ساختمان داده و الگوریتم ها

مبحث هفدهم:
درهم سازی

سجاد شیرعلی شهرضا

پاییز 1402

دوشنبه، 20 آذر 1402

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 11
- انتقال امتحانک سوم به شنبه 2 دی 1402 (در ساعت کلاس، از ساعت 8:15)

جدول درهم سازی

چه عملگرهایی برای ما مهم هستند؟

THE TASK

Again, we want to keep track of objects that have keys $\boxed{5}$ (aka, **nodes** with **keys**)

THE TASK

Again, we want to keep track of objects that have keys 5 (aka, **nodes** with **keys**)

Sorted Arrays

1	2	3	4	5	7

O(n) INSERT/DELETE: first, find the relevant element (via SEARCH) and move a bunch of elements in the array

O(log n) SEARCH: use binary search to see if an element is in A

Linked Lists

HEAD → 3 → 5 → 1 → 4 → 7 → 2

O(1) INSERT: just insert the element at the head of the linked list

O(n) SEARCH/DELETE: since the list is not necessarily sorted, you need to scan the list (delete by manipulating pointers)

BINARY SEARCH TREE PERFORMANCE

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BALANCED BST
SEARCH	O(log(n))	O(n)	O(log(n))
DELETE	O(n)	O(n)	O(log(n))
INSERT	O(n)	O(1)	O(log(n))

HASH TABLE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BALANCED BST	HASH TABLES (HOPEFULLY)
SEARCH	$O(\log(n))$	$O(n)$	$O(\log(n))$	$O(1)$
DELETE	$O(n)$	$O(n)$	$O(\log(n))$	$O(1)$
INSERT	$O(n)$	$O(1)$	$O(\log(n))$	$O(1)$

HASH TABLE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BALANCED BST	HASH TABLES (HOPEFULLY)
SEARCH	O(log(n))	O(n)	O(log(n))	O(1)
DELETE	O(n)	O(n)	O(log(n))	O(1)
INSERT	O(n)	O(1)	O(log(n))	O(1)

What is a *naive* way to achieve these runtimes?

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 1000: 2 4 5 998 999

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 1000: **2** **4** **5** **998** **999**

Reasonable Attempt: ***Direct Addressing!***

(each address/bucket stores one type of item)

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 1000: **2** **4** **5** **998** **999**

Reasonable Attempt: ***Direct Addressing!***

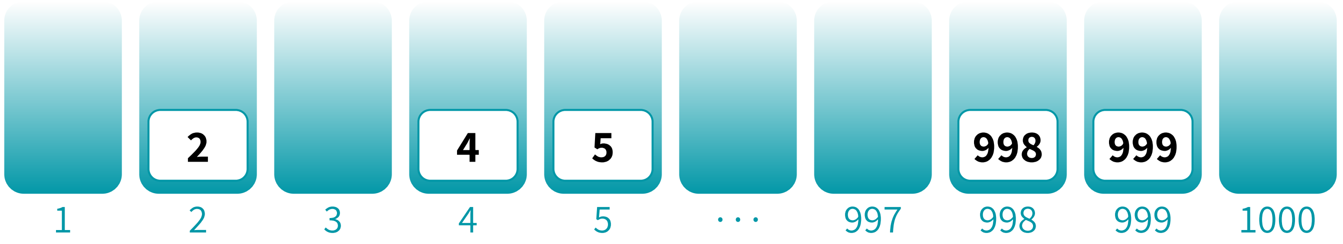
(each address/bucket stores one type of item)

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 1000: **2** **4** **5** **998** **999**

Reasonable Attempt: ***Direct Addressing!***

(each address/bucket stores one type of item)



O(1) INSERT/DELETE/SEARCH: Just index into the bucket!

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 1000: **2** **4** **5** **998** **999**

Reasonable Attempt: ***Direct Addressing!***

Not bad!

But what's the issue with this approach?

1 2 3 4 5 · · · 997 998 999 1000

O(1) INSERT/DELETE/SEARCH: Just index into the bucket!

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 10^{10}:

2	**3**	**1000**	**1002**	**10^{10}**

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 10^{10}:

2	3	1000	1002	10^{10}

Reasonable Attempt(???): ***Direct Addressing!***

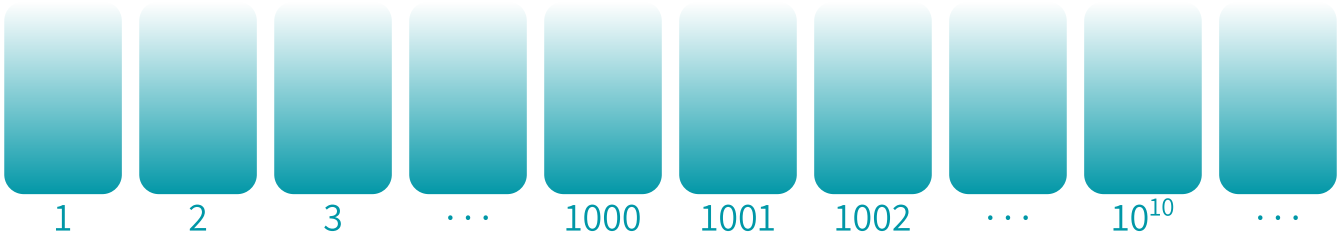
(each address/bucket stores one type of item)

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 10^{10}: **2** **3** **1000** **1002** **10^{10}**

Reasonable Attempt(???): *Direct Addressing!*

(each address/bucket stores one type of item)

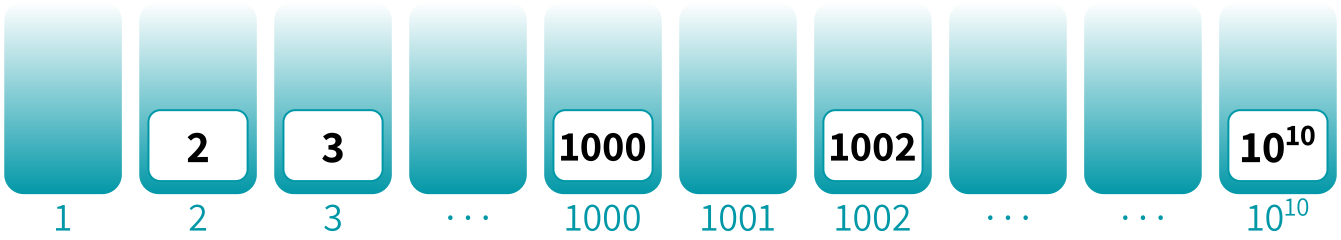


ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 10^{10}: **2** **3** **1000** **1002** **10^{10}**

Reasonable Attempt(???): *Direct Addressing!*

(each address/bucket stores one type of item)



O(1) INSERT/DELETE/SEARCH: Just index into the bucket!

ATTEMPT 1: DIRECT ADDRESSING

Suppose you're storing numbers from 1 - 10^{10}: **2** **3** **1000** **1002** **10^{10}**

Reasonable Attempt(???): ***Direct Addressing!***

(each address/bucket stores one type of item)

The space requirement is HUGE...

O(1) INSERT/DELETE/SEARCH: Just index into the bucket!

سوال؟

(ATTEMPT 2: BACK TO LINKED LISTS!)

(ATTEMPT 2: BACK TO LINKED LISTS!)

On the other extreme, we could save a lot of space by using linked lists!

HEAD → 2 → 1000 → 1002 → 3 → 10^{10}

(ATTEMPT 2: BACK TO LINKED LISTS!)

On the other extreme, we could save a lot of space by using linked lists!

HEAD → 2 → 1000 → 1002 → 3 → 10^{10}

Good news: Space is now proportional to the number of objects you deal with

Bad news: Searching for an object is now going to scale with the number of inputs you deal with… not close to our desired O(1)!

The direct-addressing approach still has merit because of it’s fast object search/access

HOW DO WE IMPROVE THIS?

We like the functionality of a direct-addressable array for constant time access (super fast INSERT/DELETE/SEARCH)

But reserving an bucket/array slot for each possible key leads to unreasonable space requirements… (kind of like CountingSort)

HOW DO WE IMPROVE THIS?

We like the functionality of a direct-addressable array for constant time access (super fast INSERT/DELETE/SEARCH)

But reserving an bucket/array slot for each possible key leads to unreasonable space requirements… (kind of like CountingSort)

We fixed CountingSort's space issues by using a kind of "binning" approach - what if we try that here?

(RadixSort put items in "bins" according to digit values)

HOW DO WE IMPROVE THIS?

We like the functionality of a direct-addressable array for constant time access (super fast INSERT/DELETE/SEARCH)

But reserving an bucket/array slot for each possible key leads to unreasonable space requirements… (kind of like CountingSort)

We fixed CountingSort's space issues by using a kind of "binning" approach - what if we try that here?

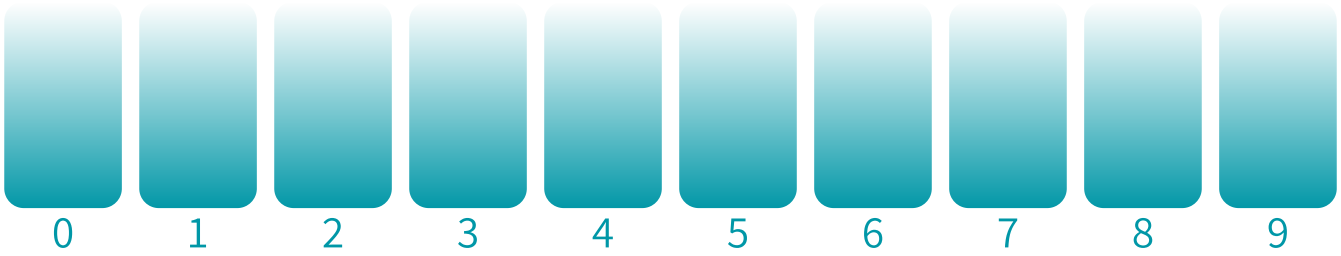
(RadixSort put items in "bins" according to digit values)

Let's try bucketing **by the least-significant digit**…

BUCKETING ATTEMPT 1:

Suppose you're storing numbers from 1 - 10^{10}: **2** **3** **1000** **1002** **10^{10}**

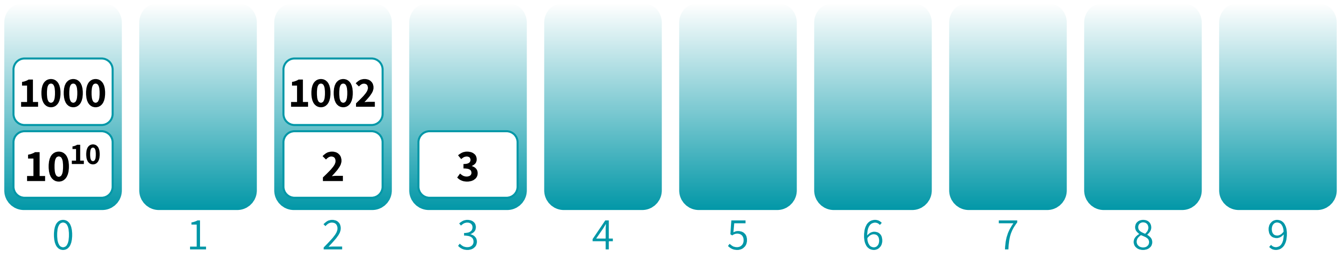
Bucket by last digit?



BUCKETING ATTEMPT 1:

Suppose you're storing numbers from 1 - 10^{10}: 2 3 1000 1002 10^{10}

Bucket by last digit?



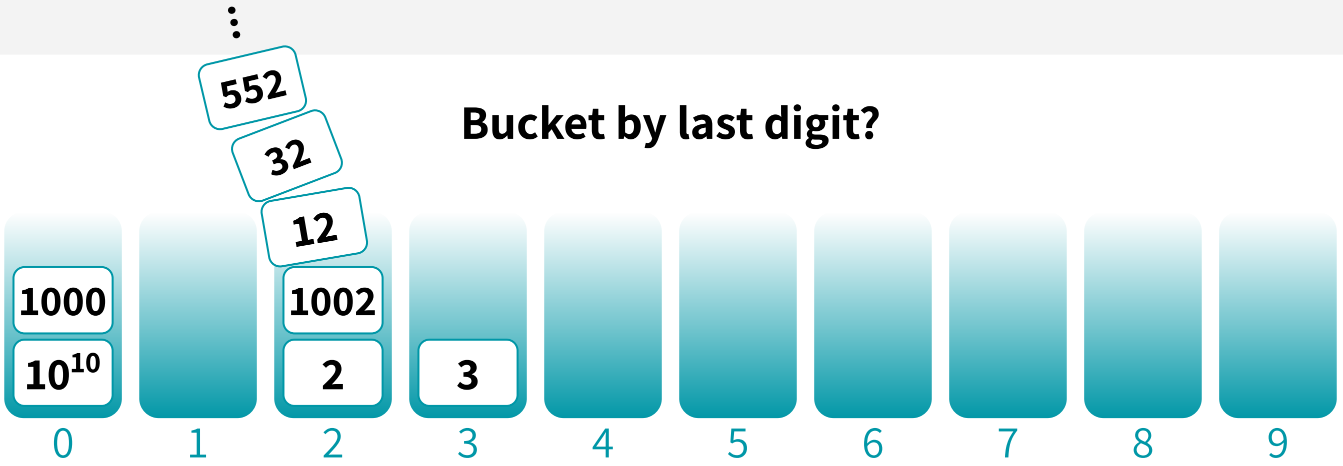
O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!

O(_) SEARCH/DELETE: Go visit bucket & search through until you find it...

BUCKETING ATTEMPT 1:

Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes…

Bucket by last digit?

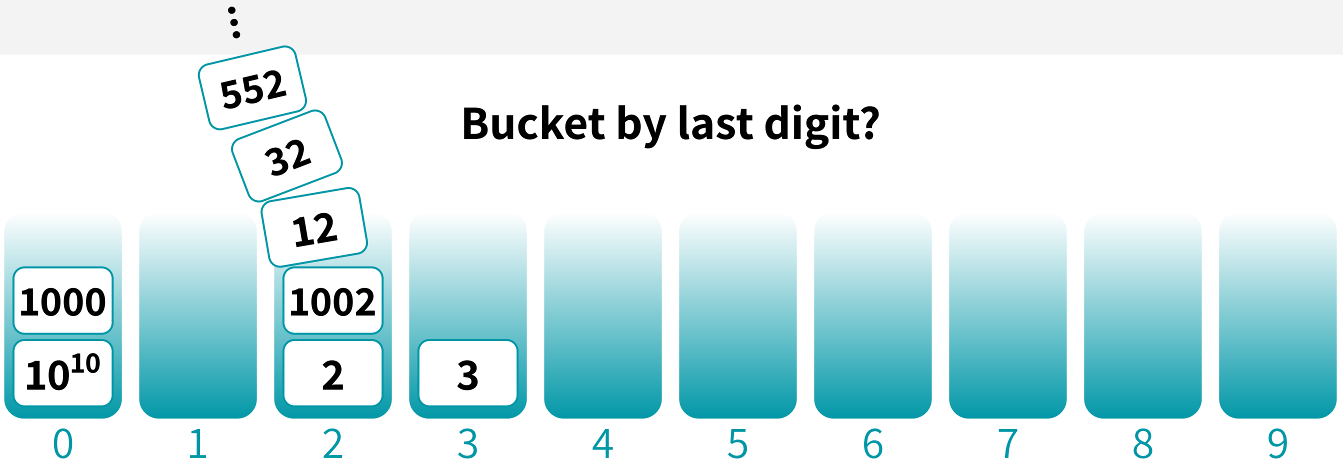


O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!

O(n) SEARCH/DELETE: Go visit bucket & search through until you find it…

BUCKETING ATTEMPT 1:

Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes...



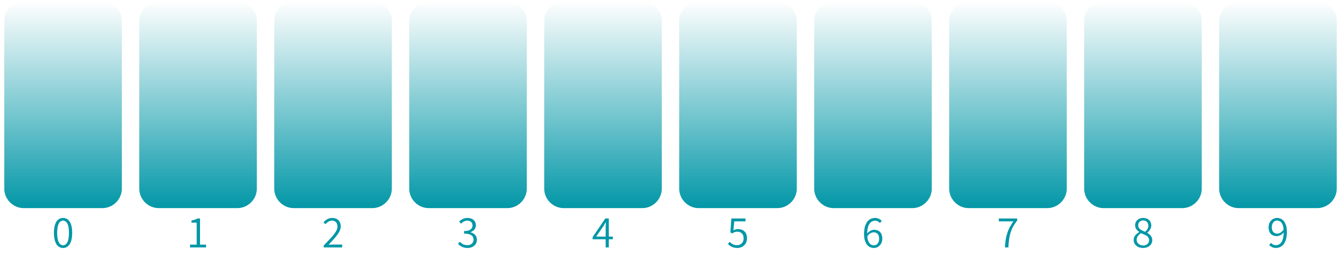
O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!
O(<u>n</u>) SEARCH/DELETE: Go visit bucket & search through until you find it...

Maybe another bucketing scheme?

BUCKETING ATTEMPT 2:

Suppose you're storing numbers from 1 - 10^{10}: 2 | 3 | 1000 | 1002 | 10^{10}

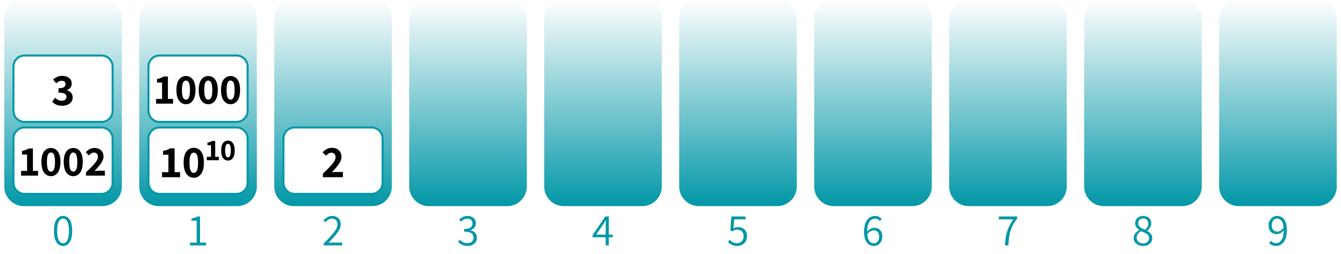
Bucket by last digit of (number * 7) mod 3



BUCKETING ATTEMPT 2:

Suppose you're storing numbers from 1 - 10^{10}: **2** **3** **1000** **1002** **10^{10}**

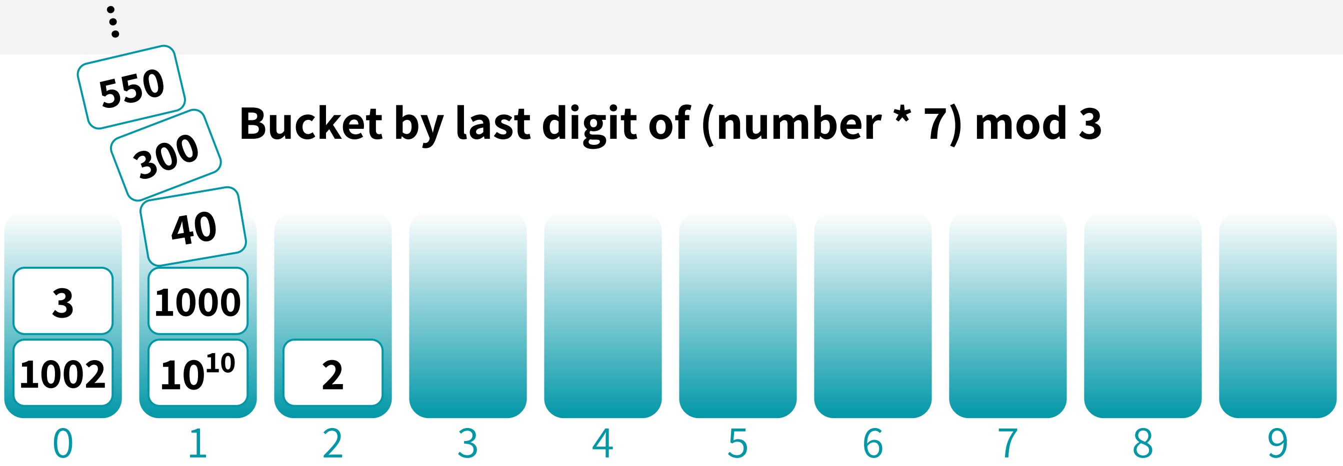
Bucket by last digit of (number * 7) mod 3



O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!
O(_) SEARCH/DELETE: Go visit bucket & search through until you find it...

BUCKETING ATTEMPT 2:

Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes...



O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!
O(<u>n</u>) SEARCH/DELETE: Go visit bucket & search through until you find it...

BUCKETING ATTEMPT 2:

Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes...

⋮

550

Bucket by last digit of (number * 7) mod 3

Seems like a bad guy could still thwart us.
There are other bucketing schemes we could use,
so to reason about them more formally,
let's talk about **HASH FUNCTIONS.**

O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!
O(n) SEARCH/DELETE: Go visit bucket & search through until you find it...

سوال؟

تابع درهم ساز

چه تابع درهم سازی خوب است؟

SOME TERMINOLOGY

There exists a universe **U** of keys, with size M.

Generally, M is *really big*. Examples:

- U = the set of all ASCII strings of length 20. M = 26^{20}
- U = the set of all IPv4 addresses. M = 2^{32}
- U = the set of all possible YouTube view stats. M = 11.83 billion

SOME TERMINOLOGY

There exists a universe **U** of keys, with size M.

Generally, M is *really big*. Examples:

- U = the set of all ASCII strings of length 20. M = 26^{20}
- U = the set of all IPv4 addresses. M = 2^{32}
- U = the set of all possible YouTube view stats. M = 12.76 billion

Our job is to store **n** keys, and we assume M >> n

Only a few (at most n) elements of U are ever going to show up. We don't know which ones will show up in advance.

SOME TERMINOLOGY

There exists a universe **U** of keys, with size M.

Generally, M is *really big*. Examples:

- U = the set of all ASCII strings of length 20. M = 26^{20}
- U = the set of all IPv4 addresses. M = 2^{32}
- U = the set of all possible YouTube view stats. M = 12.76 billion

Our job is to store **n** keys, and we assume M >> n

Only a few (at most n) elements of U are ever going to show up. We don't know which ones will show up in advance.

A hash function **h: U → {1, ..., n}**
maps elements of U to buckets 1, ..., n

SOME TERMINOLOGY

Ther   e M.

- U =
- U =
- U =

Our   >> n

Only a few (at mos   now up in advance.

NOTE:

For this lecture, I'm assuming that the # of elements I receive is the same as the # of buckets (both are **n**). This doesn't have to be the case, but we usually aim for
#buckets = O(# elements that show up)
(otherwise, we're using "too much" space)

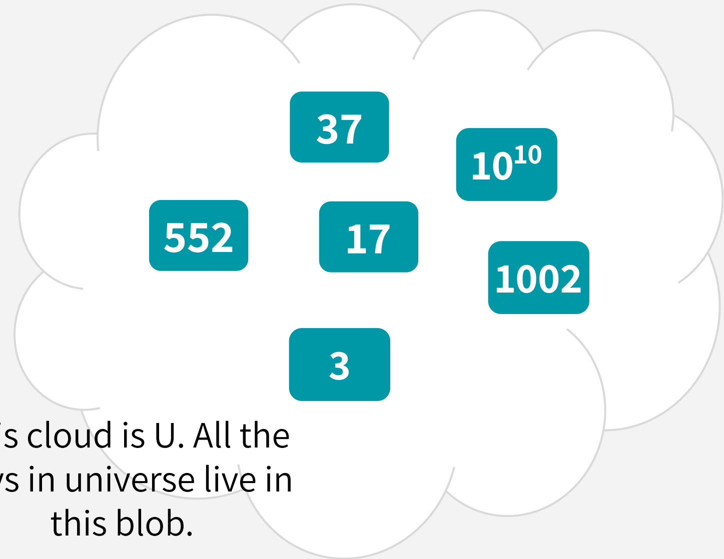
maps elements of U to buckets 1, …, n

SOME TERMINOLOGY

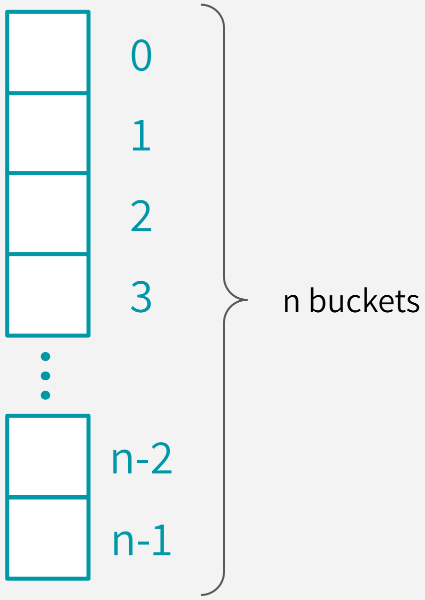
A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

SOME TERMINOLOGY

A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

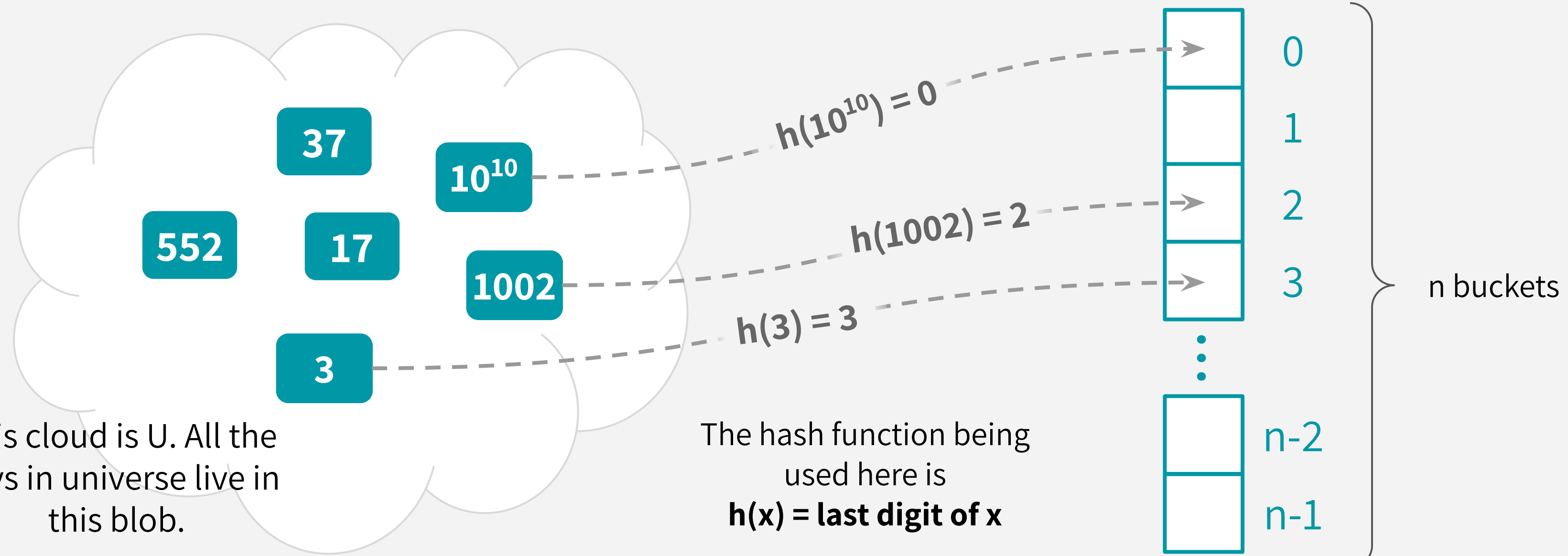


This cloud is U. All the keys in universe live in this blob.



SOME TERMINOLOGY

A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

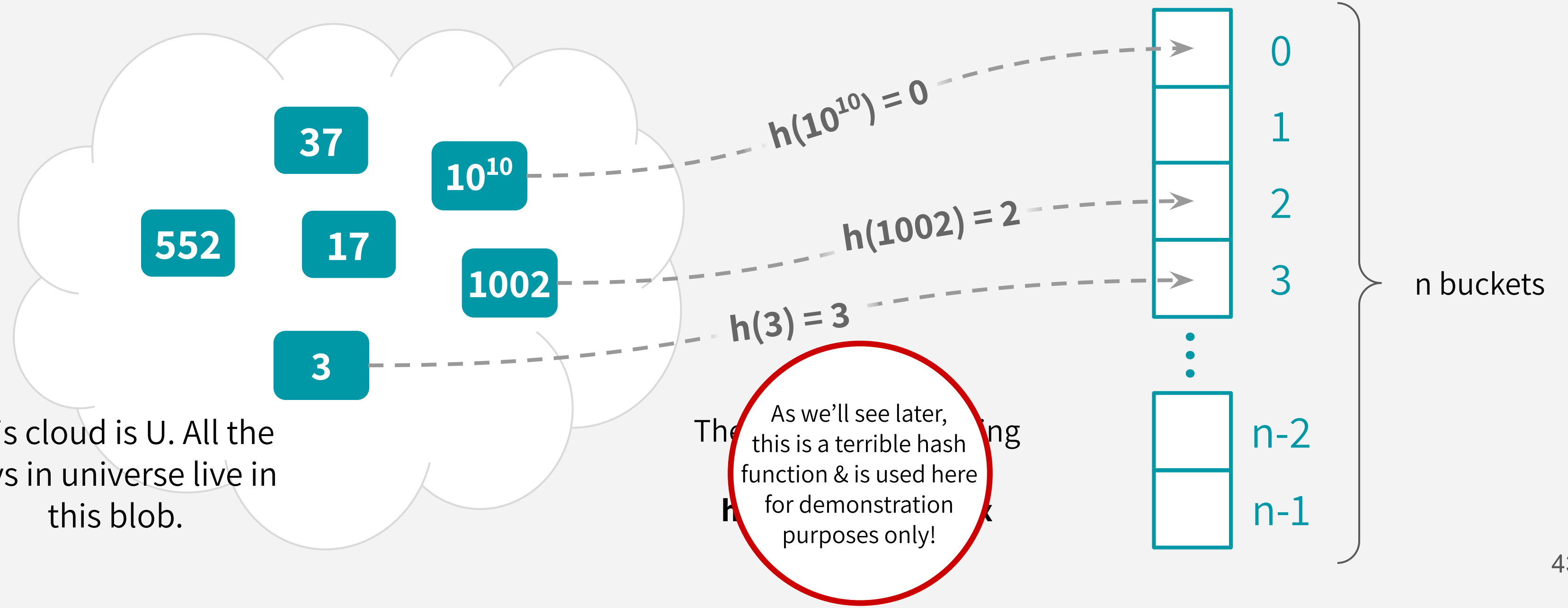


This cloud is U. All the keys in universe live in this blob.

The hash function being used here is
h(x) = last digit of x

SOME TERMINOLOGY

A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

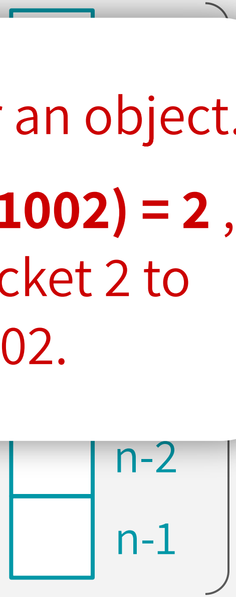


SOME TERMINOLOGY

A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

A hash function tells you where to start looking for an object.

For example, if a particular hash function **h** has **h(1002) = 2** , then we say "1002 *hashes* to 2", and we go to bucket 2 to search for 1002, or insert 1002, or delete 1002.



n buckets

This cloud is U. All the keys in universe live in this blob.

The hash function being used here is
h(x) = last digit of x

سوال؟

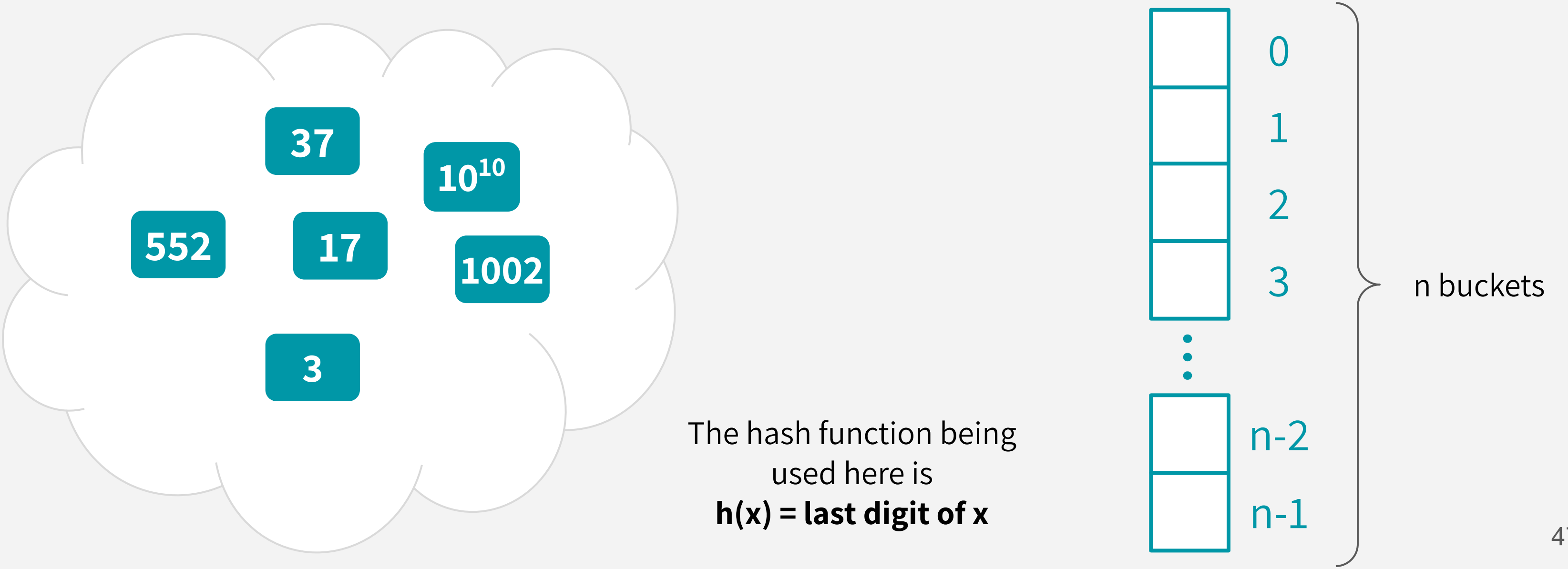
برخورد در درهم ساز!

چه مشکلی ممکن است پیش بیاید؟

COLLISIONS

Collisions (when a hash function would map 2 different keys to the same bucket) **are inevitable!**

This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.

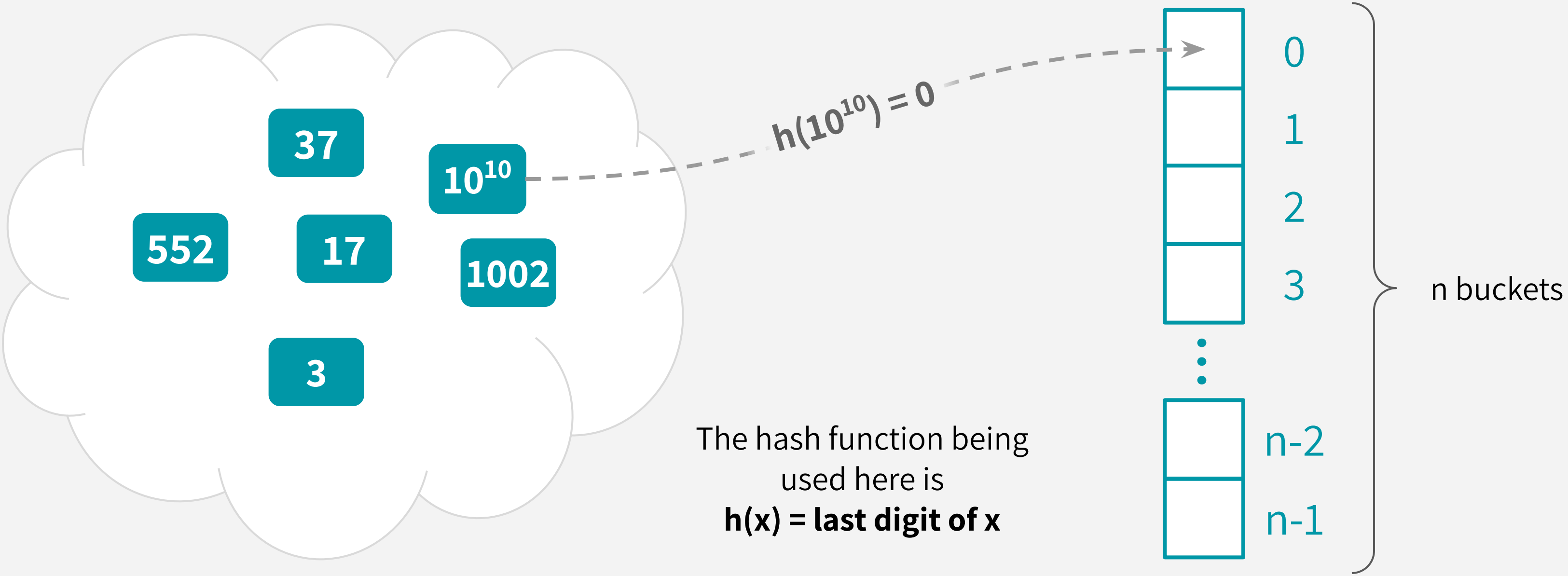


The hash function being used here is
h(x) = last digit of x

COLLISIONS

Collisions (when a hash function would map 2 different keys to the same bucket) **are inevitable!**

This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.

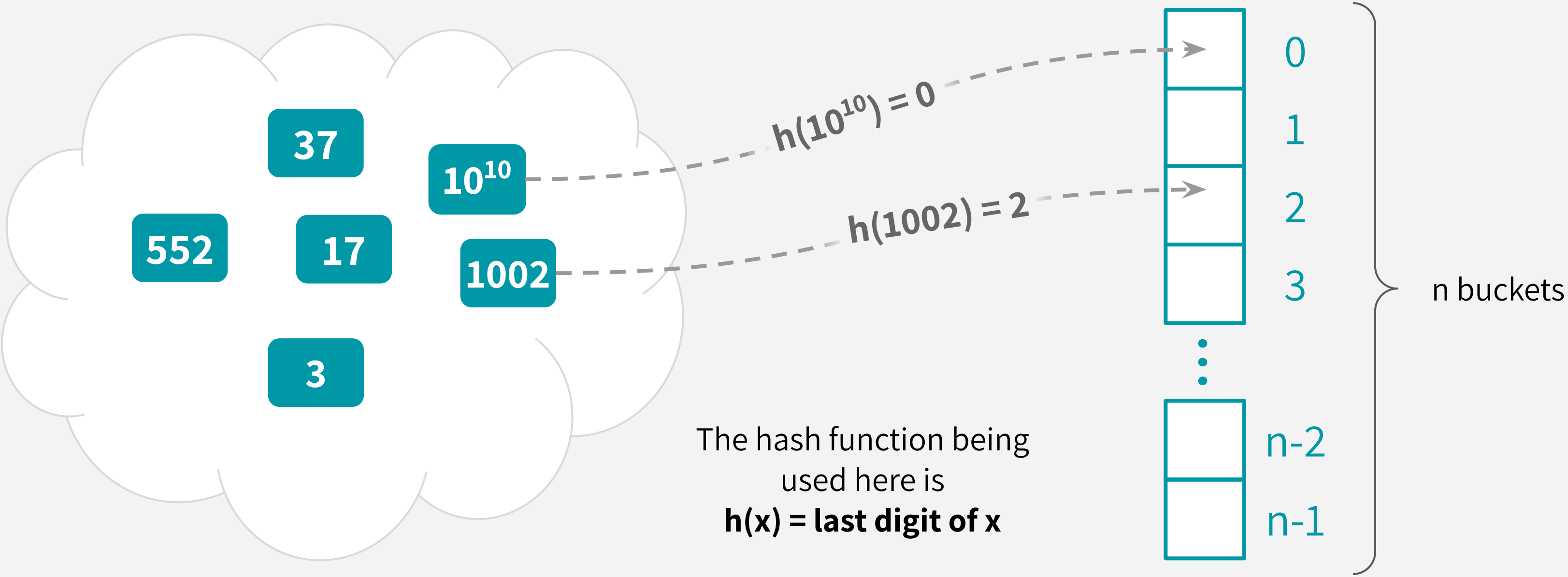


The hash function being used here is **h(x) = last digit of x**

COLLISIONS

Collisions (when a hash function would map 2 different keys to the same bucket) **are inevitable!**

This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.

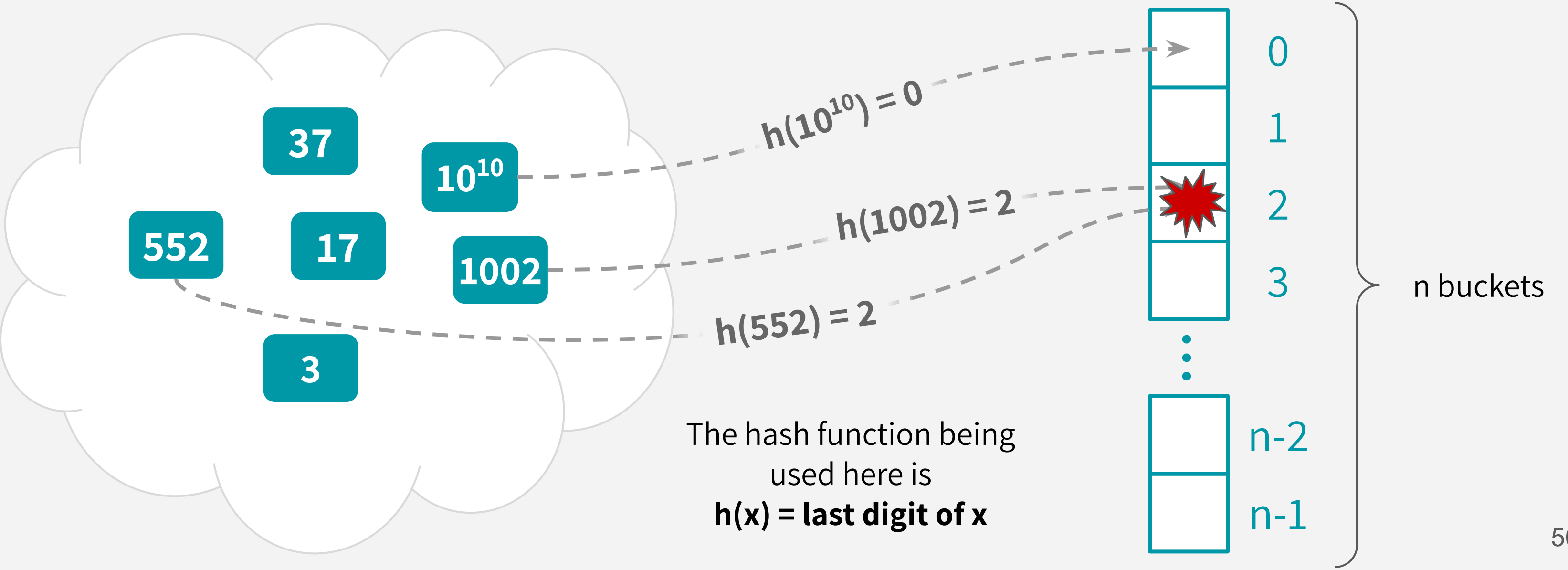


The hash function being used here is **h(x) = last digit of x**

COLLISIONS

Collisions (when a hash function would map 2 different keys to the same bucket) **are inevitable!**

This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.



The hash function being used here is **h(x) = last digit of x**

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

We're just giving a formal name to our bucketing example from earlier:
represent each bucket's contents as a *linked list*!

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

(Another method is called "Open Addressing", which we won't cover in this class)

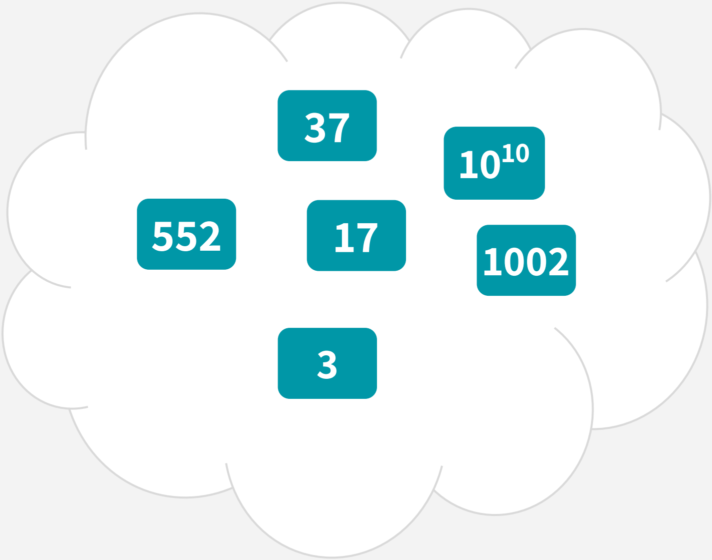
We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

(Another method is called "Open Addressing", which we won't cover in this class)

We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!



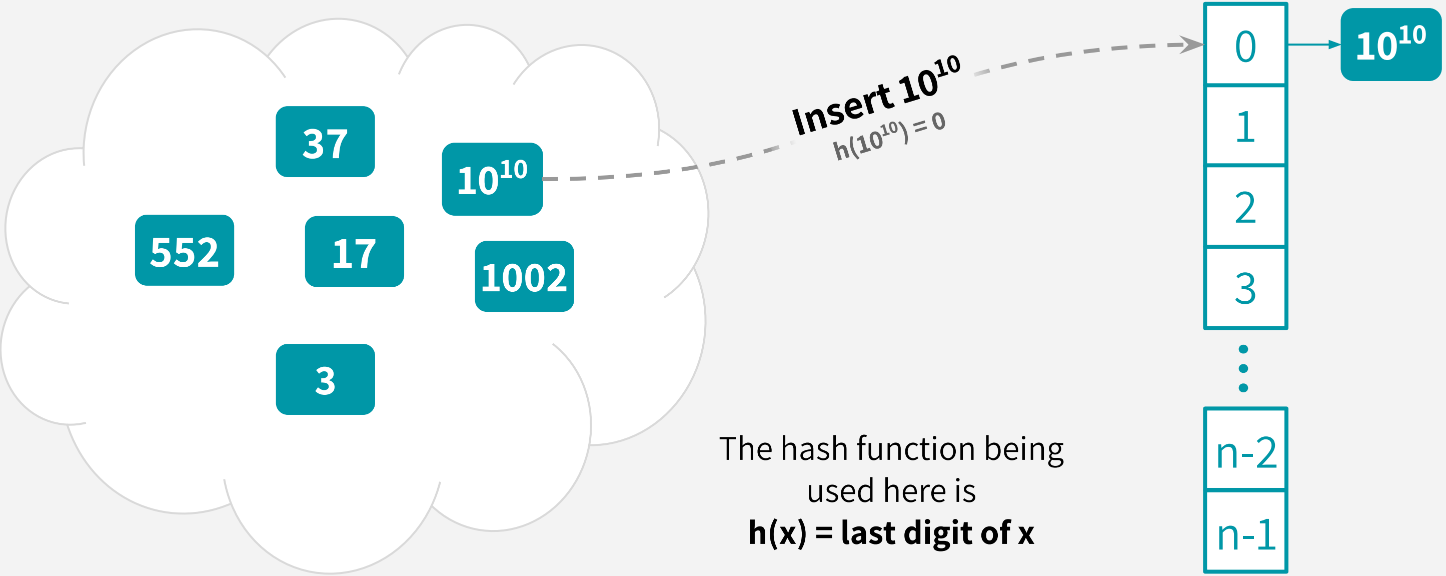
The hash function being used here is **h(x) = last digit of x**

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

(Another method is called "Open Addressing", which we won't cover in this class)

We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!



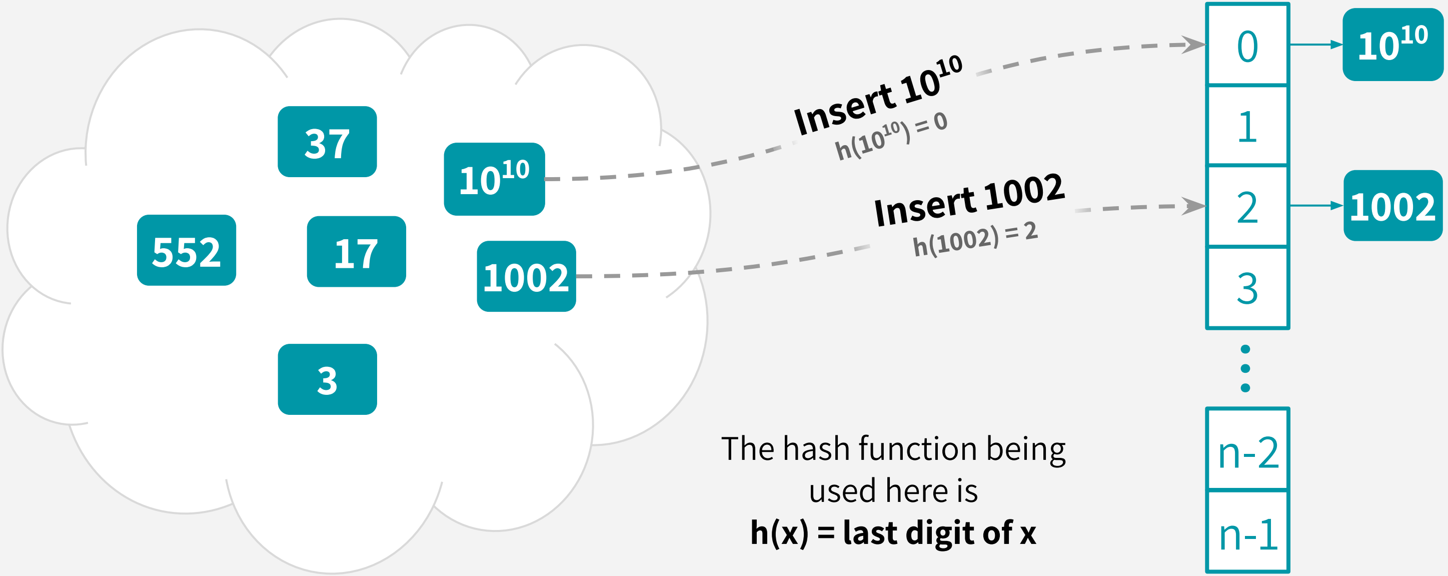
The hash function being used here is **h(x) = last digit of x**

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

(Another method is called "Open Addressing", which we won't cover in this class)

We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!



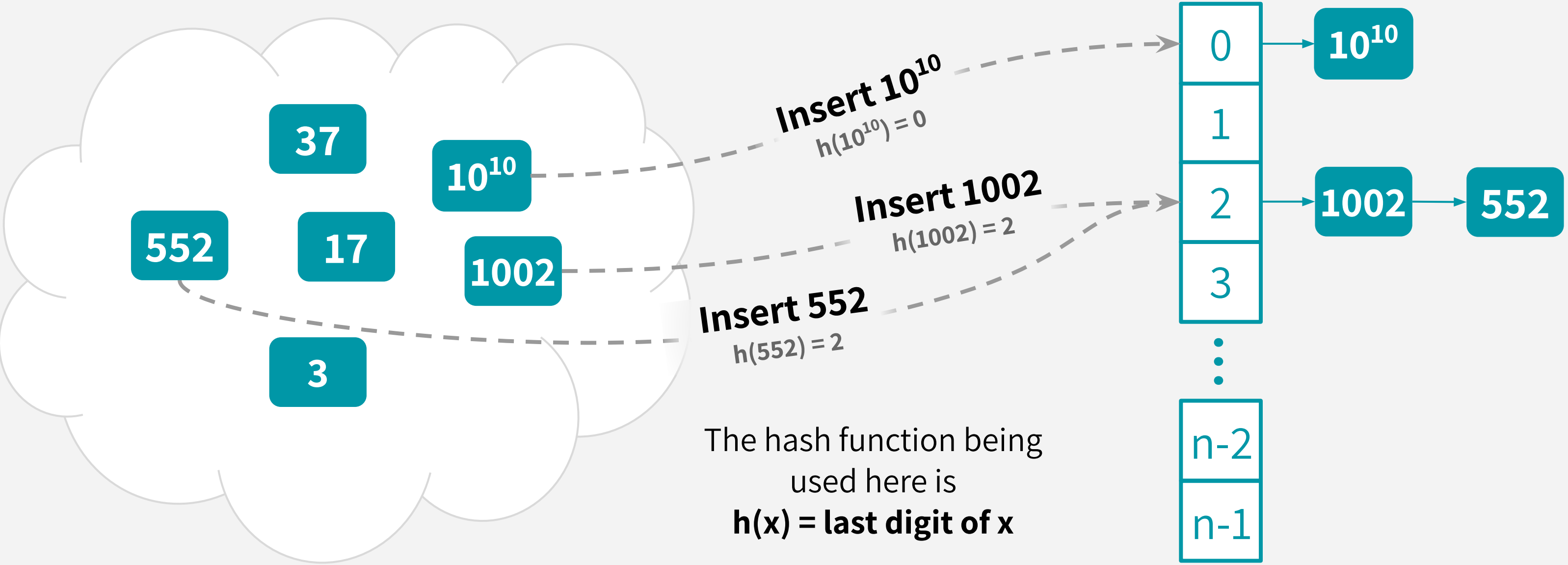
The hash function being used here is
h(x) = last digit of x

COLLISION RESOLUTION: CHAINING

To resolve collisions, one common method is to use **chaining**!

(Another method is called "Open Addressing", which we won't cover in this class)

We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!



The hash function being used here is
h(x) = last digit of x

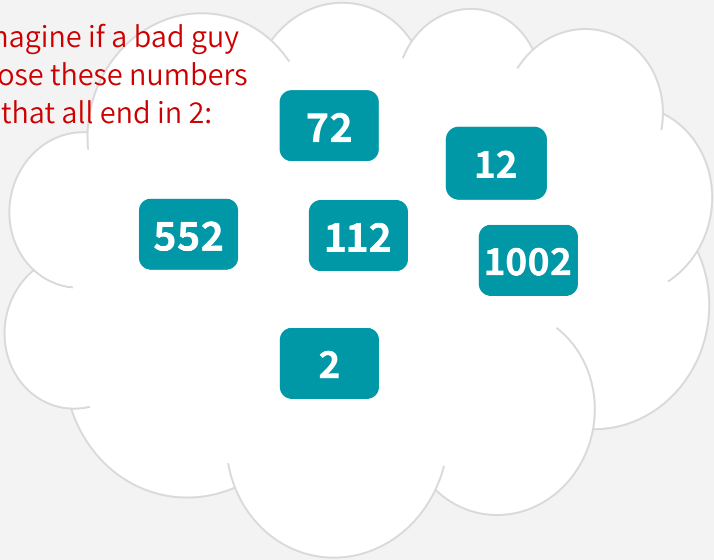
COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal…

COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket,
SEARCH/DELETE may be very slow because of the linked list traversal…

Imagine if a bad guy chose these numbers that all end in 2:

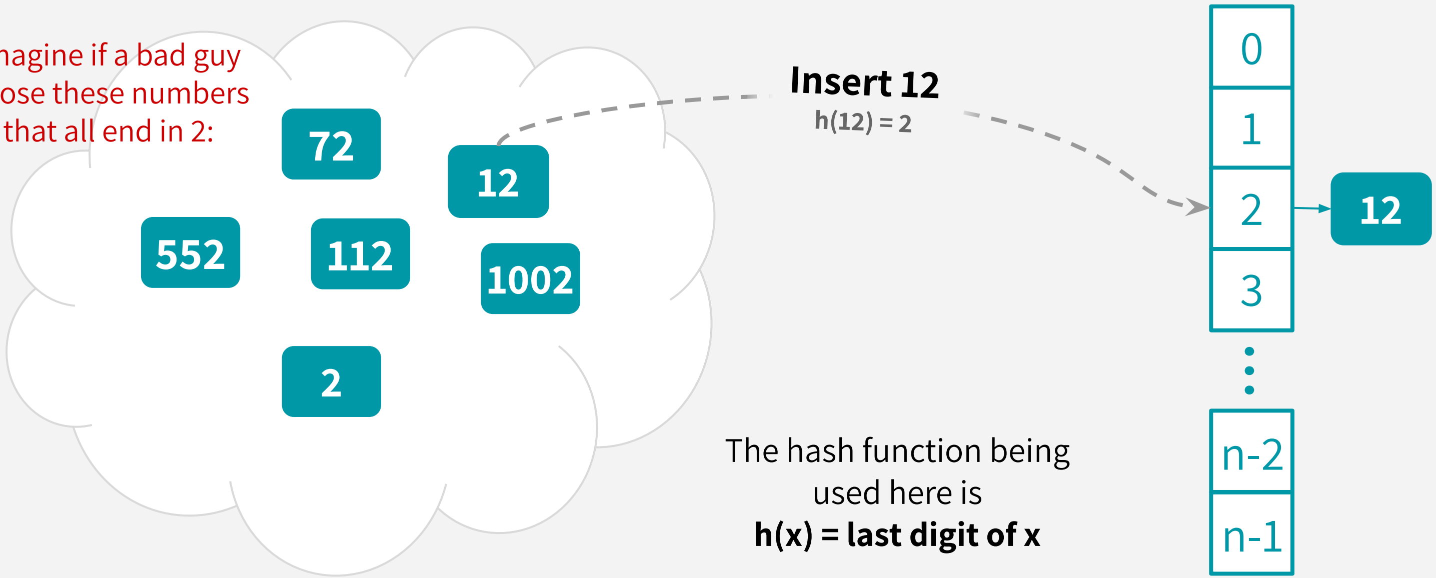


The hash function being used here is
h(x) = last digit of x

COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal…

Imagine if a bad guy chose these numbers that all end in 2:

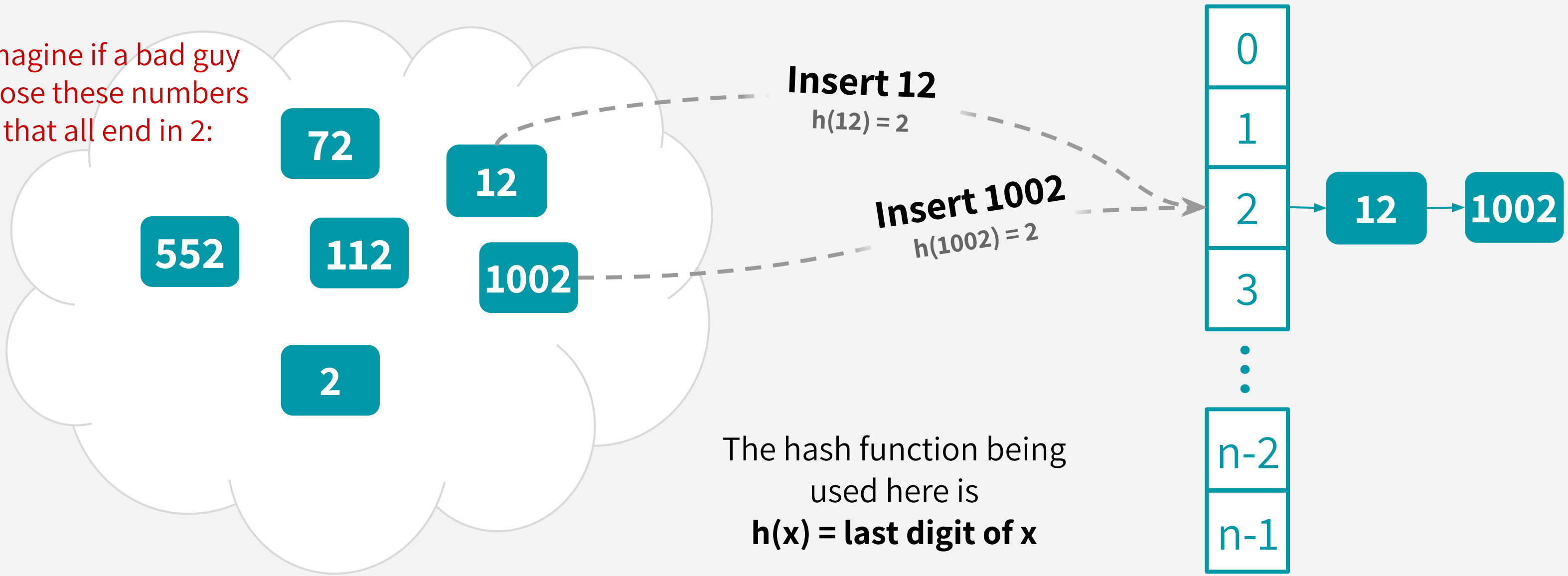


The hash function being used here is **h(x) = last digit of x**

COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal…

Imagine if a bad guy chose these numbers that all end in 2:

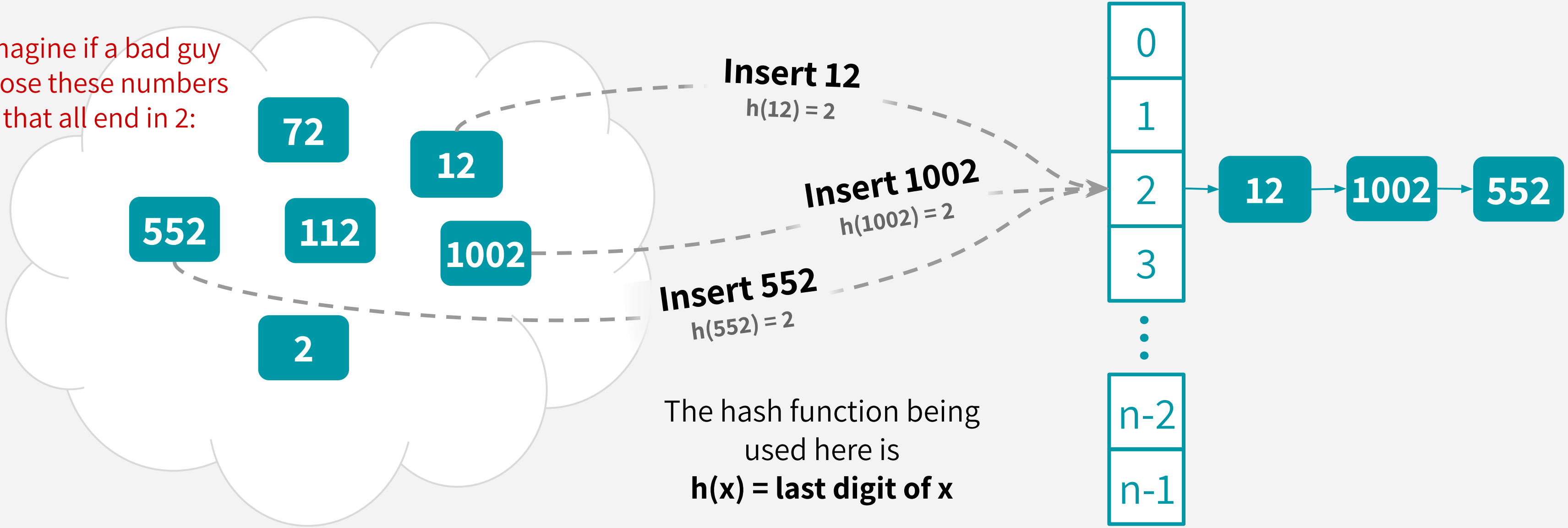


The hash function being used here is
h(x) = last digit of x

COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal...

Imagine if a bad guy chose these numbers that all end in 2:



The hash function being used here is **h(x) = last digit of x**

سوال؟

درهم سازی اعلی!

چه درهم سازی خوب خواهد بود؟

HASH TABLE GOALS

Remember worst-case analysis:

***OUR GOAL*:** Design a function **h: U** $\rightarrow$ **{1, …, n}** so that no matter what **n** items of **U** a bad guy chooses & the operations they choose to perform, the buckets will be balanced.

(Here, balanced means O(1) entries per bucket)

HASH TABLE GOALS

Remember worst-case analysis:

***OUR GOAL*:** Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses & the operations they choose to perform, the buckets will be balanced.

(Here, balanced means O(1) entries per bucket)

Then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH.

HASH TABLE GOALS

Remember worst-case analysis:

***OUR GOAL*:** Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses & the operations they choose to perform, the buckets will be balanced.

(Here, balanced means O(1) entries per bucket)

Then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH.

Can you come up with such a function?

HASH TABLE GOALS

BAD NEWS!

OUR GOAL: Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

No *deterministic* hash function can defeat worst-case input!

HASH TABLE GOALS

BAD NEWS!

OUR GOAL: Design a function $\mathbf{h: U \rightarrow \{1, \ldots, n\}}$ so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

No *deterministic* hash function can defeat worst-case input!

- The universe U has **M** items
- They get hashed into **n** buckets
- At least 1 bucket has at least **M/n** items hashed to it (Pigeonhole)
- **M** is wayyyy bigger than **n**, so **M/n** is bigger than **n**

HASH TABLE GOALS

BAD NEWS!

OUR GOAL: Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

No *deterministic* hash function can defeat worst-case input!

- The universe U has **M** items
- They get hashed into **n** buckets
- At least 1 bucket has at least **M/n** items hashed to it (Pigeonhole)
- **M** is wayyyy bigger than **n**, so **M/n** is bigger than **n**

The n items the bad guy chooses are items that all land in this very full bucket. That bucket has size Ω(n).

HASH TABLE GOALS

BAD NEWS!

OUR GOAL: Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

No *deterministic* hash function can defeat worst-case input!

- The universe U has **M** items
- They get hashed into **n** buckets
- At least 1 bucket has at least **M/n** items hashed to it (Pigeonhole)
- **M** is wayyyy bigger than **n**, so **M/n** is bigger than **n**

The n items the bad guy chooses are items that all land in this very full bucket. That bucket has size Ω(n).

The problem is that the bad guy knows our hash function beforehand.

HASH TABLE GOALS

OUR GOAL: Design a function **h: U → {1, …, n}** so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

- The
- The
- At l
- **M** is

Th … **nd in this very full bucket. That bucket has size Ω(n).**

Maybe there's a way to weaken the adversary...

Any suggestion?

The problem is that the bad guy knows our hash function beforehand.

HASH TABLE GOALS

OUR GOAL: Design a function **h: U** → **{1, …, n}** so that no matter what **n** items of **U** a bad guy chooses, the buckets will be balanced (have O(1) size).

Can you come up with such a function? **No.**

- The
- The
- At l
- **M** is

Th ... **nd in this very full bucket. That bucket has size Ω(n).**

Maybe there's a way to weaken the adversary…

LET'S BRING IN SOME

RANDOMNESS!

The problem is that the bad guy knows our hash function beforehand.

سوال؟

تصادفی کردن تابع درهم ساز

چگونه میتوان بدخواهان را تضعیف کرد؟

INTUITION

Intuitively, the adversary can’t foil a hash function that they don’t yet know.

INTUITION

Intuitively, the adversary can't foil a hash function that they don't yet know.

So, our strategy is to define a set of hash functions, and then we randomly choose a hash function **h** from this set to use!

INTUITION

Intuitively, the adversary can't foil a hash function that they don't yet know.

So, our strategy is to define a set of hash functions, and then we randomly choose a hash function **h** from this set to use!

You can think of it like a game:

1. You announce your set of hash functions, **H**.
2. The adversary chooses **n** items for your hash function to hash.
3. You then randomly pick a hash function **h** from **H** to hash the **n** items.

INTUITION

Intuitively, the adversary can't foil a hash function that they don't yet know.

So, our strategy is to define a set of hash functions, and then we randomly choose a hash function **h** from this set to use!

You can think of it like a game:

1. You announce your set of hash functions, **H**.
2. The adversary chooses **n** items for your hash function to hash.
3. You then randomly pick a hash function **h** from **H** to hash the **n** items.

What would make a "good" set of hash functions H?

تابع درهم ساز خوب

معنی خوب بودن چیست؟

WHAT DOES "GOOD" MEAN?

Consider these two goals:

Which goal better represents what we want?

WHAT DOES "GOOD" MEAN?

Consider these two goals:

Design a set **H** = $\{h_1, h_2, h_{3,} \ldots, h_k\}$ where $h_i : U \to \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Which goal better represents what we want?

WHAT DOES "GOOD" MEAN?

Consider these two goals:

Design a set **H** = $\{h_1, h_2, h_3, \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Design a set **H** = $\{h_1, h_2, h_3, \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_n}\}$ to hash,

for any item $\mathbf{u_i}$,
the **expected** # of items in $\mathbf{u_i}$'s bucket is **O(1)**

Which goal better represents what we want?

WHAT DOES "GOOD" MEAN?

Co… …goals:

Design a set **H** = {h_1, … where h_i : U → {1… that if we chose a… and after an adv… items t…

for any b… its **expected** s…

…set **H** = {h_1, h_2, h_3, …, h_k} …U → {1, …, n} such …ose a random **h** in **H** …dversary chooses **n** …$_2$, …, u_n} to hash,

…ny item u_i, …cted # of items in …bucket is **O(1)**

SUPER IMPORTANT:

The randomness is over the choice of hash function **h** from a set of hash functions **H**.

You should *not* think of it as if you've chosen a fixed hash function and are thinking about randomness over possible items the adversary could choose, or randomness over the n possible buckets in your table, or randomness over the M possible items, or anything like that.

Which goal… …hat we want?

WHAT DOES "GOOD" MEAN?

Design a set **H** = $\{h_1, h_2, h_{3,} \ldots, h_k\}$
where $h_i : U \rightarrow \{1, \ldots, n\}$ such
that if we chose a random **h** in **H**
and after an adversary chooses **n**
items to hash,

for any bucket,
its **expected** size is **O(1)**

Not what we want!

WHAT DOES "GOOD" MEAN?

Design a set **H** = {h_1, h_2, $h_{3,}$ …, h_k} where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Not what we want!

Why is this goal not a good one?

Well, this *bad* set of hash functions (which always results in chains of length **n** in a single bucket) would meet this goal:

WHAT DOES "GOOD" MEAN?

Design a set **H** = $\{h_1, h_2, h_{3,} \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Not what we want!

Why is this goal not a good one?
Well, this *bad* set of hash functions (which always results in chains of length **n** in a single bucket) would meet this goal:

$\mathbf{H = \{h_1, h_2, \ldots, h_n\}}$ where
h_i maps all elements to bucket i.

WHAT DOES "GOOD" MEAN?

Design a set **H** = $\{h_1, h_2, h_3, \ldots, h_k\}$ where $h_i : U \to \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Not what we want!

Why is this goal not a good one?

Well, this *bad* set of hash functions (which always results in chains of length **n** in a single bucket) would meet this goal:

$$\mathbf{H = \{h_1, h_2, \ldots, h_n\}} \text{ where}$$
$$h_i \text{ maps all elements to bucket i.}$$

- With prob. 1/n, all **n** elements land in bucket 1
- With prob. 1/n, all **n** elements land in bucket 2
- With prob. 1/n, all **n** elements land in bucket 3

…

- With prob. 1/n, all **n** elements land in bucket n

WHAT DOES "GOOD" MEAN?

Design a set $\mathbf{H} = \{h_1, h_2, h_{3,} \ldots, h_k\}$ where $h_i : U \to \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Not what we want!

Why is this goal not a good one?

Well, this *bad* set of hash functions (which always results in chains of length **n** in a single bucket) would meet this goal:

$$\mathbf{H = \{h_1, h_2, \ldots, h_n\}} \text{ where}$$
$$h_i \text{ maps all elements to bucket i.}$$

- With prob. 1/n, all **n** elements land in bucket 1
- With prob. 1/n, all **n** elements land in bucket 2
- With prob. 1/n, all **n** elements land in bucket 3

…

- With prob. 1/n, all **n** elements land in bucket n

Then, **E[# items in bucket i] = 1 = O(1)** for all i…

Bucket i has **n** elements with prob. 1/n, and 0 elements with prob. (n-1)/n

WHAT DOES "GOOD" MEAN?

Consider these two goals:

Design a set **H** = $\{h_1, h_2, h_3, \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items to hash,

for any bucket,
its **expected** size is **O(1)**

Design a set **H** = $\{h_1, h_2, h_3, \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$ such that if we chose a random **h** in **H** and after an adversary chooses **n** items $\mathbf{\{u_1, u_2, \ldots, u_n\}}$ to hash,

for any item $\mathbf{u_i}$,
the **expected** # of items in $\mathbf{u_i}$'s bucket is **O(1)**

We want the one on the right! It tries to control the expected number of collisions (which is what contributes to the linked-list traversal runtime)

WHAT DOES "GOOD" MEAN?

An analogy to explain the difference between the two:

Suppose a university offers 10 classes. 9 classes have only 1 student in them, and 1 class has 491 students.
Using the reasoning on the left, the university might say "Average class size is 50!" but in reality, it should instead report class sizes experienced by the average student (~482).

Desig… whe… that i… and a…

for any bucket,
its **expected** size is **O(1)**

…$_3$, …, h_k}
…} such
…m **h** in **H**
…hooses **n**
… to hash,

for any item $\mathbf{u_i}$,
the **expected** # of items in $\mathbf{u_i}$'s bucket is **O(1)**

We want the one on the right! It tries to control the expected number of collisions (which is what contributes to the linked-list traversal runtime)

WHAT WE WANT

Design a set $\mathbf{H} = \{h_1, h_2, h_{3,} \ldots, h_k\}$ where $h_i : U \rightarrow \{1, \ldots, n\}$, such that if we chose a random **h** in **H** and after an adversary chooses **n** items $\{\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_n}\}$ to hash,

for any item $\mathbf{u_i}$,
the **expected** # of items in $\mathbf{u_i}$'s bucket is **O(1)**

خانواده درهم سازی سراسری

مجموعه ای خوب از توابع درهم سازی که خیلی هم بزرگ نیست!

UNIVERSAL HASH FAMILY

A **hash family** is a fancy name for a set of hash functions.

UNIVERSAL HASH FAMILY

A **hash family** is a fancy name for a set of hash functions.

A hash family **H** is a **universal hash family** if,
when **h** is chosen uniformly at random from **H**,

$$\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j,$$

$$P_{h \in H}\left[h(u_i) = h(u_j)\right] \leq \frac{1}{n}$$

UNIVERSAL HASH FAMILY

A **hash family** is a fancy name for a set of hash functions.

A hash family **H** is a **universal hash family** if,
when **h** is chosen uniformly at random from **H**,

$$\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j,$$

$$P_{h \in H}\left[h(u_i) = h(u_j)\right] \leq \frac{1}{n}$$

Then if we randomly choose **h** from a universal hash family **H**, we'll be guaranteed that:

E[# of items in u_i's bucket] ≤ 2 = O(1)

(OVERVIEW OF THE MATH)

A hash family **H** is a **universal hash family** if,
when **h** is chosen uniformly at random from **H**,

for all $u_i, u_j \in U$ with $u_i \neq u_j$,
$$P_{h \in H}\left[h(u_i) = h(u_j)\right] \leq \frac{1}{n}$$

$$\begin{aligned}
\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] &= \sum_{j=1}^{n} P[h(u_i) = h(u_j)] \\
&= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)] \\
&= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)] \\
&\leq 1 + \sum_{j \neq i} \frac{1}{n} \\
&= 1 + \frac{n-1}{n} \leq 2
\end{aligned}$$

This inequality is now what a universal hash family guarantees!

O(1)

This is what we wanted!

AN EXAMPLE

Here is one of the more well-studied universal hash families:

Pick a prime $\mathbf{p \geq M}$

Define $\mathbf{h_{a,b}(x) = ((ax + b) \bmod p) \bmod n}$

$$\mathbf{H = \{\, h_{a,b} : \ a \in \{1, \ldots, p - 1\},\ b \in \{0, \ldots, p - 1\} \,\}}$$

AN EXAMPLE

Here is one of the more well-studied universal hash families:

Pick a prime $\mathbf{p \geq M}$

Define $\mathbf{h_{a,b}(x) = ((ax + b) \bmod p) \bmod n}$

$$\mathbf{H = \{ h_{a,b} : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \}}$$

Example: Suppose n = 3, and p = 5. Here's $\mathbf{h_{2,4}}$:

$\mathbf{h_{2,4}(1) =}$ ((2*1 + 4) mod 5) mod 3 = (6 mod 5) mod 3 = 1 mod 3 = **1**

$\mathbf{h_{2,4}(4) =}$ ((2*4 + 4) mod 5) mod 3 = (12 mod 5) mod 3 = 2 mod 3 = **2**

$\mathbf{h_{2,4}(3) =}$ ((2*3 + 4) mod 5) mod 3 = (6 mod 5) mod 3 = 1 mod 3 = **1**

AN EXAMPLE

Here is one of the more well-studied universal hash families:

Pick a prime $\mathbf{p \geq M}$

Define $\mathbf{h_{a,b}(x) = ((ax + b) \bmod p) \bmod n}$

$$\mathbf{H = \{ h_{a,b} : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \}}$$

To draw a hash function **h** from **H**:

Pick a random **a** in {1, …, p - 1}.

&

Pick a random **b** in {0, …, p - 1}.

AN EXAMPLE

To store your $\mathbf{h_{a,b}}$, you just need to store two numbers: **a** and **b**!

Since **a** and **b** are at most p-1, we need **~2·log(p) bits**.

p is a prime that's close-ish to M, so this means the space needed =

O(log M)

Pick a random **a** in {1, …, p - 1}. & Pick a random **b** in {0, …, p - 1}.

AN EXAMPLE

Claim: This **H** is a universal hash family!

The proof is a bit complicated, and relies on number theory. See CLRS (Theorem 11.5) for details if you're curious, but **YOU ARE NOT RESPONSIBLE** for the proof in this class.

What you should know:

There exists a small universal hash family! A hash function from this universal hash family is quick to compute, lightweight to store, and relies on number theory to achieve our expected O(1) operation costs!

سوال؟

جدول درهم سازی

جمع بندی مطالب درهم سازی و استفاده عملی از آن!

THE WHOLE SCHEME

You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.

H

THE WHOLE SCHEME

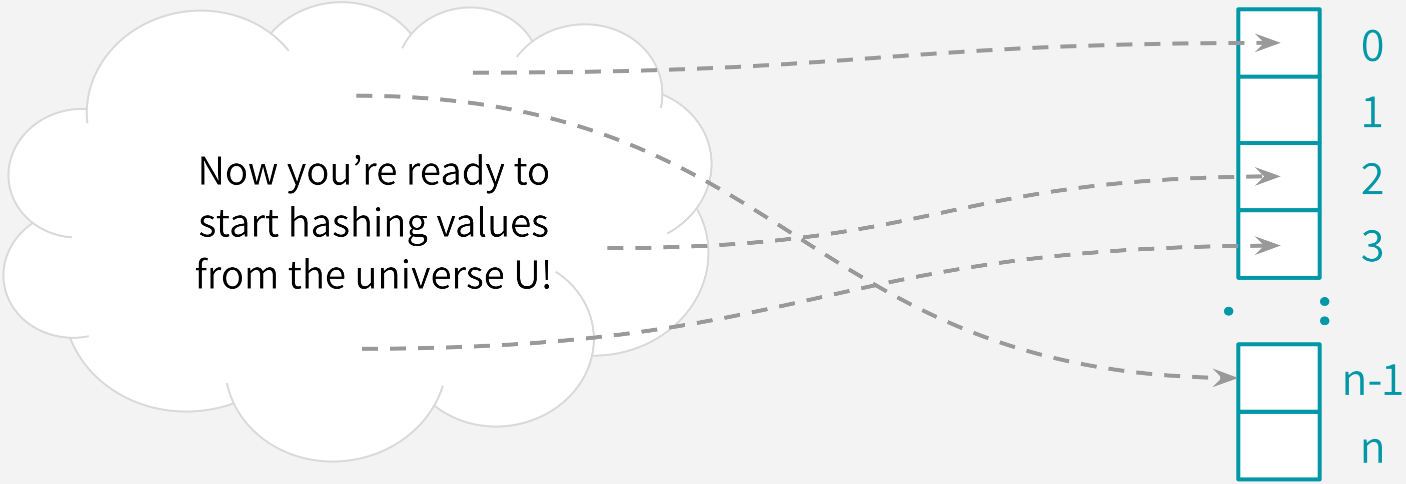
You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.

When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.

THE WHOLE SCHEME

You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.

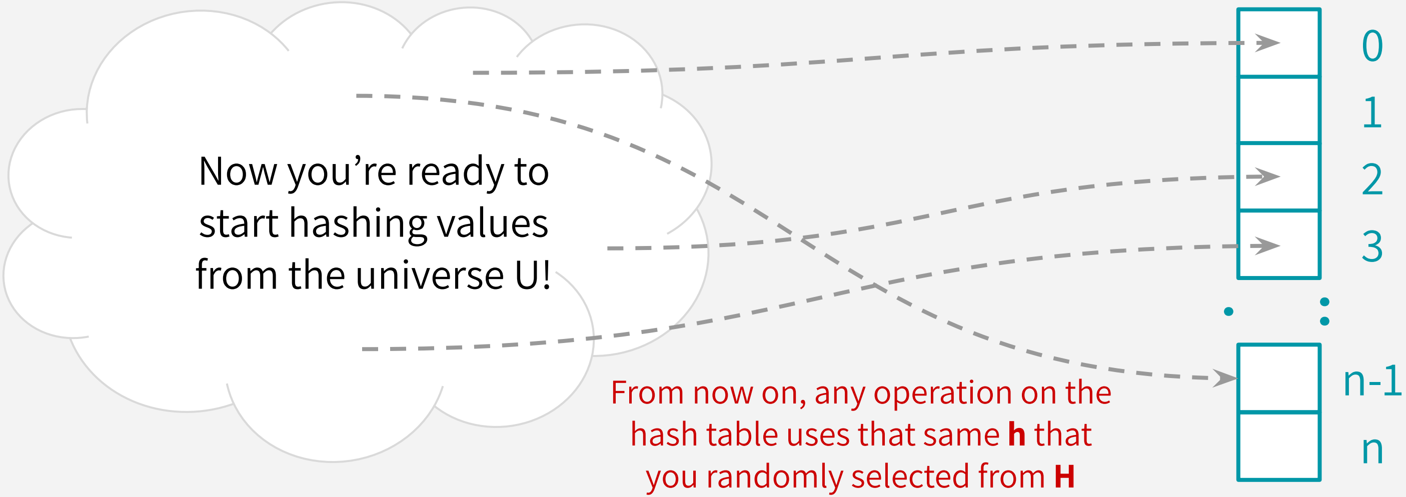
When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.



THE WHOLE SCHEME

You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.

When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.



We can now expect that these buckets will be pretty balanced

From now on, any operation on the hash table uses that same **h** that you randomly selected from **H**

HASH TABLE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	HASH TABLES (WORST-CASE)	HASH TABLES (EXPECTED)*
SEARCH	O(log(n))	O(n)	O(n)	O(1)
DELETE	O(n)	O(n)	O(n)	O(1)
INSERT	O(n)	O(1)	O(1)	O(1)

*** Assuming we implement it cleverly with a “good” hash function**

سوال؟