

# Lecture 8

## Hashing

# Announcements

- HW3 is out! Due Friday.

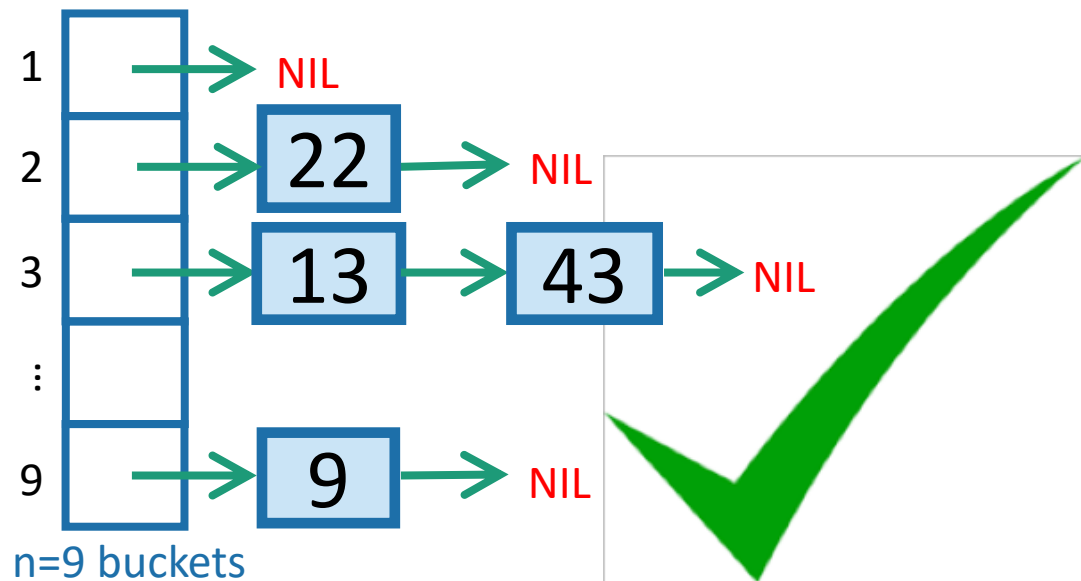
Girls Teaching Girls to Code is looking for mentors!

## Announcement:



- Girls Teaching Girls to Code is hosting our annual Code Camp, a day-long event during spring quarter that takes place on Stanford's campus and reaches several hundred high school girls. We're looking for Stanford students who are interesting in teaching and mentoring girls interested in computer science for Code Camp.
- **Check out the application**  
**here: <http://bit.ly/cc19mentorapp> (due by Saturday, February 9th)**
- Also, check out our info session about Code Camp on **Tuesday, February 5th from 4:30-5:30pm** at the Women's Community Center. Each person who RSVPs will get free boba!: <http://bit.ly/gtgtcinfoandboba>

# Today: hashing



# Outline




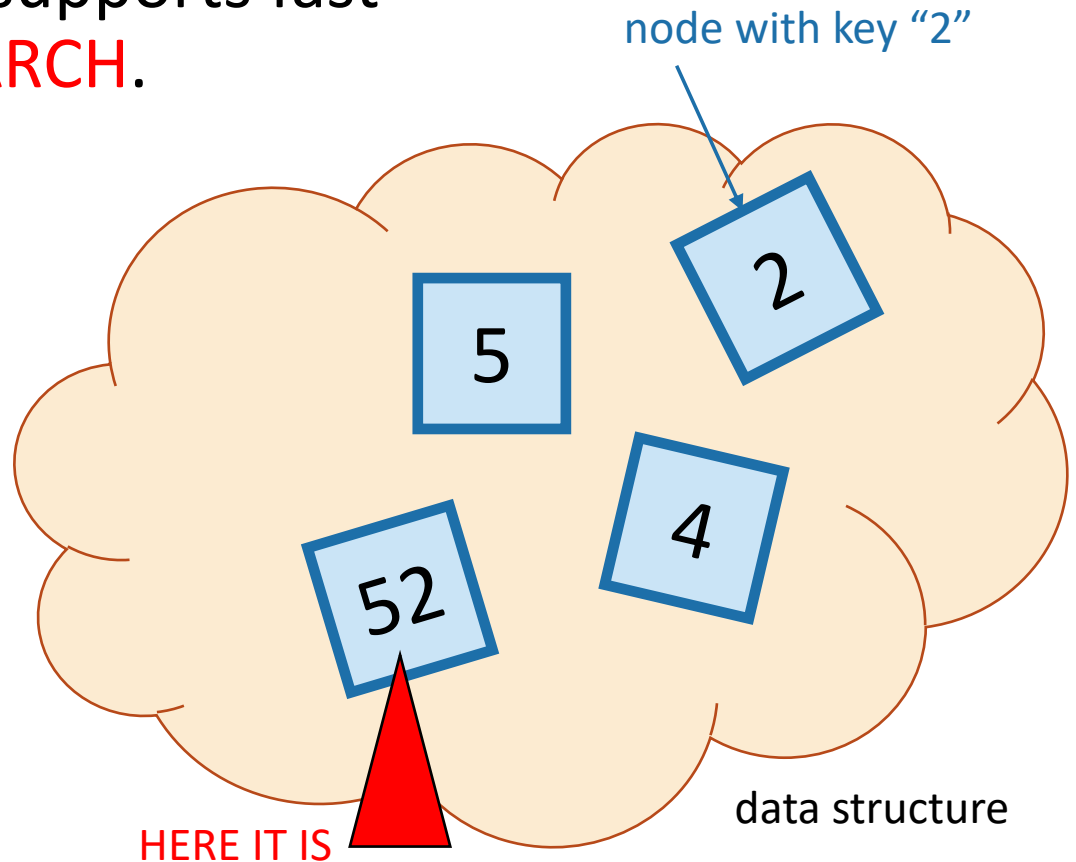
- Hash tables are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families**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

# Goal:

Just like last time

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

- **INSERT** 
- **DELETE** 
- **SEARCH** 



# 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, ...

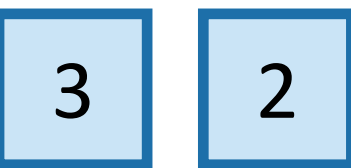
# One way to get $O(1)$ time

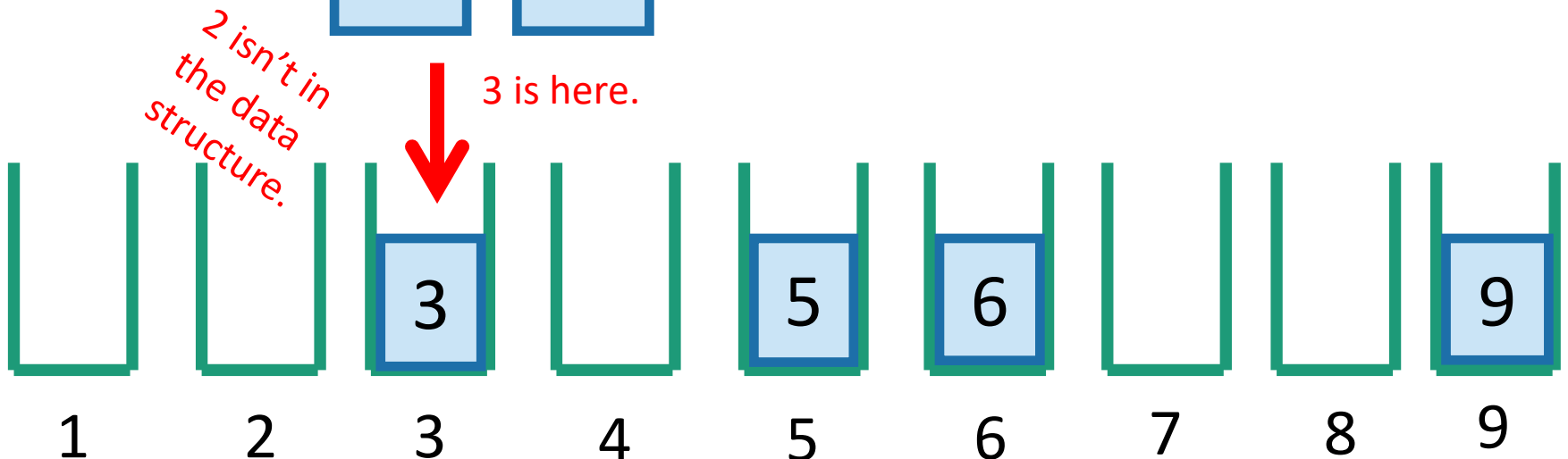
This is called  
“direct addressing”

- Say all keys are in the set  $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

• **INSERT:** 

• **DELETE:** 

• **SEARCH:** 



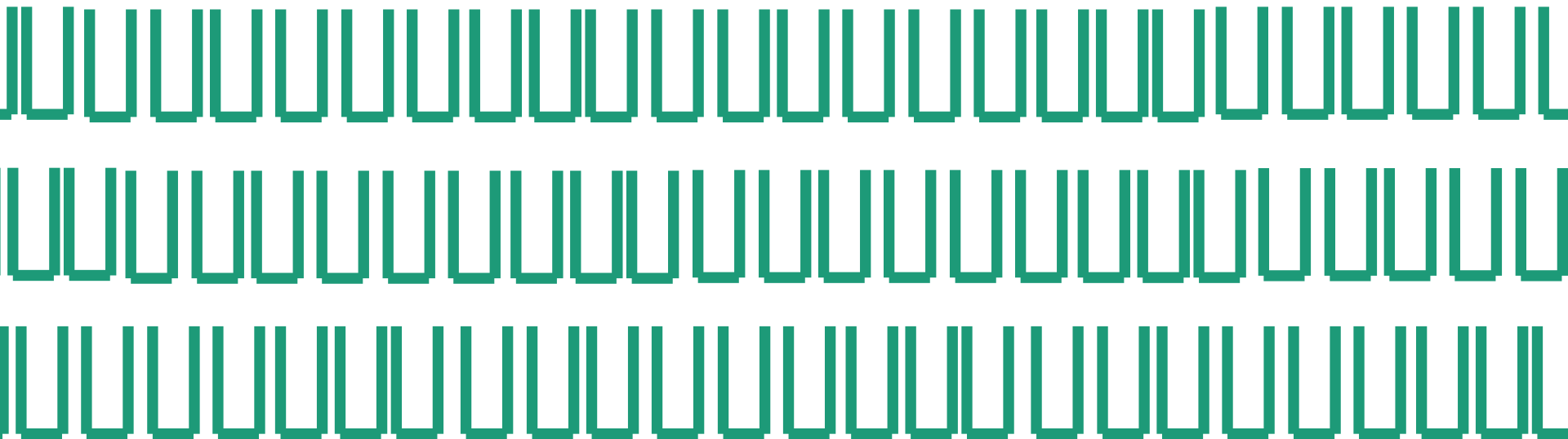


# That should look familiar



*The universe is  
really big!*

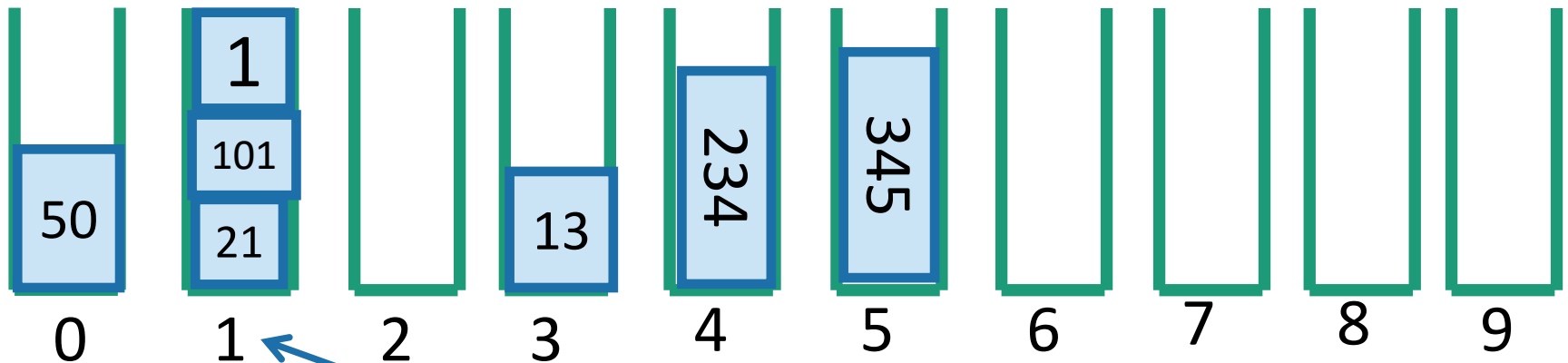
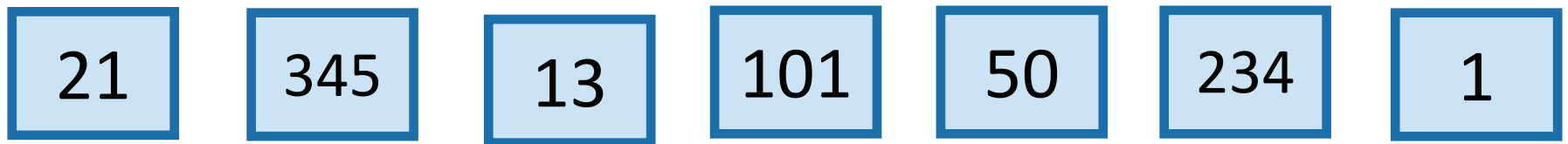
- Kind of like BUCKETSORT from Lecture 6.
- Same problem: if the keys may come from a “universe”  $U = \{1, 2, \dots, 10000000000\} \dots$



# Solution?

Put things in buckets based on one digit

INSERT:



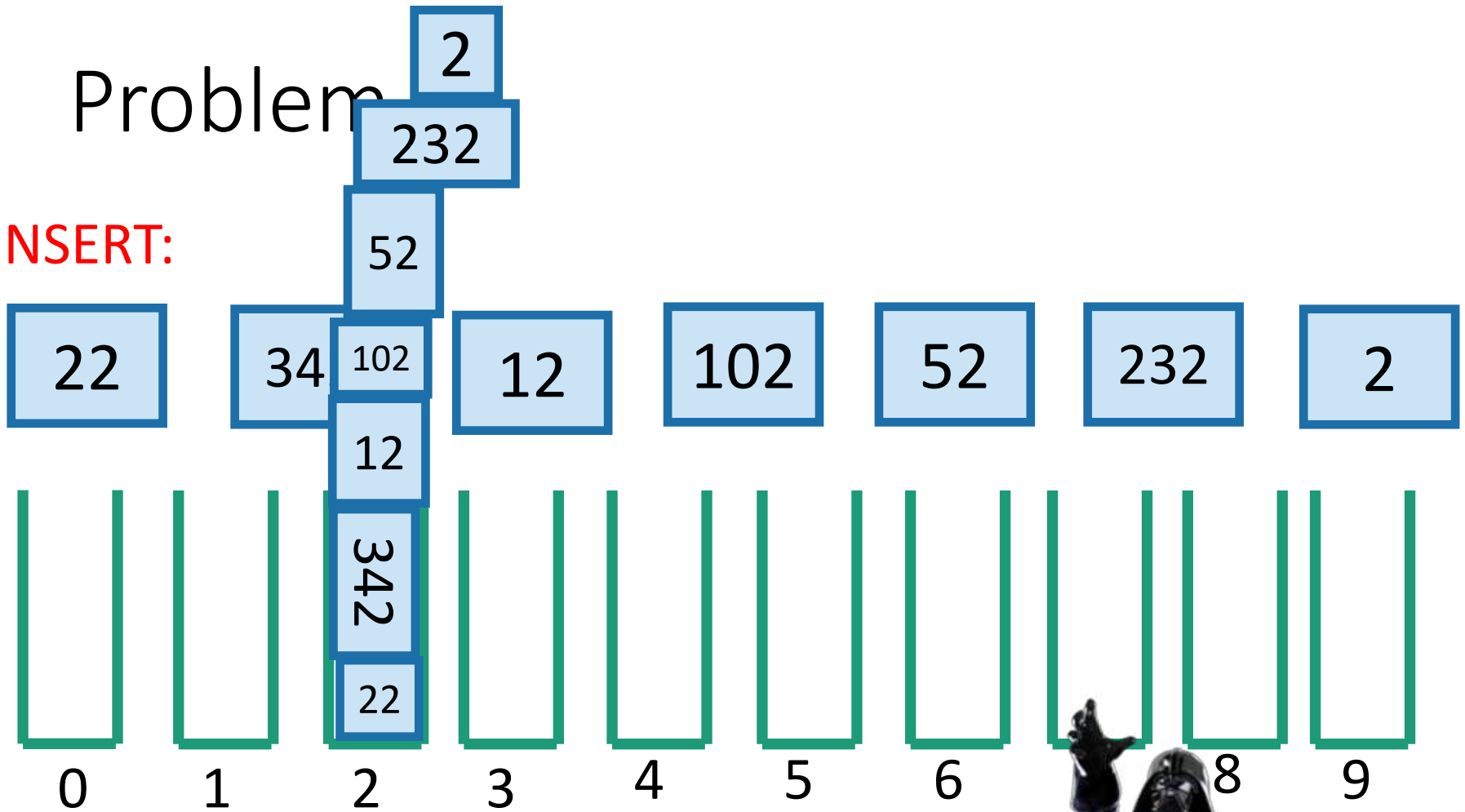
Now **SEARCH**



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

# Problem

INSERT:



Now **SEARCH**

22

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



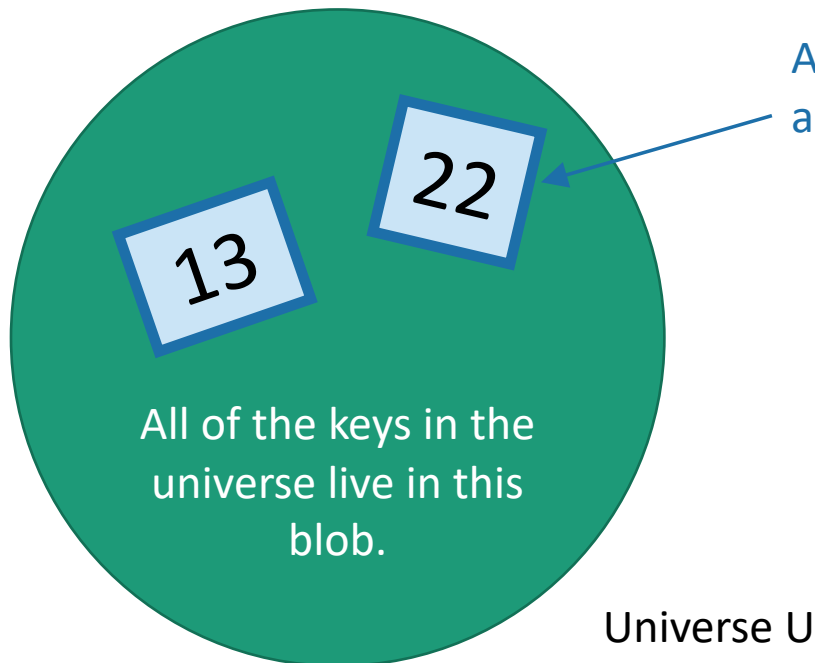
# Hash tables

- 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.



- We have a universe  $U$ , of size  $M$ .
  - $M$  is really big.
- But only a few (say at most  $n$  for today's lecture) elements of  $M$  are ever going to show up.
  - $M$  is waaaayyyyyyy bigger than  $n$ .
- But we don't know which ones will show up in advance.



A few elements are special and will actually show up.

Example:  $U$  is the set of all strings of at most 140 ascii characters. ( $128^{140}$  of them).

The only ones which I care about are those which appear as trending hashtags on twitter. [#hashinghashtags](#)

*There are way fewer than  $128^{140}$  of these.*

Examples aside, I'm going to draw elements like I always do, as blue boxes with integers in them...

# The previous example

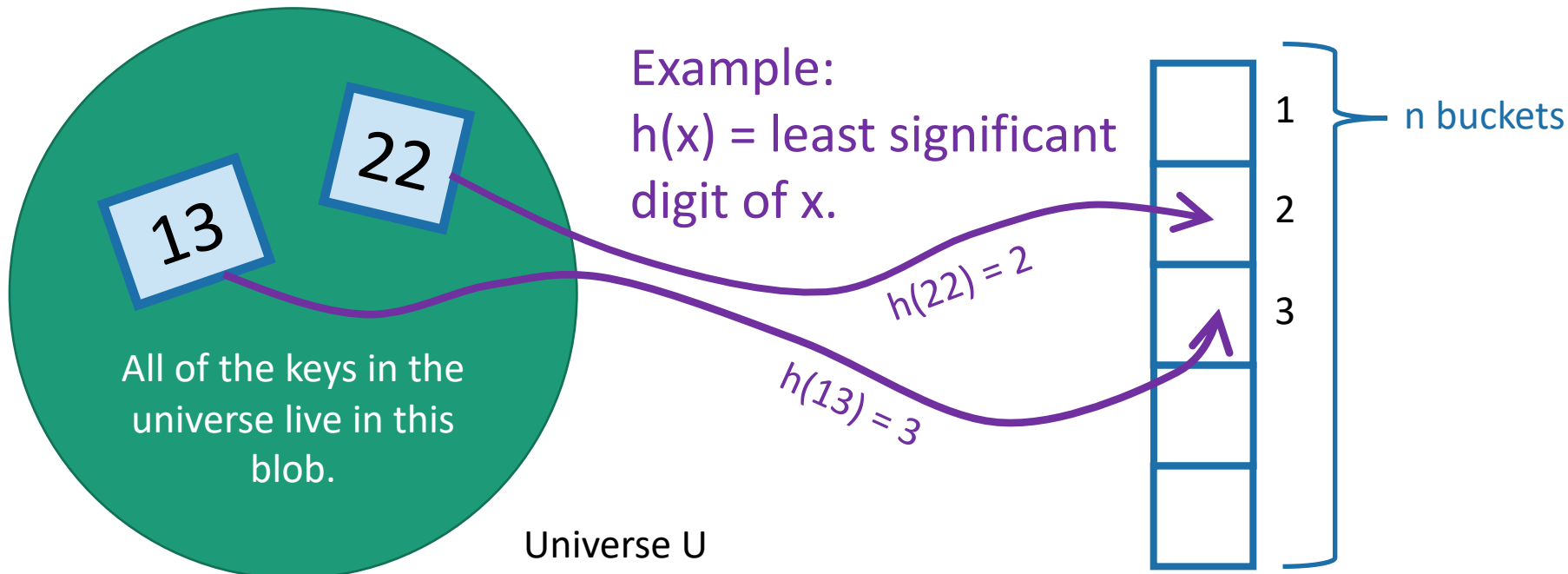
with this terminology

- We have a universe  $U$ , of size  $M$ .
  - at most  $n$  of which will show up.
- $M$  is waaaaayyyyyy bigger than  $n$ .
- We will put items of  $U$  into  $n$  buckets.
- There is a *hash function*  $h: U \rightarrow \{1, \dots, n\}$  which says what element goes in what bucket.

For this lecture, I'm assuming that the number of things is the same as the number of buckets, both are  $n$ .

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

**#buckets =  $O(\text{\#things which show up})$**



# This is a **hash table** (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  $O(\text{length}(\text{list}))$ .
- $h: U \rightarrow \{1, \dots, n\}$  can be any function:
  - but for concreteness let's stick with  $h(x) = \text{least significant digit of } x$ .

**For demonstration purposes only!**

This is a terrible hash function! Don't use this!

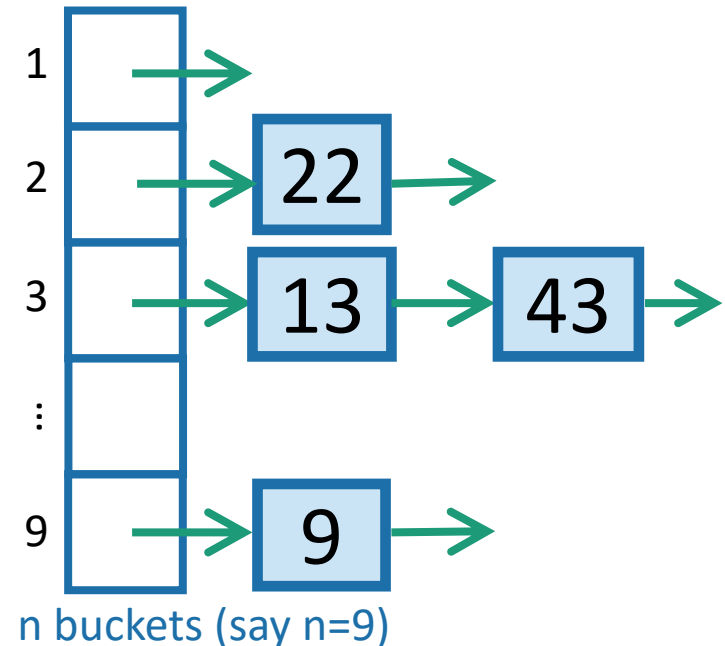


INSERT:



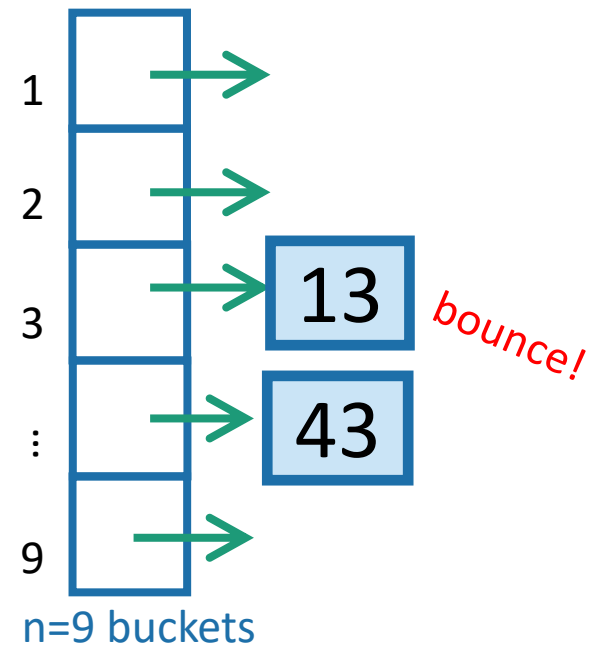
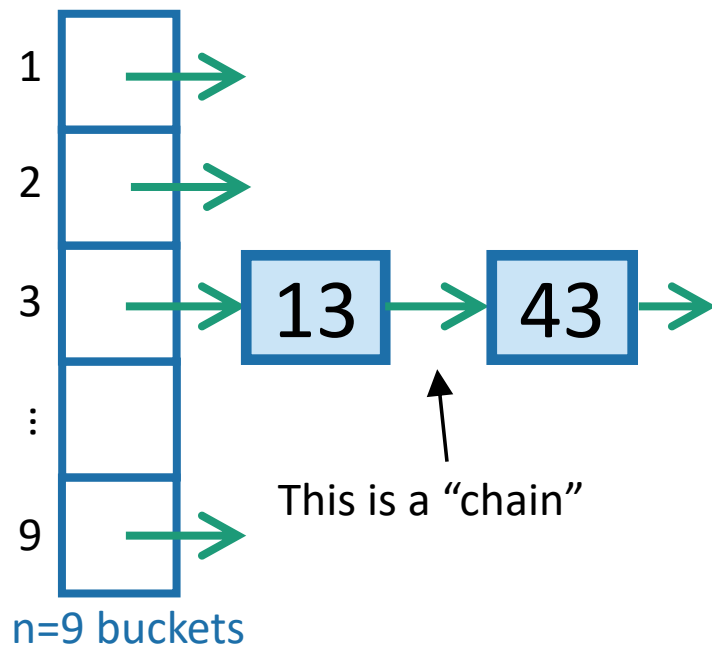
SEARCH 43:

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



## Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS if you are interested!



\end{Aside}



# This is a **hash table** (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  $O(\text{length}(\text{list}))$ .
- $h: U \rightarrow \{1, \dots, n\}$  can be any function:
  - but for concreteness let's stick with  $h(x) = \text{least significant digit of } x$ .

**For demonstration purposes only!**

This is a terrible hash function! Don't use this!

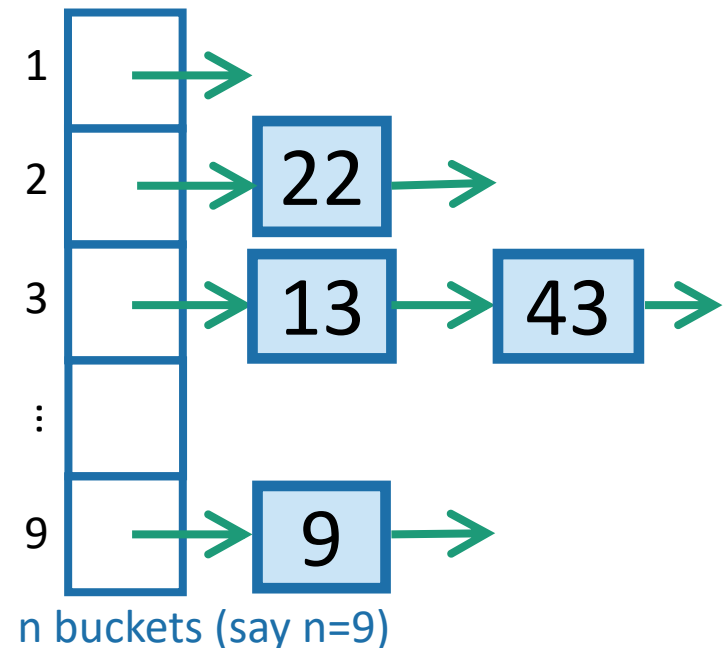


INSERT:



SEARCH 43:

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

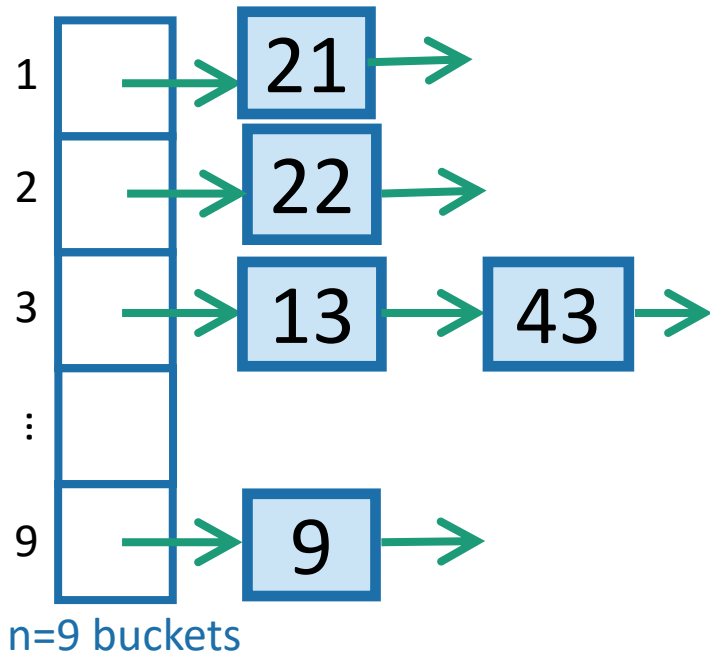


Sometimes this a **good idea**

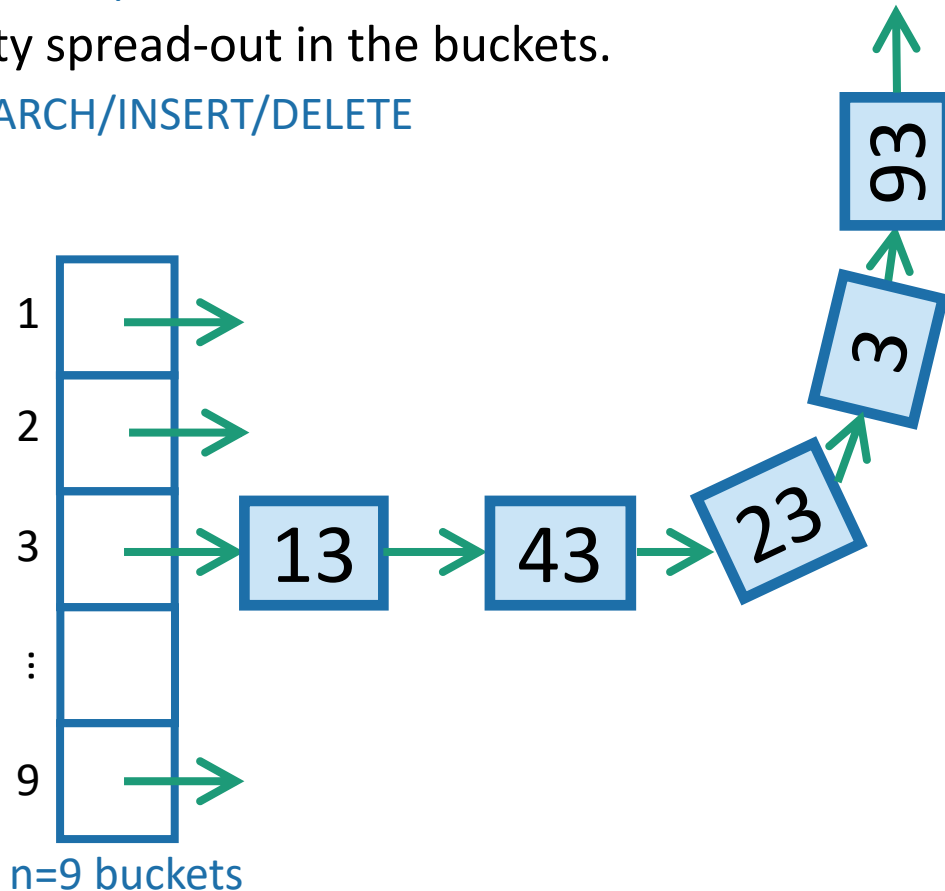
Sometimes this is a **bad idea**

- How do we pick that function so that this is a good idea?

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



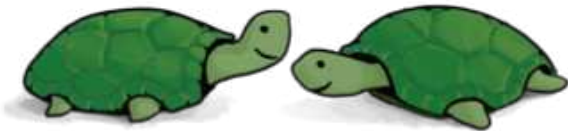
vs.



# Worst-case analysis

- Goal: Design a function  $h: U \rightarrow \{1, \dots, n\}$  so that:
  - No matter what input (fewer than  $n$  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-Pair-Share Terrapins



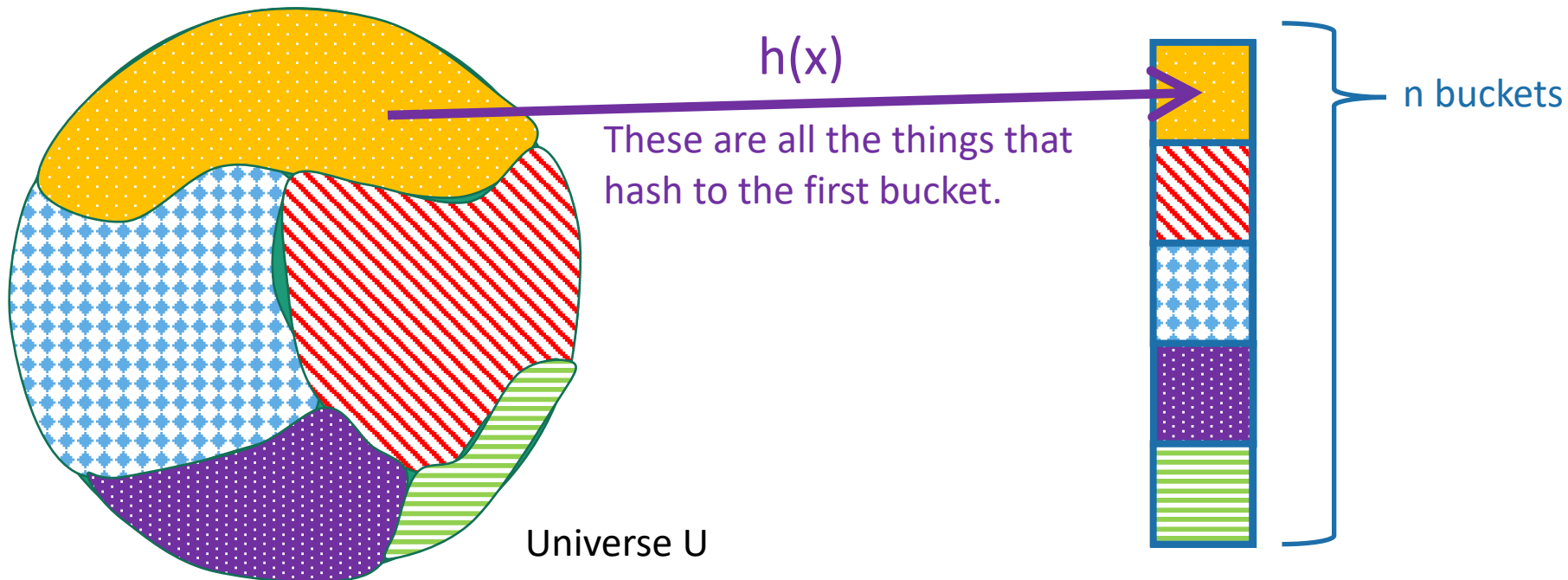
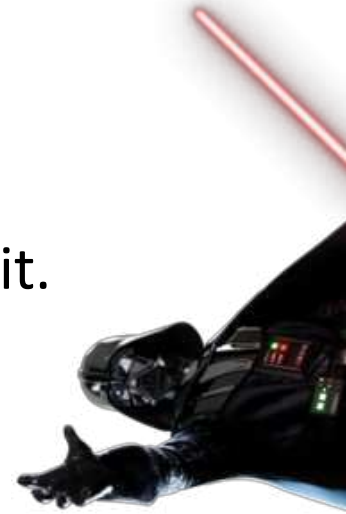
This is impossible!



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



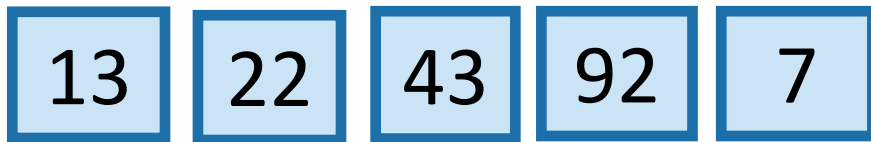
# The game



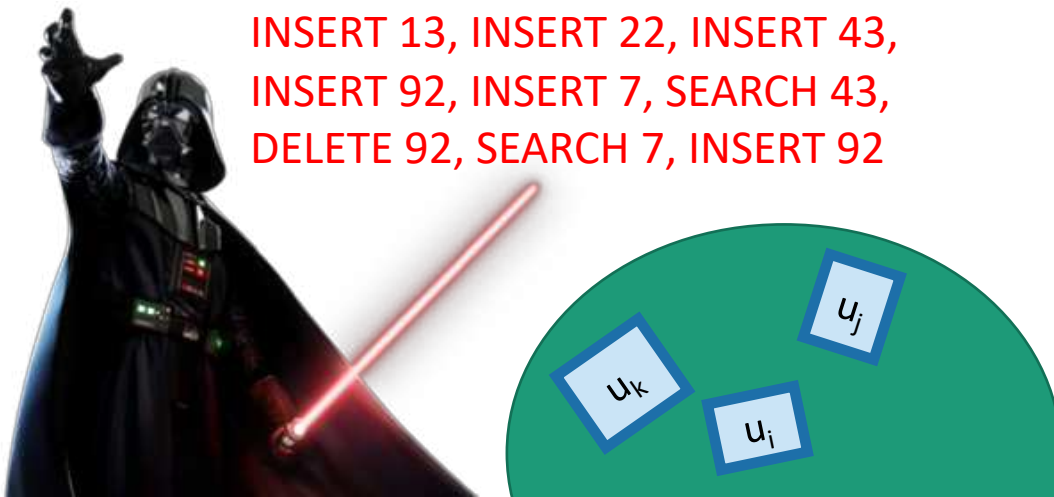
Plucky the pedantic penguin

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

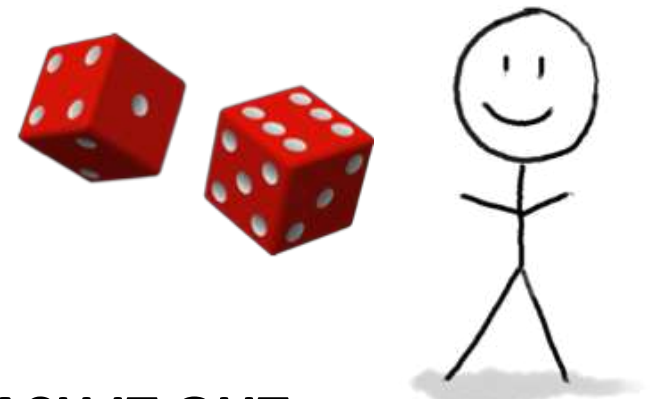
1. An adversary chooses any  $n$  items  $u_1, u_2, \dots, u_n \in U$ , and any sequence of INSERT/DELETE/SEARCH operations on those items.



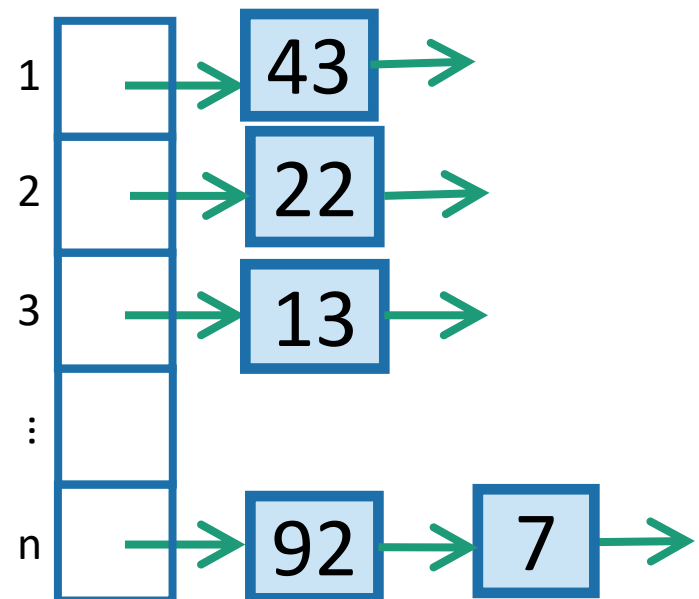
INSERT 13, INSERT 22, INSERT 43,  
INSERT 92, INSERT 7, SEARCH 43,  
DELETE 92, SEARCH 7, INSERT 92



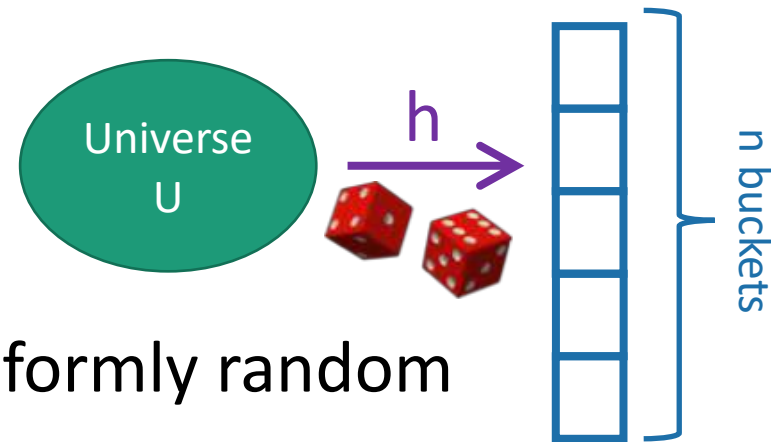
2. You, the algorithm, chooses a **random** hash function  $h: U \rightarrow \{1, \dots, n\}$ .



3. **HASH IT OUT** #hashpuns



# Example



- Say that  $h: U \rightarrow \{1, \dots, n\}$  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



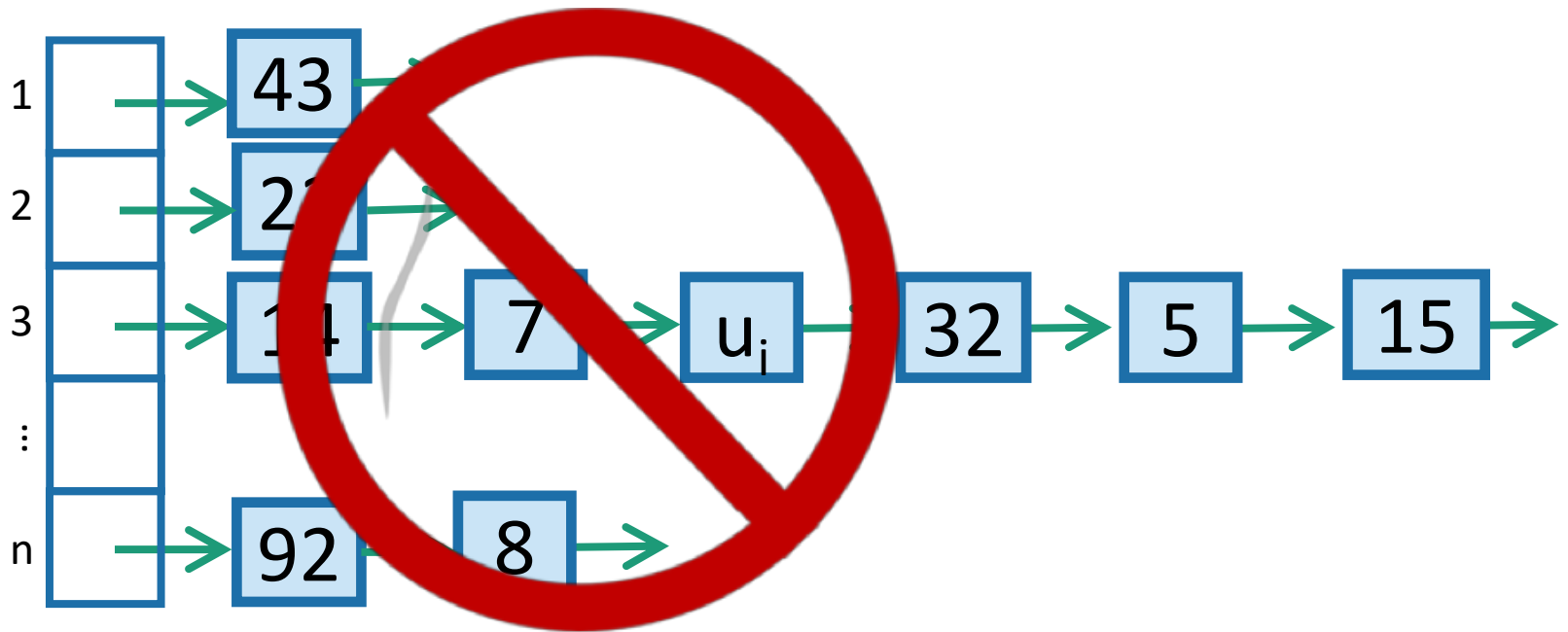
Plucky the Pedantic  
Penguin

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**.



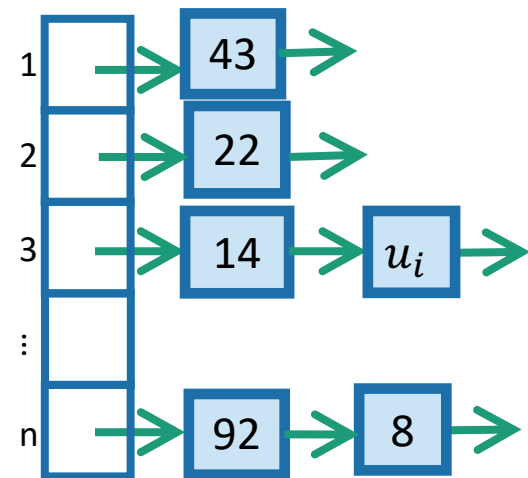
# More precisely

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  $u_1, u_2, \dots, u_n$ , to put into the hash table, and for all  $i \in \{1, \dots, n\}$ ,  
 $E[\text{number of items in } u_i \text{ 's bucket}] \leq 2.$
- If that were the case:
  - For each INSERT/DELETE/SEARCH operation involving  $u_i$ ,

$$E[\text{time of operation}] = O(1)$$

This is what we wanted at the beginning of lecture!



So we want:

- For all  $i=1, \dots, n$ ,  
     $E[\text{number of items in } u_i \text{'s bucket}] \leq 2.$

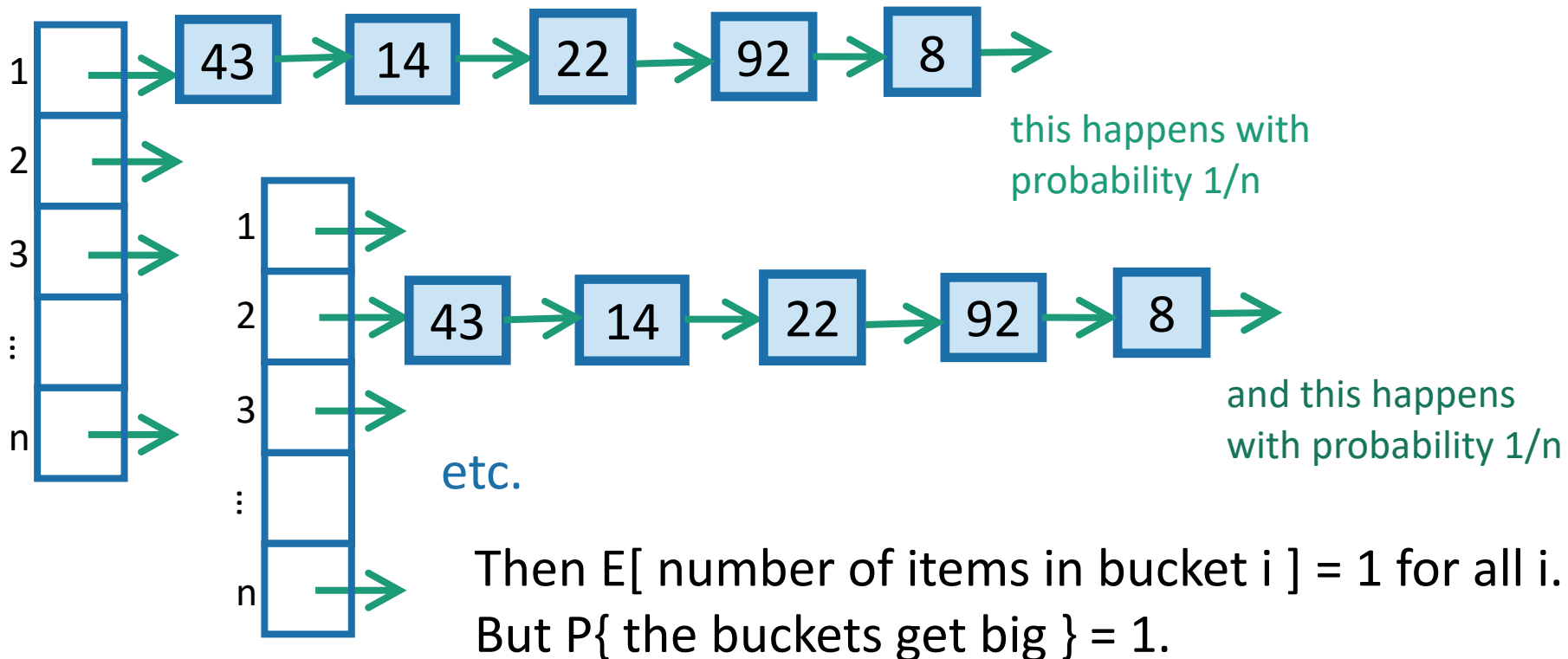
## Aside: why not:

This slide  
skipped in class

- For all  $i=1,\dots,n$ :

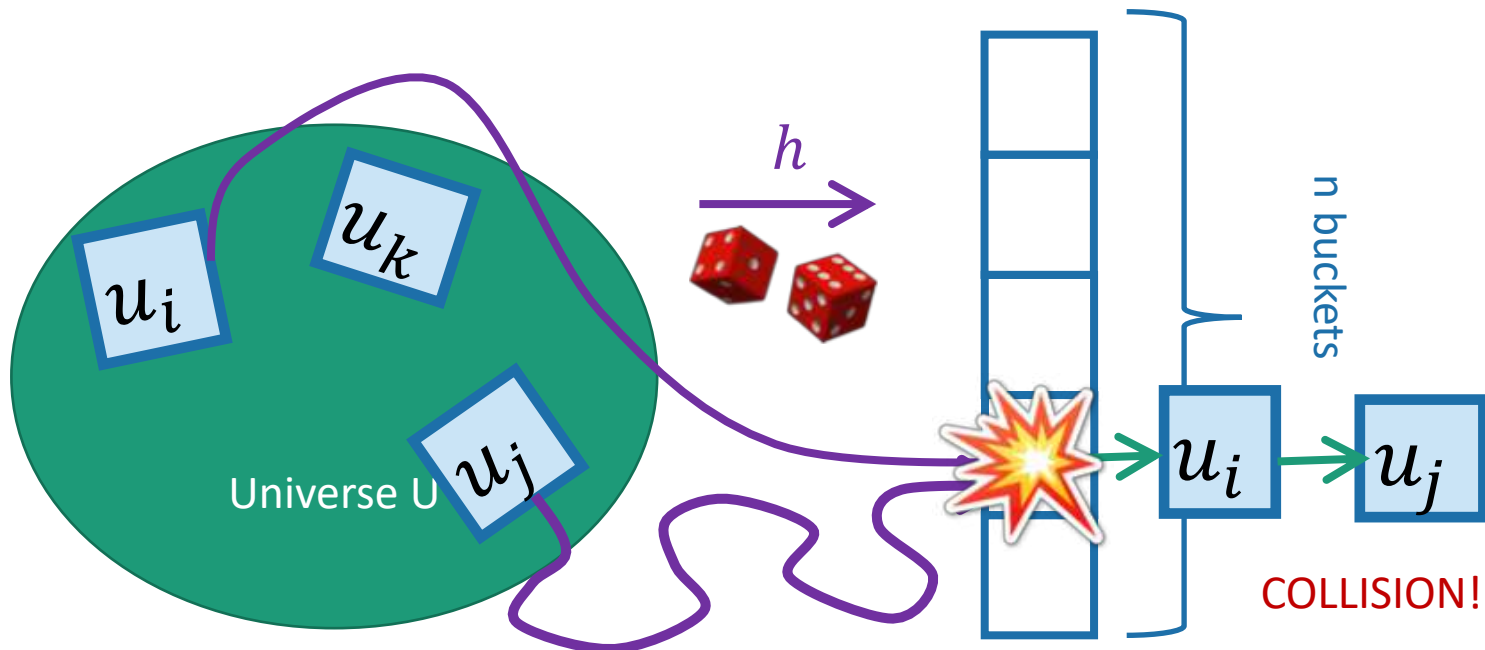
$E[\text{number of items in bucket } i] \leq \underline{\hspace{1cm}}?$

Suppose that:



Expected number of items in  $u_i$ 's bucket?

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



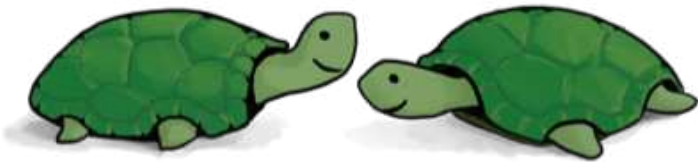
# That's great!

- We just showed:
  - For all ways a bad guy could choose  $u_1, u_2, \dots, u_n$ , to put into the hash table, and for all  $i \in \{1, \dots, n\}$ ,  
$$E[\text{number of items in } u_i \text{'s bucket}] \leq 2.$$
- Which implies:
  - No matter what sequence of operations and items the bad guy chooses,  
$$E[\text{time of INSERT/DELETE/SEARCH}] = O(1)$$
- So our solution is:

Pick a uniformly random hash function?

# What's wrong with this plan?

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



Think-Pair-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:

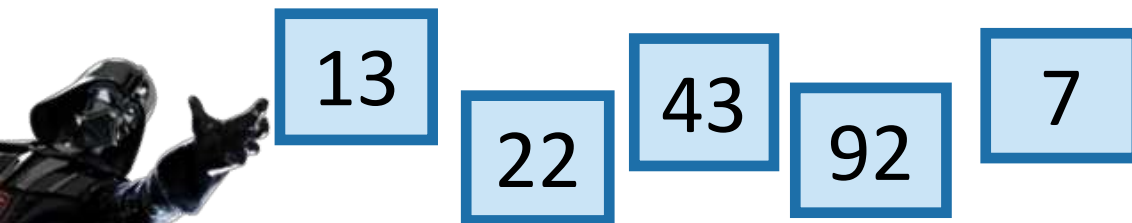
All of the  $M$  things in the universe

$x$	$h(x)$
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  $M\log(n)$  bits.
- In contrast, direct addressing (initializing a bucket for every item in the universe) requires only  $M$  bits....

# Another thought...

- Just remember  $h$  on the relevant values

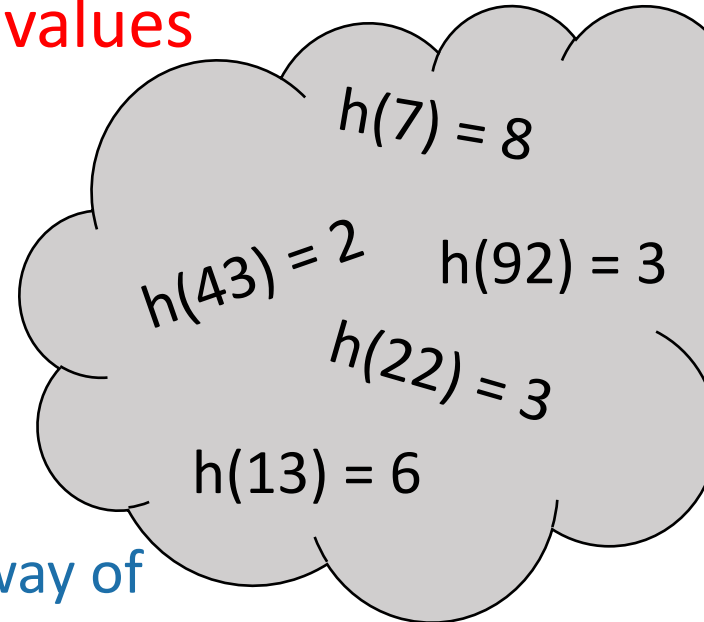


$$h(13) = 6$$

$$h(22) = 3$$

We need some way of  
storing keys and values  
with  $O(1)$

INSERT/DELETE/SEARCH...



Algorithm now



Algorithm later

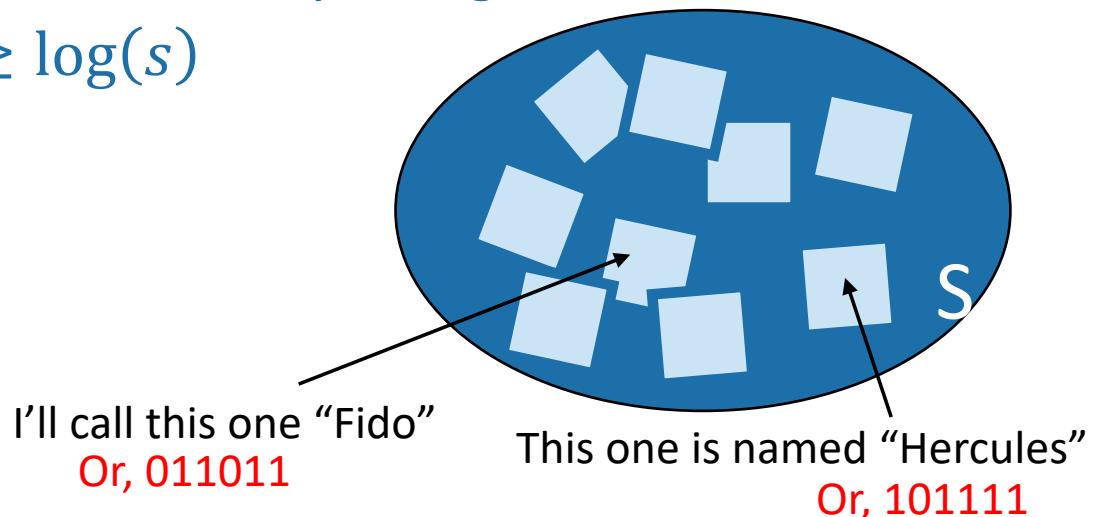
# Could we store a uniformly random $h$ without using a lookup table?

- Maybe there's a different way to store  $h$  that uses less space?



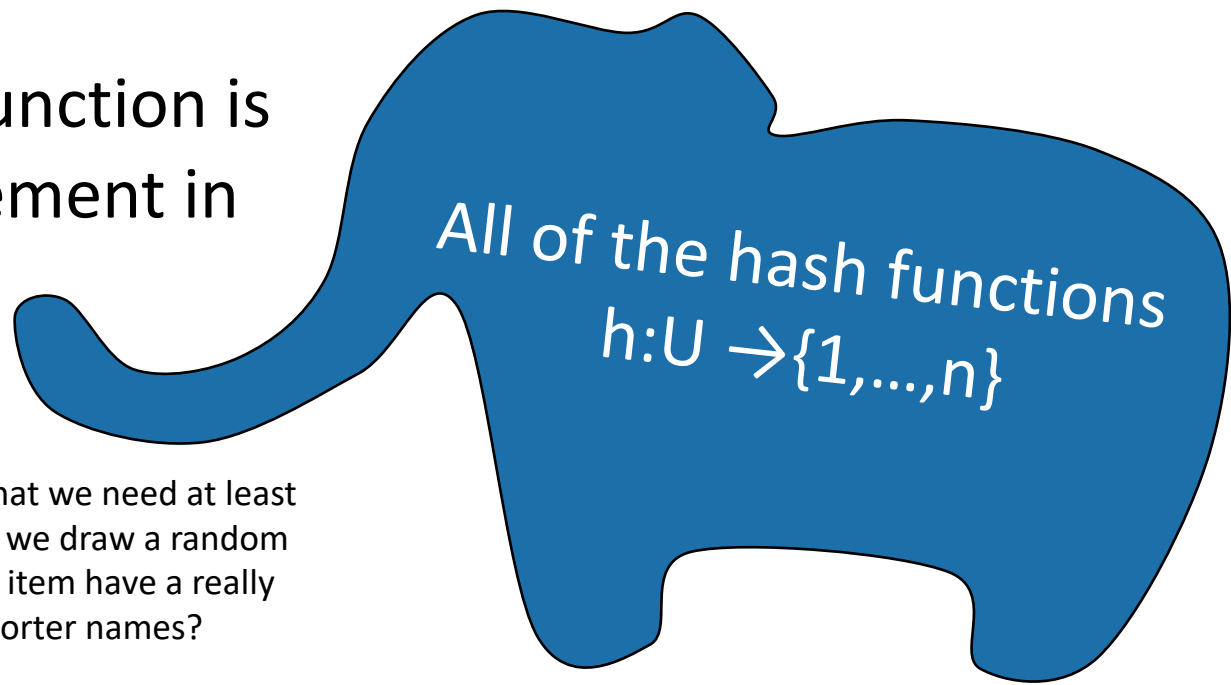
# Aside: description length

- Say I have a set  $S$  with  $s$  things in it.
- I get to write down the elements of  $S$  however I like, in binary using  $b$  bits.
- Then  $b \geq \log(s)$  :
  - There are  $2^b$  binary strings of length  $b$ .
  - I need to have at least as many strings as I have items in  $S$ .
  - So  $s \leq 2^b$  aka  $b \geq \log(s)$



# We need $M\log(n)$ bits to store a random hash function $h:U \rightarrow \{1,\dots,n\}$

- Say that this elephant-shaped blob represents the set of all hash functions.
- It has size  $n^M$ . (Really big!)
- To write down a random hash function, we need  $\log(n^M) = M\log(n)$  bits.
- A random hash function is just a random element in this set.



Technically we should argue that we need at least  $M\log(n)$  bits on average when we draw a random hash function... why can't one item have a really long name and others have shorter names?



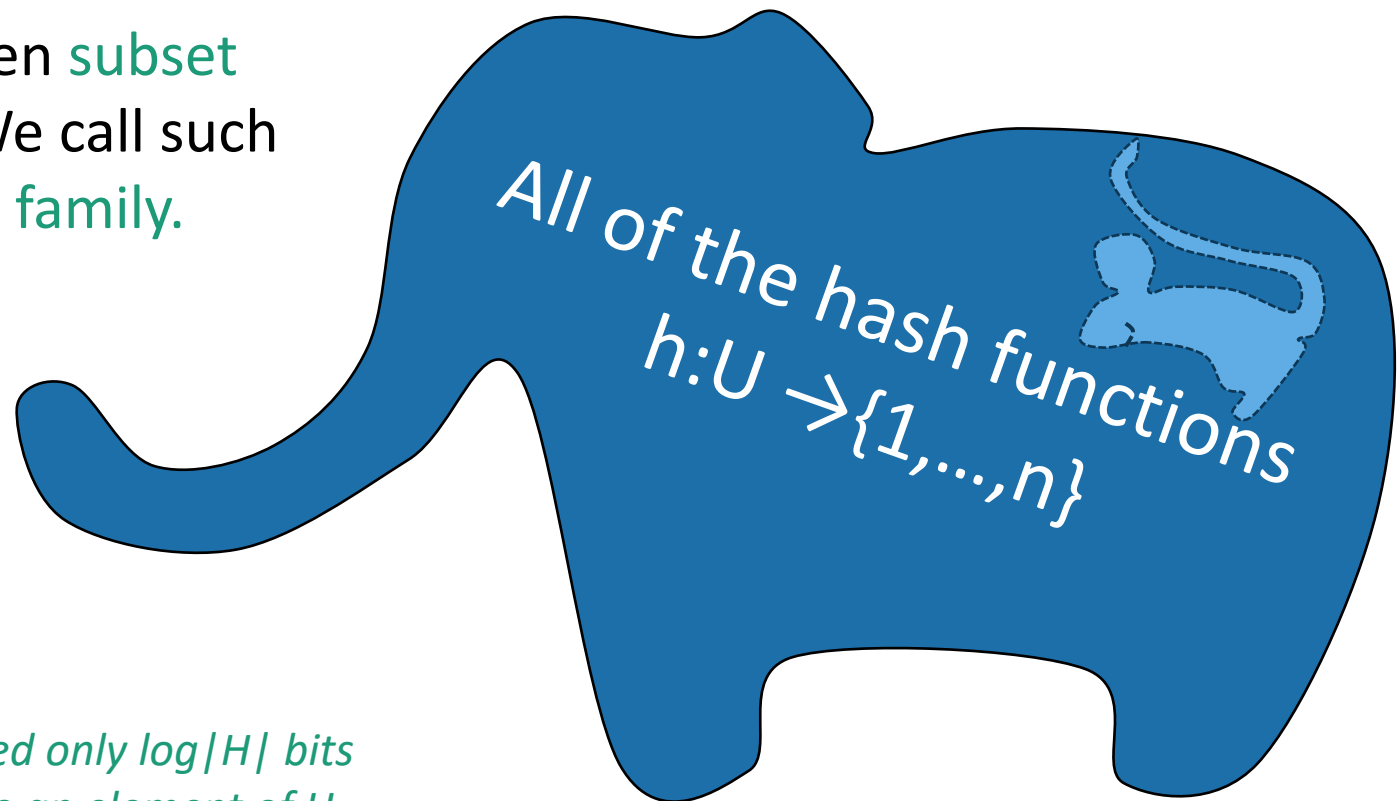
# Solution

- 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  $H$ .*



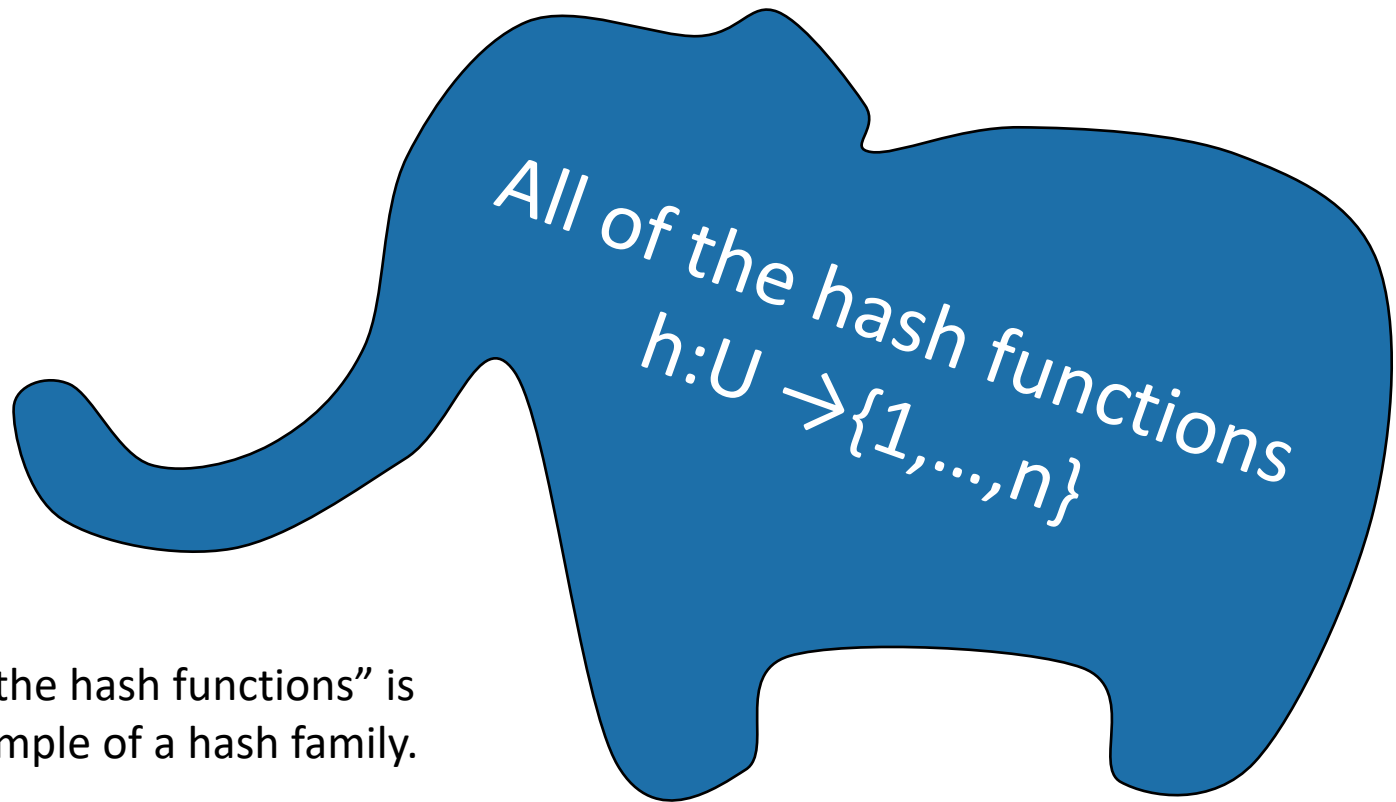
# Outline

- **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.

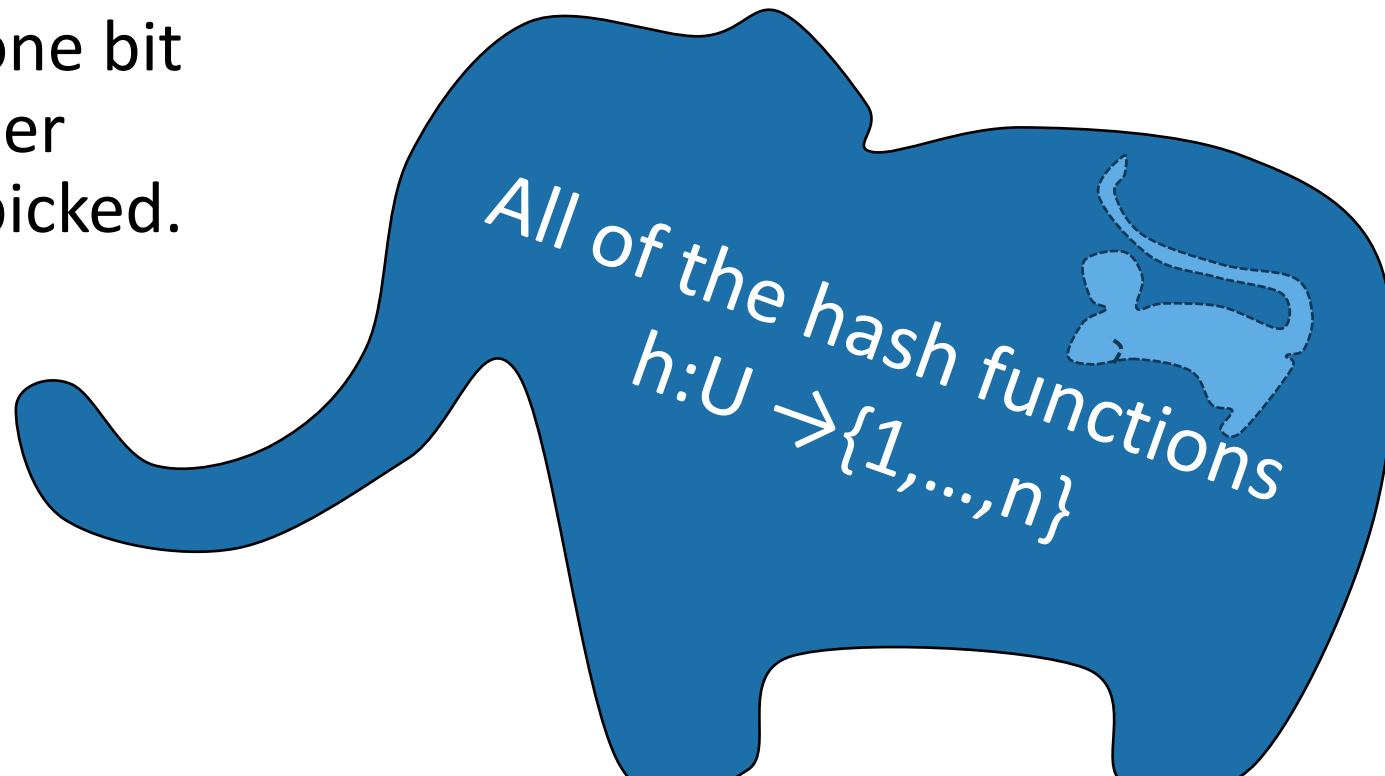
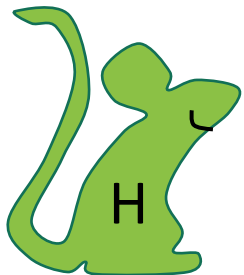


# Example:

a smaller hash family

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

- $H = \{ \text{function which returns the least sig. digit,} \\ \text{function which returns the most sig. digit} \}$
- 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

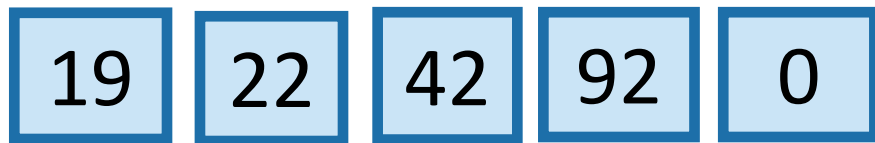
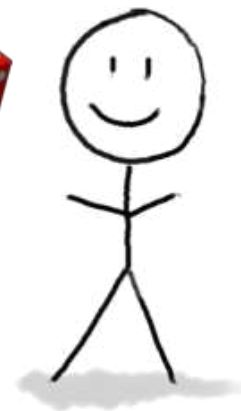
$H = \{h_0, h_1\}$

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.

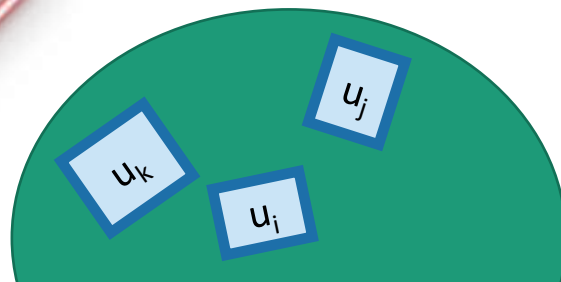
2. You, the algorithm, chooses a **random** hash function  $h: U \rightarrow \{0, \dots, 9\}$ . Choose it randomly from  $H$ .



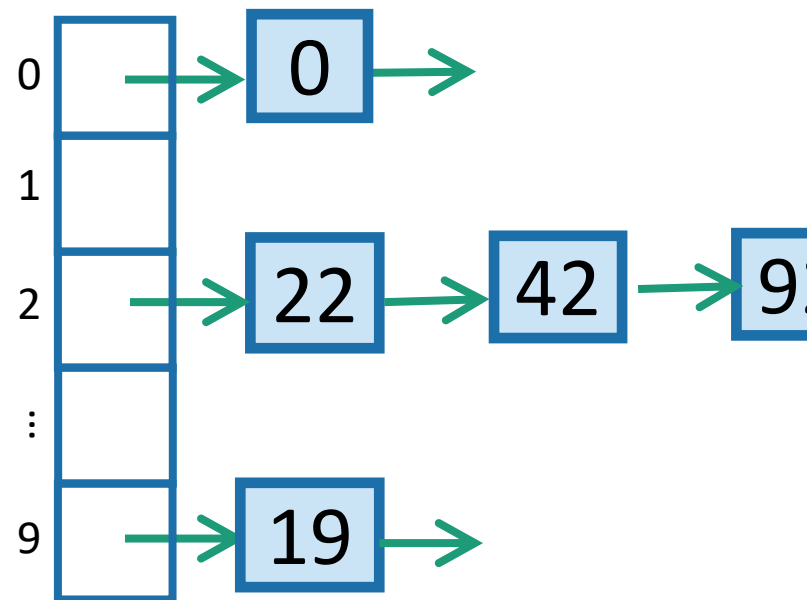
I picked  $h_1$



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

- $H = \{ \text{function which returns least sig. digit,} \\ \text{function which returns most sig. digit} \}$
- 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

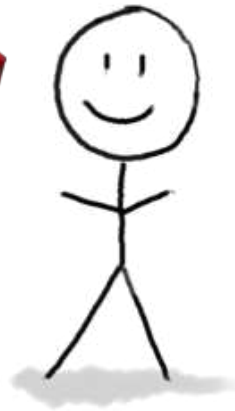
$H = \{h_0, h_1\}$

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.

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



I picked  $h_1$



## 3. HASH IT OUT #hashpuns

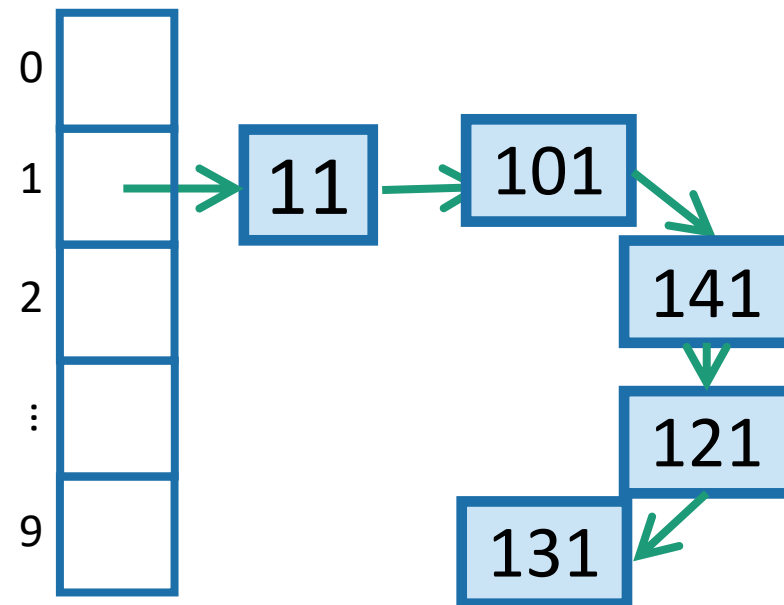
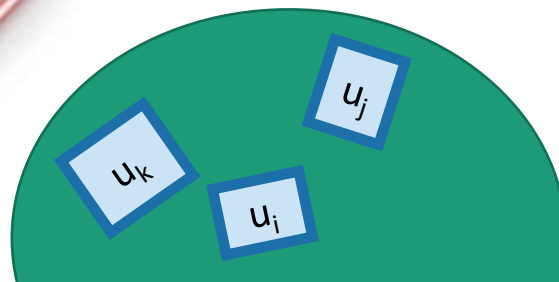
101

11

121

141

131



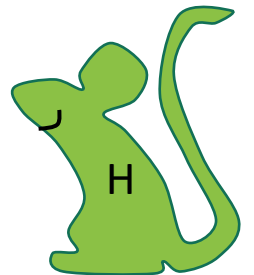
# Outline

- **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?

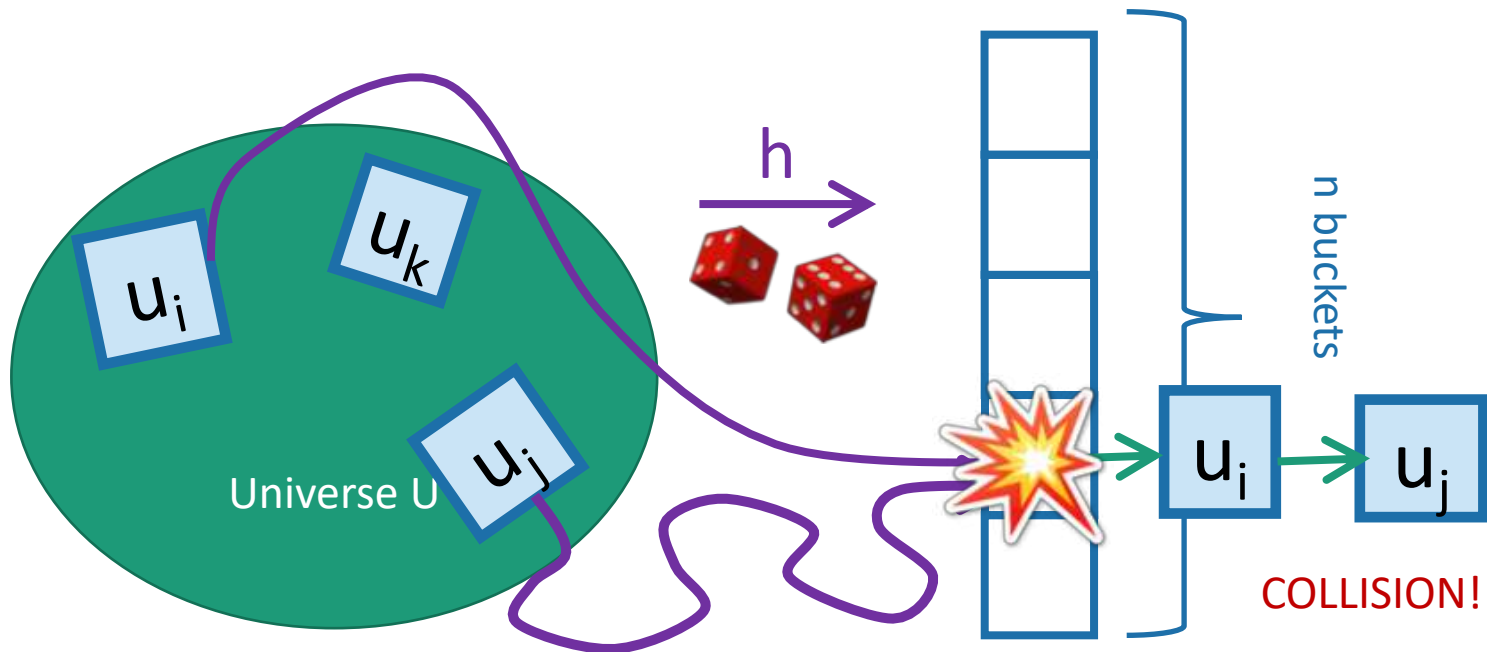
- Definitely not like in that example.
- Let's go back to that computation from earlier....



Expected number of items in  $u_i$ 's bucket?

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

All that we needed  
was that this is  $1/n$



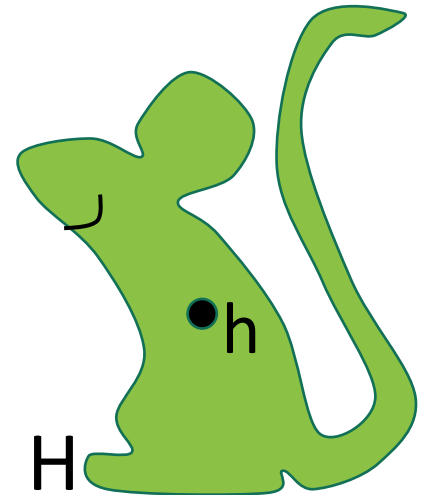
# Strategy

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

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

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

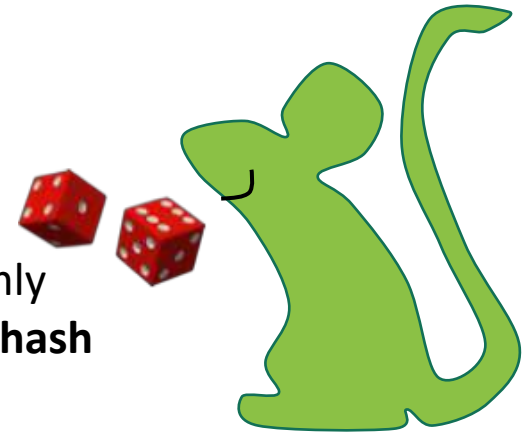
- A hash family  $H$  that satisfies this is called a **universal hash family**.
- Then we still get  $O(1)$ -sized buckets in expectation.
- But now the space we need is  $\log(|H|)$  bits.
  - Hopefully pretty small!



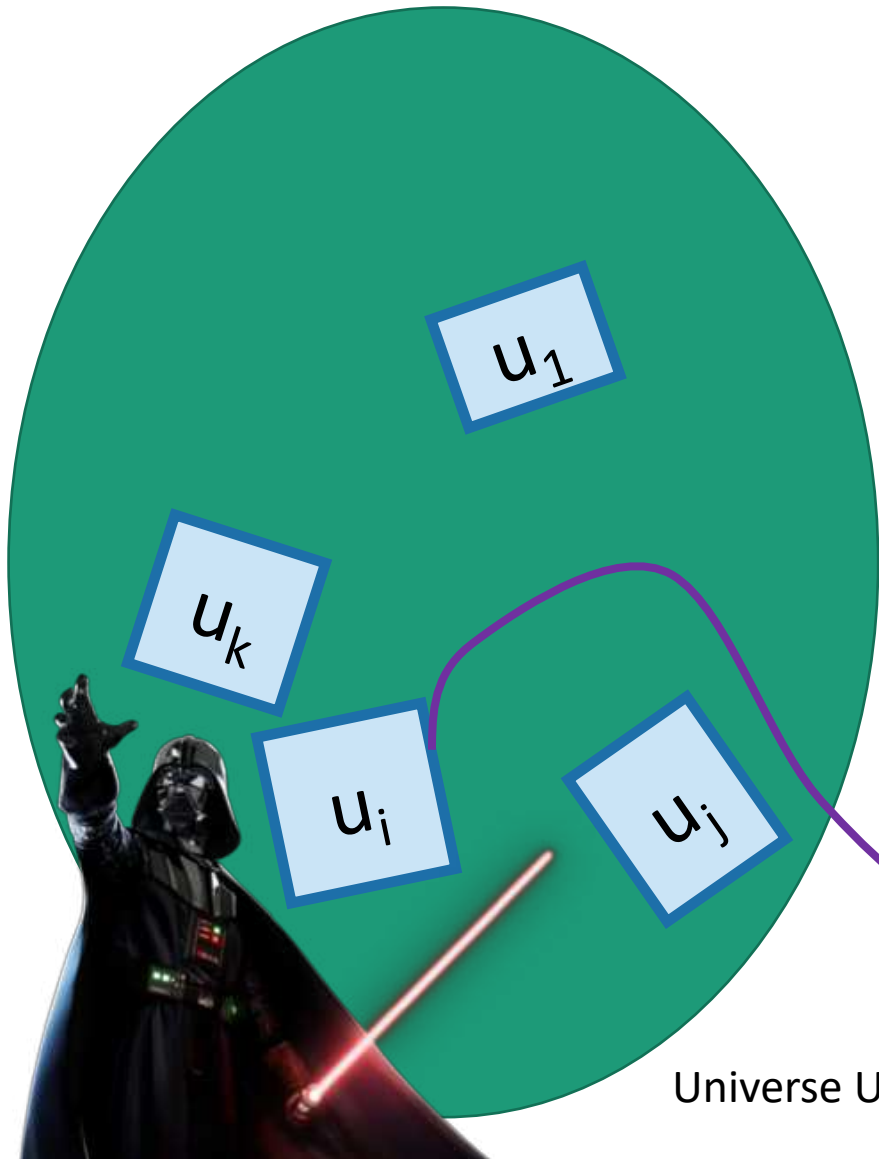


# So the whole scheme will be

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

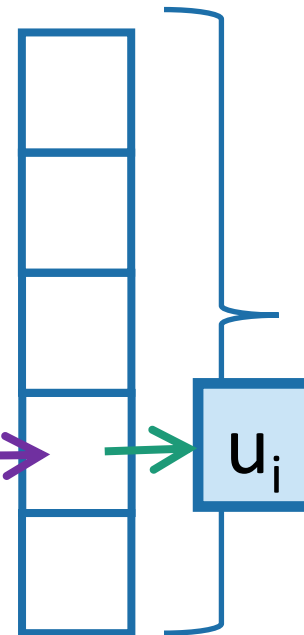


We can store  $h$  in small space  
since  $H$  is so small.



Universe  $U$

$h$



n buckets

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  $H$ ,

# Example

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

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

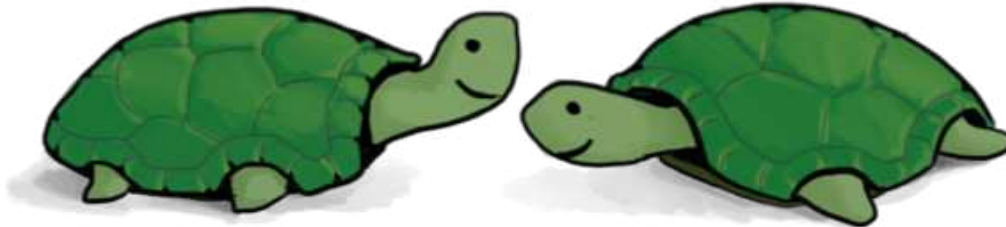
- Pick a small hash family  $H$ , so that when I choose  $h$  randomly from  $H$ ,

# Non-example

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

- $h_0 = \text{Most\_significant\_digit}$
- $h_1 = \text{Least\_significant\_digit}$
- $H = \{h_0, h_1\}$

Prove that this choice of  $H$  is  
NOT a universal hash family!



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

# Non-example

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

- $h_0 = \text{Most\_significant\_digit}$
- $h_1 = \text{Least\_significant\_digit}$
- $H = \{h_0, h_1\}$

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

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

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

- Define:

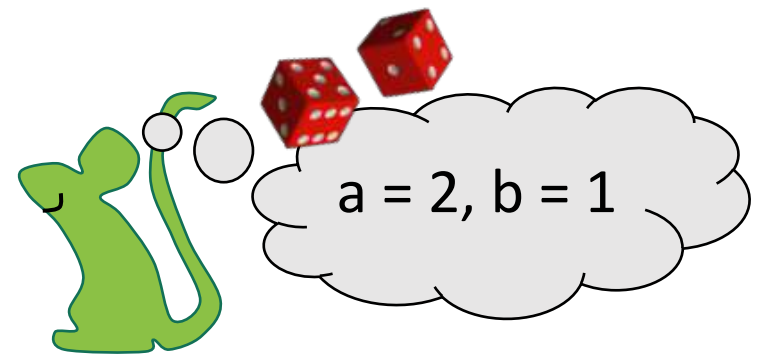
$$H = \{ h_{a,b}(x) : a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \}$$

- Claim:

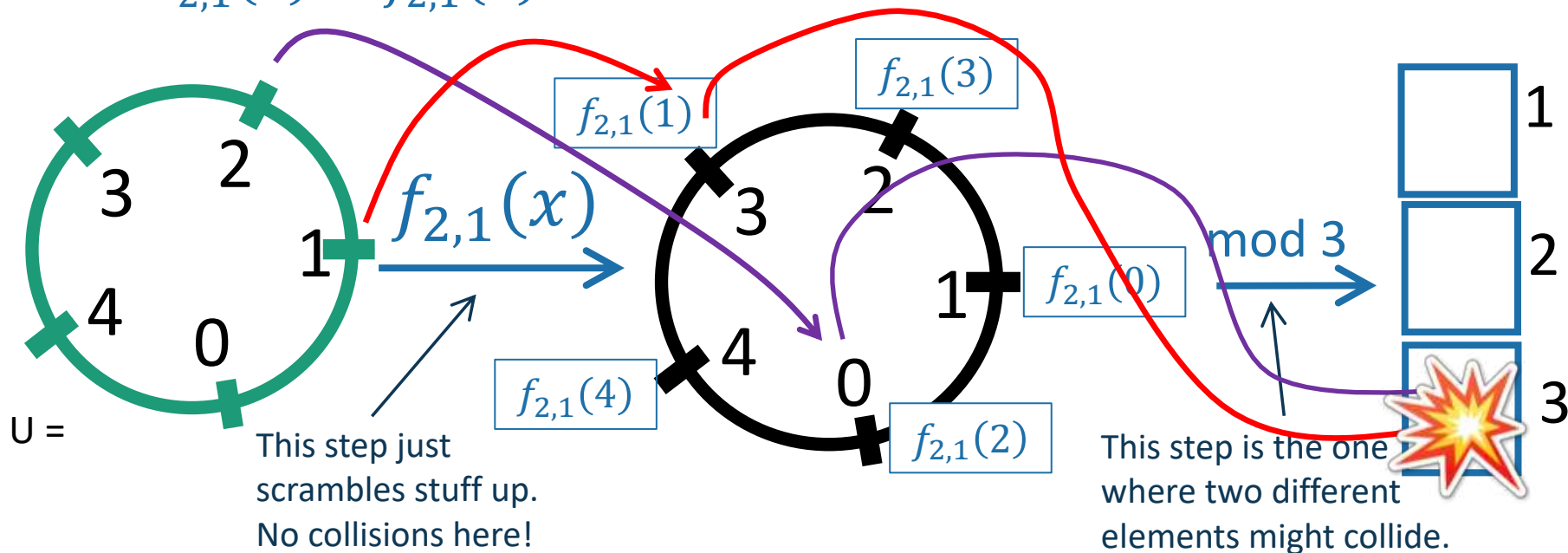
$H$  is a universal hash family.



# Say what?

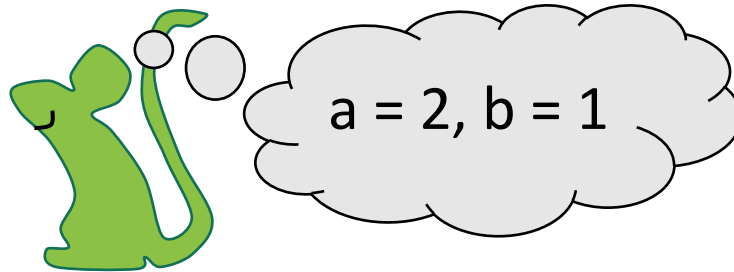


- Example:  $M = p = 5, n = 3$
- To draw  $h$  from  $H$ :
  - Pick a random  $a$  in  $\{1, \dots, 4\}$ ,  $b$  in  $\{0, \dots, 4\}$
- As per the definition:
  - $f_{2,1}(x) = 2x + 1 \mod 5$
  - $h_{2,1}(x) = f_{2,1}(x) \mod 3$



# Ignoring why this is a good idea

- Can we store  $h$  with small space?



- Just need to store two numbers:
  - $a$  is in  $\{1, \dots, p-1\}$
  - $b$  is in  $\{0, \dots, p-1\}$
  - So about  $2\log(p)$  bits
  - By our choice of  $p$ , that's  $O(\log(M))$  bits.

Compare: direct addressing was  $M$  bits!

Twitter example:  $\log(M) = 140 \log(128) = \mathbf{980}$  vs  $M = \mathbf{128^{140}}$



Another way to see that

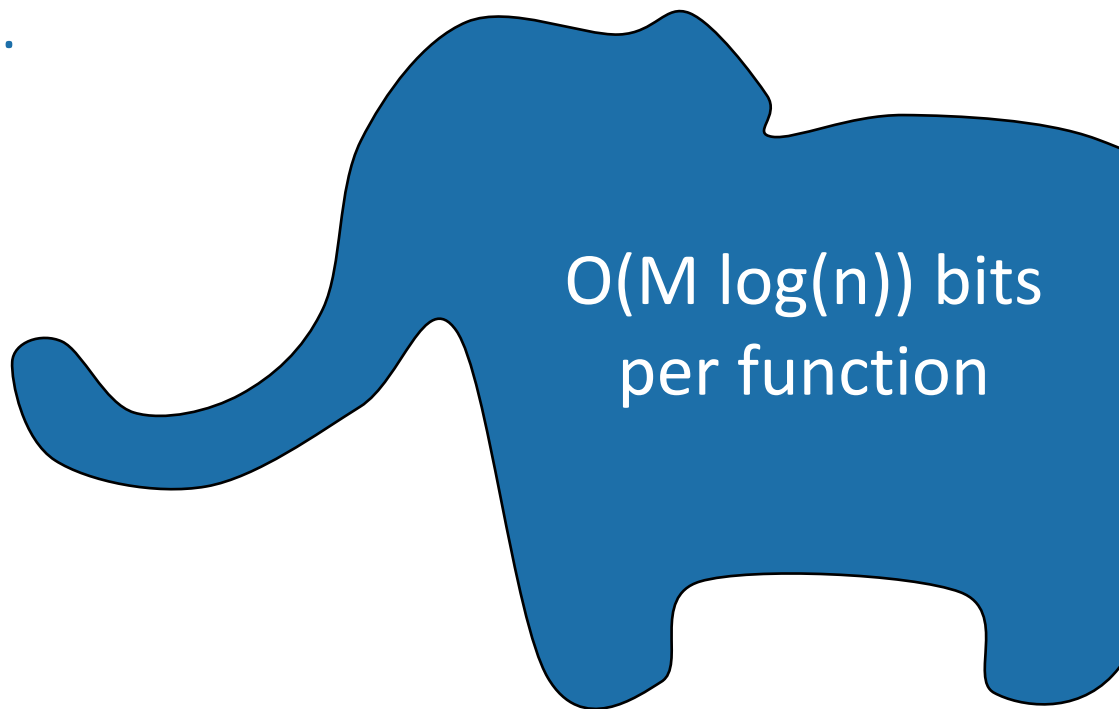
# We only need $\log(M)$ bits to store $h$

- We have  $p-1$  choices for  $a$ , and  $p$  choices for  $b$ .
  - So  $|H| = p(p-1) = O(M^2)$
- Space needed to store an element  $h$ :
  - $\log(M^2) = O(\log(M))$ .

$O(\log(M))$  bits  
per function



$O(M \log(n))$  bits  
per function



# Why does this work?

- This is actually a little complicated.
  - See CLRS (Thm 11.5) 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.

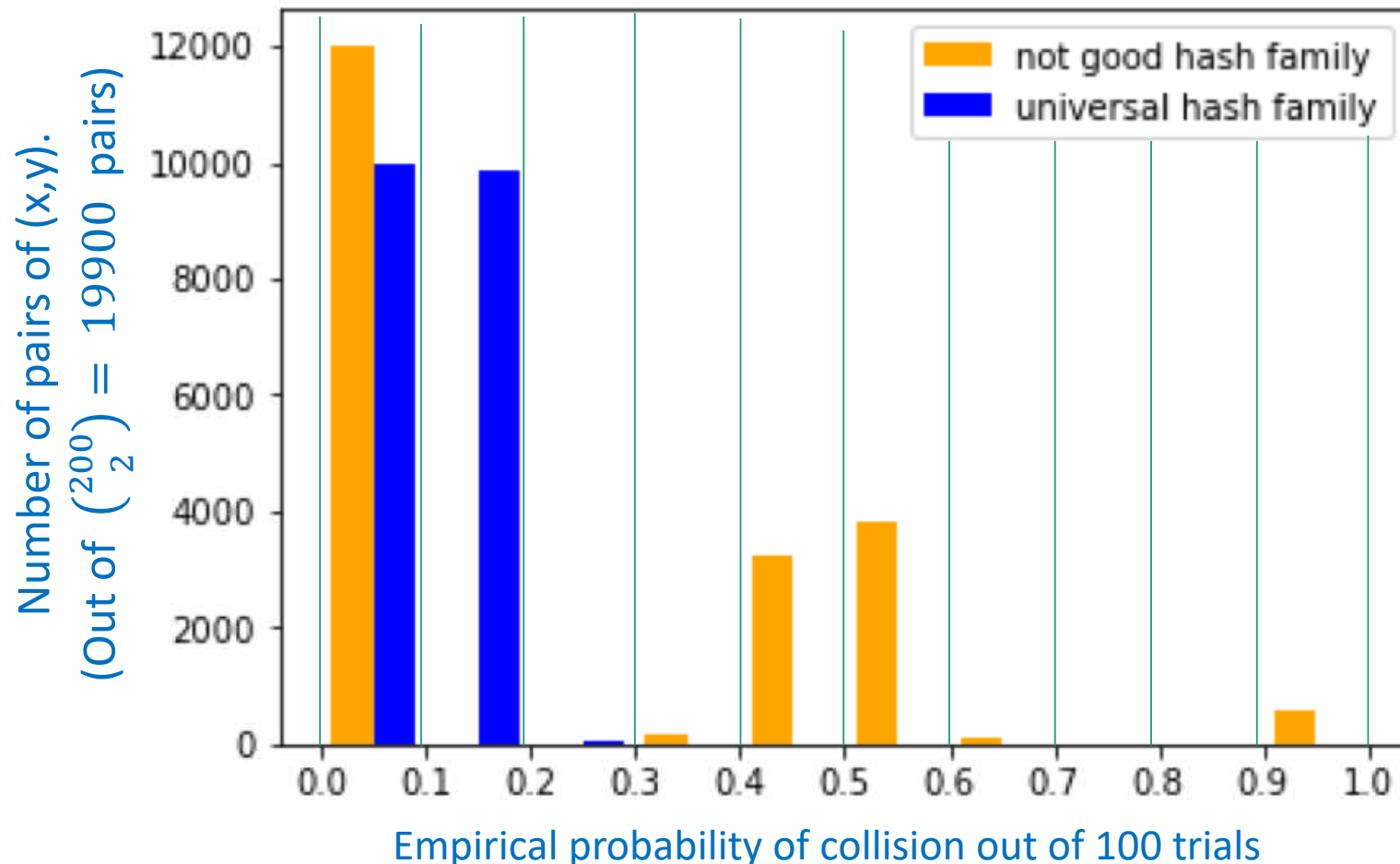
Try to prove that this is a  
universal hash family!



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

- Check out the IPython notebook for lecture 8

M=200, n=10

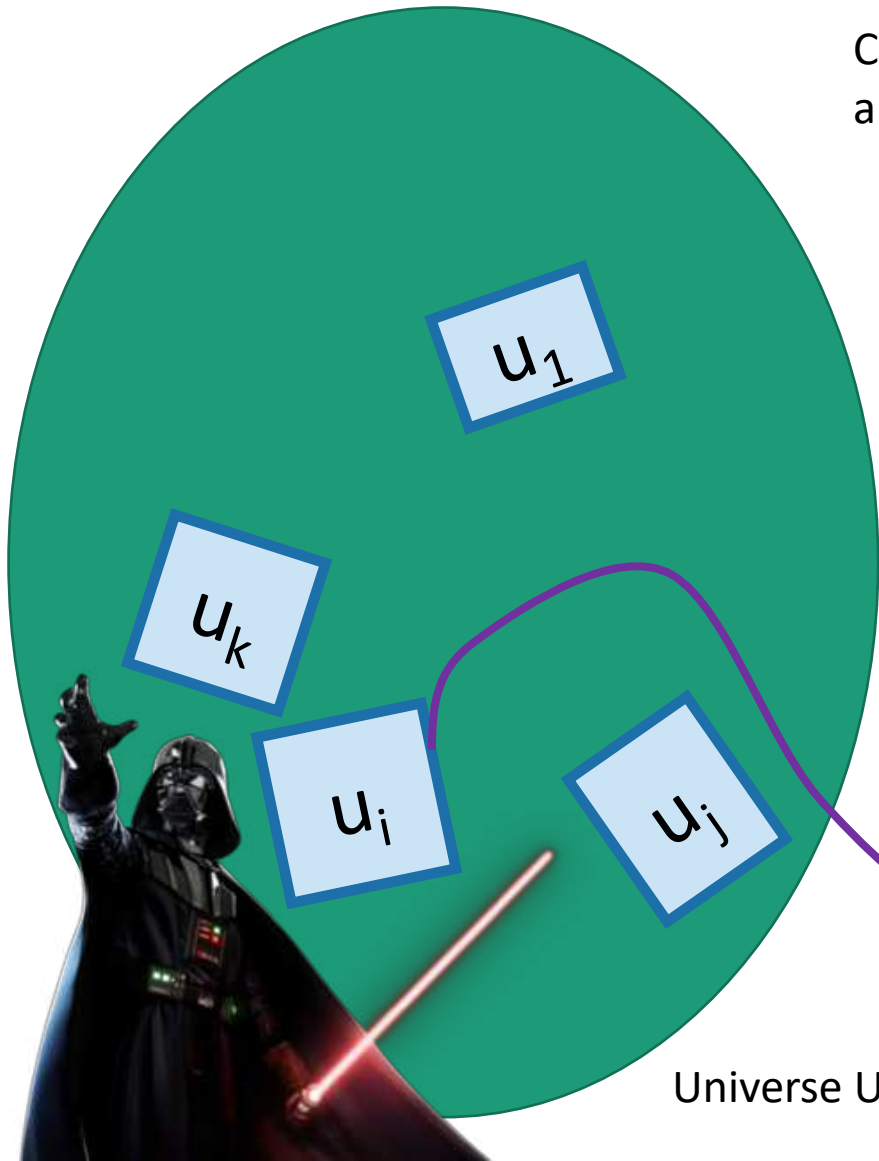


# So the whole scheme will be

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

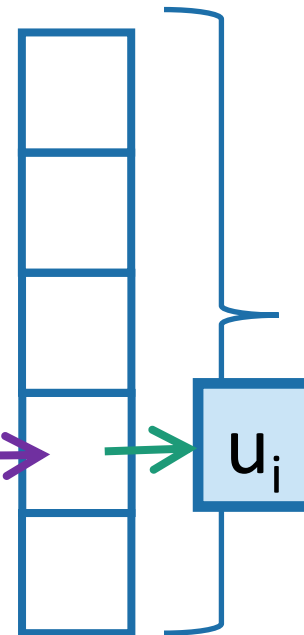


We can store  $h$  in space  
 $O(\log(M))$  since we just need  
to store  $a$  and  $b$ .



Universe U

$h_{a,b}$



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 $O(1)$

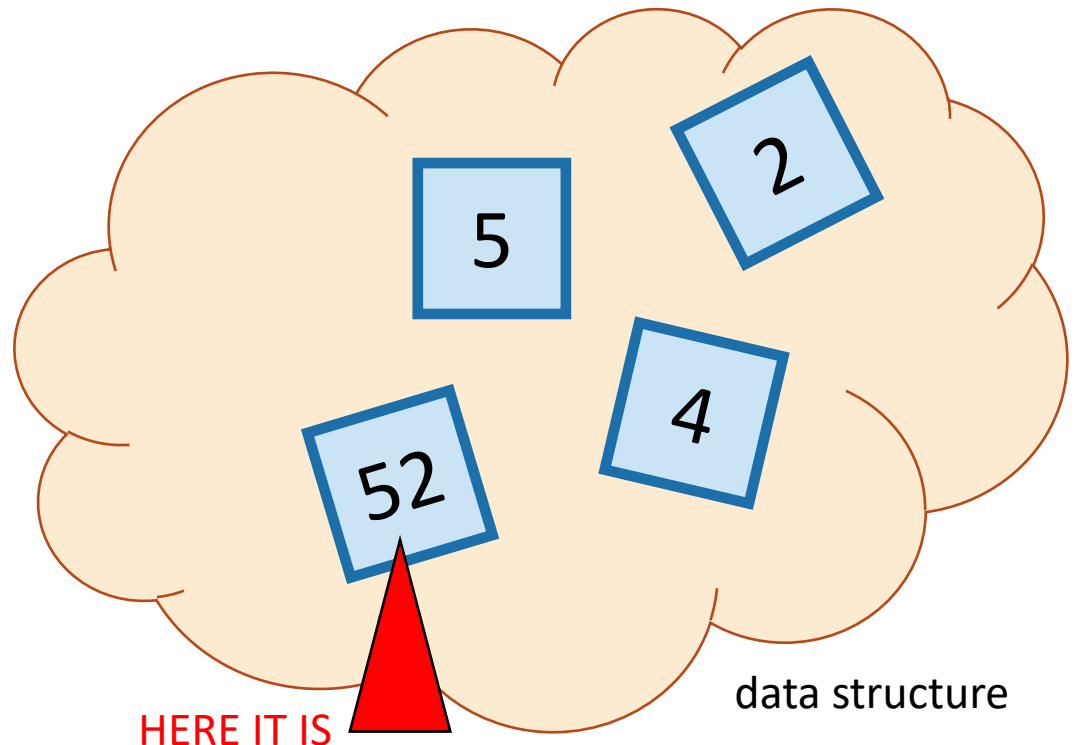
## INSERT/DELETE/SEARCH

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

- INSERT 5

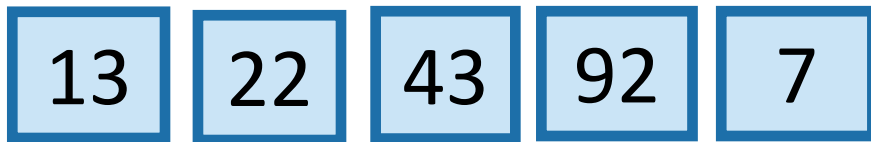
- DELETE 4

- SEARCH 52

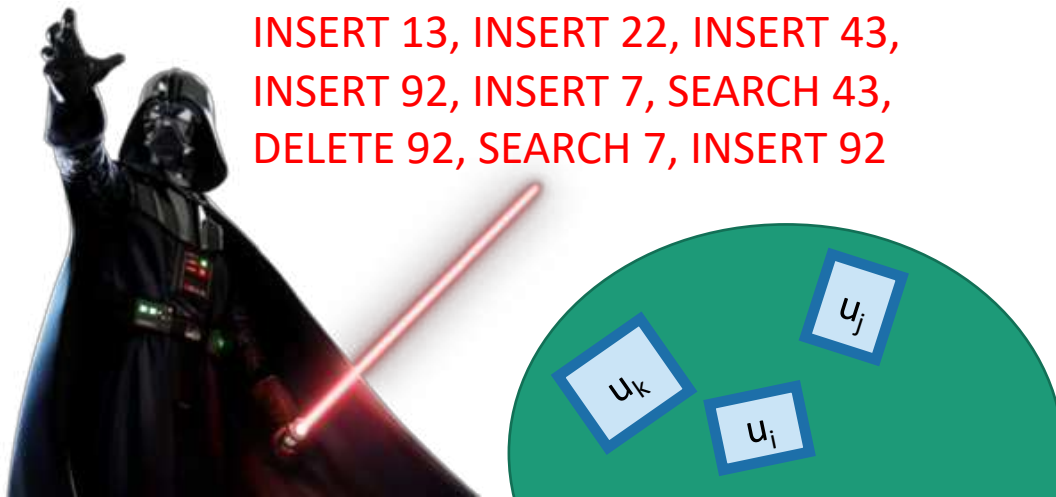


# We studied this game

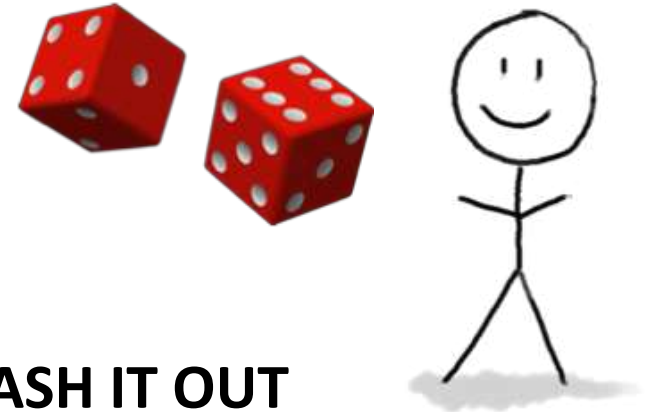
1. An adversary chooses any  $n$  items  $u_1, u_2, \dots, u_n \in U$ , and any sequence of  $L$  INSERT/DELETE/SEARCH operations on those items.



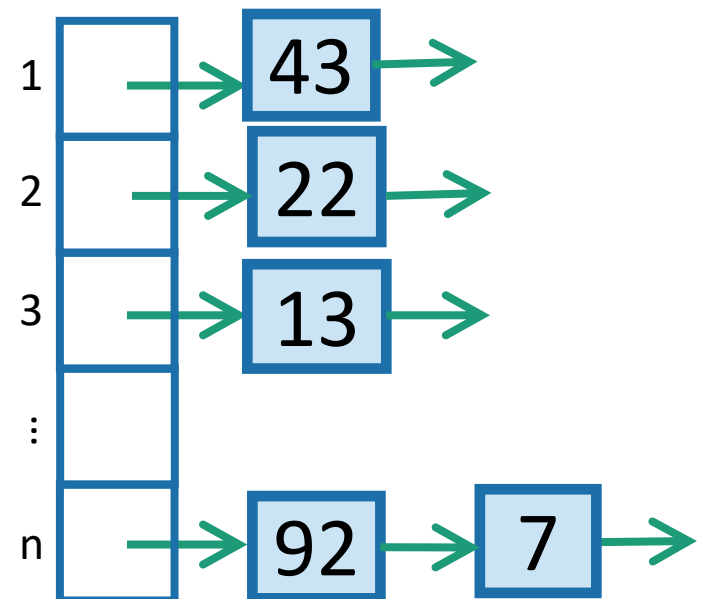
INSERT 13, INSERT 22, INSERT 43,  
INSERT 92, INSERT 7, SEARCH 43,  
DELETE 92, SEARCH 7, INSERT 92



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



## 3. HASH IT OUT



# Uniformly random h was good

- If we choose h uniformly at random,

for all  $u_i, u_j \in U$  with  $u_i \neq u_j$ ,

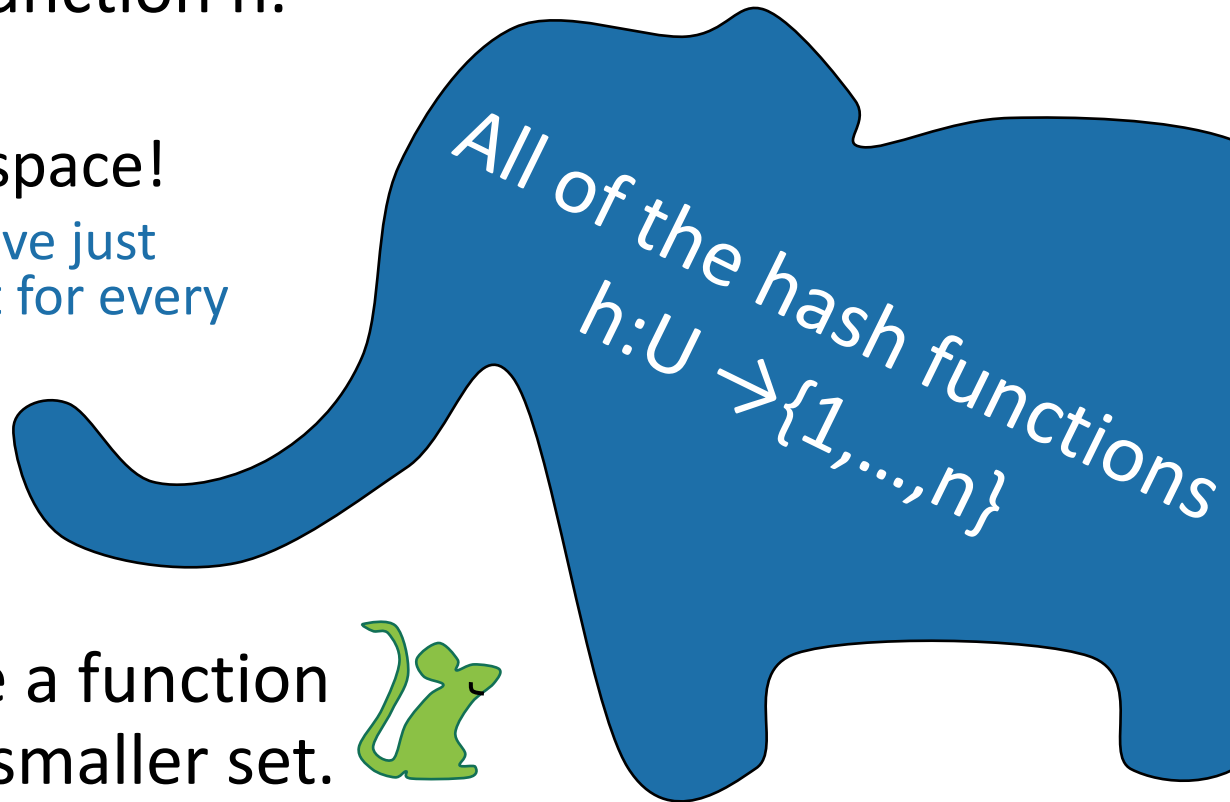
$$P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

- 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



We needed a **smaller set**  
that still has this property

- If we choose  $h$  uniformly at random in  $H$ ,  
for all  $u_i, u_j \in U$  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 call any set with that property a  
**universal hash family.**
- We gave an example of a really small one 😊

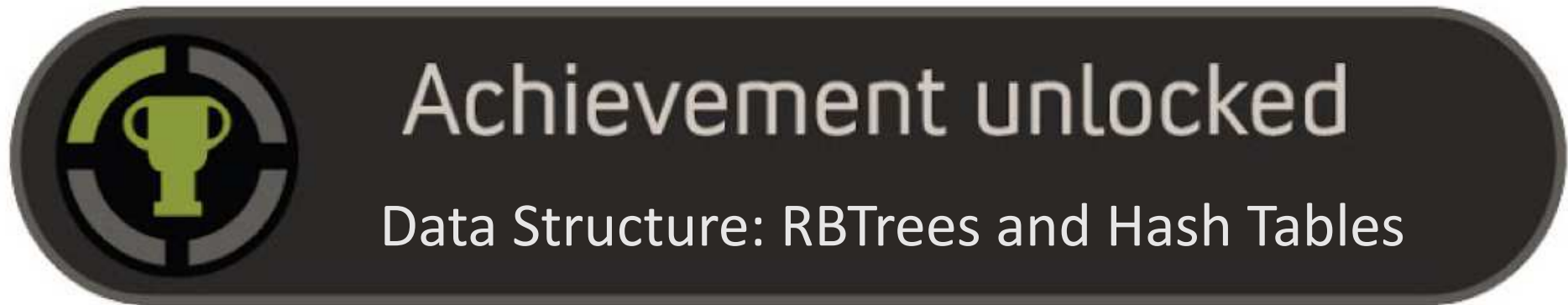


Hashing a universe of size  $M$  into  $n$  buckets, where at most  $n$  of the items in  $M$  ever show up.

## 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](#)

