Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #07

Hash Tables

FALL 2023 ❯❯ Prof. Andy Pavlo • Prof. Jignesh Patel

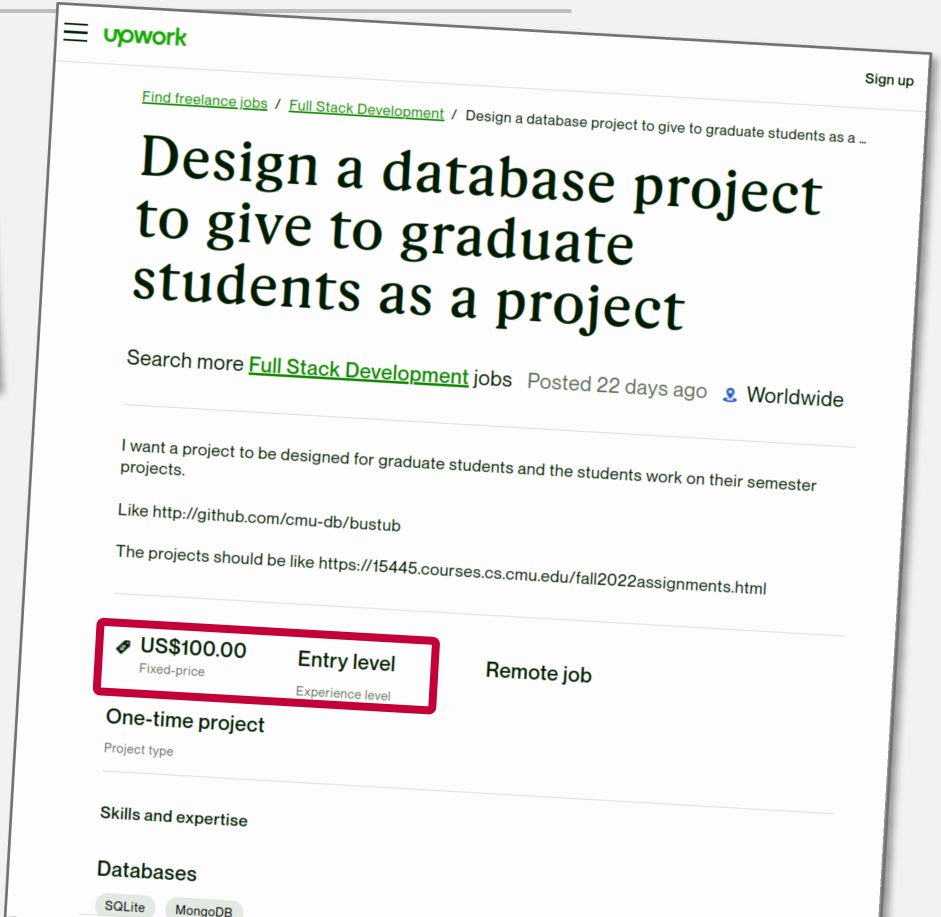
ADMINISTRIVIA

Project #1 is due Sun Oct 2nd @ 11:59pm
→ Special Office Hours: **Sat Oct 1st @ 3pm-5pm**

Homework #2 is due Wed Oct 4th @ 11:59pm

DATABASES = CASH MONEY

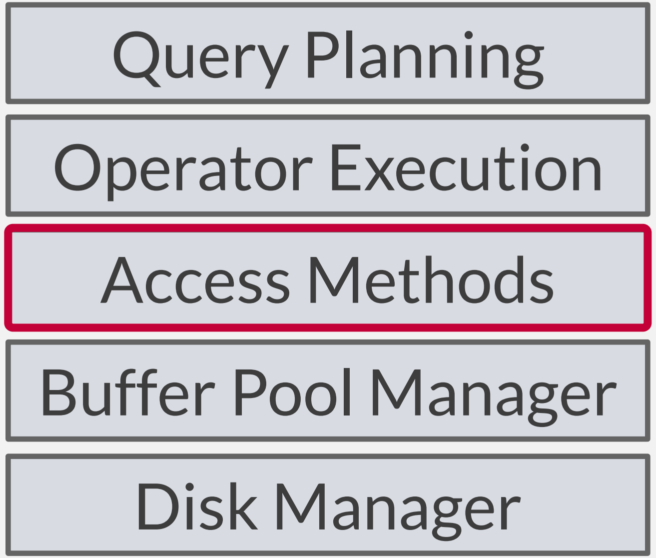
DATABASES = CASH MONEY



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 (Unordered)
→ Trees (Ordered)



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.

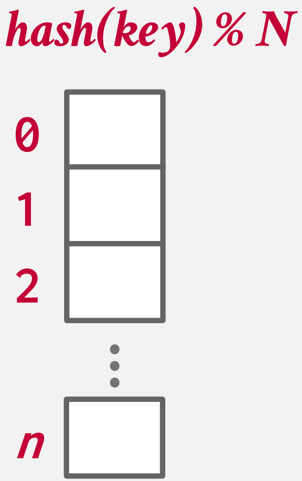
It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.

Space Complexity: **O(n)**
Time Complexity:
→ Average: **O(1)** ⬅ ***Databases care about constants!***
→ Worst: **O(n)**

STATIC HASH TABLE

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

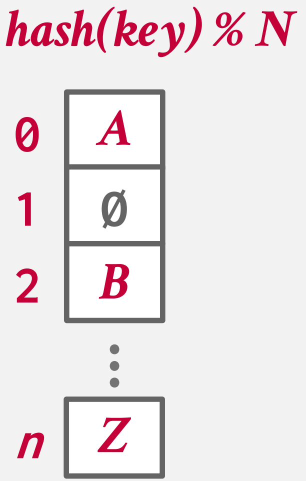
To find an entry, mod the key by the number of elements to find the offset in the array.



STATIC HASH TABLE

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

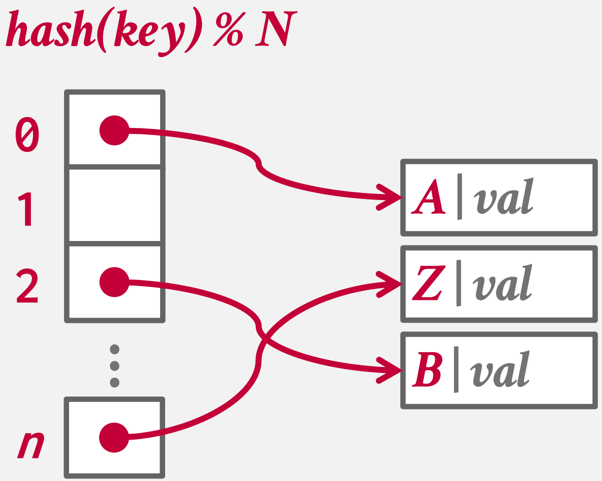
To find an entry, mod the key by the number of elements to find the offset in the array.



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.



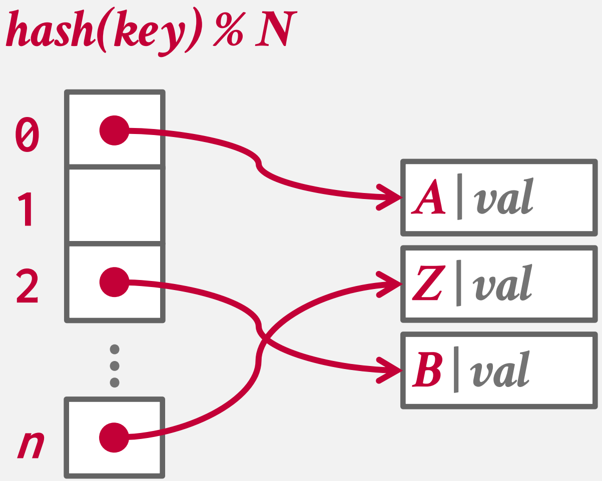
UNREALISTIC ASSUMPTIONS

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

Assumption #2: Each key is unique.

Assumption #3: Perfect hash function guarantees no collisions.

→ 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) ← State-of-the-art

→ From the creator of zstd compression.

Google FarmHash (2014)

→ Newer version of CityHash with better collision rates.

HASH FUNCTIONS

smhasher

SMhasher

Linux Build status build passing build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
donothing32	11149460.06	4.00	-	13	bad seed 0, test NOP
donothing64	11787676.42	4.00	-	13	bad seed 0, test NOP
donothing128	11745060.76	4.06	-	13	bad seed 0, test NOP
NOP_OAAT_read64	11372846.37	14.00	-	47	test NOP
BadHash	769.94	73.97	-	47	bad seed 0, test FAIL
sumhash	10699.57	29.53	-	363	bad seed 0, test FAIL
sumhash32	42877.79	23.12	-	863	UB, test FAIL
multiply_shift	8026.77	26.05	226.80 (8)	345	bad seeds & 0xfffffff0, fails most tests
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fails most tests
crc32	383.12	134.21	257.50 (11)	422	insecure, 8590x collisions, distrib, PerlinNoise
md5_32	350.53	644.31	894.12 (10)	4419	

on.

.

State-of-the-art

n rates.

HASH FUNCTIONS

smhasher

SMhasher

Linux Build status build passing build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	
donothing32	11149460.06	4.00	-	13	bad s
donothing64	11787676.42	4.00	-	13	bad s
donothing128	11745060.76	4.06	-	13	bad s
NOP_OAAT_read64	11372846.37	14.00	-	47	test
BadHash	769.94	73.97	-	47	bad
sumhash	10699.57	29.53	-	363	bad
sumhash32	42877.79	23.12	-	863	UB
multiply_shift	8026.77	26.05	226.80 (8)	345	bac
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fai
crc32	383.12	134.21	257.50 (11)	422	ins
md5_32	350.53	644.31	894.12 (10)	4419	

Summary

I added some SSE assisted hashes and fast intel/arm CRC32-C, AES and SHA HW variants. See also the old https://github.com/aappleby/smhasher/wiki, the improved, but unmaintained fork https://github.com/demerphq/smhasher, and the new improved version SMHasher3 https://gitlab.com/fwojcik/smhasher3.

So the fastest hash functions on x86_64 without quality problems are:

- xxh3low
- wyhash
- ahash64
- t1ha2_atonce
- komihash
- FarmHash (*not portable, too machine specific: 64 vs 32bit, old gcc, ...*)
- halftime_hash128
- Spooky32
- pengyhash
- nmhash32
- mx3
- MUM/mir (*different results on 32/64-bit archs, lots of bad seeds to filter out*)
- fasthash32

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Cuckoo Hashing

There are several other schemes that we will cover in the Advanced DB course:

→ Robin Hood Hashing
→ Hopscotch Hashing
→ Swiss Tables

LINEAR PROBE HASHING

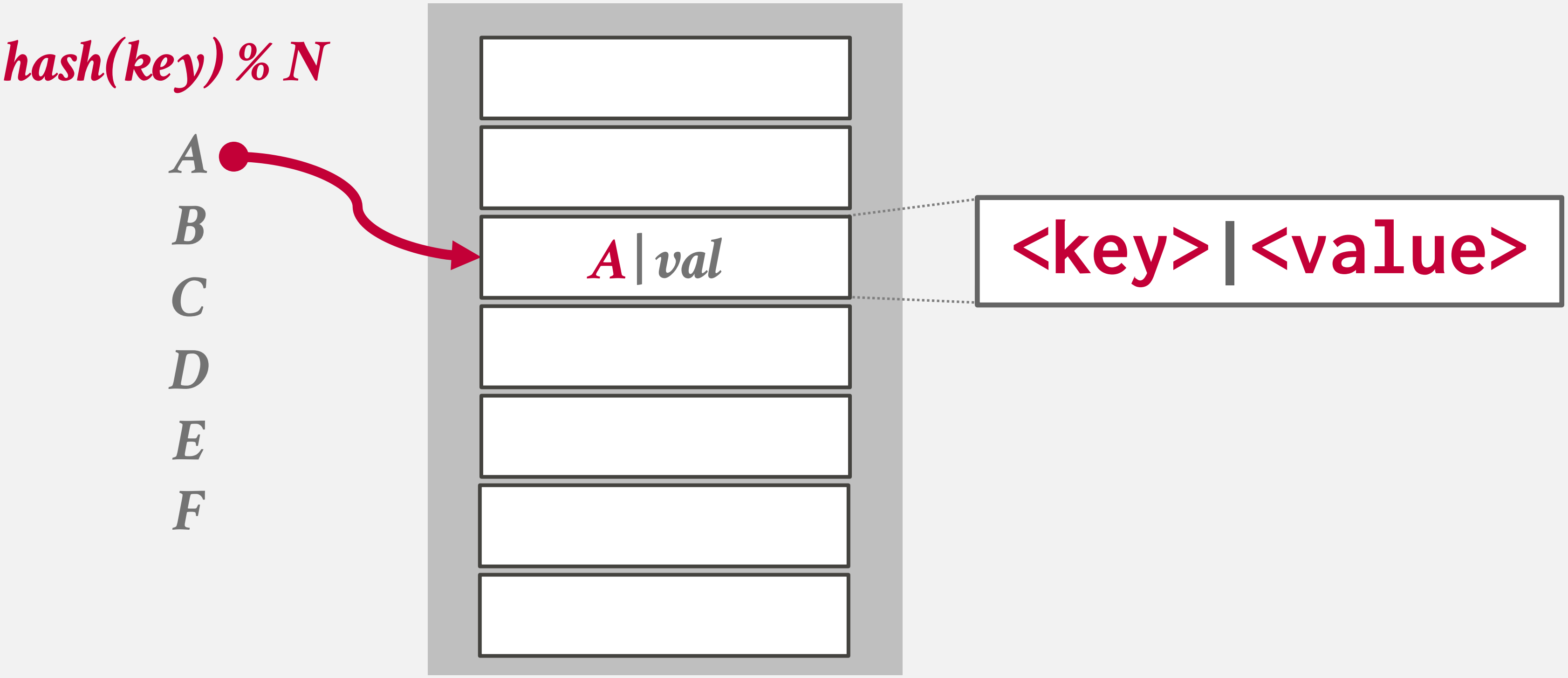
Single giant table of slots.

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

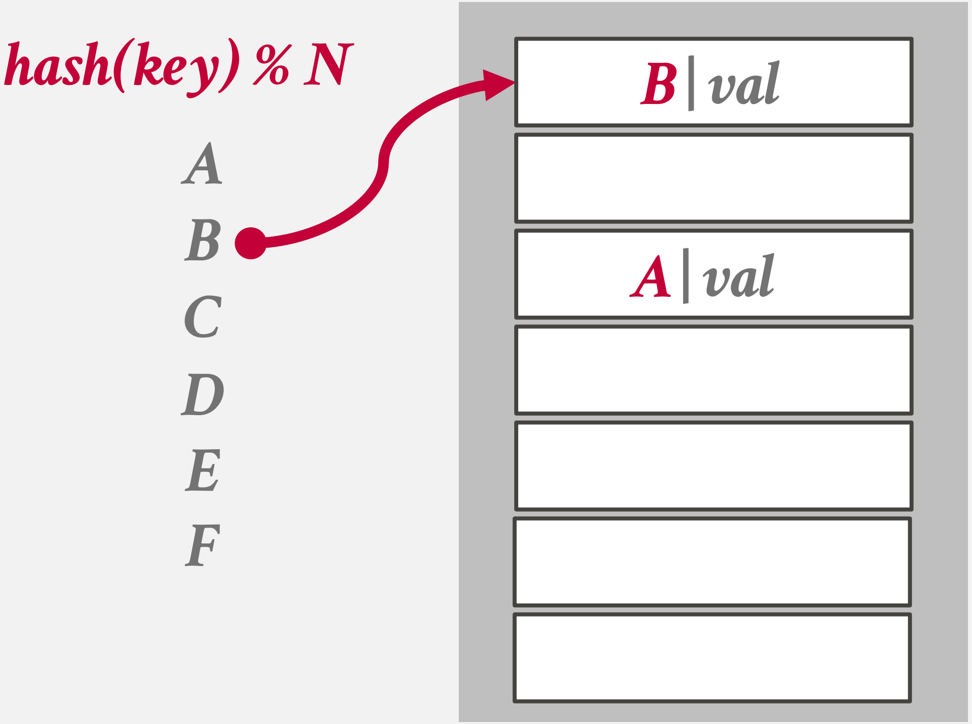
→ To determine whether an element is present, hash to a location in the index and scan for it.
→ Must store the key in the index to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.

Example: Google's `absl::flat_hash_map`

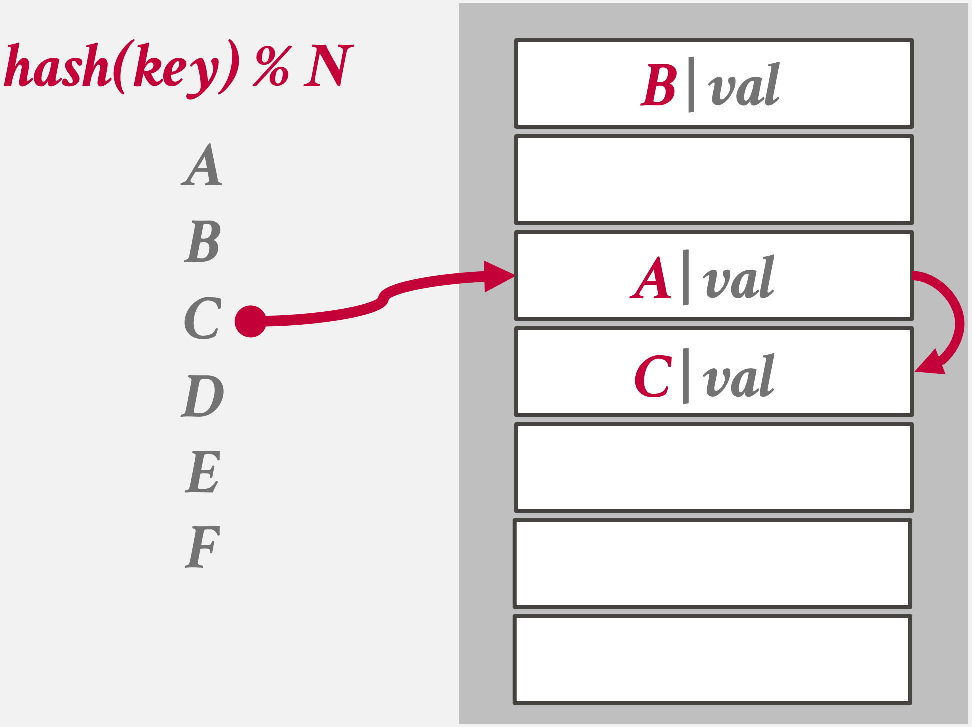
LINEAR PROBE HASHING



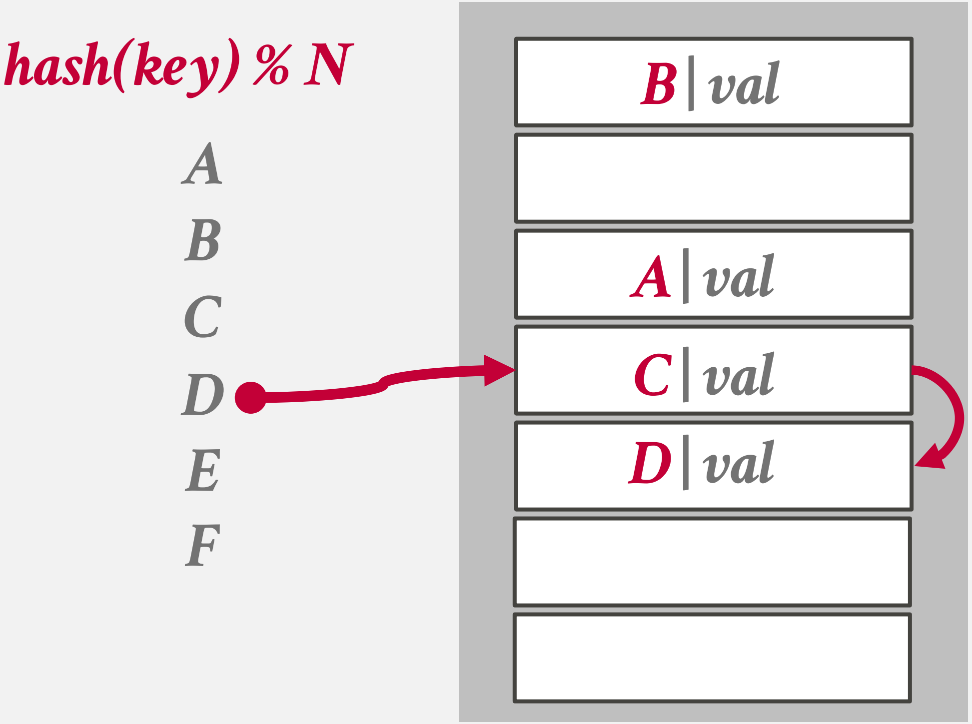
LINEAR PROBE HASHING



LINEAR PROBE HASHING



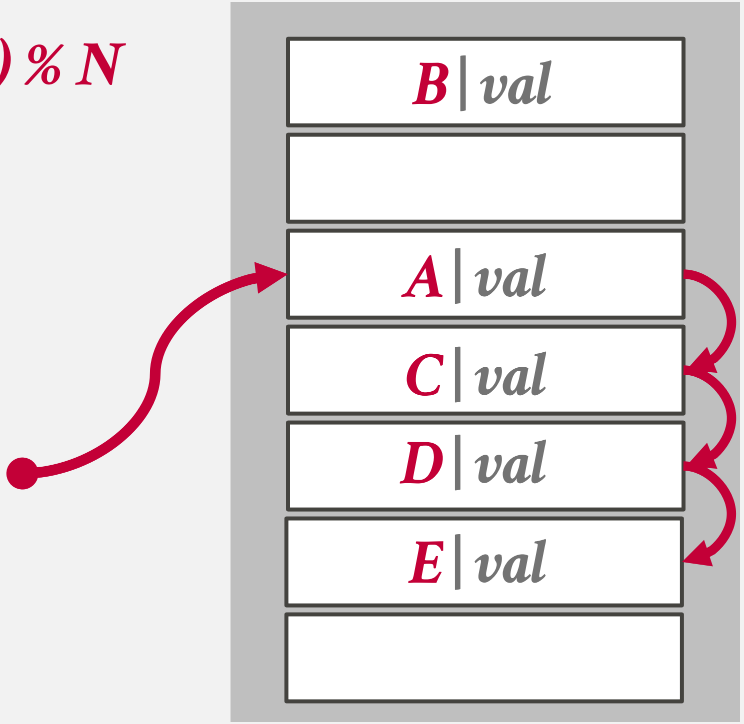
LINEAR PROBE HASHING



LINEAR PROBE HASHING

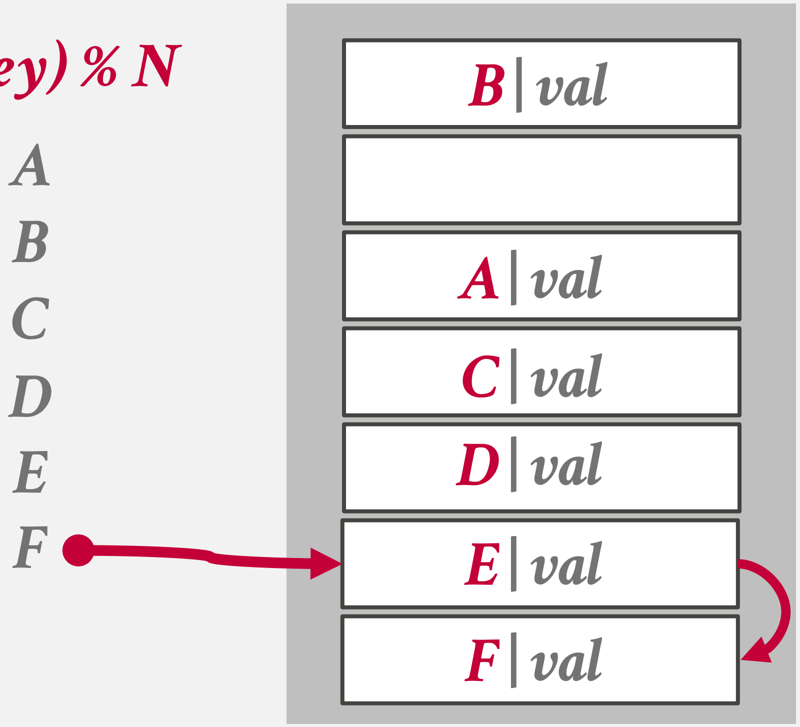
hash(key) % N

A
B
C
D
E
F

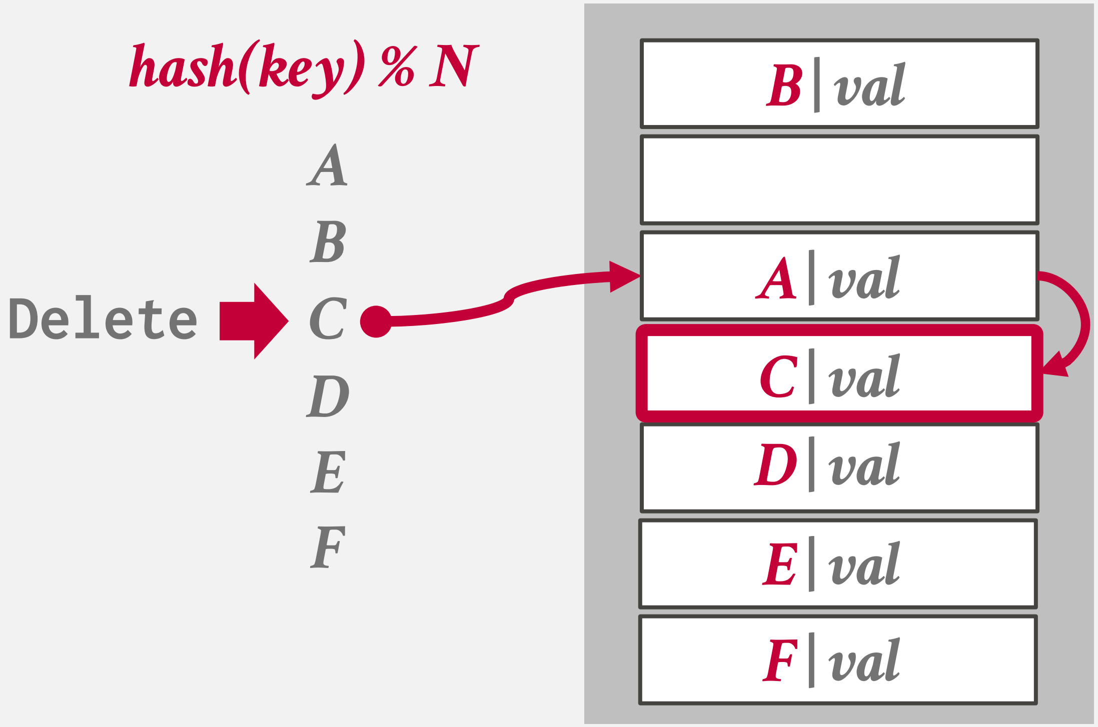


LINEAR PROBE HASHING

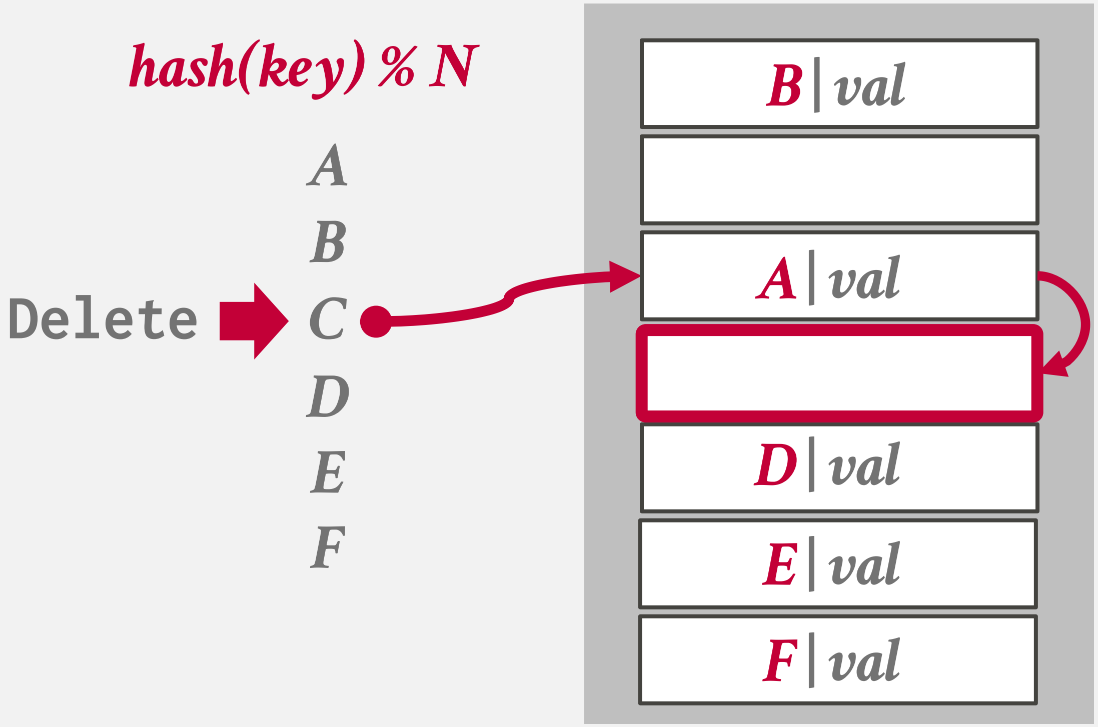
hash(key) % N



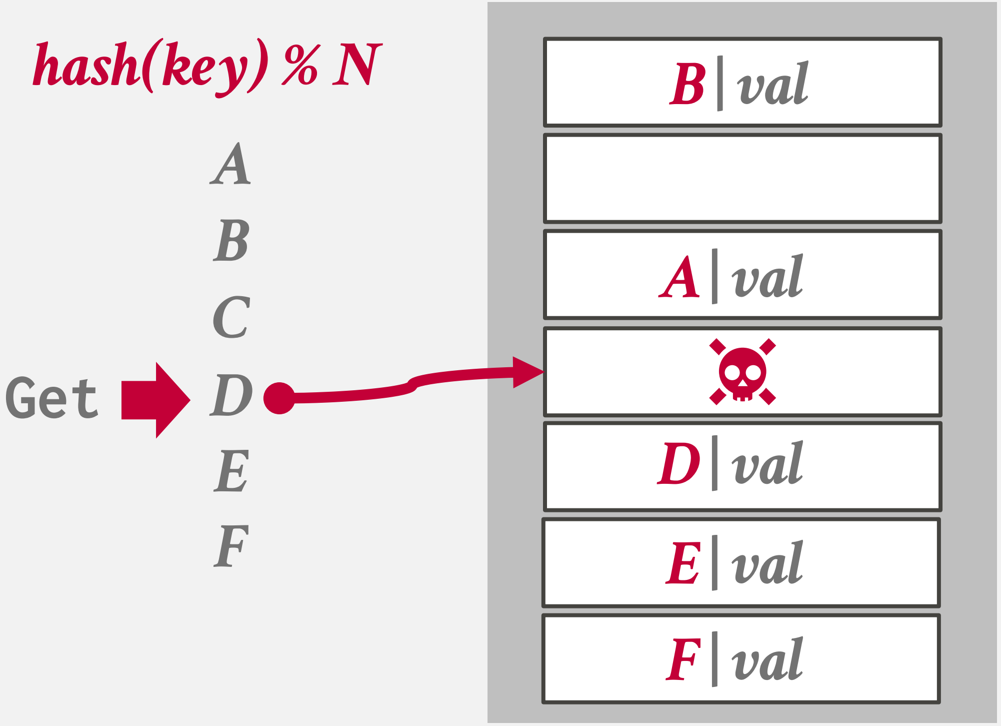
LINEAR PROBE HASHING – DELETES



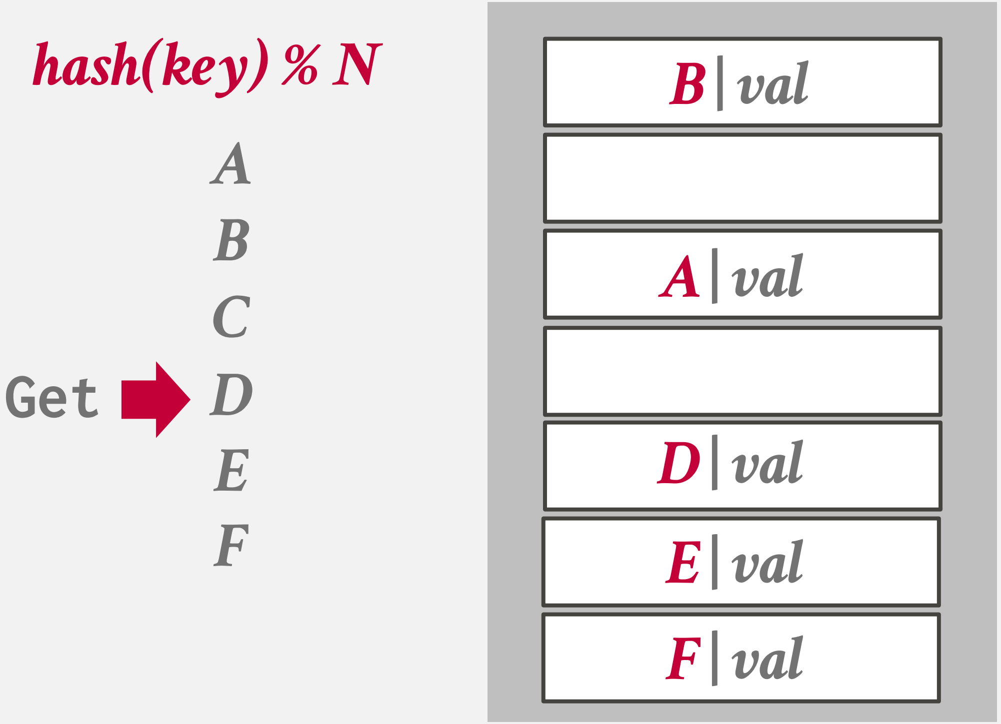
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES



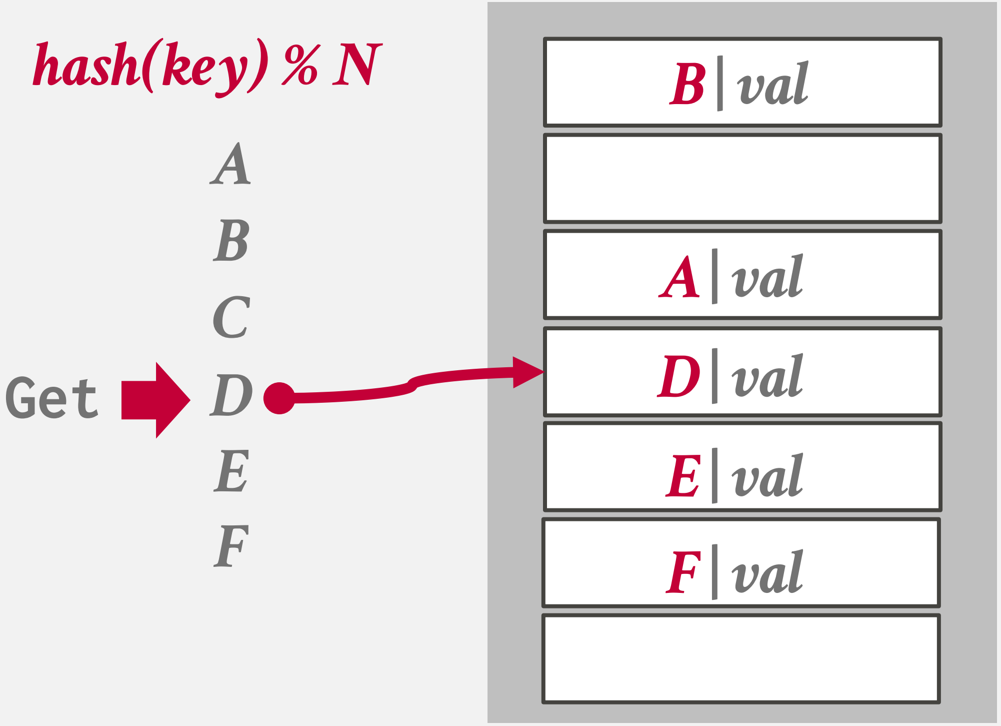
LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

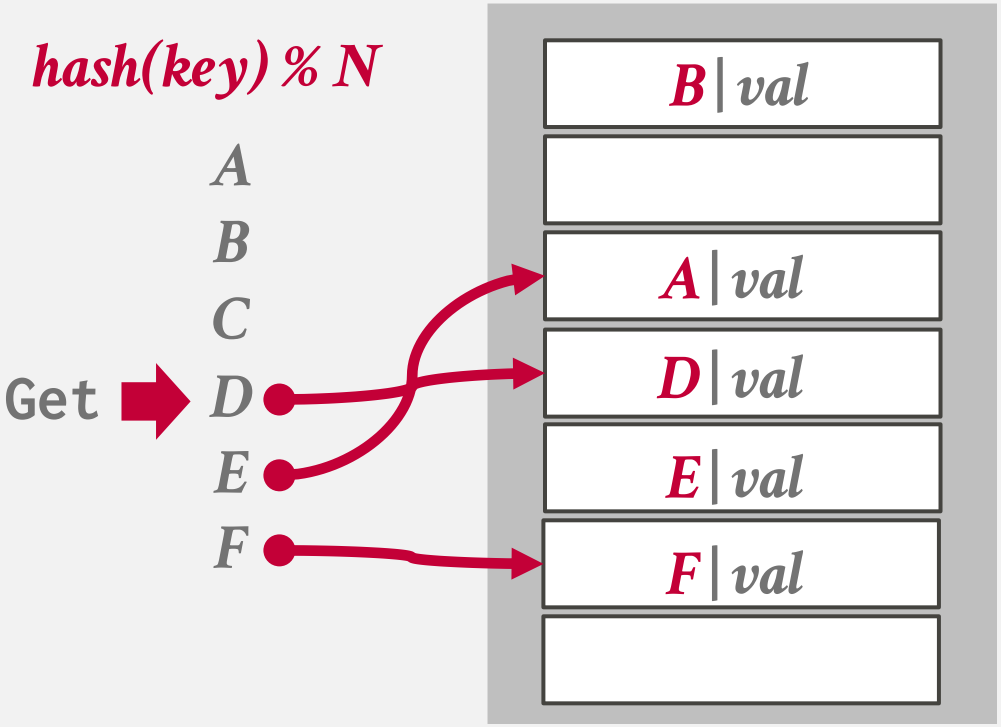
LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

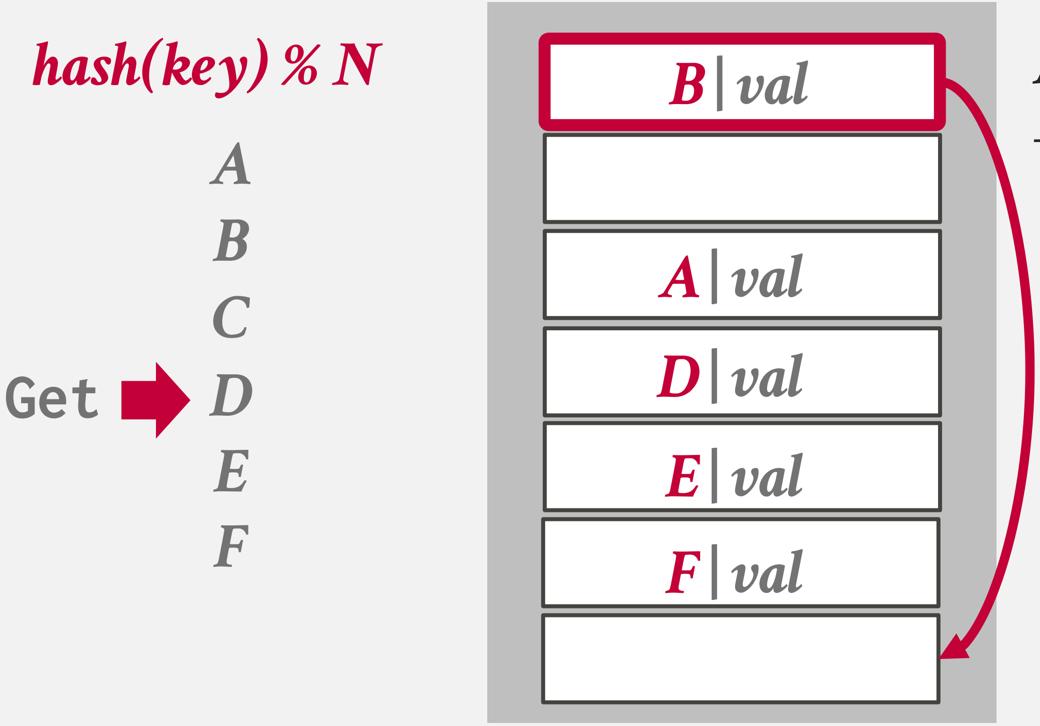
LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

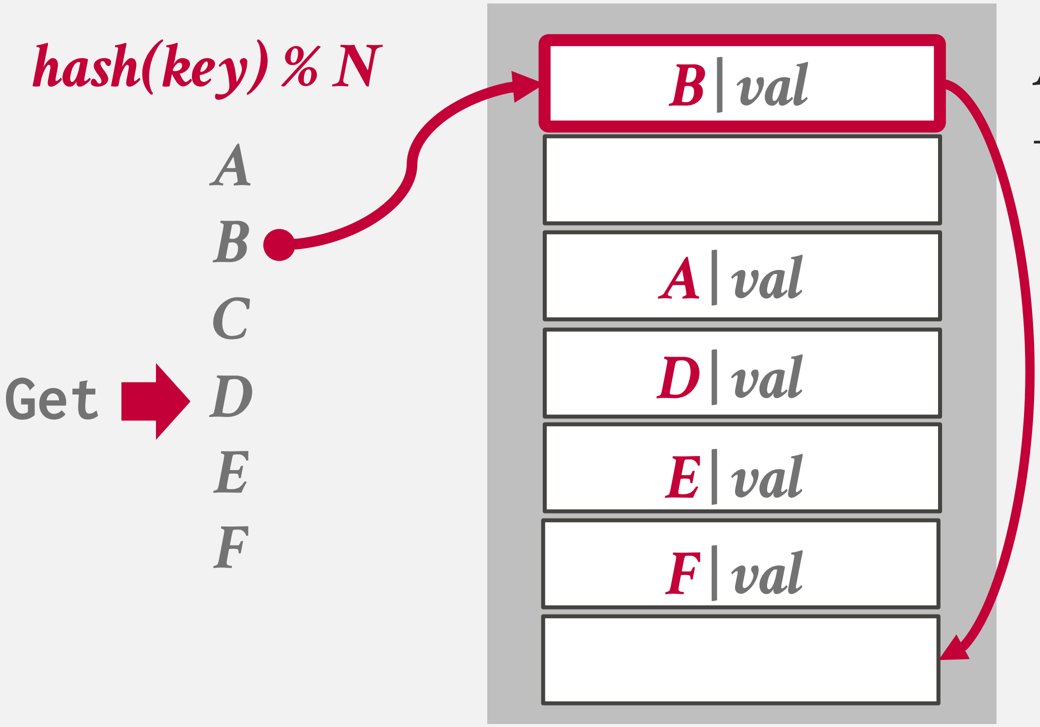
LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

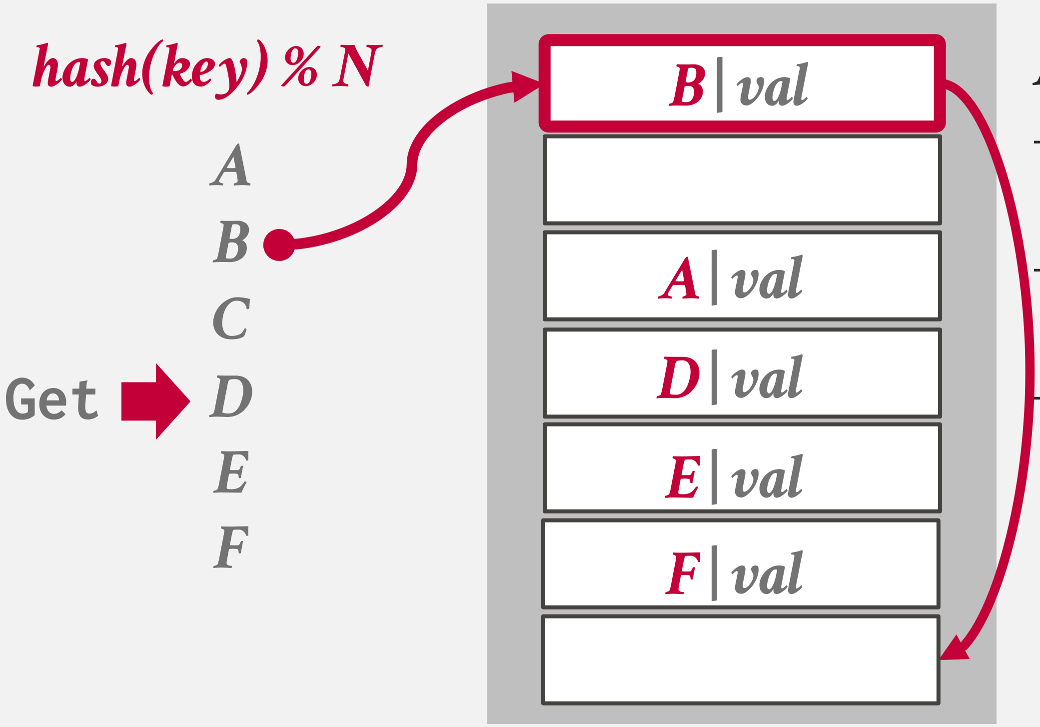
LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

LINEAR PROBE HASHING – DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

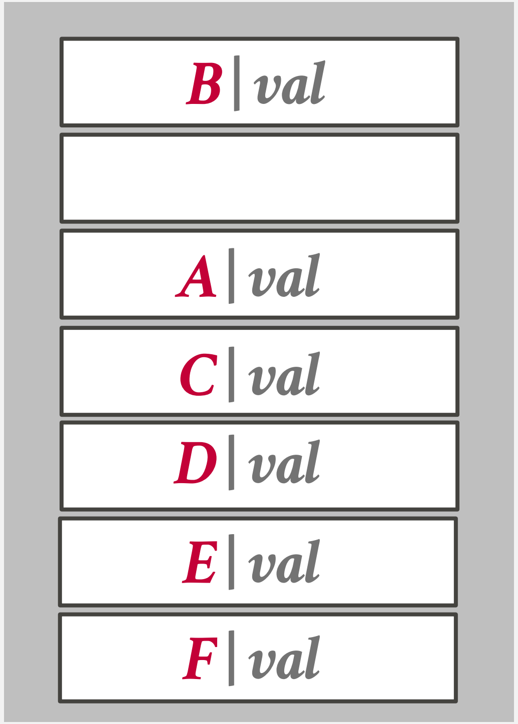
→ Expensive! May need to reorganize the entire table.

→ No DBMS does this.

LINEAR PROBE HASHING – DELETES

hash(key) % N

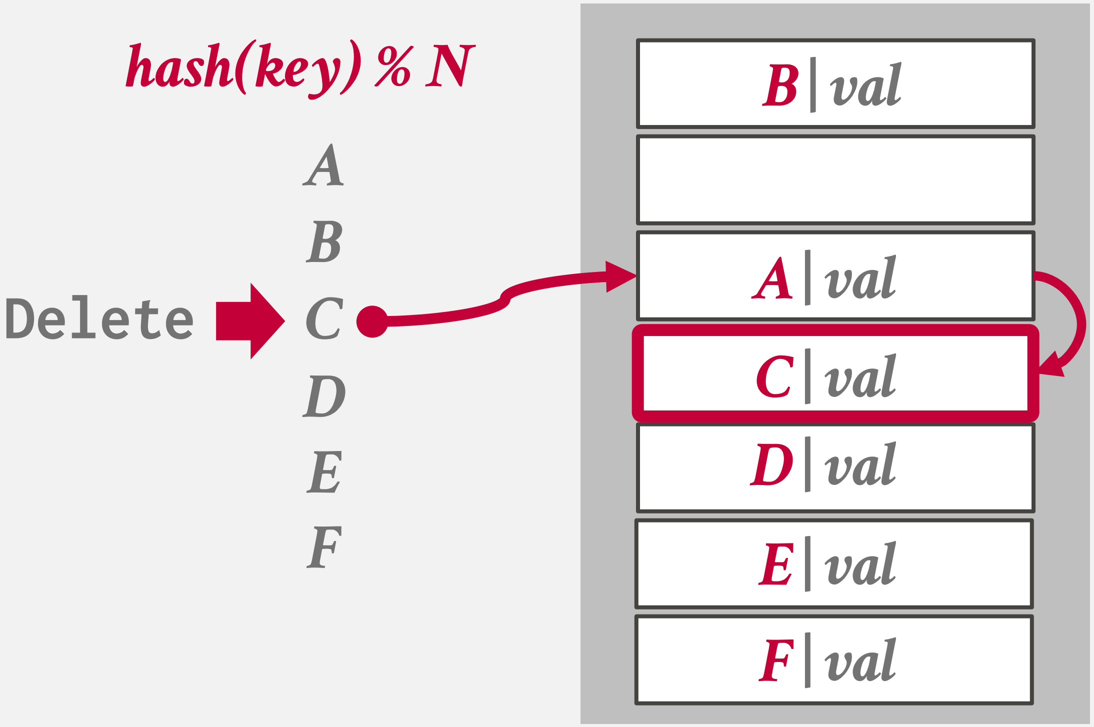
A
B
Delete → *C*
D
E
F



Approach #2: Tombstone

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.

LINEAR PROBE HASHING – DELETES



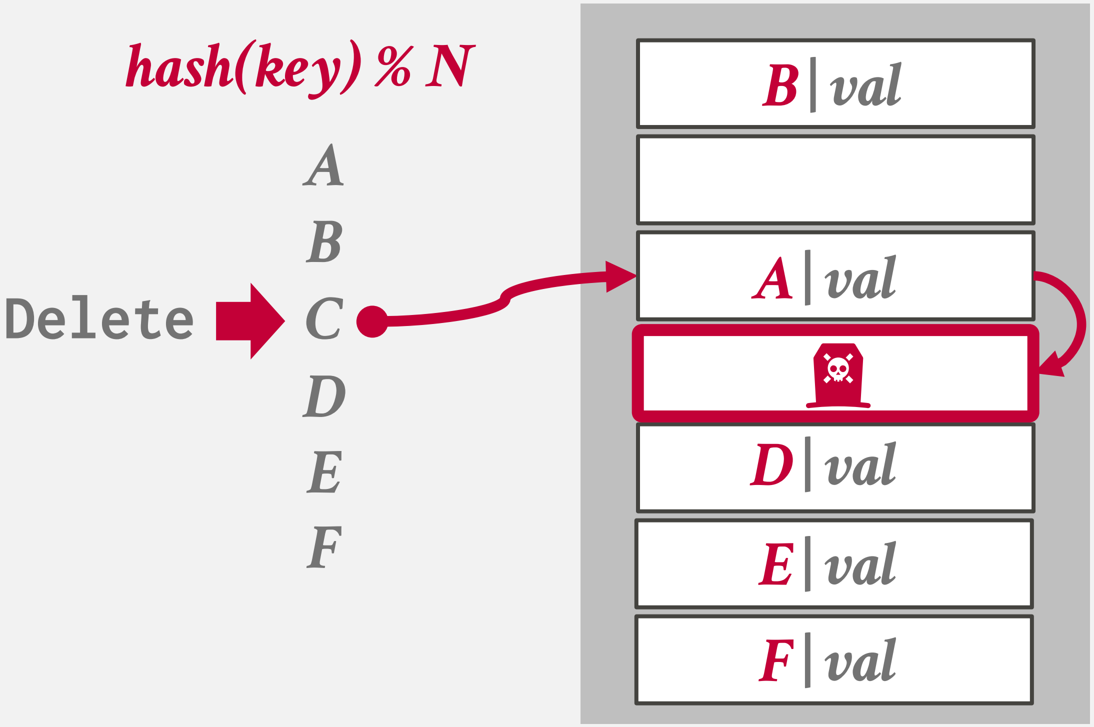
Approach #2: Tombstone

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

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING – DELETES



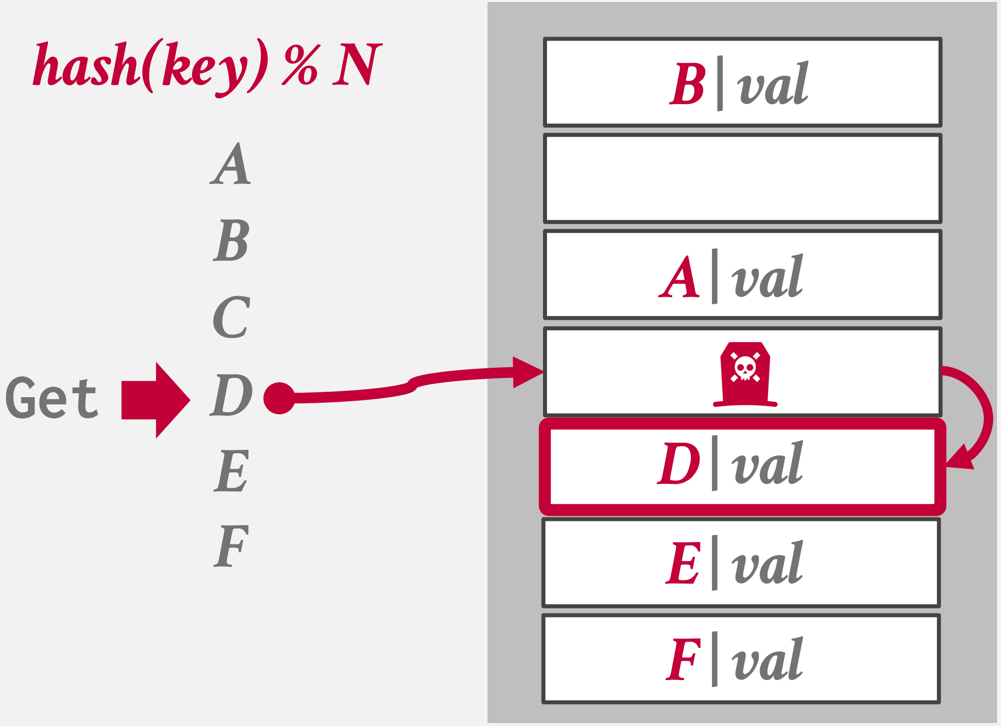
Approach #2: Tombstone

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

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

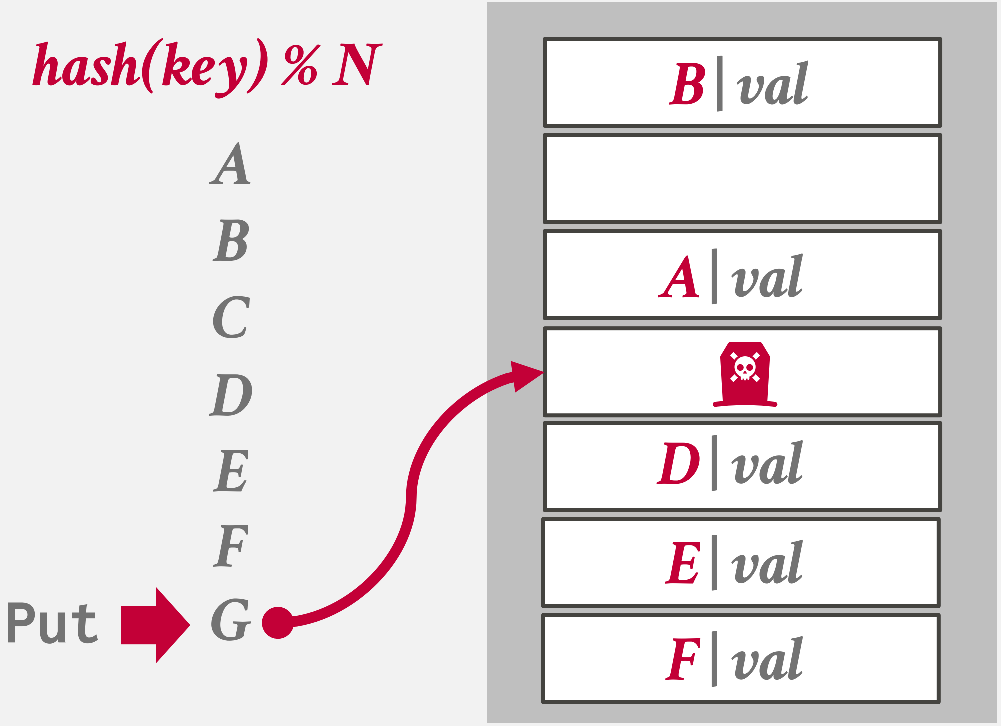
LINEAR PROBE HASHING – DELETES



Approach #2: Tombstone

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.

LINEAR PROBE HASHING – DELETES



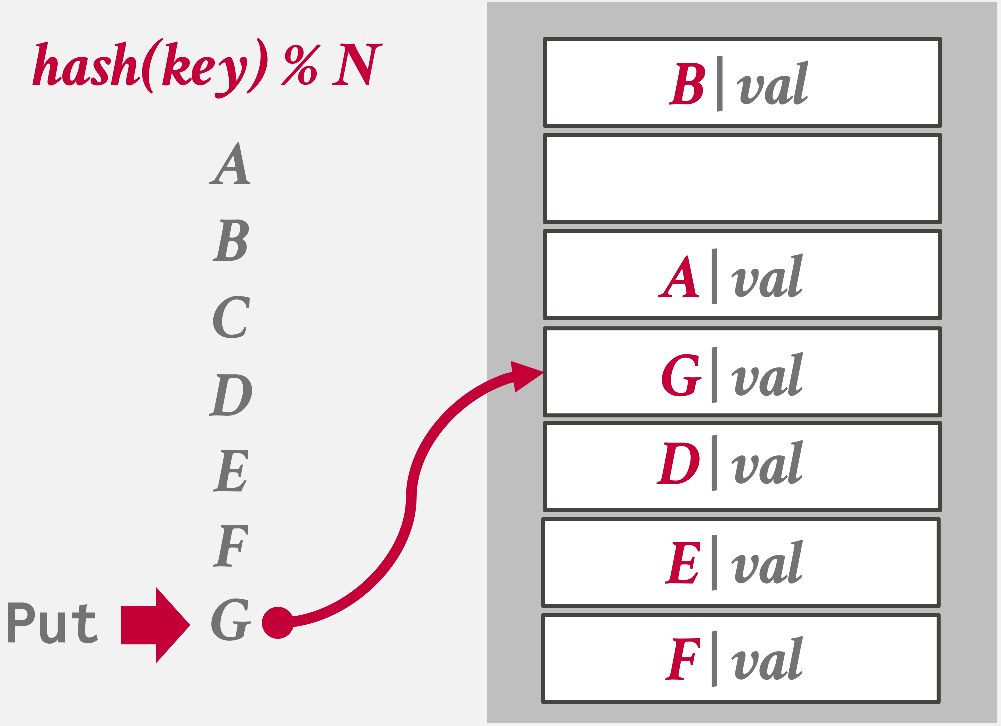
Approach #2: Tombstone

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

→ 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.

→ 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.

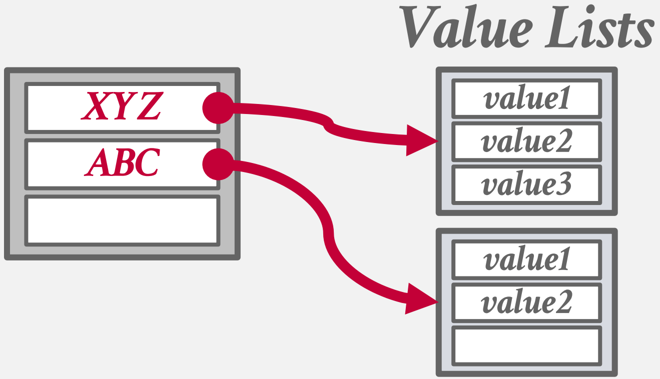
→ Value lists can overflow to multiple pages if the number of duplicates is large.

NON-UNIQUE KEYS

Choice #1: Separate Linked List

→ Store values in separate storage area for each key.

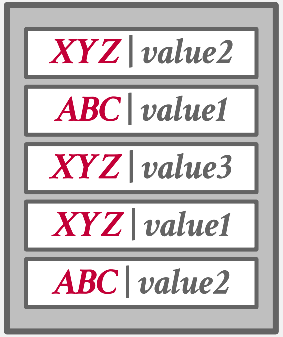
→ Value lists can overflow to multiple pages if the number of duplicates is large.



Choice #2: Redundant Keys

→ Store duplicate keys entries together in the hash table.

→ This is what most systems do.



OPTIMIZATIONS

Specialized hash table implementations based on key type(s) and sizes.

→ Example: Maintain multiple hash tables for different string sizes for a set of keys.

Store metadata separate in a separate array.

→ Packed bitmap tracks whether a slot is empty/tombstone.

Use table + slot versioning metadata to quickly invalidate all entries in the hash table.

→ Example: If table version does not match slot version, then treat the slot as empty.

Source: Maksim Kita

CUCKOO HASHING

Use multiple hash functions to find multiple locations in the hash table to insert records.

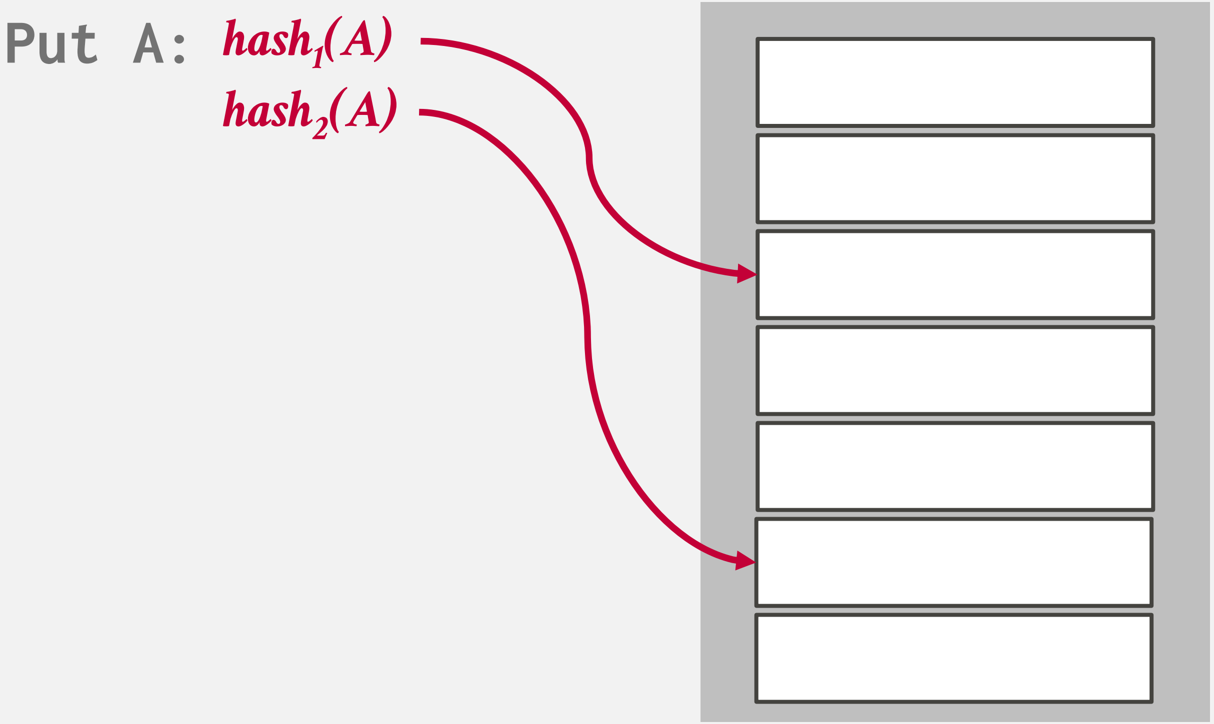
→ On insert, check multiple locations and pick the one that is empty.

→ If no location is available, 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.

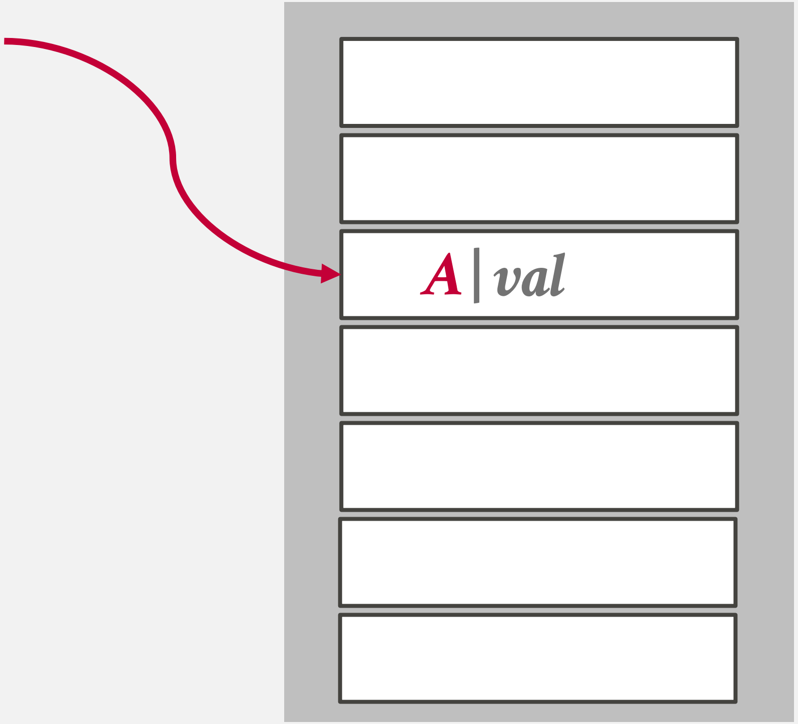
Best open-source implementation is from CMU.

CUCKOO HASHING



CUCKOO HASHING

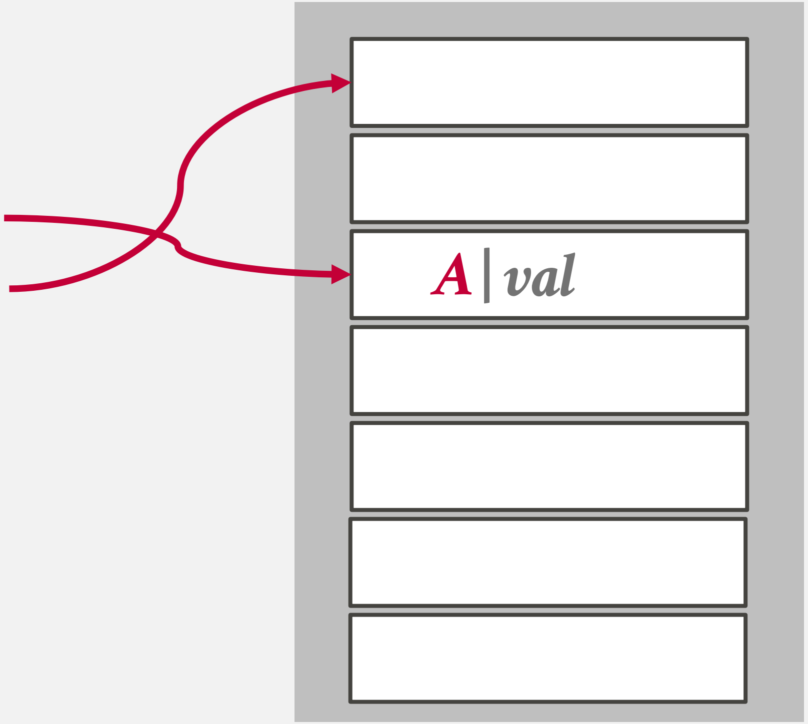
`Put A:` $hash_1(A)$
$hash_2(A)$



CUCKOO HASHING

Put A: *$hash_1(A)$*
$hash_2(A)$

Put B: *$hash_1(B)$*
$hash_2(B)$



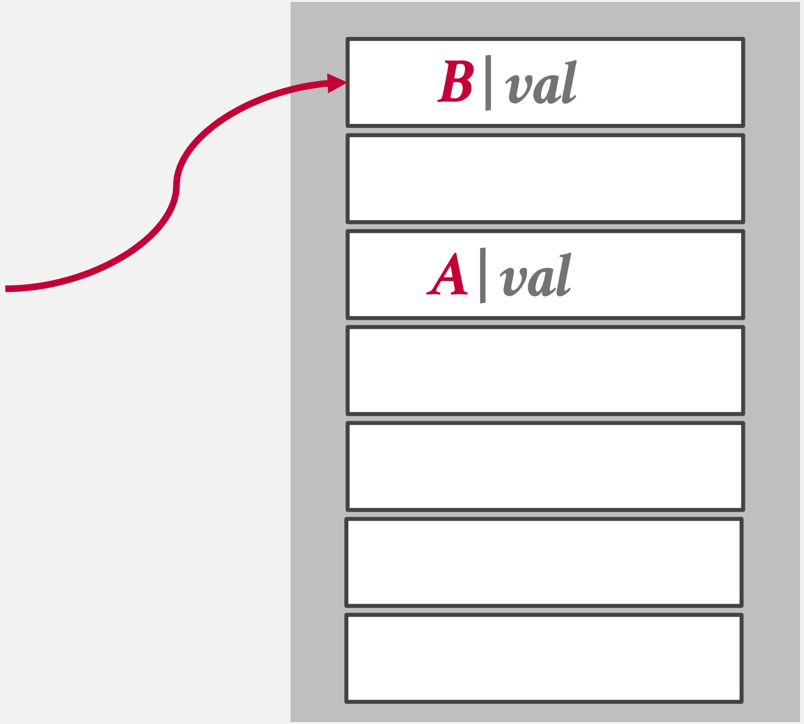
CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$



CUCKOO HASHING

Put A: $hash_1(A)$

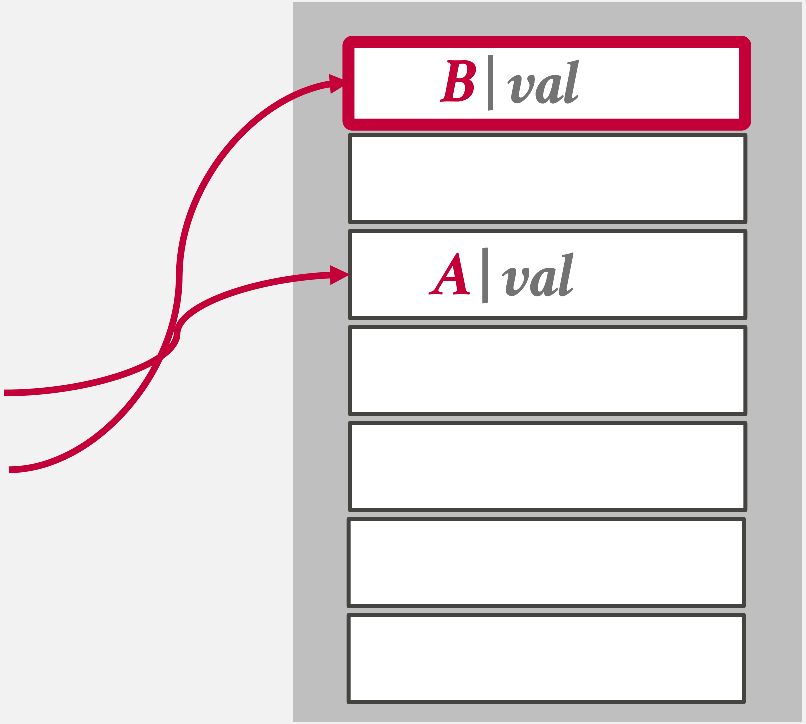
$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$

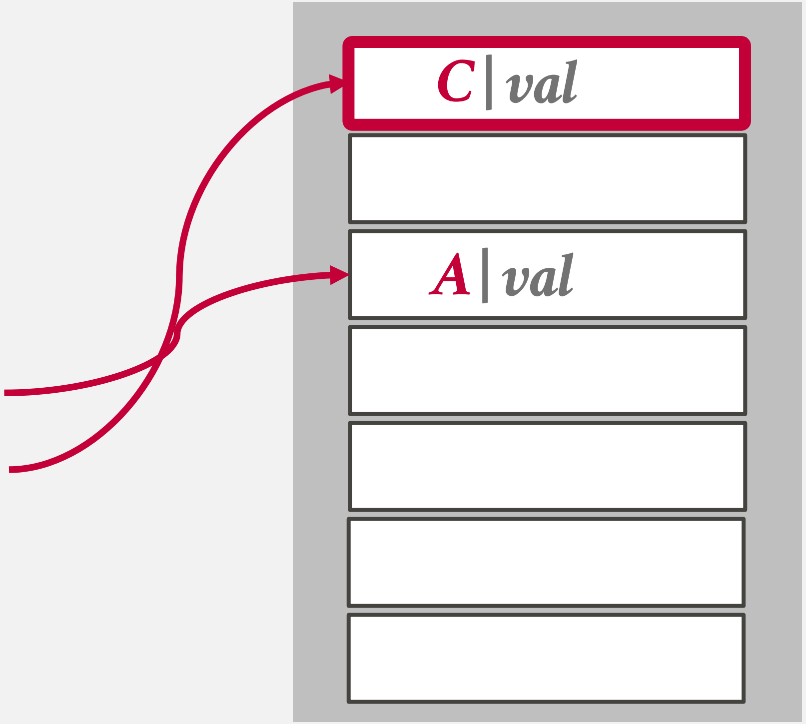
Put C: $hash_1(C)$

$hash_2(C)$



CUCKOO HASHING

Put A: *hash₁(A)* *hash₂(A)*

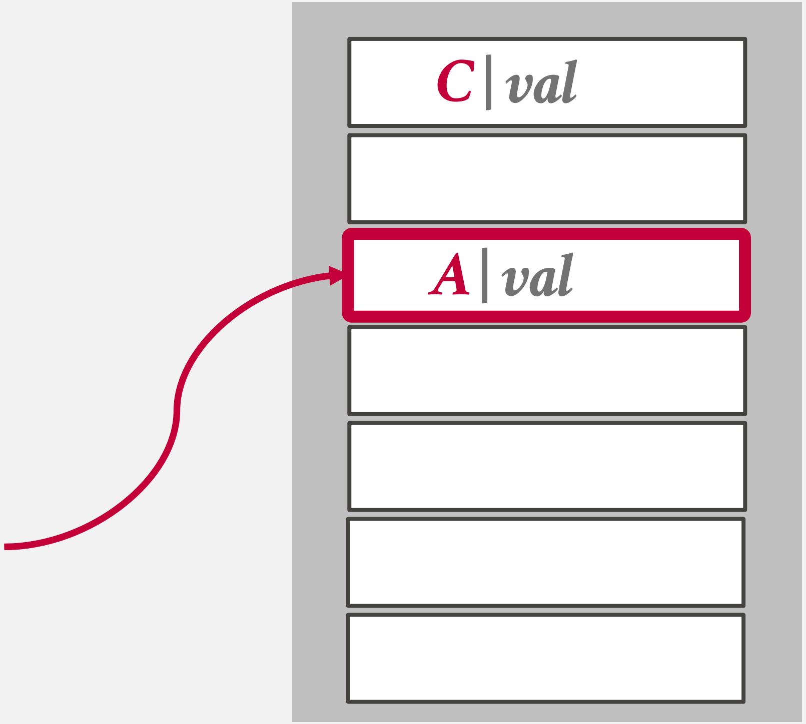


CUCKOO HASHING

Put A: *$hash_1(A)$*
$hash_2(A)$

Put B: *$hash_1(B)$*
$hash_2(B)$

Put C: *$hash_1(C)$*
$hash_2(C)$
$hash_1(B)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

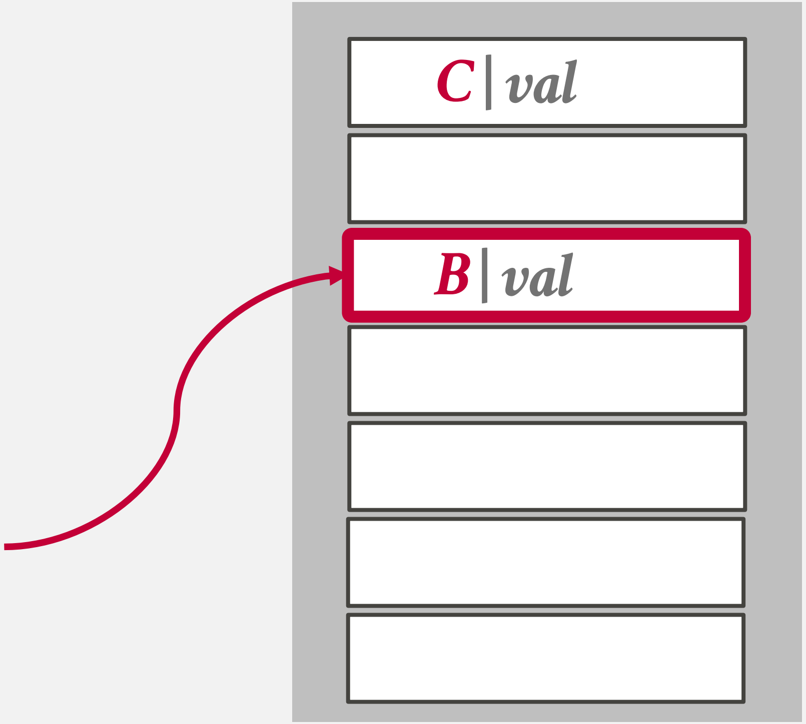
Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$

$hash_1(B)$

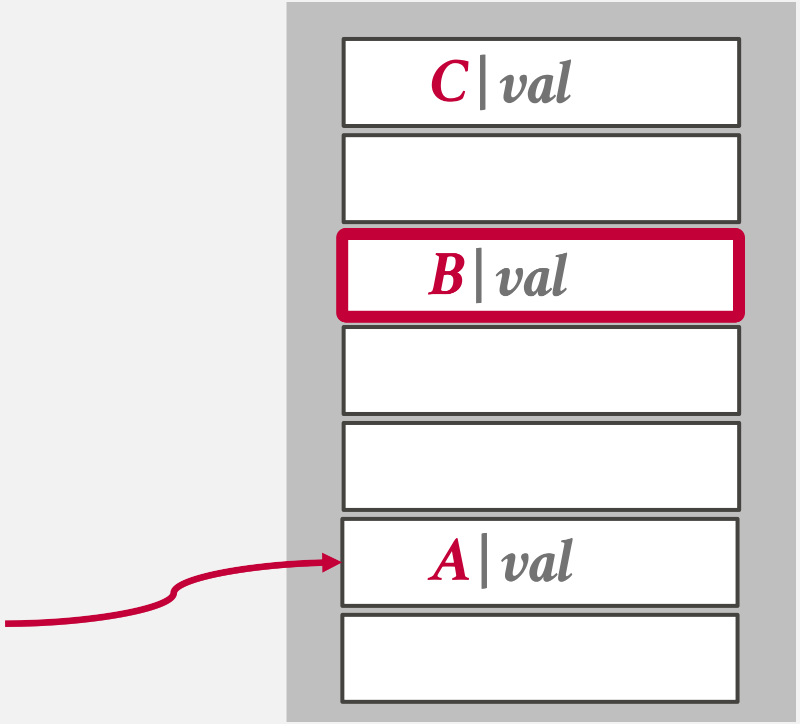


CUCKOO HASHING

Put A: *$hash_1(A)$*
$hash_2(A)$

Put B: *$hash_1(B)$*
$hash_2(B)$

Put C: *$hash_1(C)$*
$hash_2(C)$
$hash_1(B)$
$hash_2(A)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

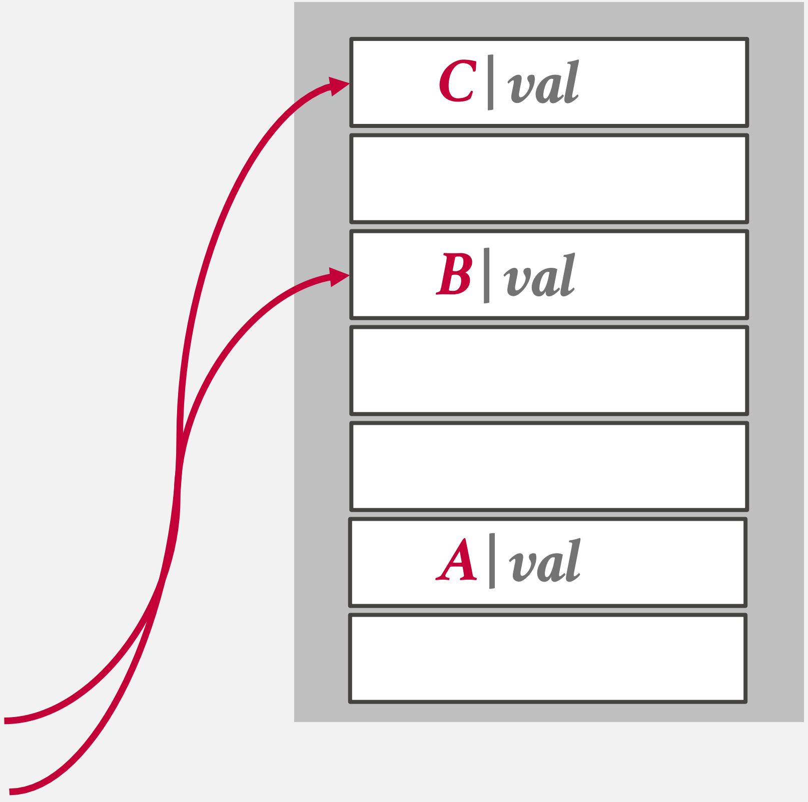
$hash_2(C)$

$hash_1(B)$

$hash_2(A)$

Get B: $hash_1(B)$

$hash_2(B)$



OBSERVATION

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

→ Otherwise, it must rebuild the table if it needs to grow/shrink in size.

Dynamic hash tables incrementally resize themselves as needed.

→ Chained Hashing
→ Extendible Hashing
→ Linear Hashing

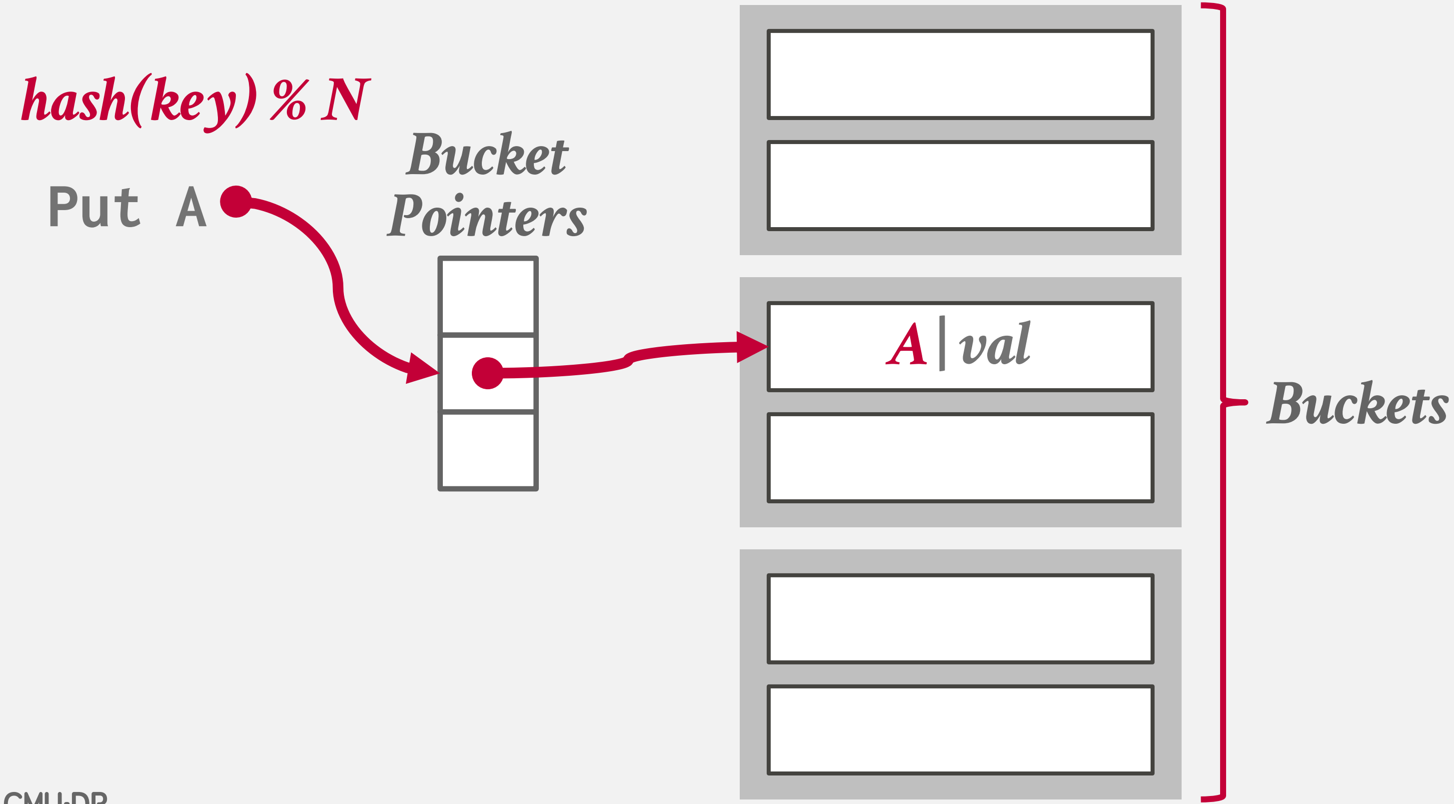
CHAINED HASHING

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

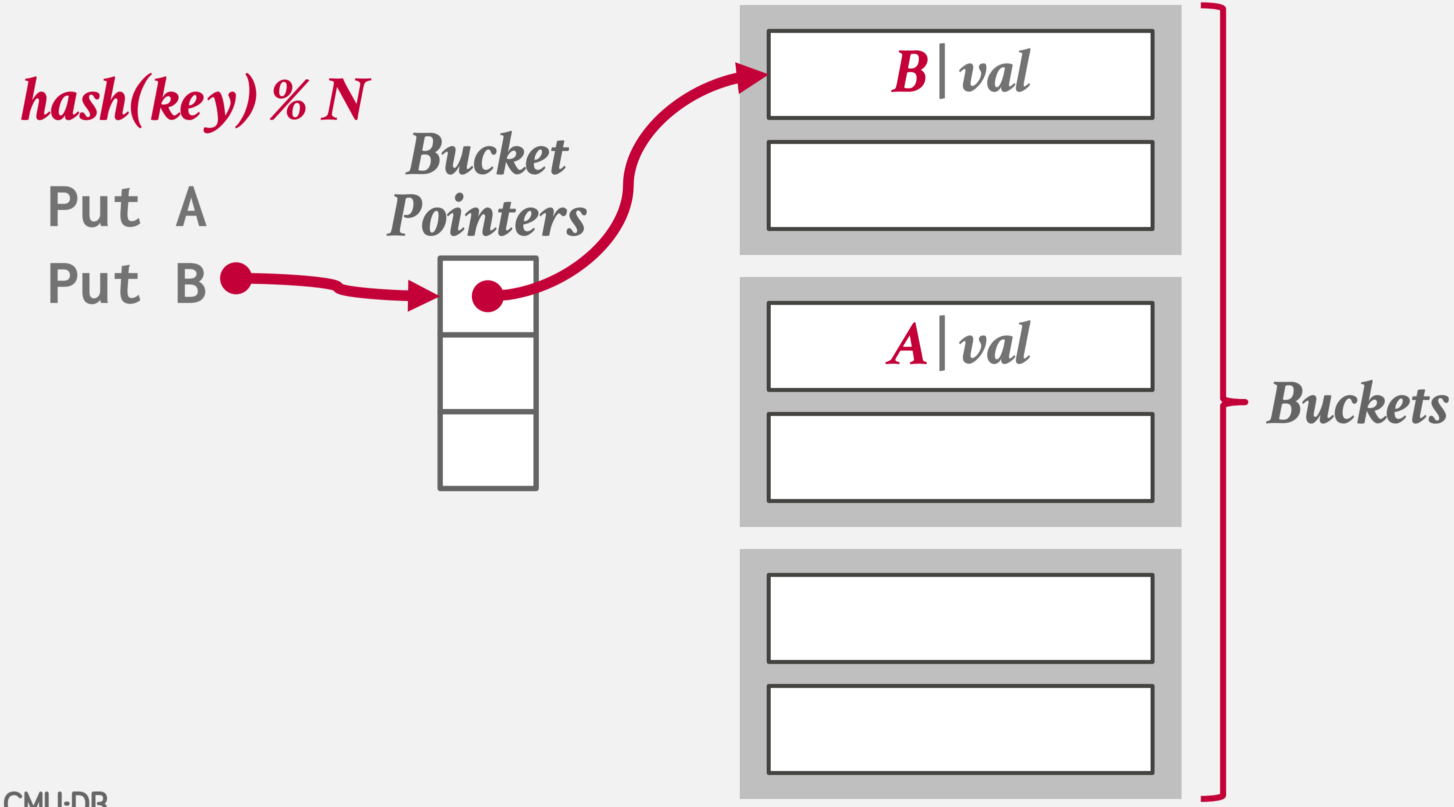
Resolve collisions by placing all elements with the same hash key into the same bucket.

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

CHAINED HASHING



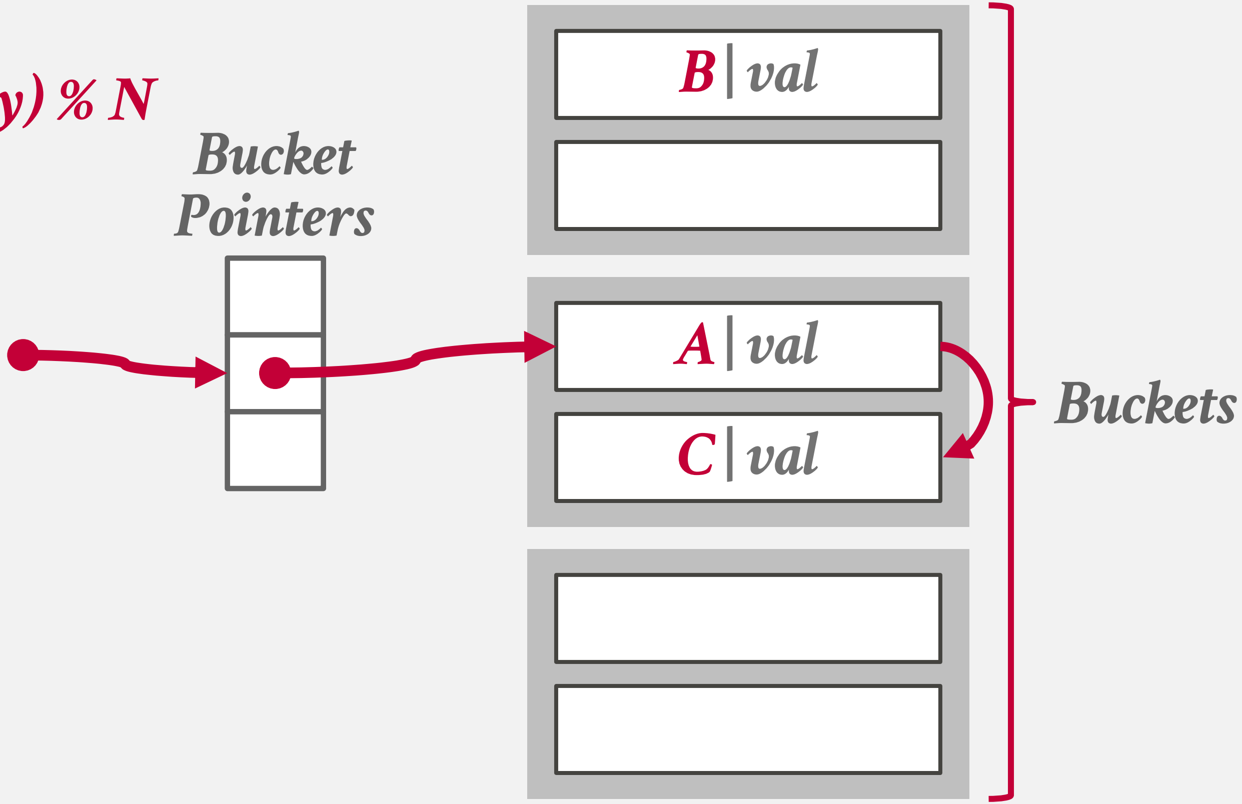
CHAINED HASHING



CHAINED HASHING

hash(key) % N

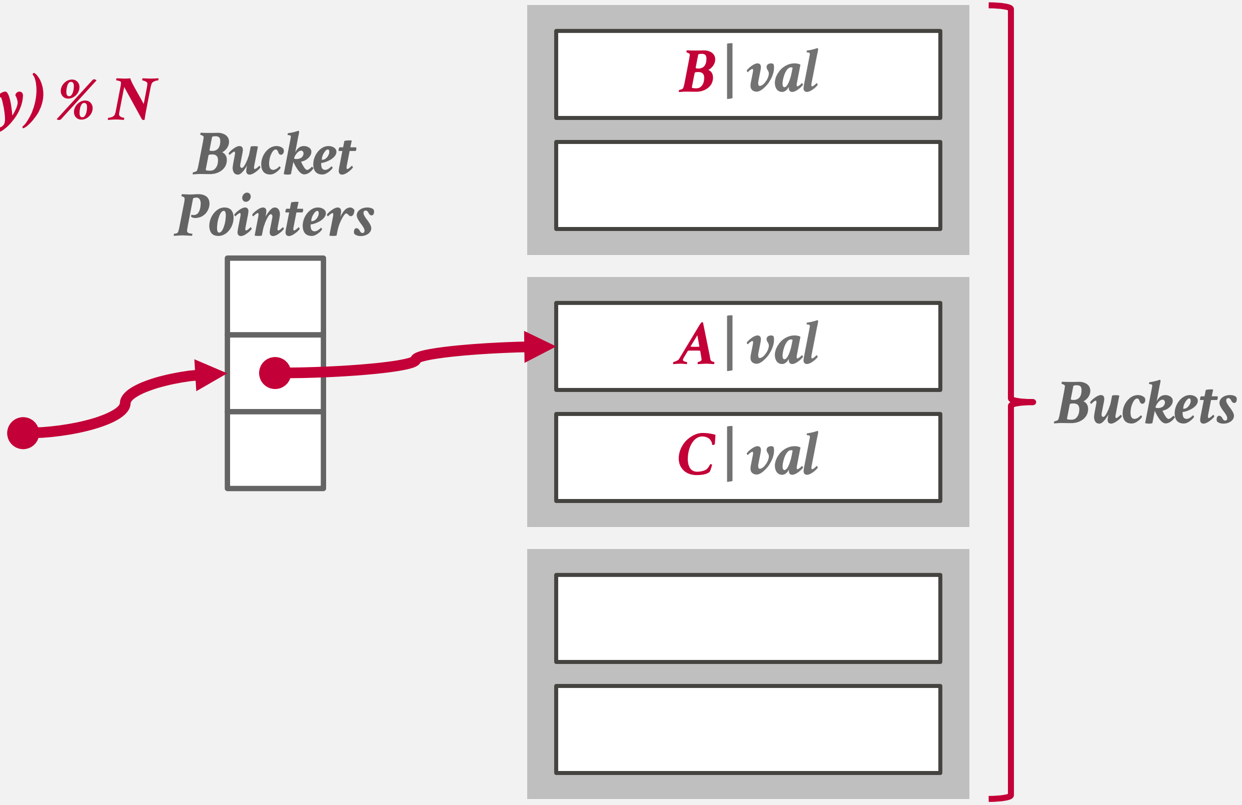
```
Put A
Put B
Put C
```



CHAINED HASHING

hash(key) % N

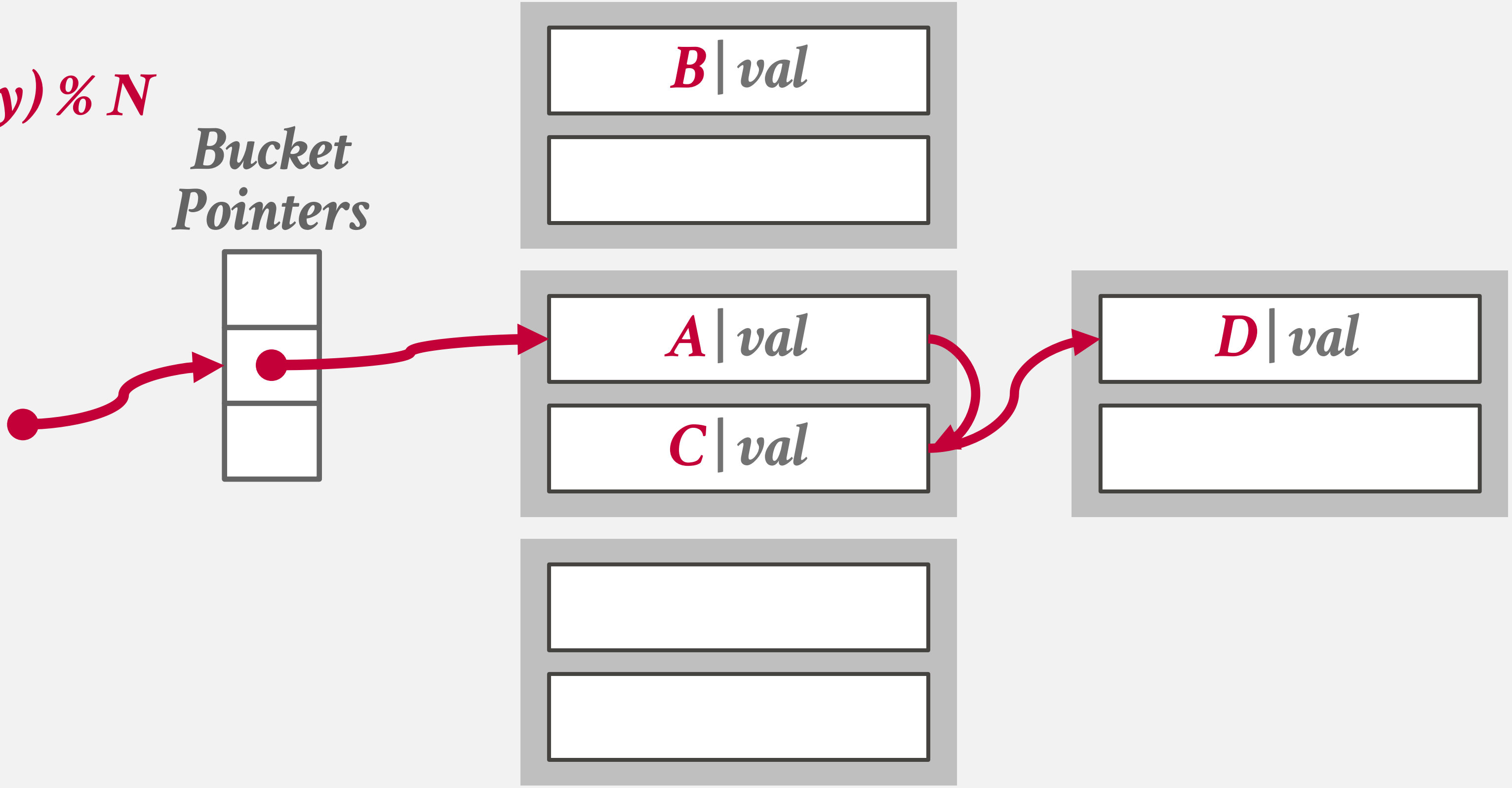
Put A
Put B
Put C
Put D



CHAINED HASHING

hash(key) % N

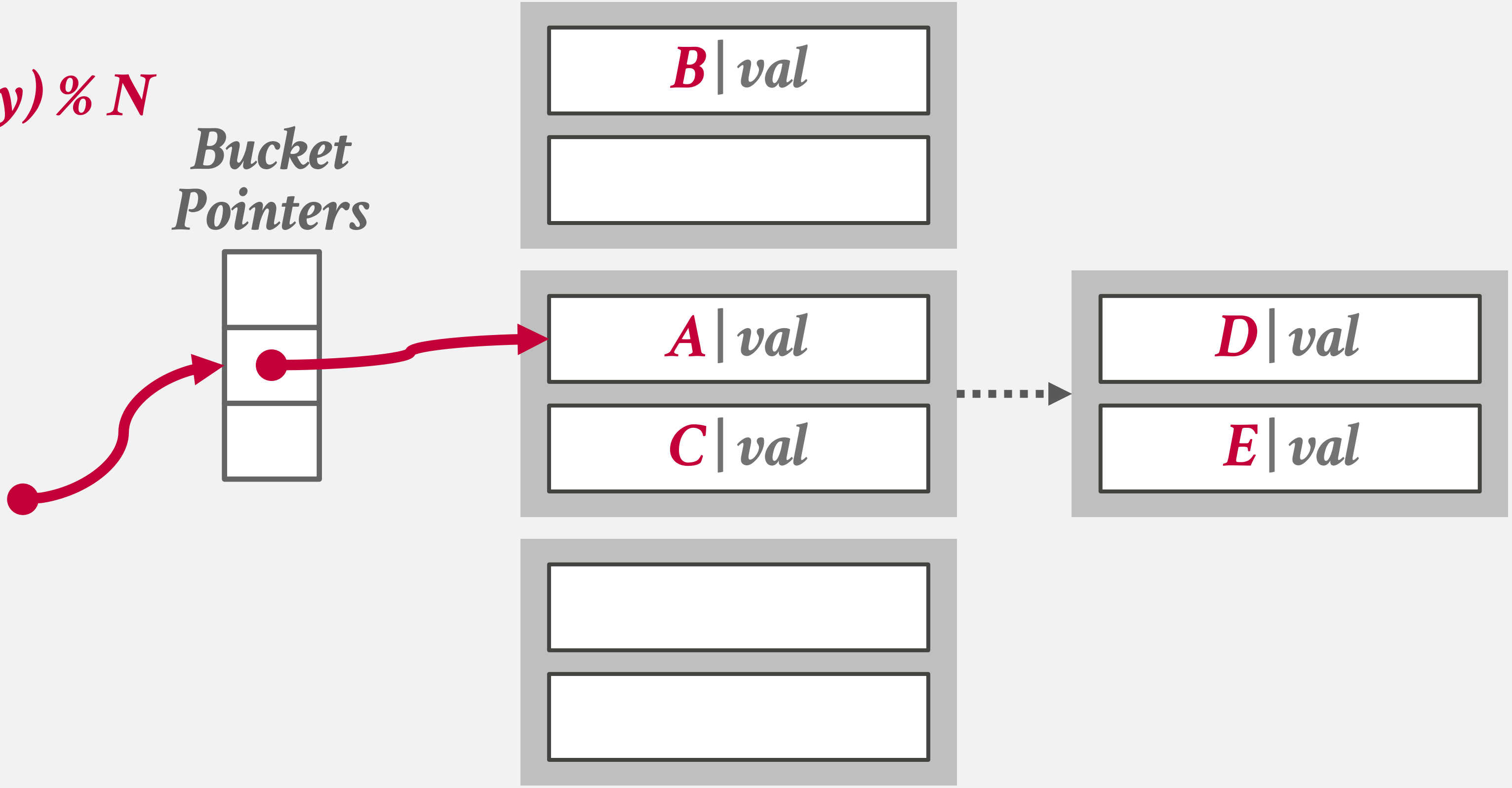
Put A
Put B
Put C
Put D



CHAINED HASHING

hash(key) % N

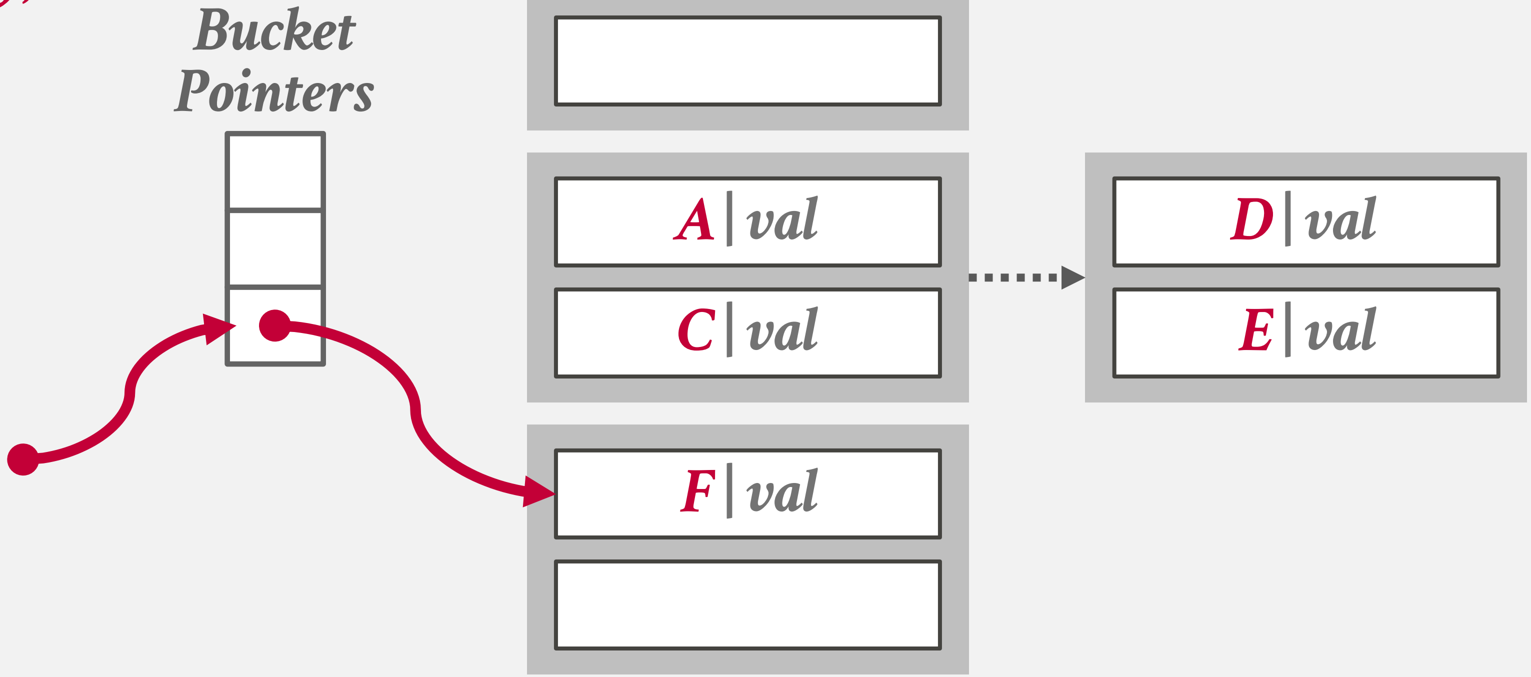
```
Put A
Put B
Put C
Put D
Put E
```



CHAINED HASHING

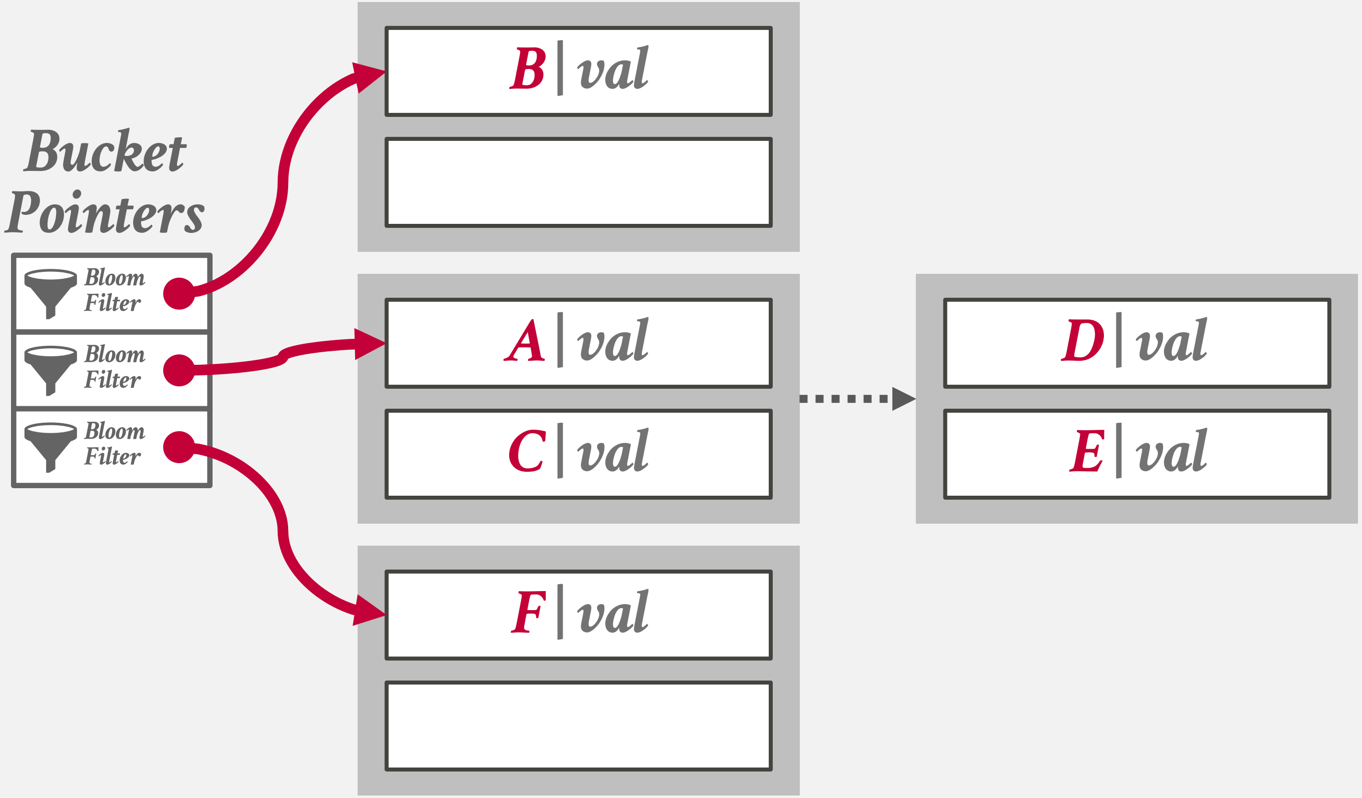
hash(key) % N

Put A
Put B
Put C
Put D
Put E
Put F

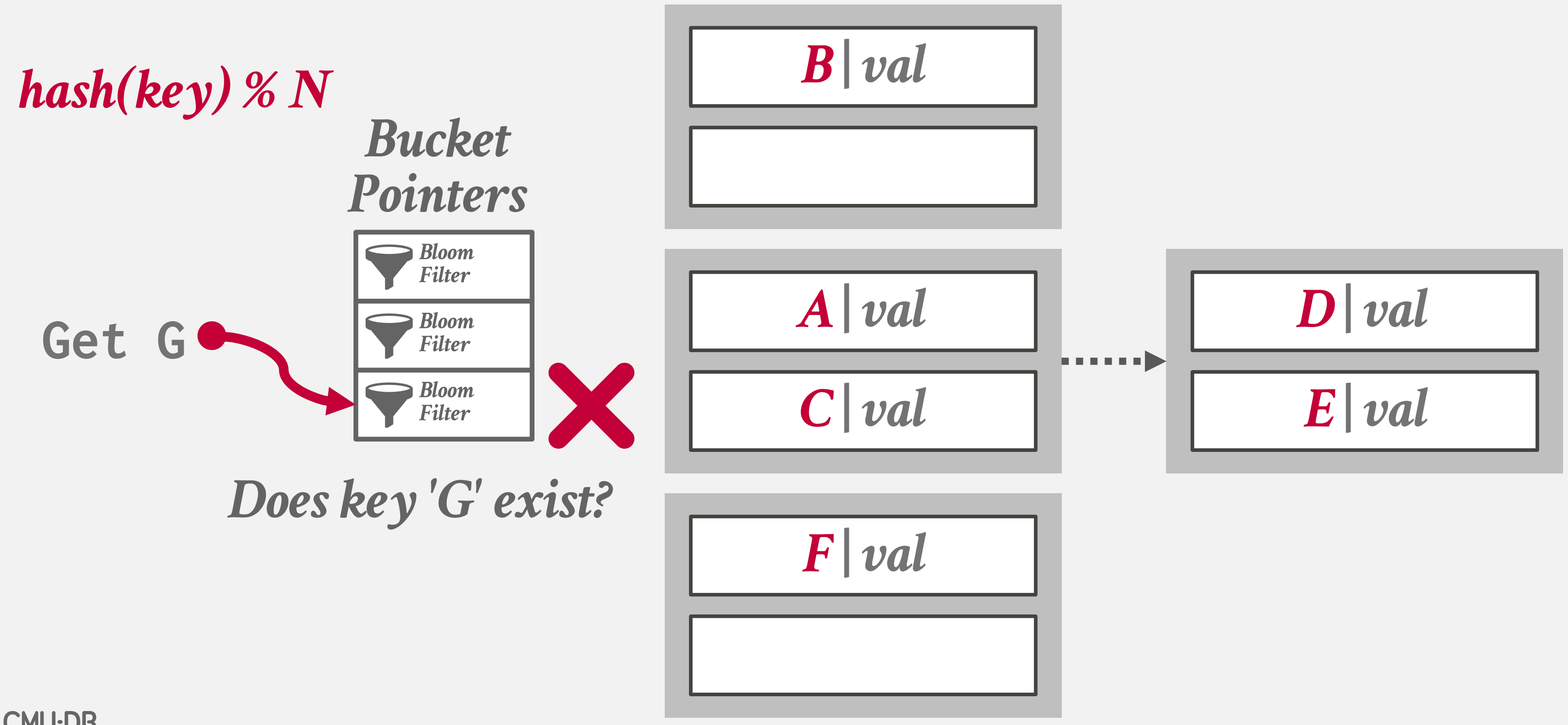


CHAINED HASHING

hash(key) % N



CHAINED HASHING



EXTENDIBLE HASHING

Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

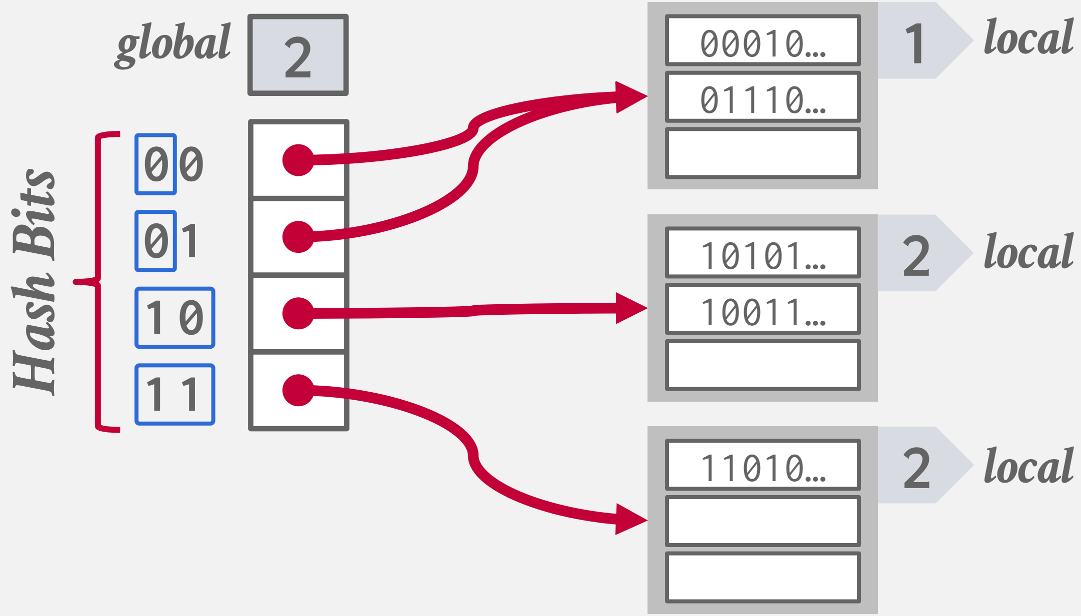
Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

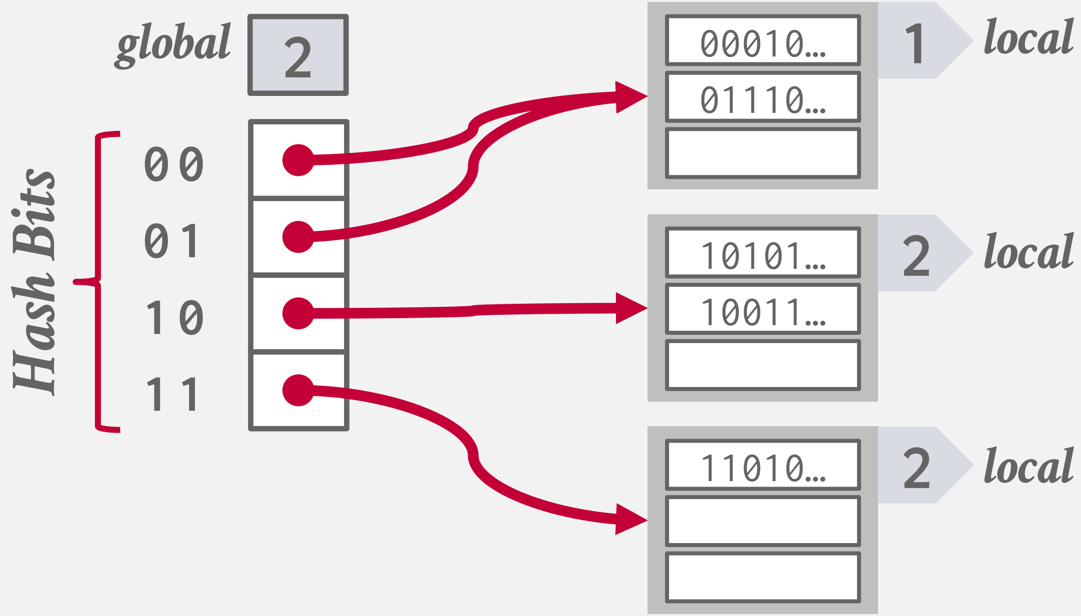
GDBM

AsterixDB

EXTENDIBLE HASHING



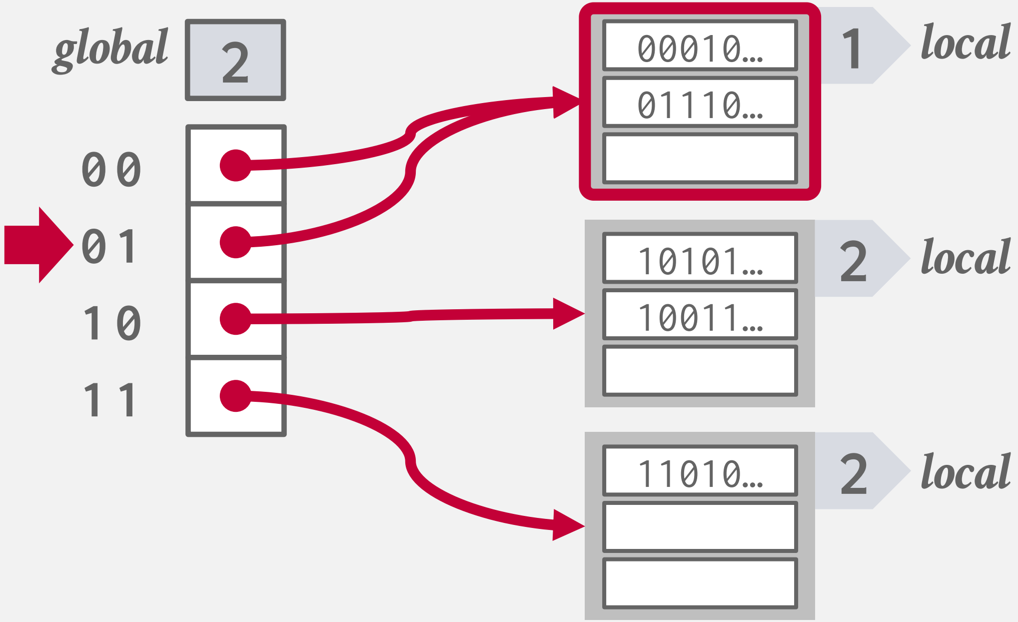
EXTENDIBLE HASHING



Get A

hash(A) = 01110…

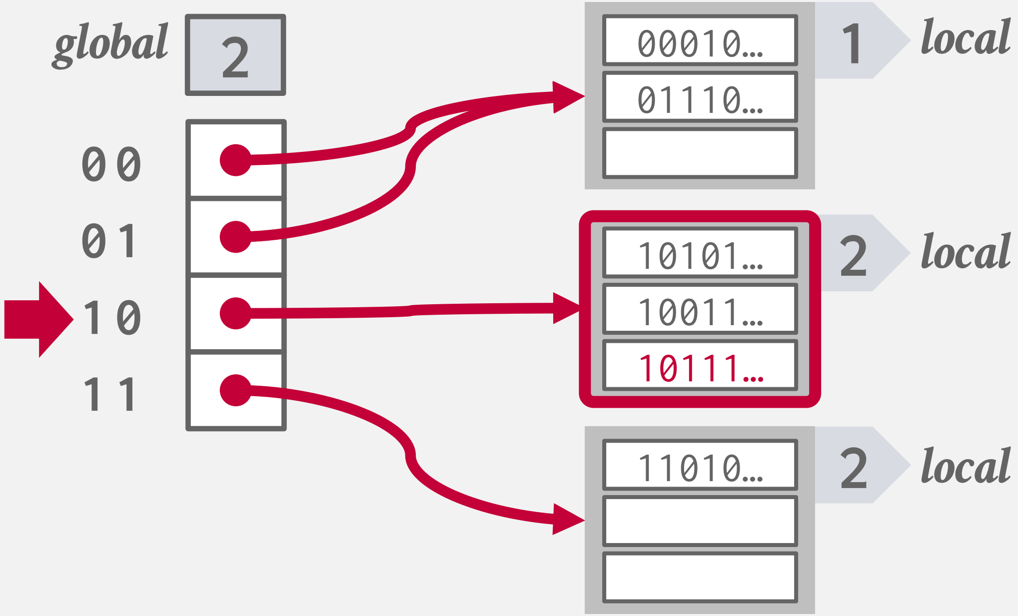
EXTENDIBLE HASHING



Get A

hash(A) = 01110...

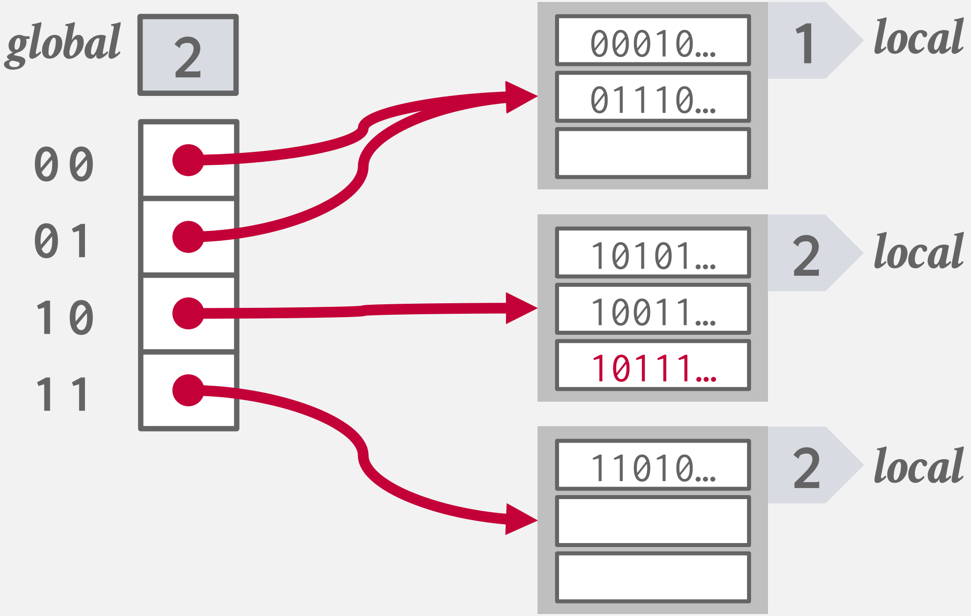
EXTENDIBLE HASHING



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

EXTENDIBLE HASHING

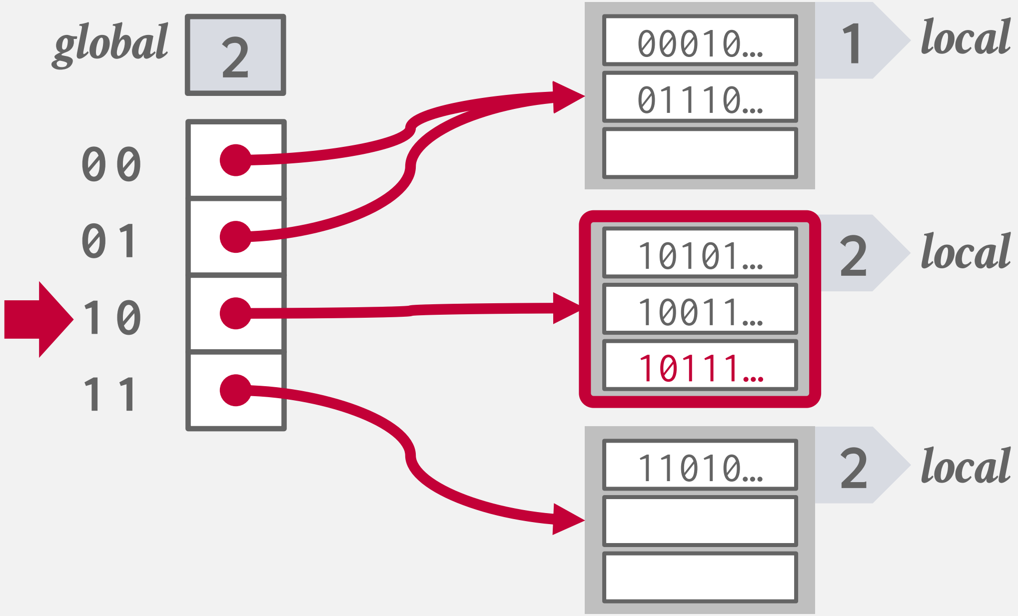


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

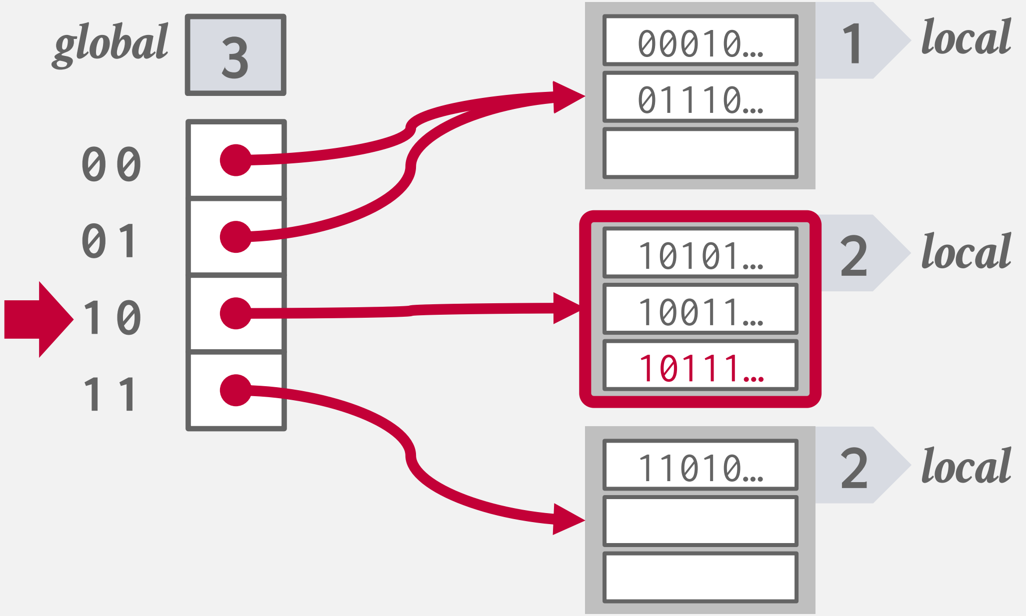


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

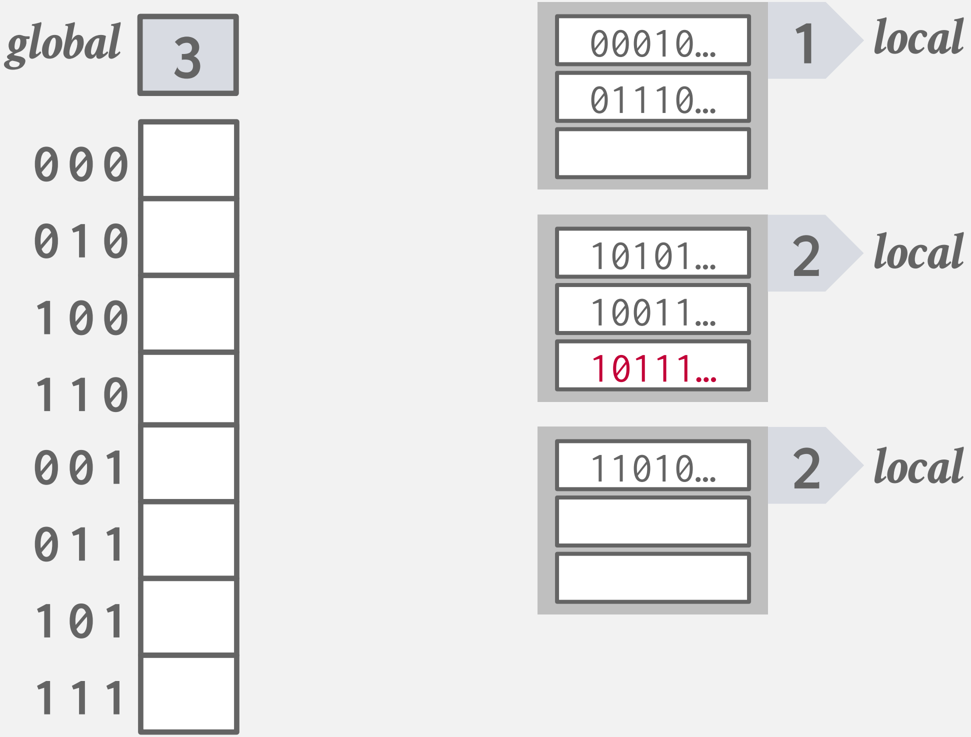


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

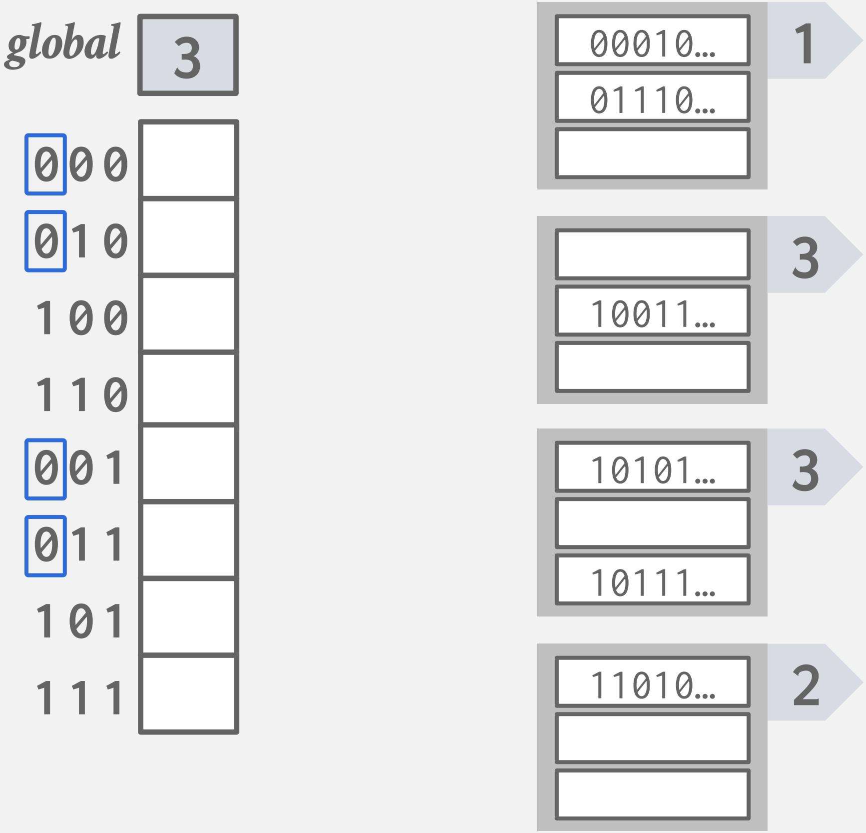


Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

EXTENDIBLE HASHING

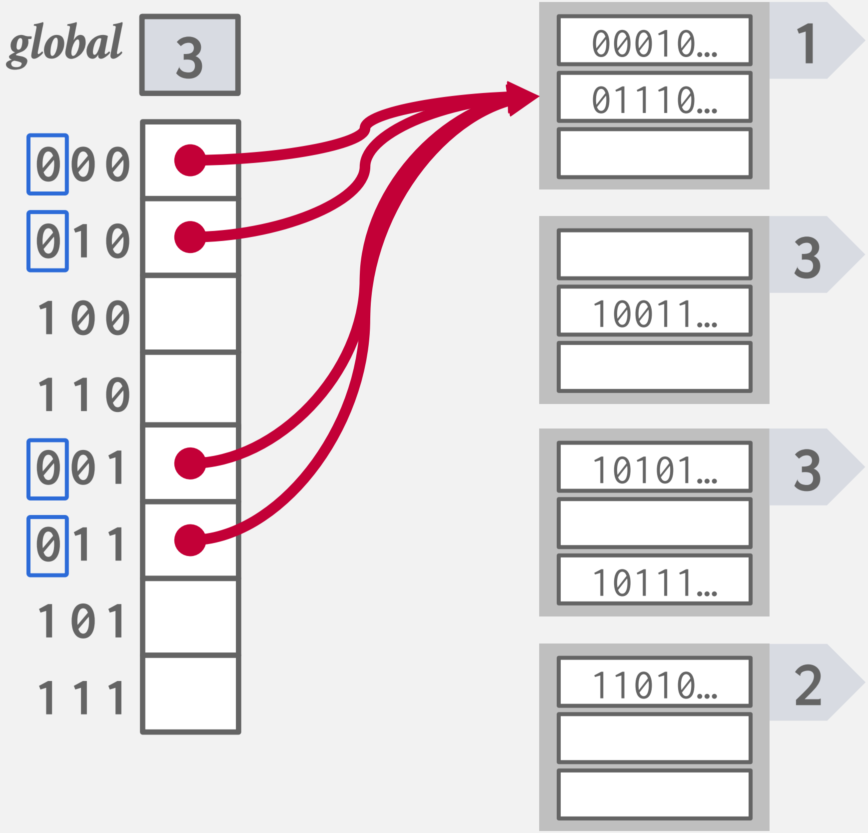


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

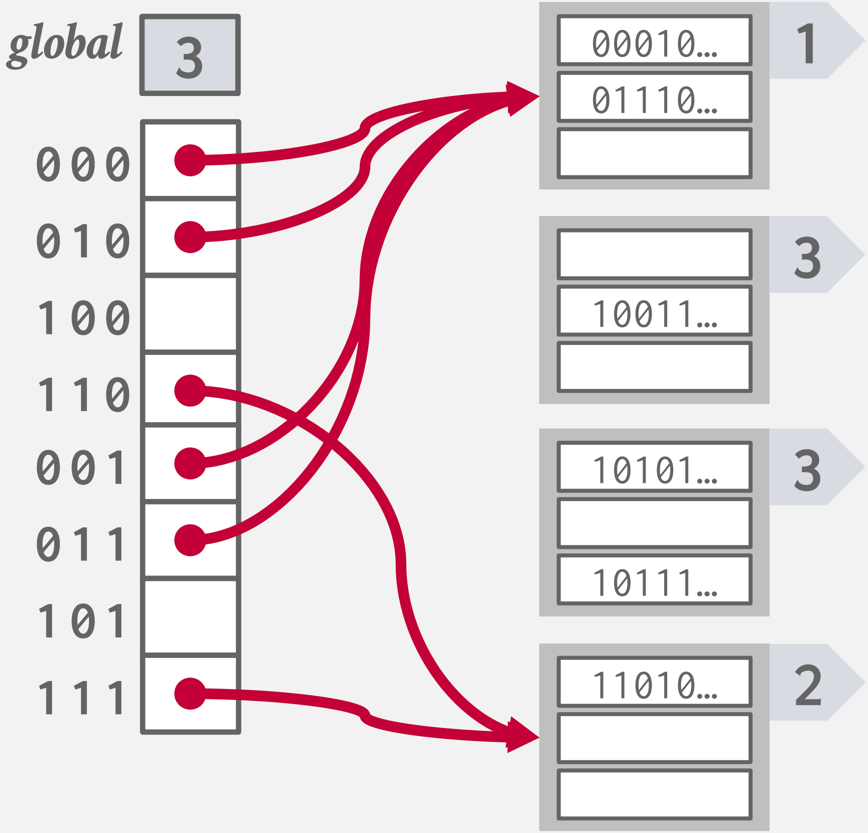


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

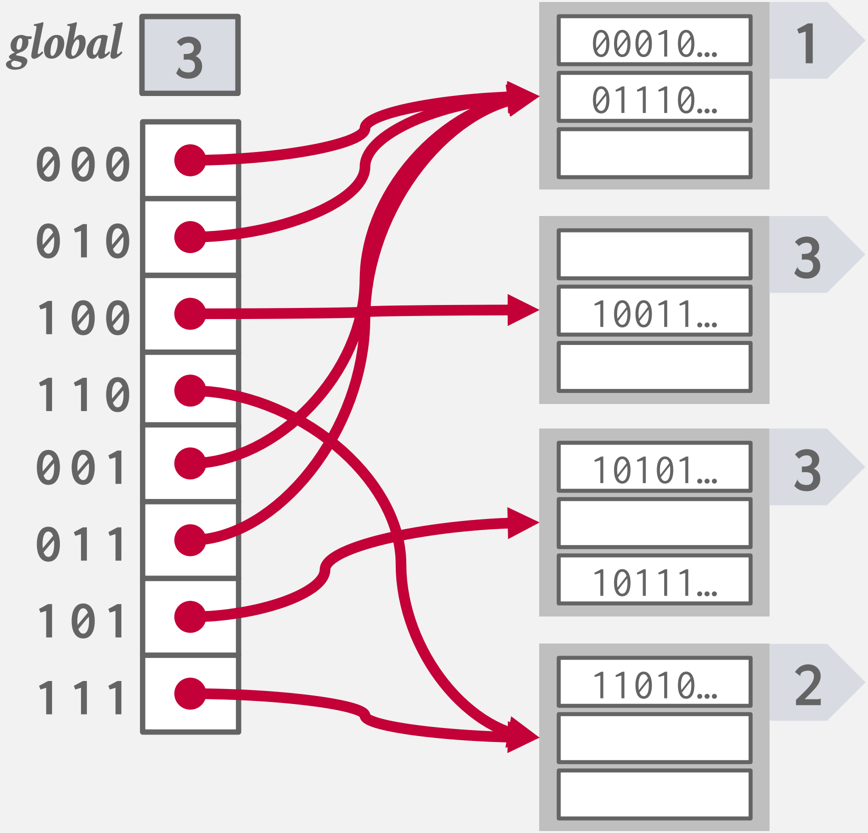


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

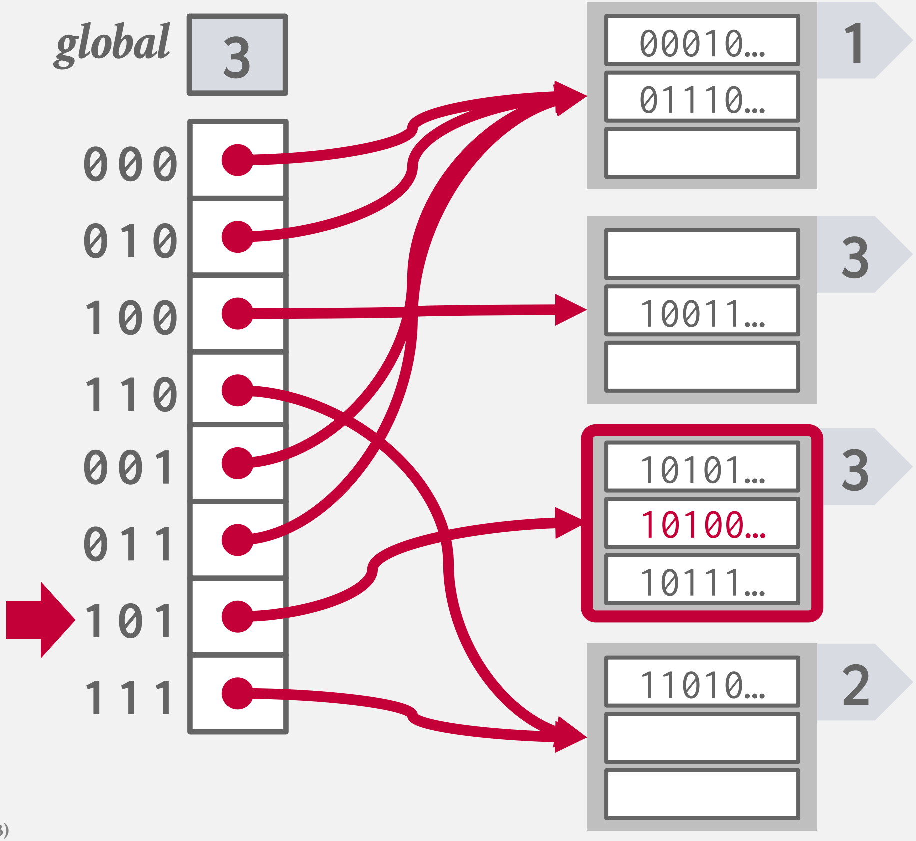


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

LINEAR HASHING

The hash table maintains a <u>pointer</u> that tracks the next bucket to split.

→ When <u>any</u> bucket overflows, split the bucket at the pointer location.

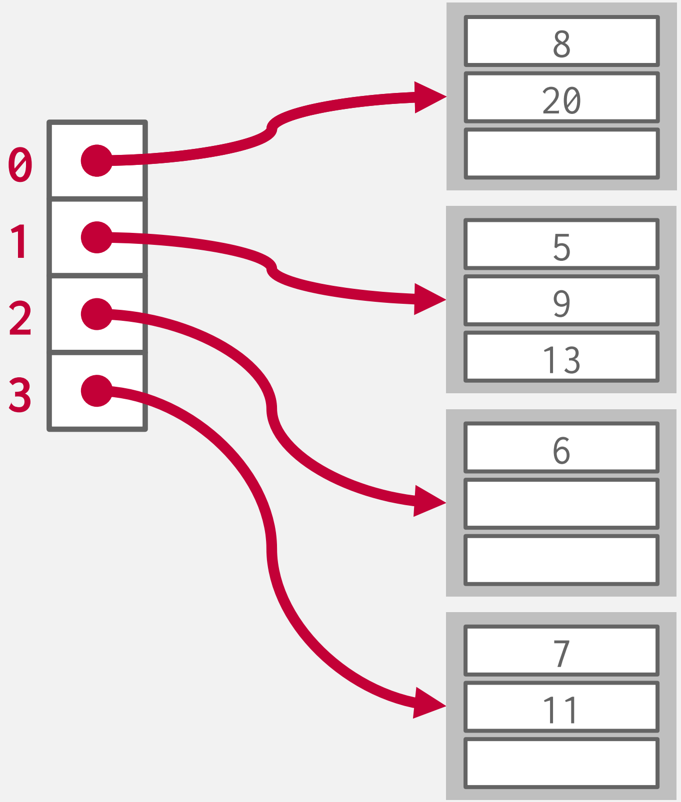
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

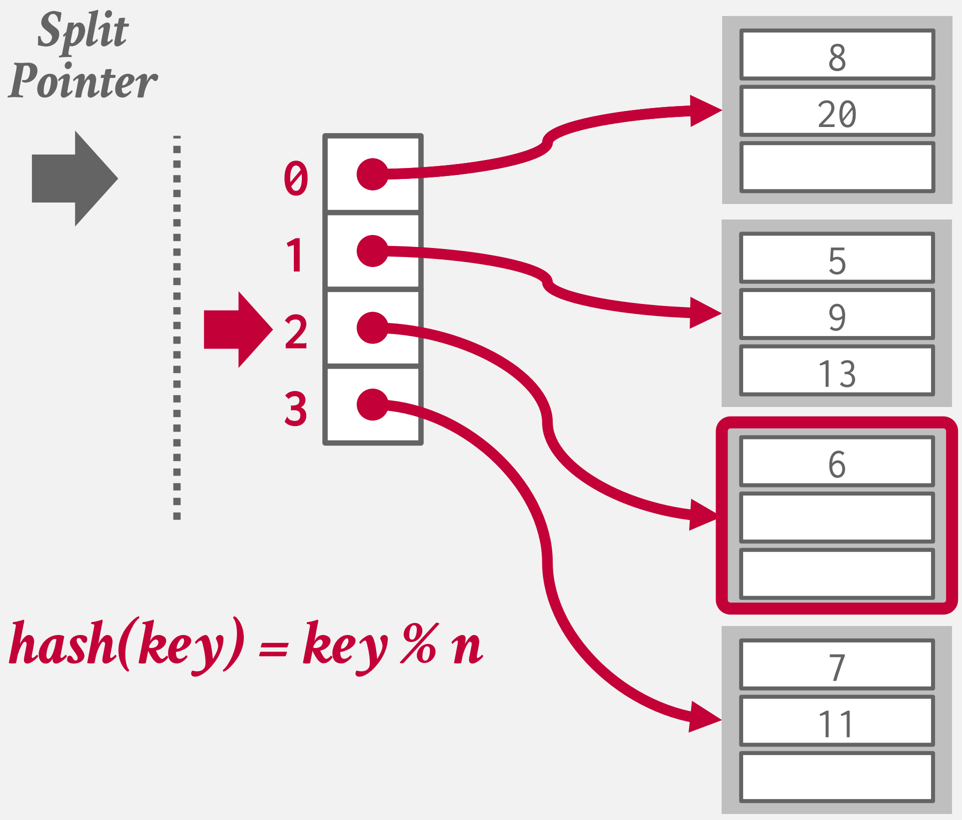
→ Space Utilization

→ Average Length of Overflow Chains

LINEAR HASHING



LINEAR HASHING

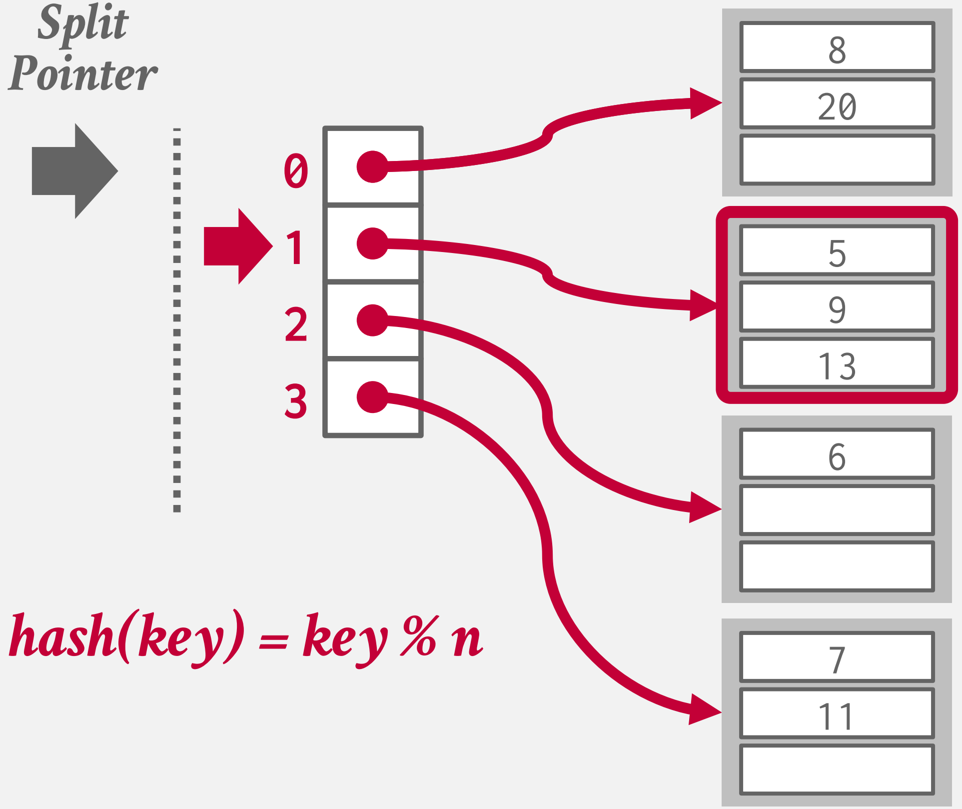


hash(key) = key % n

Get 6

hash(6) = 6 % 4 = 2

LINEAR HASHING



hash(key) = key % n

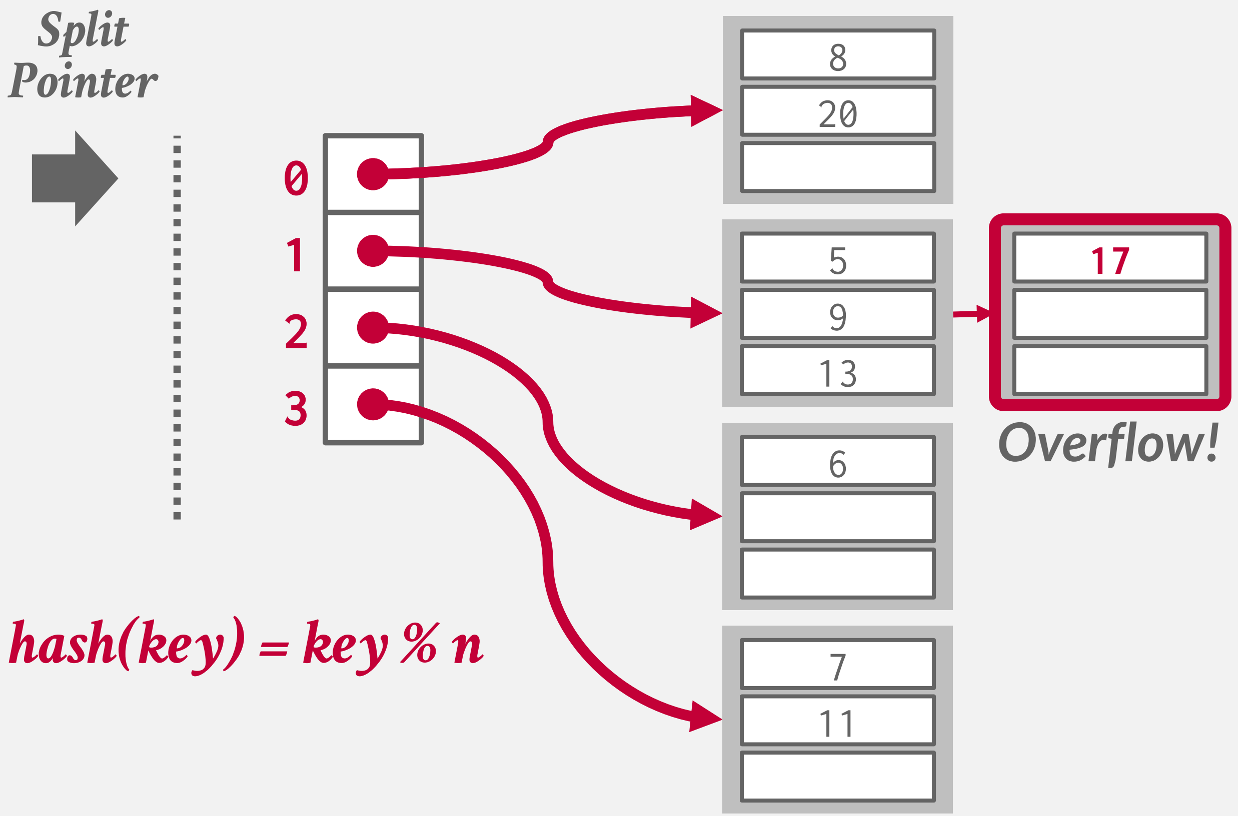
Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

LINEAR HASHING



hash(key) = key % n

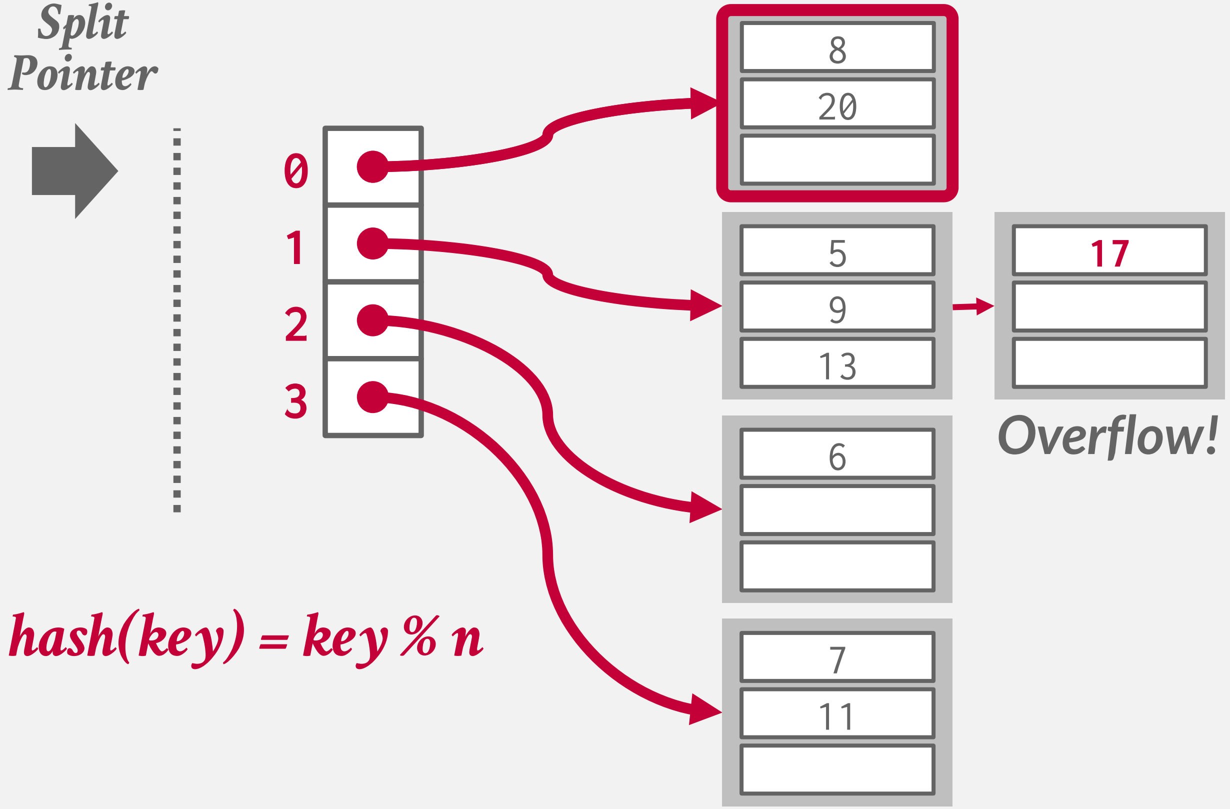
Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

LINEAR HASHING

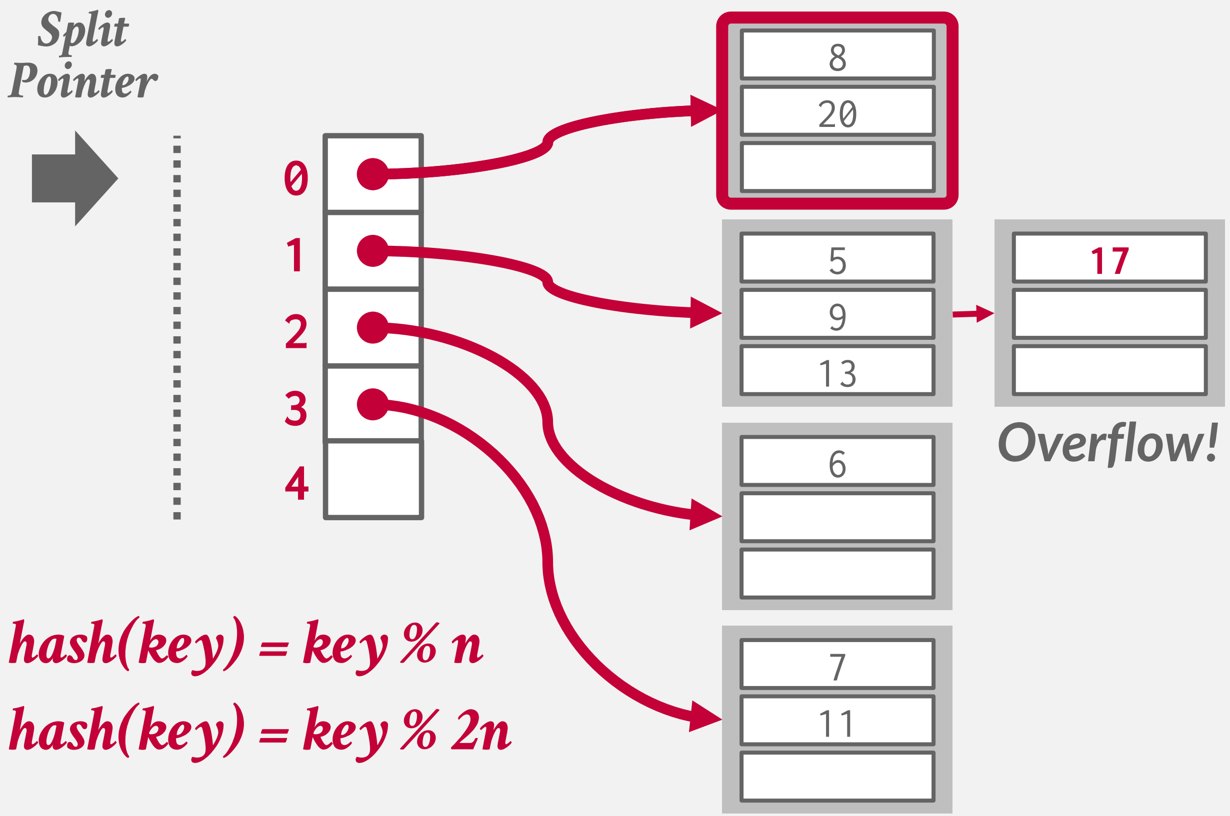


Get 6
hash(6) = 6 % 4 = 2

Put 17
hash(17) = 17 % 4 = 1

hash(key) = key % n

LINEAR HASHING



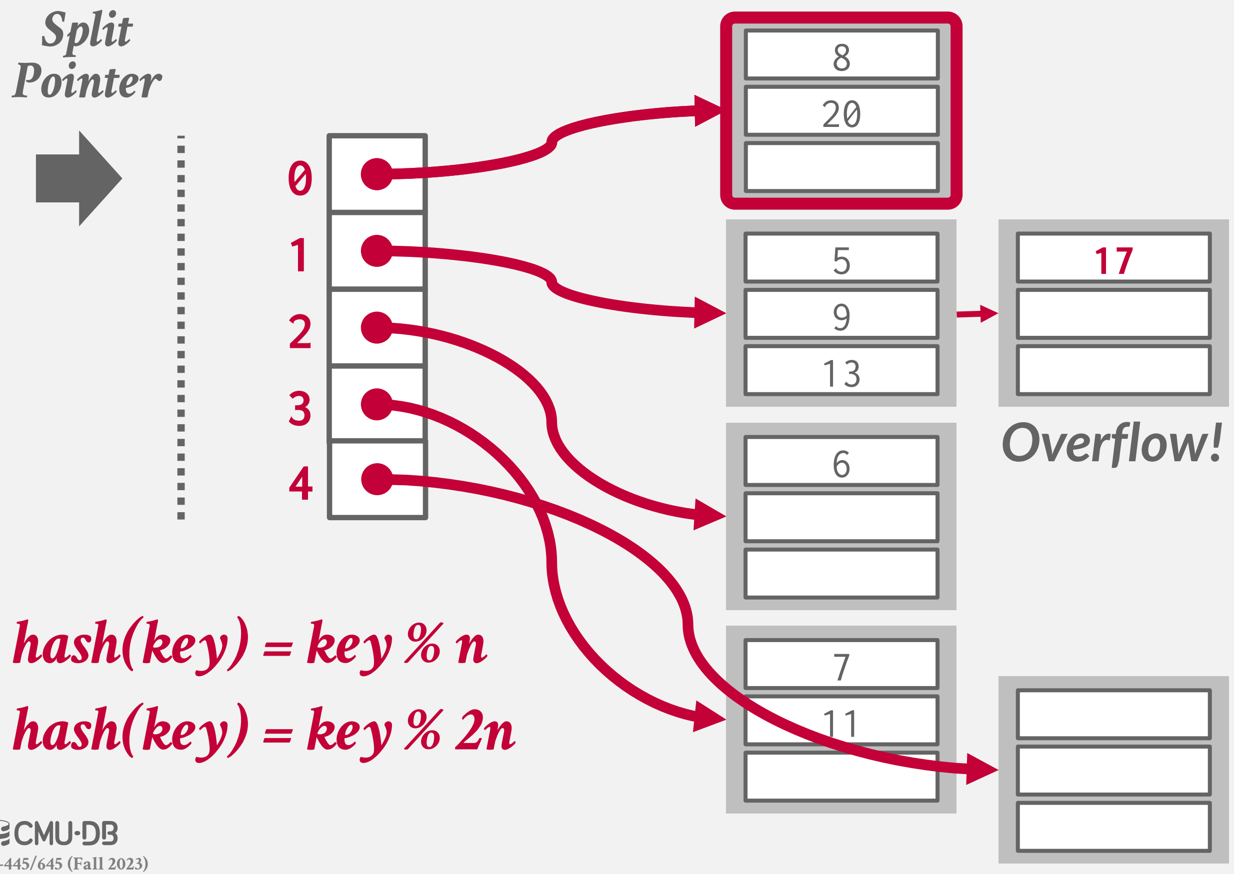
Get 6
hash(6) = 6 % 4 = 2

Put 17
hash(17) = 17 % 4 = 1

hash(key) = key % n

hash(key) = key % 2n

LINEAR HASHING

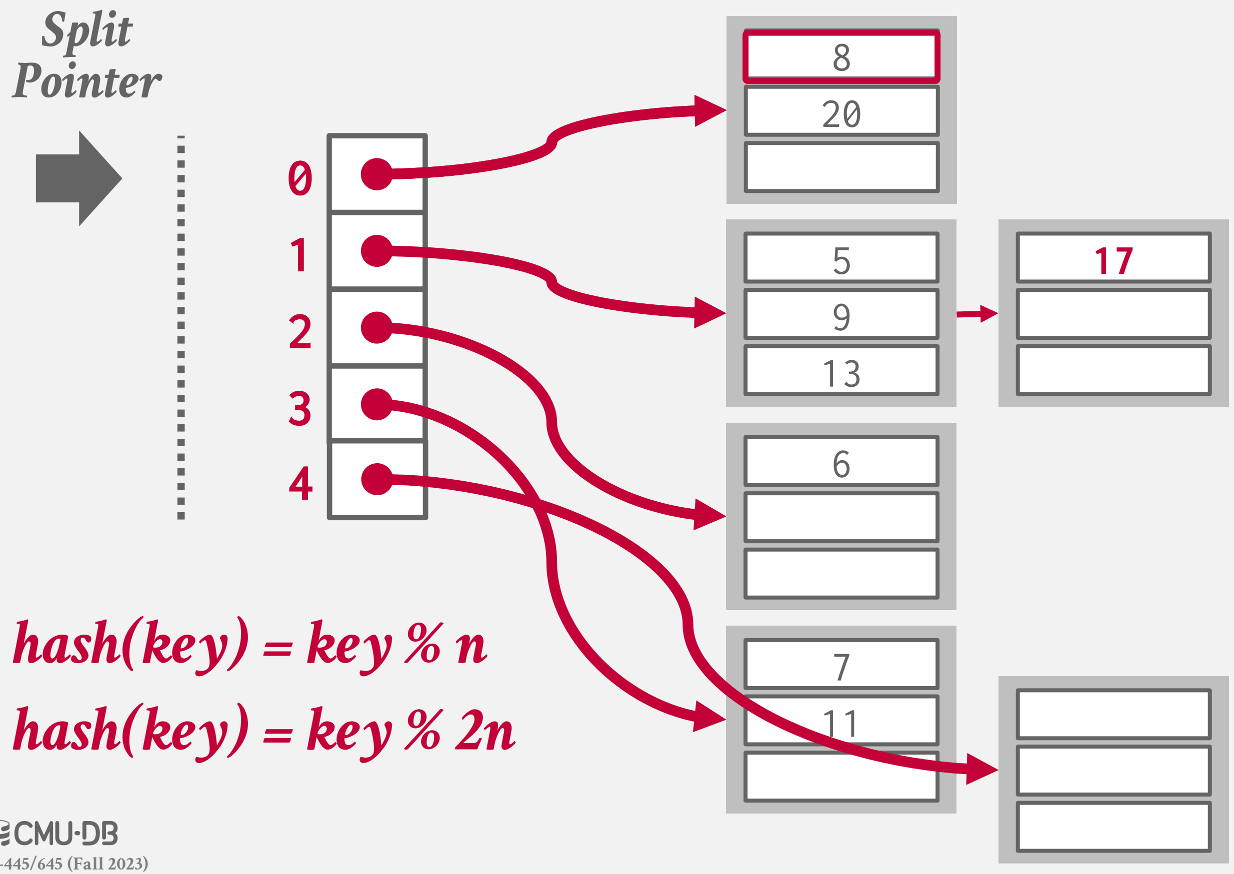


Get 6
hash(6) = 6 % 4 = 2

Put 17
hash(17) = 17 % 4 = 1

hash(key) = key % n
hash(key) = key % 2n

LINEAR HASHING

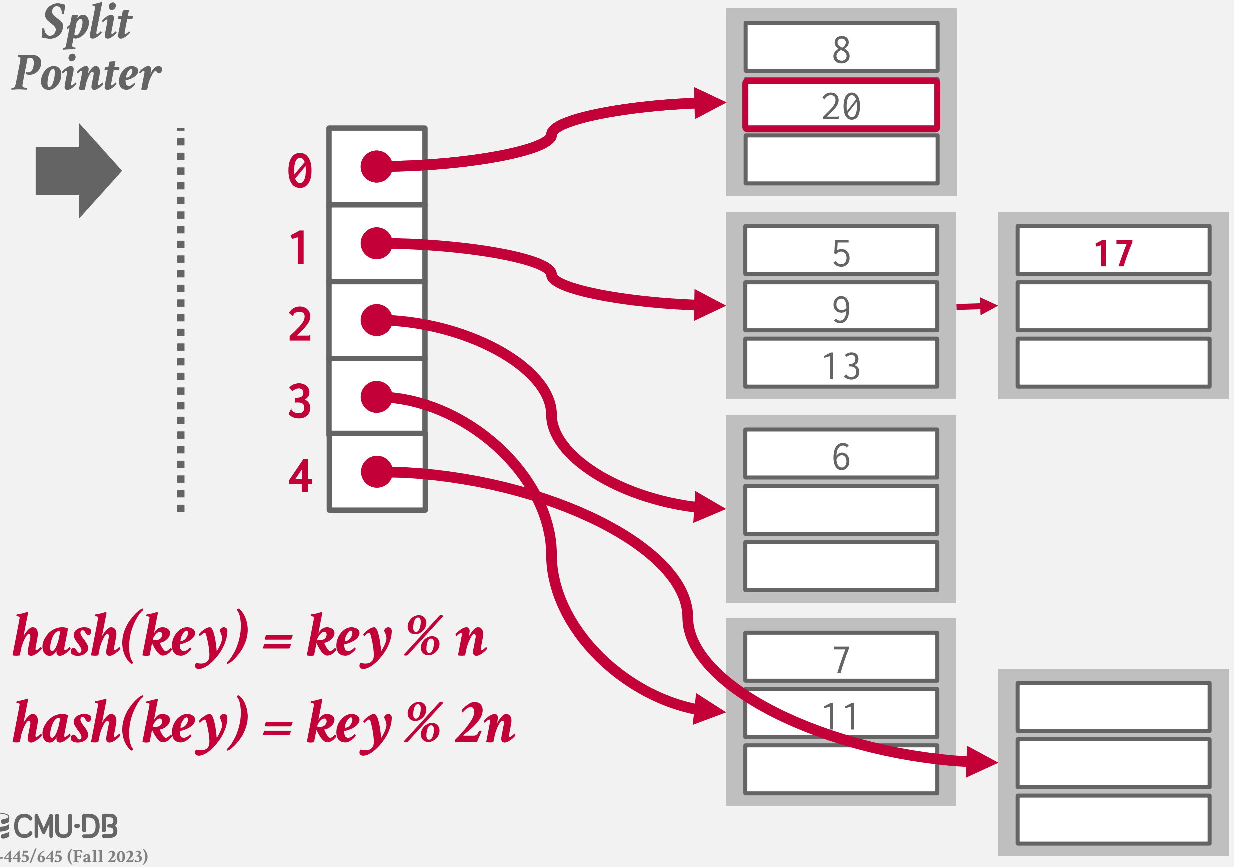


Get 6
hash(6) = 6 % 4 = 2

Put 17
hash(17) = 17 % 4 = 1
hash(8) = 8 % 8 = 0

hash(key) = key % n
hash(key) = key % 2n

LINEAR HASHING



Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

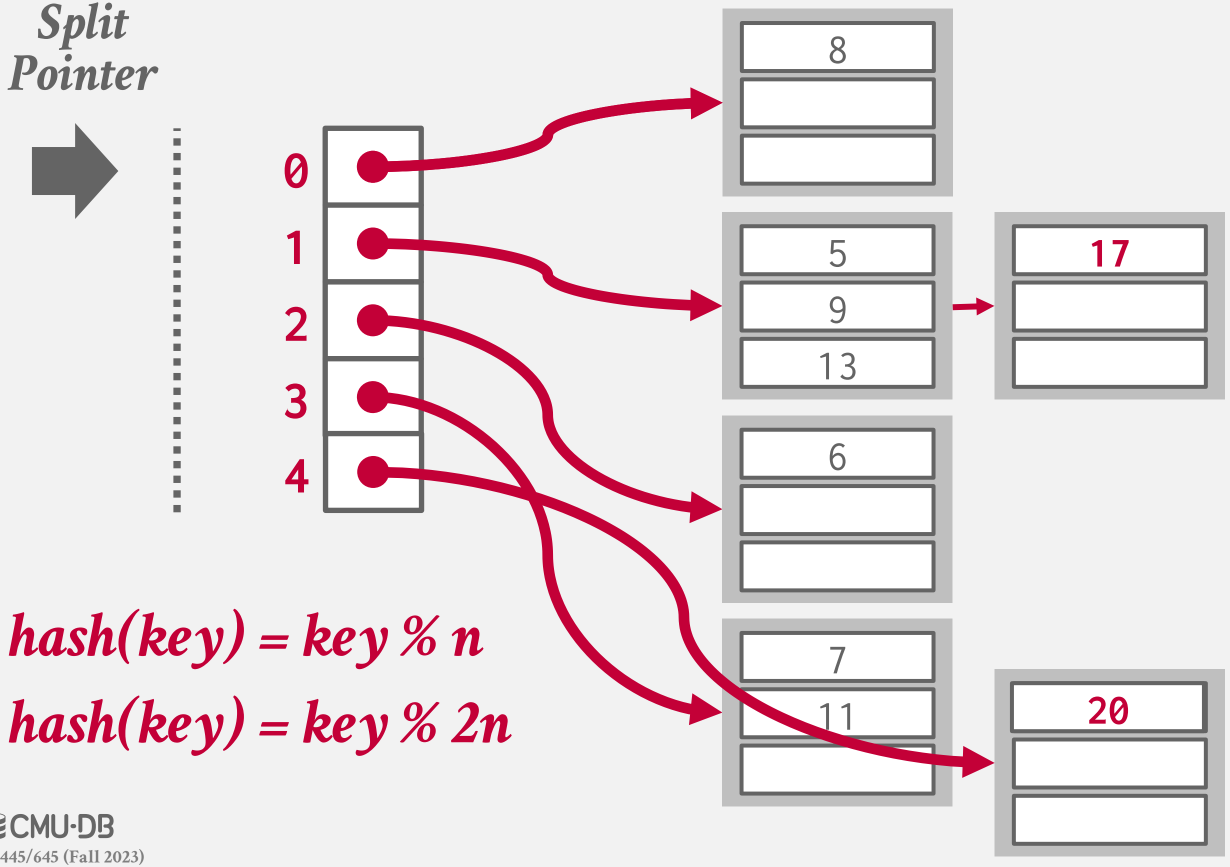
hash(8) = 8 % 8 = 0

hash(20) = 20 % 8 = 4

hash(key) = key % n

hash(key) = key % 2n

LINEAR HASHING

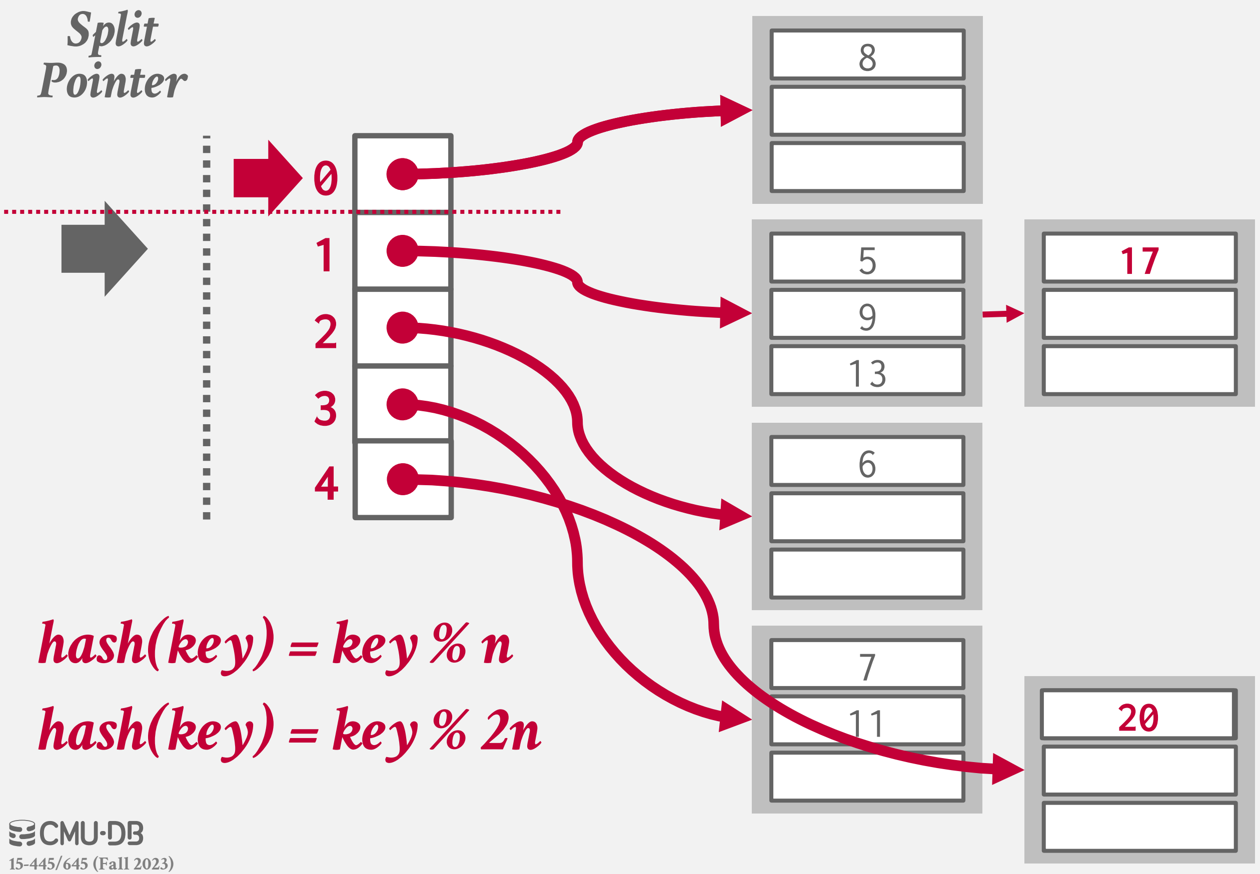


Get 6
hash(6) = 6 % 4 = 2

Put 17
hash(17) = 17 % 4 = 1
hash(8) = 8 % 8 = 0
hash(20) = 20 % 8 = 4

hash(key) = key % n
hash(key) = key % 2n

LINEAR HASHING



hash(key) = key % n

hash(key) = key % 2n

Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

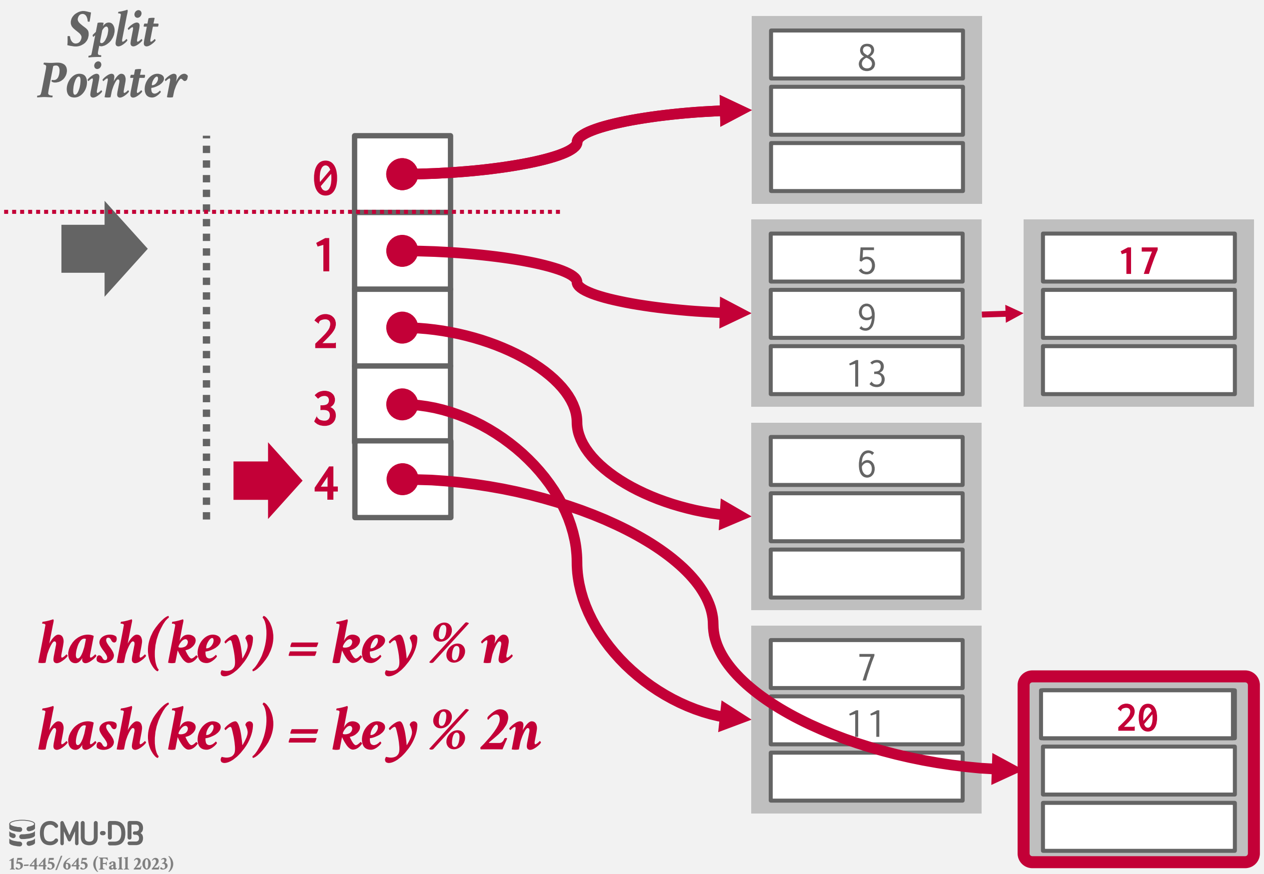
hash(8) = 8 % 8 = 0

hash(20) = 20 % 8 = 4

Get 20

hash(20) = 20 % 4 = 0

LINEAR HASHING



$hash(key) = key \% n$

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

Get 6

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

Put 17

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

$hash(8) = 8 \% 8 = 0$

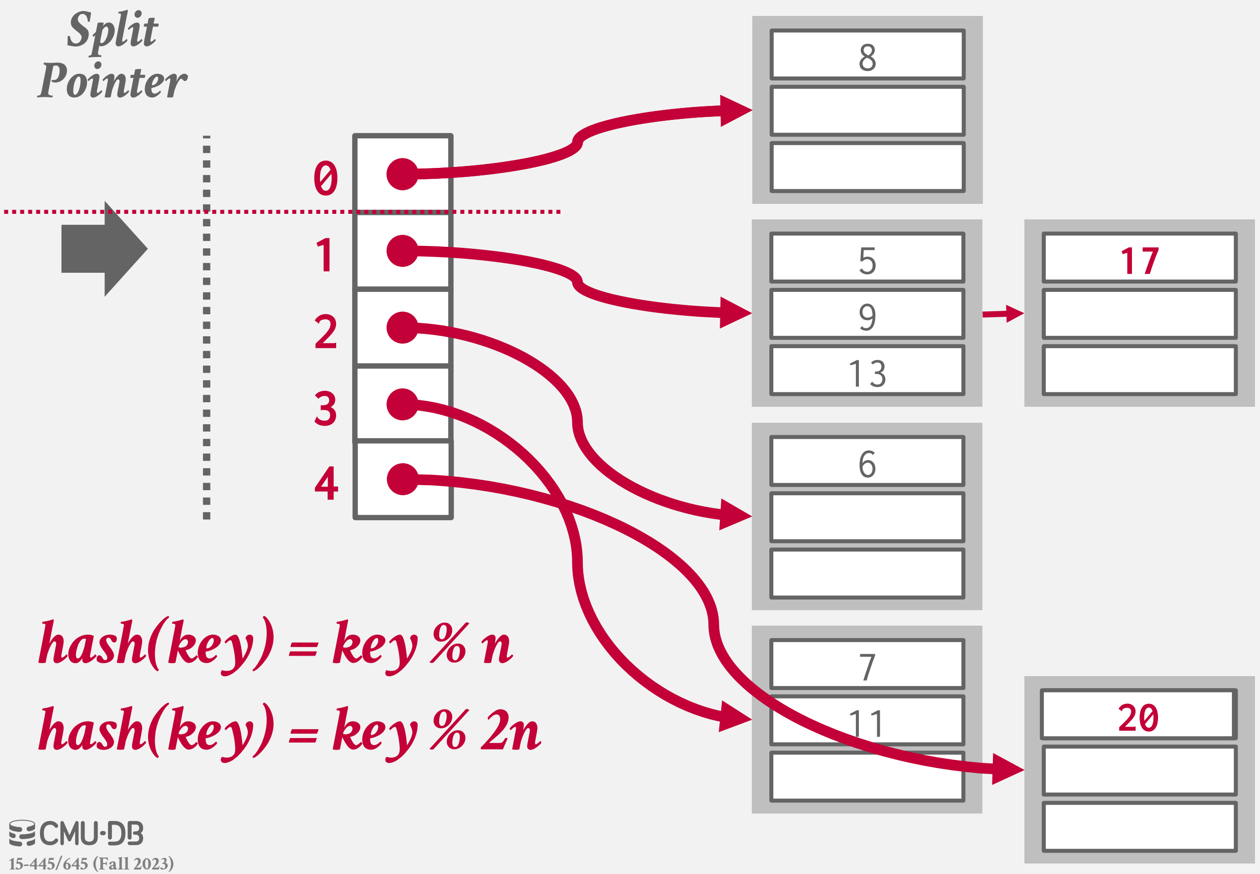
$hash(20) = 20 \% 8 = 4$

Get 20

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

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

LINEAR HASHING



Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

hash(8) = 8 % 8 = 0

hash(20) = 20 % 8 = 4

Get 20

hash(20) = 20 % 4 = 0

hash(20) = 20 % 8 = 4

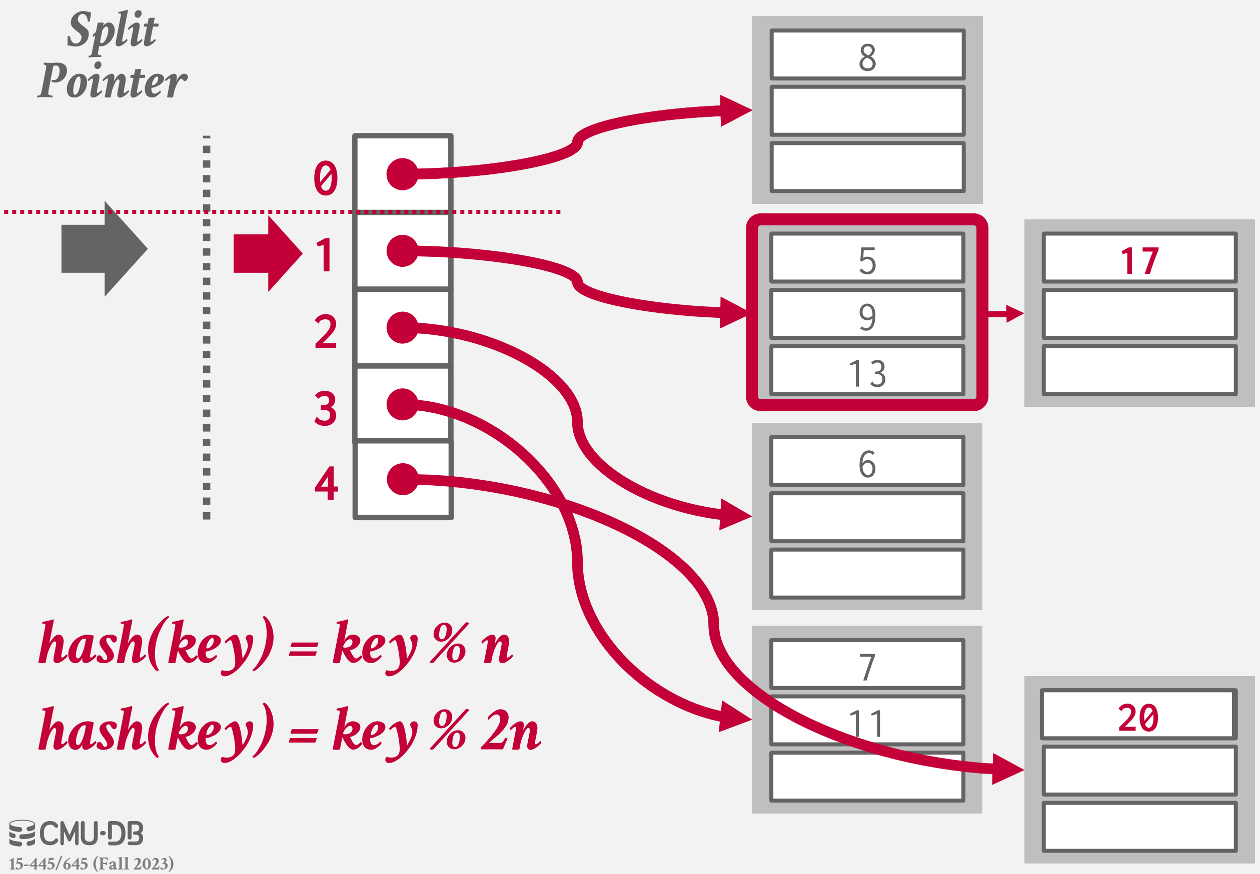
Get 9

hash(9) = 9 % 4 = 1

hash(key) = key % n

hash(key) = key % 2n

LINEAR HASHING



hash(key) = key % n

hash(key) = key % 2n

Get 6

hash(6) = 6 % 4 = 2

Put 17

hash(17) = 17 % 4 = 1

hash(8) = 8 % 8 = 0

hash(20) = 20 % 8 = 4

Get 20

hash(20) = 20 % 4 = 0

hash(20) = 20 % 8 = 4

Get 9

hash(9) = 9 % 4 = 1

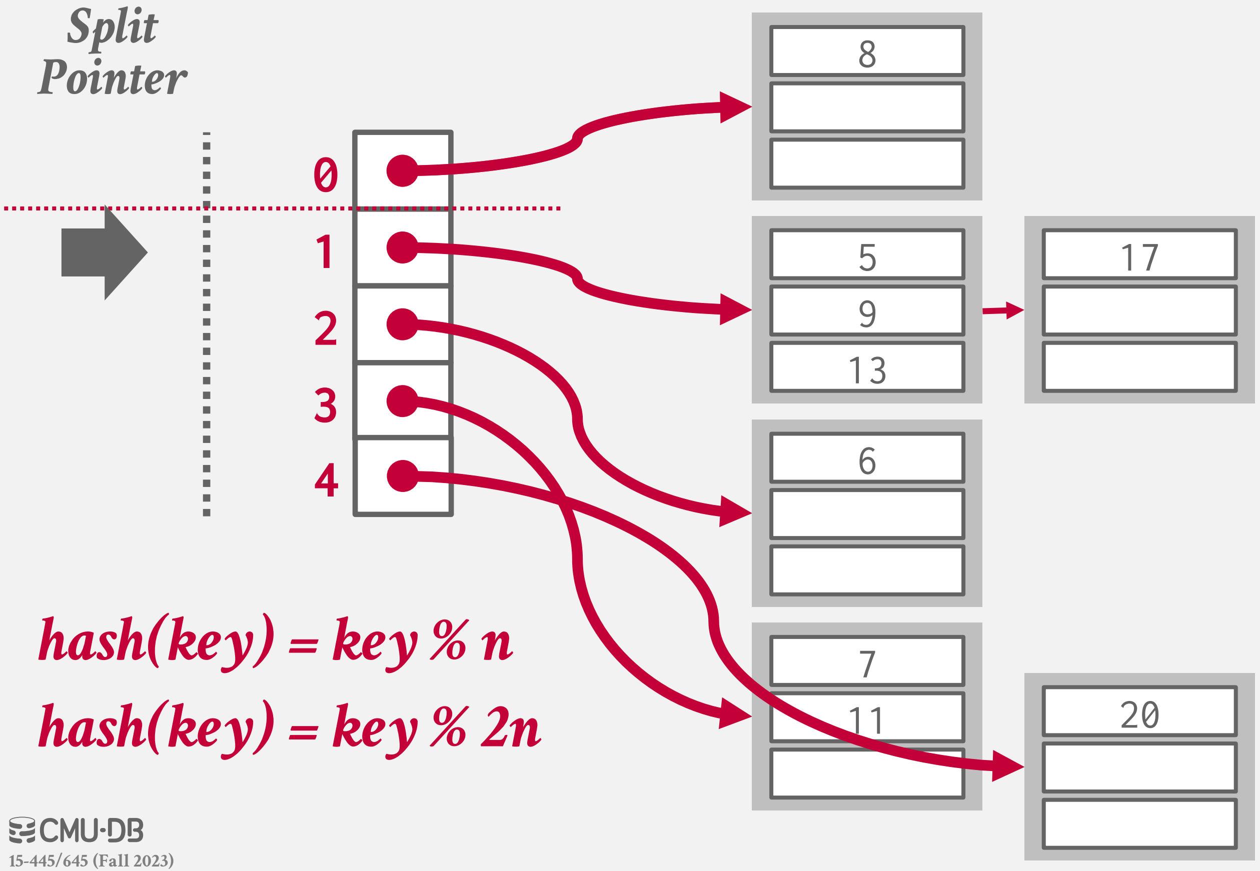
LINEAR HASHING – RESIZING

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

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

If the "highest" bucket below the split pointer is empty, the hash table could remove it and move the splinter pointer in reverse direction.

LINEAR HASHING – DELETES



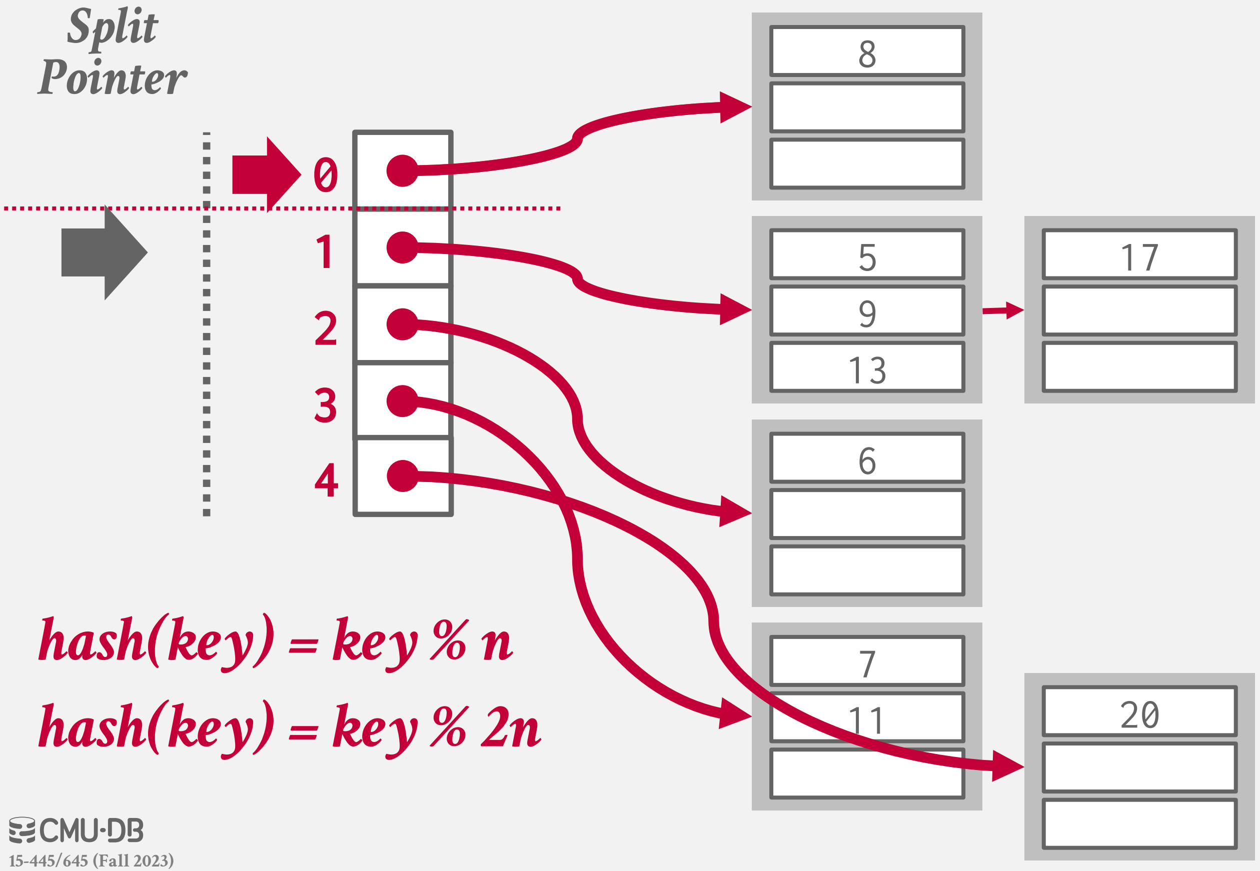
Delete 20

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

$hash(key) = key \% n$

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

LINEAR HASHING – DELETES



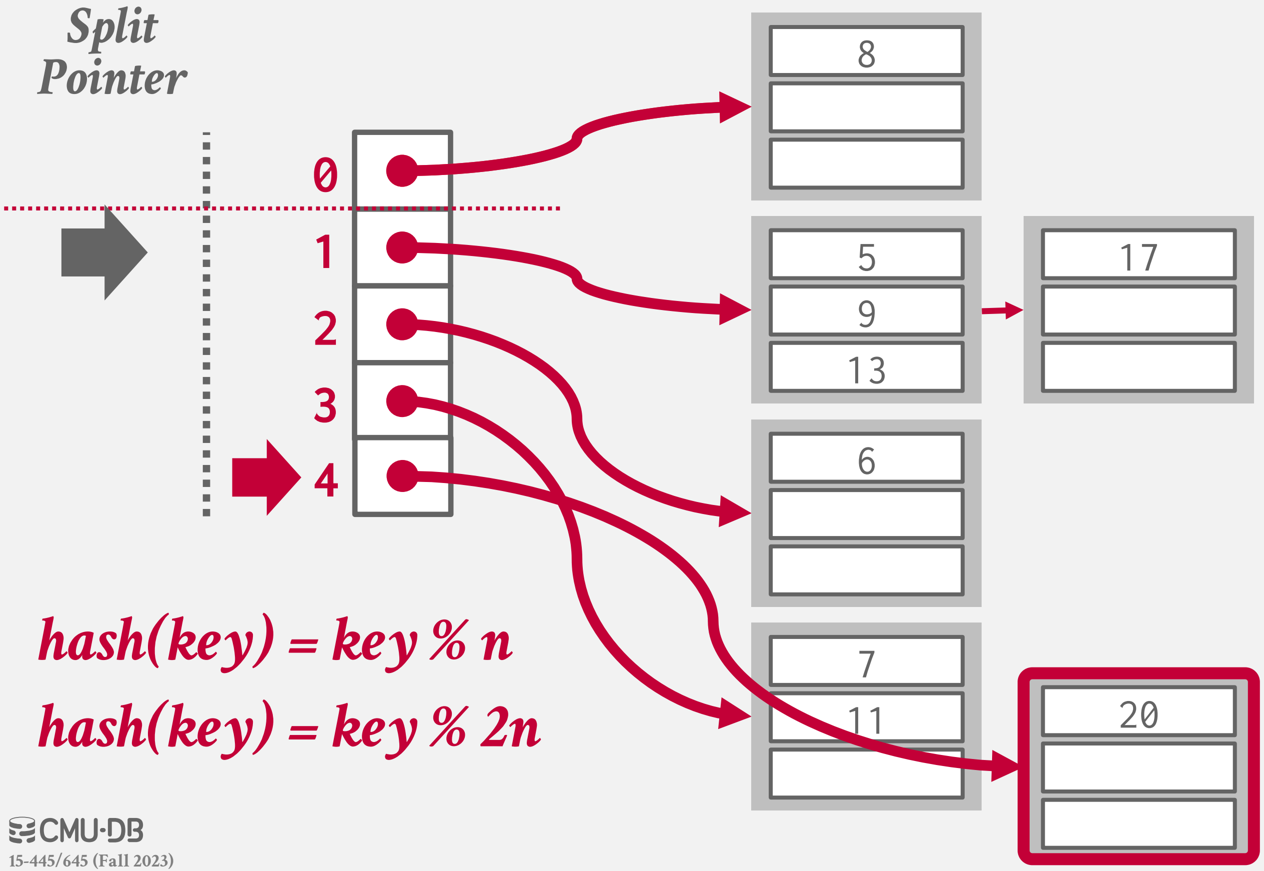
Delete 20

hash(20) = 20 % 4 = 0

hash(key) = key % n

hash(key) = key % 2n

LINEAR HASHING – DELETES



Delete 20

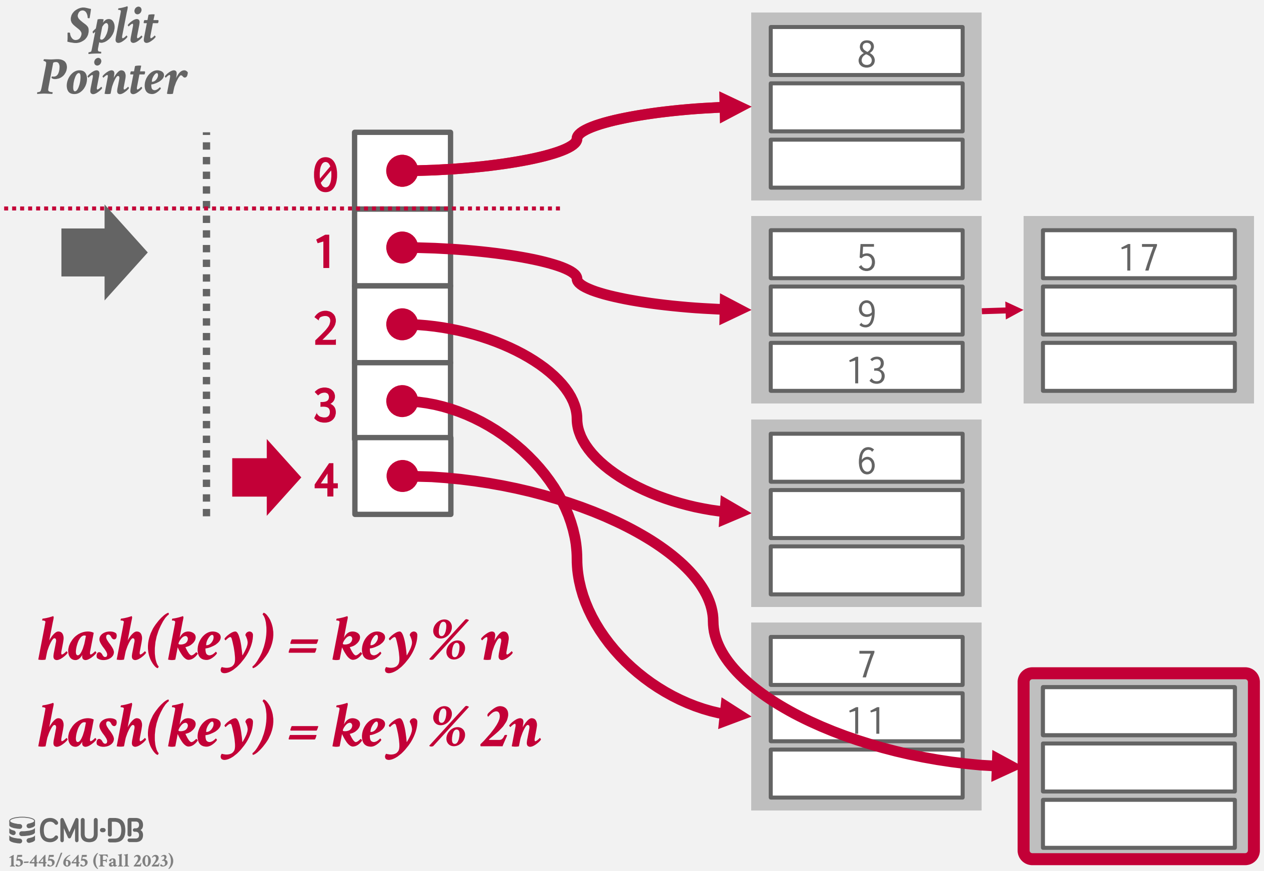
$hash(20) = 20 \% 4 = 0$

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

$hash(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

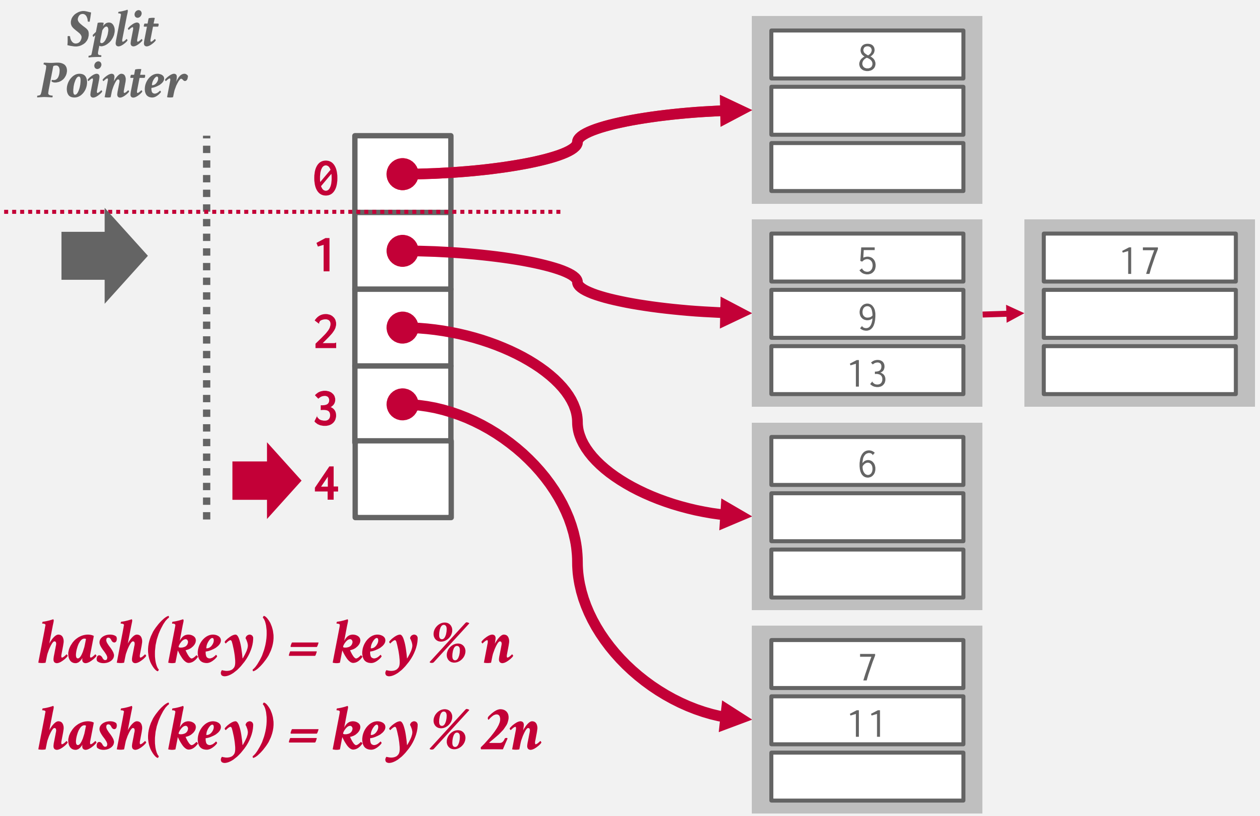
hash(20) = 20 % 4 = 0

hash(20) = 20 % 8 = 4

hash(key) = key % n

hash(key) = key % 2n

LINEAR HASHING – DELETES



Delete 20

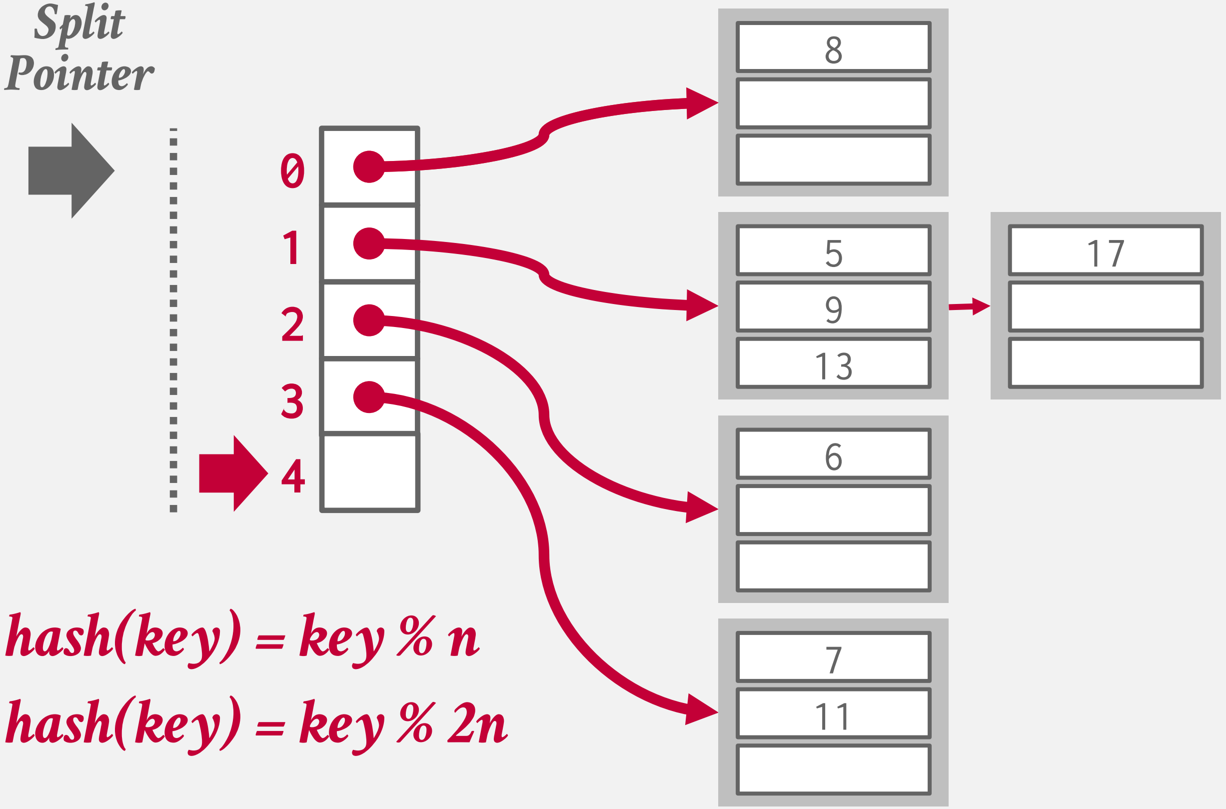
$hash(20) = 20 \% 4 = 0$

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

$hash(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

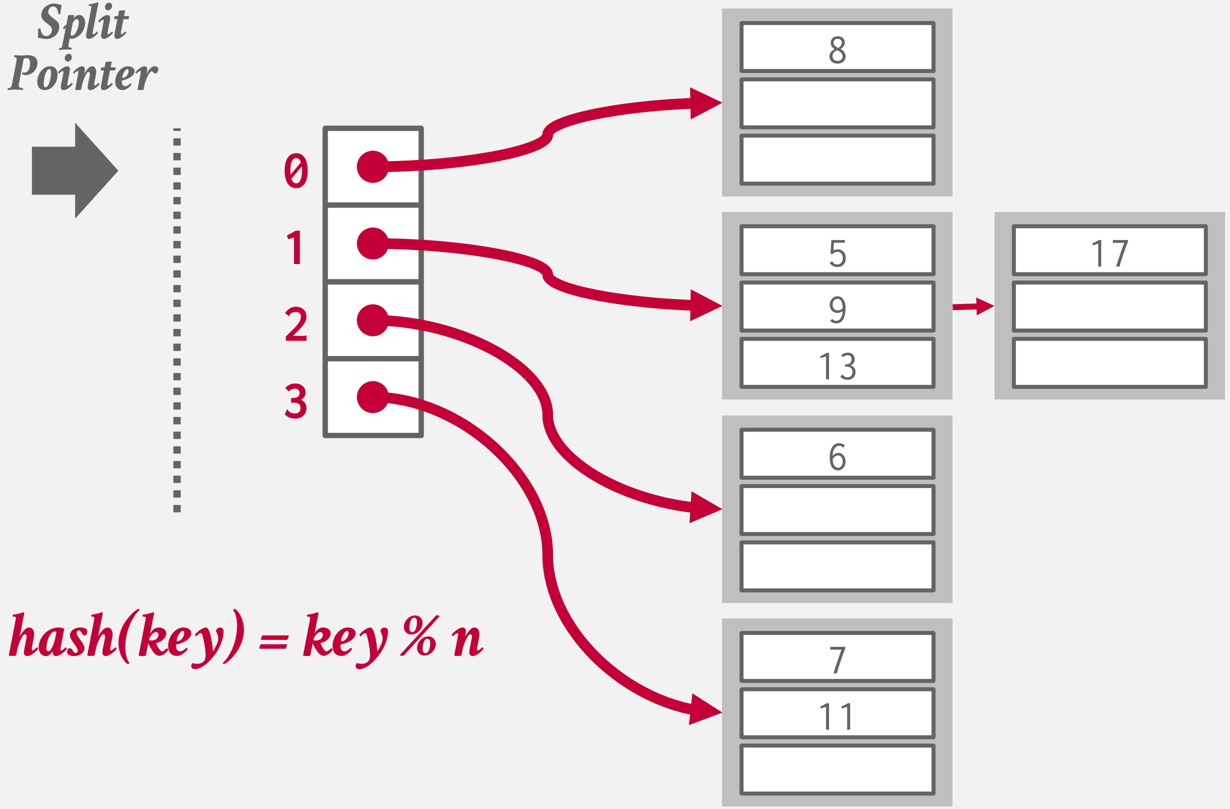
$hash(20) = 20 \% 4 = 0$

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

$hash(key) = key \% n$

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

LINEAR HASHING – DELETES



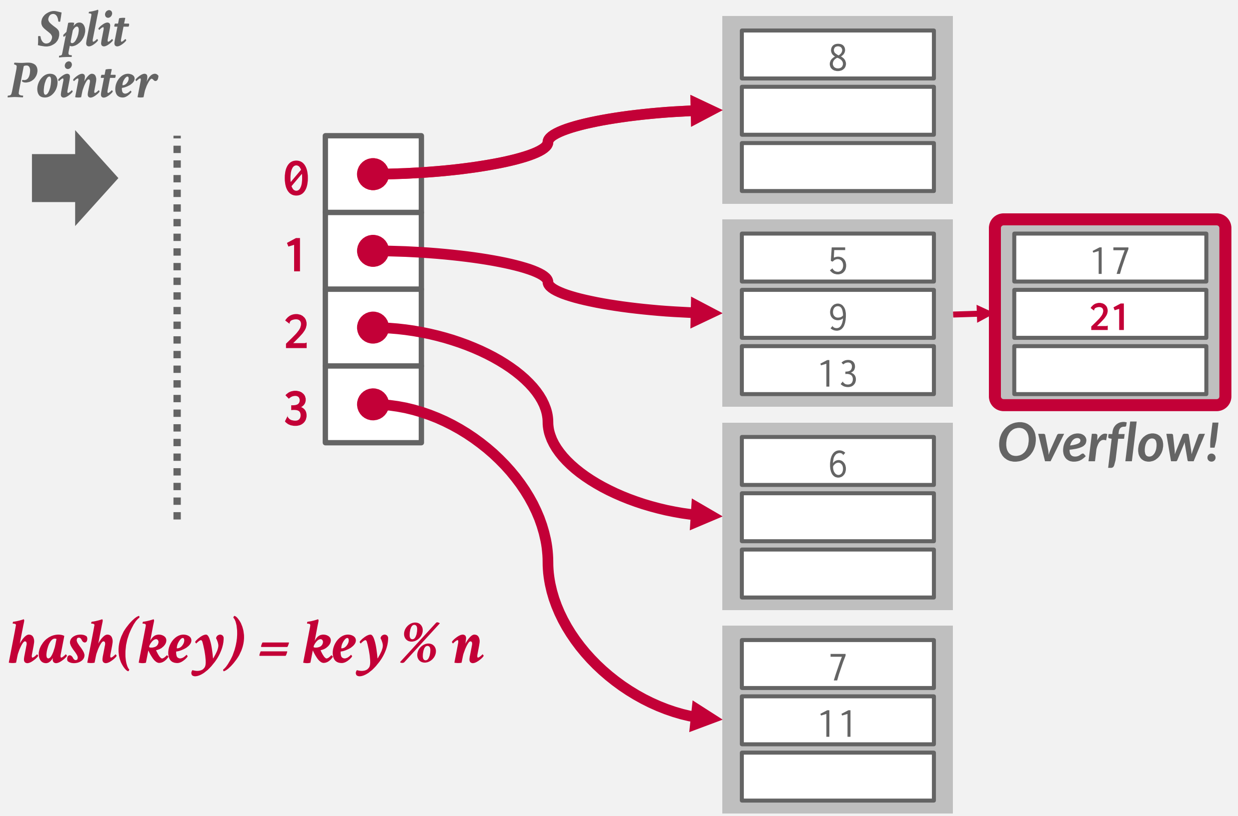
hash(key) = key % n

Delete 20

hash(20) = 20 % 4 = 0

hash(20) = 20 % 8 = 4

LINEAR HASHING – DELETES



Delete 20
hash(20) = 20 % 4 = 0
hash(20) = 20 % 8 = 4

Put 21
hash(21) = 21 % 4 = 1

hash(key) = key % n

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"