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.

Concurrency means more than 1 thread is able to access data structure at same time without any problem.

Unordered

HASH TABLES (Hash map)

keys (arrays) are mapped to values.

A **hash table** implements an unordered associative array that maps keys to values.

We use hash function to find the key or offset.

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)** → To store the n elements we need array.

Time Complexity:

→ Average: **O(1)** ⬅ ***Databases need to care about constants!***

→ Worst: **O(n)**

↳ Due to collision if all no.s point to same offset.

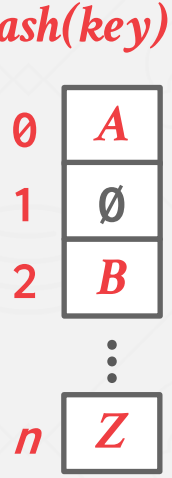
STATIC HASH TABLE

→ Static

Size doesn't change

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.



∴ N is fixed.

key = hash(value) % N → hash fxn used.

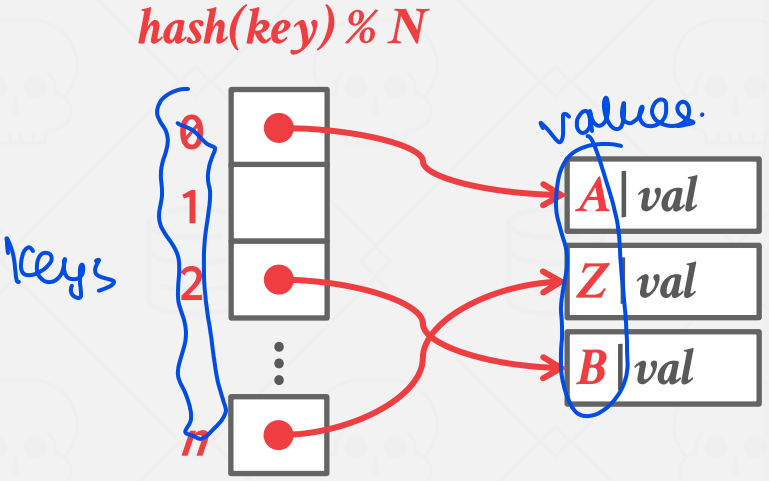
i.e.

∴ static N is fixed we allocate a giant array to store element.

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.



→ Fixed size

→ unique elements

ASSUMPTIONS

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

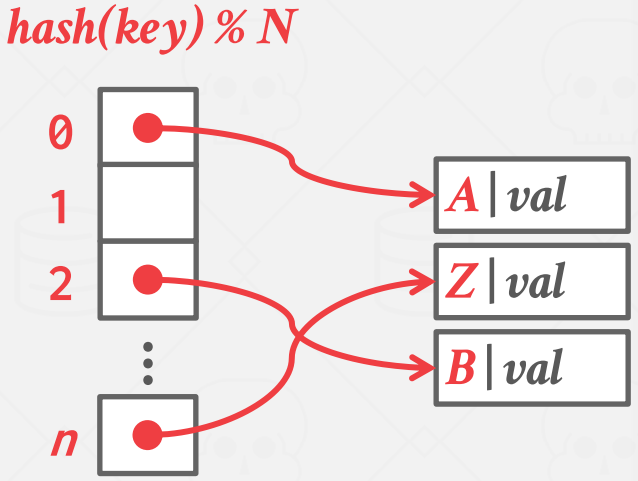
← (Static definition itself)

Assumption #2: Each key is unique.

(No collision) ↓

Assumption #3: Perfect hash function.

→ If key1≠key2, then hash(key1)≠hash(key2)



∴ Perfect hash fxn.

i.e. 5 ≠ 10 but if N=5 hash(5) = 5 % 5 = 0

= hash(10 % 5)

∴ not perfect hash fxn here.

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

For DBMS hash tables we just use normal hashing usage of cryptographic hashing will make it more complex.

HASH FUNCTIONS

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

↳ offset has to be returned for the input 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.

Hence we don't prefer cryptographic hashing

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.

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

Approach #3: Cuckoo Hashing

If collision happens we will linearly search for the next free slot in table.

LINEAR PROBE HASHING

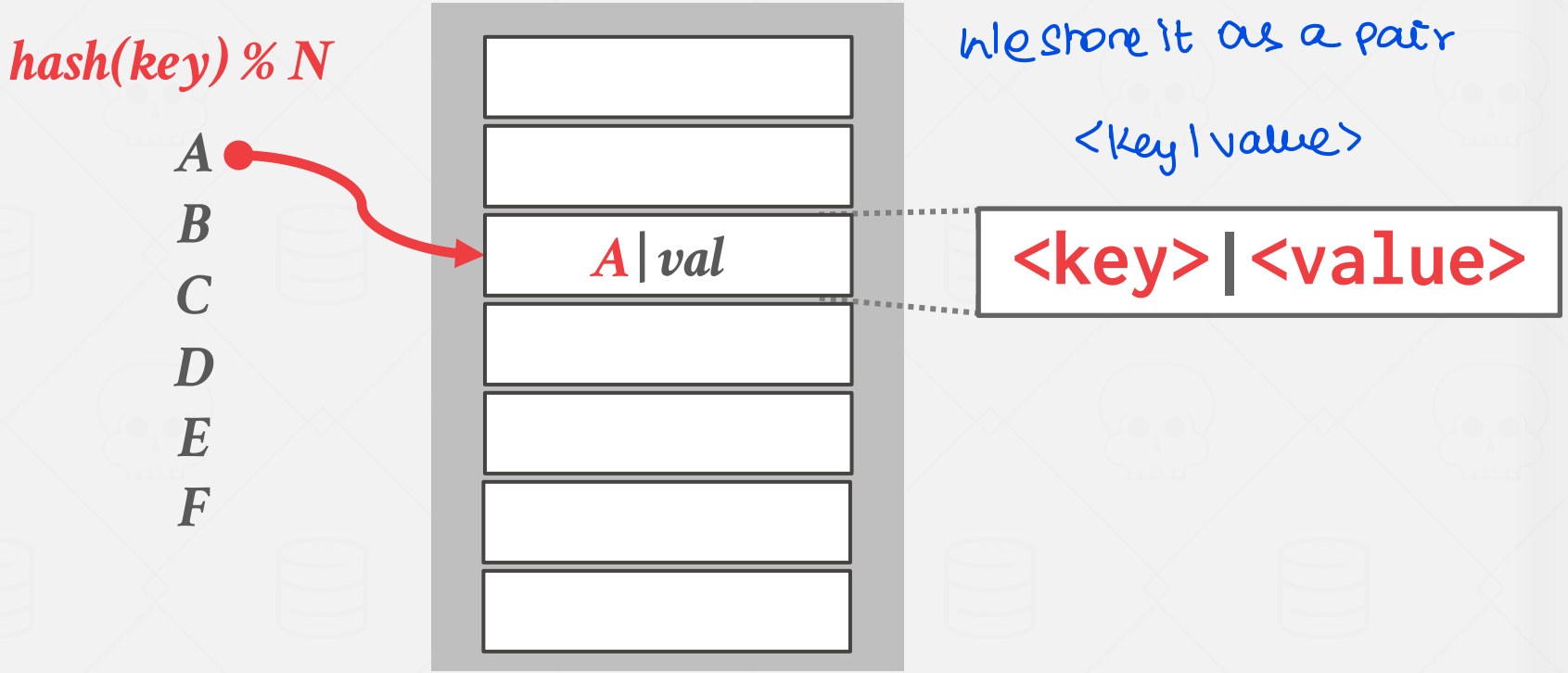
Single giant table of slots. (Similar to static Hashing).

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

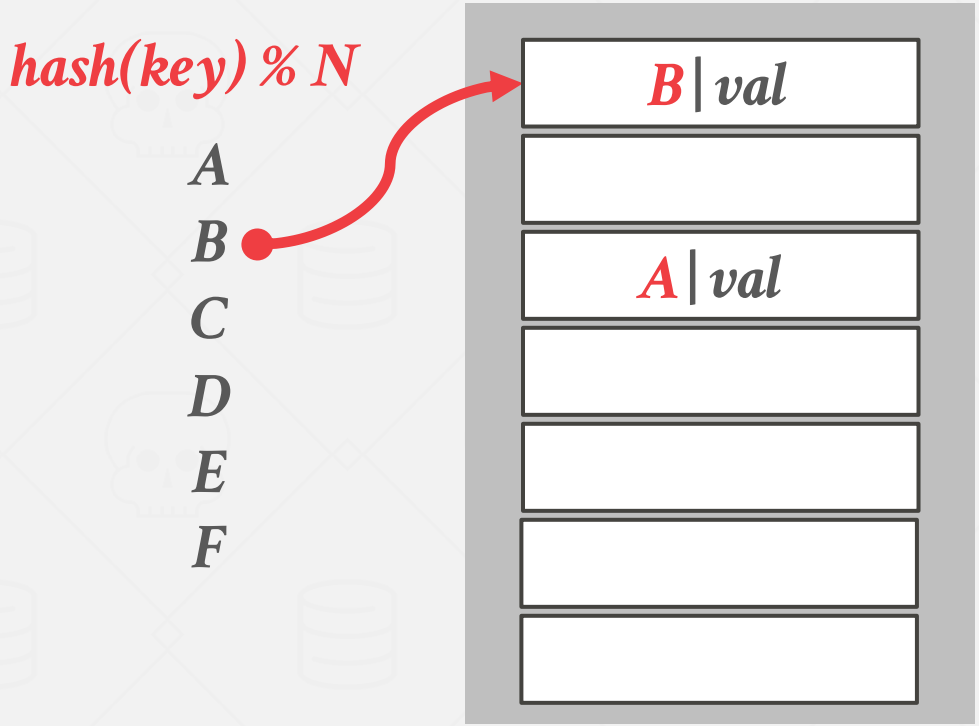
→ To determine whether an element is present, hash to a location in the index and scan for it.
→ Must store the key in the index to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.

search(value)? offset = hash(value) go to that offset & scan for it.

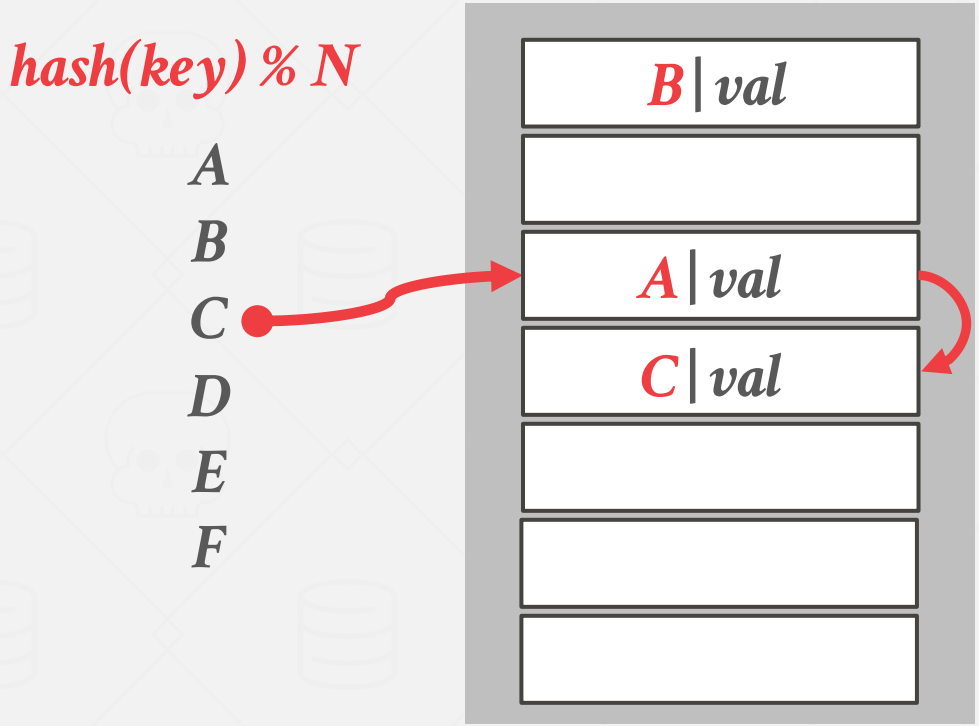
LINEAR PROBE HASHING



LINEAR PROBE HASHING



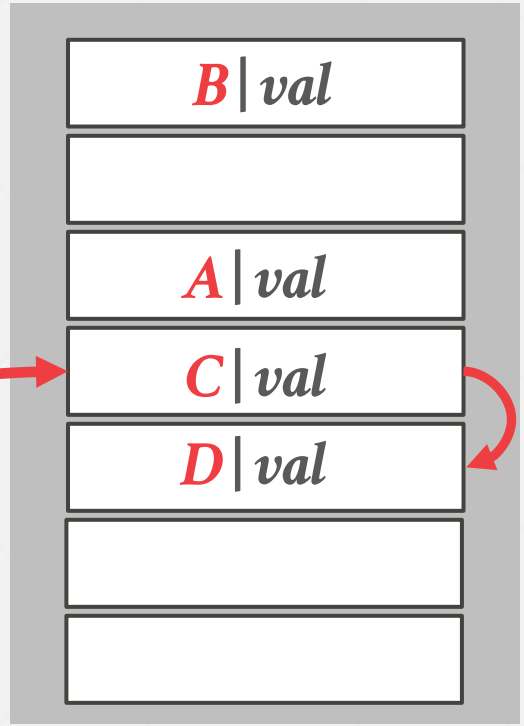
LINEAR PROBE HASHING



LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F



In search let us search for x it is at key 'i'

now we go to i as per hash fxn but it is not there

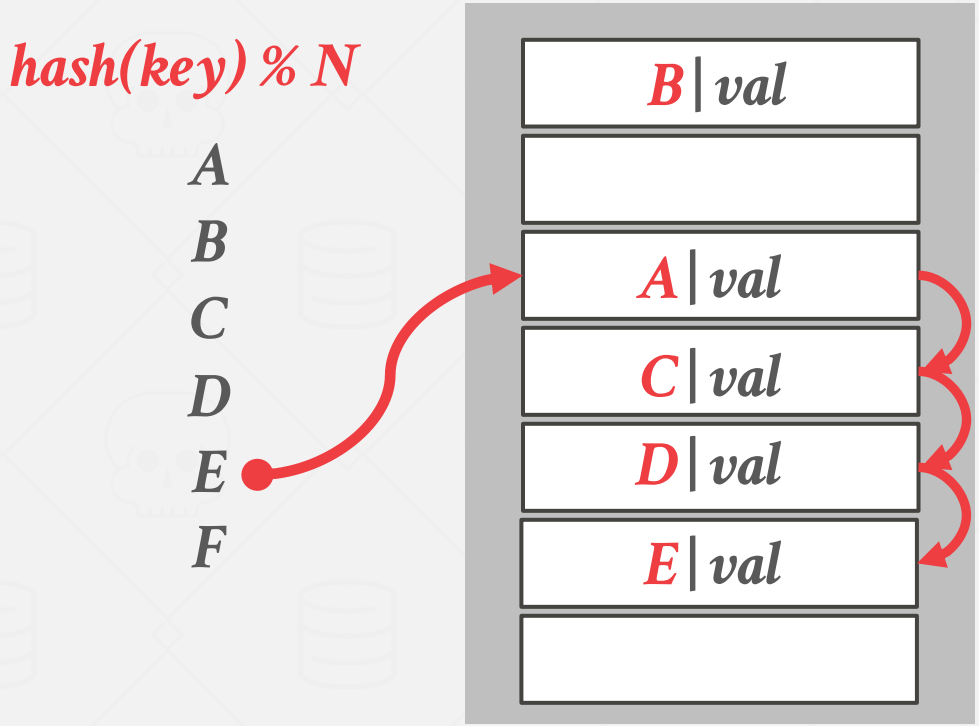
we search whole table.

we don't find x how to stop?

∴ If the index = i then it means we reached start again.

∴ stop. or middle if we find x we stop our search there.

LINEAR PROBE HASHING

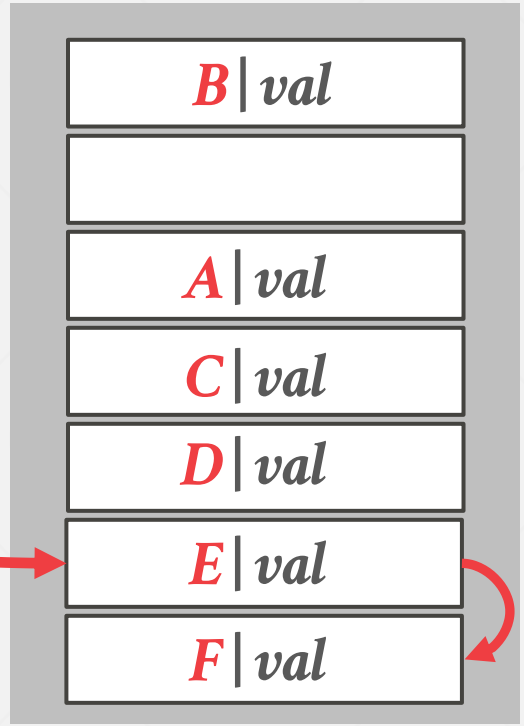


See we are searching for next available free space in case of collision

LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

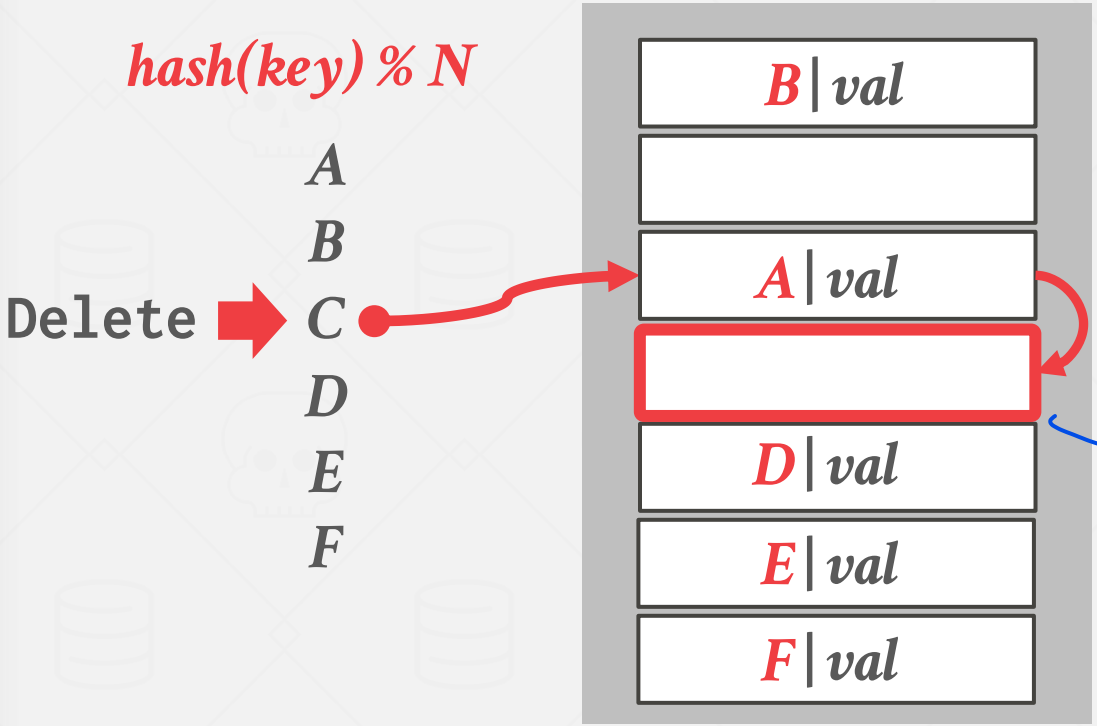


In lookup or Search we go to offset calc. by hash fxn and starting from there.

either until we find Empty slot or find the element.

So if delete removes element causing white space then there would be problem for lookup?

LINEAR PROBE HASHING – DELETES



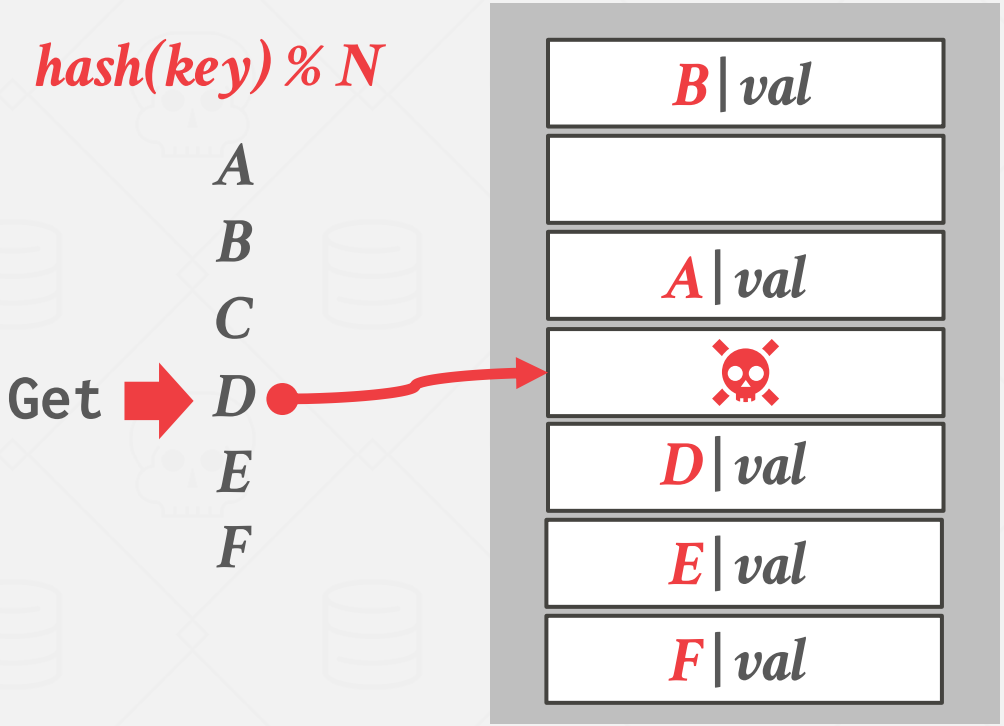
What happens during Delete?

We deleted C now we are searching for d.

As per hash it points here given in above slide.

But there we have empty slot- we stop search.

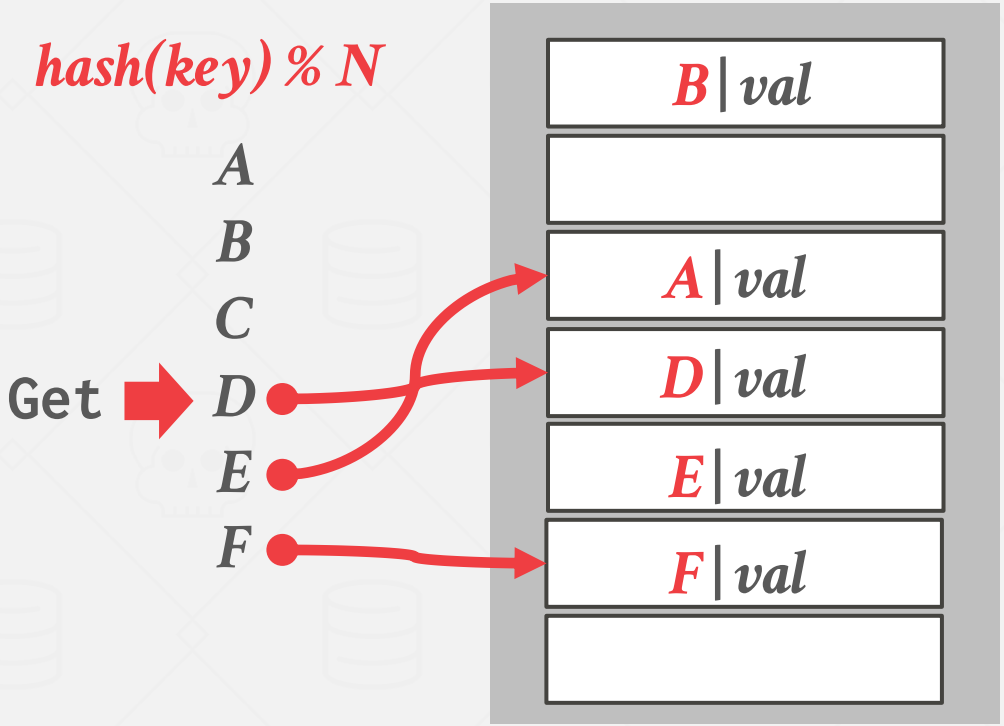
LINEAR PROBE HASHING – DELETES



∴ Saying D not found but actually D is below the empty space.

∴ Search gives wrong answer now

LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

→ Nobody actually does this.

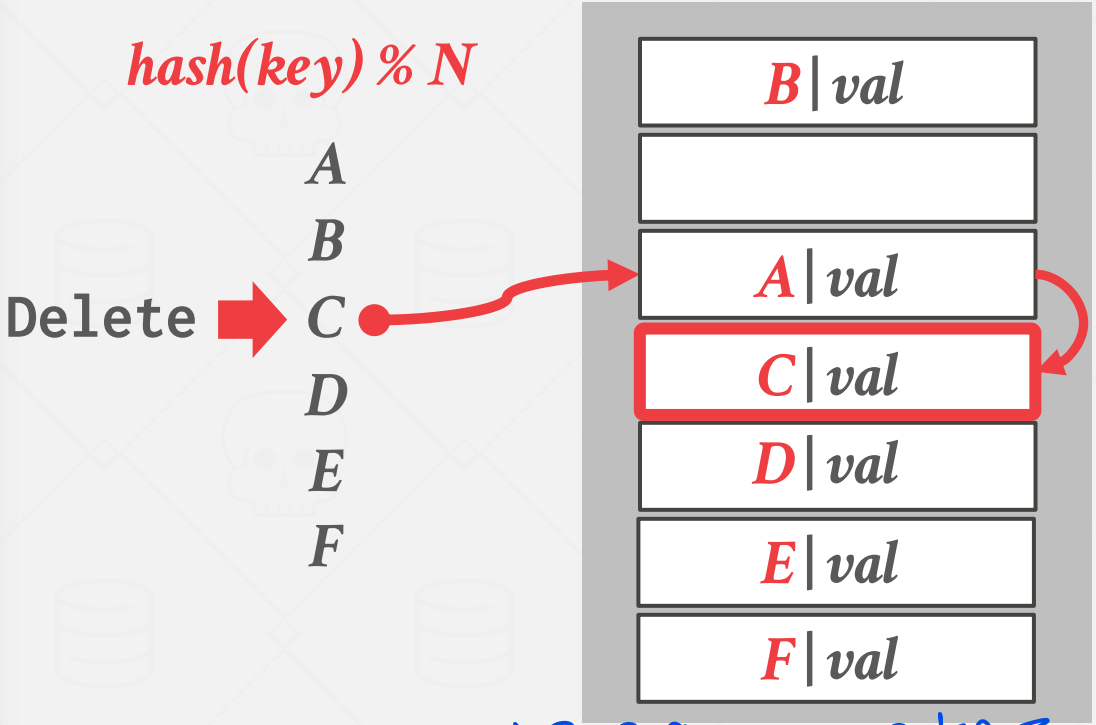
Move all keys DEF up. This is never preferred.

In tombstone we use marker to denote deleted keys. Later we

LINEAR PROBE HASHING – DELETES

cause

these slots too.



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.

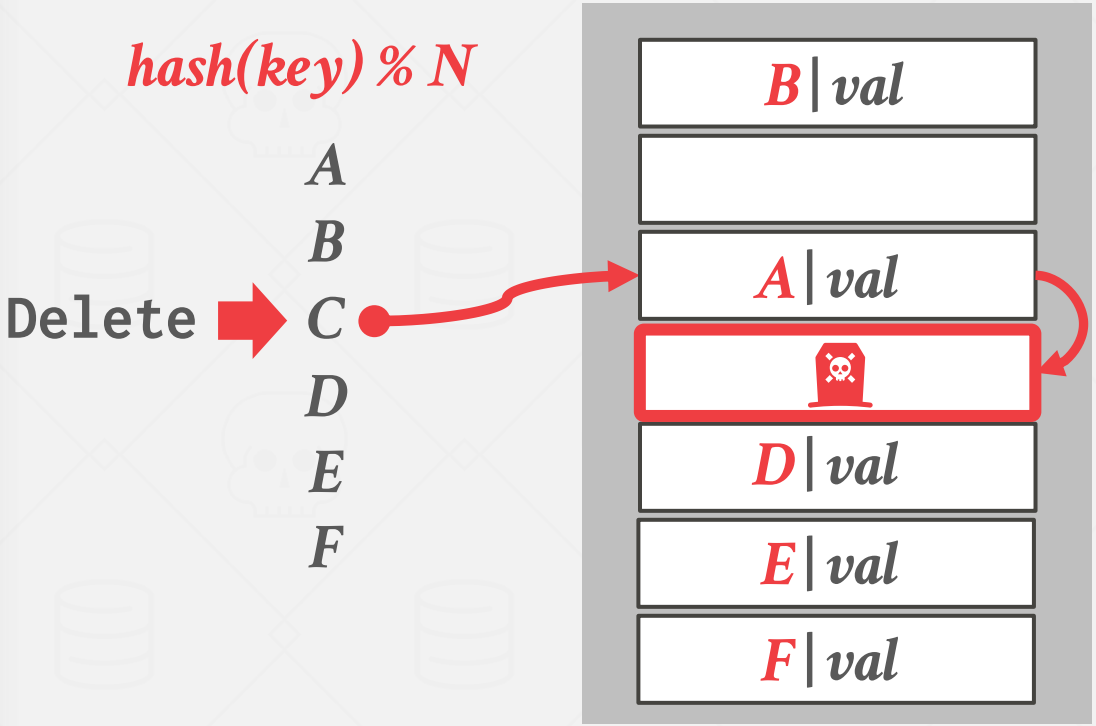
→ May need periodic garbage collection.

But the trade-off is

deletes ↑ making many markers then we may have to

do garbage collection to collect them.

LINEAR PROBE HASHING – DELETES



Approach #2: Tombstone

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

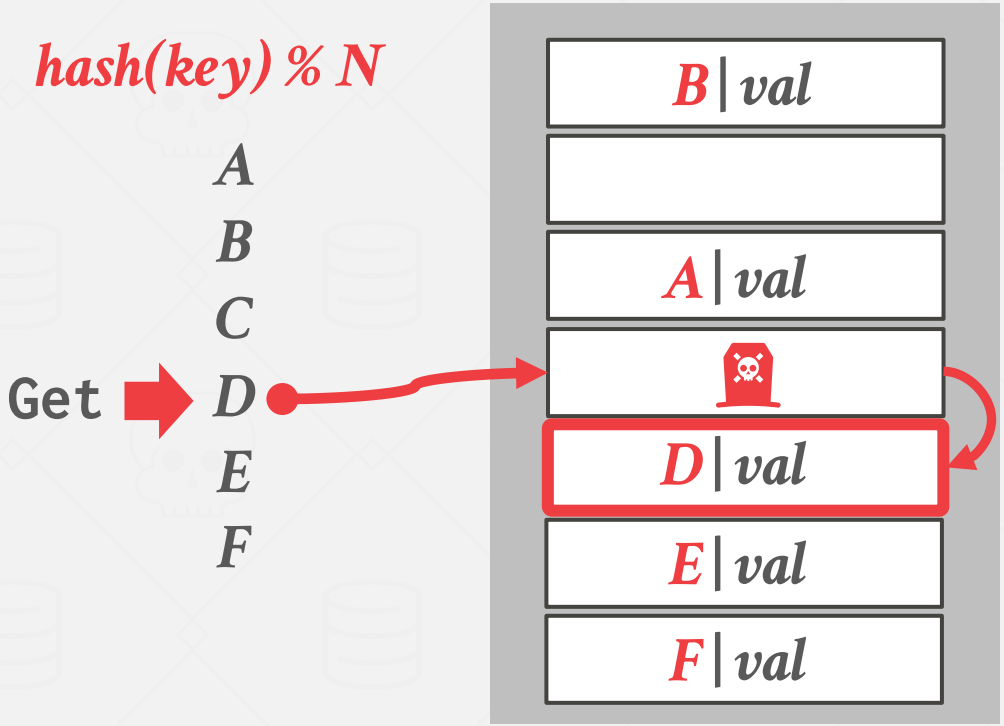
→ You can reuse the slot for new keys.

→ May need periodic garbage collection.

In this case at search if certain slot if we

find marker then we just move next to it we don't stop there.

LINEAR PROBE HASHING – DELETES



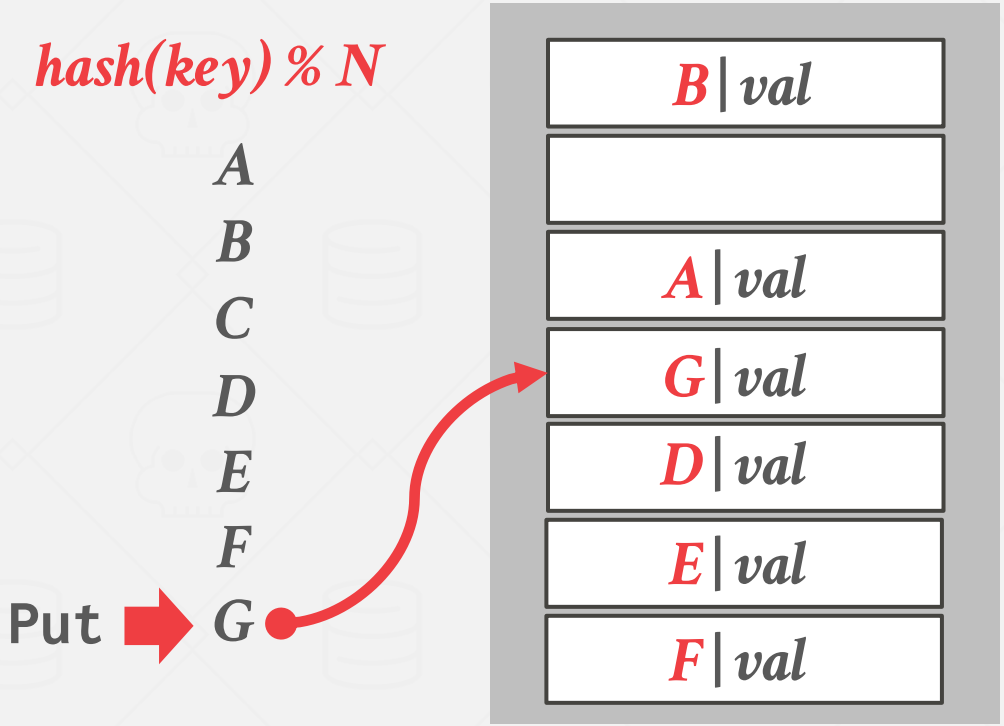
Approach #2: Tombstone

→ 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.

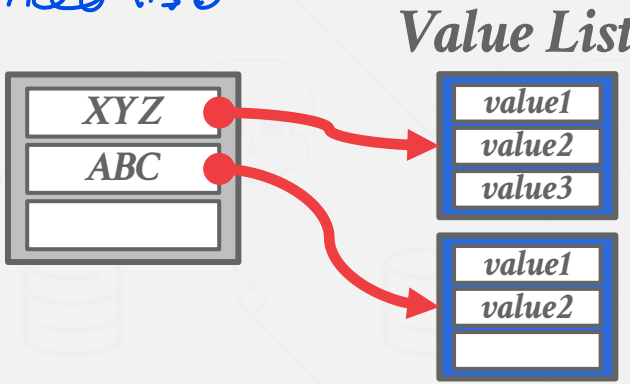
If a key has more than 1 value then we store all those values of single key in a seperate linked list.

NON-UNIQUE KEYS

Choice #1: Separate Linked List

→ Store values in separate storage area for each key.

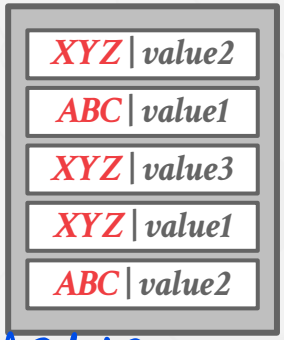
What are the examples?



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.



Redundant => Duplicate keys. Store duplicate keys ourselves in hash table

A variant of Linear Probe Hashing that steals slots from rich keys & give them to poor keys.

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.

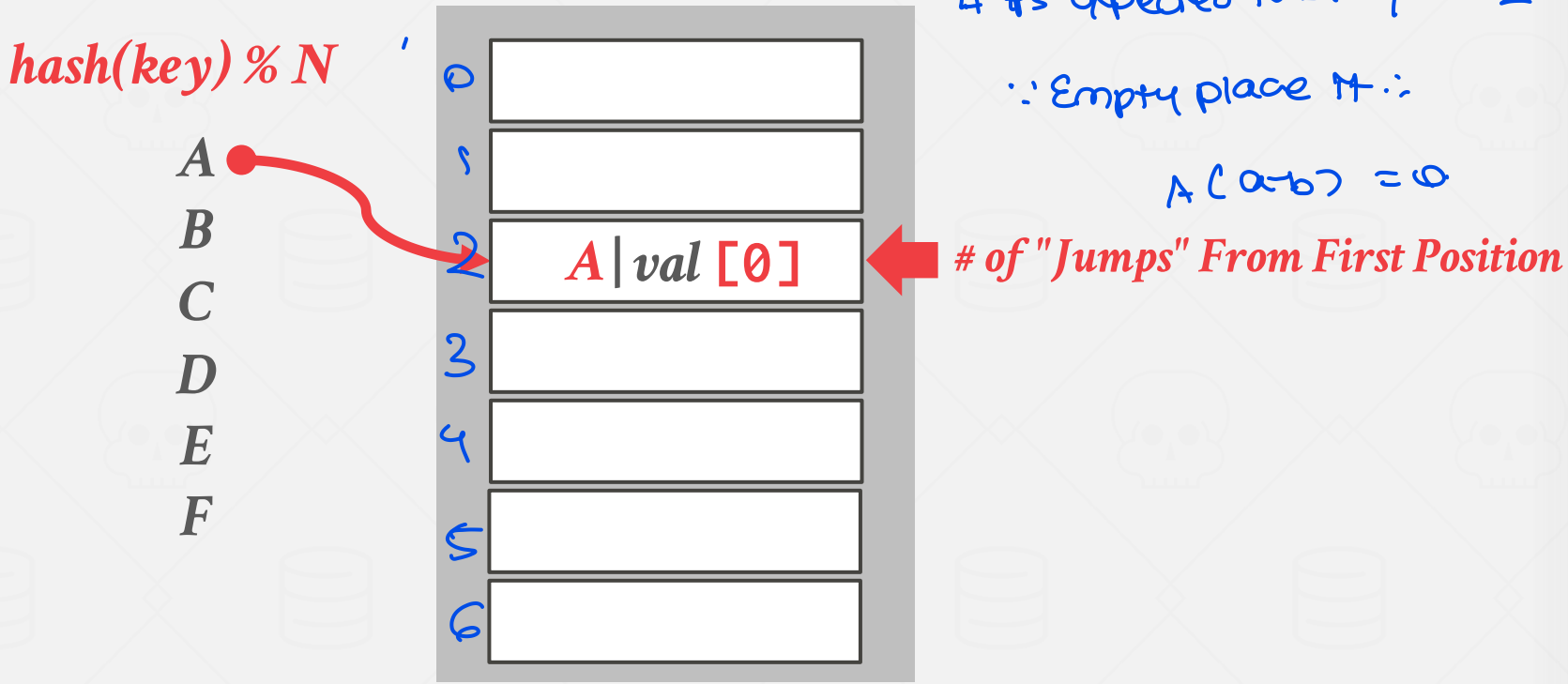
→ i.e. we find expected position from hash fxn. Let it be a.

During insert a key takes position of another key if its (a-b) is larger than the 2nd key's (a-b).

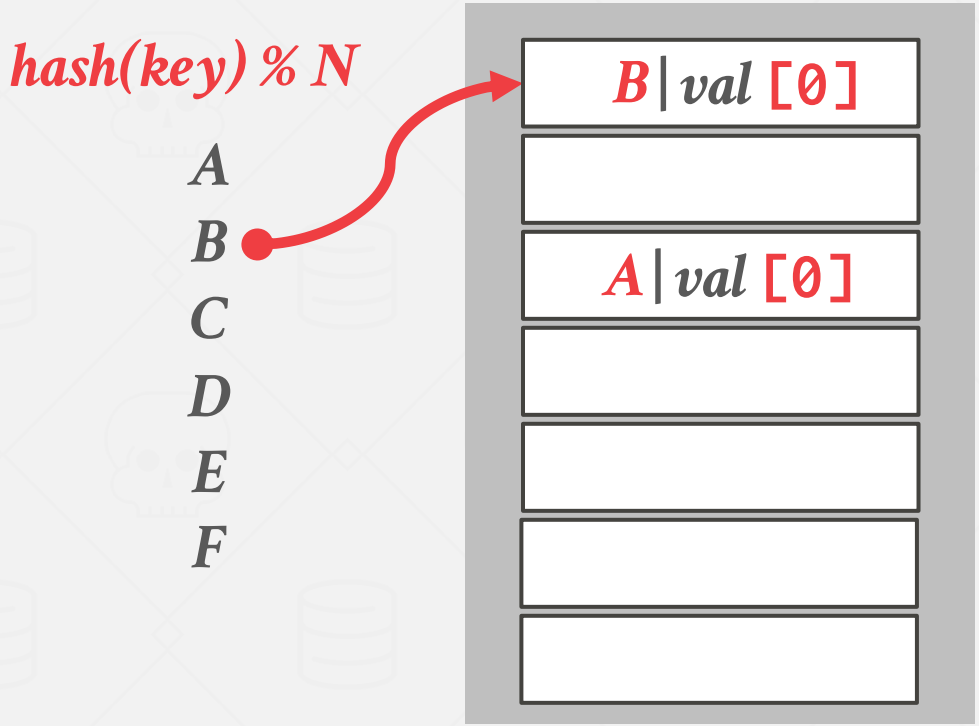
But it is somewhere at b.

∴ we keep track of this (a-b) for all keys.

ROBIN HOOD HASHING



ROBIN HOOD HASHING

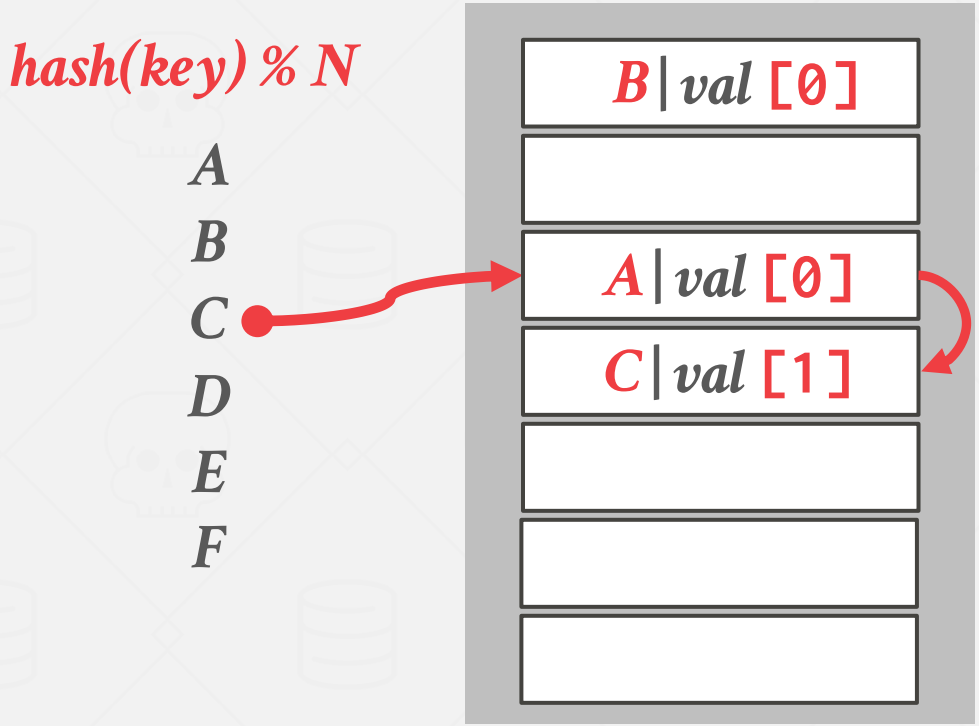


placeB → as per hfxn,

B should be at 0 if I3

∴ B (at 0) = 0

ROBIN HOOD HASHING



C should be at 2 but not there

∴ Insert(C) is happening.

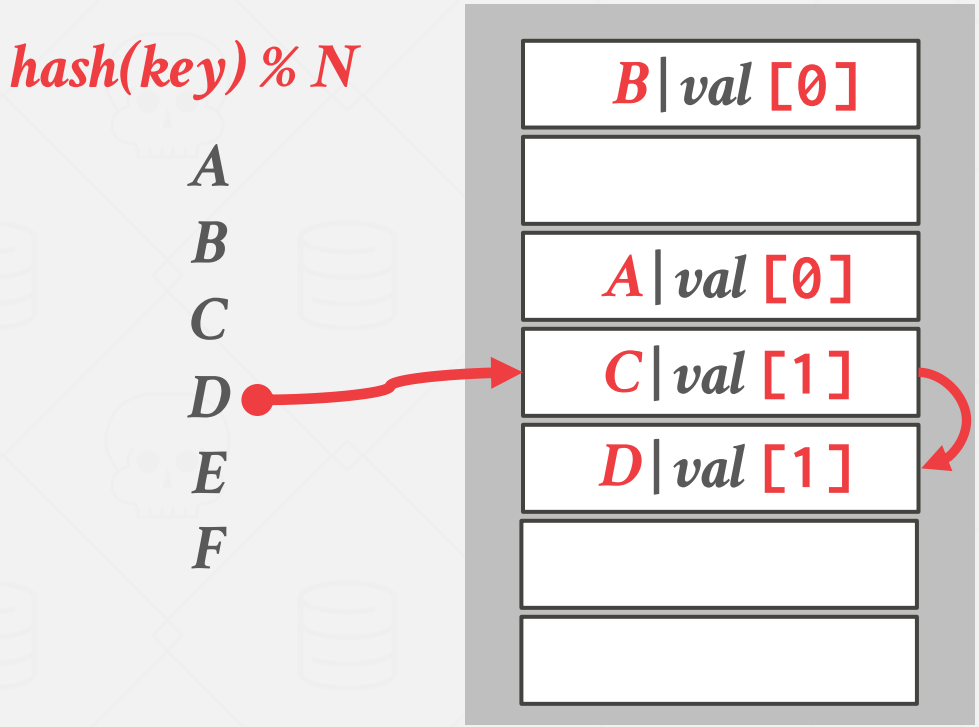
C takes position of A

if C(a-b) > A(a-b)

both zero. ∴ no

C(a-b) ↑ C goes down

ROBIN HOOD HASHING



Insert C Now

Insert D.

D(a-b) = 0.

Collision key → C

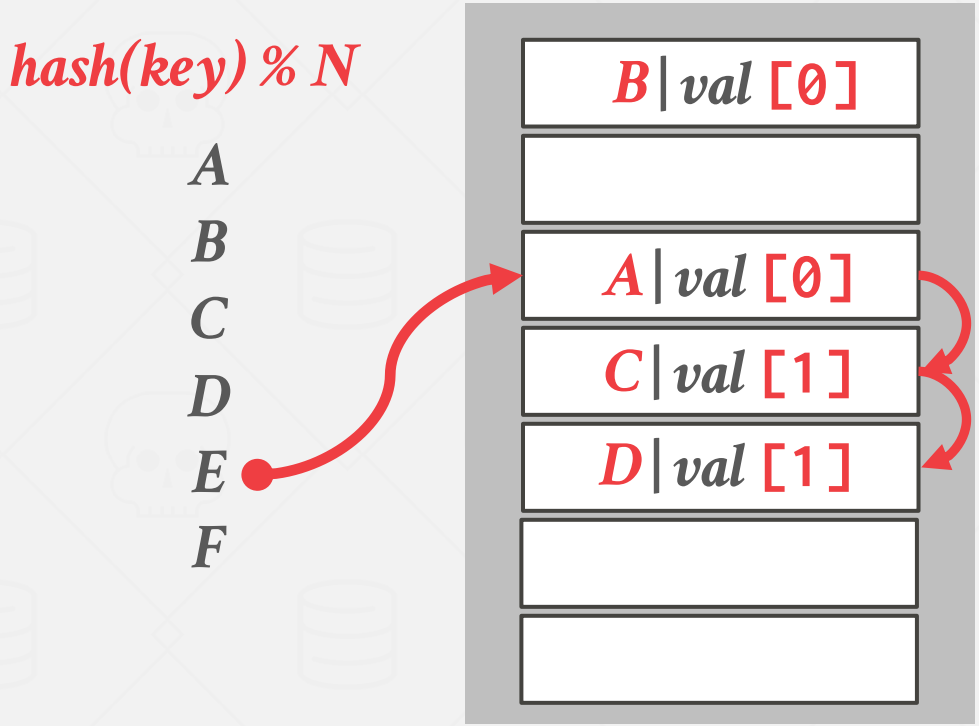
C(a-b) = 1

0 > 1

∴ D goes down ∵ it cannot take C's position

My increment D's (a-b)

ROBIN HOOD HASHING



Insert E.

Collision → A

E(a→b):=0 A(a→b):=0

0 > 0 ✗.

E(a→b) → 1

Collision → C

1 > 1 ✗.

E(a→b) → 2

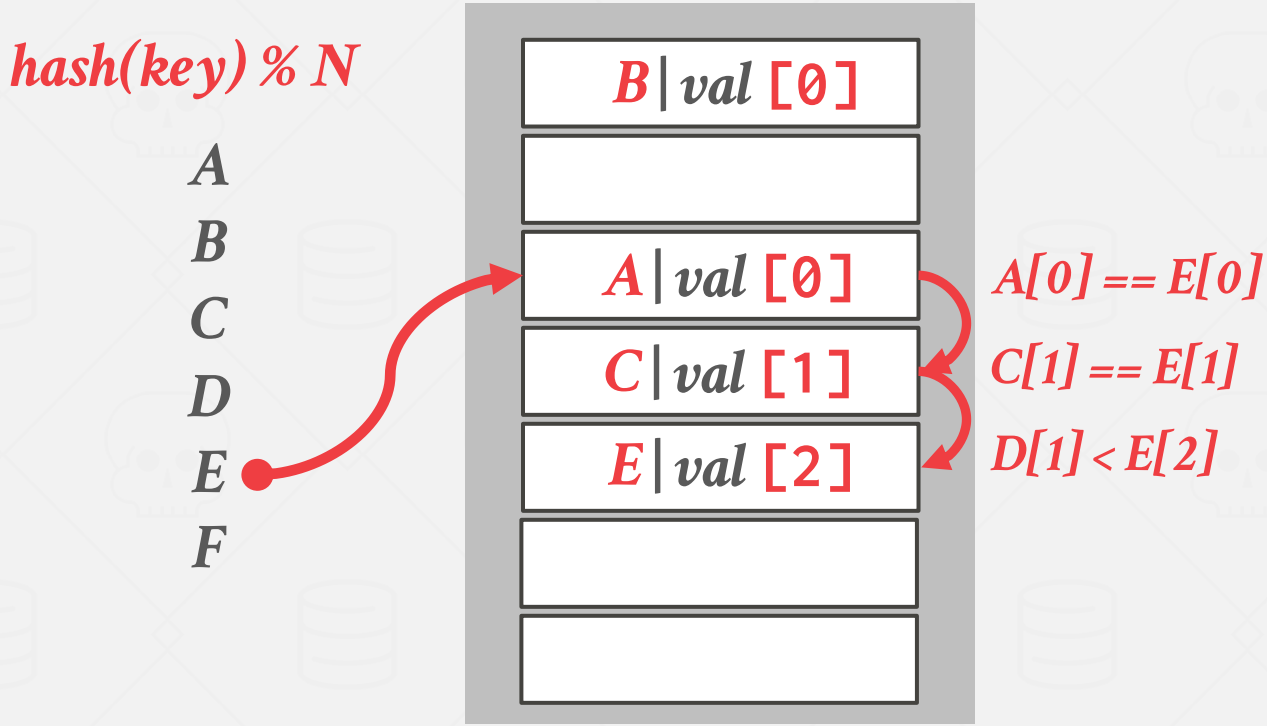
Collision → D.

2 > 1 ∴ E takes position of D

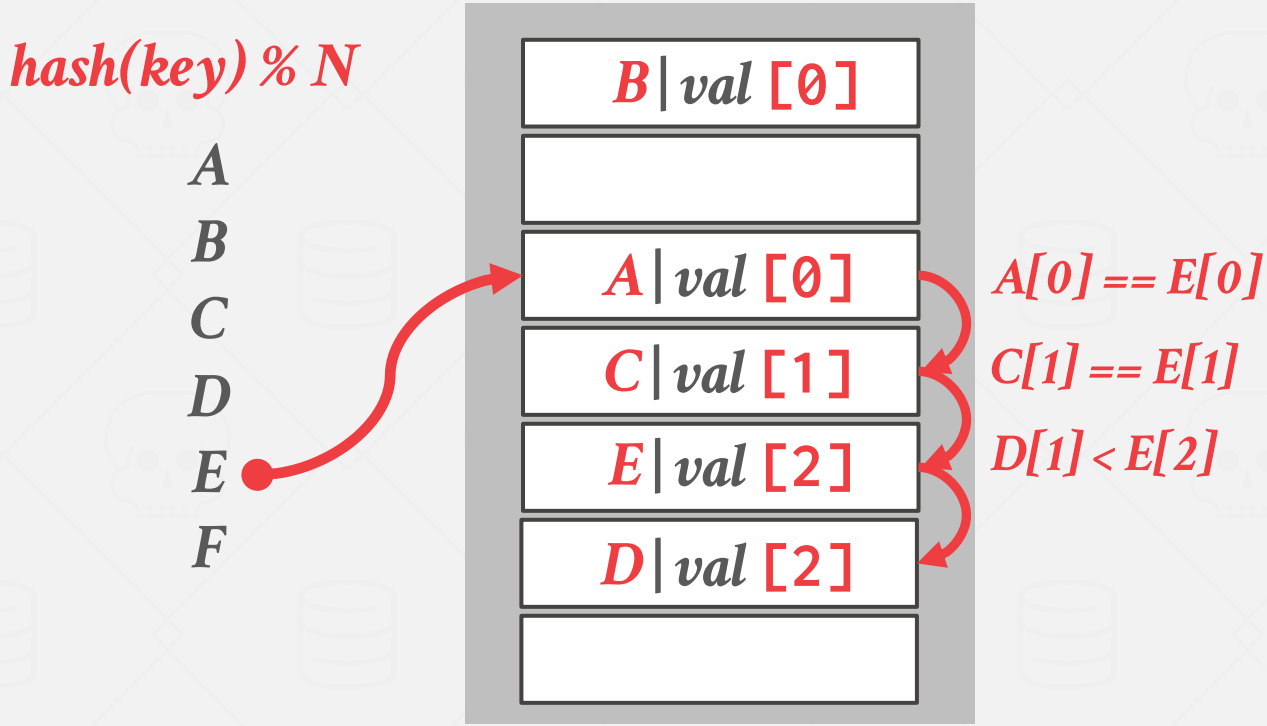
D moves down

Subsequently D(a→b) also changes

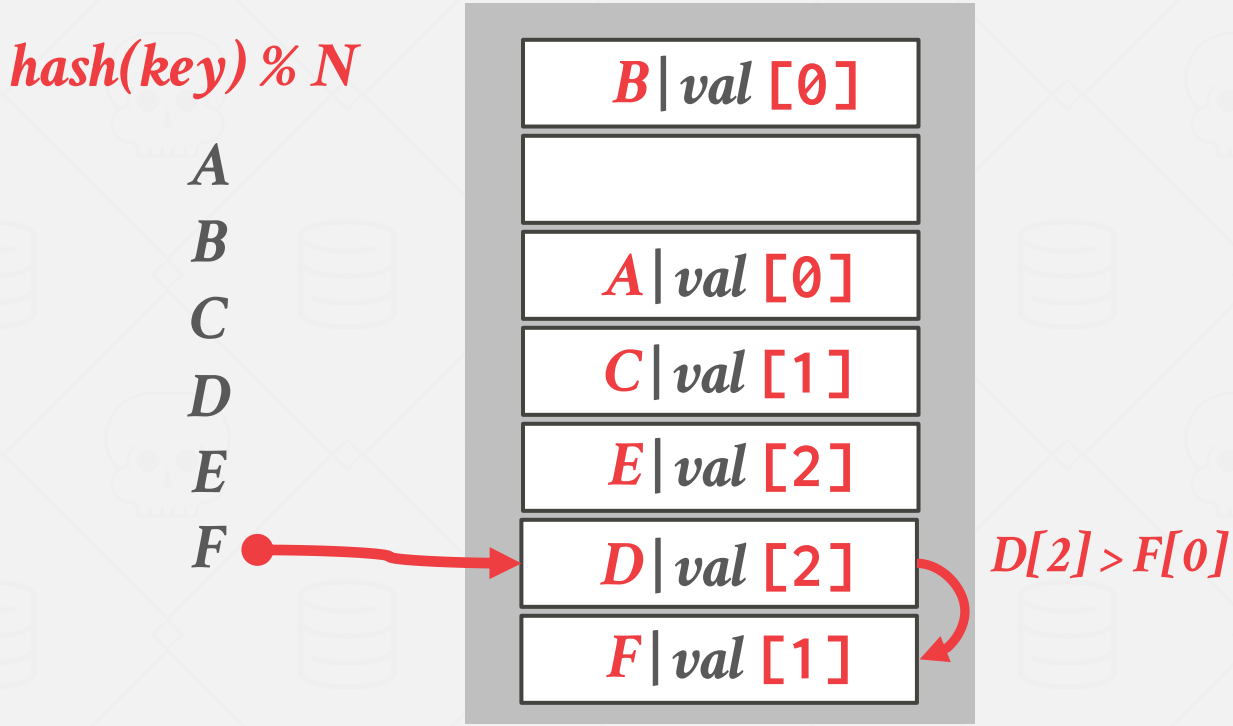
ROBIN HOOD HASHING



ROBIN HOOD HASHING



ROBIN HOOD HASHING



We use multiple hash tables & each table has its own hash function.

CUCKOO HASHING

On insert check every table & pick up the available free slot.

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.

If no table has free slot, evict an element from one of them & place it there.

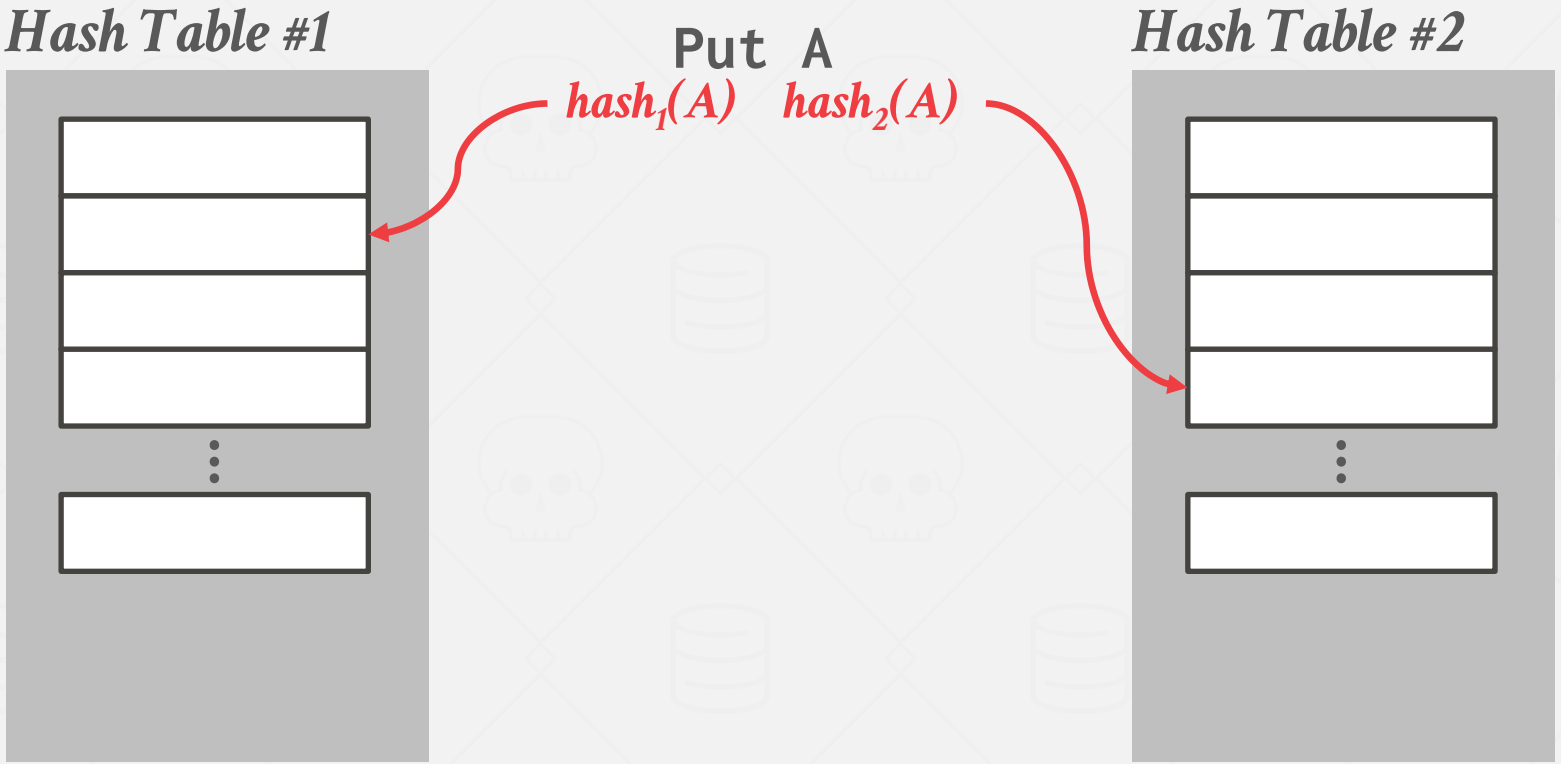
Look-ups and deletions are always **O(1)** because only one location per hash table is checked.

Lookups & deletions are O(1) ∵ only one location in each table is checked.

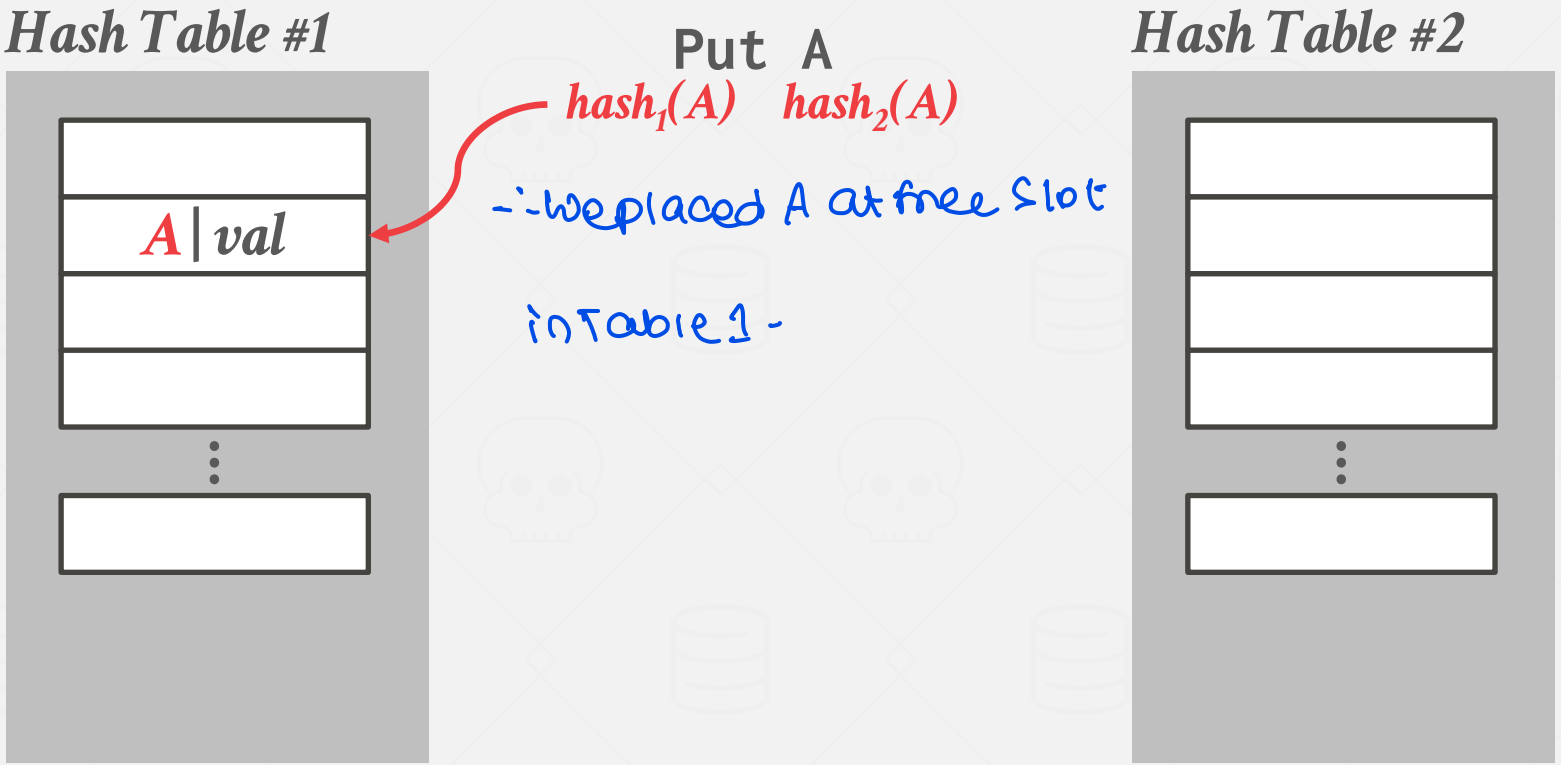
Removed ele needs to be rehashed to new location

Best open-source implementation is from CMU.

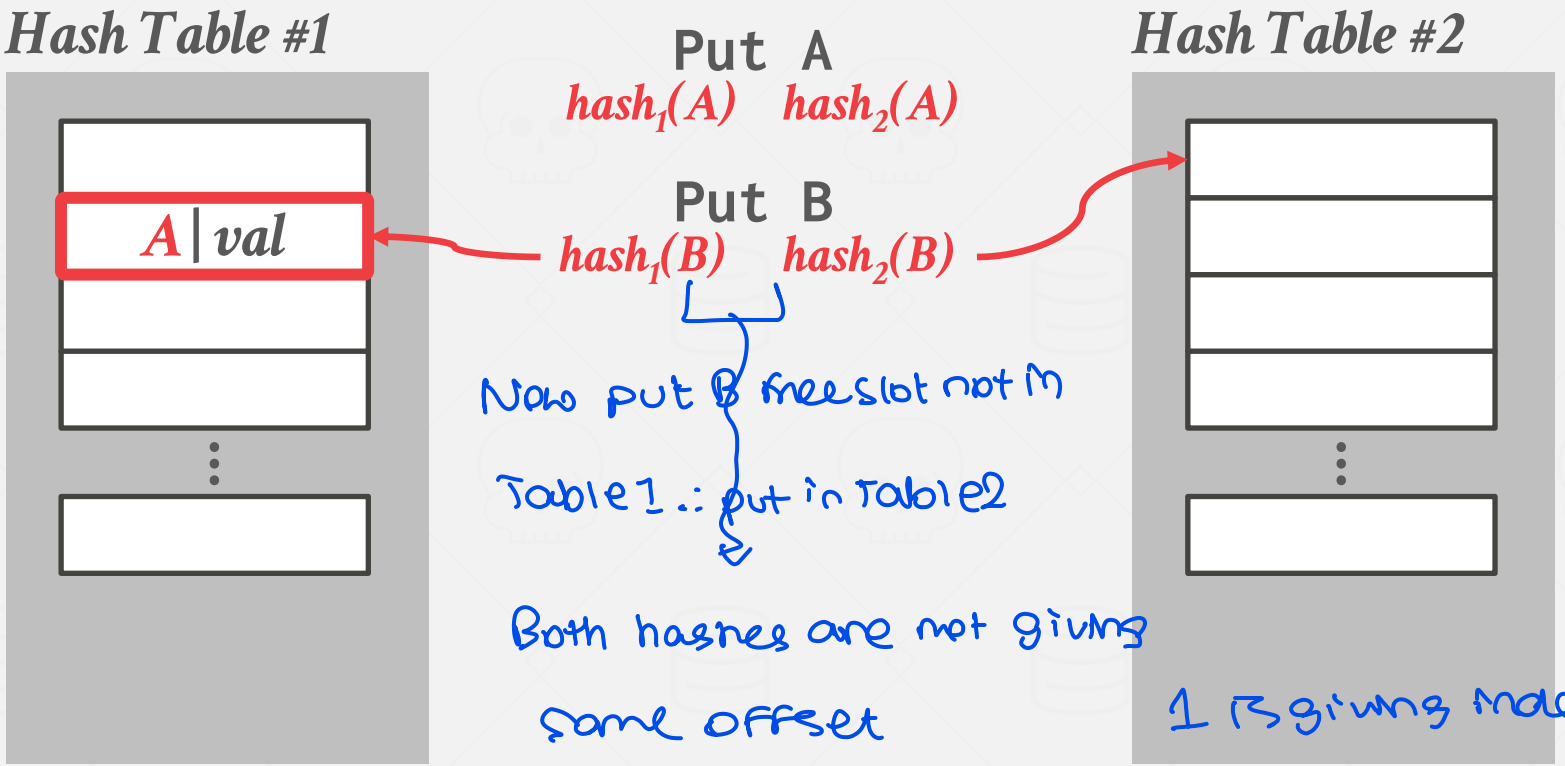
CUCKOO HASHING



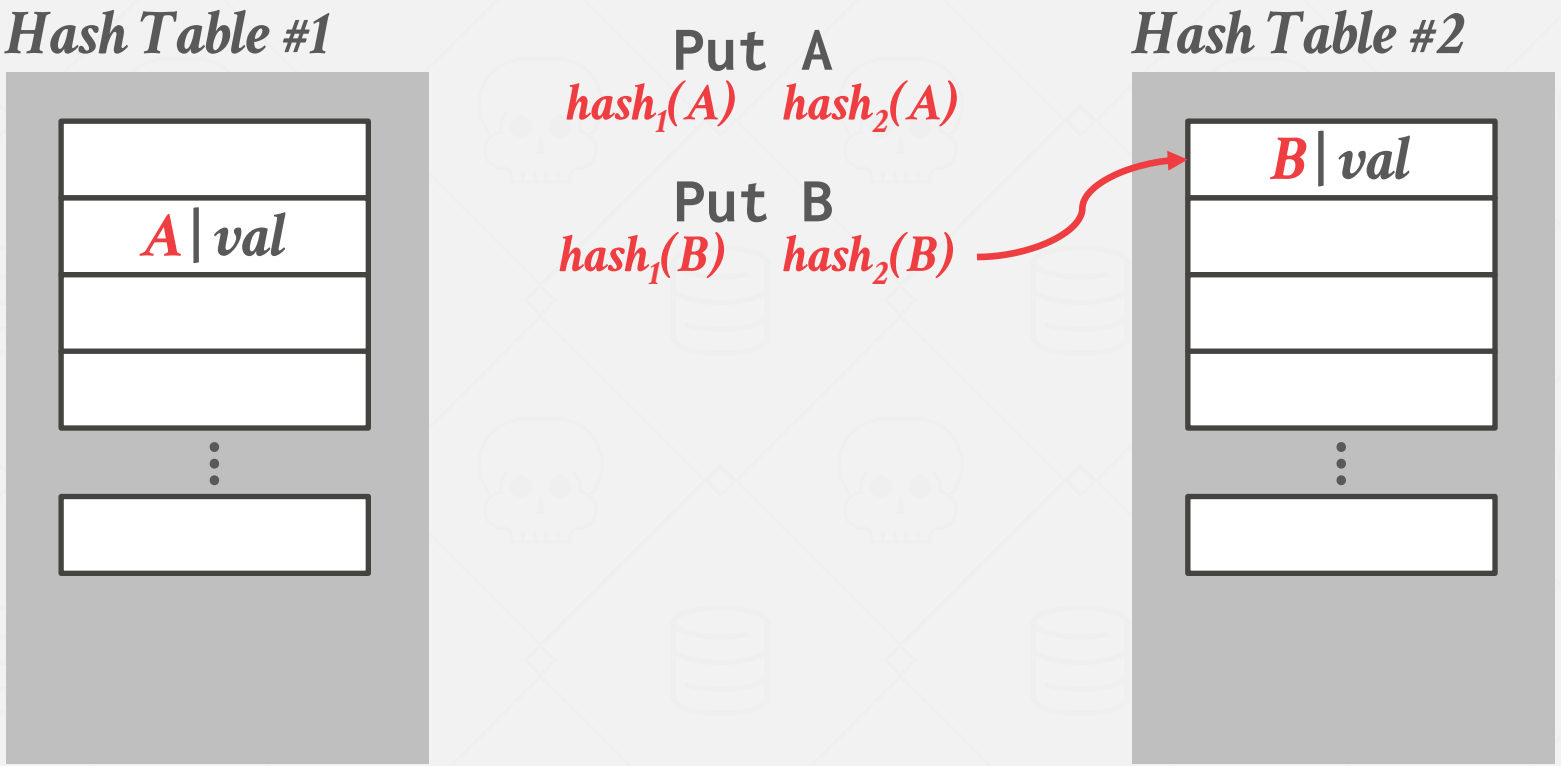
CUCKOO HASHING



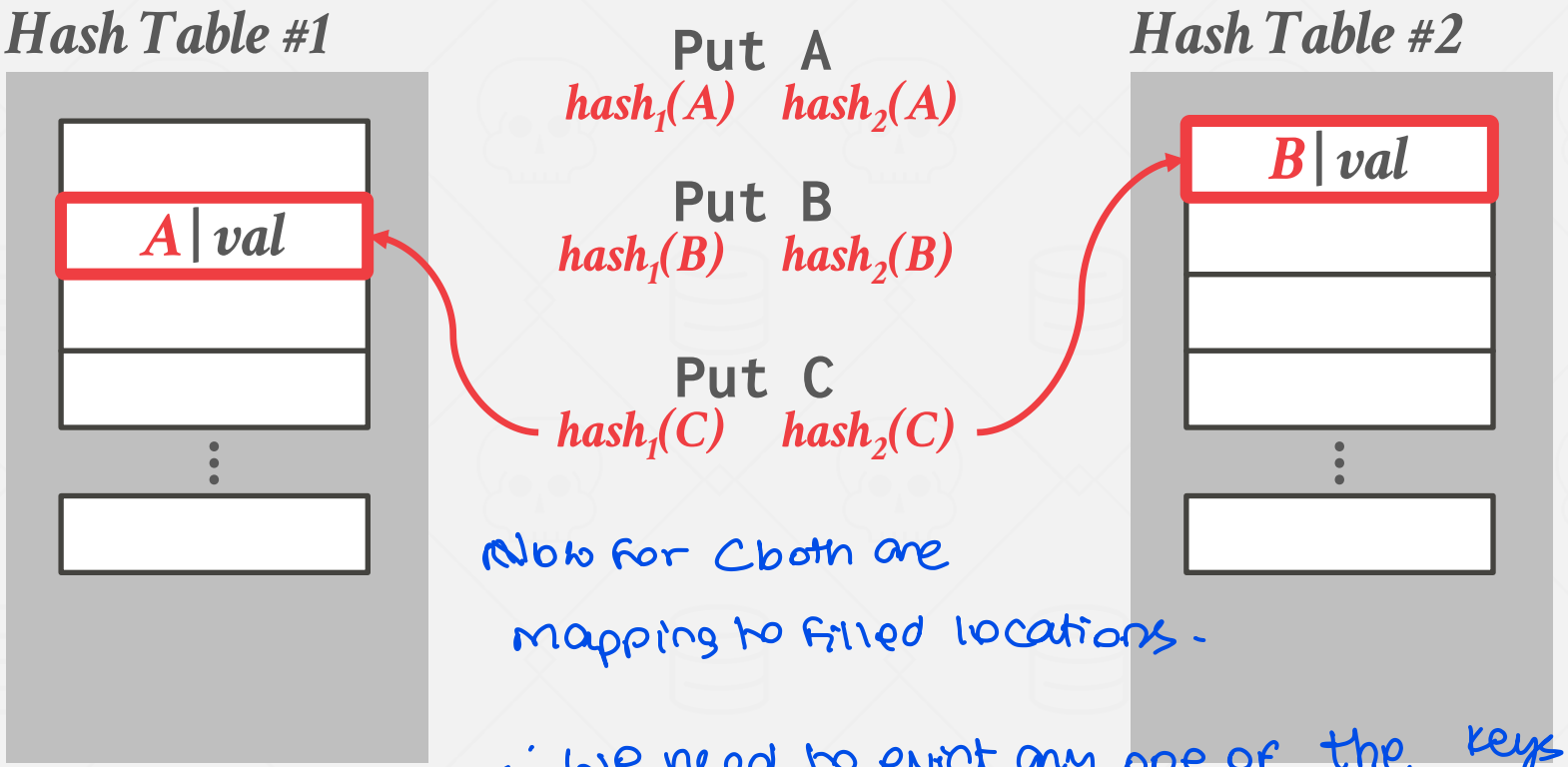
CUCKOO HASHING



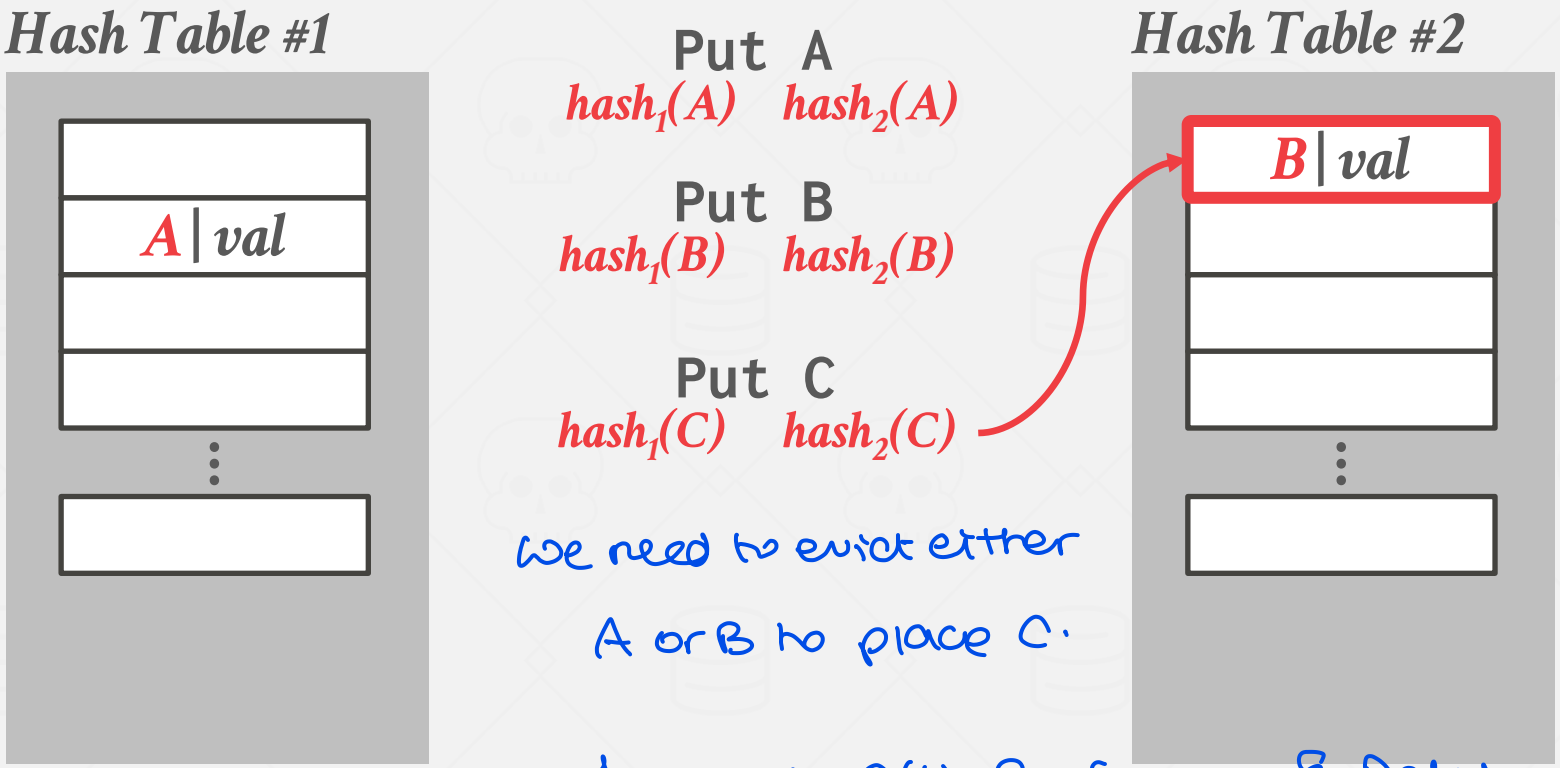
CUCKOO HASHING



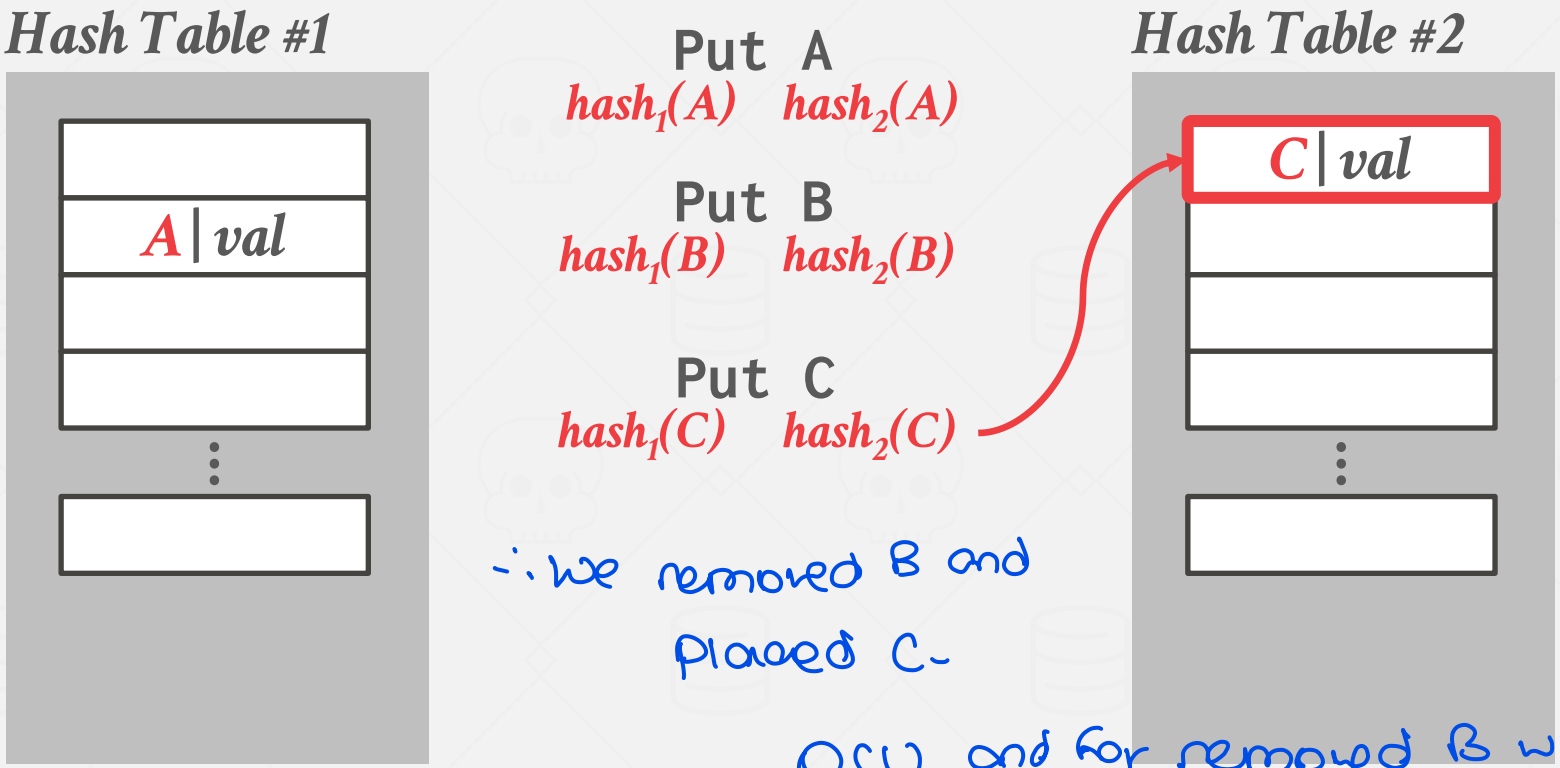
CUCKOO HASHING



CUCKOO HASHING



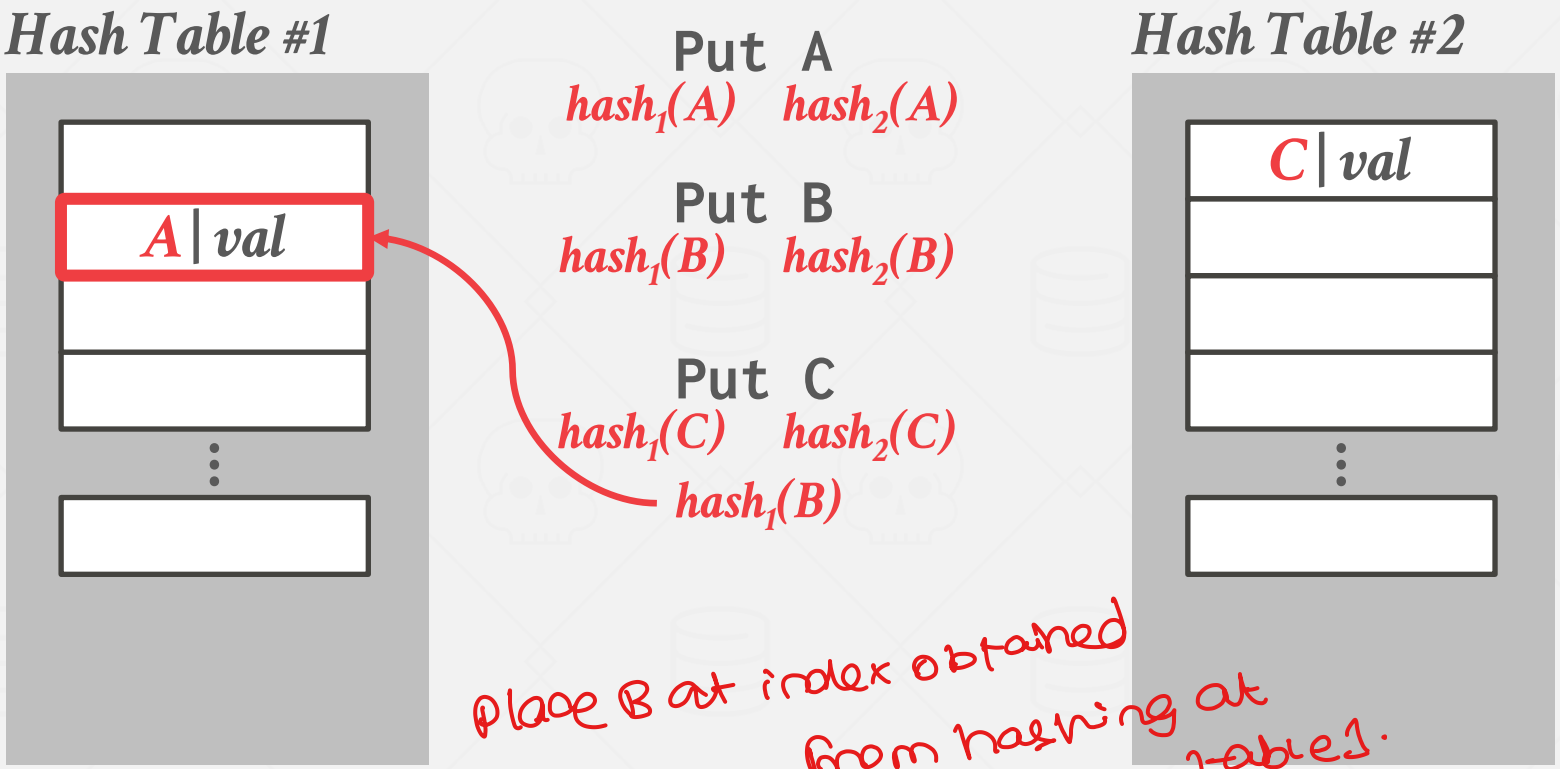
CUCKOO HASHING



∵ B is removed from table2

place it in new location

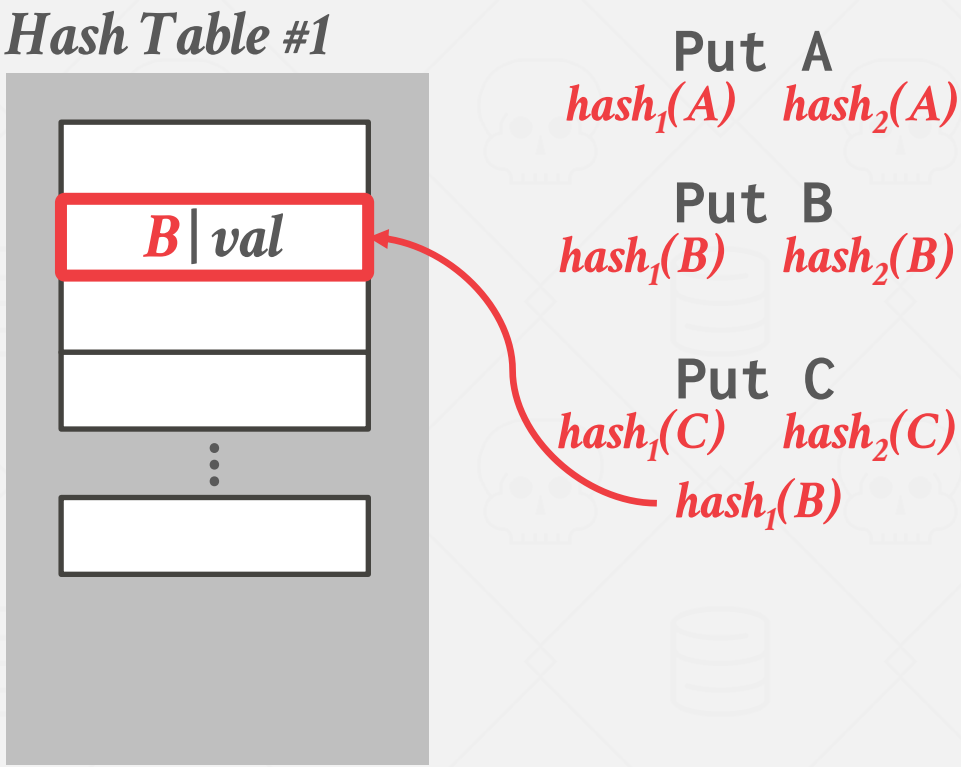
CUCKOO HASHING



Place B at index obtained from hashing at table1.

CUCKOO HASHING

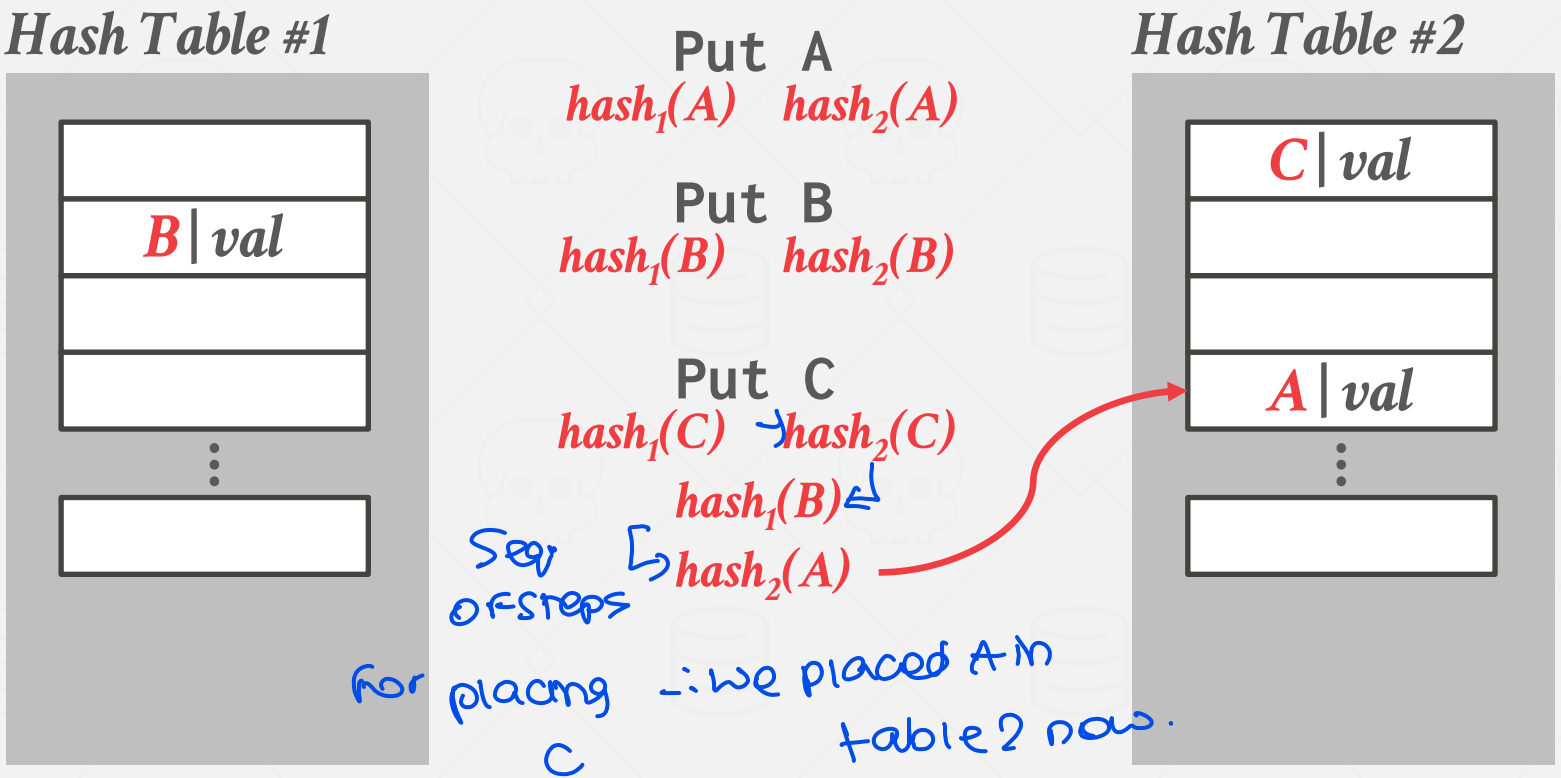
But B is now colliding with A.



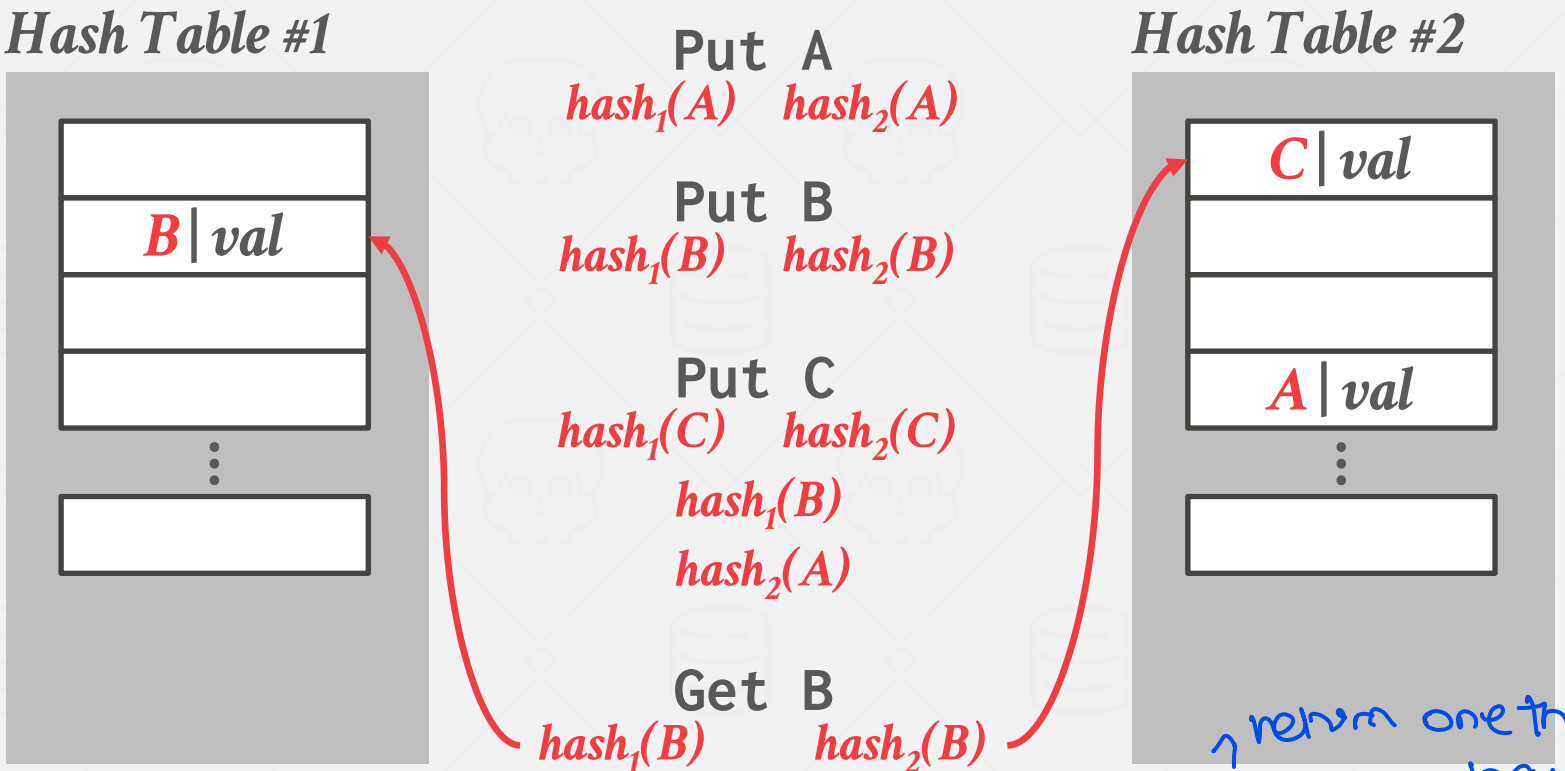
= Evict A place B rehash

use hash fn of table 2 to place A

CUCKOO HASHING



CUCKOO HASHING



OBSERVATION

All these req. no. of elements we have to store.

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.

we simply cannot maintain more than 1 hash table or a giant table.

Dynamic hash tables resize themselves on demand.

→ Chained Hashing
→ Extendible Hashing
→ Linear Hashing

Each slot in hash table is allored linked list of buckets.

CHAINED HASHING

For search we hash to bucket & scan it

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

Whenever collision happens we place all elements with same hash key into same bucket.

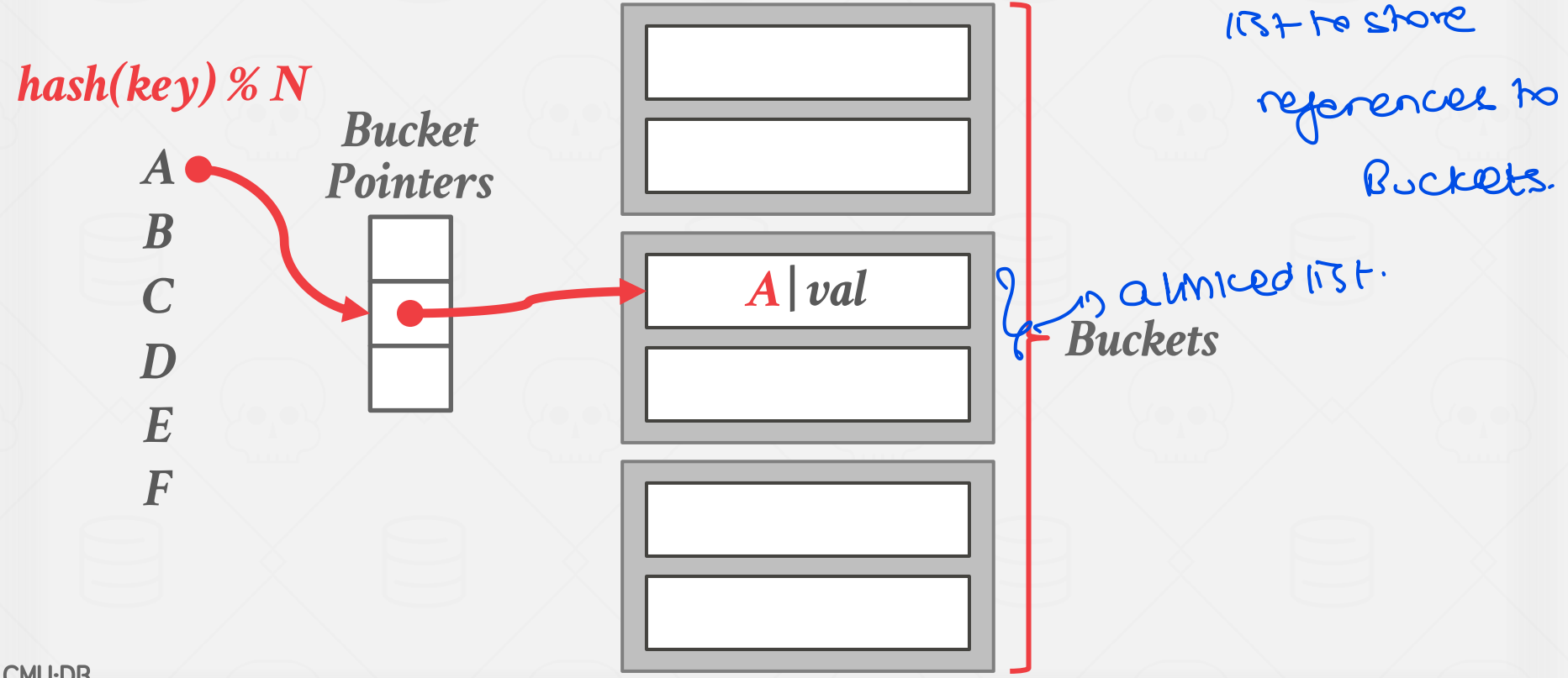
Resolve collisions by placing all elements with the same hash key into the same bucket.

→ To determine whether an element is present, hash to its bucket and scan for it.

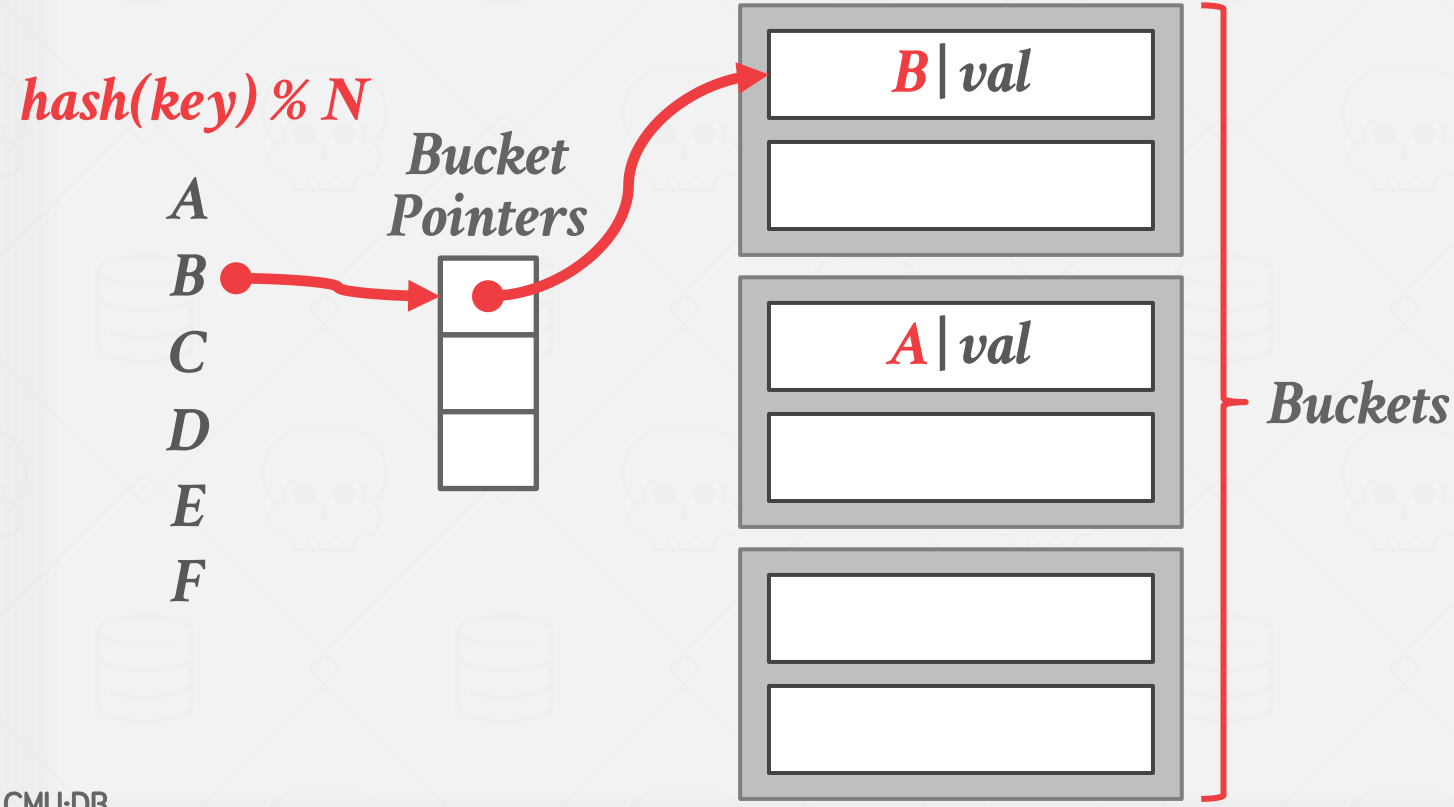
→ Insertions and deletions are generalizations of lookups.

CHAINED HASHING

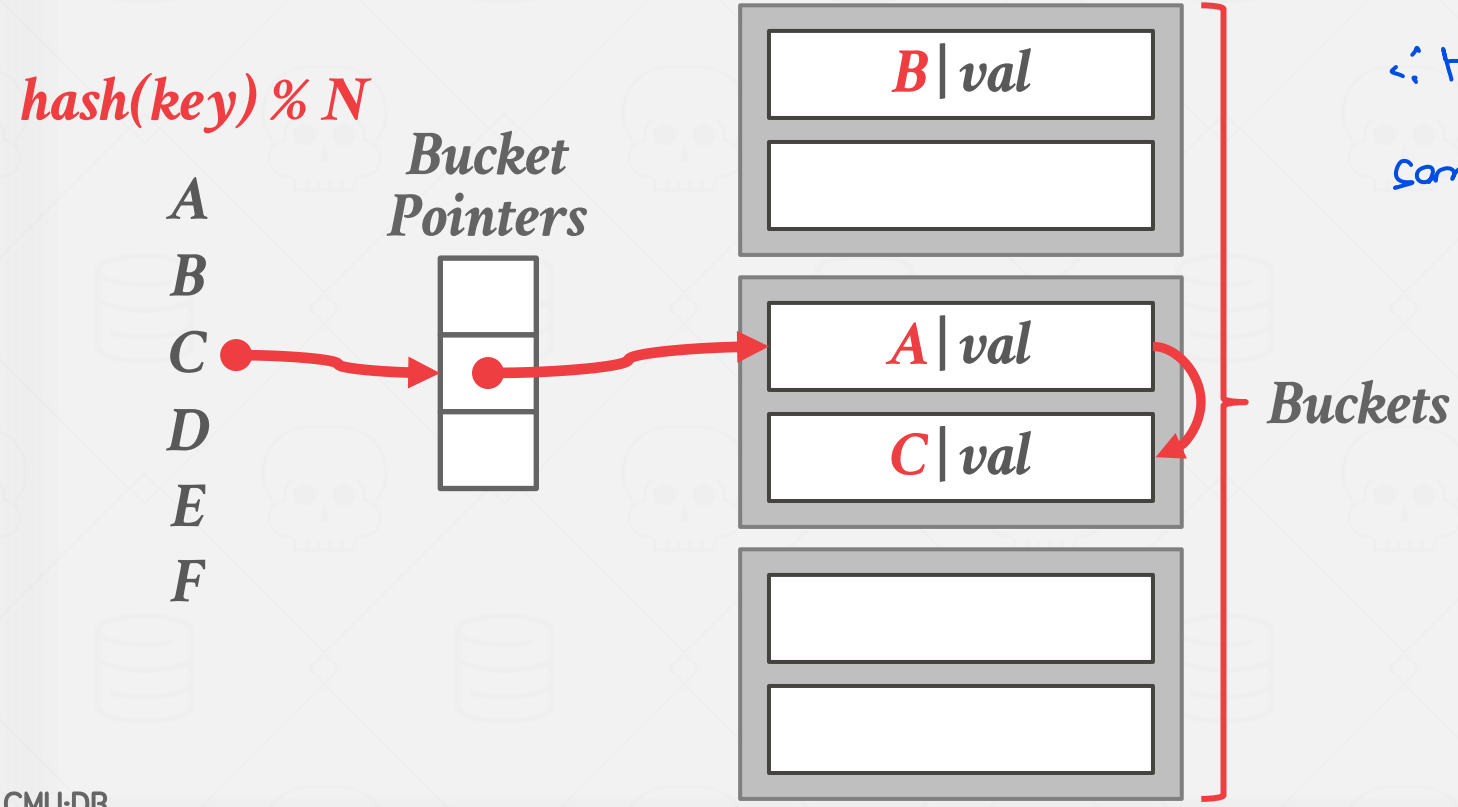
To maintain buckets we have list to store references to Buckets.



CHAINED HASHING



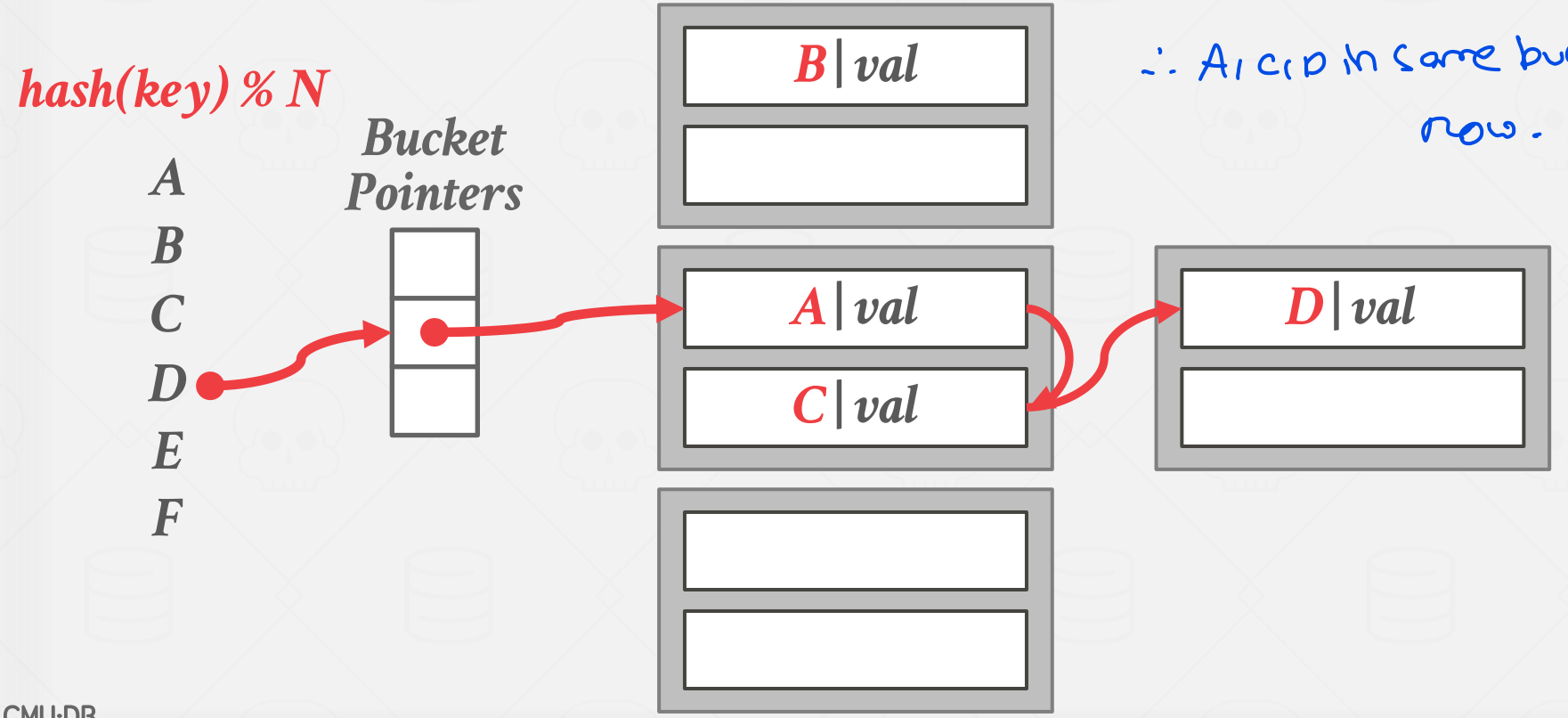
CHAINED HASHING



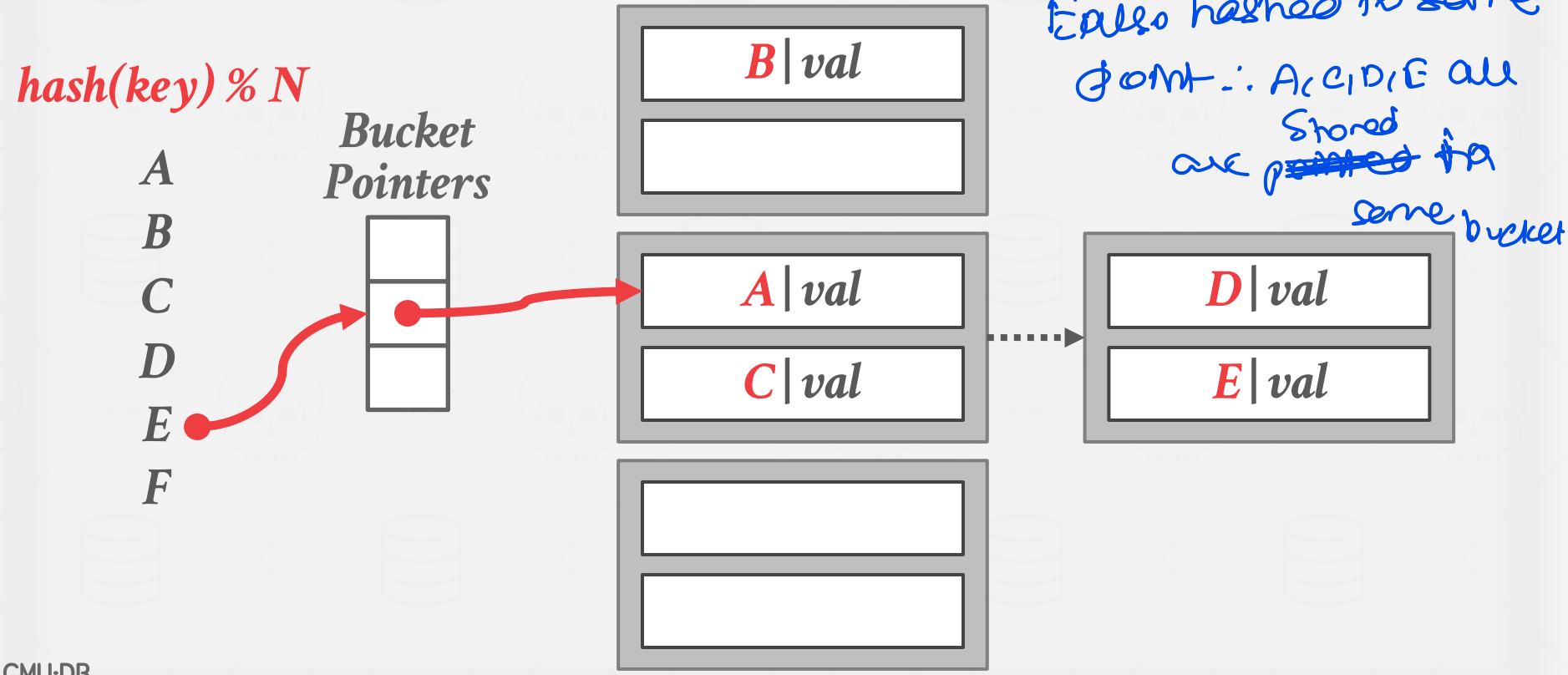
∴ Here A & C same key ∴ They both set into same bucket.

CHAINED HASHING

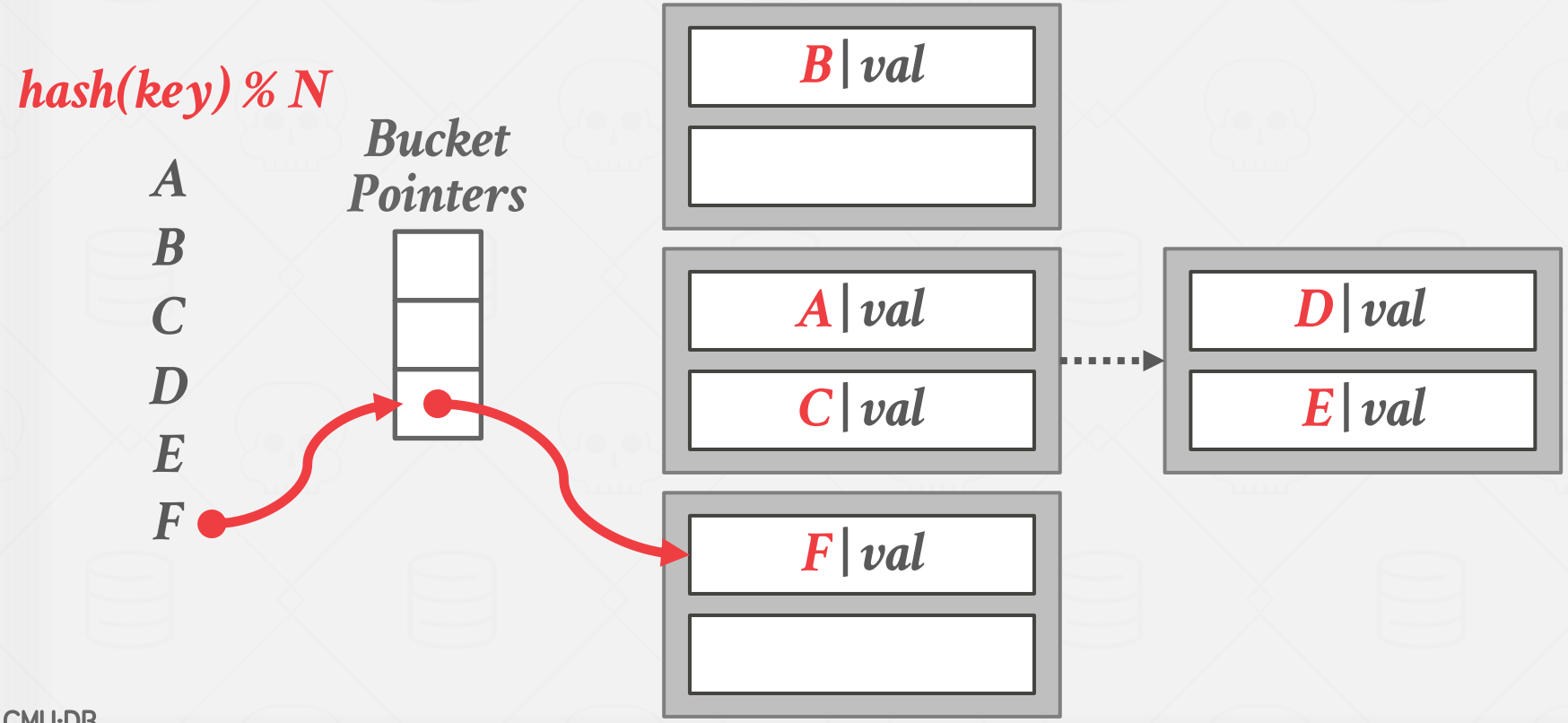
Now D also.
∴ add another list
∴ A, C, D in same bucket now.



CHAINED HASHING



CHAINED HASHING



Variant of Chained Hashing where we split buckets instead of letting linked lists grow forever.

EXTENDIBLE HASHING

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

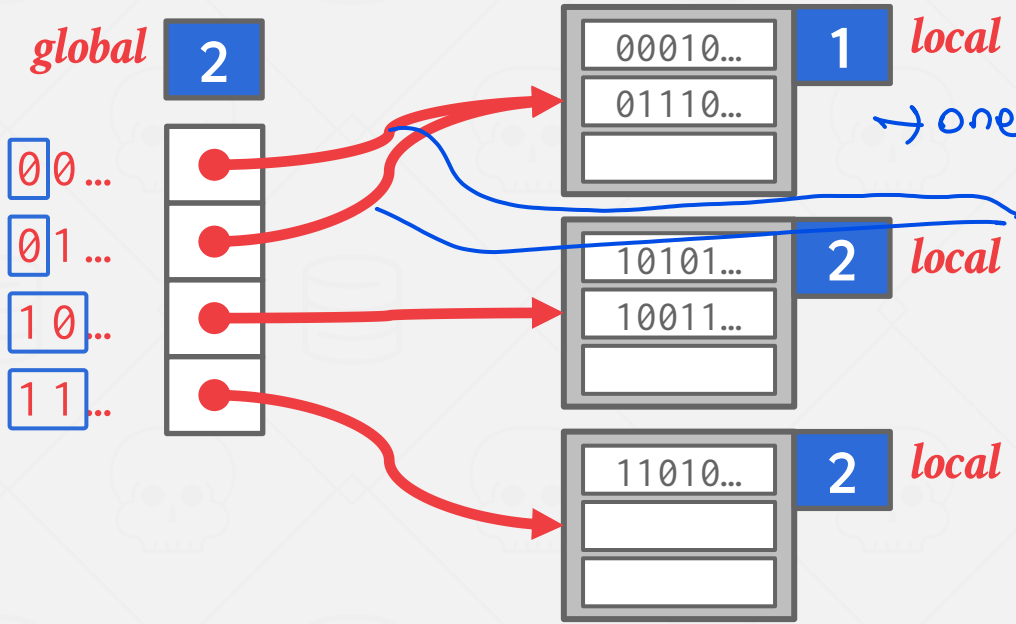
Multiple slot locations can point to the same bucket chain.

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

→ Data movement is localized to just the split chain.

During split reshuffle the bucket entries & increase the no. of bits to examine

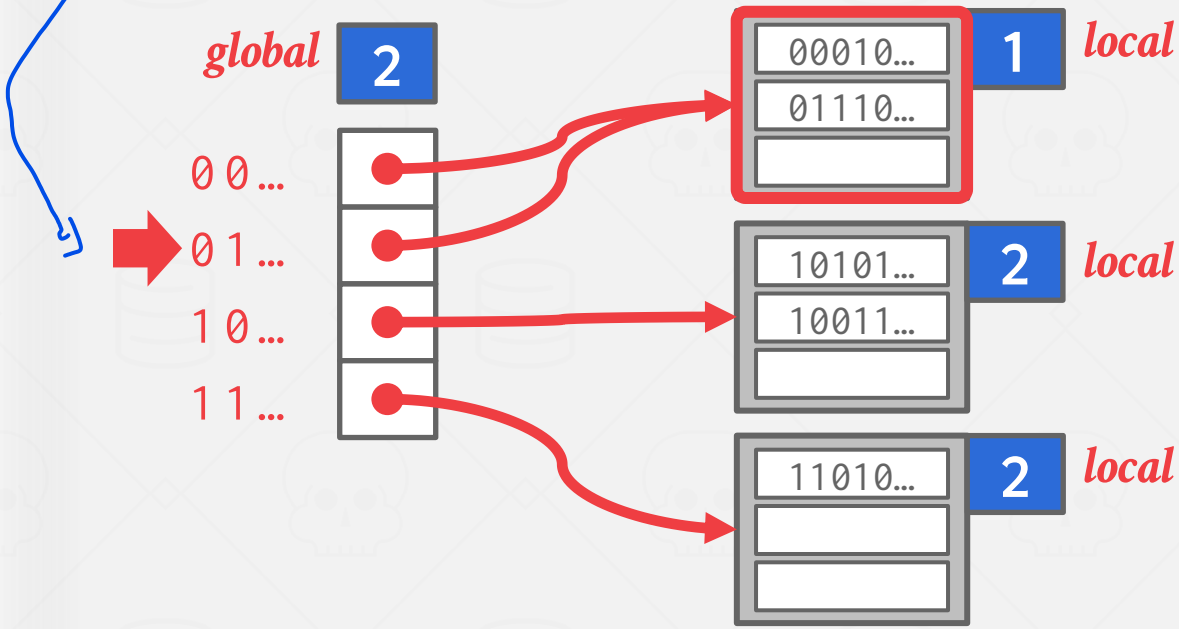
EXTENDIBLE HASHING



→ one bucket chain.

more than 1 pointer to single bucket chain.

EXTENDIBLE HASHING



Get A
hash(A) = 01110...

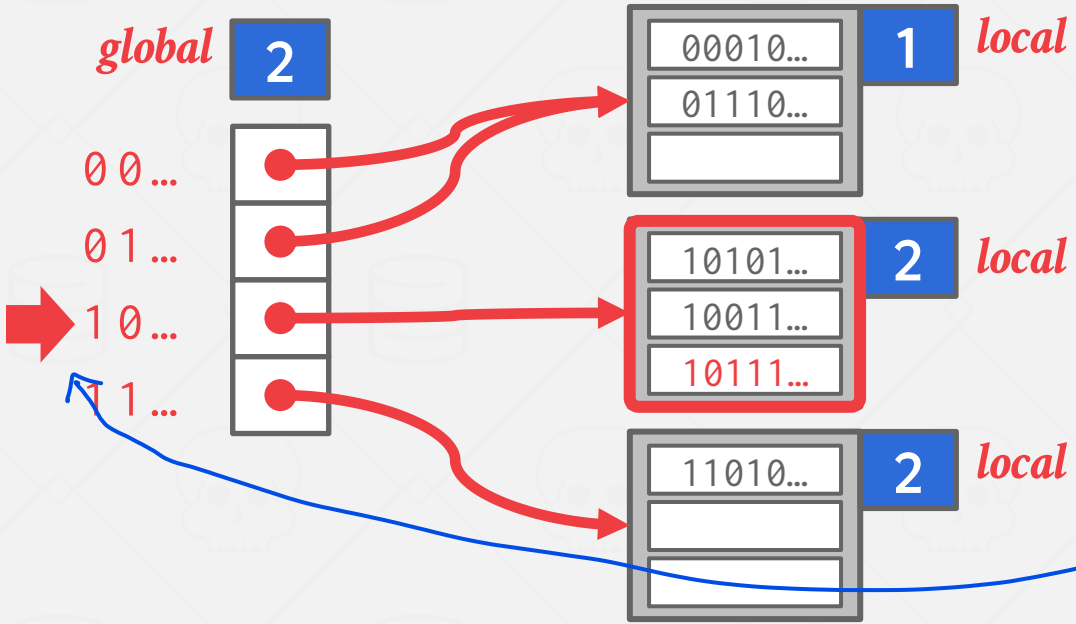
way to check for offset

Upon lookup.

∴ Go to bucket chain

& scan for it

EXTENDIBLE HASHING



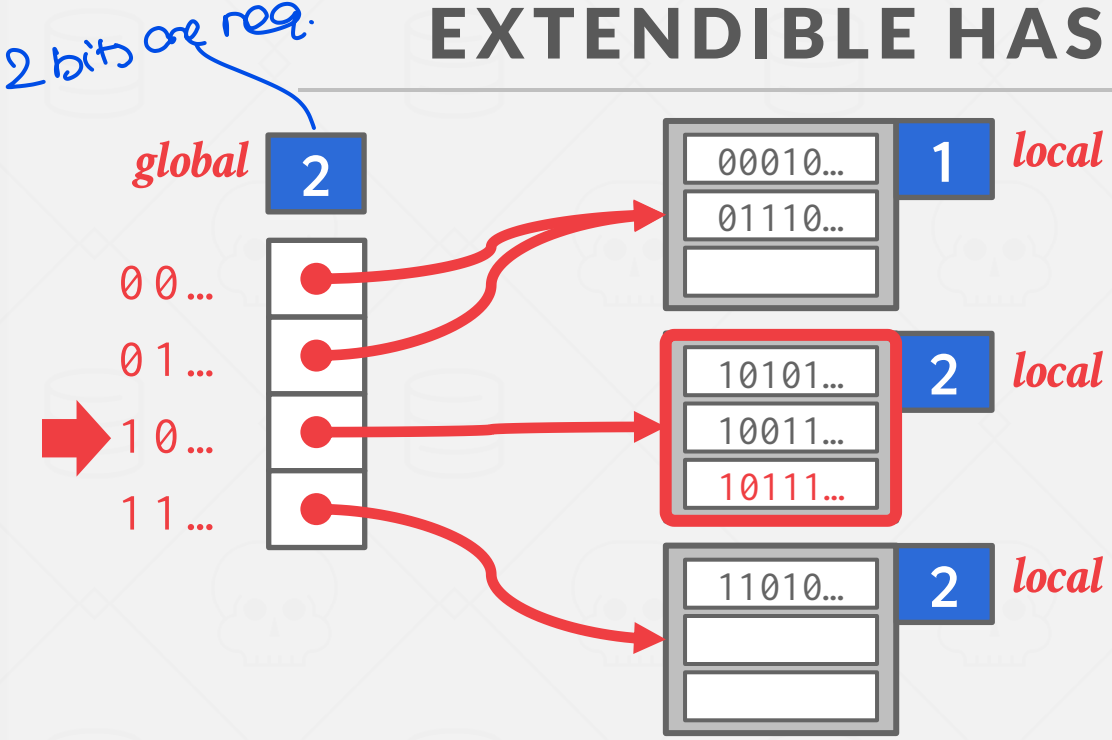
Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

∴ Go to bucket chain 2

Place B there.

EXTENDIBLE HASHING



Get A
hash(A) = 01110...

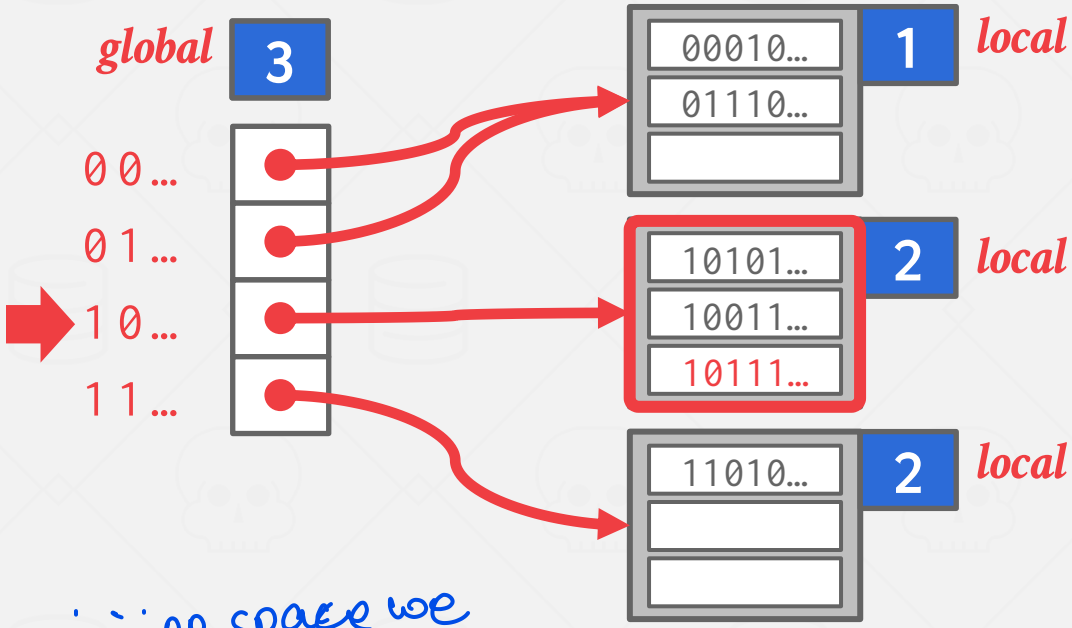
Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

go to bucket chain 2

check for empty space.

EXTENDIBLE HASHING



Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

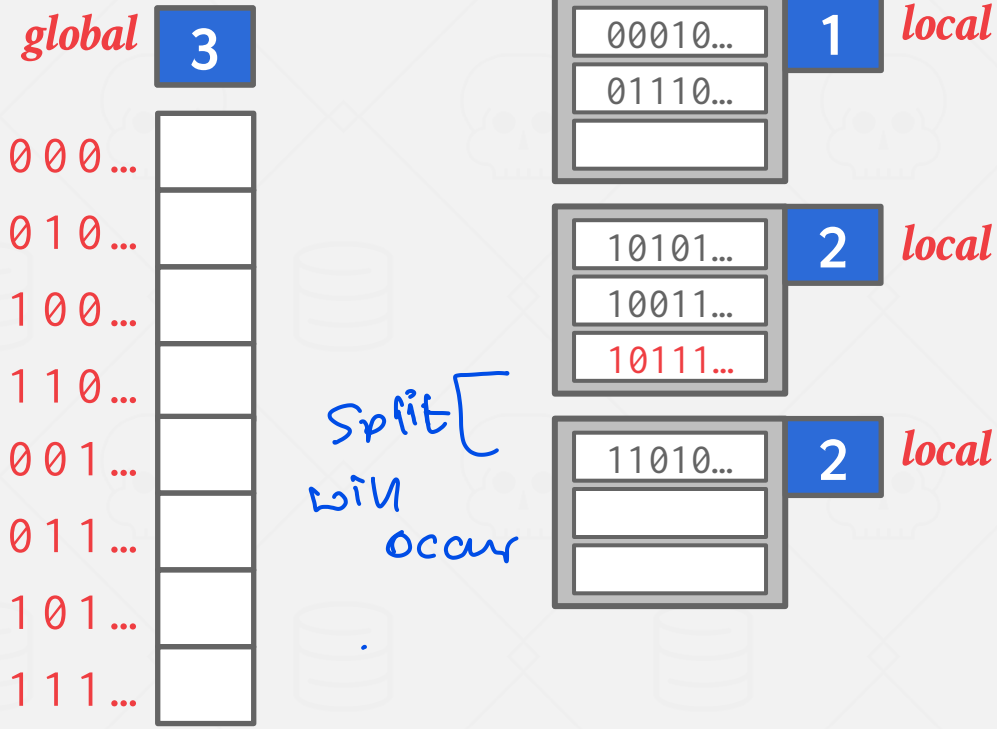
Put C
hash(C) = 10100...

∴ ∵ no space we need to split & increase the no. of bits for lookup.

C does not have space to be put in bucket chain 2

2→3 § split occurs § entries are reshuffled -

EXTENDIBLE HASHING

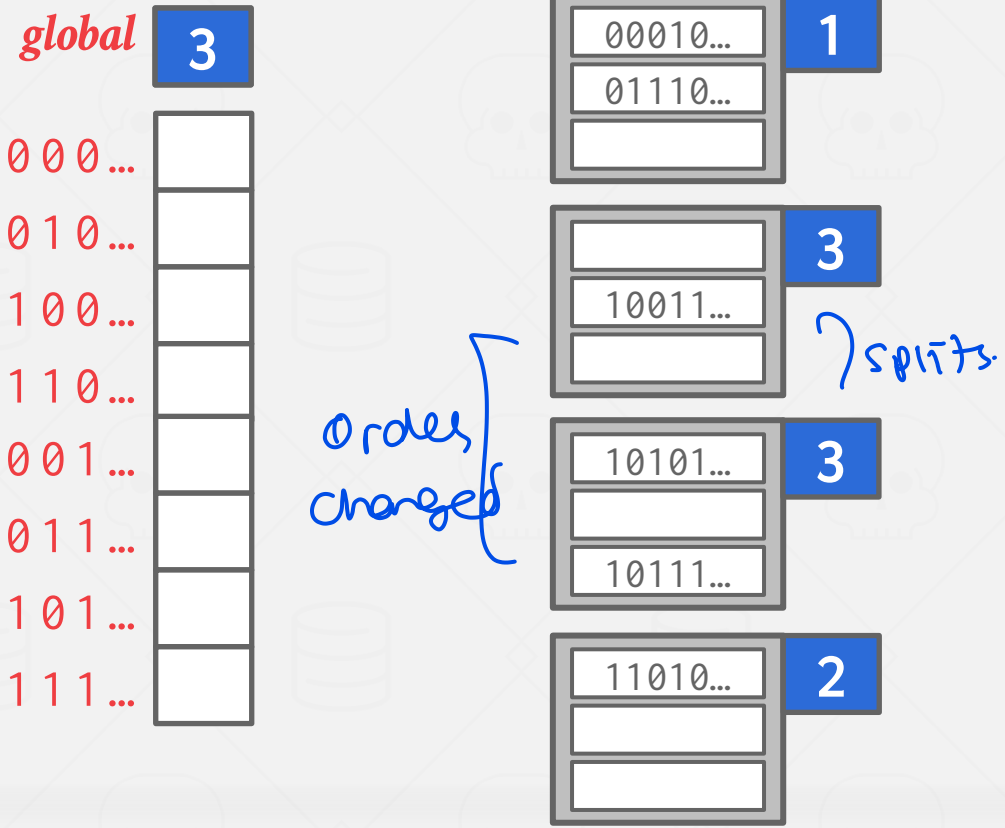


Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

EXTENDIBLE HASHING



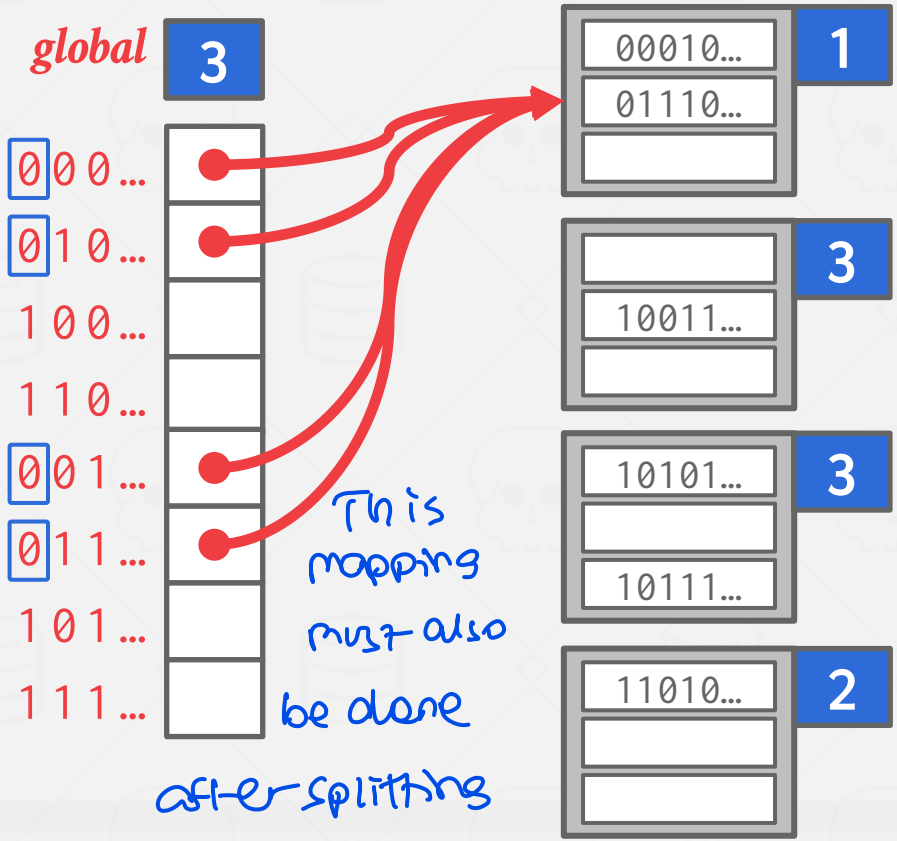
Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

now we put C

EXTENDIBLE HASHING

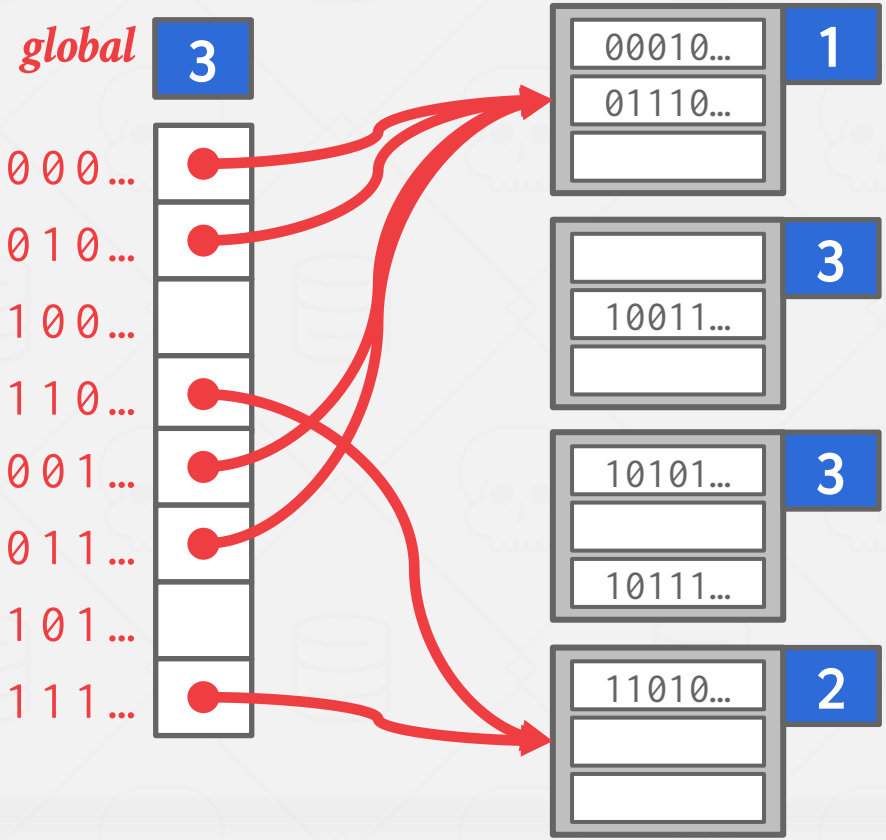


Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

EXTENDIBLE HASHING

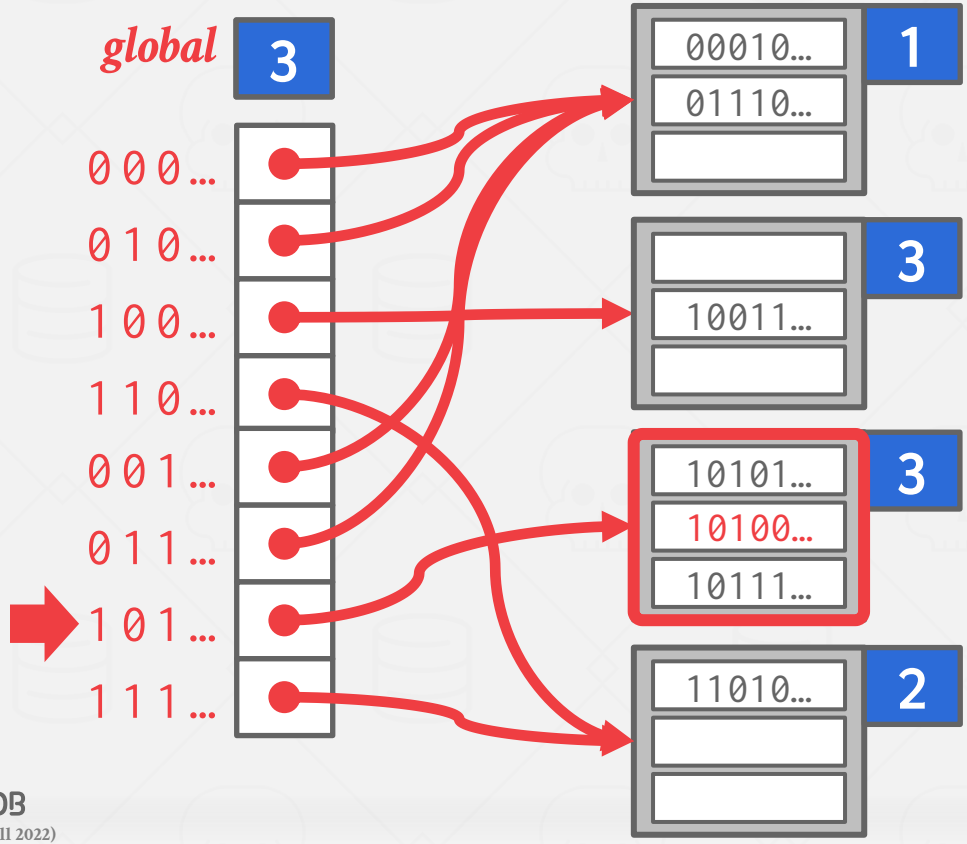


Get A
hash(A) = 01110...

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...

Explanation of how this Mapping occurs ??

We use a hash table to maintain reference to the next bucket we want to split.

LINEAR HASHING

When any bucket overflows we split the bucket at that

The hash table maintains a pointer that tracks the next bucket to split. location.

→ When any bucket overflows, split the bucket at the pointer location.

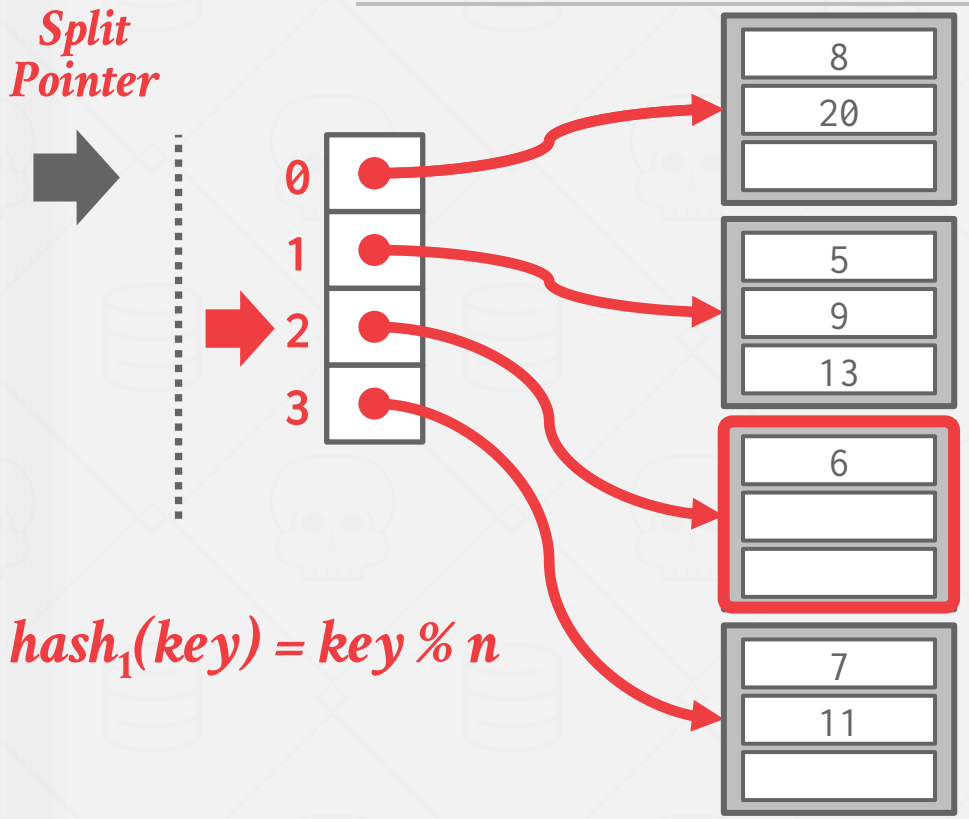
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

→ Space Utilization

→ Average Length of Overflow Chains

LINEAR HASHING

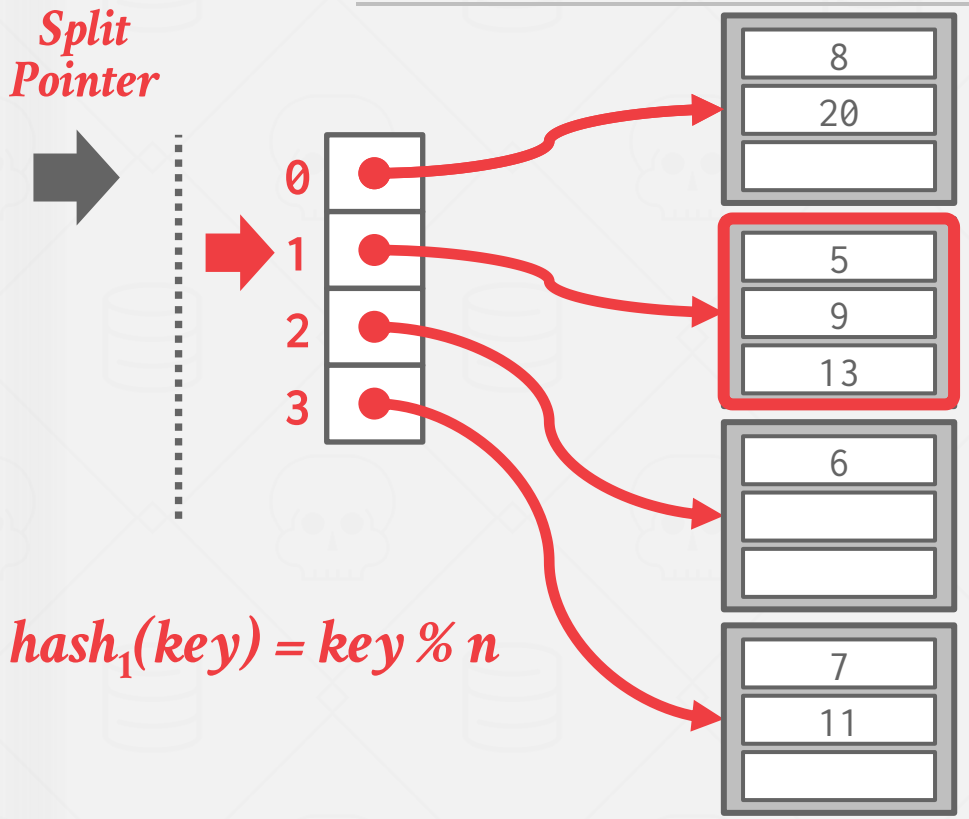


Get 6

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

$hash_1(key) = key \% n$

LINEAR HASHING



$hash_1(key) = key \% n$

Get 6

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

Put 17

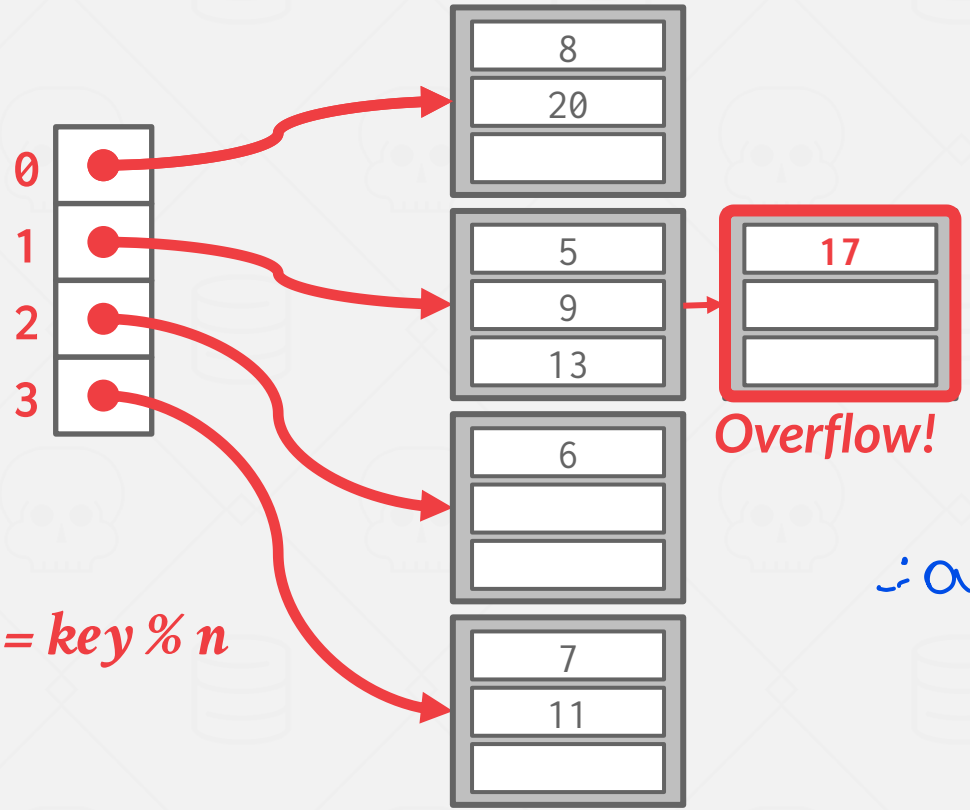
$hash_1(17) = 17 \% 4 = 1$

But in bucket 1 not there but

bucket overflow split

bucket 1 _

LINEAR HASHING



Get 6

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

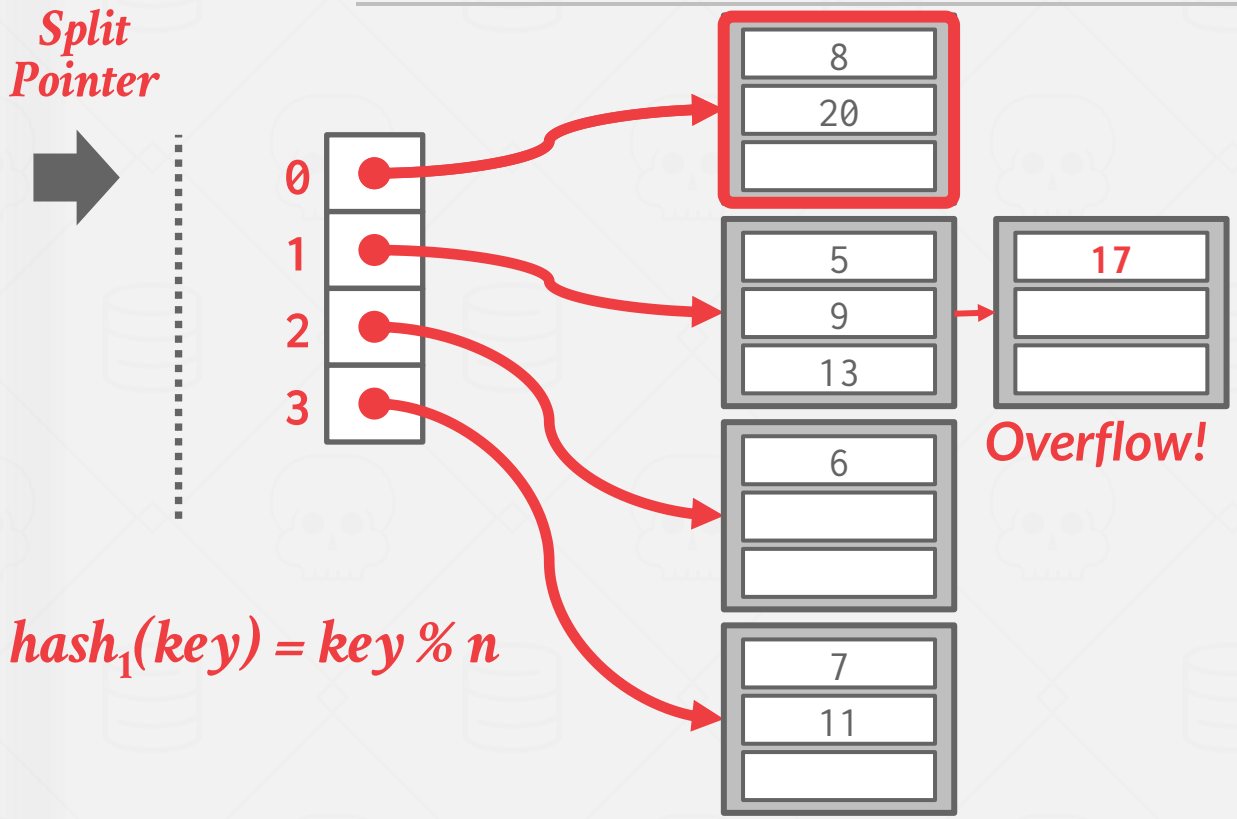
Put 17

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

$hash_1(key) = key \% n$

∴ overflow 17 placed in Split.

LINEAR HASHING



Get 6

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

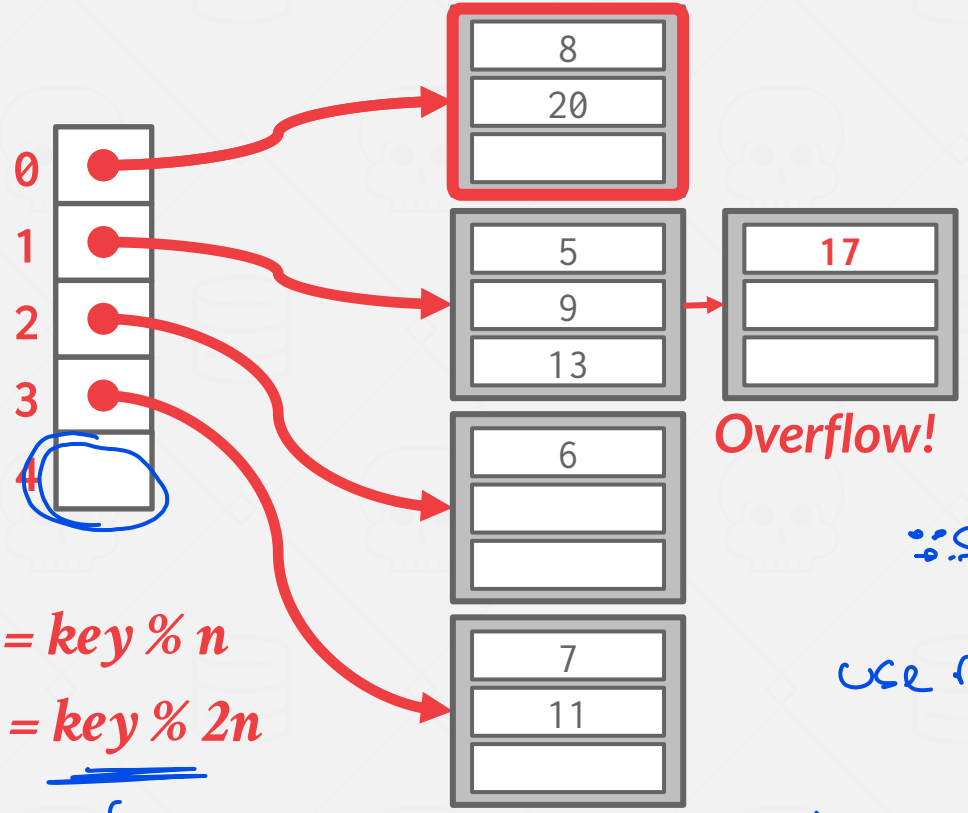
Put 17

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

LINEAR HASHING

∧ ↑ by 1

Split Pointer



Get 6

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

Put 17

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

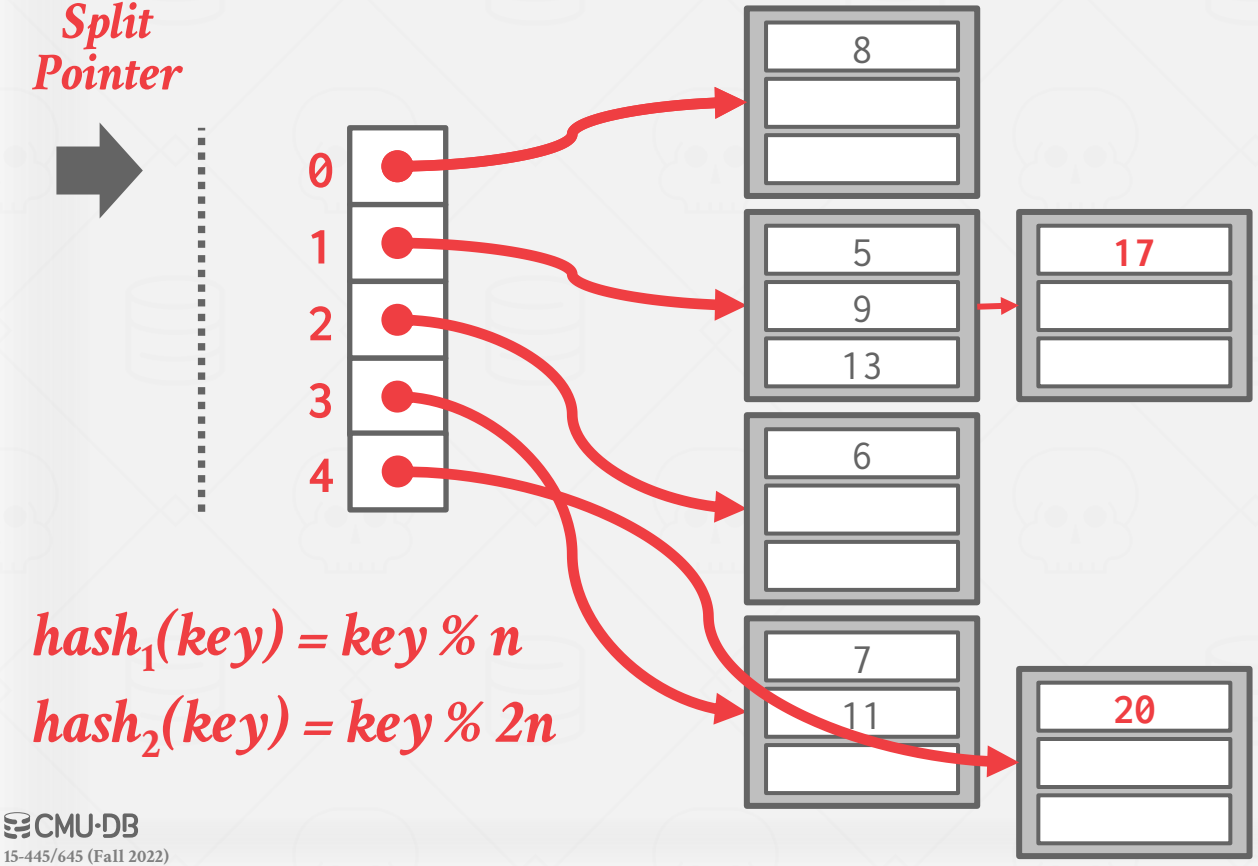
$hash_1(key) = key \% n$

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

∴ Split happened now we use multiple hash functions.

↳ How did we design this hash2 ?? for input key.

LINEAR HASHING



$hash_1(key) = key \% n$

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

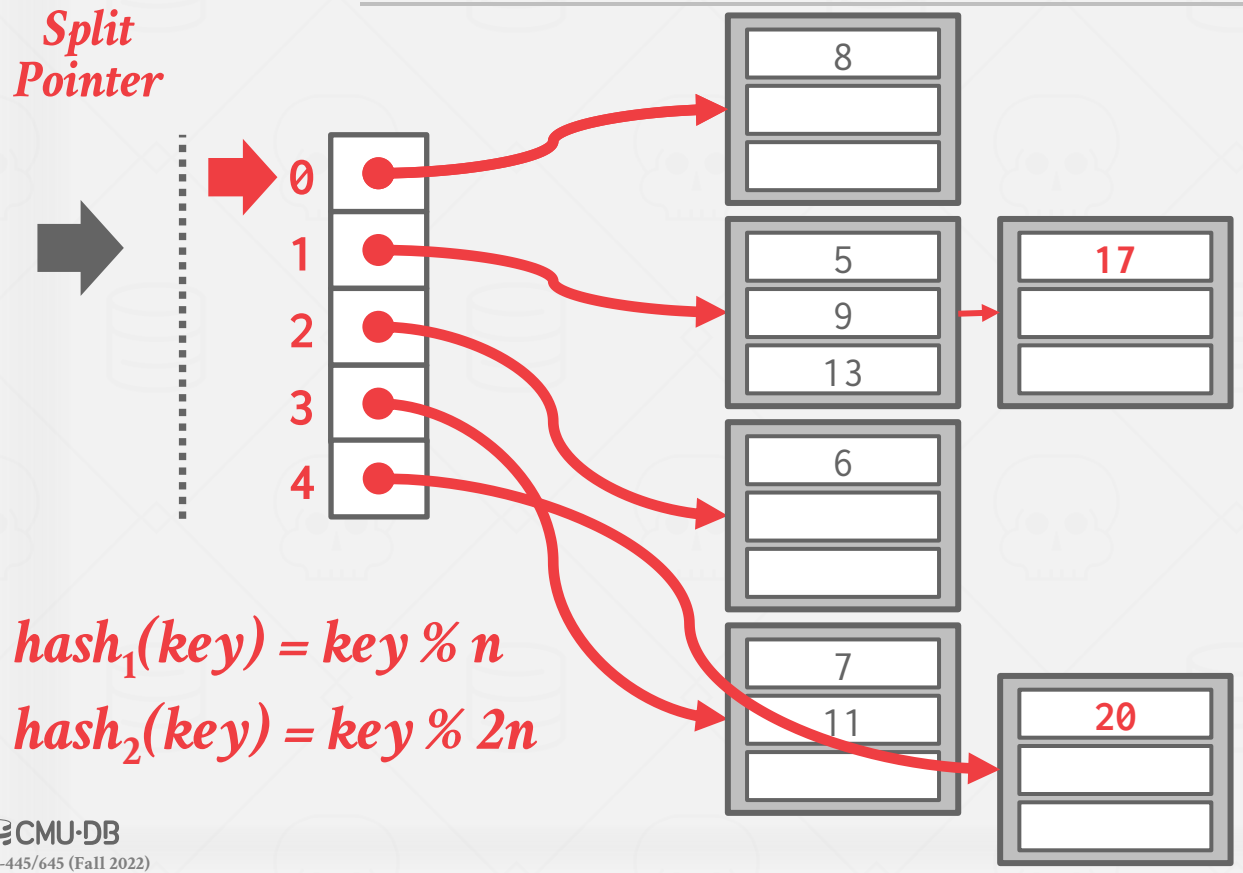
Get 6

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

Put 17

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

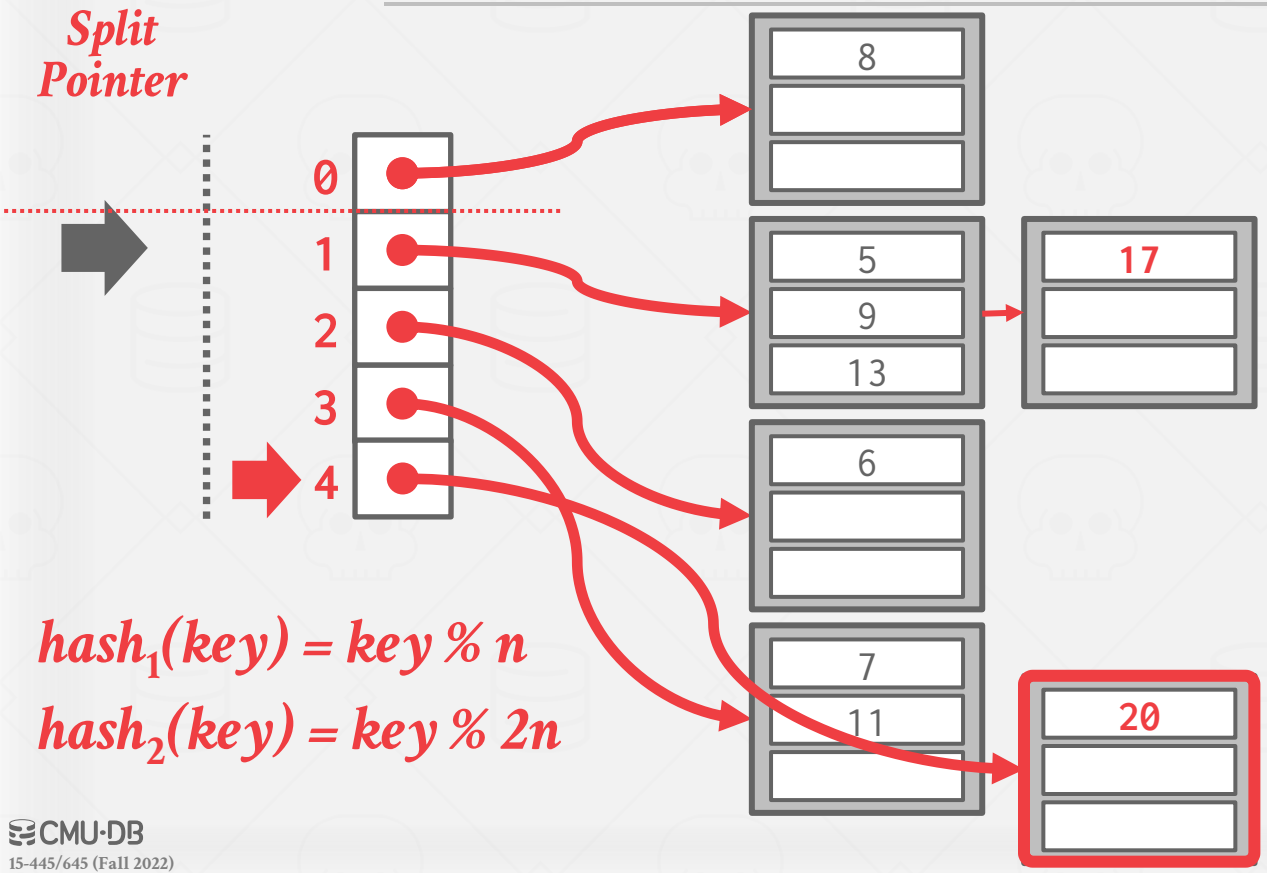
Put 17

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

Get 20

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

Put 17

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

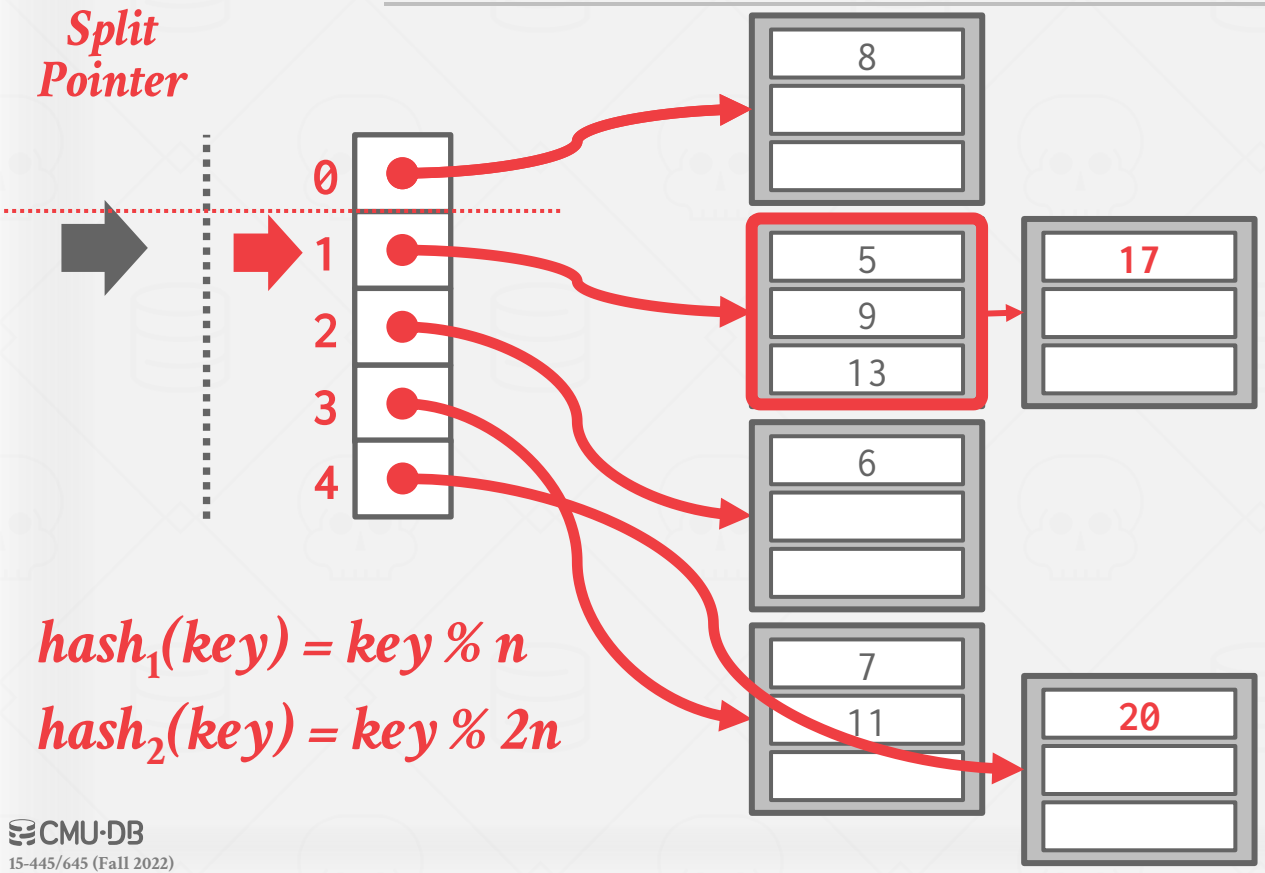
Get 20

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

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

2 different values.

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

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

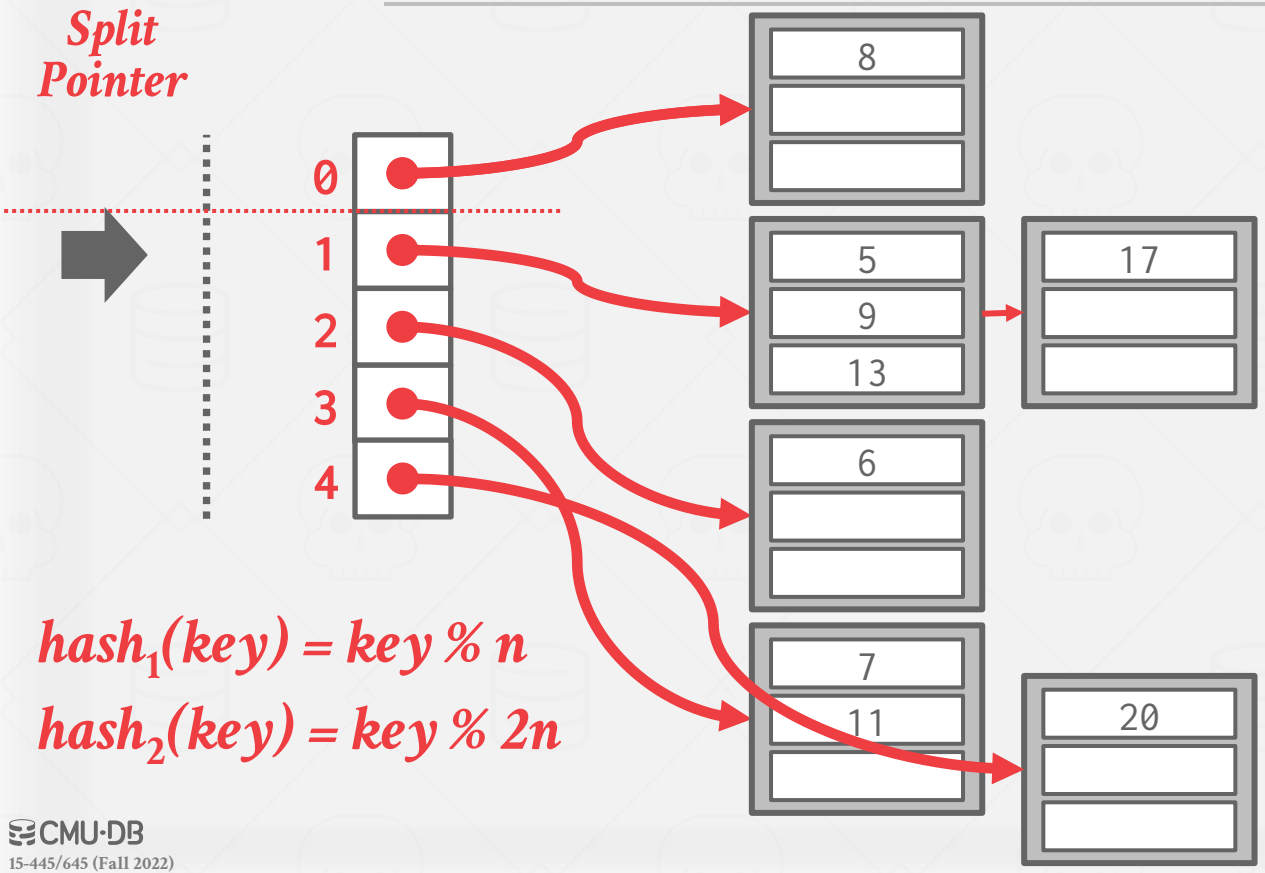
LINEAR HASHING

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

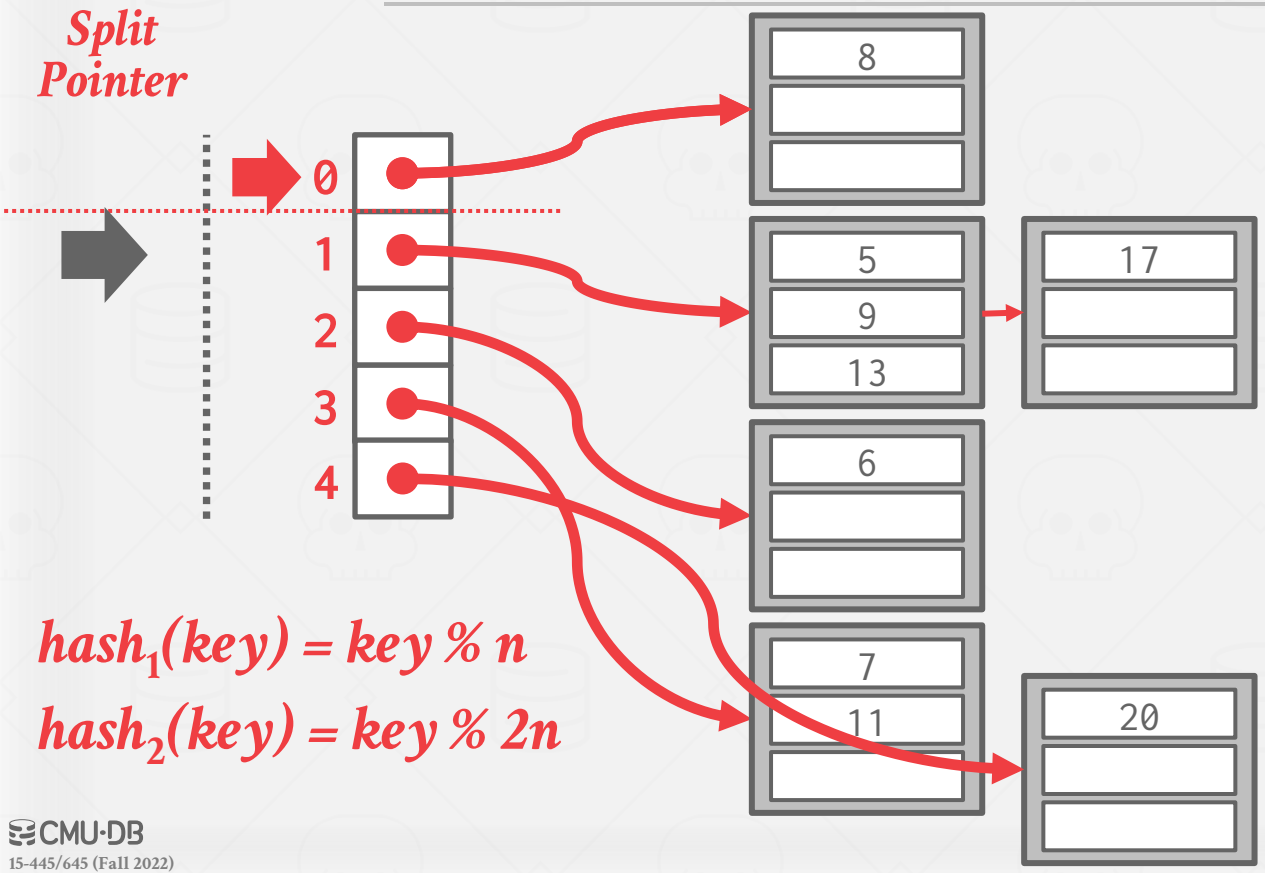
→ When the pointer reaches the last slot, delete the first hash function and move back to beginning.

Did not understand Linear Hashing =

LINEAR HASHING – DELETES



LINEAR HASHING – DELETES



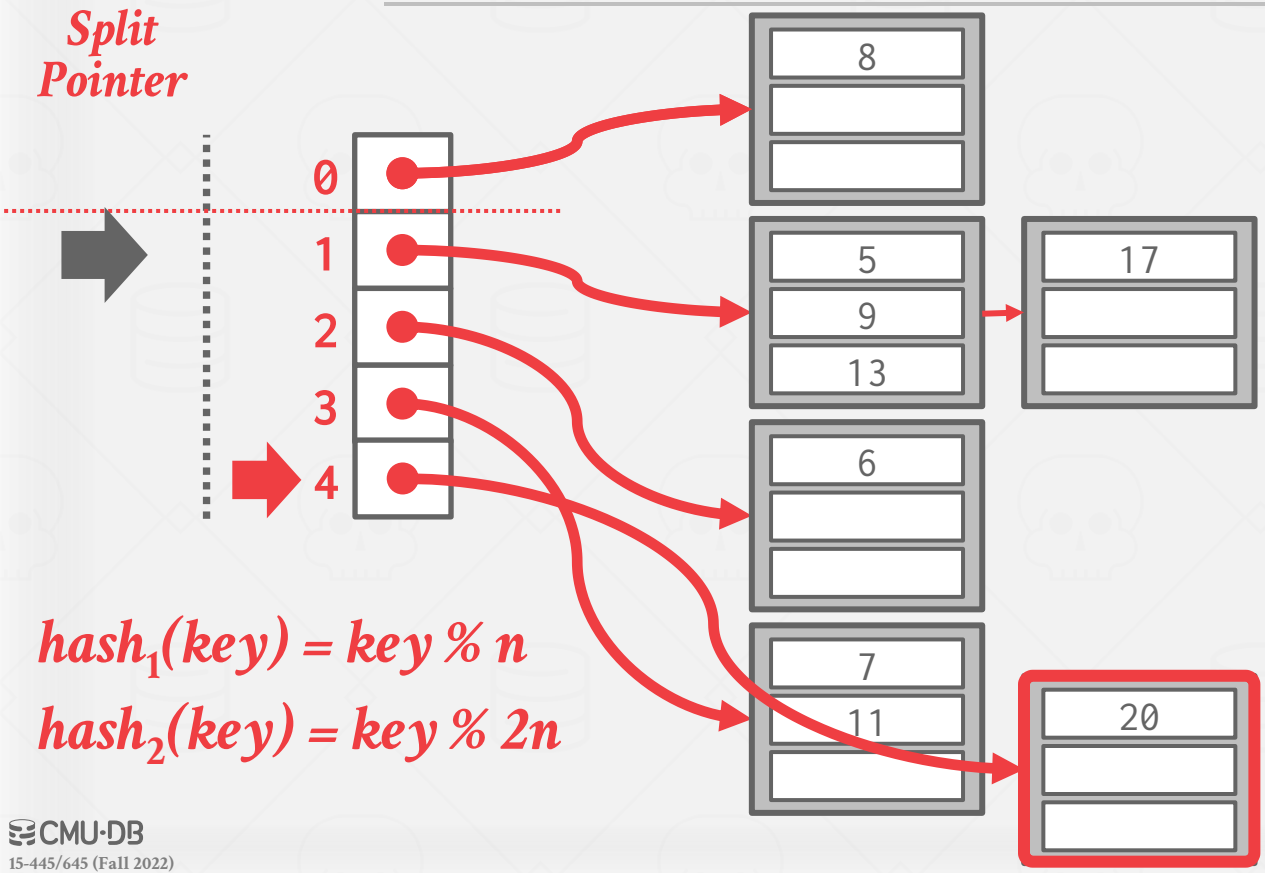
Delete 20

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

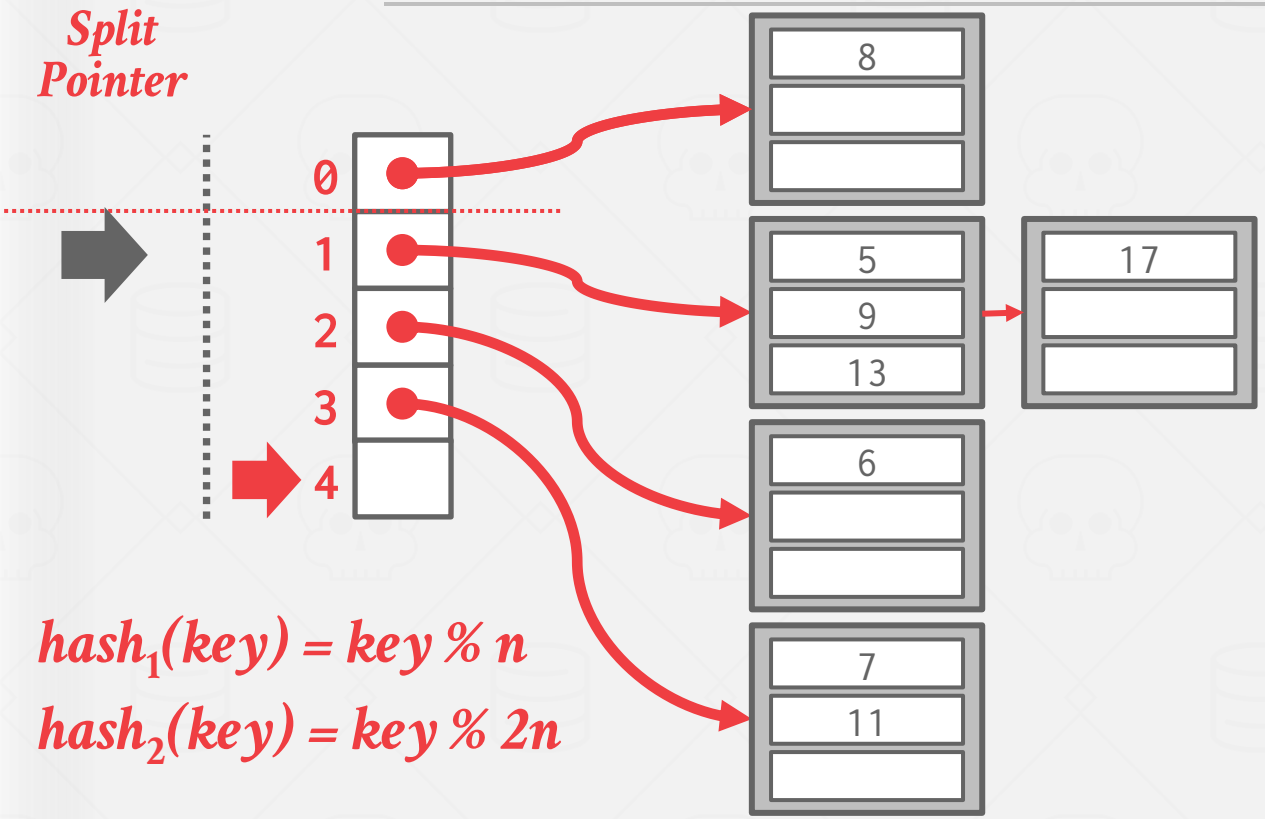
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

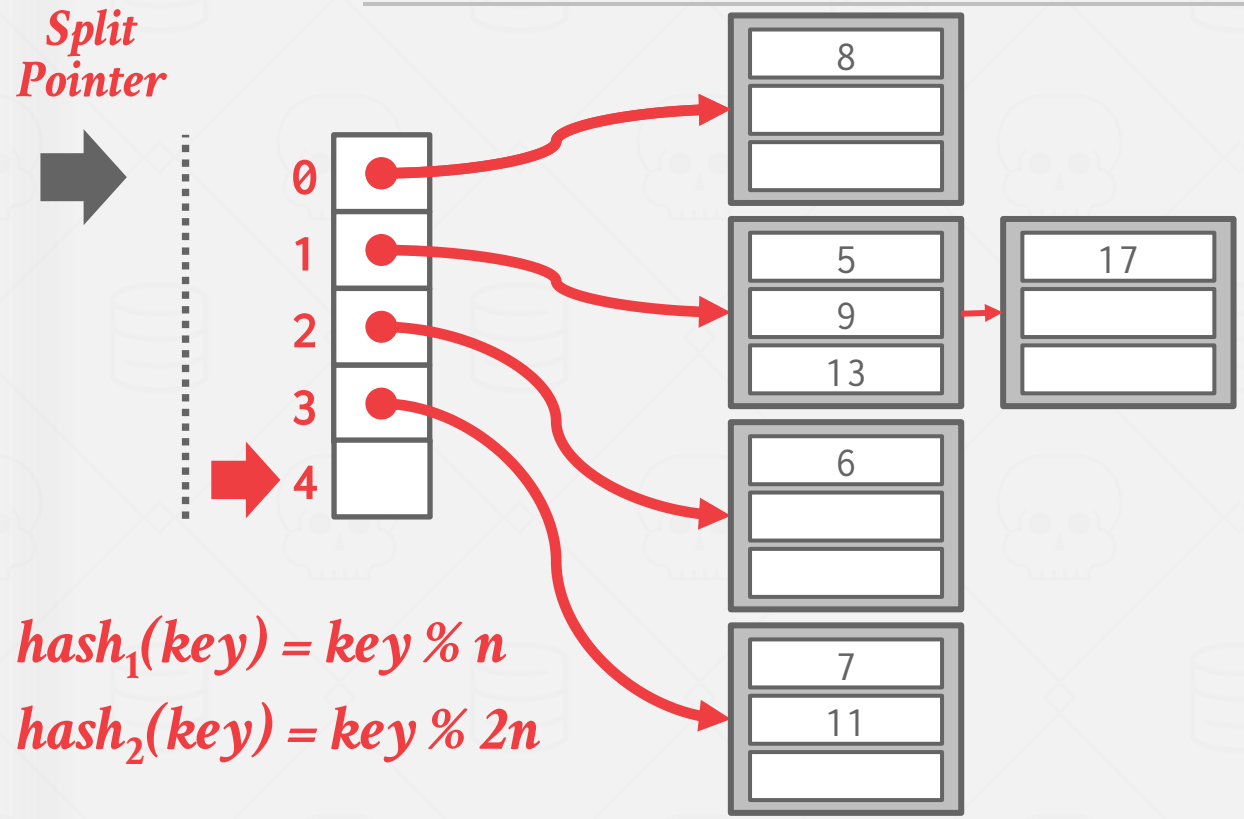
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



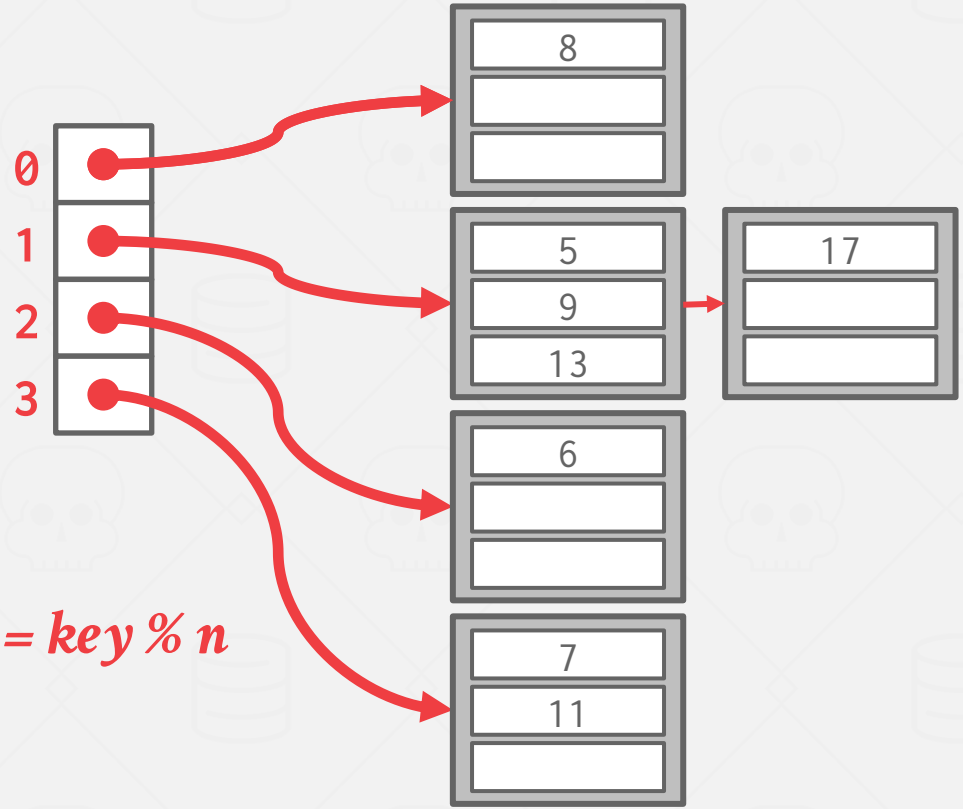
Delete 20

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

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

LINEAR HASHING – DELETES

Split Pointer



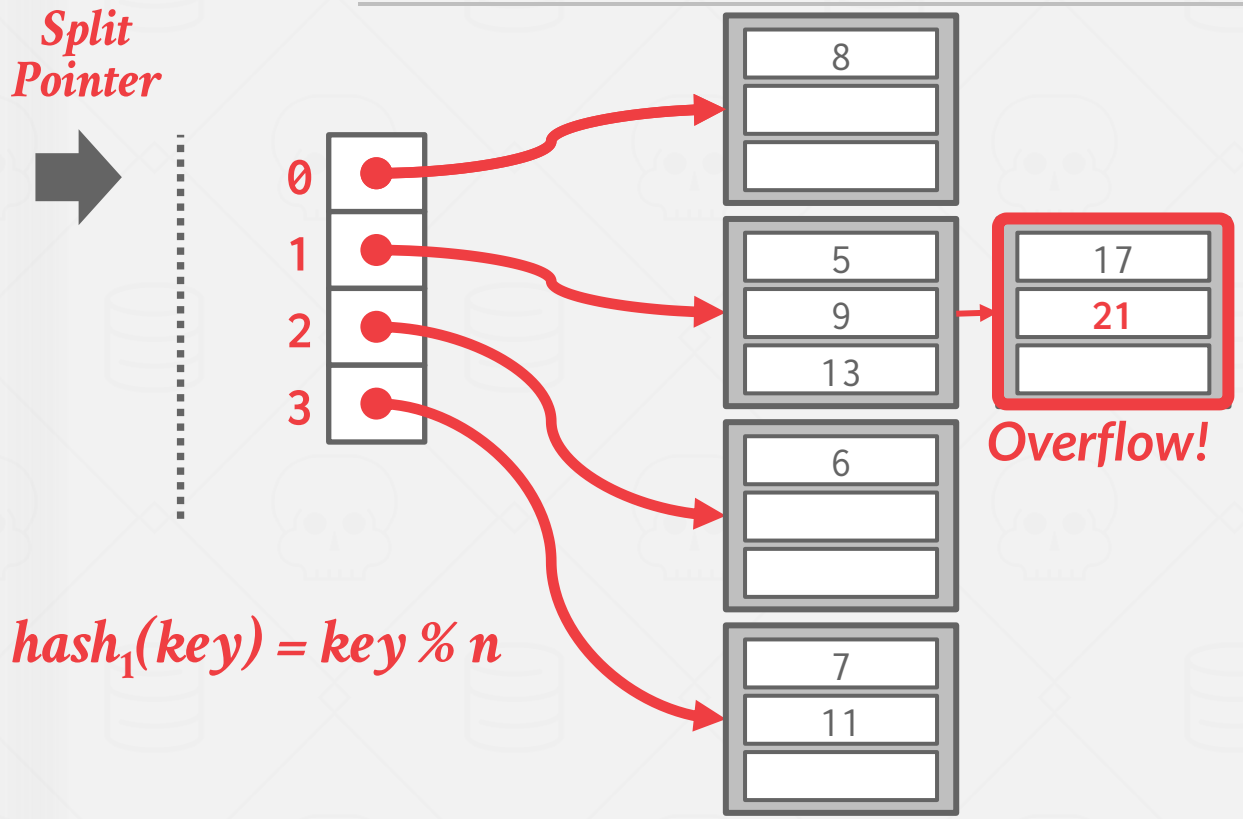
$hash_1(key) = key \% n$

Delete 20

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

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

LINEAR HASHING – DELETES



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"