Lecture 8
Hashing
Announcements

• HW3 due 30 minutes ago!
• There will not be new HW posted today, because it’s time to study for the...
Midterm!

• Thursday, May 4, 6-9pm!
• Stay tuned for location information.

• See website for details:
  • Covers up through today (Lecture 8)
  • Practice exam will be posted real soon now!
    • Plus additional old exams, *caveat discipula*
How to study for the midterm?

• Go over lecture + homework + section + textbook material

• DO PRACTICE PROBLEMS.
  • Algorithms Illuminated, CLRS have great problems in them!
  • Practice exam(s).

• HW party this week is a Midterm Review Party!

• Monday’s class is ½ review!

• Office Hours!
  • Most effective if you come with specific questions/topics
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.

• **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”

data structure

HERE IT IS
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

3 is here.
2 isn’t in the data structure.
That should look familiar

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

Put things in buckets based on one digit

**INSERT:**

21 345 13 101 50 234 1

Now **SEARCH**

21

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

INSERT:

22  34  12  102  2
232 52  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 **U** are ever going to show up.
  - **M** is waaaayyyyyyy 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.

Universe **U**

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

Example:
- $h(x) = \text{least significant digit of } x$.
- $h(13) = 3$
- $h(22) = 2$

Note! For this lecture, $n$ is both #buckets and #(things that might show up).
- That doesn’t need to be the case, but in general we should think of those two things as being on the same order.
Hash Tables (with chaining)

A hash table consists of:

- An 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 $h: U \rightarrow \{1, \ldots, n\}$.
  - 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.
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.

\[\text{n=9 buckets}\]

This is a “chain”

\[\text{n=9 buckets}\]
A **hash table** consists of:

- 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 $h: U \rightarrow \{1, \ldots, n\}$.
  - 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!
Outline

• **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - (We still need to figure out how to do the bucketing)

  **Interlude**: motivation for hash families.

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

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

\[
\begin{array}{c}
\text{n=9 buckets}\\
1 \quad 21 \quad \rightarrow \quad 22 \quad \rightarrow \quad 13 \quad \rightarrow \quad 43 \\
2 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
3 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
\vdots \\
9 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
\text{n=9 buckets}\\
1 \quad 13 \quad \rightarrow \quad 43 \\
2 \quad \rightarrow \quad \rightarrow \\
3 \quad \rightarrow \quad \rightarrow \\
\vdots \\
9 \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

vs.

\[
\begin{array}{c}
\text{n=9 buckets}\\
1 \quad 23 \\
2 \quad \rightarrow \quad \rightarrow \\
3 \quad \rightarrow \quad \rightarrow \\
\vdots \\
9 \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
\text{n=9 buckets}\\
1 \quad 3 \\
2 \quad \rightarrow \quad \rightarrow \\
3 \quad \rightarrow \quad \rightarrow \\
\vdots \\
9 \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
\text{n=9 buckets}\\
1 \quad 93 \\
2 \quad \rightarrow \quad \rightarrow \\
3 \quad \rightarrow \quad \rightarrow \\
\vdots \\
9 \quad \rightarrow \quad \rightarrow \\
\end{array}
\]

Worst-case analysis

• Goal: Design a function $h: U \to \{1, \ldots, n\}$ 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

*Assuming it takes time $O(1)$ to evaluate $h(u)$
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 $\frac{M}{n}$ items hashed to it.
- M is waayyyy bigger then n, so $\frac{M}{n}$ 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 \textbf{random} hash function $h: U \to \{1, \ldots, n\}$.

3. \textbf{HASH IT OUT} #hashpuns

\begin{align*}
\begin{array}{cccc}
13 & 22 & 43 & 92 & 7 \\
\end{array}
\end{align*}

INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92
Example of a random hash function

• $h: U \rightarrow \{1, \ldots, n\}$ 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 $h(1), h(2), \ldots, h(M-1)$. 
Randomness can help!

Intuitively: The bad guy can’t foil a hash function that they don’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...
Intuitive goal

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

- Let $h$ be a random hash function.
- 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 $u_i$,
    \[ E[\text{time of operation}] = O(1) \]

*Assuming $h(u)$ takes $O(1)$ time to compute
Goal:

- Come up with a distribution on hash functions so that:
  - For all $i = 1, \ldots, n$, 
    $$
    \mathbb{E}[\text{number of items in } u_i\text{'s bucket}] \leq 2.
    $$
Aside

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

vs

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

Are these the same?

Think-Pair-Share Terrapins

No! (This was your pre-lecture exercise!)
Aside

- For all $i = 1, \ldots, n$,

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

**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 $E[\text{number of items in 43’s bucket}] = n$.
Goal:

• Come up with a distribution on hash functions so that:
  • For all $i = 1, \ldots, n$,
    \[ E[ \text{number of items in } u_i \text{'s bucket} ] \leq 2. \]

Claim:

• The goal is achieved by a uniformly random hash function.
Proof of Claim

• Let \( h \) be a uniformly random hash function.
• Then for all \( i = 1, \ldots, n \),
  \[ E[ \text{number of items in } u_i\text{'s bucket} ] \leq 2. \]

You will formally verify this in HW2. Intuitively, there are \( n \) possibilities where \( u_j \) can land, and only one of them is \( h(u_i) \).
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 $i \in \{1, \ldots, n\}$,
    $$E[\text{number of items in } u_i \text{'s bucket}] \leq 2.$$  
  • Which implies*:
    • No matter what sequence of operations and items the bad guy chooses,
      $$E[\text{time of INSERT/DELETE/SEARCH}] = O(1)$$

• So our solution is:
  Pick a uniformly random hash function?

*Assuming $h(u)$ takes $O(1)$ time to compute
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 $h(1)$, $h(2)$, ..., $h(n-1)$.
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>ZZZZZZY</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.

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

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.
Example: a smaller hash family

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

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

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

I picked $h_1$

$H = \{h_0, h_1\}$

$\begin{array}{cccc}
19 & 22 & 42 & 92 \\
\end{array}$

INSERT 19, INSERT 22, INSERT 42, INSERT 92, INSERT 0, SEARCH 42, DELETE 92, SEARCH 0, INSERT 92
This is not a very good hash family

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

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

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

I picked $h_0$
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....
Proof of Claim

• \(E\left[\text{number of items in } u_i\text{'s bucket}\right] =\)
  \[= E\left[\sum_{j=1}^{n} 1\{h(u_i) = h(u_j)\}\right]\]
  \[= \sum_{j=1}^{n} P\{h(u_i) = h(u_j)\}\]
  \[= 1 + \sum_{j \neq i} P\{h(u_i) = h(u_j)\}\]
  \[= 1 + \sum_{j \neq i} 1/n\]
  \[= 1 + \frac{n-1}{n} \leq 2.\]

• Let \(h\) be a uniformly random hash function.
• Then for all \(i = 1, \ldots, n,\)
  \(E[\text{number of items in } u_i\text{'s bucket }] \leq 2.\)

All that we needed was that this is \(1/n\)
Universal hash families

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

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

• Earlier analysis shows: if we draw $h$ uniformly at random from a universal hash family $H$, we will have expected time* $O(1)$ INSERT/DELETE/SEARCH!

• And if $H$ is small, we can store a random $h \in H$ efficiently!
The whole scheme will be

Choose $h$ randomly from $H$

We can store $h$ using $\log |H|$ bits.

Probably these buckets will be pretty balanced.
Universal hash families

- 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 \( u_i \neq u_j \),
• **Universal hash family:** if you choose $h$ randomly 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 $u_i \neq u_j$.

Example

• $H =$ 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

- \( h_0 = \text{Most\_significant\_digit} \)
- \( h_1 = \text{Least\_significant\_digit} \)
- \( H = \{ h_0, h_1 \} \)

**Universal hash family:** if you choose \( h \) randomly from \( H \),

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

Prove that this choice of \( H \) is NOT a universal hash family!

2 minutes think
1 minute pair and share
Non-example

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

NOT a universal hash family:

$$P_{h \in H}\{h(101) = h(111)\} = 1 > \frac{1}{10}$$
A small universal hash family??

• Here’s one:
  • Pick a prime $p \geq M$. (And not much bigger than $M$)
  • Define
    \[ f_{a,b}(x) = ax + b \mod p \]
    \[ h_{a,b}(x) = f_{a,b}(x) \mod n \]
  • Define:
    \[ H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \} \]
A small universal hash family??

• Here’s one:
  • Pick a prime $p \geq M$. (And not much bigger than $M$)
  • Define
    \[ f_{a,b}(x) = ax + b \mod p \]
    \[ h_{a,b}(x) = f_{a,b}(x) \mod n \]
  • Define:
    \[ H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \} \]

• Claims:
  H is a universal hash family.
  A random $h \in H$ takes $O(\log M)$ bits to store.

See CLRS (Thm 11.5) if you are curious, but you don’t need to know why this is true for this class.
A random \( h \in H \) takes \( O(\log M) \) bits to store (And more!)

\[
H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \}
\]
\[
|H| = p \cdot (p - 1) = O(M^2)
\]

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

• Also, given \( a \) and \( b \), \( h \) is fast to evaluate!
  • It takes time \( O(1) \) to compute \( h(x) \).

• Compare: direct addressing was \( M \) bits!
  • Example: If \( M = 128^{280} \), \( \log(M) = 1960 \).
A small universal hash family??

• Here’s one:
  • Pick a prime $p \geq M$. (And not much bigger than $M$)
  • Define
    $$ f_{a,b}(x) = ax + b \mod p $$
    $$ h_{a,b}(x) = f_{a,b}(x) \mod n $$
  • Define:
    $$ H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \} $$

• Claims:
  H is a universal hash family.
  A random $h \in H$ takes $O(\log M)$ bits to store.
So the whole scheme will be

\[ H = \{ h_{a,b}(x) : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \} \]

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 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

   ![Diagram showing hash function application and operations sequence]

   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.

All of the hash functions

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

Data Structure: RBTrees and Hash Tables

Now we can use these going forward!
Next Time

• Embedded EthiCS and Review Session!

Before Next Time

• Come with questions!