

06 |

# Hash Tables



Intro to Database Systems  
15-445/15-645  
Fall 2019

AP

Andy Pavlo  
Computer Science  
Carnegie Mellon University

# ADMINISTRIVIA

---

**Project #1** is due Fri Sept 27<sup>th</sup> @ 11:59pm

**Homework #2** is due Mon Sept 30<sup>th</sup> @ 11:59pm



# COURSE STATUS

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

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

# 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.

# HASH TABLES

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.

We are able to map arbitrary keys to arbitrary values

Space Complexity: **O(n)**

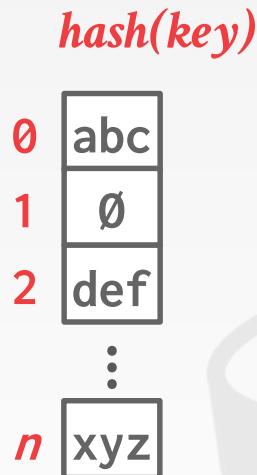
Operation Complexity:

- Average: **O(1)** ← *Money cares about constants!*
- Worst: **O(n)**

# STATIC HASH TABLE

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

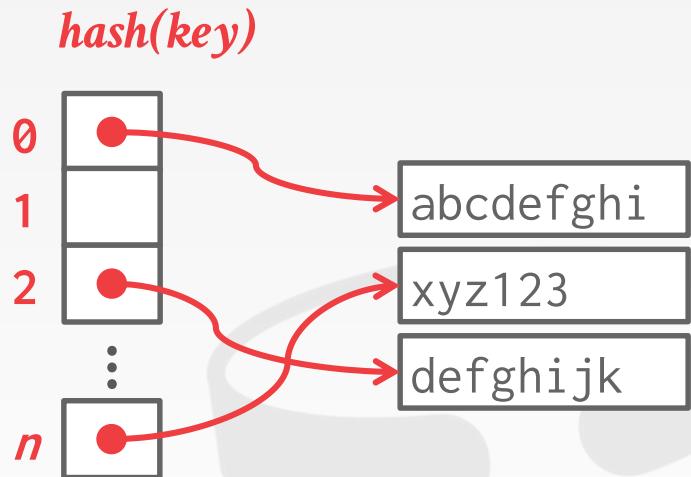
To find an entry, mod the key by the number of elements to find the offset in the array.



# 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.



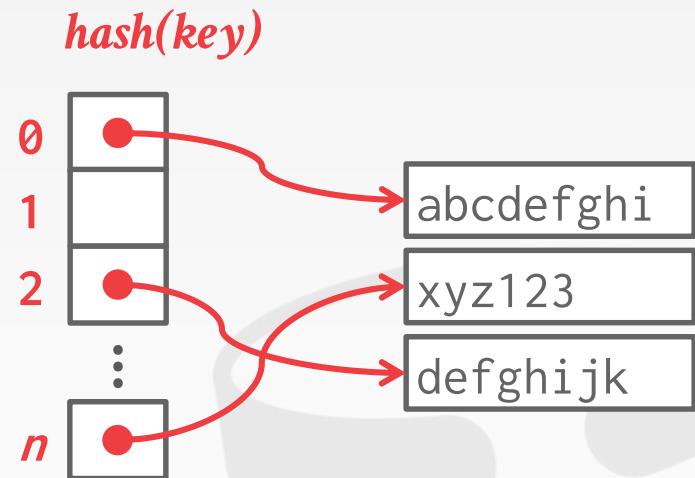
# ASSUMPTIONS

You know the number of elements ahead of time.

Each key is unique.

Perfect hash function.

→ If  $\text{key}_1 \neq \text{key}_2$ , then  
 $\text{hash}(\text{key}_1) \neq \text{hash}(\text{key}_2)$



# HASH TABLE

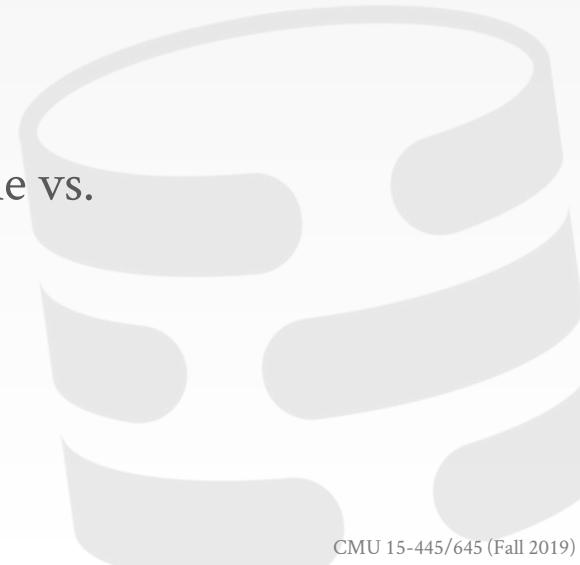
---

## 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 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

---

Reasonable collision rate, but super slow

## 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)

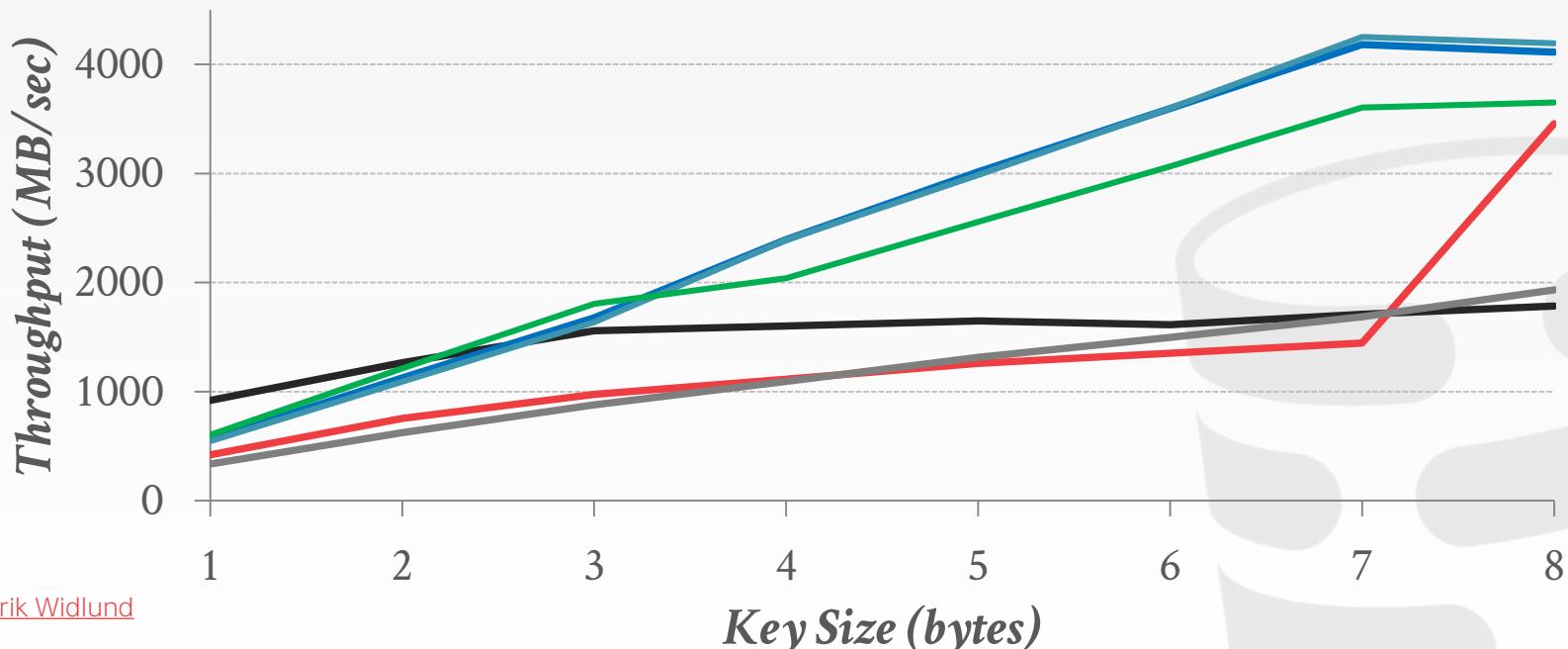
- Newer version of CityHash with better collision rates.



# HASH FUNCTION BENCHMARK

*Intel Core i7-8700K @ 3.70GHz*

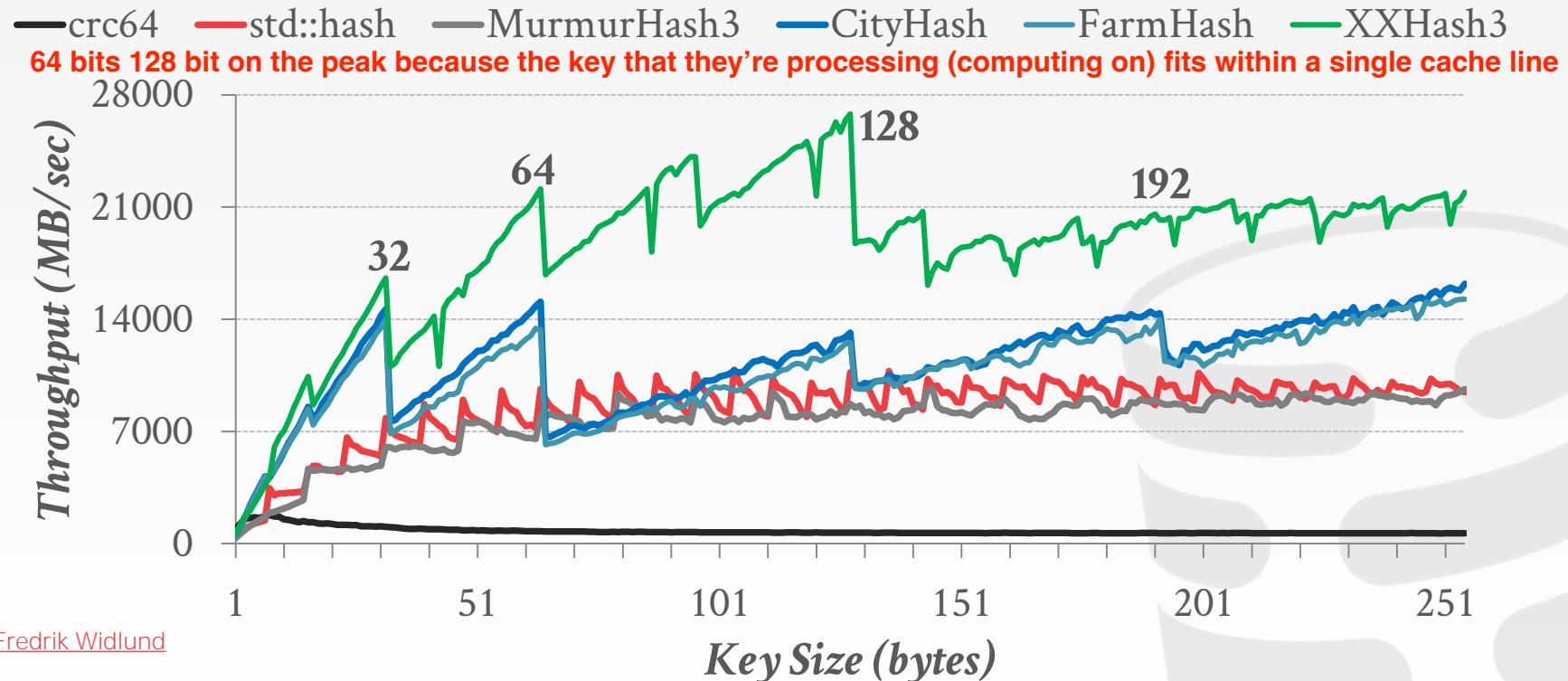
—crc64    —std::hash    —MurmurHash3    —CityHash    —FarmHash    —XXHash3



Source: [Fredrik Widlund](#)

# HASH FUNCTION BENCHMARK

*Intel Core i7-8700K @ 3.70GHz*



Source: [Fredrik Widlund](#)

# STATIC HASHING SCHEMES

---

**Approach #1: Linear Probe Hashing**

**Approach #2: Robin Hood Hashing**

**Approach #3: Cuckoo Hashing**



# LINEAR PROBE HASHING

If we hash into a slot and we find something that's already there, I'm trying to insert something there.  
We just keep scanning down to the next position and keep going until we find the first open slot

Single giant table of slots.

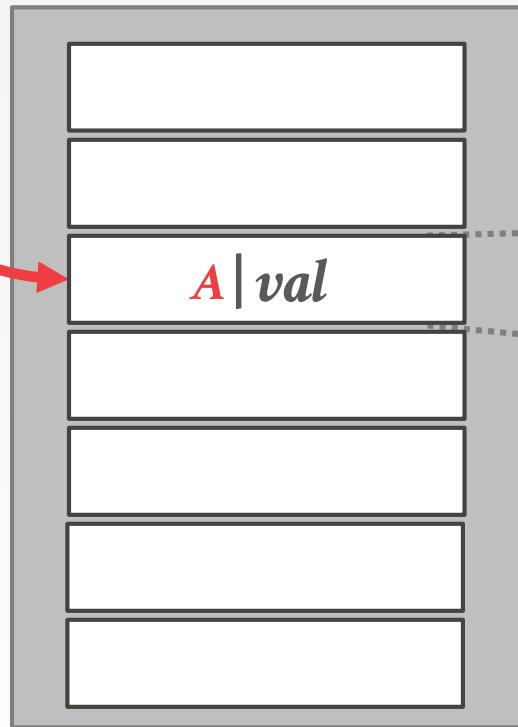
Resolve collisions by linearly searching for the next free slot in the table.

- To determine whether an element is present, hash to a location in the index and scan for it.
- Have to store the key in the index to know when to stop scanning.
- Insertions and deletions are generalizations of lookups.

# LINEAR PROBE HASHING

*hash(key)*

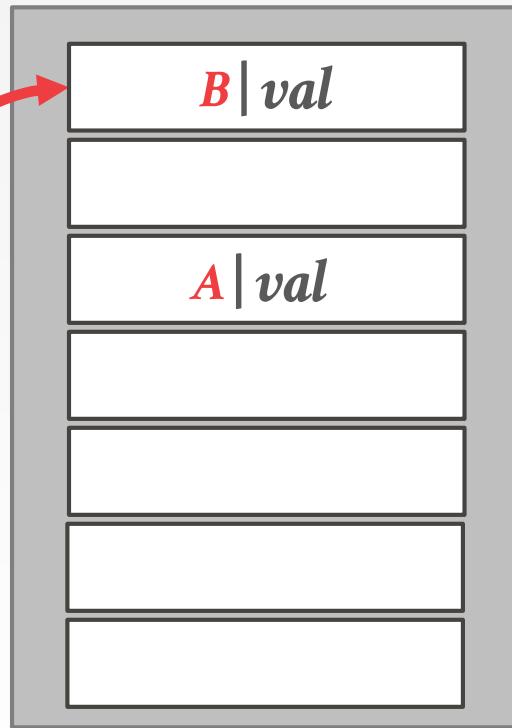
A
B
C
D
E
F



# LINEAR PROBE HASHING

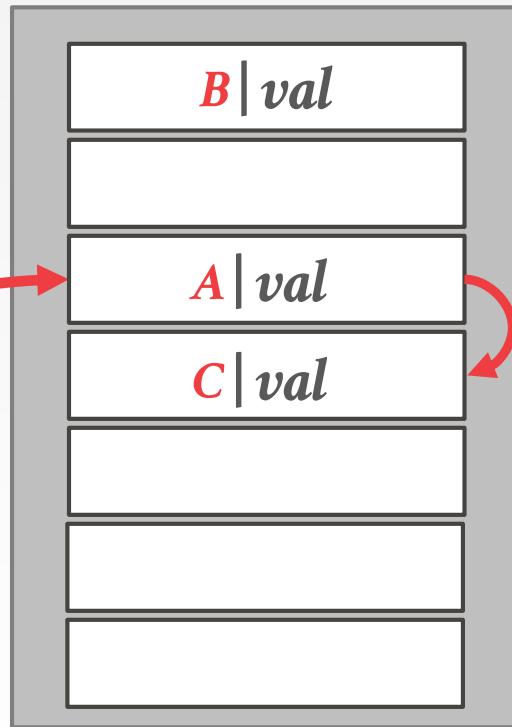
*hash(key)*

A
B
C
D
E
F



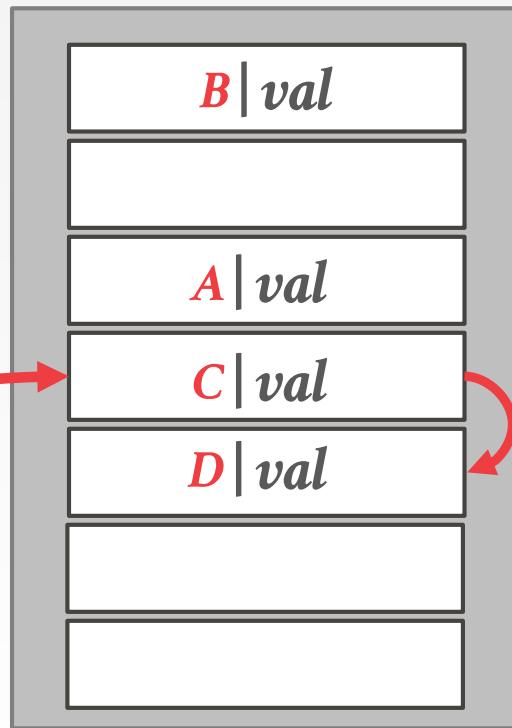
# LINEAR PROBE HASHING

*hash(key)*



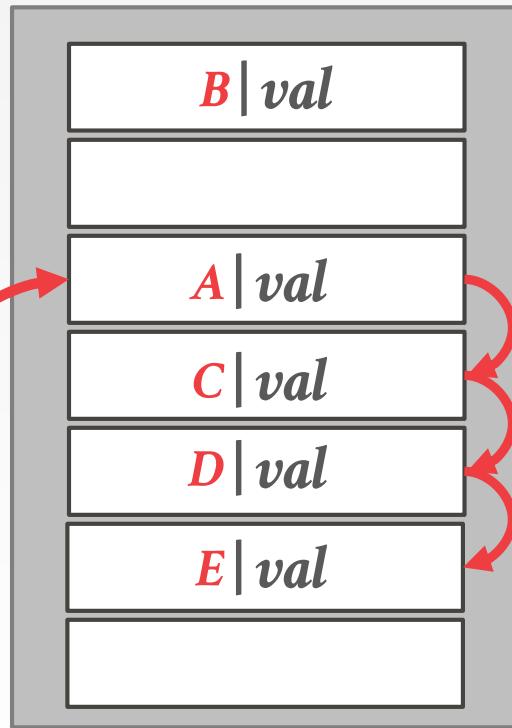
# LINEAR PROBE HASHING

*hash(key)*



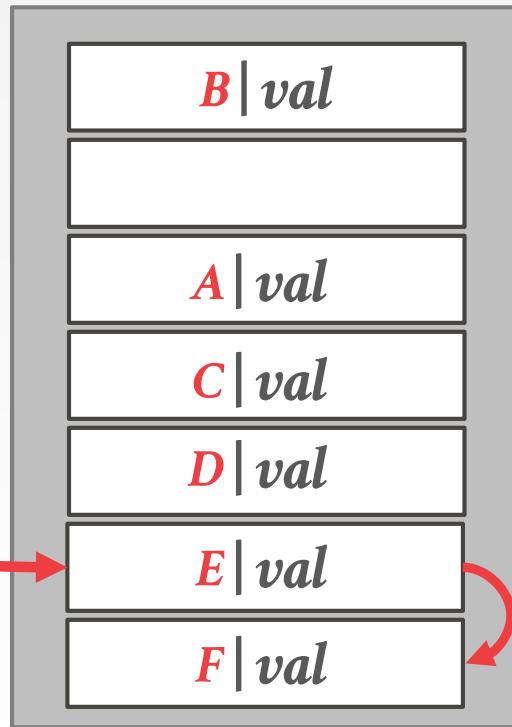
# LINEAR PROBE HASHING

*hash(key)*

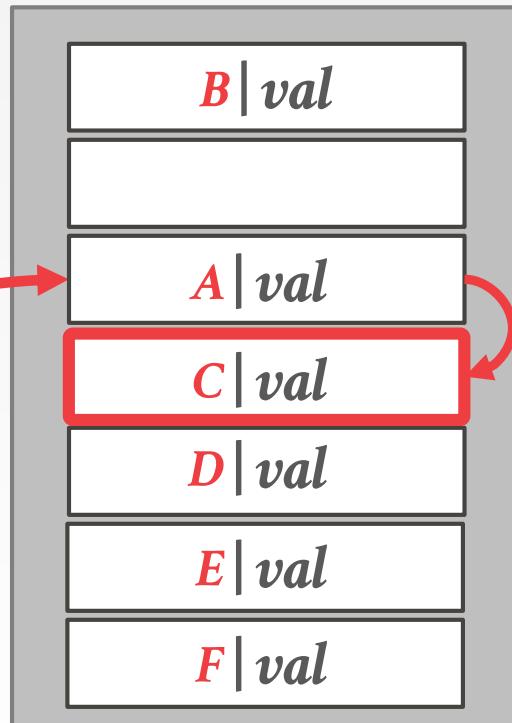
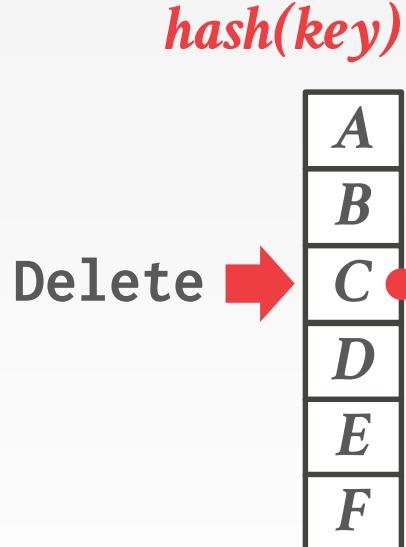


# LINEAR PROBE HASHING

*hash(key)*



# LINEAR PROBE HASHING – DELETES

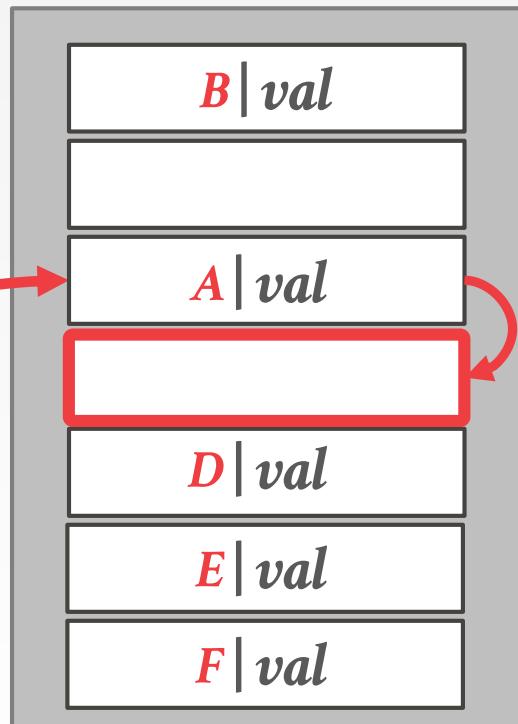


# LINEAR PROBE HASHING – DELETES

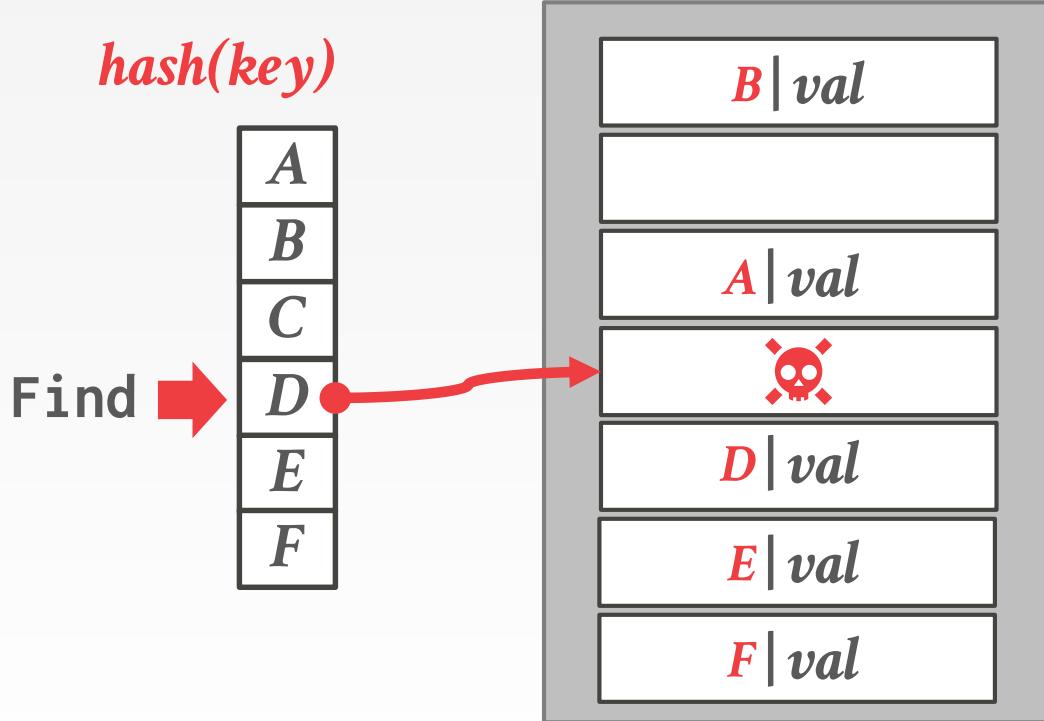
*hash(key)*

Delete 

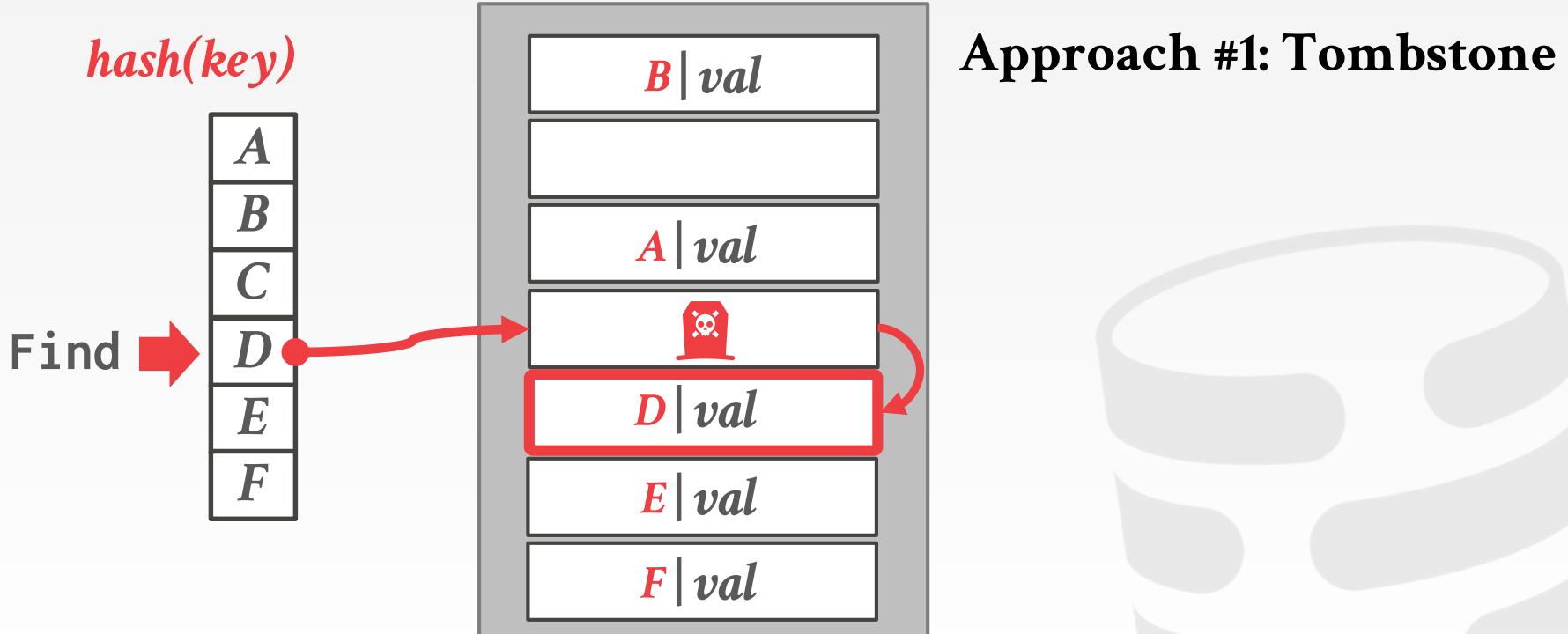
A
B
C
D
E
F



# LINEAR PROBE HASHING – DELETES

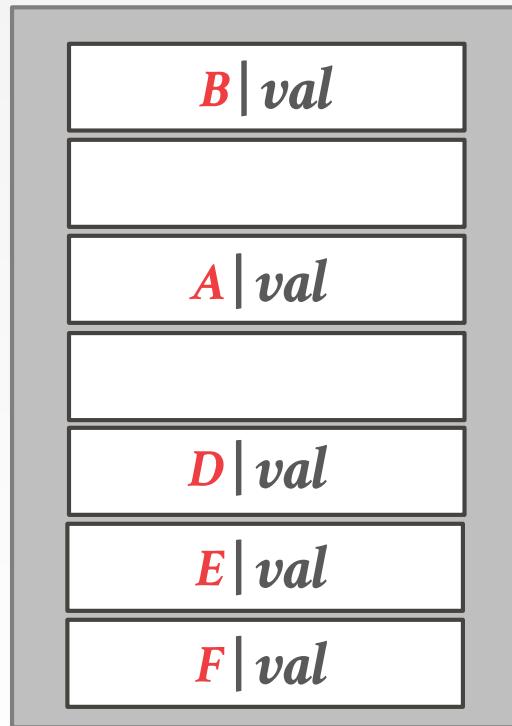
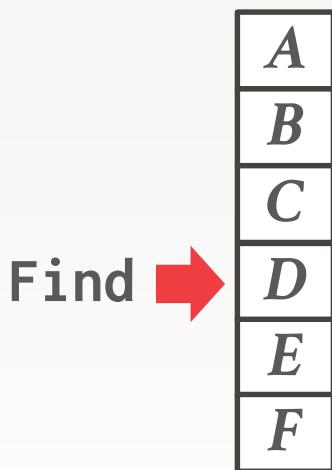


# LINEAR PROBE HASHING – DELETES



# LINEAR PROBE HASHING – DELETES

*hash(key)*

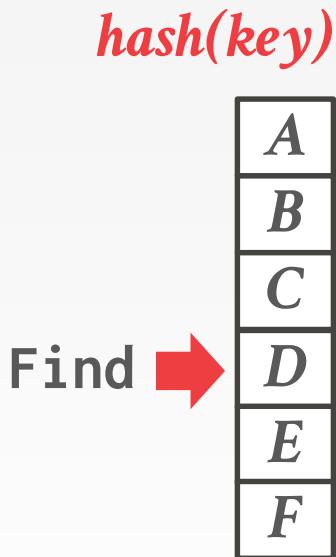


**Approach #1: Tombstone**

**Approach #2: Movement**



# LINEAR PROBE HASHING – DELETES

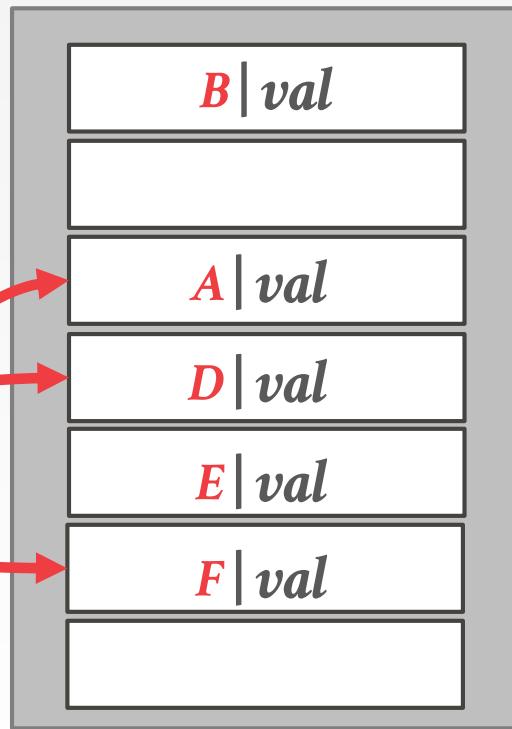
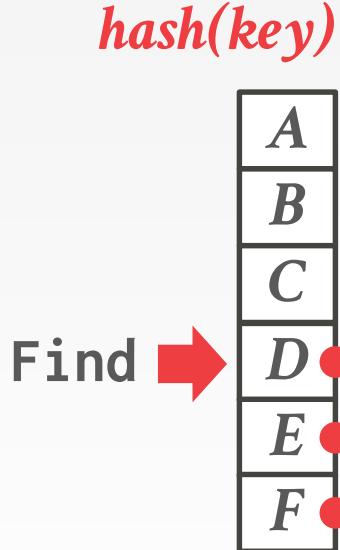


**Approach #1: Tombstone**

**Approach #2: Movement**



# LINEAR PROBE HASHING – DELETES

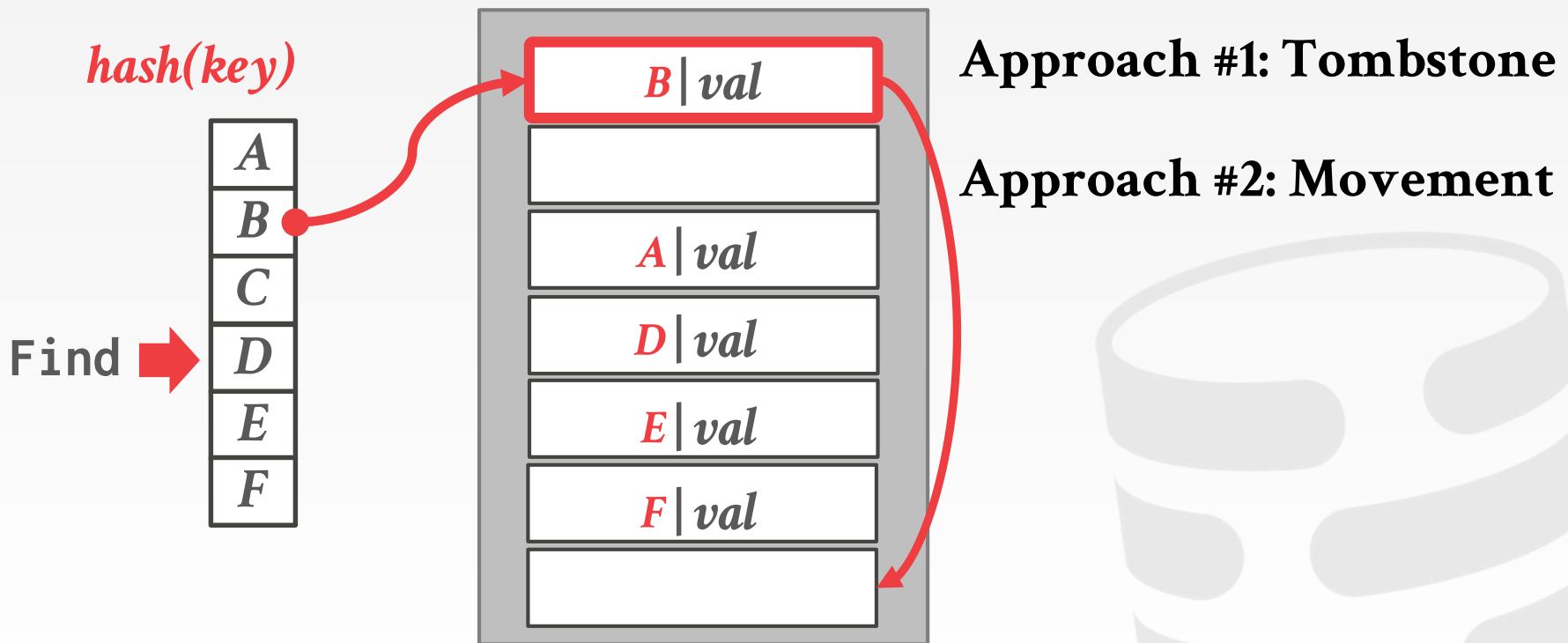


**Approach #1: Tombstone**

**Approach #2: Movement**



# LINEAR PROBE HASHING – DELETES



# NON-UNIQUE KEYS

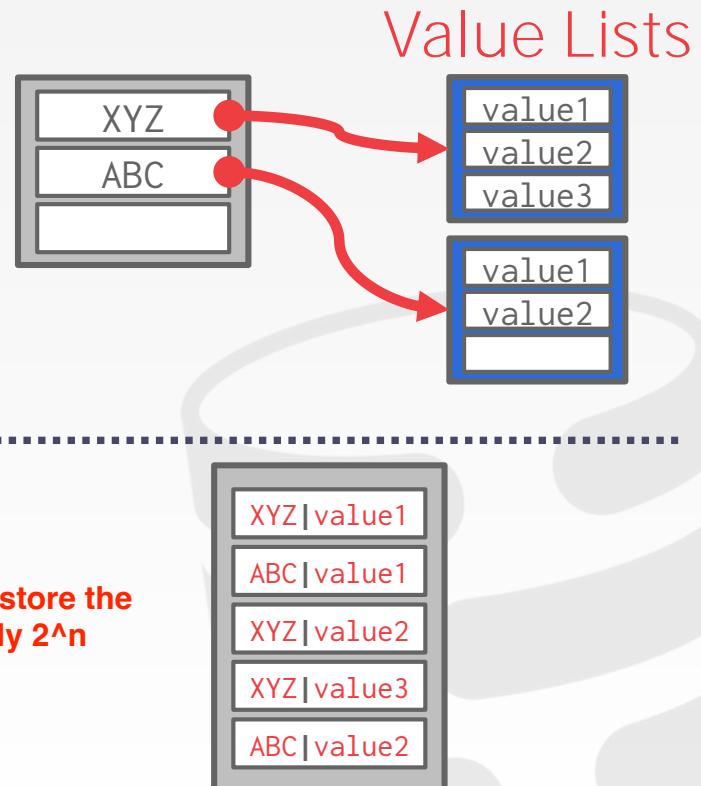
## Choice #1: Separate Linked List

- Store values in separate storage area for each key.

## Choice #2: Redundant Keys

- Store duplicate keys entries together in the hash table.

In practice, people use the second one  
 Even though it is a little bit waste space (because we have to store the keys multiple times and the slots for key-value pairs usually  $2^n$  (where n is the number of key-value pairs ))



# ROBIN HOOD HASHING

---

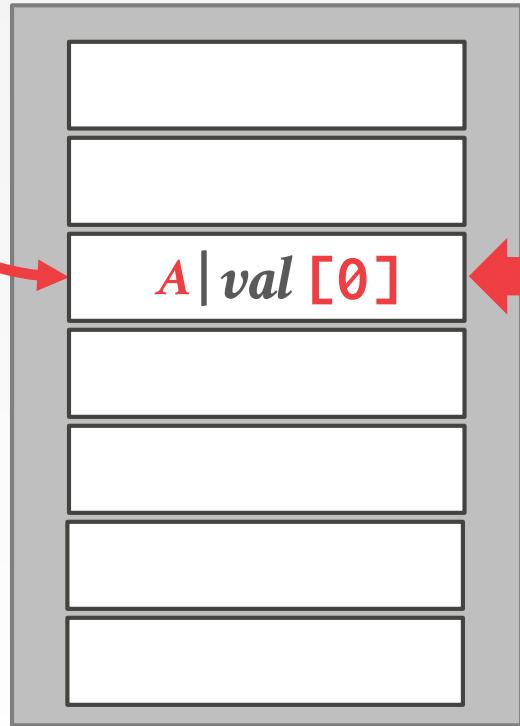
1985

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)*

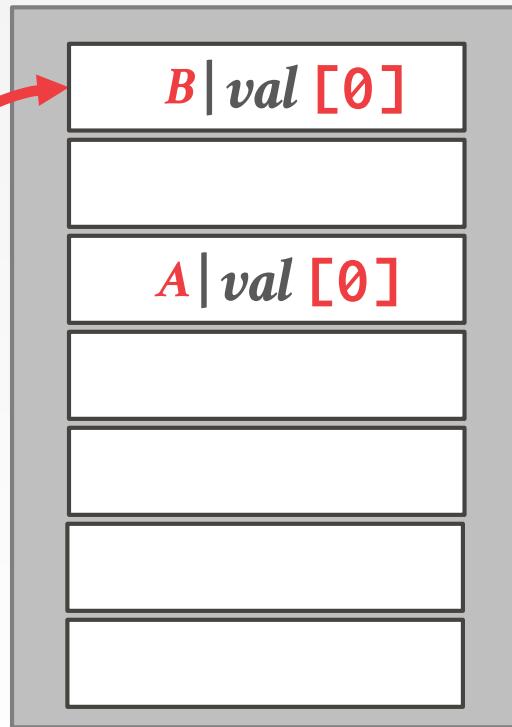


*# of "Jumps" From First Position*

# ROBIN HOOD HASHING

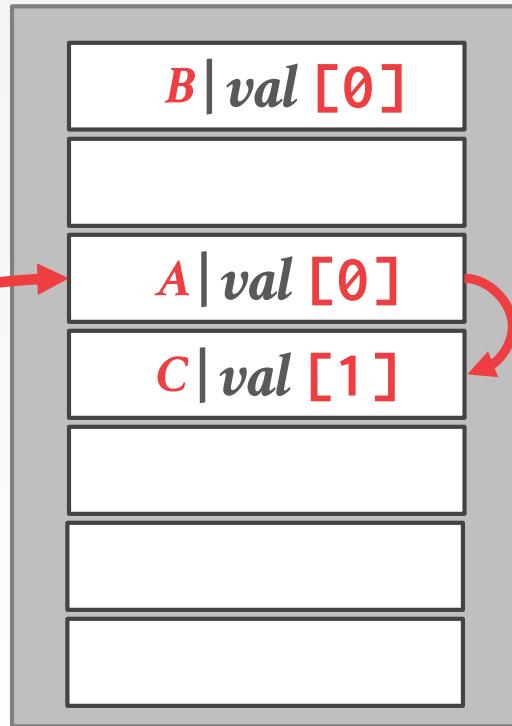
*hash(key)*

A
B
C
D
E
F



# ROBIN HOOD HASHING

*hash(key)*

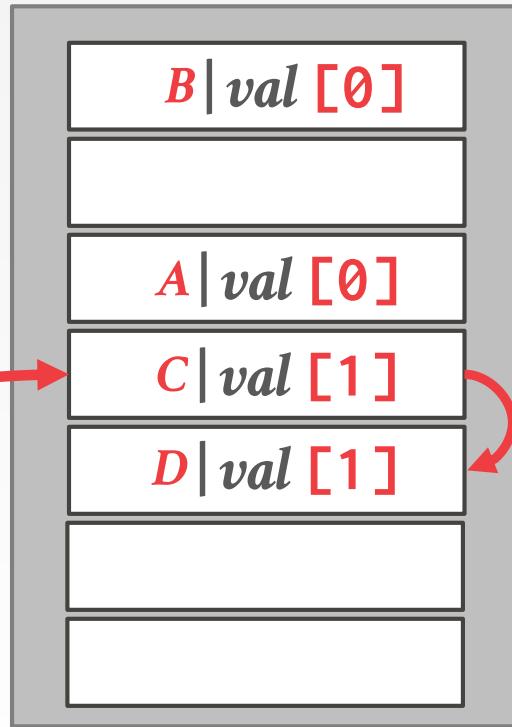


$A[0] == C[0]$



# ROBIN HOOD HASHING

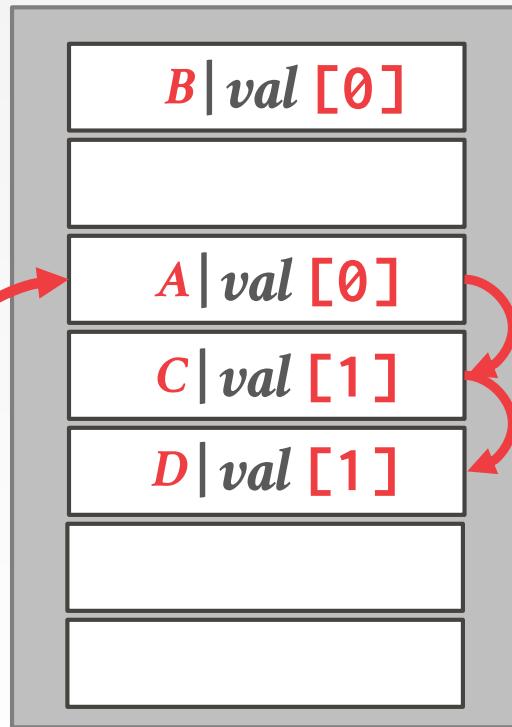
*hash(key)*



$C[1] > D[0]$

# ROBIN HOOD HASHING

*hash(key)*



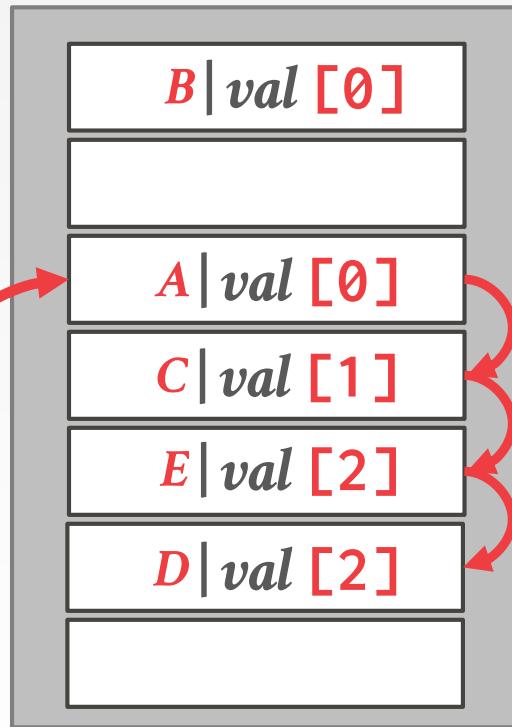
$$A[0] == E[0]$$

$$C[1] == E[1]$$

$$D[1] < E[2]$$

# ROBIN HOOD HASHING

*hash(key)*



Higher count means you are pooler

$A[0] == E[0]$

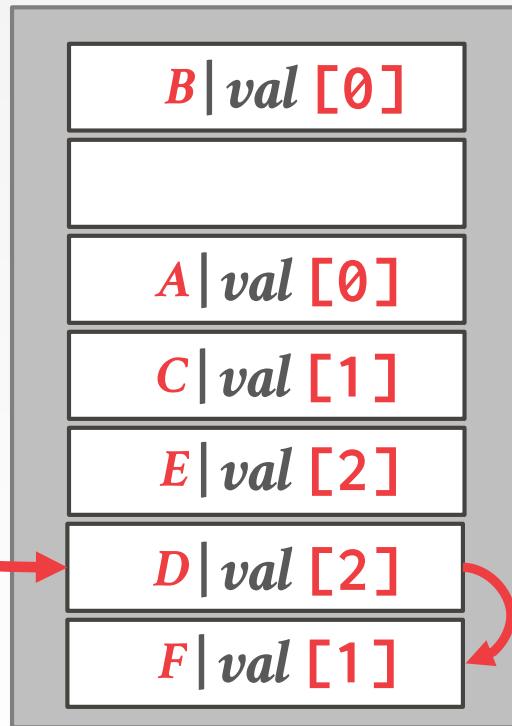
$C[1] == E[1]$

$D[1] < E[2]$

# ROBIN HOOD HASHING

In practice, linear probe hashing crashes everything

*hash(key)*



$D[2] > F[0]$

# CUCKOO HASHING

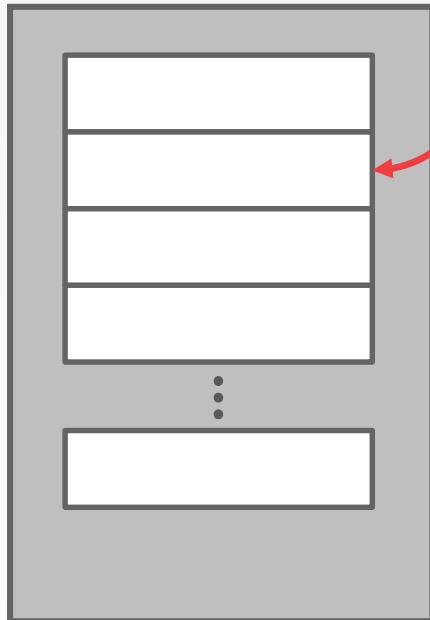
Use multiple hash tables with different hash function seeds.

- 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.

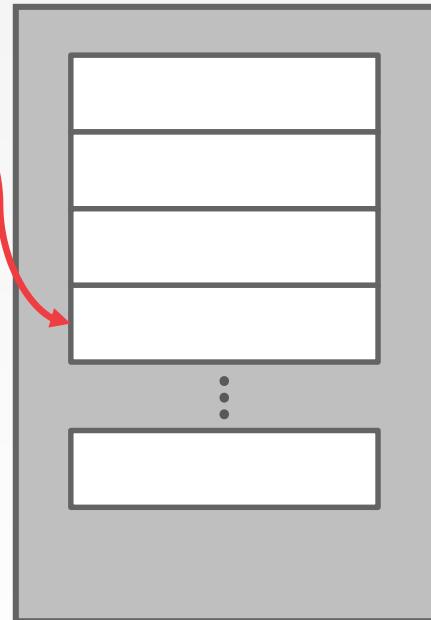
# CUCKOO HASHING

Hash Table #1



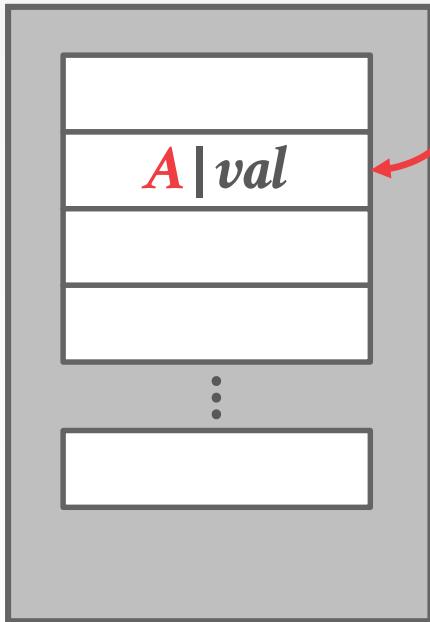
Insert A  
 $hash_1(A)$     $hash_2(A)$

Hash Table #2



# CUCKOO HASHING

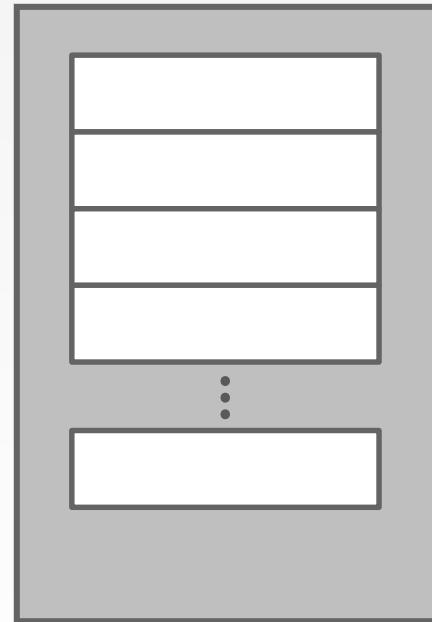
Hash Table #1



Insert A

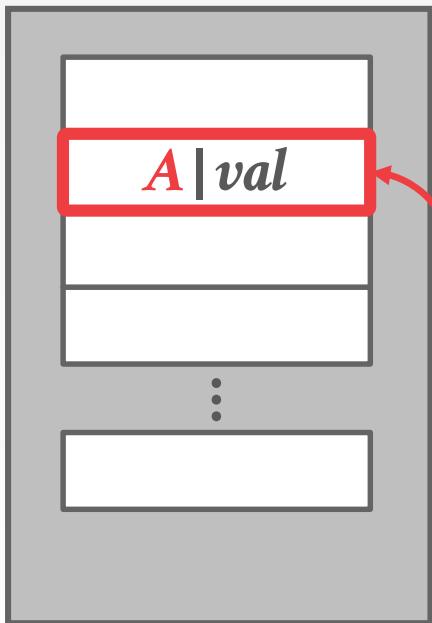
$hash_1(A)$   $hash_2(A)$

Hash Table #2



# CUCKOO HASHING

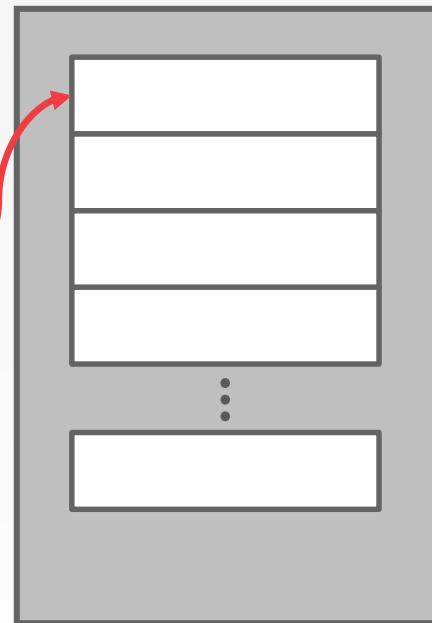
Hash Table #1



**Insert A**  
 $hash_1(A)$     $hash_2(A)$

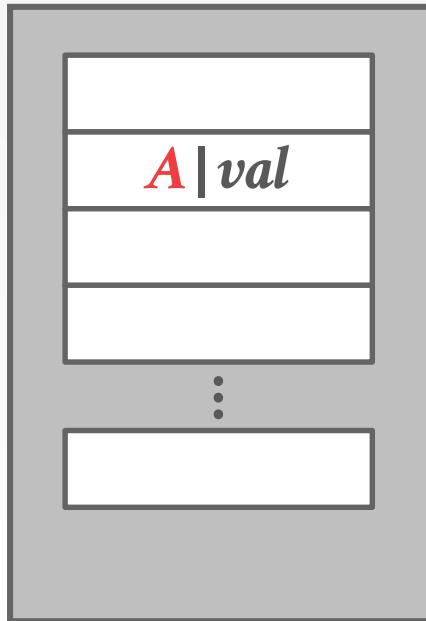
**Insert B**  
 $hash_1(B)$     $hash_2(B)$

Hash Table #2



# CUCKOO HASHING

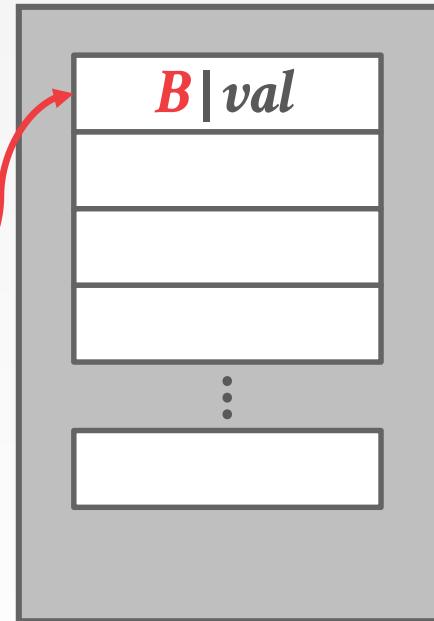
Hash Table #1



Hash Table #2

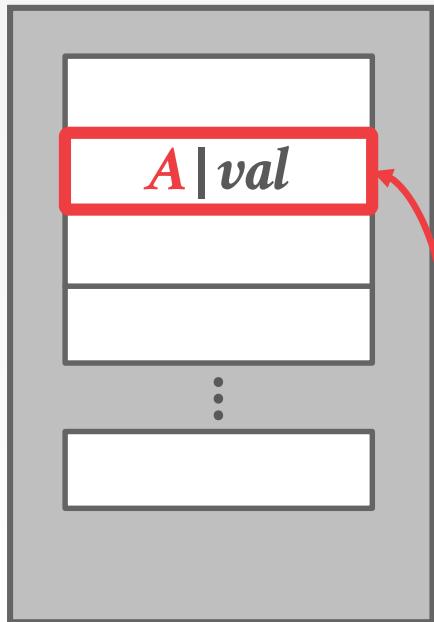
**Insert A**  
 $hash_1(A)$     $hash_2(A)$

**Insert B**  
 $hash_1(B)$     $hash_2(B)$



# CUCKOO HASHING

Hash Table #1

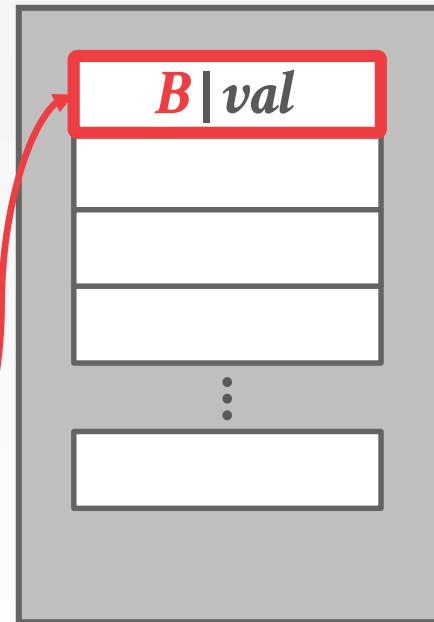


**Insert A**  
 $hash_1(A)$     $hash_2(A)$

**Insert B**  
 $hash_1(B)$     $hash_2(B)$

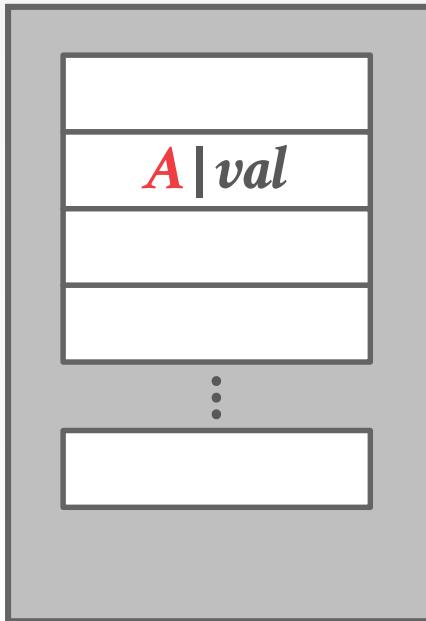
**Insert C**  
 $hash_1(C)$     $hash_2(C)$

Hash Table #2



# CUCKOO HASHING

Hash Table #1

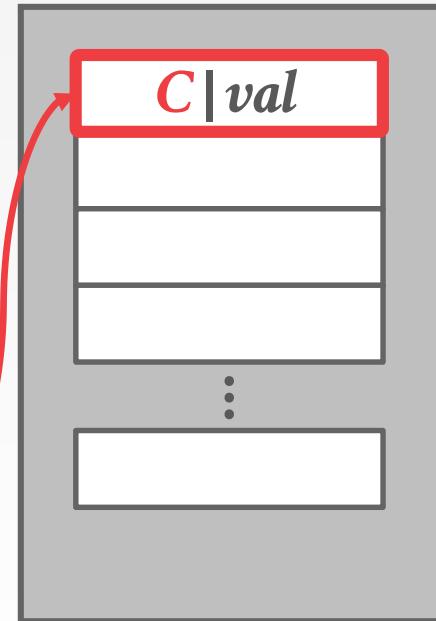


**Insert A**  
 $hash_1(A)$     $hash_2(A)$

**Insert B**  
 $hash_1(B)$     $hash_2(B)$

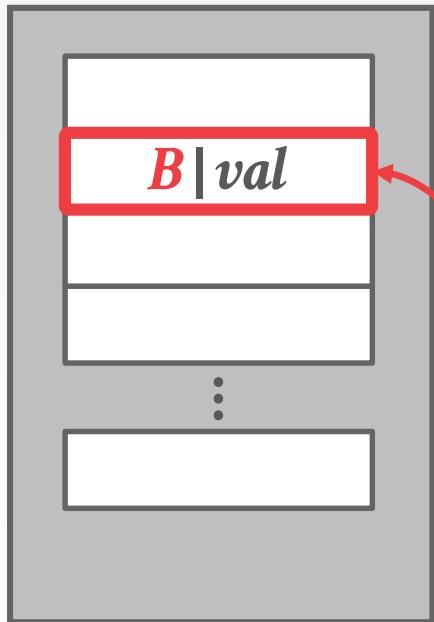
**Insert C**  
 $hash_1(C)$     $hash_2(C)$

Hash Table #2



# CUCKOO HASHING

Hash Table #1



Insert A

$hash_1(A)$   $hash_2(A)$

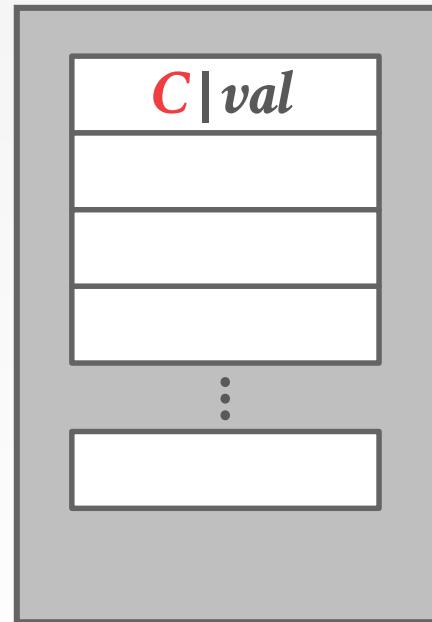
Insert B

$hash_1(B)$   $hash_2(B)$

Insert C

$hash_1(C)$   $hash_2(C)$   
 $hash_1(B)$

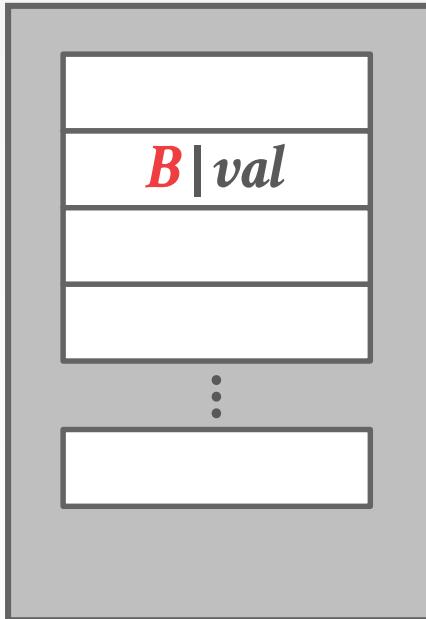
Hash Table #2



# CUCKOO HASHING

Maybe will stuck in an infinite loop, and that's when you resize

Hash Table #1

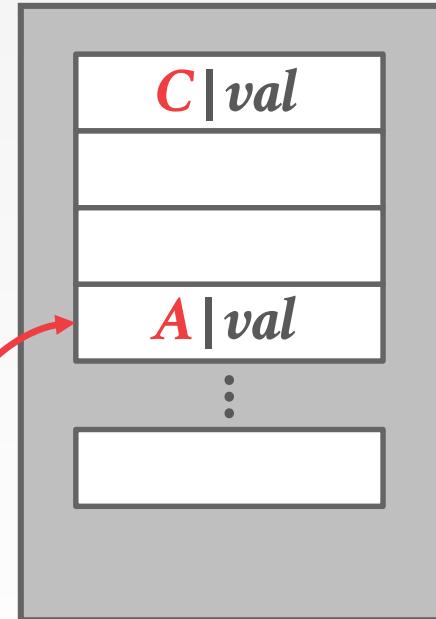


**Insert A**  
 $hash_1(A)$     $hash_2(A)$

**Insert B**  
 $hash_1(B)$     $hash_2(B)$

**Insert C**  
 $hash_1(C)$     $hash_2(C)$   
 $hash_1(B)$   
 $hash_2(A)$

Hash Table #2



# OBSERVATION

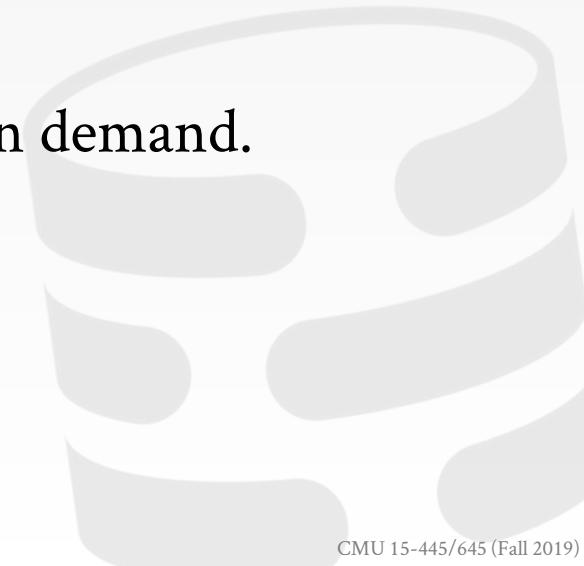
---

The previous hash tables require the DBMS to know the number of elements it wants to store.

- Otherwise it has 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

---

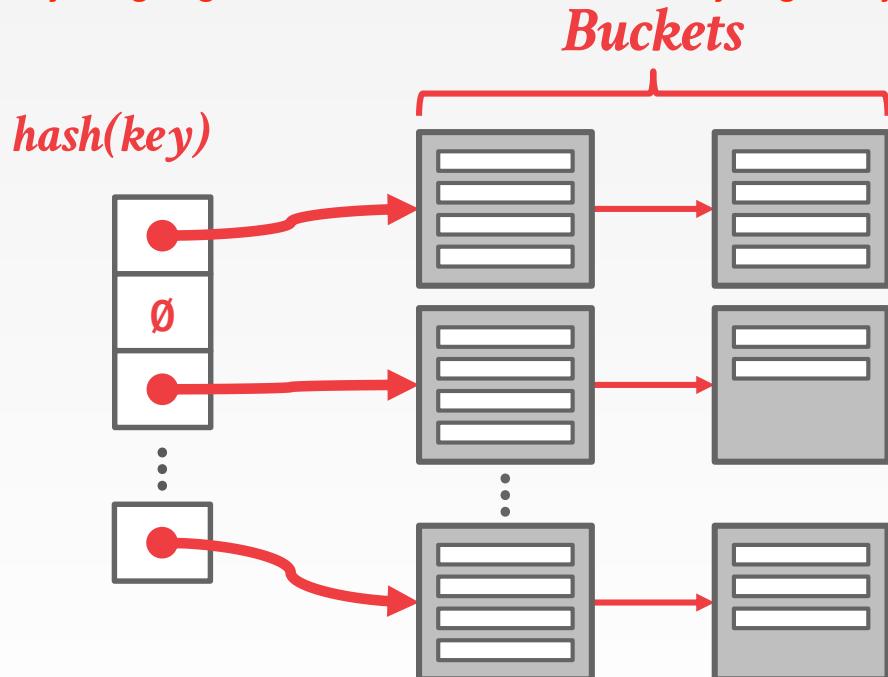
Maintain a linked list of **buckets** for each slot in the hash table.

Resolve collisions by placing all elements with the same hash key into the same bucket.

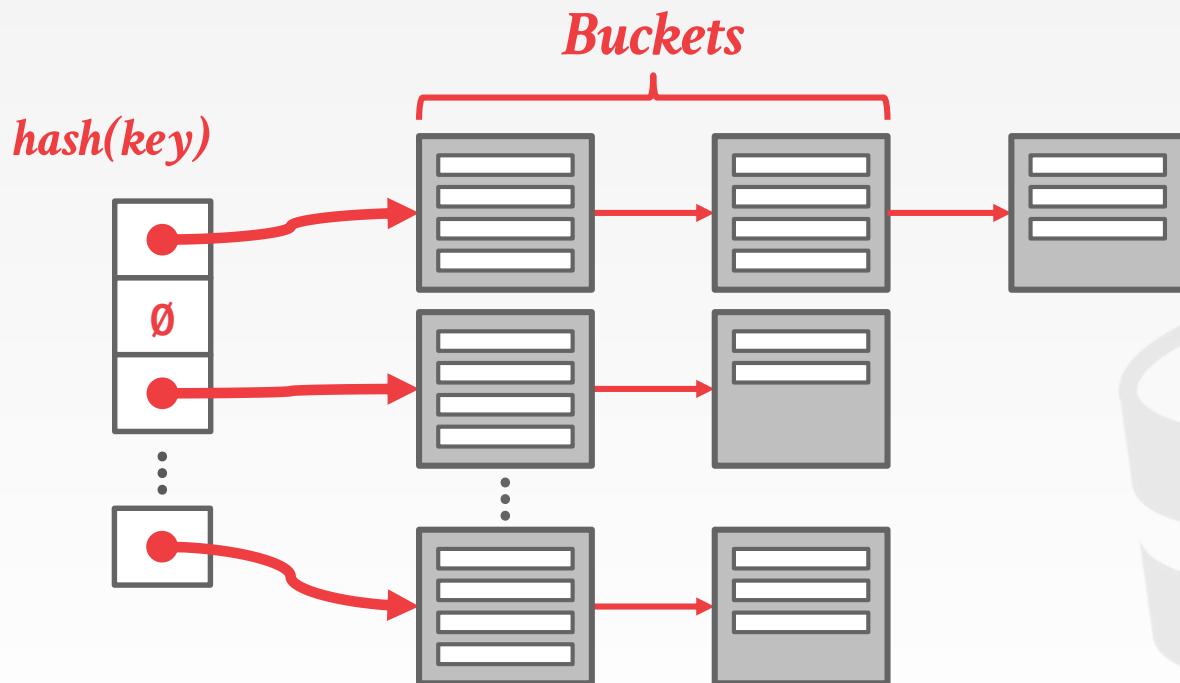
- To determine whether an element is present, hash to its bucket and scan for it.
- Insertions and deletions are generalizations of lookups.

# CHAINED HASHING

They way that they are going to deal with collisions is that they're gonna just keep appending to the end of this bucket chain



# CHAINED HASHING



# EXTENDIBLE HASHING

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

**We are only going to split the chain that overflowed.**

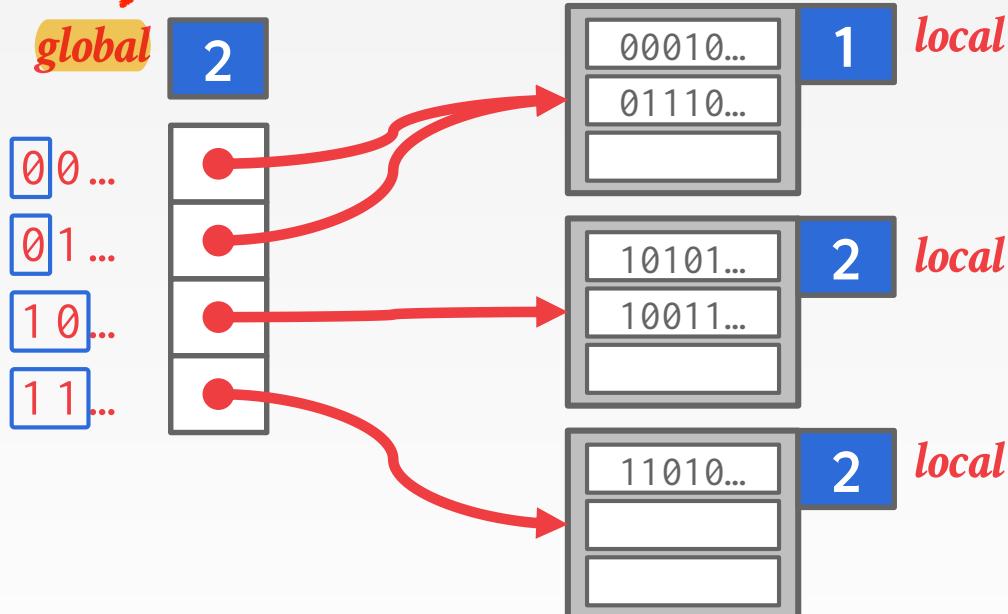
Multiple slot locations can point to the same bucket chain.

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

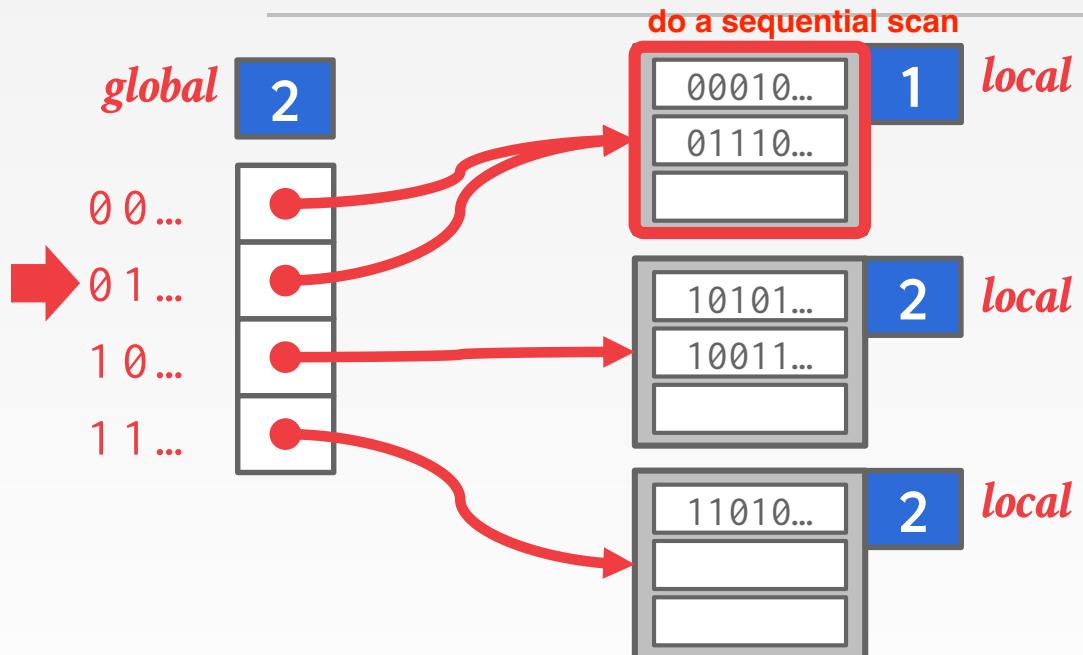
→ Data movement is localized to just the split chain.

The number of bits we have to consider when we want to figure out what bucket to look at

# EXTENDIBLE HASHING



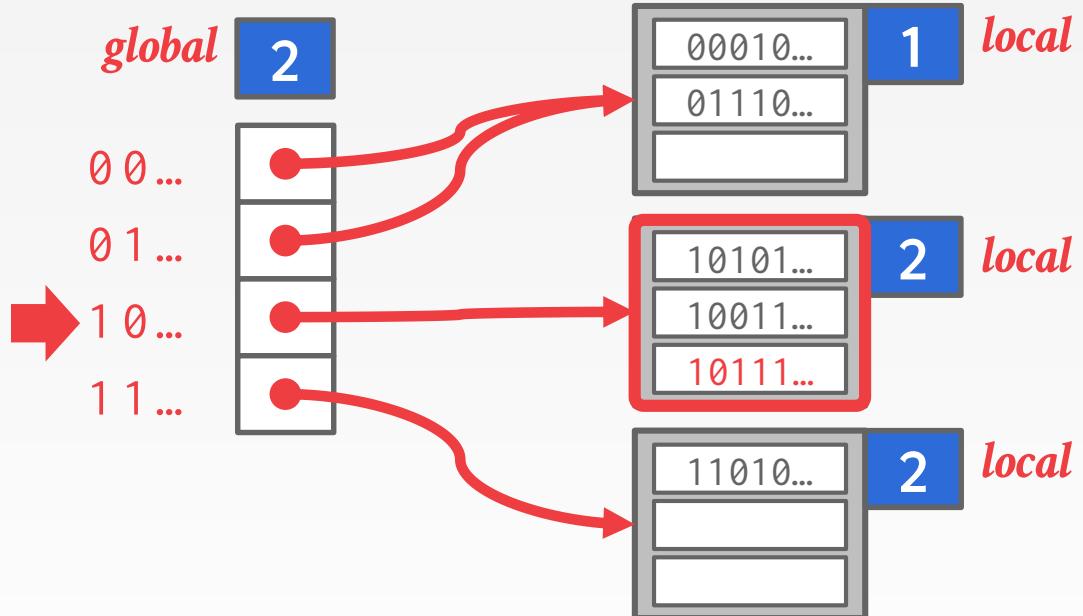
# EXTENDIBLE HASHING



Find A  
 $hash(A) = 01110...$



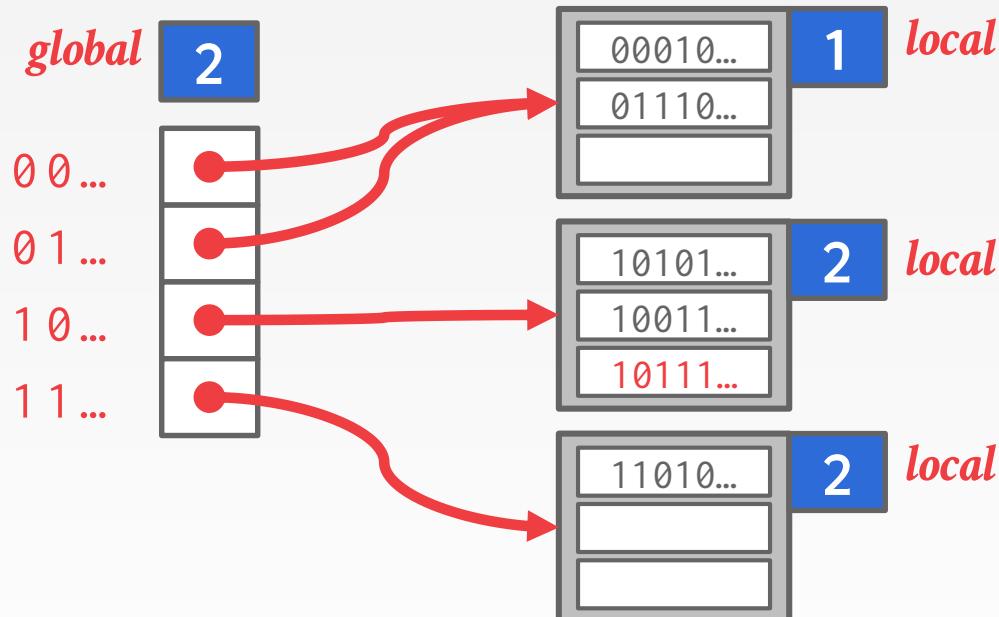
# EXTENDIBLE HASHING



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

Insert B  
 $\text{hash}(B) = \boxed{10}111...$

# EXTENDIBLE HASHING

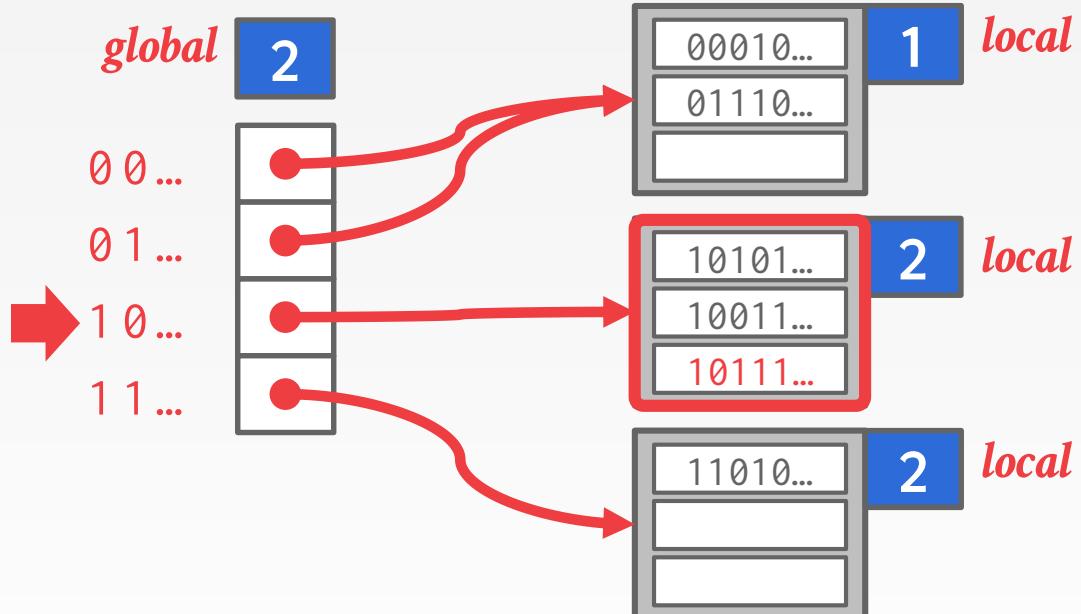


Find A  
 $\text{hash}(A) = 01110...$

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

Insert C  
 $\text{hash}(C) = \boxed{10100...}$

# EXTENDIBLE HASHING

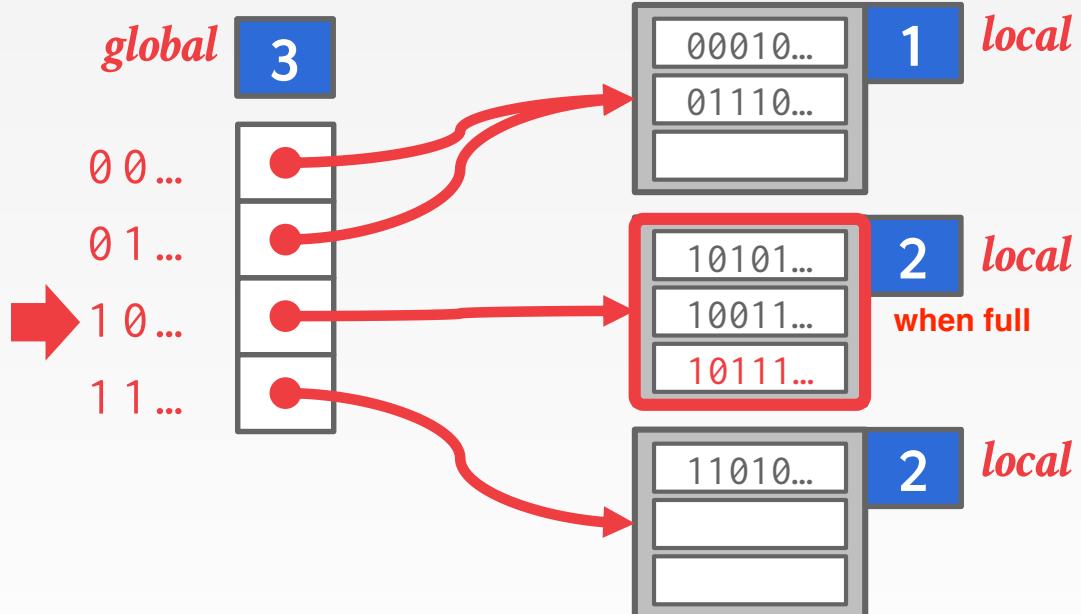


Find A  
 $\text{hash}(A) = 01110...$

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

Insert C  
 $\text{hash}(C) = \boxed{10100...}$

# EXTENDIBLE HASHING



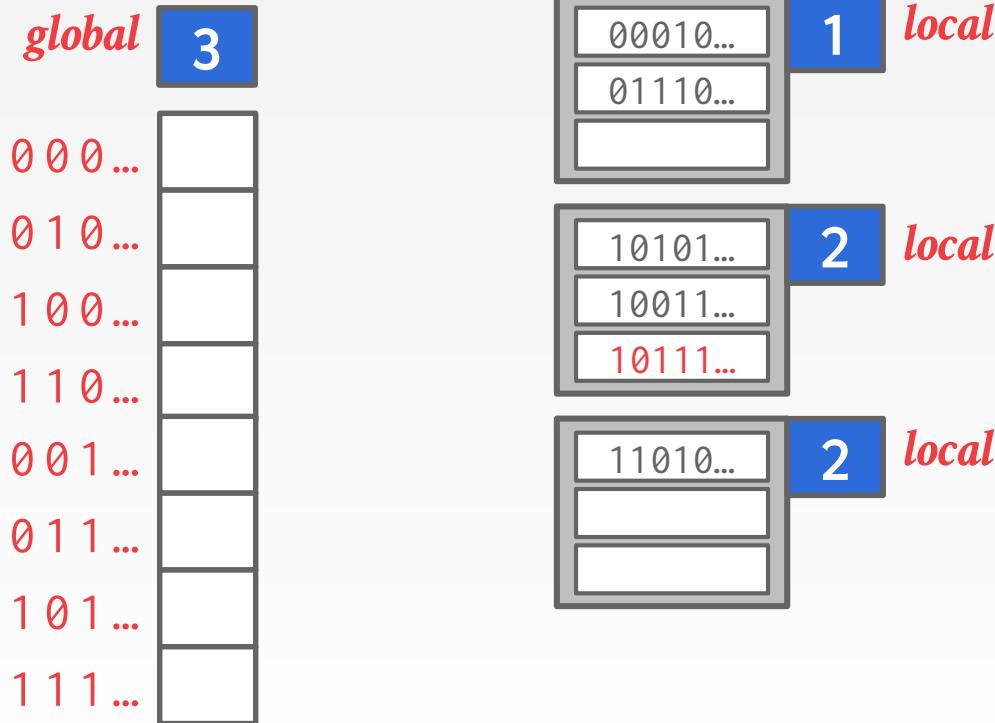
**Find A**  
 $\text{hash}(A) = 01110\dots$

**Insert B**  
 $\text{hash}(B) = 10111\dots$

**Insert C**  
 $\text{hash}(C) = \boxed{10100\dots}$

# EXTENDIBLE HASHING

Double the size of my slot array, won't cost too much  
Since they are just pointers (in sequence)



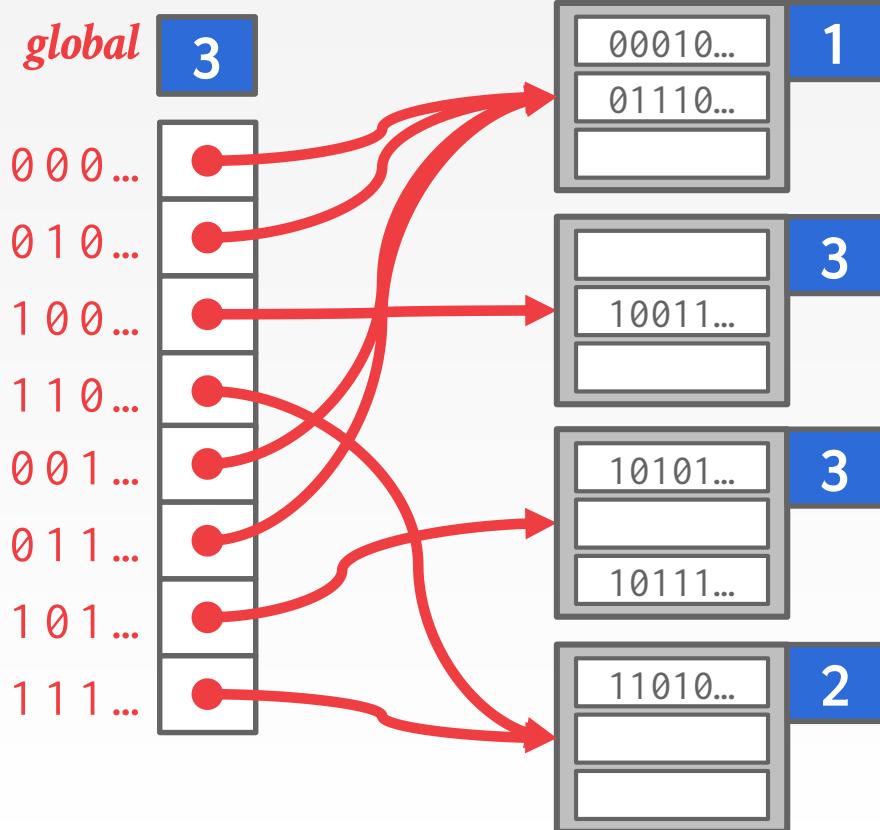
Find A  
 $\text{hash}(A) = 01110...$

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

Insert C  
 $\text{hash}(C) = \boxed{10100...}$

# EXTENDIBLE HASHING

Just splitting that one page I had before to make another page (just two pages)



**Find A**

$\text{hash}(A) = 01110\dots$

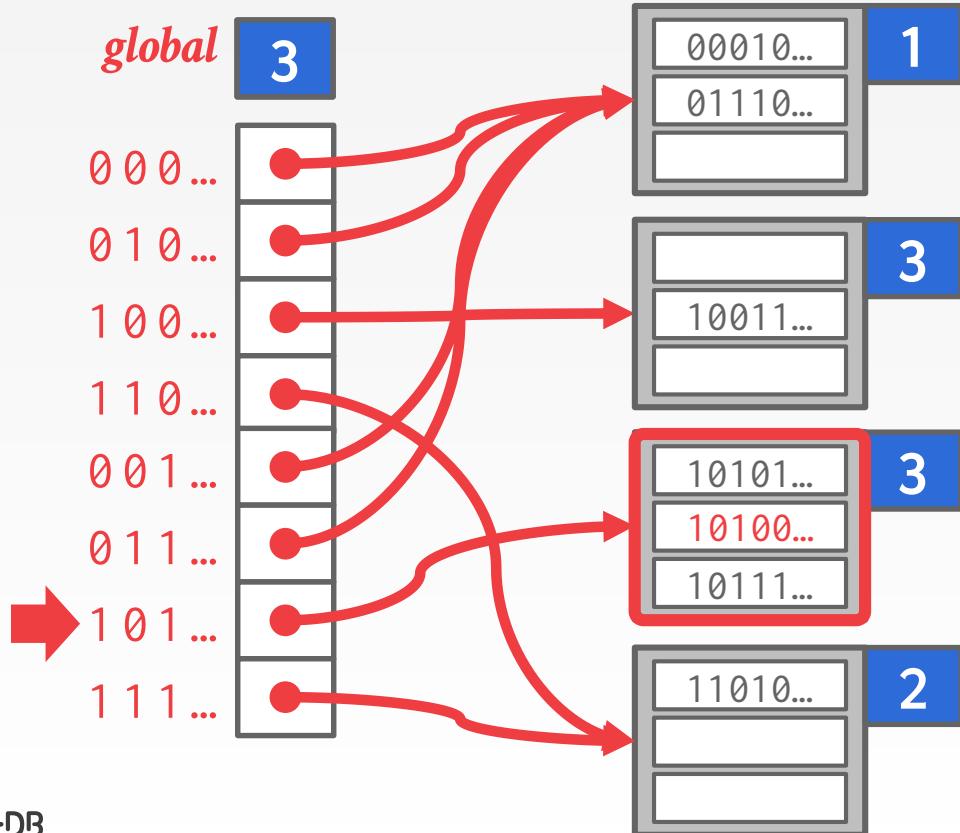
**Insert B**

$\text{hash}(B) = 10111\dots$

**Insert C**

$\text{hash}(C) = 10100\dots$

# EXTENDIBLE HASHING



Find A

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

Insert B

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

Insert C

$\text{hash}(C) = \boxed{101}00...$

# 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.

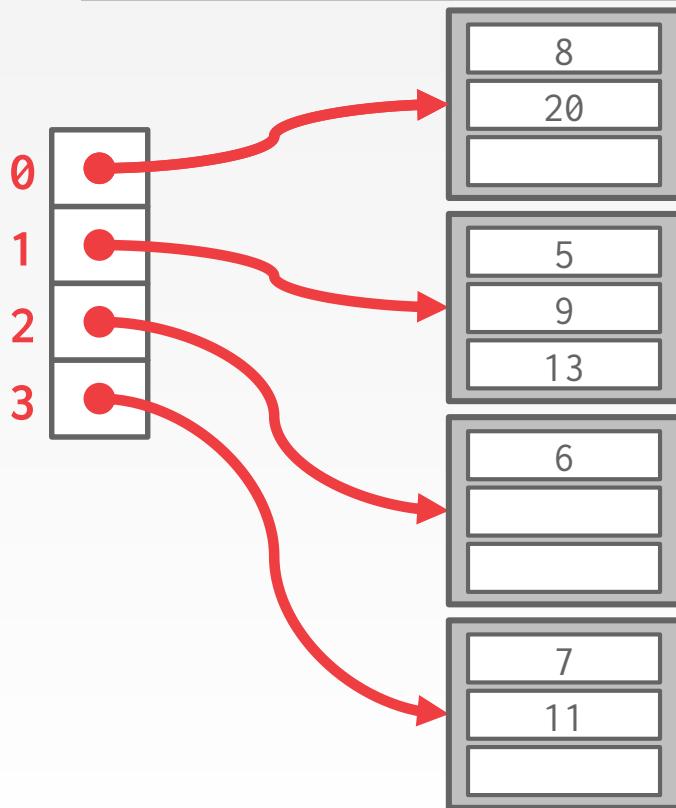
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

- Space Utilization
- Average Length of Overflow Chains

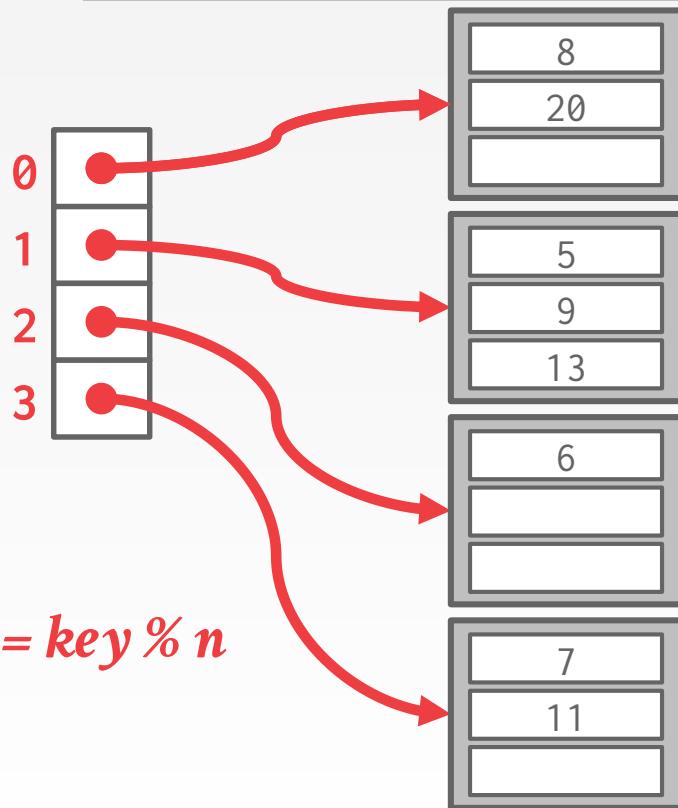


# LINEAR HASHING



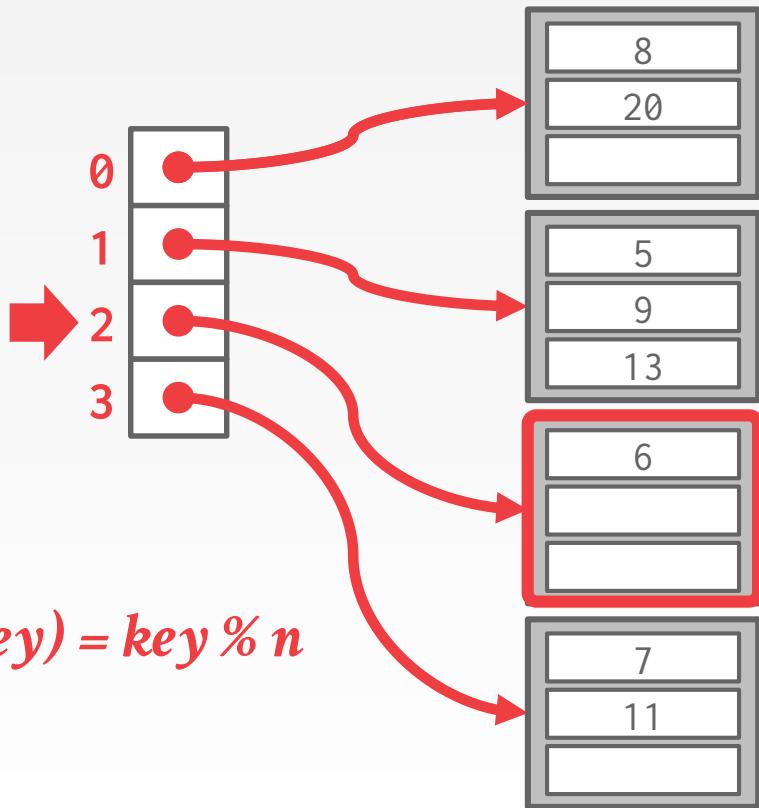
# LINEAR HASHING

*Split  
Pointer*



# LINEAR HASHING

*Split  
Pointer*



Find 6

$$hash_1(6) = 6 \% 4 = 2$$

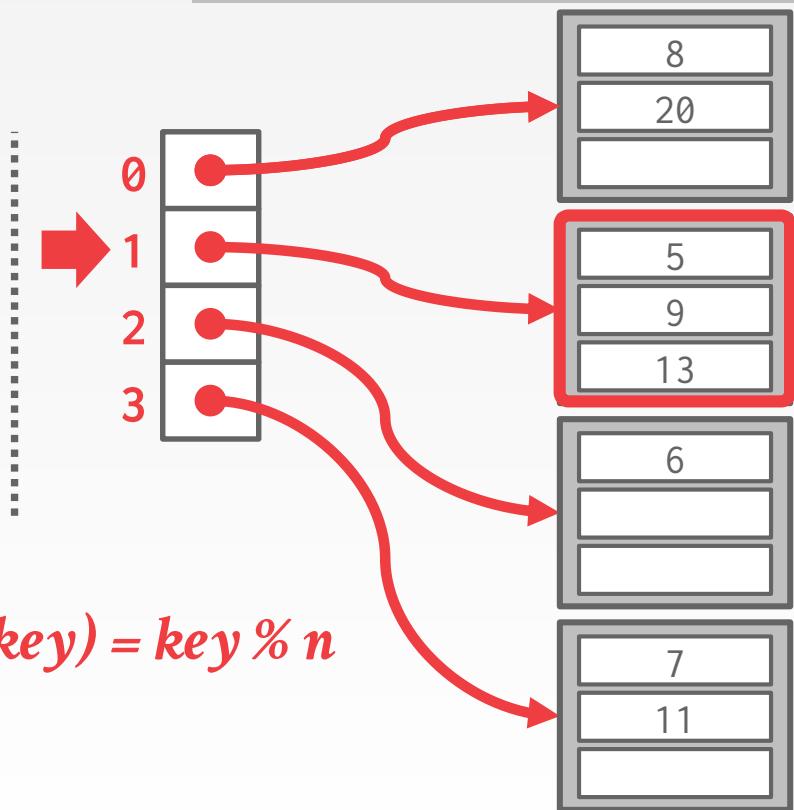


How far along we split in our split function

The next bucket we want to split whenever we have an overflow anywhere in the hash table That is

## LINEAR HASHING

*Split  
Pointer*



Find 6

$$hash_1(6) = 6 \% 4 = 2$$

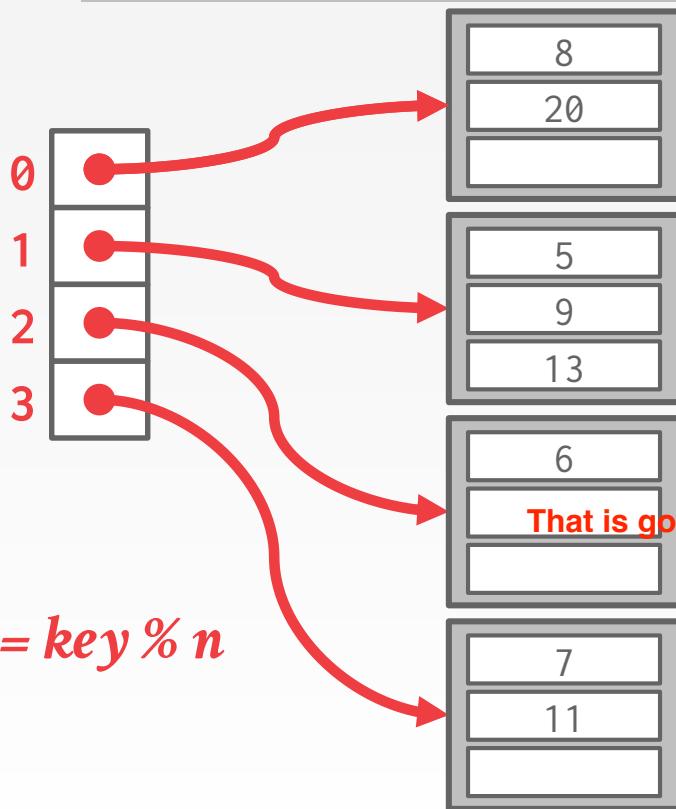
Insert 17

$$hash_1(17) = 17 \% 4 = 1$$



# LINEAR HASHING

*Split  
Pointer*



Find 6

$$hash_1(6) = 6 \% 4 = 2$$

Insert 17

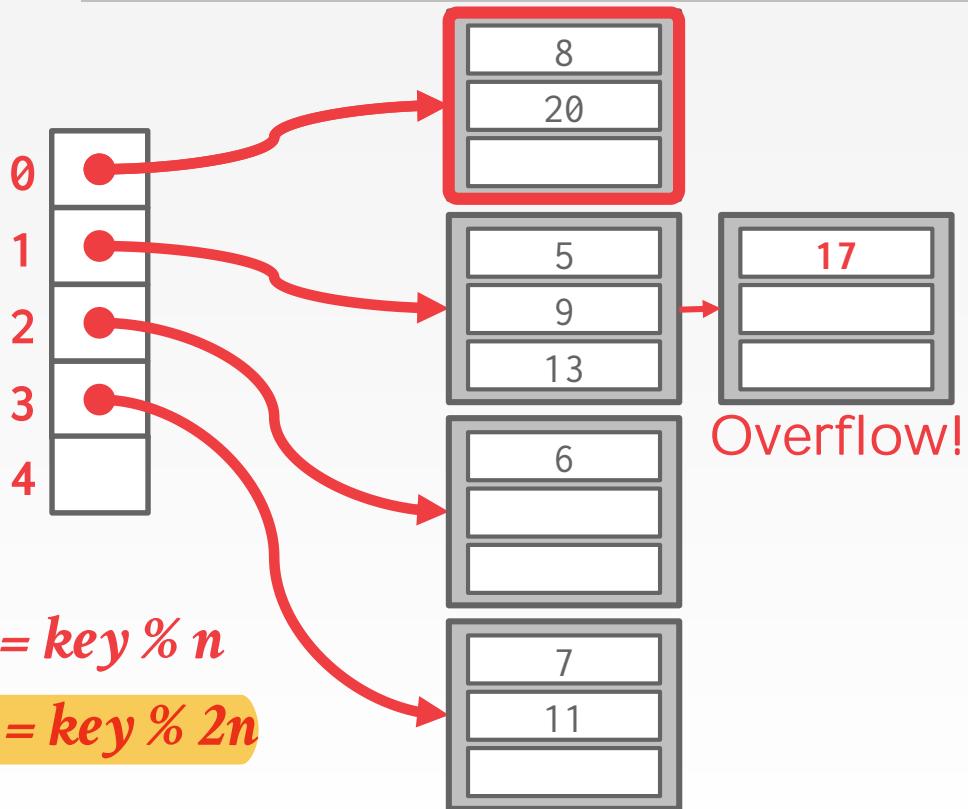
$$hash_1(17) = 17 \% 4 = 1$$

Overflow!

That is going to trigger a split wherever the split pointer is pointing at

# LINEAR HASHING

*Split  
Pointer*



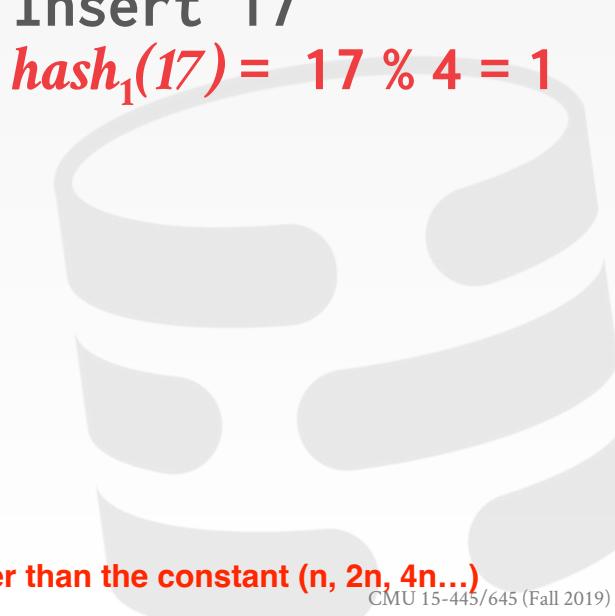
Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

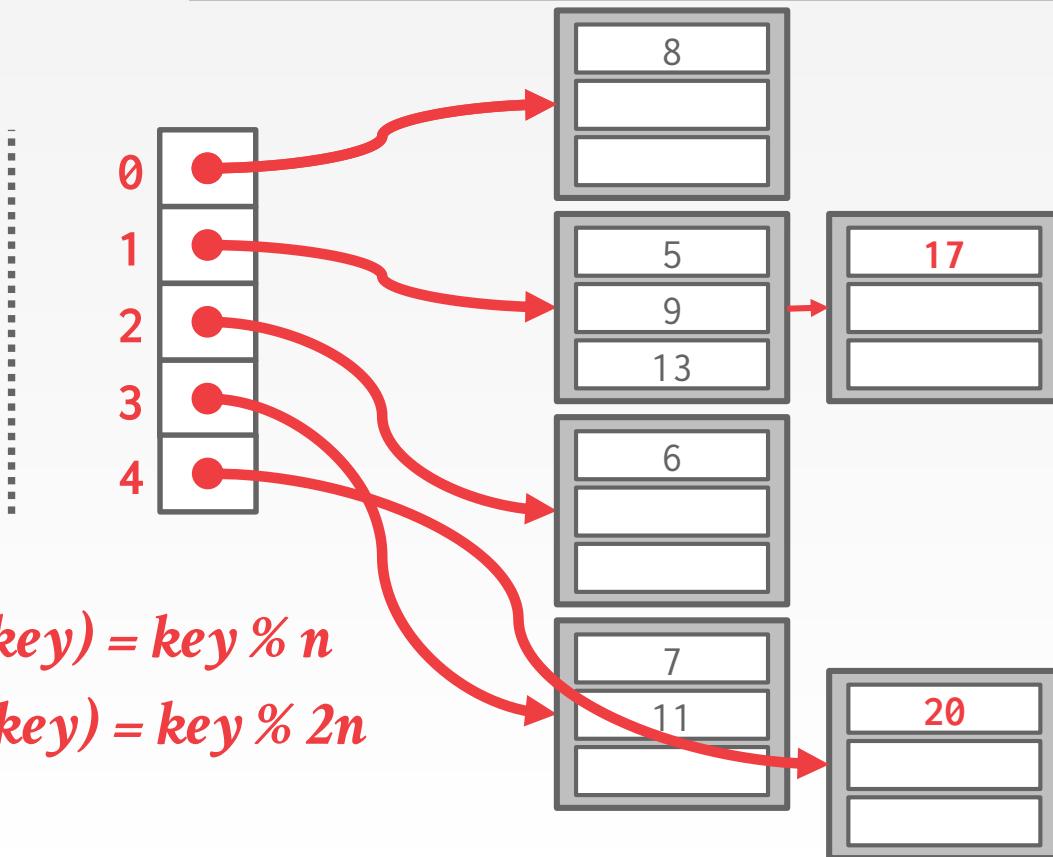
Overflow!



We double only whenever the number of splitting (num of entries) greater than the constant (n, 2n, 4n...)

# LINEAR HASHING

*Split  
Pointer*



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

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

**Find 6**

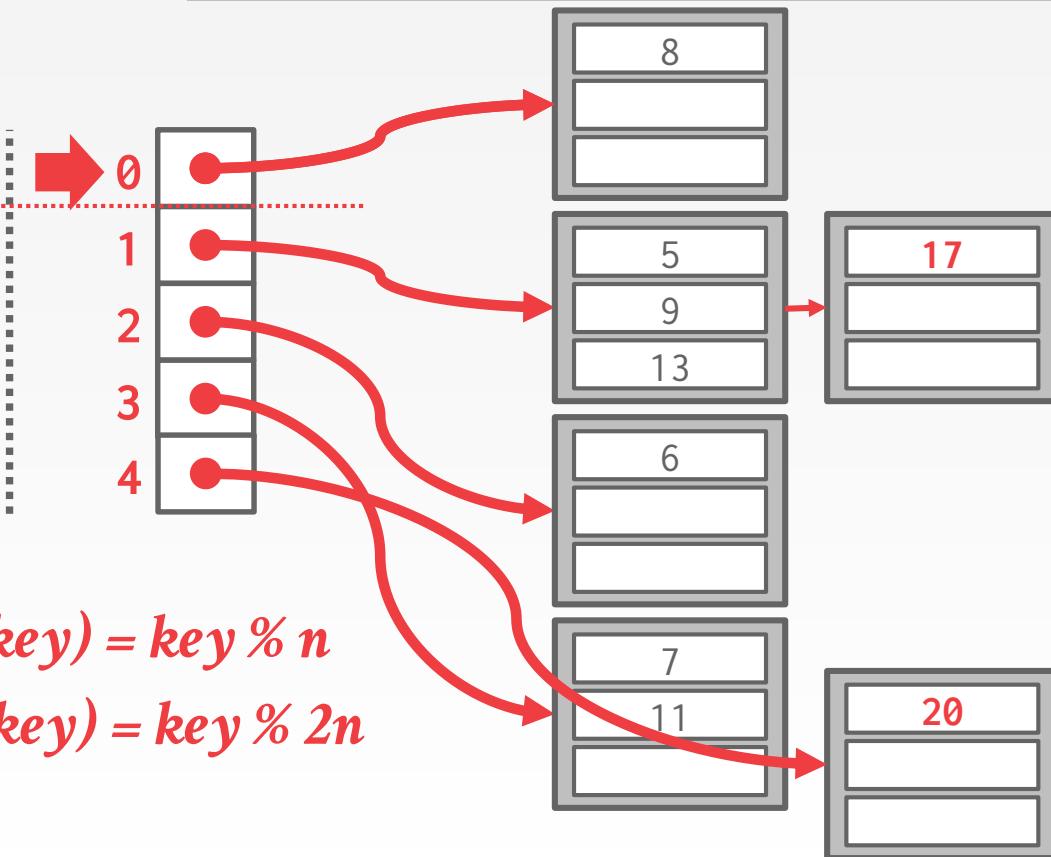
$$\text{hash}_1(6) = 6 \% 4 = 2$$

**Insert 17**

$$\text{hash}_1(17) = 17 \% 4 = 1$$

# LINEAR HASHING

*Split  
Pointer*



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

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

**Find 6**

$$\text{hash}_1(6) = 6 \% 4 = 2$$

**Insert 17**

$$\text{hash}_1(17) = 17 \% 4 = 1$$

**Find 20**

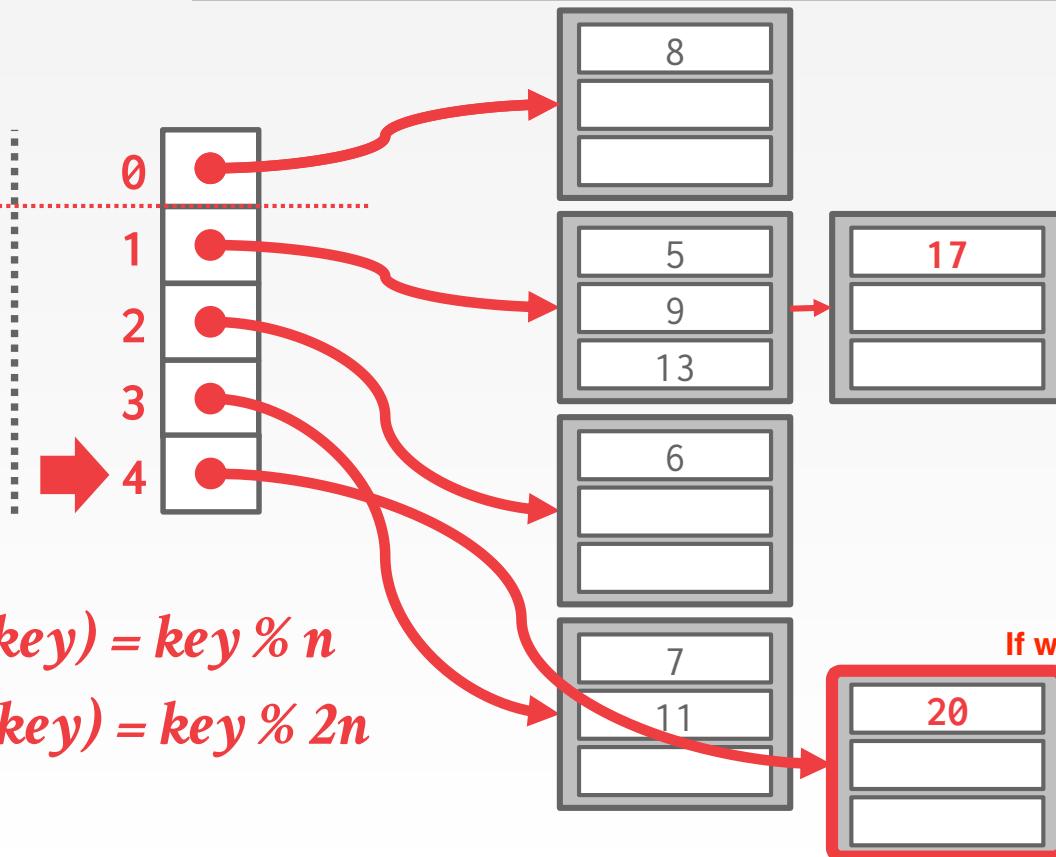
$$\text{hash}_1(20) = 20 \% 4 = 0$$

If we call Find 17 at that time:  
 $\text{hash}(17) = 17 \% 4 = 1$ , found in  
the overflow page!

Whenever I want to do a lookup, I first hash the first hash function.  
 and if the thing I am pointing to is above where that demarcation line (for the split pointer)  
 And then I know that the bucket that I'm looking at has already been split (or need to be split).  
 therefore now I need to look at the second function

## LINEAR HASHING

*Split  
Pointer*



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

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

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

**Insert 17**  
 $\text{hash}_1(17) = 17 \% 4 = 1$

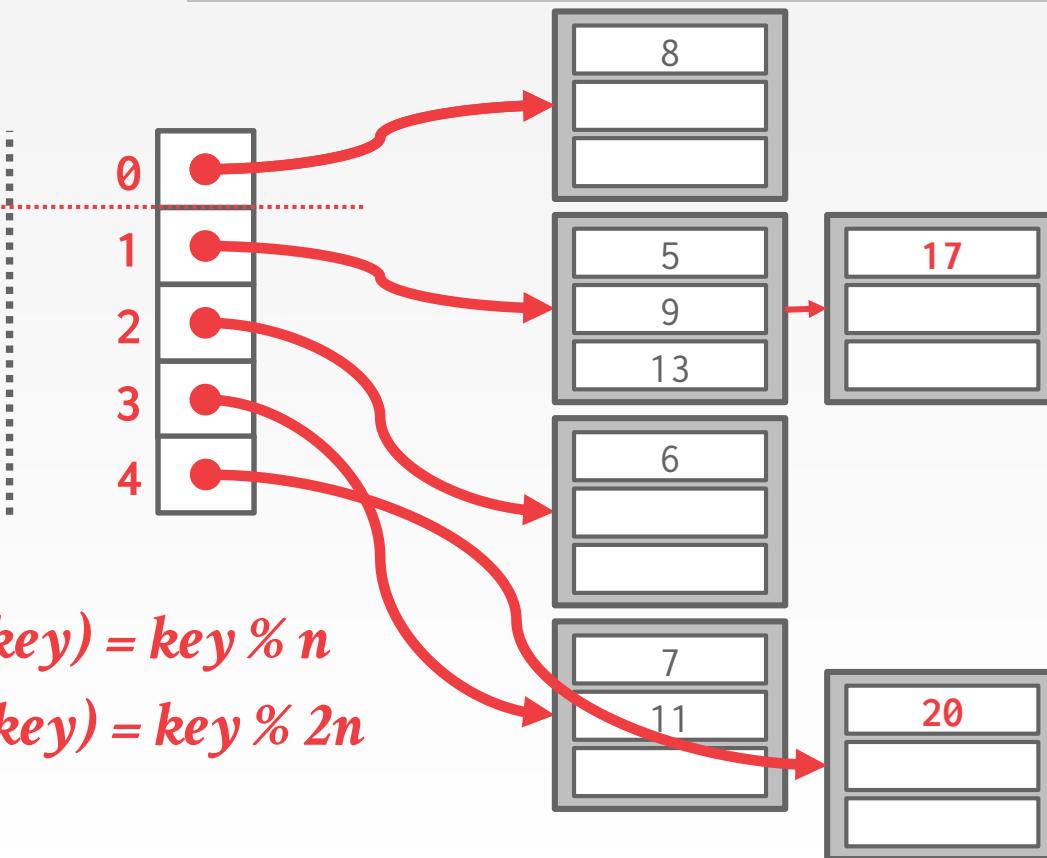
**Find 20**  
 $\text{hash}_1(20) = 20 \% 4 = 0$   
 $\text{hash}_2(20) = 20 \% 8 = 4$

If we insert more numbers across different pages  
 And if an overflow occur, then we need  
 to inc the split pointer  
 next page:

Let's say the split pointer == 3, if we wants to Find 13, then we need to redistribute it using the 2nd hash function

# LINEAR HASHING

*Split  
Pointer*



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

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

**Find 6**

$$\text{hash}_1(6) = 6 \% 4 = 2$$

**Insert 17**

$$\text{hash}_1(17) = 17 \% 4 = 1$$

**Find 20**

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

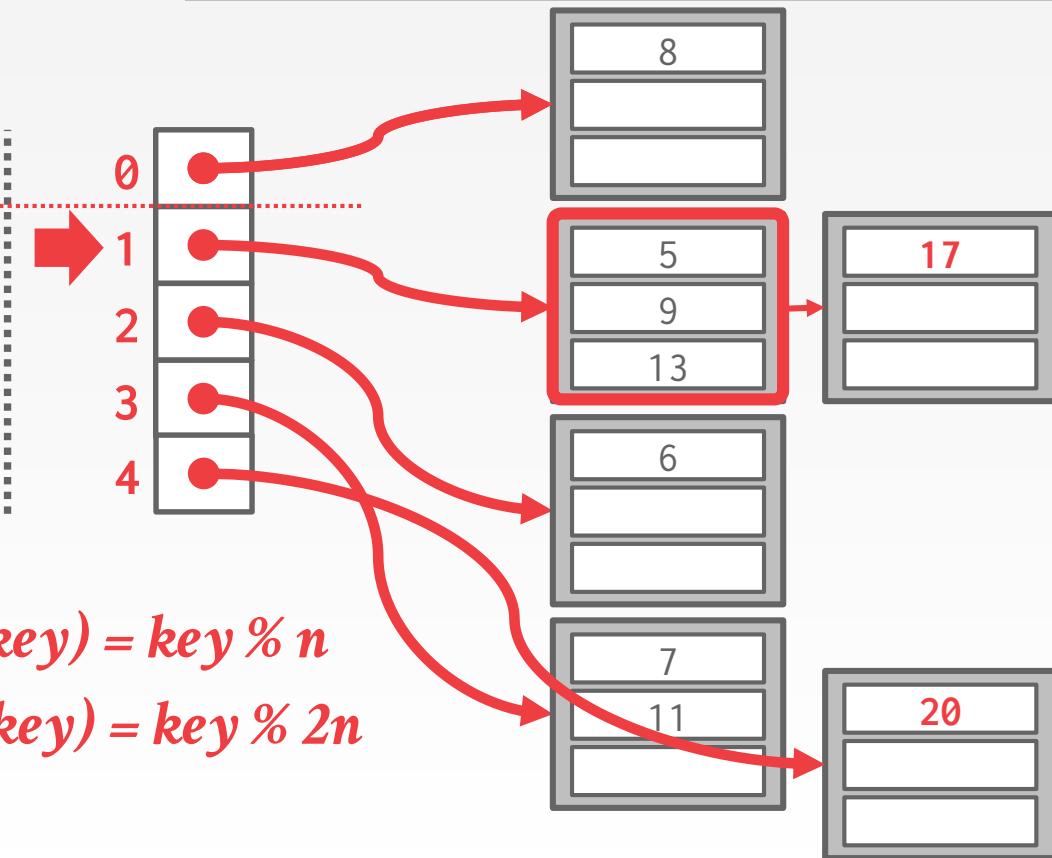
$$\text{hash}_2(20) = 20 \% 8 = 4$$

**Find 9**

$$\text{hash}_1(9) = 9 \% 4 = 1$$

# LINEAR HASHING

*Split  
Pointer*



**Find 6**

$$\text{hash}_1(6) = 6 \% 4 = 2$$

**Insert 17**

$$\text{hash}_1(17) = 17 \% 4 = 1$$

**Find 20**

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

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

**Find 9**

$$\text{hash}_1(9) = 9 \% 4 = 1$$

# LINEAR HASHING

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.

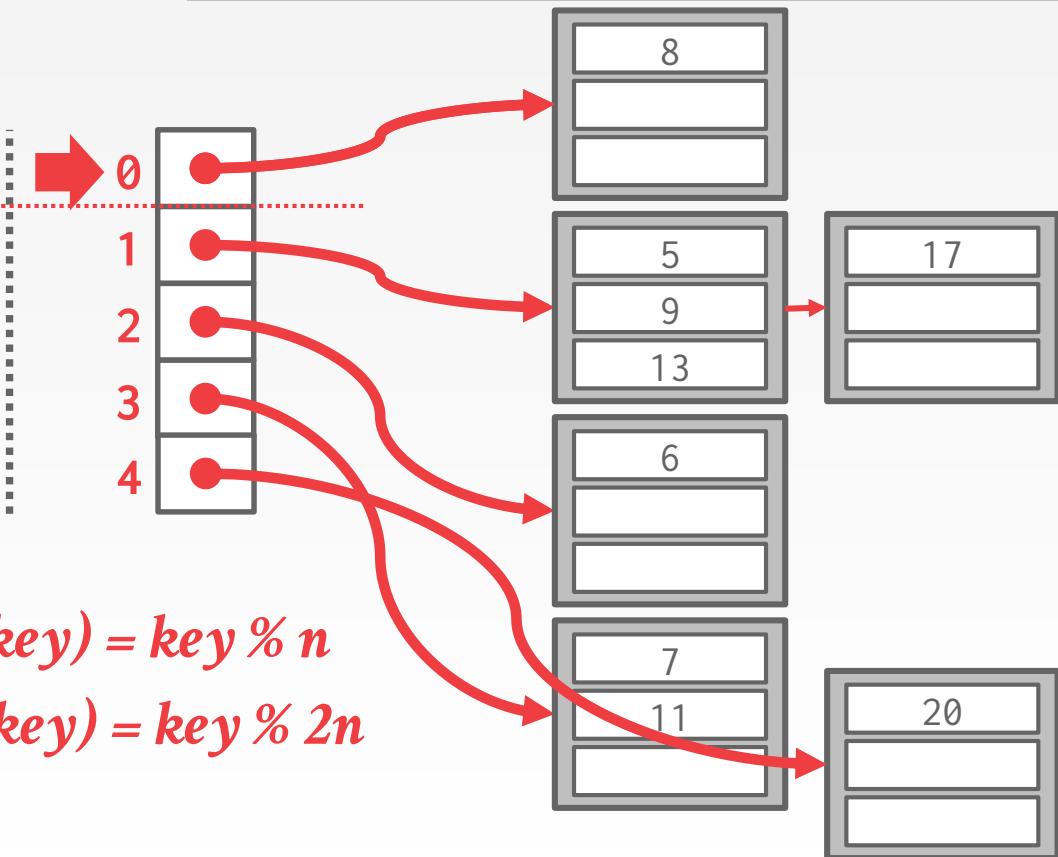
→ When the pointer reaches the last slot, delete the first hash function and move back to beginning.

The pointer can also move backwards when buckets are empty.

You split whatever the split pointer works at, no matter whether it was one that overflowed or not. Because eventually, like if this thing keeps on overflowing that will keep moving the split pointer down by 1 eventually I will get to everyone split.

# LINEAR HASHING – DELETES

*Split  
Pointer*



Delete 20

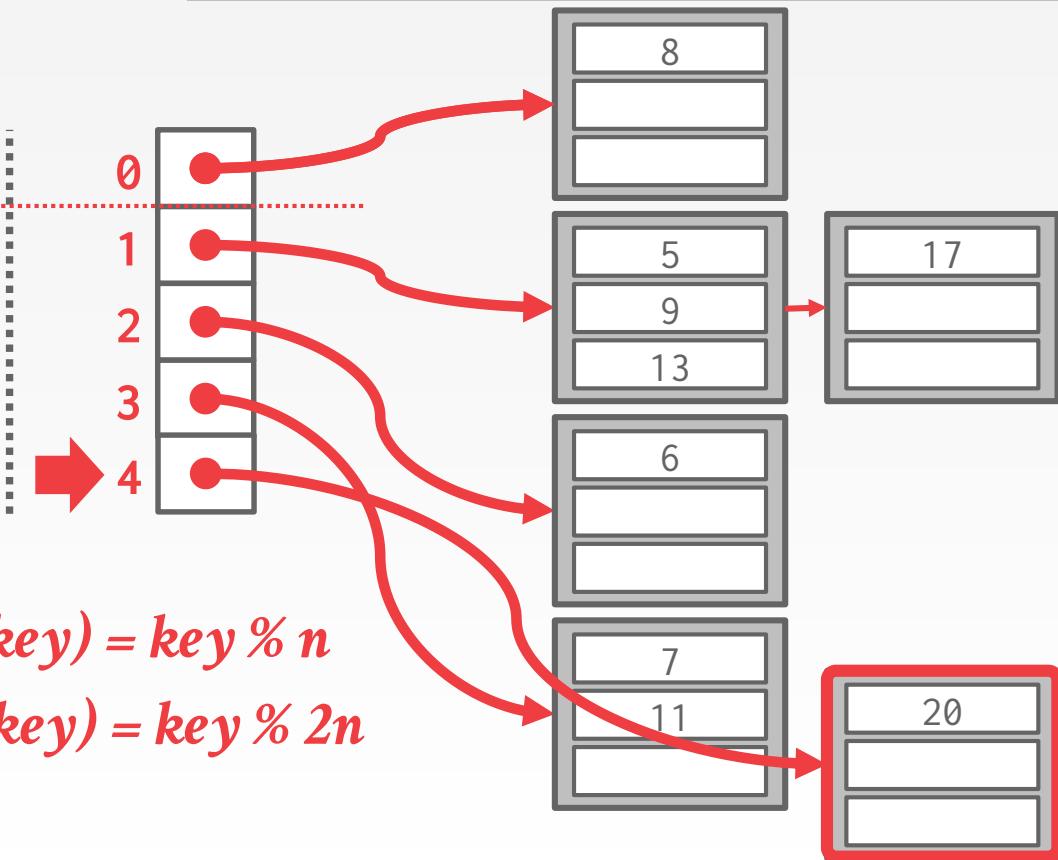
$$\text{hash}_1(20) = 20 \% 4 = 0$$

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

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

# LINEAR HASHING – DELETES

*Split  
Pointer*



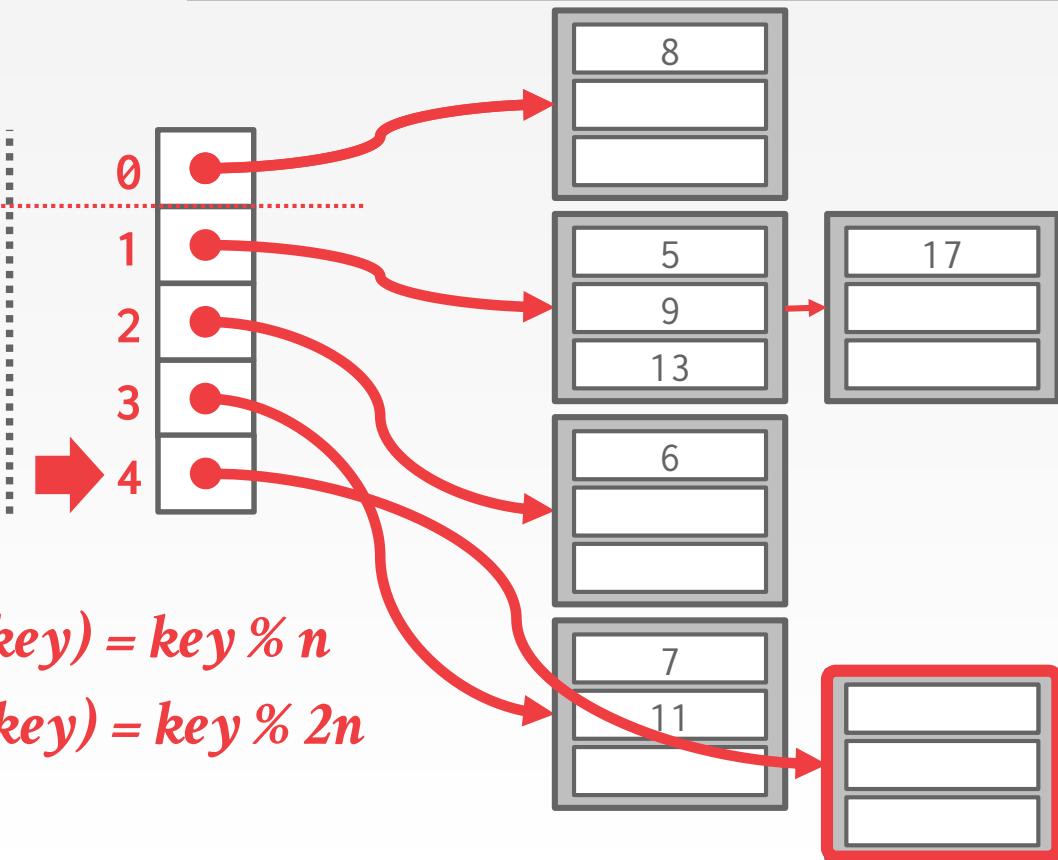
Delete 20

$$\begin{aligned} \text{hash}_1(20) &= 20 \% 4 = 0 \\ \text{hash}_2(20) &= 20 \% 8 = 4 \end{aligned}$$



# LINEAR HASHING – DELETES

*Split  
Pointer*



Delete 20

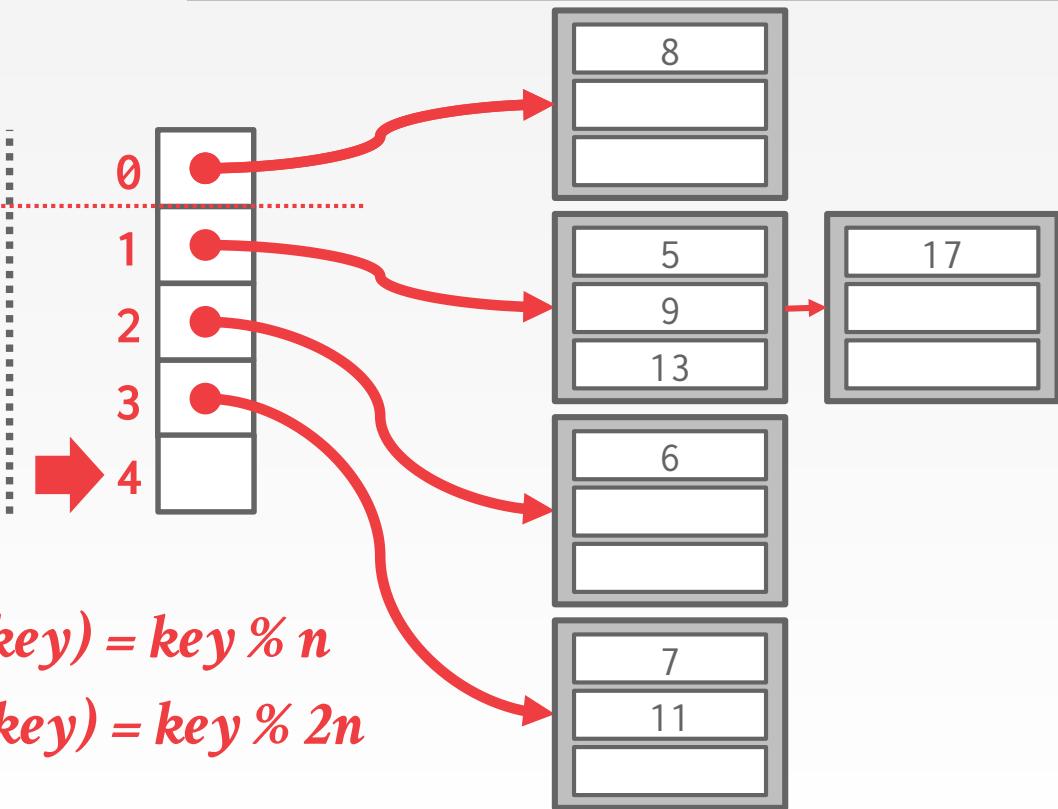
$$\text{hash}_1(20) = 20 \% 4 = 0$$

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



# LINEAR HASHING – DELETES

*Split  
Pointer*



Delete 20

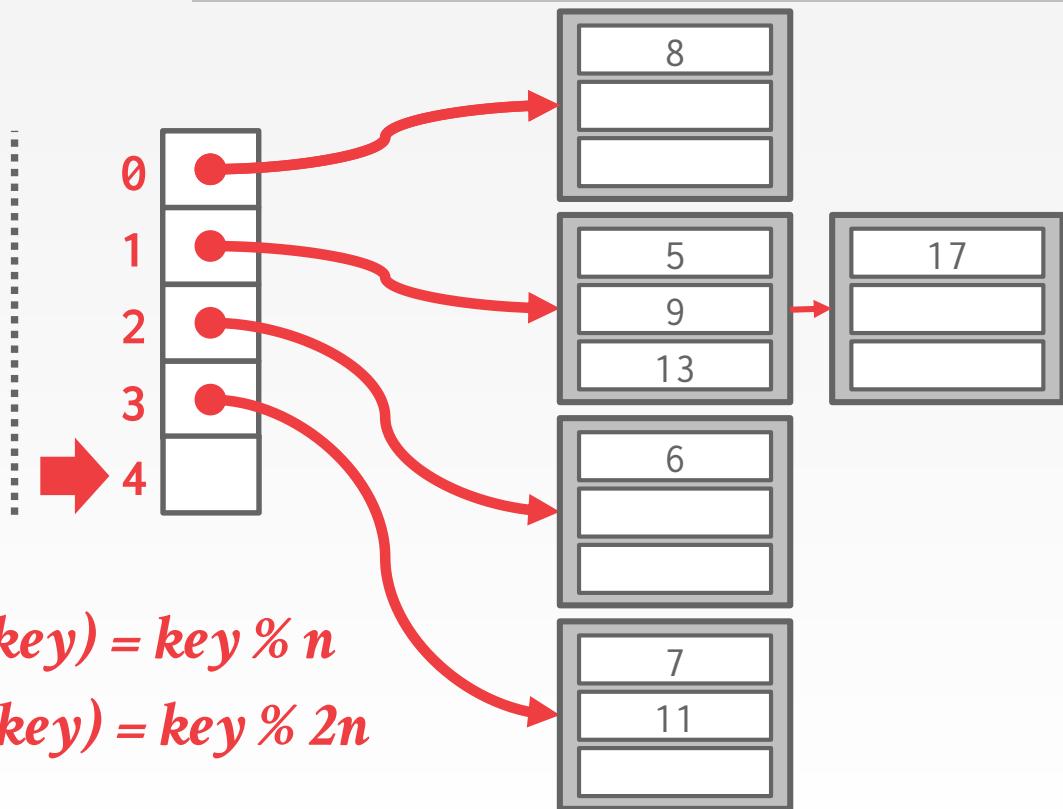
$$\text{hash}_1(20) = 20 \% 4 = 0$$

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



# LINEAR HASHING – DELETES

*Split  
Pointer*



Delete 20

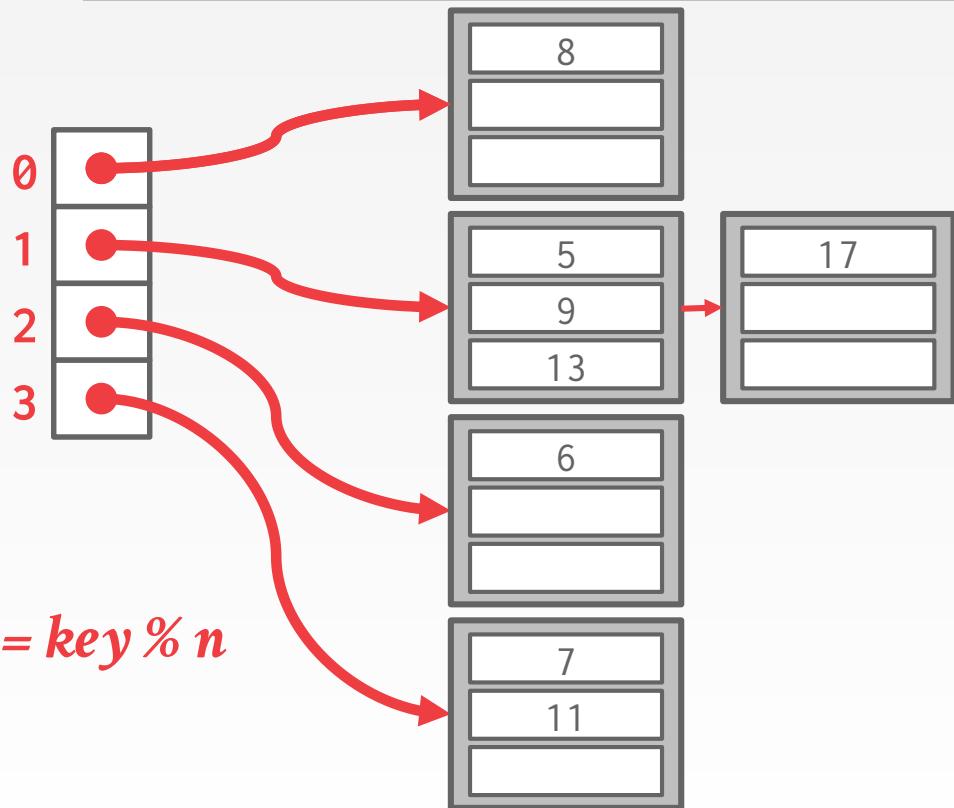
$$\text{hash}_1(20) = 20 \% 4 = 0$$

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



# LINEAR HASHING – DELETES

*Split  
Pointer*



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

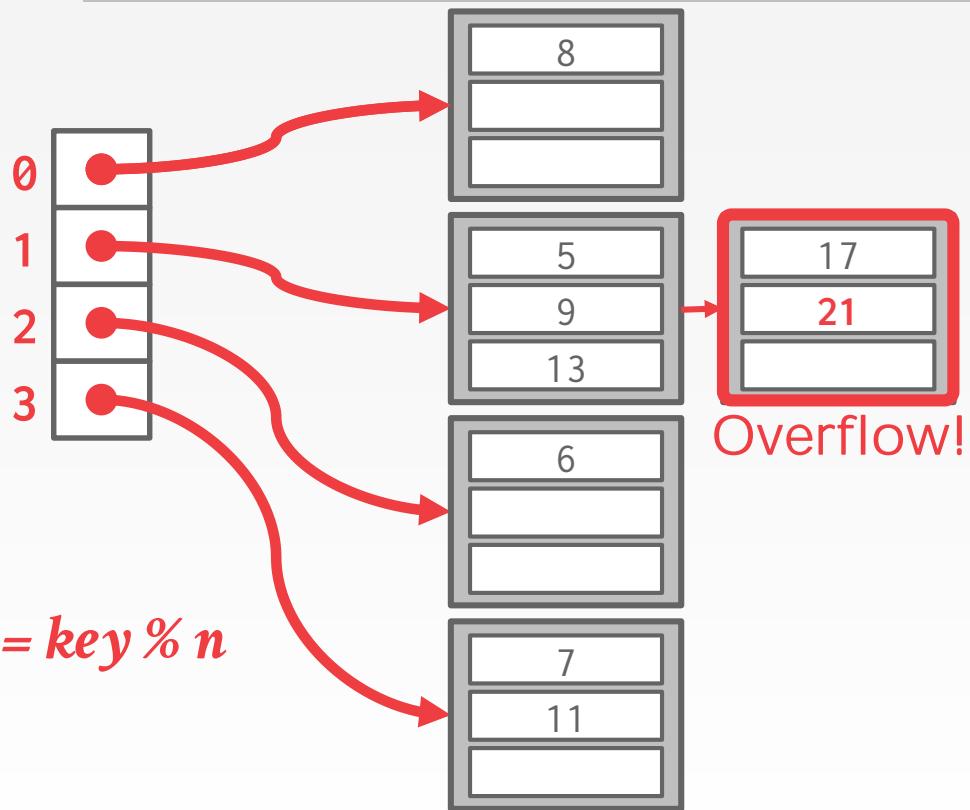
Delete 20

$$\begin{aligned}\text{hash}_1(20) &= 20 \% 4 = 0 \\ \text{hash}_2(20) &= 20 \% 8 = 4\end{aligned}$$



# LINEAR HASHING – DELETES

*Split  
Pointer*



Delete 20

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

$$hash_2(20) = 20 \% 8 = 4$$

Insert 21

$$hash_1(21) = 21 \% 4 = 1$$

# 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!"

