THE UNIVERSITY *of* EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #15:

Hash-Based Indexing

R&G: Chapter 11

Recap: File Organisations

Method of arranging a file of records on secondary storage

Heap Files

Store records in no particular order

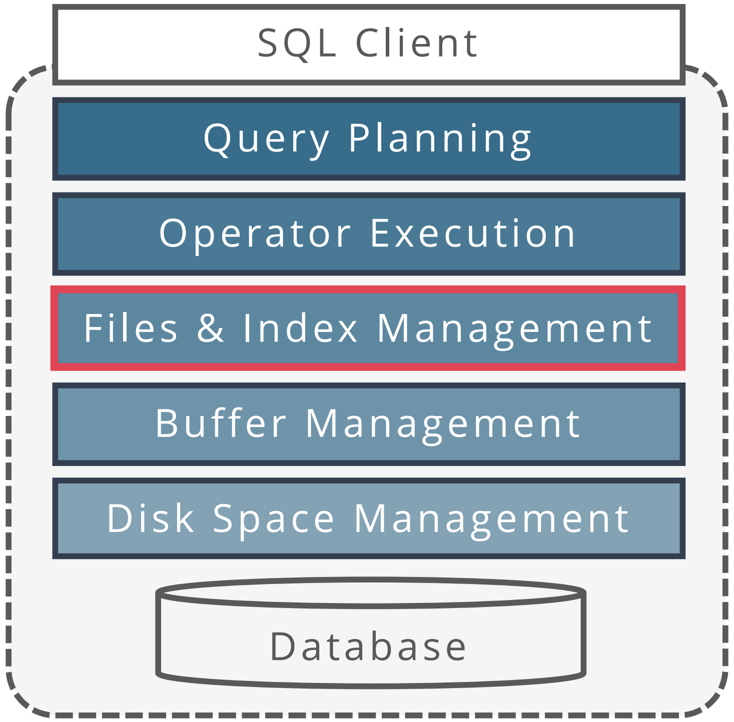
Sorted Files

Store records in sorted order, based on search key fields

Index Files

Store records to enable fast lookup and modifications

Tree-based & hash-based indexes



Recap: In-Memory Hash Table

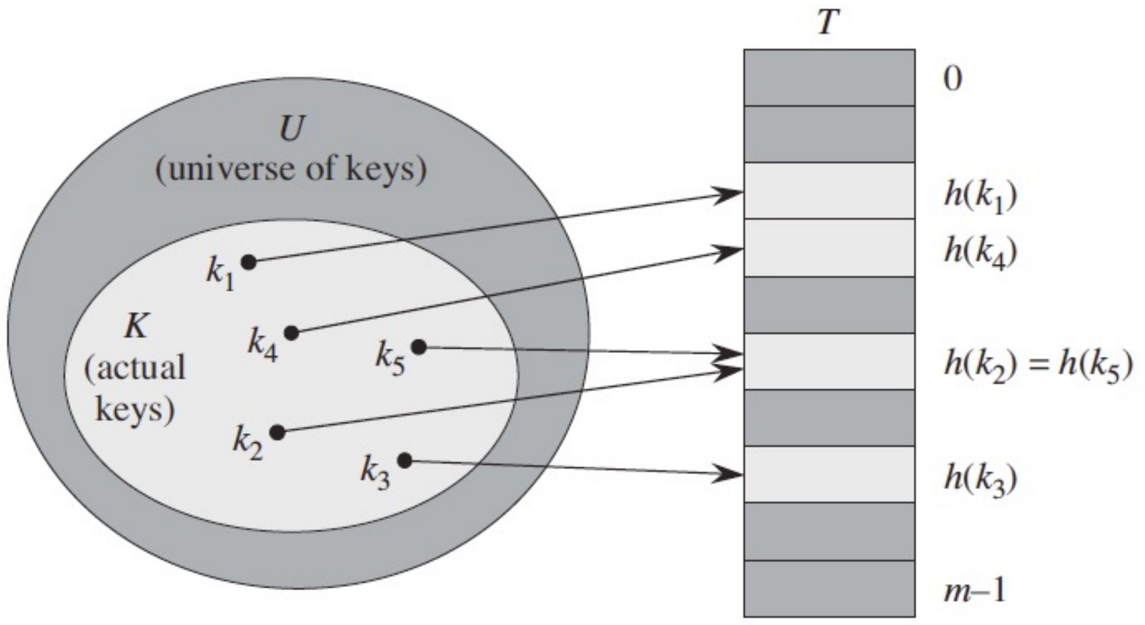
(From Algorithms & Data Structures Course)

A hash table implements an associative array (dictionary)

- Data is stored as a collection of **key-value** pairs

It uses a **hash function** to compute an offset into an array of buckets (slots)

- From which the desired value can be found



collision

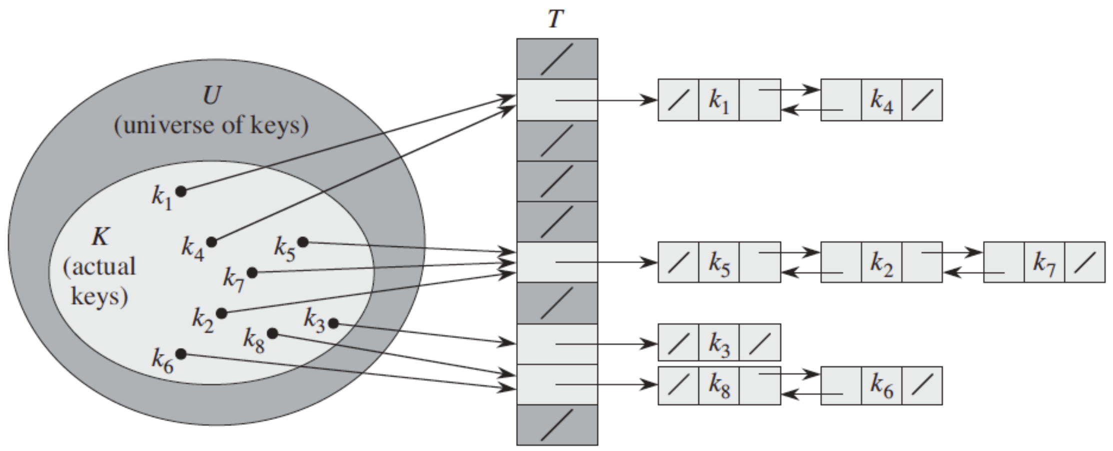
Source: Introduction to Algorithms, 3rd edition

Collision Resolution

By chaining

Link together entries hashed to the same value

Long chains can degrade search performance



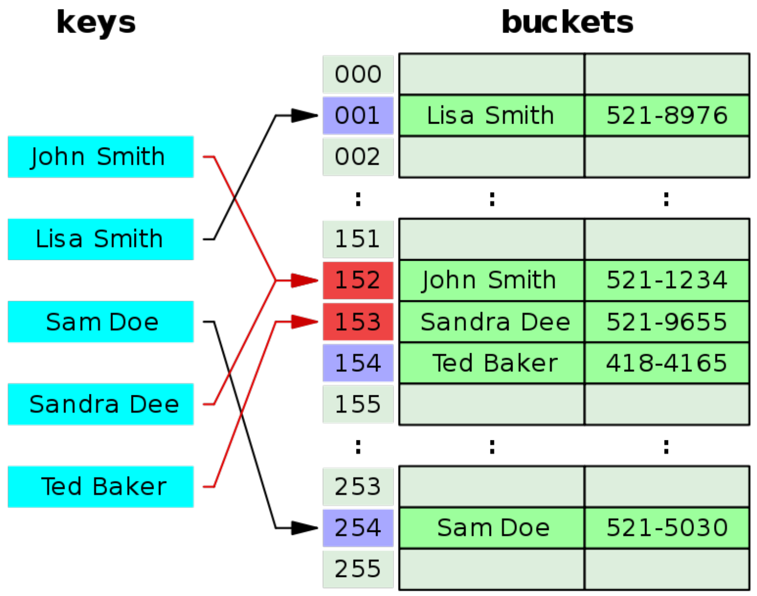
Source: Introduction to Algorithms, 3rd edition

Open Addressing

Single giant table of slots

Hash to slot, then probe until a free slot is found

Variants: Linear Probing, Cuckoo, Robin Hood, …



Source: https://en.wikipedia.org/wiki/Hash_table

Hashing in Databases

We want to be able to group together tuples with the same key value

Partition the data with hash function(s) applied on the key

- All tuples with a certain key will be in the same partition

Useful for:

- Removing duplicates (all duplicates will be grouped together)
- Grouping data (for GROUP BY)
- Looking up data using hash indexes

Hash-Based Indexing

Suitable for **equality-based predicates**

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

Cannot support range queries

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

Dynamic hashing schemes

- Extendible hashing
- Linear hashing

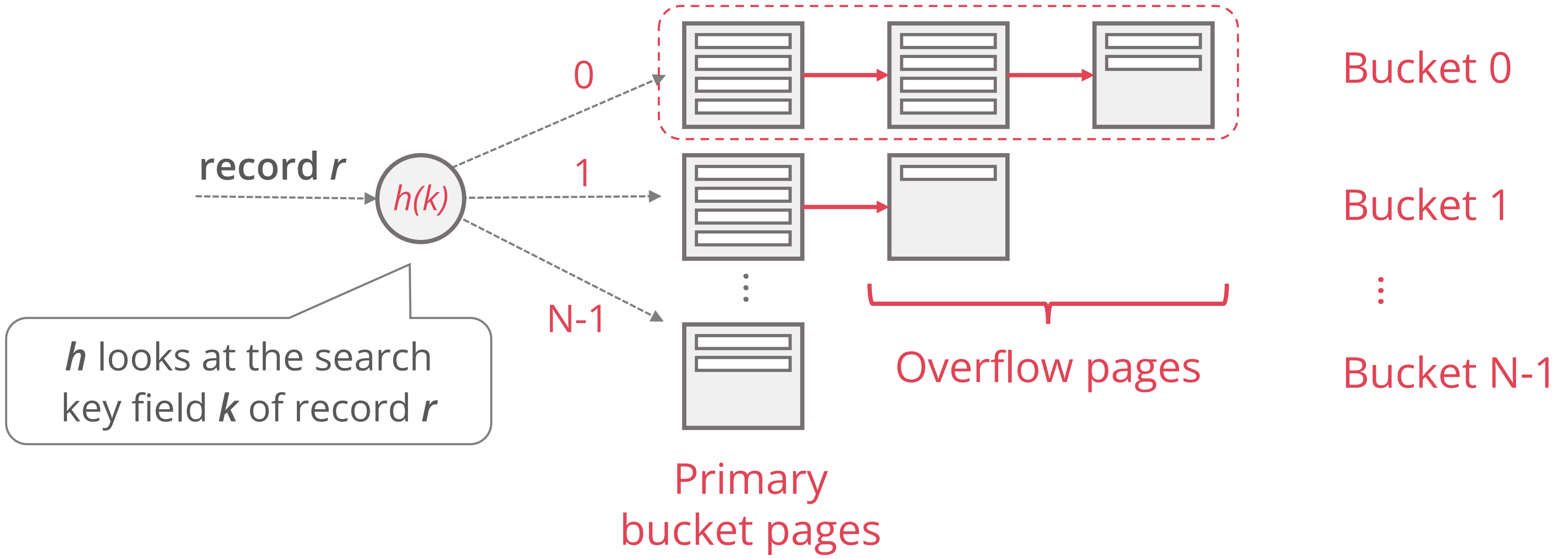
Static Chained Hashing

- 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 Chained Hash Table

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

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



Static Chained Hash Table Management

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

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

If overflow chain access is avoidable

- search requires a single I/O operation
- insert and delete require two I/O operations

Hash Collisions and Overflow Chains

Hash collisions are unavoidable

- For search keys k and k', can happen $h(k) = h(k')$
- Search keys may not be unique (e.g., student age)
- Even if unique, the search key space is much larger than # of buckets
  - Having as many primary bucket pages as different search keys in database ⇒ waste of space

Long overflow chains can degrade performance

- Operation costs become non-uniform and unpredictable for a query optimiser
- 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)

Hash Functions

How to map a large key space into a smaller domain

- Real distributions of search key values are often non-uniform (skewed)

Trade-off between being fast vs. collision rate

- We want a lightweight (non-cryptographic) hash function with a low collision rate

Simple hash function: ***h(k)*** **= k mod N**

- Guarantees the range of ***h(k)*** to be **[0,N-1]**
- Choosing **N** = 2^d for some **d** effectively considers the least **d** bits of **k** only
- Prime numbers work best for **N**

Better hash functions used in practice

- xxHash (+ benchmark), MurmurHash, Google CityHash, Google FarmHash, CLHash

Static Hashing and Dynamic Files

If the data file **grows**,

- the development of overflow chains spoils the index I/O behaviour (1–2 I/O operations)

If the data file **shrinks**,

- a significant fraction of primary buckets may be (almost) empty – a waste of space

We may **periodically rehash** the data file to restore the ideal situation (20% free space, no overflow chains)

- Expensive – the index not usable while rehashing is in progress

As for ISAM, static hashing has advantages with concurrent access

- Only need to lock one bucket page to store a new entry or extend the overflow chain

Extendible Hashing

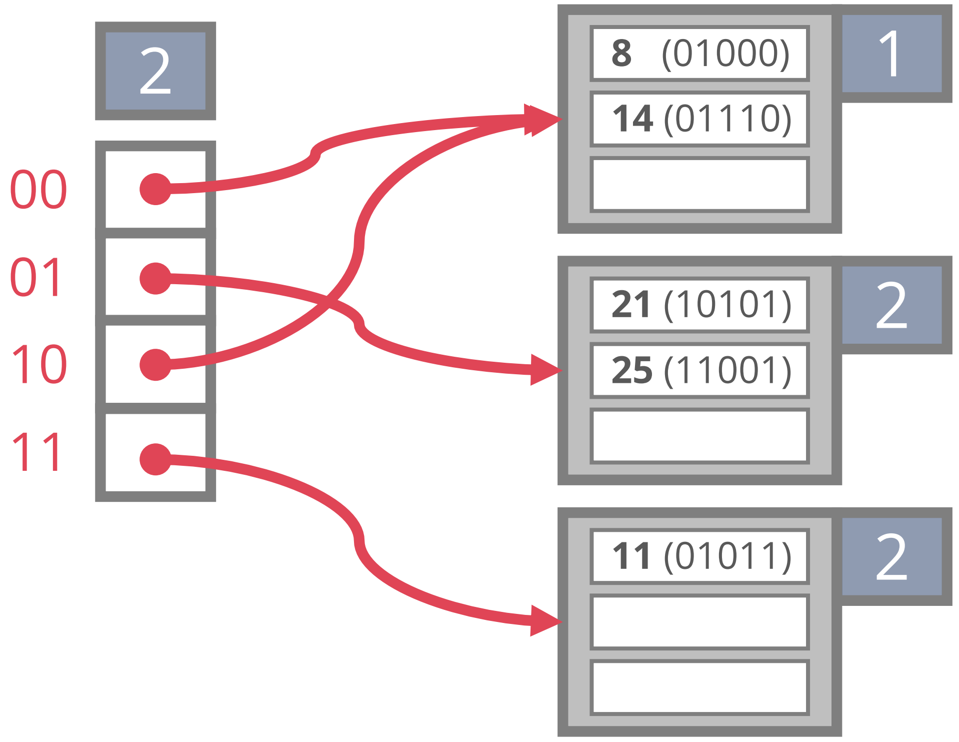
Situation: Bucket (primary page) is full and we want to insert. Why not reorganize the index by doubling # of buckets?

- Reading and writing all pages is expensive!

Idea: 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
- Only one page of data entries is split
- *No overflow pages!*

Extendible Hashing



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

Global and Local Depth

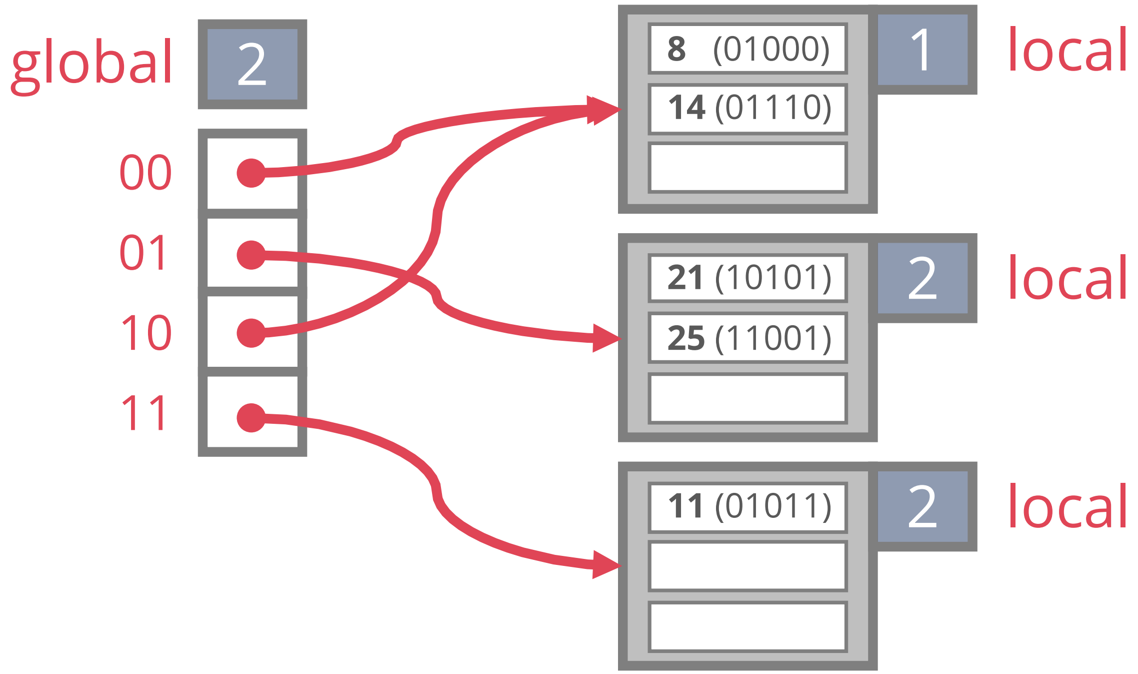
Global depth (*n* at directory)

Use the least *n* bits of ***h(k)*** to find a bucket pointer in the directory

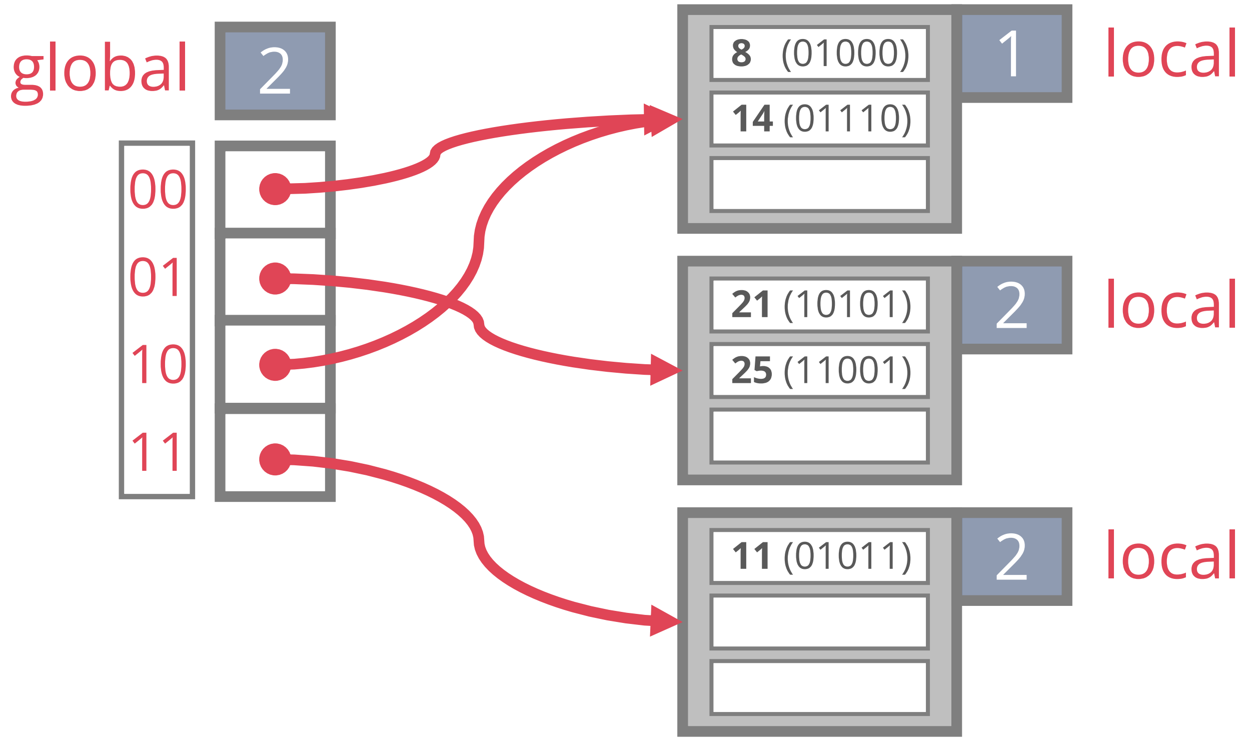
The directory size is 2^n

Local depth (*d* at individual buckets)

The hash values ***h(k)*** of all entries in this bucket agree on their least *d* bits



Extendible Hashing

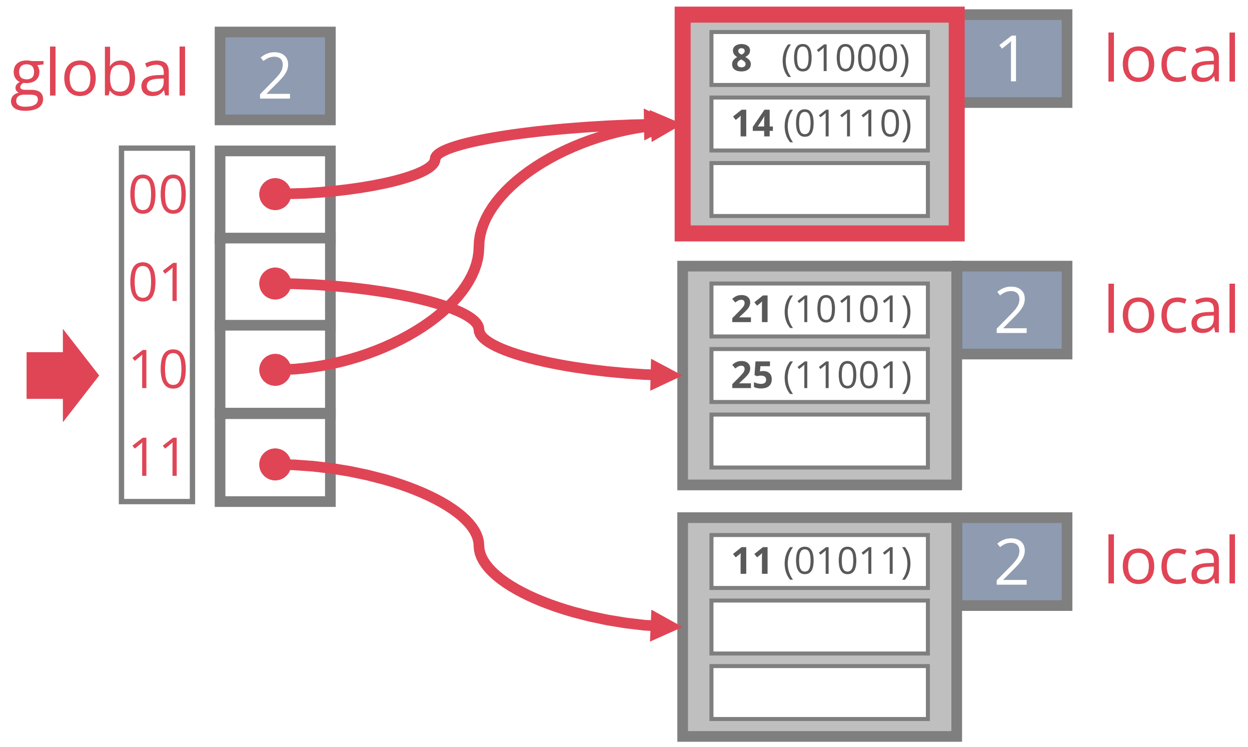


Find A

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

To find a bucket for A, take the least 2 bits of hash(A)

Extendible Hashing

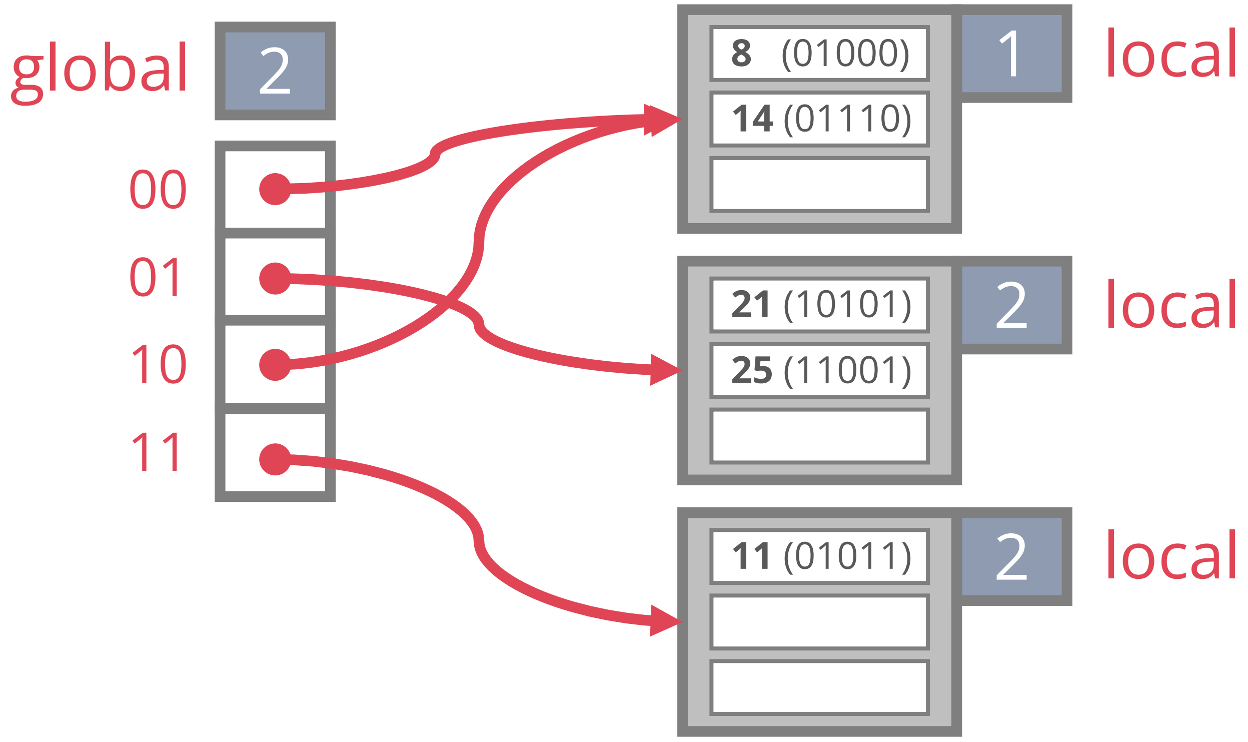


Find A

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

Check if the bucket contains key A. Need to compare keys due to collisions!

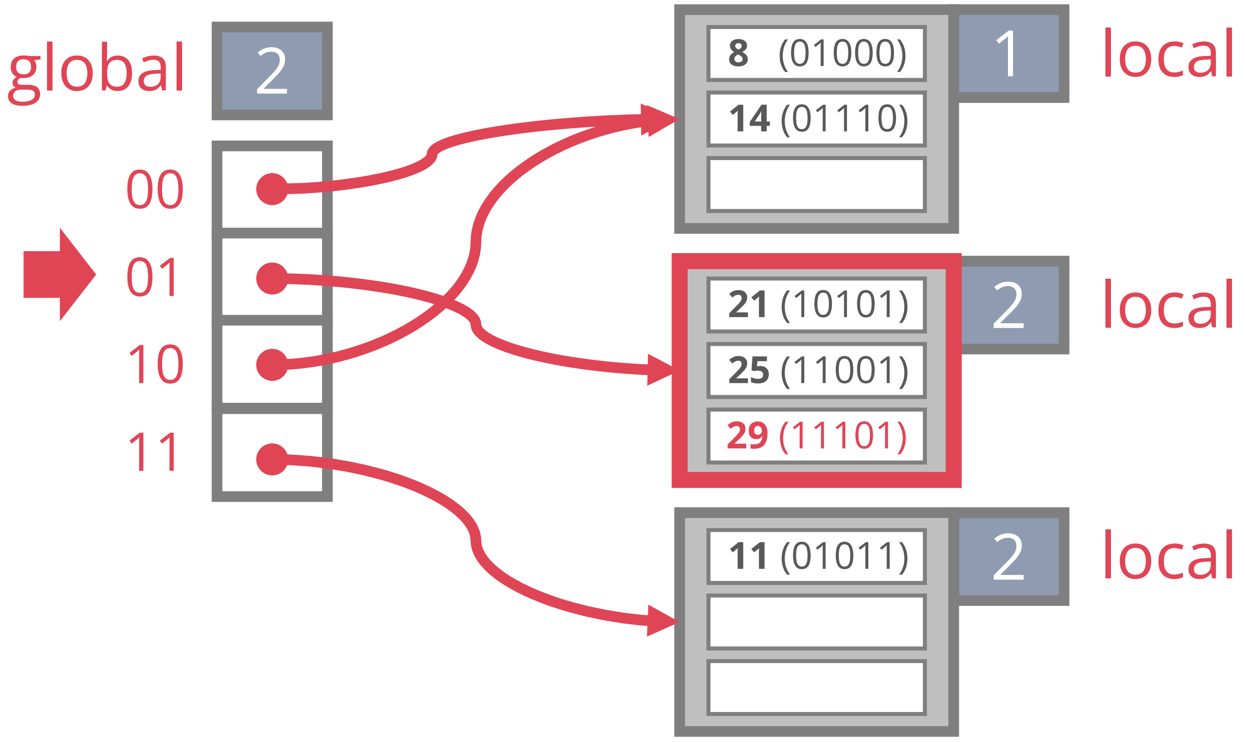
Extendible Hashing



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

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

Extendible Hashing

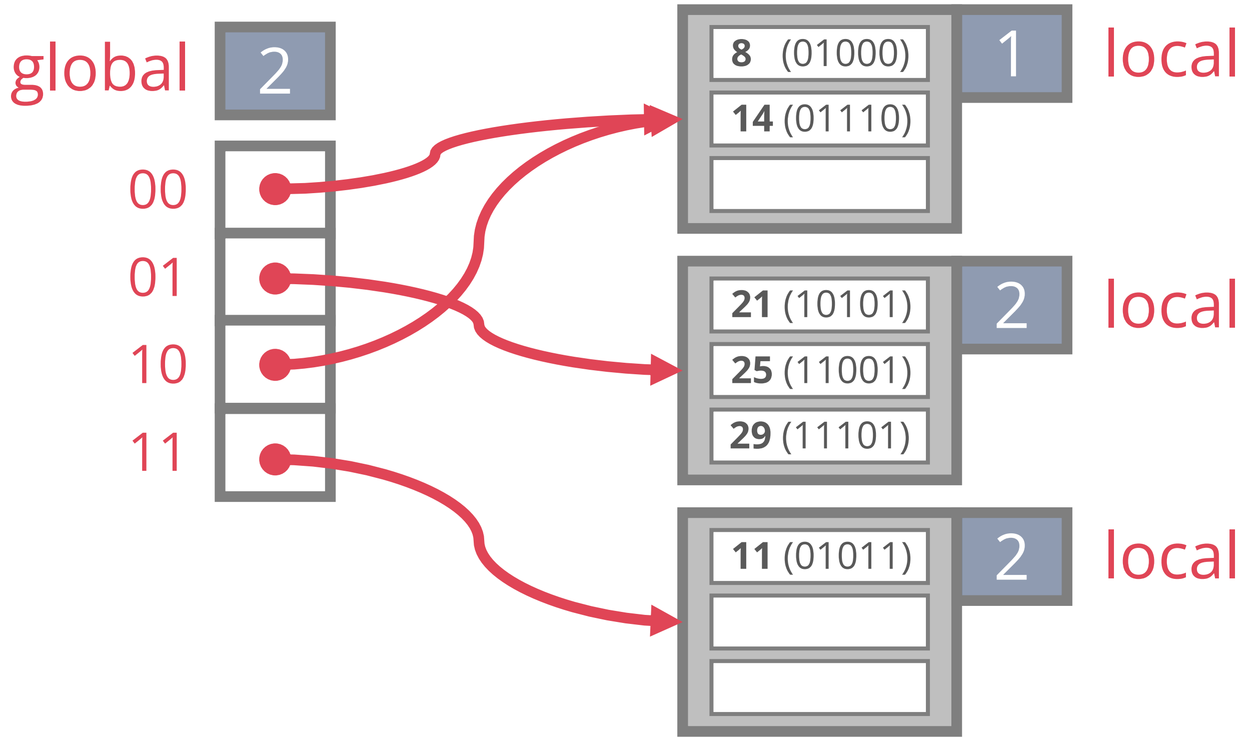


Find A
hash(A) = **14** = 01110_2

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

If the bucket still has capacity, store the index entry in it

Extendible Hashing

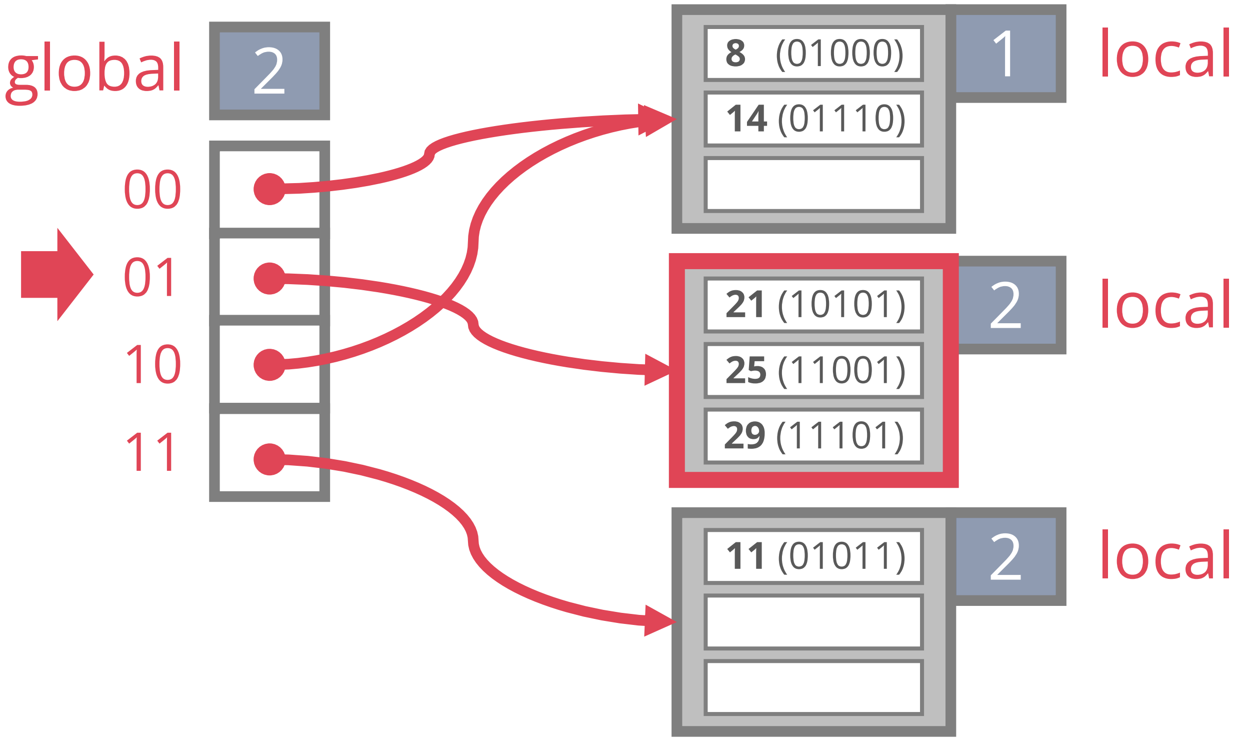


Find A
hash(A) = **14** = 01110_2

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

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

Extendible Hashing



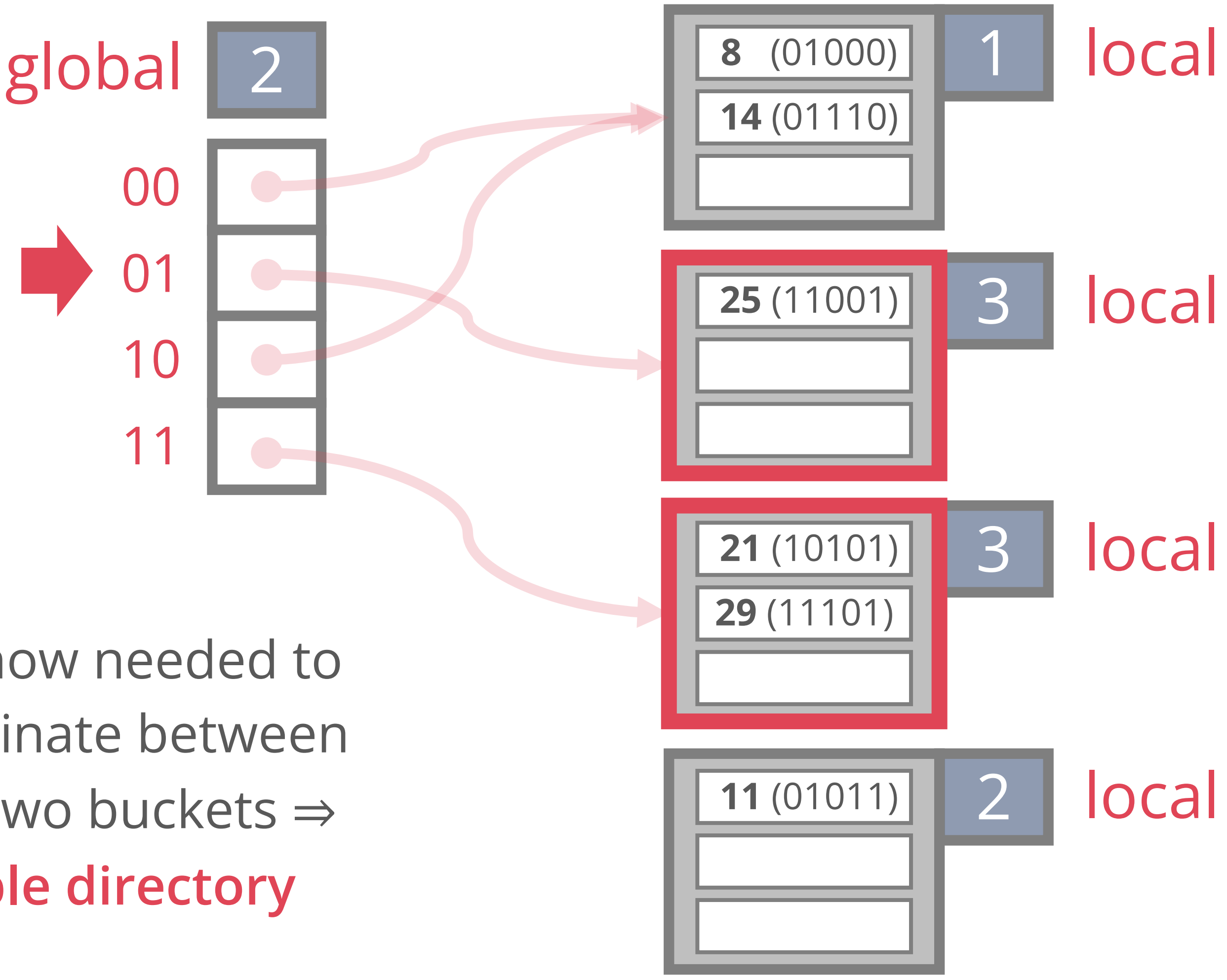
Find A
hash(A) = **14** = 01110_2

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

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

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

Extendible Hashing



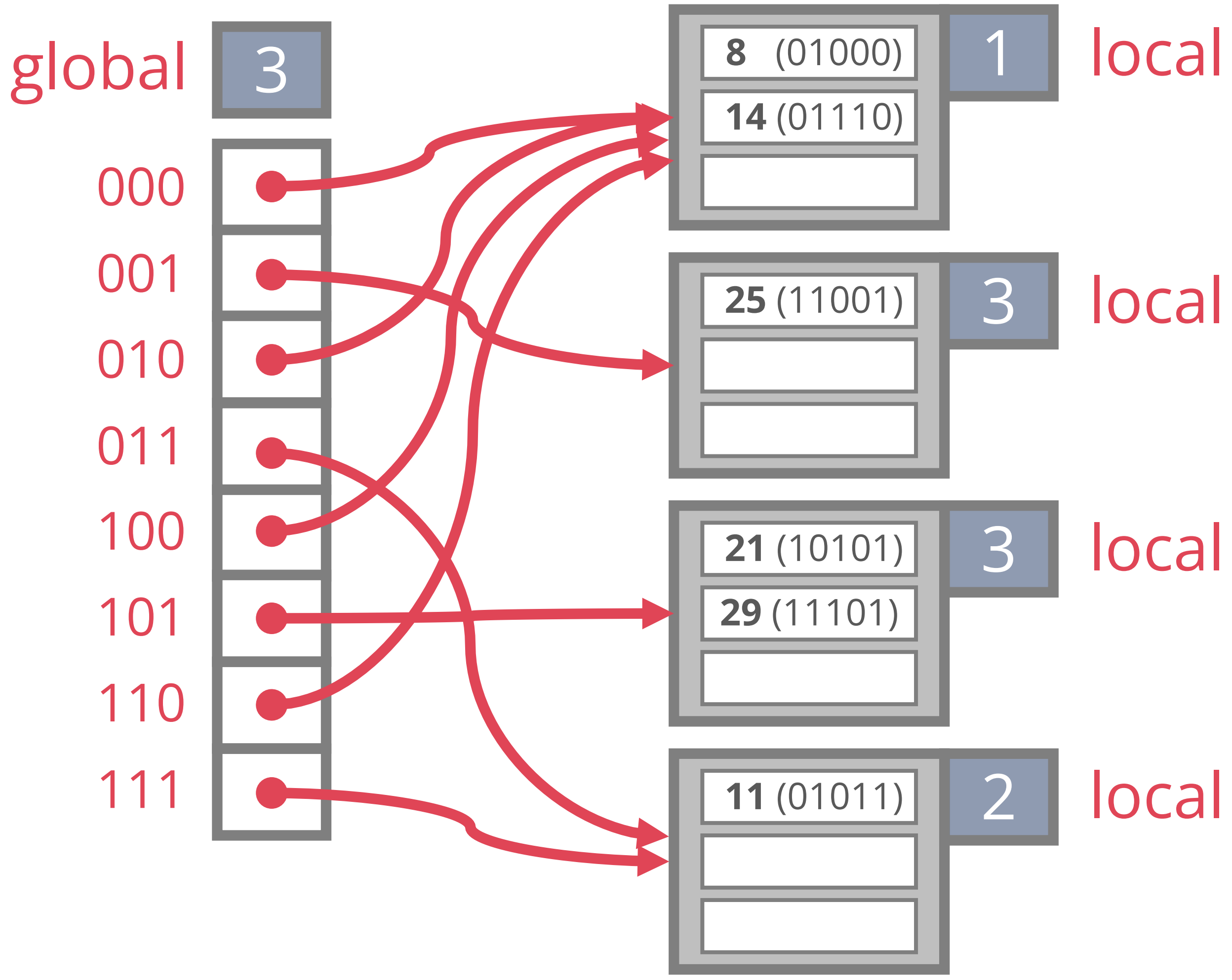
3 bits now needed to discriminate between these two buckets ⇒ **double directory**

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

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

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

Extendible Hashing

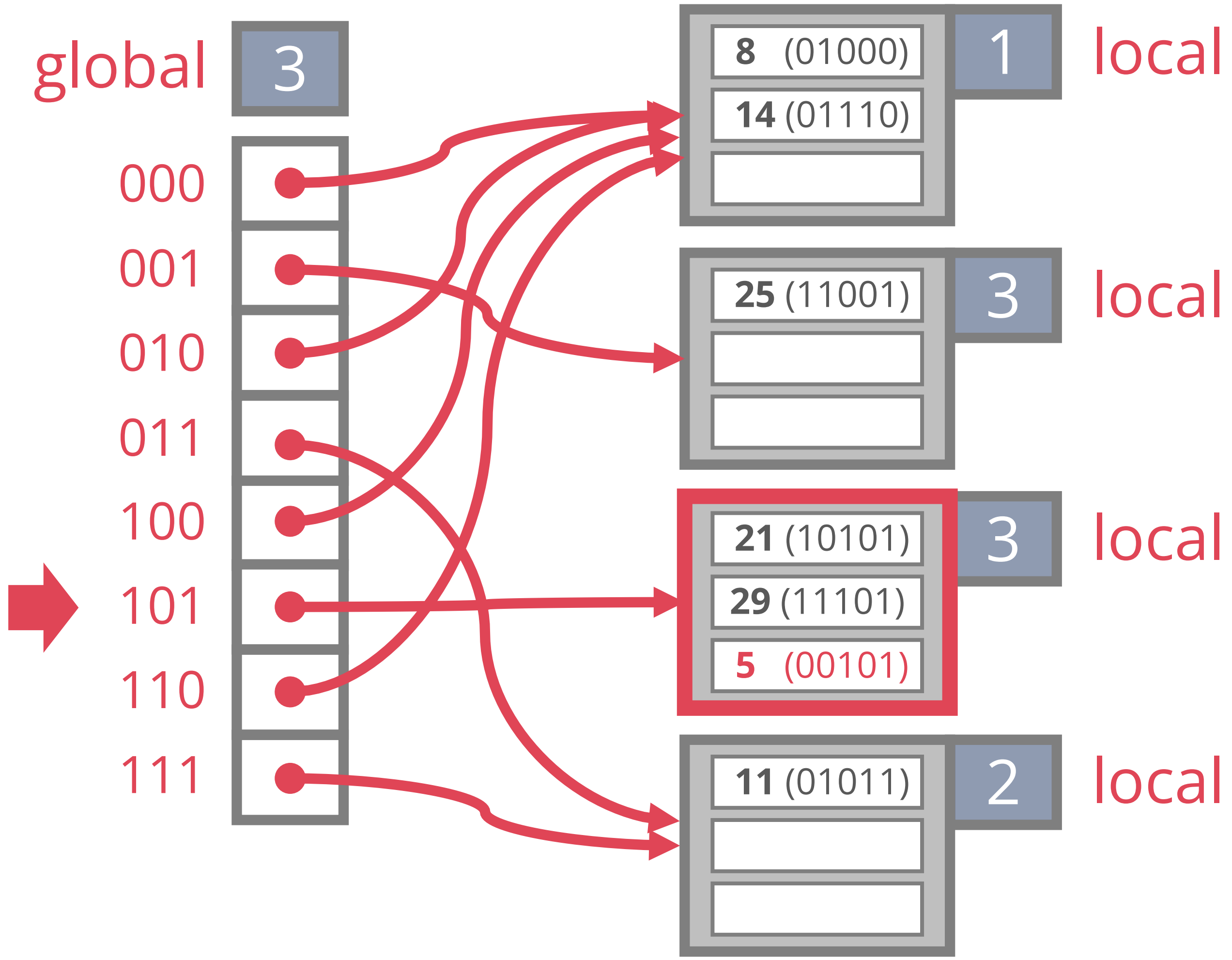


Find A
hash(A) = **14** = 01110_2

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

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

EXTENDIBLE HASHING



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

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

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

Directory Doubling

- Double directory by **copying** its original pointers and ”fixing” pointer to split bucket
  - Use of least significant bits enables efficient doubling via copying!
- 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
  - Modifying one or more bucket pointers in directory is sufficient
- Directory can also shrink when buckets become empty

Linear Hashing

Linear hashing adapts its data structure to record insertions and deletions

- Handles the problem of long overflow chains without using a directory

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

- The subscript is called the hash function’s level
- $\boldsymbol{h}_{\mathit{level+1}}$ doubles the range of $\boldsymbol{h}_{\mathit{level}}$

Split buckets in rounds

- One by one from the first to the last bucket
- In round *level*, use $\boldsymbol{h}_{\mathit{level}}$ for unsplit buckets and $\boldsymbol{h}_{\mathit{level+1}}$ for split buckets

Hash Function Family

Given an initial hash function ***h*** and an initial hash table with ***N*** buckets

Range of ***h*** is **<u>not</u>** 0 to N - 1

Define a family of hash functions $\boldsymbol{h}_0, \boldsymbol{h}_1, \boldsymbol{h}_2, \ldots$

$\boldsymbol{h}_{level}(k) = \boldsymbol{h}(k) \bmod (2^{level} \cdot N)$ $(level = 0, 1, 2, \ldots)$

Example:

Initial hash function $\boldsymbol{h}(k) = k$

N = 4 initial buckets

$\boldsymbol{h}_0(k) = k \bmod 4$ $\boldsymbol{h}_1(k) = k \bmod 8$ $\boldsymbol{h}_2(k) = k \bmod 16$ …

Linear Hashing

Maintains a **pointer** that tracks the next bucket to split

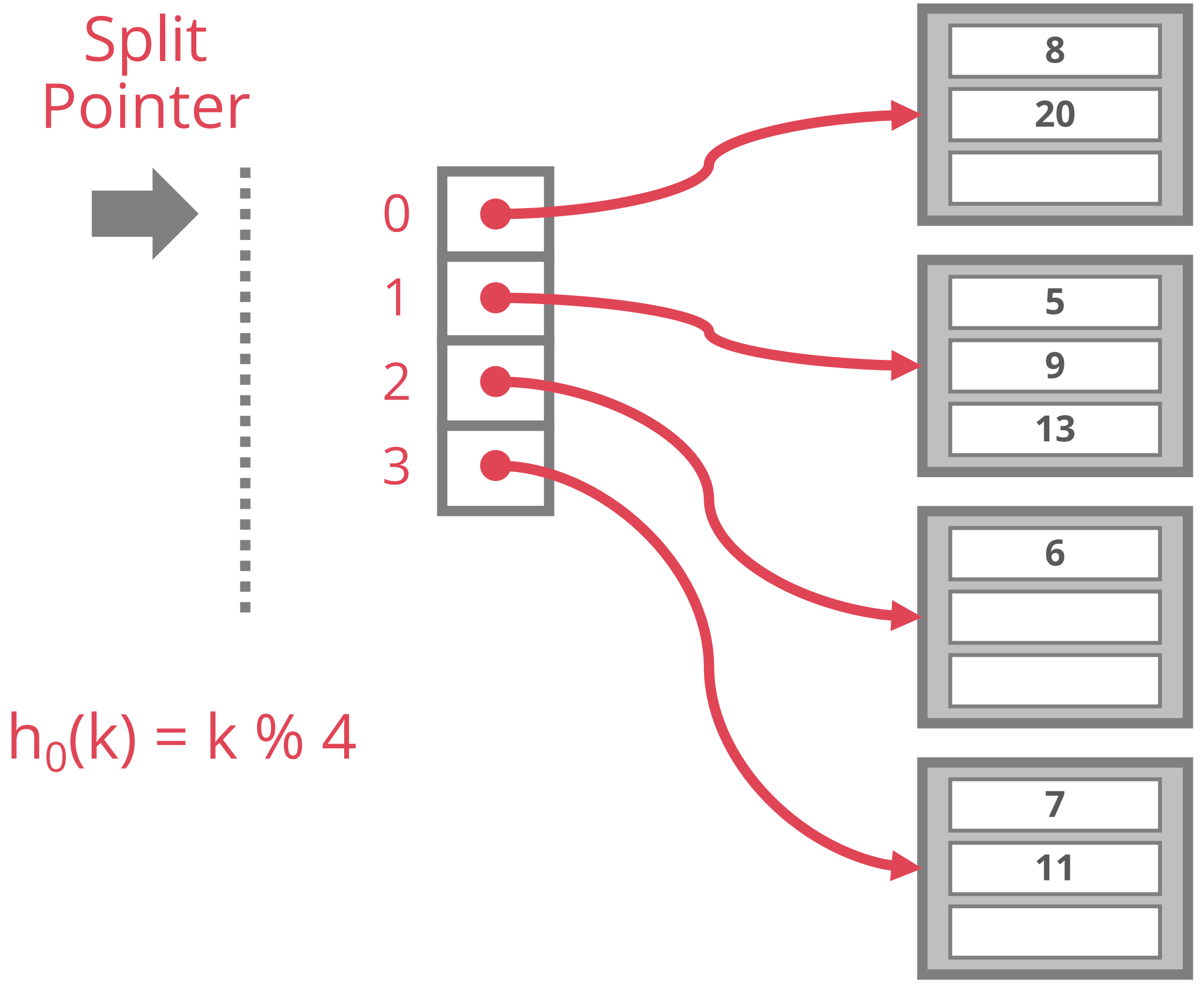
When any bucket overflows, split the bucket at the pointer location

- This may not be the bucket that triggered the split!

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



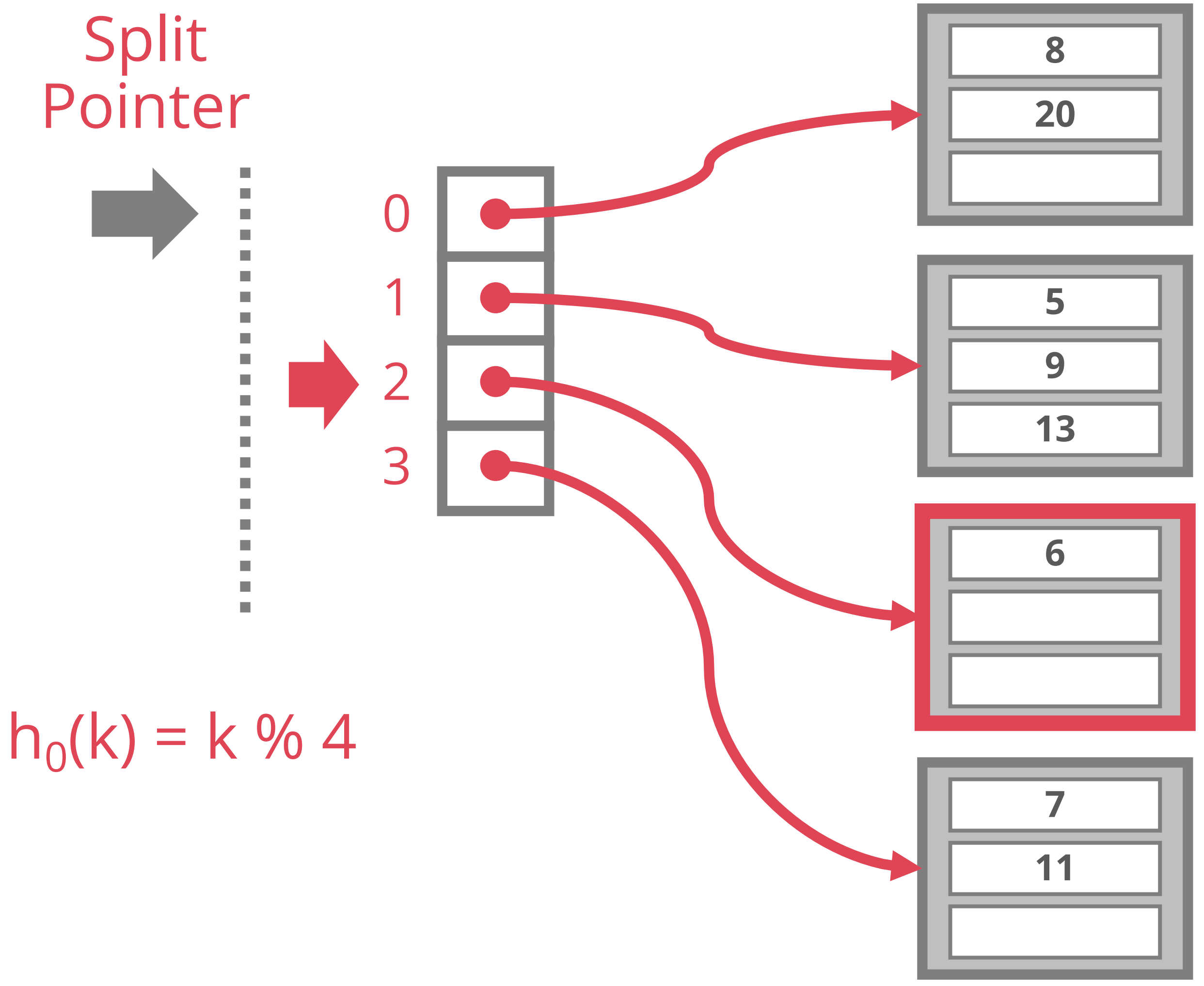
$h_0(k) = k \% 4$

Level = 0

Use h_0 for all buckets

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

Linear Hashing

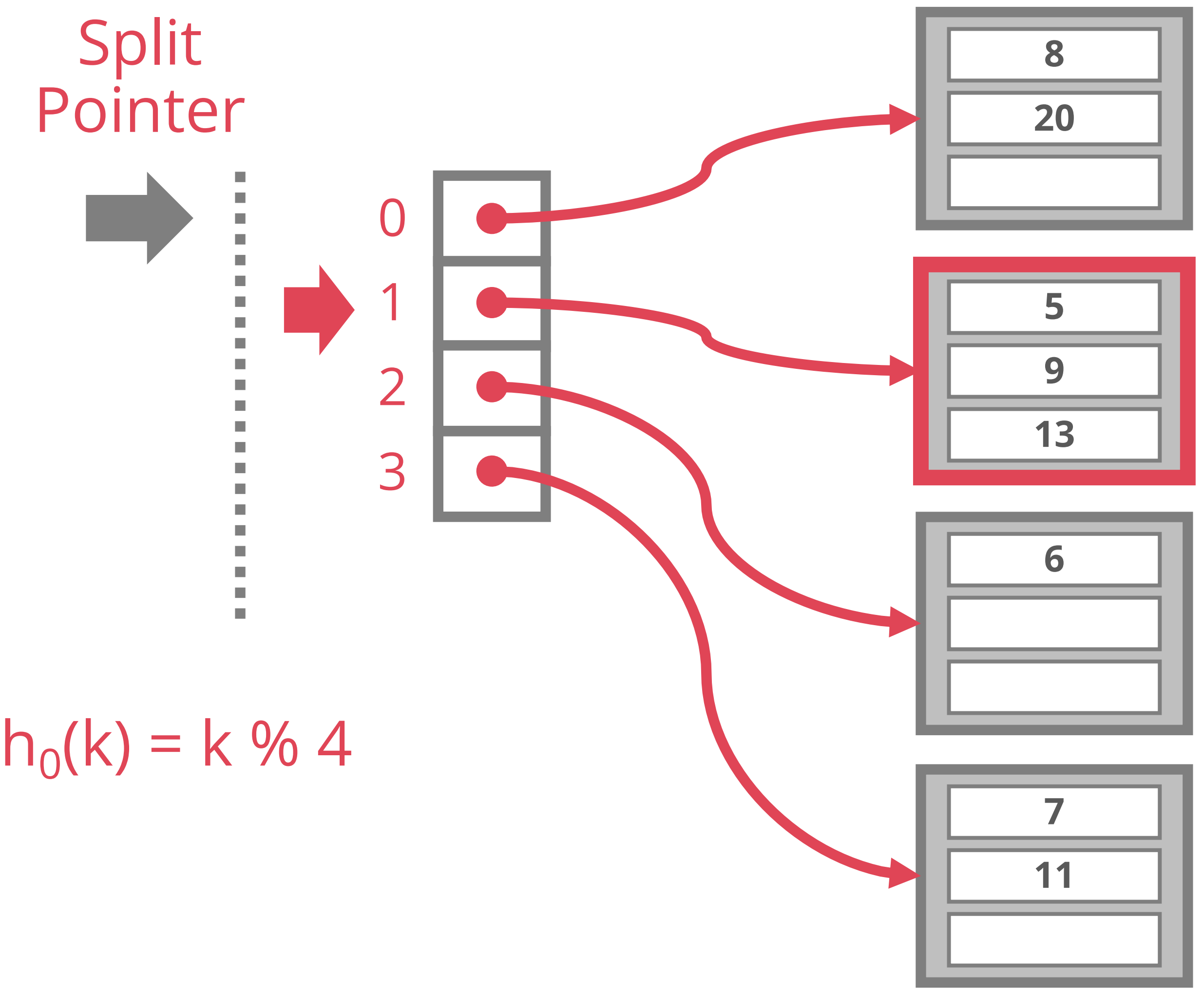


$h_0(k) = k \% 4$

Find 6

$h_0(6) = 6 \% 4 = 2$

LINEAR HASHING



$h_0(k) = k \% 4$

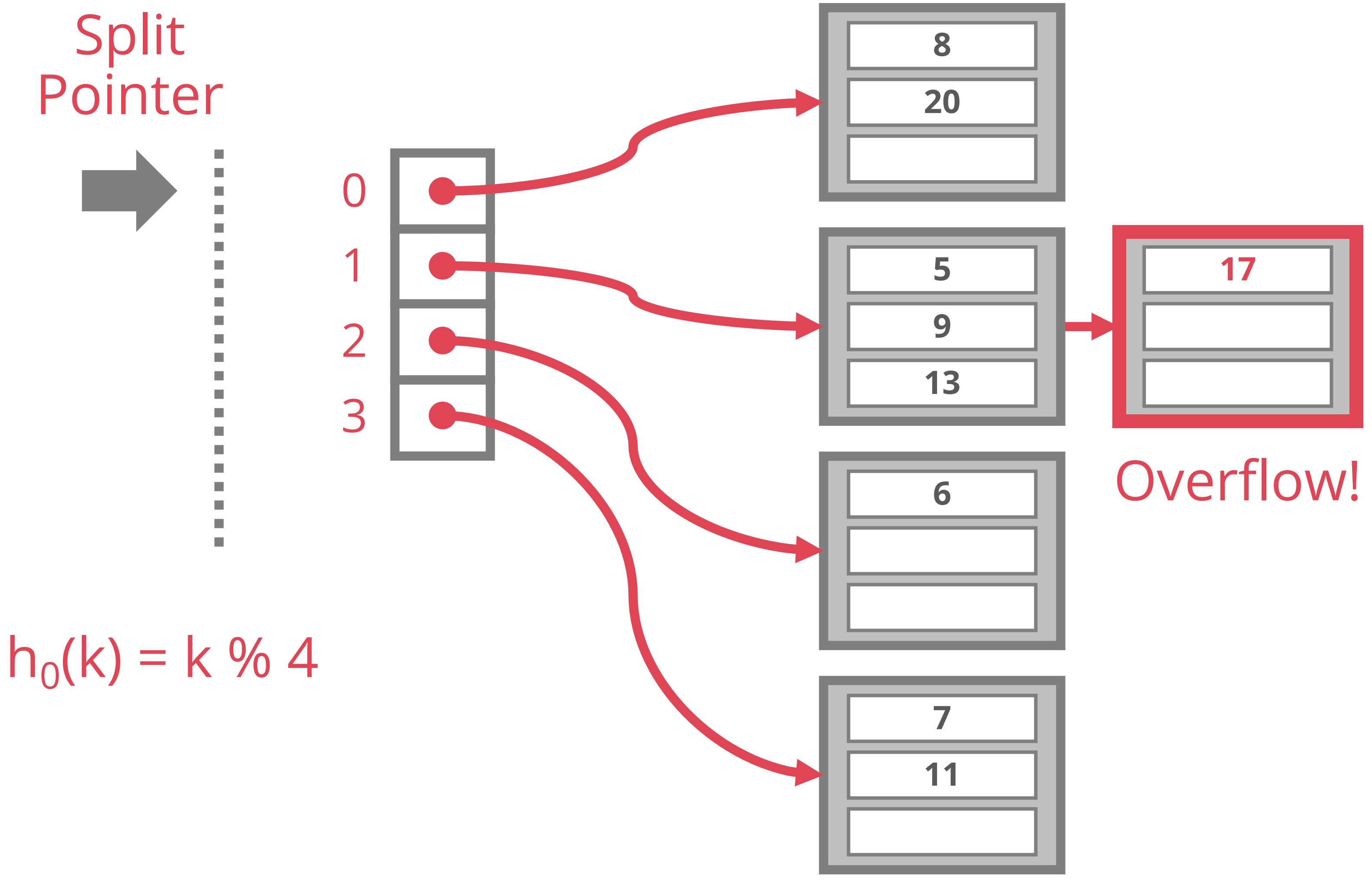
Find 6

$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

LINEAR HASHING



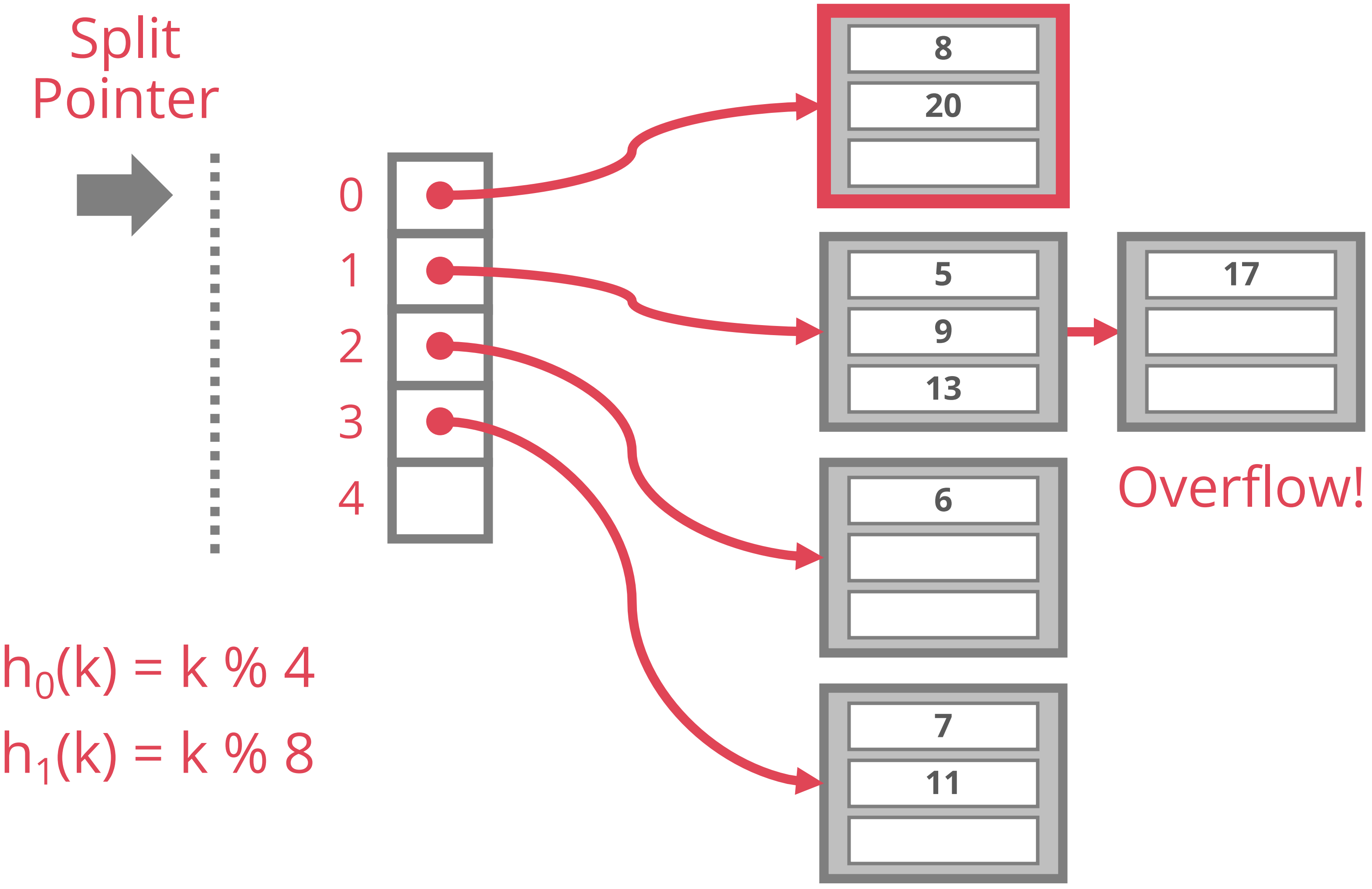
Find 6

$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

Linear Hashing



$h_0(k) = k \% 4$

$h_1(k) = k \% 8$

Find 6

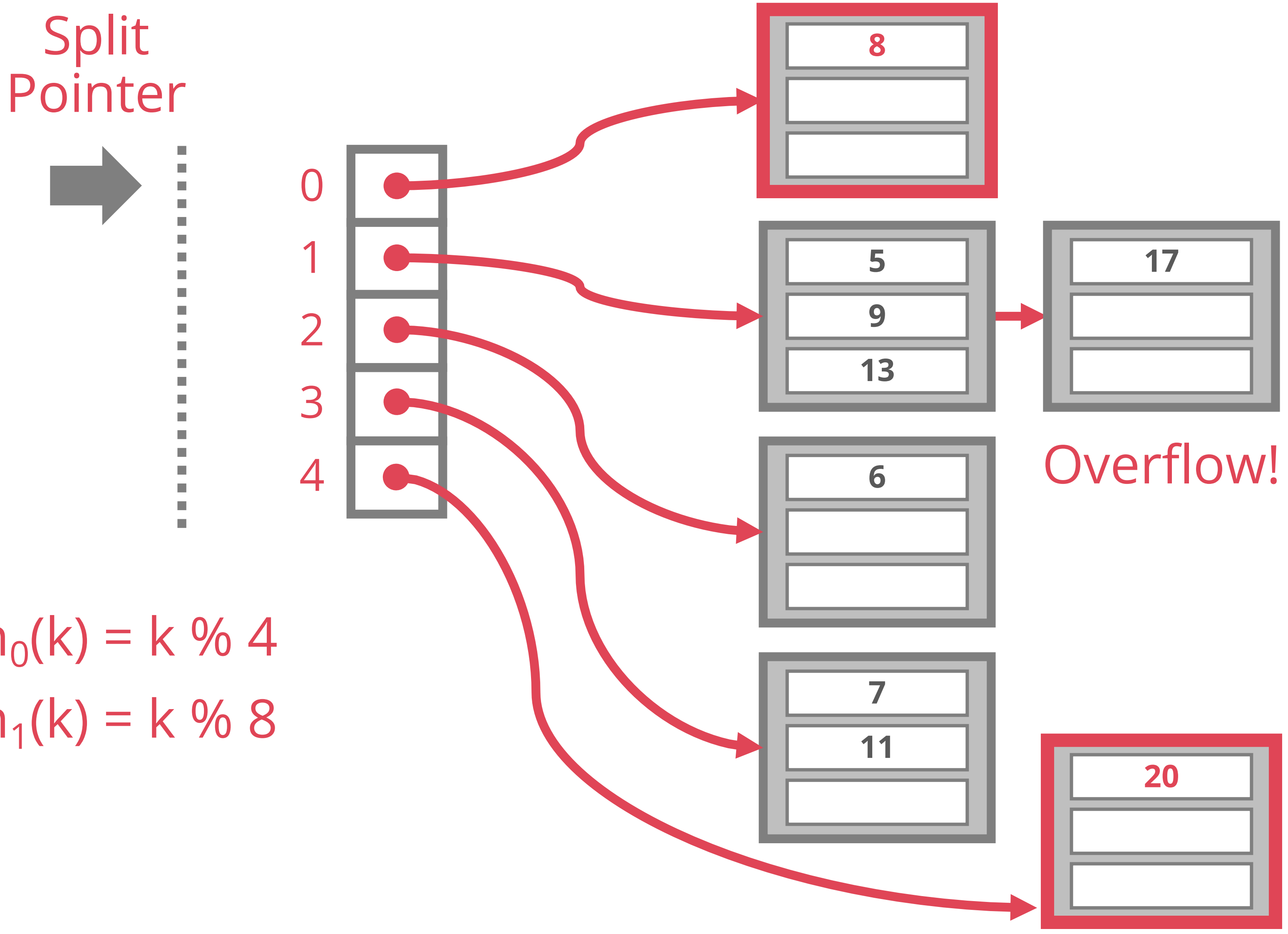
$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

Split bucket 0 using h_1

Linear Hashing



$h_0(k) = k \% 4$

$h_1(k) = k \% 8$

Find 6

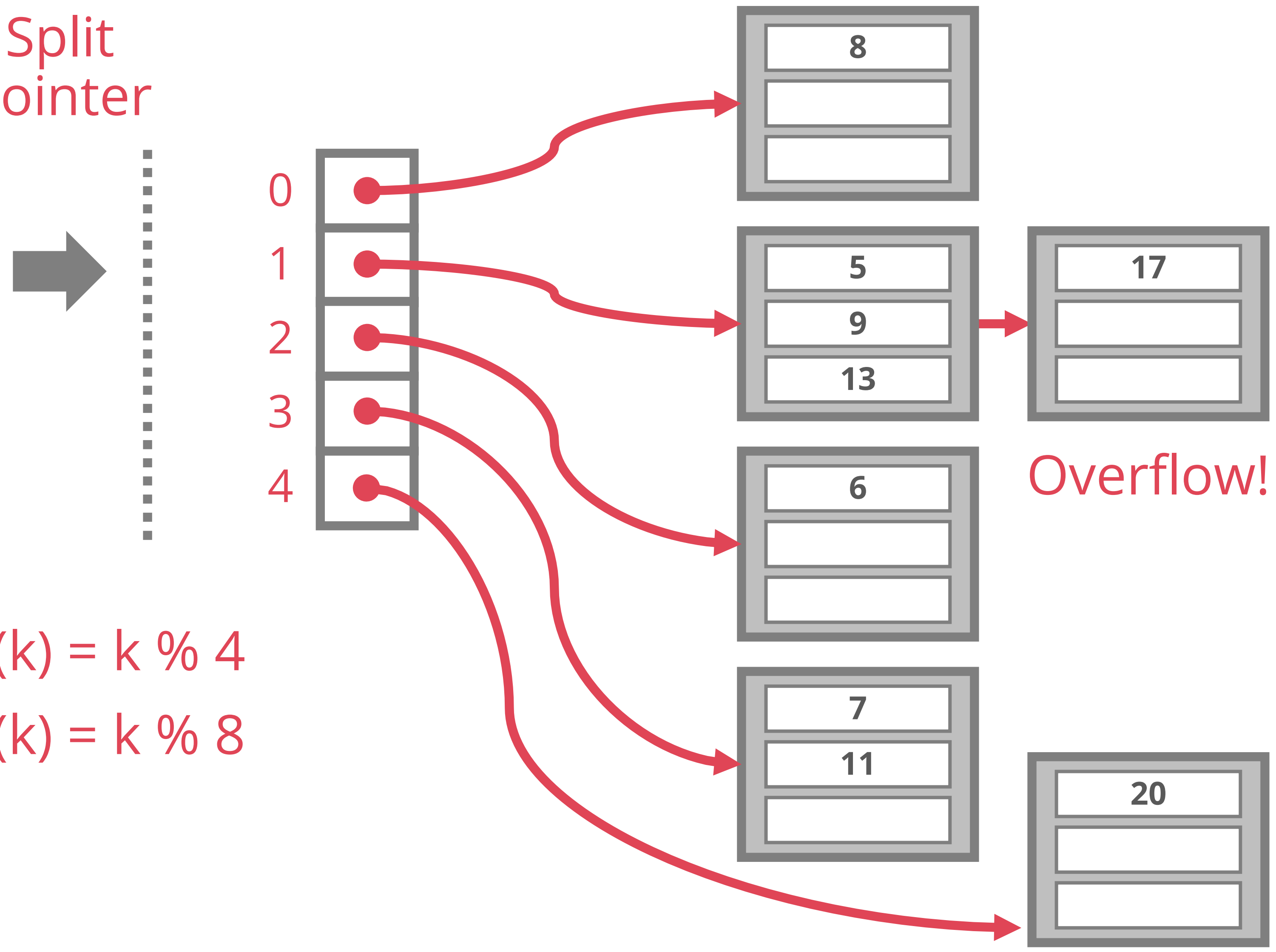
$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

Advance split pointer

Linear Hashing



$h_0(k) = k \% 4$

$h_1(k) = k \% 8$

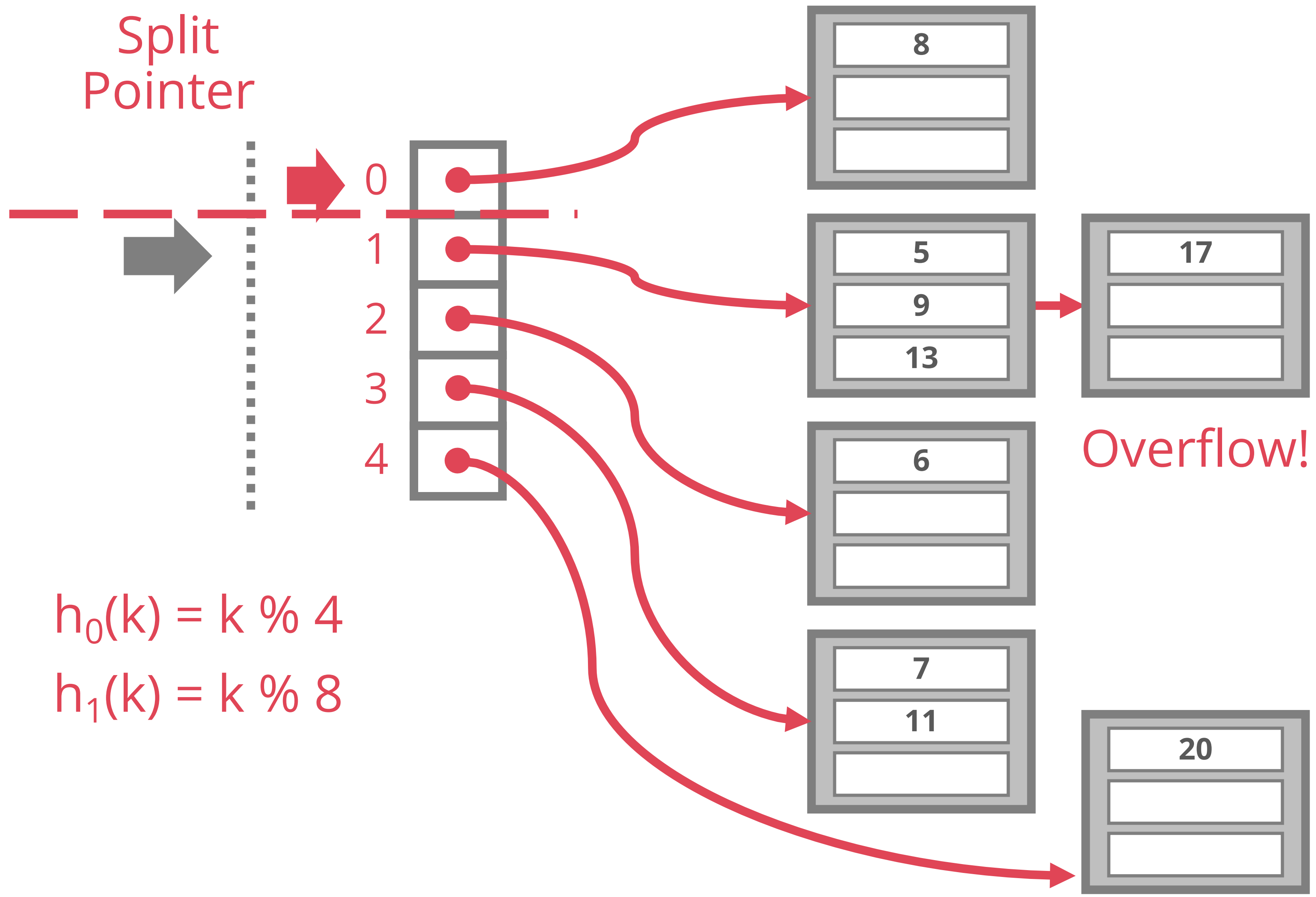
Find 6

$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

LINEAR HASHING



$h_0(k) = k \% 4$

$h_1(k) = k \% 8$

Find 6

$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

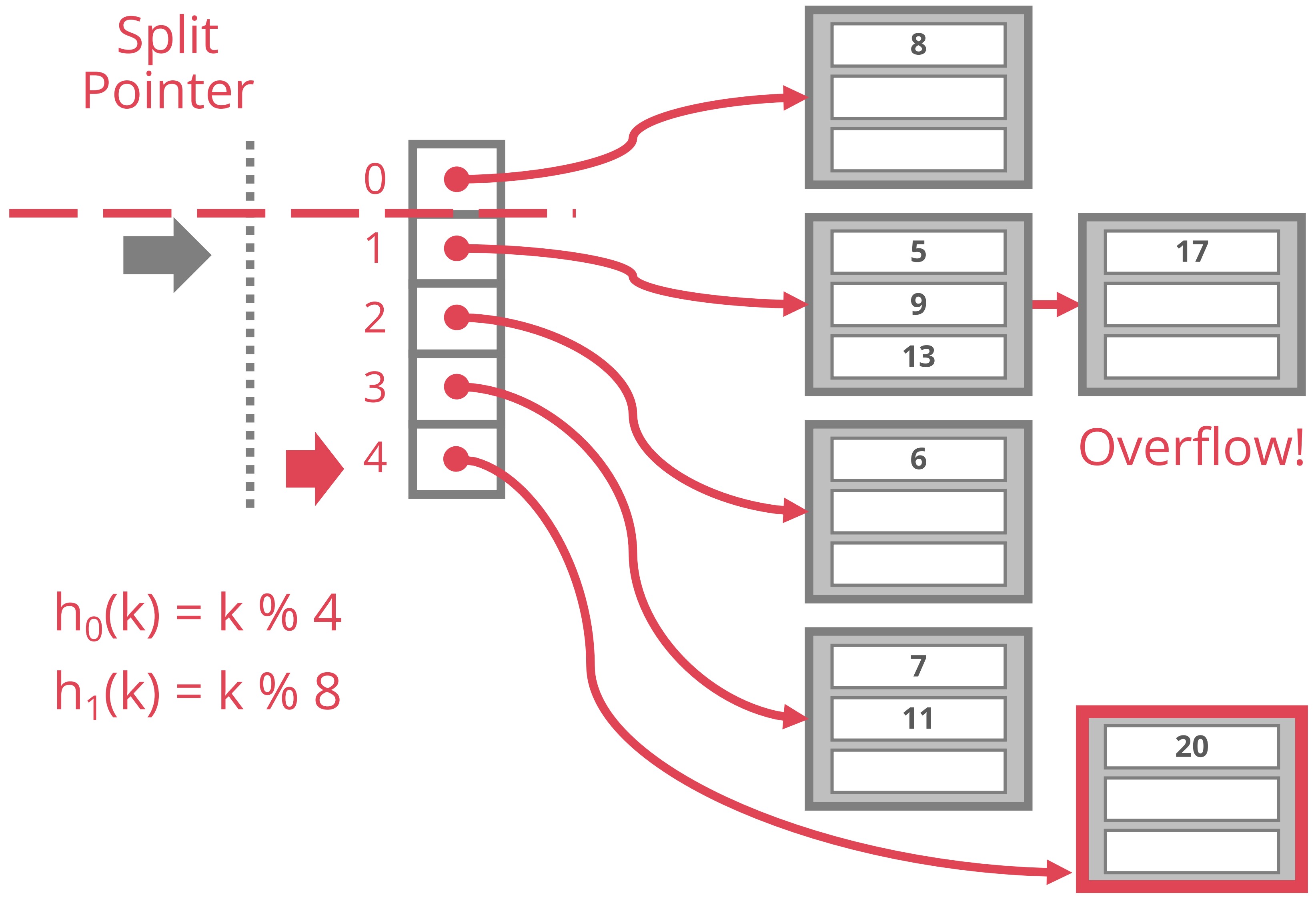
Find 20

$h_0(20) = 20 \% 4 = 0$

Bucket 0 is split (behind pointer)

⇒ use h_1

Linear Hashing



$h_0(k) = k \% 4$

$h_1(k) = k \% 8$

Find 6

$h_0(6) = 6 \% 4 = 2$

Insert 17

$h_0(17) = 17 \% 4 = 1$

Find 20

$h_0(20) = 20 \% 4 = 0$

$h_1(20) = 20 \% 8 = 4$

Linear Hashing

- Since buckets are split round-robin, **long overflow chains don't develop**!
  - After splitting the last bucket, start a new round: delete the first hash function, increase *level*, 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**!

Summary

Hash-based indexes

Best for equality searches, cannot support range searches

Static hashing

Can lead to long overflow chains

Extendible hashing

Avoids overflow chains by splitting a full bucket when a new entry is to be added to it

Linear hashing

Avoids directory by splitting buckets round-robin and using overflow pages