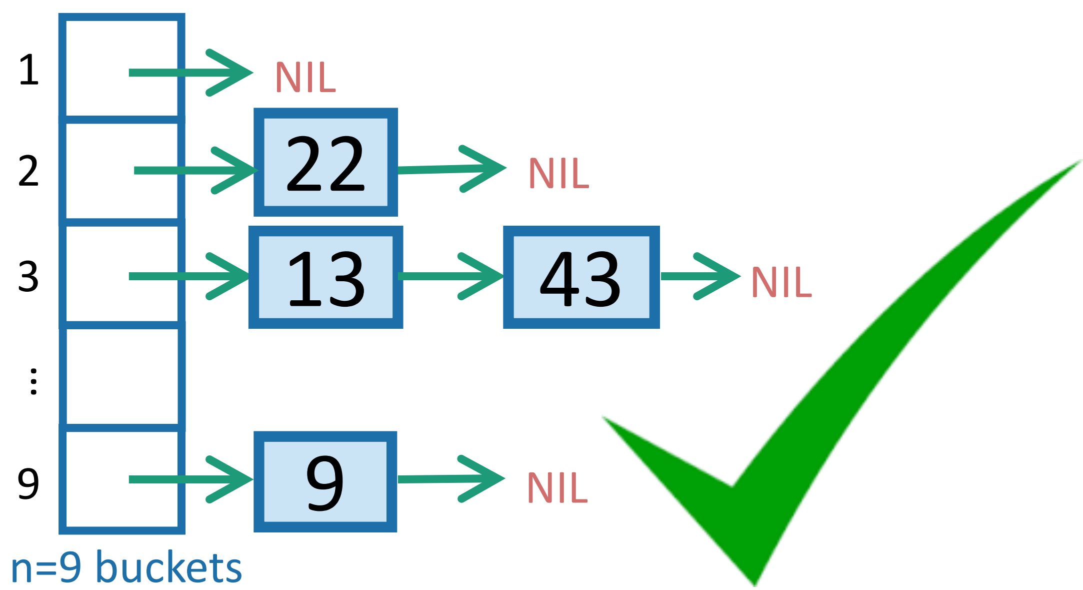
Lecture 8

Hashing

Announcements

- Midterm approaching: Thu, Feb 15 (6pm – 9pm)
- Midterm covers up to (and incl.) lecture 7 (up to and including homework 4). This week's lectures are not included.

Today: hashing



Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

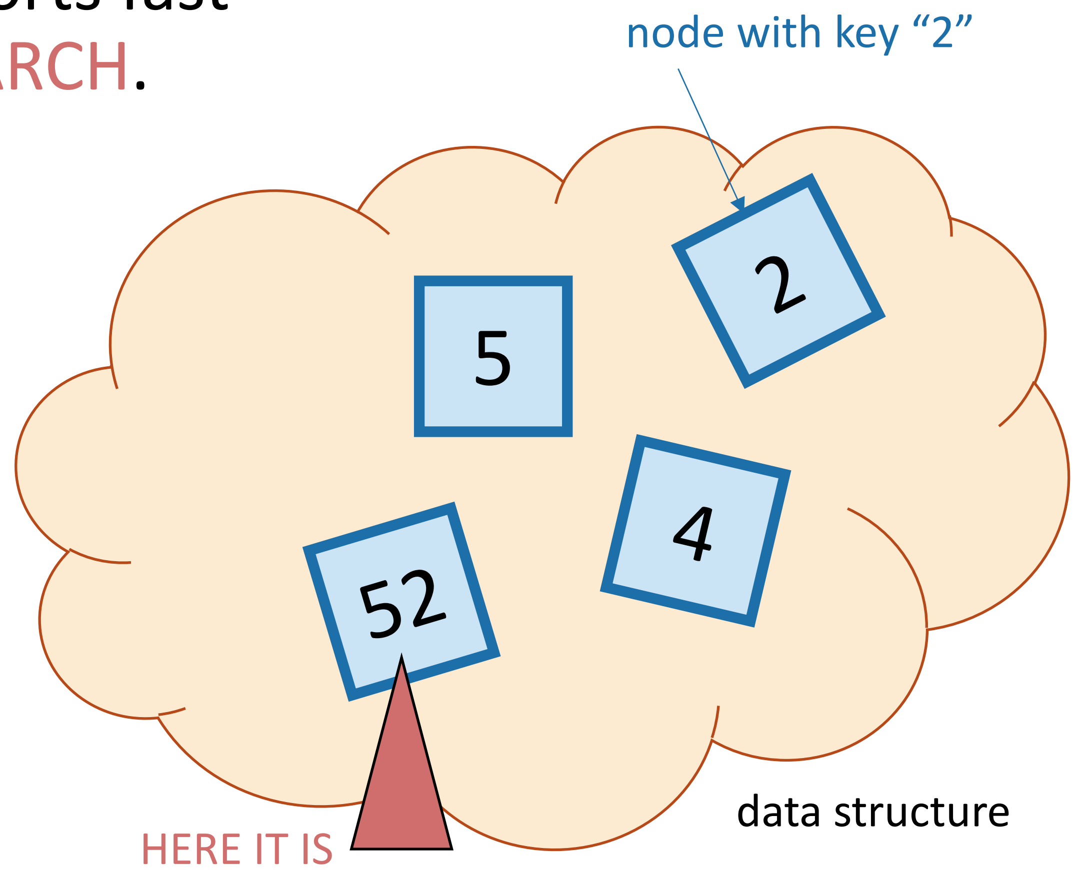
- **Hash families** are the magic behind hash tables.

- **Universal hash families** are even more magical.

Goal

- We want to store nodes with keys in a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT 5
- DELETE 4
- SEARCH 52



Last time

- Self balancing trees:
  - O(log(n)) deterministic INSERT/DELETE/SEARCH

#prettysweet

Today:

- Hash tables:
  - O(1) expected time INSERT/DELETE/SEARCH
- Worse worst-case performance, but often great in practice.

#evensweeterinpractice

eg, Python's `dict`, Java's `HashSet`/`HashMap`, C++'s `unordered_map`

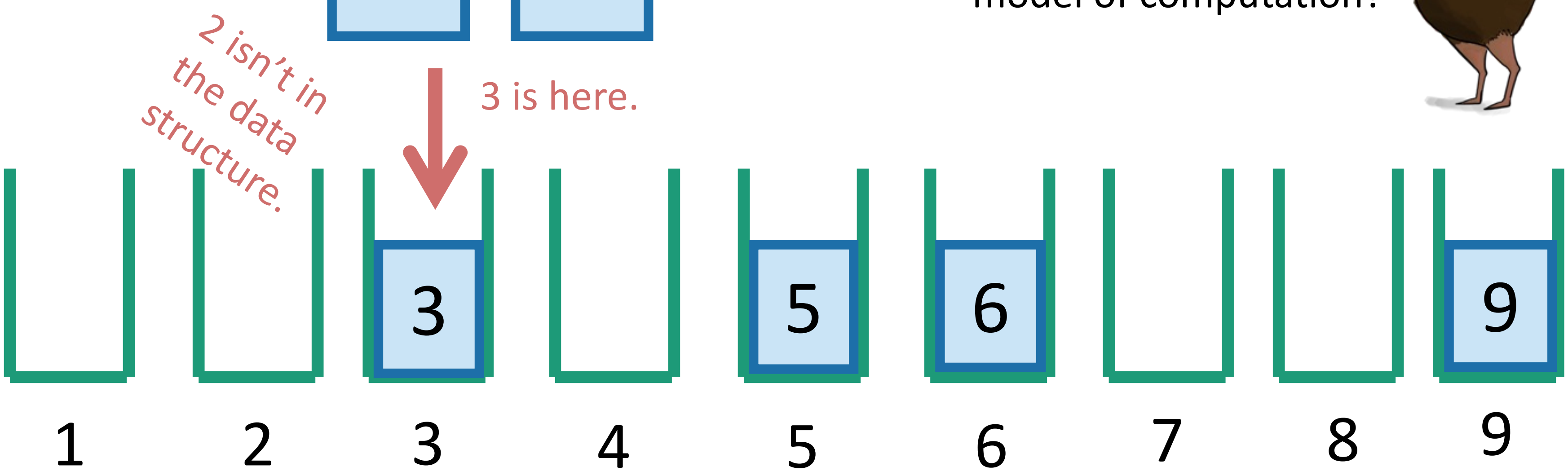
Hash tables are used for databases, caching, object representation, …

One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5
- DELETE: 6
- SEARCH: 3 2

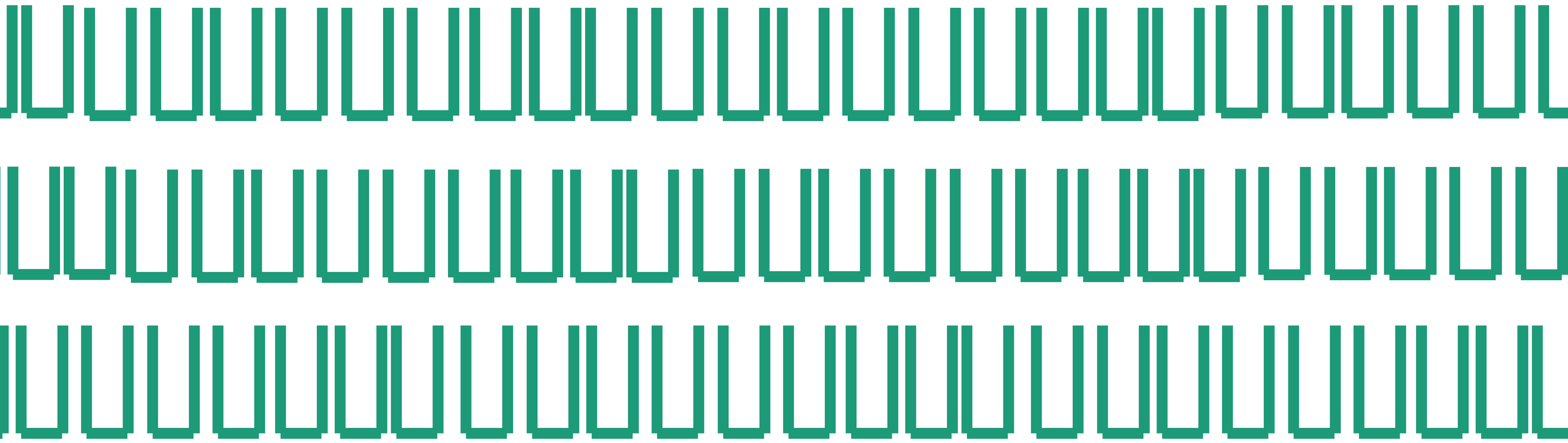
Are we delegating to hardware/memory? What are the assumptions behind our model of computation?



That should look familiar

The universe is really big!

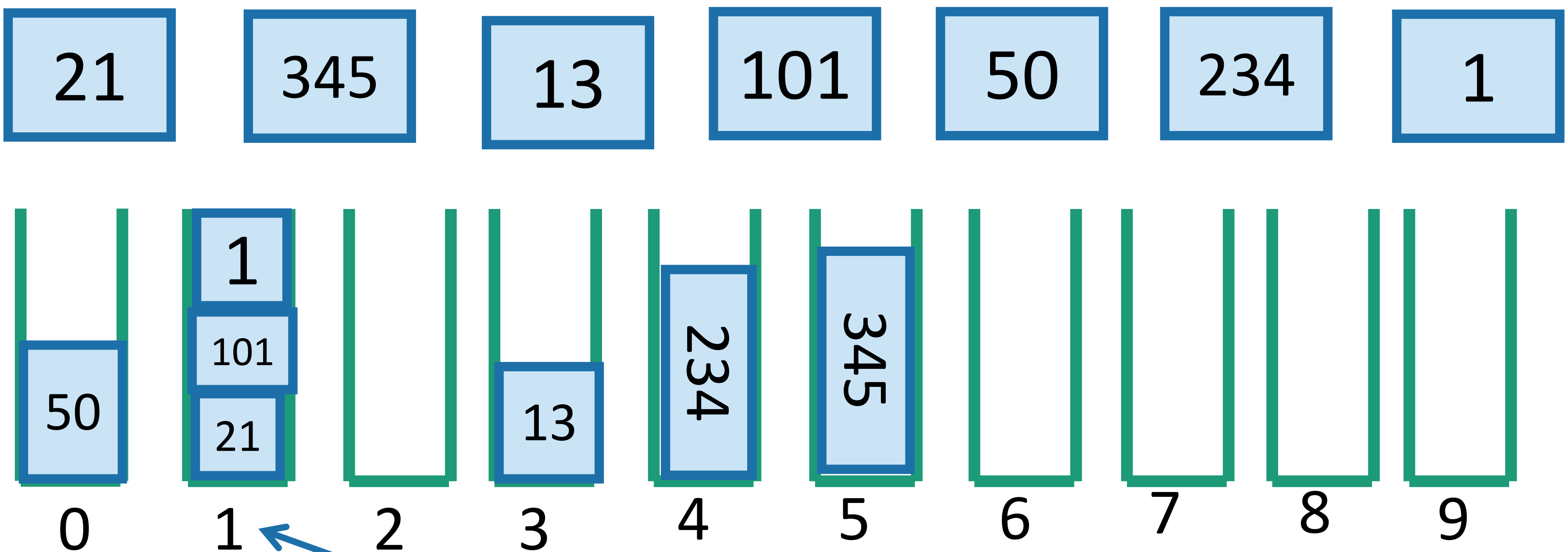
- Kind of like CountingSort from Lecture 6.
- Same problem: if the keys may come from a "universe" U = {1,2, …., 10000000000}, it takes a lot of space.



Solution?

Put things in buckets based on one digit

INSERT:

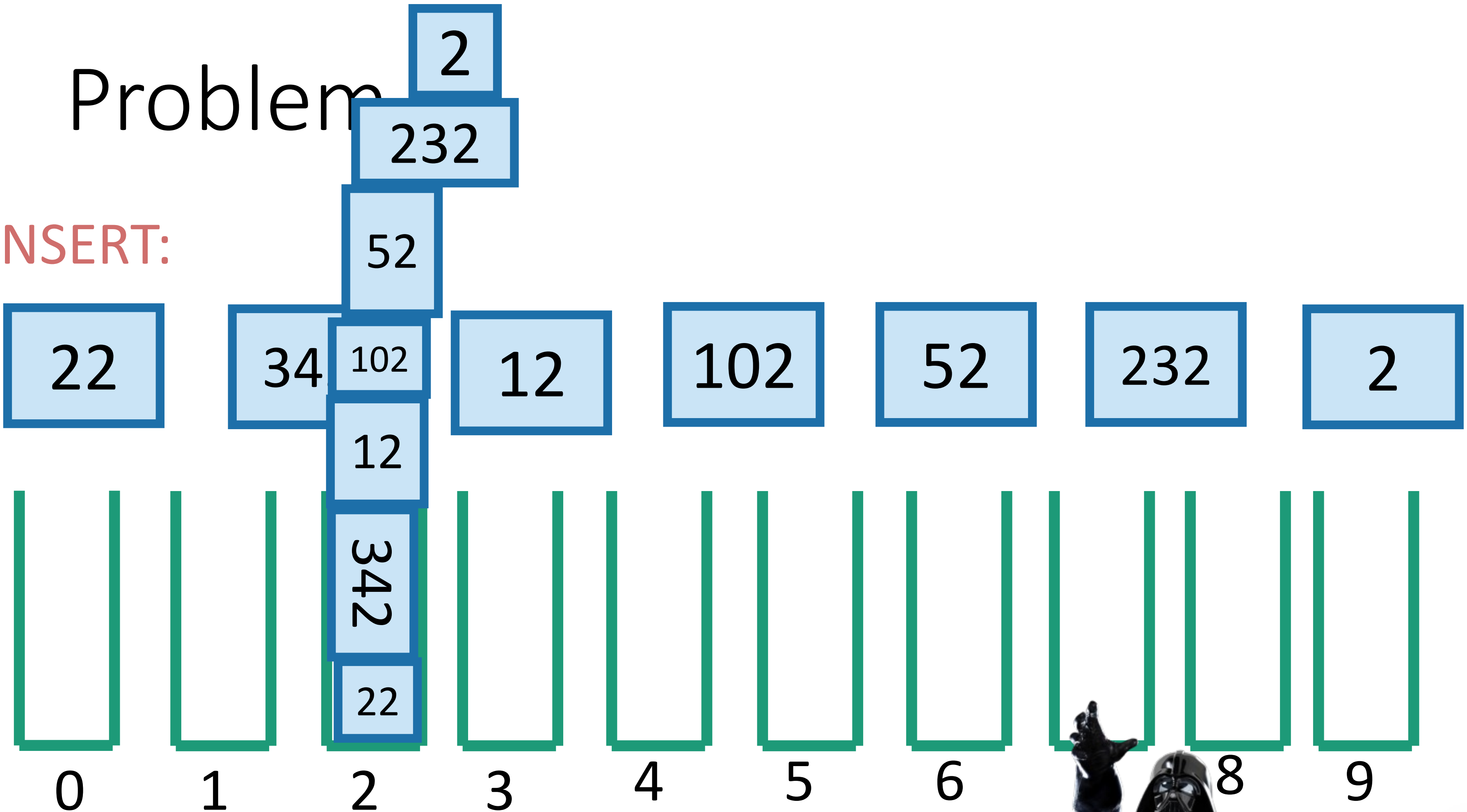


It’s in this bucket somewhere…
go through until we find it.

Now SEARCH 21

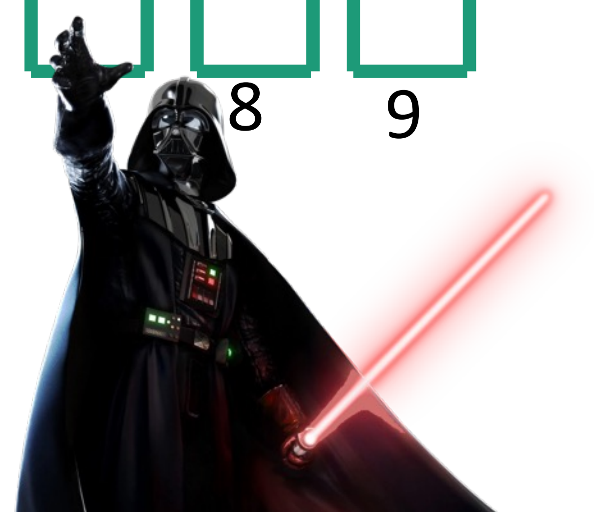
Problem

INSERT:



Now SEARCH 22

....this hasn't made our lives easier...

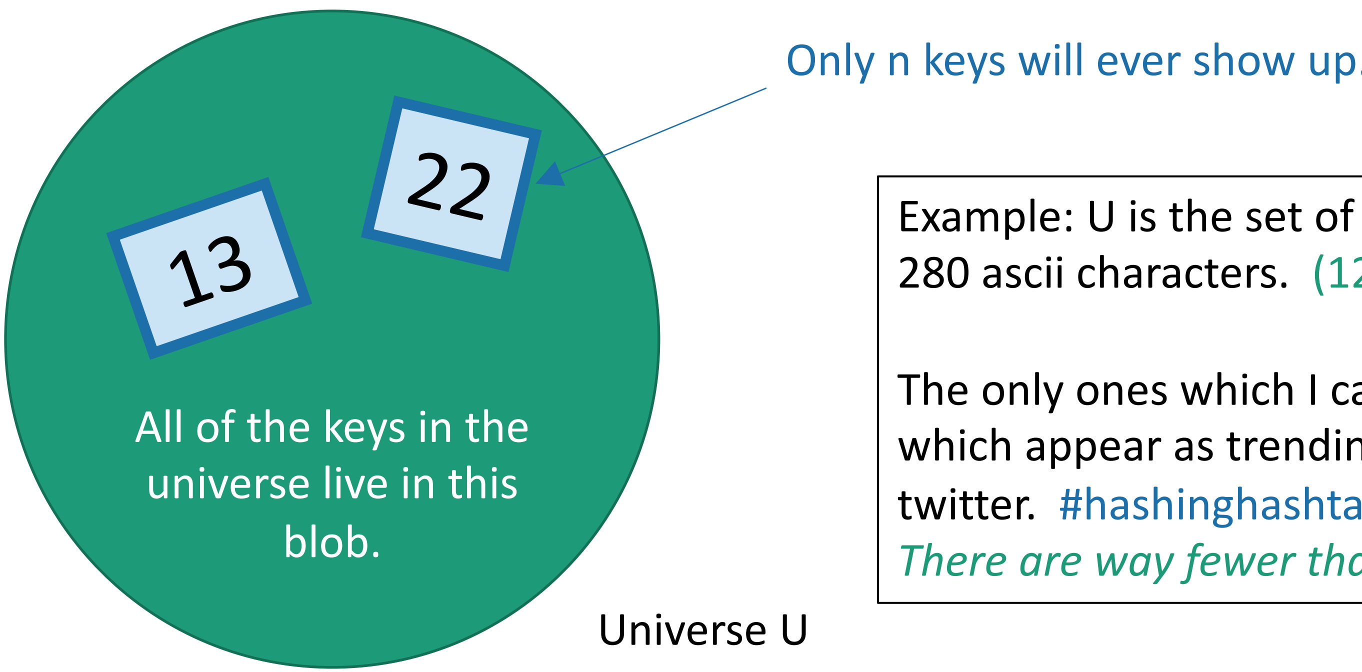


Hash tables

- That was an example of a hash table.
  - not a very good one, though.
- We will be **more clever** (and less deterministic) about our bucketing.
- This will result in fast (expected time) INSERT/DELETE/SEARCH.

But first! Terminology.

- U is a *universe* of size M.
  - M is really big.
- But only a few (at most n) elements of U are ever going to show up.
  - M is waaaayyyyyyy bigger than n.
- But we don't know which ones will show up in advance.



Example: U is the set of all strings of at most 280 ascii characters. (128^{280} of them).

The only ones which I care about are those which appear as trending hashtags on twitter. #hashinghashtags
There are way fewer than 128^{280} of these.

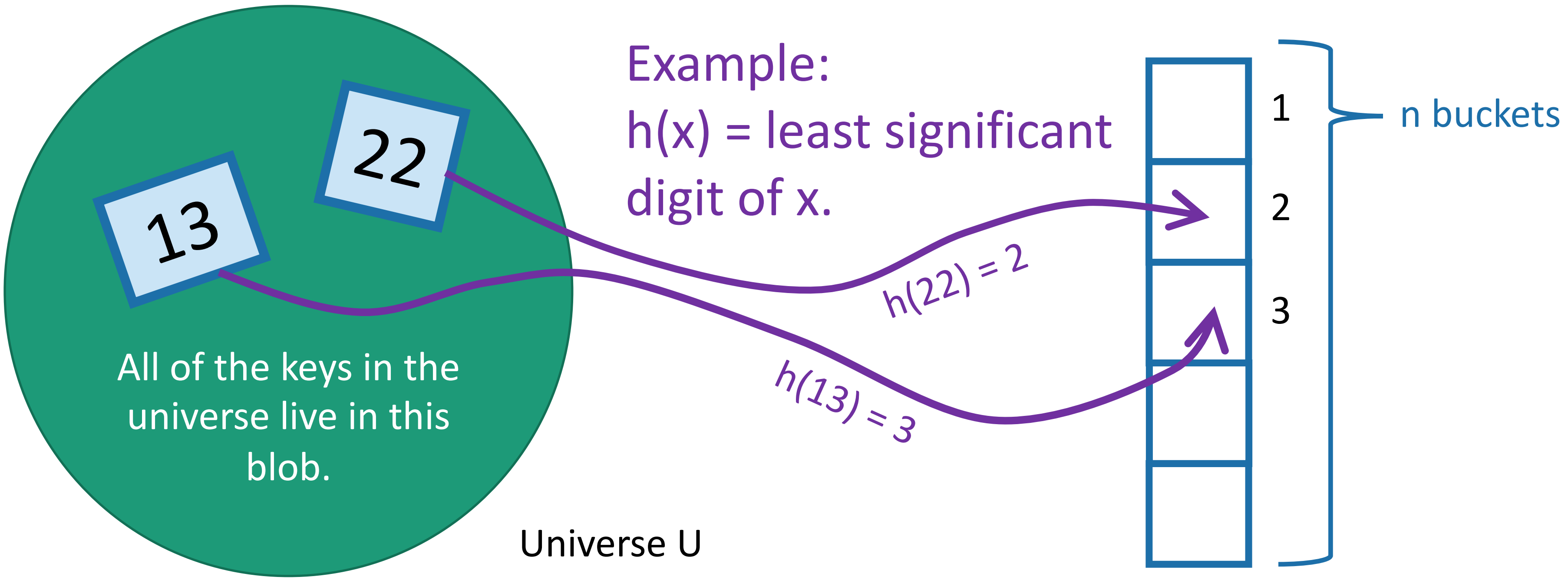
Hash Functions

- A *hash function* $h: U \to \{1, \ldots, n\}$ is a function that maps elements of U to buckets 1, ..., n.

For this lecture, we are assuming that the number of things that show up is the same as the number of buckets, both are n.

This doesn't have to be the case, although we do want:

#buckets = O(#things which show up)



Hash Tables (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- A hash function $h: U \to \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

For demonstration purposes only!
This is a terrible hash function! Don't use this!

INSERT:

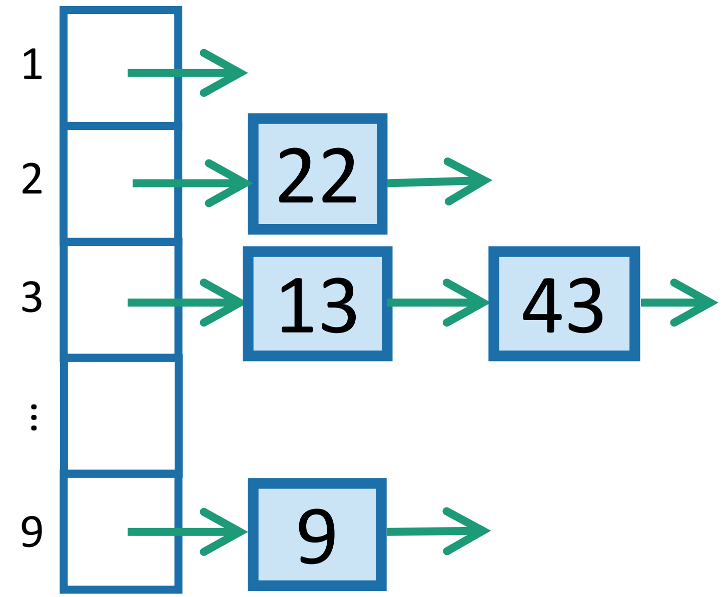
13 22 43 9

SEARCH 43:

Scan through all the elements in bucket h(43) = 3.

DELETE 43:

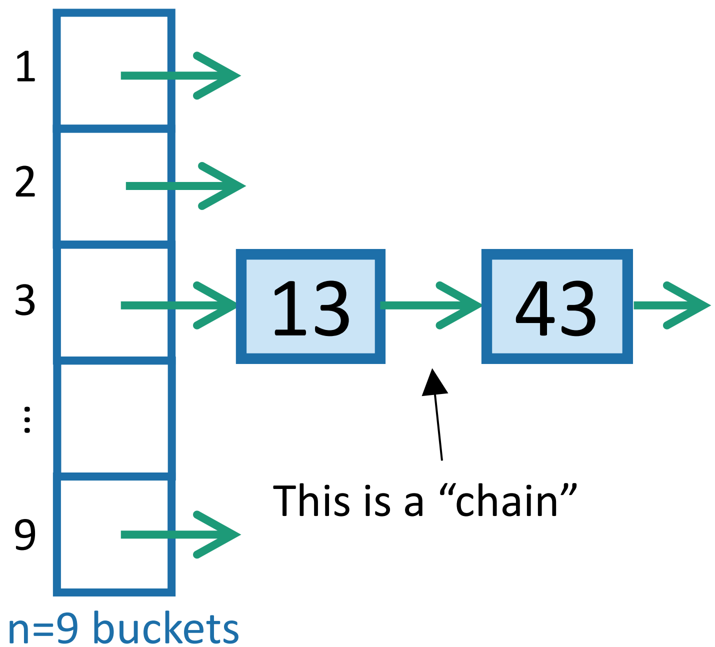
Search for 43 and remove it.

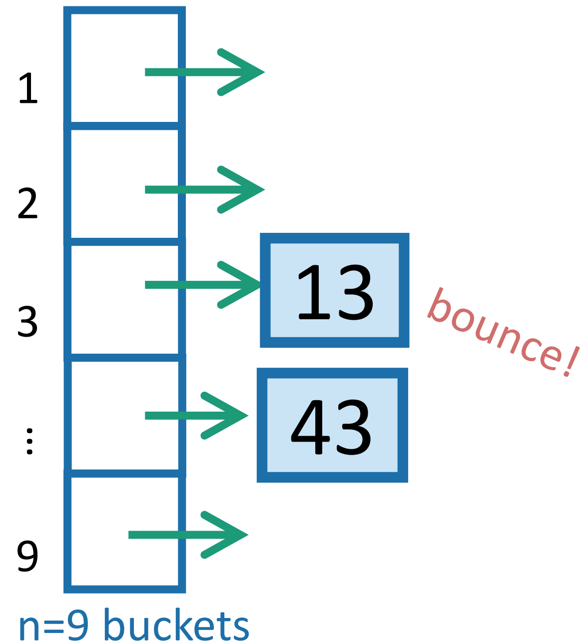


n buckets (say n=9)

Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- You don't need to know about it for this class.





\end{Aside}

Hash Tables (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- A hash function $h: U \to \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

For demonstration purposes only!
This is a terrible hash function! Don't use this!

INSERT:

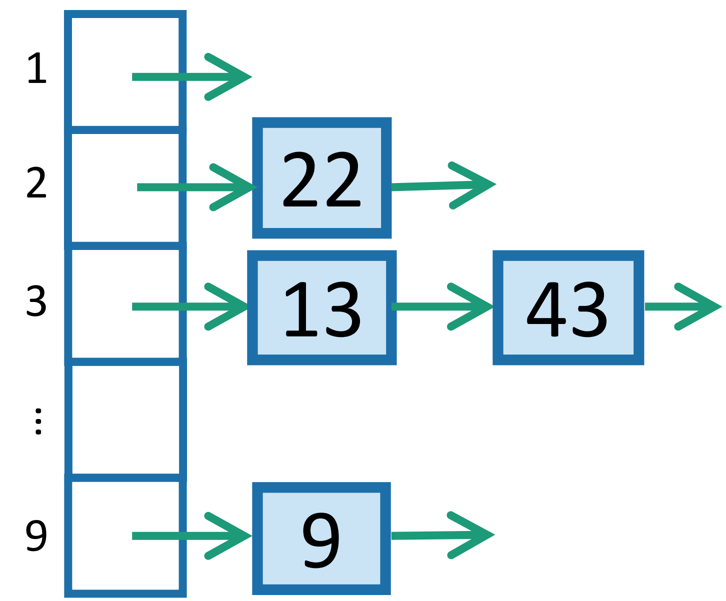
13 22 43 9

SEARCH 43:

Scan through all the elements in bucket h(43) = 3.

DELETE 43:

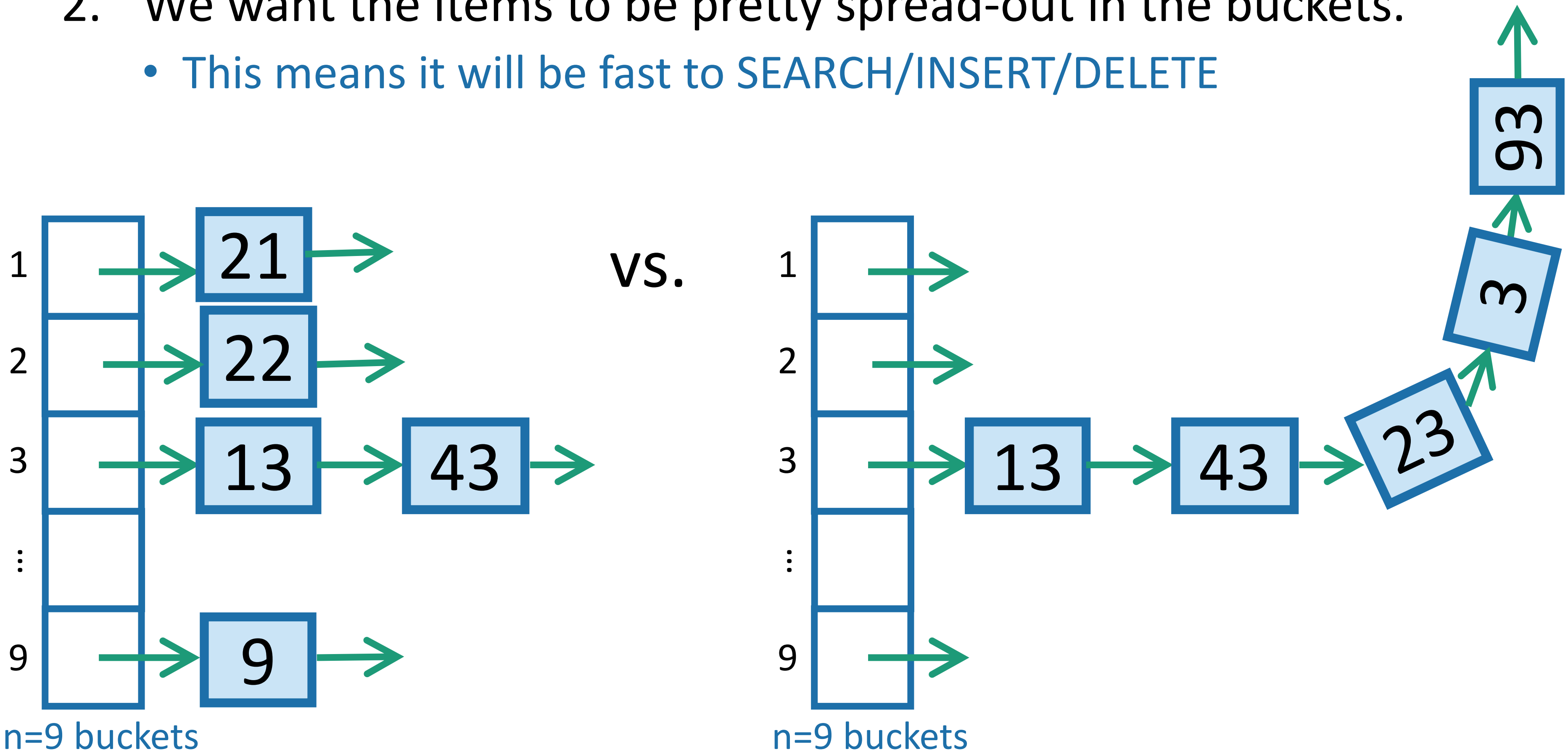
Search for 43 and remove it.



n buckets (say n=9)

What we want from a hash table

1. We want there to be not many buckets (say, n).
   - This means we don't use too much space
2. We want the items to be pretty spread-out in the buckets.
   - This means it will be fast to SEARCH/INSERT/DELETE

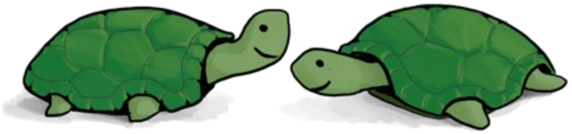


Worst-case analysis

- Goal: Design a function $h: U \rightarrow \{1, \ldots, n\}$ so that:
  - No matter what n items of U a bad guy chooses, the buckets will be balanced.
  - Here, balanced means O(1) entries per bucket.

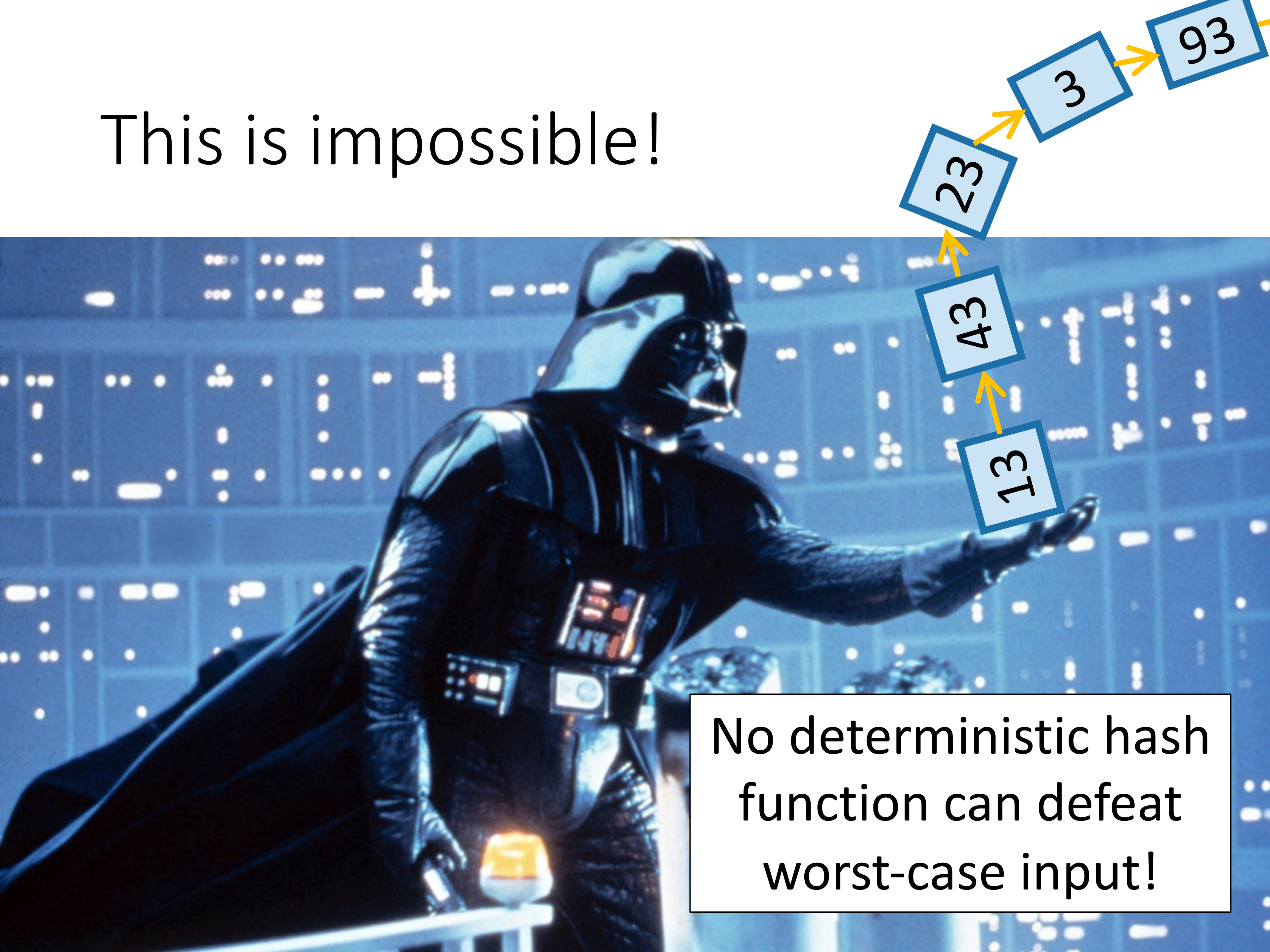
- If we had this, then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH

Can you come up with such a function?



Think-Share Terrapins

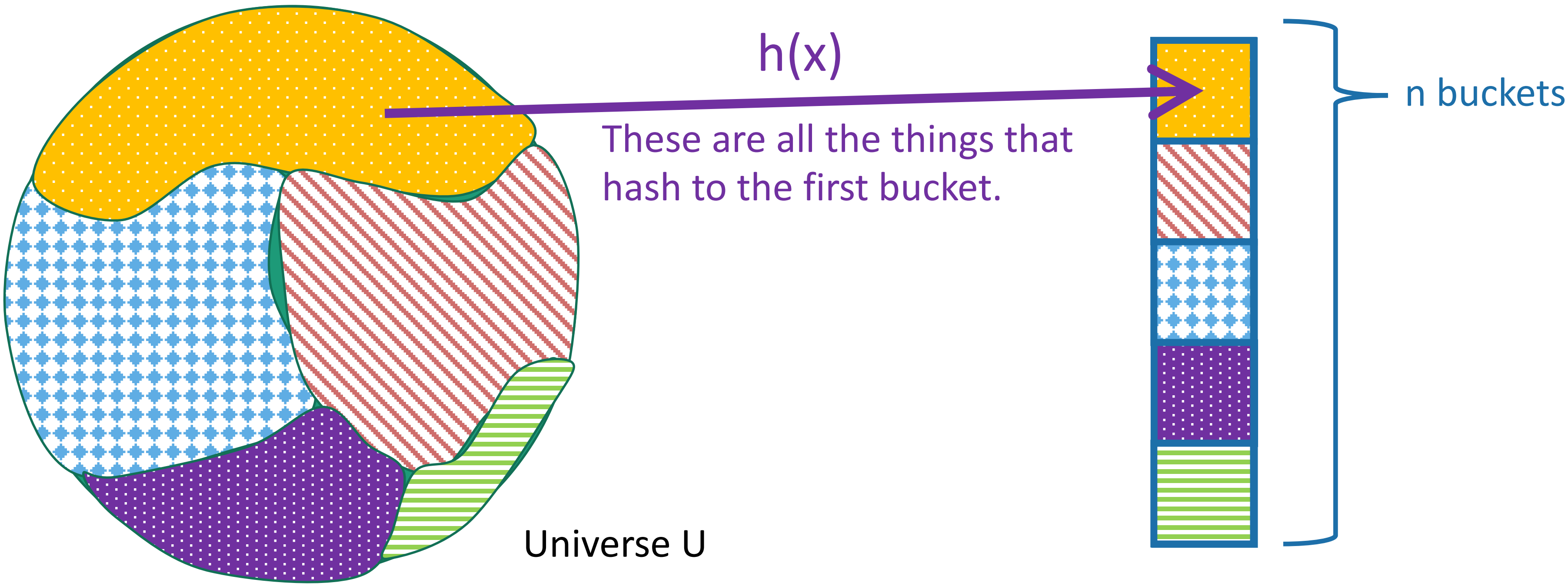
This is impossible!



No deterministic hash function can defeat worst-case input!

We really can't beat the bad guy here.

- The universe U has M items
- They get hashed into n buckets
- At least one bucket has at least M/n items hashed to it.
- M is waayyyy bigger than n, so M/n is bigger than n.
- **Bad guy chooses n of the items that landed in this very full bucket.**



Solution:
Randomness

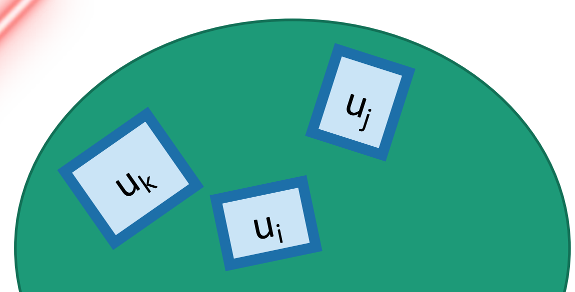
The game

What does **random** mean here? Uniformly random?

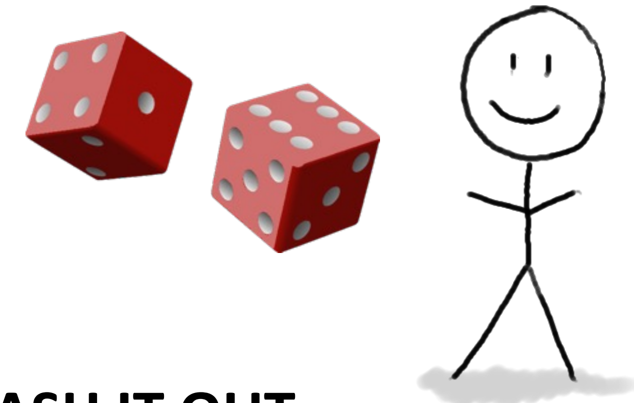
Plucky the pedantic penguin

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

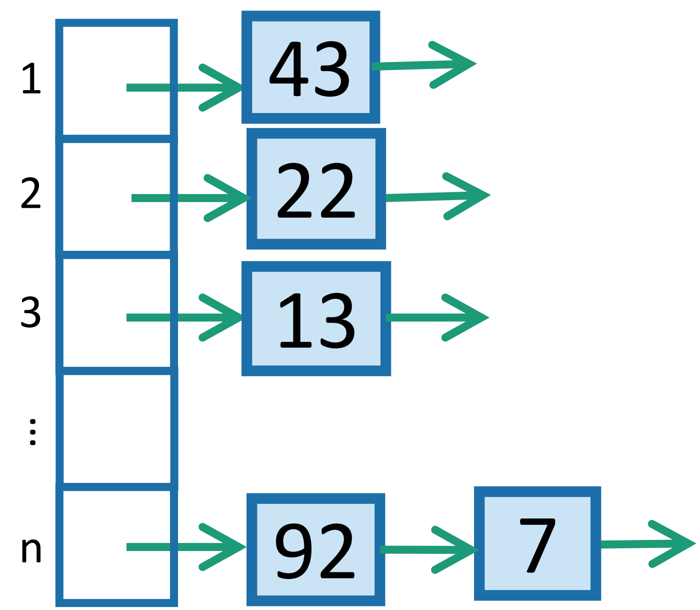
INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92



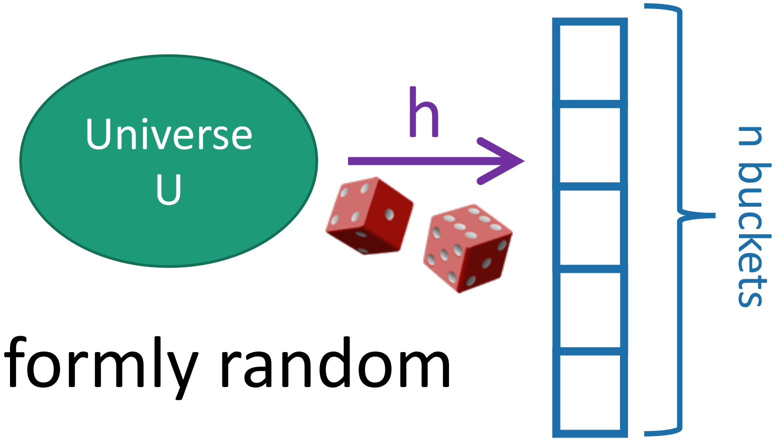
2. You, the algorithm, choose a **random** hash function $h: U \to \{1, \ldots, n\}$.



3. **HASH IT OUT** #hashpuns



Example of a random hash function



- Say that $h: U \to \{1, \ldots, n\}$ is a uniformly random function.
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(M)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(M-1).

Randomness helps

Intuitively: The bad guy can't foil a hash function that he doesn't yet know.

Lucky the Lackadaisical Lemur

Plucky the Pedantic Penguin

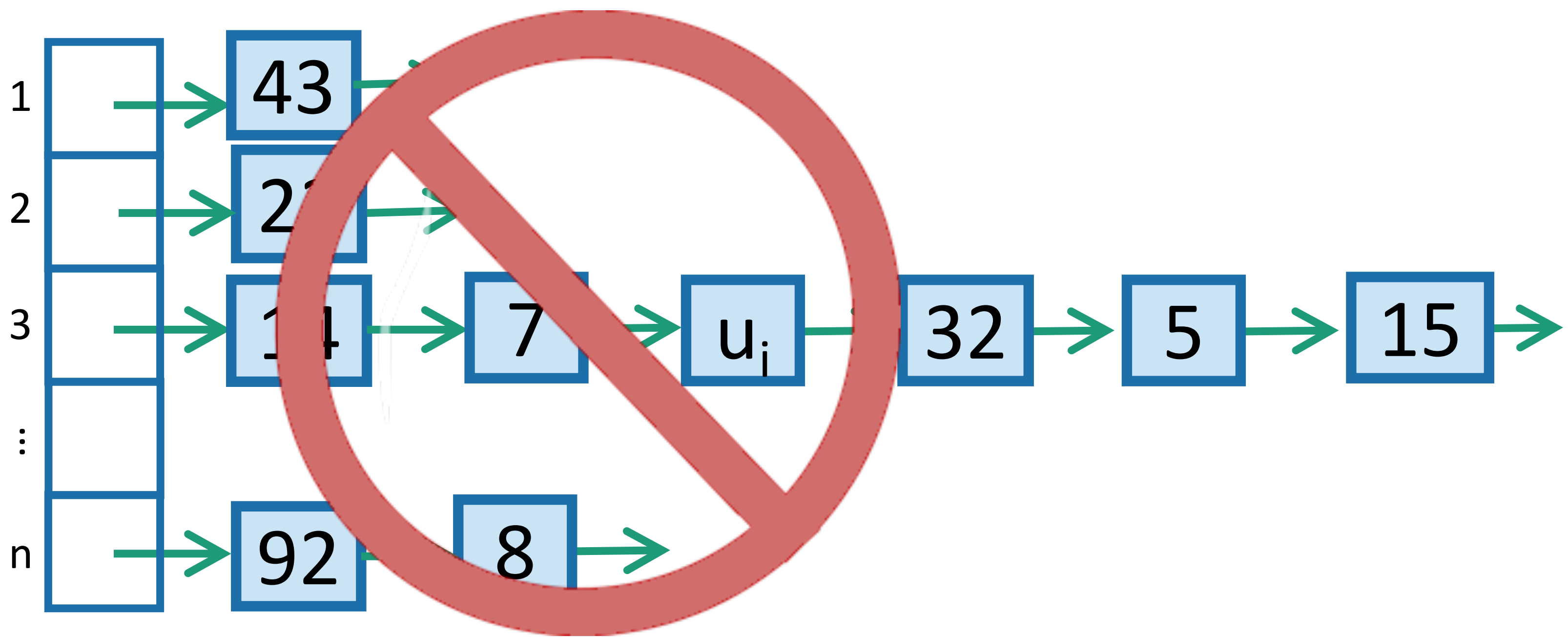
Why not? What if there's some strategy that foils a random function with high probability?

We'll need to do some analysis…

What do we want?

It's **bad** if lots of items land in u_i's bucket.

So we want **not that**.



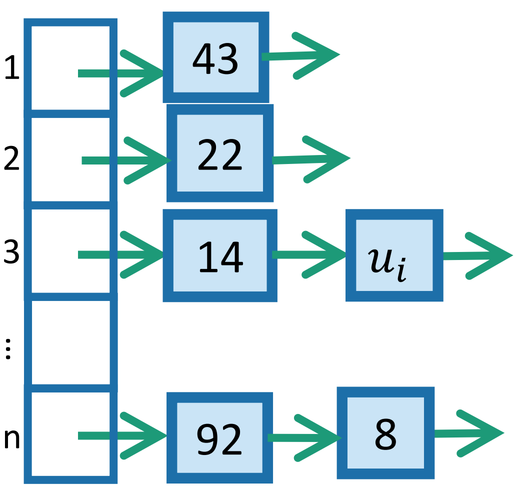
More precisely

We could replace "2" here with any constant; it would still be good. But "2" will be convenient.

- We want:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

    E[number of items in u_i's bucket] ≤ 2.
- If that were the case:
  - For each INSERT/DELETE/SEARCH operation involving u_i,

    E[time of operation] = O(1)



Note that the expected size of u_i's linked list is not the same as the expected {maximum size of linked lists}. What is the latter?

This is what we wanted at the beginning of lecture!

So we want:

- For all $i=1, \ldots, n$,

  E[number of items in u_i's bucket] ≤ 2.

Aside

- For all i=1, …, n,

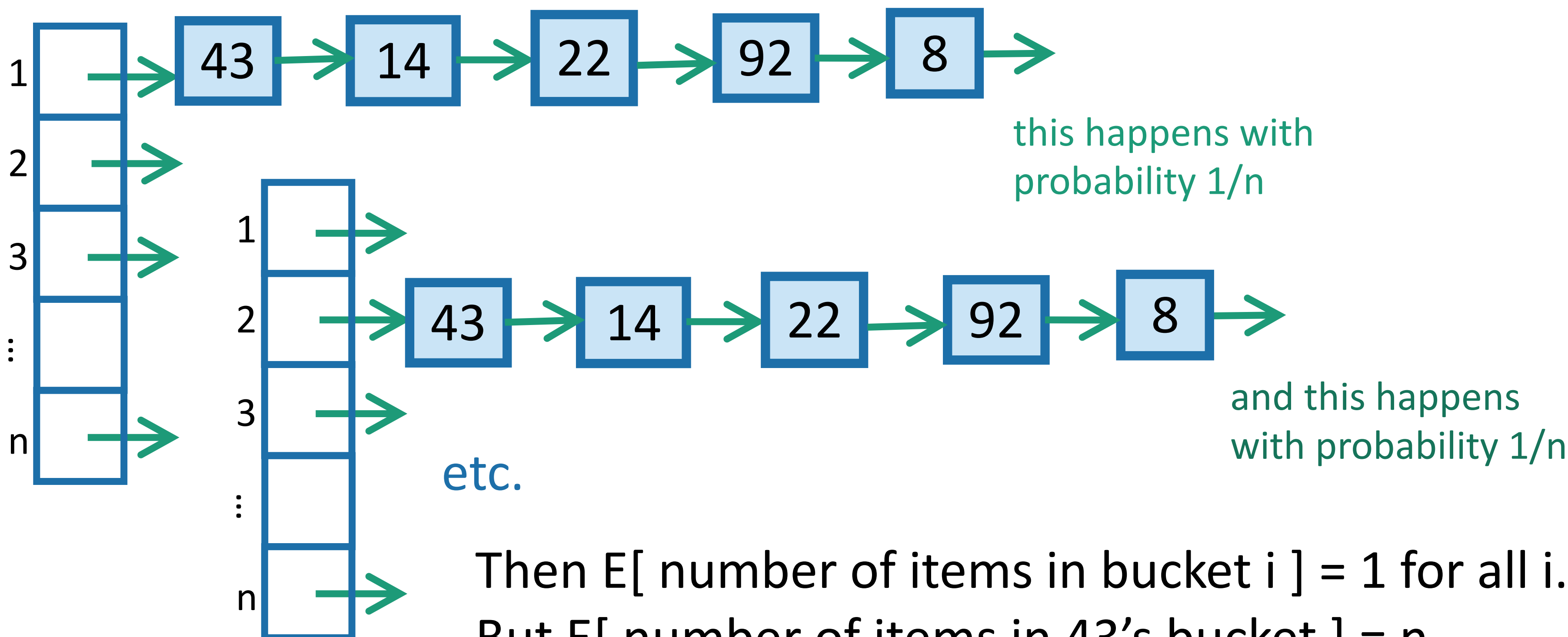
E[number of items in u_i 's bucket] $\leq$ 2.

vs

- For all i=1,…,n:

E[number of items in bucket i] $\leq$ 2

Suppose that:



Then E[number of items in bucket i] = 1 for all i.

But E[number of items in 43's bucket] = n

This distinction came up on your pre-lecture exercise!

- Solution to pre-lecture exercise:
  - E[number of items in bucket 1] = n/6
  - E[number of items that land in the same bucket as item 1] = n

So we want:

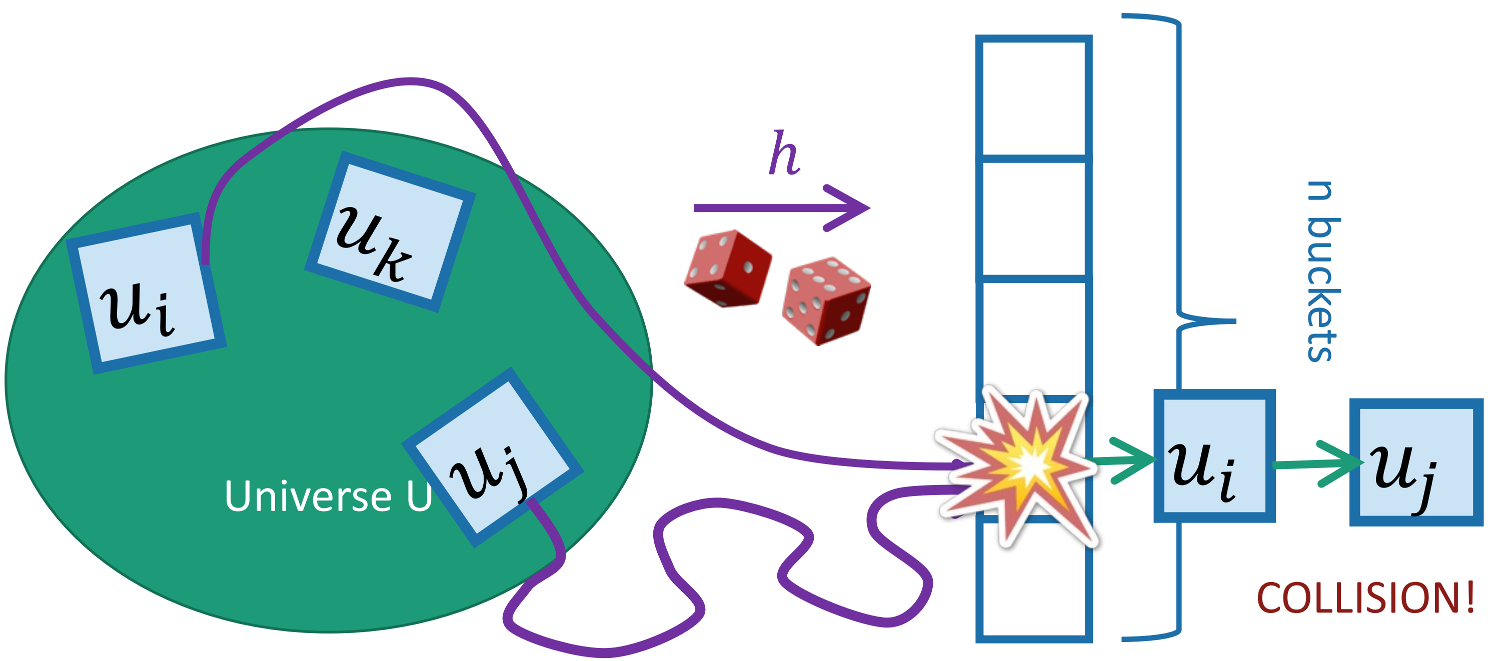
- For all i=1, …, n,

  E[number of items in u_i's bucket] ≤ 2.

h is uniformly random

Expected number of items in u_i's bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$ That's what we wanted!



A uniformly random hash function leads to balanced buckets

- We just showed:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

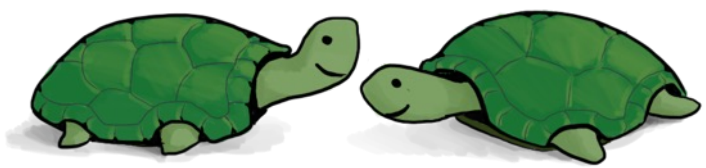
    E[number of items in u_i 's bucket] ≤ 2.
- Which implies:
  - No matter what sequence of operations and items the bad guy chooses,

    E[time of INSERT/DELETE/SEARCH] = O(1)
- So, our solution is:

Pick a uniformly random hash function?

What's wrong with this plan?

- Hint: How would you implement (and store) a uniformly random function $h: U \to \{1, \ldots, n\}$?



Think-Share Terrapins

- If h is a uniformly random function:
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(n)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(n-1).

A uniformly random hash function is not a good idea.

- In order to store/evaluate a uniformly random hash function, we'd use a lookup table:

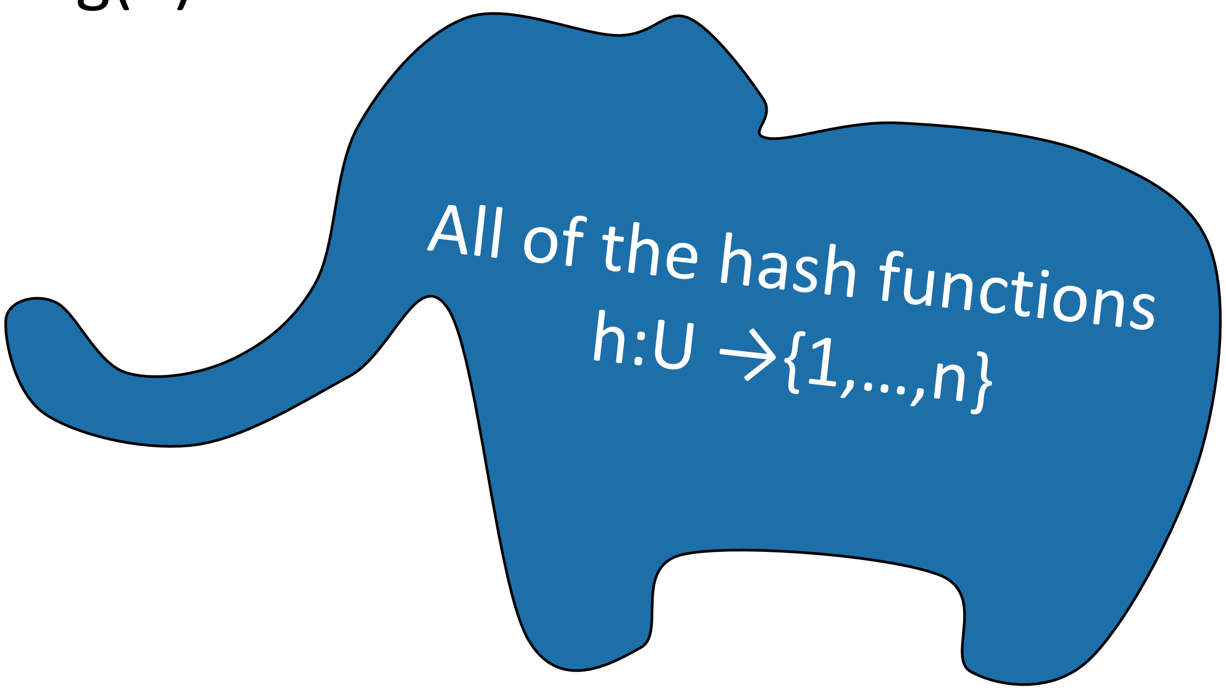
All of the M things in the universe

x	h(x)
AAAAAA	1
AAAAAB	5
AAAAAC	3
AAAAAD	3
…	
ZZZZZY	7
ZZZZZZ	3

- Each value of h(x) takes log(n) bits to store.
- Storing M such values requires Mlog(n) bits.
- In contrast, direct addressing (initializing a bucket for every item in the universe) requires only M bits.

Another way to say this

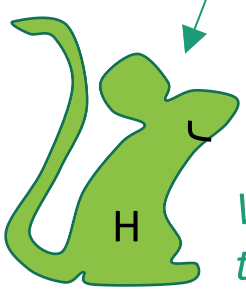
- There are lots of hash functions.
- There are n^M of them.
- Writing down a random one of them takes $\log(n^M)$ bits, which is $M \log(n)$.



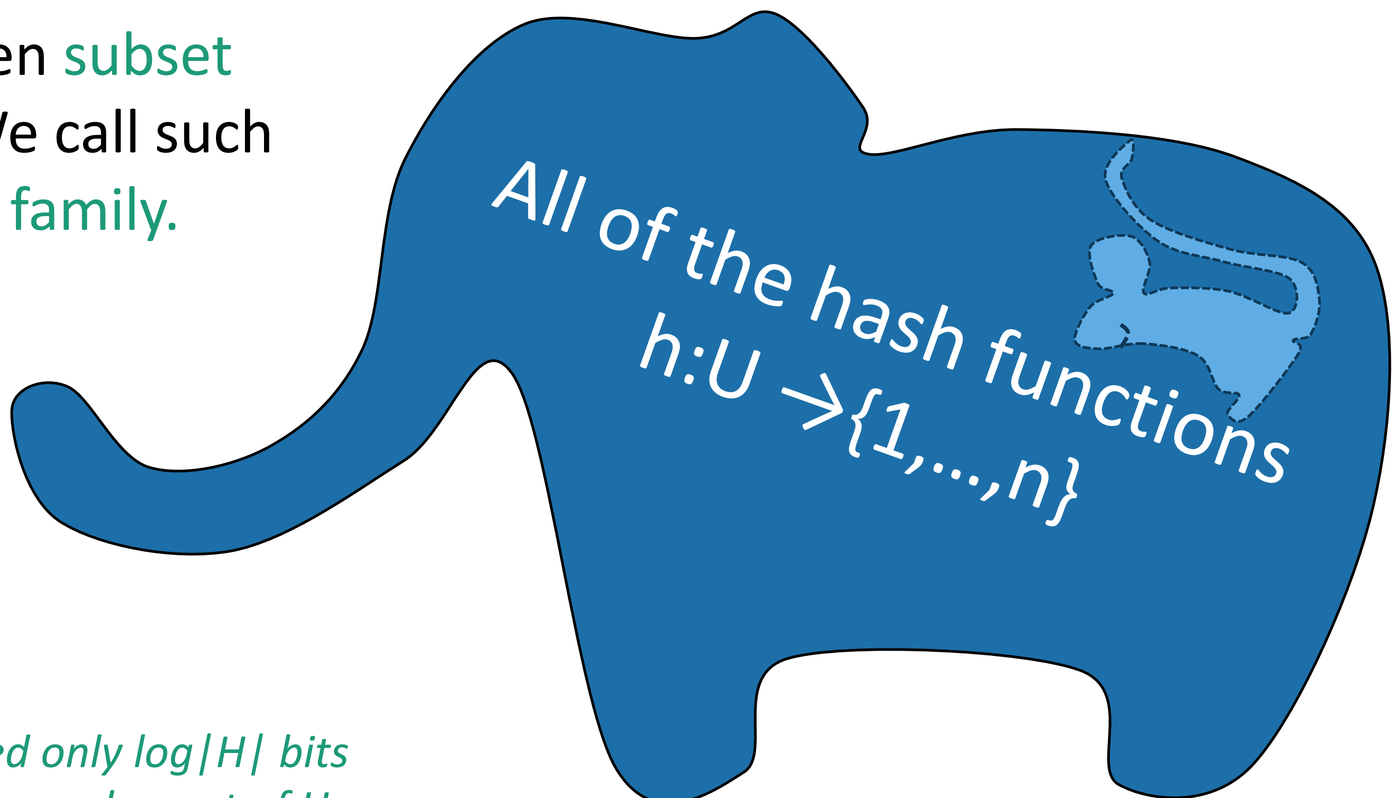
Solution

- Pick from a smaller set of functions.

A cleverly chosen subset of functions. We call such a subset a hash family.



We need only log|H| bits to store an element of H.



Outline

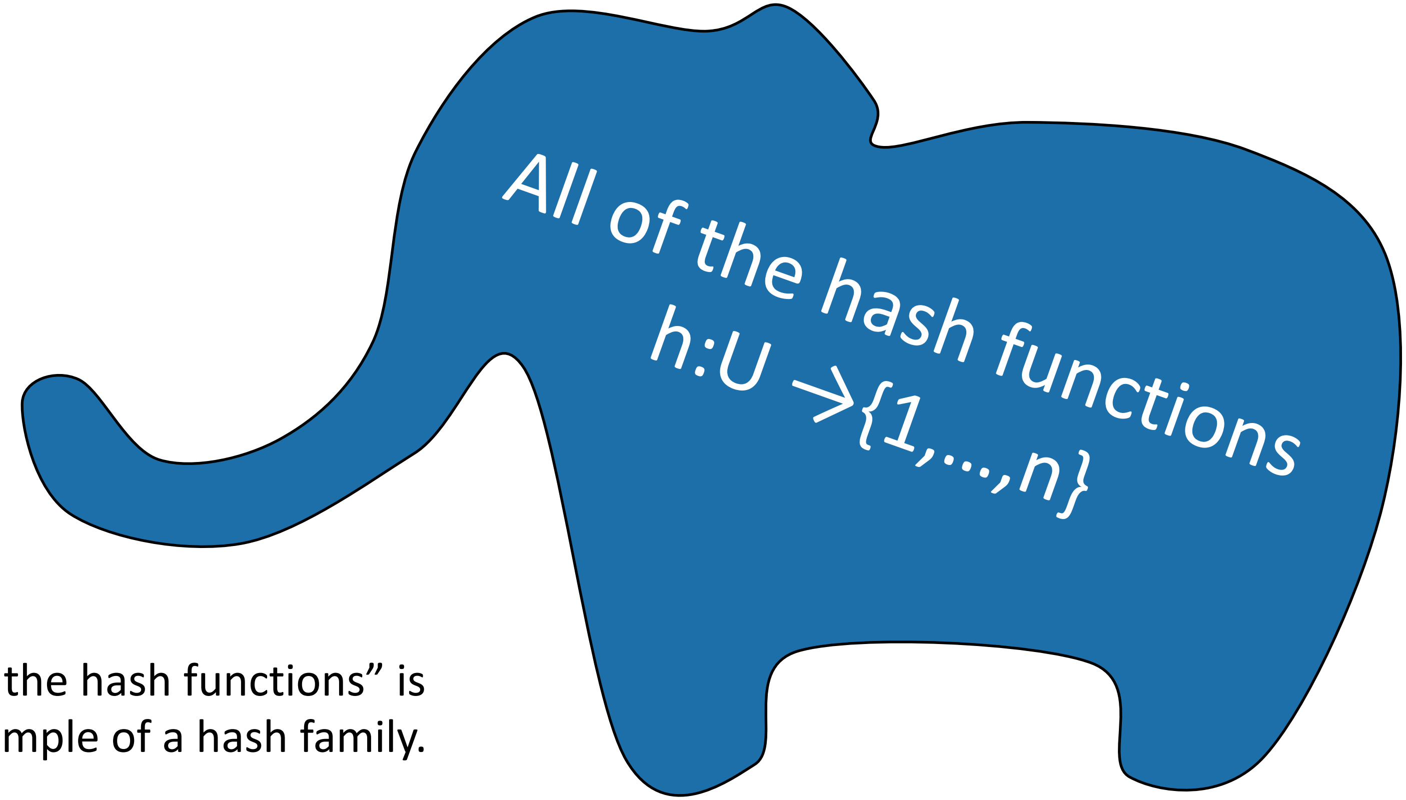
- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

- **Hash families** are the magic behind hash tables.

- **Universal hash families** are even more magic.

Hash families

- A hash family is a collection of hash functions.

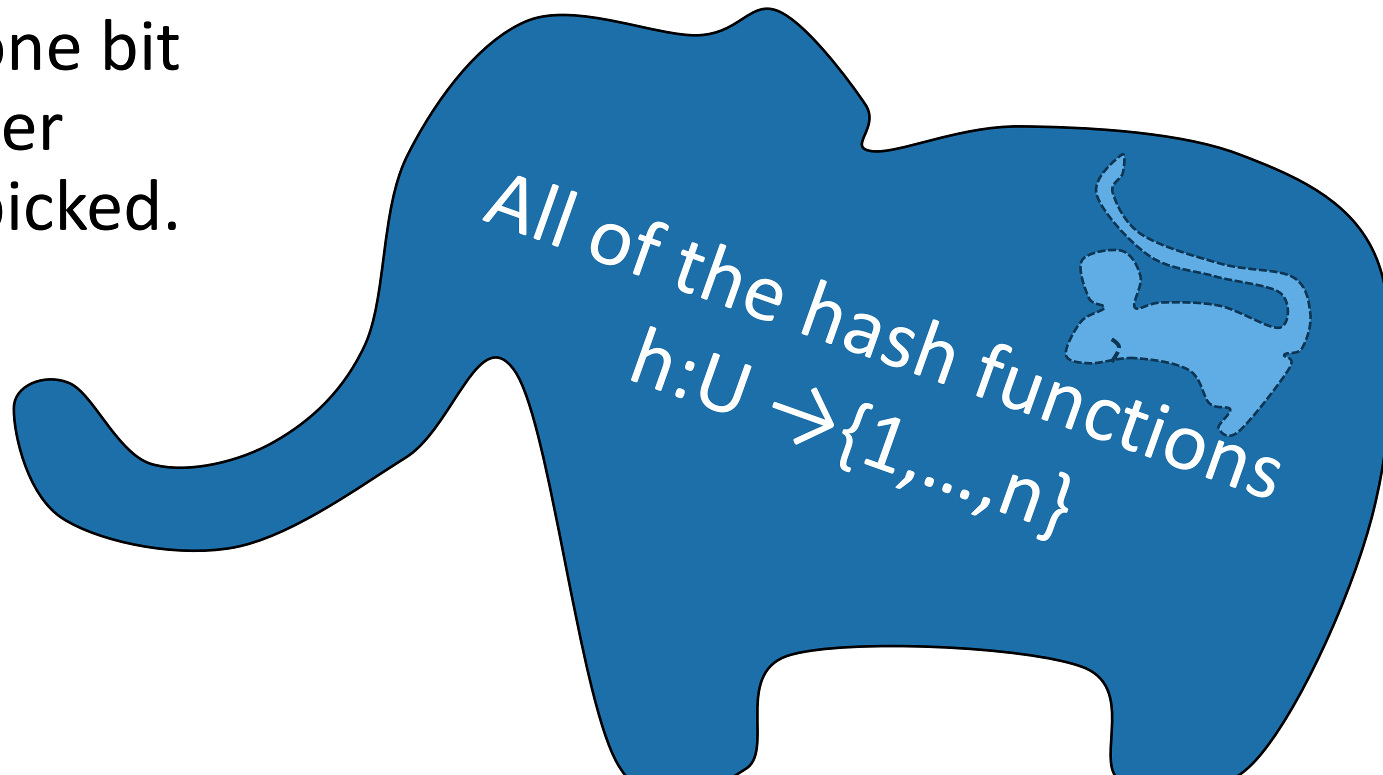


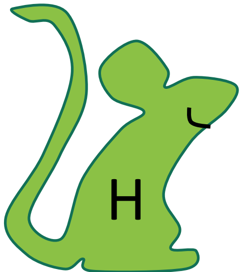
"All of the hash functions" is an example of a hash family.

Example:
a smaller hash family

This is still a terrible idea!
Don't use this example!
For pedagogical purposes only!

- H = { function which returns the least sig. digit,
  function which returns the most sig. digit }
- Pick h in H at random.
- Store just one bit to remember which we picked.





The game

h_0 = Most_significant_digit
h_1 = Least_significant_digit
H = $\{h_0, h_1\}$

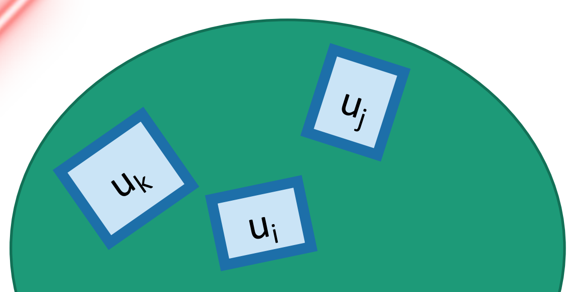
2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \dots, 9\}$. Choose it randomly from H.

I picked h_1

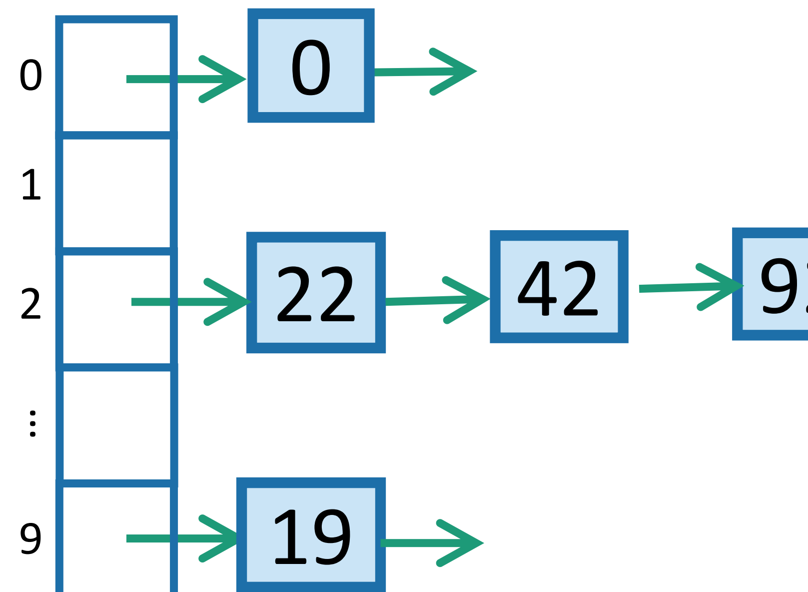
1. An adversary (who knows H) chooses any n items $u_1, u_2, \dots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

19	22	42	92	0

INSERT 19, INSERT 22, INSERT 42, INSERT 92, INSERT 0, SEARCH 42, DELETE 92, SEARCH 0, INSERT 92



3. **HASH IT OUT** #hashpuns



This is not a very good hash family

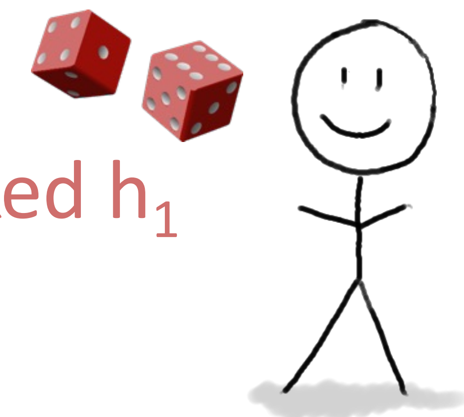
- H = { function which returns least sig. digit,

  function which returns most sig. digit }
- On the previous slide, the adversary could have been a lot more adversarial…

The game

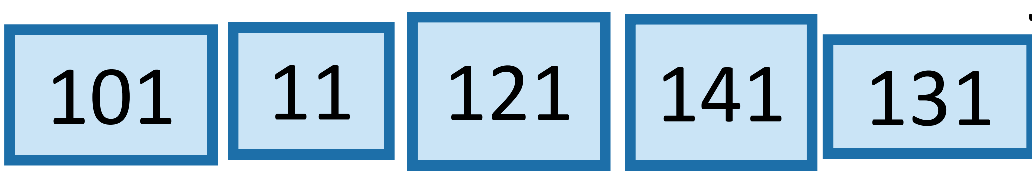
h_0 = Most_significant_digit
h_1 = Least_significant_digit
H = $\{h_0, h_1\}$

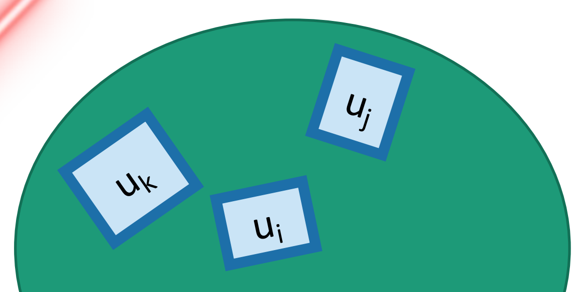
2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.



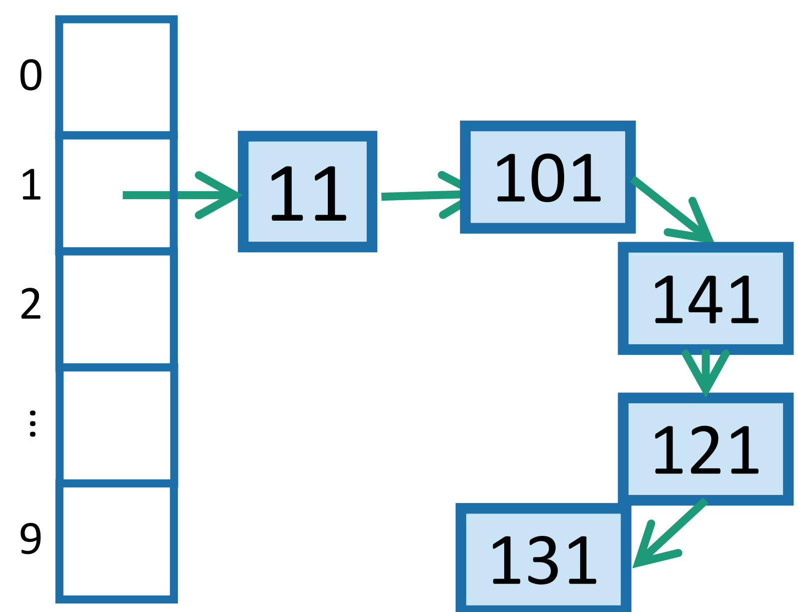
I picked h_1

1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.





3. **HASH IT OUT** #hashpuns



Outline

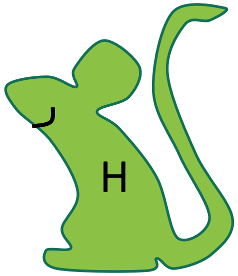
- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

- **Hash families** are the magic behind hash tables.

- **Universal hash families** are even more magic.

How to pick the hash family?

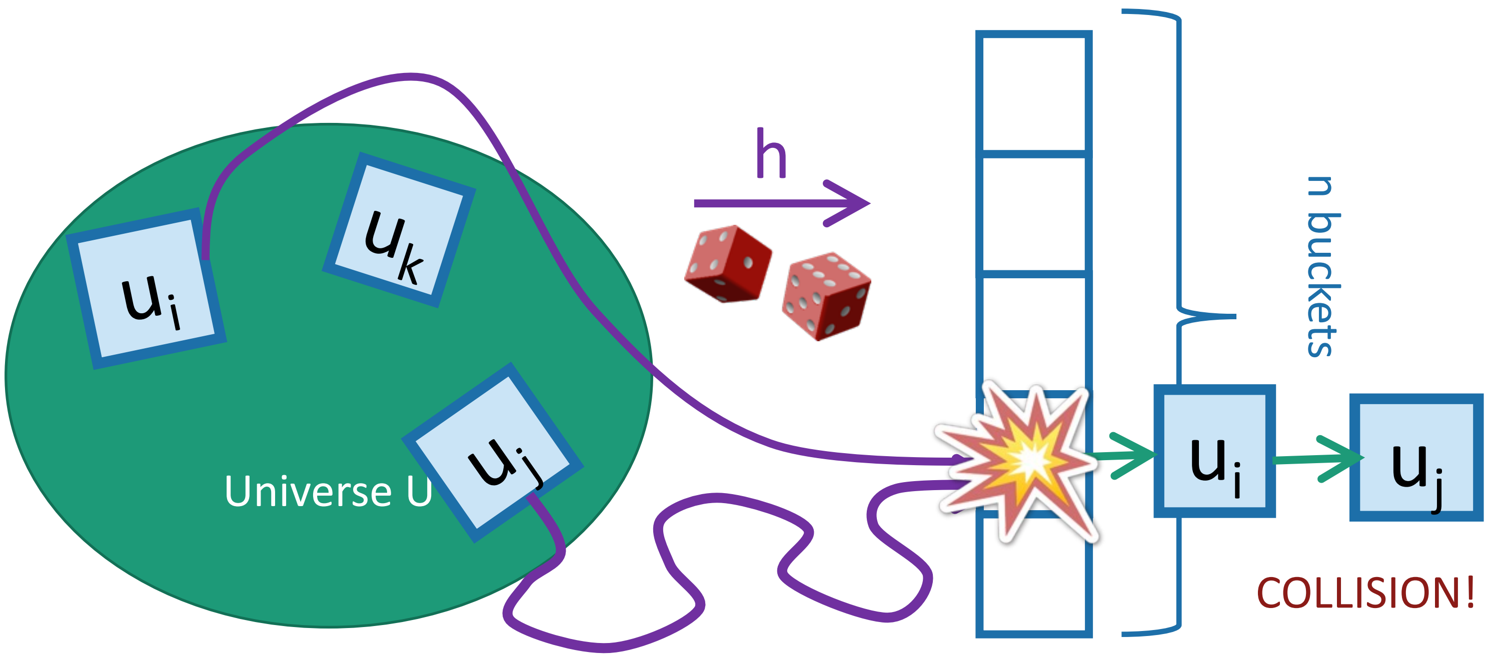
- Definitely not like in that example.
- Let’s go back to that computation from earlier….



Expected number of items in u_i's bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{\, h(u_i) = h(u_j) \}$
- $\quad = 1 + \sum_{j \neq i} P\{\, h(u_i) = h(u_j) \}$
- $\quad = 1 + \sum_{j \neq i} 1/n$
- $\quad = 1 + \frac{n-1}{n} \leq 2.$

All that we needed was that this is 1/n



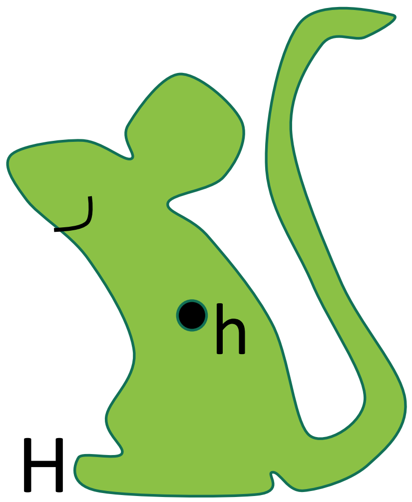
Strategy

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

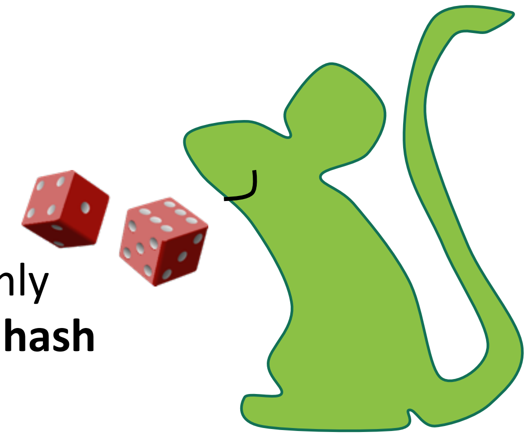
In English: fix any two elements of U. The probability that they collide under a random h in H is small.

- A hash family H that satisfies this is called a **universal hash family**.

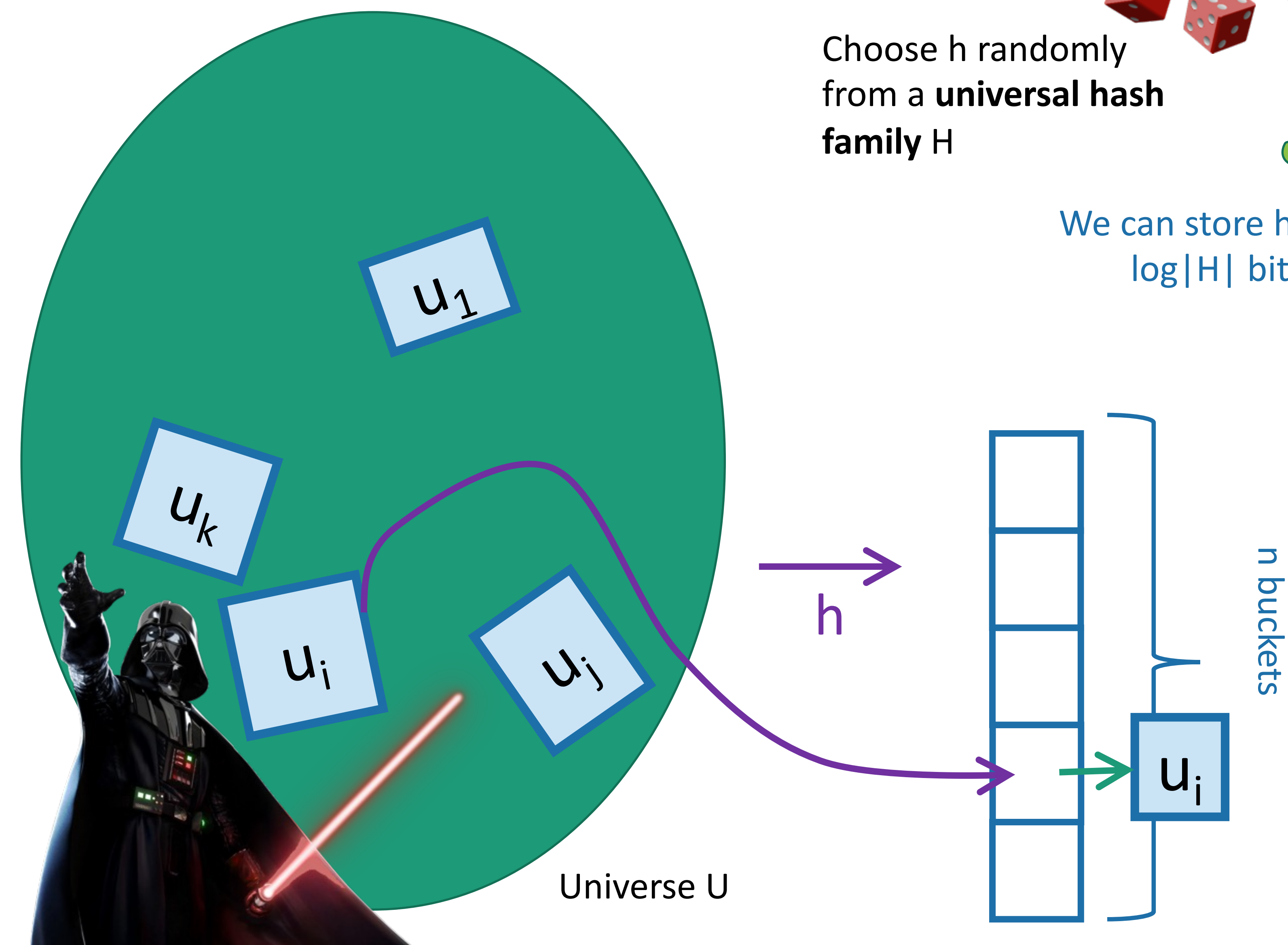


So the whole scheme will be

Choose h randomly from a **universal hash family** H



We can store h using log|H| bits.



Probably these buckets will be pretty balanced.

Universal hash family

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

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

Example

- H = the set of all functions $h: U \to \{1, \ldots, n\}$
  - We saw this earlier – it corresponds to picking a uniformly random hash function.
  - Unfortunately, this H is really really large.

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

Non-example

- h_0 = Most_significant_digit
- h_1 = Least_significant_digit
- H = $\{h_0, h_1\}$

NOT a universal hash family:

$$P_{h \in H}\{h(101) = h(111)\} = 1 > \frac{1}{10}$$

A small universal hash family??

- Here's one:
  - Pick a prime $p \geq M$.
  - Define

$$f_{a,b}(x) = ax + b \quad mod\ p$$

$$h_{a,b}(x) = f_{a,b}(x) \quad mod\ n$$

  - Define:

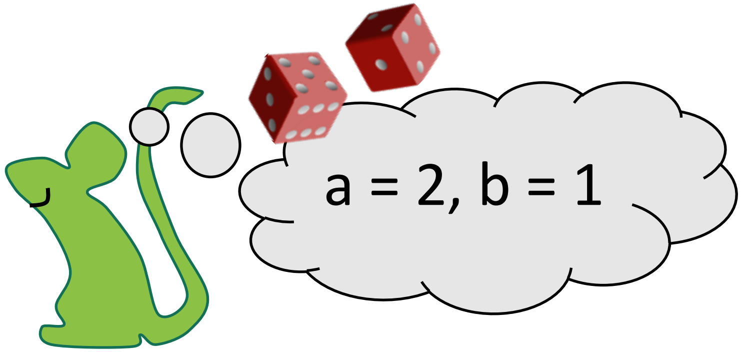
$$H = \{\ h_{a,b}(x)\ :\ a \in \{1, \ldots, p-1\}, b \in \{0, \ldots, p-1\}\ \}$$

How do you pick the prime number p that's not too larger than M?

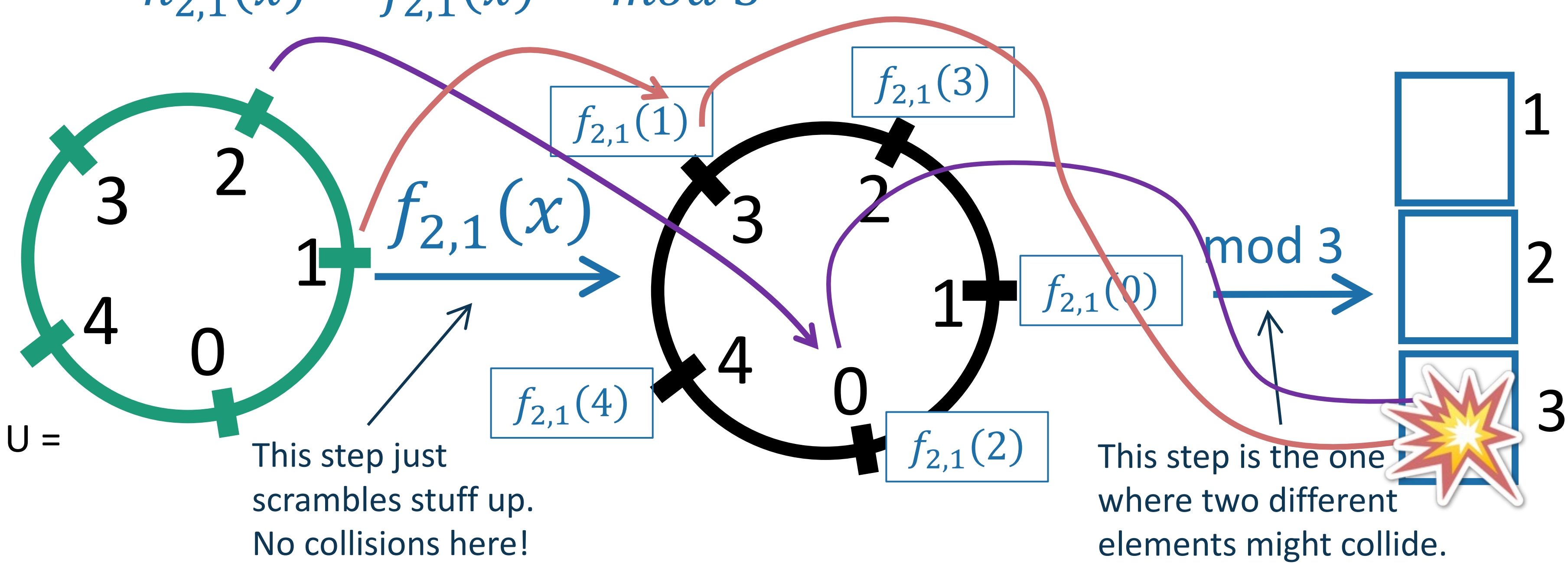
- Claim:

  H is a universal hash family.

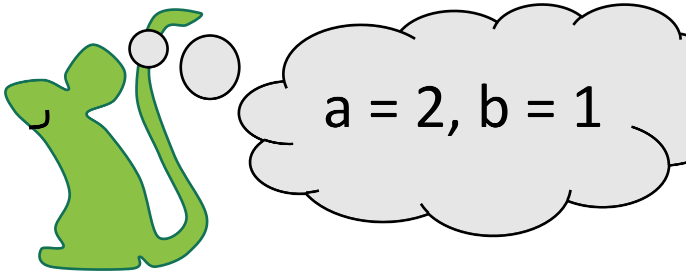
Say what?



- Example: M = p = 5, n = 3
- To draw h from H:
  - Pick a random a in {1,…,4}, b in {0,…,4}
- As per the definition:
  - $f_{2,1}(x) = 2x + 1 \quad mod\ 5$
  - $h_{2,1}(x) = f_{2,1}(x) \quad mod\ 3$



h takes O(log M) bits to store



- Just need to store two numbers:
  - a is in {1,…,p-1}
  - b is in {0,…,p-1}
  - So about 2log(p) bits
  - By our choice of p (close to M), that's O(log(M)) bits.
- Also, given a and b, h is fast to evaluate!
  - It takes time O(1) to compute h(x).
- Compare: direct addressing was M bits!
  - Twitter example: 2log(M) =2*280 log(128) = **3920** vs M = $\mathbf{128^{280}}$

Why does this work?

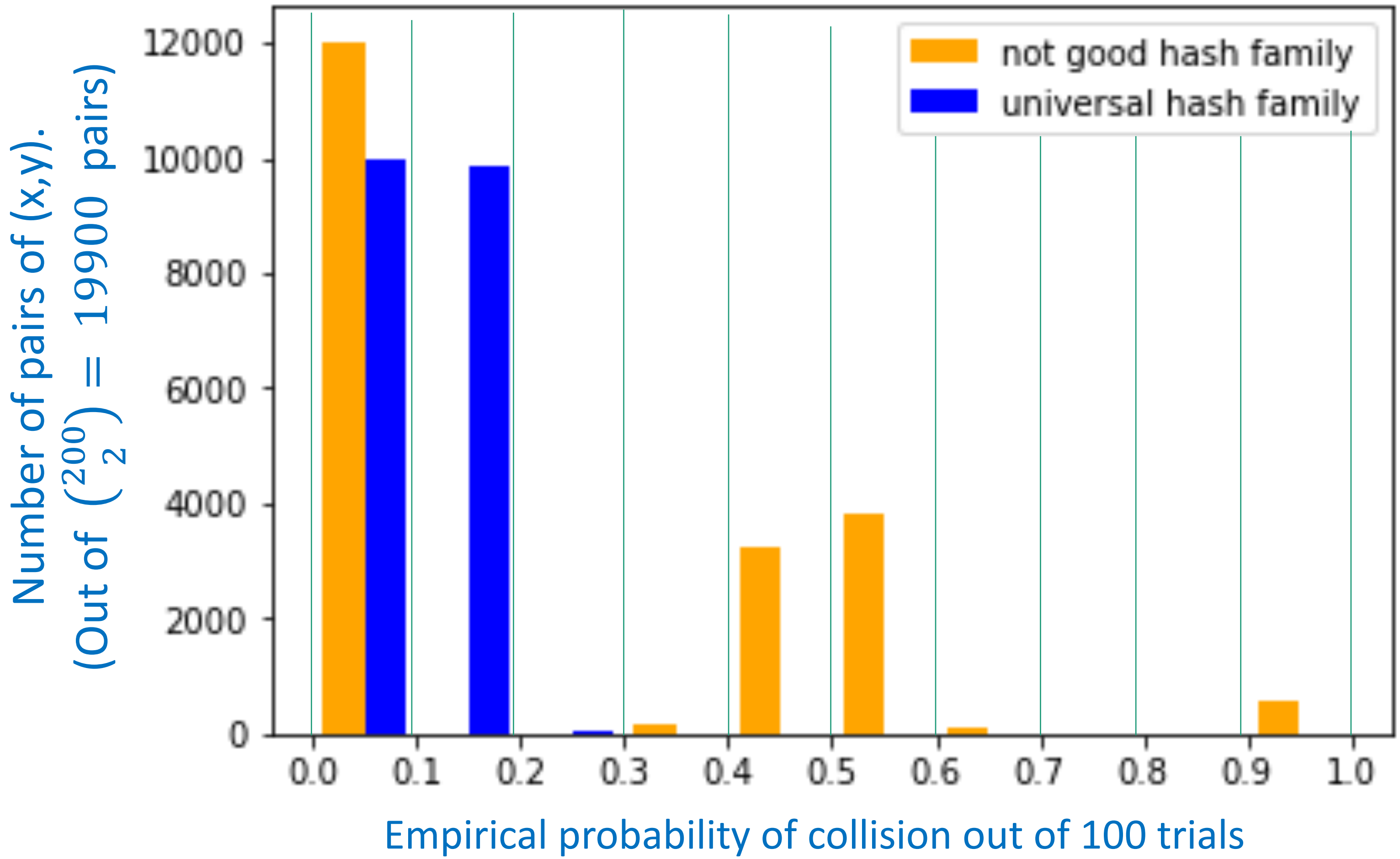
- This is actually a little complicated.
    - See lecture note if you are curious.
    - You are NOT RESPONSIBLE for the proof in this class.
    - But you should know that a universal hash family of size $O(M^2)$ exists.

Try to prove that this is a universal hash family!

But let's check that it **does** work

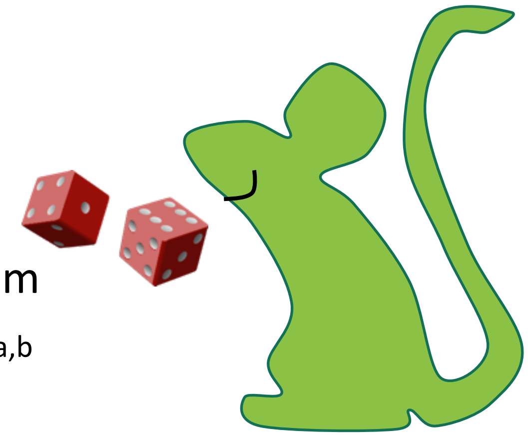
- Check out the Python notebook for lecture 8

M=200, n=10

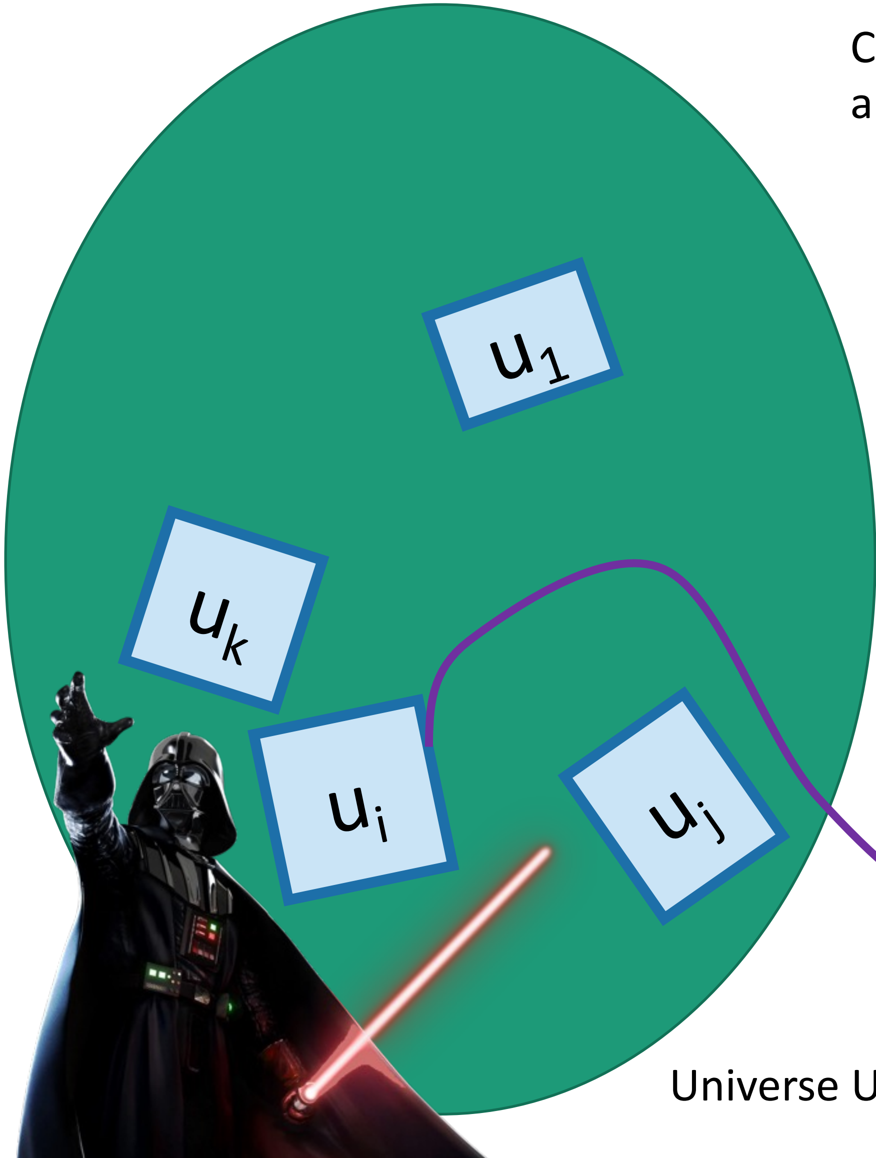


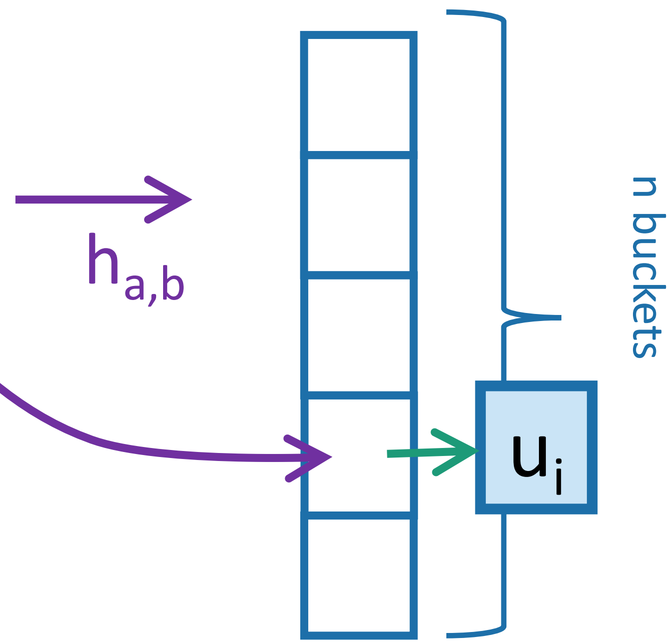
So the whole scheme will be

Choose a and b at random
and form the function $h_{a,b}$



We can store h in space
O(log(M)) since we just need
to store a and b.





Probably
these
buckets will
be pretty
balanced.

Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

- **Hash families** are the magic behind hash tables.

- **Universal hash families** are even more magic.

Recap

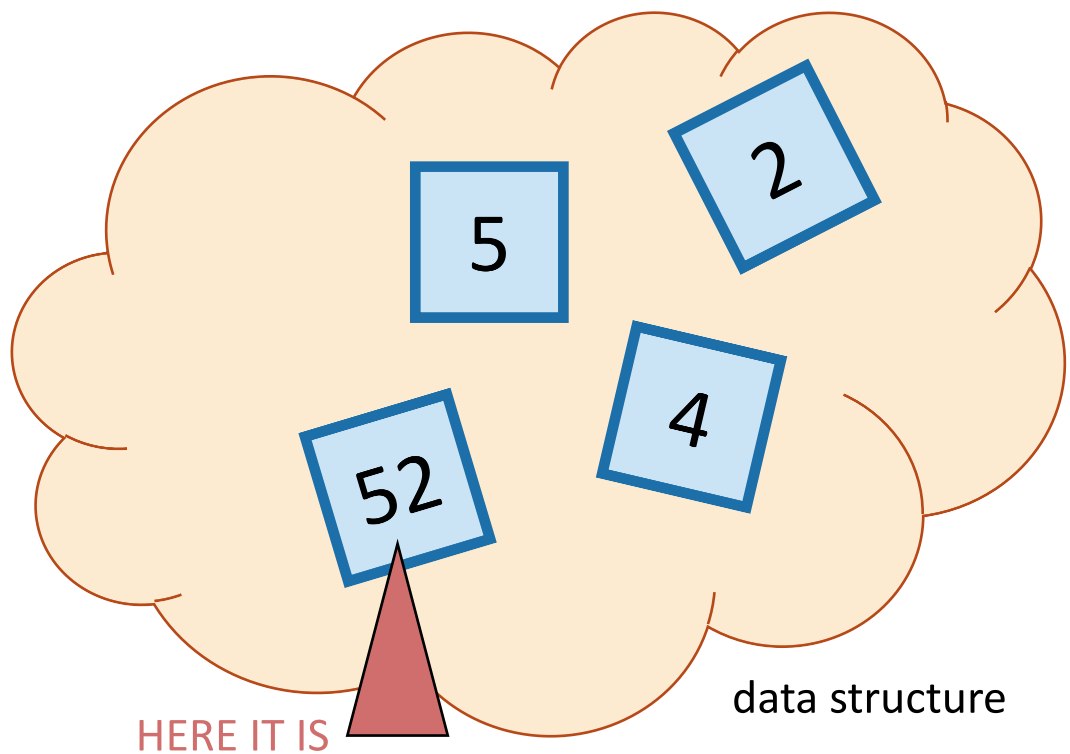
Want O(1) INSERT/DELETE/SEARCH

- We are interested in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT 

- DELETE 

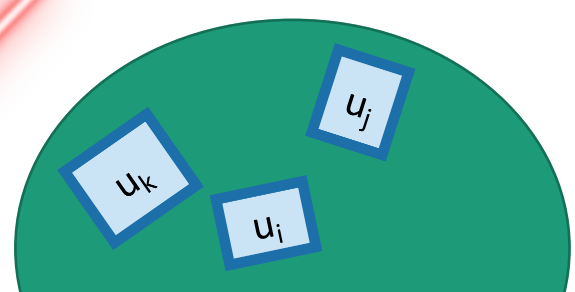
- SEARCH 



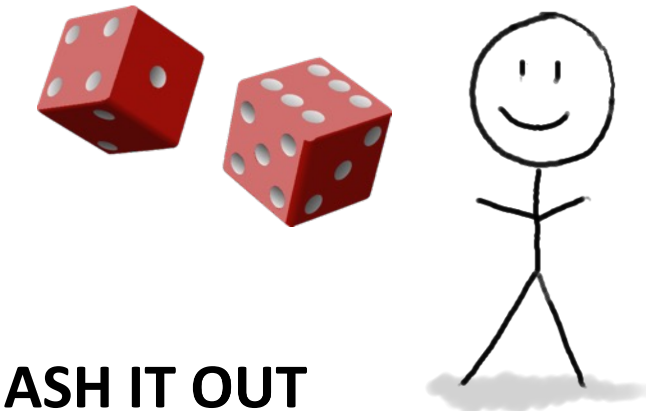
We studied this game

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.

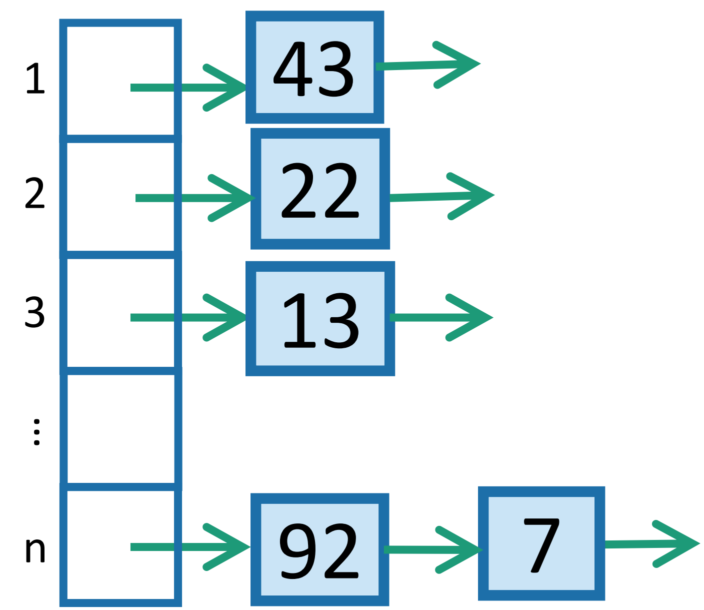
INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92



2. You, the algorithm, choose a **random** hash function $h: U \rightarrow \{1, \ldots, n\}$.



3. **HASH IT OUT**



Uniformly random h was good

- If we choose h uniformly at random,

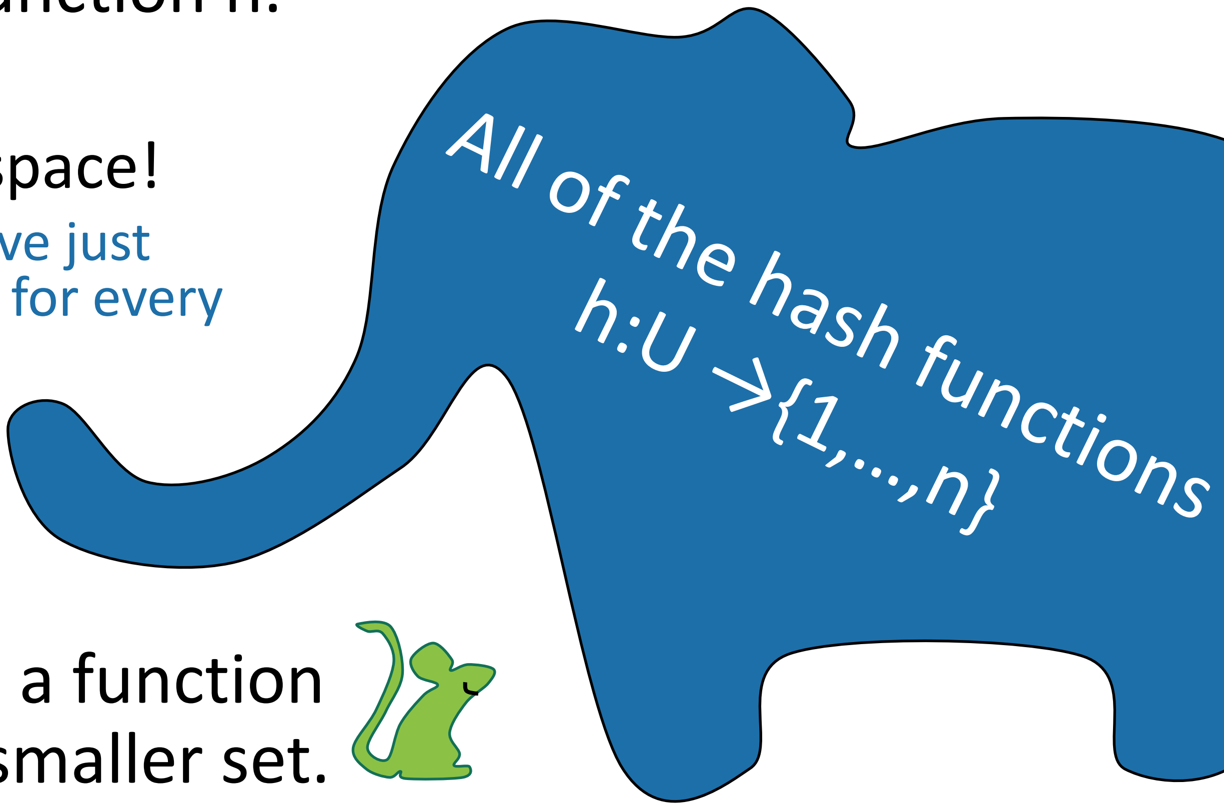
$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- That was enough to ensure that all INSERT/DELETE/SEARCH operations took O(1) time in expectation, even on adversarial inputs.

Uniformly random h was bad

- If we actually want to implement this, we have to store the hash function h.



- That takes a lot of space!
  - We may as well have just initialized a bucket for every single item in U.

- Instead, we chose a function randomly from a smaller set.

Universal Hash Families

H is a universal hash family if:

- If we choose h uniformly at random in H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{\, h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

This was all we needed to make sure that the buckets were balanced in expectation!

- We gave an example of a really small universal hash family, of size $O(M^2)$
- That means we need only O(log M) bits to store it.

Hashing a universe of size M into n buckets, where at most n of the items in M ever show up.

Conclusion:

- We can build a hash table that supports INSERT/DELETE/SEARCH in O(1) expected time
- Requires O(n log(M)) bits of space.
    - O(n) buckets
    - O(n) items with log(M) bits per item
    - O(log(M)) to store the hash function

That’s it for data structures (for now)

Now we can use these going forward!

Next Time

- Graph algorithms!

Before Next Time

- Pre-lecture exercise for Lecture 9
  - Intro to graphs