







7/6 Lecture Agenda

- **Announcements**
- Part 3-1: Hash Tables and Universal Hashing
- 10 minute break!
- Part 3-2:  Bloom  Filters 

Announcements!

- HW2 Problem 5 (distributed median-finding) had some issues
 - Runtime for (a) was updated a couple days ago, method still the same
 - Part (b) is too broken, anyone who submits HW2 gets the 2 points (but see my comments in the template about what I intended!)
 - I've extended the HW2 deadline by a day
- Please check out the HW2 template if you haven't looked in a while – many small helpful updates
- Gradescope had some issues with C++, fixed now (thanks, Lucas/Ricky!)
 - **For all coding problems this quarter**, no late days will be charged, but the final late deadline is still tight.
- Pre-HW3 out tonight
 - Includes a quick feedback form / check on how things are going

7/6 Lecture Agenda

- Announcements
- Part 3-1: Hash Tables and Universal Hashing
- 10 minute break!
- Part 3-2:  Bloom  Filters 

WORLD 8-1

Hash for that Cash



Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

Finally, it's Indy's favorite data structure!

- Hash tables are stereotypically *but not actually* part of the answer to every technical interview question.
- First, let's look at them from the user's perspective. Then let's look under the hood.

Hash table guarantees

we'll explain this later



- On average, you can:
 - Insert an element in $O(1)$ time.
 - Query an element in $O(1)$ time.
 - Delete an element in $O(1)$ time.
- The stored elements can be just keys (in which case we have a set), or key-value pairs (in which case we have an associative array, AKA dictionary)
 - The values just come along for the ride!

Indy's warm-up problem: 2-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any two that sum to some given value k .

Indy's warm-up problem: 2-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any two that sum to some given value k .
- The solution:
 - Make one pass through the list and put everything in a hash table.
 - Go through the list again. For each element e_i , find $k - e_i$ and see if it's in the table. If so, return True. If this never happens, return False.
 - also make sure not to use the same element twice

Indy's warm-up problem: 2-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any two that sum to some given value k .
- The solution:
 - Make one pass through the list and put everything in a hash table. $O(1) * n = O(n)$
 - Go through the list again. For each element e_i , find $k - e_i$ and see if it's in the table. If so, return True. If this never happens, return False. $(O(1) + O(1)) * n = O(n)$
 - also make sure not to use the same element twice
could, e.g., store counts in the table as values. Still $O(n)$ overall.

Indy's next problem: 3-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any **three** that sum to some given value k .

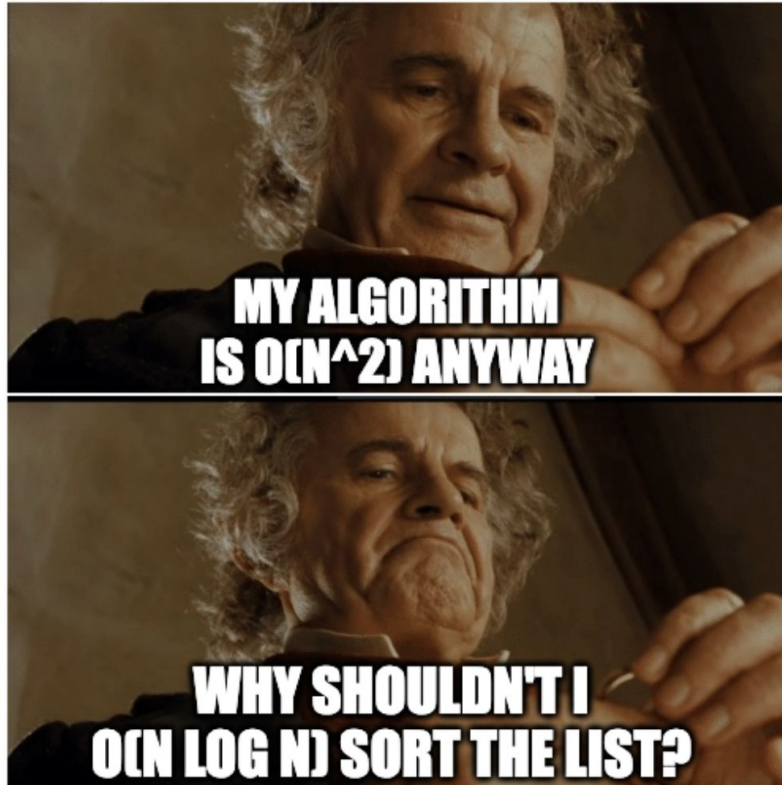
Indy's next problem: 3-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any **three** that sum to some given value k .
- A solution:
 - Make one pass through the list and put everything in a hash table.
 - Go through the list again, but now considering every pair (e_i, e_j) of elements. For each such element, find $k - e_i - e_j$ and see if it is in the hash table. If so, return True. If this never happens, return False.
 - As before, avoid reusing the same element. Now we can throw in an $O(n \log n)$ sort to put duplicates close together?

Indy's next problem: 3-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any **three** that sum to some given value k .
- A solution:
 - Make one pass through the list and put everything in a hash table. $O(1) * n = O(n)$
 - Go through the list again, but now considering every pair (e_i, e_j) of elements. For each such element, find $k - e_i - e_j$ and see if it is in the hash table. If so, return True. If this never happens, return False. $(O(1) + O(1)) * n^2 = O(n^2)$
 - As before, avoid reusing the same element. Now we can throw in an $O(n \log n)$ sort to put duplicates close together?

A useful observation



3-SUM: Can we do better?

- Some people finally did in 2014.
- It's even better now, but in a sense, still not much better than $O(n^2)$.

[linear decision tree](#) complexity of 3SUM is $O(n^{3/2} \sqrt{\log n})$. These bounds were subsequently improved.^{[2][3][4]} The current best known algorithm for 3SUM runs in $O(n^2 (\log \log n)^{O(1)} / \log^2 n)$ time.^[4] Kane, Lovett, and Moran showed that the [6-linear decision tree](#) complexity of 3SUM is $O(n \log^2 n)$.^[5] The latter bound is tight (up to a logarithmic factor). It is still conjectured that 3SUM is unsolvable in $O(n^{2-\Omega(1)})$ expected time.^[6]

Indy's bonus problem: 4-SUM

- Given a list of n (not necessarily distinct) integers, determine whether there are any **four** that sum to some given value k .



What do you think?

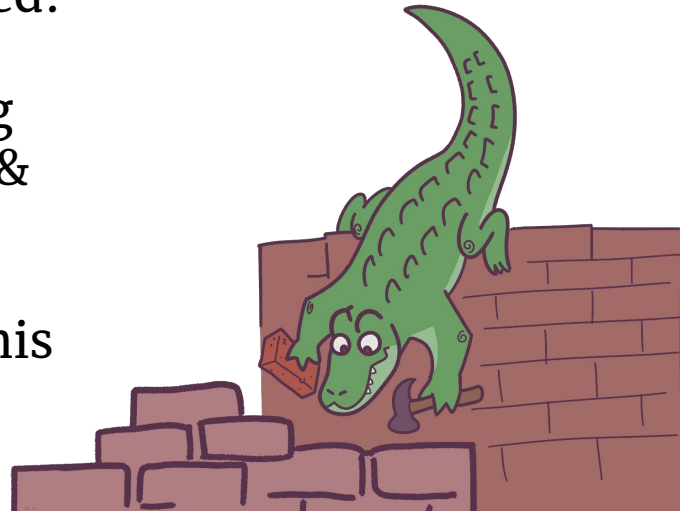
4-SUM solution

- Following the pattern, we'd expect to be stuck at $O(n^3)$, but surprisingly, we can do better – almost as well as for 3-SUM!
- An $O(n^2)$ solution: *possibly with another $\log n$ factor depending on implementation*
 - Make one pass through the list, considering every pair (e_i, e_j) of elements. Find $e_i + e_j$. Put this in as a key and (i, j) as a value.
 - Make another pass through the list, again considering every pair (e_i, e_j) of elements. For each such element, find $k - e_i - e_j$ and see if it is in the hash table. If so, and if there is a non-overlapping set with that sum, return True.
 - If this never happens, return False.

We traded time for space

But in the real world, space can also cost time!

- If the data no longer fits nicely in the L1 cache (or in L2 cache, or...), things could be slow.
 - If it no longer fits in RAM, we're doomed?
- Focusing on improving time while ignoring space is like a college gaming the US News & World Report rankings.
 - That said, we will frequently commit this sin since this is a theory class.



An extreme (and silly) example

As long as the numbers are in the range, say, $[1, 10^9]$, and n is no larger than 10^9 , and k is no larger than $4 * 10^9$, I can solve 4-SUM in $O(1)$ time*

** by precomputing the solutions to all 4 billion * (1 billion)^(1 billion) possible instances of the problem offline first, then just looking up the one we need*

Please don't do this on an exam.

But this isn't totally ridiculous. It sometimes makes sense to precompute all possible values when the number of them is tractable. See CS166 / "The Method of Four Russians" if curious.

So the answer is always hash tables?

- Not really. Be a little careful about jumping right to a hash table solution in an interview problem unless you have at least *some* basis for doing so.
 - Otherwise you may look like Interview Prep Jones
- A hash table is often a useful auxiliary structure in a solution, but not the *entire* focus of a solution.
- What about, say, the Equal Sum problem (can a list be divided into two subsets of equal sum)? For that we'll use **dynamic programming**, not hashing.

OK, but how do they work?

The structure has:

- some number n of "buckets"
- a **hash function** h that maps any input value (in the universe U , e.g., all possible IP addresses) to one of these buckets, and...

OK, but how do they work?

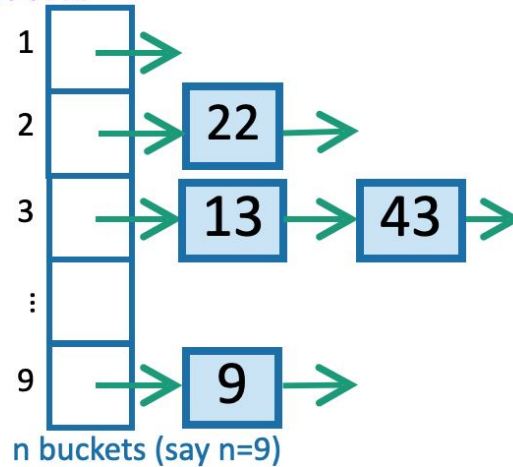
The structure has:

- some number n of "buckets"
- a **hash function** h that maps any input value (in the universe U , e.g., all possible IP addresses) to one of these buckets, and...
 - is deterministic
 - doesn't take too long to evaluate
 - spreads the values out pretty evenly among the buckets (this is critical!)

Hash Tables (with chaining)

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

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



Hash Tables (with chaining)

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

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

INSERT:

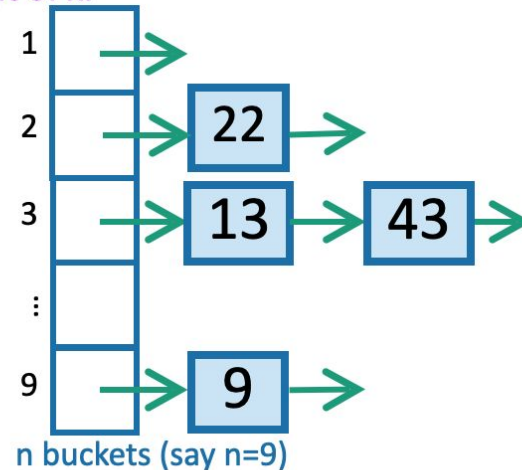


SEARCH 43:

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

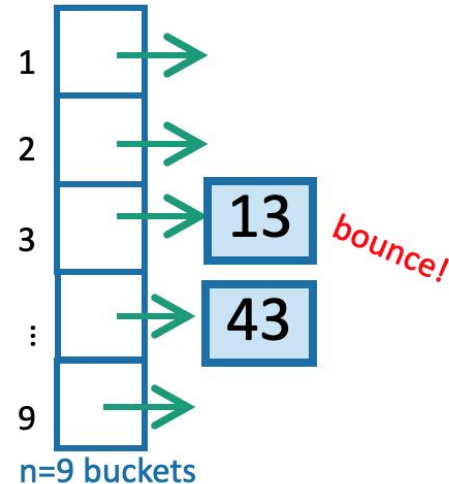
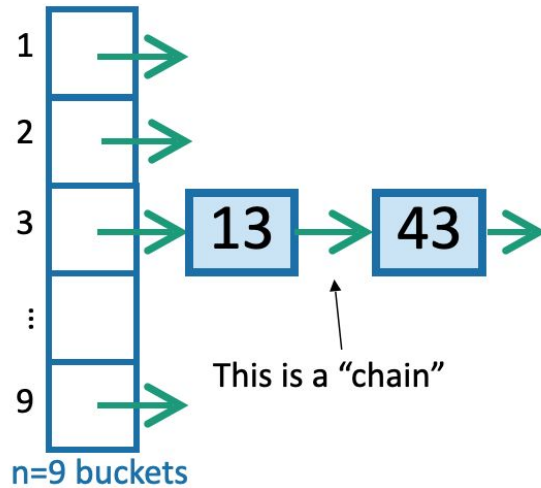
DELETE 43:

Search for 43 and remove it.



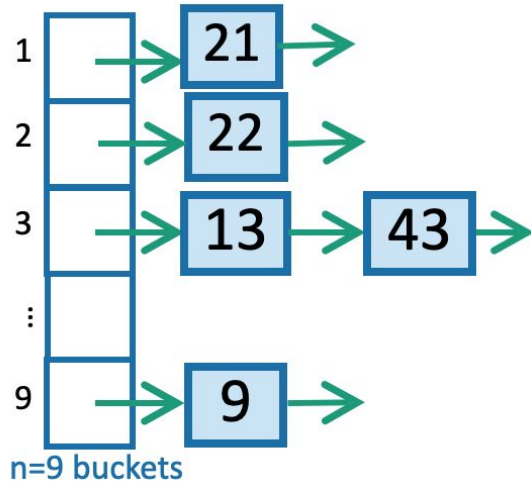
Aside: Hash tables with open addressing

- 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.

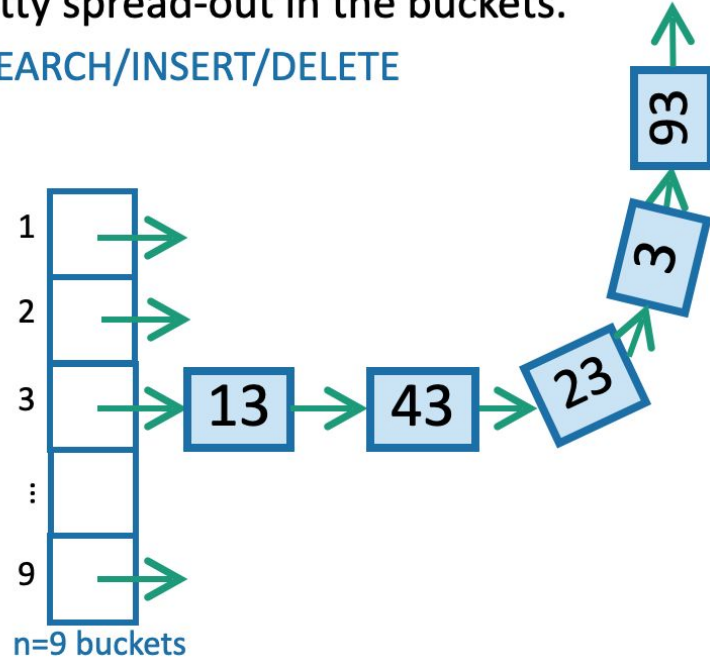


What we want from a hash table

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



vs.



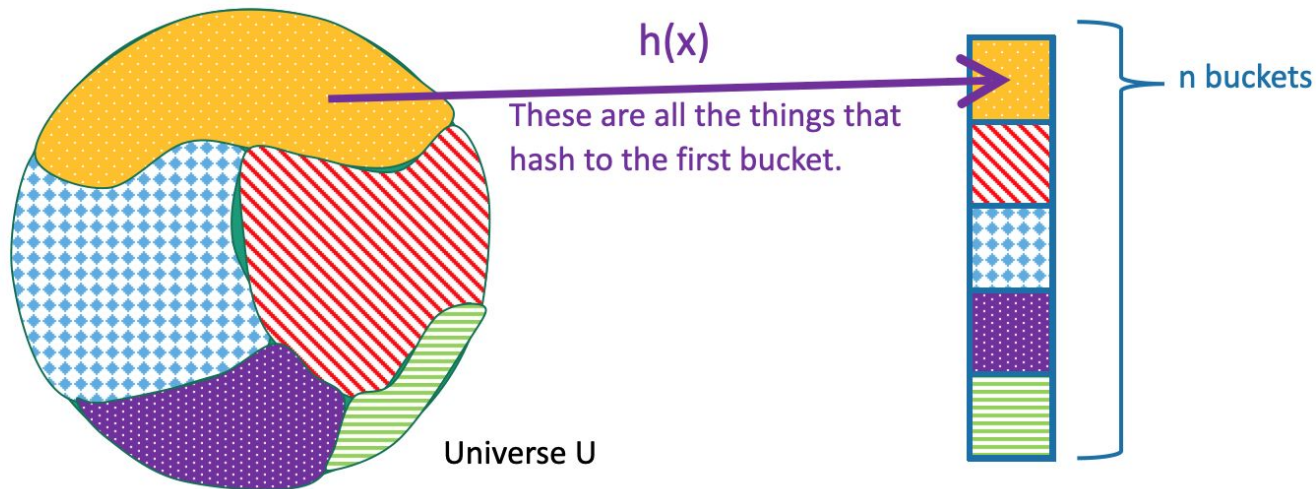
Worst-case analysis

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

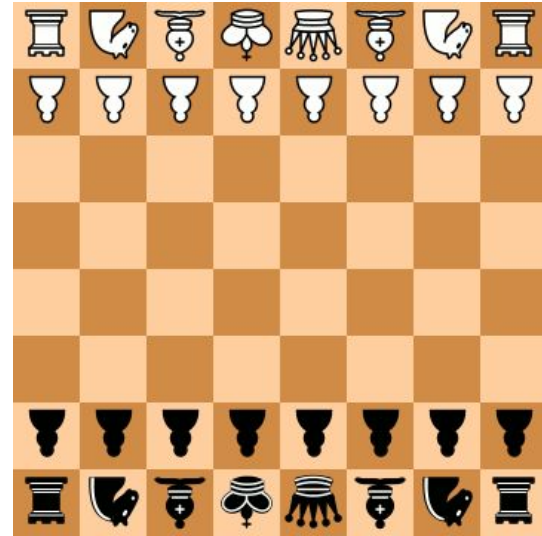


We really can't beat the bad guy here.

- 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.**



Let's do what I did as a petulant child...



...change the rules of the game!

The game

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.

13 22 43 92 7

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



The game

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.

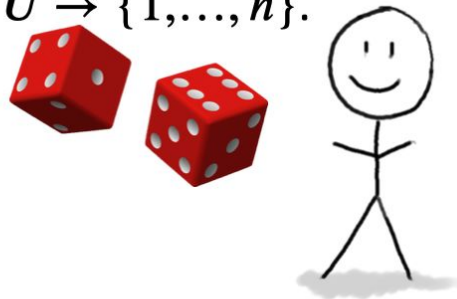
13 22 43 92 7



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

$$h: U \rightarrow \{1, \dots, n\}.$$



The game

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.

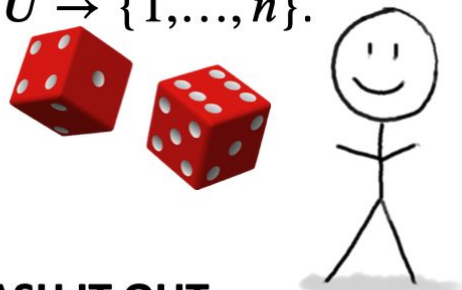


INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92

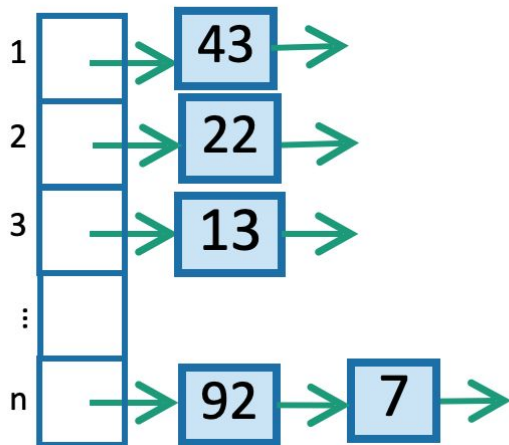


2. You, the algorithm, chooses a **random** hash function


$$h: U \rightarrow \{1, \dots, n\}.$$



3. **HASH IT OUT** #hashpuns



Expected number of items in u_i 's bucket?



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

h is uniformly random

A uniformly random hash function leads to balanced buckets

- We can show:
 - 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 $h: U \rightarrow \{1, \dots, n\}$?
 - 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:

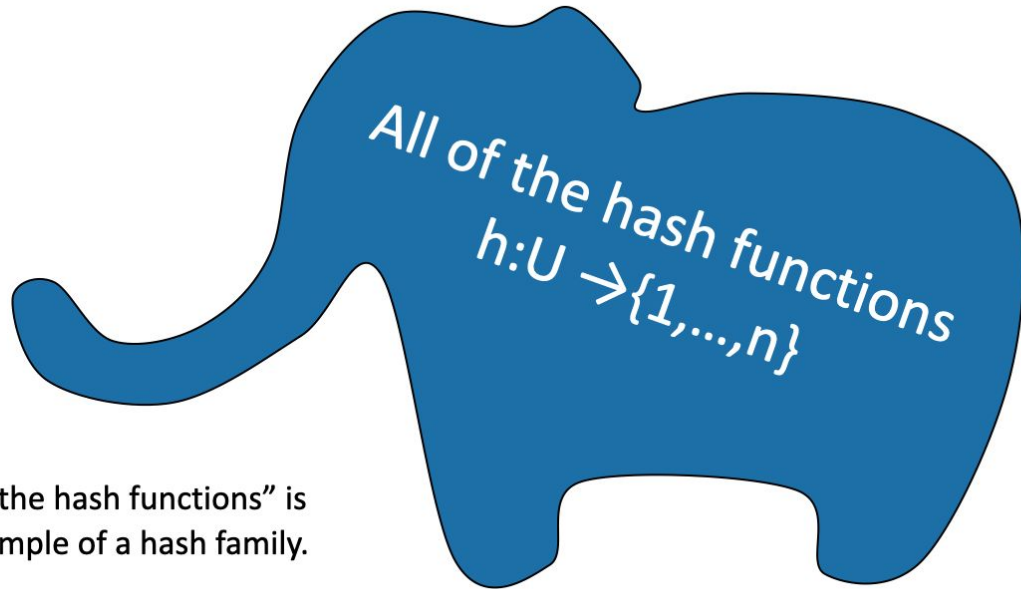
All of the M things in the universe

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.

Hash families

- A hash family is a collection of hash functions.



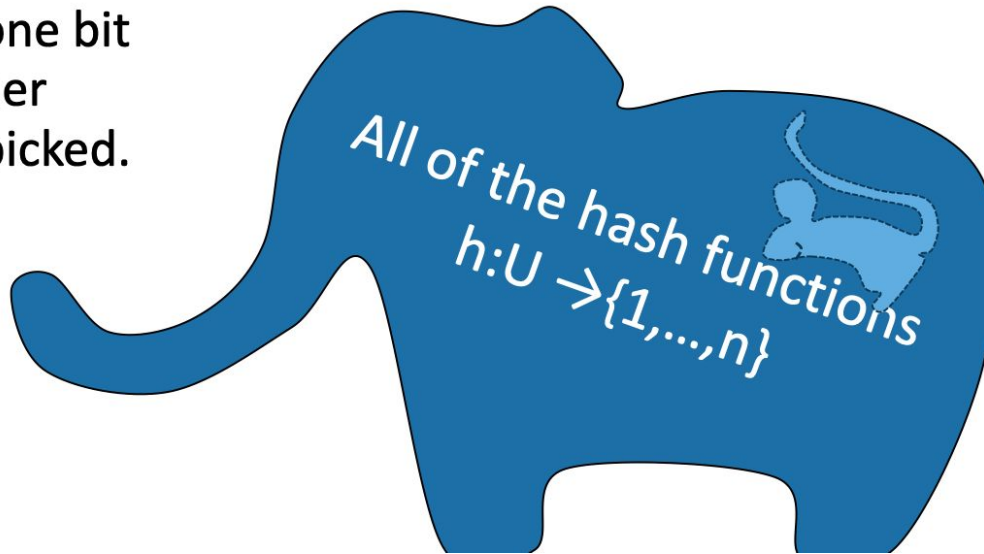
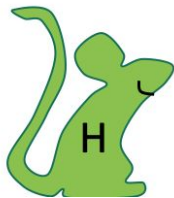
"All of the hash functions" is an example of a hash family.

Example:

a smaller hash family

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

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



The game

$h_0 = \text{Most_significant_digit}$

$h_1 = \text{Least_significant_digit}$

$H = \{h_0, h_1\}$

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.

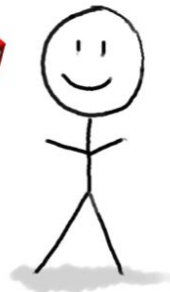


INSERT 19, INSERT 22, INSERT 42,
INSERT 92, INSERT 0, SEARCH 42,
DELETE 92, SEARCH 0, INSERT 92

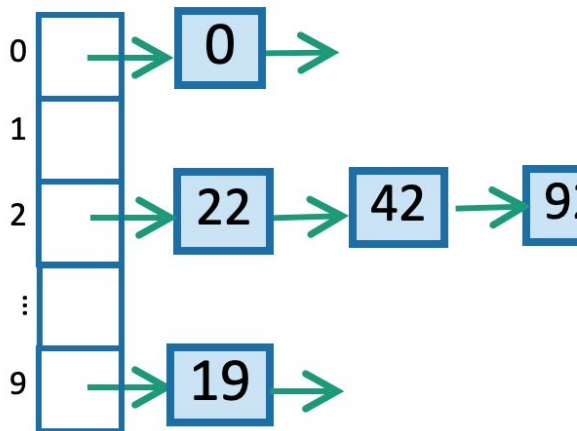
2. You, the algorithm, chooses a **random** hash function $h: U \rightarrow \{0, \dots, 9\}$. Choose it randomly from H .



I picked h_1



3. HASH IT OUT #hashpuns



The game

$h_0 = \text{Most_significant_digit}$

$h_1 = \text{Least_significant_digit}$

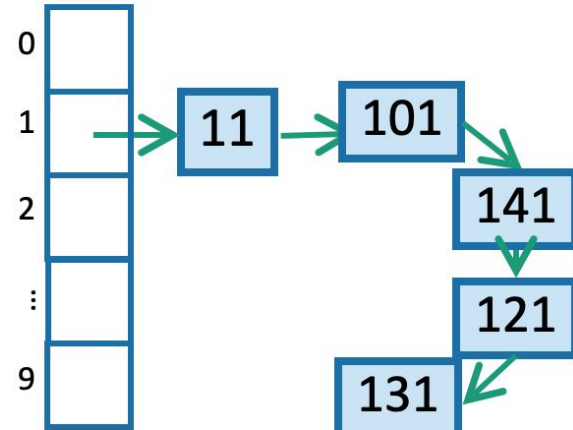
$H = \{h_0, h_1\}$

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 .



INSERT 11, INSERT 101, INSERT 141,
INSERT 121, INSERT 131, SEARCH
131, SEARCH 121, SEARCH 101...

3. HASH IT OUT #hashpuns



Strategy

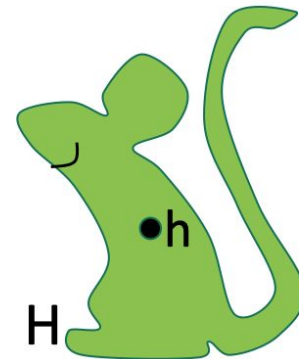
- Pick a small hash family H , so that when I choose h randomly from H ,

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

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

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

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



Universal hash family

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

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

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

Example

- H = the set of all 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

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

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

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

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

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

- Define:

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

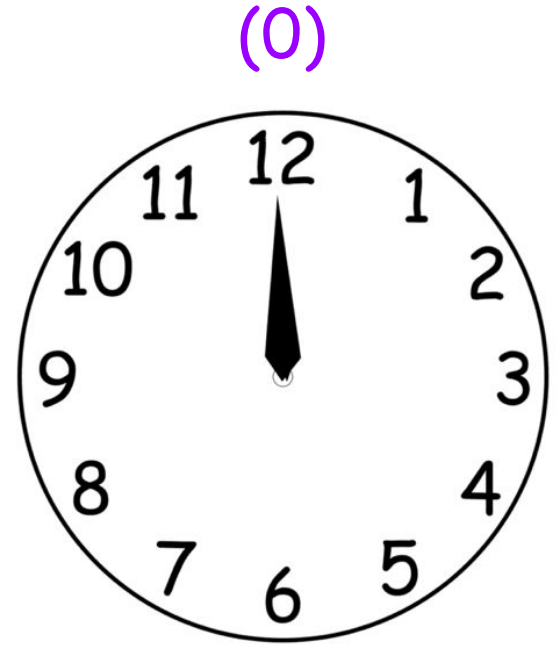
- Claim:

H is a universal hash family.



Modular Arithmetic

- Think of working modulo m as adding times on a clock with m hours.
 - except hour m is labeled 0
- e.g., 9:00 + 5 hours = 2:00
 $(9 + 5) \bmod 12 = 2$
- 2:00 - 5 hours = 9:00
 $(2 - 5) \bmod 12 = 9$



Let's pick $n = 3$ (3 buckets), $M = 4$ (size of universe), $p = 5$. Then:

$$f_{a,b}(x) = (ax + b) \bmod 5$$

$$h_{a,b}(x) = f_{a,b}(x) \bmod 3$$

Our universal hash family is: $H = \{h_{a,b}(x) : a \in \{1, 2, 3, 4\}, b \in \{0, 1, 2, 3, 4\}\}$

Let's pick $n = 3$ (3 buckets), $M = 4$ (size of universe), $p = 5$. Then:

$$f_{a,b}(x) = (ax + b) \bmod 5$$

$$h_{a,b}(x) = f_{a,b}(x) \bmod 3$$

Our universal hash family is: $H = \{h_{a,b}(x) : a \in \{1, 2, 3, 4\}, b \in \{0, 1, 2, 3, 4\}\}$

Looking at just one of the hash functions in this family, $h_{2,4}(x)$:

$$h_{2,4}(x) = f_{2,4}(x) \bmod 3 = ((2x + 4) \bmod 5) \bmod 3$$

What does this do to every value in the universe, $\{0, 1, 2, 3\}$?

- $h_{2,4}(0) = ((2 \cdot 0 + 4) \bmod 5) \bmod 3 = 4 \bmod 3 = 1$
- $h_{2,4}(1) = ((2 \cdot 1 + 4) \bmod 5) \bmod 3 = 1 \bmod 3 = 1$
- $h_{2,4}(2) = ((2 \cdot 2 + 4) \bmod 5) \bmod 3 = 3 \bmod 3 = 0$
- $h_{2,4}(3) = ((2 \cdot 3 + 4) \bmod 5) \bmod 3 = 0 \bmod 3 = 0$

Let's pick $p = 5$, $n = 3$ (3 buckets), $M = 4$ (size of universe). Then:

$$f_{a,b}(x) = (ax + b) \bmod 5$$

$$h_{a,b}(x) = f_{a,b}(x) \bmod 3$$

Our universal hash family is: $H = \{h_{a,b}(x) : a \in \{1, 2, 3, 4\}, b \in \{0, 1, 2, 3, 4\}\}$

Looking at just one of the hash functions in this family, $h_{2,4}(x)$:

$$h_{2,4}(x) = f_{2,4}(x) \bmod 3 = ((2x + 4) \bmod 5) \bmod 3$$

What does this do to every value in the universe, $\{0, 1, 2, 3\}$?

- $h_{2,4}(0) = ((2 \cdot 0 + 4) \bmod 5) \bmod 3 = 4 \bmod 3 = 1$
- $h_{2,4}(1) = ((2 \cdot 1 + 4) \bmod 5) \bmod 3 = 1 \bmod 3 = 1$
- $h_{2,4}(2) = ((2 \cdot 2 + 4) \bmod 5) \bmod 3 = 3 \bmod 3 = 0$
- $h_{2,4}(3) = ((2 \cdot 3 + 4) \bmod 5) \bmod 3 = 0 \bmod 3 = 0$

This looks like a terrible hash function! It's not even using bucket 2...

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

Context

- The yellow cells show how our $h_{2,4}(x)$ hashes the possible inputs 0, 1, 2, 3.
- But we only require that for any two values in the universe, the probability (over the randomness of which of the 20 hash functions we choose) is $\leq 1/3$. Is that true?

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

- The probability of a collision between items 0 and 1 is $4/20 = 1/5$, which is $\leq 1/3$.
- But we have to check all pairs of items...

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

a = v	b =>	0	1	2	3	4	
		0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	0	1 (item 1 in universe)
	2	2	0	1	0	1	
	3	0	1	0	1	2	
	4	1	0	1	2	0	
		0	1	2	3	4	
	1	2	0	1	0	1	2 (item 2 in universe)
	2	1	0	1	2	0	
	3	1	2	0	1	0	
	4	0	1	0	1	2	
		0	1	2	3	4	
	1	0	1	0	1	2	3 (item 3 in universe)
	2	1	2	0	1	0	
	3	1	0	1	2	0	
	4	2	0	1	0	1	

So we're OK!

a = v	b =>	0	1	2	3	4	
	1	0	1	2	0	1	0 (item 0 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	1	2	0	1	2	1 (item 1 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	2	0	1	2	0	2 (item 2 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	
		0	1	2	3	4	
	1	0	1	2	0	1	3 (item 3 in universe)
	2	0	1	2	0	1	
	3	0	1	2	0	1	
	4	0	1	2	0	1	

What if we chose a non-prime? ($p = 12$ here)




So what did we do?

- We exhibited a construction for universal hash families of size only $O(M^2)$, where M is the universe size. (We had to pick $p \geq M$, and a and b can both be as large as $p-1$)
 - All we have to do is pick a, b uniformly at random and the bad guy is foiled!
 - Also, the hashing calculations are fast!
- We did *not* actually prove that this construction works in general, and for that I'll refer you to CLRS.




Takeaways

- Hash tables are awesome and support average-case $O(1)$ insertion, lookup, and deletion
 - and there are lots of other kinds too! If you're curious, look up "cuckoo hashing..."
- Universal hash functions give us a sort of guarantee that there *probably* won't be too many collisions
 - Of course, it could still happen due to bad luck, just like Quicksort's worst-case.
- Designing your own hash functions is like designing your own cryptography: **d o n ' t**
 - **It's too easy to miss a subtle issue.**

7/6 Lecture Agenda

- Announcements
- Part 3-1: Hash Tables and Universal Hashing
- 10 minute break!
- Part 3-2:  Bloom  Filters 

7/6 Lecture Agenda

- Announcements
- Part 3-1: Hash Tables and Universal Hashing
- 10 minute break!
- Part 3-2:  Bloom  Filters 

WORLD 8-2

Bloom Filters: Store Seeds, Not
Whole Plants

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

Motivation

- You are writing a Web browser that knows / remembers which URLs are spammy or malicious.
- How do you keep track of which URLs are bad?

Indy's solution: Put the URLs in a hash table!

- (A hash set as opposed to a hash map.)
- When a URL is reported as bad, put it in the hash table.
- When visiting a new URL, first check to see if it's in the hash table. (If so, do not proceed.)
- What's a big problem with this approach?

- When a URL is reported as bad, put it in the hash table.
- When visiting a new URL, first check to see if it's in the hash table. (If so, do not proceed.)
- What's a big problem with this approach?
 - URLs can change and no longer be spammy: *not a huge problem*. These can be removed from the table.
 - Multiple URLs might hash to the same bucket: *that's fine*. We still check everything in a bucket, so we'll find the right one if it's there.

URLs are big



HugeURL, Because TinyURL Is For Wimps

For instance why would you want to simply link to <http://wired.com>, when you can link to [http://www.hugeurl.com/?

OGMzNjgzNjIOYjQ1MwJlODBjMwIyZWUyNzQ5

YzliZjMmMTMmVmOwd2QyUXlVWGxWV0d4WFIUSm9WMVl3Wkc5V1ZsbDNXa2M1

YWxKcldqQlVWbHBQVjBaYWMySkVUbGhoTVVwVVZtcEdZV015U2tWVWJHaG9U]

(http://www.hugeurl.com/?

OGMzNjgzNjIOYjQ1MwJlODBjMwIyZWUyNzQ5YzliZjMmMTMmVmOwd2QyUXlVWGxWV0d4W

WMVl3Wkc5V1ZsbDNXa2M1YWxKcldqQlVWbHBQVjBaYWMySkVUbGhoTVVwVVZtcEdZV015U

JHaG9UV3N3ZUZacVFtRlRNazEIVTjOVlZXSkhhRzIVVm1oRFZWwMfKRlZHV214U2JHdzFWa2Q

zjGclNuUmhSemxWVmpOT00xcFZXbUZrUjA1RlPFWINUbFpVvmtwV2JURXdZVEZrU0ZOclpHc

VXBZVkJZWYwQxTkdVbFZTYIVacVZtdGFNRIZOZUZOVWJVWVTjVbFjHVjFaRmIzZFdha1poVjBaT2N

tSkdTbWxTTW1oWIYxZDRiMkl3TUhoWGJHUIIzbfZhy2xWclVrZFhiR3QzV2tSUlZrMXjJRWxhUO

hCSFZqSkZlVIZZWkZwv1JWcHIWVEJhVDjOc2NFaGpSbEpUVmxoQ1dsWnjXbGRoTVZWNVZXNU9hb

Ep0VWxsWmjGWmhZMvpzY2xkdFjteFdiVko1VmpjMWExWXDnVZTYTFwVlRwktSRlpxUVhoalZsW

V3N3ZUZacVFtRlRNazEIVTjOVlZXSkhhRzIVVm1oRFZWwMfKRlZHV214U2JH

dzFWa2QwYzjGclNuUmhSemxWVmpOT00xcFZXbUZrUjA1RlPFWINUbFpVvmtw

V2JURXdZVEZrU0ZOclpHcFRSvXBZVkJZWYwQxTkdVbFZTYIVacVZtdGFNRIZO

ZUZOVWJVWVTjVbFjHVjFaRmIzZFdha1poVjBaT2NtSkdTbWxTTW1oWIYxZDRi

Mkl3TUhoWGJHUIIzbfZhy2xWclVrZFhiR3QzV2tSUlZrMXjJRWxhU0hCSFZq

SkZlVIZZWkZwv1JWcHIWVEJhVDjOc2NFaGpSbEpUVmxoQ1dsWnjXbGRoTVZW

TRENDI



Internet

Most



We can try to deal with big URLs...

- Hash them to smaller hash codes and store those. But:
 - Now there can be false positives – an innocent URL that happens to hash to the same value as a known malicious one would be rejected.
 - This might *still* be too much space, for **the heart of man is an unfathomable wellspring of evil** (i.e., there are a lot of sketchy URLs)
 - What if we're on a smartphone or something? What if space really is the limiting factor?

Bloom Filters

We need:

- Some number b of bits, all initialized to 0.
- Some set k of hash functions h_1, \dots, h_k that each hash to the range $[0, b-1]$.
 - For simplicity here, we assume that when given the same element x , the hash functions all choose *different* values in that range $[0, b-1]$. This is not trivial to achieve, but we'll assume it for now for convenience. It's not super unrealistic when b is very large relative to k (since then it kinda happens anyway)

Insertion

When we see a **new item** x (say, a URL), we:

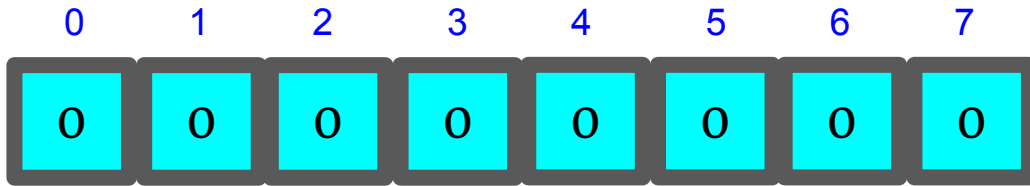
- Put it in every hash function
 - i.e., calculate $h_1(x), \dots, h_k(x)$
- Set each bit $h_1(x), \dots, h_k(x)$ to 1.
 - If any of these are already 1, that's fine. They stay 1.

Querying

When we want to know if we **have seen** an item x , we:

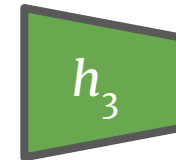
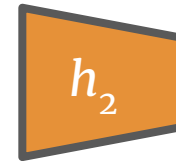
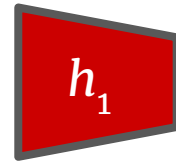
- Put it in every hash function
 - i.e., calculate $h_1(x), \dots, h_k(x)$
- Check each bit $h_1(x), \dots, h_k(x)$.
- If at least one of these is 0, we have **definitely not** seen this item before.
 - But what if all of them are 1?

Suppose $b = 8$, $k = 3$.

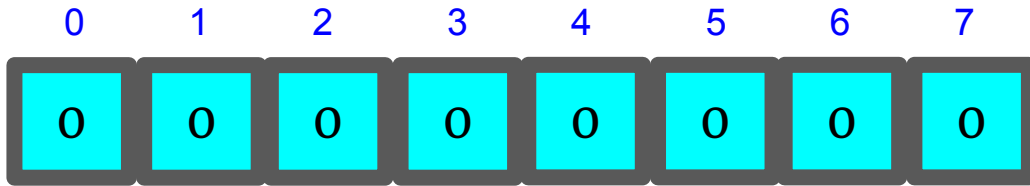


8 bits

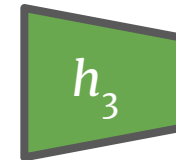
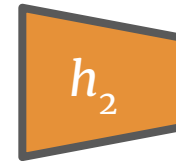
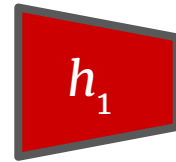
3 hash
functions



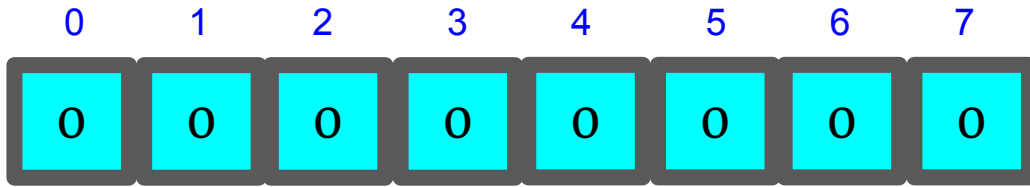
An Insertion



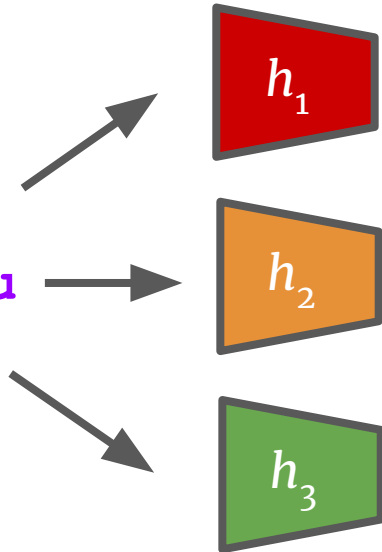
cs161.stanford.edu



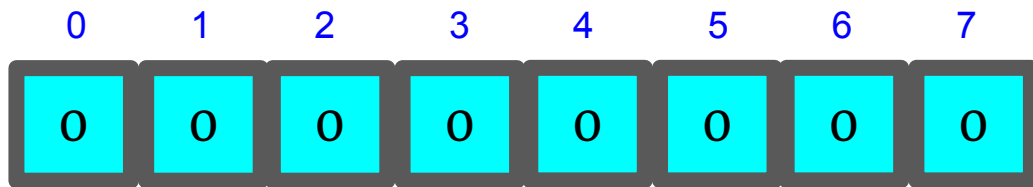
An Insertion



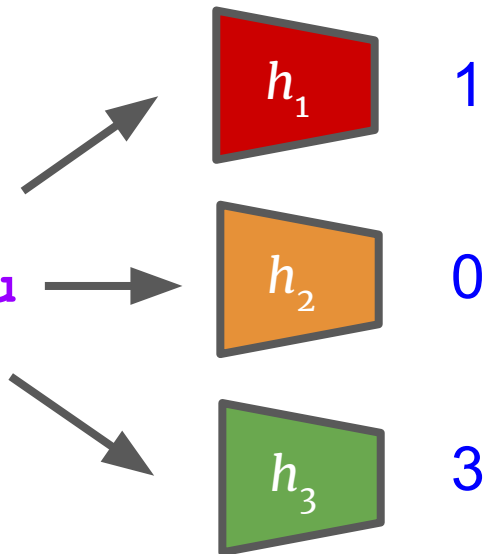
`cs161.stanford.edu`



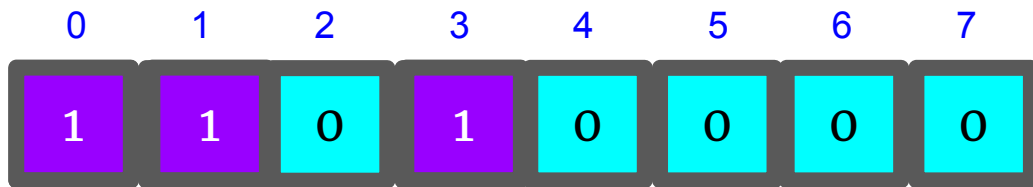
An Insertion



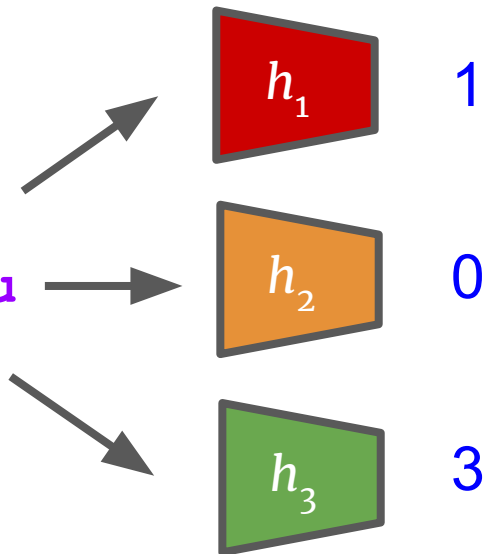
`cs161.stanford.edu`



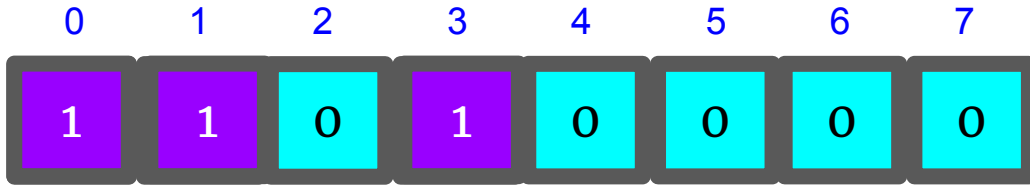
An Insertion



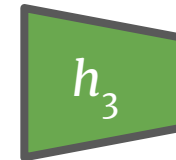
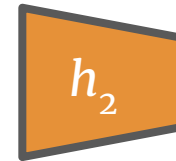
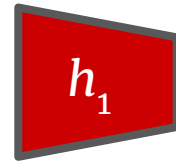
cs161.stanford.edu



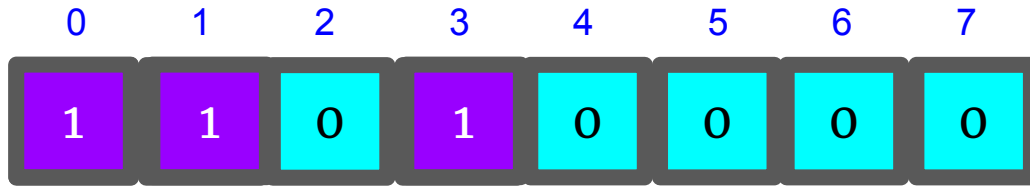
An Insertion



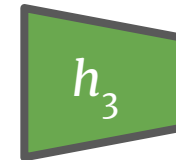
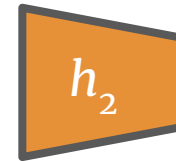
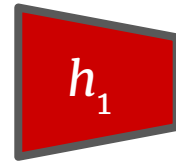
Note: At this point the structure has completely forgotten the specific string `cs161.stanford.edu`



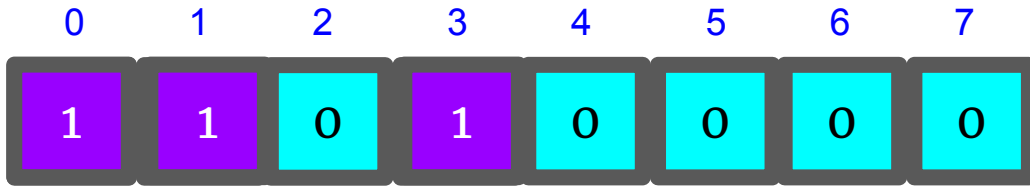
Another Insertion



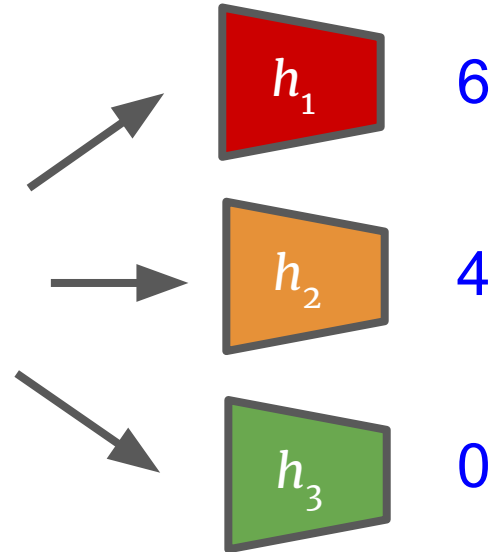
sketchytimeshares.com/sign-me-up.php



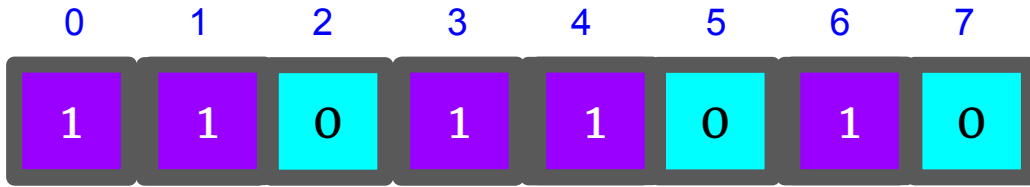
Another Insertion



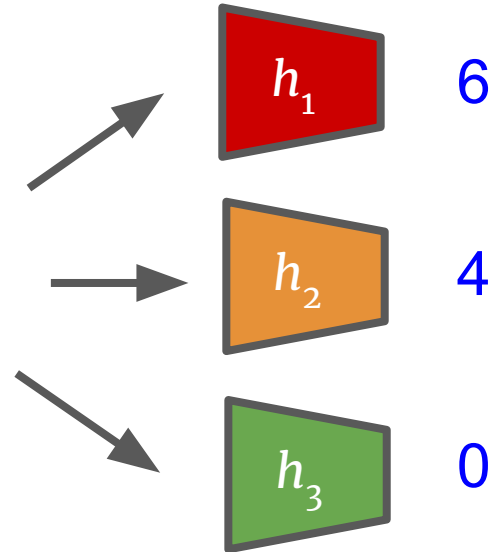
sketchytimeshares.com/sign-me-up.php



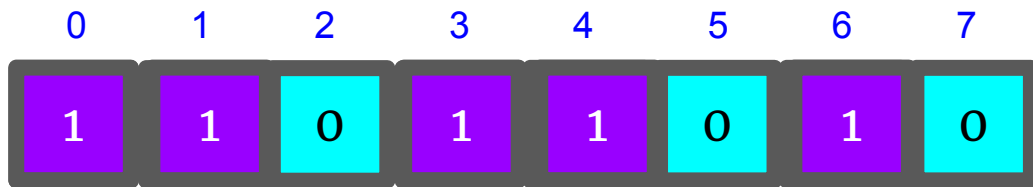
Another Insertion



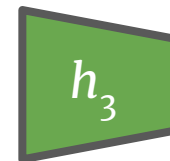
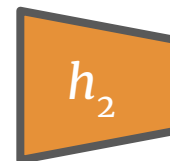
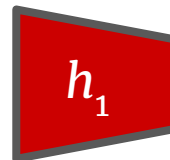
sketchytimeshares.com/sign-me-up.php



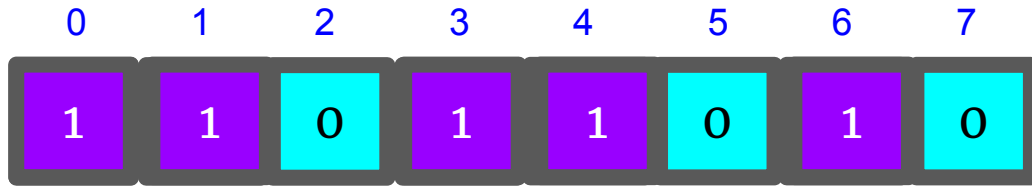
Querying Something We've Seen



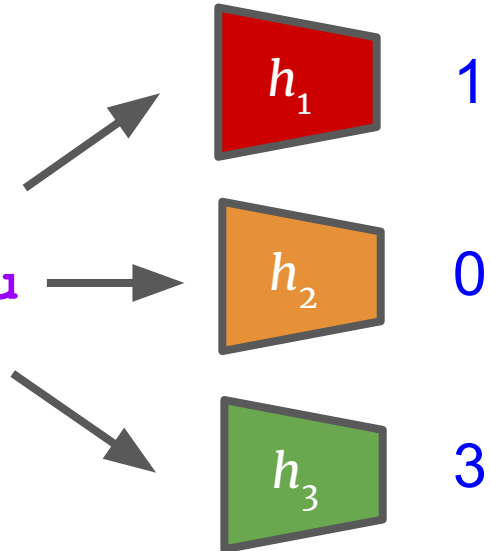
`cs161.stanford.edu`



Querying Something We've Seen

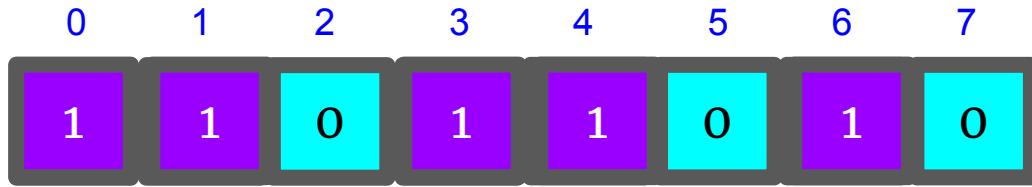


`cs161.stanford.edu`

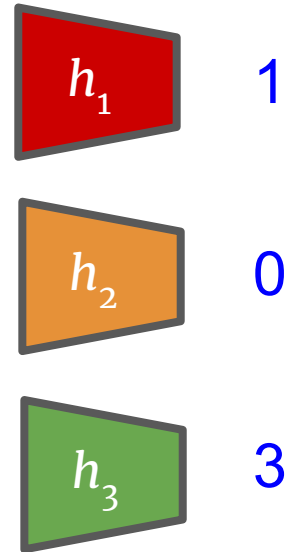


Seems inefficient to do this again. Couldn't we somehow remember that we'd calculated the hashes before? ...well then we'd be back to storing URLs

Querying Something We've Seen

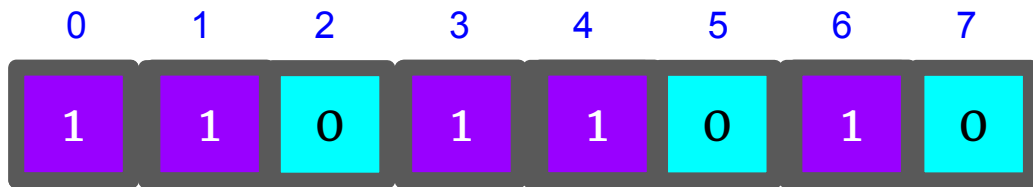


cs161.stanford.edu

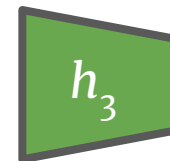
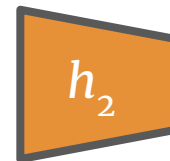
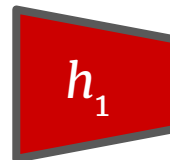


Sure, we've seen it before!

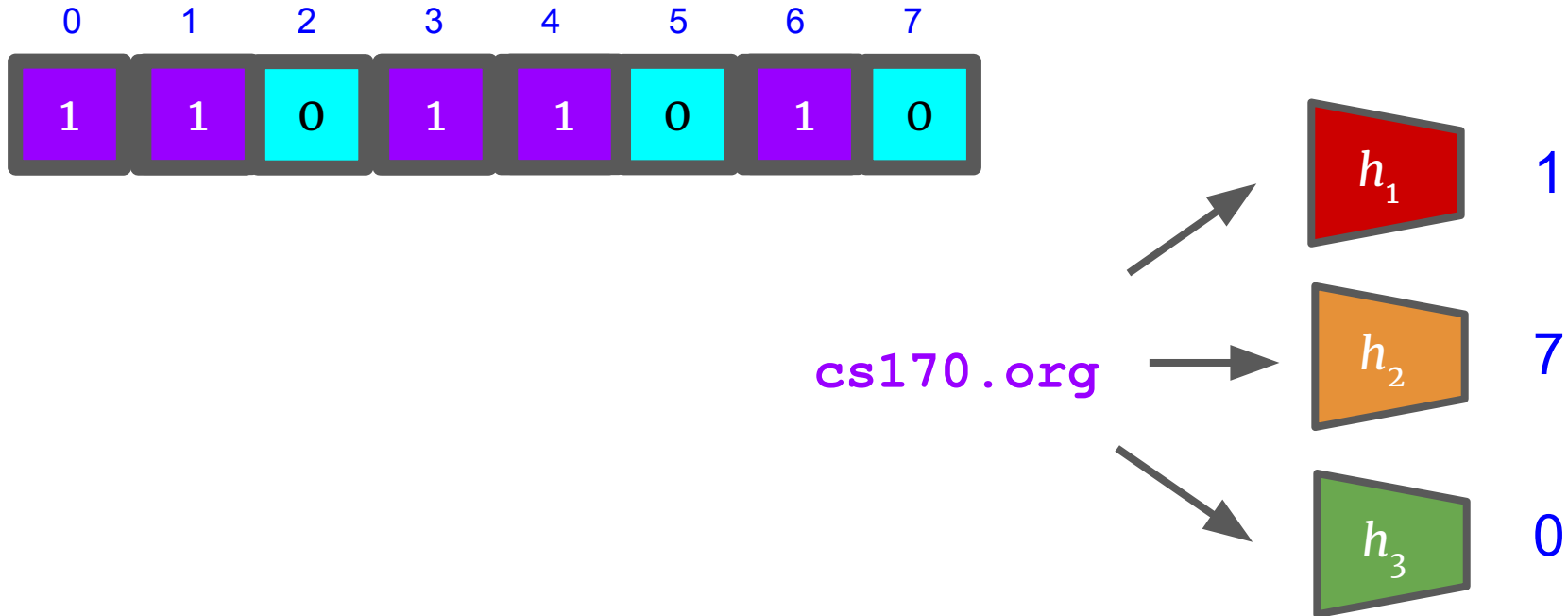
Querying Something We Haven't Seen



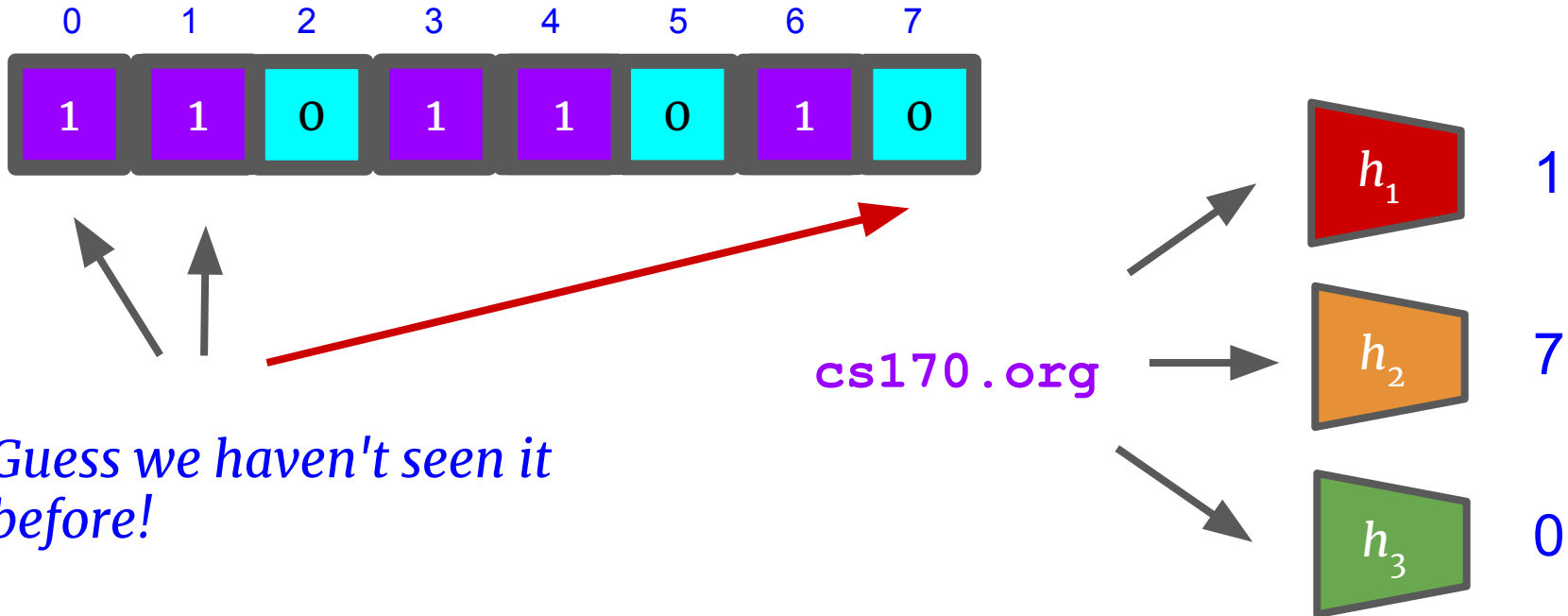
cs170.org



Querying Something We Haven't Seen



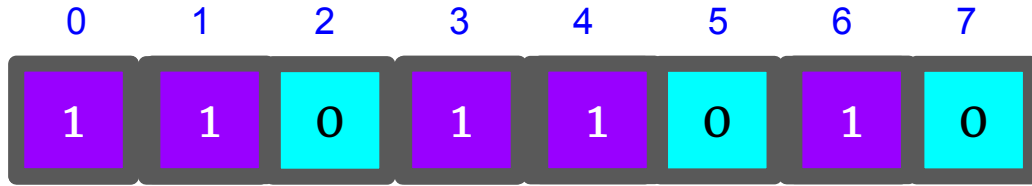
Querying Something We Haven't Seen



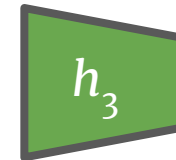
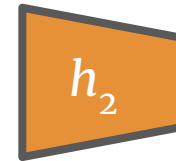
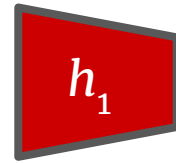
Homework 13 and a project!!!!

4/21					
Tu 4/26	Randomized Algorithms Prof. Wright's notes	DPV §1.3 notes §3			
Th 4/28	Quantum Algorithms Prof. Wright's notes (Part 1) Prof. Wright's notes (Part 2)		Section 13, solutions		HW 13, solutions Project

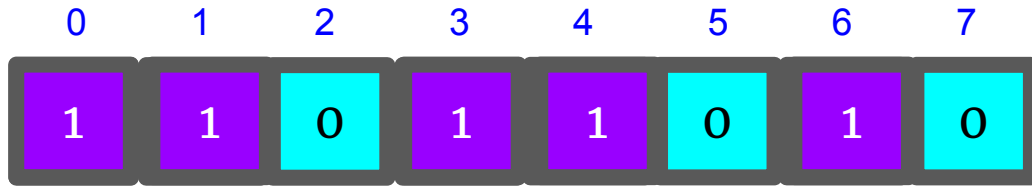
Querying Something Else We Haven't Seen



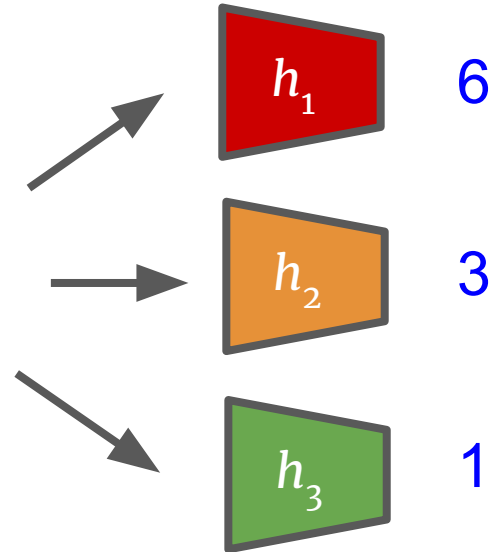
doctorswithoutborders.org



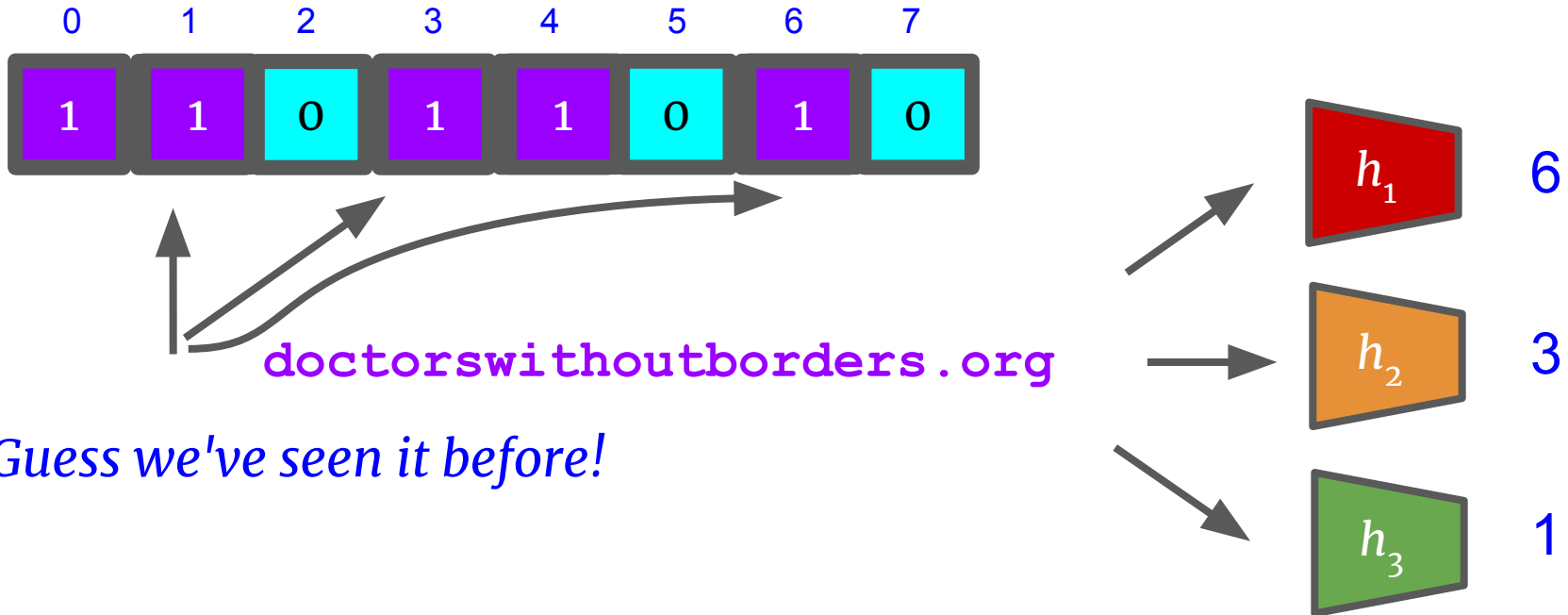
Querying Something Else We Haven't Seen



doctorswithoutborders.org



Querying Something Else We Haven't Seen



Oh no! A false positive!

Bloom filters can return false positives – saying we've seen something before when we haven't.

Can they also return false negatives, i.e., say we haven't seen something before when we actually have?



But no false negatives

Bloom filters can return false positives – saying we've seen something before when we haven't.

Can they also return false negatives, i.e., say we haven't seen something before when we actually have?

No. If we've seen something before, all of the bits it hashes to are definitely set.



How likely are false positives?

- Say we have inserted n items into a Bloom filter with b bits and k hash functions.
- Now we query for some item x we haven't seen. What is the probability of a false positive?
 - This means every one of the bits that x is hashed to (we'll call them "targeted bits") must already be a 1. Let's first look at just one of them.

Probability that a single targeted bit is 1

- As is often the case, it turns out to be easier to solve for the probability that the bit is 0, and then subtract that from 1. (Either the bit is 0, or it's 1.)
- This only happens if *none* of the previously inserted items set this bit to 1.
- What is the probability that any one of them did?

Let's focus on some arbitrary bit. What's the probability that it is 0 at the time of our new insertion?

- This only happens if *none* of the n previously inserted items set this bit to 1.
- What is the probability that any one of them did?
 - Each item is hashed to k different bits. So with probability k/b , an insertion sets this bit. *Note: other analyses that you may see might not assume this*
 - Therefore, with probability $1 - (k/b)$, the insertion does *not* set the bit.
 - And with probability $[1 - (k/b)]^n$, none of the insertions do. (Note: this assumes that the different inserted values hash independently)

Probability that *all* targeted bits are 1

- For any given targeted bit, it is 0 with probability $[1 - (k/b)]^n$. So it is 1 with probability $1 - [1 - (k/b)]^n$.
- If we apply the argument to each of the k targeted bits independently, they are **all** 1 with probability $(1 - [1 - (k/b)]^n)^k$.
 - Is this legitimate? These values are not really independent! If one bit is set, we conditionally know that others are less likely to be set.
 - We could sidestep this non-independence by finding the *expected* number of 1s, but that doesn't give us a probability. (It can let us *estimate* one, but that's more like CS265 material...)

This is hard! Let's let someone else do it!

Goel and Gupta,^[10] however, give a rigorous upper bound that makes no approximations and requires no assumptions. They show that the false positive probability for a finite Bloom filter with b bits ($b > 1$), n elements, and k hash functions is at most

$$\varepsilon \leq \left(1 - e^{-\frac{k(n+0.5)}{b-1}}\right)^k.$$

.

So, suppose $n = 1000$, $k = 5$. Then:

- If $b = 10000$, this probability is about 0.009.
- If $b = 1000$, this probability is about 0.97.
- If $b = 100$, this probability is basically 1.



Ashish (Goel) is a prof here!

This is hard! Let's let someone else do it!

Goel and Gupta,^[10] however, give a rigorous upper bound that makes no approximations and requires no assumptions. They show that the false positive probability for a finite Bloom filter with b bits ($b > 1$), n elements, and k hash functions is at most

$$\varepsilon \leq \left(1 - e^{-\frac{k(n+0.5)}{b-1}}\right)^k.$$

.

So, suppose $n = 1000$, $k = 5$. Then:

- If $b = 10000$, this probability is about 0.009.
- If $b = 1000$, this probability is about 0.97.
- If $b = 100$, this probability is basically 1.

*Isn't this really bad performance for $n = b$?
But remember that we're comparing entire URLs with
single bits! Normally we can have $b \gg n$*



*Ashish (Goel) is a
prof here!*

- You're not responsible for the math of the probability of a false positive, or for that formula.
- I want you to understand the overall ideas:
 - More bits = better performance, of course, but the whole idea is to save space, so we can't go wild here. (If this stops fitting in cache space, it's extra bad)
 - More hash functions = lower collision probability for different items, but the filter fills up faster
 - Can tune these parameters based on the anticipated number of items

Some final observations

- Insertion and querying both take constant time!
- Deletion? uh... doesn't really work.
 - You'll investigate this on HW3!
- If the table fills up too much, you're also out of luck. The best option may be to keep the old one but also start a new larger one
- One good modern application: forbidden passwords (like "qwerty123456" and "password")