ساختمان داده و الگوریتم ها (CE203)

جلسه هجدهم:
درهم سازی

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 13 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 11

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

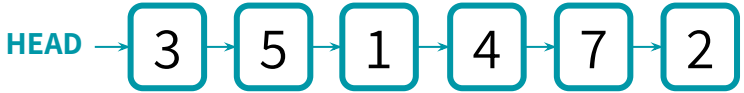
Again, we want to keep track of objects that have keys $\boxed{5}$ (aka, **nodes** with **keys**)

Sorted Arrays

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



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)

HASH TABLE MOTIVATION

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

HASH TABLE MOTIVATION

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

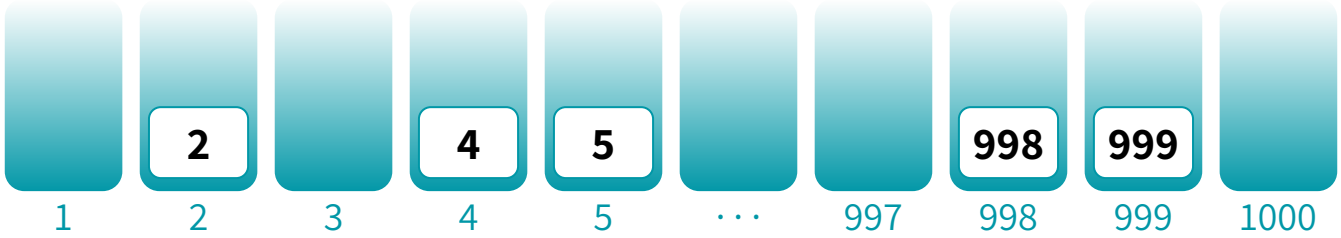
(each address/bucket stores one type of item)

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!*

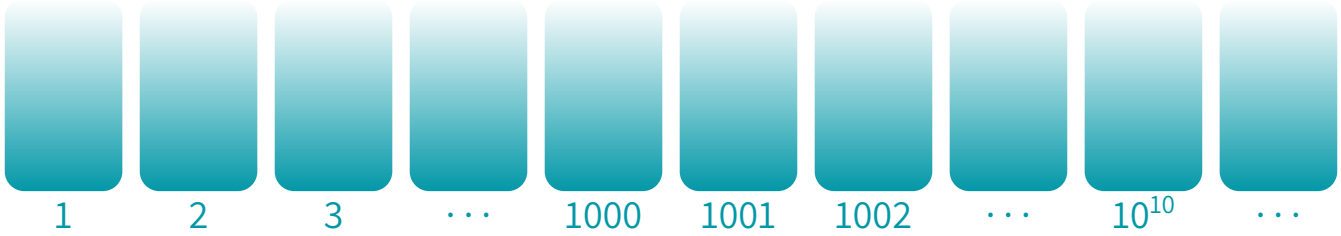
(each address/bucket stores one type of item)

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)

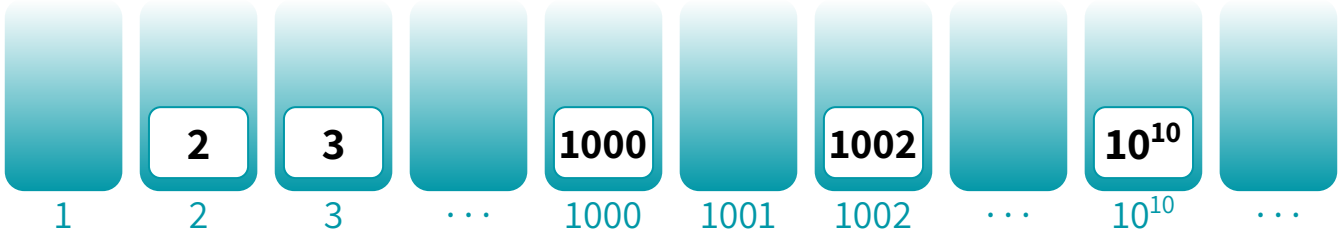


ATTEMPT 1: DIRECT ADDRESSING

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)

But the space requirement is HUGE...

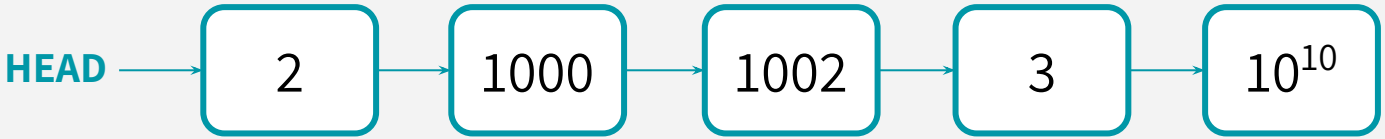
O(1) INSERT/DELETE/SEARCH: Just index into the bucket!

سوال؟

(ATTEMPT 2: BACK TO LINKED LISTS!)

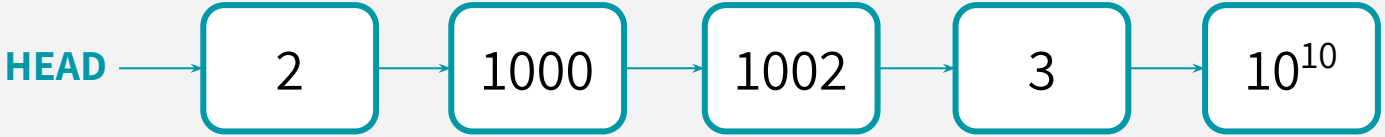
(ATTEMPT 2: BACK TO LINKED LISTS!)

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



(ATTEMPT 2: BACK TO LINKED LISTS!)

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



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?

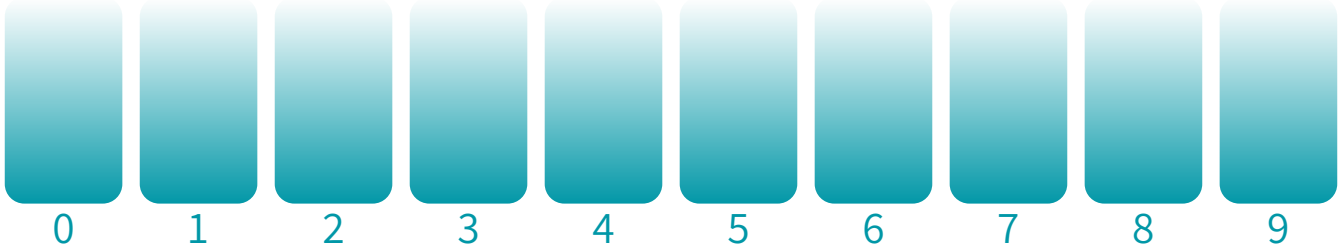
(RadixSort put items in "bins" according to digit values)

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}**

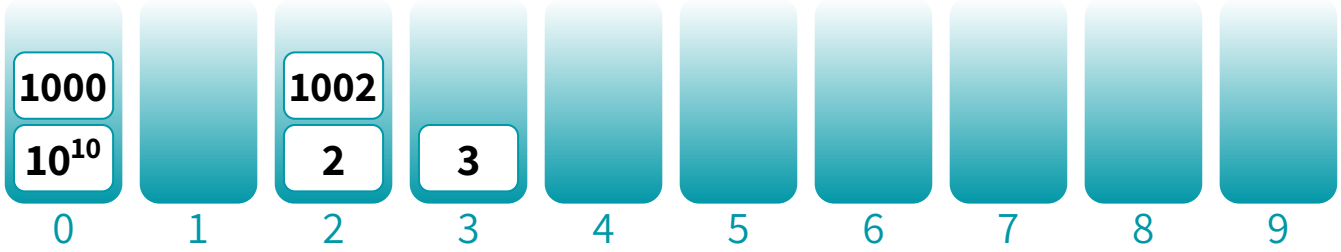
Bucket by last digit?



BUCKETING ATTEMPT 1:

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

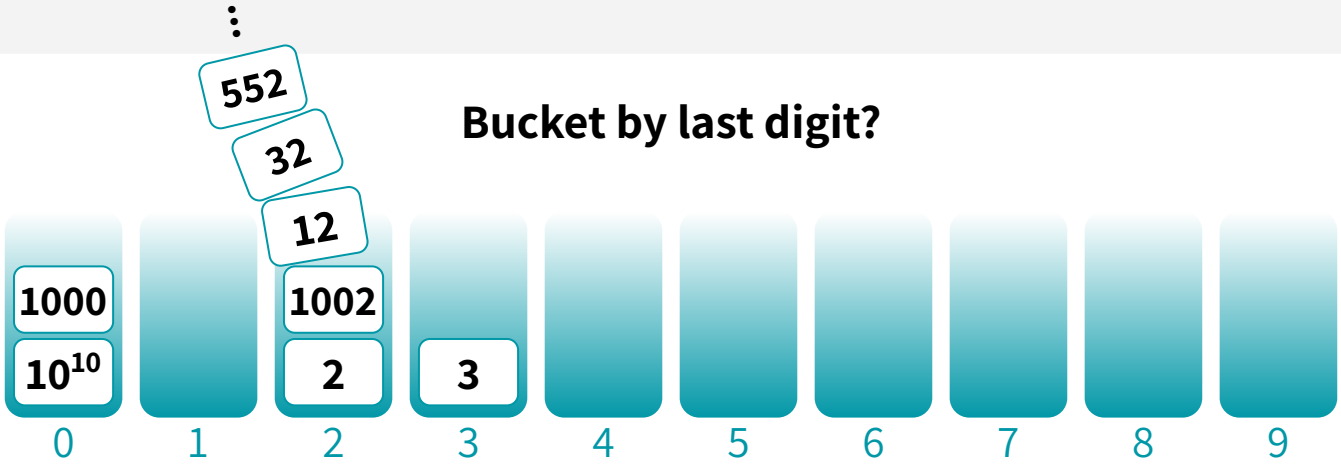
Bucket by last digit?



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:

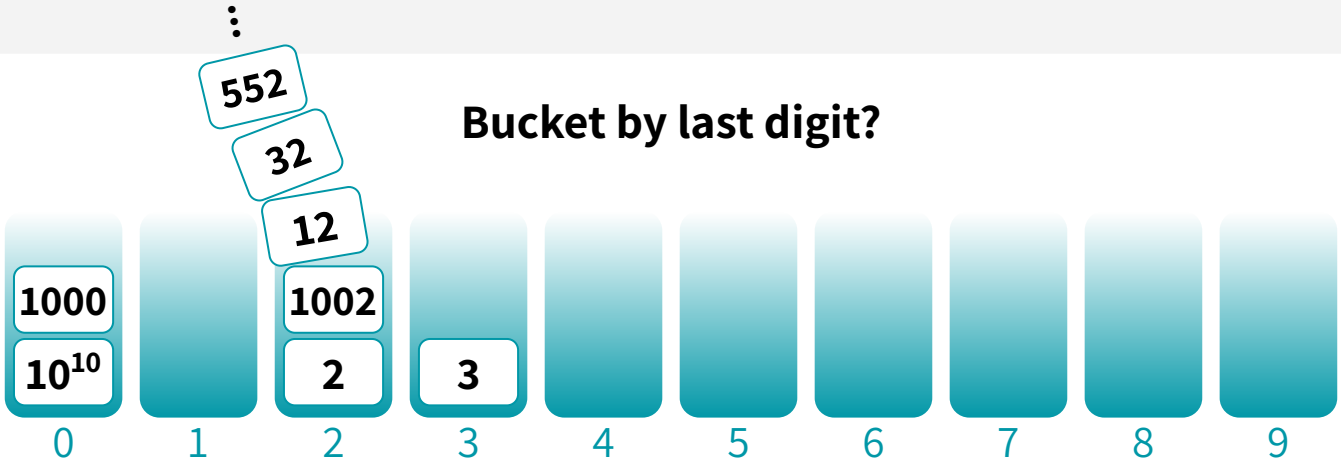
Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes...



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...



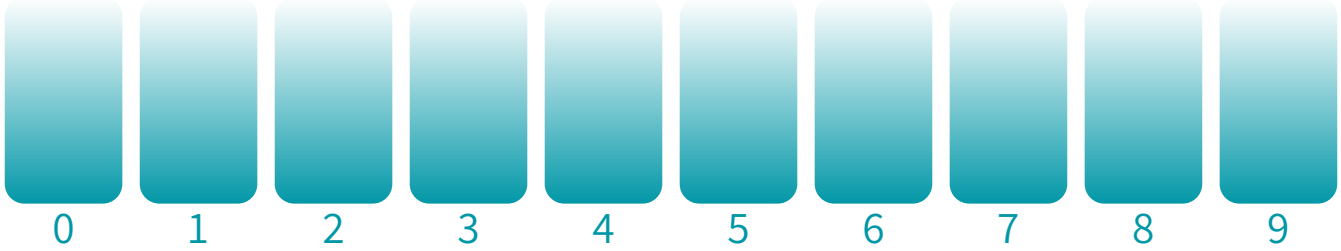
O(1) INSERT: Just index into the bucket (& insert at front of a linked list)!
O(n) SEARCH/DELETE: Go visit bucket & search through until you find it...

Maybe another bucketing scheme?

BUCKETING ATTEMPT 2:

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

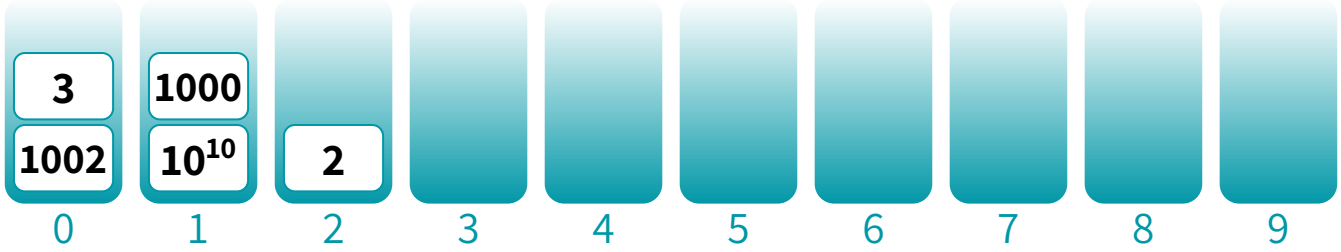
Bucket by last digit of (number * 7) mod 3



BUCKETING ATTEMPT 2:

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

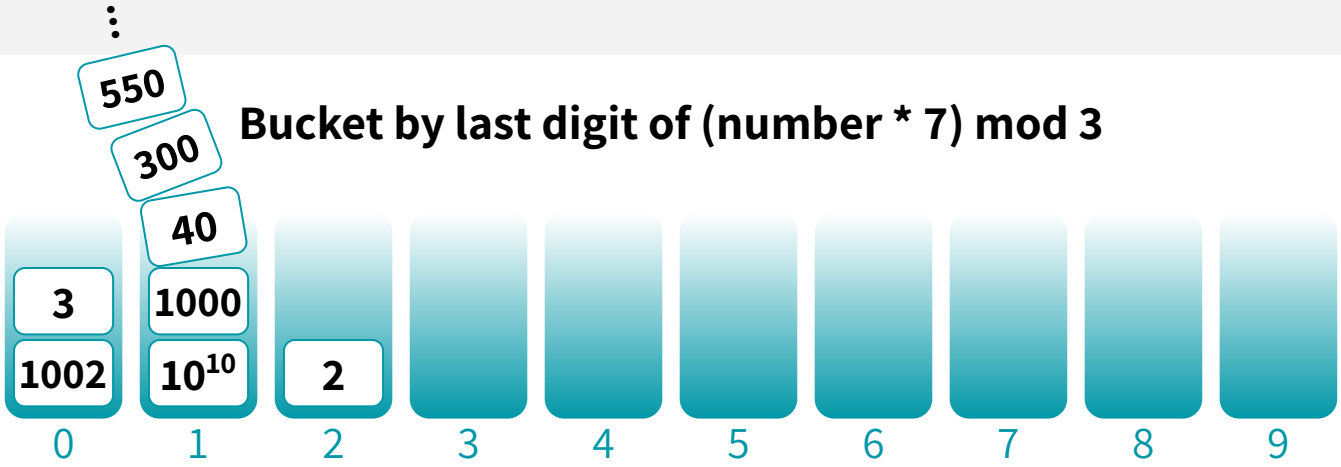
Bucket by last digit of (number * 7) mod 3



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:

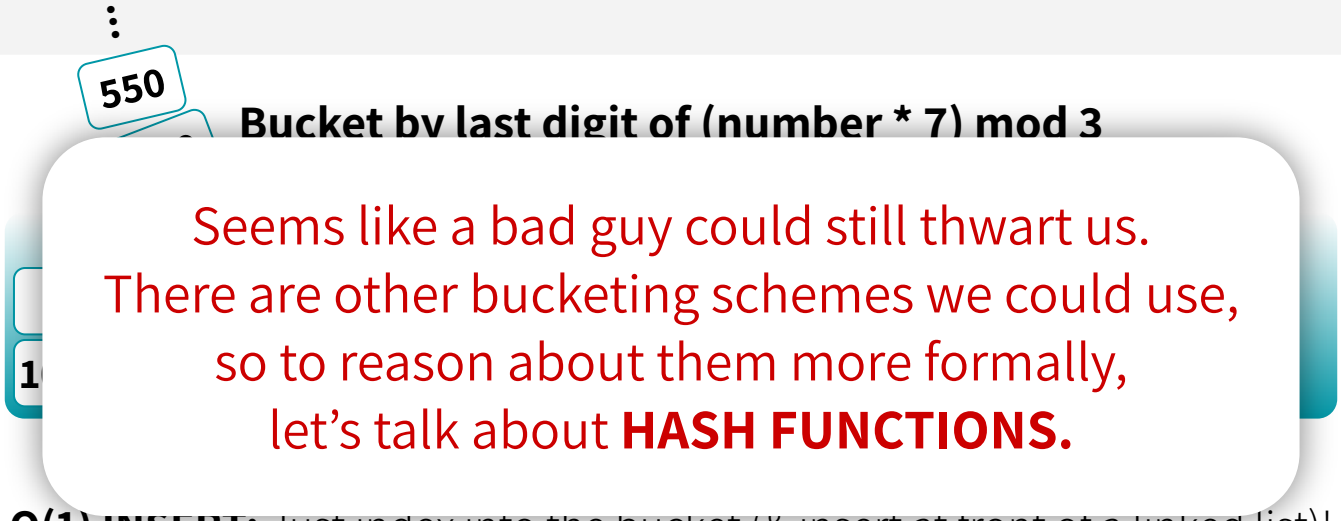
Under this scheme, a bad guy could give us inputs that yields quite ugly worst-case runtimes...



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...



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...

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 = 8.6 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 = 8.6 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 = 8.6 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 how up in advance.

maps elements of U to buckets 1, …, n

NOTE:

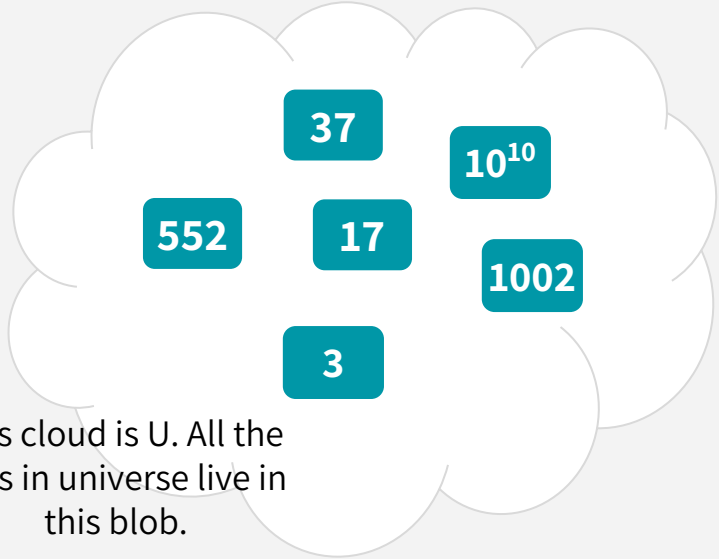
For this lecture, I'm assuming that the # of elements I receive is the same as the # of buckets (both are **n**). This doesn't have to be the case, but we usually aim for **#buckets = O(# elements that show up)** (otherwise, we're using "too much" space)

SOME TERMINOLOGY

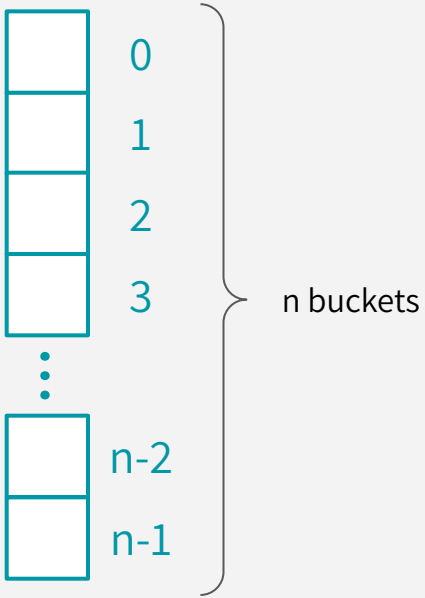
A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

SOME TERMINOLOGY

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

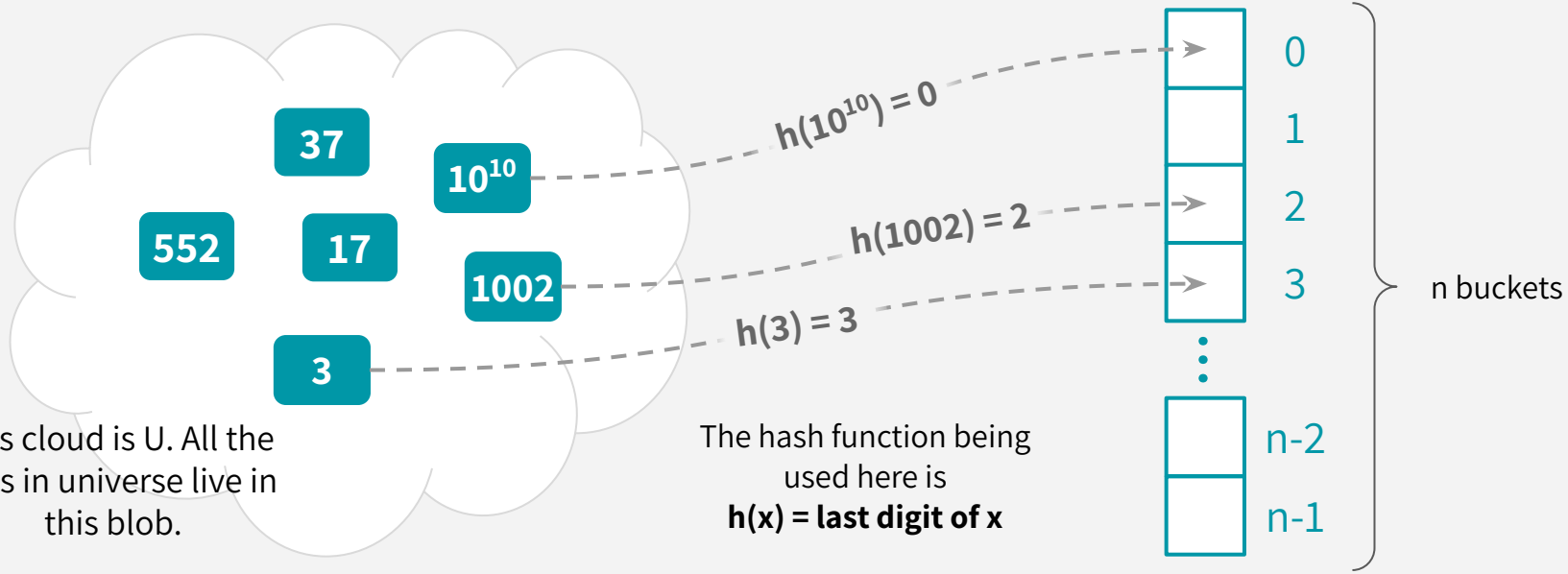


This cloud is U. All the keys in universe live in this blob.



SOME TERMINOLOGY

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

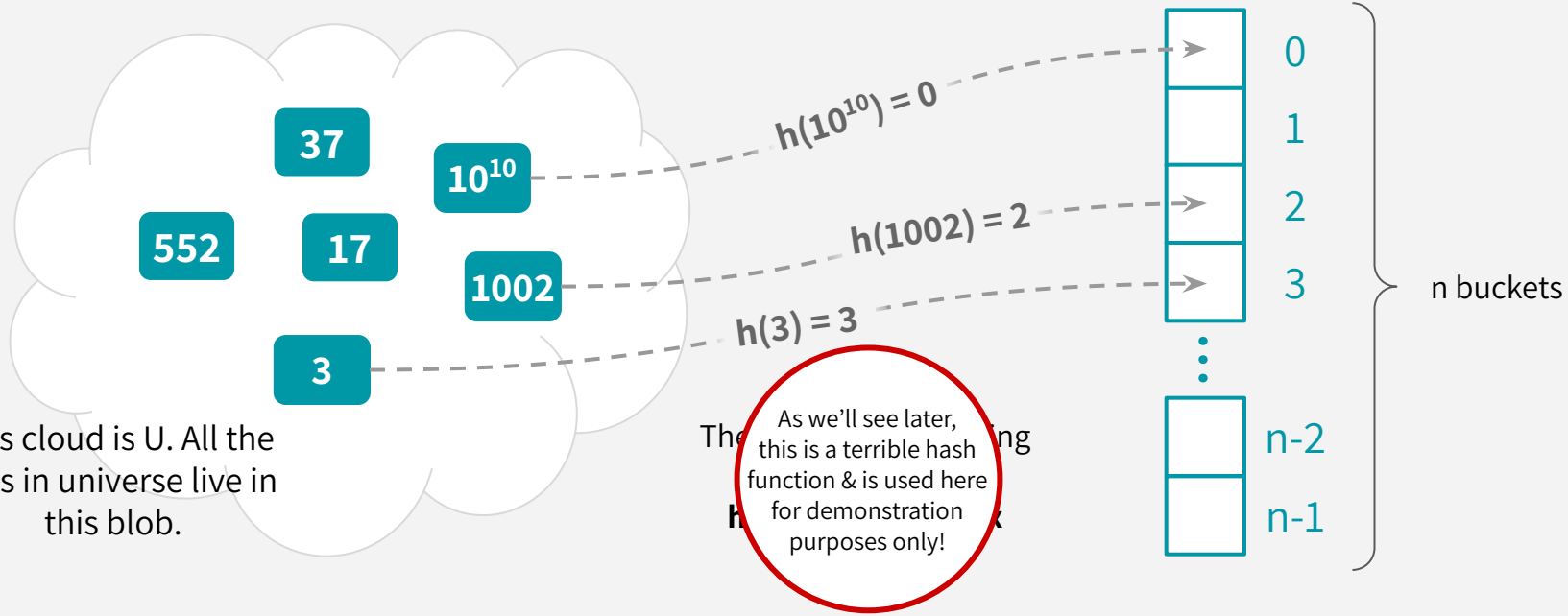


This cloud is U. All the keys in universe live in this blob.

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

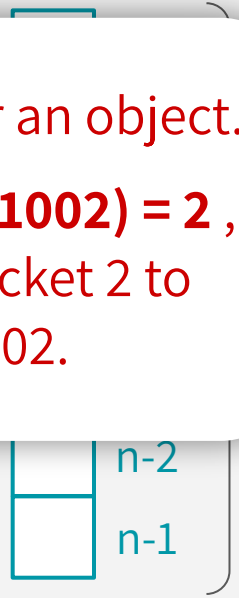
A hash function tells you where to start looking for an object.

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؟

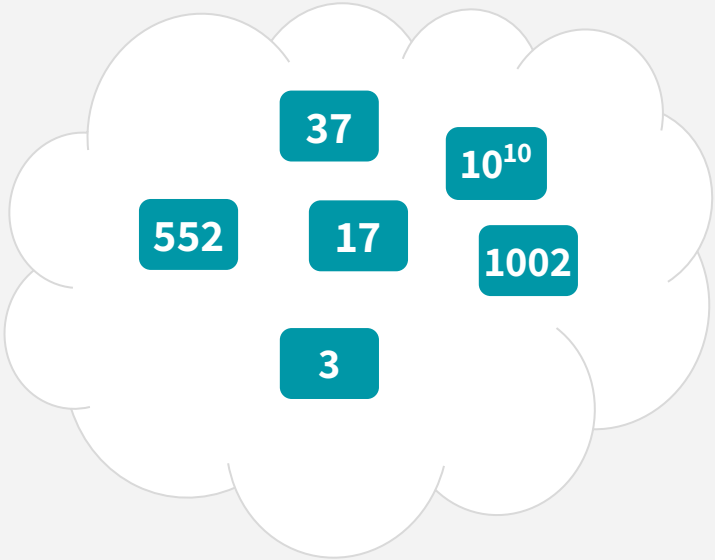
برخورد در درهم ساز!

چه مشکلی ممکن است پیش بیاید؟

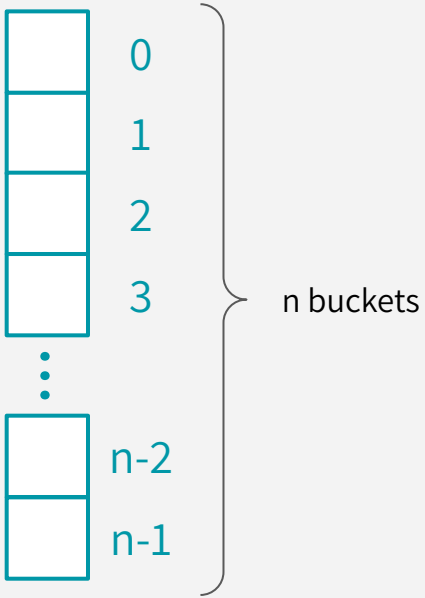
COLLISIONS

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.



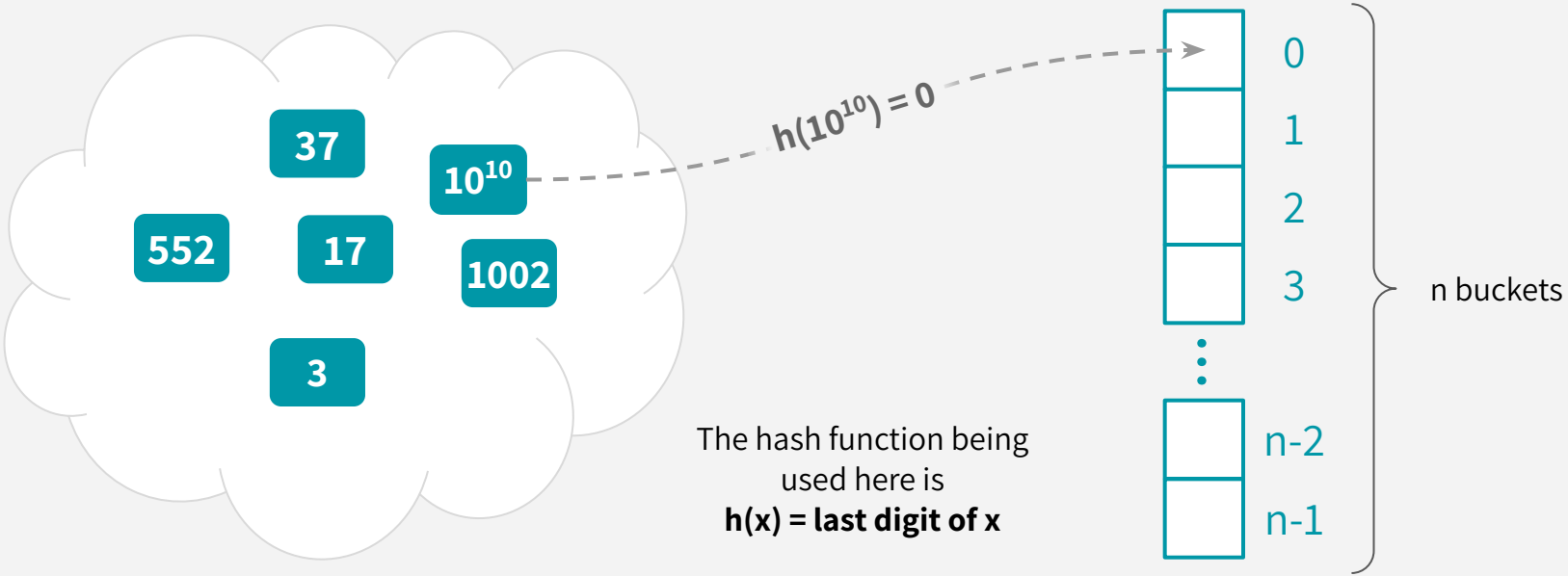
The hash function being used here is
h(x) = last digit of x



COLLISIONS

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

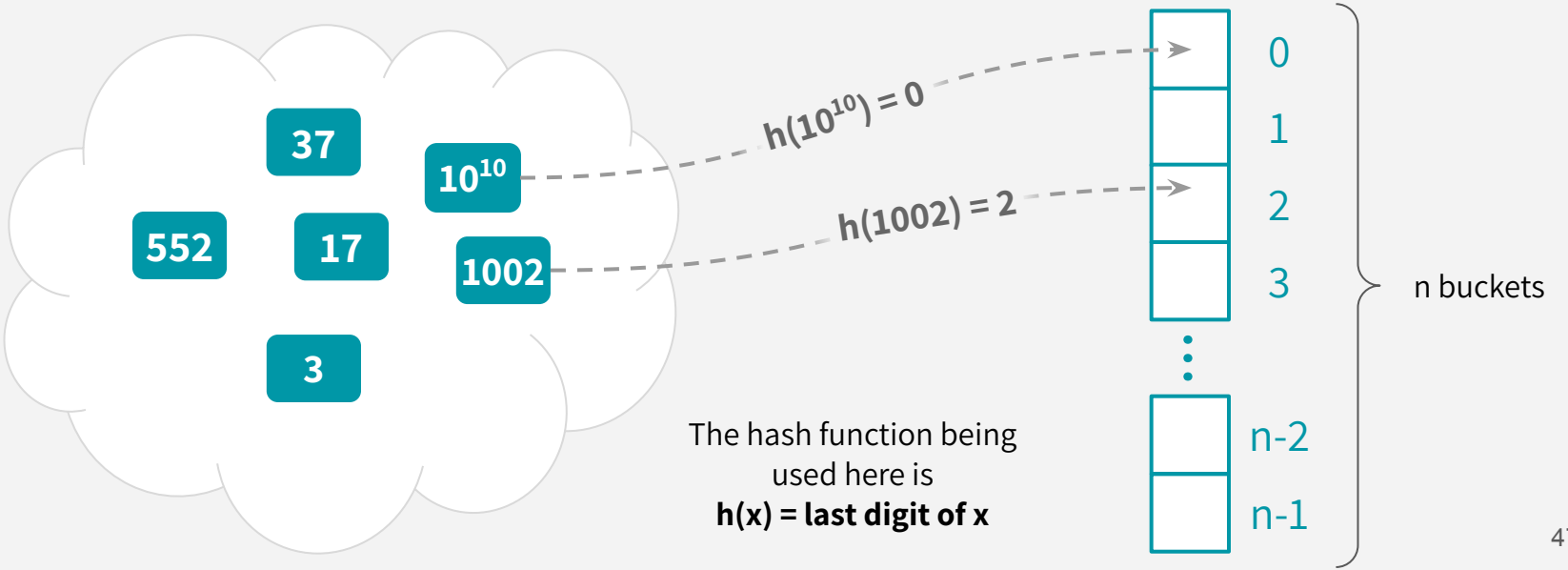
This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.



COLLISIONS

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

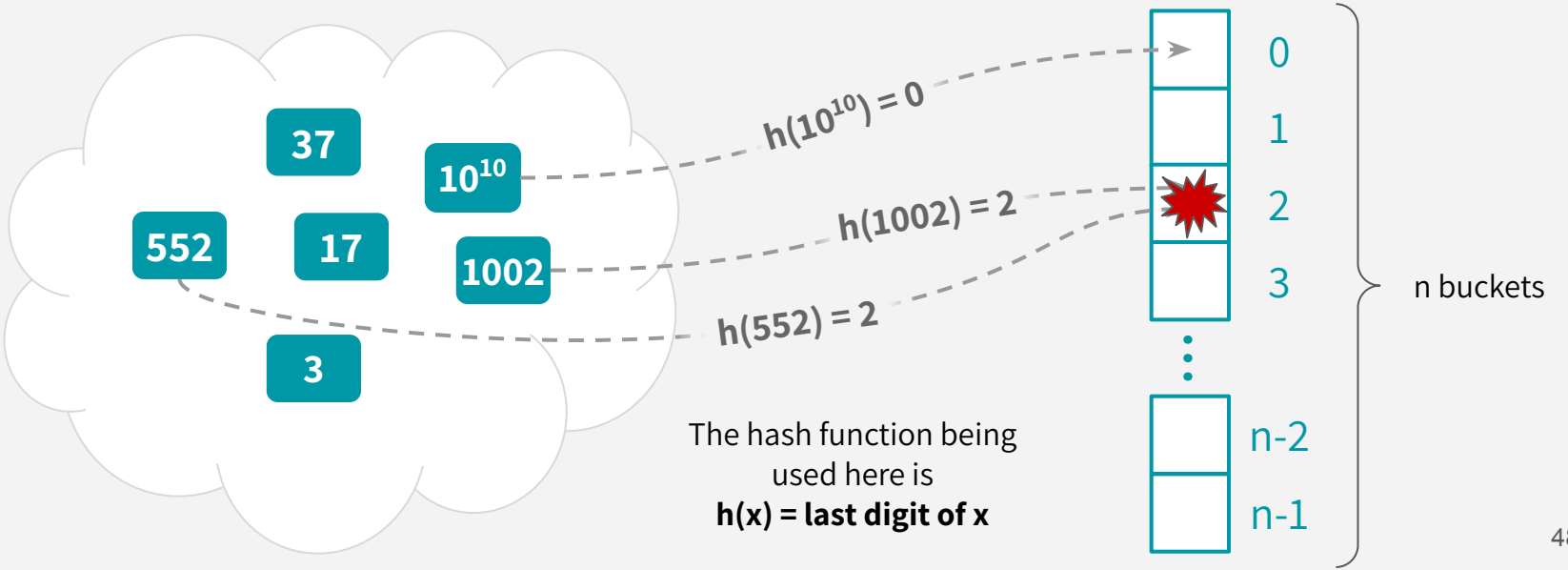
This is because of the *Pigeonhole Principle*. Since the size of universe U > # of buckets, every hash function (no matter how clever), suffers from at least one collision.



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.



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)

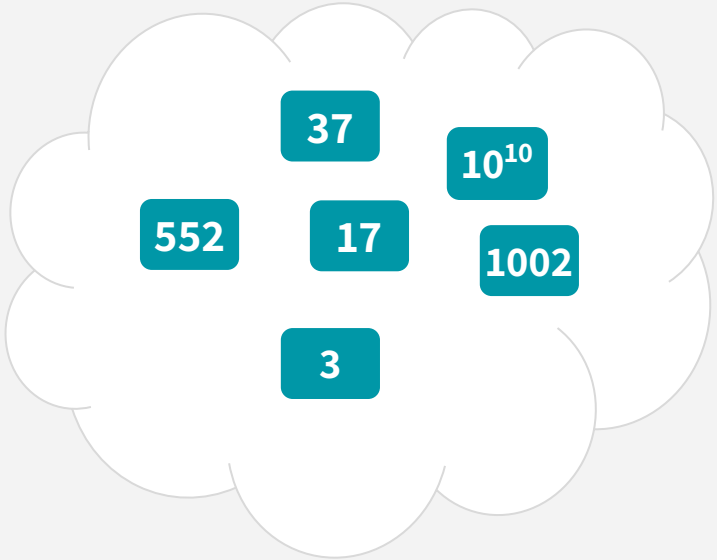
We're just giving a formal name to our bucketing example from earlier: represent each bucket's contents as a *linked list*!

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*!



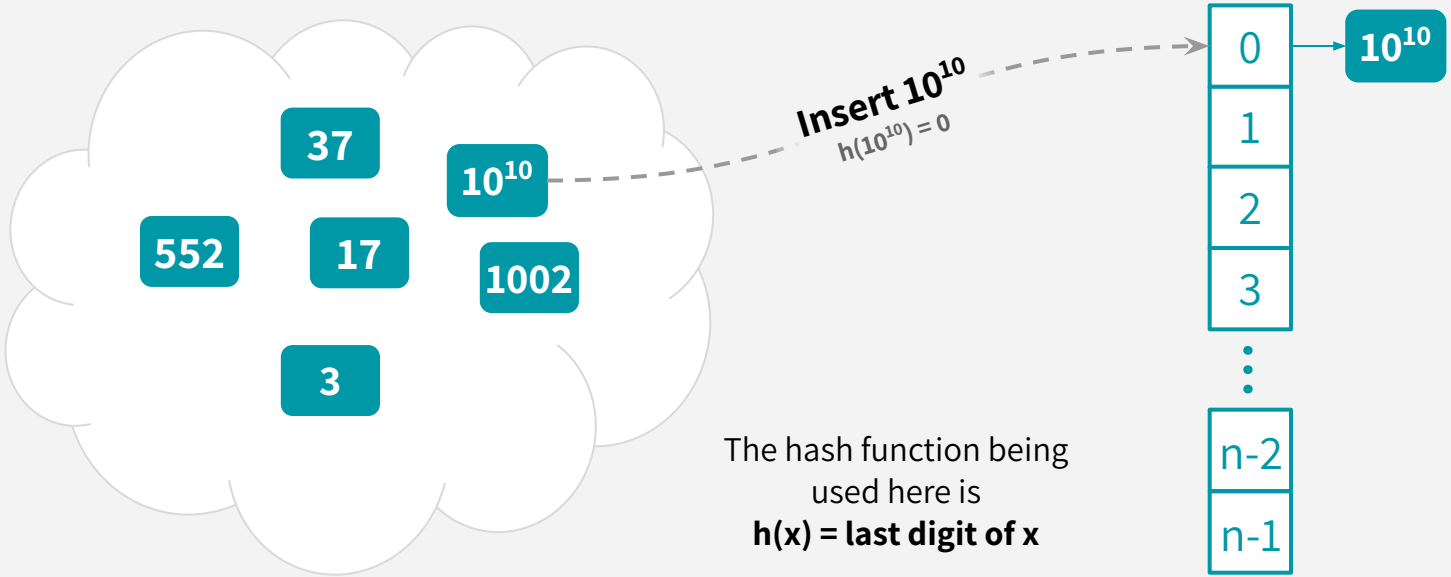
The hash function being used here is
h(x) = last digit of x

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*!



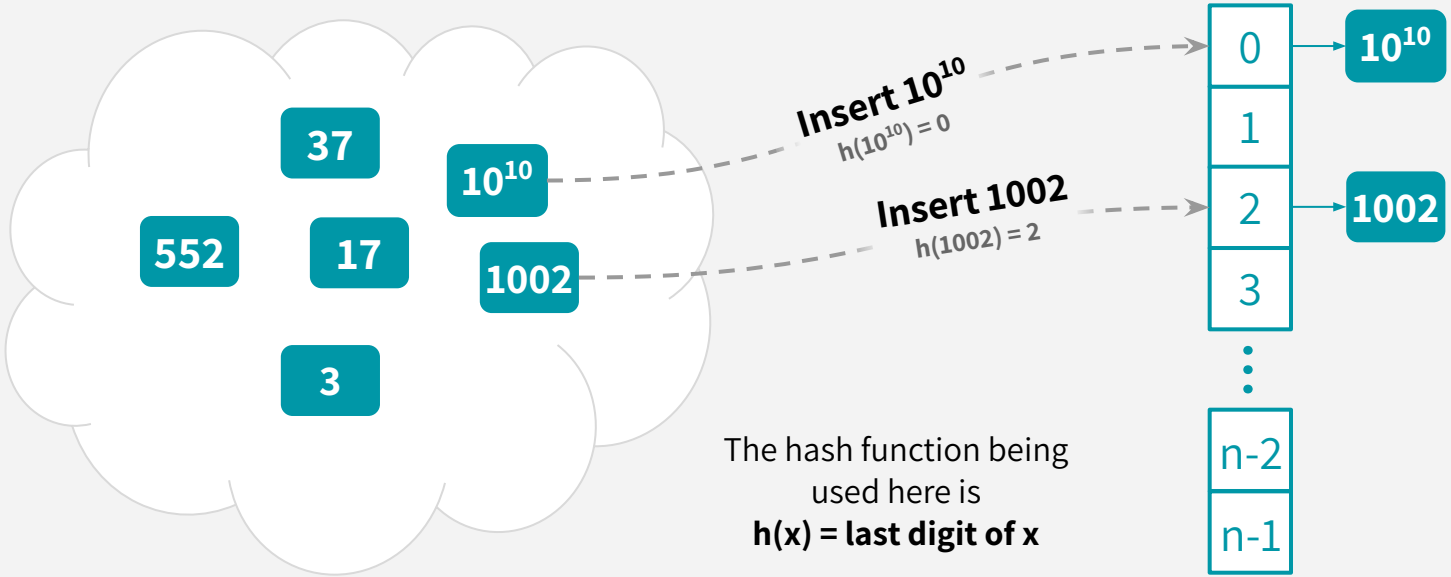
The hash function being used here is
h(x) = last digit of x

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*!



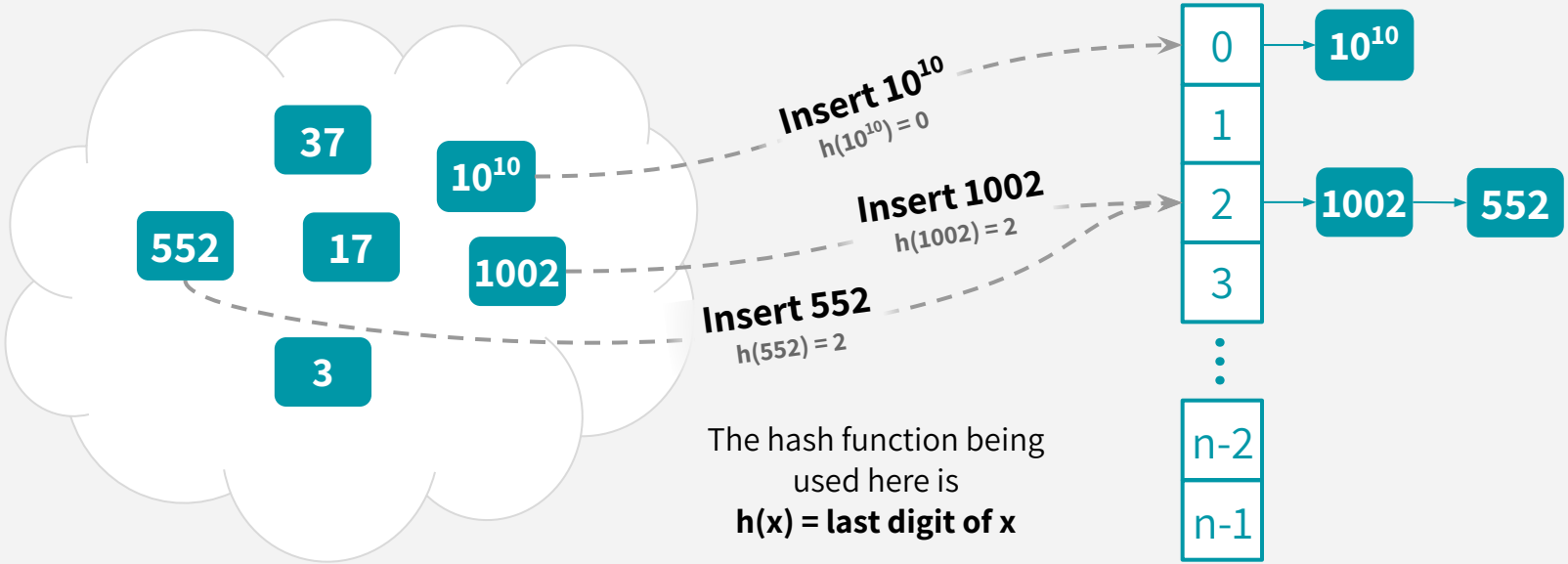
The hash function being used here is
h(x) = last digit of x

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

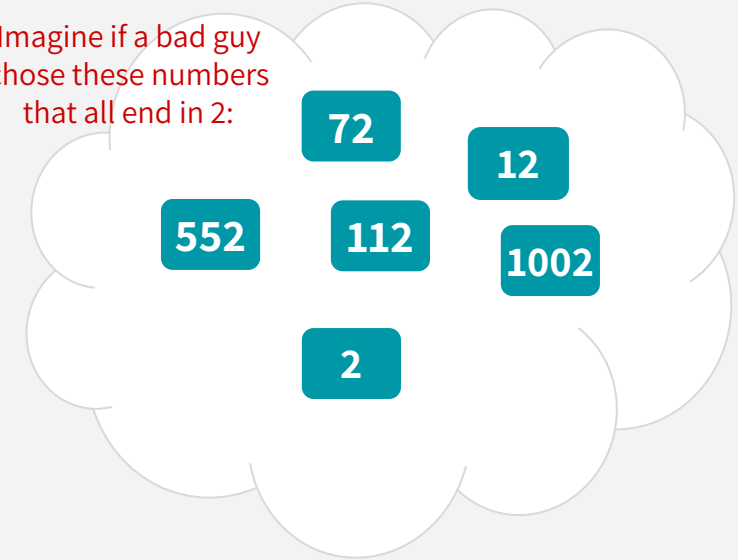
COLLISION RESOLUTION: CHAINING

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:

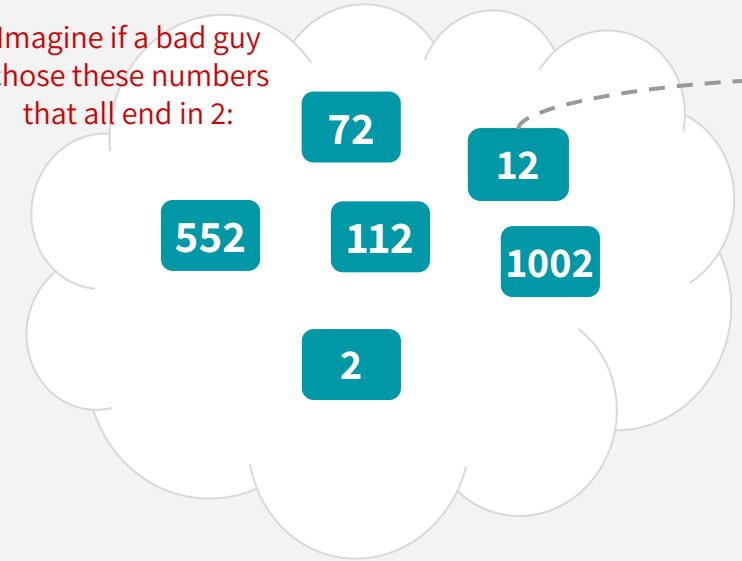


The hash function being used here is
h(x) = last digit of x

COLLISION RESOLUTION: CHAINING

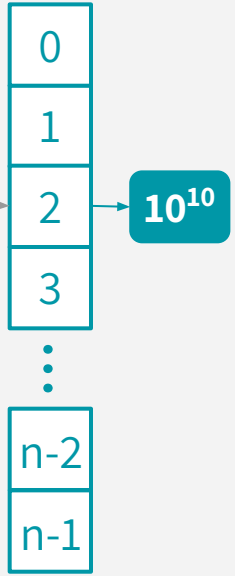
But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal…

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



Insert 12

h(12) = 2

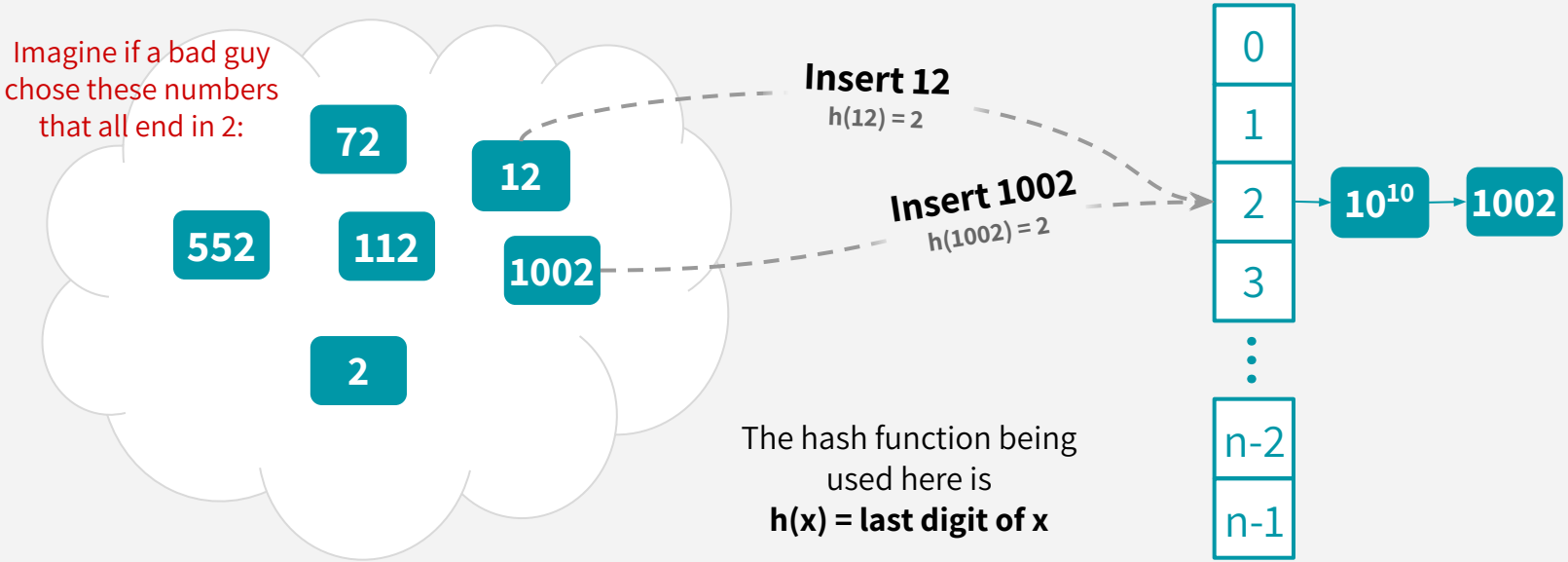


The hash function being used here is

h(x) = last digit of x

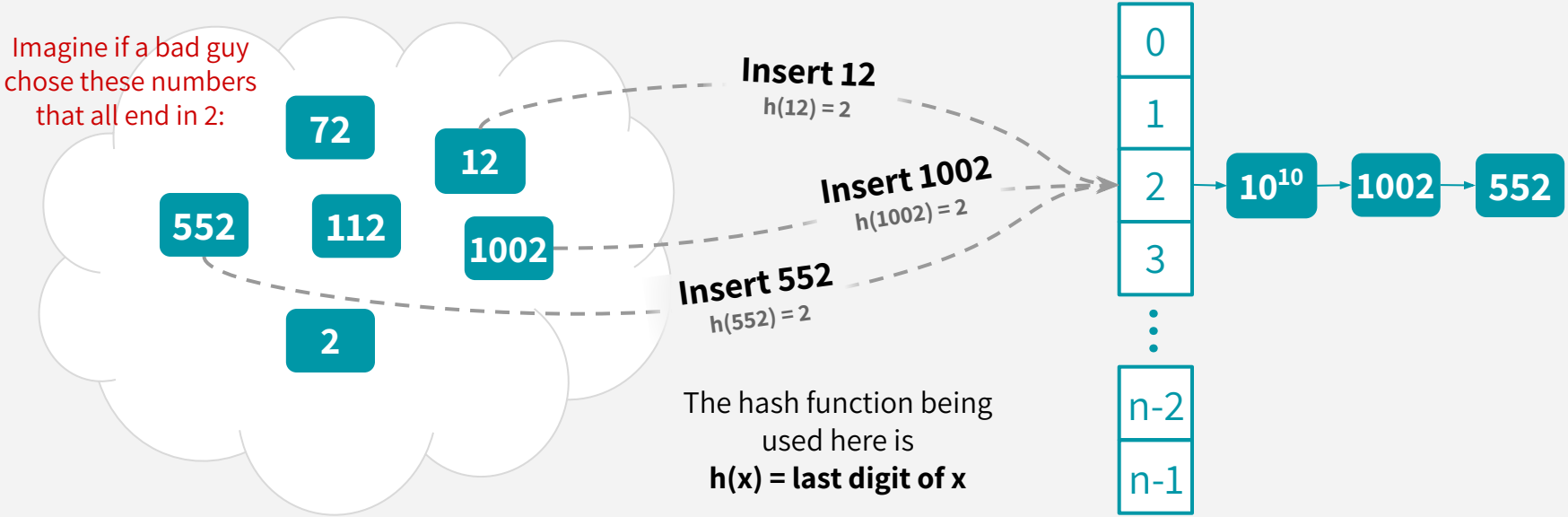
COLLISION RESOLUTION: CHAINING

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



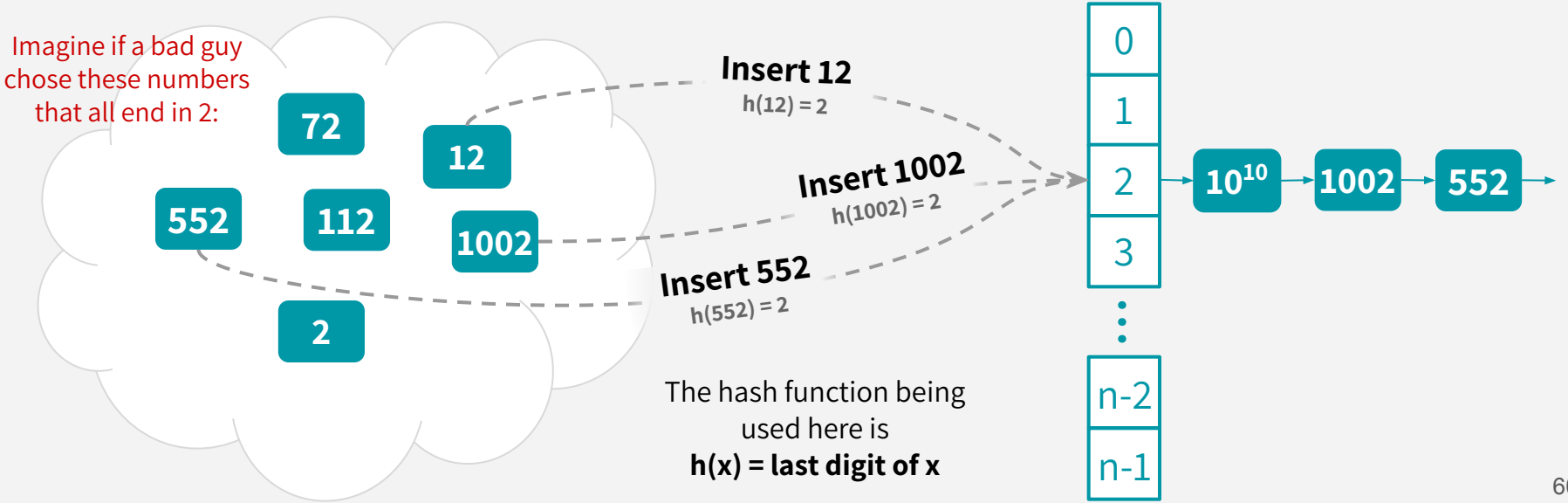
COLLISION RESOLUTION: CHAINING

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…



سوال؟

درهم سازی اعلی!

هدف جدول درهم سازی چیست؟

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)

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

***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

***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**

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.**

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

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…

LET'S BRING IN SOME

RANDOMNESS!

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!

In next class!

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

سوال؟