Hash Tables

Administrivia

Read the Jon 10/.

- Assignment 2 has been released.
- We will be having a couple of guest lectures later in the semester.

Recap

Access Methods

Access methods are alternative **ways for retrieving specific tuples** from a relation.

- Typically, there is more than one way to retrieve tuples.
- Depends on the availability of **indexes** and the conditions specified in the query for selecting the tuples
- Includes **sequential scan** method of **unordered table heap**
- Includes **index scan** of different types of **index structures**

Hash Tables

Index Structures: Design Decisions

Meta-Data

e7:
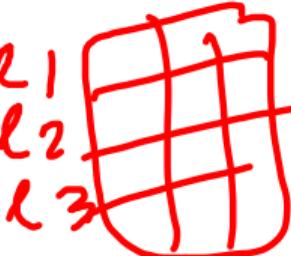
- **Meta-Data Organization**
  - ▶ How to organize meta-data on disk or in memory to support efficient access to specific tuples?
- **Concurrency**
  - ▶ How to allow multiple threads to access the derived data structure at the same time without causing problems?

Today's Agenda

- Hash Tables
- Hash Functions
- Static Hashing Schemes
- Dynamic Hashing Schemes

Hash Tables

Hash Tables

- A hash table implements an **unordered associative array** that maps keys to values.
  - ▶ mymap.insert('a', 50)
  - ▶ mymap['b']=100;
  - ▶ mymap.find('a')
  - ▶ mymap['a']
- It uses a **hash function** to compute an offset into the array for a given key, from which the desired value can be found.
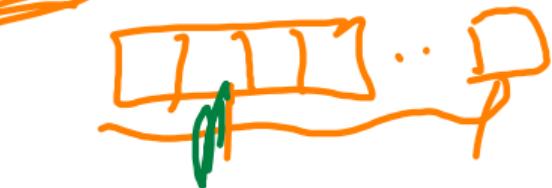
Hash Tables

$O(n)$

$O(\lg n)$

$\dfrac{3n}{\log n}$

- Operation Complexity:
  - ▶ Average: $O(1)$
  - ▶ Worst: $O(n)$
- Space Complexity: $O(n)$
- Constants matter in practice.
- **<u>Reminder:</u>** In theory, there is no difference between theory and practice. But in practice, there is.

Naïve Hash Table

$$hash("def") = 2$$

$$= 2$$

hash(key)

- 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.

0	abc
1	∅
2	def
⋮	
n	xyz
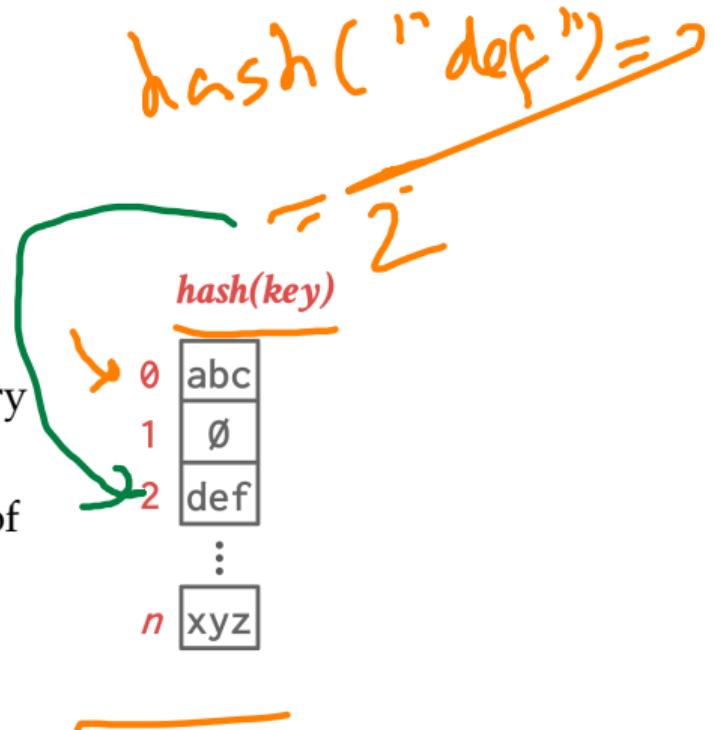
Naïve Hash Table

10 elements

100 slots

- 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.

$O(1)$

hash(key)

0	●
1	
2	●
⋮	
n	●

| abcdefghi |
| xyz123 |
| defghijk |
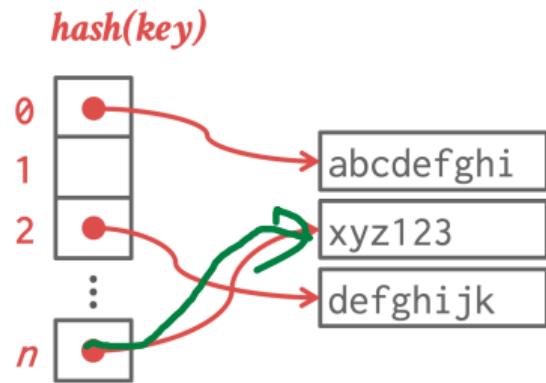
Assumptions

- You know the number of elements ahead of time.
- Each key is unique (*e.g.*, SSN ID ⟶ Name).
- Perfect hash function (no **collision**).
  - If key1 != key2, then hash(key1) != hash(key2)

Peter Maria

hash("Peter") = 2
hash("Maria") = 2

Hash Table: Design Decisions

- Design Decision 1: **Hash Function**
  - ▶ How to map a large key space into a smaller domain of array offsets.
  - ▶ 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 steps to find/insert keys.

Hash Functions

Hash Functions

keys $\in$ [0..100] $\longrightarrow$ [0..10]

- For any input key, return an integer representation of that key.
- We want to map the key space to a smaller domain of array offsets.
- We do **not** want to use a cryptographic hash function for DBMS hash tables.
- 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 to a fast, general purpose hash function.
- Google CityHash (2011)
  - ▶ Designed to be faster for short keys (<64 bytes).
  - ▶ New assembly instructions have been added recently to accelerate hashing
- Facebook XXHash (2012)
  - ▶ From the creator of zstd compression.
- Google FarmHash (2014)
  - ▶ Newer version of CityHash with better collision rates.

Hash Function Benchmark

std::hash (C++)

Intel Core i7-8700K @ 3.70GHz
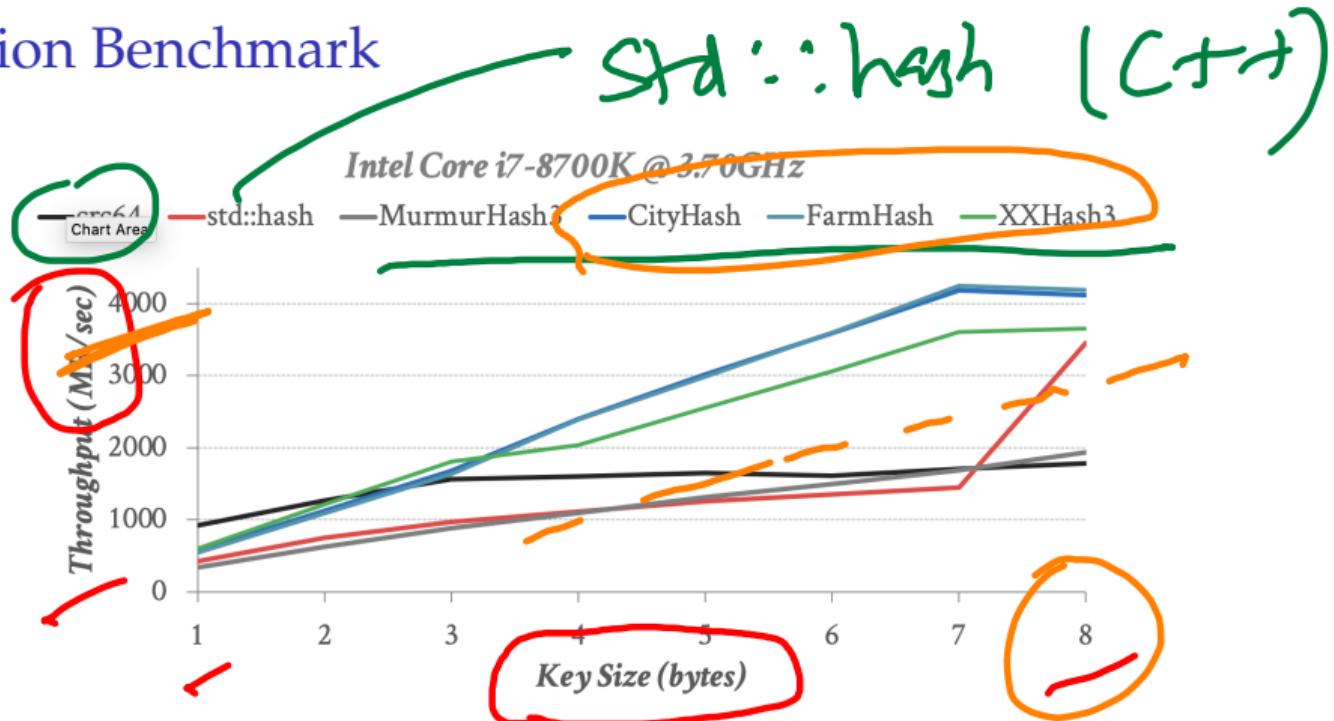
- Source
- Intel Core i7-8700K @ 3.70GHz

Hash Function Benchmark
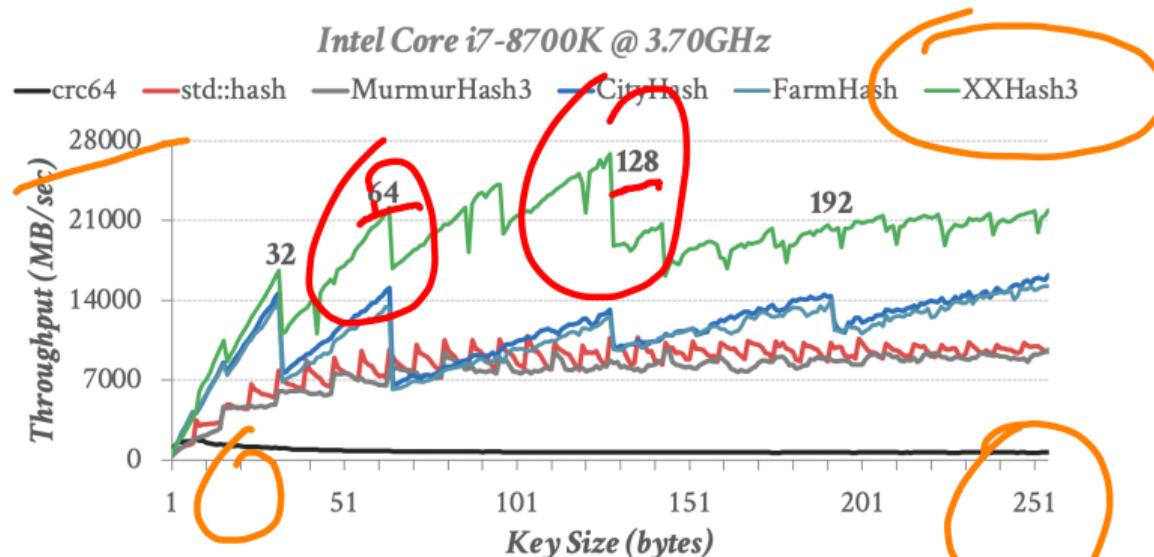
- Source
- Intel Core i7-8700K @ 3.70GHz

Static Hashing Schemes

Static Hashing Schemes

- These schemes are typically used when you have an **upper bound** on the number of keys that you want to store in the hash table.
- These are often used during query execution because they are **faster than dynamic hashing schemes**.
  - ▶ Approach 1: Linear Probe Hashing
  - ▶ Approach 2: Robin Hood Hashing
  - ▶ Approach 3: Cuckoo Hashing

resize quickly

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.
    - Have to store the key in the index to know when to stop scanning.
   - Insertions and deletions are generalizations of lookups.

# Linear Probe Hashing
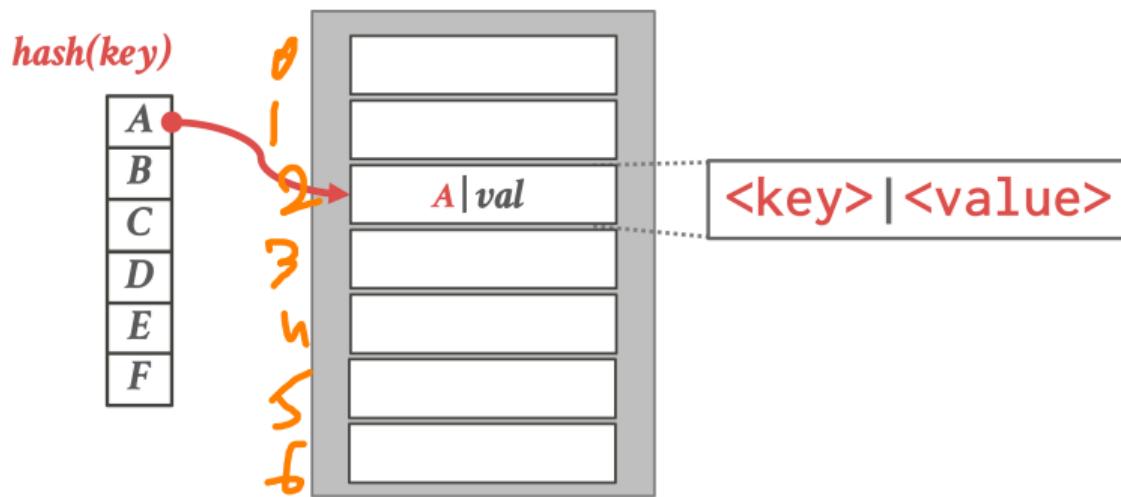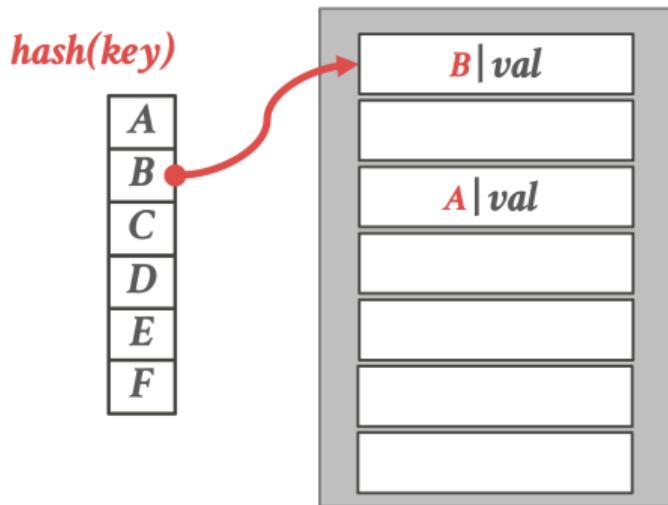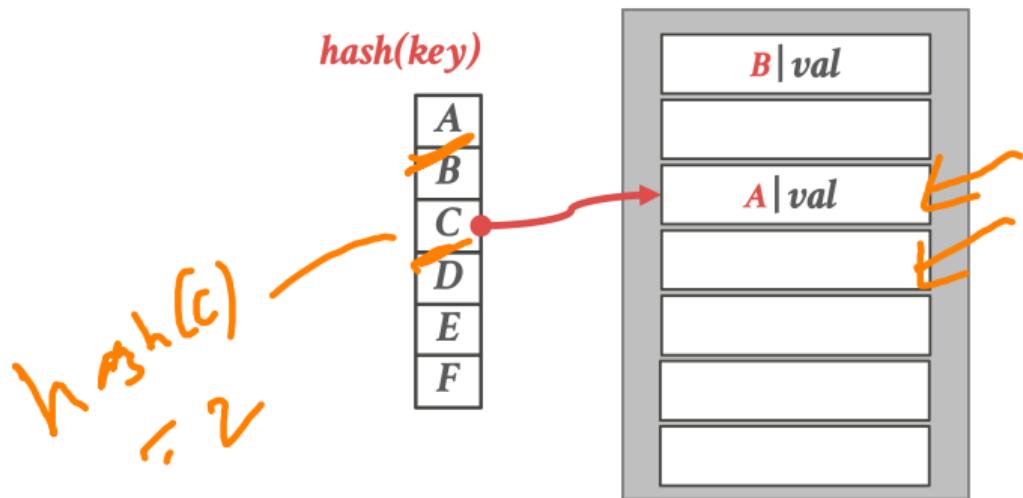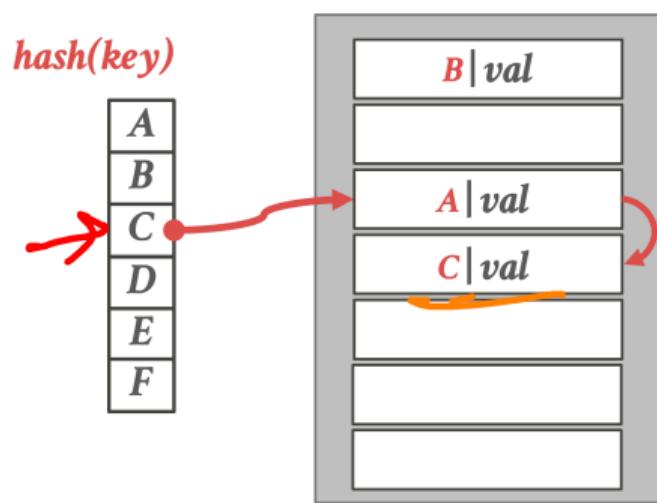
Linear Probe Hashing

Linear Probe Hashing

Linear Probe Hashing

Linear Probe Hashing
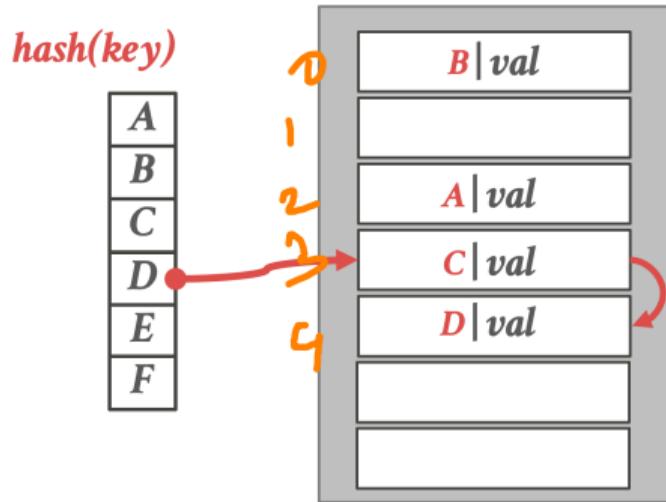
# Linear Probe Hashing
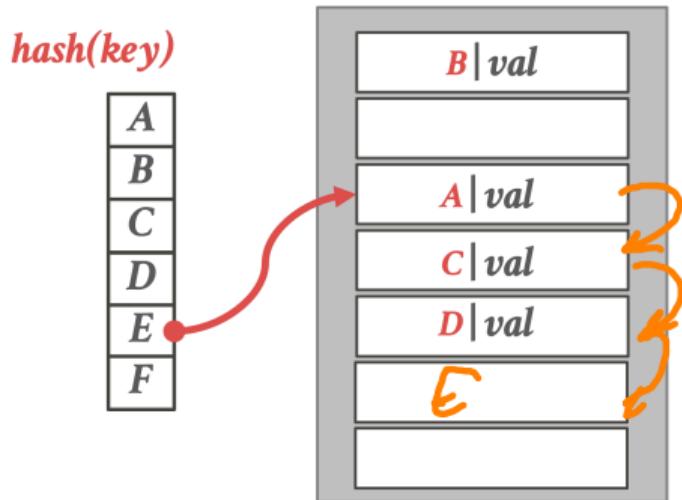
Linear Probe Hashing
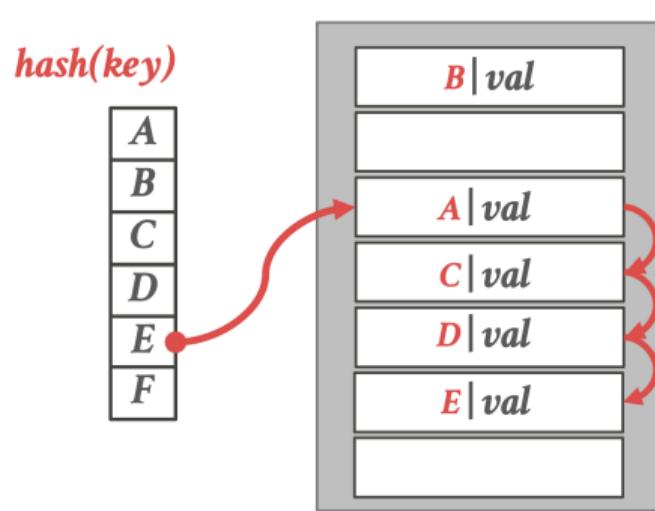
Linear Probe Hashing
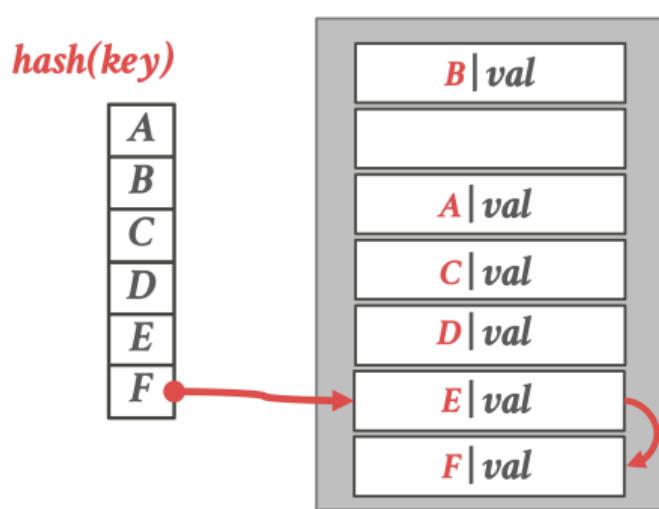
Linear Probe Hashing – Delete

- It is **<u>not sufficient</u>** to simply delete the key
- This would affect searches for other keys that have a hash value earlier than the emptied cell, but that are stored in a position later than the emptied cell.
- Solutions:
  - ▶ Approach 1: Tombstone
  - ▶ Approach 2: Movement *of keys*

Linear Probe Hashing – Delete

Linear Probe Hashing – Delete

# Linear Probe Hashing – Delete

Linear Probe Hashing – Delete

Linear Probe Hashing – Delete
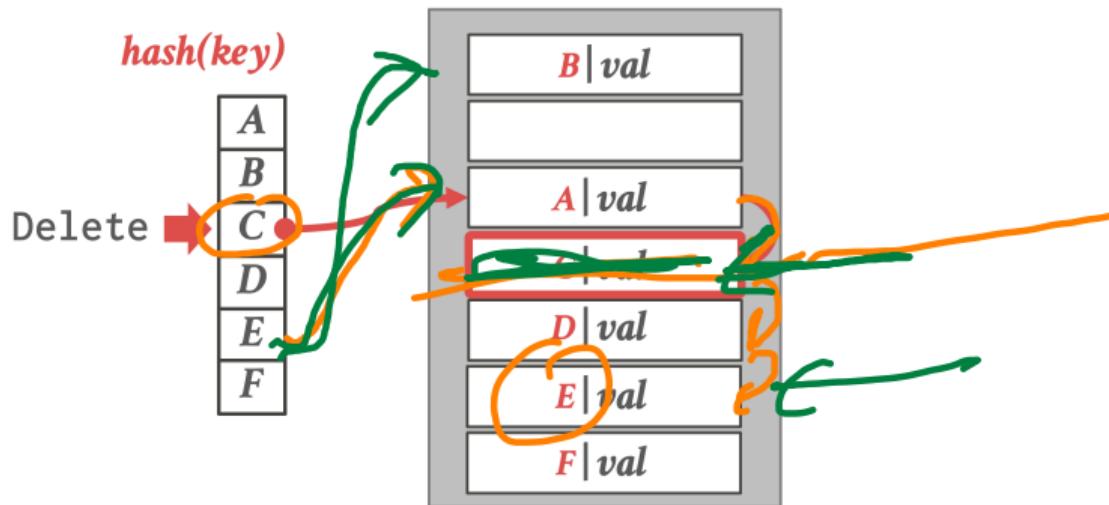
Linear Probe Hashing – Delete

hash(key)
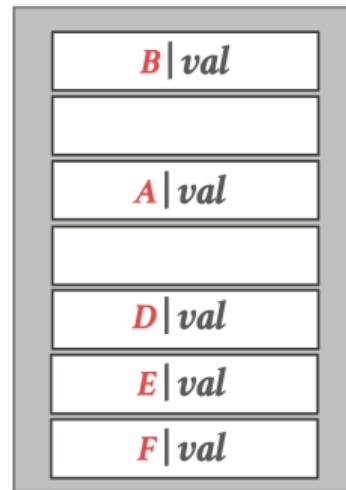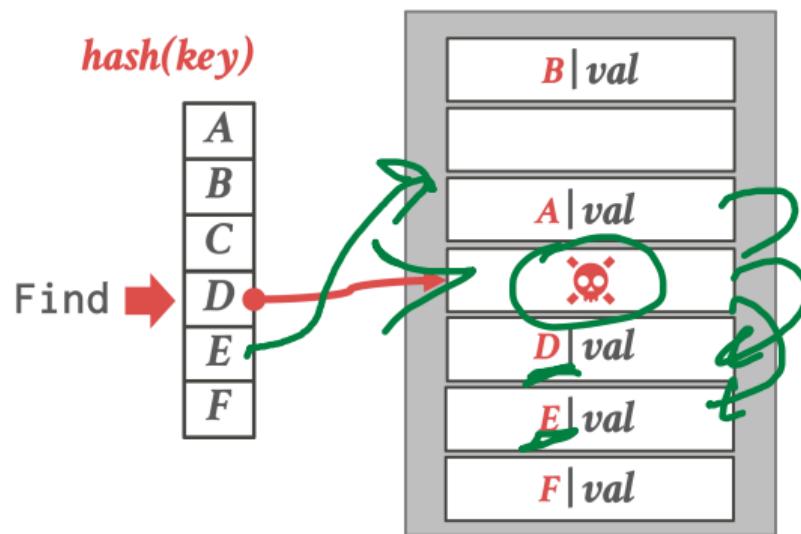
Find

A
B
C
D
E
F

B | val

A | val

D | val

E | val

F | val

Compression

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.
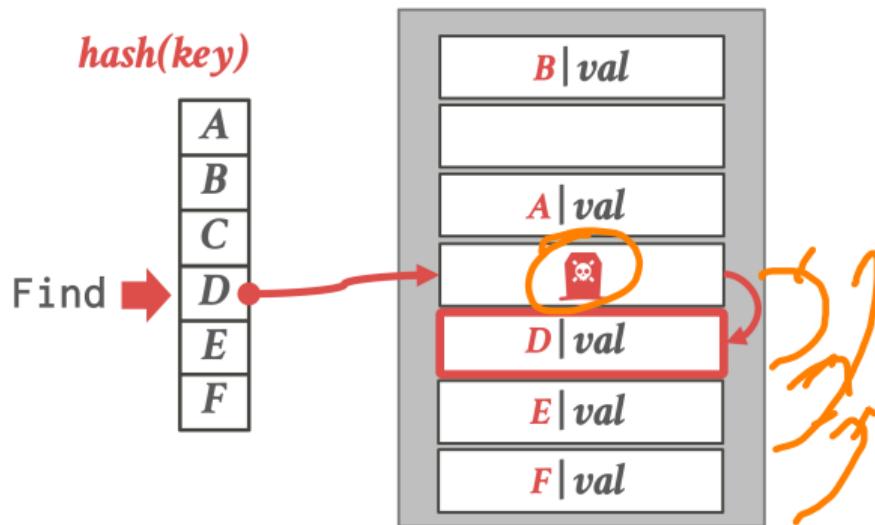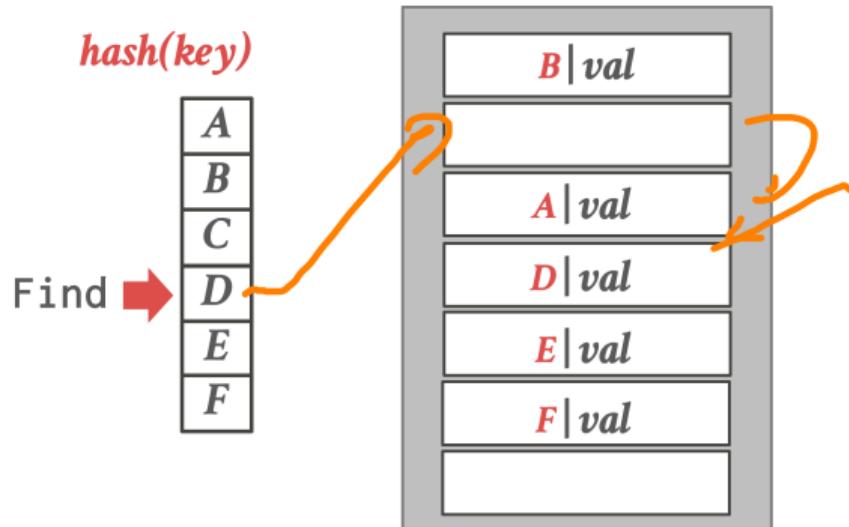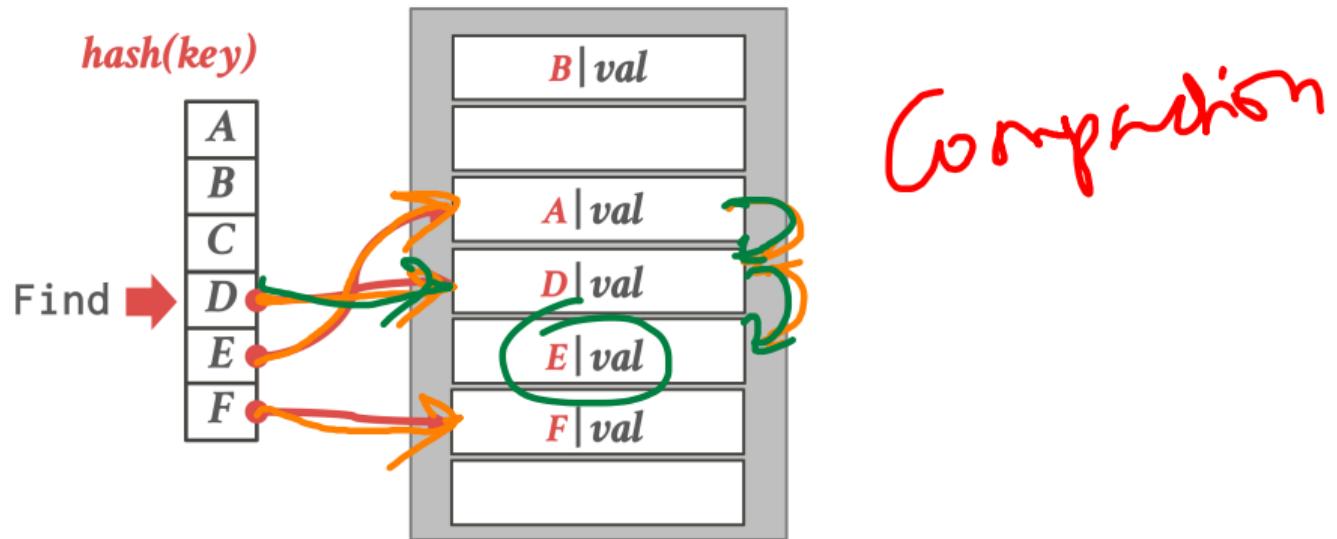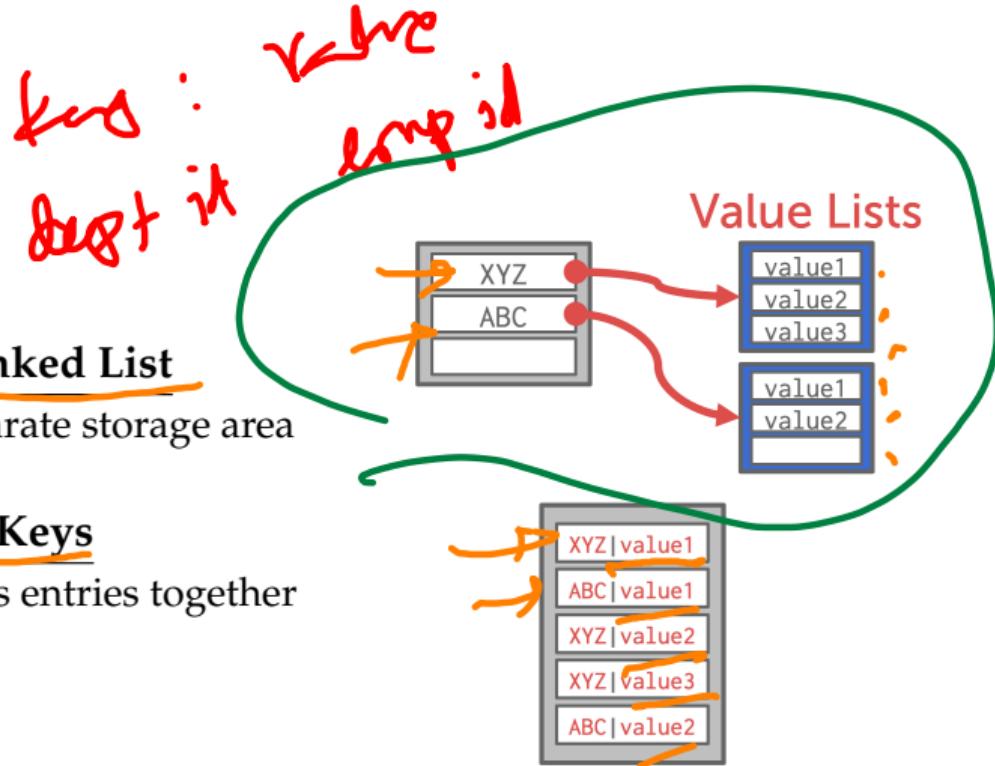
Robin Hood Hashing

- 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.
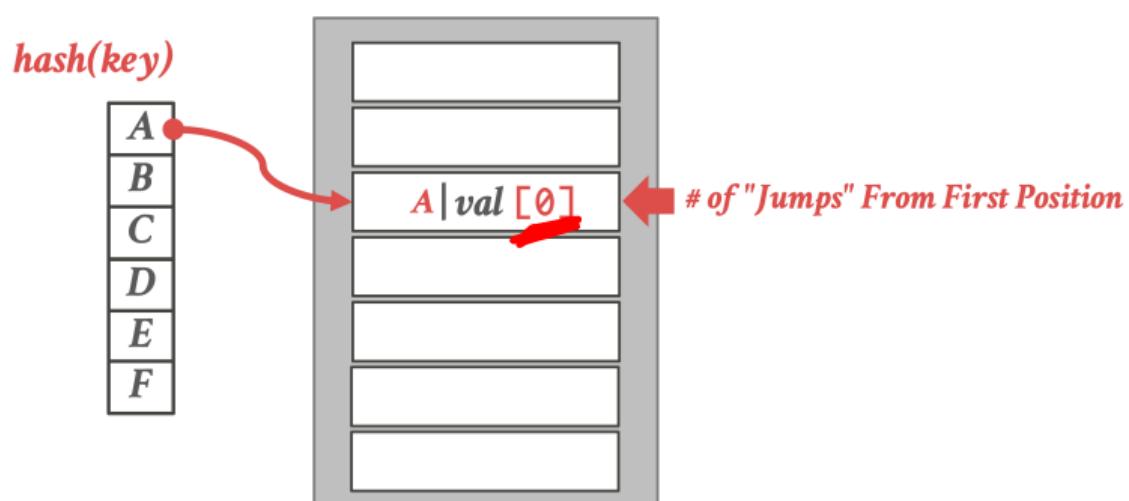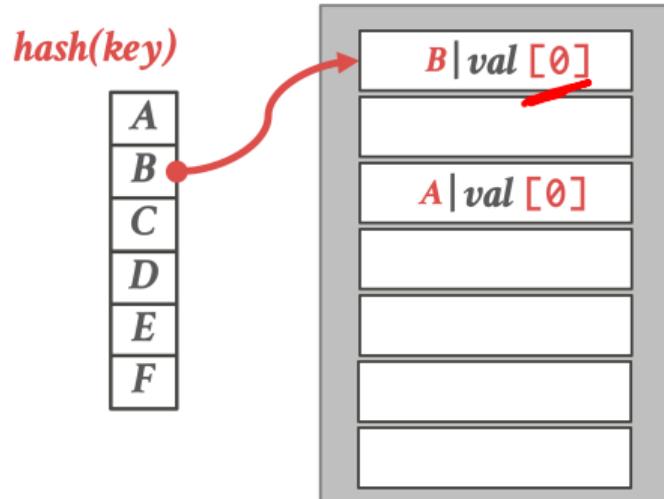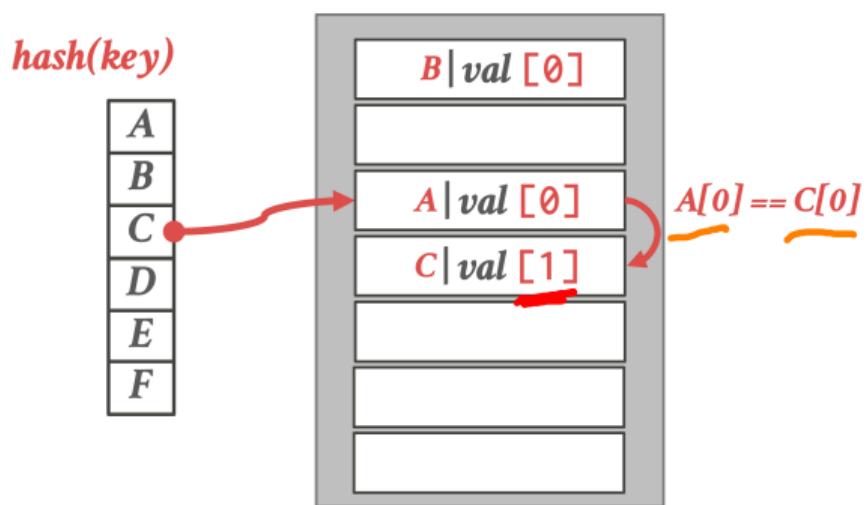
Robin Hood Hashing

hash(key)

A | *val* [0]

of "Jumps" From First Position

Robin Hood Hashing

Robin Hood Hashing

Robin Hood Hashing

Robin Hood Hashing

hash(key)

A
B
C
D
E
F

B | val [0]

A | val [0]
C | val [1]
D | val [1]

A[0] == E[0]
C[1] == E[1]
D[1] < E[2]

Robin Hood Hashing

Robin Hood Hashing

in-memory hash table

hash(key)

	B | *val* [0]
A	
B	*A* | *val* [0]
C	*C* | *val* [1]
D	*E* | *val* [2]
E	*D* | *val* [2]
F	

A[0] == E[0]

C[1] == E[1]

D[1] < E[2]

D 1
E 3

O(n)

Robin Hood Hashing

hash(key)

A	
B	
C	
D	
E	
F	

B | val [0]

A | val [0]
C | val [1]
E | val [2]
D | val [2]
F | val [1]

D[2] > F[0]

$O(n)$
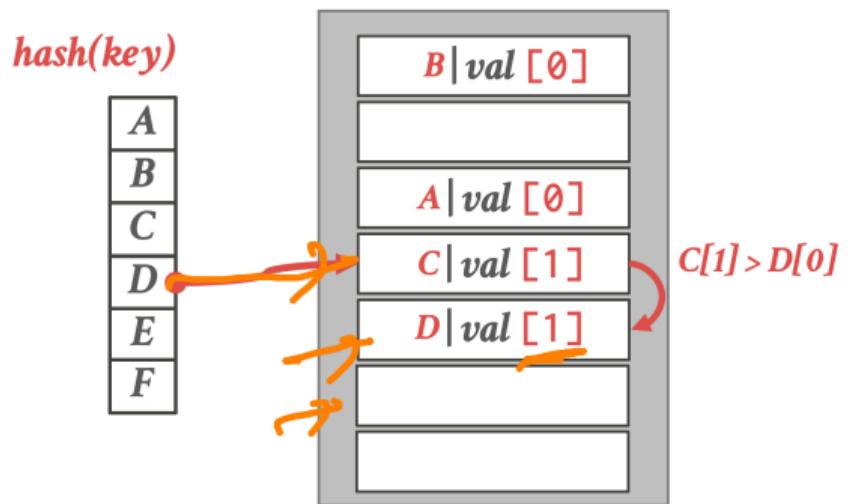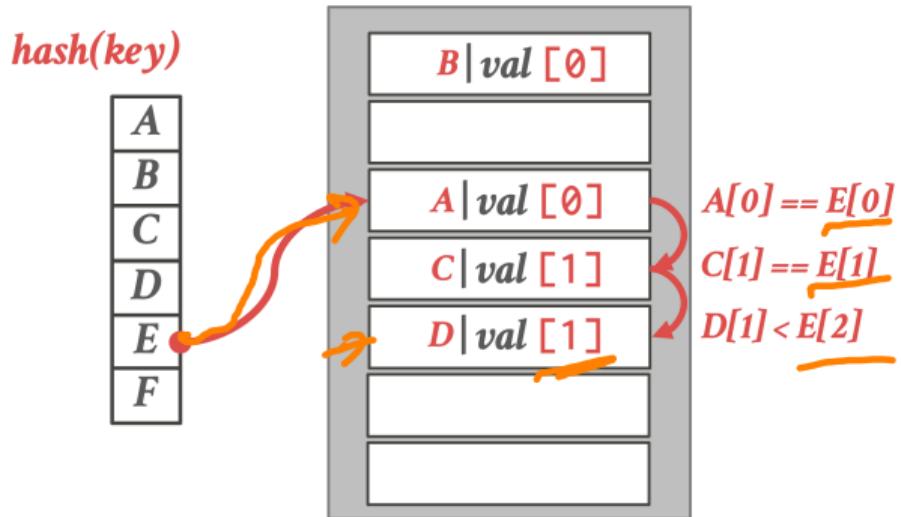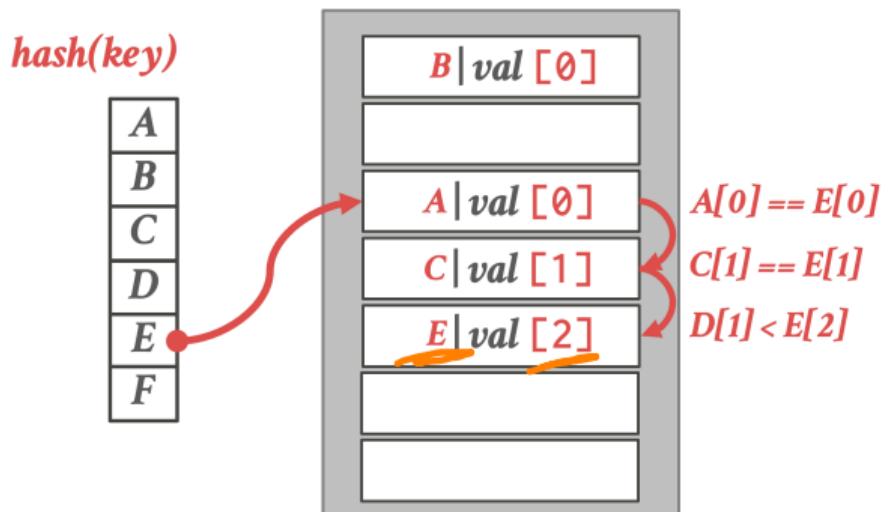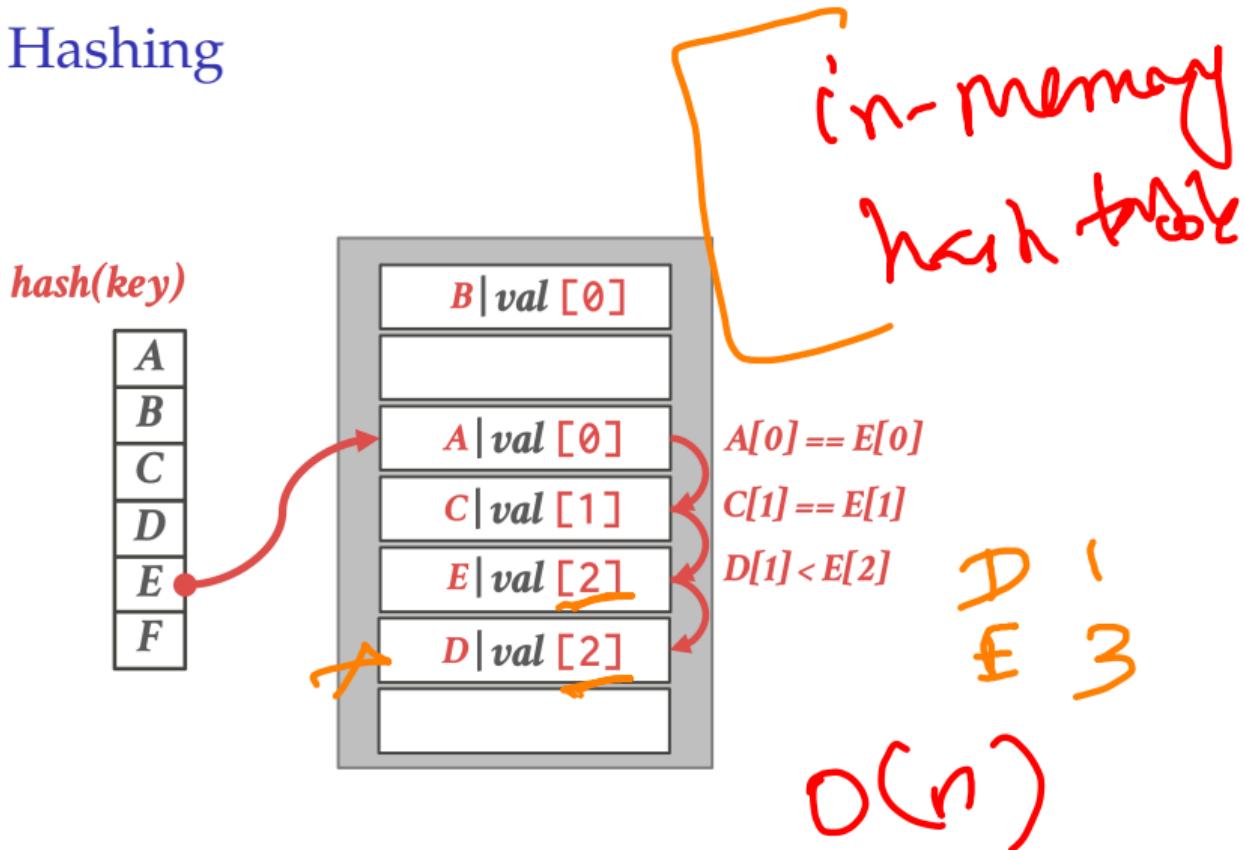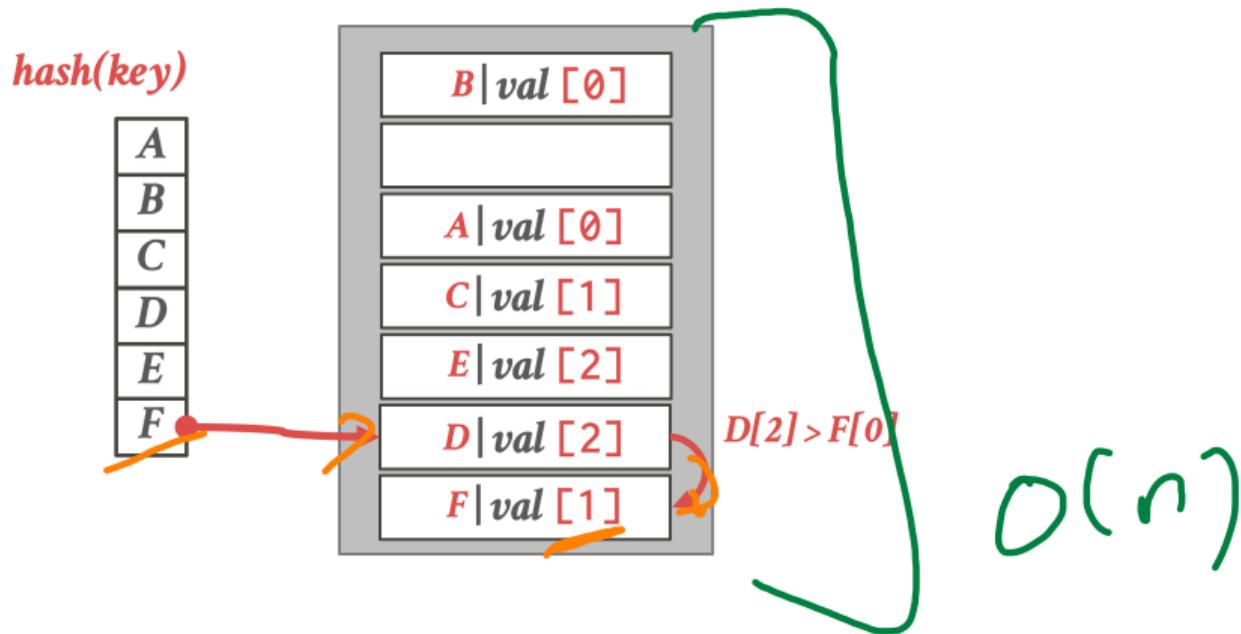
Cuckoo Hashing

- Use **multiple hash tables** with different hash function seeds.
 - ▶ On insert, check every table and pick anyone that has a free slot.
  - ▶ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.
- Look-ups and deletions are always O(1) because only one location **per hash table** is checked.

# of hash tables

Cuckoo Hashing
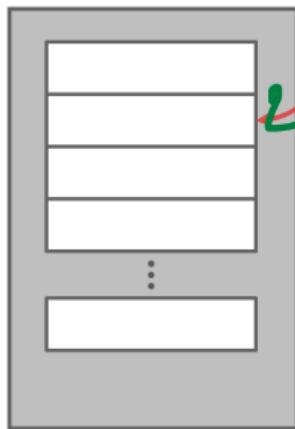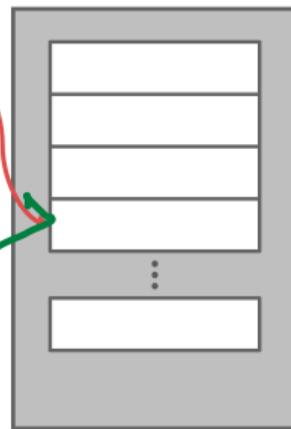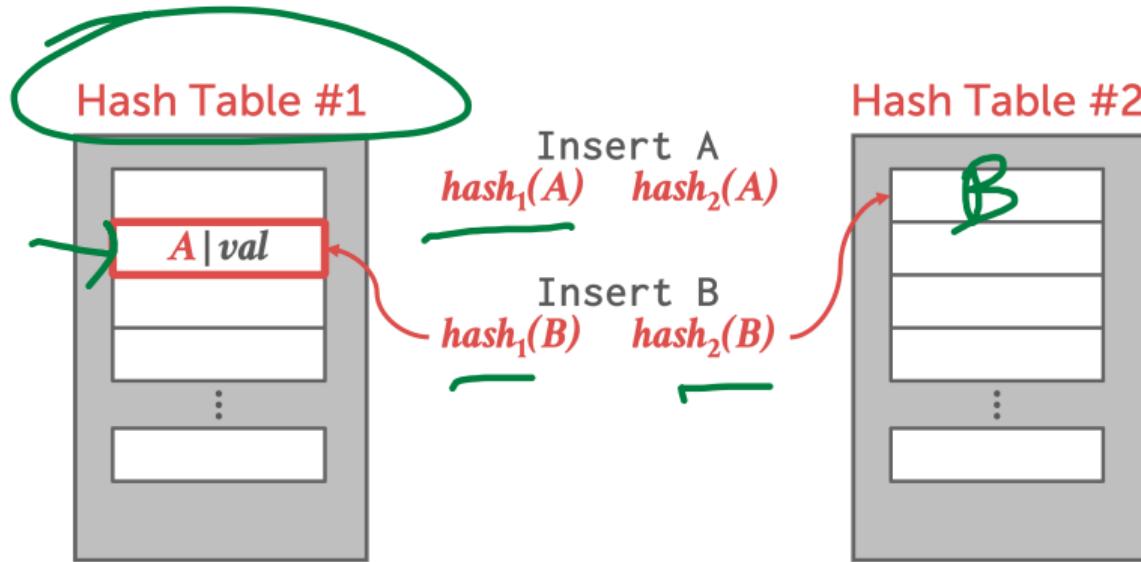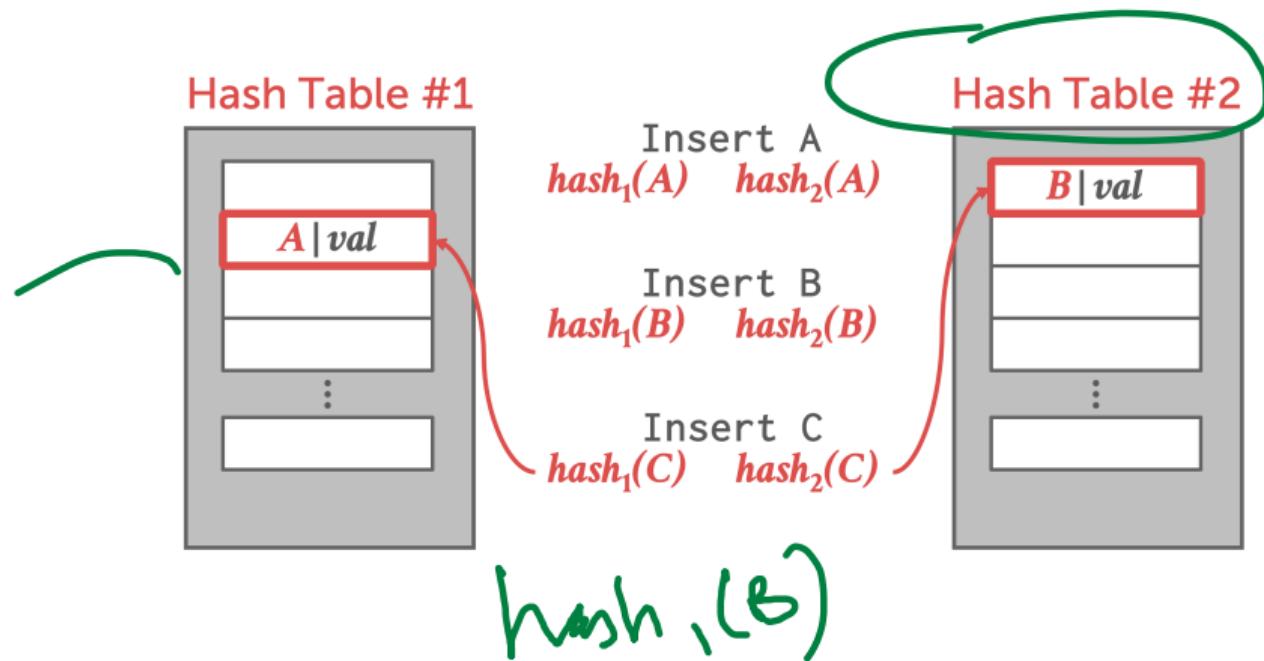
Cuckoo Hashing

Cuckoo Hashing

Hash Table #1

Hash Table #2

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

$A|val$

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

$B|val$

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

hash₁(B)

Cuckoo Hashing

Cuckoo Hashing

Hash Table #1

Hash Table #2

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

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

Insert C
$hash_1(C)$ $hash_2(C)$
$hash_1(B)$

$A\,|\,val$

$C\,|\,val$
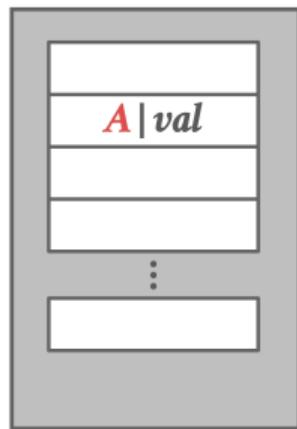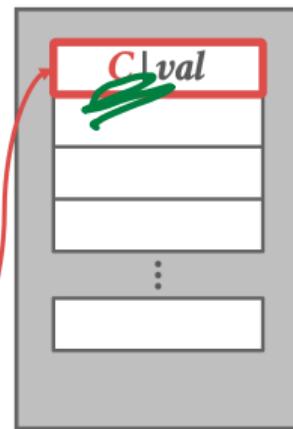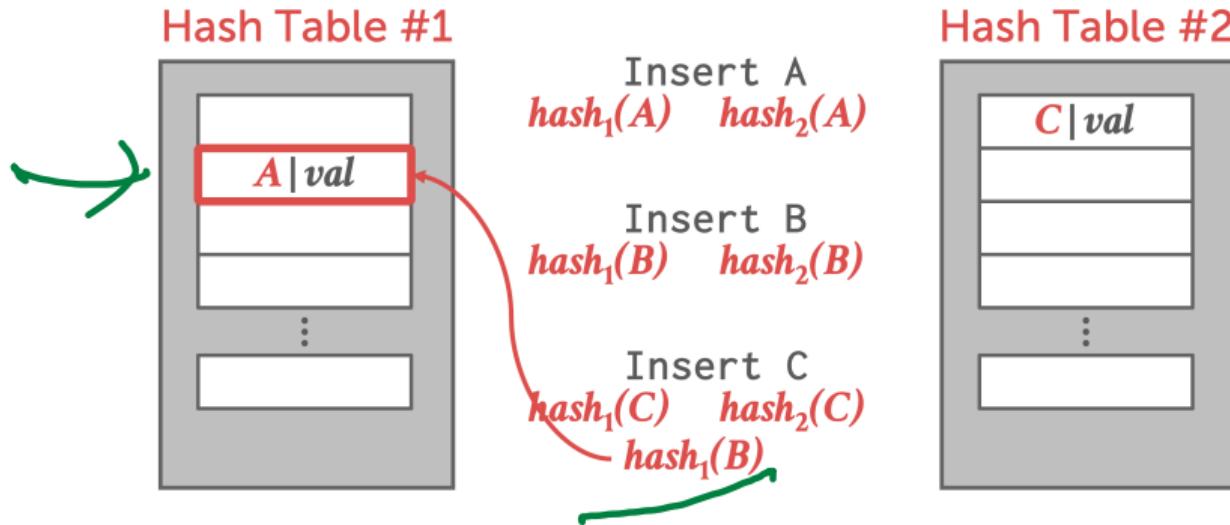
Cuckoo Hashing

Hash Table #1

Hash Table #2

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

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

Insert C
$hash_1(C)$ $hash_2(C)$
$hash_1(B)$
$hash_2(A)$

$B | val$

$C | val$

$A | val$
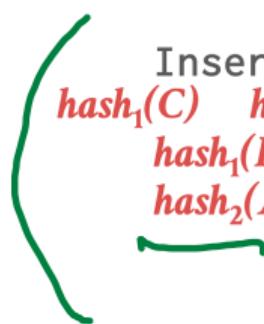
Observation

- Static hashing schemes require the DBMS to know the number of keys to be stored.
  - Otherwise it has to rebuild the table if it needs to grow/shrink the table in size. Why?
    - You would have to take a latch on the entire hash table to prevent threads from adding new entries.
- Dynamic hashing schemes resize themselves on demand.
  - Approach 1: Chained Hashing
  - Approach 2: Extendible Hashing
  - Approach 3: Linear Hashing
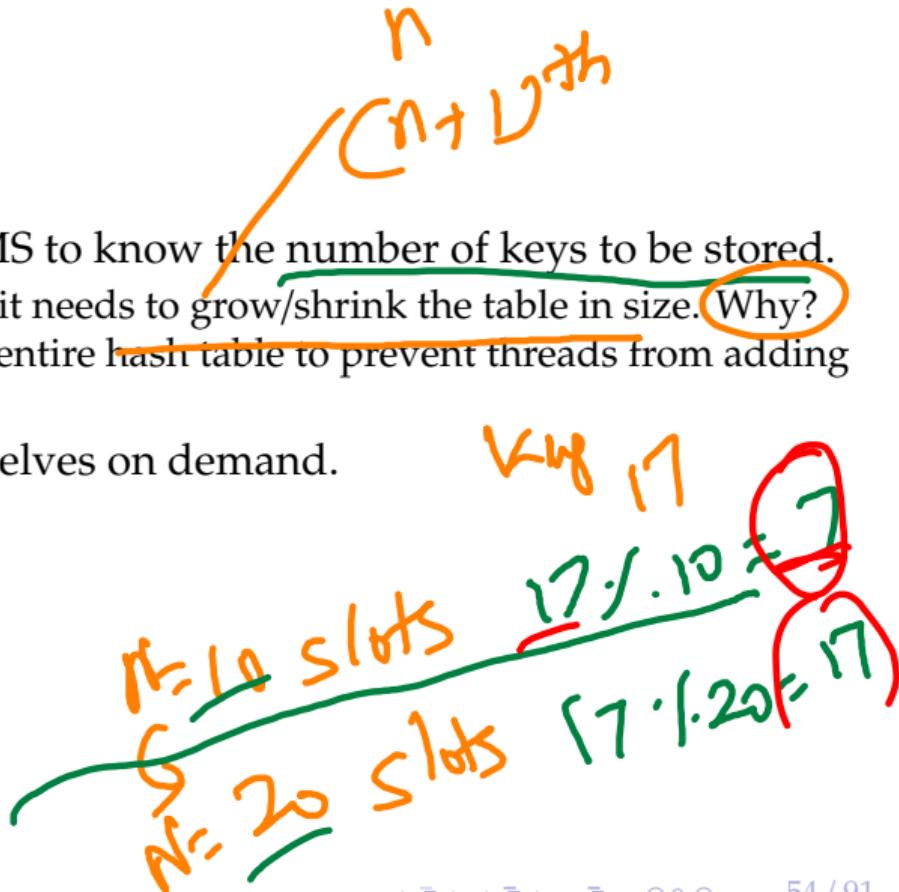
Dynamic Hashing Schemes

Chained Hashing

hash (my) =

- Maintain a linked list of **buckets for each slot** in the hash table.
- Resolve collisions by placing all keys with the same hash value 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

perfect *no collision*

- Unlike static hashing schemes, two different keys may hash to the same offset
- If you want to enforce **unique keys**, then you have perform an additional comparison of each key to determine whether they exactly match
- So, unlike static hashing schemes, need to retain the original key in the table

Chained Hashing

keys

Buckets

hash(key)

Ø
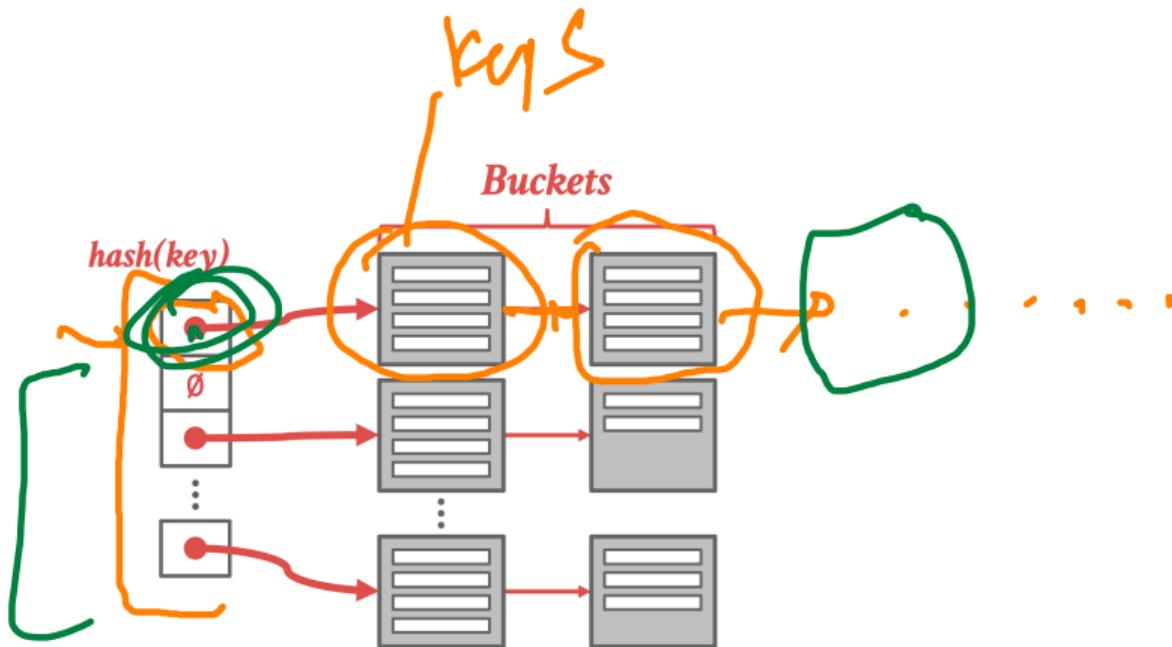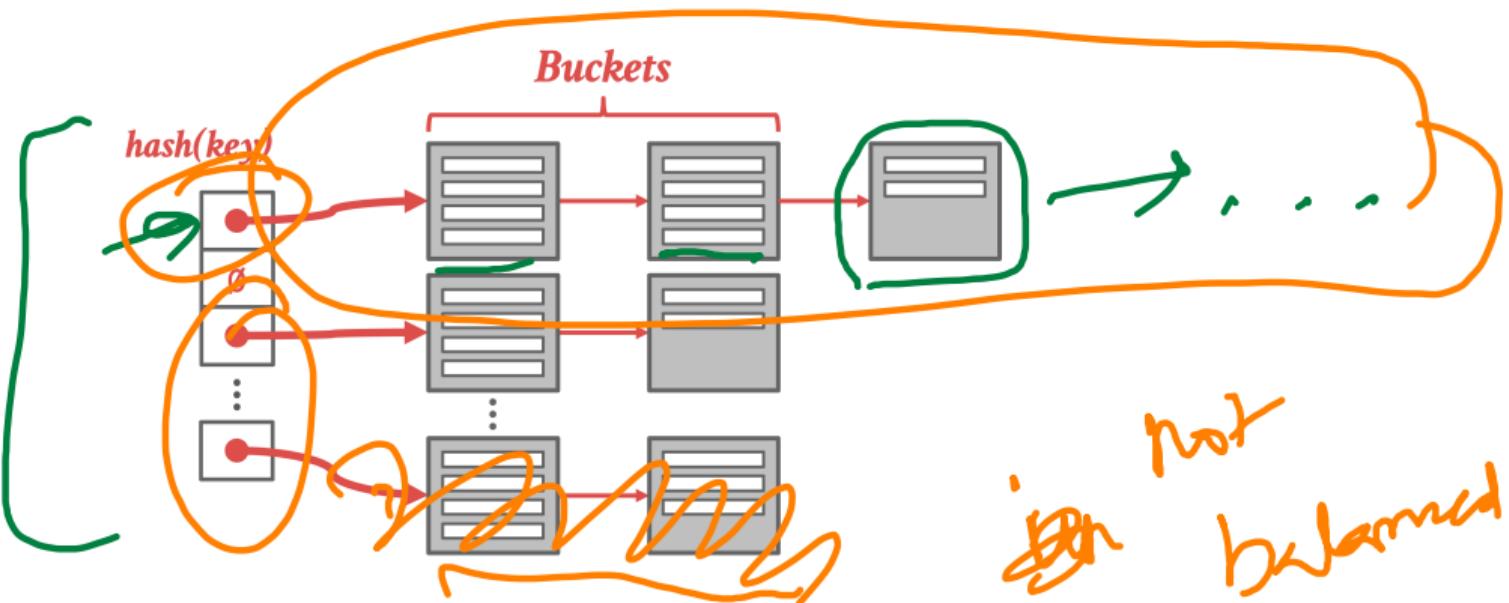
Chained Hashing
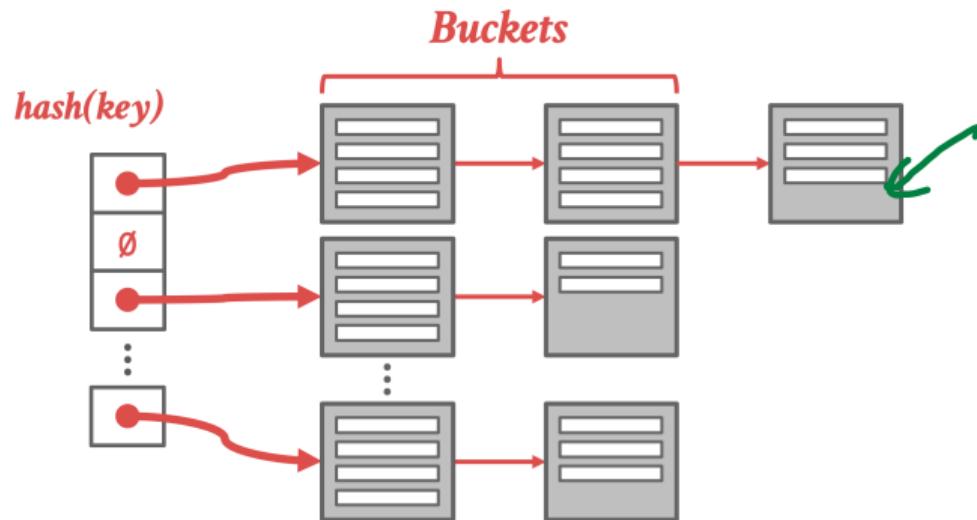
Chained Hashing

Chained Hashing

- The hash table can grow infinitely because you just keep adding new buckets to the linked list.
- You only need to take a latch on the bucket to store a new entry or extend the linked list.

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.
- Reshuffling bucket entries on split and increase the number of bits to examine.
  - Data movement is **localized** to just the split chain.

Extendible Hashing

- The slot array maps hashes to buckets.
- A hash value may occupy an arbitrary number of bits.
- With extendible hashing, the **number of bits** that the hash table uses to map hashes to buckets **changes over time**.
  - ▶ **Global counter** keeps track of the number of bits that the the hash table uses.
  - ▶ **Local counter** in each bucket tracks the number of hash bits used by that bucket.

Extendible Hashing
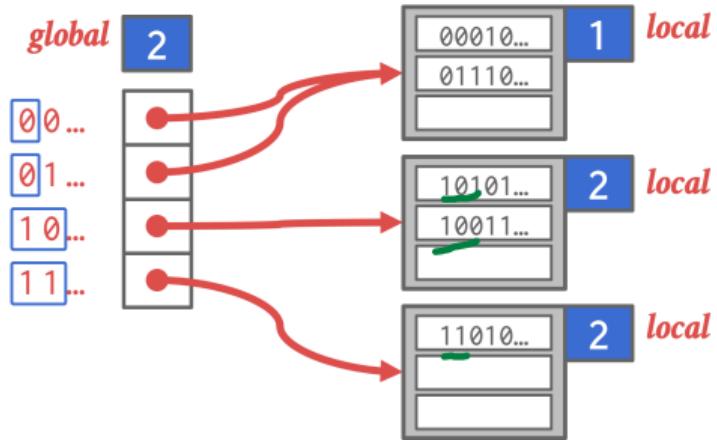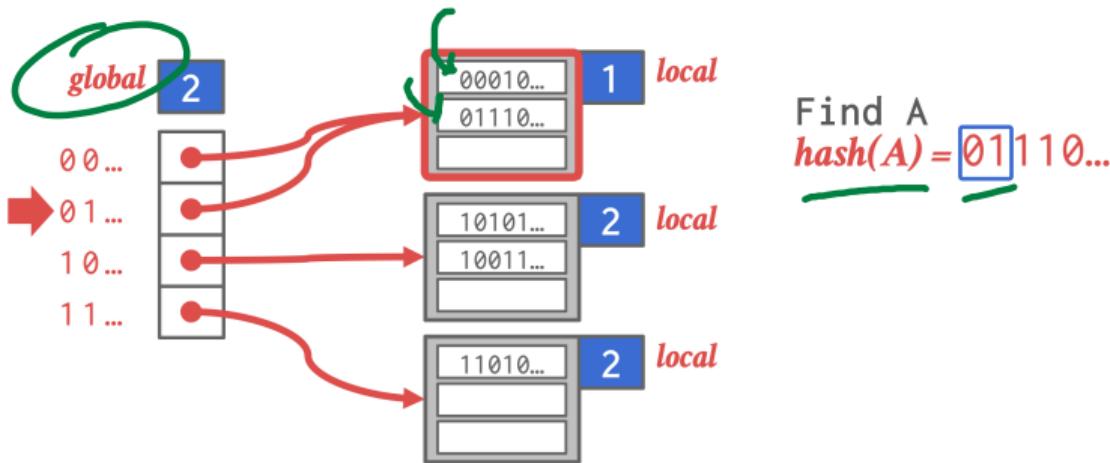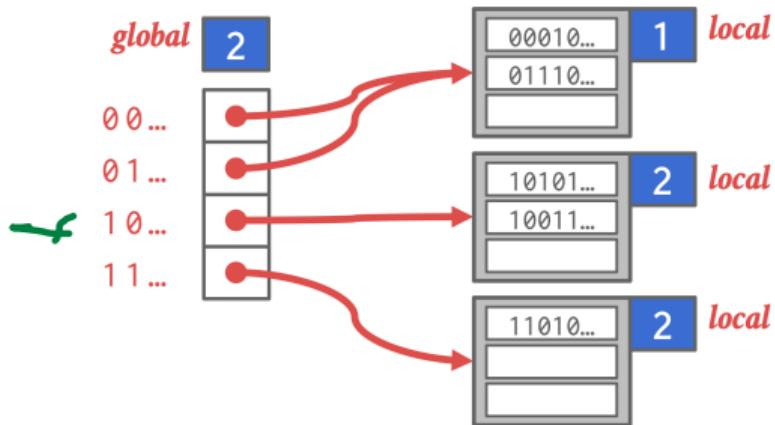
hash(key) = 01

global **2**

00...
01...
10...
11...

00010...	1
01110...	

10101...	2
10011...	

11010...	2

local

0

Extendible Hashing

Extendible Hashing

Extendible Hashing

Extendible Hashing

Extendible Hashing

Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

Extendible Hashing

Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

Extendible Hashing

global 3

```
000…
010…
100…
110…
001…
011…
101…
111…
```

00010…	1
01110…	

	3
10011…	

10101…	3
10100…	
10111…	

11010…	2
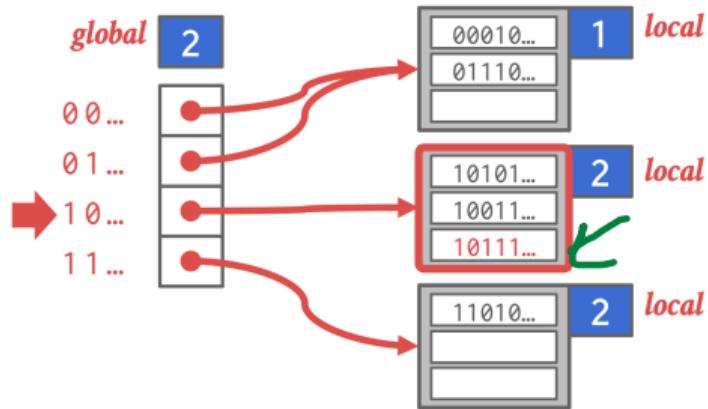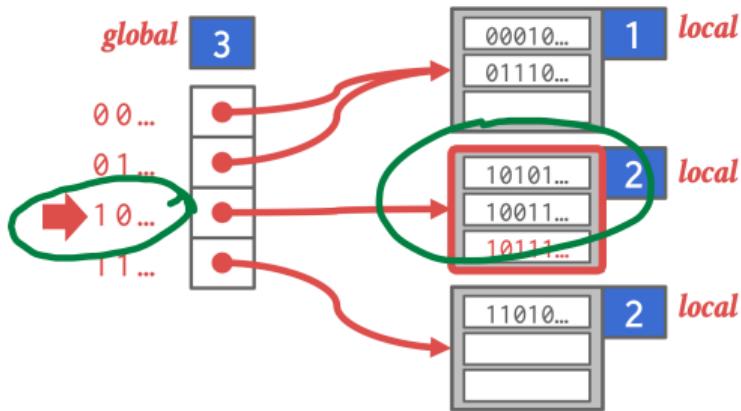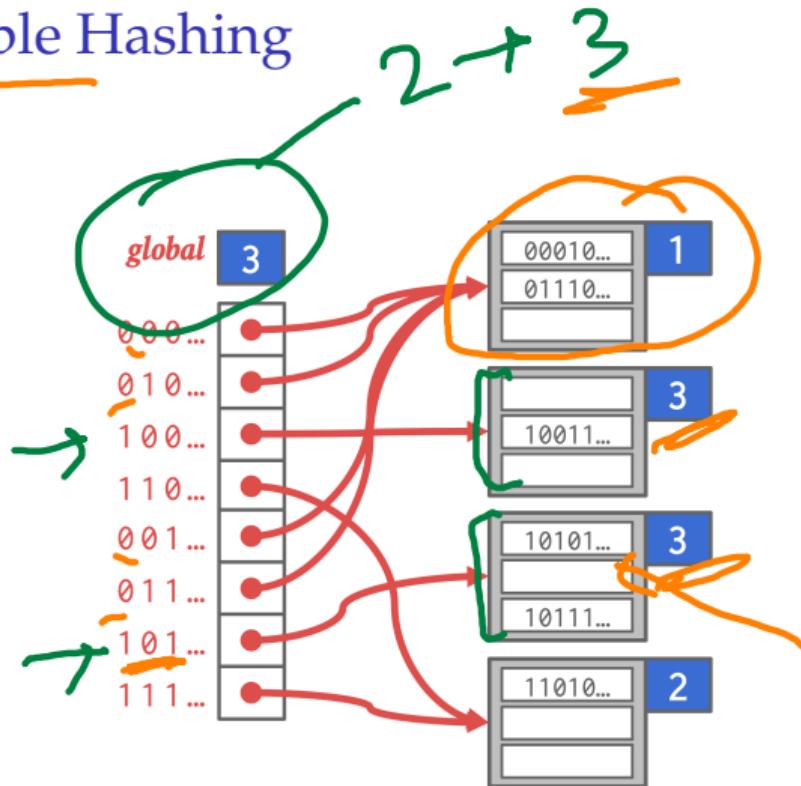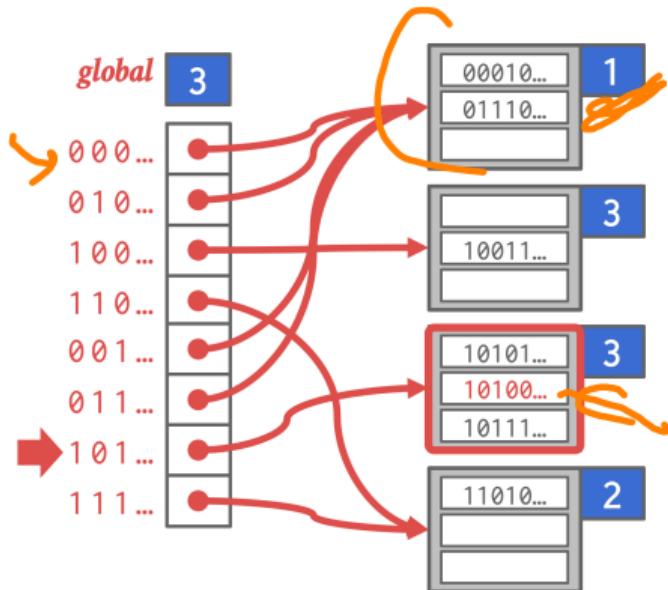
Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert 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

$20\% \cdot h = 0$

Split Pointer

0
1
2
3

8
20

5
9
13

6

7
11

$n = 4$

hash₁(key) = key % n

Linear Hashing

Linear Hashing

Split Pointer

$hash_1(key) = key \% n$

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

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

Linear Hashing

Linear Hashing

Split Pointer

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

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

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

Overflow!

Linear Hashing

Split Pointer

0
1
2
3
4

8

5
9
13

17

6

7
11

20

hash₁(key) = key % n

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

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

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

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

Linear Hashing

Split Pointer

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

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

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

Linear Hashing

Split Pointer

0
1
2
3
4

8

5
9
13

17

6

7

20

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

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

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

Find 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

Linear Hashing

Split Pointer

0
1
2
3
4

$hash_1(key) = key \% n$

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

8

5
9
13

6

7
11

17

20

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

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

Find 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

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

Linear Hashing - Delete

- Splitting buckets based on the split pointer will eventually get to all overflowed buckets.
  - ▶ When the pointer reaches the last slot, delete the first hash function and move back to beginning.
- The pointer can also **move backwards** when buckets are empty.

Linear Hashing – Delete

Split Pointer

0
1
2
3
4

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

| 8 | |

| 5 |
| 9 |
| 13 |

| 17 | |

| 6 | |

| 7 |
| 11 |

| 20 | |

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

Linear Hashing – Delete

Split
Pointer

0

1

2

3

4

$hash_1(key) = key \% n$

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

8

5
9
13

17

6

7
11

0

Delete 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

Linear Hashing – Delete

Split
Pointer

Delete 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

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

L1
L2

underflow

Linear Hashing – Delete

Split
Pointer

0
1
2
3

8	
5	9	13
17	
6	
7	11

Delete 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

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

Linear Hashing – Delete

Split Pointer

0
1
2
3

$hash_1(key) = key \% n$

8

5
9
13

17

6

7
11

Delete 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

Linear Hashing – Delete

Split
Pointer

0
1
2
3

$hash_1(key) = key \% n$

8

5
9
13

17
21

6

7
11

Overflow!

Delete 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

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

Linear Hashing vs. Extendible Hashing

2 bits → 3 bits

- Moving from hash_i to hash_{i+1} in Linear Hashing corresponds to
- Bumping up the global counter in Extendible Hashing
- Linear Hashing
  - Directory is gradually doubled over the course of a round
  - A directory can be avoided by a clever choice of the buckets to split
  - More flexibility: need not always split the appropriate dense bucket

Conclusion

STA :: unordued-map

- Hash tables are fast data structures that support O(1) look-ups
- Used all throughout the DBMS internals.
  - ▶ Examples: Page Table (Buffer Manager), Lock Table (Lock Manager)
- Trade-off between speed and flexibility.

Conclusion

- Hash tables are usually **<u>not</u>** what you want to use for a indexing tables
  - ▶ Lack of ordering in widely-used hashing schemes
  - ▶ Lack of locality of reference $\longrightarrow$ more disk seeks
  - ▶ Persistent data structures are much more complex (logging and recovery)
  - ▶ Reference
- We will cover B+Trees in the next lecture
  - ▶ *a.k.a.,* "The Greatest Data Structure of All Time!"