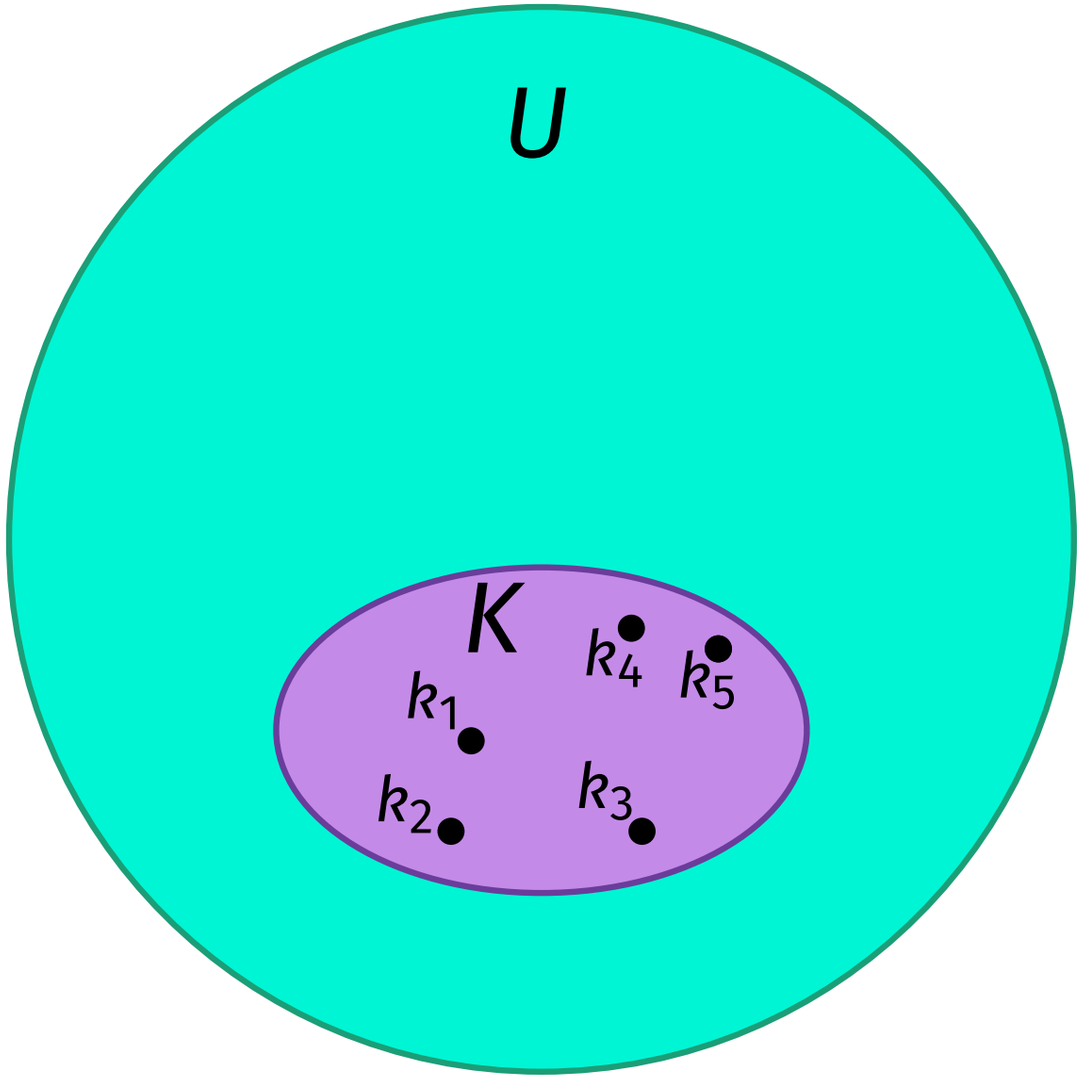
Wie man eine HashMap baut

Tim Hegemann

Offenes Kolloquium am 5. Juni 2024

Recap: Hash Tables

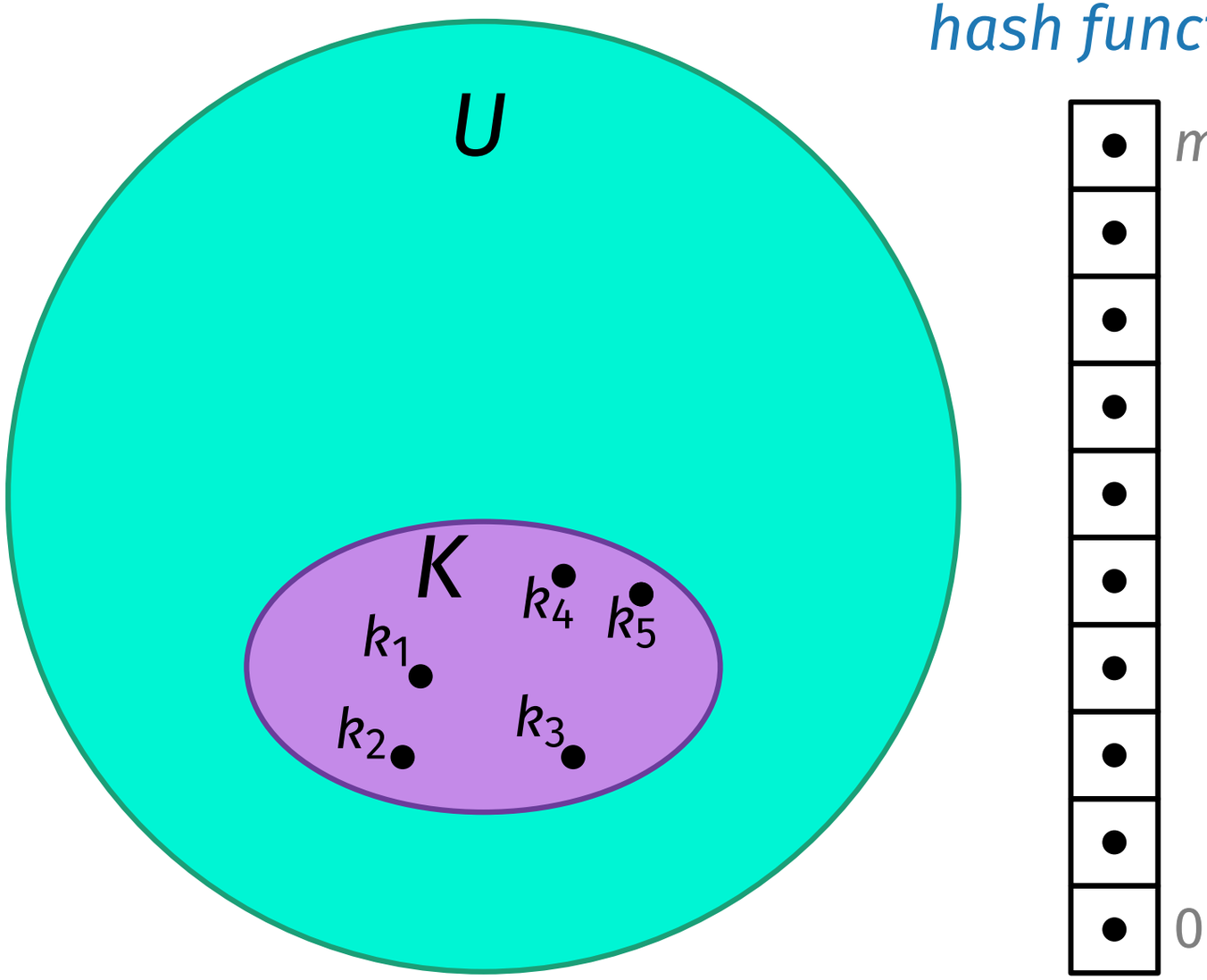
Assumption: large universe U (that means $|U| \gg |K|$)



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

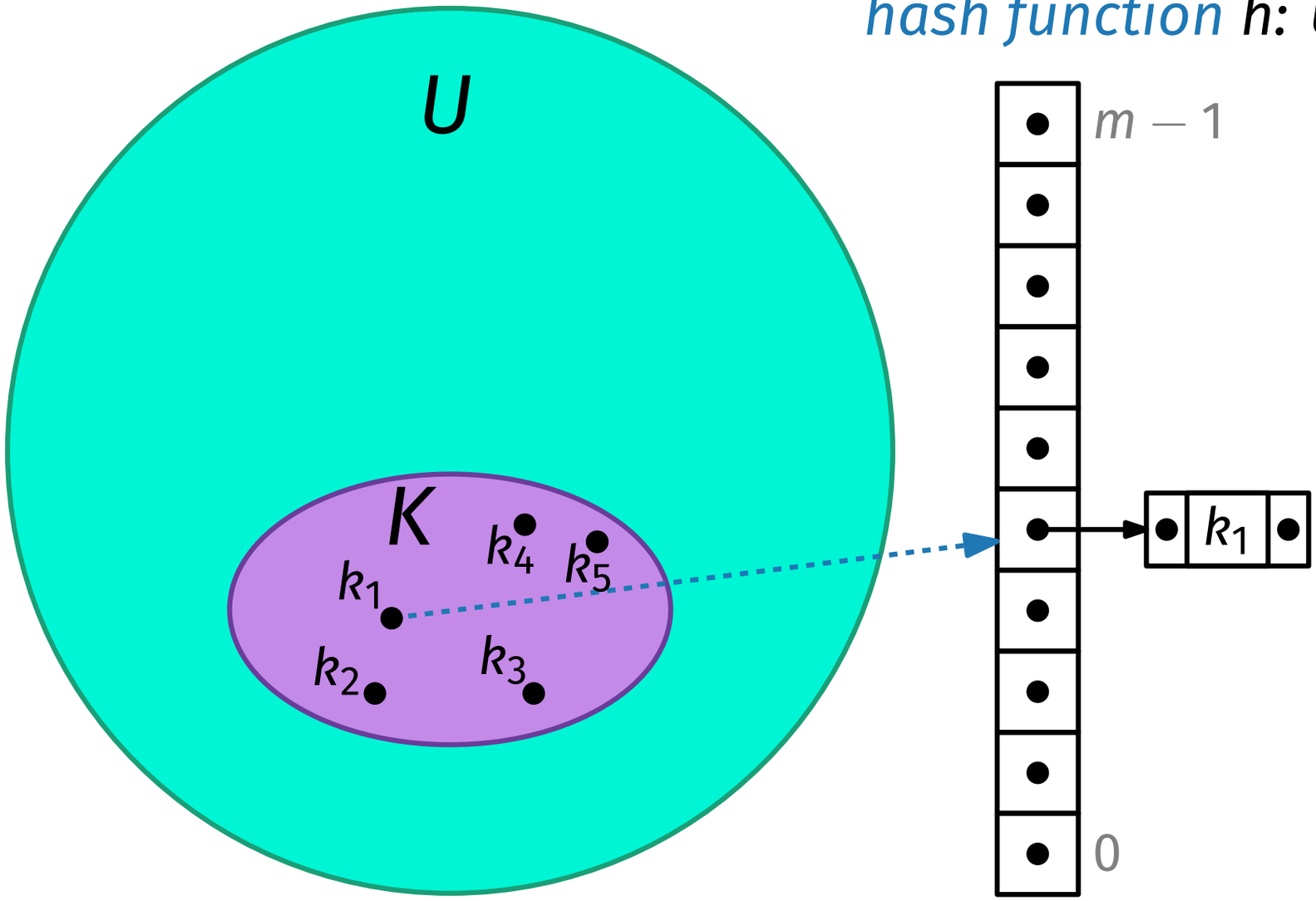
hash function h: $U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

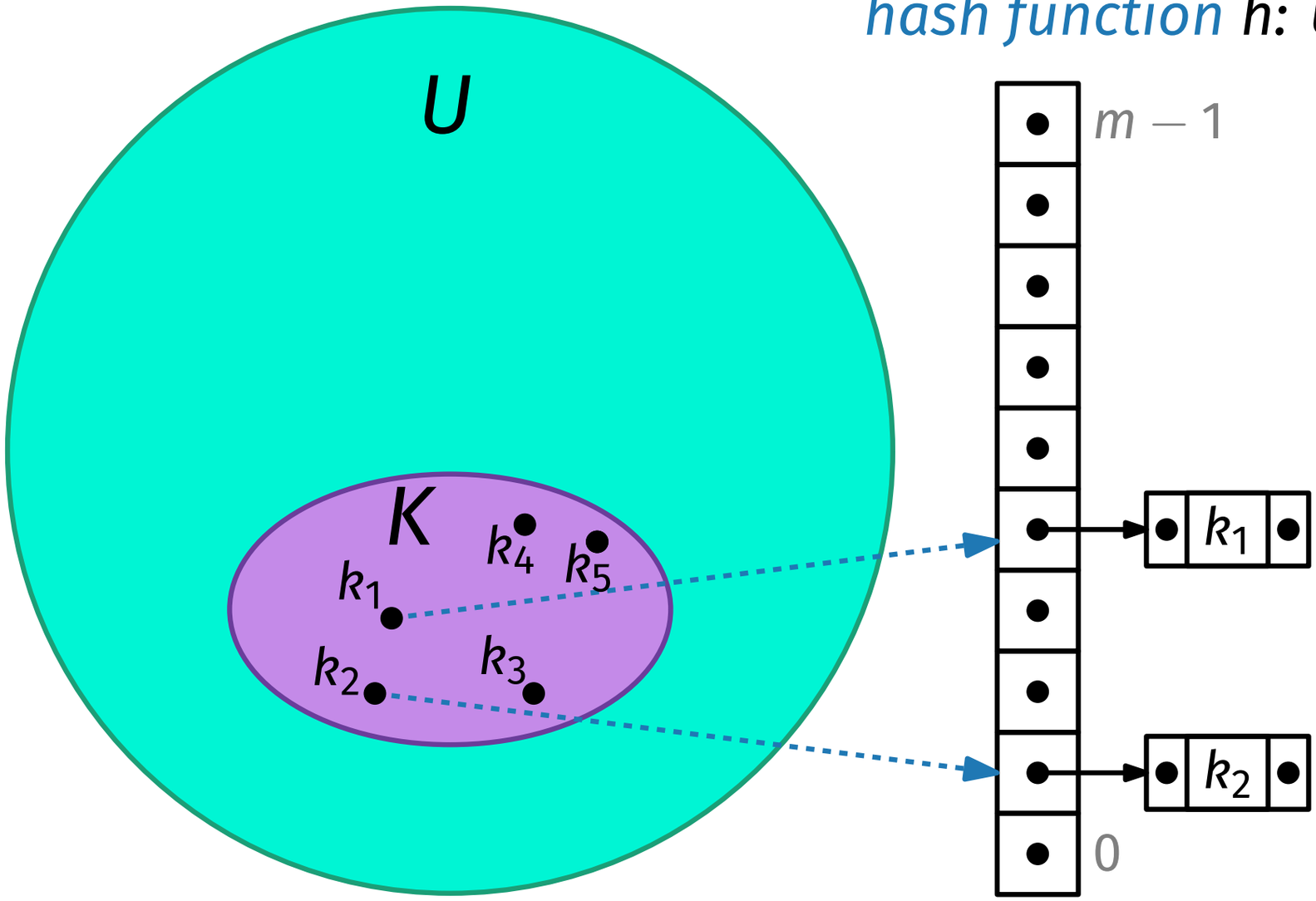
hash function h: $U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

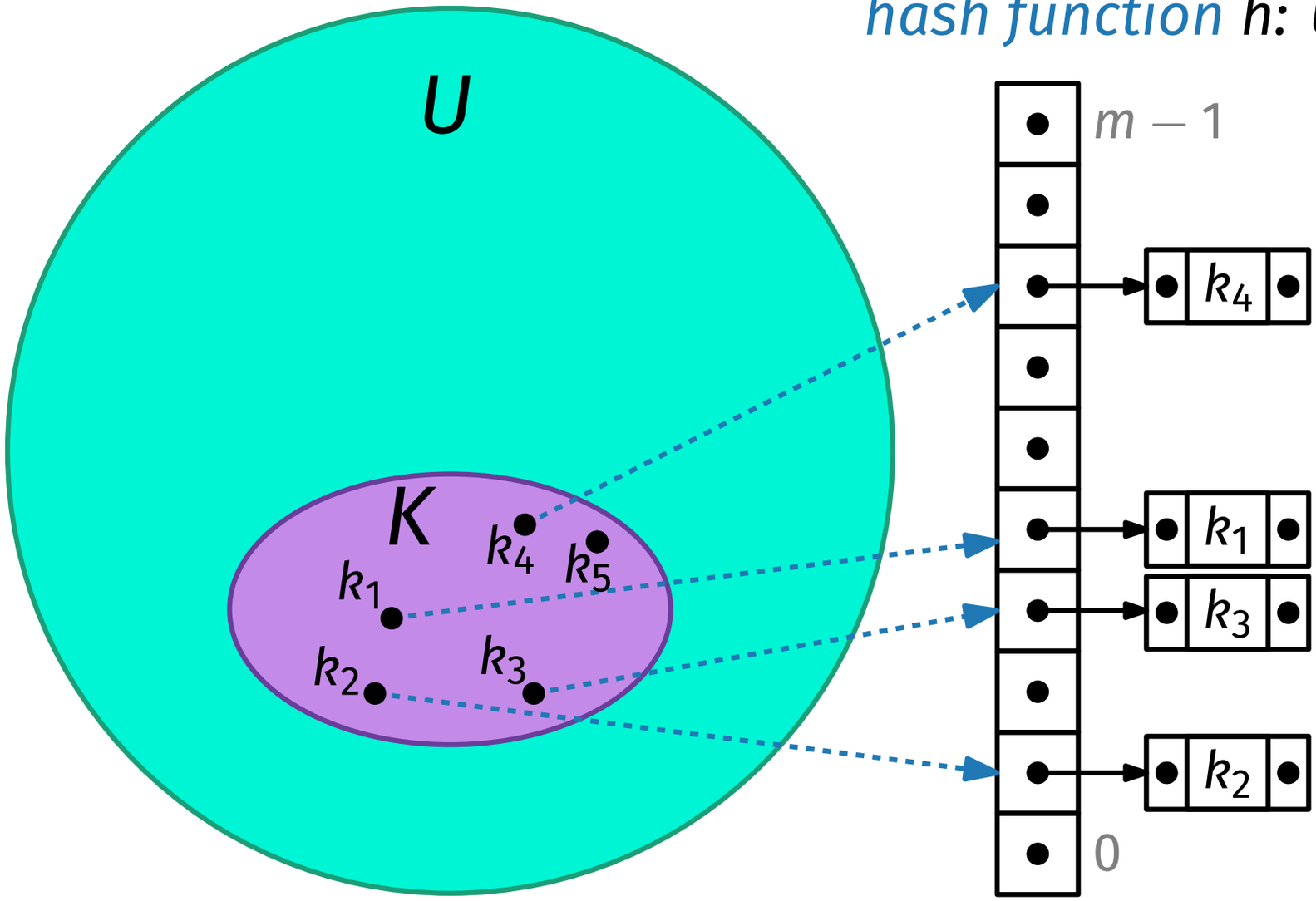
hash function $h: U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

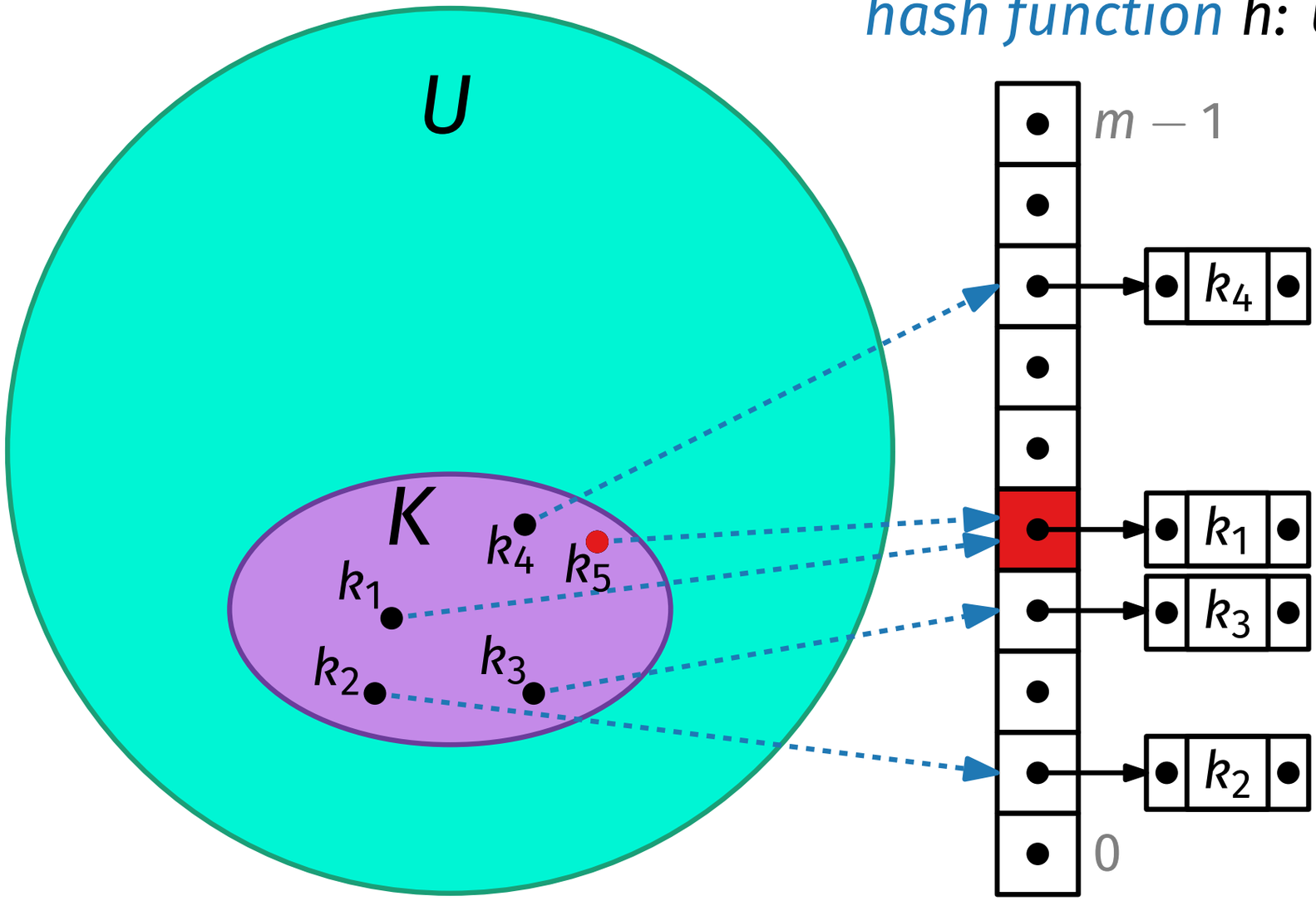
hash function $h: U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

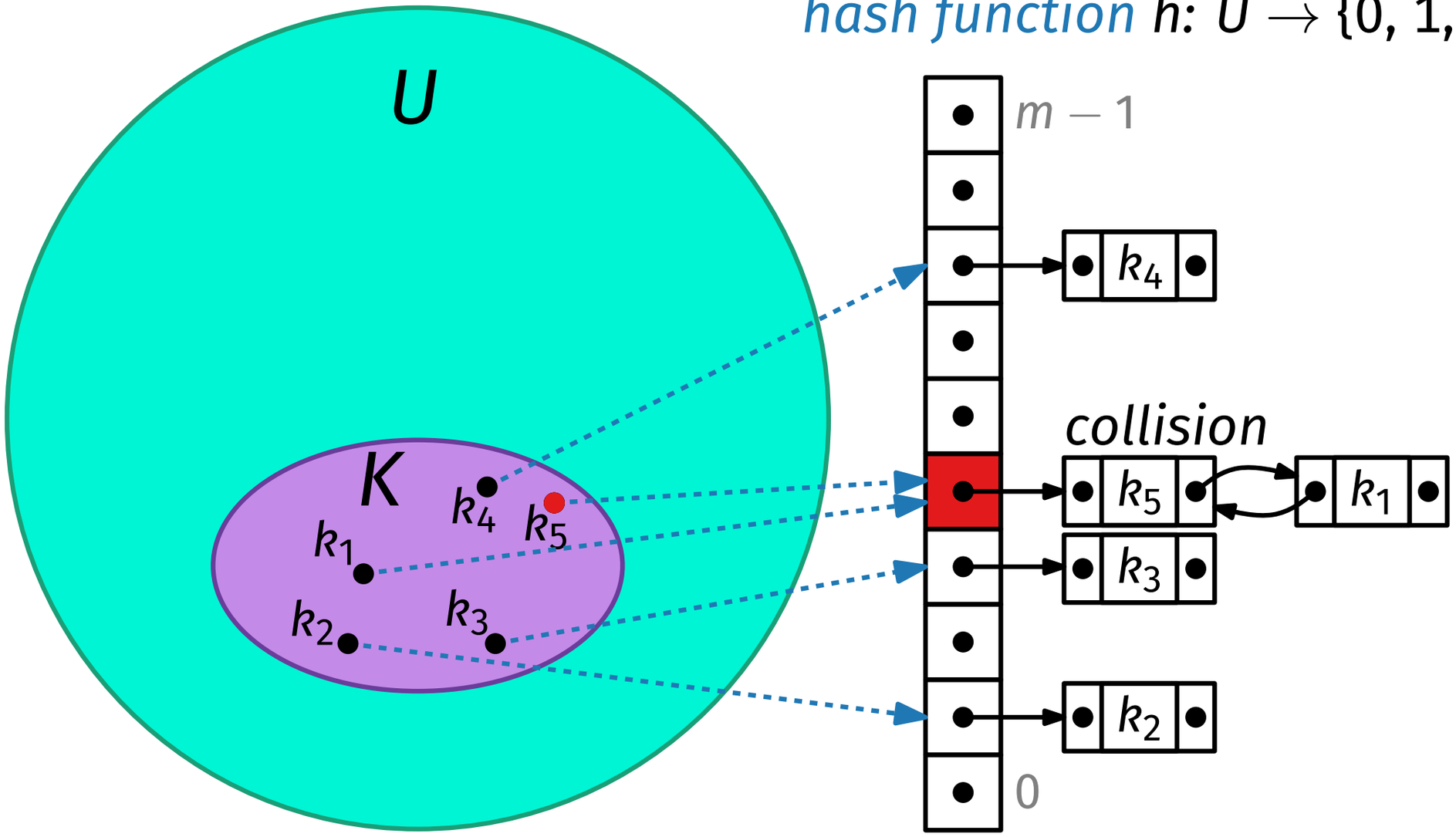
hash function h: $U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

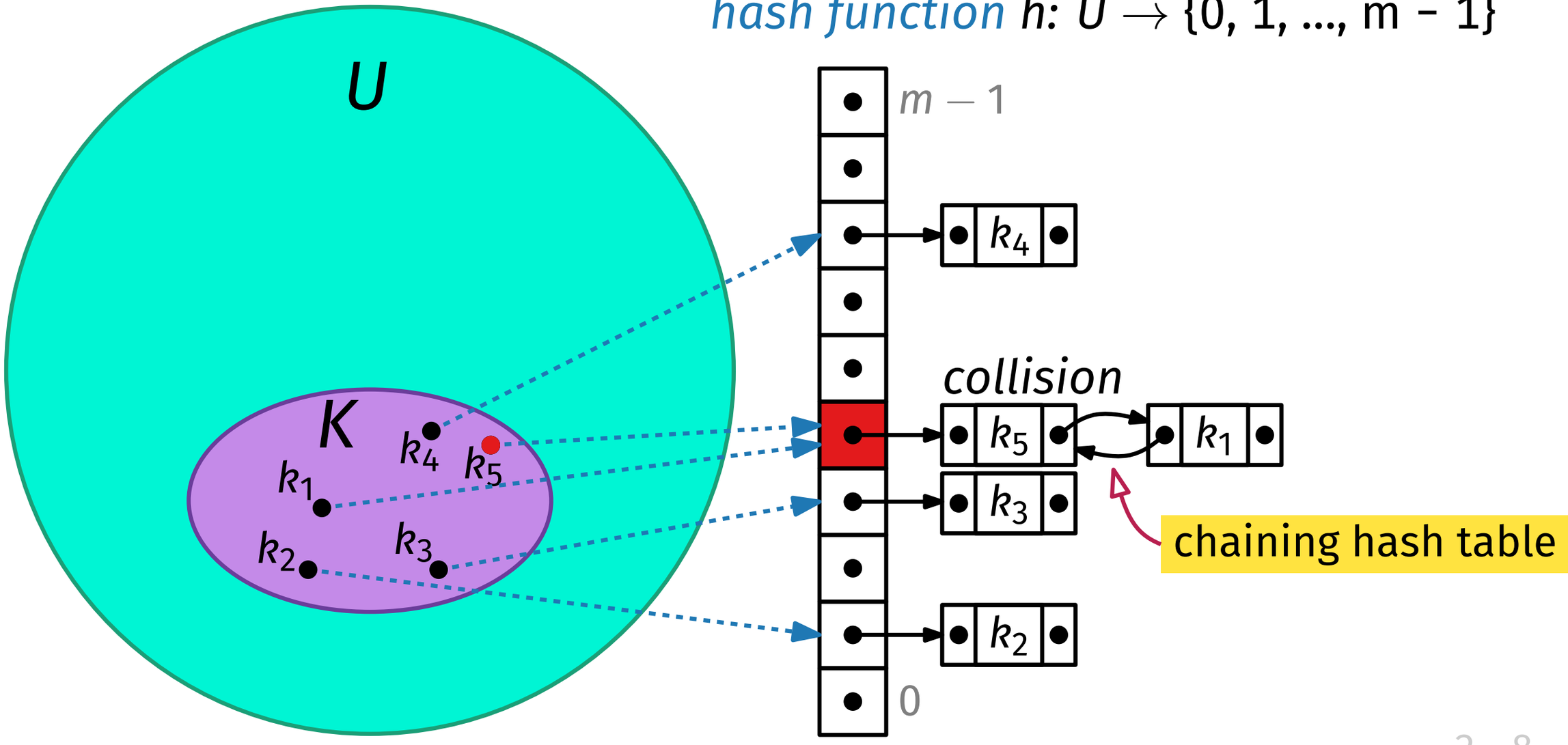
hash function $h: U \to \{0, 1, \ldots, m - 1\}$



Recap: Hash Tables

Assumption: large universe U (that means $|U| \gg |K|$)

hash function h: $U \to \{0, 1, \ldots, m - 1\}$



Asymptotic Runtimes

Asymptotic Runtimes

find	$O(1)$
insert	$O(1)$
delete	$O(1)$

Asymptotic Runtimes

find	$O(1)$ *
insert	$O(1)$ *
delete	$O(1)$ *

* expected

Asymptotic Runtimes

find	$O(1)$ *
insert	$O(1)$ * [1]
delete	$O(1)$ * [(1)]

* expected

[1] amortized

Asymptotic Runtimes

find	$O(1)$ * ¶
insert	$O(1)$ * 1 ¶
delete	$O(1)$ * (1) ¶

load factor: $\alpha = \frac{|K|}{m}$

* expected

1 amortized

¶ if your load factor is reasonably low

Asymptotic Runtimes

find	$O(1)$ * ¶ †
insert	$O(1)$ * 1 ¶ †
delete	$O(1)$ * (1) ¶ †

load factor: $\alpha = \frac{|K|}{m}$

* expected

1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Asymptotic Runtimes

		worst case
find	$O(1)$ * ¶ †	$O(n)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	$O(1)$ * (1) ¶ †	$O(n)$

load factor: $\alpha = \frac{|K|}{m}$

* expected

1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Worst Case – Does it Matter?

Worst Case – Does it Matter?

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"foo":"xyz","bar":"abc"}' \
  http://localhost:3000/api/json}}
```

Worst Case – Does it Matter?

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"foo":"xyz","bar":"abc"}' \
  http://localhost:3000/api/json}}
```

That JSON object will most likely end up in a hash map.

Worst Case – Does it Matter?

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"foo":"xyz","bar":"abc"}' \
  http://localhost:3000/api/json}}
```

That JSON object will most likely end up in a hash map.

Adding n elements to a hash map has worst-case runtime $O(n^2)$

Worst Case – Does it Matter?

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"foo":"xyz","bar":"abc"}' \
  http://localhost:3000/api/json}}
```

That JSON object will most likely end up in a hash map.

Adding n elements to a hash map has worst-case runtime $O(n^2)$

Algorithmic Complexity Attack (commonly known as HashDoS)

Worst Case – Does it Happen?

Worst Case – Does it Happen?

java.lang.String.hashCode()

```
int result = 0;
int end = fromIndex + length;
for (int i = fromIndex; i < end; i++) {
    result = 31 * result + JLA.getUTF16Char(value, i);
}
return result;
```

Worst Case – Does it Happen?

java.lang.String.hashCode()

```
int result = 0;
int end = fromIndex + length;
for (int i = fromIndex; i < end; i++) {
    result = 31 * result + JLA.getUTF16Char(value, i);
}
return result;
```

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

Worst Case – Does it Happen?

java.lang.String.hashCode()

```
int result = 0;
int end = fromIndex + length;
for (int i = fromIndex; i < end; i++) {
    result = 31 * result + JLA.getUTF16Char(value, i);
}
return result;
```

$h(s) = \Sigma\ 31^{n-i} \cdot s_i$

$h(\text{"ABC"}) = 31 \cdot (31 \cdot (65) + 66) + 67$

Worst Case – Does it Happen?

java.lang.String.hashCode()

```
int result = 0;
int end = fromIndex + length;
for (int i = fromIndex; i < end; i++) {
    result = 31 * result + JLA.getUTF16Char(value, i);
}
return result;
```

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

$$h(\text{"ABC"}) = 31 \cdot (31 \cdot (65) + 66) + 67$$

$$h(\text{"ABC"}) = 31^2 \cdot 65 + 31^1 \cdot 66 + 31^0 \cdot 67$$

Equivalent Substrings

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tta".hashCode() = 115169

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tta".hashCode() = 115169

"uUa".hashCode() = 115169

"v6a".hashCode() = 115169

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tttt".hashCode() = 3570944

"tta".hashCode() = 115169

"uUa".hashCode() = 115169

"v6a".hashCode() = 115169

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tta".hashCode() = 115169

"uUa".hashCode() = 115169

"v6a".hashCode() = 115169

"tttt".hashCode() = 3570944

"ttuU".hashCode() = 3570944

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tta".hashCode() = 115169

"uUa".hashCode() = 115169

"v6a".hashCode() = 115169

"tttt".hashCode() = 3570944

"ttuU".hashCode() = 3570944

"ttv6".hashCode() = 3570944

"uUtt".hashCode() = 3570944

"v6tt".hashCode() = 3570944

"v6uU".hashCode() = 3570944

…

Equivalent Substrings

$$h(s) = \Sigma\ 31^{n-i} \cdot s_i$$

"tt".hashCode() = 3712

"uU".hashCode() = 3712

"v6".hashCode() = 3712

"tta".hashCode() = 115169

"uUa".hashCode() = 115169

"v6a".hashCode() = 115169

"tttt".hashCode() = 3570944	0 0
"ttuU".hashCode() = 3570944	0 1
"ttv6".hashCode() = 3570944	0 2
"uUtt".hashCode() = 3570944	1 0
"v6tt".hashCode() = 3570944	2 0
"v6uU".hashCode() = 3570944	2 1
…	…

Equivalent Substrings

How do we solve this?

Equivalent Substrings

How do we solve this?

Randomize your hash function!

Further Reading – Part I

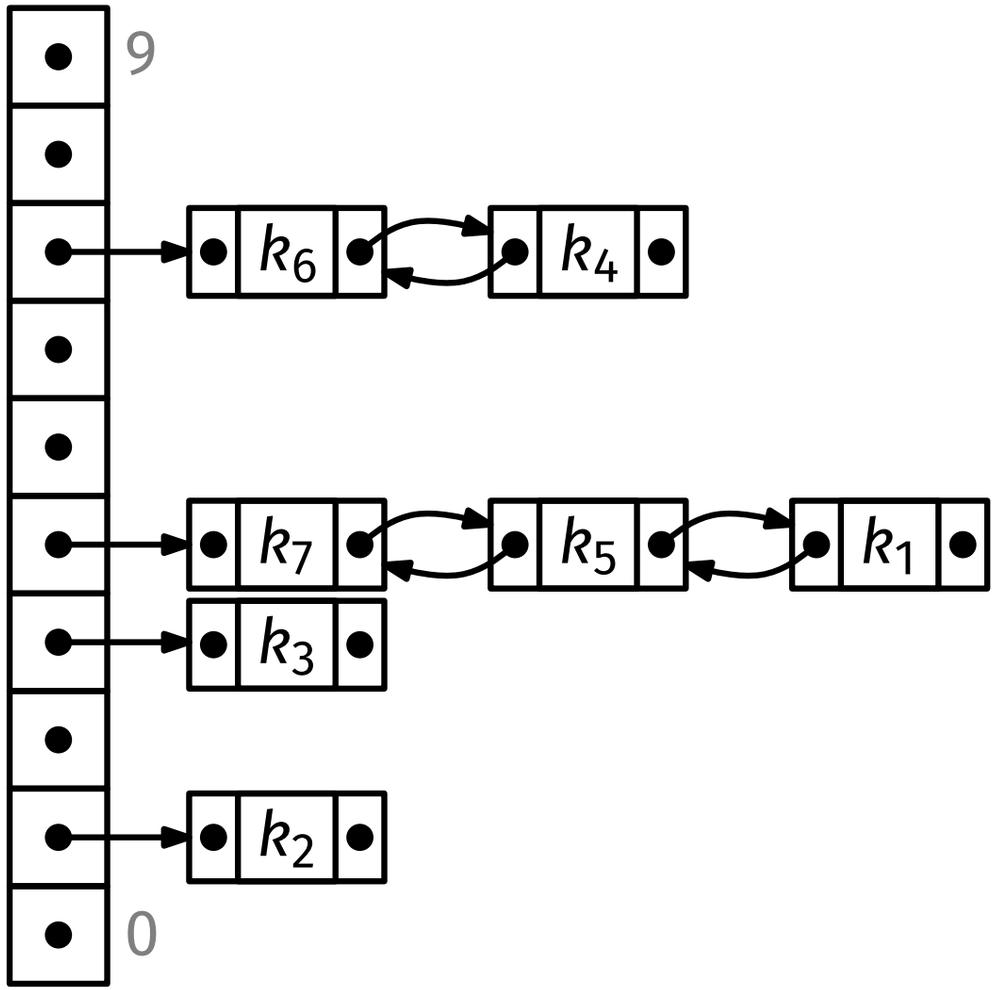
“Effective Denial of Service attacks against web application platforms” by alech and zeri

[https://media.ccc.de/v/28c3-4680]

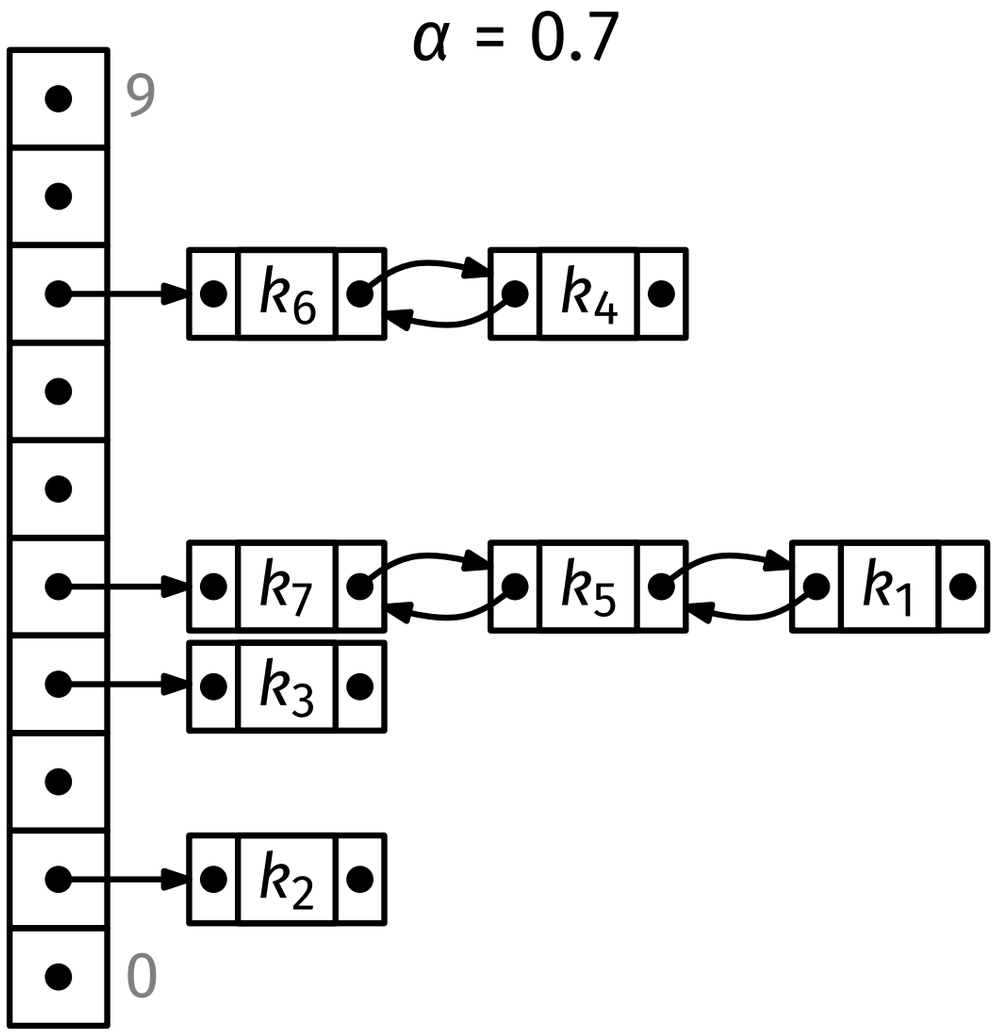
“Hash-flooding DoS reloaded: attacks and defenses” by djb, Jean-Philippe Aumasson, and Martin Bloßlet

[https://media.ccc.de/v/29c3-5152]

Chaining Hash Maps – Operations



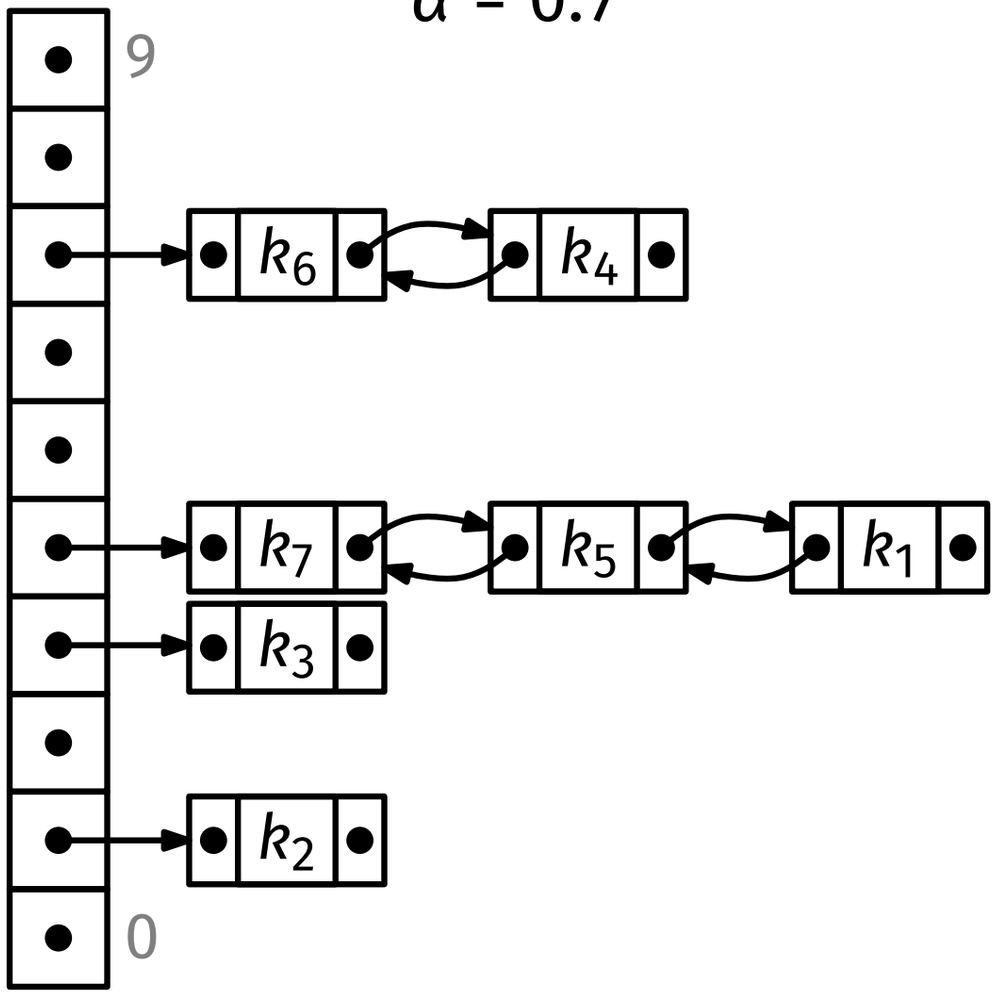
Chaining Hash Maps – Operations



Chaining Hash Maps – Operations

$\alpha = 0.7$

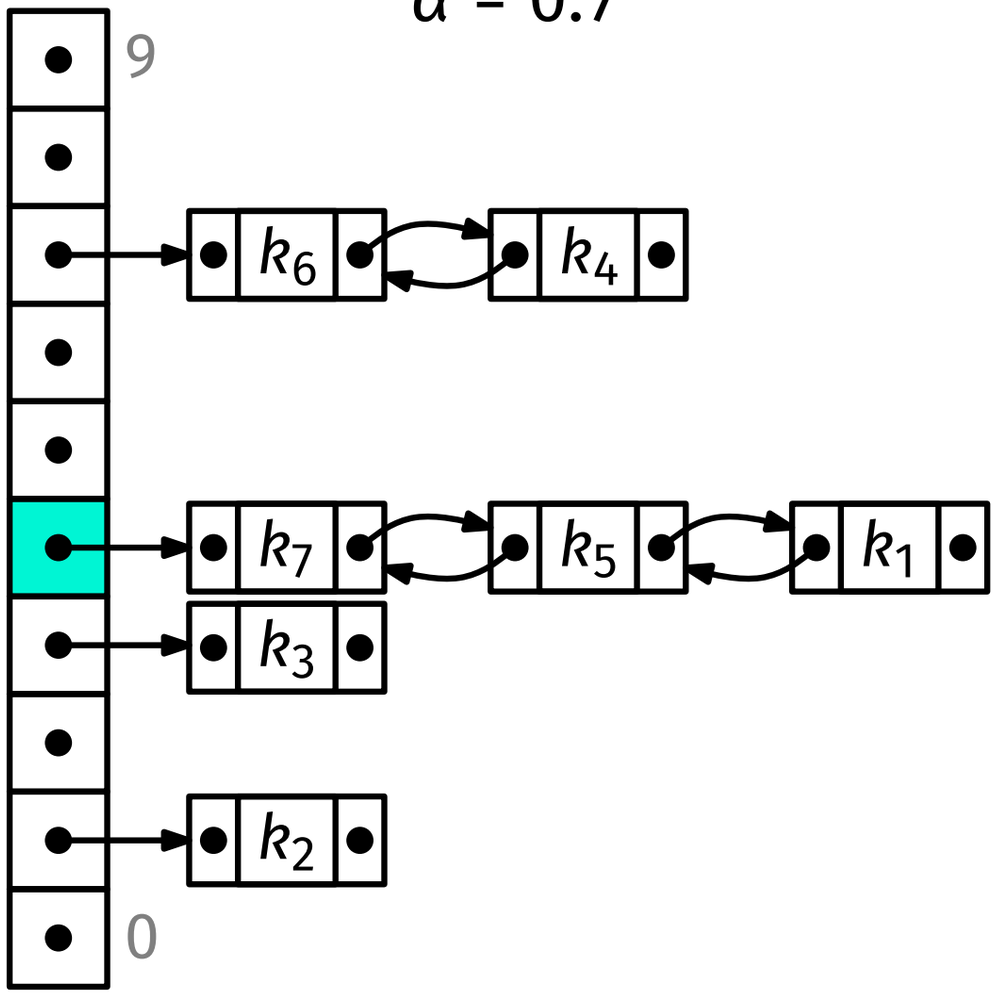
find(k_5) with $h(k_5) = 4$



Chaining Hash Maps – Operations

$\alpha = 0.7$

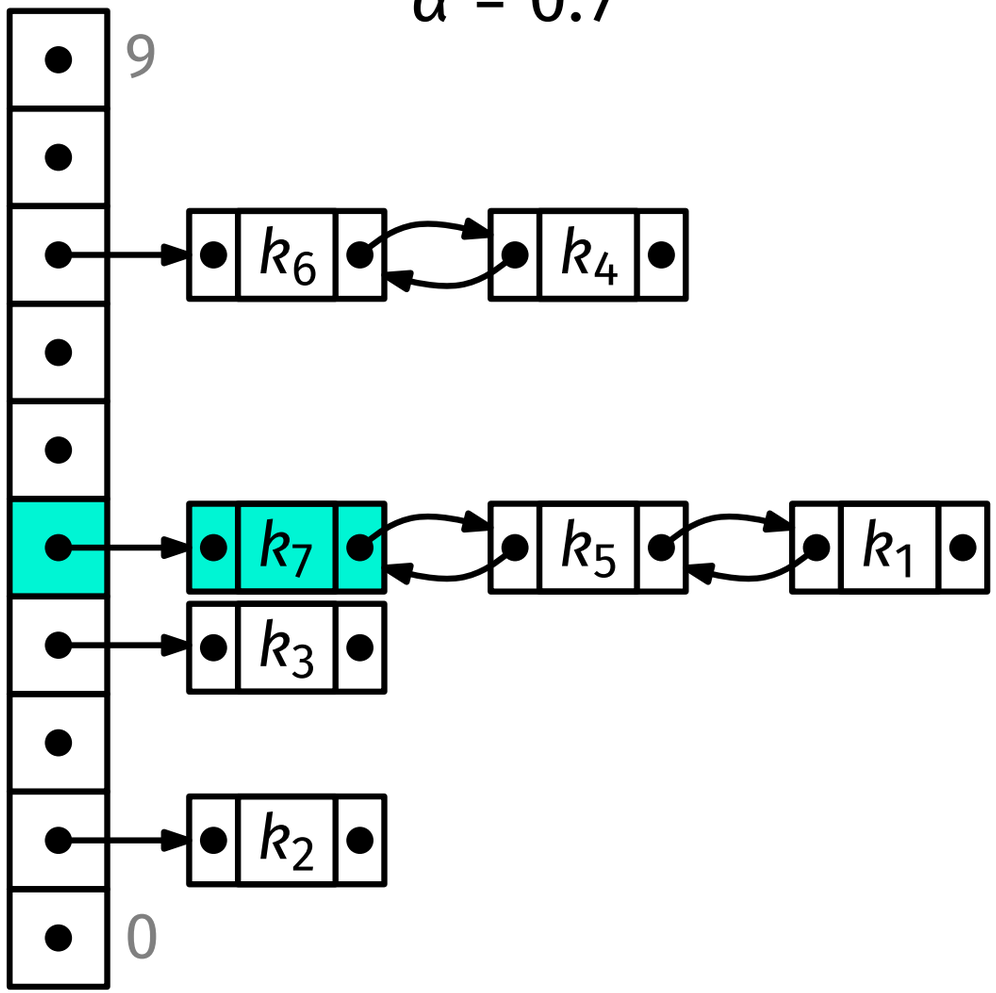
find(k_5) with $h(k_5) = 4$



Chaining Hash Maps – Operations

$\alpha = 0.7$

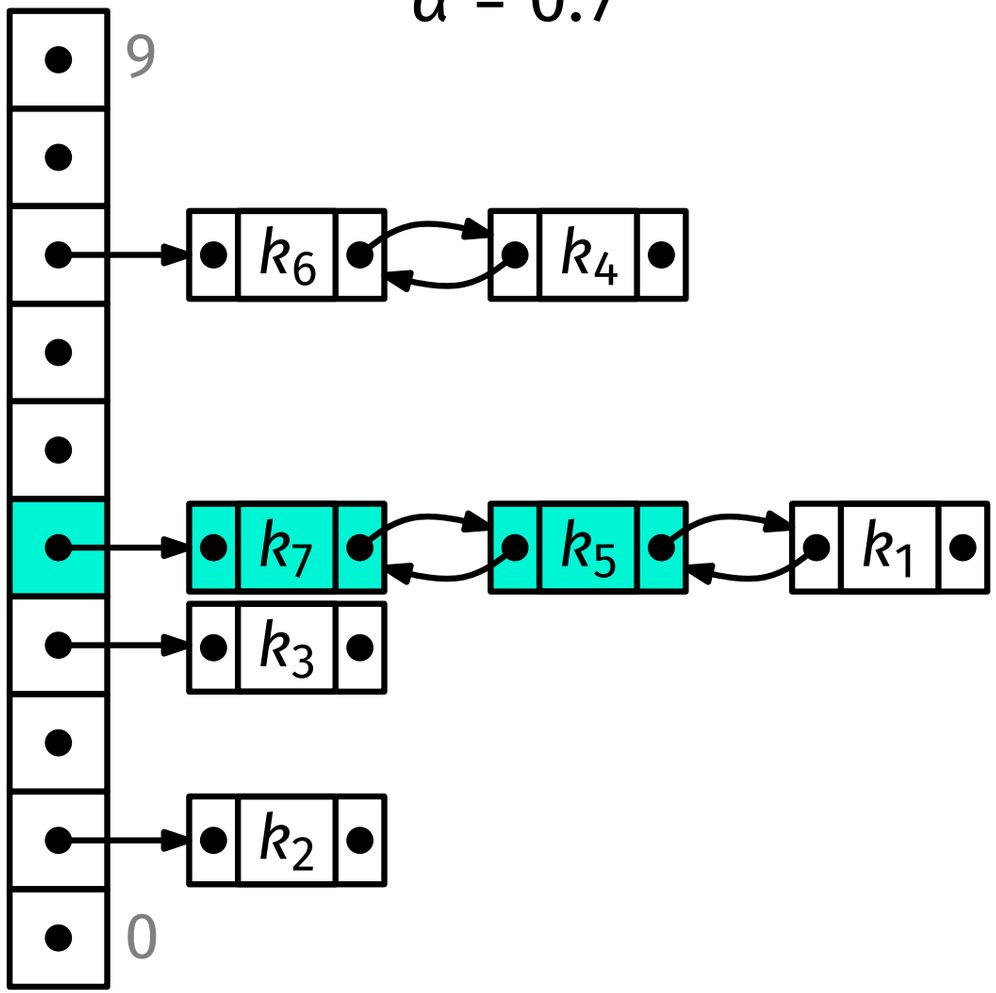
find(k_5) with $h(k_5) = 4$



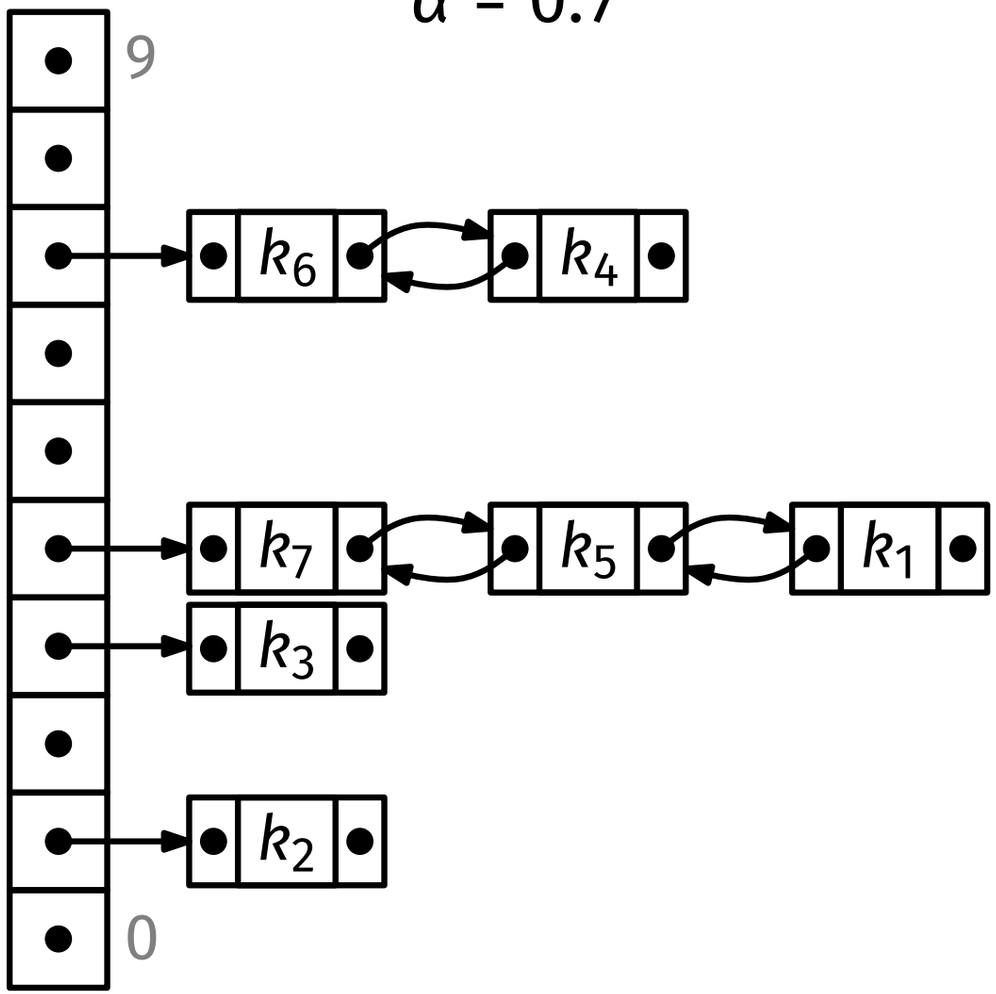
Chaining Hash Maps – Operations

$\alpha = 0.7$

find(k_5) with $h(k_5) = 4$



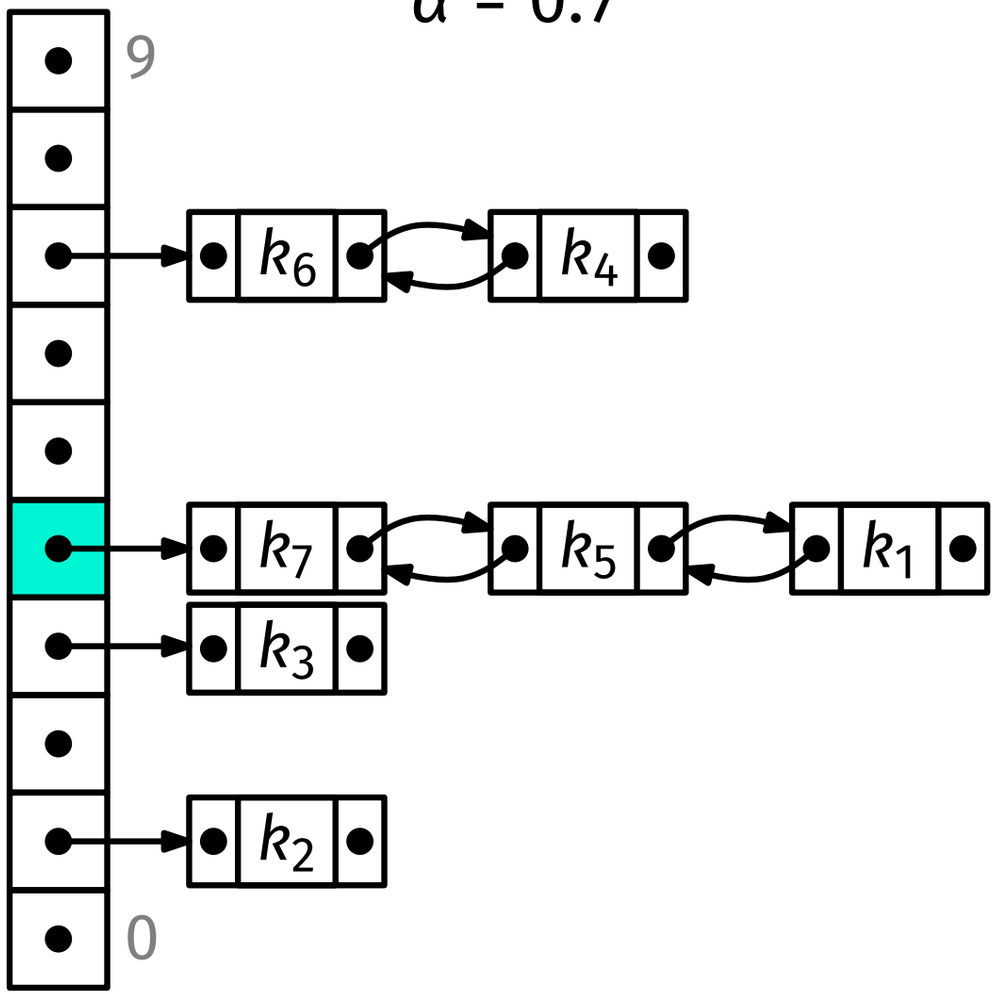
Chaining Hash Maps – Operations



find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

Chaining Hash Maps – Operations



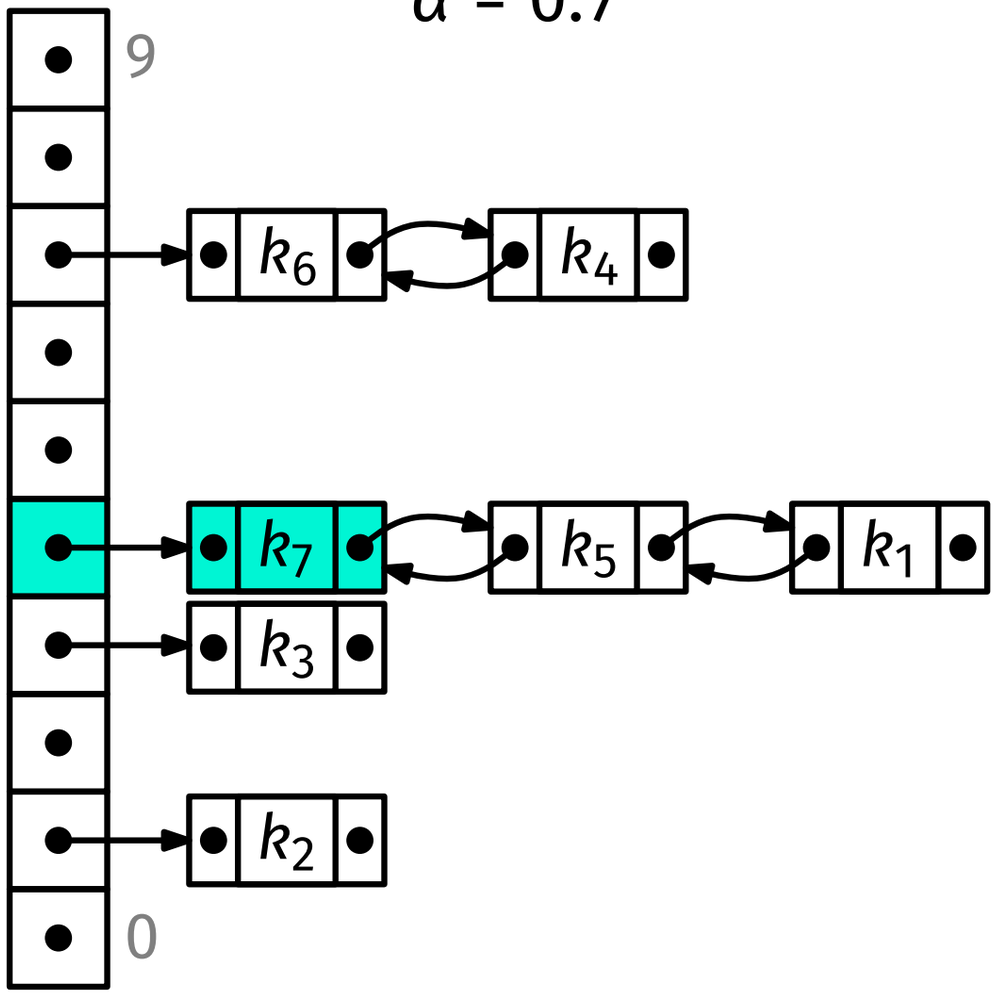
$\alpha = 0.7$

find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

Chaining Hash Maps – Operations

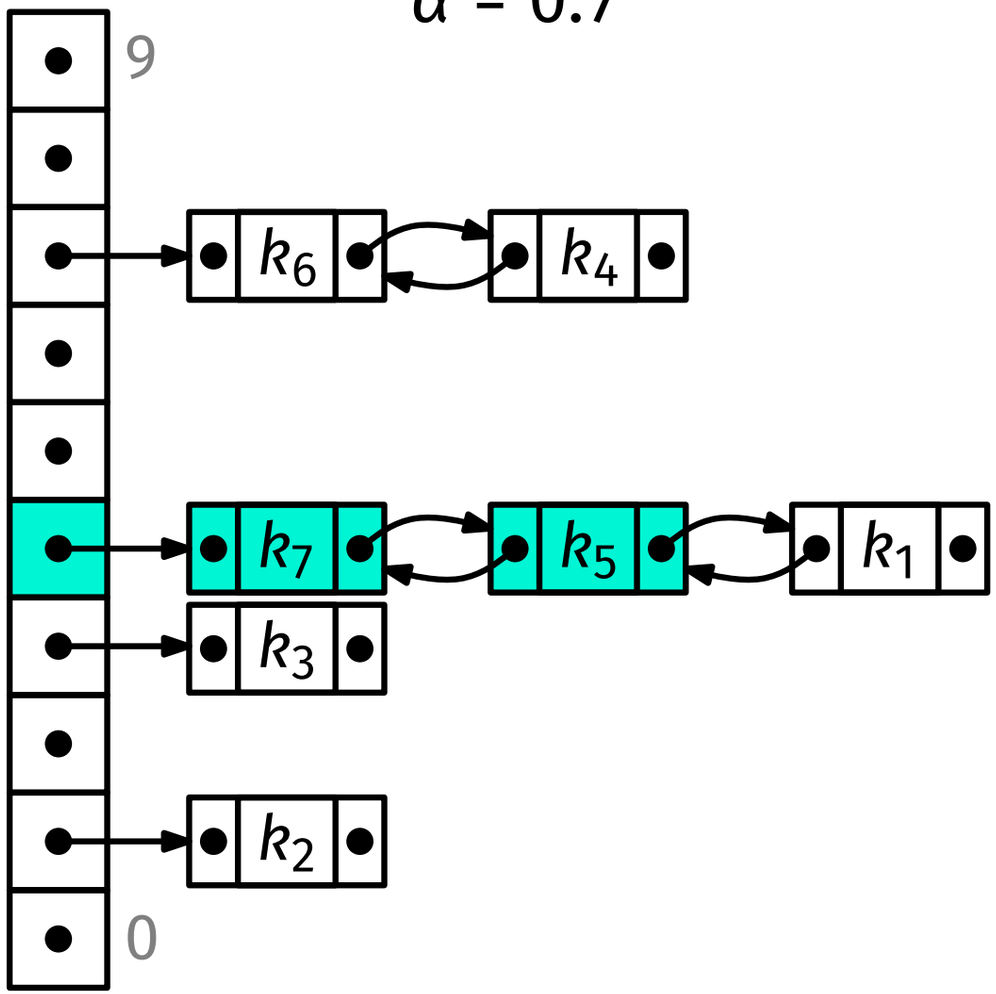
$\alpha = 0.7$



find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

Chaining Hash Maps – Operations

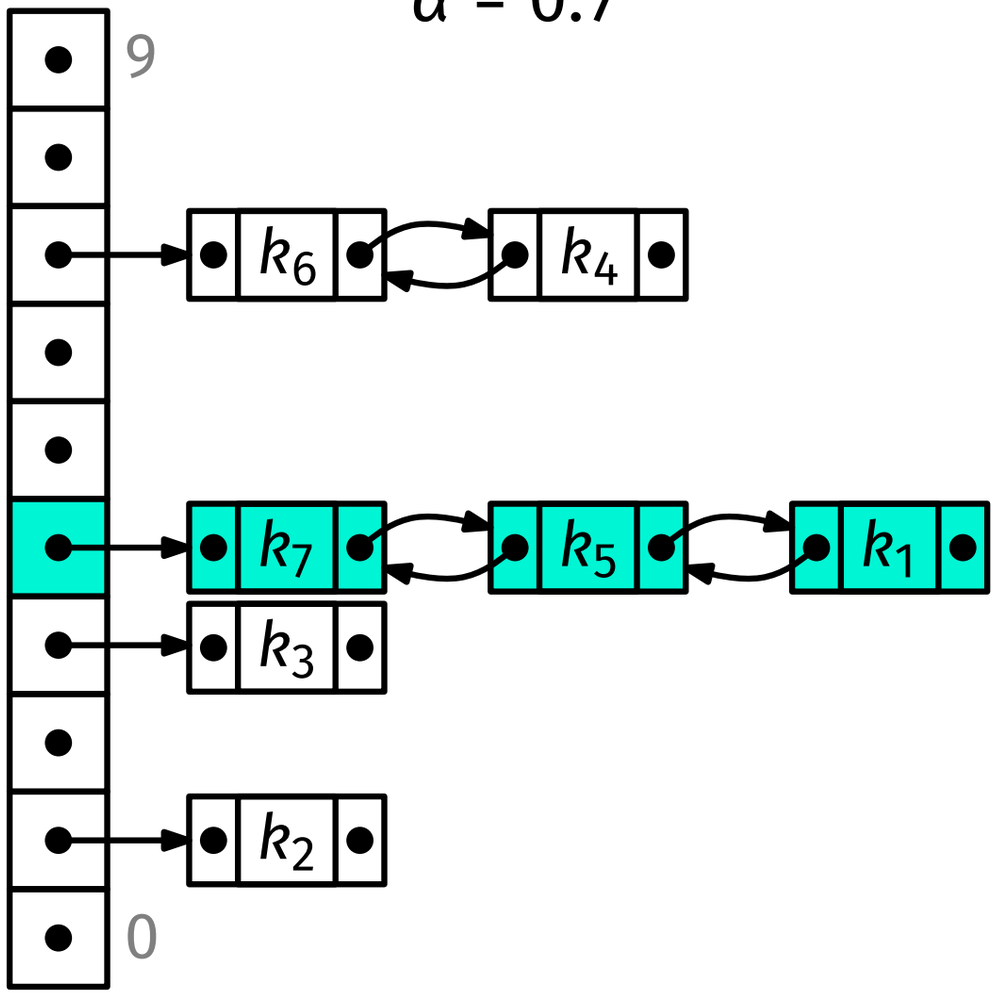


find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

Chaining Hash Maps – Operations

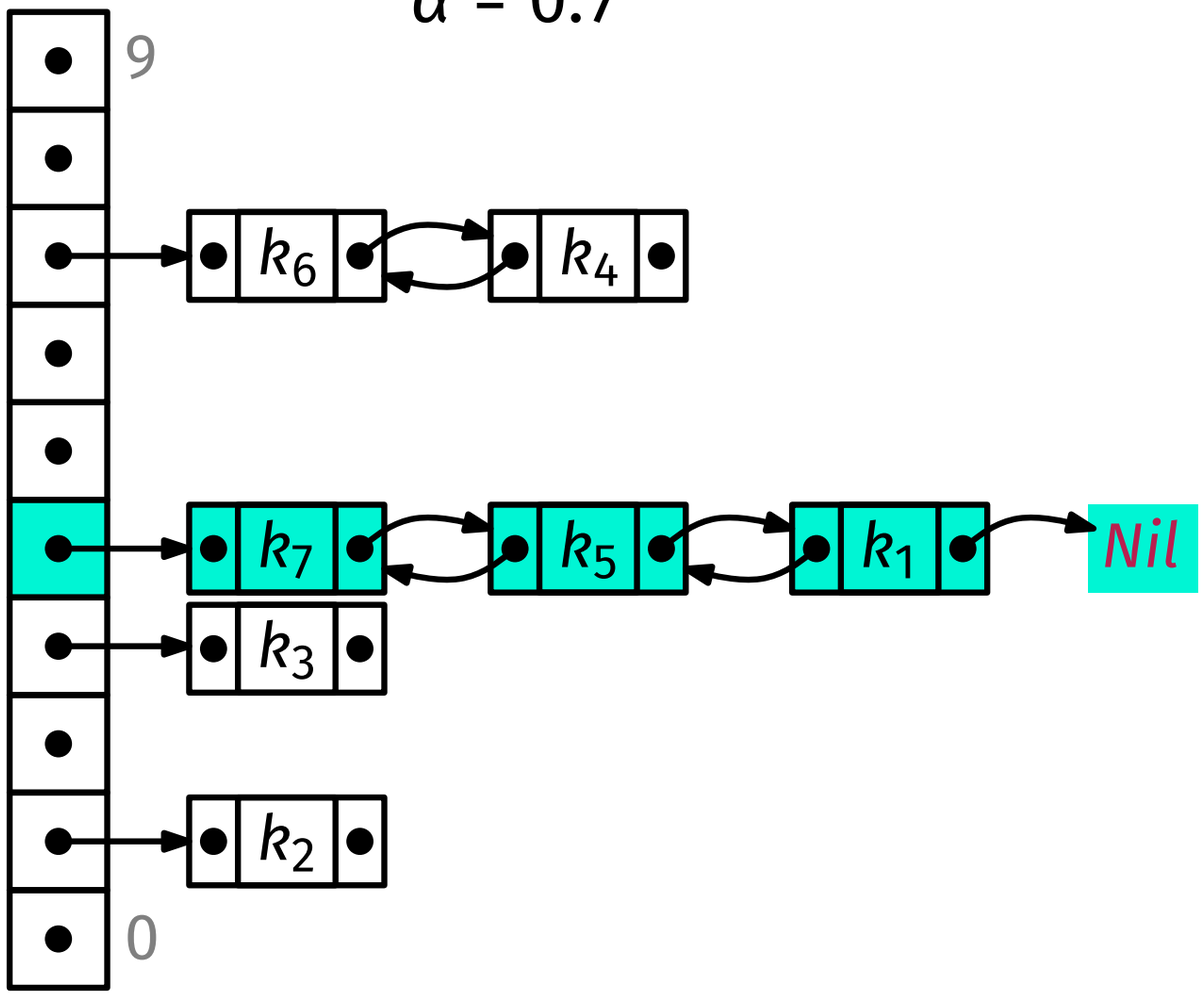
$\alpha = 0.7$



find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

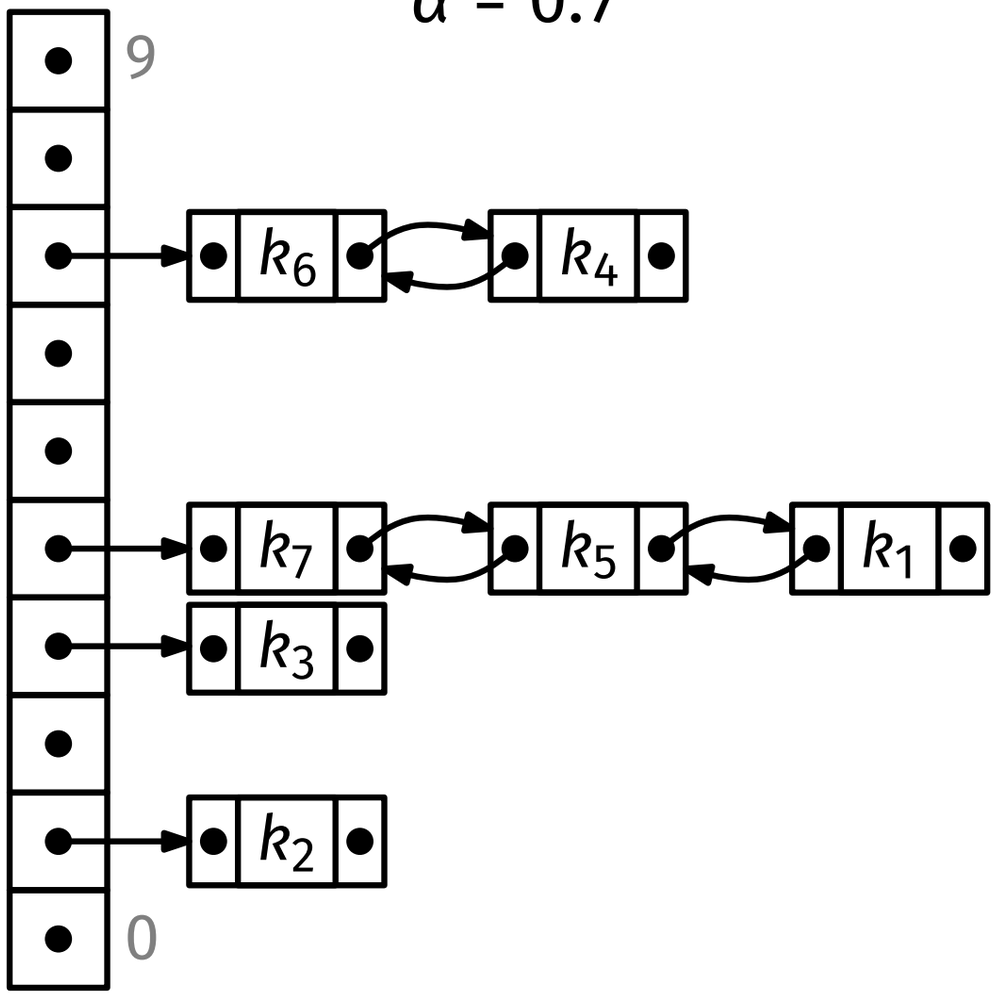
Chaining Hash Maps – Operations



find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

Chaining Hash Maps – Operations

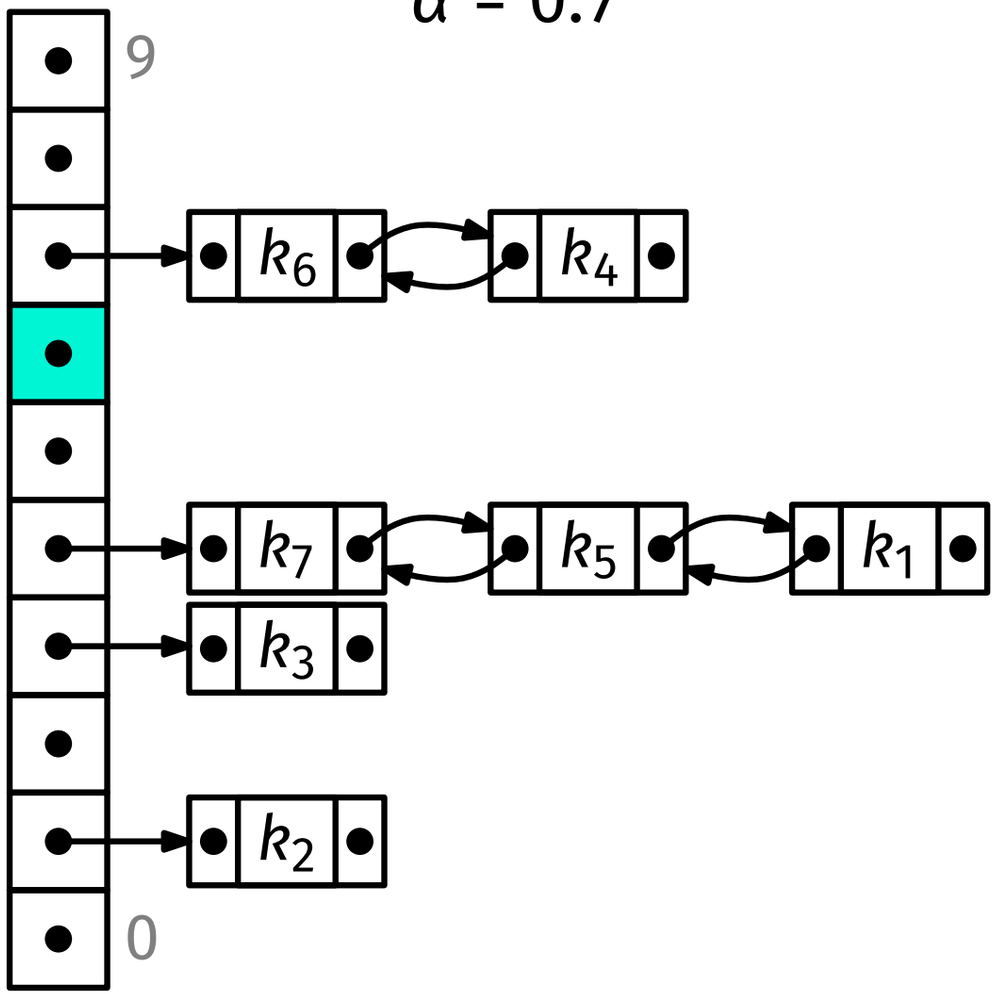


find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

Chaining Hash Maps – Operations

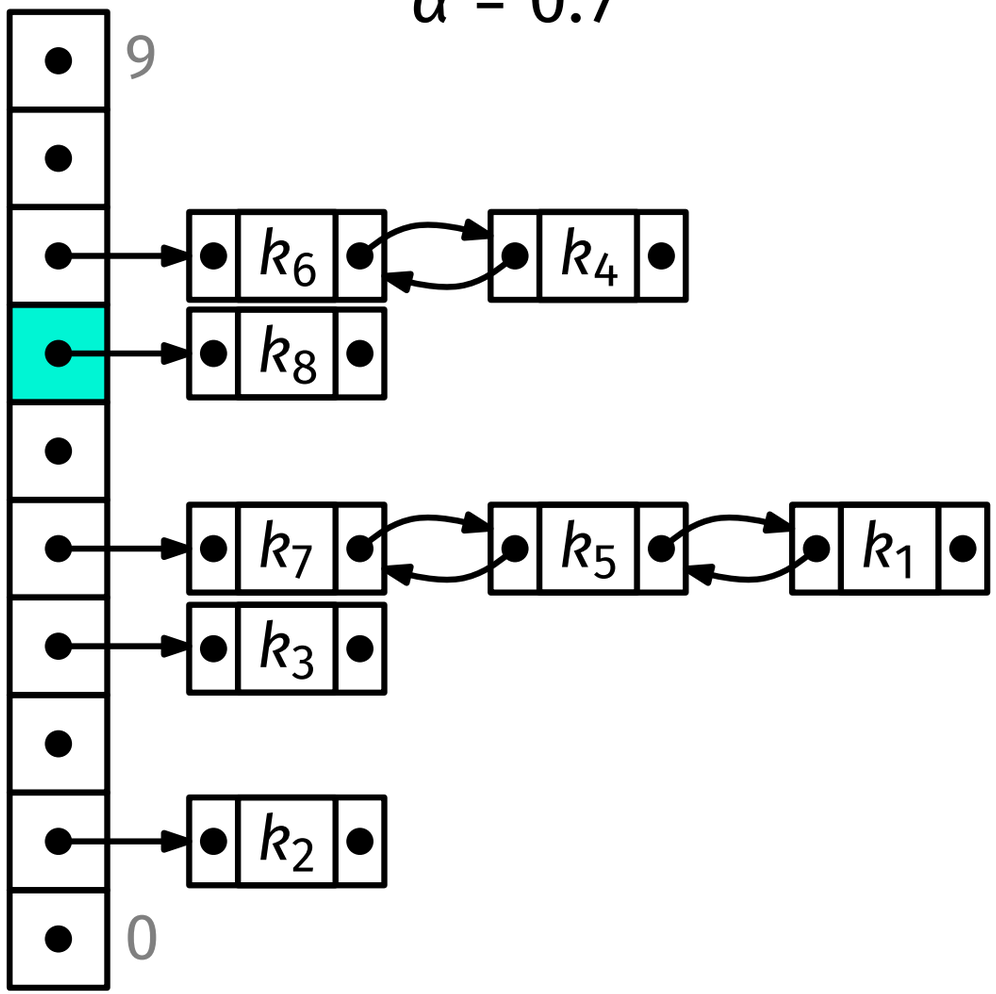


find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

Chaining Hash Maps – Operations

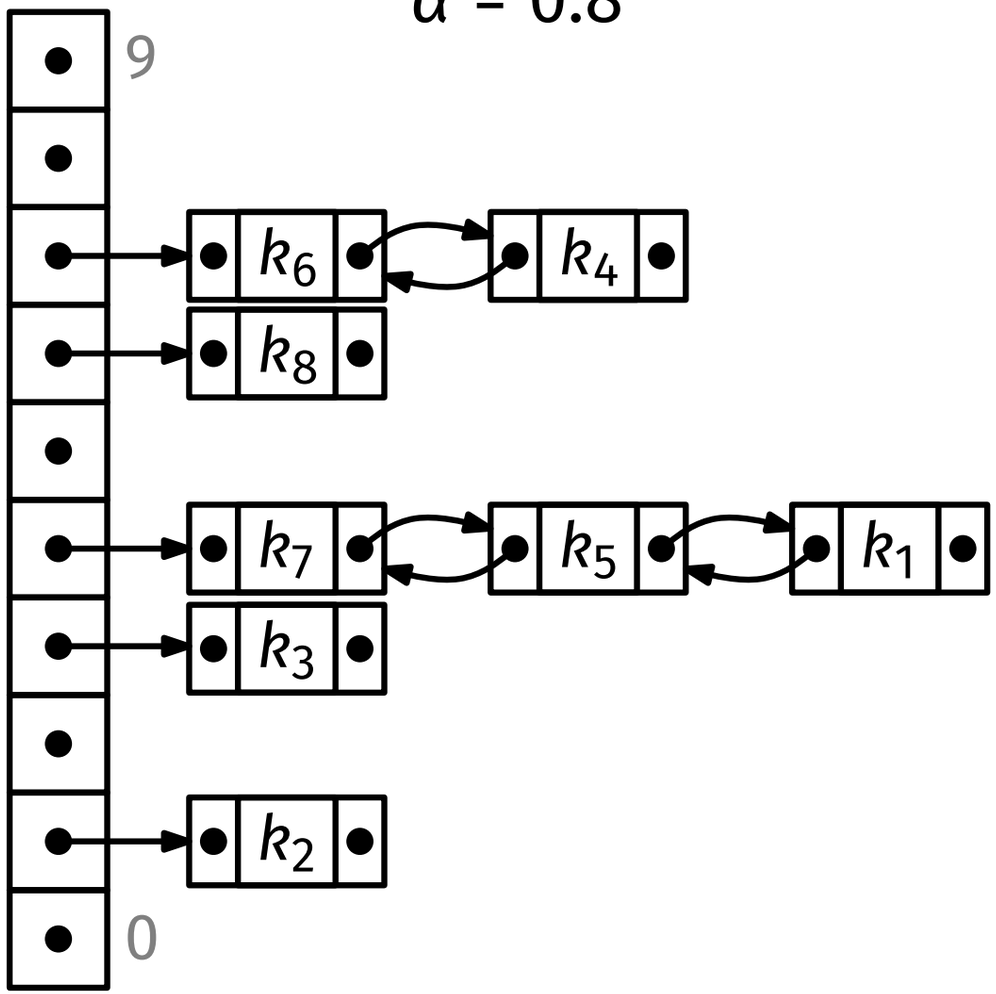


find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

Chaining Hash Maps – Operations

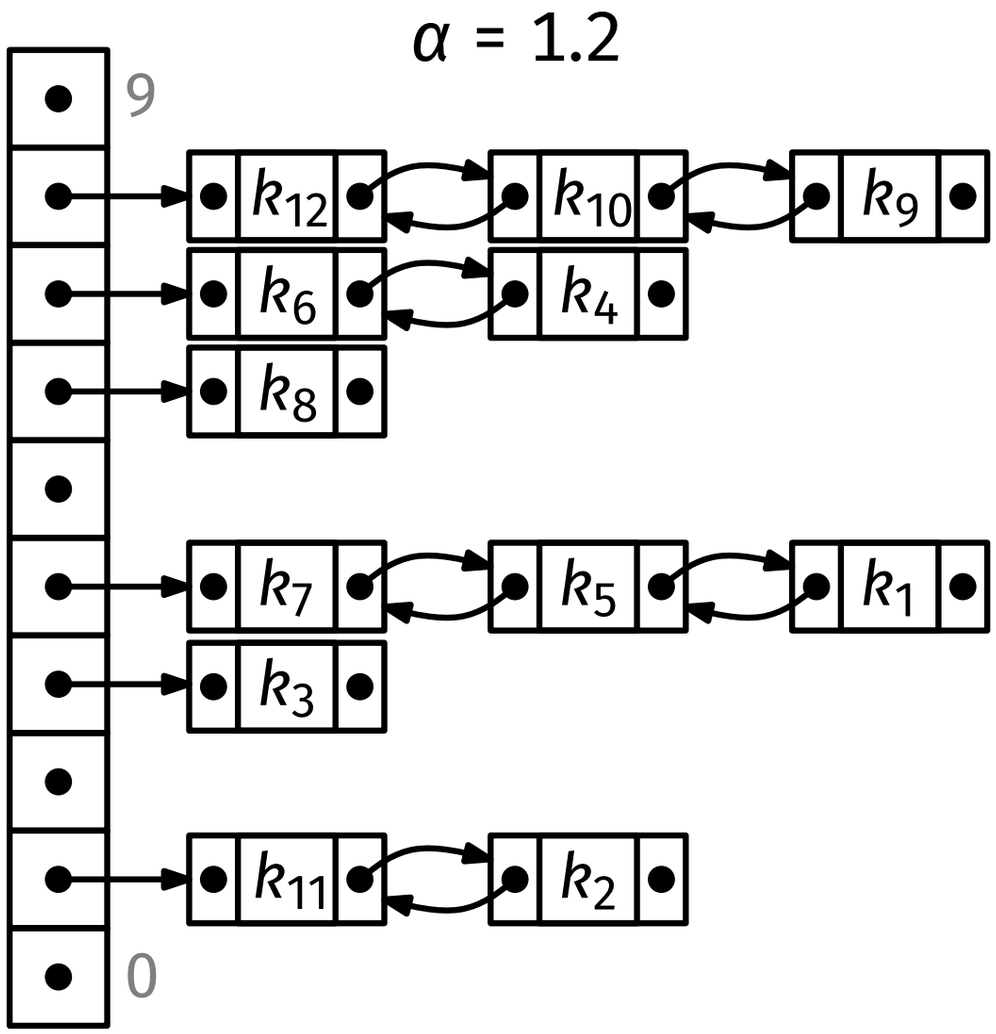


find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

Chaining Hash Maps – Operations



find(k_5) with $h(k_5) = 4$

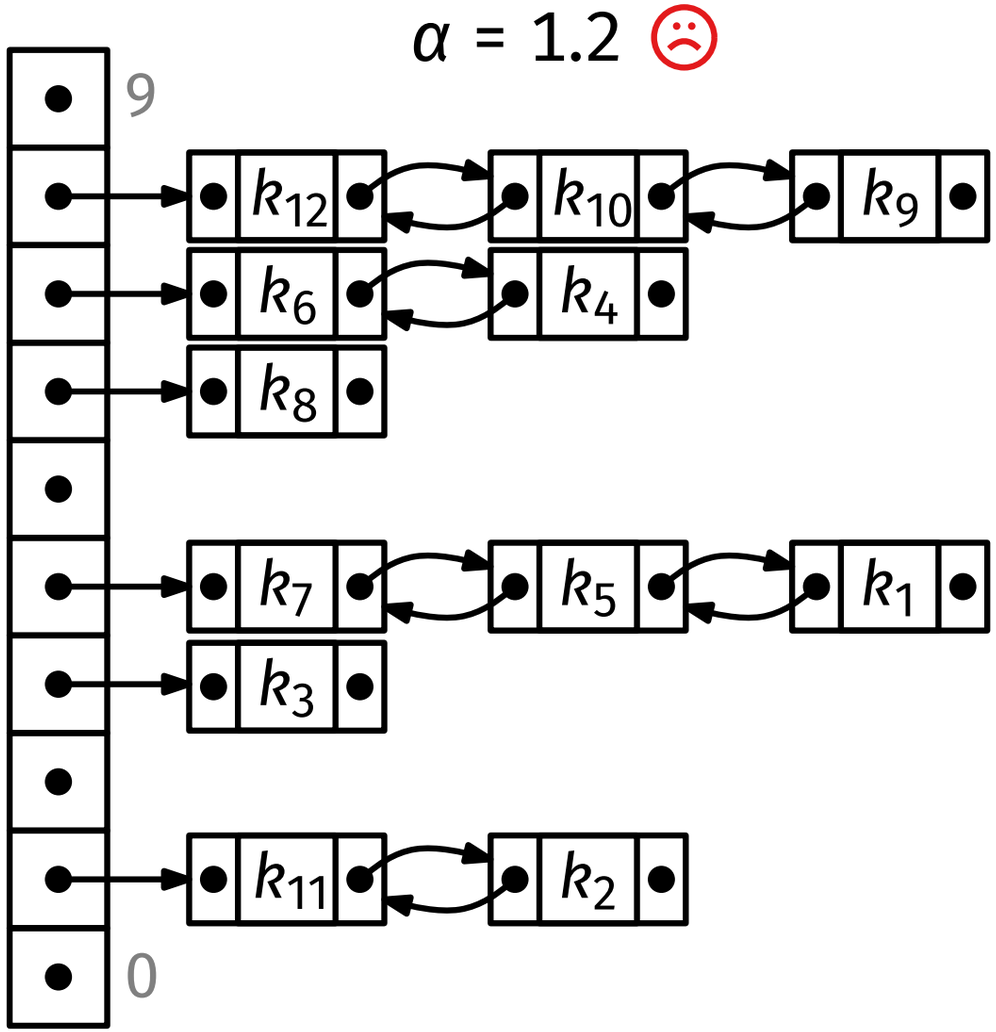
find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

...

insert(k_{12}) with $h(k_{12}) = 8$

Chaining Hash Maps – Operations



find(k_5) with $h(k_5) = 4$

find(k_x) with $h(k_x) = 4$

insert(k_8) with $h(k_8) = 6$

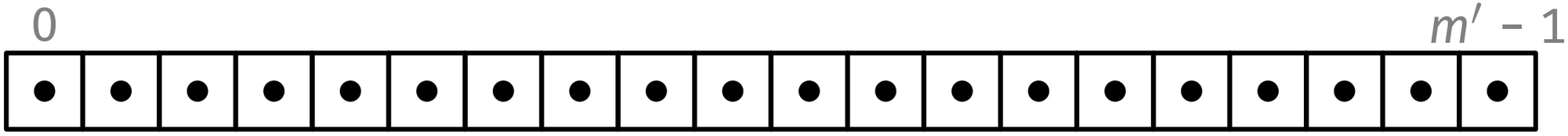
...

insert(k_{12}) with $h(k_{12}) = 8$

Chaining Hash Maps – Operations

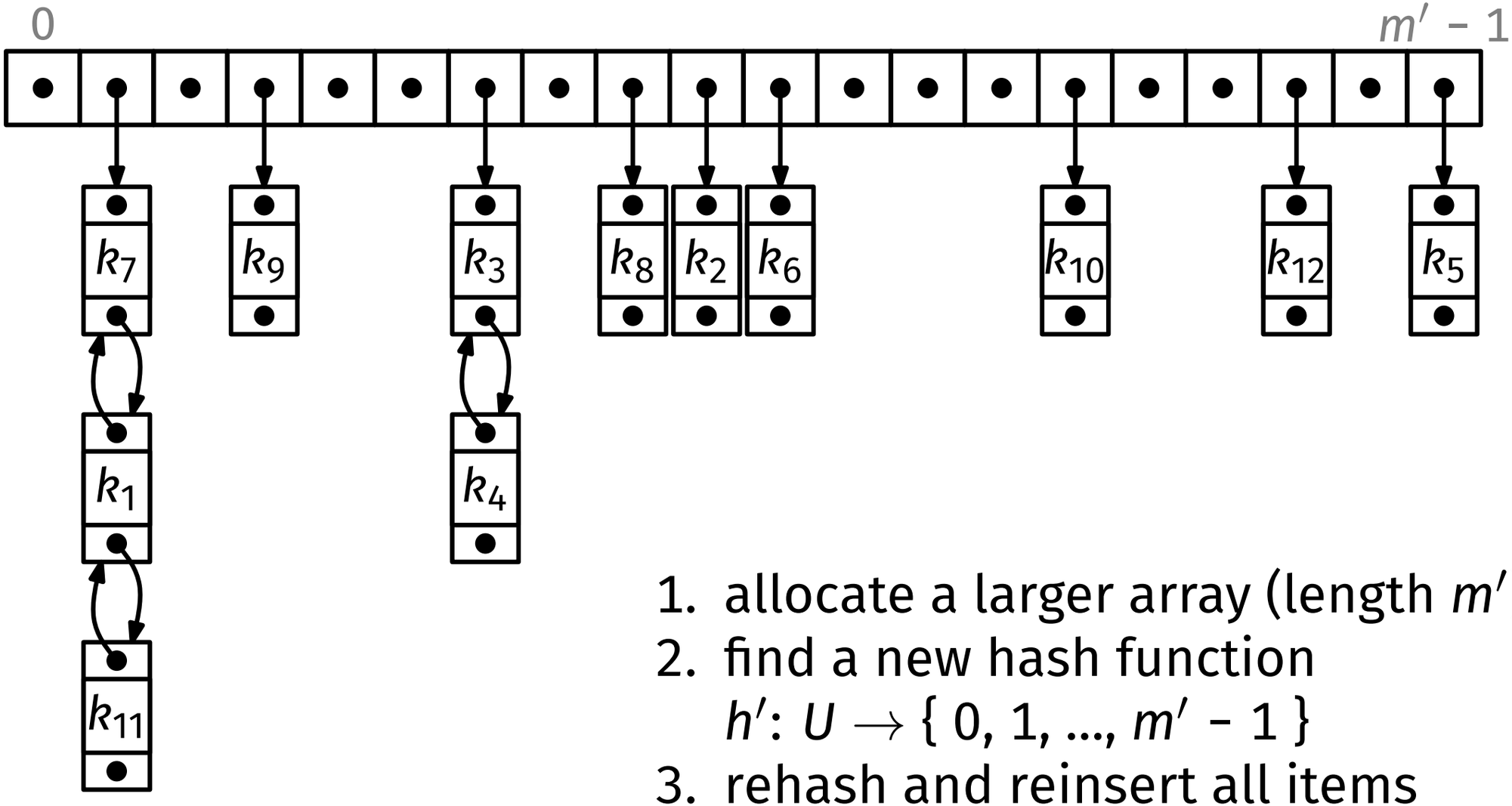
1. allocate a larger array (length m')

Chaining Hash Maps – Operations



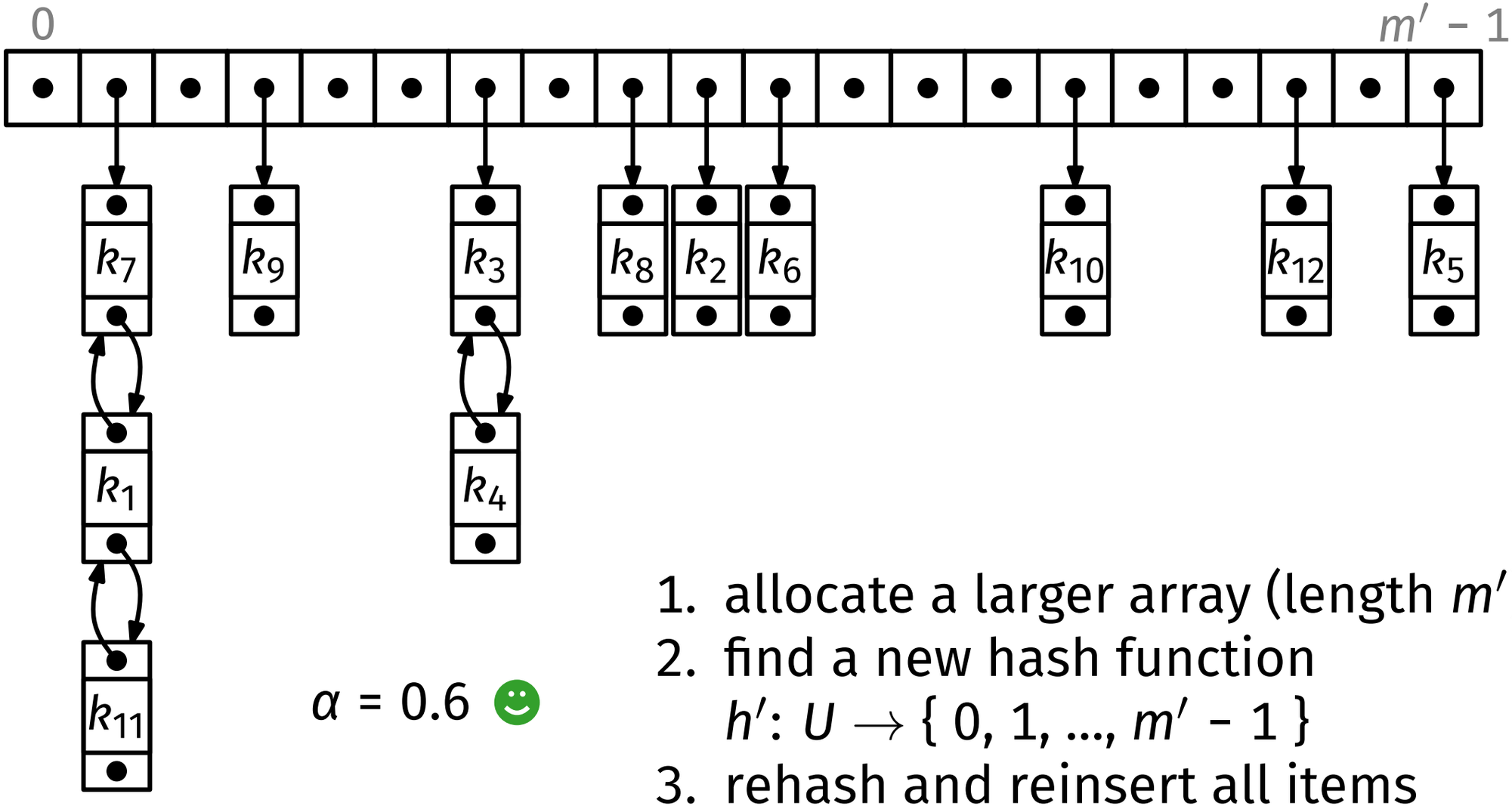
1. allocate a larger array (length m')
2. find a new hash function
   h': $U \rightarrow \{ 0, 1, ..., m' - 1 \}$

Chaining Hash Maps – Operations



1. allocate a larger array (length m')
2. find a new hash function
   h': $U \rightarrow \{ 0, 1, ..., m' - 1 \}$
3. rehash and reinsert all items

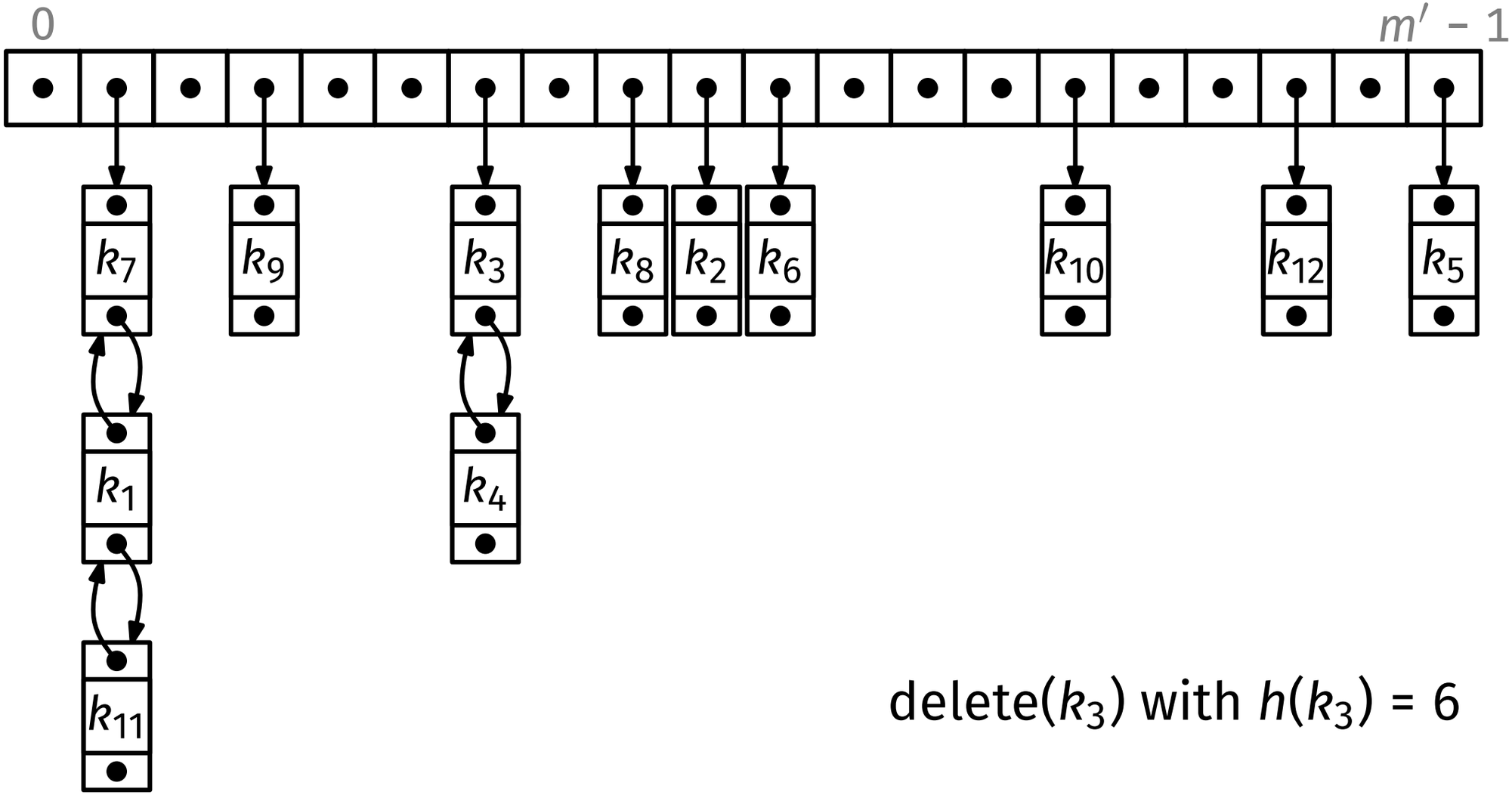
Chaining Hash Maps – Operations



$\alpha = 0.6$

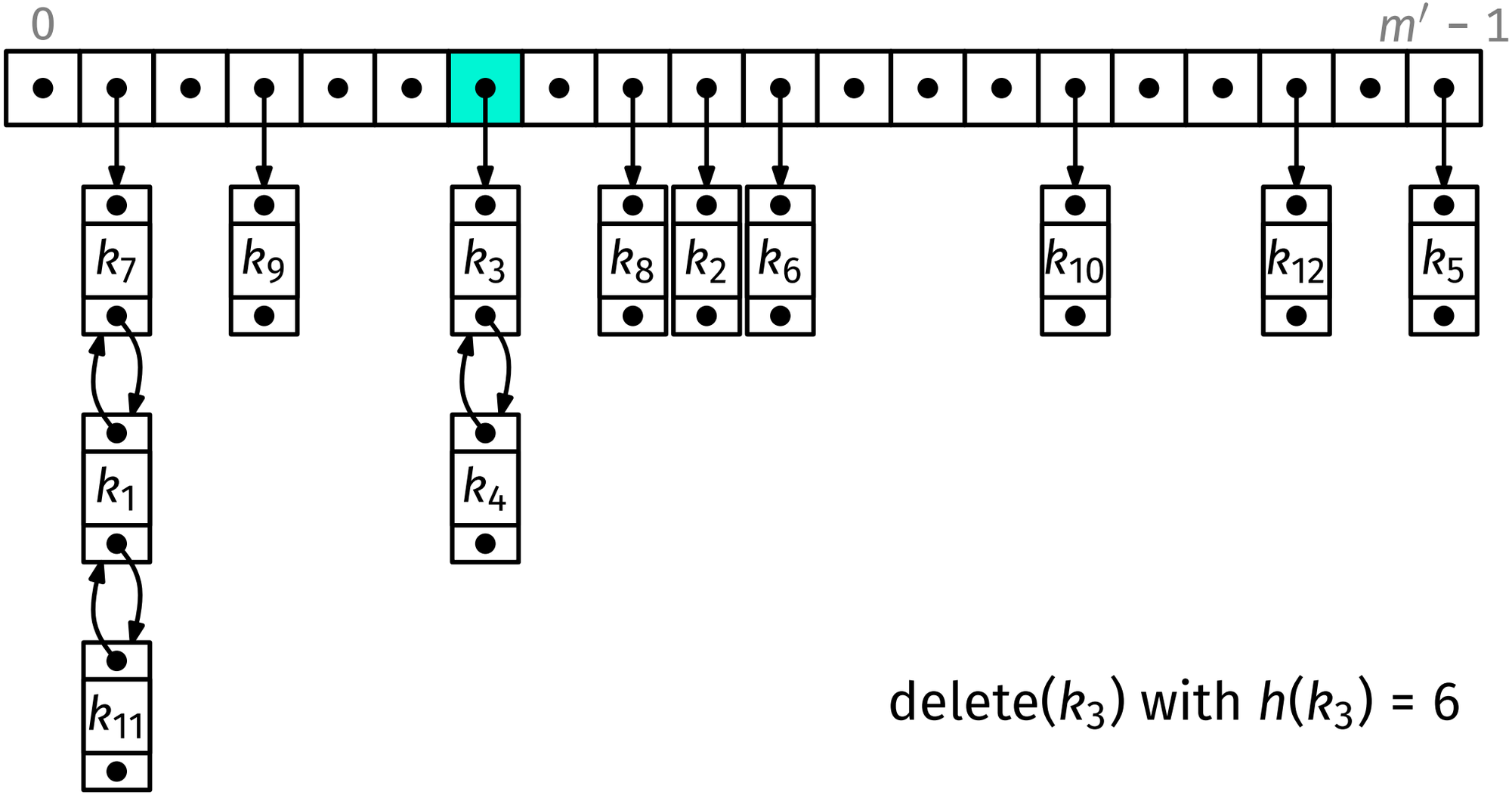
1. allocate a larger array (length m')
2. find a new hash function
   h': $U \rightarrow \{ 0, 1, \ldots, m' - 1 \}$
3. rehash and reinsert all items

Chaining Hash Maps – Operations



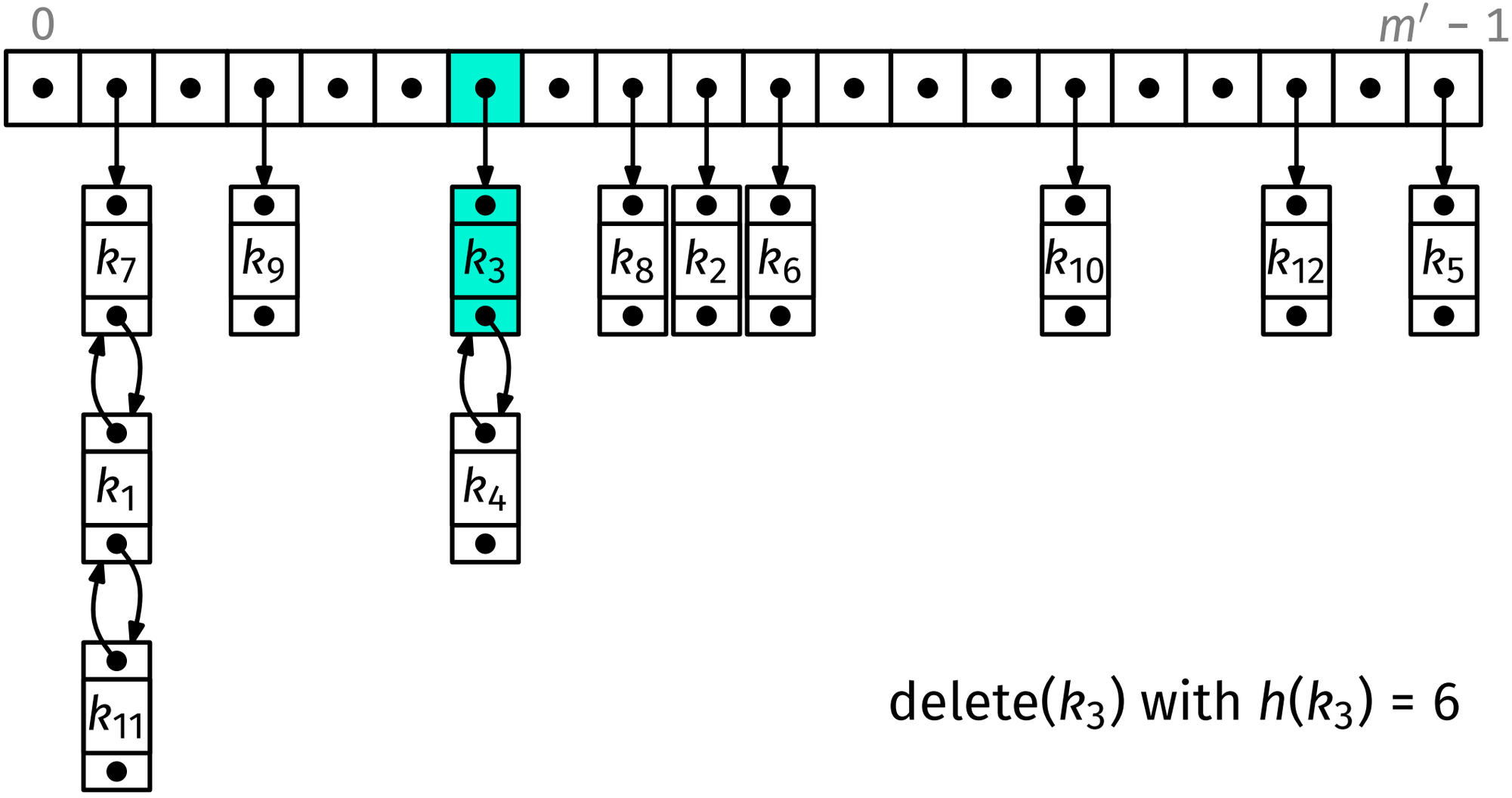
delete(k_3) with $h(k_3) = 6$

Chaining Hash Maps – Operations



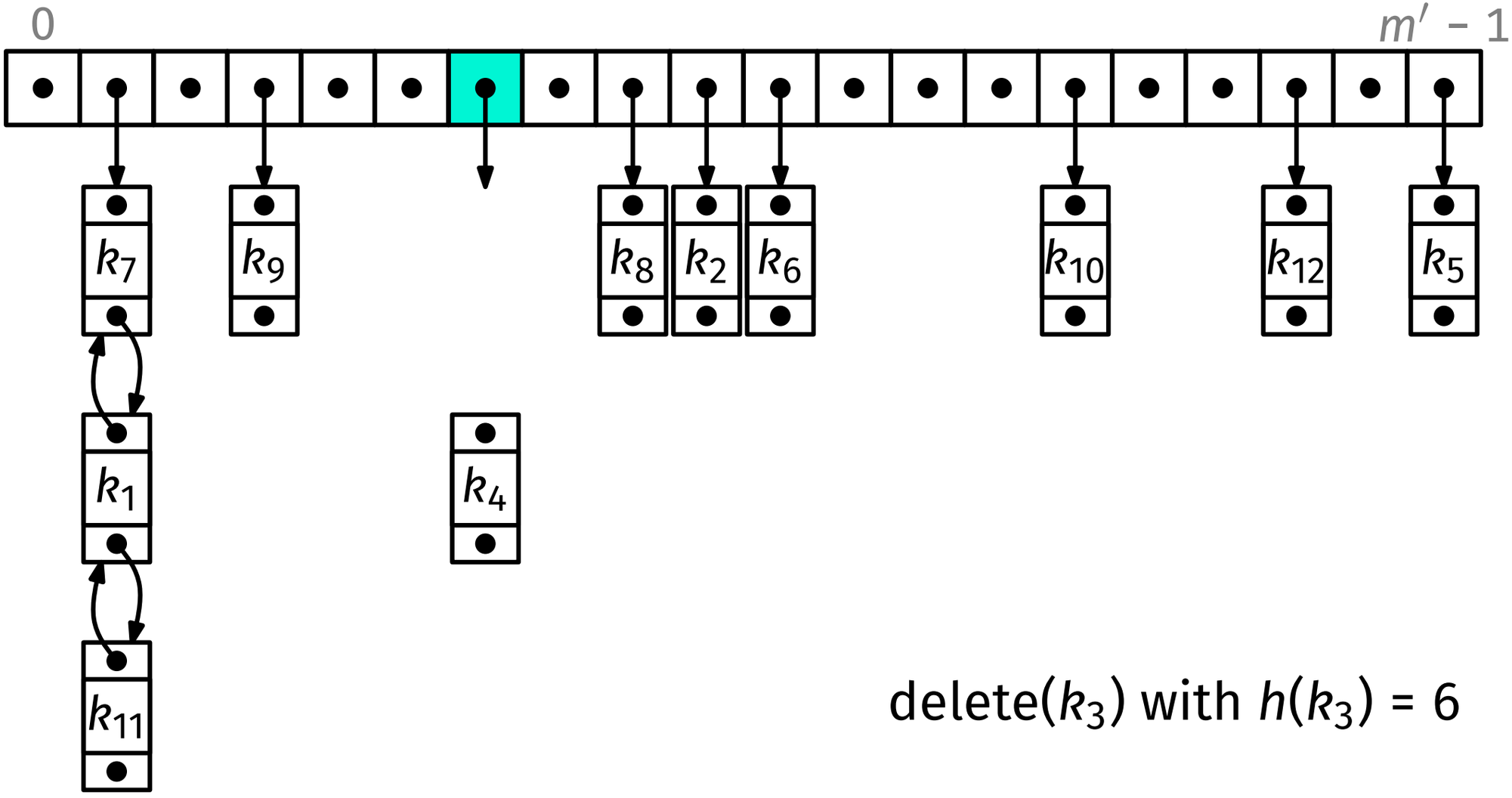
delete(k_3) with $h(k_3) = 6$

Chaining Hash Maps – Operations



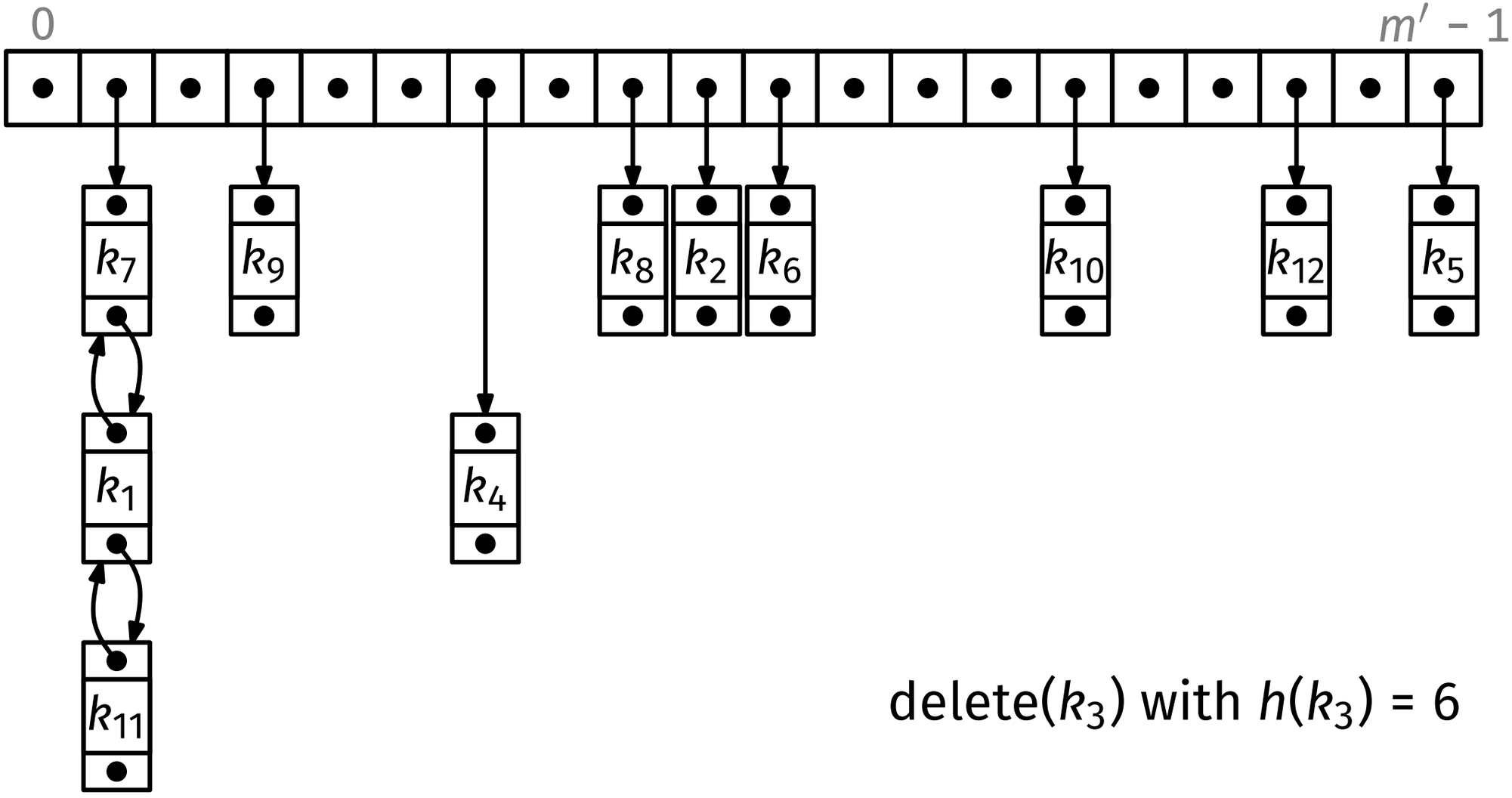
delete(k_3) with $h(k_3) = 6$

Chaining Hash Maps – Operations



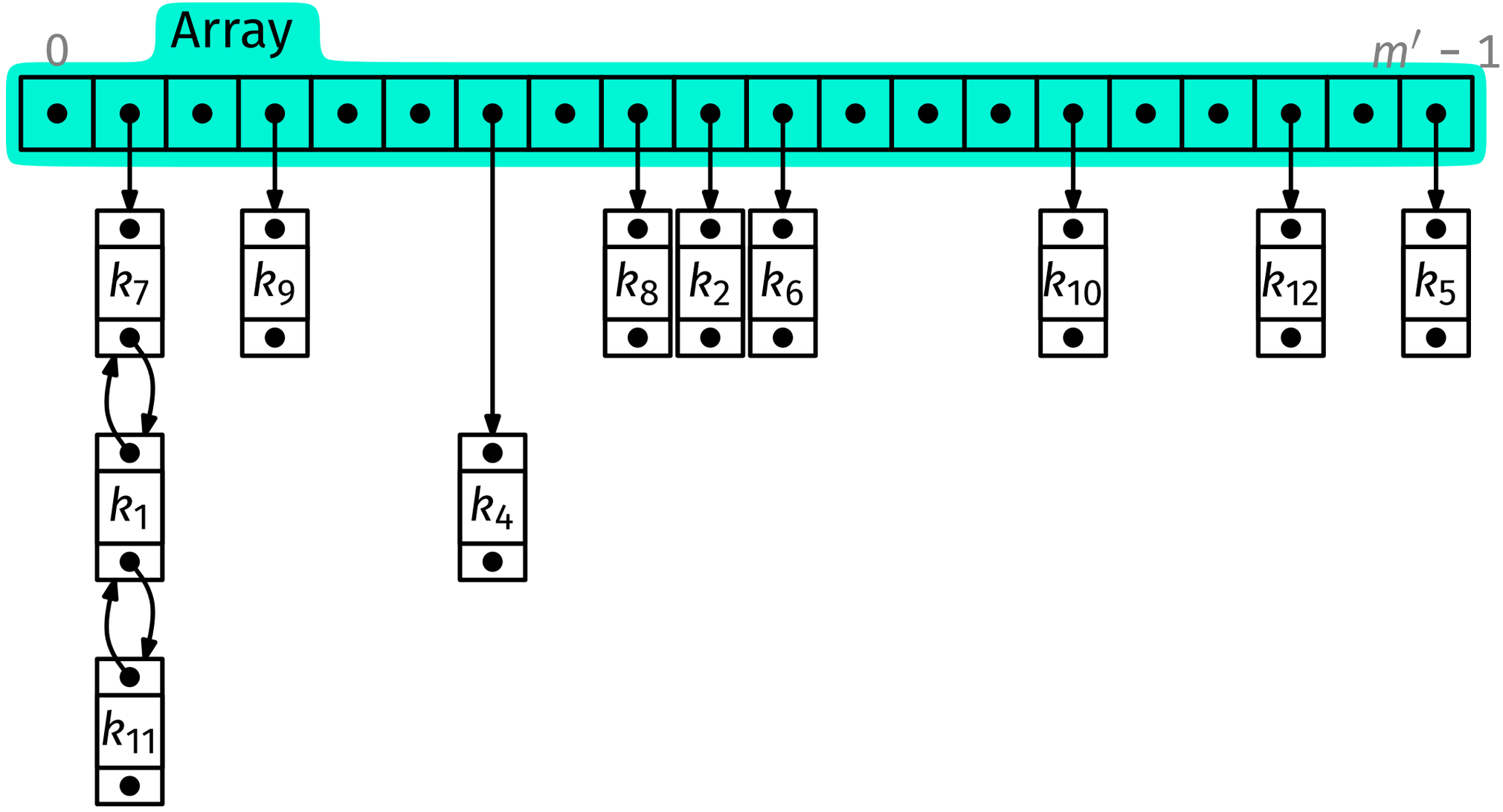
delete(k_3) with $h(k_3) = 6$

Chaining Hash Maps – Operations

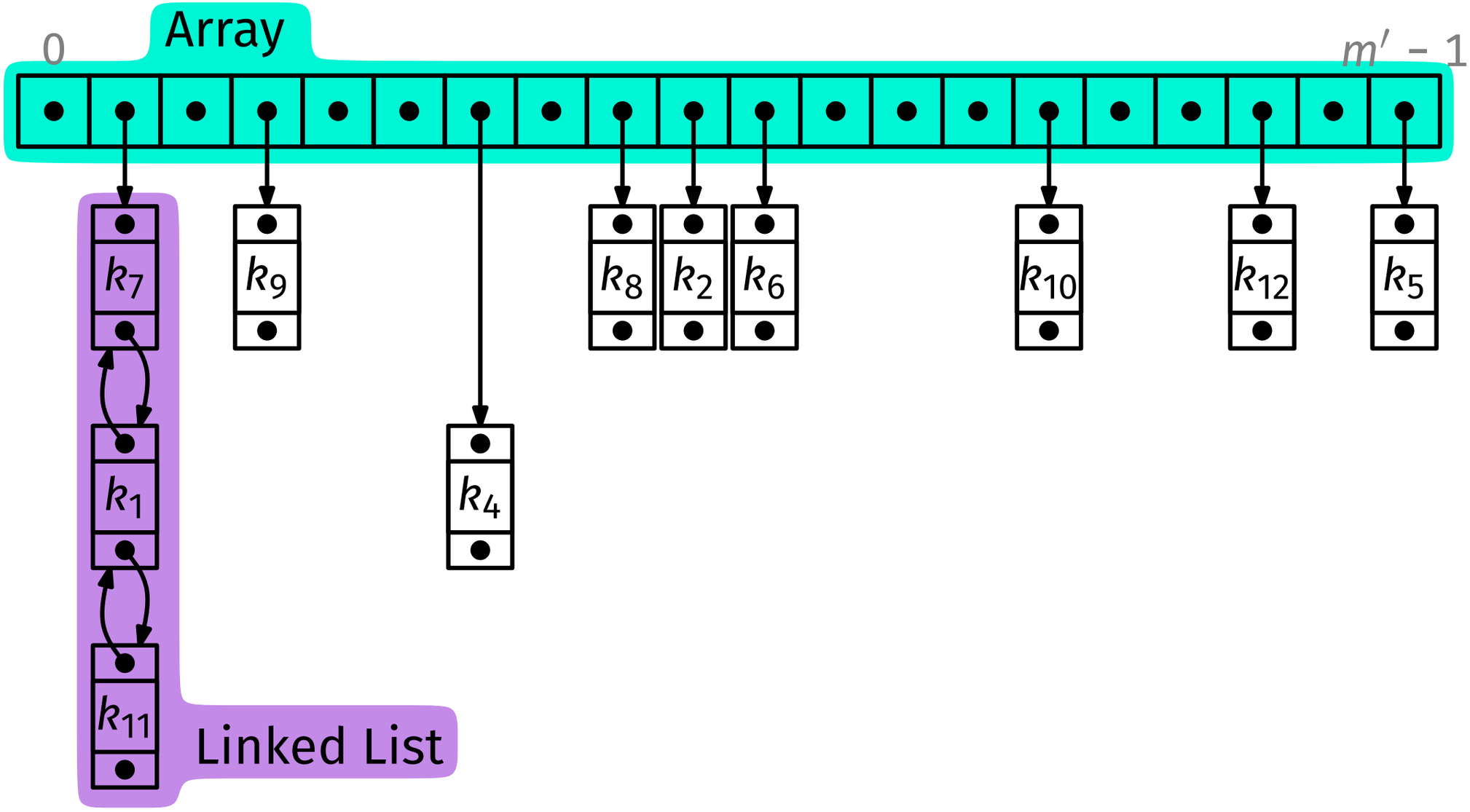


delete(k_3) with $h(k_3) = 6$

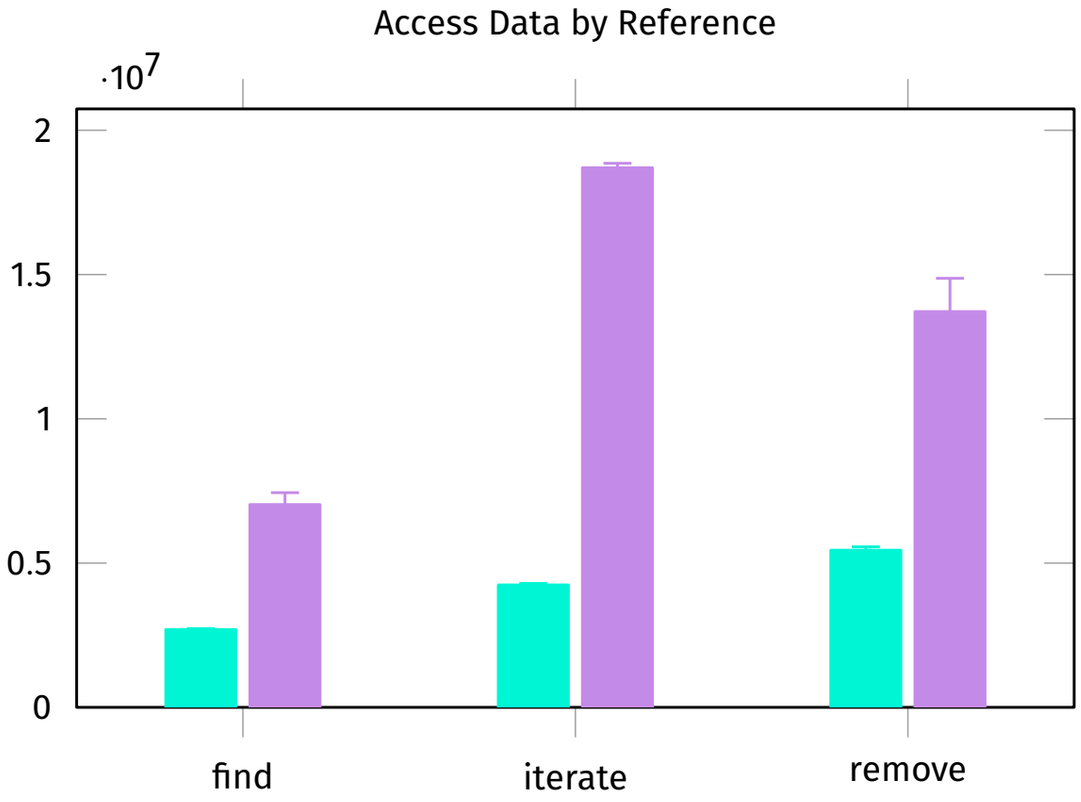
Chaining Hash Maps – Operations



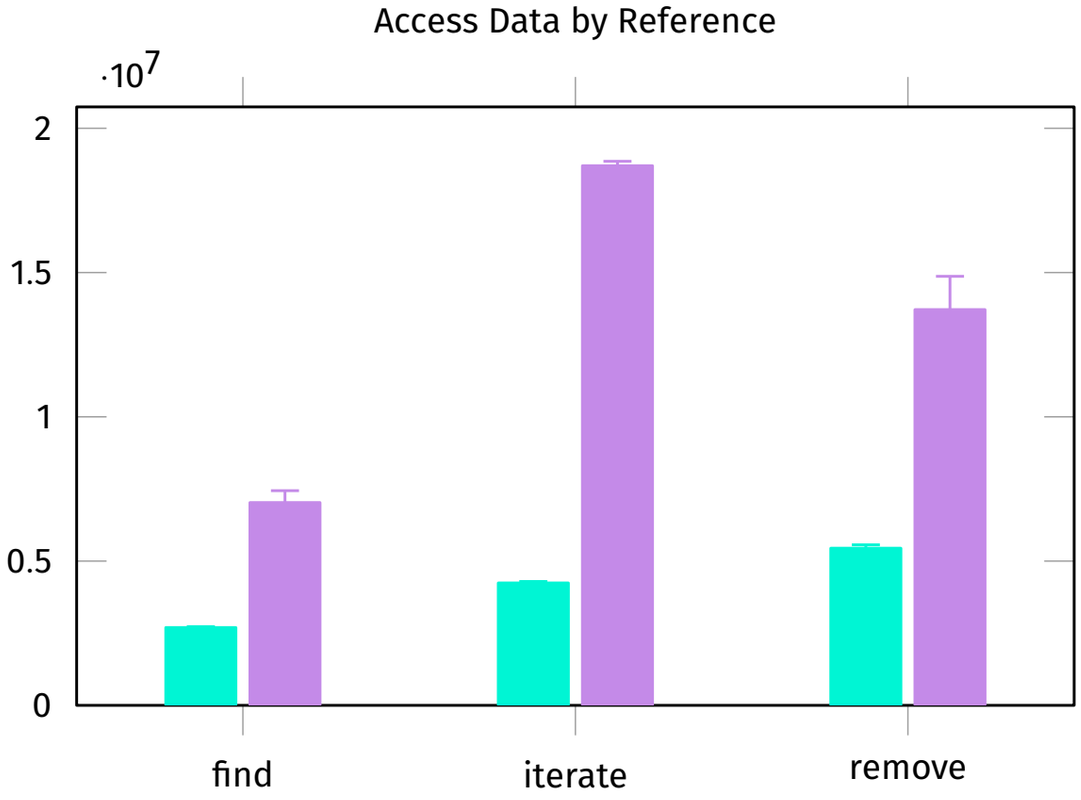
Chaining Hash Maps – Operations

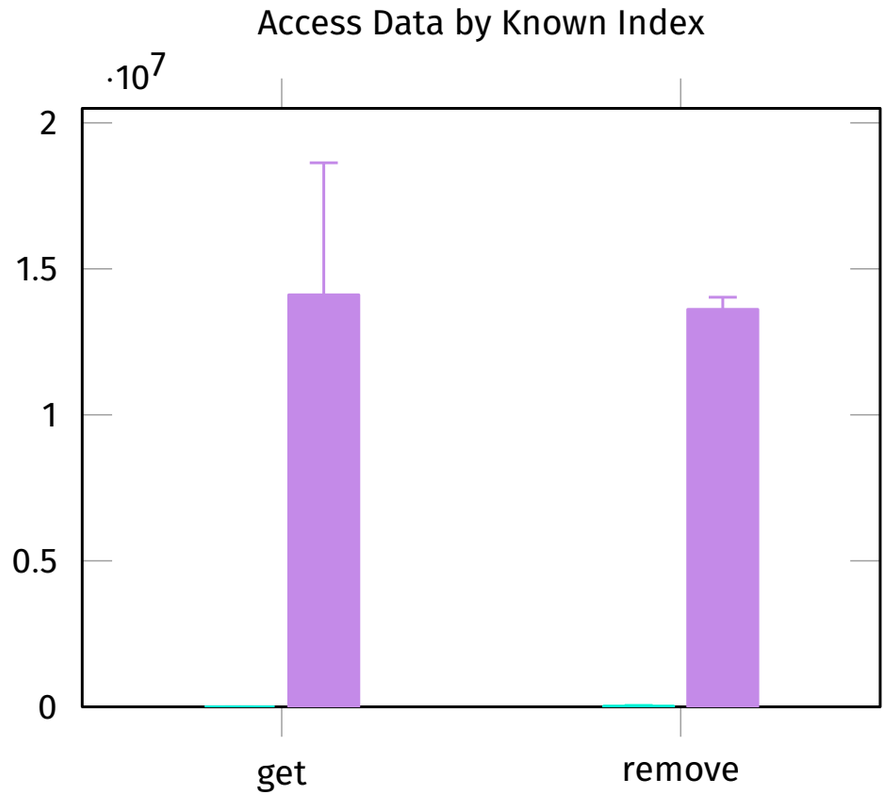


Linked List vs Array List

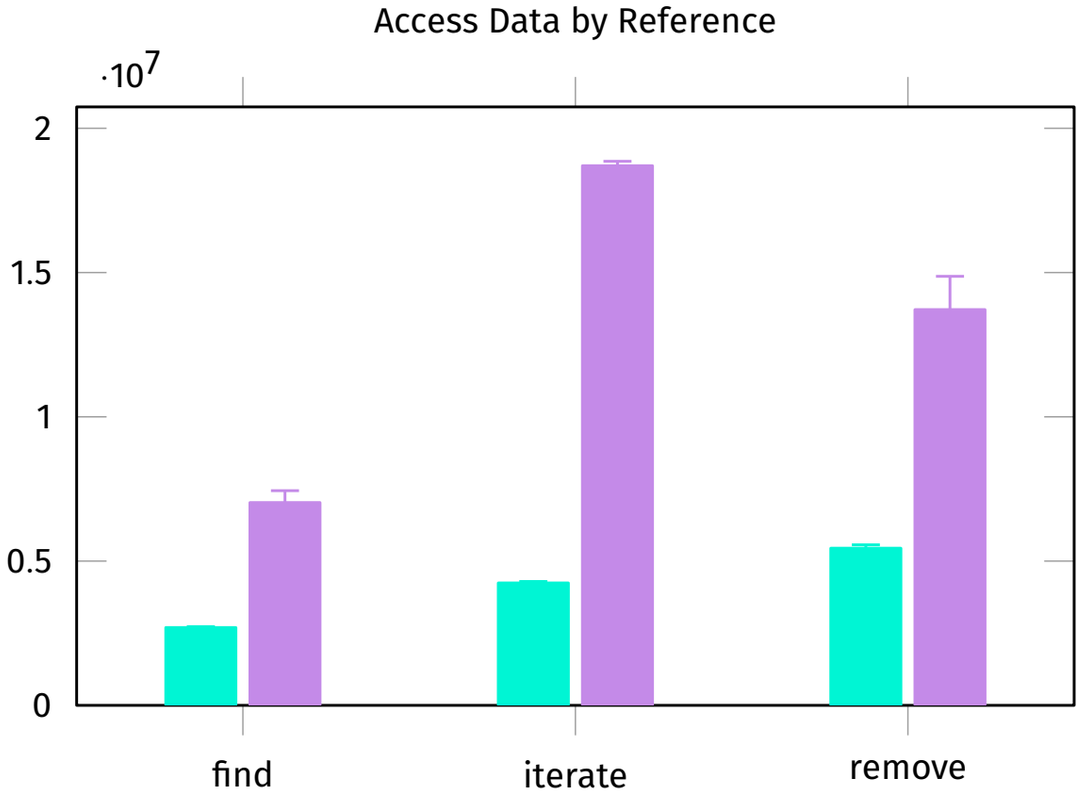


Linked List vs Array List

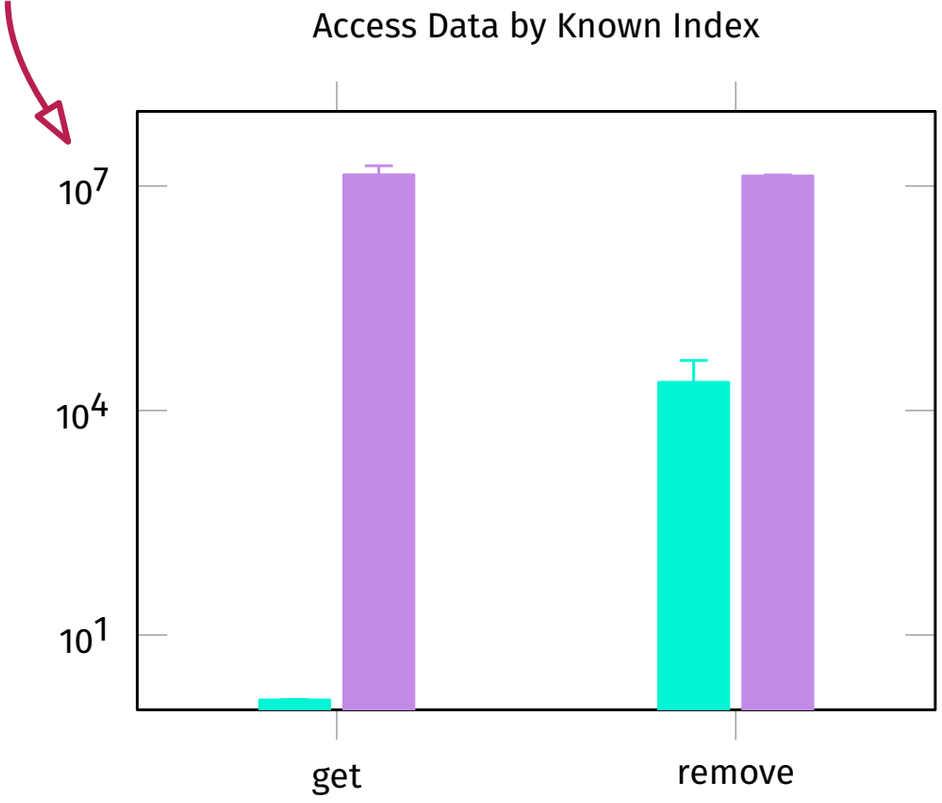




Linked List vs Array List



beware!



Linked List vs Array List

Every data structure is bad at something. Linked lists are just bad at most things.

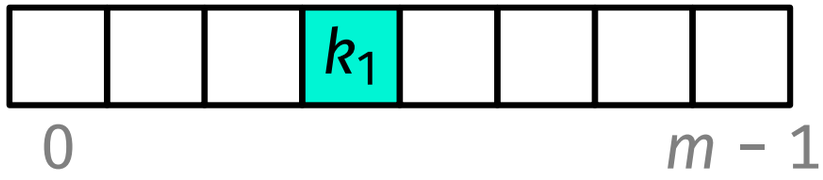
Probing Hash Table

Probing Hash Table

insert(k_1)

$h(k_1) = 3$

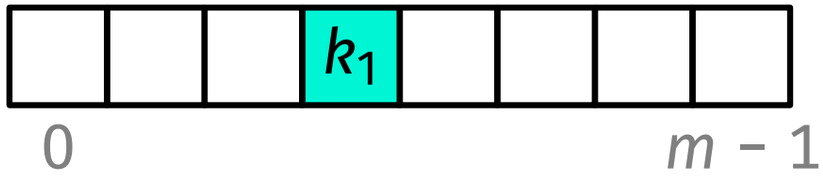
Probing Hash Table



insert(k_1)

$h(k_1) = 3$

Probing Hash Table



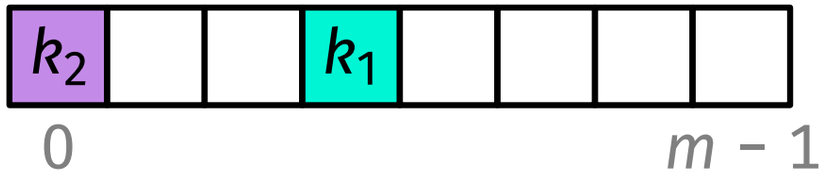
insert(k_1)

insert(k_2)

$h(k_1) = 3$

$h(k_2) = 0$

Probing Hash Table



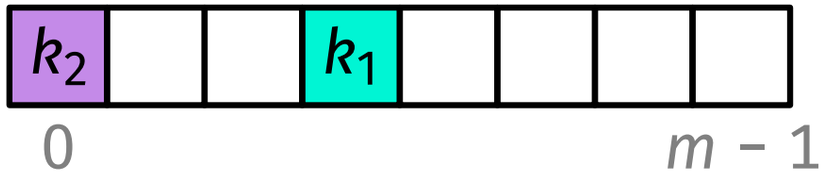
insert(k_1)

insert(k_2)

$h(k_1) = 3$

$h(k_2) = 0$

Probing Hash Table



insert(k_1)

insert(k_2)

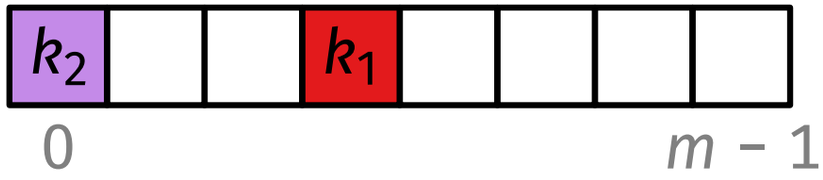
insert(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

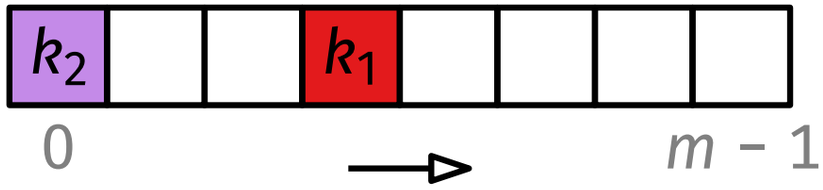
insert(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

Probing Hash Table



"linear probing"

insert(k_1)

insert(k_2)

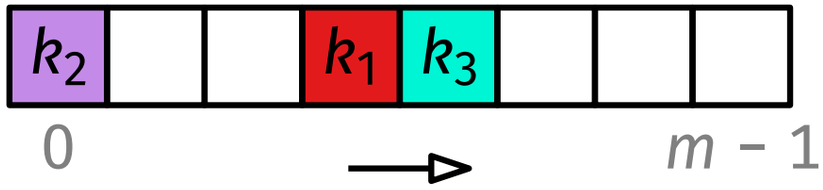
insert(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

Probing Hash Table



"linear probing"

insert(k_1)

insert(k_2)

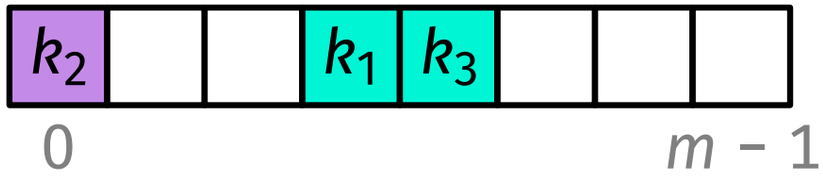
insert(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

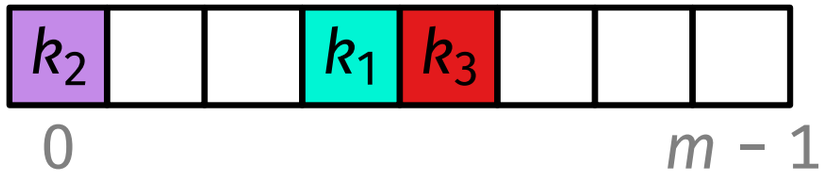
$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

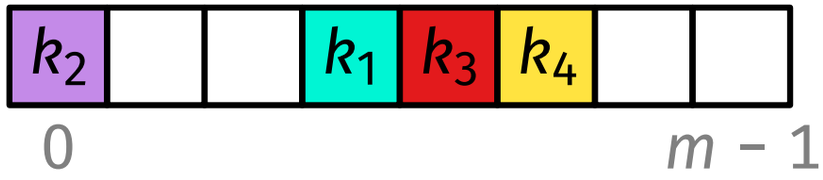
$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

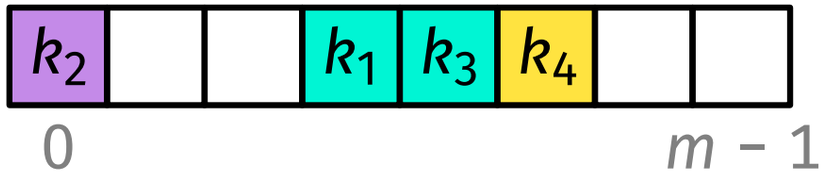
$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

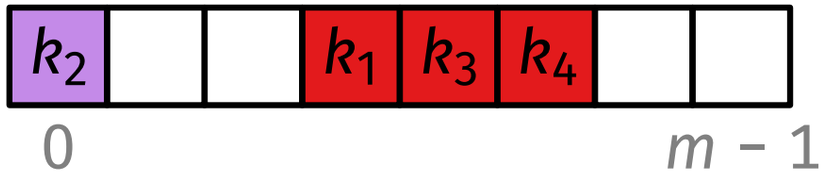
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

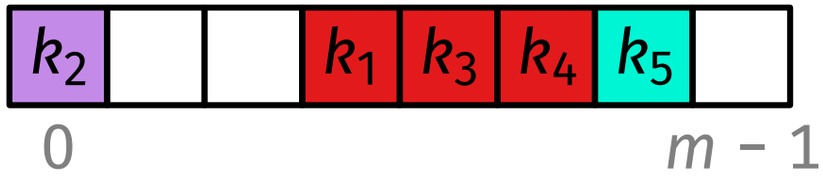
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

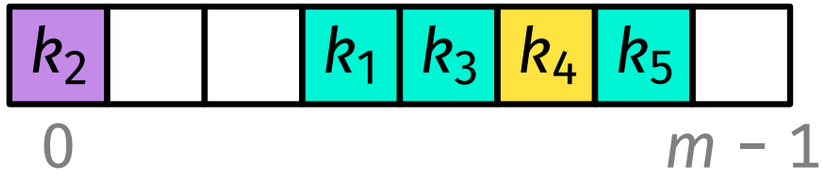
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

$h(k_1) = 3$

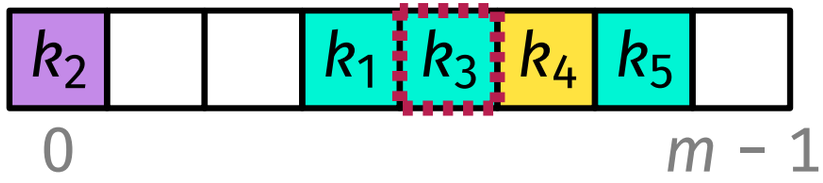
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

$h(k_1) = 3$

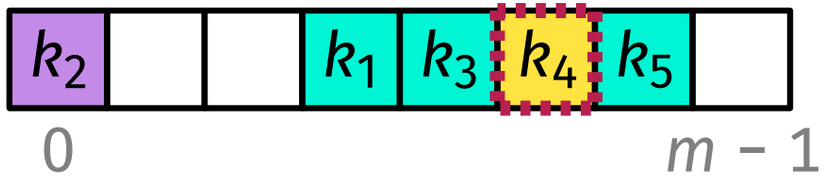
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

$h(k_1) = 3$

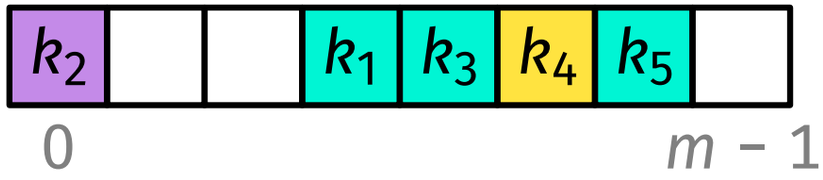
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

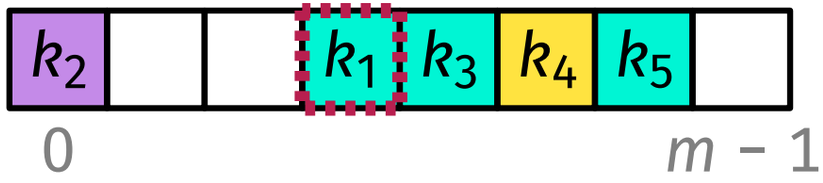
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

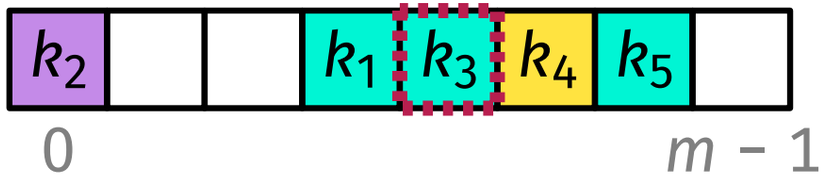
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

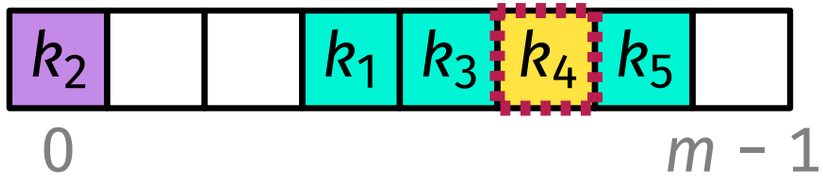
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

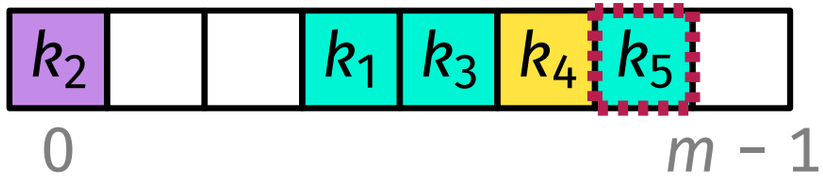
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

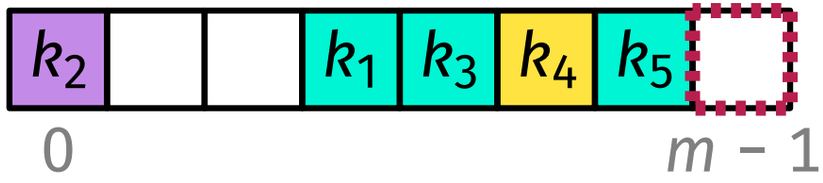
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

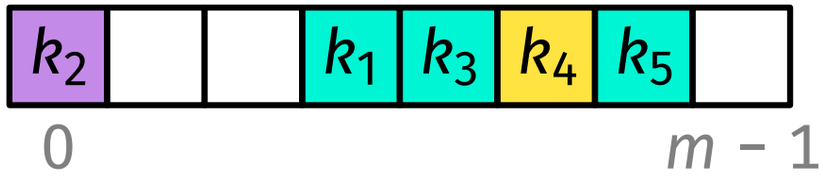
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

load factor: $\alpha = \frac{|K|}{m}$

α_{max}?

$h(k_1) = 3$

$h(k_2) = 0$

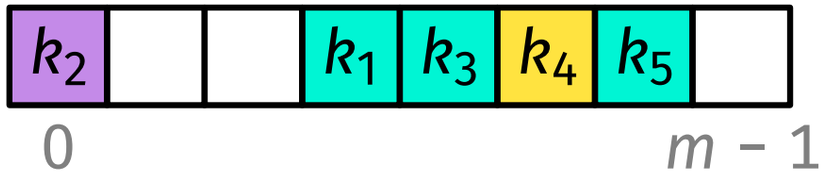
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

find(k_4)

find(k_x)

load factor: $\alpha = \frac{|K|}{m}$

$\alpha_{max} < 1$

$h(k_1) = 3$

$h(k_2) = 0$

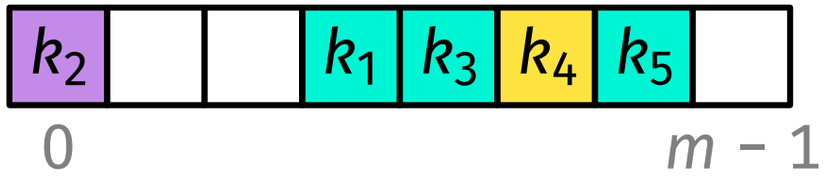
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

$h(k_1) = 3$

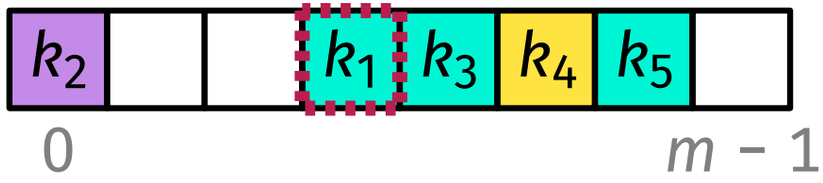
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

$h(k_1) = 3$

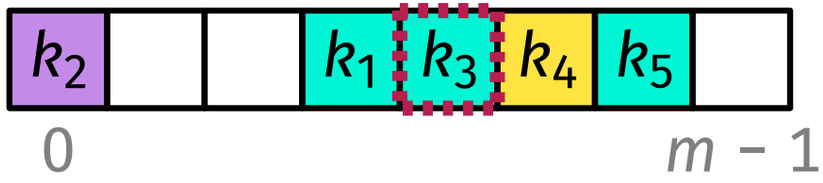
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

$h(k_1) = 3$

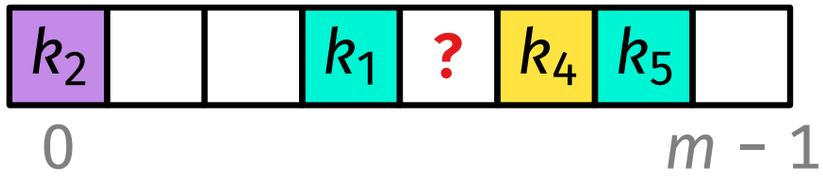
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

$h(k_1) = 3$

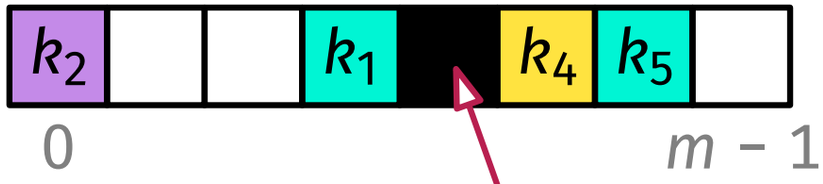
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table

insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

including tombstones!

load factor: $\alpha = \frac{|K|}{m}$

$h(k_1) = 3$

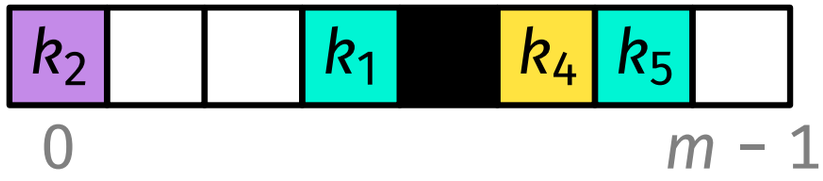
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Probing Hash Table



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

delete(k_3)

- occupied *when* find
- free *when* insert
- ignore *when* rebuild

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Is it Any Better?

		worst case
find	$O(1)$ * ¶ †	$O(n)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	$O(1)$ * (1) ¶ †	$O(n)$

* expected

1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Is it Any Better?

		worst case
find	$O(1)$ * ¶ †	$O(n)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	$O(1)$ * (1) ¶ †	$O(n)$

Weaknesses

probe sequence is bad when:

- α is high
- hash function builds clusters
- especially for unsuccessful lookups

* expected

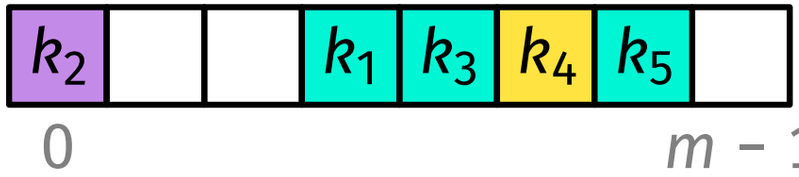
1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Robin Hood to the Rescue

Robin Hood to the Rescue



$h(k_1) = 3$

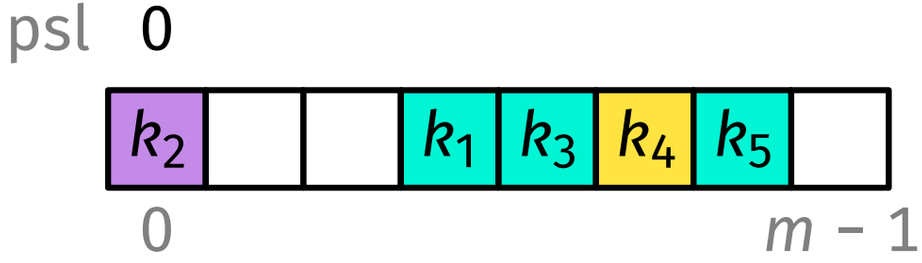
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

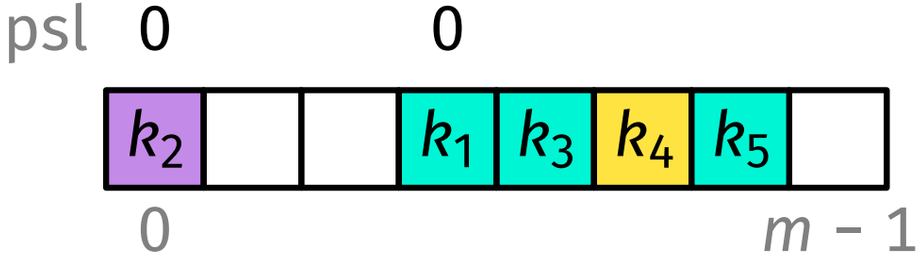
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

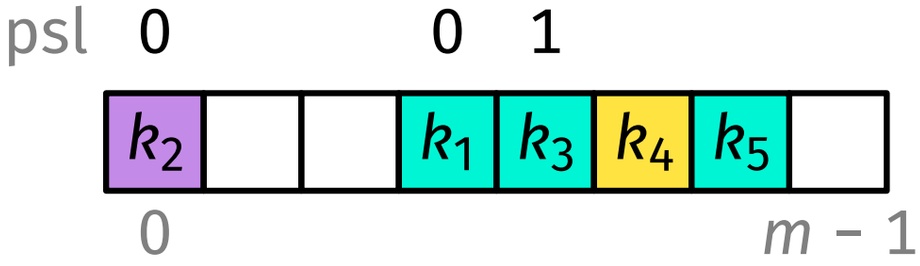
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

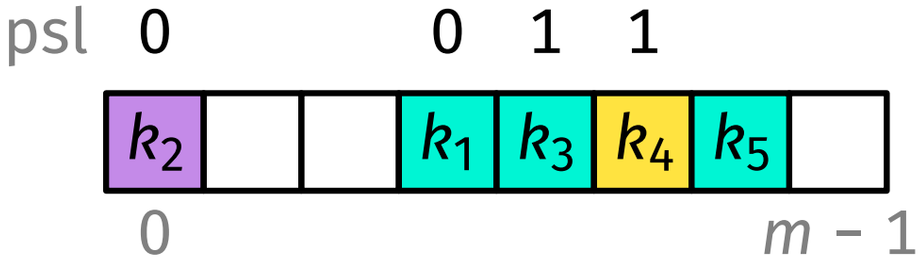
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

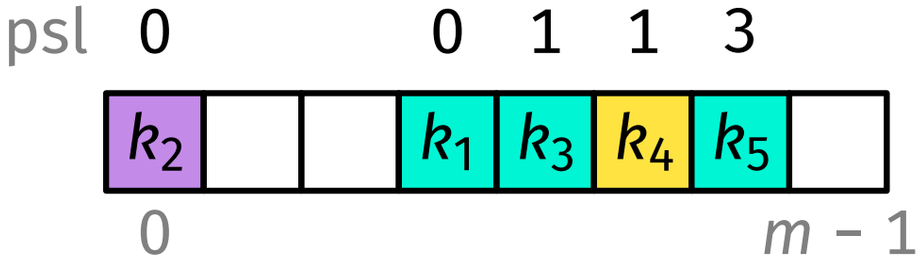
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

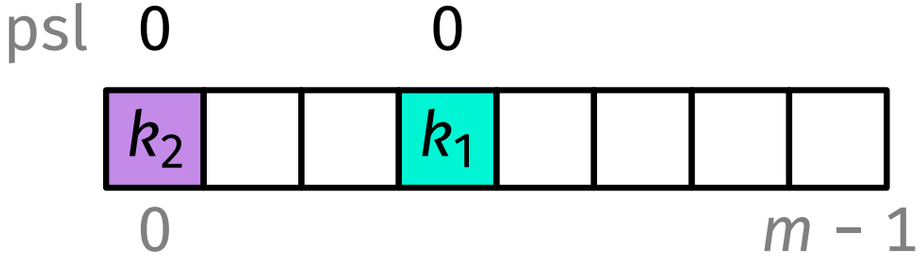
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

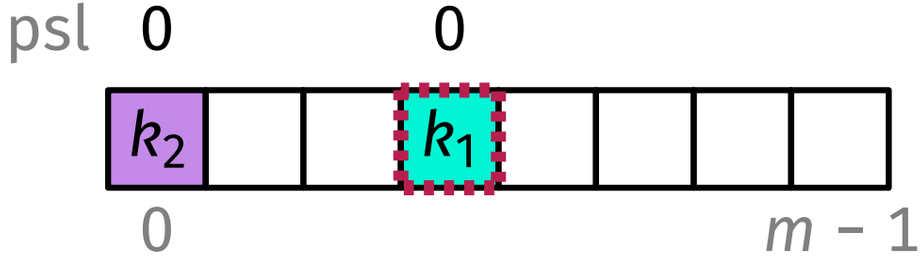
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



insert(k_3)

$h(k_1) = 3$

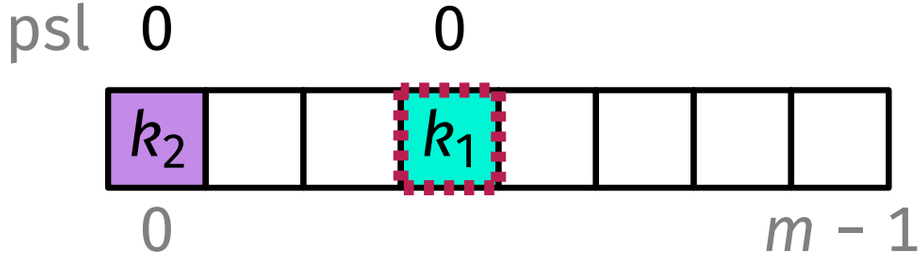
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 0

$h(k_1) = 3$

$h(k_2) = 0$

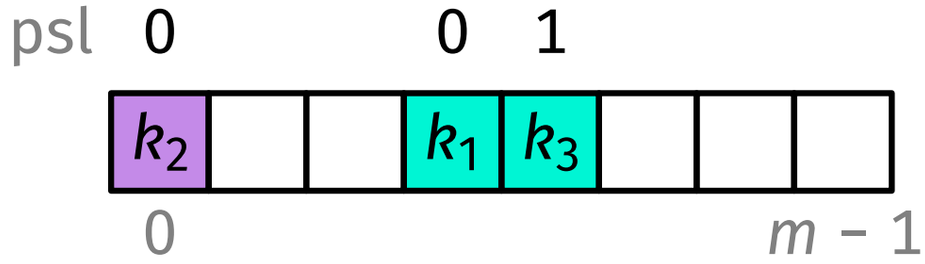
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

insert(k_3)

Robin Hood to the Rescue



insert(k_3)

$h(k_1) = 3$

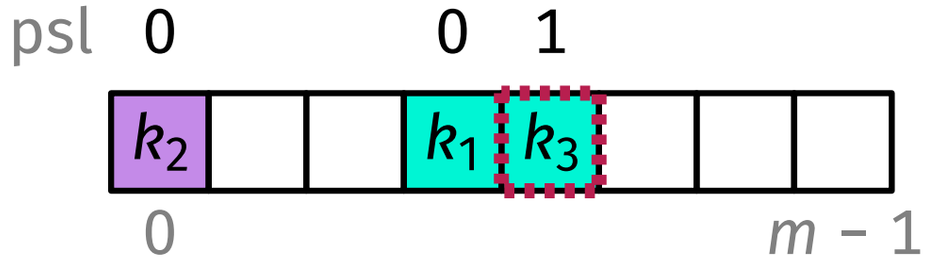
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 0

insert(k_3)

insert(k_4)

$h(k_1) = 3$

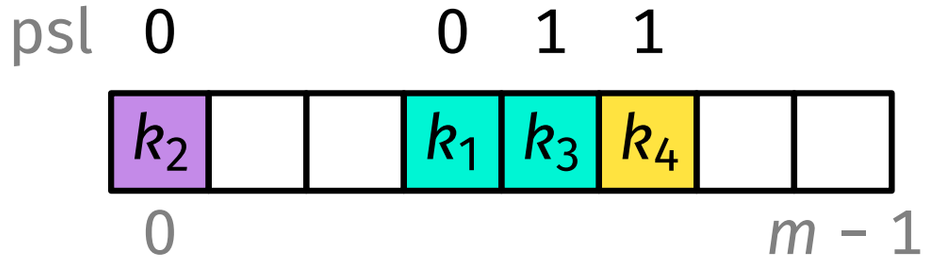
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 1

insert(k_3)

insert(k_4)

$h(k_1) = 3$

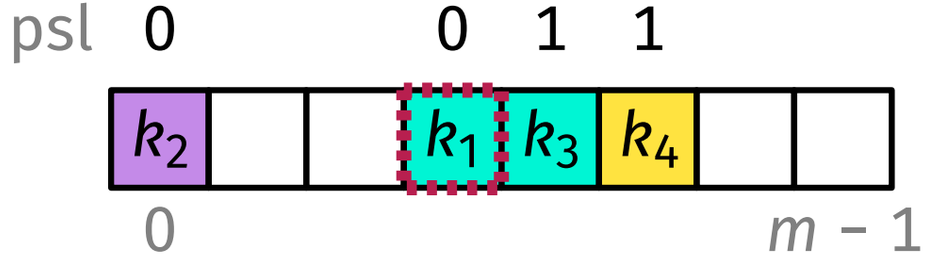
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 0

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

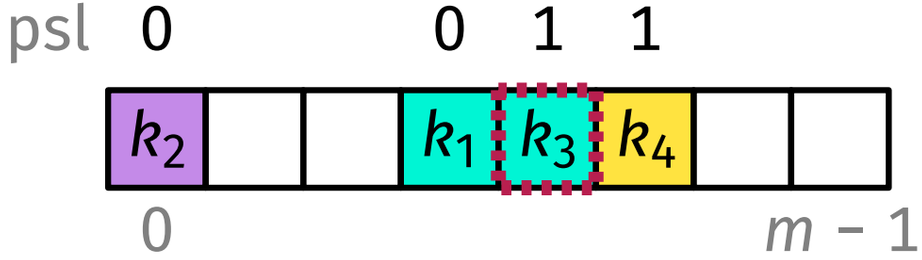
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 1

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

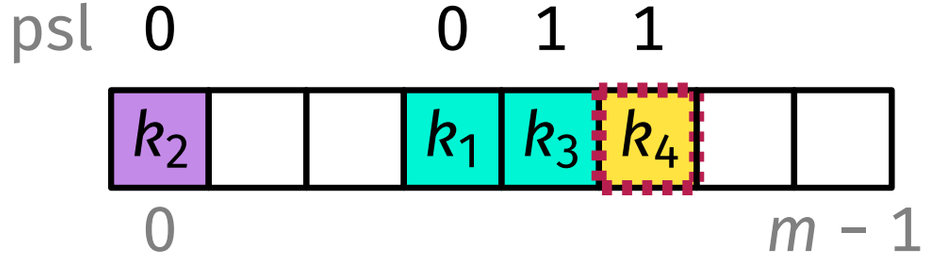
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



psl 2

insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

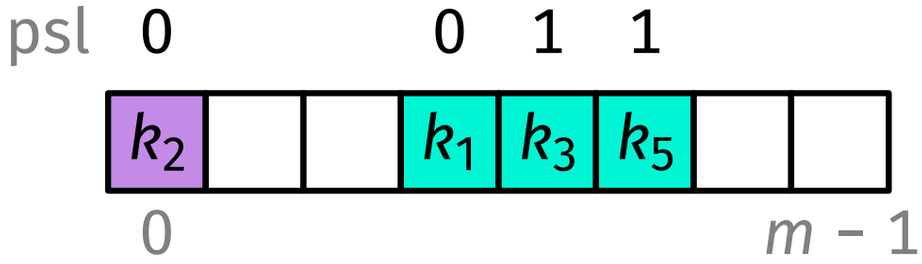
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

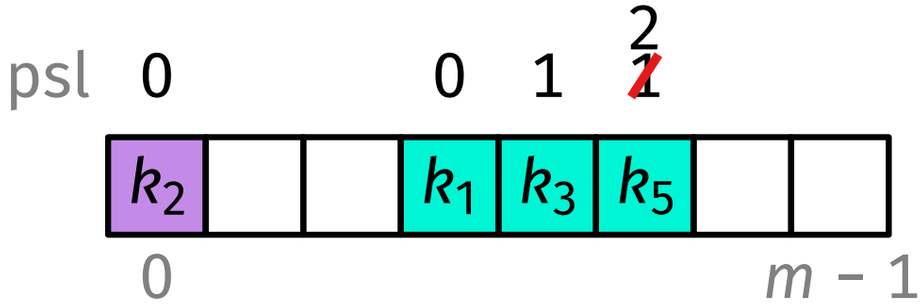
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

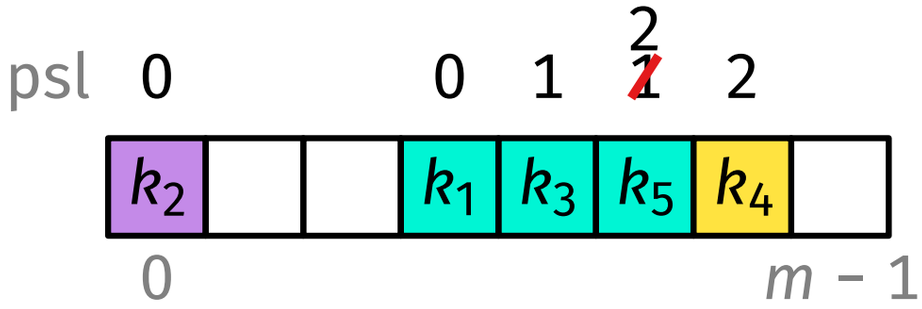
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

$h(k_1) = 3$

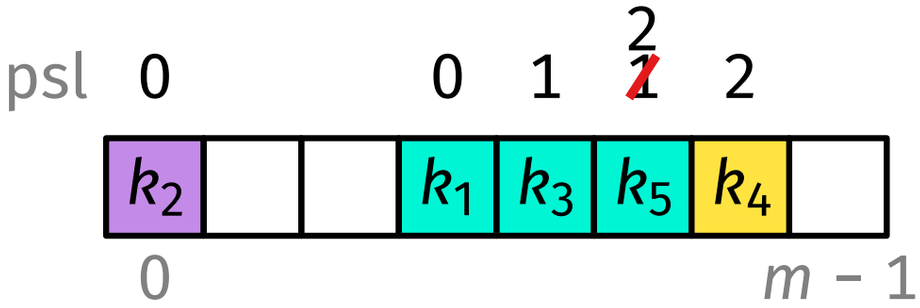
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

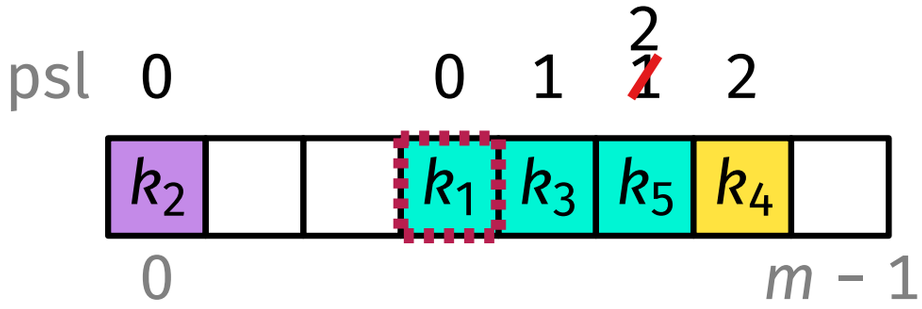
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

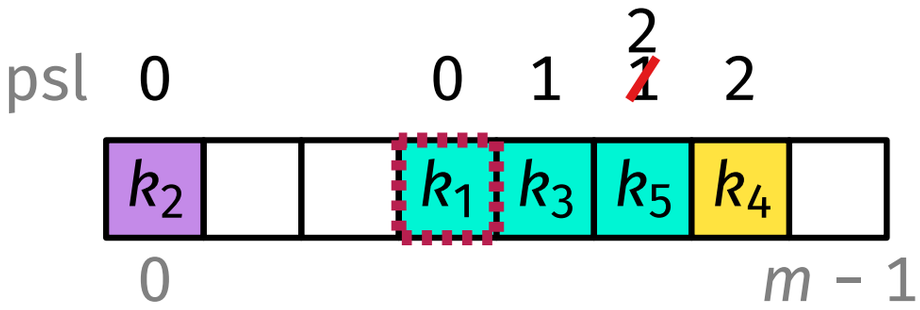
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



psl 0

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

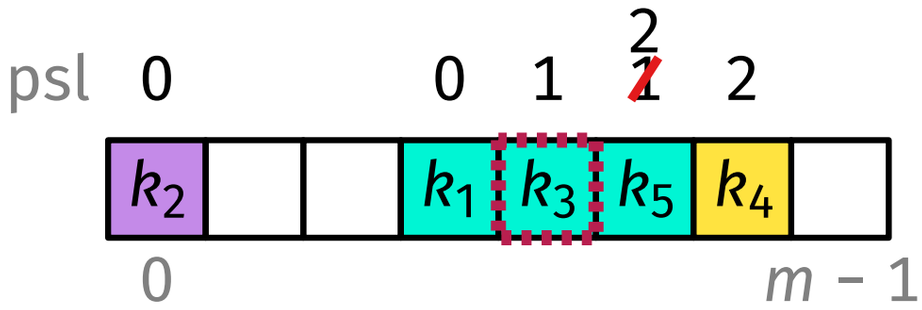
insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

Robin Hood to the Rescue



psl 1

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

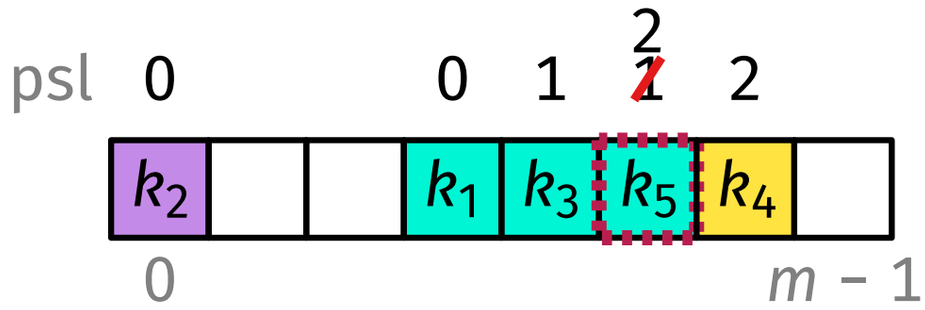
insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

Robin Hood to the Rescue



psl 2

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

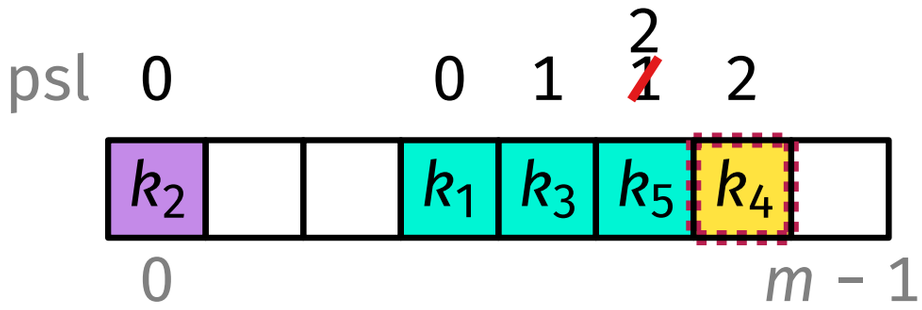
insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

Robin Hood to the Rescue



psl 3

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

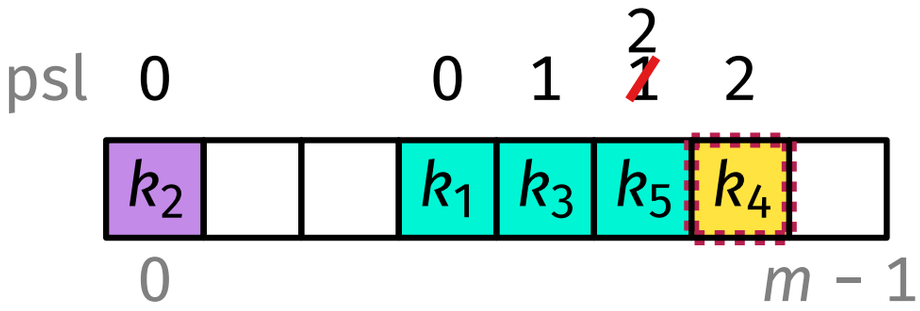
insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

Robin Hood to the Rescue



psl 3 ✓

insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

$h(k_1) = 3$

$h(k_2) = 0$

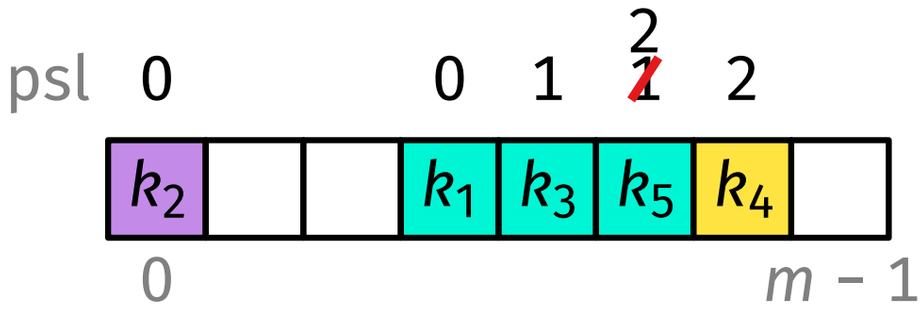
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

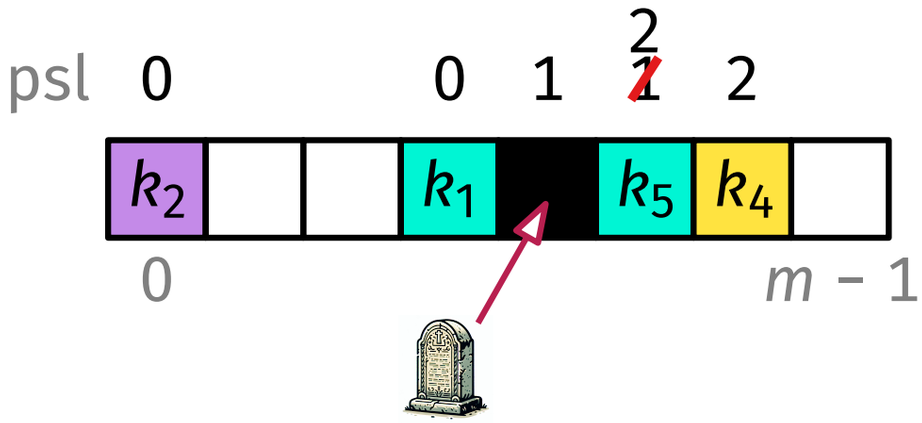
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

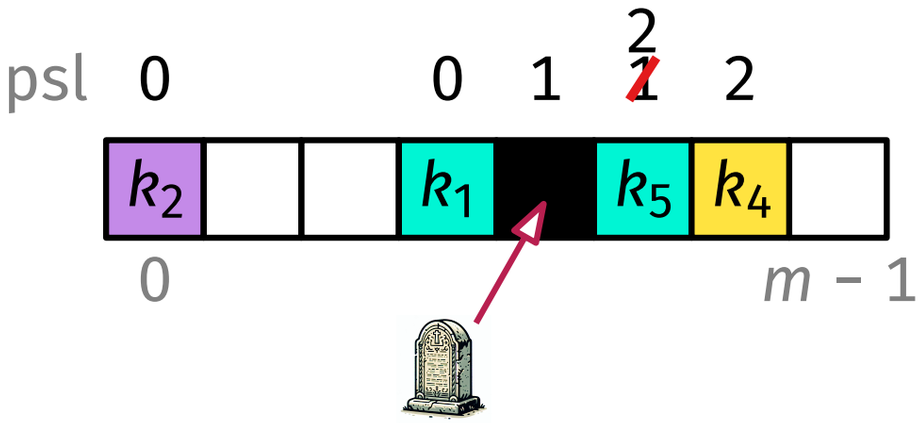
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

insert(k_6)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

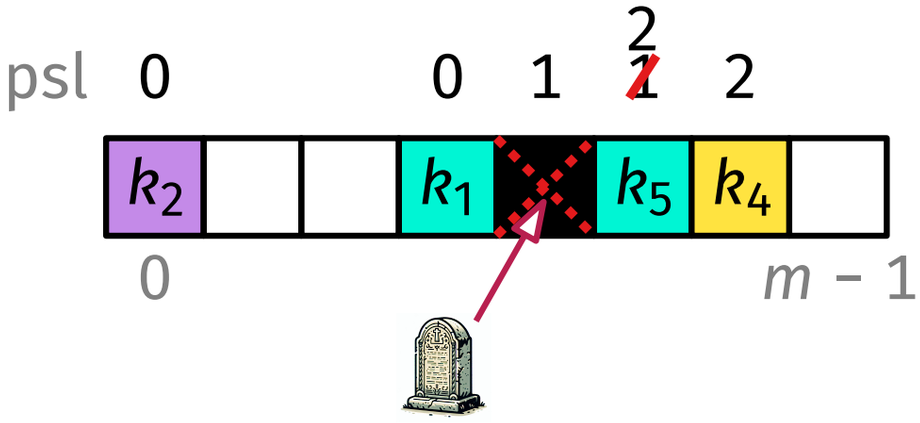
$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

$h(k_6) = 4$

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

insert(k_6)

$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

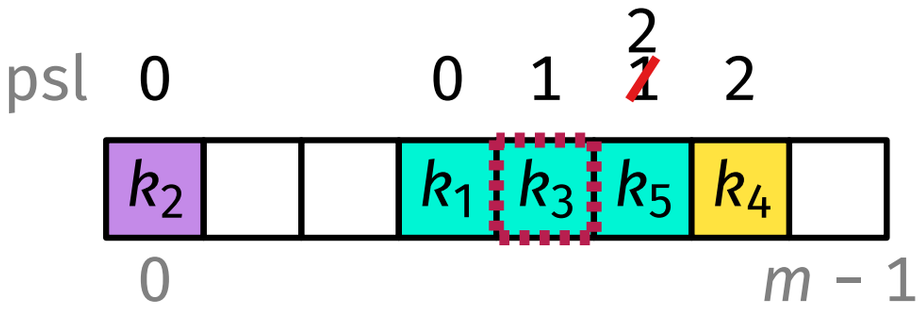
$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

$h(k_6) = 4$

Robin Hood to the Rescue



$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

insert(k_3)

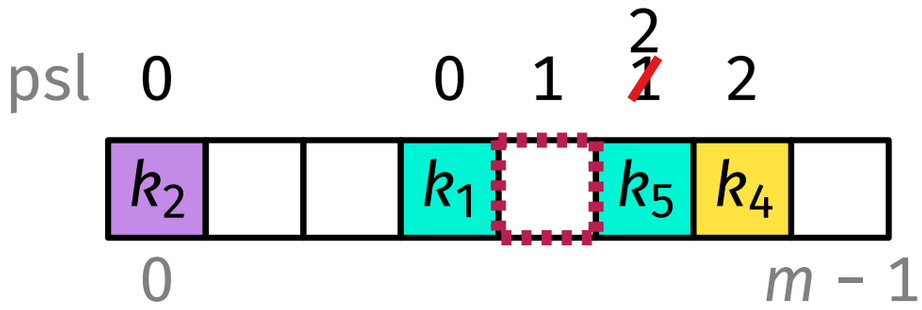
insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

Robin Hood to the Rescue



insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

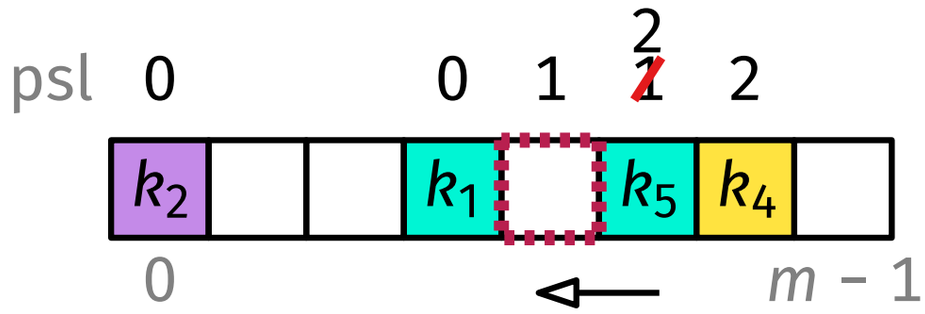
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



"backshifting"

insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

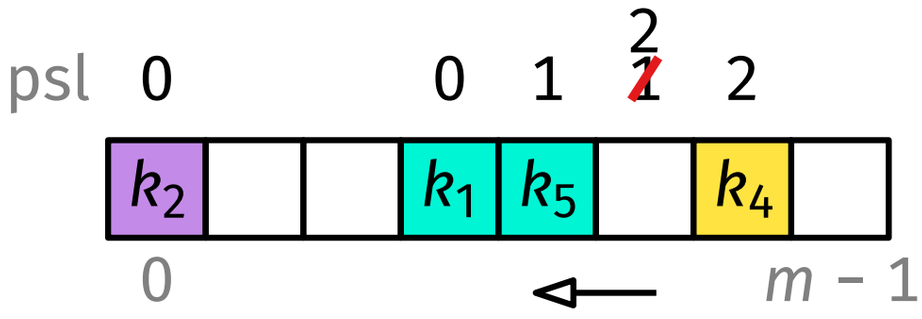
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



"backshifting"

insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

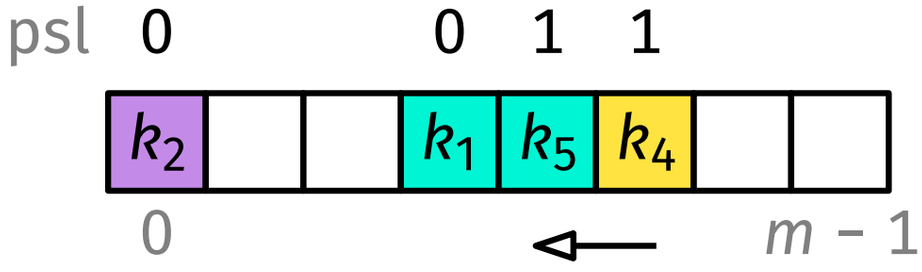
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



"backshifting"

insert(k_3)

insert(k_4)

insert(k_5)

find(k_x)

delete(k_3)

$h(k_1) = 3$

$h(k_2) = 0$

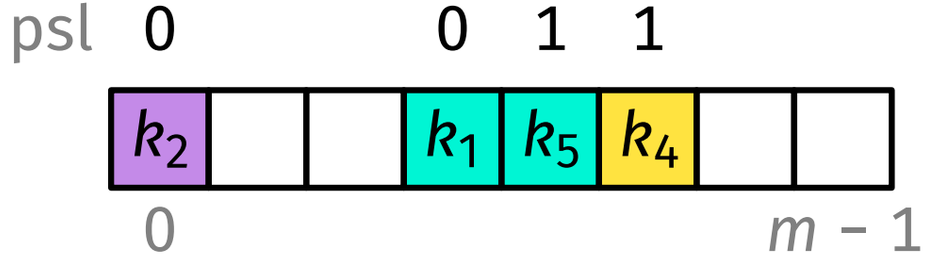
$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

$h(k_x) = 3$

Robin Hood to the Rescue



$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

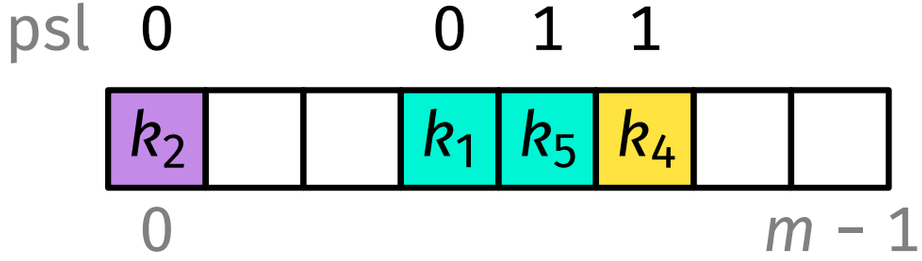
$h(k_4) = 4$

$h(k_5) = 3$

Weaknesses

- memory consumption

Robin Hood to the Rescue



$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

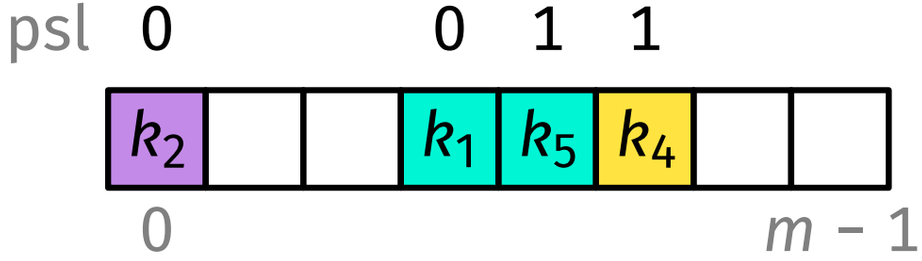
$h(k_4) = 4$

$h(k_5) = 3$

Weaknesses

- memory consumption
- average psl not better

Robin Hood to the Rescue



$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

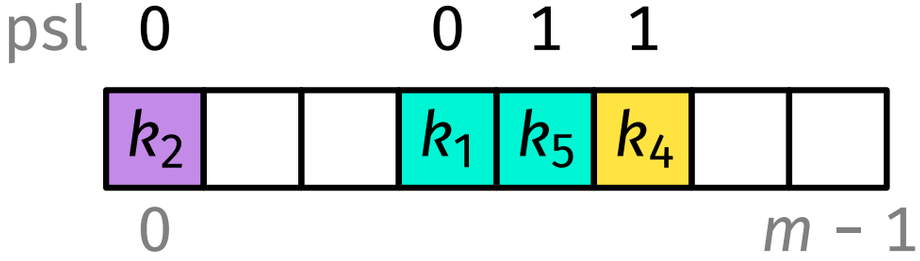
$h(k_4) = 4$

$h(k_5) = 3$

Weaknesses

- memory consumption
- average psl not better
- deletes still slow and complicated

Robin Hood to the Rescue



$h(k_1) = 3$

$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

Like Robin Hood, it does not change the average wealth (mean probe length), only its distribution

Cuckoo Hashing

		worst case
find	$O(1)$ * ¶ †	$O(n)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	$O(1)$ * (1) ¶ †	$O(n)$

* expected

1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Cuckoo Hashing

		worst case
find	$O(1)$ ~~* ¶ †~~	~~$O(n)$~~ $O(1)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	$O(1)$ * (1) ¶ †	$O(n)$

* expected

1 amortized

¶ if your load factor is reasonably low

† if you believe in your hash function

Cuckoo Hashing

		worst case
find	~~$O(1)$ * ¶ †~~	~~$O(n)$~~ $O(1)$
insert	$O(1)$ * 1 ¶ †	$O(n)$
delete	~~$O(1)$ * (1) ¶ †~~	~~$O(n)$~~ $O(1)$

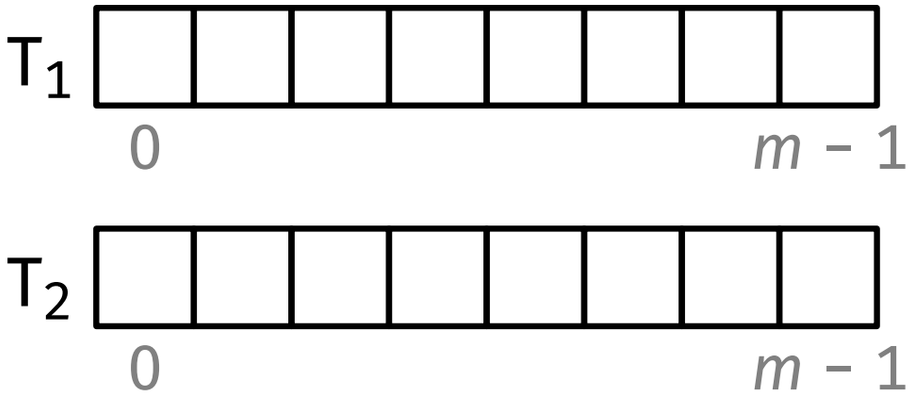
* expected

1 amortized

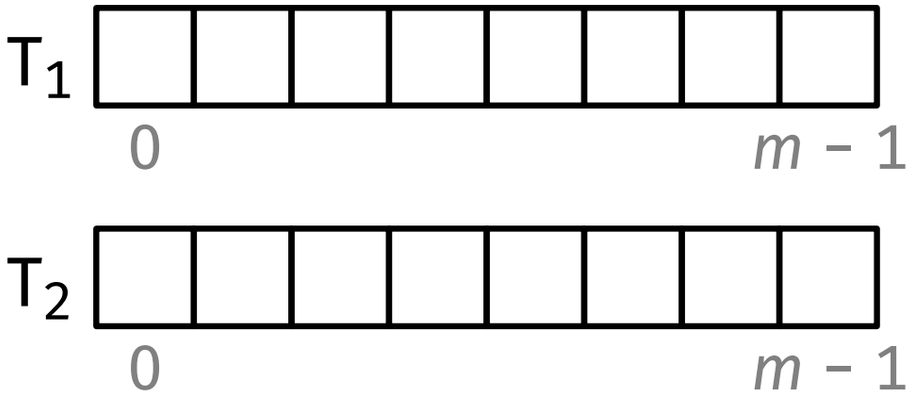
¶ if your load factor is reasonably low

† if you believe in your hash function

Cuckoo Hashing



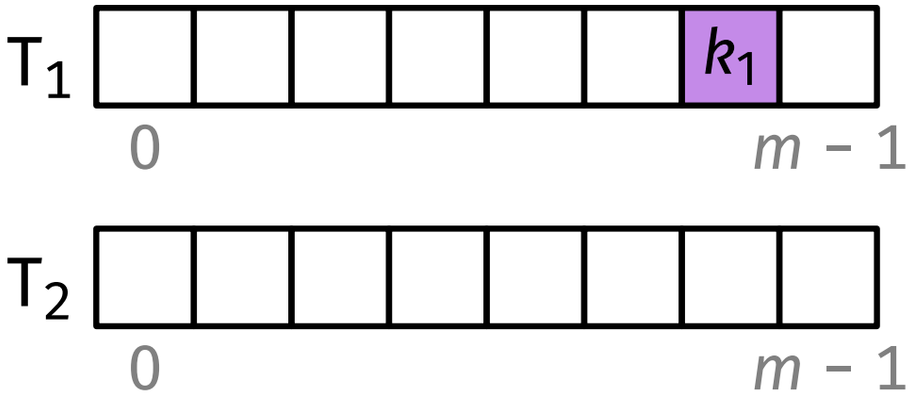
Cuckoo Hashing



	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3

insert(k_1)

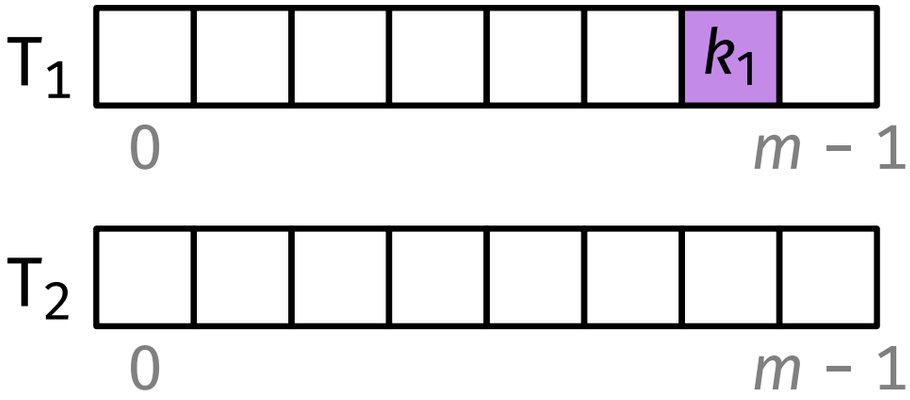
Cuckoo Hashing



	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3

insert(k_1)

Cuckoo Hashing

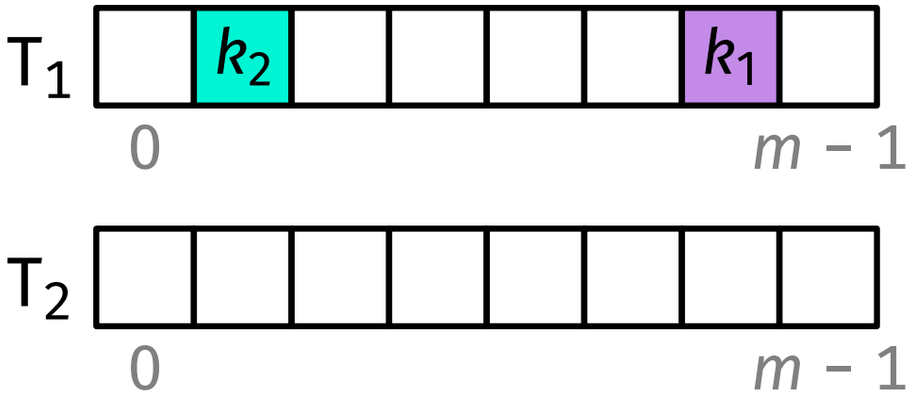


	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1

insert(k_1)

insert(k_2)

Cuckoo Hashing

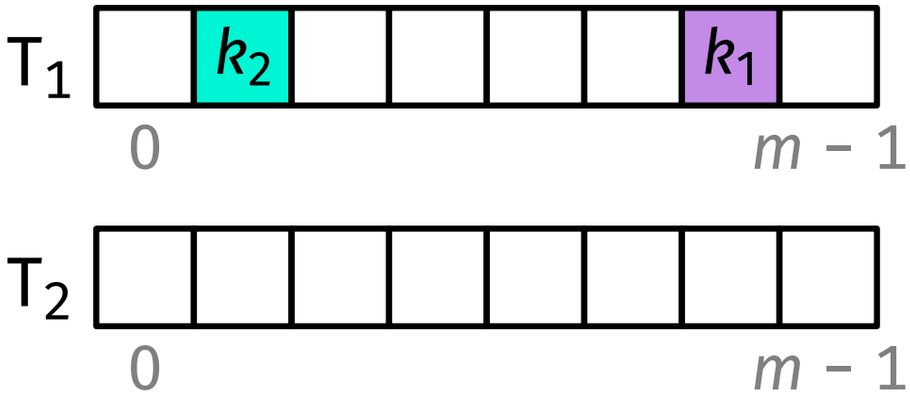


	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1

insert(k_1)

insert(k_2)

Cuckoo Hashing



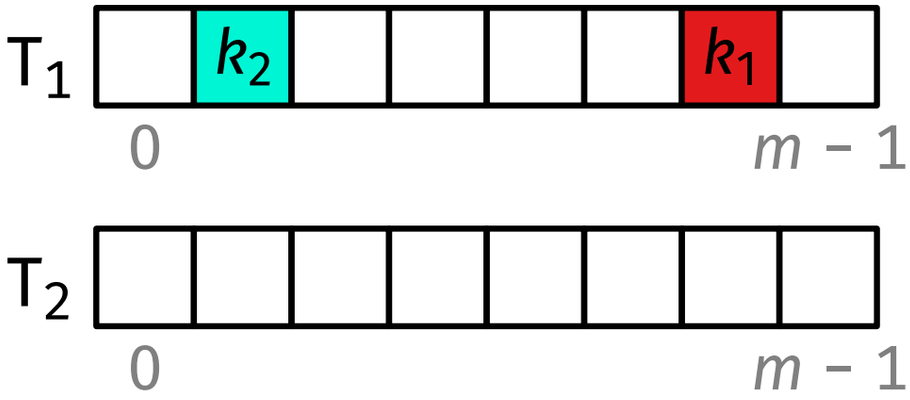
	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5

insert(k_1)

insert(k_2)

insert(k_3)

Cuckoo Hashing



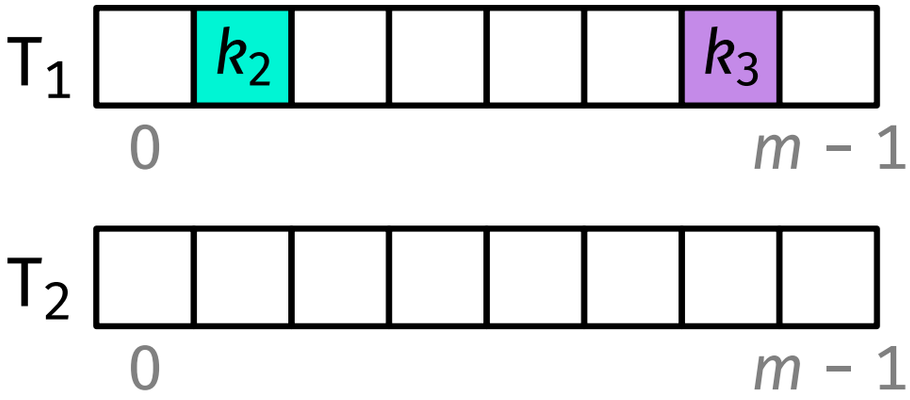
	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5

insert(k_1)

insert(k_2)

insert(k_3)

Cuckoo Hashing



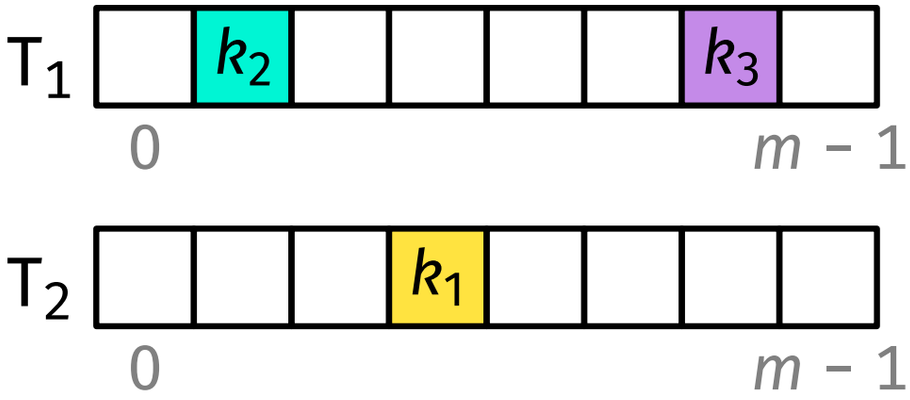
	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5

insert(k_1)

insert(k_2)

insert(k_3)

Cuckoo Hashing



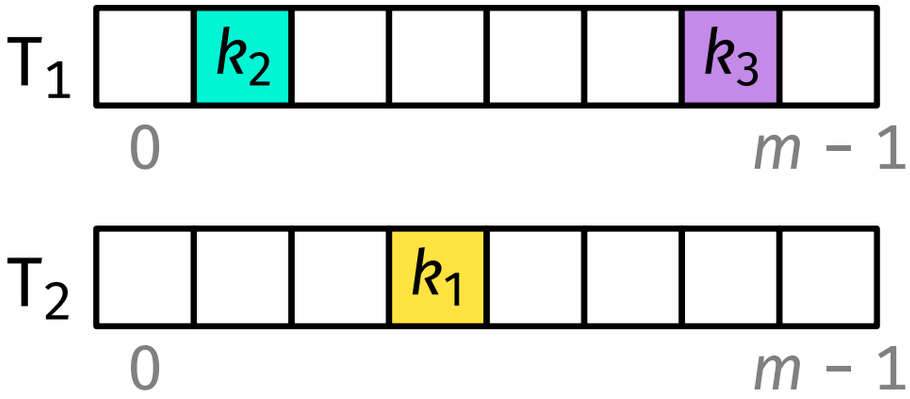
	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5

insert(k_1)

insert(k_2)

insert(k_3)

Cuckoo Hashing



insert(k_1)

insert(k_2)

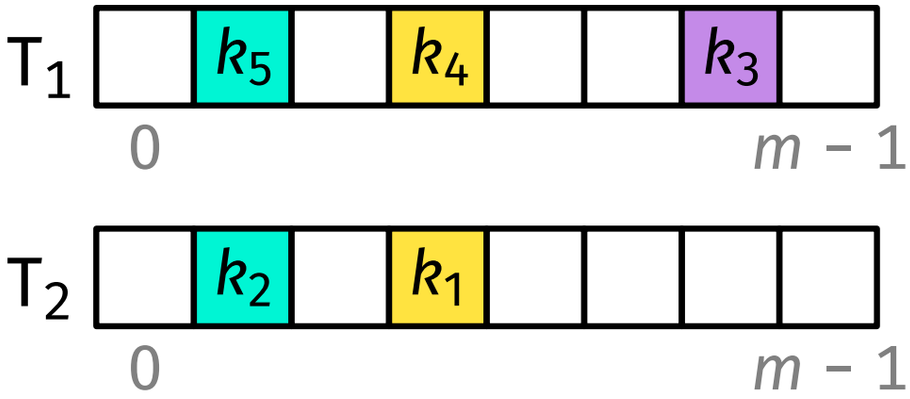
insert(k_3)

insert(k_4)

insert(k_5)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3

Cuckoo Hashing



	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3

insert(k_1)

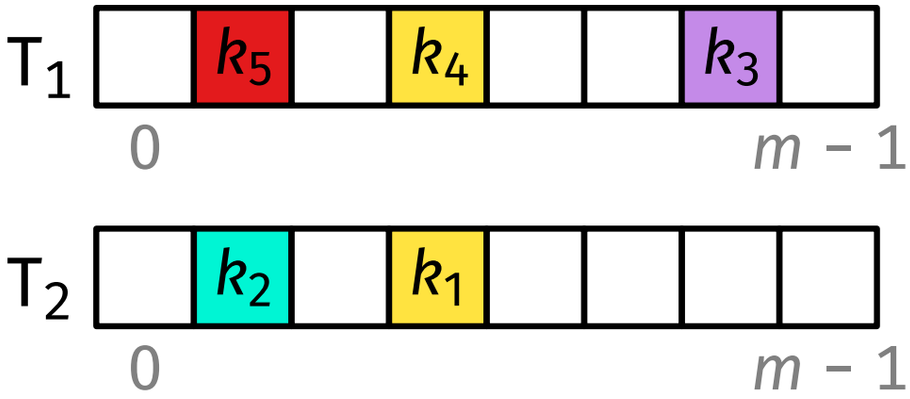
insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

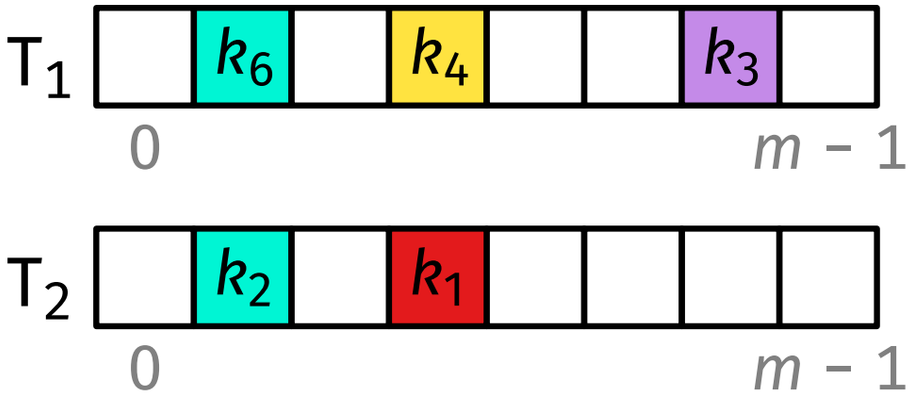
insert(k_4)

insert(k_5)

insert(k_6)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

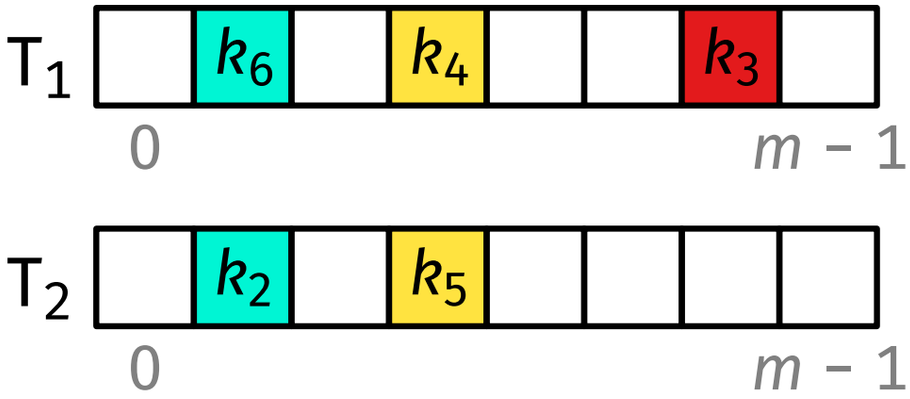
insert(k_4)

insert(k_5)

insert(k_6)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

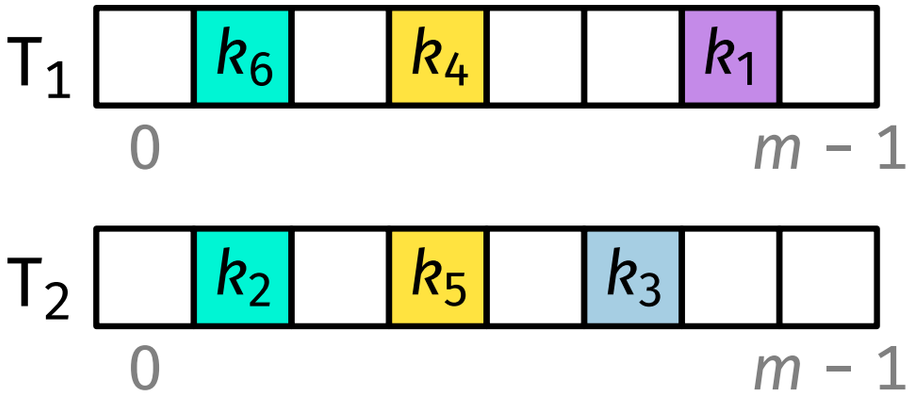
insert(k_4)

insert(k_5)

insert(k_6)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

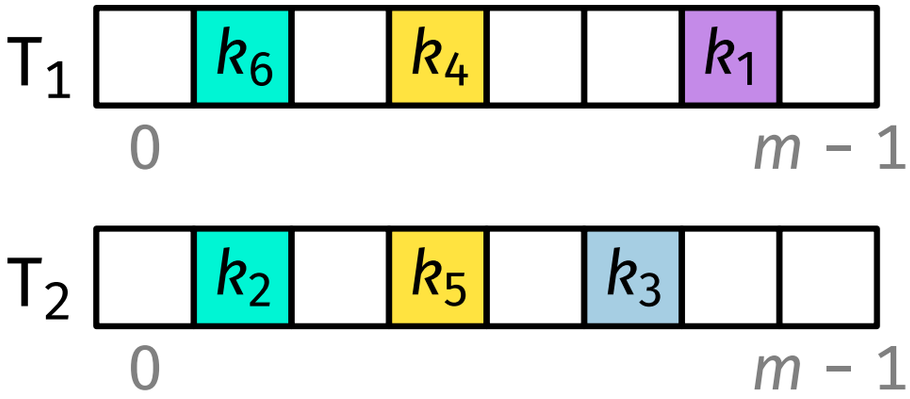
insert(k_4)

insert(k_5)

insert(k_6)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

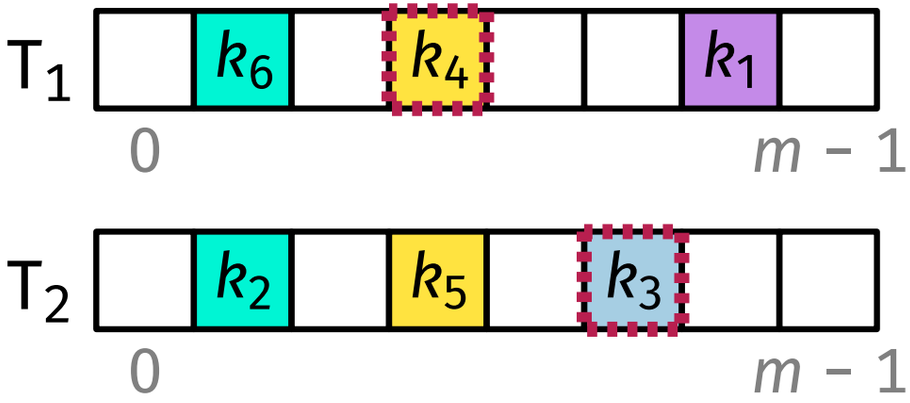
insert(k_5)

insert(k_6)

find(k_4)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

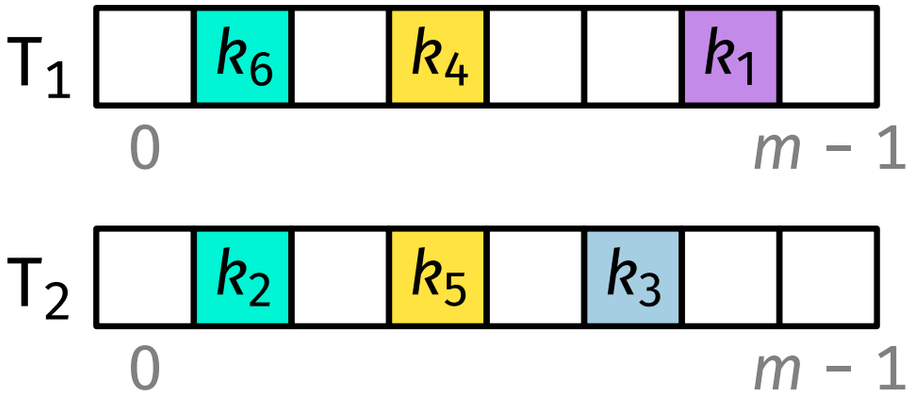
insert(k_5)

insert(k_6)

find(k_4)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

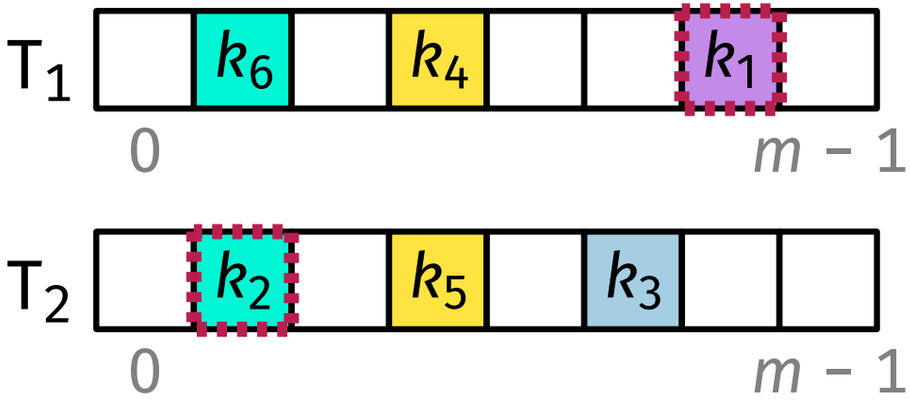
insert(k_6)

find(k_4)

find(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

Cuckoo Hashing



insert(k_1)

insert(k_2)

insert(k_3)

insert(k_4)

insert(k_5)

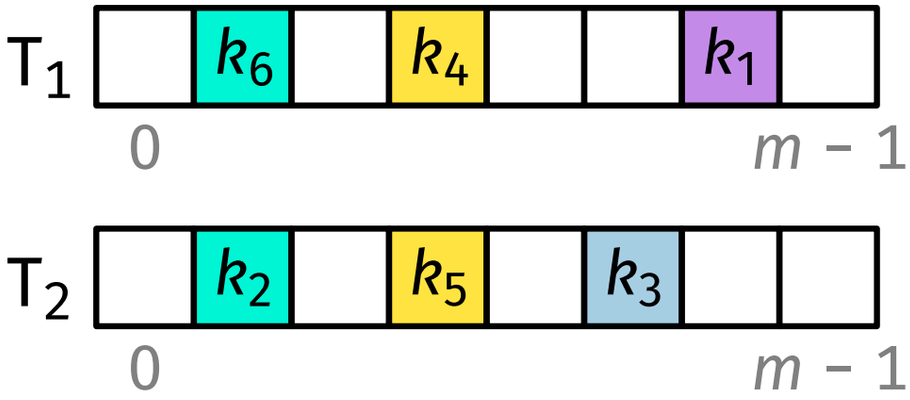
insert(k_6)

find(k_4)

find(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

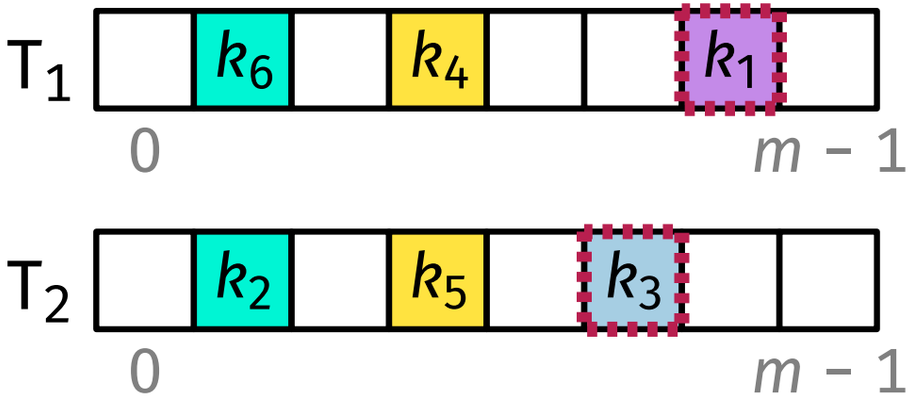
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

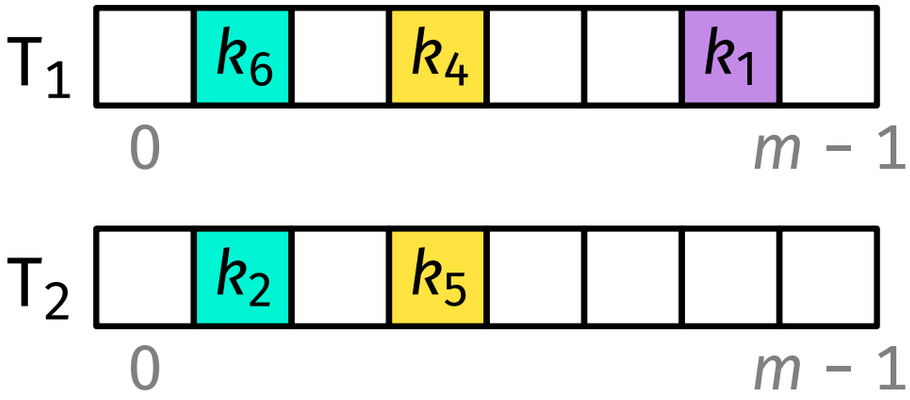
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

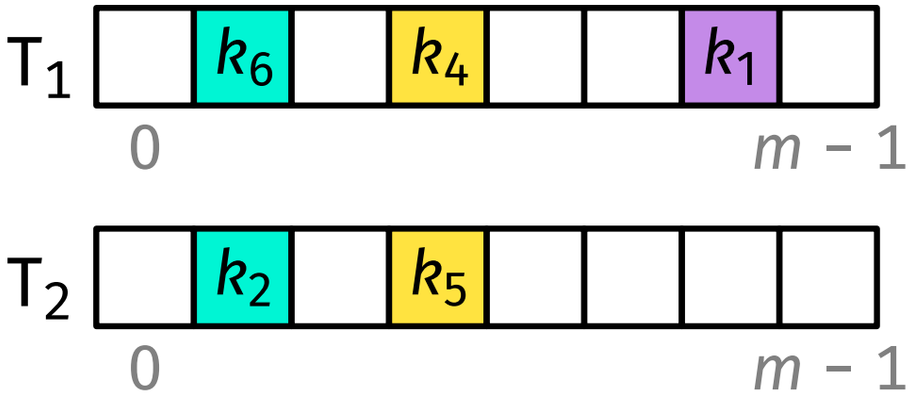
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

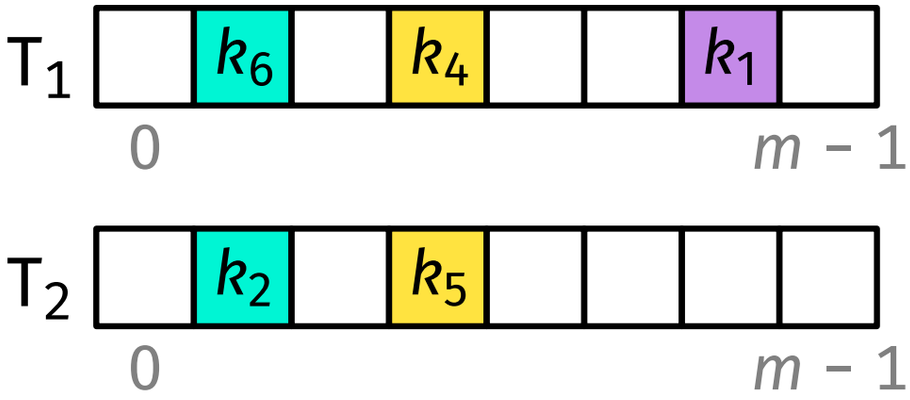
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

Cuckoo Hashing

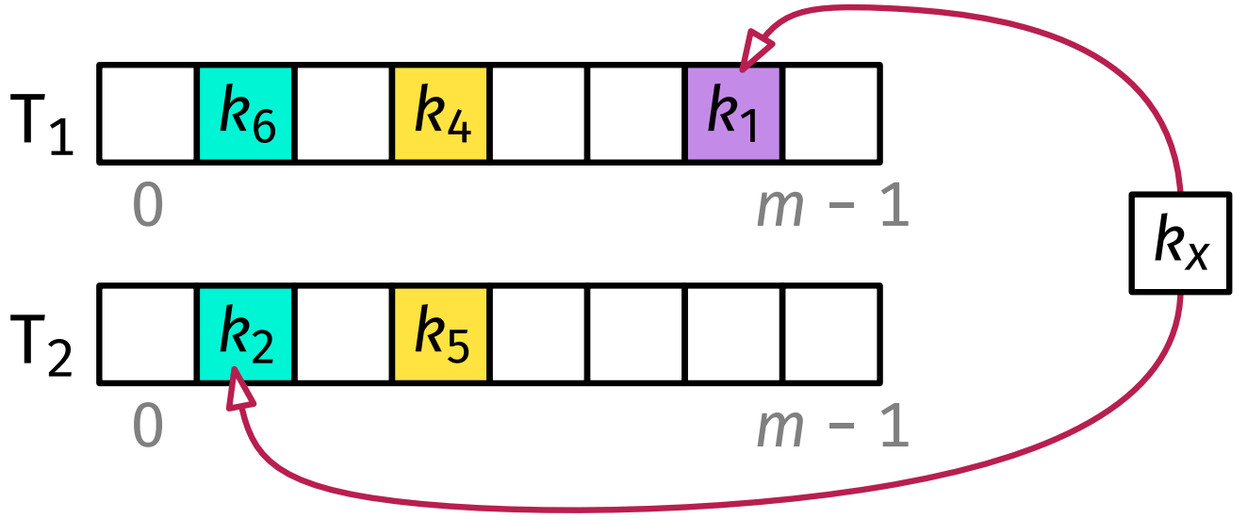


k_x

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

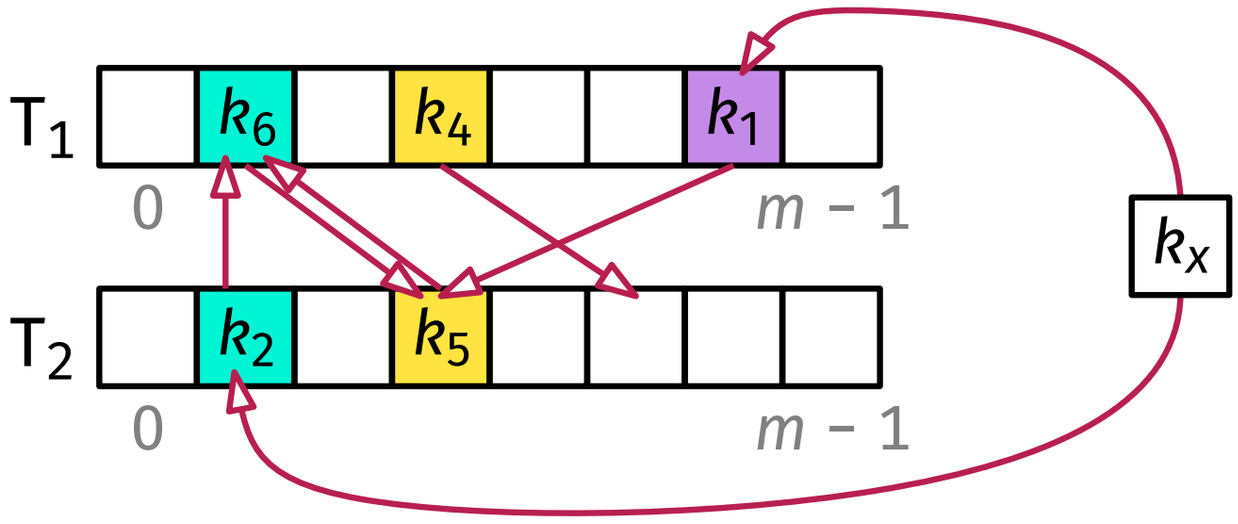
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

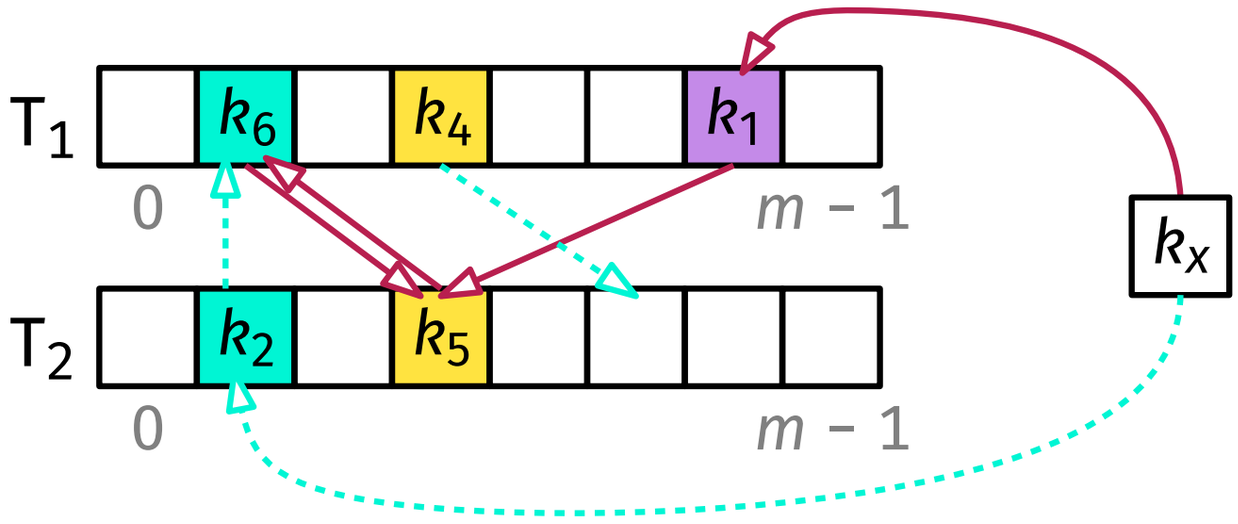
Cuckoo Hashing



	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

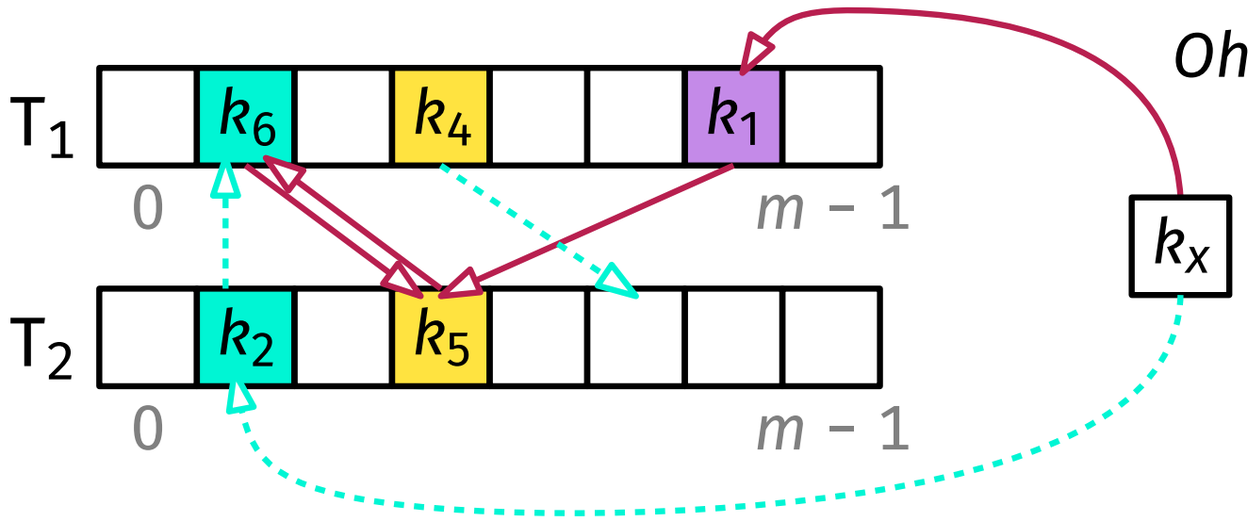
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

Cuckoo Hashing

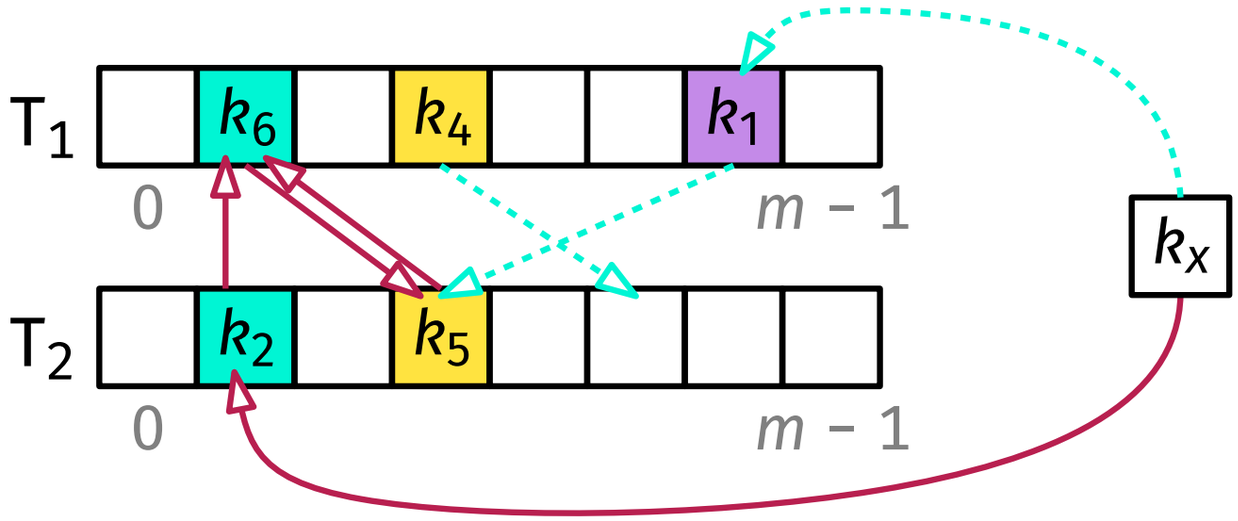


Oh no! A cycle ☹

insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

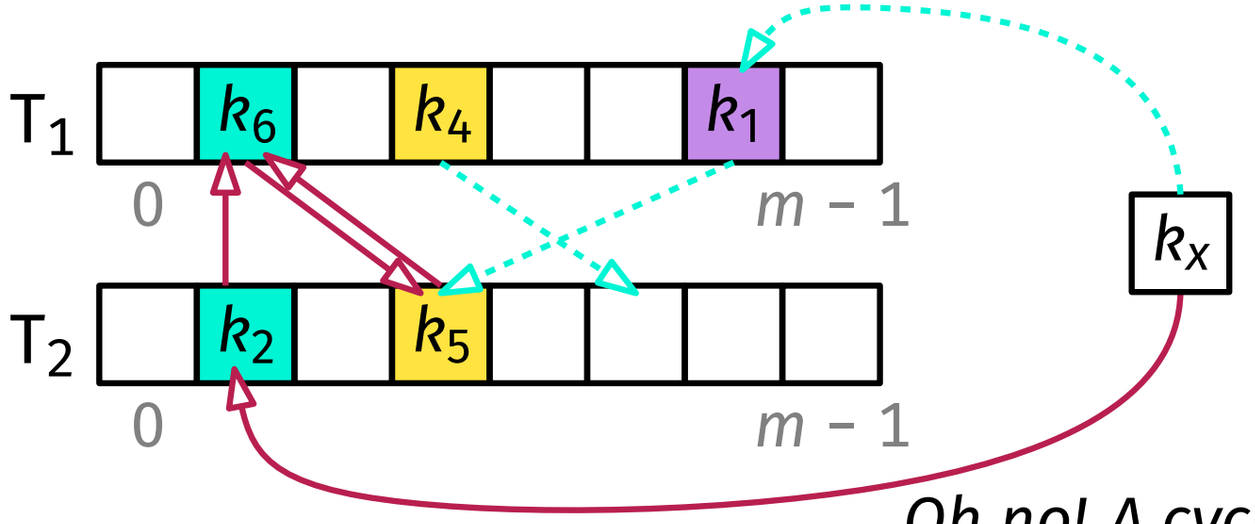
Cuckoo Hashing



insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

Cuckoo Hashing

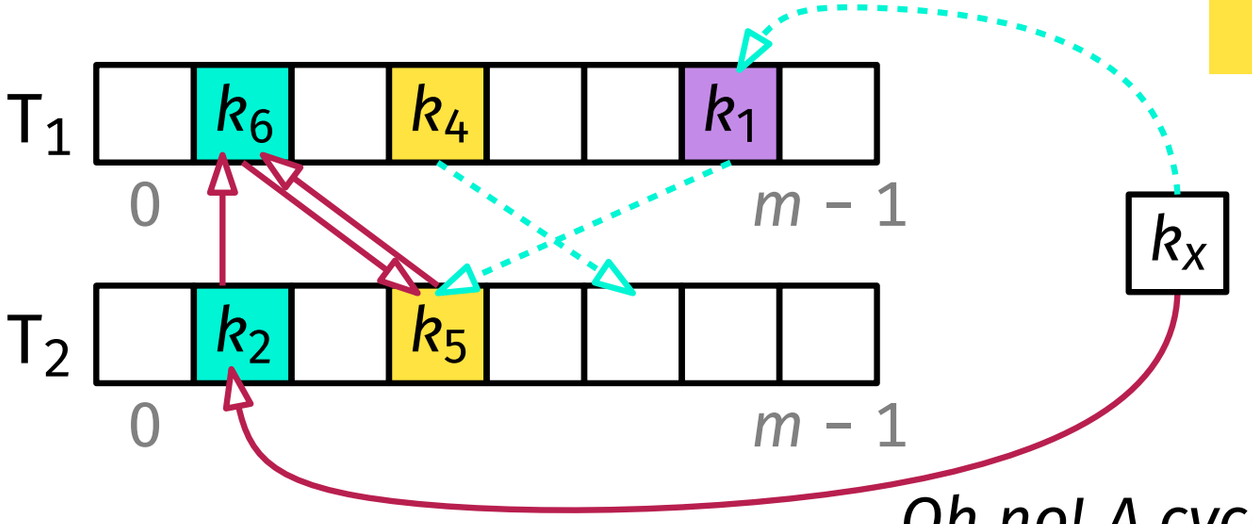


Oh no! A cycle ☹

insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

Cuckoo Hashing



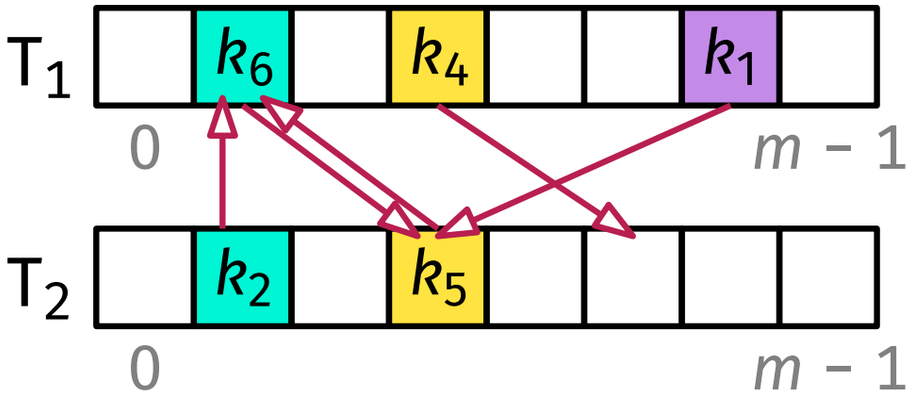
It's time to rebuild your hash table!

Oh no! A cycle ☹

	$h_1(\cdot)$	$h_2(\cdot)$
	~~1~~ 3	~~2~~ 4
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

insert(k_1)
insert(k_2)
insert(k_3)
insert(k_4)
insert(k_5)
insert(k_6)
find(k_4)
find(k_x)
delete(k_3)
insert(k_x)

Cuckoo Hashing



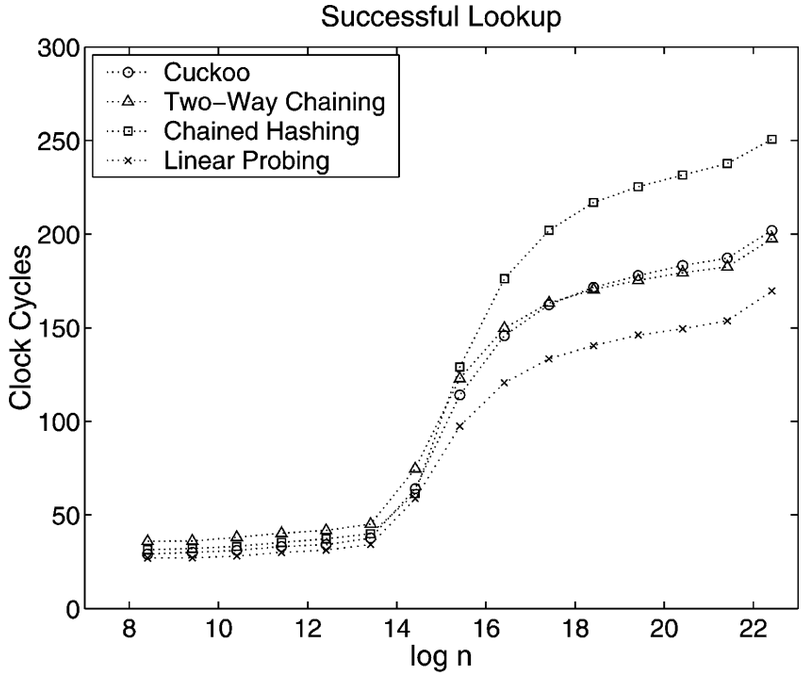
Problems

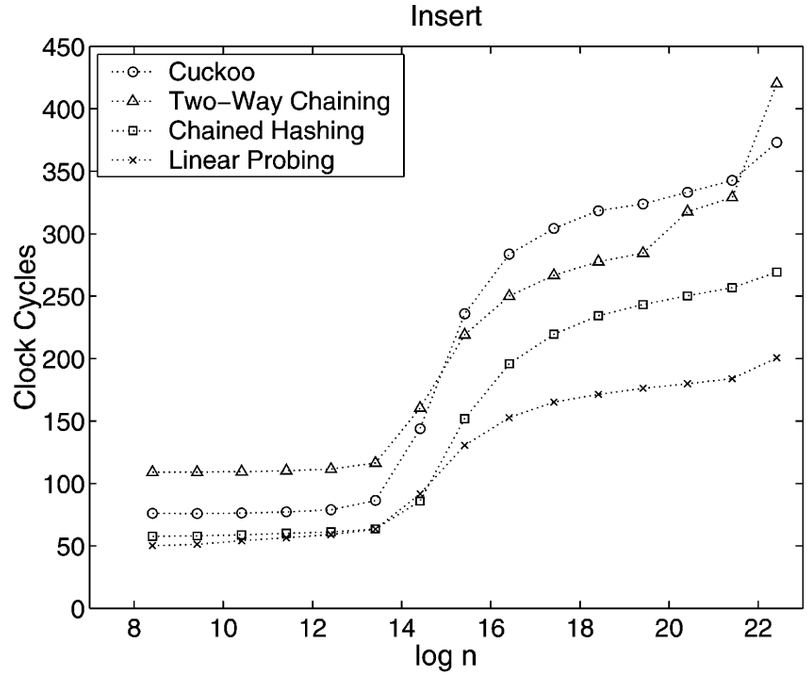
- Insert in $O(1)$*¹¶† only as long as $\alpha < 0.5$
- requires a good hash function
- not cache friendly

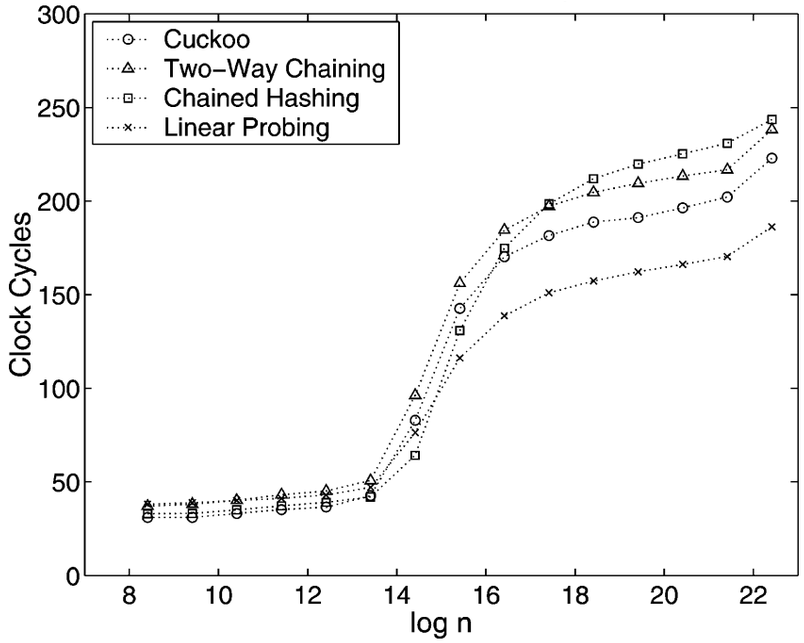
	$h_1(\cdot)$	$h_2(\cdot)$
k_1	6	3
k_2	1	1
k_3	6	5
k_4	3	5
k_5	1	3
k_6	1	3
k_x	6	1

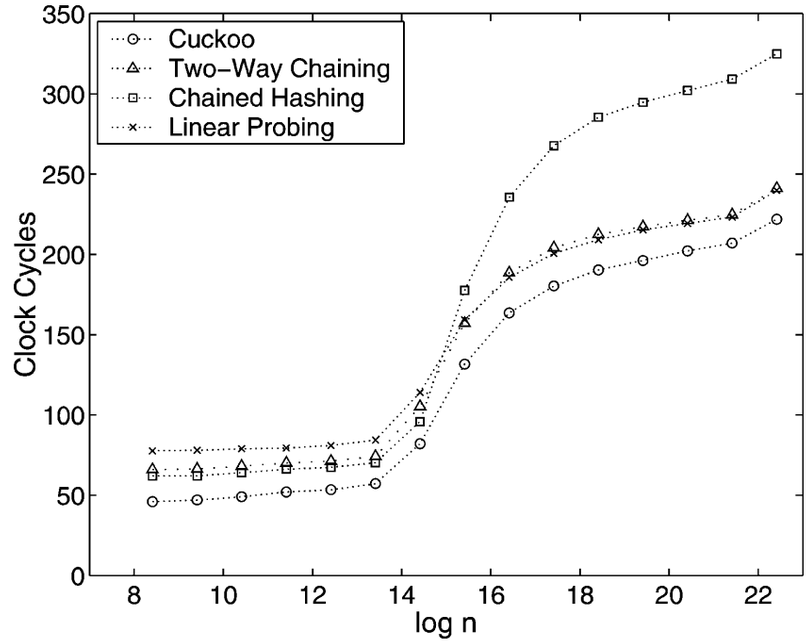
“In theory, there is no difference between theory and practice;

“In theory, there is no difference between theory and practice; but, in practice, there is.”

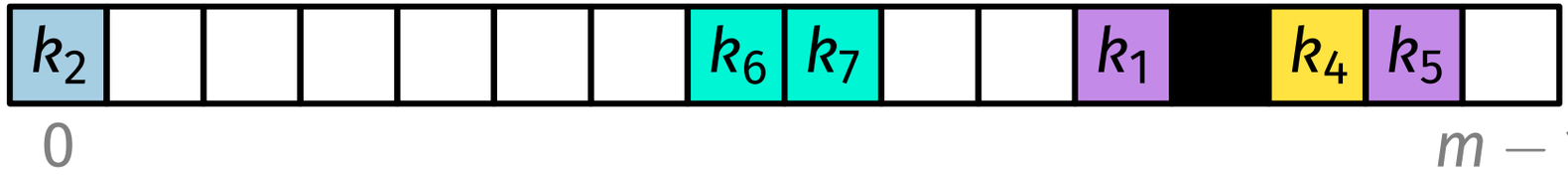

Successful Lookup
Cuckoo
Two−Way Chaining
Chained Hashing
Linear Probing
Clock Cycles
log n
0
50
100
150
200
250
300
8
10
12
14
16
18
20
22


Insert
Cuckoo
Two−Way Chaining
Chained Hashing
Linear Probing
Clock Cycles
log n
0
50
100
150
200
250
300
350
400
450
8
10
12
14
16
18
20
22

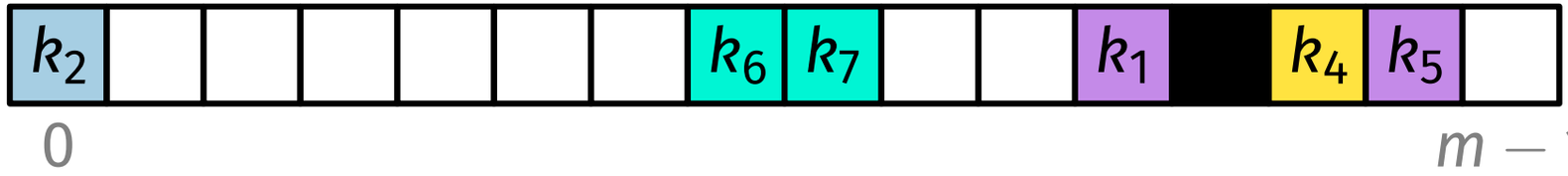

Unsuccessful Lookup
Cuckoo
Two−Way Chaining
Chained Hashing
Linear Probing
Clock Cycles
log n
0
50
100
150
200
250
300
8
10
12
14
16
18
20
22


Delete
Cuckoo
Two−Way Chaining
Chained Hashing
Linear Probing
Clock Cycles
log n
0
50
100
150
200
250
300
350
8
10
12
14
16
18
20
22

Swisstable



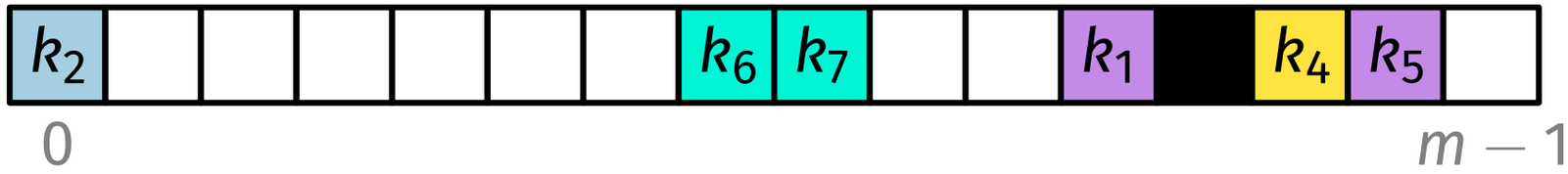
Swisstable



Idea

Use a little bit of extra memory for parallel lookups with uncertainty

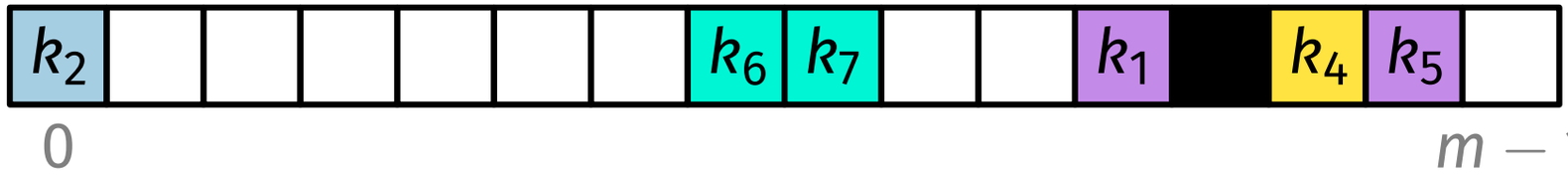
Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Swisstable

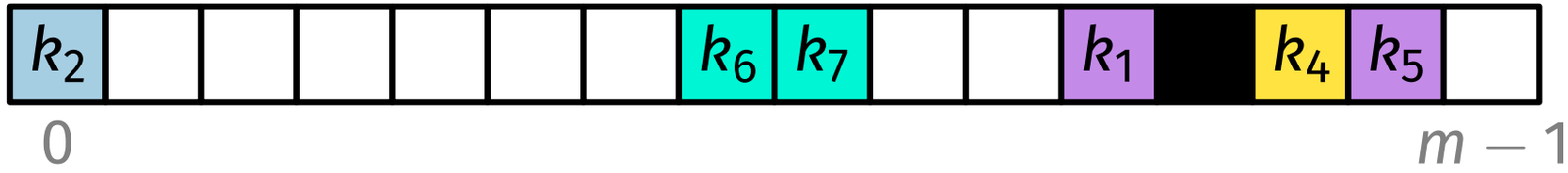


Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

$h\colon U \to \mathrm{u64}$

Swisstable



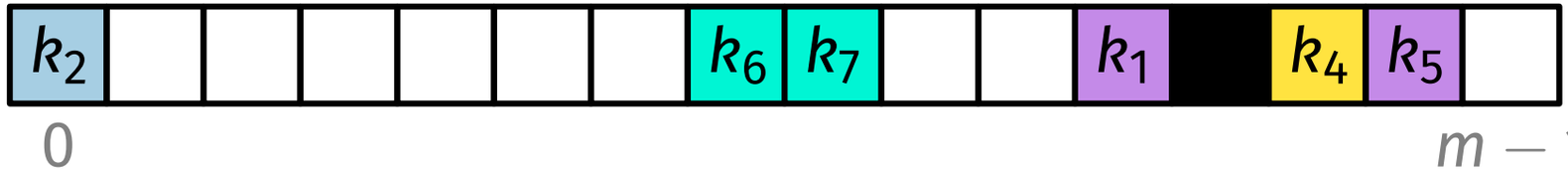
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

$h\colon U \to \mathrm{u64}$

$H_1(k) = h(k) \bmod m$

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

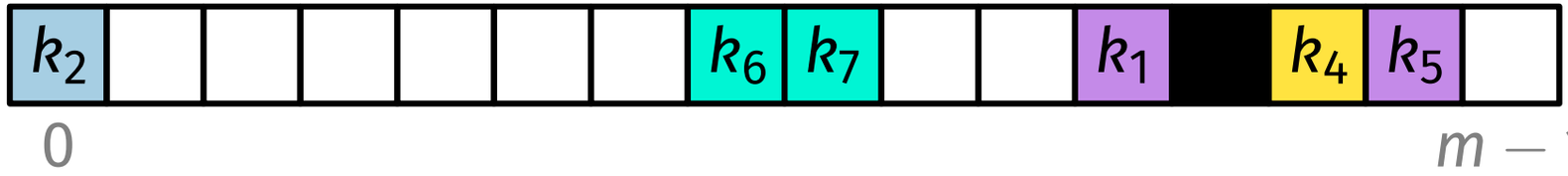
$h\colon U \to \mathrm{u64}$

$H_1(k) = h(k) \bmod m$

$H_2(k) = h(k) \gg 57$

"first 7 bits"

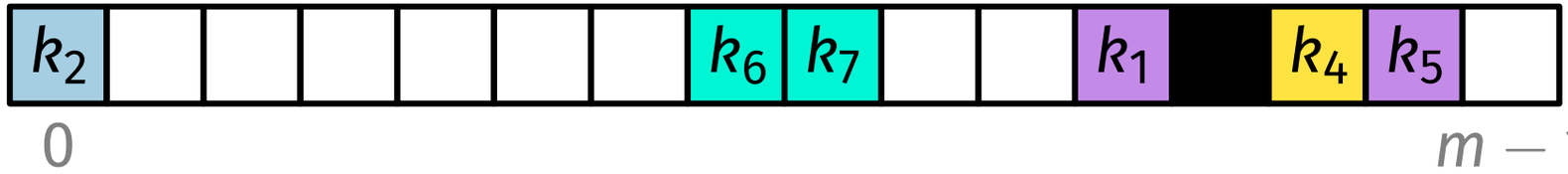
Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Swisstable



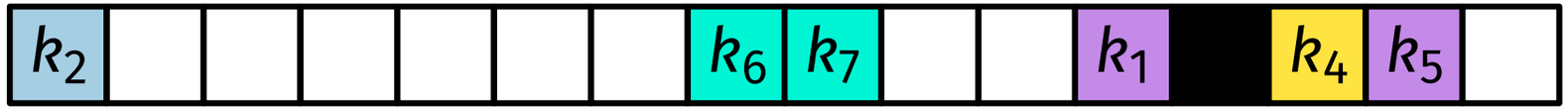
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

m bytes of

`0×FF`	= empty
`0×80`	= deleted 
`0×00`…`0×7F`	= occupied

Swisstable



Ingredients:

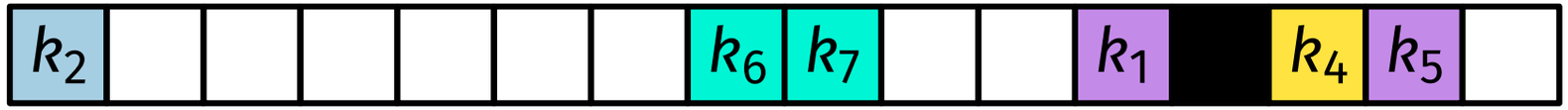
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

m bytes of

`0×FF`	= empty
`0×80`	= deleted 
`0×00`…`0×7F`	= occupied

use H_2 for the occupied bytes

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

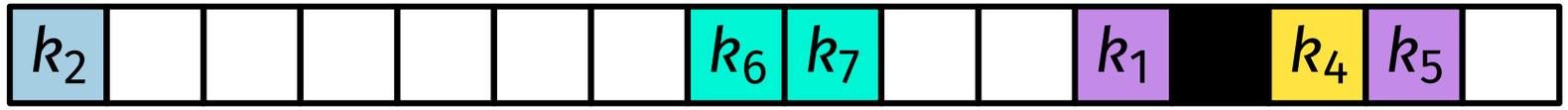
m bytes of

`0×FF`	= empty
`0×80`	= deleted 
`0×00`…`0×7F`	= occupied

use H_2 for the occupied bytes

\+ reprise of the first 16 bytes

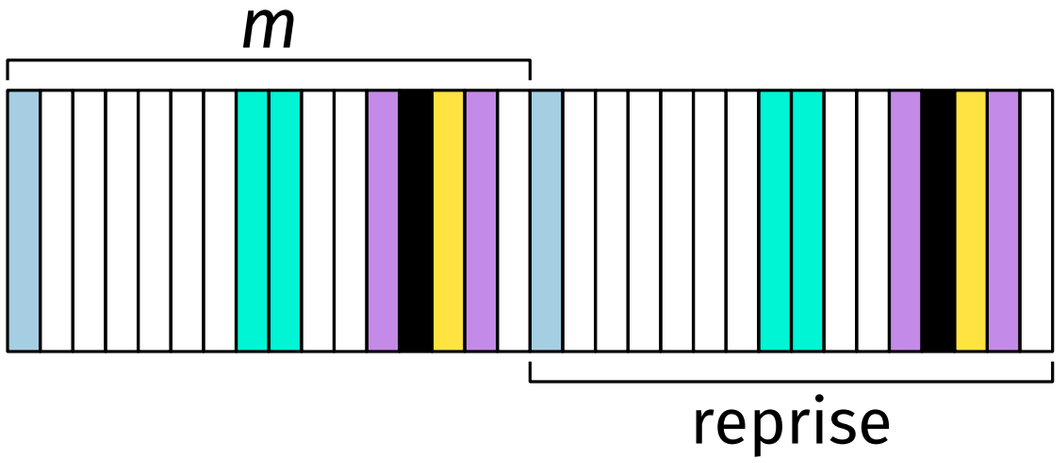
Swisstable



0 $m-1$

Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions



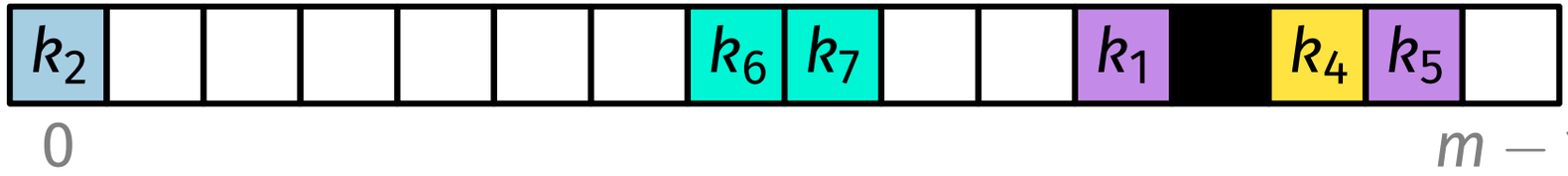
m bytes of

`0×FF`	= empty
`0×80`	= deleted 
`0×00`…`0×7F`	= occupied

use H_2 for the occupied bytes

\+ reprise of the first 16 bytes

Swisstable



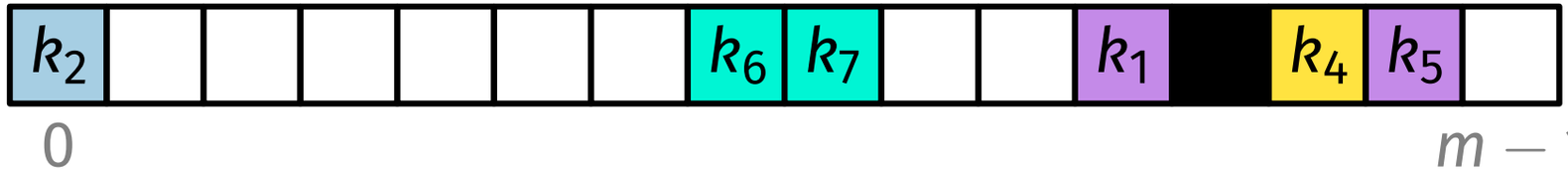
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

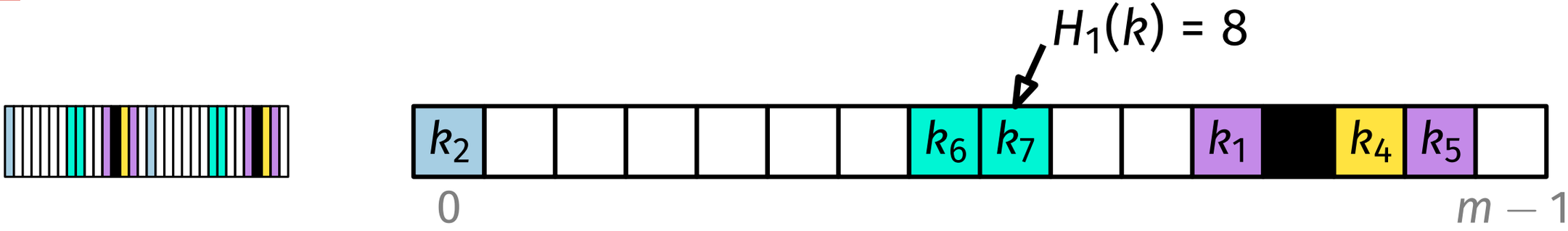
Steps:

1. start at bucket $H_1(k)$

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

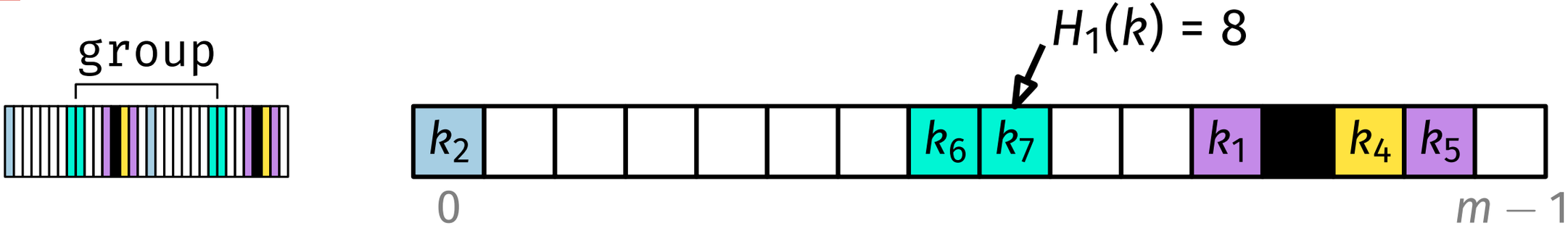
Steps:

1. start at bucket $H_1(k)$

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

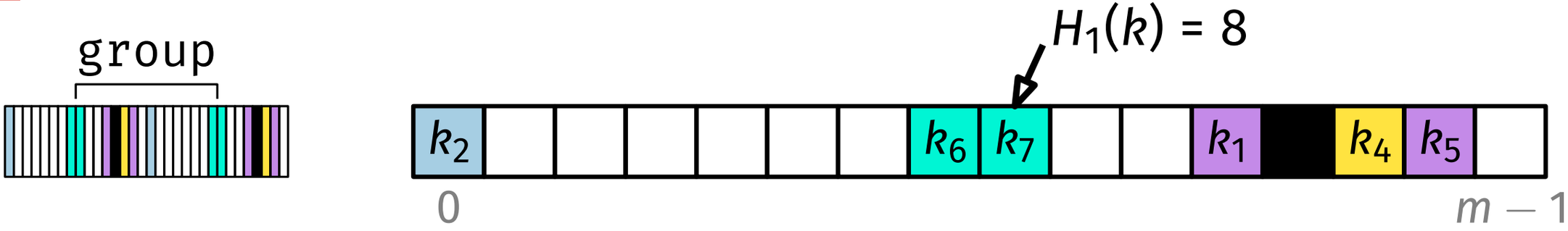
Steps:

1. start at bucket $H_1(k)$
2. search `group` for $H_2(k)$

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable

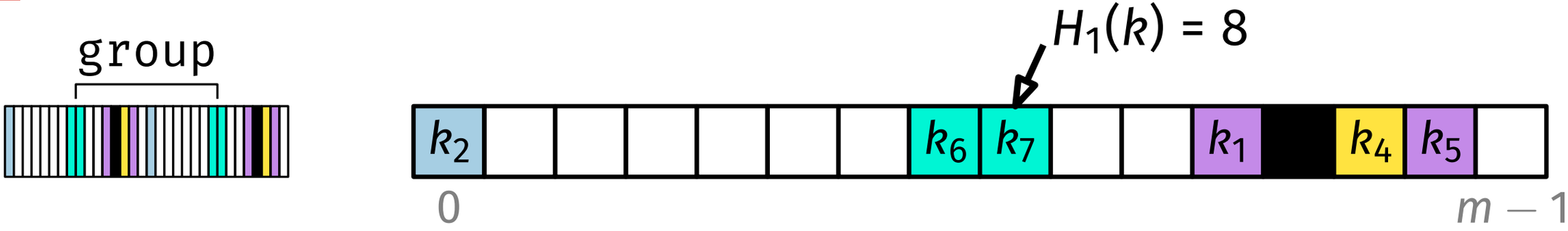


Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0x3F}$

Swisstable



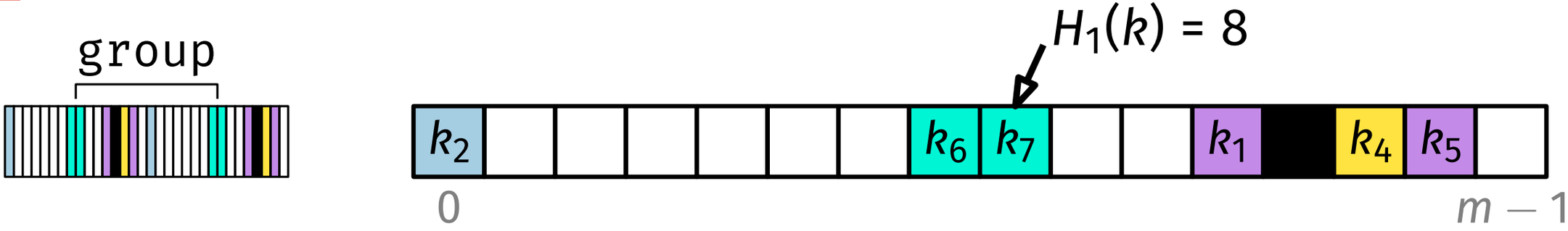
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0x3F}$

`_mm_set1_epi8`

Swisstable



Ingredients:

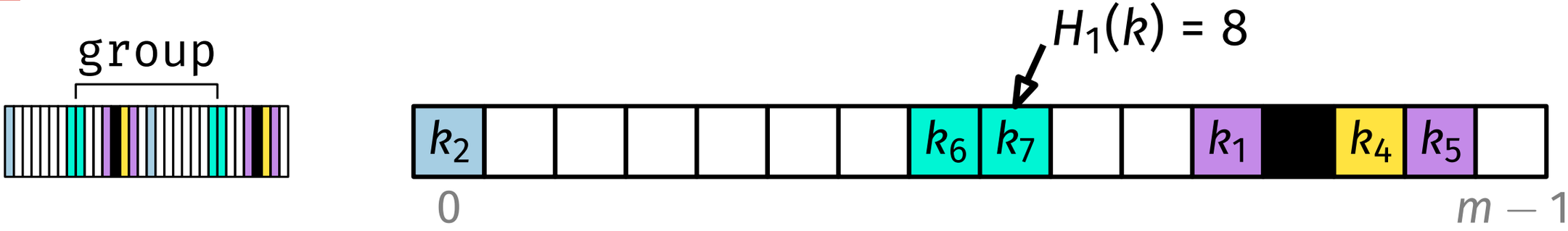
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0×3F}$

`_mm_set1_epi8`

`0×3F` $\mapsto$ | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F | 3F |

Swisstable



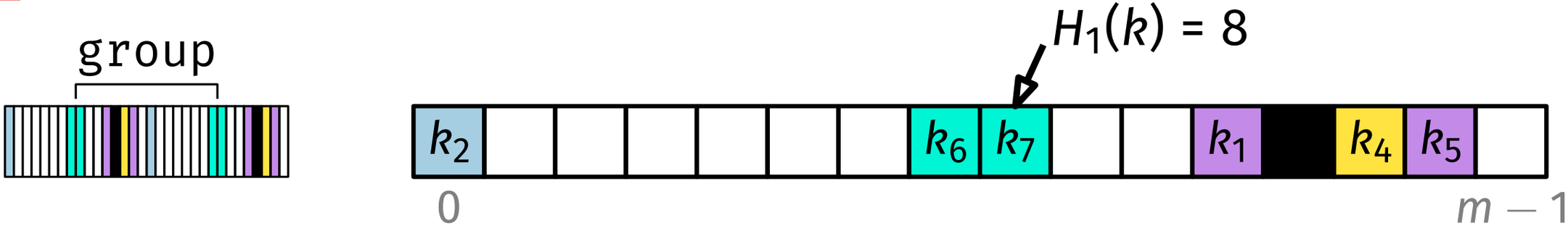
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0×3F}$

`_mm_cmpeq_epi8`

Swisstable



Ingredients:

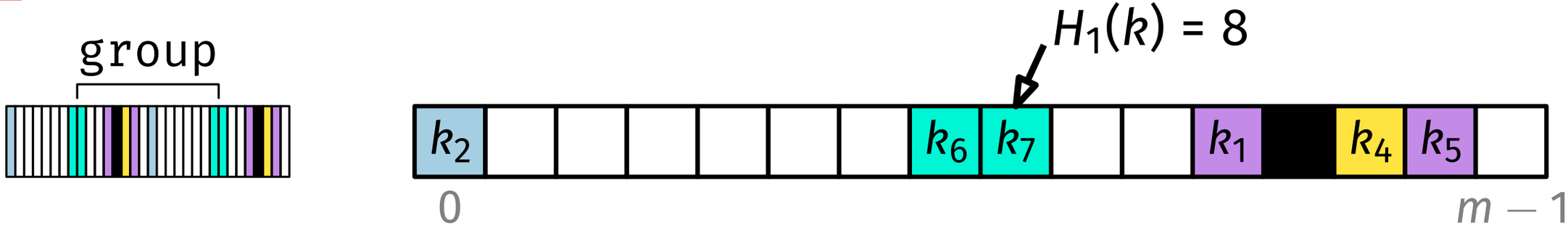
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0×3F}$

`_mm_cmpeq_epi8`

`group`	05	7B	3F	28	FF	40	6A	37	42	3F	FF	58	80	FF	36	00
`0×3F` $\mapsto$	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F

Swisstable



Ingredients:

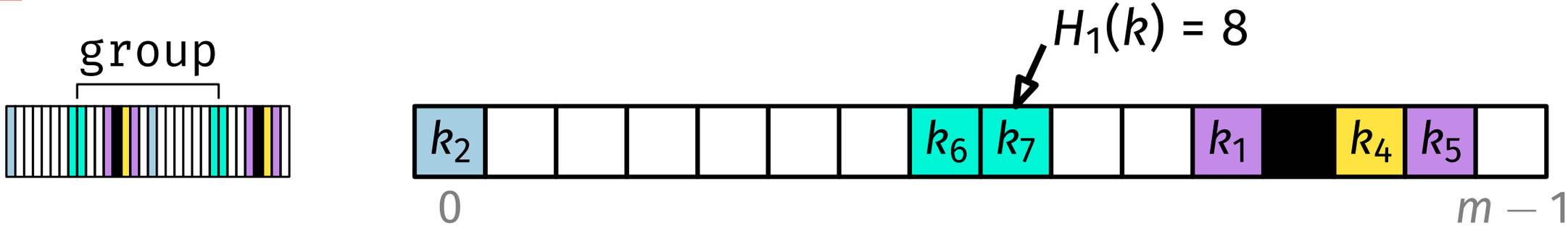
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0×3F}$

`_mm_cmpeq_epi8`

`group`	05	7B	3F	28	FF	40	6A	37	42	3F	FF	58	80	FF	36	00
`0×3F` $\mapsto$	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F
	00	00	FF	00	00	00	00	00	00	FF	00	00	00	00	00	00

Swisstable



Ingredients:

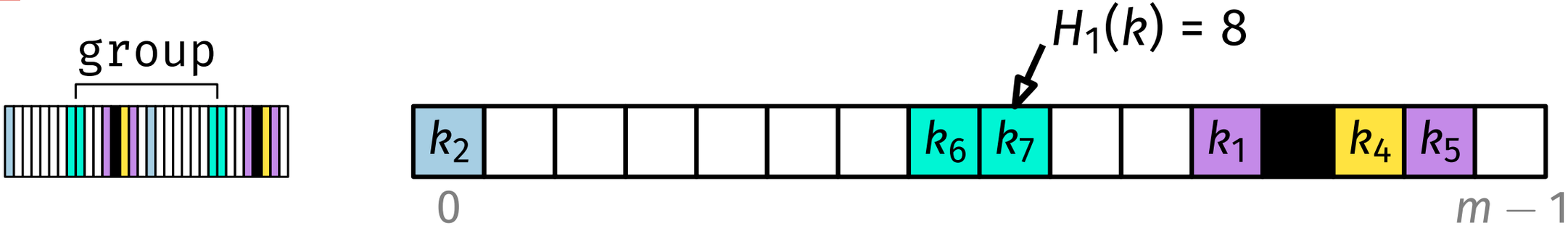
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = \texttt{0×3F}$

`_mm_movemask_epi8`

`group`	05	7B	3F	28	FF	40	6A	37	42	3F	FF	58	80	FF	36	00
`0×3F` $\mapsto$	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F
	00	00	FF	00	00	00	00	00	00	FF	00	00	00	00	00	00

Swisstable



Ingredients:

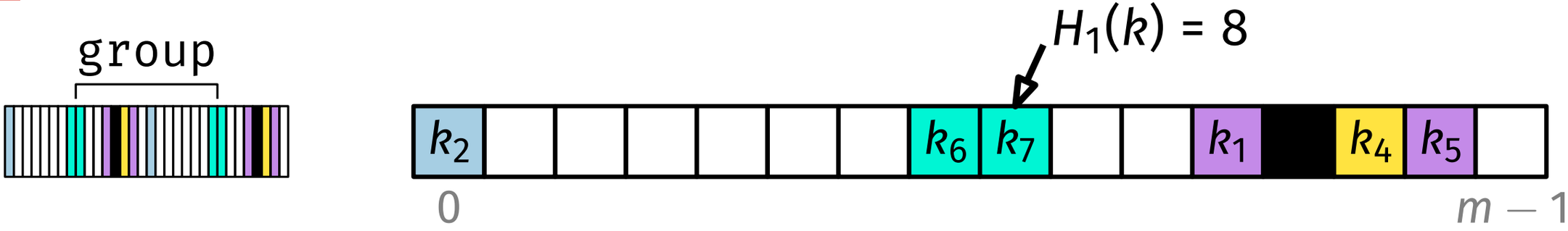
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Search `group` for k with $H_2(k) = 0\times3F$

`_mm_movemask_epi8`

`group`	05	7B	3F	28	FF	40	6A	37	42	3F	FF	58	80	FF	36	00
`0×3F` ↦	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F	3F
`0×2040` ↤	00	00	FF	00	00	00	00	00	00	FF	00	00	00	00	00	00

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

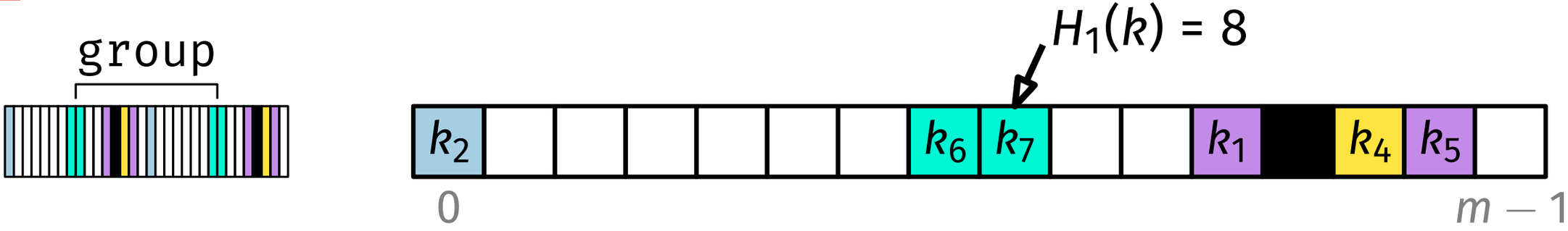
Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Steps:

1. start at bucket $H_1(k)$
2. search `group` for $H_2(k)$
3. for each match, check keys, return `true` if found

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

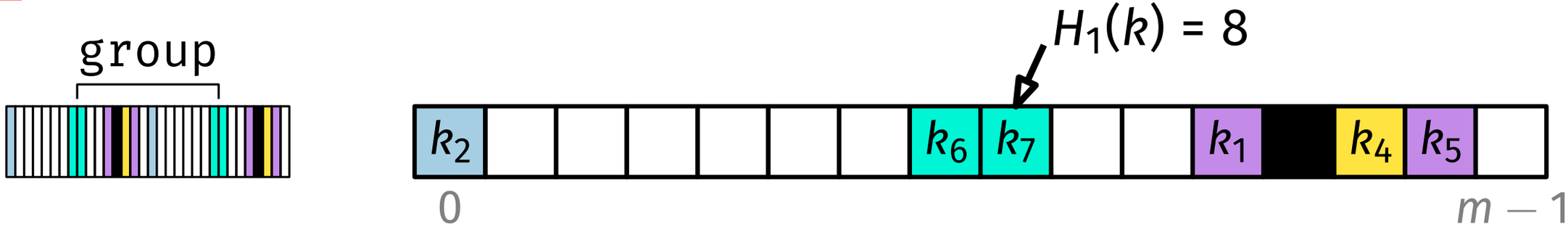
Operations:

- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)

Steps:

1. start at bucket $H_1(k)$
2. search `group` for $H_2(k)$
3. for each match, check keys, return `true` if found
4. search `group` for an empty bucket

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

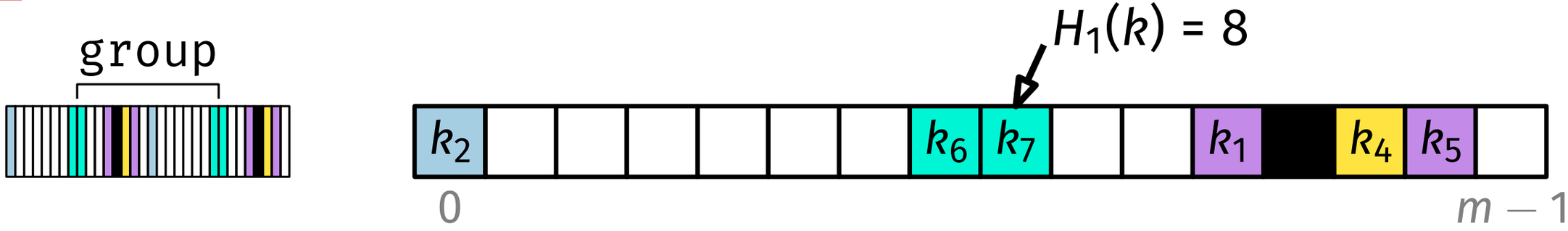
- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)

Steps:

1. start at bucket $H_1(k)$
2. search `group` for $H_2(k)$
3. for each match, check keys, return `true` if found
4. search `group` for an empty bucket

= *search* `group` *for* `0×FF`

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

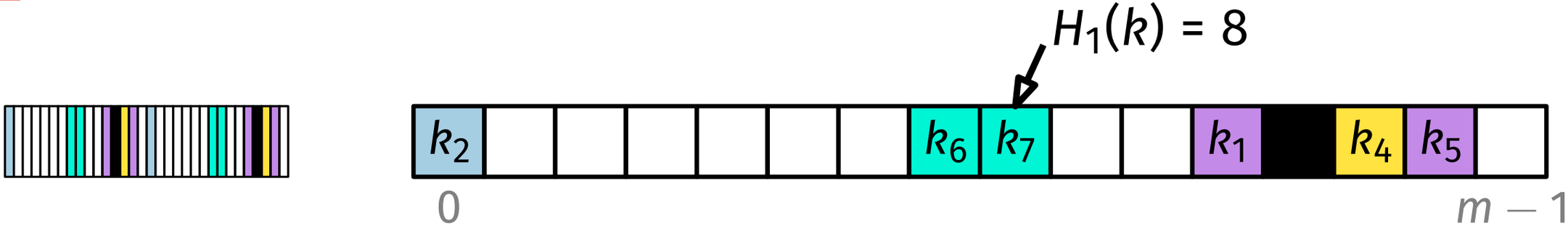
Operations:

- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)

Steps:

1. start at bucket $H_1(k)$
2. search `group` for $H_2(k)$
3. for each match, check keys, return `true` if found
4. search `group` for an empty bucket
5. return `false` if found
6. otherweise, check the next group

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

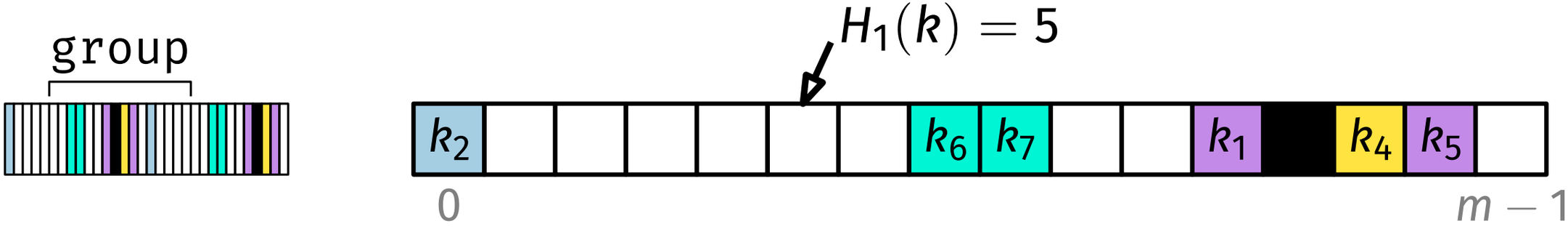
Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Case 1: k is in the table

find k, then replace it

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

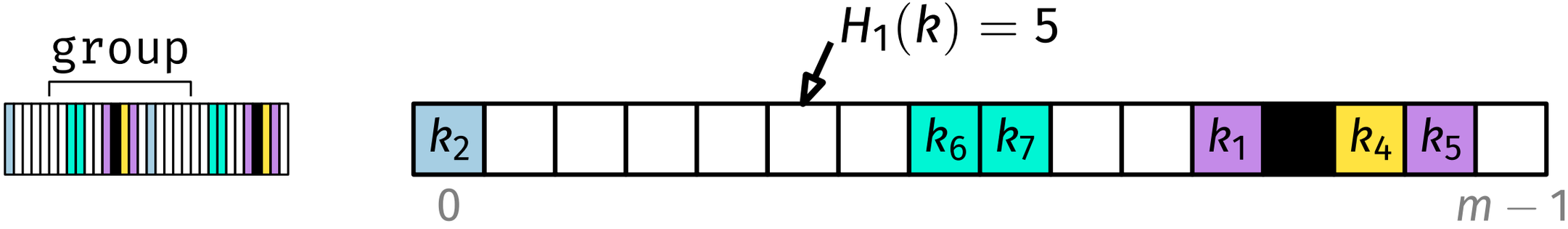
Case 1: k is in the table

find k, then replace it

Case 2: k is not in the table

1. start at bucket $H_1(k)$
2. search `group` for empty or deleted

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)

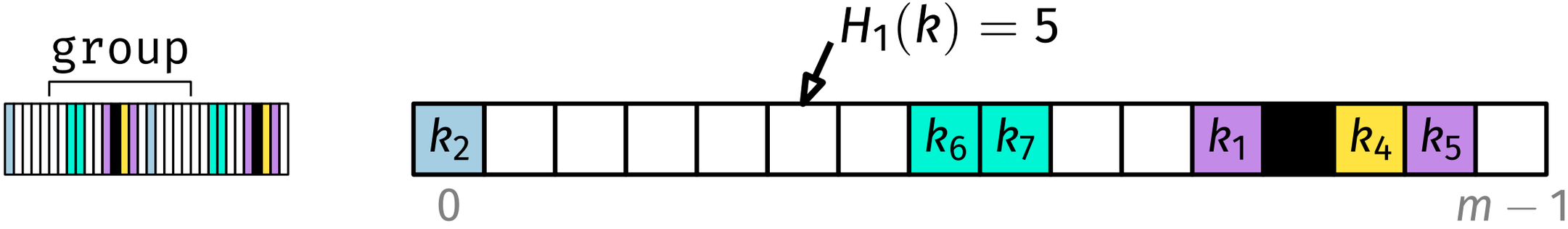
Case 1: k is in the table
find k, then replace it

Case 2: k is not in the table

1. start at bucket $H_1(k)$
2. search `group` for empty or deleted

= `movemask` *on* `group`

Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

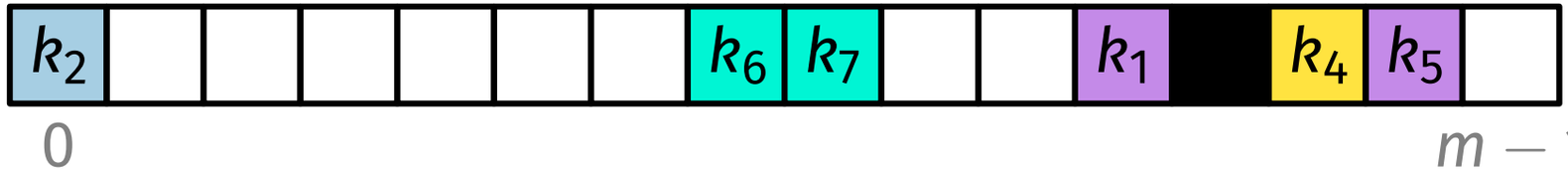
Case 1: k is in the table

find k, then replace it

Case 2: k is not in the table

1. start at bucket $H_1(k)$
2. search `group` for empty or deleted
3. if found, insert k
4. otherwise, check the next group

Swisstable



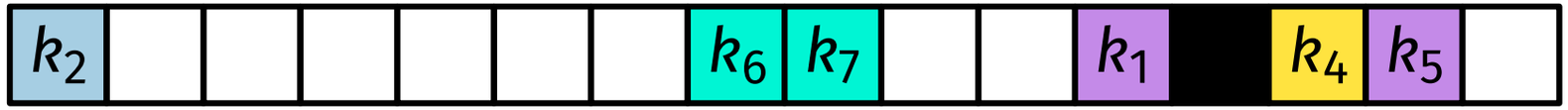
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable



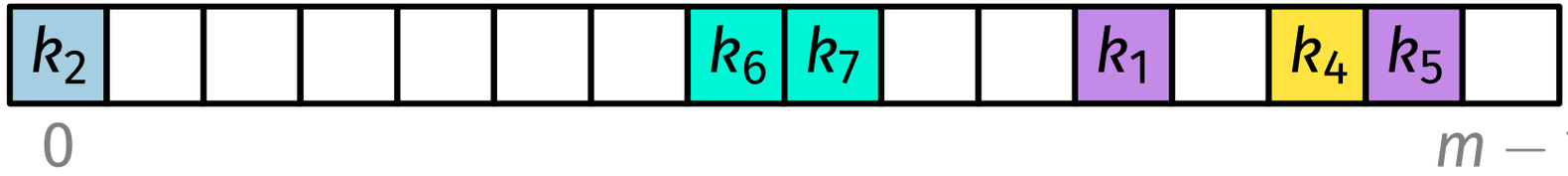
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable



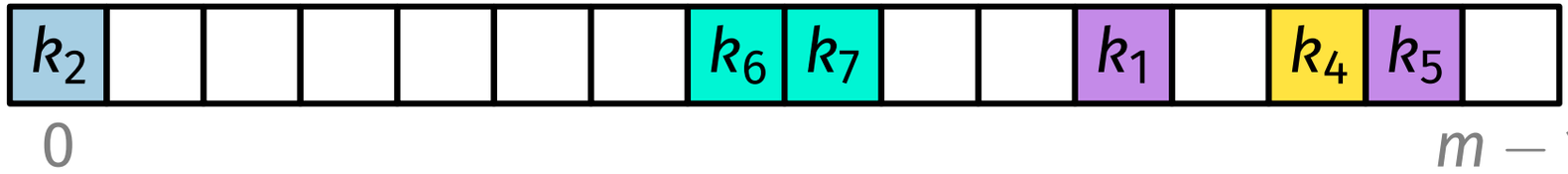
Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Swisstable

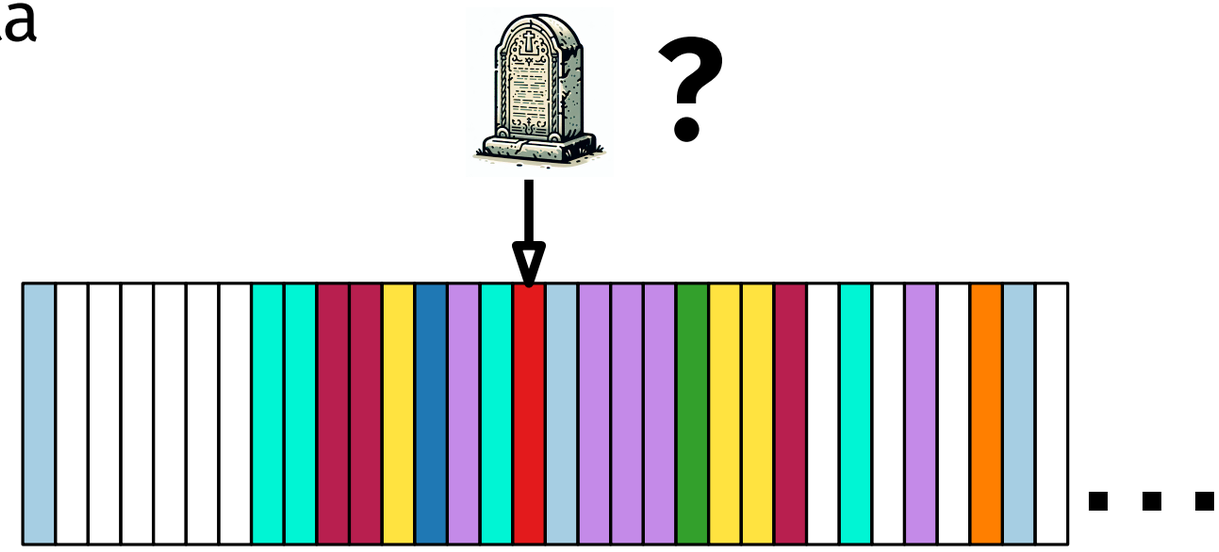


Ingredients:

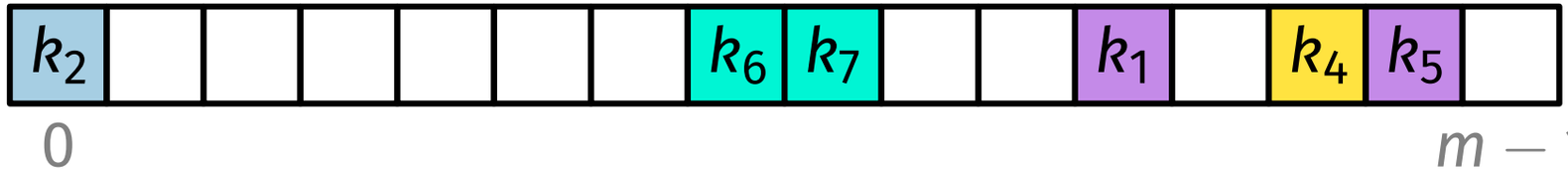
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)



Swisstable

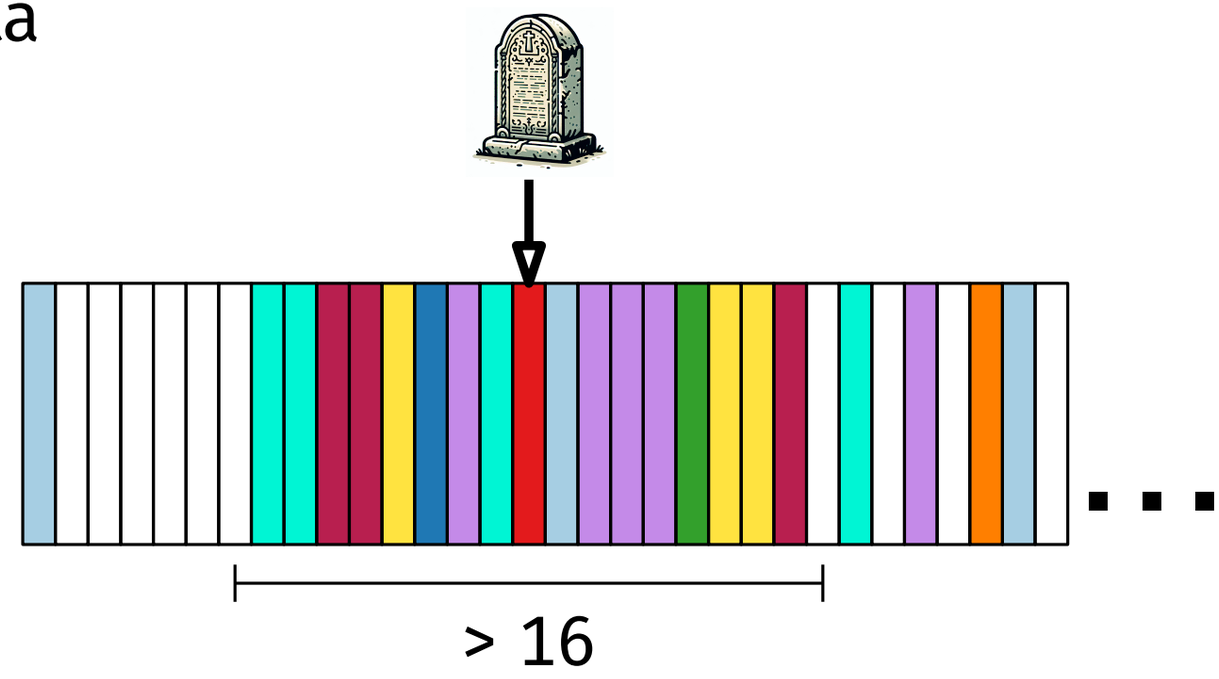


Ingredients:

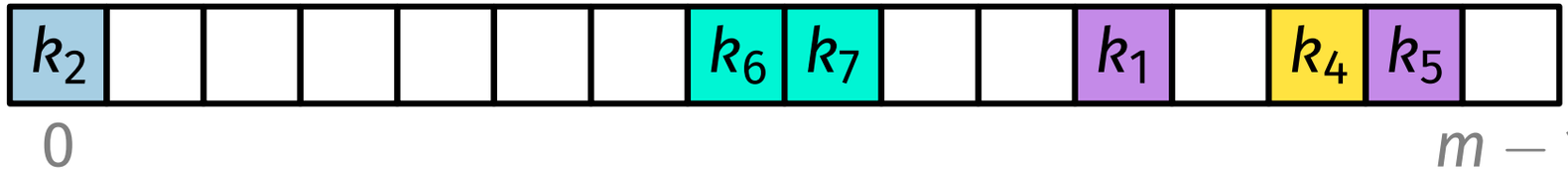
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)



Swisstable

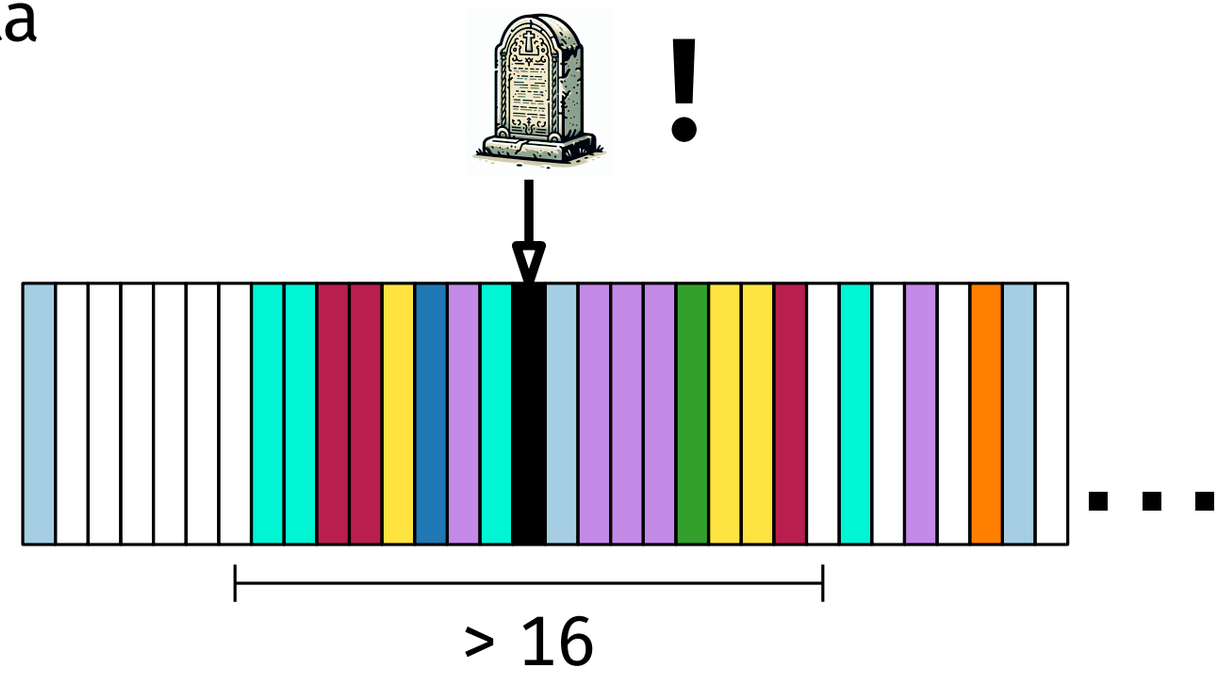


Ingredients:

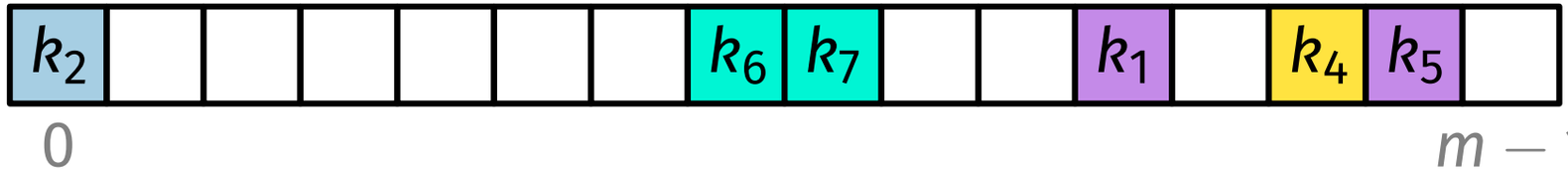
- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(*k*: Key)
- insert(*k*: Key)
- delete(*k*: Key)



Swisstable



Ingredients:

- derive two hash functions
- use $m + 16$ bytes of metadata
- SIMD instructions

Operations:

- find(k: Key)
- insert(k: Key)
- delete(k: Key)

Weaknesses

- requires SIMD instructions
- hash function must use its high bits

Further Reading – Part II

"Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step" by Matt Kulukundis

[https://youtu.be/ncHmEUmJZf4?si=grOfZfZklwi2FFhV]

"Swisstable, a Quick and Dirty Description" by Aria Beingessner

[https://faultlore.com/blah/hashbrown-tldr/]

Pedro Celis, Per-Ake Larson and James Ian Munro, "Robin hood hashing," *Foundations of Computer Science (SFCS)*, pp. 281–288, 1985, doi:10.1109/SFCS.1985.48.

Rasmus Pagh and Flemming Friche Rodler, "Cuckoo hashing," *Journal of Algorithms*, 51(2):281–288, 2004, doi:10.1016/j.jalgor.2003.12.002.