Intro to Database Systems (15-445/645)

07 Hash Tables

Carnegie Mellon University

FALL 2022

Andy Pavlo

ADMINISTRIVIA

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

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.

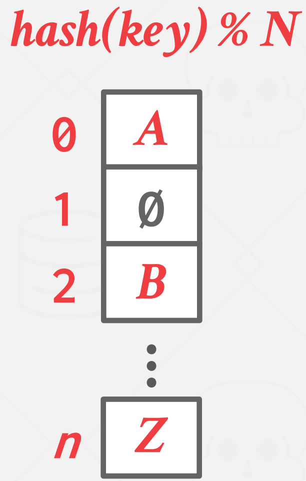
It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.

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.

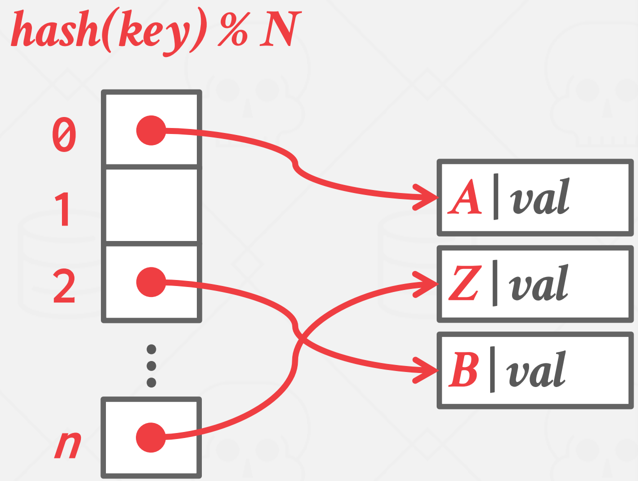
To find an entry, mod the key by the number of elements to find the offset in the array.



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.



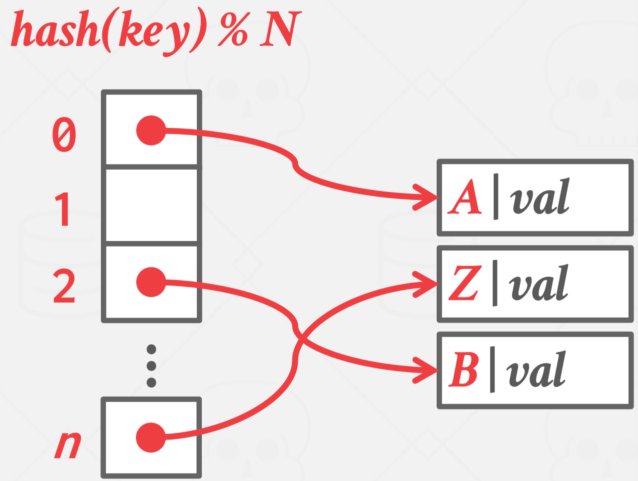
ASSUMPTIONS

Assumption#1: Number of elements is known ahead of time and fixed.

Assumption #2: Each key is unique.

Assumption #3: 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 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).

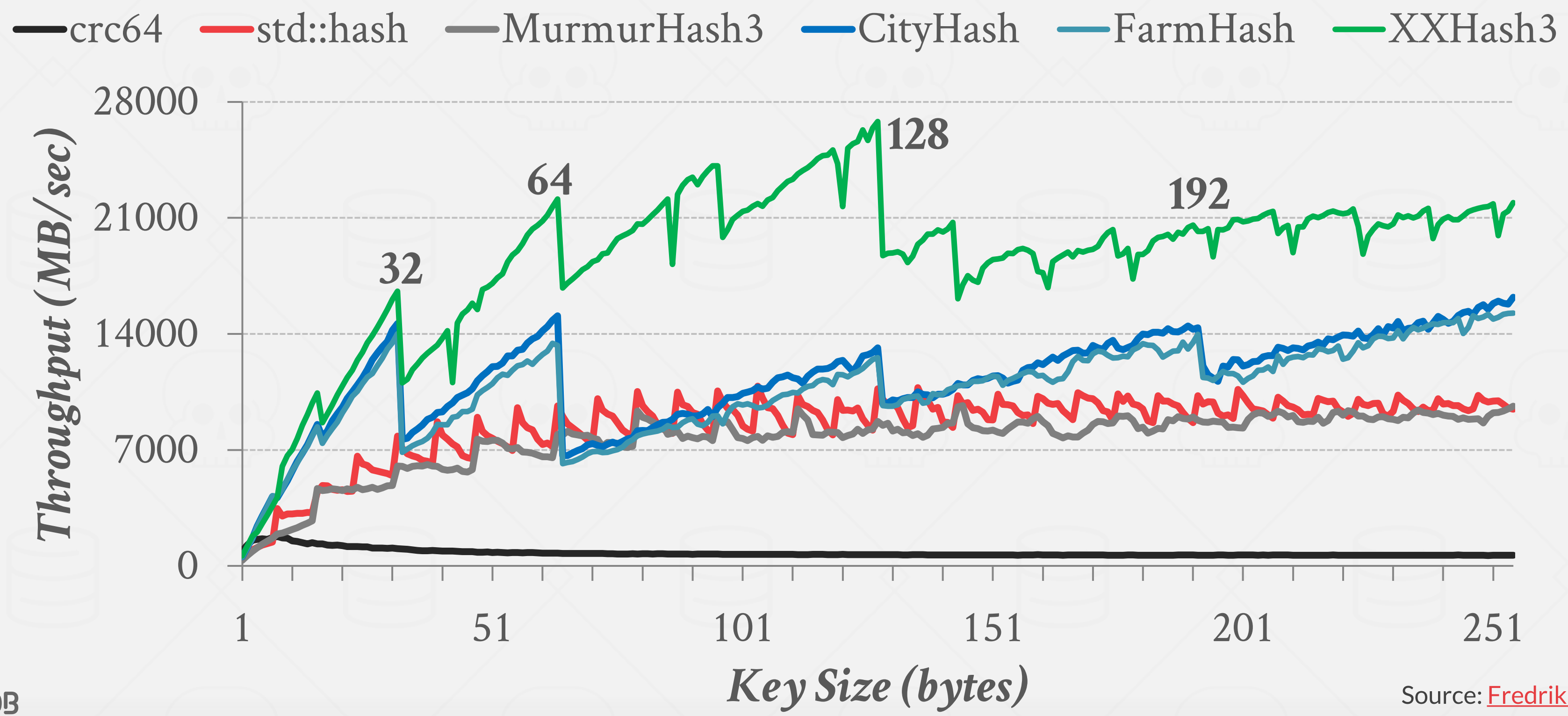
Facebook XXHash (2012)
→ From the creator of zstd compression.

← State-of-the-art

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

HASH FUNCTION BENCHMARK

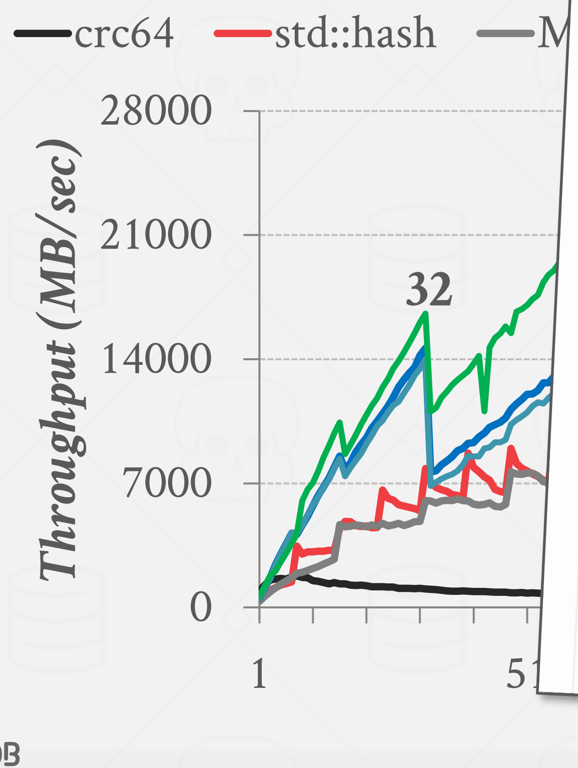
Intel Core i7-8700K @ 3.70GHz



Source: Fredrik Widlund

HASH FU

Inte



rurban / **smhasher** Public

Sponsor | Notifications | Fork 135 | Star 1.1k

Code | Issues 26 | Pull requests 1 | Actions | Projects | Wiki | Security | Insights

master ▾ **smhasher** / **README.md**

Go to file | ...

rurban results of fwojcik's fixes ... ✕

Latest commit 9a5e665 on Aug 19 | History

7 contributors

`444 lines (395 sloc) | 37.4 KB`

Raw | Blame

SMhasher

build passing | build failing | build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
donothing32	15316474.36	6.00	-	13	bad seed 0, test NOP
donothing64	15330019.19	6.00	-	13	bad seed 0, test NOP
donothing128	15278983.09	6.00	-	13	bad seed 0, test NOP
NOP_OAAT_read64	28467.50	18.48	-	47	test NOP
BadHash	524.81	96.20	-	47	bad seed 0, test FAIL
sumhash	7169.08	27.12	-	363	bad seed 0, test FAIL
sumhash32	22556.18	22.98	-	863	UB, test FAIL
multiply_shift	5418.36	28.69	157.11 (3)	345	bad seeds & 0xfffffff0, fails most tests
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fails most tests

Source: Fredrik Widlund

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

Approach #3: Cuckoo Hashing

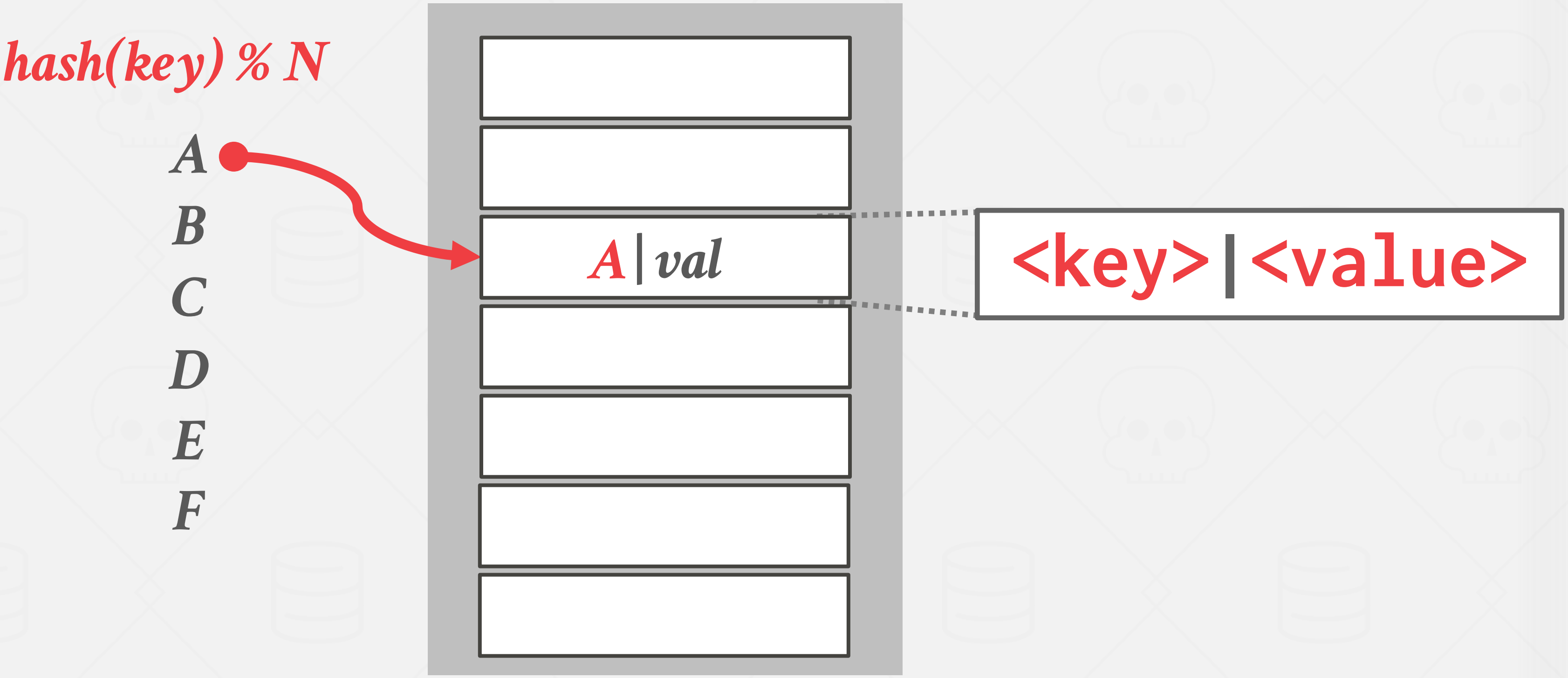
LINEAR PROBE HASHING

Single giant table of slots.

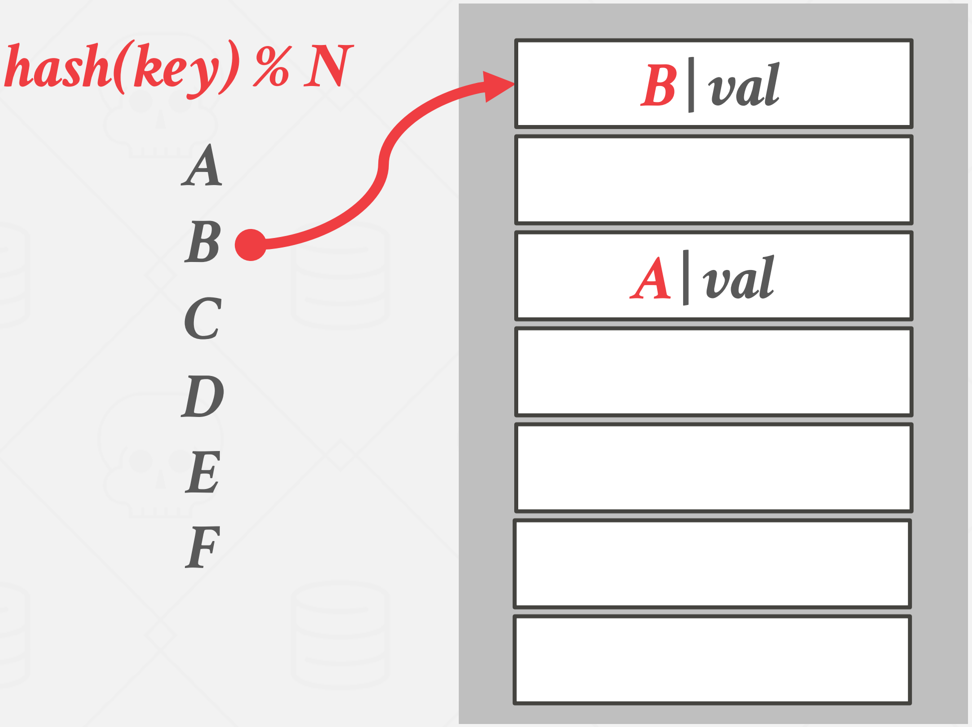
Resolve collisions by linearly searching for the next free slot in the table.

→ 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

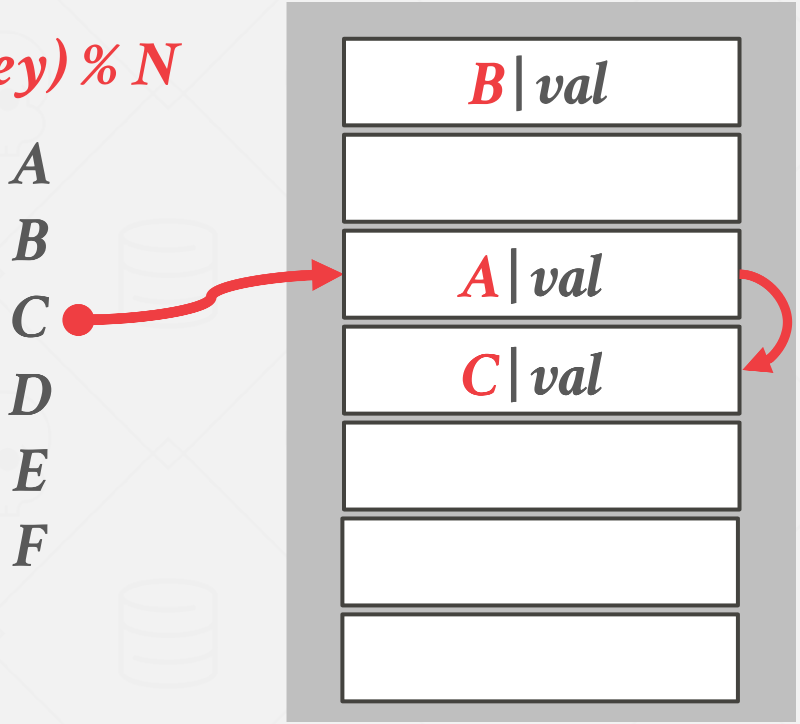


LINEAR PROBE HASHING



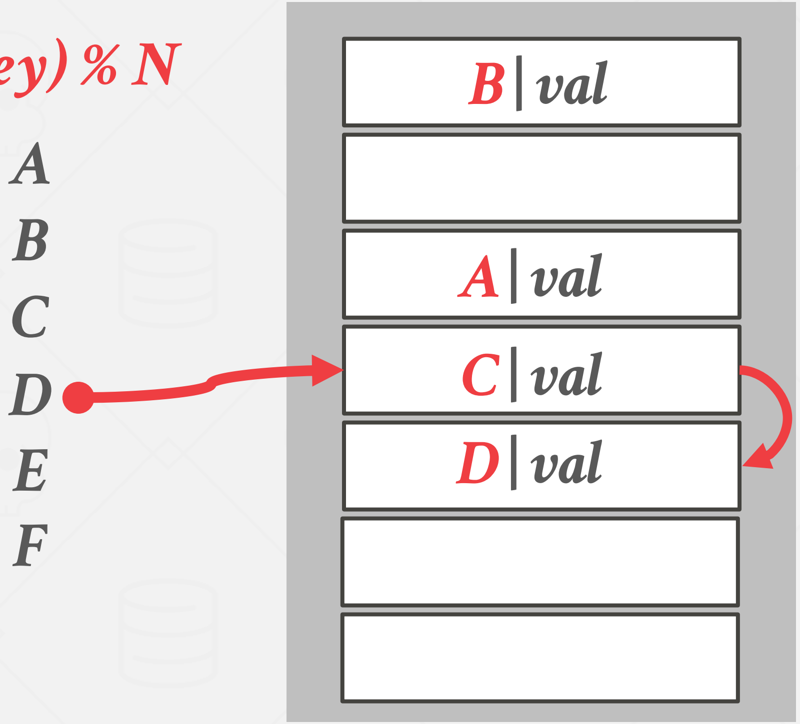
LINEAR PROBE HASHING

hash(key) % N

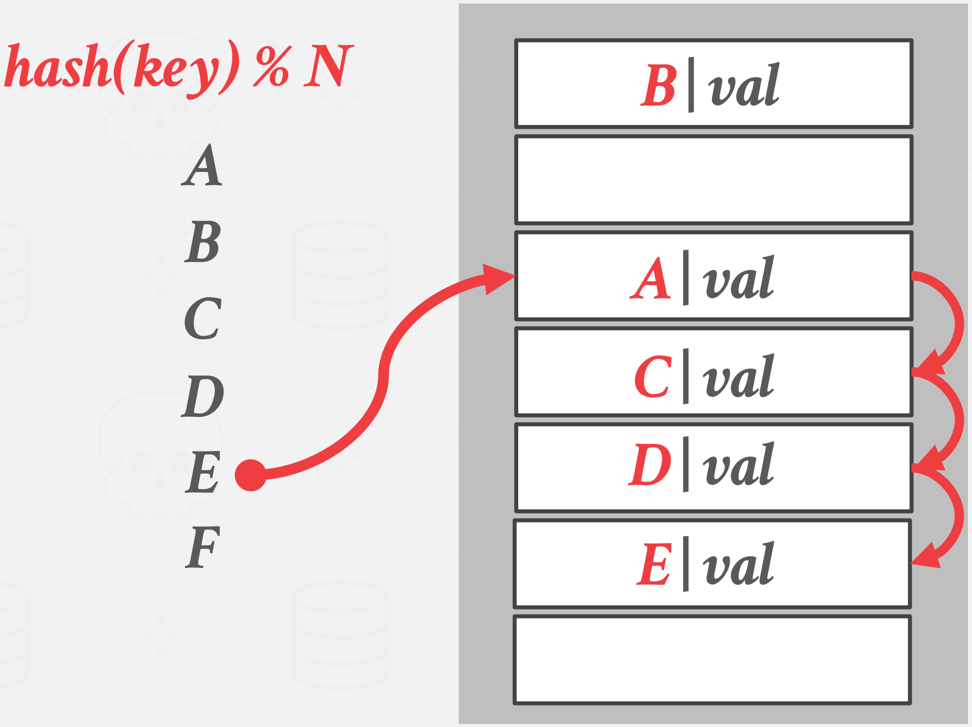


LINEAR PROBE HASHING

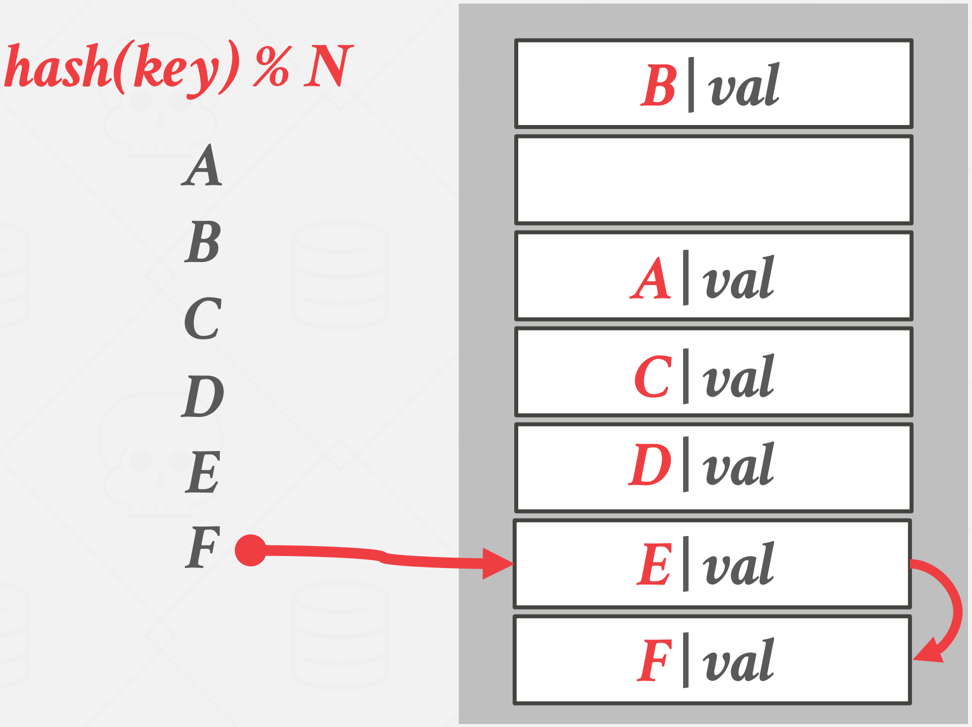
hash(key) % N



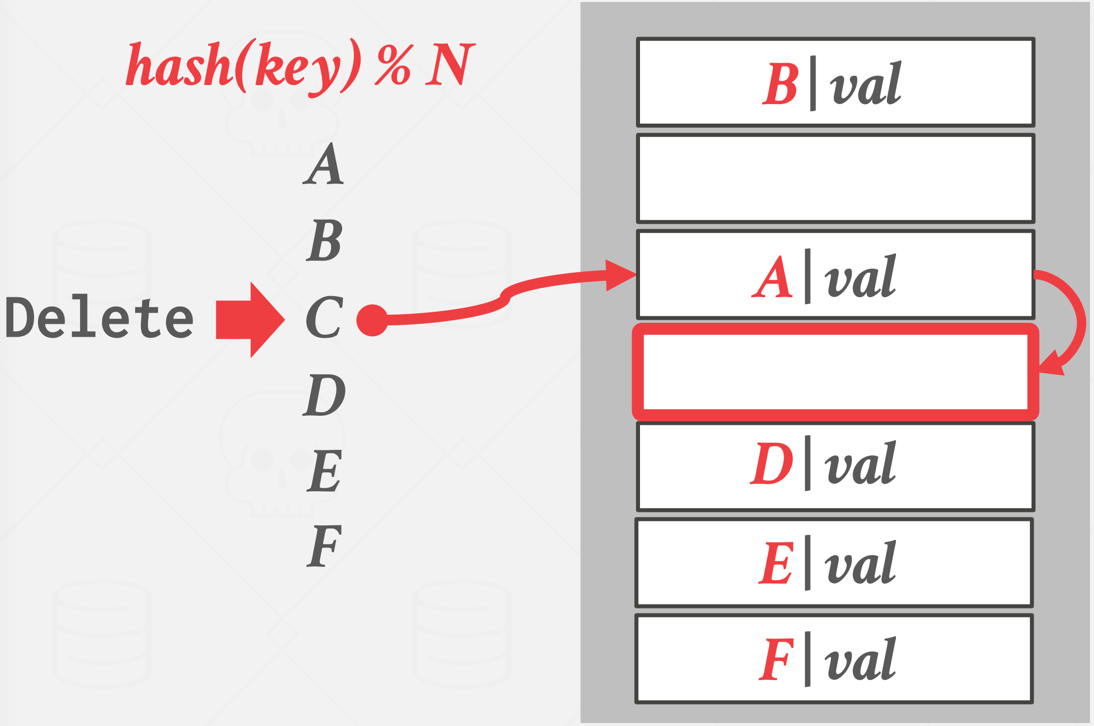
LINEAR PROBE HASHING



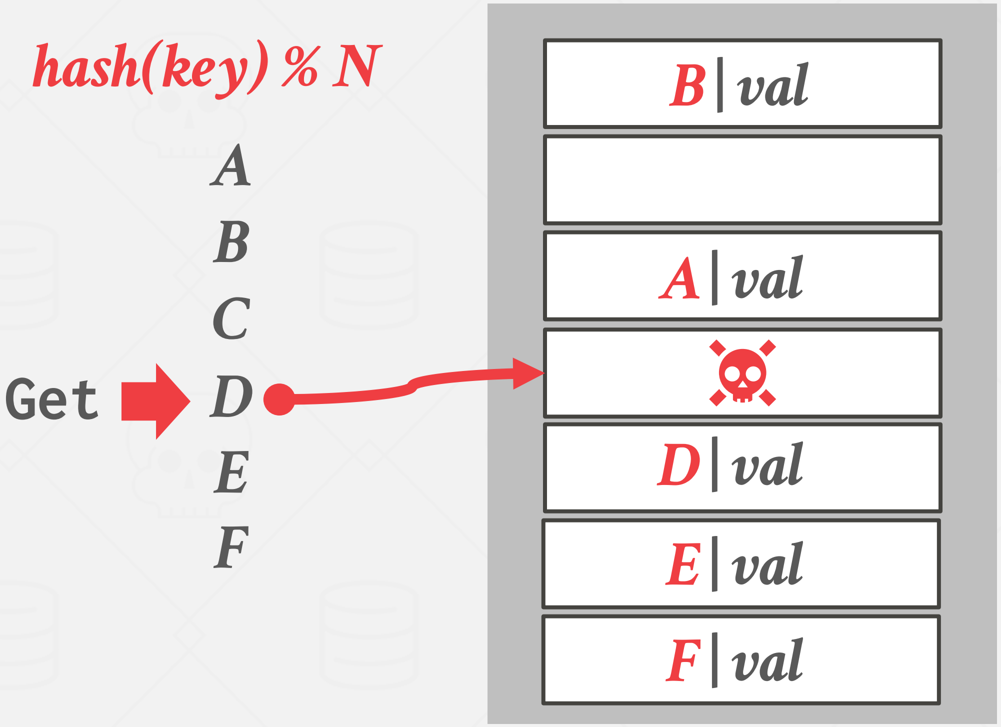
LINEAR PROBE HASHING



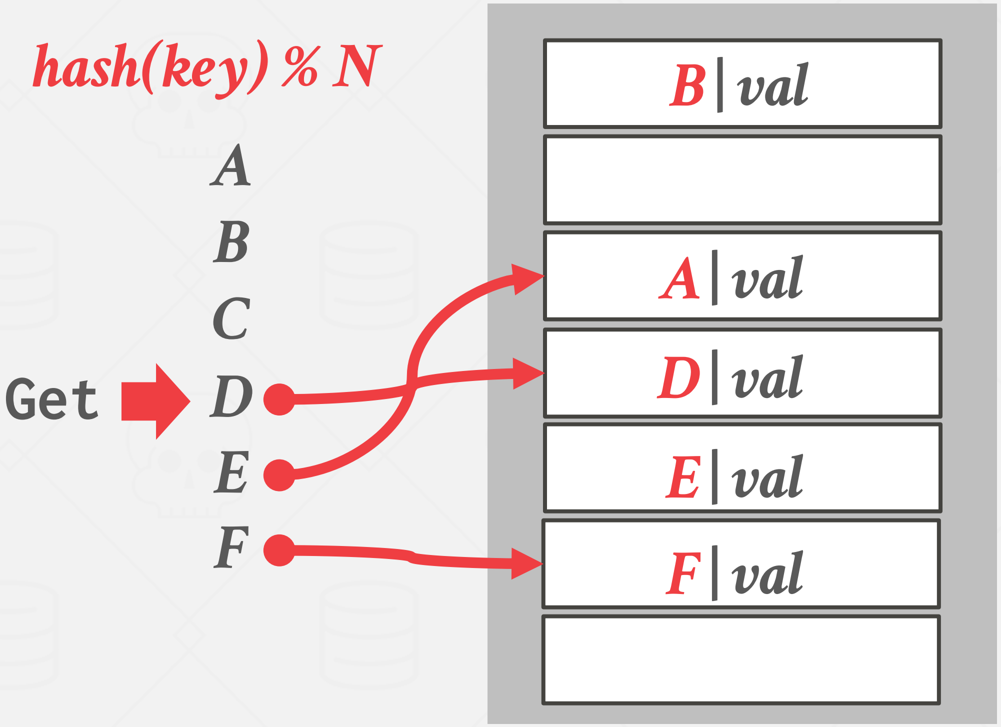
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES

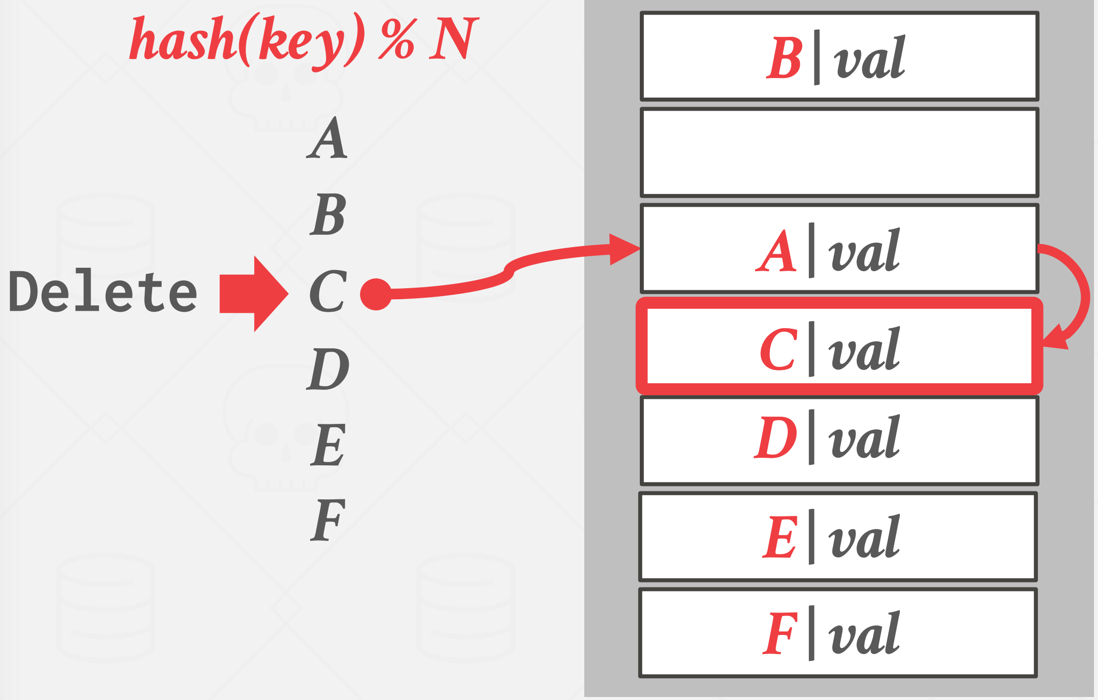


Approach #1: Movement

→ Rehash keys until you find the first empty slot.

→ Nobody actually does this.

LINEAR PROBE HASHING – DELETES



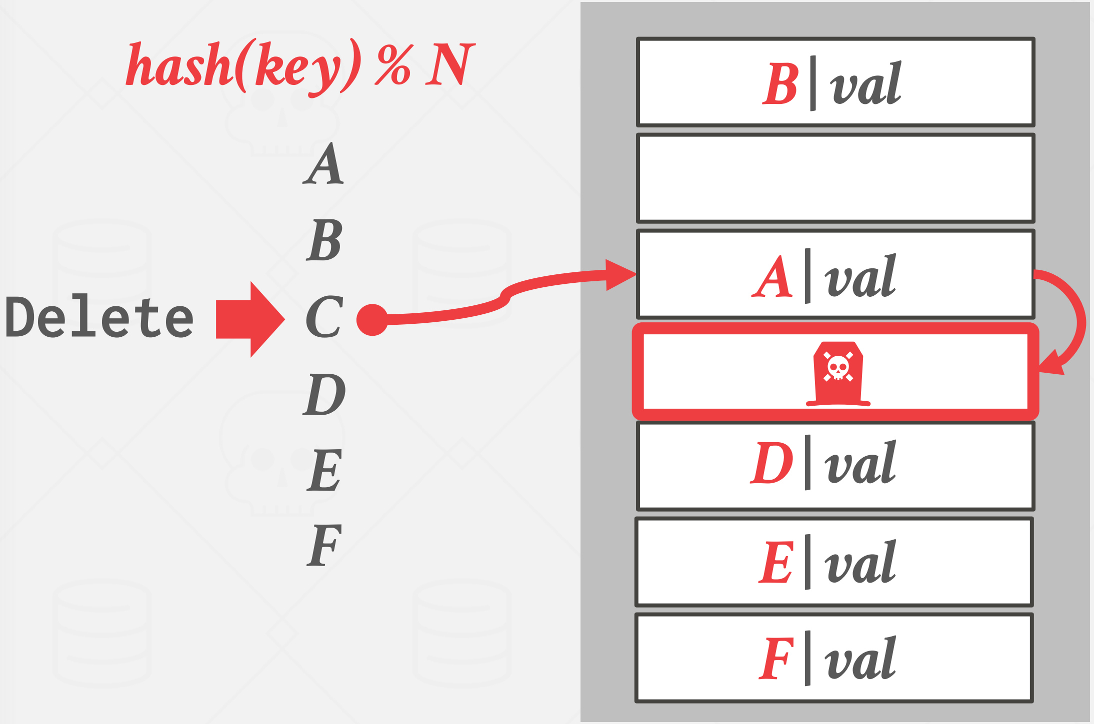
Approach #2: Tombstone

→ 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.

LINEAR PROBE HASHING – DELETES



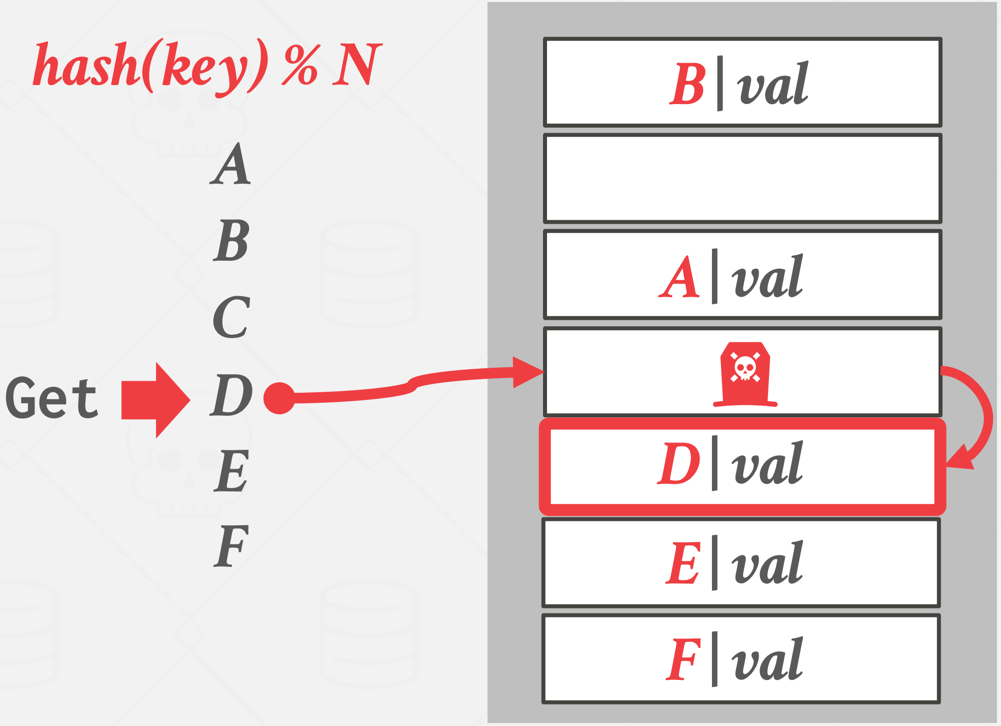
Approach #2: Tombstone

→ 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.

LINEAR PROBE HASHING – DELETES



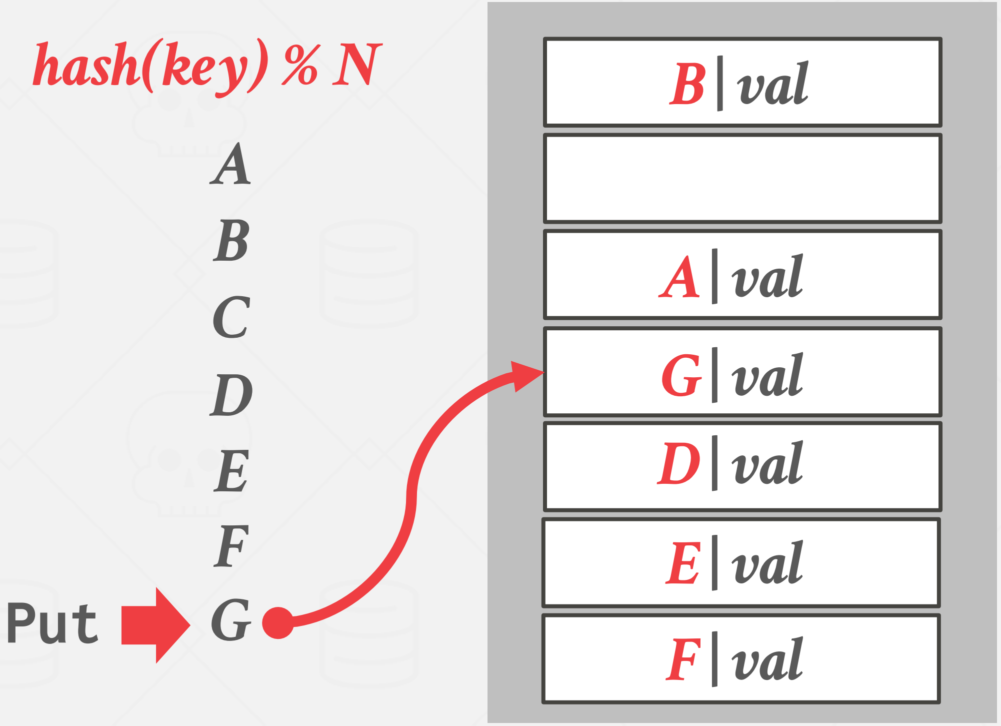
Approach #2: Tombstone

→ 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.

LINEAR PROBE HASHING – DELETES



Approach #2: Tombstone

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

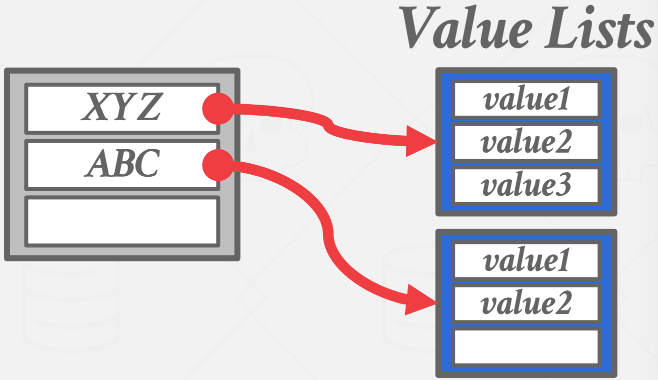
→ You can reuse the slot for new keys.

→ May need periodic garbage collection.

NON-UNIQUE KEYS

Choice #1: Separate Linked List

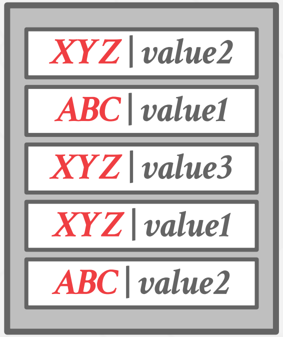
→ Store values in separate storage area for each key.



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.



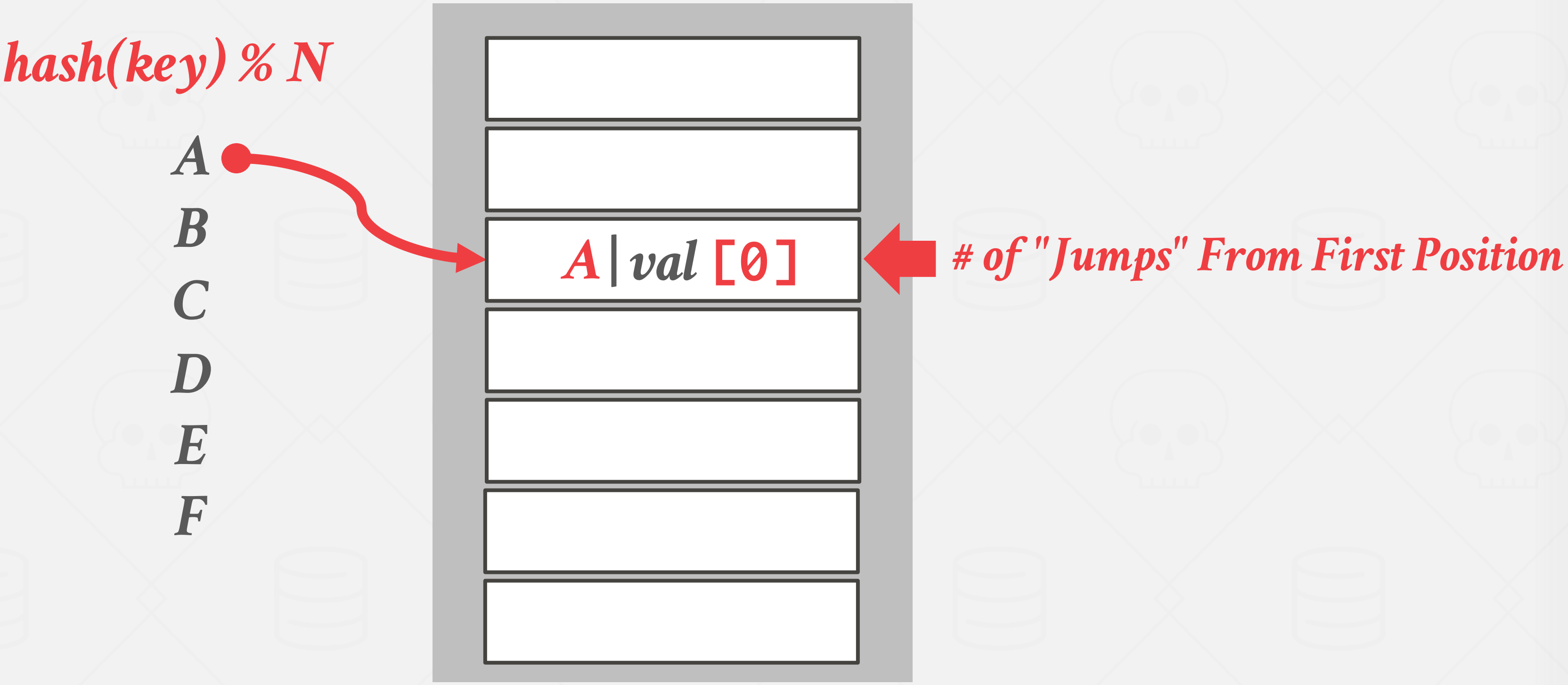
ROBIN HOOD HASHING

Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.

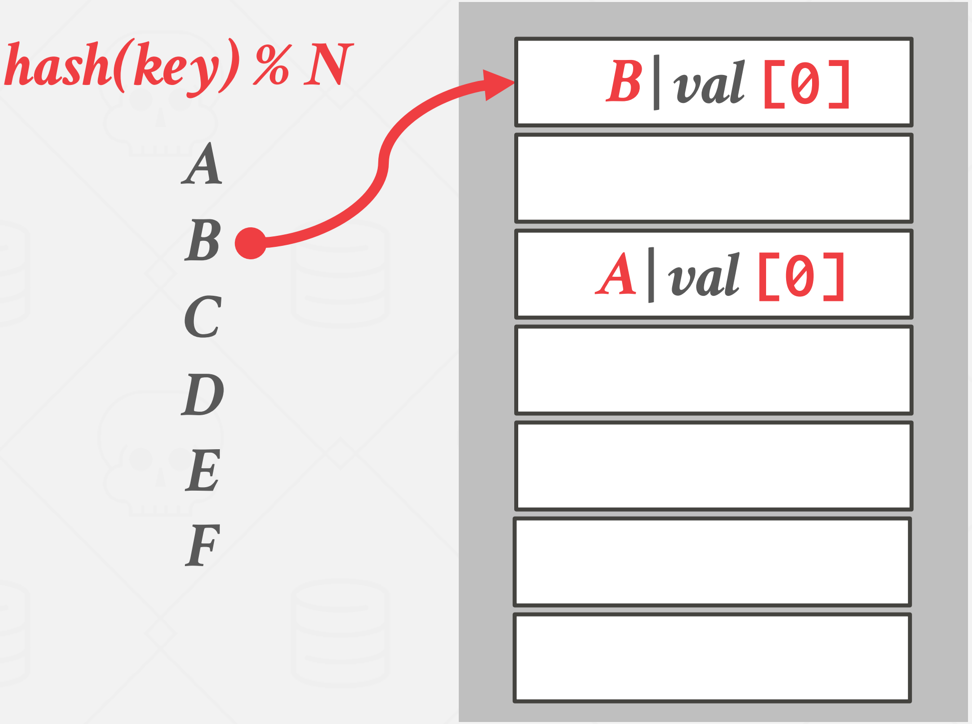
→ Each key tracks the number of positions they are from where its optimal position in the table.

→ 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.

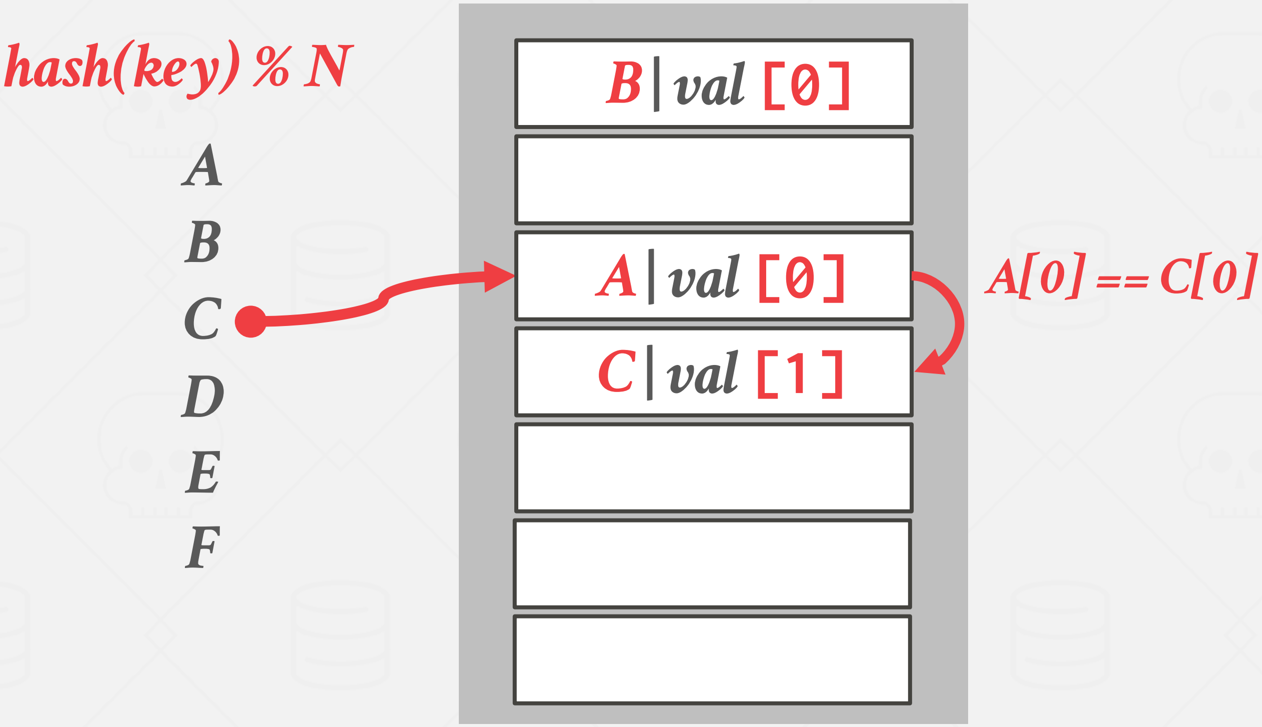
ROBIN HOOD HASHING



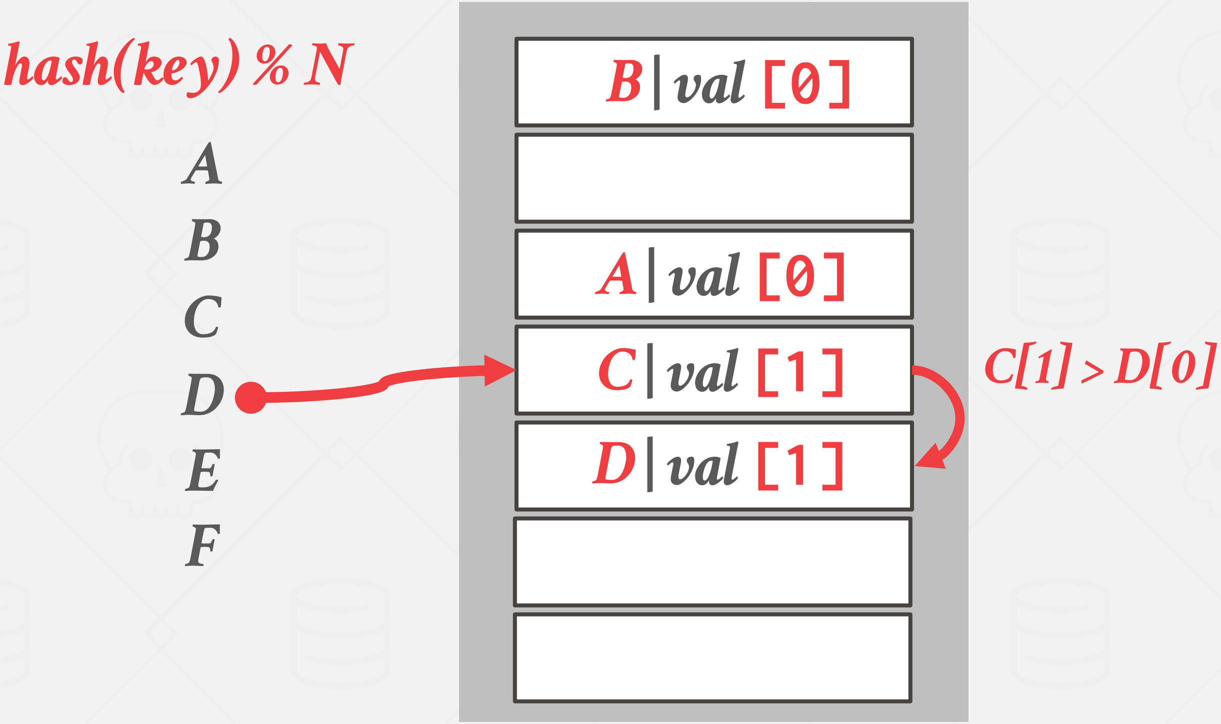
ROBIN HOOD HASHING



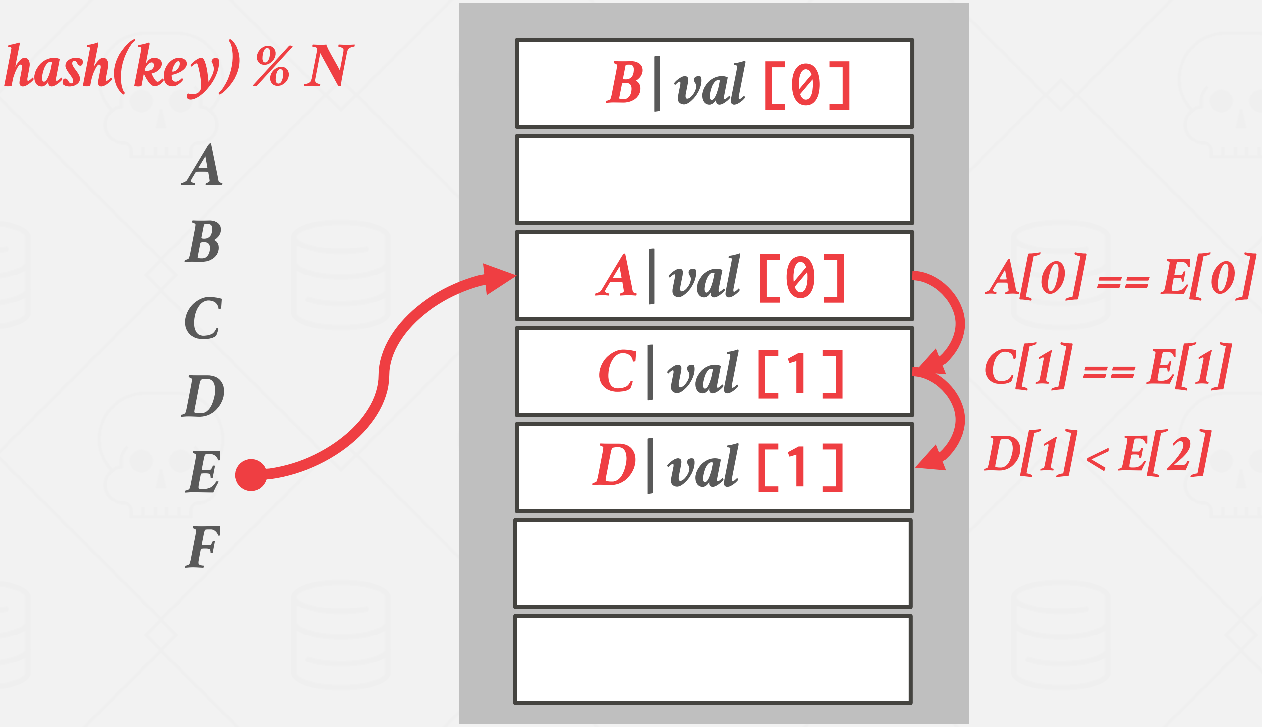
ROBIN HOOD HASHING



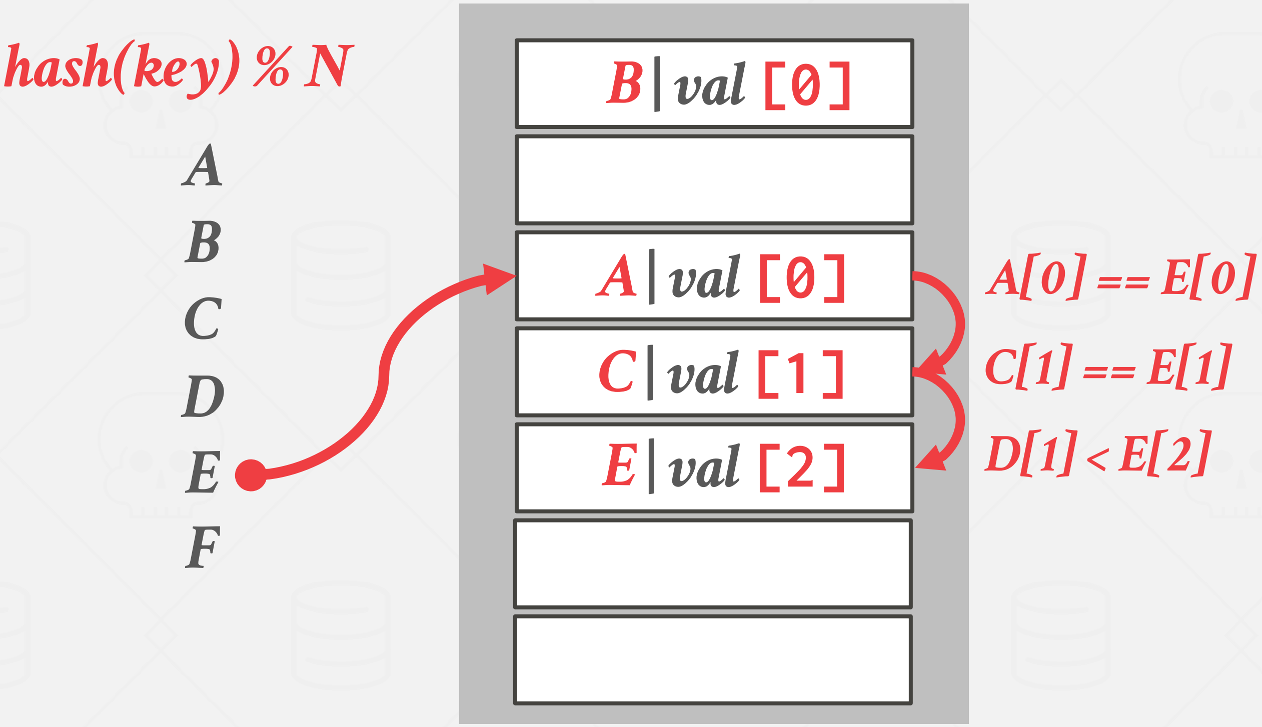
ROBIN HOOD HASHING



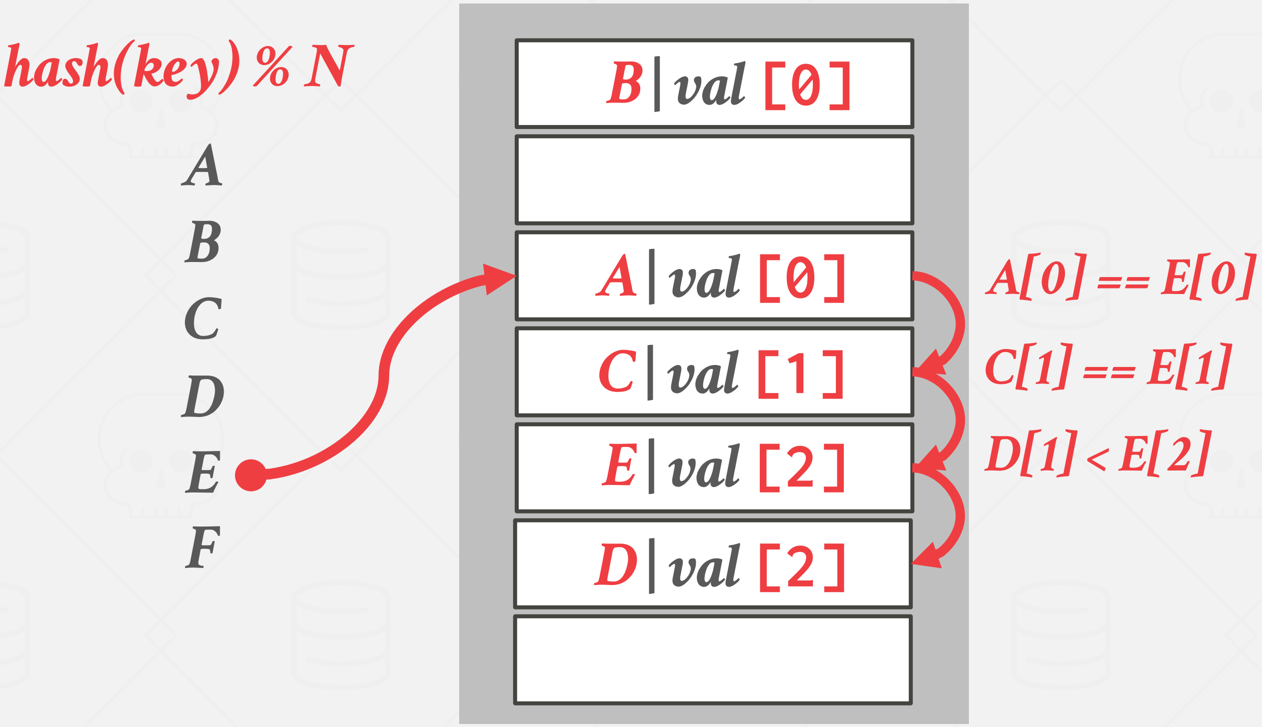
ROBIN HOOD HASHING



ROBIN HOOD HASHING

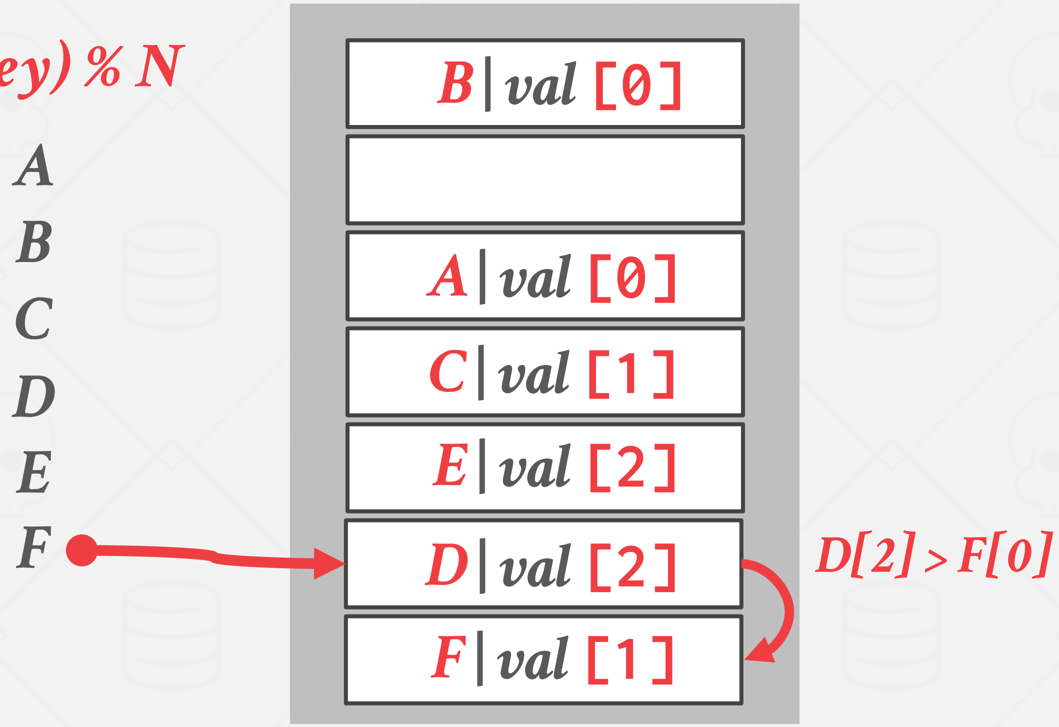


ROBIN HOOD HASHING



ROBIN HOOD HASHING

hash(key) % N



CUCKOO HASHING

Use multiple hash tables with different hash function seeds.

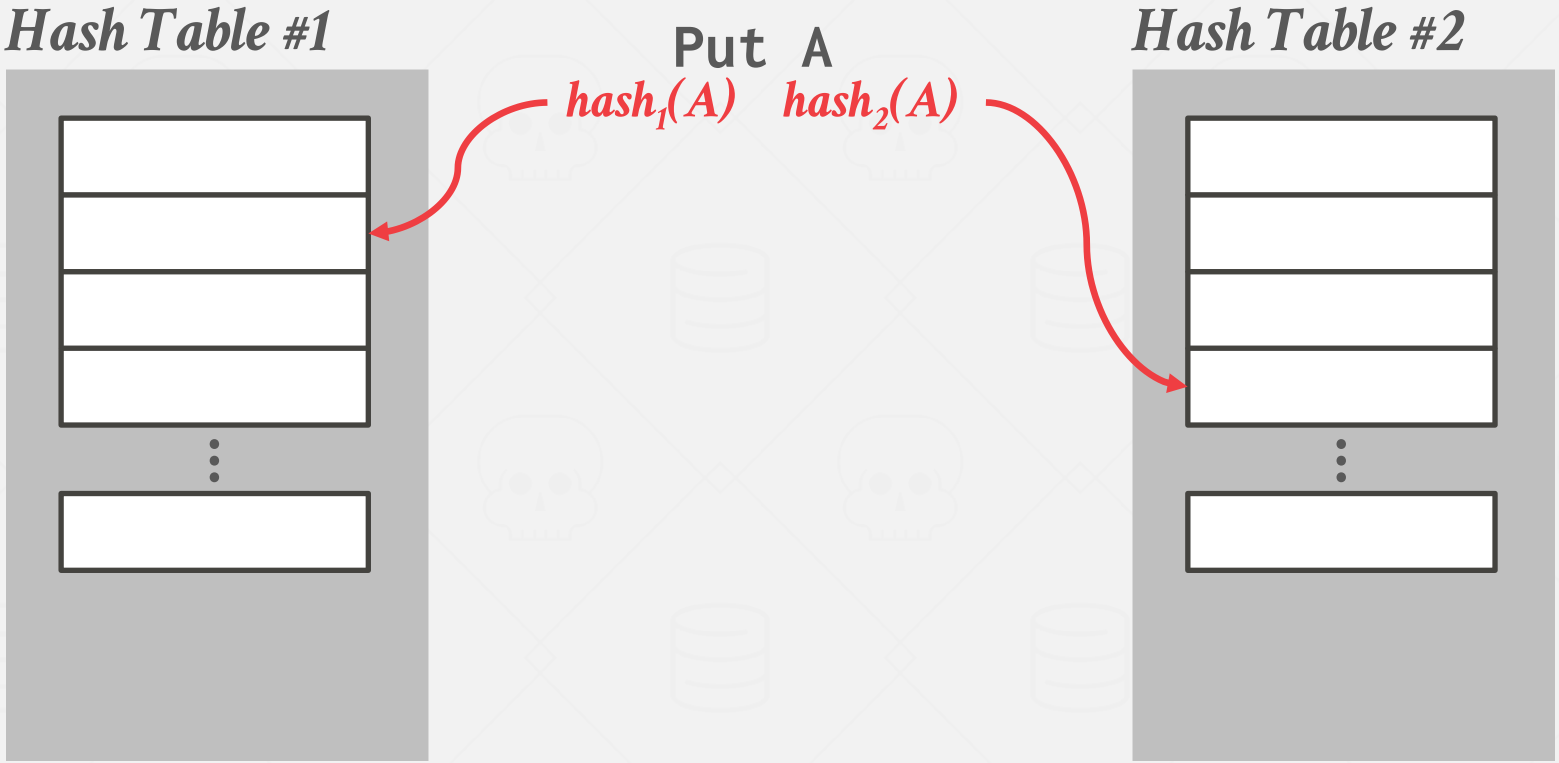
→ On insert, check every table and pick anyone that has a free slot.

→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

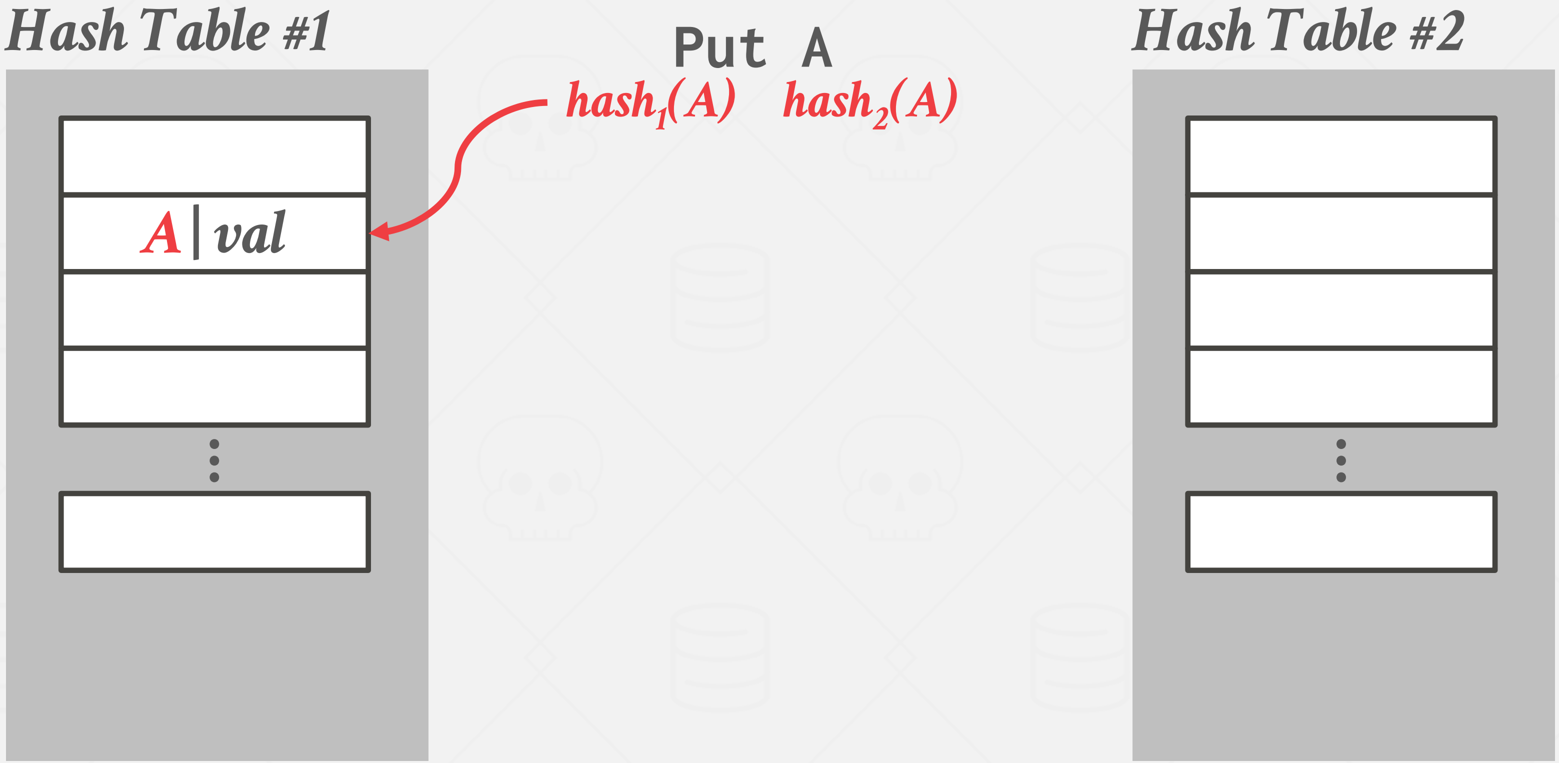
Look-ups and deletions are always **O(1)** because only one location per hash table is checked.

Best open-source implementation is from CMU.

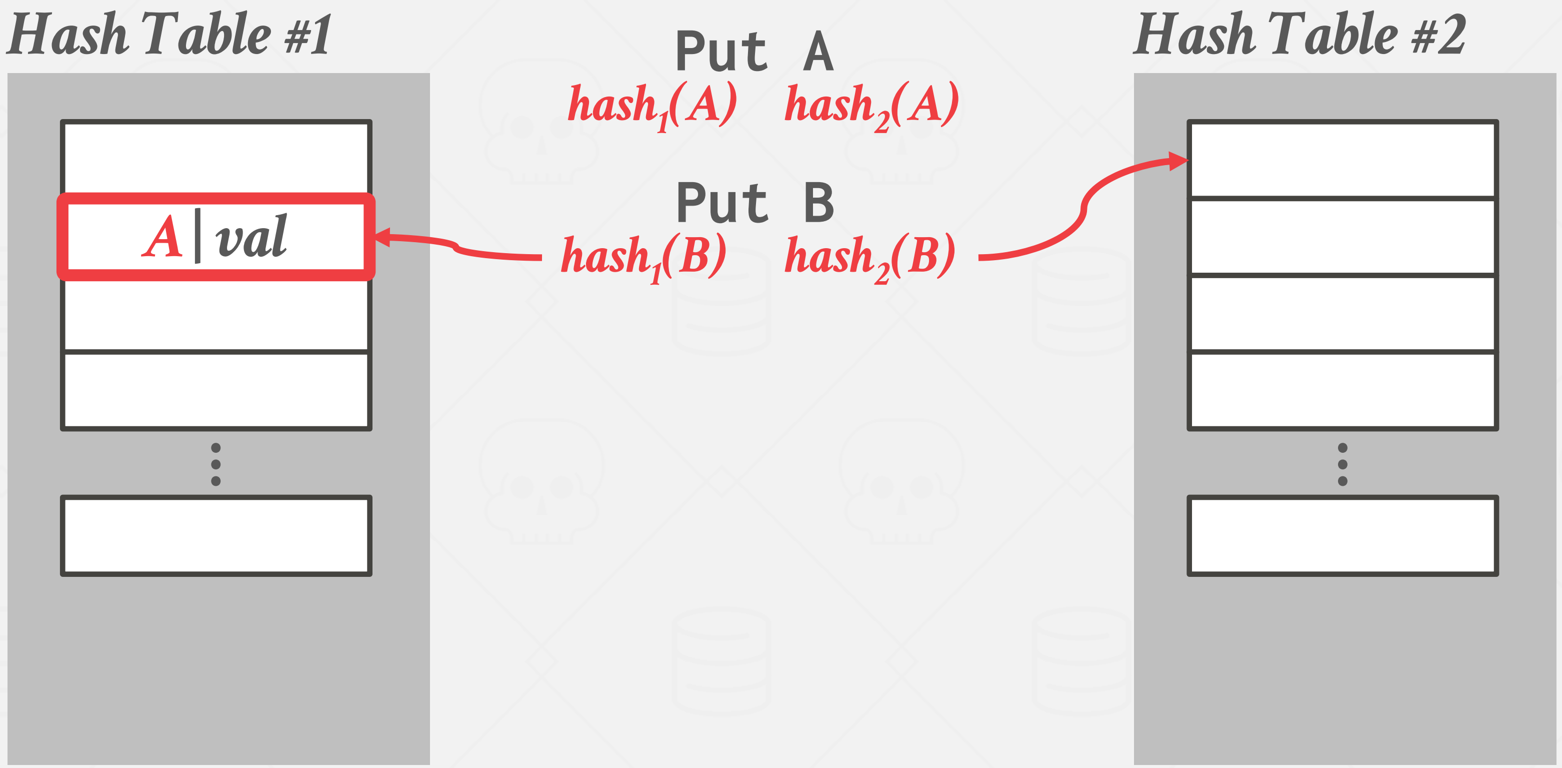
CUCKOO HASHING



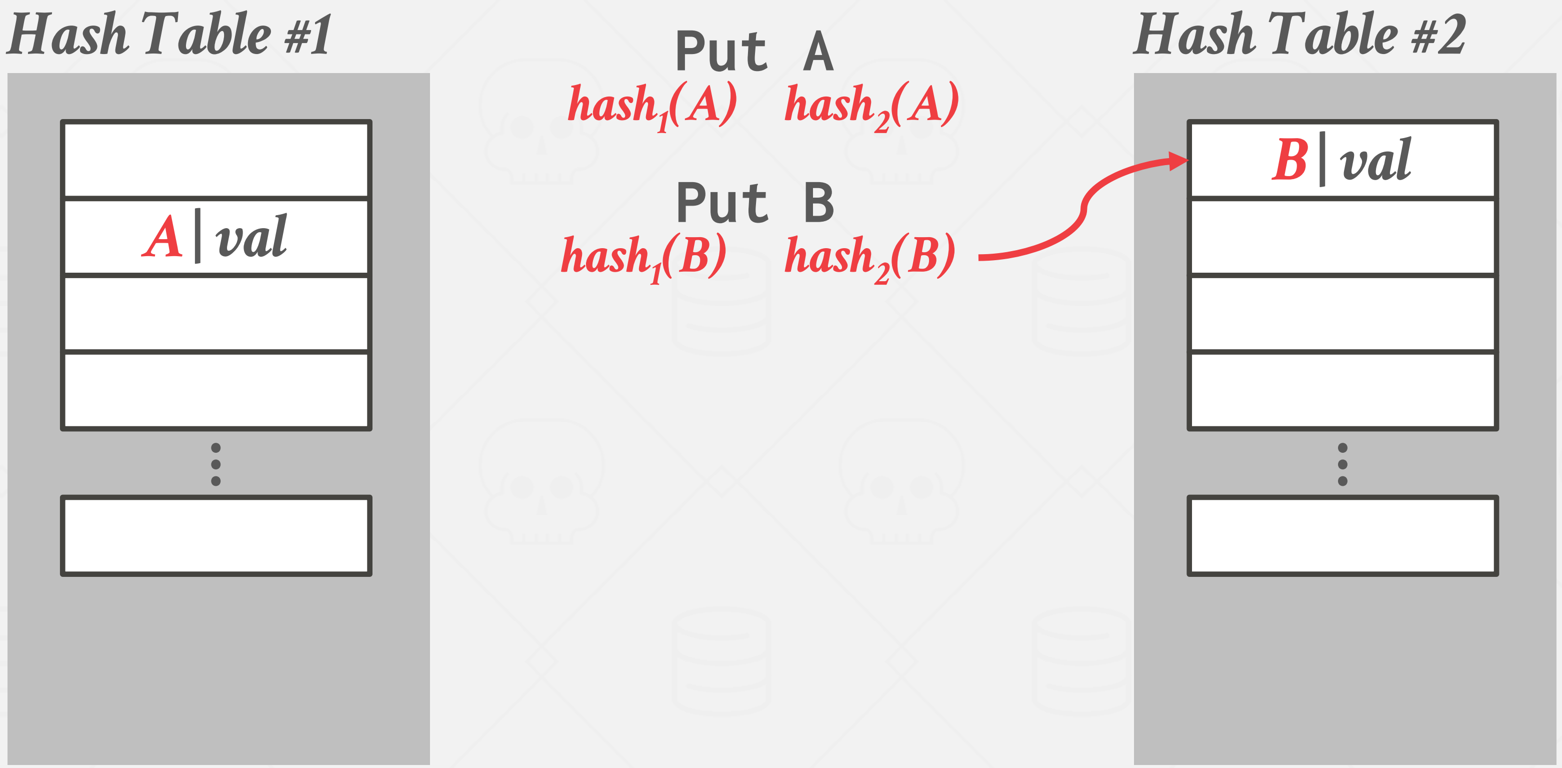
CUCKOO HASHING



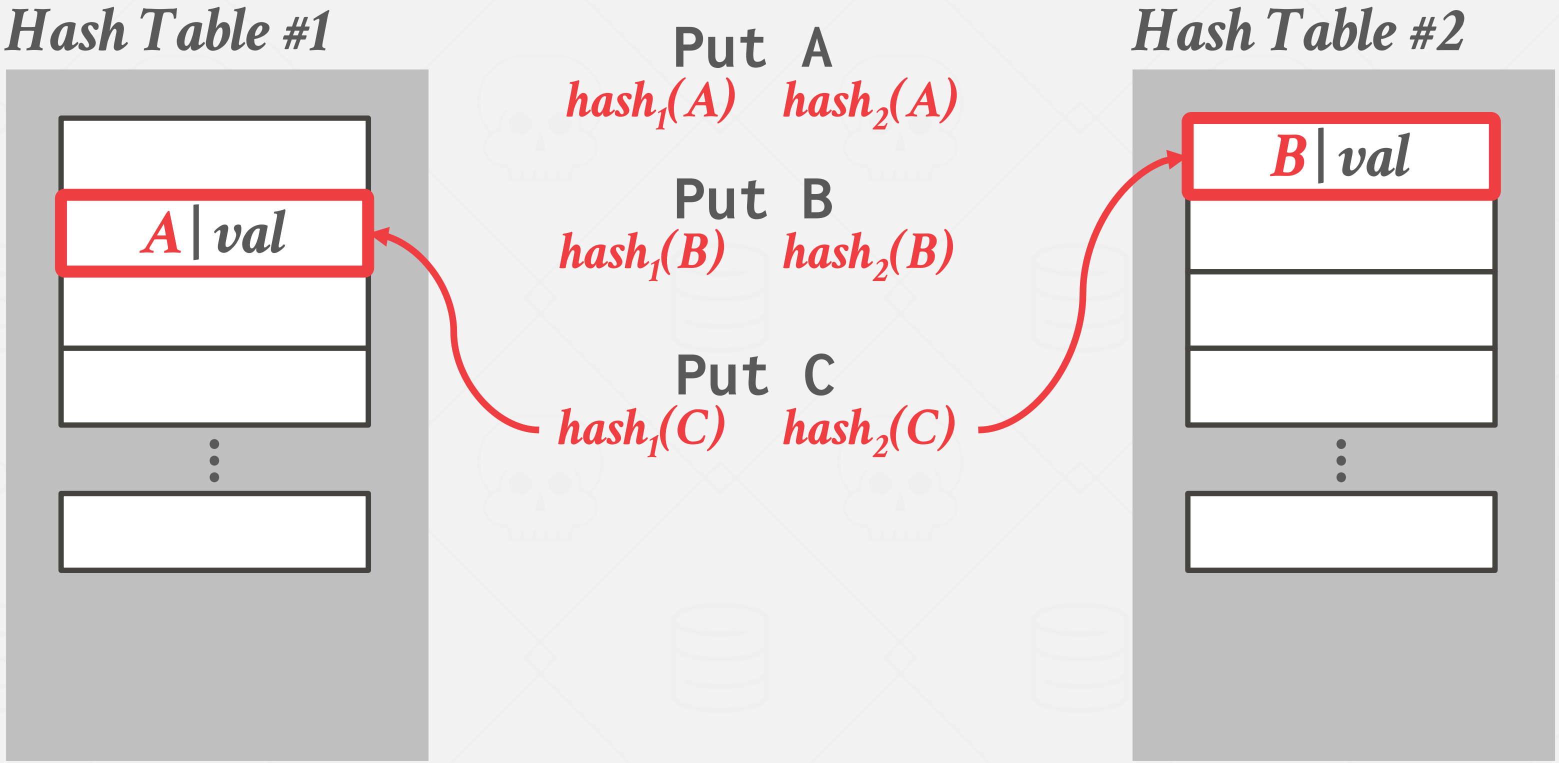
CUCKOO HASHING



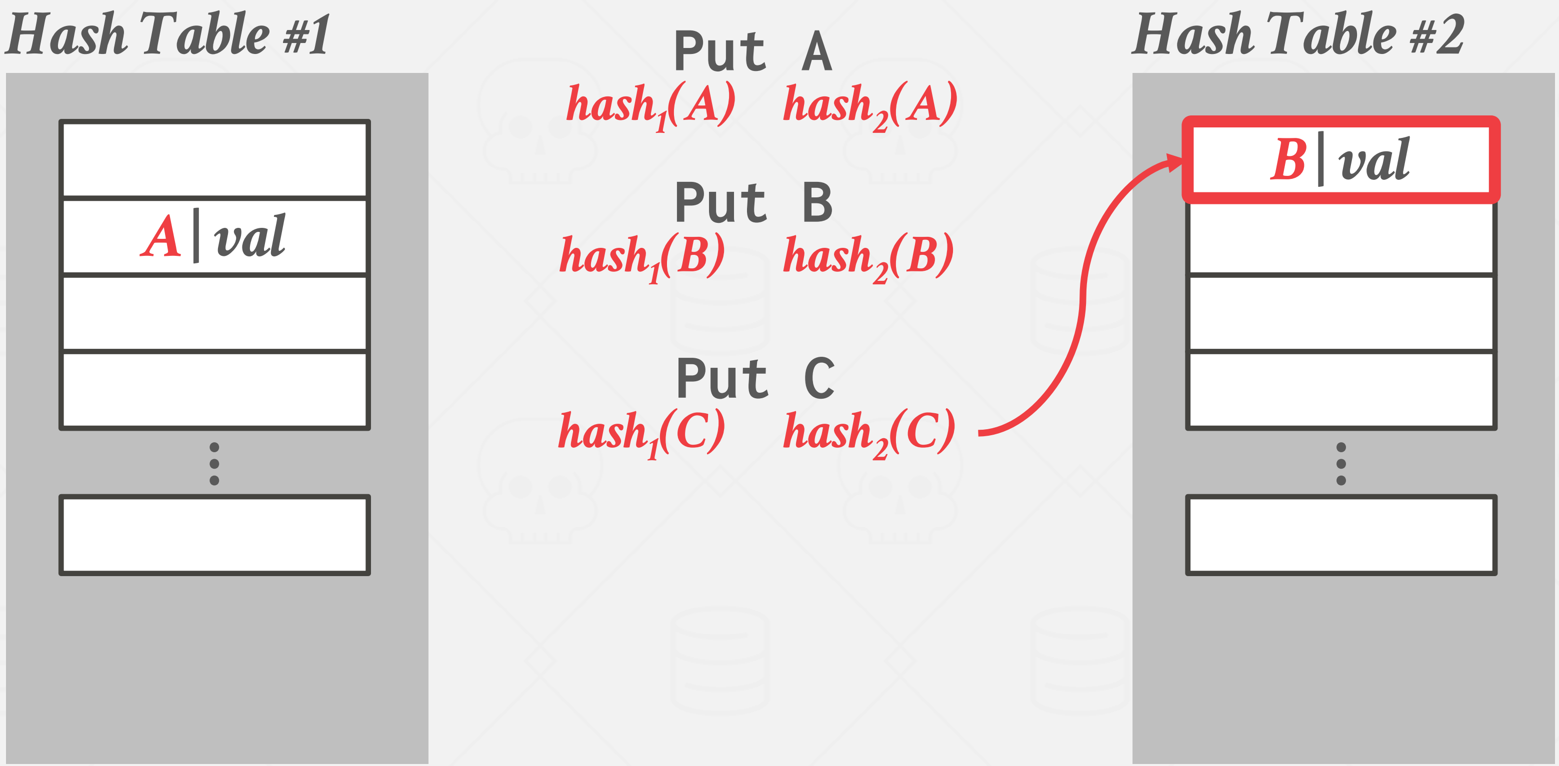
CUCKOO HASHING



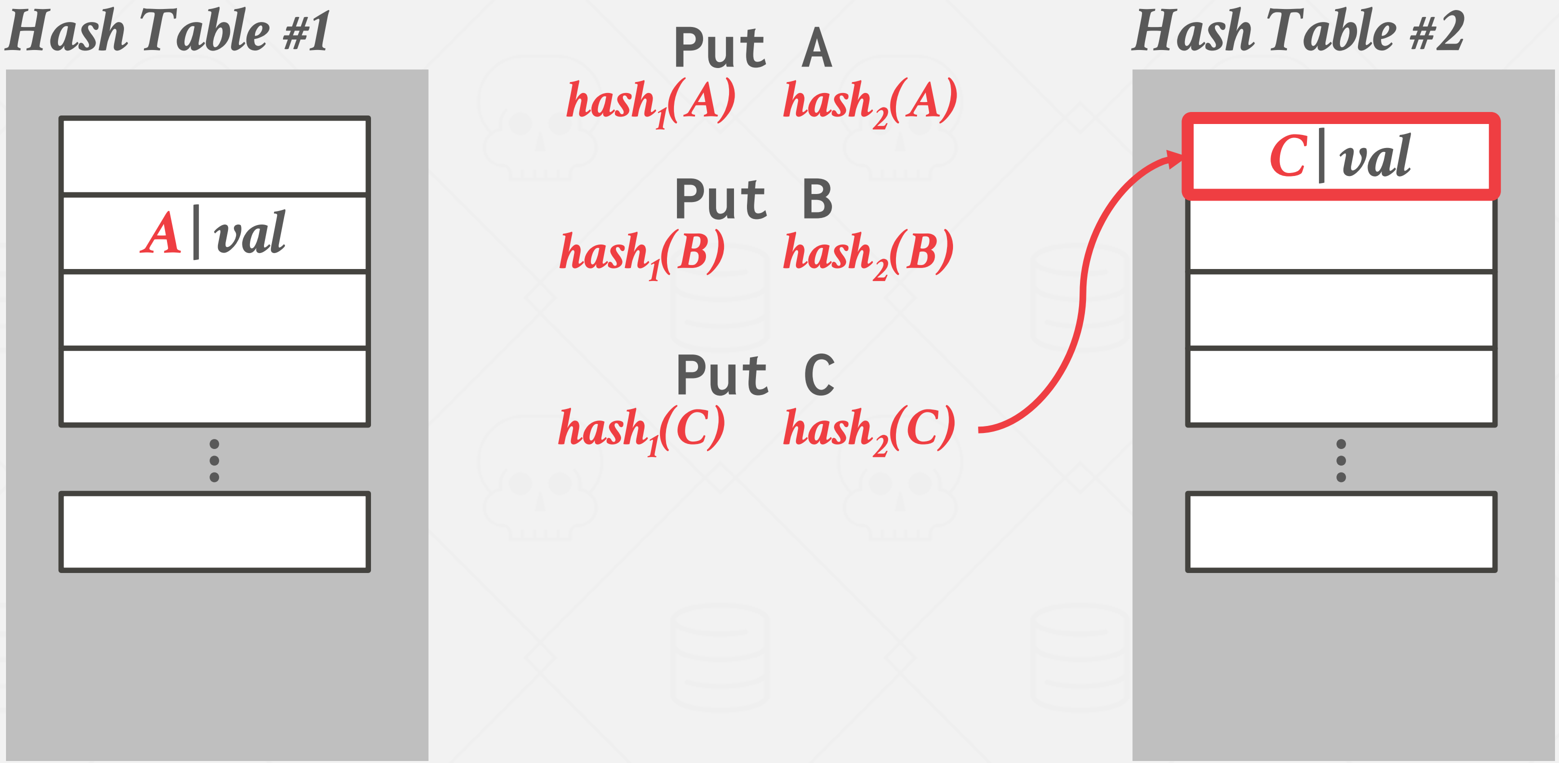
CUCKOO HASHING



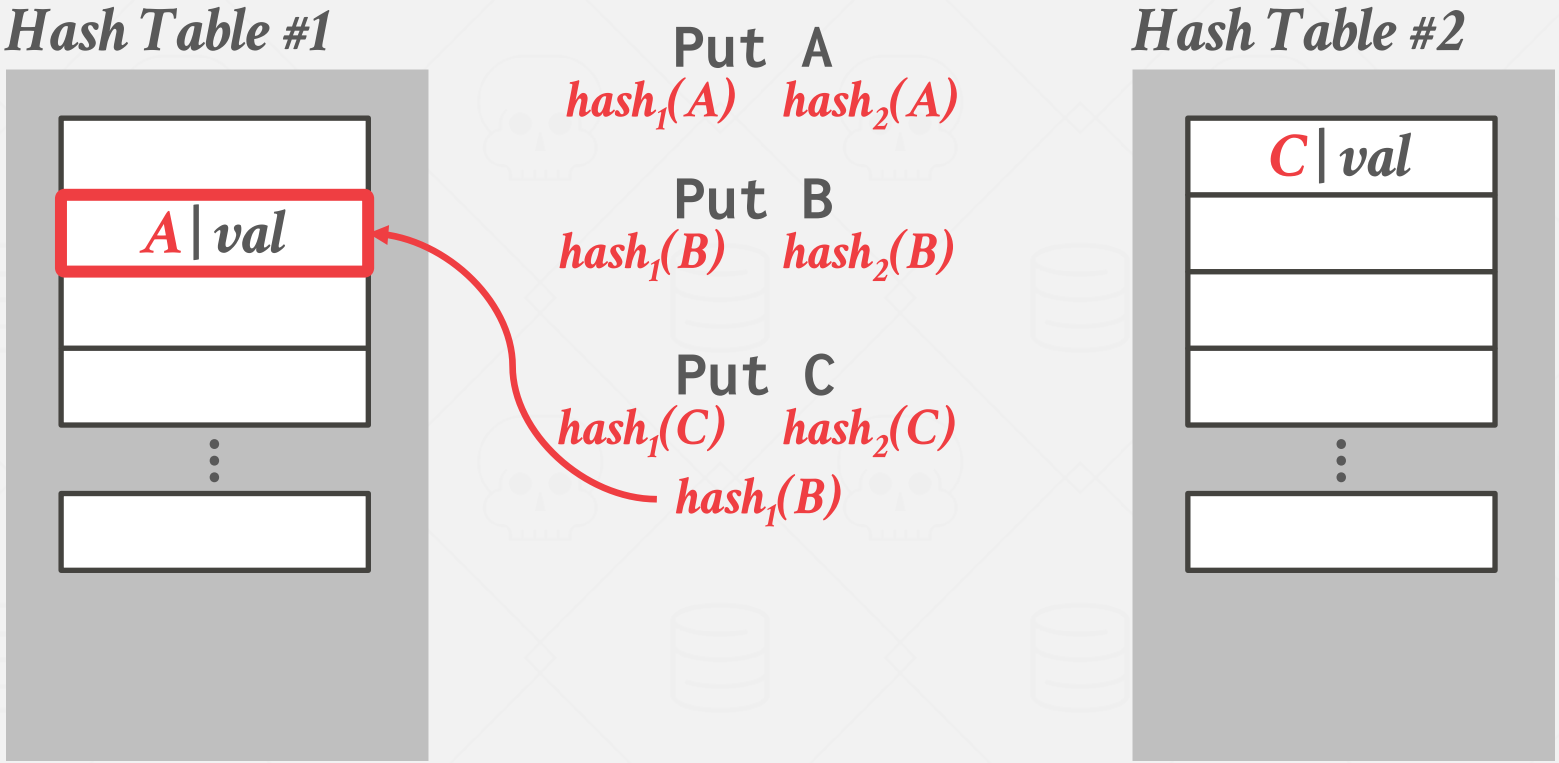
CUCKOO HASHING



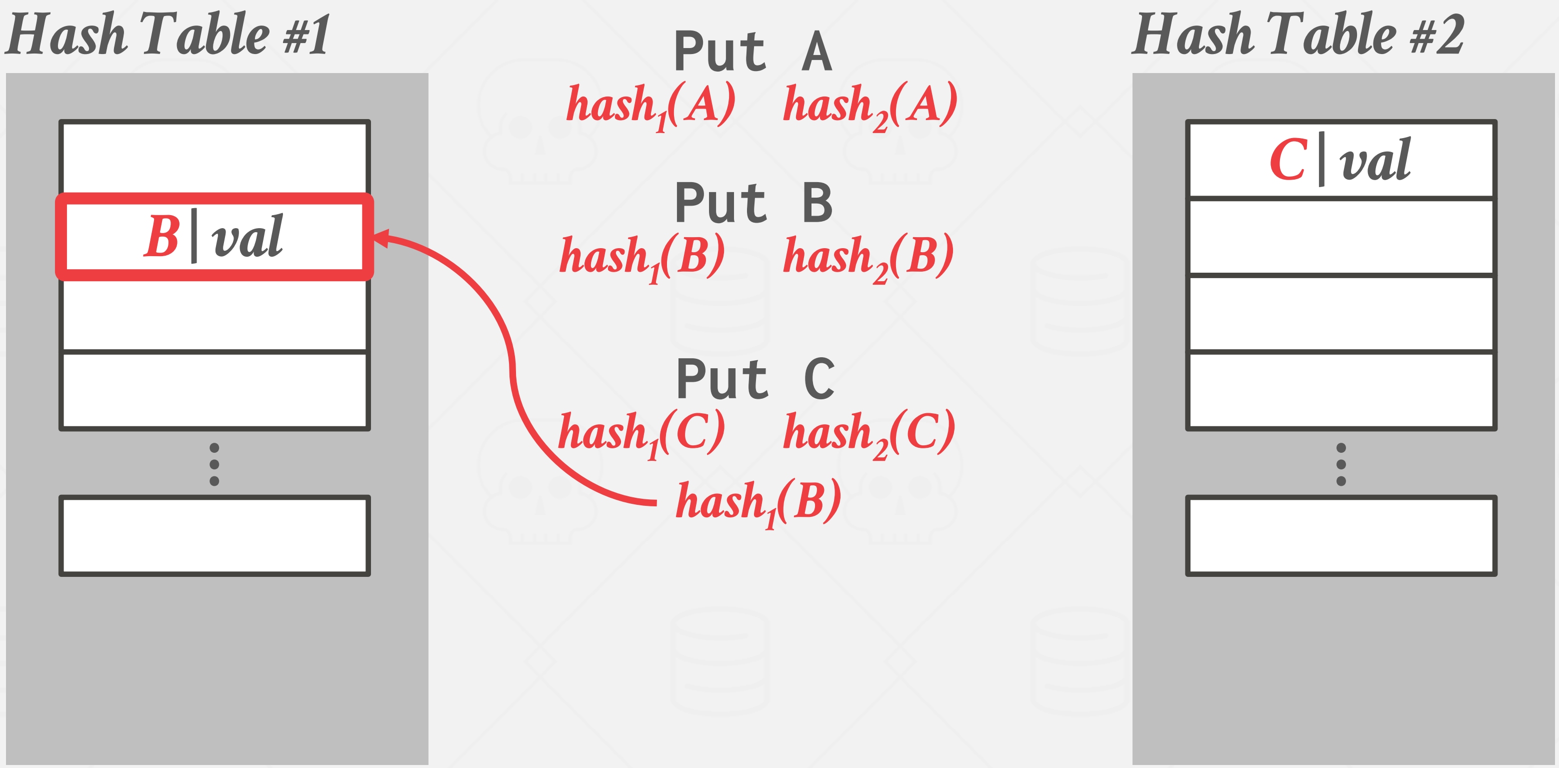
CUCKOO HASHING



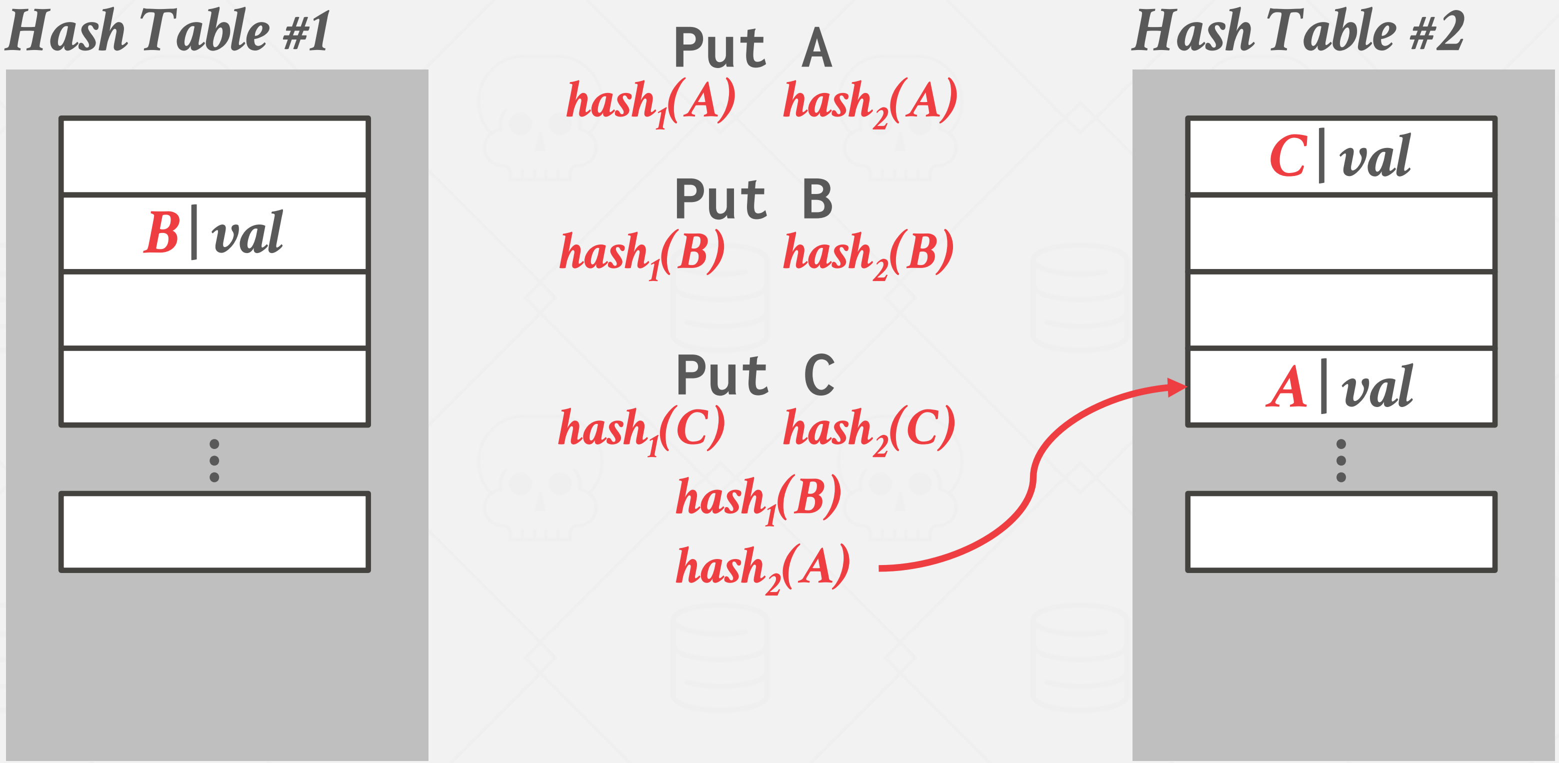
CUCKOO HASHING



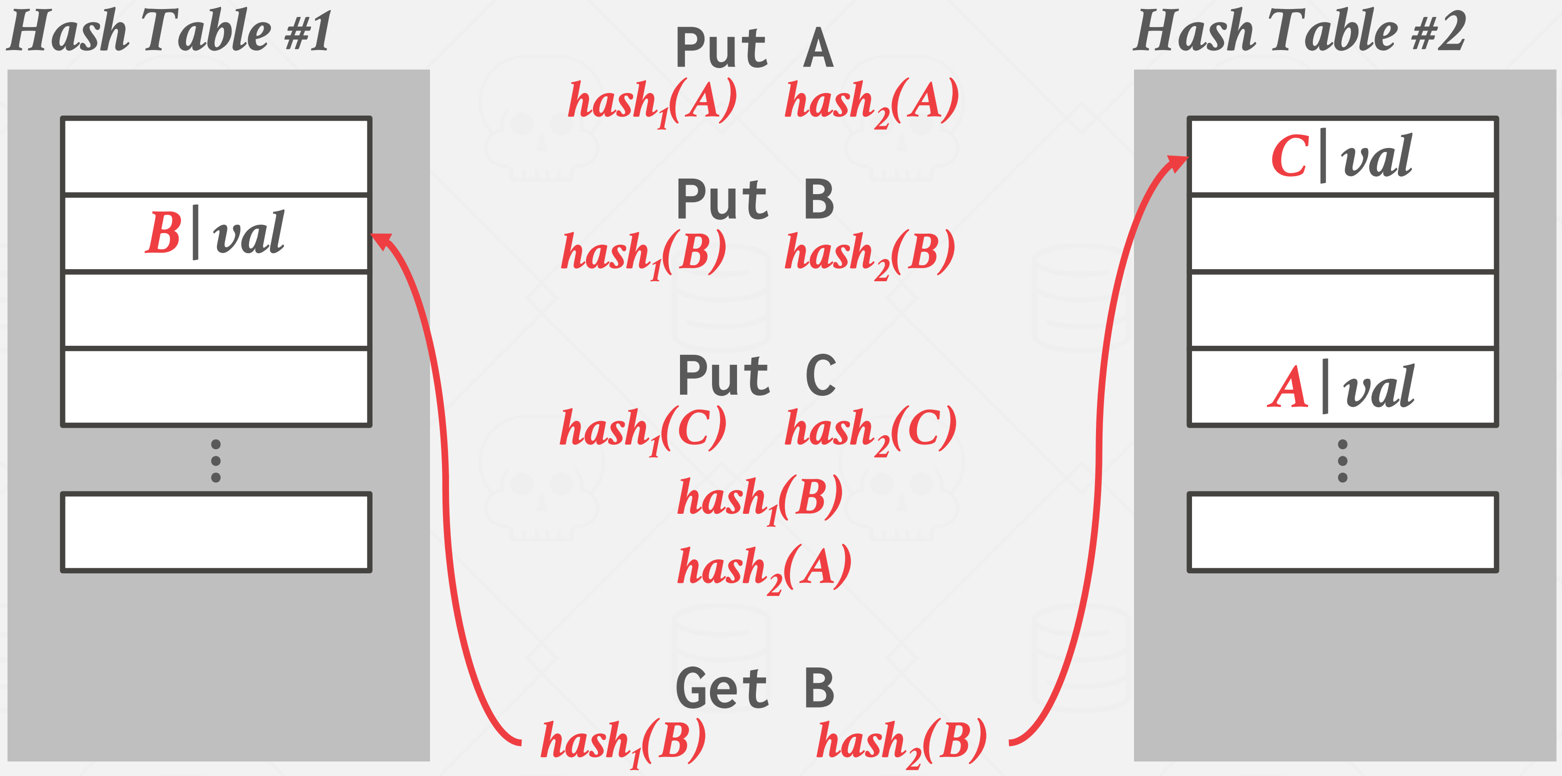
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



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

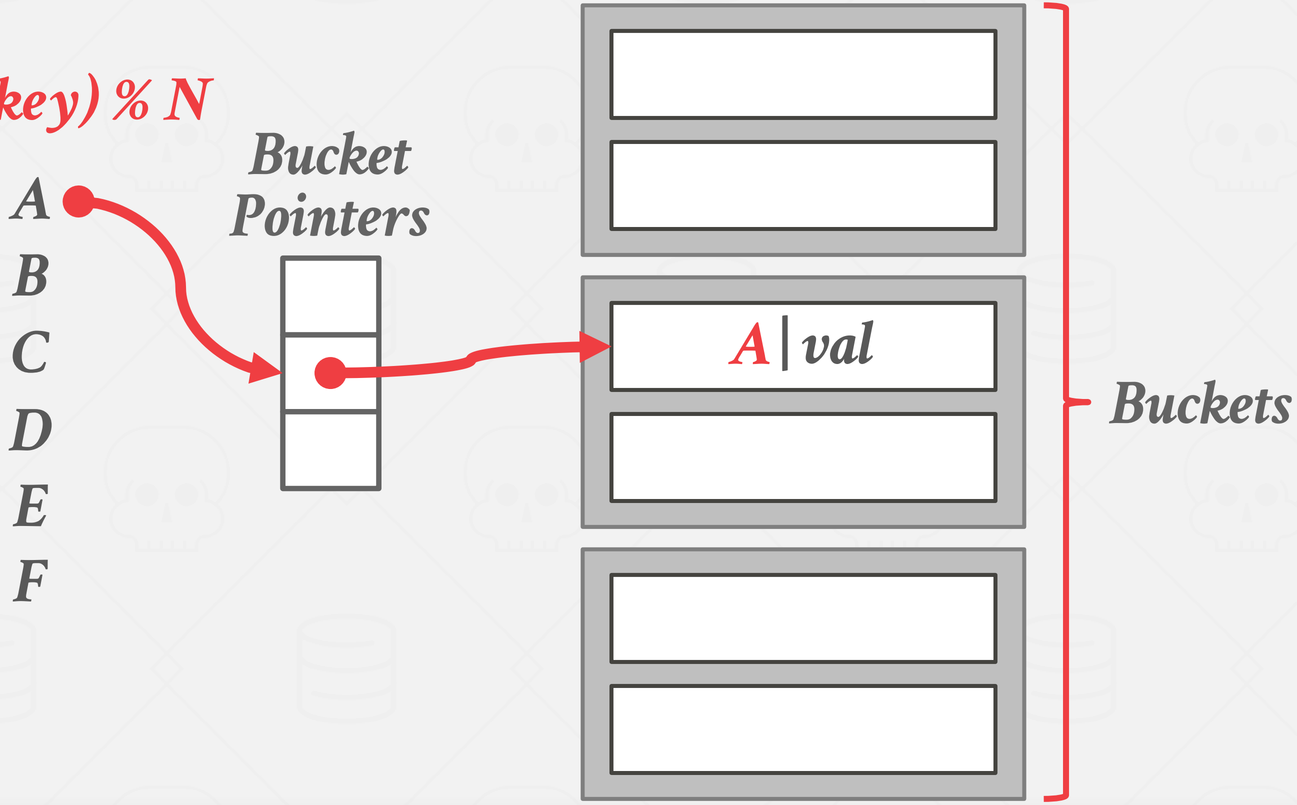
Maintain a linked list of buckets for each slot in the hash table.

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

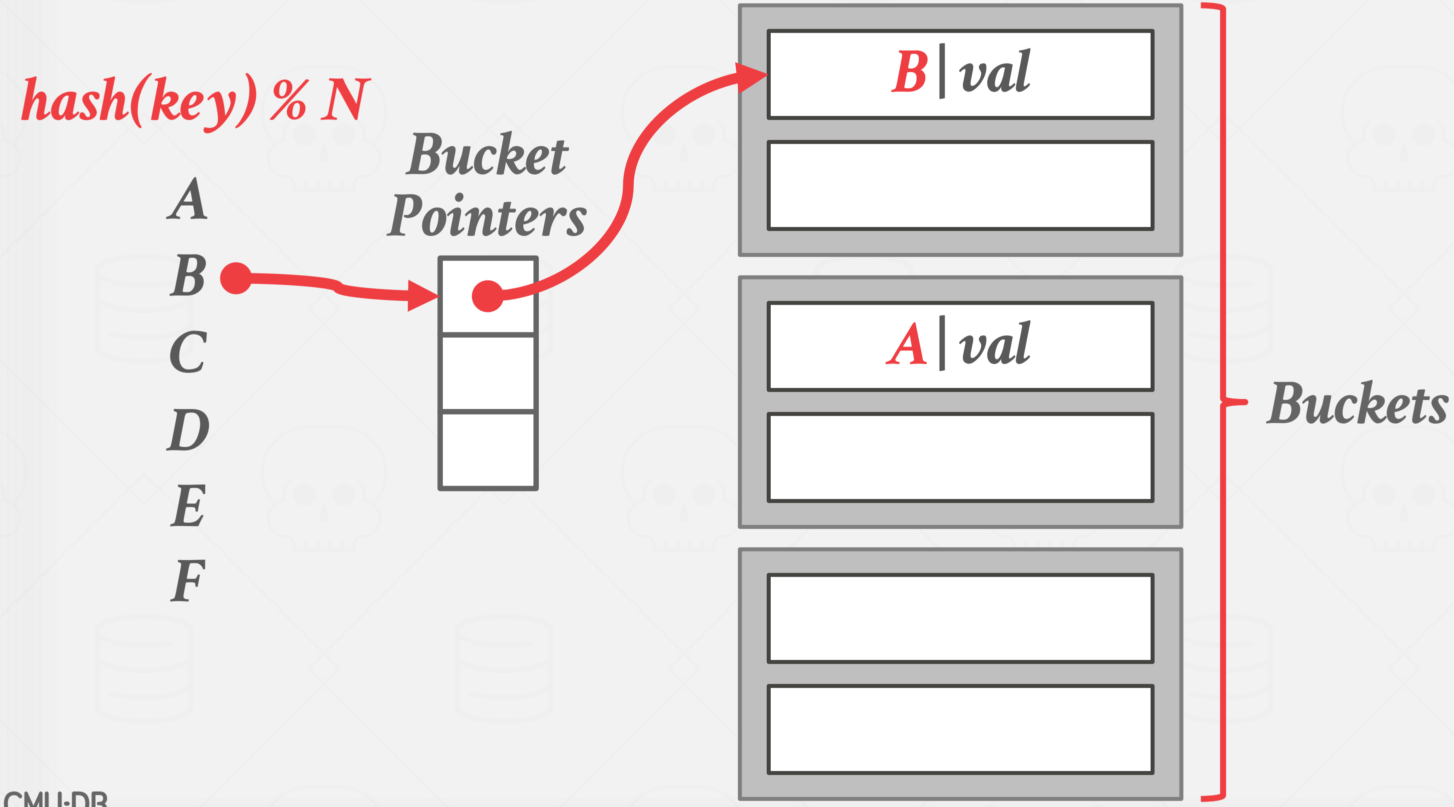
→ To determine whether an element is present, hash to its bucket and scan for it.
→ Insertions and deletions are generalizations of lookups.

CHAINED HASHING

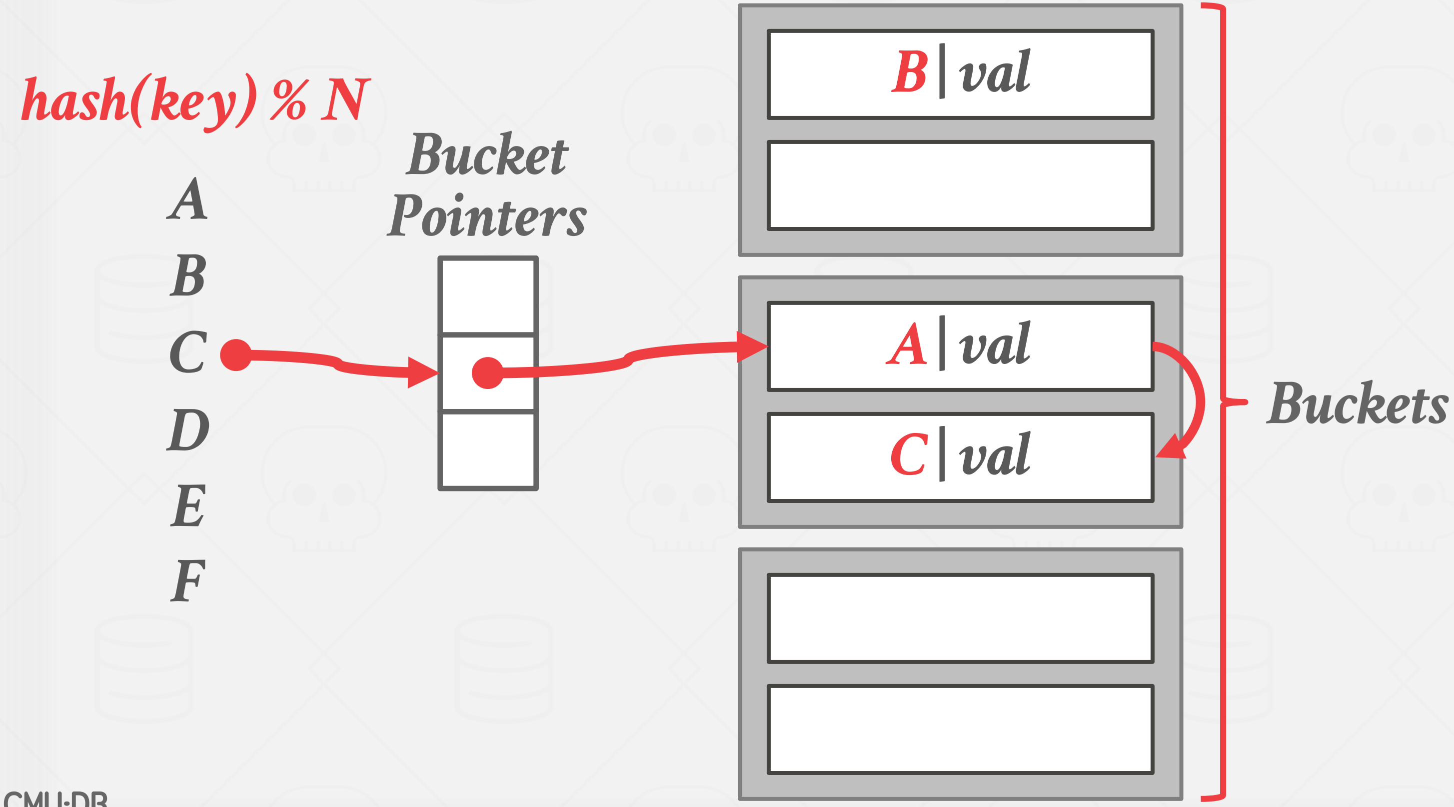
hash(key) % N



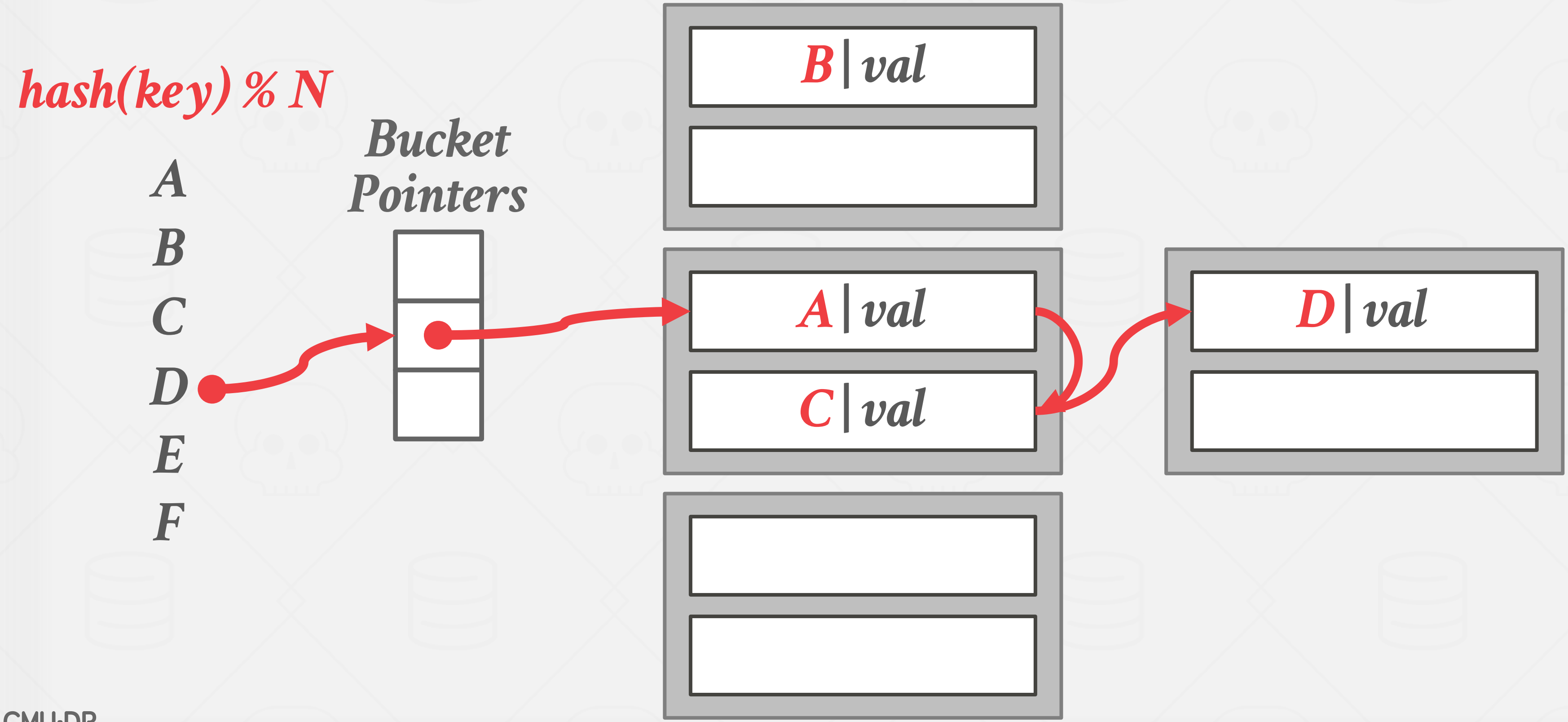
CHAINED HASHING



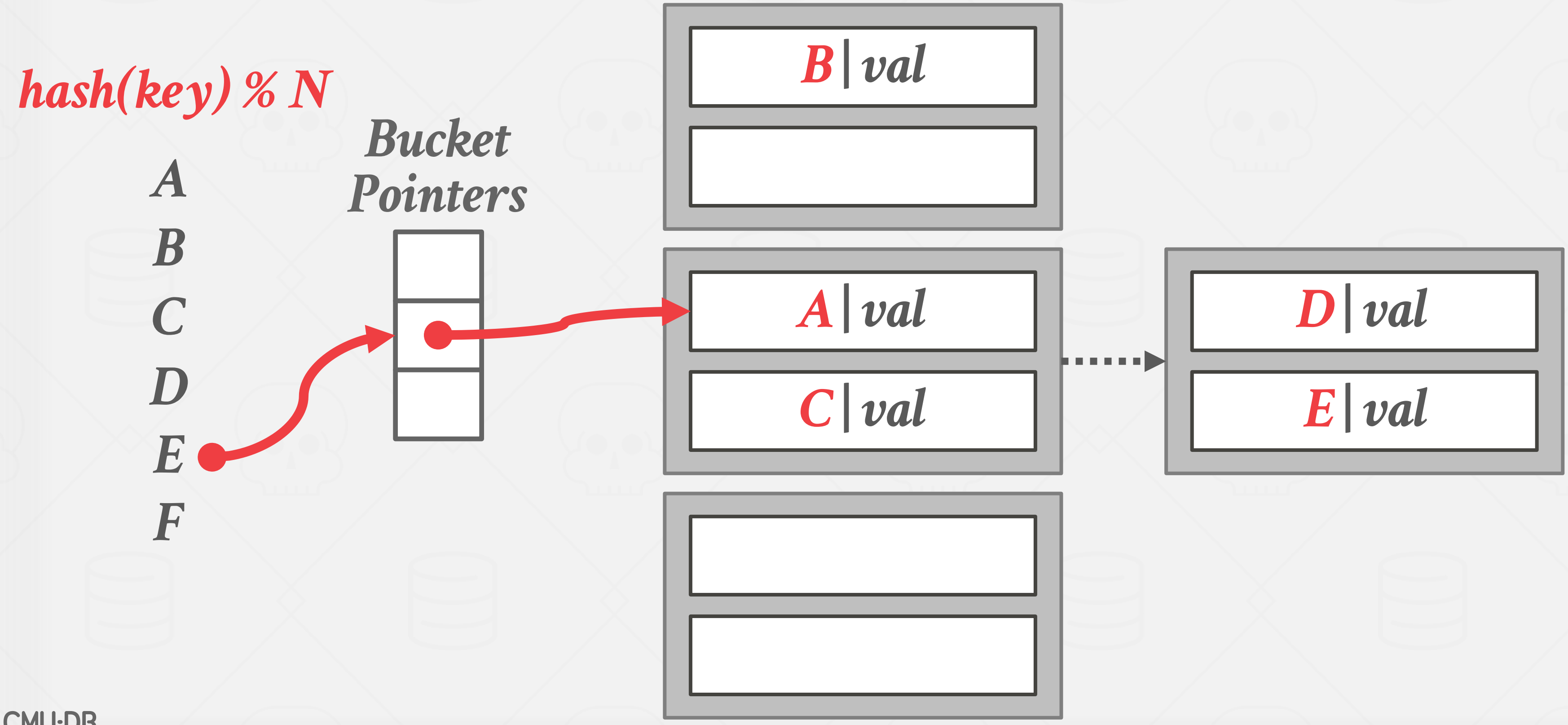
CHAINED HASHING



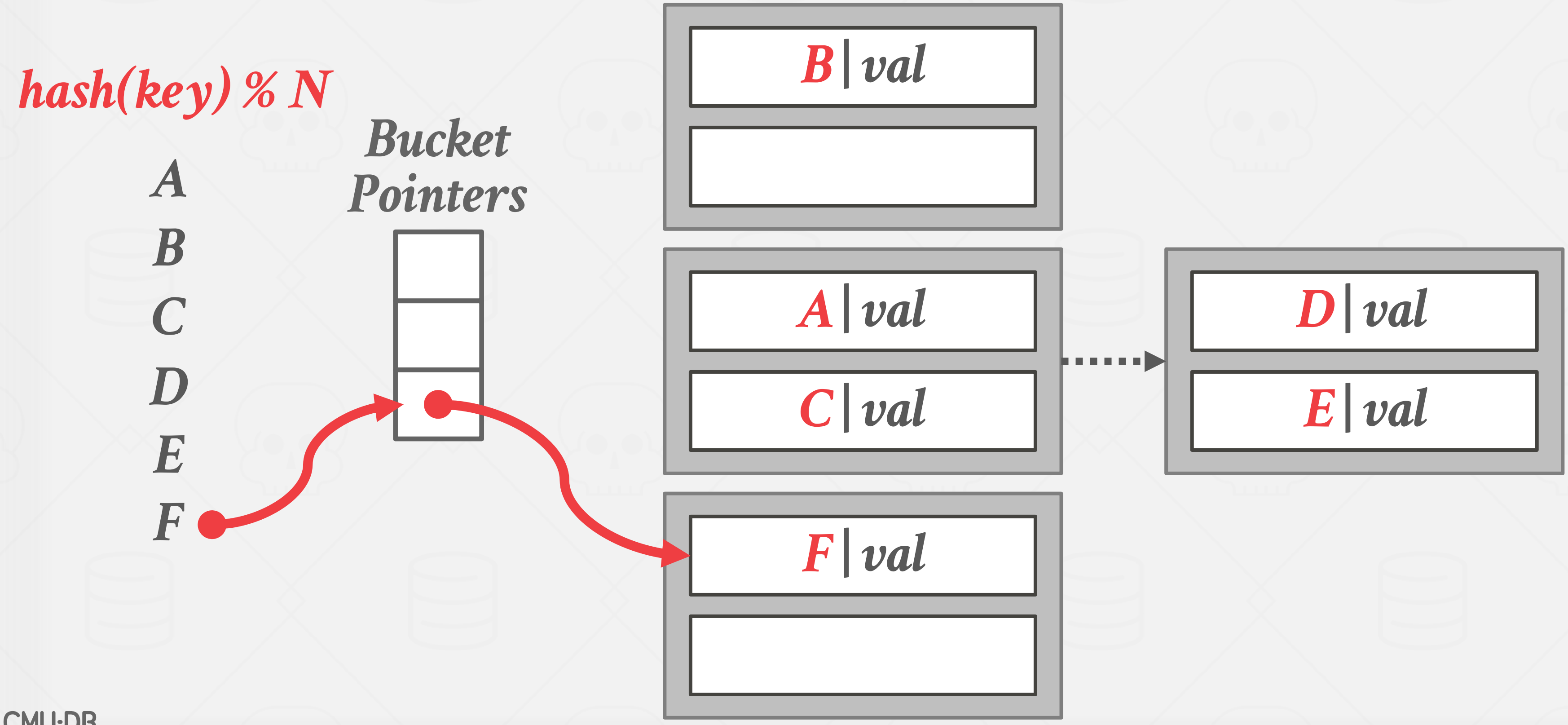
CHAINED HASHING



CHAINED HASHING



CHAINED HASHING



EXTENDIBLE HASHING

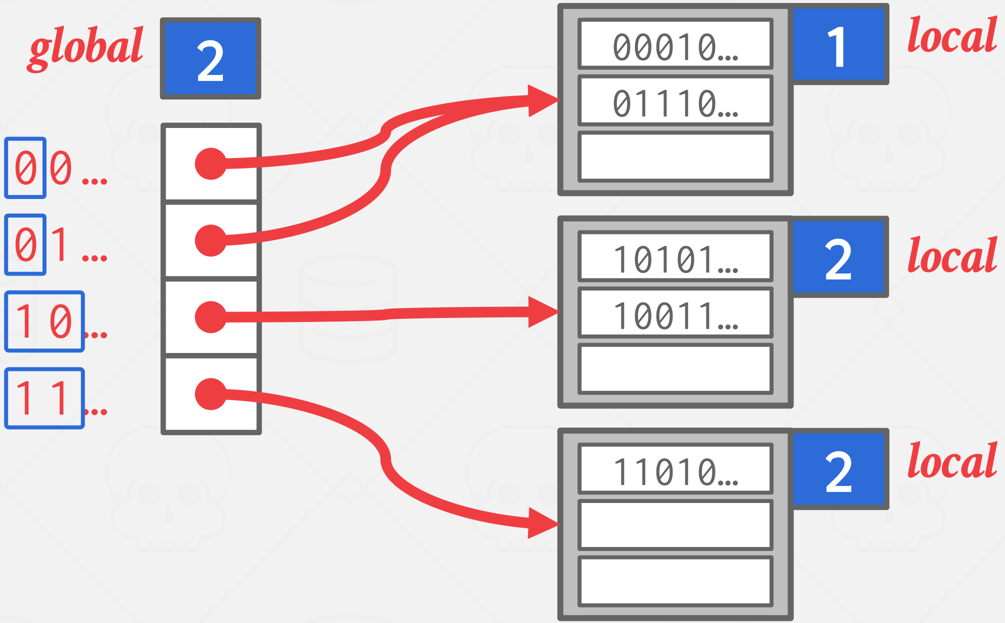
Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

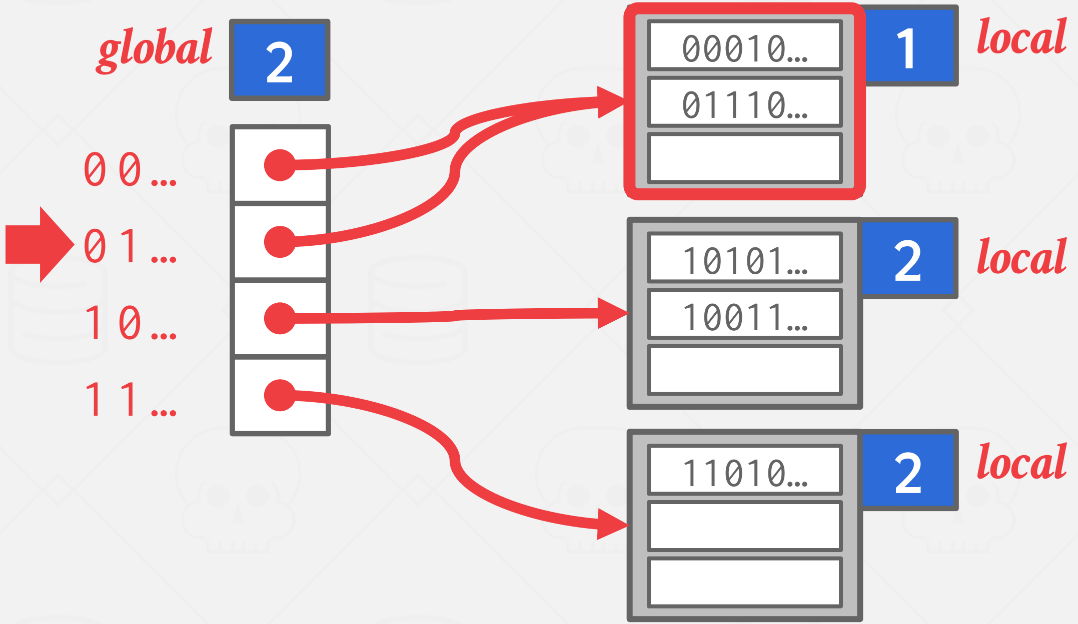
Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

EXTENDIBLE HASHING



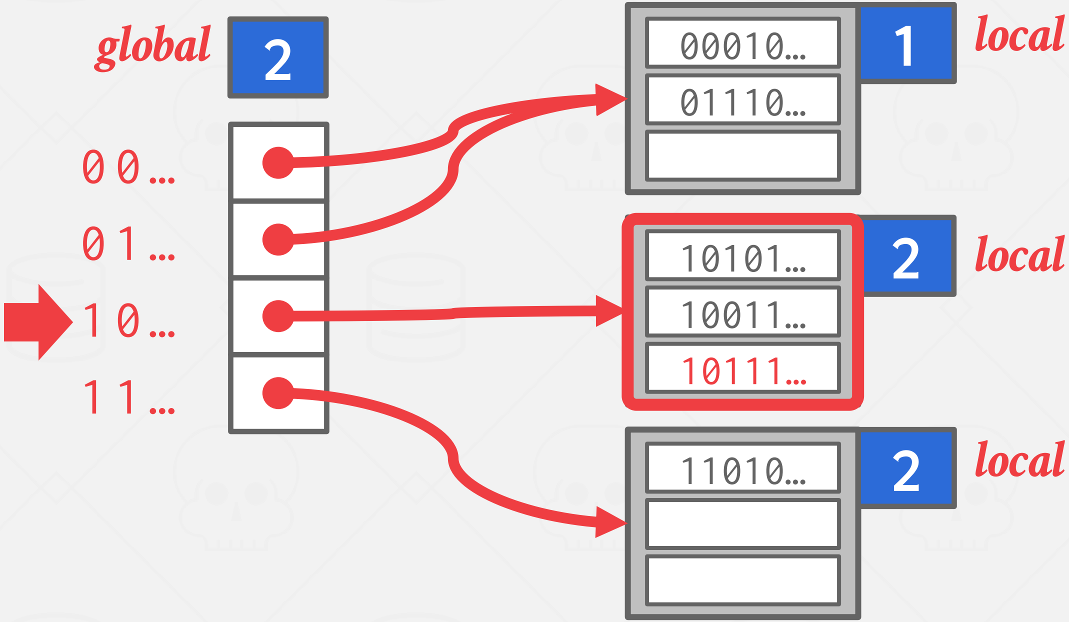
EXTENDIBLE HASHING



Get A

hash(A) = $\boxed{01}110\ldots$

EXTENDIBLE HASHING



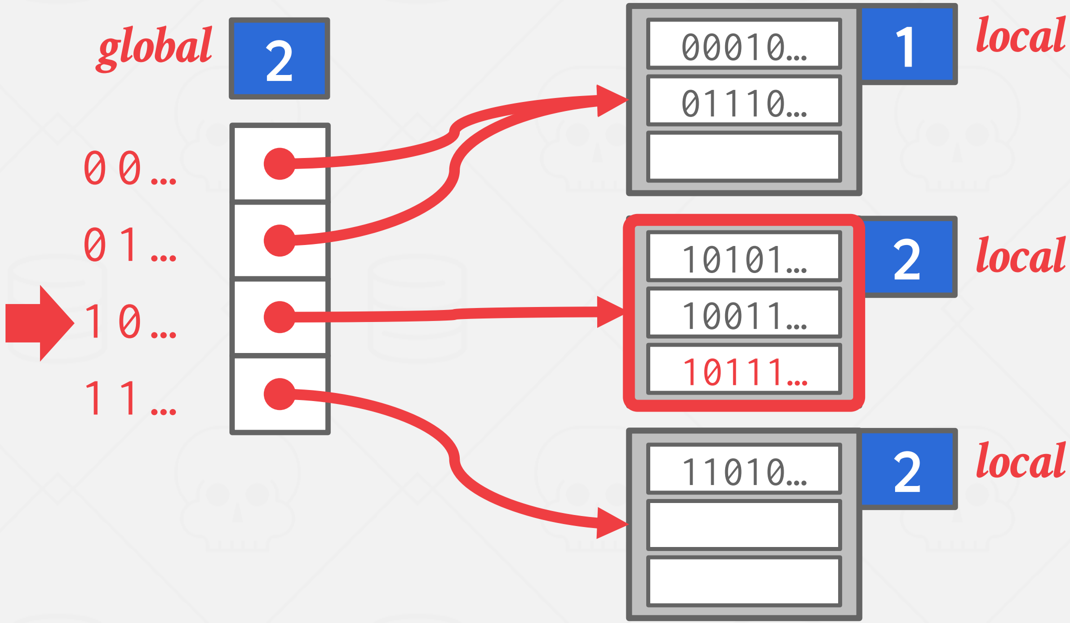
Get A

hash(A) = 01110…

Put B

hash(B) = 10111…

EXTENDIBLE HASHING

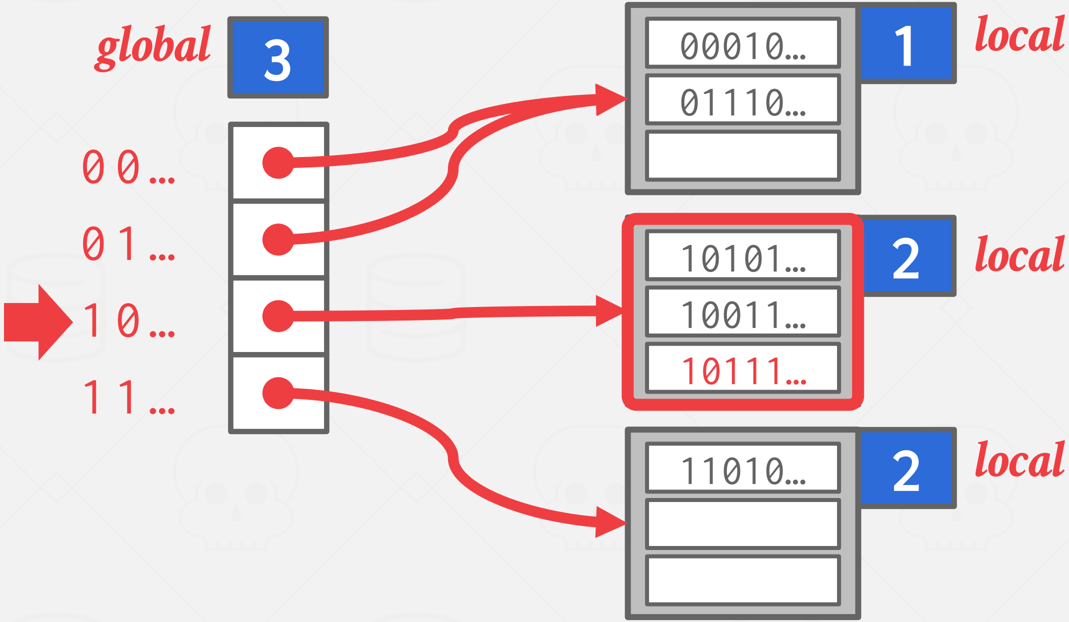


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

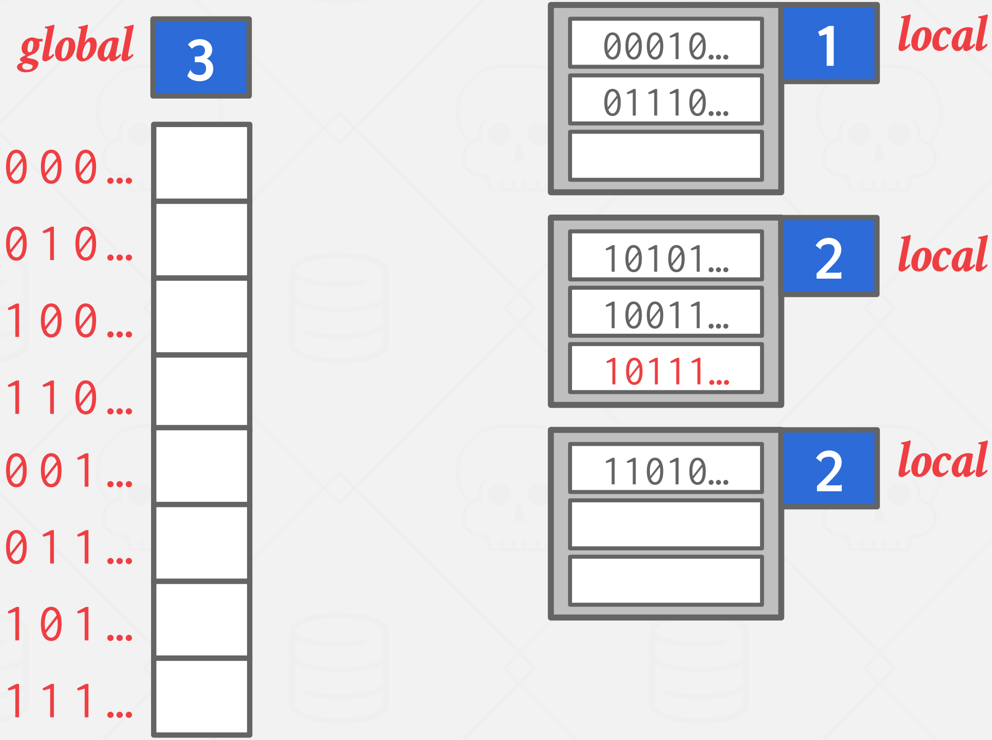


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING



Get A

hash(A) = 01110…

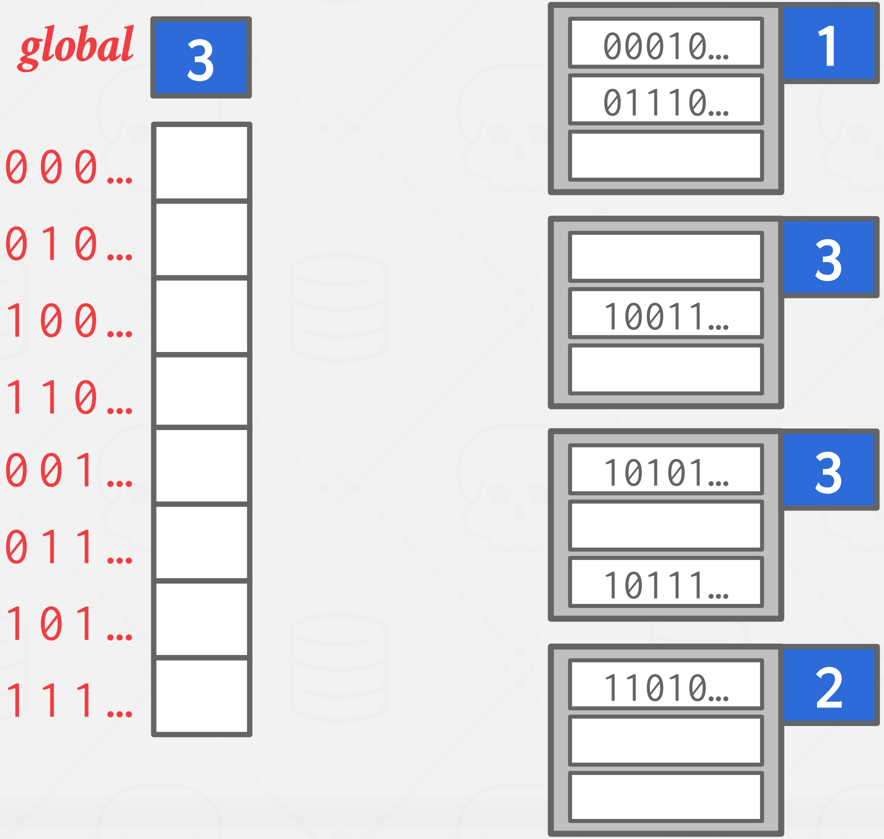
Put B

hash(B) = 10111…

Put C

hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING

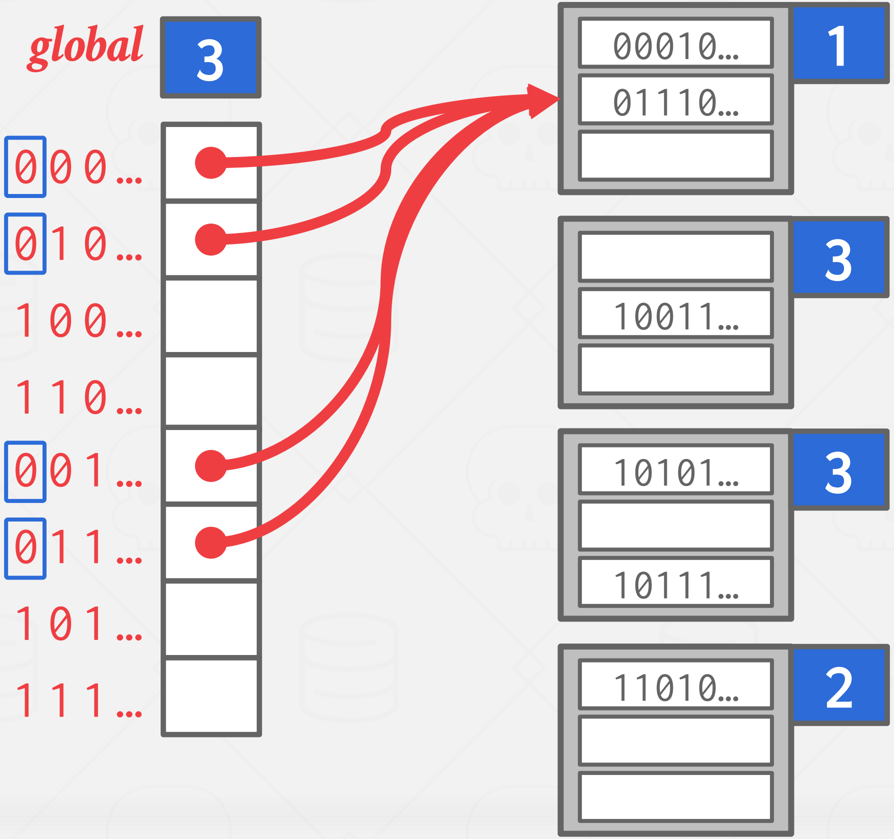


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

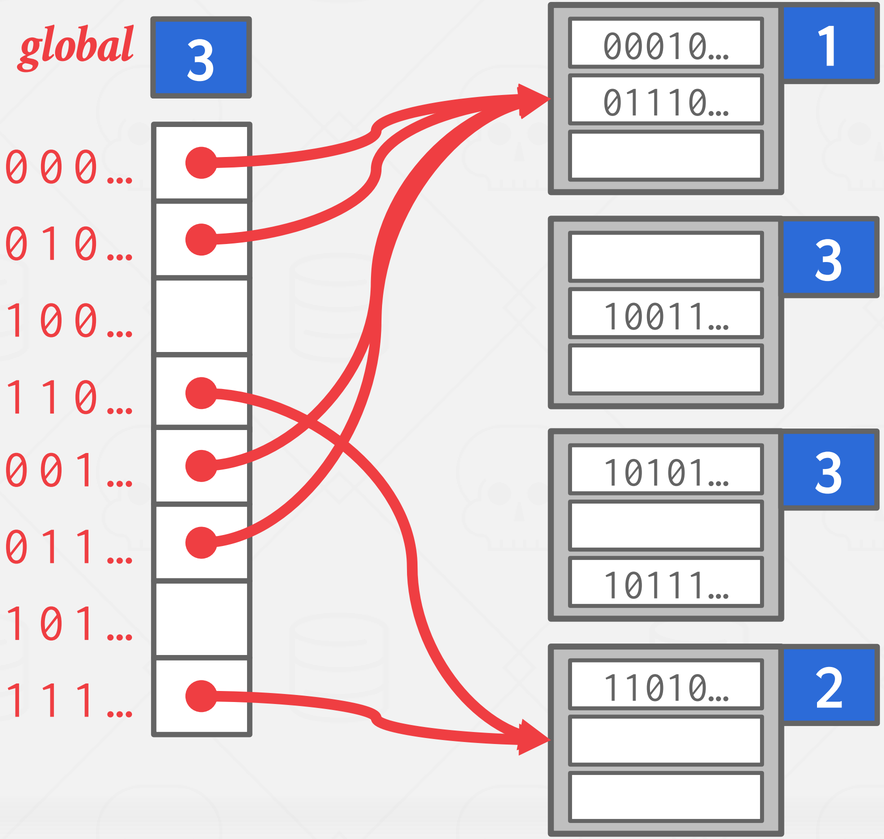


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

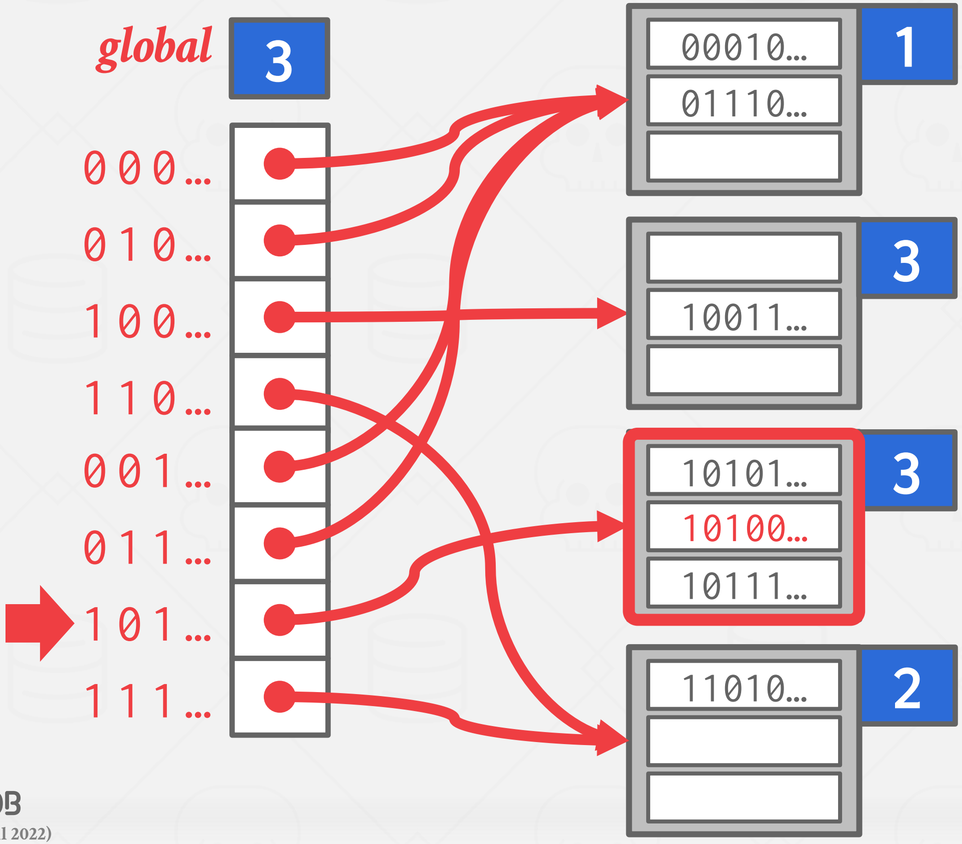


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
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.

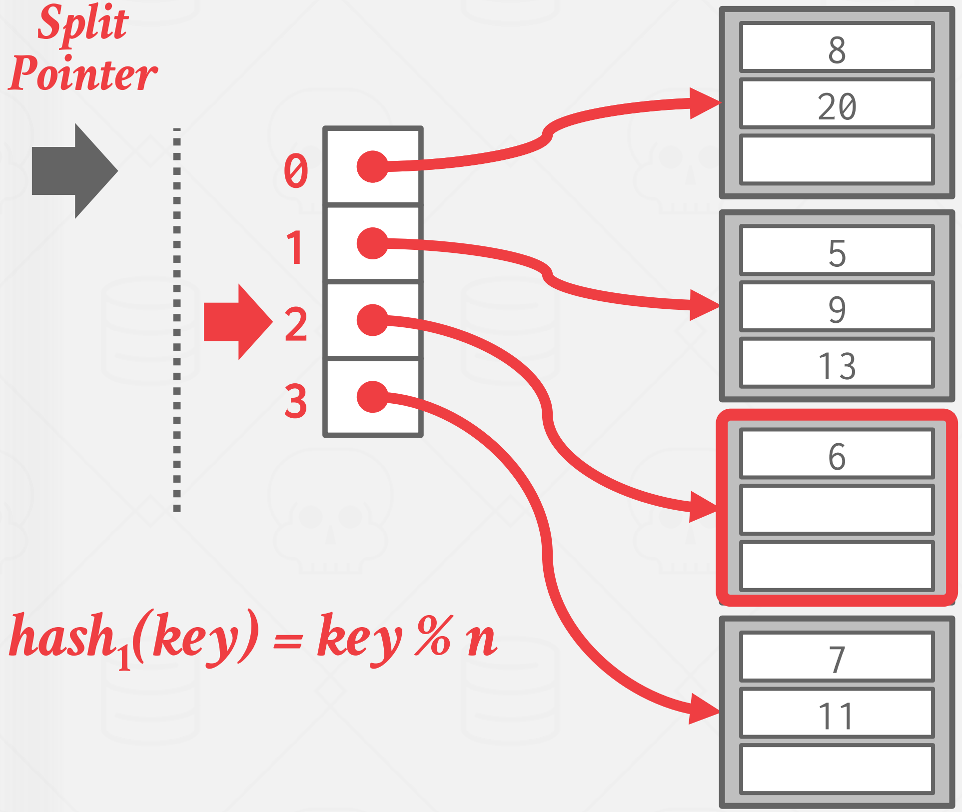
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

→ Space Utilization

→ Average Length of Overflow Chains

LINEAR HASHING

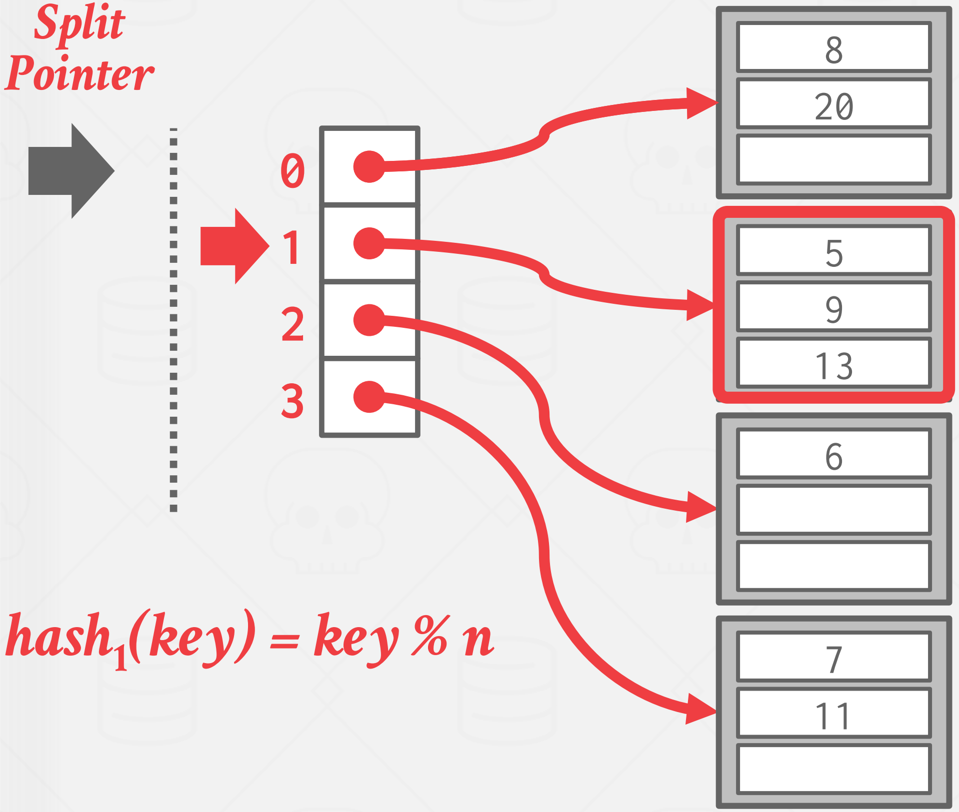


Get 6

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

$hash_1(key) = key \% n$

LINEAR HASHING

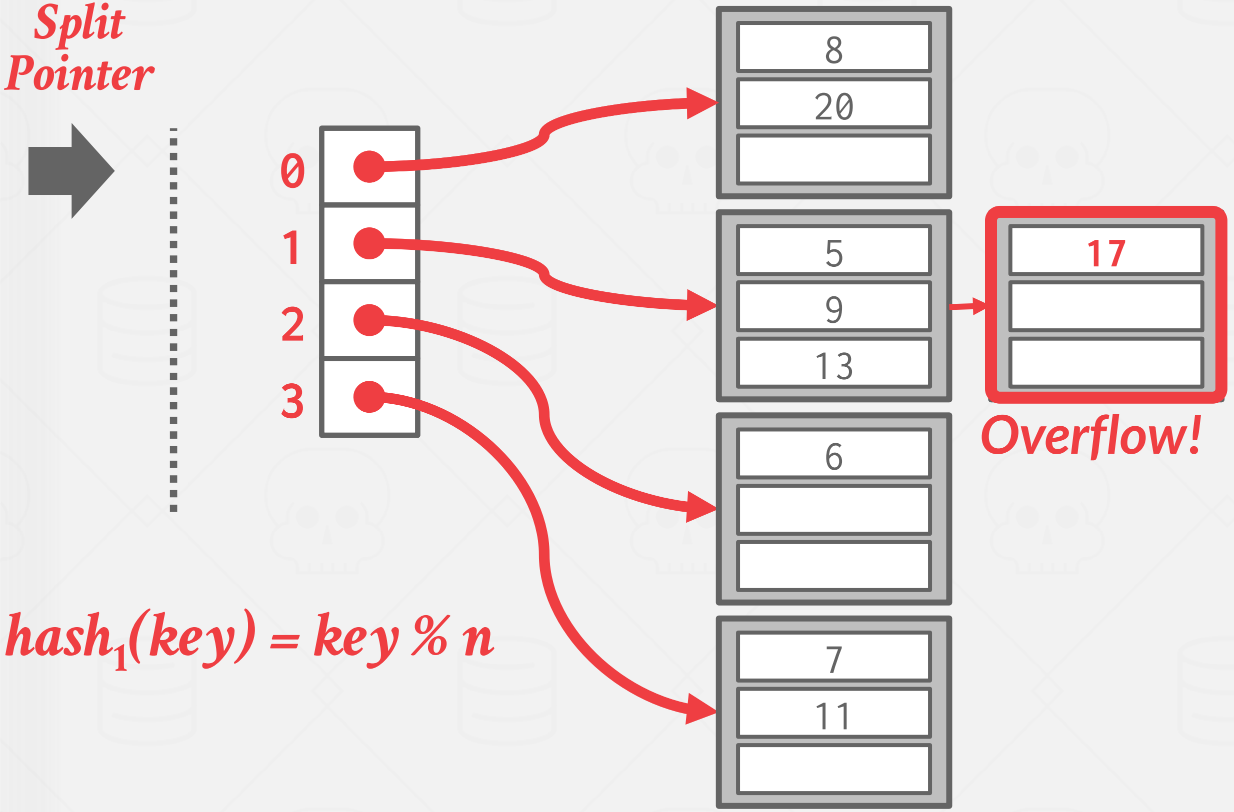


Get 6
$hash_1(6) = 6 \% 4 = 2$

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

$hash_1(key) = key \% n$

LINEAR HASHING



Get 6

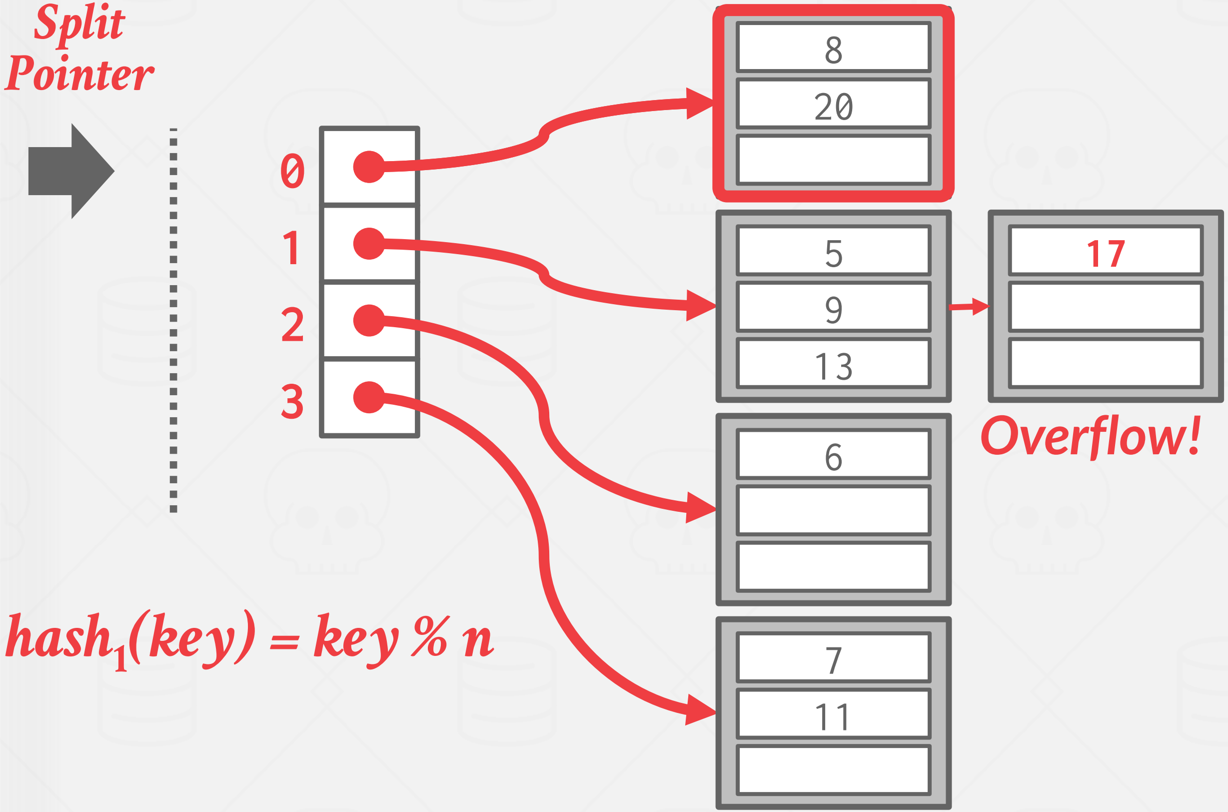
$hash_1(6) = 6 \% 4 = 2$

Put 17

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

$hash_1(key) = key \% n$

LINEAR HASHING



$hash_1(key) = key \% n$

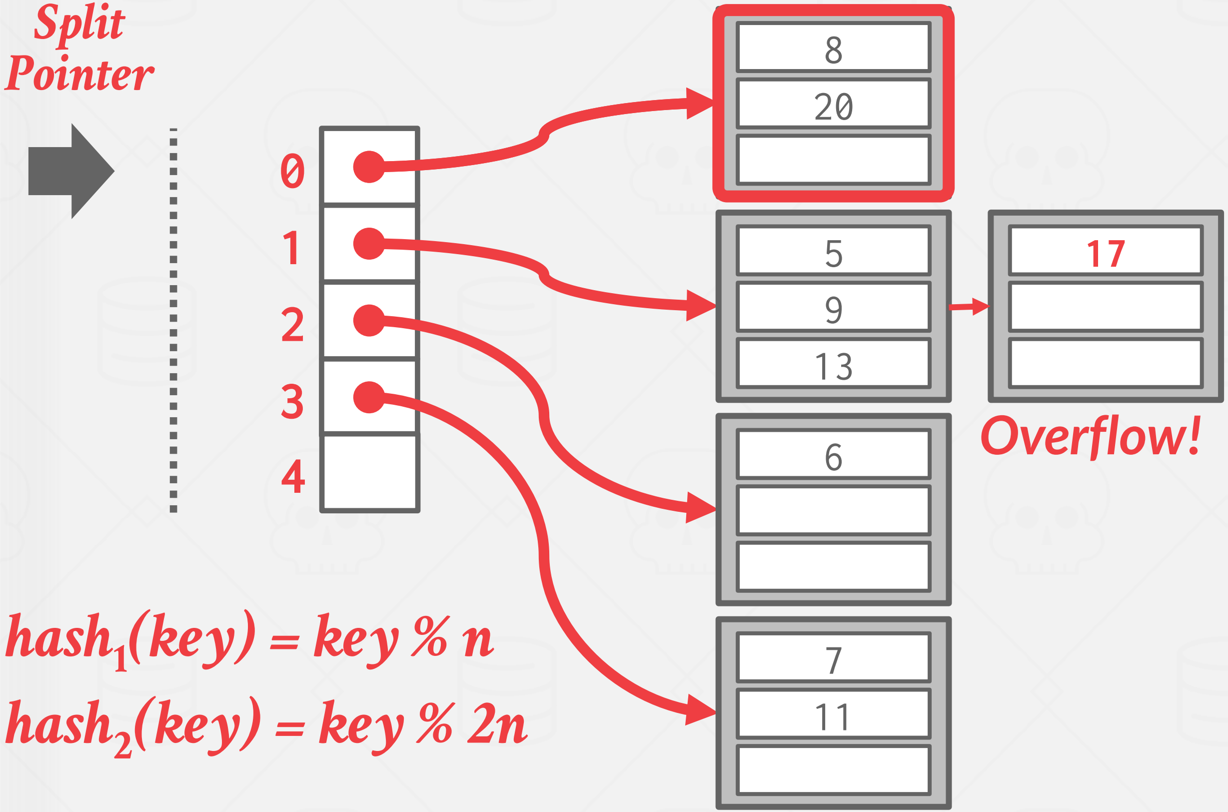
Get 6

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

Put 17

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

LINEAR HASHING



Get 6

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

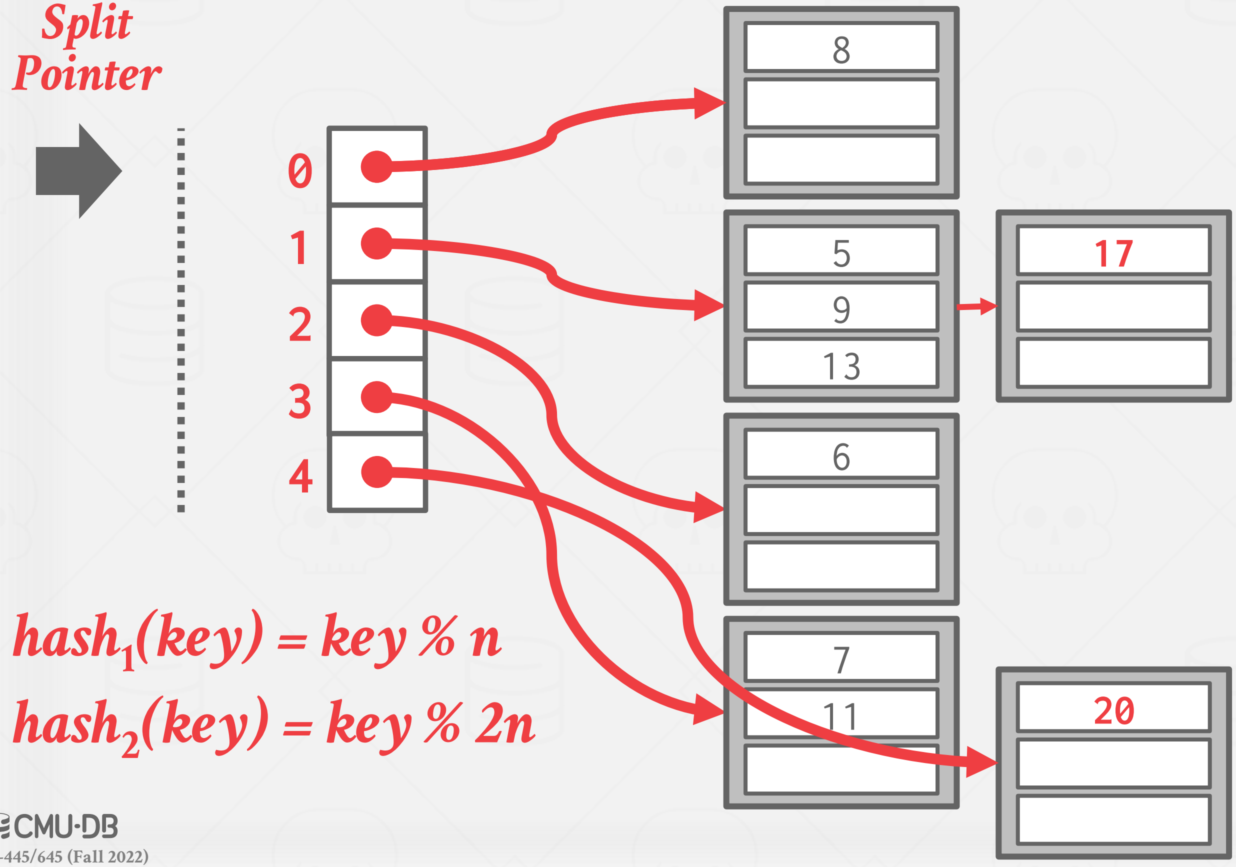
Put 17

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING



Get 6

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

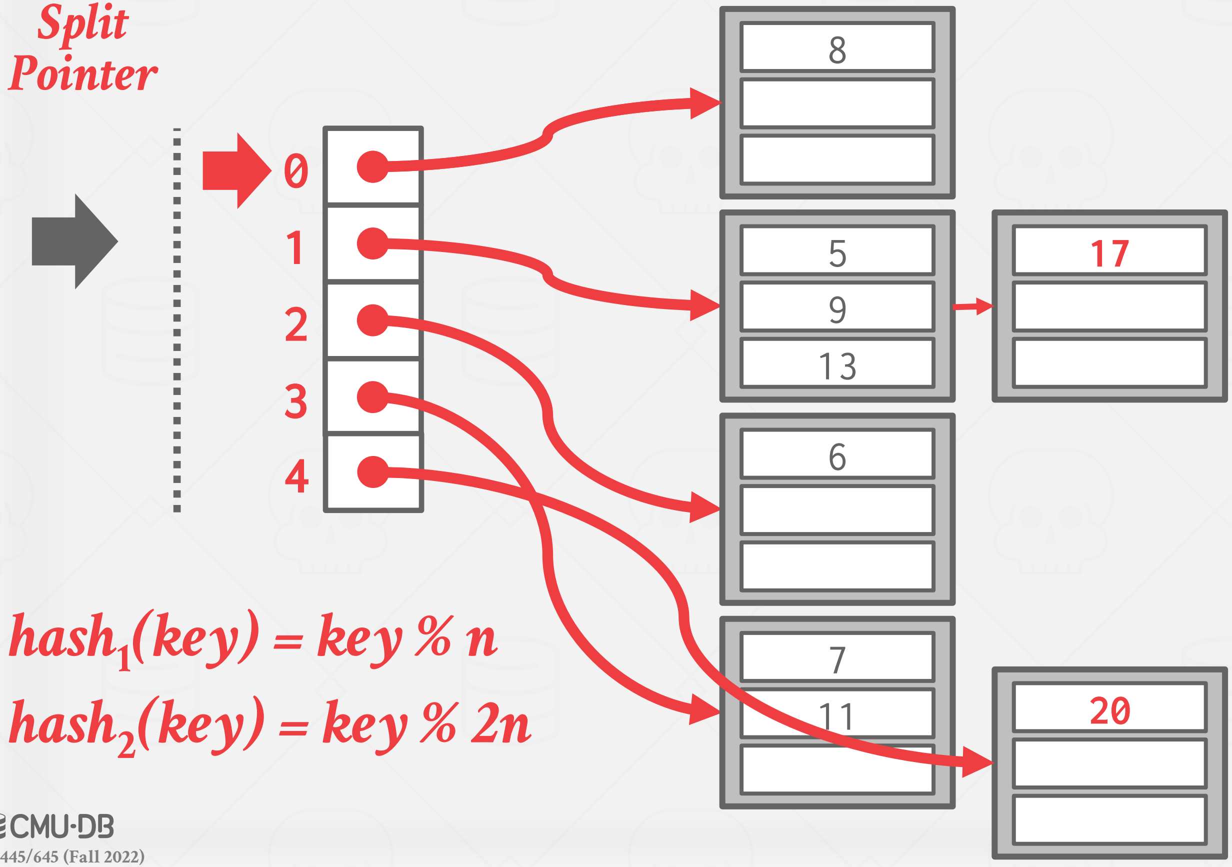
Put 17

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING



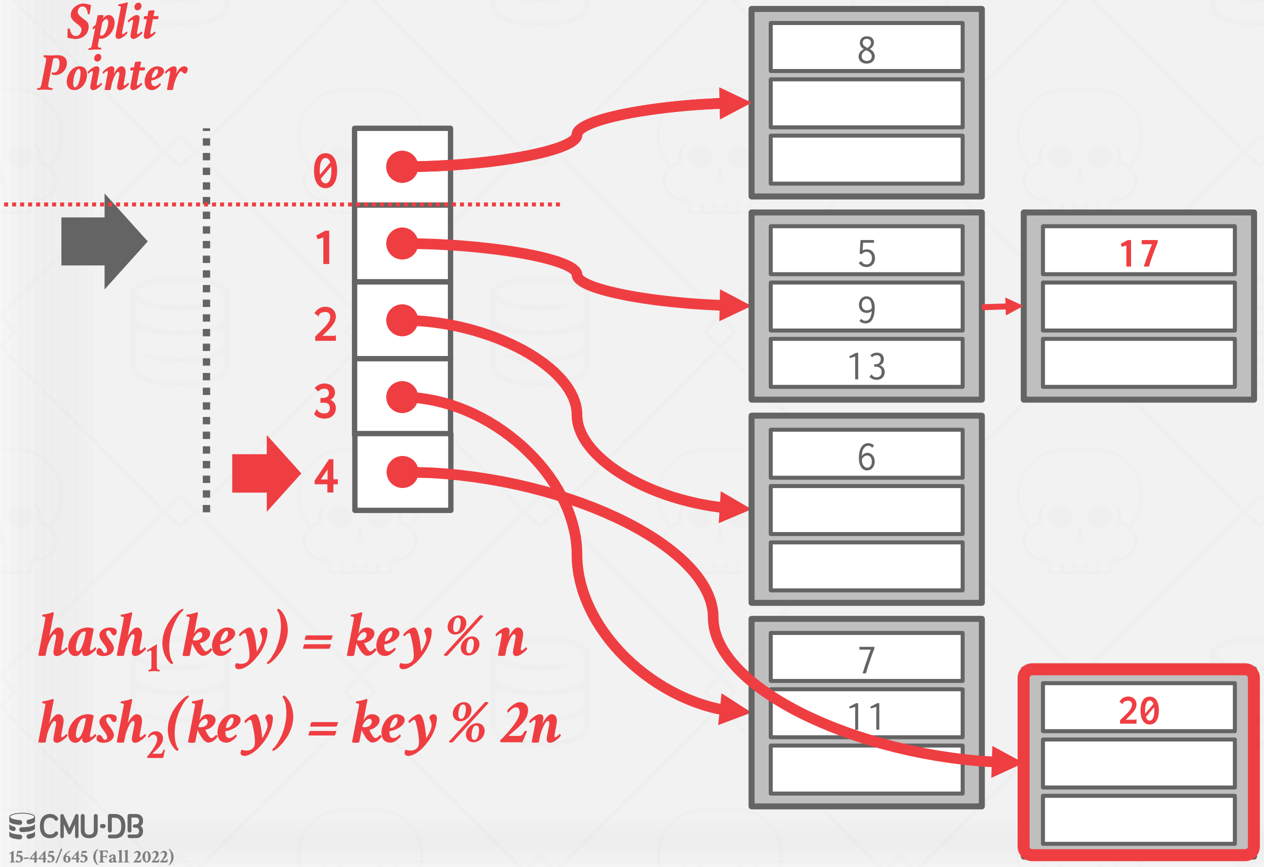
Get 6
$hash_1(6) = 6 \% 4 = 2$

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

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

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

LINEAR HASHING



Get 6

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

Put 17

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

Get 20

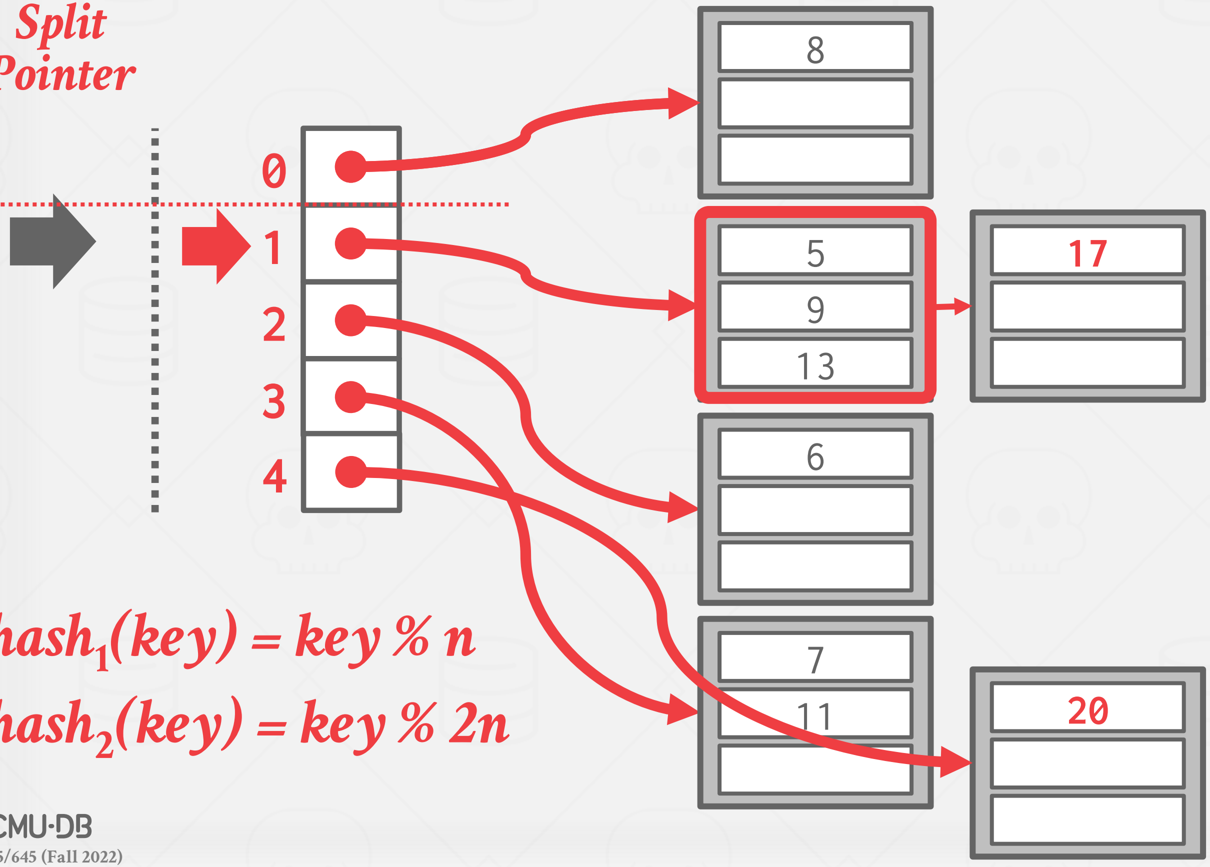
$hash_1(20)$ = 20 % 4 = 0

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING



$hash_1(key) = key \,\%\, n$

$hash_2(key) = key \,\%\, 2n$

Get 6

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

Put 17

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

Get 20

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

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

Get 9

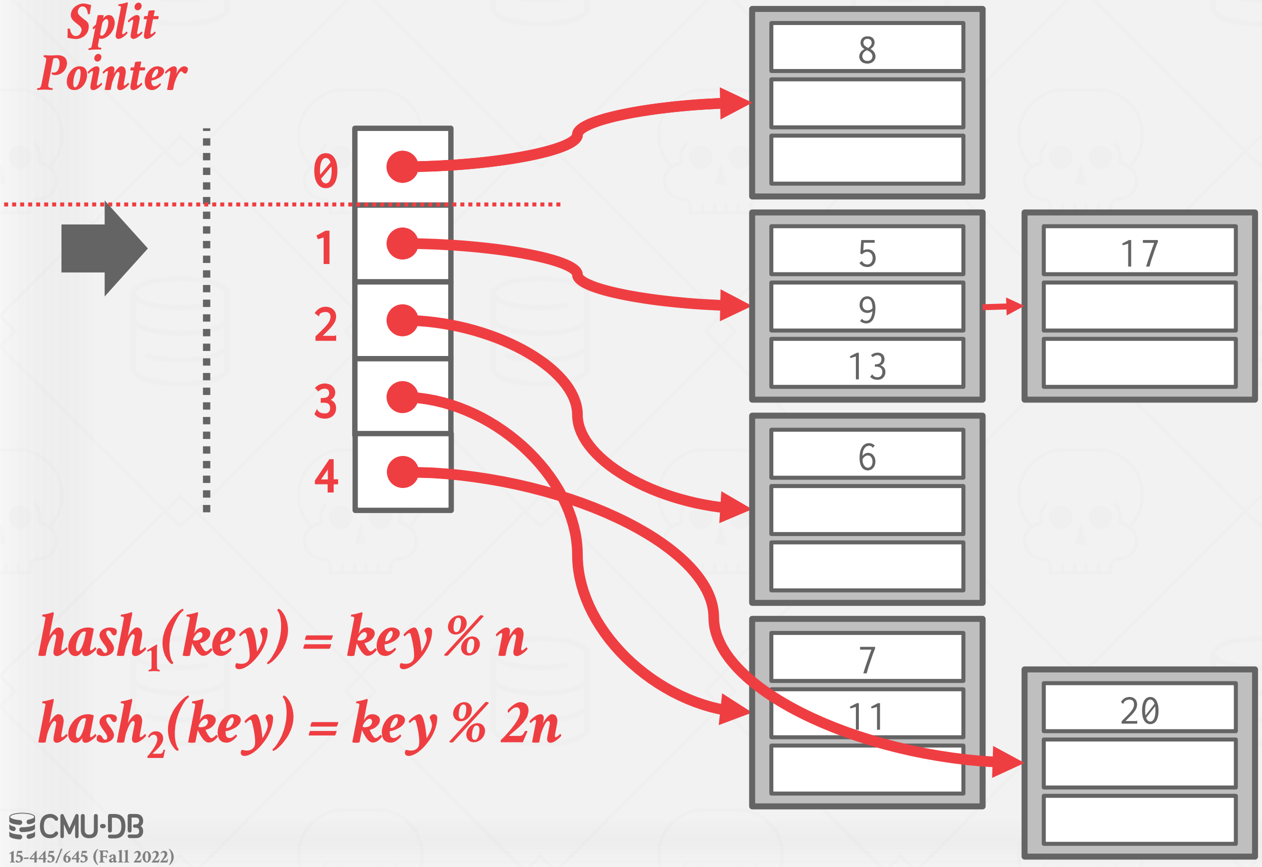
$hash_1(9) = 9 \% 4 = 1$

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.

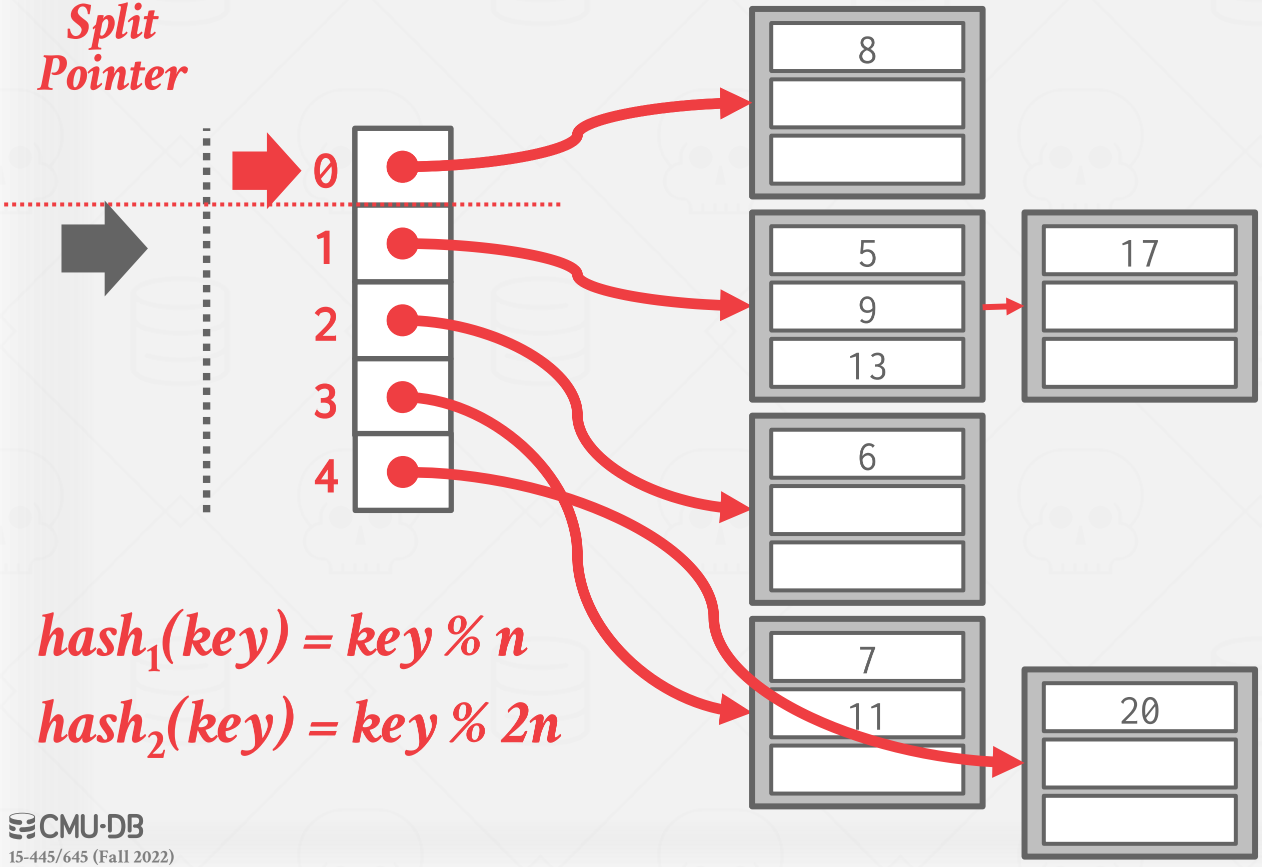
LINEAR HASHING – DELETES



$hash_1(key) = key \ \% \ n$

$hash_2(key) = key \ \% \ 2n$

LINEAR HASHING – DELETES



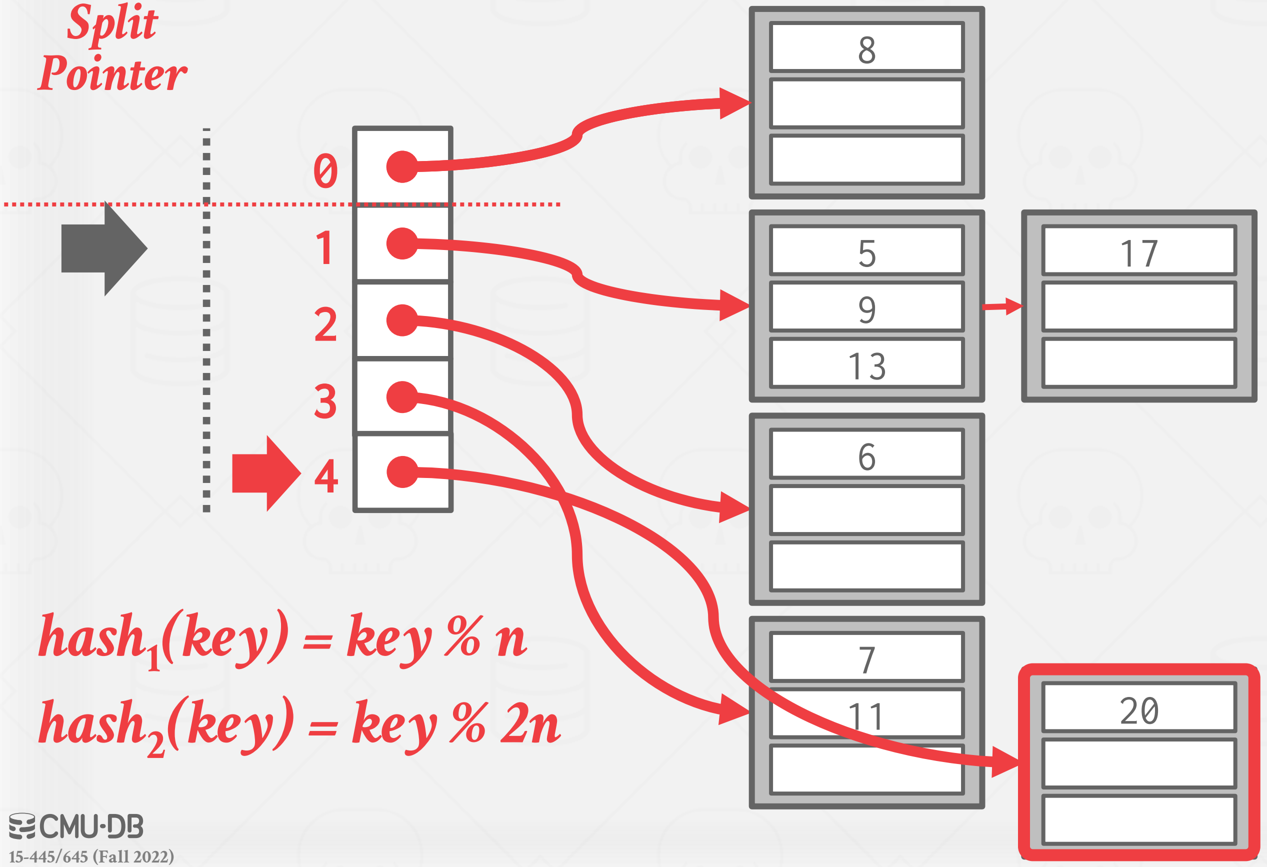
Delete 20

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING – DELETES



Delete 20

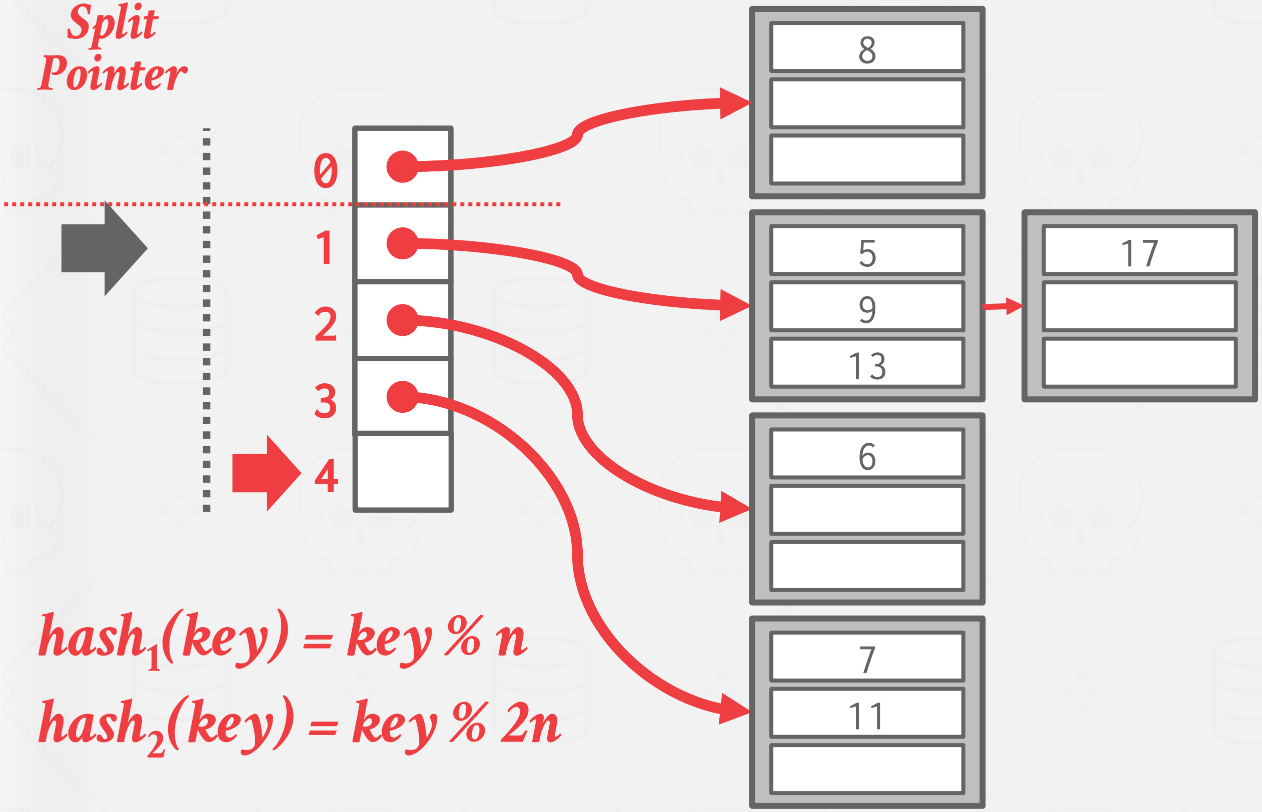
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING – DELETES



Delete 20

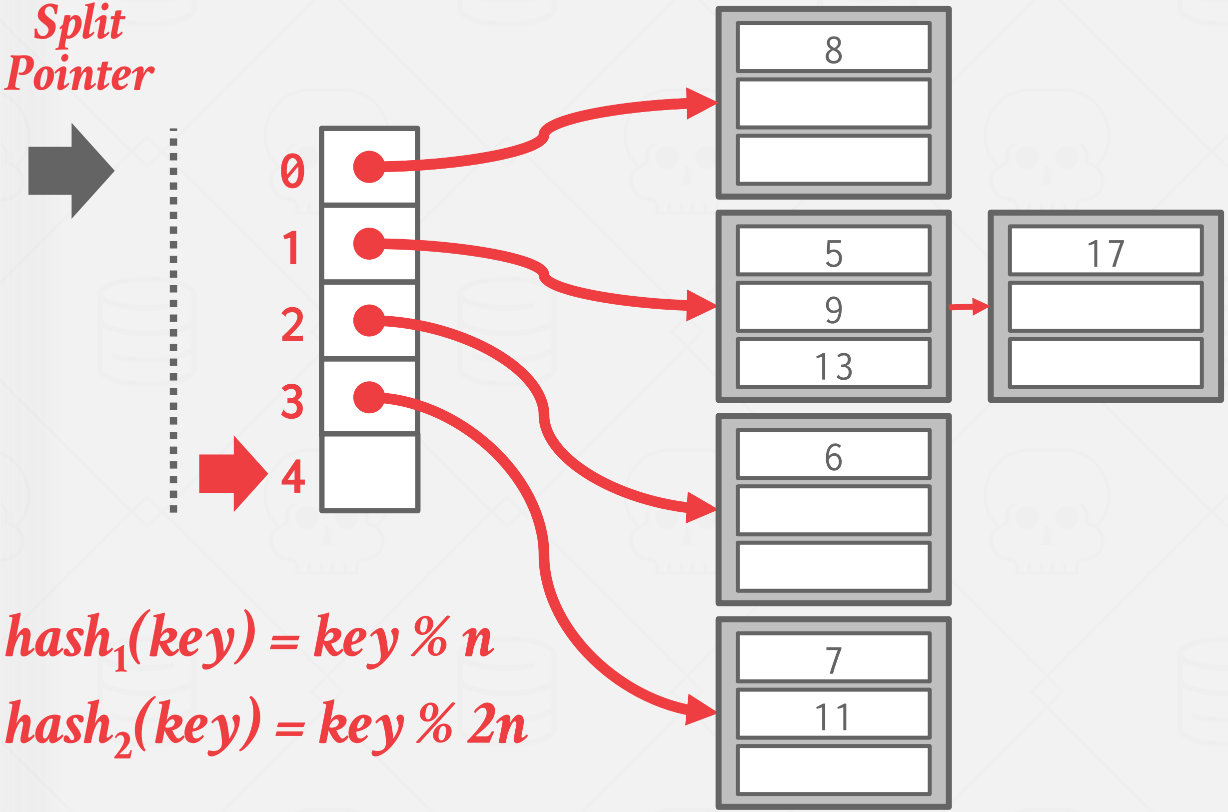
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING – DELETES



Delete 20

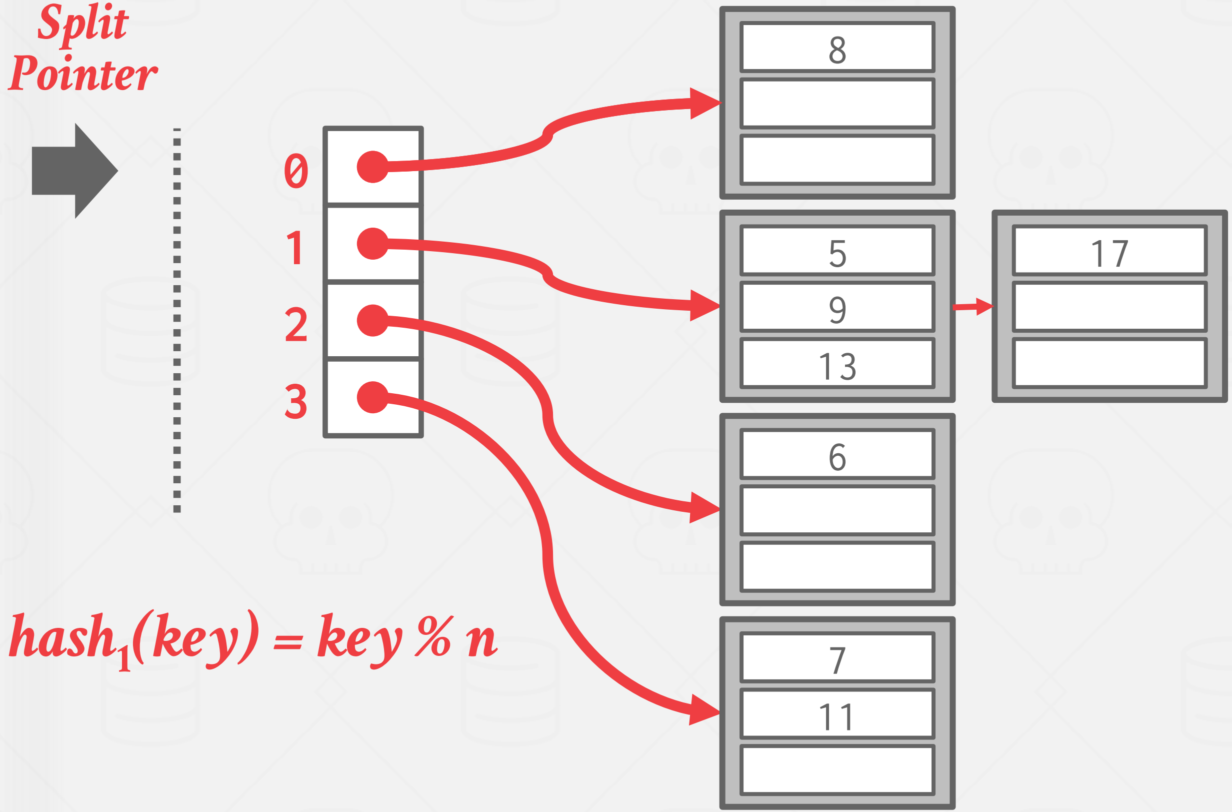
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

LINEAR HASHING – DELETES



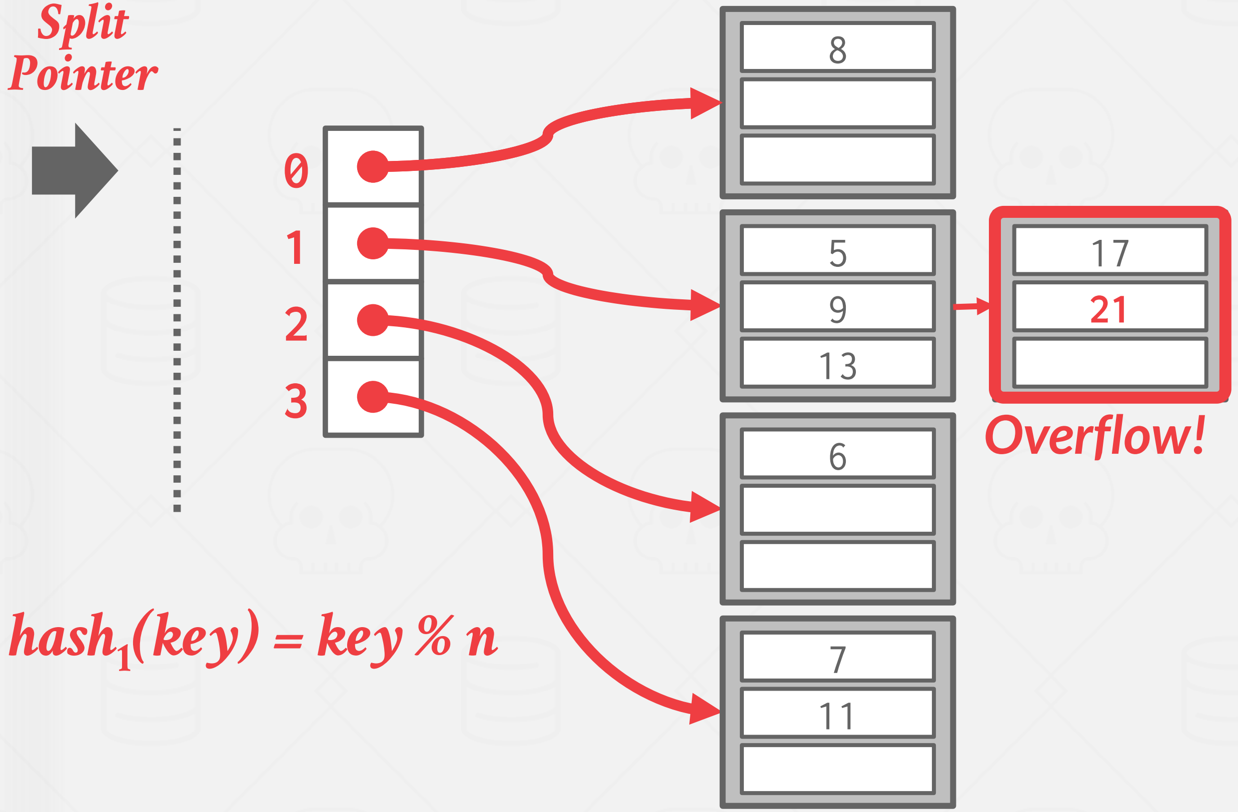
Delete 20

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

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

$hash_1(key) = key \% n$

LINEAR HASHING – DELETES



$hash_1(key) = key \% n$

Delete 20

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

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

Put 21

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

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"