Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

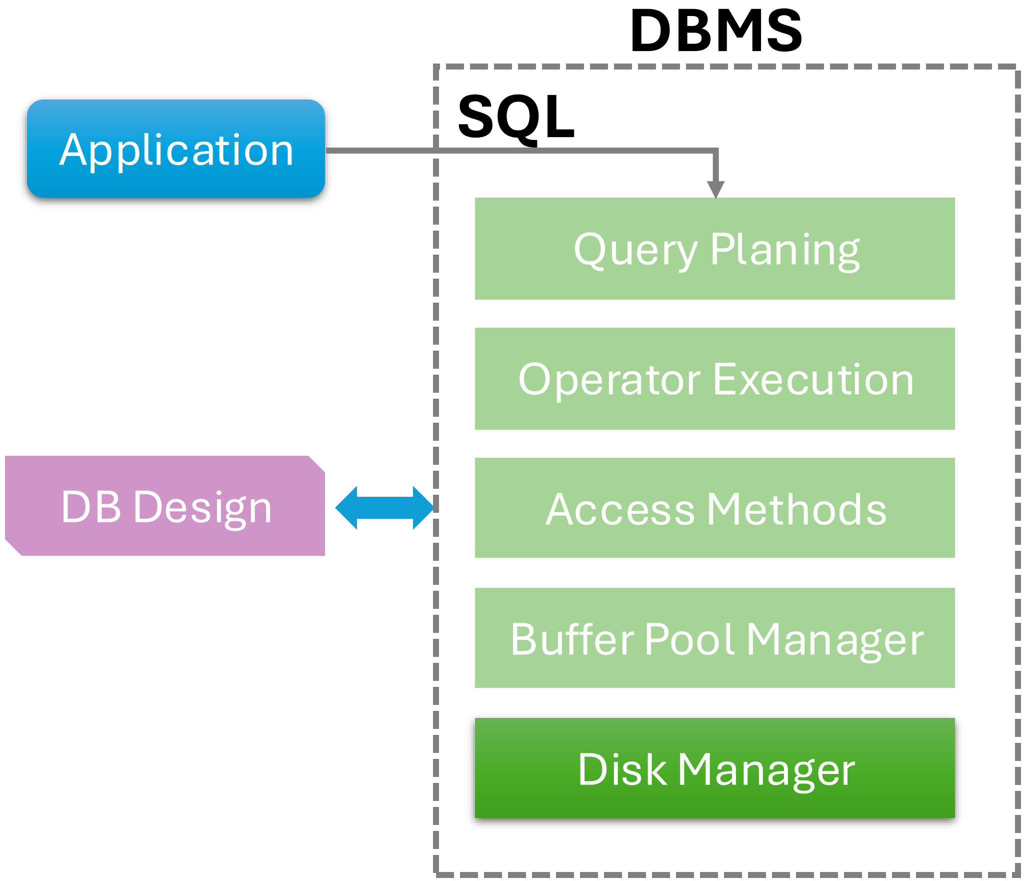
CSC3170
7: Hash Tables

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

Course Status

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

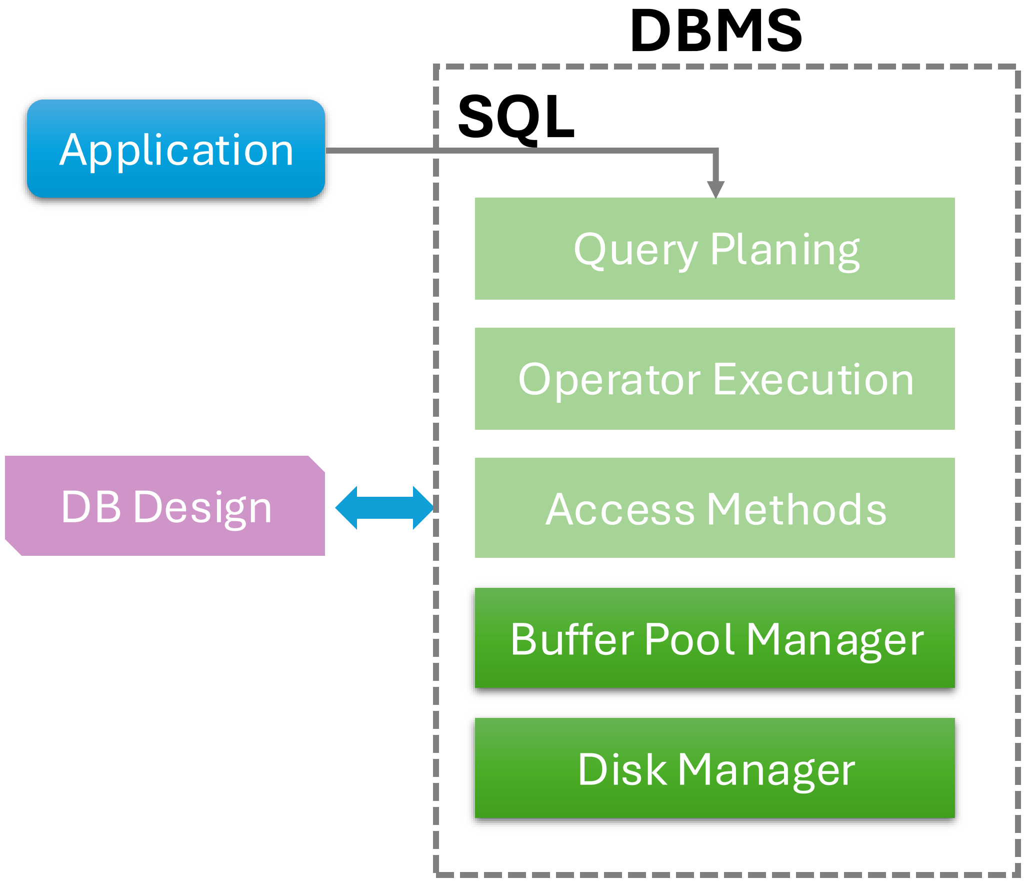
- Two types of data structures:
  - Hash Tables (Unordered)
  - Trees (Ordered)



Course Status

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

- Two types of data structures:
  - Hash Tables (Unordered)
  - Trees (Ordered)



Data Structures

- Internal Meta-data
- Core Data Storage
- Temporary Data Structures
- Table Indexes

Design Decisions

- **Data Organization**
  - How we layout data structure in memory/pages and what information to store to support efficient access.
- **Concurrency**
  - How to enable multiple threads to access the data structure at the same time without causing problems.

Hash Tables

- A **hash table** implements an unordered associative array that maps keys to values.
- It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.
- Space Complexity: $\boldsymbol{O(n)}$
  Time Complexity:
  - Average: $\boldsymbol{O(1)}$
  - Worst: $\boldsymbol{O(n)}$

Hash Tables

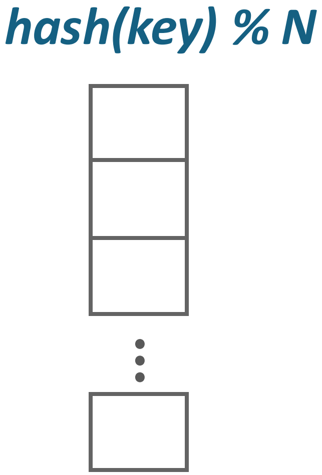
- A **hash table** implements an unordered associative array that maps keys to values.
- It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.
- Space Complexity: $\boldsymbol{O(n)}$
  Time Complexity:
  - Average: $\boldsymbol{O(1)}$
  - Worst: $\boldsymbol{O(n)}$

⬅ ***Databases care about constants!***

Static Hash Table

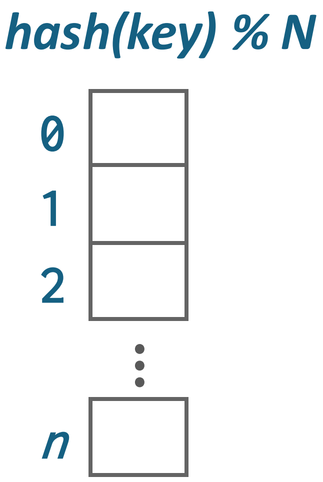
- Allocate a giant array that has one slot for every element you need to store.

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



Static Hash Table

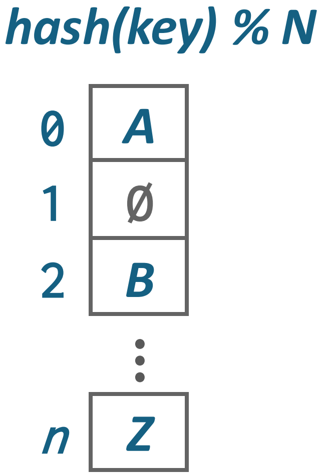
- Allocate a giant array that has one slot for every element you need to store.
- To find an entry, mod the key by the number of elements to find the offset in the array.



Static Hash Table

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

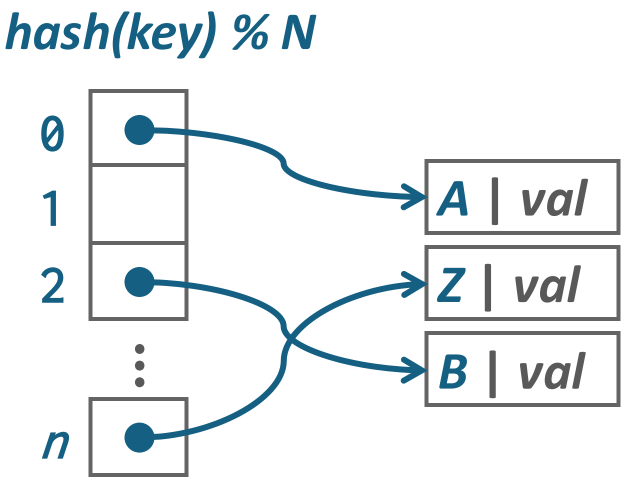
- To find an entry, mod the key by the number of elements to find the offset in the array.



Static Hash Table

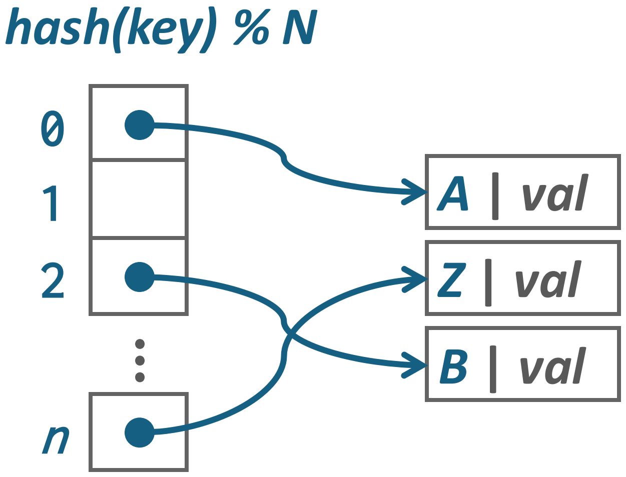
- Allocate a giant array that has one slot for every element you need to store.

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



Unrealistic Assumptions

- **Assumption #1:** Number of elements is known ahead of time and fixed.
- **Assumption #2:** Each key is unique.
- **Assumption #3:** Perfect hash function guarantees no collisions.
  - If key1≠key2, then hash(key1)≠hash(key2)



Hash Table

- **Design Decision #1: Hash Function**
  - How to map a large key space into a smaller domain.
  - Trade-off between being fast vs. collision rate.
- **Design Decision #2: Hashing Scheme**
  - How to handle key collisions after hashing.
  - Trade-off between allocating a large hash table vs. additional instructions to get/put keys.

This Lecture

- Hash Functions
- Static Hashing Schemes
- Dynamic Hashing Schemes

Hash Functions

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.
- **Google FarmHash (2014)**
    - Newer version of CityHash with better collision rates.

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

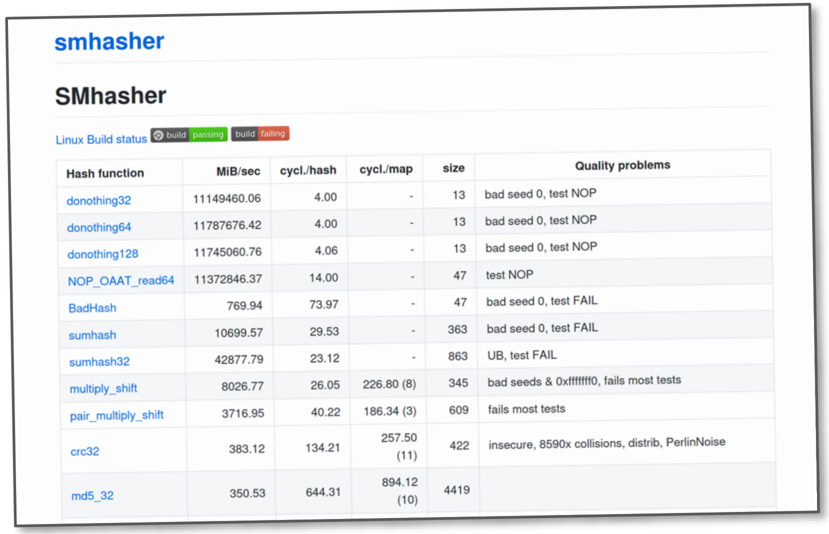
- Google FarmHash **(2014)**
  - Newer version of CityHash with better collision rates.

Hash Functions

- 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.



smhasher

SMhasher

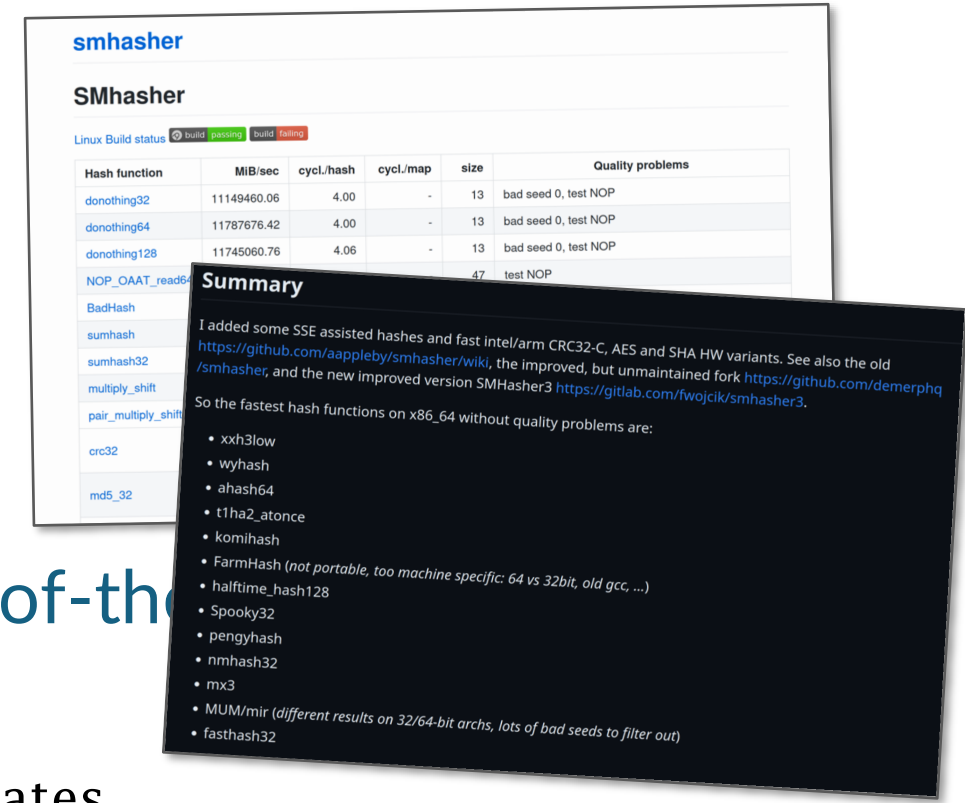
Linux Build status build passing build failing

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

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.
- Google FarmHash **(2014)**
  - Newer version of CityHash with better collision rates.

← State-of-the



smhasher

SMhasher

Linux Build status build passing build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
donothing32	11149460.06	4.00	-	13	bad seed 0, test NOP
donothing64	11787676.42	4.00	-	13	bad seed 0, test NOP
donothing128	11745060.76	4.06	-	13	bad seed 0, test NOP
NOP_OAAT_read64				47	test NOP
BadHash					
sumhash					
sumhash32					
multiply_shift					
pair_multiply_shift					
crc32					
md5_32					

Summary

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

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

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

Static Hashing Schemes

- **Approach #1: Linear Probe Hashing**
- **Approach #2: Cuckoo Hashing**
- There are several other schemes (not covered in this course):
    - Robin Hood Hashing
    - Hopscotch Hashing
    - Swiss Tables

Static Hashing Schemes

- **Approach #1: Linear Probe Hashing**
- **Approach #2: Cuckoo Hashing**

← Both are members of a broader class called "**Open Addressing**".

- There are several other schemes (not covered in this course):
  - Robin Hood Hashing
  - Hopscotch Hashing
  - Swiss Tables

Linear Probe Hashing

Static Hashing Schemes

Linear Probe Hashing

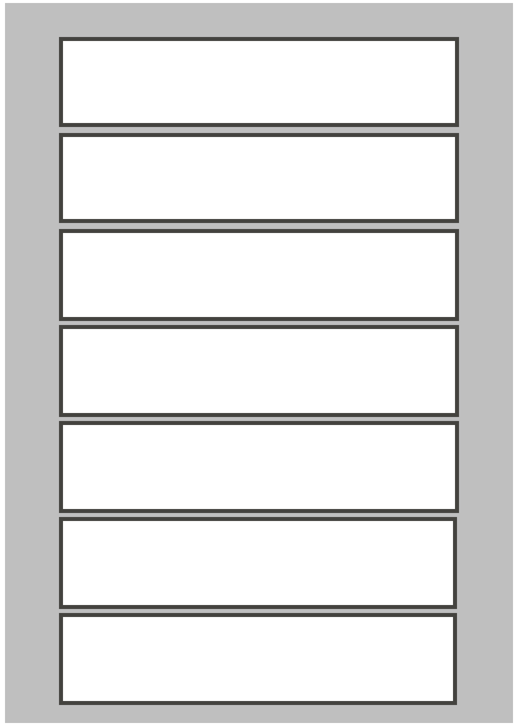
- Single giant table of slots.
- Resolve collisions by linearly searching for the next free slot in the table.
  - To determine whether an element is present, hash to a location in the index and scan for it.
  - Must store the key in the index to know when to stop scanning.
  - Insertions and deletions are generalizations of lookups.

- **Example**: Google's `absl::flat_hash_map`

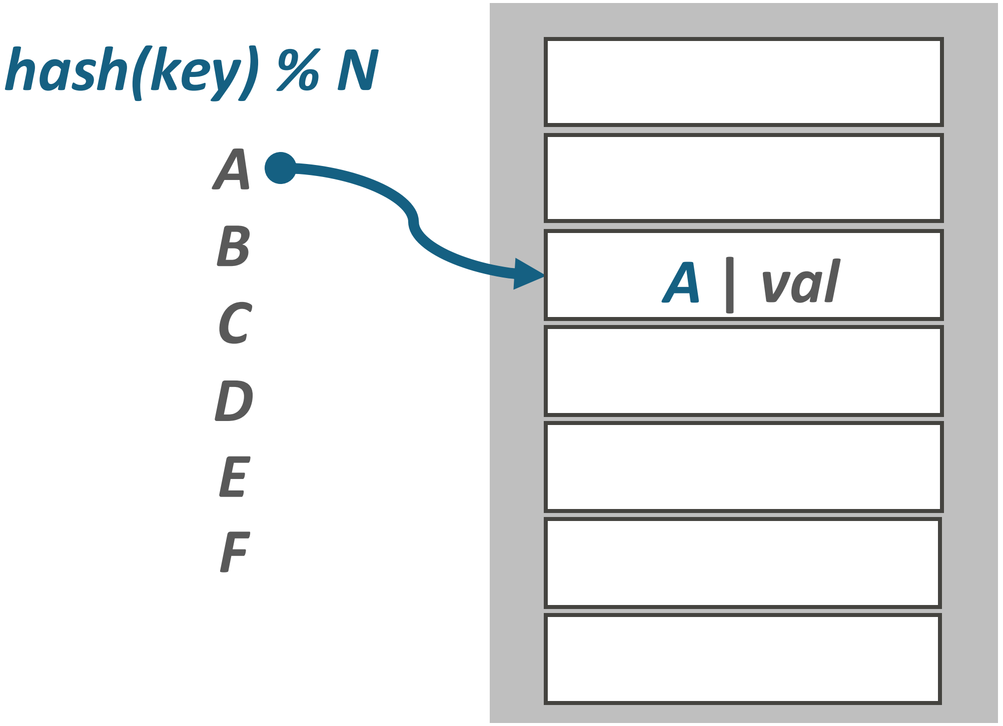
Linear Probe Hashing

hash(key) % N

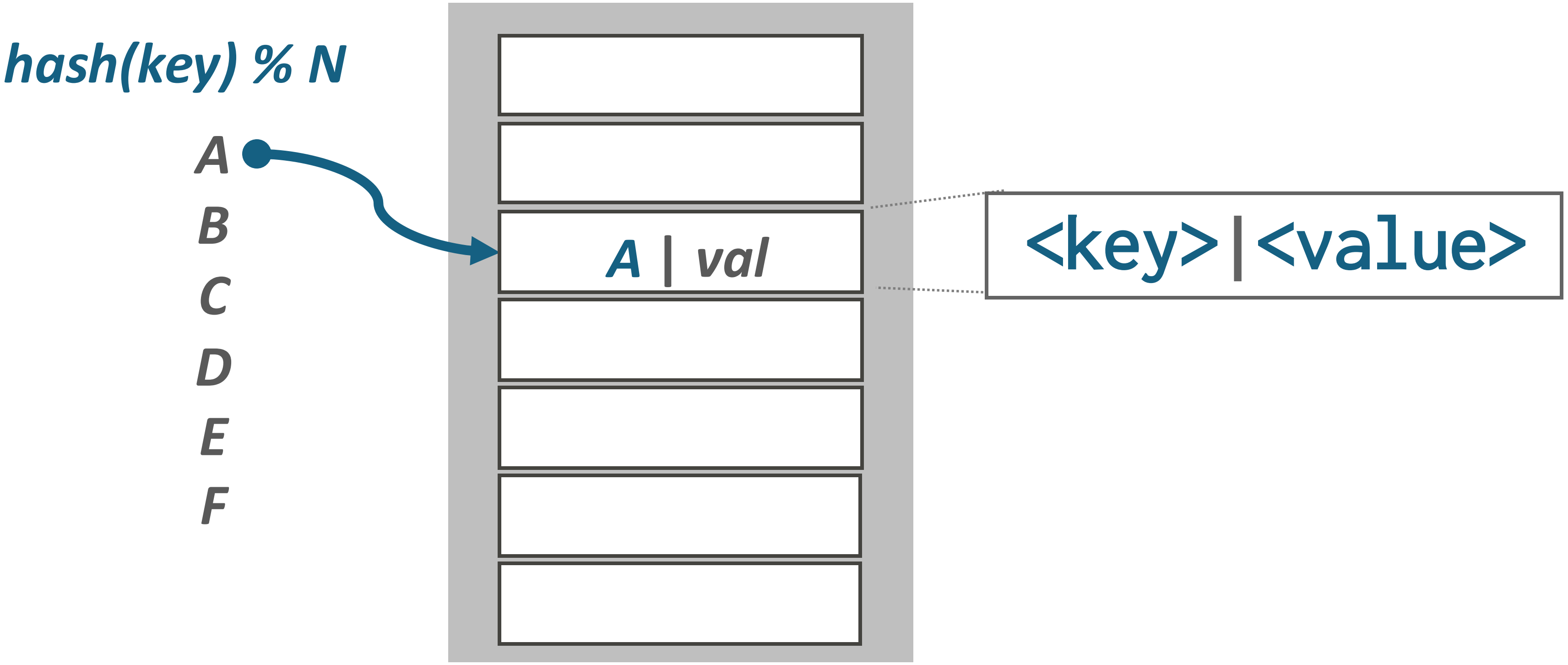
A
B
C
D
E
F



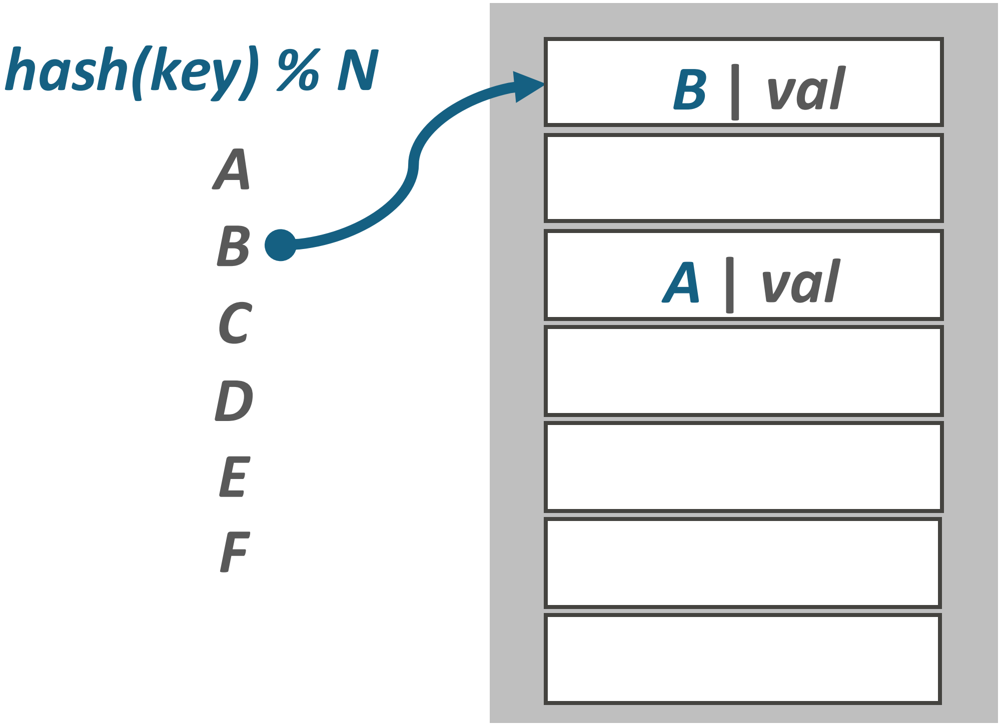
Linear Probe Hashing



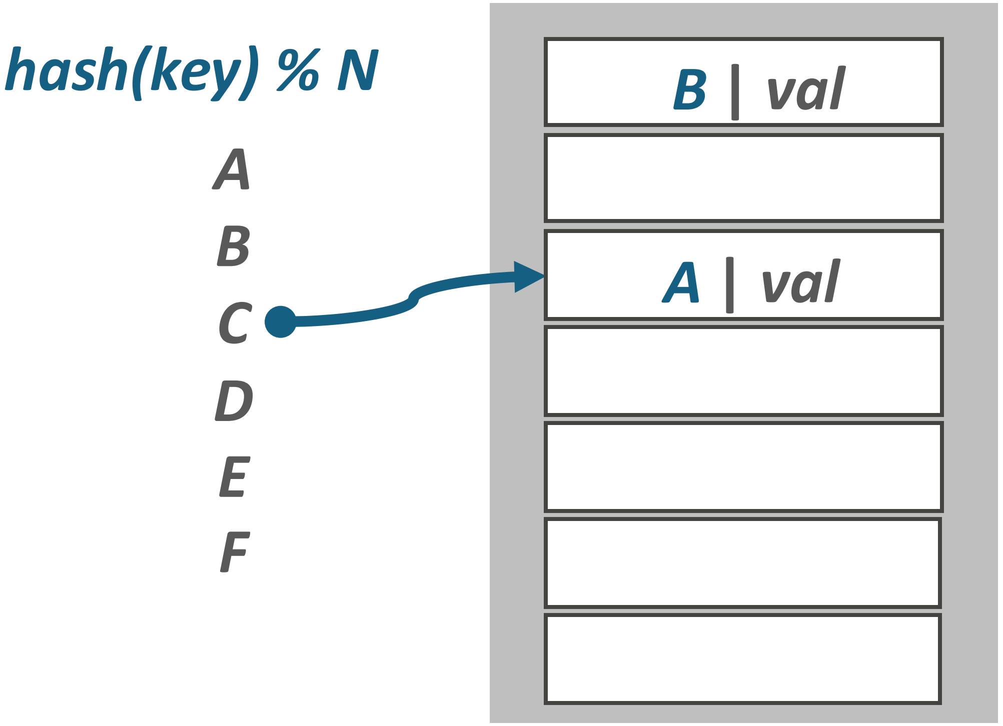
Linear Probe Hashing



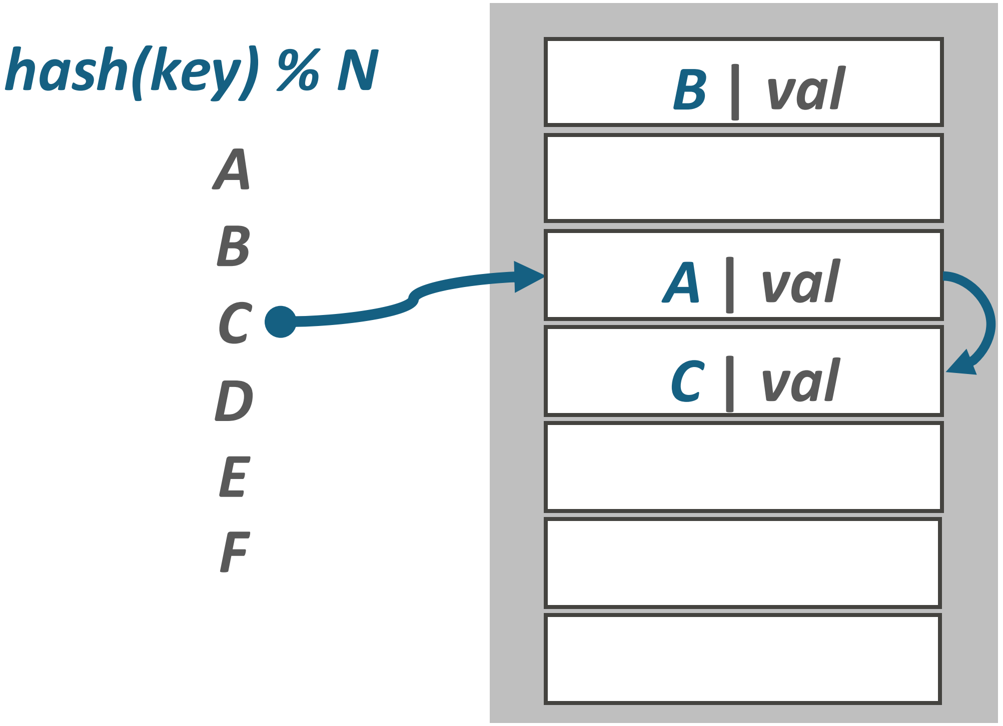
Linear Probe Hashing



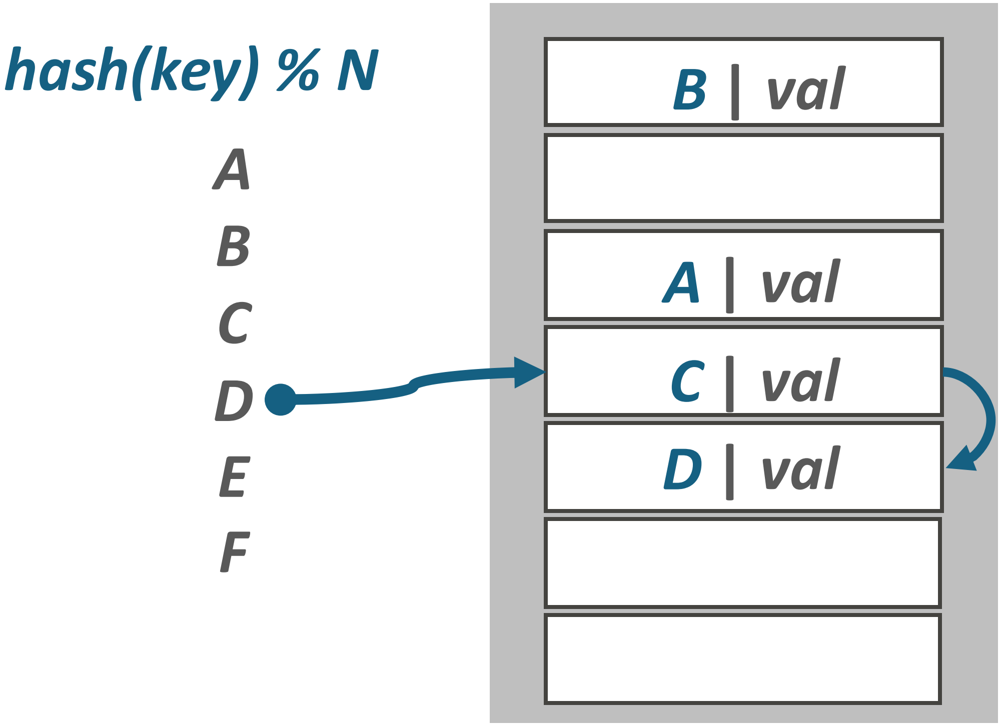
Linear Probe Hashing



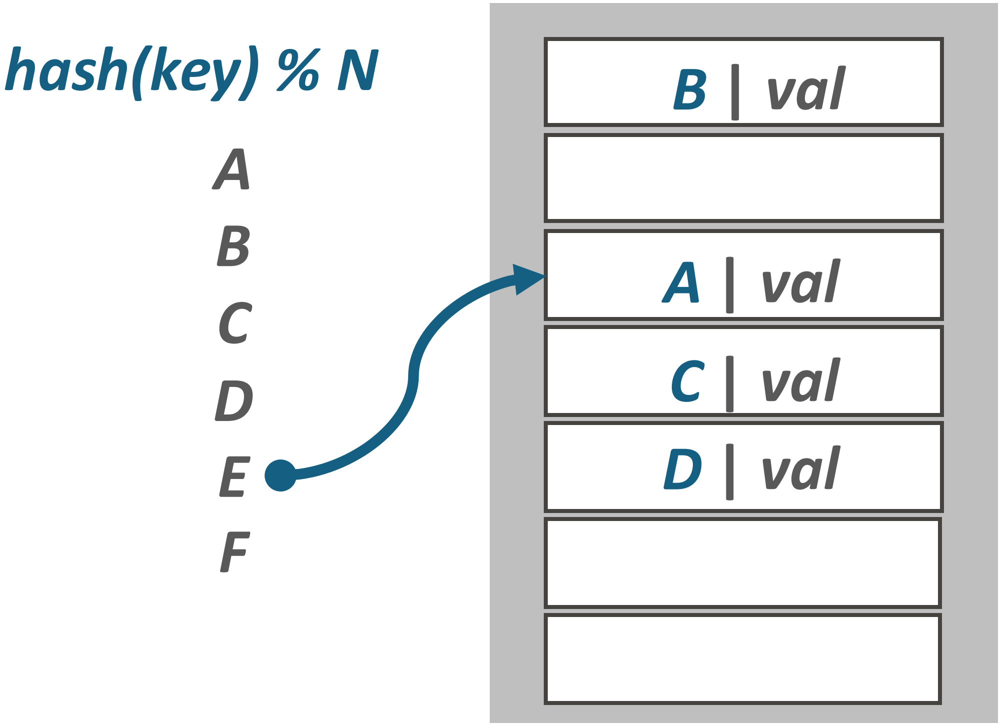
Linear Probe Hashing



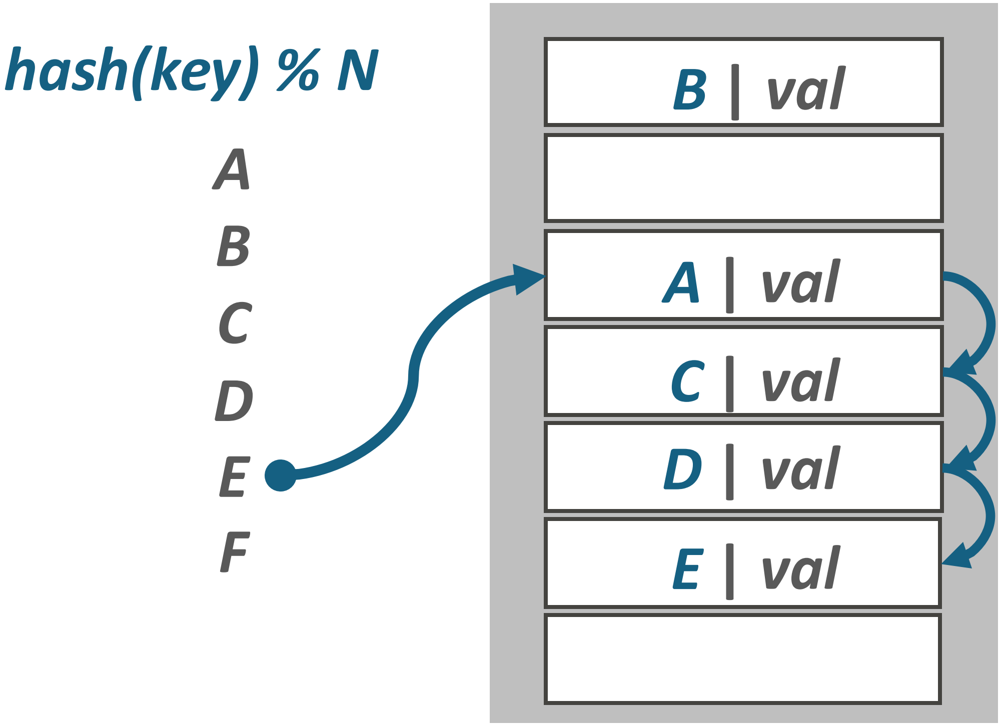
Linear Probe Hashing



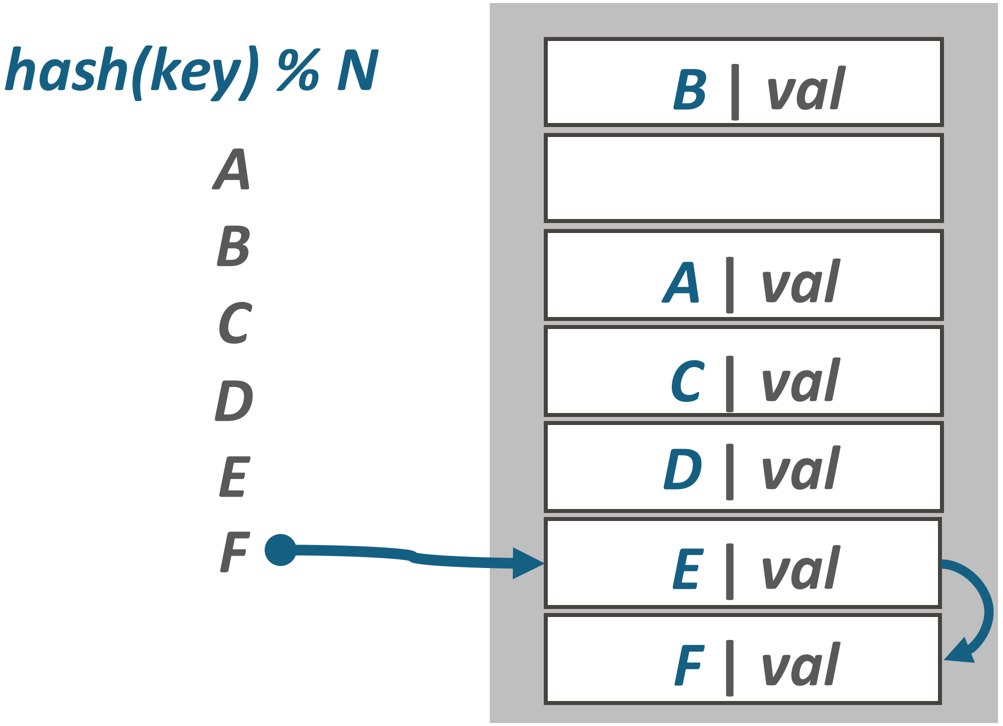
Linear Probe Hashing



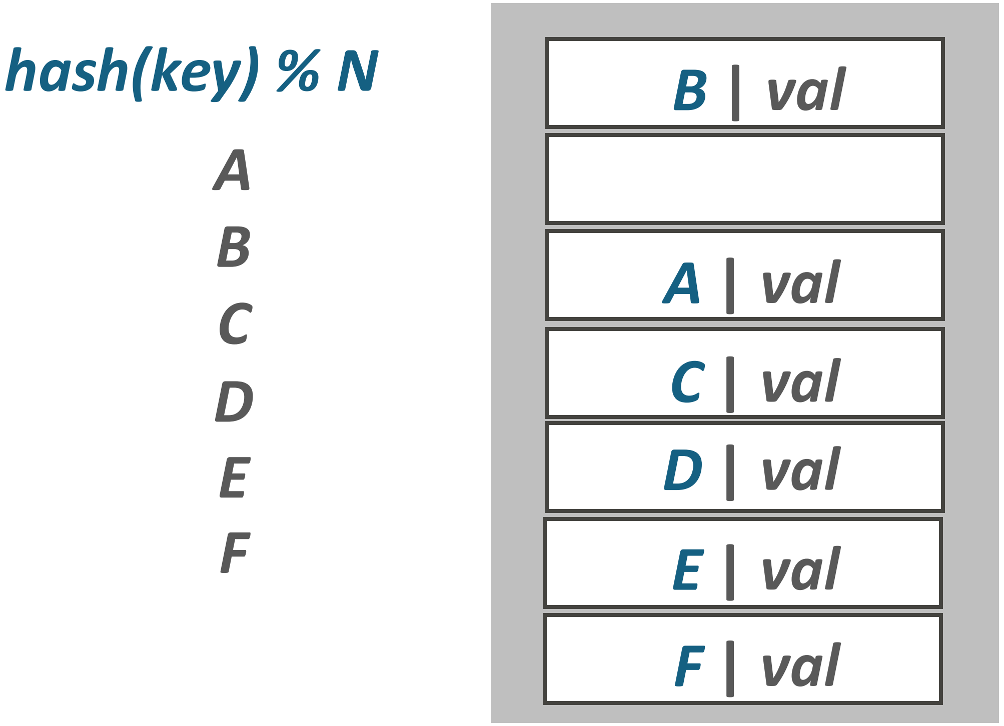
Linear Probe Hashing



Linear Probe Hashing



Linear Probe Hashing - Deletes



hash(key) % N

A
B
C
D
E
F

B | val

A | val

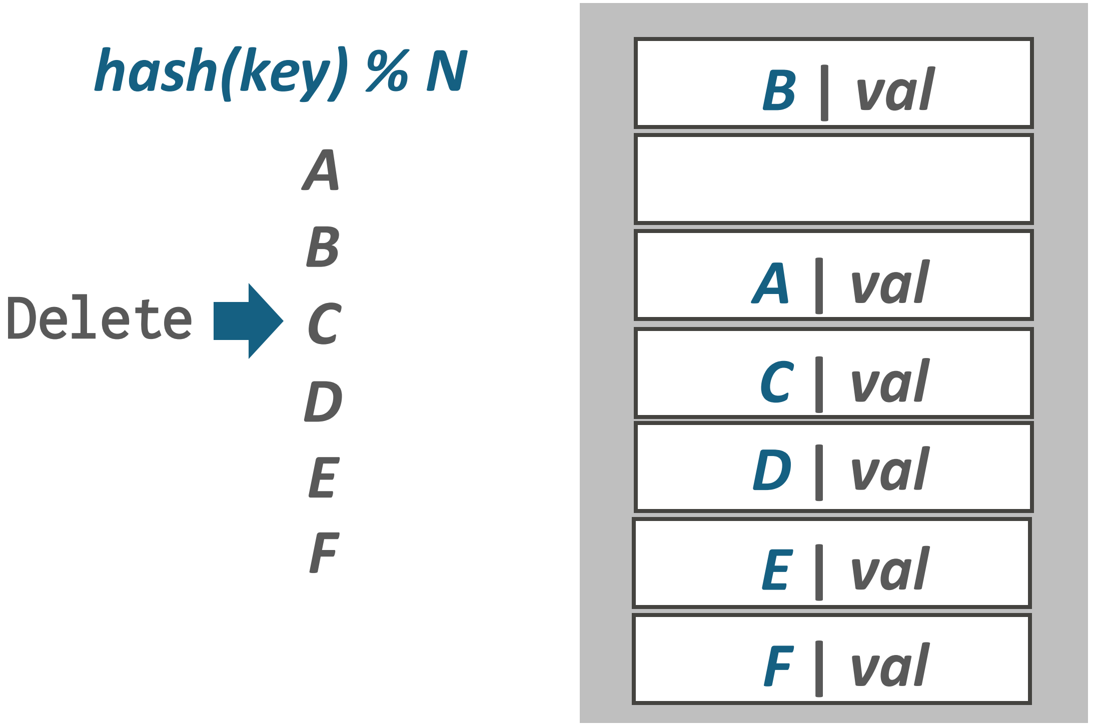
C | val

D | val

E | val

F | val

Linear Probe Hashing - Deletes



hash(key) % N

A

B

Delete *C*

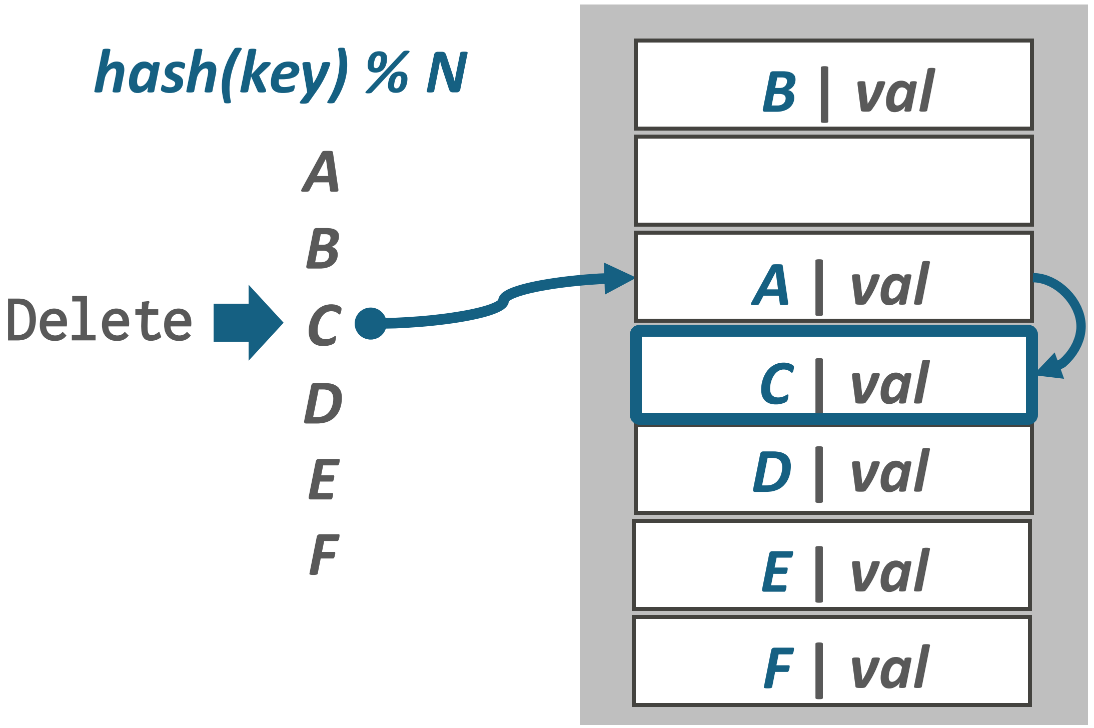
D

E

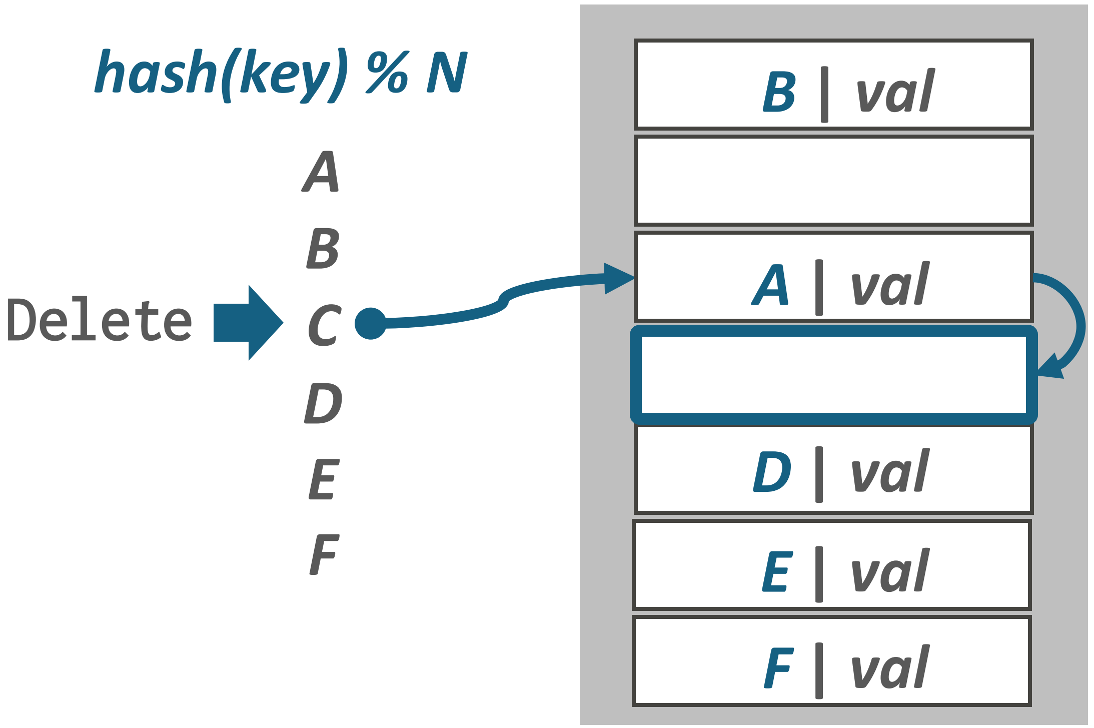
F

B | val
A | val
C | val
D | val
E | val
F | val

Linear Probe Hashing - Deletes



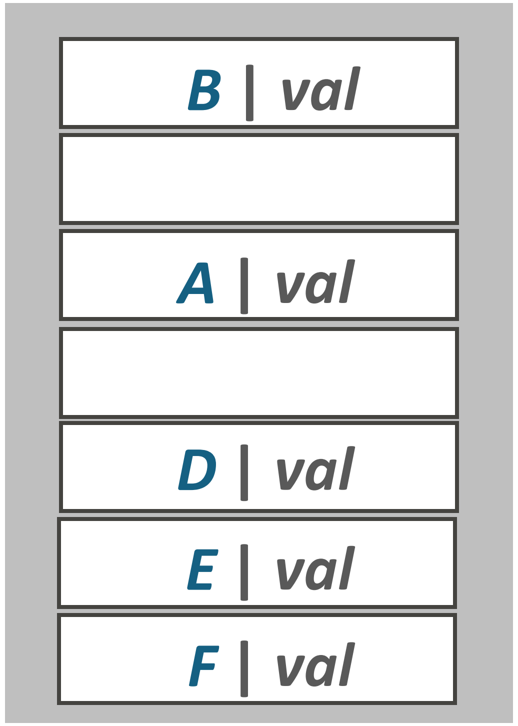
Linear Probe Hashing - Deletes



Linear Probe Hashing - Deletes

hash(key) % N

A
B
C
D
E
F



B | val

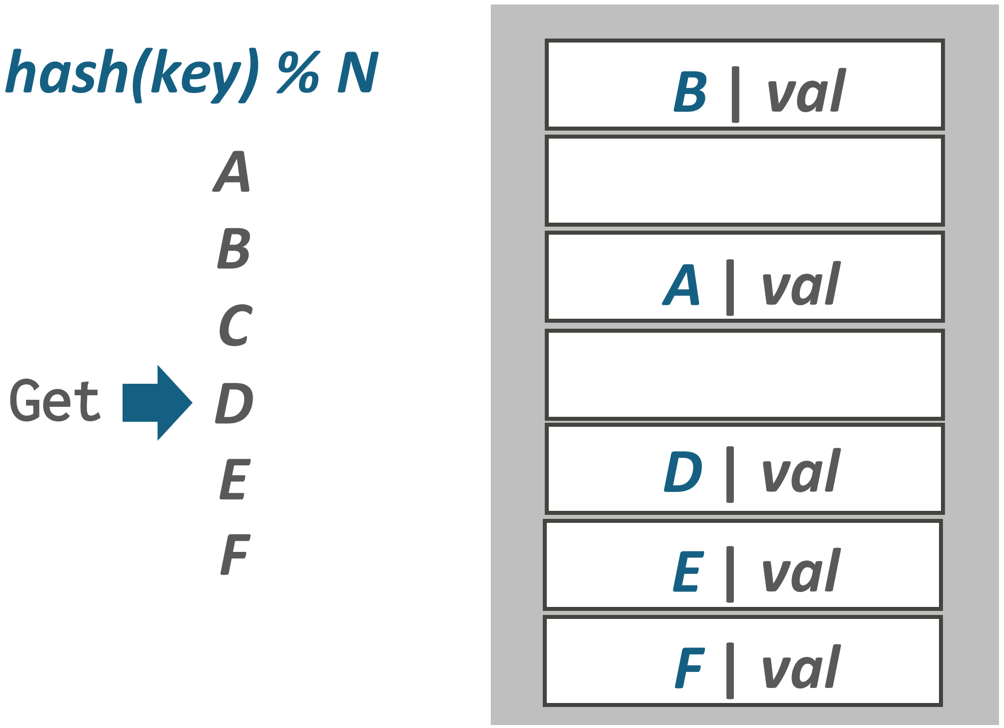
A | val

D | val

E | val

F | val

Linear Probe Hashing - Deletes



hash(key) % N

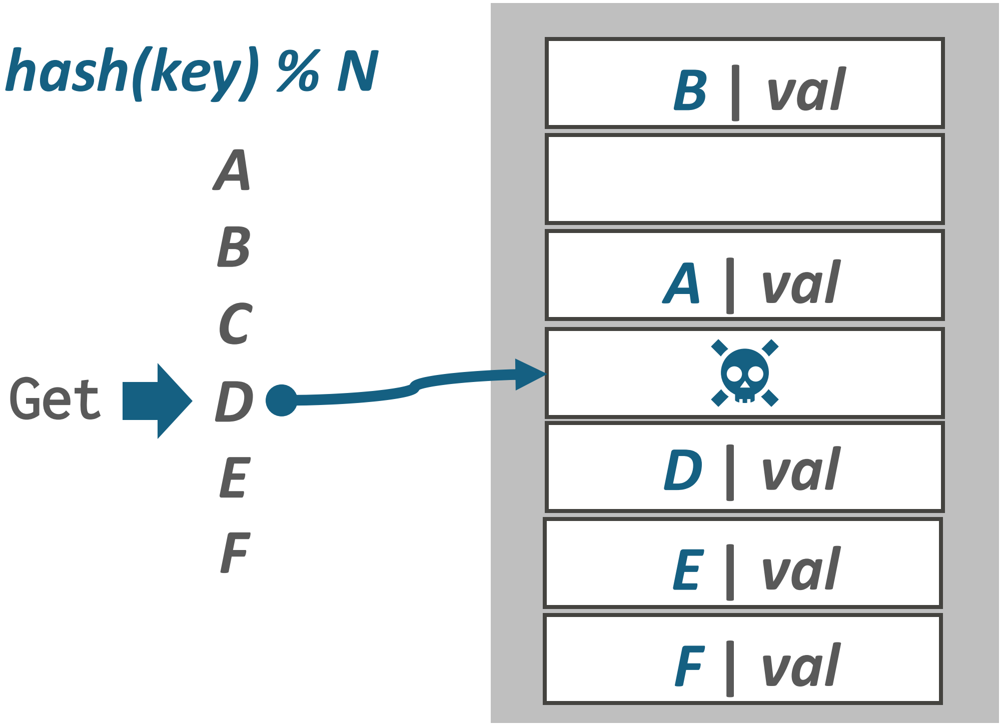
A
B
C
Get *D*
E
F

B | val

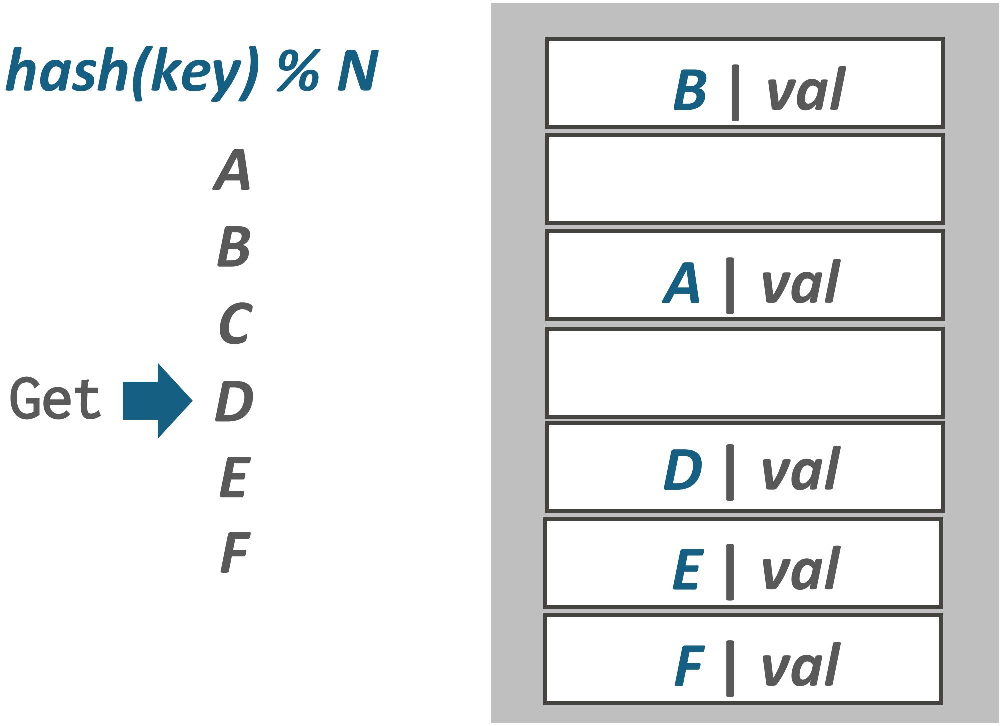
A | val

D | val
E | val
F | val

Linear Probe Hashing - Deletes



Linear Probe Hashing - Deletes

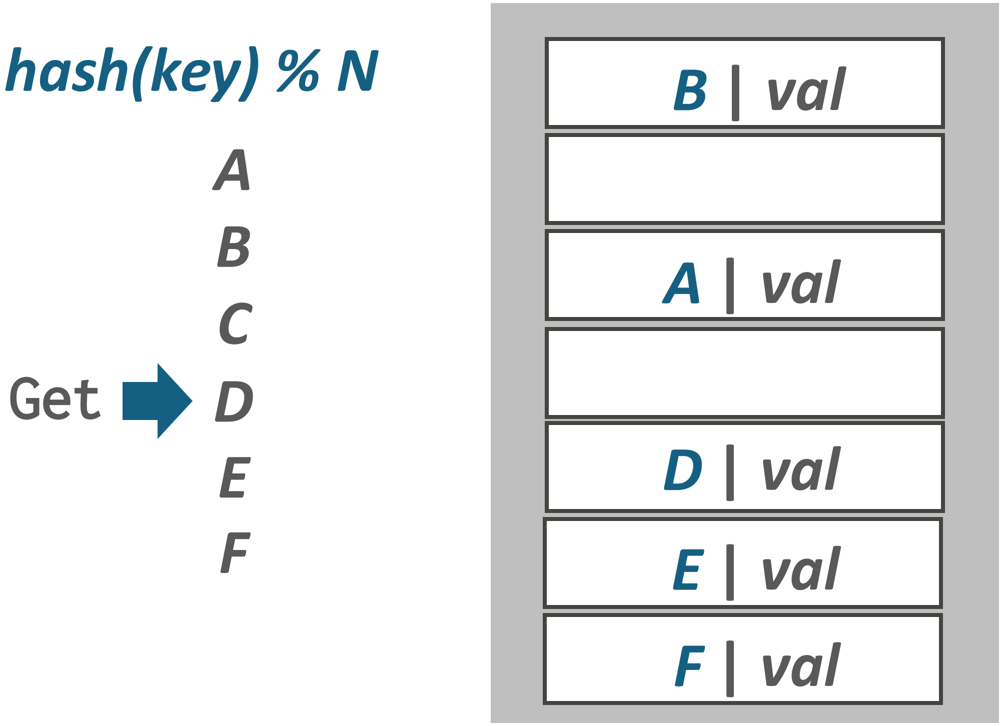


hash(key) % N

A
B
C
Get *D*
E
F

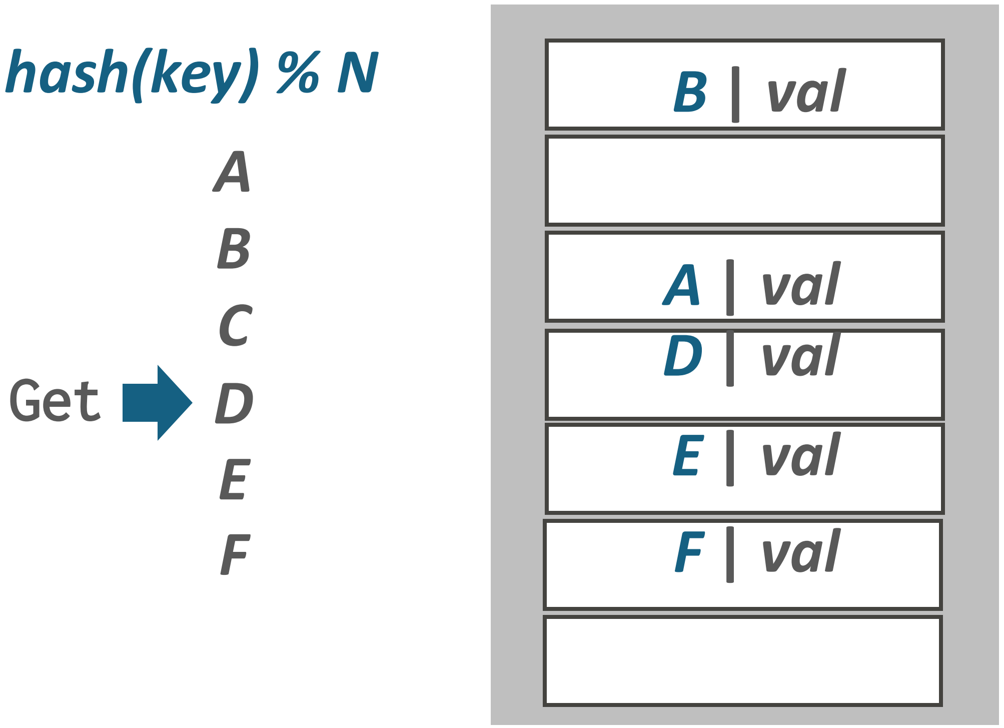
B | val
A | val
D | val
E | val
F | val

Linear Probe Hashing - Deletes



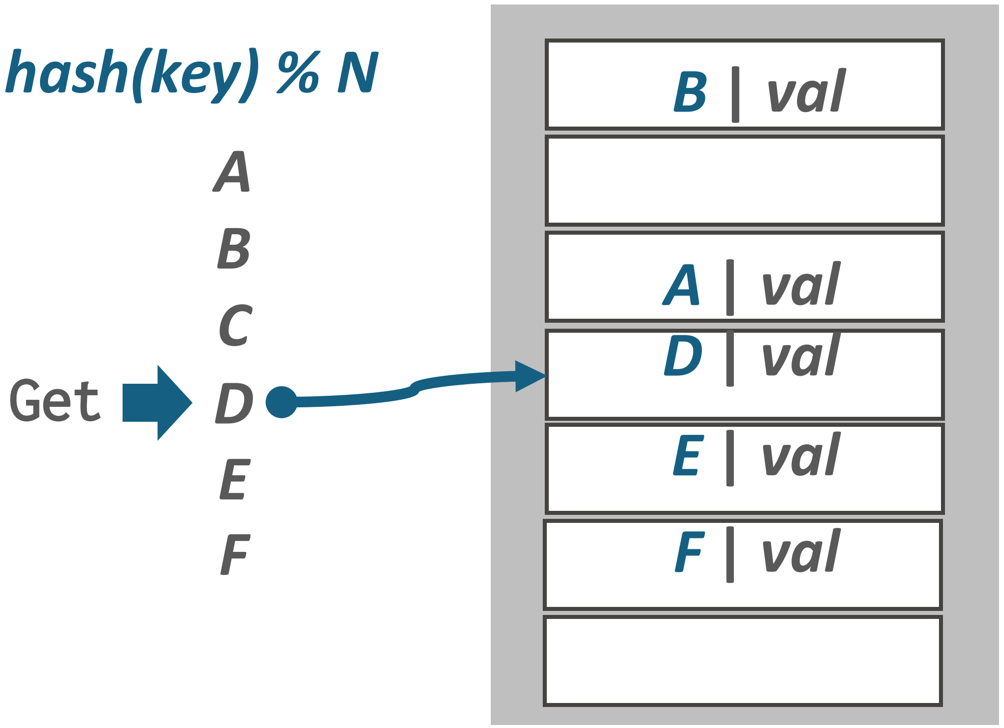
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



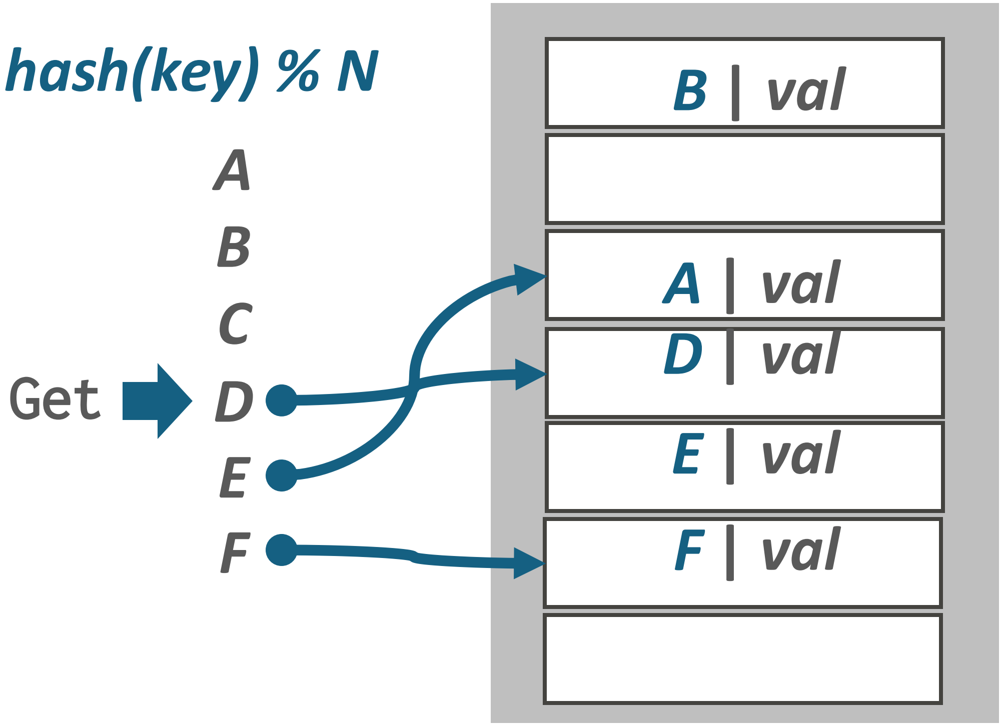
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



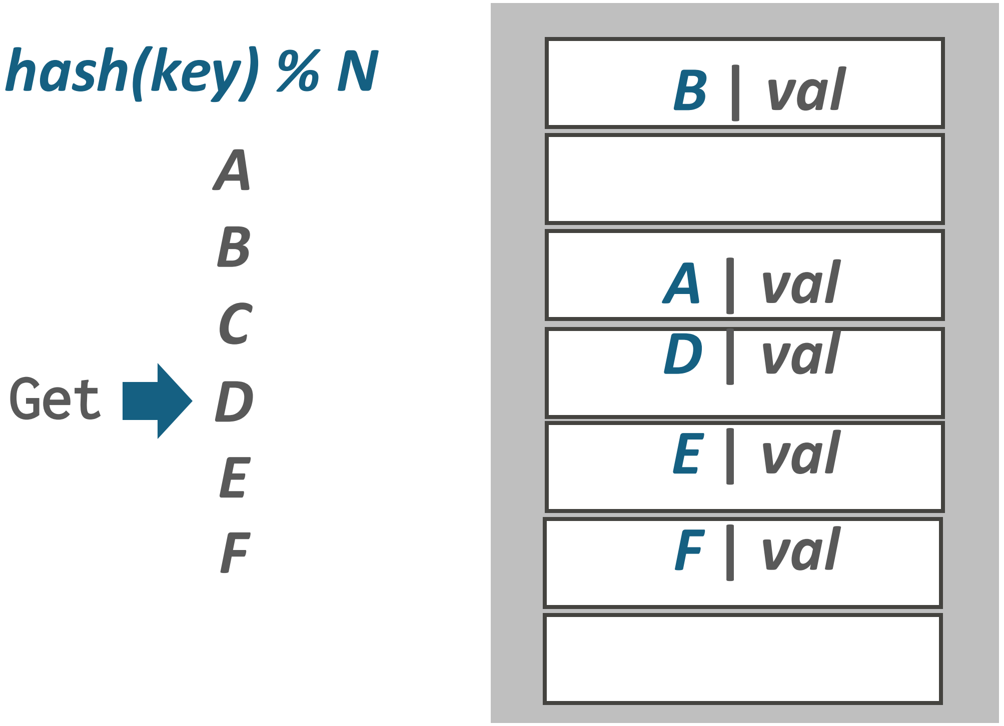
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



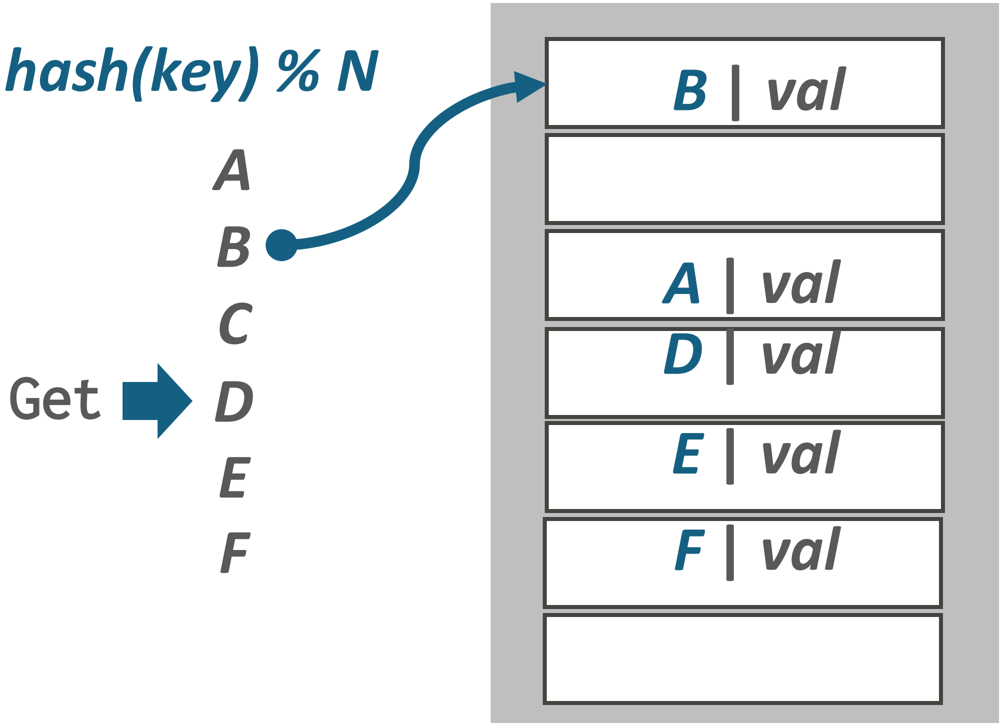
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



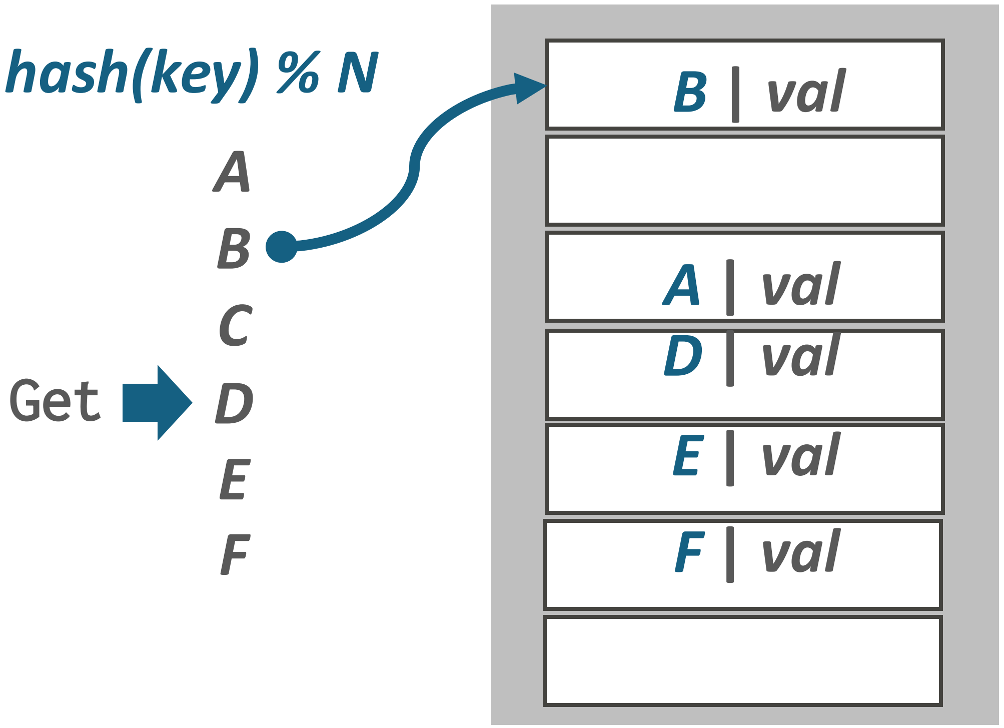
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



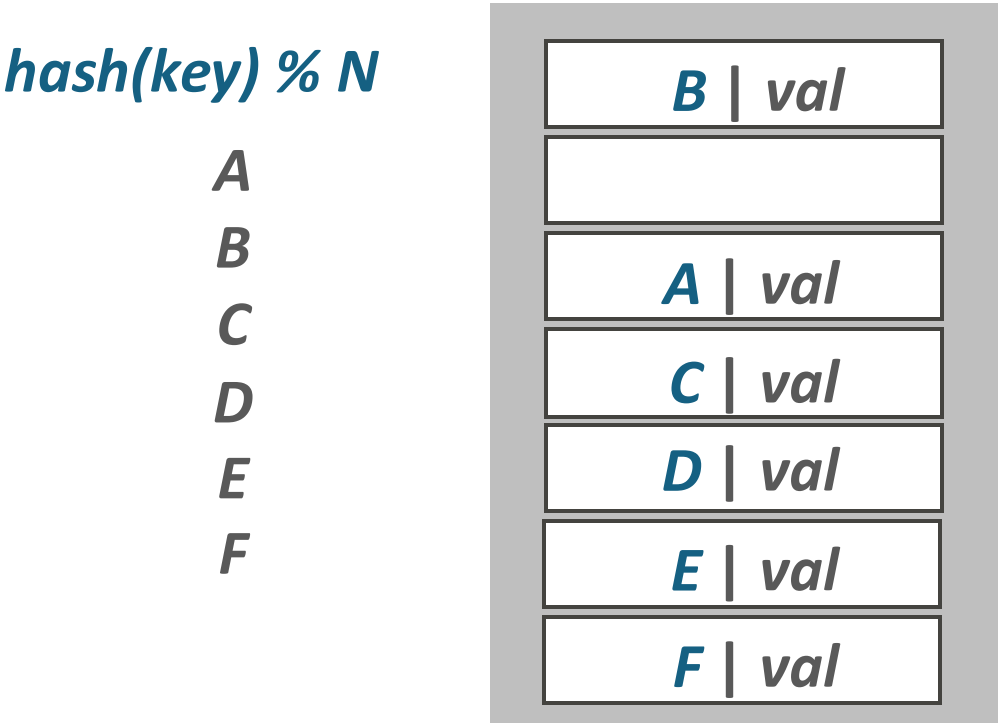
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.

Linear Probe Hashing - Deletes



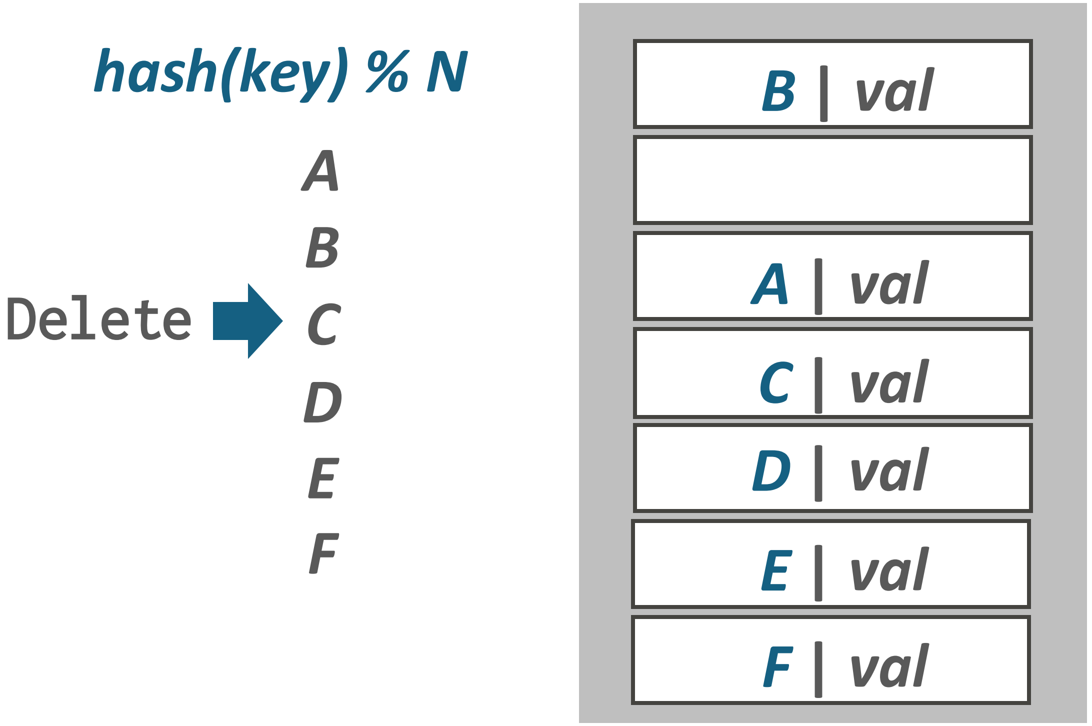
- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.
  - Expensive! May need to reorganize the entire table.
  - No DBMS does this.

Linear Probe Hashing - Deletes



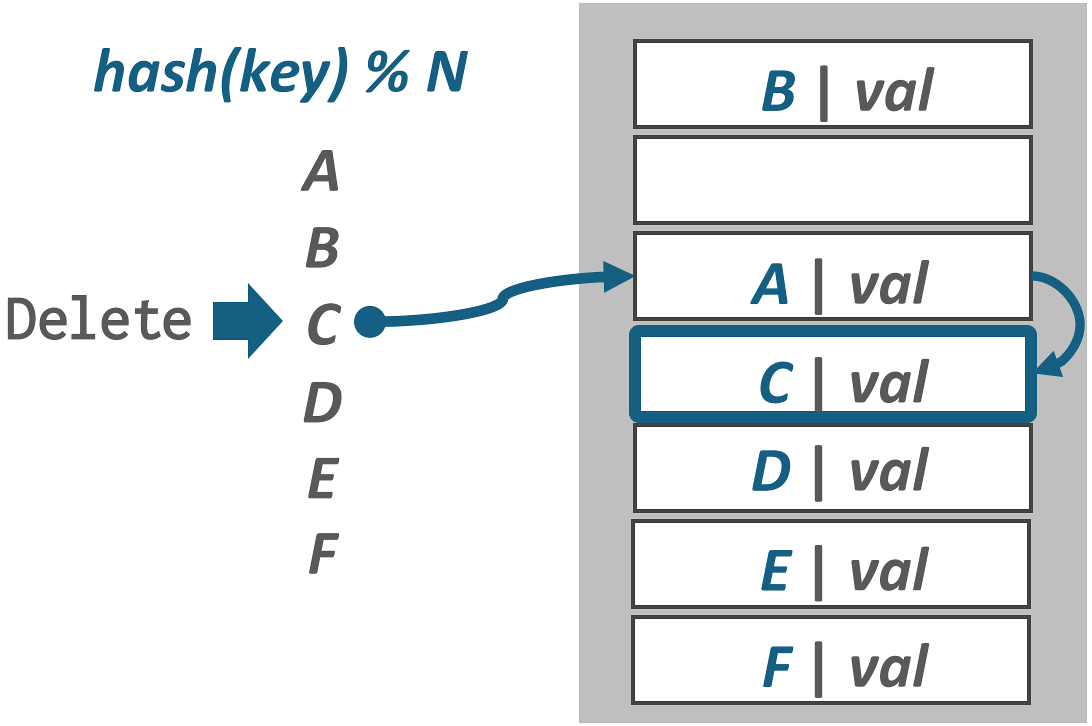
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



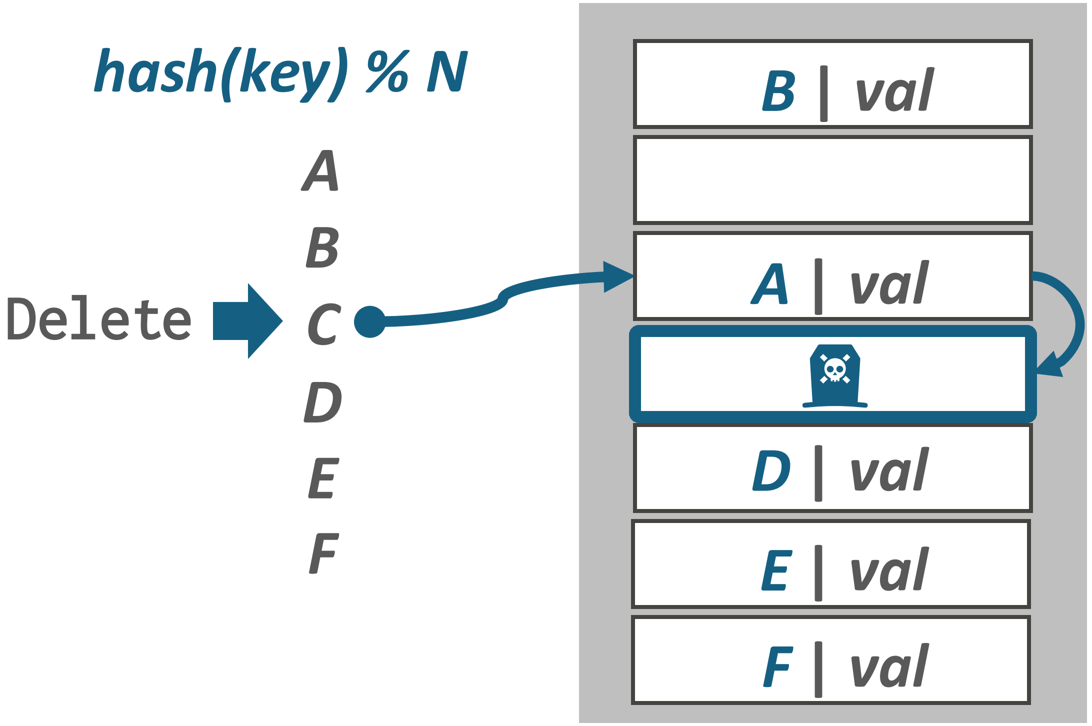
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



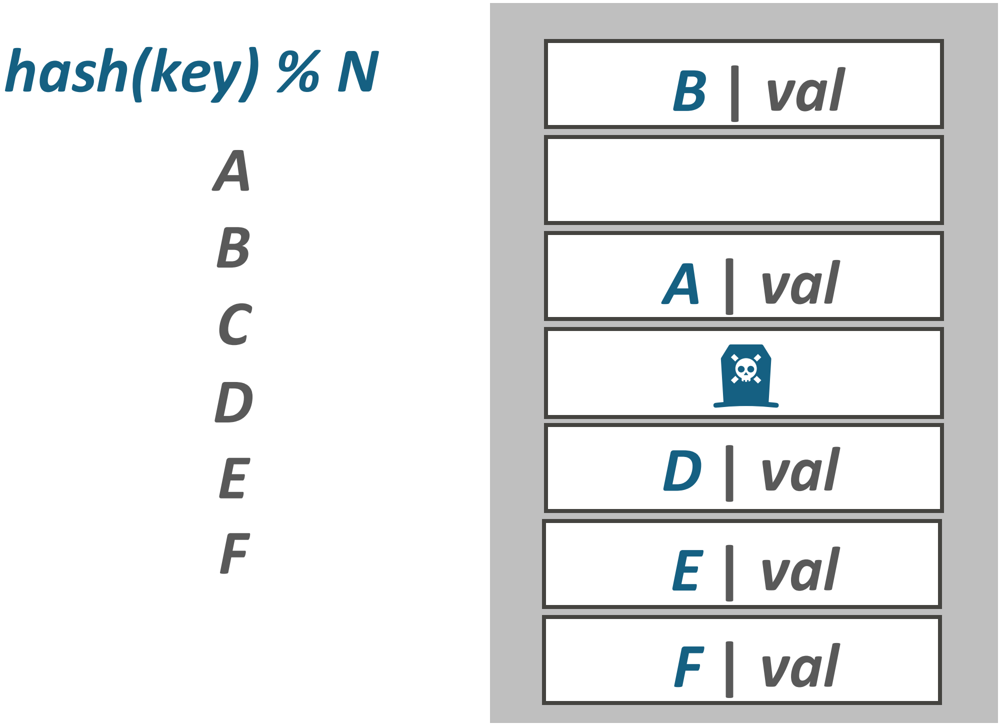
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



hash(key) % N

A

B

C

D

E

F

B | val

A | val

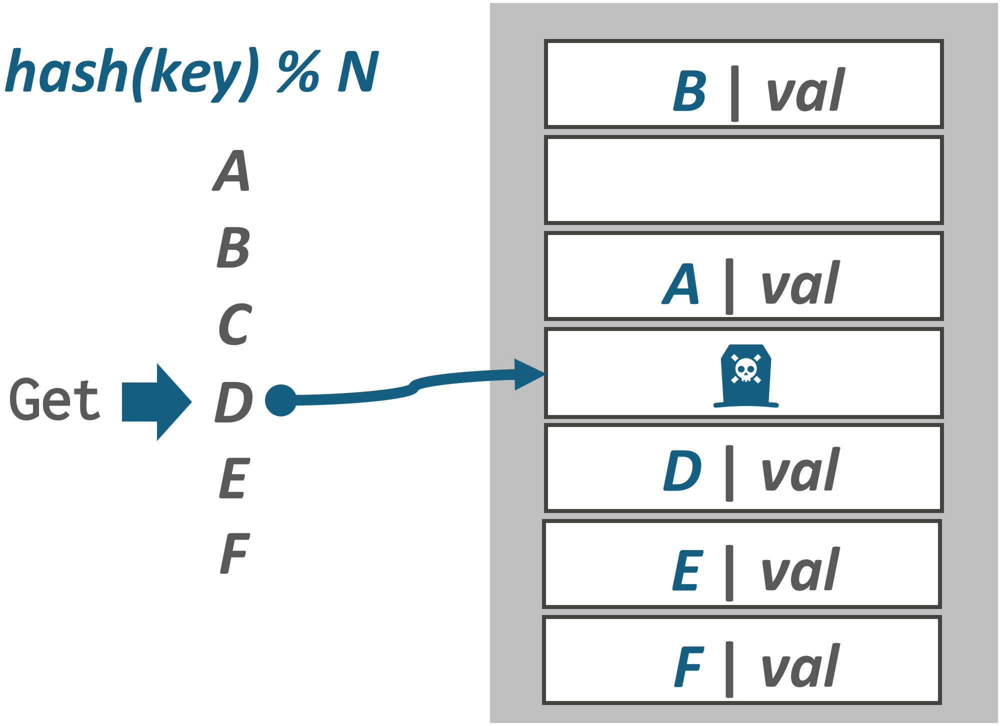
D | val

E | val

F | val

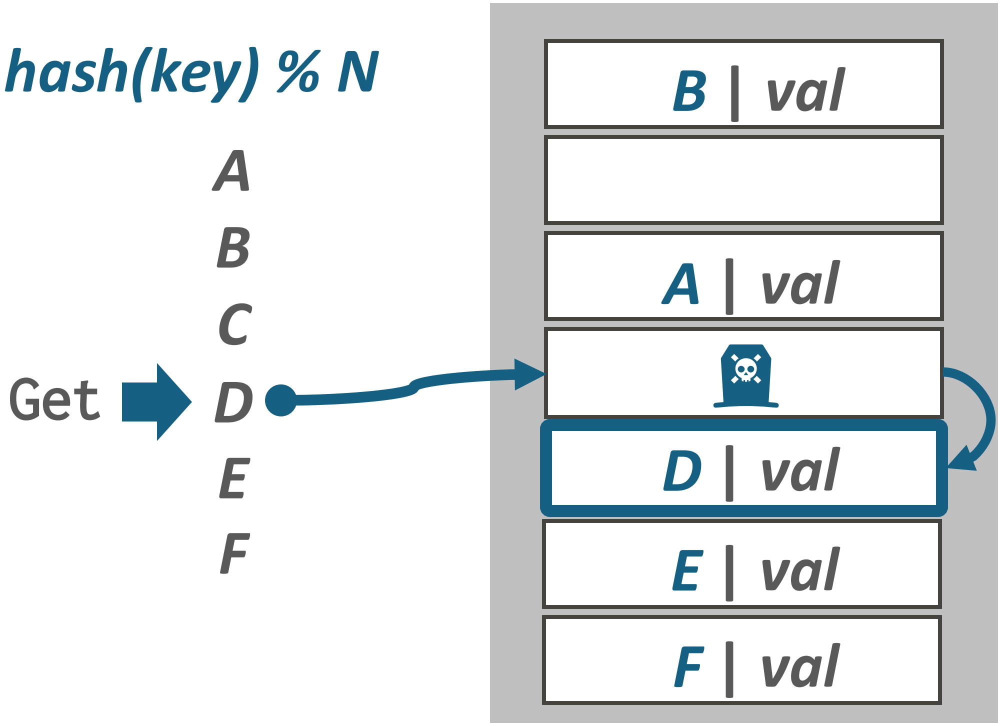
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



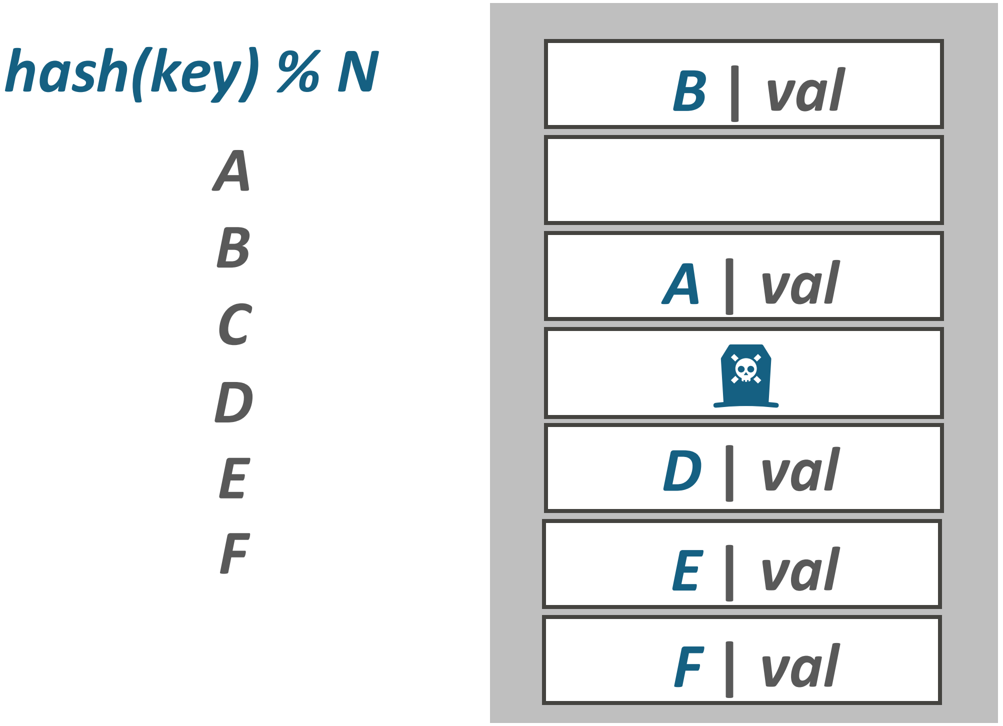
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



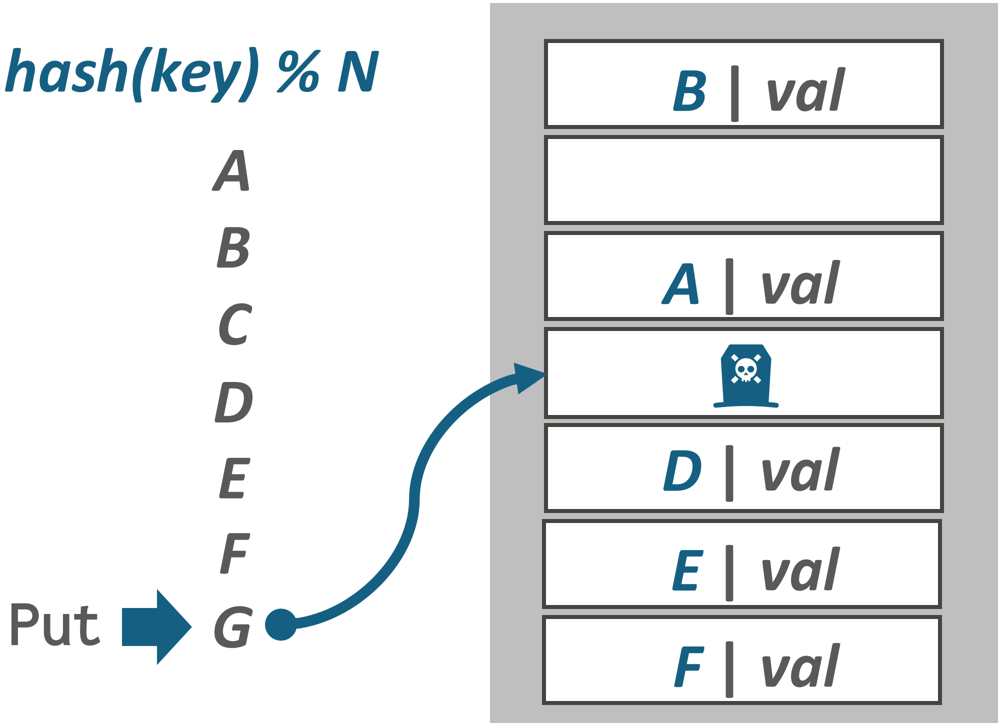
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



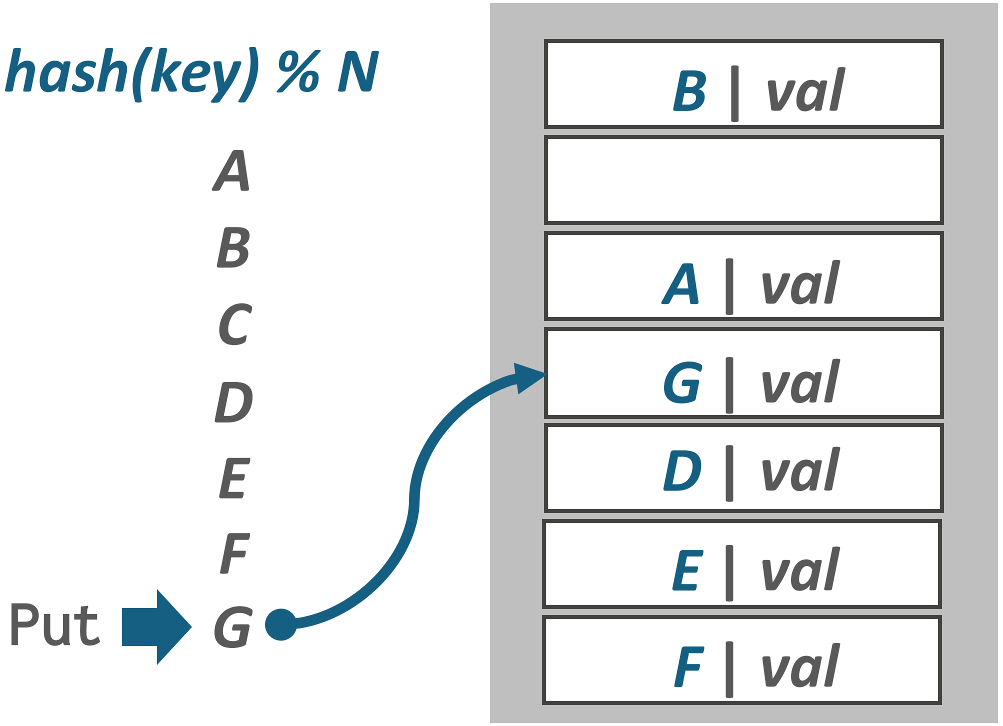
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Linear Probe Hashing - Deletes



- **Approach #2: Tombstone**
    - Set a marker to indicate that the entry in the slot is logically deleted.
    - Reuse the slot for new keys.
    - May need periodic garbage collection.

Linear Probe Hashing - Deletes



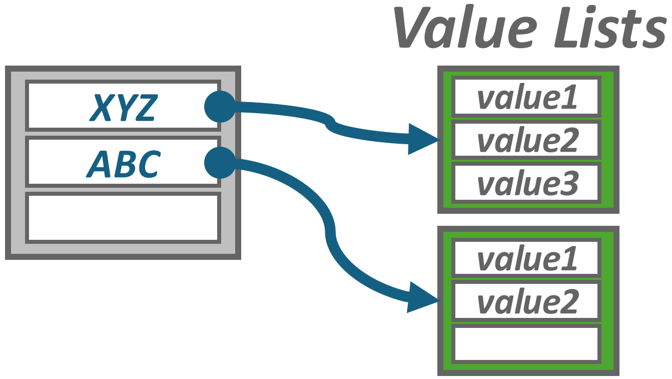
- **Approach #2: Tombstone**
  - Set a marker to indicate that the entry in the slot is logically deleted.
  - Reuse the slot for new keys.
  - May need periodic garbage collection.

Non-Unique Keys

- **Choice #1: Separate Linked List**
  - Store values in separate storage area for each key.
  - Value lists can overflow to multiple pages if the number of duplicates is large.

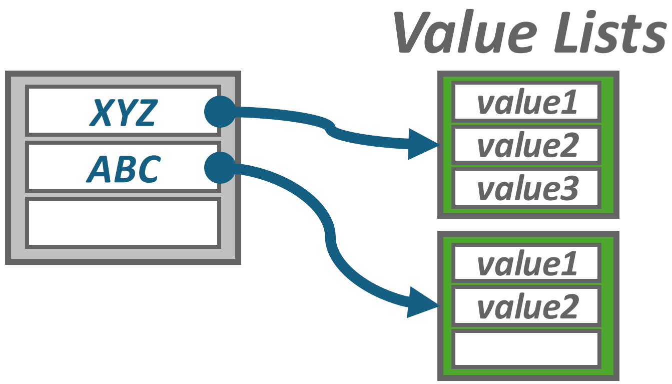
Non-Unique Keys

- **Choice #1: Separate Linked List**
  - Store values in separate storage area for each key.
  - Value lists can overflow to multiple pages if the number of duplicates is large.



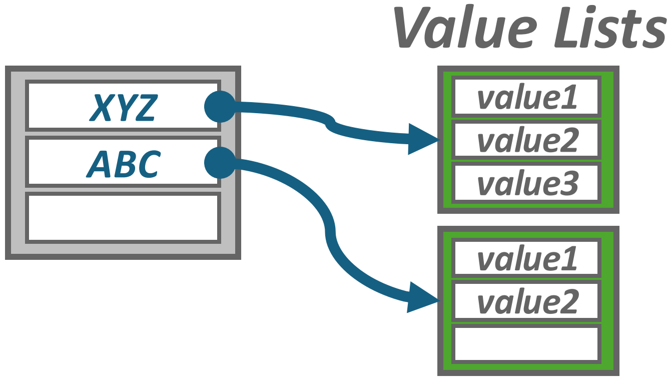
Non-Unique Keys

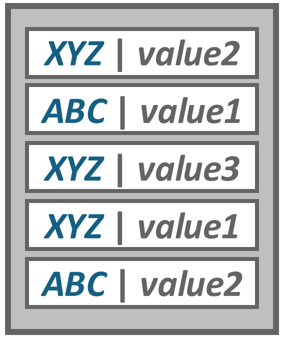
- **Choice #1: Separate Linked List**
  - Store values in separate storage area for each key.
  - Value lists can overflow to multiple pages if the number of duplicates is large.
- **Choice #2: Redundant Keys**
  - Store duplicate keys entries together in the hash table.
  - This is what most systems do.



Non-Unique Keys

- **Choice #1: Separate Linked List**
  - Store values in separate storage area for each key.
  - Value lists can overflow to multiple pages if the number of duplicates is large.
- **Choice #2: Redundant Keys**
  - Store duplicate keys entries together in the hash table.
  - This is what most systems do.





Optimizations

- Specialized hash table implementations based on key type(s) and sizes.
  - Example: Maintain multiple hash tables for different string sizes for a set of keys.
- Store metadata separate in a separate array.
  - Packed bitmap tracks whether a slot is empty/tombstone.
- Use table + slot versioning metadata to quickly invalidate all entries in the hash table.
  - Example: If table version does not match slot version, then treat the slot as empty.

Source: Maksim Kita

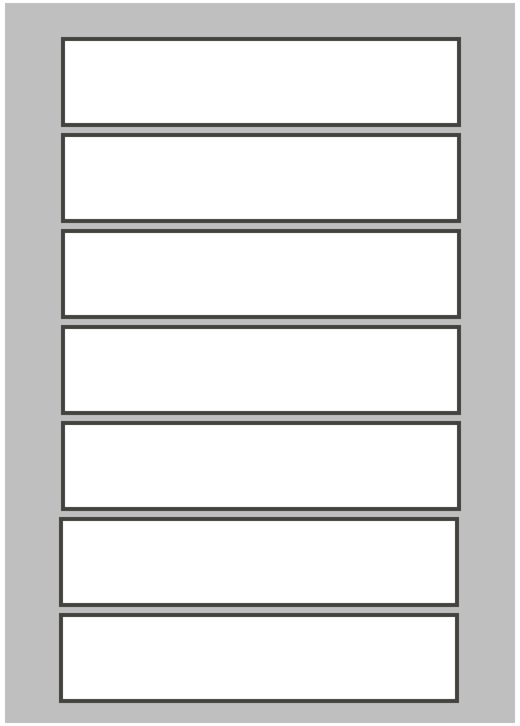
Cuckoo Hashing

Static Hashing Schemes

Cuckoo Hashing

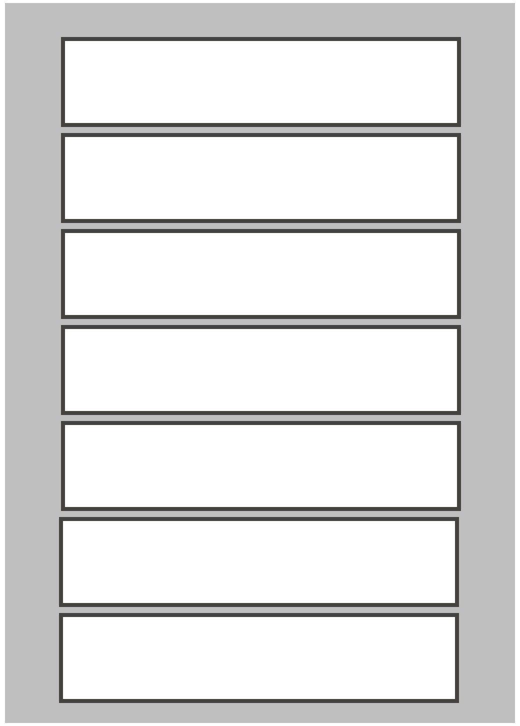
- Use multiple hash functions to find multiple locations in the hash table to insert records.
  - On insert, check multiple locations and pick the one that is empty.
  - If no location is available, evict the element from one of them and then re-hash it find a new location.
- Look-ups and deletions are always $\boldsymbol{O(1)}$ because only one location per hash table is checked.
- Best open-source implementation is from CMU.

Cuckoo Hashing



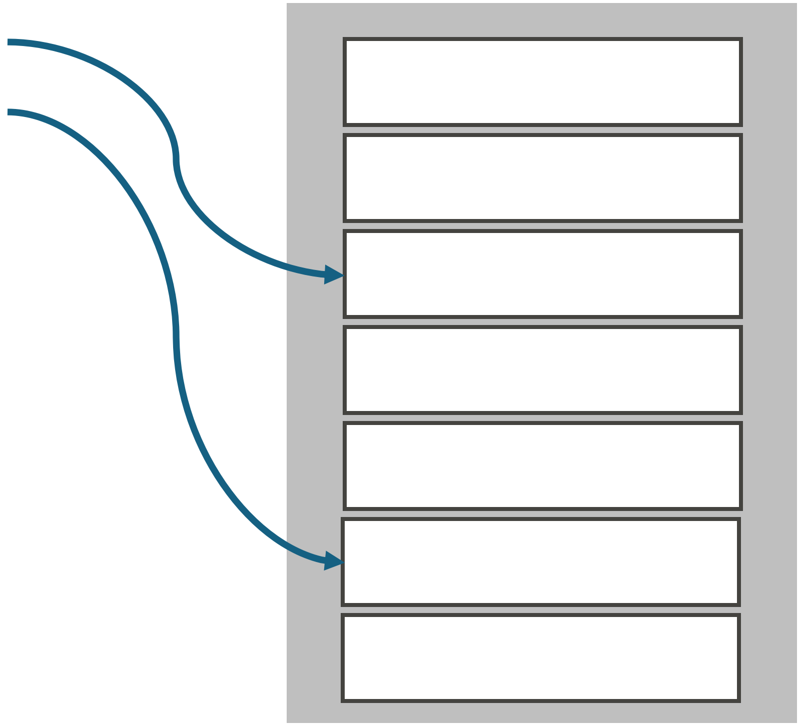
Cuckoo Hashing

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

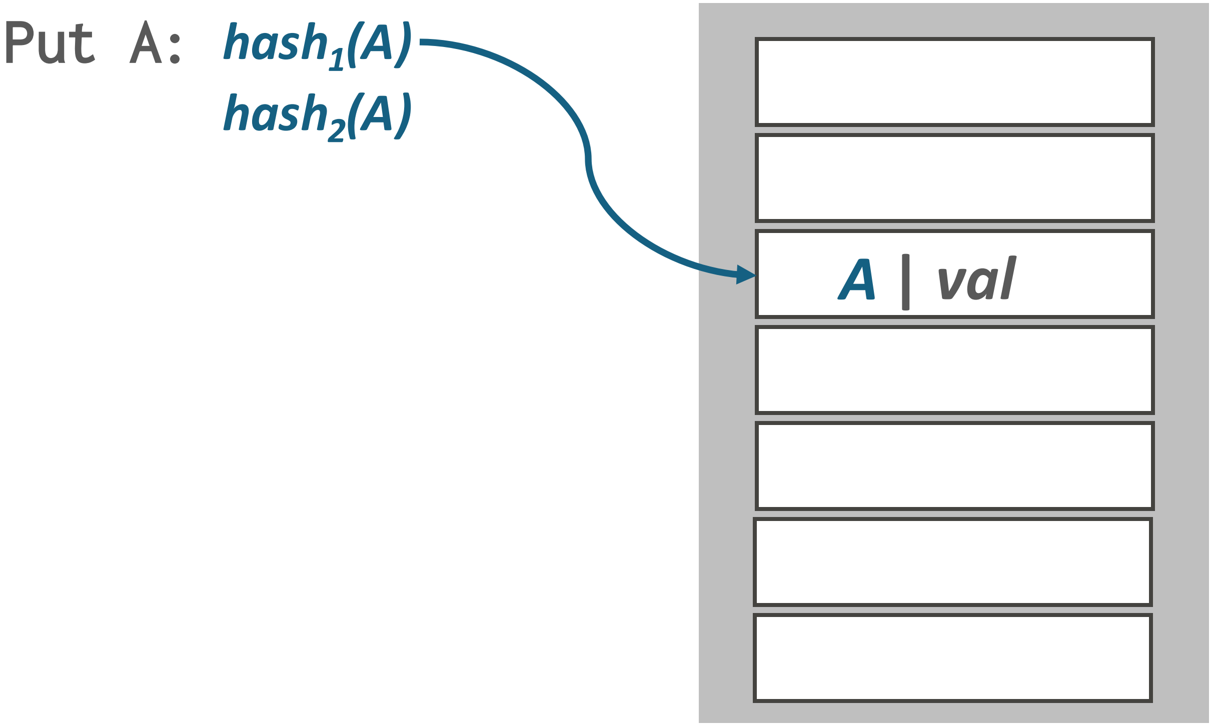


Cuckoo Hashing

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



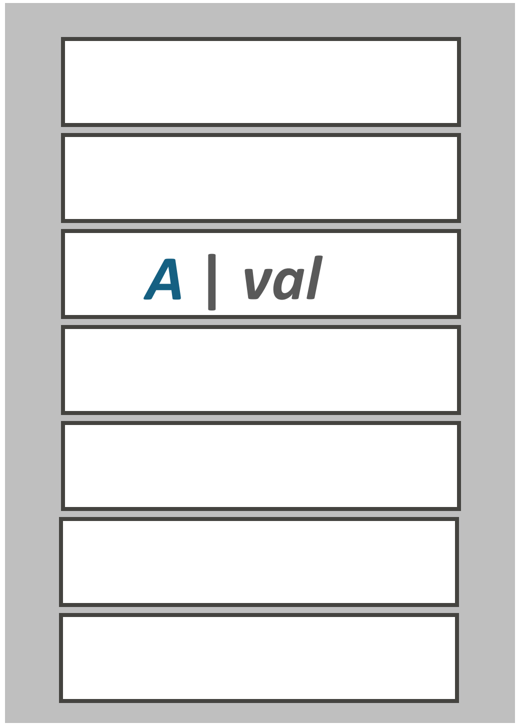
Cuckoo Hashing



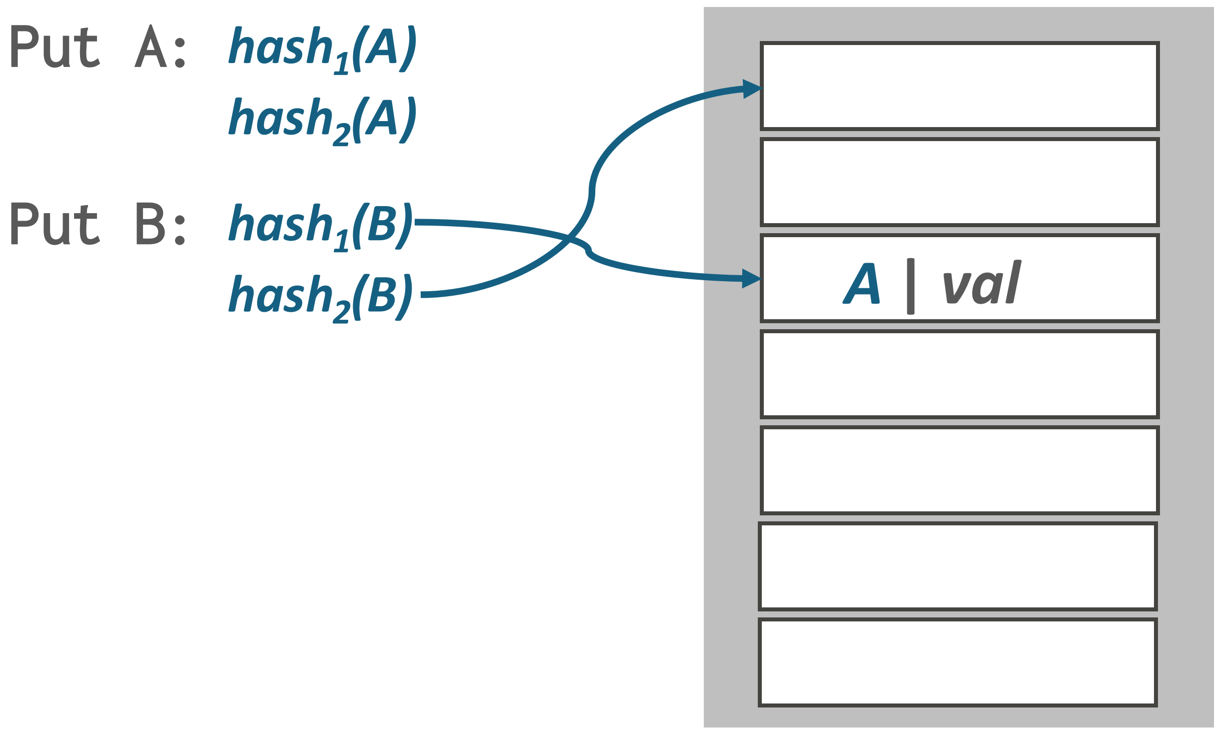
Cuckoo Hashing

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

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



Cuckoo Hashing

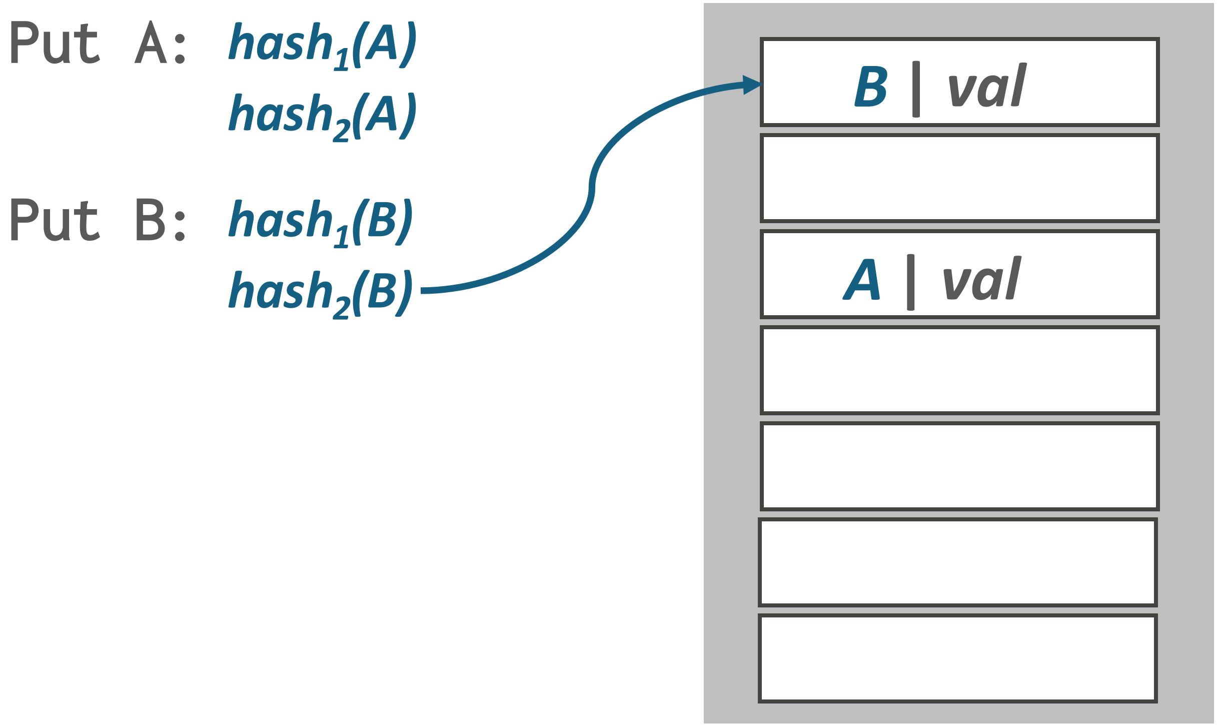


Put A: $hash_1(A)$
$hash_2(A)$

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

A | val

Cuckoo Hashing



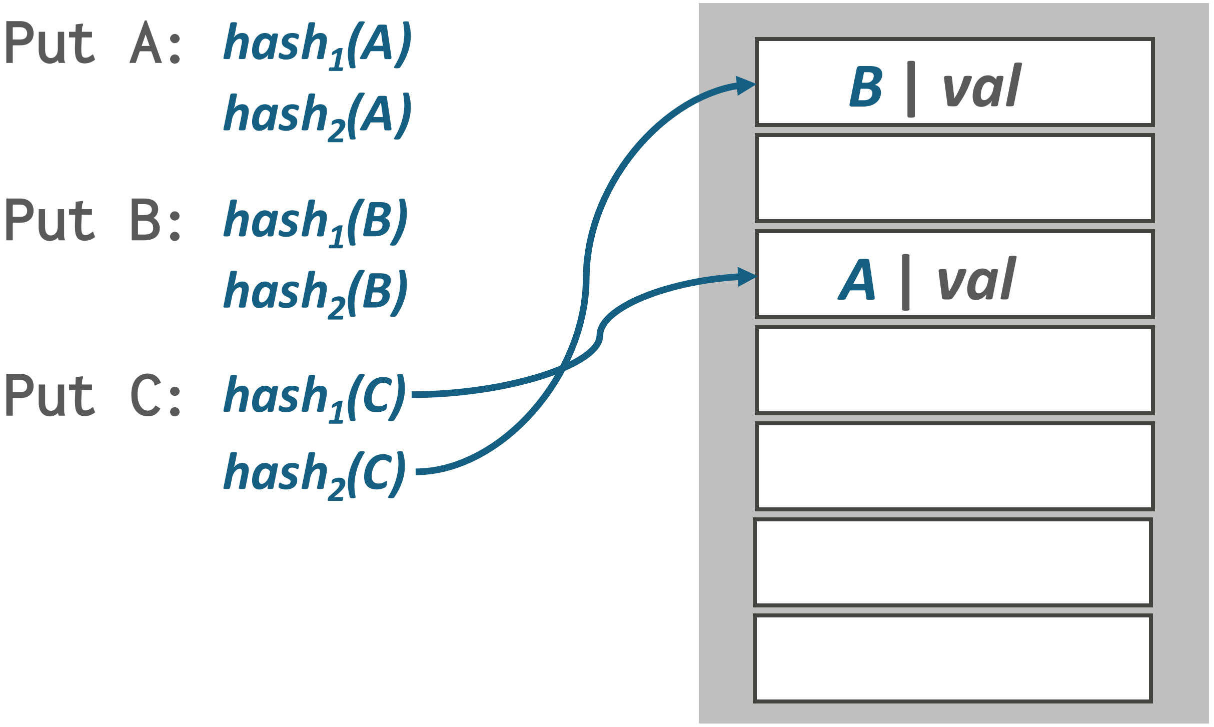
Put A: $hash_1(A)$
$hash_2(A)$

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

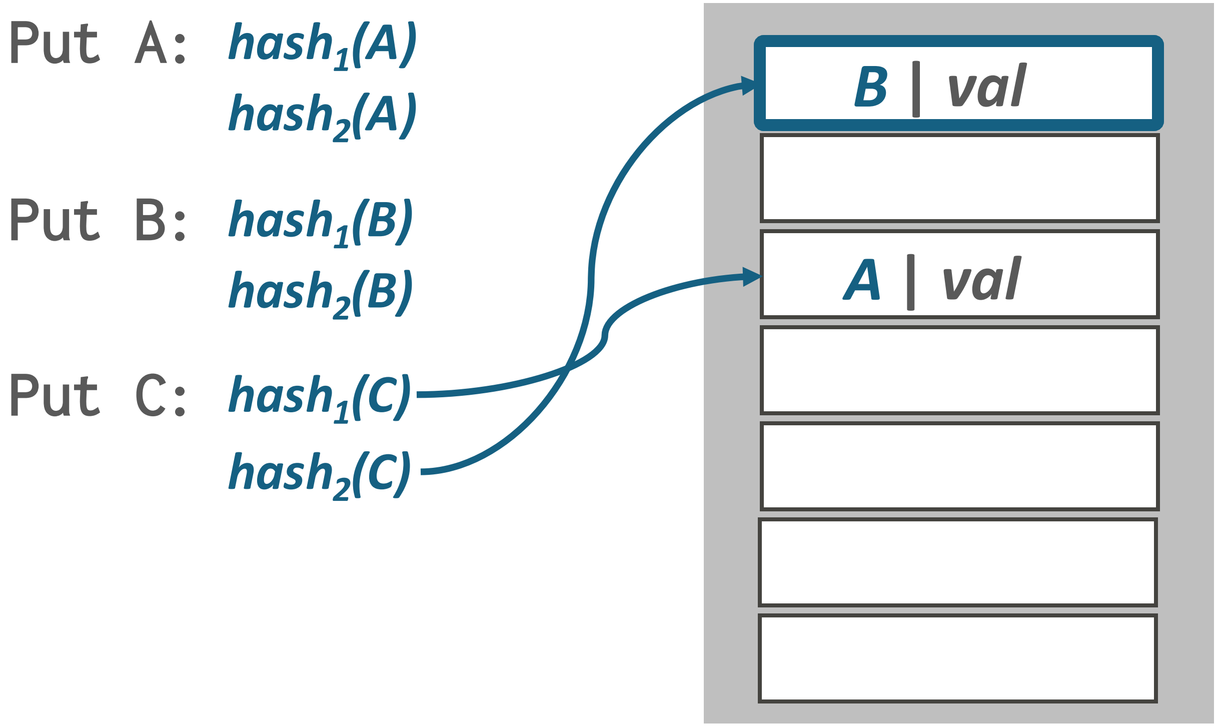
B | val

A | val

Cuckoo Hashing



Cuckoo Hashing



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

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

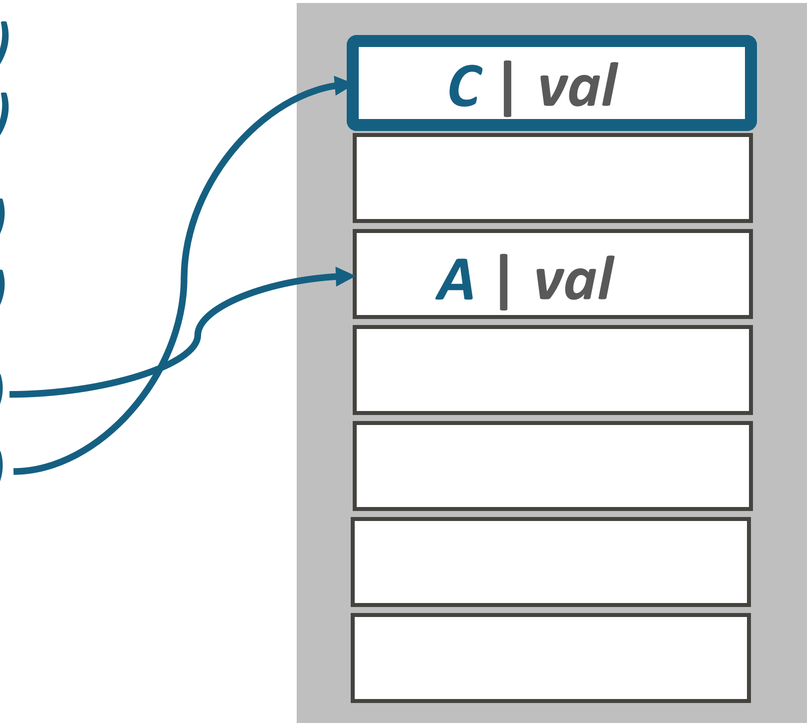
Put C: $hash_1(C)$
$hash_2(C)$

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)

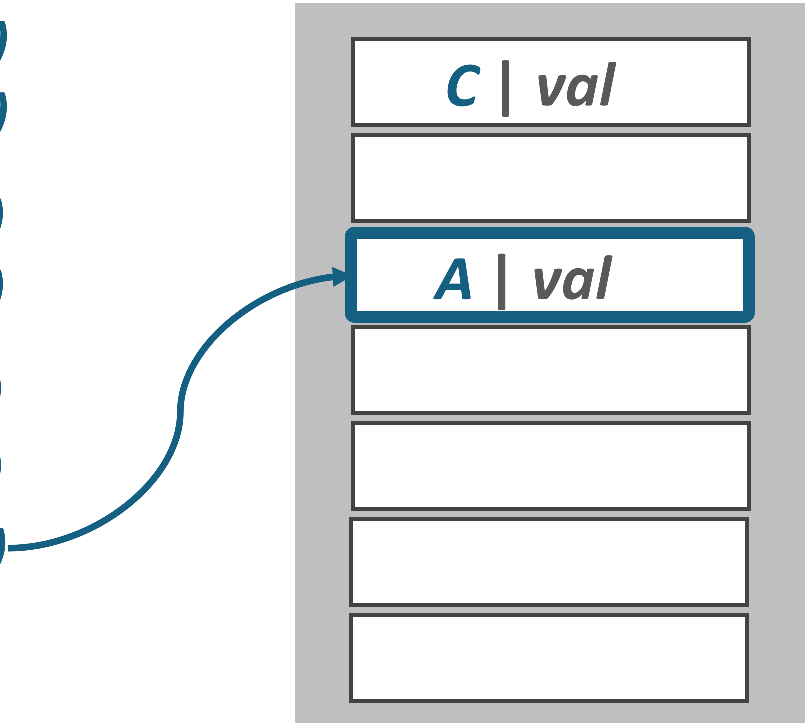


Cuckoo Hashing

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: $hash_1(A)$

$hash_2(A)$

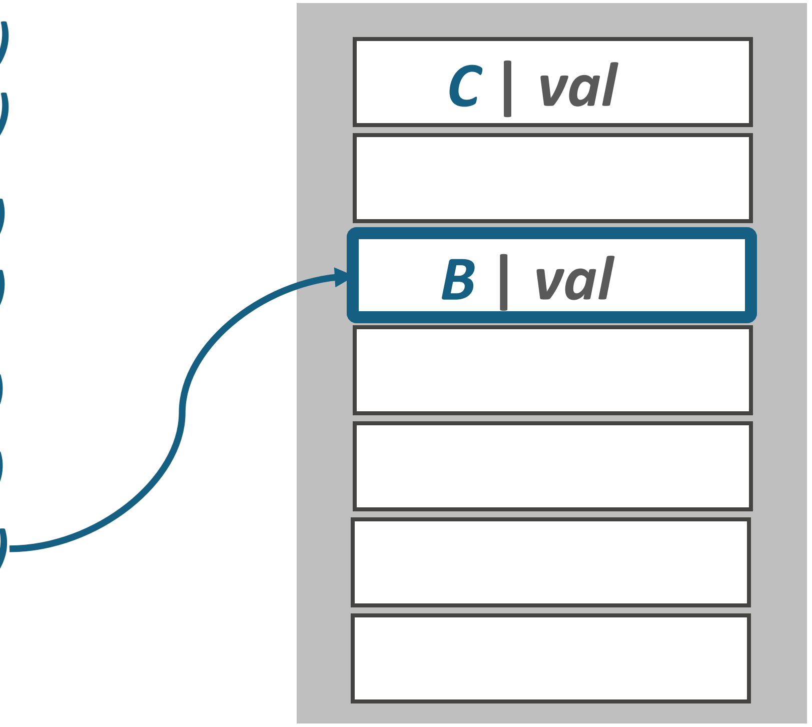
Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$

$hash_1(B)$



Cuckoo Hashing

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

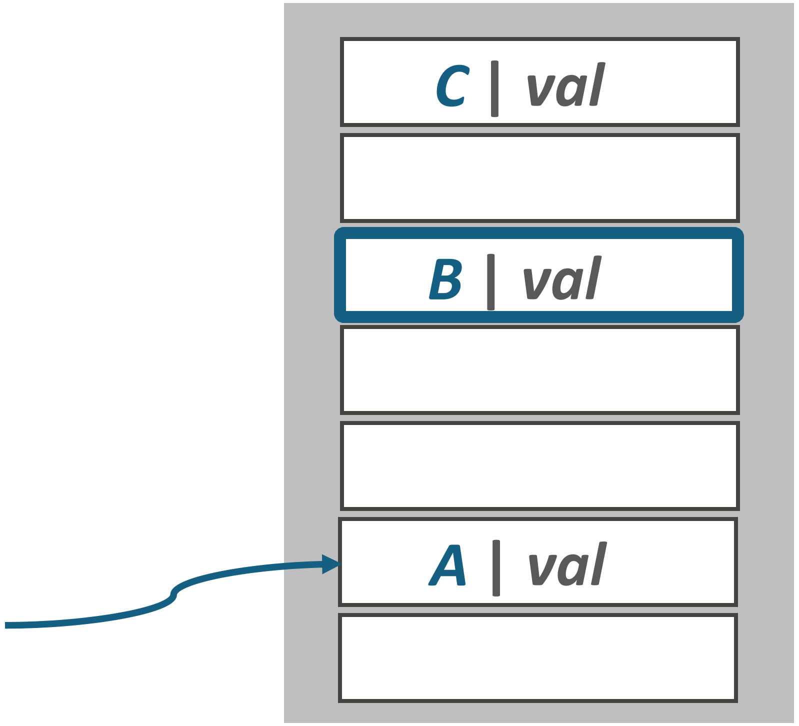
$hash_2(B)$

Put C: $hash_1(C)$

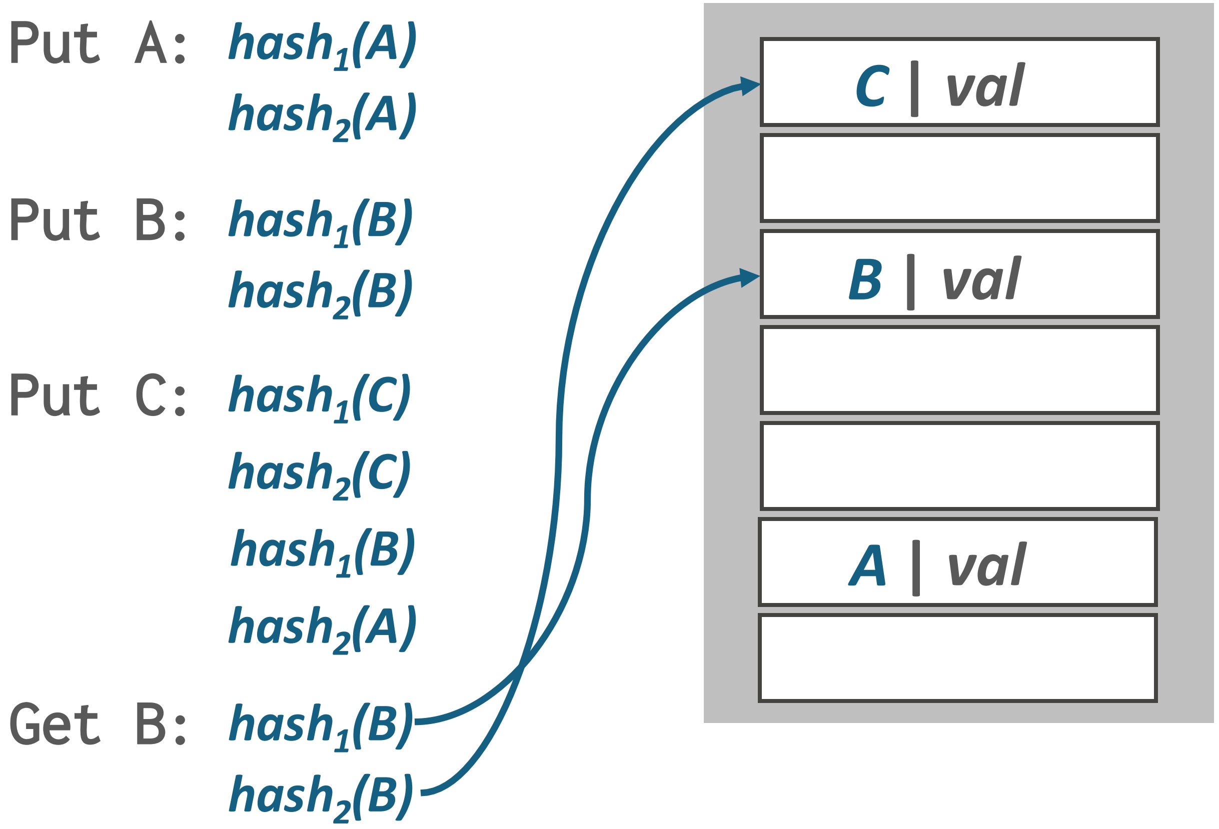
$hash_2(C)$

$hash_1(B)$

$hash_2(A)$



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

Chained Hash Table

Dynamic Hashing Schemes

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

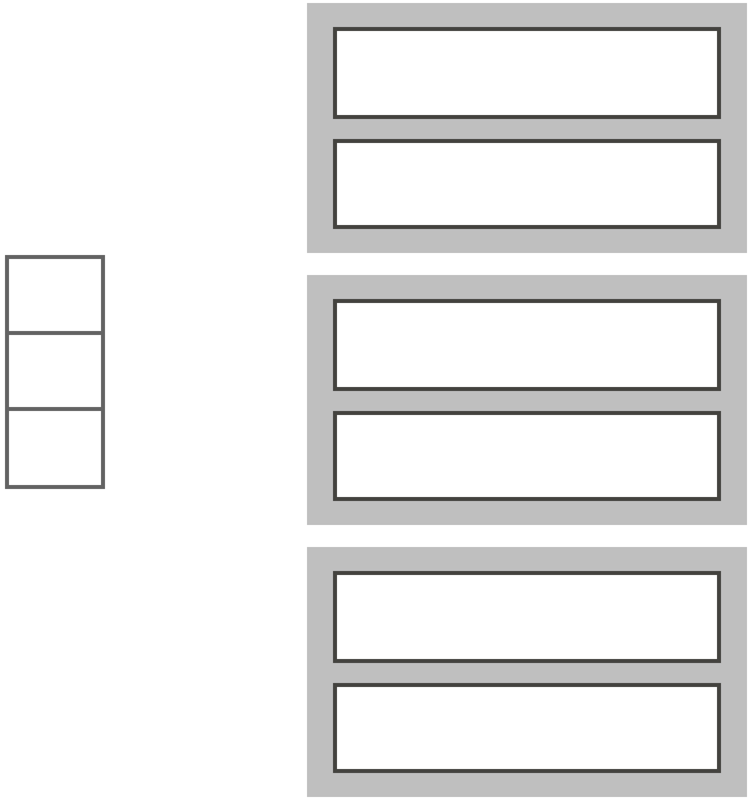
Chained Hashing

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

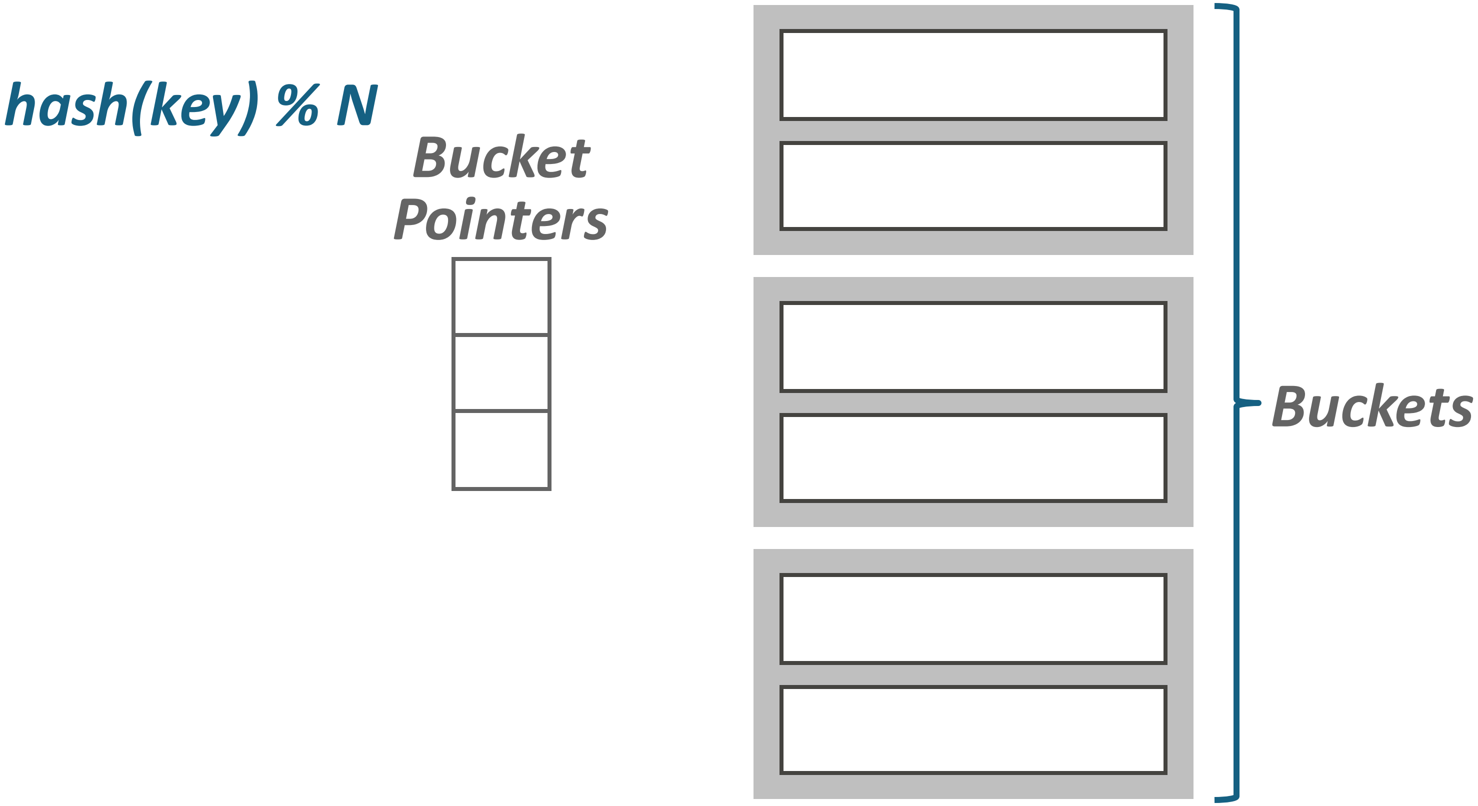
- Resolve collisions by placing all elements with the same hash key into the same bucket.
    - To determine whether an element is present, hash to its bucket and scan for it.
    - Insertions and deletions are generalizations of lookups.

Chained Hashing

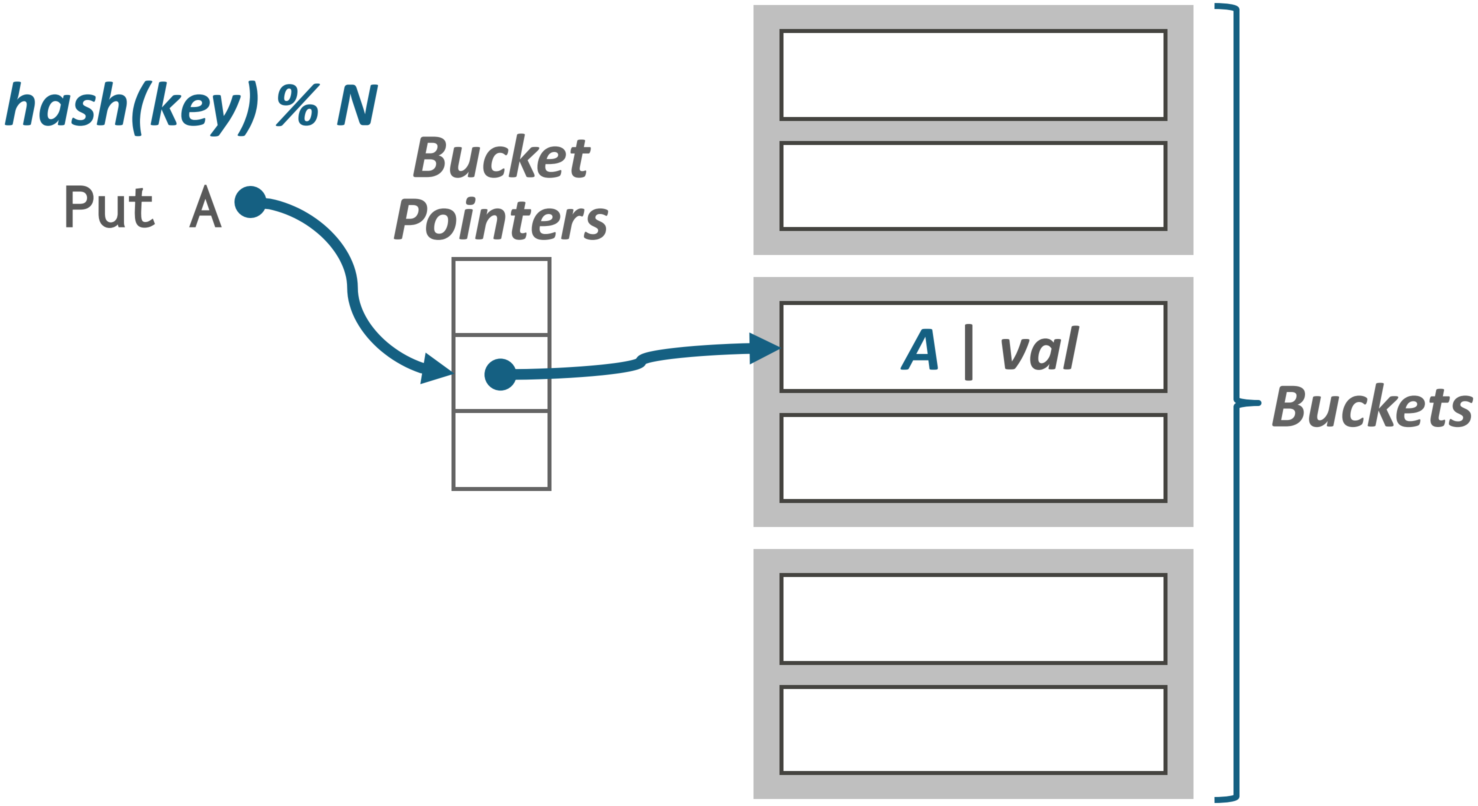
hash(key) % N



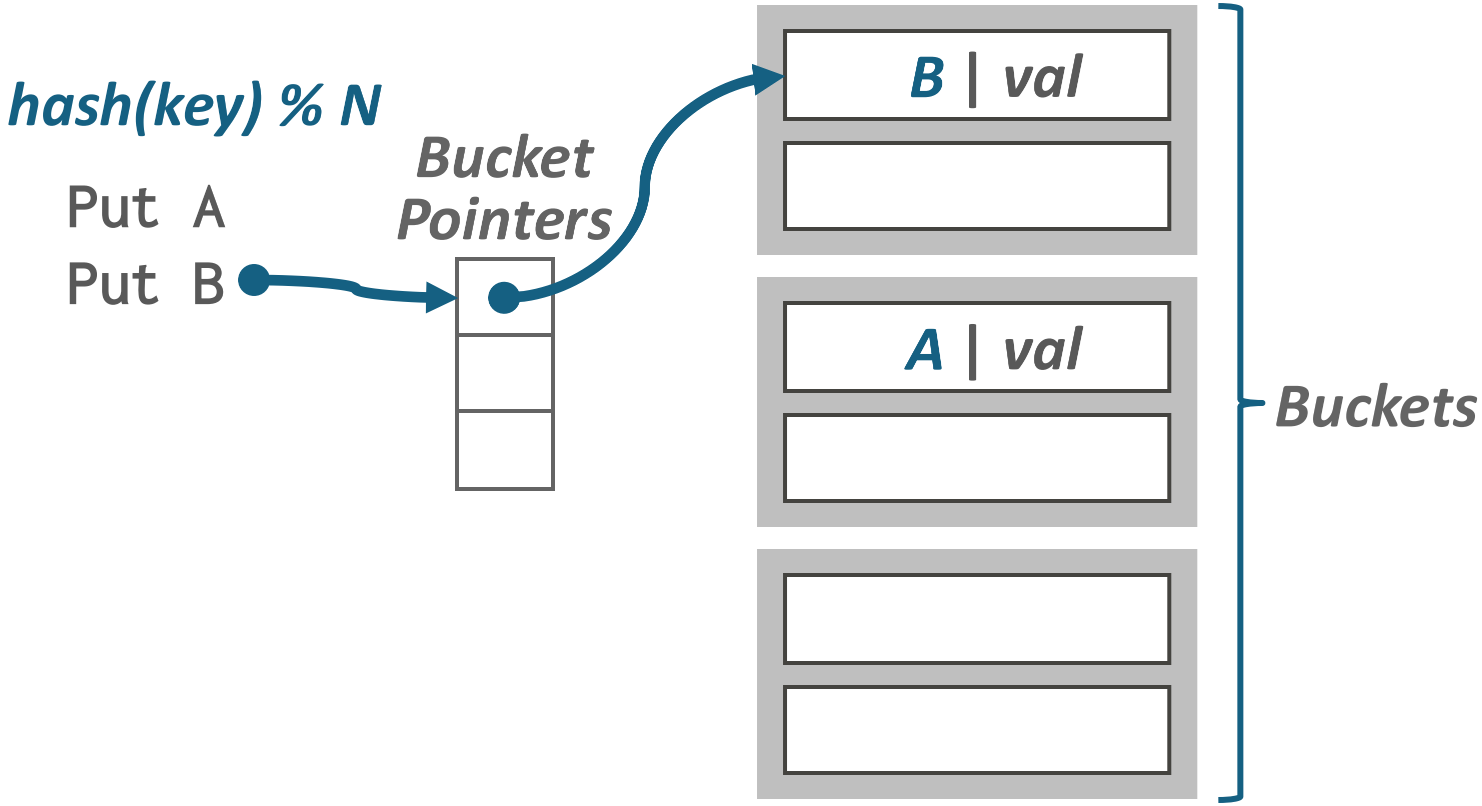
Chained Hashing



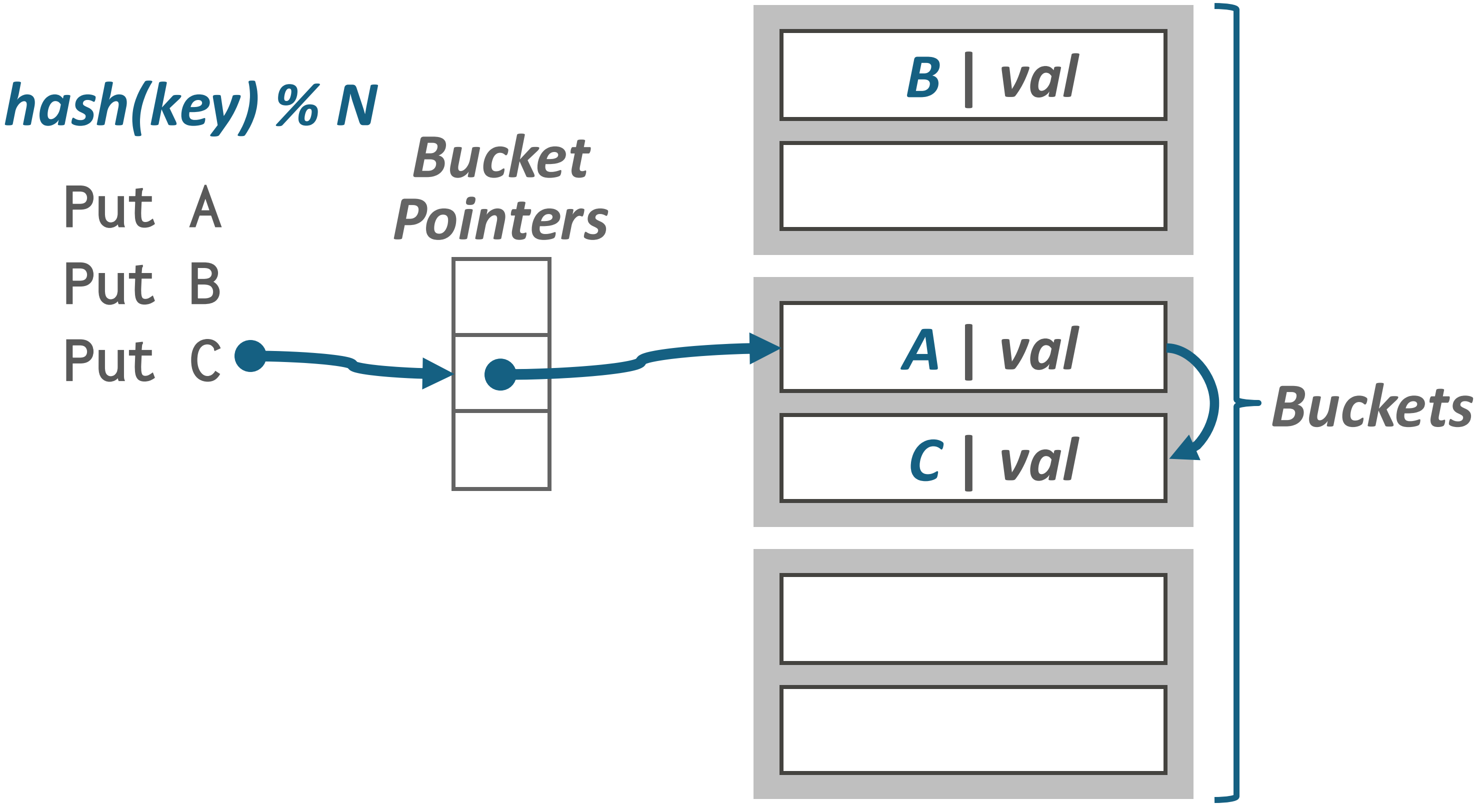
Chained Hashing



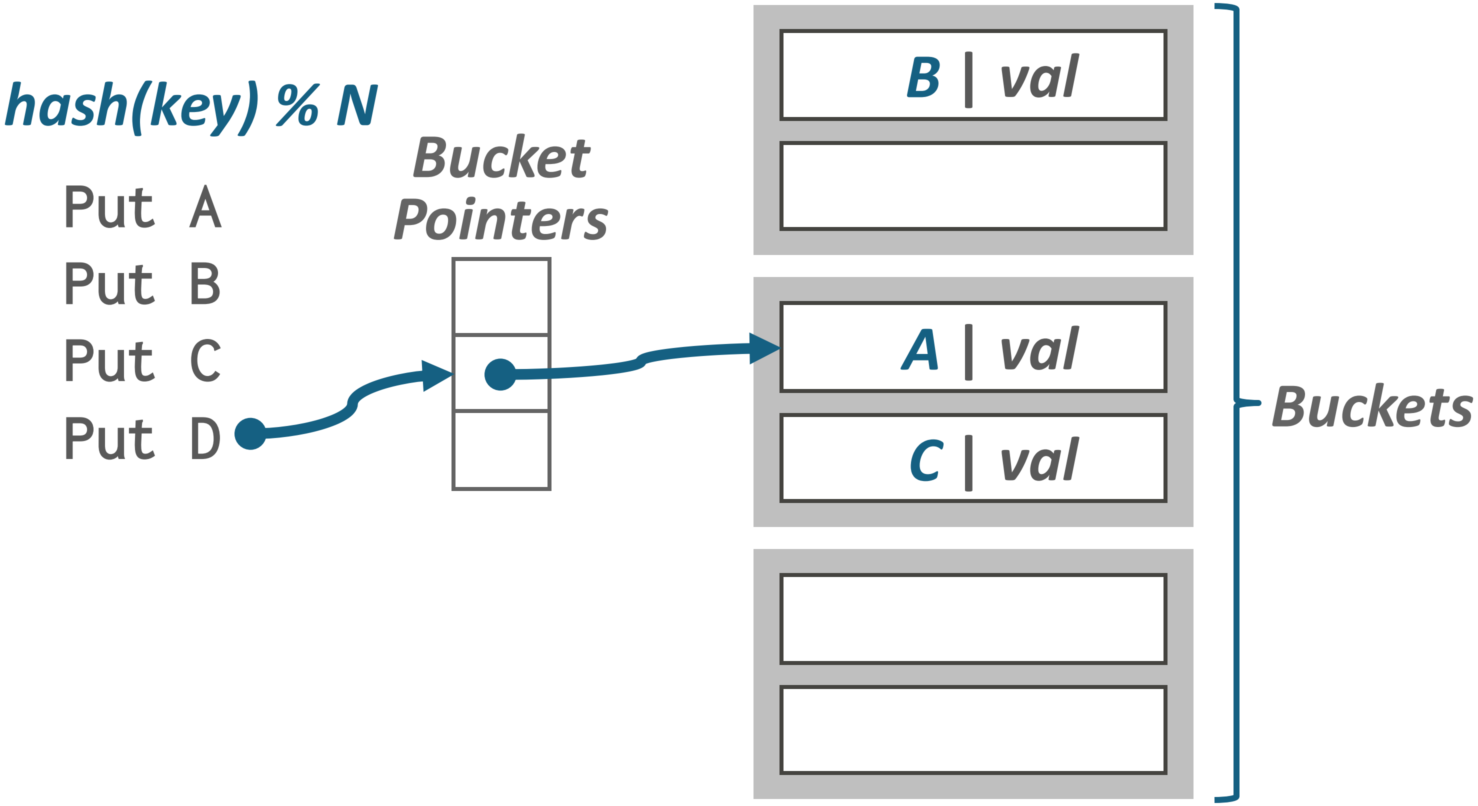
Chained Hashing



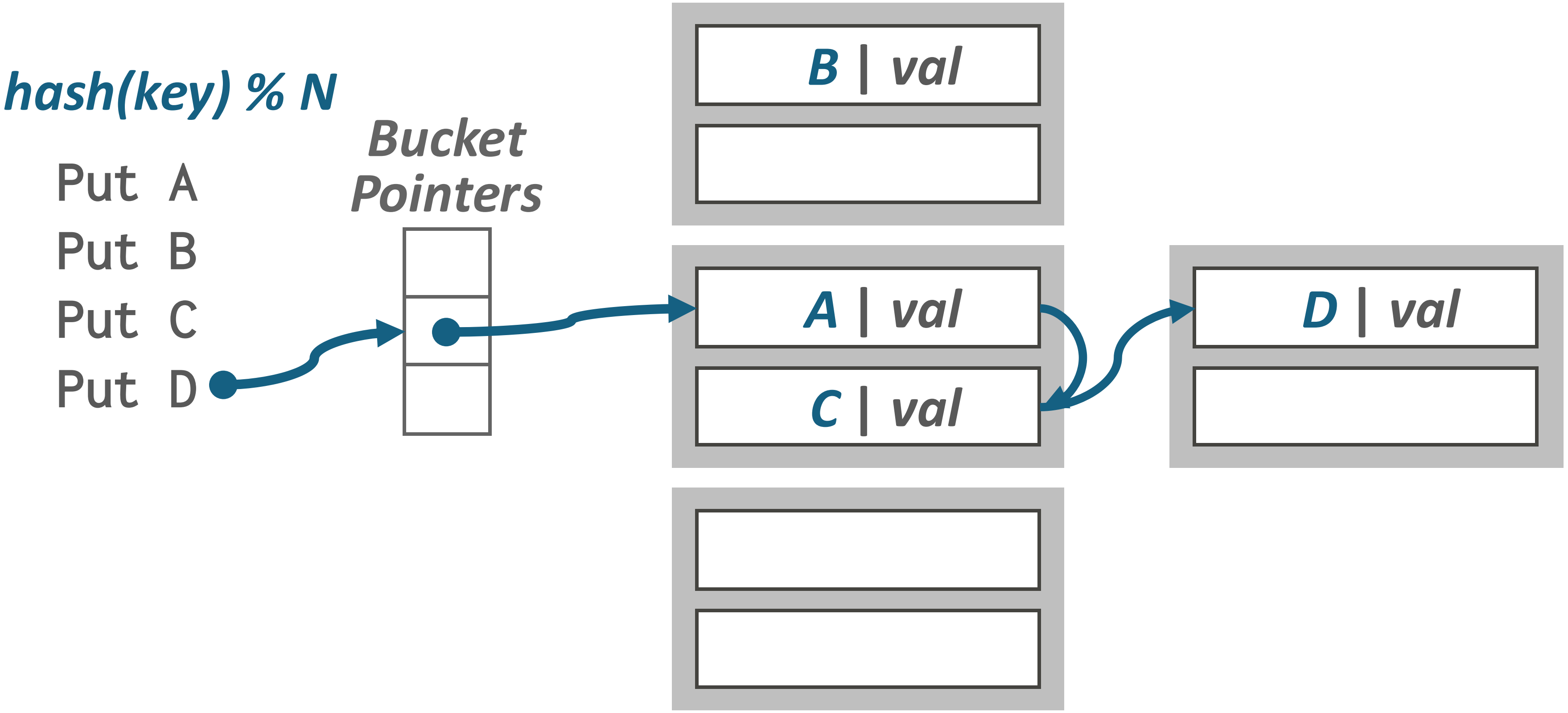
Chained Hashing



Chained Hashing



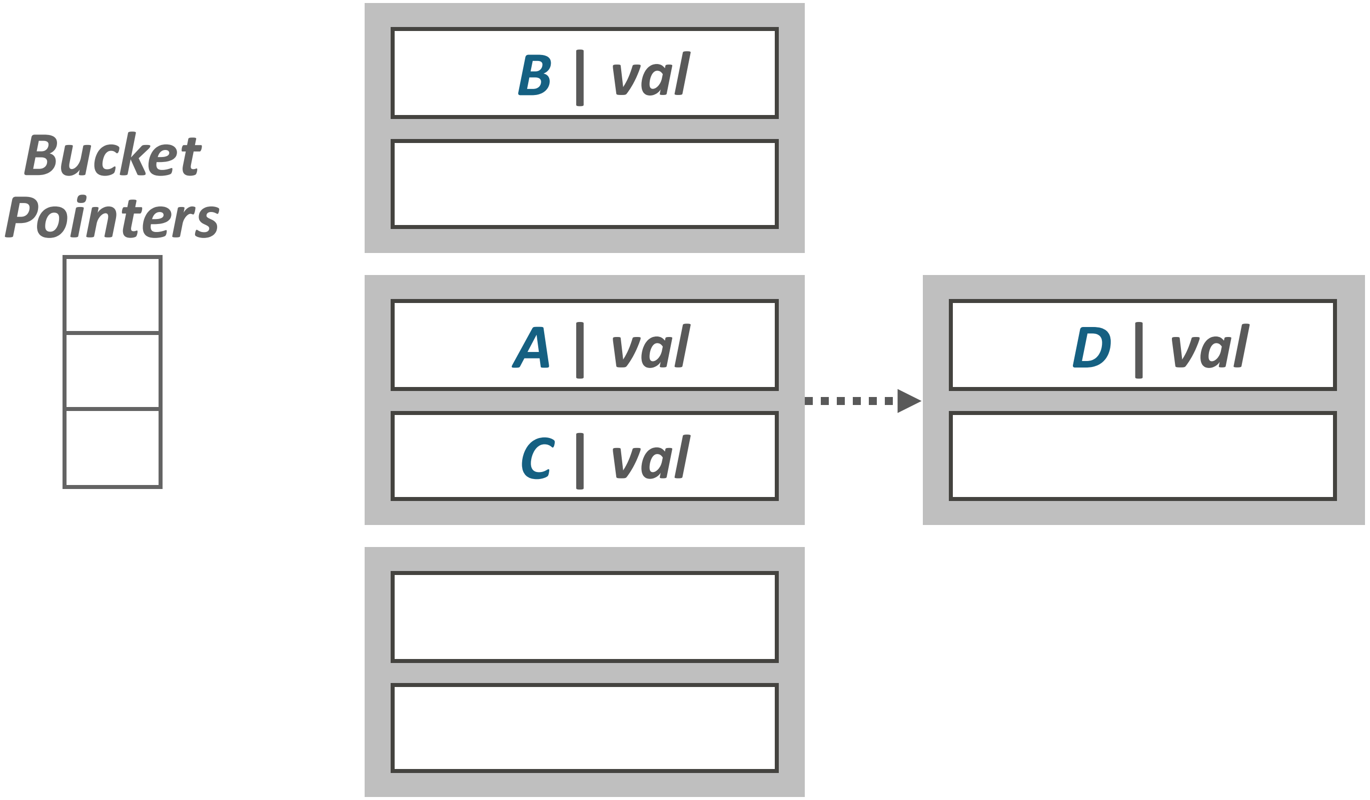
Chained Hashing



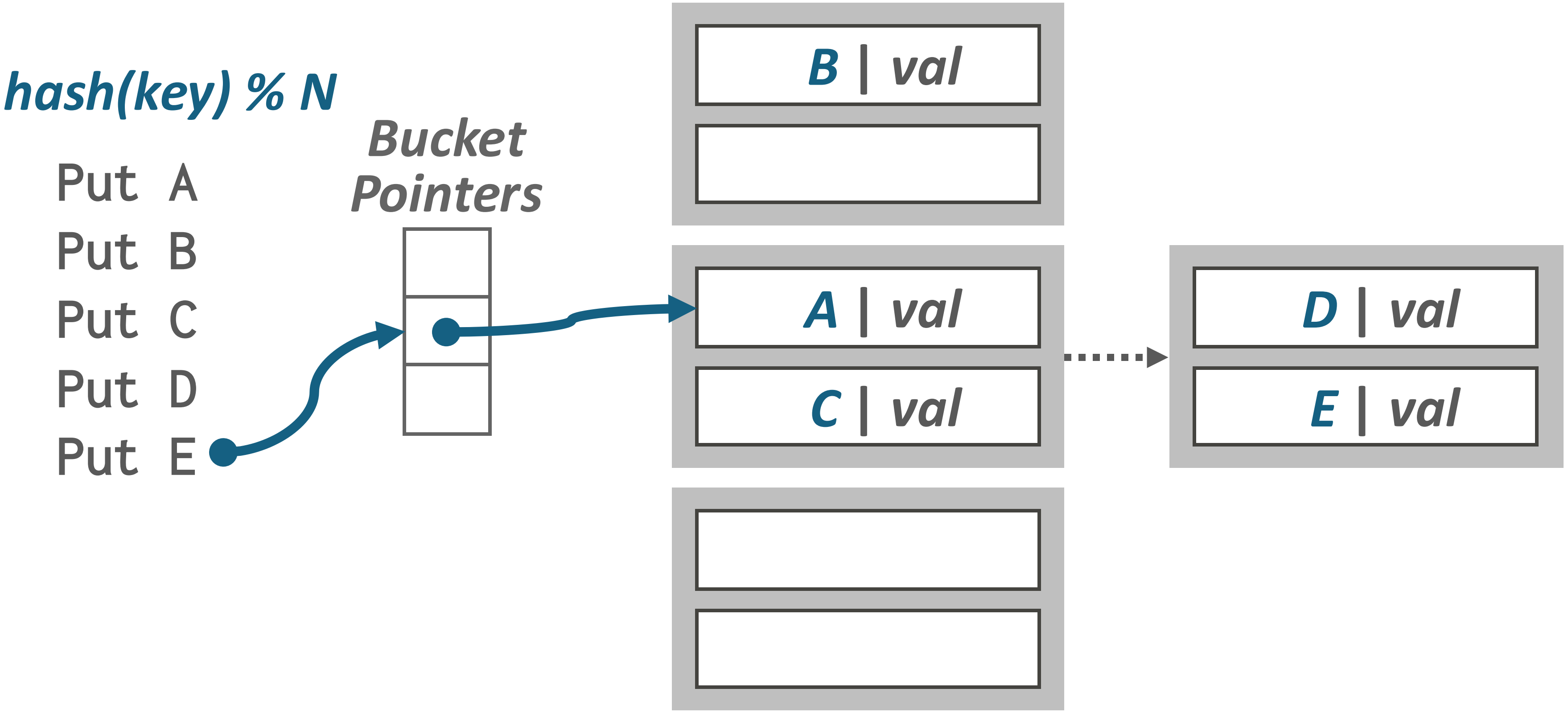
Chained Hashing

hash(key) % N

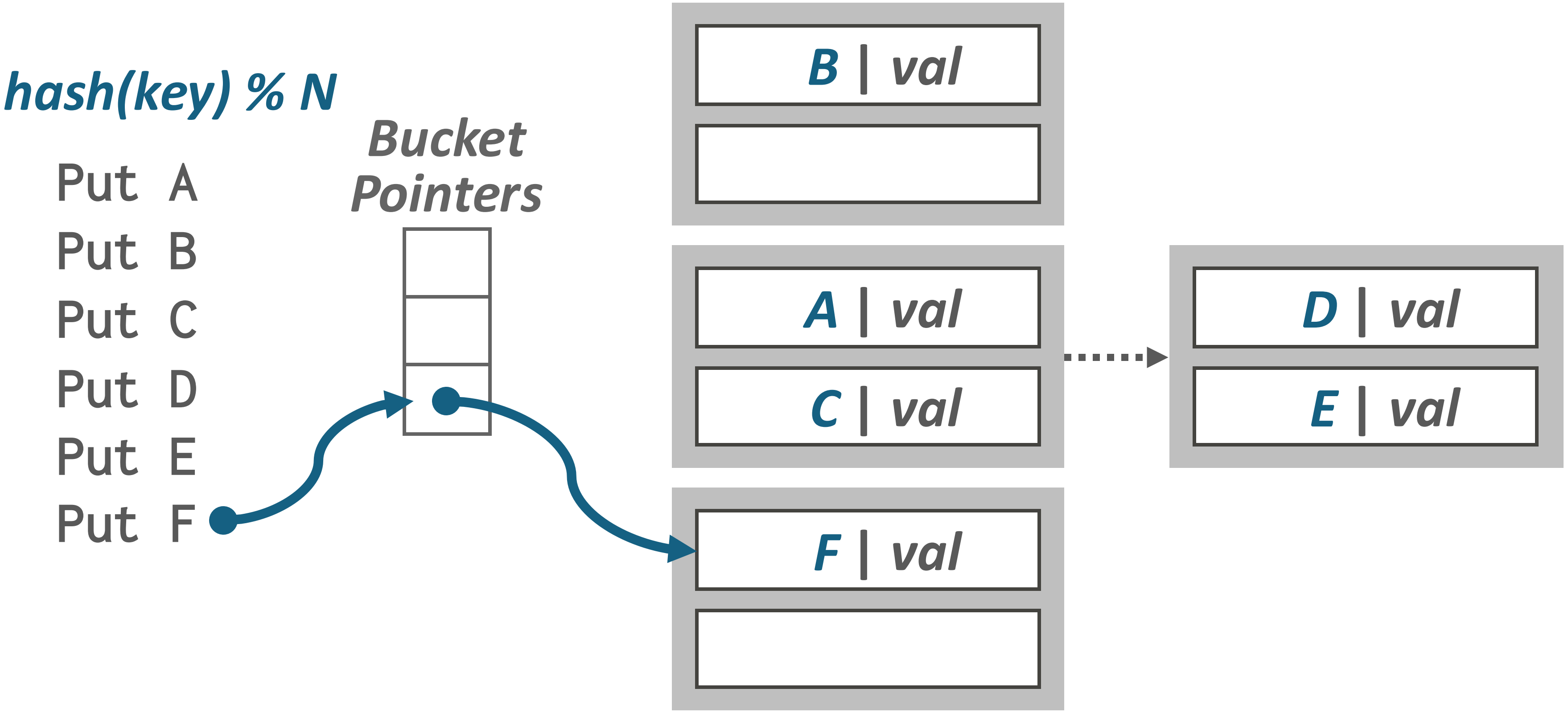
Put A
Put B
Put C
Put D
Put E



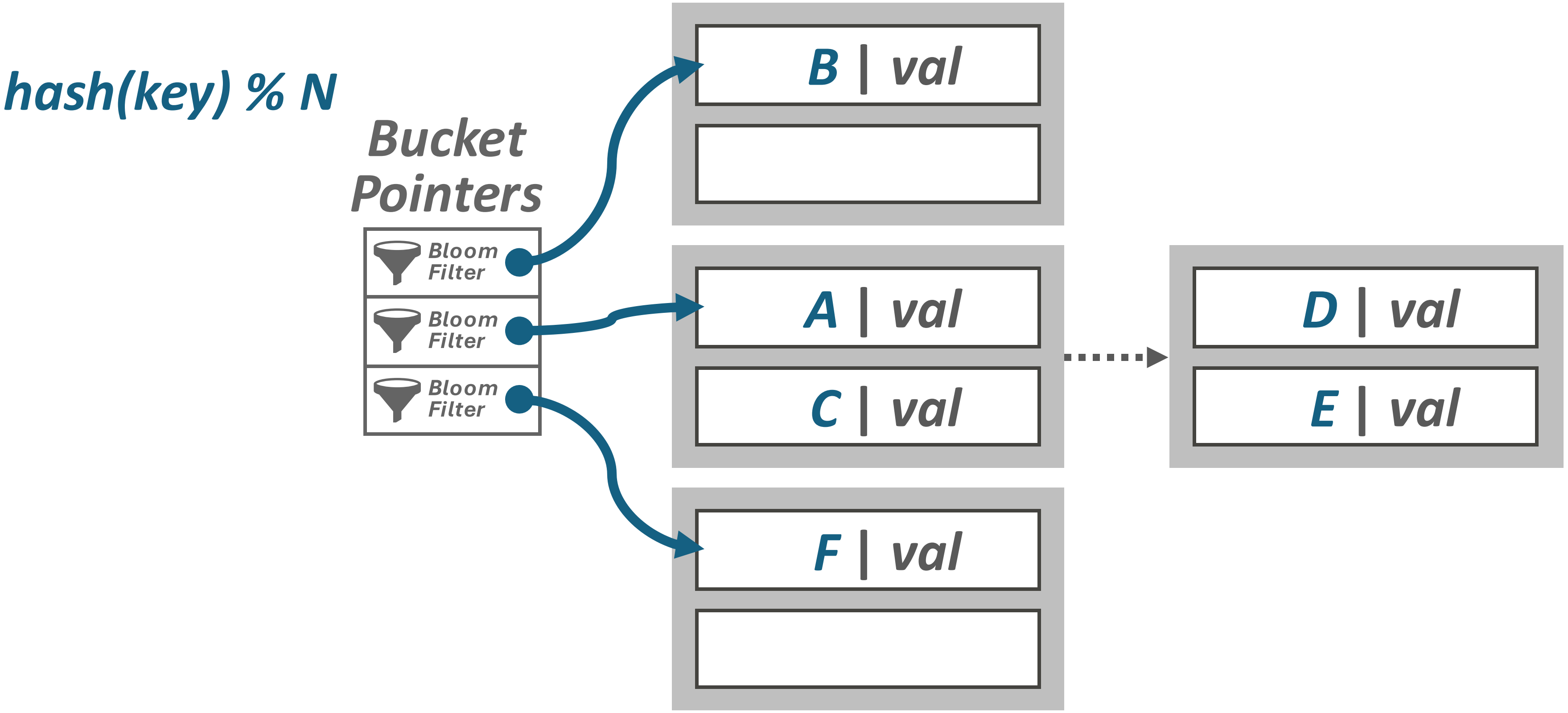
Chained Hashing



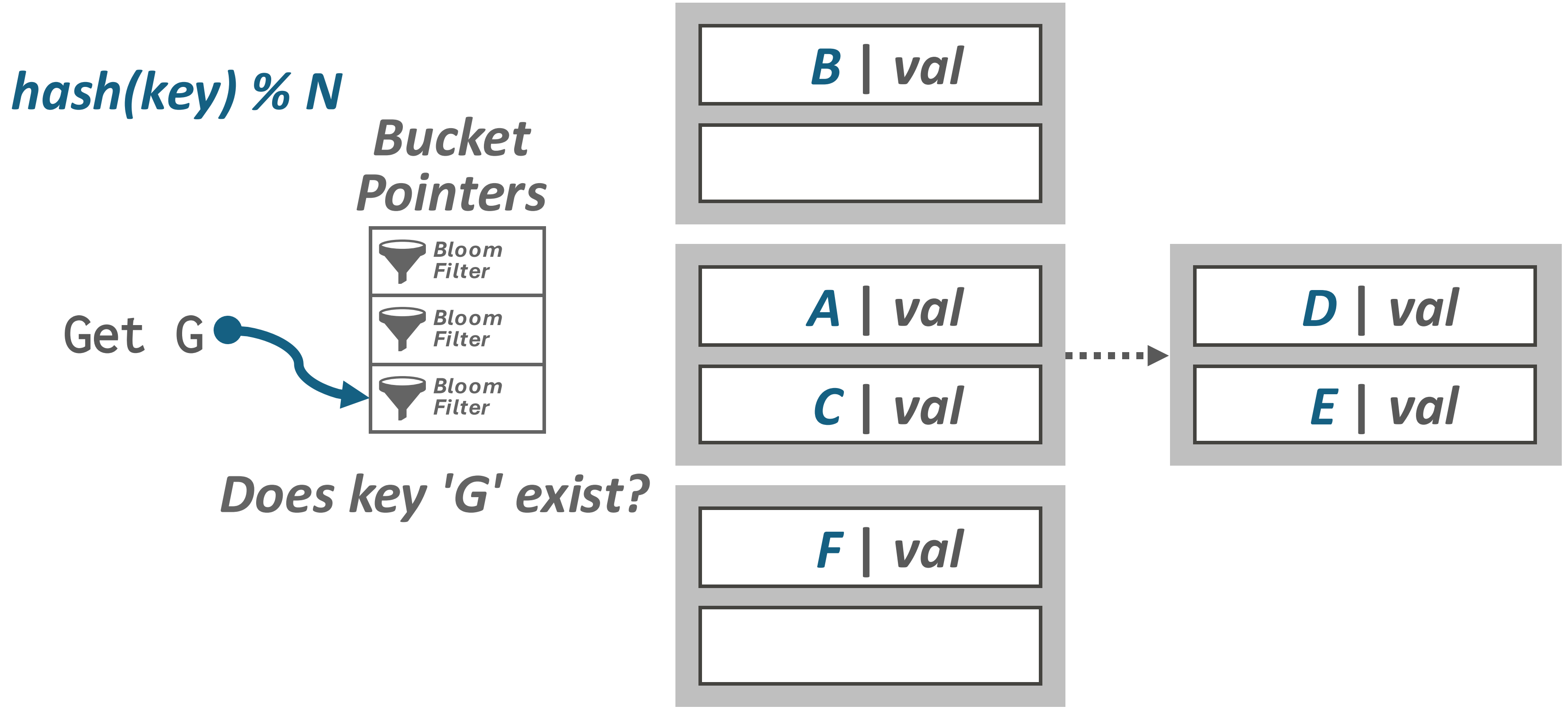
Chained Hashing



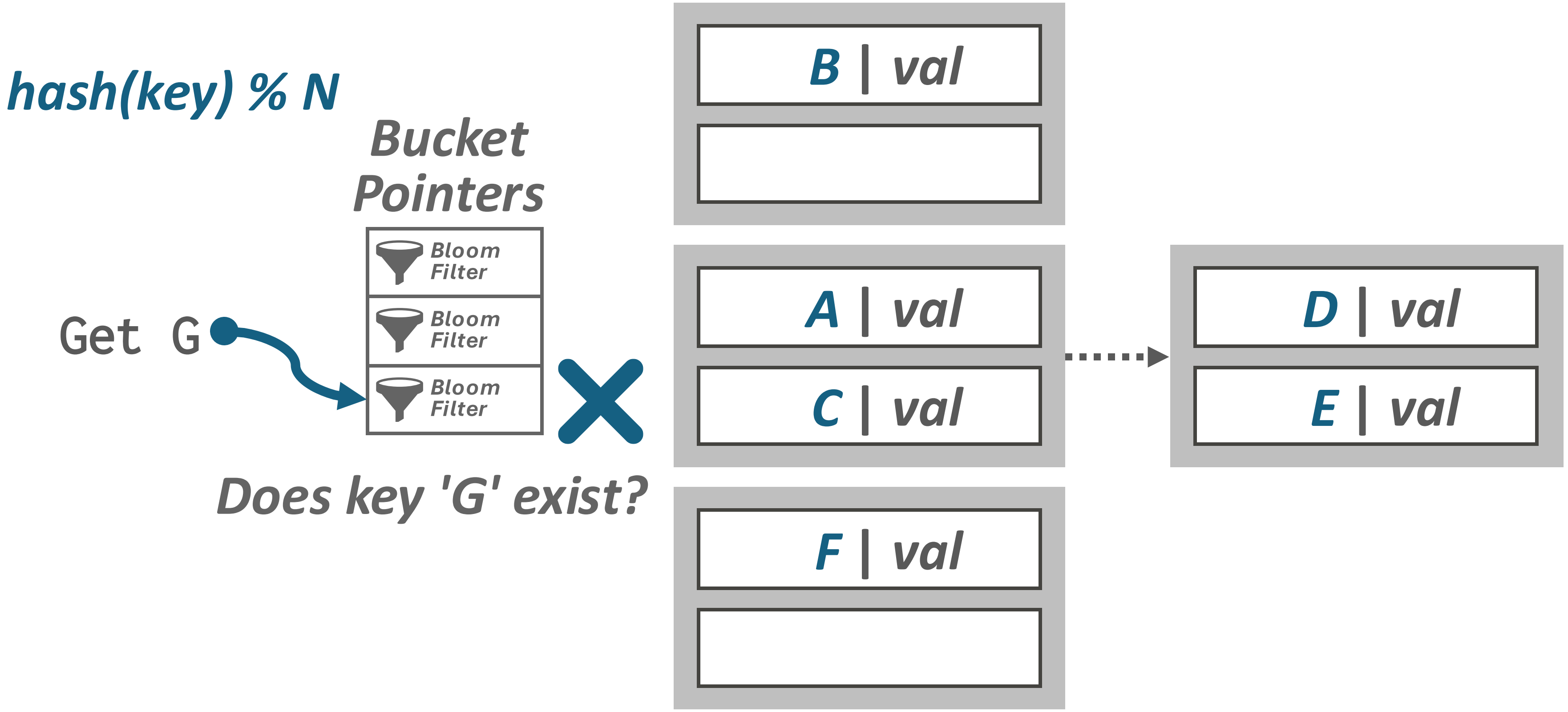
Chained Hashing



Chained Hashing



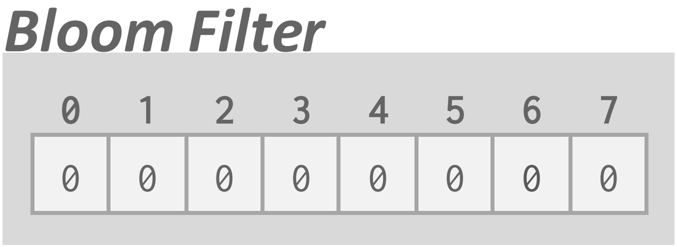
Chained Hashing



Bloom Filters

- Probabilistic data structure (bitmap) that answers set membership queries.
  - **False negatives** will **never occur**.
  - **False positives** can **sometimes occur**.
  - See Bloom Filter Calculator.
- **Insert(x):**
  - Use *k* hash functions to set bits in the filter to 1.
- **Lookup(x):**
  - Check whether the bits are 1 for each hash function.

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

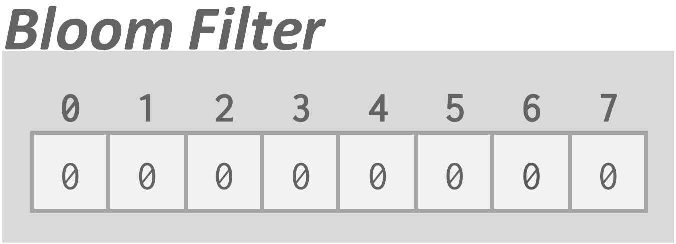
Bloom Filters

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

- Insert('RZA')

Bloom Filters



Bloom Filter

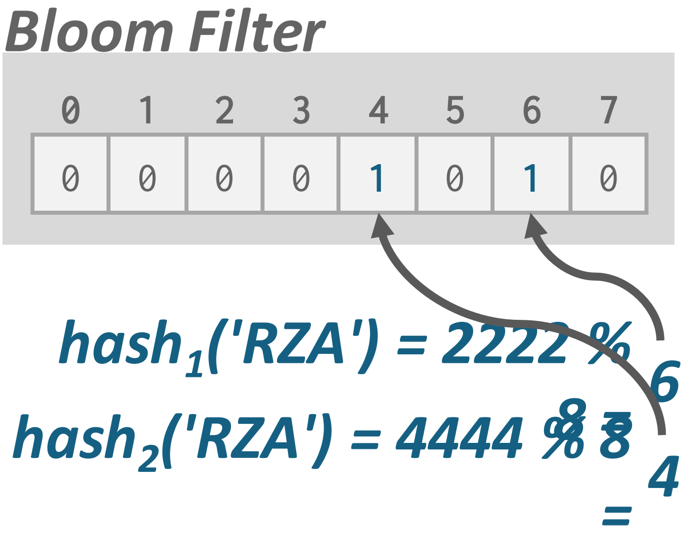
0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

$hash_1$('RZA') = 2222 % 8 = 6

$hash_2$('RZA') = 4444 % 8 = 4

- Insert('RZA')

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

$hash_1$('RZA') = 2222 % 8 = 6

$hash_2$('RZA') = 4444 % 8 = 4

- Insert('RZA')

Bloom Filters

Bloom Filter

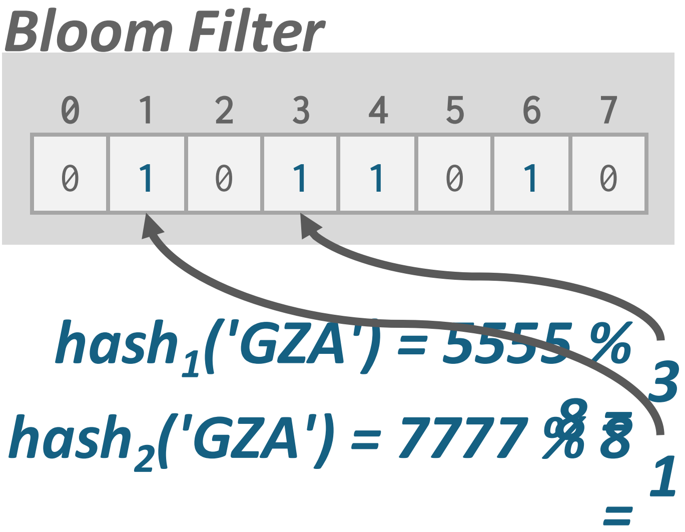
0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

$hash_1('GZA') = 5555 \% 8 = 3$

$hash_2('GZA') = 7777 \% 8 = 1$

- Insert('RZA')
- Insert('GZA')

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('GZA') = 5555 \% 8 = 3$

$hash_2('GZA') = 7777 \% 8 = 1$

- Insert('RZA')
- Insert('GZA')

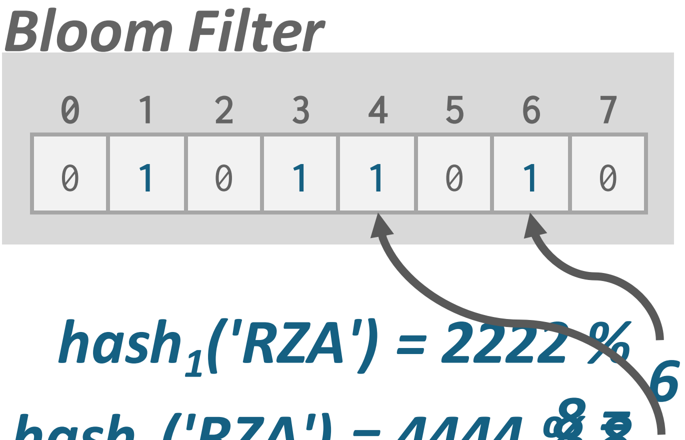
Bloom Filters

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA')

Bloom Filters



Bloom Filter

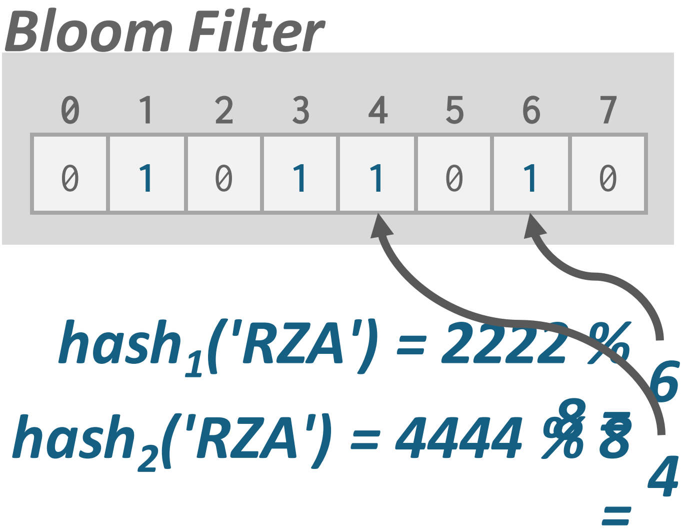
0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('RZA') = 2222 \% 8 = 6$

$hash_2('RZA') = 4444 \% 8 = 4$

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA')

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

hash$_1$('RZA') = 2222 % 8 = 6

hash$_2$('RZA') = 4444 % 8 = 4

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***

Bloom Filters

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon')

Bloom Filters

Bloom Filter

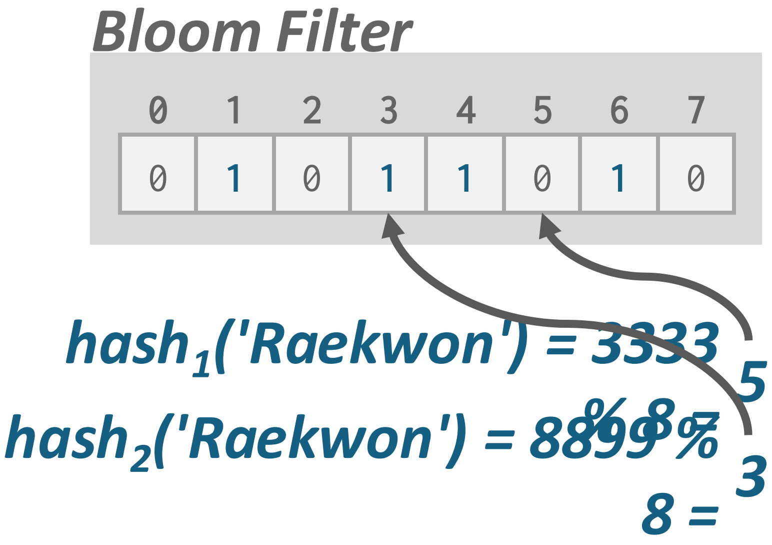
0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon')

Bloom Filters



Bloom Filter

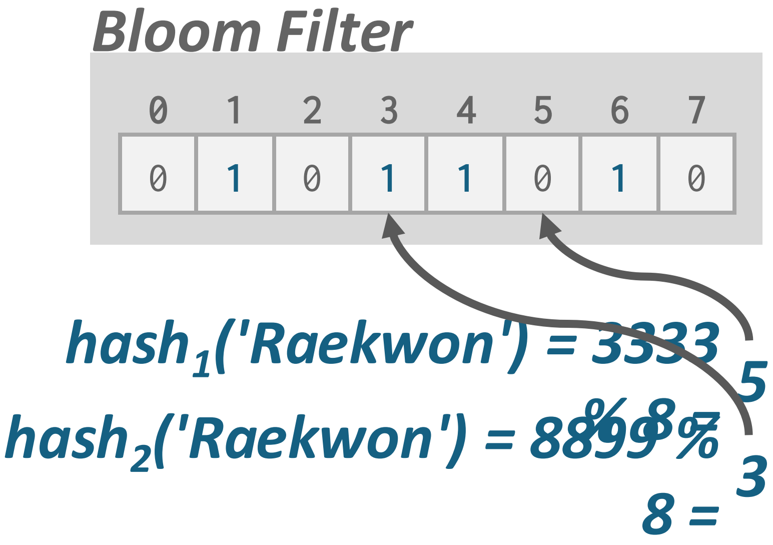
0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon')

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon') → ***FALSE***

Bloom Filters

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon') → ***FALSE***

Bloom Filters

Bloom Filter

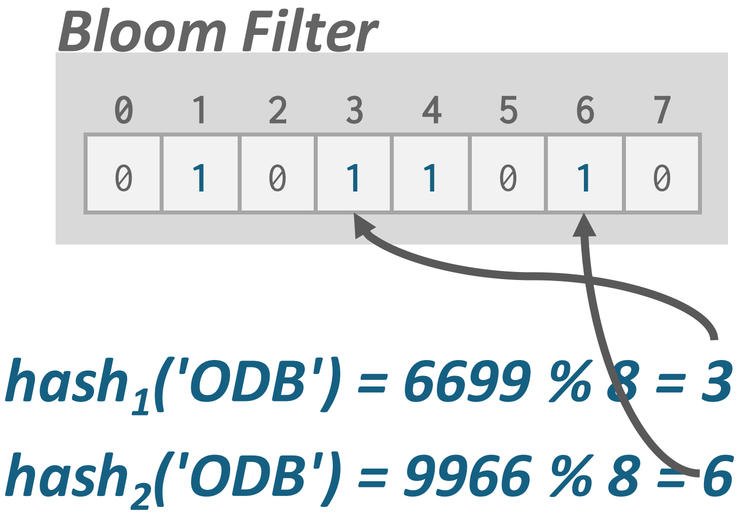
0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('ODB') = 6699 % 8 = 3

$hash_2$('ODB') = 9966 % 8 = 6

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon') → ***FALSE***
- Lookup('ODB')

Bloom Filters



Bloom Filter

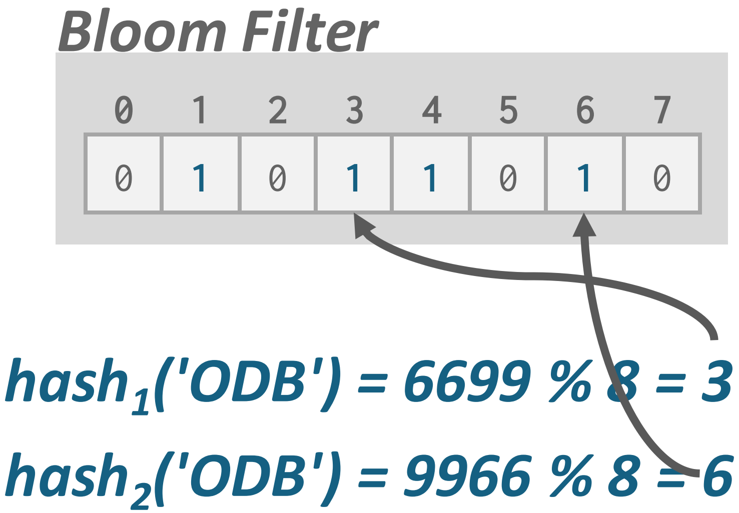
0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1('ODB') = 6699 \% 8 = 3$

$hash_2('ODB') = 9966 \% 8 = 6$

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon') → ***FALSE***
- Lookup('ODB')

Bloom Filters



Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

hash$_1$***('ODB') = 6699 % 8 = 3***

hash$_2$***('ODB') = 9966 % 8 = 6***

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → ***TRUE***
- Lookup('Raekwon') → ***FALSE***
- Lookup('ODB') → ***TRUE*** ***(false positive)***

Extendible Hashing

Dynamic Hashing Schemes

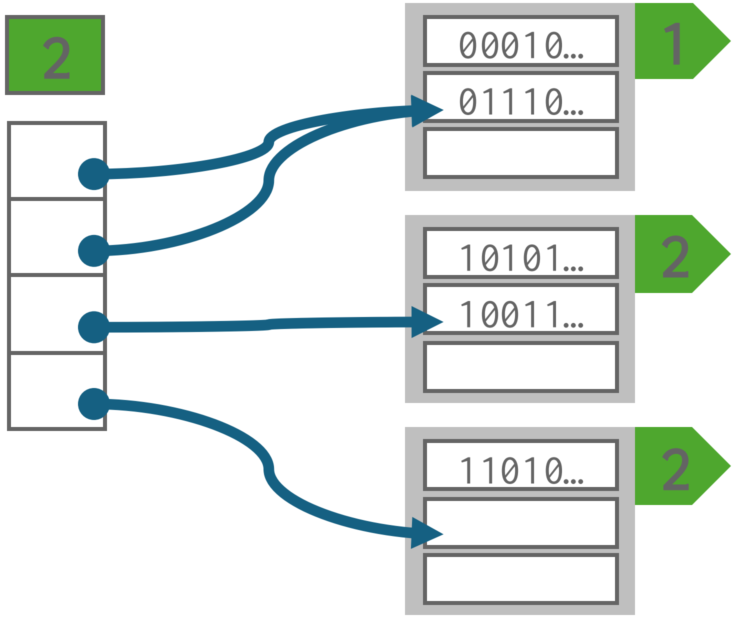
Extendible Hashing

- Extendible-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.
- Multiple slot locations can point to the same bucket chain.
- Reshuffle bucket entries on split and increase the number of bits to examine.
    - Data movement is localized to just the split chain.

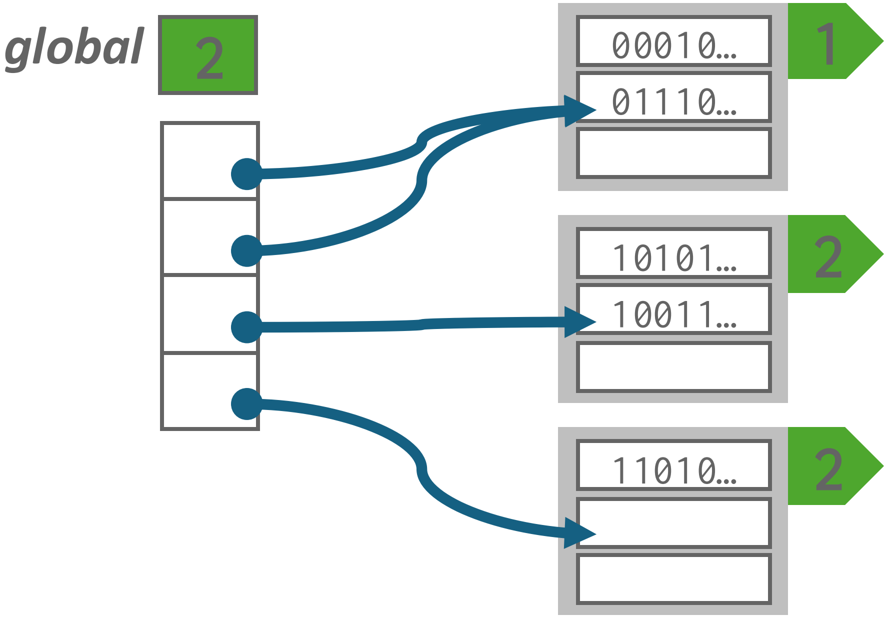
GDBM

AsterixDB

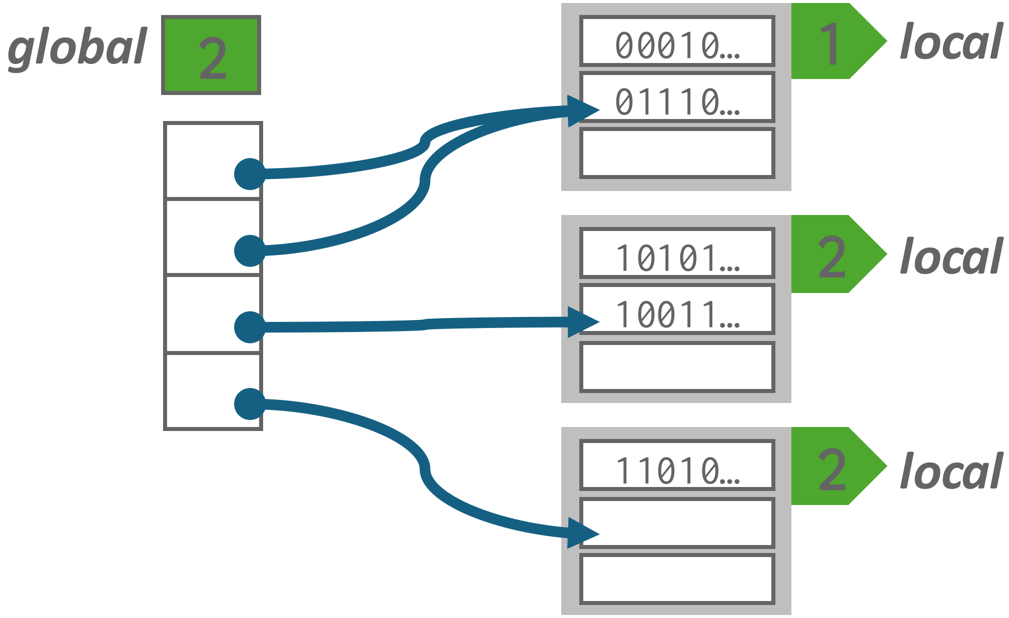
Extendible Hashing



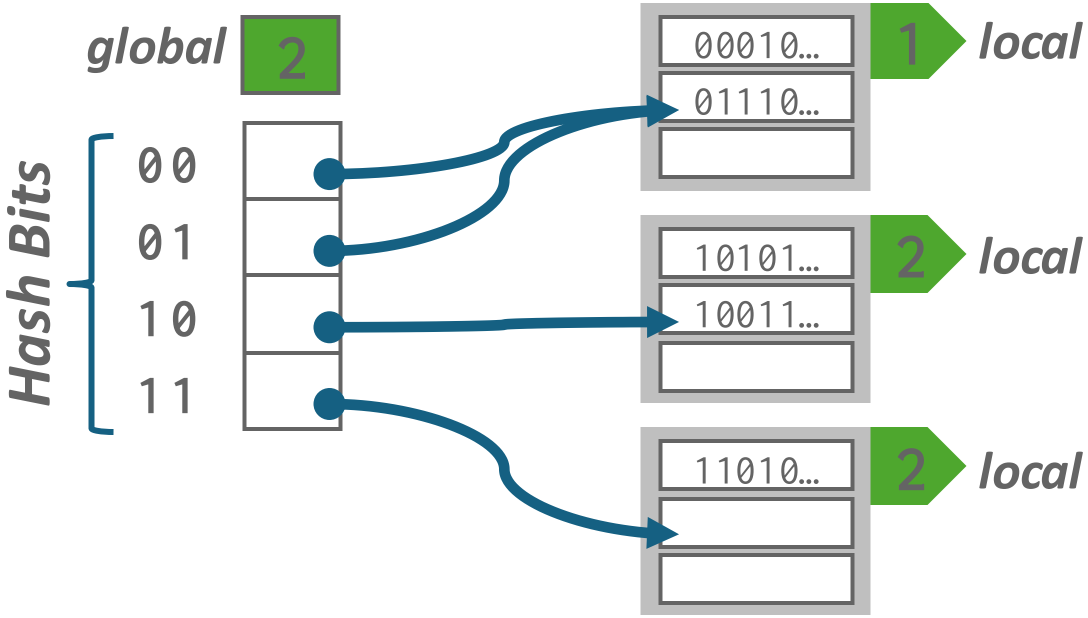
Extendible Hashing



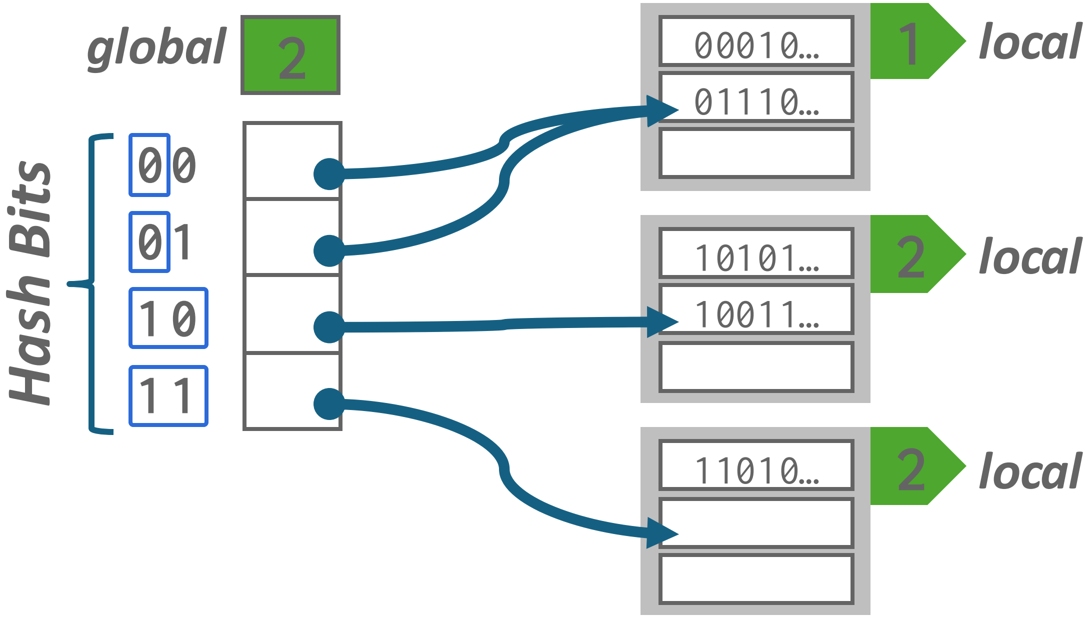
Extendible Hashing



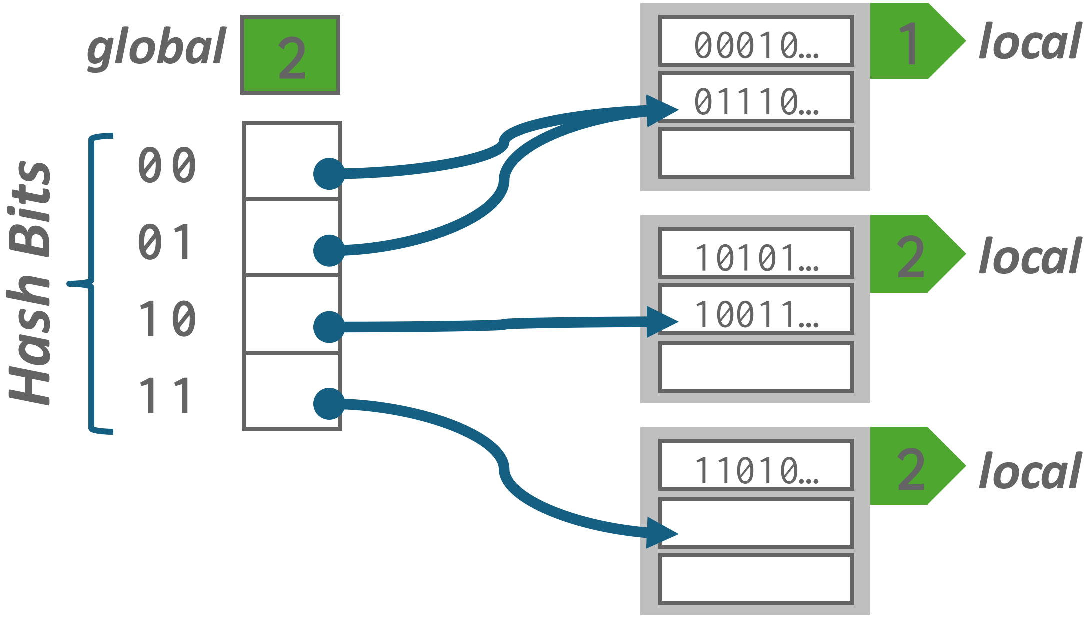
Extendible Hashing



Extendible Hashing



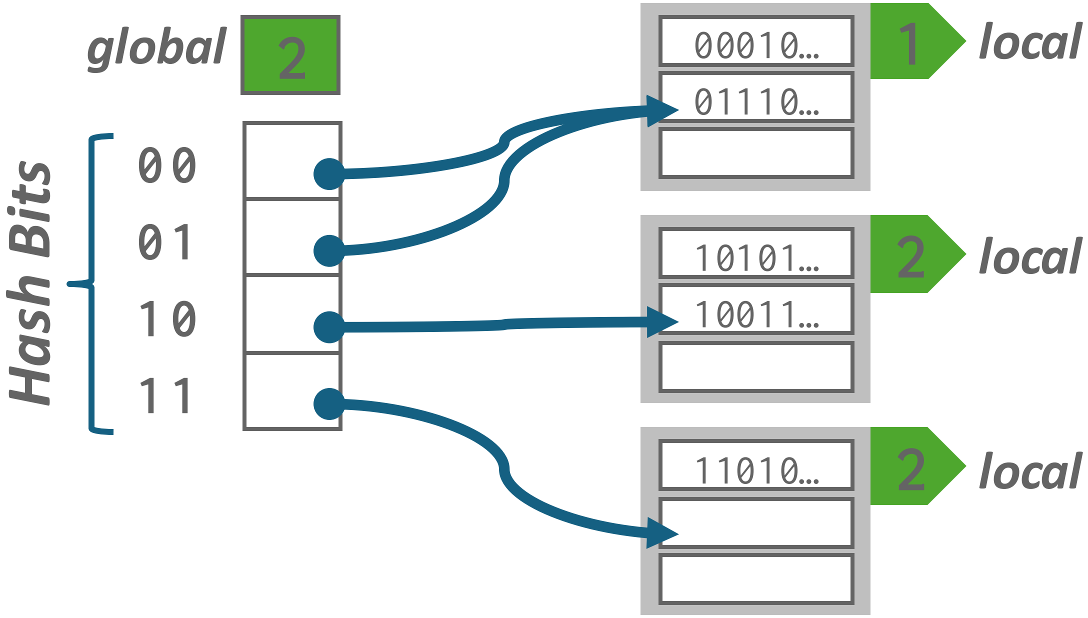
Extendible Hashing



Get A

hash(A) = 01110…

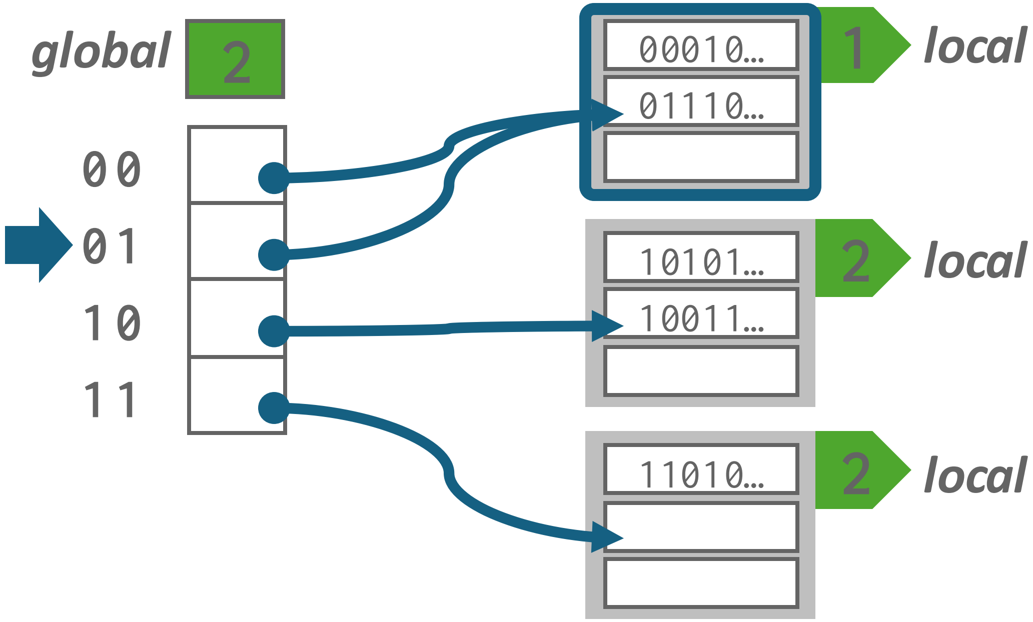
Extendible Hashing



Get A

hash(A) = $\boxed{01}$110…

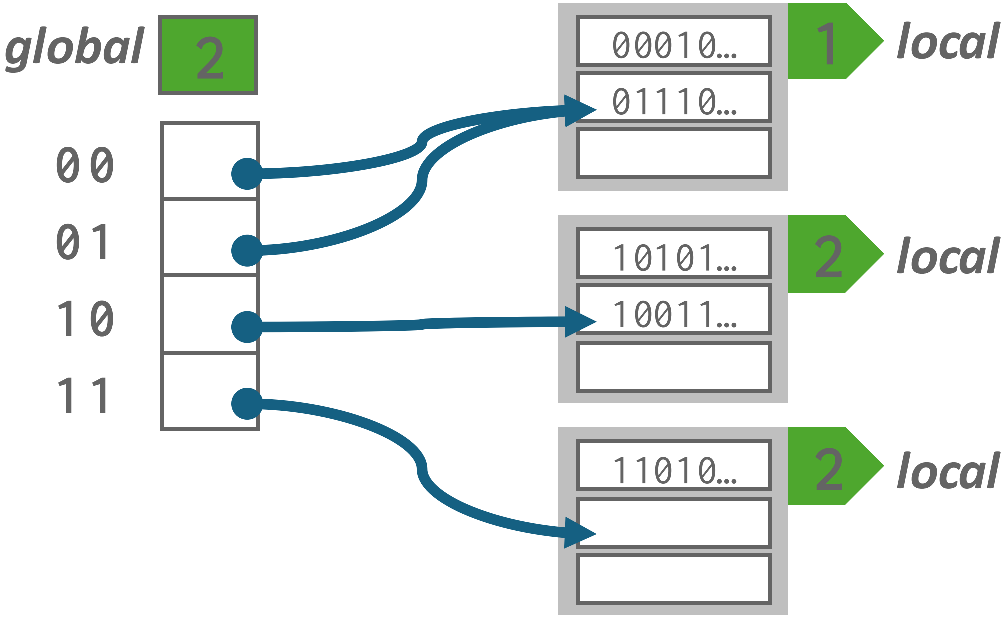
Extendible Hashing



Get A

hash(A) = $\boxed{01}$110…

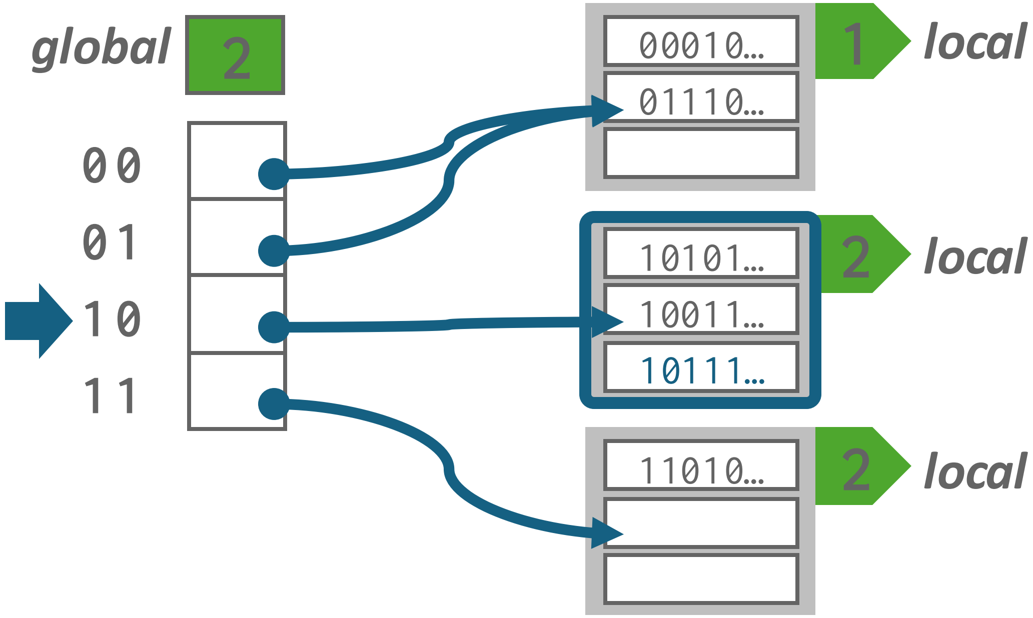
Extendible Hashing



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

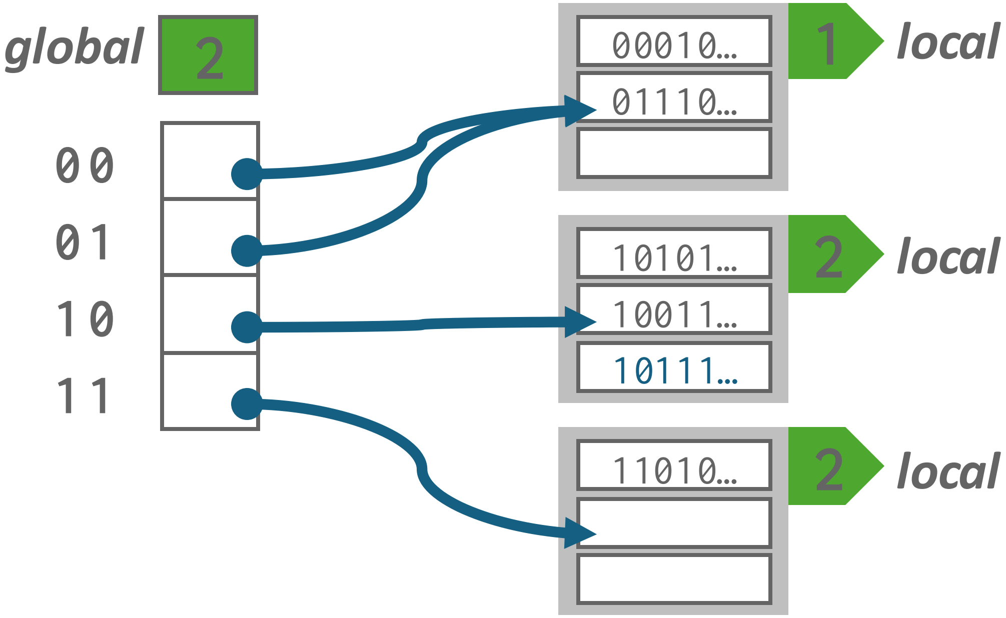
Extendible Hashing



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Extendible Hashing

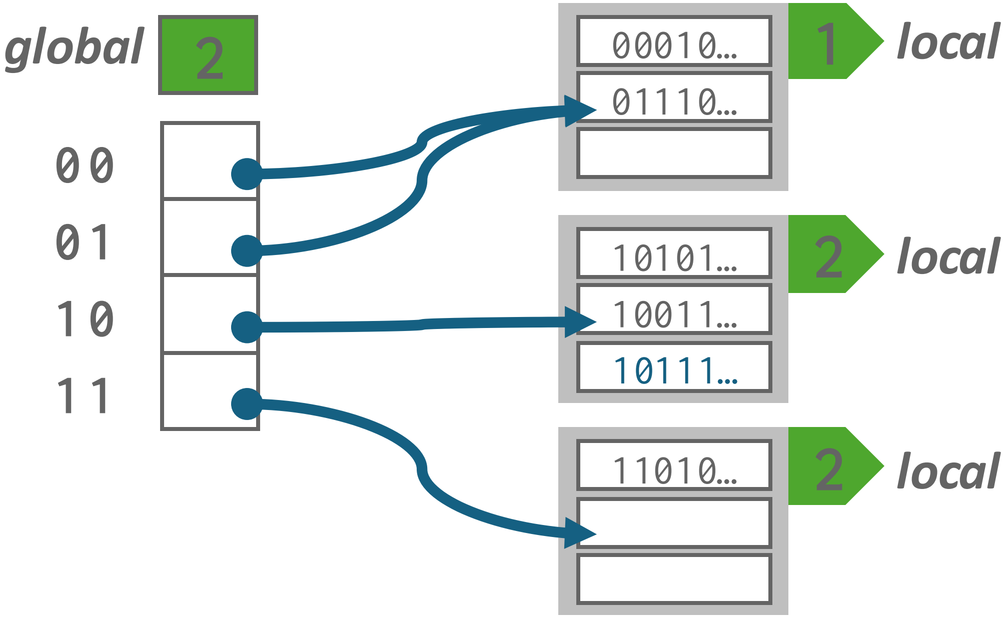


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

Extendible Hashing

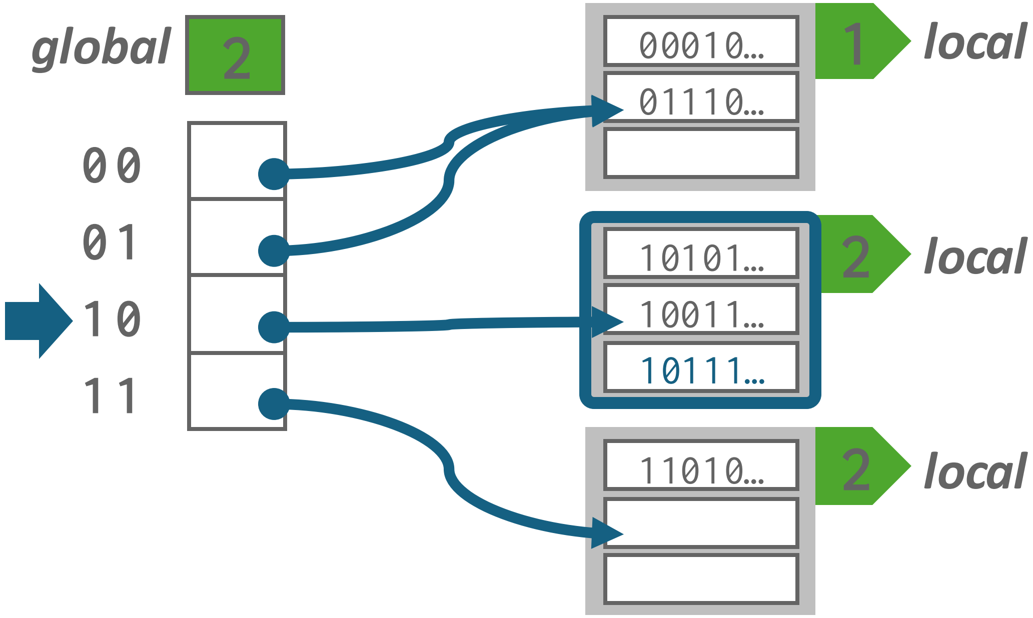


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

Extendible Hashing



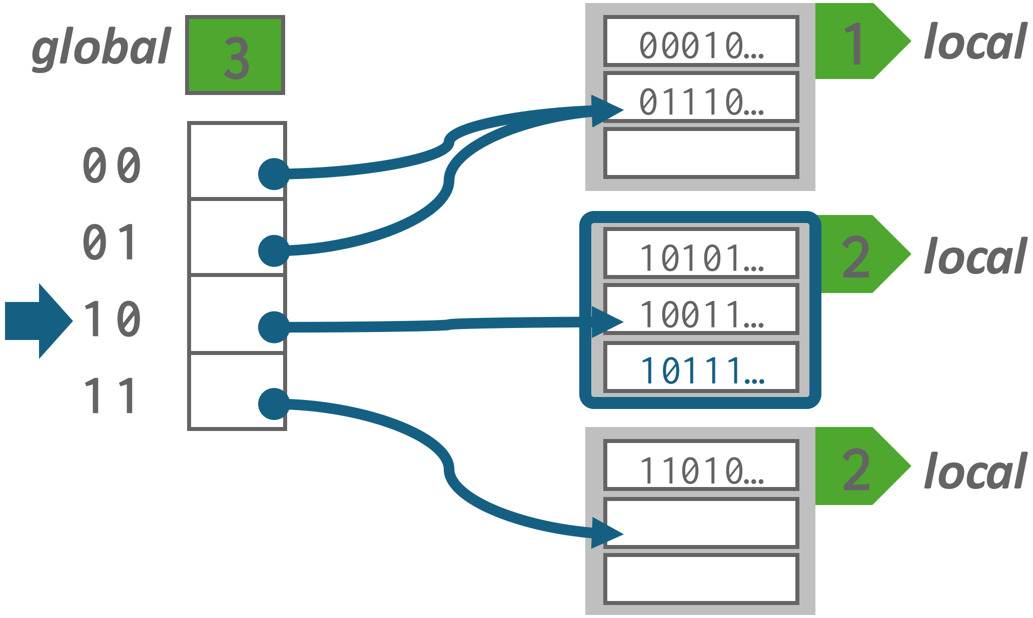
Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

SCHOOL OF DATA SCIENCE

Extendible Hashing

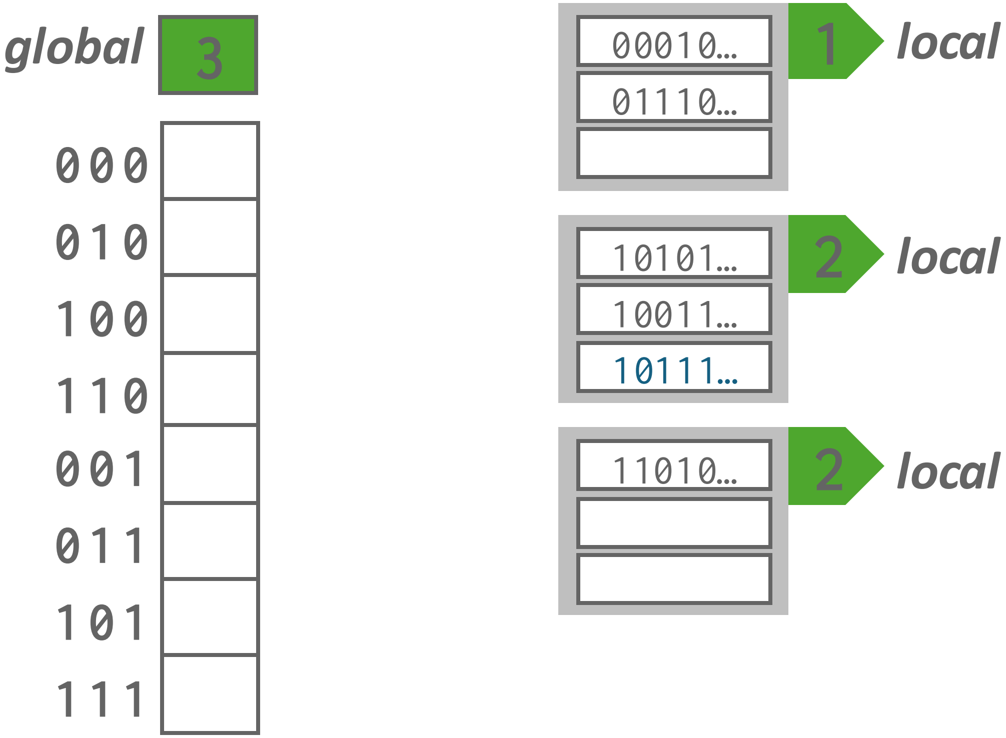


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

Extendible Hashing

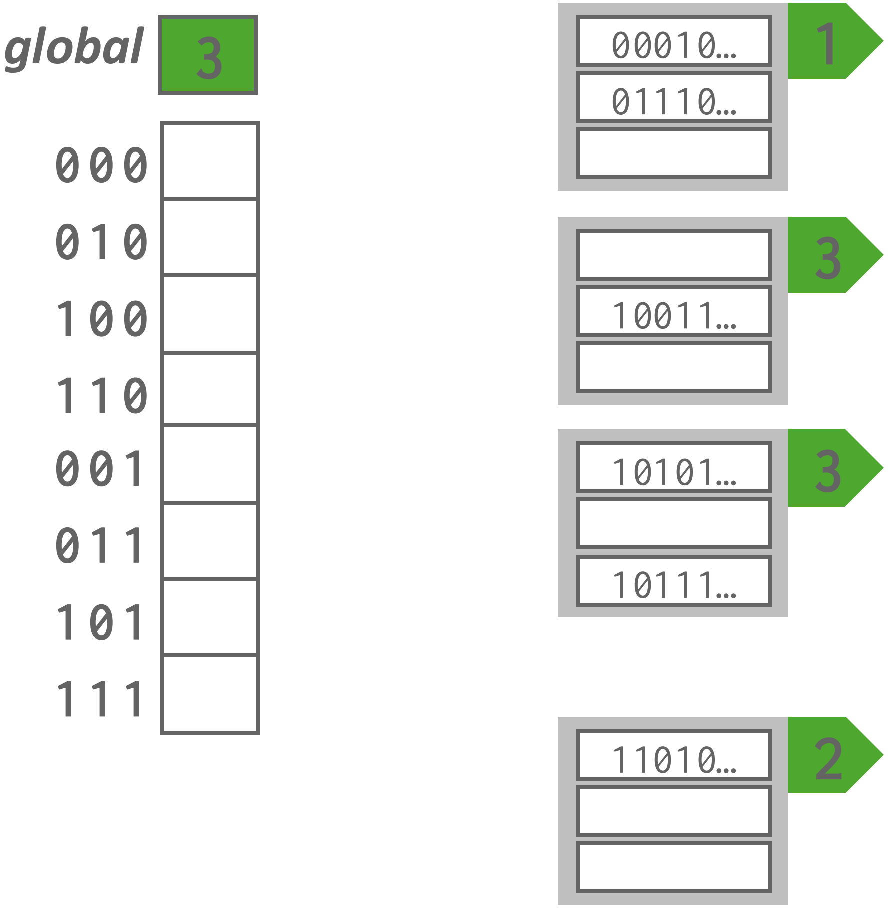


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

Extendible Hashing

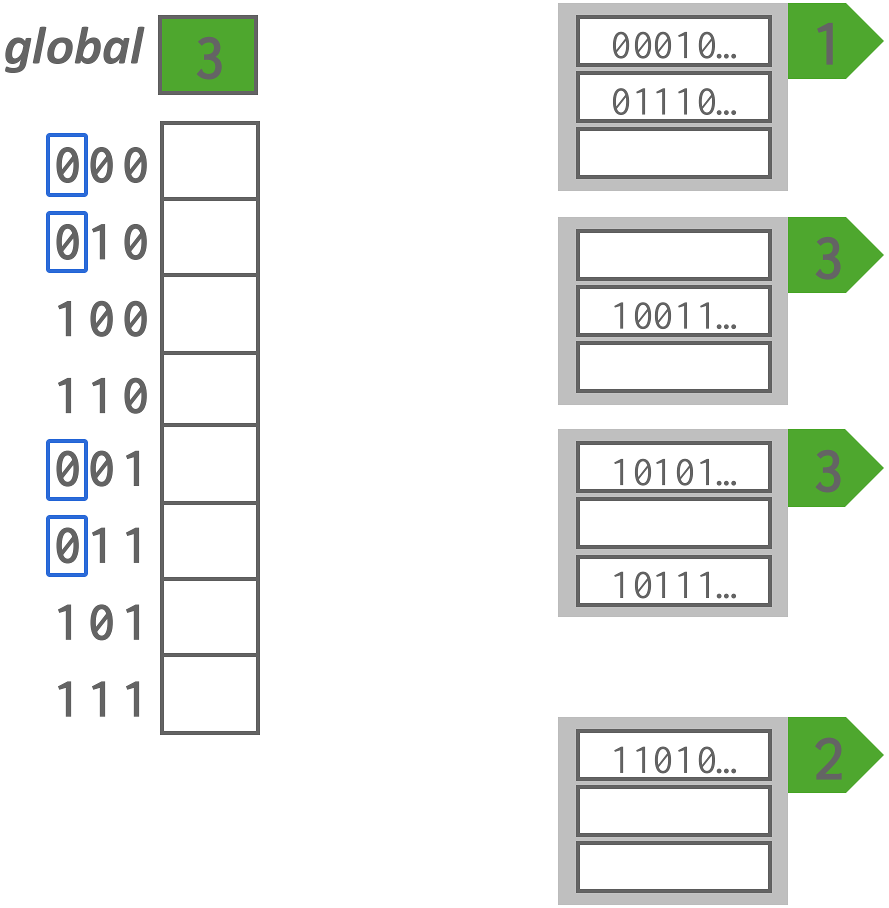


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

Extendible Hashing

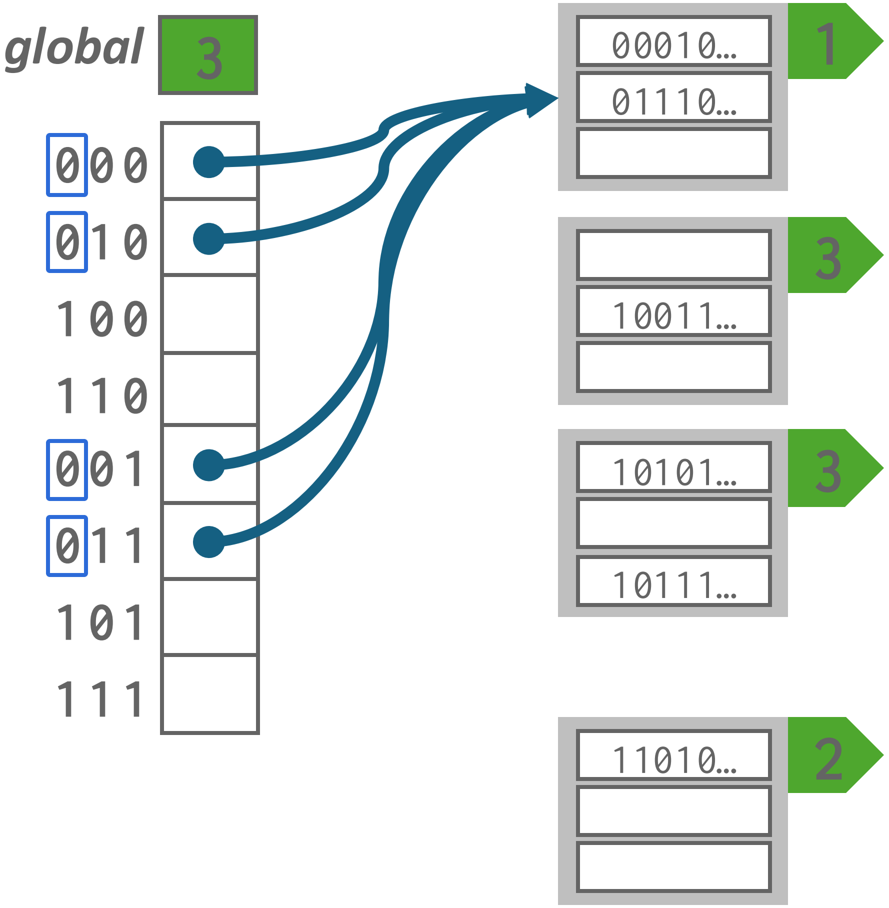


Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

Extendible Hashing

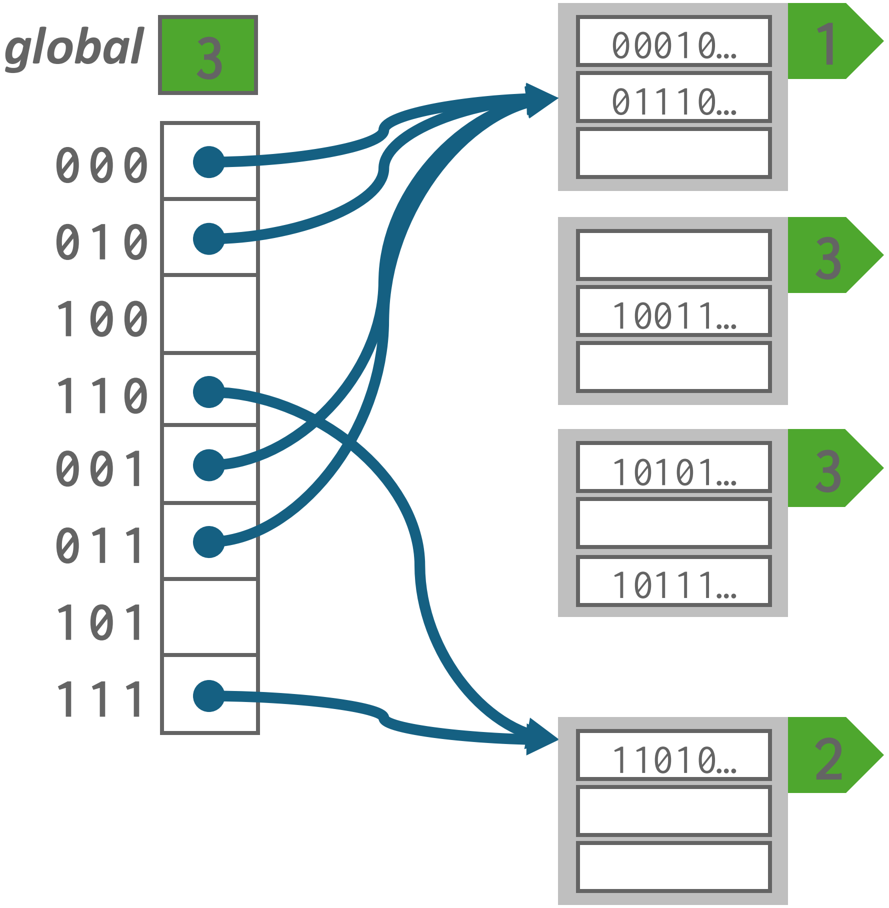


Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

Extendible Hashing

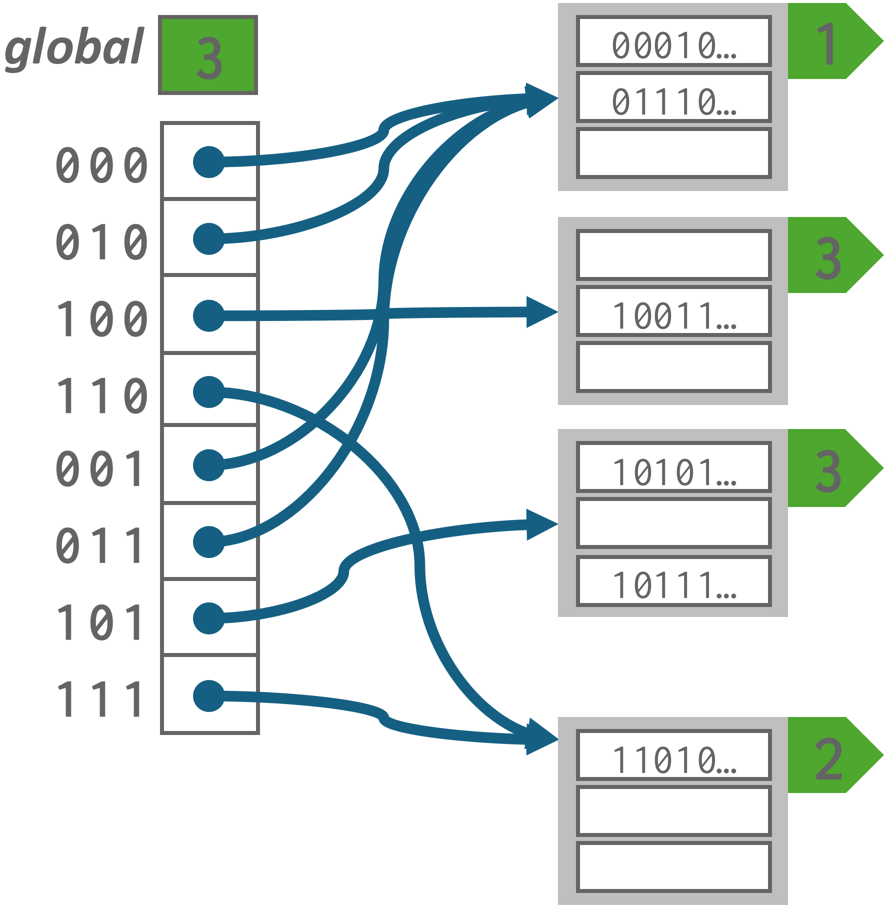


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

Extendible Hashing

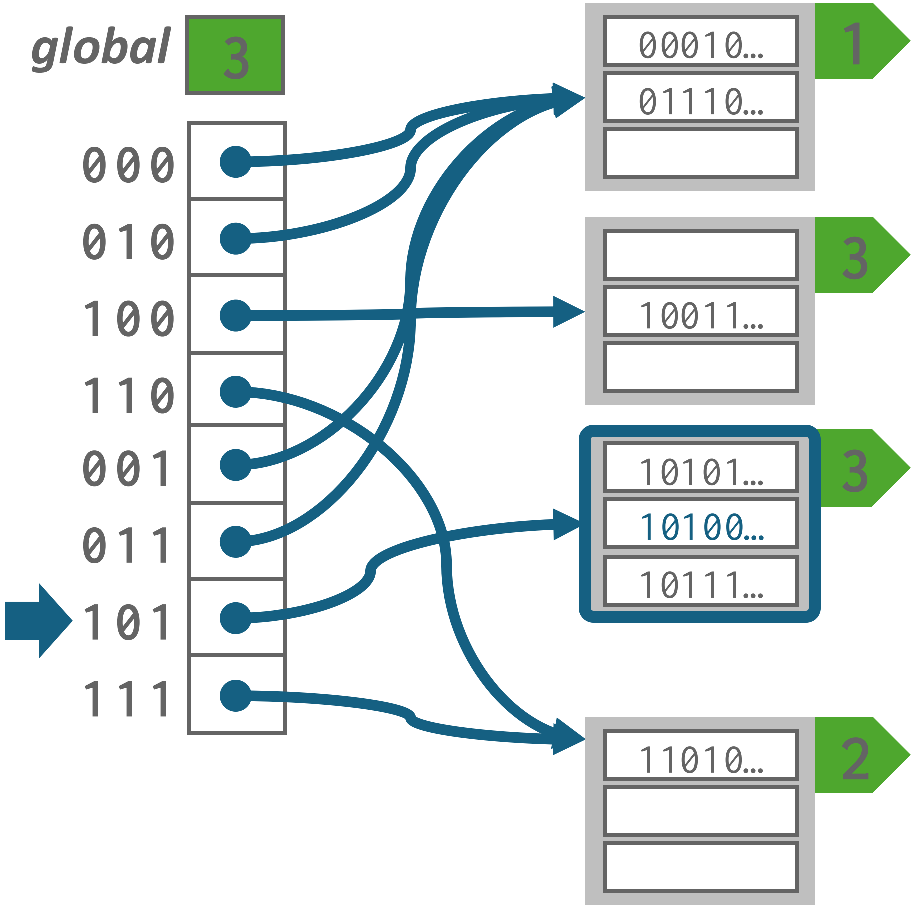


Get A
hash(A) = 01110…

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

Dynamic Hashing Schemes

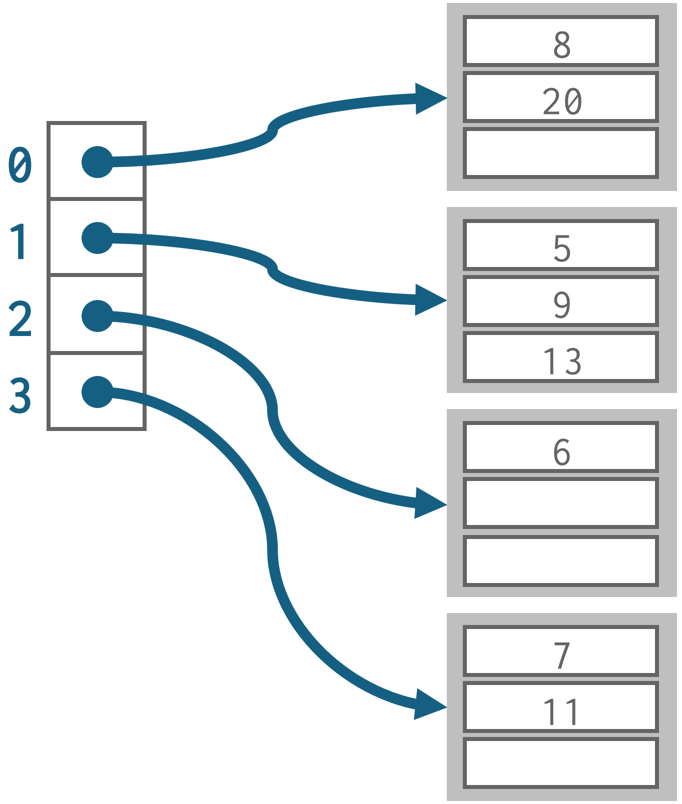
Linear Hashing

- 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

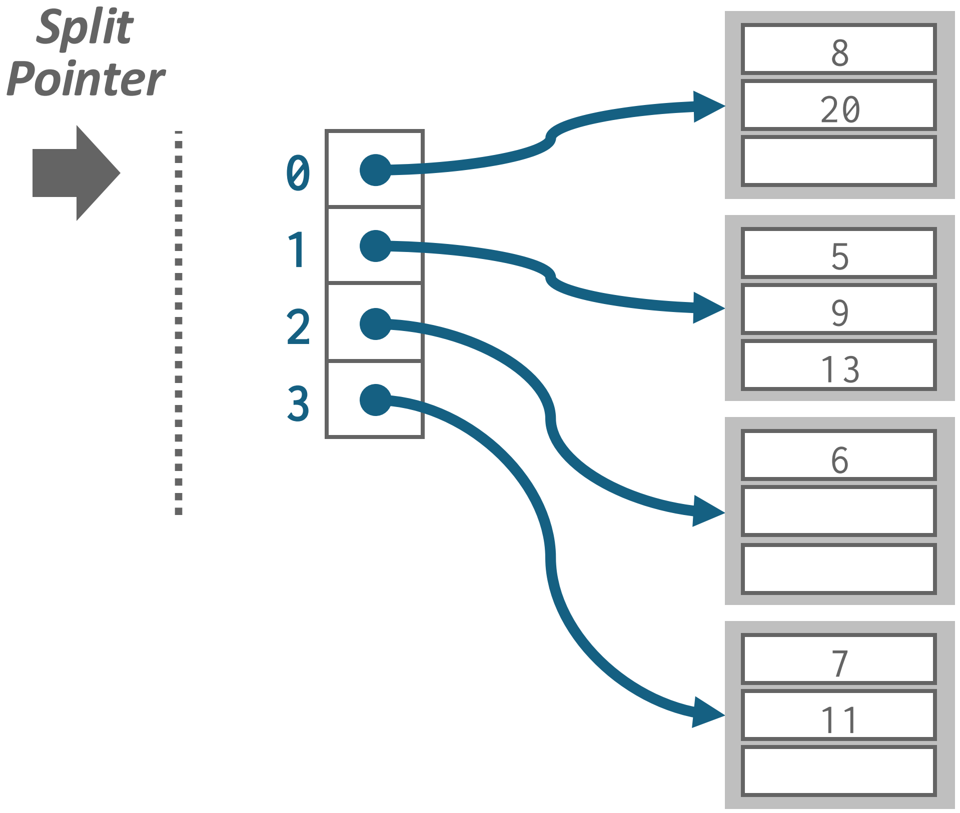
SLEEPYCAT
SOFTWARE
Makers of Berkeley DB

PostgreSQL

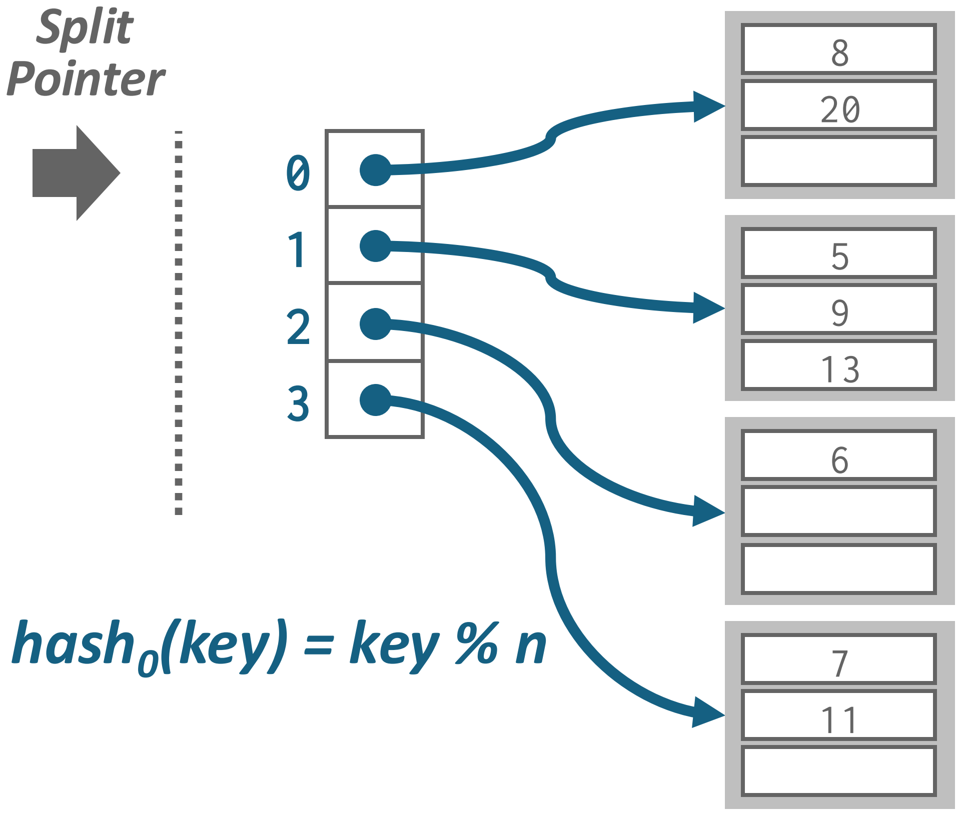
Linear Hashing



Linear Hashing

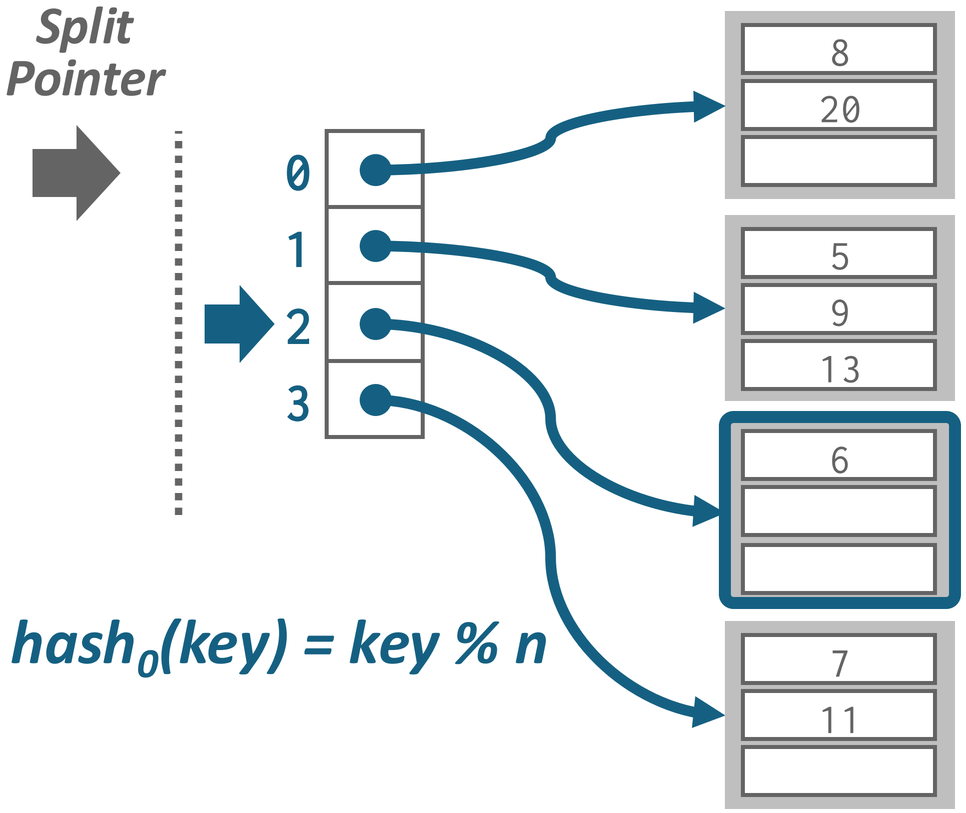


Linear Hashing



hash$_0$(key) = key % n

Linear Hashing

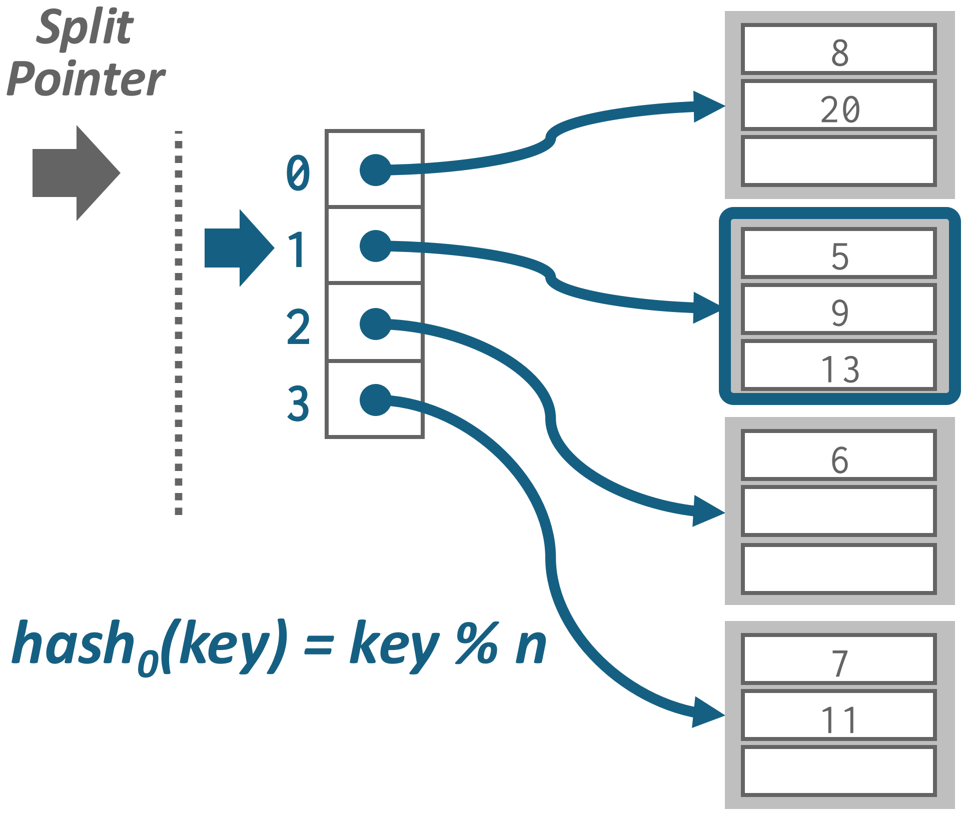


$hash_0(key) = key \% n$

Get 6

$hash_0(6) = 6 \% 4 = 2$

Linear Hashing



$hash_0$(key) = key % n

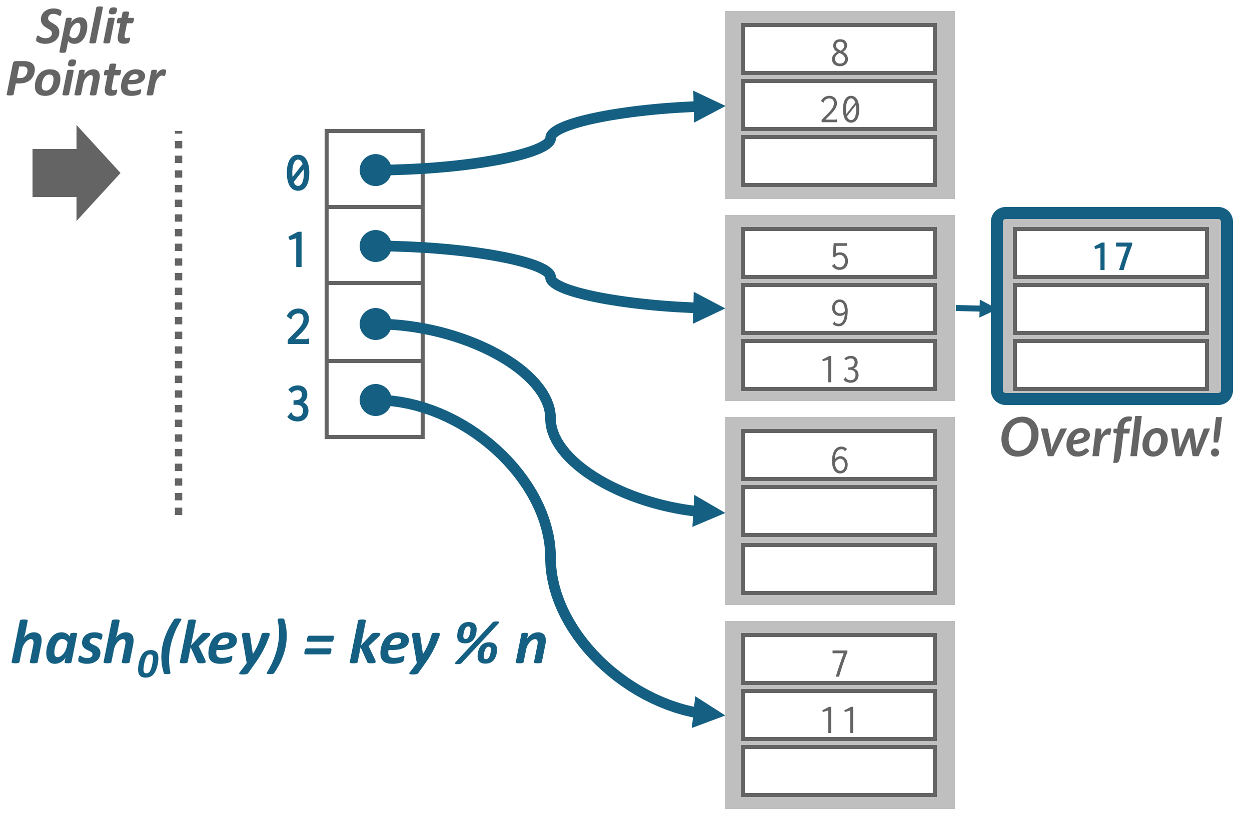
Get 6

$hash_0$(6) = 6 % 4 = 2

Put 17

$hash_0$(17) = 17 % 4 = 1

Linear Hashing



$hash_0(key) = key \% n$

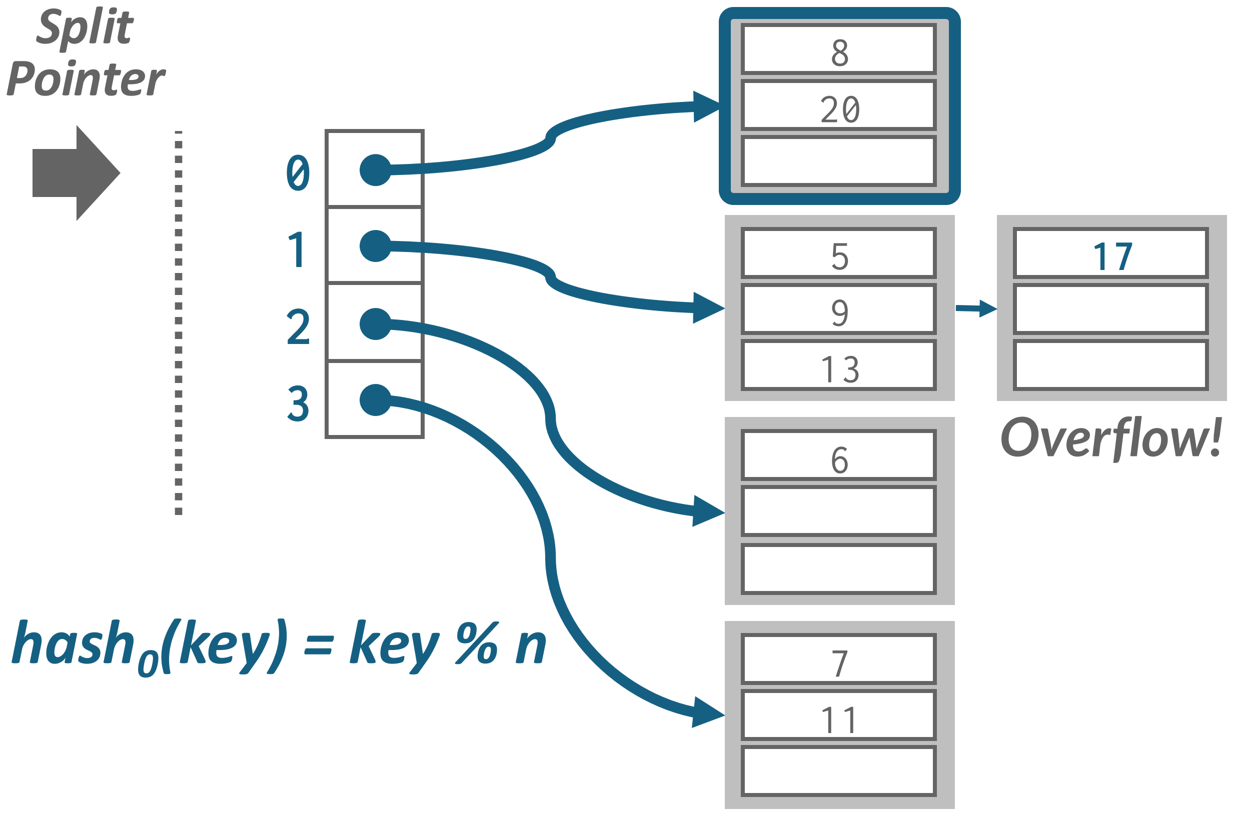
Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

Linear Hashing



$hash_0(key) = key \% n$

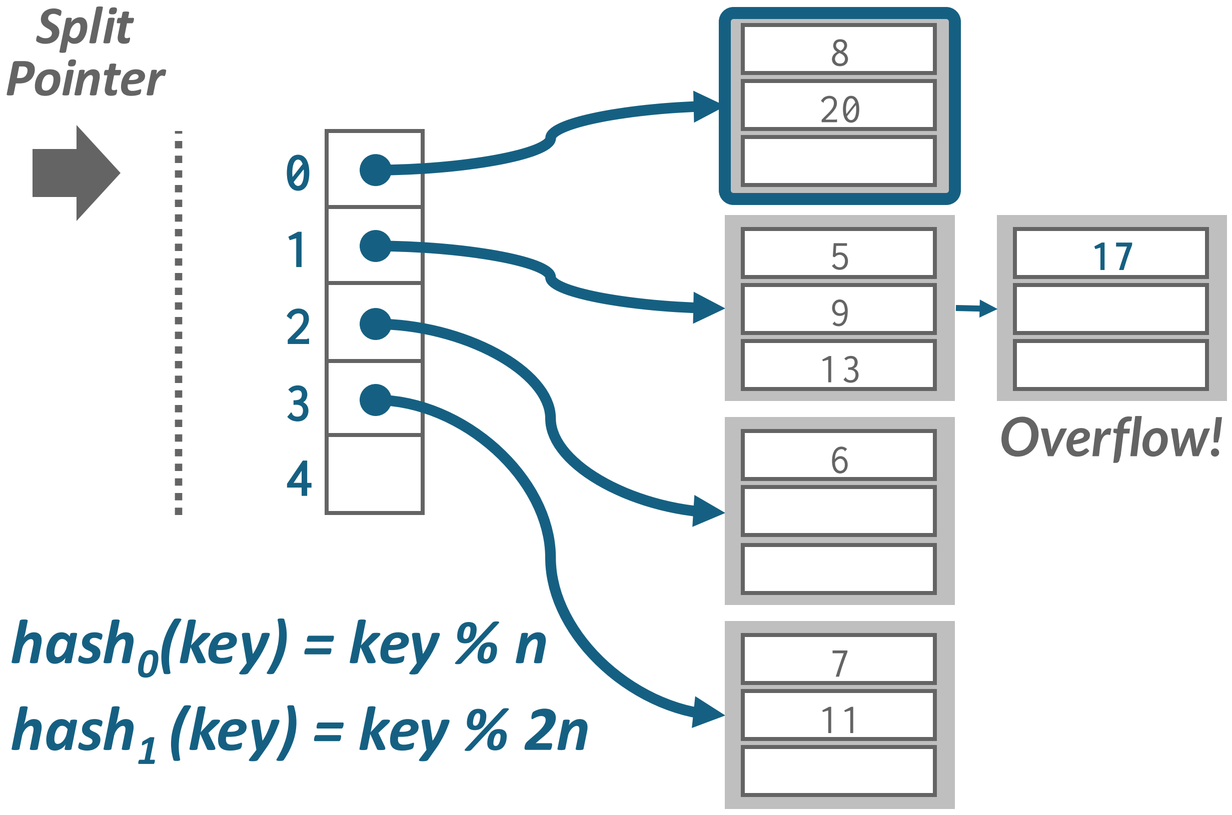
Get 6

$hash_0(6)$ = 6 % 4 = 2

Put 17

$hash_0(17)$ = 17 % 4 = 1

Linear Hashing



$hash_0(key) = key \% n$

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

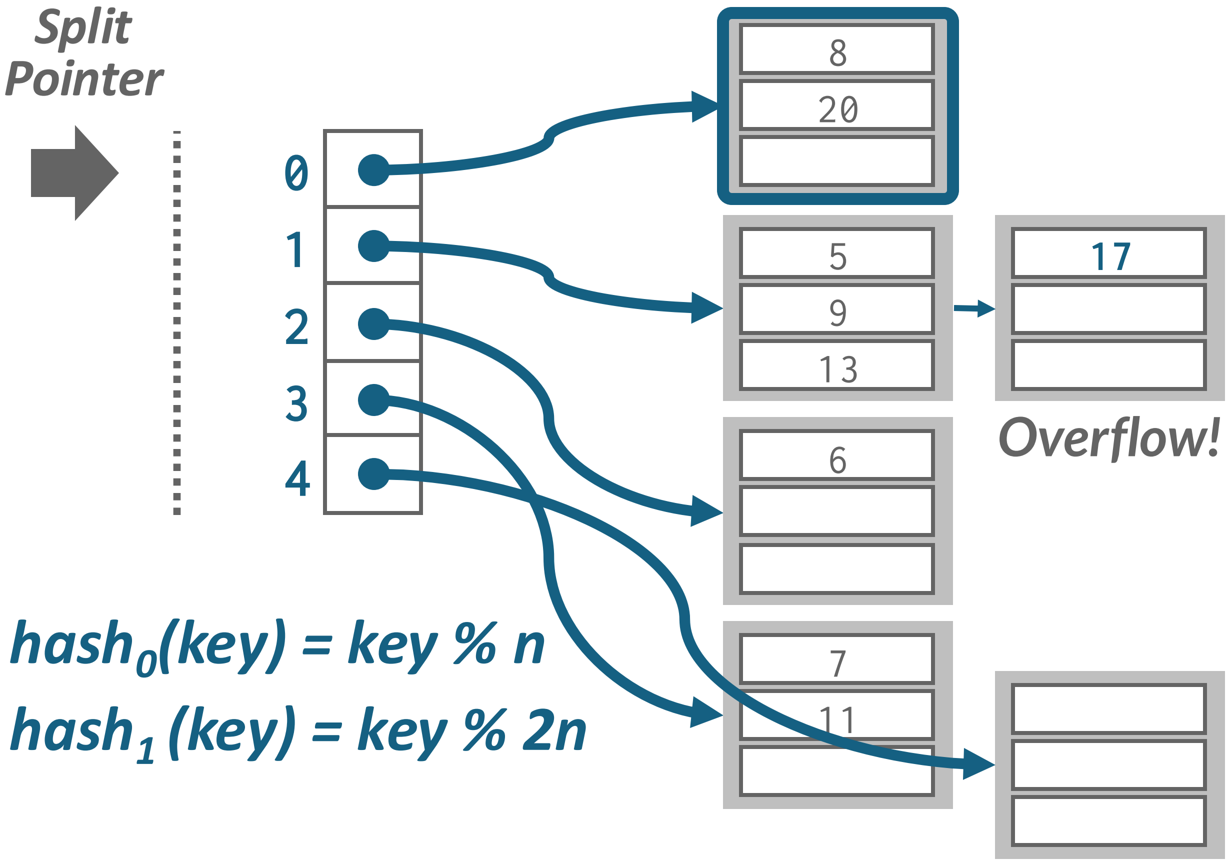
Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

Linear Hashing



$hash_0$(key) = key % n

$hash_1$ (key) = key % 2n

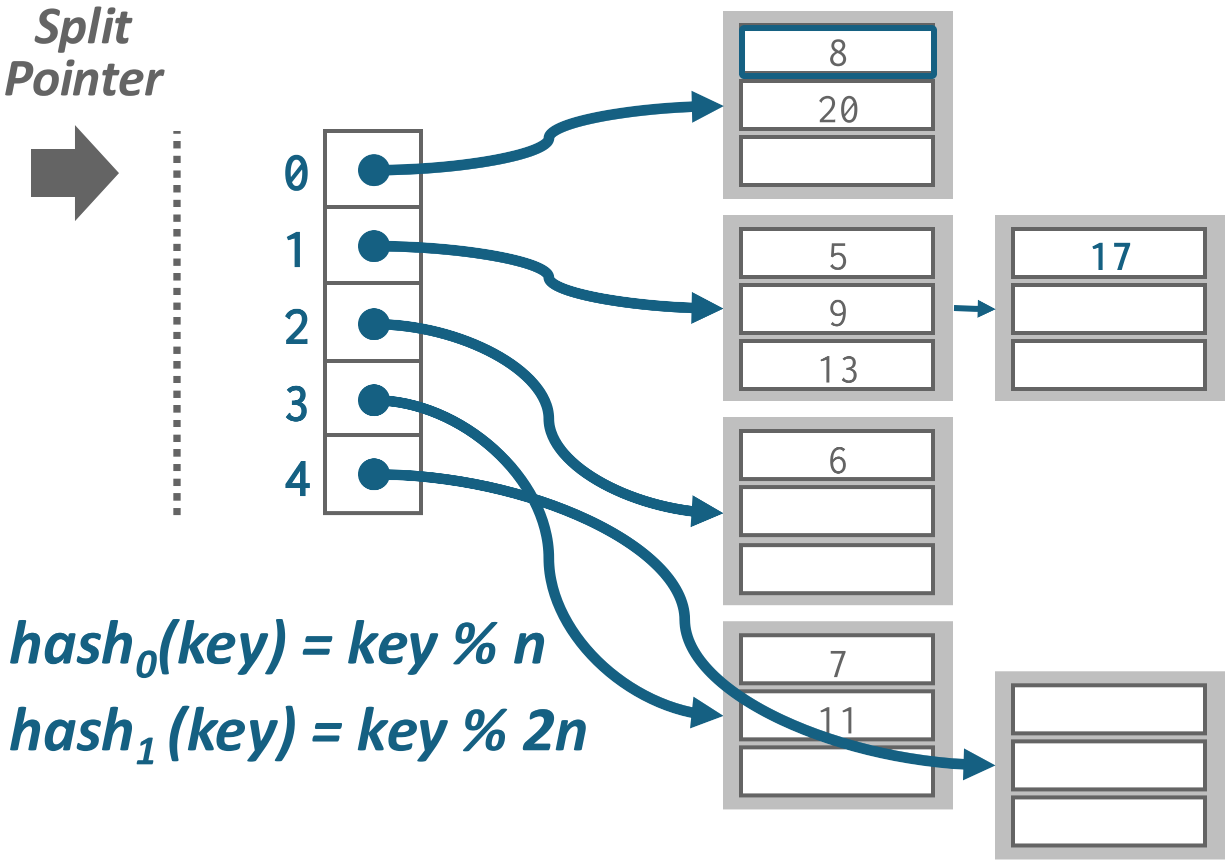
Get 6

$hash_0$(6) = 6 % 4 = 2

Put 17

$hash_0$(17) = 17 % 4 = 1

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

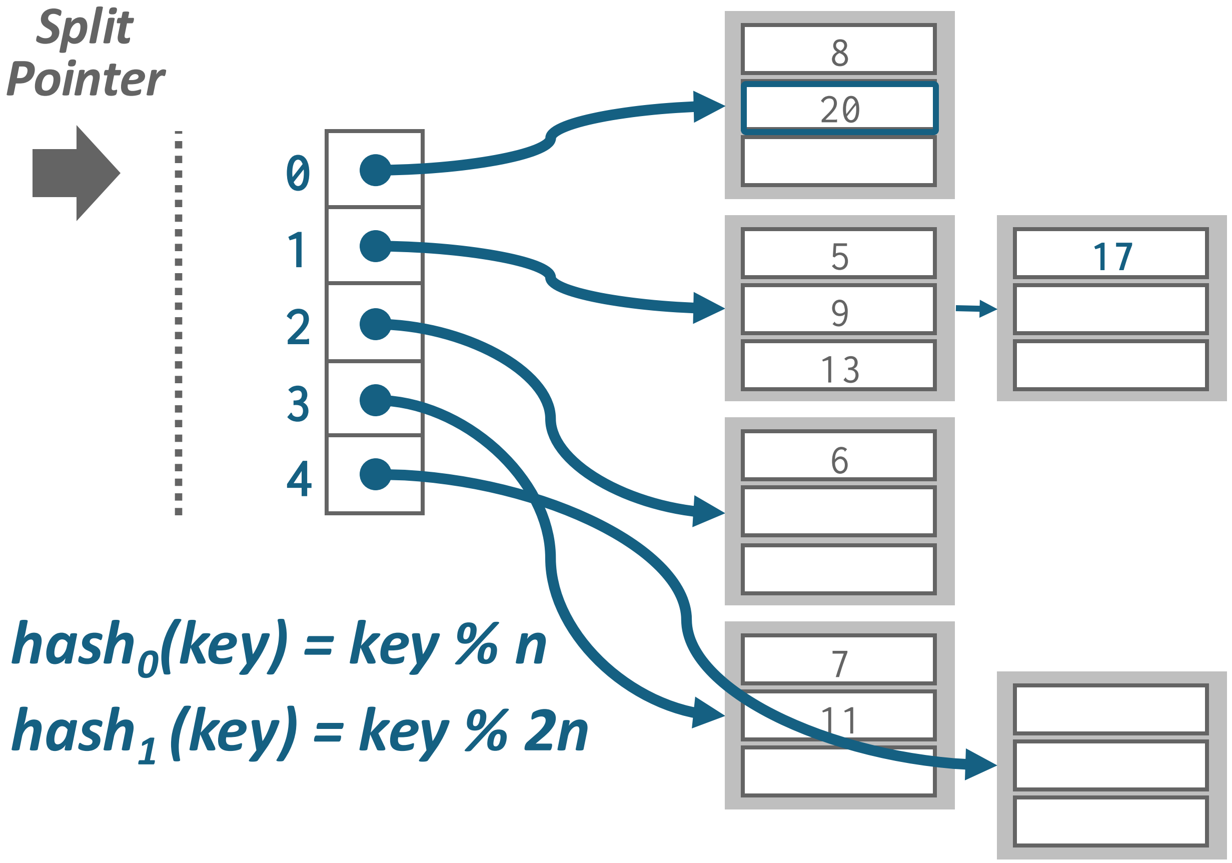
$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6) = 6 \% 4 = 2$

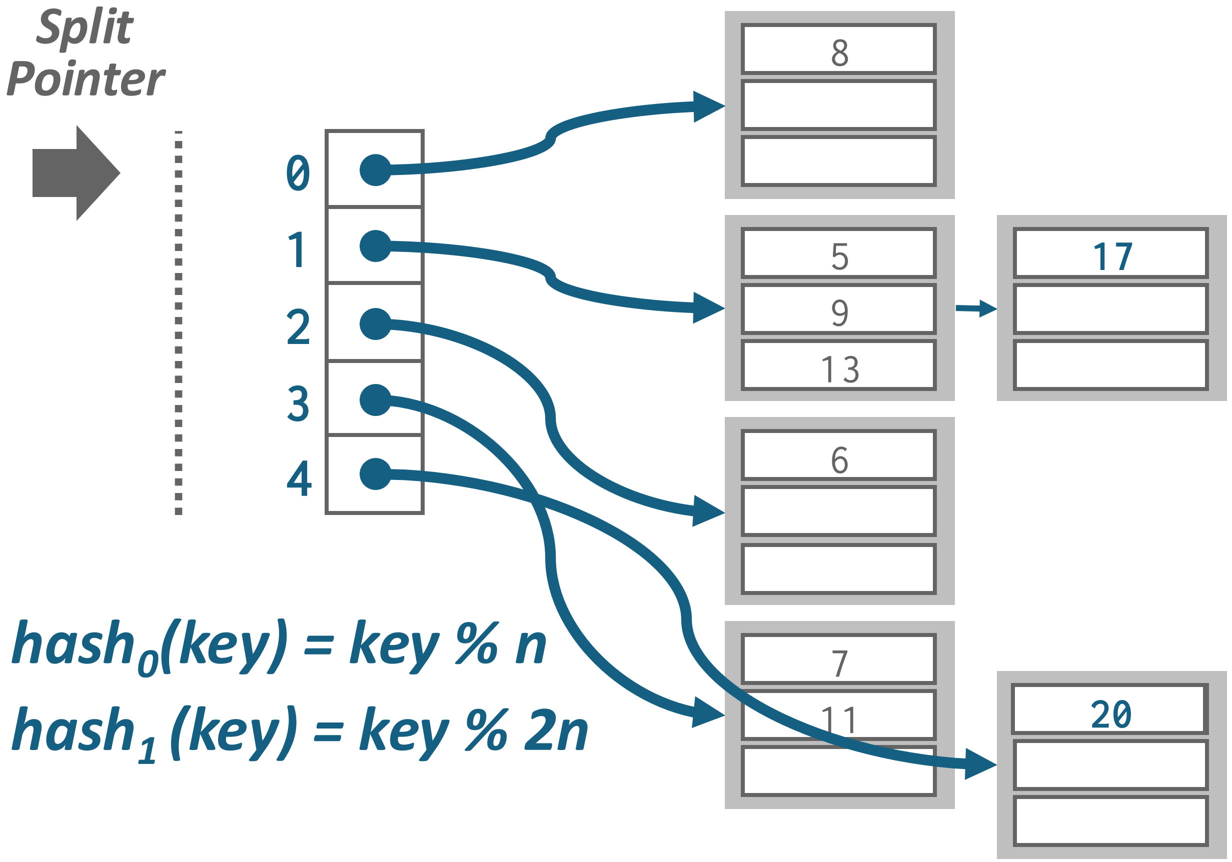
Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6) = 6 \% 4 = 2$

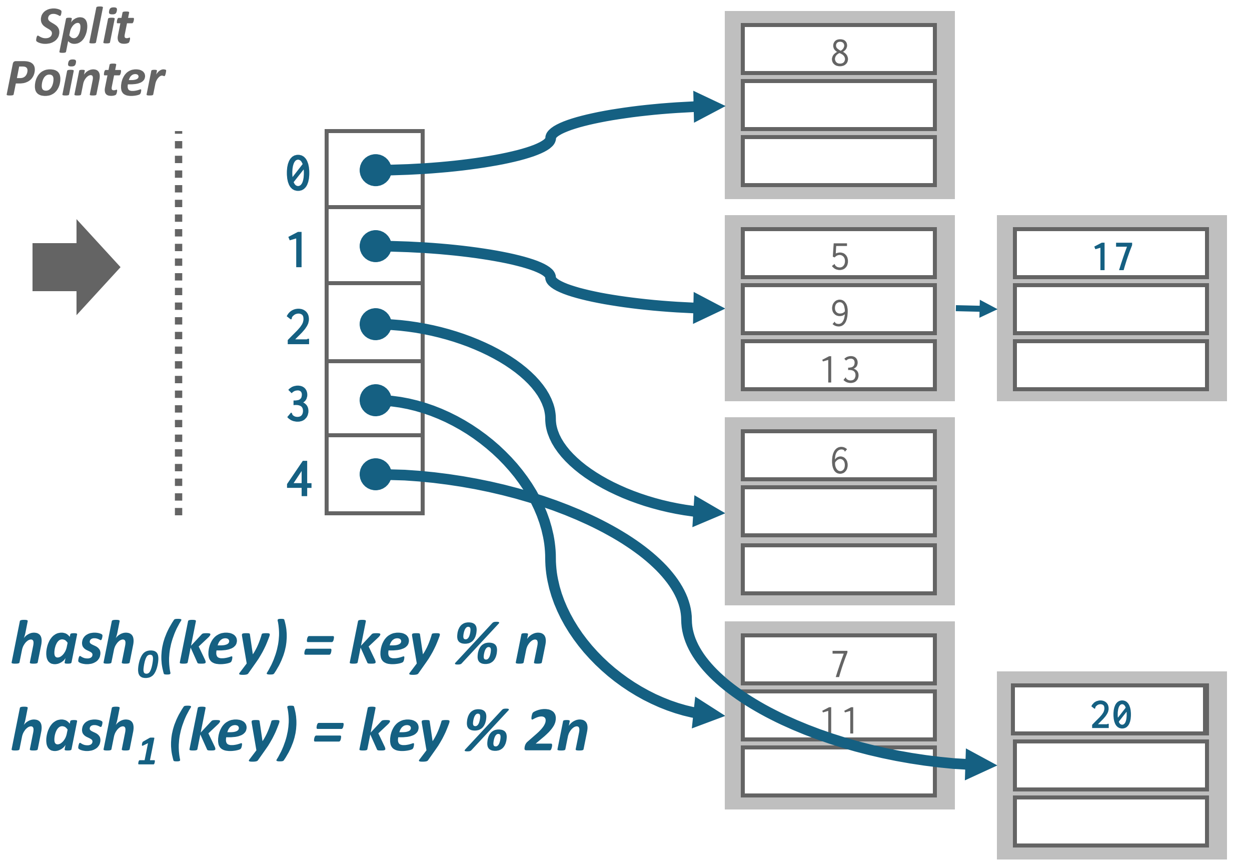
Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing



$hash_0$(key) = key % n

$hash_1$(key) = key % 2n

Get 6

$hash_0$(6) = 6 % 4 = 2

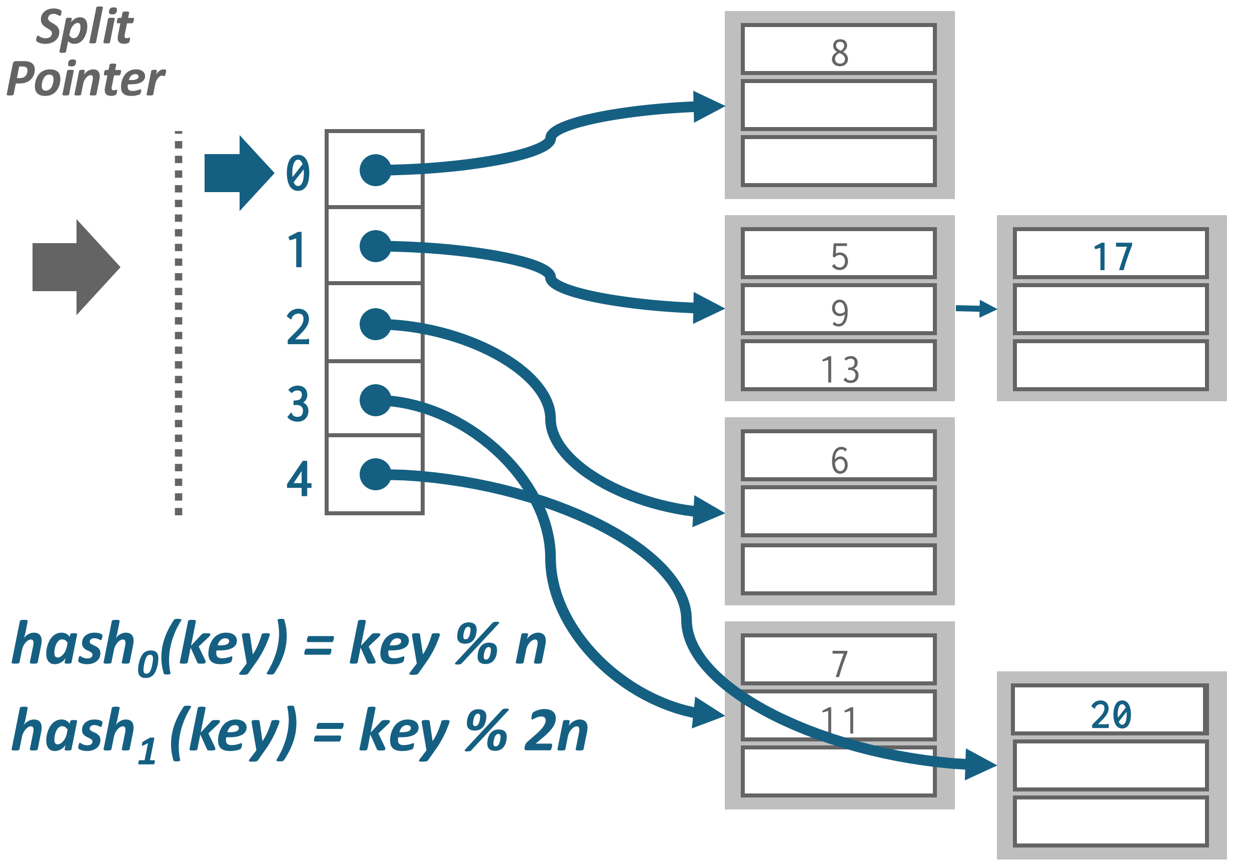
Put 17

$hash_0$(17) = 17 % 4 = 1

$hash_1$(8) = 8 % 8 = 0

$hash_1$(20) = 20 % 8 = 4

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6)$ = 6 % 4 = 2

Put 17

$hash_0(17)$ = 17 % 4 = 1

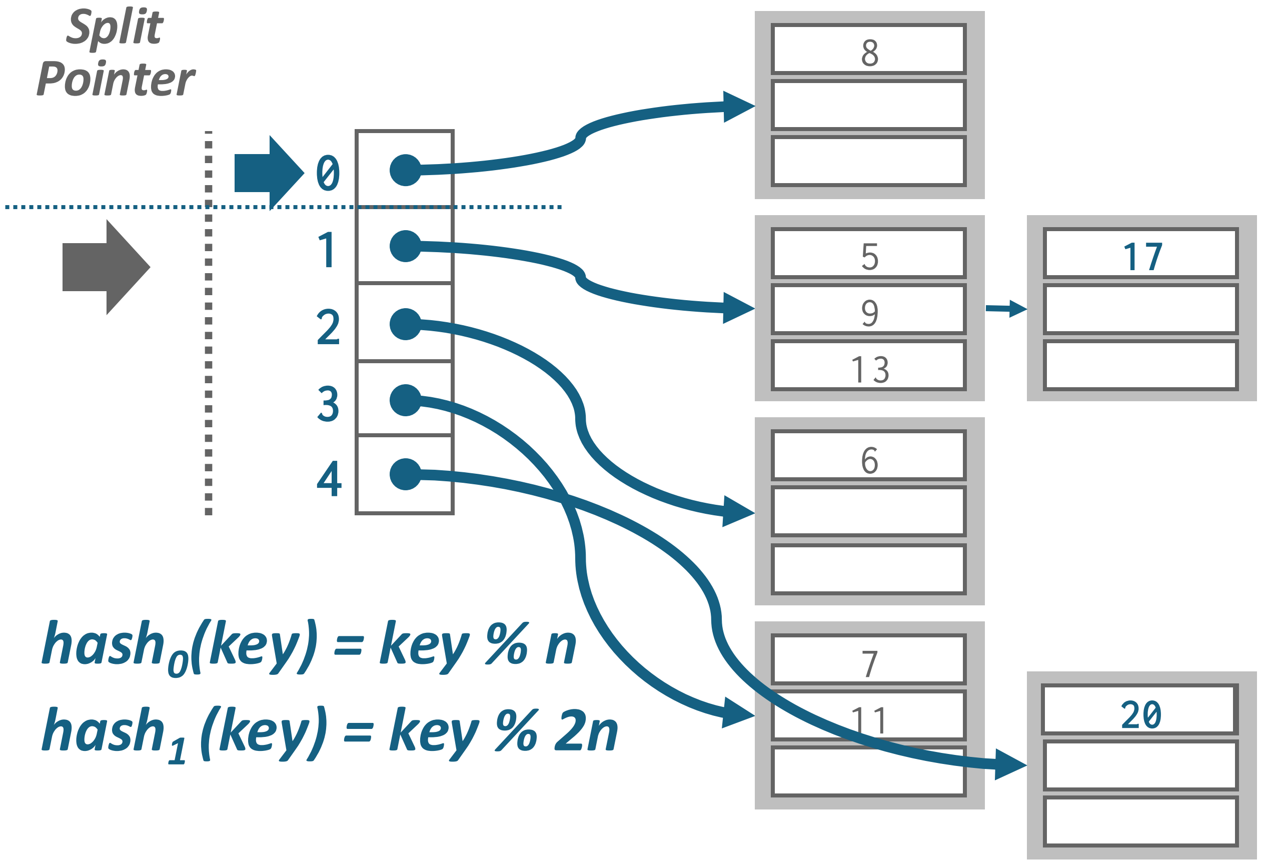
$hash_1(8)$ = 8 % 8 = 0

$hash_1(20)$ = 20 % 8 = 4

Get 20

$hash_0(20)$ = 20 % 4 = 0

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

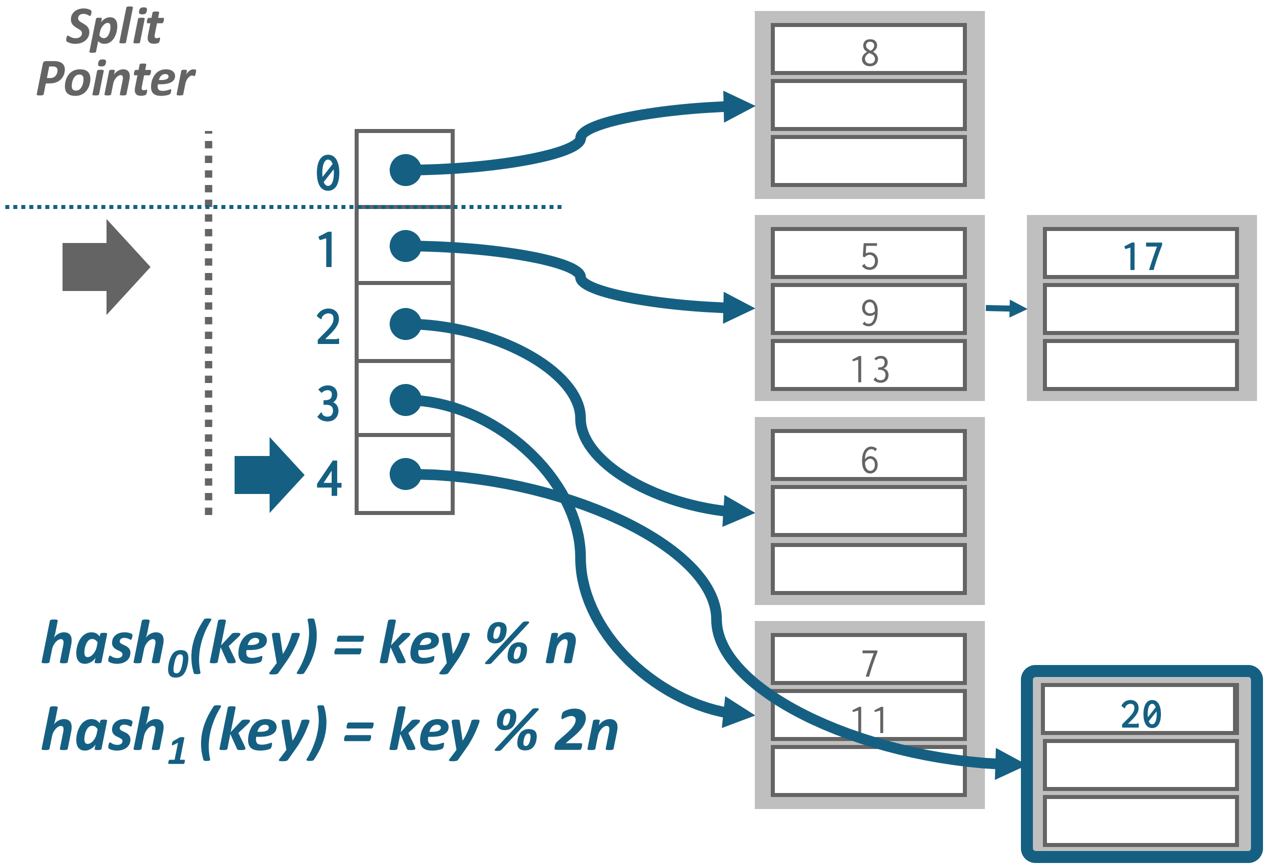
$hash_1(8) = 8 \% 8 = 0$

$hash_1(20) = 20 \% 8 = 4$

Get 20

$hash_0(20) = 20 \% 4 = 0$

Linear Hashing



$hash_0$(key) = key % n

$hash_1$ (key) = key % 2n

Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

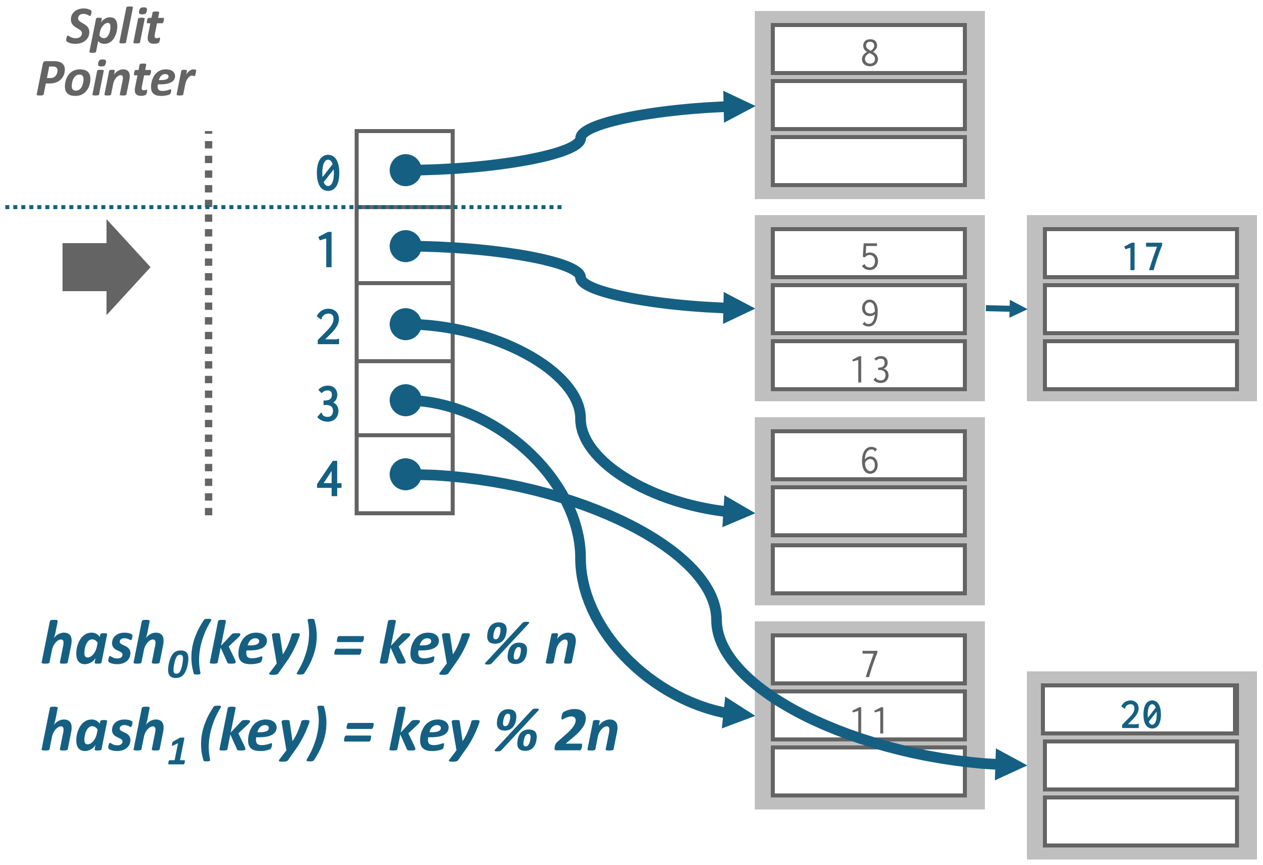
$hash_1(20) = 20 \% 8 = 4$

Get 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

$hash_1(20) = 20 \% 8 = 4$

Get 20

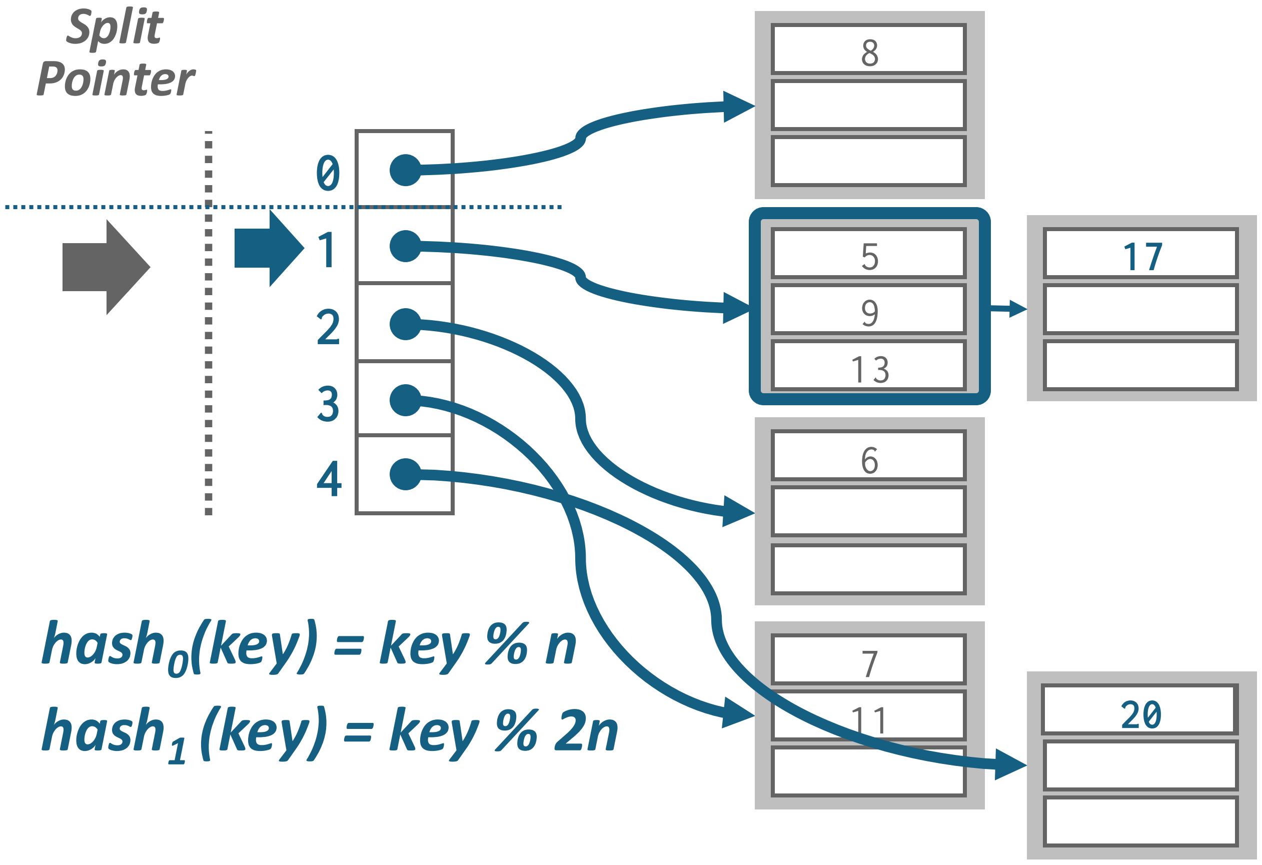
$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Get 9

$hash_0(9) = 9 \% 4 = 1$

Linear Hashing



$hash_0(key) = key \% n$

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

Get 6

$hash_0(6) = 6 \% 4 = 2$

Put 17

$hash_0(17) = 17 \% 4 = 1$

$hash_1(8) = 8 \% 8 = 0$

$hash_1(20) = 20 \% 8 = 4$

Get 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Get 9

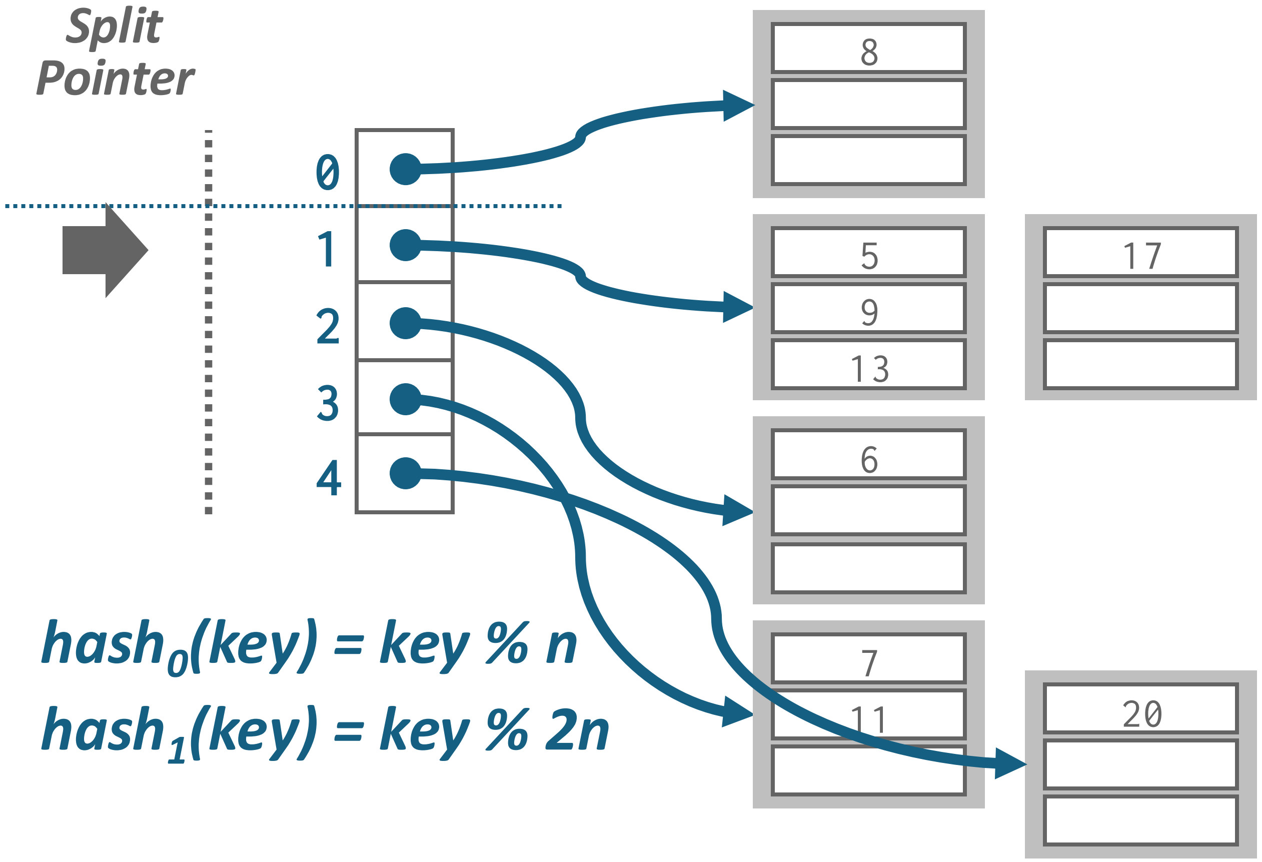
$hash_0(9) = 9 \% 4 = 1$

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.

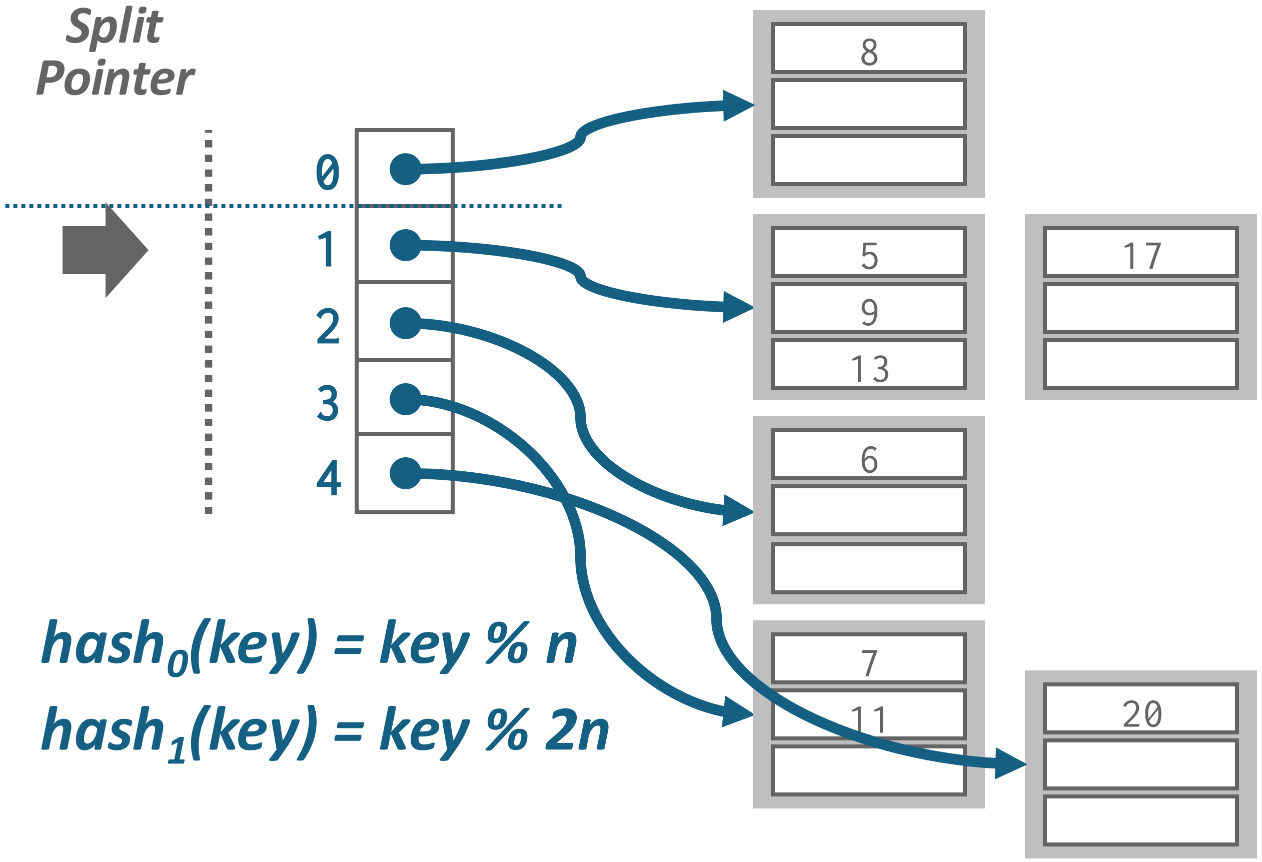
Linear Hashing - Deletes



$hash_0(key) = key \% n$

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

Linear Hashing - Deletes



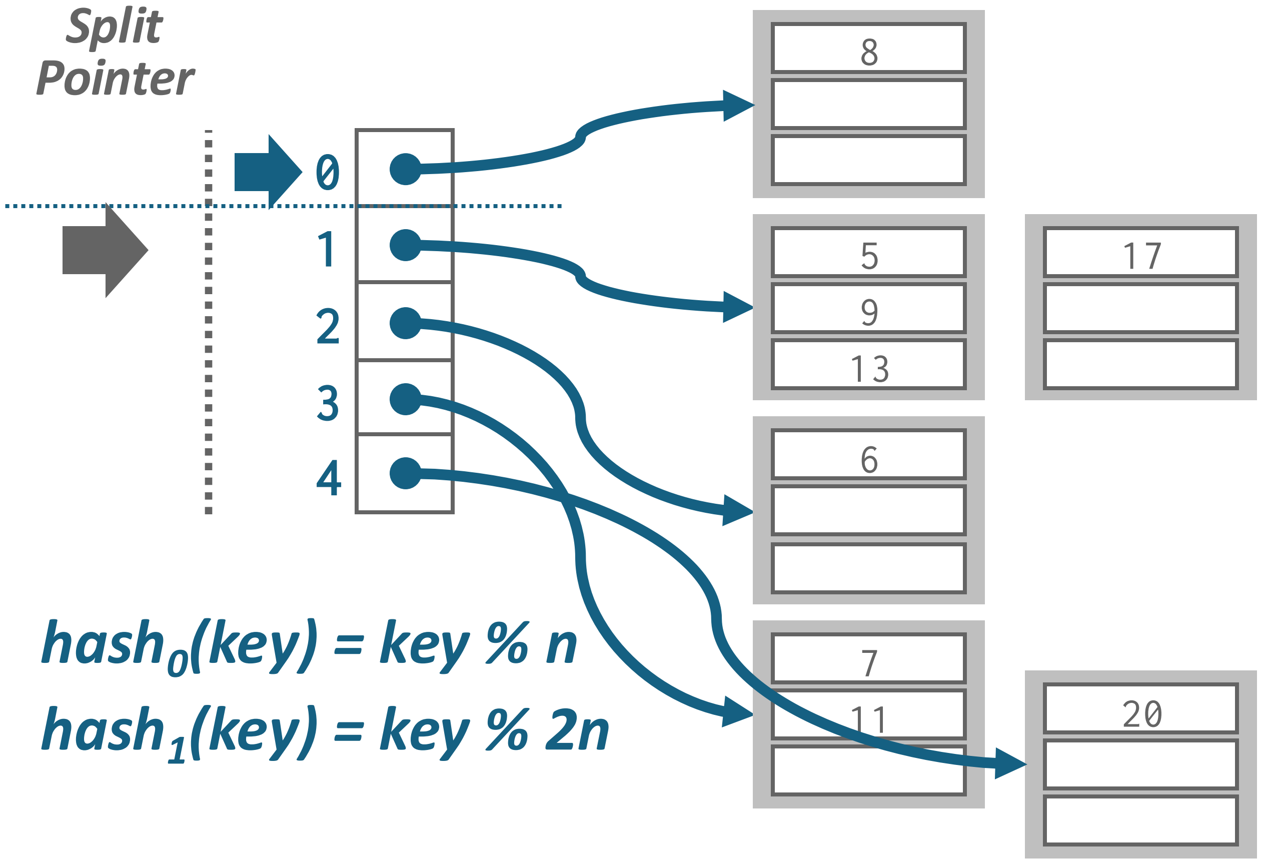
hash$_0$(key) = key % n

hash$_1$(key) = key % 2n

Delete 20

hash$_0$(20) = 20 % 4 = 0

Linear Hashing - Deletes



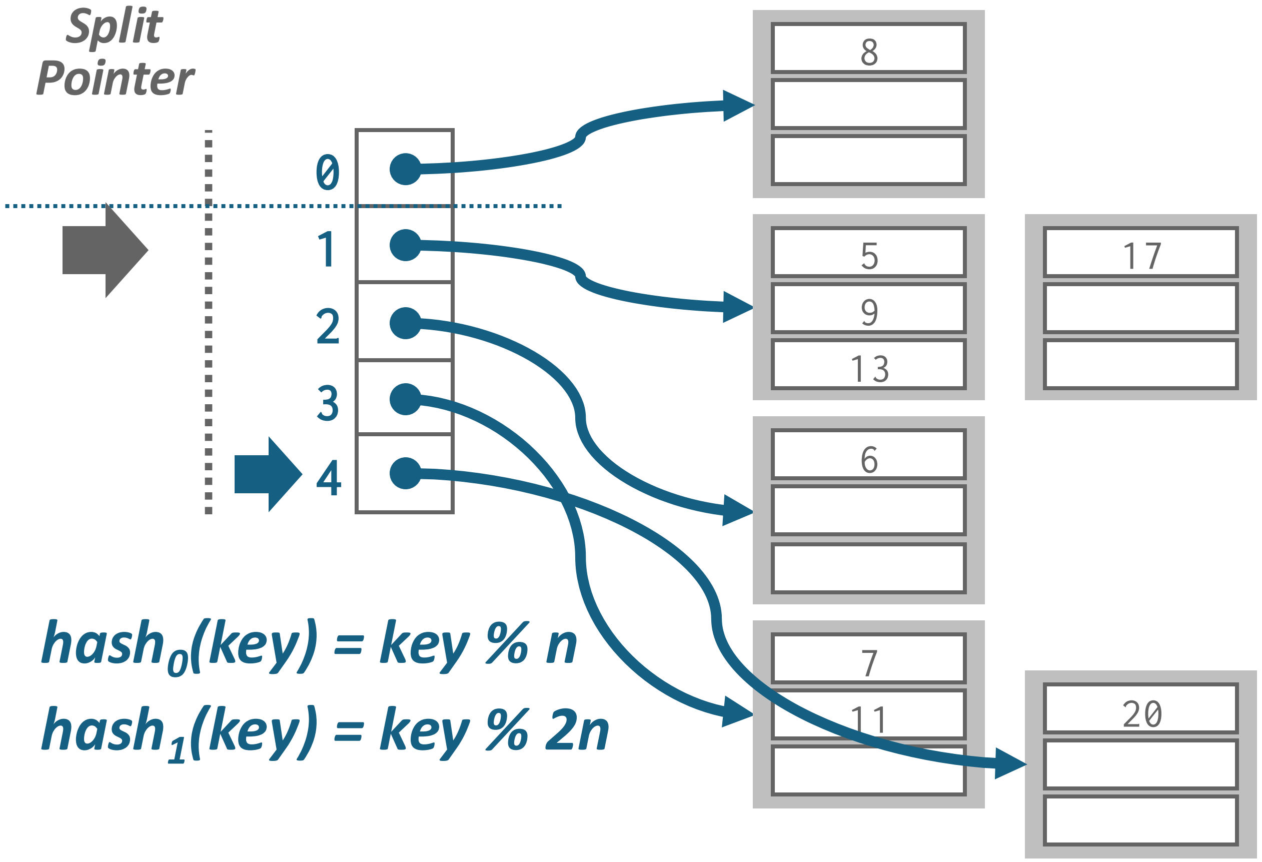
hash$_0$*(key) = key % n*

hash$_1$*(key) = key % 2n*

Delete 20

hash$_0$*(20)* = 20 % 4 = 0

Linear Hashing - Deletes



$hash_0(key) = key \% n$

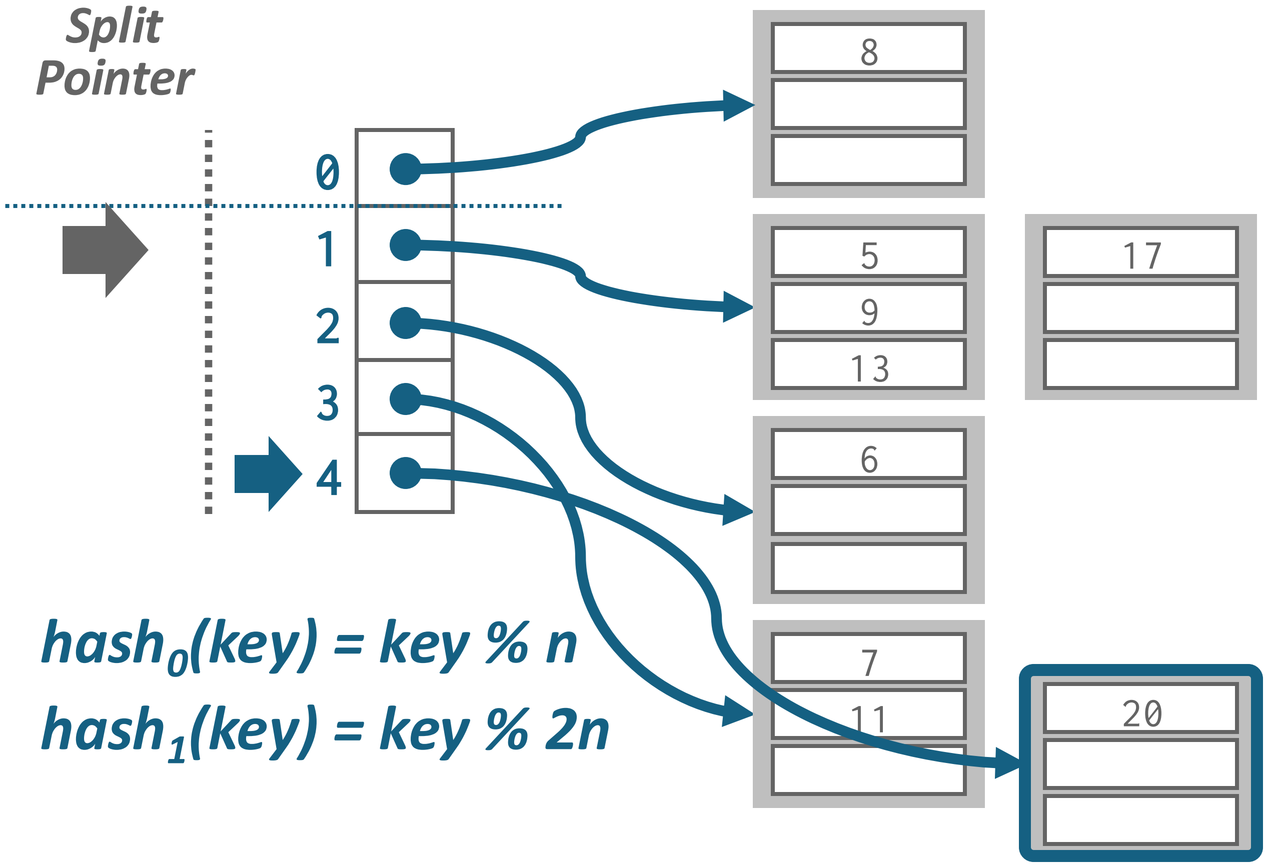
$hash_1(key) = key \% 2n$

Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing - Deletes



$hash_0$(key) = key % n

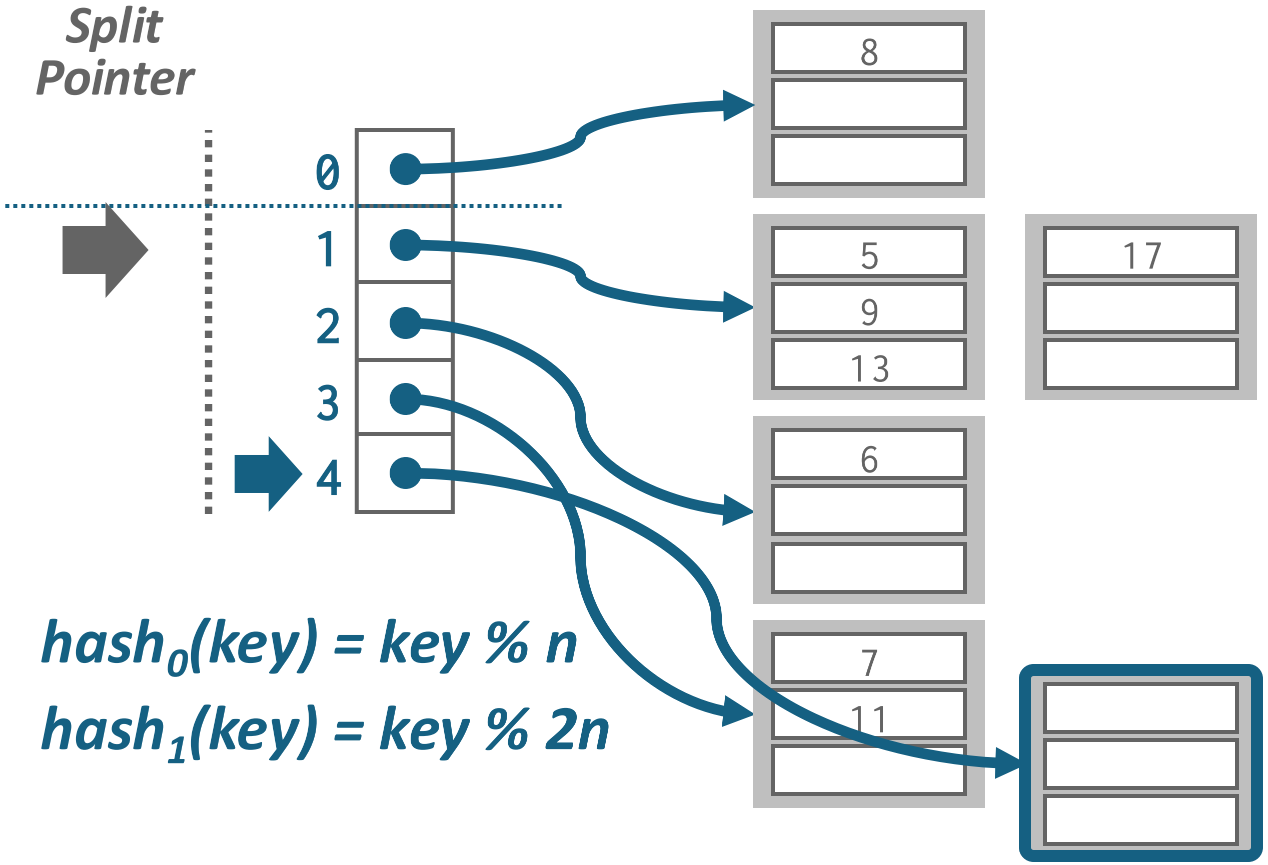
$hash_1$(key) = key % 2n

Delete 20

$hash_0$(20) = 20 % 4 = 0

$hash_1$(20) = 20 % 8 = 4

Linear Hashing - Deletes



$hash_0(key) = key \% n$

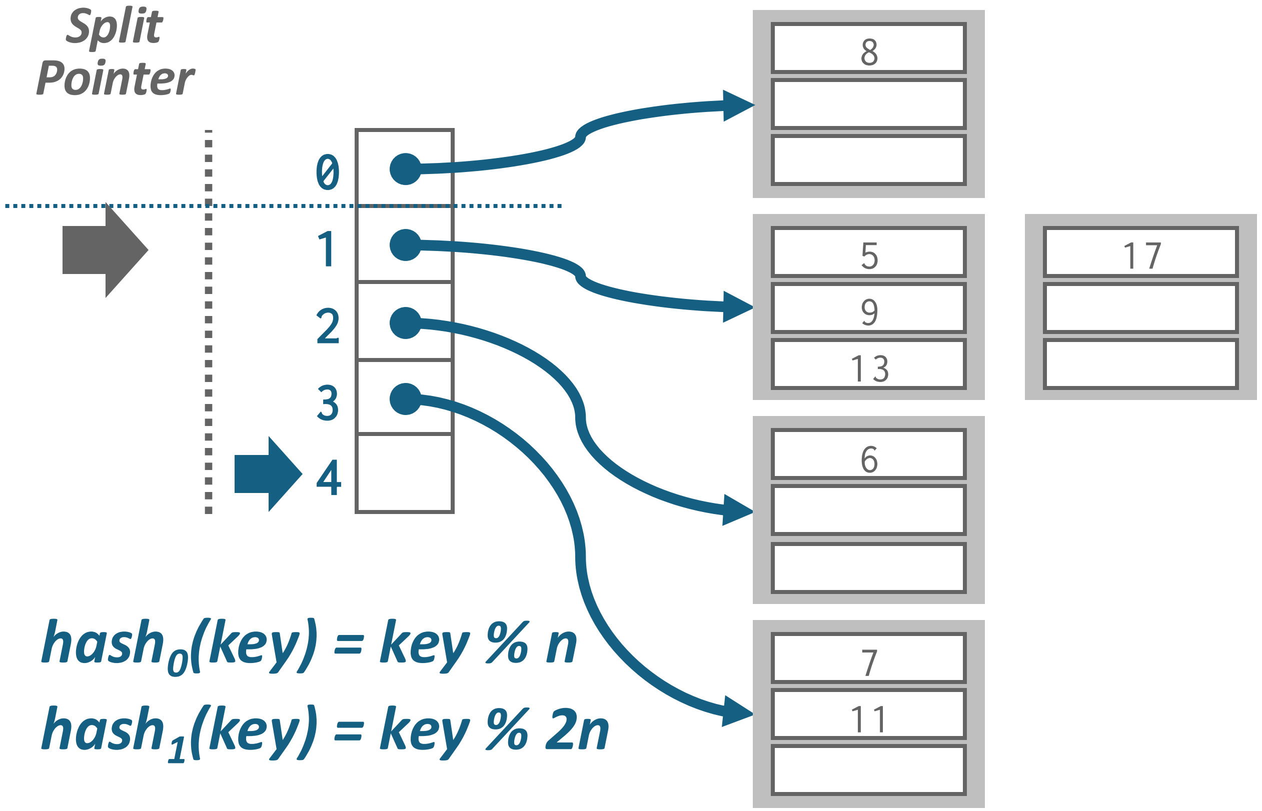
$hash_1(key) = key \% 2n$

Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing - Deletes



hash$_0$*(key) = key % n*

hash$_1$*(key) = key % 2n*

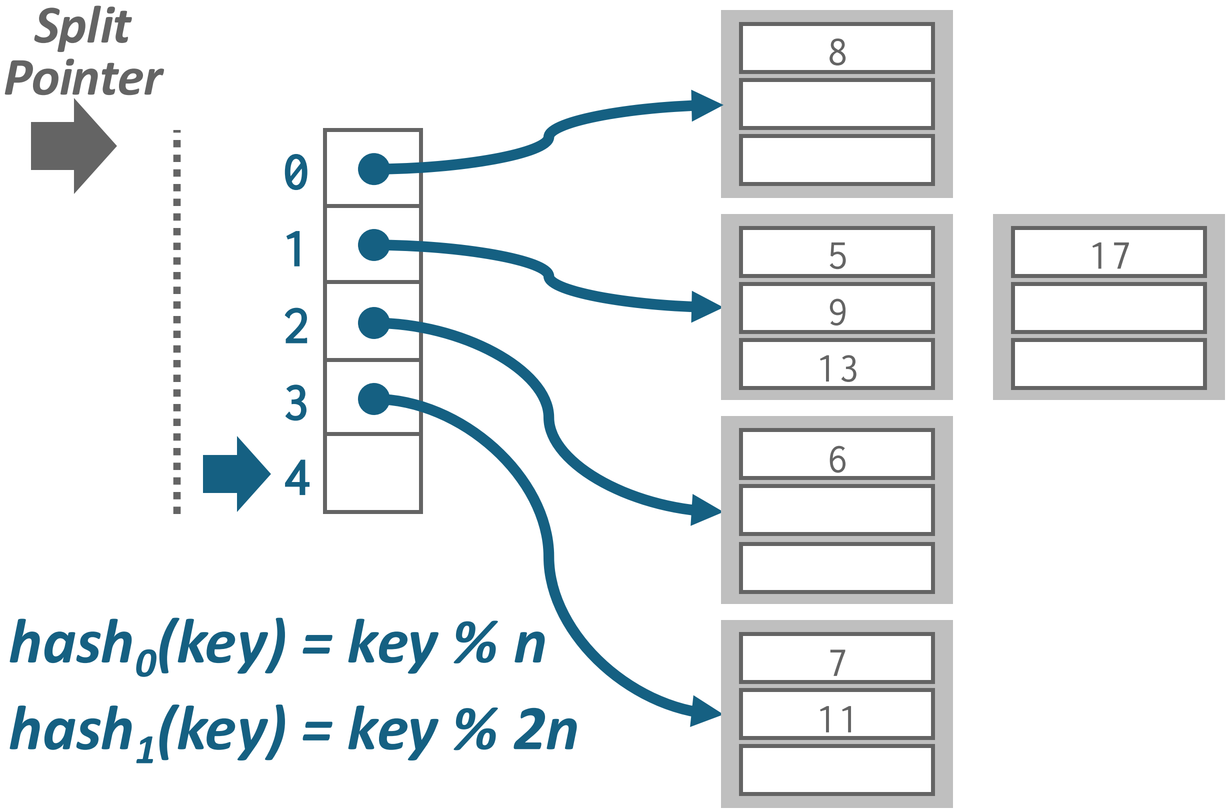
Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

SCHOOL OF DATA SCIENCE

Linear Hashing - Deletes



$hash_0$(key) = key % n

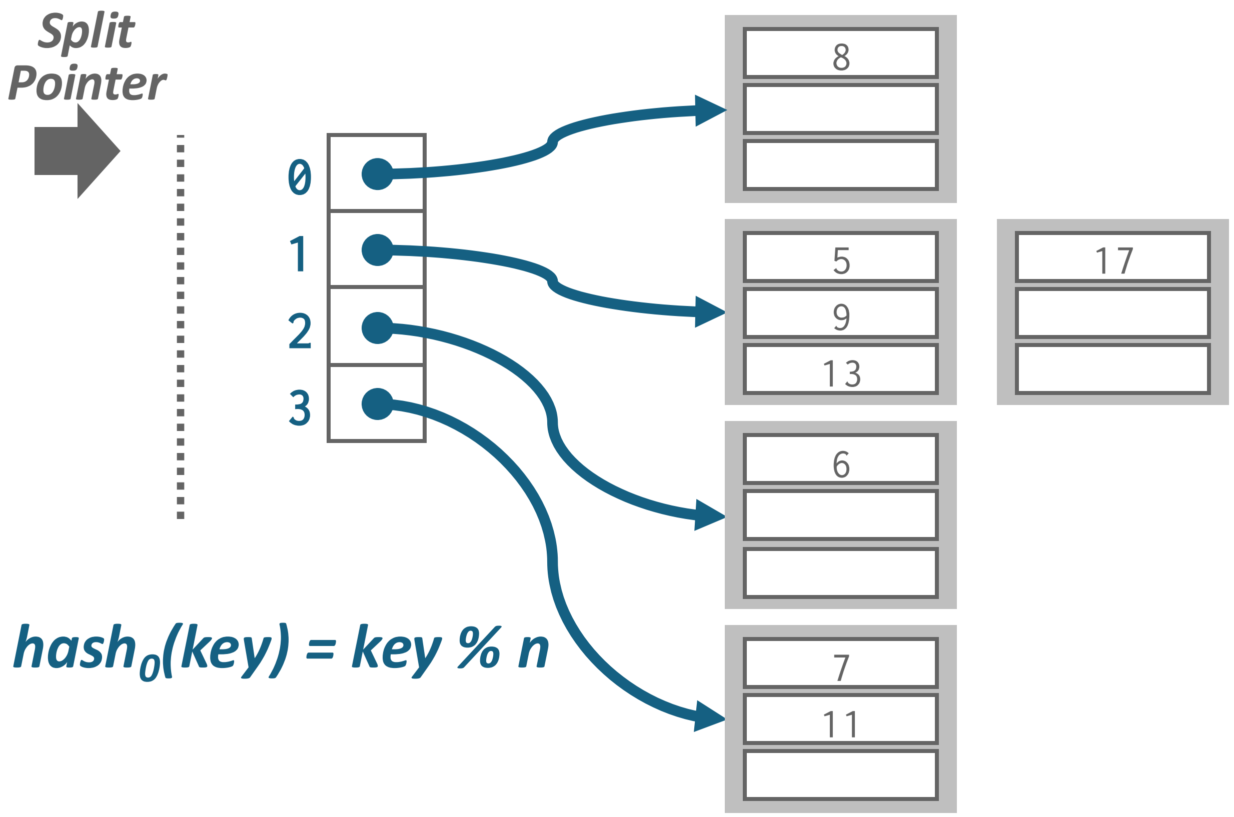
$hash_1$(key) = key % 2n

Delete 20

$hash_0$(20) = 20 % 4 = 0

$hash_1$(20) = 20 % 8 = 4

Linear Hashing - Deletes



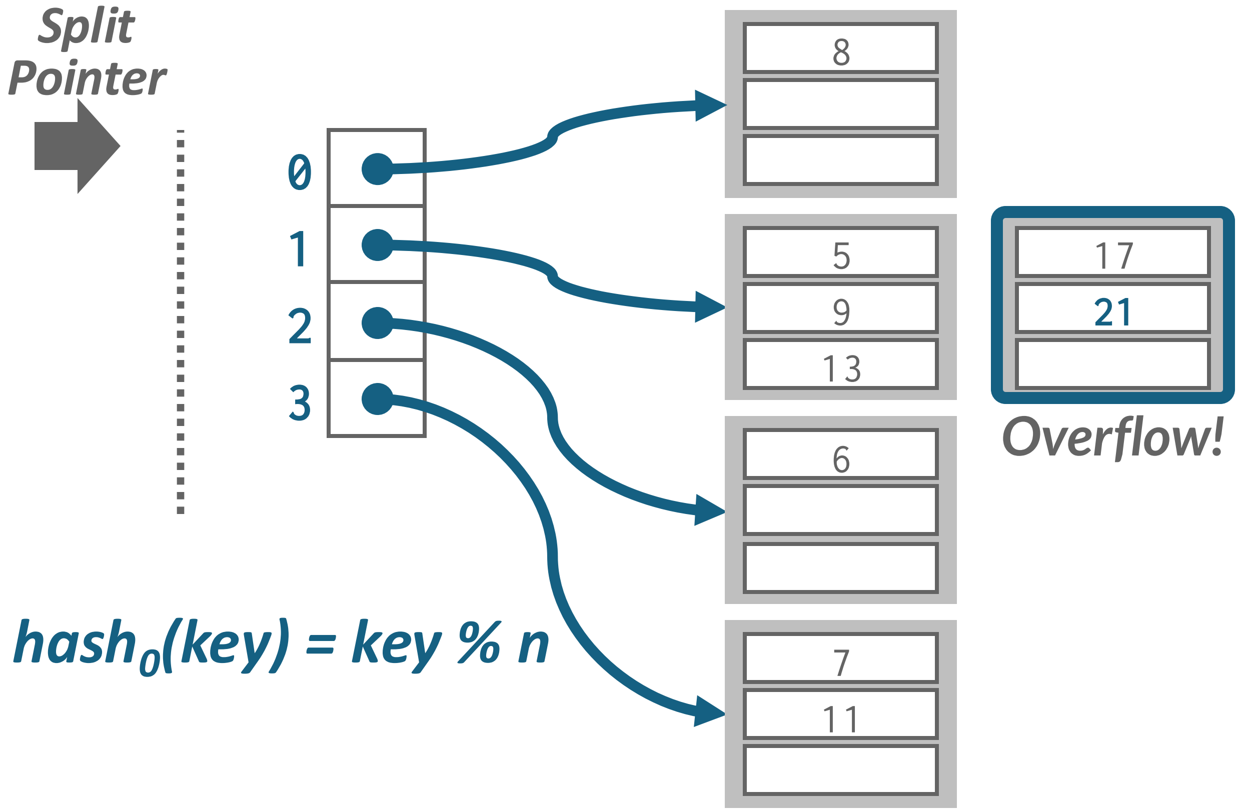
$hash_0(key) = key \% n$

Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Linear Hashing - Deletes



$hash_0(key) = key \% n$

Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$

Put 21

$hash_0(21) = 21 \% 4 = 1$

Conclusion

- Fast data structures that support $O(1)$ look-ups that are used all throughout DBMS internals.
  - Trade-off between speed and flexibility.

- Hashing schemes get used for both in-memory and on-disk.
  - Linear Probing, Cuckoo Hashing, Chained Hashing: Generally used for in-memory hash tables; e.g., in hash join and aggregate operators.
  - Extendible Hashing, Linear Hashing: for disk-based hash indexing.

Next Lecture

- **B+Trees**
  - aka “The Greatest Data Structure of All Time”