Hash Tables
Homework #2 is due September 25th @ 11:59pm

Project #1 is due October 2nd @ 11:59pm
→ Q&A Session: Thursday September 22nd @ 8:00pm
→ Special Office Hours: Saturday October 1st @ 3pm-5pm
UPCOMING DATABASE TALKS

**Rockset**  
→ Monday Sept 26 @4:30pm

**Odyssey Proxy**  
→ Monday Oct 3rd @4:30pm

**Litestream**  
→ Monday Oct 10th @4:30pm
We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:
→ Hash Tables
→ Trees
DATA STRUCTURES

- Internal Meta-data
- Core Data Storage
- Temporary Data Structures
- Table Indexes
DESIGN DECISIONS

Data Organization
→ How we layout data structure in memory/pages and what information to store to support efficient access.

Concurrency
→ How to enable multiple threads to access the data structure at the same time without causing problems.
A **hash table** implements an unordered associative array that maps keys to values.

It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.

Space Complexity: \( O(n) \)

Time Complexity:

\(-\) Average: \( O(1) \) ← *Databases need to care about constants!*
\(-\) Worst: \( O(n) \)
STATIC HASH TABLE

Allocate a giant array that has one slot for every element you need to store.

To find an entry, mod the key by the number of elements to find the offset in the array.

\[
\text{hash(key)} \% N
\]

<table>
<thead>
<tr>
<th>0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>Z</td>
</tr>
</tbody>
</table>
Allocate a giant array that has one slot for every element you need to store.

To find an entry, mod the key by the number of elements to find the offset in the array.

\[ \text{hash(key)} \% N \]
Assumption #1: Number of elements is known ahead of time and fixed.

Assumption #2: Each key is unique.

Assumption #3: Perfect hash function.

→ If $key_1 \neq key_2$, then $hash(key_1) \neq hash(key_2)$
## HASH TABLE

### Design Decision #1: Hash Function
- How to map a large key space into a smaller domain.
- Trade-off between being fast vs. collision rate.

### Design Decision #2: Hashing Scheme
- How to handle key collisions after hashing.
- Trade-off between allocating a large hash table vs. additional instructions to get/put keys.
TODAY'S AGENDA

Hash Functions
Static Hashing Schemes
Dynamic Hashing Schemes
HASH FUNCTIONS

For any input key, return an integer representation of that key.

We do not want to use a cryptographic hash function for DBMS hash tables (e.g., SHA-2).

We want something that is fast and has a low collision rate.
HASH FUNCTIONS

CRC-64 (1975)
→ Used in networking for error detection.

MurmurHash (2008)
→ Designed as a fast, general-purpose hash function.

Google CityHash (2011)
→ Designed to be faster for short keys (<64 bytes).

Facebook XXHash (2012)
→ From the creator of zstd compression.

Google FarmHash (2014)
→ Newer version of CityHash with better collision rates.
HASH FUNCTION BENCHMARK

Intel Core i7-8700K @ 3.70GHz

Throughput (MB/sec) vs. Key Size (bytes)

- std::hash
- MurmurHash3
- CityHash
- FarmHash
- XXHash3

Source: Fredrik Widlund
Throughput (MB/sec) vs Key Size (bytes)

- **crc64**
- **std::hash**
- **MurmurHash3**
- **CityHash**
- **FarmHash**
- **XXHash3**

**Hash Function Benchmark**

**Source:** Fredrik Widlund

**Intel Core i7-8700K @ 3.70GHz**

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>MB/sec</th>
<th>cycl/hash</th>
<th>cycl/map</th>
<th>size</th>
<th>Quality problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>donothing32</td>
<td>1531.6474.36</td>
<td>6.50</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>donothing64</td>
<td>1533.0619.19</td>
<td>6.50</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>donothing128</td>
<td>1527.8933.09</td>
<td>6.03</td>
<td>-</td>
<td>13</td>
<td>bad seed 0, test NOP</td>
</tr>
<tr>
<td>NOP_DAA1_read64</td>
<td>2846.750</td>
<td>18.48</td>
<td>-</td>
<td>47</td>
<td>test NOP</td>
</tr>
<tr>
<td>BadHash</td>
<td>524.81</td>
<td>98.26</td>
<td>-</td>
<td>47</td>
<td>bad seed 0, test FAIL</td>
</tr>
<tr>
<td>sumhash</td>
<td>7159.08</td>
<td>27.12</td>
<td>-</td>
<td>363</td>
<td>bad seed 0, test FAIL</td>
</tr>
<tr>
<td>sumhash32</td>
<td>22556.18</td>
<td>22.98</td>
<td>-</td>
<td>863</td>
<td>UB, test FAIL</td>
</tr>
<tr>
<td>multiply_shift</td>
<td>5418.36</td>
<td>28.69</td>
<td>157.11</td>
<td>3</td>
<td>346</td>
</tr>
<tr>
<td>pae_multiply_shift</td>
<td>3716.95</td>
<td>40.22</td>
<td>186.34</td>
<td>3</td>
<td>609</td>
</tr>
</tbody>
</table>
STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing
Approach #2: Robin Hood Hashing
Approach #3: Cuckoo Hashing
LINEAR PROBE HASHING

Single giant table of slots.

Resolve collisions by linearly searching for the next free slot in the table.
   → To determine whether an element is present, hash to a location in the index and scan for it.
   → Must store the key in the index to know when to stop scanning.
   → Insertions and deletions are generalizations of lookups.
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

A | val

<key> | <value>
LINEAR PROBE HASHING

\[ \text{hash(key)} \mod N \]

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

B | val
A | val
C | val
LINEAR PROBE HASHING

$\text{hash(key)} \% N$

A
B
C
D
E
F

B | val
A | val
C | val
D | val
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>val</td>
<td></td>
<td>val</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{hash(key)} \% N \] is used to determine the initial position for storing the key-value pair in the hash table. If the position is occupied, linear probing is used to find the next available slot.
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

\[ \begin{array}{c|c}
A & \text{val} \\
B & \text{val} \\
C & \text{val} \\
D & \text{val} \\
E & \text{val} \\
F & \text{val} \\
\end{array} \]
LINEAR PROBE HASHING - DELETES

\[ \text{hash(key)} \% N \]

Delete

\[ \begin{array}{c|c|c|c}
A & B & C \\
B & A & D \\
C & & E \\
D & & F \\
E & & \\
F & & \\
\end{array} \]
LINEAR PROBE HASHING – DELETES

\( \text{hash(key)} \% N \)

A
B
C
D
E
F

Get

\( B | \text{val} \)
\( A | \text{val} \)
\( D | \text{val} \)
\( E | \text{val} \)
\( F | \text{val} \)
**LINEAR PROBE HASHING – DELETES**

**Approach #1: Movement**

→ Rehash keys until you find the first empty slot.
→ Nobody actually does this.
LINEAR PROBE HASHING - DELETES

hash(key) % N

A
B
C
D
E
F

Delete

Approach #2: Tombstone

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ You can reuse the slot for new keys.
→ May need periodic garbage collection.
**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ You can reuse the slot for new keys.
→ May need periodic garbage collection.
**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ You can reuse the slot for new keys.
→ May need periodic garbage collection.
**LINEAR PROBE HASHING - DELETES**

**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.

→ You can reuse the slot for new keys.

→ May need periodic garbage collection.

\[
\text{hash(key)} \% N
\]

| A | val |
| B | val |
| C |    |
| D | val |
| E | val |
| F | val |
| G | val |

Put G
NON-UNIQUE KEYS

Choice #1: Separate Linked List
→ Store values in separate storage area for each key.

Choice #2: Redundant Keys
→ Store duplicate keys entries together in the hash table.
→ This is easier to implement so this is what most systems do.
ROBIN HOOD HASHING

Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.
→ Each key tracks the number of positions they are from where its optimal position in the table.
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.
ROBIN HOOD HASHING

hash(key) \% N

# of "Jumps" From First Position
ROBIN HOOD HASHING

$\text{hash(key)} \mod N$

\[
\begin{array}{c|c}
\text{A} & \text{val [0]} \\
\text{B} & \\
\text{C} & \\
\text{D} & \\
\text{E} & \\
\text{F} & \\
\end{array}
\]
**ROBIN HOOD HASHING**

hash(key) \% N

A
B
C
D
E
F

B | val [0]
A | val [0]
C | val [1]

A[0] == C[0]
ROBIN HOOD HASHING

\[
\text{hash(key) \% N}
\]

\[
\begin{align*}
A & : \text{val [0]} \\
B & : \text{val [0]} \\
C & : \text{val [1]} \\
D & : \text{val [1]} \\
E & : \\
F & : \\
\end{align*}
\]

\[C[1] > D[0]\]
ROBIN HOOD HASHING

$hash(key) \% N$

A | val [0]
B | val [0]
C | val [1]
D | val [1]
E
F

- $A[0] == E[0]$
- $C[1] == E[1]$
ROBIN HOOD HASHING

hash(key) % N

<table>
<thead>
<tr>
<th>A</th>
<th>val [0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>val [1]</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>val [2]</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

Let's consider the hashing function $\text{hash(key)} \mod N$ and some examples.

- For key A, the hash value is $A \mod N = 0$.
- For key B, the hash value is $B \mod N = 1$.
- For key C, the hash value is $C \mod N = 2$.
- For key D, the hash value is $D \mod N = 2$.
- For key E, the hash value is $E \mod N = 2$.
- For key F, the hash value is $F \mod N = 2$.

In the hash table, we have the following values:

- B\[0\]
- A\[0\]
- C\[1\]
- E\[2\]
- D\[2\]

The diagram shows the placement of keys in the hash table and the corresponding values.

- $A[0] == E[0]$
- $C[1] == E[1]$
Robin Hood Hashing

$hash(key) \% N$

A
B
C
D
E
F

$D[2] > F[0]$
CUCKOO HASHING

Use multiple hash tables with different hash function seeds.
→ On insert, check every table and pick anyone that has a free slot.
→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

Look-ups and deletions are always $O(1)$ because only one location per hash table is checked.

Best open-source implementation is from CMU.
Cuckoo Hashing

Hash Table #1

Hash Table #2

Put A

$\text{hash}_1(A)$  $\text{hash}_2(A)$
**CUCKOO HASHING**

**Hash Table #1**

\[
A | val
\]

**Hash Table #2**

\[
\text{hash}_1(A) \quad \text{hash}_2(A)
\]

Put A

---

15-445/645 (Fall 2022)
Cuckoo Hashing

Hash Table #1

Put A

\[\text{hash}_1(A) \quad \text{hash}_2(A)\]

Put B

\[\text{hash}_1(B) \quad \text{hash}_2(B)\]

Hash Table #2

\[A | \text{val}\]
Cuckoo Hashing

**Hash Table #1**

- Put A
  - hash\_1(A)
  - hash\_2(A)

- Put B
  - hash\_1(B)
  - hash\_2(B)

**Hash Table #2**

- B | val

---

15-445/645 (Fall 2022)
CUCKOO HASHING

Hash Table #1

Put A
\[ \text{hash}_1(A) \quad \text{hash}_2(A) \]

Put B
\[ \text{hash}_1(B) \quad \text{hash}_2(B) \]

Put C
\[ \text{hash}_1(C) \quad \text{hash}_2(C) \]

Hash Table #2

B | val
**CUCKOO HASHING**

**Hash Table #1**

- Put A
  - $hash_1(A)$
  - $hash_2(A)$

- Put B
  - $hash_1(B)$
  - $hash_2(B)$

- Put C
  - $hash_1(C)$
  - $hash_2(C)$

**Hash Table #2**

- Put B
  - $B | val$

- ...
CUCKOO HASHING

Hash Table #1

Put A
\[ \text{hash}_1(A) \quad \text{hash}_2(A) \]

Put B
\[ \text{hash}_1(B) \quad \text{hash}_2(B) \]

Put C
\[ \text{hash}_1(C) \quad \text{hash}_2(C) \]

Hash Table #2

\[ C | \text{val} \]
CUCKOO HASHING

Hash Table #1

Put A
hash₁(A) hash₂(A)

Put B
hash₁(B) hash₂(B)

Put C
hash₁(C) hash₂(C)

hash₁(B)

Hash Table #2

C | val

...
CUCKOO HASHING

Hash Table #1

Put A
\( \text{hash}_1(A) \quad \text{hash}_2(A) \)

Put B
\( \text{hash}_1(B) \quad \text{hash}_2(B) \)

Put C
\( \text{hash}_1(C) \quad \text{hash}_2(C) \)
\( \text{hash}_1(B) \)

Hash Table #2

C
\( \text{val} \)

...
CUCKOO HASHING

Hash Table #1

Put A
hash\(_1\)(A)  hash\(_2\)(A)

Put B
hash\(_1\)(B)  hash\(_2\)(B)

Put C
hash\(_1\)(C)  hash\(_2\)(C)
hash\(_1\)(B)  hash\(_2\)(A)

Hash Table #2

C | val

A | val
CUCKOO HASHING

Hash Table #1

B | val

Hash Table #2

C | val

A | val

Put A
hash_1(A) hash_2(A)

Put B
hash_1(B) hash_2(B)

Put C
hash_1(C) hash_2(C)

hash_1(B) hash_2(A)

Get B
hash_1(B) hash_2(B)
**OBSERVATION**

The previous hash tables require the DBMS to know the number of elements it wants to store. → Otherwise, it must rebuild the table if it needs to grow/shrink in size.

Dynamic hash tables resize themselves on demand.  
→ Chained Hashing  
→ Extendible Hashing  
→ Linear Hashing
Maintain a linked list of buckets for each slot in the hash table.

Resolve collisions by placing all elements with the same hash key into the same bucket.
→ To determine whether an element is present, hash to its bucket and scan for it.
→ Insertions and deletions are generalizations of lookups.
CHAINED HASHING

$hash(key) \% N$

Bucket Pointers

Buckets

A
B
C
D
E
F

A | val
CHAINED HASHING

hash(key) % N

A
B
C
D
E
F

Bucket
Pointers

B | val
A | val

Buckets
**CHAINED HASHING**

hash(key) % N

A  B  C  D  E  F

Bucket Pointers

B | val
A | val
C | val

Buckets
**CHAINED HASHING**

$\text{hash(key)} \% N$

Bucket Pointers

A
B
C
D
E
F

B | val
A | val
C | val
D | val
CHAINED HASHING

\[ hash(key) \mod N \]

Bucket Pointers

A
B
C
D
E
F

B | val

A | val

C | val

D | val

E | val
CHAINED HASHING

hash(key) \% N

A
B
C
D
E
F

Bucket Pointers

B | val
A | val
C | val
F | val

D | val
E | val
EXTENDIBLE HASHING

Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.
EXTENDIBLE HASHING

**global**

2

00...
01...
10...
11...

**local**

1
0010...
0110...

2
10101...
10011...

2
11010...
11010...

**local**

2
EXTENDIBLE HASHING

Get A

\[ \text{hash}(A) = 01110... \]
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]
**EXTENDIBLE HASHING**

Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

Diagram:
- Global key: 2
- Local key 1: 00010...
  - 01110...
- Local key 2: 10101...
  - 10011...
  - 10111...
- Local key 3: 11010...

EXTENDIBLE HASHING

Get $A$
$\text{hash}(A) = 01110...$

Put $B$
$\text{hash}(B) = 10111...$

Put $C$
$\text{hash}(C) = 10100...$
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]

Put C
\[ \text{hash}(C) = 10100... \]
EXTENDIBLE HASHING

**Get A**

\[ hash(A) = 01110... \]

**Put B**

\[ hash(B) = 10111... \]

**Put C**

\[ hash(C) = 10100... \]
EXTENDIBLE HASHING

- Get A
  \[ \text{hash}(A) = 01110... \]
- Put B
  \[ \text{hash}(B) = 10111... \]
- Put C
  \[ \text{hash}(C) = 10100... \]
EXTENDIBLE HASHING

Get A
\text{hash}(A) = 01110\ldots

Put B
\text{hash}(B) = 10111\ldots

Put C
\text{hash}(C) = 10100\ldots
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]

Put C
\[ \text{hash}(C) = 10100... \]
LINEAR HASHING

The hash table maintains a pointer that tracks the next bucket to split.
→ When any bucket overflows, split the bucket at the pointer location.

Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:
→ Space Utilization
→ Average Length of Overflow Chains
LINEAR HASHING

Get 6

\[
hash_1(6) = 6 \mod 4 = 2
\]

Split Pointer

\[
hash_1(key) = key \mod n
\]
LINEAR HASHING

\[ \text{hash}_1(\text{key}) = \text{key} \mod n \]

Get 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Split Pointer

0
1
2
3
**LINEAR HASHING**

- **Split Pointer**
- **Get 6**
  \[ \text{hash}_1(6) = 6 \mod 4 = 2 \]
- **Put 17**
  \[ \text{hash}_1(17) = 17 \mod 4 = 1 \]
- **Overflow!**

**hash_1(key) = key \mod n**
LINEAR HASHING

Get 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Overflow!

\[ \text{hash}_1(\text{key}) = \text{key} \mod n \]

Split Pointer

0 1 2 3
**LINEAR HASHING**

Split Pointer

Get 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Overflow!

**hash_1(key) = key % n**

**hash_2(key) = key % 2n**
LINEAR HASHING

Split Pointer

Get 6
\[\text{hash}_1(6) = 6 \mod 4 = 2\]

Put 17
\[\text{hash}_1(17) = 17 \mod 4 = 1\]

\[\text{hash}_1(key) = key \mod n\]
\[\text{hash}_2(key) = key \mod 2n\]
**Linear Hashing**

- **Split Pointer**
- **Get 6**
  \[\text{hash}_1(6) = 6 \mod 4 = 2\]
- **Put 17**
  \[\text{hash}_1(17) = 17 \mod 4 = 1\]
- **Get 20**
  \[\text{hash}_1(20) = 20 \mod 4 = 0\]

**Hash Functions**

- \[\text{hash}_1(key) = key \mod n\]
- \[\text{hash}_2(key) = key \mod 2n\]
**LINEAR HASHING**

\[
\text{hash}_1(key) = key \mod n \\
\text{hash}_2(key) = key \mod 2n
\]

- **Get 6**
  \[\text{hash}_1(6) = 6 \mod 4 = 2\]

- **Put 17**
  \[\text{hash}_1(17) = 17 \mod 4 = 1\]

- **Get 20**
  \[\text{hash}_1(20) = 20 \mod 4 = 0\]
  \[\text{hash}_2(20) = 20 \mod 8 = 4\]
LINEAR HASHING

\[ \text{hash}_1(key) = key \mod n \]
\[ \text{hash}_2(key) = key \mod 2n \]

Get 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Get 20
\[ \text{hash}_1(20) = 20 \mod 4 = 0 \]
\[ \text{hash}_2(20) = 20 \mod 8 = 4 \]

Get 9
\[ \text{hash}_1(9) = 9 \mod 4 = 1 \]
LINEAR HASHING

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.
→ When the pointer reaches the last slot, delete the first hash function and move back to beginning.
**LINEAR HASHING - DELETES**

Split Pointer

1. $\text{hash}_1(key) = key \mod n$
2. $\text{hash}_2(key) = key \mod 2n$
LINEAR HASHING – DELETES

Split Pointer

Delete 20

$hash_1(20) = 20 \% 4 = 0$

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$
Linear Hashing - Deletes

$\text{Split Pointer}$

Delete 20

$\text{hash}_1(20) = 20 \mod 4 = 0$

$\text{hash}_2(20) = 20 \mod 8 = 4$

$\text{Split Pointer}$

$\text{hash}_1(\text{key}) = \text{key} \mod n$

$\text{hash}_2(\text{key}) = \text{key} \mod 2n$
LINEAR HASHING - DELETES

\[ \text{hash}_1(key) = key \mod n \]
\[ \text{hash}_2(key) = key \mod 2n \]

Delete 20
\[ \text{hash}_1(20) = 20 \mod 4 = 0 \]
\[ \text{hash}_2(20) = 20 \mod 8 = 4 \]
LINEAR HASHING - DELETES

\[ \text{hash}_1(key) = key \mod n \]
\[ \text{hash}_2(key) = key \mod 2n \]

Delete 20
\[ \text{hash}_1(20) = 20 \mod 4 = 0 \]
\[ \text{hash}_2(20) = 20 \mod 8 = 4 \]
**LINEAR HASHING - DELETES**

- **Split Pointer**

$$\text{hash}_1(key) = key \% n$$

Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$
$$\text{hash}_2(20) = 20 \% 8 = 4$$
LINEAR HASHING - DELETES

\[ h_1(key) = key \mod n \]

Delete 20
\[ h_1(20) = 20 \mod 4 = 0 \]
\[ h_2(20) = 20 \mod 8 = 4 \]

Put 21
\[ h_1(21) = 21 \mod 4 = 1 \]

Overflow!
CONCLUSION

Fast data structures that support $O(1)$ look-ups that are used all throughout DBMS internals.

→ Trade-off between speed and flexibility.

Hash tables are usually not what you want to use for a table index...
NEXT CLASS

B+Trees
→ aka "The Greatest Data Structure of All Time"