Lecture 8

HASHING!!!!!
Announcements

• HW3 due Friday!
• HW4 posted Friday!
Today: hashing

n=9 buckets

1: NIL
2: 22 → NIL
3: 13 → 43 → NIL
9: 9 → NIL

n=9 buckets
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.
    • Like QuickSort vs. MergeSort

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

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

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

#prettysweet

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 solution then was...

- Put things in buckets based on one digit.

**INSERT:**

1 2 3 4 5 6 7 8 9 0

Now **SEARCH** 21

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

INSERT:

Now SEARCH

....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 waaaayyyyyyyyy 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. (\( 128^{140} \) of them).

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

There are way fewer than \( 128^{140} \) of these.
The previous example with this terminology

- We have a **universe** $U$, of size $M$.
  - at most $n$ of which will show up.
- $M$ is **waaaayyyyy** 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:

$\#\text{buckets} = O(\ #\text{things which show up} \ )$
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 $O(\text{length(list)})$.
- $h: U \rightarrow \{1, \ldots, n\}$ 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”
• You’ll see it on your homework 😊

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 $O(\text{length(list)})$.
- $h: U \rightarrow \{1, \ldots, n\}$ 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 $h(43) = 3$.

This is a good idea as long as there are not too many elements in that bucket!
The main question

- 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

1. vs. 2.
Worst-case analysis

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

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

Take a minute to talk to the person next to you. Can you come up with such a function?
YOU CANNOT ESCAPE THE DARK SIDE

WITH DETERMINISTIC HASH FUNCTIONS
We really can’t beat Darth Vader here.

- The universe $U$ has $M$ items
- They get hashed into $n$ buckets
- At least one bucket receives at least $\frac{M}{n}$ items
- $M$ is WAAYYYYY bigger than $n$, so $\frac{M}{n}$ is bigger than $n$.
- Darth Vader 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 $h : U \to \{1, \ldots, n\}$.

3. HASH IT OUT

INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92
Why should this help?

- 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 \( h(1), h(2), ..., h(n-1) \).
What do we want?

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

- Suppose that for all $u_i$ that the bad guy chose
  - $E[\text{number of items in } u_i \text{'s bucket }] \leq 2.$
- Then for each operation involving $u_i$
  - $E[\text{time of operation }] = O(1)$

- By linearity of expectation,
  - $E[\text{time to do a bunch of operations}]$
    - $= E[\sum_{\text{operations}} \text{time of operation}]$
    - $= \sum_{\text{operations}} E[\text{time of operation}]$
    - $= \sum_{\text{operations}} O(1)$
    - $= O(\text{number of operations})$

aka, $O(1)$ per 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 just:

- For all $i=1,...,n$:

$$E[\text{number of items in bucket } i] \leq 2?$$

Suppose:

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

this happens with probability $1/n$

and this happens with probability $1/n$

etc.
So we want:

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

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

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 $L$ INSERT/DELETE/SEARCH operations on any $n$ elements of $U$, the expected runtime (over the random choice of $h$) is $O(L)$. 

aka, anything Darth Vader might pick in Step 1 of the game.

aka, O(1) per operation.
The elephant in the room
The elephant in the room

“Pick a uniformly random hash function”

h(1) = 2    h(11) = 4    h(4511) = 3
h(2) = 7    h(12) = 5    h(4512) = 7
h(3) = 9    h(13) = 7    h(4513) = 2
h(4) = 1    h(14) = 3    h(4514) = 6
h(5) = 0    h(15) = 2    h(4515) = 3
h(6) = 7    h(16) = 9    h(4516) = 1
h(7) = 2    h(17) = 3    h(4517) = 0
h(8) = 3    h(18) = 2    h(4518) = 0
h(9) = 7    h(19) = 1    h(4519) = 3
h(10) = 3   h(20) = 5   h(4520) = 1
Randomization is fine... but we need to be able to store our choice of $h$!

- Say that this elephant-shaped blob represents the set of all hash functions.
- How big is this set?
  - $n^{|U|} = n^M = \text{REALLY BIG}$.
- In order to write down an arbitrary element of a set of size $A$, we need $\log(A)$ bits.
- So we’d need about $M \log(n)$ bits to remember one of these hash functions. **THAT’S ENOUGH TO DO DIRECT ADDRESSING!!!!**
Another thought...

• Just remember $h$ on the relevant values

But that’s what we wanted to begin with...

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

Algorithm now

Algorithm later
Solution

• Pick from a smaller set of functions.

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

We need only \( \log |H| \) bits to store an element of \( H \).
How to pick the hash family?

• 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....

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

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

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

• \[ \leq 1 + \sum_{j \neq i} 1/n \]

• \[ = 1 + \frac{n-1}{n} \leq 2. \]

• All we needed was that this \leq 1/n.
Strategy

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

$$\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j,$$

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

• 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.
What is this universal hash family?

Here’s one:

• Pick a prime \( p \geq M \).

• Define

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

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

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

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...

how big is $H$?

• We have $p-1$ choices for $a$, and $p$ choices for $b$.
• So $|H| = p(p-1) = O(M^2)$
• This is much better than $n^M$!!!!
• space needed to store $h$: $O(\log(M))$. 

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

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

• This is actually a little complicated.
• I’ll go over the argument now, because it’s a good example of how to reason about hash functions.
  • Fancy counting!
• BUT! don’t worry if you don’t follow all the calculations right now.
  • You can always take a look back at the slides or lecture notes later.

• The important part is the structure of the argument.
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 $u_i=0$, $u_j=1$.

Let’s consider $U = \{0, 1, 2, 3, 4\}$ with $f_{a,b}(x) = ax + b \mod p$, where $a = 2$, $b = 1$, and $p = 3$.

Apply $f_{a,b}(x)$ to $U$.

- $f_{2,1}(0) = 2 \mod 3 = 2$
- $f_{2,1}(1) = 2 \mod 3 = 2$
- $f_{2,1}(2) = 2 \mod 3 = 2$
- $f_{2,1}(3) = 2 \mod 3 = 2$
- $f_{2,1}(4) = 2 \mod 3 = 2$

The result is $\{2, 2, 2, 2, 2\}$.

Convince yourself that it will be the same for any pair!
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 \( b \).

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

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 \( f_{a, b}(1) = y_1 \).
    - \( y_0 = y_1 \mod n \).

\( U = \) \[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
\end{array}
\]

\( f_{a, b}(x) = \) \[
ax + b \mod p
\]

\( \mod 3 \)

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 $1/n$ of the remaining $p-1$ are valid choices for $y_1$...
  - So at most $p \cdot \left( \frac{p-1}{n} \right)$.

\[ f_{a,b}(x) = ax + b \mod p \]

$U = \{ 0, 1, 2, 3, 4 \}$

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 # 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.
So the whole scheme will be

Choose \( h \) randomly from \( H \)

Universe \( U \) of size \( M \)

We can store \( h \) in space \( O(\log(M)) \).

The expected time to do any \( L \) operations on these \( n \) elements is \( O(L) \).
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 $h: U \to \{1, \ldots, n\}$.

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 $L$ INSERT/DELETE/SEARCH operations on any $n$ elements of $U$, the expected runtime (over the random choice of $h$) is $O(L)$.

aka, $O(1)$ 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.

All of the hash functions $h: U \rightarrow \{1, ..., n\}$
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 were able to come up with 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 waaaayyyyyy less than the size $M$ of the universe.

• The space to implement this hash table is $O(n \log(M))$.

• $M$ is waaayyyyyy bigger than $n$, but $\log(M)$ probably isn’t.
Next Week

• Graph algorithms!