THE UNIVERSITY *of* EDINBURGH

Advanced Databases

Spring 2020

Lecture #06:

Hash-Based Indexing

Milos Nikolic

Hash-Based Indexing

Suitable for **equality selections**

```
SELECT * FROM Customer WHERE A = constant
```

Cannot support range search

Other query operations internally generate a flood of equality tests

- E.g.: nested loop join, where hash index can make a real difference

Support in commercial DBMSs

- Tree-structured indexes preferred since they cover the more general range predicates
- But hash-based indexes are used for (index) nested loop joins

Overview

Static and dynamic hashing techniques exist

- Trade-offs similar to ISAM vs. B+ trees

Static hashing schemes

- Chained hashing
- Linear probing
- Robin Hood hashing
- Cuckoo hashing

Dynamic hashing schemes

- Extendible hashing
- Linear hashing

Static Chained Hashing

Hash index is a collection of **buckets**

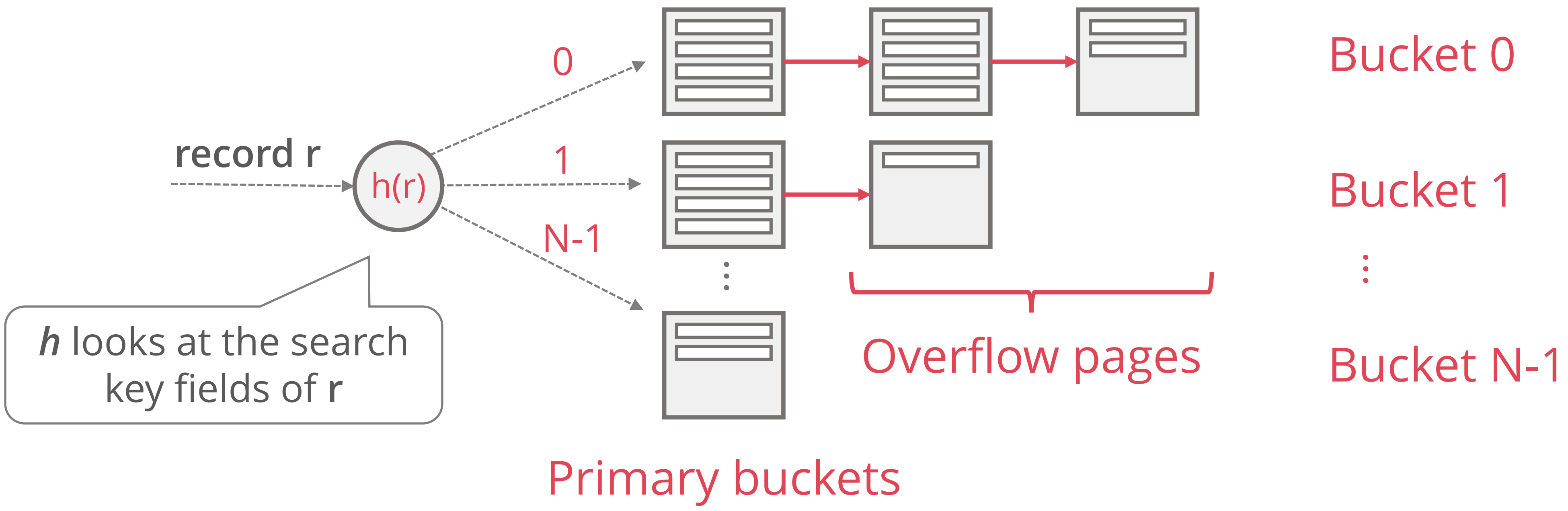
Build static hash index on column A

- Allocate a fixed area of N (successive) pages, the so-called **primary buckets**
- In each bucket, install a pointer to a chain of **overflow** pages (initially set to **null**)
- Define a **hash function *h*** with range [0, ..., N-1]
- The domain of ***h*** is the type of A
  - e.g., h : INTEGER ⟶ [0, ..., N-1], if A is of type INTEGER
- The hash function determines the bucket where the desired value can be found

STATIC HASH TABLE

Bucket = **primary page** plus zero or more **overflow pages**

Buckets contain index data entries **k*** implemented using any of the variants A, B, C



Static Hash Table Management

Operations: **search**, **insert**, **delete**

- Compute ***h***(***r***) on the search key fields of record ***r***
- Access the primary bucket page with ***h***(***r***)
- Search for/insert/delete record on this page and, if needed, overflow pages

Long overflow chains can degrade performance

- Operation costs become non-uniform and unpredictable
- To reduce this problem, ***h*** needs to scatter search keys evenly across [0, ..., N-1]
- Large # of entries can still cause long chains (dynamic hashing to fix this)

Linear Probe Hashing

Single giant table of slots

Resolve collisions by linearly searching for the next free slot

- To find a record, 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 generalisations of lookups

Good locality of reference

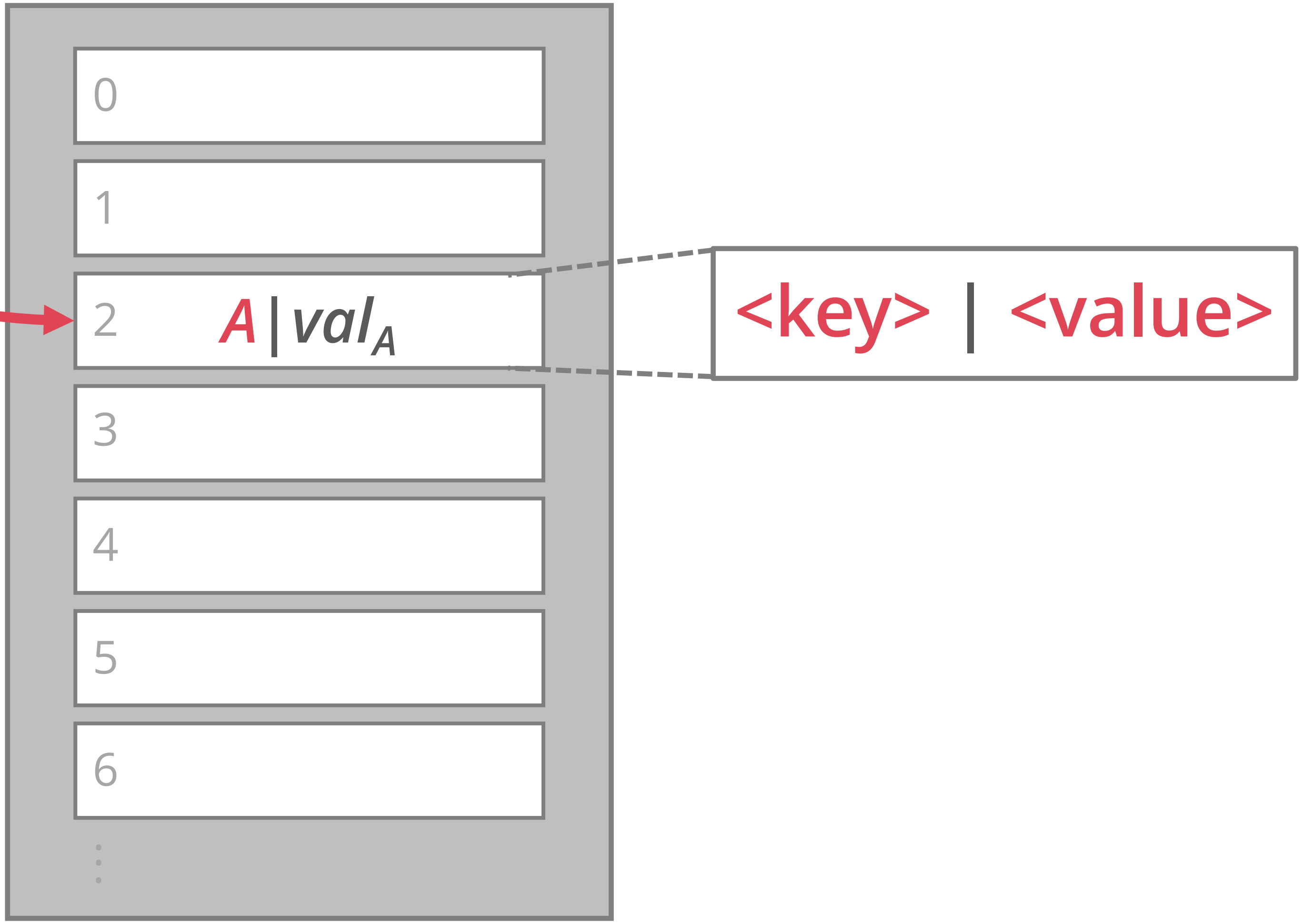
Search/insert/delete in O(1) expected time as long as the load factor < 1

\# keys / capacity

LINEAR PROBE HASHING

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

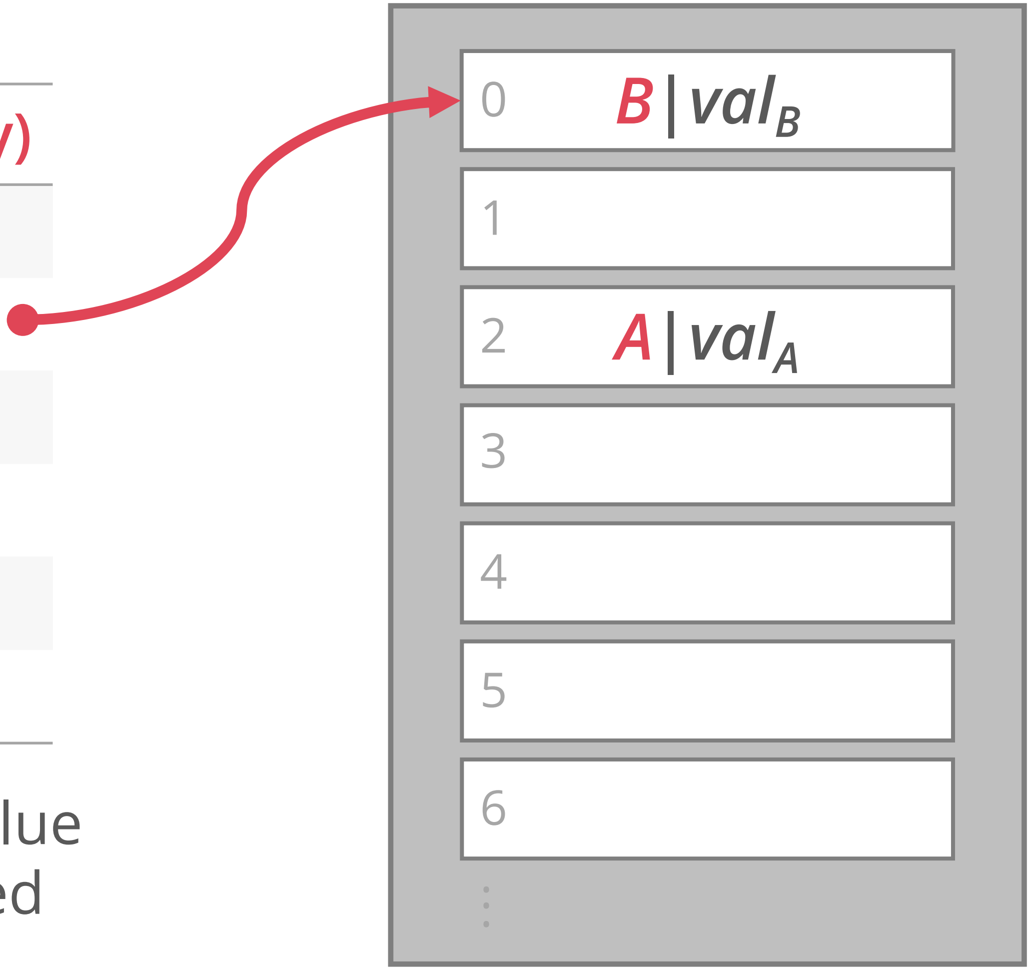
Sequence of key-value pairs to be inserted



Linear Probe Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

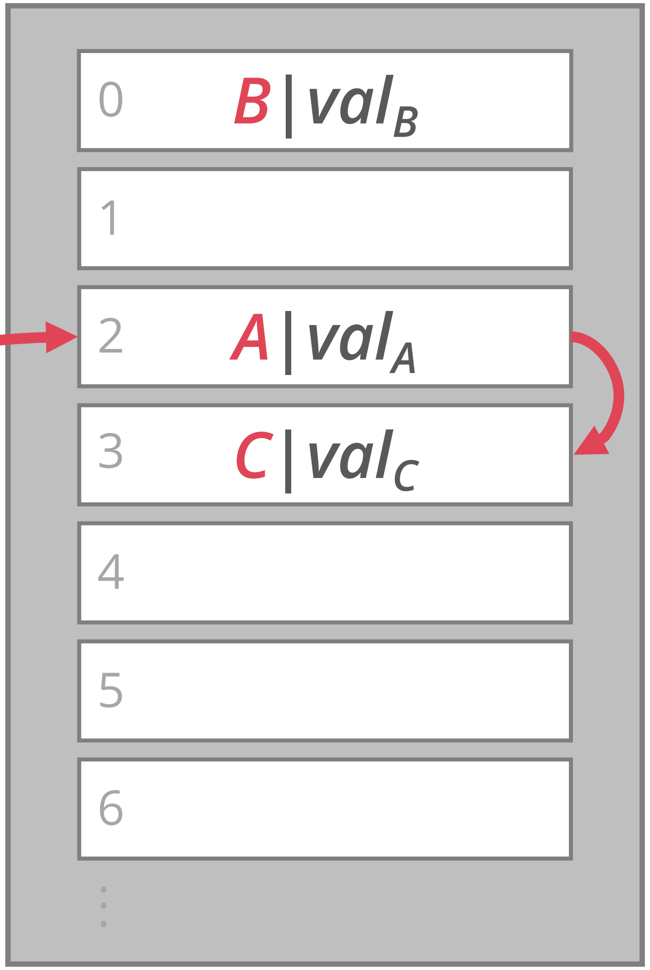
Sequence of key-value pairs to be inserted



Linear Probe Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

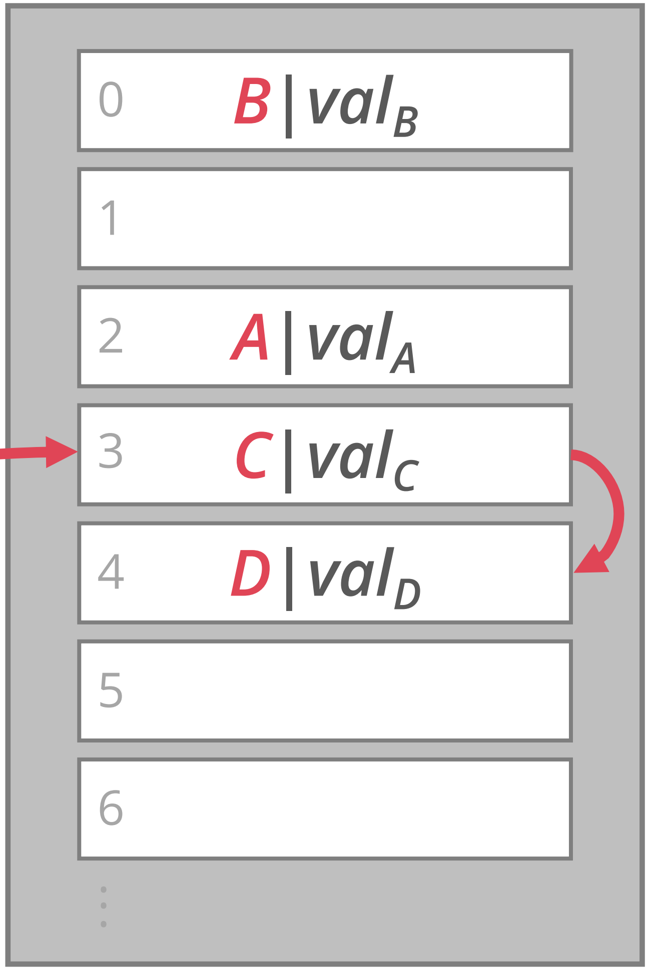
Sequence of key-value pairs to be inserted



LINEAR PROBE HASHING

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

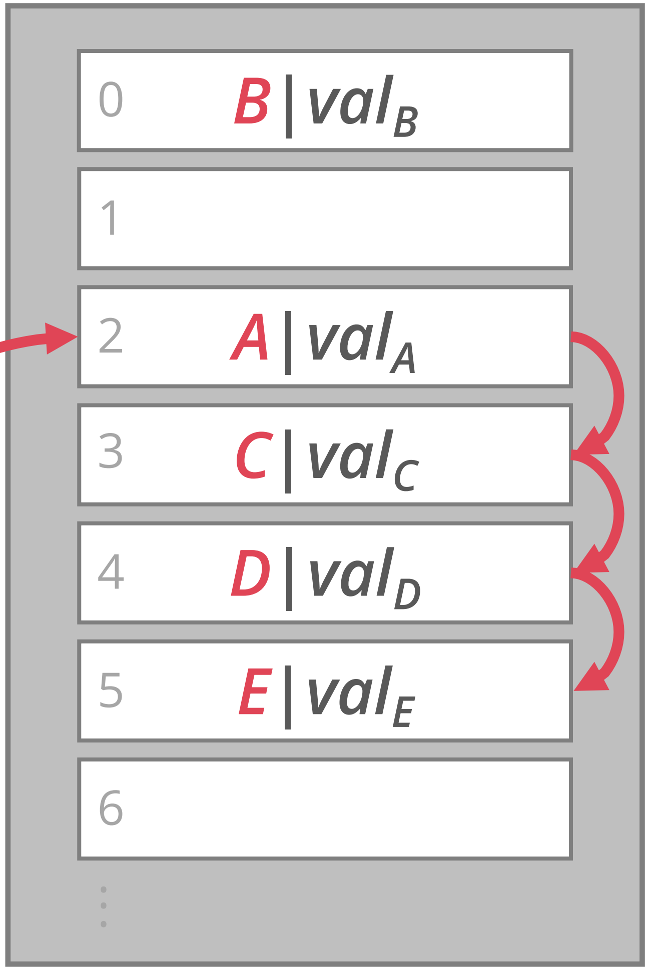
Sequence of key-value pairs to be inserted



Linear Probe Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

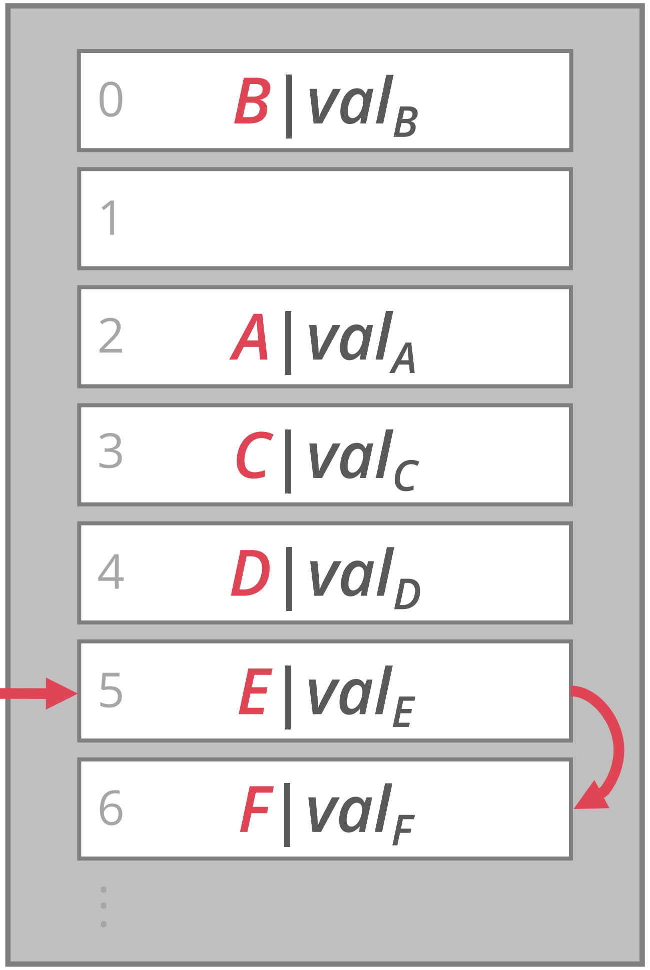
Sequence of key-value pairs to be inserted



Linear Probe Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

Sequence of key-value pairs to be inserted



Linear Probe Hashing

Search for X

First look at h(X)

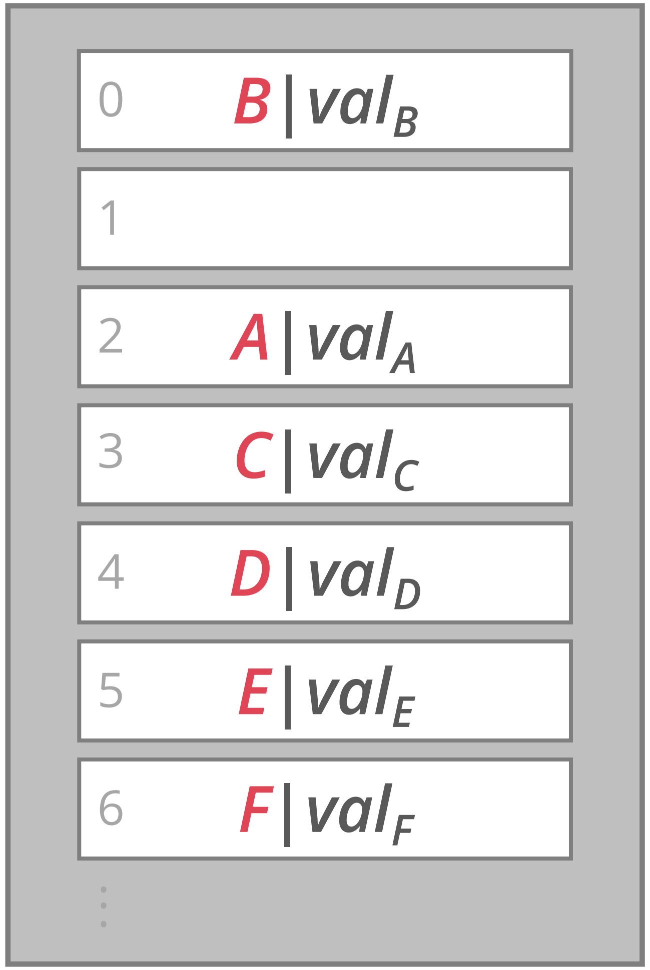
If not there, look at next slot until empty

Delete X

Find X at slot *i*, empty slot *i*

Search forward until finding

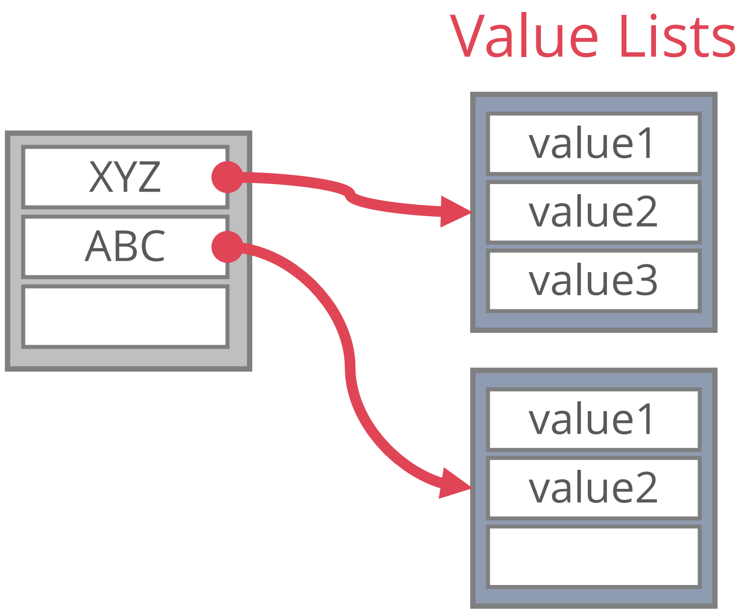
1) an empty slot or
2) a key X at slot *j* movable to *i* (i.e., h(X) ≤ *i*); move the key to *i*, repeat this for slot *j*



Non-Unique Keys

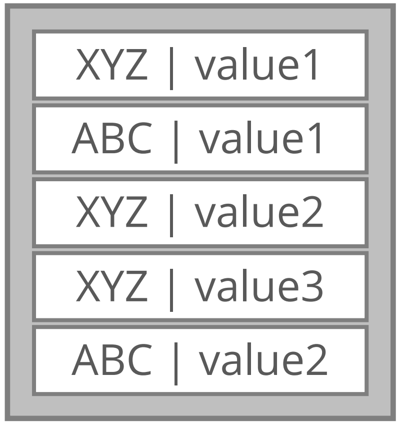
Choice #1: Separate Linked List

Store values in separate storage area for each key



Choice #2: Redundant Keys

Store duplicate key entries together in the table



Observation

To reduce the # of wasteful comparisons, it is important to avoid collisions of hashed keys

This requires a hash table with ~2x the number of slots as the number of expected elements

Robin Hood Hashing

Variant of linear probe hashing

Steals slots from **"rich" keys** and give them to **"poor" keys**

- Each key tracks the number of positions they are from their 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

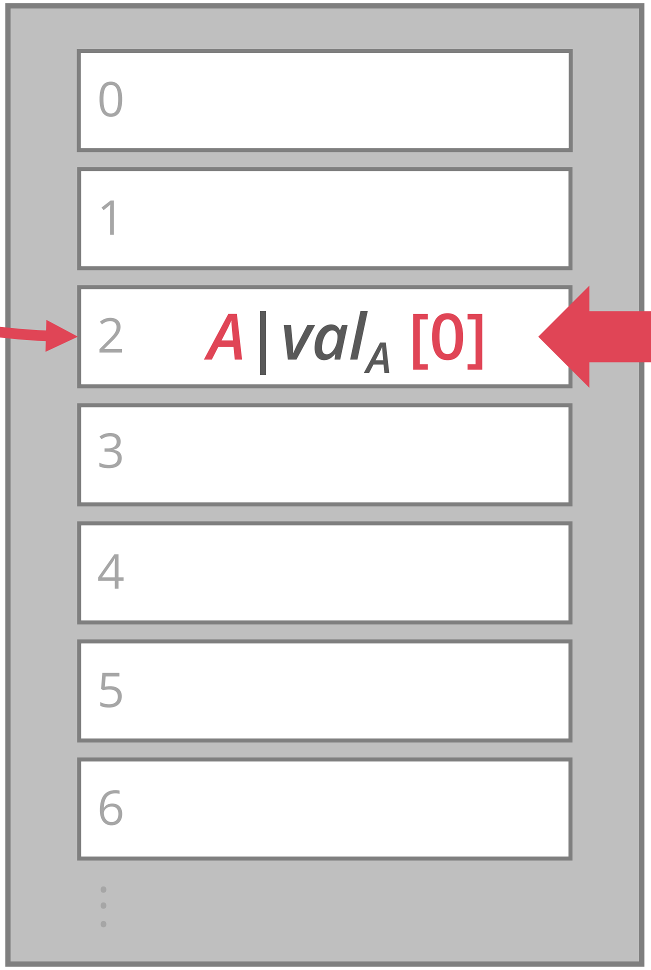
Reduces the variance of the expected # of probes

"Robin Hood Hashing should be your default Hash Table implementation", Sebastian Sylvan

Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

Sequence of key-value pairs to be inserted

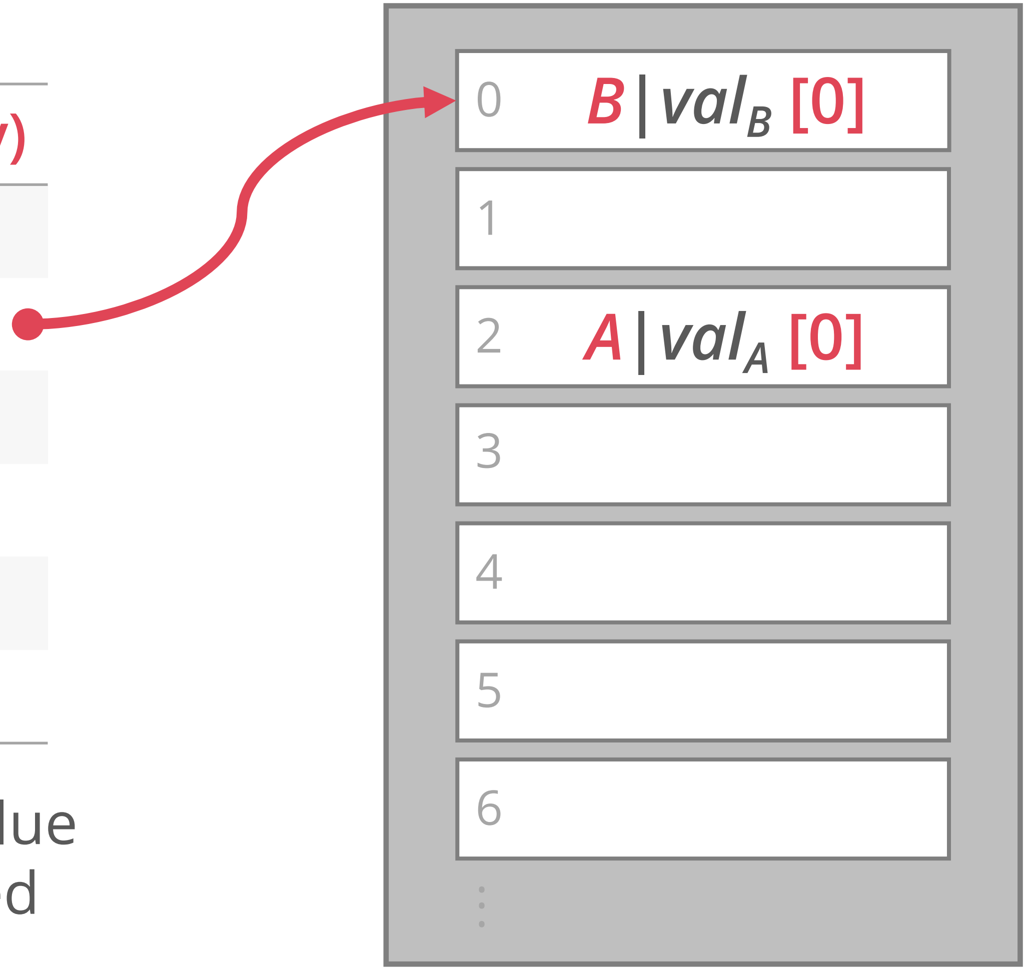


# of slots away from its ideal position

Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

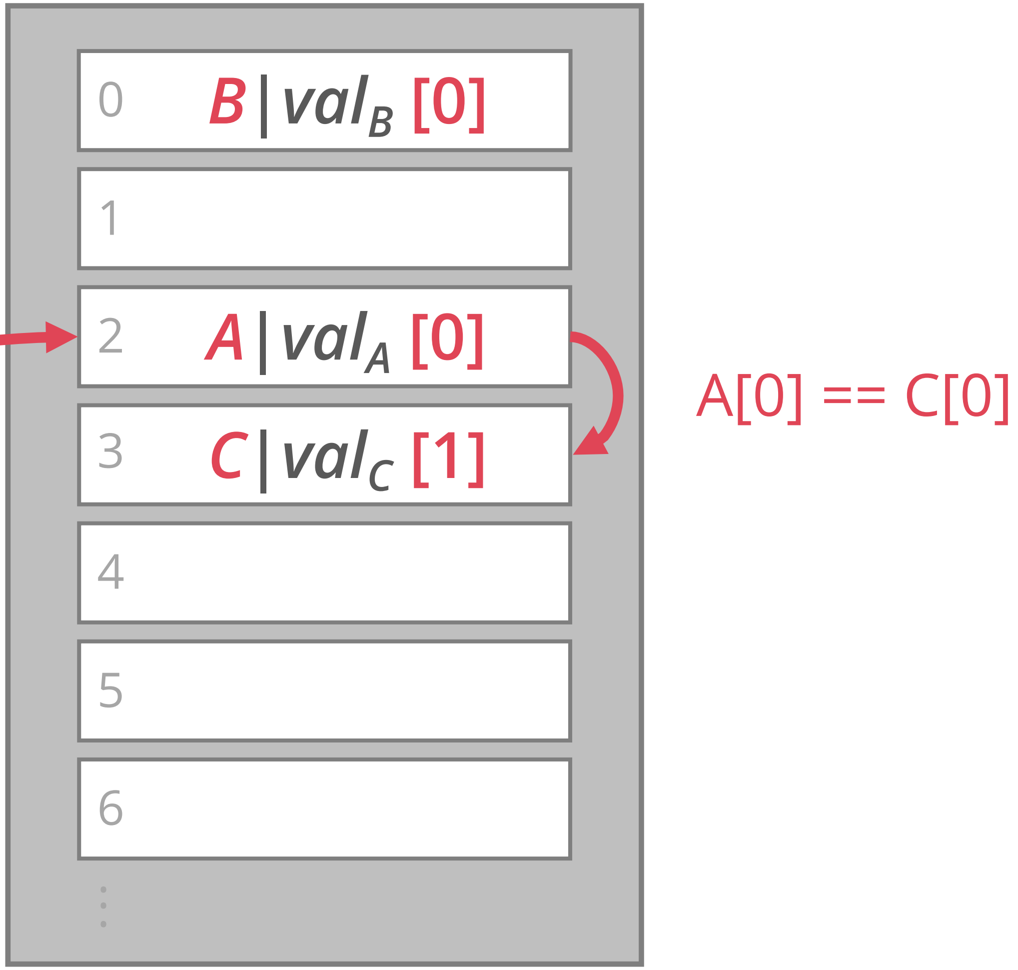
Sequence of key-value pairs to be inserted



Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

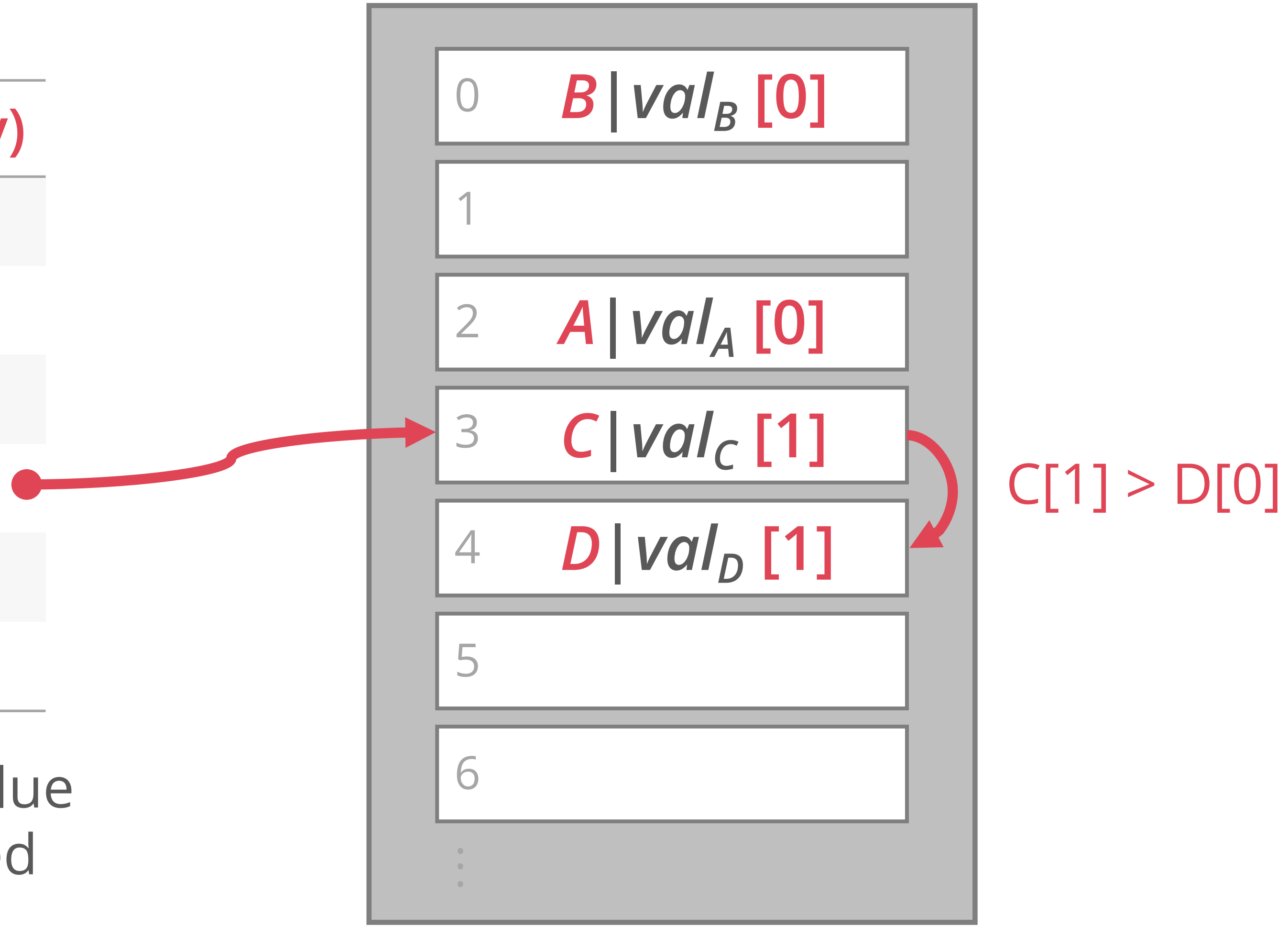
Sequence of key-value pairs to be inserted



Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

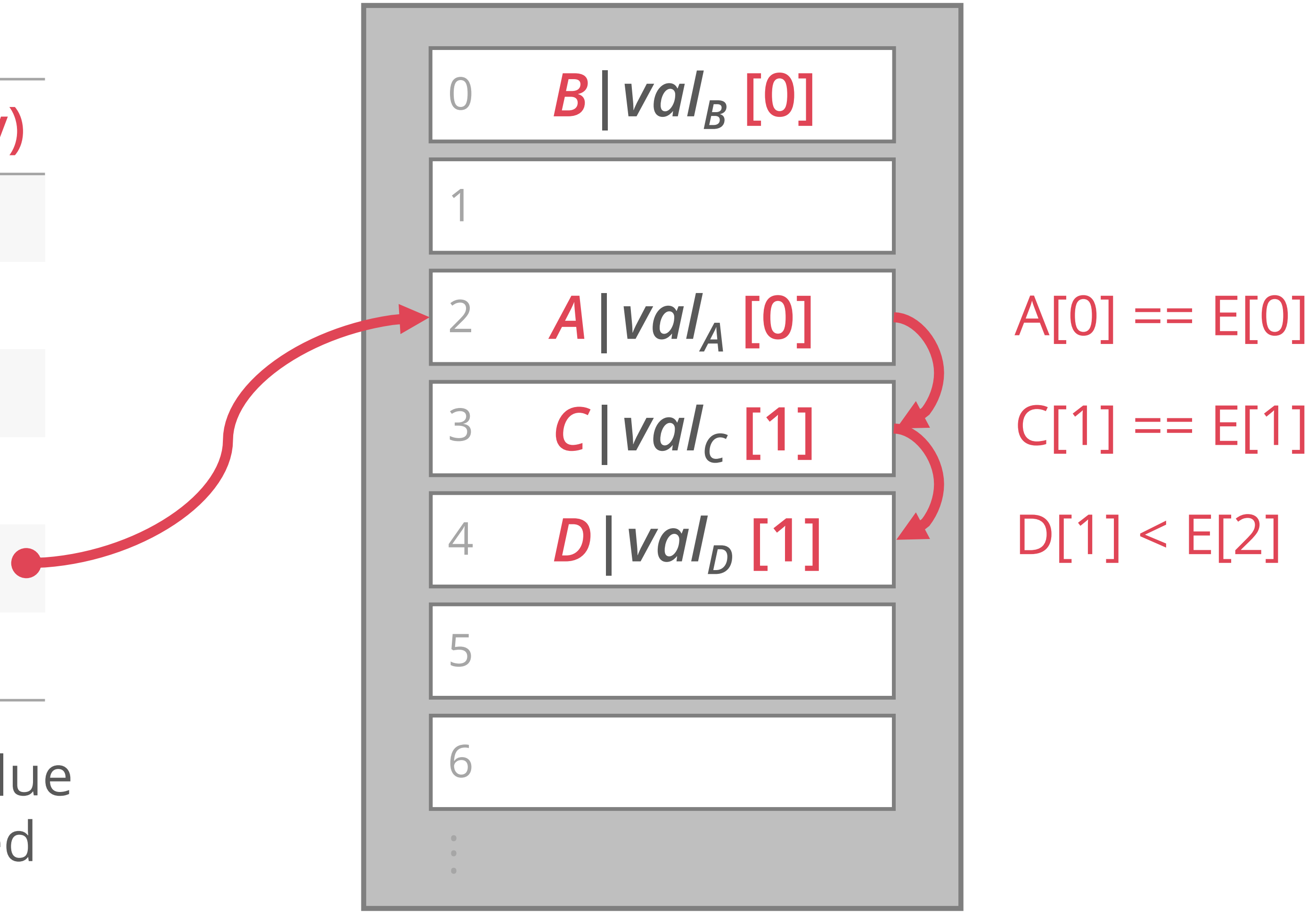
Sequence of key-value pairs to be inserted



Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

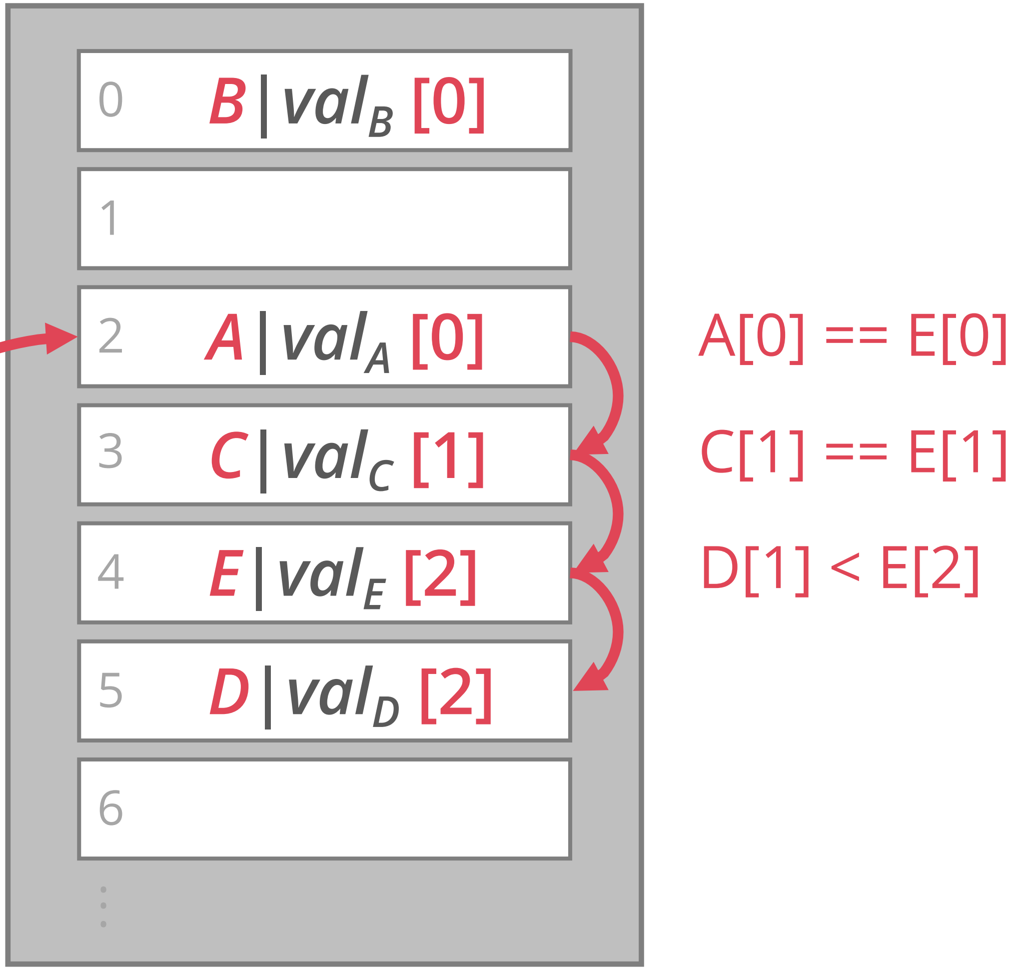
Sequence of key-value pairs to be inserted



Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

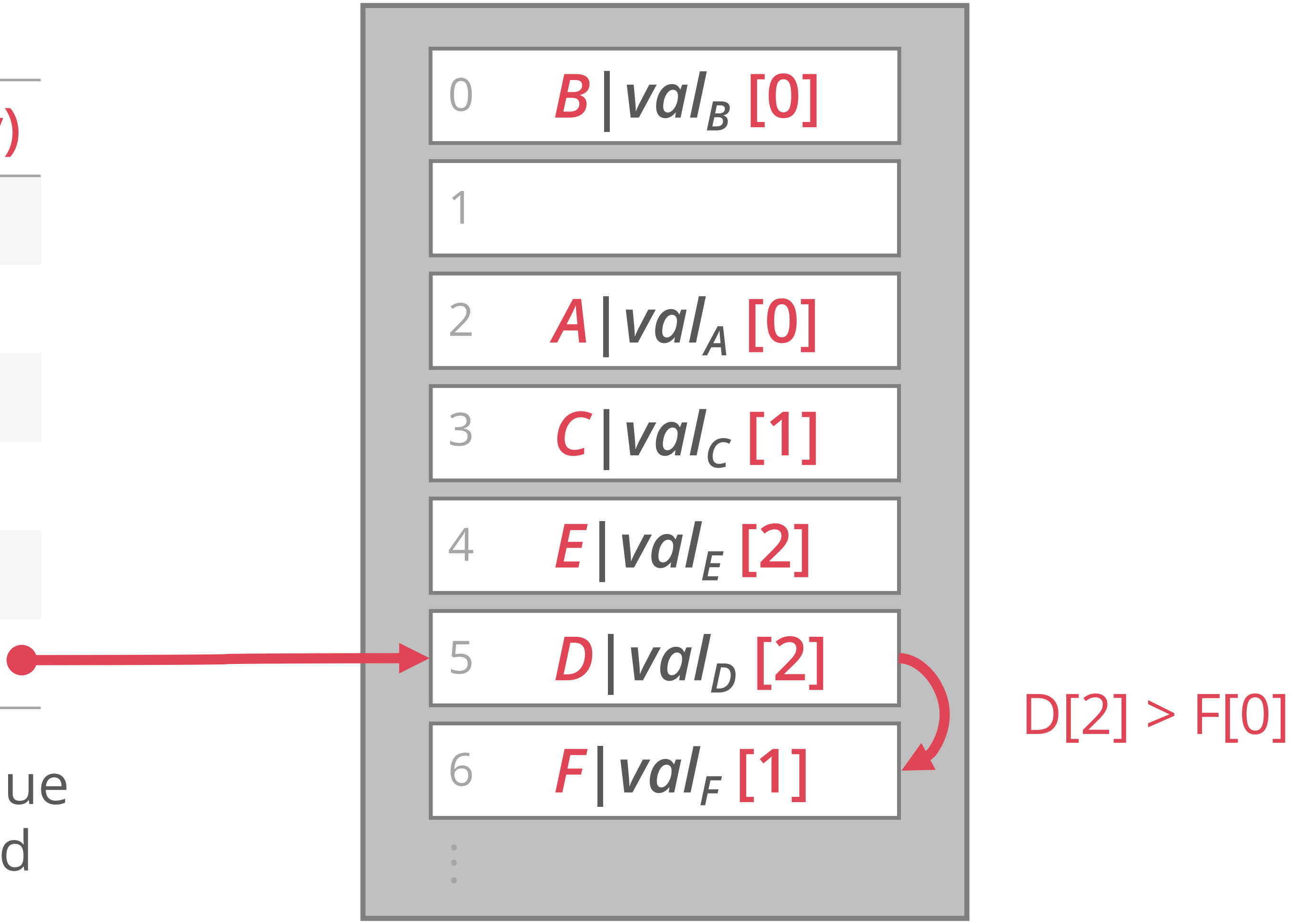
Sequence of key-value pairs to be inserted



Robin Hood Hashing

key	value	h(key)
A	val_A	2
B	val_B	0
C	val_C	2
D	val_C	3
E	val_E	2
F	val_F	5

Sequence of key-value pairs to be inserted



Cuckoo Hashing

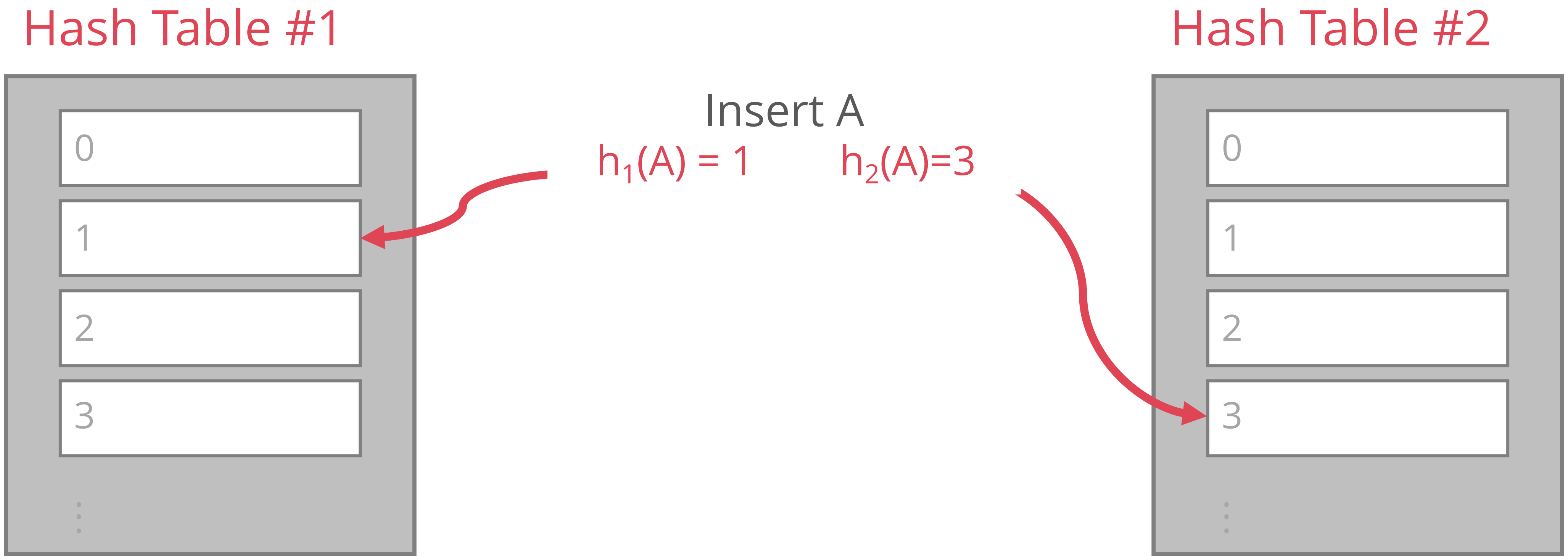
Use **multiple hash tables** with **different hash functions**

- 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 to find a new location

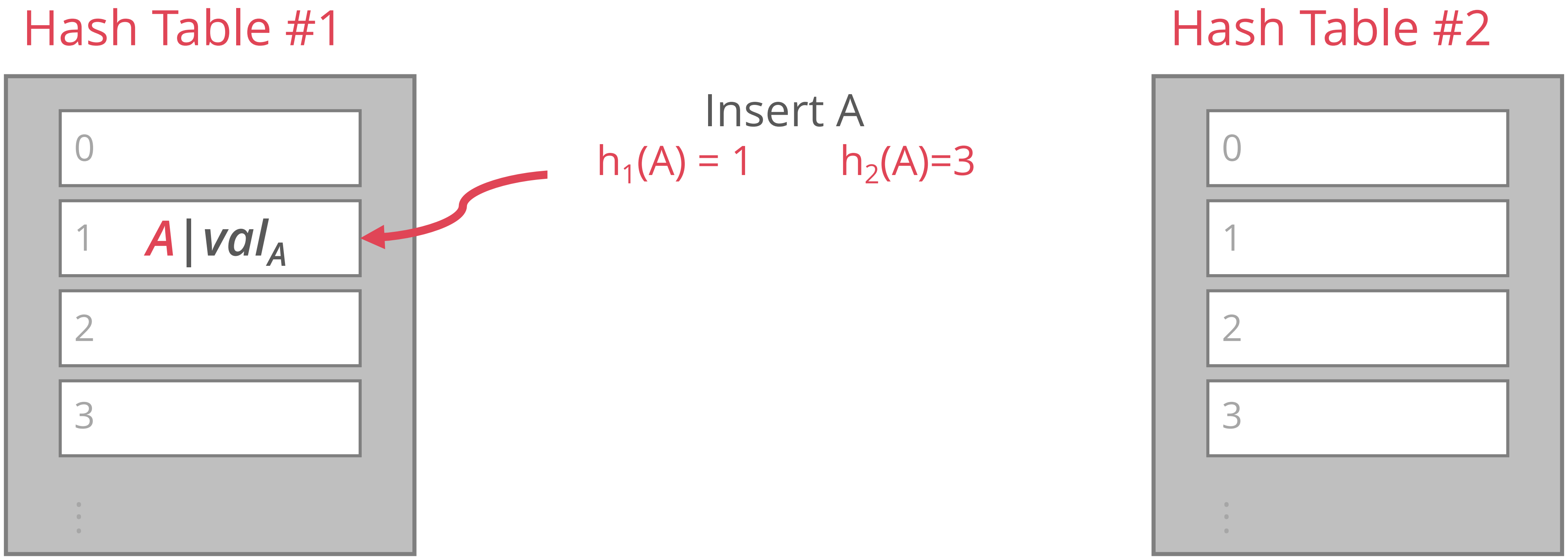
Lookups and deletions are always O(1) because only one location per hash table is checked

Insertions in expected O(1) time as long as the load factor < 1/2

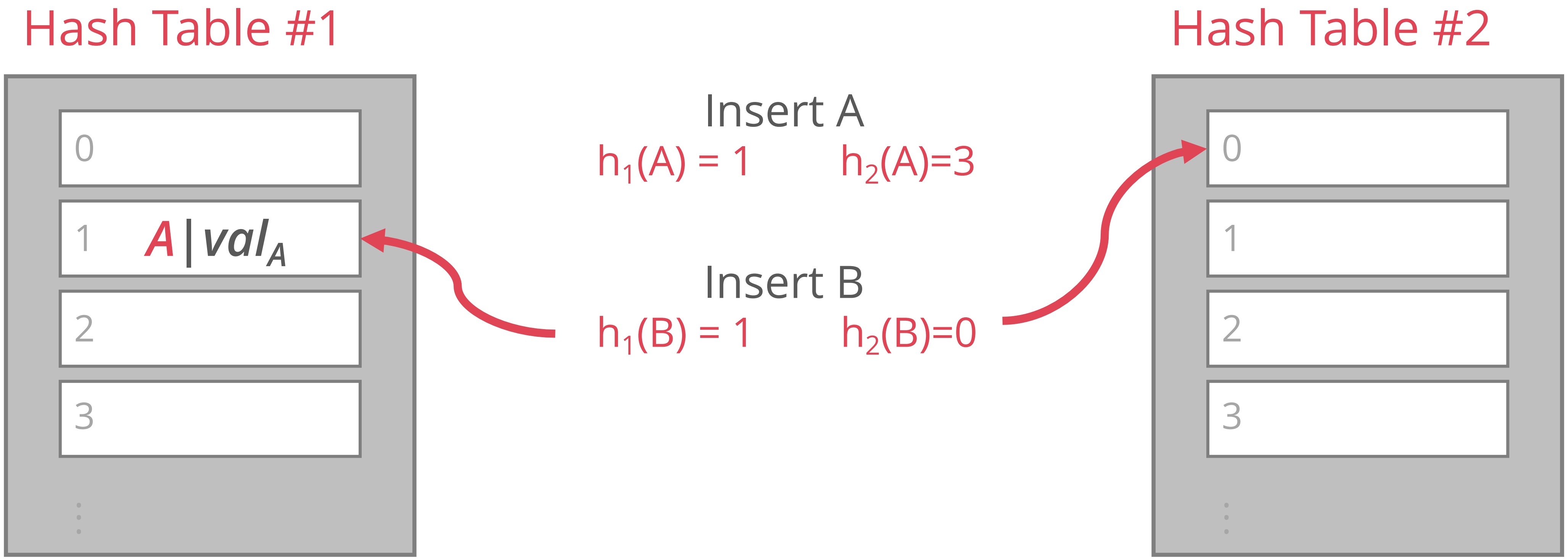
Cuckoo Hashing



Cuckoo Hashing

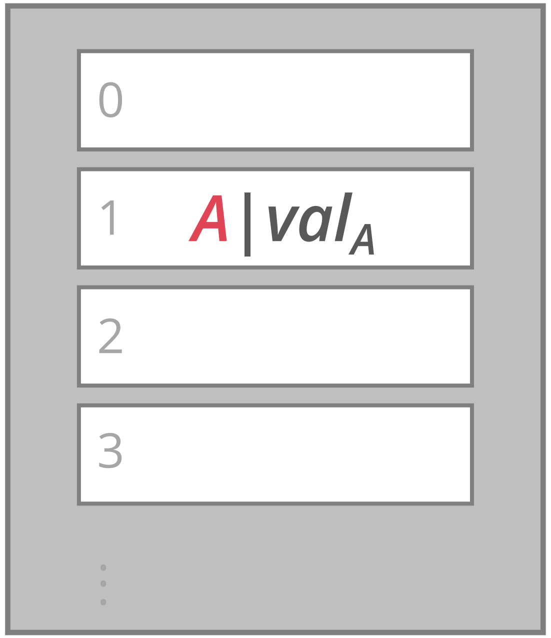


Cuckoo Hashing



CUCKOO HASHING

Hash Table #1



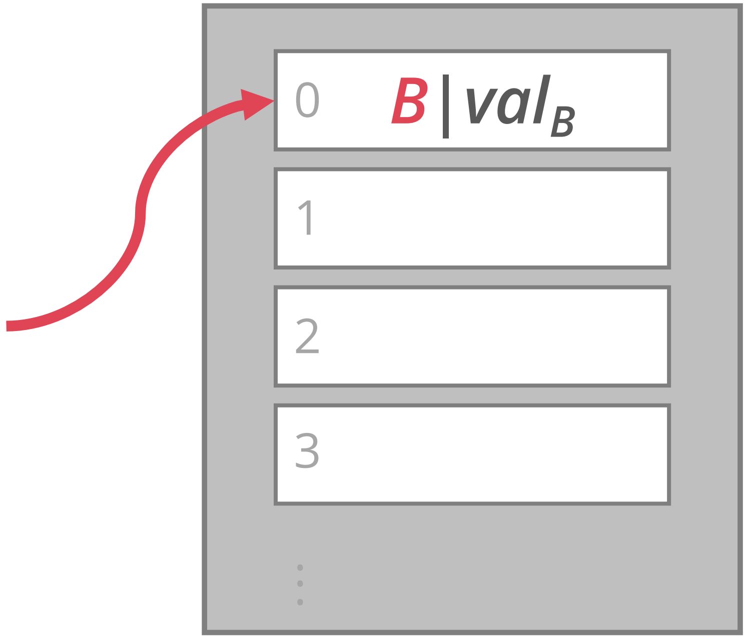
Insert A

$h_1(A) = 1$ $h_2(A)=3$

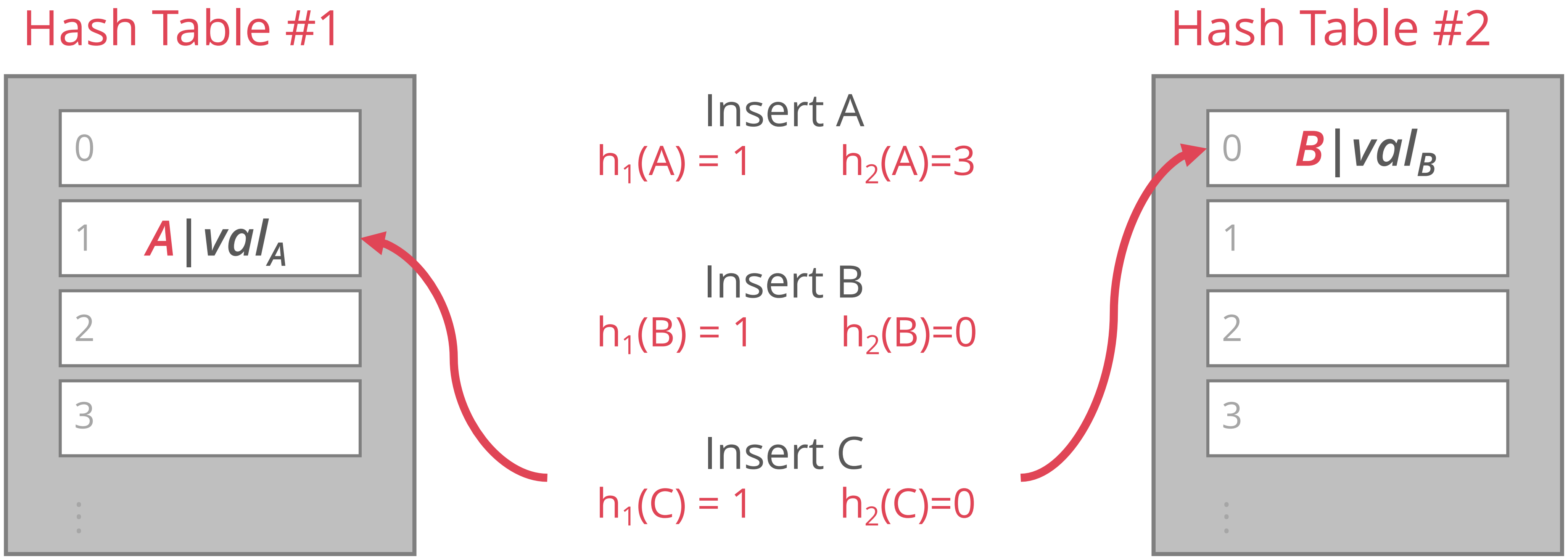
Insert B

$h_1(B) = 1$ $h_2(B)=0$

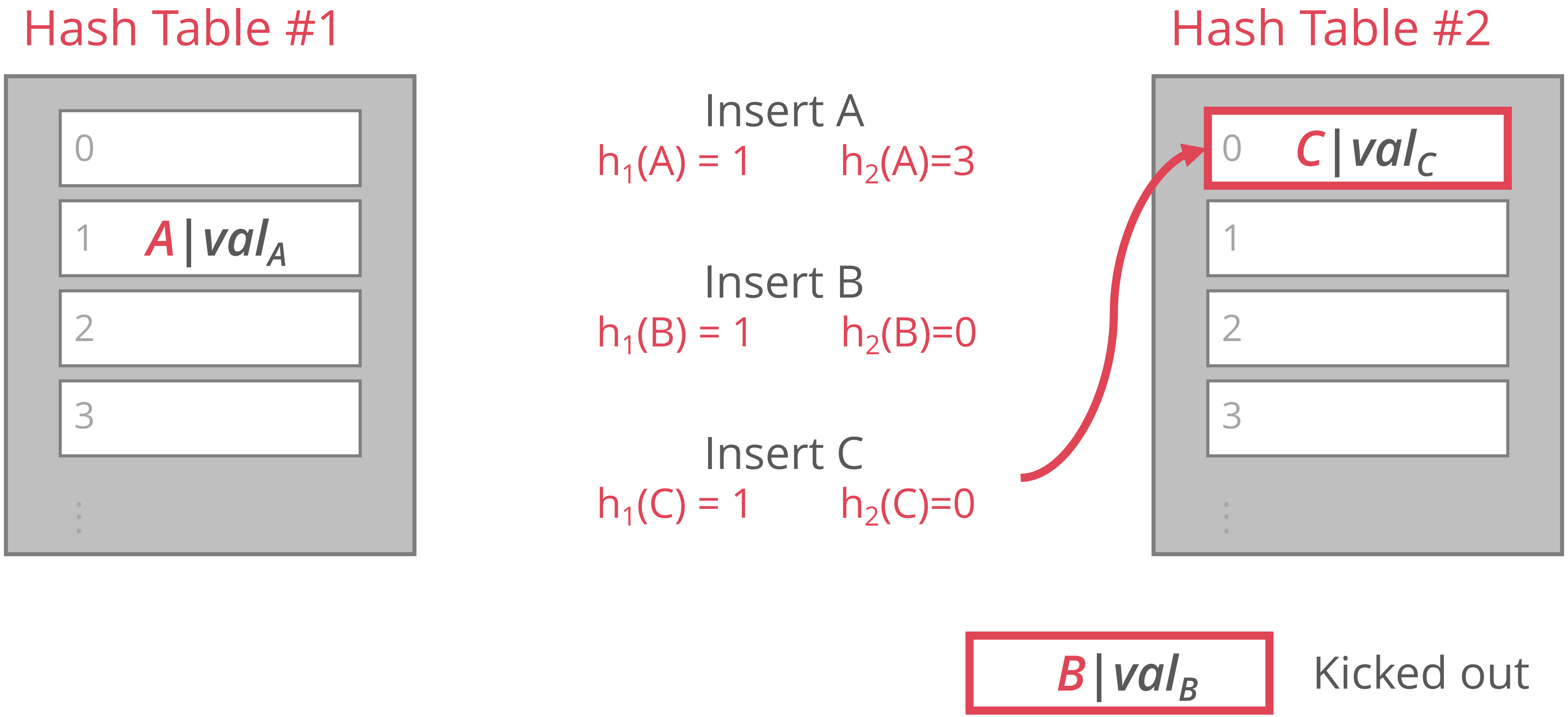
Hash Table #2



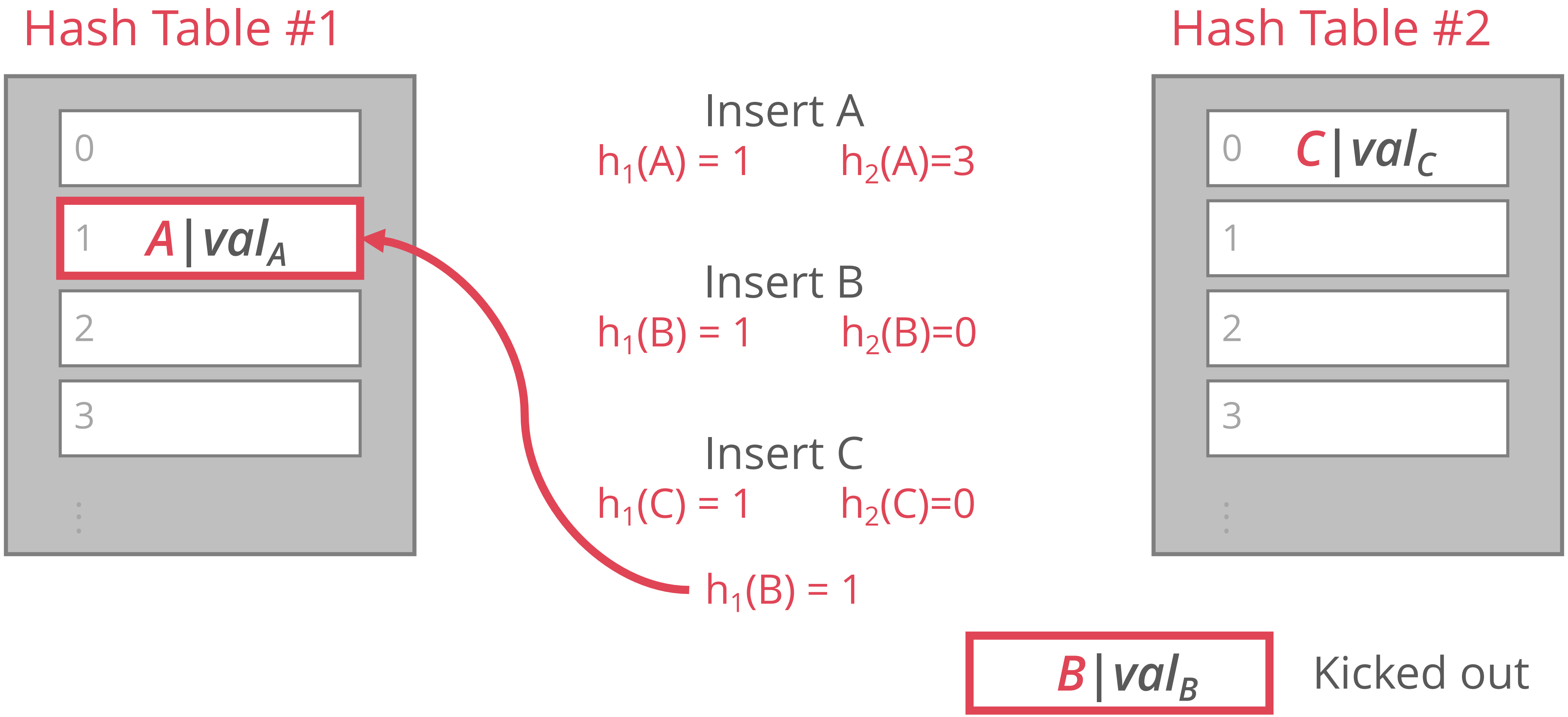
Cuckoo Hashing



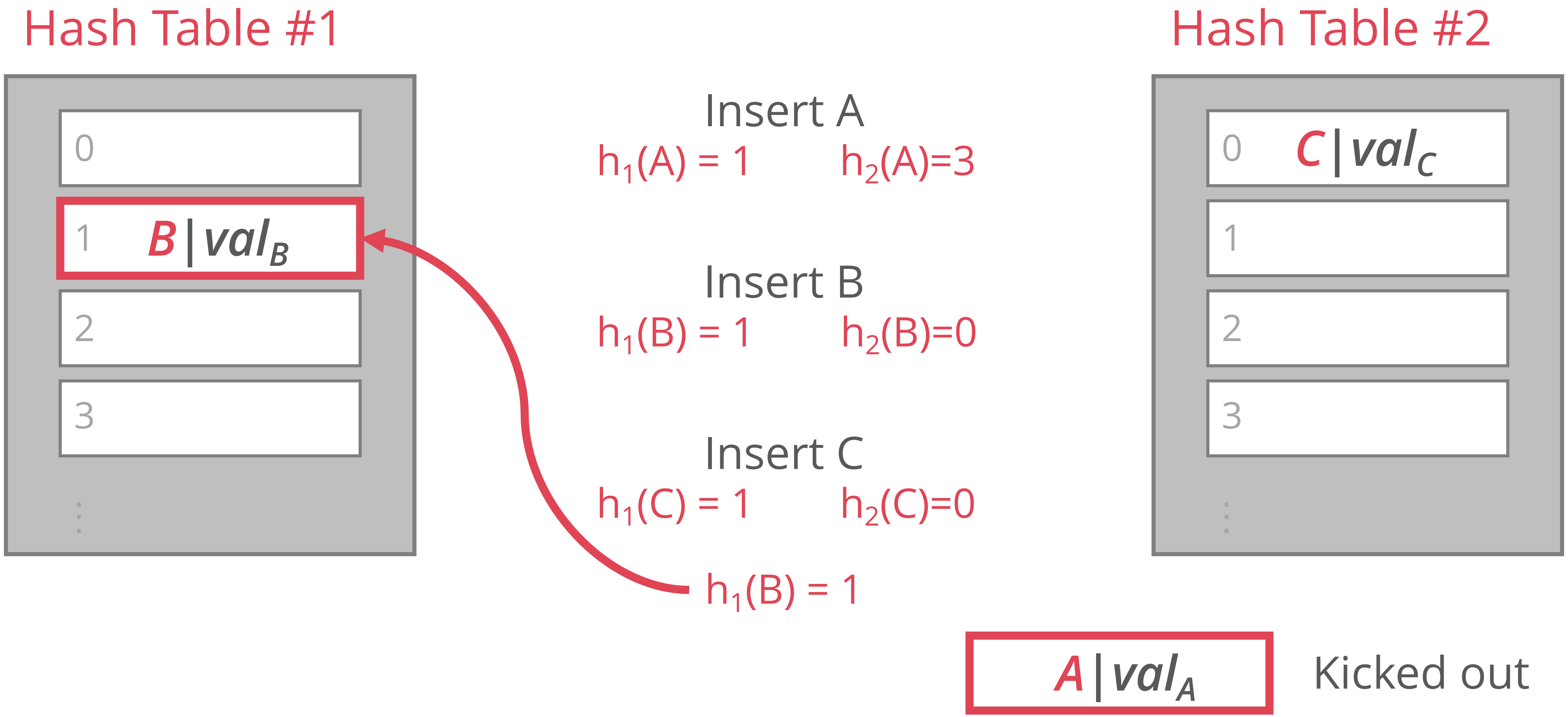
Cuckoo Hashing



Cuckoo Hashing



Cuckoo Hashing



Cuckoo Hashing

Hash Table #1

0	
1	$B\|val_B$
2	
3	
⋮	

Insert A

$h_1(A) = 1$ $h_2(A)=3$

Insert B

$h_1(B) = 1$ $h_2(B)=0$

Insert C

$h_1(C) = 1$ $h_2(C)=0$

$h_1(B) = 1$

$h_2(A) = 3$

Hash Table #2

0	$C\|val_C$
1	
2	
3	$A\|val_A$
⋮	

Cuckoo Hashing

Make sure we don't get stuck in an infinite loop when moving keys

If we find a cycle, then we can **rebuild** the entire hash table with new hash functions

- With **two** hash functions, we (probably) won't need to rebuild the table until it is at about 50% full
- With **three** hash functions, we (probably) won't need to rebuild the table until it is at about 90% full

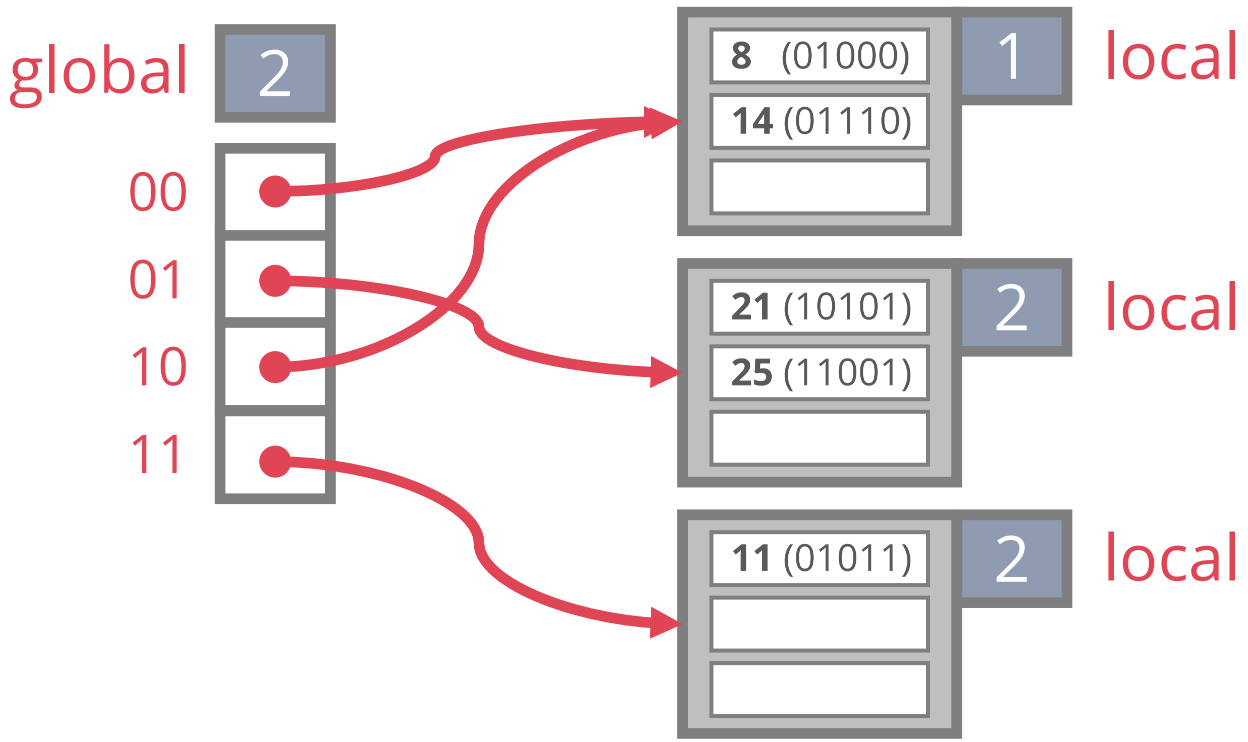
Extendible Hashing

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

Use **directory of pointers to buckets**, double # of buckets by doubling the directory, splitting just the bucket that overflowed

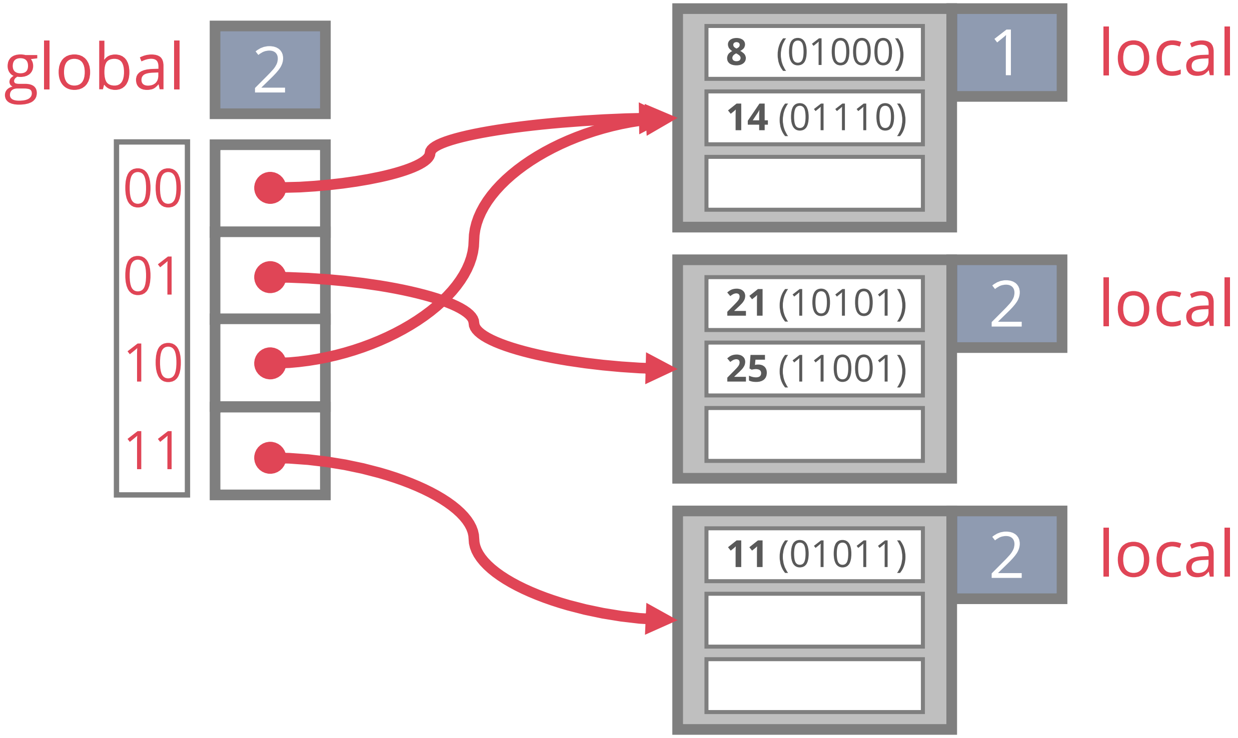
Directory much smaller than file, so doubling it is much cheaper

Extendible Hashing



Note: we depict as index data entries ***h***(***k***) instead of ***k****

Extendible Hashing

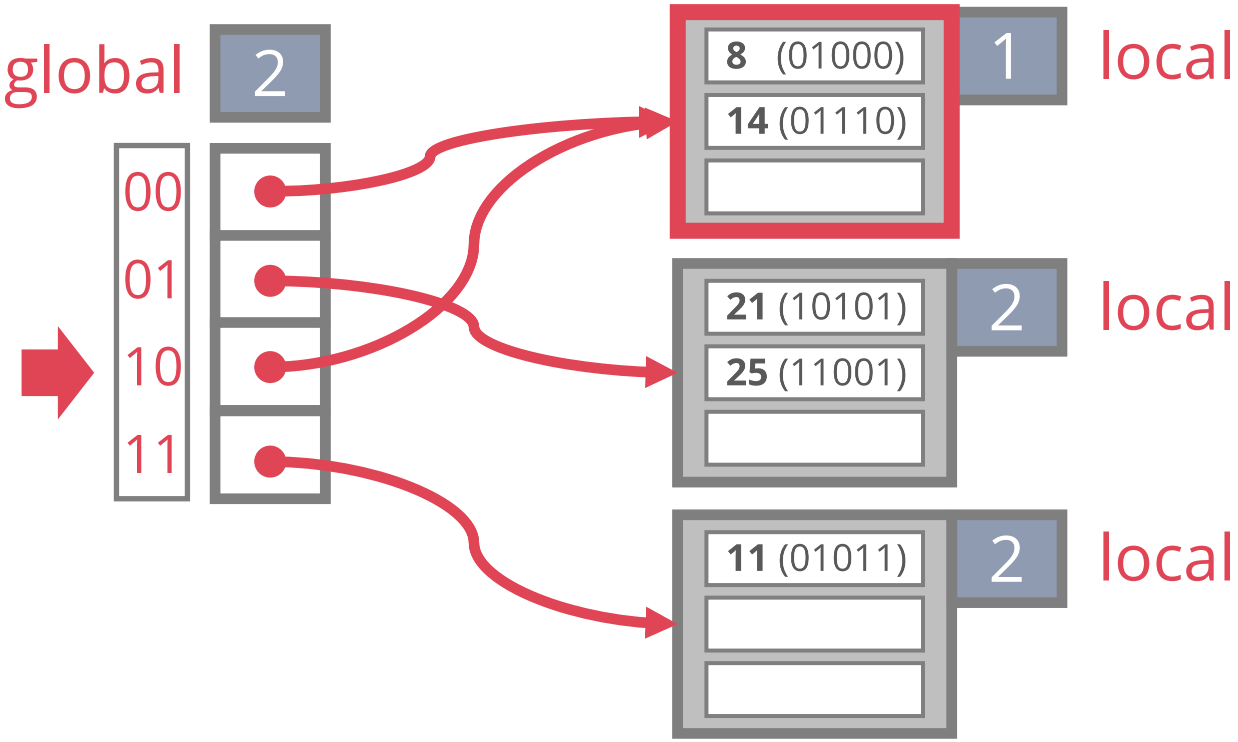


Find A

hash(A) = **14** = $011\boxed{10}_2$

To find a bucket for A, take last "global depth" # of bits of hash(A)

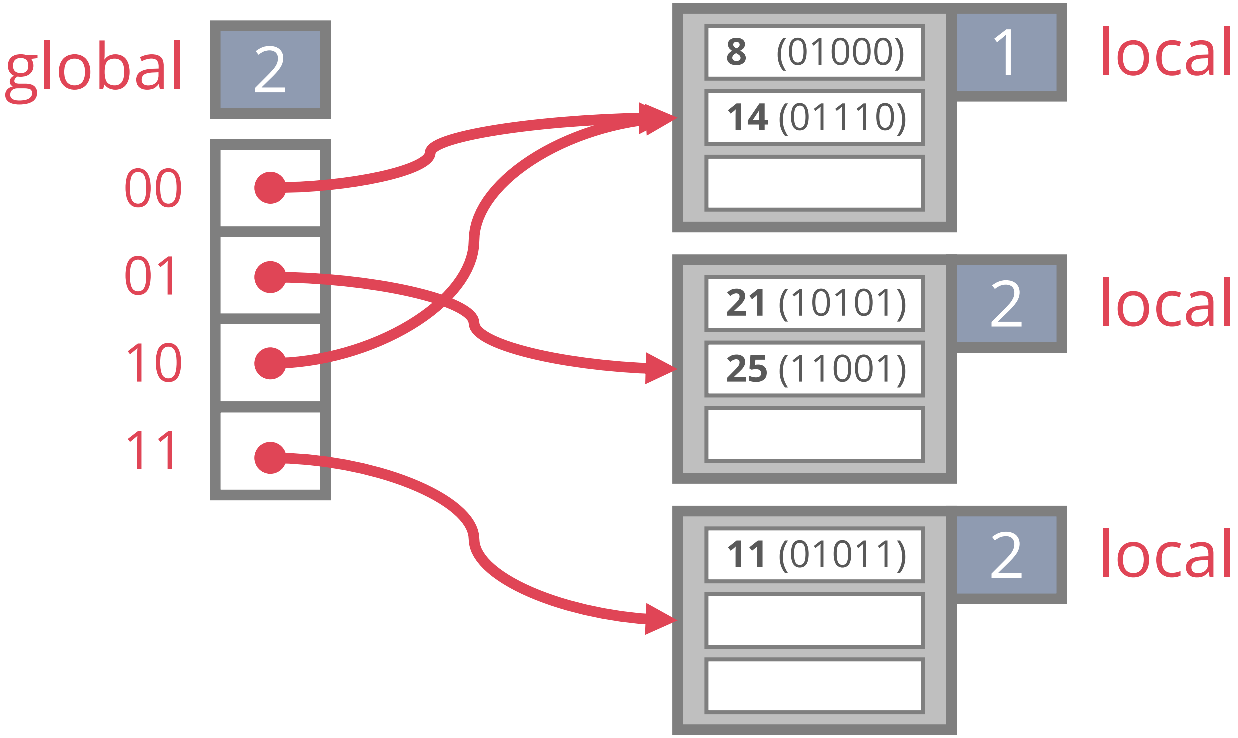
EXTENDIBLE HASHING



Find A

hash(A) = **14** = $011\boxed{10}_2$

Extendible Hashing



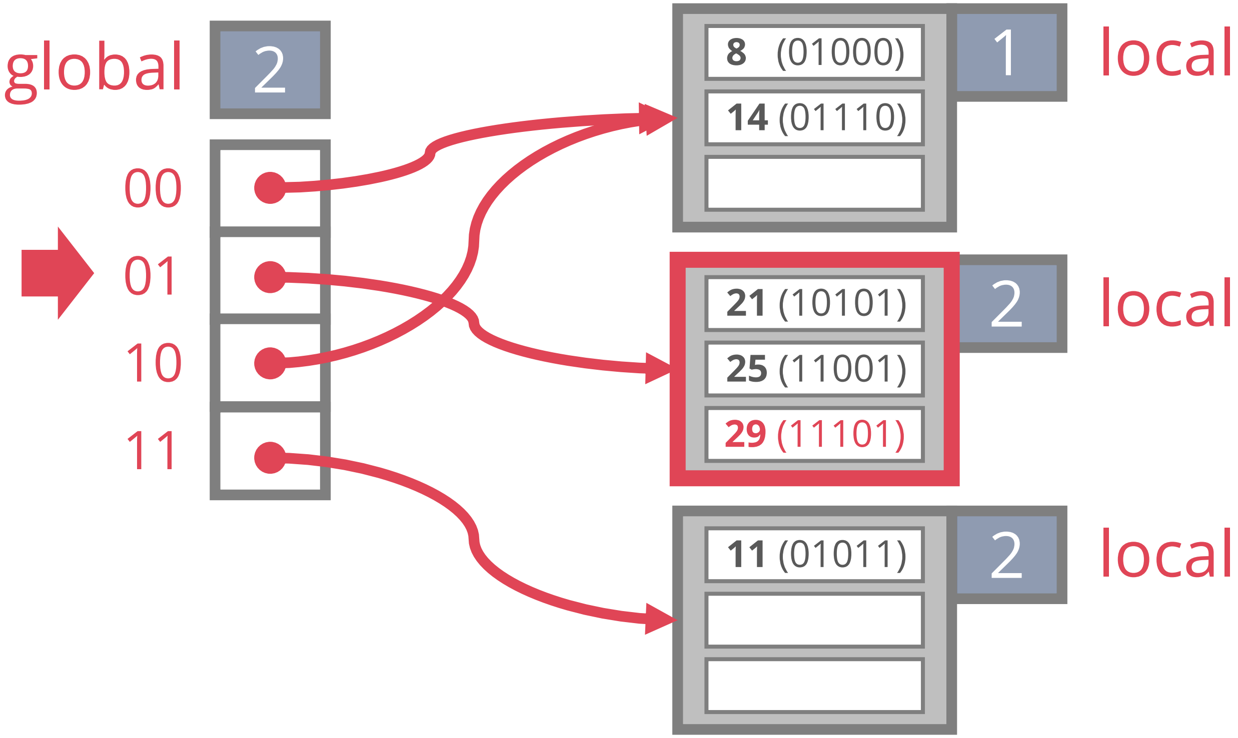
Find A

hash(A) = **14** = 01110_2

Insert B

hash(B) = **29** = 11101_2

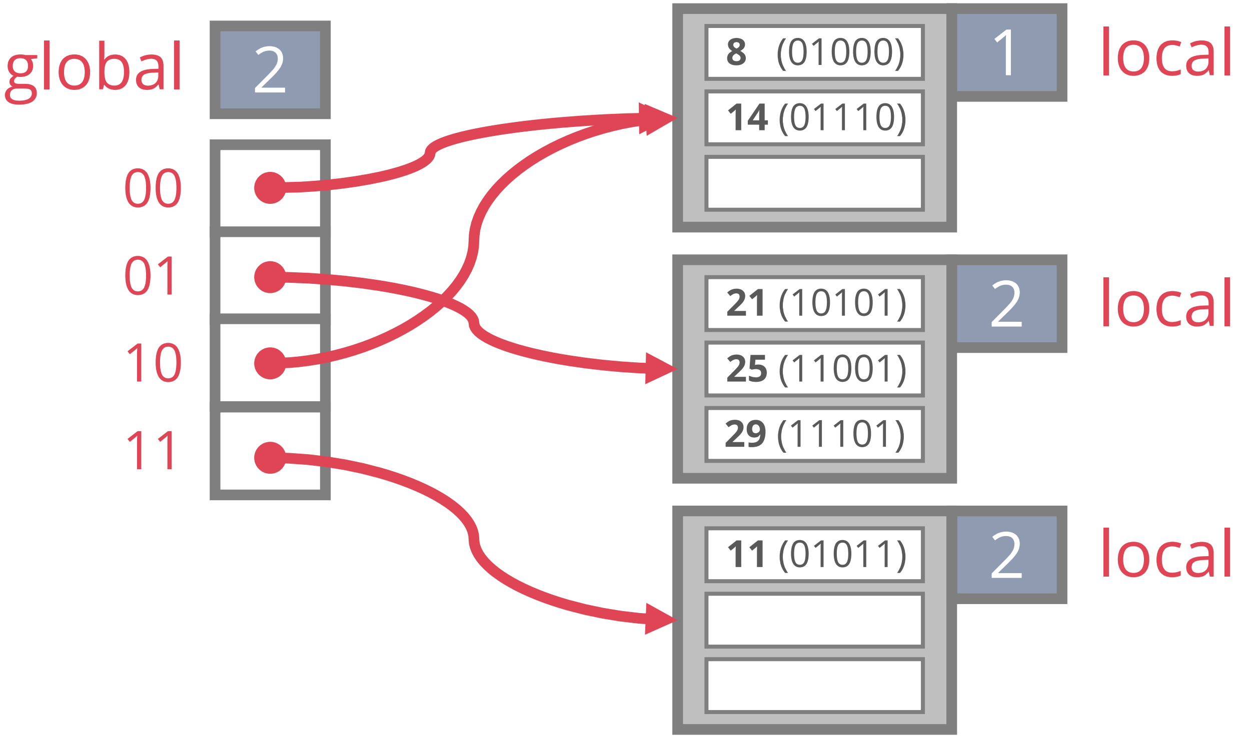
Extendible Hashing



Find A
hash(A) = **14** = 01110_2

Insert B
hash(B) = **29** = $111\boxed{01}_2$

Extendible Hashing



Find A

hash(A) = **14** = 01110_2

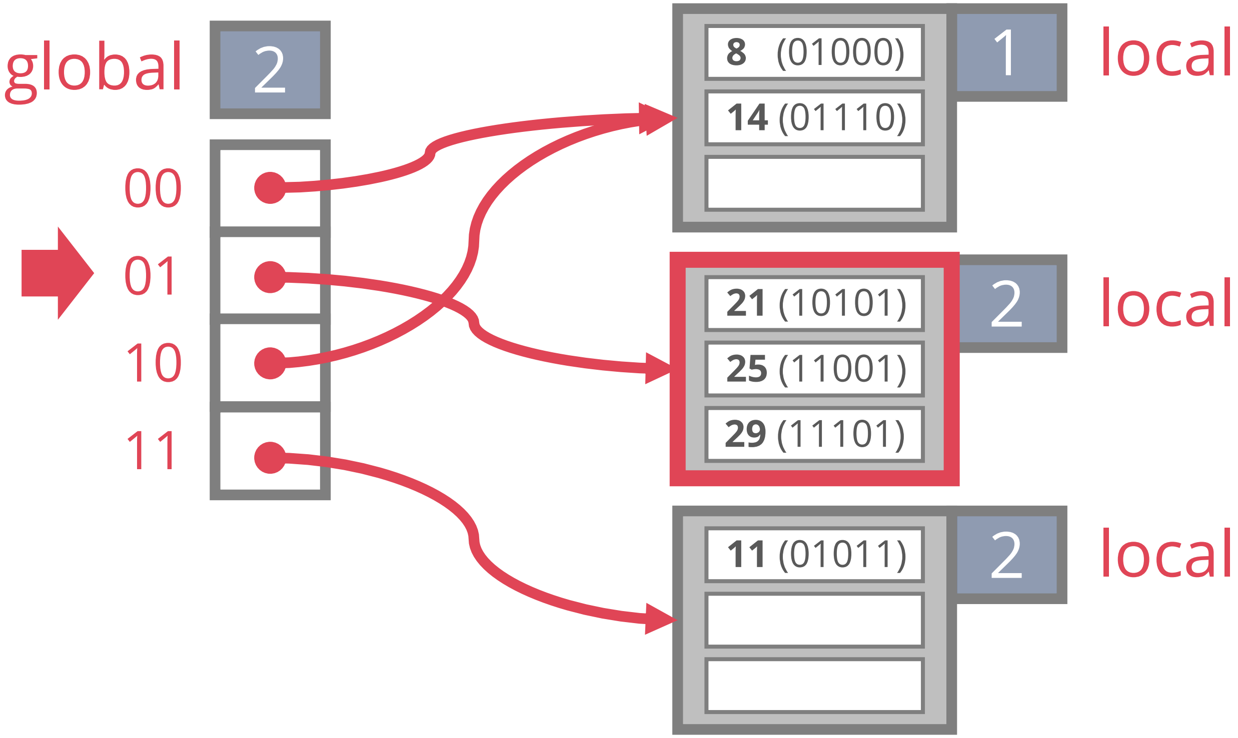
Insert B

hash(B) = **29** = 11101_2

Insert C

hash(C) = **5** = $001\boxed{01}_2$

Extendible Hashing



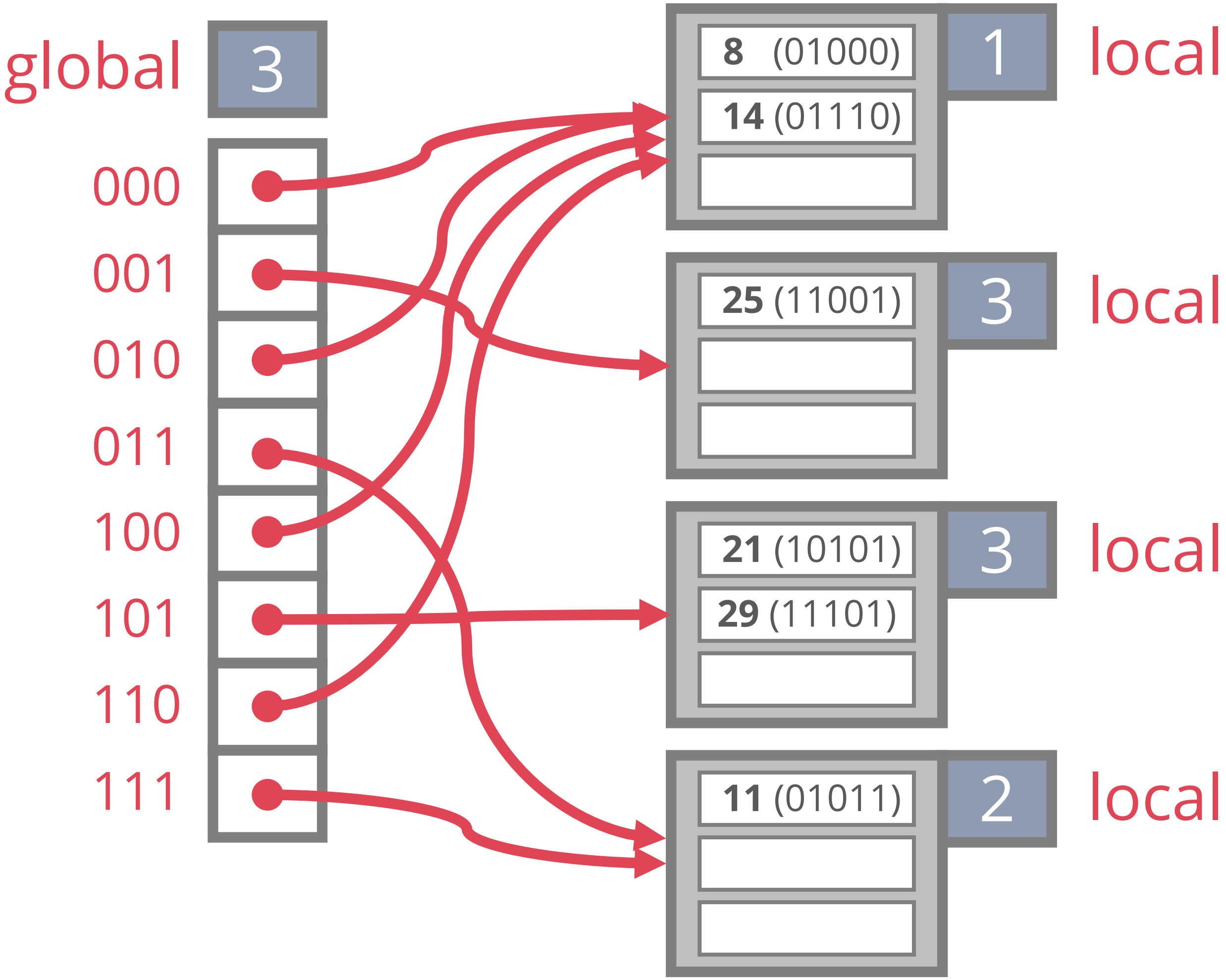
Find A
hash(A) = **14** = 01110_2

Insert B
hash(B) = **29** = 11101_2

Insert C
hash(C) = **5** = $001\boxed{01}_2$

Split bucket if full (allocate new page, redistribute, increase local & global)

EXTENDIBLE HASHING

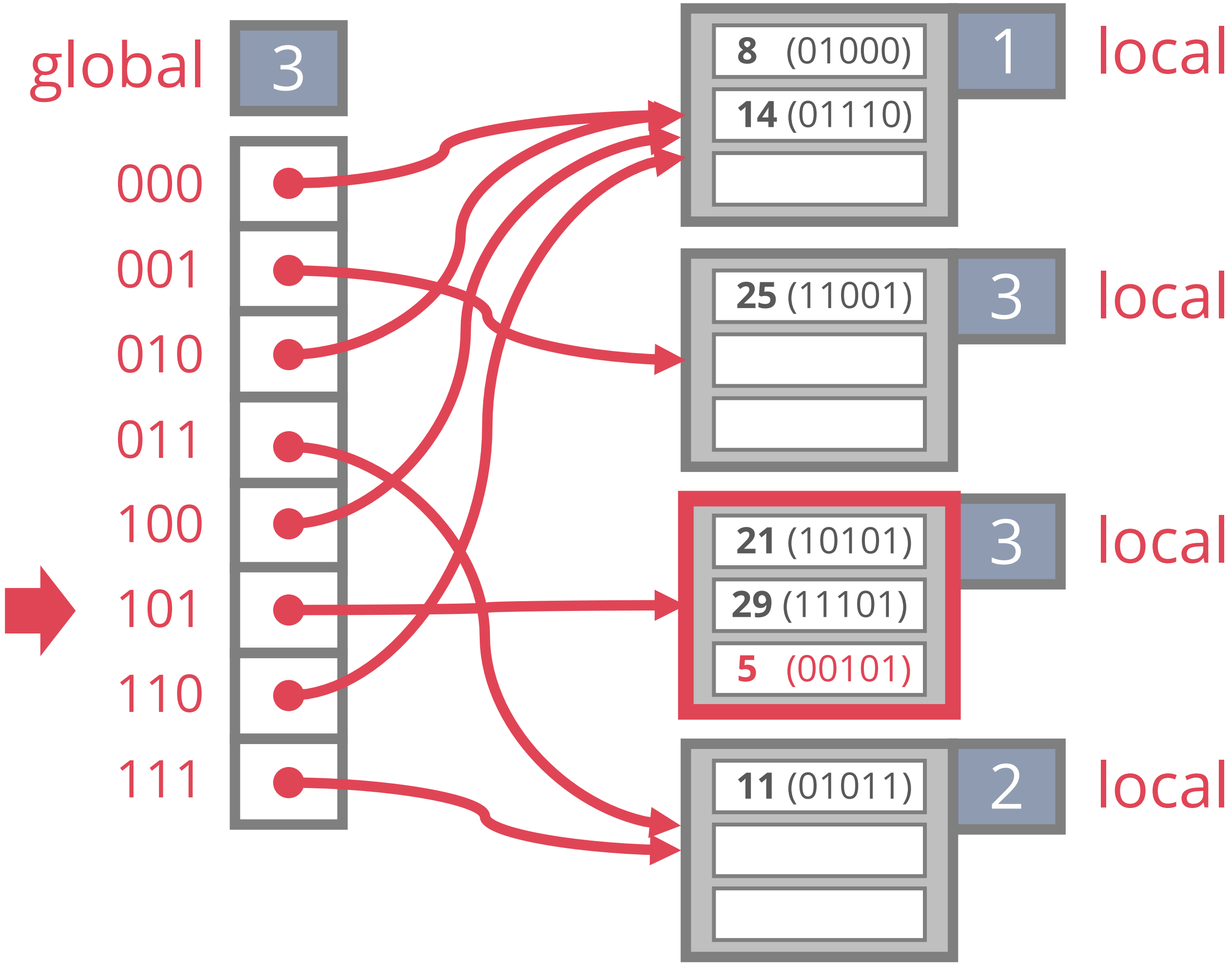


Find A
hash(A) = **14** = 01110_2

Insert B
hash(B) = **29** = 11101_2

Insert C
hash(C) = **5** = 00101_2

Extendible Hashing



Find A
hash(A) = **14** = 01110_2

Insert B
hash(B) = **29** = 11101_2

Insert C
hash(C) = **5** = $00\boxed{101}_2$

Extendible Hashing

Global depth of directory = max # of bits needed to tell which bucket an entry belongs to

Local depth of a bucket = # of bits used to determine if an entry belongs to this bucket

Splitting a bucket does not always require doubling the directory

- Buckets with local depth < global depth have multiple pointers to them
- Splitting such buckets does not require doubling

Directory is doubled by **copying it over** and ”fixing” pointer to split image page (use of least significant bits enables efficient doubling via copying!)

LINEAR HASHING

Handles the problem of long overflow chains without using a directory

Idea: Use a family of hash functions $\boldsymbol{h}_0$, $\boldsymbol{h}_1$, $\boldsymbol{h}_2$, …

- N = initial # of buckets
- $\boldsymbol{h}$ is some hash function (range is **not** 0 to N - 1)
- $\boldsymbol{h}_i(\text{key}) = \boldsymbol{h}(\text{key}) \bmod (2^i N)$
- If $N = 2^{d_0}$, for some d_0, then $\boldsymbol{h}_i$ consists of applying $\boldsymbol{h}$ and looking at last d_i bits, where $d_i = d_0 + i$
- $\boldsymbol{h}_{i+1}$ doubles the range of $\boldsymbol{h}_i$ (similar to directory doubling)

Linear Hashing

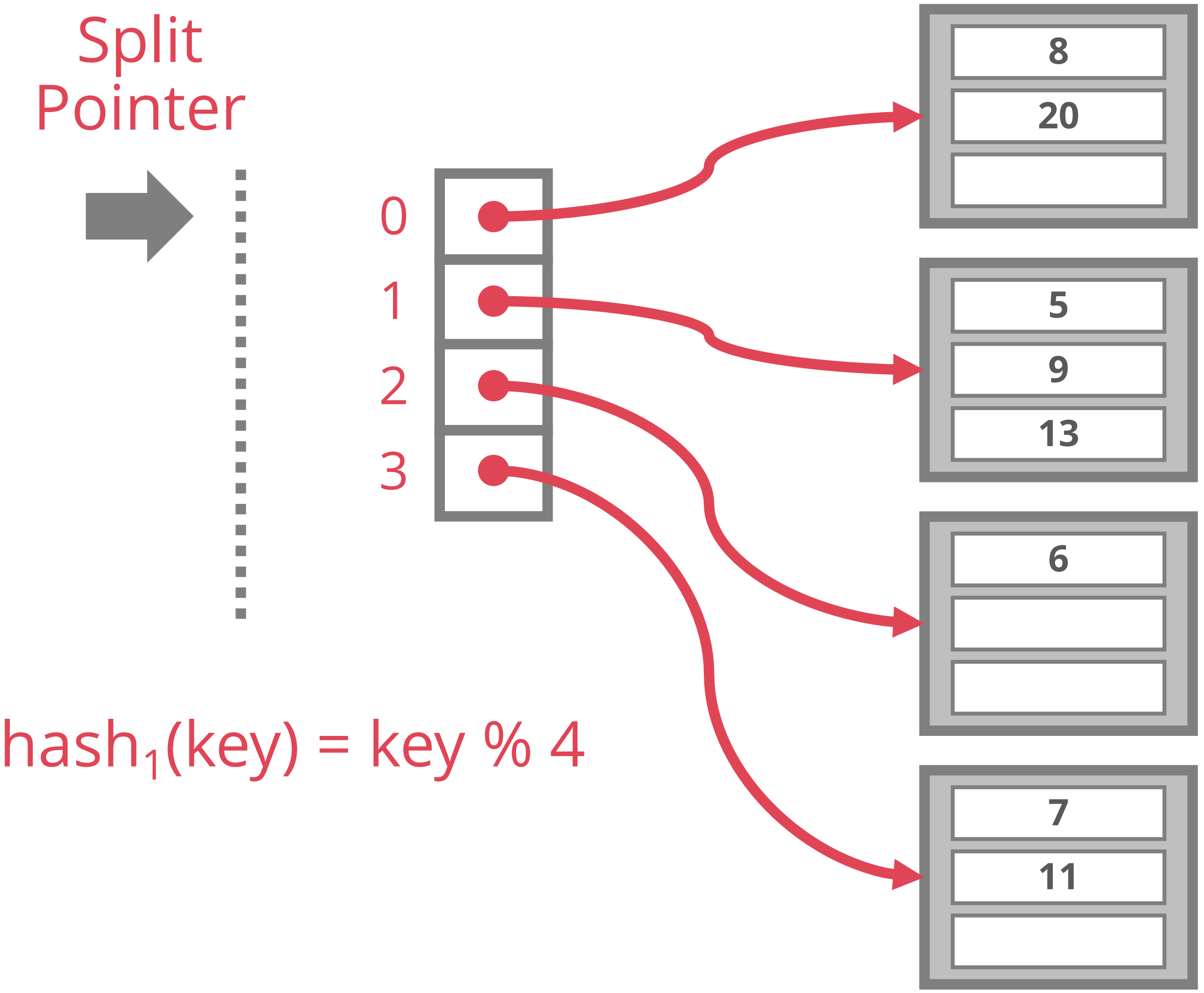
Maintains a pointer that tracks the next bucket to split

When any bucket overflows, split the bucket at the pointer location

Split criterion is left up to the implementation

- Space utilization of a bucket beyond some % capacity, or
- Average length of overflow chains longer than *p* pages

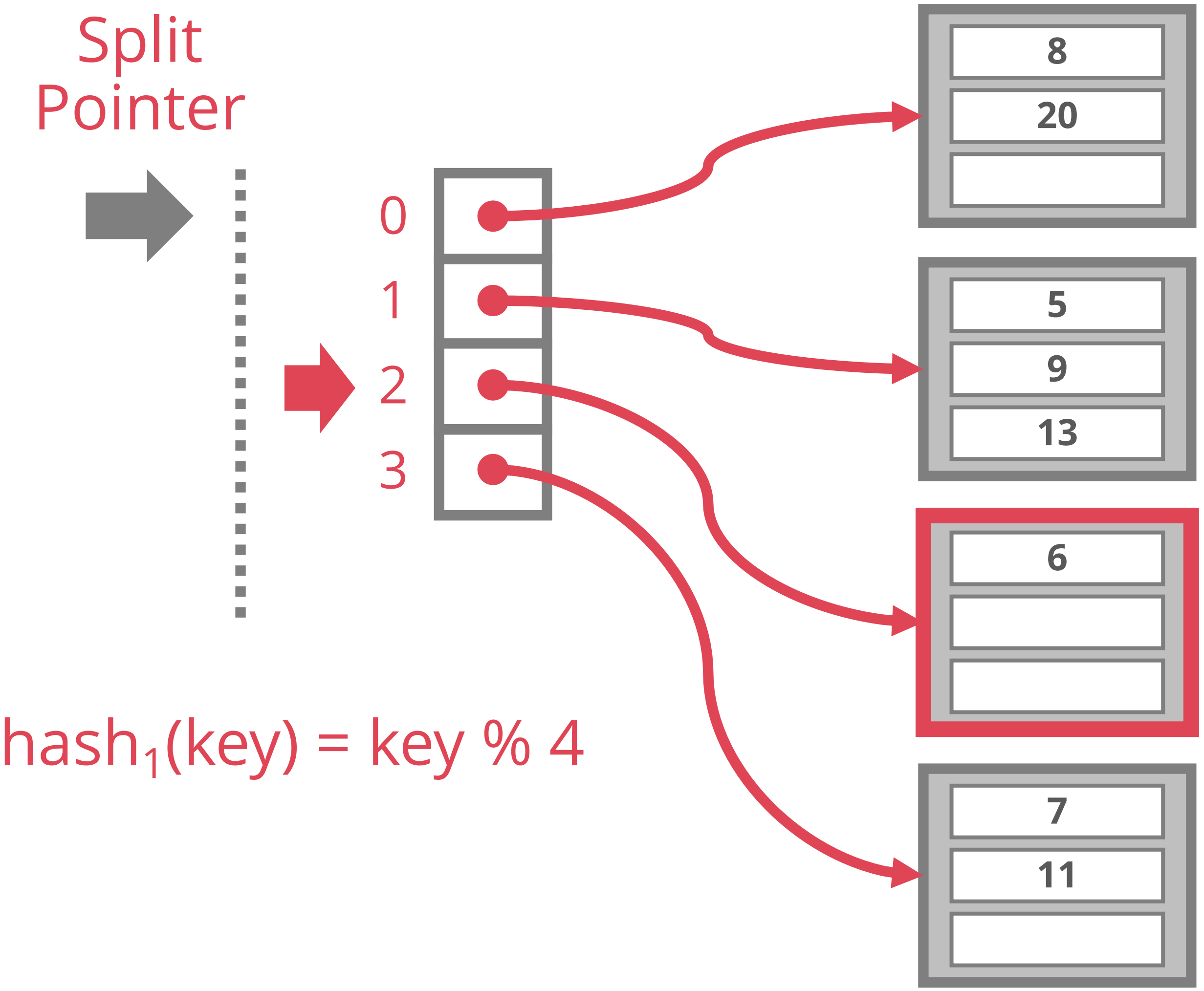
Linear Hashing



$hash_1(key) = key \% 4$

Note: the directory is shown here for presentation purpose, not needed in practice

LINEAR HASHING

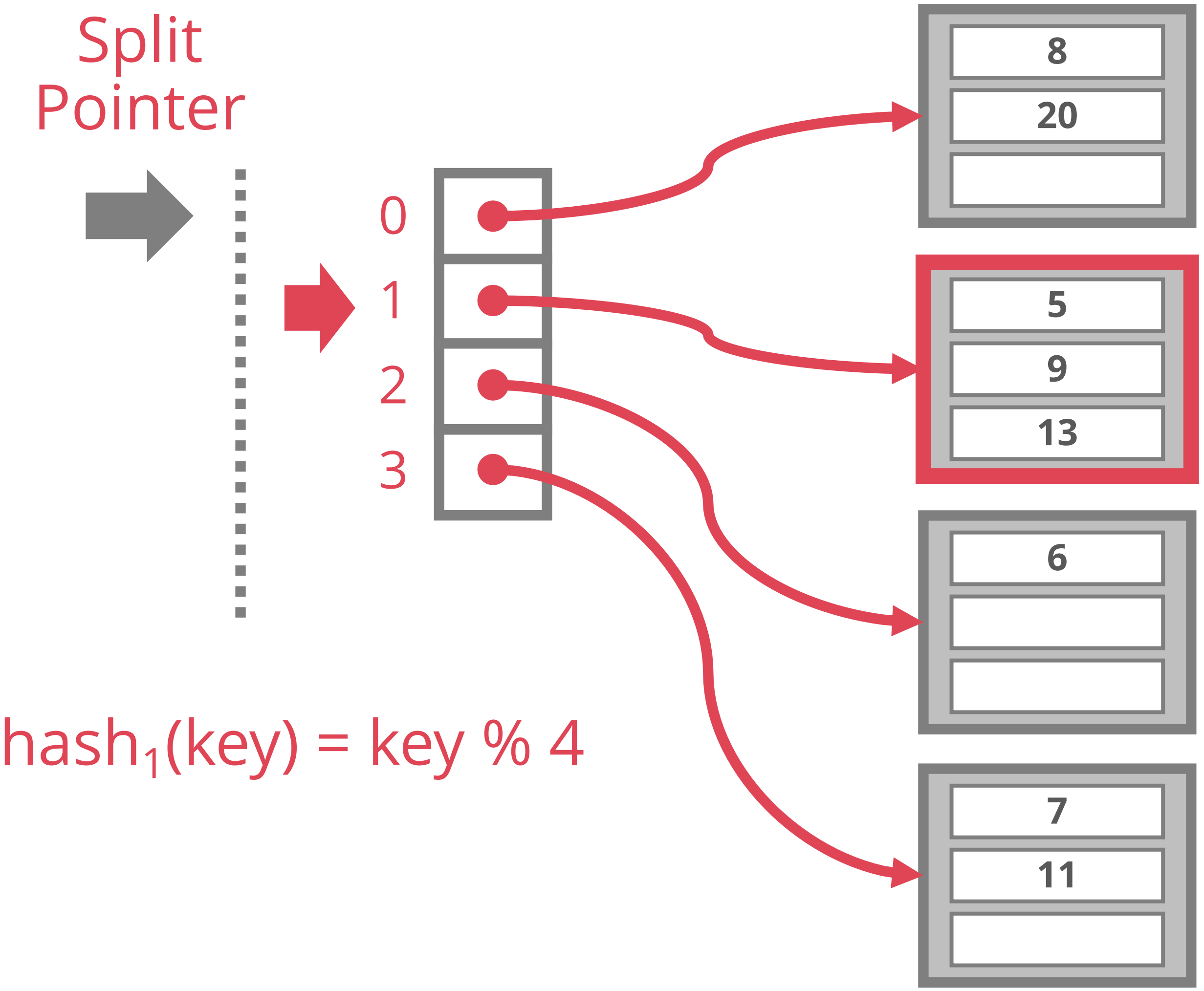


$hash_1(key) = key \% 4$

Find 6

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

LINEAR HASHING



$hash_1(key) = key \% 4$

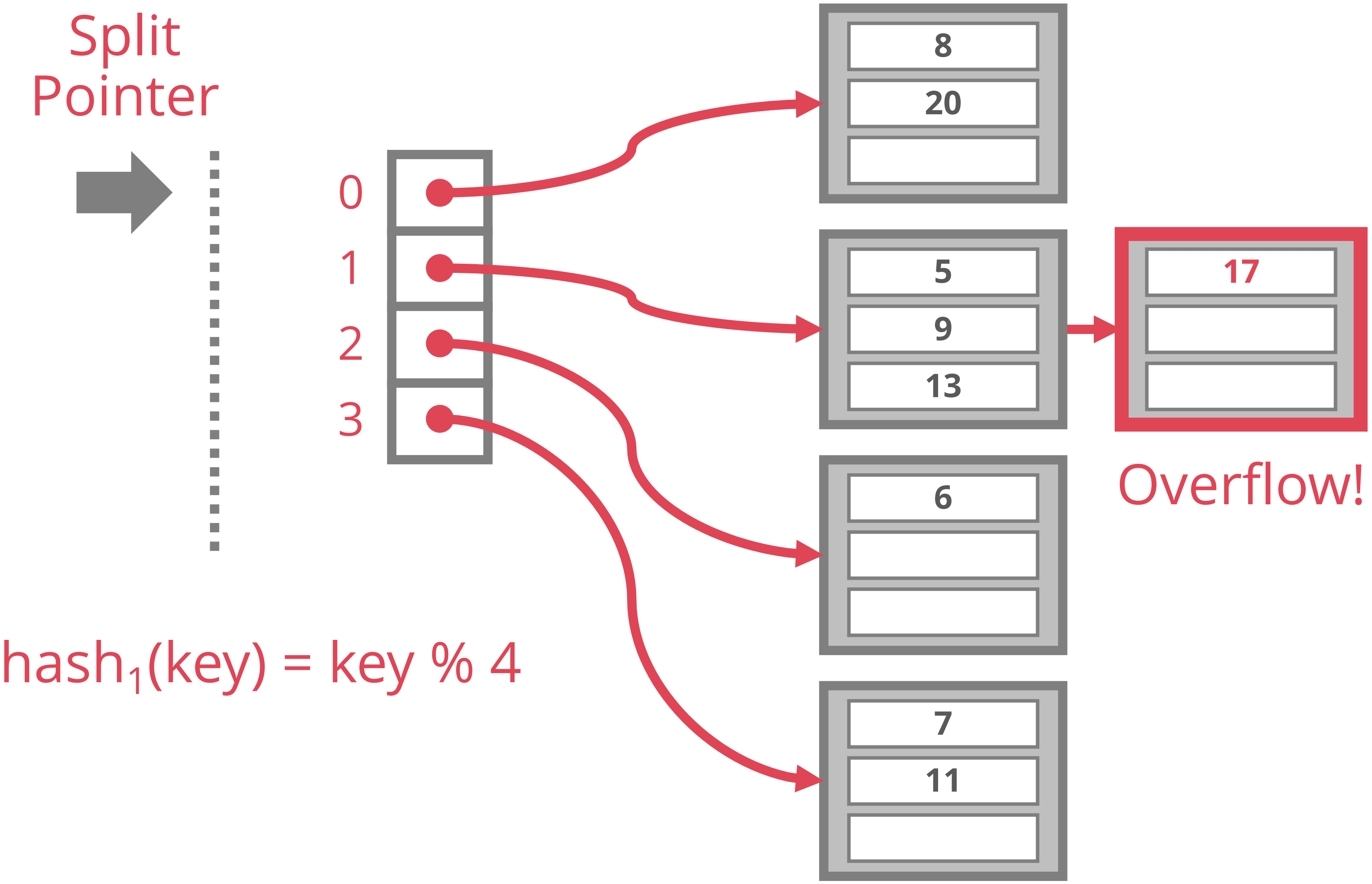
Find 6

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

Insert 17

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

Linear Hashing

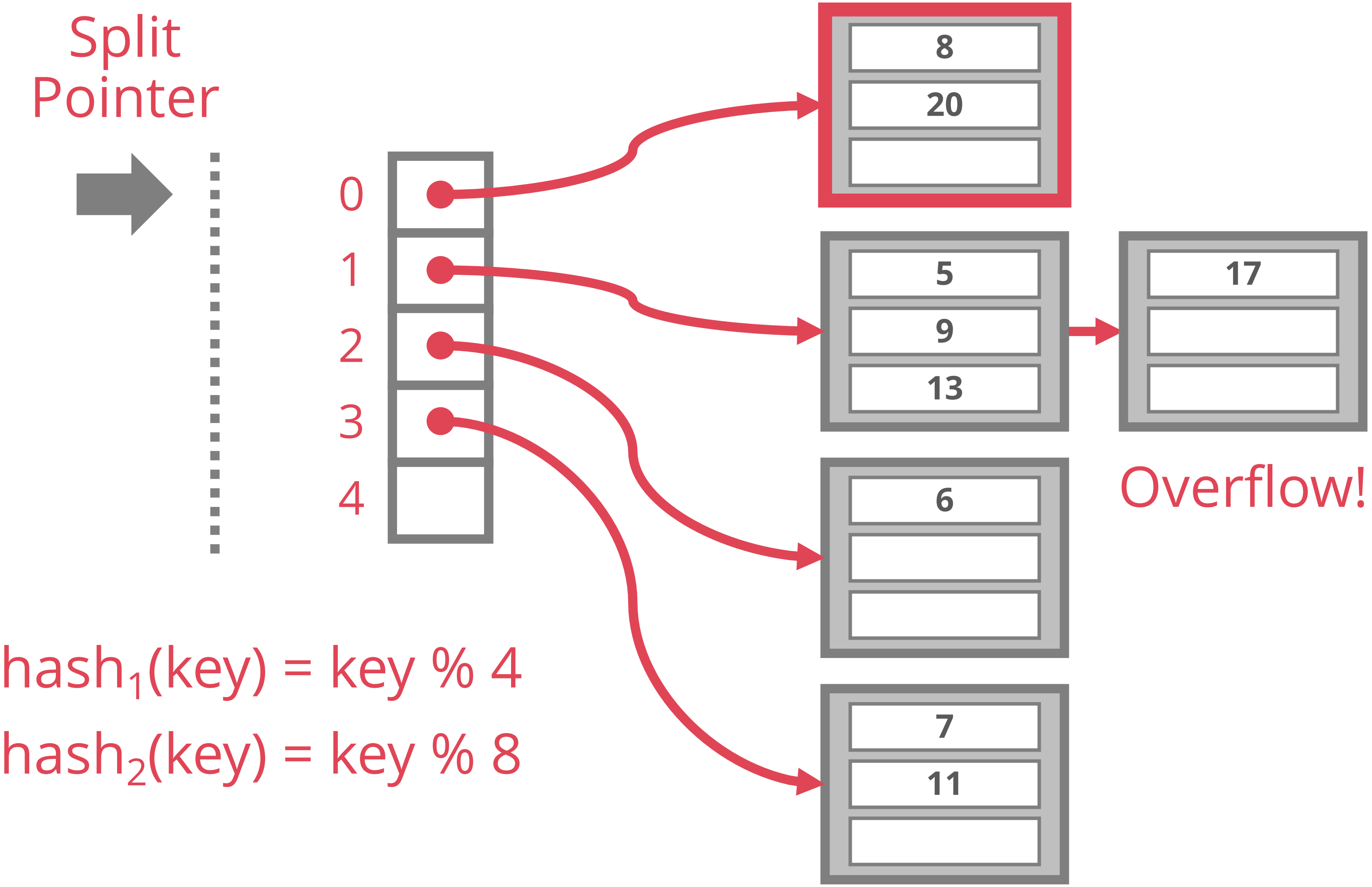


Find 6
$hash_1(6) = 6 \% 4 = 2$

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

$hash_1(key) = key \% 4$

LINEAR HASHING

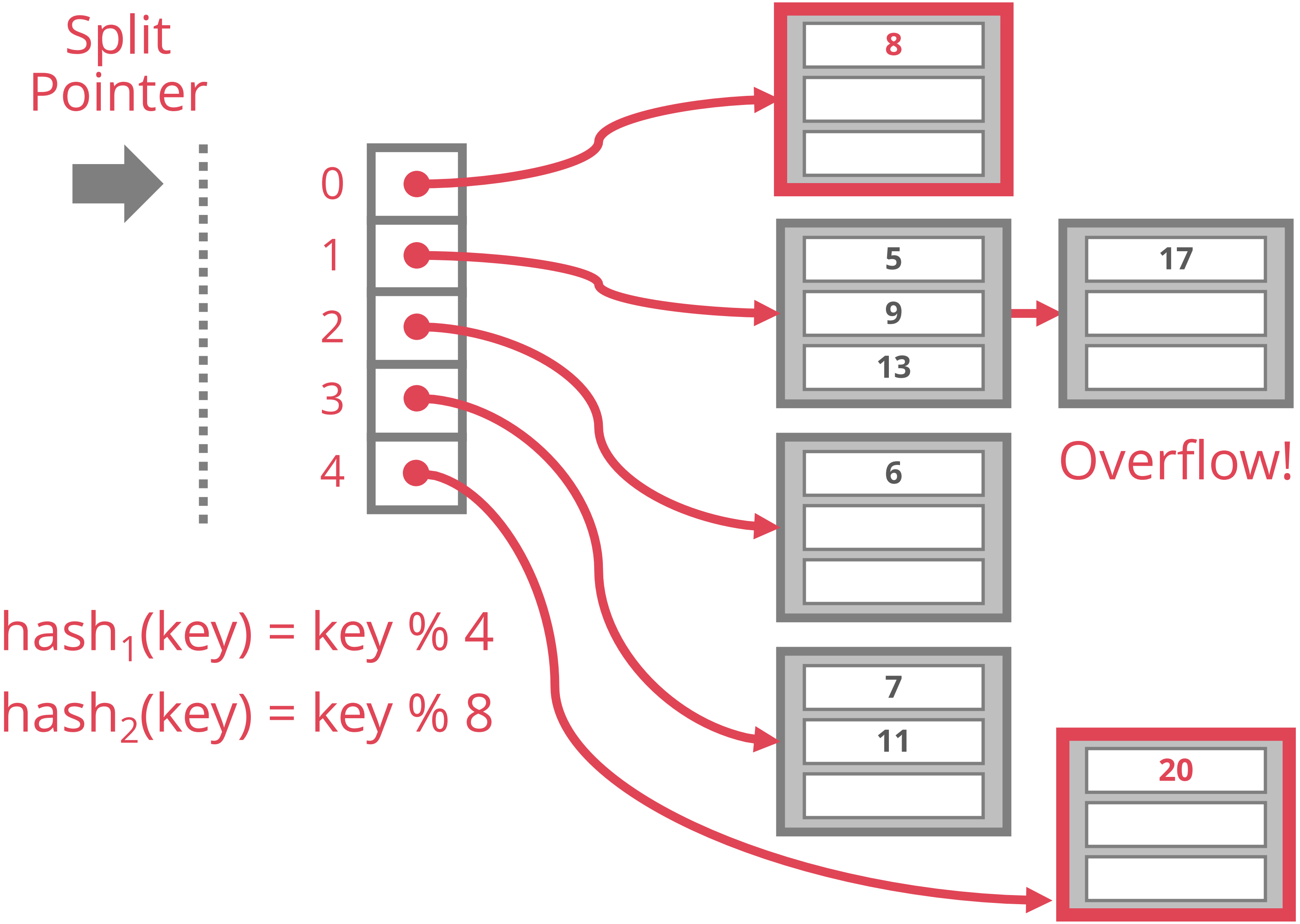


Find 6
$hash_1(6) = 6 \% 4 = 2$

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

$hash_1(key) = key \% 4$
$hash_2(key) = key \% 8$

LINEAR HASHING



$hash_1(key) = key \% 4$

$hash_2(key) = key \% 8$

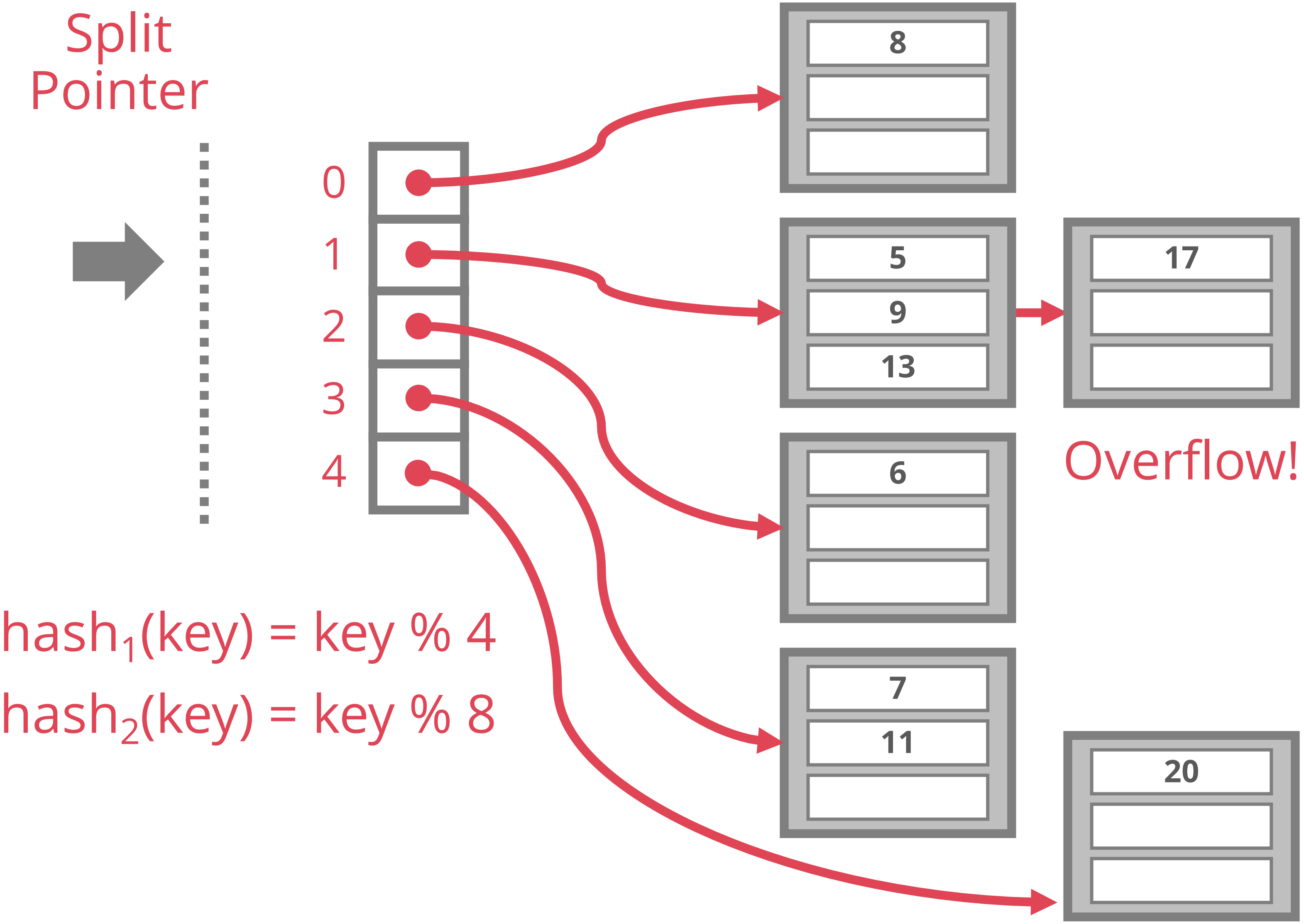
Find 6

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

Insert 17

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

Linear Hashing



$hash_1(key) = key \% 4$

$hash_2(key) = key \% 8$

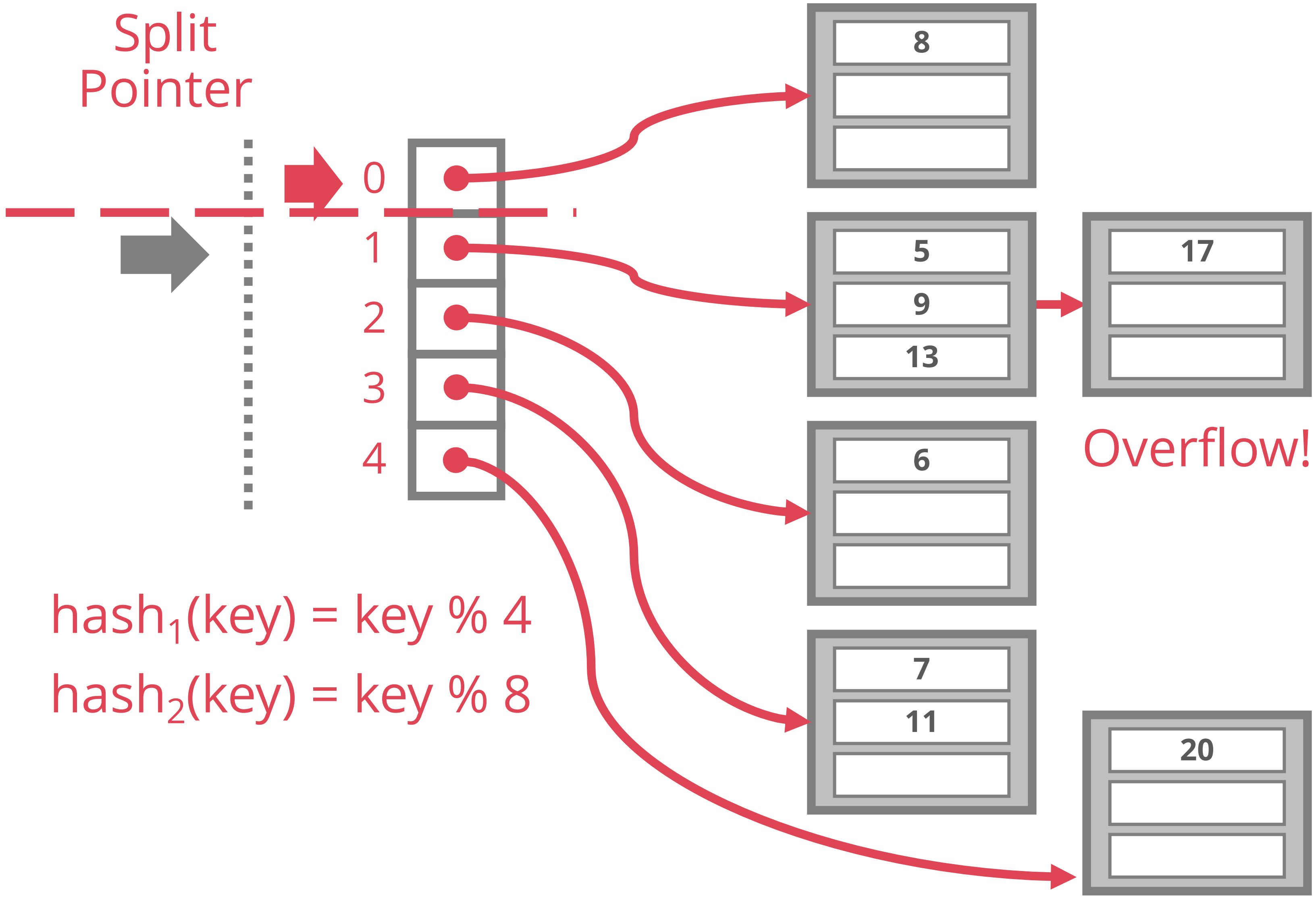
Find 6

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

Insert 17

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

Linear Hashing



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

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

Find 6

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

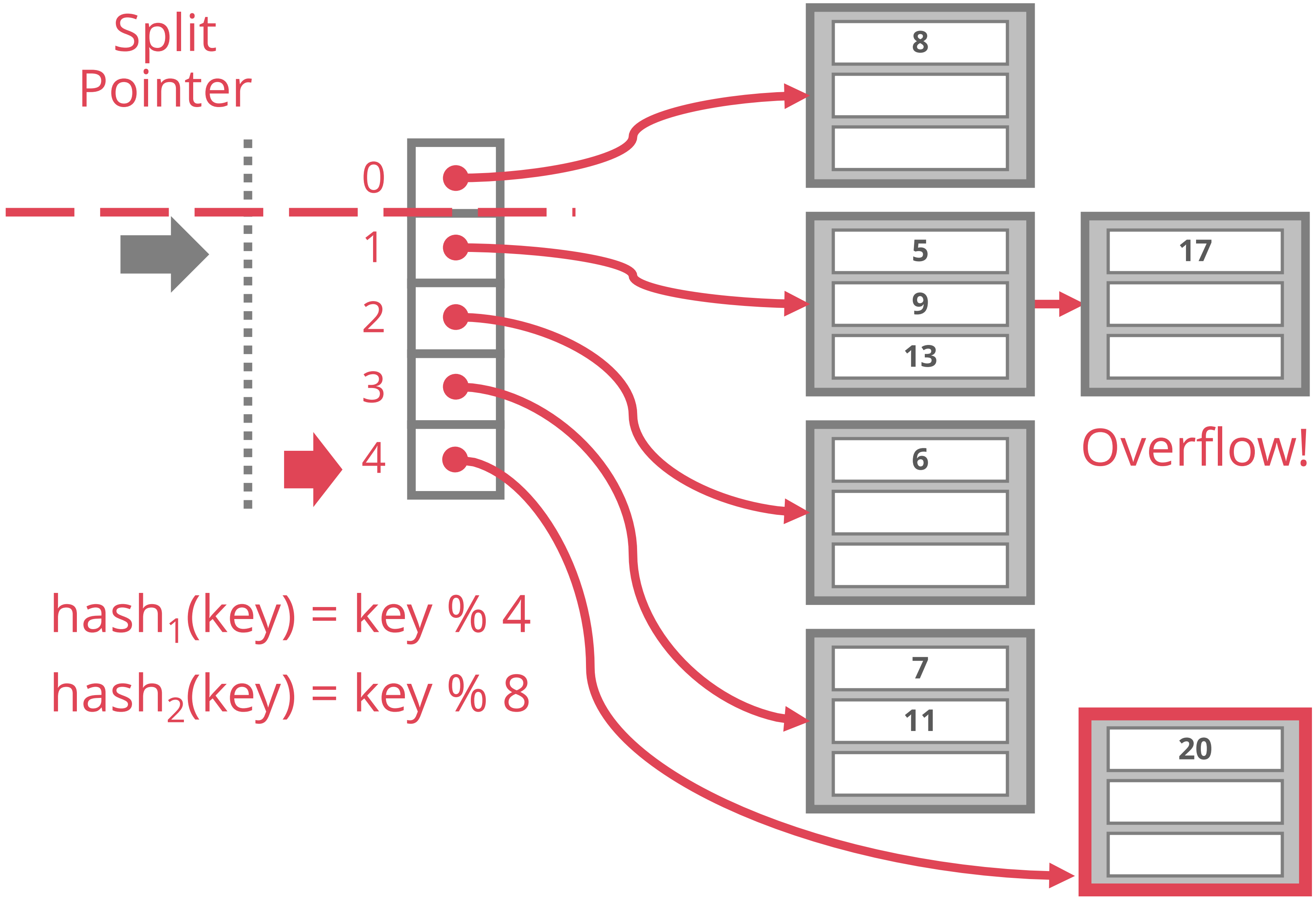
Insert 17

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

Find 20

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

Linear Hashing



$hash_1(key) = key \% 4$

$hash_2(key) = key \% 8$

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

Since buckets are split round-robin, **long overflow chains don't develop**!

- 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

Doubling of directory in Extendible Hashing is similar

- Linear hashing doubles the directory gradually

Primary bucket pages are **created in order**. If they are allocated in sequence too (so that finding i-th is easy), we **don't need a directory**!

Conclusion

Hash-based indexes: best for equality searches, cannot support range searches

Static hashing can lead to long overflow chains

Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it

Linear Hashing avoids directory by splitting buckets round-robin and using overflow pages