Hash-Based Indexes

Chapter 11 Ramakrishnan & Gehrke
(Sections 11.1-11.4)

What you will learn from this set of lectures

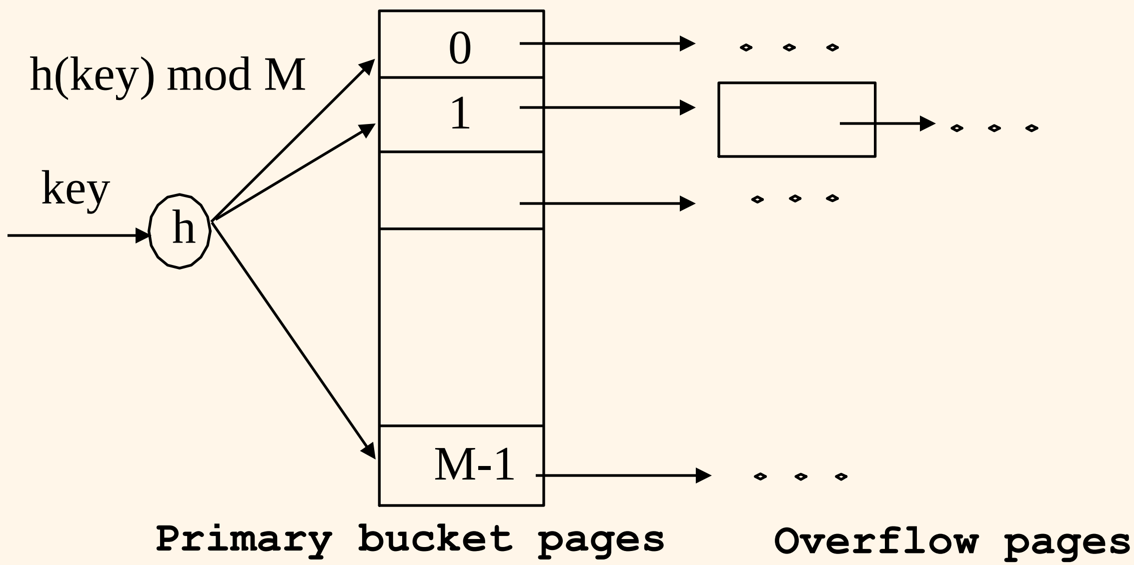
- Review of static hashing
- How to adjust hash structure dynamically against inserts and deletes?
  - Extendible hashing
  - Linear hashing.
  - Relative strengths of B+trees and Hashing: when to use what.

Introduction

- *Hash-based* indexes are best for *equality selections*
  - no traversal; direct computation of where **k*** should be
  - ***cannot*** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees, on a certain level.

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* *(i.e., k*)* belongs. (M = # of buckets)



Static Hashing (Contd.)

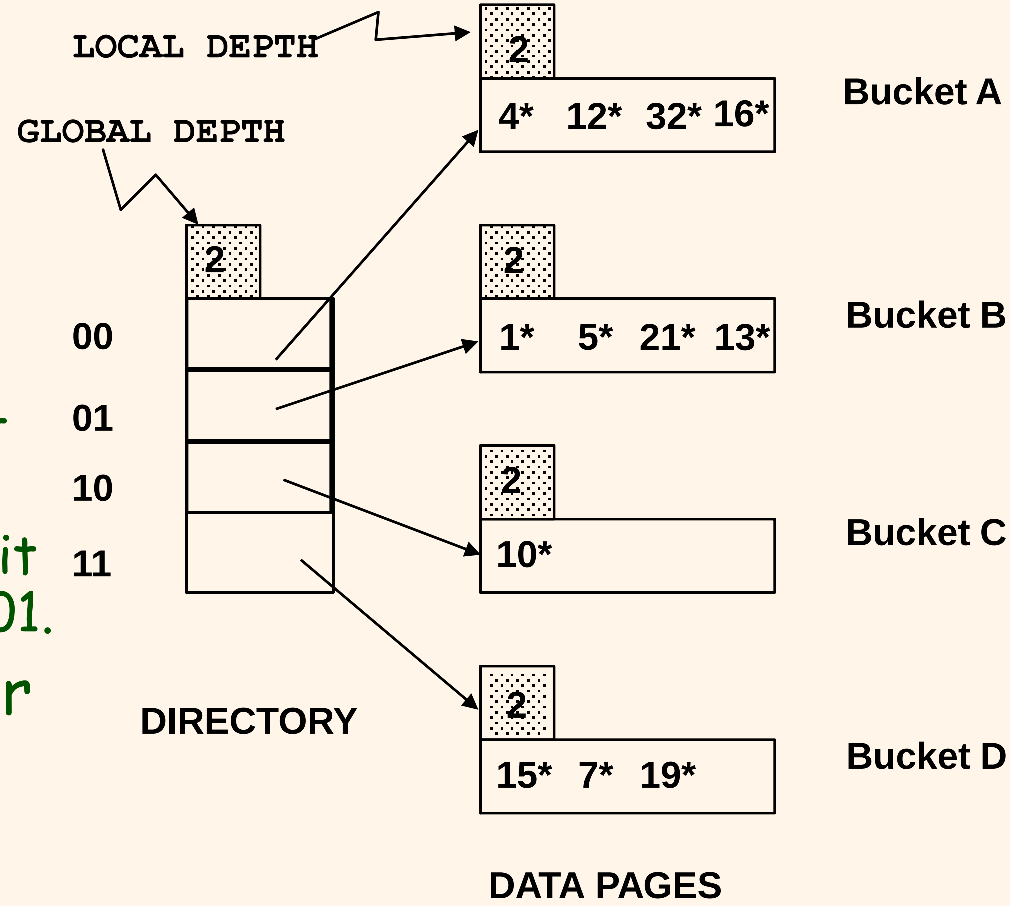
- Buckets contain *data entries*.
- Bucket size could be more than 1 block.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (*key* mod M) usually works well for prime M.
  - lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance (when there are updates).
  - *Extendible* and *Linear Hashing*: two major 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!
    - and is needlessly prodigal on resource use.
  - *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!

†Not always necessary!

Example



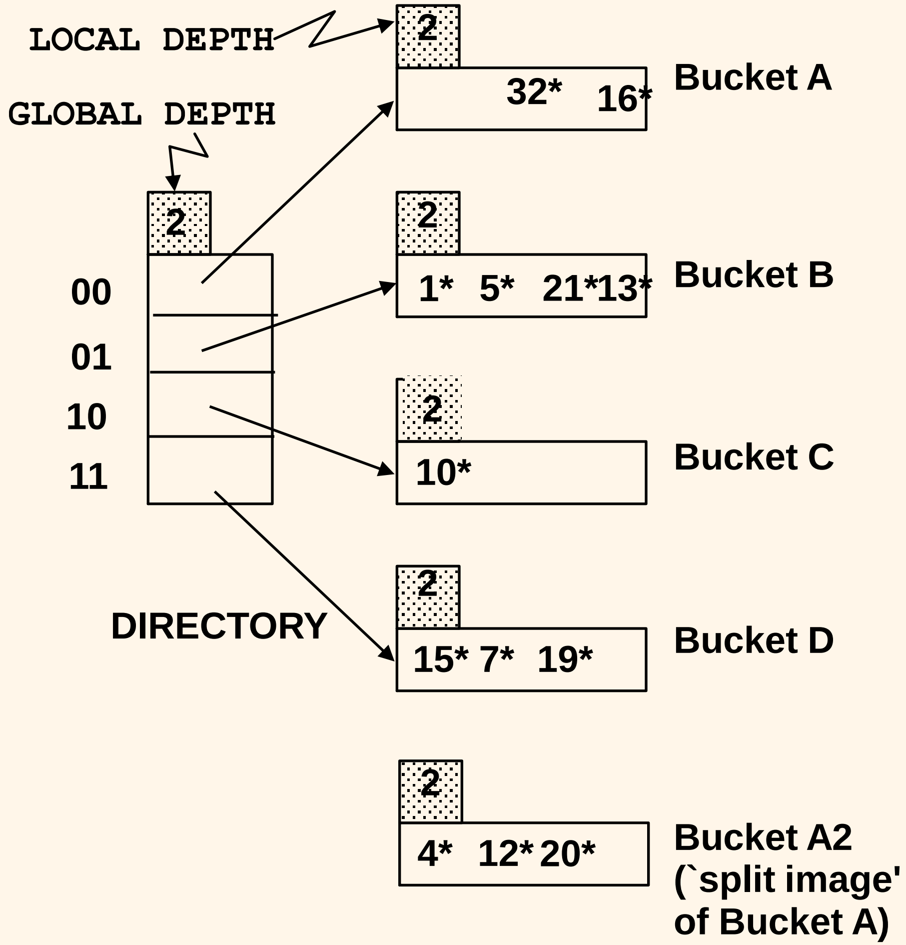
- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*)
  - e.g., **h**(*r*) = 5 = binary 1<u>01</u>, it is in bucket pointed to by 01.
- hash fn used: h(k) = k (for illustration only).

- <u>**Insert**</u>: If bucket is full, <u>*split*</u> it (*allocate new page, re-distribute data entries*). E.g., consider insert 20*.
- *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.)

Example – Remarks.

- Depth – deals with how many bits from the hash address suffix we examine at a given time.
- Global depth = what's the #bits needed to correctly find the home bucket for an arbitrary data entry, in general?
- Local depth of bkt B = how many bits did I really need to look at to get to bucket B?
- Global depth >= local depth.
- Check this on examples.
- Is this possible: GD > all LDs?

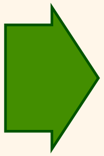
Insert h(r)=20 - Part 1

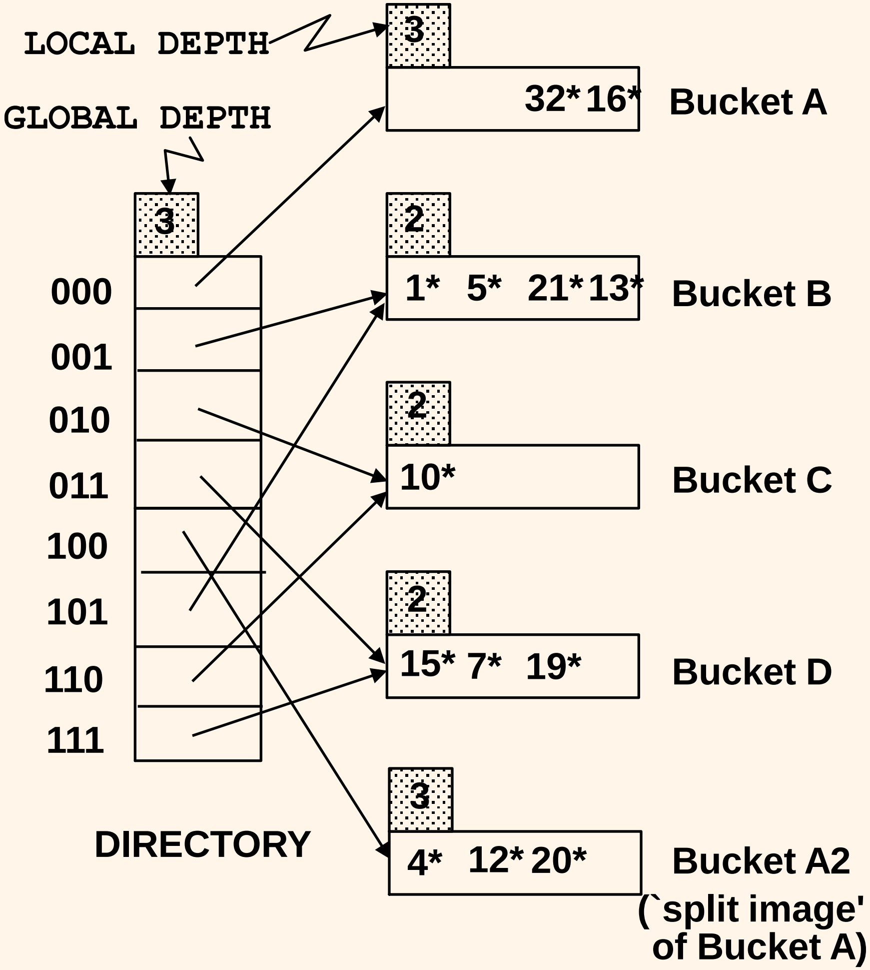


- Suppose h(*k*) = *k* for this example.
- Bucket A split into 2 using an extra bit, i.e., 3 bits
- A divisible by 8, i.e., 1000
- A2 divisible by 4, i.e., 100
- note that *only one bucket needs to be re-distributed*, i.e., re-hashed
- B, C, D remain unchanged
- Where to link A2?

Insert h(r)=20 - Part 2

- double the directory
- add 1 to global depth & to local depth of A/A2.
- now can distinguish between A and A2
- notice the difference in local depth between buckets
- multiple pointers to the same bucket
- Review properties of LD & GD.





Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in *A* or *A2*. Last **3** bits needed to tell which.
  - *Global depth of directory*: min # of bits needed to tell which bucket an entry belongs to = max{local depths}.
  - *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!)
- What happens when 9* is inserted?

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K page; contains 1,000,000 records (as data entries); 40 records/page ⇒ $10^6/40 = 25{,}000$ pages of data entries; as many directory elements; can handle using 15bit addresses; 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.
- **Delete**: If removal of data entry makes bucket empty,
  - check to see whether all `split images' can be merged
  - if each directory element points to the same bucket as its split image, can halve directory
  - rarely done in practice (e.g., leave room for future insertions).

Linear Hashing

- An extension to Extendible Hashing, in spirit.
- LH tries to avoid the creation/maintenance of a directory.
- <u>*Idea*</u>: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - N = initial # buckets = 2^{d0}
  - **h** is some hash function (range is *not* 0 to N-1)
  - $\mathbf{h}_i$ 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)
  - e.g., **h** = binary representation, *d0 = 2, d1 = 3, d2 = 4, ...*

Overview of LH File

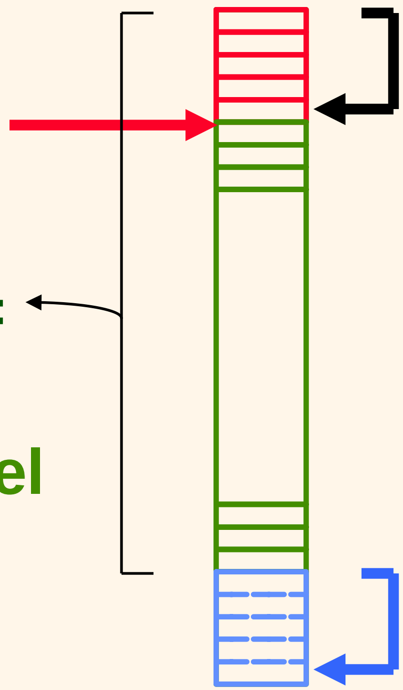
- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
- Note: bucket split need ***not*** be bucket where insertion and/or overflow occurred.
  - Next – pointer to current bucket, i.e., next bucket likely to be split.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round *R*) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R-1 yet to be split.
  - Current round number is *Level*.
  - Level and R used interchangeably.

Overview of LH File (Contd.)

- In the middle of a round.

Buckets to be split **Next**

Buckets that existed at the beginning of this round: this is the range of h_{Level}

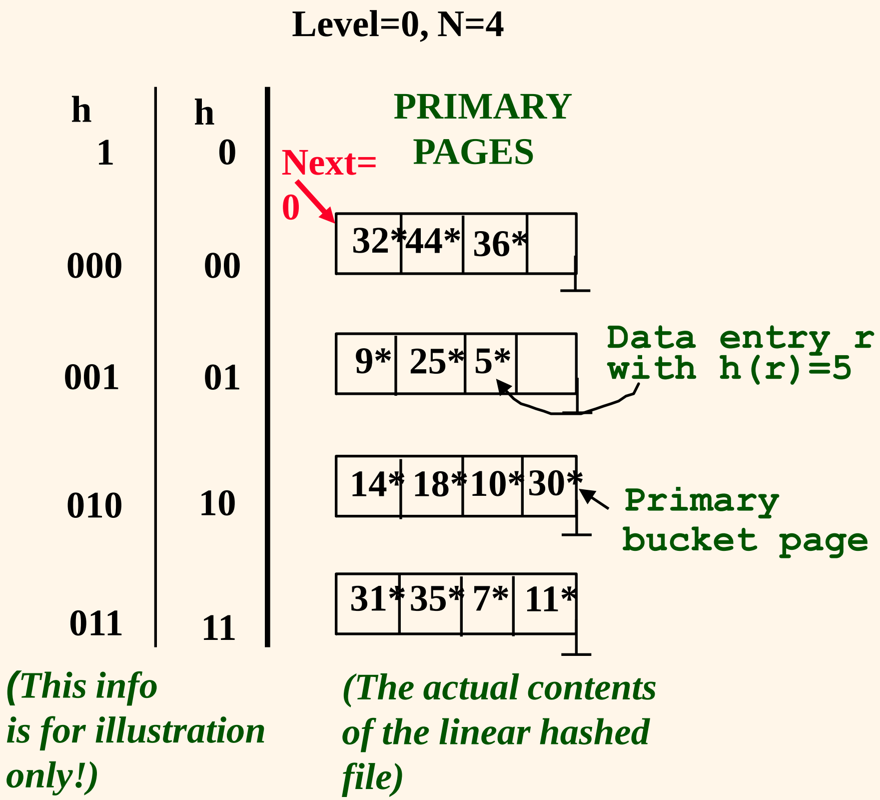


Buckets split in this round: If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in `split image' bucket.

`split image' buckets: created (through splitting of other buckets) in this round

Level = R.

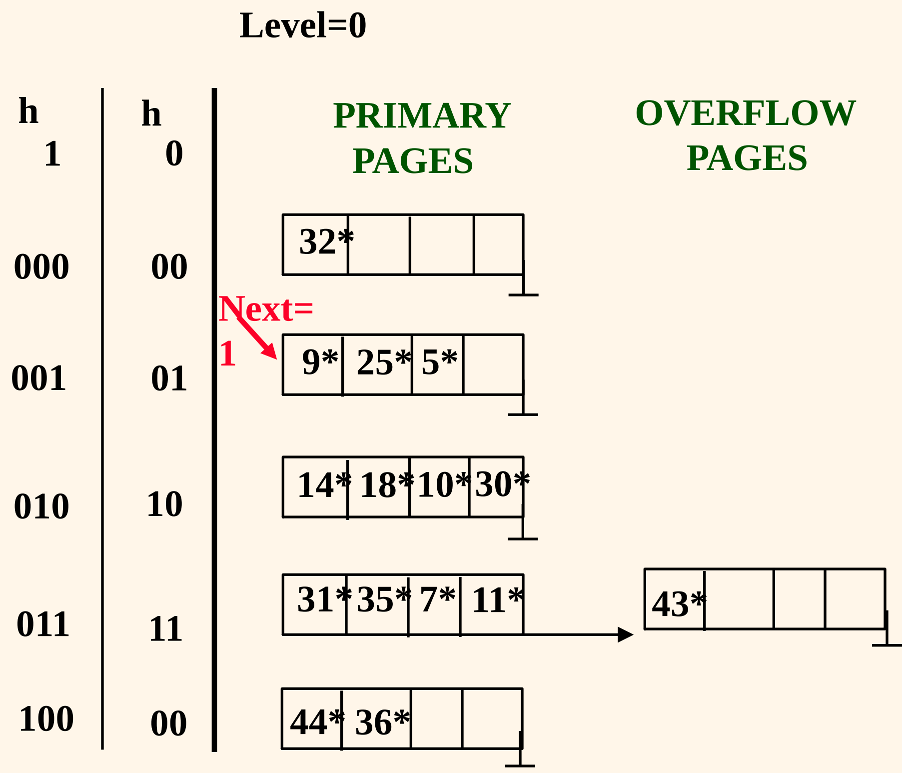
Example of Linear Hashing



- starts with 4 buckets
- all buckets to be split in a round-robin fashion, starting from the first one

Example - Inserting 43*

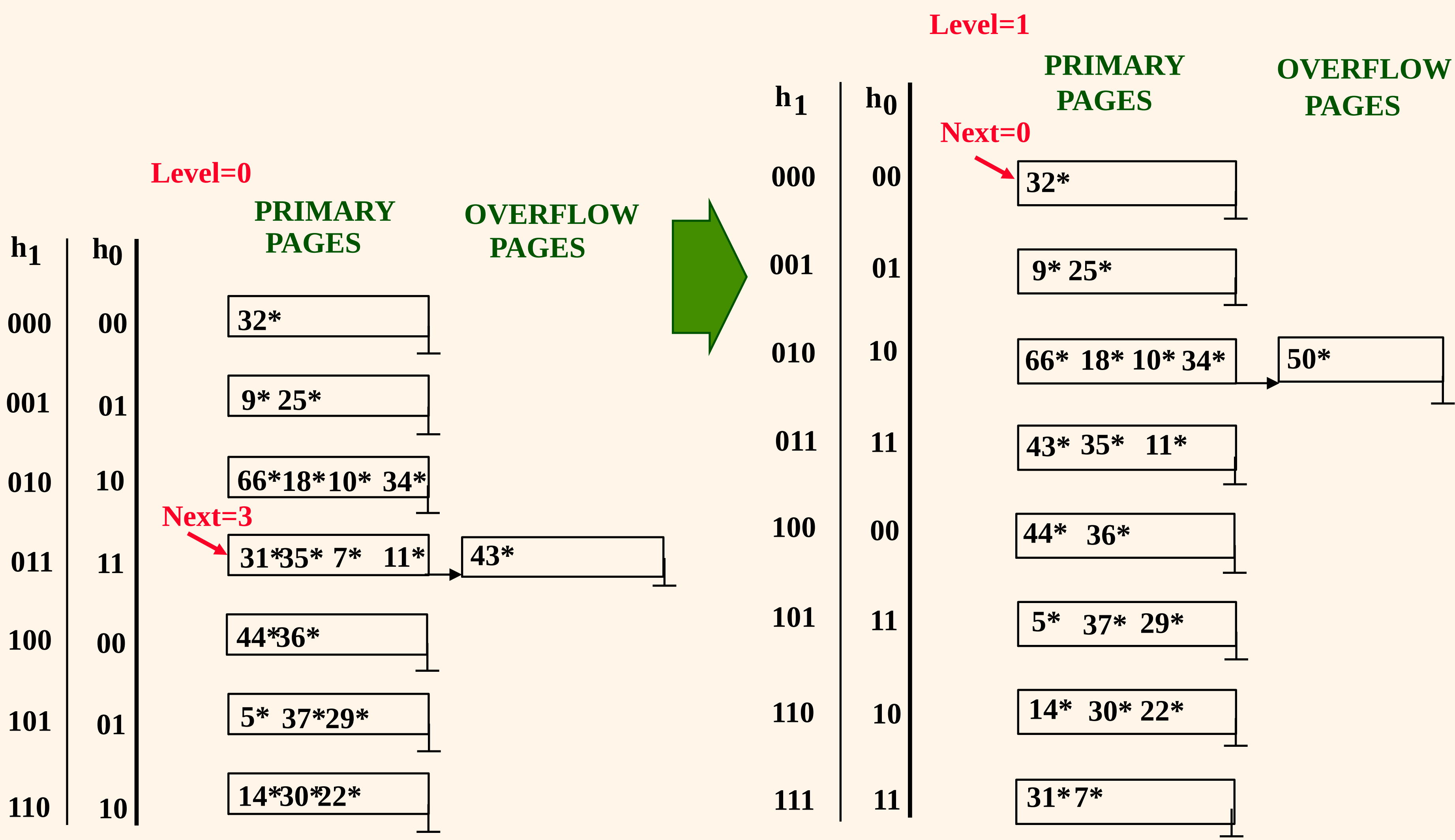
- h_0 (43) = 11 => overflow
- overflow page exists!
- splitting occurs but to the *Next* bucket



Linear Hashing - insertions

- **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!

Example: End of a Round (Inserting 37*,29*, 22*,66*,34*,50*)



Linear Hashing - Searching

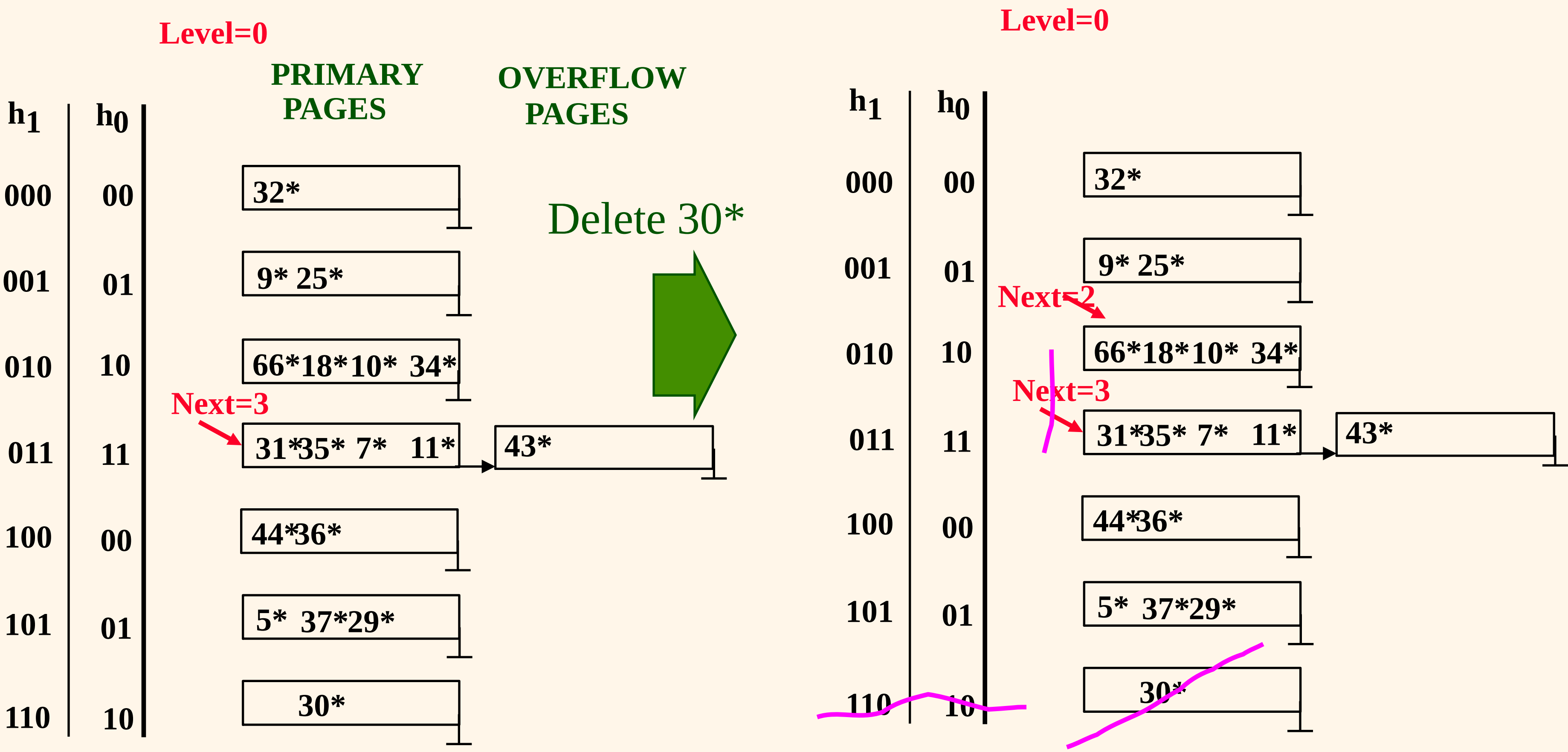
- **Search:** To find bucket for data entry *r*, find $\mathbf{h}_{Level}(r)$:
  - If $\mathbf{h}_{Level}(r)$ in range \`*Next* to N_R-1', *r* belongs here.
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

LH – Deletion

- Inverse of insertion.
- If last bkt is empty, remove it and decrement Next.
- More generally, can combine last bkt with its split image even if non-empty. Criterion may be based on bkt occupancy level.

LH – Deletion (example)

After deleting 14*, 22*



Summary

- Hash-based indexes: best for equality searches.
- Static Hashing can lead to long overflow chains.
- EH avoids overflow pages by splitting a full bucket when a new data entry is to be added to it.
  - 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.
- LH avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.