Indian Institute of Information Technology Allahabad

Data Structures and Algorithms

Hashing

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

Today

- Hashing!
    - What operations are we trying to support?
    - Hash Functions
    - Dealing with collisions
    - What makes a good hash function?
    - *Universal* hash families are what we're looking for!

Hash Tables Overview

What operations does it support?

The Task

Again, we want to keep track of objects that have keys 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)

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)

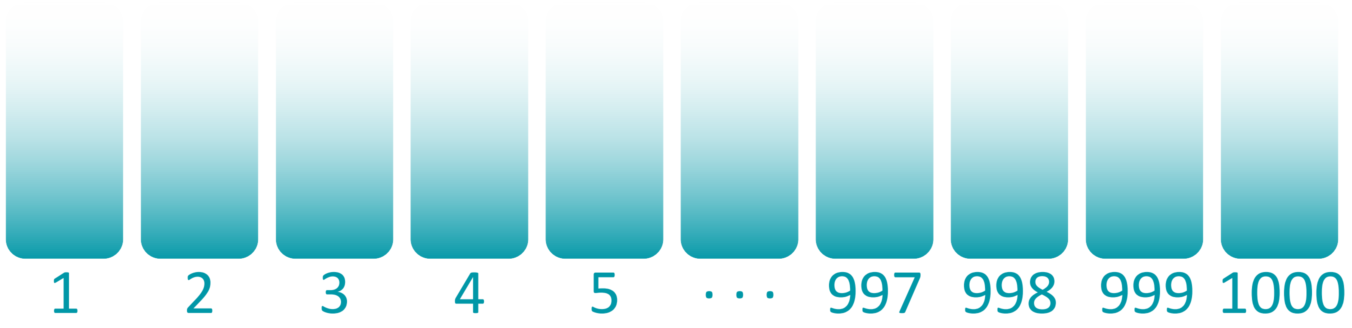
Attempt 1: Direct Addressing

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**

Not bad!

But what's the issue with this approach?

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)

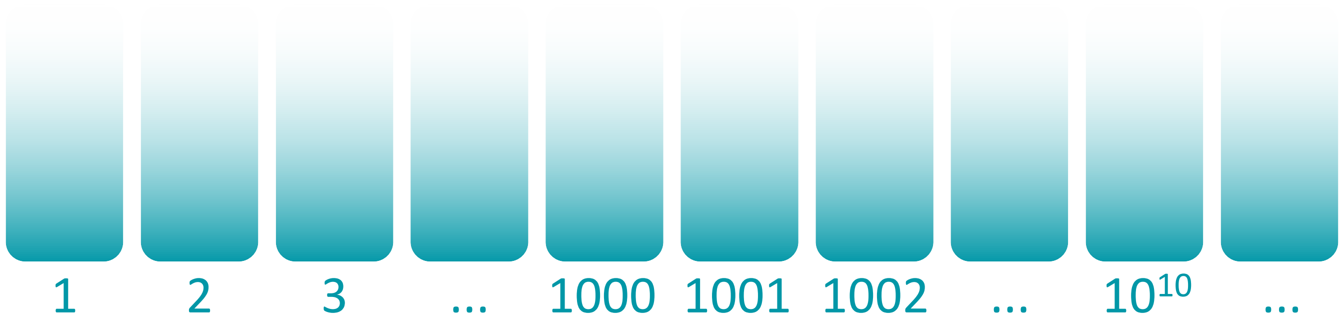
Attempt 1: Direct Addressing

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

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

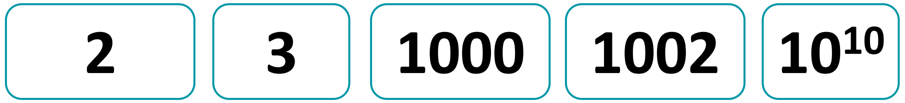
Reasonable Attempt (???): *Direct Addressing!*

(each address/bucket stores one type of item)



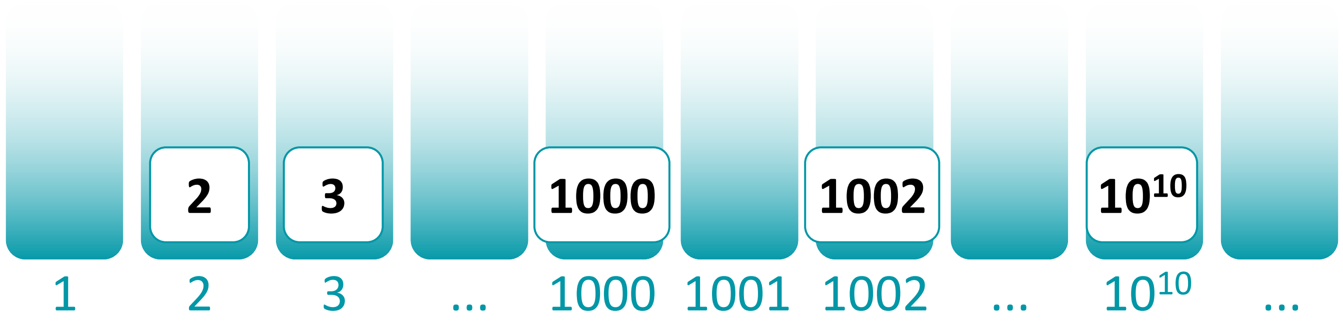
Attempt 1: Direct Addressing

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



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}**

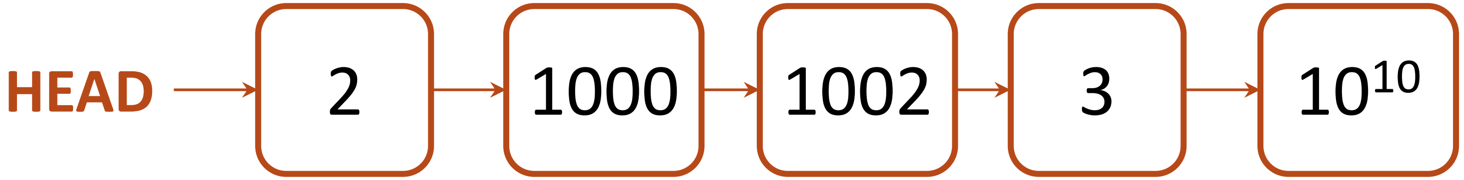
Reasonable Attempt (???): ***Direct Addressing!***

But the space requirement is HUGE...

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

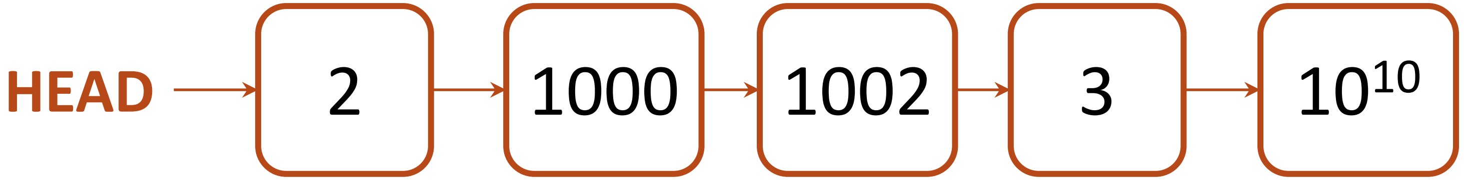
Attempt 2: Back to Linked List

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



Attempt 2: Back to Linked List

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 to 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

How to 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

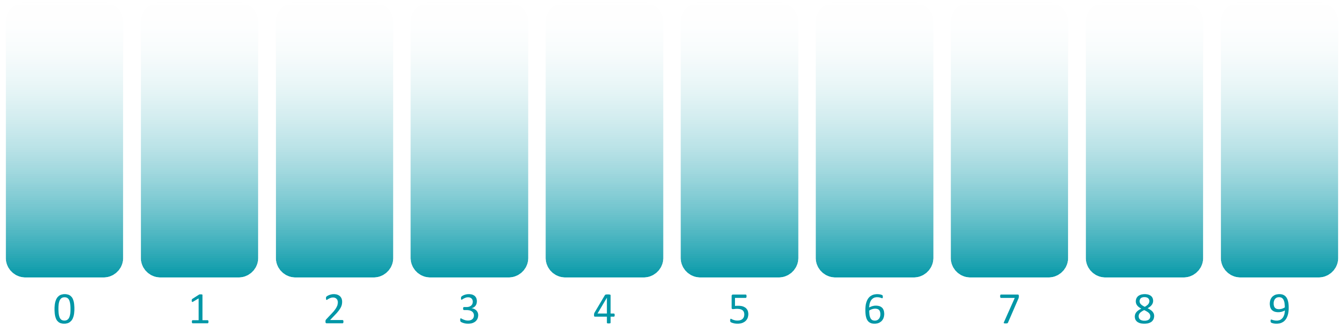
Let's try bucketing **by the least-significant digit**…

Bucketing Attempt 1

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

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

Bucket by last digit?

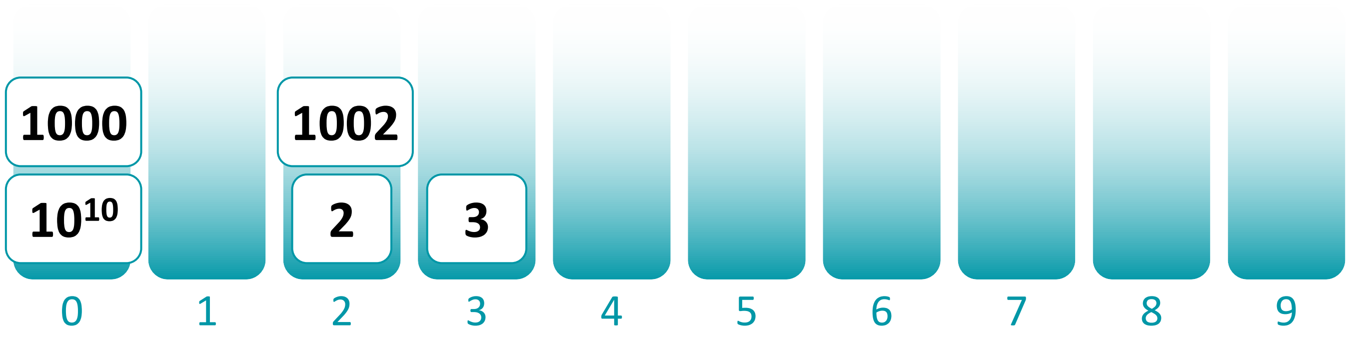


Bucketing Attempt 1

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

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

Bucket by last digit?

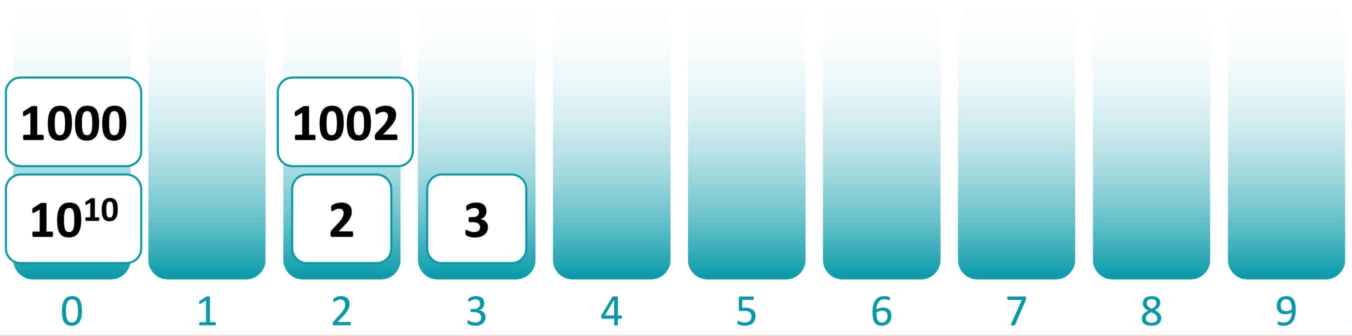


Bucketing Attempt 1

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

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

Bucket by last digit?



O(1) INSERT:

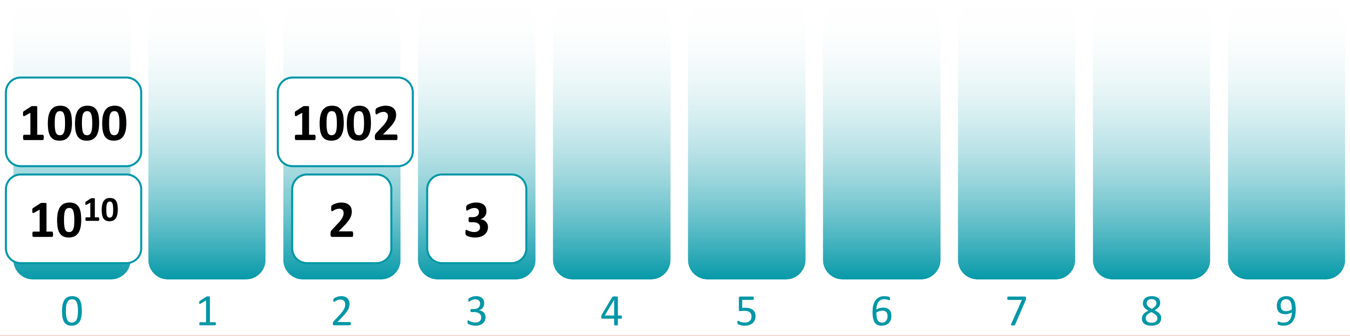
Just index into the bucket (& insert at front of a linked list)!

Bucketing Attempt 1

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

2	3	1000	1002	10^{10}

Bucket by last digit?

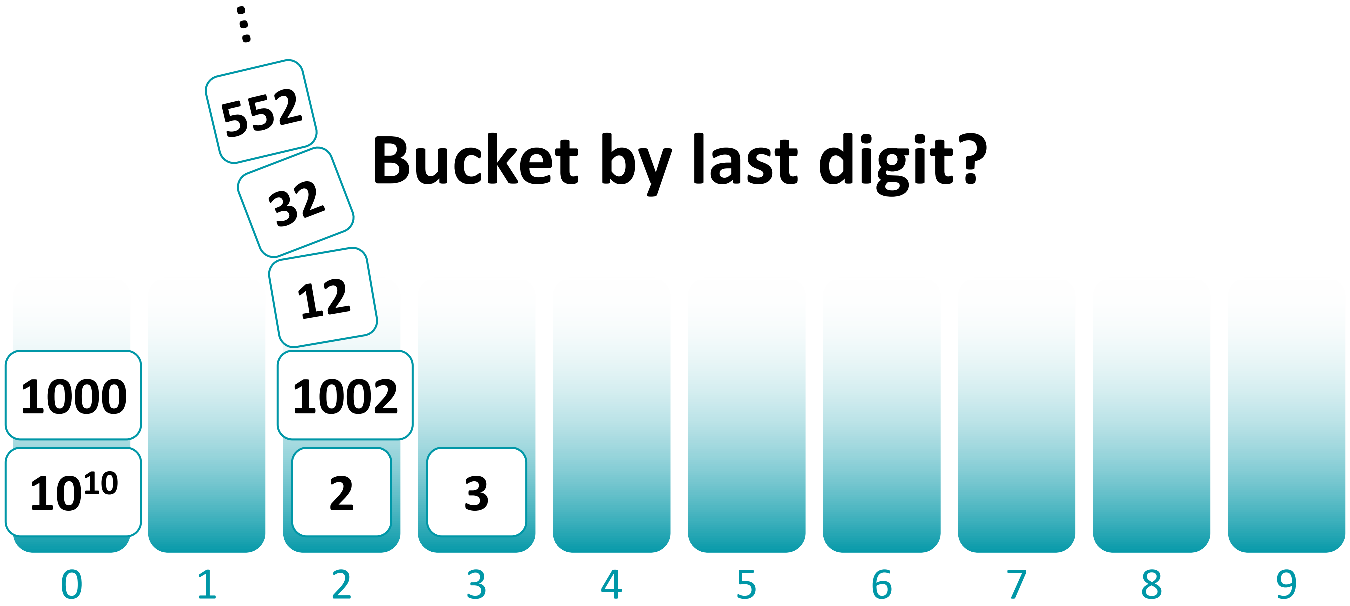


O(??????) SEARCH/DELETE:

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...

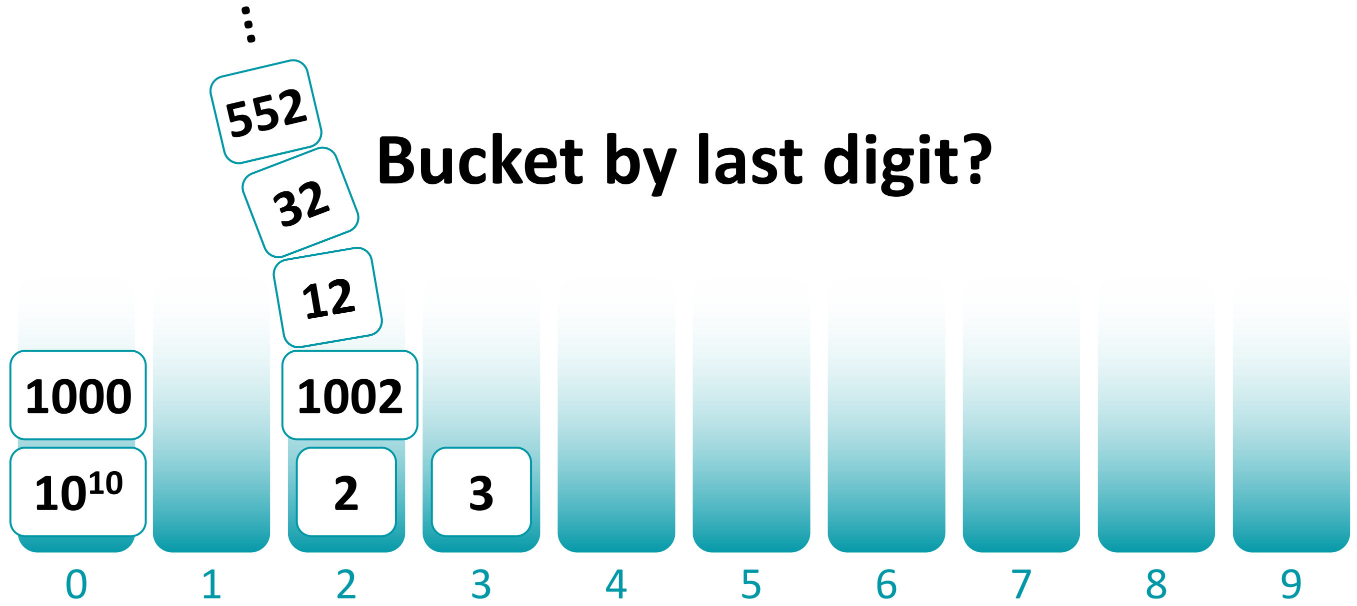


O(n) SEARCH/DELETE:

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...



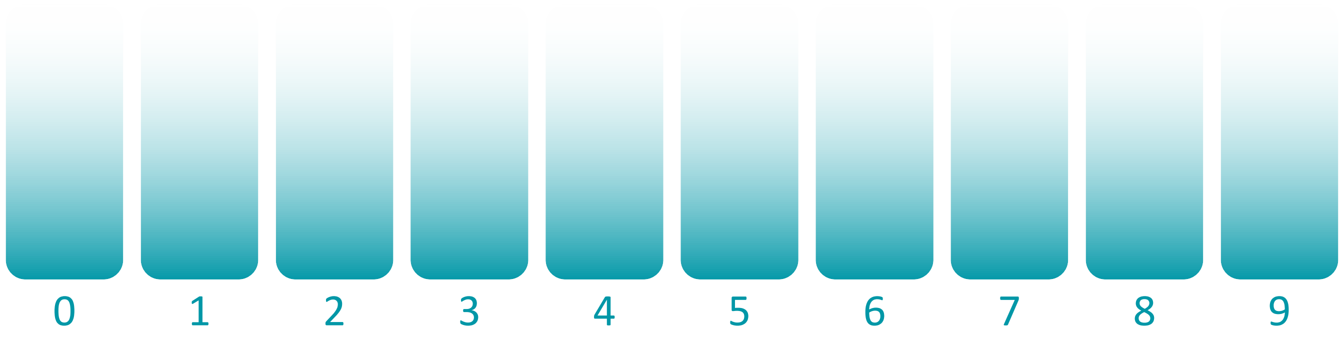
Maybe another bucketing scheme?

Bucketing Attempt 2

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

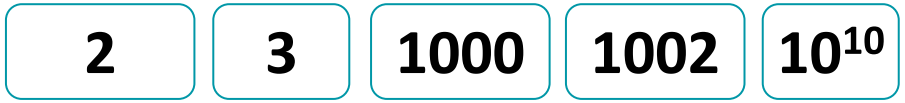
2 | 3 | 1000 | 1002 | 10^{10}

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

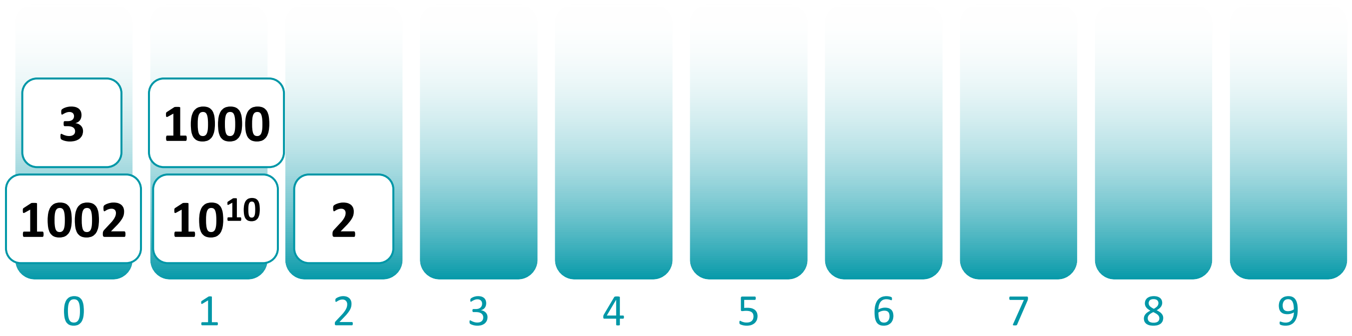


Bucketing Attempt 2

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

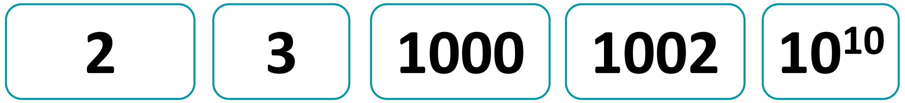


Bucket by last digit of (number * 7) mod 3

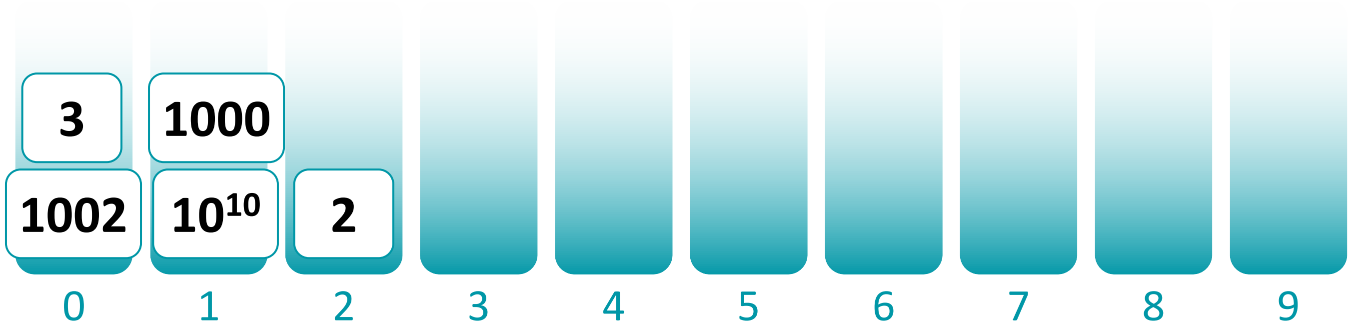


Bucketing Attempt 2

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



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



O(1) INSERT:

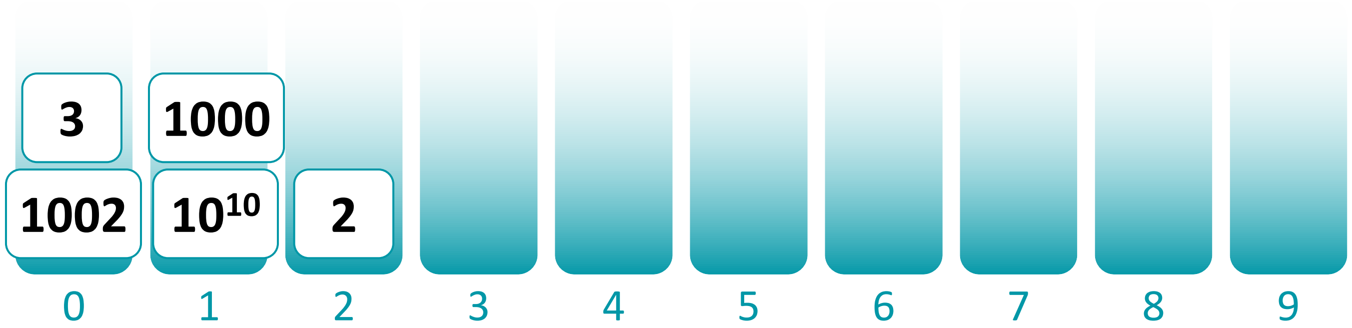
Just index into the bucket (& insert at front of a linked list)!

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

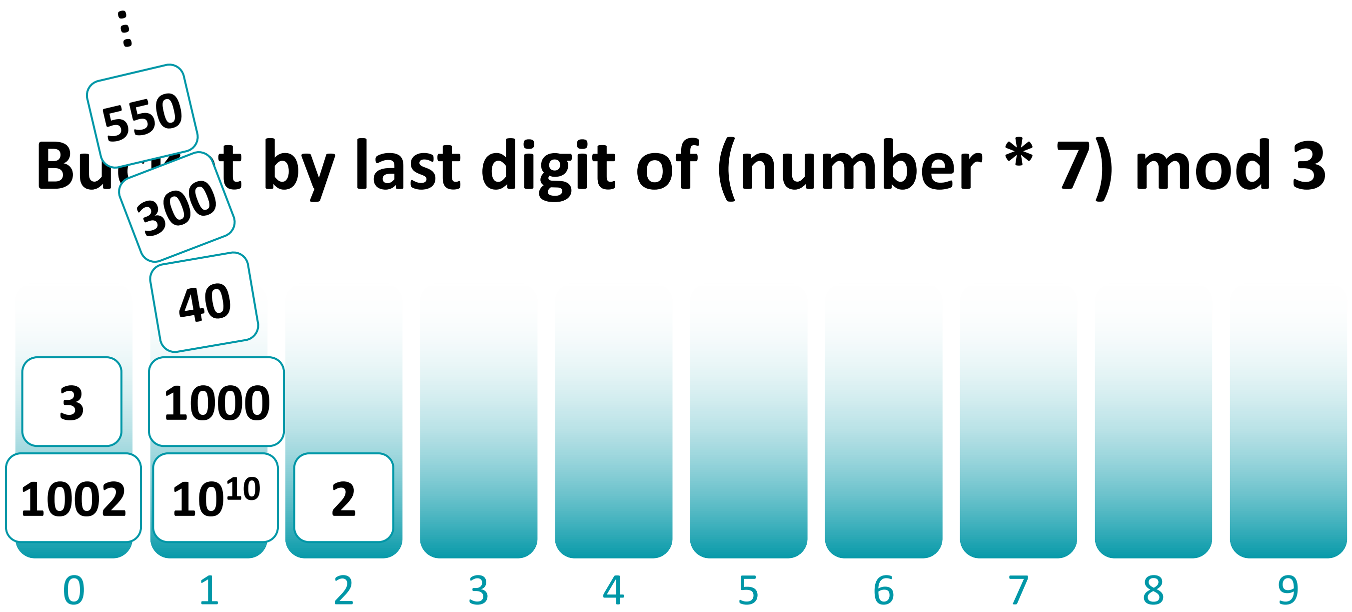


O(??????) SEARCH/DELETE:

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(n) 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.**

Go visit bucket & search through until you find it...

Hash Functions

What are “good” hash functions?

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 = 6.8 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 = 6.8 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 = 6.8 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

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

• U

• U

• U

Ou n

• c

• v

NOTE:

For this lecture, I'm assuming that the

# of elements I receive = # 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)

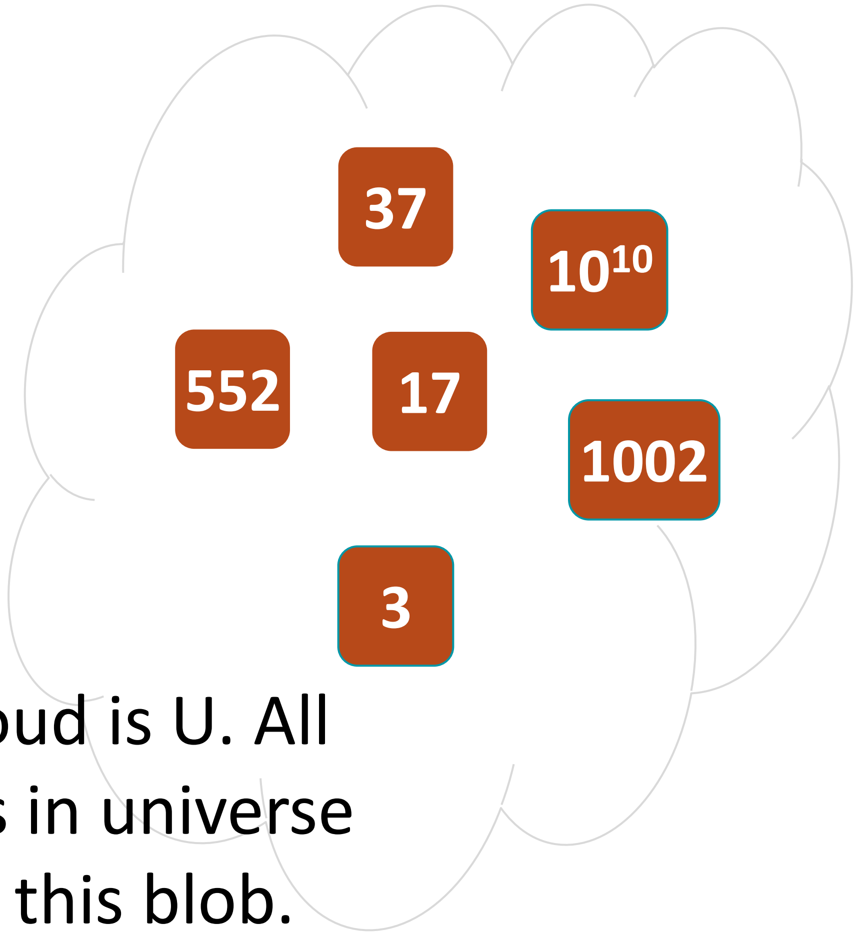
Some Terminology

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

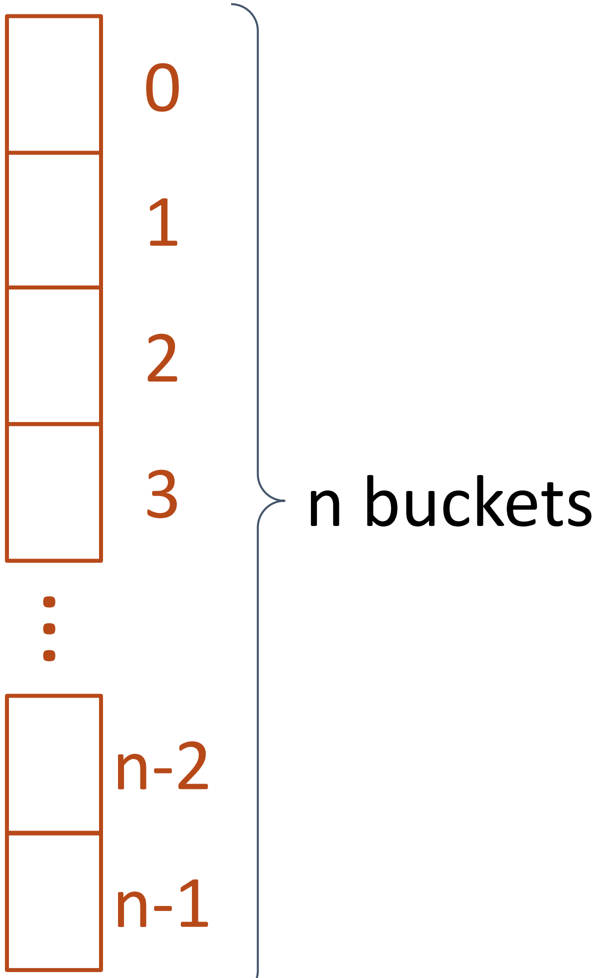
Some Terminology

A hash function **h: U → {1, …, n}**

maps elements of U to buckets 1, …, n

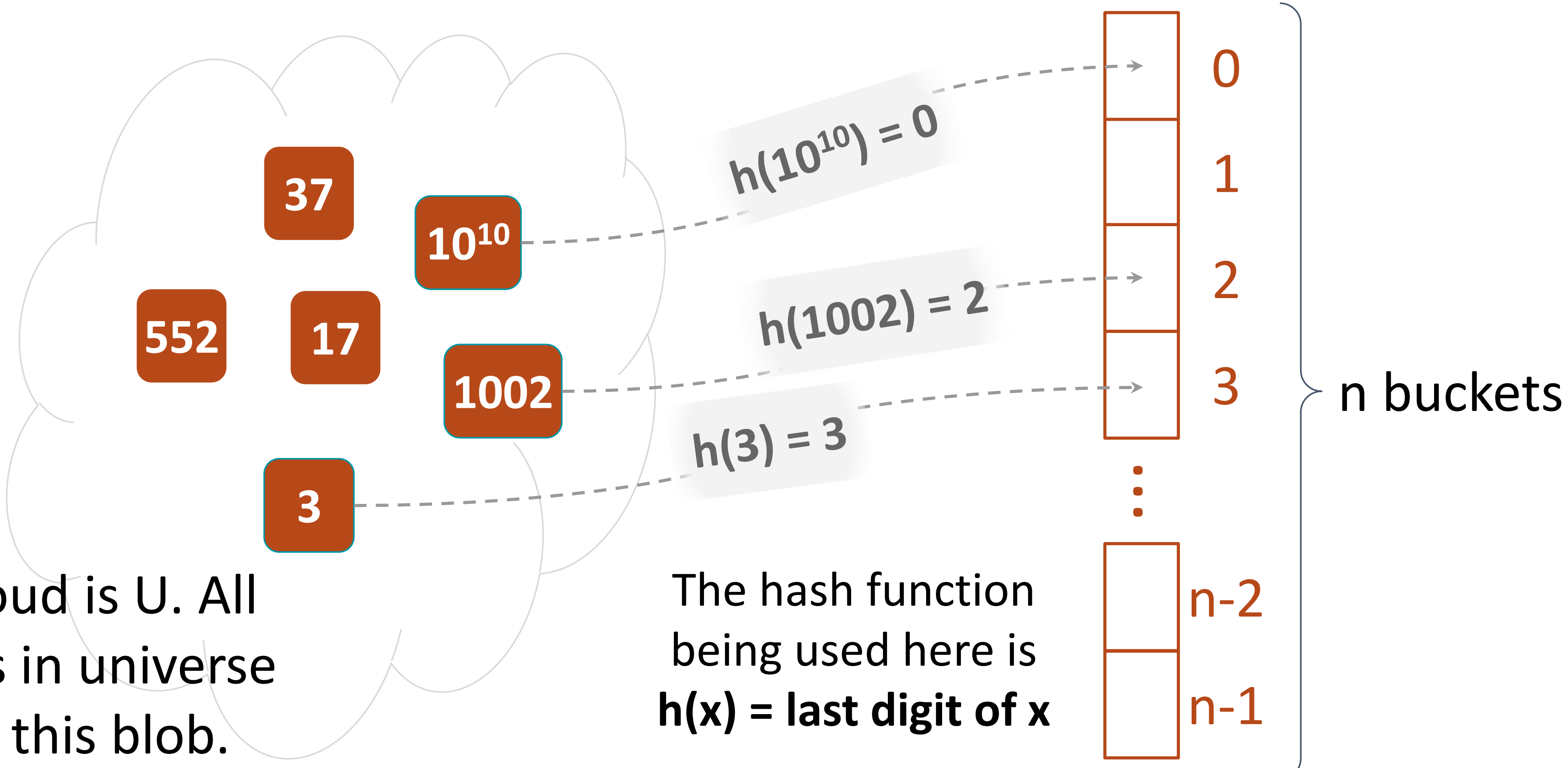


This cloud is U. All the keys in universe live in this blob.



Some Terminology

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

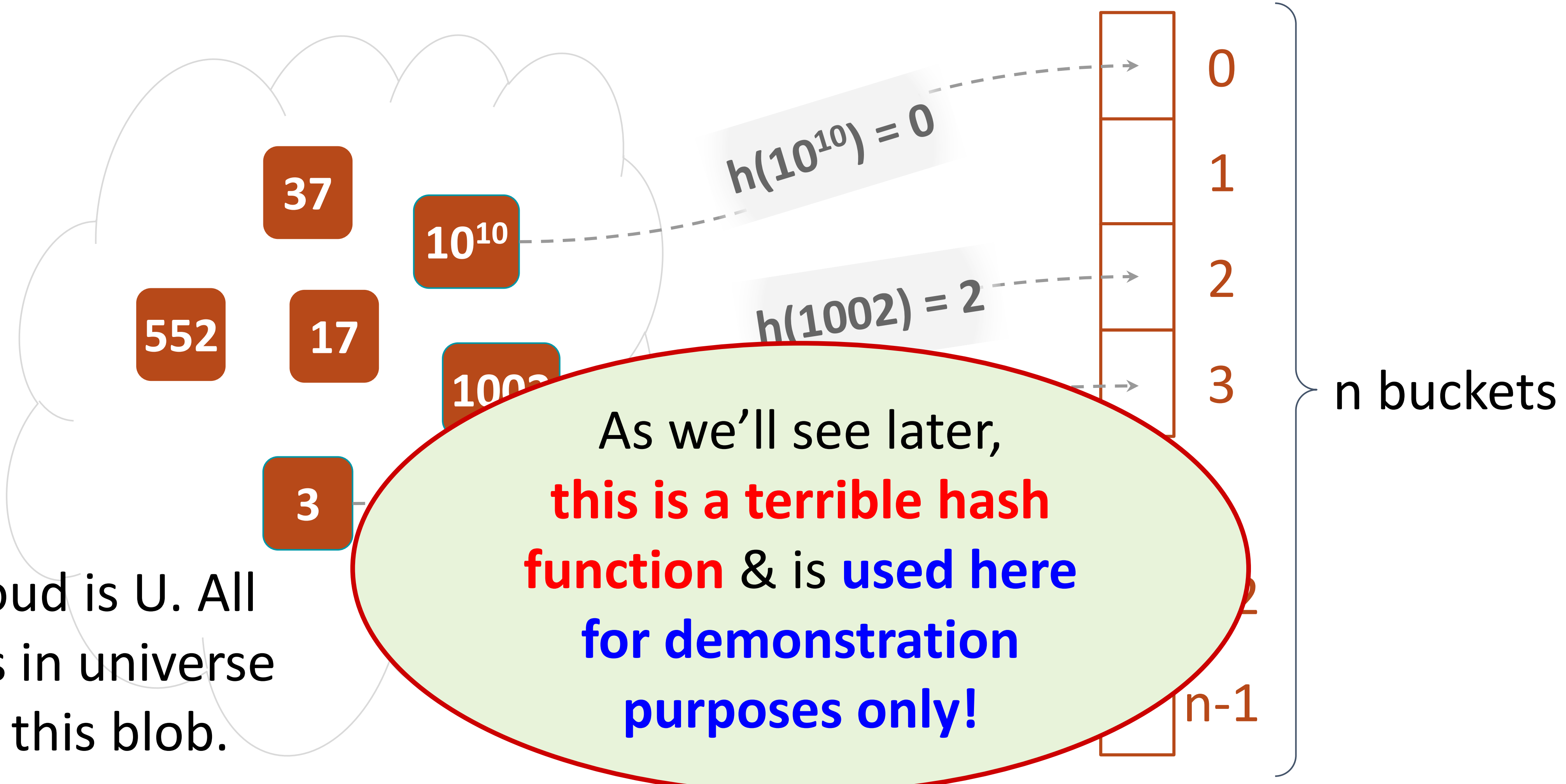


This cloud is U. All the keys in universe live in this blob.

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



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

As we'll see later, **this is a terrible hash function** & is **used here for demonstration purposes only!**

Some Terminology

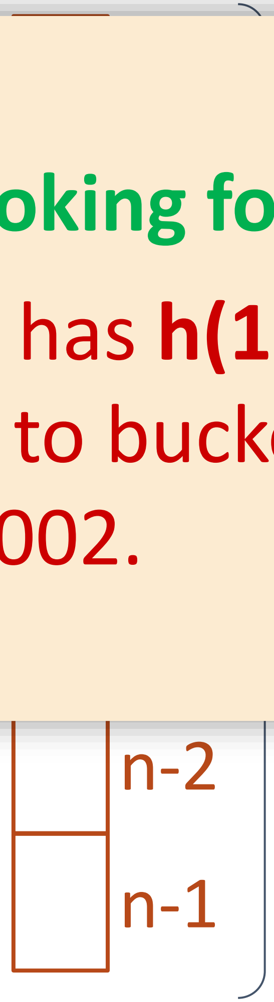
A hash function **h: U → {1, …, n}**

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.

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**

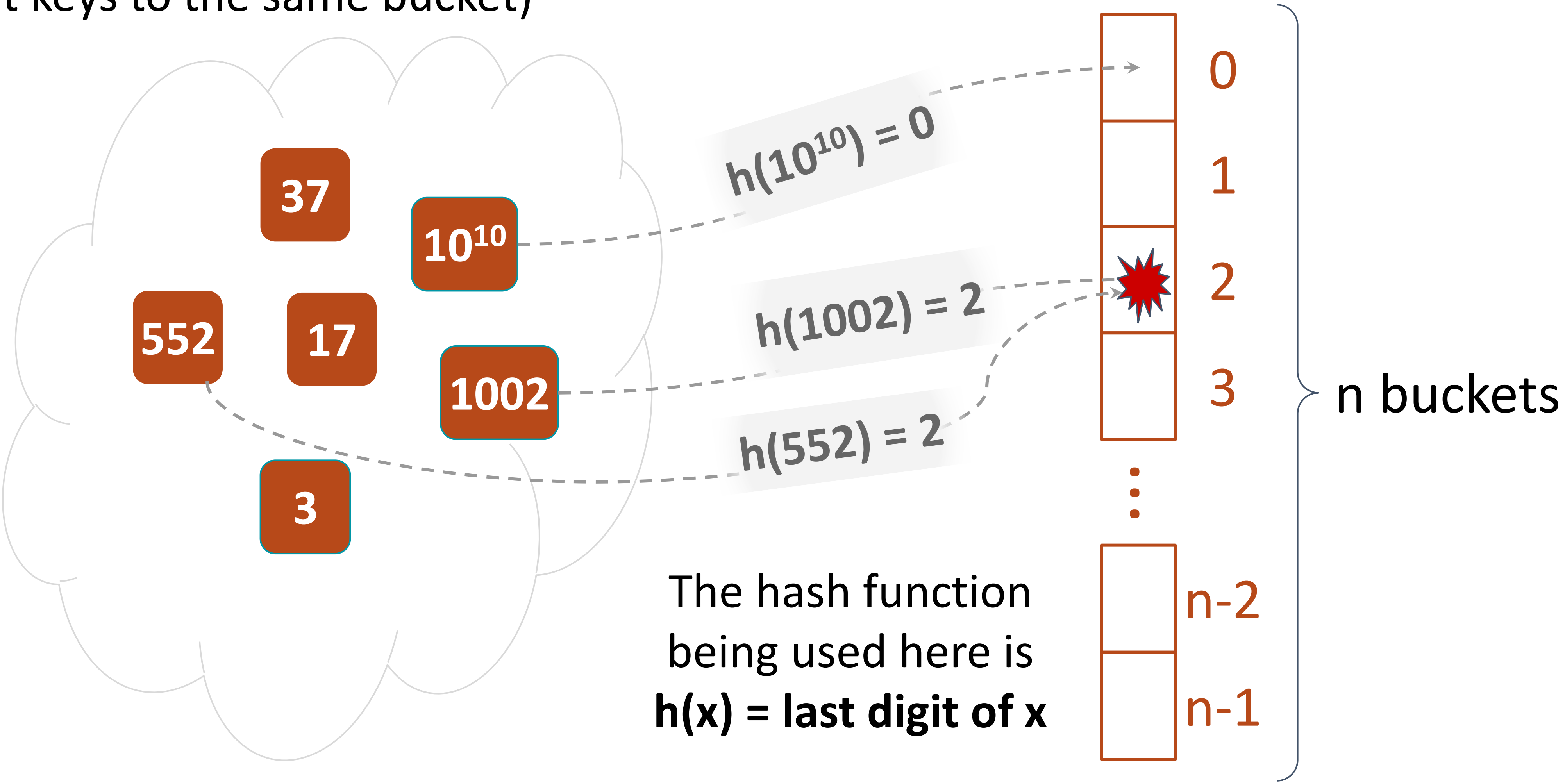


Collisions

Collisions are inevitable!

(when a hash function would map 2 different keys to the same bucket)

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*** !

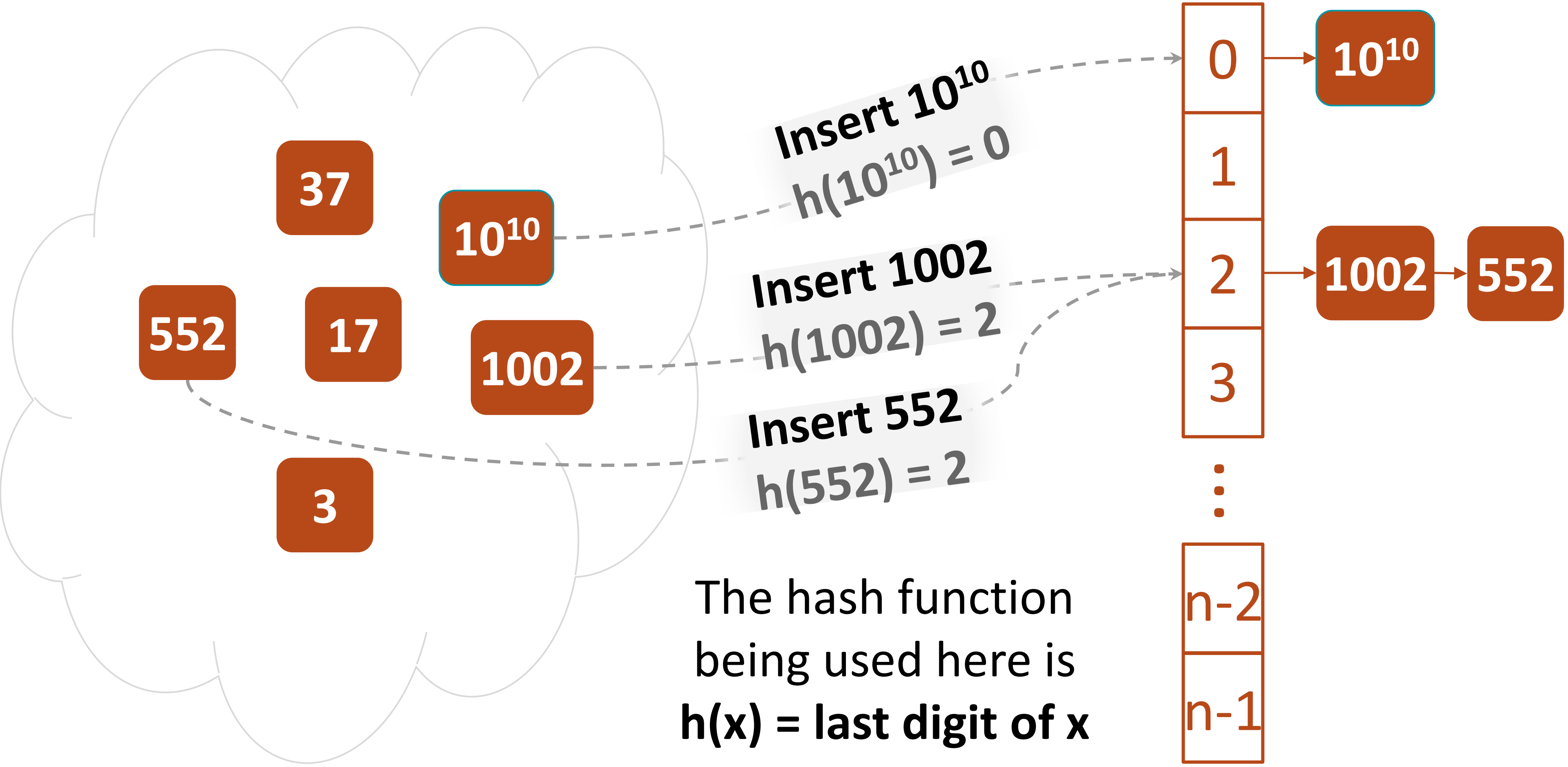
(Another method is called "Open Addressing")

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*** !

(Another method is called "Open Addressing")

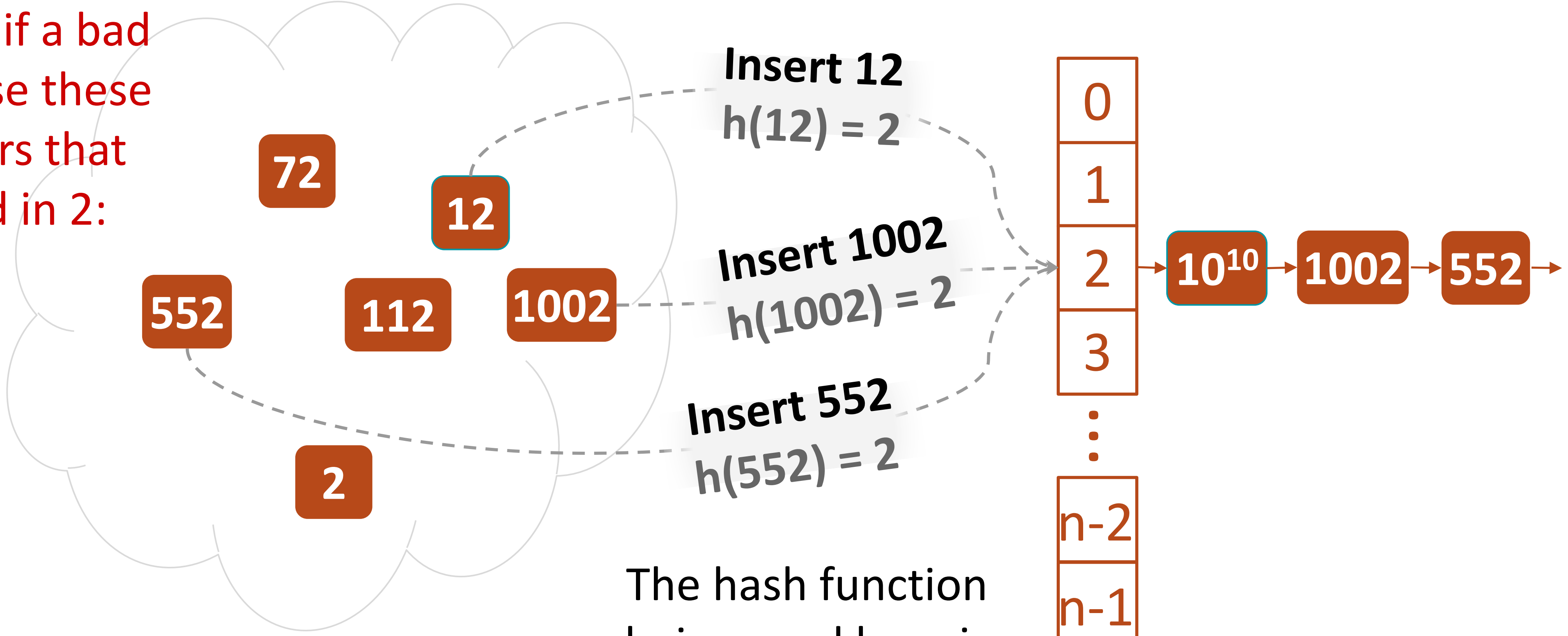


The hash function being used here is **h(x) = last digit of x**

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**

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**

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).

- The univer
- They get h
- At least 1
- **M** is wayy

T **d**

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 Functions and Randomness

What it means to weaken the adversary & ways to do it

Intuition

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

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?

Intuition

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

What we want

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 **$\{u_1, u_2, \ldots, u_n\}$** to hash,

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

Let's see an example of a set of hash functions H that achieves this goal!

H = Exhaustive Set of All Hash Functions

WHAT WE WANT:

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 uniformly random **h** in **H** and after an adversary chooses **n** items $\{u_1, u_2, \ldots, u_n\}$ to hash,

for any item u_i,

the **expected** # of items in u_i's bucket is **O(1)**

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

H = Exhaustive Set of All Hash Functions

WHAT WE WANT:

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

for any item **u_i**,

the **expected** # of items in **u_i**'s bucket is **O(1)**

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

Here is an example where **U** = {"a", "b", "c"} so **M = 3.** Also, we have **n = 2.**

	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8
"a"	0	0	0	0	1	1	1	1
"b"	0	0	1	1	0	0	1	1
"c"	0	1	0	1	0	1	0	1

The 0's and 1's represent the binary buckets i.e. h_8 will hash "b" to bucket 1.

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] =$$

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

This probability is taken over the random choice of hash function!

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

This probability is taken over the random choice of hash function!

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

This probability is taken over the random choice of hash function!

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

This probability is taken over the random choice of hash function!

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} \frac{1}{n}$$

How do we know that **$P[h(u_i) = h(u_j)] = 1/n$** ?

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

This probability is taken over the random choice of hash function!

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

How do we know that **$P[h(u_i) = h(u_j)] = 1/n$?**

$$= 1 + \sum_{j \neq i} \frac{1}{n}$$

$$= 1 + \frac{n-1}{n}$$

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

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

This probability is taken over the random choice of hash function!

How do we know that **$P[h(u_i) = h(u_j)] = 1/n$** ?

O(1)

This is what we wanted!

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n

Good News:

H achieves our goal!

If we choose a *uniformly random hash function from* Exhaustive Set of All Hash Functions, then INSERT/DELETE/SEARCH on any **n** elements will have **expected runtime of O(1)**.

H = Exhaustive Set of All Hash Functions

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n

of

Bad News:

How many bits does it take to store a uniformly random hash function?

A lot!

n

How many bits does it take to store a uniformly random hash function?

We'd use a lookup table: one entry per element of U, each storing which bucket to hash that element to.

(**M** elements) * (**log(n)** bits to write down a bucket #) = **M log n** bits

This is HUGE… (& enough to do direct addressing!)

How many bits does it take to store a uniformly random hash function?

We'd use a lookup table: one entry per element of U, each storing which bucket to hash that element to.

(**M** elements) * (**log(n)** bits to write down a bucket #) = **M log n** bits

This is HUGE… (& enough to do direct addressing!)

How do we fix this size issue?

Universal Hash Families

"Good" sets of hash functions that aren't as large!

What we wanted

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} \frac{1}{n}$$

$$= 1 + \frac{n-1}{n} \leq 2$$

The fact that **$P[h(u_i)=h(u_j)] = 1/n$** did all the work here

O(1)

This is what we wanted!

What we wanted

H = the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n.

H contains a total of n^M hash functions.

The exhaustive set of all hash functions achieved our goal but was way too big, so let's pick **h** from a ***smaller*** **hash family** where

$$\mathbf{P[h(u_i) = h(u_j)] \leq 1/n}$$

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**,

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

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

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)

Flashback 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 $u_i \neq u_j$,

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

$$\mathbb{E}[\text{\# of items in } u_i \text{ 's bucket}] = \sum_{j=1}^{n} P[h(u_i) = h(u_j)]$$

$$= P[h(u_i) = h(u_i)] + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$= 1 + \sum_{j \neq i} P[h(u_i) = h(u_j)]$$

$$\leq 1 + \sum_{j \neq i} \frac{1}{n}$$

$$= 1 + \frac{n-1}{n} \leq 2$$

This inequality is now what a universal hash family guarantees!

O(1)

This is what we wanted!

A Small Universal Hash Family?

Are there smaller ones universal hash families?

A Non-Example

$\mathbf{H = \{h_0, h_1\}}$ where

$\mathbf{h_0}$ = MOST_SIGNIFICANT_DIGIT

$\mathbf{h_1}$ = LEAST_SIGNIFICANT_DIGIT

Why is this not a universal hash family?

A Non-Example

H = {h_0, h_1} where

h_0 = MOST_SIGNIFICANT_DIGIT

h_1 = LEAST_SIGNIFICANT_DIGIT

Why is this not a universal hash family?

$$P_{h \in H}\left[h(153) = h(173)\right] = 1 > \frac{1}{n}$$

There's a ½ probability of choosing **h_0**, and **h_0(153) = h_0(173) = bucket 1**

There's a ½ probability of choosing **h_1**, and **h_1(153) = h_1(173) = bucket 3**

Probability that a randomly chosen **h** from **H** collides 153 & 173 is 1!

An Example

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

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

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

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

An Example

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

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

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 **p ≥ M**

Define **$h_{a,b}(x)$ = ((ax + b) mod p) mod n**

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

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

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

To store $h_{a,b}$, 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)

This is so much better than O(M log n)!

in {1, ..., p - 1}. in {0, ..., p - 1}.

Hash Tables

Putting everything together, what’s the scheme?

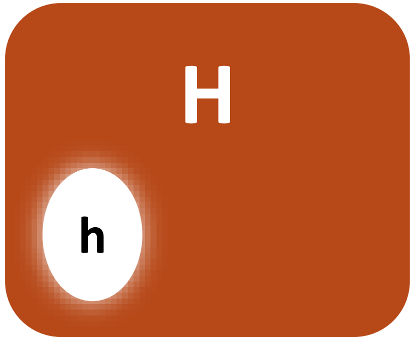
The Whole Scheme

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

H

The Whole Scheme

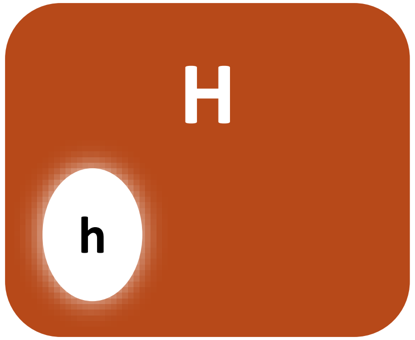
You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.



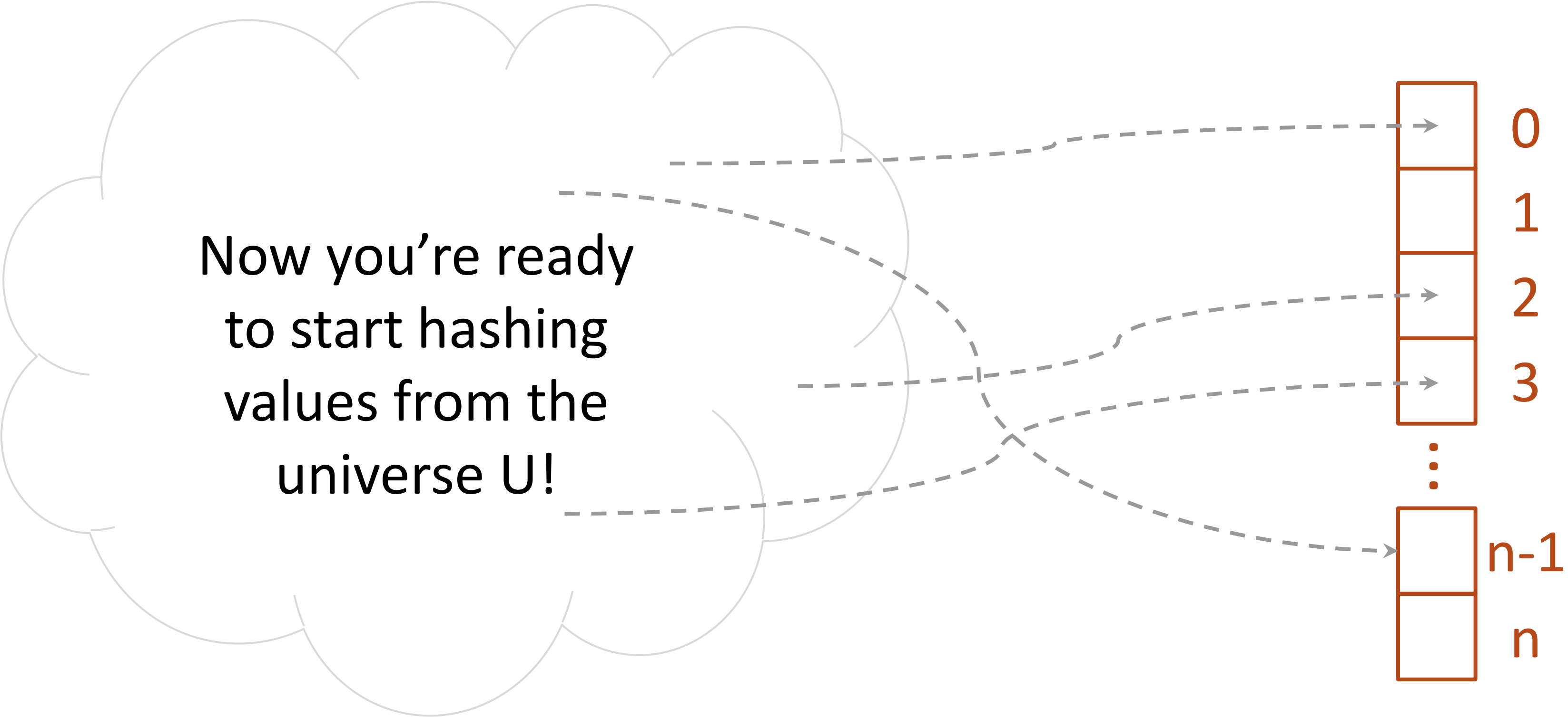
When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.

The Whole Scheme

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

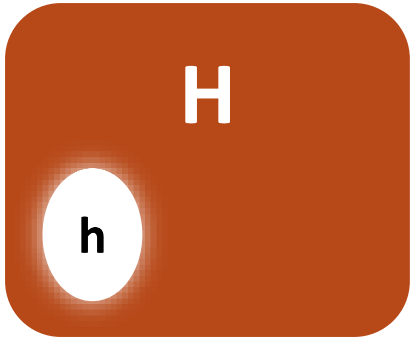


When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.



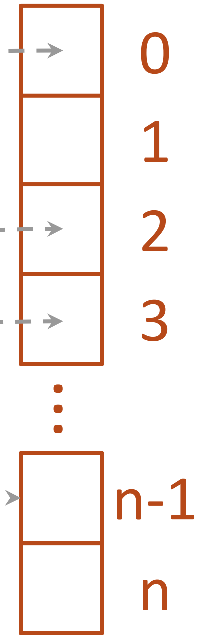
The Whole Scheme

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.

Now you're ready to start hashing values from the universe U!



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 (HOPEFULLY)
SEARCH	O(log(n))	O(n)	O(1)
DELETE	O(n)	O(n)	O(1)
INSERT	O(n)	O(1)	O(1)

*** Assuming we implement it cleverly with a "good" hash function**

Acknowledgement

- Stanford University

Thank You