Rewatch lectures:

3/11, 3/16, 3/18, and 3/23

Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Hashing, bloom filters

Outline

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

The problem

A data structure maintaining a dynamic subset S of a huge universe U.

- Typically, $|S| \ll |U|$

The data structure should support

- efficient **insertion**
- efficient **deletion**
- efficient **search**

We will call such a data structure a **dictionary**.

Dictionary data structure

A dictionary maintains a subset S of a universe U so that inserting, deleting and searching is efficient.

Operations supported by a dictionary

1. `Create()`: initialize a dictionary with $S = \emptyset$
2. `Insert`(x): add x to S, if $x \notin S$ ← so to add an element you must search for it first.
   - additional information about x might be stored in the dictionary as part of a record for x
3. `Delete`(x): delete x from S, if $x \in S$ ← Search required here too.
4. `Lookup`(x): determine if $x \in S$

Notice: Both Insert(x) and Delete(x) require Lookup(x)

A concrete example

We want to maintain a dynamic list of 250 IP addresses

- e.g., these correspond to addresses of currently active customers of a Web service
- each IP address consists of 32 bits, e.g. 128.32.168.80

To motivate this and illustrate the idea, assume there are no constraints on space.

We are trying to maintain elements in a set S from a huge universe U s.t.

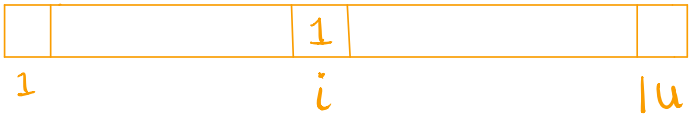
$$|S| << |U|$$

The question is: How can we do this efficiently?

Suppose that space is not an issue. How would you design a data structure to maintain a subset S of a huge universe U so that Insert(x), Delete(x), and Lookup(x) are efficient?

Idea 1:

Consider a Boolean array of size $|U|$, one entry for every element in the universe. If element $i \in U$ belongs to S place a 1 in the i^{th} entry of the array and 0 otherwise. (Initially place a 0 since initializing array w/ 0's).



if $i \in S$

if element needs to be deleted from the set S, reset the entry to 0.

U

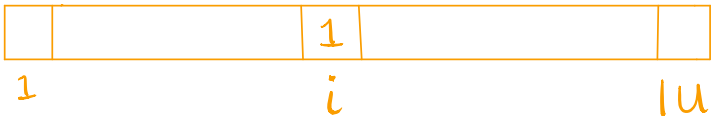
$|S| << |U|$

Space: $O(|U|)$

Time: Lookup(x) $\rightarrow O(1)$

Insert(x) $\rightarrow O(1)$

Delete(x) $\rightarrow O(1)$



This is very efficient in terms of time, but bad in terms of space. You will not be able to maintain an array anywhere close to the size of U.

- So we should NOT do this.

other ideas so that Lookup, Insert, and Delete are efficient, and the space required to store S is not enormous.

Idea 2: Use a linked-list to store the elements:

The size of the linked-list is the size of S.

Space: $O(|S|)$ using exactly what is required.

Running-Time? Lookup(x) $\rightarrow O(|S|)$

(in the worst-case, you will have to traverse the entire linked list to find the element x.)

Insert(x) → O(|S|)

(Insert requires a Lookup(x) call plus time to add the node/element if needed which takes constant time.)

Delete(x) → O(|S|)

(Same argument as Insert).

Space: O(|S|)

Time: O(|S|)

} Space is great, but now the time is proportional to the size of the set you are trying to maintain which is not desirable.

Space good!

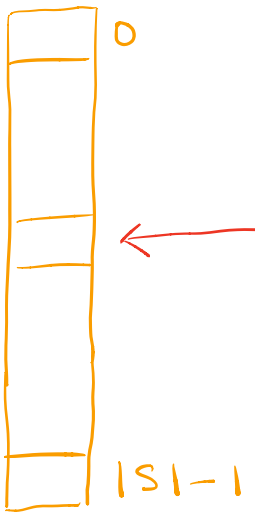
Time Bad!

Need something that is the best of both worlds.

Idea 3: For sure we want the space to be proportional to the size of S. Don't want to spend more space than the size of S.
For sure going to maintain an array of size $|S|$.
Can't have a linked list because this will worsen the running time.

To make sure our operations are fast we need to be able to do the following.

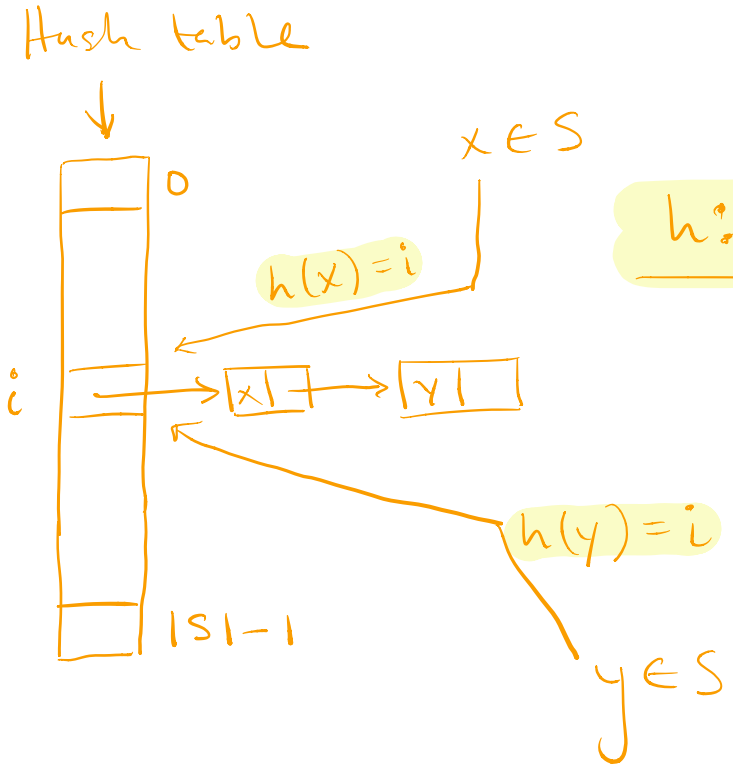
Given an $x \in U$, we need to be able to quickly map it to some position in the array.



Obviously, x will not be matched to location x (index x) since we do not have that many positions in this array.

One way to do this is to use a hash function.
A hash function h takes elements from the universe and maps them to some position in the array.

$$h : U \longrightarrow \{0, \ldots, |S|-1\}$$



$h: u \longrightarrow \{0, \ldots, |S|-1\}$

- Use linked list to manage multiple entries in the same position.

Hashing with chaining

Time: Lookup(x) → Compute $h(x)$ → $O(1)$

→ Scan the linked list at $h(x)$.

In the worst-case it is $O(|S|)$.

More precisely, it takes

O(size of the linked list)

Therefore, we want a hash table that spreads out the

the elements well. You want to maintain the time for Lookup(x) to be small., and so the size of the linked list at every position needs to be small. To achieve this, you need a hash function that spreads out elements well.

GOAL: Design a hash function that spreads out elements well.

The challenge: U is enormous, that is, $|U| \gg |S|$

1. Maintain **array** S of size $|U|$ such that $S[i] = 1$ if and only if $i \in S$
   - `Insert`, `Delete`, `Lookup` require $O(1)$ time

   Can't store an array of size anywhere close to $|U|$!
   - S should have $|U| = 2^{32} \approx 4$ billion entries
   - S would be mostly empty (huge waste of space)

2. Store S in a **linked list**
   - Space: proportional to $|S| = 250$
   - Time for `Lookup`: proportional to $|S|$; too slow

Can we support fast `Insert`, `Delete`, `Lookup` *(as in array implementation) but only use space proportional to* $|S|$ *(linked list implementation)?*

Work with array of size $|S|$ rather than one of size $|U|$

Idea: assign a short *nickname* to each element in U

- Each of the 2^{32} IP addresses is assigned a number between 1 and $|S| = 250$
  - range will be slightly adjusted
- Total amount of storage: approximately $|S|$, independent of $|U|$
- If not too many IP addresses per nickname, then `Lookup` is efficient (*details coming up*)

How can we assign a short name?

By hashing: use a hash function $h : U \to \{0, \ldots, n-1\}$

- Typically, $n \ll |U|$ and is close to $|S|$

For example,

- $h : \{0, \ldots, 2^{32} - 1\} \to \{0, \ldots, 249\}$
- IP address x gets name $h(x)$
- Hash table H of size 250: store address x at entry $h(x)$

So `Insert`(x) takes constant time. *What if we try to insert* $y \neq x$, *with* $h(x) = h(y)$*?*

Collisions

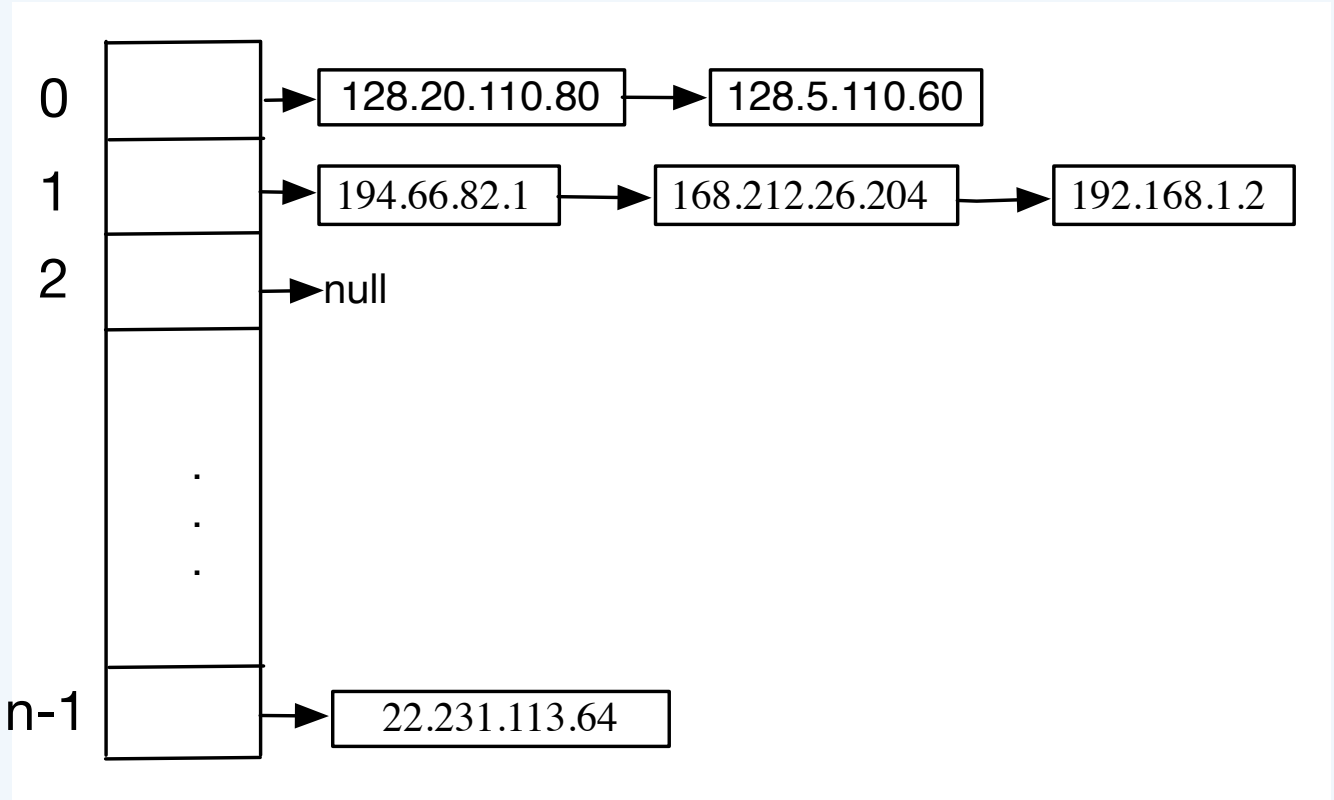
Collision: elements $x \neq y$ such that $h(x) = h(y)$

Easiest way to deal with collisions: chain hashing

- Entry i in the hash table is a **linked list** of elements x such that $h(x) = i$
- Alternatively, can think of every entry in the hash table as a **bin** containing the elements that hash to the same location

Chain hashing

Maintain a linked list at $H[i]$ for all x such that $h(x) = i$.



Chain hashing: running time for Lookup(x)

Time for `Lookup`(x):

1. time to compute $h(x)$; typically, constant
2. time to scan the linked list at position $h(x)$ in hash table
   - proportional to the *length* of the linked list at $h(x)$, which is proportional to the # elements that collide with x

Goal: find a hash function that "spreads out" the elements well

Simple hash functions might not work

Deterministic hash functions

Consider the following two simple hash functions that hash an IP address x from $\{0, \ldots, 2^{32} - 1\}$ to $\{0, \ldots, 255\}$:

- assign the last 8 bits of x as its name
- assign the first 8 bits of x as its name

$\approx |S|$

hash value

hash value

Remark 1.

Nothing is inherently wrong with these hash functions: the problem is that our 250 IP addresses might not be drawn uniformly at random from among all 2^{32} possibilities.

It might be the case that the first 8 bits of the IP address trying to maintain are the same for too many entries, which causes too many collisions.

For a deterministic (fixed) hash function there will always be an input that causes way too many collisions, assuming that the universe is large enough.

See following slide for a better description of this. ↓

No single hash function can work well on *all* data sets

- **Fix** the hash function h.
- h distributes $|U|$ elements into n names.

$\Rightarrow$ exists data set of at least $\frac{|U|}{n}$ elements that all map to the same name

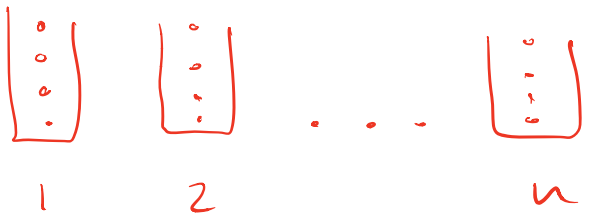
$\Rightarrow$ if our customers come from this data set, lots of collisions

Fact: for any **fixed** (**deterministic**) $h : U \to \{0, 1, \ldots, n-1\}$ where $|U| \geq n^2$, there exists some set S of n elements that all map to the same position.

Suppose we have:

- n^2 balls (elements in U)
- n bins (range of hash function; $|S| = n$)

Lower bound on the maximum load of any bin? Answer: n



under the most balanced allocation, each bin will have exactly n balls

- Under the most balanced allocation, the max load of every bin is n.
- If the allocation is not as balanced: at least one bin will have fewer than n balls, but those balls must be in another bin, so other bins will have $> n$ balls. Therefore, the max load of any bin is at least n.

Using Randomness in the Hash function

$h: U \longrightarrow \{0, 1, \ldots, n-1\}$ (where $|S| = n$)

- $\forall x \in U$ independently

  set $h(x) = j$ for $0 \leq j \leq n-1$

  with probability $\frac{1}{n}$.

- Is this a good hash function?

  Need to know prob. of a collision (i.e. $h(x) = h(y)$) to determine this.

- $\Pr[h(x) = h(y)] =$

$$= \sum_{i=0}^{n-1} \Pr[\{h(x) = i\} \wedge \{h(y) = i\}]$$

$$= \sum_{i=0}^{n-1} \Pr[h(x) = i] \cdot \Pr[h(y) = i]$$

$$= \sum_{i=0}^{n-1} \frac{1}{n^2} = \frac{1}{n}$$

Suppose $x \in S$

Let X be the number of other items from S that collide with x.
(Gives us length of linked list @ $h(x)$).

We want $E[X]$.

Let $X_i = \begin{cases} 1, & \text{if item } i \in S \text{ collides with } x \ (h(i) = h(x)) \\ 0, & \text{o.w.} \end{cases}$

$X = \sum_{i=1}^{n-1} X_i$ ($n-1$ other items — not including x)

$$E[X] = E\left[\sum_{i=1}^{n-1} X_i\right] = \sum_{i=1}^{n-1} E[X_i]$$

$$= \sum_{i=1}^{n-1} Pr[X_i = 1] = \sum_{i=1}^{n-1} \frac{1}{n} = \frac{n-1}{n} < 1$$

Therefore, for any entry in our hash table, it is expected that the size of the linked list at a given entry is less than 2

(the item from our set plus at most one more item).

The expected size of every linked list is < 2.

In expectation:

Lookup(x) ⟶ ... O(1)

<scan list> O(1) in expectation

Issue: Don't know where we stored x.

What we proposed is not a function.

Not using this hash function either.

Why the fuck do you keep showing us the wrong way??

Show us the right thing first!!

Randomization can help

- **Extreme example:** for every $0 \leq j \leq n-1$, assign name j to element x with probability $\frac{1}{n}$.
  - Fix $x, y \in U$. Then $\Pr[h(x) = h(y)] = \frac{1}{n}$.
  - This doesn't quite work. (Think `Lookup`(x): *where is* x*?*)

- However, intuitively, hash functions that spread things around in a *random* way can effectively reduce collisions.

$\Rightarrow$ Trade-off in hash function design: h must be "random" to scatter things around for all inputs but still be a function

Goal: design h that allows for efficient dictionary operations with high probability

A careful use of randomization

- Randomize over the **choice** of the hash function from a suitable class of functions into $[0, n-1]$ (*details coming up*)
- h must have a compact representation

Universal hash function

Idea: choose h at random from a carefully selected class of functions H with the following properties:

1. h behaves almost like a completely random hash function.
   - For $x, y \in U$. The probability that a randomly chosen $h \in H$ satisfies $h(x) = h(y)$ is at most $1/n$.
2. Can select a random h efficiently.
3. Given h, can compute $h(x)$ efficiently.

Such hash functions are called universal; their design relies on number theoretic facts.

Hash function is universal if collision occurs w/ very small probability.

Example of a universal hash function:

$U = \{a, b, c, d, e, f\}$

$n = 2$

$H = \{h_1, h_2\}$ $\qquad h_i : U \longrightarrow \{0, \ldots, n-1\}$

$h_1 : U \longrightarrow \{0, 1\}$

$h_2 : U \longrightarrow \{0, 1\}$

	a	b	c	d	e	f
h_1	0	0	0	1	1	1
h_2	0	1	0	1	0	1

Fix a, b

$$\Pr_{h \in H}[h(a) = h(b)] = \Pr[h_1(a) = h_1(b) \mid h_1] \cdot \Pr[h_1] + \Pr[h_2(a) = h_2(b) \mid h_2] \cdot \Pr[h_2]$$

$$= (1) \cdot \left(\frac{1}{2}\right) + (0) \cdot \left(\frac{1}{2}\right)$$

$$= \frac{1}{2}$$

Thus, for a and b, the hash functions satisfy the universal property — since $\Pr[h(a) = h(b)] \leq \frac{1}{n} = \frac{1}{2}$

We must show that this is true for all pairs before concluding that this is a universal hash function.

Fix c, d.

$$\Pr_{h \in H}[h(c) = h(d)] = \Pr[h_1(c) = h_1(d) \mid h_1] \cdot \Pr[h_1] + \Pr[h_2(c) = h_2(d) \mid h_2] \cdot \Pr[h_2]$$

$$= (0) \cdot \left(\tfrac{1}{2}\right) + (0) \cdot \left(\tfrac{1}{2}\right)$$

$$= 0$$

$$\Rightarrow \Pr[h(c) = h(d)] = 0 \leq \frac{1}{2} \quad \checkmark$$

hash functions satisfy universal property for c and d.

//

Fix a, c.

$$\Pr_{h \in H}[h(a) = h(c)] = \Pr[h_1(a) = h_1(c) \mid h_1] \cdot \Pr[h_1] + \Pr[h_2(a) = h_2(c) \mid h_2] \cdot \Pr[h_2]$$

$$= (1) \cdot \left(\tfrac{1}{2}\right) + (1) \cdot \left(\tfrac{1}{2}\right)$$

$$= 1$$

$$\Rightarrow \Pr[h(a) = h(c)] > \frac{1}{2}$$

Therefore, this family of hash functions is <u>NOT</u> a universal hash function.

Example of universal hash function

- Pick a prime p close to $|S| = 250$; set $n = p$
  - E.g., pick $p = 257$; set the size n of the hash table to 257
- Look at IP address x as (x_1, x_2, x_3, x_4), where x_1, x_2, x_3, x_4 are integers $\mod n$.
- Define $h : U \to \{0, 1, \ldots, n-1\}$ as follows:
  - Choose a_1, a_2, a_3, a_4 randomly from $\{0, 1, \ldots, n-1\}$
    - E.g., $a_1 = 80, a_2 = 35, a_3 = 168, a_4 = 220$
  - Map IP address x to $h(x) = \left(\sum_{i=1}^{4} a_i x_i\right) \mod n$
    - E.g., $x = 128.32.168.80$,
      $h(x) = (80 \cdot 128 + 35 \cdot 32 + 168 \cdot 168 + 220 \cdot 80) \mod 257$

h is a universal hash function

Claim 1.

Consider any pair $x = (x_1, x_2, x_3, x_4)$, $y = (y_1, y_2, y_3, y_4)$. *If* $a_1, \ldots, a_4$ *are chosen uniformly at random from* $\{0, \ldots, n-1\}$, *then*

$$\Pr[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)] = \frac{1}{n}$$

The proof relies on elementary number theory.

Corollary 1.

Fix $x \in U$. *The expected #elements colliding with* x *is less than 1. Hence the expected lookup time is constant.*

Ideal hash functions

Completely random hash functions do not exist, but can use universal hash functions in their place.

From now on, assume a *completely random hash function* exists.

△ *Does not exist! But can provide a good rough idea of how hashing schemes perform in practice.*

- Let $h : U \to \{0, 1, \ldots, n-1\}$ be a completely random (ideal) hash function. For all $x \in U$, $0 \leq j \leq n-1$

$$Pr[h(x) = j] = \frac{1}{n}$$

Remark 2.

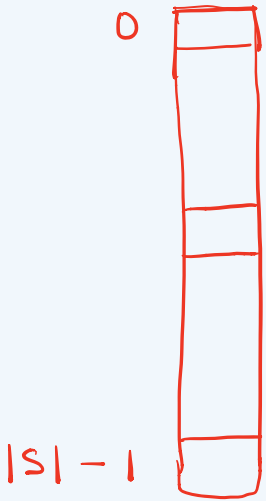
$h(x)$ is **fixed** *for every x: it just takes* **one** *of the n possible values with equal probability.*

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?



balls $\longleftrightarrow$ $x \in S$

bins $\longleftrightarrow$ entries in the hash table.

$\underline{O}(\sqrt{n}) = \Omega(\sqrt{n})$

$|S| = n$

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

This is just an occupancy problem!

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Occupancy problems, revisited: find the distribution of balls into bins when m balls are thrown independently and uniformly at random into n bins.

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Occupancy problems, revisited: find the distribution of balls into bins when m balls are thrown independently and uniformly at random into n bins.

Hashing as an occupancy problem:

- balls correspond to elements from U
- bins are slots in the hash table
- each ball falls into one of the n bins independently and with probability $1/n$

Hashing modeled as a balls and bins problem

Q1: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Hashing as an occupancy problem:

- balls correspond to elements from U
- bins are slots in the hash table
- each ball falls into one of the n bins independently and with probability $1/n$

Q1 (rephrased): How many balls can we throw before it is more likely than not that some bin contains at least two balls?

Answer: $\Omega(\sqrt{n})$ (see the birthday paradox)

Towards analyzing time/space efficiency of hash table

Lookup(x) involves 2 things : (i) Computing $h(x)$ $\longrightarrow O(1)$

(ii) Scanning the linked list at the position $h(x)$

- *What is the expected time for* Lookup(x)*?*

  Proportional to the expected # of balls in a bin : $O(1)$

- *What is the expected wasted space in the hash table?*

  $> \frac{n}{3}$

- *What is the worst-case time for* Lookup(x)*?*

  $O(n)$, but this is VERY unlikely.

  Better question to ask is: ↓

- *What is the worst-case time for* Lookup(x)*, with high probability?* Better Question! Answered Below.

Towards analyzing time/space efficiency of hash table

- *What is the expected time for* `Lookup`(x)*?*
  Corresponds to expected load of a bin.
- *What is the expected wasted space in the hash table?*
  Corresponds to expected number of empty bins.
- *What is the worst-case time for* `Lookup`(x)*?*
  Corresponds to load of the fullest bin.

What is the load of the fullest bin, with high probability, when we throw n balls into n bins independently and uniformly at random?

"With high probability" means the probability increases as the input size n grows very large. For instance, an event that happens with at least any of the following probabilities:

- $1 - \frac{1}{n}$
- $1 - \frac{1}{e^n}$
- $1 - \frac{1}{n^2}$
- $1 - \frac{1}{\sqrt{n}}$

Which of this acceptable for the particular application depends on the application. For most, however, $\geq 1 - \frac{1}{n}$ is good enough.

- If you can show the event happens w/ prob. 1 minus the inverse of the input size, then that is a "high probability" event.

What is the load of the fullest bin, with high probability, when we throw n balls into n bins independently and uniformly at random?

Want to show that

every bin has $\leq k^*$ balls with probability $\geq 1 - \frac{1}{n}$

What is the complement of this event?

$\exists$ bin that has $\geq k^*$ balls with probability $\leq 1 - \frac{1}{n}$.

Find k^* s.t.

$$\Pr[\exists \text{ bin w/ } \geq k^* \text{ balls}] \leq \frac{1}{n}$$

(These events are NOT mutually exclusive.)

$$\Rightarrow \Pr[\{\text{bin 1 has} \geq k^* \text{ balls}\} \text{ OR } \{\text{bin 2 has} \geq k^* \text{ balls}\} \ldots \text{OR } \{\text{bin } n \text{ has} \geq k^* \text{ balls}\}]$$

$$\leq \sum_{i=1}^{n} \Pr[\text{bin } i \text{ has} \geq k^* \text{ balls}]$$

$$= n \cdot \Pr[\text{fixed bin has} \geq k^* \text{ balls}] \leq \frac{1}{n}$$

(since bins are indistinguishable).

$$n \cdot \Pr[\text{fixed bin has} \geq k^* \text{ balls}] \leq \frac{1}{n}$$

$$\Rightarrow \boxed{\Pr[\text{fixed bin has} \geq k^* \text{ balls}] \leq \frac{1}{n^2}}$$

Now, we need to compute an expression for this probability, which will be in terms of k^*. Then upper bound this expression by $1/n^2$ and solve for the minimum k^* that satisfies this inequality.

To motivate this, consider the following:

$$\Pr[\text{bin has} \geq k \text{ balls}] = \sum_{\ell=k}^{n} \binom{n}{\ell} \cdot \left(\frac{1}{n}\right)^{\ell} \left(1 - \frac{1}{n}\right)^{n-\ell}$$

FINISH LATER.

Rewatch lectures:

3/11, 3/16, 3/18, and 3/23

Towards analyzing time/space efficiency of hash table

For $n = m$

- *What is the expected time for* `Lookup`(x)*?*
  $O(1)$.
- *What is the expected wasted space in the hash table?*
  At least a third of the slots are empty.
- *What is the worst-case time for* `Lookup`(x)*, with high probability?*
  $\Theta(\ln n / \ln \ln n)$, with high probability.

Max load in any bin, with high probability (case $m = n$)

Proposition 1.

When throwing n balls into n bins uniformly and independently at random, the maximum load in any bin is $\Theta(\ln n / \ln \ln n)$ with probability close to 1 as n grows large.

Two-sentence sketch of the proof.

1. Upper bound the probability that **any** bin contains more than k balls by a union bound: $\sum_{j=1}^{n} \sum_{\ell=k}^{n} \binom{n}{\ell} \left(\frac{1}{n}\right)^{\ell} \left(1 - \frac{1}{n}\right)^{n-\ell}$.
2. Compute the smallest possible k^* such that the probability above is less than $1/n$ (which becomes negligible as n grows large).

□

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

A password checker

- We want to maintain a dictionary for a set S of 2^{16} **bad** passwords so that, when a user tries to set up a password, we can check as quickly as possible if it belongs to S and reject it.

- We assume that each password consists of 8 ASCII characters
  - hence each password requires 8 bytes (64 bits) to represent

A dictionary data structure that uses less space

Let S be the set of **bad** passwords.

Input: a 64-bit password x, and a query of the form *"Is x a* **bad** *password?"*

Output: a dictionary data structure for S that answers queries as above and

- is **small**: uses **less space** than explicitly storing all bad passwords
- allows for erroneous **yes** answers occasionally
  - that is, we occasionally answer "$x \in S$" even though $x \notin S$

Approximate set membership

The password checker belongs to a broad class of problems, called *approximate set membership* problems.

Input: a large set $S = \{s_1, \ldots, s_m\}$, and queries of the form *"Is $x \in S$?"*

We want a dictionary for S that is **small** (smaller than the explicit representation provided by a hash table).
To achieve this, we allow for some probability of error

- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Output: small probability of false positives, no false negatives

Fingerprints: hashing for saving space

- Use a hash function $h : \{0, \ldots, 2^{64} - 1\} \to \{0, \ldots, 2^{32} - 1\}$ to map each password into a 32 bit string.
- This string will serve as a short *fingerprint* of the password.
- Keep the *fingerprints* in a sorted list.
- To check if a proposed password is **bad**:
  1. calculate its *fingerprint*
  2. binary search for the *fingerprint* in the list of fingerprints; if found, declare the password **bad** and ask the user to enter a new one.

Setting the length b of the fingerprint

Why did we map passwords to 32-bit fingerprints?

Motivation: make fingerprints long enough so that the false positive probability is acceptable

Let b be the number of bits used by our hash function to map the m bad passwords into fingerprints, thus

$$h : \{0, 1, \ldots, 2^{64} - 1\} \to \{0, \ldots, 2^b - 1\}$$

We will choose b so that the probability of a false positive is acceptable, e.g., at most $1/m$.

Determining the false positive probability

There are 2^b possible strings of length b.
Let x be a **good** password.
Fix a $y \in S$ (recall that all m passwords in S are **bad**).

- $\Pr[x \text{ has the same fingerprint as } y] = 1/2^b$
- $\Pr[x \text{ does } \textit{not} \text{ have the same fingerprint as } y] = 1 - 1/2^b$
- let $p = 1 - 1/2^b$
- $\Pr[x \text{ does } \textit{not} \text{ have the same fingerprint as } \textit{any } w \in S] = p^m$
- $\Pr[x \text{ has the same fingerprint as some } w \in S] = 1 - p^m$

Hence the false positive probability is

$$1 - p^m = 1 - (1 - 1/2^b)^m \approx 1 - e^{-m/2^b}$$

Constant false positive probability and bound for b

To make the probability of a false positive less than, say, a constant c, we require

$$1 - e^{-m/2^b} \leq c \Rightarrow b \geq \log_2 \frac{m}{\ln(1/(1-c))}.$$

So $b = \Omega(\log_2 \frac{m}{\ln(1/(1-c))})$ bits.

Improved false positive probability and bound for b

Now suppose we use $b = 2 \log_2 m$.

Plugging back into the original formula for the probability of false positive, which is $1 - (1 - 1/2^b)^m$, we get

$$1 - \left(1 - \frac{1}{m^2}\right)^m \leq 1 - \left(1 - \frac{1}{m}\right) = \frac{1}{m}$$

Thus if our dictionary has $|S| = m = 2^{16}$ bad passwords, using a hash function that maps each of the m passwords to 32 bits yields a false positive probability of about $1/2^{16}$.

Today

1. Hashing
2. Analyzing hash tables using balls and bins
3. Saving space: hashing-based fingerprints
4. Bloom filters

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

To achieve the above, allow for some probability of error

- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Fast approximate set membership

Input: a *large* set S, and queries of the form *"Is $x \in S$?"*

We want a **data structure** that answers the queries

- **fast** (faster than searching in S)
- is **small** (smaller than the explicit representation provided by hash table)

To achieve the above, allow for some probability of error

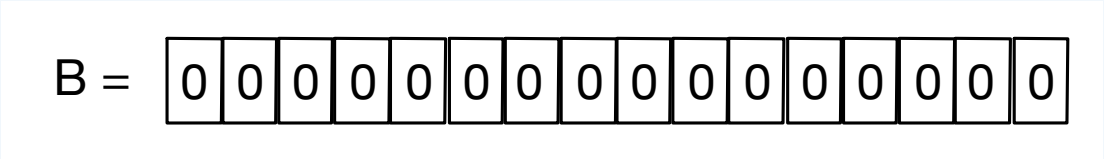
- False positives: answer **yes** when $x \notin S$
- False negatives: answer **no** when $x \in S$

Output: small probability of false positives, no false negatives

Bloom filter

A Bloom filter consists of:

1. an array B of n **bits**, initially all set to 0.



2. k independent random hash functions $h_1, \ldots, h_k$ with range $\{0, 1, \ldots, n-1\}$.

A basic Bloom filter supports

- `Insert`(x)
- `Lookup`(x)

Representing a set $S = \{x_1, \ldots, x_m\}$ using a Bloom filter

`SetupBloomFilter`$(S, h_1, \ldots, h_k)$

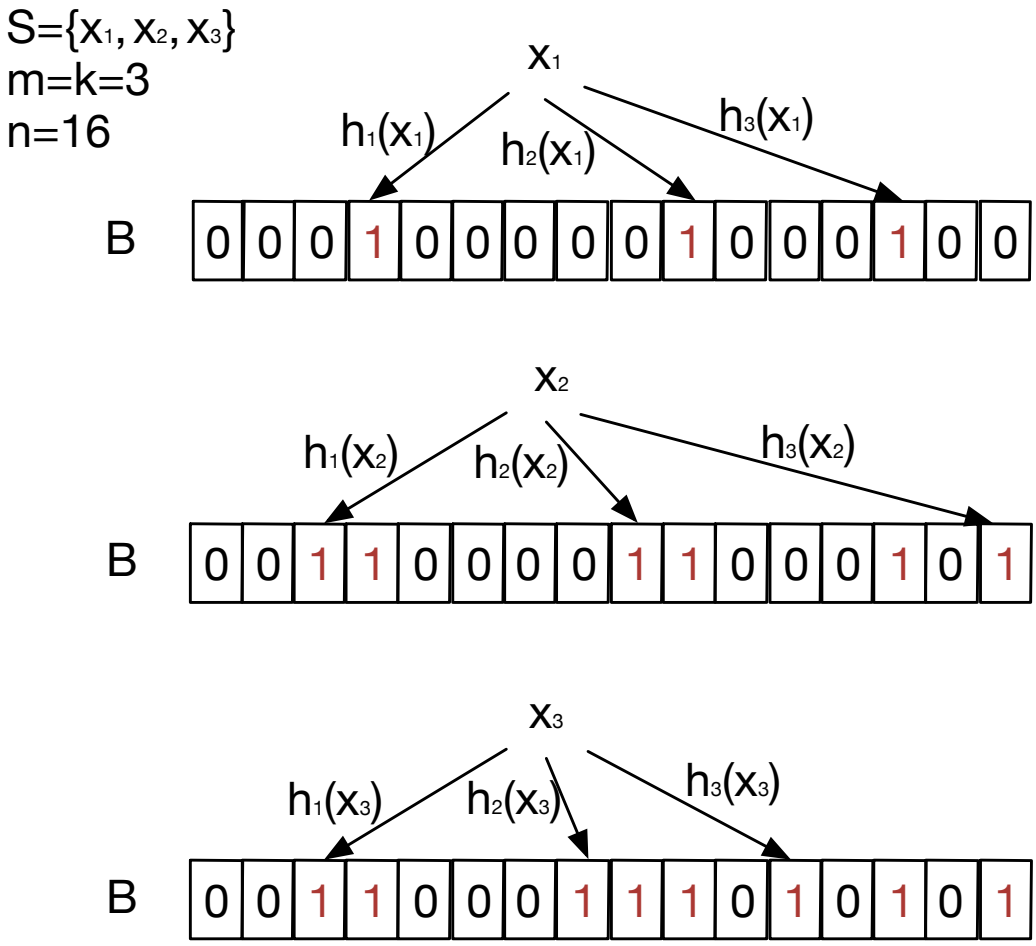
Initialize array B of size n to all zeros
for $i = 1$ to m **do**
    Insert(x_i)
end for

`Insert`(x)

for $i = 1$ to k **do**
    compute $h_i(x)$
    set $B[h_i(x)] = 1$
end for

Remark: an entry of B may be set multiple times; only the first change has an effect.

Setting up the Bloom filter



Bloom filter: Lookup

To check membership of an element x in S do:

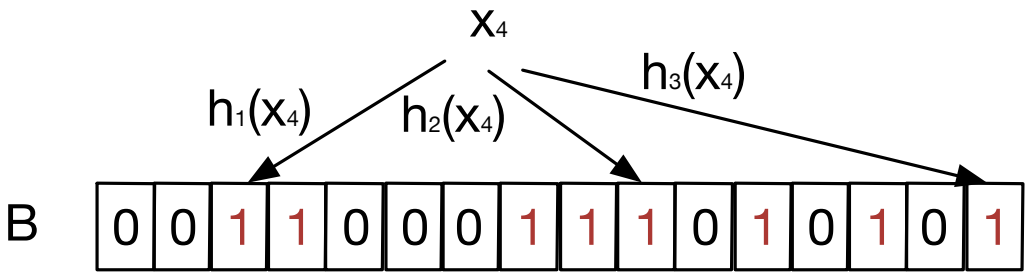
```
Lookup(x)
    for i = 1 to k do
        compute h_i(x)
        if B[h_i(x)] == 0 then
            return no
        end if
    end for
    return yes
```

Remark 3.

- *If $B[h_i(x)] \neq 1$ for some i, then clearly $x \notin S$.*
- *Otherwise, answer "$x \in S$" —might be a false positive!*

False positive example

Query: “$x_4 \in S$?”



Lookup(x_4): $h_1(x_4)=h_2(x_4)=h_3(x_4)=1$

Answer: “yes”

Probability of false positive

- After all elements from S have been hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{n}\right)^{km} \approx e^{-km/n} = p.$$

- To simplify the analysis, *assume* that the probability that a specific bit is still 0 is **exactly** p.
- The probability of a false positive is the probability that all k hashes evaluate to 1:

$$f = (1 - p)^k$$

Optimal number of hash functions

$$f = (1 - p)^k = (1 - e^{-km/n})^k$$

- Trade-off between k and p: using more hash functions
  - gives us more chances to find a 0 when $x \notin S$;
  - but reduces the number of 0s in the array!
- Compute optimal number k^* of hash functions by minimizing f as a function of k:

$$k^* = (n/m) \cdot \ln 2$$

- Then the **false positive probability** is given by

$$f = (1/2)^{k^*} \approx (0.6185)^{n/m}$$

Big savings in space

- **Space** required by Bloom filter *per element of* S: n/m bits.
- For example, set $n = 8m$. Then $k^* = 6$ and $f \approx 0.02$.

$\Rightarrow$ Small constant false positive probability by using only 8 bits (1 byte) per element of S, **independently** of the size of S!

Summary on Bloom filters

Bloom filter can answer approximate set membership in

- **"constant"** time (time to hash)
- **constant** space to represent an element from S
- **constant** false positive probability f.

Application 1 (historical): spell checker

- Spelling list of $210KB$, $25K$ words.
- Use 1 byte per word.
- Maintain $25KB$ Bloom filter.
- False positive = accept a misspelled word.

Application 2: implementing joins in database

- **Join:** Combine two tables with a common domain into a single table.
- **Semi-join:** A join in distributed DBs in which only the joining attribute from one site is transmitted to the other site and used for selection. The selected records are sent back.
- **Bloom-join:** A semi-join where we send only a BF of the joining attribute.

Example

Empl	Sal	Add	City
Bale	90K	...	New York
Jones	45K	...	New York
Fletcher	45K	...	Pittsburg
Rodriguez	80K	...	Chicago
Shaw	45K	...	Chicago

City	Cost Of Living
New York	60K
Chicago	55K
Pittsburg	40K

Create a table of all employees that make $< 50K$ and live in city where Cost Of Living = COL $> 50K$.

Empl	Sal	Add	City	COL

- **Join:** send (City, COL) for COL > 50.
- **Semi-join:** send just (City) for COL > 50.
- **Bloom-join:** send a Bloom filter for all cities with COL > 50