Universität Konstanz

Database System Architecture and Implementation

Module 4

Hash-Based Indexes

November 19, 2018

Orientation

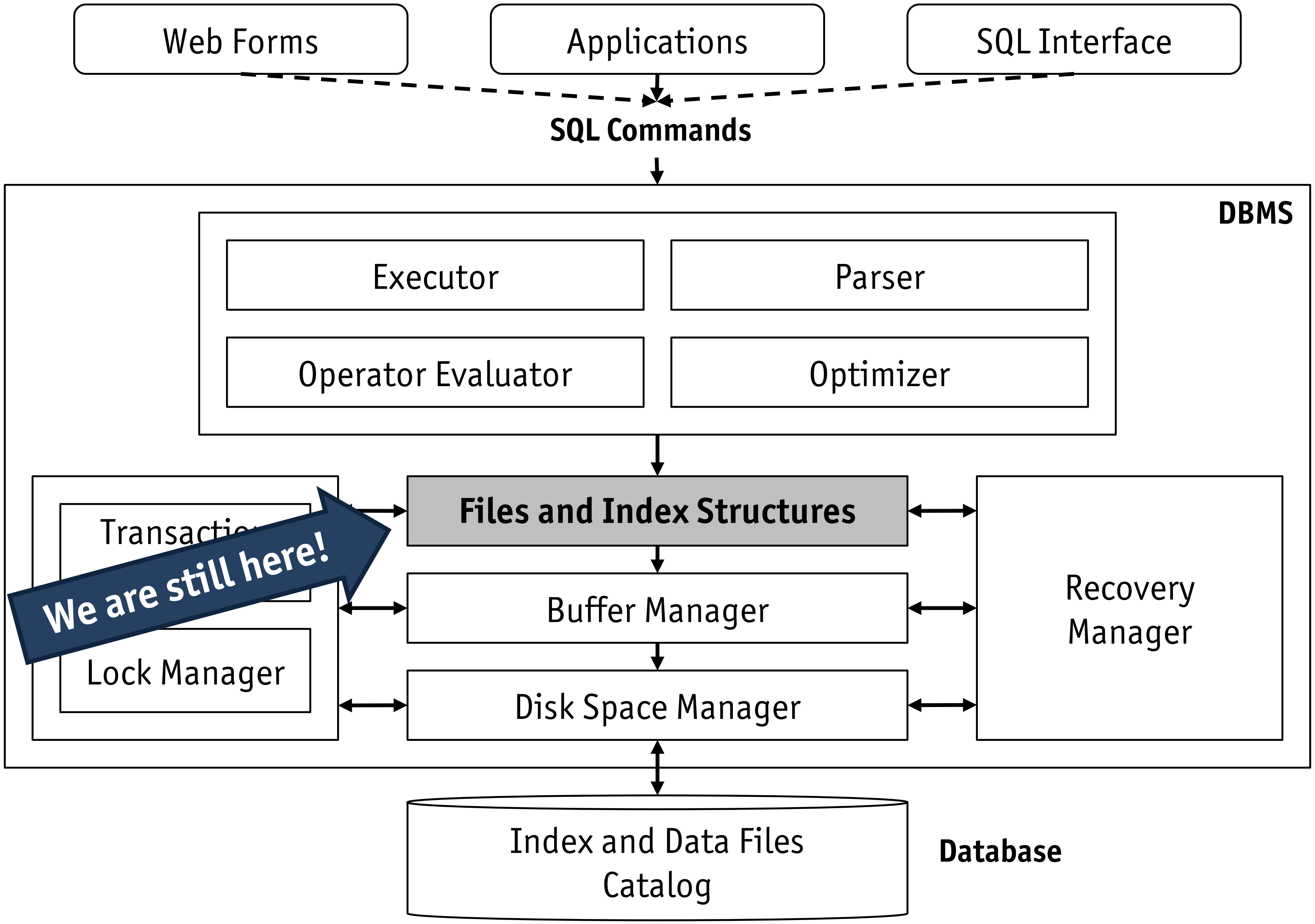
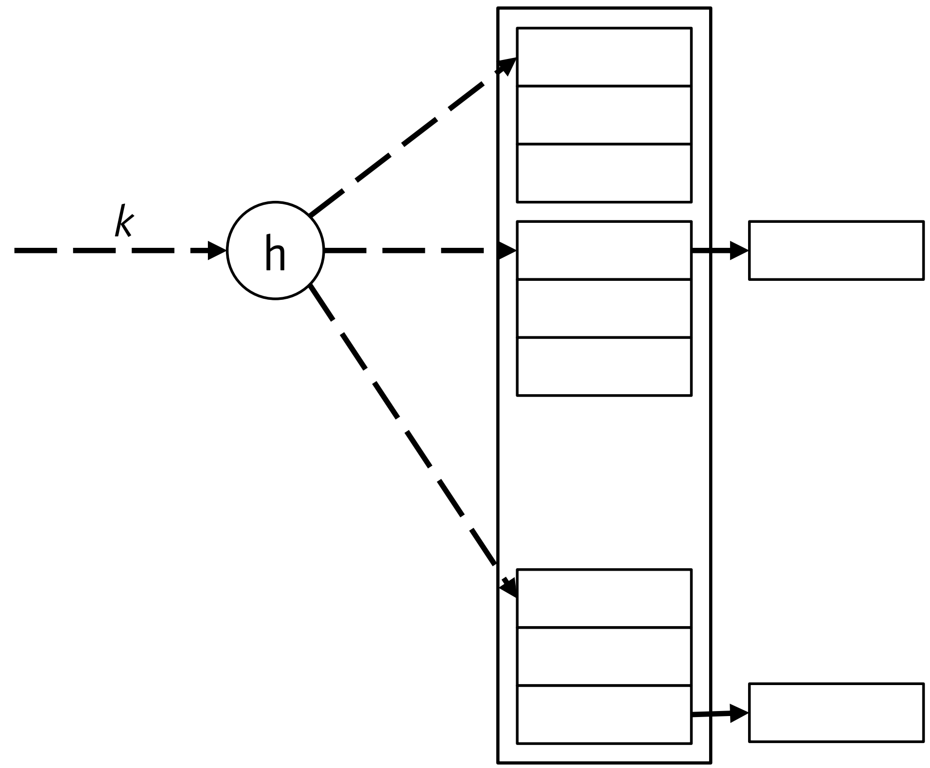


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Overview of hash-based indexing
- Static hashing
- Extendible hashing
- Linear hashing



Hash-Based Indexing

Equality selection

```
SELECT *
FROM   R
WHERE  A = k
```

- In addition to tree-structured indexes (B+ trees), typical DBMS also provide support for **hash-based index structures**
  - "unbeatable" when it comes to support **equality selections**
  - can answer equality such queries using a **single I/O operation** (more precisely 1.2 operations), if the hash index is carefully maintained while the underlying data file for relation **R** grows and shrinks
  - other query operations, like (equality joins) internally require **a large number of equality tests**
  - presence (or absence) of support for hash indexes can make a real difference in such scenarios

Hash Indexes vs. B+ Tree Indexes

- Locating a record with key k
  - B+ tree search **compares** k to other keys k' organized in a (tree-shaped) search data structure
  - hash indexes **use the bits of k itself** (independent of all other stored records and their keys) to find (i.e., **compute the address of**) the record
- Range queries
  - B+ trees handle range queries efficiently by leveraging the **sequence set**
  - hash indexes provide **no support for range queries** (hash indexes are also known as **scatter storage**)

Overview of Hash-Based Indexing

- Static hashing
  - used to illustrate **basic concepts** of hashing
  - much like ISAM, static hashing does **not** handle updates well
- Dynamic hashing
  - **extendible hashing** and **linear hashing**
  - refine the hashing principle and adapt well to record insertions and deletions
- Hashing granularity
  - in contrast to in-memory applications where record-oriented hashing prevails, DBMS typically use **bucket-oriented hashing**
  - a bucket **can contain several records** and may have an **overflow chain**
  - a bucket is a **(set of) page(s)** on secondary memory

Static Hashing

Build a static hash index on attribute `A`

1. Allocate a fixed area of N (successive) disk pages, the so-called **primary buckets**
2. In each bucket, install a pointer to a chain of **overflow pages**
   initially, set this pointer to `null`
3. Define a **hash function** *h* with *range* [0, …, N – 1], the *domain* of *h* is the type of `A`, e.g.,

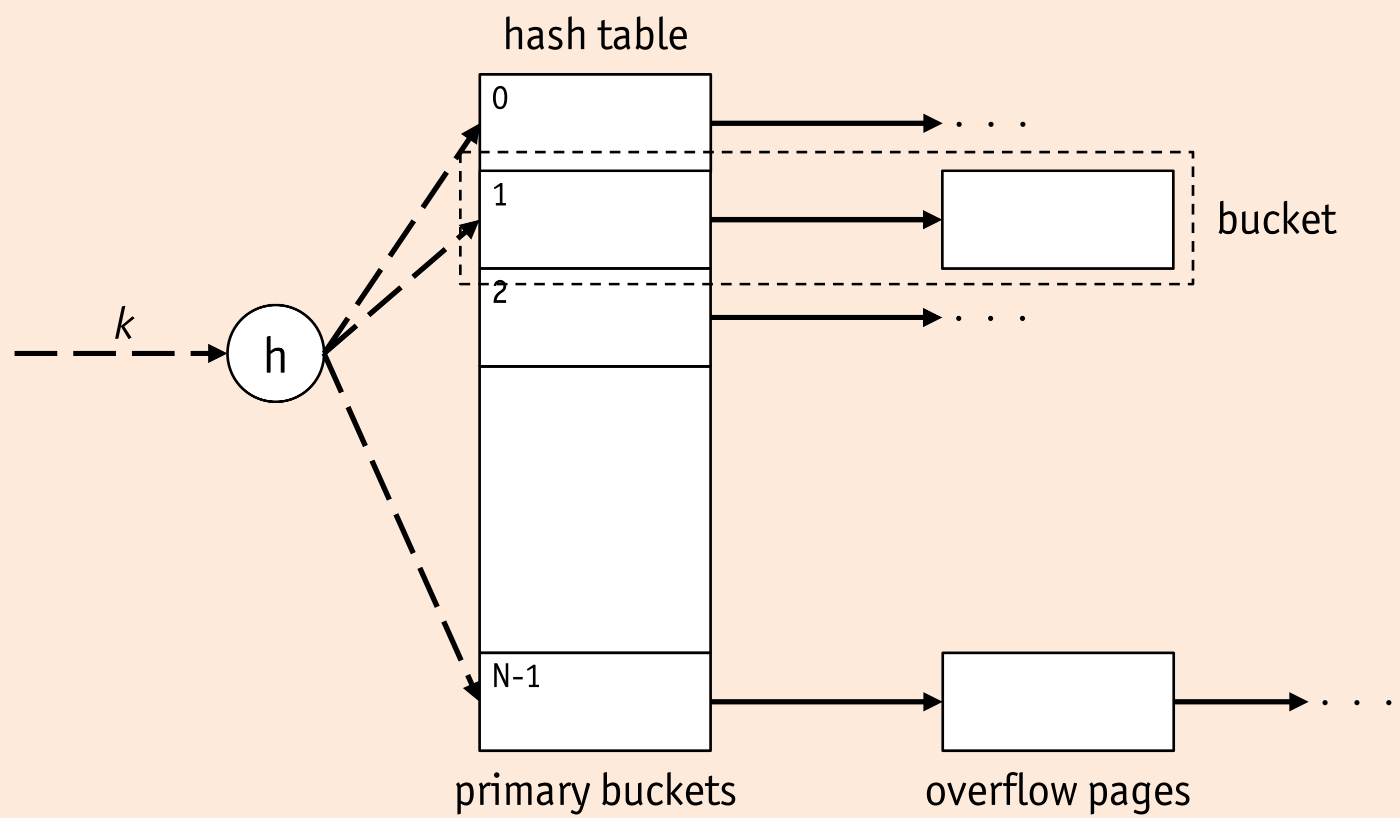
   $$h : \texttt{INTEGER} \rightarrow [0, \ldots, N - 1]$$

   if A has the SQL type `INTEGER`

- Evaluating the hash function *h* on a given data value is **cheap:** it only involves a few CPU instructions

Static Hashing

Static hash table



- A primary bucket and its chain of overflow pages is referred to as a **bucket**
- Each bucket contains index entries $k*$, which can be implemented using any of the variants ❶, ❷, and ❸

Static Hashing

- Operations **`hsearch`**(*k*), **`hinsert`**(*k*), and **`hdelete`**(*k*) for a record with key **`A`** = *k* depend on the **hashing scheme**

Static hashing scheme

1. **apply hash function** *h* to key value, i.e., compute $h(k)$
2. **access primary bucket page** with number $h(k)$
3. **search, insert, or delete** the record with key *k* on that page or, if necessary, **access the overflow chain** of bucket $h(k)$

- If the hashing scheme works well and overflow chain access can be avoided altogether
  - **`hsearch`**(*k*) requires a **single I/O operation**
  - **`hinsert`**(*k*) and **`hdelete`**(*k*) require **two I/O operations**

Collisions and Overflow Chains

- At least for static hashing, **overflow chain management** is important
  - generally, we do **not** want hash function *h* to avoid collisions, i.e.,

$$h(k) = h(k') \quad \text{even if} \quad k \neq k'$$

    (otherwise as many primary buckets as different keys in the data file or even in **A**'s domain would be required)
  - however, it is important that *h* **scatters** the domain of **A evenly across** [1, …, N – 1] in order to avoid long overflow chains for few buckets
  - otherwise, the I/O behavior of the hash table becomes non-uniform and unpredictable for a query optimizer
  - unfortunately, such "good" hash functions are hard to discover

Probability of Collisions

The birthday paradox

Consider the people in a group as the **domain** and use their birthday as **hash function** *h* (i.e., $h : Person \mapsto [0, \ldots, 364]$)

> *If the group has 23 or more members, chances are 50% that two people share the same birthday* **(collision)**

Check for yourself

1. Compute the probability that n people all have different birthdays

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ P(n-1) \times \dfrac{365 - (n-1)}{365} & if\ n > 1 \end{cases}$$

2. Try to find "birthday mates" at the next larger party

Hash Functions

- If key values were purely **random**, a “good” hash function could simply extract a few bits and use them as a hash value
  - key value distributions found in databases are **not** random
  - it is **impossible** to generate truly random hash values from non-random key values
- But is it possible to define hash functions that scatter even better than a random function?
- Fairly good hash functions can be found rather easily by
  - **division** of the key value
  - **multiplication** of the key value

Hash Functions

Design of a hash function

1. **By division**: simply define

$$\boldsymbol{h(k) = k \bmod N}$$

   - this guarantees that range of $h(k)$ to be $[0, \dots, N-1]$
   - **prime numbers** work best for N
   - choosing $N = 2^d$ for some d effectively considers the least d bits of k only

2. **By multiplication:** extract the fractional part of $Z \cdot k$ (for a specific Z) and multiply by hash table size N

$$\boldsymbol{h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor}$$

   - the (inverse) **golden ratio** $Z = {}^{2}/_{\sqrt{5}+1} \approx 0.6180339887$ is a good choice (according to D. E. Knuth, *"Sorting and Searching"*)
   - for $Z = {}^{\dot{Z}}/_{2^w}$ and $N = 2^d$ (w is the number of bits in a CPU word), we simply have $h(k) = msb_d(\dot{Z} \cdot k)$, where $msb_d(x)$ denotes the d **most significant bits** of x (e.g., $msb_3(42) = 5$)

Static Hashing and Dynamic Files

- Effects of dynamic files on static hashing
  - if the underlying **data file grows**, developing overflow chains spoil the otherwise predictable I/O behavior (1-2 I/O operations)
  - if the underlying **data file shrinks**, a significant fraction of primary hash buckets may be (almost) empty and waste space
  - in the worst case, a hash table can **degrade into a linear list** (one long chain of overflow buckets)
- As in the case of ISAM case, static hashing has **advantages** when it comes to concurrent access
  - allocating a hash table of size 125% of the expected data capacity, i.e., only 80% full, will typically give good results
  - data file could be rehashed periodically to restore this ideal situation (**expensive** operation and the index **cannot be used** during rehashing)

Dynamic Hashing

- Dynamic hashing scheme have been devised to overcome these limitations of static hashing by
  - **combining** the use of hash functions with directories that guide the way to the data records (e.g., extendible hashing)
  - **adapting** the hash function (e.g., linear hashing)

Curb your enthusiasm!

Stand-alone hash indexes are **very rare**!

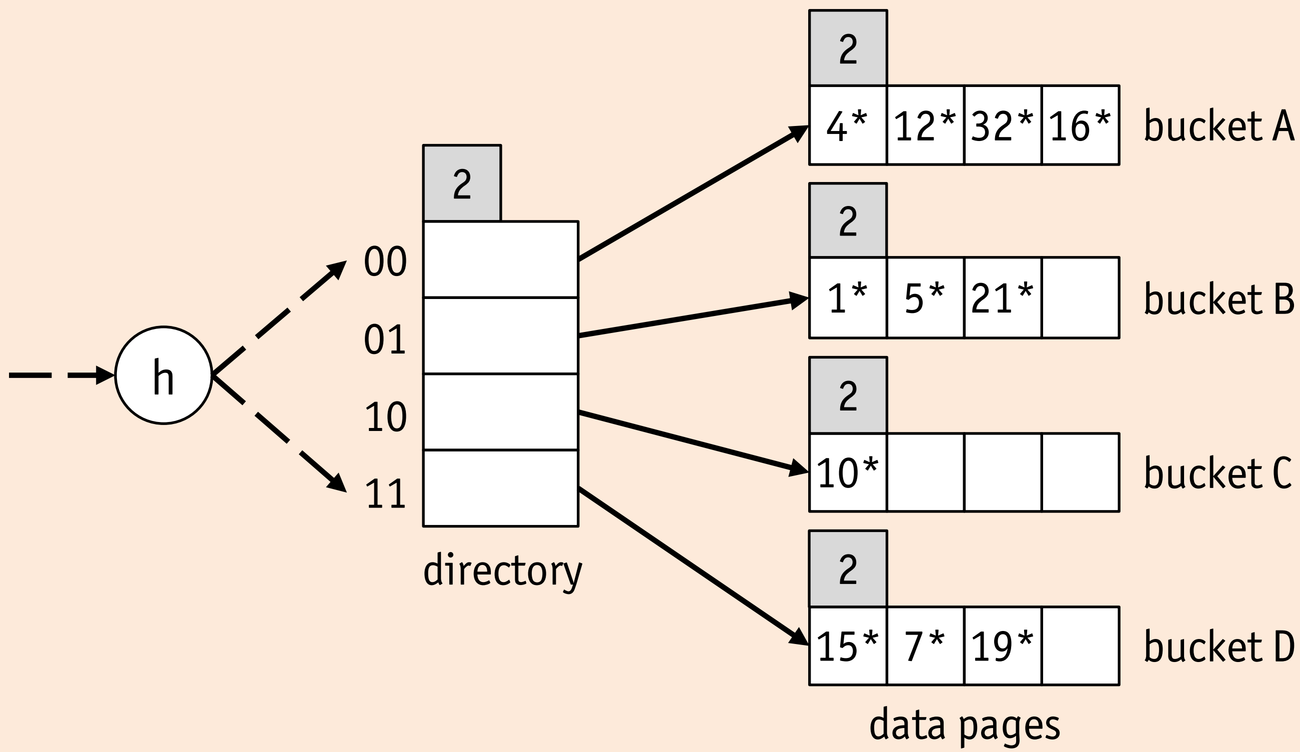
- **Microsoft SQL Server**, **Oracle**, and **DB2:** support for B+ tree indexes only
- **PostgreSQL:** support for both B+ tree and hash indexes (linear hashing)
- **MySQL:** depending on storage engine, both B+ tree and hash indexes are supported
- **Berkeley DB:** support for both B+ tree and hash indexes (linear hashing)

↳ However, almost all of these systems implement the **Hybrid Hash Join** (physical) operator that uses hashing to compute the equijoin of two relations (see L. D. Shapiro: **"Join Processing in Database Systems with Large Main Memories"**, 1986)

Extendible Hashing

- **Extendible hashing** adapts to growing (or shrinking) data files
- To keep track of the actual primary buckets that are part of the current hash table, an **in-memory bucket directory** is used

Example: Extendible hash table setup (ignore the $\boxed{2}$ fields for now)



Note: This figure depicts the entries as $h(k)*$, not $k*$

Extendible Hashing Search

Search for a record with key *k*

1. Apply *h*, i.e., compute *h*(*k*)
2. Consider the last $\boxed{2}$ bits of h(k) and follow the corresponding directory pointer to find the bucket

- The meaning of the $\boxed{}$ fields might become clear now

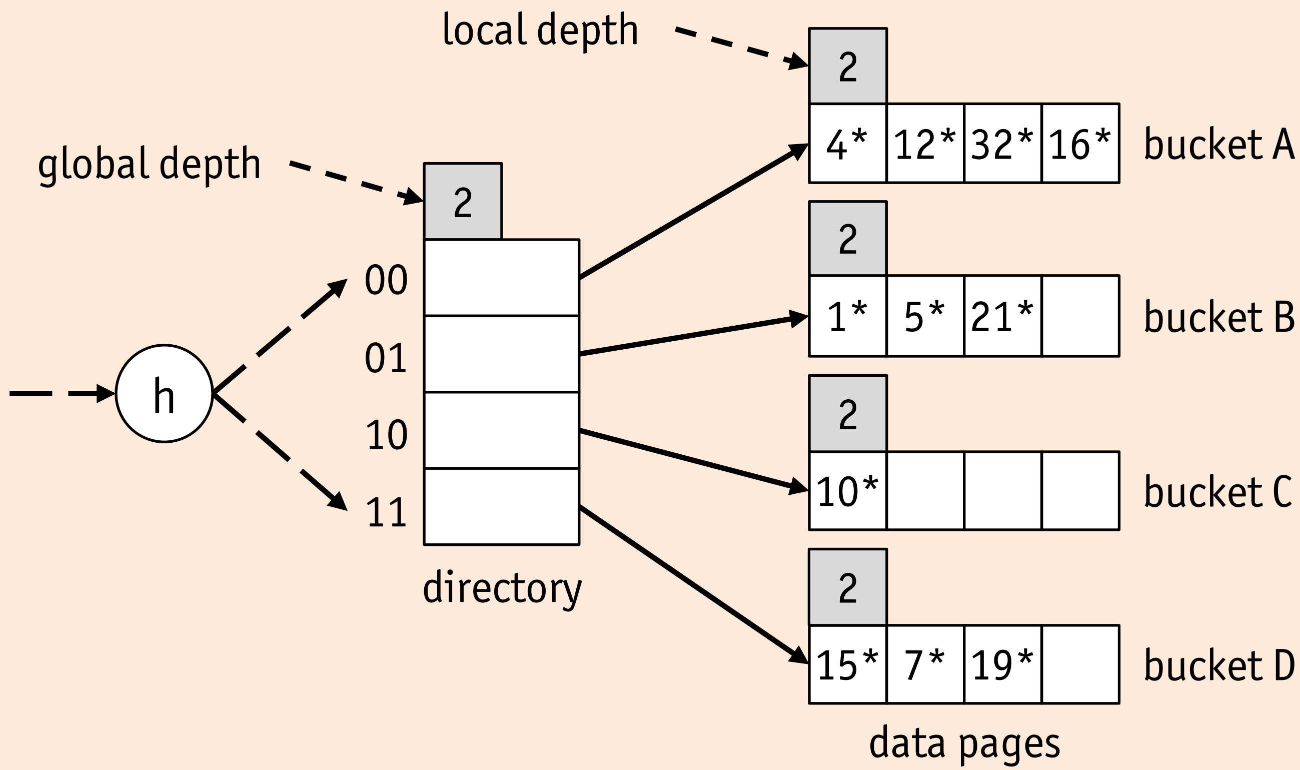
Global and local depth annotations

- **Global depth** ($\boxed{n}$ at hash directory)
  Use the last n bits of h(k) to lookup a bucket pointer in the directory
  (the directory size is 2^n)
- **Local depth** ($\boxed{d}$ at individual buckets)
  The hash values h(k) of all entries in this bucket agree on their last d bits

Extendible Hashing Search

Example: Find a record with key *k* such that *h*(*k*) = 5

Example: To find a record with key k such that $h(k) = 5 = 101_2$, follow the second directory pointer ($101_2 \wedge 11_2 = 01_2$) to bucket B, then use entry 5* to access the record



Extendible Hashing Search

Searching in extendible hashing

function hsearch (k) **:** $\uparrow$*bucket*
  $n \leftarrow \boxed{n}$; (global depth of hash directory)
  $b \leftarrow h(k)$ **&** $(2n - 1)$; (mask all but the low n bits)
  $\uparrow$*bucket* $\leftarrow$ *bucket*$[b]$;
end

- Remarks
  - *bucket*$[0, \ldots, 2^n - 1]$ is an **in-memory array** whose entries point to the hash buckets
  - search returns a pointer to hash bucket containing **potential** hit(s)
  - **&** and **|** denote **bit-wise and** and **bit-wise or** (like in C, C++, Java, etc.)

Extendible Hashing

Insert a record with key *k*

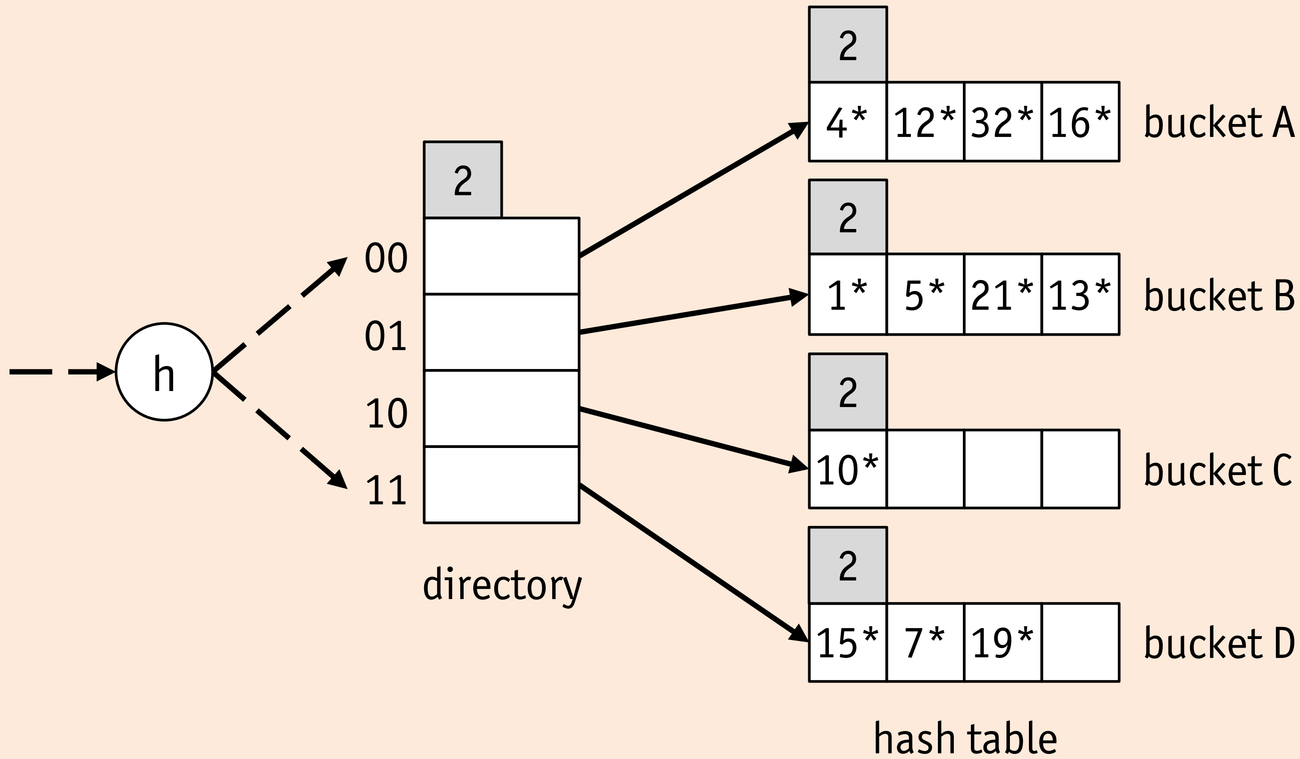
1. Apply h, i.e., compute $h(k)$
2. Use the last $\boxed{2}$ bits of $h(k)$ to lookup the bucket pointer in the directory
3. If the **primary bucket** still has capacity, store $h(k)^*$ in it
   Otherwise...?

- We *cannot* start an overflow chain hanging off the primary bucket as that would compromise uniform I/O behavior
- We *cannot* place $h(k)^*$ in another primary bucket since that would invalidate the hashing principle

Extendible Hashing Insert

Example: Insert a record with $h(k) = 13$

To insert a record with key k such that $h(k) = 13 = 11\underline{01}_2$, follow the second directory pointer (entry 01) to bucket B (which still has empty slots) and place 13* there



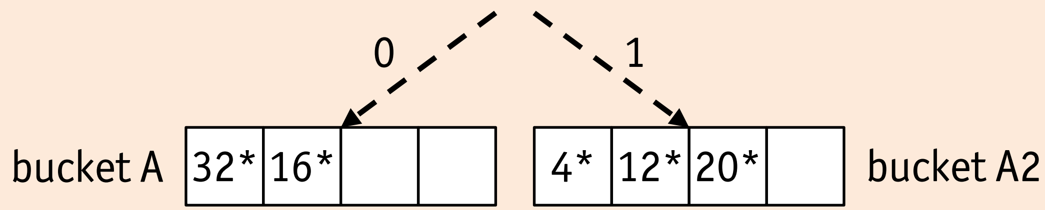
Extendible Hashing Insert

Example: Insert a record with $h(k) = 20$

Inserting a record with key k such that $h(k) = 20 = 101\underline{00}_2$ causes an **overflow in primary bucket** A and therefore a **bucket split** for A

1. Split bucket A by creating a new bucket A2 and use bit position $\boxed{d}$ + 1 to redistribute the entries

$$
\begin{aligned}
4 &= \underline{1}00_2 \\
12 &= 1\underline{1}00_2 \\
32 &= 100\underline{0}00_2 \\
16 &= 10\underline{0}00_2 \\
20 &= 10\underline{1}00_2
\end{aligned}
$$

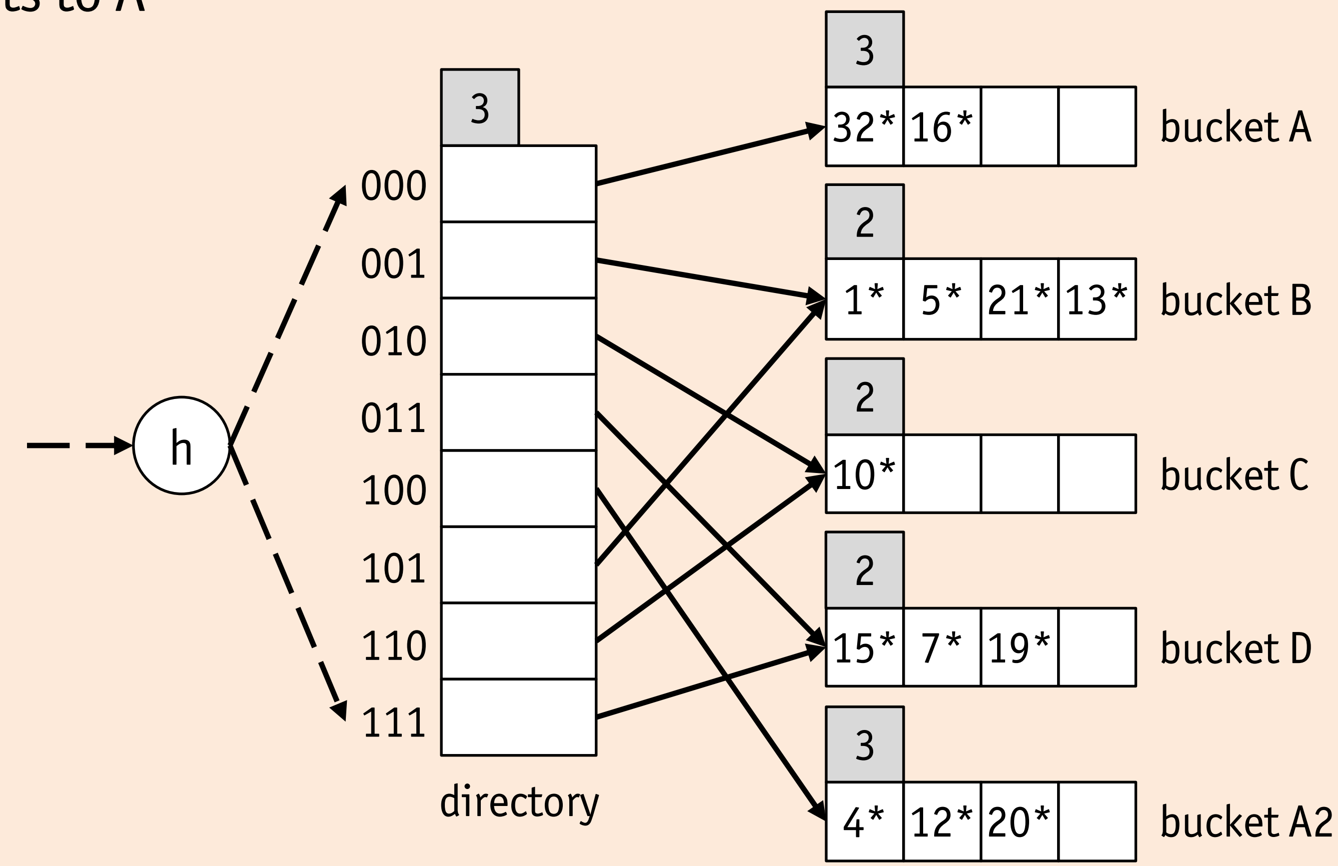


Note that now $\boxed{3}$ bits are used to discriminate between the old bucket A and the new split bucket A2

Extendible Hashing Insert

Example: Insert a record with $h(k) = 20$

2. To address the new bucket, the directory needs to be **doubled** by simply copying its original pages (bucket pointer lookups now use $\boxed{2}$ + 1 = $\boxed{3}$ bits)
3. Let bucket pointer for $\underline{1}00_2$ point to A2, whereas the directory pointer for $\underline{0}00_2$ still points to A



Extendible Hashing Insert

✎ Doubling the directory

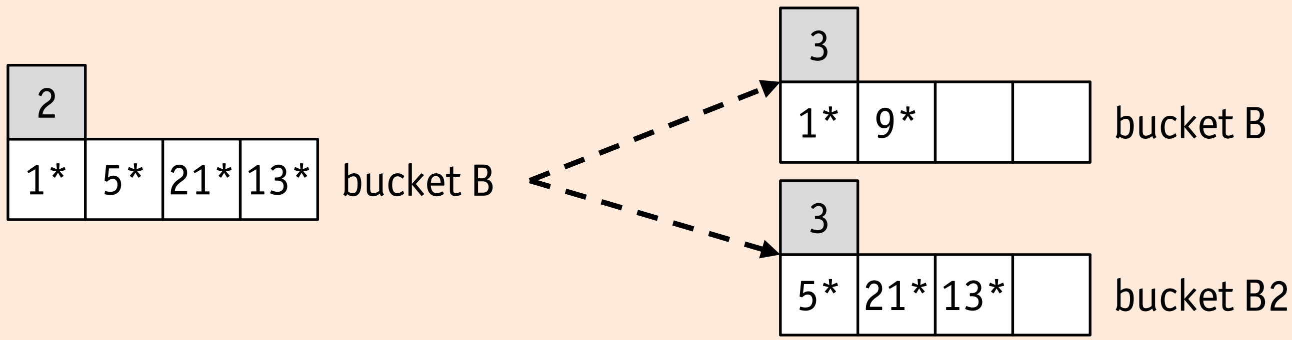
In the previous example, the directory had to be double to address the new split bucket. Is doubling the directory always necessary when a bucket is split? Or, how could you tell whether directory doubling is required or not?

Extendible Hashing Insert

- If the local depth of the split bucket is smaller than then global depth, i.e., $\boxed{d} < \boxed{n}$, directory doubling is **not** necessary

Example: Insert a record with $h(k)$ = 9

- Insert record with key k such that $h(k) = 9 = \underline{1}001_2$
- The associated bucket B is split by creating a new bucket B2 and redistributing the entries

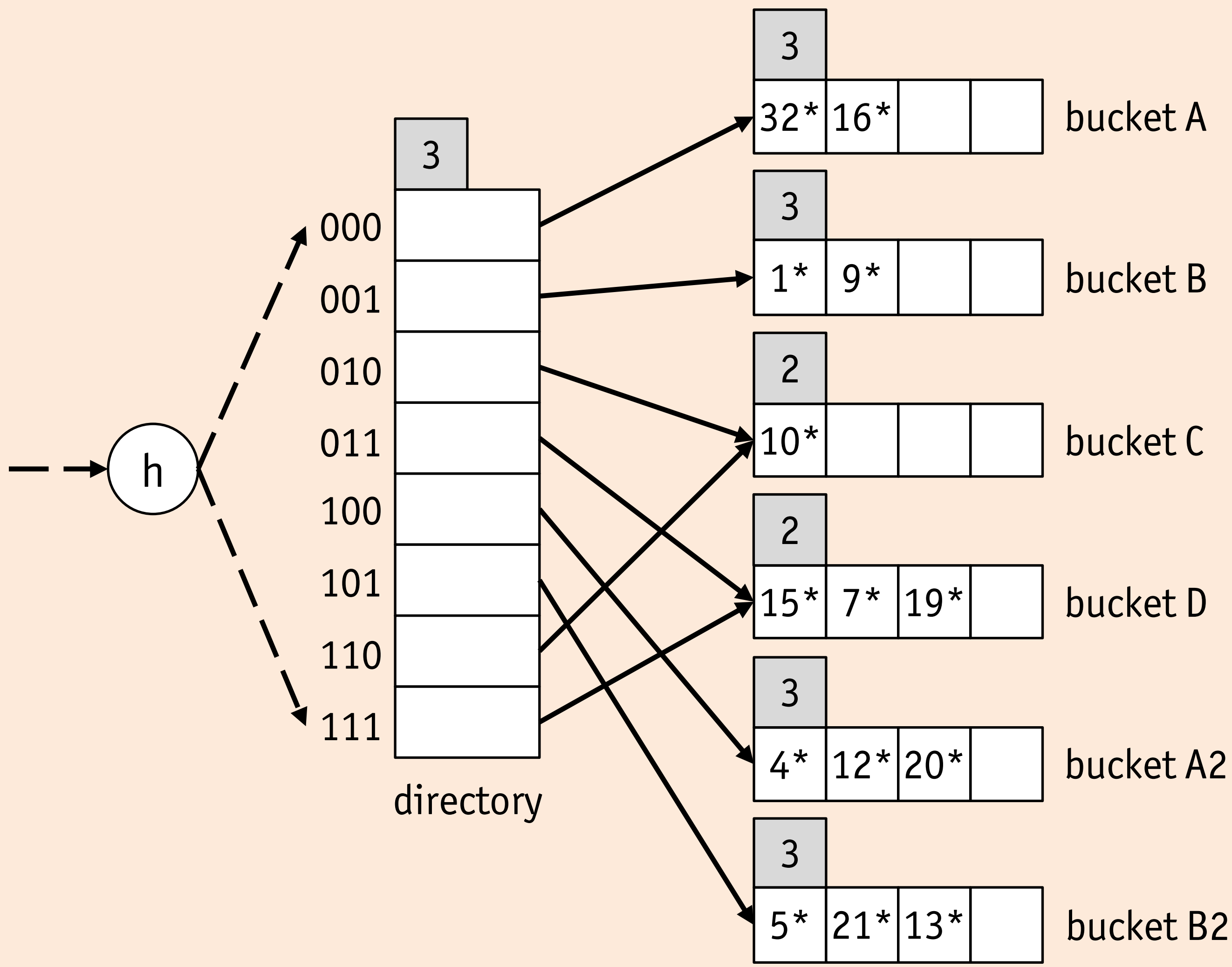


- The new local depth of B and B2 is $\boxed{3}$ and thus does **not** exceed the global depth of $\boxed{3}$

↳ Modifying the directory's bucket pointer for $\underline{101}_2$ is sufficient (see next slide)

Extendible Hashing Insert

Example: Insert a record with $h(k) = 9$ (cont'd)



Extendible Hashing Insert

Insert in extendible hashing

function hinsert(k^*)
$n \leftarrow \boxed{n}$; (global depth of hash directory)
$b \leftarrow$ **hsearch**(k);
if b has capacity **then**
place k^* in bucket b;
else ...
end

Extendible Hashing Insert

Insert in extendible hashing (cont'd)

```
function hinsert(k*)
   n ← [n];                                              (global depth of hash directory)
   b ← hsearch(k);
   if b has capacity then ...
   else
      d ← [d];                                               (local depth of bucket b)
      create a new empty bucket b2;
      foreach k'* in bucket b do          (redistribute entries of bucket b including k*)
         if h(k') & 2^d ≠ 0 then move k'* to bucket b2;
      [d] ← d + 1;                            (new local depths for buckets b and b2)
      if n < d + 1 then                                 (directory has to be doubled)
         allocate 2^n directory entries bucket[2^n, ..., 2^(n+1) − 1];
         copy bucket[0, ..., 2^n − 1] into bucket[2^n, ..., 2^(n+1) − 1];
         [n] ← n + 1;
      bucket[(h(k) & (2^n − 1)) | 2^n] ← @(b2);
end
```

Overflow Chains in Extendible Hashing

✎ Overflow chains

Extendible hashing uses overflow chains hanging off a bucket only as a last resort.
Under which circumstances will extendible hashing create an overflow chain?

Extendible Hashing Delete

- Routine **`hdelete(`***k**`)`** locates and removes entry *k**
  - deleting an entry *k** from a bucket may leave this bucket **empty**
  - an empty buckets can be **merged** with its split bucket
  - however, this step is often **omitted** in practice

✎ Delete in extendible hashing

When is the **local depth** decreased?

When is the **global depth** decreased?

Linear Hashing

- Similar to extendible hashing, **linear hashing** can adapt its underlying data structure to record insertions and deletions
  - linear hashing **does not need a hash directory** in addition to the actual hash table buckets
  - linear hashing can define **flexible criteria** that determine when a bucket is to be split
  - linear hashing may perform badly if the **key distribution is skewed**

Linear Hashing

- Linear hashing uses an **ordered family of hash functions**
  - sequence of hash functions h_0, h_1, h_2, ... (subscript is often called *level*)
  - range of $h_{level+1}$ is **twice as large** as range of h_{level} (for *level* = 0, 1, 2, ...)

Hash Function Family

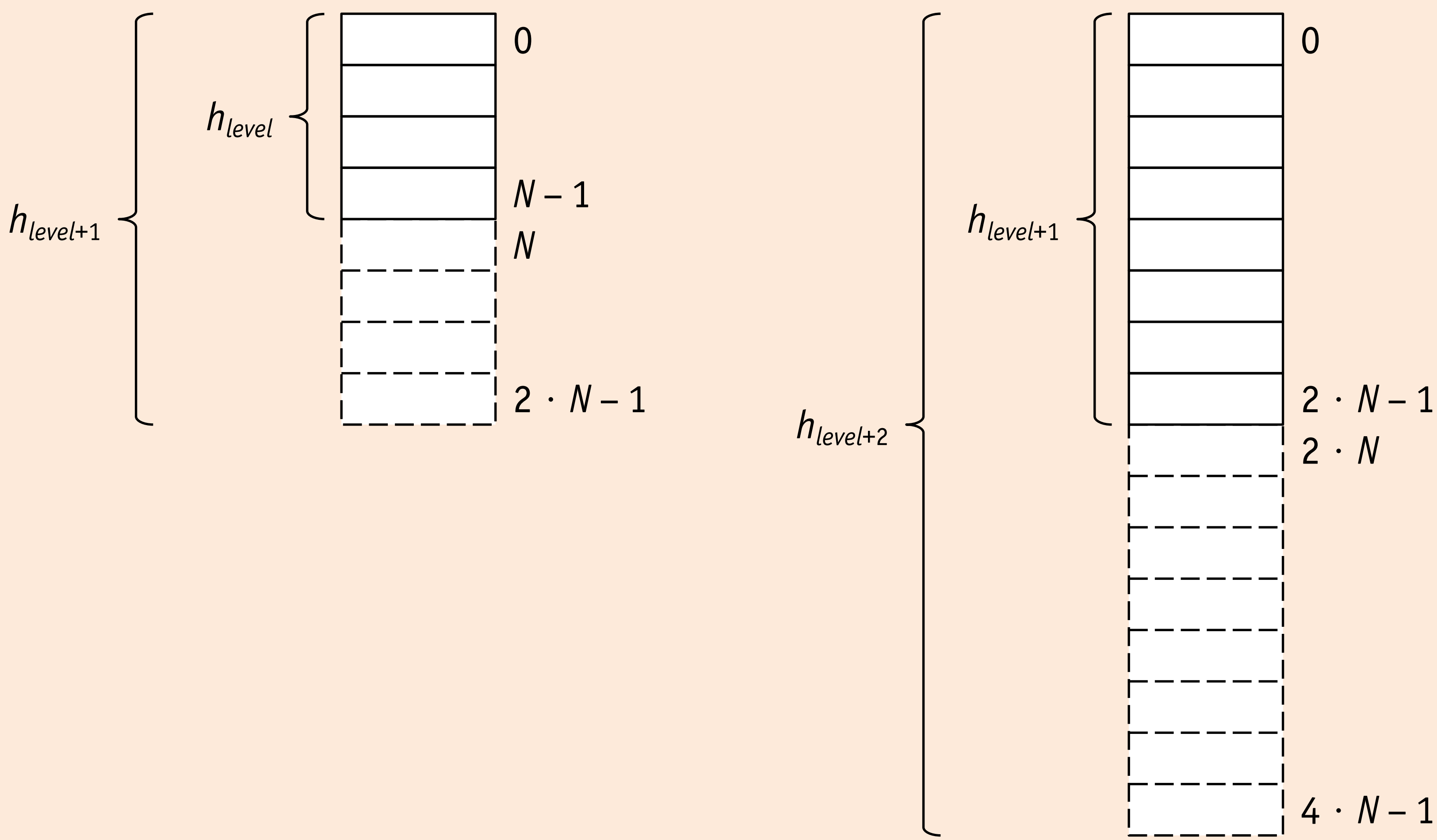
Given an initial hash function h and an initial hash table size N, one approach to define such a family of hash functions $h_0, h_1, h_2, \ldots$ would be

$$h_{level}(k) = h(k)\ mod\ \left(2^{level} \cdot N\right)$$

where $level = 0, 1, 2, \ldots$

Linear Hashing

Example: h_{level} with range [0, …, N – 1]



Linear Hashing

Basic linear hashing scheme

1. Initialize *level* ← 0 and *next* ← 0
2. The **current hash function** in use for searches (insertions/deletions) is h_{level}, **active hash buckets** are those in the range of h_{level}, i.e., $[0, \ldots, 2^{level} \cdot N - 1]$
3. Whenever the current **hash table overflows**
   - insertions filled a primary bucket beyond c% occupancy
   - overflow chain of a bucket grew longer than *p* pages
   - or (*insert your criterion here*)

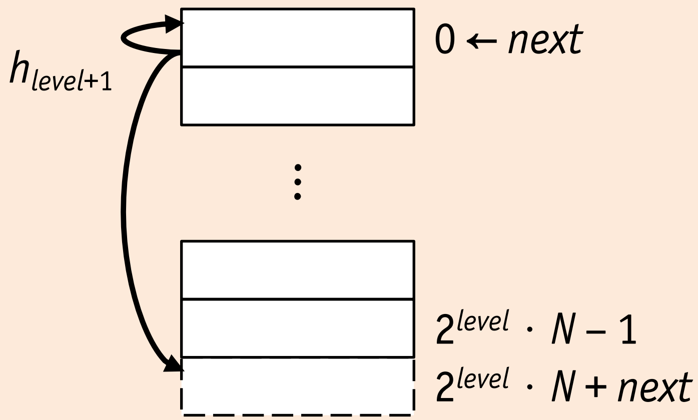
   the bucket **at hash table position** *next* **is split**

Note: In general the bucket that is split is **not** the bucket that triggered the split!

Linear Hashing

Bucket split

1. **Allocate a new bucket and append** it to the hash table at position $2^{level} \cdot N = next$
2. **Redistribute** the entries in bucket next by **rehashing** them via $h_{level+1}$ (some entries will remain in bucket *next*, some will move to bucket $2^{level} \cdot N + next$)



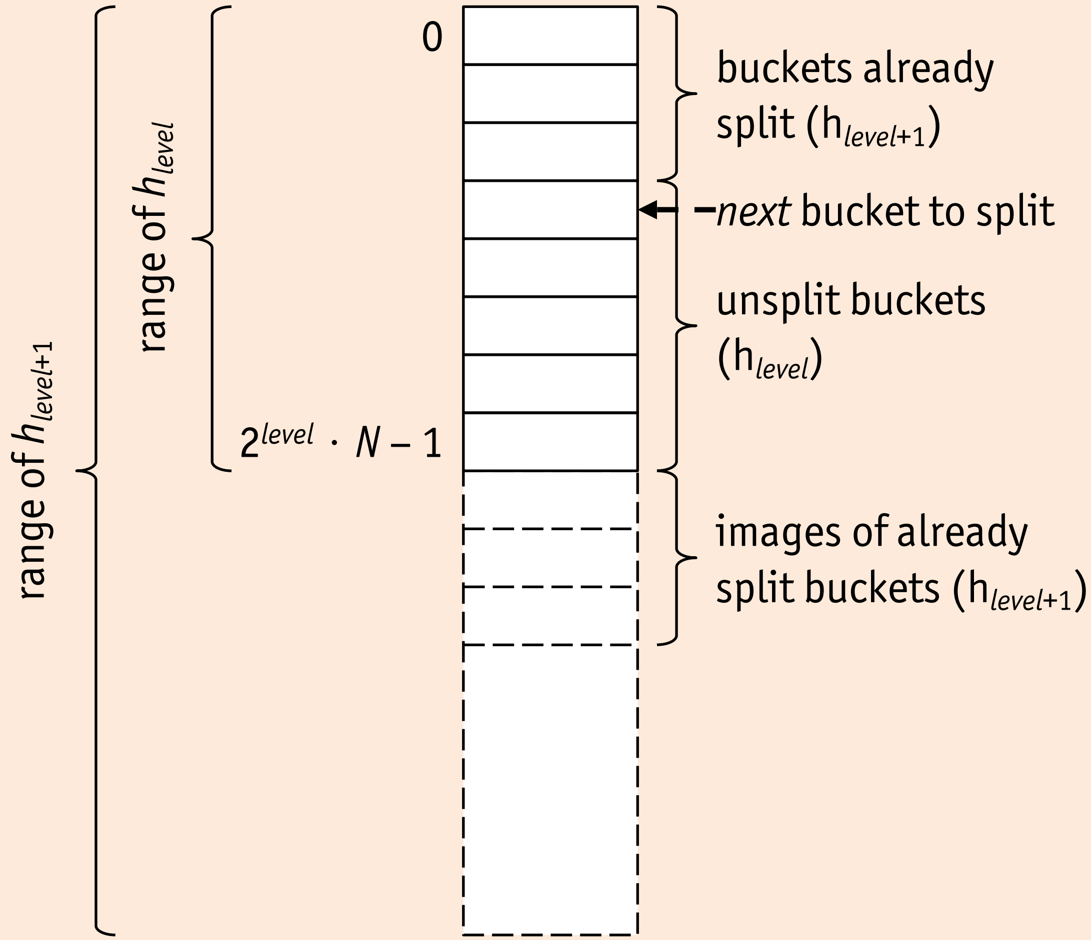
3. **Increment** *next* by 1

↳ All buckets with positions < *next* have been rehashed

Linear Hashing

Rehashing

With every bucket split, next walks down the hash table. Therefore, hashing via h_{level} (search, insert, and delete) needs to take **current** *next* **position** into account.



$$h_{level}(k) \begin{cases} < next: & \text{bucket already split, } \textbf{rehash:} \text{ find record in bucket } h_{level+1}(k) \\ \geq next: & \text{bucket not yet split, i.e., } \textbf{bucket found} \end{cases}$$

Linear Hashing

✎ Split rounds: what happens if *next* is incremented beyond the hash table size?

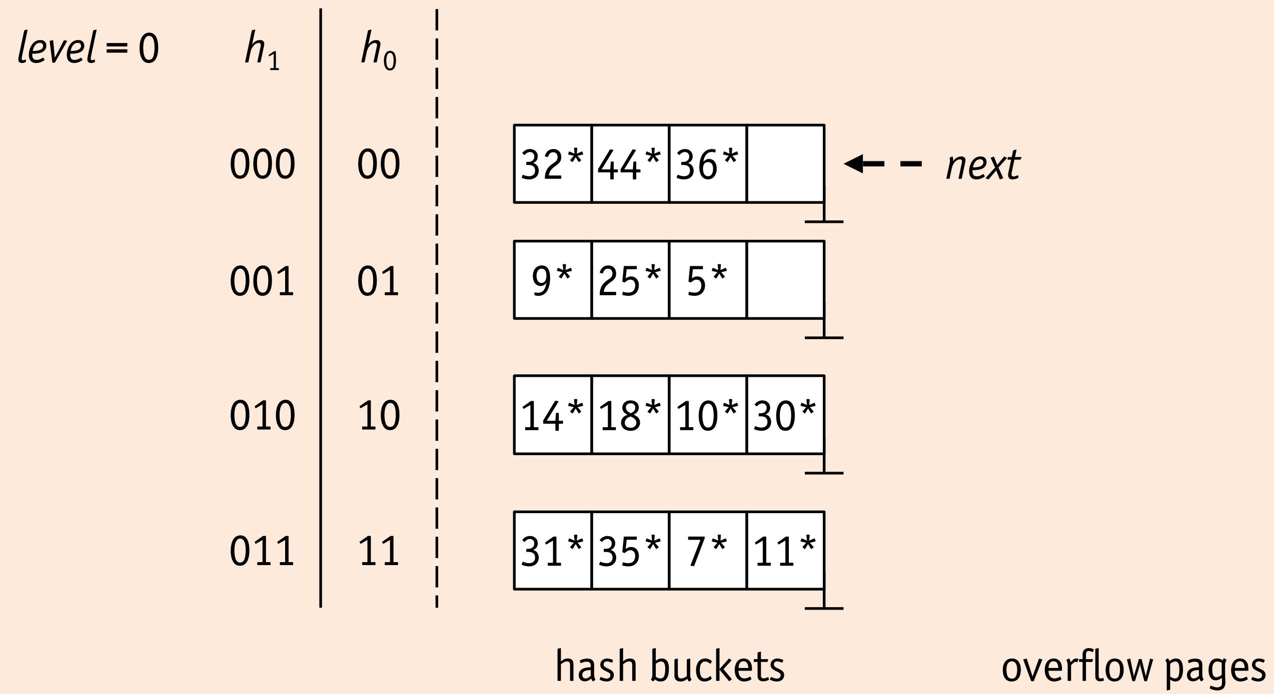
A bucket split increments *next* by 1 to mark the next bucket to be split. How would you propose to handle the situation when *next* is incremented **beyond** the currently last hash table position, i.e.,

$$next > 2^{level} \cdot N - 1?$$

Linear Hashing

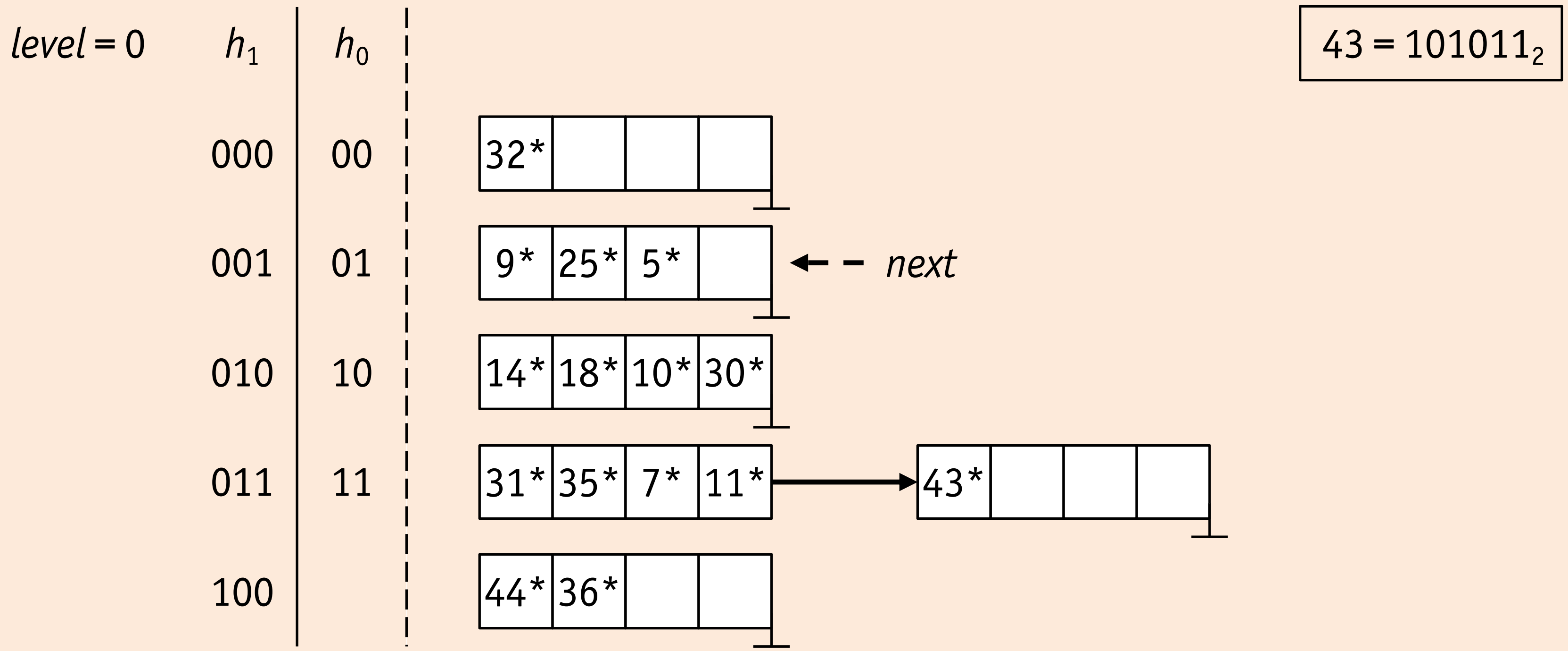
- Setup of linear hash table used in running example
  - bucket capacity of 4, initial hash table size $N = 4$, $level = 0$, $next = 0$
  - split criterion: allocation of a page in an overflow chain

Example: linear hash table ($h_{level}(k)$* shown)



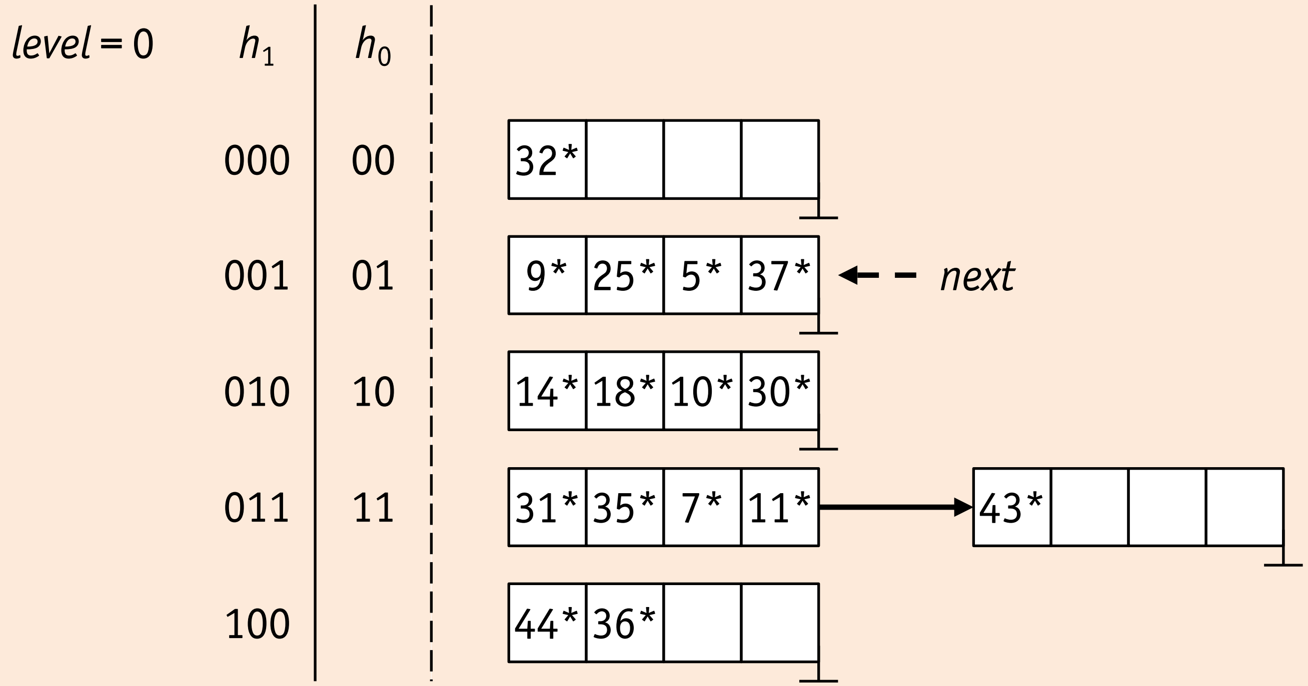
Linear Hashing

Example: insert record with key k such that $h_0(k) = 43$



Linear Hashing

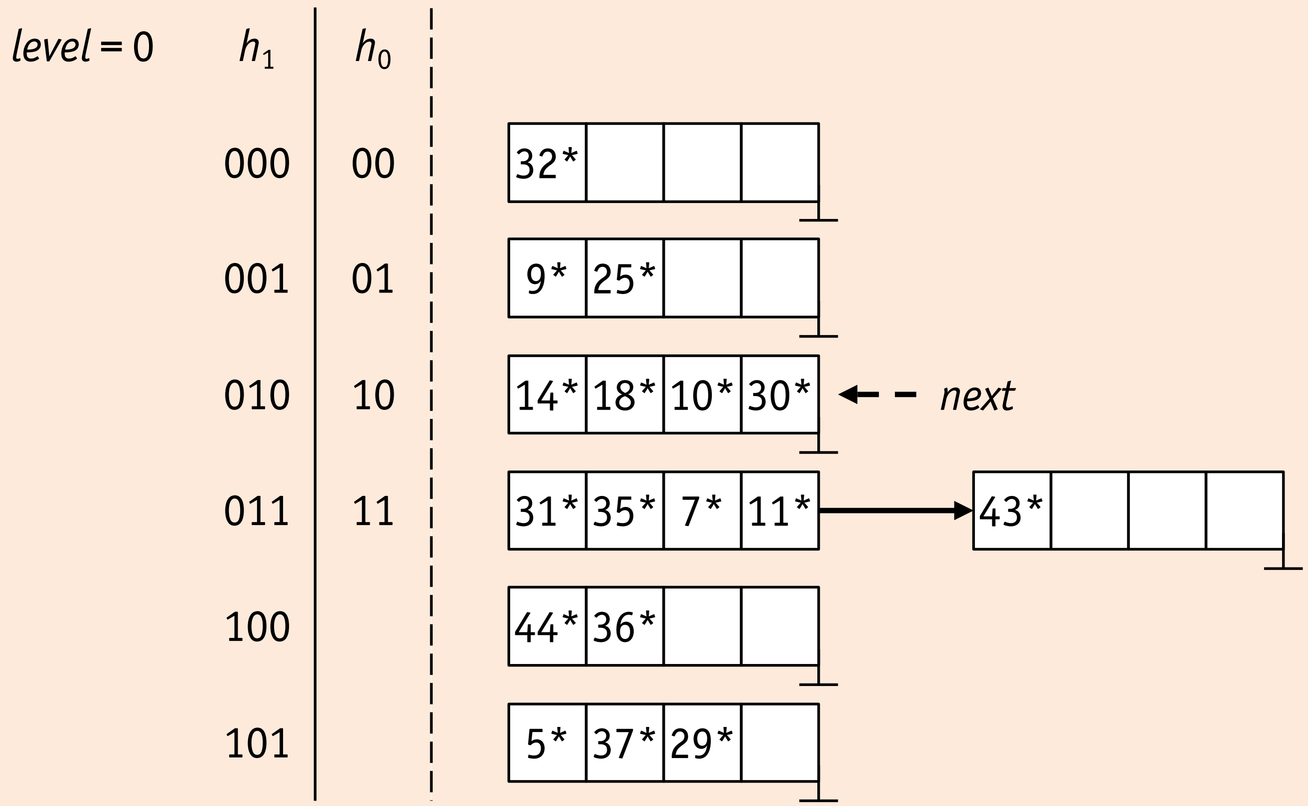
Example: insert record with key k such that $h_0(k) = 37$



$37 = 100101_2$

Linear Hashing

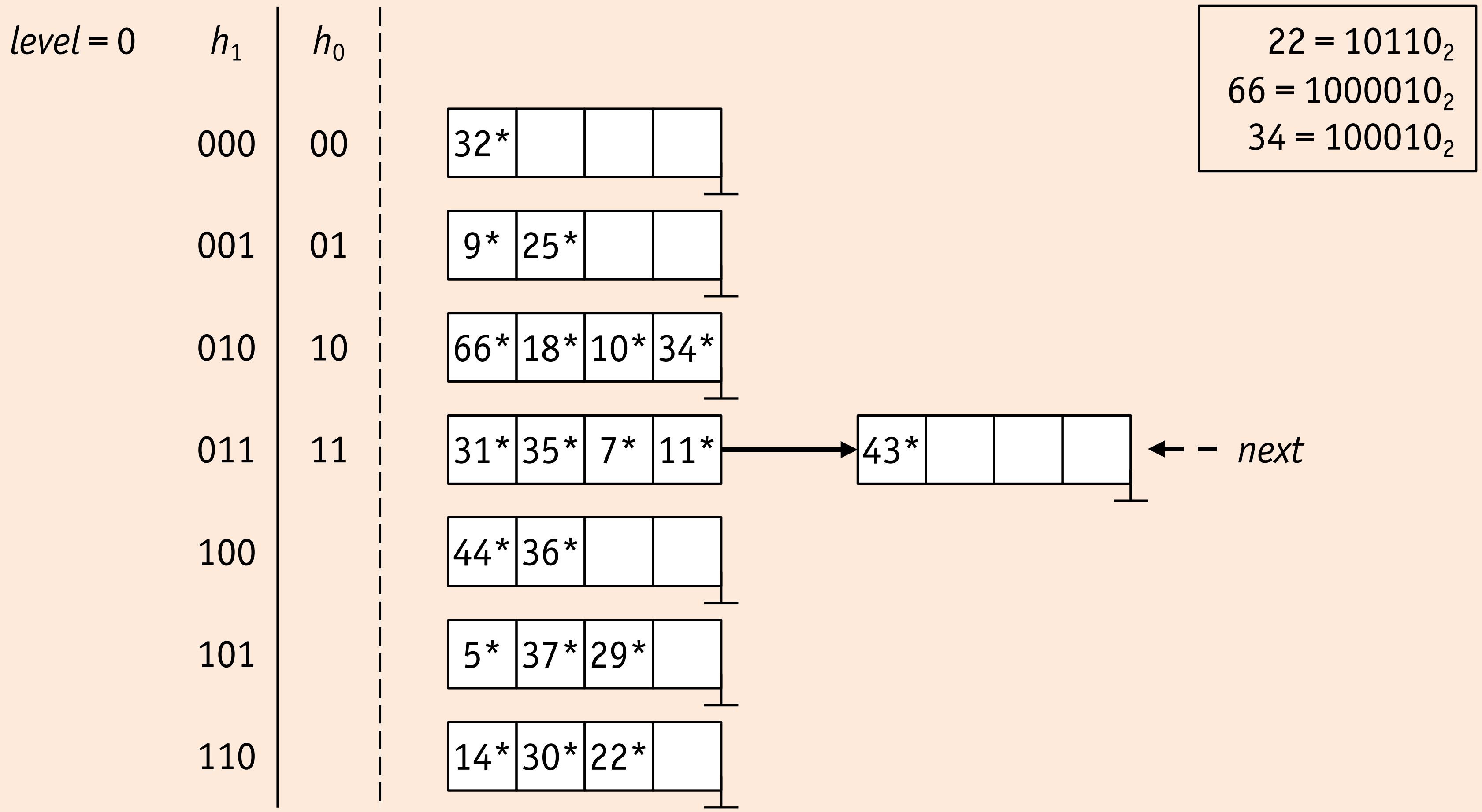
Example: insert record with key k such that $h_0(k) = 29$



$29 = 11101_2$

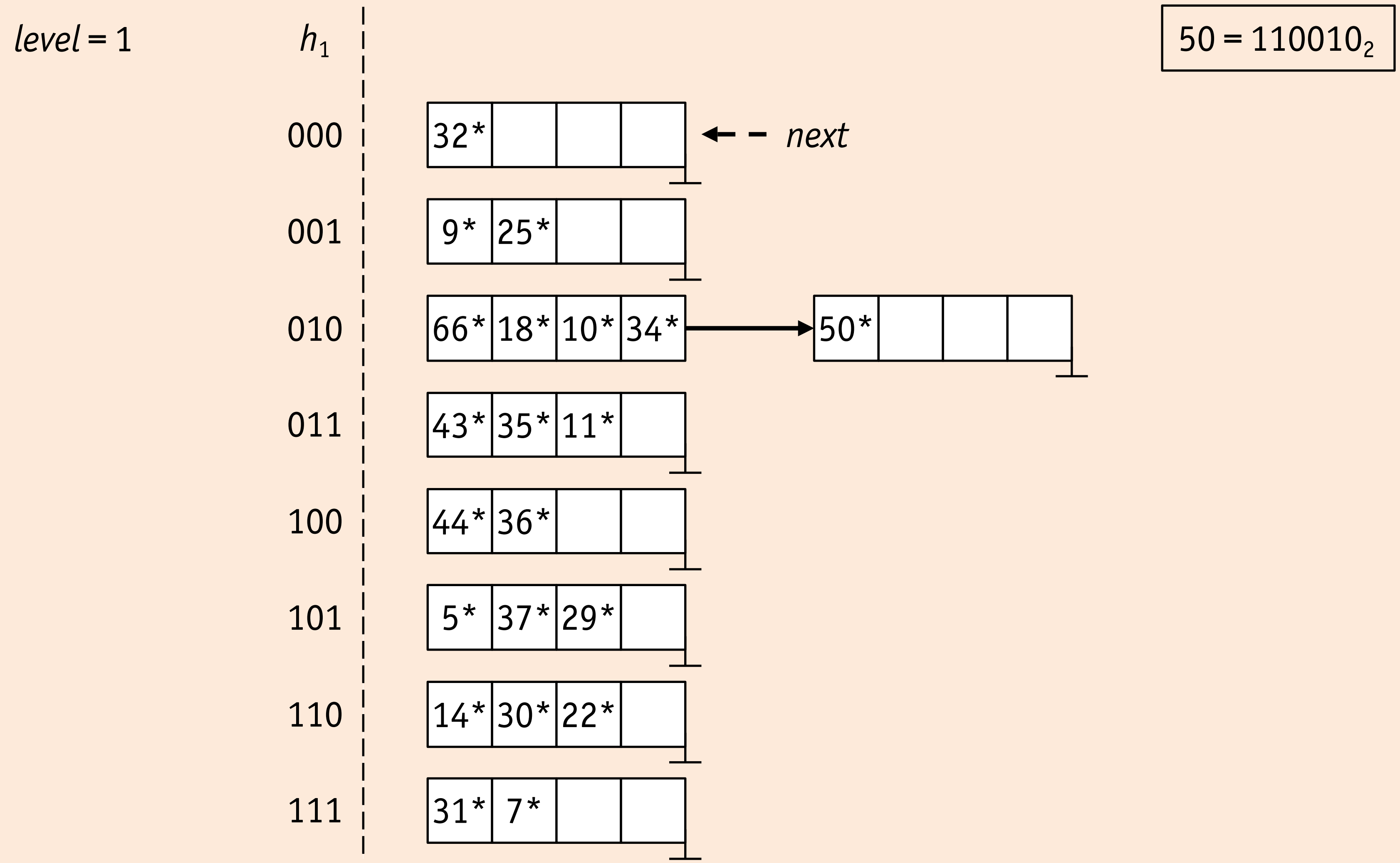
Linear Hashing

Example: insert record with key k such that $h_0(k)$ = 22, 66, and 34



Linear Hashing

Example: insert record with key *k* such that $h_0(k) = 50$



Note: Rehashing a bucket means to **rehash its overflow chain** as well.

Linear Hashing Search

Search in linear hashing

```
function hsearch(k)
   b ← h_level(k);
   if b < next then
      b ← h_level+1(k);
   return bucket[b];
end
```

(b has already been split, record for key may be in bucket b or bucket $2^{level} \cdot N + b$ → **rehash**)

- Remarks
  - *bucket*[0, …, $2^{level} \cdot N - 1$] is an **in-memory array** containing hash table bucket (page) addresses
  - variables *level* and *next* are **global variables** of the linear hash table, N is constant

Linear Hashing Search

Insert in linear hashing

```
function hinsert(k*)
    b ← h_level(k);
    if b < next then                                              (rehash)
        b ← h_level+1(k);
    place k* in bucket[b];
    if overflow(bucket[b]) then          (last insertion triggered a split of bucket next)
        allocate a new bucket b';
        bucket[2^level · N + next] ← @(b');                (grow hash table by one page)
        foreach entry k'* in bucket[next] do          (rehash to redistribute entries)
            place entry k'* in bucket[h_level+1(k')];
        next ← next + 1;
        if next > 2^level · N − 1 then             (every bucket of the hash table been split)
            level ← level + 1;
            next ← 0;                        (hash table size has doubled, start a new round)
end
```

Note: Predicate `overflow(·)` is a tunable parameter to control triggering of splits.

Linear Hashing Delete (Sketch)

Insert in linear hashing

```
function hdelete(k)
  □
  if empty(bucket[2^level · N + next]) then          (deletion left last bucket empty)
    remove page pointed to by bucket[2^level · N + next] from hash table ;
    next ← next − 1 ;
    if next < 0 then                                  (round-robin scheme for deletion)
      level ← level − 1 ;
      next ← 2^level · N + next ;
end
```

- Remarks
  - linear hashing deletion is essentially the "inverse" of **`hinsert(·)`**
  - possible to replace **`empty(·)`** with a suitable **`underflow(·)`** predicate

Extendible vs. Linear Hashing

- Directory vs. no directory
  - suppose linear hashing also used a directory with elements [0, ..., N – 1]
  - since first split is at bucket 0, element N is added to the directory
  - imagine the directory is actually doubled at this point
  - since element 1 is the same as element N + 1, element 2 is the same as element N + 2, and so on, copying these elements can be avoided
  - at end of the round, all N buckets are split and directory doubled in size
- Directory vs. hash function family
  - choice of hashing functions is very similar to effect of directories
  - moving from h_i to h_{i+1} corresponds to doubling the directory: both operations double effective range into which key values are hashed
  - doubling range in a single step vs. doubling range gradually
- New idea behind linear hashing is that directory **can be avoided** by a clever choice of the bucket to split

Universität
Konstanz

Database System Architecture and Implementation

TO BE CONTINUED...