Introduction to Algorithms

L5. Hashing
(Randomized Algorithms)

Instructor : Kilho Lee

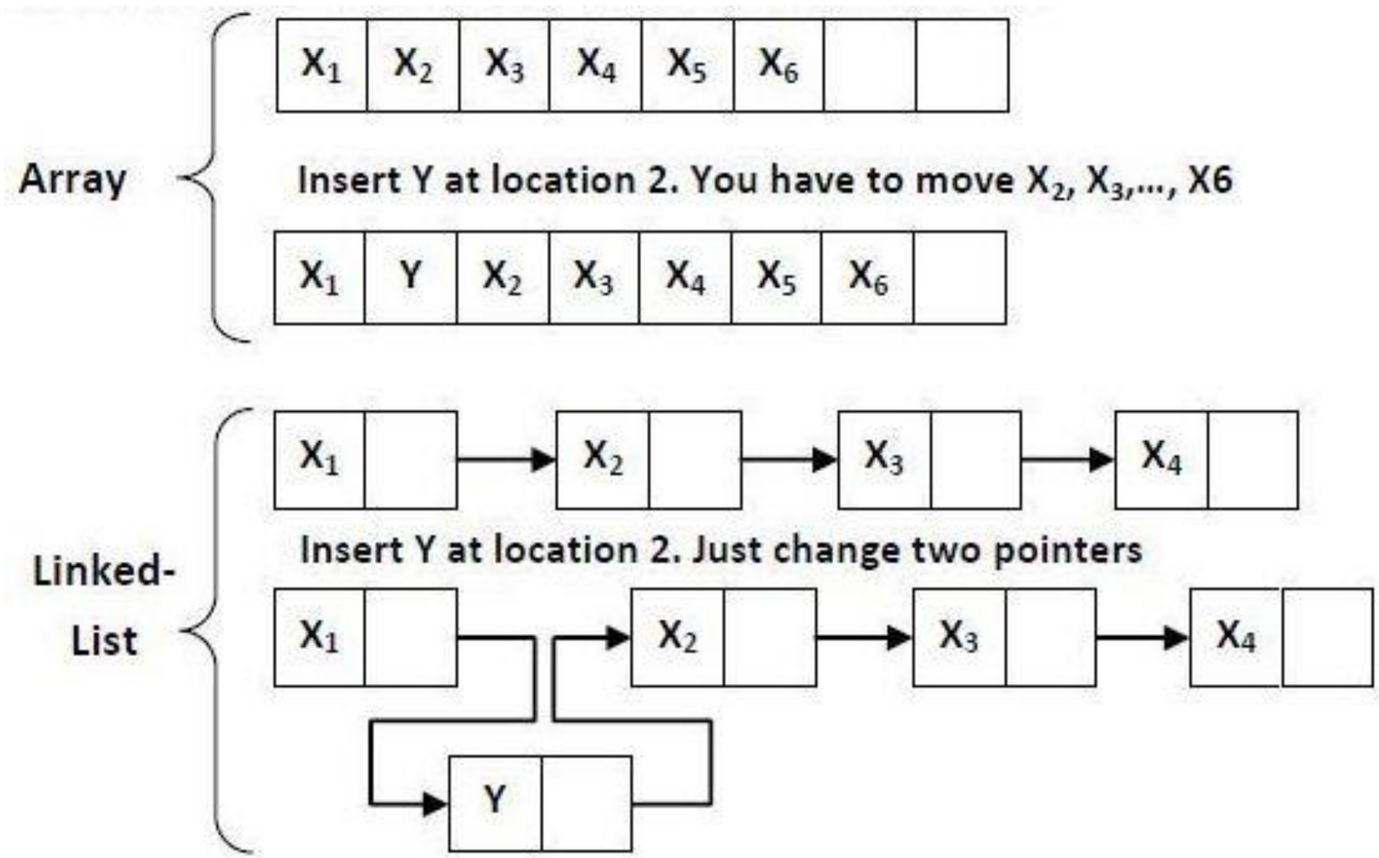
Course Overview

- Algorithmic Analysis
- Divide and Conquer
- **Randomized Algorithms (Hashing)**
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Hashing
  - Direct-address tables, hash tables, hash functions
  - Reading: CLRS: 11

Data Structures



Data Structures

	Sorted linked lists	Sorted arrays
Search	`O(n)` expected & worst-case	`O(log n)` expected & worst-case
Insert/ Delete	`O(n)` expected & worst-case without a pointer to the element	`O(n)` expected & worst-case

Data Structures

	Sorted linked lists	Sorted arrays	Hash tables
Search	`O(n)` expected & worst-case	`O(log n)` expected & worst-case	`O(1)` expected `O(n)` worst-case
Insert/ Delete	`O(n)` expected & worst-case without a pointer to the element	`O(n)` expected & worst-case	`O(1)` expected `O(n)` worst-case without a pointer to the element

Hashing Basics

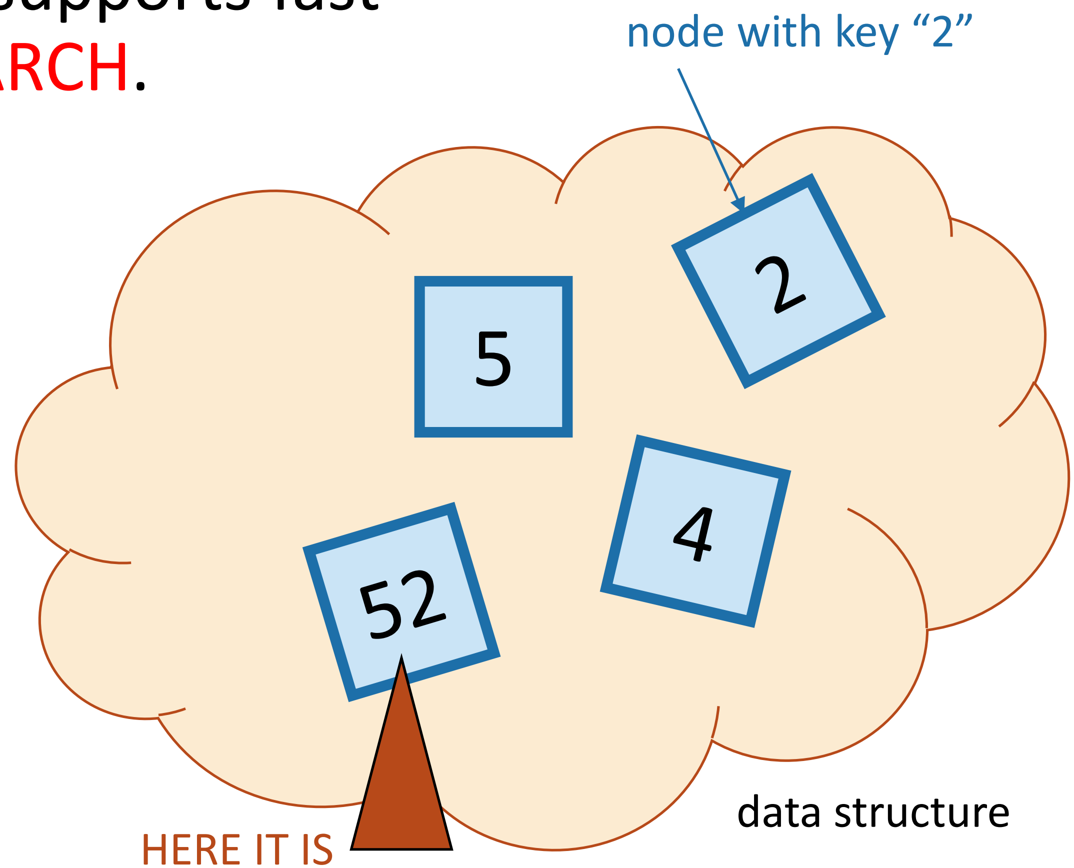
Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.

Goal

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

- INSERT 5
- DELETE 4
- SEARCH 52



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, …

Applications of Dictionary

- Perhaps, the most popular data structures in CS
  - Database: Dictionary, Spell Checking/Correcting, Web Search
  - Compiler/Interpreter: Variable Name → Physical Address
  - Network Router: IP Address → Wire
  - Network Server: Port Number → Socket/App
  - Virtual Memory: Virtual Address → Physical Address
  - Substring Search: grep in UNIX, etc.
  - String Commonalities: DNA
  - File/directory synchronization: Dropbox
  - Cryptography: File Transfer & Identification, etc.
  - …

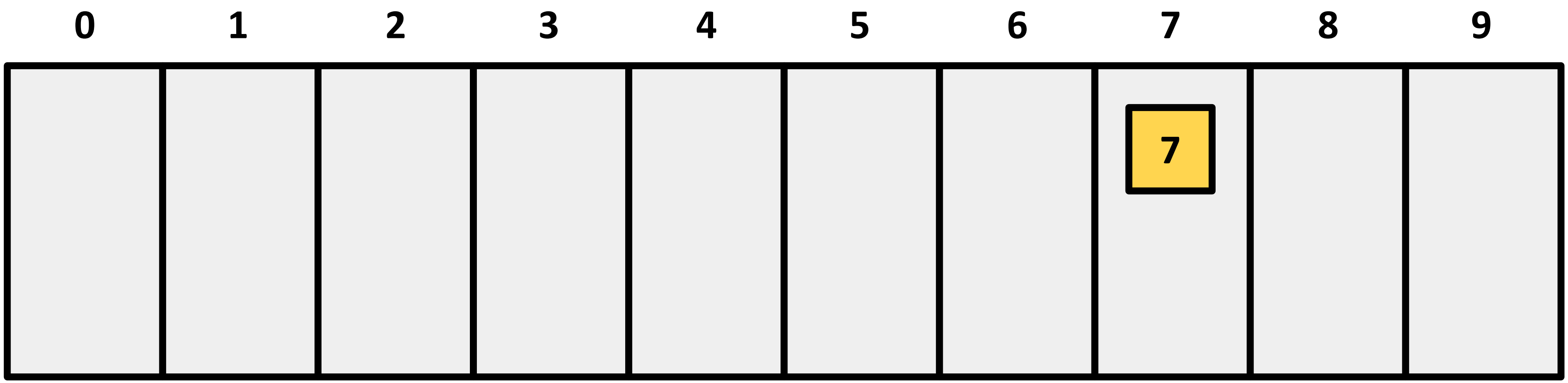
Direct Addressing

How might we get **O(1)**-time?

Try direct addressing!

One type of item per address.

`insert(7)`



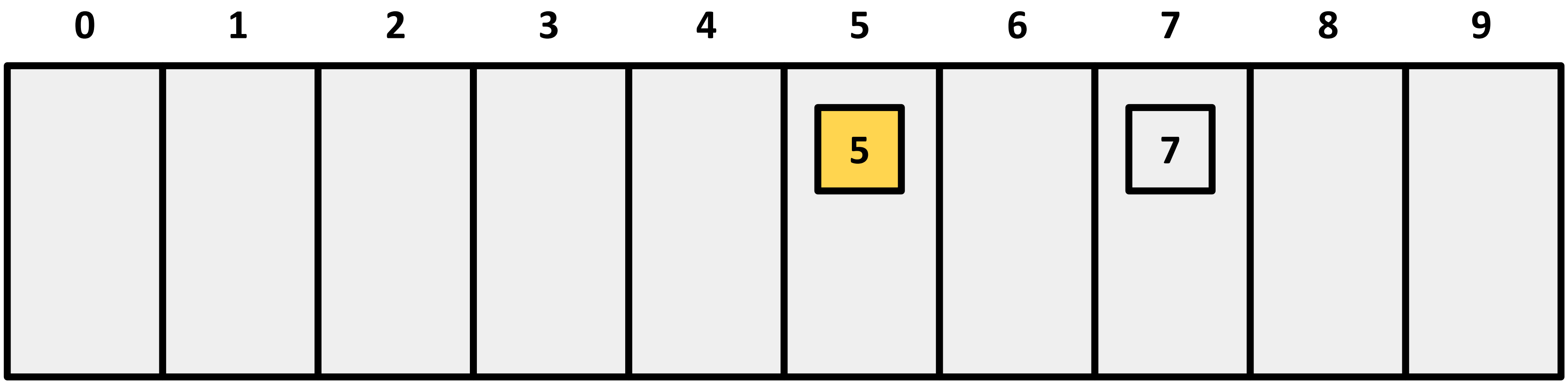
Direct Addressing

How might we get $O(1)$-time?

Try direct addressing!

One type of item per address.

```
insert(7)
insert(5)
```



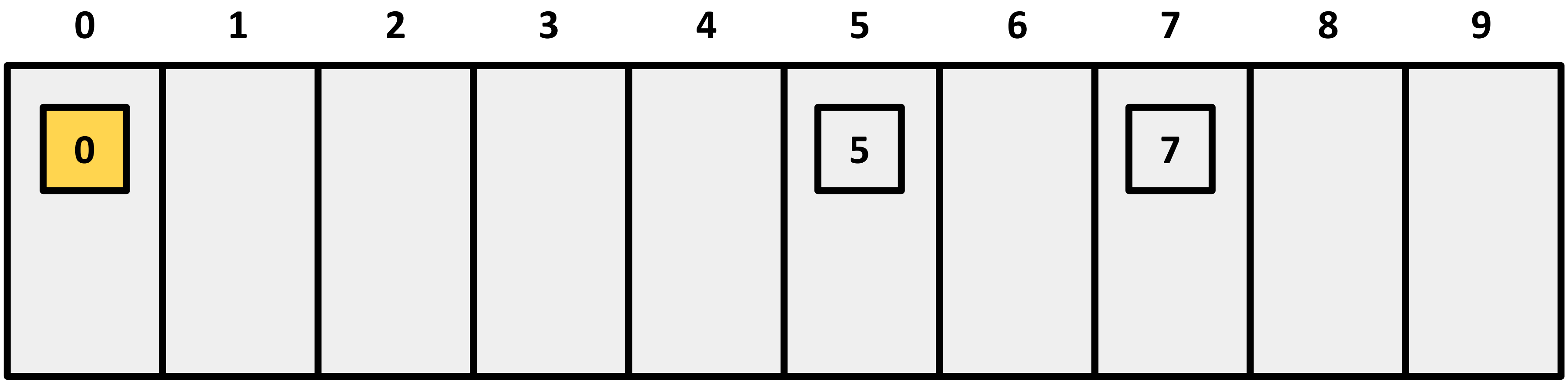
Direct Addressing

How might we get $O(1)$-time?

Try direct addressing!

One type of item per address.

```
insert(7)
insert(5)
insert(0)
```



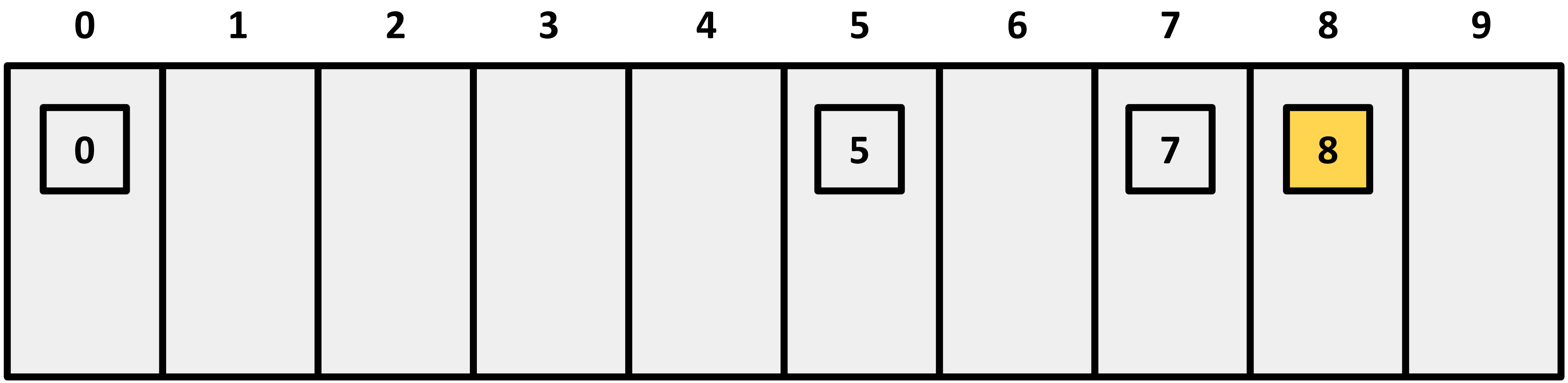
Direct Addressing

How might we get $O(1)$-time?

Try direct addressing!

One type of item per address.

```
insert(7)
insert(5)
insert(0)
insert(8)
```



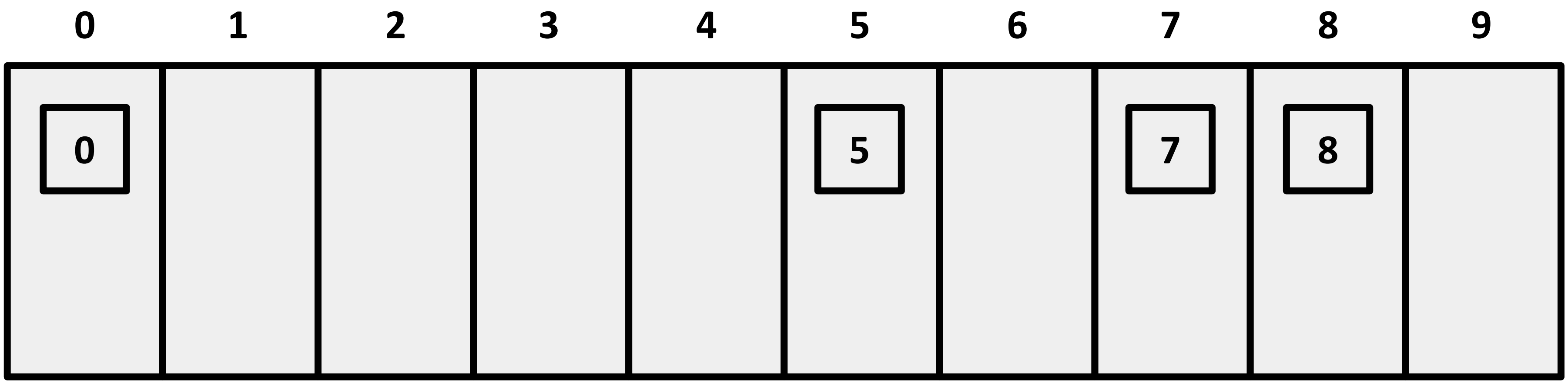
Direct Addressing

How might we get $O(1)$-time?

Try direct addressing!

One type of item per address.

```
insert(7)    search(7)
insert(5)    search(2)
insert(0)
insert(8)
```



Direct Addressing

How might we get `O(1)`-time?

Try direct addressing! 🤔

- What's the issue with this approach?

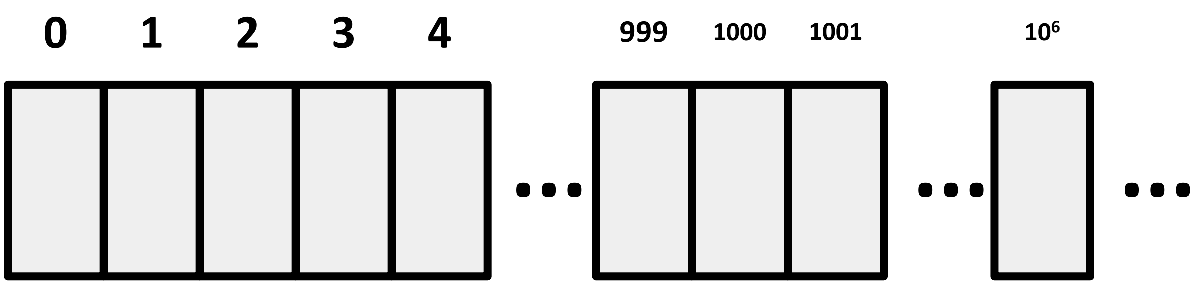
Direct Addressing

How might we get `O(1)`-time?

Try direct addressing!

What's the issue with this approach?

Similar to `counting_sort` and `bucket_sort` (for $k \leq$ num_buckets),
if the set of items being inserted/deleted (e.g. $\{0, 1, 2, \ldots, 999, 1000, \ldots, 10^6, \ldots\}$)
is **large**,
then the **space** required to maintain this data structure becomes an issue.



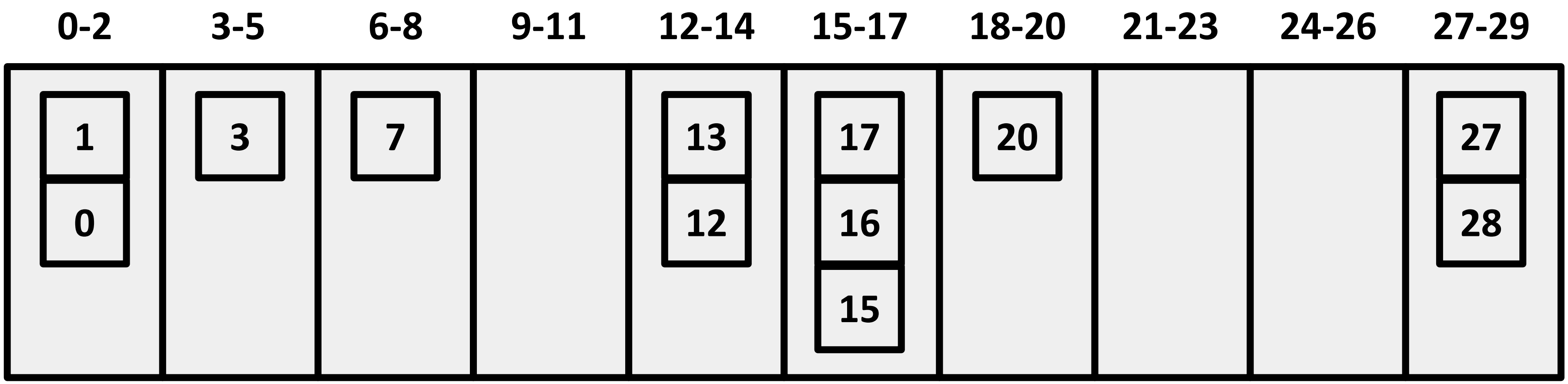
Direct Addressing

How might we get `O(1)`-time?

Try direct addressing!

Can we fix this issue by assigning multiple types of item per address, like the case of `bucket_sort`?

Sometimes, this binning approach is useful. `search(12)` still runs pretty fast.



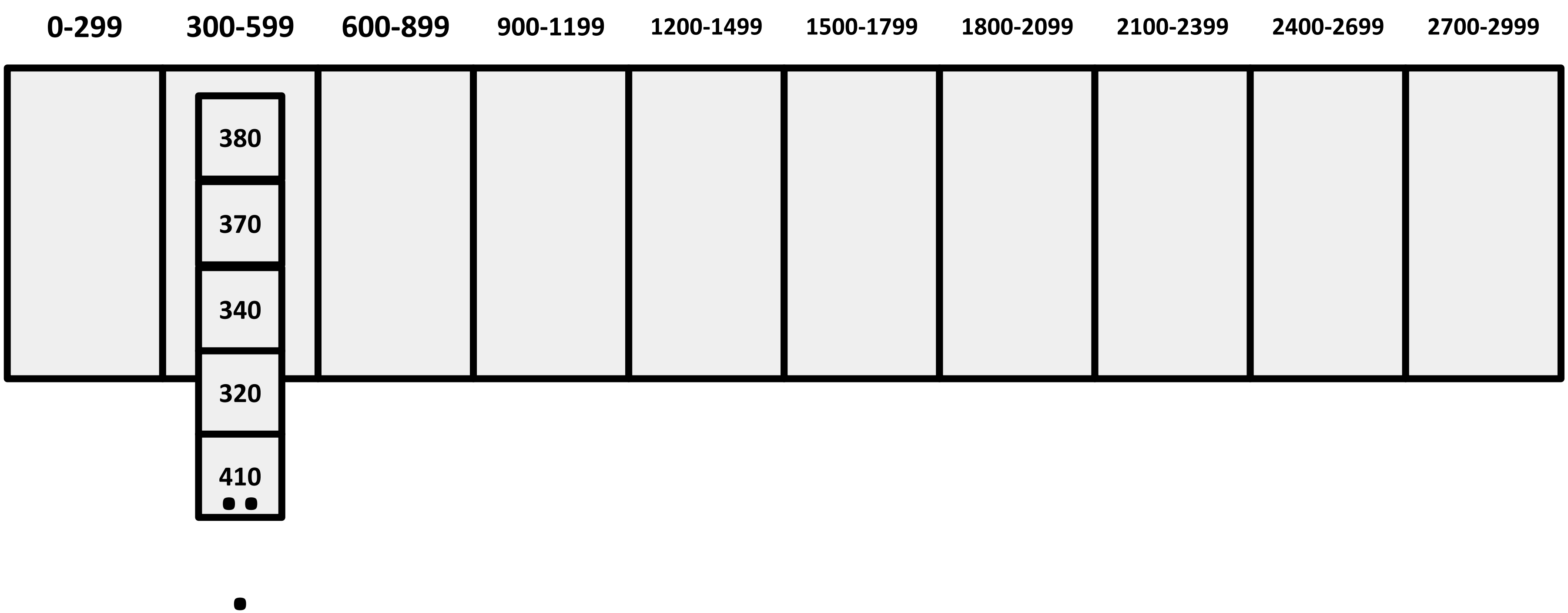
Direct Addressing

How might we get $O(1)$-time?

Try direct addressing!

Can we fix this issue by assigning multiple types of item per address, like the case of `bucket_sort`?

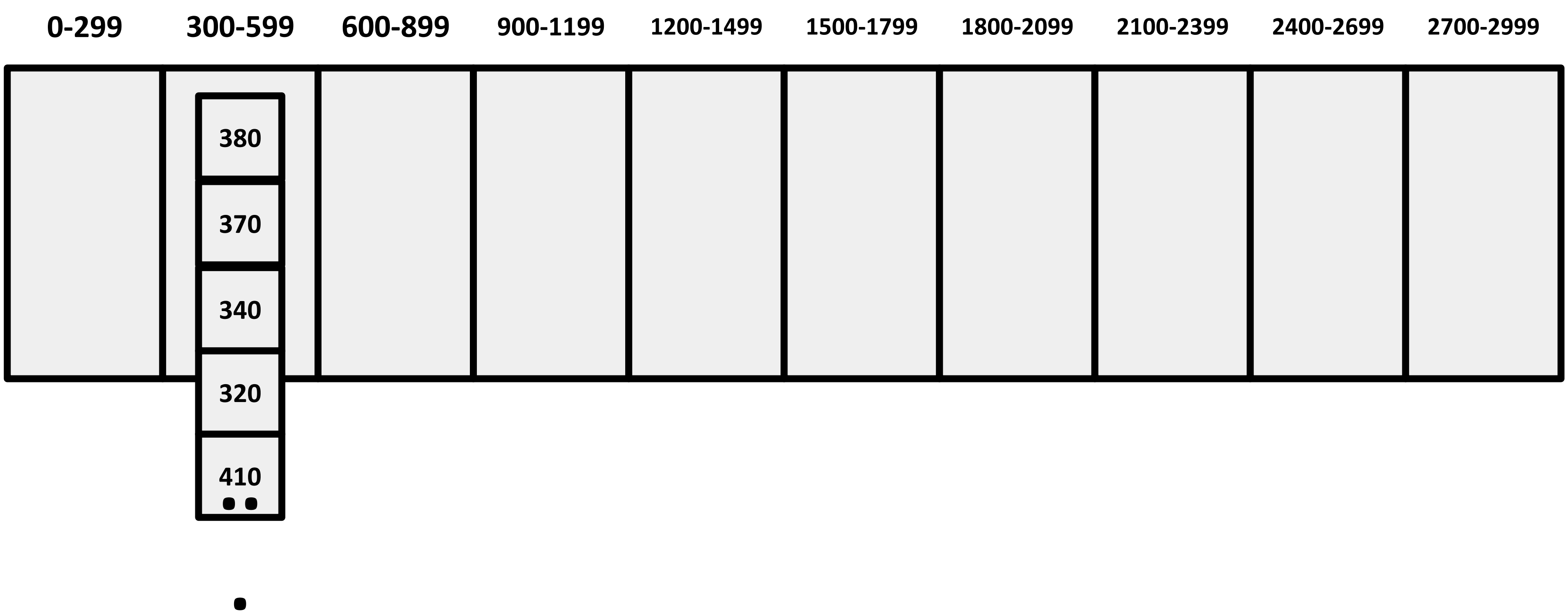
Other times, it causes an issue. `search(432)` is slow.



Direct Addressing

This is an example of a hash table.

- One with a basic bucketing scheme.
- Can we do better?

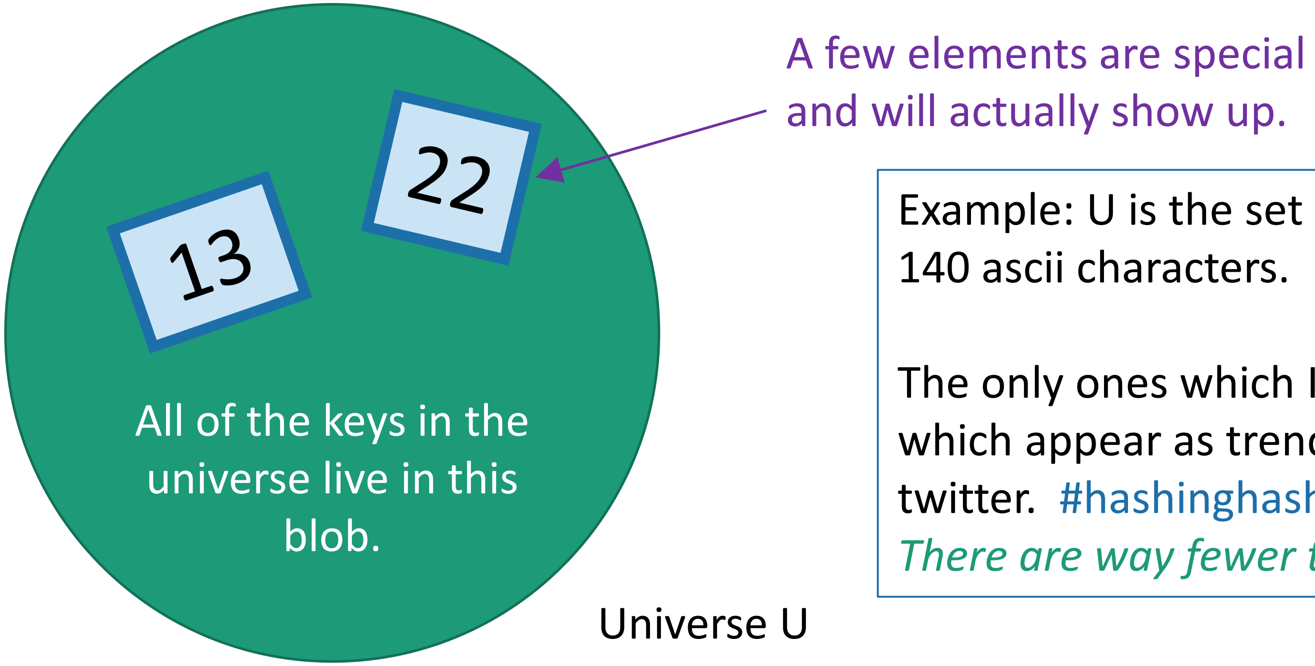


Hash tables

- 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.

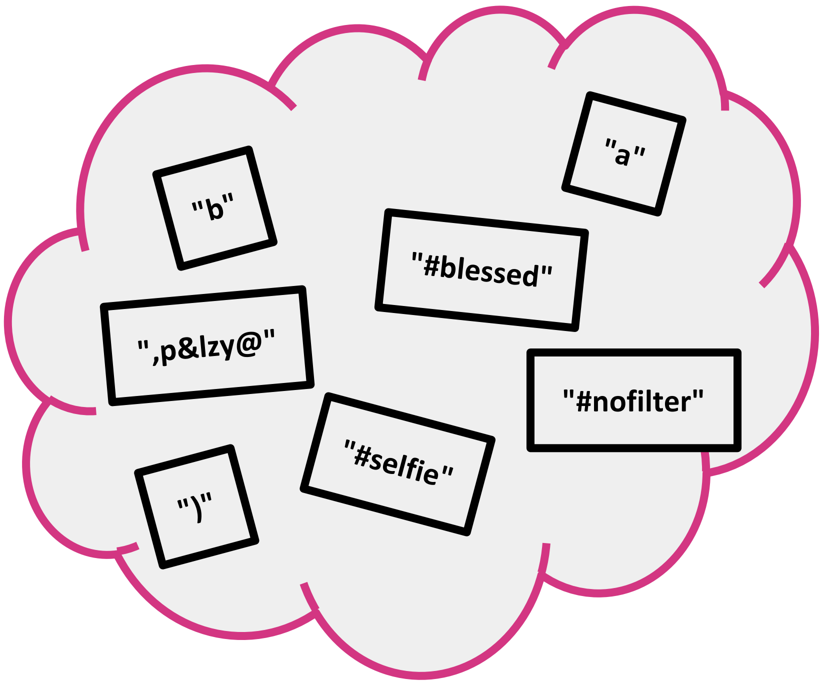
- We have a universe U, of size M.
  - M is really big.
- But only a few (say at most n for today's lecture) elements of M are going to show up.
  - M is waaaayyyyyyy bigger than n.
- But we don't know which ones will show up in advance.

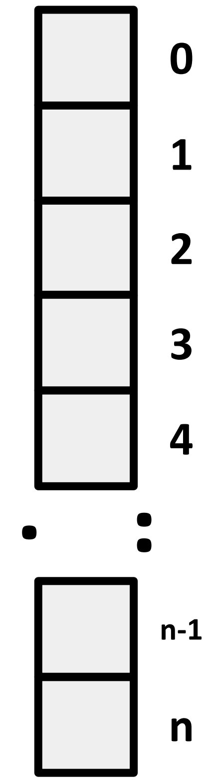


Example: U is the set of all strings of at most 140 ascii characters. (128^{140} of them).

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.

An Example

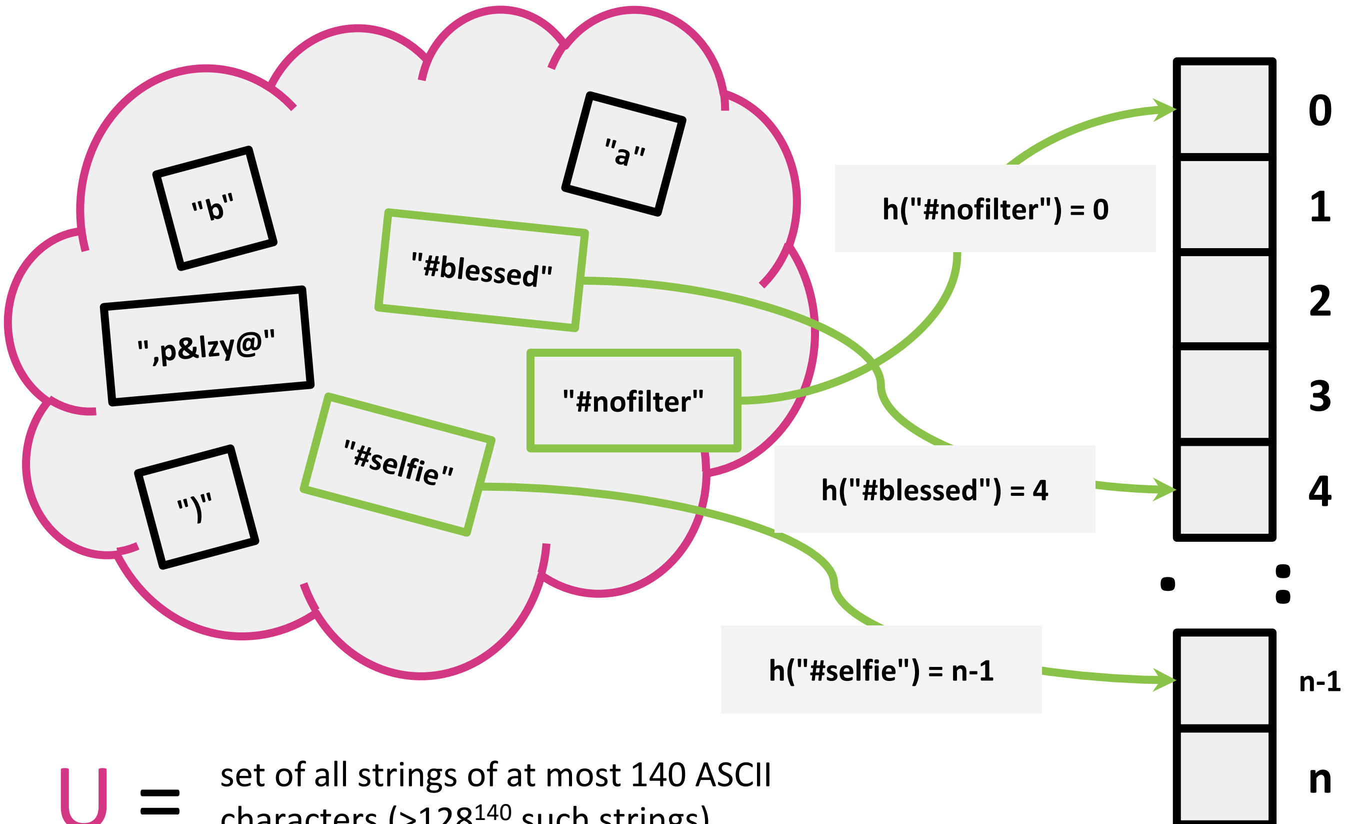




U = set of all strings of at most 140 ASCII characters (128^{140} such strings)

And we'll need to store a small subset of U
(say, the ones that might be trending hashtags on Twitter);
There are way fewer than 128^{140} of these.

An Example

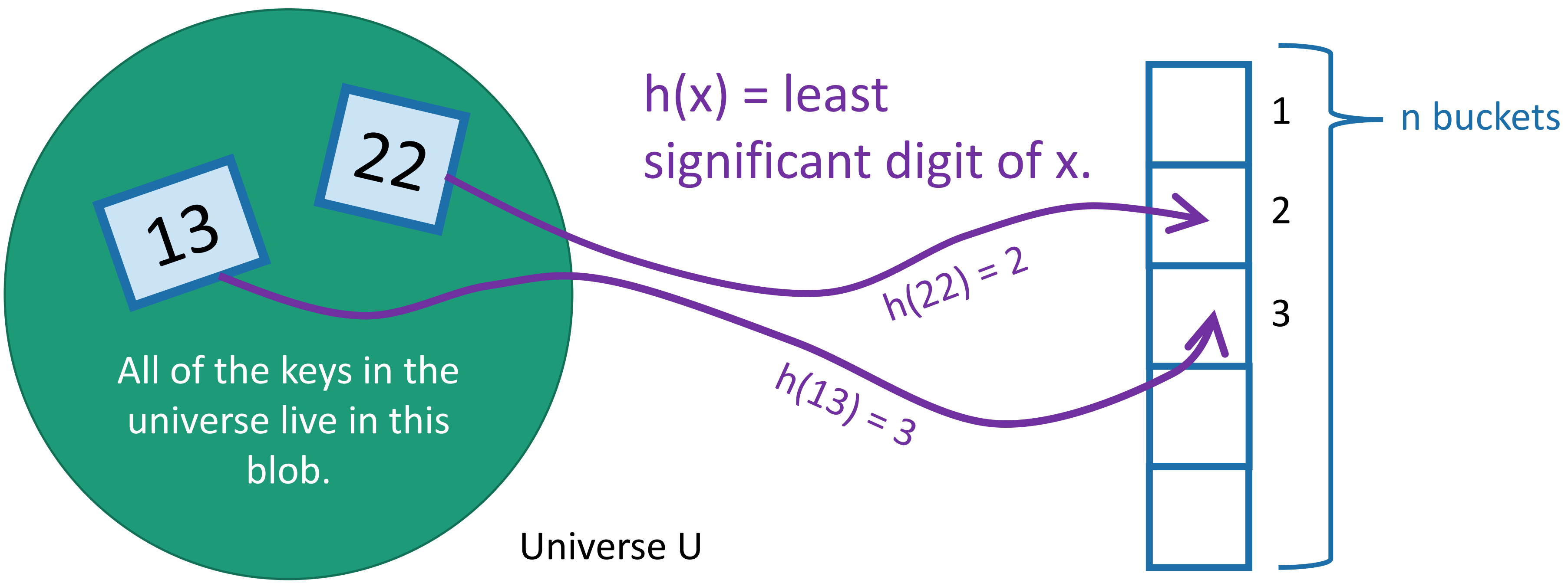


U = set of all strings of at most 140 ASCII characters (>128^{140} such strings)

And we'll need to store a small subset of U
(say, the ones that might be trending hashtags on Twitter);
There are way fewer than 128^{140} of these.

The previous example
with this terminology

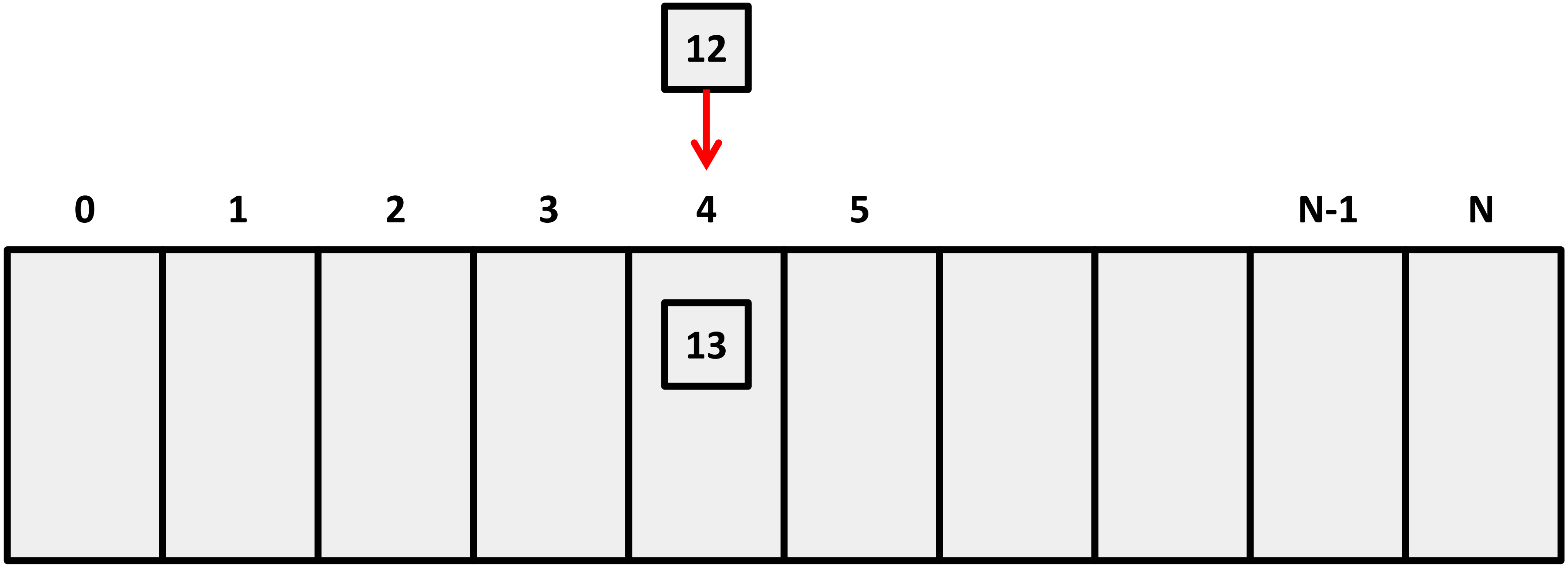
- We have a universe U, of size M.
  - at most n of which will show up.
- M is waaaayyyyyy bigger than n.
- We will put items of U into n buckets.
- There is a hash function h:U → {1,…,n} which says what element goes in what bucket.



Hash table

What happens if two values have the same hash value:

→ **Hash collision**



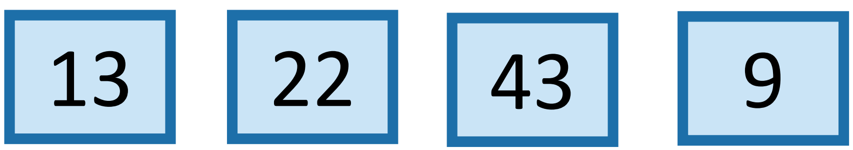
This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores an unsorted linked list.
  - `insert` in **`O(1)`** since it's unsorted;
  - `search` in **`O(length(list))`**.
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness,

  let's stick with h(x) = least significant digit of x.

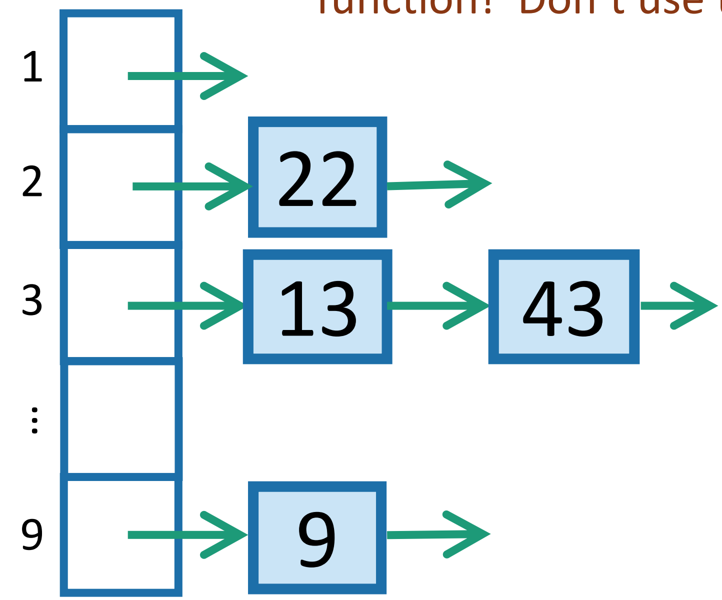
For demonstration purposes only!
This is a terrible hash function! Don't use this!

INSERT:



SEARCH 43:

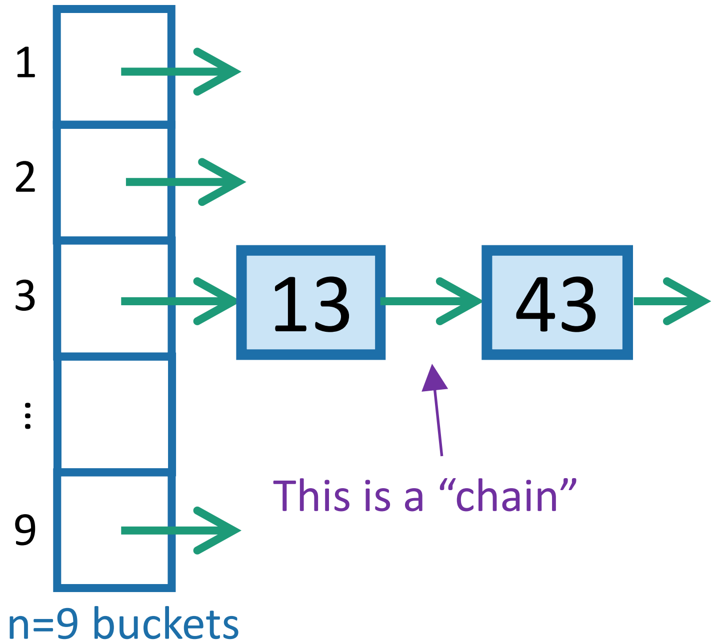
Scan through all the elements in bucket h(43) = 3.

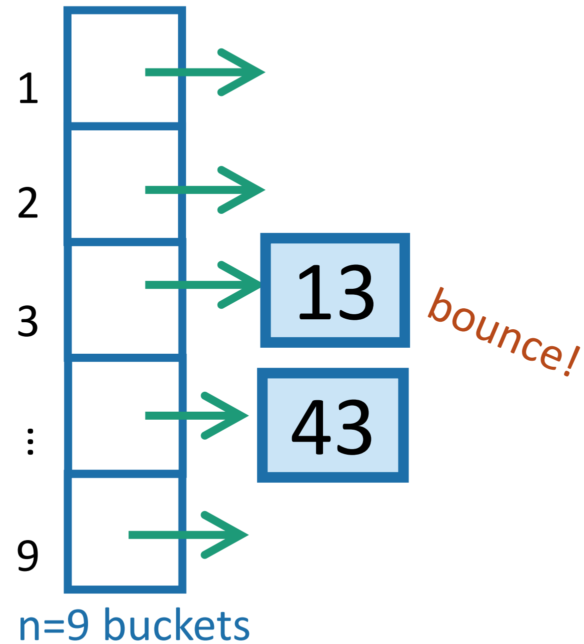


n buckets (say n=9)

Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS 11.4 if you are interested!





\end{Aside}

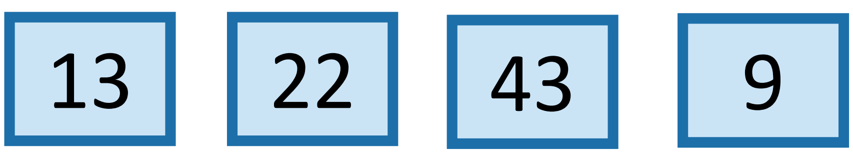
This is a **hash table** (with chaining)

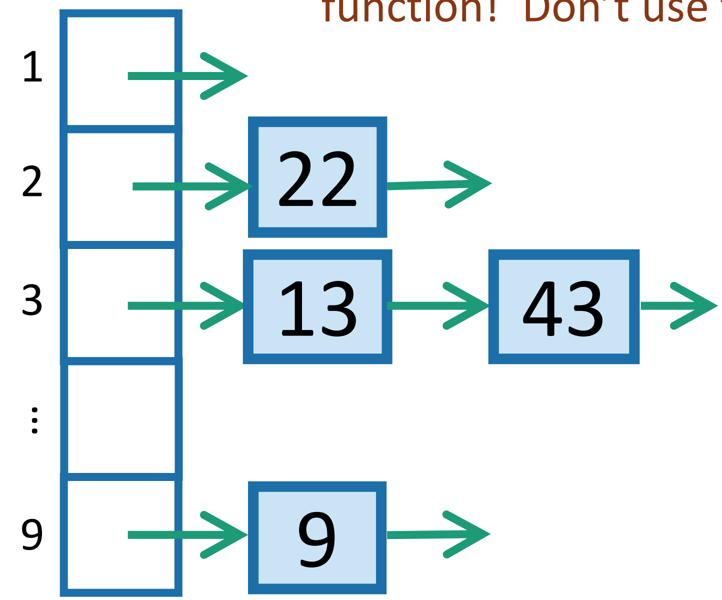
- Array of n buckets.
- Each bucket stores a linked list.
  - `insert` in `O(1)` since it's unsorted;
  - `search` in `O(length(list))`.
- h:U → {1,…,n} can be any function:
  - but for concreteness,

  let's stick with h(x) = least significant digit of x.

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

INSERT:





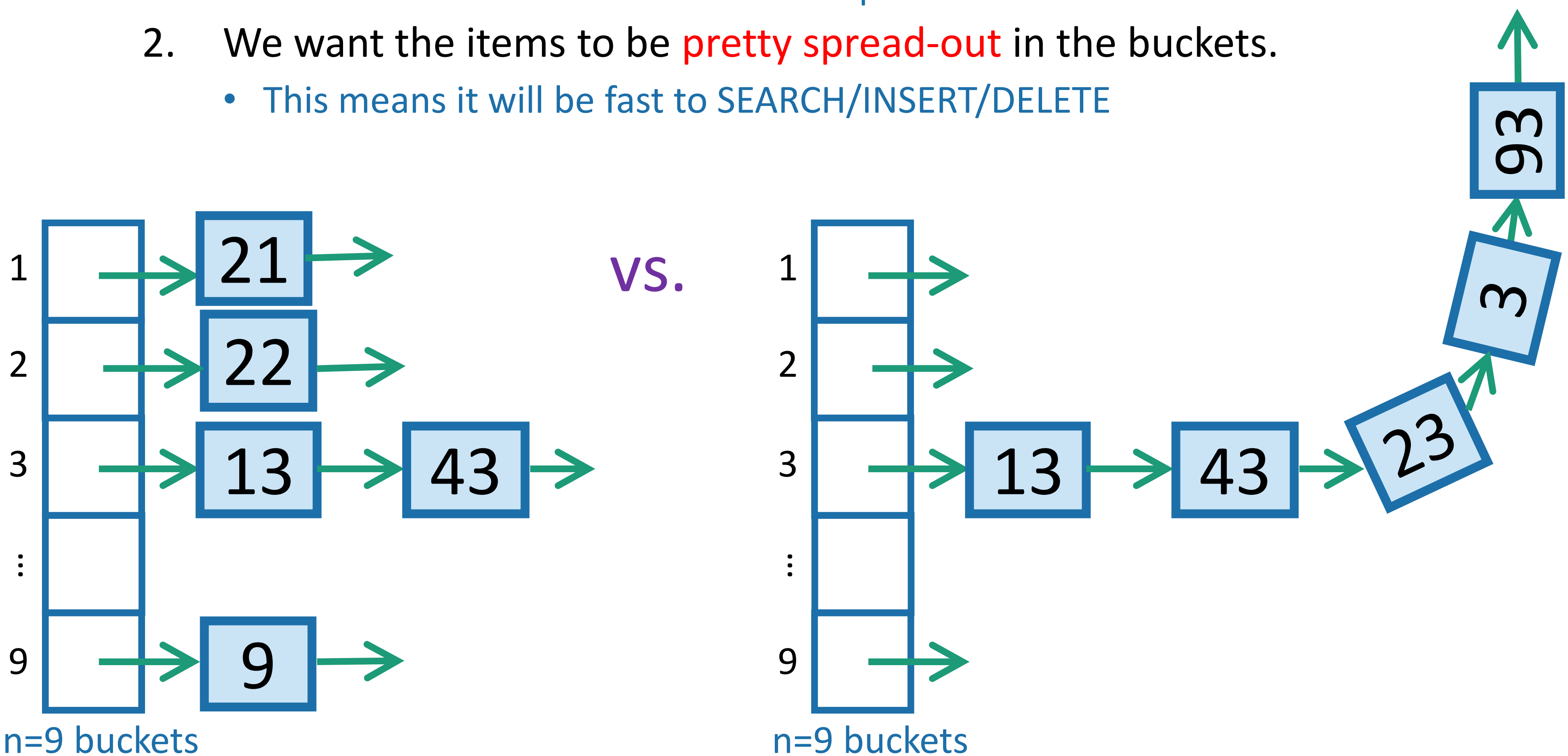
n buckets (say n=9)

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



Worst-case analysis

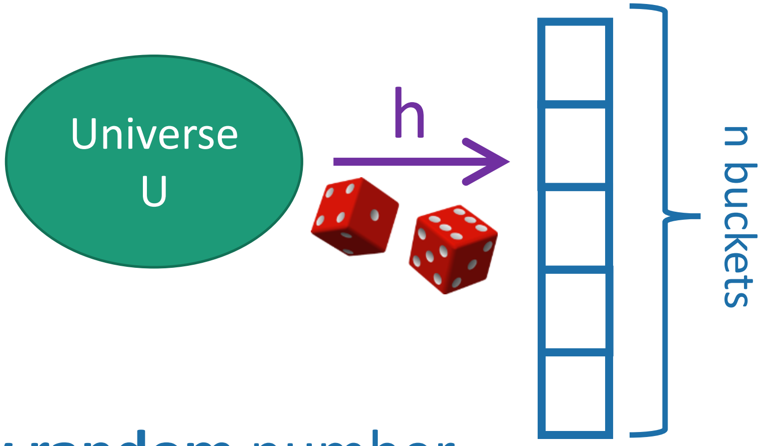
- Design a function h: U -> {1,…,n} so that:
  - No matter what input (fewer than n items of U) **a bad guy** chooses, the buckets will be balanced.
  - Here, balanced means O(1) entries per bucket.

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

Can you come up with such a function?

Solution:
Randomness

Example



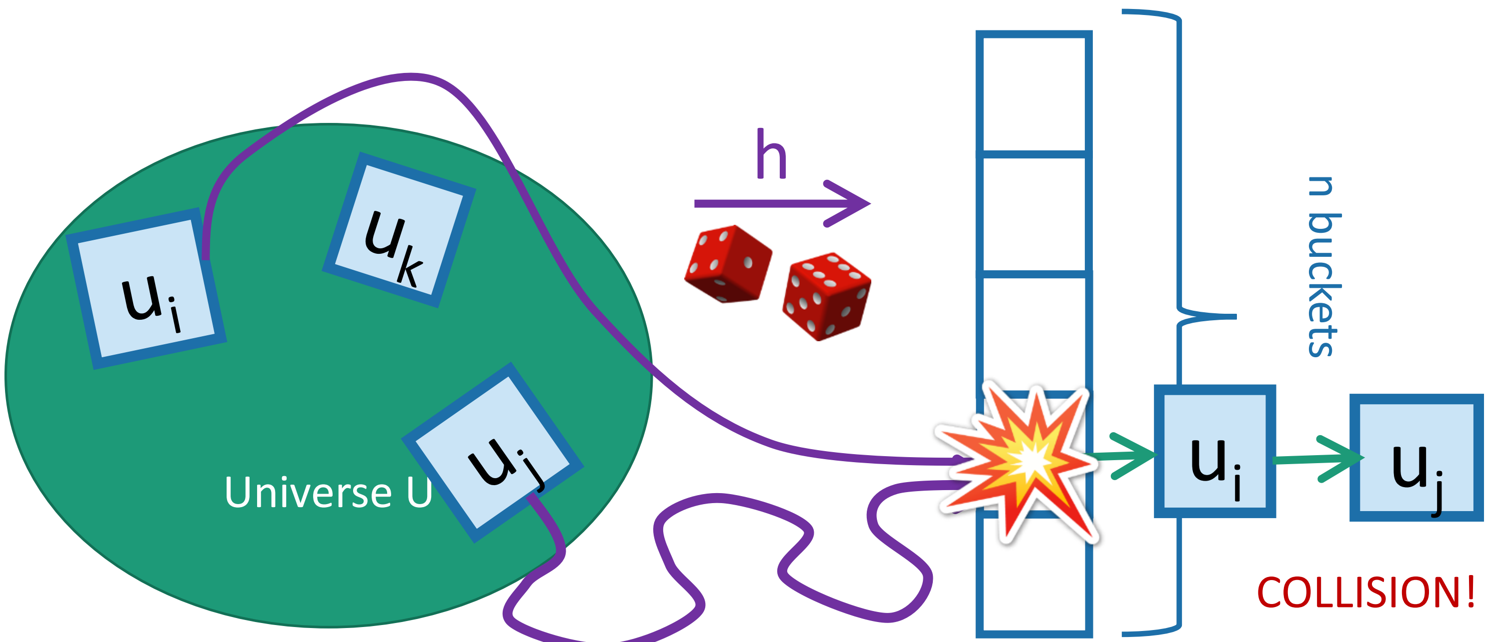
- Say that h *is* uniformly random.
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
 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

h is uniformly random

Expected number of items in u_i's bucket?

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

That's what we wanted.



That's great!

- For all i=1, …, n,
  - E[number of items in u_i 's bucket] ≤ 2

- This implies (as we saw before):
  - For any sequence of INSERT/DELETE/SEARCH operations on any n elements of U, the expected runtime (over the random choice of h) is ***O(1) per operation.***

So, the solution is:

pick a uniformly random hash function.

A simple (deterministic) hash function

- Here's one:

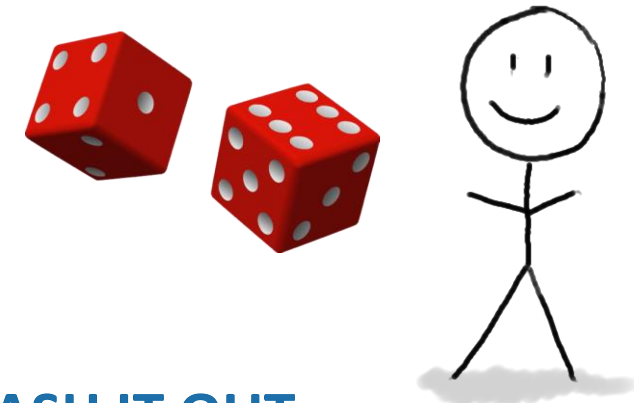
To hash an integer x in {0, …, M-1} to a bucket {1, …, n}:

  - Pick a prime n.
  - Define

$$h(x) = x \, mod \, n$$

The game

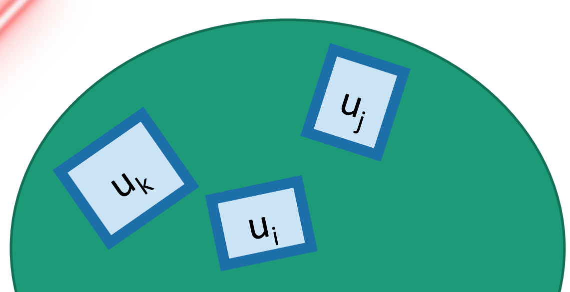
2. You, the algorithm, chooses a **random** hash function $h: U \to \{1, \ldots, n\}$.



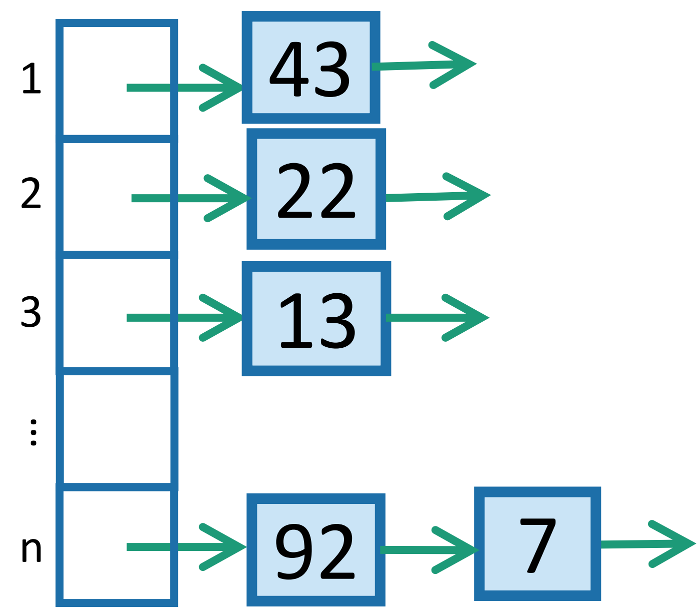
1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

13 22 43 92 7

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



3. **HASH IT OUT**



Why should that help?

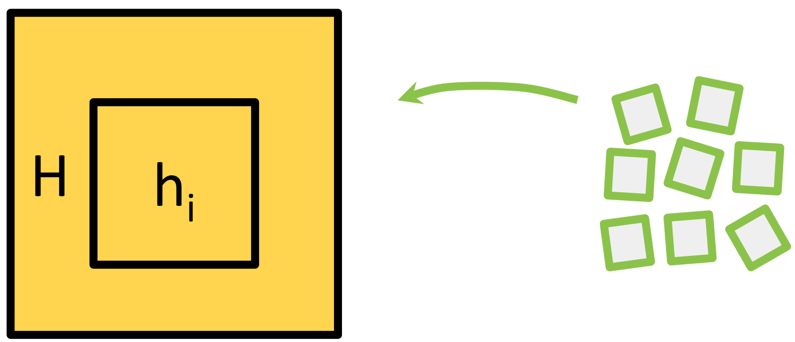
Intuitively: The bad guy can’t foil a hash function that he doesn’t yet know.

We’ll need to do some analysis…

Lots of h's?

(3) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h_i: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, all buckets will have **expected** size O(1) ~~after hashing any n items?~~ after an adversary chooses n items to hash?

1. You choose your set of hash functions H.



2. An adversary gives your hash function n items to hash.

3. You randomly pick a hash function h_i from H to hash the n items.

Is it possible to construct H such that you're guaranteed that all buckets will have expected size O(1)? This would be good.

A universal hash family??

- Here's one: To hash an integer x in {0, …, M-1} to a bucket {1, …, n}:
  - Pick a prime $p \geq M$.
  - Define

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

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

  - Claim:

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

is a universal hash family.

Summary

- Direct addressing
  - Pros: O(1) INSERT/DELETE/SEARCH
  - Cons: large space required
- Hash table with chaining
  - Pros: small space than direct addressing
  - Issues: `insert` in `O(1)`, `search` in `O(length(list))`.
    - We want the items to be pretty spread-out in the buckets.
    - How to design a hash function?
- Hash function
  - Deterministic hash function
  - Universal hash family

Today's Outline

- Hashing
  - ~~Direct-address tables, hash tables, hash functions~~ **Done!**
  - Reading: CLRS: 11

Post any question
On the Q&A board.