Lecture 8

HASHING!!!!!
Announcements

• HW3 due Friday!
• HW4 posted Friday!

• Q: Where can I see examples of proofs?
  • Lecture Notes
  • CLRS
  • HW Solutions

• Office hours: lines are long 😞
• Solutions:
  • We will be (more) mindful of throughput.
  • Get more TAs
  • Stop assigning homework
  • Use Piazza!
  • Start early. (There are no lines on Monday!)
Today: hashing

![Diagram of hashing with 9 buckets and 9 elements placed at positions 1, 2, 3, 9, 13, 22, 43, 9, NIL]
• **Hash tables** are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

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

• **Universal hash families** are even more magic.
Goal:
Just like on Monday

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

- INSERT 5
- DELETE 4
- SEARCH 52

node with key “2”

HERE IT IS

data structure
On Monday:

• Self balancing trees:
  • $O(\log(n))$ deterministic INSERT/DELETE/SEARCH

Today:

• Hash tables:
  • $O(1)$ expected time INSERT/DELETE/SEARCH
  • Worse worst-case performance, but often great in practice.

eg, Python’s dict, Java’s HashSet/HashMap, C++’s unordered_map
Hash tables are used for databases, caching, object representation, ...
One way to get $O(1)$ time

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

This is called “direct addressing.”
That should look familiar

- Kind of like **BUCKETSORT** from Lecture 6.
- Same problem: if the keys may come from a universe $U = \{1,2, \ldots, 10000000000\}$....

The universe is really big!
The solution then was...

- Put things in buckets based on one digit.

**INSERT:**

- 21
- 345
- 13
- 101
- 50
- 234
- 1

It’s in this bucket somewhere...
go through until we find it.

Now **SEARCH** 21
Now SEARCH 22 ....this hasn’t made our lives easier...
Hash tables

• That was an example of a hash table.
  • not a very good one, though.

• We will be more clever (and less deterministic) about our bucketing.

• This will result in fast (expected time) INSERT/DELETE/SEARCH.
But first! Terminology.

• We have a **universe U**, of size M.
  • M is really big.

• But only a few (say at most n for today’s lecture) elements of M are ever going to show up.
  • M is waaaaayyyyyyyy bigger than n.

• But we don’t know which ones will show up in advance.

All of the keys in the universe live in this blob.

A few elements are special and will actually show up.

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

The only ones which I care about are those which appear as trending hashtags on twitter. **#hashinghashtags**

*There are way fewer than \(128^{140}\) of these.*

Examples aside, I’m going to draw elements like I always do, as blue boxes with integers in them...
The previous example with this terminology

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

For this lecture, I’m assuming that the number of things is the same as the number of buckets, both are $n$. This doesn’t have to be the case, although we do want:

#buckets = $O(\ #\text{things which show up} )$
This is a hash table (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 $O(\text{length(list)})$.
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let’s stick with $h(x) = \text{least significant digit of } x$.

**INSERT:**

- 13
- 22
- 43
- 9

**SEARCH 43:**

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

For demonstration purposes only! This is a terrible hash function! Don’t use this!
Aside: Hash tables with open addressing

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

This is a “chain”
This is a **hash table** (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 $O(\text{length(list)})$.
- $h: \mathbb{U} \rightarrow \{1, \ldots, n\}$ can be any function:
  - but for concreteness let’s stick with $h(x) =$ least significant digit of $x$.

**INSERT:**

| 13 | 22 | 43 | 9 |

**SEARCH 43:**

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

For demonstration purposes only!
This is a terrible hash function! Don’t use this!
IPython notebook time

• (Seems to work!)
• (Will this example be a good idea?)
Sometimes this a **good idea**
Sometimes this is a **bad idea**

- How do we pick that function **so that this is a good idea**?
  1. We want there to be not many buckets (say, \( n \)).
    - This means we don’t use too much space
  2. We want the items to be **pretty spread-out** in the buckets.
    - This means it will be fast to SEARCH/INSERT/DELETE

\[
\begin{array}{c}
1 \\
2 \\
3 \\
\vdots \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
2 \\
3 \\
\vdots \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
21 \\
22 \\
13 \\
43 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
2 \\
3 \\
\vdots \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
13 \\
43 \\
23 \\
93 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
2 \\
3 \\
\vdots \\
9 \\
\end{array}
\]
Worst-case analysis

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

• If we had this, then we’d achieve our dream of $O(1)$ INSERT/DELETE/SEARCH

Can you come up with such a function?
YOU CANNOT ESCAPE THE DARK SIDE
WITH DETERMINISTIC HASH FUNCTIONS
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 WAAYYYYY bigger than $n$, so $M/n$ is bigger than $n$.
- Bad guy chooses $n$ of the items that landed in this very full bucket.
Solution:
Randomness
The game

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

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

3. HASH IT OUT

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

What does random mean here? Uniformly random?

Plucky the pedantic penguin

#hashpuns
Example

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

• ...

• $h(n)$ is also a **uniformly random** number between 1 and $n$, independent of $h(1)$, $h(2)$, ..., $h(n-1)$. 
Why should that help?

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

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

We’ll need to do some analysis...
What do we want?

It’s **bad** if lots of items land in $u_i$’s bucket. So we want **not that**.
More precisely

• We want:
  • For all $u_i$ that the bad guy chose
  • $E[\text{number of items in } u_i \text{'s bucket}] \leq 2$.

• If that were the case,
  • For each operation involving $u_i$
  • $E[\text{time of operation}] = O(1)$

So, in expectation, it would takes $O(1)$ time per INSERT/DELETE/SEARCH operation.
So we want:

- For all $i=1, \ldots, n$,
  
  $$E[\text{number of items in } u_i \text{'s bucket}] \leq 2.$$
Aside: why not:

- For all $i=1,...,n$:
  
  $E[\text{number of items in bucket } i] \leq ___$?

Suppose that:

- this happens with probability $1/n$
- and this happens with probability $1/n$
- etc.

Then $E[\text{number of items in bucket } i] = 1$ for all $i$. **But $P\{\text{the buckets get big}\} = 1$.**
Expected number of items in $u_i$’s bucket?

1. $E[\cdot] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j) \}$
2. $= 1 + \sum_{j\neq i} P\{ h(u_i) = h(u_j) \}$
3. $= 1 + \frac{n-1}{n}$
4. $\leq 2.$

That’s what we wanted.

$h$ is uniformly random

That’s what we wanted.

You will verify this on HW.
That’s great!

• For all $i=1, \ldots, n$,
  • $E[\text{number of items in } u_i \text{'s bucket}] \leq 2$

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

So, the solution is:

*pick a uniformly random hash function.*
The elephant in the room
The elephant in the room.

“How do we do that?”

“Pick a uniformly random hash function”

How do we do that?
Let’s implement this!

• IPython Notebook for Lecture 8
Let’s **NOT** implement this!

**Issues:**

- Suppose $U = \{ \text{all of the possible hashtags} \}$
- If we completely choose the random function up front, we have to iterate through all of $U$.
  - $128^{140}$ possible ASCII strings of length 140.
  - (More than the number of particles in the universe)
- And even ignoring the time considerations
  - We have to store $h(x)$ for every $x$. 
Another thought...

• Just remember $h$ on the relevant values

$\begin{align*}
13 & \quad h(13) = 6 \\
22 & \quad h(22) = 3 \\
43 & \\
92 & \\
7 & \\
\end{align*}$

$\begin{align*}
h(7) & = 8 \\
h(43) & = 2 \\
h(92) & = 3 \\
h(22) & = 3 \\
h(13) & = 6
\end{align*}$

We need some way of storing keys and values with $O(1)$ INSERT/DELETE/SEARCH...

Algorithm now

Algorithm later
How much space does it take to store $h$?

- For each element $x$ of $U$:
  - store $h(x)$
  - (which is a random number in $\{1,\ldots,n\}$).

- Storing a number in $\{1,\ldots,n\}$ takes $\log(n)$ bits.
- So storing $M$ of them takes $M\log(n)$ bits.
- In contrast, direct addressing would require $M$ bits.
Hang on now

• Sure, *that* way of storing the function h won’t work.
• But maybe there’s another way?
Aside: description length

• Say I have a set S with s things in it.
• I get to write down the elements of S however I like.
  • (in binary)
• How many bits do I need?

On board: the answer is $\log(s)$

I’ll call this one “Fido”
Or, 01101011

This one is named “Hercules”
Or, 101
Space needed to store a random fn h?

• Say that this elephant-shaped blob represents the set of all hash functions.
• It has size $n^M$. (Really big!)
• To write down a random hash function, we need $\log(n^M) = M\log(n)$ bits. 😞
Solution

- Pick from a smaller set of functions.

A cleverly chosen subset of functions. We call such a subset a hash family.

All of the hash functions $h : U \rightarrow \{1, \ldots, n\}$

We need only $\log|H|$ bits to store an element of $H$. 
Outline

• **Hash tables** are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
  • like self-balancing binary trees
  • The difference is we can get better performance in expectation by using randomness.

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

• **Universal hash families** are even more magic.
Hash families

• A hash family is a collection of hash functions.

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

\[ h:U \rightarrow \{1,\ldots,n\} \]
Example: a smaller hash family

- \( H = \{ \text{function which returns the least sig. digit,} \)
  \begin{align*}
  \text{function which returns the most sig. digit} \}
  \end{align*}

- Pick \( h \) in \( H \) at random.

- Store just one bit to remember which we picked.

\[ h:U \to \{1,\ldots,n\} \]
The game

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

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

3. HASH IT OUT #hashpuns

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

I picked $h_1$
The game

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

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

3. HASH IT OUT

This adversary could have been more adversarial!
Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

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

- **Universal hash families** are even more magic.
How to pick the hash family?

• Definitely not like in that example.
• Let’s go back to that computation from earlier....
Expected number of items in $u_i$’s bucket?

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

So the number of items in $u_i$’s bucket is $O(1)$. You will verify this on HW.
How to pick the hash family?

• Let’s go back to that computation from earlier....

\[ E[ \text{number of things in bucket } h(u_i) ] \]

\[ = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j) \} \]

\[ = 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \} \]

\[ \leq 1 + \sum_{j \neq i} \frac{1}{n} \]

\[ = 1 + \frac{n-1}{n} \leq 2. \]

• All we needed was that this \( \leq \frac{1}{n} \).
Strategy

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

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

• A hash family $H$ that satisfies this is called a \textbf{universal hash family}.

• Then we still get $O(1)$-sized buckets in expectation.

• But now the space we need is $\log(|H|)$ bits.
  • Hopefully pretty small!
So the whole scheme will be

Choose $h$ randomly from a universal hash family $H$.

We can store $h$ in small space since $H$ is so small.

Probably these buckets will be pretty balanced.
Universal hash family
Let’s stare at this definition

• H is a *universal hash family* if:
  • When h is chosen uniformly at random from H,

\[
P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
\]

for all \( u_i, u_j \in U \) with \( u_i \neq u_j \),

You actually saw this in your pre-lecture exercise!

Toads = hash fns
Ice cream = items
”Like” and “Dislike” = buckets
Check our understanding...

• H is a **universal hash family** if:
   • When h is chosen uniformly at random from H,
     
     \[
     \text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j, \quad P_{h \in H}\{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
     \]

• H is **[something else]** if:
   • When h is chosen uniformly at random from H,
     
     \[
     \text{for all } u \in U, \text{ for all } x \in \{0, \ldots, n - 1\}, \quad P_{h \in H}\{ h(u_i) = x \} \leq \frac{1}{n}
     \]

*Are these different?*
Pre-lecture exercise

Statement 1: \( P[ \text{random toad likes vanilla} ] = \frac{1}{2}, \quad P[ \text{random toad likes chocolate} ] = \frac{1}{2} \)
\[ P[ \text{“vanilla” lands in the bucket “like”} ] = \frac{1}{2} \]

Statement 2: \( P[ \text{random toad feels the same about chocolate and vanilla} ] = \frac{1}{2} \)
\[ P[ \text{vanilla and chocolate land in the same bucket} ] = \frac{1}{2} \]
Pre-lecture exercise

Statement 1: \( P[\text{random toad likes vanilla}] = \frac{1}{2},\; P[\text{random toad likes chocolate}] = \frac{1}{2} \)

\( P[\text{“vanilla” lands in the bucket “like”}] = \frac{1}{2} \)

Statement 2: \( P[\text{random toad feels the same about chocolate and vanilla}] = \frac{1}{2} \)

\( P[\text{vanilla and chocolate land in the same bucket}] = \frac{1}{2} \)

Seem like they might be the same...?
Pre-lecture exercise

Statement 1: \( P[ \text{random toad likes vanilla }] = \frac{1}{2}, P[ \text{random toad likes chocolate }] = \frac{1}{2} \)
\( P[ \text{“vanilla” lands in the bucket “like” }] = \frac{1}{2} \)

Statement 2: \( P[ \text{random toad feels the same about chocolate and vanilla }] = \frac{1}{2} \)
\( P[ \text{vanilla and chocolate land in the same bucket }] = \frac{1}{2} \)

But no! 1 is true but 2 is not.

Universe = \{ vanilla, chocolate \}
Buckets = \{ like, dislike \}
Toads = different possible ways of distributing items
Check our understanding...

• H is a universal hash family if:
  • When h is chosen uniformly at random from H,
    \[
    \Pr_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
    \]
    for all \( u_i, u_j \in U \) with \( u_i \neq u_j \),

• H is [something else] if:
  • When h is chosen uniformly at random from H,
    \[
    \Pr_{h \in H} \{ h(u) = x \} \leq \frac{1}{n}
    \]
    for all \( u \in U \), for all \( x \in \{0, \ldots, n - 1\} \),
Example

• Uniformly random hash function $h$
  • [We just saw this]
  • [Of course, this one has other downsides...]

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

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

for all $u_i, u_j \in U$ with $u_i \neq u_j$. 
Non-example

• $h_0 = \text{Most\_significant\_digit}$
• $h_1 = \text{Least\_significant\_digit}$
• $H = \{h_0, h_1\}$
  • [discussion on board]

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

\[
\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j, \quad P_{h \in H}\{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
\]
A small universal hash family??

• Here’s one:
  • Pick a prime $p \geq M$.
  • Define
    \[
    f_{a,b}(x) = ax + b \mod p
    \]
    \[
    h_{a,b}(x) = f_{a,b}(x) \mod n
    \]
  • Claim:
    \[
    H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \}
    \]
    is a universal hash family.
Say what?

• Example:  \( M = p = 5, \ n = 3 \)

• To draw \( h \) from \( H \):
  • Pick a random \( a \) in \{1,\ldots,4\}, \( b \) in \{0,\ldots,4\}

• As per the definition:
  • \( f_{2,1}(x) = 2x + 1 \mod 5 \)
  • \( h_{2,1}(x) = f_{2,1}(x) \mod 3 \)

\( U = \begin{bmatrix} 1 & 2 & 3 & 4 & 0 \end{bmatrix} \mod 3 \)

This step just scrambles stuff up. No collisions here!

This step is the one where two different elements might collide.
Ignoring why this is a good idea

• Can we store h with small space?

  ![Mouse with a thought bubble: a = 2, b = 1]

• Just need to store two numbers:
  • a is in \{1,\ldots,p-1\}
  • b is in \{0,\ldots,p-1\}
  • So about 2\log(p) bits
  • By our choice of p, that’s \(O(\log(M))\) bits.

Compare: direct addressing was \(M\) bits!
Twitter example: \(\log(M) = 140 \log(128) = 980\) vs \(M = 128^{140}\)
Another way to see this using only the size of $H$

- We have $p-1$ choices for $a$, and $p$ choices for $b$.
- So $|H| = p(p-1) = O(M^2)$
- Space needed to store an element $h$:
  - $\log(M^2) = O(\log(M))$. 

$O(M \log(n))$ bits per function

$O(\log(M))$ bits per function
Why does this work?

• This is actually a little complicated.
  • There are some hidden slides here about why.
  • Also see the lecture notes.

• The thing we have to show is that the collision probability is not very large.

• **Intuitively**, this is because:
  • for any (fixed, not random) pair \( x \neq y \) in \( \{0,\ldots,p-1\} \),
  • If \( a \) and \( b \) are random,
  • \( ax + b \) and \( ay + b \) are independent random variables. (why?)
Why does this work?

• Want to show:
  
  • for all \( u_i, u_j \in U \) with \( u_i \neq u_j \), \( P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n} \)

• aka, the probability of any two elements colliding is small.

• Let’s just fix two elements and see an example.
  
  • Let’s consider \( u_i = 0, \ u_j = 1 \).
The probability that 0 and 1 collide is small

- Want to show:
  - \( P_{h \in H} \{ h(0) = h(1) \} \leq \frac{1}{n} \)
  - For any \( y_0 \neq y_1 \in \{0,1,2,3,4\} \), how many \( a,b \) are there so that \( f_{a,b}(0) = y_0 \) and \( f_{a,b}(1) = y_1 \) ?
- **Claim**: it’s exactly one.
  - Proof: solve the system of eqs. for \( a \) and \( b \).

\[
\begin{align*}
\begin{cases}
a \cdot 0 + b &= y_0 \mod p \\
a \cdot 1 + b &= y_1 \mod p
\end{cases}
\end{align*}
\]

\( U = \) eg, \( y_0 = 3, y_1 = 1 \).
The probability that 0 and 1 collide is small

• Want to show:
  • \( P_{h \in H} \{ h(0) = h(1) \} \leq \frac{1}{n} \)

• For any \( y_0 \neq y_1 \in \{0,1,2,3,4\} \), exactly one pair \( a,b \) have \( f_{a,b}(0) = y_0 \) and \( f_{a,b}(1) = y_1 \).

• If 0 and 1 collide it’s b/c there’s some \( y_0 \neq y_1 \) so that:
  • \( f_{a,b}(0) = y_0 \) and \( f_{a,b}(1) = y_1 \).
  • \( y_0 = y_1 \mod n \).

eg, \( y_0 = 3, y_1 = 1 \).
The probability that 0 and 1 collide is small

- Want to show:
  - \( P_{h \in H} \{ h(0) = h(1) \} \leq \frac{1}{n} \)

- The number of \( a, b \) so that 0,1 collide under \( h_{a,b} \) is at most the number of \( y_0 \neq y_1 \) so that \( y_0 = y_1 \mod n \).

- How many is that?
  - We have \( p \) choices for \( y_0 \), then at most \( \frac{1}{n} \) of the remaining \( p-1 \) are valid choices for \( y_1 \)...
  - So at most \( p \cdot \left( \frac{p-1}{n} \right) \).

This slide skipped in class – here for reference!
The probability that 0 and 1 collide is small

• Want to show:
  \[ P_{h \in H} \{ h(0) = h(1) \} \leq \frac{1}{n} \]

• The # of \((a, b)\) so that 0,1 collide under \(h_{a, b}\) is \( \leq p \cdot \left( \frac{p-1}{n} \right) \).

• The probability (over a,b) that 0,1 collide under \(h_{a,b}\) is:

  \[
  P_{h \in H} \{ h(0) = h(1) \} \leq \frac{p \cdot \left( \frac{p-1}{n} \right)}{|H|}
  = \frac{p \cdot \left( \frac{p-1}{n} \right)}{p(p-1)}
  = \frac{1}{n}.
  \]
The same argument goes for any pair

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

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

That’s the definition of a universal hash family.

So this family $H$ indeed does the trick.
But let’s check that it **does** work

- Back to IPython Notebook for Lecture 8...

\[
M = 200, \ n = 10
\]

---

![Graph showing empirical probability of collision out of 100 trials for two hash families. The orange bar represents the 'not good hash family' and the blue bar represents the 'universal hash family'. The graph indicates a comparison of the number of pairs (x,y) for different empirical probability values.]
So the whole scheme will be

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

We can store $h$ in space $O(\log(M))$ since we just need to store $a$ and $b$.

Probably these buckets will be pretty balanced.
Outline

• **Hash tables** are another sort of data structure that allows fast **INSERT/DELETE/SEARCH**.
  • like self-balancing binary trees
  • The difference is we can get better performance in expectation by using randomness.

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

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

Recap
Want O(1) INSERT/DELETE/SEARCH

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

• INSERT 5
• DELETE 4
• SEARCH 52

HERE IT IS data structure
We studied this game

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

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

3. HASH IT OUT

   INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92
Uniformly random h was good

• If we choose h uniformly at random, for all $u_i, u_j \in U$ with $u_i \neq u_j$,
  $$P_{h \in H}\{ h(u_i) = h(u_j) \} \leq \frac{1}{n}$$

• That was enough to ensure that, in expectation, a bucket isn’t too full.

A bit more formally:

For any sequence of INSERT/DELETE/SEARCH operations on any $n$ elements of U, the expected runtime (over the random choice of h) is $O(1)$ per operation.
Uniformly random $h$ was bad

- If we actually want to implement this, we have to store the hash function $h$.
- That takes a lot of space!
  - We may as well have just initialized a bucket for every single item in $U$.
- Instead, we chose a function randomly from a smaller set.
We needed a smaller set that still has this property

• If we choose \( h \) uniformly at random, for all \( u_i, u_j \in U \) with \( u_i \neq u_j \),

\[
P_{h \in H} \{ h(u_i) = h(u_j) \} \leq \frac{1}{n}
\]

This was all we needed to make sure that the buckets were balanced in expectation!

• We call any set with that property a universal hash family.

• We gave an example of a really small one 😊
Conclusion:

• We can build a hash table that supports \texttt{INSERT/DELETE.SEARCH} in O(1) expected time,
  • if we know that only n items are every going to show up, where n is waaaayyyyyyy less than the size M of the universe.

• The space to implement this hash table is \texttt{O(n \log(M))} bits.
  • O(n) buckets
  • O(n) items with \log(M) bits per item
  • O(\log(M)) to store the hash fn.

• M is waaayyyyyyy bigger than n, but \log(M) probably isn’t.
That’s it for data structures (for now)

Achievement unlocked
Data Structure: RBTrees and Hash Tables

Now we can use these going forward!
Next Time

- Graph algorithms!

Before Next Time

- Pre-lecture exercise for Lecture 9
  - Intro to graphs