Hash-Based Indexes

Midterm moved to 10/6 to accommodate Grace Hopper Conf.

Stay on top of PS#3

Introduction

- *Hashing maps a search key directly to page containing the <search key, pid> information. This page might lead to a page-overflow chain.*
- Doesn’t require intermediate page fetches for internal “steering nodes” of tree-based indices.
- *Hash-based* indexes are best for *equality selections*. They do not support efficient range searches.
- Static and dynamic hashing techniques exist with trade-offs similar to ISAM vs. B+ trees.

The case for "equality" only

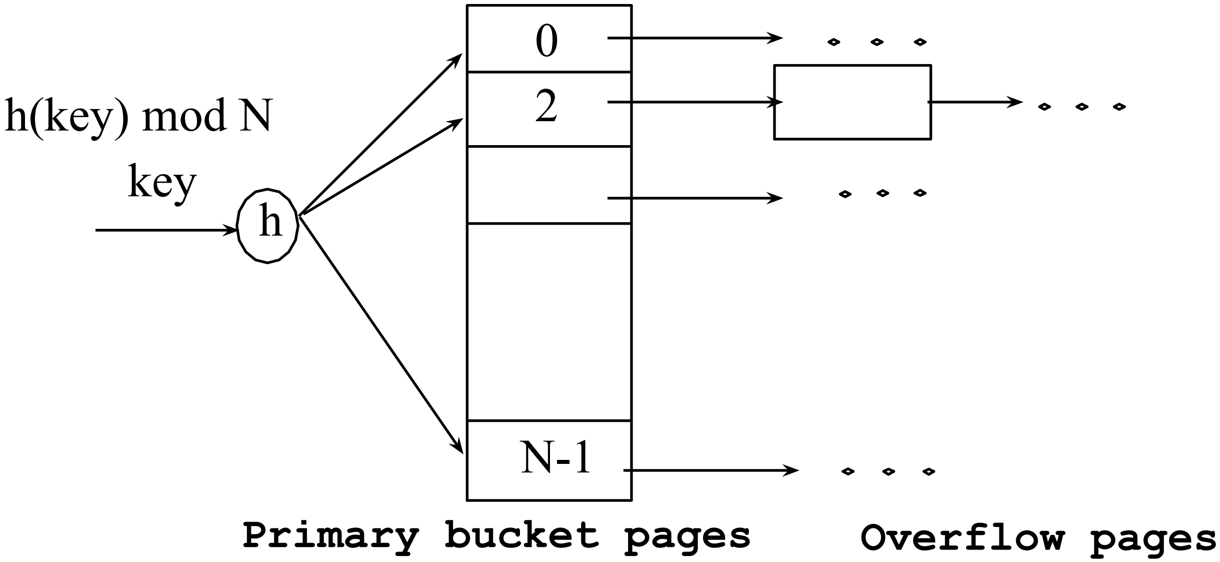
Clearly a tree based index can handle both equality and range searches. So why support an index with a limited function index?

- Equality test of keys are central to joins
- Equality tests of non-keys are common
- Needs to be a significant speed-up over alternatives

Equality Now

Static Hashing

- # primary *index* pages are fixed, they are allocated sequentially on their storage volume, they are never deallocated; overflow pages are allocated if needed.
- **h**(*search key*) mod M = bucket index in which any <*search key, rid*> will be placed if one exists. (M = # of buckets)
- When many records map to the same bucket the overflow are created and linked

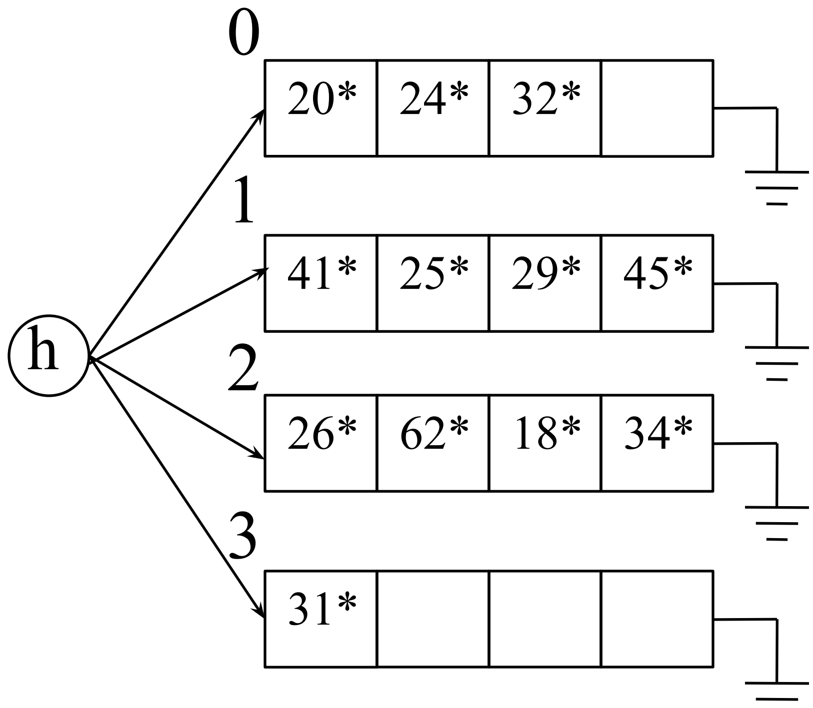


Static Hashing (Contd.)

- Buckets potentially contain many unrelated <*search key, rid*> records, and they must be scanned to find desired search keys
- Hash function maps a *search key* to a bin number *h(key)* □ 0 … M-1. *Ideally uniformly*.
  - in practice **h**(*key*) = (A * *key* + B) mod M, works well.
  - Where A and B are relatively prime constants
  - Lots of research about how to tune **h**.
- Long overflow chains can develop and degrade performance.
- Hence, dynamic hashing techniques (*Extendible* and *Linear Hashing*) address this problem.

Static Hashing Example

- Initially built over "Ages" attribute of our Sailing club database, with 4 records/page and $h(Age)$ = Age mod 4



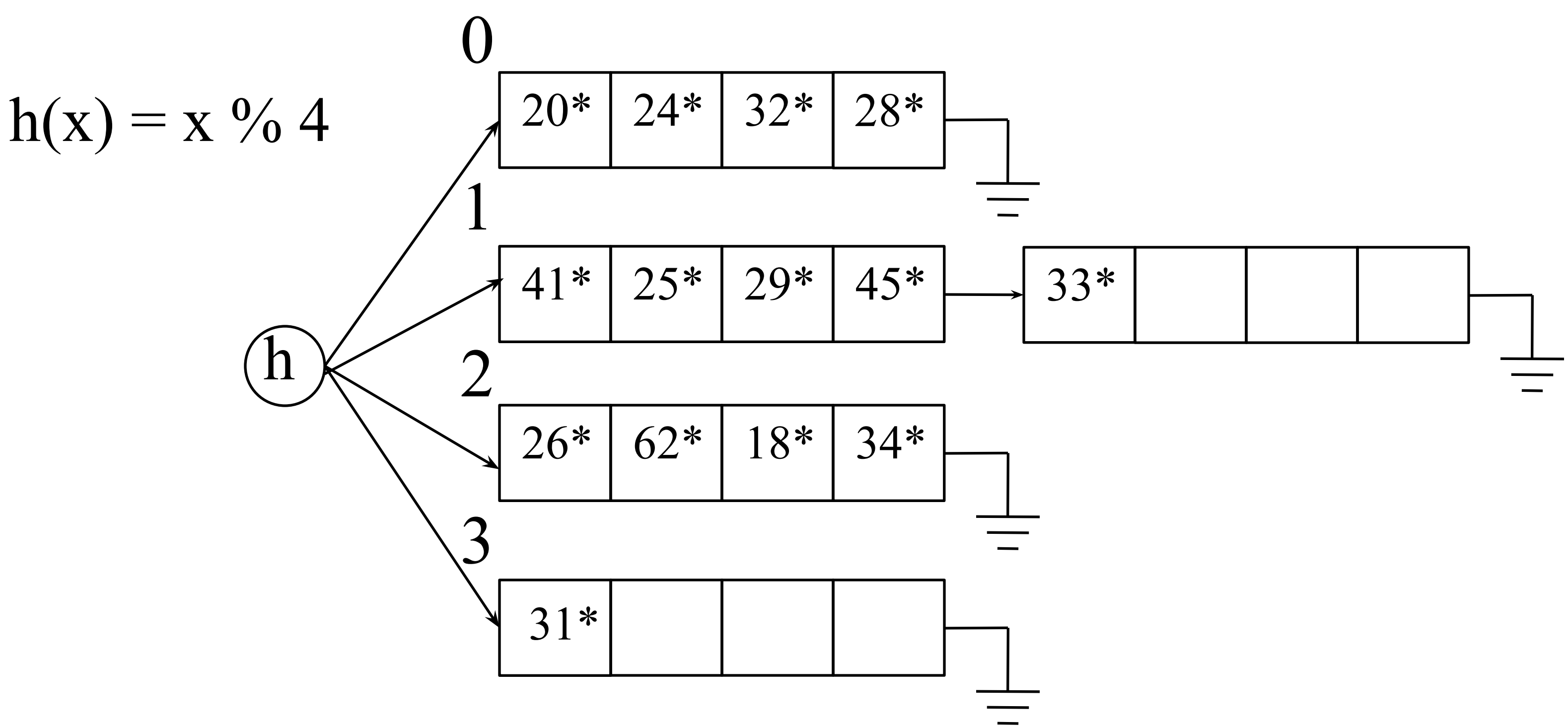
Initial Index

Note: records need not be ordered

Average Occupancy?

Static Hashing Example

- Adding 28, 33
- Deleting 31, (leads to empty page)



Hashing's "Achilles Heel"

- Maintaining Balance
  - Data is often "clustered"
  - Ideal hash functions should uniformly distribute keys over buckets.
  - A good hash function today might be less optimal tomorrow.
- Address overflows and imbalance together
  - If M buckets are not enough, redistribute rather than overflow! Solution: a new hash function
  - Families of hash functions

    $h_0(key), h_1(key), \ldots\ h_n(key)$
  - Desired feature: When transitioning between hash functions we only need to redistribute overflowed buckets

Extendible Hashing

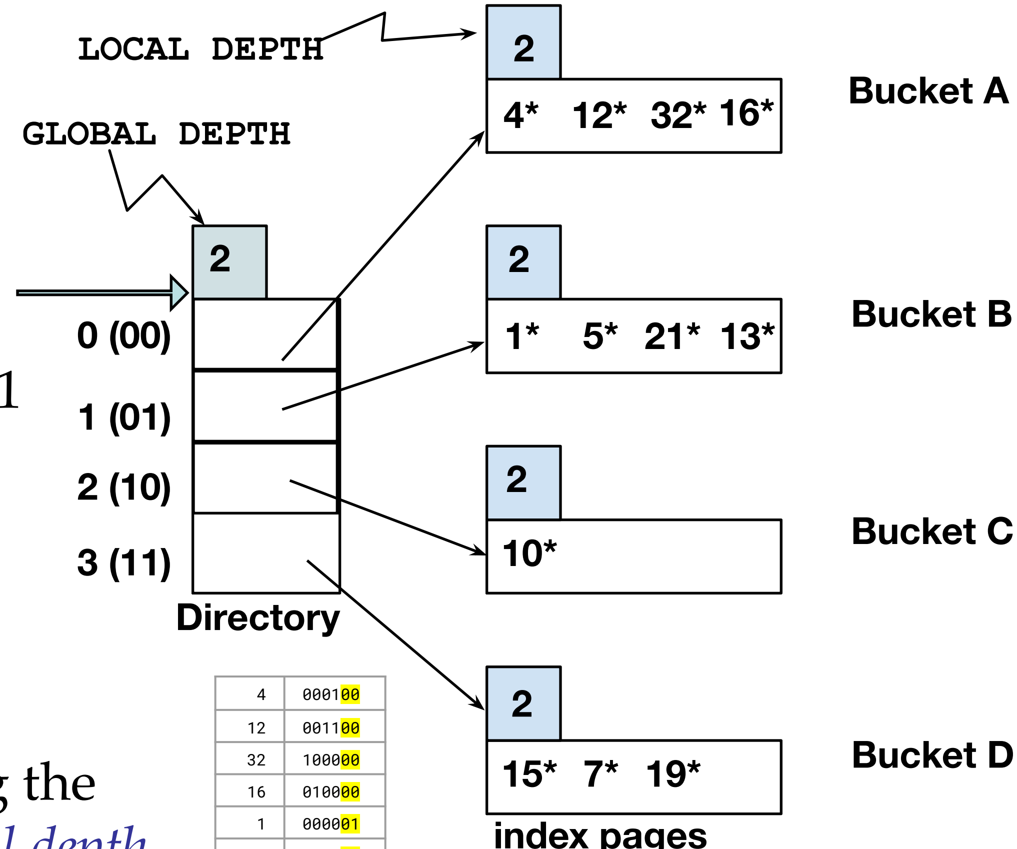
- Situation: Bucket (primary page) becomes full.
  - Change hashing function and reorganize
  - New hash distributes over *twice* # of buckets
  - Hash function's modulo changes to 2M
  - Reading and writing all pages is expensive!
- *Key Idea*: Use *directory of pointers to buckets* double # of buckets by *doubling the directory,* but split only the bucket that overflowed!
  - This directory is much smaller than file, so doubling it is cheap. Only spilt pages are split. *No overflows*!
  - Trick lies in how hash function is adjusted!

Example

- Directory starts with 4 entries
- To find bucket for *r*, take last *global depth* # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(5) = 5 % 4 = 1
    In binary 101, last two bits 01

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

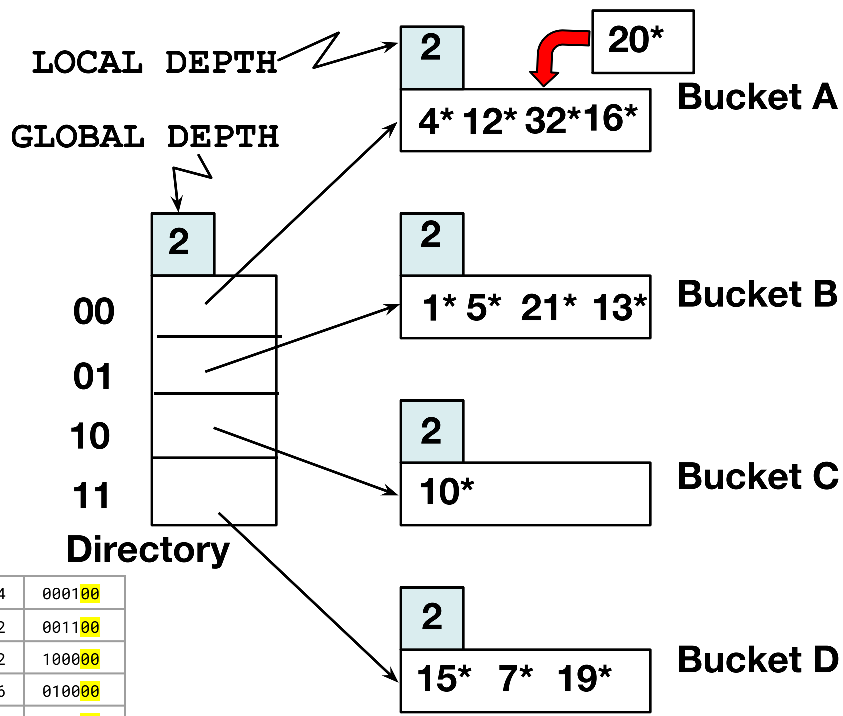
- *If necessary*, double the directory. (Decision is based on comparing the directory's *global depth* with *local depth* of the bucket.)

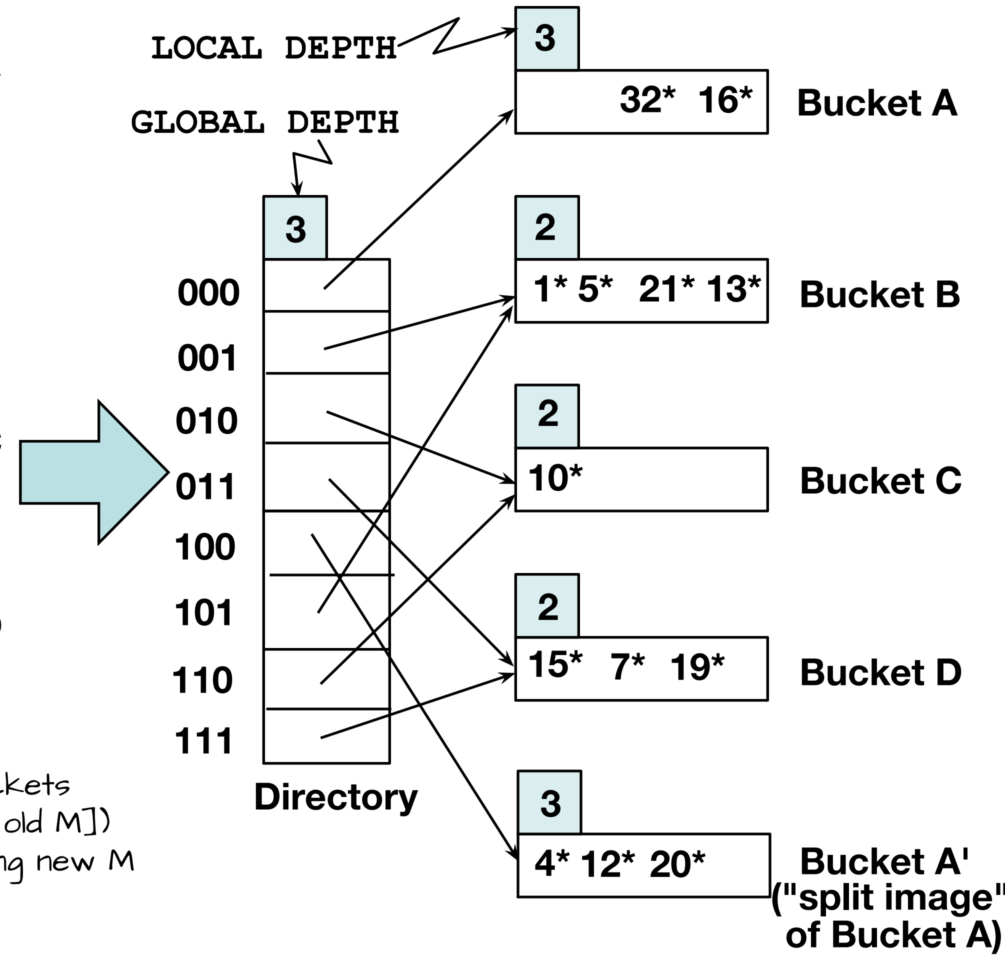


4	000100
12	001100
32	100000
16	010000
1	000001
5	000101
21	010101
13	001101
10	001010
15	001111
7	000111
19	010011

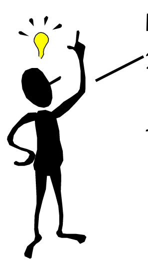
20 = 10100 → 00

Insert h(r)=20 (Causes Doubling)





4	000100
12	001100
32	100000
16	010000
1	000001
5	000101
21	010101
13	001101
10	001010
15	001111
7	000111
19	010011
20	010100



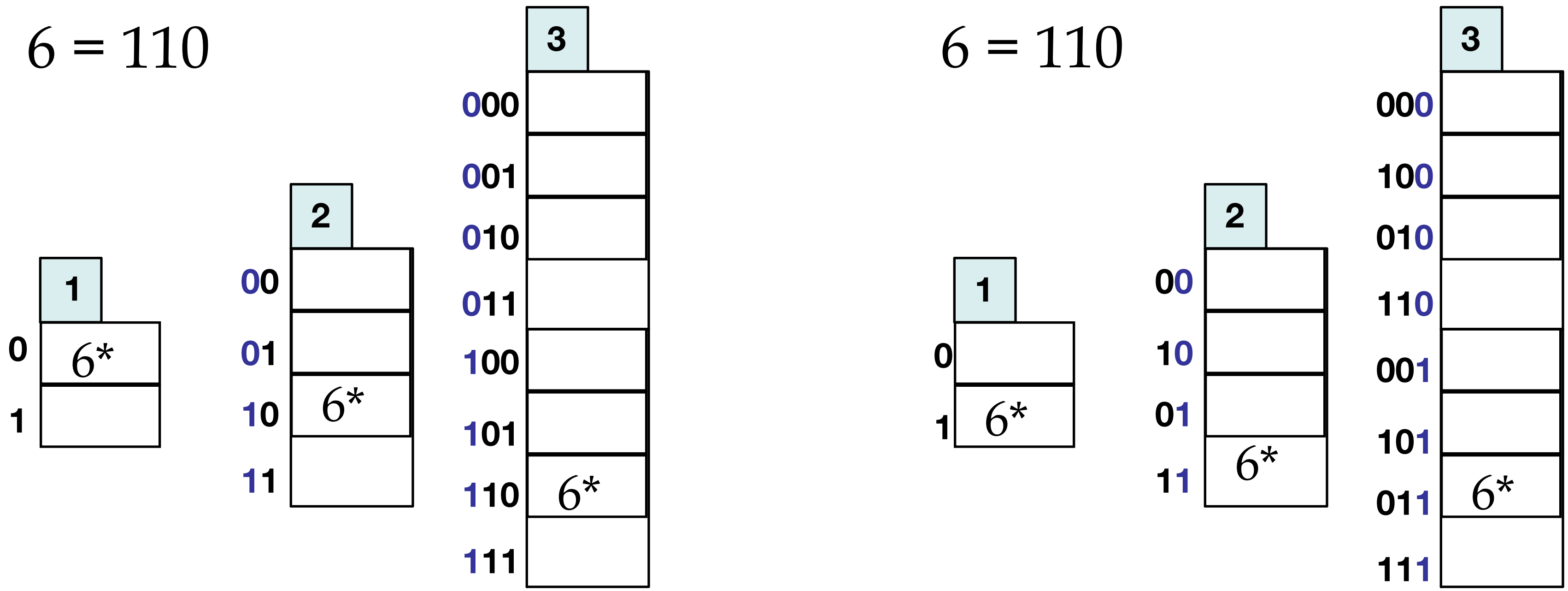
1) Double the directory size
2) Add new links pointing to old buckets
(i mod new M = directory[i mod old M])
3) Split the overflowed bucket using new M

Points to Note

- 4 (100), 12 (1100), and 20 (10100).
  Last **3** bits (100) tell us *r* belongs in A or A'.
  - *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 in its 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!)

Directory Doubling

Why use least significant bits in directory?
Allows for doubling via copying!



Least Significant vs. Most Significant

Comments on Extendible Hashing

- If directory fits in memory, or is pinned in page buffer, equality searches are answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, contains 1,000,000 records. A hash with 16,384 directory entries, with 40 bytes per <*search key, rid*> using 4Kb pages has a capacity of 100 search keys per bucket and a capacity of 1,638,400 keys; Chances are high that the directory will fit in memory.
  - Directories grows in spurts, and, if the distribution *of hash values* is skewed, the directory size can grow large.
  - Multiple entries with *same hash value* cause problems!
- **Delete**: If removal of data entry makes a bucket empty, it can be merged with its 'split image'. If directory element M/2 pairs point to the same bucket, can halve the directory.

Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH avoids the need for a directory, yet *avoids* the problem of "*long*" overflow chains.
- *Idea*: Uses 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 (Contd.)

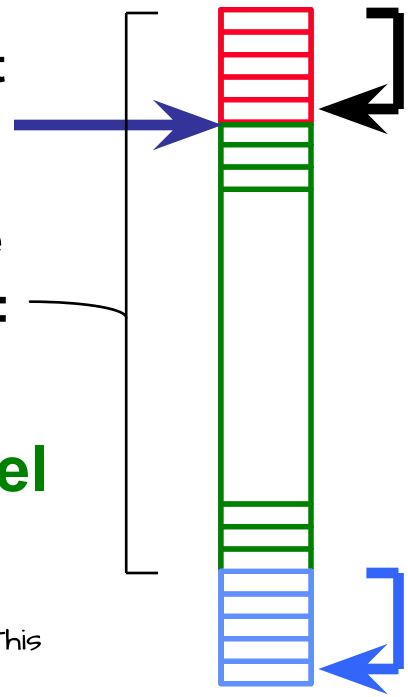
- Directory avoided in LH by *allowing overflow pages*, and always splitting the *next* bucket (*in a round-robin fashion*).
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry *r*, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range *Next* to N_R , *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.

Overview of LH File

- In the middle of a round.

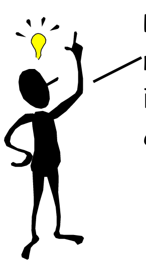
Bucket to be split
Next

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



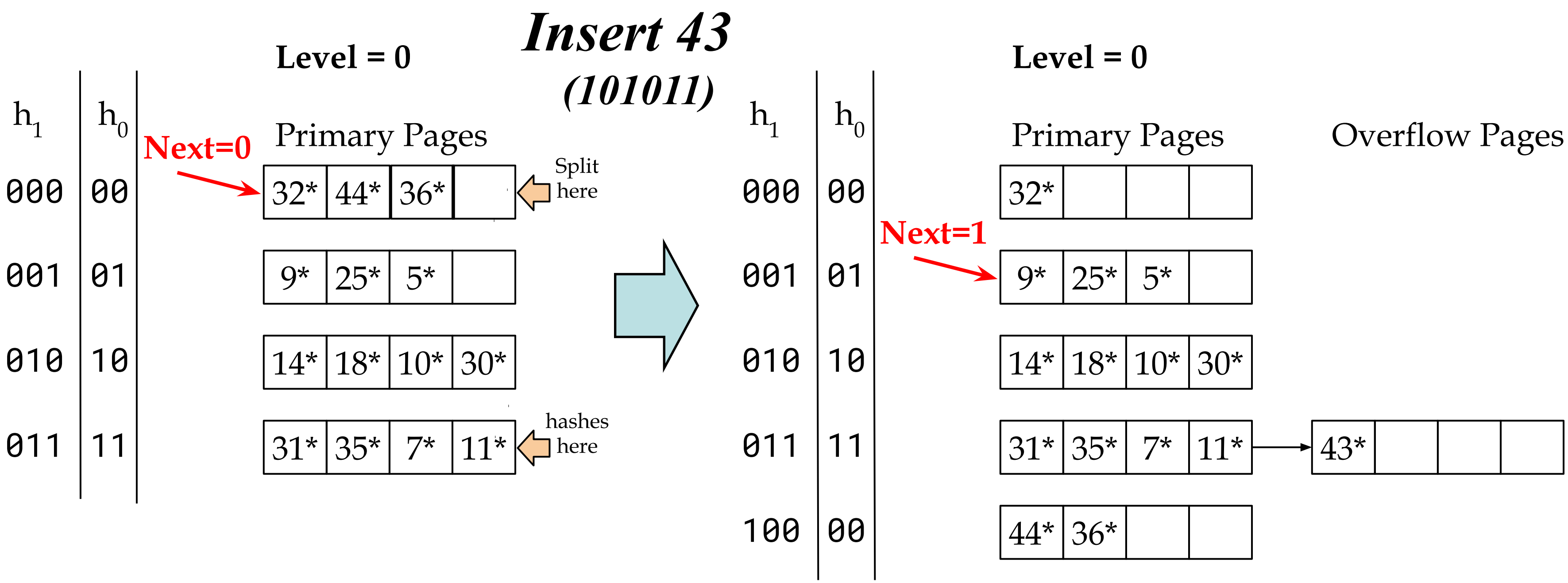
In linear hashing bucket pages must be allocated sequentially. This is not a requirement for extensible hashing.

Linear Hashing (Contd.)

- **Insert**: Find bucket by applying two hashes $\mathbf{h}_{Level}$, $\mathbf{h}_{Level+1}$
  - If $\mathbf{h}_{Level}$ < *next* use it otherwise $\mathbf{h}_{Level+1}$
  - If bucket to insert into is full:
    - Add an overflow page and insert data entry.
    - Split and redistribute *Next* bucket and its associated overflow pages and increment *Next*.
    - The bucket that is split may not be the same as the one that overflowed!
    - Once *next* reaches M of h_{Level}, reset it to 0, increase level
- *Next* is updated sequentially. 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

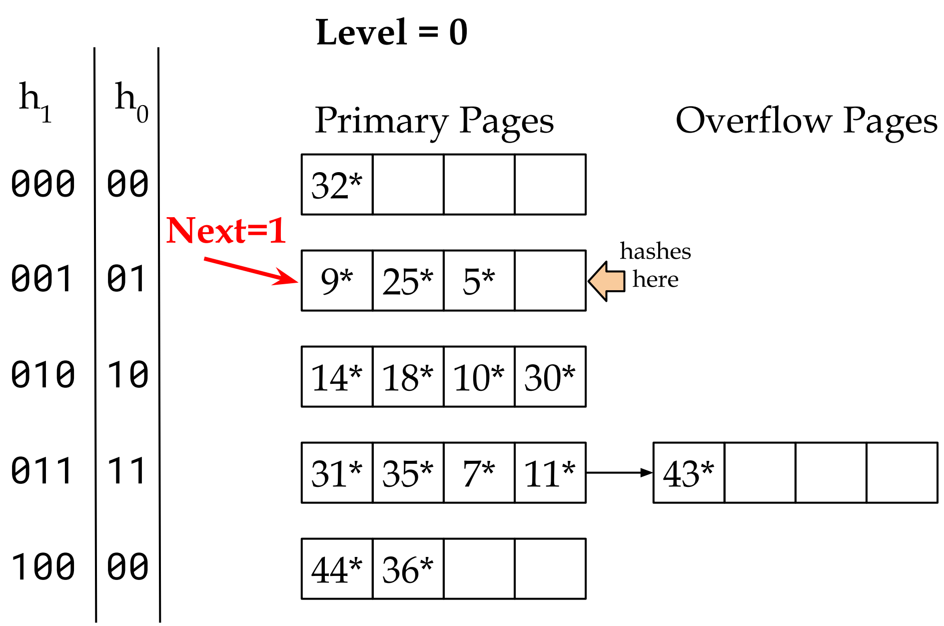
Example of Linear Hashing

- On split, $h_{Level+1}$ is used to redistribute entries.
- If bucket is full, Spill, Split 'Next', Move 'Next'



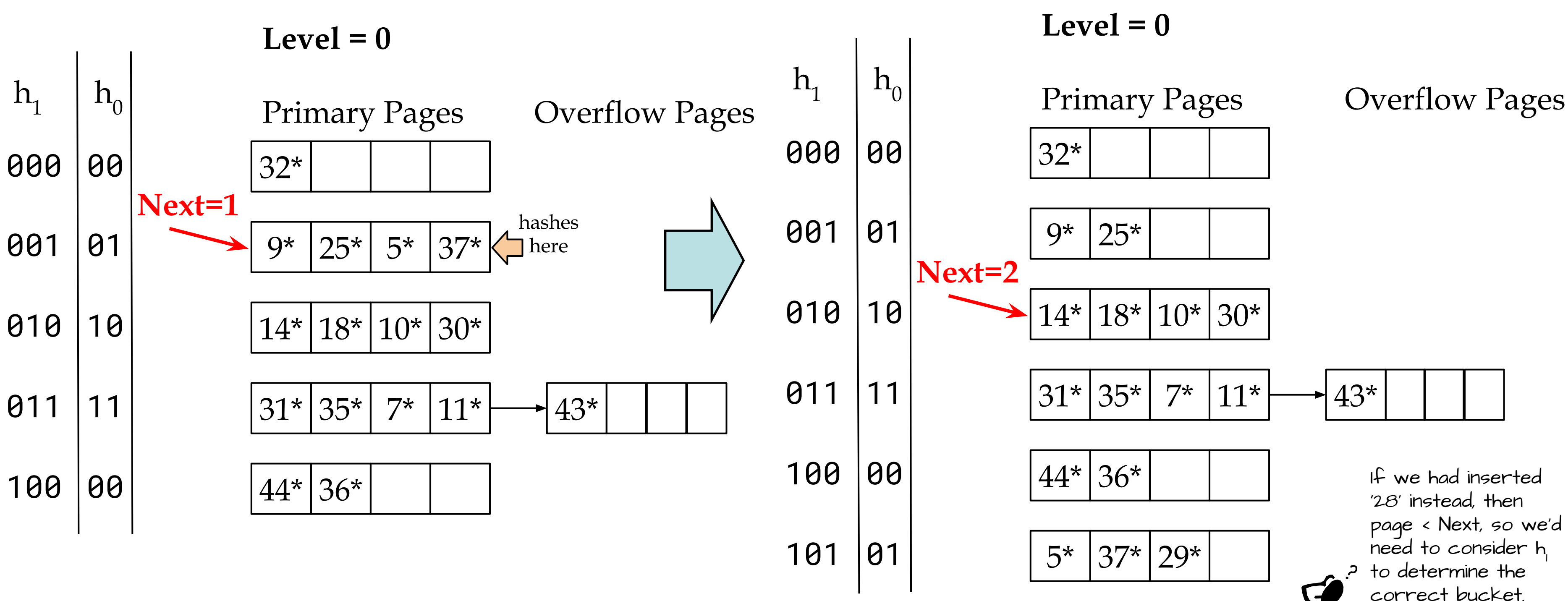
Insert 37 (00100101)

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



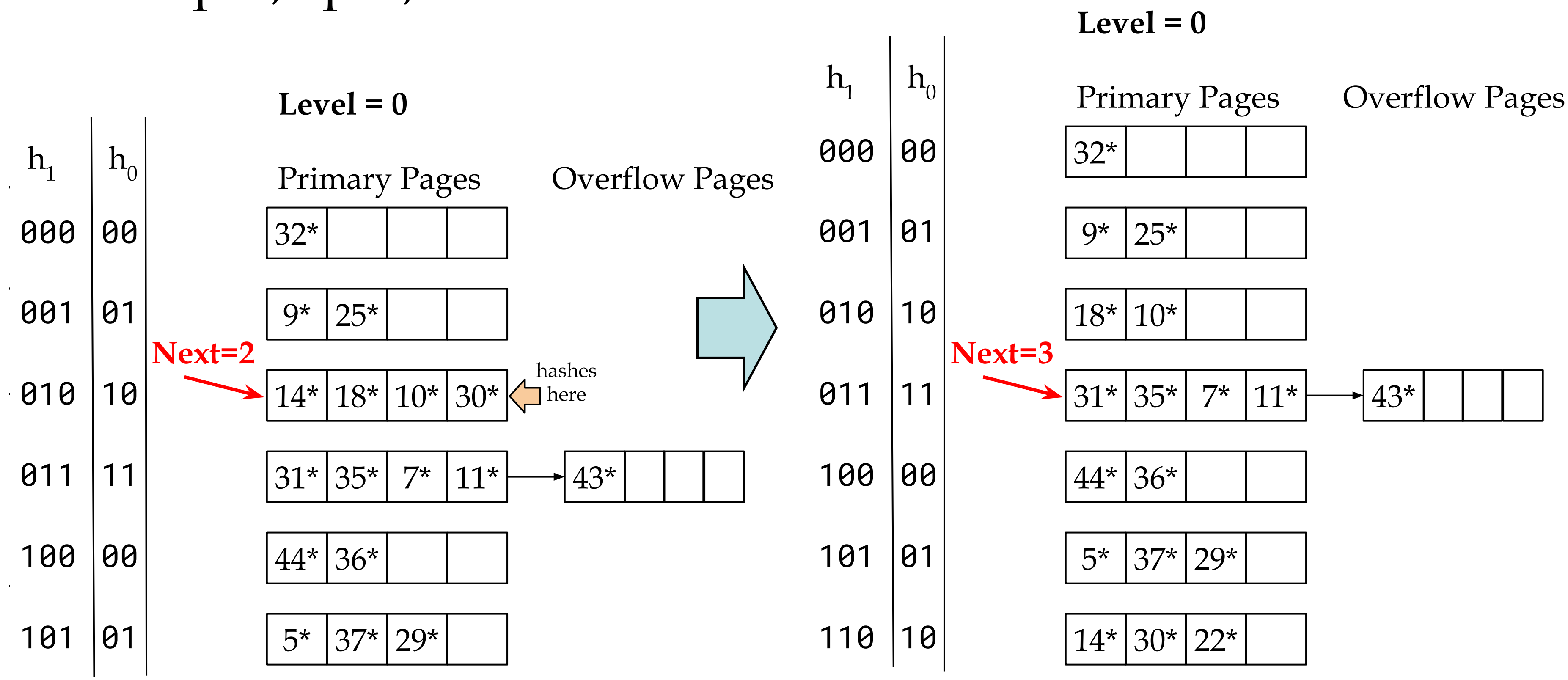
Insert 29 (00011101)

- References page ≥ "Next", check h_0 page
- Spill, split, move Next

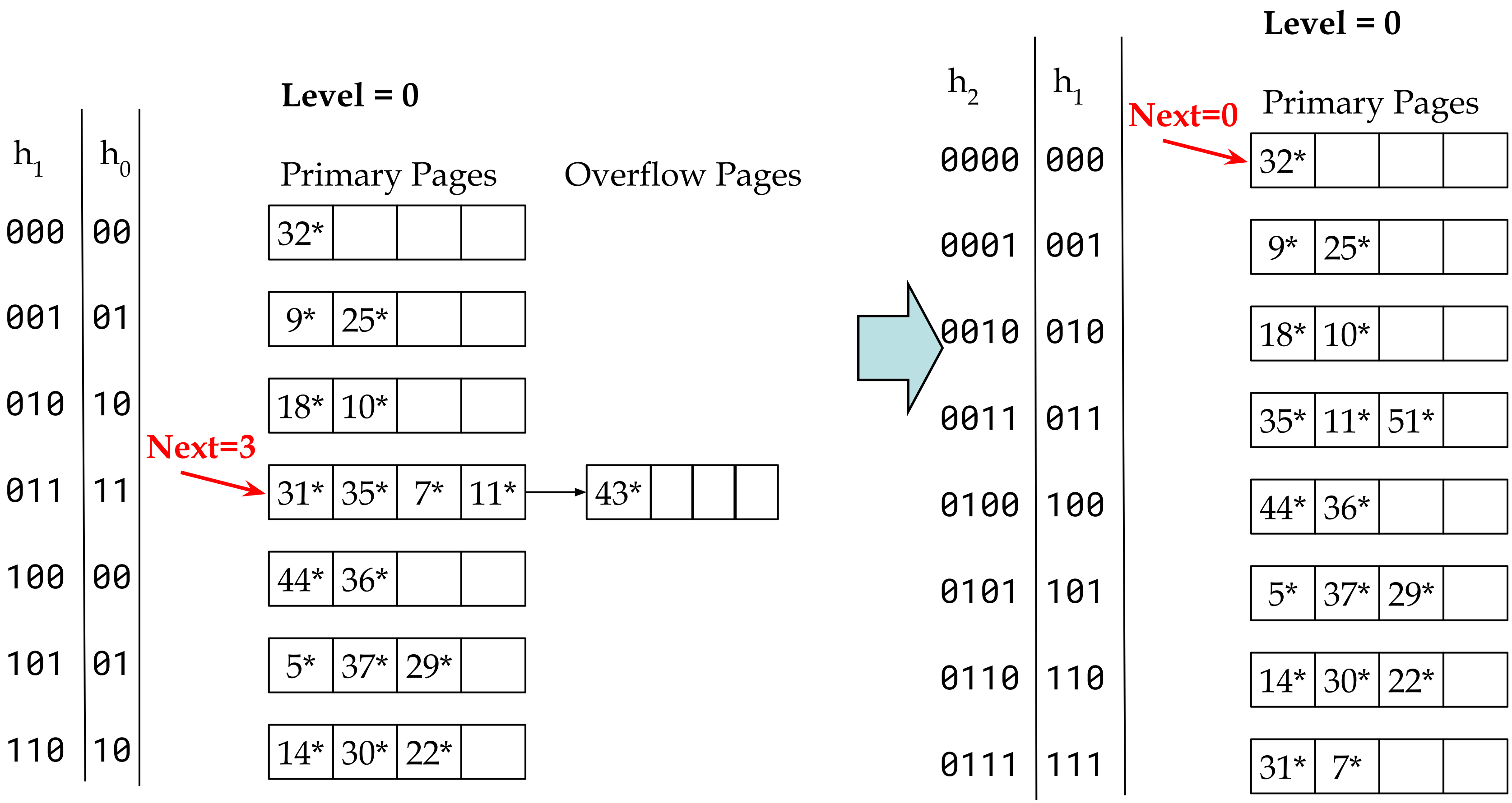


Insert 22 (00010110)

- References page ≥ "Next", check h_0 page
- spill, split, move Next



Add 51 (00110011): End of a Round



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 a directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long, nor around for 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!