06 Hash Tables
**ADMINISTRIVIA**

**Project #1** is due Sunday Sept 27th

Q&A Session about the project on **Monday Sept 21st @ 8:00pm ET**.
→ In-Person: GHC 4401
→ [https://cmu.zoom.us/j/98100285498?pwd=a011L0E2eWFwTndKMG9KNVhzb2tDdz09](https://cmu.zoom.us/j/98100285498?pwd=a011L0E2eWFwTndKMG9KNVhzb2tDdz09)
UPCOMING DATABASE TALKS

Snowflake Query Optimizer
→ Today @ 5pm ET

CockroachDB Query Optimizer
→ Monday Sept 28ᵗʰ @ 5pm ET

Apache Arrow
→ Monday Oct 5ᵗʰ @ 5pm
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 the array for a given key, from which the desired value can be found.

Space Complexity: $O(n)$
Operation Complexity:
→ Average: $O(1)$ ← **Money cares about constants!**
→ Worst: $O(n)$
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.
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.
ASSUMPTIONS

You know the number of elements ahead of time.

Each key is unique.

Perfect hash function.
→ If key1≠key2, then hash(key1)≠hash(key2)
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 find/insert 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.

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 to 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)

Source: Fredrik Widlund
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)

A
B
C
D
E
F

A | val

<key> | <value>
LINEAR PROBE HASHING

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

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

hash(key)

A
B
C
D
E
F
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

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

hash(key)

A
B
C
D
E
F

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

### hash(key)

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

The diagram shows a hash table with items A, B, C, D, E, and F. The hash function maps each key to its corresponding value. For example, key A maps to value B, and key F maps to value F. The linear probing technique is used to handle collisions by sequentially searching for an available slot.
LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
D
E
F

Delete

B | val
A | val
C | val
D | val
E | val
F | val
LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
D
E
F

Delete

B | val
A | val

D | val
E | val
F | val
LINEAR PROBE HASHING – DELETES

Find

hash(key)

A
B
C
D
E
F

B \mid \text{val}
A \mid \text{val}
\text{C}
D \mid \text{val}
E \mid \text{val}
F \mid \text{val}
### Linear Probe Hashing – Deletes

#### Approach #1: Tombstone

<table>
<thead>
<tr>
<th>hash(key)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

**Find →**

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

- Tombstones are placeholders for deleted items.
- Hash function determines the initial position.
- Linear probing is used to find the next available slot.
- When an item is deleted, it is marked as a tombstone.
LINEAR PROBE HASHING – DELETES

Approach #1: Tombstone

Find

hash(key)

A
B
C
D
E
F

B | val
A | val
C
D | val
E | val
F | val
LINEAR PROBE HASHING – DELETES

Approach #1: Tombstone

Find

Approach #2: Movement

<table>
<thead>
<tr>
<th>hash(key)</th>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>
LINEAR PROBE HASHING – DELETES

**hash(key)**

A
B
C
D
E
F

---

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

Approach #1: Tombstone

Approach #2: Movement
LINEAR PROBE HASHING – DELETES

Approach #1: Tombstone
Approach #2: Movement

Find

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

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

Approach #1: Tombstone

Approach #2: Movement

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

<table>
<thead>
<tr>
<th>hash(key)</th>
<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></td>
<td>(B</td>
<td>\text{val})</td>
<td></td>
<td>A</td>
<td>(D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(E</td>
<td>\text{val})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Find

15-445/645 (Fall 2020)
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.
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)

A
B
C
D
E
F

# of "Jumps" From First Position

val [0]
ROBIN HOOD HASHING

hash(key)

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

A | val [0]

B | val [0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

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

```plaintext
hash(key)
```

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

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

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

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

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

#### hash(key)

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

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

hash(key)

A
B
C
D
E
F

B | val [0]
A | val [0]
C | val [1]
E | val [2]
D | val [2]

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

**hash(key)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val [0]</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
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

Insert A

$hash_1(A)$

$hash_2(A)$

Hash Table #2
Cuckoo Hashing

Hash Table #1

| A | val |

Insert A

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

Hash Table #2
CUCKOO HASHING

Hash Table #1

A | val

Hash Table #2

Insert A
hash₁(A)  hash₂(A)

Insert B
hash₁(B)  hash₂(B)
**Cuckoo Hashing**

**Hash Table #1**

- Insert A
  - $\text{hash}_1(A)$
  - $\text{hash}_2(A)$

- Insert B
  - $\text{hash}_1(B)$
  - $\text{hash}_2(B)$

**Hash Table #2**

- Insert B
  - $\text{hash}_1(B)$
  - $\text{hash}_2(B)$
CUCKOO HASHING

Hash Table #1

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

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

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

Hash Table #2

A | val

B | val
CUCKOO HASHING

Hash Table #1

Insert A
hash₁(A) hash₂(A)

Insert B
hash₁(B) hash₂(B)

Insert C
hash₁(C) hash₂(C)

Hash Table #2

C | val
CUCKOO HASHING

Hash Table #1

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

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

Insert 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

Insert A
\( hash_1(A) \quad hash_2(A) \)

Insert B
\( hash_1(B) \quad hash_2(B) \)

Insert C
\( hash_1(C) \quad hash_2(C) \)
\( hash_1(B) \quad hash_2(A) \)

Hash Table #2

C | val

A | val
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
CHAINED 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)

A
B
C
D
E
F

Bucket Pointers

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

Bucket Pointers

A | val

Buckets
**CHAINED HASHING**

- **hash(key)**
- **Bucket Pointers**
- **Buckets**

- A
- B
- C
- D
- E
- F

Example:
- Hash(key)
- Bucket Pointers
- Buckets

- A | val
- B | val
CHAINED HASHING

$\text{hash(key)}$

A
B
C
D
E
F

Bucket Pointers

B | val
A | val
C | val

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

Bucket Pointers

B | val

A | val

C | val

Buckets
CHAINED HASHING

hash(key)

A
B
C
D
E
F

Bucket Pointers

B | val
A | val
C | val
D | val

E

F
CHAINED HASHING

Hash function: $\text{hash(key)}$

Values:
- A
- B
- C
- D
- E
- F

Bucket Pointers

- A
- B
- C
- D
- E

D | val
E | val

A | val
C | val
B | val
CHAINED HASHING

hash(key)

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.

Reshuffling 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...

- **Local**: 2
  - 10101...
  - 10011...

- **Local**: 2
  - 11010...
EXTENDIBLE HASHING

Find A
hash(A) = \textbf{01110...}
EXTENDIBLE HASHING

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Illustration of extendible hashing with global keys and their hash values for finding and inserting data:

- **Find A**: hash(A) = 01110...
  - Insert B: hash(B) = 10111...
  - Insert C: hash(C) = 10100...

Each key is hashed to a specific segment, and the hashing process is shown for finding and inserting elements.
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

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

Split Pointer

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

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

Split Pointer

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

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

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

- **Split Pointer**
- **Find 6**
  \[ \text{hash}_1(6) = 6 \% 4 = 2 \]
- **Insert 17**
  \[ \text{hash}_1(17) = 17 \% 4 = 1 \]

**hash\_1(key) = key \% n**

Insertion and lookup operations in a linear hashing scheme with an initial hash function \( \text{hash}_1 \). The diagram shows the process of finding 6 and inserting 17, with an overflow occurring when attempting to insert 17 into the slot for key 5.
**LINEAR HASHING**

Find 6

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

Insert 17

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

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

**Split Pointer**

- Hash function: \[ \text{hash}_1(\text{key}) = \text{key} \mod n \]
- Find 6: \[ \text{hash}_1(6) = 6 \mod 4 = 2 \]
- Insert 17: \[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Overflow!
**LINEAR HASHING**

**Split Pointer**

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

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

---

**Find 6**

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

**Insert 17**

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

Overflow!
**LINEAR HASHING**

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

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

**Find 6**
\[ \text{hash}_1(6) = 6 \% 4 = 2 \]

**Insert 17**
\[ \text{hash}_1(17) = 17 \% 4 = 1 \]
LINEAR HASHING

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

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

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

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

Find 20
\[ \text{hash}_1(20) = 20 \mod 4 = 0 \]
LINEAR HASHING

hash_1(key) = key % n
hash_2(key) = key % 2n

Find 6
hash_1(6) = 6 % 4 = 2

Insert 17
hash_1(17) = 17 % 4 = 1

Find 20
hash_1(20) = 20 % 4 = 0
hash_2(20) = 20 % 8 = 4
**LINEAR HASHING**

- **Split Pointer**
- **hash₁(key)** = key % n
- **hash₂(key)** = key % 2n

Find 6
\[hash₁(6) = 6 \mod 4 = 2\]

Insert 17
\[hash₁(17) = 17 \mod 4 = 1\]

Find 20
\[hash₁(20) = 20 \mod 4 = 0\]
\[hash₂(20) = 20 \mod 8 = 4\]

Find 9
\[hash₁(9) = 9 \mod 4 = 1\]
**LINEAR HASHING**

- **Split Pointer**

- **Hash Function**
  - $hash_1(key) = key \% n$
  - $hash_2(key) = key \% 2n$

- **Example Calculations**
  - $hash_1(6) = 6 \% 4 = 2$
  - Insert 17
    - $hash_1(17) = 17 \% 4 = 1$
    - $hash_2(20) = 20 \% 8 = 4$
  - Find 20
  - $hash_1(9) = 9 \% 4 = 1$
  - Find 9

- **Example Insertion**
  - Insert 17
    - Hash for 17: $17 \% 4 = 1$
    - Insert into slot 1
  - Find 20
    - Hash for 20: $20 \% 4 = 0$
    - Hash for 20 (second level): $20 \% 8 = 4$
    - Insert into slot 4
  - Find 6
    - Hash for 6: $6 \% 4 = 2$
    - Insert into slot 2

- **Example Findings**
  - Find 6
  - Insert 17
  - Find 20
  - Find 9
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.

The pointer can also move backwards when buckets are empty.
LINEAR HASHING – DELETES

Split Pointer

hash_1(key) = key % n
hash_2(key) = key % 2n

Delete 20
hash_1(20) = 20 % 4 = 0
LINEAR HASHING – DELETES

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

$\text{hash}_2(key) = \text{key} \mod 2n$

Delete 20

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

$\text{hash}_2(20) = 20 \mod 8 = 4$
LINEAR HASHING – DELETES

Split Pointer

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

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

**Split Pointer**

- `hash_1(key) = key % n`
- `hash_2(key) = key % 2n`

**Delete 20**
- `hash_1(20) = 20 % 4 = 0`
- `hash_2(20) = 20 % 8 = 4`
**LINEAR HASHING – DELETES**

**Split Pointer**

- **Delete 20**
  - \( hash_1(20) = 20 \mod 4 = 0 \)
  - \( hash_2(20) = 20 \mod 8 = 4 \)

- **Insert 21**
  - \( hash_1(21) = 21 \mod 4 = 1 \)

**hash_1(key) = key \mod n**
CONCLUSION

Fast data structures that support $O(1)$ look-ups that are used all throughout the 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!"