

# COURSE STATUS

---

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:

- Hash Tables
- Trees

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

# DESIGN DECISIONS

---

## Data Organization

→ How we layout data structure in memory/pages and what information to store to support efficient access.

## Concurrency

→ How to enable multiple threads to access the data structure at the same time without causing problems.

# HASH TABLES

---

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

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

Space Complexity:  $O(n)$

Time Complexity:

- Average:  $O(1)$  ← *Databases need to care about constants!*
- Worst:  $O(n)$

# STATIC HASH TABLE

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

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

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

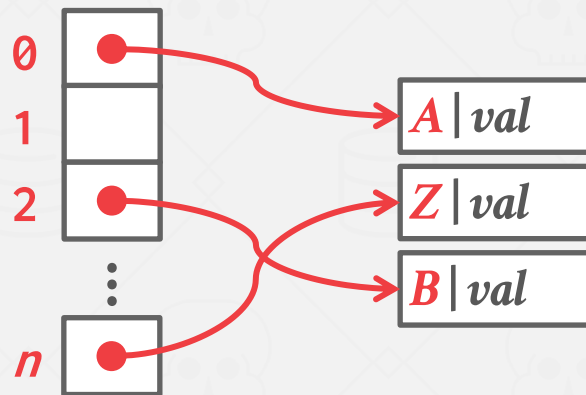
|   |   |
|---|---|
| 0 | A |
| 1 | Ø |
| 2 | B |
| ⋮ |   |
| n | Z |

# 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.

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



# ASSUMPTIONS

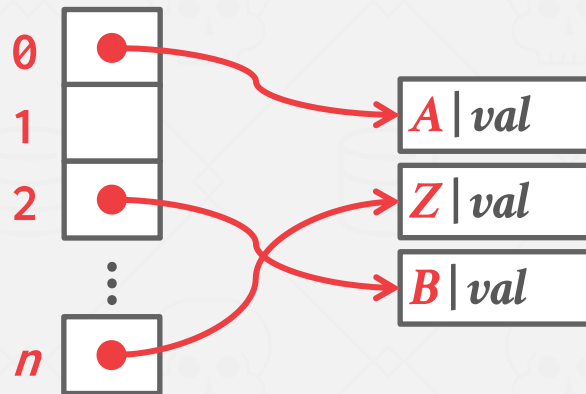
**Assumption#1: Number of elements is known ahead of time and fixed.**

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

**Assumption #3: Perfect hash function.**

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

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



# 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.

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

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



# HASH FUNCTIONS

---

## CRC-64 (1975)

→ Used in networking for error detection.

## MurmurHash (2008)

→ Designed as a fast, general-purpose hash function.

## Google CityHash (2011)

→ Designed to be faster for short keys (<64 bytes).

## Facebook XXHash (2012)

→ From the creator of zstd compression.

← State-of-the-art

## Google FarmHash (2014)

→ Newer version of CityHash with better collision rates.

# STATIC HASHING SCHEMES

---

**Approach #1: Linear Probe Hashing**

**Approach #2: Robin Hood Hashing**

**Approach #3: Cuckoo Hashing**

# LINEAR PROBE HASHING

---

Single giant table of slots.

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

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

# LINEAR PROBE HASHING

$hash(key) \% N$

A

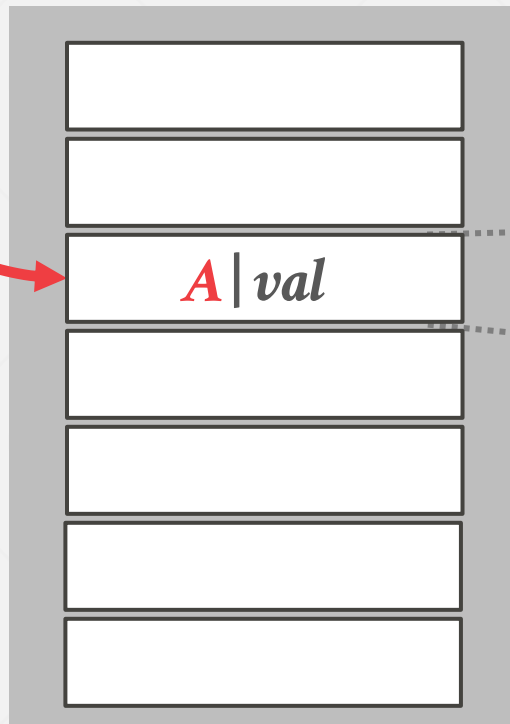
B

C

D

E

F



**<key> | <value>**

# LINEAR PROBE HASHING

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

A

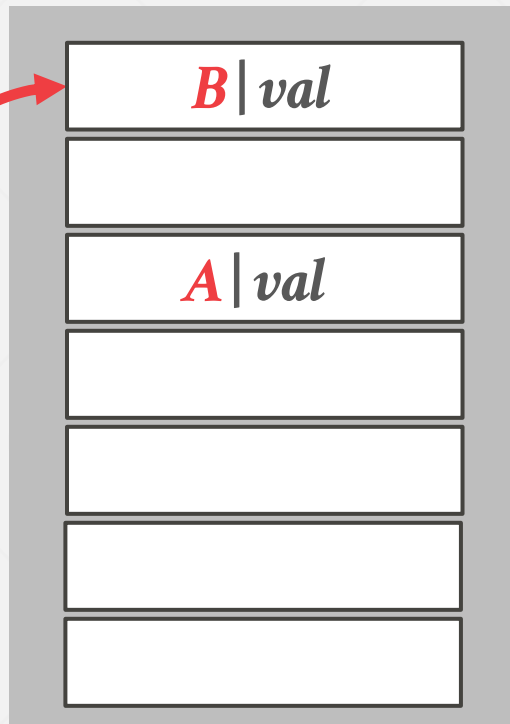
B

C

D

E

F



# LINEAR PROBE HASHING

$hash(key) \% N$

A

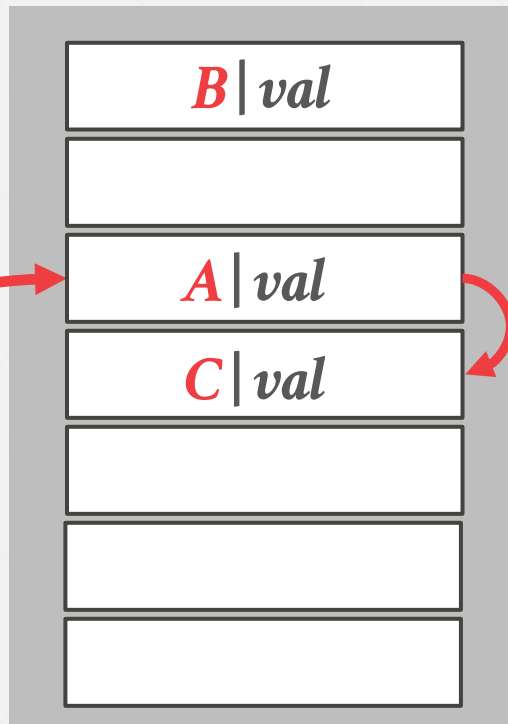
B

C

D

E

F



# LINEAR PROBE HASHING

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

A

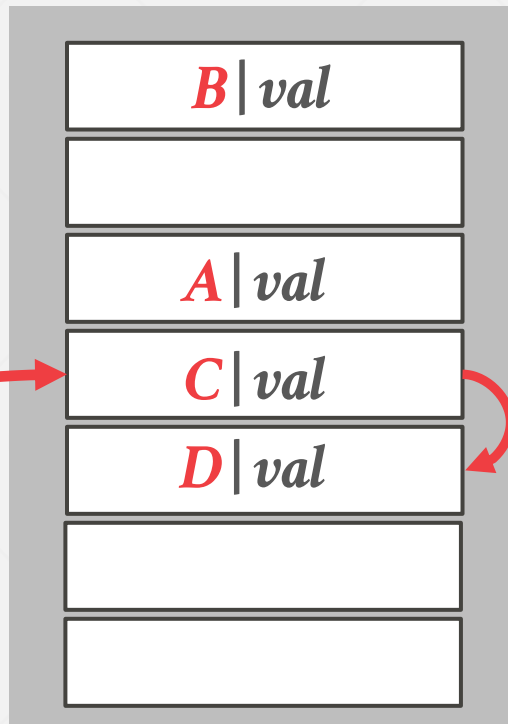
B

C

D

E

F



# LINEAR PROBE HASHING

$hash(key) \% N$

A

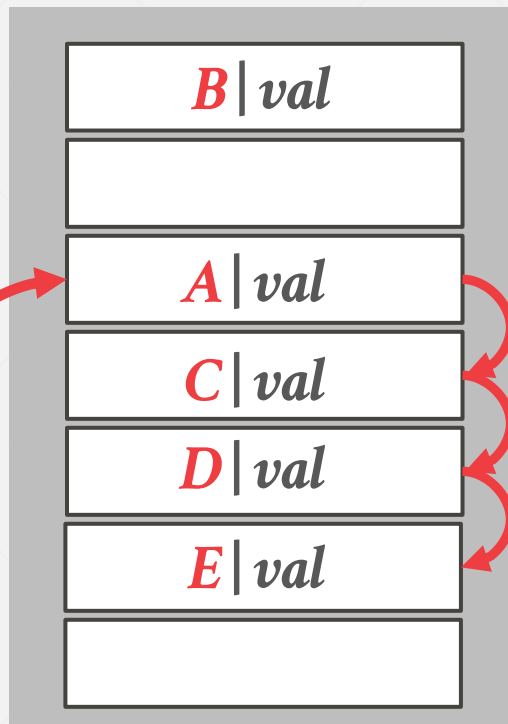
B

C

D

E

F





# LINEAR PROBE HASHING

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

A

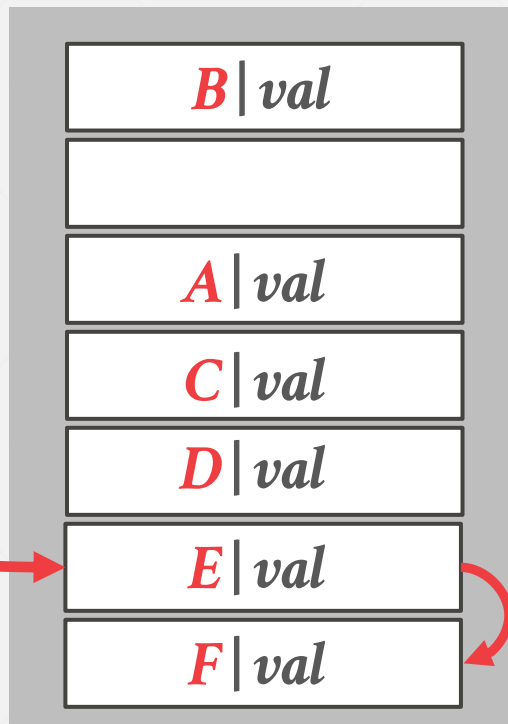
B

C

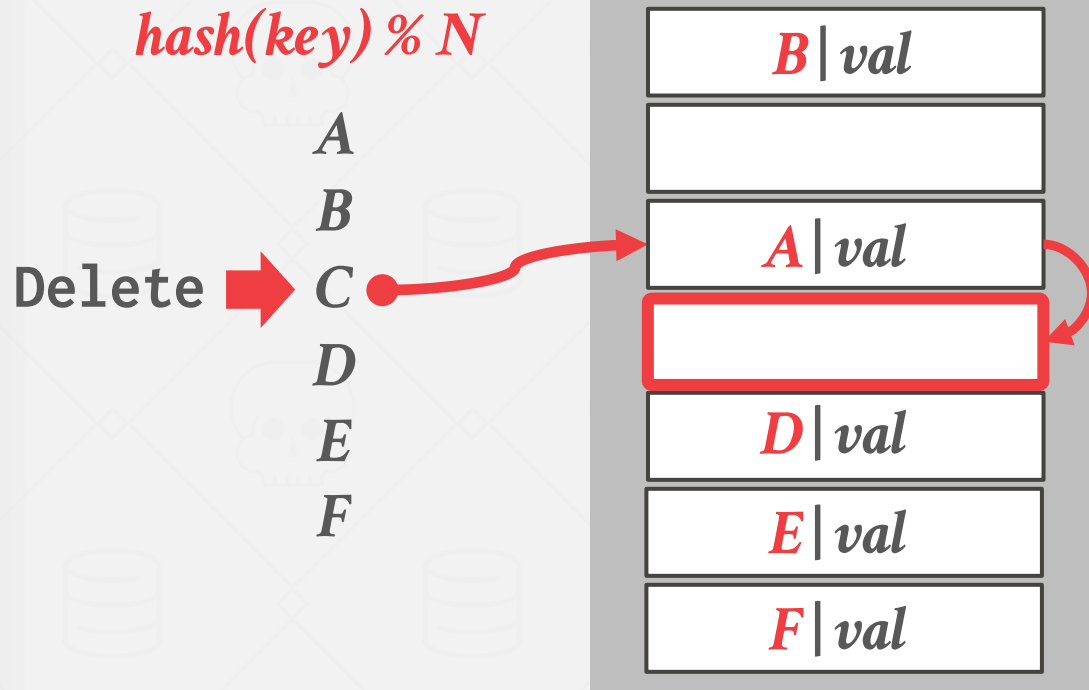
D

E

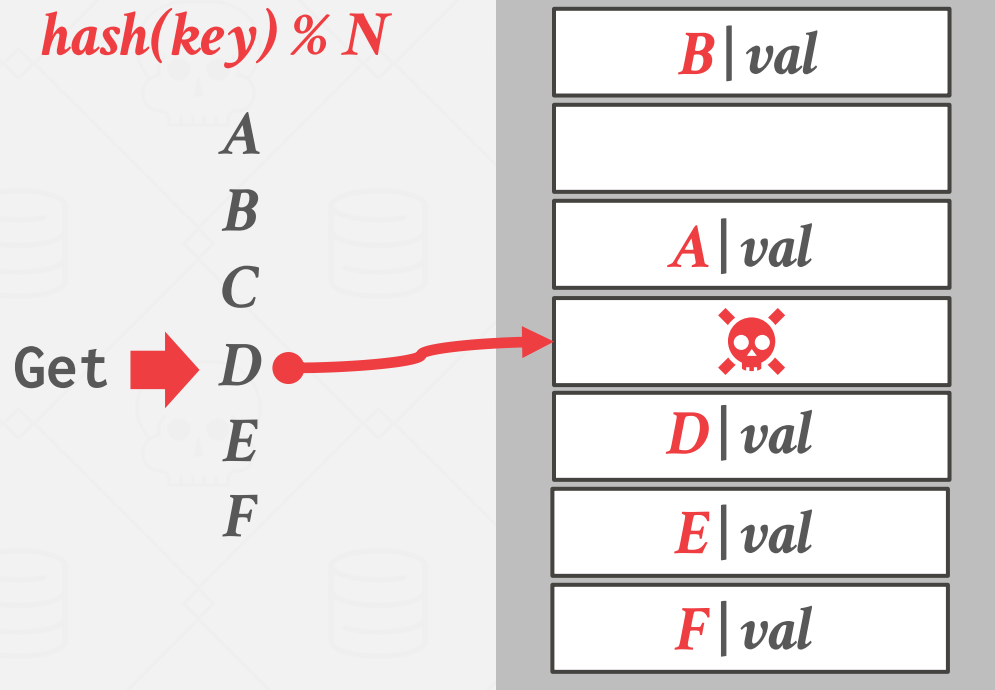
F



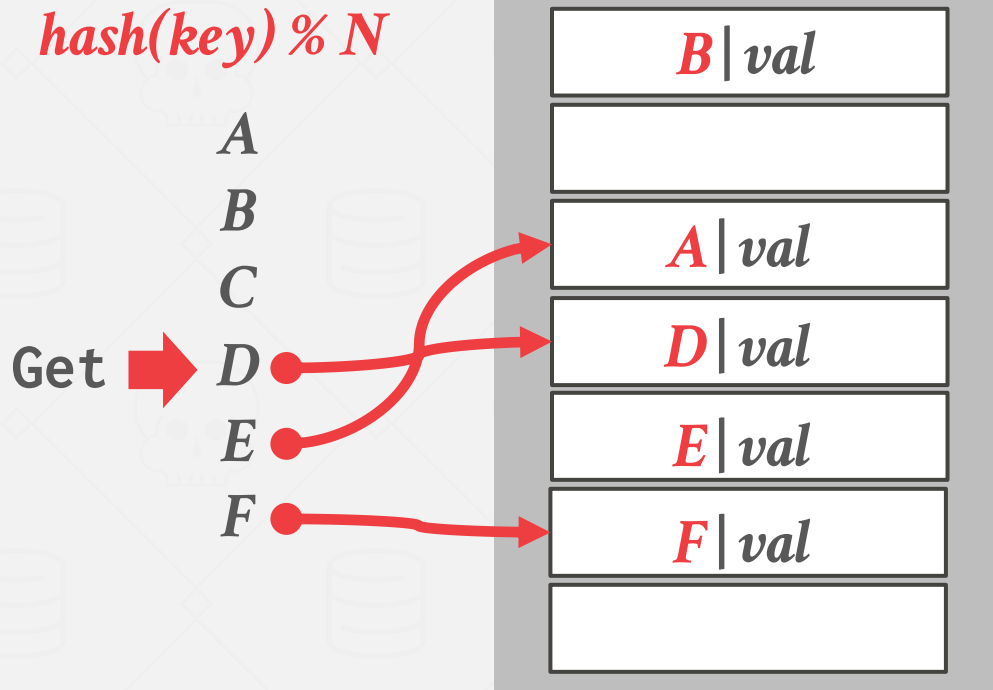
# LINEAR PROBE HASHING - DELETES



# LINEAR PROBE HASHING - DELETES



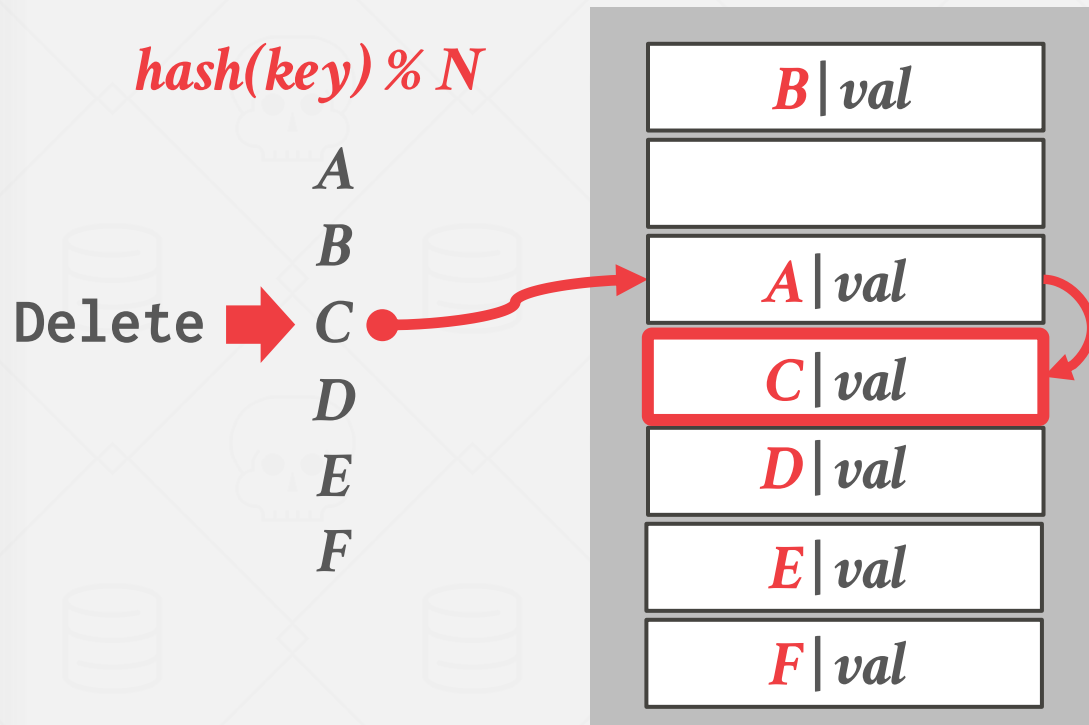
# LINEAR PROBE HASHING - DELETES



## Approach #1: Movement

- Rehash keys until you find the first empty slot.
- Nobody actually does this.

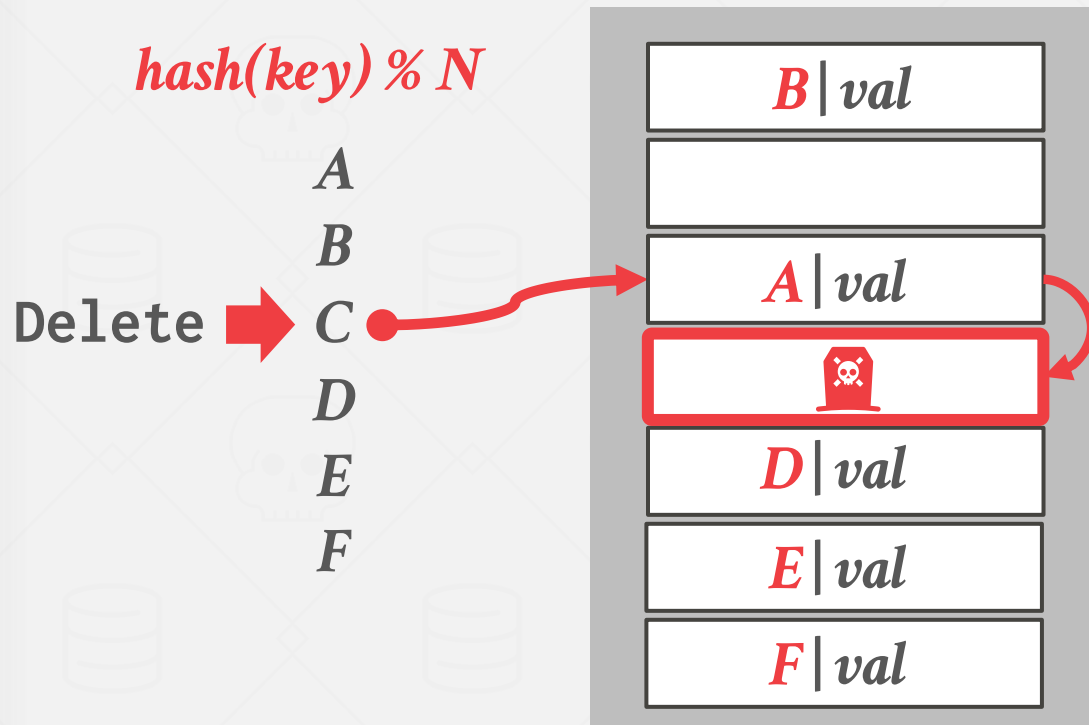
# LINEAR PROBE HASHING - DELETES



## Approach #2: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted.
- You can reuse the slot for new keys.
- May need periodic garbage collection.

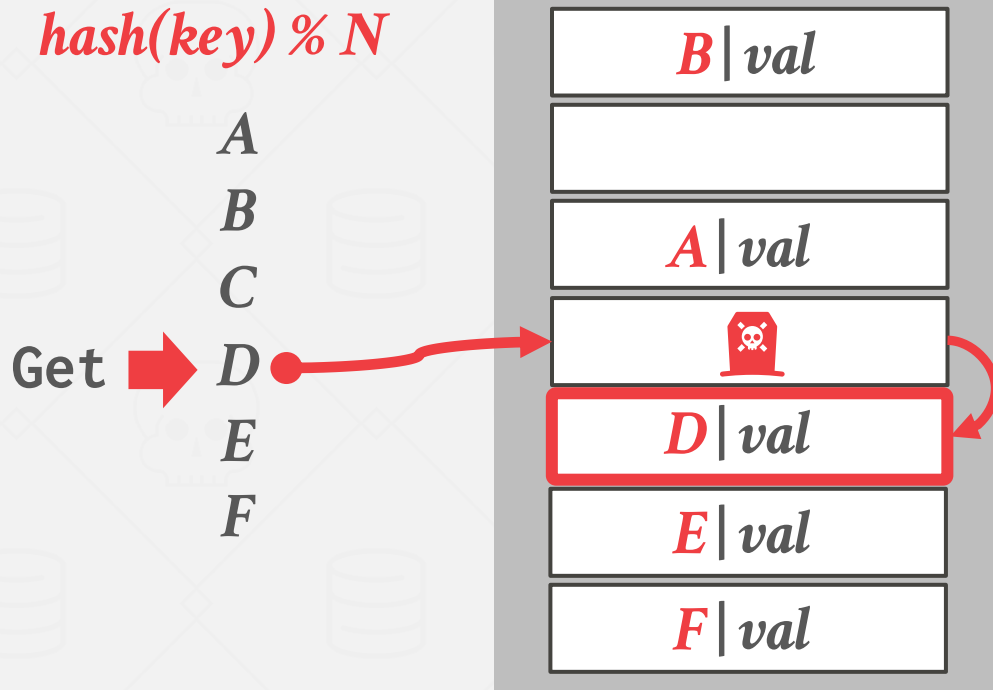
# LINEAR PROBE HASHING - DELETES



## Approach #2: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted.
- You can reuse the slot for new keys.
- May need periodic garbage collection.

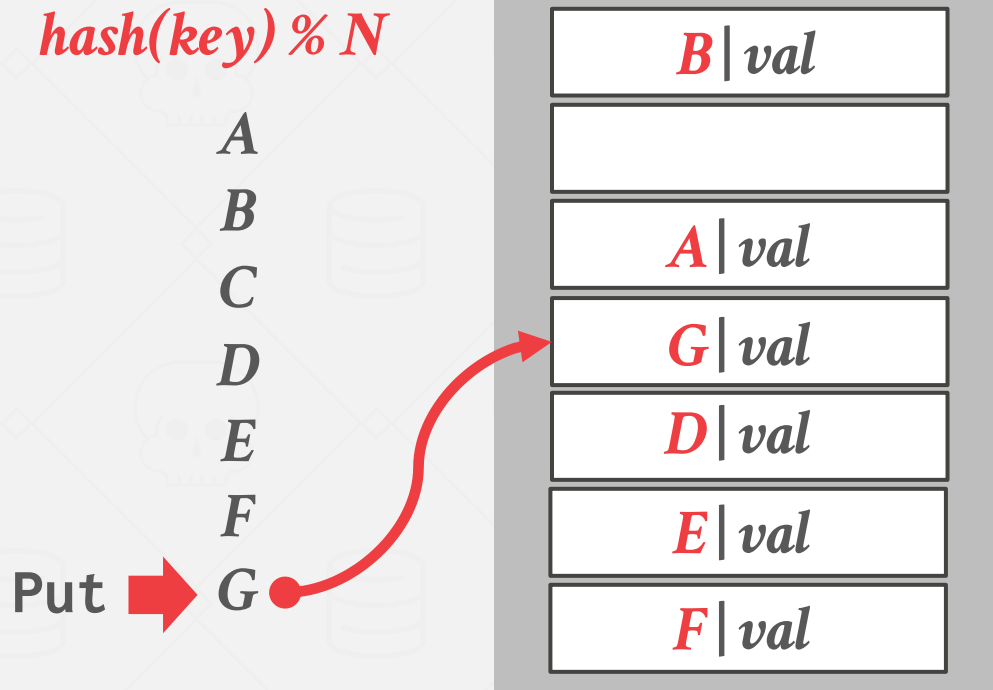
# LINEAR PROBE HASHING - DELETES



## Approach #2: Tombstone

- Set a marker to indicate that the entry in the slot is logically deleted.
- You can reuse the slot for new keys.
- May need periodic garbage collection.

# LINEAR PROBE HASHING - DELETES



## Approach #2: Tombstone

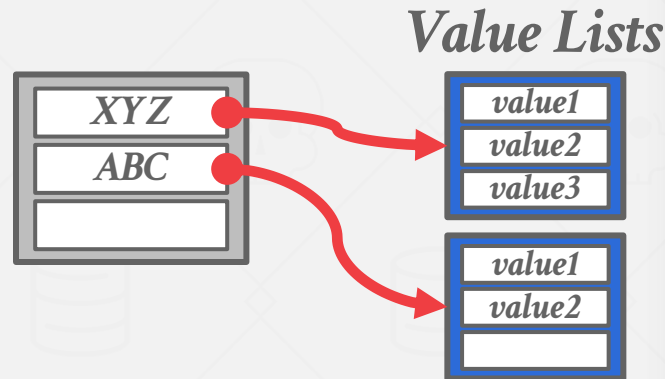
- Set a marker to indicate that the entry in the slot is logically deleted.
- You can reuse the slot for new keys.
- May need periodic garbage collection.



# NON-UNIQUE KEYS

## Choice #1: Separate Linked List

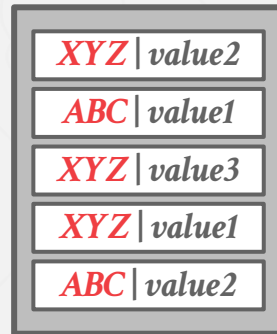
→ Store values in separate storage area for each key.



## Choice #2: Redundant Keys

→ Store duplicate keys entries together in the hash table.

→ This is easier to implement so this is what most systems do.



# ROBIN HOOD HASHING

---

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

- Each key tracks the number of positions they are from where its optimal position in the table.
- On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

# ROBIN HOOD HASHING

$hash(key) \% N$

A

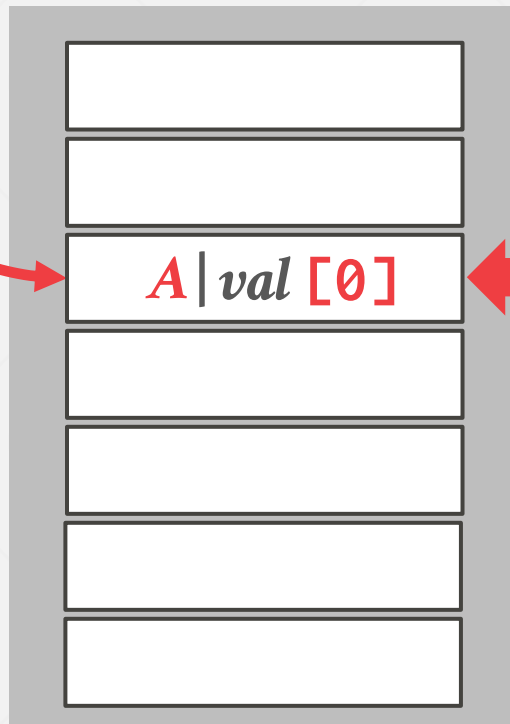
B

C

D

E

F



*# of "Jumps" From First Position*

# ROBIN HOOD HASHING

$hash(key) \% N$

A

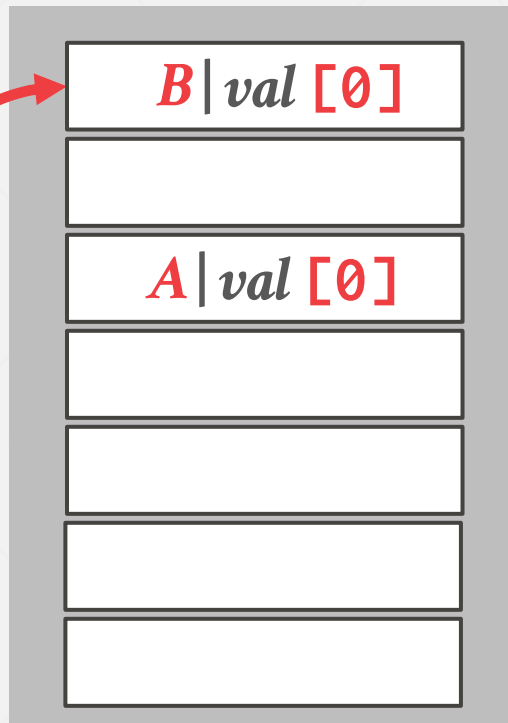
B

C

D

E

F



# ROBIN HOOD HASHING

$hash(key) \% N$

A

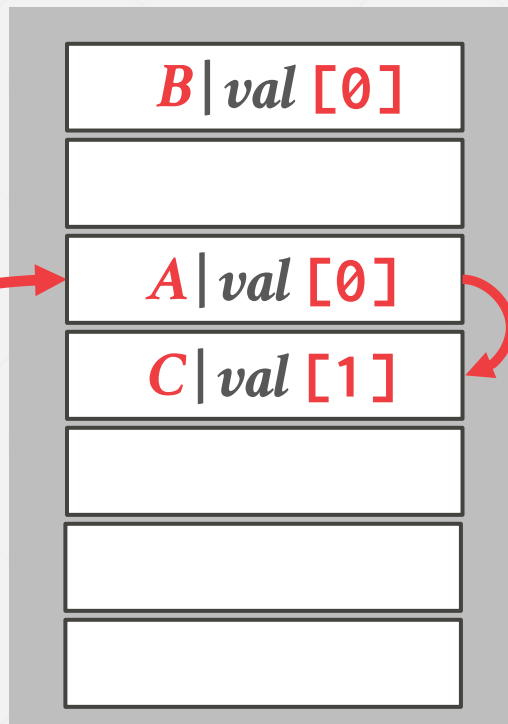
B

C

D

E

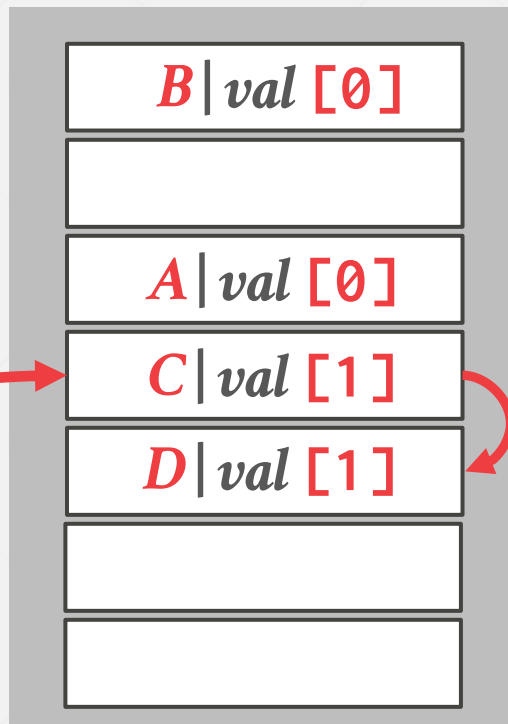
F



# ROBIN HOOD HASHING

$hash(key) \% N$

A  
B  
C  
D  
E  
F

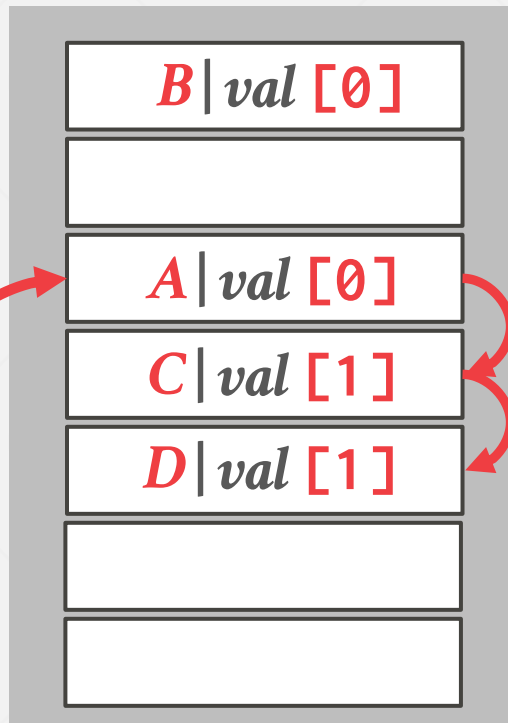


$C[1] > D[0]$

# ROBIN HOOD HASHING

$hash(key) \% N$

A  
B  
C  
D  
E  
F



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

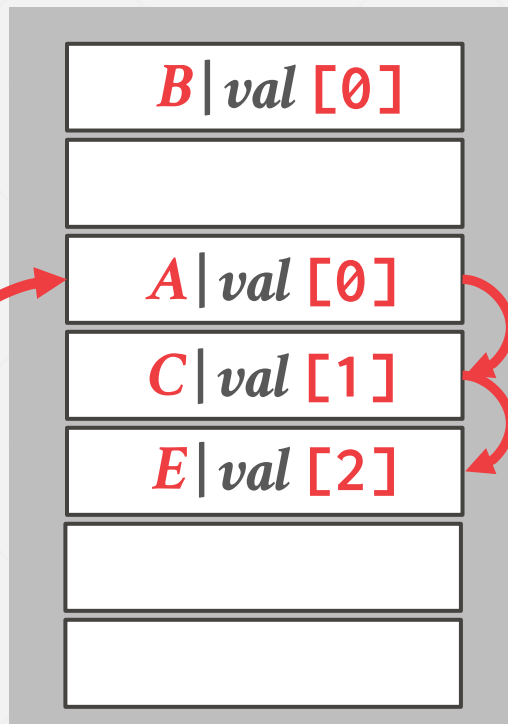
$C[1] == E[1]$

$D[1] < E[2]$

# ROBIN HOOD HASHING

$hash(key) \% N$

A  
B  
C  
D  
E  
F



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

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

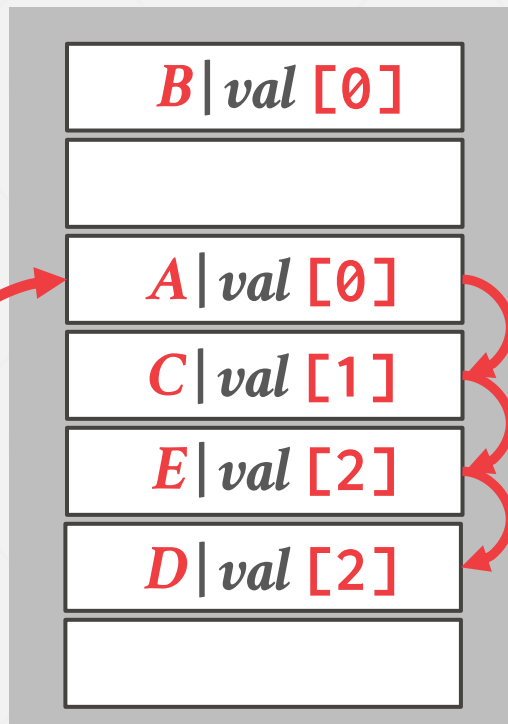
$D[1] < E[2]$



# ROBIN HOOD HASHING

$hash(key) \% N$

A  
B  
C  
D  
E  
F



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

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

$D[1] < E[2]$

# ROBIN HOOD HASHING

$hash(key) \% N$

A

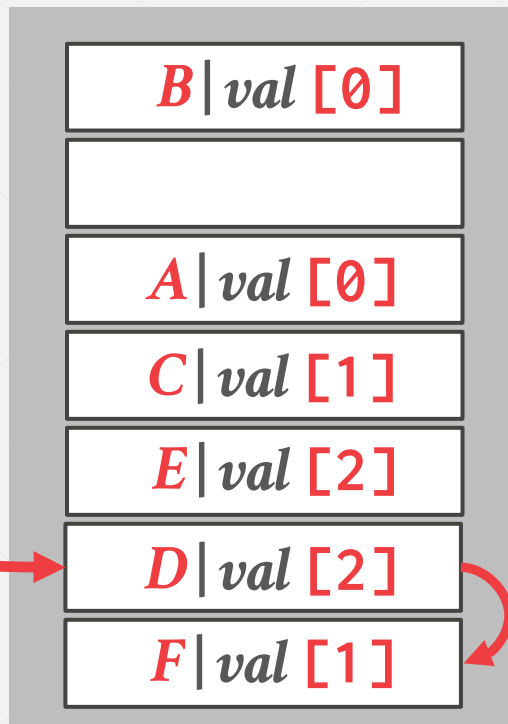
B

C

D

E

F



$D[2] > F[0]$

# CUCKOO HASHING

---

Use multiple hash tables with different hash function seeds.

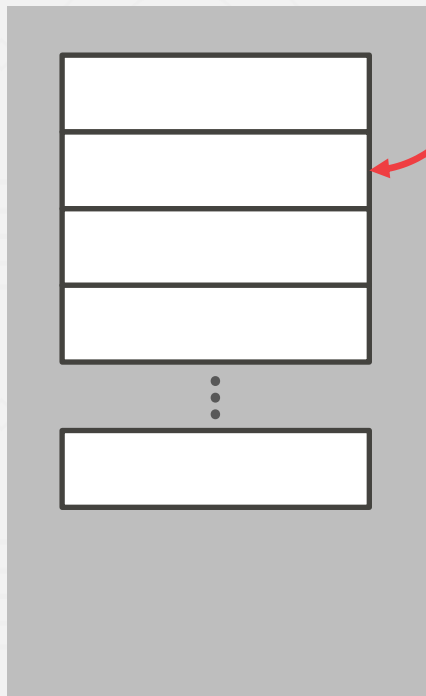
- On insert, check every table and pick anyone that has a free slot.
- If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

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

Best open-source implementation is from CMU.

# CUCKOO HASHING

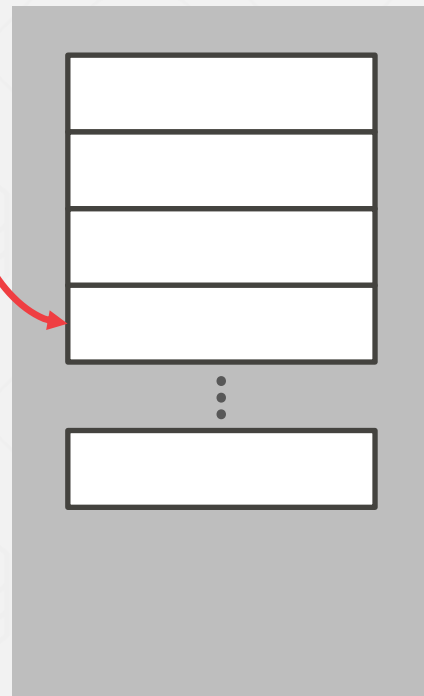
*Hash Table #1*



Put A

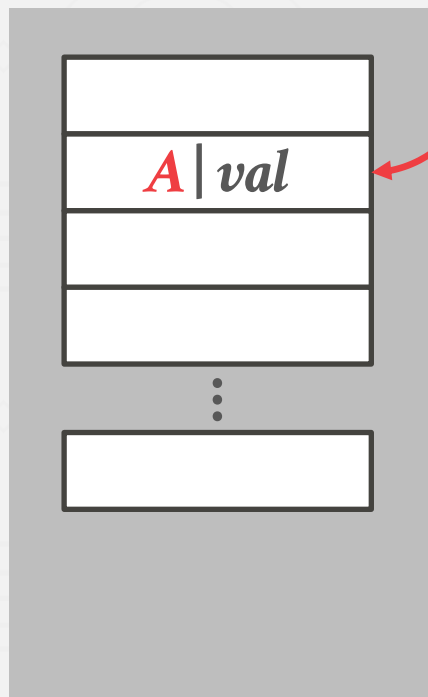
$hash_1(A)$   $hash_2(A)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*

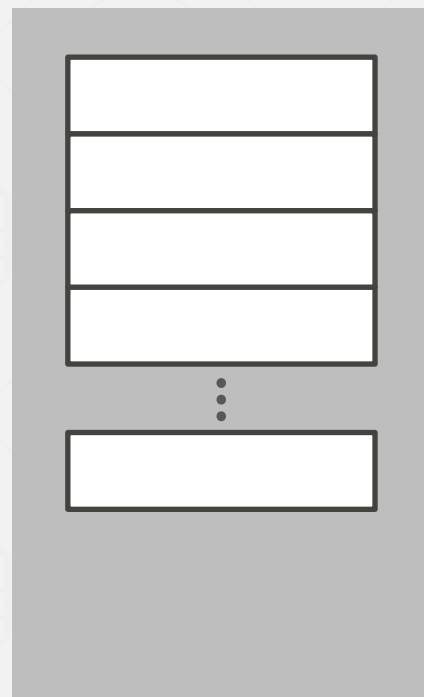


Put A

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

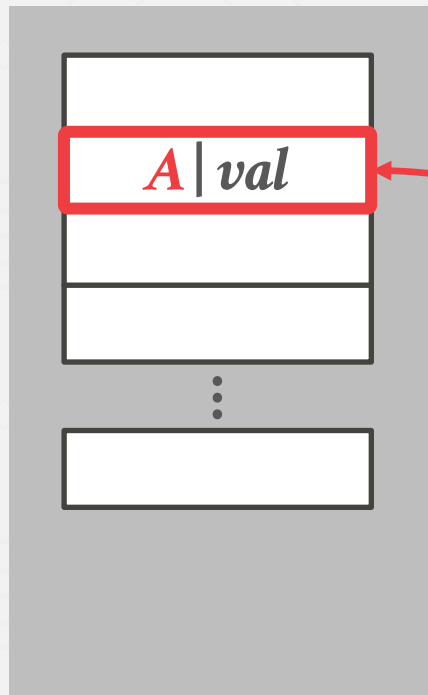


*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



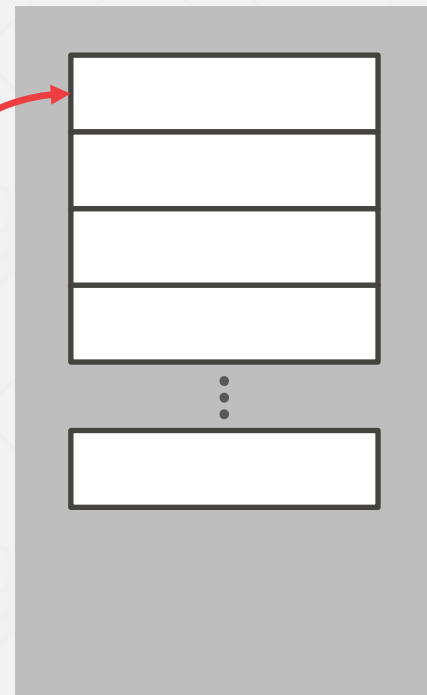
Put A

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

Put B

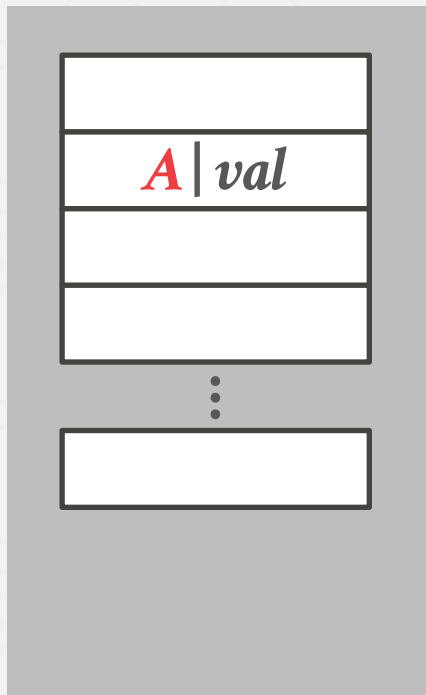
$hash_1(B)$   $hash_2(B)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



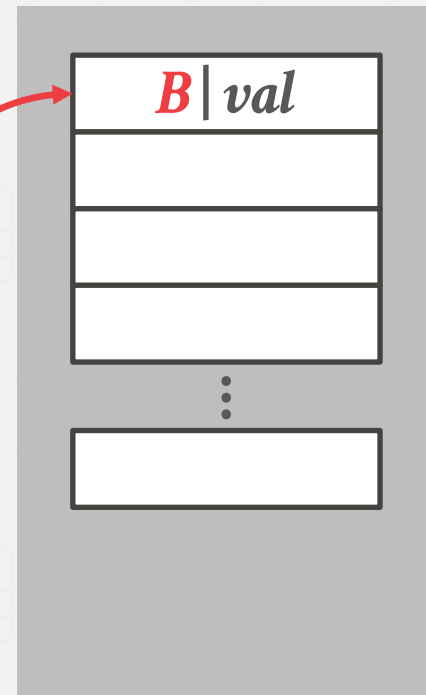
Put A

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

Put B

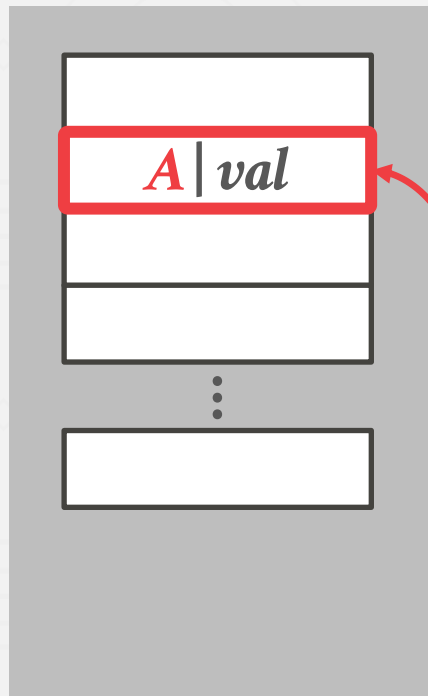
$hash_1(B)$   $hash_2(B)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

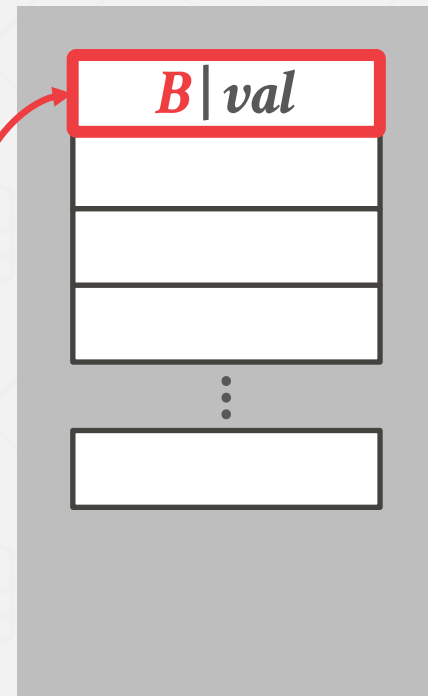
Put B

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

Put C

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

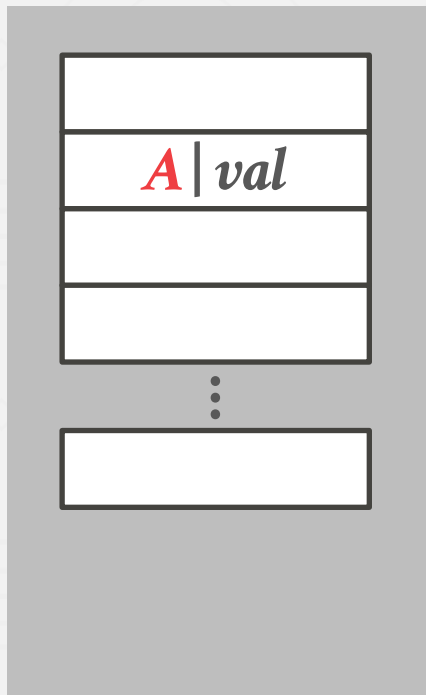
*Hash Table #2*





# CUCKOO HASHING

*Hash Table #1*



Put A

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

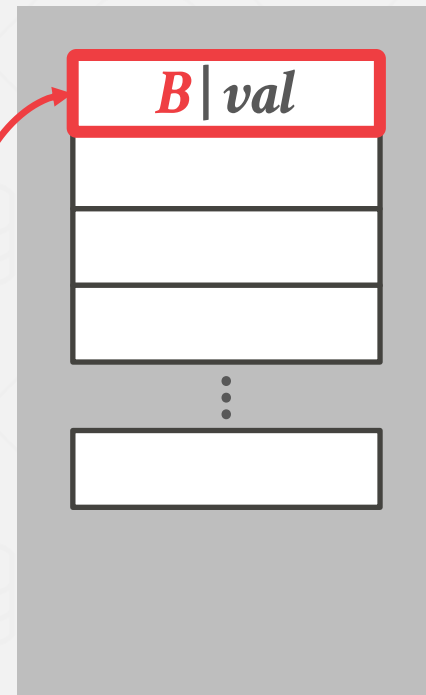
Put B

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

Put C

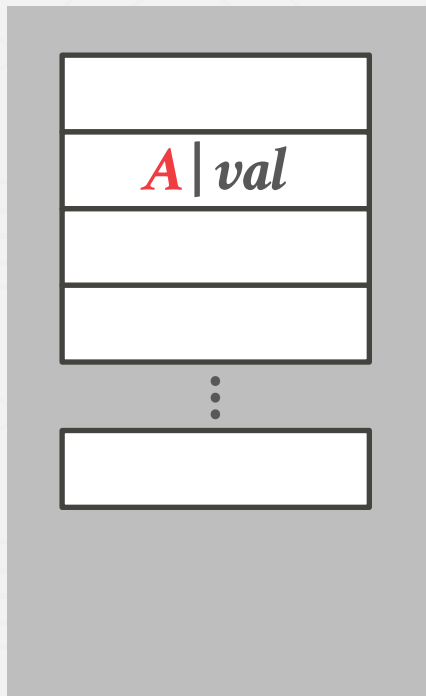
$hash_1(C)$   $hash_2(C)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

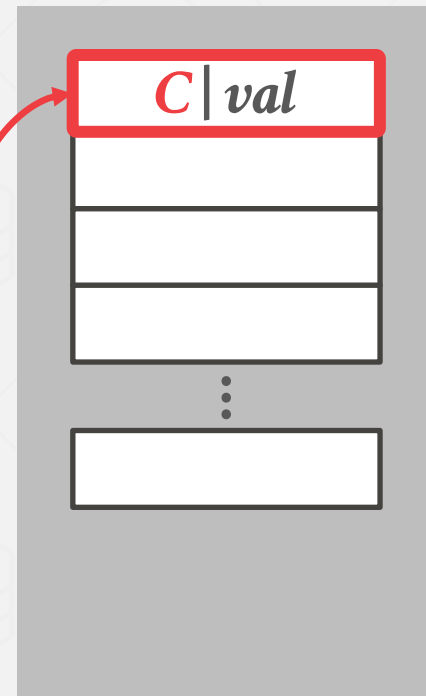
Put B

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

Put C

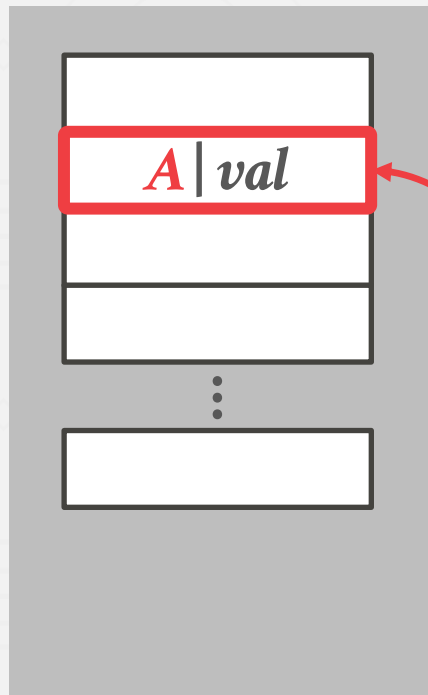
$hash_1(C)$   $hash_2(C)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

Put B

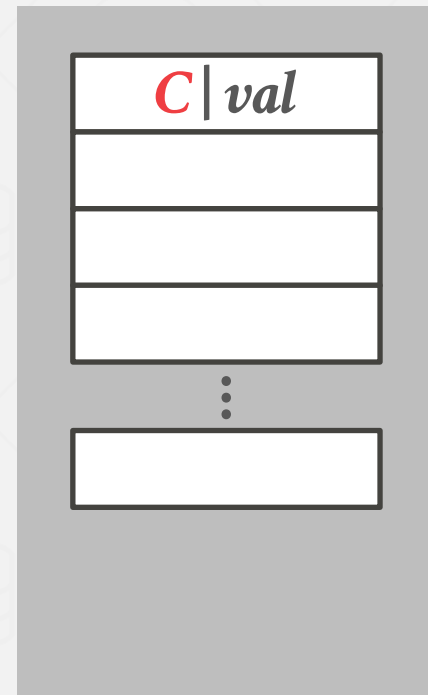
$hash_1(B)$   $hash_2(B)$

Put C

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

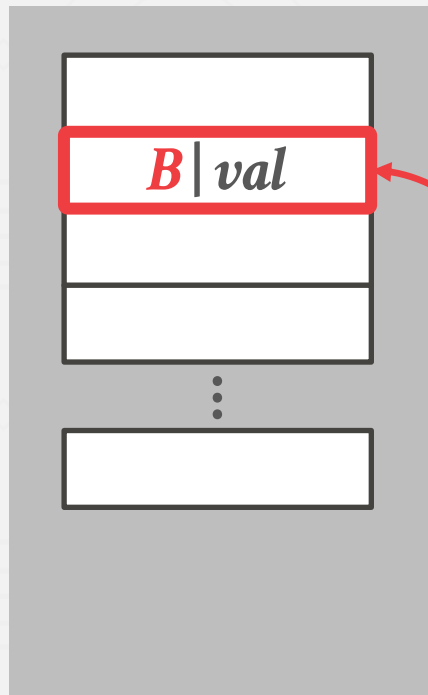
$hash_1(B)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

Put B

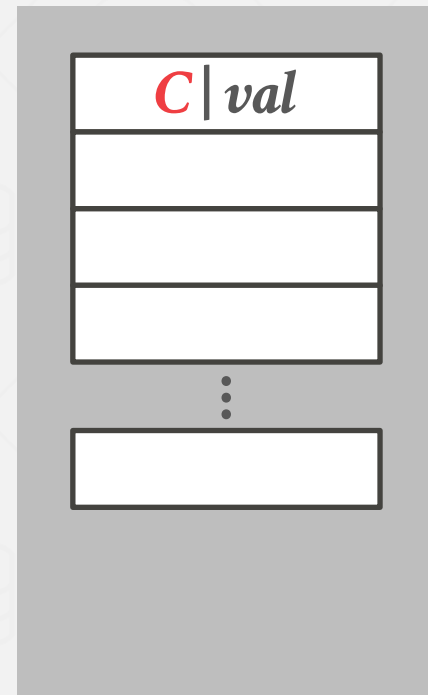
$hash_1(B)$   $hash_2(B)$

Put C

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

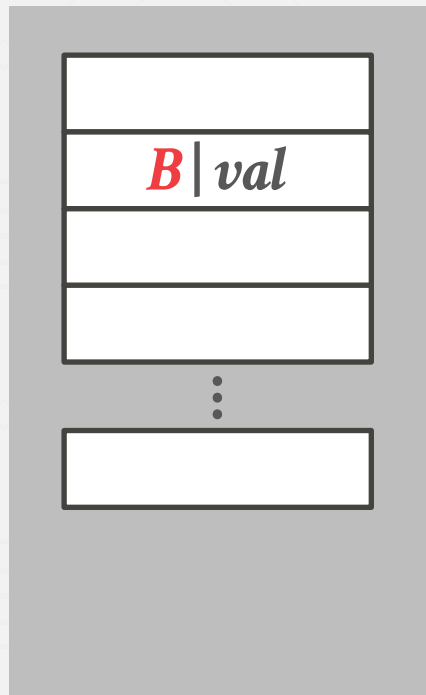
$hash_1(B)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

Put B

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

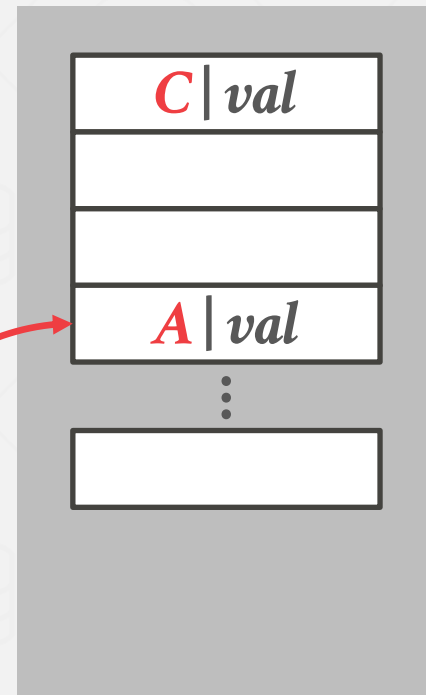
Put C

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

$hash_1(B)$

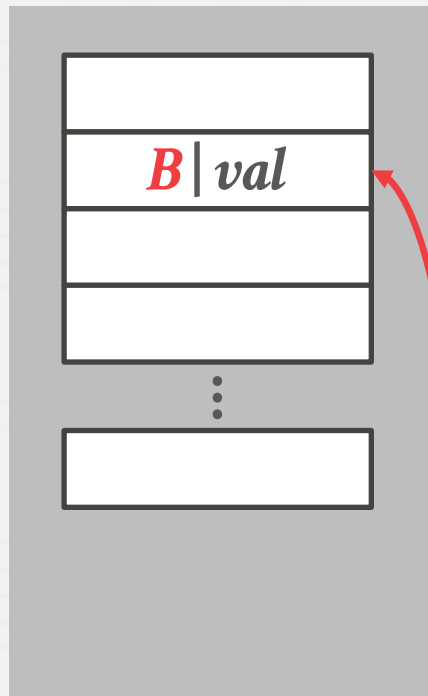
$hash_2(A)$

*Hash Table #2*



# CUCKOO HASHING

*Hash Table #1*



Put A

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

Put B

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

Put C

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

$hash_1(B)$

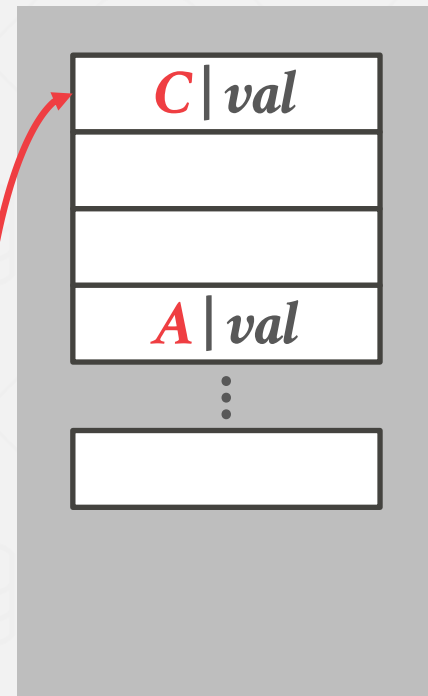
$hash_2(A)$

Get B

$hash_1(B)$

$hash_2(B)$

*Hash Table #2*



# OBSERVATION

---

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

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

Dynamic hash tables resize themselves on demand.

- Chained Hashing
- Extendible Hashing
- Linear Hashing

# CHAINED HASHING

---

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

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

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



# CHAINED HASHING

$hash(key) \% N$

A

B

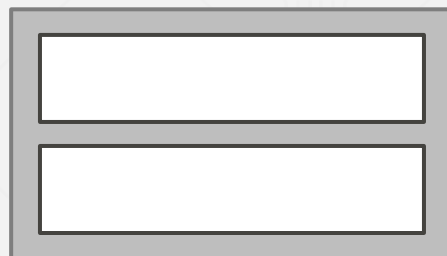
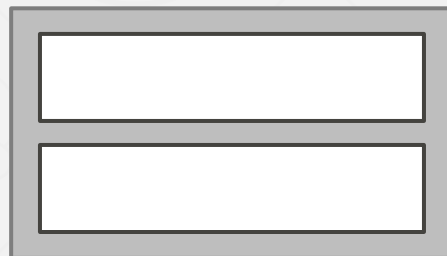
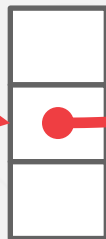
C

D

E

F

*Bucket  
Pointers*



*Buckets*

# CHAINED HASHING

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

A

B

C

D

E

F

*Bucket  
Pointers*



*B | val*

*A | val*

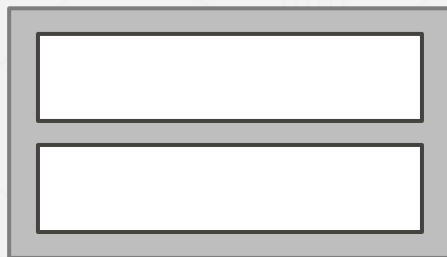
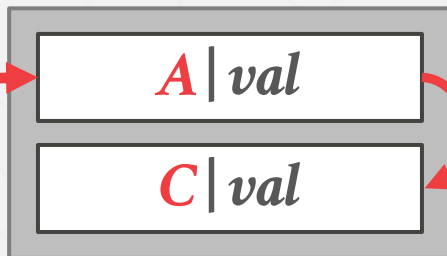
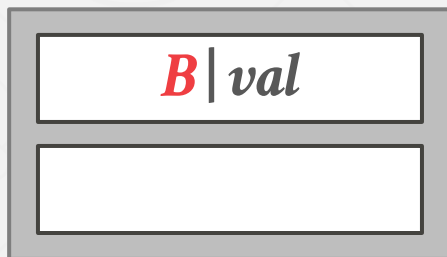
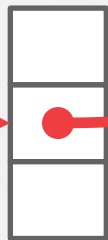
*Buckets*

# CHAINED HASHING

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

A  
B  
C  
D  
E  
F

*Bucket  
Pointers*



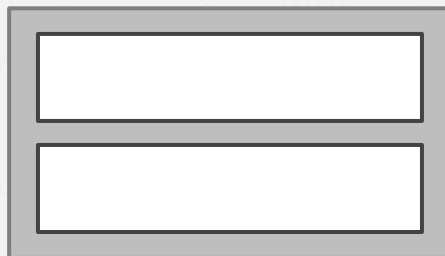
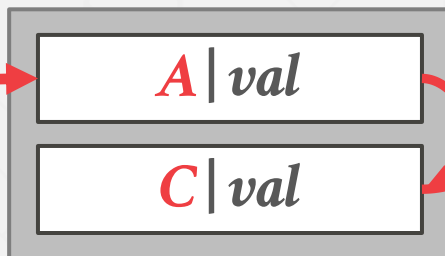
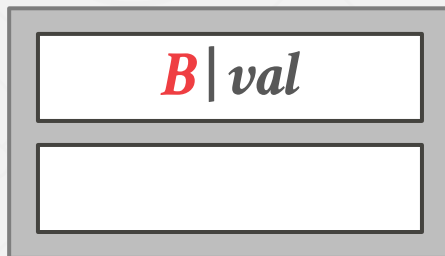
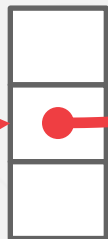
*Buckets*

# CHAINED HASHING

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

A  
B  
C  
D  
E  
F

Bucket  
Pointers

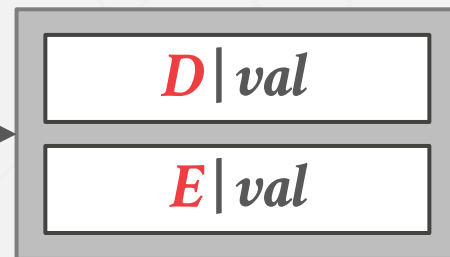
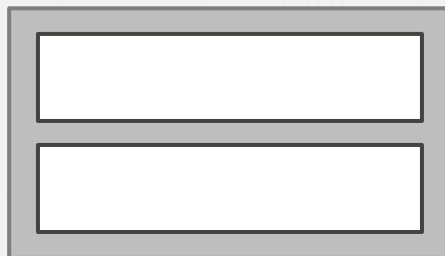
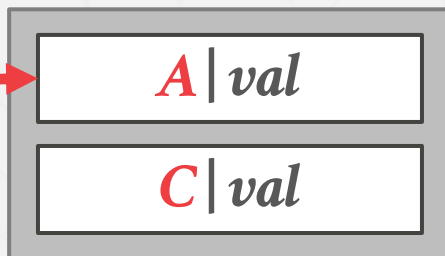
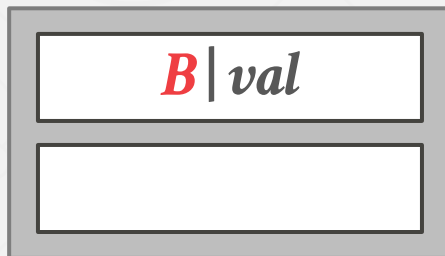
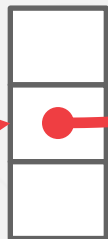


# CHAINED HASHING

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

A  
B  
C  
D  
E  
F

Bucket  
Pointers

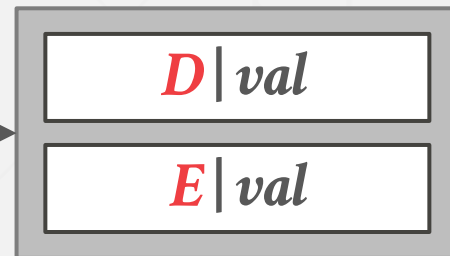
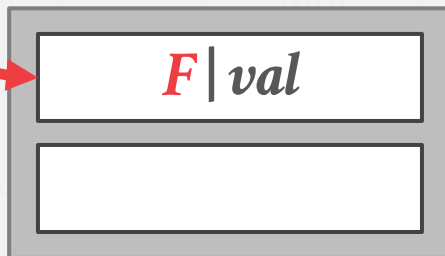
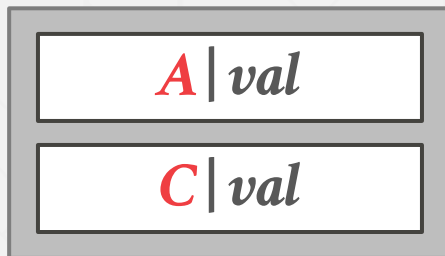
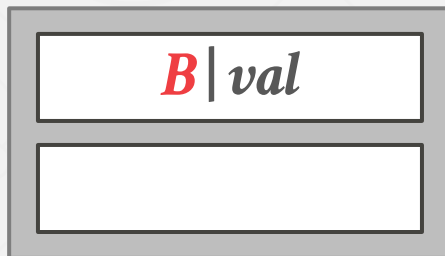


# CHAINED HASHING

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

A  
B  
C  
D  
E  
F

Bucket  
Pointers



# EXTENDIBLE HASHING

---

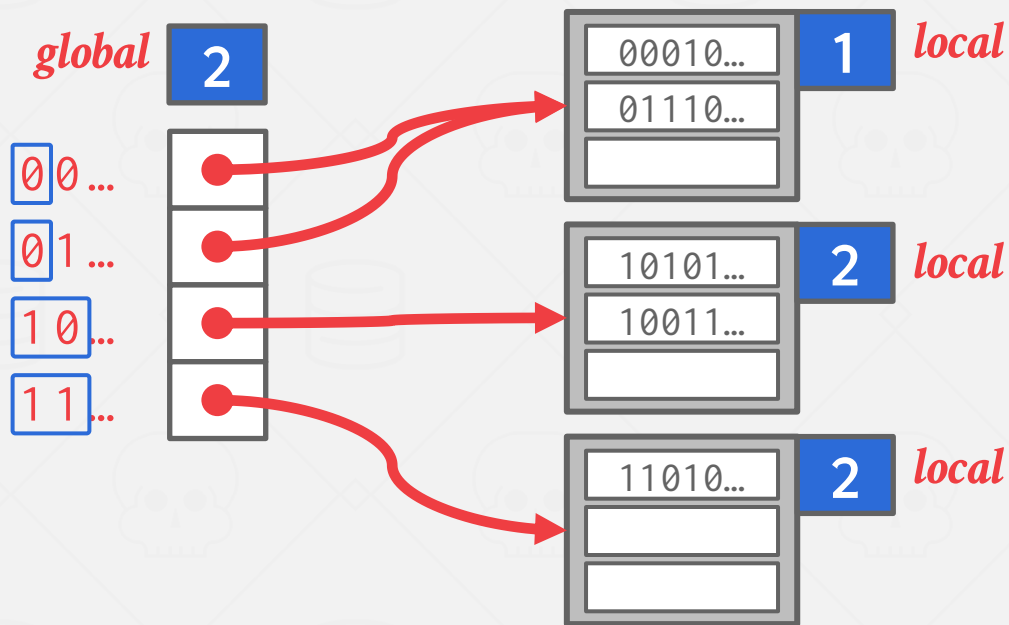
Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

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

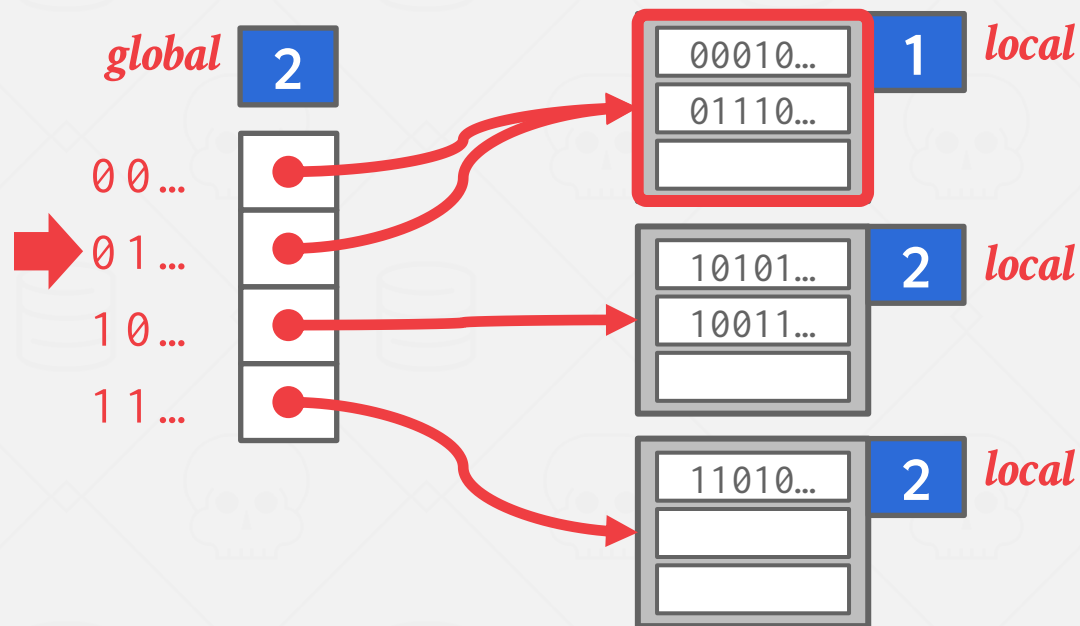
→ Data movement is localized to just the split chain.

# EXTENDIBLE HASHING



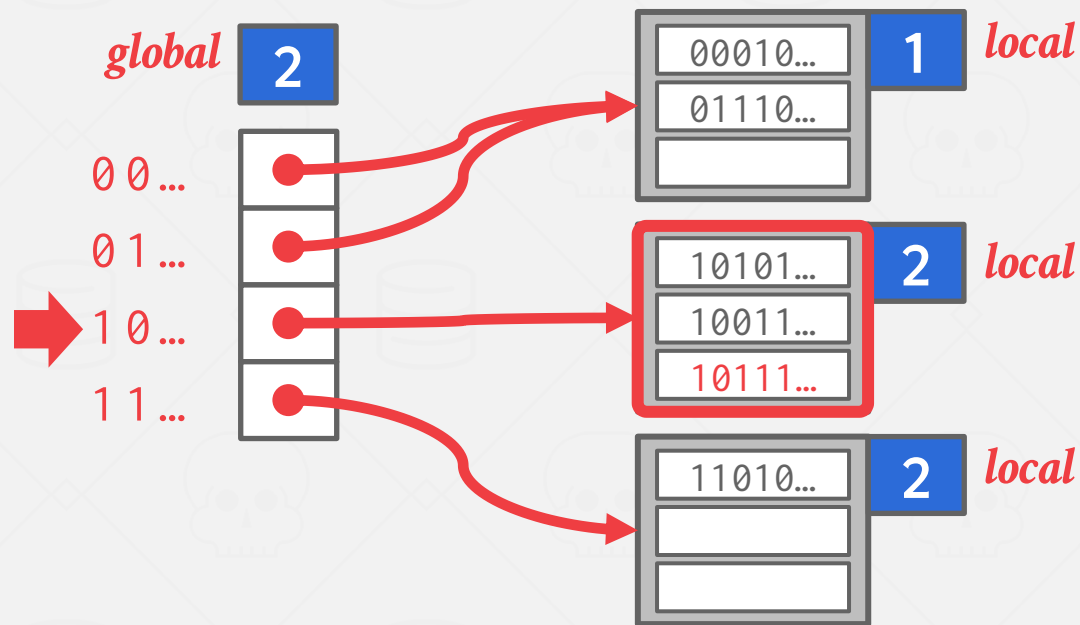


# EXTENDIBLE HASHING



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

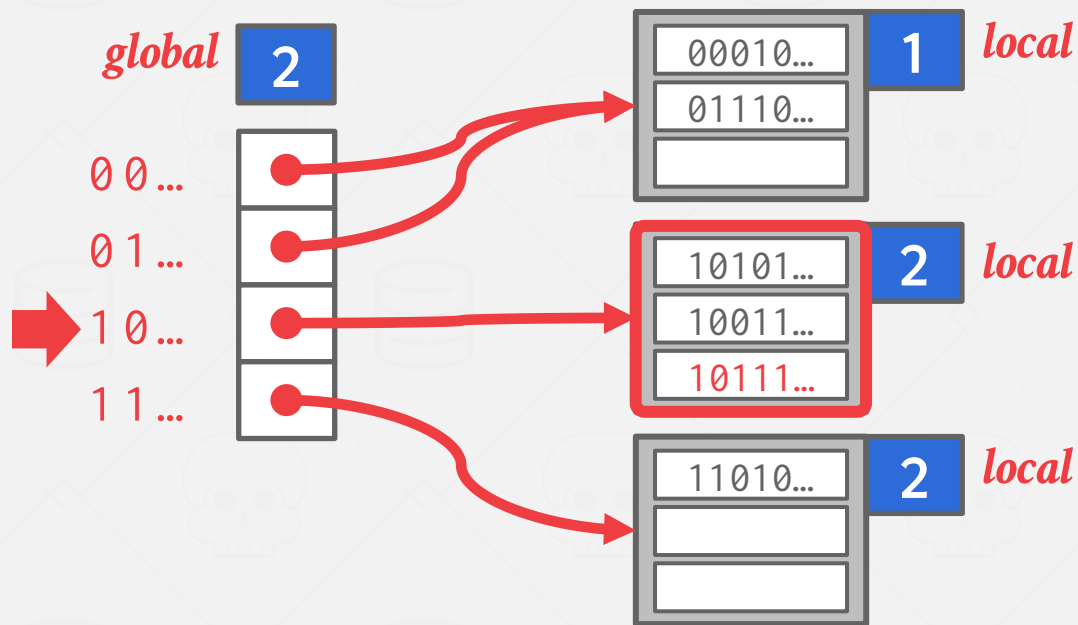
# EXTENDIBLE HASHING



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

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

# EXTENDIBLE HASHING

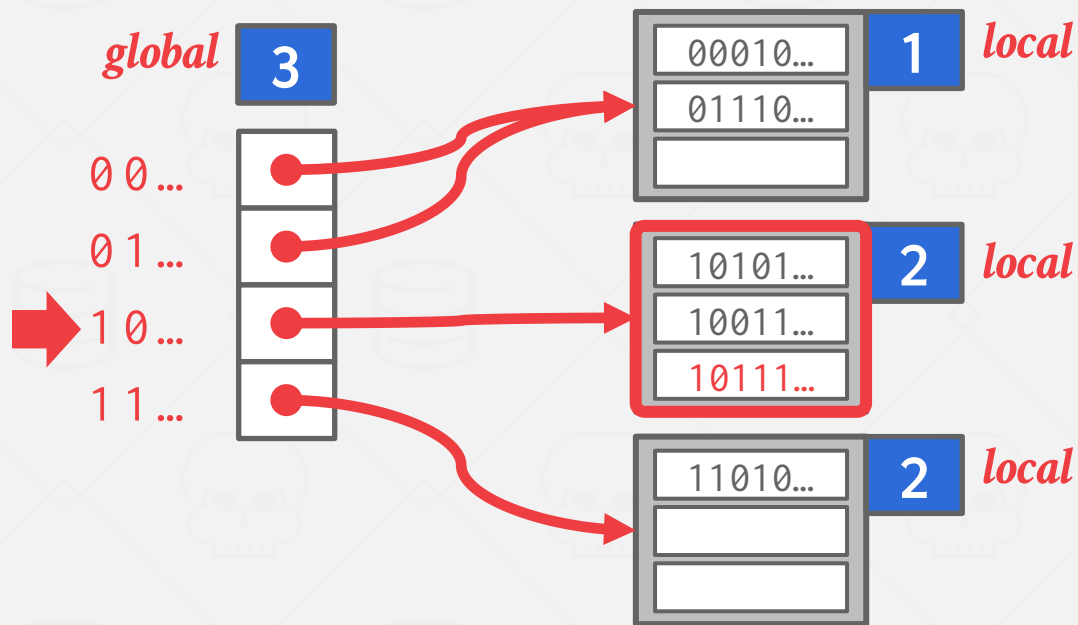


Get A  
 $hash(A) = 01110\dots$

Put B  
 $hash(B) = 10111\dots$

Put C  
 $hash(C) = \boxed{10}100\dots$

# EXTENDIBLE HASHING

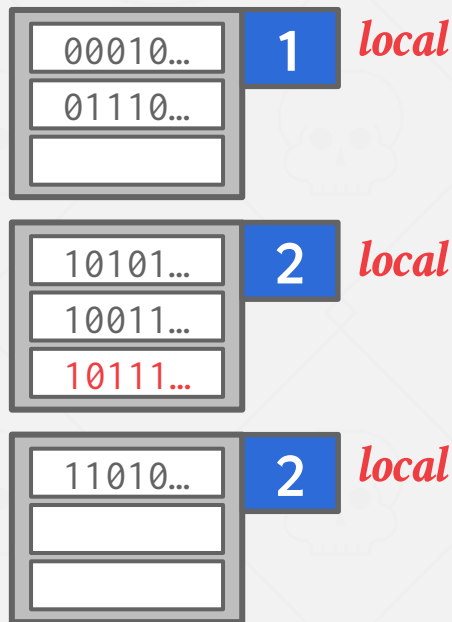
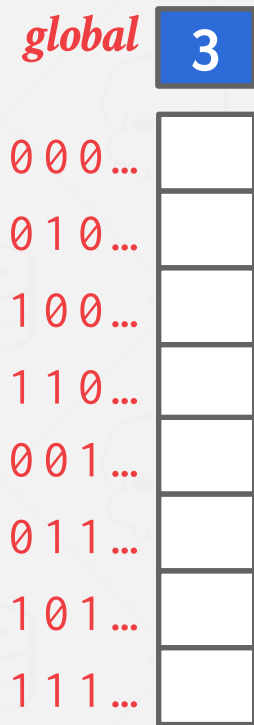


Get A  
 $hash(A) = 01110...$

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

*global*

3

0 0 0 ...

0 1 0 ...

1 0 0 ...

1 1 0 ...

0 0 1 ...

0 1 1 ...

1 0 1 ...

1 1 1 ...

00010...

1

01110...

10011...

3

10101...

3

10111...

11010...

2

Get A

*hash(A) = 01110...*

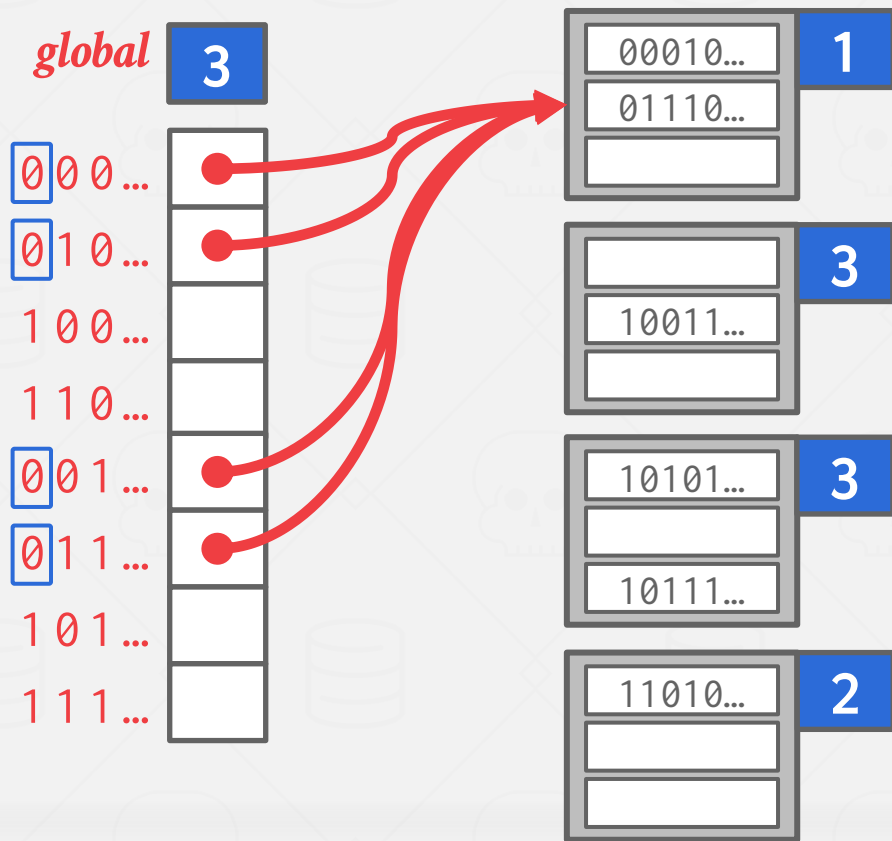
Put B

*hash(B) = 10111...*

Put C

*hash(C) = 10100...*

# EXTENDIBLE HASHING

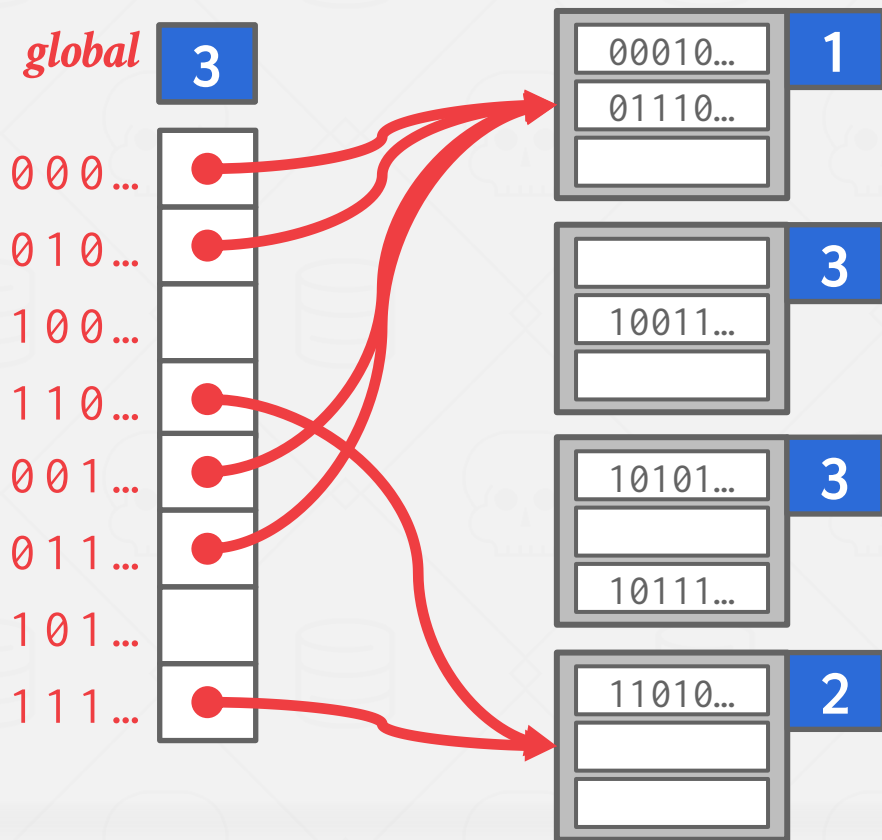


Get A  
 $hash(A) = 01110...$

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

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

# EXTENDIBLE HASHING



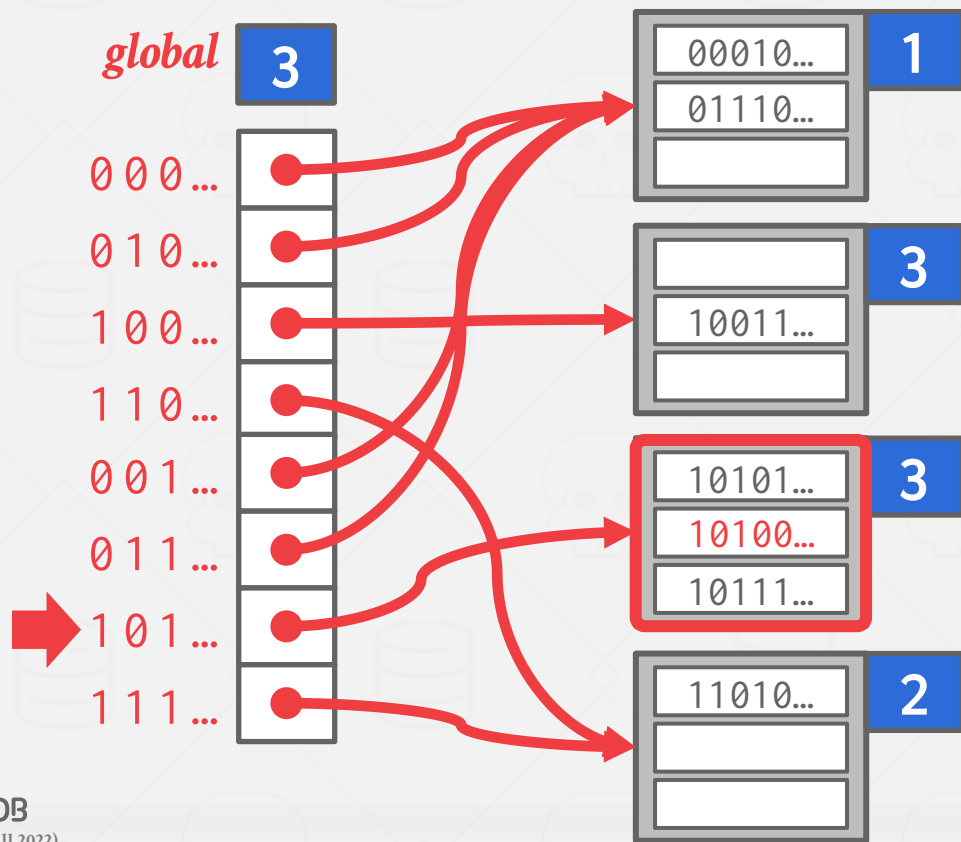
Get A  
 $hash(A) = 01110...$

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

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



# EXTENDIBLE HASHING



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

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

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