Hash-Based Indexes

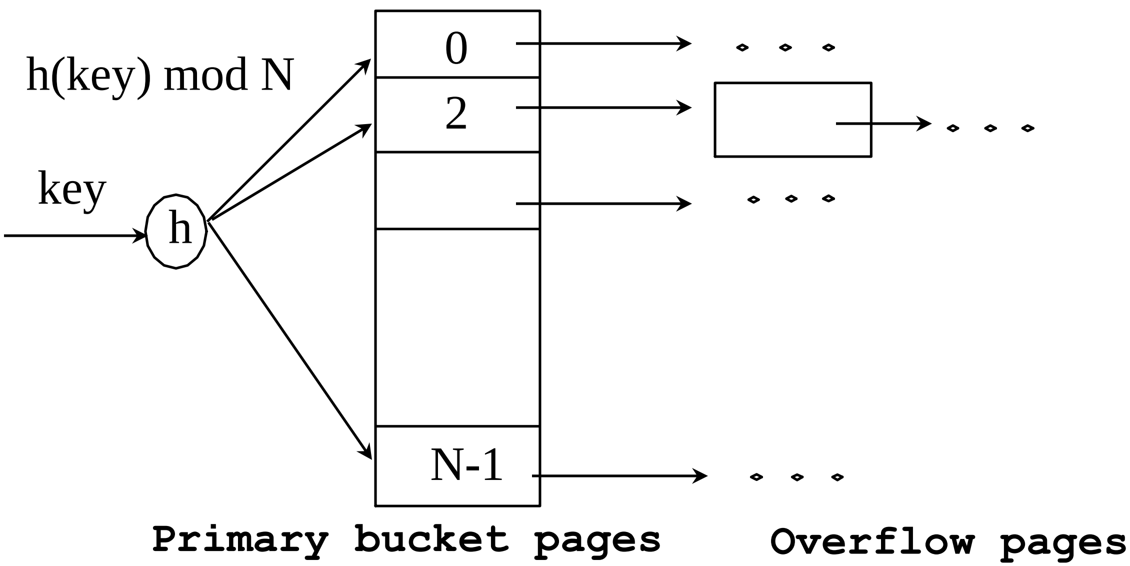
Chapter 11

Introduction

- *Hashing maps a search key directly to the pid of the containing page/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.

Static Hashing

- # primary *index* 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)

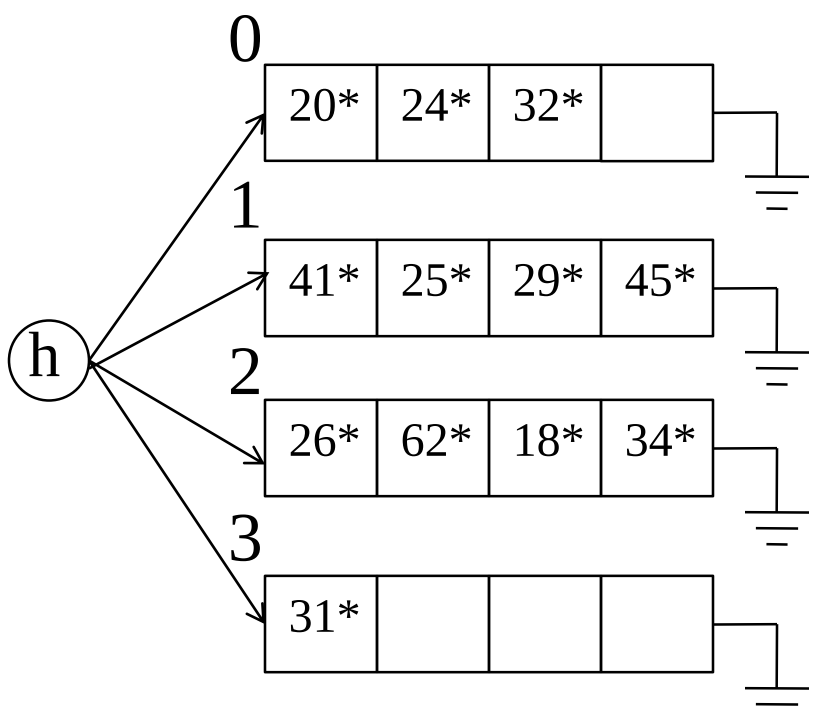


Static Hashing (Contd.)

- Buckets contain *data entries* (*<search key>*, *<rid>*).
- 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



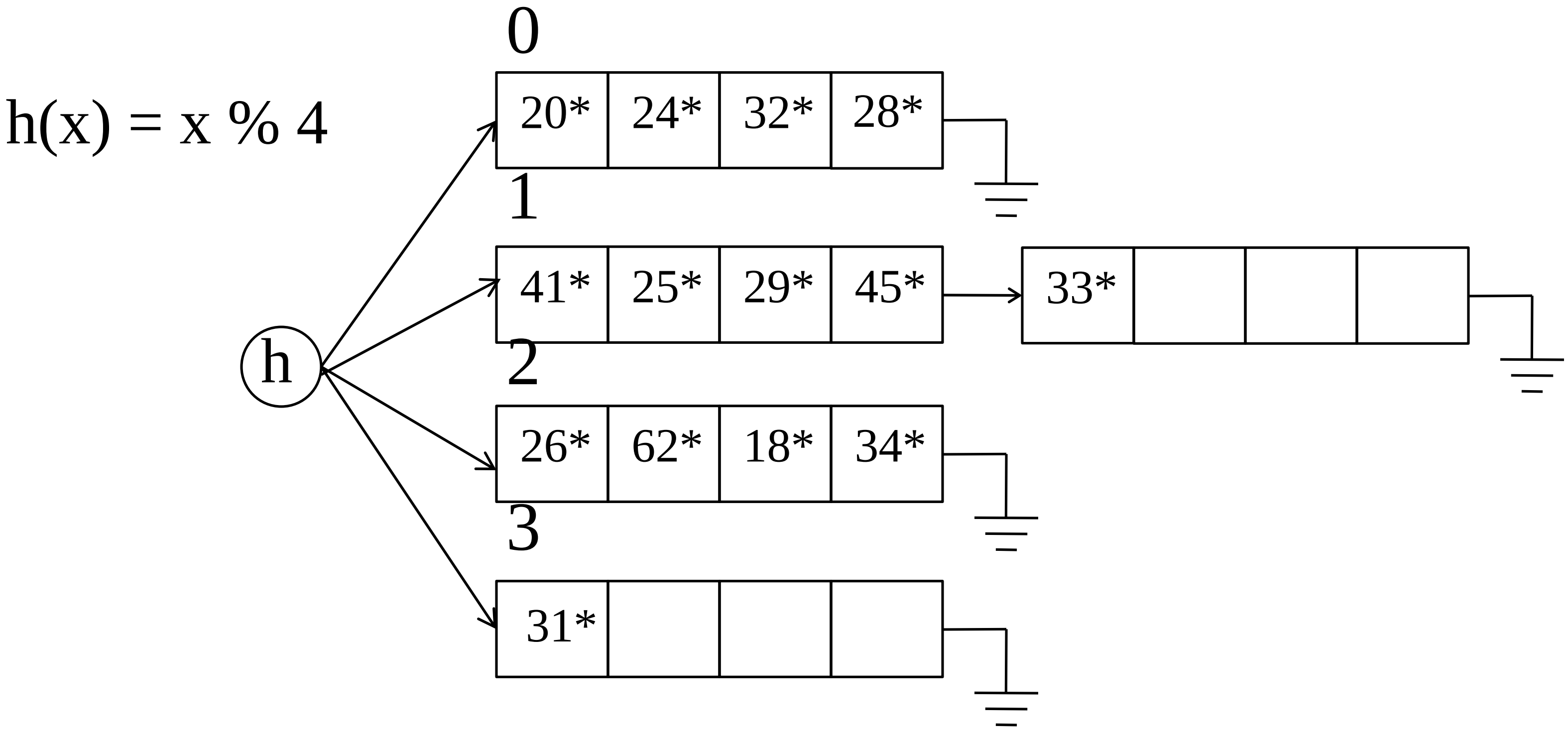
Initial Index

Note: records need not be ordered

Average Occupancy?

Static Hashing Example

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



Hashing's "Achilles Heel"

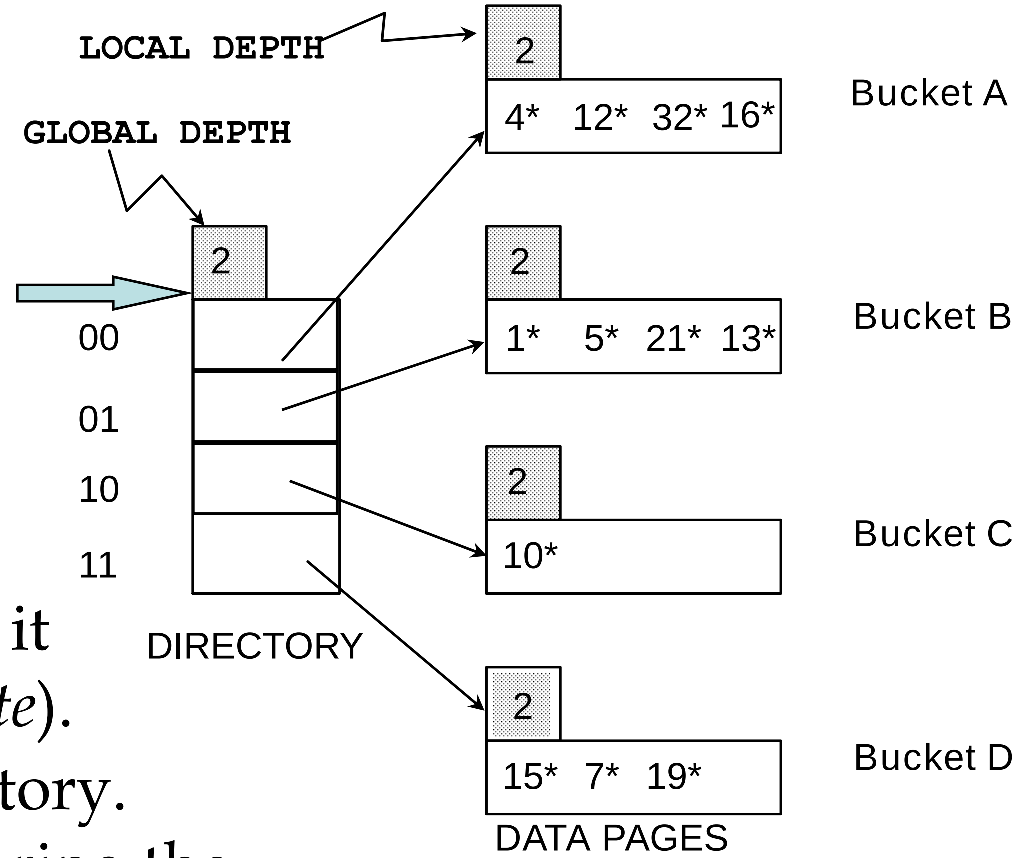
- Maintaining Balance
  - Data is often "clustered"
  - Ideal hash functions should uniformly distribute keys over buckets. Demands a good hash function (lots of research in this area)
- Bucket Spills
  - What if M buckets are not enough?
    Solution: new hash function
  - Families of hash functions
    $h_0(key), h_1(key), \ldots\ h_n(key)$
  - Transitions only redistribute overflowed buckets

Extendible Hashing

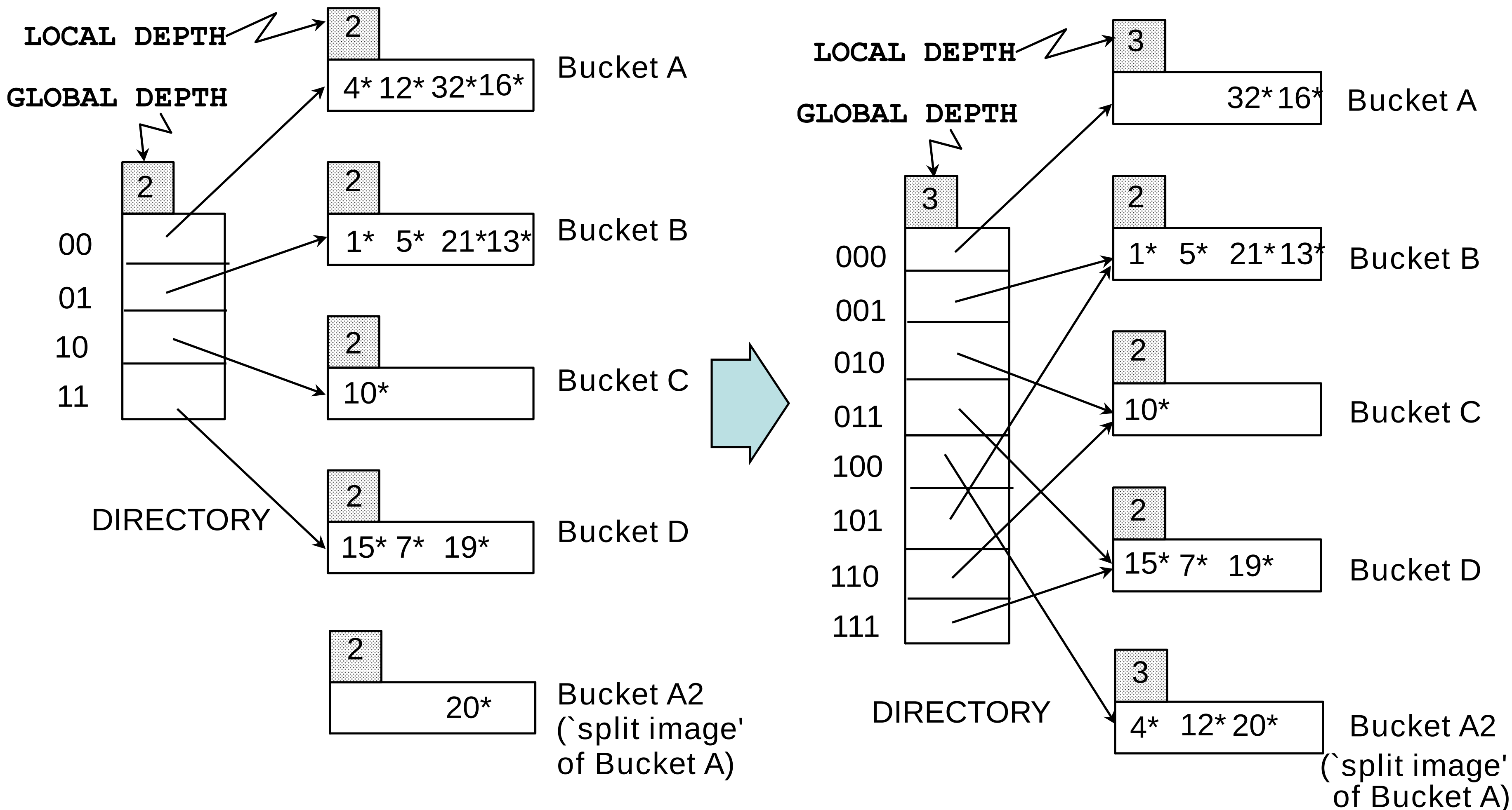
- Situation: Bucket (primary page) becomes full. Change hashing function and reorganize. Reorganizes index by *doubling* # of buckets
  - Reading and writing all pages is expensive!
- *Key Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only spilt pages are split. *No overflows*!
  - 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*); 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. (Decision is based on comparing the directory's *global depth* with *local depth* of the 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, it can be merged with its 'split image'. If each directory element points to same bucket as its split image, can halve 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*: 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 (Contd.)

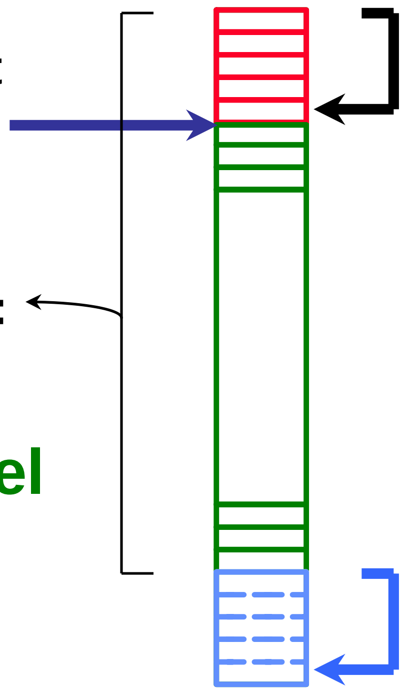
- Directory avoided in LH by *allowing overflow pages*, and always splitting the *next* bucket (*in a round-robin fashion*).
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current 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.

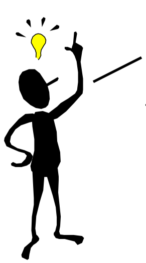
Bucket to be split
Next

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.

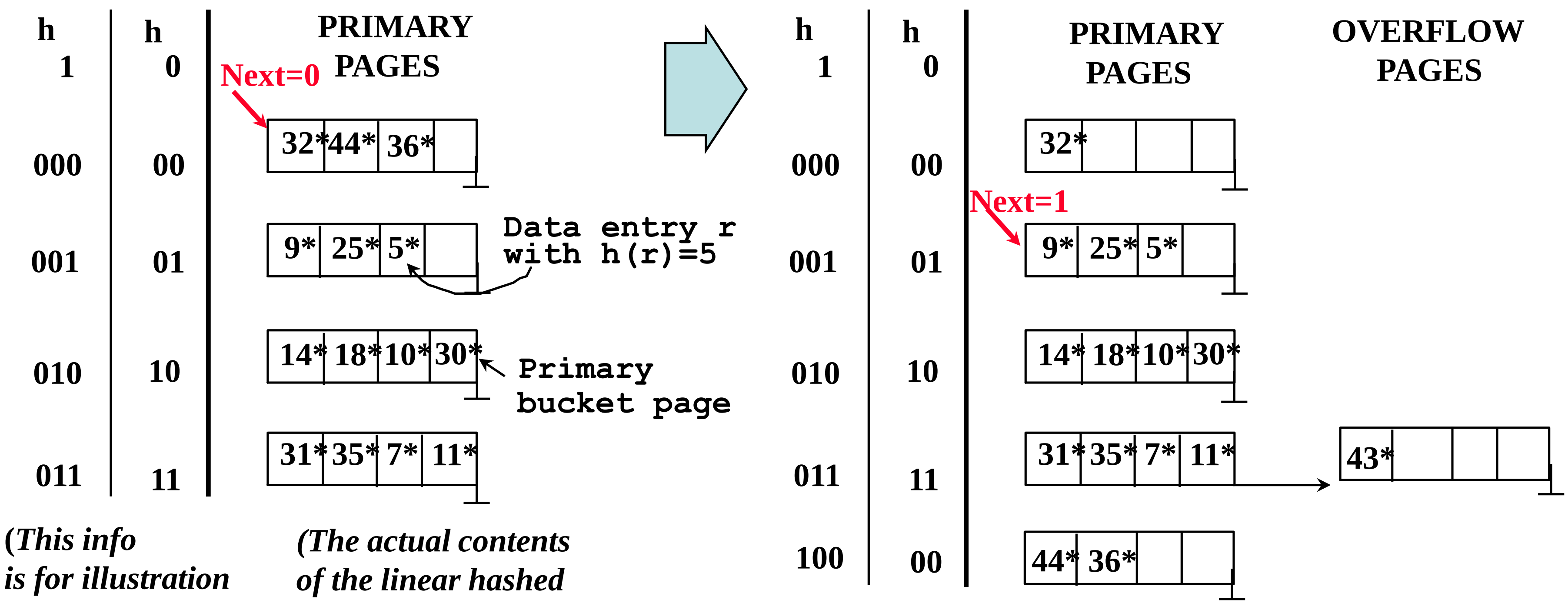
Linear Hashing (Contd.)

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - Split *Next* bucket and any associated overflow pages and increment *Next*.
    - The bucket that is split may not be the same as the one that overflowed!
- Can choose alternate criterions to 'trigger' split
- *Next* must be 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, $\mathbf{h}_{Level+1}$ is used to redistribute entries.
- If bucket is full, Spill, Split 'Next', Move 'Next'

Level=0, N=4 ***Insert 43*** Level=0

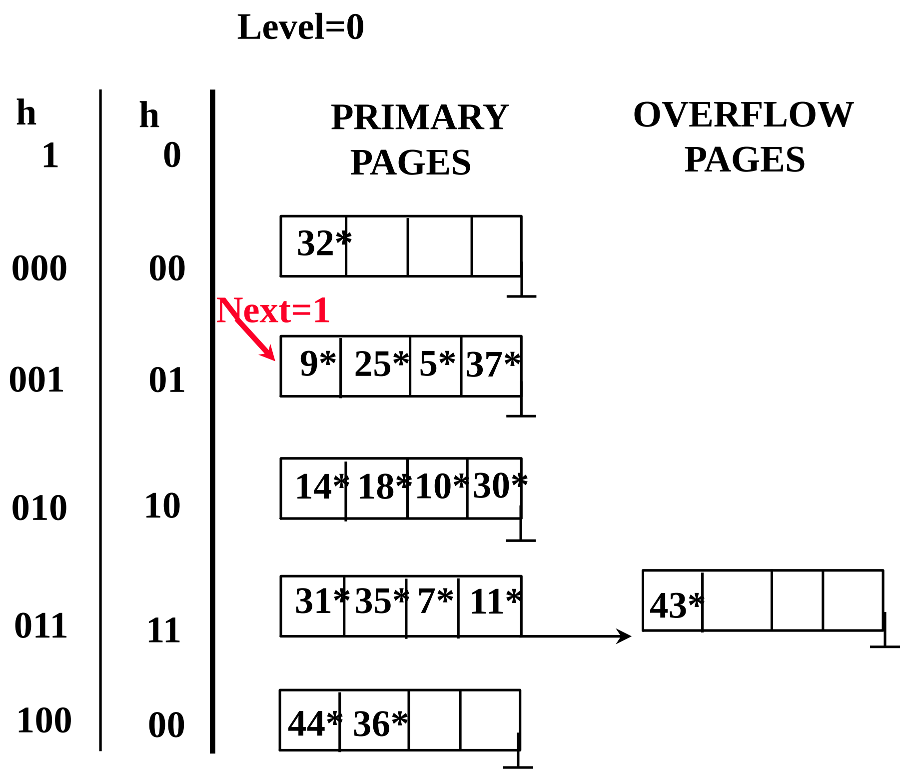


(This info is for illustration only!)

(The actual contents of the linear hashed file)

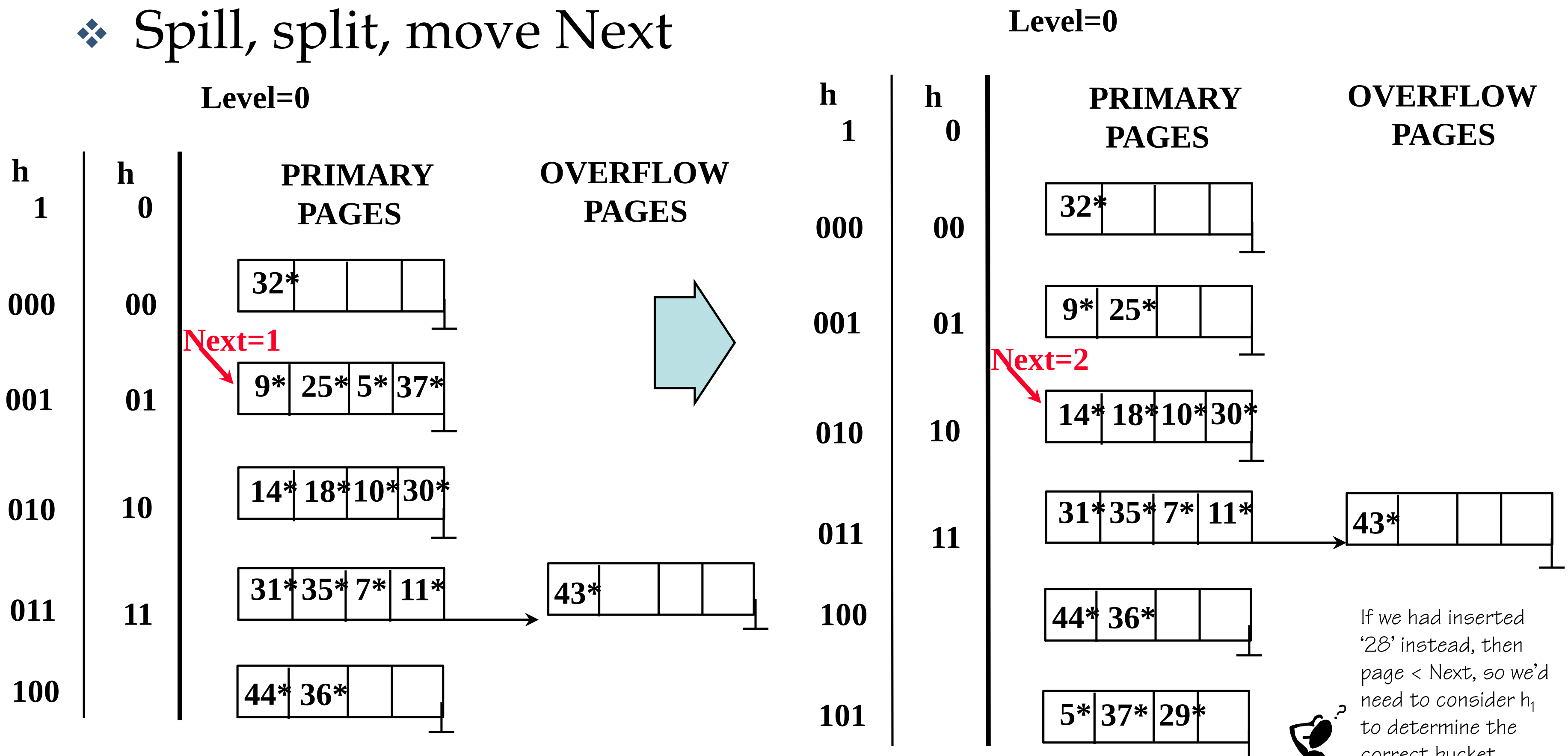
Insert 37 (00100101)

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



Insert 29 (00011101)

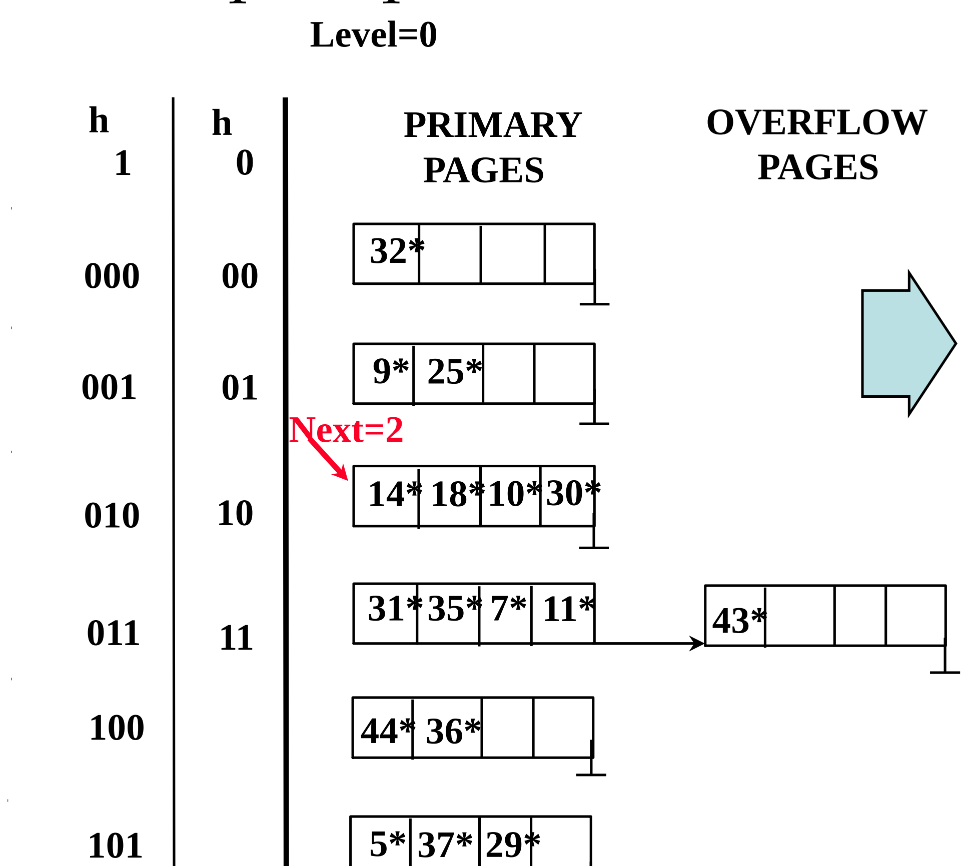
- References page ≥ "Next", check h_0 page
- Spill, split, move Next

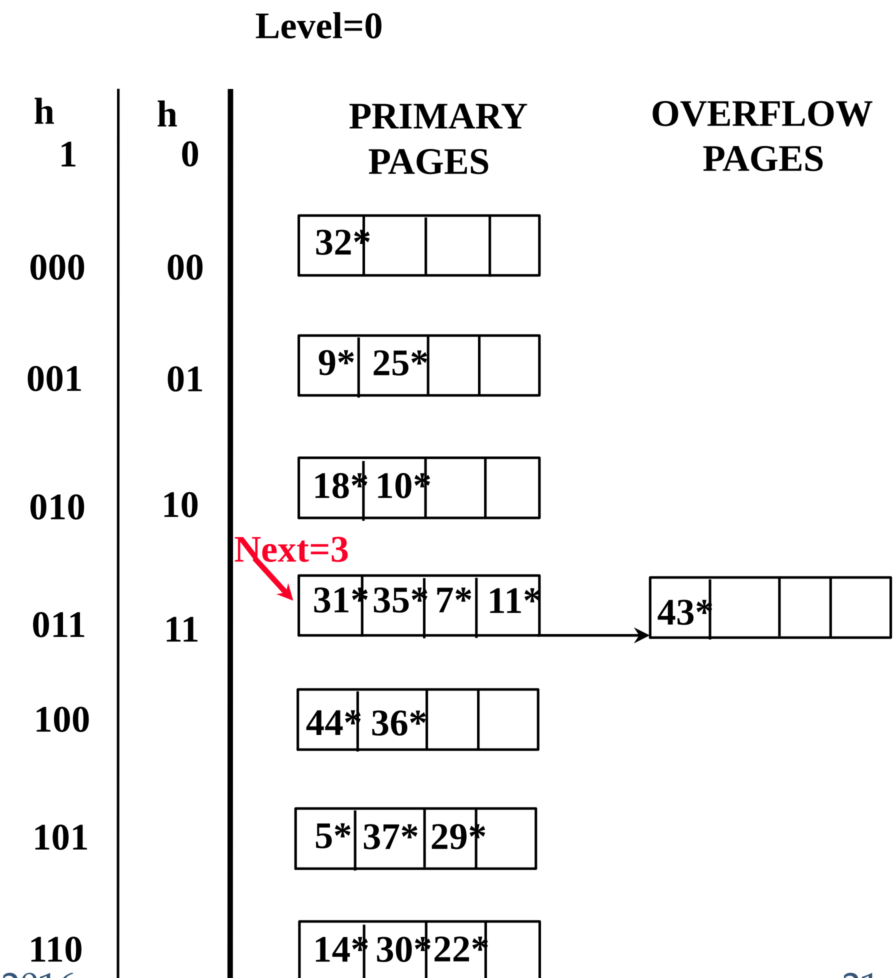


If we had inserted '28' instead, then page < Next, so we'd need to consider h_1 to determine the correct bucket.

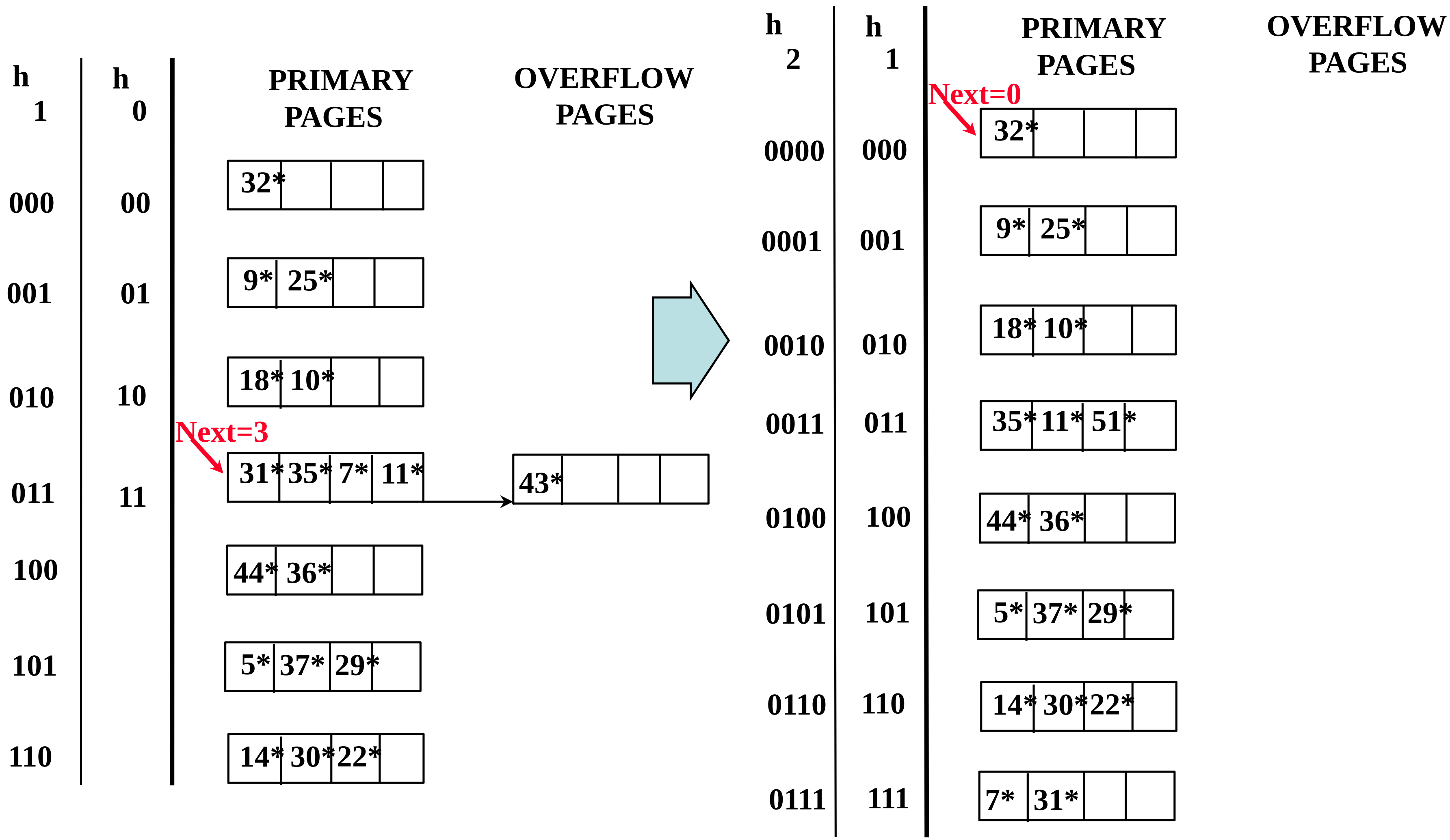
Insert 22 (00010110)

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





Add 51 (00110011): End of a Round



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 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!