Announcements

• HW4 due Wednesday.
• There will not be new HW posted on Wednesday, because it’ll be time to study for the...
Midterm!

- Take-home
- Timed: 3 hours
- You may use one double-sided “cheat sheet” that you have prepared yourself.
- Otherwise closed notes, closed internet, closed discussion, etc.
- Basic structure:
  - Some multiple choice
  - Some short answer
  - Some proofs
  - Some algorithm design
  - One challenge problem

- A practice exam with this structure will be posted soon.
- Keep an eye out for a Piazza post with more details about the logistics.
  - There was already a Piazza post with some of the basic logistics.
How to study for the midterm?

• Go over lecture + homework + section material
• DO PRACTICE PROBLEMS.
  • Algorithms Illuminated, CLRS have great problems in them!
  • We will post a practice exam soon.
• Review Session!
  • No regular section this week.
  • Instead, there will be a (recorded!) review session on Thursday 3pm-ish. (Exact time/room TBD)
• Thursday HW party is a Midterm Review Party!
  • Monday 2/10 HW party next week is canceled
• Go to office hours!
  • Office Hour Schedule will be the same through Sunday
  • No OH Monday 2/10 or Tuesday 2/11 next week.
Today: hashing

n=9 buckets

[Diagram of a hash table with 9 buckets, each containing a value or NIL, indicating successful hashing of keys 9, 13, 22, 43 into the table with no collisions.]
• **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 magical.
Goal

• We want to store nodes with keys in a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT 5
- DELETE 4
- SEARCH 52

node with key “2”

HERE IT IS

data structure
Last time

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

#prettysweet

Today:

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

#evensweeterinpractice

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.”

2 isn’t in the data structure.

3 is here.
That should look familiar

• Kind of like COUNTINGSORT from Lecture 6.
• Same problem: if the keys may come from a “universe” $U = \{1, 2, \ldots, 10000000000\}$, it takes a lot of space.
Solution?

Put things in buckets based on one digit

**INSERT:**

```
21  345  13  101  50  234  1
```

```
0  1  2  3  4  5  6  7  8  9
```

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

Now **SEARCH** 21
Problem:

INSERT:

22  34  102  12  102  52  232  2

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.

- U is a *universe* of size M.
  - M is really big.
- But only a few (at most n) 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.

Example: U is the set of all strings of at most 280 ascii characters. \((128^{280} \text{ of them})\).

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

There are way fewer than \(128^{280}\) of these.
Hash Functions

• A hash function $h : U \rightarrow \{1, \ldots, n\}$ is a function that maps elements of $U$ to buckets 1, ..., $n$.

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 = $O(\#things which show up)$

Example:

$h(x) =$ least significant digit of $x$.

$h(13) = 3$

$h(22) = 2$

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

Universe $U$
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 $O(\text{length(list)})$.
- A hash function $h: U \rightarrow \{1, ..., n\}$.
  - For example, $h(x) =$ least significant digit of $x$.

**INSERT:**

13  22  43  9

**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.

\begin{center}
\begin{tikzpicture}
\node[draw] (1) at (0,0) {1};
\node[draw] (2) at (0,-1) {2};
\node[draw] (3) at (0,-2) {3};
\node[draw] (9) at (0,-4) {9};
\node[draw] (13) at (1,-2) {13};
\node[draw] (43) at (2,-2) {43};
\draw (1) -- (2);
\draw (2) -- (3);
\draw (3) -- (9);
\draw (9) -- (13);
\draw (13) -- (43);
\node at (0,-3) {This is a “chain”};
\node at (0,-5) {n=9 buckets};
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
\node[draw] (1) at (0,0) {1};
\node[draw] (2) at (0,-1) {2};
\node[draw] (3) at (0,-2) {3};
\node[draw] (9) at (0,-4) {9};
\node[draw] (13) at (1,-2) {13};
\node[draw] (43) at (2,-2) {43};
\draw (1) -- (2);
\draw (2) -- (3);
\draw (3) -- (9);
\draw (9) -- (13);
\draw (13) -- (43);
\node at (0,-3) {bounce!};
\node at (0,-5) {n=9 buckets};
\end{tikzpicture}
\end{center}
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(length(list)).
- A hash function $h: U \rightarrow \{1, \ldots, n\}$.
  - For example, $h(x) =$ least significant digit of x.

**INSERT:**

13 22 43 9

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

**DELETE 43:**
Search for 43 and remove it.

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

n=9 buckets

n=9 buckets
Worst-case analysis

• Goal: Design a function \( h: U \rightarrow \{1, \ldots, n\} \) 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?

Think-Pair-Share Terrapins
2 min. think. 1 min. pair+share
This is impossible!

No deterministic hash function can defeat worst-case input!
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.**
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

   - $u_k$, $u_i$, $u_j$
Example of a random hash function

• Say that $h: U \to \{1, \ldots, n\}$ 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 $h(1)$, $h(2)$.
  
  • ...
  
  • $h(M)$ is also a uniformly random number between 1 and $n$, independent of $h(1)$, $h(2)$, ..., $h(M-1)$. 
Randomness helps

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 ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$, $E[\text{number of items in } u_i\text{'s bucket}] \leq 2$.

- If that were the case:
  - For each INSERT/DELETE/SEARCH operation involving $u_i$, $E[\text{time of operation}] = O(1)$

We could replace “2” here with any constant; it would still be good. But “2” will be convenient.

This is what we wanted at the beginning of lecture!
So we want:

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

• For all $i=1,\ldots,n$,

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

vs

• For all $i=1,\ldots,n$:

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

Suppose that:

then $E[\text{number of items in bucket } i] = 1$ for all $i$.

But $E[\text{number of items in 43's bucket}] = n$
This distinction came up on your pre-lecture exercise!

• Solution to pre-lecture exercise (skipped in class):
  • $E[\text{number of items in bucket 1}] = \frac{n}{6}$
  • $E[\text{number of items that land in the same bucket as item 1}] = n$
So we want:

• For all $i=1, \ldots, n$,

\[ E[ \text{number of items in } u_i\text{'s bucket } ] \leq 2. \]
Expected number of items in $u_i$’s bucket?

- $E[\cdot] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j) \}$
- $= 1 + \sum_{j \neq i} \frac{1}{n}$ (You will verify this on HW.)
- $= 1 + \frac{n-1}{n} \leq 2$. That’s what we wanted!

$h$ is uniformly random
A uniformly random hash function leads to balanced buckets

• We just showed:
  • For all ways a bad guy could choose \( u_1, u_2, \ldots, u_n \) to put into the hash table, and for all \( i \in \{1, \ldots, n\} \),
    \[
    \mathbb{E}[ \text{number of items in } u_i \text{'s bucket } ] \leq 2.
    \]

• Which implies:
  • No matter what sequence of operations and items the bad guy chooses,
    \[
    \mathbb{E}[ \text{time of INSERT/DELETE/SEARCH } ] = O(1)
    \]

• 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 $h: U \rightarrow \{1, \ldots, n\}$?

• 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 $h(1)$, $h(2)$, ..., $h(n-1)$. 

Think-Pair-Share Terrapins
1 minute think
1 minute pair and share
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:

<table>
<thead>
<tr>
<th>x</th>
<th>h(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAA</td>
<td>1</td>
</tr>
<tr>
<td>AAAAAAB</td>
<td>5</td>
</tr>
<tr>
<td>AAAAAAC</td>
<td>3</td>
</tr>
<tr>
<td>AAAAAAD</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ZZZZZZ</td>
<td>7</td>
</tr>
<tr>
<td>ZZZZZZZZ</td>
<td>3</td>
</tr>
</tbody>
</table>

- 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.
Another way to say this

- There are lots of hash functions.
- There are $n^M$ of them.
- Writing down a random one of them takes $\log(n^M)$ bits, which is $M \log(n)$.
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 $H$. All of the hash functions $h: U \rightarrow \{1,...,n\}$
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.
Example: a smaller hash family

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

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

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

\[ h_0 = \text{Most_significant_digit} \]
\[ h_1 = \text{Least_significant_digit} \]
\[ H = \{ h_0, h_1 \} \]

I picked \( h_1 \)

\[ \text{INSERT} 19, \text{INSERT} 22, \text{INSERT} 42, \]
\[ \text{INSERT} 92, \text{INSERT} 0, \text{SEARCH} 42, \]
\[ \text{DELETE} 92, \text{SEARCH} 0, \text{INSERT} 92 \]

#hashpuns
This is not a very good hash family

- $H = \{ \text{function which returns least sig. digit,} \$
  
  \text{function which returns most sig. digit} \}$

- On the previous slide, the adversary could have been a lot more adversarial...
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.

$$h_0 = \text{Most\_significant\_digit}$$
$$h_1 = \text{Least\_significant\_digit}$$
$$H = \{h_0, h_1\}$$

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

I picked $h_1$

3. HASH IT OUT #hashpuns

| 101 | 11 | 121 | 141 | 131 |

0 1 2 9

11 101 141 121 131
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?

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

All that we needed was that this is $1/n$. COLLISION!
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 **universal hash family**.
So the whole scheme will be

Choose \( h \) randomly from a universal hash family \( H \)

We can store \( h \) using \( \log|H| \) bits.

Probably these buckets will be pretty balanced.
Universal hash family

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

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

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

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

  $\forall u_i, u_j \in U$ \hspace{1em} with $u_i \neq u_j$,

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

- $H = \text{the set of all functions } h: U \rightarrow \{1, \ldots, n\}$
  
  - We saw this earlier – it corresponds to picking a uniformly random hash function.
  
  - Unfortunately this $H$ is really really large.
Non-example

• $h_0 = \text{Most\_significant\_digit}$
• $h_1 = \text{Least\_significant\_digit}$
• $H = \{h_0, h_1\}$

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

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

Prove that this choice of $H$ is NOT a universal hash family!

2 minutes think
1 minute pair and share
Non-example

• $h_0 =$ Most_significant_digit
• $h_1 =$ 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
    $$f_{a,b}(x) = ax + b \mod p$$
    $$h_{a,b}(x) = f_{a,b}(x) \mod n$$
  • Define:
    $$H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \}$$

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

\( U = 1, 2, 3, 4, 5 \)
\( a = 2, \ b = 1 \)

This step just scrambles stuff up. No collisions here!

This step is the one where two different elements might collide.
h takes $O(\log M)$ bits to store

• 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 $p$ (close to $M$), that’s $O(\log(M))$ bits.

• Compare: direct addressing was $M$ bits!
  • Twitter example: $\log(M) = 140 \log(128) = 980$ vs $M = 128^{280}$
Another way to see that

\[ h \text{ takes } O(\log M) \text{ bits to store} \]

• 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 \( h \) in \( H \):
  • \( \log(M^2) = O(\log M) \).
Why does this work?

• This is actually a little complicated.
  • See CLRS (Thm 11.5) if you are curious.
  • You are NOT RESPONSIBLE for the proof in this class.
  • But you should know that a universal hash family of size $O(M^2)$ exists.

Try to prove that this is a universal hash family!
But let’s check that it **does** work

- Check out the IPython notebook for lecture 8

![Graph showing empirical probability of collision](image)

M=200, n=10

Number of pairs of \((x,y)\).

Out of \(\binom{200}{2} = 19900\) pairs

Empirical probability of collision out of 100 trials
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.
Want $O(1)$ \textbf{INSERT/DELETE/SEARCH}.

- We are interesting in putting nodes with keys into a data structure that supports fast \textbf{INSERT/DELETE/SEARCH}.

\begin{itemize}
  \item INSERT 5
  \item DELETE 4
  \item SEARCH 52
\end{itemize}

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

| 13 | 22 | 43 | 92 | 7 |

- 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 all INSERT/DELETE/SEARCH operations took $O(1)$ time in expectation, even on adversarial inputs.
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.
Universal Hash Families

H is a universal hash family if:

- If we choose \( h \) uniformly at random in \( H \), 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 gave an example of a really small universal hash family, of size \( O(M^2) \)

- That means we need only \( O(\log M) \) bits to store it.
Conclusion:

- We can build a hash table that supports **INSERT/DELETE/SEARCH** in $O(1)$ expected time.
- Requires $O(n \log(M))$ bits of space.
  - $O(n)$ buckets
  - $O(n)$ items with $\log(M)$ bits per item
  - $O(\log(M))$ to store the hash function

Hashing a universe of size $M$ into $n$ buckets, where at most $n$ of the items in $M$ ever show up.
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