HASH-BASED INDEXES

Chapter 11

Introduction

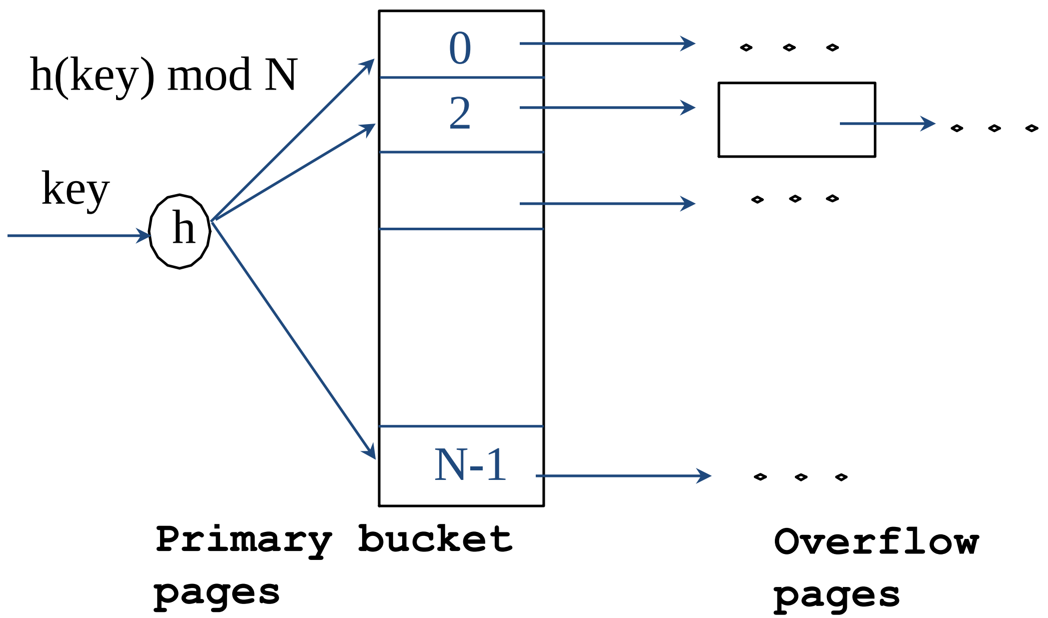
- *As for any index, 3 alternatives for data entries* **k** :
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
  - Choice orthogonal to the *indexing technique*

- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.

- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Static Hashing (Contd.)

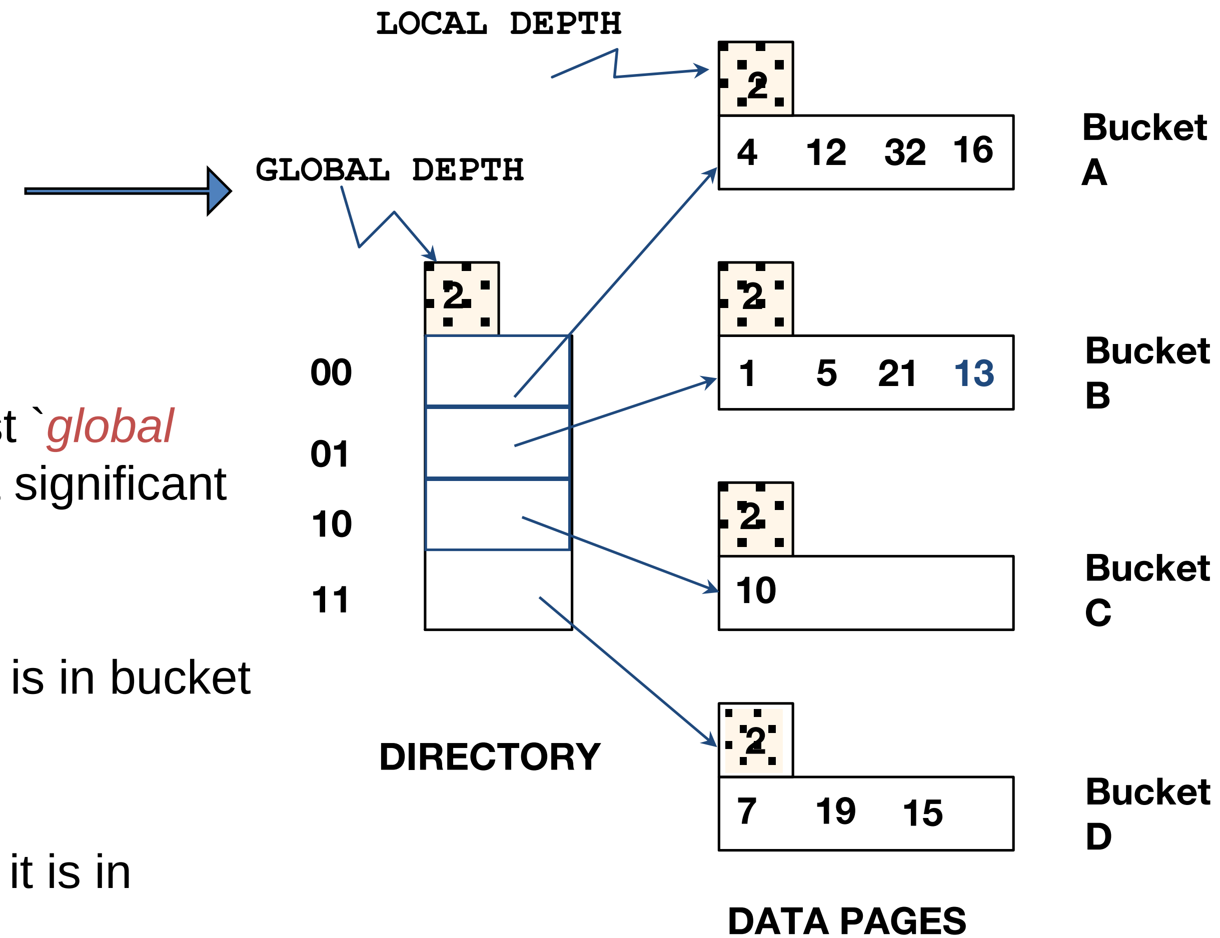
- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r.*
- Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full.
- Why not re-organize file 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 page*!
  - Trick lies in how hash function is adjusted!

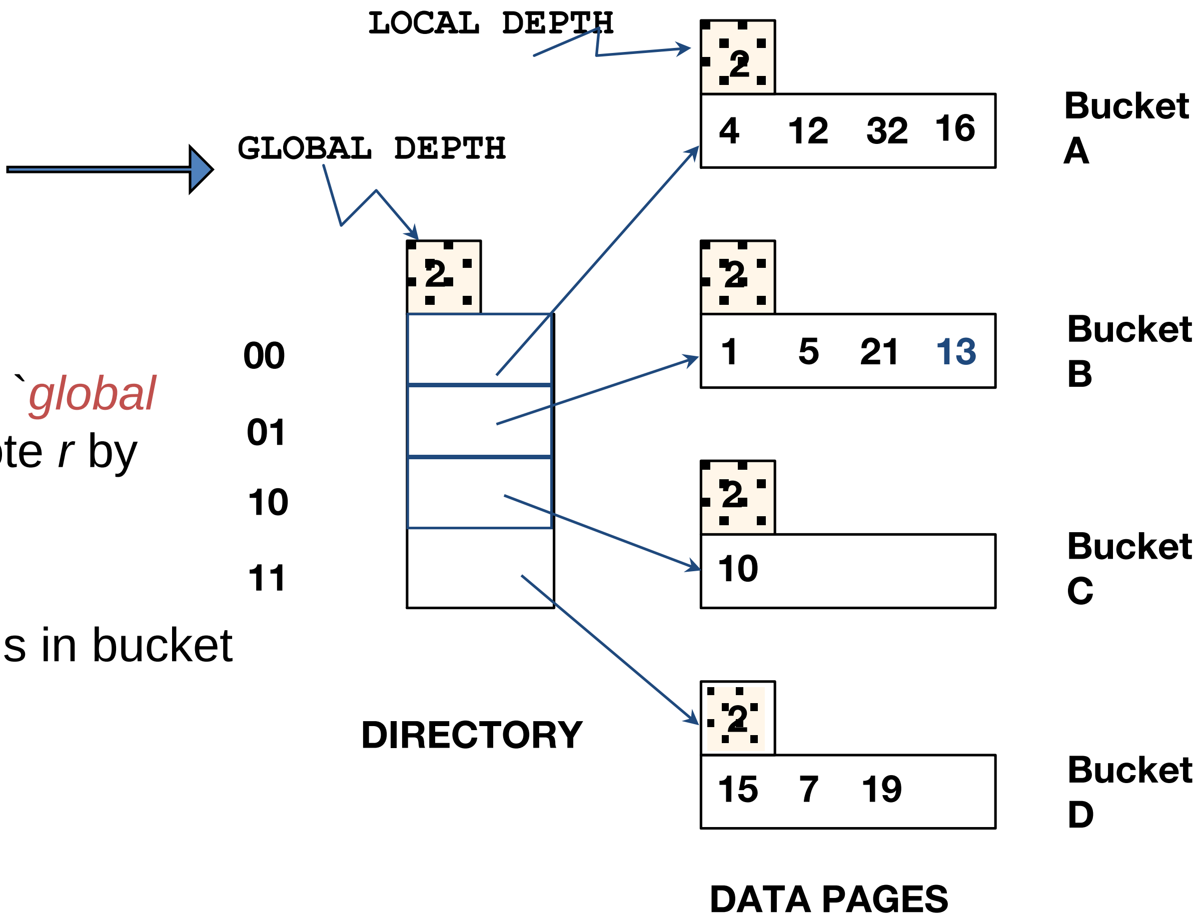
Example

- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*) (or least significant bits) ; we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
  - if **h**(r) = 15 = binary 1111, it is in bucket pointed to by ???
  - if **h(**r) = 16 = binary 10000, it is in bucket pointed to by ???



Example

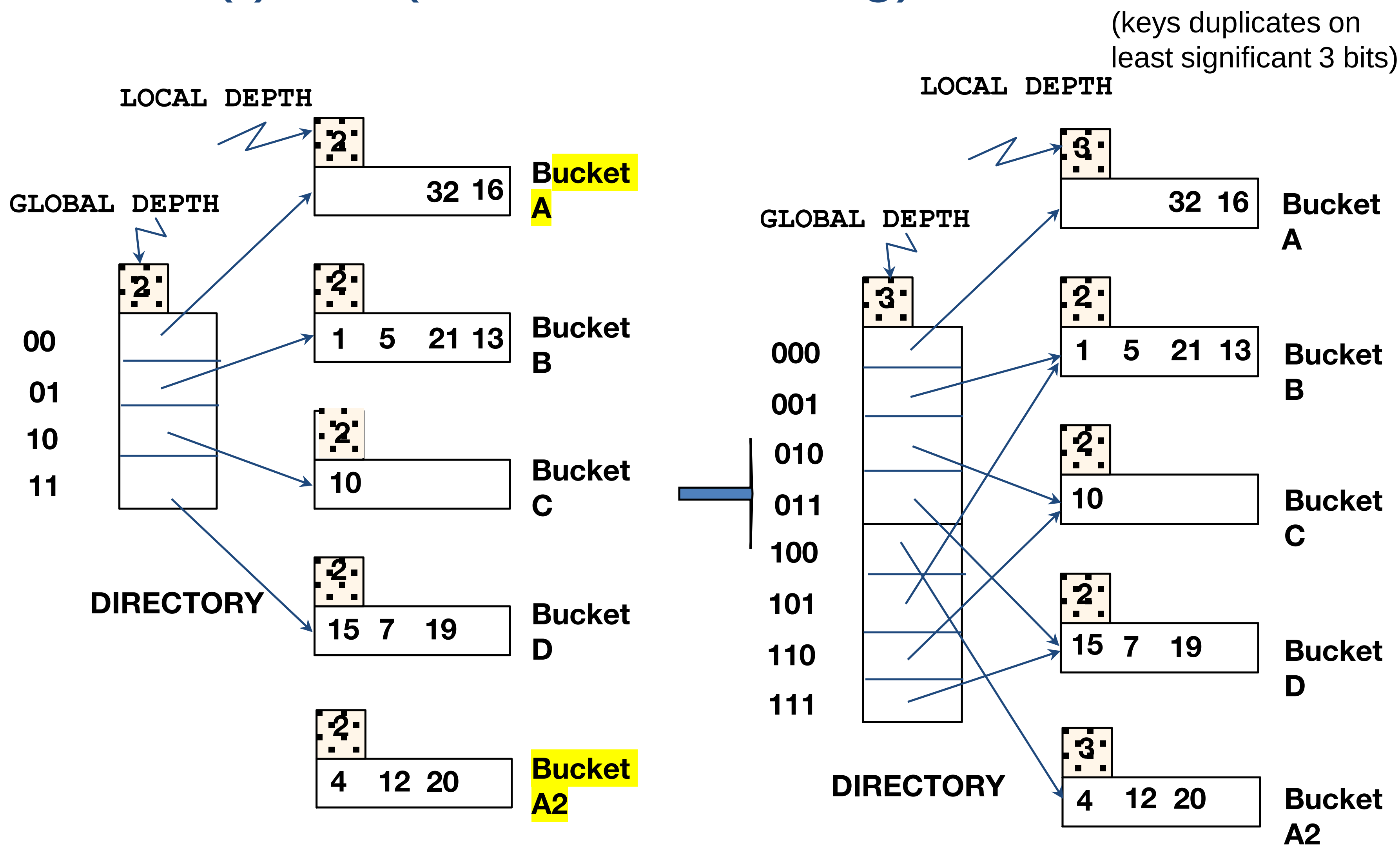
- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.



❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert **h**(r)=20 (Causes Doubling)



Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **<u>3</u>** bits needed to tell which.
  - *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.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

What is Linear Hashing?

- Linear Hashing is a form of dynamic hashing scheme (an alternative to Extendible Hashing);
- What makes Linear Hashing different from other schemes:
  - There is no directory required;
  - capable of handling long overflow chain;
  - more flexible with respect to the timing of bucket splits;
  - allows you to grow one slot at a time.

Linear Hashing

Terminology	Description
h0, h1, h2, ….	A family of hash functions, where each function's range is twice of its predecessor
N	Initial number of buckets
d0	The number of bits to represent N
Level	Indicate the number of split cycle completed, initially 0
Next	Pointer to the next bucket inline to be split, initially points to the first bucket in the table.

- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

Linear Hashing (Cont.)

Terminology	Description
h0, h1, h2, ….	A family of hash functions, where each function's range is twice of its predecessor
N	Initial number of buckets ($N = 2^{d0}$)
d0	The number of bits to represent N
Level	Indicate the number of split cycle completed, initially 0
Next	Pointer to the next bucket inline to be split, initially points to the first bucket in the table.

- If we decide that N (number of buckets) = 4, then lets compute **d0**
  - Since $N = 2^{d0}$, it means that $4 = 2^{d0}$, hence d0 must equal 2
  - If Level = 0, this tells us that we must look at the last two bits when adding / searching for a key in the index.
  - If Level = i, then we use $di = d0 + i$
  - Will define what we mean by level later. We will start with Level = 0

Linear Hashing (Cont.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.

Linear Hashing (Contd.)

- How to do a search on the index???

  - Will need to maintain **Level & NEXT**
  - Use hash-function $h_i(key) = h(key) \bmod (2^i N)$

  - **Search:** To find bucket for data entry *r,* find $\mathbf{h}_{Level}(r)$*:*
    - *Hash-function then is :* $h_0(r) = h(r) \bmod (2^0 N) = r \bmod 4$

      Example : $h_0(5) = 5 \bmod 4 = 1 \Rightarrow 01$

      $h_0(44) = 44 \bmod 4 = 0 \Rightarrow 00$

      $h_0(30) = 30 \bmod 4 = 2 \Rightarrow 10$

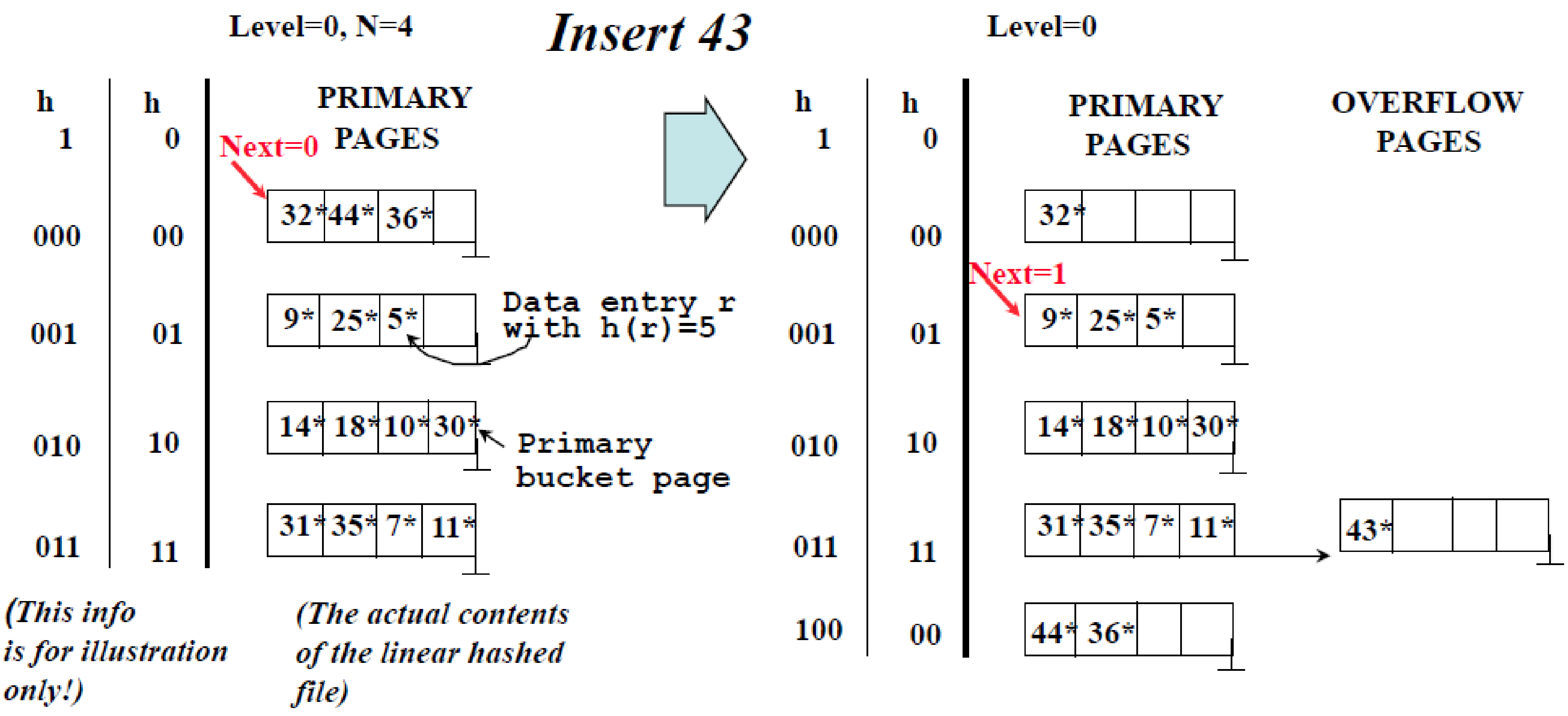
    - Once we have value, then:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to *N'*, *r* belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N$*;* must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Linear Hashing (Contd.)

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- Can choose any criterion to `trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

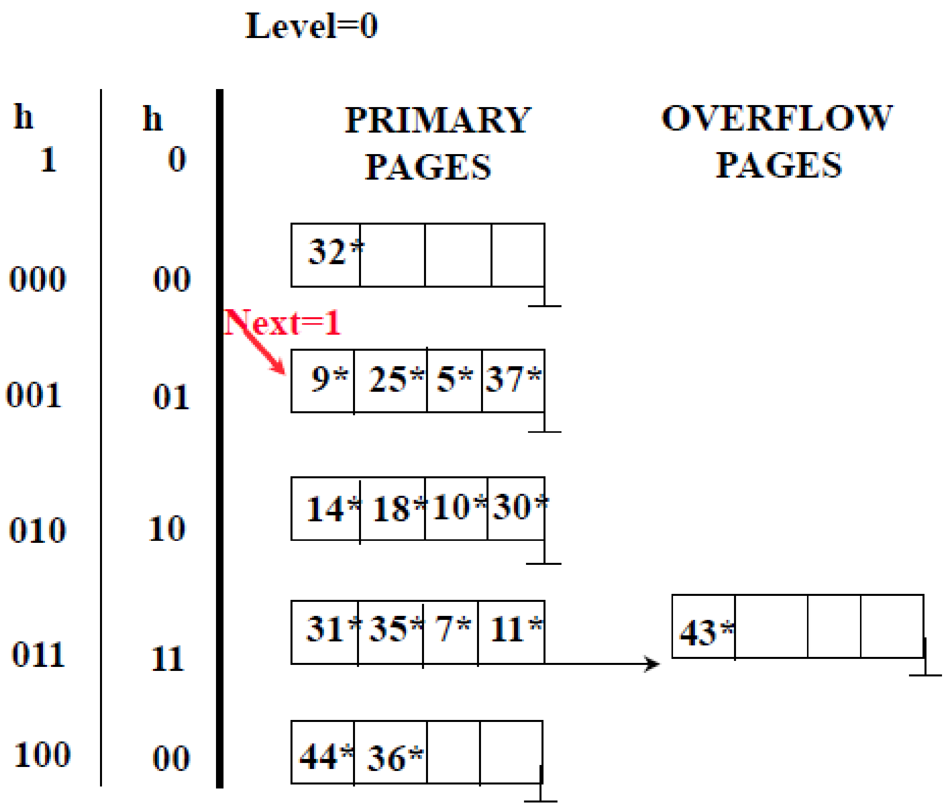
Example of Linear Hashing

- On split, **h**$_{Level+1}$ is used to re-distribute entries.



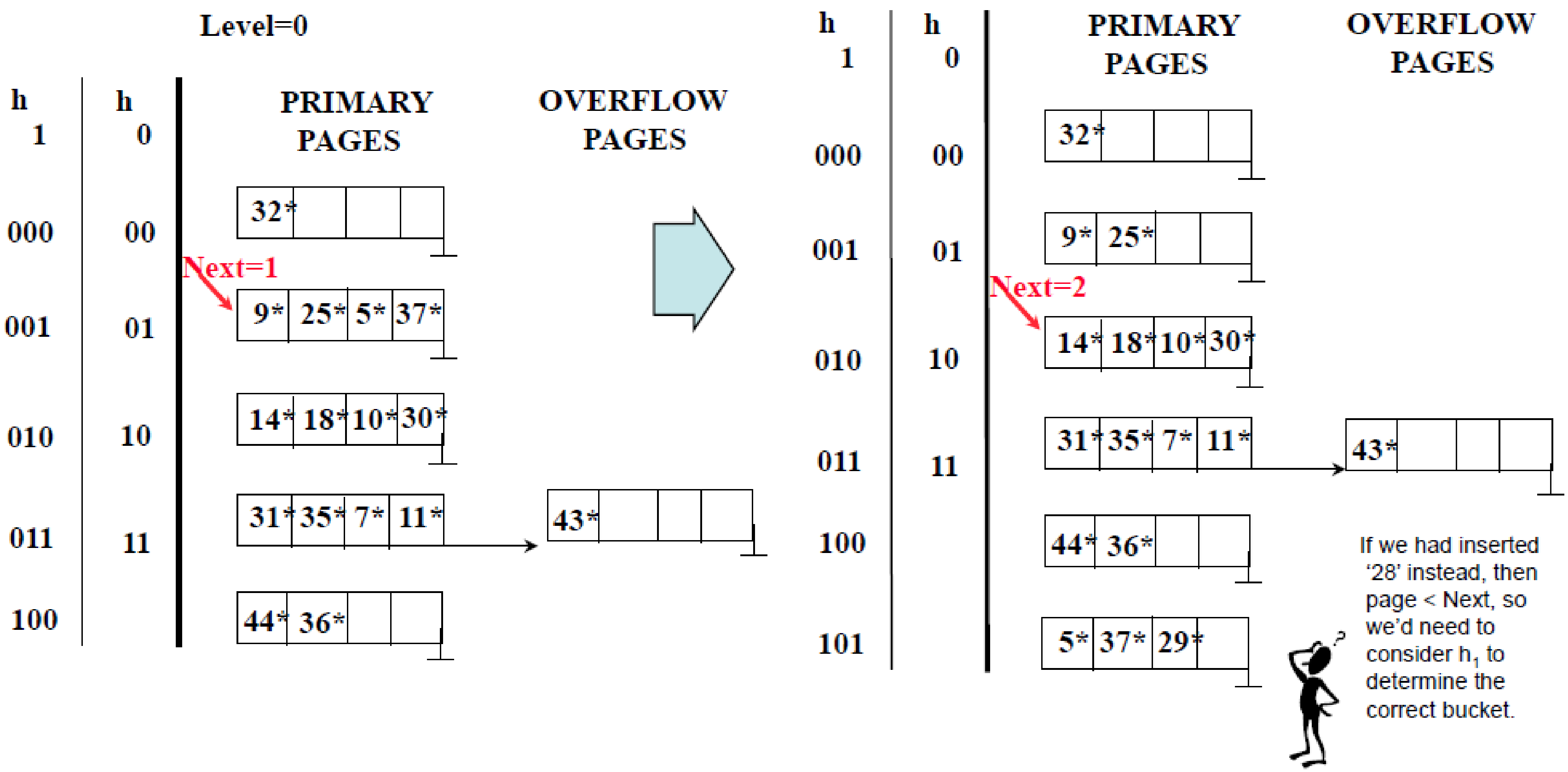
Insert 37 (00100101)

- References page ≥ "Next", check h_0, fits, no action



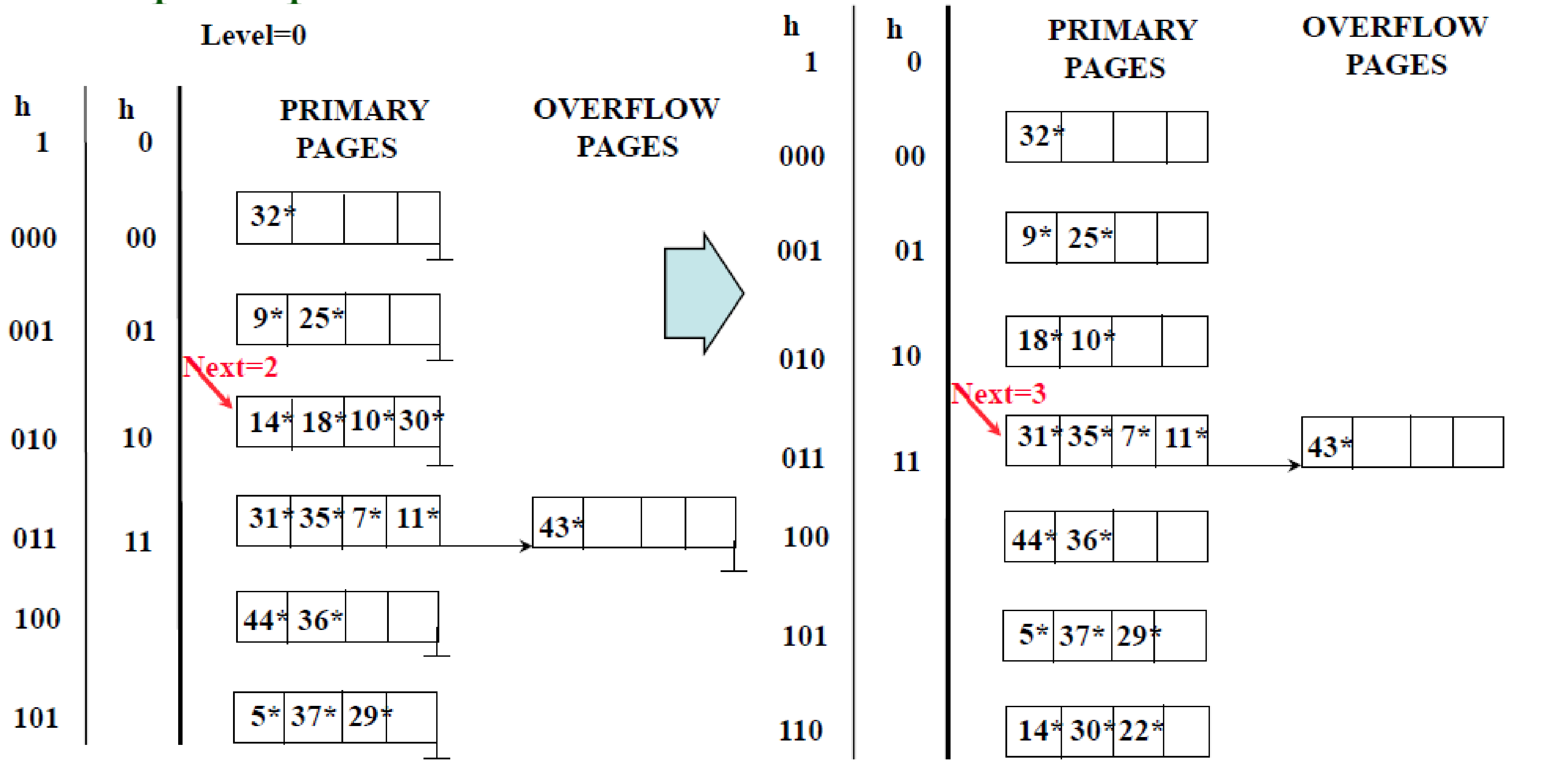
Insert 29 (00011101)

- References page ≥ "Next", check h_0, fits, no action
- Spill, split, move next

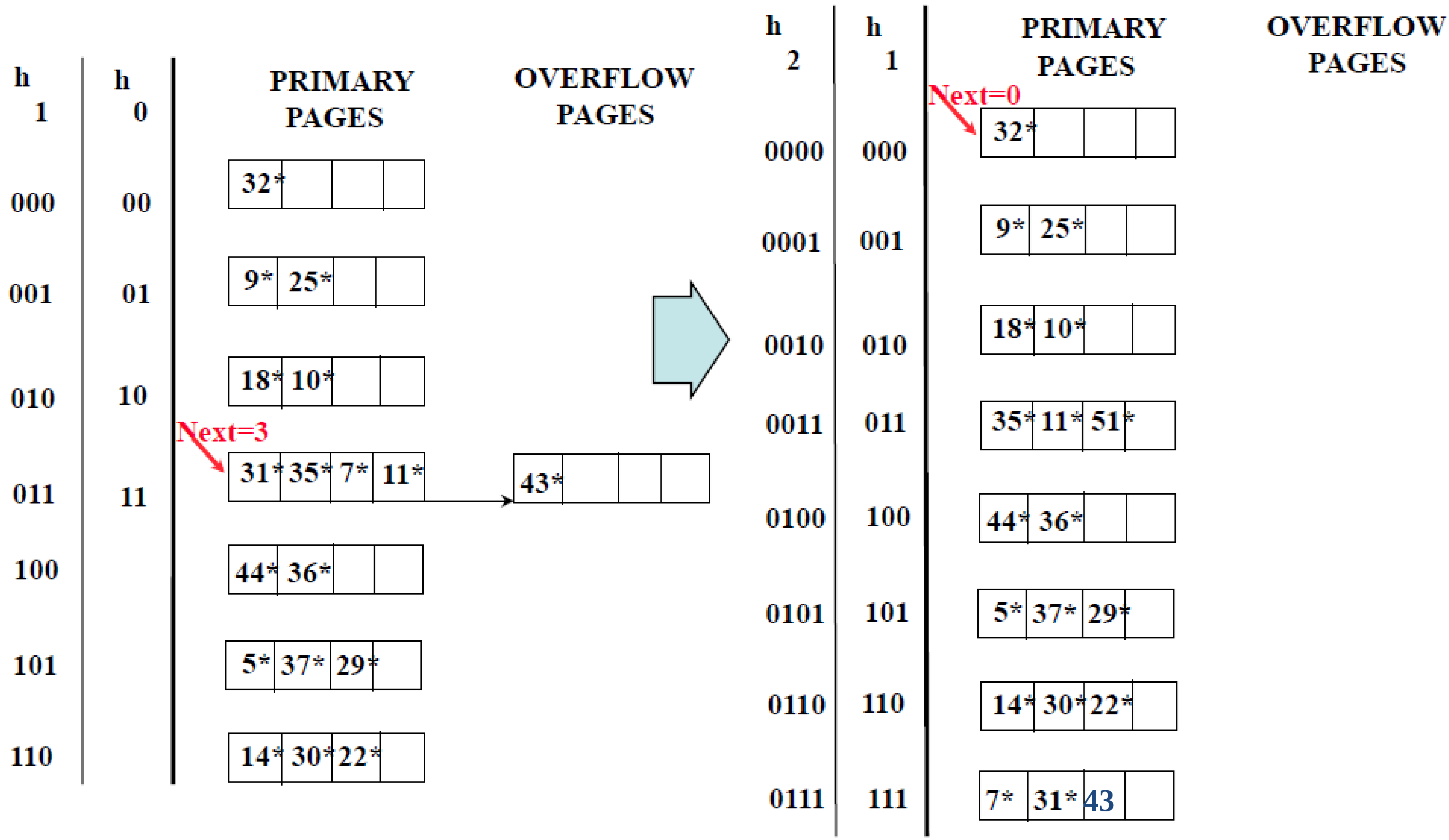


Insert 22 (00010110)

- References page ≥ "Next", check h_0, fits, no action
- Spill, split, move next



Insert 51 (00110011)



LH Described as a Variant of EH

- The two schemes are actually quite similar:
  - Begin with an EH index where directory has *N* elements.
  - Use overflow pages, split buckets round-robin.
  - First split is at bucket 0. (Imagine directory being doubled at this point.) But elements <1,*N*+1>, <2,*N*+2>, ... are the same. So, need only create directory element *N*, which differs from 0, now.
    - When bucket 1 splits, create directory element *N*+1, etc.

- So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding i'th is easy), we actually don't need a directory! Voila, LH.

Summary

- 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. (*Duplicates may require overflow pages.*)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!

Plan for next lecture

Lets do worksheet exercises next lecture to practice

- B+ Trees
- Extensible Hashing
- Linear Hashing