CS11212 - **Spring** 2022

Data Structures & Introduction to **Algorithms**

Data Structures

Hashing

Ibrahim Albluwi

Where are we?

Problem. Design a data structure that supports *search*, *insertion* and *deletion*
(without duplicates)

Where are we?

Problem. Design a data structure that supports *search*, *insertion* and *deletion* (without duplicates)

Candidate implementations.

	`insert(val)`	`remove(val)`	`contains(val)`
Unordered DLL	`O(n)`	`O(n)`	`O(n)`
Unordered SLL	`O(n)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(n)`	`O(n)`
Ordered SLL	`O(n)`	`O(n)`	`O(n)`
Unordered Array	`O(n)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(n)`	`O(log n)`
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`

Where are we?

Problem. Design a data structure that supports *search*, *insertion* and *deletion*
(without duplicates)

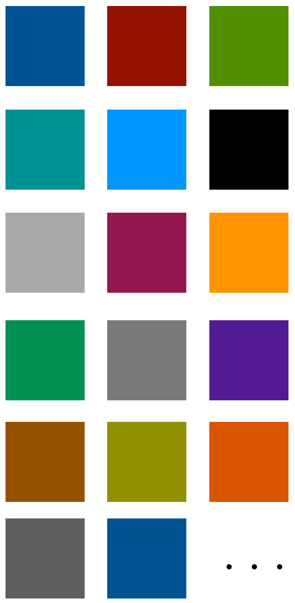
Candidate implementations.

	`insert(val)`	`remove(val)`	`contains(val)`
Unordered DLL	`O(n)`	`O(n)`	`O(n)`
Unordered SLL	`O(n)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(n)`	`O(n)`
Ordered SLL	`O(n)`	`O(n)`	`O(n)`
Unordered Array	`O(n)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(n)`	`O(log n)`
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`

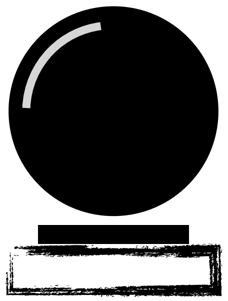
?

Can we do better?
Can we improve over the performance of balanced BSTs, such that *search*, *insertion* and/or *deletion* run(s) in $O(1)$?

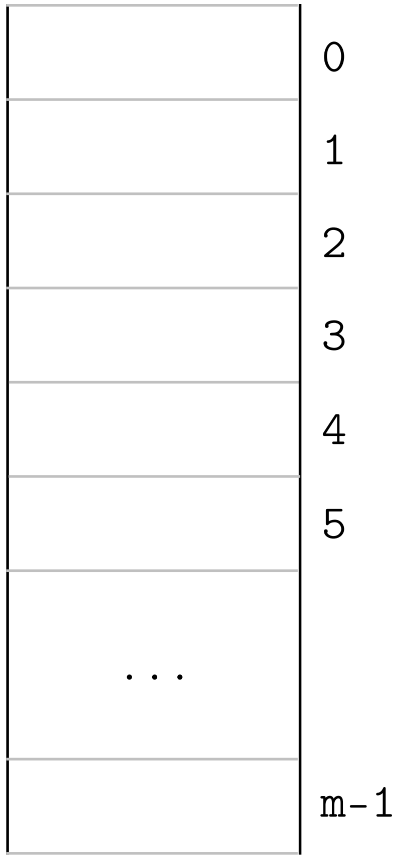
I have a dream!



n elements to be stored



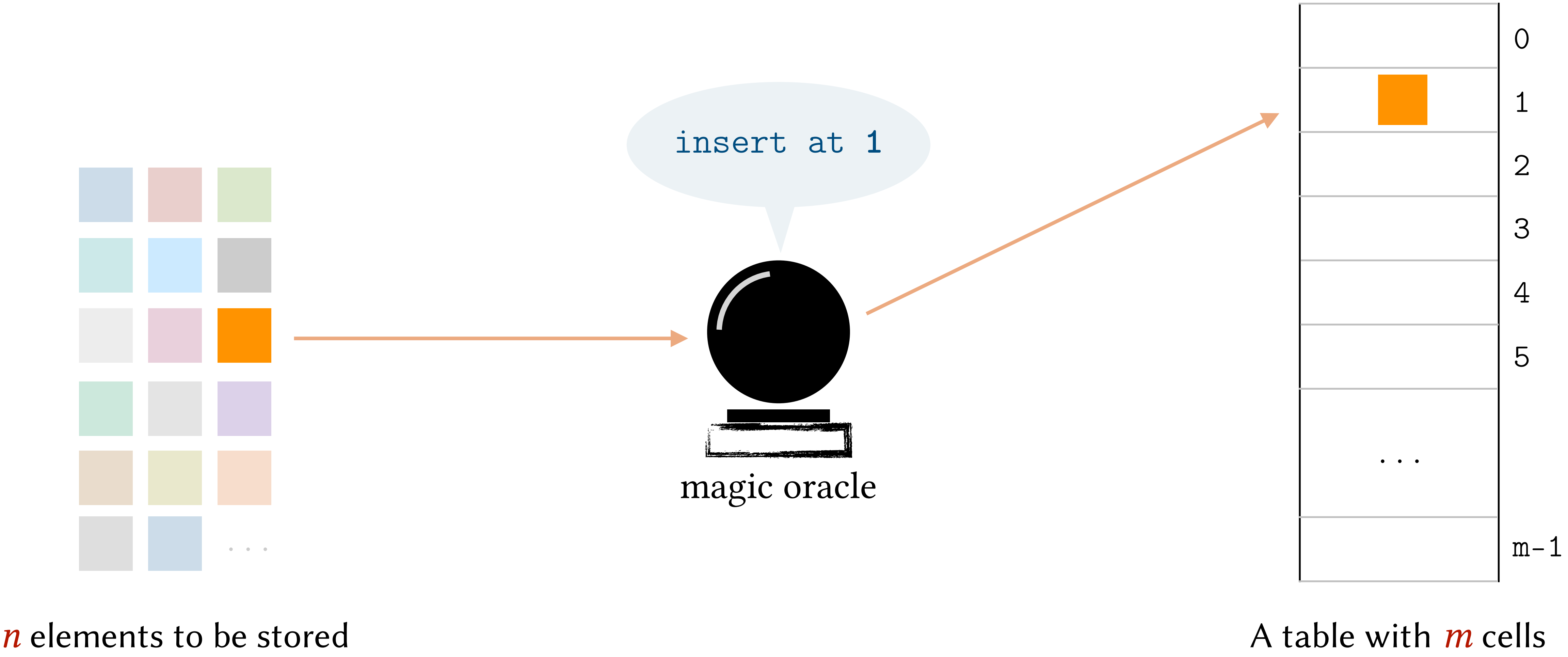
magic oracle



A table with *m* cells

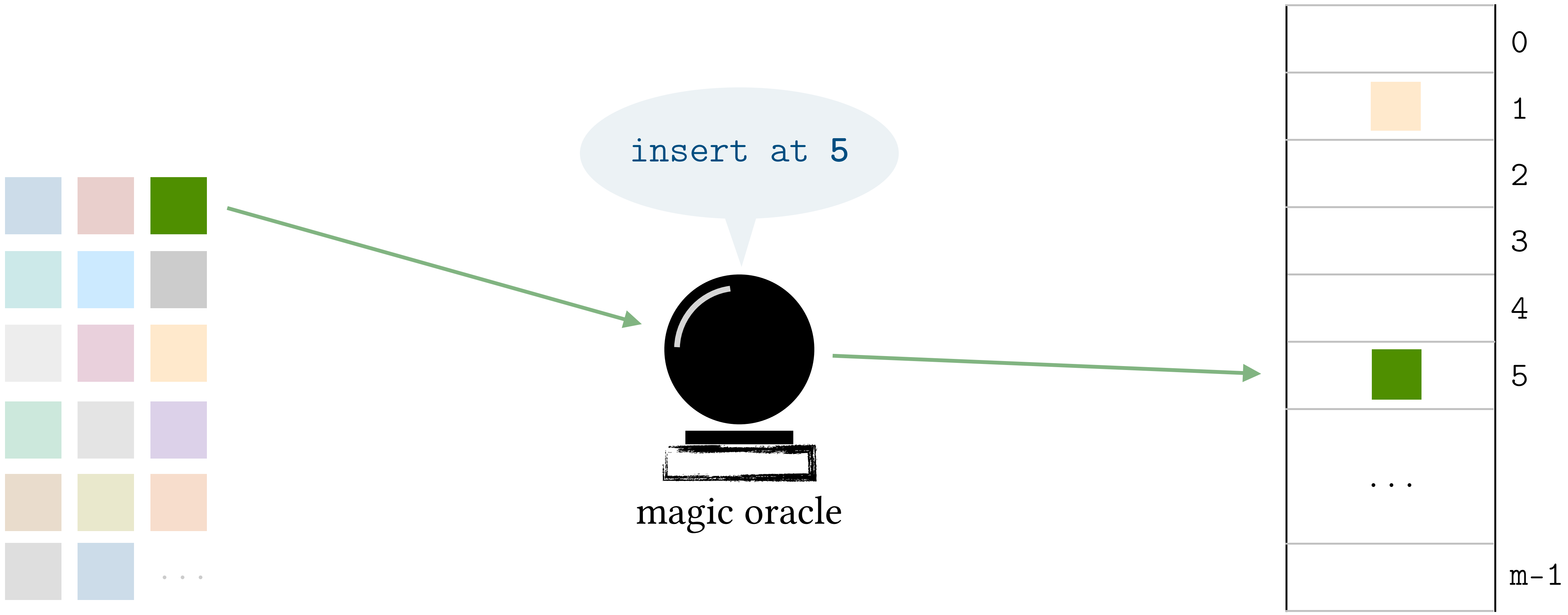
I Have a Dream: A *magic oracle* that knows exactly in which cell each element should be stored or could be found!

I have a dream!



Insertion: The oracle knows exactly which index each element should go to.

I have a dream!

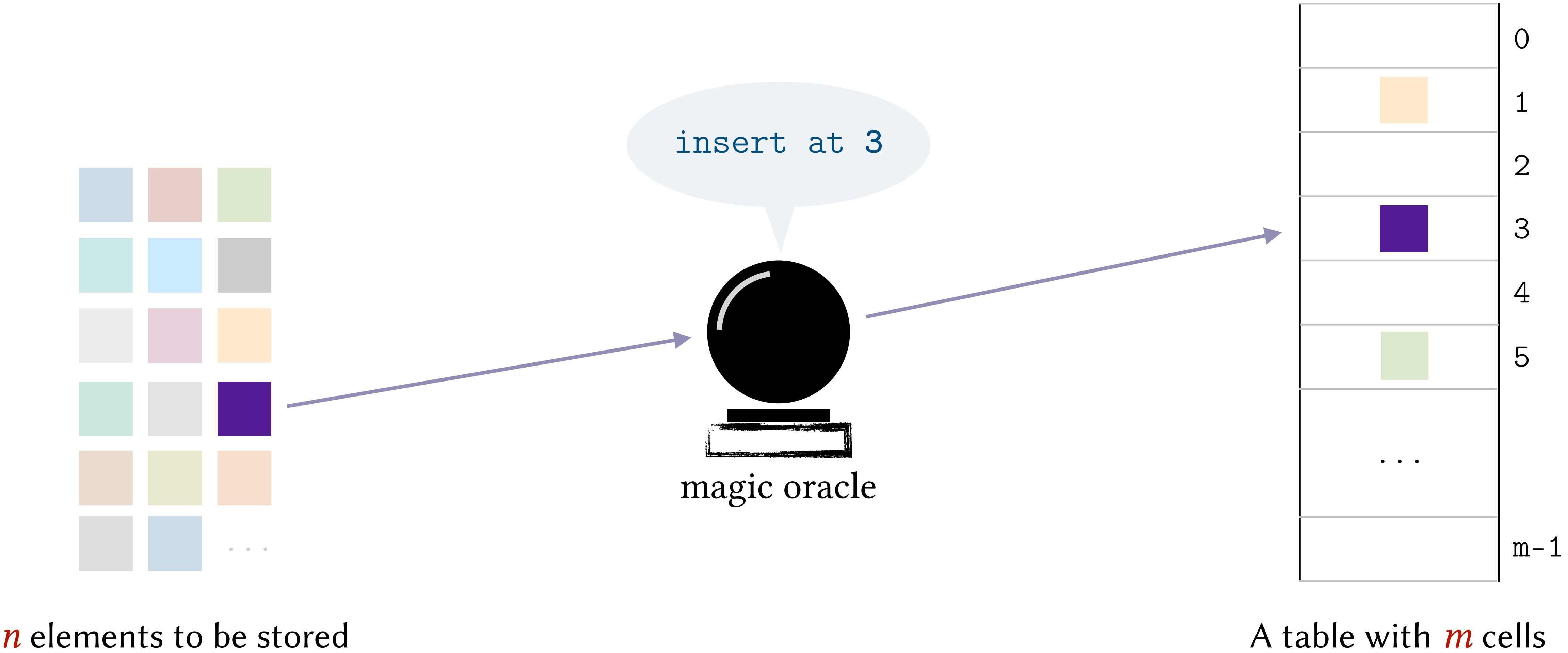


n elements to be stored

A table with *m* cells

Insertion: The oracle knows exactly which index each element should go to.

I have a dream!

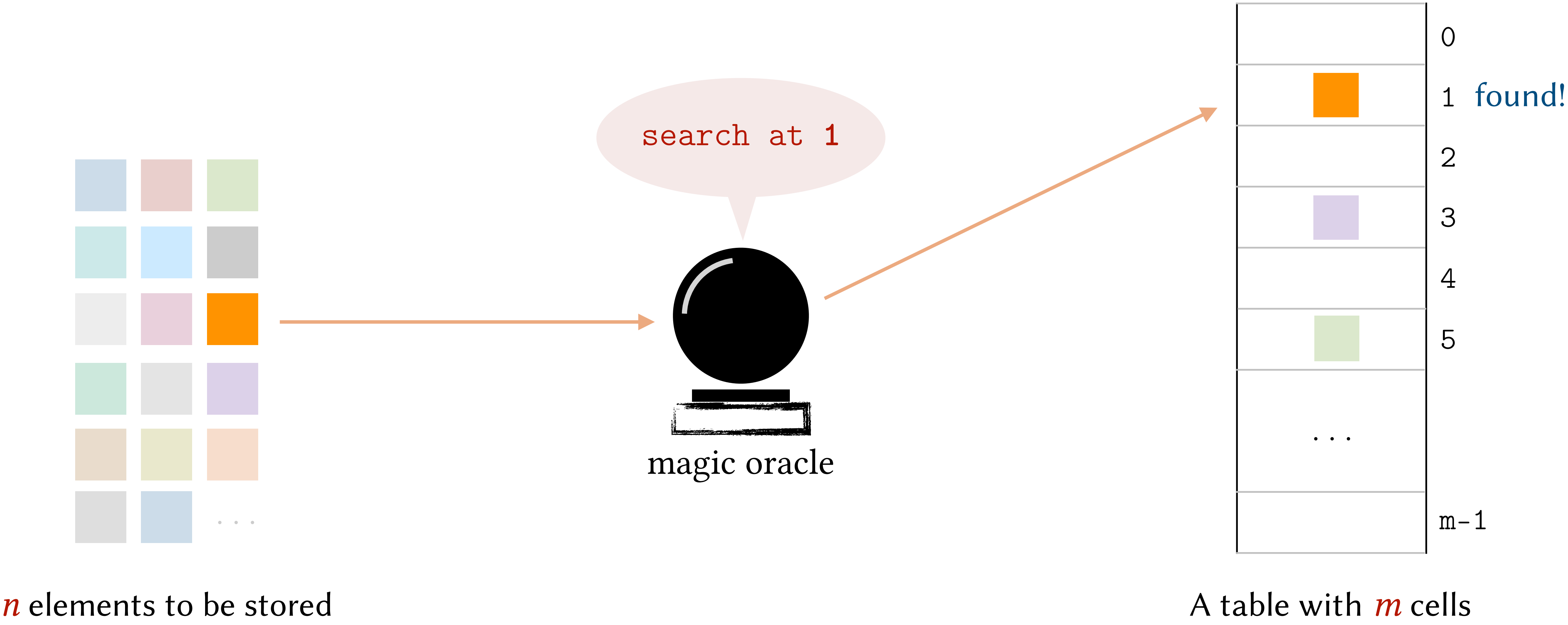


n elements to be stored

A table with m cells

Insertion: The oracle knows exactly which index each element should go to.

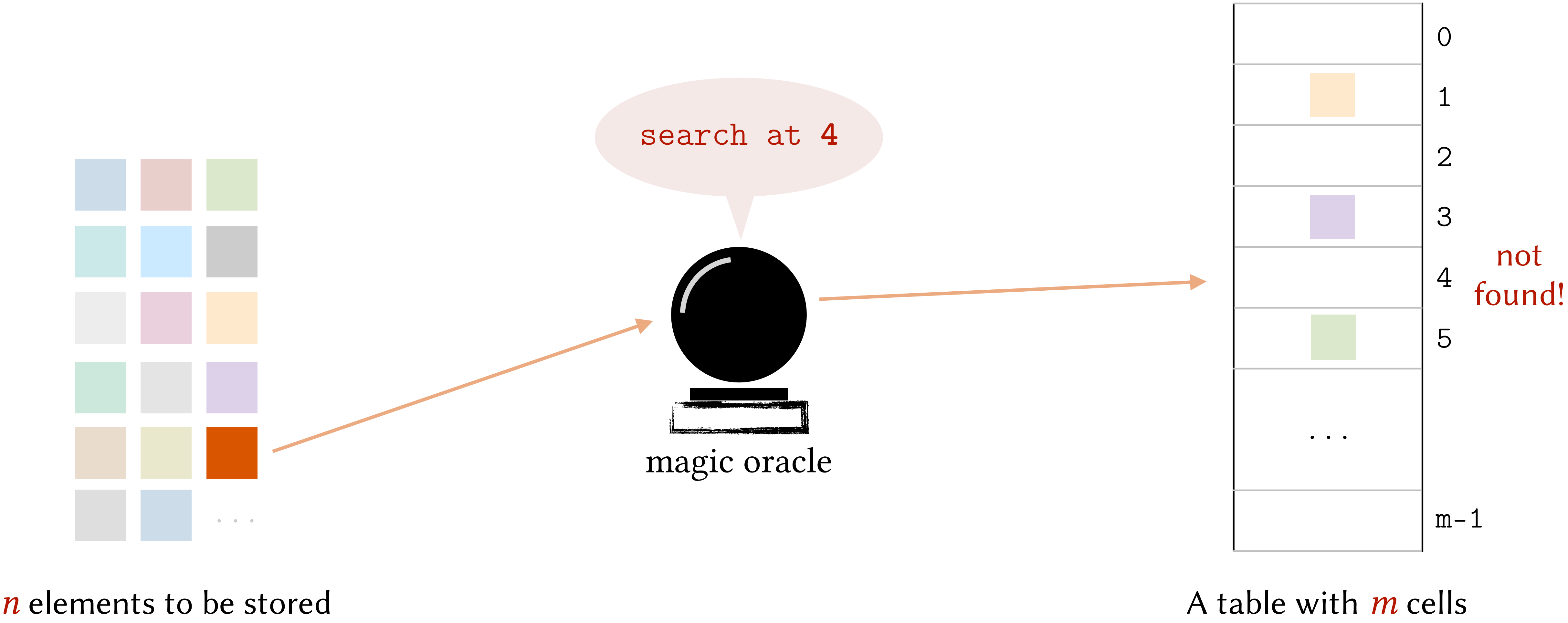
I have a dream!



Insertion: The oracle knows exactly which index each element should go to.

Search: The oracle knows exactly which index to search in.

I have a dream!

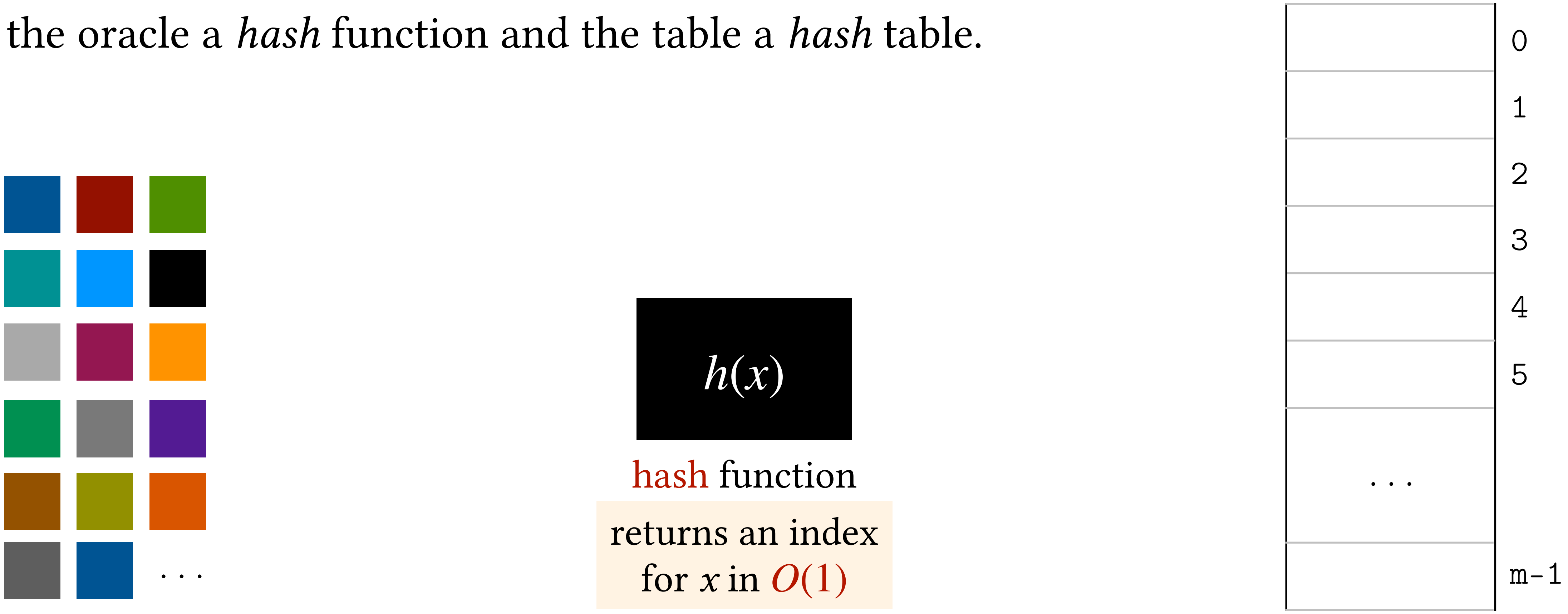


Insertion: The oracle knows exactly which index each element should go to.

Search: The oracle knows exactly which index to search in.

I have a dream!

Let's call the oracle a *hash* function and the table a *hash* table.



hash function

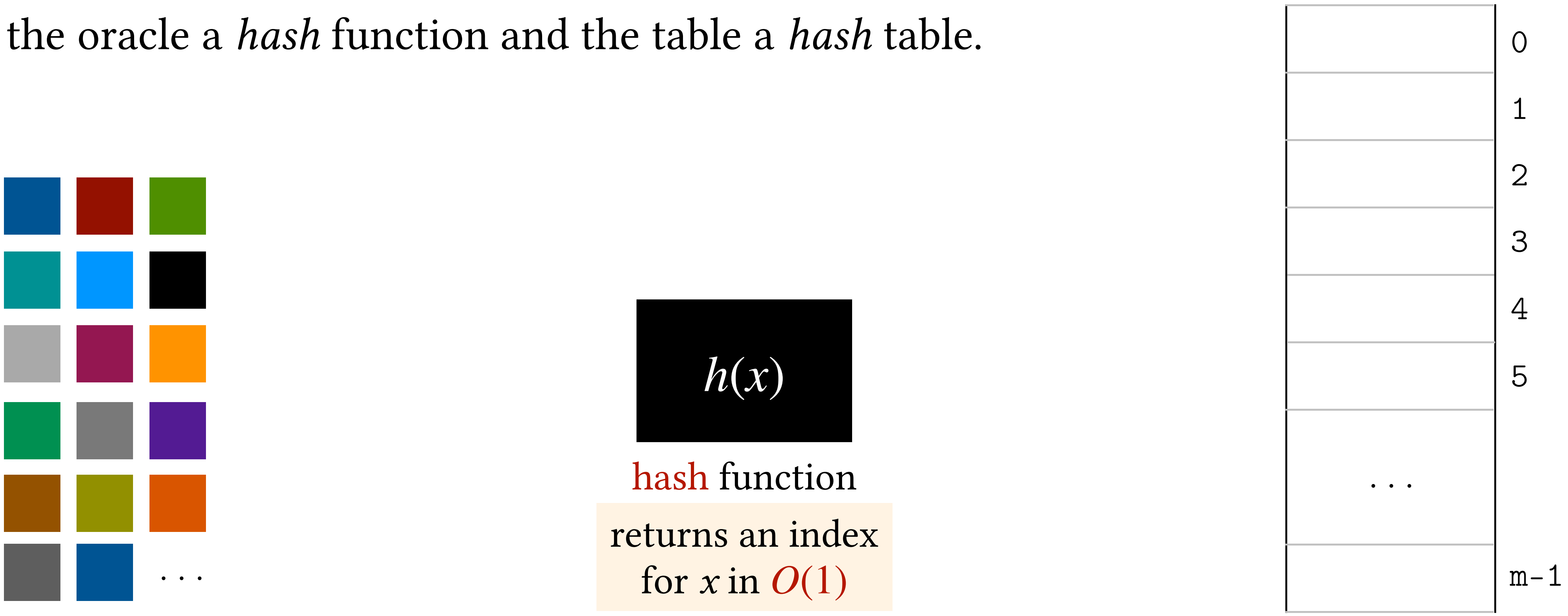
returns an index for x in $O(1)$

n elements to be stored

A hash table with m cells

I have a dream!

Let's call the oracle a *hash* function and the table a *hash* table.



n elements to be stored

hash function

returns an index for x in $O(1)$

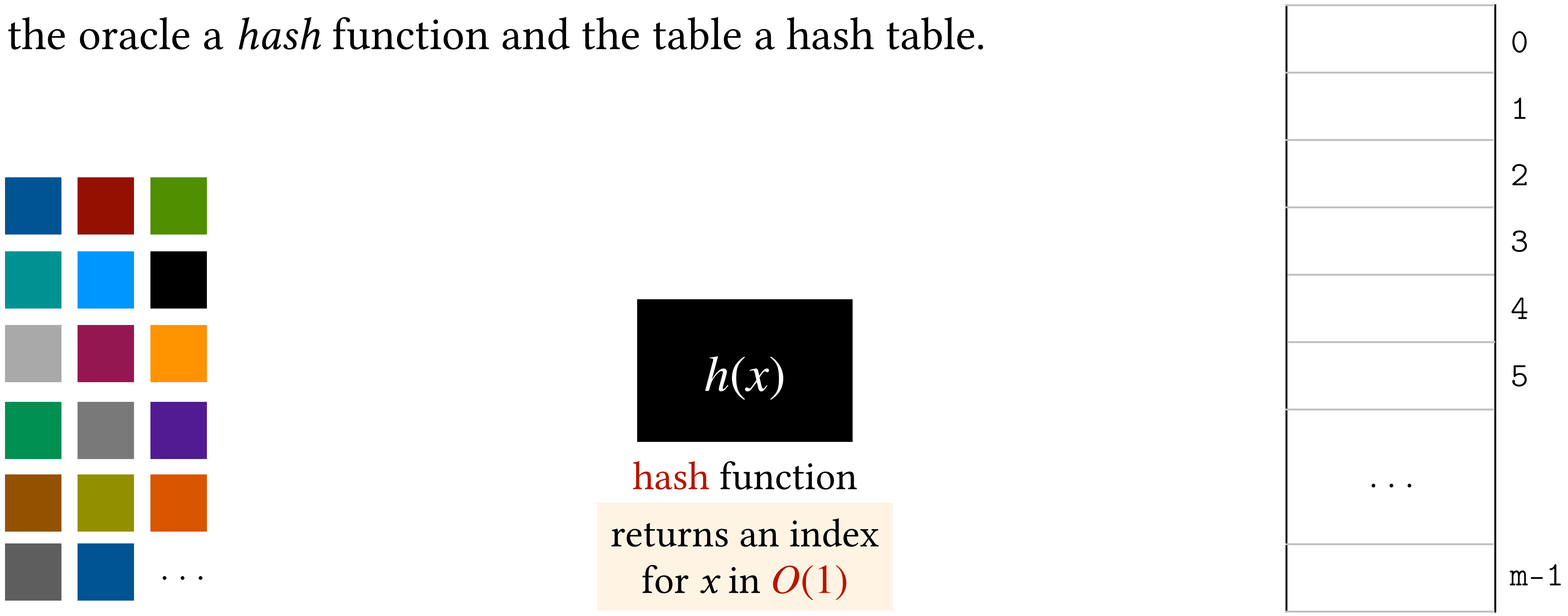
A hash table with m cells

The implementation is simple:

```
insert(x) : table[h(x)] = x
remove(x) : table[h(x)] = dummy value
search(x) : return table[h(x)] != dummy value
```

I have a dream!

Let's call the oracle a *hash* function and the table a hash table.



hash function

returns an index for x in $O(1)$

n elements to be stored

A hash table with m cells

The implementation is simple:

```
insert(x) : table[h(x)] = x
remove(x) : table[h(x)] = dummy value
search(x) : return table[h(x)] != dummy value
```

All operations are done in $O(1)$!

🤔 Is this possible?

Dream Comes True?

Consider n distinct non-negative integers all in the range $[0,\ 10^9]$.
How can we support *search*, *insert* and *remove* in $O(1)$?

```
555591  887
72  98666  0
4335   1342
119  233  5
11  9999994
847  9   ...
```

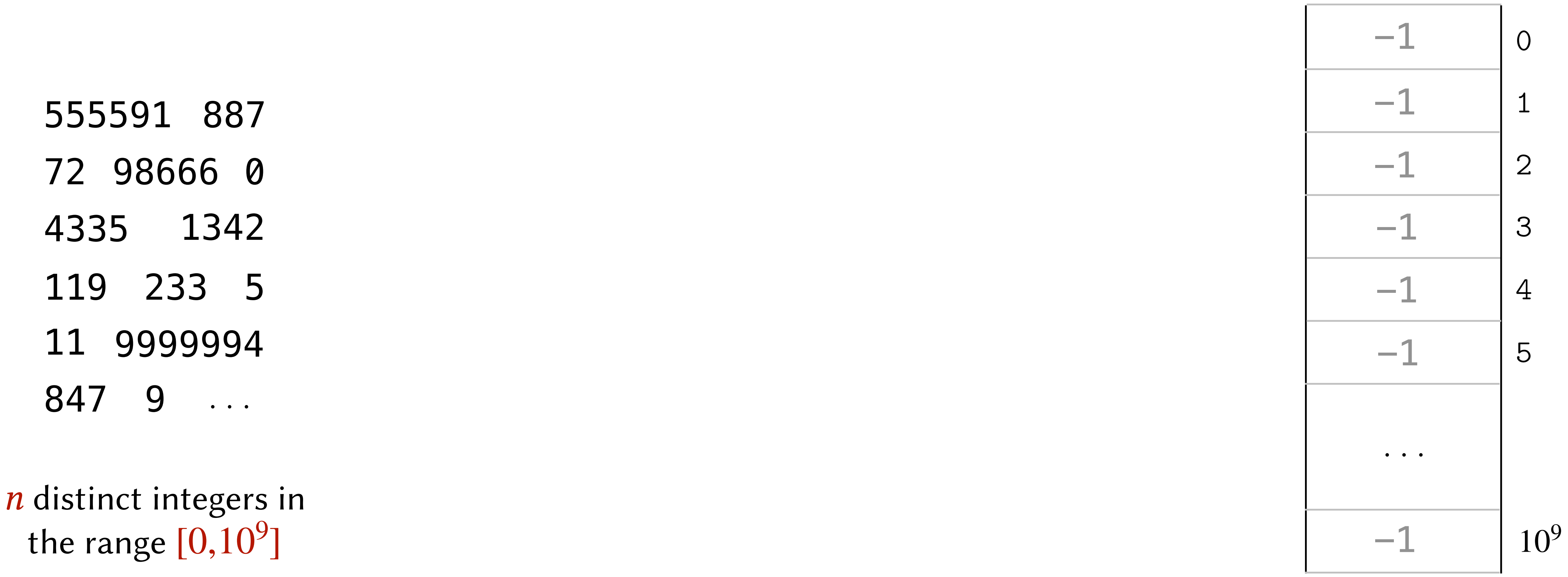
n distinct integers in
the range $[0,10^9]$

Dream Comes True?

Consider n distinct non-negative integers all in the range $[0,\ 10^9]$.
How can we support *search*, *insert* and *remove* in $O(1)$?

Answer.

1. Create a hash table of size $10^9 + 1$ (indices are from 0 to 10^9).
2. Use -1 as a dummy value in empty cells.



```
555591  887
72  98666  0
4335   1342
119  233  5
11  9999994
847  9  . . .
```

n distinct integers in the range $[0,10^9]$

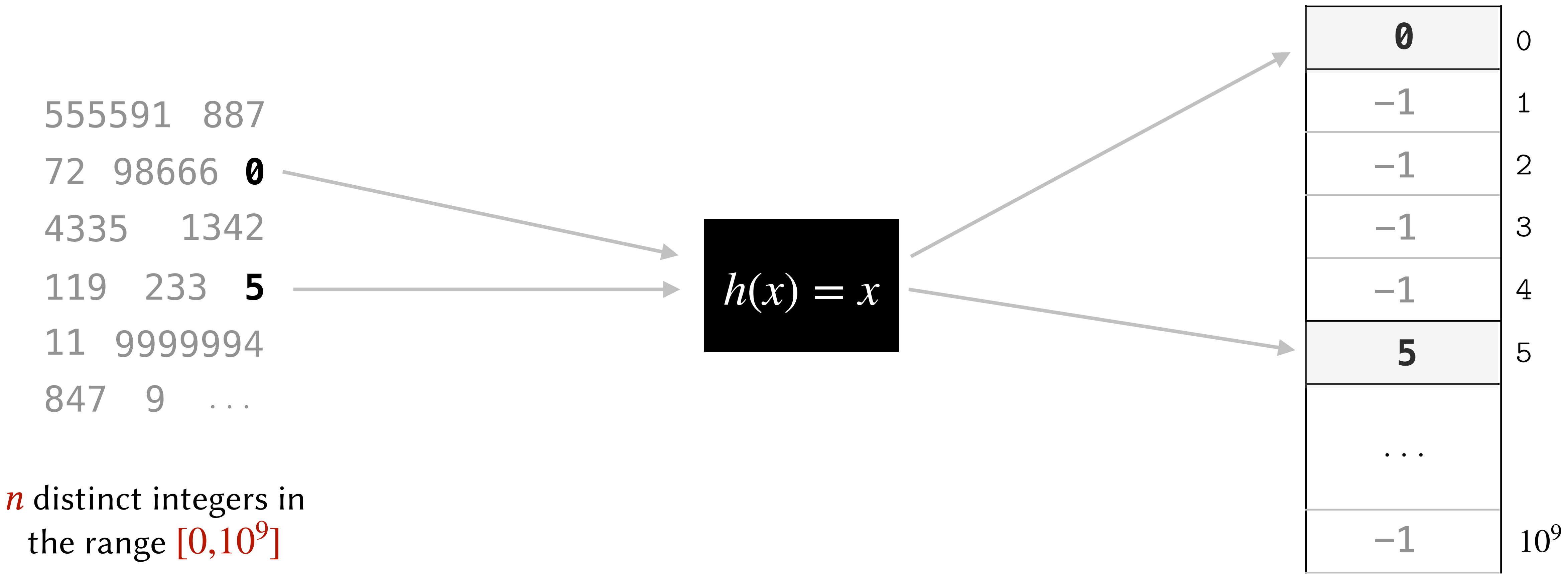
-1	0
-1	1
-1	2
-1	3
-1	4
-1	5
$\dots$	
-1	10^9

Dream Comes True?

Consider n distinct non-negative integers all in the range $[0,\ 10^9]$.
How can we support *search*, *insert* and *remove* in $O(1)$?

Answer.

1. Create a hash table of size $10^9 + 1$ (indices are from 0 to 10^9).
2. Use -1 as a dummy value in empty cells.
3. Use the following hash function: $h(x) = x$.
   i.e. 0 goes to index 0, 1 to index 1, 2 to index 2, etc.

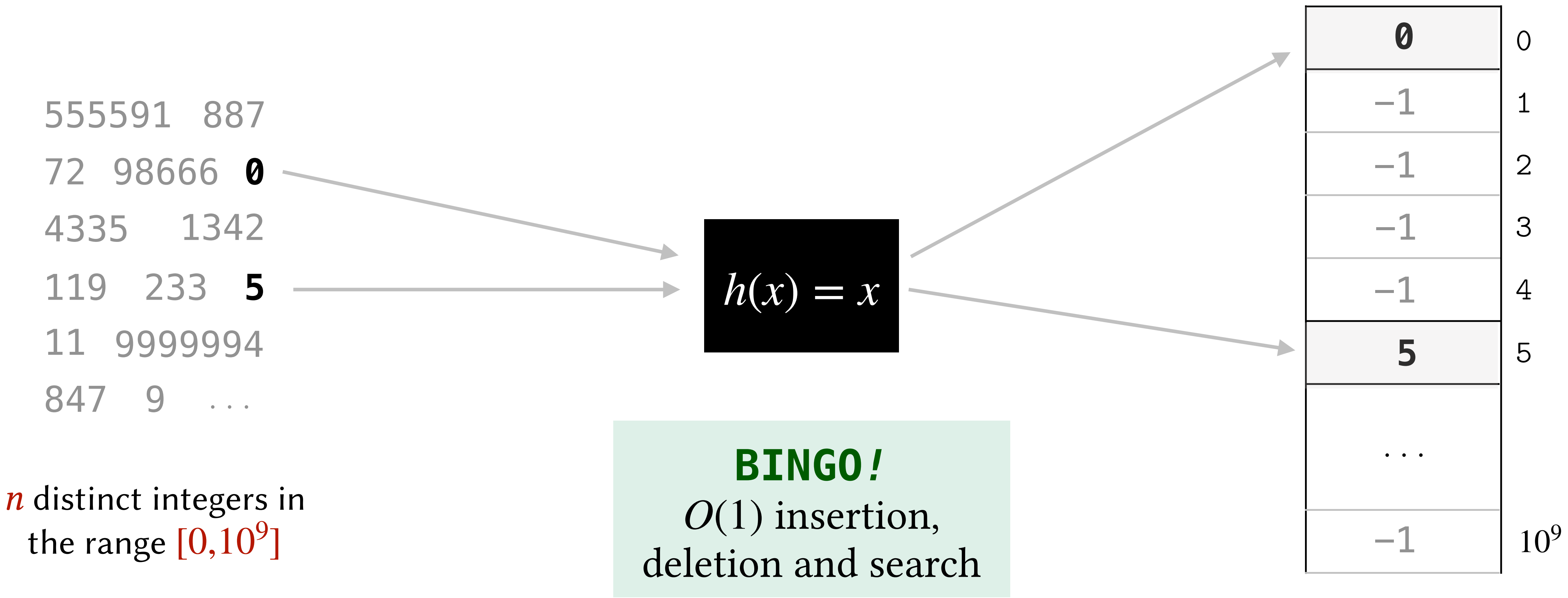


Dream Comes True?

Consider n distinct non-negative integers all in the range $[0,\ 10^9]$.
How can we support *search*, *insert* and *remove* in $O(1)$?

Answer.

1. Create a hash table of size $10^9 + 1$ (indices are from 0 to 10^9).
2. Use -1 as a dummy value in empty cells.
3. Use the following hash function: $h(x) = x$.
   i.e. 0 goes to index 0, 1 to index 1, 2 to index 2, etc.



n distinct integers in the range $[0,10^9]$

BINGO!
$O(1)$ insertion, deletion and search

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if $h(x_1) = h(x_2)$ implies $x_1 = x_2$

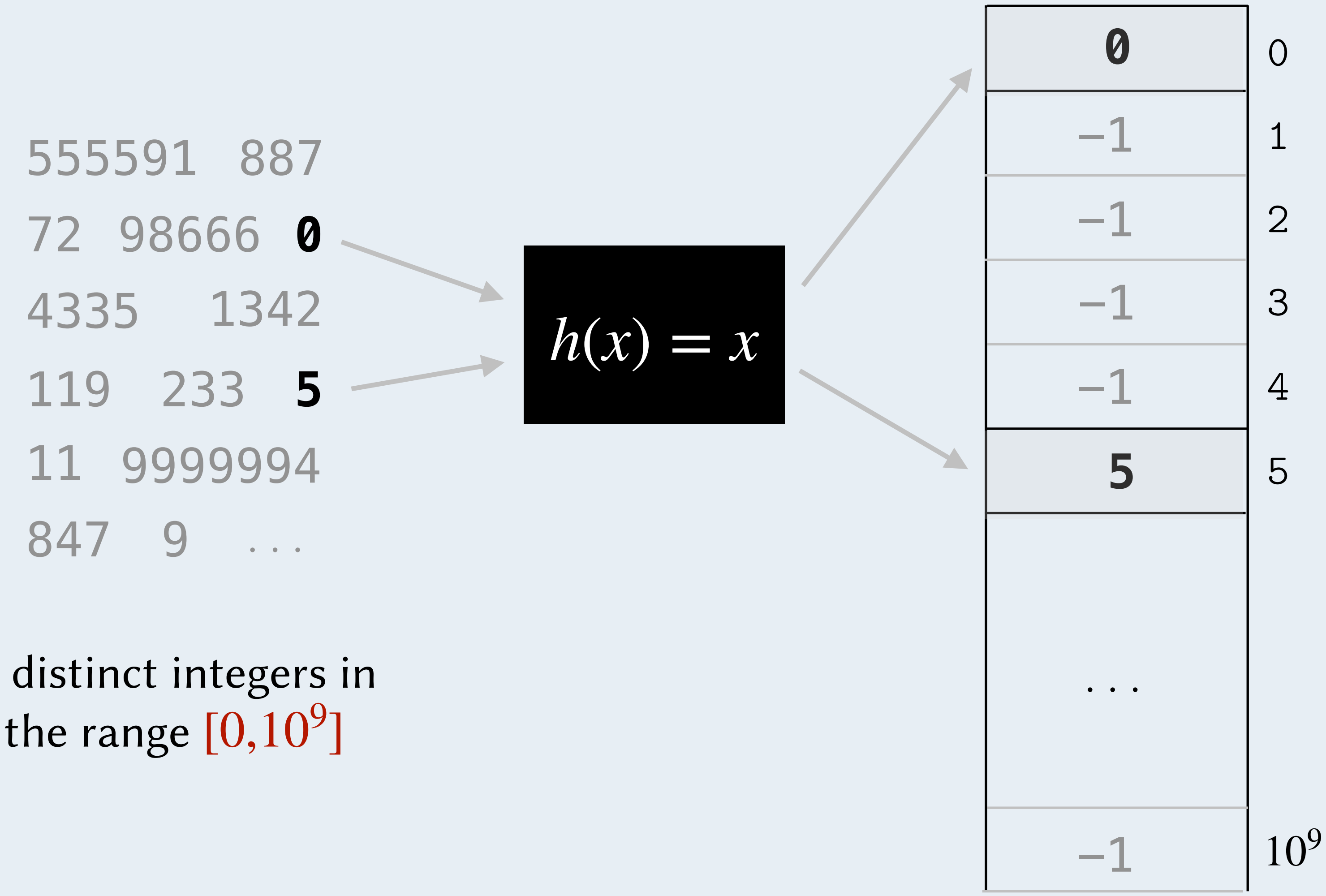
In other words, if $h(x)$ is *perfect,* no two distinct elements have the same hash value.

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if
$h(x_1) = h(x_2)$ implies $x_1 = x_2$

In other words, if $h(x)$ is *perfect,* no two distinct elements have the same hash value.

Example. $h(x) = x$ is a *perfect* hash function.



n distinct integers in the range $[0,10^9]$

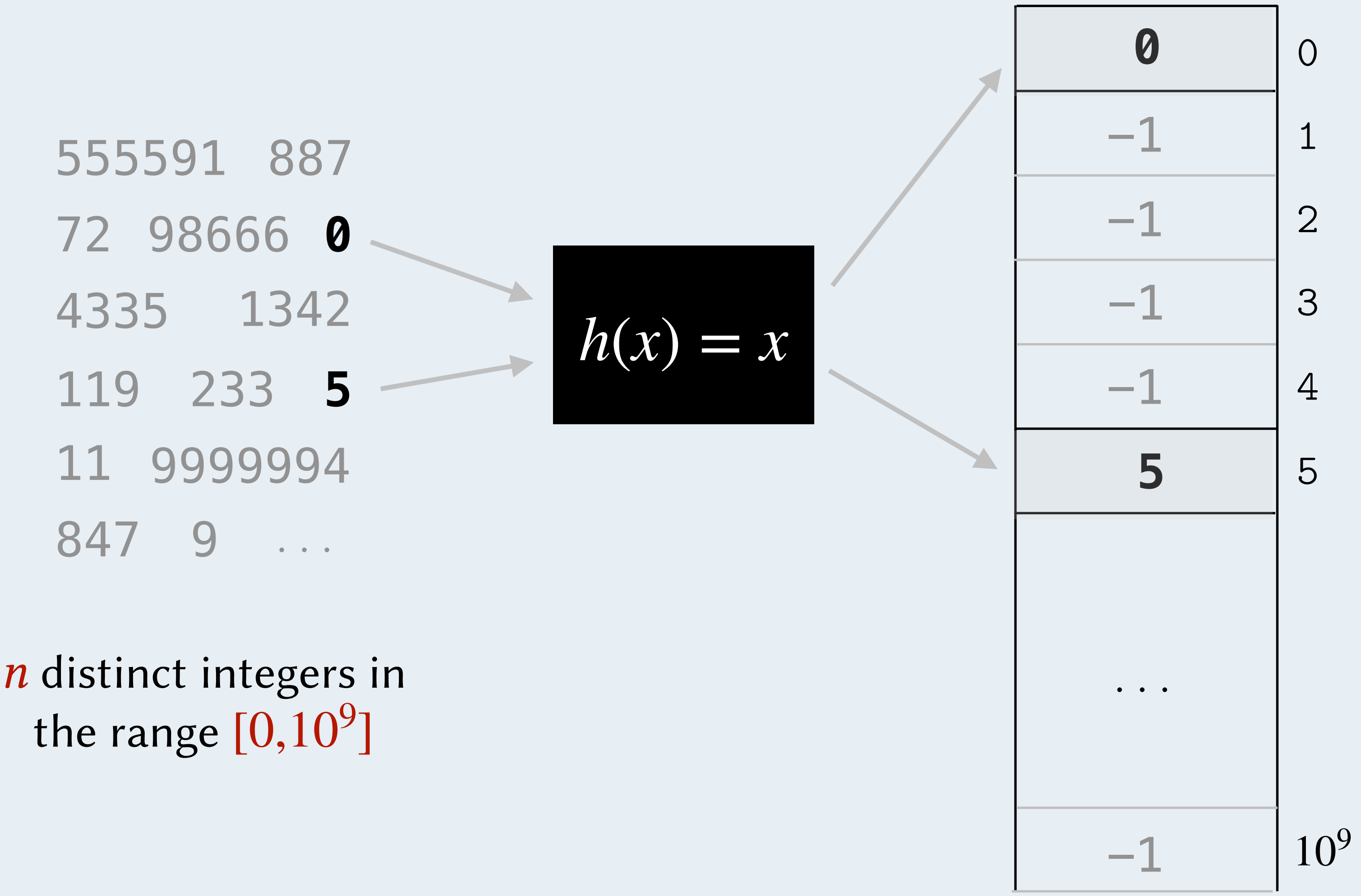
a table fo size $m = 10^9 + 1$

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if $h(x_1) = h(x_2)$ implies $x_1 = x_2$

In other words, if $h(x)$ is *perfect*, no two distinct elements have the same hash value.

Example. $h(x) = x$ is a *perfect* hash function.



n distinct integers in the range $[0,10^9]$

a table fo size $m = 10^9 + 1$

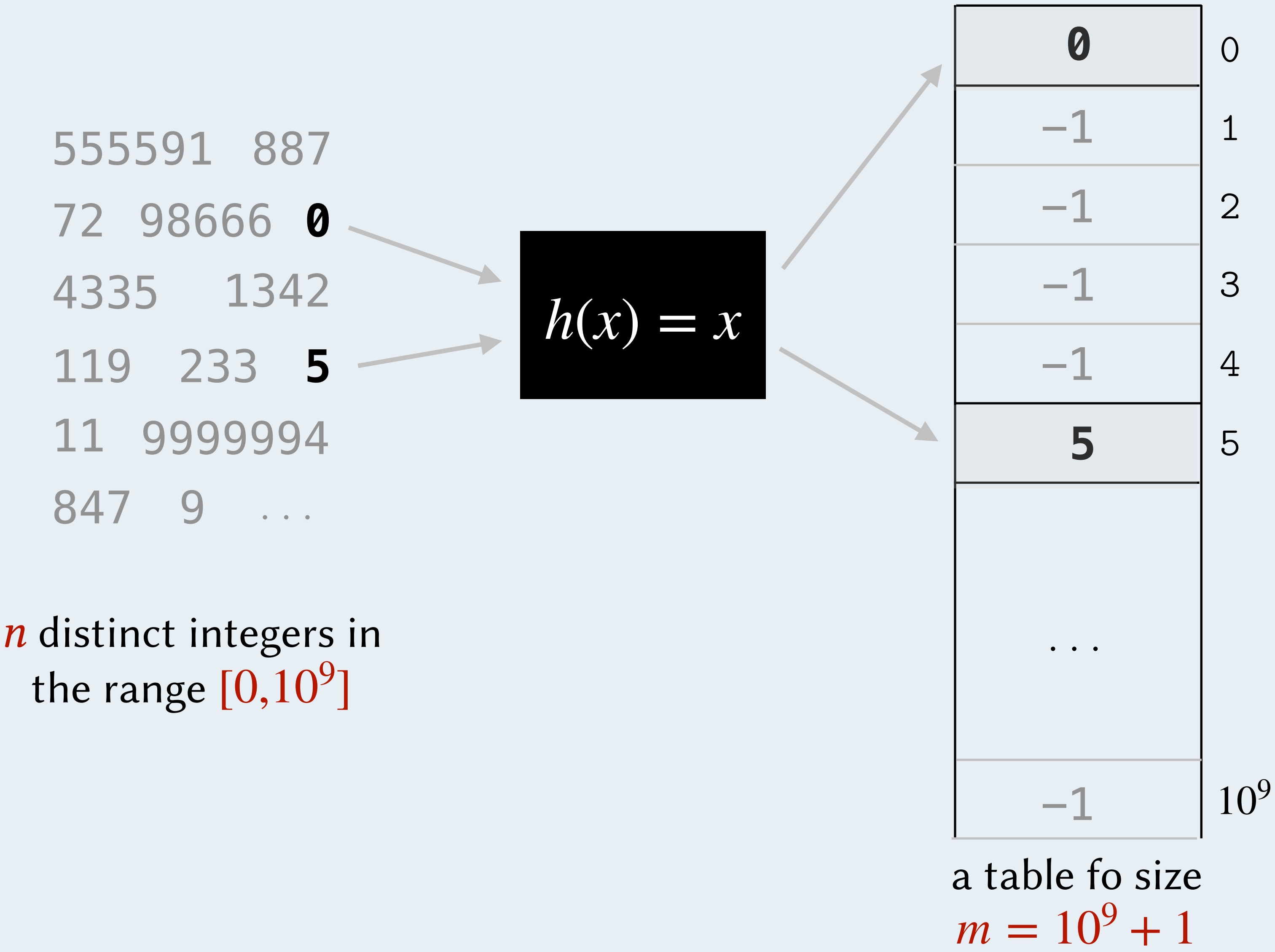
Any Problem?

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if $h(x_1) = h(x_2)$ implies $x_1 = x_2$

In other words, if $h(x)$ is *perfect,* no two distinct elements have the same hash value.

Example. $h(x) = x$ is a *perfect* hash function.



n distinct integers in the range $[0, 10^9]$

a table fo size $m = 10^9 + 1$

Any Problem?

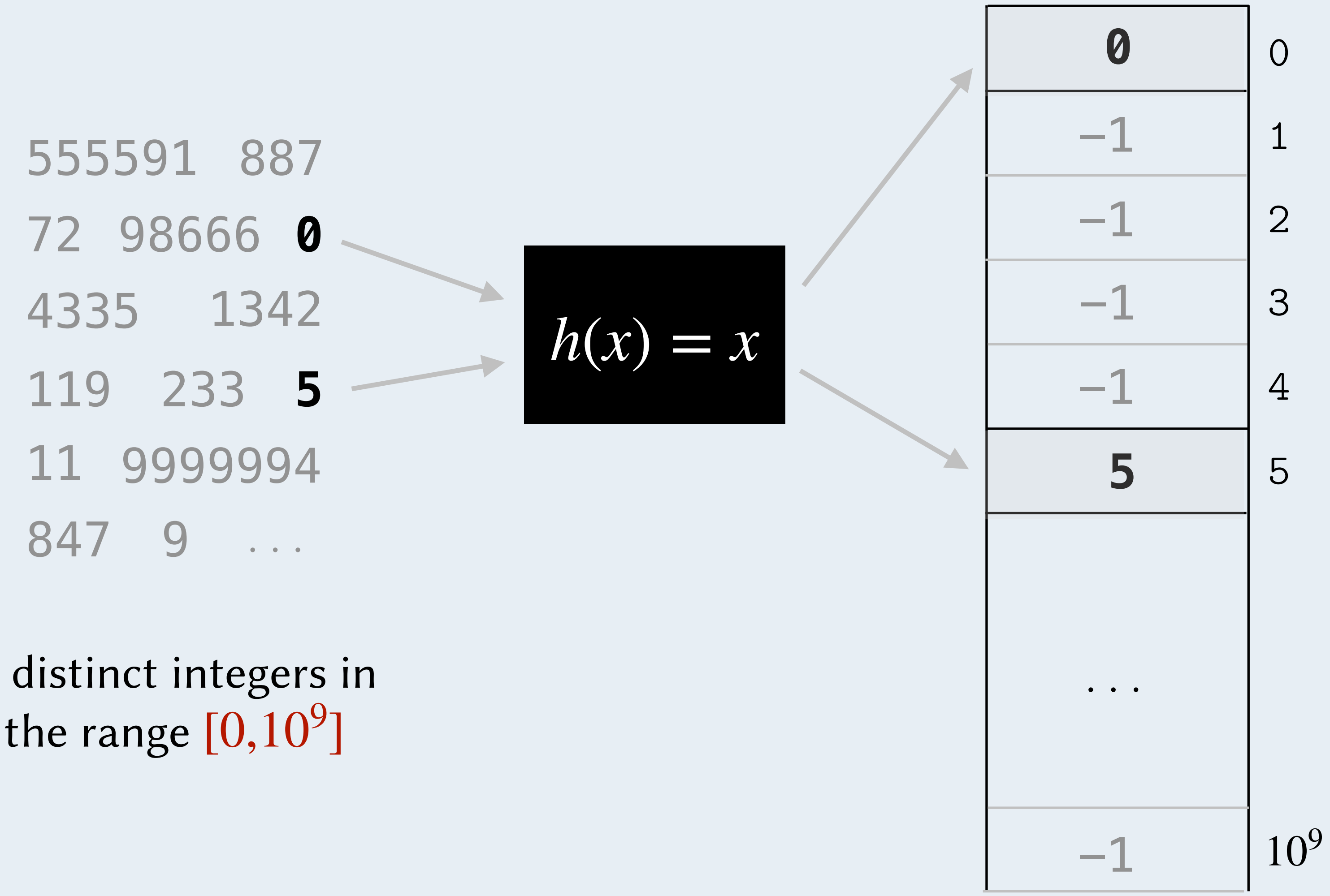
What if $n = 10$?

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if
$h(x_1) = h(x_2)$ implies $x_1 = x_2$

In other words, if $h(x)$ is *perfect*, no two distinct elements have the same hash value.

Example. $h(x) = x$ is a *perfect* hash function.



```
555591   887
72   98666   0
4335     1342
119   233   5
11   9999994
847   9   ...
```

$h(x) = x$

0	0
−1	1
−1	2
−1	3
−1	4
5	5
. . .	
−1	10^9

n distinct integers in the range $[0,10^9]$

a table fo size $m = 10^9 + 1$

Any Problem?

What if $n = 10$?

We still need a table of size $m = 10^9 + 1$

IMPRACTICAL

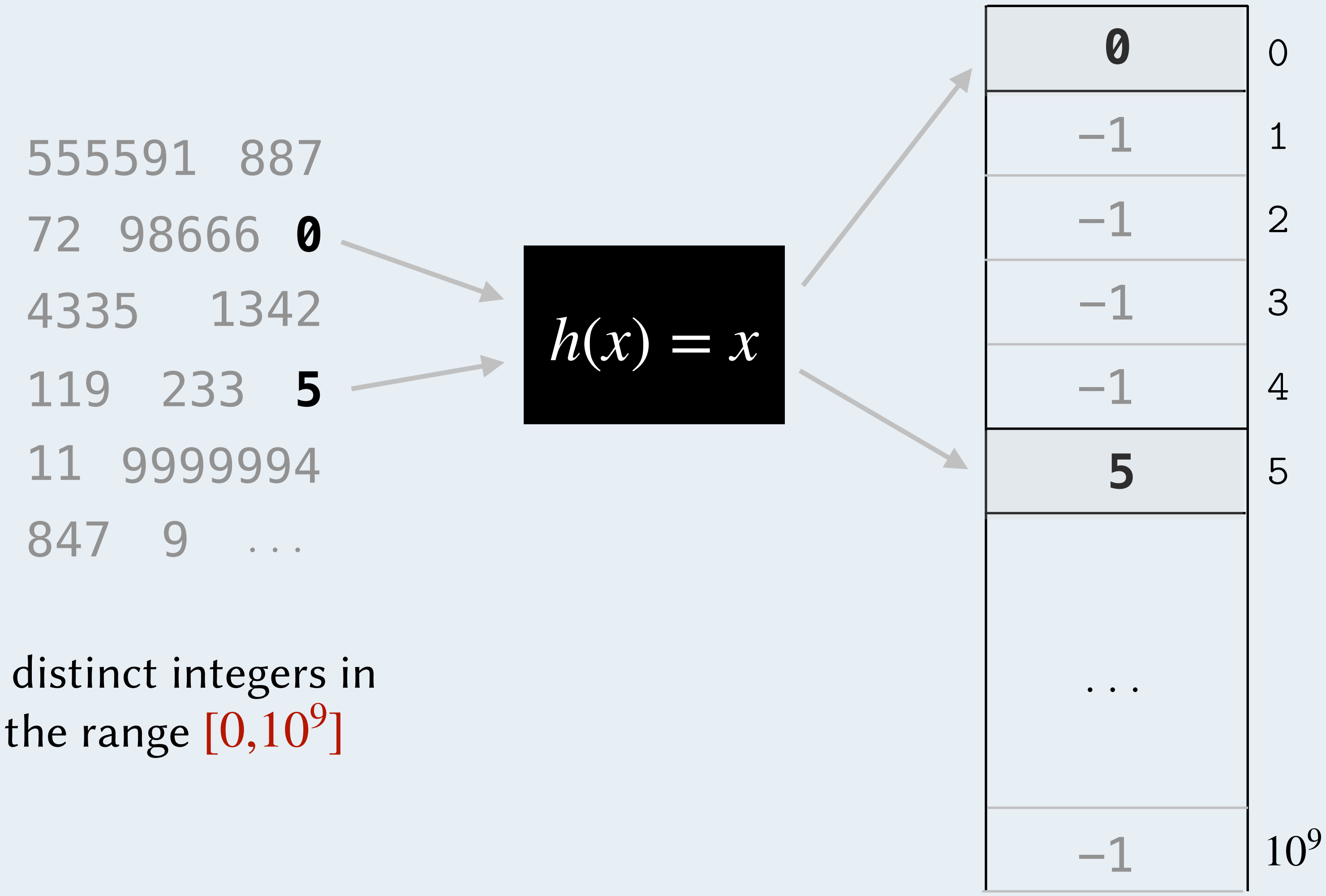
The table size depends on the *range of possible values* regardless of the number of elements to be stored (n)

A Perfect Hash Function

Definition. A hash function $h(x)$ is *perfect* if $h(x_1) = h(x_2)$ implies $x_1 = x_2$

In other words, if $h(x)$ is *perfect,* no two distinct elements have the same hash value.

Example. $h(x) = x$ is a *perfect* hash function.



n distinct integers in the range $[0,10^9]$

a table fo size $m = 10^9 + 1$

Any Problem?

What if $n = 10$?

We still need a table of size $m = 10^9 + 1$

IMPRACTICAL

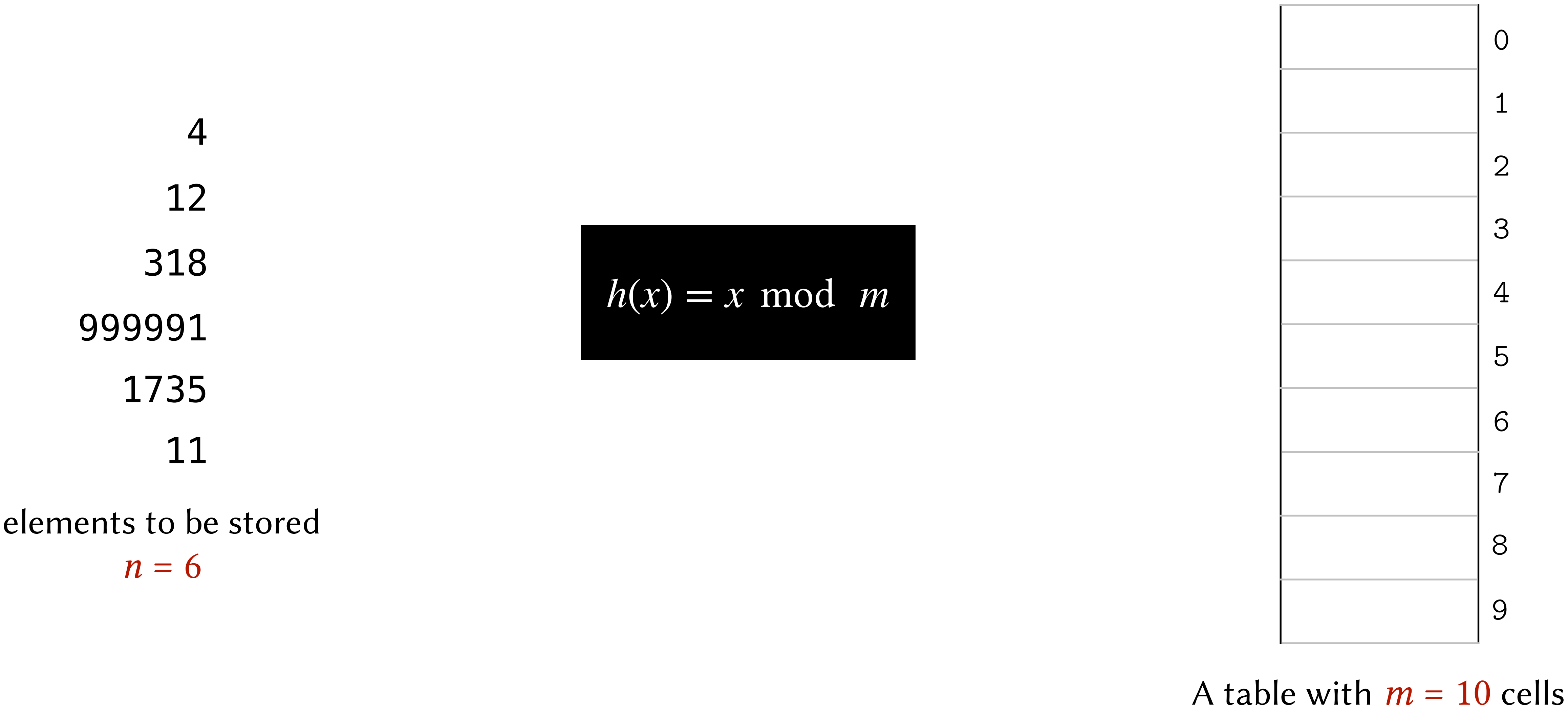
The table size depends on the *range of possible values* regardless of the number of elements to be stored (n)

We want to limit m to be not much larger than n.

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.
2. Use the following hash function: $h(x) = x \mod m$.

Example.



4

12

318

999991

1735

11

elements to be stored

$n = 6$

$h(x) = x \mod m$

	0
	1
	2
	3
	4
	5
	6
	7
	8
	9

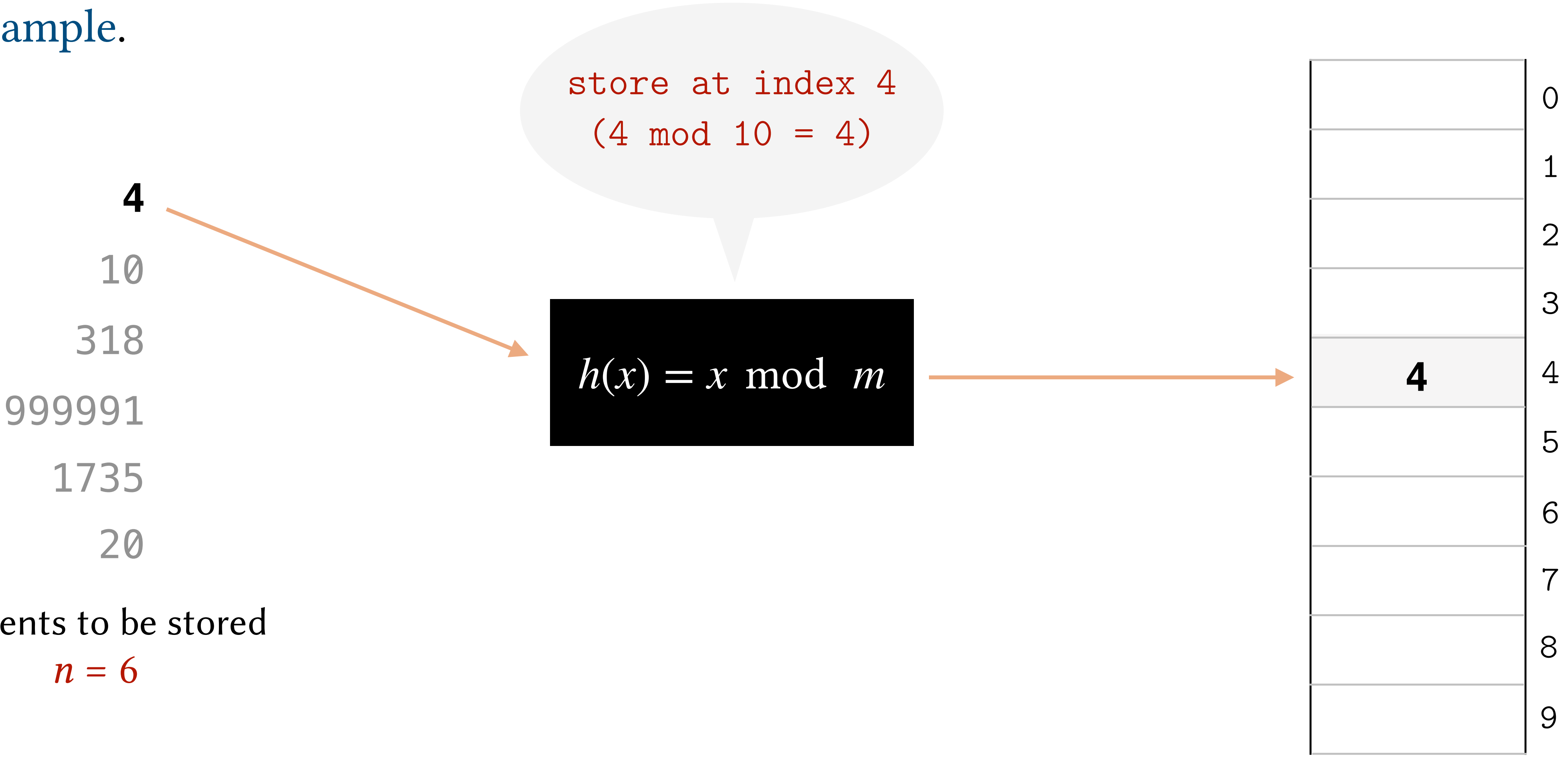
A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.

2. Use the following hash function: $h(x) = x \mod m$.

Example.



```
store at index 4
(4 mod 10 = 4)
```

elements to be stored
$n = 6$

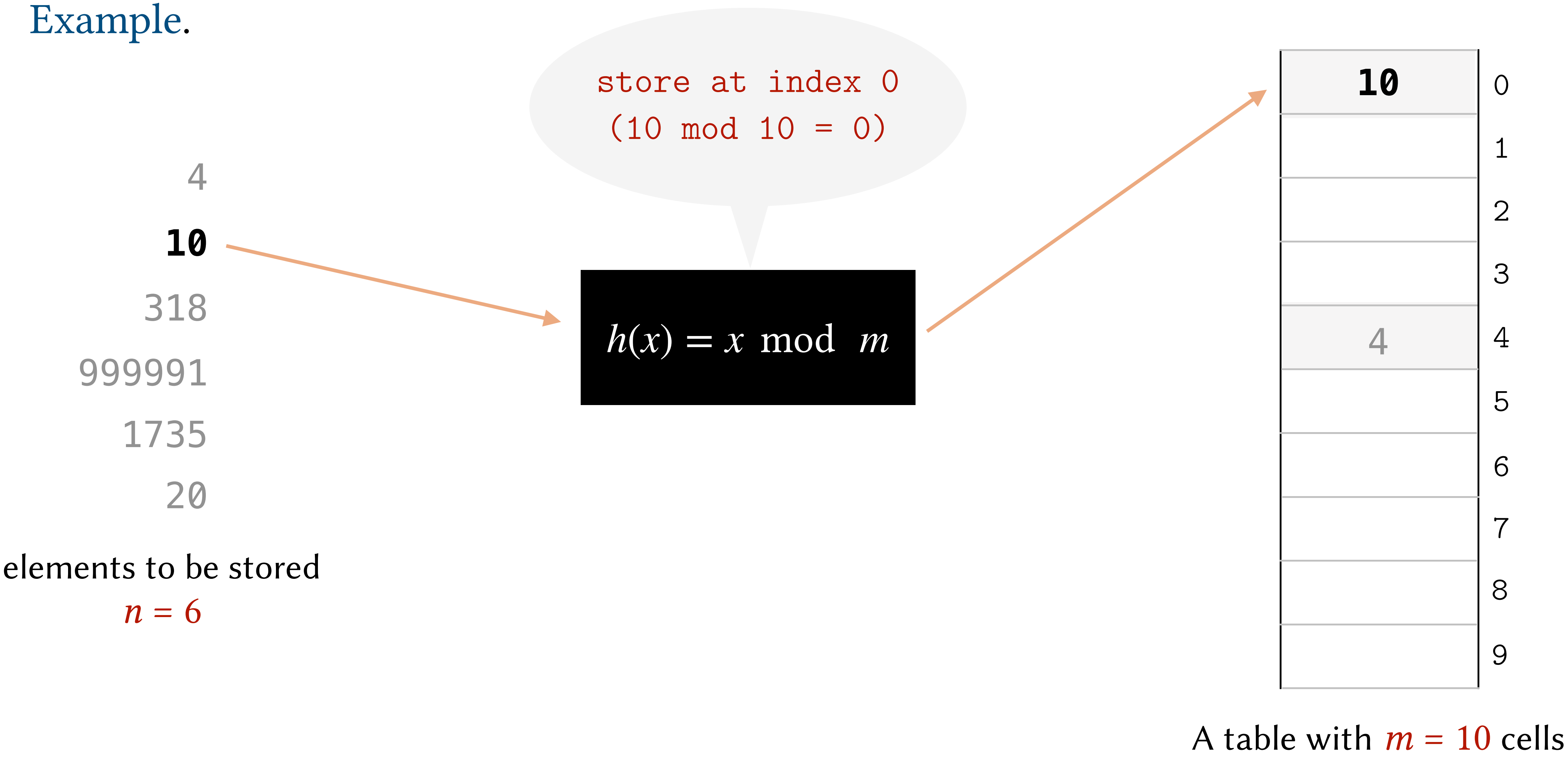
A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.

2. Use the following hash function: $h(x) = x \mod m$.

Example.



elements to be stored
$n = 6$

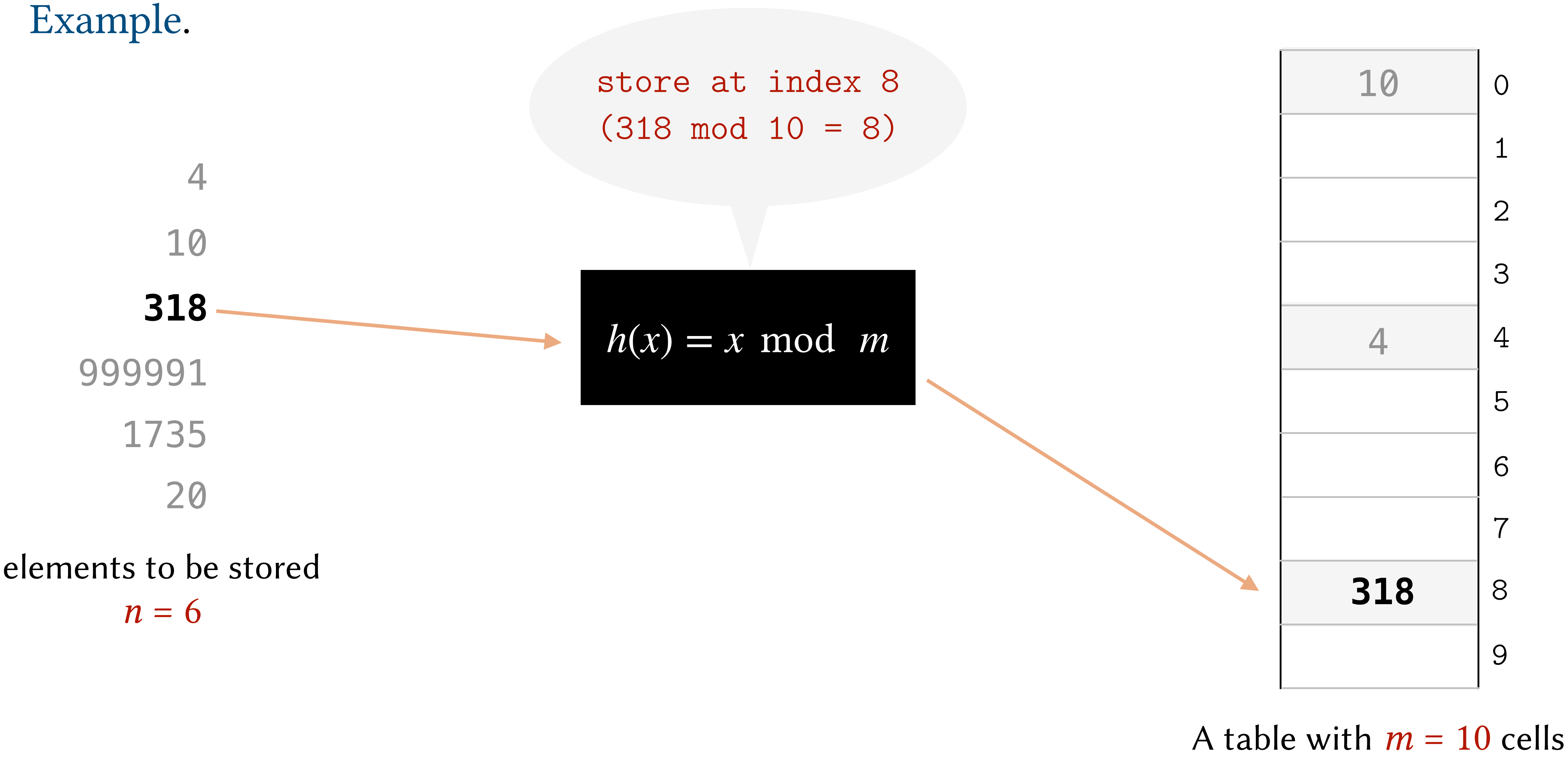
A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.

2. Use the following hash function: $h(x) = x \mod m$.

Example.



store at index 8
(318 mod 10 = 8)

$h(x) = x \mod m$

4
10
318
999991
1735
20

elements to be stored
$n = 6$

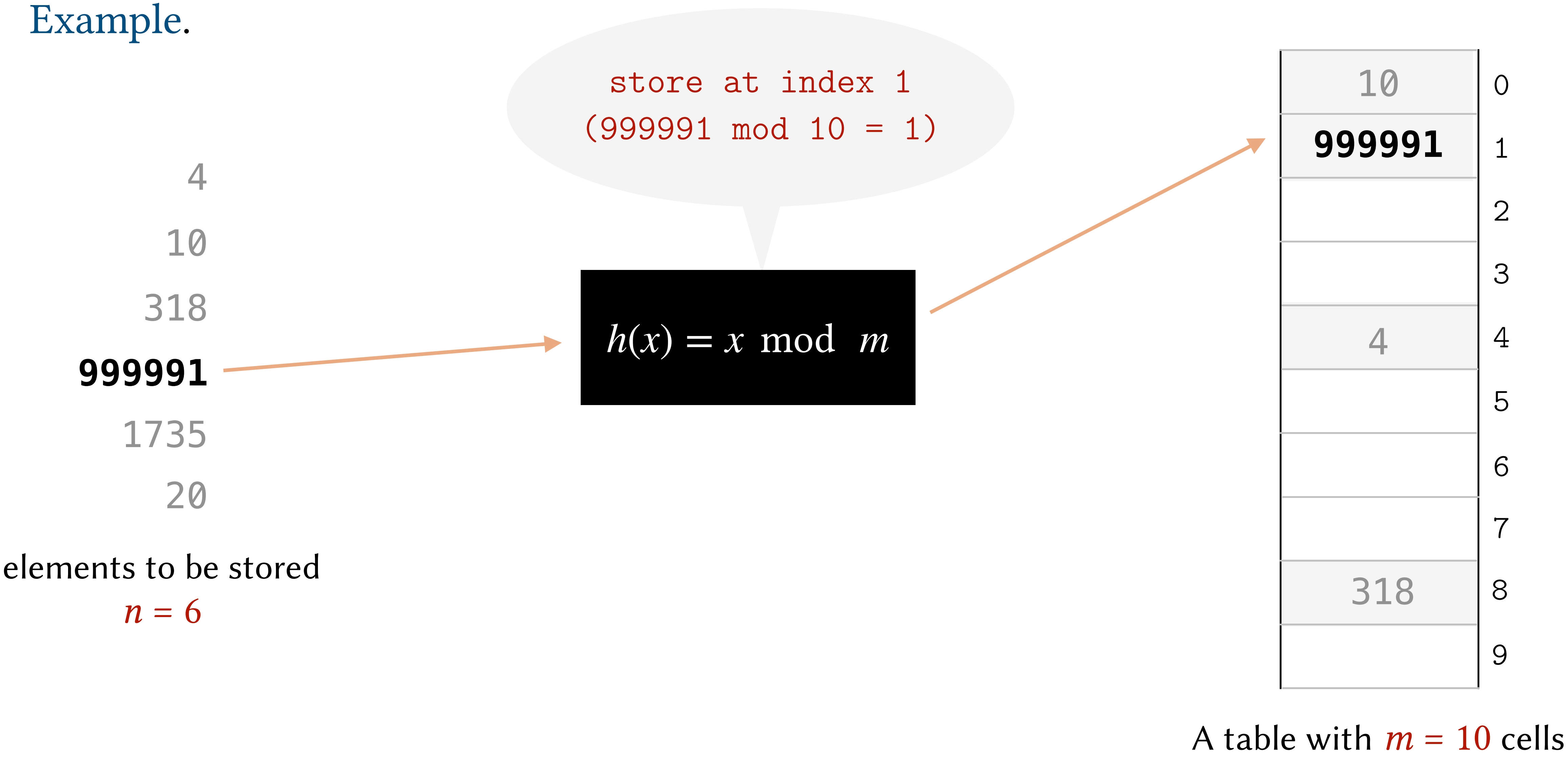
10	0
	1
	2
	3
4	4
	5
	6
	7
318	8
	9

A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.
2. Use the following hash function: $h(x) = x \mod m$.

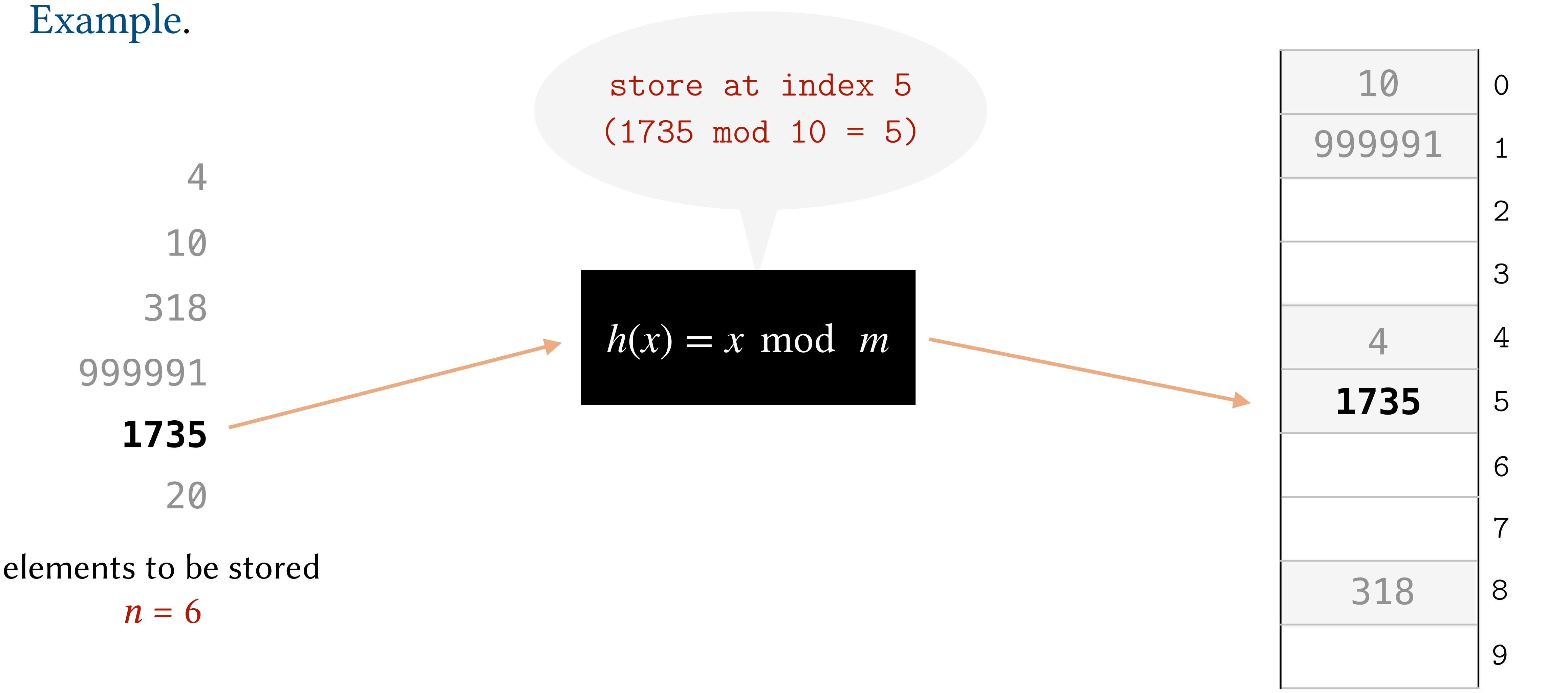
Example.



Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.
2. Use the following hash function: $h(x) = x \mod m$.

Example.



elements to be stored
$n = 6$

A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size m that is not much larger than the number of elements to be stored n.
2. Use the following hash function: $h(x) = x \mod m$.

COLLISION!

Example.

4

10

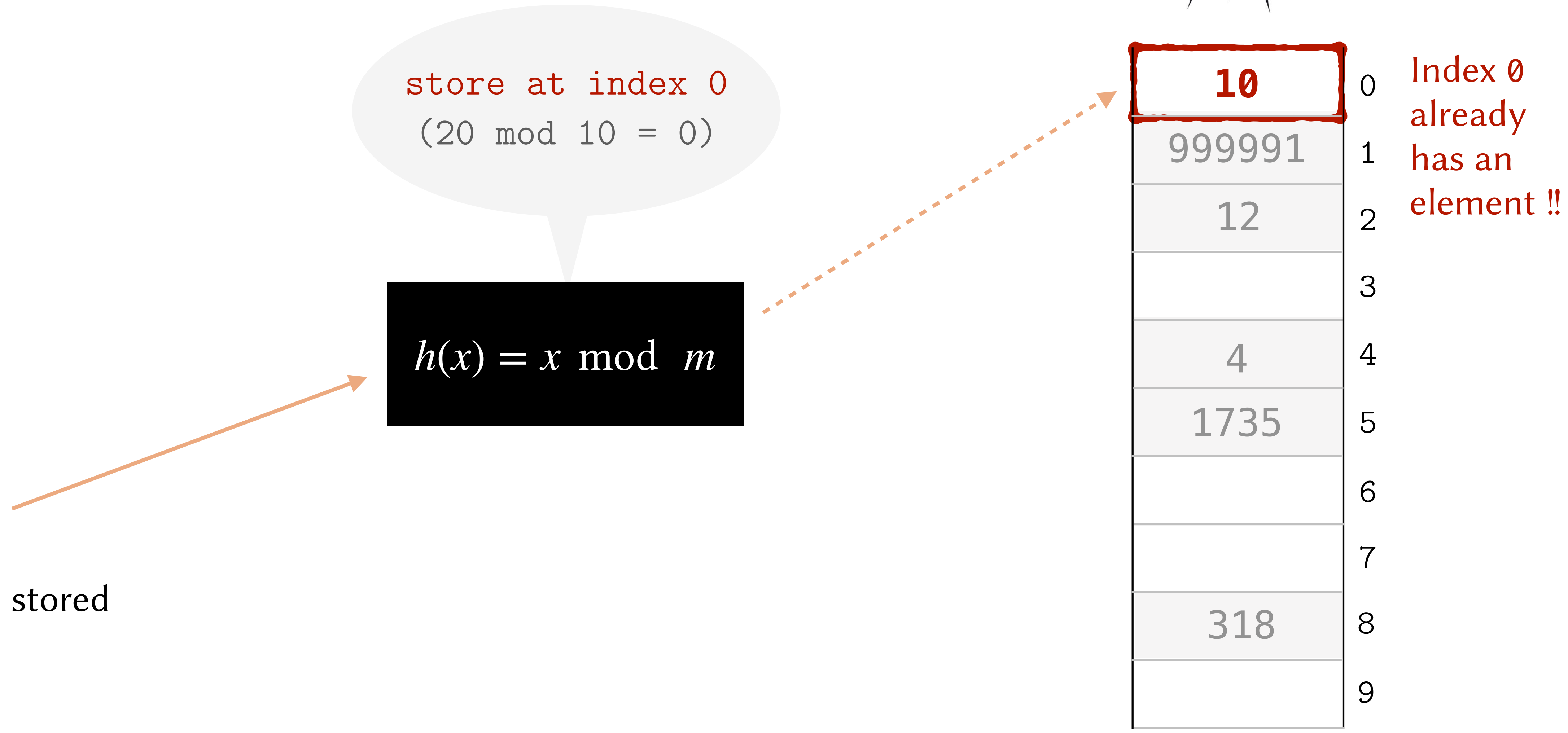
318

999991

1735

20

elements to be stored
$n = 6$



store at index 0
(20 mod 10 = 0)

$h(x) = x \mod m$

0	**10**
1	999991
2	12
3	
4	4
5	1735
6	
7	
8	318
9	

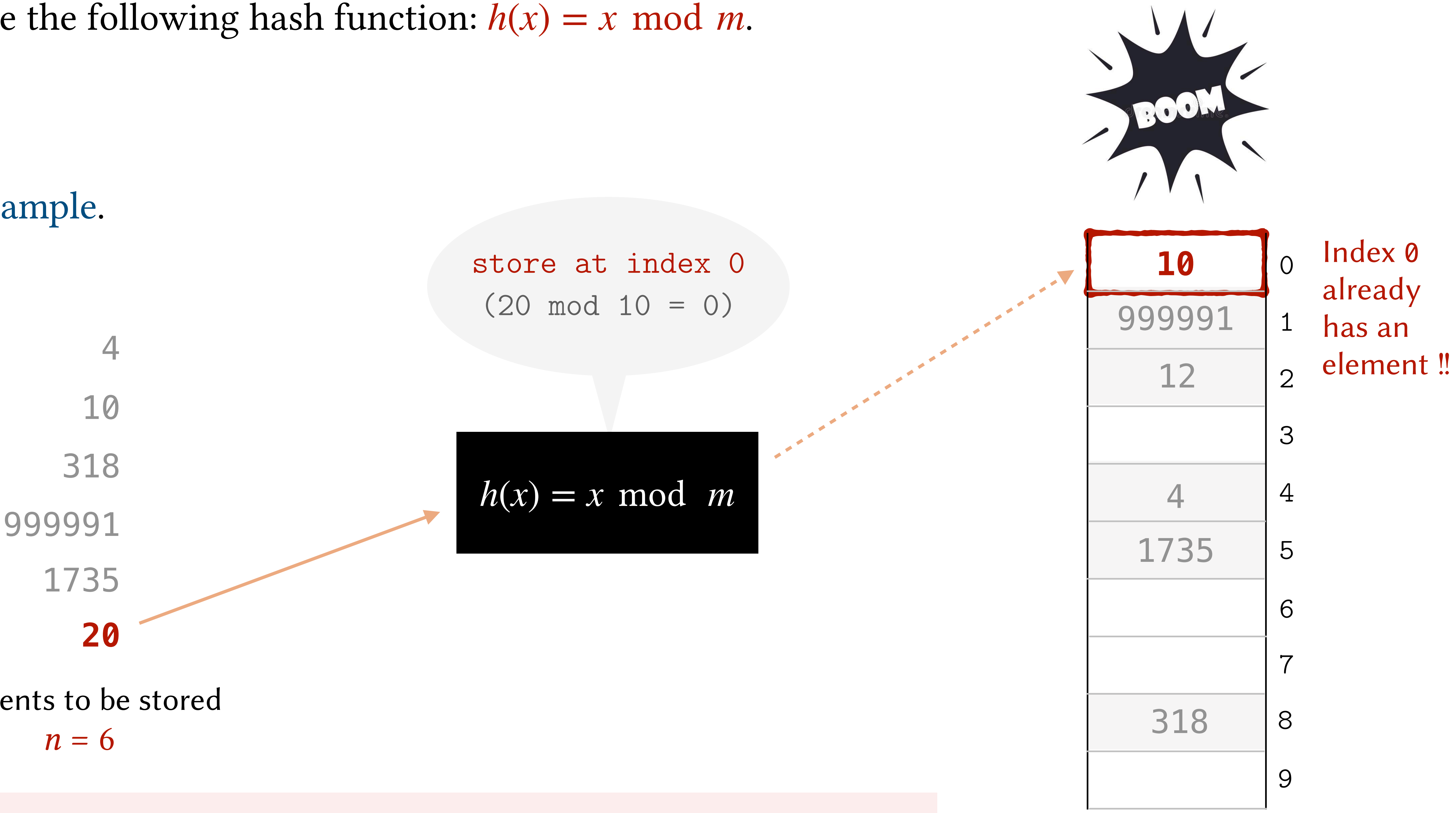
Index 0 already has an element ‼

A table with $m = 10$ cells

Modular Hashing

1. Pick a hash table size *m* that is not much larger than the number of elements to be stored *n*.
2. Use the following hash function: $h(x) = x \mod m$.

Example.



COLLISION!

store at index 0
(20 mod 10 = 0)

4
10
318
999991
1735
20

$h(x) = x \mod m$

10	0
999991	1
12	2
	3
4	4
1735	5
	6
	7
318	8
	9

Index 0 already has an element ‼

elements to be stored
$n = 6$

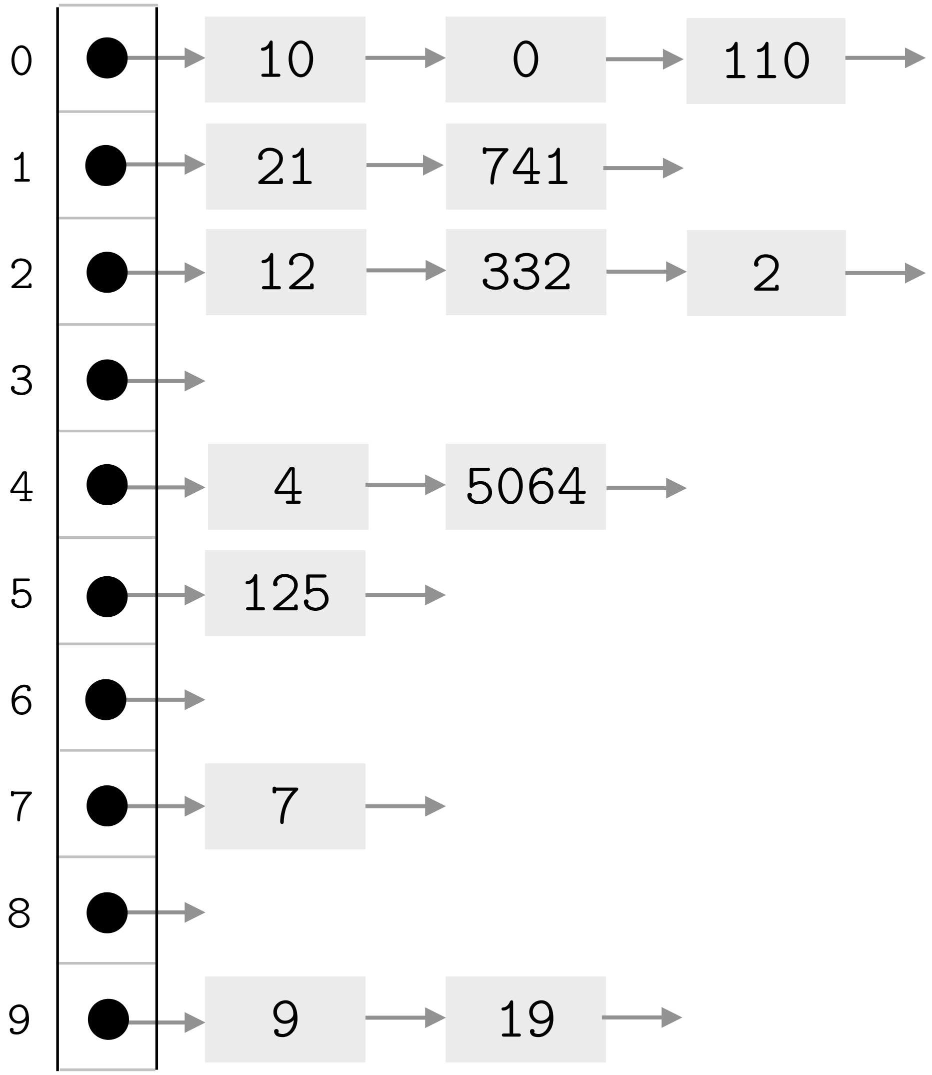
A table with $m = 10$ cells

!

Since `m = 10`: `0, 10, 20, 30,` etc. all map to index `0`,
`1, 11, 21, 31,` etc. all map to index `1`, etc.
How can we deal with such collisions?

Collision Resolution using Separate Chaining

Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

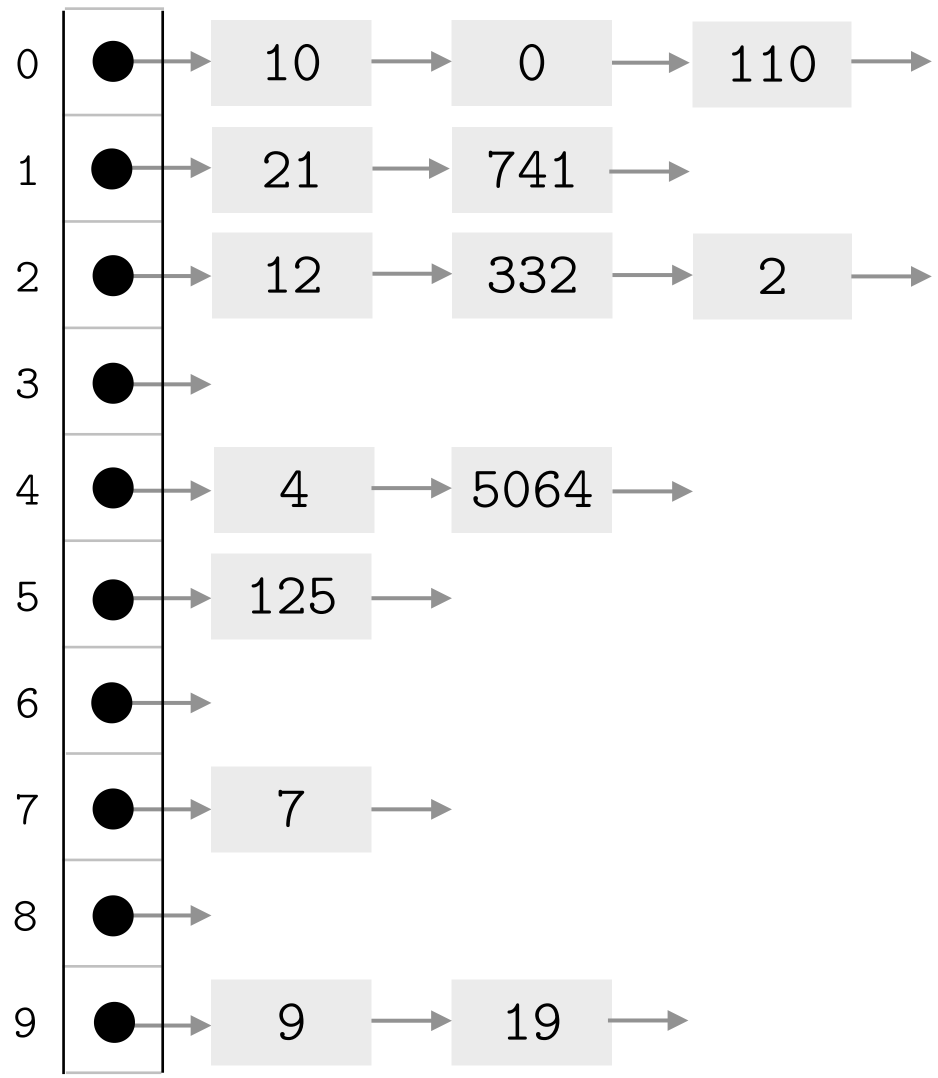


An array of
$m = 10$
linked lists

Collision Resolution using Separate Chaining

Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

```
insert(x) : table[h(x)].addToTail(x)
```



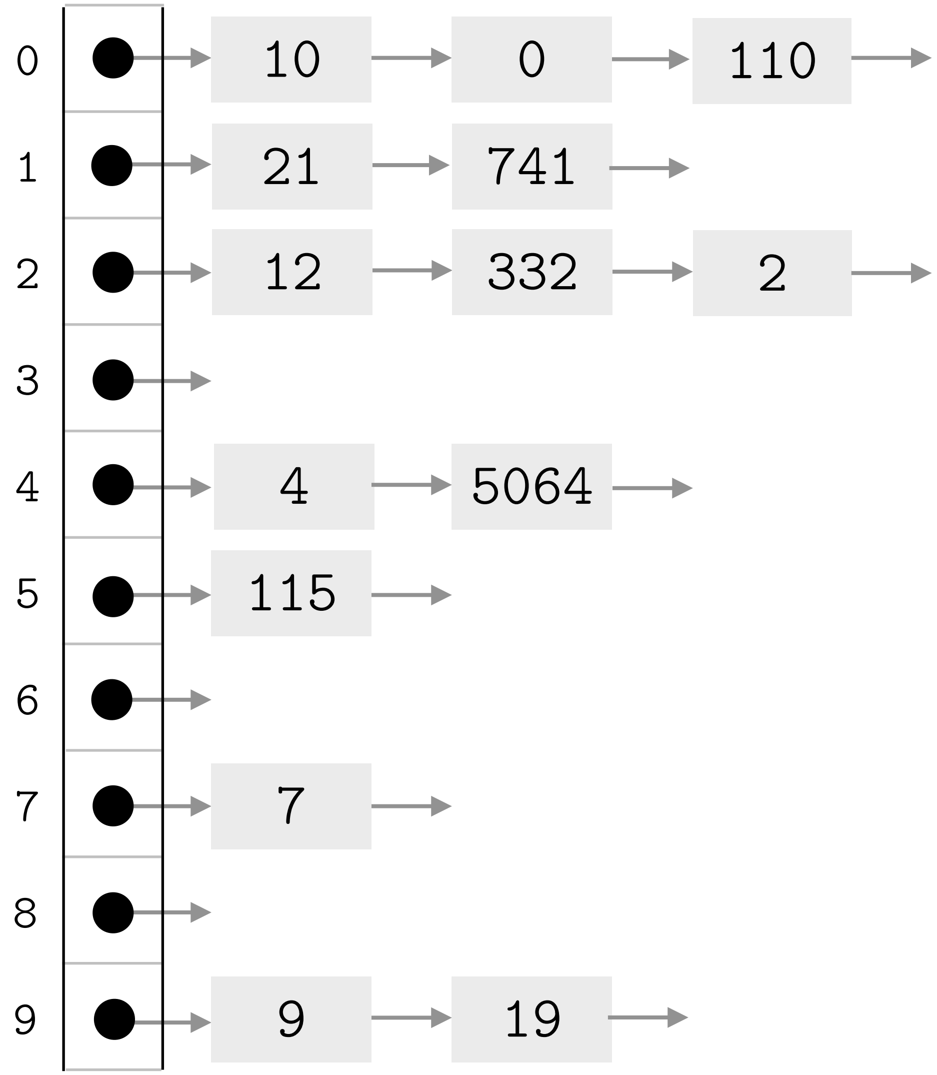
An array of
$m = 10$
linked lists

Collision Resolution using Separate Chaining

Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

```
insert(x) : table[h(x)].addToTail(x)
```

use the hash function to know which linked list x should be added to



An array of $m = 10$ linked lists

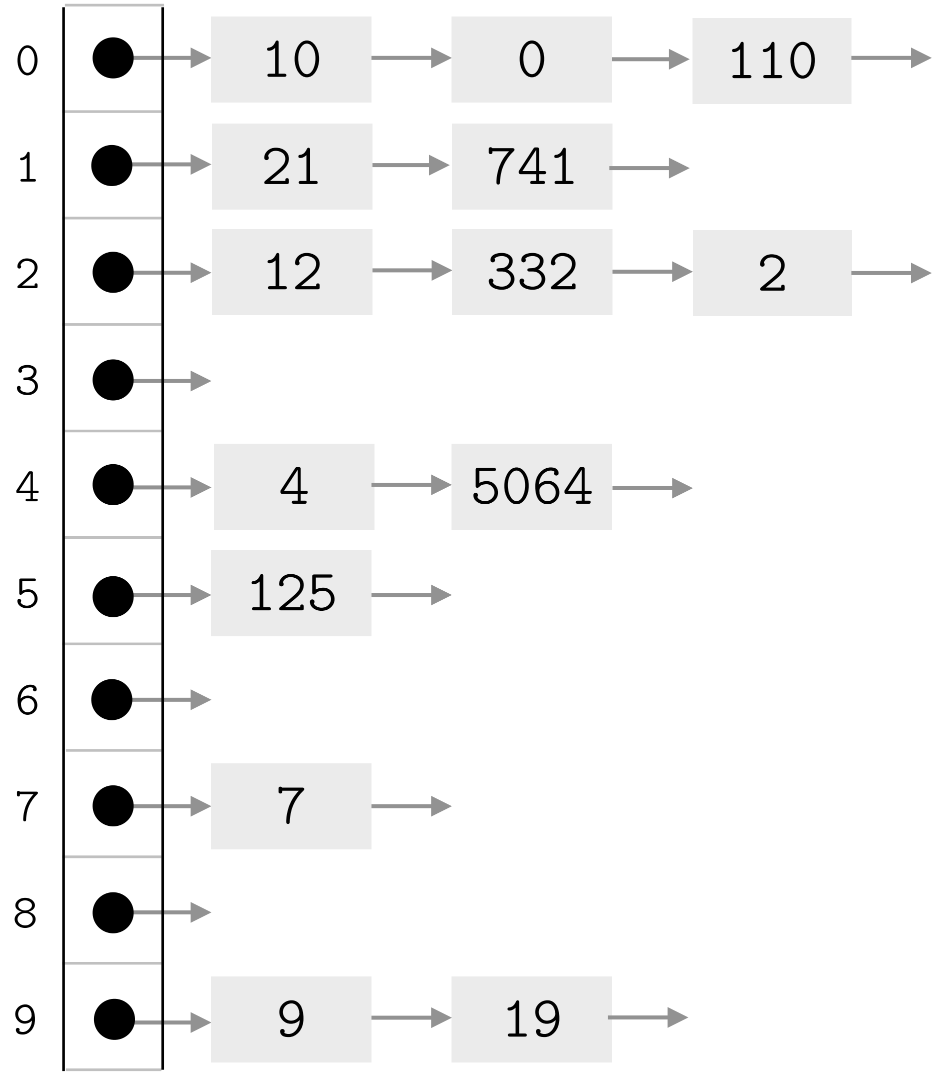
Collision Resolution using Separate Chaining

Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

```
insert(x) : table[h(x)].addToTail(x)

remove(x) : table[h(x)].remove(x)
```

search the linked list for x
and remove it if found



An array of
m = 10
linked lists

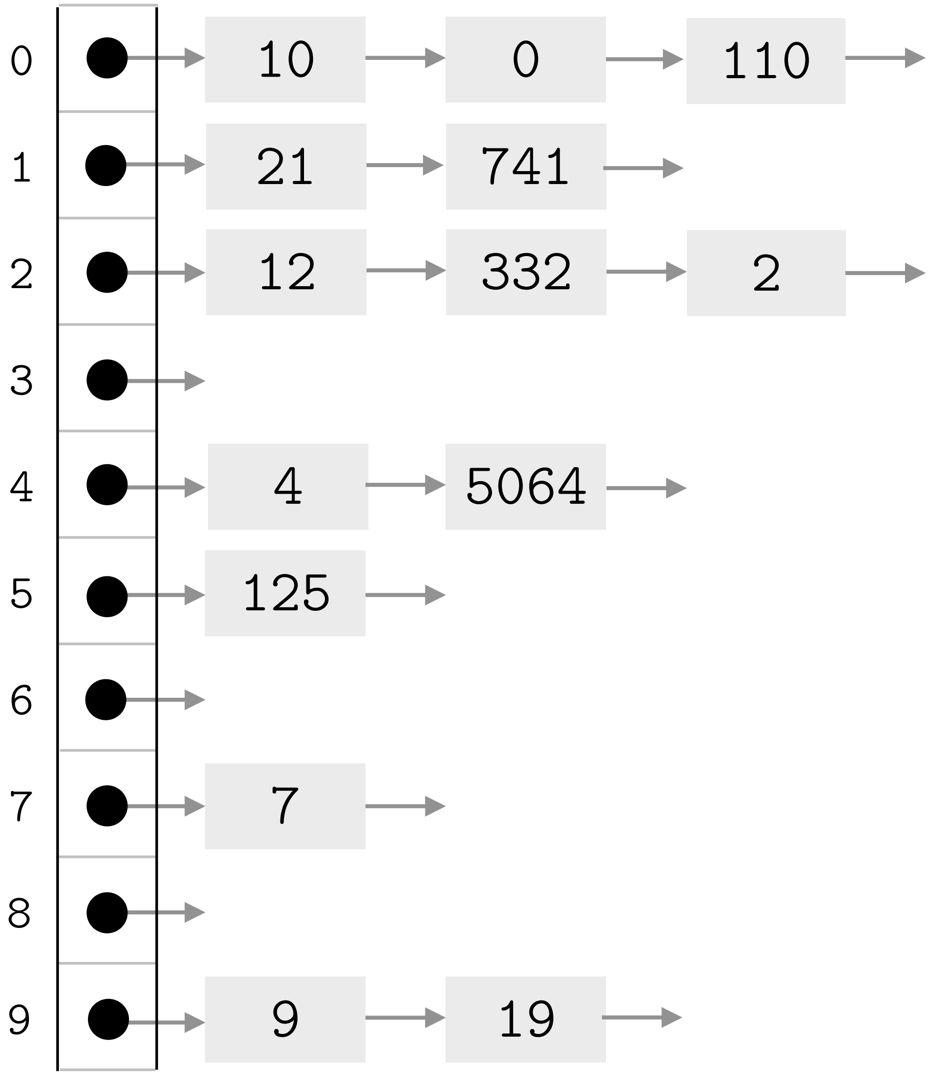
Collision Resolution using Separate Chaining

Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

```
insert(x) : table[h(x)].addToTail(x)

remove(x) : table[h(x)].remove(x)

search(x) : return table[h(x)].find(x)
```



An array of
$m = 10$
linked lists

Collision Resolution using Separate Chaining

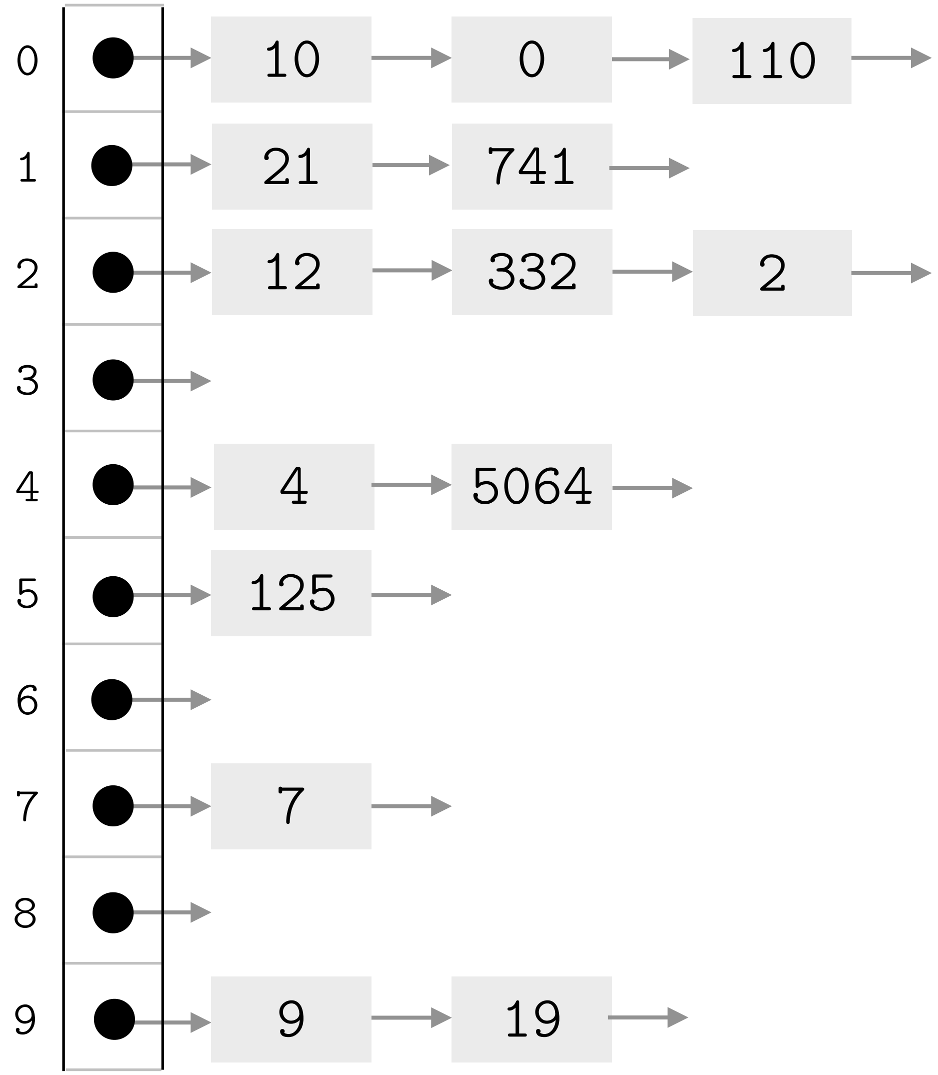
Idea. Allow each cell in the table to hold more than one element.
Implementation. Define the hash table as an array of linked lists.

```
insert(x) : table[h(x)].addToTail(x)

remove(x) : table[h(x)].remove(x)

search(x) : return table[h(x)].find(x)
```

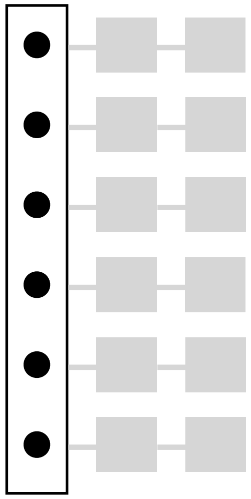
🤔 Is the running time still $O(1)$?



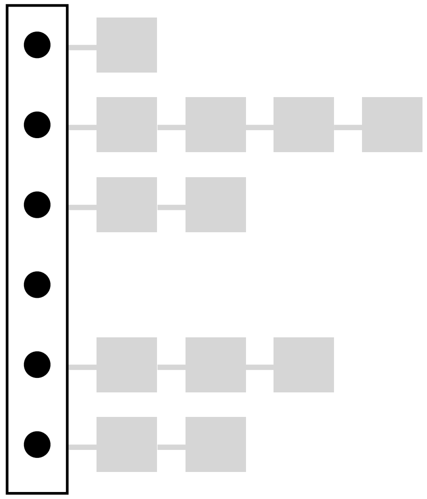
An array of
$m = 10$
linked lists

Running Time

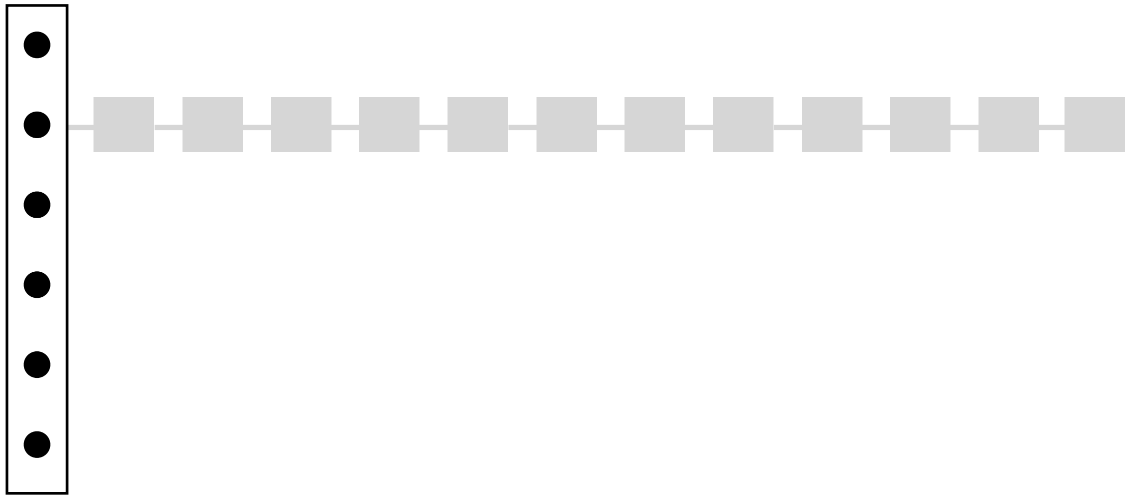
Different chain lengths?



$m = 6$
$n = 12$



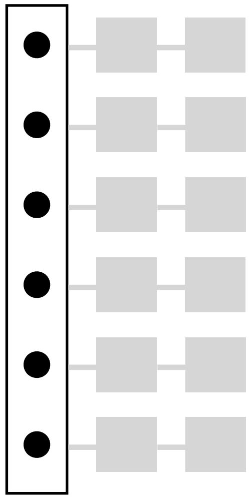
$m = 6$
$n = 12$



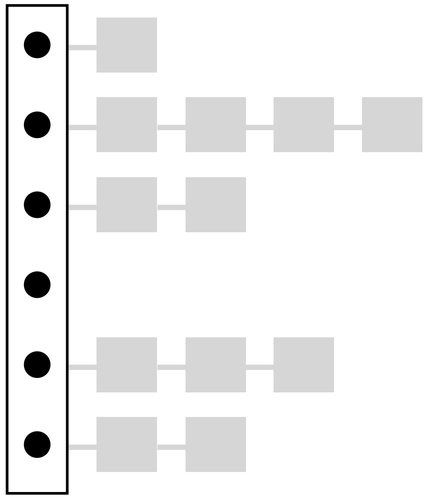
$m = 6$
$n = 12$

Running Time

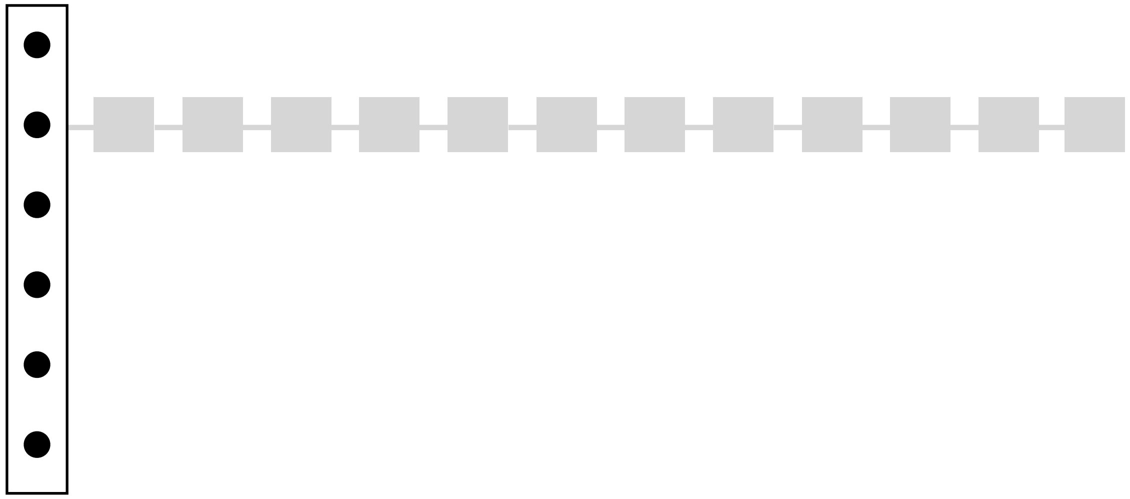
Different chain lengths?



$m = 6$
$n = 12$



$m = 6$
$n = 12$

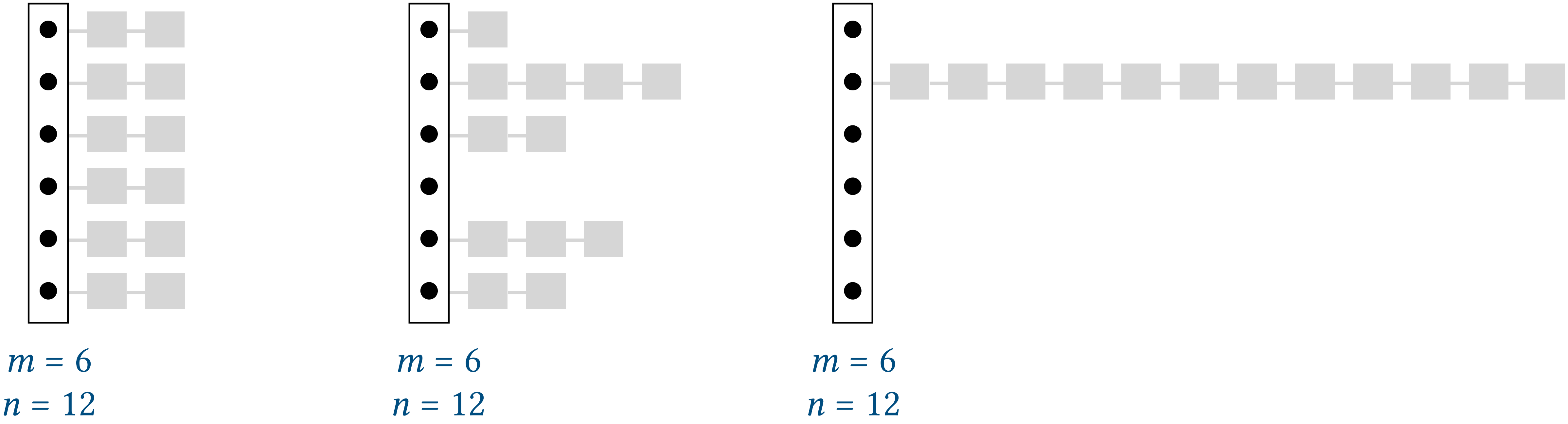


$m = 6$
$n = 12$

operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`		

Running Time

Different chain lengths?

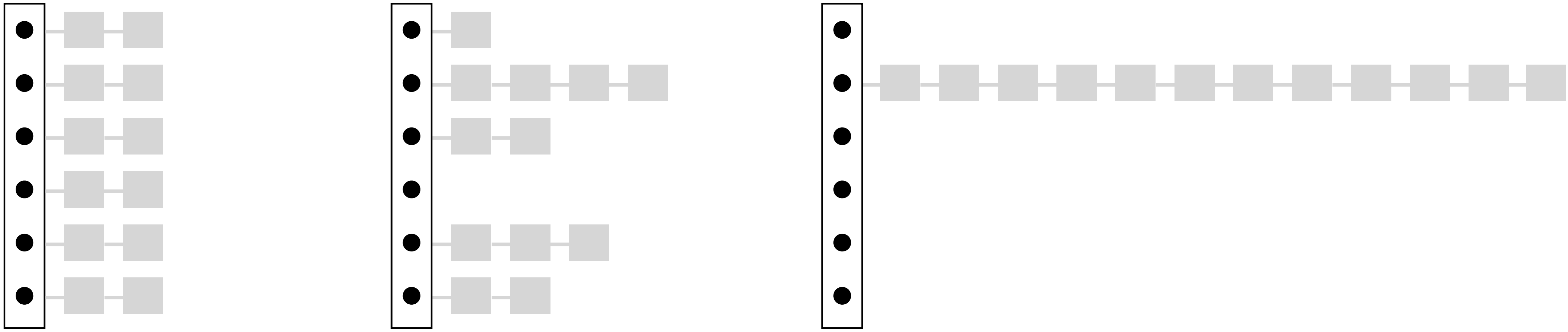


operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$

the running time is independent of the chain length!

Running Time

Different chain lengths?



$m = 6$
$n = 12$

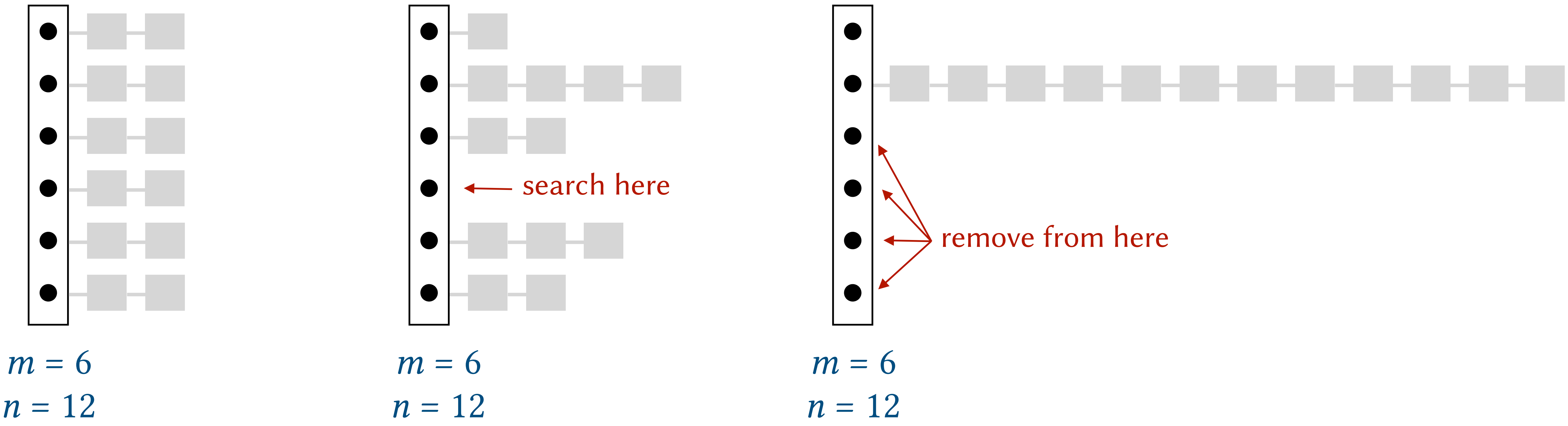
$m = 6$
$n = 12$

$m = 6$
$n = 12$

operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$
`remove(x)`	`table[h(x)].remove(x)`		
`search(x)`	`return table[h(x)].find(x)`		

Running Time

Different chain lengths?

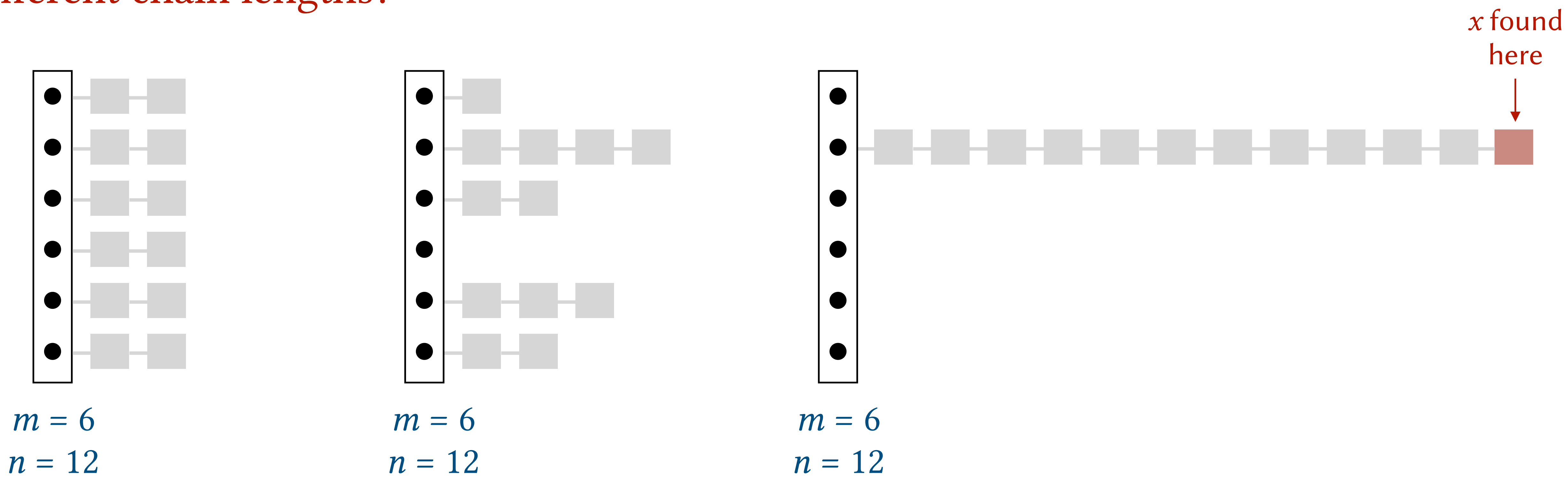


operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$
`remove(x)`	`table[h(x)].remove(x)`	$O(1)$	
`search(x)`	`return table[h(x)].find(x)`	$O(1)$	

if the chain
is empty

Running Time

Different chain lengths?

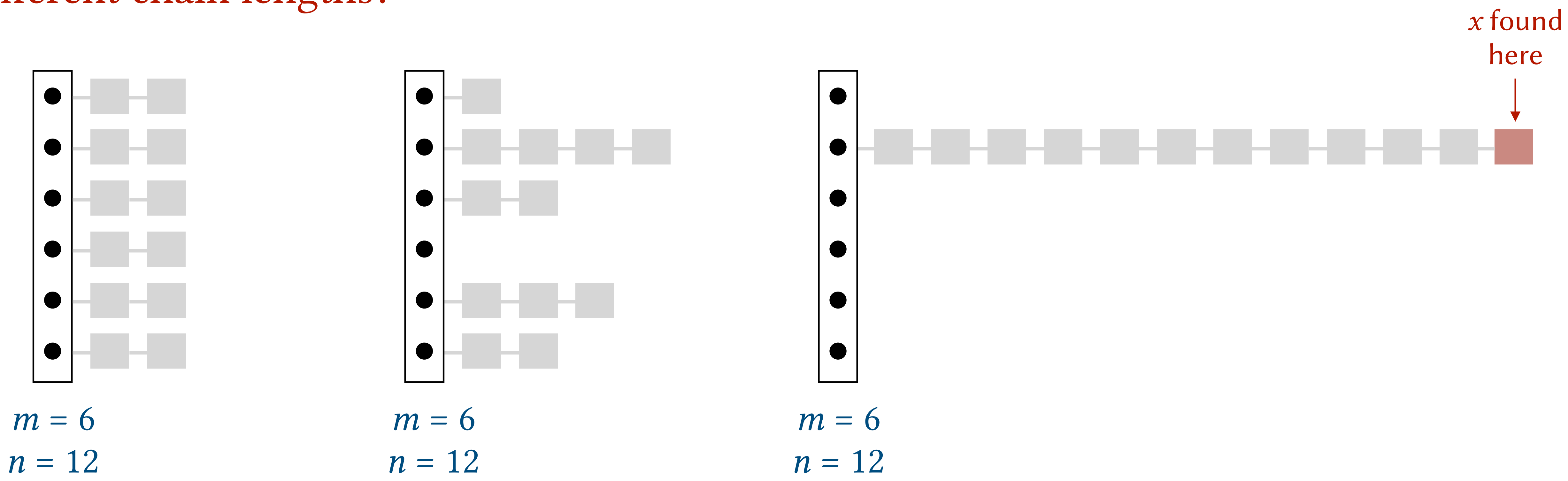


operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$
`remove(x)`	`table[h(x)].remove(x)`	$O(1)$	$O(n)$
`search(x)`	`return table[h(x)].find(x)`	$O(1)$	$O(n)$

if all the elements are in one chain
and x is found at the end of that chain

Running Time

Different chain lengths?

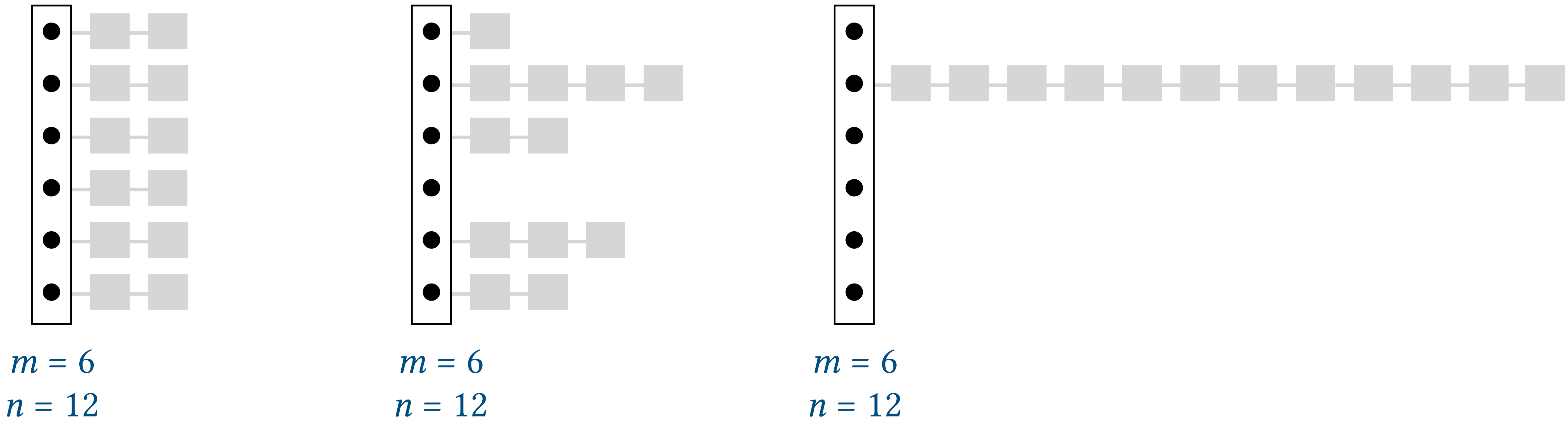


operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$
`remove(x)`	`table[h(x)].remove(x)`	$O(1)$	$O(n)$
`search(x)`	`return table[h(x)].find(x)`	$O(1)$	$O(n)$

😭

Running Time

Different chain lengths?



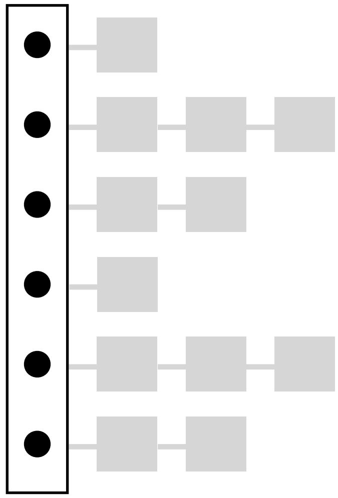
operation	implementation	best case	worst case
`insert(x)`	`table[h(x)].addToTail(x)`	$O(1)$	$O(1)$
`remove(x)`	`table[h(x)].remove(x)`	$O(1)$	$O(n)$
`search(x)`	`return table[h(x)].find(x)`	$O(1)$	$O(n)$

! **Good news.** The running time is $O(1)$ in many practical applications.

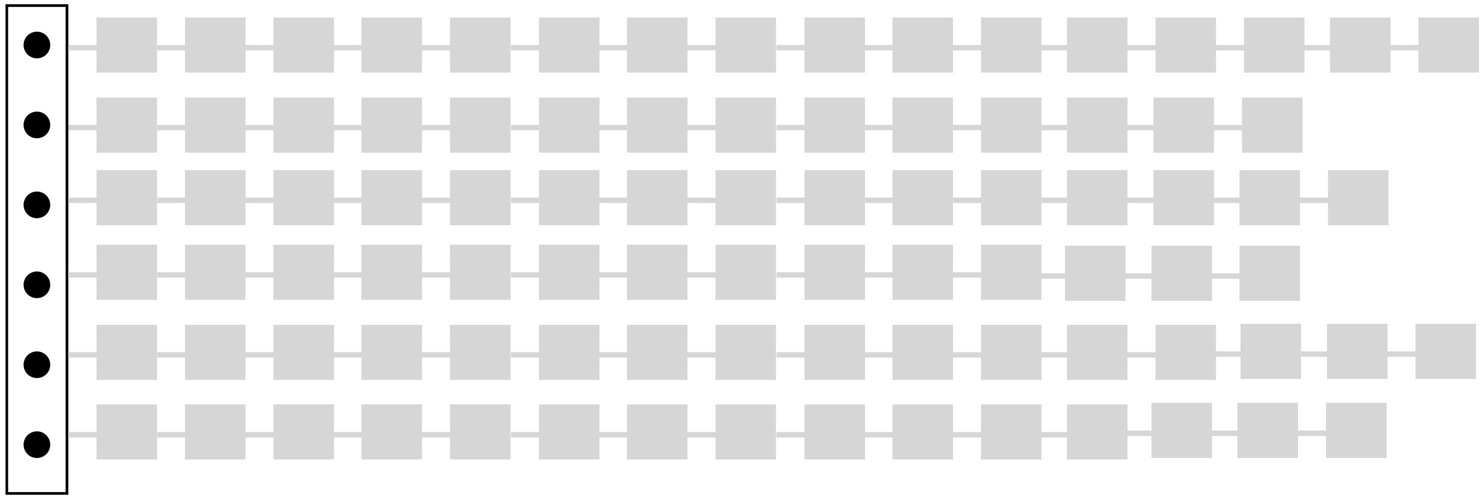
When do hash tables perform well?

Load Factor. The average chain length in the table $= n/m$.

Examples.



$m = 6$
$n = 12$
Load factor $(n\,/m) = 2$

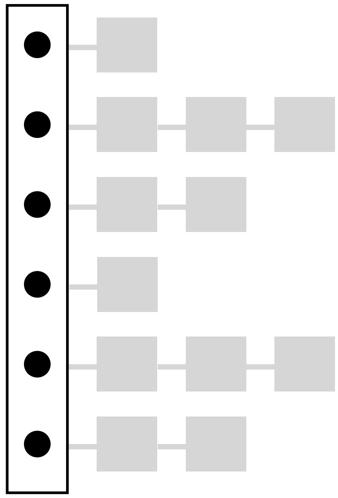


$m = 6$
$n = 90$
Load factor $(n\,/m) = 15$

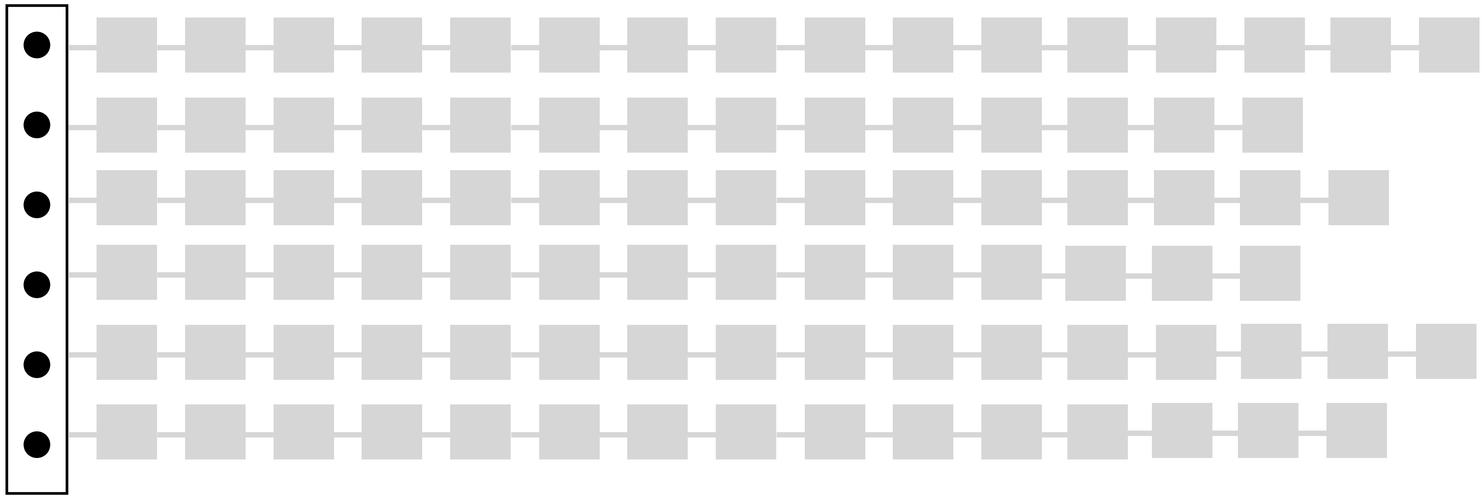
When do hash tables perform well?

Load Factor. The average chain length in the table $= n/m$.

Examples.



$m = 6$
$n = 12$
Load factor $(n\,/\,m) = 2$



$m = 6$
$n = 90$
Load factor $(n\,/\,m) = 15$

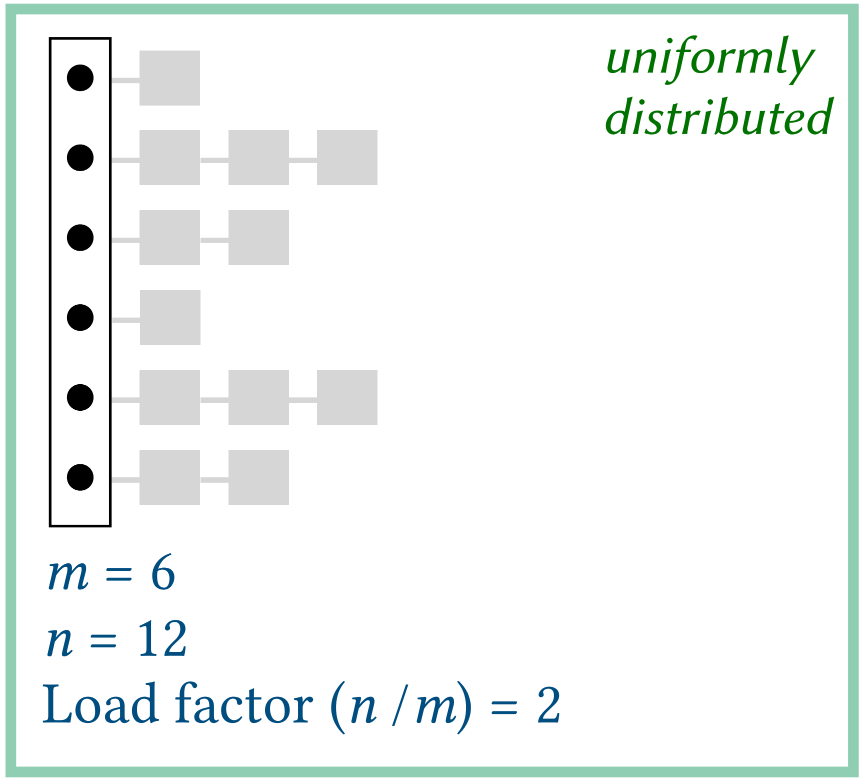
Assumption 1. Elements are *distributed uniformly* in the table.

Under this assumption, *search* and *remove* run in $O(n/m)$

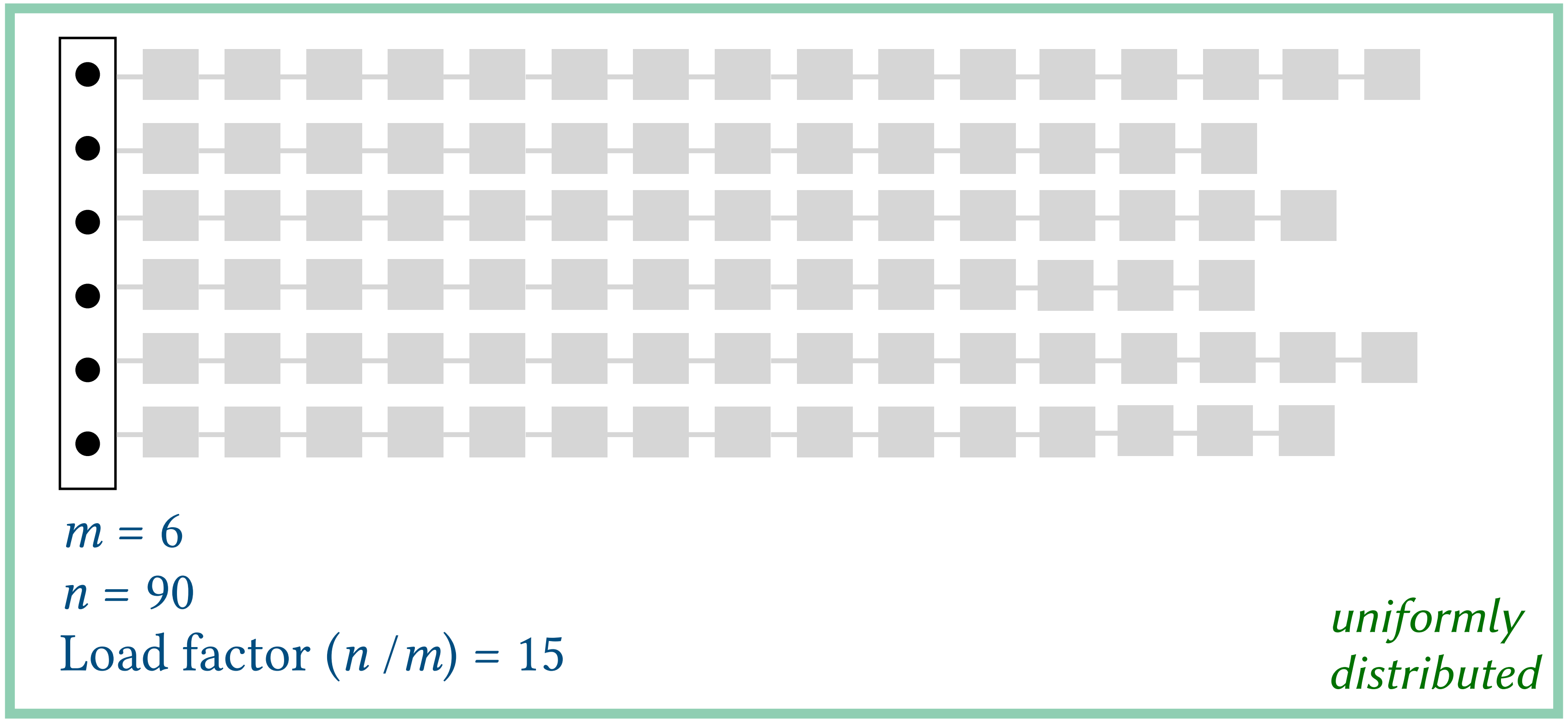
When do hash tables perform well?

Load Factor. The average chain length in the table $= n/m$.

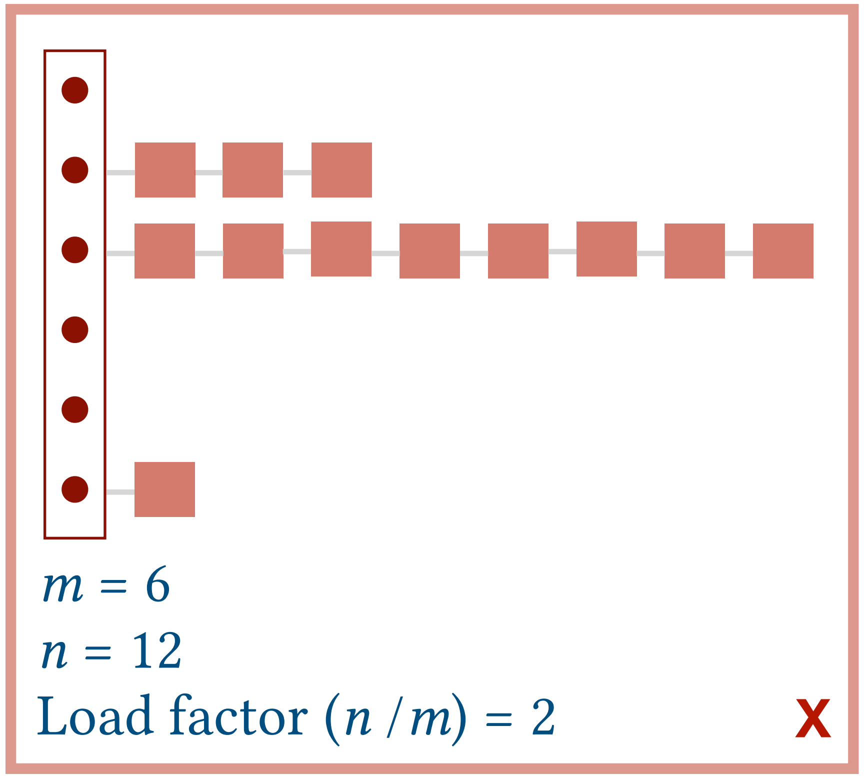
Examples.



$m = 6$
$n = 12$
Load factor $(n/m) = 2$



$m = 6$
$n = 90$
Load factor $(n/m) = 15$



$m = 6$
$n = 12$
Load factor $(n/m) = 2$

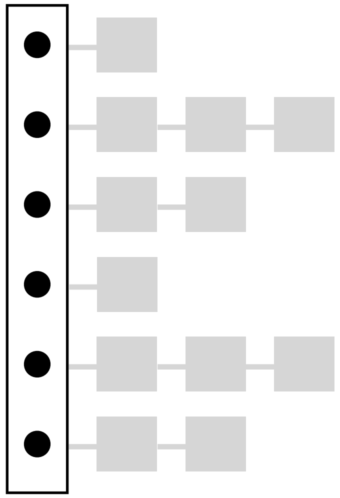
Assumption 1. Elements are *distributed uniformly* in the table.

Under this assumption, *search* and *remove* run in $O(n/m)$

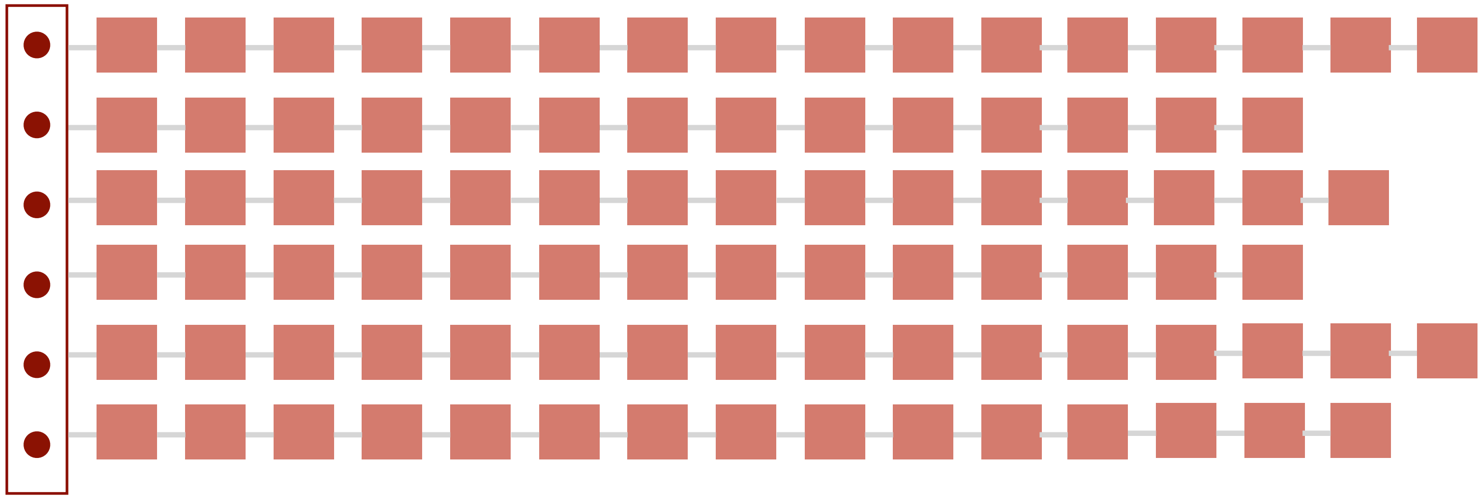
When do hash tables perform well?

Load Factor. The average chain length in the table $= n/m$.

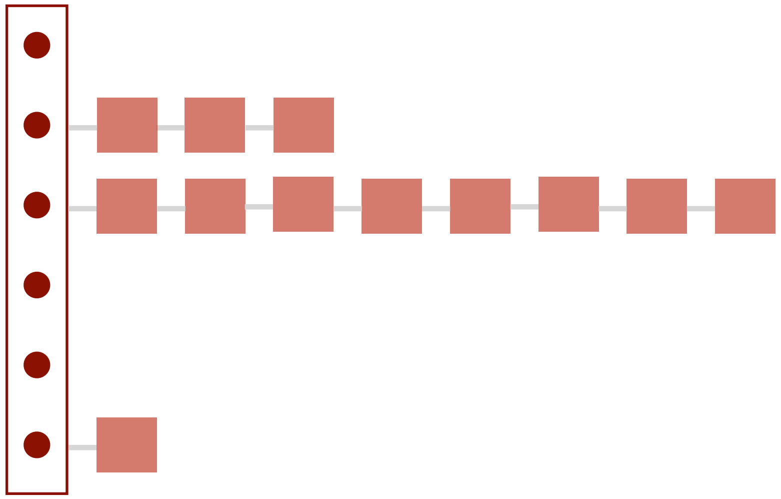
Examples.



$m = 6$
$n = 12$
Load factor $(n\,/m) = 2$ ✓



$m = 6$
$n = 90$
Load factor $(n\,/m) = 15$ X



$m = 6$
$n = 12$
Load factor $(n\,/m) = 2$ X

Assumption 1. Elements are *distributed uniformly* in the table.

Under this assumption, *search* and *remove* run in $O(n/m)$

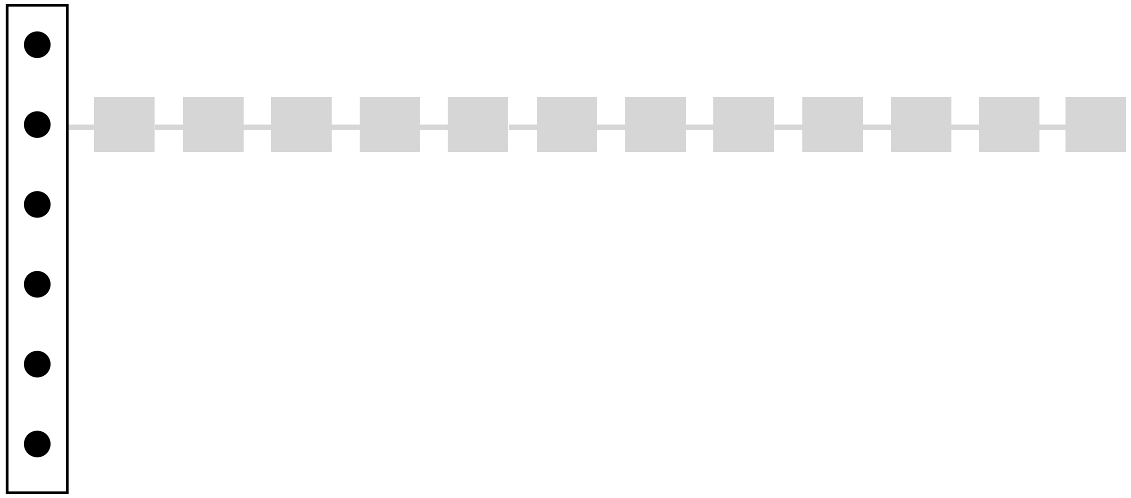
Assumption 2. n is not much larger or much smaller than m.

Under this assumption, n/m is a small constant, which means that $O(n/m) = O(1)$

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

If not true, chains can become very long (of length n in the worst case).



When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

If not true, chains can become very long (of length n in the worst case).

Not guaranteed to be true, but true in many practical applications.

Examples.

✓ Hashing phone numbers of PSUT students.

✓ Hashing birth days (day and month) of PSUT students.

X Hashing timestamps of assignment submissions across a year.
clustered around certain hours of the day

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

If not true, chains can become very long (of length n in the worst case).

Not guaranteed to be true, but true in many practical applications.

Examples.

✓ Hashing phone numbers of PSUT students.
✓ Hashing birth days (day and month) of PSUT students.

✗ Hashing timestamps of assignment submissions across a year.
clustered around certain hours of the day

Denial of Service Attacks

If an adversary has enough information about your hash function and hash table, they can send a large set of carefully chosen elements that hash to the same chain. This will heavily degrade the performance of the hash table!

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

- If not true, chains can become very long (of length n in the worst case).
- Not guaranteed to be true, but true in many practical applications.
- If true, *search* and *remove* run in $O(n/m)$

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

- If not true, chains can become very long (of length n in the worst case).
- Not guaranteed to be true, but true in many practical applications.
- If true, *search* and *remove* run in $O(n/m)$

Assumption 2.
n is not much larger or much smaller than m.

- If $m \gg n$: wasted space
  If $m \ll n$: very long chains

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

- If not true, chains can become very long (of length n in the worst case).
- Not guaranteed to be true, but true in many practical applications.
- If true, *search* and *remove* run in $O(n/m)$

Assumption 2.
n is not much larger or much smaller than m.

- If $m \gg n$: wasted space
  If $m \ll n$: very long chains
- Can be guaranteed by resizing the table up/down to keep m around $\frac{1}{4}n$.

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

- If not true, chains can become very long (of length n in the worst case).
- Not guaranteed to be true, but true in many practical applications.
- If true, *search* and *remove* run in $O(n/m)$

Assumption 2.
n is not much larger or much smaller than m.

- If $m \gg n$: wasted space
If $m \ll n$: very long chains
- Can be guaranteed by resizing the table up/down to keep m around $\frac{1}{4}n$.
- If true, $O(n/m) = O(1)$

When do hash tables perform well?

Assumption 1.
Elements are *distributed uniformly* in the table.

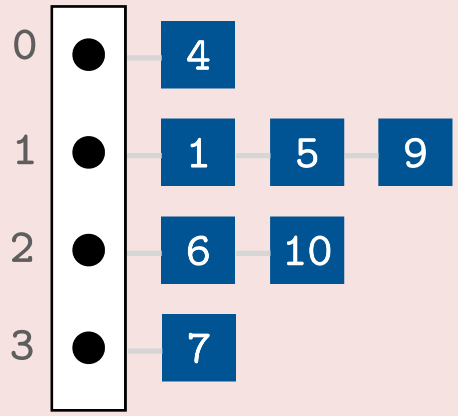
- If not true, chains can become very long (of length n in the worst case).
- Not guaranteed to be true, but true in many practical applications.
- If true, *search* and *remove* run in $O(n/m)$

Assumption 2.
n is not much larger or much smaller than m.

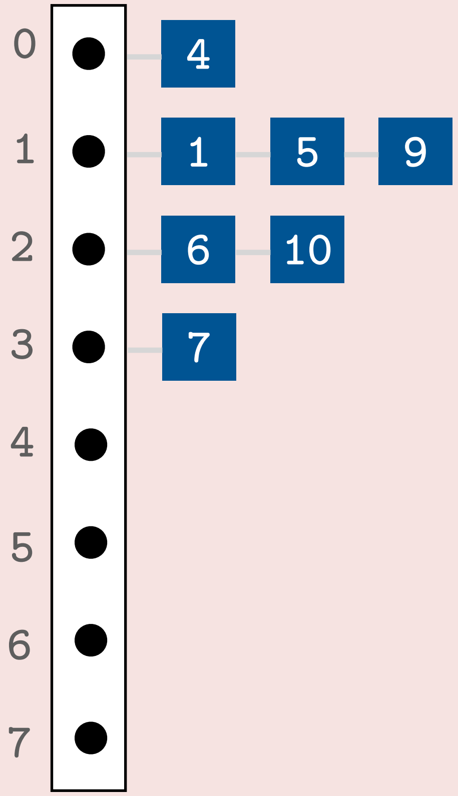
- If $m \gg n$: wasted space
  If $m \ll n$: very long chains
- Can be guaranteed by resizing the table up/down to keep m around $\frac{1}{4}n$.
- If true, $O(n/m) = O(1)$

✓ Conclusion. Hash tables implemented with separate chaining perform the `insert`, `search` and `remove` operations in $O(1)$ assuming the load factor is a small constant and the elements are distributed uniformly across the chains in the table.

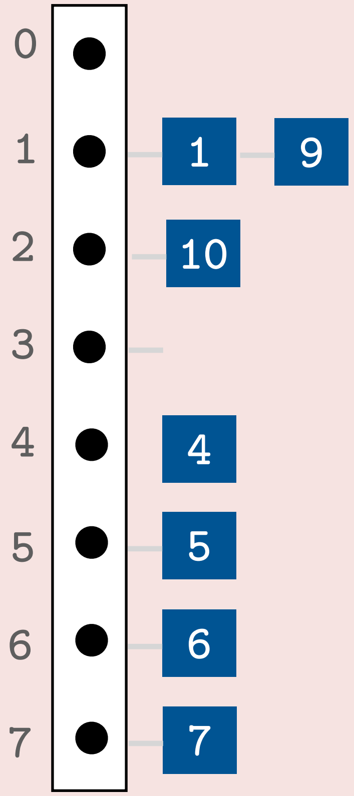
Exercise. Resizing Hash Tables



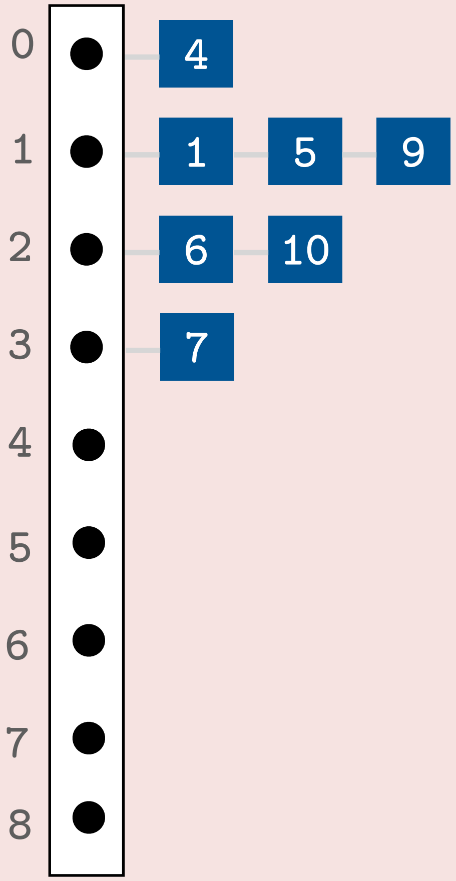
How does the above hash table look like after resizing it to become of size m=8?



A

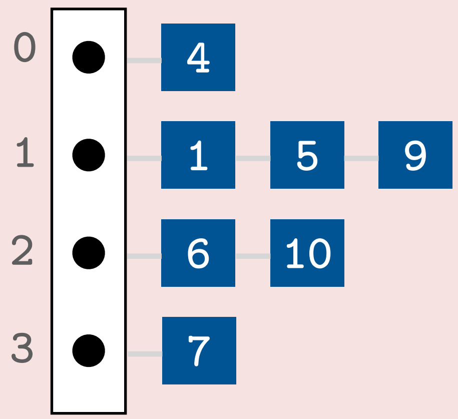


B

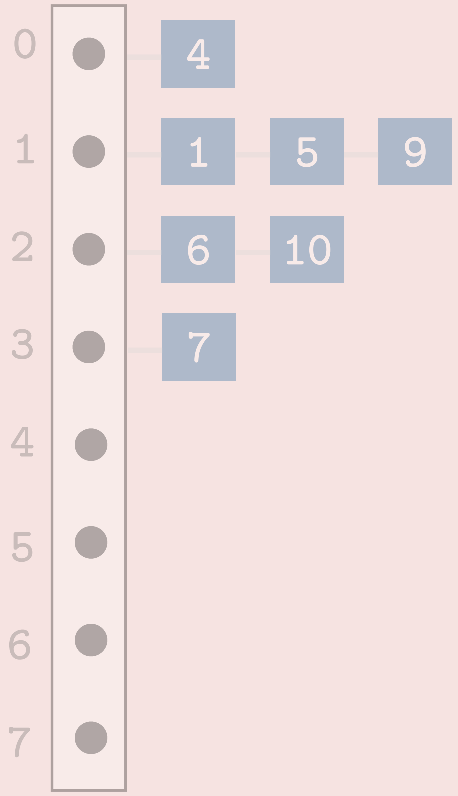


C

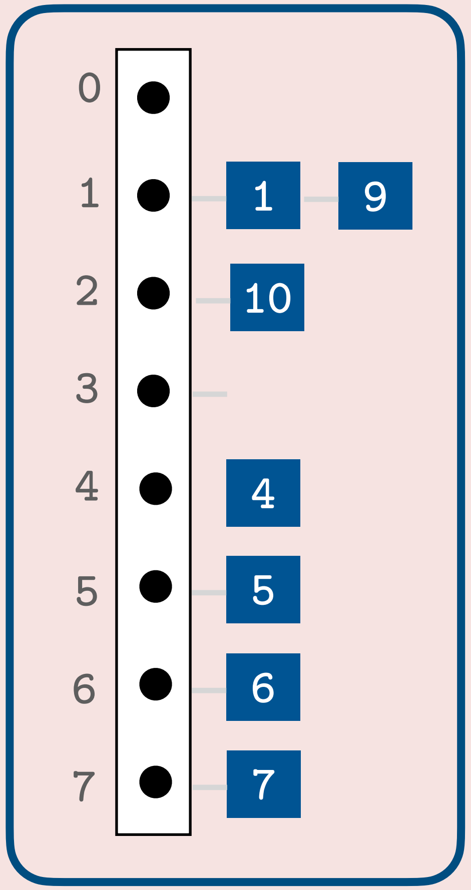
Exercise. Resizing Hash Tables



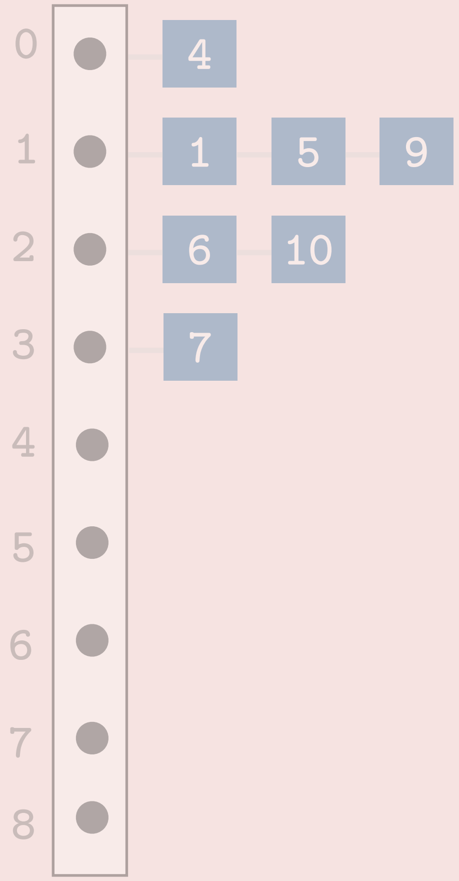
How does the above hash table look like after resizing it to become of size m=8?



A



B

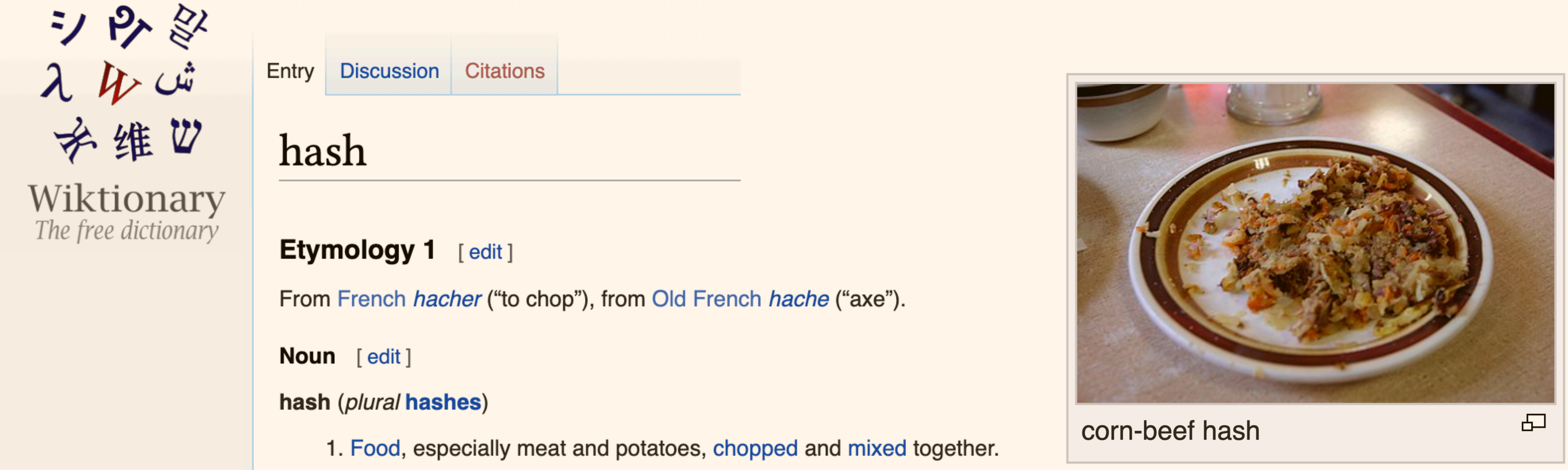


C

All items need to be rehashed after resizing the table!

Coding Demo

What's in a Name?



Wiktionary
The free dictionary

Entry Discussion Citations

hash

Etymology 1 [edit]

From French *hacher* (“to chop”), from Old French *hache* (“axe”).

Noun [edit]

hash (*plural* **hashes**)

1. Food, especially meat and potatoes, chopped and mixed together.

corn-beef hash

Hatchet (English)
Hache (French)

Chopped parsley (English)
Persil hachée (French)

Chopped cilantro (English)
Coriandre hachée (French)

Ground meet (English)
Viande hachée (French)

Coding Interview Question

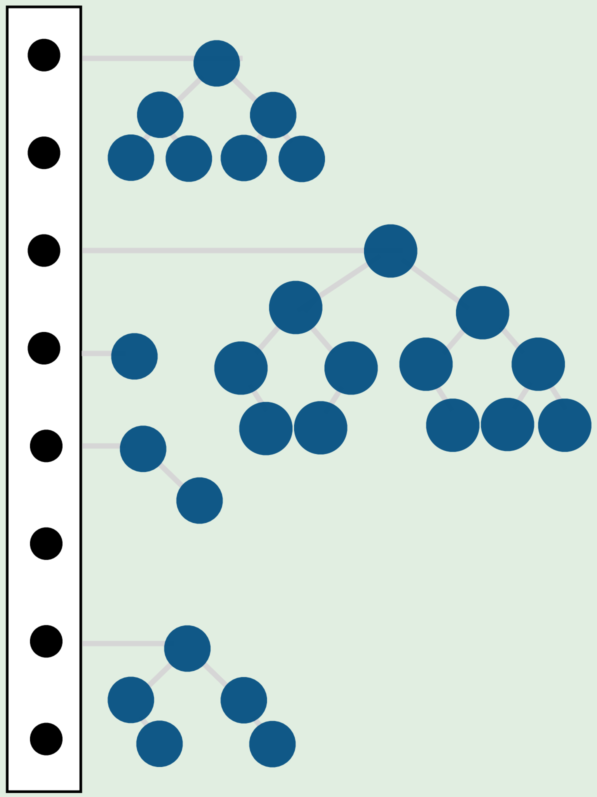
Design a data structure that supports `insert`, `search` and `remove` in $O(\log n)$ in the worst case and in $O(1)$ in most practical applications.

Coding Interview Question

Design a data structure that supports `insert`, `search` and `remove` in $O(\log n)$ in the worst case and in $O(1)$ in most practical applications.

Answer.

Use separate chaining with AVL trees instead of singly linked lists!

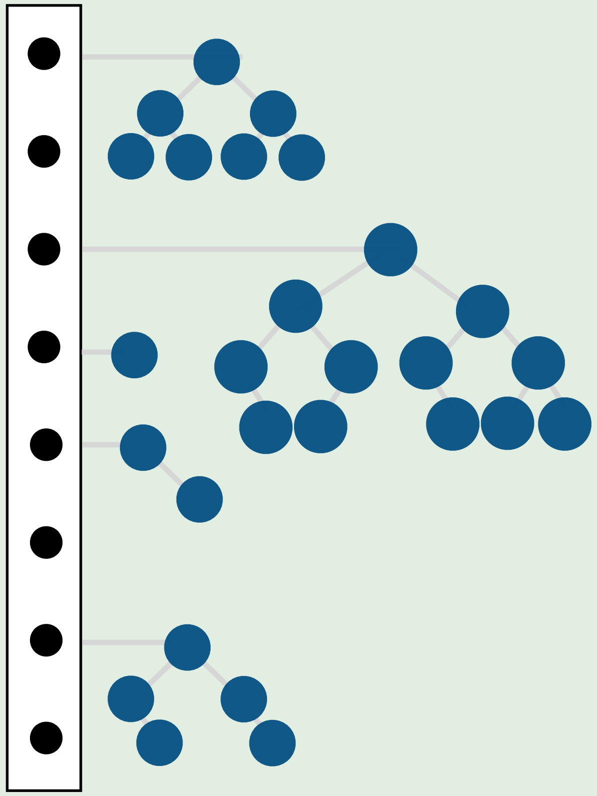


Coding Interview Question

Design a data structure that supports `insert`, `search` and `remove` in $O(\log n)$ in the worst case and in $O(1)$ in most practical applications.

Answer.

Use separate chaining with AVL trees instead of singly linked lists!



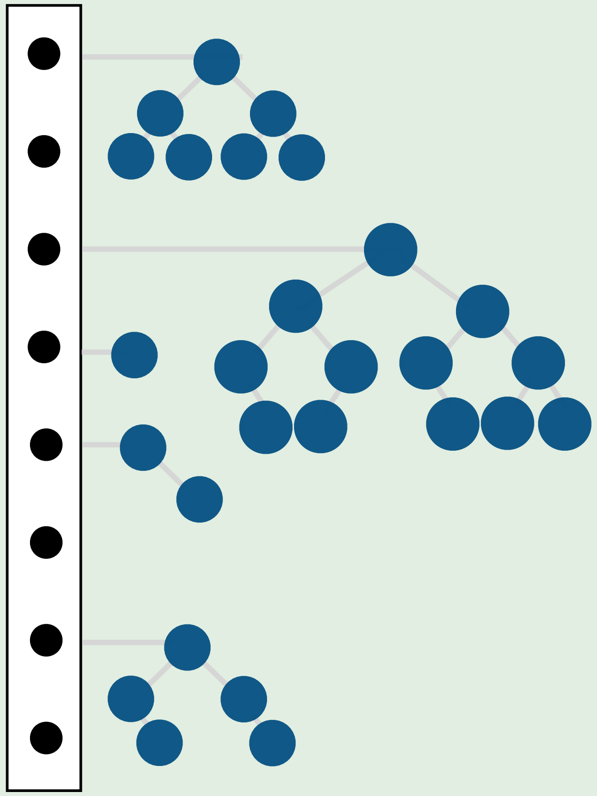
🤔 Any reason to use singly-linked lists for chaining instead of AVL trees?

Coding Interview Question

Design a data structure that supports `insert`, `search` and `remove` in $O(\log n)$ in the worst case and in $O(1)$ in most practical applications.

Answer.

Use separate chaining with AVL trees instead of singly linked lists!



🤔 Any reason to use singly-linked lists for chaining instead of AVL trees?

- Singly-linked lists are simpler and require less memory than AVL trees.
- They also can be faster than AVL trees if the number of elements they store is very small.
- BSTs require a definition of order (<, > and ==), whereas linked lists require only a definition for equality.

Java's hash table implementation uses linked lists. However, if a chain's length exceeds a certain threshold, the chain is converted to a balanced BST.

Hashing Strings

How can strings be hashed?

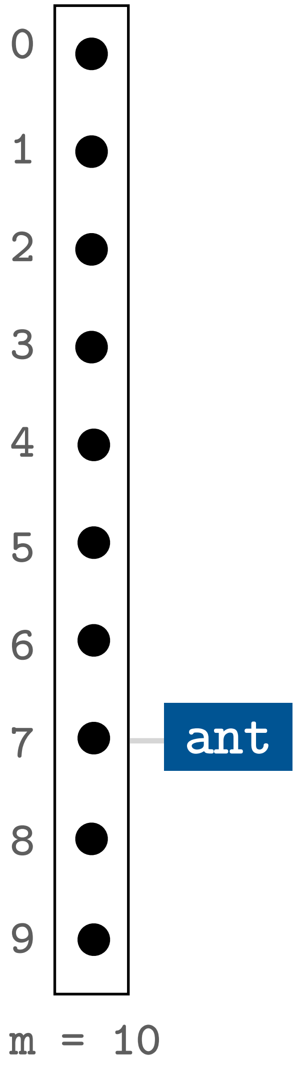
Hashing Strings

How can strings be hashed?

Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant" → 97`



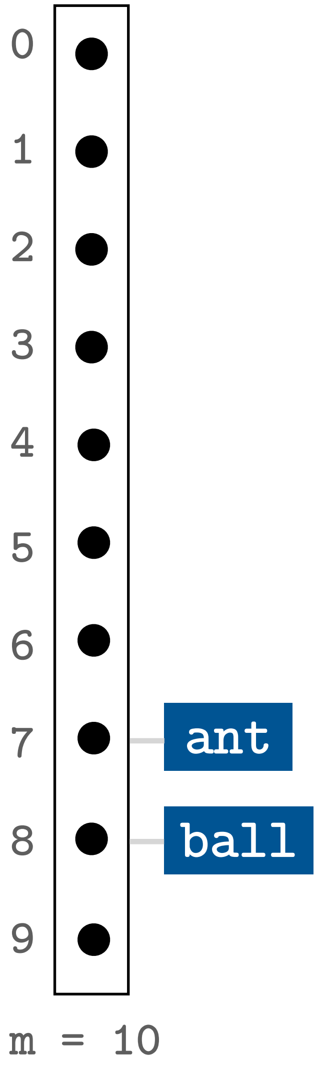
Hashing Strings

How can strings be hashed?

Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant" → 97, "ball" → 98`



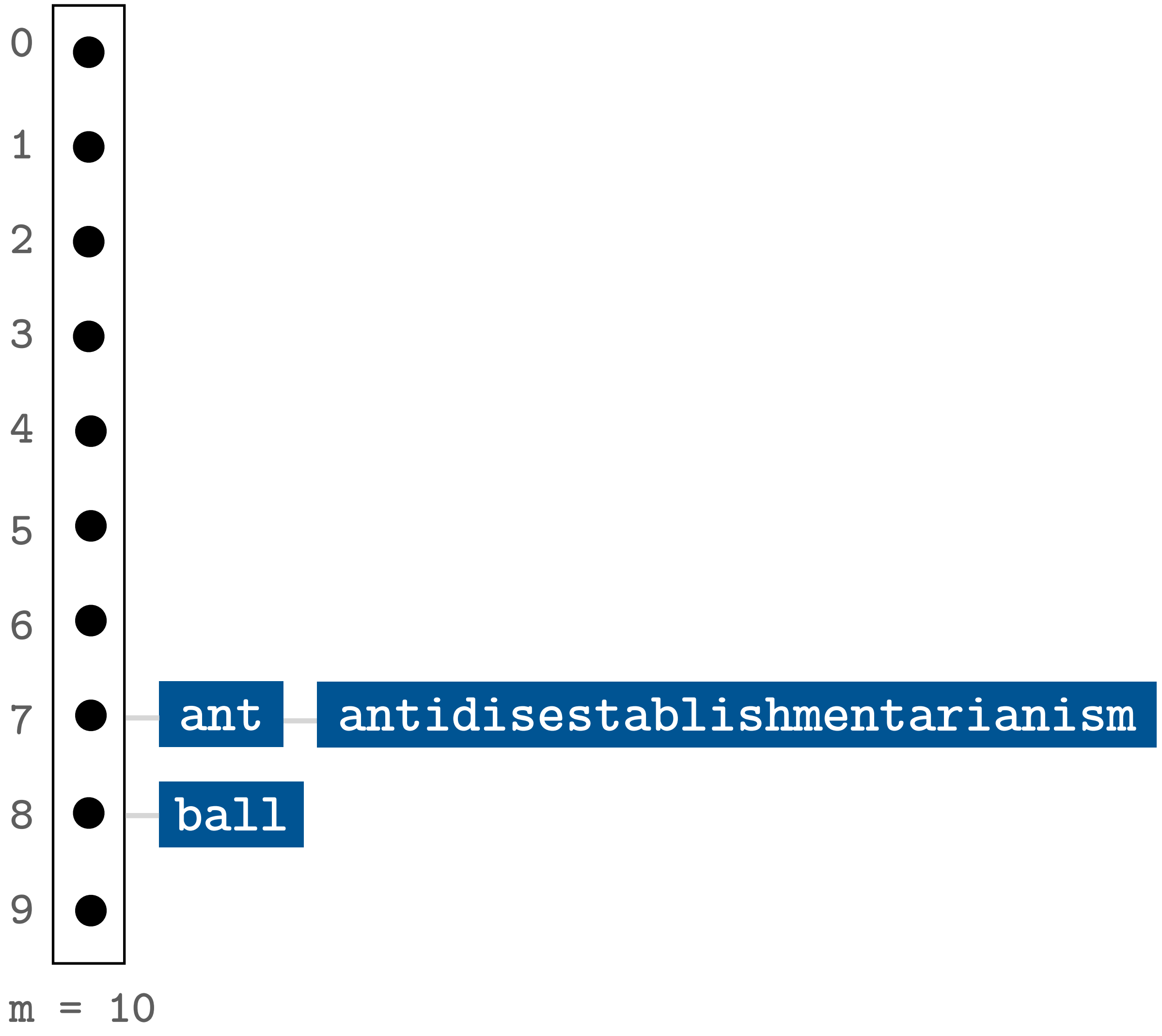
Hashing Strings

How can strings be hashed?

Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant"` → `97`, `"ball"` → `98`, `"antidisestablishmentarianism"` → `97`.



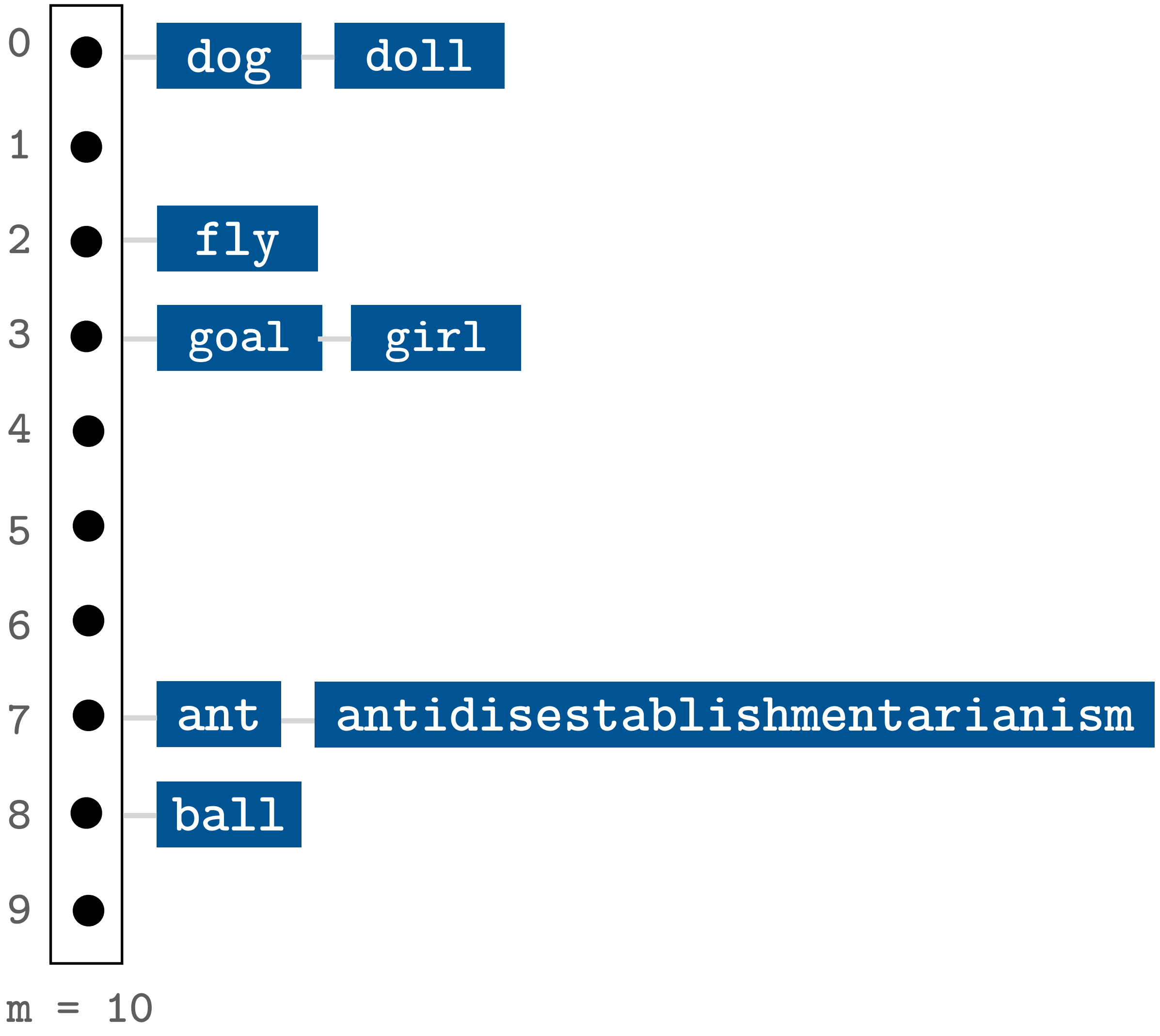
Hashing Strings

How can strings be hashed?

Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant"` → `97`, `"ball"` → `98`, `"antidisestablishmentarianism"` → `97`.
`"dog"` → `100`, `"doll"` → `100`, `"fly"` → `102`, `"goal"` → `103`, `"girl"`→ `103`



Hashing Strings

How can strings be hashed?

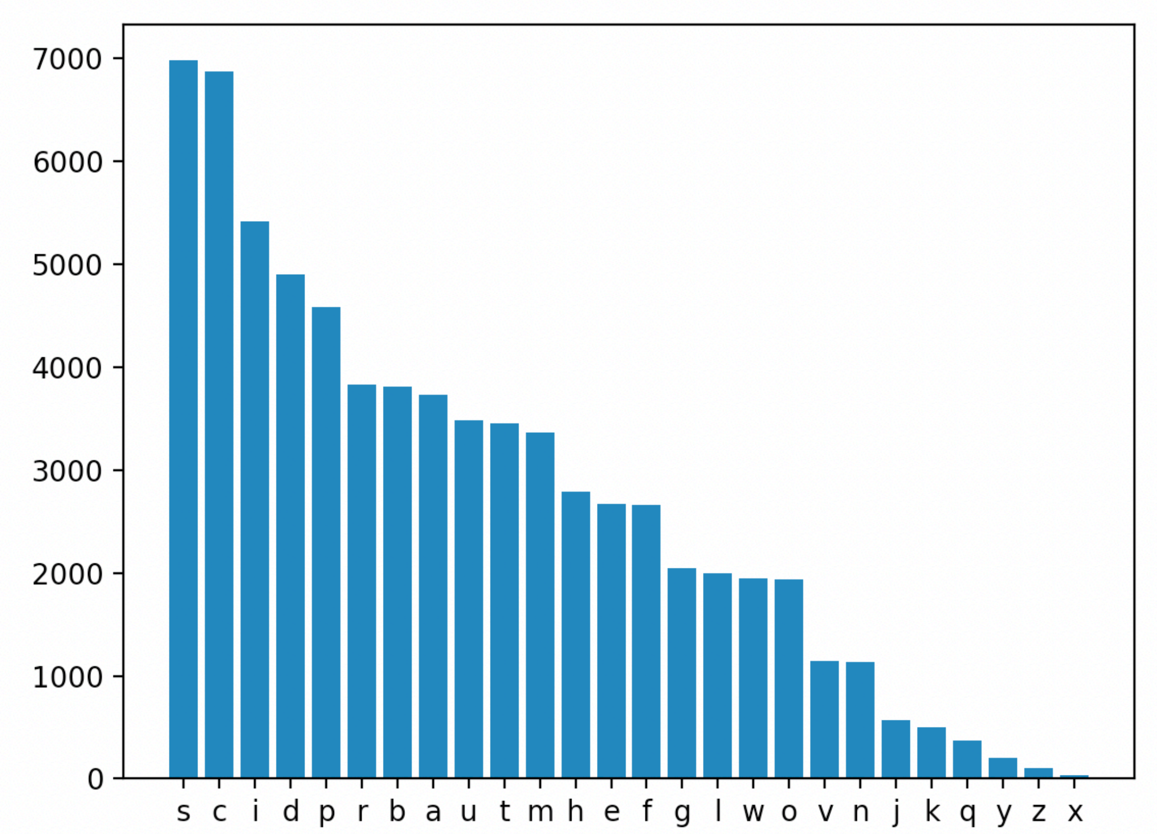
Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant"` → `97`, `"ball"` → `98`, `"antidisestablishmentarianism"` → `97`.

Problem. The hashed strings are unlikely to be uniformly distributed in the table.

1. The distribution of first character frequencies is not uniform in the English language and in many practical applications.



First letter frequencies in an English dictionary

Hashing Strings

How can strings be hashed?

Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant"` → `97`, `"ball"` → `98`, `"antidisestablishmentarianism"` → `97`.

Problem. The hashed strings are unlikely to be uniformly distributed in the table.

1. The distribution of first character frequencies is not uniform in the English language.
2. There will be a very limited number of chains used (e.g. 26, 52, 127 or 256) regardless of the table size.

Hashing Strings

How can strings be hashed?

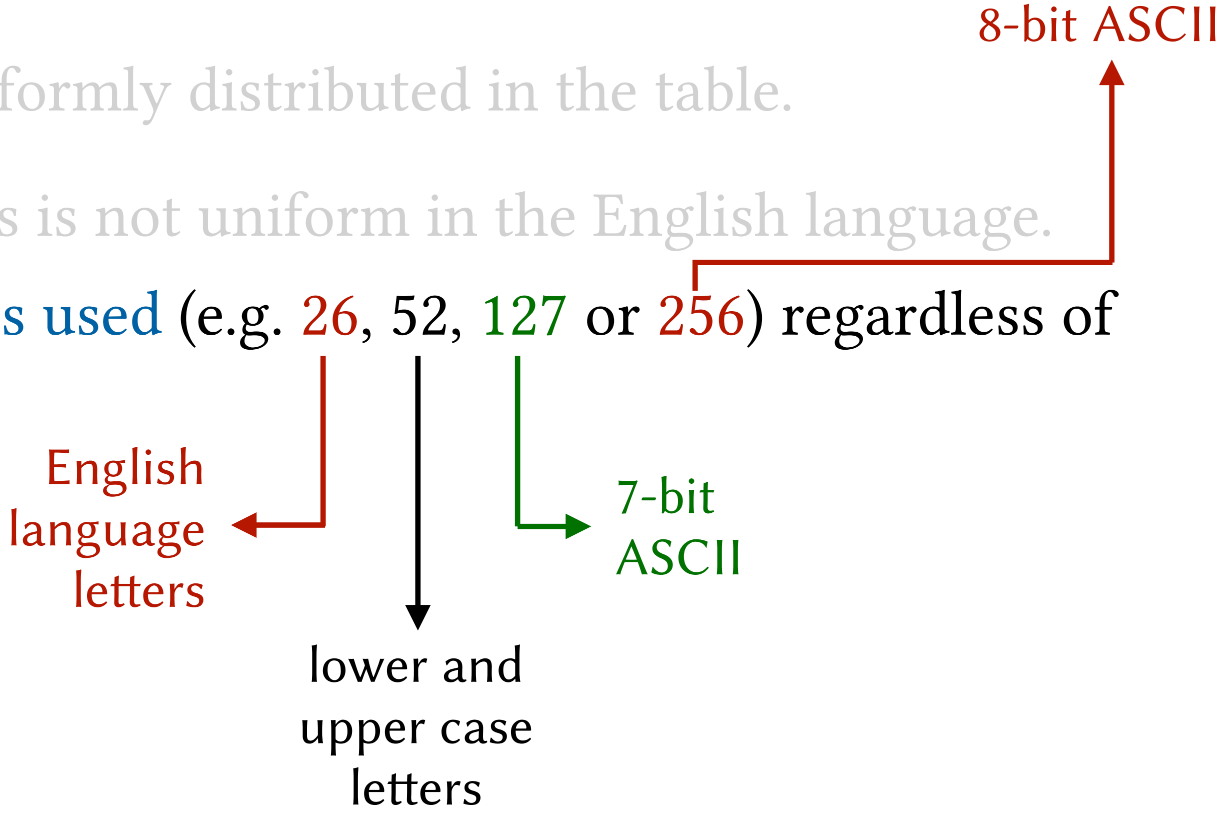
Solution # 1.

Convert to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant" → 97, "ball" → 98, "antidisestablishmentarianism" → 97.`

Problem. The hashed strings are unlikely to be uniformly distributed in the table.

1. The distribution of first character frequencies is not uniform in the English language.
2. There will be a very limited number of chains used (e.g. 26, 52, 127 or 256) regardless of the table size.



8-bit ASCII

English language letters

lower and upper case letters

7-bit ASCII

Hashing Strings

How can strings be hashed?

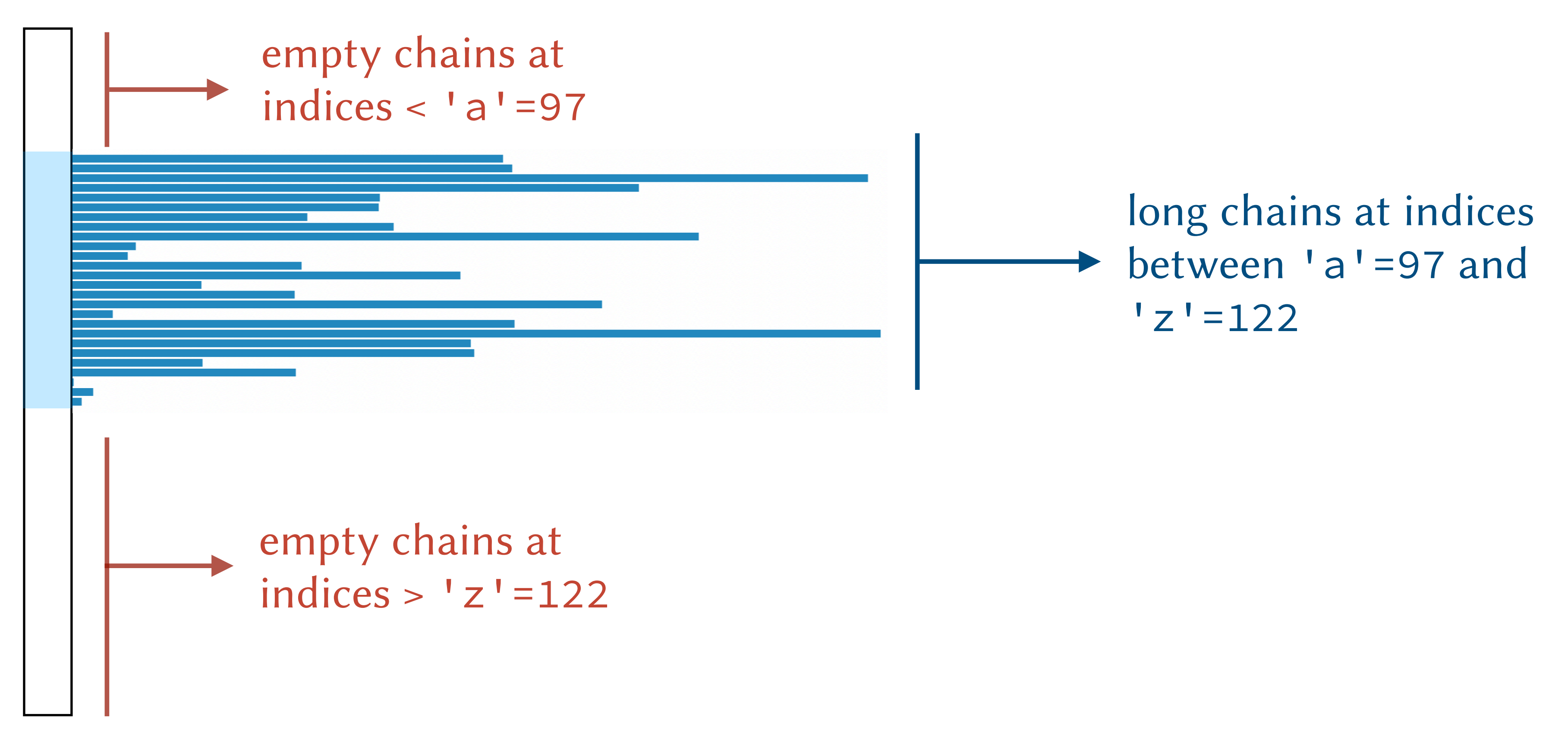
Solution # 1.

Convert the string to an integer using the `ASCII` value of the first character of the string.

Examples. `"ant" → 97, "ball" → 98, "antidisestablishmentarianism" → 97.`

Problem. The hashed strings are unlikely to be uniformly distributed in the table.

1. The distribution of first character frequencies is not uniform in the English language.
2. There will be a very limited number of chains used (e.g. 26, 52, 127 or 256) regardless of the table size.



An illustration of chains in a hash table storing dictionary words based on their first character

Hashing Strings

How can strings be hashed?

Solution # 2.

Convert to an integer by summing the `ASCII` values of all the characters in the string.

Example. `"a"` → `97`, `"am"` → `97+155=252`, `"ant"` → `97+156+164=417`, etc.

Hashing Strings

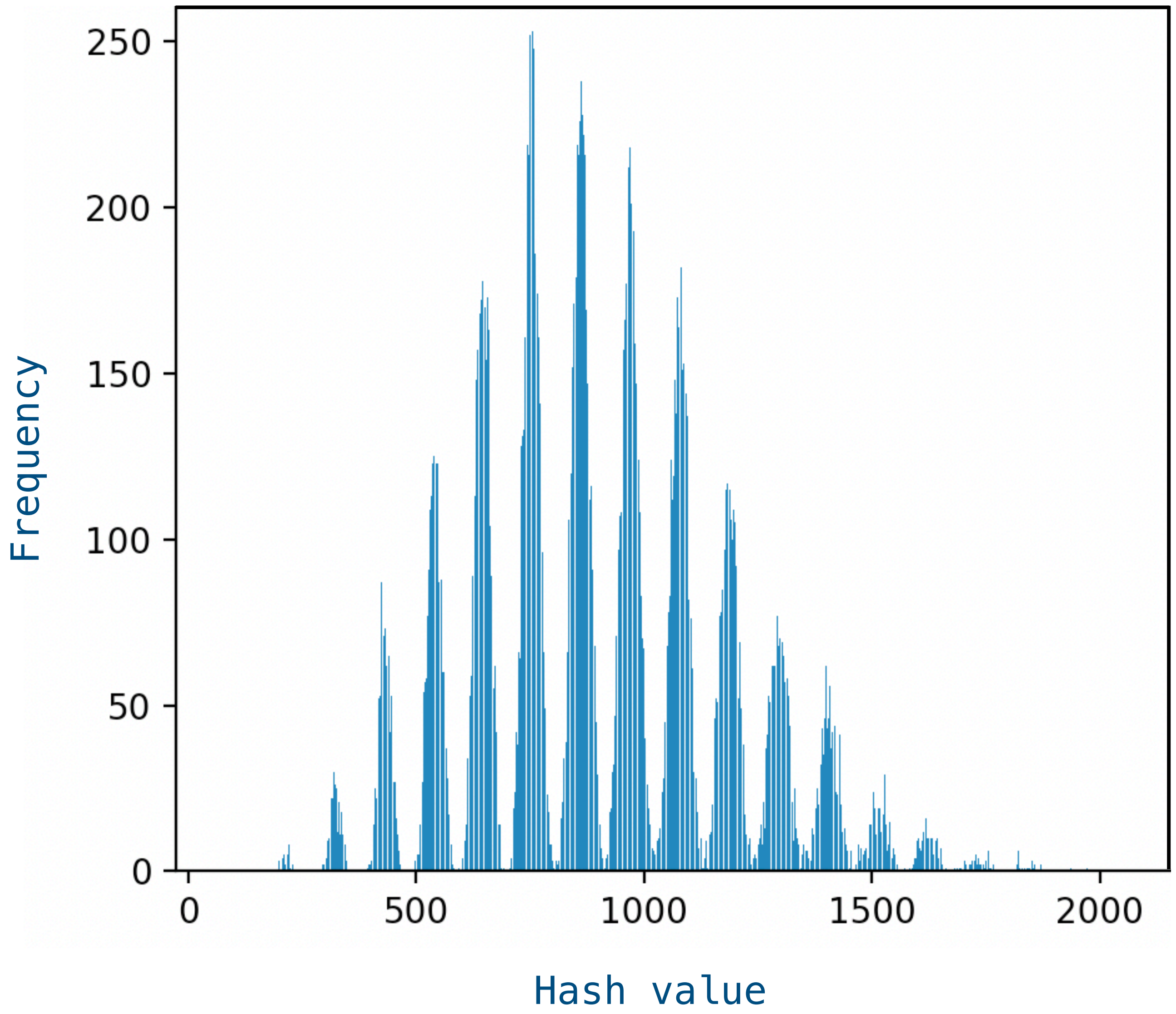
How can strings be hashed?

Solution # 2.

Convert to an integer by summing the `ASCII` values of all the characters in the string.

Example. `"a"` → `97`, `"am"` → `97+155=252`, `"ant"` → `97+156+164=417`, etc.

Problem. In many applications, some hash values are much more likely to occur than others.



Frequency of hash values of words in the dictionary

Hashing Strings

How can strings be hashed?

Solution # 2.

Convert to an integer by summing the `ASCII` values of all the characters in the string.

Example. `"a" → 97,` `"am" → 97+155=252,` `"ant" → 97+156+164=417,` etc.

Problem. In many applications, some hash values are much more likely to occur than others.

Problem. Very different strings get the same integer value (many collisions). For example:

int value	strings
394	All permutations of "abcd" (e.g. abdc, acdb, acbd, adbc, etc.)
455	snow, soup, tusk, suez, winy
456	guys, lust, rots, runs, sort, sums, town, twit
574	wormy, stunt, puppy, tutor
796	pursuit, puzzler, stylist, sunspot, uproots
900	portrays, pronouns, protests, robustly, textures, typhoons
1120	interrupts, introverts, oppressors, repository, transports
1726	multidimensional, terminologically, unaccountability

Hashing Strings

How can strings be hashed?

Solution # 2.

Convert to an integer by summing the `ASCII` values of all the characters in the string.
Example. `"a" → 97,` `"am" → 97+155=252,` `"ant" → 97+156+164=417,` etc.

Problem. In many applications, some hash values are much more likely to occur than others.

Problem. Very different strings get the same integer value (many collisions). For example:

int value	strings
394	All permutations of "abcd" (e.g. abdc, acdb, acbd, adbc, etc.)
455	snow, soup, tusk, suez, winy
456	guys, lust, rots, runs, sort, sums, town,
574	wormy, stunt, puppy, tutor
796	pursuit, puzzler, stylist, sunspot, uproots
900	portrays, pronouns, protests, robustly, textures, typhoons
1120	interrupts, introverts, oppressors, repository, transports
1726	multidimensional, terminologically, unaccountability

Goal
Different strings get different integer values

Hashing Strings

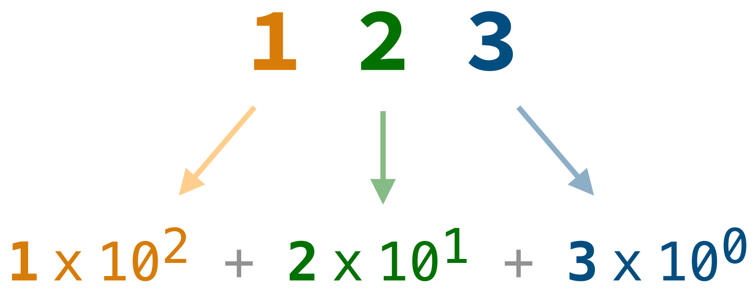
How can strings be hashed?

Solution # 3.

Assign weights to the characters based on their position in the string and compute a weighted sum of the ASCII values of the characters.

Decimal System:

radix = 10



1 2 3

$1 \times 10^{2} + 2 \times 10^{1} + 3 \times 10^{0}$

Hashing Strings

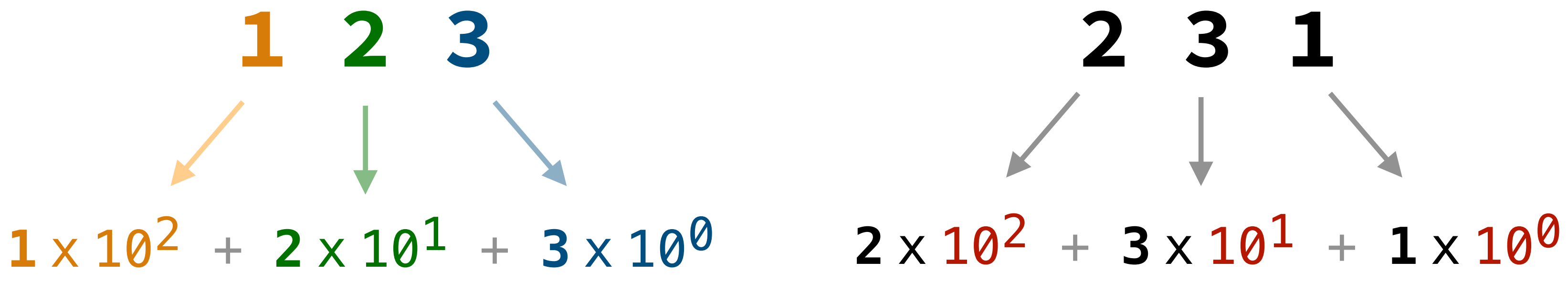
How can strings be hashed?

Solution # 3.

Assign weights to the characters based on their position in the string and compute a weighted sum of the `ASCII` values of the characters.

Decimal System:

radix = 10



1 2 3

$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$

2 3 1

$2 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$

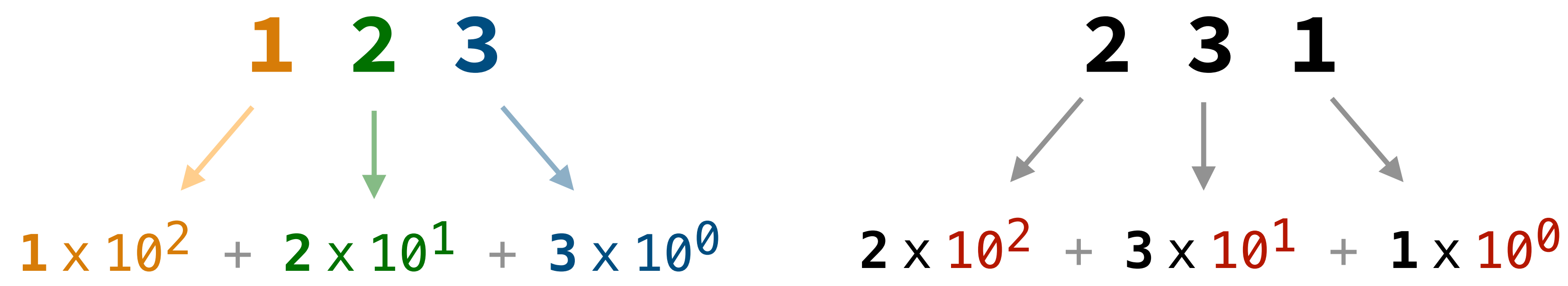
Hashing Strings

How can strings be hashed?

Solution # 3.

Assign weights to the characters based on their position in the string and compute a weighted sum of the `ASCII` values of the characters.

Decimal System:
radix = 10



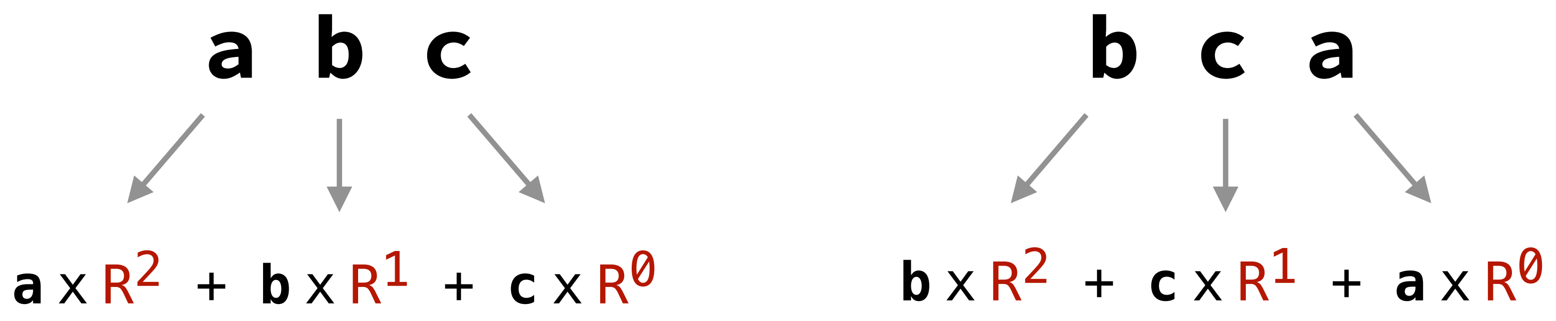
1 2 3

$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$

2 3 1

$2 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$

A Positional System for Characters:
pick some radix **R**



a b c

$a \times R^2 + b \times R^1 + c \times R^0$

b c a

$b \times R^2 + c \times R^1 + a \times R^0$

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Go through the characters right to left

Example.

```
                   i
hash_value(" A B C D ")

sum = 0
R   = 1
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Multiply the character by R

Example.

```
                          i
hash_value(" A B C D ")

sum = (1*D)
R   = 1
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Increase the exponent of R for the next iteration
(multiply R by 26)

Example.

```
                   i
hash_value(" A B C D ")

sum = (1*D)
R   = 1*26
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Multiply the character by R

Example.

```
                  i
hash_value(" A B C D ")

sum = (1*D) + (26*C)
R   = 1*26
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Increase the exponent of R for the next iteration
(multiply R by 26)

Example.

```
                   i
hash_value(" A B C D ")

sum = (1*D) + (26*C)
R   = 1*26*26
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
         sum += R*str[i];
         R *= 26;
    }

    return sum % m;
}
```

Multiply the character by R

```
                         i
Example.  hash_value(" A B C D ")

          sum = (1*D) + (26*C) + (26^2*B)
          R   = 1*26*26
```

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
         sum += R*str[i];
         R *= 26;
    }

    return sum % m;
}
```

Increase the exponent of R for the next iteration

(multiply R by 26)

```
                       i
Example.   hash_value(" A B C D ")
```

sum = (1*D) + (26*C) + (26^2*B)

R = 1*26*26*26

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
        sum += R*str[i];
        R *= 26;
    }

    return sum % m;
}
```

Multiply the character by R

i

Example. hash_value(" A B C D ")

sum = (1*D) + (26*C) + (26^2*B) + (26^3*A)

R = 1*26*26*26

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
         sum += R*str[i];
         R *= 26;
    }

    return sum % m;
}
```

i

Example. **hash_value**(" A B C D ")

sum = (1*D) + (26*C) + (26^2*B) + (26^3*A)

R = 1*26*26*26*26

A Sketch Implementation (assuming R = 26)

```
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
          sum += R*str[i];
          R *= 26;
    }

    return sum % m;
}
```

R and sum
can **overflow**!

Example.

```
           i
hash_value(" A B C D ")
```

$\text{sum} = (1*D) + (26*C) + (26^2*B) + (26^3*A)$

$R = 1*26*26*26*26$

A Sketch Implementation (assuming R = 26)

```cpp
int hash_value(string & str) {
    int sum=0, R=1;

    for (int i=str.length()-1; i>=0; i--) {
         sum += R*str[i];
         R *= 26;
    }

    return sum % m;
}
```

R and sum
can **overflow**!

```cpp
int hash_value(string & str) {
    int sum=0, R=26;

    for (int i=0; i<str.length(); i++)
         sum = (sum*R + str[i]) % m;

    return abs(sum);
}
```

No overflow!
(assuming m is not too large)

A Sketch Implementation (assuming R = 26)

```
int has_value(string & str) {
    int sum=0, R=26;

    for (int i=0; i<str.length(); i++)
        sum = (sum*R + str[i]) % m;

    return abs(sum);
}
```

Go through the characters left to right

A Sketch Implementation (assuming R = 26)

```
int has_value(string & str) {
    int sum=0, R=26;

    for (int i=0; i<str.length(); i++)
        sum = (sum*R + str[i]) % m;

    return abs(sum);
}
```

Each iteration in the loop multiplies the sum by R and adds one character.

This is similar to how 9375 in decimal (for example) can be computed:

```
sum = 0
sum = sum * 10 + 9 = 9
sum = sum * 10 + 3 = 93
sum = sum * 10 + 7 = 937
sum = sum * 10 + 5 = 9375
```

A Sketch Implementation (assuming R = 26)

```
int has_value(string & str) {
    int sum=0, R=26;

    for (int i=0; i<str.length(); i++)
          sum = (sum*R + str[i]) % m;

    return abs(sum);
}
```

$(x_1 + x_2 + x_3 + \ldots + x_n) \% m$

is equivalent to:

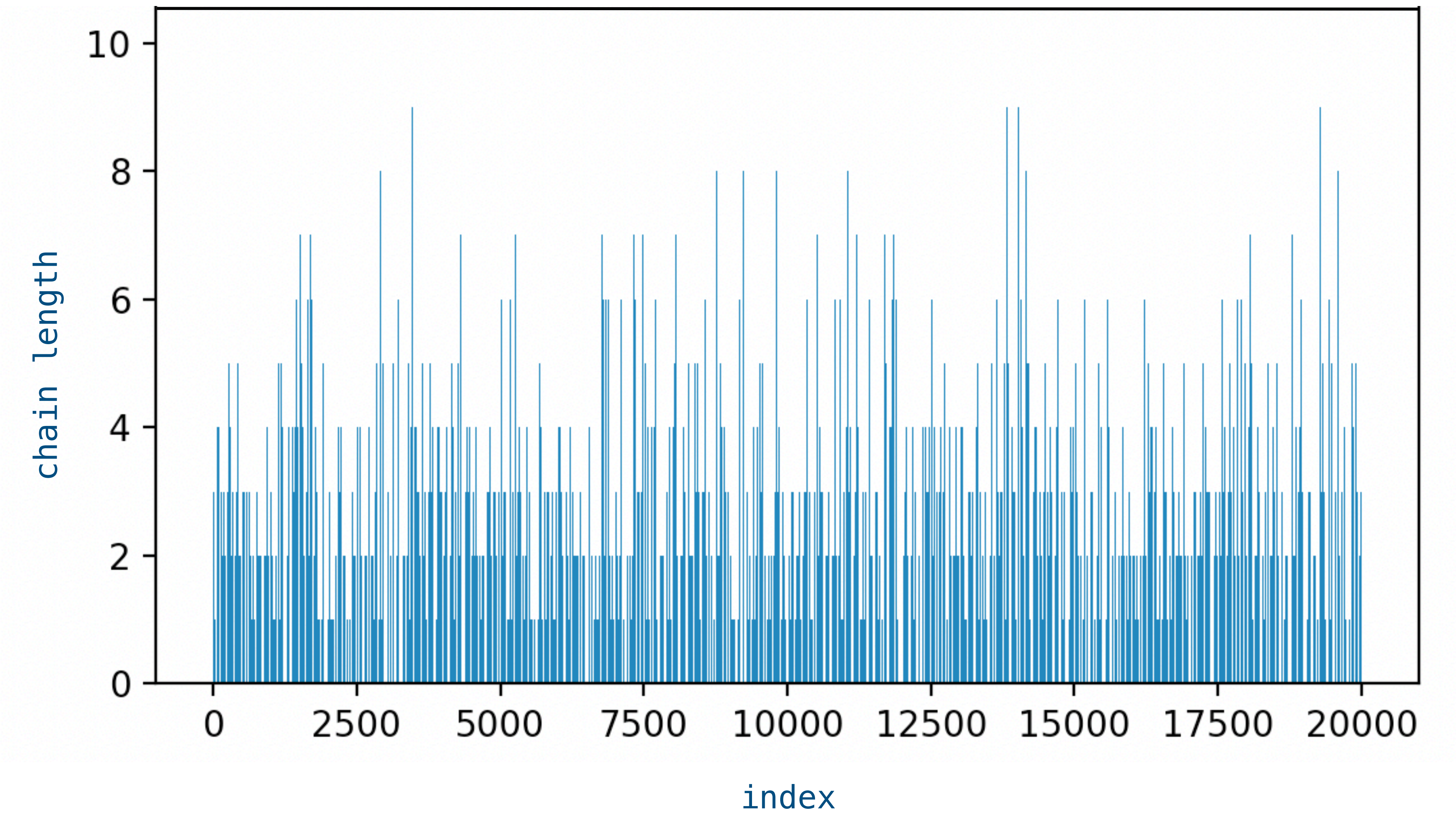
$((x_1 \% m) + x_2) \% m) + x_3) \% m \ldots + x_n) \% m$

Example:

```
(5 + 6 + 23) % 10 = 34 % 10  = 4

(((5 % 10) + 6) % 10) + 23) % 10 =
(((5     ) + 6) % 10) + 23) % 10 =
((11          ) % 10) + 23) % 10 =
((1                 ) + 23) % 10 =
(24                       ) % 10 = 4
```

Hashing Strings



Result of hashing words from the dictionary
(n=70566) into a hash table with m=20000 chains
(using R=31)

Hash Tables vs Balanced BSTs

Asymptotic Analysis

	insert		remove		search	
	average	worst	average	worst	average	worst
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`
Hash Table with Separate Chaining	`O(1)`	`O(1)`	`O(1)`	`O(n)`	`O(1)`	`O(n)`

Under reasonable assumptions

Hash Tables vs Balanced BSTs

Asymptotic Analysis

	`insert`		`remove`		`search`	
	average	worst	average	worst	average	worst
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`	`O(log n)`
Hash Table with Separate Chaining	`O(1)`	`O(1)`	`O(1)`	`O(n)`	`O(1)`	`O(n)`

Experimental Analysis.

Insert, remove and search for `10,000,000` random integers.

	Balanced BST	**Hash Table**
`Insert`	`14.6784 sec`	`6.11673 sec`
`Search`	`13.2523 sec`	`3.25825 sec`
`Remove`	`16.5524 sec`	`5.39692 sec`

Notes. Tests were performed using the `C++ STL set` container as the balanced BST and the `C++ STL unordered_set` container as the hash table. Each insert operation performs a search for the element before inserting it to avoid duplicates.
(Using a MacBook Pro with `2.6 GHz 6-Core Intel Core i7` and `16 GB DDR4 RAM`)

✓ Hash tables are faster on average but do not guarantee good performance for all applications. A balanced BST is typically slightly slower but is guaranteed not to perform badly.

Hash Tables vs Balanced BSTs

Asymptotic Analysis

	insert		remove		search	
	average	worst	average	worst	average	worst
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table with Separate Chaining	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)

Experimental Analysis.

Insert, remove and search for `10,000,000` random integers.

	Balanced BST	**Hash Table**
Insert	14.6784 sec	6.11673 sec
Search	13.2523 sec	3.25825 sec
Remove	16.5524 sec	5.39692 sec

Other Factors.

Hash tables do not support ordered operations efficiently like BSTs (e.g. `max()`, `min()`, `median()`, `count_less_than(x)`, `smallest_above(x)`, `largest_below(x)`, etc.)

Finding the max!

```
template <class T>
T HashTable<T>::max() const {
    if (is_empty())
        throw "Attempting to get the max from an empty table."

    DLLNode<T>* max_node = nullptr;
    for (int i = 0; i < m; i++) {
        DLLNode<T>* c = table[i].head_node();
        while (c != nullptr) {
            if      (max_node == nullptr)                max_node = c;
            else if (c->get_val() > max_node->get_val()) max_node = c;
            c = c->get_next();
        }
    }

    return max_node->get_val();
}
```

Finding the max!

```
template <class T>
T HashTable<T>::max() const {
    if (is_empty())
        throw "Attempting to get the max from an empty table."

    DLLNode<T>* max_node = nullptr;
    for (int i = 0; i < m; i++) {
        DLLNode<T>* c = table[i].head_node();
        while (c != nullptr) {
            if      (max_node == nullptr)                max_node = c;
            else if (c->get_val() > max_node->get_val()) max_node = c;

            c = c->get_next();
        }
    }

    return max_node->get_val();
}
```

go through every chain
in the table.

go through every node
in that chain

Running Time. $O(n)$ Data compares

$O(n + m)$ Total amount of work.

Even if the table is empty, the code still creates a pointer for every empty chain!

OPTIONAL

djb2 String Hash Function

```
int hash(char * str) {
    int sum = 5381;
    int c;

    while (c = *str++)
        sum  = ((sum << 5)+sum) + c;

    return (sum & 0x7fffffff) % m;
}
```

djb2 String Hash Function

```c
int hash(char * str) {
    int sum = 5381;
    int c;

    while (c = *str++)
        sum  = ((sum << 5)+sum) + c;

    return (sum & 0x7fffffff) % m;
}
```

A random prime seed for the first cycle.

Could have been set to 0 or to another value, but this was found experimentally to produce a good distribution of hash values.

Loop through the characters from left to right

`sum << 5` ≡ `sum * 32`
Adding `sum` again makes it equivalent to `sum * 33`

Shifting and adding is faster than multiplying

`33` was found experimentally to distribute the hash values well.

Equivalent to (but faster than) returning `abs(sum) % m`

Assuming `int` is 32 bits

Hashing Other Than Integers and Strings

Floating point numbers. Given floating point numbers between MIN and MAX, the numbers can be normalized to be between 0 and 1 and then multiplied by the number of chains:

```
int hash(float x) {
    return abs((x-MIN) / (MAX-MIN) * m);
}
```

Composite types. Hashing an array, a user defined object or any composite type can be done using the same logic as that of the **djb2** algorithm:

```
sum = 0
sum = sum * 33 + hash(1st element)
sum = sum * 33 + hash(2nd element)
sum = sum * 33 + hash(3rd element)
```

etc.

The elements can be array elements or data members in a class or a struct.

Picking a Good Hash Table Size

If the hashed keys are random, then any hash table size m that is around $\frac{1}{4}n$ should be fine.

If the hashed keys might follow a pattern, then care must be taken when choosing the table size.

Examples.

- If the hash table size is **`m=12`** and all the hashed keys are **even** numbers, only half of the chains will be used no matter how many keys are hashed.
  (`0%12=0`, `2%12=2`, `4%12=4`, `6%12=6`, `8%12=8`, `10%12=10`, `12%12=0`, `14%12=2`, `16%12=4`, etc.)

- If the hash table size is **`m=2`**$^{\mathbf{x}}$, then only the **least significant x bits** will play a role in determining the chain indices.

- Using a **prime** number for the hash table size guards against such issues.

The **`GCC`** maintains the following **precomputed** array of hash table sizes that are prime and as close as possible to powers of 2:

```
[7,          13,        31,        61,         127,        251,        509,
 1021,       2039,      4093,      8191,       16381,      32749,      65521,
 131071,     262139,    524287,    1048573,    2097143,    4194301,    8388593,
 16777213,   33554393, 67108859, 134217689, 268435399, 536870909,
 1073741789, 2147483647]
```