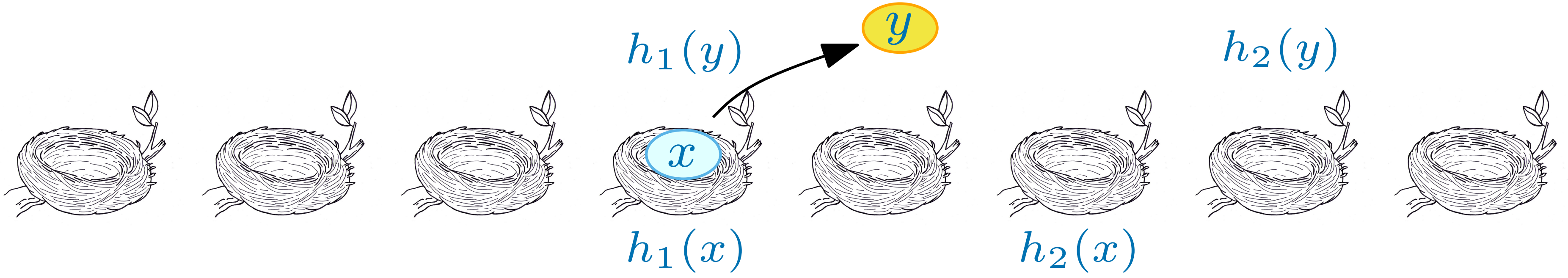
Step 2: Let y be the key currently in position $h_1(x)$

evict key y and replace it with key x

where should we put key y*?*

in the *other* position it's allowed in

Inserts in Cuckoo hashing



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

if that position is empty, stop

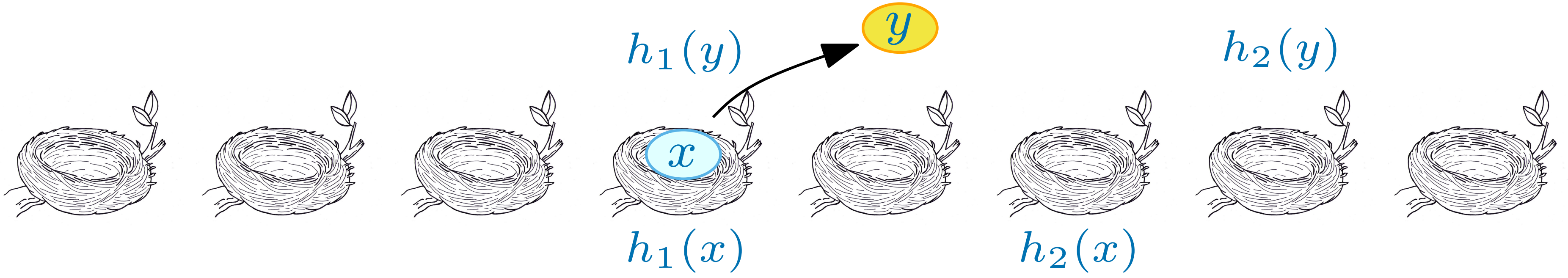
Step 2: Let y be the key currently in position $h_1(x)$

evict key y and replace it with key x

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

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

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

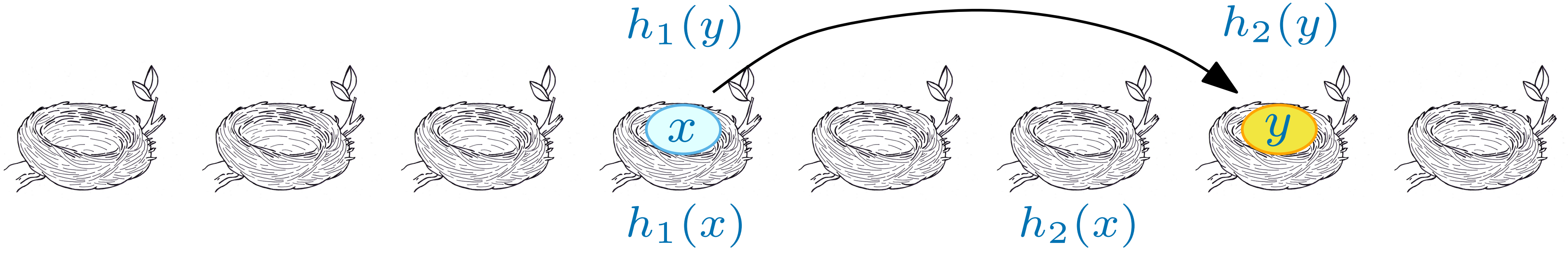
Step 3: Let pos be the *other* position y is allowed to be in

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

Step 4: Attempt to put y in position pos

if that position is empty, stop

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

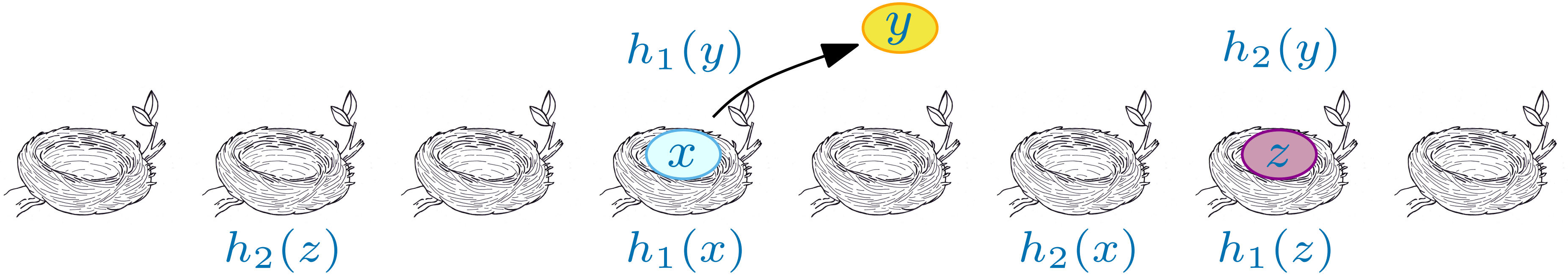
Step 3: Let pos be the *other* position y is allowed to be in

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

Step 4: Attempt to put y in position pos

if that position is empty, stop

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

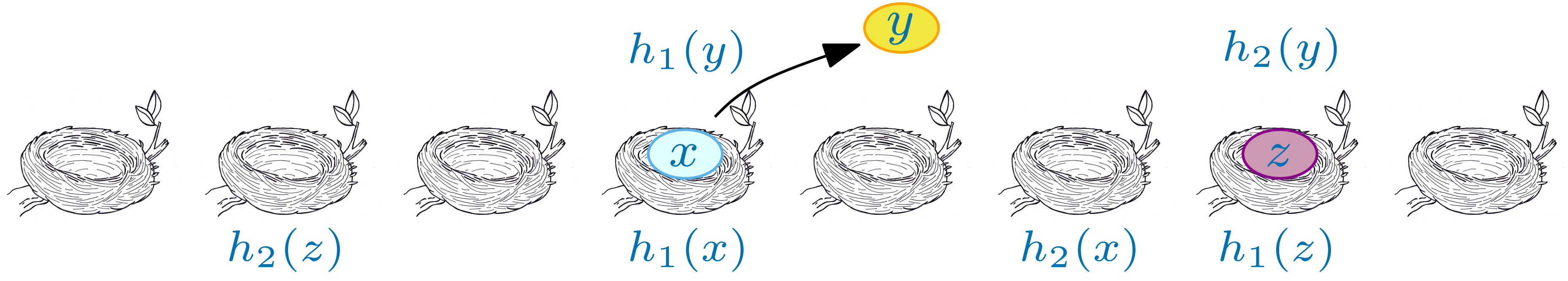
Step 3: Let pos be the *other* position y is allowed to be in

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

Step 4: Attempt to put y in position pos

if that position is empty, stop

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

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

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

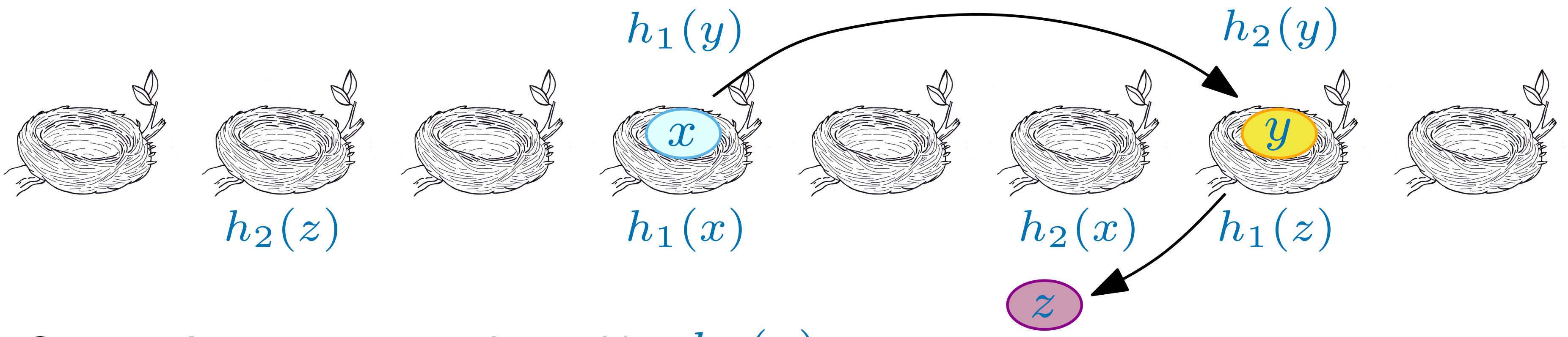
Step 4: Attempt to put y in position pos

if that position is empty, stop

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

evict key z and replace it with key y

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

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

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

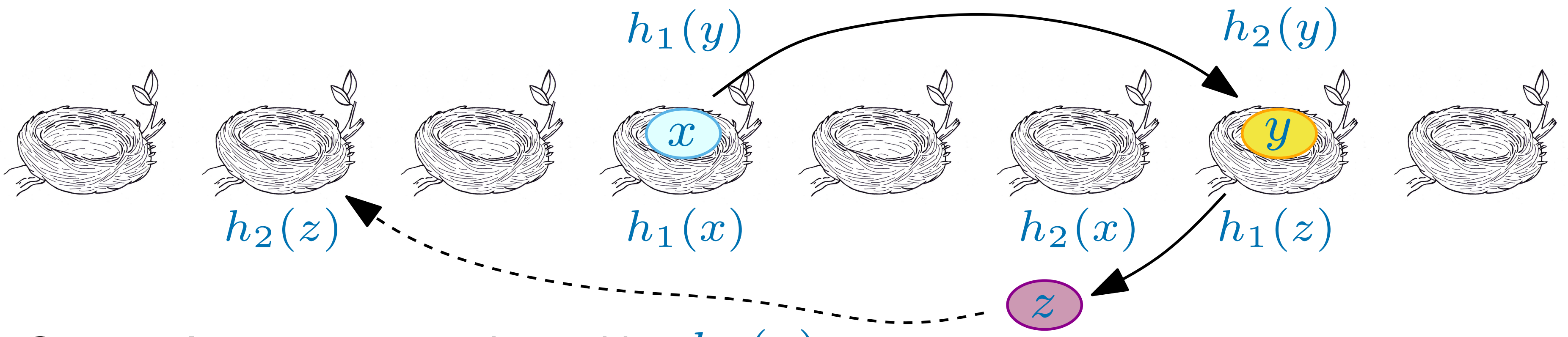
Step 4: Attempt to put y in position pos

if that position is empty, stop

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

evict key z and replace it with key y

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

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

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

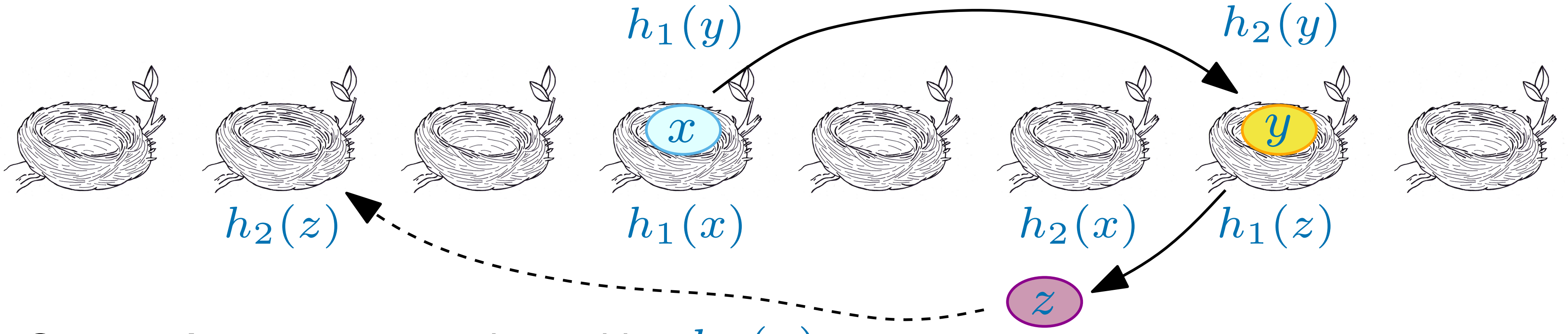
Step 4: Attempt to put y in position pos

if that position is empty, stop

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

evict key z and replace it with key y

Inserts in Cuckoo hashing



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

if that position is empty, stop

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

evict key y and replace it with key x

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

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

Step 4: Attempt to put y in position pos

if that position is empty, stop

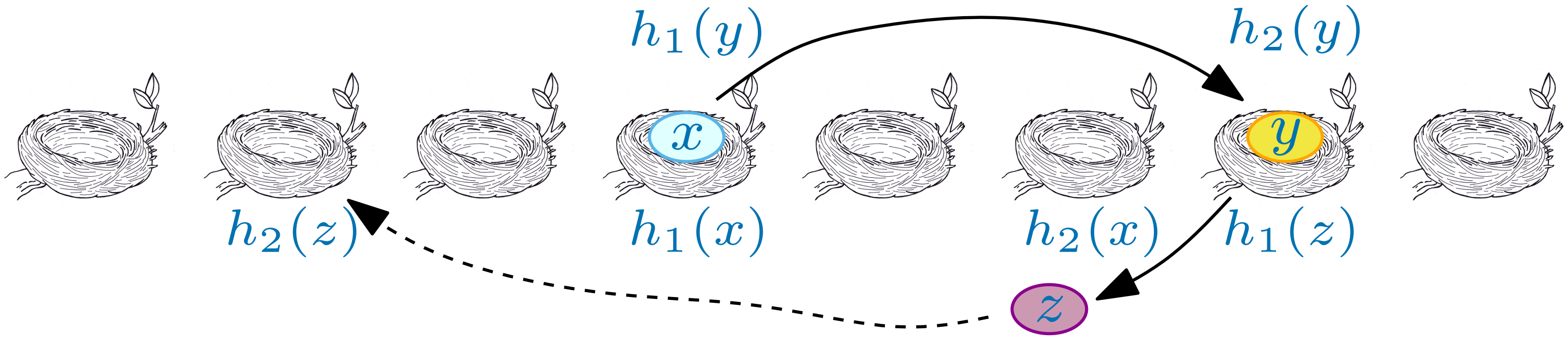
Step 5: Let z be the key currently in position pos

evict key z and replace it with key y

and so on...

Pseudocode

```
add($x$):
  pos ← $h_1(x)$
  Repeat at most $n$ times:
    If $T[\text{pos}]$ is empty then $T[\text{pos}] \leftarrow x$.
    Otherwise,
      $y \leftarrow T[\text{pos}]$,
      $T[\text{pos}] \leftarrow x$,
      pos ← the other possible location for $y$.
        (i.e. if $y$ was evicted from $h_1(y)$ then pos ← $h_2(y)$, otherwise pos ← $h_1(y)$.)
    $x \leftarrow y$.
  Repeat
  Give up and rehash the whole table.
      i.e. empty the table, pick two new hash functions and reinsert every key
```



Rehashing

If we fail to insert a new key x,

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

then we declare the table “rubbish” and rehash.

Rehashing

If we fail to insert a new key x,

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

then we declare the table “rubbish” and rehash.

What does rehashing involve?

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

Rehashing

If we fail to insert a new key x,

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

then we declare the table “rubbish” and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

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

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

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

Build a *new* empty hash table of the same size

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

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

Build a *new* empty hash table of the same size

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

one by one, using the normal add operation.

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

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

Build a *new* empty hash table of the same size

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

one by one, using the normal add operation.

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

Rehashing

If we fail to insert a new key x,

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

then we declare the table "rubbish" and rehash.

What does rehashing involve?

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

at the time of we fail to insert key x.

To rehash we:

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

Build a *new* empty hash table of the same size

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

one by one, using the normal add operation.

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

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

Assumptions

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

Assumptions

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

We make the following assumptions:

Assumptions

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

We make the following assumptions:

h_1 and h_2 are independent

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

Assumptions

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

We make the following assumptions:

h_1 and h_2 are independent

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

h_1 and h_2 are truly random

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

Assumptions

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

We make the following assumptions:

h_1 and h_2 are independent

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

h_1 and h_2 are truly random

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

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

Assumptions

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

We make the following assumptions:

h_1 and h_2 are independent

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

h_1 and h_2 are truly random

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

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

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

Assumptions

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

We make the following assumptions:

REASONABLE ASSUMPTION

h_1 and h_2 are independent

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

h_1 and h_2 are truly random

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

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

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

Assumptions

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

We make the following assumptions:

REASONABLE ASSUMPTION

h_1 and h_2 are independent

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

UNREASONABLE ASSUMPTION

h_1 and h_2 are truly random

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

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

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

Assumptions

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

We make the following assumptions:

REASONABLE ASSUMPTION

h_1 and h_2 are independent

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

UNREASONABLE ASSUMPTION

h_1 and h_2 are truly random

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

QUESTIONABLE ASSUMPTION

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

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

Assumptions

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

We make the following assumptions:

REASONABLE ASSUMPTION

h_1 and h_2 are independent

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

UNREASONABLE ASSUMPTION

h_1 and h_2 are truly random

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

QUESTIONABLE ASSUMPTION

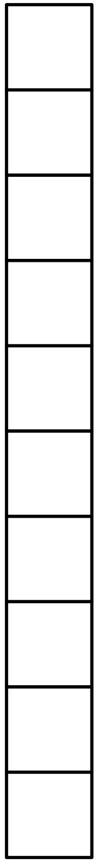
Computing the value of $h_1(x)$ and $h_2(x)$ takes $O(1)$ worst-case time

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

NOT ACTUALLY AN ASSUMPTION

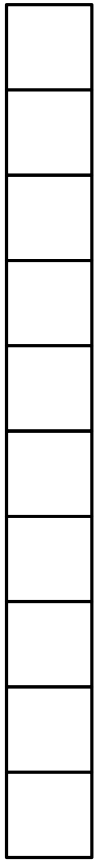
Cuckoo graph

Hash table
(size m)



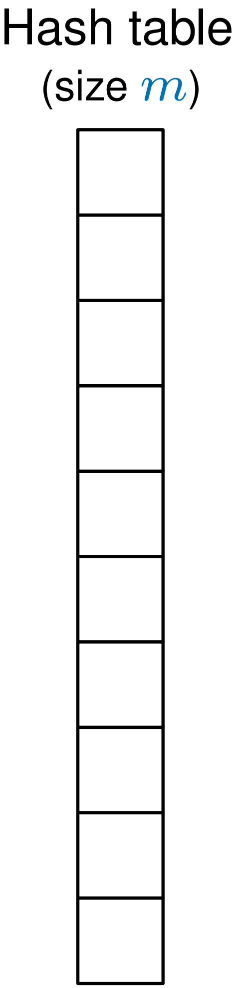
Cuckoo graph

Hash table
(size m)



The **cuckoo graph**:

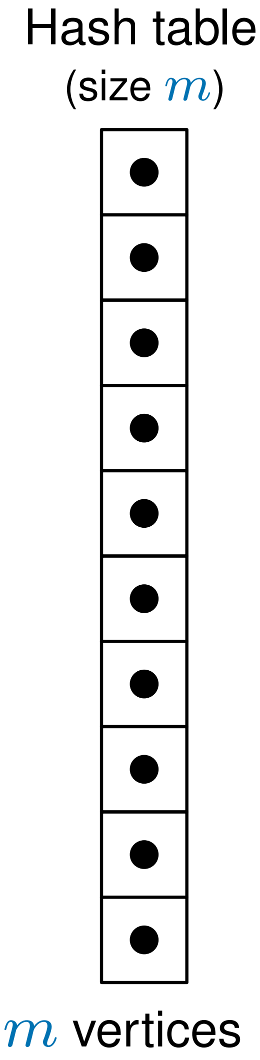
Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

Cuckoo graph



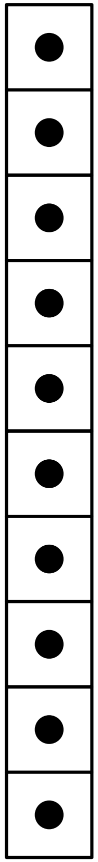
The **cuckoo graph**:

A vertex for each position of the table.

Cuckoo graph

Hash table

(size m)



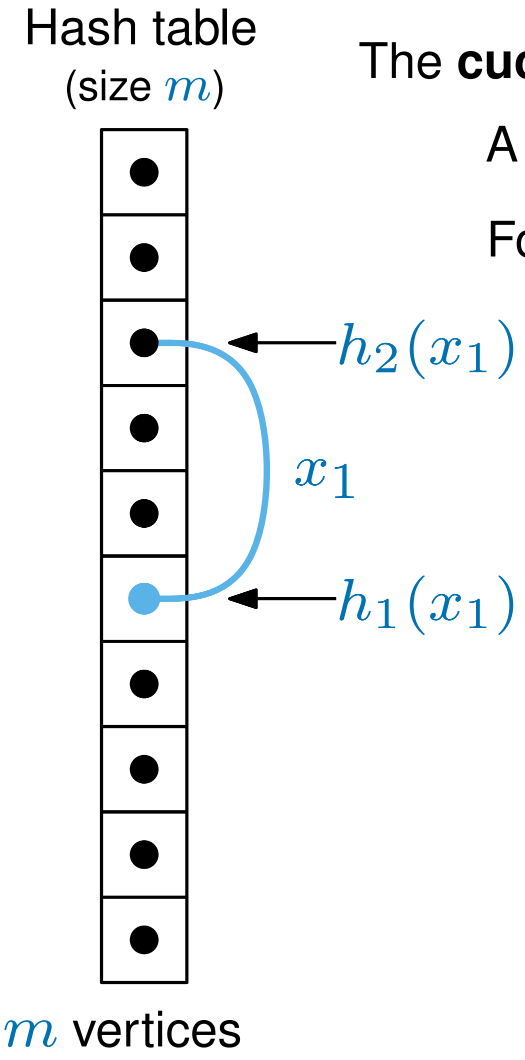
m vertices

The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

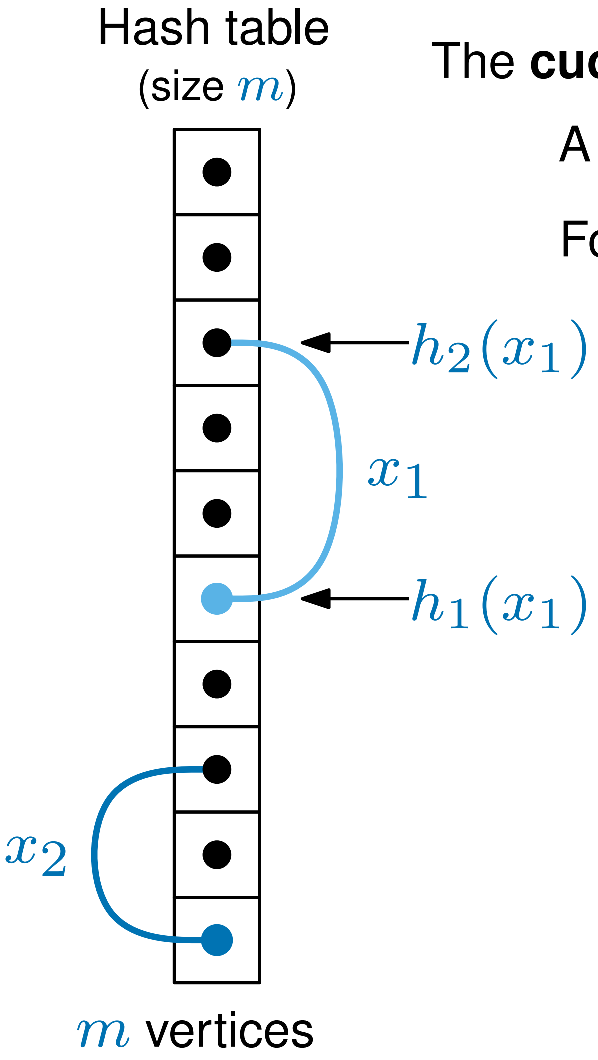


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

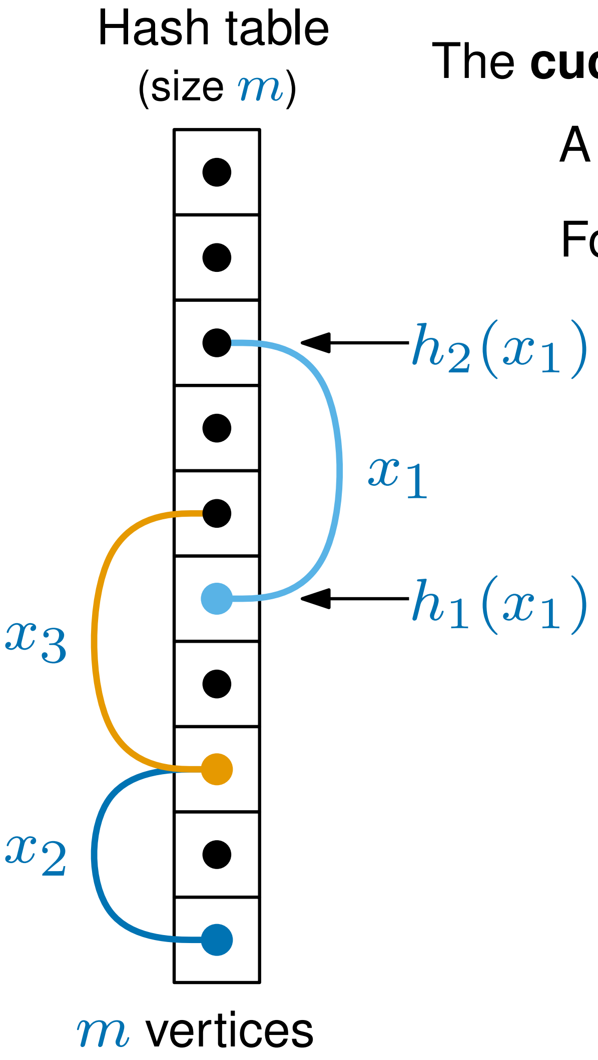


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

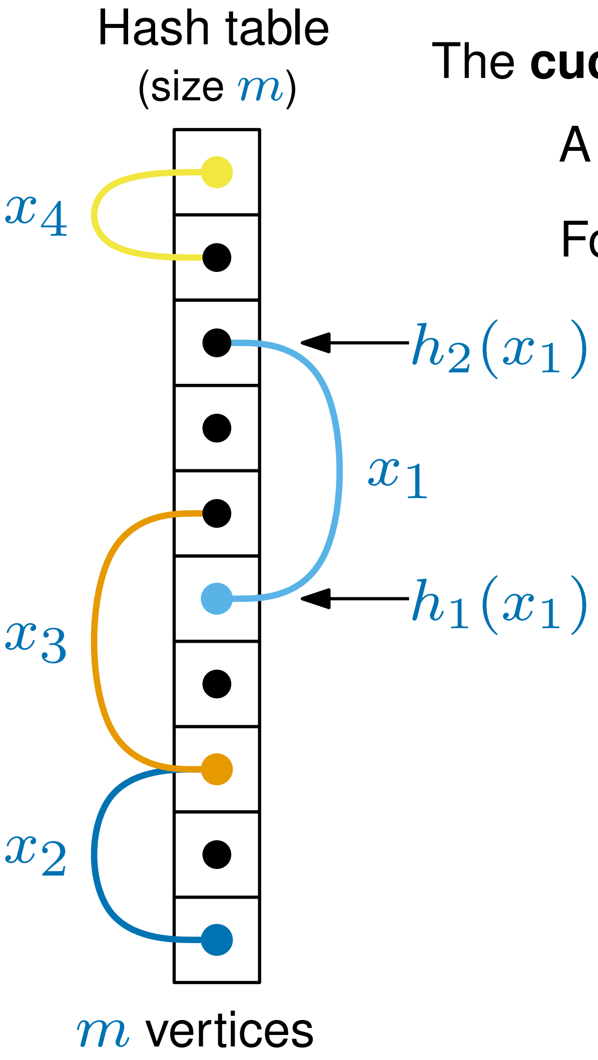


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

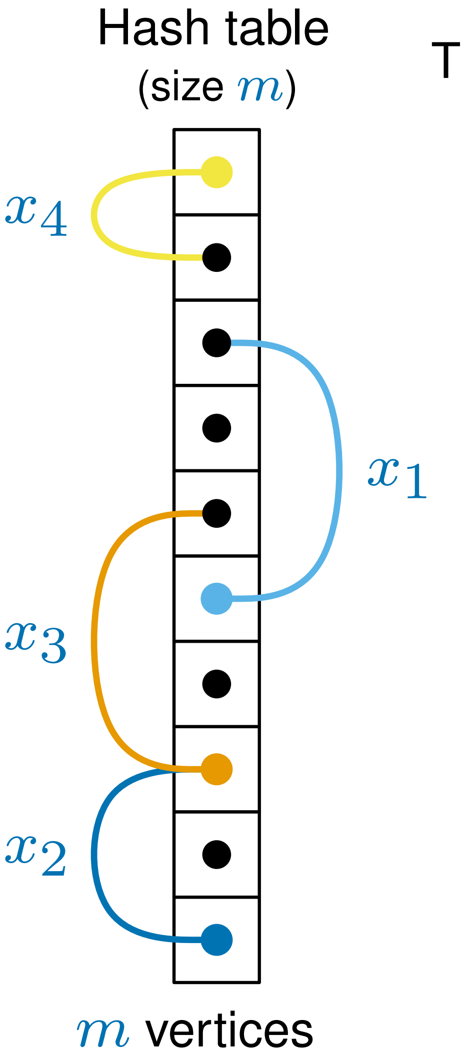


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

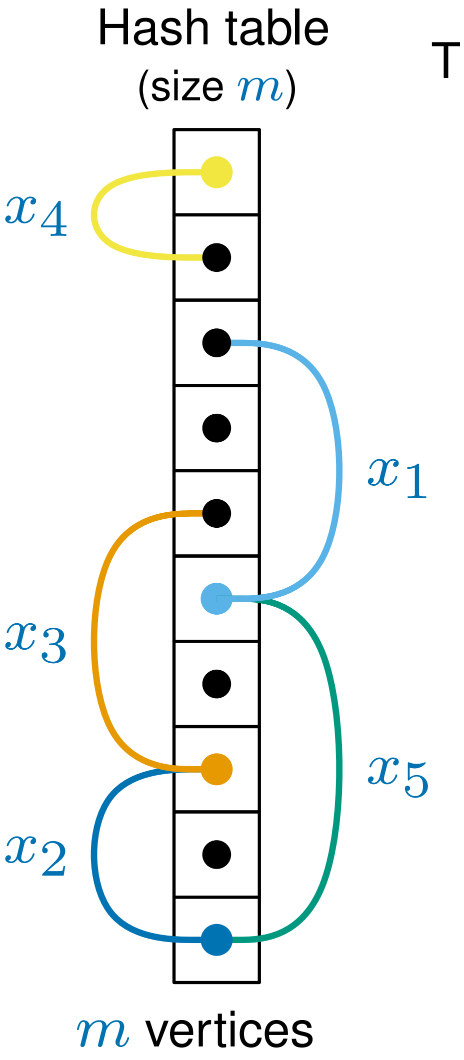


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

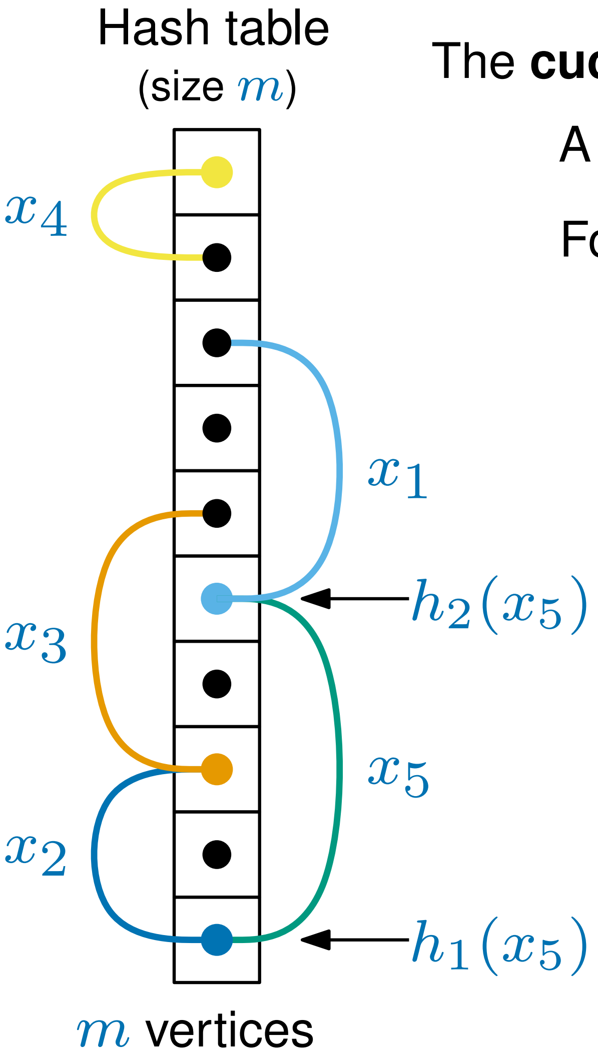


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph

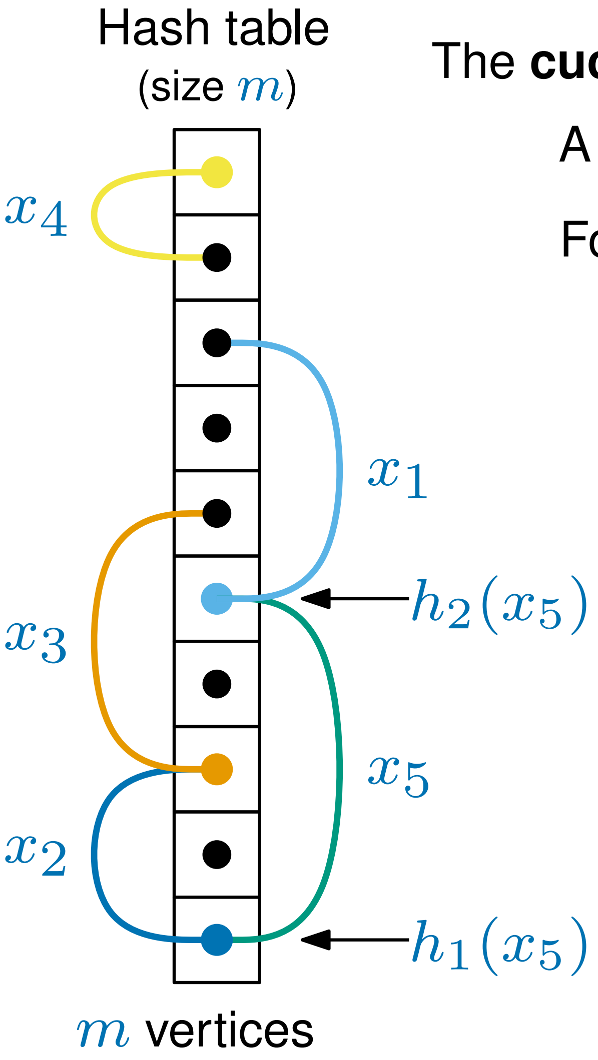


The **cuckoo graph**:

A vertex for each position of the table.

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

Cuckoo graph



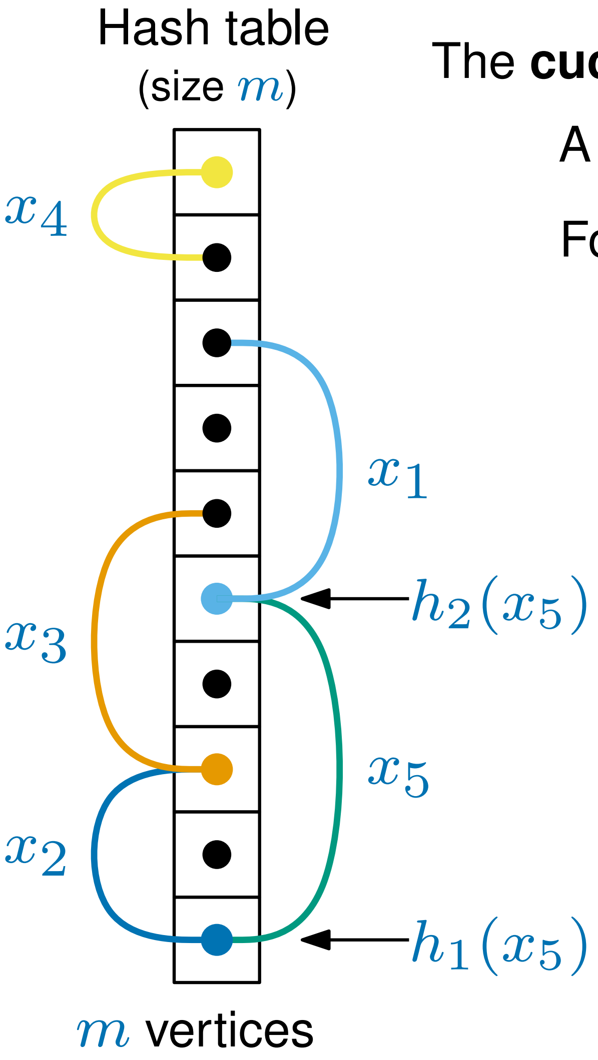
The **cuckoo graph**:

A vertex for each position of the table.

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

There is no space for $x_5\ldots$

Cuckoo graph



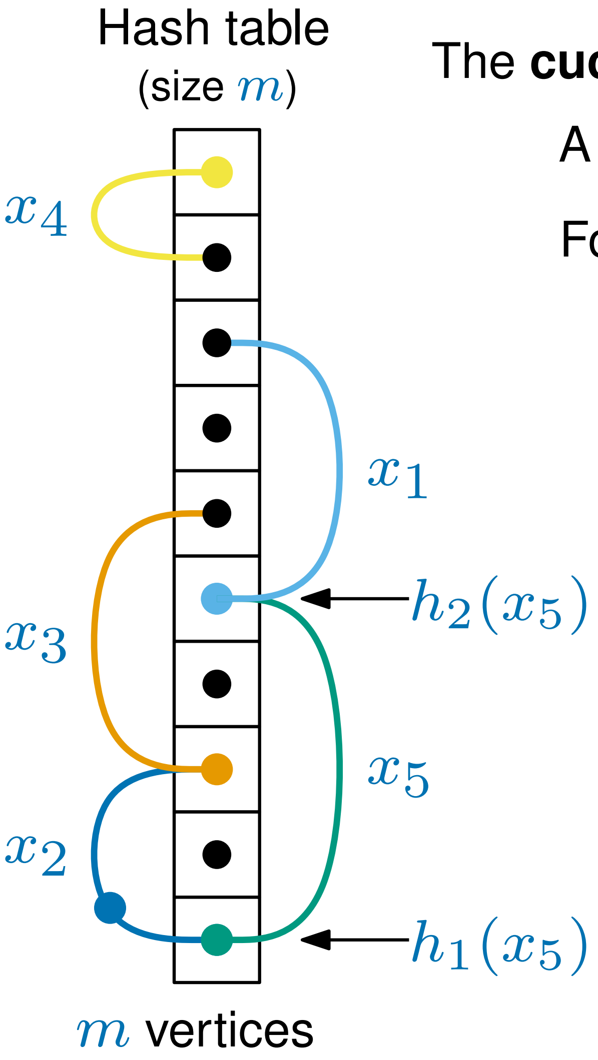
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



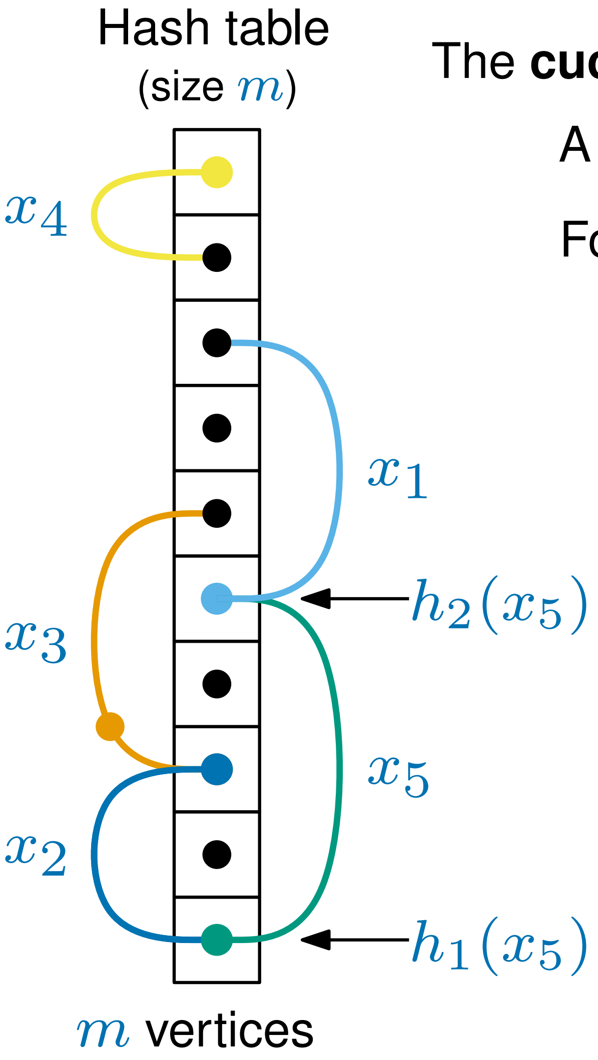
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



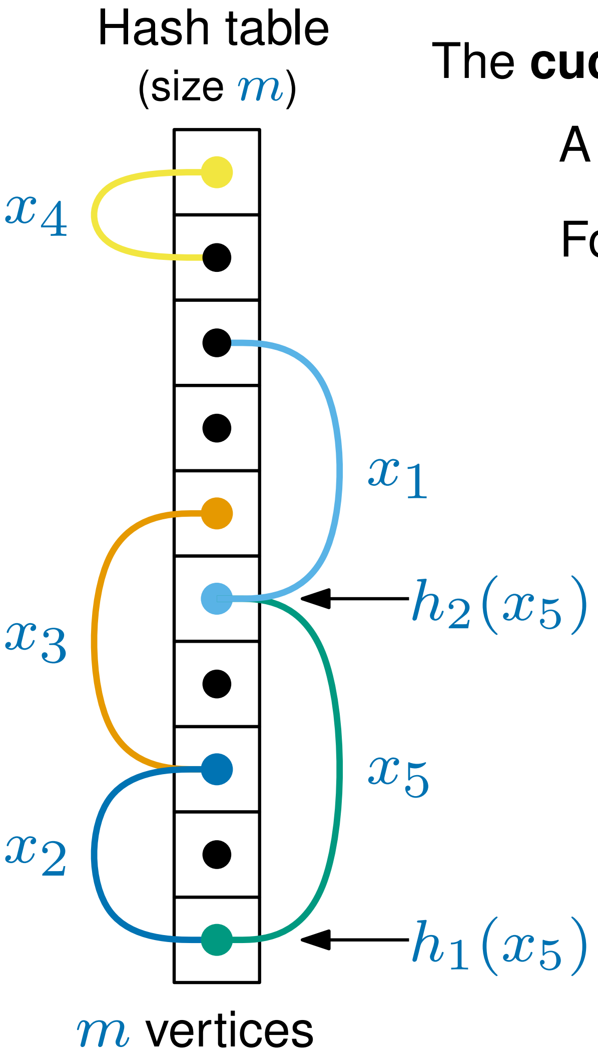
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



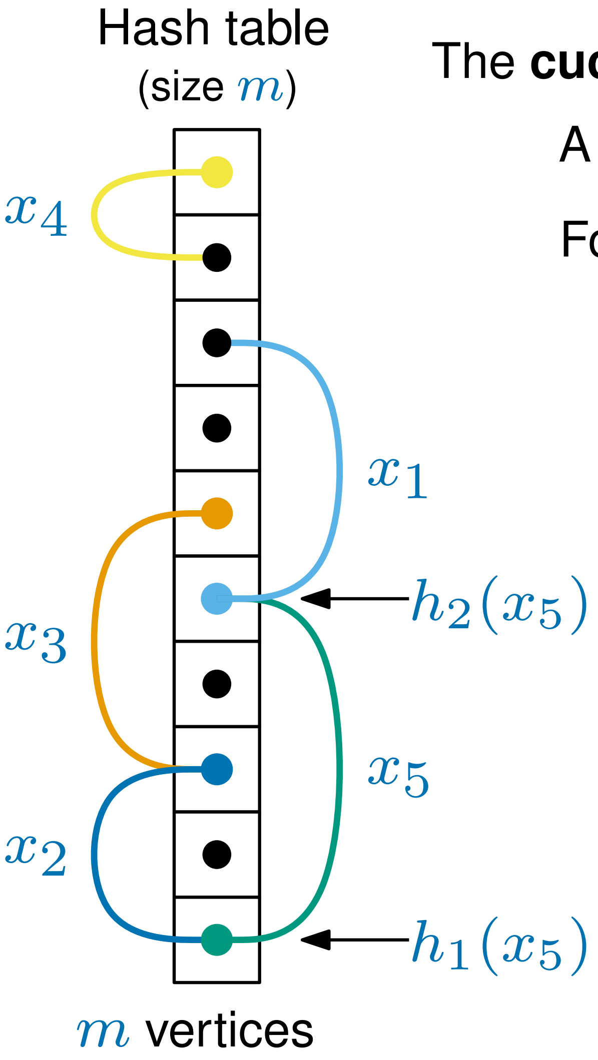
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



The **cuckoo graph**:

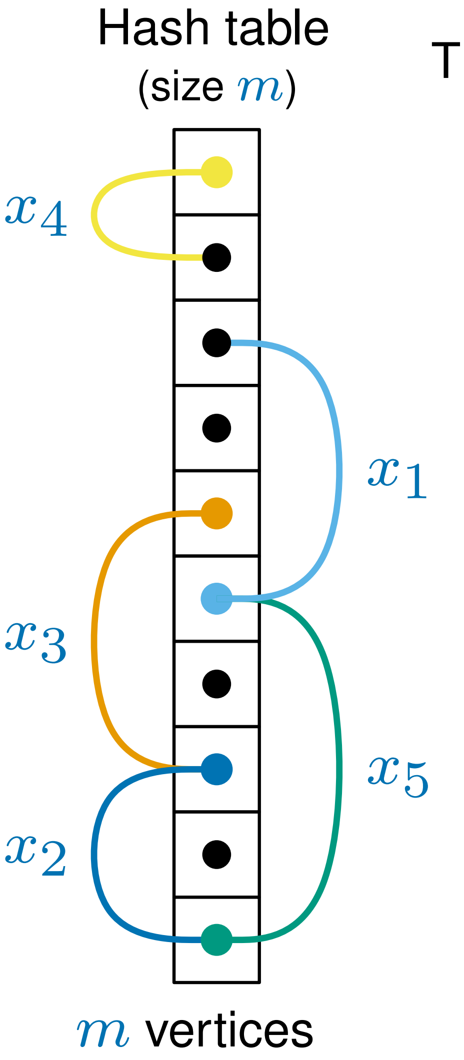
A vertex for each position of the table.

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

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

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

Cuckoo graph



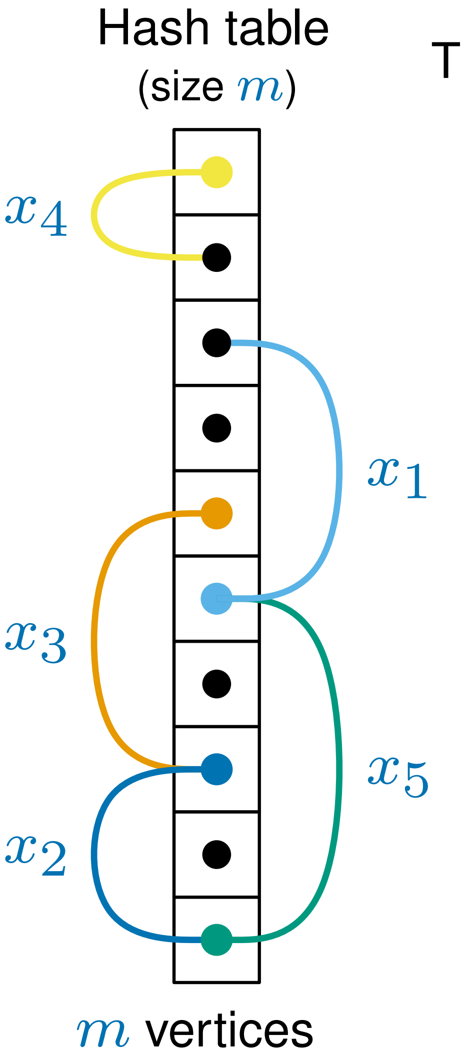
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



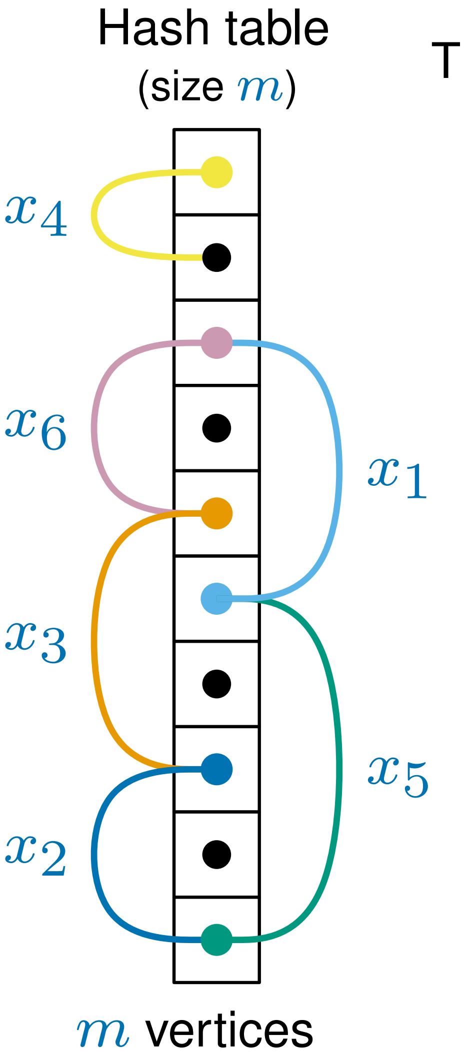
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



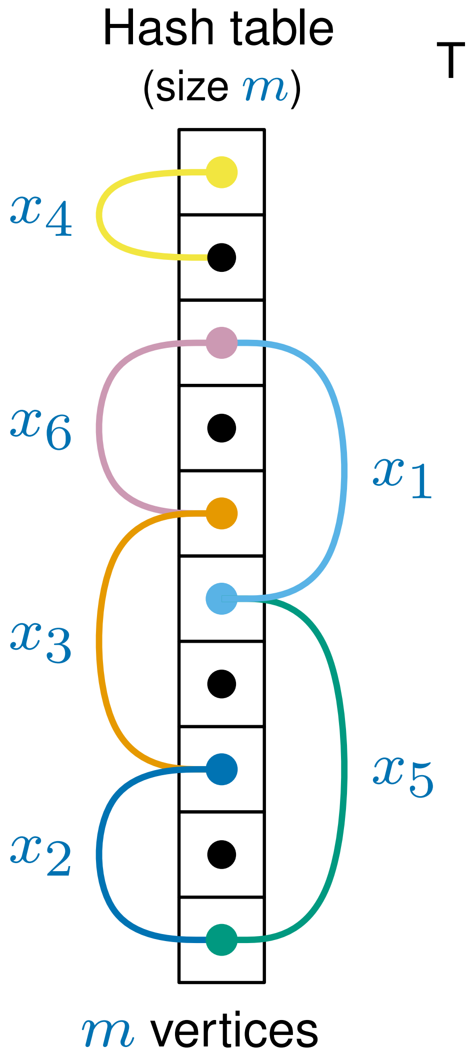
The **cuckoo graph**:

A vertex for each position of the table.

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

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

Cuckoo graph



The **cuckoo graph**:

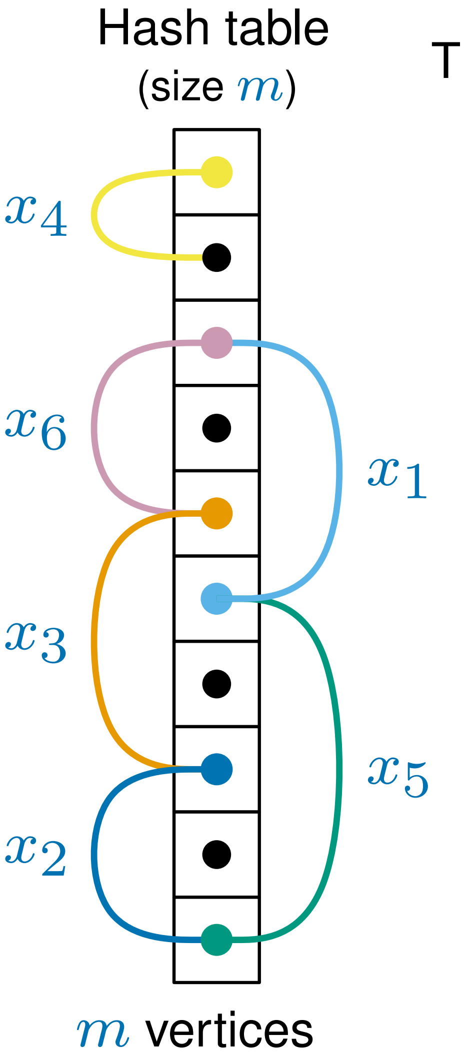
A vertex for each position of the table.

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

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

Inserting key x_6 creates a cycle.

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

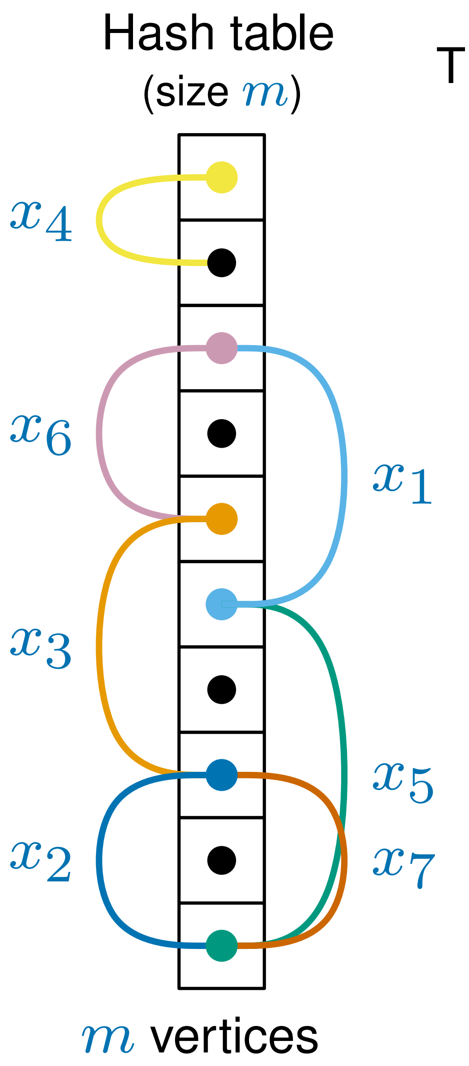
For each key x there is an undirected edge between $h_1(x)$ and $h_2(x)$.

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

Inserting key x_6 creates a cycle.

Cycles are dangerous. . .

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

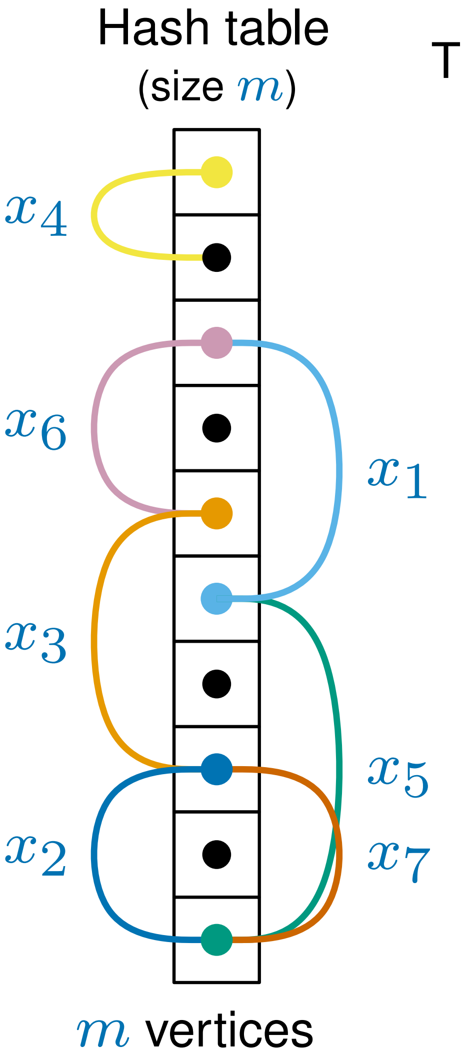
For each key x there is an undirected edge between $h_1(x)$ and $h_2(x)$.

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

Inserting key x_6 creates a cycle.

Cycles are dangerous. . .

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

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

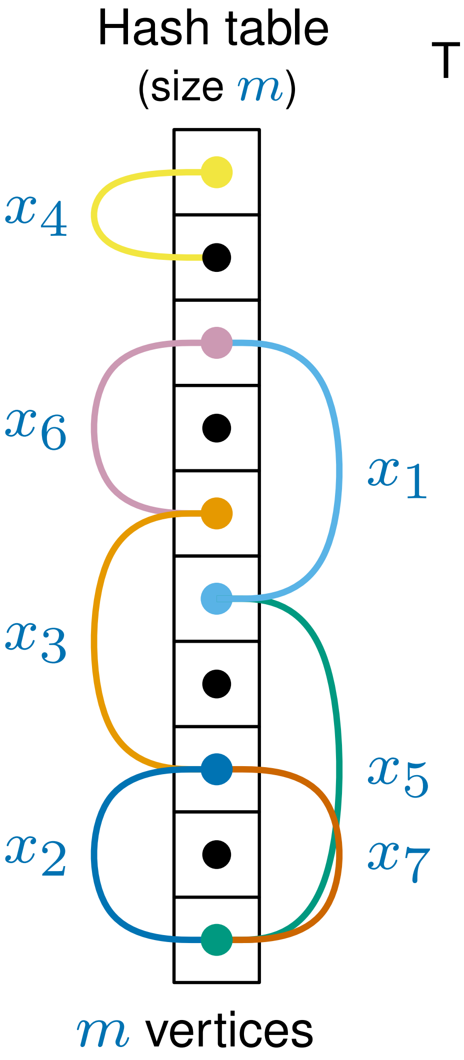
The number of moves performed while adding a key is the length of the corresponding path in the cuckoo graph

Inserting key x_6 creates a cycle.

Cycles are dangerous. . .

When key x_7 is inserted where does it go?

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

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

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

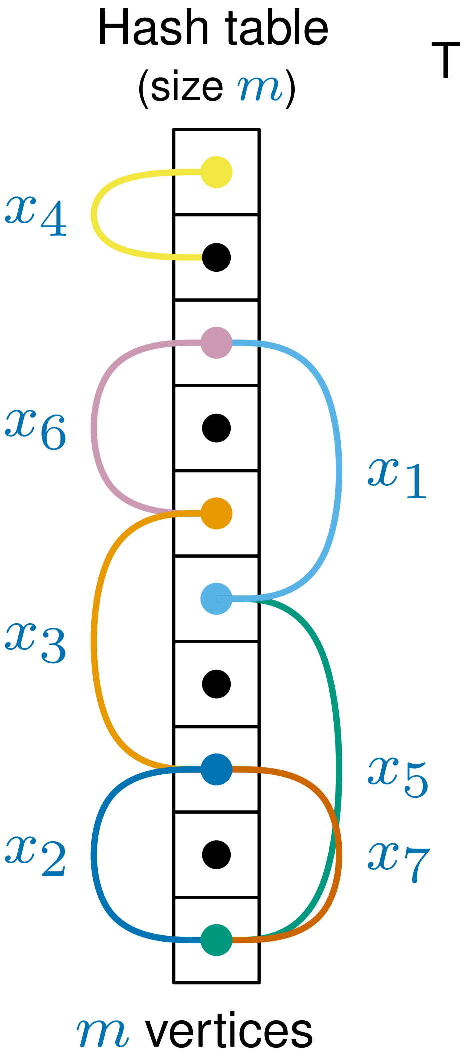
Inserting key x_6 creates a cycle.

Cycles are dangerous. . .

When key x_7 is inserted where does it go?

there are 6 keys but only 5 spaces

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

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

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

Inserting key x_6 creates a cycle.

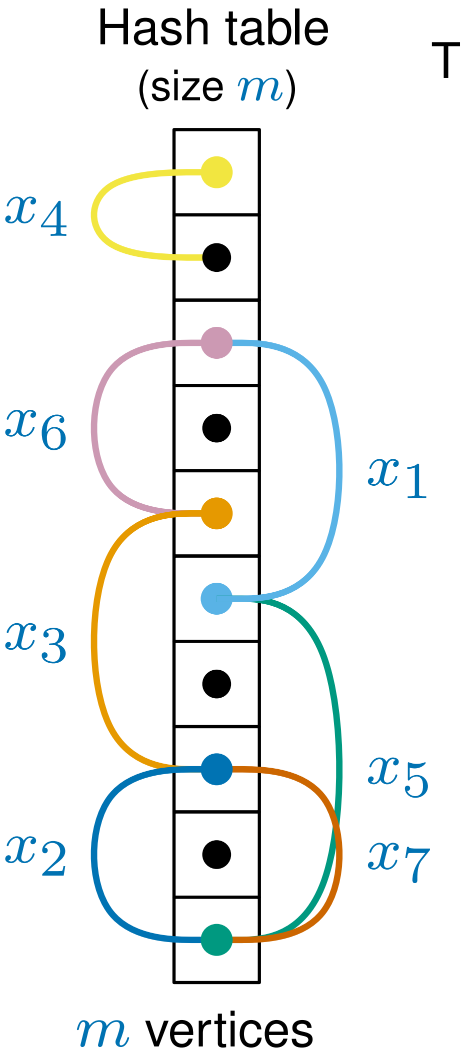
Cycles are dangerous...

When key x_7 is inserted where does it go?

there are 6 keys but only 5 spaces

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

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

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

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

Inserting key x_6 creates a cycle.

Cycles are dangerous. . .

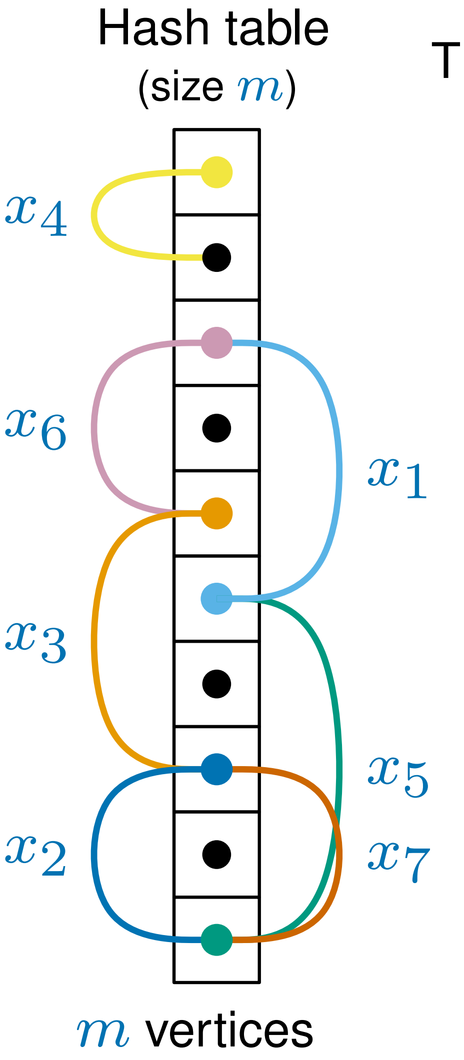
When key x_7 is inserted where does it go?

there are 6 keys but only 5 spaces

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

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

Cuckoo graph



The **cuckoo graph**:

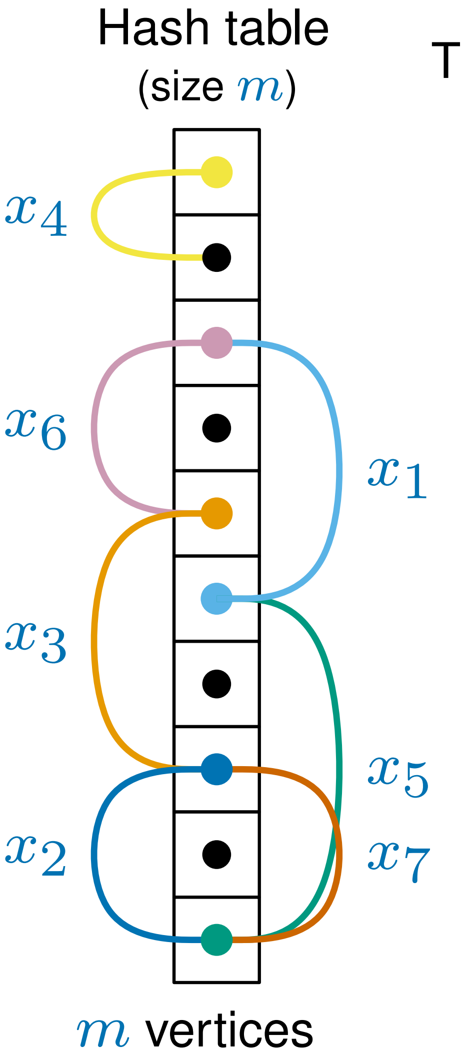
A vertex for each position of the table.

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

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

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

Cuckoo graph



The **cuckoo graph**:

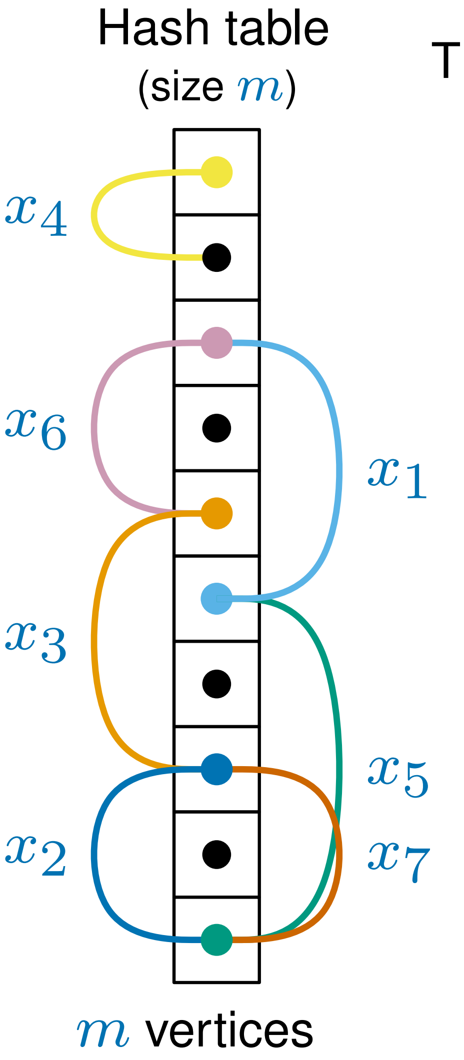
A vertex for each position of the table.

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

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

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

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

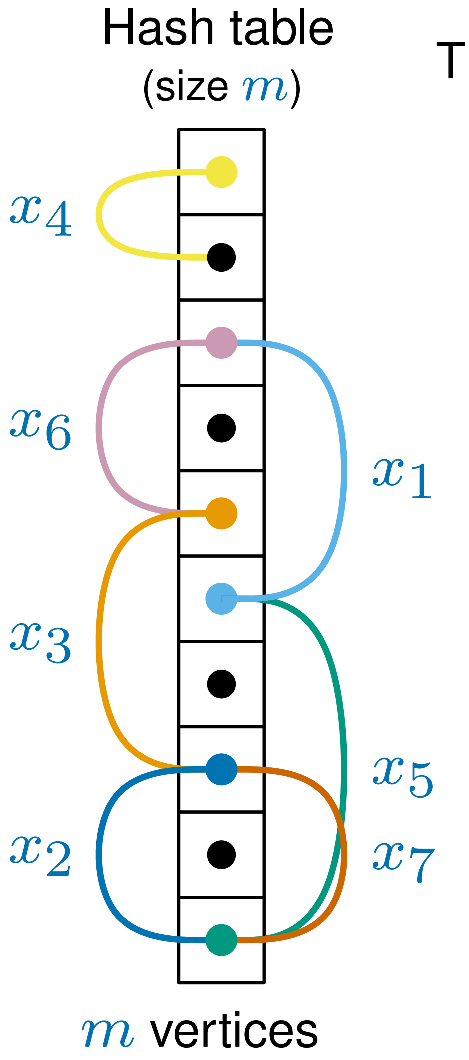
For each key x there is an undirected edge between $h_1(x)$ and $h_2(x)$.

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

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

This is the only way a rehash can happen

Cuckoo graph



The **cuckoo graph**:

A vertex for each position of the table.

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

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

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

This is the only way a rehash can happen

We will analyse the probability of either a cycle or a long path occuring in the graph while inserting any n keys.

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

What does this say?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

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

table size is m

Paths in the cuckoo graph

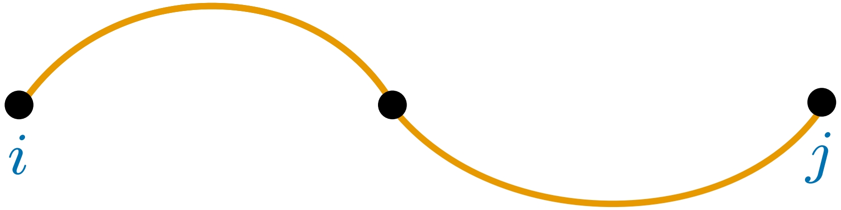
n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?



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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

Probability of a shortest path of length *3* is at most $\frac{1}{8 \cdot m}$

table size is m

Paths in the cuckoo graph

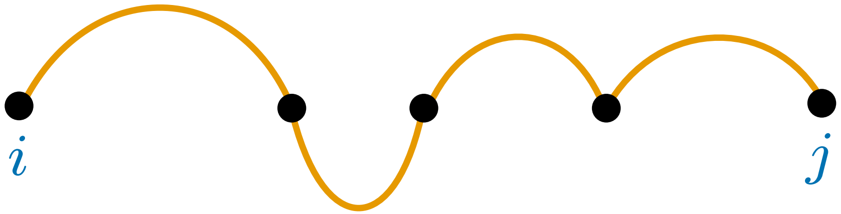
n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?



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

table size is m

Paths in the cuckoo graph

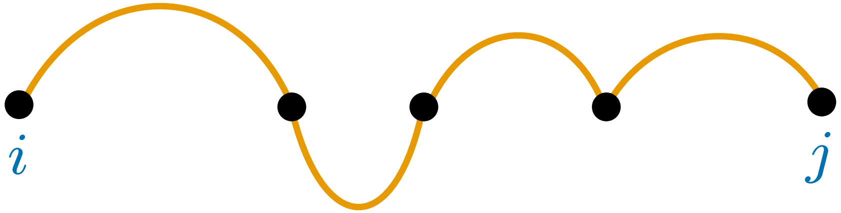
n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?



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

How likely is it that there even is a path?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

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

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

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

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

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

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

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

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

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

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

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

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

(let $c = 2$ for simplicity)

What does this say?

How likely is it that there even is a path?

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

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

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

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

So a path from i to j is rather unlikely to exist

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

What is the proof?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

What is the proof?

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

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

What is the proof?

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

Can we at least see the pictures?

table size is m

Paths in the cuckoo graph

n keys

LEMMA

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

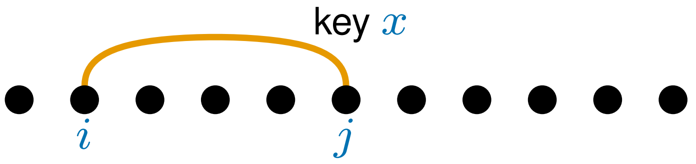
What is the proof?

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

Can we at least see the pictures?

The proof is by induction on the length ℓ:

Base case: $\ell = 1$.

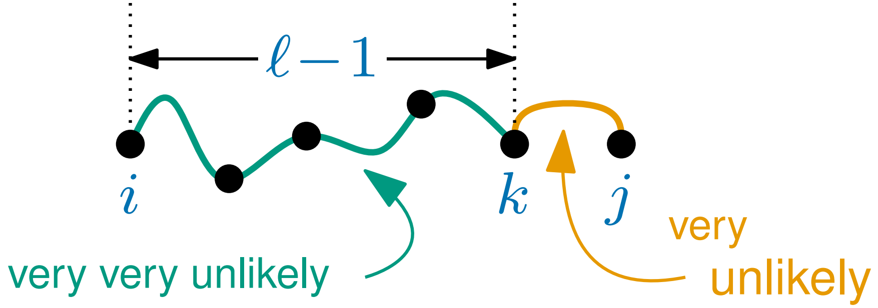


Argue that each key has prob $\frac{2}{m^2}$ to create an edge (i,j)

Union bound over all n keys

Inductive step:

Pick a third point k to split the path



Union bound over all k *then* all keys

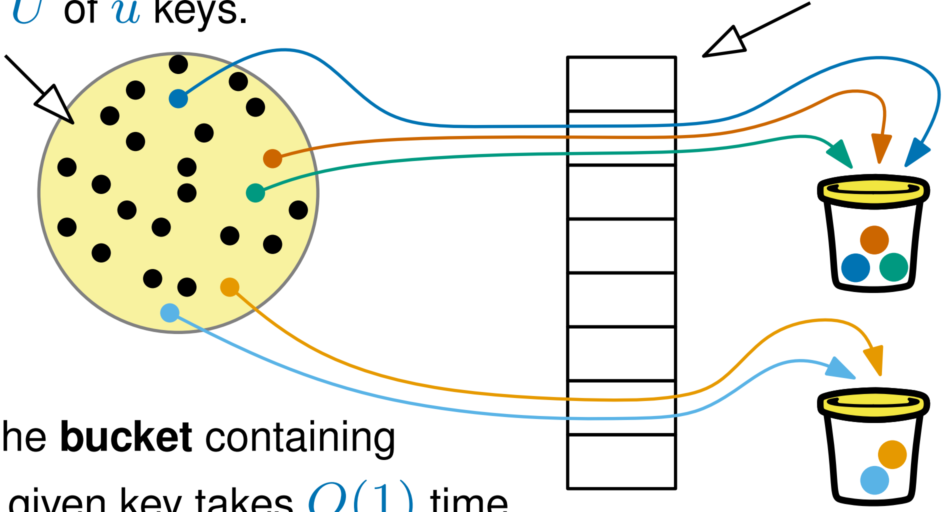
table size is m

Back to the start (again) (again)

n keys

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

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



Universe U of u keys.

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

Collisions are fixed by **~~chaining~~**

bucketing

We require that we can recover

any key from its **bucket** in $O(s)$ time

where s is the number of keys in the ***bucket***

Locating the **bucket** containing

a given key takes $O(1)$ time

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

If our construction has the property that,

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

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

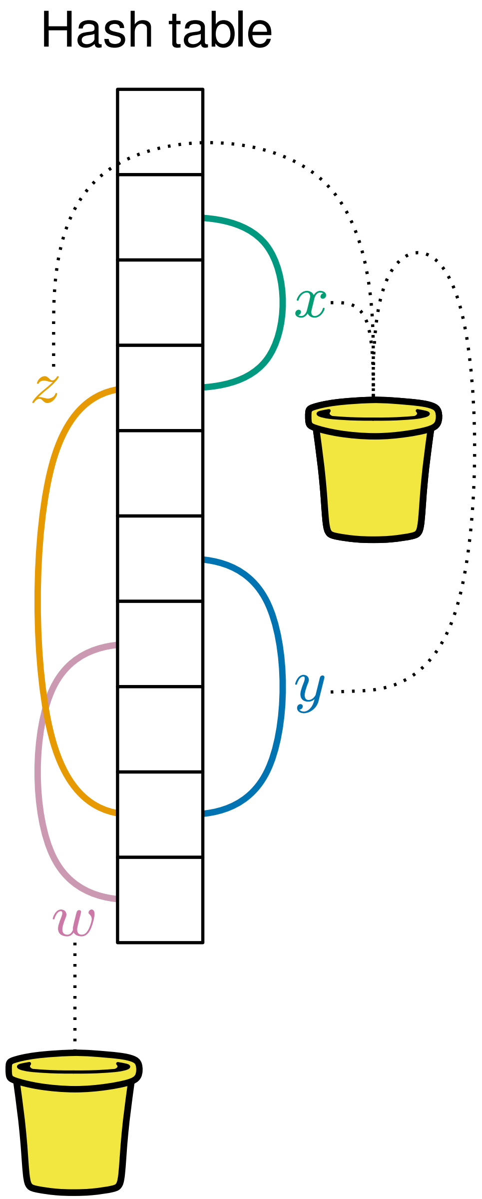
For *any* n operations, the *expected* run-time is $O(1)$ per operation.

table size is m

Don't put all your eggs in one bucket

n keys

Hash table



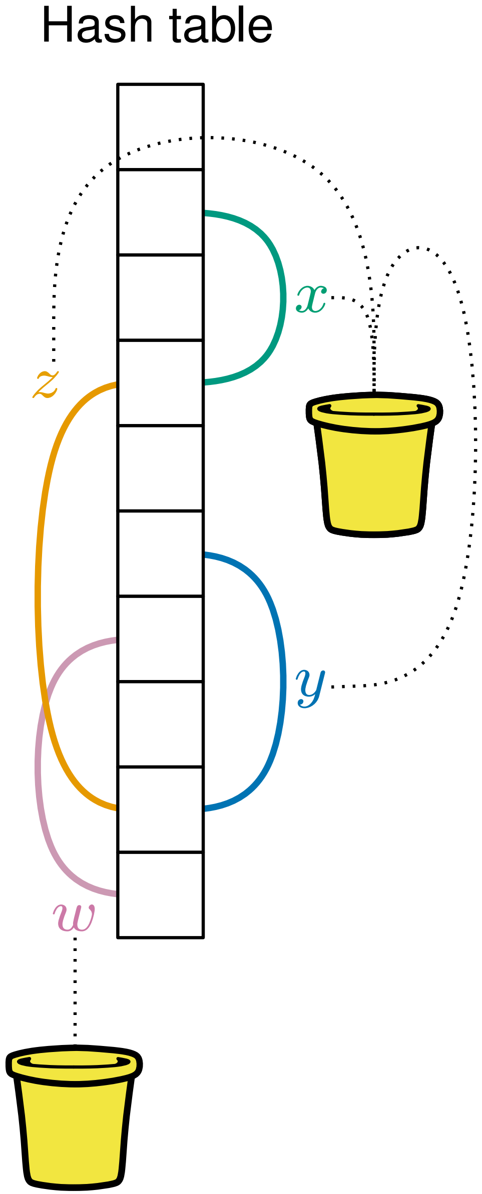
We say that two keys x, y are in the same **bucket** (conceptually)
iff there is a path between $h_1(x)$ and $h_1(y)$
in the cuckoo graph.

table size is m

Don't put all your eggs in one bucket

n keys

Hash table



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

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

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

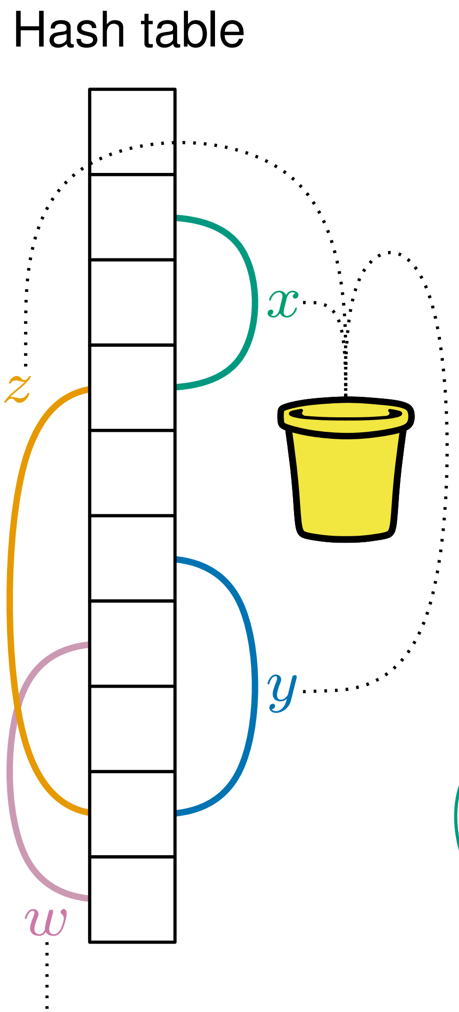
where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

table size is m

Don't put all your eggs in one bucket

n keys



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

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

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

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

LEMMA

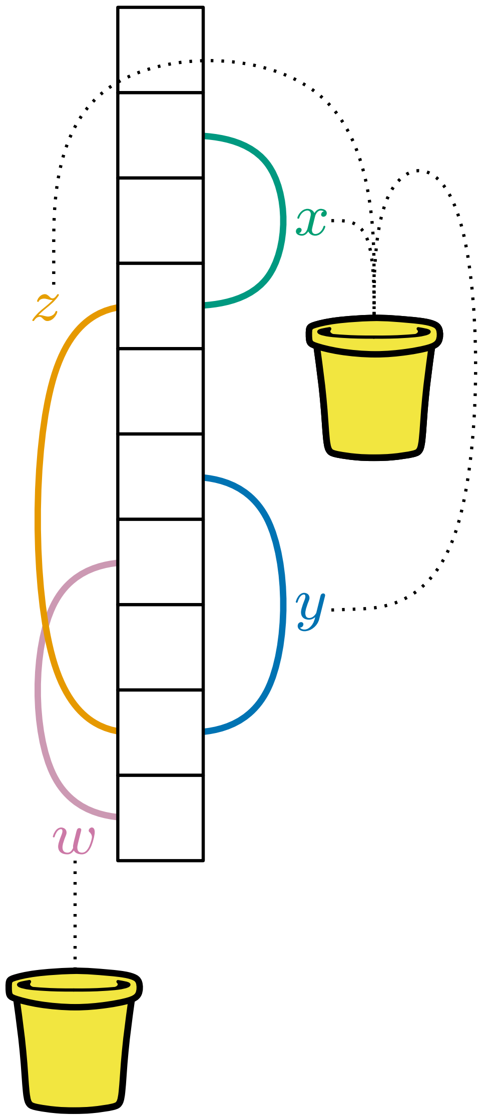
For any positions i and j, and any constant $c > 1$, if $m \geqslant 2cn$ then the probability that there exists a shortest path in the cuckoo graph from i to j with length $\ell \geqslant 1$, is at most $\frac{1}{c^\ell \cdot m}$.

table size is m

Don't put all your eggs in one bucket

n keys

Hash table



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

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

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

where $c > 1$ is a constant.

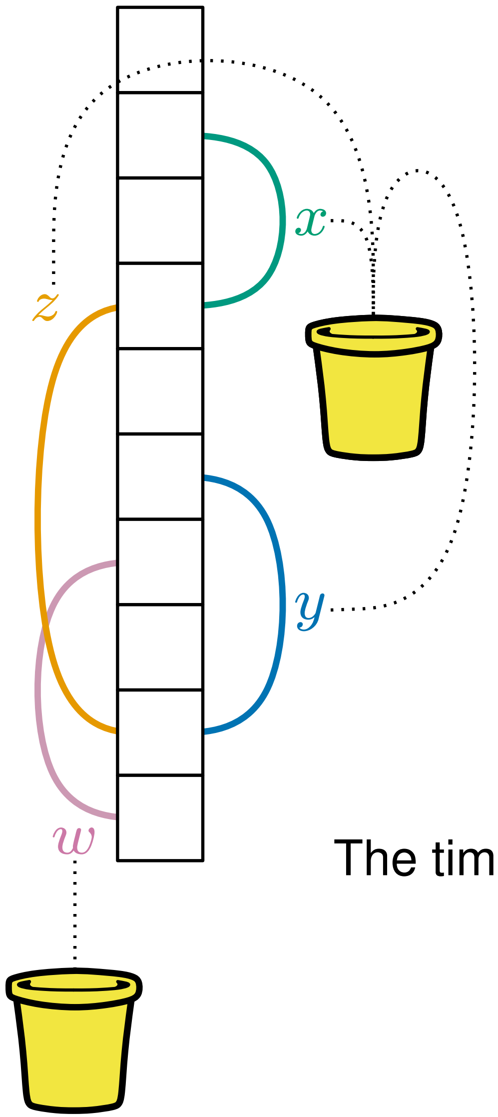
(another union bound over all possible path lengths.)

table size is m

Don't put all your eggs in one bucket

n keys

Hash table



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

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

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

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

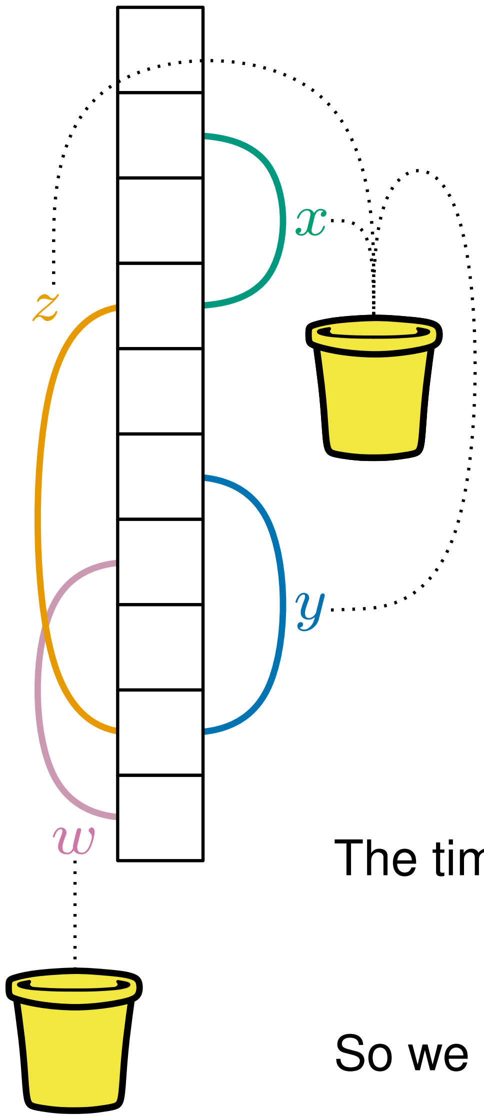
The time for an operation on x is bounded by the number of items in the bucket. *(assuming there are no cycles.)*

table size is m

Don't put all your eggs in one bucket

n keys

Hash table



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

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

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

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

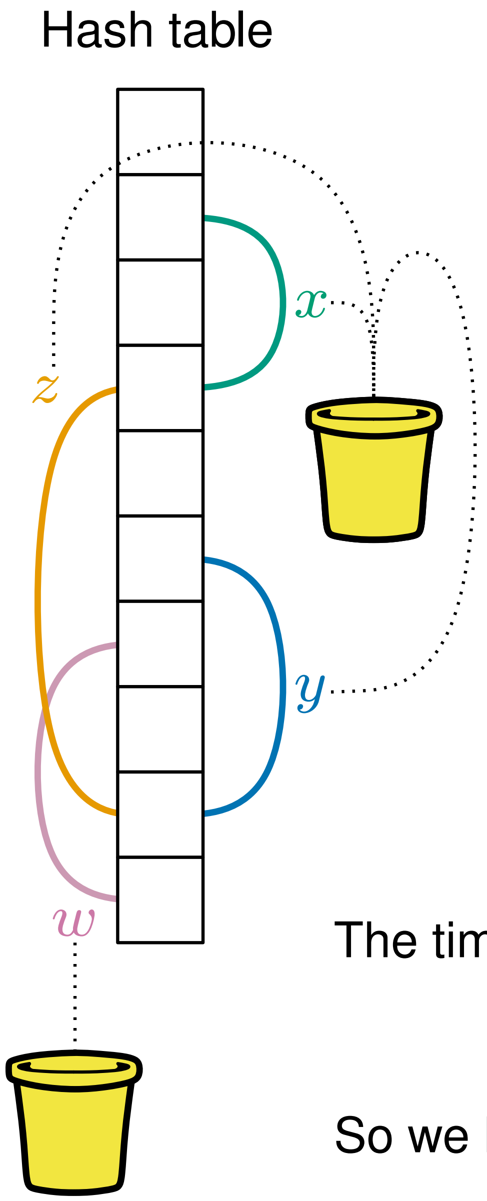
The time for an operation on x is bounded by the number of items in the bucket. *(assuming there are no cycles.)*

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

table size is m

Don't put all your eggs in one bucket

n keys



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

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

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

where $c > 1$ is a constant.

(another union bound over all possible path lengths.)

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

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

Further, lookups take $O(1)$ time in the *worst case*.

Rehashing

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

Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

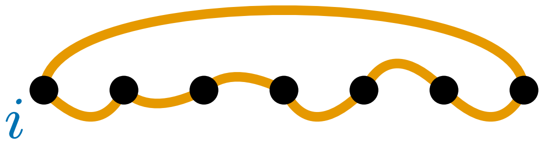
Consider inserting n keys into the table...

Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)



Consider inserting n keys into the table...

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

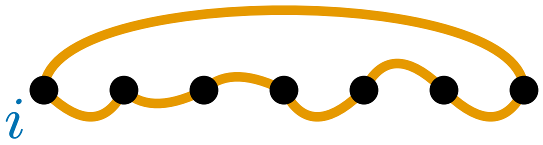
Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting n keys into the table...



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

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

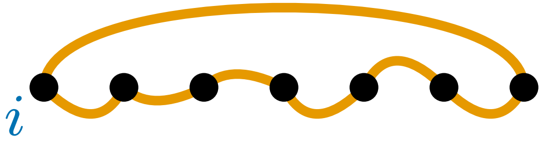
Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)

Consider inserting n keys into the table...



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

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

Lemma

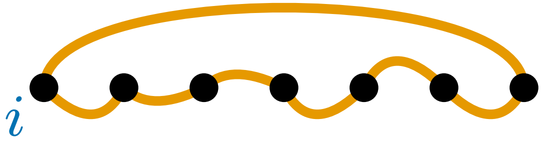
For any positions i and j, and any constant $c > 1$, if $m \geqslant 2cn$ then the probability that there exists a shortest path in the cuckoo graph from i to j with length $\ell \geqslant 1$, is at most $\frac{1}{c^\ell \cdot m}$.

Rehashing

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

However, we would expect there to be cycles every now and then, causing a rehash.

How often does this happen? (sketch proof)



Consider inserting n keys into the table...

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

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

LEMMA

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

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

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

(another union bound over all possible path lengths.)

Rehashing

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

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

(another union bound over all possible path lengths.)

Rehashing

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

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

(another union bound over all possible path lengths.)

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

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

Rehashing

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

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

(another union bound over all possible path lengths.)

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

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

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

Rehashing

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

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

(another union bound over all possible path lengths.)

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

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

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

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

Rehashing

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

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

(another union bound over all possible path lengths.)

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

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

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

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

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

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

Rehashing

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

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

(another union bound over all possible path lengths.)

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

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

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

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

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

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

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

Rehashing

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

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

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

Rehashing

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

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

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

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

Rehashing

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

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

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

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

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

Rehashing

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

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

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

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

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

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

Rehashing

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

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

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

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

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

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

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

Rehashing

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

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

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

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

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

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

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

(you can do this using breadth-first search)

Rehashing

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

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

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

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

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

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

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

(you can do this using breadth-first search)

If there is no cycle, insert all the elements,

this takes $O(n)$ time in expectation (as we have seen).

A word about the assumptions

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

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$ *(where h is picked uniformly at random from H)*

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

A set H of hash functions is **weakly universal** if for any two distinct keys $x, y \in U$,
$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$ *(where h is picked uniformly at random from H)*

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

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

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

$\Pr\left(h(x) = h(y)\right) \leqslant \frac{1}{m}$ *(where h is picked uniformly at random from H)*

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

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

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

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

$$\Pr\left(\bigcap_i h(x_i) = v_i\right) = \frac{1}{m^k}$$

(where h is picked uniformly at random from H)

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

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

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

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

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

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

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

such that $h(x)$ can be computed in $O(1)$ time

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

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

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

such that $h(x)$ can be computed in $O(1)$ time

By changing the cuckoo hashing algorithm to perform a rehash after $\log n$ moves

it can be shown (via a similar but harder proof) that the results still hold

A word about the assumptions

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

We have seen that weakly universal hash families are realistic

where any two keys x, y are independent

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

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

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

such that $h(x)$ can be computed in $O(1)$ time

By changing the cuckoo hashing algorithm to perform a rehash after $\log n$ moves

it can be shown (via a similar but harder proof) that the results still hold

THEOREM

In the **Cuckoo hashing** scheme:

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