Lecture 8

Hashing

Announcements

Announcements

- Great job on the first midterm!

- Max: 100
- Median: 63
- Std. Dev: 16

Note: we will curve grades up at the end of the quarter! Don’t worry about a low numerical score!

The midterm was on the long end

for 60 minutes

- Again, the curve will take care of low numerical scores, but…
- **We will try to make the next one a bit shorter!**

If you didn't do as well on the midterm as you wanted…

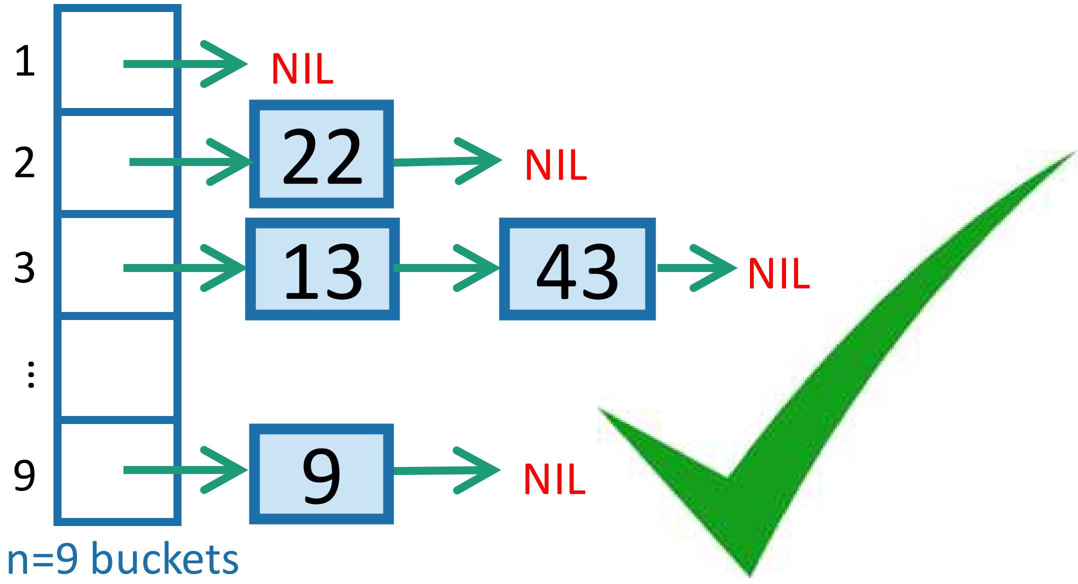
- That's okay! There's lots of time to come back.
- My suggestion: **Practice problems!**
  - Look in the book (Algorithms Illuminated), and in CLRS
  - Recommended exercises on the slides
  - By popular demand, we will be putting more practice problems on section material
  - Make sure you really understand the HW, and try to do it yourself *before* going to office hours, relying on your HW team, etc..
- If there's a concept you are struggling with, come to office hours and ask! We're here to help!

Announcements

- Embedded EthiCS Lecture 1 is posted!
  - Find it on Ed/Canvas!
  - You are responsible for this material for future HW/Exams.

- HW3 due Friday ☺

Today: hashing



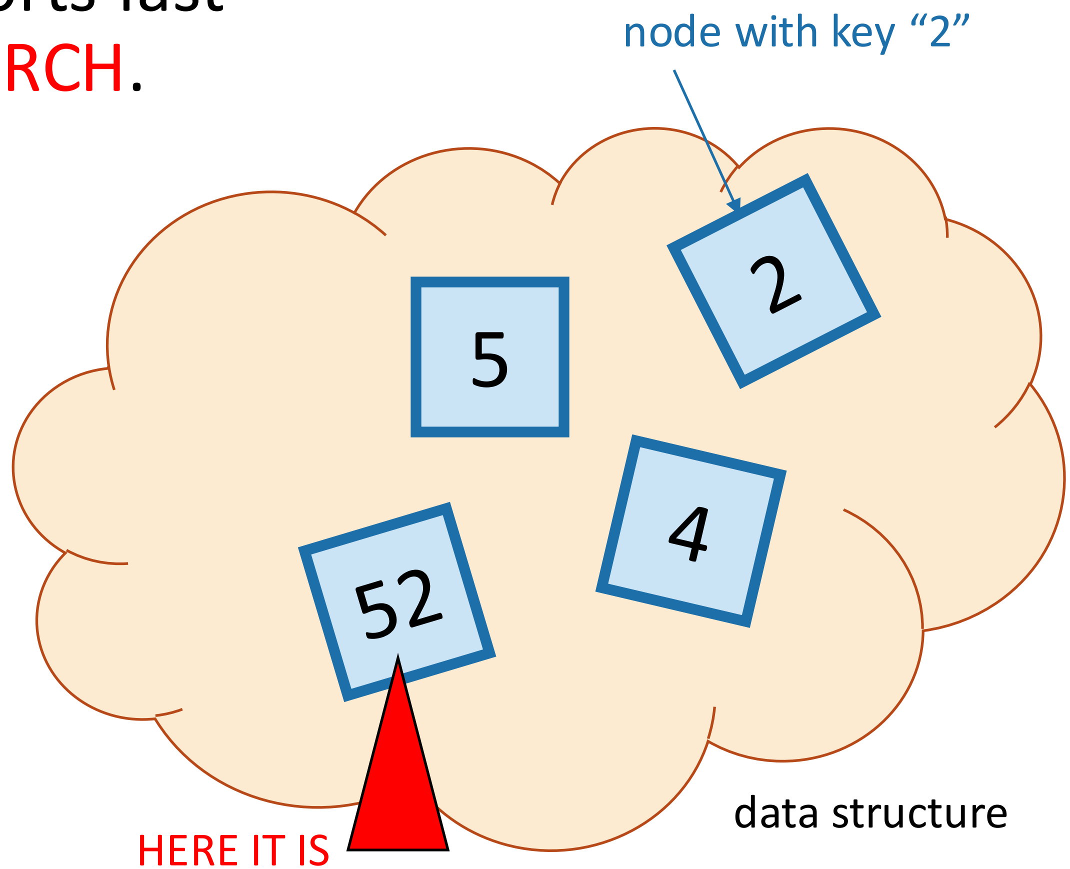
Outline

- **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

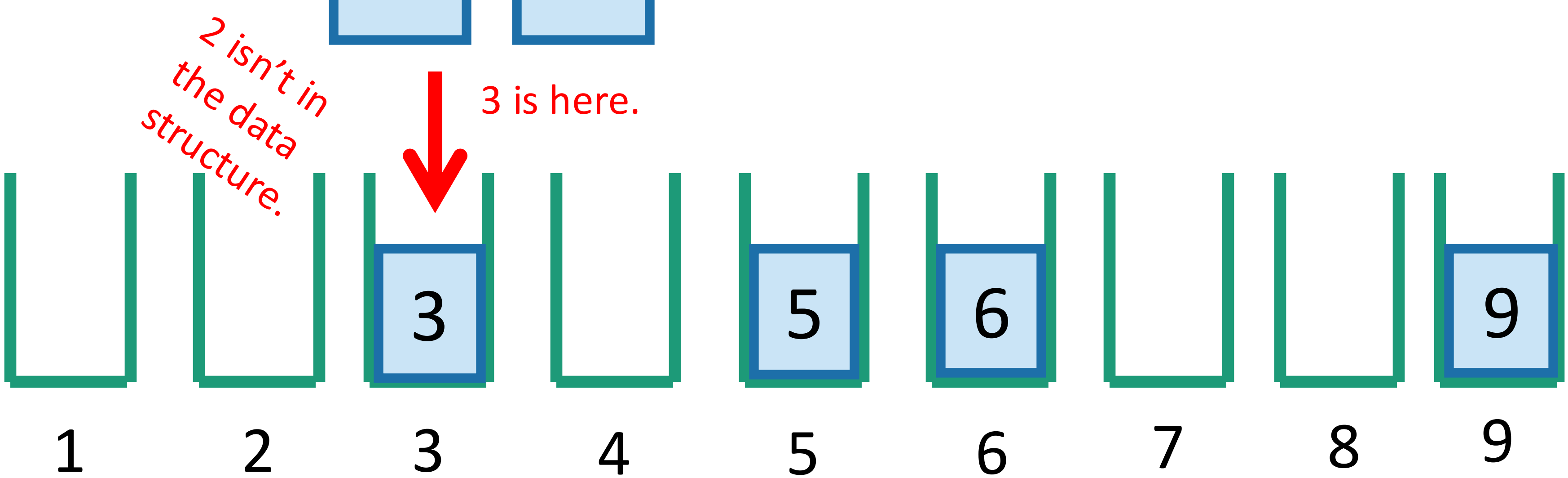
eg, Python's `dict`, Java's `HashSet/HashMap`, C++'s `unordered_map`

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

One way to get O(1) time

This is called "direct addressing"

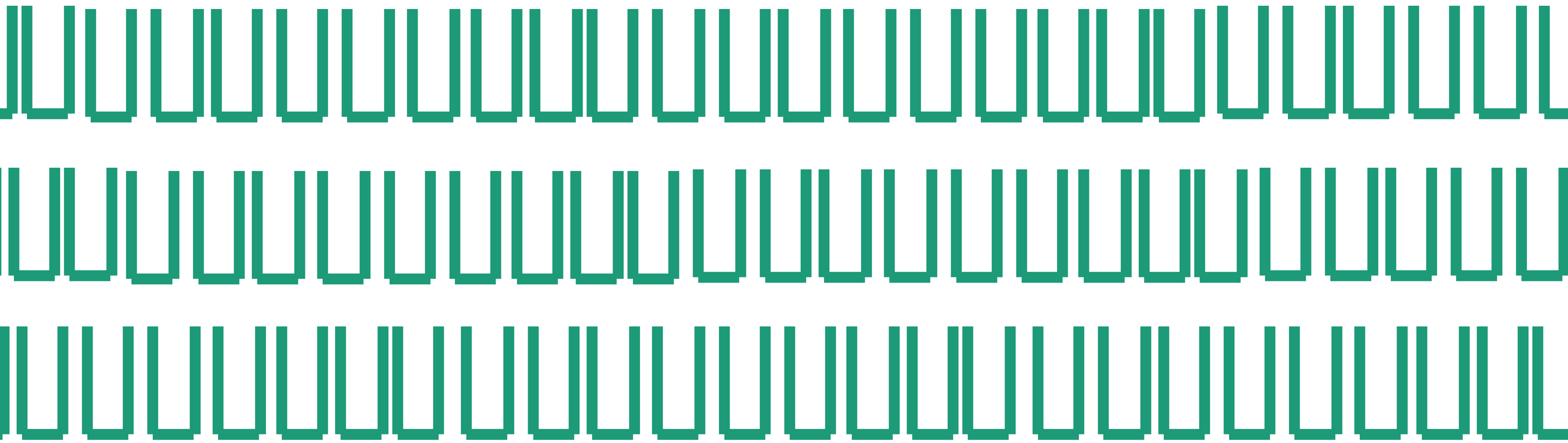
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5
- DELETE: 6
- SEARCH: 3 2



That should look familiar

The universe is really big!

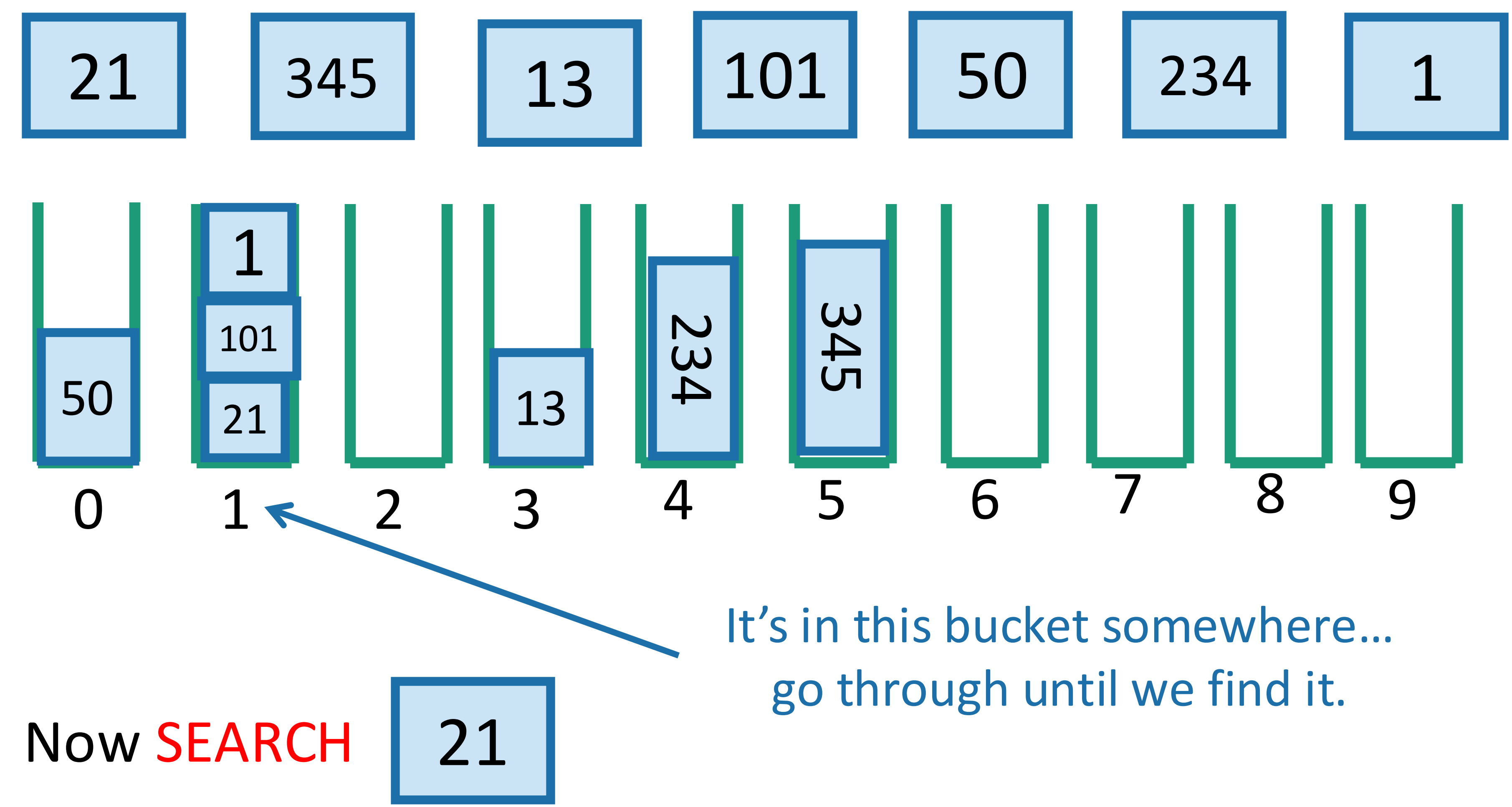
- Kind of like COUNTINGSORT from Lecture 6.
- Same problem: if the keys may come from a "universe" $U = \{1,2, \ldots , 10000000000\}$, direct addressing takes a lot of space.



Solution?

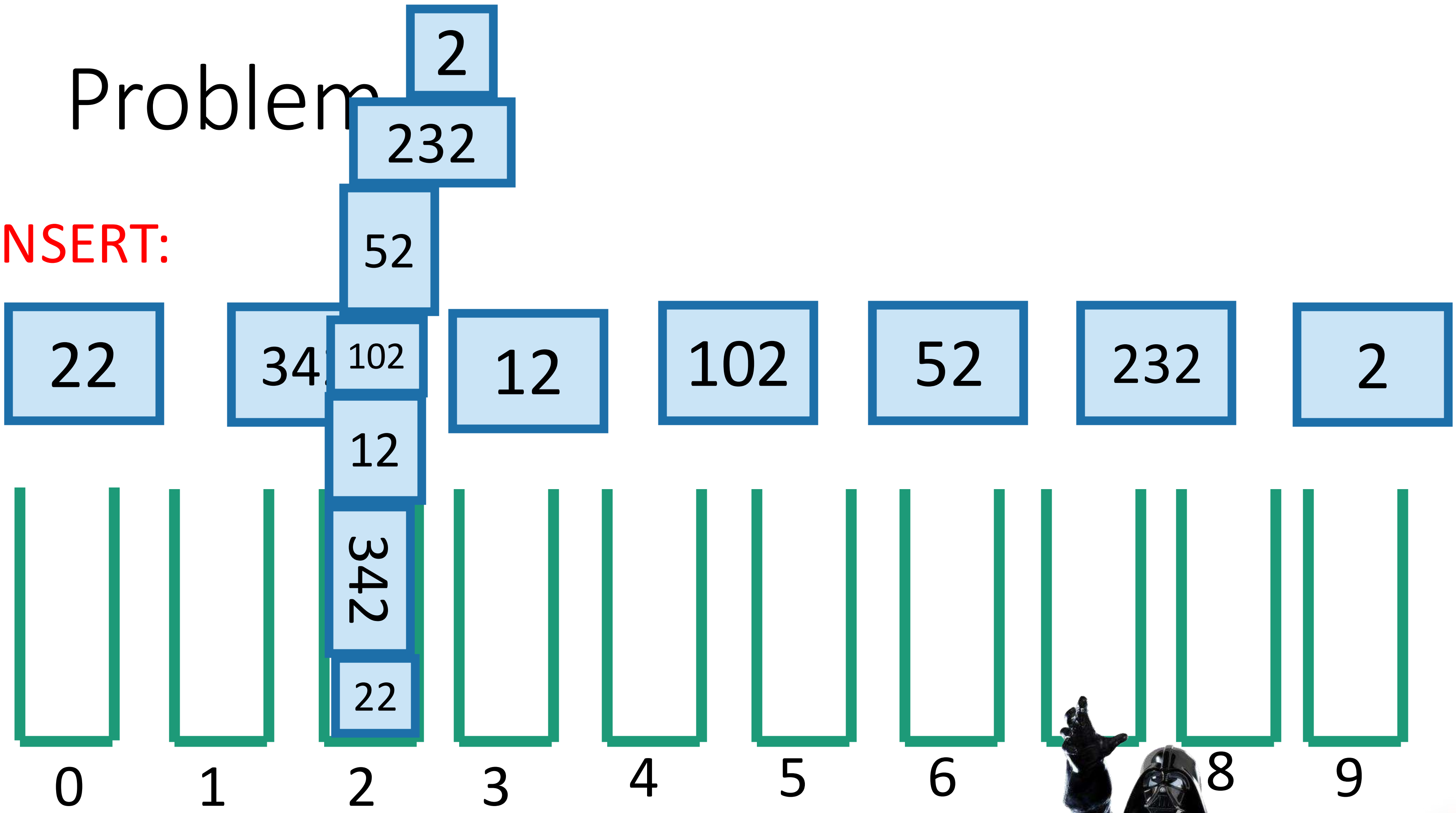
Put things in buckets based on one digit

INSERT:



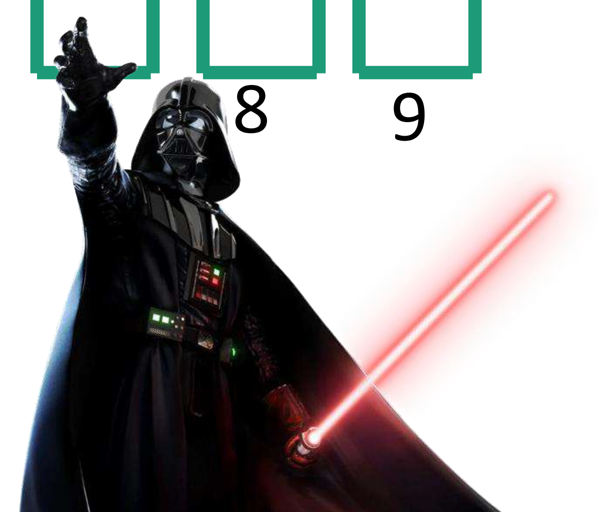
Problem

INSERT:



Now SEARCH 102

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



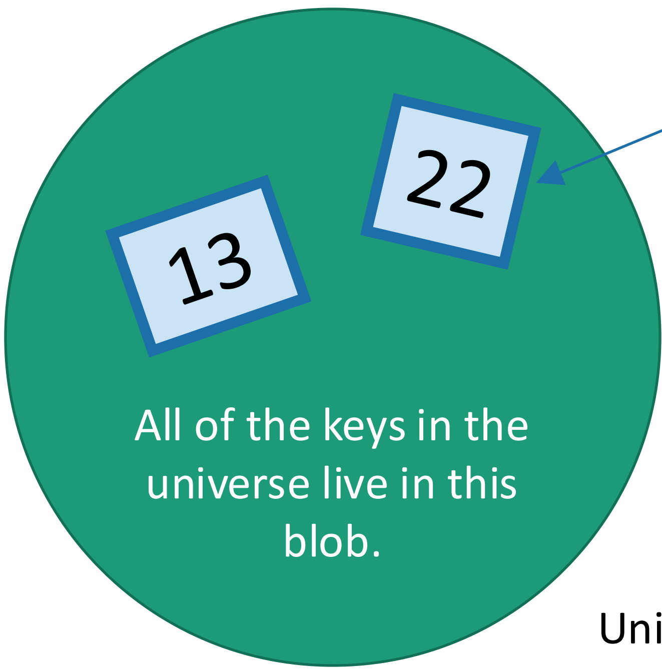
Hash tables

- 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.

Only a n keys will ever show up.



Universe U

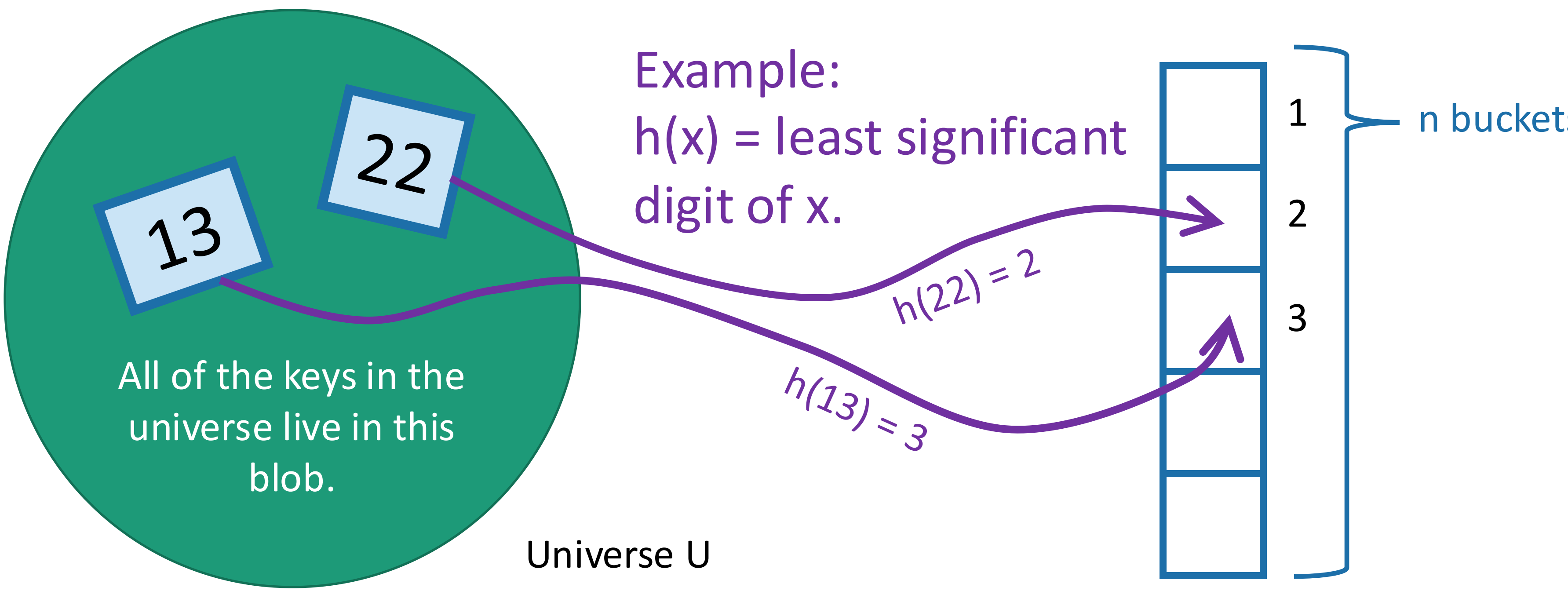
Example: U is the set of all strings of at most 280 ascii characters. (128^{280} of them).

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

Hash Functions

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

- **Note!** For this lecture, n is **both** #buckets and #(things that might show up).
- That doesn't need to be the case, but in general we should think of those two things as being on the same order.



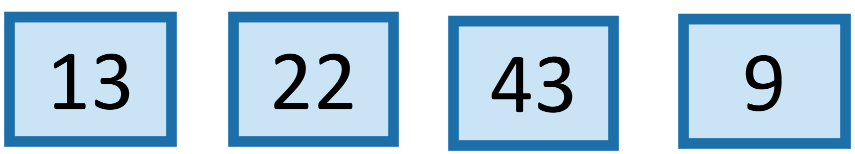
Hash Tables (with chaining)

A **hash table** consists of:

- An 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 $h: U \rightarrow \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

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

INSERT:

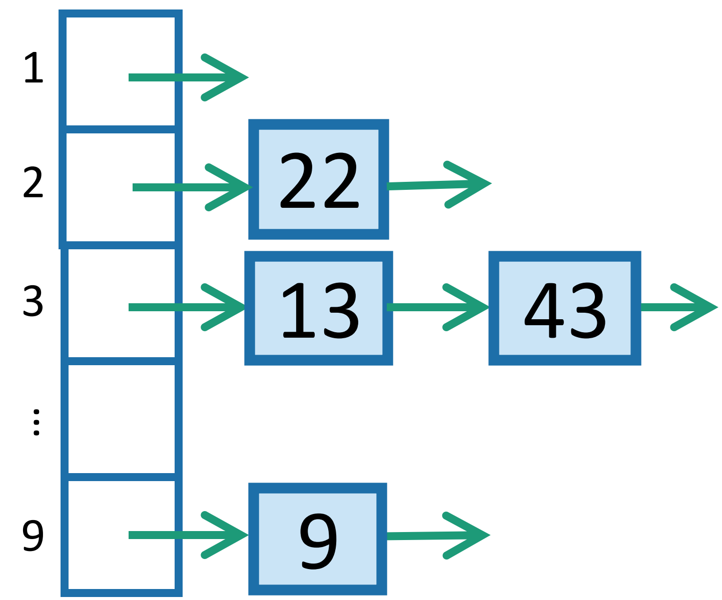


SEARCH 43:

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

DELETE 43:

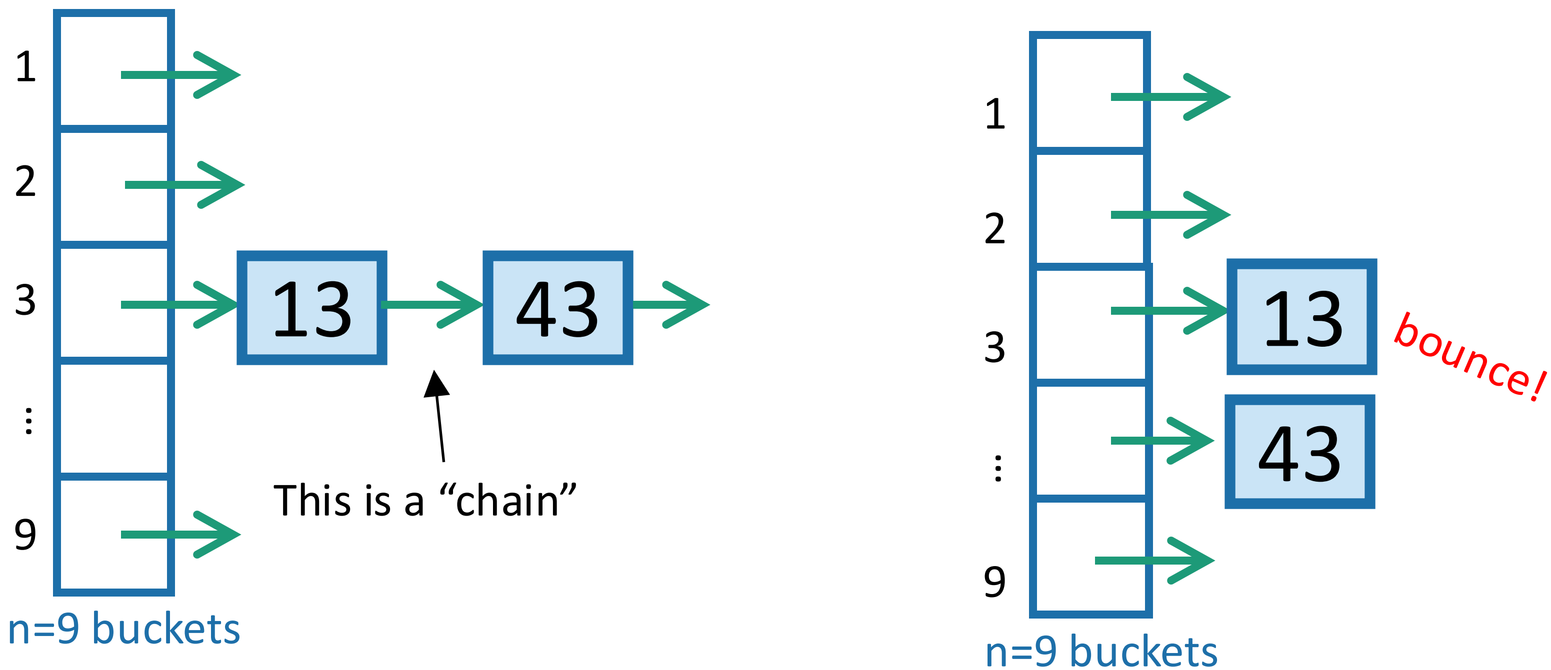
Search for 43 and remove it.



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}

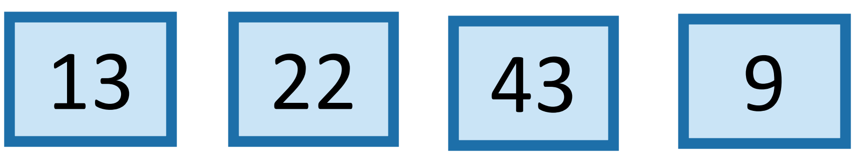
Hash Tables (with chaining)

A **hash table** consists of:

- 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 $h: U \rightarrow \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

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

INSERT:

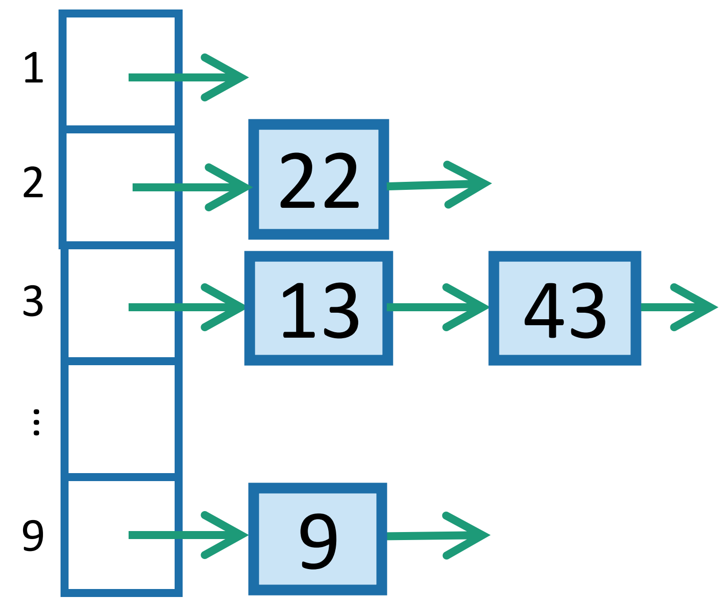


SEARCH 43:

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

DELETE 43:

Search for 43 and remove it.



n buckets (say n=9)

Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - (We still need to figure out how to do the bucketing)

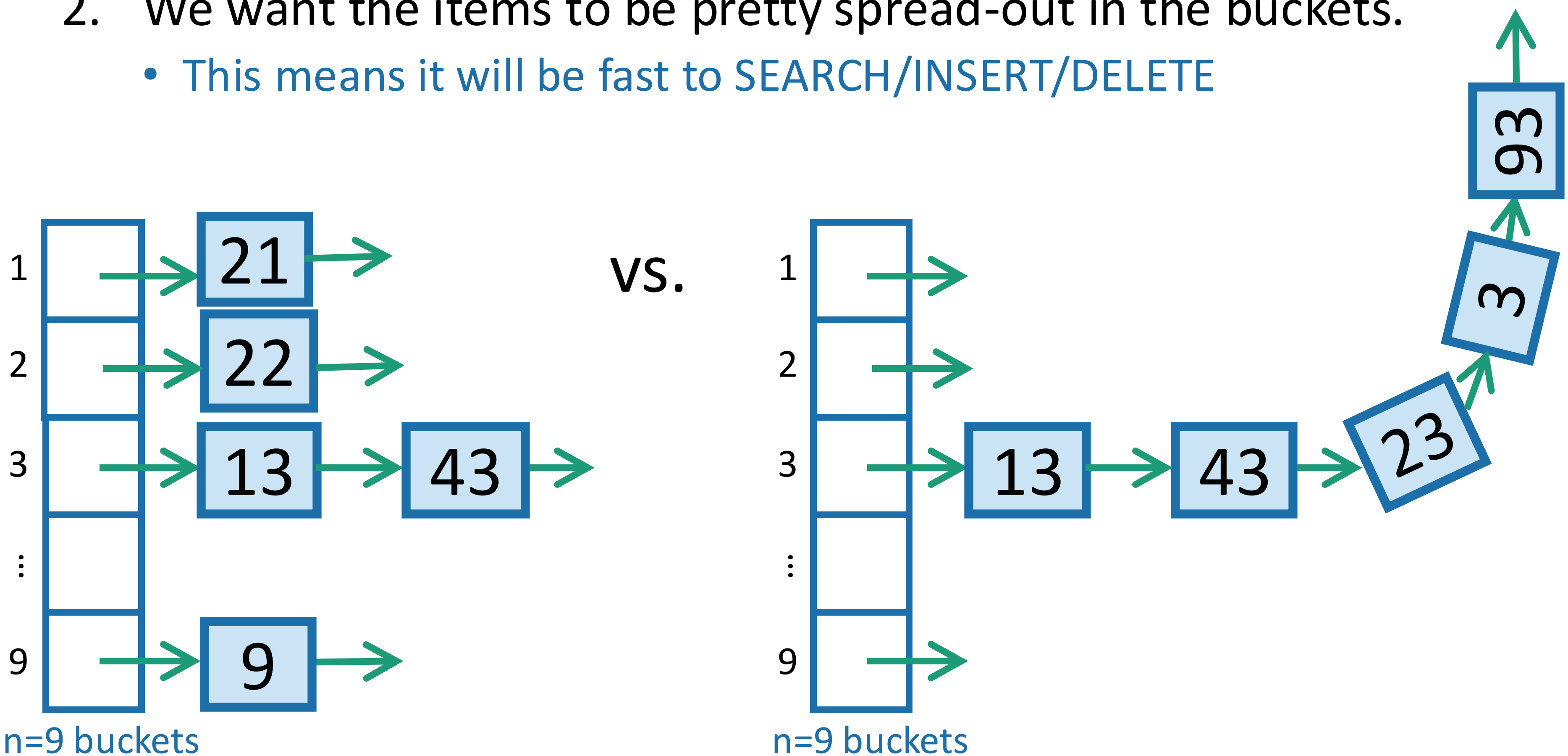
Interlude: motivation for hash families.

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

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

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



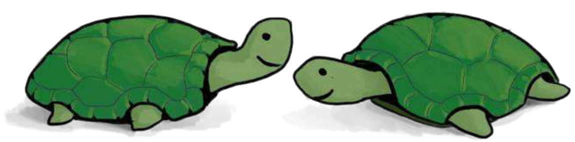
*Assuming it takes time O(1) to evaluate h(u)

Worst-case analysis

- Goal: Design a function $h: U \to \{1, \dots, n\}$ 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.

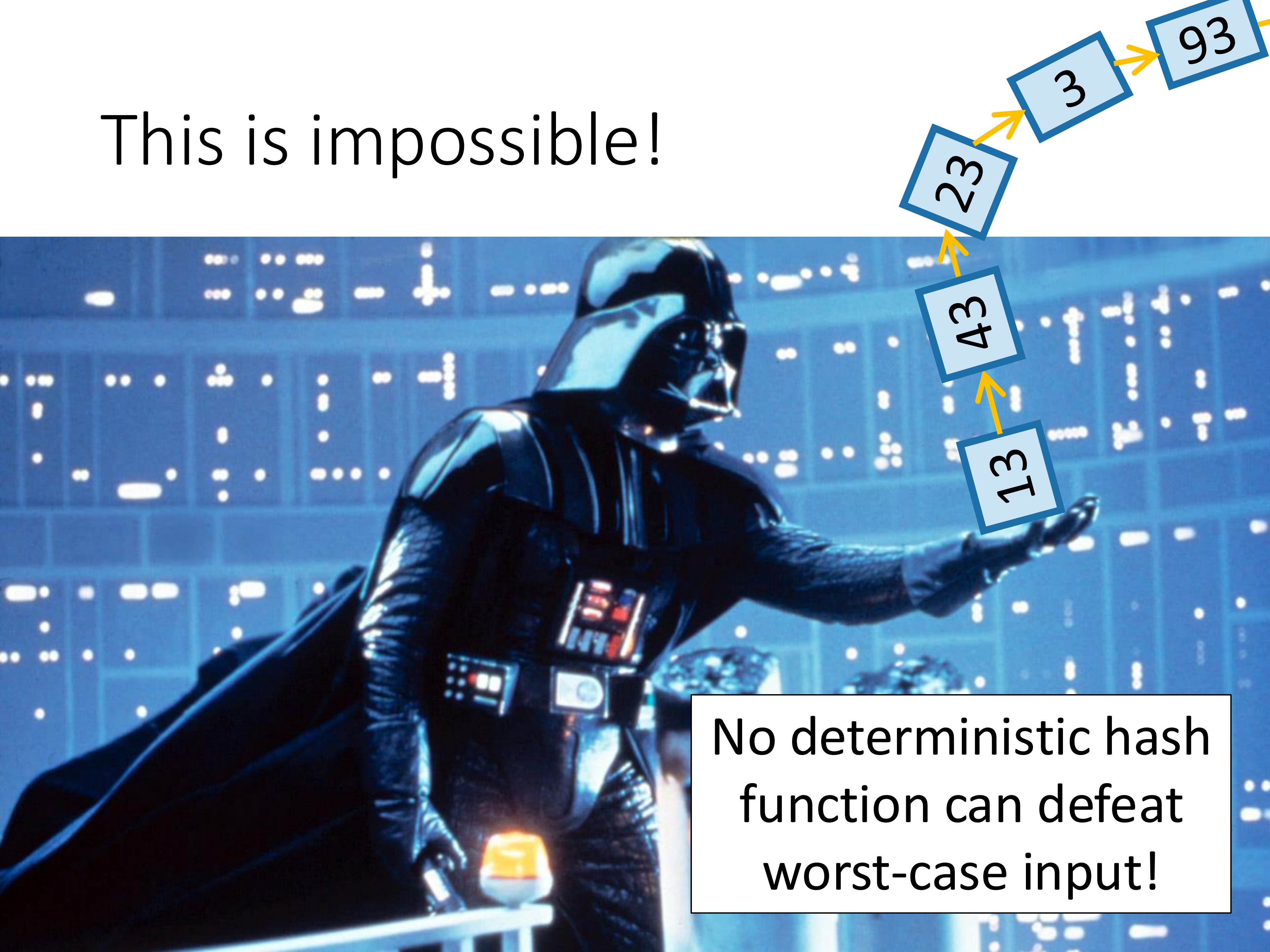
- If we had this*, then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH

Can you come up with such a function?



Think-Pair-Share Terrapins

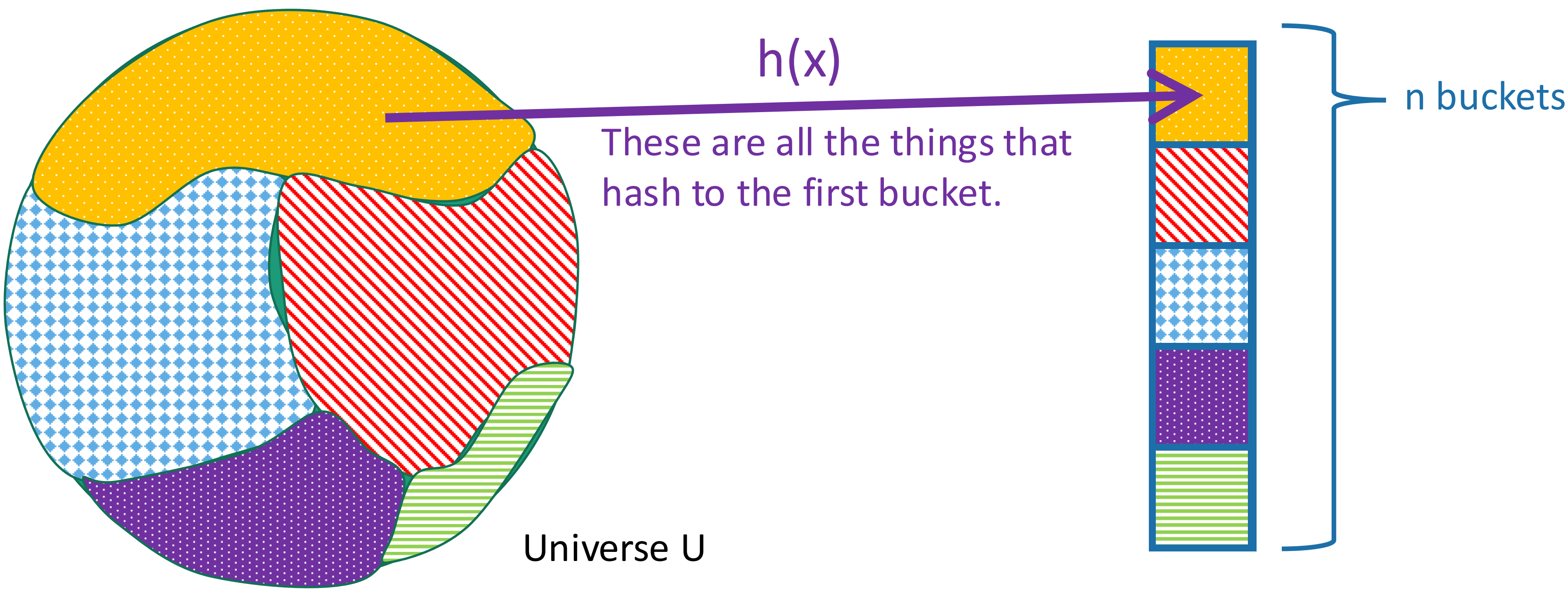
This is impossible!



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 then 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

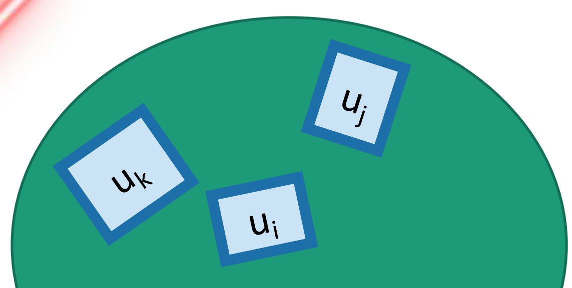
What does **random** mean here? Uniformly random?

Plucky the pedantic penguin

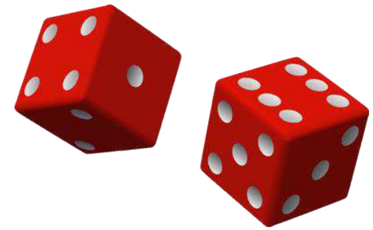
1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

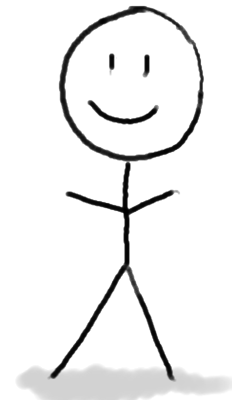
13 22 43 92 7

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

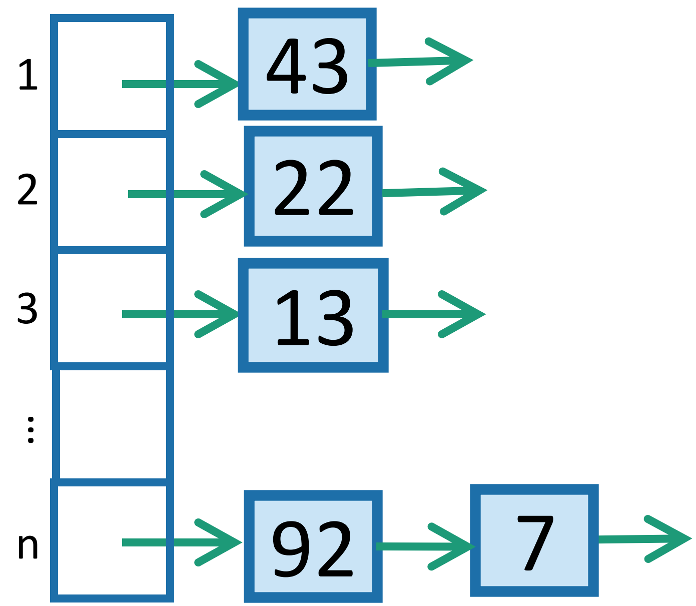


2. You, the algorithm, chooses a **random** hash function $h: U \rightarrow \{1, \ldots, n\}$.



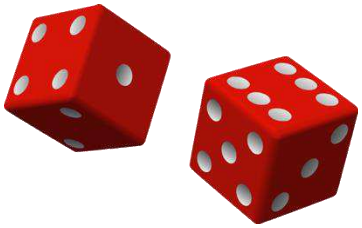


3. **HASH IT OUT** #hashpuns



What is a “random hash function”?

- It’s a function that’s picked using some randomness, but there are lots of ways to do that!
- One way: uniformly random:



All of the M things in the universe

x	$h(x)$
AAAAAA	3
AAAAAB	1
AAAAAC	6
AAAAAD	1
…	
ZZZZZY	7
ZZZZZZ	2

- Now we have a well-defined function $h: U \to \{1,2,\ldots,n\}$

Note: we could have drawn a random function from a different distribution than uniform… we’ll come back to that later.

Why does randomness help?

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

Lucky the
Lackadaisical Lemur

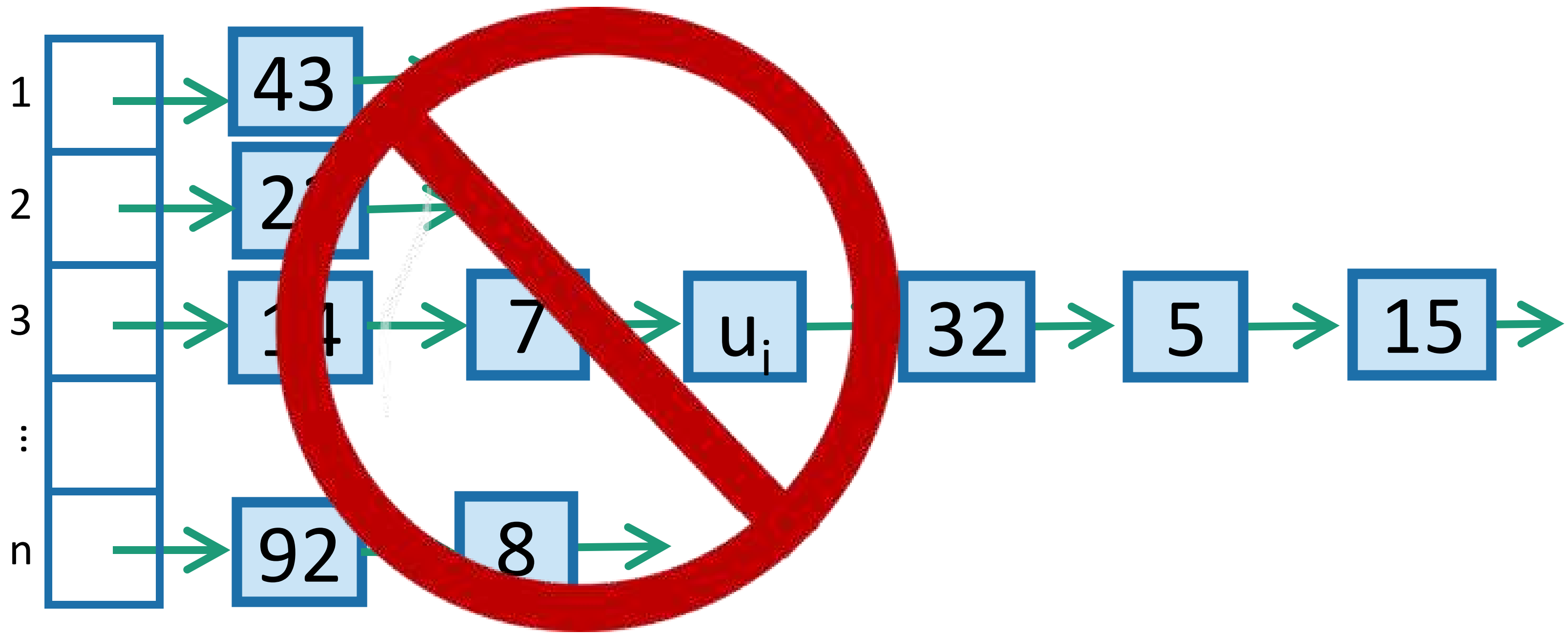
Plucky the Pedantic
Penguin

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

We'll need to do some analysis…

Intuitive goal

It's **bad** if lots of items land in u_i's bucket.
So we want **not that**.



Formal goal

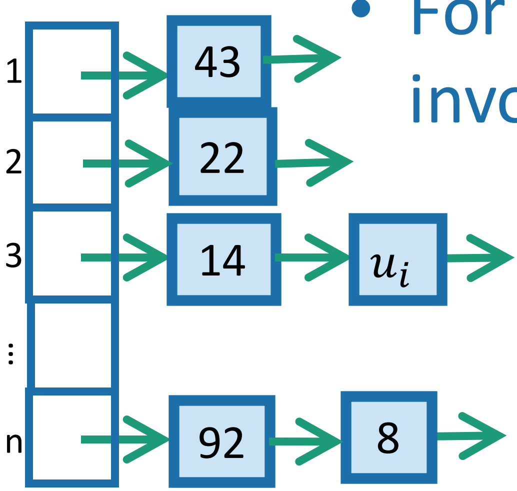
We could replace "2" here with any constant; it would still be good. But "2" will be convenient.

- Let h be a random hash function.
- 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 $i \in \{1, \ldots, n\}$,

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

- 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!

*Assuming h(u) takes O(1) time to compute

Goal:

- Come up with a distribution on hash functions so that:
- For all i=1, …, n,

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

Aside

- For all i=1, …, n,

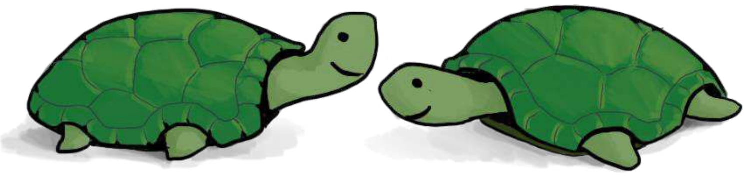
$$E[\text{ number of items in } u_i \text{ 's bucket }] \leq 2.$$

vs

- For all i=1,…,n:

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

Are these the same?



Think-Pair-Share Terrapins

Aside

- For all i=1, …, n,

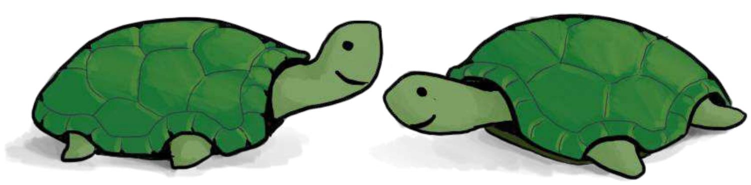
$$E[\text{ number of items in } u_i \text{ 's bucket }] \leq 2.$$

vs

- For all i=1,…,n:

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

Are these the same?



Think-Pair-Share Terrapins

No! (This was your pre-lecture exercise!)

Aside

- For all i=1, …, n,

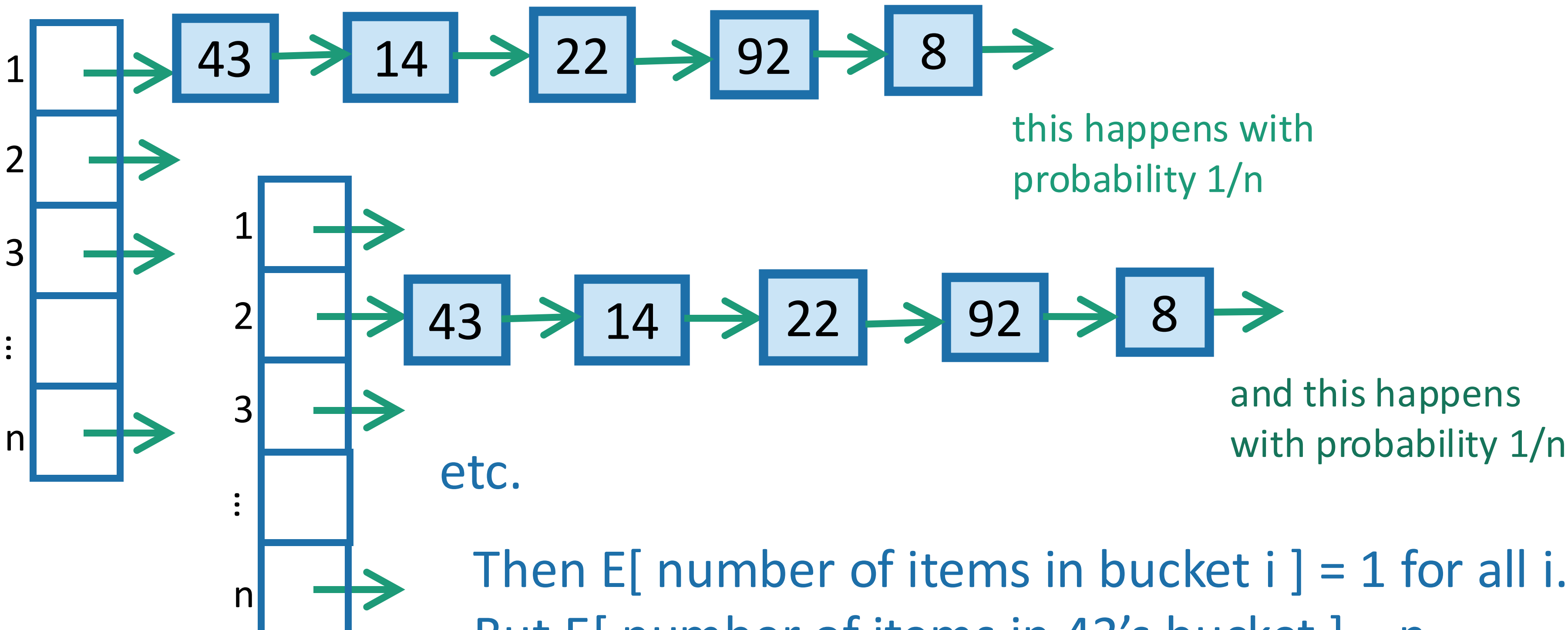
$$E[\text{ number of items in } u_i\text{ ‘s bucket }] \leq 2.$$

vs

- For all i=1,…,n:

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

Suppose that the distribution on h led to:



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

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

Goal:

- Come up with a distribution on hash functions so that:
- For all $i = 1, \dots, n$,

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

Claim:

- The goal is achieved by a uniformly random hash function.

Proof of Claim

- Let h be a uniformly random hash function.
- Then for all $i = 1, \dots, n$,
  E[number of items in u_i's bucket] ≤ 2.

- $E[$ # items in u_i's bucket $] =$
- $= E\left[\sum_{j=1}^{n} \mathbf{1}\{h(u_i) = h(u_j)\}\right]$
- $= \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \le 2.$

You will formally verify this on HW. Intuitively, there are n possibilities where u_j can land, and only one of them is $h(u_i)$.

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 $i \in \{1, \ldots, n\}$,

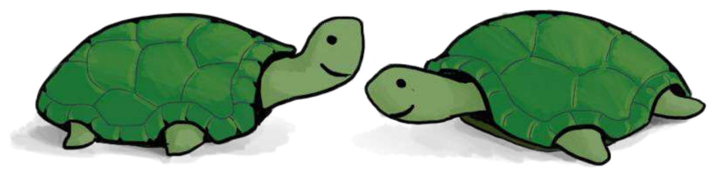
    E[number of items in u_i 's bucket] $\leq$ 2.
- Which implies*:
  - No matter what sequence of operations and items the bad guy chooses,

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

Pick a uniformly random hash function?

*Assuming h(u) takes O(1) time to compute

What’s wrong with this plan?



Think-Pair-Share Terrapins

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

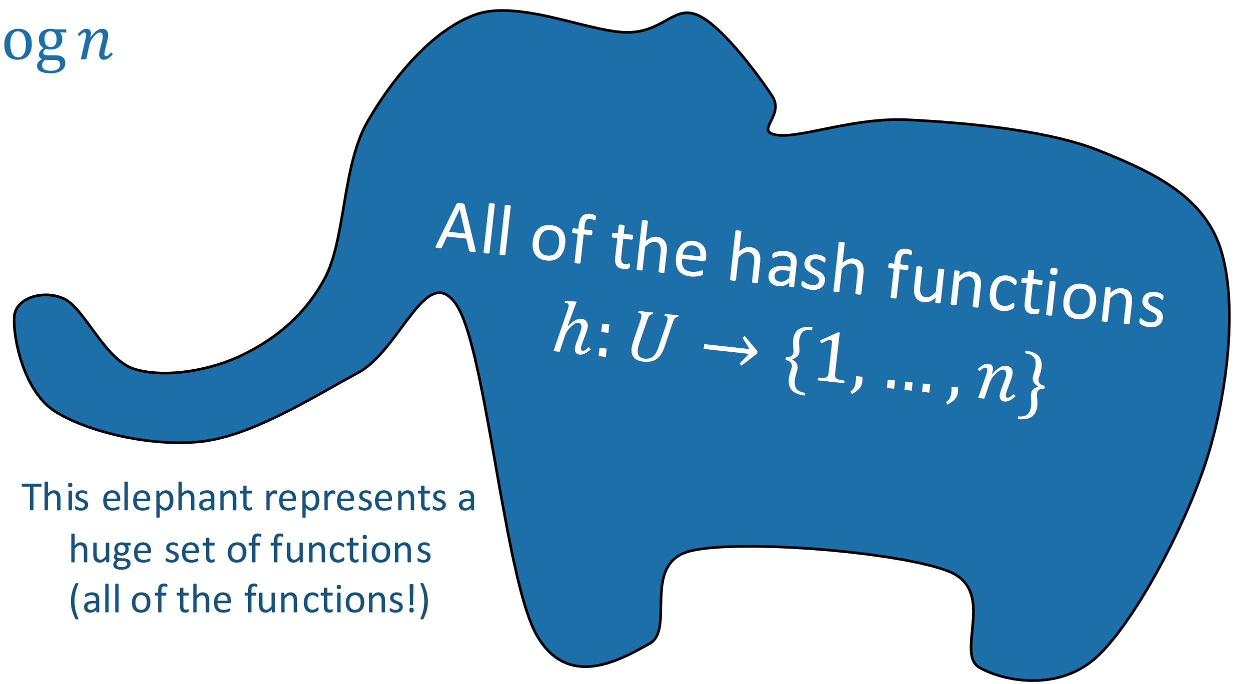
x	$h(x)$
AAAAAA	1
AAAAAB	5
AAAAAC	3
AAAAAD	3
…	
ZZZZZY	7
ZZZZZZ	3

- 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.

The problem is with the **size** of the set of functions we are considering

- **FACT:** It takes *at least* $\log_2 |A|$ bits to represent a random element from a set A (on average).
  - If $A =$"all of the functions from U to $\{1, \ldots, n\}$", then…
  - $|A| = n^M$
  - $\log_2|A| = M \log n$

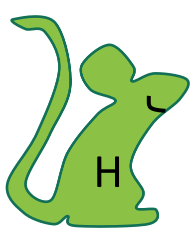
- So it takes at least $M \log n$ bits to represent a random function from this set. ☹



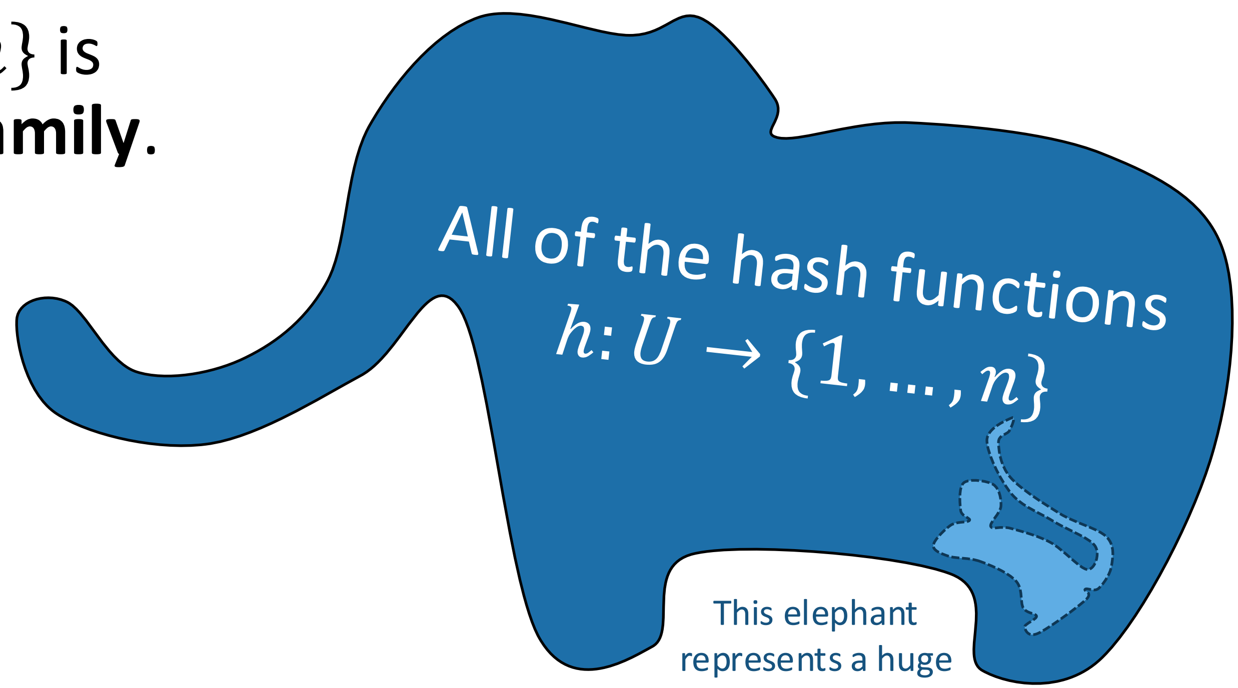
This elephant represents a huge set of functions (all of the functions!)

Solution

- Pick from a smaller set of functions H.
- If H is small, we have a hope of being able to represent $h \in H$ with fewer bits.
- A set H of functions $h: U \to \{1, \dots, n\}$ is called a **hash family**.



This mouse represents a much smaller SUBSET of functions.



This elephant represents a huge SET of functions

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.

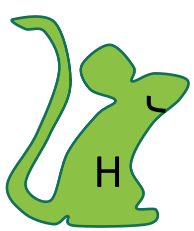
- **Universal hash families** are even more magic.

Hash families

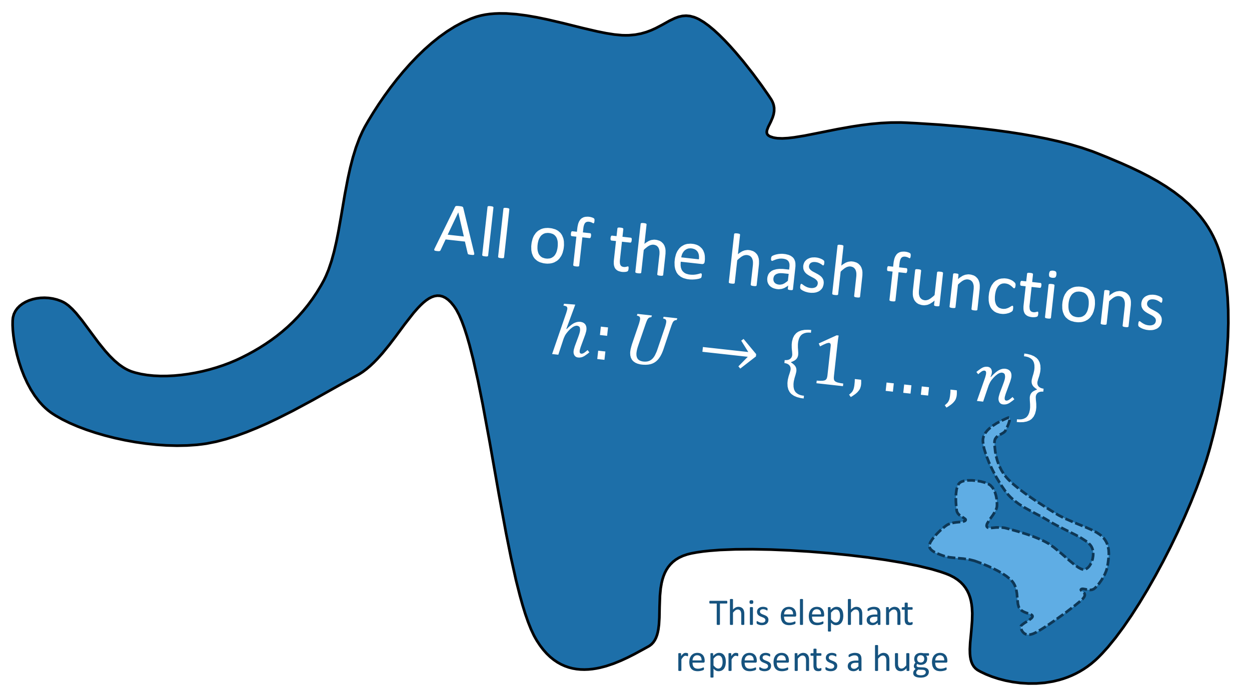
- A **hash family** is a collection of hash functions.

“All of the functions” is a hash family

Any subset of
functions is also
a hash family.



This mouse represents a much
smaller SUBSET of functions.

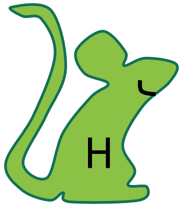


This elephant
represents a huge
SET of functions

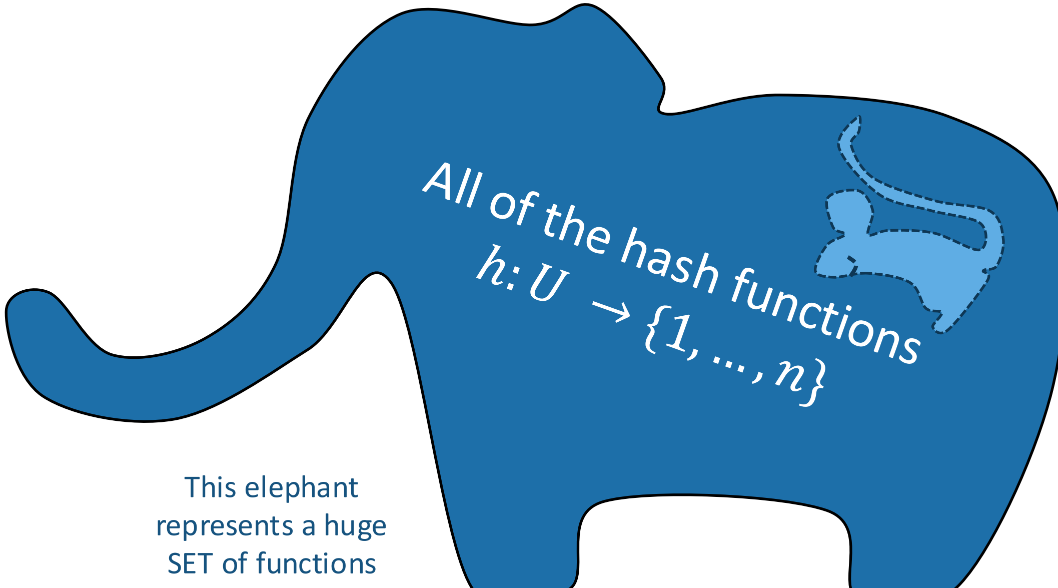
This is still a terrible idea!
Don't use this example!
For pedagogical purposes only!

Example
a smaller hash family

- H = { function which returns the least sig. digit,
  function which returns the most sig. digit }
- If we pick h in H at random, we need to store only **one bit** to remember which we picked.



This mouse represents a much smaller SUBSET of functions.
In this case, just two functions: $\{h_{\text{least-sig}}, h_{\text{most-sig}}\}$



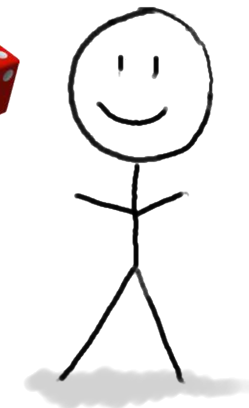
This elephant represents a huge SET of functions

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 $h: U \rightarrow \{0, \dots, 9\}$. Choose it randomly from H.

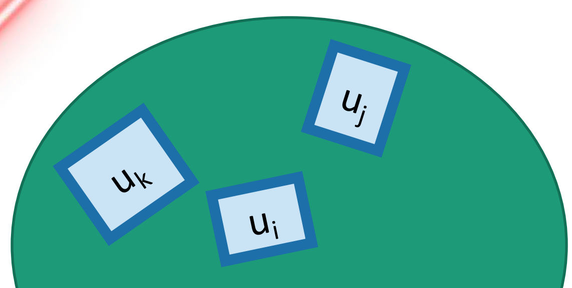
I picked h_1



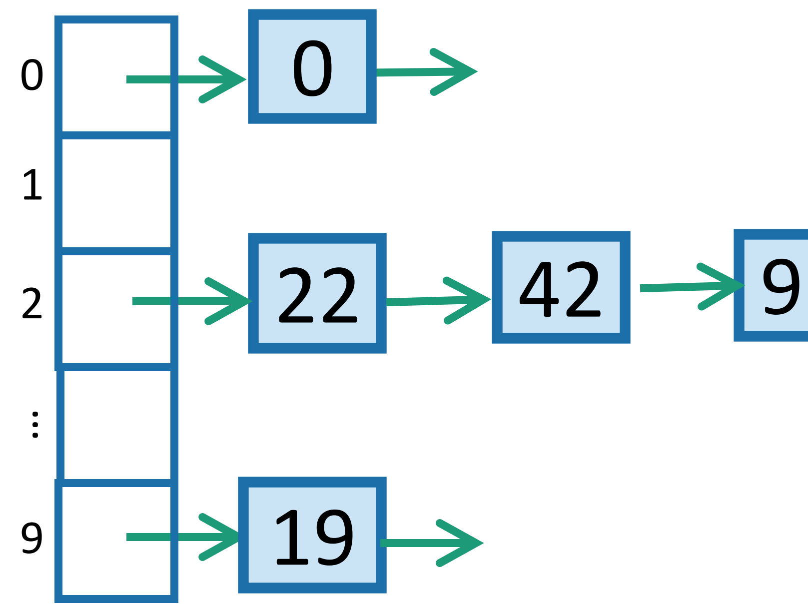
1. An adversary (who knows H) chooses any n items $u_1, u_2, \dots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

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

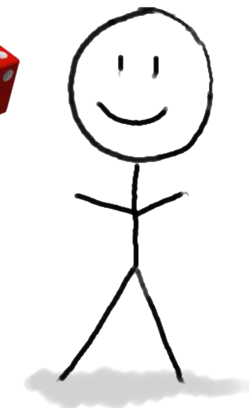
- H = { function which returns least sig. digit,

  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
$H = \{h_0, h_1\}$

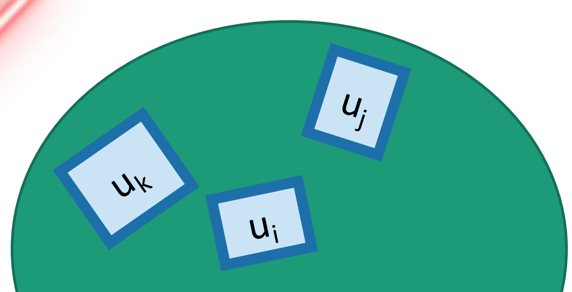
2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.



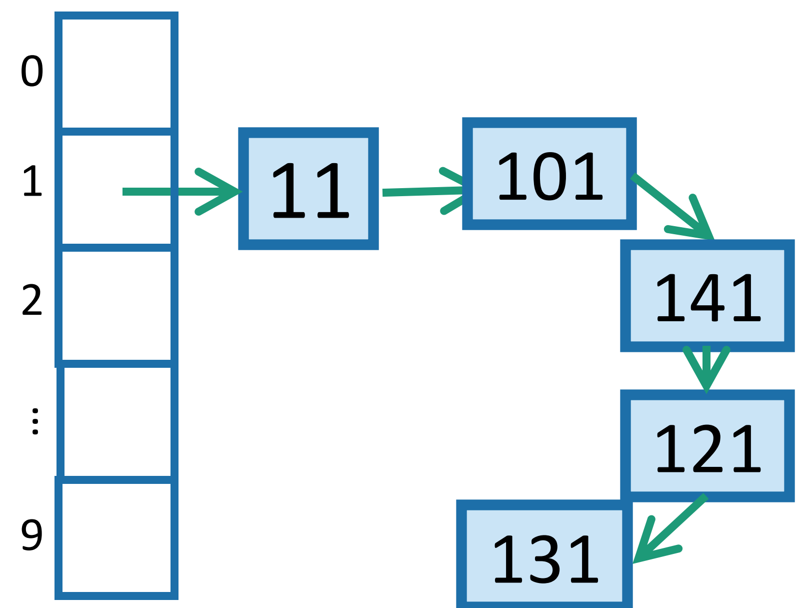
I picked h_0

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.

101 | 11 | 121 | 141 | 131



3. **HASH IT OUT** #hashpuns

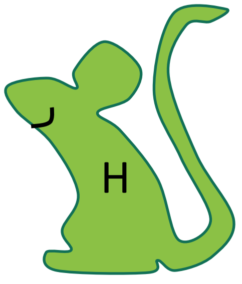


Outline

- **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….



Proof of Claim

- Let h be a uniformly random hash function.
- Then for all $i = 1, \dots, n$,
  E[number of items in u_i's bucket] $\leq$ 2.

- $E\left[\begin{array}{l}\text{\# items in}\\ u_i\text{'s bucket}\end{array}\right] =$
- $= E\left[\sum_{j=1}^{n} \mathbf{1}\{h(u_i) = h(u_j)\}\right]$
- $= \sum_{j=1}^{n} P\{\, h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{\, h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$

All that we needed was that this is 1/n

*assuming h(u) takes O(1) time to compute.

Universal hash families

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

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

- Earlier analysis shows:

  If we draw h uniformly at random from a universal hash family H, the resulting hash table will have expected time* O(1) INSERT/DELETE/SEARCH

- And if H is small (and appropriately structured), we can store a random $h \in H$ efficiently!

The whole scheme will be

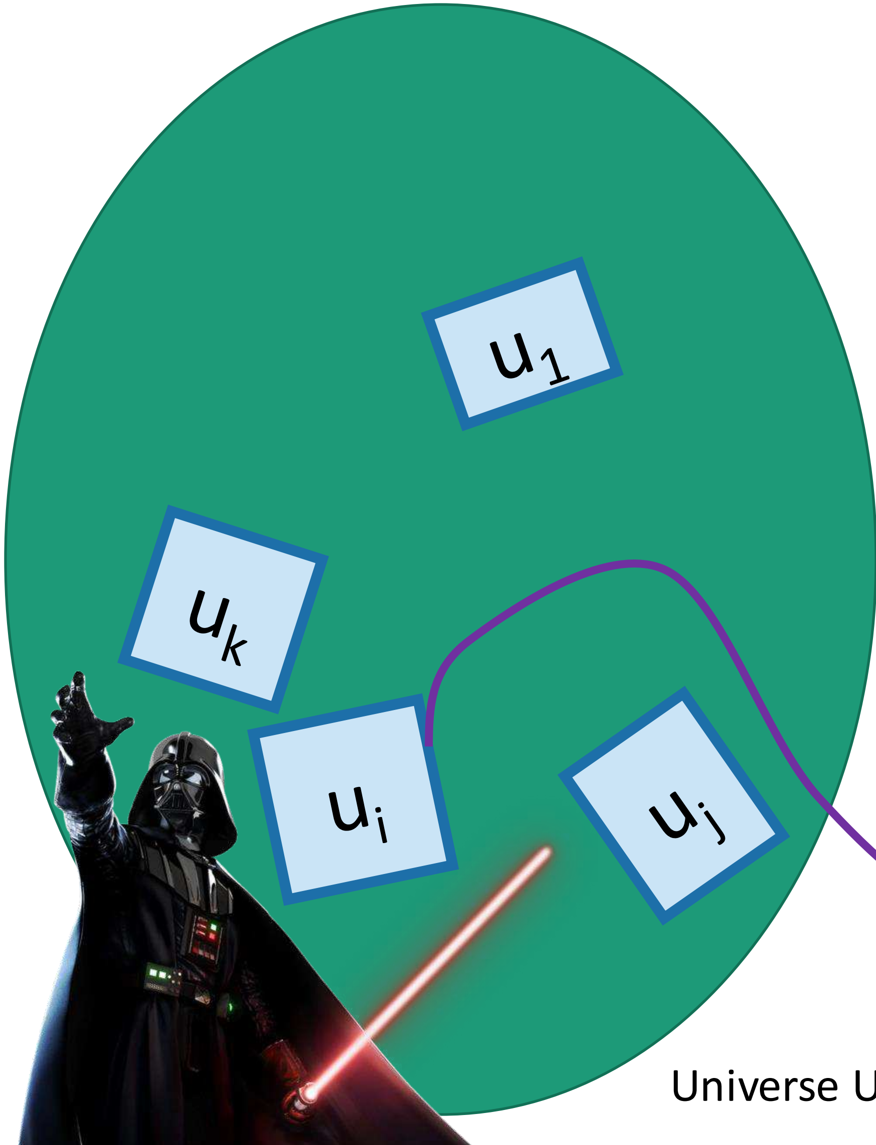
Small universal hash family H

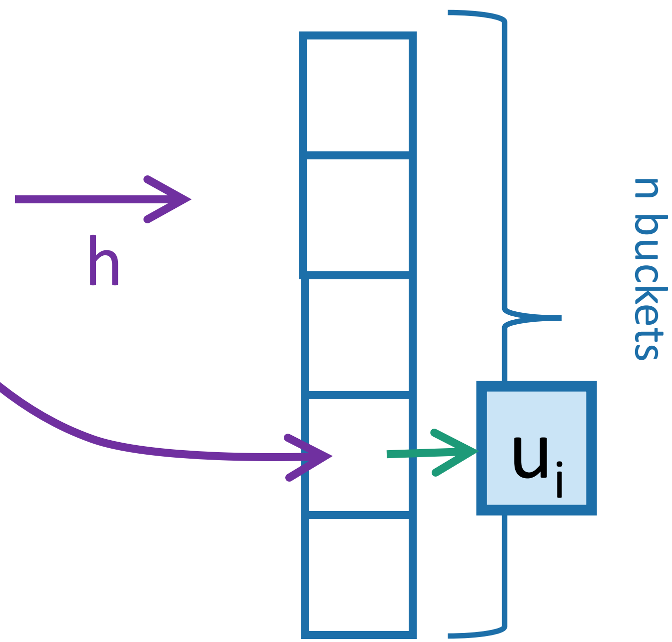
This mouse represents a small subset of functions

Choose h randomly from H

Assumption: We can store h using log|H| bits and apply it in time $O(1)$.

(We'll see a family where we can do this later)





Probably these buckets will be pretty balanced.

Universal hash families

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

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

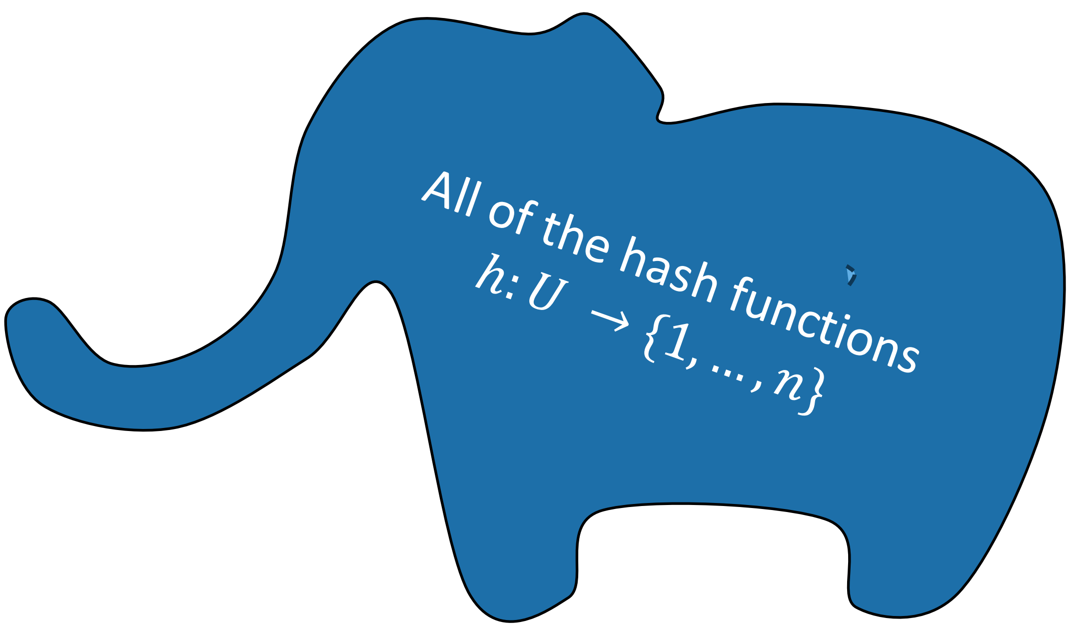
Let's see some examples
and some non-examples!

- **Universal hash family:** if you 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.



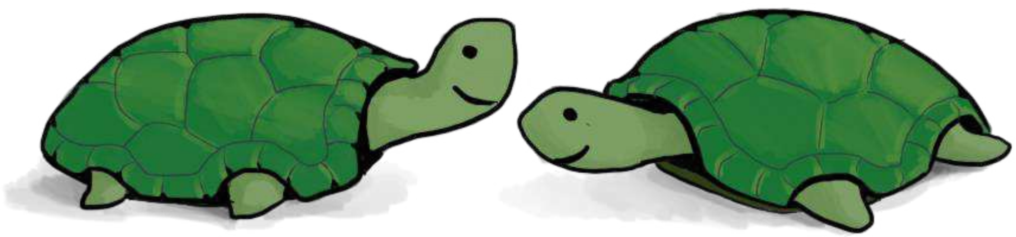
- **Universal hash family:** if you choose h randomly from H,

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

Non-example

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

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



- **Universal hash family:** if you 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 $p \geq M$. (And not much bigger than M)
  - Define

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

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

  - Define:

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

A small universal hash family??

- Here's one:
  - Pick a prime $p \geq M$. (And not much bigger than M)
  - Define

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

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

  - Define:

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

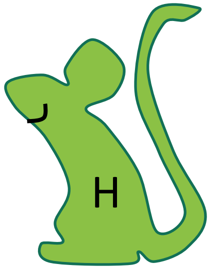
- Claims:

H is a universal hash family.

A random $h \in H$ takes $O(\log M)$ bits to store.

See CLRS (Thm 11.5) if you are curious, but you don't need to know why this is true for this class.

A random $h \in H$ takes $O(\log M)$ bits to store and is fast to evaluate!



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

$|H| = p \cdot (p-1) = O(M^2)$

- Need to store two numbers to represent $h \in H$:
  - a is in $\{1, \ldots, p-1\}$
  - b is in $\{0, \ldots, p-1\}$
  - Store a and b with $2\log(p)$ 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!
  - Example: If M = $\mathbf{128^{280}}$, log(M) = **1960.**

A small universal hash family??

- Here's one:
  - Pick a prime $p \geq M$. (And not much bigger than M)
  - Define

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

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

  - Define:

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

See CLRS (Thm 11.5) if you are curious, but you don't need to know why this is true for this class.

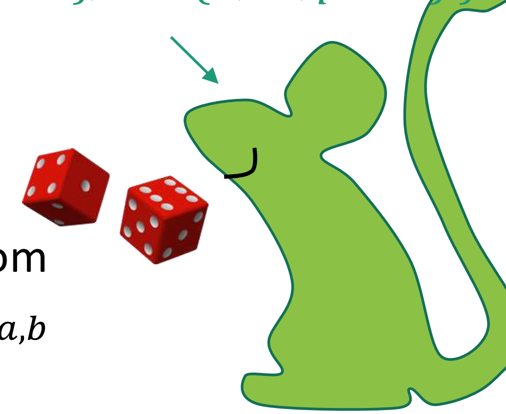
- Claims:

  H is a universal hash family.

  A random $h \in H$ takes $O(\log M)$ bits to store.

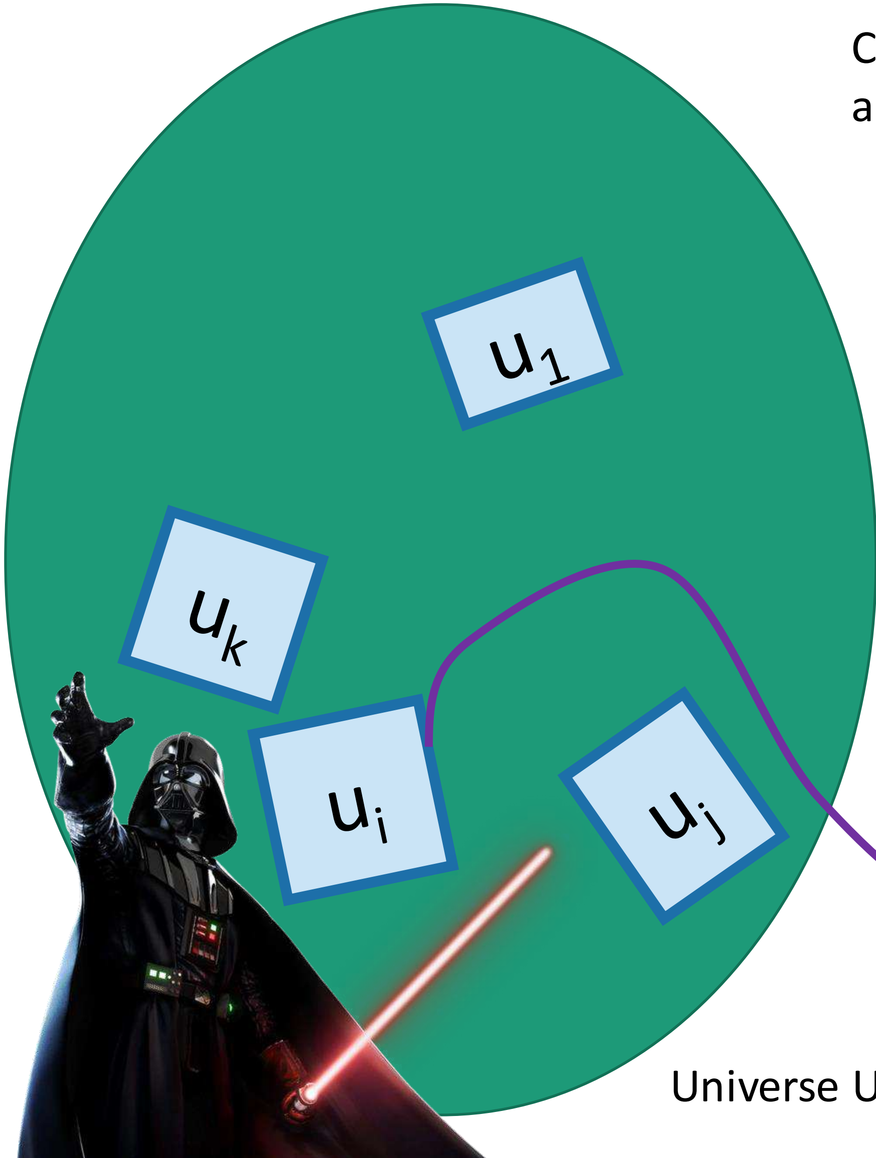
$H = \{\, h_{a,b}(x) \;:\; a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \,\}$

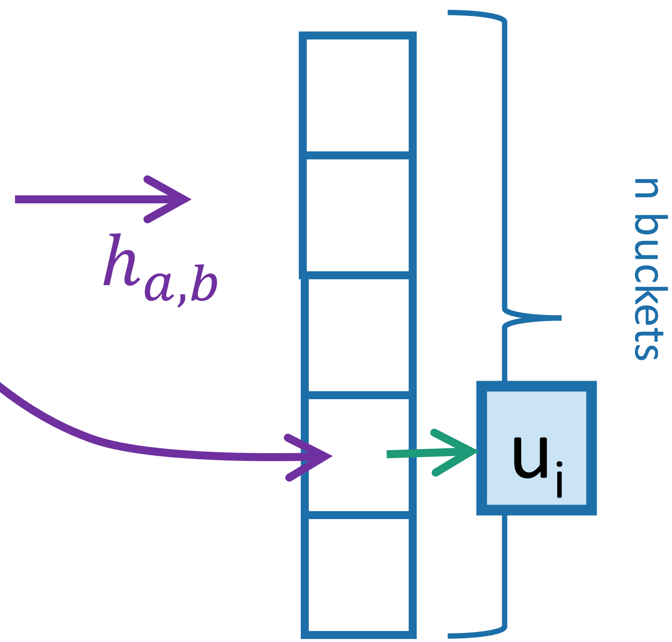
So the whole scheme will be



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

We can store $h_{a,b}$ 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

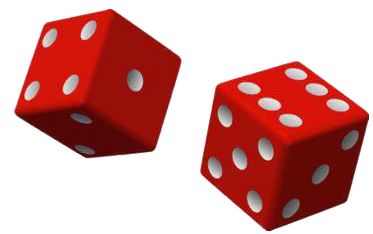
Want O(1)

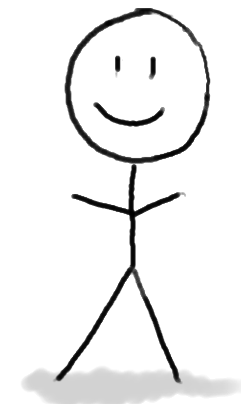
INSERT/DELETE/SEARCH

We studied this game

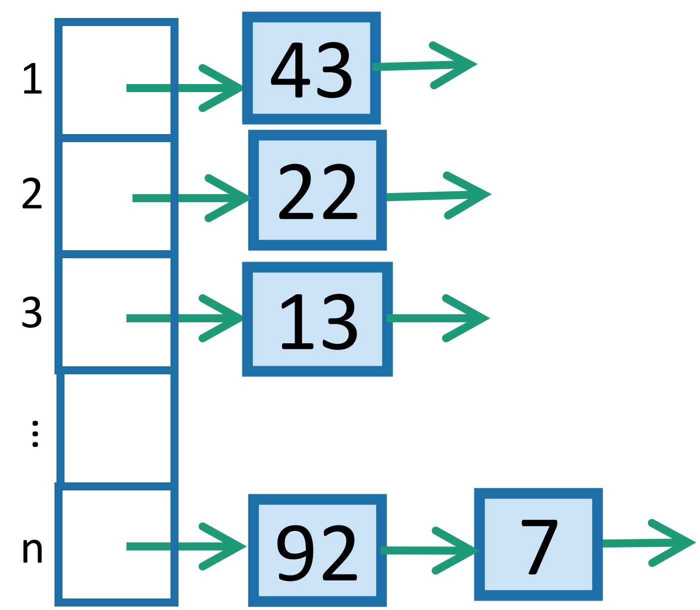
1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.

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

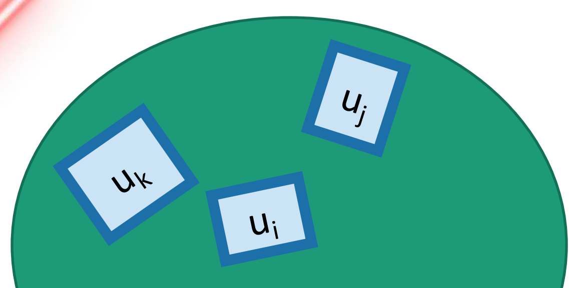




3. **HASH IT OUT**



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



Uniformly random h was good

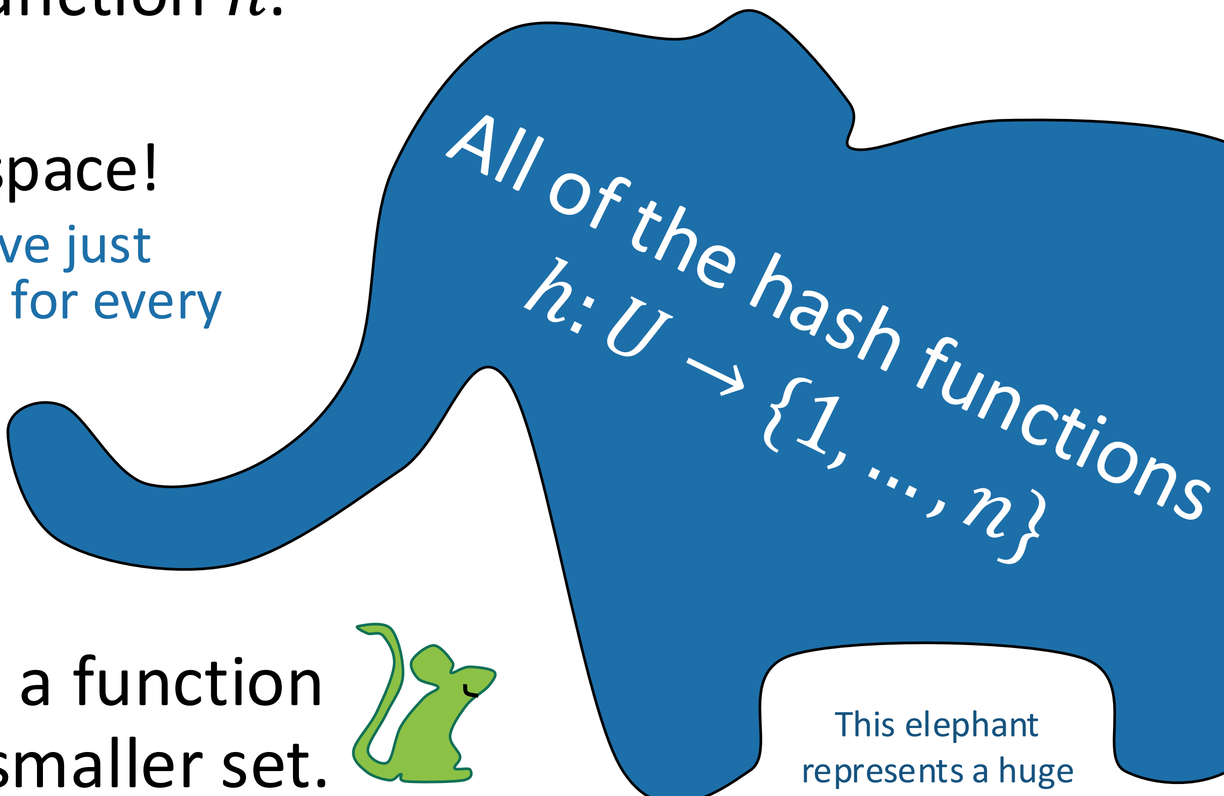
- If we choose h uniformly at random,

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

- 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.



This elephant represents a huge SET of functions

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

This mouse represents a much smaller SET of functions.

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 H, of size $O(M^2)$
  - We needed only $O(\log M)$ bits to store it!
  - And we can apply any $h \in H$ in time $O(1)$!

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

- New Unit! Graphs!

Before Next Time

- Pre-lecture exercise!