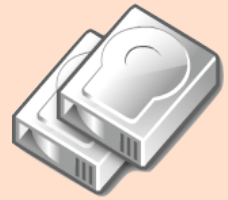
Chapter 6

Hash-Based Indexing

Efficient Support for Equality Search

Architecture and Implementation of Database Systems
Winter 2008/09

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen



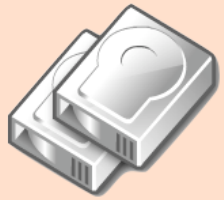
Hash-Based Indexing

- We now turn to a different family of index structures: **hash indexes.**
- Hash indexes are "unbeatable" when it comes to support for **equality selections**:

Equality selection

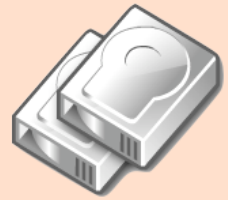
```
1 SELECT *
2 FROM   R
3 WHERE  A = k
```

- Further, other query operations internally generate a flood of equality tests (*e.g.*, nested-loop join).
  (Non-)presence of hash index support can make a real difference in such scenarios.



Hashing *vs.* B$^+$-trees

- Hash indexes provide **no support for range queries**, however (hash indexes are also know as **scatter storage**).
- In a B$^+$-tree-world, to locate a record with key k means to **compare** k with 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) to find the location of the associated record.
- We will now briefly look into **static hashing** to illustrate the basics.
  - Static hashing does *not* handle updates well (much like ISAM).
  - Later, we introduce **extendible hashing** and **linear hashing** which refine the hashing principle and adapt well to record inserions and deletions.



Static Hashing

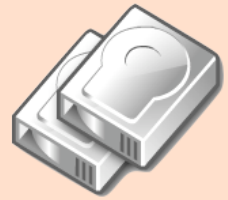
- To build a **static hash index** on attribute `A`:

Build static hash index on column `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 the pointer to **null**).
3. Define a **hash function** h with *range* $[0, \ldots, N-1]$. The *domain* of h is the type of `A`, *e.g.*.

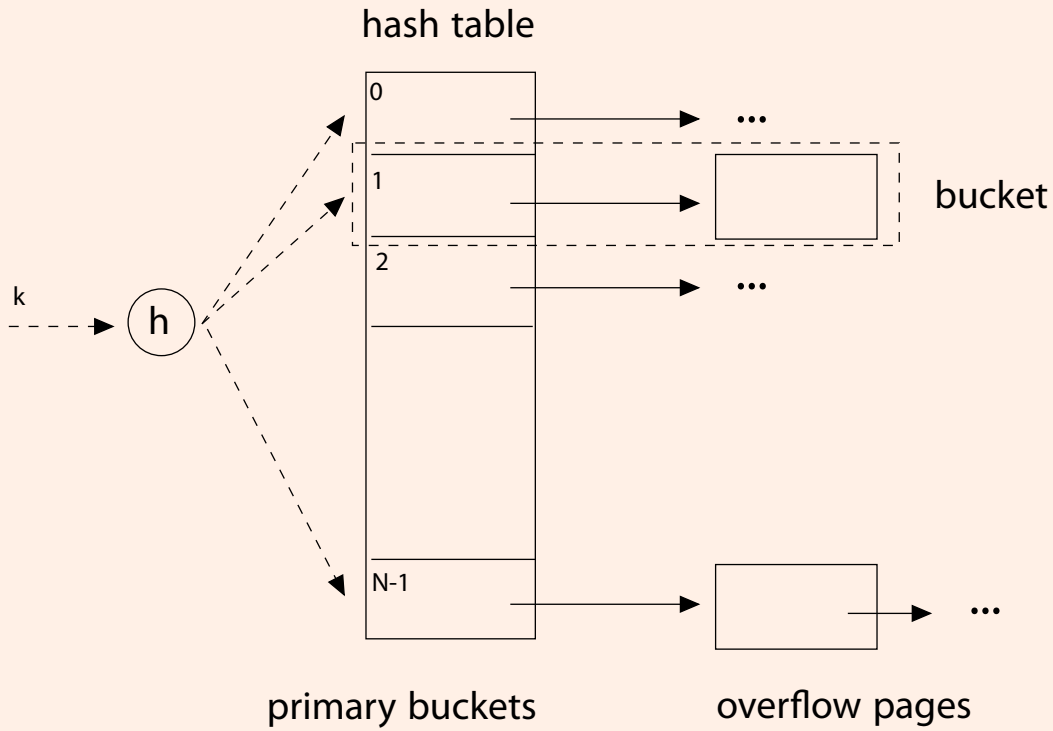
$$h : \texttt{INTEGER} \longrightarrow [0, \ldots, N-1]$$

if `A` is of SQL type `INTEGER`.

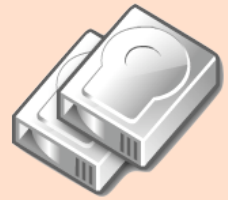


Static Hashing

Static hash table



- A primary bucket and its associated chain of overflow pages is referred to as a **bucket** (above).
- Each bucket contains **index entries** $k*$ (implemented using any of the variants A, B, C, see slide 0.0).



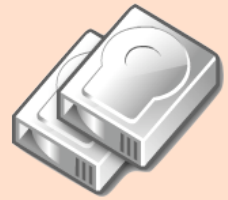
Static Hashing

- To perform `hsearch(`k`)` (or `hinsert(`k`)`/`hdelete(`k`)`) for a record with key `A` $= k$:

Static hashing scheme

1. **Apply hash function** h to the key value, *i.e.*, compute $h(k)$.
2. **Access the primary bucket page** with number $h(k)$.
3. Search (insert/delete) subject record on this pages or, if required, **access the overflow chain** of bucket $h(k)$.

- If the hashing scheme works well and overflow chain access is avoidable,
  - `hsearch(`k`)` requires a **single I/O operation**,
  - `hinsert(`k`)`/`hdelete(`k`)` require **two I/O operations**.



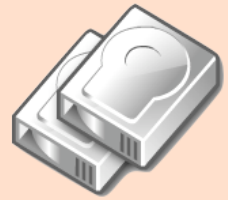
Static Hashing: Collisions and Overflow Chains

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

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

  (otherwise we would need as many primary bucket pages as different key values in the data file).
- At the same time, we want h to **scatter** the key attribute domain **evenly** across $[0, \ldots, N-1]$ to avoid the development of long overflow chains for few buckets. This makes the hash tables' I/O behavior non-uniform and unpredictable for a query optimizer.

- Such "good" hash functions are hard to discover, unfortunately.



The Birthday Paradox (Need for Overflow Chain Management)

Example (The birthday paradox)

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

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

Check: Compute the probability that n people *all have different birthdays*:

1 **Function:** `different_birthday` (n)

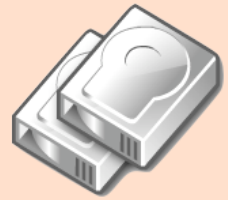
2 **if** $n = 1$ **then**

3 **return** 1;

4 **else**

return $\underbrace{\texttt{different\_birthday}(n-1)}_{\substack{\text{probability that } n-1 \text{ persons} \\ \text{have different birthdays}}} \times \underbrace{\frac{365-(n-1)}{365}}_{\substack{\text{probability that } n\text{th person} \\ \text{has birthday different from} \\ \text{first } n-1 \text{ persons}}}$;

5



Hash Functions

- If key values would be purely **random**, we could arbitrarily extract a few bits and use these for the hash function. Real key distributions found in database instances are far from random, though.

Hash function

1. **By division**. Simply define

$$h(k) = k \bmod N \ .$$

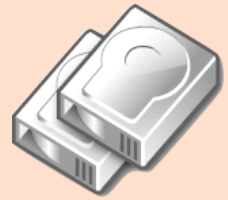
This guarantees the range of $h(k)$ to be $[0, \ldots, N-1]$.

Note: Choosing $N = 2^d$ for some d effectively considers the least d bits of k only. **Prime numbers** work best for N.

2. **By multiplication**. Extract the fractional part of $Z \cdot k$ (for a specific Z[1]) and multiply by arbitrary hash table size N:

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

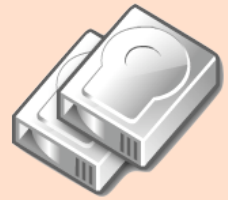
[1] The (inverse) **golden ratio** $Z = 2/(\sqrt{5}+1) \approx 0.6180339887$ is a good choice. See D.E.Knuth, "*Sorting and Searching*."



Static Hashing and Dynamic Files

- For a static hashing scheme:
  - If the underlying **data file grows**, the development of overflow chains spoils the otherwise predictable behavior hash 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—a waste of page space.
- As in the ISAM case, however, static hashing has advantages when it comes to concurrent access.
- We may perodicially **rehash** the data file to restore the ideal situation (20 % free space, no overflow chains).

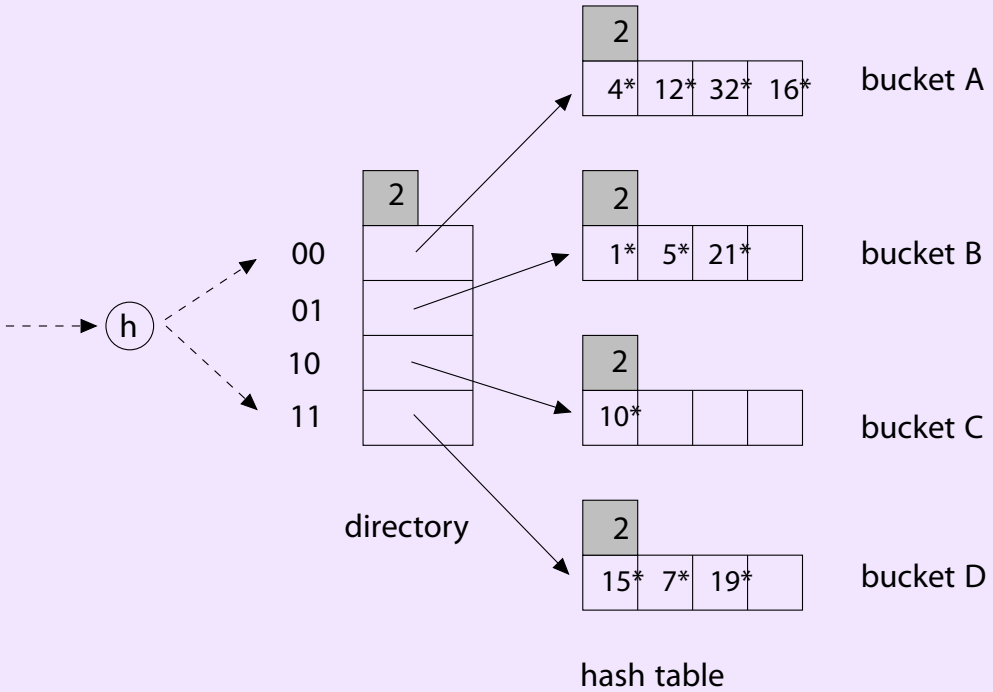
  $\Rightarrow$ Expensive and the index cannot be used while rehashing is in progress.



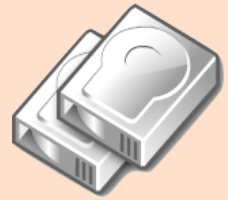
Extendible Hashing

- **Extendible Hashing** can adapt to growing (or shrinking) data files.
- To keep track of the actual primary buckets that are part of the current hash table, we hash via an **in-memory bucket directory**:

Example (Extendible hash table setup; ignore the 2 fields for now[2])



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



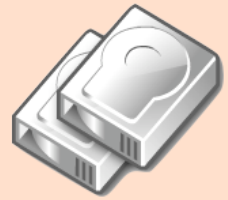
Extendible Hashing: Search

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.

Example (Search for a record)

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 wanted record.

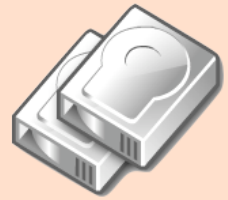


Extendible Hashing: Global and Local Depth

Global and local depth annotations

- **Gloabl depth** (n at hash directory):
  Use the last n bits of $h(k)$ to lookup a bcuket 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 last d bits.

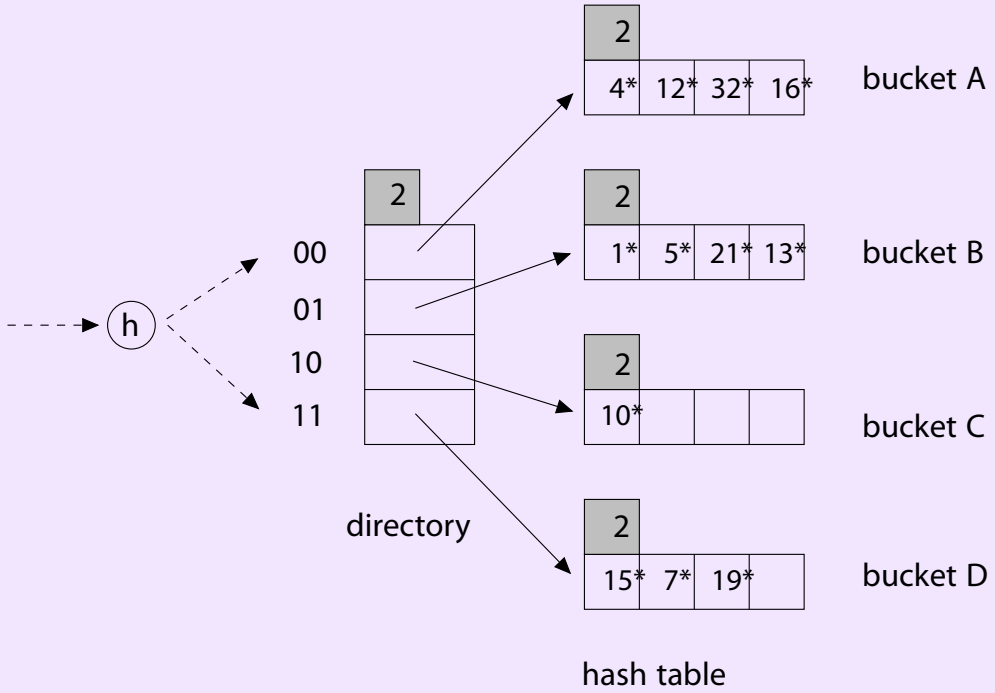


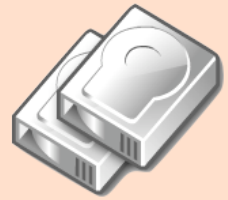
Extendible Hashing: Insert

Insert record with key k

1. Apply h, *i.e.*, compute $h(k)$.
2. Use the last n 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 . . . ?)

Example (Insert record with $h(k) = 13 = 1101_2$)



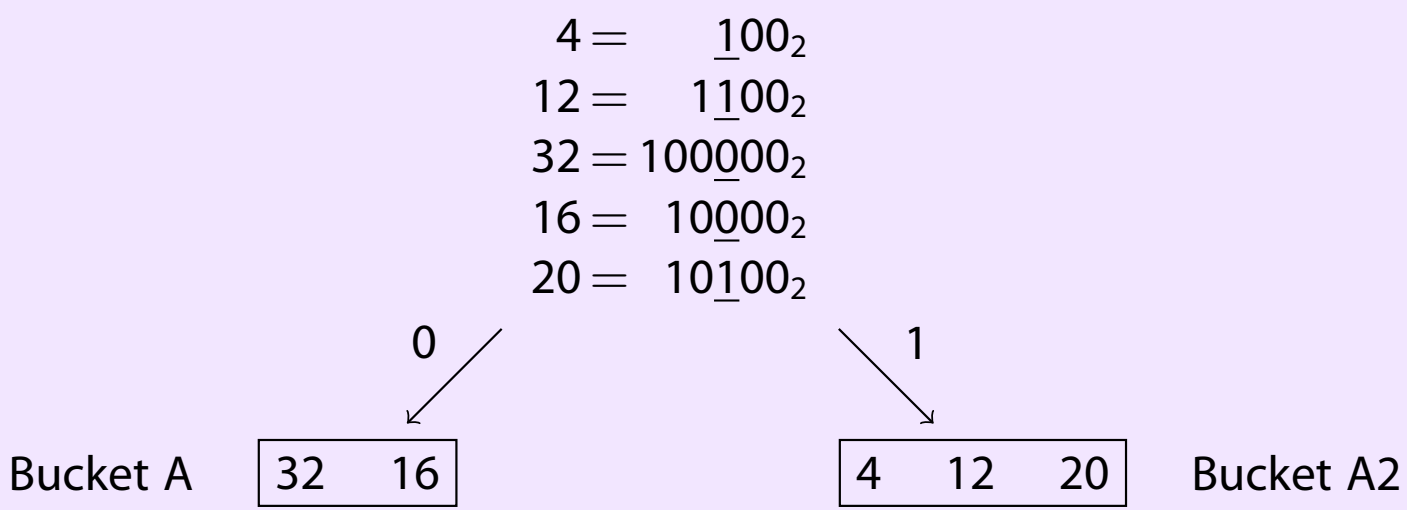


Extendible Hashing: Insert, Bucket Split

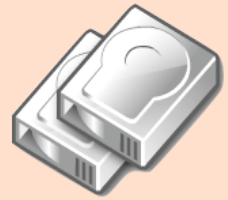
Example (Insert record with $h(k) = 20 = 10100_2$)

Insertion of a record with $h(k) = 20 = 10100_2$ leads to **overflow in primary bucket** A. Initiate a **bucket split** for A.

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



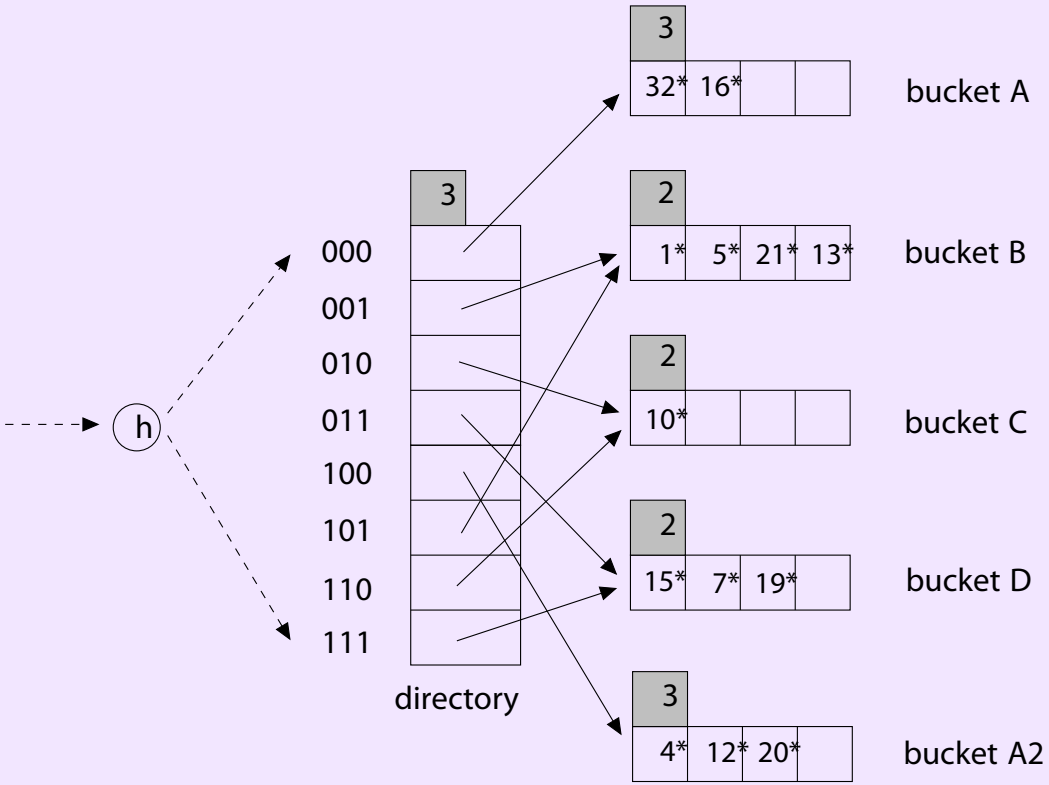
Note: We now need $\boxed{3}$ bits to discriminate between the old bucket A and the new split bucket A2.

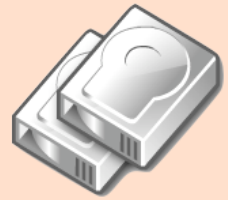


Extendible Hashing: Insert, Directory Doubling

Example (Insert record with $h(k) = 20 = 10100_2$)

2. *In the present case*, we need to **double the directory** by simply copying its original pages (we now use $\boxed{2} + 1 = \boxed{3}$ bits to lookup a bucket pointer).
3. Let bucket pointer for $\underline{1}00_2$ point to A2 (the directory pointer for $\underline{0}00_2$ still points to bucket A):





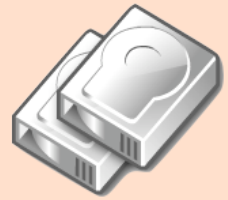
Extendible Hashing: Insert

If we split a bucket with local depth $\boxed{d} < \boxed{n}$ (global depth), directory doubling is *not* necessary:

Example (Insert record with $h(k) = 9 = 1\underline{001}_2$)

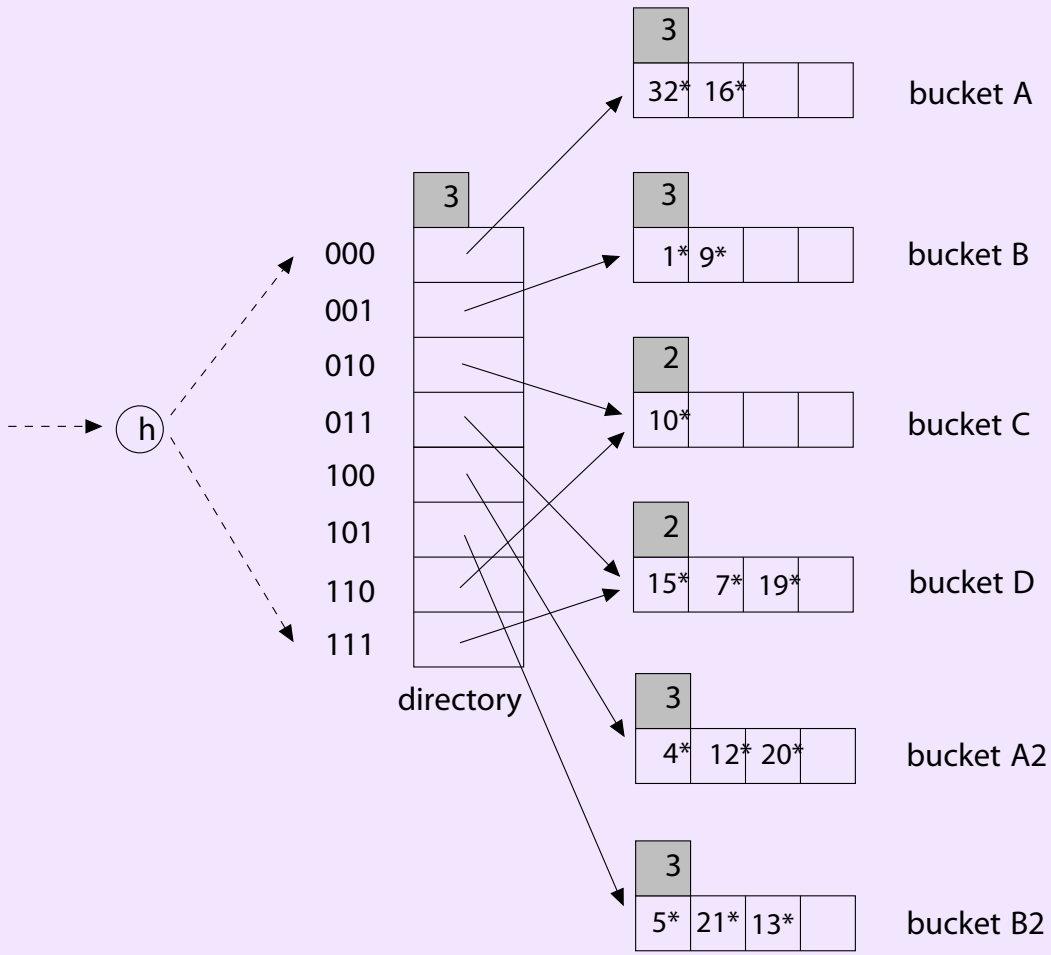
- Insert record with key k such that $h(k) = 9 = 1\underline{001}_2$.
- The associated bucket B is split, creating a new bucket B2. Entries are redistributed. New local depth of B and B2 is $\boxed{3}$ and thus does *not* exceed the global depth of $\boxed{3}$.

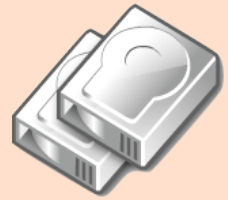
$\Rightarrow$ Modifying the directory's bucket pointer for $\underline{101}_2$ is sufficient (see following slide).



Extendible Hashing: Insert

Example (After insertion of record with $h(k) = 9 = 1001_2$)



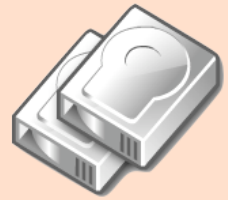


Extendible Hashing: Search Procedure

- The following `hsearch(·)` and `hinsert(·)` procedures operate over an in-memory array representation of the bucket directory $bucket[0, \ldots, 2^{\boxed{n}} - 1]$.

Extendible Hashing: Search

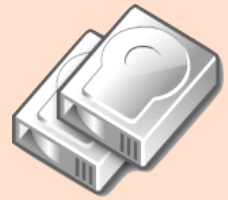
```
1 Function: hsearch(k)
2 n ← n ;                                   /* global depth */
3 b ← h(k) & (2^n − 1) ;    /* mask all but the low n bits */
4 return bucket[b] ;
```

Extendible Hashing: Insert Procedure

Extendible Hashing: Insertion

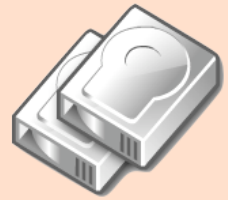
```
1  Function: hinsert(k*)
2  n ← [n] ;                                   /* global depth */
3  b ← hsearch(k) ;
4  if b has capacity then
5  |   Place k* in bucket b ;
6  |   return;
   /* overflow in bucket b, need to split                    */
7  d ← [d]_b ;                  /* local depth of hash bucket b */
8  Create a new empty bucket b2 ;
   /* redistribute entries of b including k*                 */
9  ⋮
```

Extendible Hashing: Insert Procedure (continued)

Extendible Hashing: Insertion (cont'd)

```
1  ⋮
   /* redistribute entries of b including k*                    */
2  foreach k'* in bucket b do
3      if h(k') & 2^d ≠ 0 then
4          Move k'* to bucket b2 ;
   /* new local depths for buckets b and b2                     */
5  d_b ← d + 1 ;
6  d_b2 ← d + 1 ;
7  if n < d + 1 then
       /* we need to double the directory                       */
8      Allocate 2^n new directory entries bucket[2^n, ..., 2^(n+1) − 1] ;
9      Copy bucket[0, ..., 2^n − 1] into bucket[2^n, ..., 2^(n+1) − 1] ;
10     n ← n + 1 ;
       /* update the bucket directory to point to b2            */
11     bucket[(h(k) & (2^n − 1)) | 2^n] ← addr(b2)
```

Extendible Hashing: Overflow Chains? / Delete

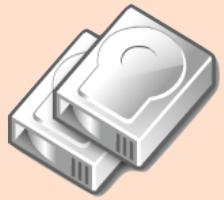
✎ Overflow chains?

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

- Deleting an entry $k*$ from a bucket may leave its bucket completely (or almost) empty.
- Extendible hashing then tries to **merge** the empty bucket and its associated partner bucket.

✎ Extendible hashing: deletion

When is local depth decreased? When is global depth decreaesed? (Try to work out the details on your own.)

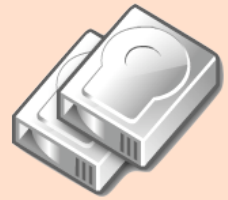


Linear Hashing

- **Linear hashing** can, just like extendible hashing, 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, however, may perform bad if the key distribution in the data file is *skewed*.

- We will now investigate linear hashing in detail and come back to the points above as we go along.

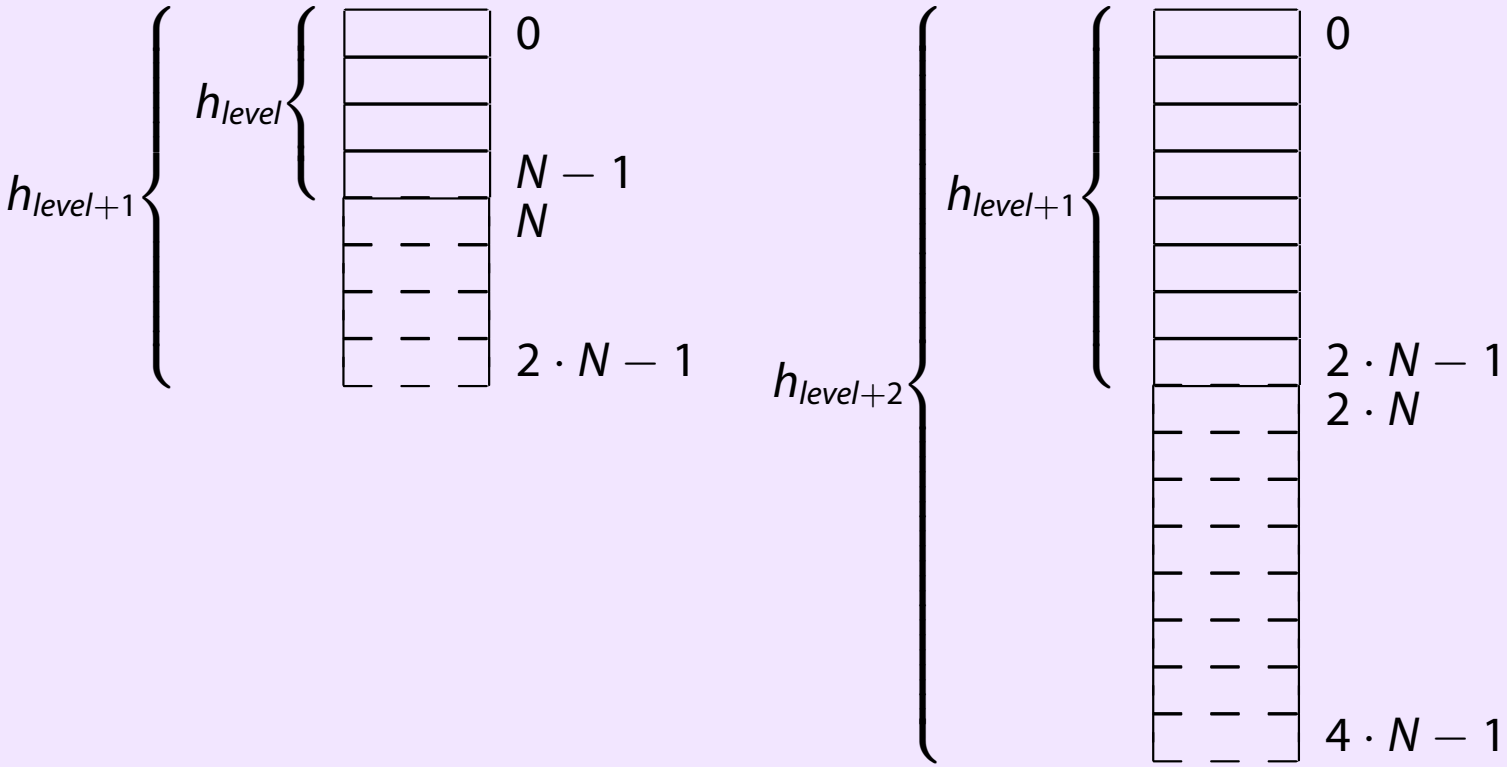
- The core idea behind linear hashing is to use an **ordered family of hash functions**, $h_0, h_1, h_2, \ldots$
  (traditionally the subscript is called the hash function's *level*).

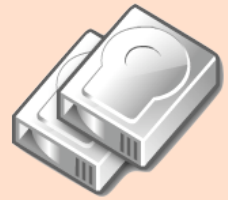


Linear Hashing: Hash Function Family

- We design the family so that the **range of $h_{level+1}$ is twice as large as the range of h_{level}** (for $level = 0, 1, 2, \ldots$).

Example (h_{level} with range $[0, \ldots, N-1]$)



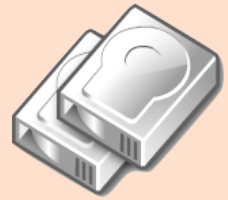


Linear Hashing: Hash Function Family

- 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, ... would be:

Hash function family

$$h_{level}(k) = h(k) \bmod (2^{level} \cdot N) \qquad (level = 0, 1, 2, \dots)$$



Linear Hashing: Basic Scheme

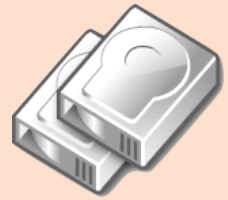
Basic linear hashing scheme

1. Initialize: $level \leftarrow 0$, $next \leftarrow 0$.

2. The **current hash function** in use for searches (insertions/deletions) is h_{level}, active hash table buckets are those in h_{level}'s range: $[0, \ldots, 2^{level} \cdot N - 1]$.

3. **Whenever** we realize that the current **hash table overflows**, *e.g.*,
   - insertions filled a primary bucket beyond c % capacity,
   - or the overflow chain of a bucket grew longer than p pages,
   - or ⟨*insert your criterion here*⟩

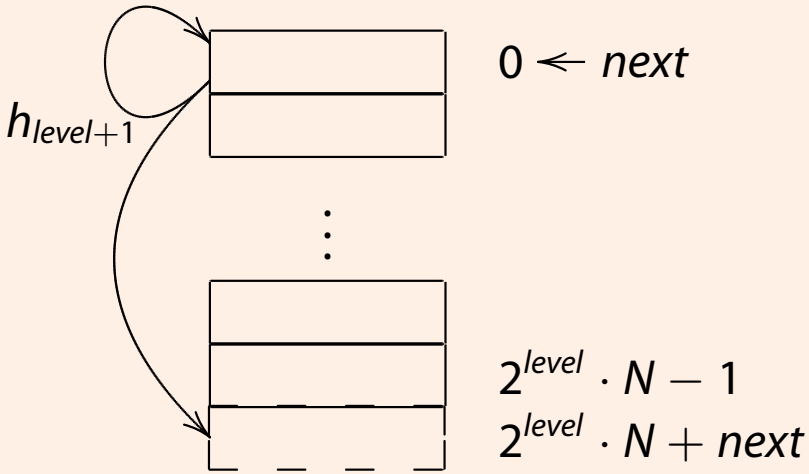
   we **split the bucket** at hash table position *next* (in general, this is **not the bucket which triggered the split!**)



Linear Hashing: Bucket Split

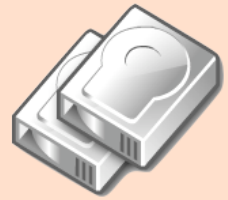
Linear hashing: bucket split

1. **Allocate a new bucket, append** it to the hash table (its position will be $2^{level} \cdot N + next$).
2. **Redistribute** the entries in bucket *next* by **rehashing** them via $h_{level+1}$ (some entries will remain in bucket *next*, some go to bucket $2^{level} \cdot N + next$). For $next = 0$:



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

$\Rightarrow$ All buckets with positions $<$ *next* have been rehashed.

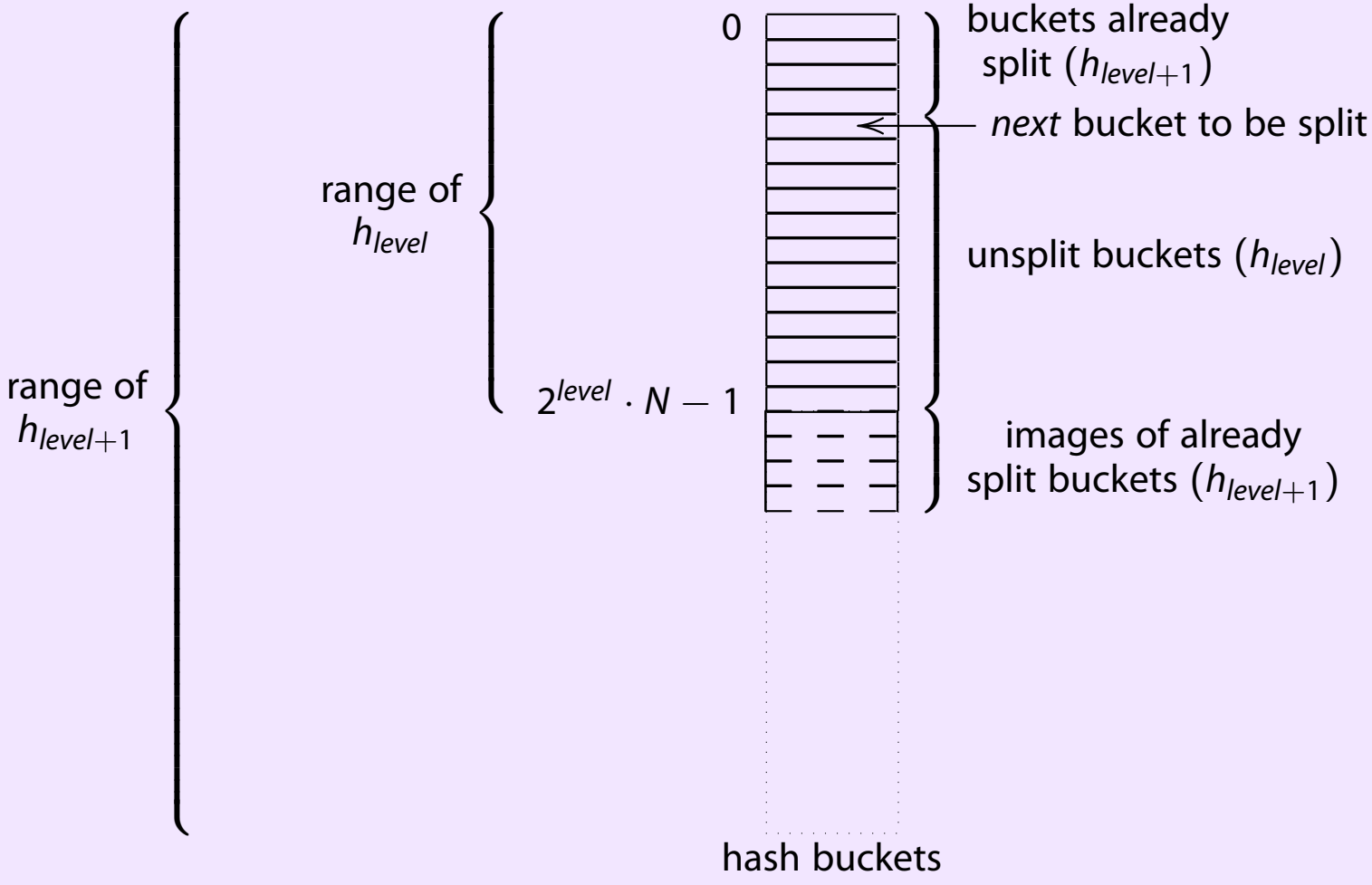


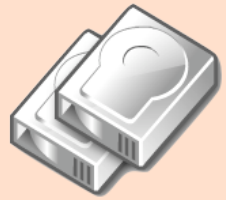
Linear Hashing: Rehashing

Searches need to take current *next* position into account

$h_{level}(k)$ $\begin{cases} < next : \text{we hit an already split bucket, } \textbf{rehash} \\ \geqslant next : \text{we hit a yet unsplit bucket, bucket found} \end{cases}$

Example (Current state of linear hashing scheme)





Linear Hashing: Split Rounds

✎ When *next* is incremented beyond 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 last current hash table position, *i.e.*

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

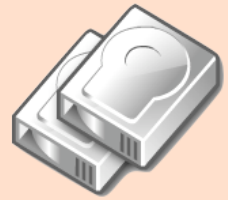
Answer:

- If $next > 2^{level} \cdot N - 1$, **all buckets** in the current hash table are hashed via function $h_{level+1}$.

$\Rightarrow$ Proceed in a **round-robin fashion**:
If $next > 2^{level} \cdot N - 1$, then

1. increment *level* by 1,
2. $next \leftarrow 0$ (start splitting from hash table top again).

- In general, an overflowing bucket is *not* split immediately, but—due to round-robin splitting—no later than in the following round.

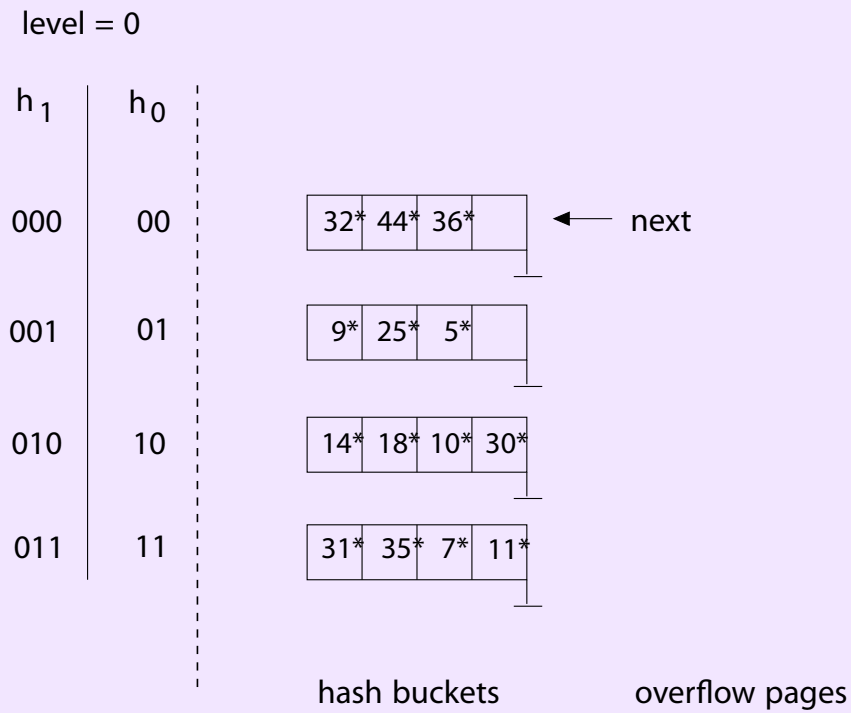


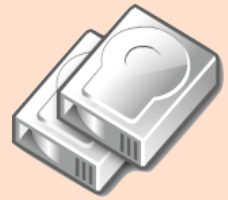
Linear Hashing: Running Example

Linear hash table setup:

- Bucket capacity of 4 entries, initial hash table size $N = 4$.
- Split criterion: allocation of a page in an overflow chain.

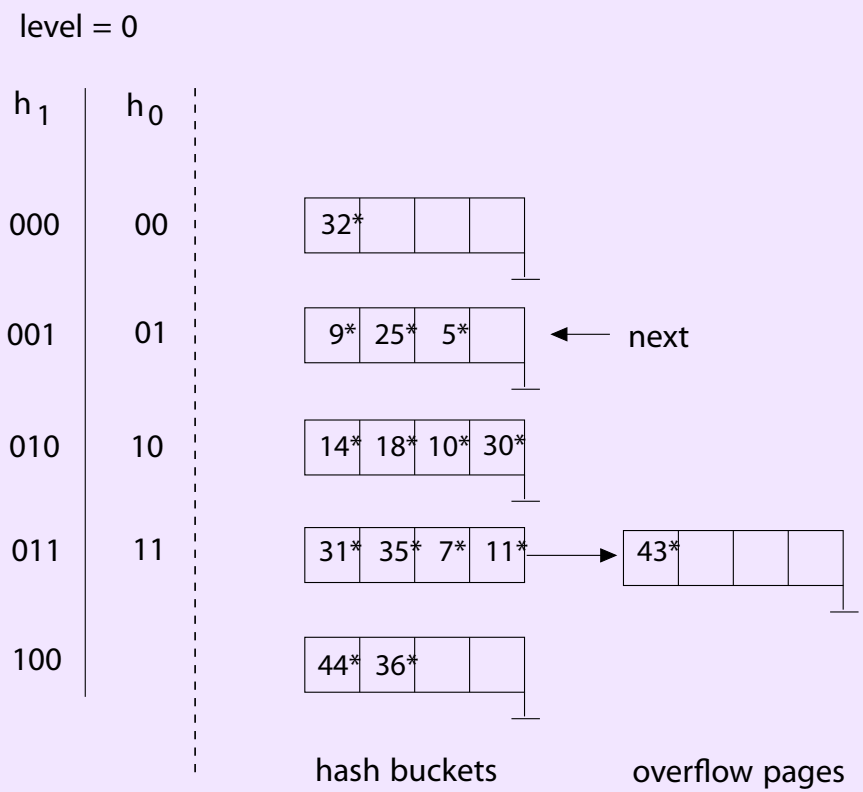
Example (Linear hash table, $h_{level}(k)*$ shown)

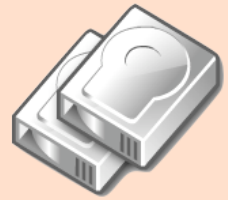




Linear Hashing: Running Example

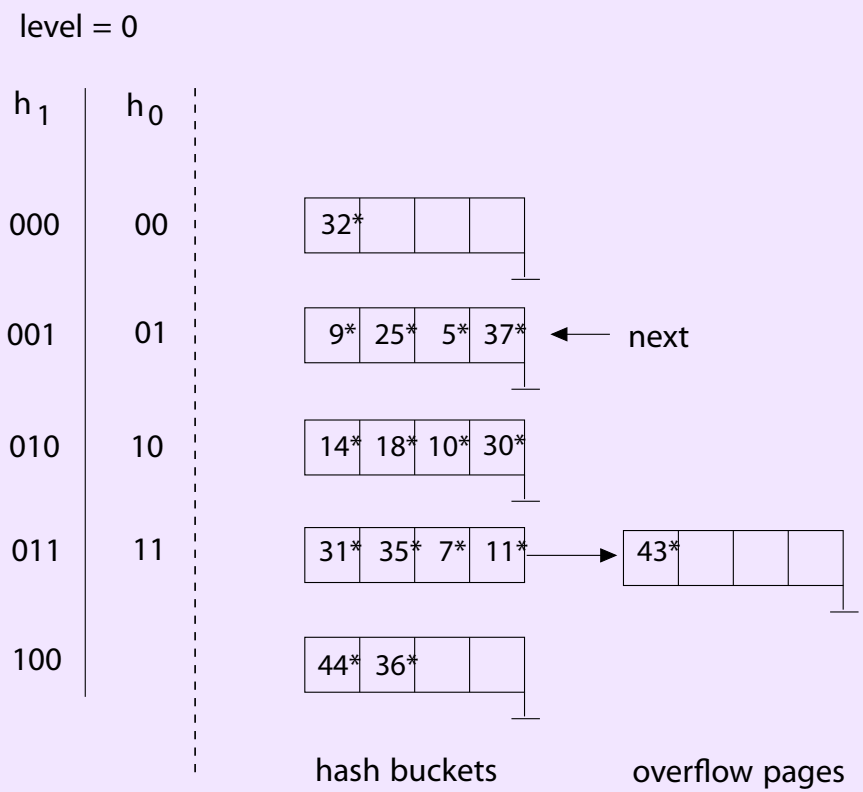
Example (Insert record with key k such that $h_0(k) = 43 = 101011_2$)

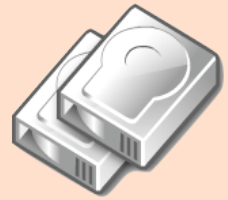




Linear Hashing: Running Example

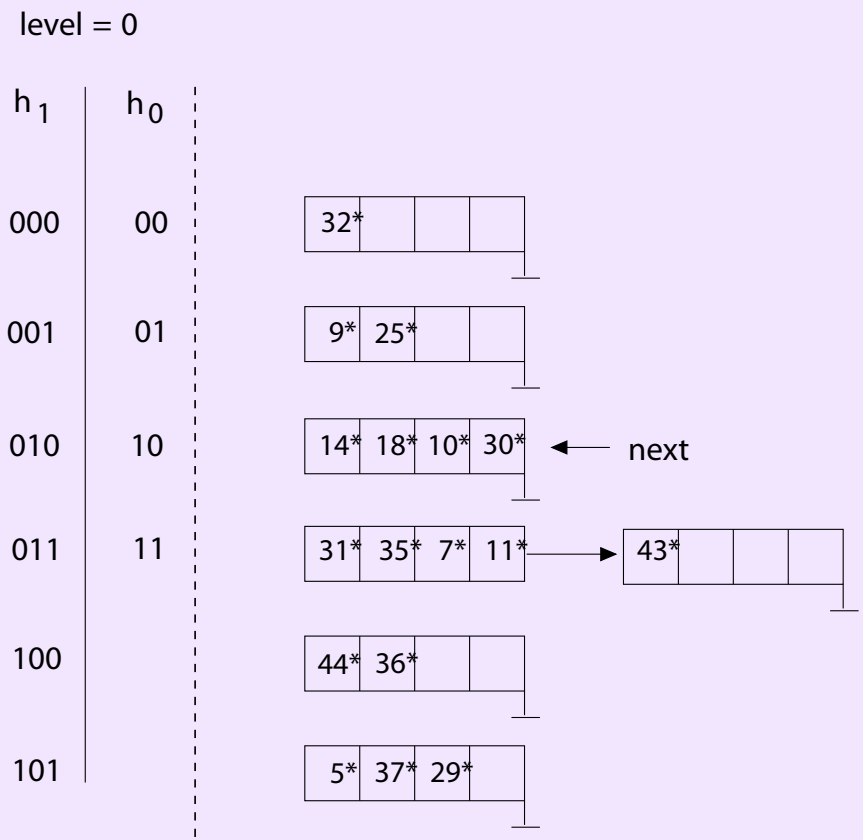
Example (Insert record with key k such that $h_0(k) = 37 = 100101_2$)

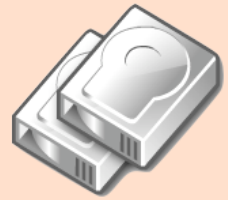




Linear Hashing: Running Example

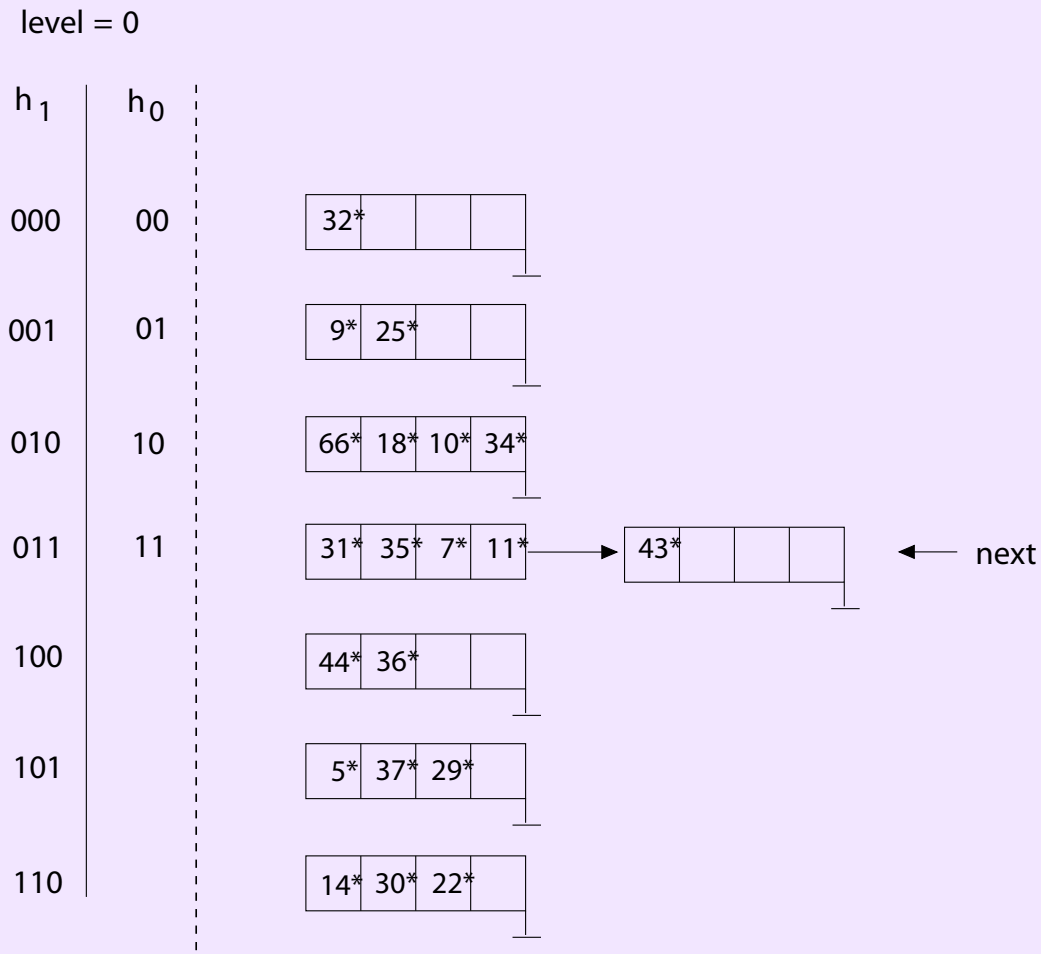
Example (Insert record with key k such that $h_0(k) = 29 = 11101_2$)

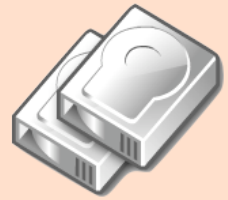




Linear Hashing: Running Example

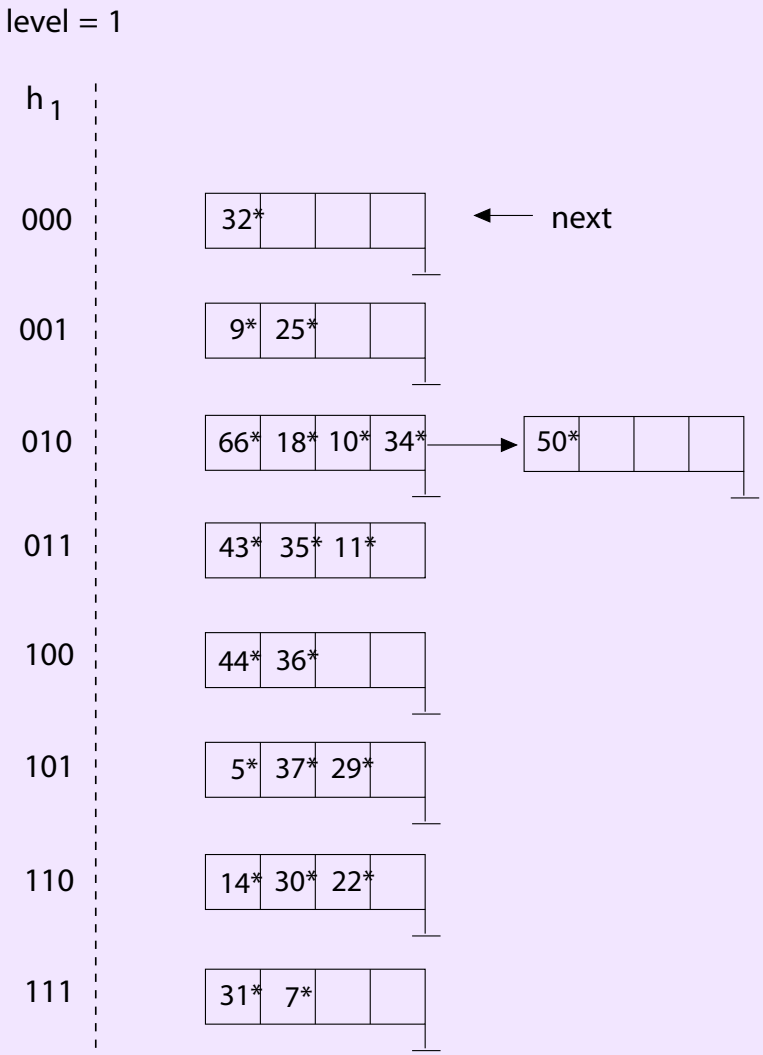
Example (Insert three records with key k such that $h_0(k) = 22 = 10110_2$ / $66 = 1000010_2$ / $34 = 100010_2$)



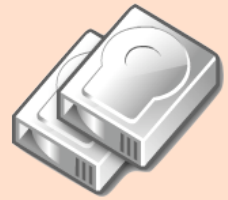


Linear Hashing: Running Example

Example (Insert record with key k such that $h_0(k) = 50 = 110010_2$)



Rehashing a bucket requires rehashing its overflow chain, too.

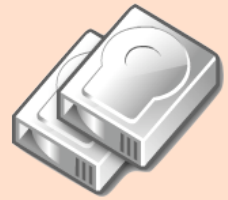


Linear Hashing: Search Procedure

- Procedures operate over hash table bucket (page) address array $bucket[0, \ldots, 2^{level} \cdot N - 1]$.
- Variables *level*, *next* are hash-table globals, N is constant.

Linear hashing: search

```
1 Function: hsearch(k)
2 b ← h_level(k) ;
3 if b < next then
      /* b has already been split, record for key k      */
      /* may be in bucket b or bucket 2^level · N − 1 + b */
      /* ⇒ rehash                                        */
4     b ← h_{level+1}(k) ;
  /* return address of bucket at position b             */
5 return bucket[b] ;
```

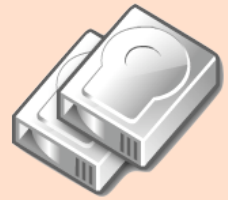



Linear Hashing: Insert Procedure

Linear hashing: insert

```
1 Function: hinsert(k*)
2 b ← h_level(k) ;
3 if b < next then
      /* rehash                                   */
4     b ← h_{level+1}(k) ;
5 Place h(k*) in bucket[b] ;
6 if overflow(bucket[b]) then
7     Allocate new page b' ;
      /* Grow hash table by one page              */
8     bucket[2^level · N + next] ← addr(b') ;
9     ⋮
```

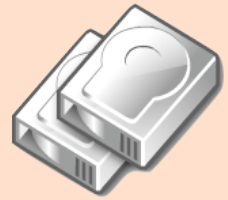
- Predicate *overflow*$(\cdot)$ is a tunable parameter: whenever *overflow*(*bucket*[*b*]) returns *true*, trigger a split.



Linear Hashing: Insert Procedure (continued)

Linear hashing: insert (cont'd)

```
1  ⋮
2  if overflow(···) then
       ⋮
3      foreach entry k'* in bucket[next] do
           /* redistribute                                   */
4          Place k'* in bucket[h_{level+1}(k')] ;
5      next ← next + 1 ;
6      /* did we split every bucket in the hash?           */
       if next > 2^level · N − 1 then
7          /* hash table size doubled, split from top      */
           level ← level + 1 ;
8          next ← 0 ;
9  return;
10
```

Linear Hashing: Delete Procedure (Sketch)

- Linear hashing deletion essentially behaves as the "inverse" of `hinsert(·)`:

Linear hashing: delete (sketch)

```
1 Function: hdelete(k)
2 ⋮
  /* does record deletion leave last bucket empty?   */
3 if empty(bucket[2^level · N + next]) then
4     Remove page pointed to by bucket[2^level · N + next] from hash
      table ;
5     next ← next − 1 ;
6     if next < 0 then
          /* round-robin scheme for deletion          */
7         level ← level − 1 ;
8         next ← 2^level · N − 1 ;
9 ⋮
```

- Possible: replace *empty*(·) by suitable *underflow*(·) predicate.