

Carnegie Mellon University

# Database Systems

## Hash Tables



15-445/645 FALL 2024 » PROF. ANDY PAVLO

# ADMINISTRIVIA

---

**Homework #2** is due Sept 22<sup>nd</sup> @ 11:59pm

**Project #1** is due Sept 29<sup>th</sup> @ 11:59pm

→ Recitation on Thursday Sept 19<sup>th</sup> @ 6:00pm (See [@144](#))

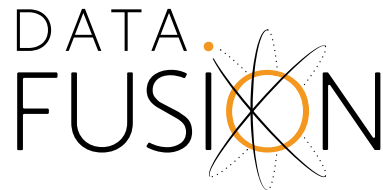
# UPCOMING DATABASE TALKS

---

## DataFusion (DB Seminar)

→ Monday Sept 23<sup>rd</sup> @ 4:30pm ET

→ Zoom



## DataFusion Comet (DB Seminar)

→ Monday Sept 30<sup>th</sup> @ 4:30pm ET

→ Zoom



## Oracle Talk (DB Group)

→ Tuesday Oct 1<sup>st</sup> @ 12:00pm ET

→ GHC 6501



# UPCOMING D

## DataFusion (DB Seminar)

→ Monday Sept 23<sup>rd</sup> @ 4:30pm ET

→ Zoom

## DataFusion Comet (DB Sem

→ Monday Sept 30<sup>th</sup> @ 4:30pm ET

→ Zoom

## Oracle Talk (DB Group)

→ Tuesday Oct 1<sup>st</sup> @ 12:00pm ET

→ GHC 6501

# Larry Ellison becomes world's second-richest man, dethroning Jeff Bezos as Oracle stock surges

PUBLISHED MON, SEP 16 2024 1:40 PM EDT UPDATED MON, SEP 16 2024 4:06 PM EDT



Annie Palmer  
@IN/ANNIERPALMER/

WATCH LIVE

## KEY POINTS

- Oracle Chairman Larry Ellison is now the second-richest person in the world, with a net worth of \$206 billion, unseating Amazon founder Jeff Bezos who had held the title on and off since 2016.
- Shares of Oracle have surged 20% in September, putting them on track for their best month since October 2022.
- Oracle's stock success is partly due to the company's role in the artificial intelligence boom.

# COURSE OUTLINE

---

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)

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

# TODAY'S AGENDA

---

Background

Hash Functions

Static Hashing Schemes 固定数量的键值对元素

Dynamic Hashing Schemes

DB Flash Talk: RelationalAI

# DATA STRUCTURES

---

Internal Meta-data

Core Data Storage

Temporary Data Structures JOIN 算子生成的哈希表

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:  $O(n)$

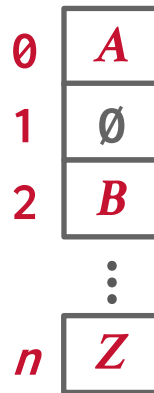
# STATIC HASH TABLE

---

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.

$$\text{hash}(\text{key}) \% N$$

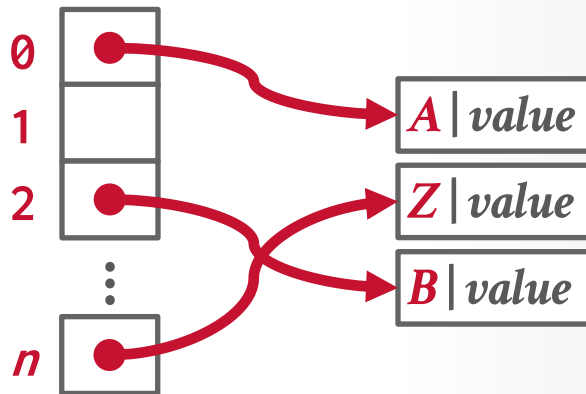


# STATIC HASH TABLE

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.

$$\text{hash}(\text{key}) \% N$$



# UNREALISTIC ASSUMPTIONS

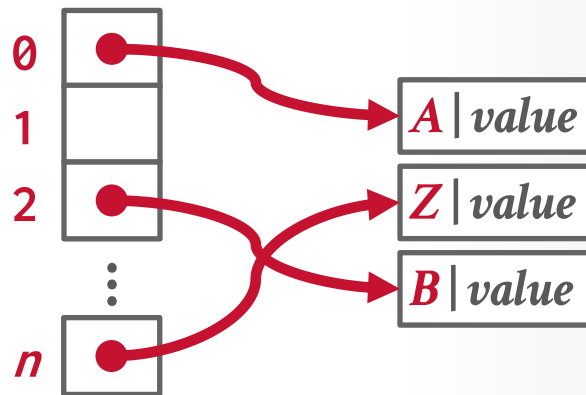
**Assumption #1:** Number of elements is known ahead of time and fixed.

**Assumption #2:** Each key is unique.

**Assumption #3:** Perfect hash function guarantees no collisions.

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

*hash(key) % N*



# 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.

# HASH FUNCTIONS

---

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

→ Converts arbitrary byte array into a fixed-length code.

We want something that is fast and has a low collision rate.

速率与碰撞率之间的权衡.

We do not want to use a cryptographic hash function for DBMS hash tables (e.g., SHA-2).

# 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 FUNCTIONS

## smhasher

### SMhasher

Linux Build status  build passing  build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
<a href="#">donothing32</a>	11149460.06	4.00	-	13	bad seed 0, test NOP
<a href="#">donothing64</a>	11787676.42	4.00	-	13	bad seed 0, test NOP
<a href="#">donothing128</a>	11745060.76	4.06	-	13	bad seed 0, test NOP
<a href="#">NOP_OAAT_read64</a>	11372846.37	14.00	-	47	test NOP
<a href="#">BadHash</a>	769.94	73.97	-	47	bad seed 0, test FAIL
<a href="#">sumhash</a>	10699.57	29.53	-	363	bad seed 0, test FAIL
<a href="#">sumhash32</a>	42877.79	23.12	-	863	UB, test FAIL
<a href="#">multiply_shift</a>	8026.77	26.05	226.80 (8)	345	bad seeds & 0xffffffff0, fails most tests
<a href="#">pair_multiply_shift</a>	3716.95	40.22	186.34 (3)	609	fails most tests
<a href="#">crc32</a>	383.12	134.21	257.50 (11)	422	insecure, 8590x collisions, distrib, PerlinNoise
<a href="#">md5_32</a>	350.53	644.31	894.12 (10)	4419	

on.

State-of-the-art

rates.



# HASH FUNCTIONS

## smhasher

### SMhasher

Linux Build status  

Hash function	MiB/sec	cycl./hash	cycl./map	size
<a href="#">donothing32</a>	11149460.06	4.00	-	13
<a href="#">donothing64</a>	11787676.42	4.00	-	13
<a href="#">donothing128</a>	11745060.76	4.06	-	13
<a href="#">NOP_OAAT_read64</a>	11372846.37	14.00	-	4
<a href="#">BadHash</a>	769.94	73.97	-	4
<a href="#">sumhash</a>	10699.57	29.53	-	36
<a href="#">sumhash32</a>	42877.79	23.12	-	8
<a href="#">multiply_shift</a>	8026.77	26.05	226.80 (8)	3
<a href="#">pair_multiply_shift</a>	3716.95	40.22	186.34 (3)	6
<a href="#">crc32</a>	383.12	134.21	257.50 (11)	2
<a href="#">md5_32</a>	350.53	644.31	894.12 (10)	4

### 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/fwojck/smhasher3>.

So the fastest hash functions on x86\_64 without quality problems are:

- rapidhash (an improved wyhash)
- xxh3low
- wyhash
- umash (even universal!)
- 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**

← **Open Addressing**

相同 key 每次计算出的位置是随机的，  
不一定是相同的。

There are several other schemes covered in the

Advanced DB course:

- Robin Hood Hashing
- Hopscotch Hashing
- Swiss Tables

当哈希冲突发生时，如何处理：移动想要空间里的元素还是寻找其他位置？

# LINEAR PROBE HASHING

对于给定 key 经过哈希函数计算后, 对槽数组长度取模获得键值对插入的位置:  $\text{pos} = \text{hash}(\text{key}) \% n$

Single giant table of fixed-length 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 table and scan for it.
- Store keys in table to know when to stop scanning.
- Insertions and deletions are generalizations of lookups.

在 hash join 计算时, 通常会使用线性探测法.

The table's **load factor** determines when it is becoming too full and should be resized. 通常设置为 70% 或 80% 的比例.

- Allocate a new table twice as large and rehash entries.

尽量避免哈希表的动态扩容: 需要对已有元素进行重新哈希计算.

# LINEAR PROBE HASHING

$hash(key) \% N$

A

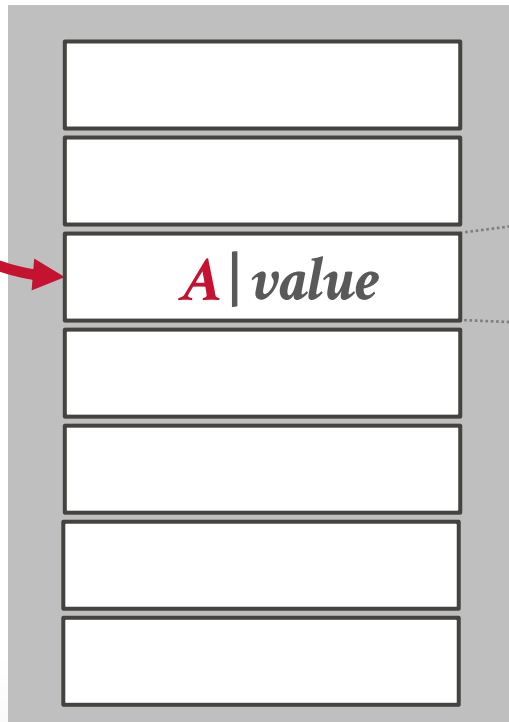
B

C

D

E

F



**<key> | <value>**

# LINEAR PROBE HASHING

$\text{hash}(\text{key}) \% N$

A

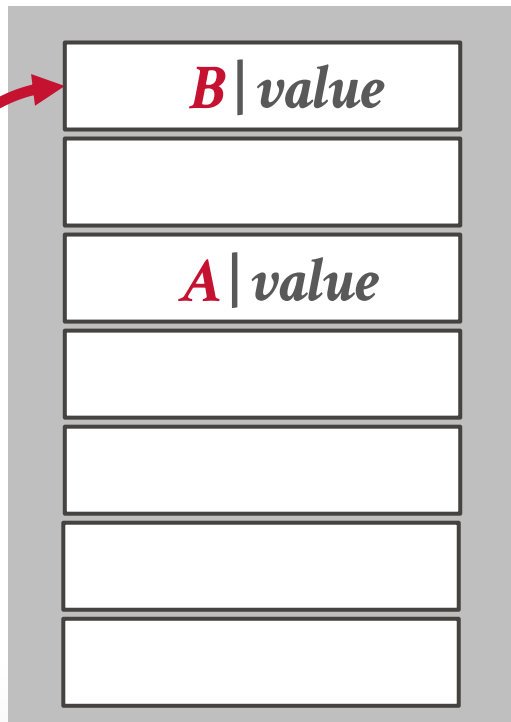
B

C

D

E

F



# LINEAR PROBE HASHING

$\text{hash}(\text{key}) \% N$

A

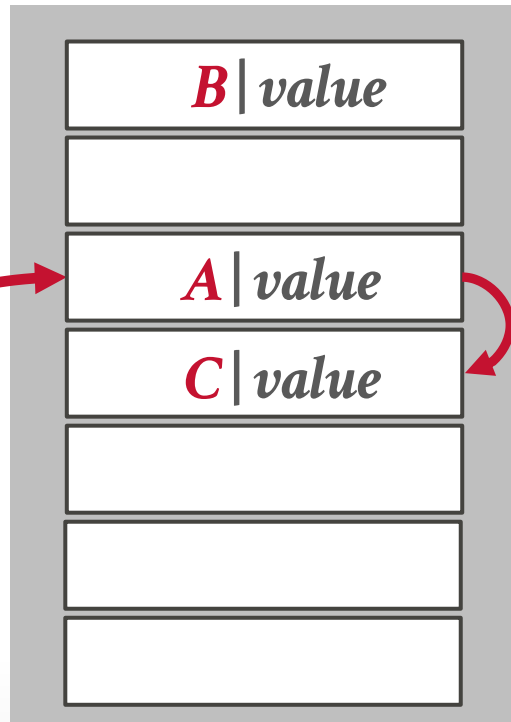
B

C

D

E

F



# LINEAR PROBE HASHING

$\text{hash}(\text{key}) \% N$

A

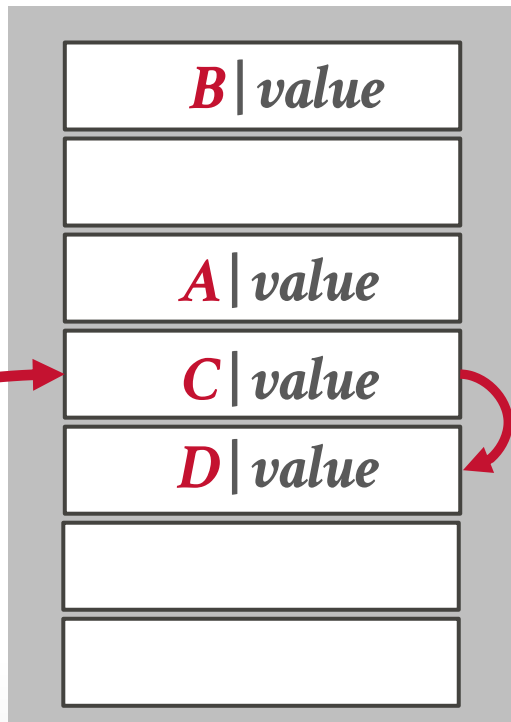
B

C

D

E

F



# LINEAR PROBE HASHING

$hash(key) \% N$

A

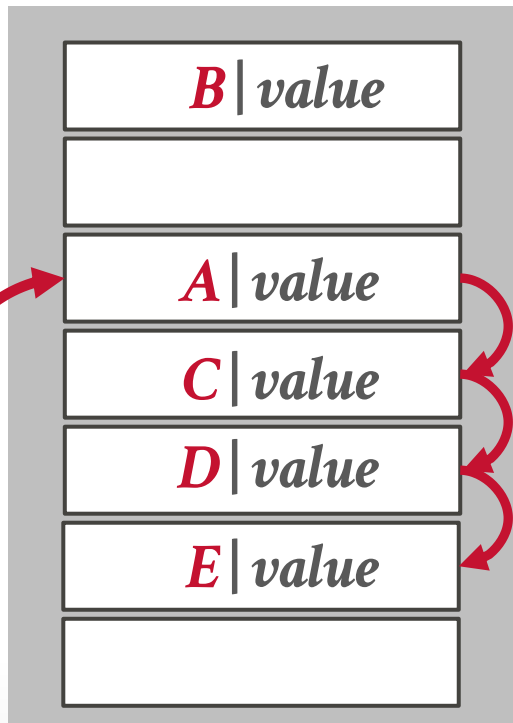
B

C

D

E

F



发生哈希冲突时, 线性扫描直至获取下一个空闲位置.



# LINEAR PROBE HASHING

$hash(key) \% N$

A

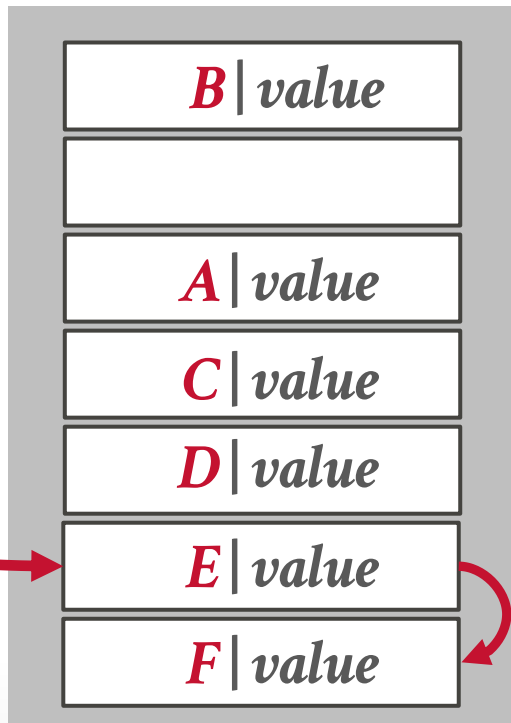
B

C

D

E

F



# HASH TABLE – KEY/VALUE ENTRIES

## Fixed-length Key/Values:

- Store inline within the hash table pages.
- **Optional: Store the key's hash with the key for faster comparisons.**

空间换时间：少量的空间存储哈希值，以加快查找速度。

hash	key	value
hash	key	value
hash	key	value

⋮

## Variable-length Key/Values:

- Insert key/value data in separate a private temporary table.
- Store the hash as the key and use the record id pointing to its corresponding entry in the temporary table as the value.

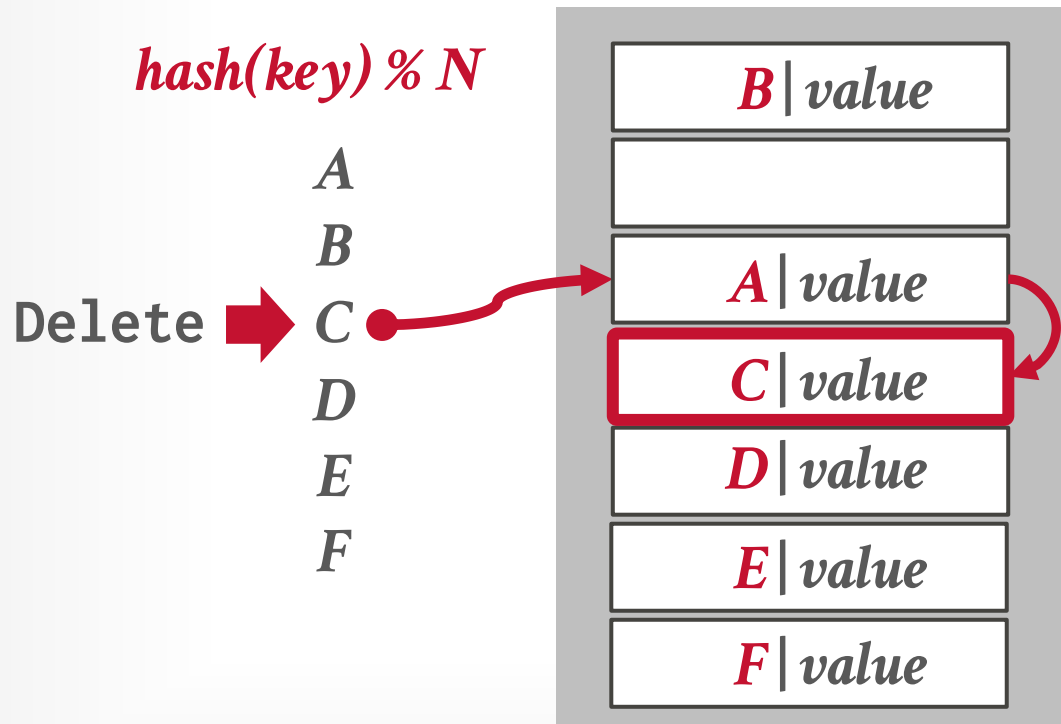
hash	RecordId
hash	RecordId
hash	RecordId

⋮

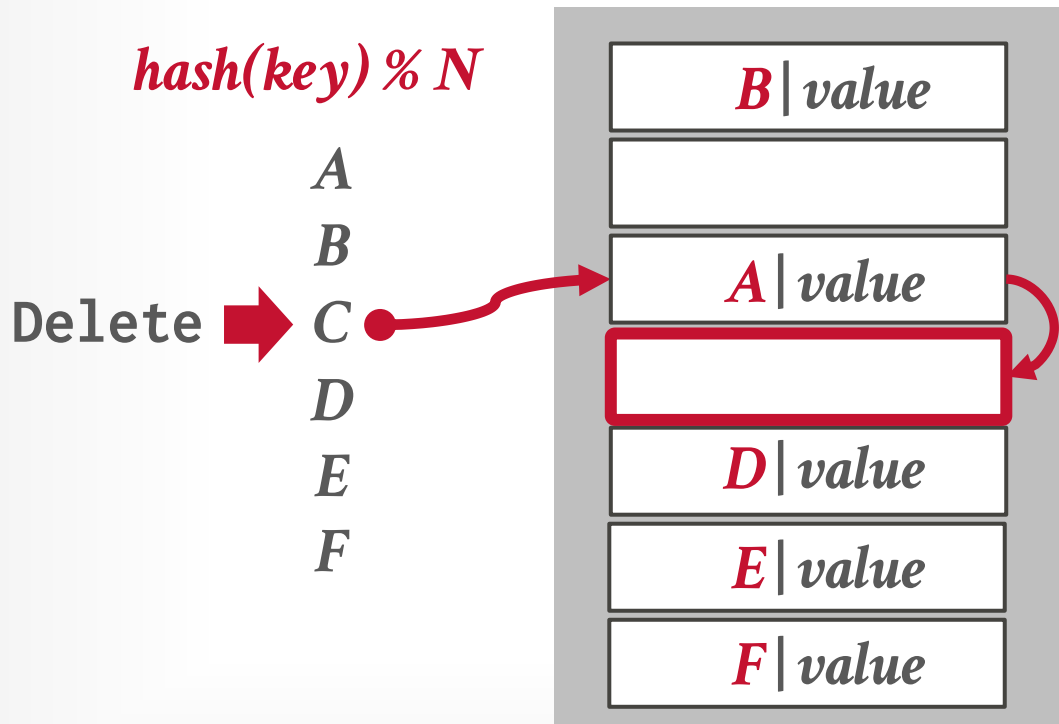
### *Temp Table Page*

key   value	
key   value	
key   value	

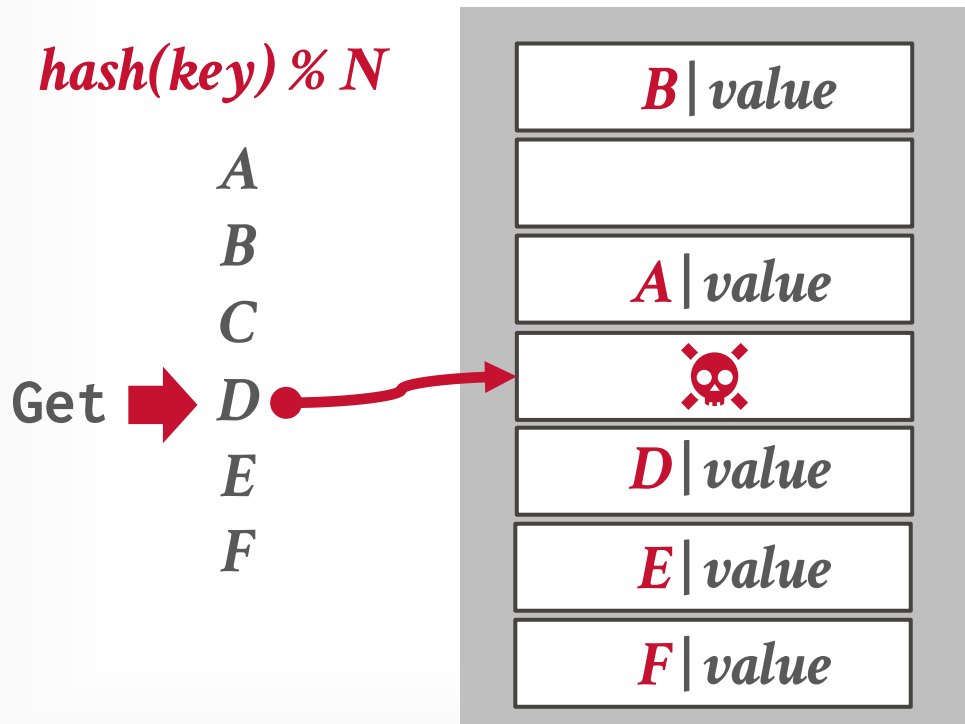
# LINEAR PROBE HASHING – DELETES



# LINEAR PROBE HASHING – DELETES



# LINEAR PROBE HASHING – DELETES



查找元素: 如果发现空槽位, 则结束查找流程, 表示该元素不在哈希表中.

# LINEAR PROBE HASHING – DELETES

$\text{hash}(\text{key}) \% N$

A

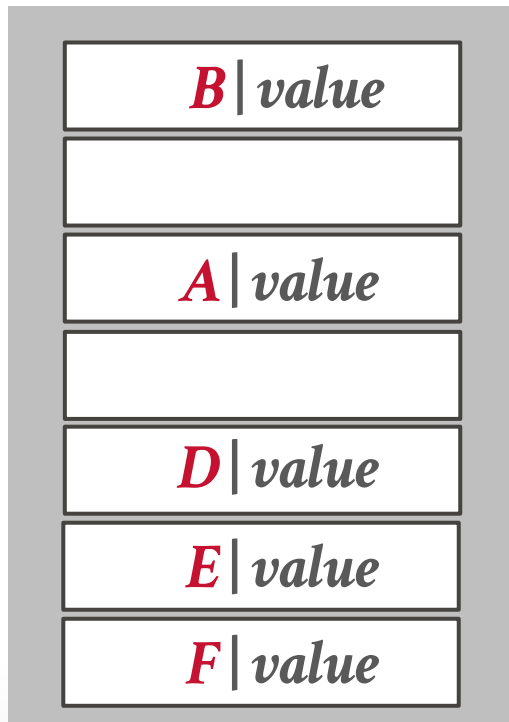
B

C

Get  D

E

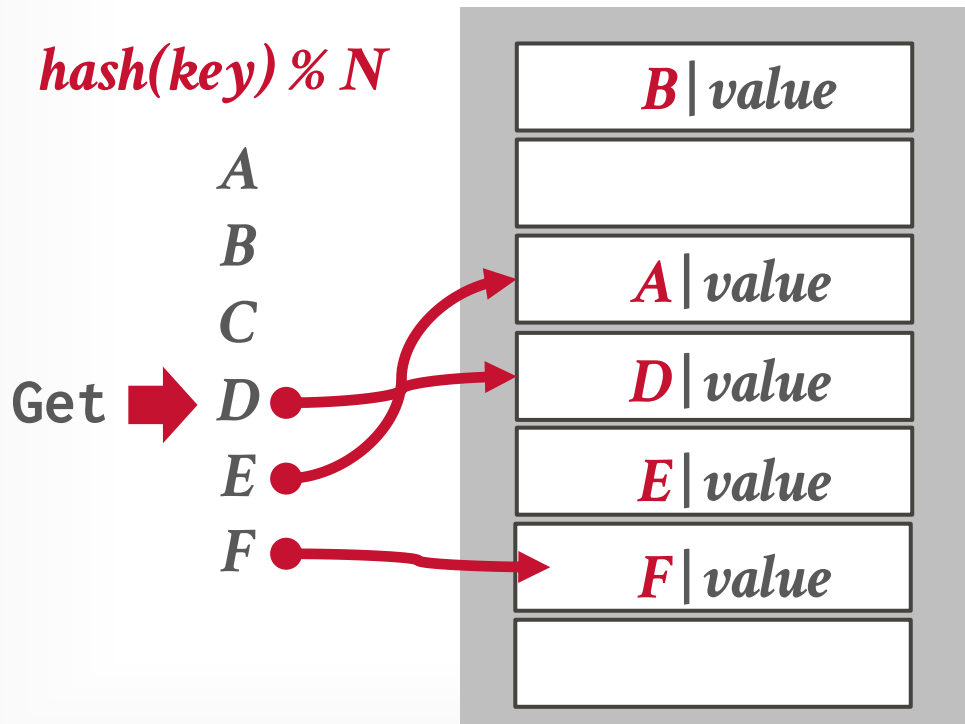
F



**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

$hash(key) \% N$

A

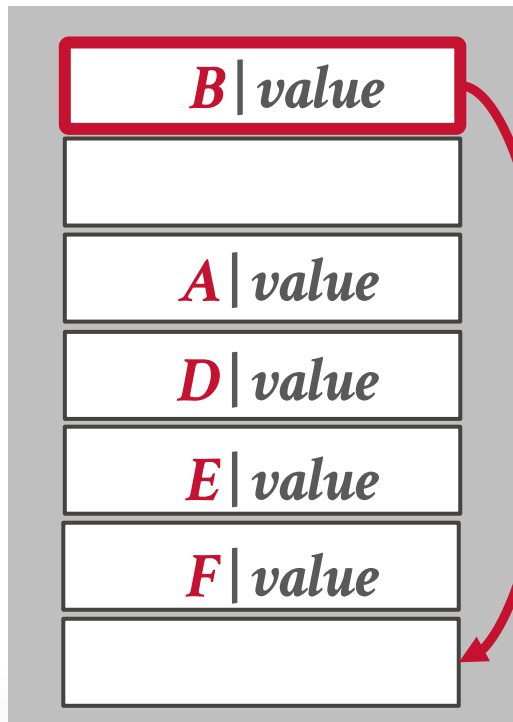
B

C

Get → D

E

F

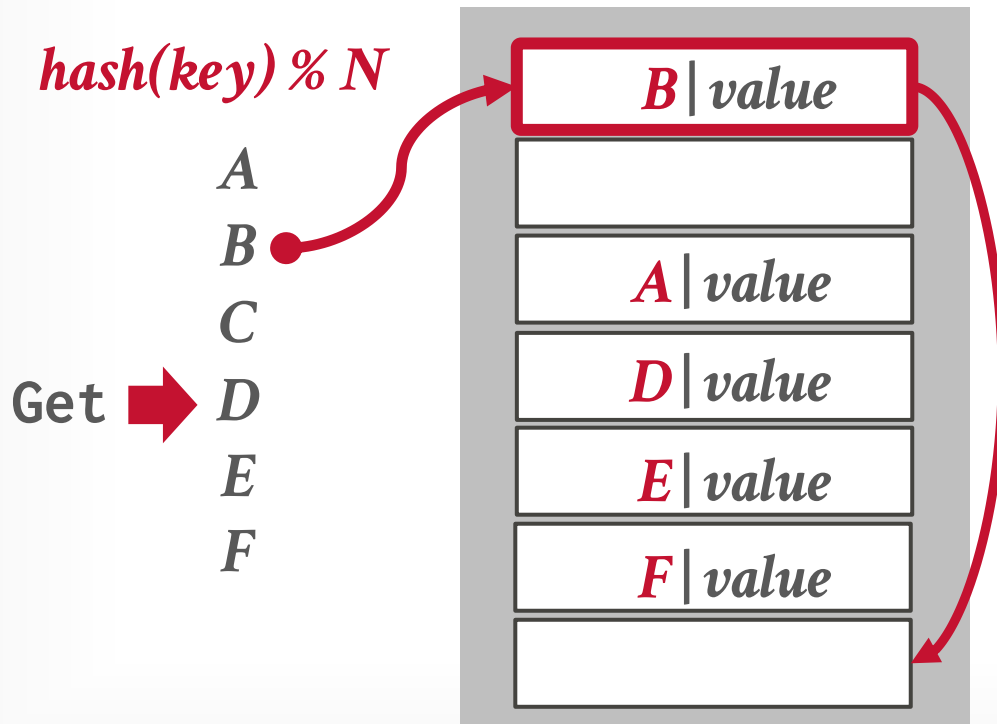


**Approach #1: Movement**

→ Rehash keys until you find the first empty slot.



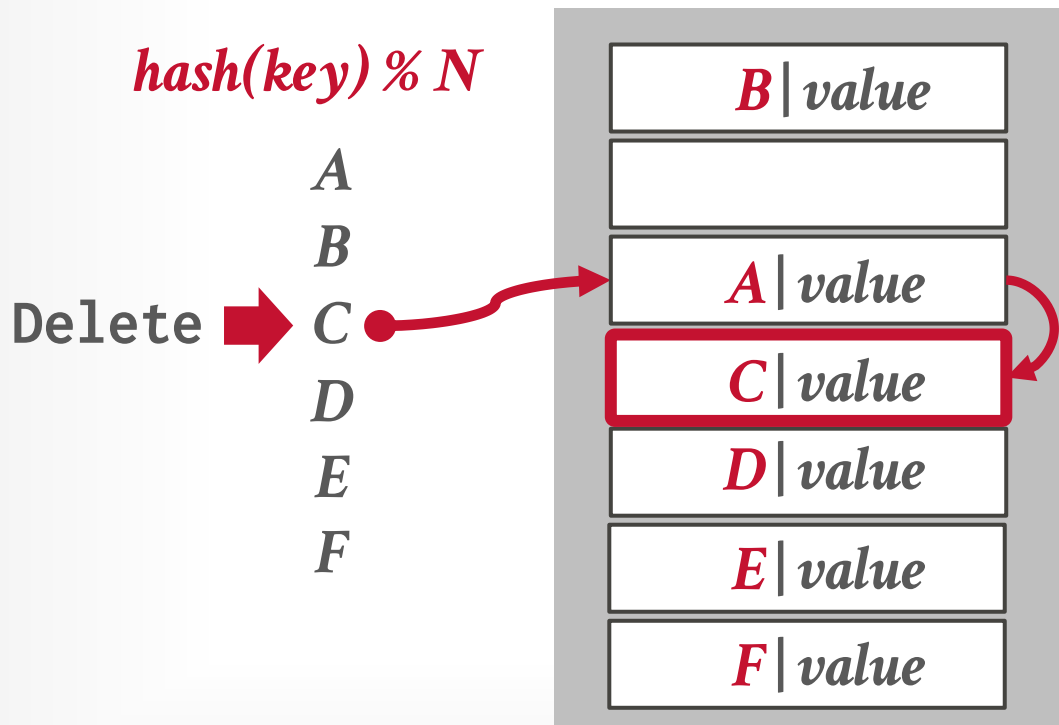
# LINEAR PROBE HASHING – DELETES



## 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

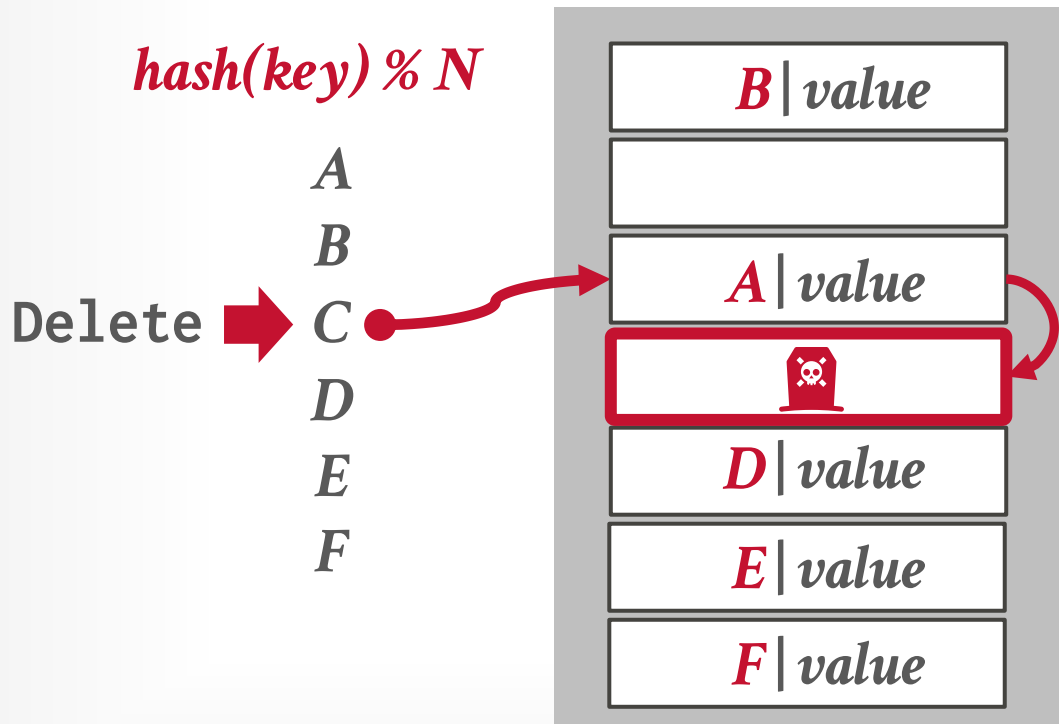


使用逻辑删除位进行标记删除的元素。

## Approach #2: Tombstone

- Maintain **separate bit map** to indicate that the entry in the slot is logically deleted.
- Reuse the slot for new keys.
- May need periodic garbage collection.

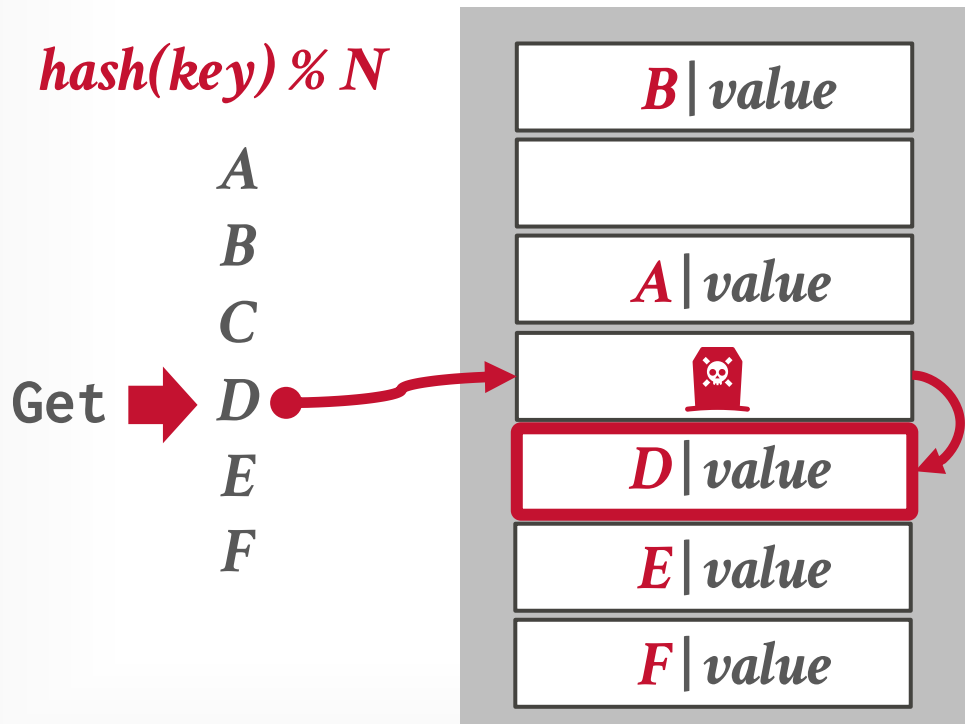
# LINEAR PROBE HASHING – DELETES



## Approach #2: Tombstone

- Maintain separate bit map to indicate that the entry in the slot is logically deleted.
- Reuse the slot for new keys.
- May need periodic garbage collection.

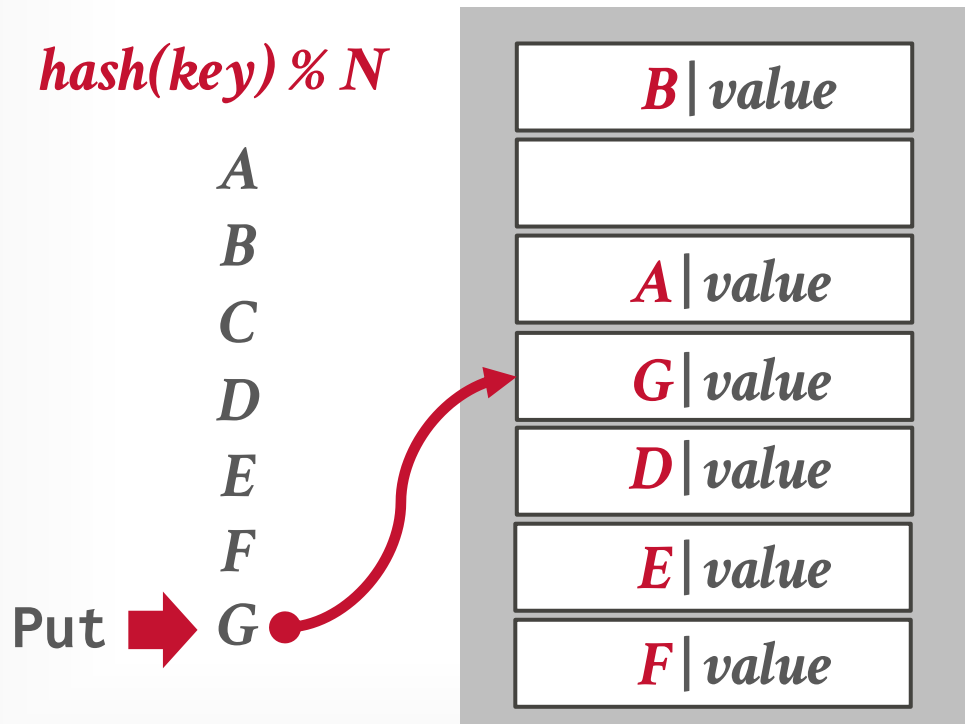
# LINEAR PROBE HASHING – DELETES



## Approach #2: Tombstone

- Maintain separate bit map 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



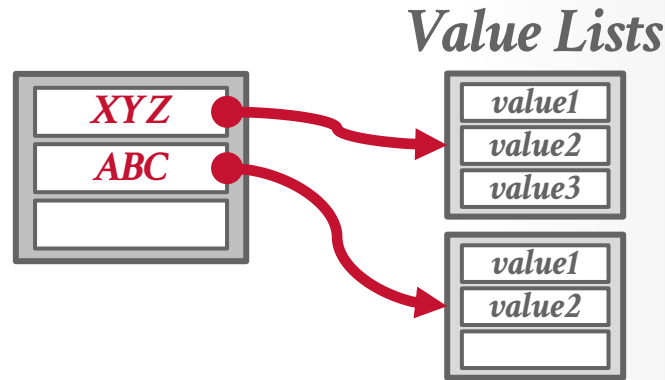
## Approach #2: Tombstone

- Maintain separate bit map to indicate that the entry in the slot is logically deleted.
- Reuse the slot for new keys.
- May need periodic garbage collection.

# HASH TABLE – NON-UNIQUE KEYS

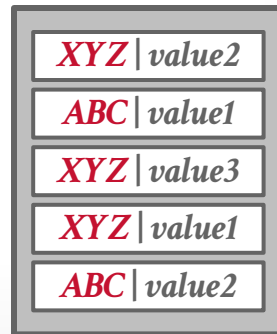
## Choice #1: Separate Linked List

- 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. clickhouse 提供不同类型的哈希表实现.

→ 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.

# OPTIMIZATIONS

Specialized hash table implementation for different key type(s) and sizes.

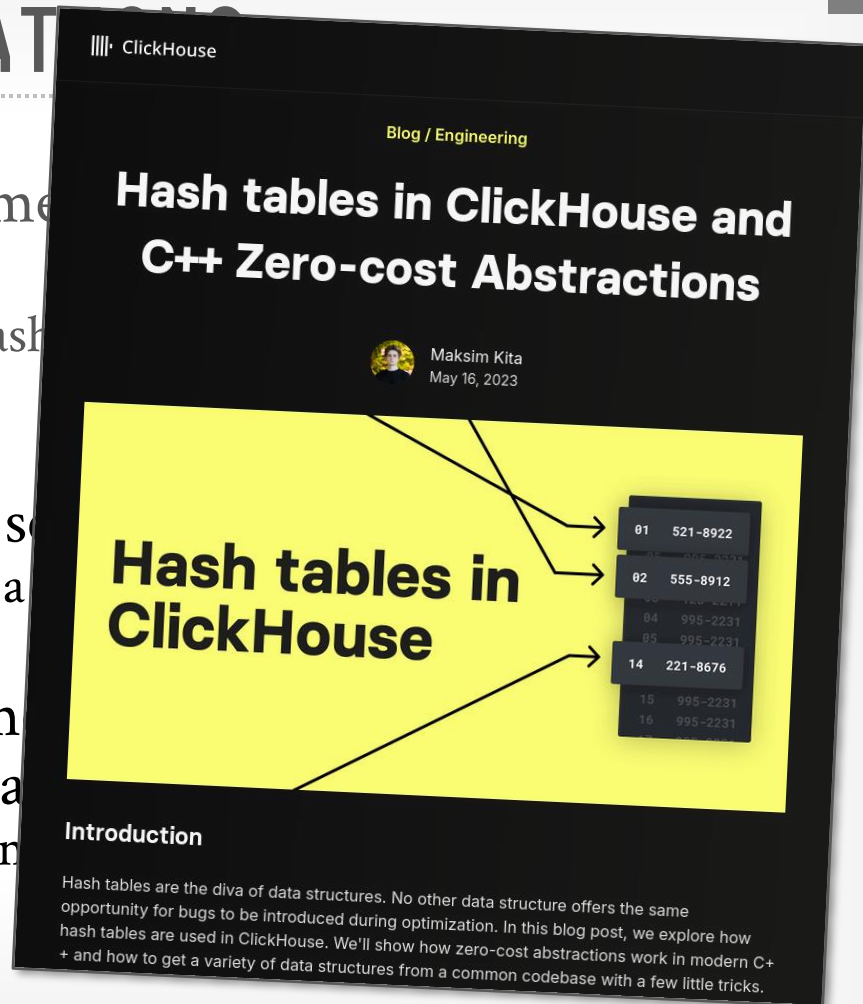
→ Example: Maintain multiple hash table sizes for a set of keys.

Store metadata separate in a separate structure.

→ Packed bitmap tracks whether a slot is valid.

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

→ Example: If table version does not match, 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.

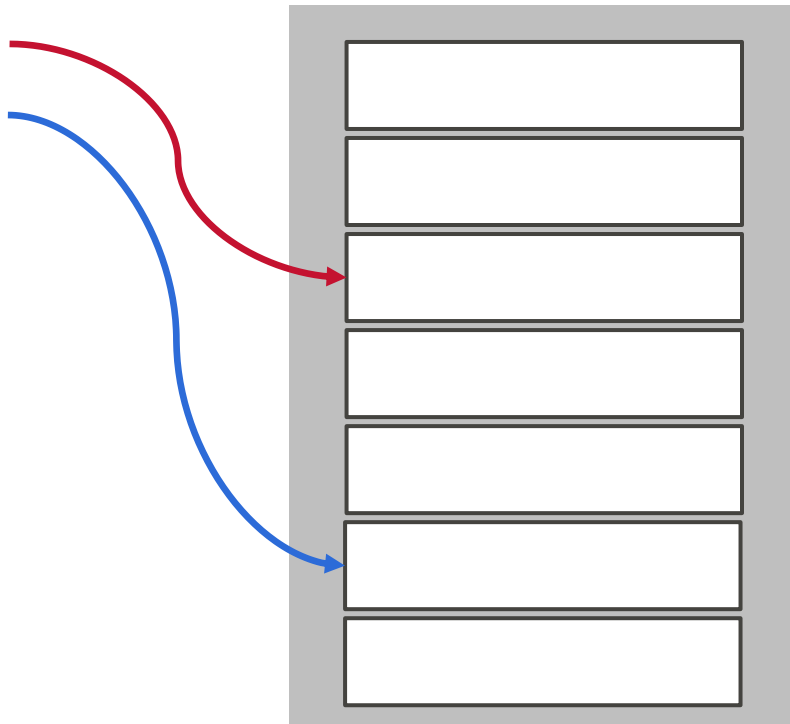
Look-ups and deletions are always  **$O(1)$**  because only one location per hash table is checked.

Best open-source implementation is from CMU.

# CUCKOO HASHING

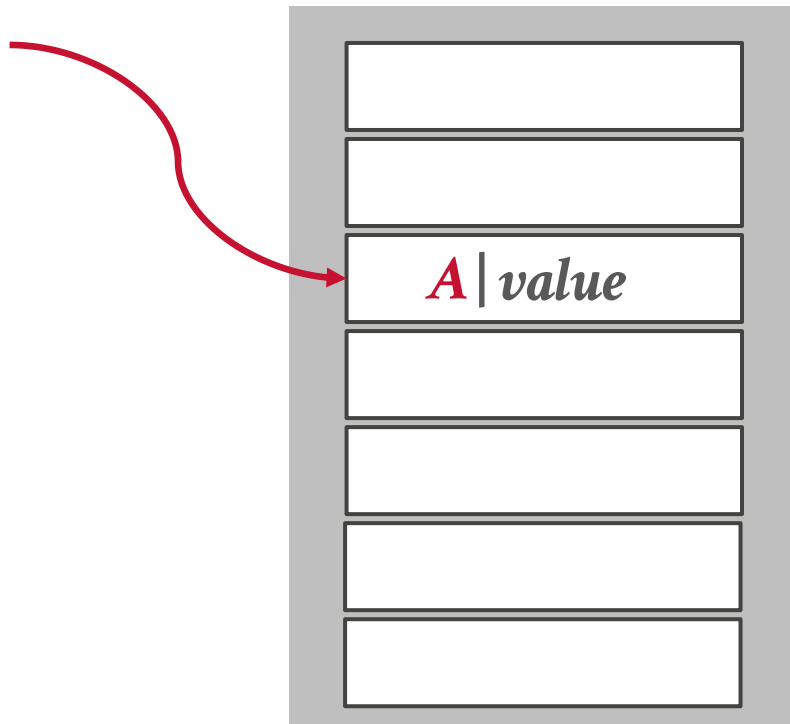
Put A:  $hash_1(A)$

$hash_2(A)$



# CUCKOO HASHING

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



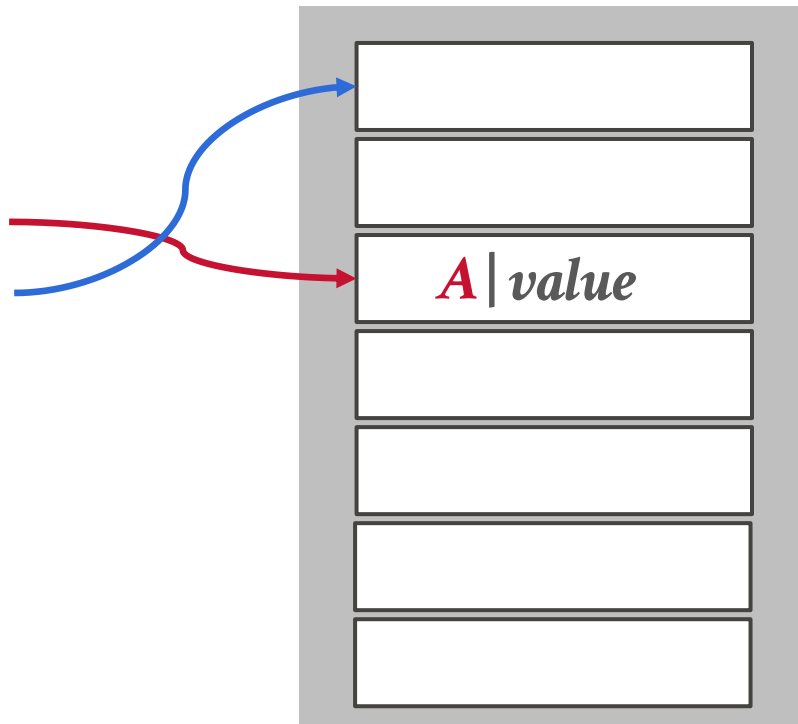
# CUCKOO HASHING

Put A:  $hash_1(A)$

$hash_2(A)$

Put B:  $hash_1(B)$

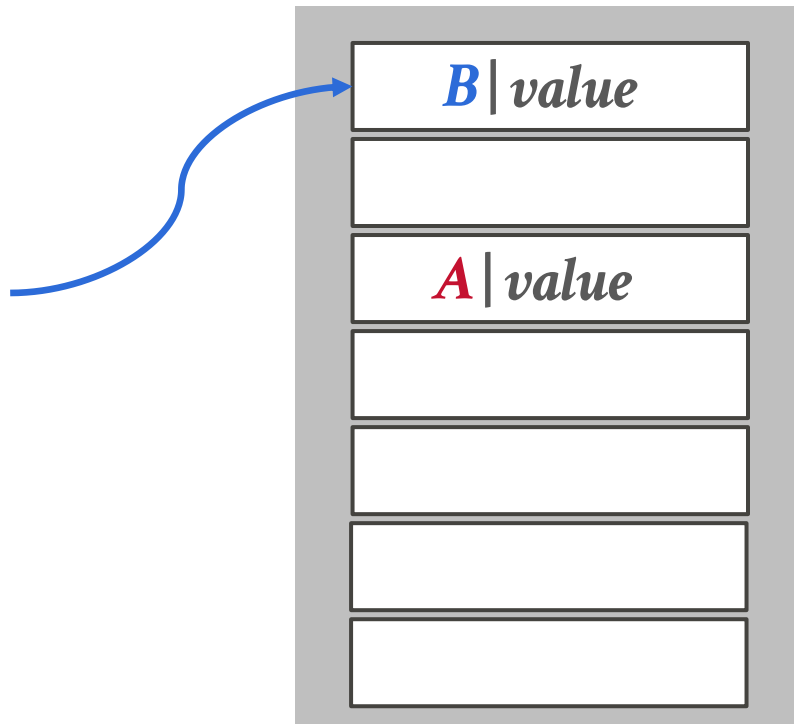
$hash_2(B)$



# CUCKOO HASHING

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

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

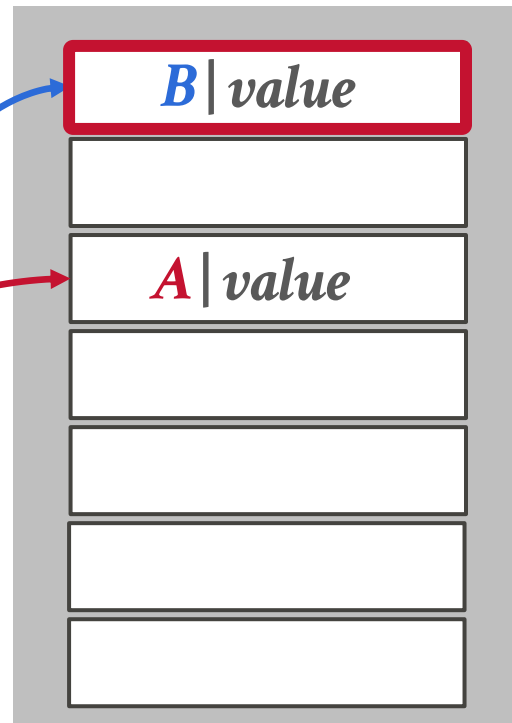


# CUCKOO HASHING

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

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

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

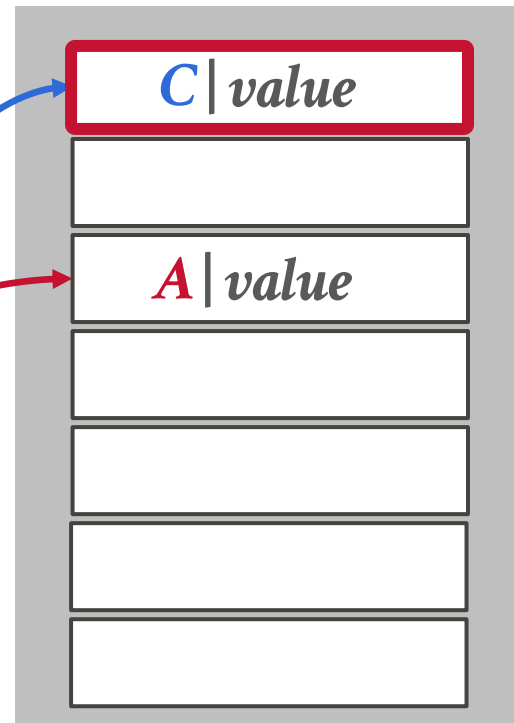


# CUCKOO HASHING

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

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

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

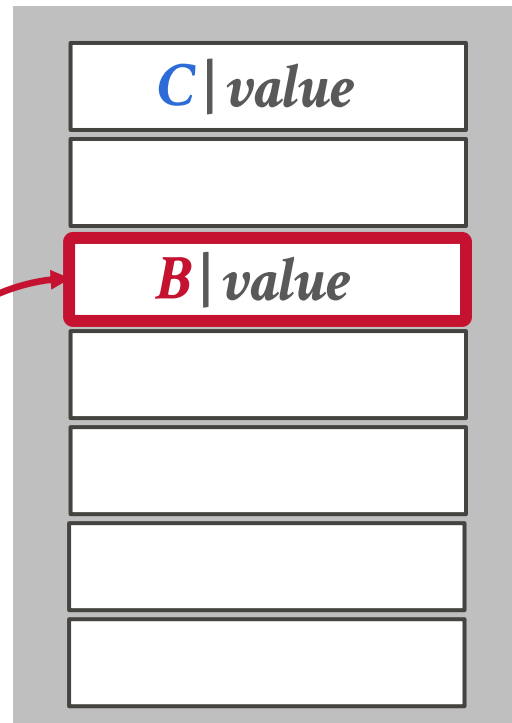


# CUCKOO HASHING

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

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

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



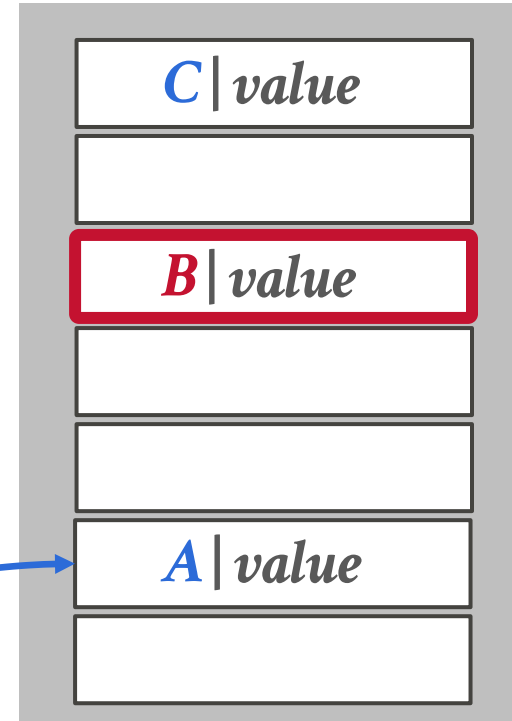


# CUCKOO HASHING

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

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

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



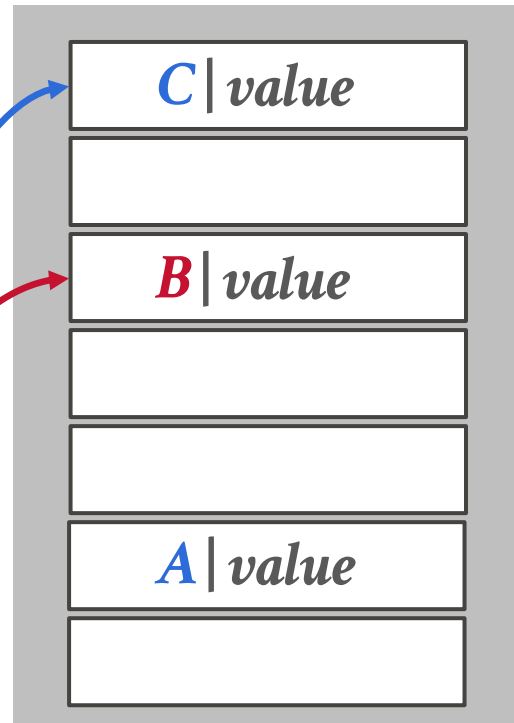
# CUCKOO HASHING

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

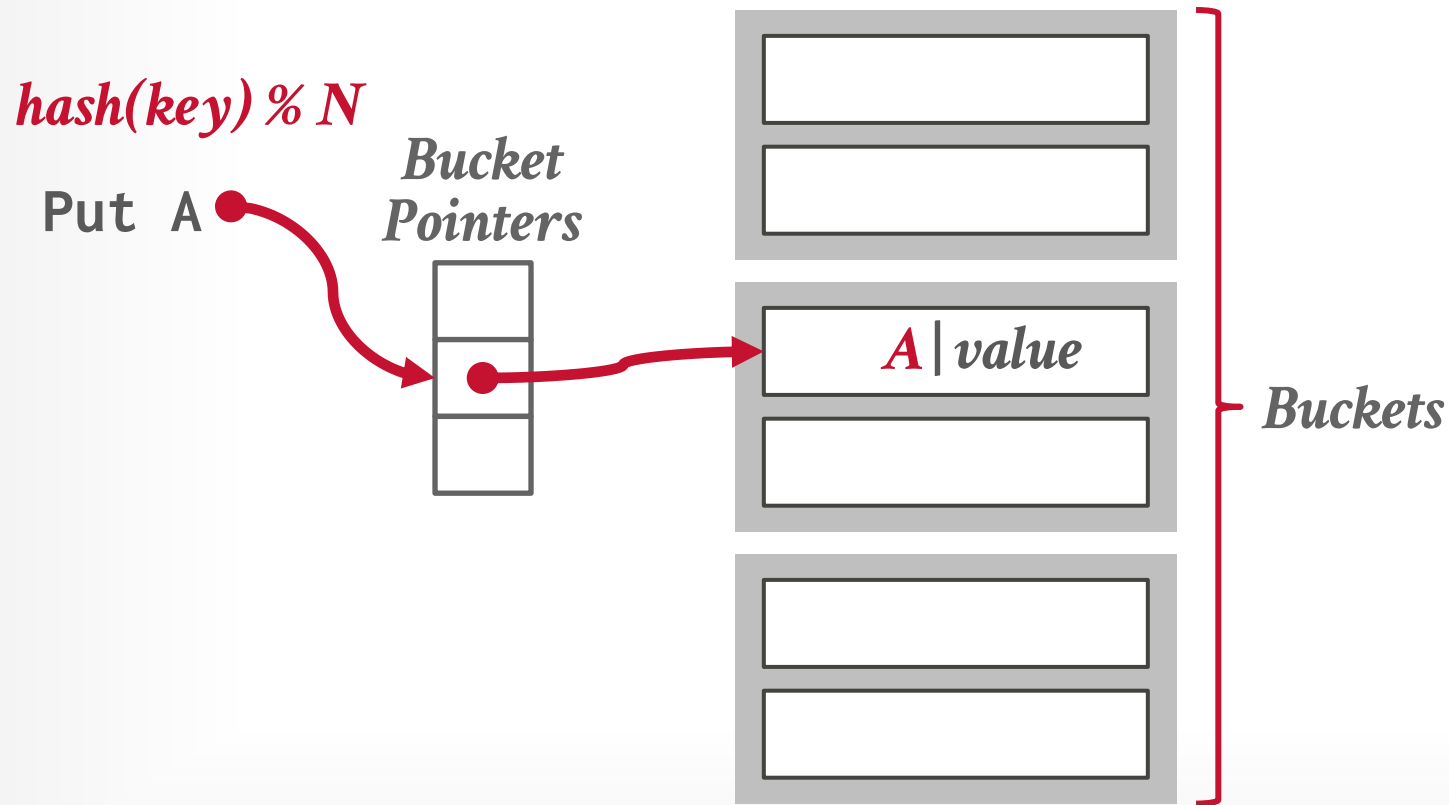
---

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

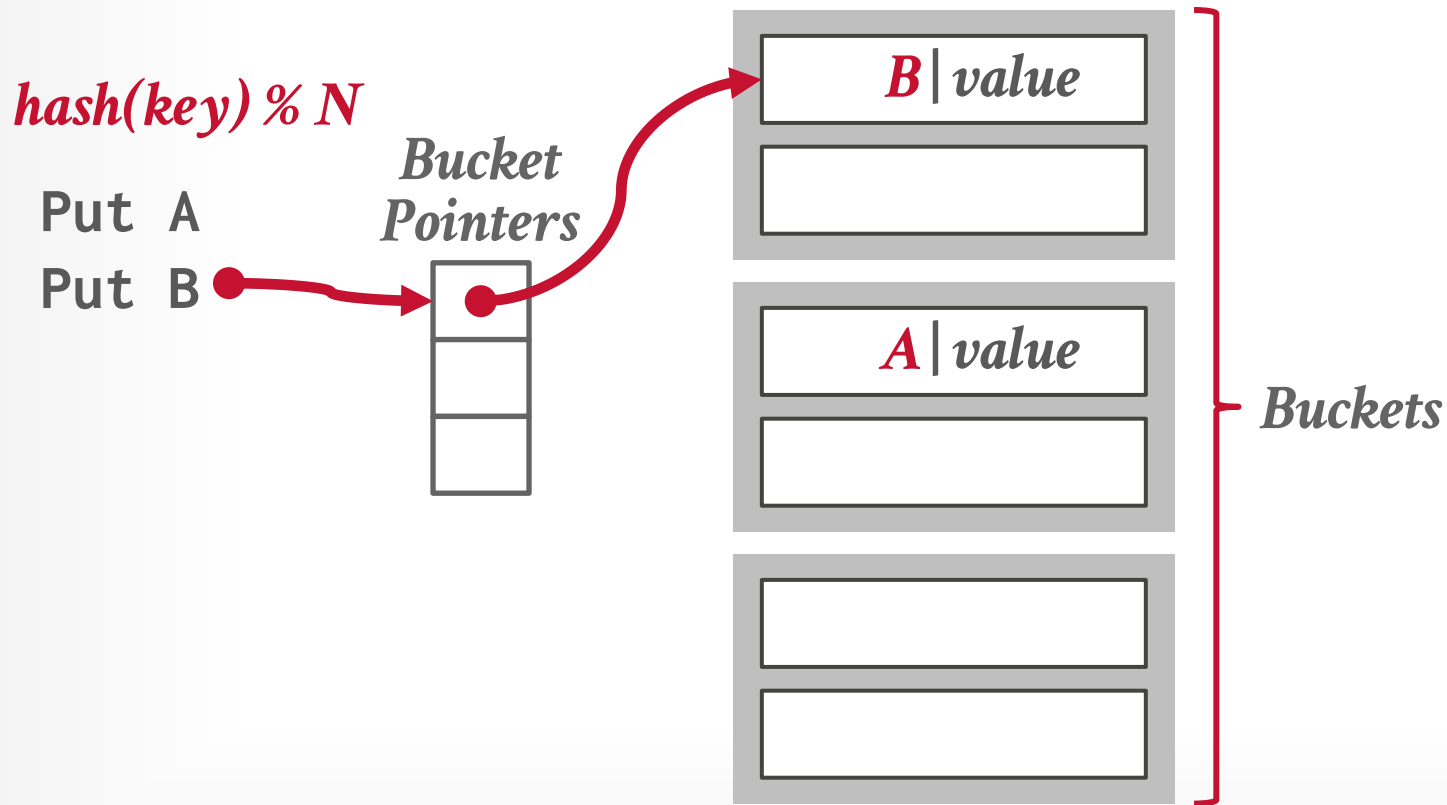
Resolve collisions by placing all elements with the same hash key into the same bucket.

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

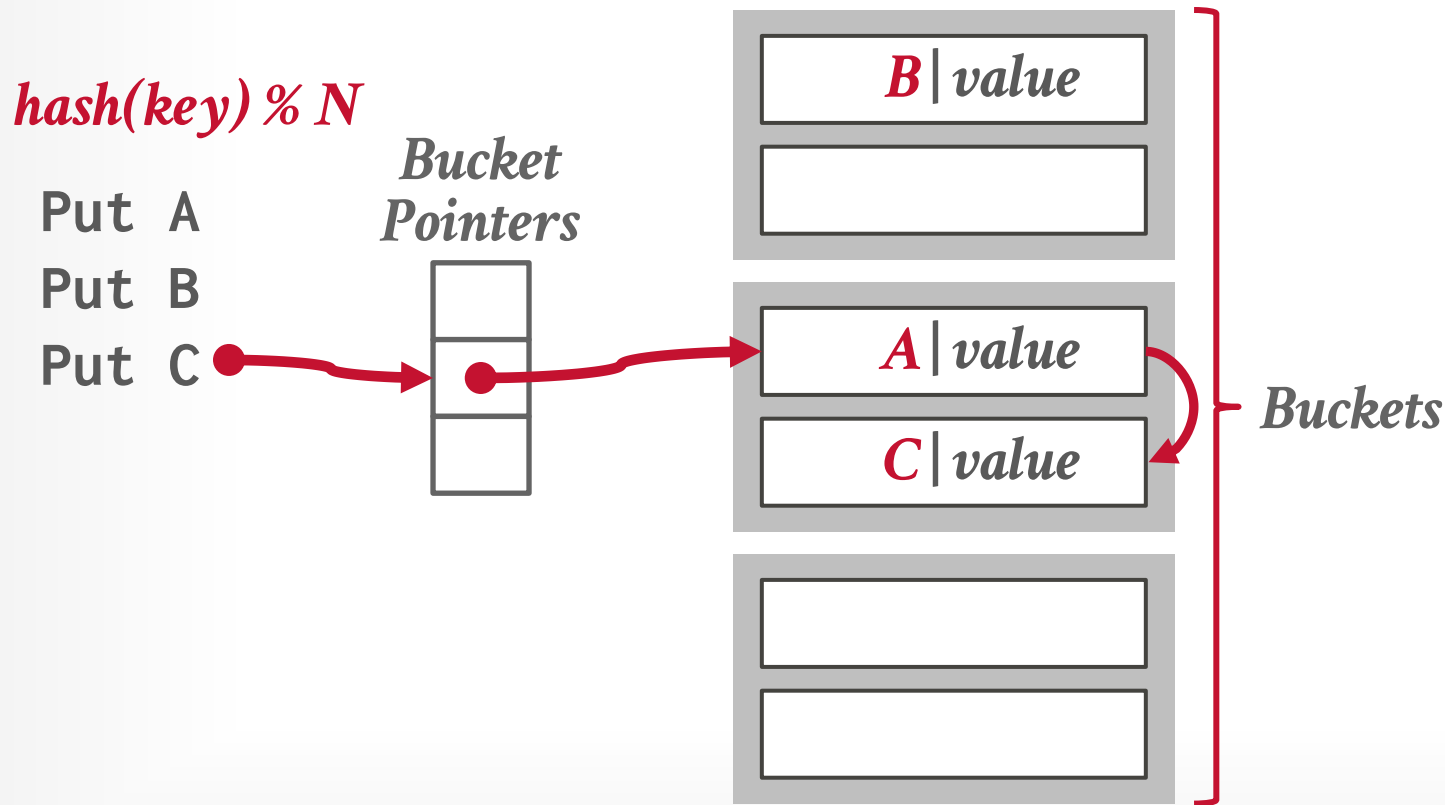
# CHAINED HASHING



# CHAINED HASHING



# CHAINED HASHING



# CHAINED HASHING

$\text{hash}(\text{key}) \% N$

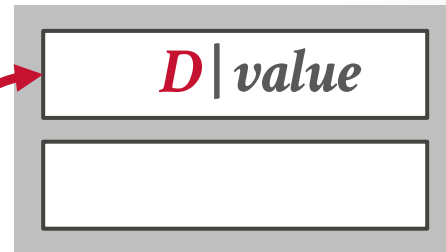
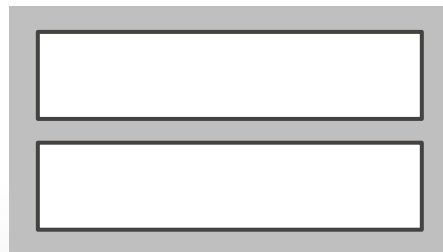
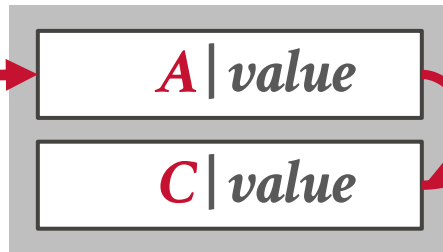
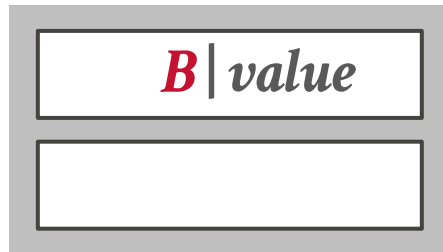
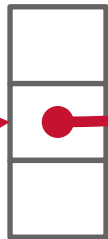
Put A

Put B

Put C

Put D

*Bucket  
Pointers*



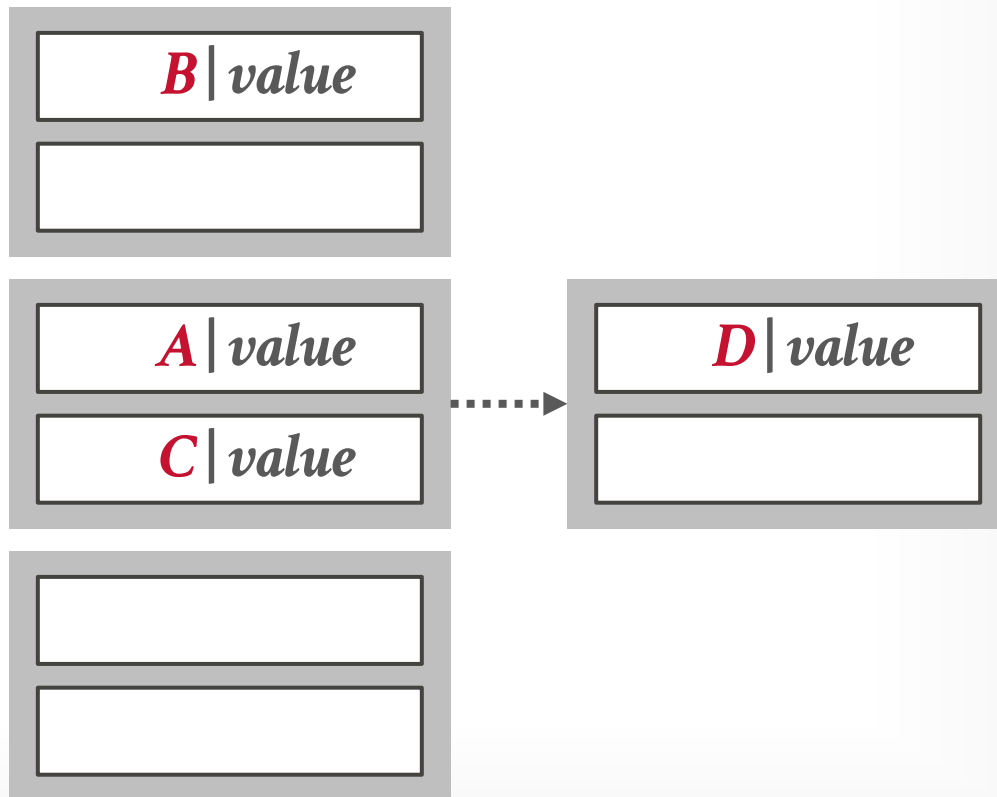


# CHAINED HASHING

$\text{hash}(\text{key}) \% N$

Put A  
Put B  
Put C  
Put D  
Put E

*Bucket  
Pointers*



# CHAINED HASHING

$hash(key) \% N$

Put A

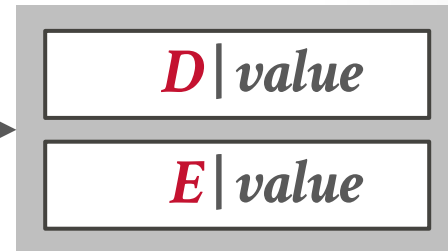
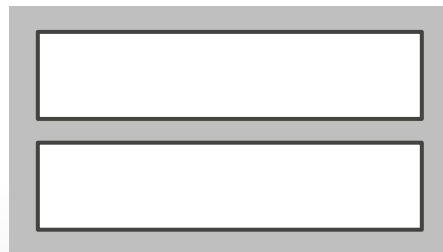
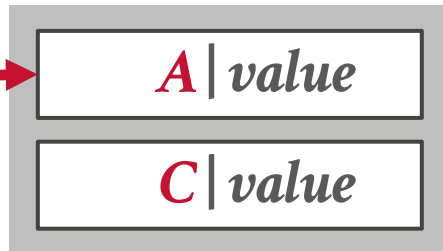
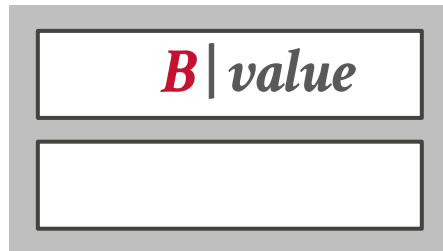
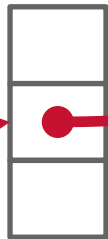
Put B

Put C

Put D

Put E

*Bucket  
Pointers*



# CHAINED HASHING

$hash(key) \% N$

Put A

Put B

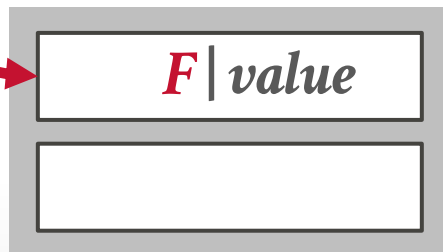
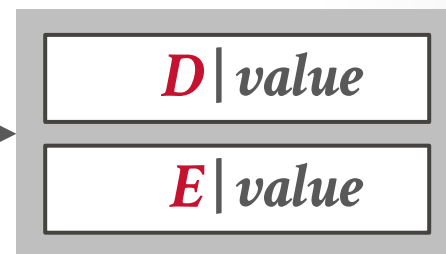
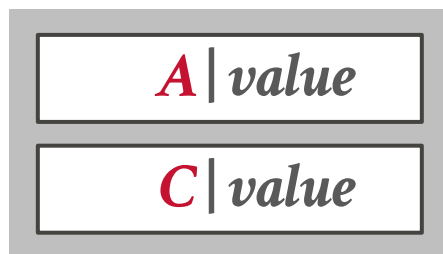
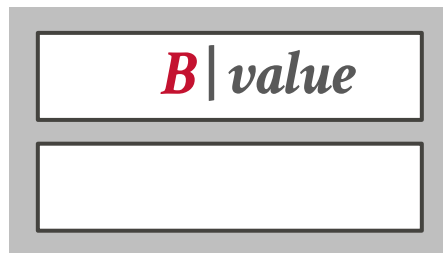
Put C

Put D

Put E

Put F

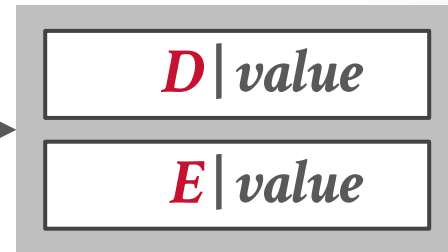
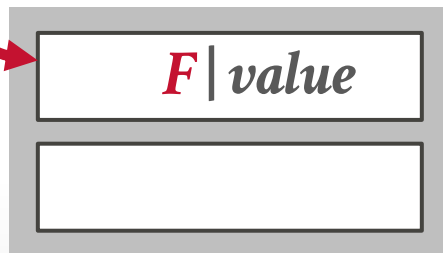
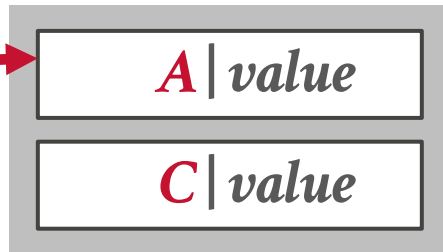
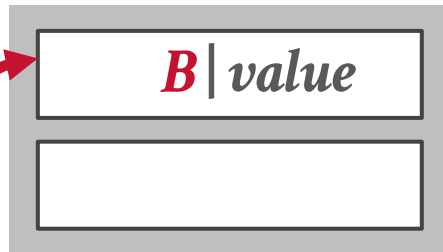
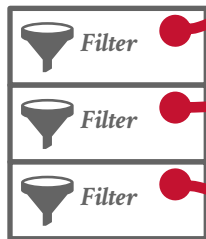
*Bucket  
Pointers*



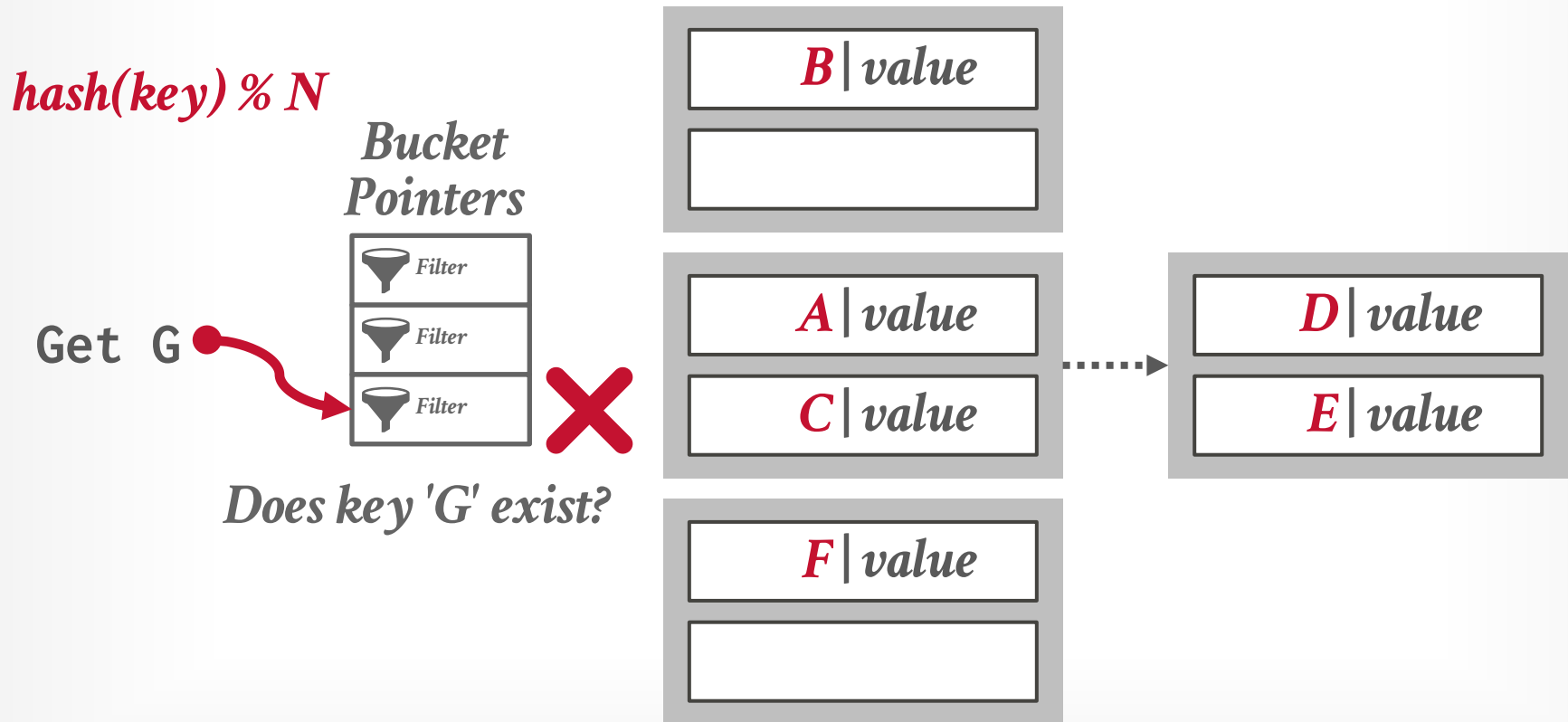
# CHAINED HASHING

$\text{hash}(\text{key}) \% N$

*Bucket  
Pointers*



# 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.

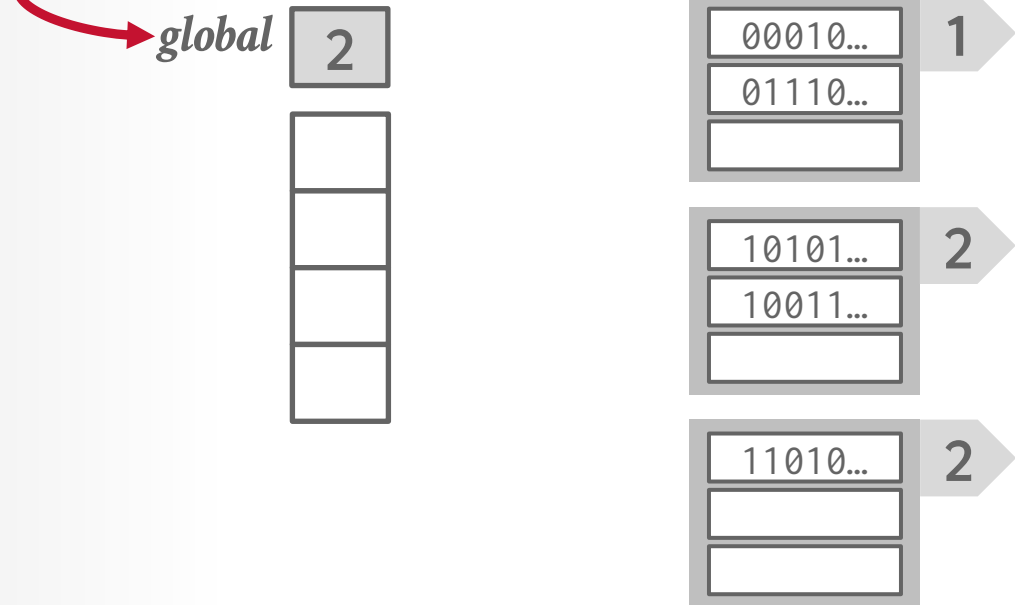
Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

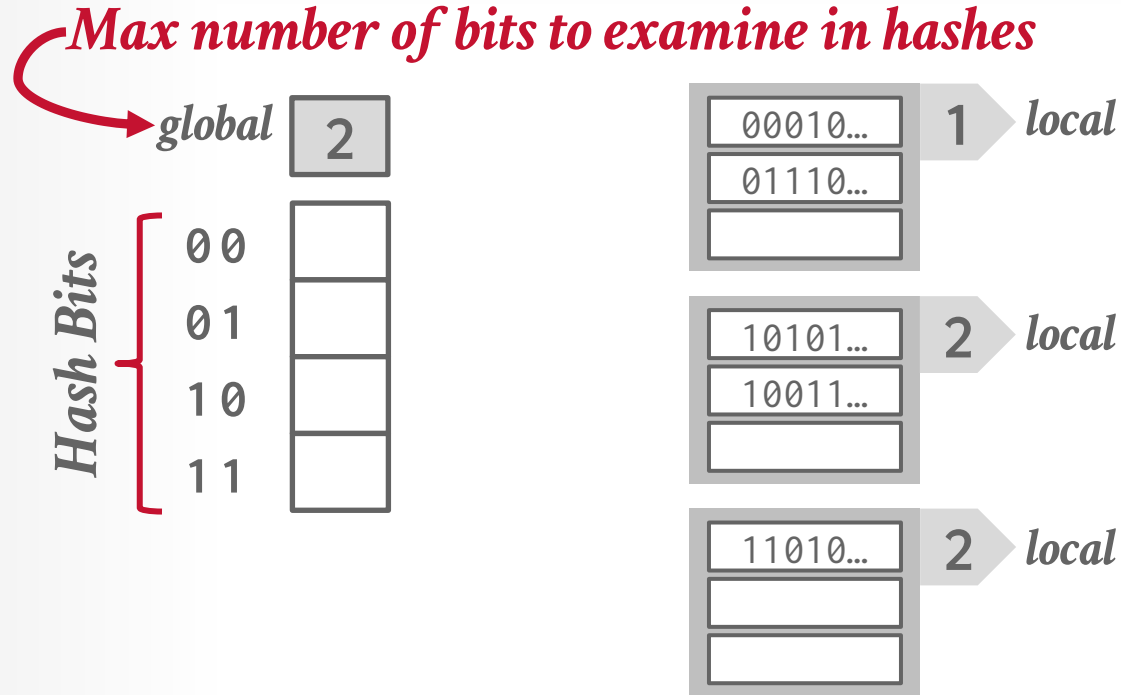


# EXTENDIBLE HASHING

*Max number of bits to examine in hashes*

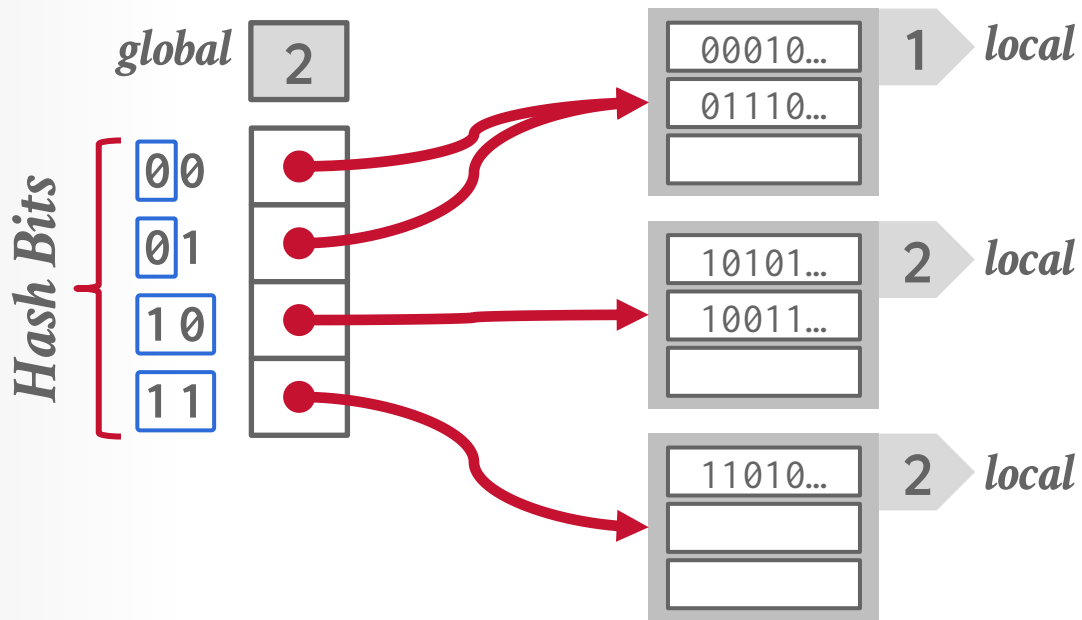


# EXTENDIBLE HASHING

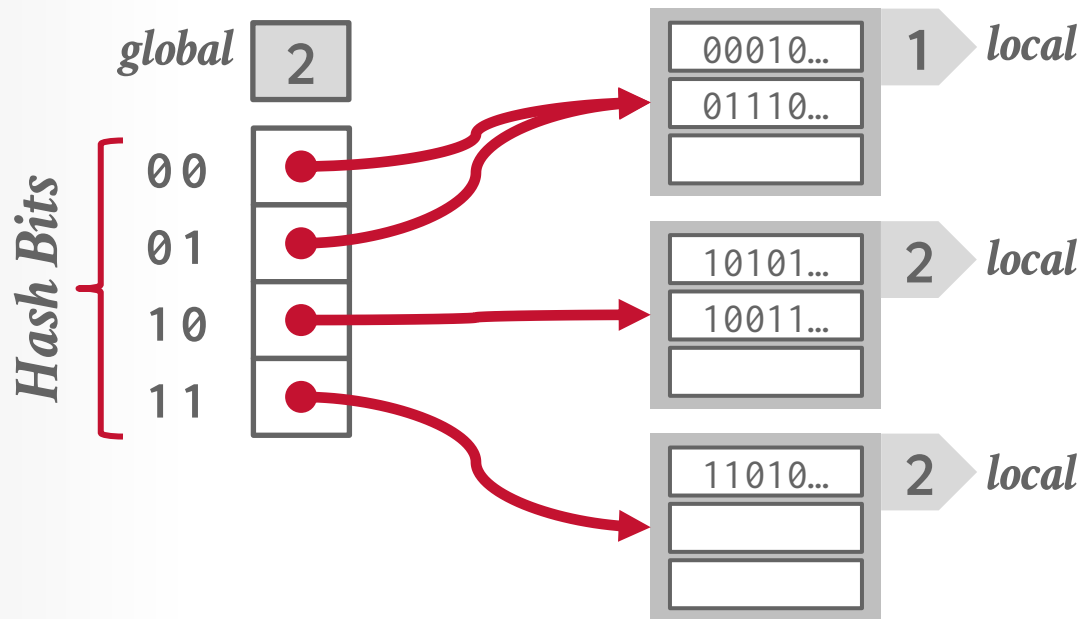




# EXTENDIBLE HASHING

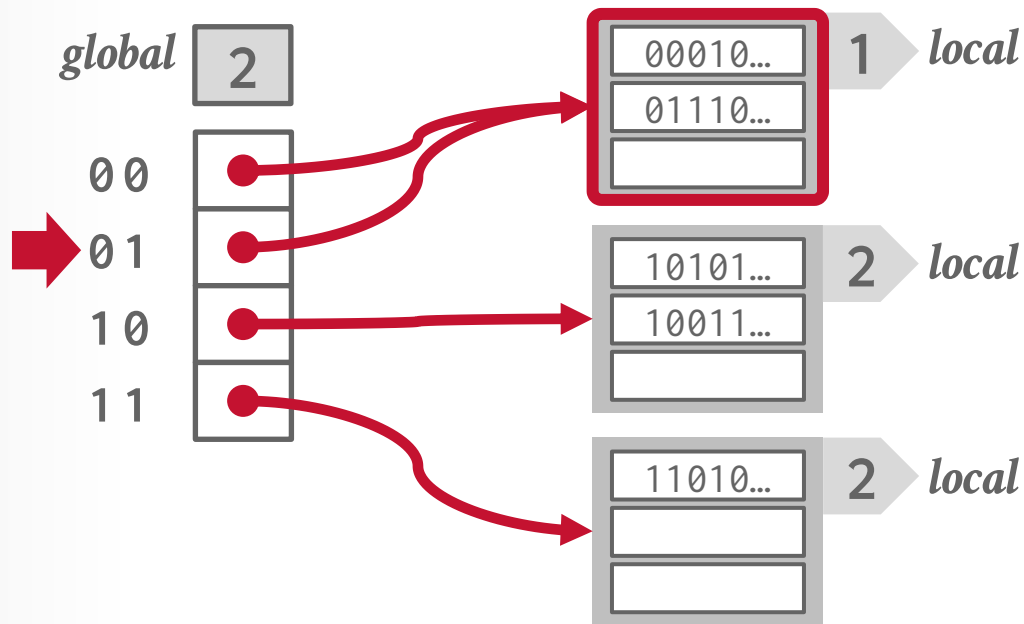


# EXTENDIBLE HASHING



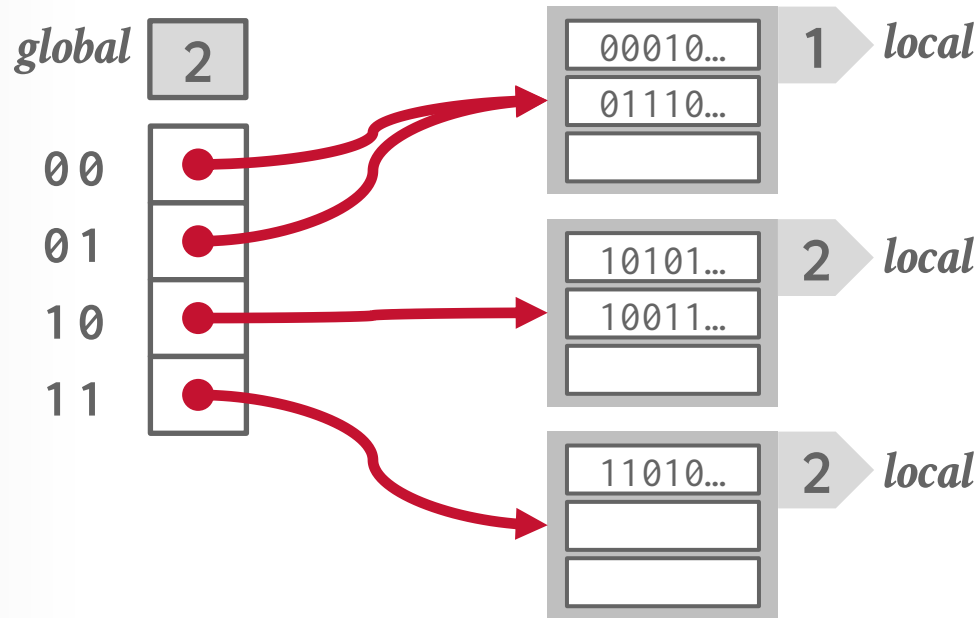
Get A  
 $\text{hash}(A) = \boxed{01}110\dots$

# EXTENDIBLE HASHING



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

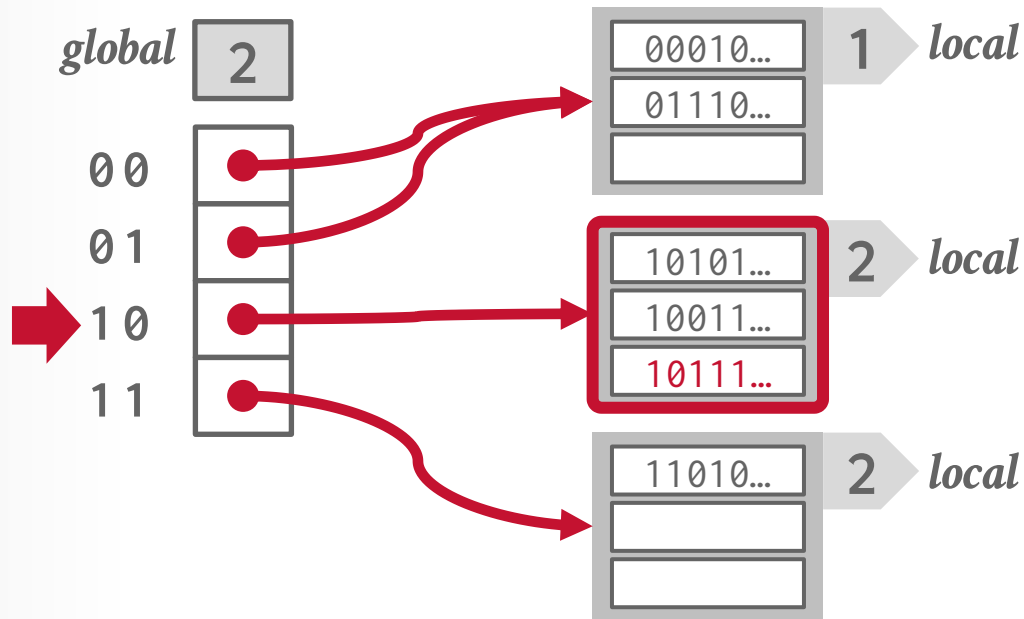
# EXTENDIBLE HASHING



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

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

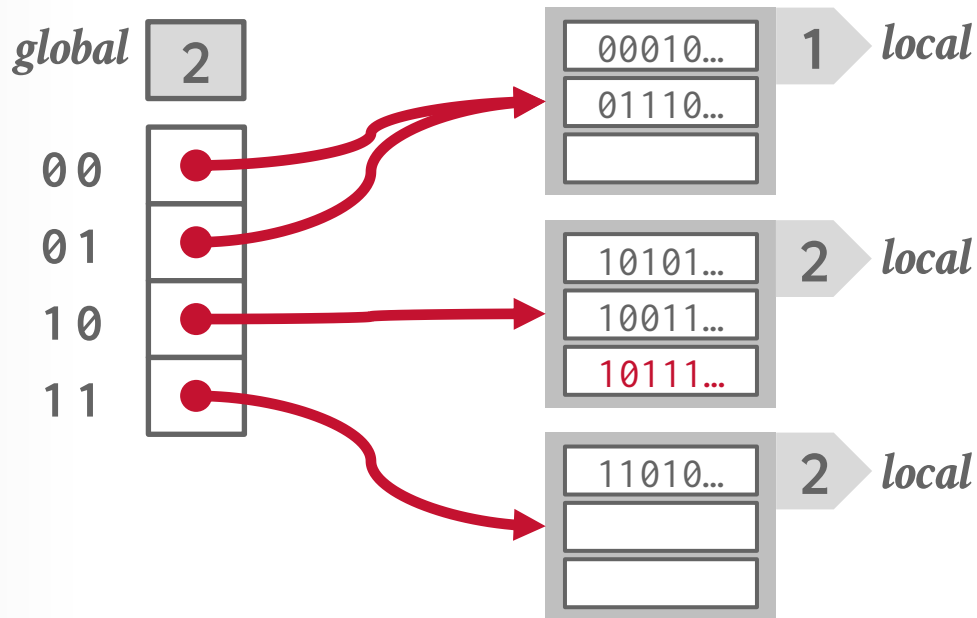
# EXTENDIBLE HASHING



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

Put B  
 $hash(B) = 10111...$

# EXTENDIBLE HASHING

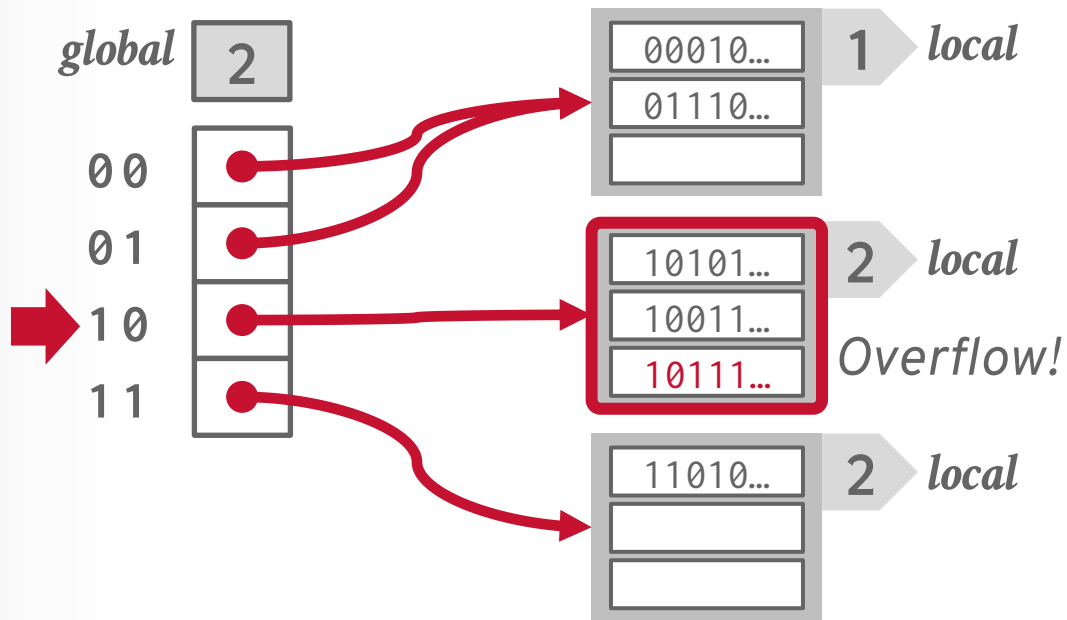


Get A  
 $hash(A) = 01110...$

Put B  
 $hash(B) = 10111...$

Put C  
 $hash(C) = 10100...$

# EXTENDIBLE HASHING

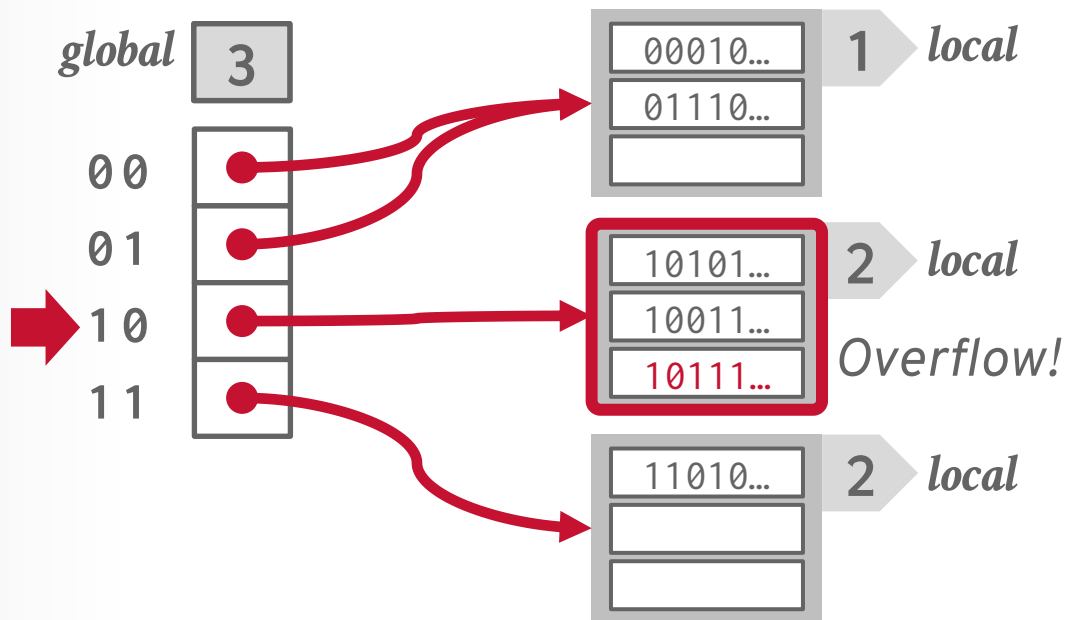


Get A  
 $hash(A) = 01110...$

Put B  
 $hash(B) = 10111...$

Put C  
 $hash(C) = 10100...$

# EXTENDIBLE HASHING



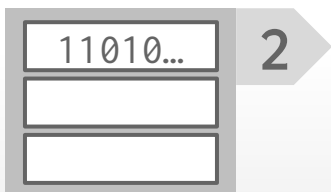
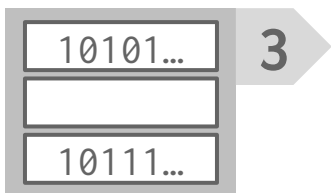
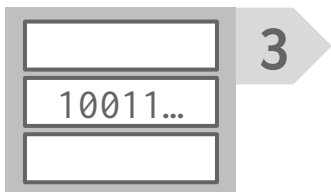
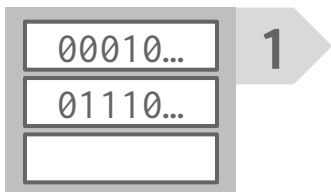
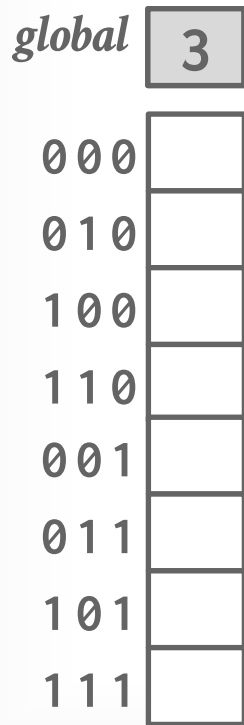
Get A  
 $hash(A) = 01110...$

Put B  
 $hash(B) = 10111...$

Put C  
 $hash(C) = 10100...$



# EXTENDIBLE HASHING



Get A

*hash(A) = 01110...*

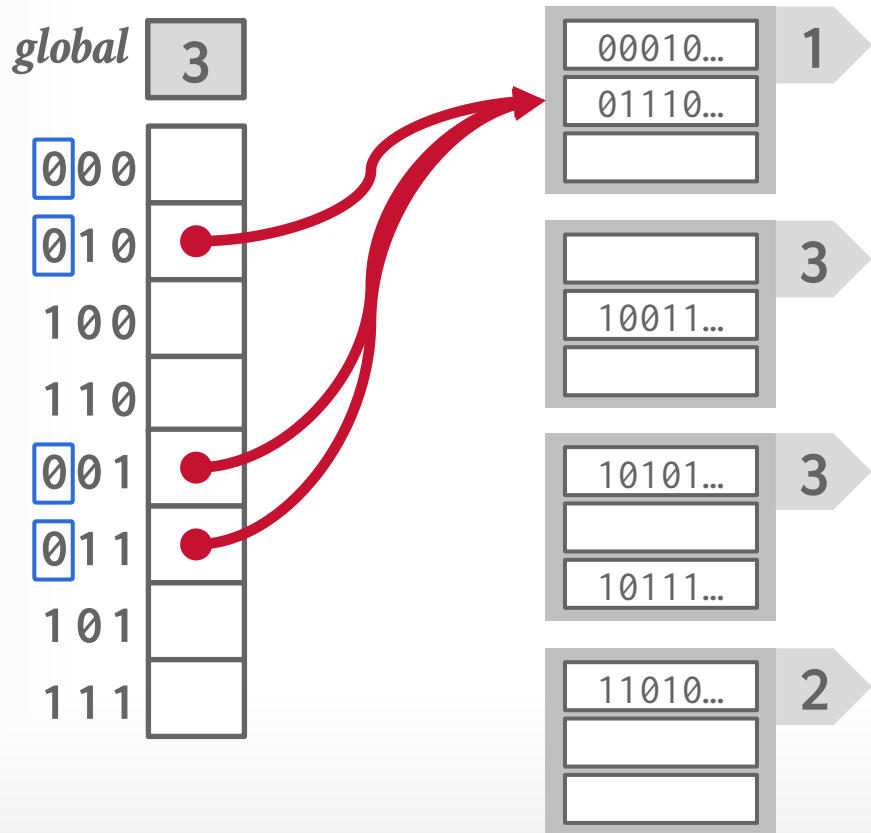
Put B

*hash(B) = 10111...*

Put C

*hash(C) = 10100...*

# EXTENDIBLE HASHING



Get A

$hash(A) = 01110...$

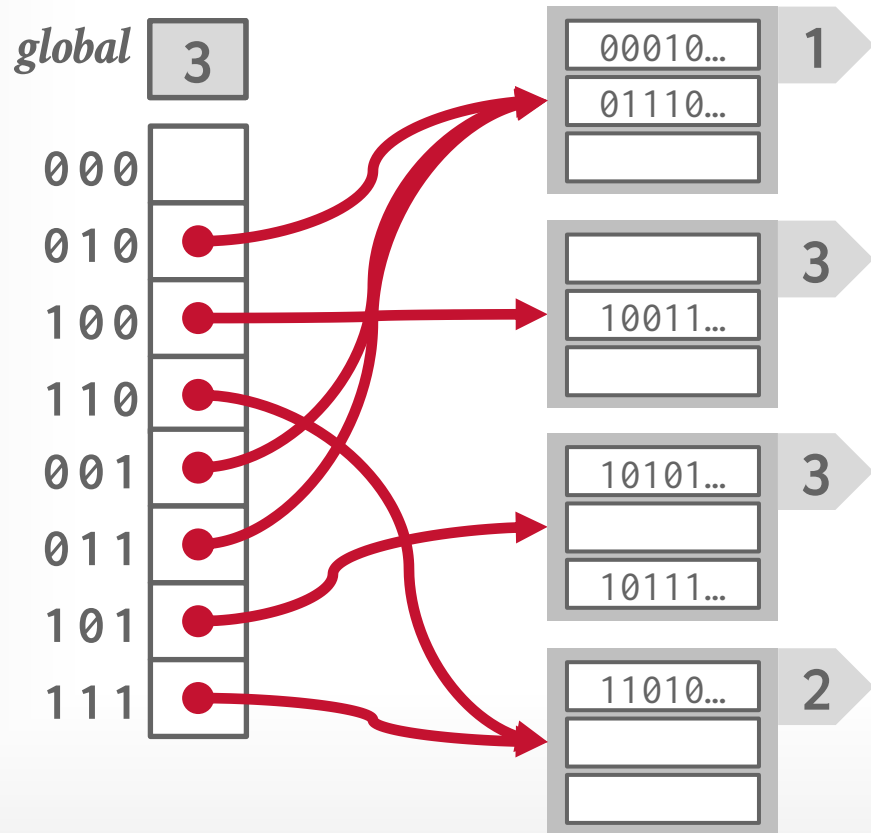
Put B

$hash(B) = 10111...$

Put C

$hash(C) = 10100...$

# EXTENDIBLE HASHING



Get A

*hash(A) = 01110...*

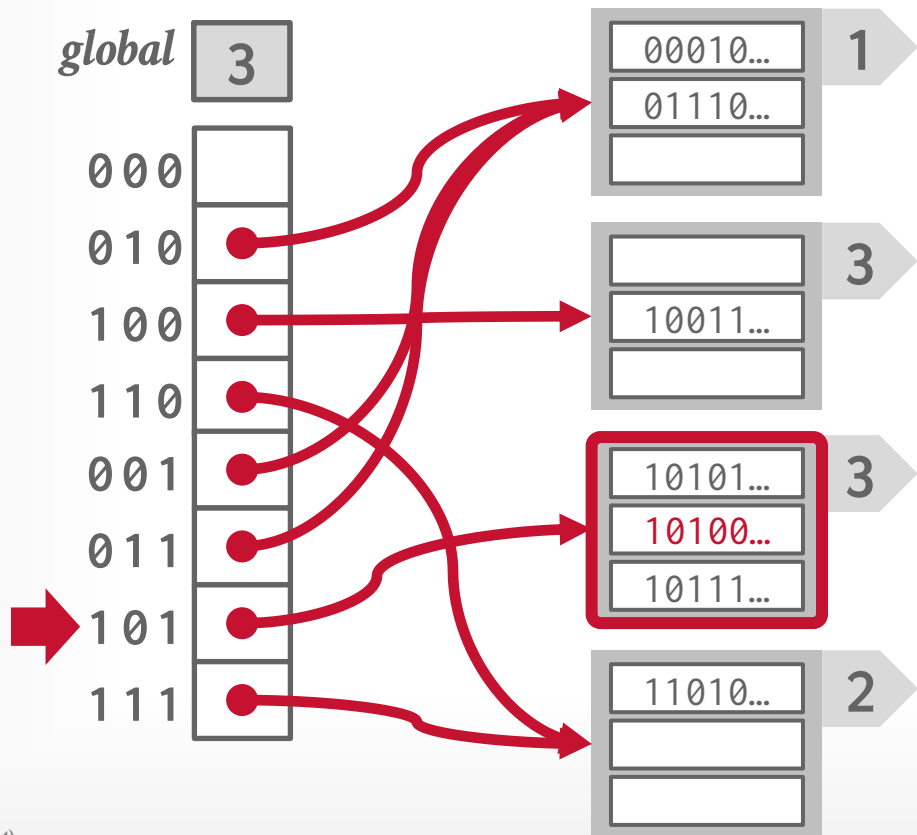
Put B

*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...*

# 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:

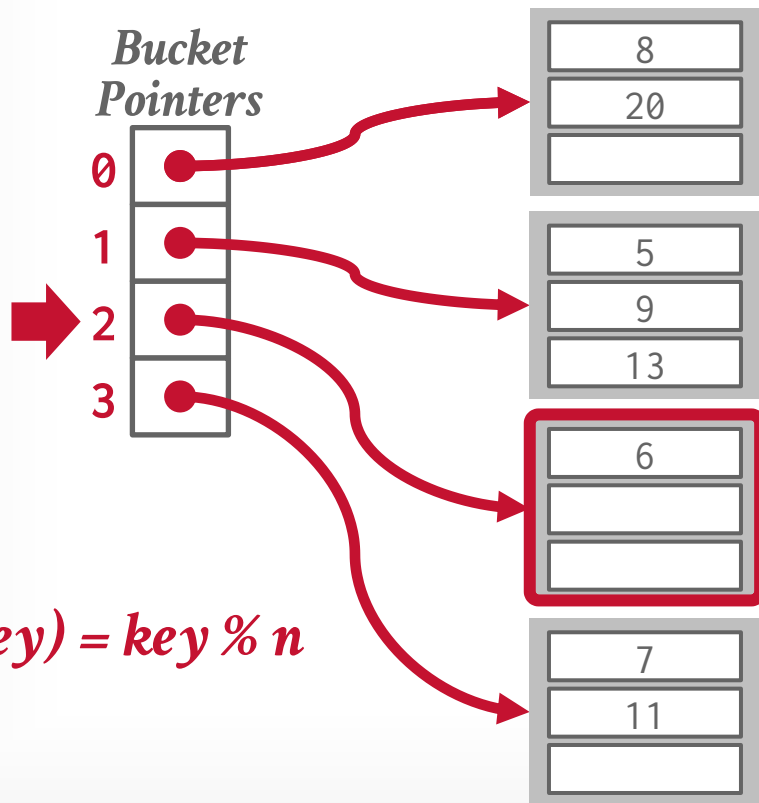
→ Space Utilization

→ Average Length of Overflow Chains



# LINEAR HASHING

Split  
Pointer

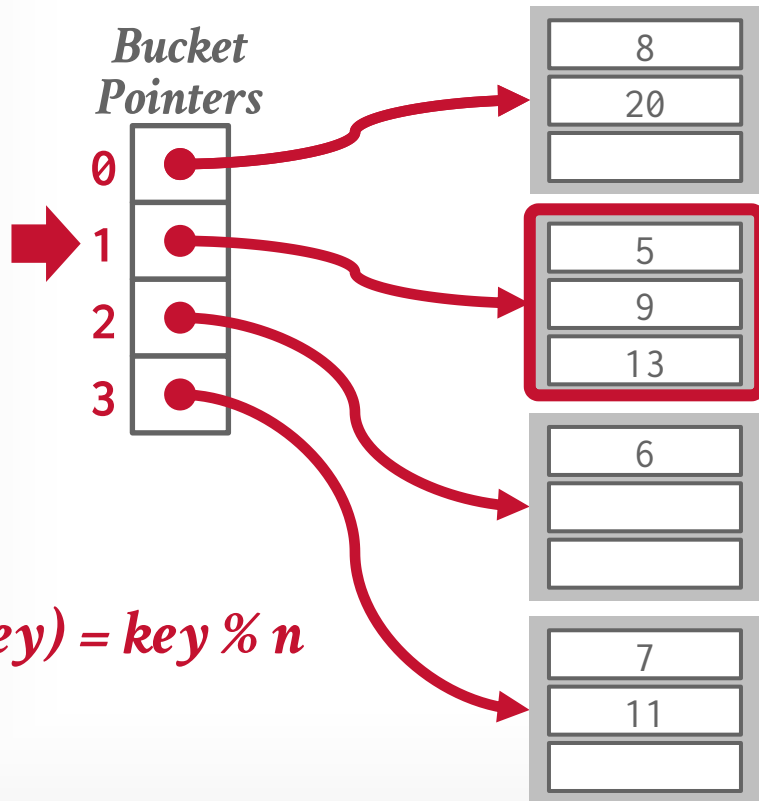


Get 6

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

# LINEAR HASHING

Split  
Pointer



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

Get 6

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

Put 17

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

# LINEAR HASHING

Split  
Pointer



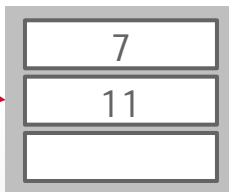
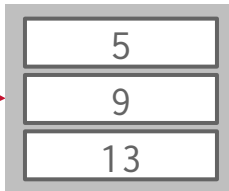
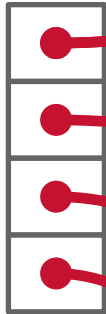
Bucket  
Pointers

0

1

2

3



Overflow!

Get 6

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

Put 17

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

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

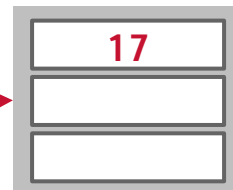
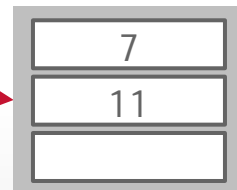
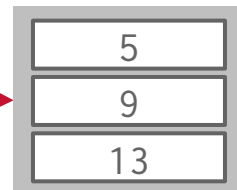
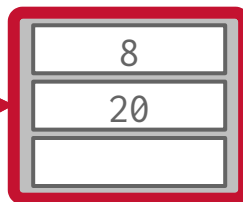
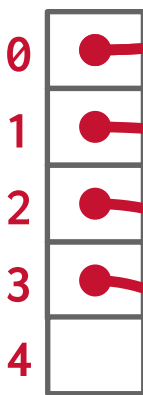


# LINEAR HASHING

Split  
Pointer



Bucket  
Pointers



Overflow!

Get 6

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

Put 17

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

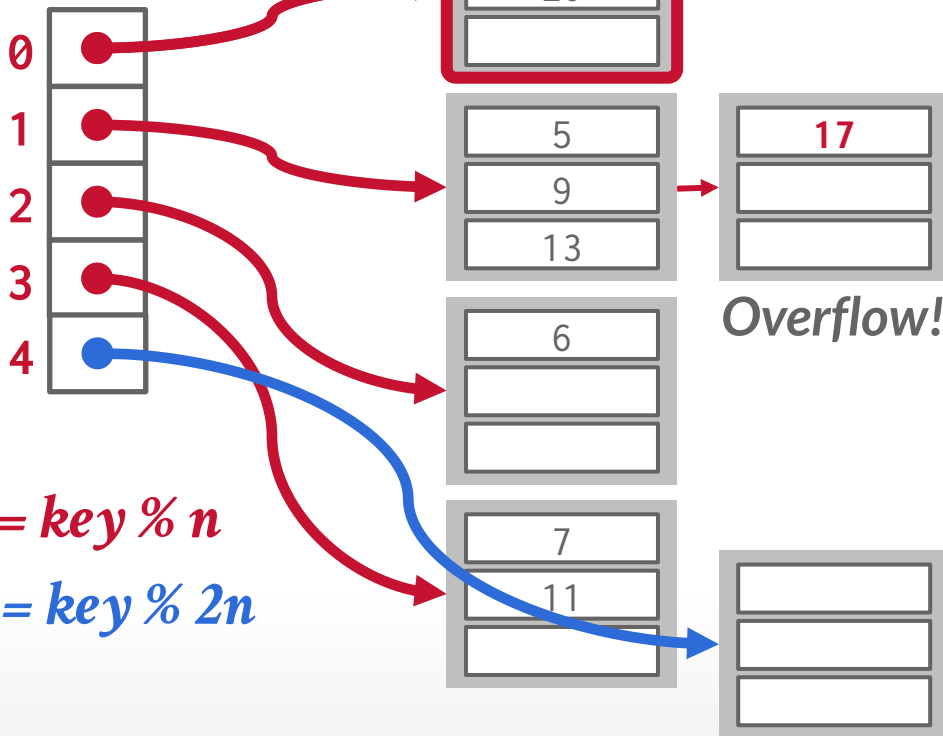
$$\text{hash}_1(\text{key}) = \text{key} \% n$$

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

# LINEAR HASHING

Split  
Pointer  
➔

Bucket  
Pointers



Get 6

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

Put 17

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

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

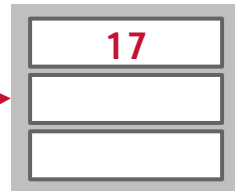
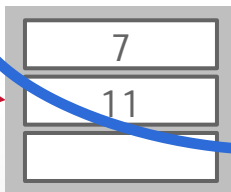
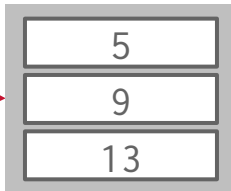
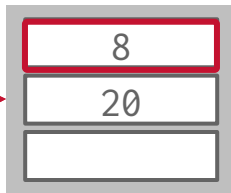
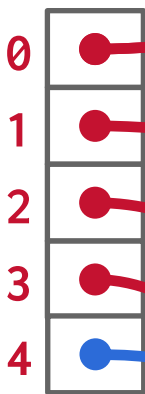
$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING

Split  
Pointer



Bucket  
Pointers



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

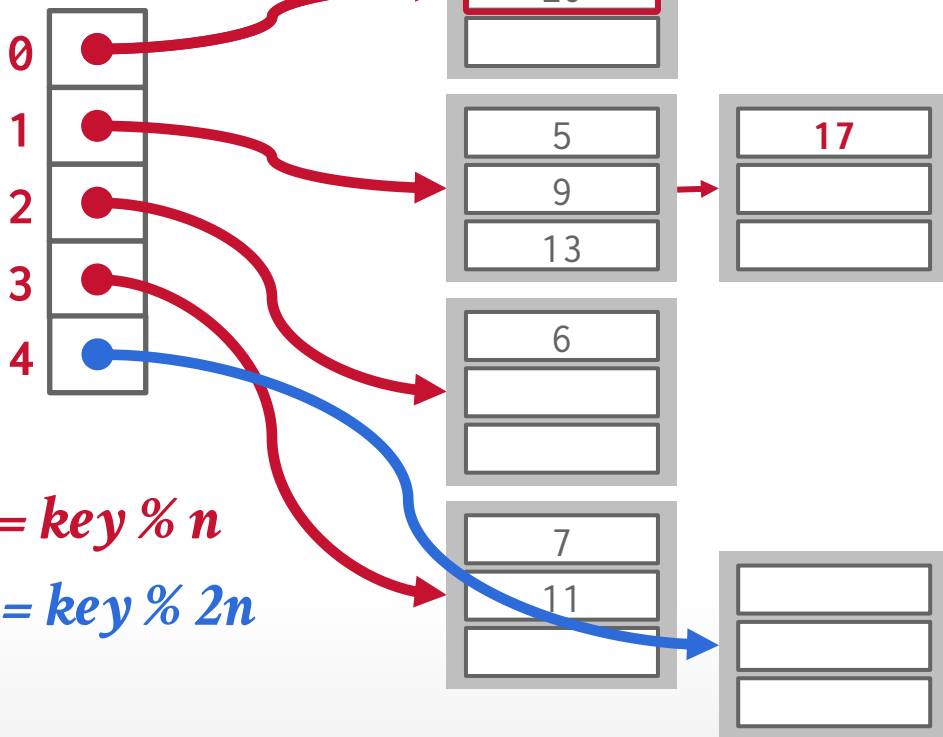
$$\text{hash}_1(\text{key}) = \text{key} \% n$$

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

# LINEAR HASHING

Split  
Pointer  
➔

Bucket  
Pointers



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

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

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

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

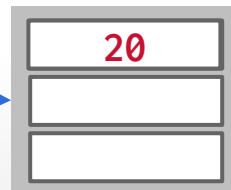
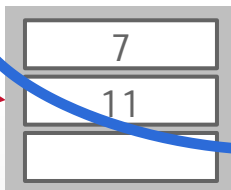
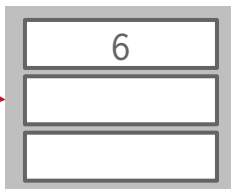
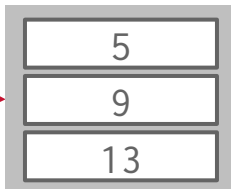
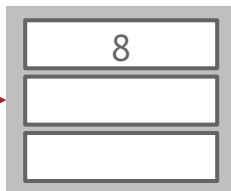
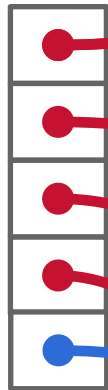
# LINEAR HASHING

Split  
Pointer



Bucket  
Pointers

0  
1  
2  
3  
4



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

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

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

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

# LINEAR HASHING

Split  
Pointer



Bucket  
Pointers

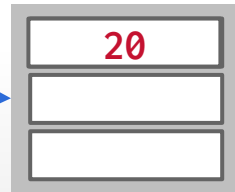
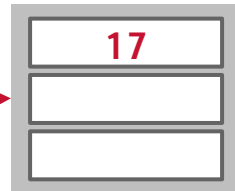
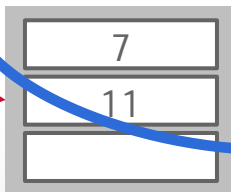
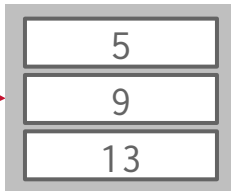
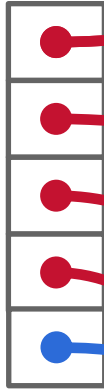
0

1

2

3

4



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

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

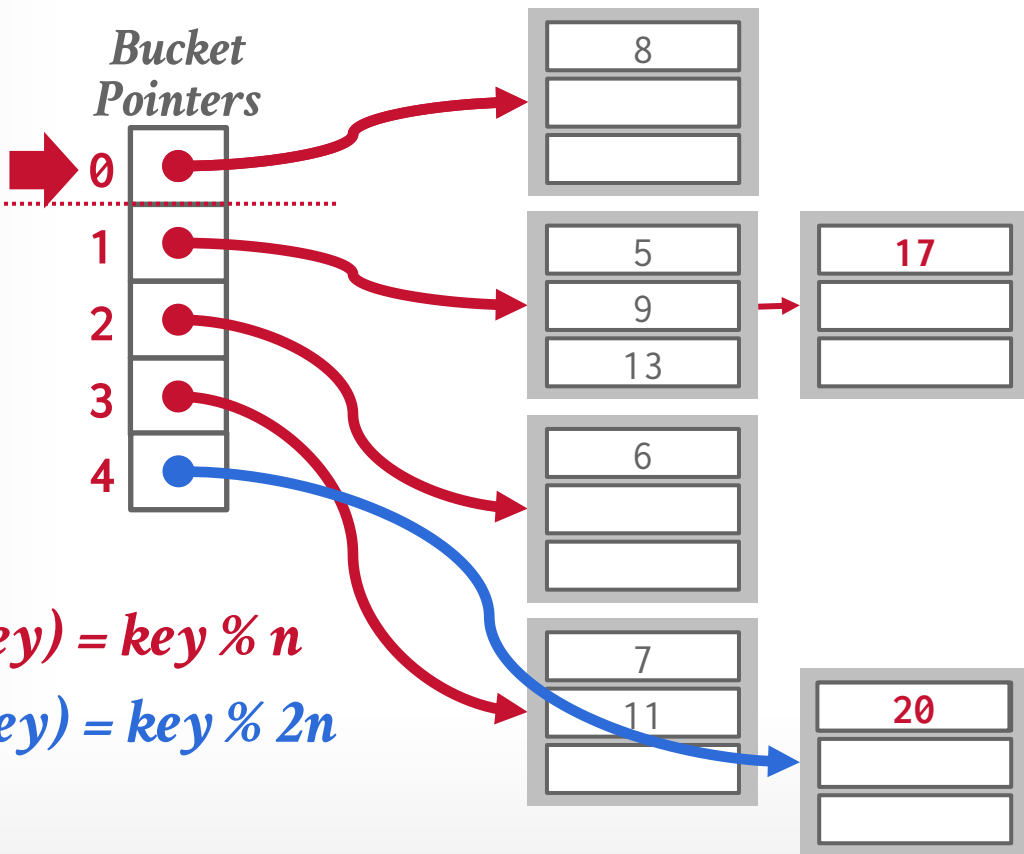
$$\text{hash}_1(\text{key}) = \text{key} \% n$$

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

# LINEAR HASHING

Split  
Pointer

Bucket  
Pointers



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

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

Get 6

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

Put 17

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

$$hash_2(8) = 8 \% 8 = 0$$

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

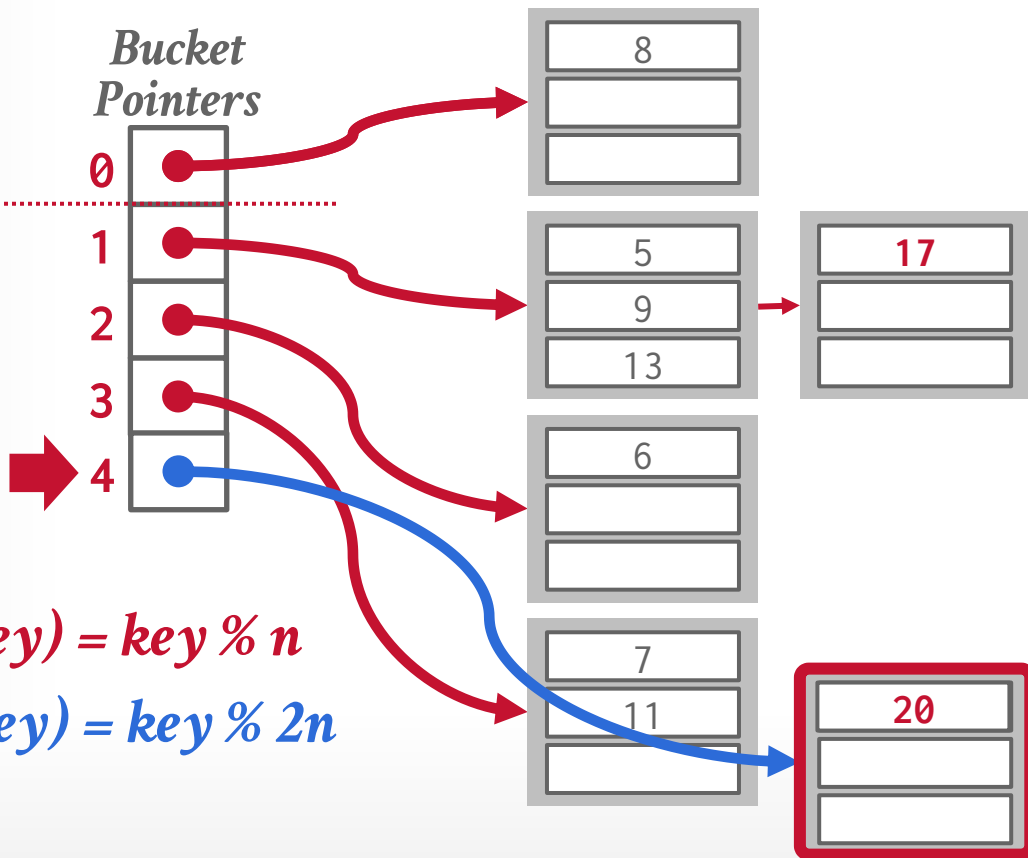
Get 20

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

# LINEAR HASHING

Split  
Pointer

Bucket  
Pointers



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

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

Get 20

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

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

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

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



# LINEAR HASHING

Split  
Pointer

Bucket  
Pointers

0

1

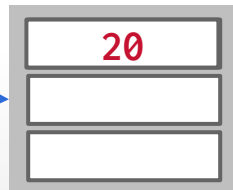
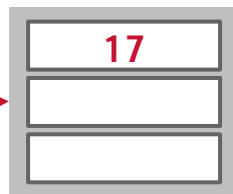
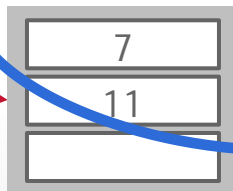
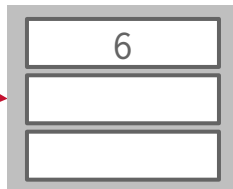
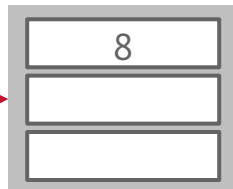
2

3

4

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

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



Get 6

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

Put 17

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

$$\text{hash}_2(8) = 8 \% 8 = 0$$

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

Get 20

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

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

Get 9

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

# LINEAR HASHING – RESIZING

---

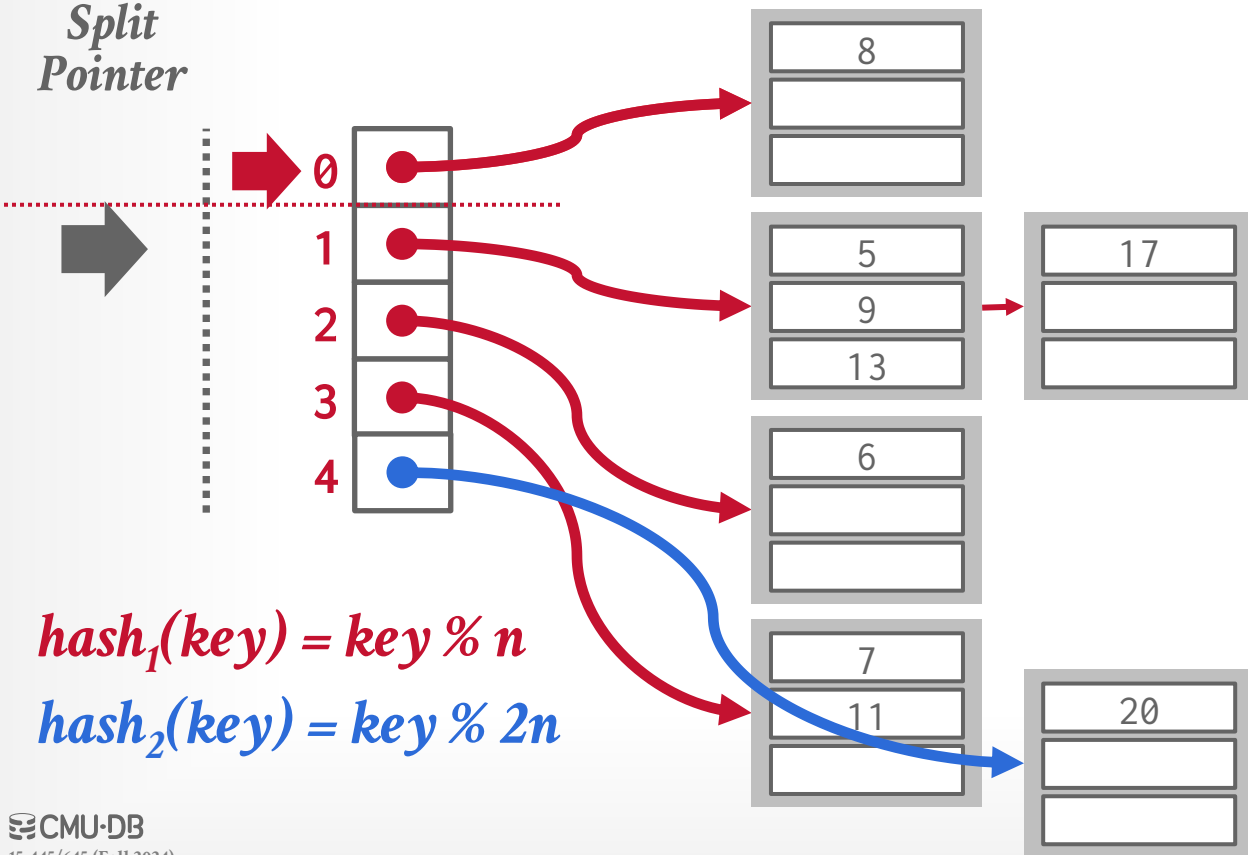
Splitting buckets based on the split pointer will eventually get to all overflowed buckets.

→ 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

Split  
Pointer

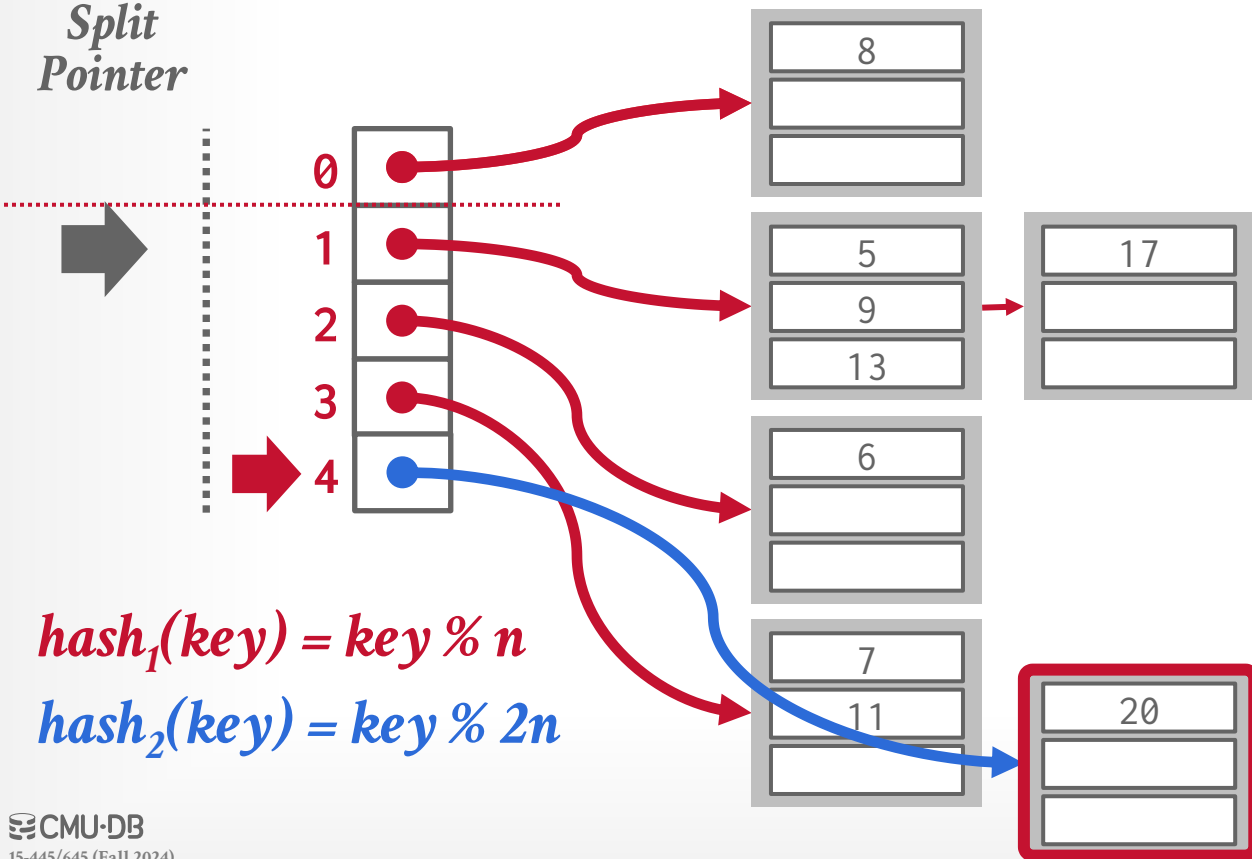


Delete 20

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

# LINEAR HASHING - DELETES

Split  
Pointer



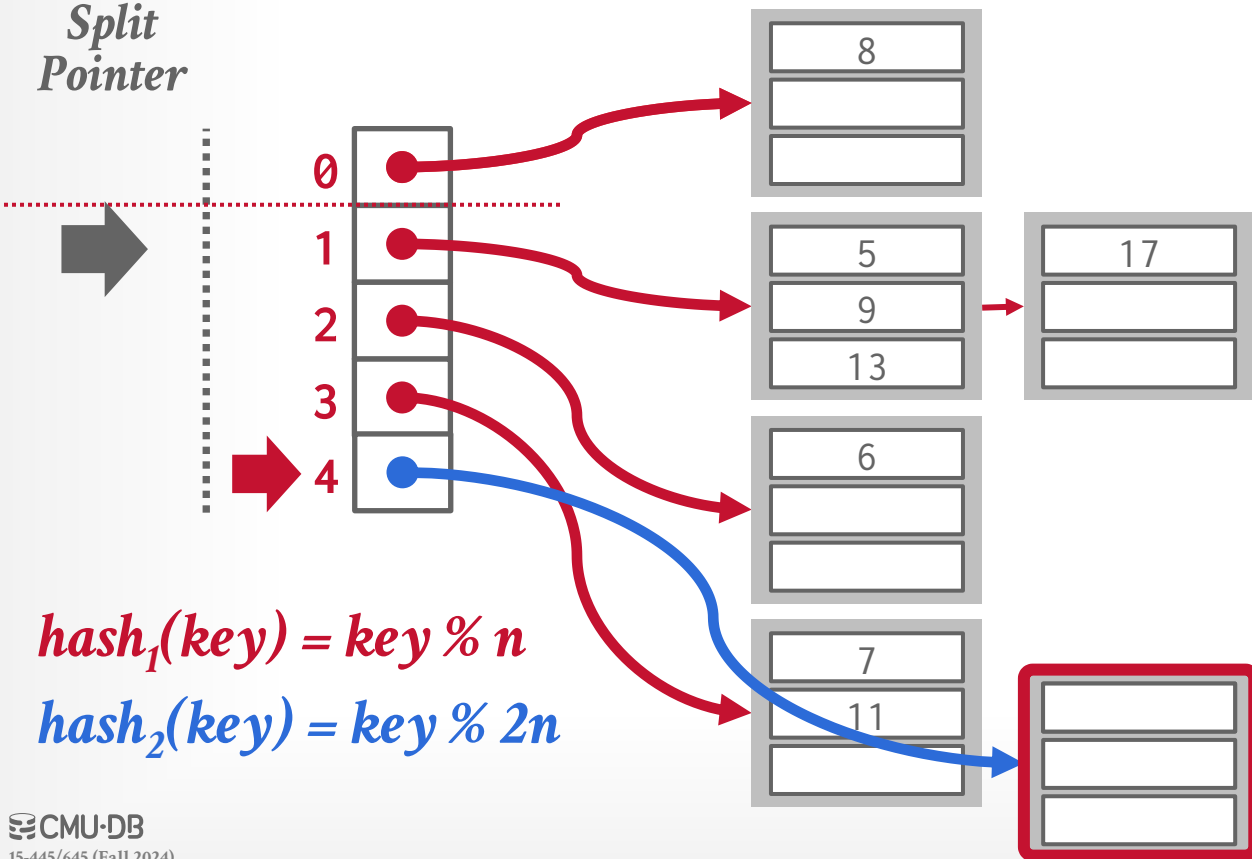
Delete 20

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

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

# LINEAR HASHING - DELETES

Split  
Pointer



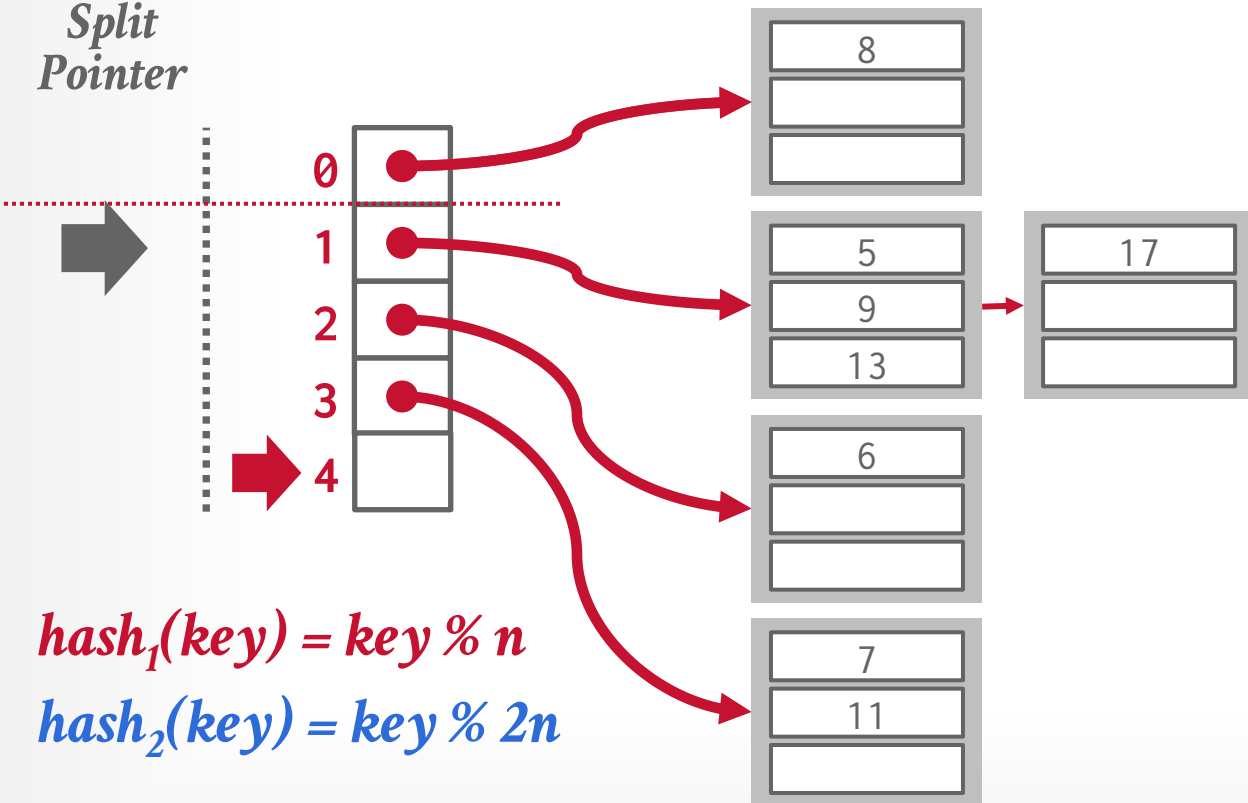
Delete 20

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

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

# LINEAR HASHING - DELETES

Split  
Pointer



Delete 20

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

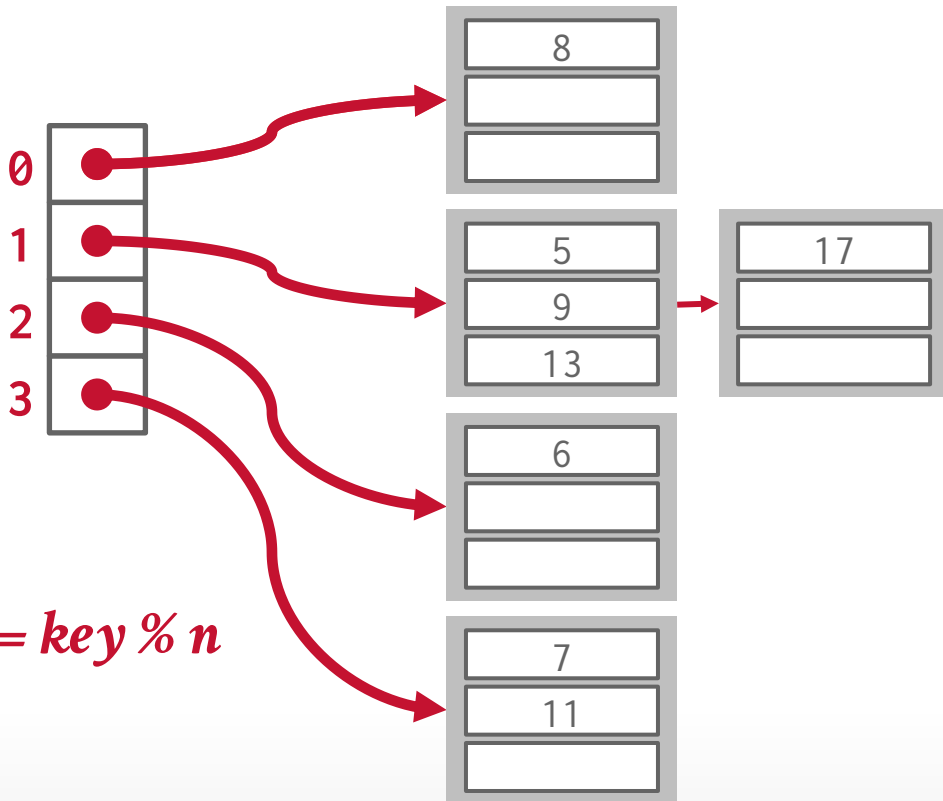
$$\text{hash}_2(20) = 20 \% 8 = 4$$

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

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

# LINEAR HASHING - DELETES

Split  
Pointer



Delete 20

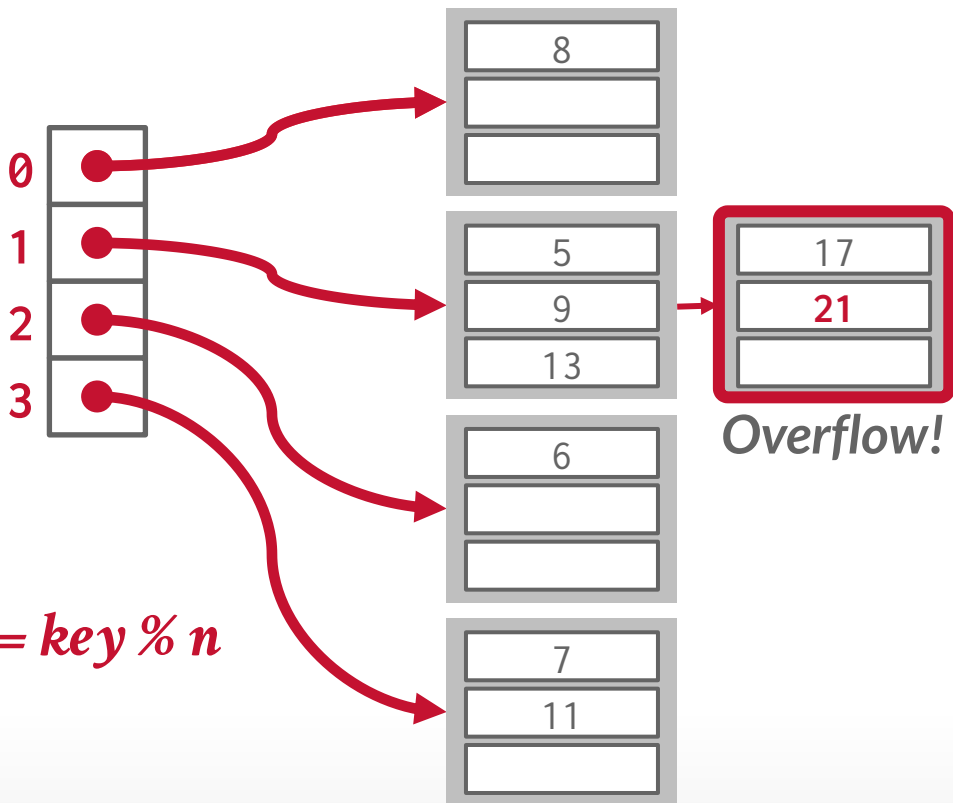
$$\text{hash}_1(20) = 20 \% 4 = 0$$

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

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

# LINEAR HASHING - DELETES

Split  
Pointer



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

Delete 20

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

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

Put 21

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

Overflow!



# 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...

# 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...

PostgreSQL



```
CREATE INDEX ON xxx (val);
```

```
CREATE INDEX ON xxx USING BTREE (val);
```

```
CREATE INDEX ON xxx USING HASH (val);
```



# NEXT CLASS

---

## Order-Preserving Indexes ft. B+Trees

→ aka "The Greatest Data Structure of All Time"