Randomized Algorithms II
Hashtables

Outline for Today

More randomized algorithms!

- Hashing Basics
- Universal Hash Functions
- What's the Source of the Randomness?

Hashing Basics

Randomized Algorithms

A randomized algorithm is an algorithm that incorporates randomness as part of its operation.

Often aim for properties like …

- Good average-case behavior
- Getting exact answers with high probability
- Getting answers that are close to the right answer

Data Structures

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

Data Structures

	Sorted linked lists	Sorted arrays	Balanced BSTs	Hash tables
Search	`O(n)` **expected & worst-case**	`O(log n)` **expected & worst-case**	`O(log n)` **expected & worst-case** O(n) worst-case for generic BSTs	`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(log n)` **expected & worst case**	`O(1)` **expected** `O(n)` **worst-case** without a pointer to the element

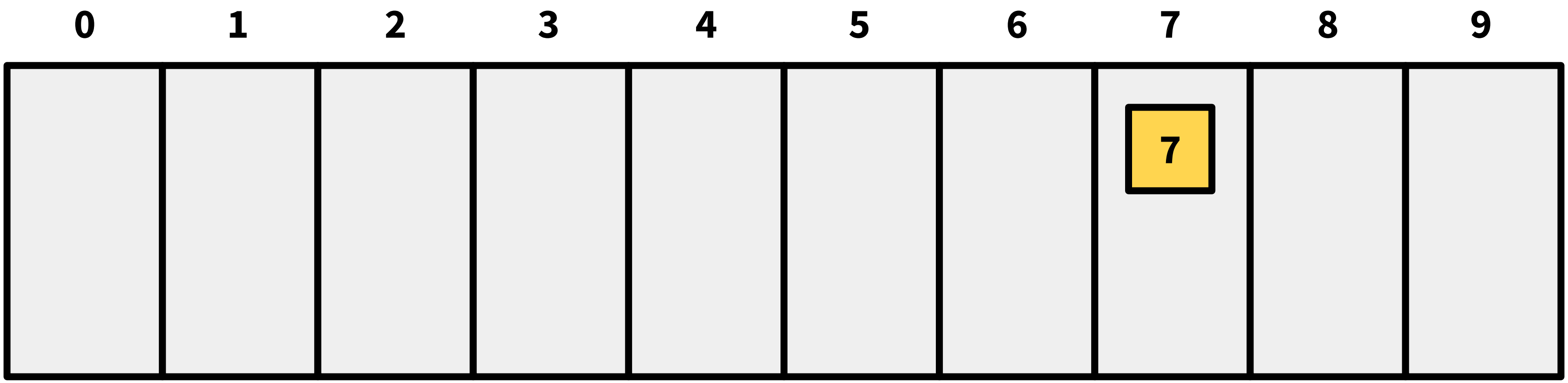
Direct Addressing

Direct addressing means we can know the memory position of an element based on the element directly.

How might we get **`O(1)`**-time? Try direct addressing!

A simple case: #Buckets > Maximum possible number, so one item per address.

`insert(7)`

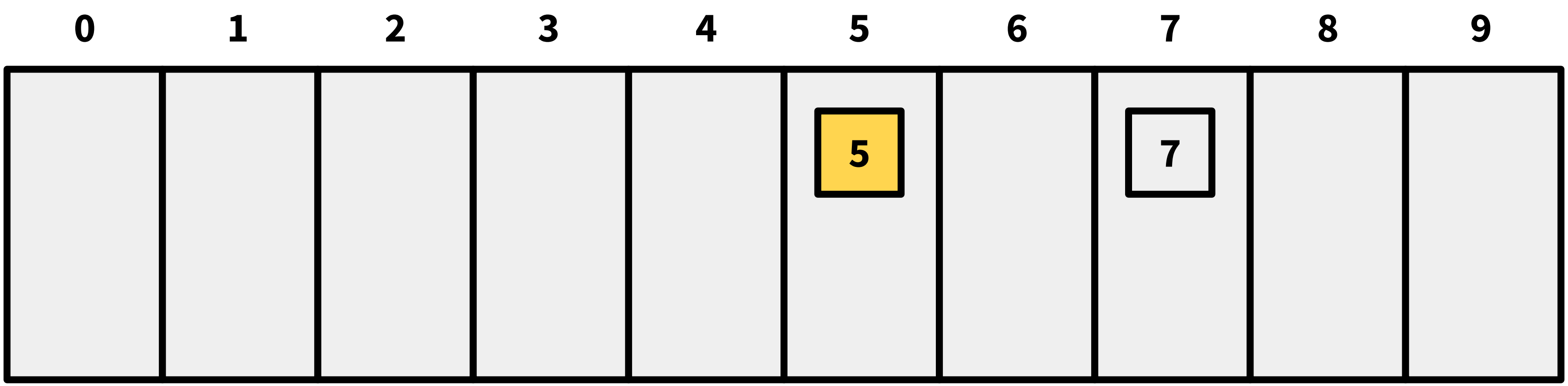


Direct Addressing

How might we get $O(1)$-time? Try direct addressing!

One item per address.

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

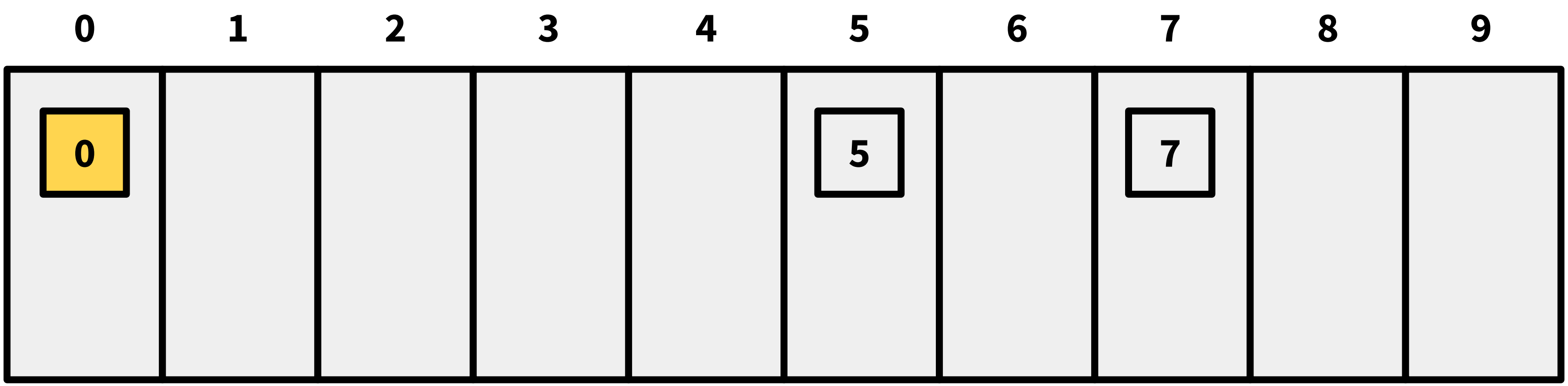


Direct Addressing

How might we get `O(1)`-time? Try direct addressing!

One item per address.

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

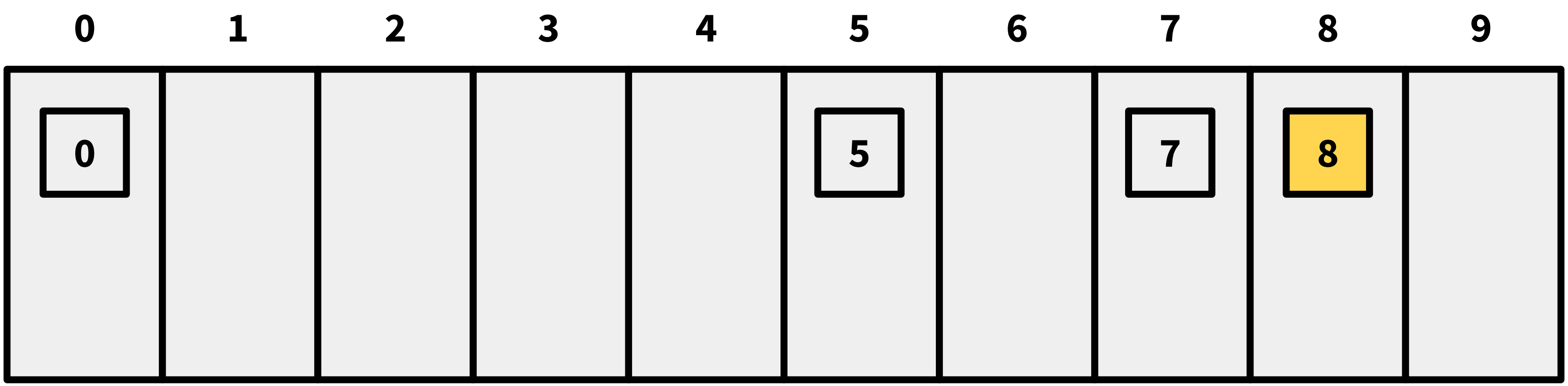


Direct Addressing

How might we get `O(1)`-time? Try direct addressing!

One item per address.

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

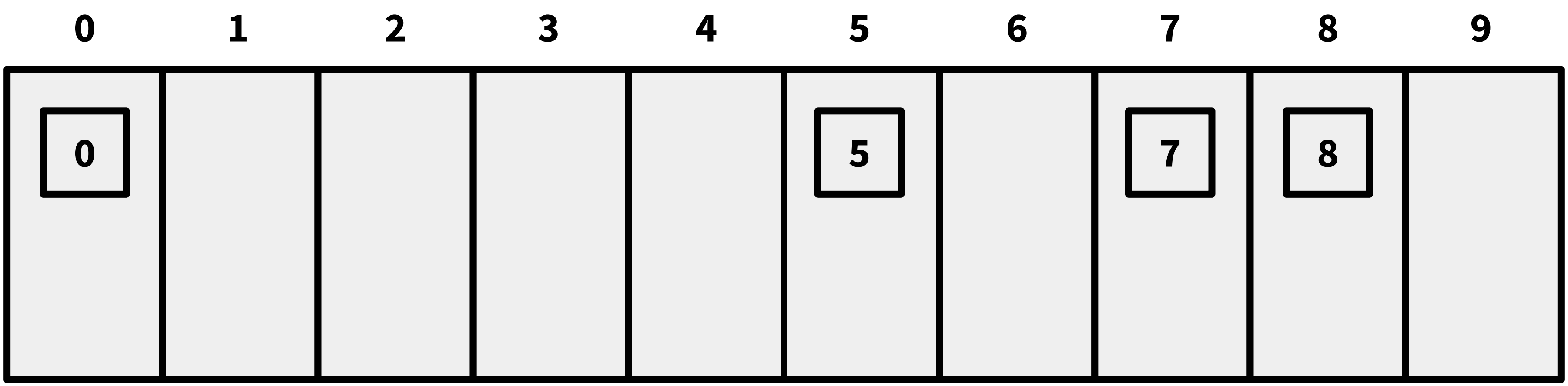


Direct Addressing

How might we get **O(1)**-time? Try direct addressing!

One 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!

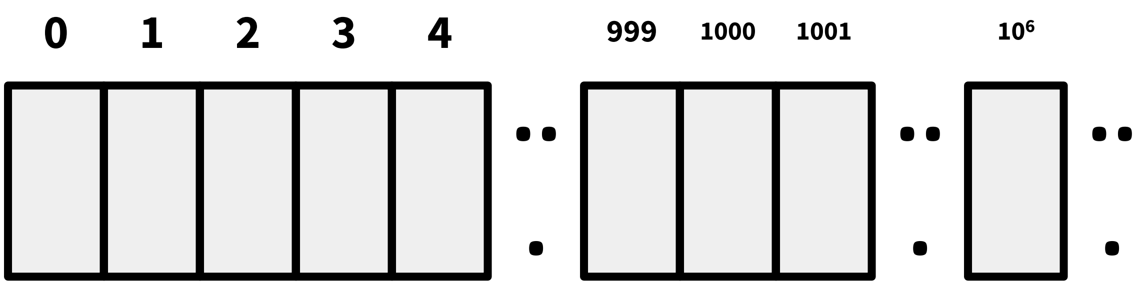
- What's the issue with this approach? 🤔

Direct Addressing

How might we get `O(1)`-time? Try direct addressing!

What's the issue with this approach? 🤔

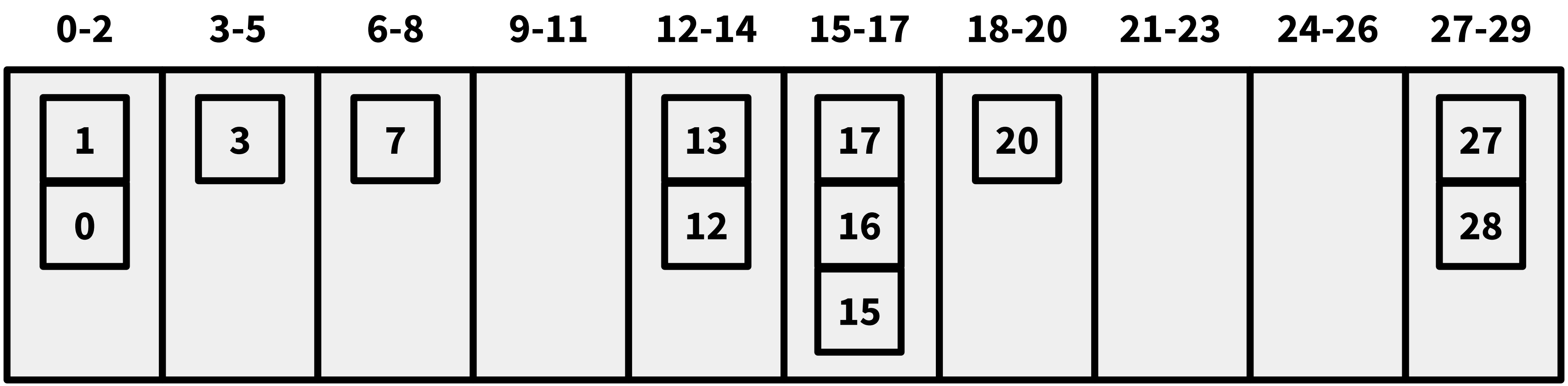
Similar to `counting_sort`, if the set of items being inserted/deleted is large (e.g. $\{0, 1, 2, \ldots, 999, 1000, \ldots, 10^6, \ldots\}$),
then the sheer space required to maintain this data structure becomes an issue.



Direct Addressing

How might we get `O(1)`-time? Try direct addressing!

- Can we fix this issue by assigning multiple items per address, like case (2) of `bucket_sort`?
- Sometimes, this binning approach is useful. `search(12)` still runs pretty fast.

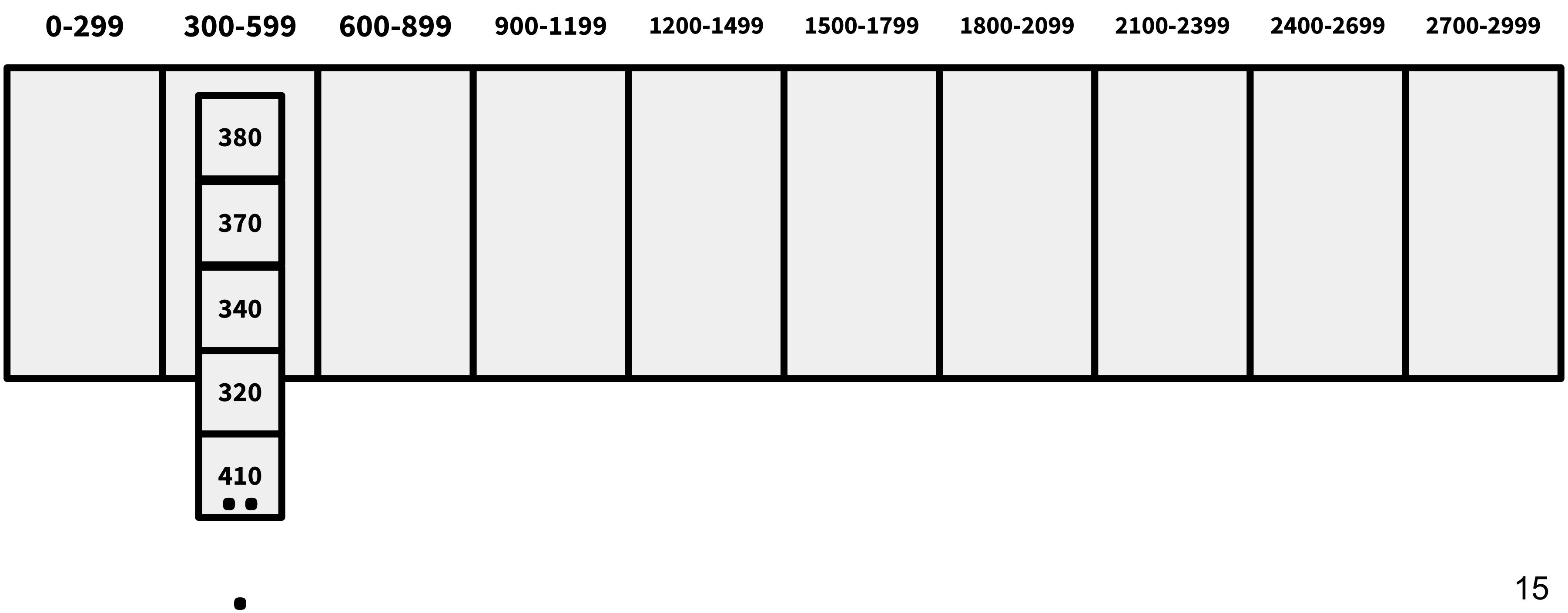


Direct Addressing

How might we get `O(1)`-time? Try direct addressing!

Can we fix this issue by assigning multiple items per address, like case (2) of `bucket_sort`?

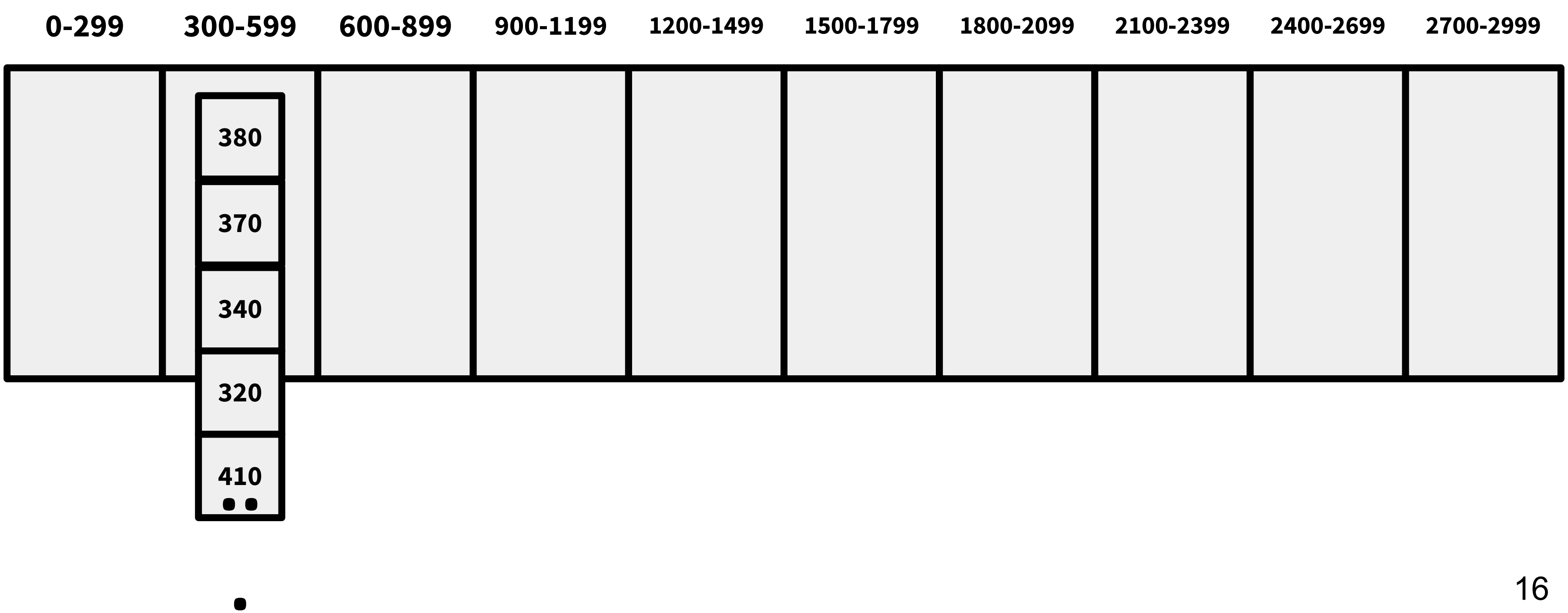
Other times, it causes an issue. `search(432)` is slow.



Direct Addressing

This is an example of a hash table.

- Although one with a very basic bucketing strategy.
- Can we do better?



Terminology

There exists a universe U of keys, size |U|.

- |U| is really big.
- What is |U| if U is the set of ASCII strings of length 16? 🤔

Terminology

There exists a universe U of keys, size |U|.

- |U| is really big.
- What is |U| if U is the set of ASCII strings of length 16? 🤔 $|U| = 128^{16} \approx 5.2 \times 10^{33}$

Terminology

There exists a universe U of keys, size |U|.

- |U| is really big.
- What is |U| if U is the set of ASCII strings of length 16? 🤔 $|U| = 128^{16} \approx 5.2 \times 10^{33}$

We hash the keys to n buckets.

- |U| >>> n; i.e. |U| is a lot bigger than n.
- We don't know which of the |U| possible keys we'll need to store;
  - e.g. all the valid twitter sentences
- Unless otherwise stated, let's assume #keys to store ≤ n (for convenience of analysis)

Terminology

There exists a universe U of keys, size |U|.

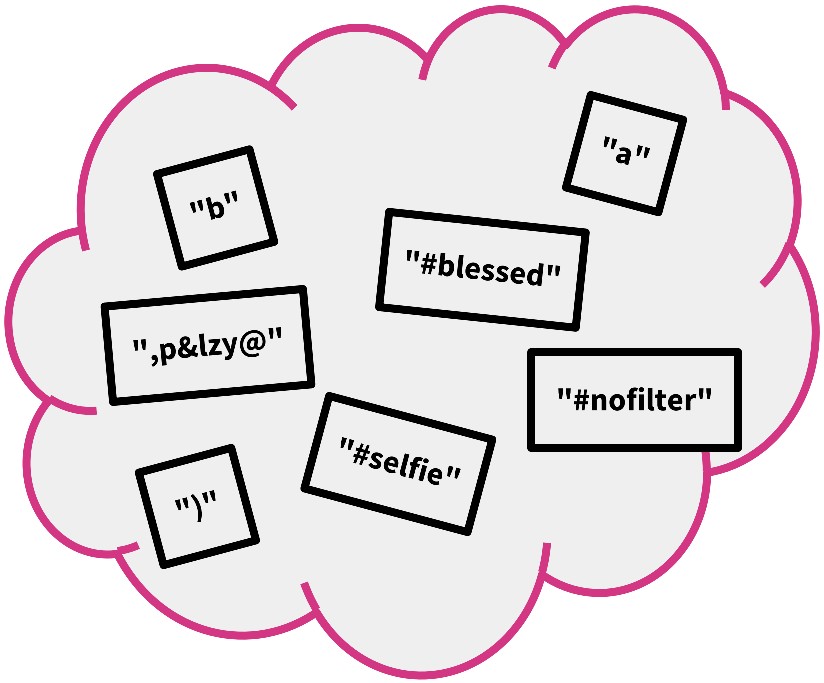
- |U| is really big.
- What is |U| if U is the set of ASCII strings of length 16? 🤔 $|U| = 128^{16} \approx 5.2 \times 10^{33}$

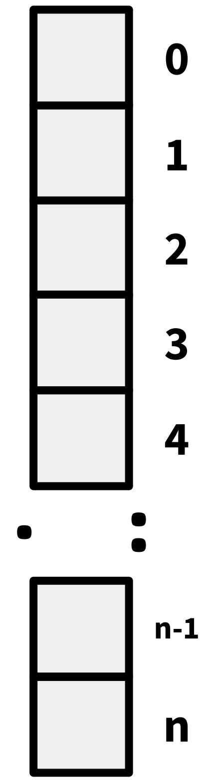
We hash the keys to n buckets.

- |U| >>> n; i.e. |U| is a lot bigger than n.
- We don't know which of the |U| possible keys we'll need to store;
    - e.g. all the valid twitter sentences
- Unless otherwise stated, let's assume #keys to store ≤ n (for convenience of analysis)

There's a hash function $h: U \to \{1, \ldots, n\}$ that maps keys to buckets.

An Example

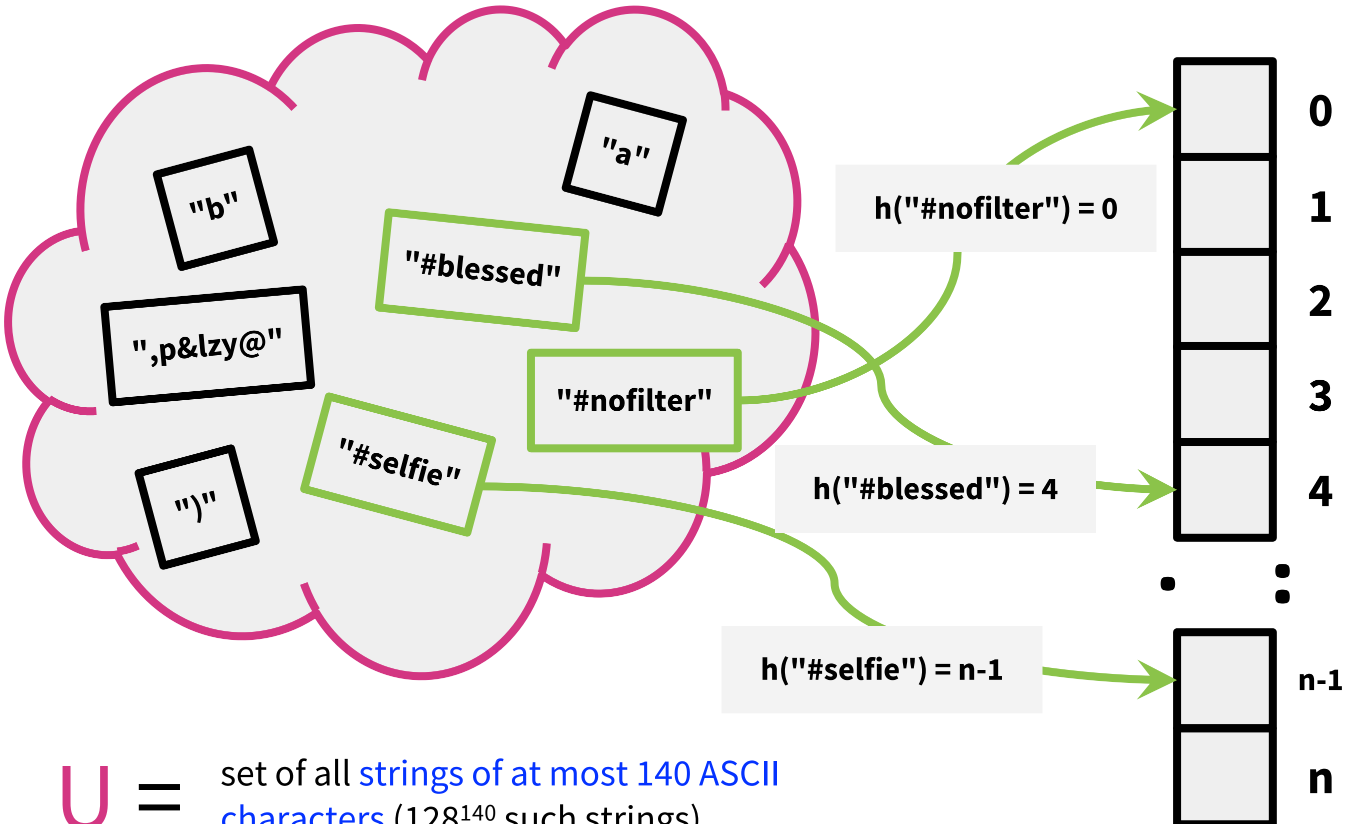




U = set of all strings of at most 140 ASCII characters (128^{140} such strings)

And we'll need to store a small subset of U (say, the ones that might be trending hashtags on Twitter); we're assuming the number of hashtags ≤ n, the number of buckets.

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); we're assuming the number of hashtags ≤ n, the number of buckets.

Hash Tables (with chaining)

List of n buckets.

Each bucket stores an unsorted linked list.

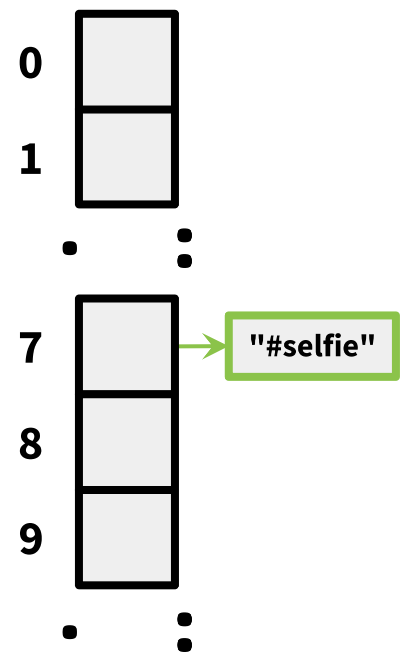
- `insert` in `O(1)` since it's unsorted; `search` in `O(n)`.

h: U → {1, …, n} can be any function

- For example, suppose it's `length`.

Suppose we `insert` a bunch of keys and then `search`.

```
insert("#selfie")
```



Hash Tables (with chaining)

List of n buckets.

Each bucket stores an unsorted linked list.

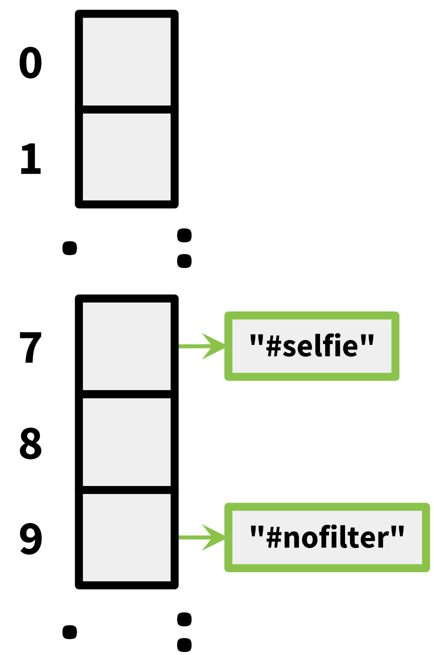
`insert` in `O(1)` since it's unsorted; `search` in `O(n)`.

h: U → {1, …, n} can be any function

For example, suppose it's `length`.

Suppose we `insert` a bunch of keys and then `search`.

```
insert("#selfie")
insert("#nofilter")
```



Hash Tables (with chaining)

List of n buckets.

Each bucket stores an unsorted linked list.

`insert` in `O(1)` since it's unsorted; `search` in `O(n)`.

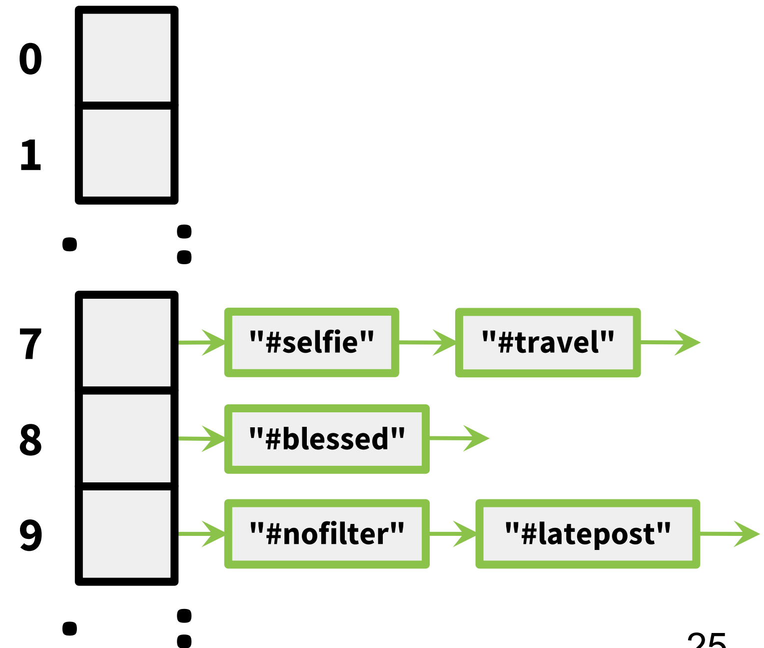
h: U → {1, …, n} can be any function

For example, suppose it's `length`.

Suppose we `insert` a bunch of keys and then `search`.

```
insert("#selfie")
insert("#nofilter")
insert("#blessed")
insert("#travel")
insert("#latepost")
search("#travel")
```

Scans through all elements in bucket h("#travel")



Hash Tables (with chaining)

Is choosing h: U → {1, …, n} to be `length` a good idea? 🤔

Hash Tables (with chaining)

Is choosing h: U → {1, …, n} to be `length` a good idea? 🤔

- Not really. In fact, it's quite terrible since there are a lot of hashtags that share the same lengths.

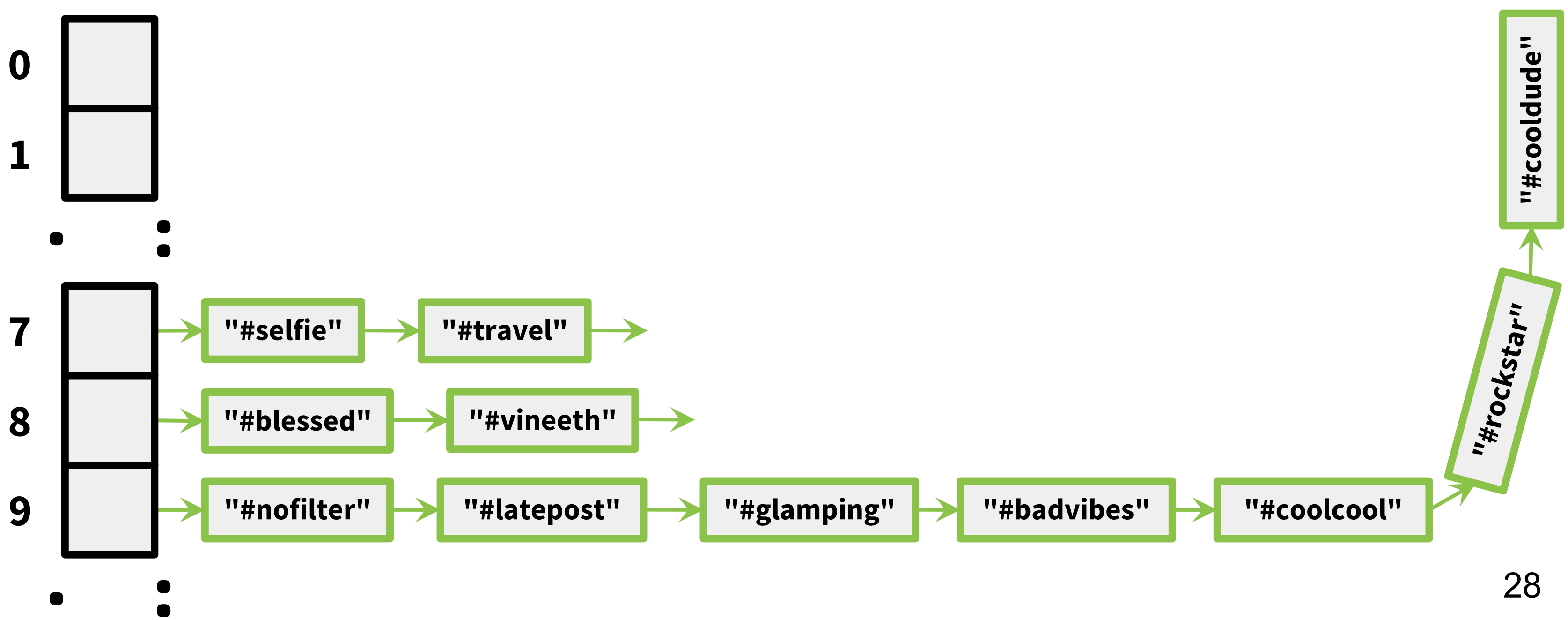
Hash Tables (with chaining)

Is choosing h: U → {1, …, n} to be `length` a good idea? 🤔

Not really. In fact, it's quite terrible since there are a lot of hashtags that share the same lengths.

So how do we choose a better h?

The items need to be spread out in the buckets.



Designing Hash Functions

One h to Rule Them All?

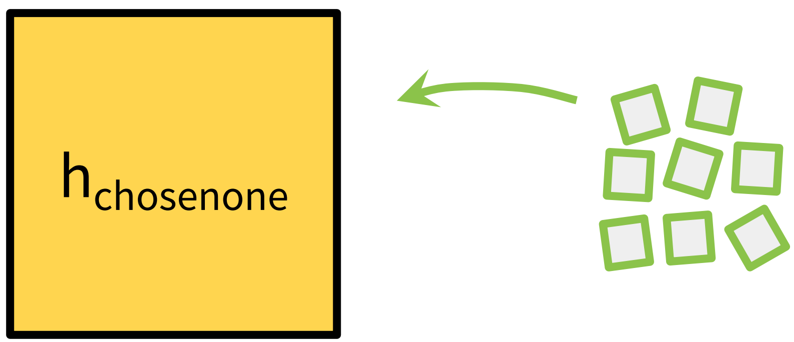
(1) Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) after hashing any n items?

One h to Rule Them All?

(1) Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) ~~after hashing any n items?~~ after an adversary chooses n items to hash?

One h to Rule Them All?

(1) Can we design a single $h_{chosenone}$: $U \rightarrow \{1, \ldots, n\}$ such that all buckets will have size $O(1)$ ~~after hashing any n items?~~ after an adversary chooses n items to hash?



1. You choose your hash function $h_{chosenone}$.

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

Is it possible to construct $h_{chosenone}$ such that you're guaranteed that all buckets will have size O(1)? This would be ideal. 🤔

One h to Rule Them All?

(1) Can we design a single $h_{chosenone}$: $U \rightarrow \{1, \ldots, n\}$ such that all buckets will have size $O(1)$ ~~after hashing any n items?~~ after an adversary chooses n items to hash?

You probably couldn't think of how. Why not? It's impossible!

One h to Rule Them All?

(1) Can we design a single $h_{chosenone}$: $U \rightarrow \{1, \ldots, n\}$ such that all buckets will have size $O(1)$ ~~after hashing any n items?~~ after an adversary chooses n items to hash?

You probably couldn't think of how. Why not? It's impossible!

$h_{chosenone}$ is defined from a domain of $|U|$ items to a range of n buckets. By the pigeonhole principle, at least one of the buckets receives at least $|U|/n$ items. Recall that $|U| >> n$, so $|U|/n > n$; therefore at least one of the buckets receives at least n items.

Notation indicating $U_{bigbucket}$ is a function of $h_{chosenone}$

Let's call the set of items that get hashed to this bucket $U_{bigbucket}(h_{chosenone})$ where $U_{bigbucket} \subset U$. The adversary could choose to hash n items from $U_{bigbucket}$. This is a valid set of n items, and results in one bucket with all n items, by construction. Therefore, **(1)** is impossible.

One h to Rule Them All?

~~**(1)** Can we design a single $h_{chosenone}$: $U \rightarrow \{1, \ldots, n\}$ such that all buckets will have size $O(1)$ after hashing any n items?~~

Impossible!

(2) Can we design a single $h_{chosenone}$: $U \rightarrow \{1, \ldots, n\}$ such that all buckets will have **expected** size $O(1)$ ~~after hashing any n items?~~ after an adversary chooses n items to hash?

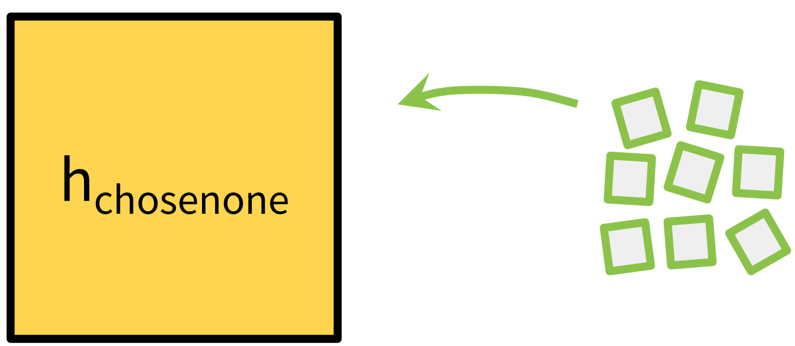
One h to Rule Them All?

~~**(1)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) after hashing any n items?~~

Impossible!

(2) Can we design a single $h_{chosenone}$: U → {1, …, n} such that 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 hash function $h_{chosenone}$.



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

Is it possible to construct $h_{chosenone}$ such that you're guaranteed that all buckets will have expected size O(1)? This would be good. 🤔

One h to Rule Them All?

~~**(1)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) after hashing any n items?~~

Impossible!

(2) Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have **expected** size O(1) ~~after hashing any n items?~~ after an adversary chooses n items to hash?

Can you think of such an $h_{chosenone}$? 🤔

Probably not. This is the same question as **(1)**! Since the adversary is choosing the n items, there's no randomness anywhere in the process.

As a result, for at least one bucket, the **expected** size of the bucket will be trivially just the size.

One h to Rule Them All?

~~**(1)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) after hashing any n items?~~

Impossible!

~~**(2)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have **expected** size O(1) after hashing any n items?~~

Impossible as stated. But if we introduce randomness …

In order for all buckets to have **expected** size O(1) after hashing any n items, we need to introduce randomness.

One h to Rule Them All?

~~**(1)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have size O(1) after hashing any n items?~~

Impossible!

~~**(2)** Can we design a single $h_{chosenone}$: U → {1, …, n} such that all buckets will have **expected** size O(1) after hashing any n items?~~

Impossible as stated. But if we introduce randomness …

In order for all buckets to have **expected** size O(1) after hashing any n items, we need to introduce randomness.

Where? Well there's only one option … in our choice of hash function.

We will randomly choose h from a large set of hash functions!
(There won't be an h to rule them all).

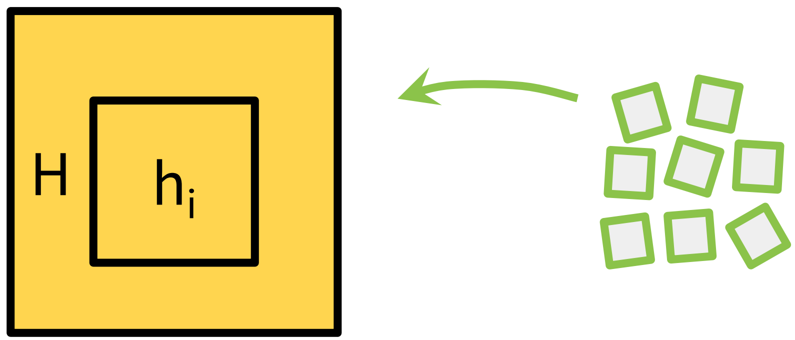
Lots of h's?

(3) Can we design a set $H = \{h_1, \dots, h_k\}$ where h: U → {1, …, 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?

Lots of h's?

(3) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, 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 design 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.

Lots of h's?

(3) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, 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?

Yes! But it's not very useful.

Let H be the set of n hash functions where h_i hashes all keys in the entire universe to bucket i. With probability 1/n, h_b will be chosen, then bucket b will have all the n keys hashed to it. Otherwise, bucket b will be empty.

$$
\begin{aligned}
E[\text{size\_of}(b)] &= P(\text{all keys hashed to it}) \cdot n + P(\text{0 keys hashed to it}) \cdot 0 \\
&= (1/n) \cdot n \\
&= \mathbf{1}
\end{aligned}
$$

But P(lots of keys get hashed to one bucket) = 1.

This is not good. Requiring all buckets to have expected O(1) size is not enough! Maybe we should be using a different metric.

Lots of h's?

(3) Can we design a set $H = \{h_1, \ldots, h_k\}$ where h: U → {1, …, 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?

Not useful!

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where h: U → {1, …, n}, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

Lots of h's?

(3) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, 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?

Not useful!

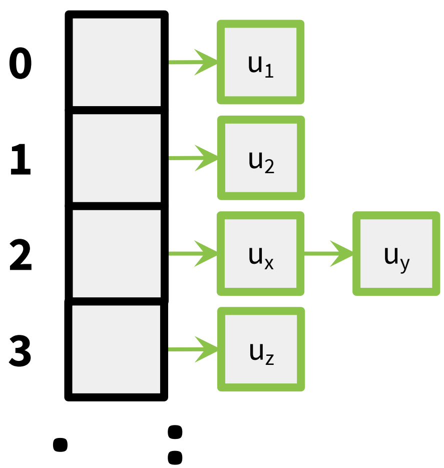
(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

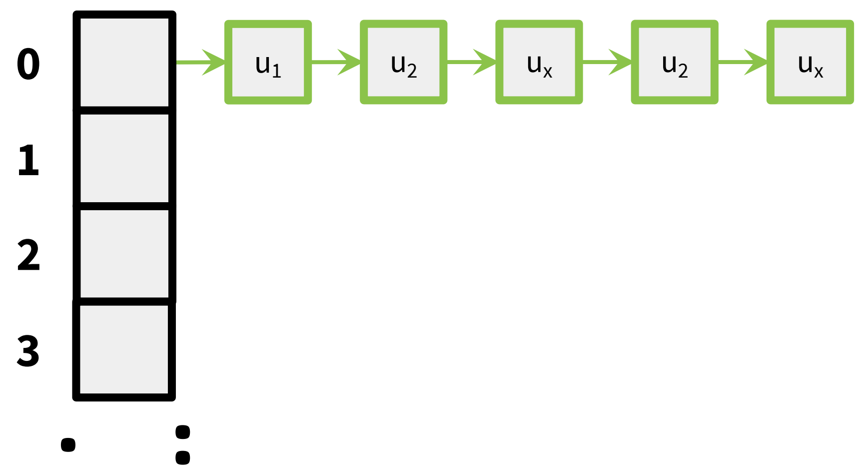
As an analogy for the difference between **(3)** and **(4)**, consider the "small classes illusion." Suppose a university offers 10 classes, 9 of which have 1 person in them and the last of which has 500 persons in them. Using reasoning from **(3)**, the university might tout average class sizes of ~50, when in reality, it should report much higher class sizes experienced by the average student, as in **(4)**.

Lots of h's?

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is $O(1)$?

We can think of this statement in terms of minimizing the expected number of collisions.

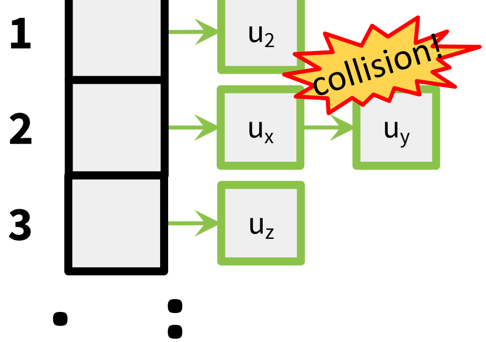


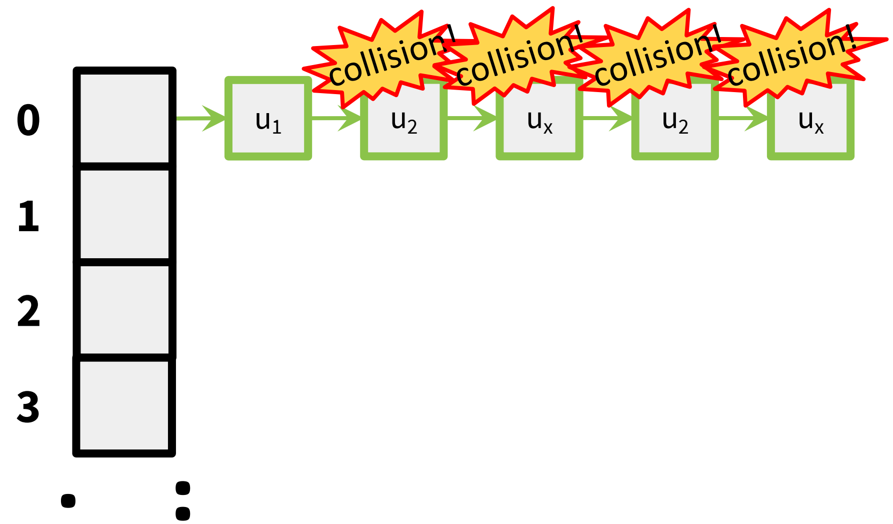


Lots of h's?

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is $O(1)$?

We can think of this statement in terms of minimizing the expected number of collisions.





Lots of h's?

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where h: U → {1, …, n}, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

Yes! This time it's possible.

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

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

Lots of h's?

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

Yes! This time it's possible.

Let H be the exhaustive set of all hash functions that map elements in the universe U to buckets 1 to n, which has size $|H| = n^{|U|}$.

e.g. Suppose U = {"a", "b", "c"} and n = 2 (there are 2 buckets). H would be a set of 8 hash functions. One h would map "a", "b", and "c" all to bucket 0. Another h would map "a" and "b" to bucket 0 and "c" to bucket 1. etc. etc.

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

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

Lots of h's?

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \to \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

$$E[\text{number of items in } u_x\text{'s bucket}] = \sum_{y=1}^{n} P[h(u_x) = h(u_y)]$$

$$= 1 + \sum_{y \neq x} P[h(u_x) = h(u_y)]$$

$$= 1 + \sum_{y \neq x} 1/n$$

Percentage of hash functions that hash u_x and u_y to the same bucket: $P[h(u_x)=h(u_y)]=n/n^2=1/n$

This is also the probability of a collision!

$$= 1 + (n-1/n)$$

$$\leq 2$$

The Good News

(4) Can we design a set $H = \{h_1, \ldots, h_k\}$ where $h: U \rightarrow \{1, \ldots, n\}$, such that if we chose a random h in H, after an adversary chooses n items $\{u_1, \ldots, u_n\}$ to hash, the **expected** number of items in u_x's bucket is O(1)?

Yes! This is great news! It means that we can choose H to be the exhaustive set of all hash functions, and the `insert`, `delete`, `search` operations on any n elements will have an expected runtime of O(1) per operation.

The Bad News

The exhaustive set of all hash functions is HUUUGE!!!

Like really huge.

How many bits would it take to represent one of the $n^{|U|}$ hash functions in this H? 🤔

The Bad News

The exhaustive set of all hash functions is HUUUGE!!!

Like really huge.

How many bits would it take to represent one of the $n^{|U|}$ hash functions in this H? 🤔 $\log n^{|U|} = |U| \log n$.

To see why, consider how much memory it would take to write down the name of one of the 8 hash functions from earlier. You could assign h_1 the id 000, h_2 the id 001, etc. So 8 hash functions requires $\log 8 = 3$ bits to write down.

$|U| \log n$ bits is even enough to do direct addressing! So it's pointless to spend efforts for hashing.

H Is Too Big

How can we fix this issue of the size of H?

Universal Hash Functions

H Is Too Big

How can we fix this issue of the size of H?

Pick from a smaller set H, that still guarantees **(4)**.

Recall the bound that allowed us to achieve this guarantee:

$$E[\text{number of items in } u_x\text{'s bucket}] = \sum_{y=1}^{n} P[h(u_x) = h(u_y)]$$

$$= 1 + \sum_{y \neq x} P[h(u_x) = h(u_y)]$$

$$= 1 + \sum_{y \neq x} 1/n$$

$$= 1 + (n\text{-}1/n)$$

$$\leq 2$$

This step is the key!

Percentage of hash functions that hash u_x and u_y to the same bucket: $P[h(u_x)=h(u_y)]=n/n^2=1/n$

This is also the probability of a collision!

Universal Hash Family

This bound is so important, there's a special name for sets H that satisfy it.

A **hash family** is a fancy name for a set of hash functions.

A **universal hash family** describes a set of hash functions that satisfy the bound: $P_{h \in H}[h(u_x) = h(u_y)] \leq 1/n$, i.e., the probability of collision is bounded by 1/n.

The exhaustive set of hash functions is an example of a universal hash family but, as discussed previously, it's too big to be practical.

A Smaller Universal Hash Family

Identifying new smaller universal hash families is an active field of research in Computer Science, especially in Cryptology.

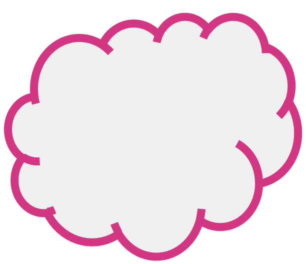
One of the well-studied universal hash families:

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

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

for some prime $p \geq |U|$ and $a \in \{1, \ldots, p - 1\}$ and $b \in \{0, \ldots, p - 1\}$

To select an $h_{a,b}$ from this family:



p

a

b

1. **Determine |U|.**
   e.g. 100, x in 0~99
2. **Find the smallest prime p ≥ |U|.**
   e.g., 101
3. **Let a be a random number in {1, …, p - 1}.**
   e.g., 10
4. **Let b be a random number in {0, …, p - 1}.**
   e.g. 5

Show an example on board: $h_{10,5}(x) = ((10x+5)\ mod\ 101)\ mod\ 10$

How Small Is This H?

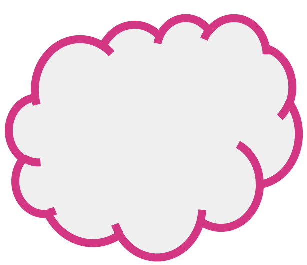
There are p-1 choices for **a** and p choices for **b**,
so $|H| = p(p-1) = O(p^2) = O(|U|^2)$.

(the last step is based on the prime gap theorem)

That's much better than $n^{|U|}$.

The number of bits need to store an h is $\log |U|^2 = O(\log |U|) << O(|U| \log n)$.

Why is it a universal hash family?
Briefly explain P(h(x)-h(y)=0) = P((a(x-y) *mod* p) *mod* n = 0) = 1/n



p **a** **b**

1. **Determine |U|.**
   e.g. 100, x in 0~99
2. **Find the smallest prime p ≥ |U|.**
   e.g., 101
3. **Let a be a random number in {1, …, p - 1}.**
   e.g., 10
4. **Let b be a random number in {0, …, p - 1}.**
   e.g. 5

Show an example on board: $h_{10, 5}(x)$ = ((10x+5) *mod* 101) *mod* 10

Another Universal Hash Family

Another of the well-studied universal hash families (using matrix multiplication!):

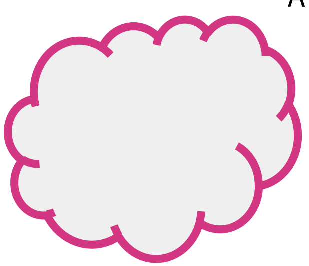
To hash a u-bit string x (i.e. bit string of length u) to a bucket {1, …, n} (i.e. bit string of length $b = \log(n)$)

E.g., hash 8-bit strings like 10100110 to 4 buckets, each bucket is represented by a 2-bit string like 01.

$h_A(x) = (Ax)\ \mathit{mod}\ 2$

for some $b \times u$ (e.g. 2 x 8) matrix A of 0's and 1's, using binary (modulo 2) arithmetic.

To select an h_A from this family:



1. Determine |U|.

u

2. u = log(|U|).

b

3. b = log(n).

A

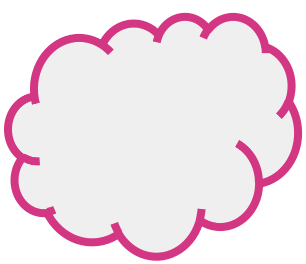
4. Let A be a $b \times u$ random matrix of 0's and 1's.

How Small Is This H?

How many possible binary matrices of size $b \times u$ for **A**?

$2^{ub} = O(|U|^{\log(n)})$.

That's much better than $n^{|U|}$, but larger than the other universal hash family $O(|U|^2)$.



1. Determine |U|.

u

2. u = log(|U|).

b

3. b = log(n).

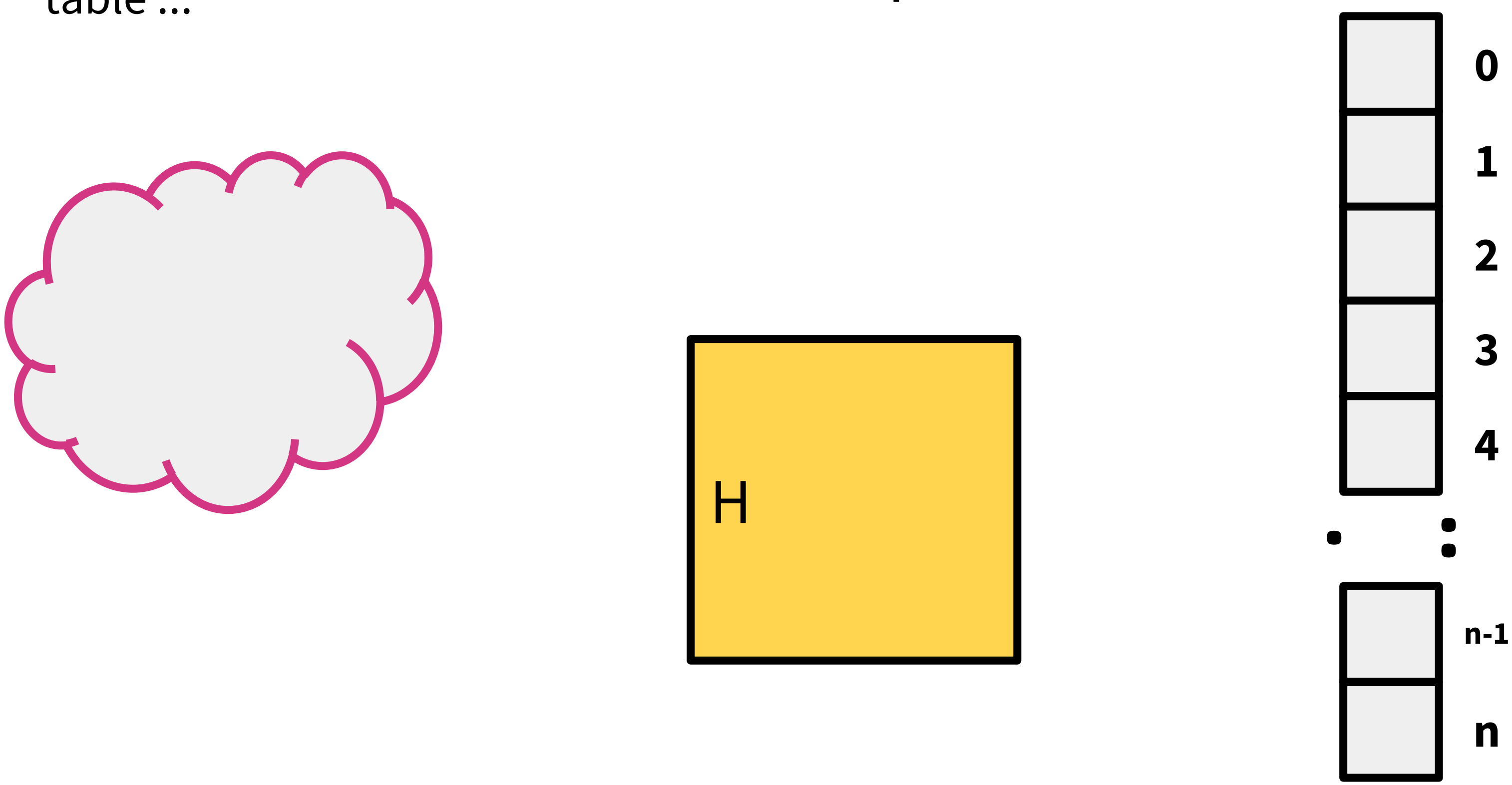
A

4. Let A be a $b \times u$ random matrix of 0's and 1's.

Hash Tables

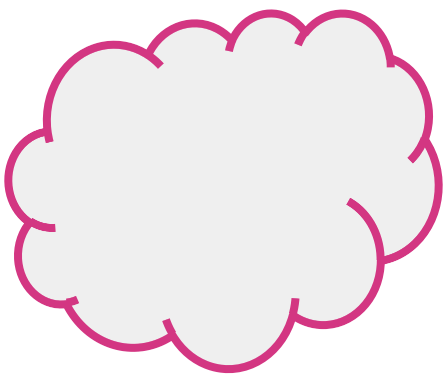
Let's say you wanted to implement a hash table ...

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



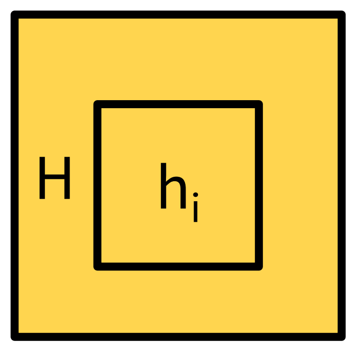
Hash Tables

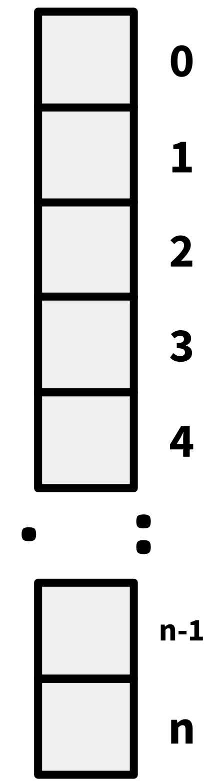
Let's say you wanted to implement a hash table ...



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

2. When the client initializes a hash table, you randomly pick a hash function h_i from H to use in the hash table to hash the items.



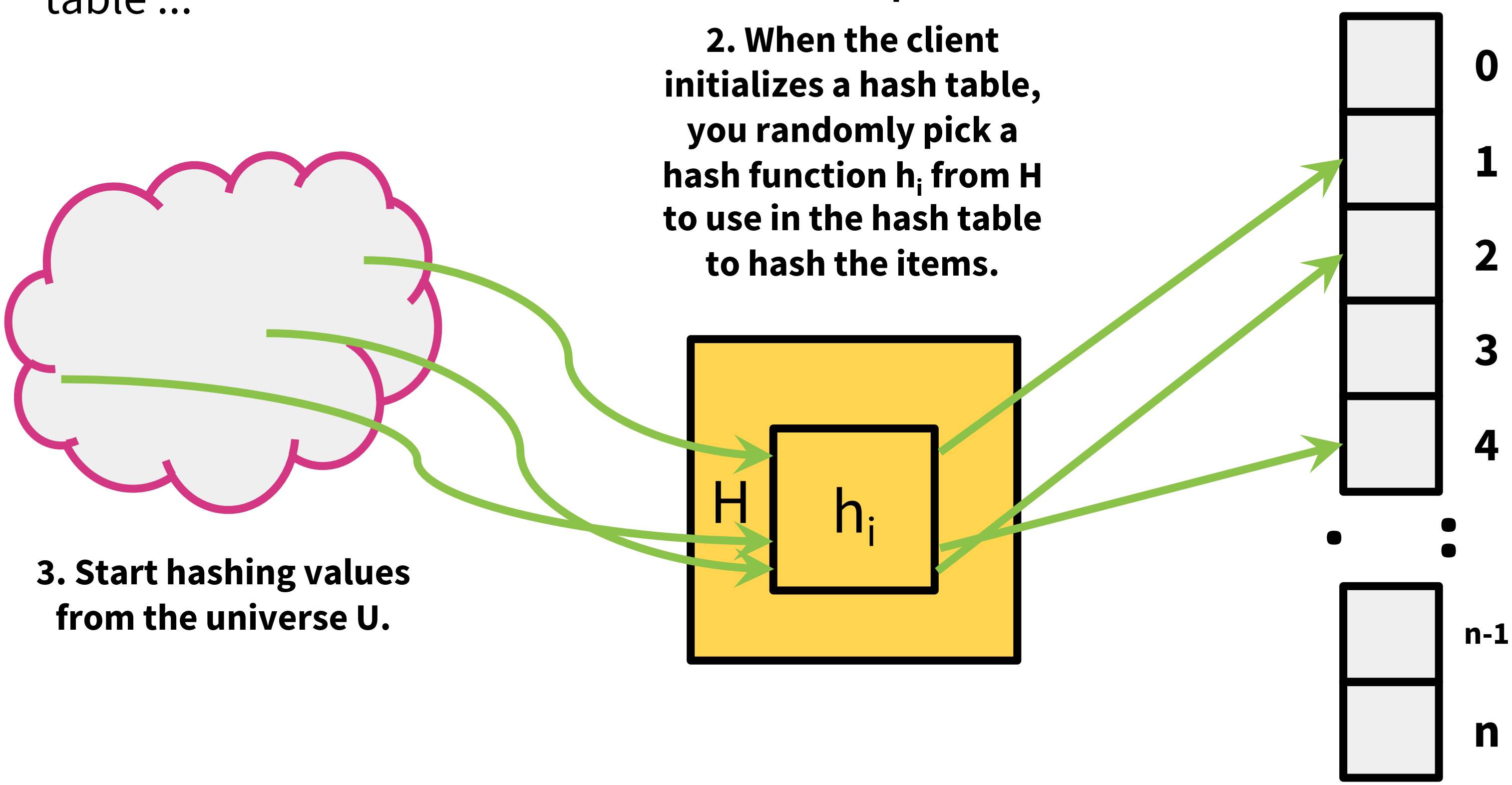


Hash Tables

Let's say you wanted to implement a hash table ...

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

2. When the client initializes a hash table, you randomly pick a hash function h_i from H to use in the hash table to hash the items.



3. Start hashing values from the universe U.

What's the Source of the Randomness?

As was the case in `quicksort`, we want the average-case runtime for a specific input to be low.

- This is why it was important for us to select our pivot randomly as opposed to select, say, the first element in the sublist in `quicksort`.

What's the Source of the Randomness?

As was the case in `quicksort`, we want the average-case runtime for a specific input to be low.

- This is why it was important for us to select our pivot randomly as opposed to select, say, the first element in the sublist in `quicksort`.
- If we selected the pivot as the first element, then the expected runtime over all of the inputs would be low, but for certain inputs (namely, the one in reverse order), runtime would be guaranteed to be high.

What's the Source of the Randomness?

As was the case in `quicksort`, we want the average-case runtime for a specific input to be low.

- This is why it was important for us to select our pivot randomly as opposed to select, say, the first element in the sublist in `quicksort`.
- If we selected the pivot as the first element, then the expected runtime over all of the inputs would be low, but for certain inputs (namely, the one in reverse order), runtime would be guaranteed to be high.
- Instead, since our algorithm supplied the randomness, for any specific input (even the one in reverse order), the expected runtime is low.

What's the Source of the Randomness?

As was the case in `quicksort`, we want the average-case runtime for a specific input to be low.

- This is why it was important for us to select our pivot randomly as opposed to select, say, the first element in the sublist in `quicksort`.
- If we selected the pivot as the first element, then the expected runtime over all of the inputs would be low, but for certain inputs (namely, the one in reverse order), runtime would be guaranteed to be high.
- Instead, since our algorithm supplied the randomness, for any specific input (even the one in reverse order), the expected runtime is low.
- Note this does not say anything about the worst-case runtime, which you can think of as the case in which the adversary controls the randomness we're using. This remains the same even though we introduced randomness into our algorithm.

What's the Source of the Randomness?

As was the case in `quicksort`, we want the average-case runtime for a specific input to be low.

- This is why it was important for us to select our pivot randomly as opposed to select, say, the first element in the sublist in `quicksort`.
- If we selected the pivot as the first element, then the expected runtime over all of the inputs would be low, but for certain inputs (namely, the one in reverse order), runtime would be guaranteed to be high.
- Instead, since our algorithm supplied the randomness, for any specific input (even the one in reverse order), the expected runtime is low.
- Note this does not say anything about the worst-case runtime, which you can think of as the case in which the adversary controls the randomness we're using. This remains the same even though we introduced randomness into our algorithm.

Same thing here with hash tables.

Summary

Randomized Algorithms

Hashing Basics and Terminology
Designing Hash Functions
Universal Hash Functions

Summary

Randomized Algorithms

Hashing Basics and Terminology
Designing Hash Functions
Universal Hash Functions

Acknowledgement: Part of the materials are adapted from Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.