CS 133: Databases

Fall 2019
Lec 6 – 09/24
Hash-based Indexes

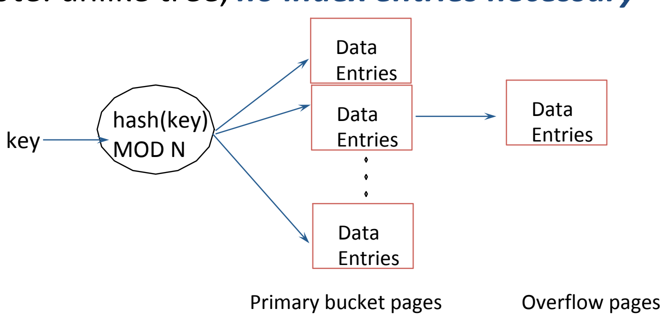
Prof. Beth Trushkowsky

Goals for Today

- Learn how hash-based indexes are constructed
- Understand how operations work on static and dynamic hash indexes, and the impact on cost in I/Os
- Reason about the tradeoffs between approaches to dynamic hash indexes

Anatomy of an Index: Hash-based

- Apply a hash function to search key *k* to determine which **data entries** ***bucket***
  – N number of buckets, find bucket as **hash(k) MOD N**
- Note: unlike tree, ***no index entries necessary***



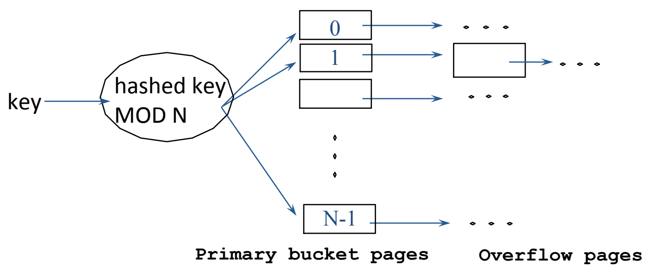
Hashing Functions

- Hash function works on ***search key*** field(s) of record
- Desirable properties for hash function:
  – Uniform distribution: the same number of search key values map to each bucket, for all possible values
  – Random distribution: at any given point in time, each bucket has the same number of search key values
- In practice
  – Typically operate on a binary representation of the data
  – Can tune hash function to achieve desirable properties (e.g., cryptographic techniques)

We'll use integers in our examples, *assume already hashed*
Bucket # = integer MOD N

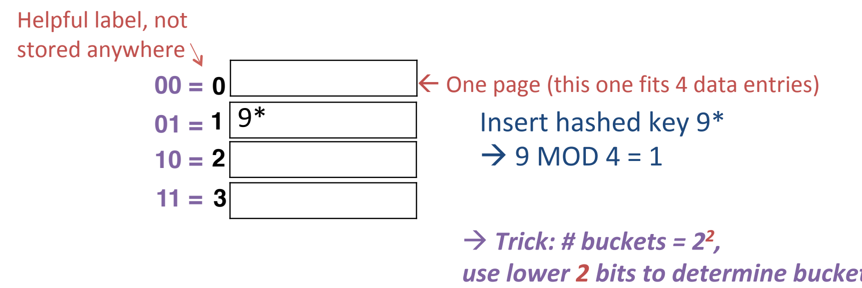
Static Hashing

- Number of primary bucket pages fixed
  - Allocated sequentially
  - Never de-allocated; chain of ***overflow pages*** if needed.



Static Hashing

- Example:
  - # buckets N = 4
  - Bucket number = **hashed key MOD 4**

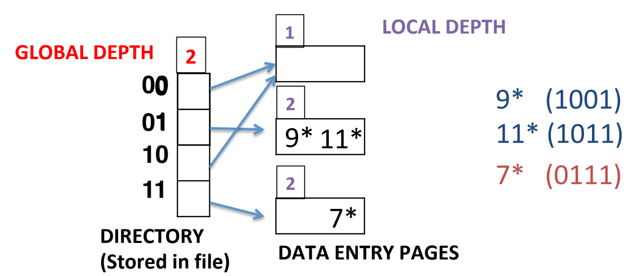


MOAR Buckets

- Situation: Bucket (primary page) becomes full.
  - Want to avoid chains of overflow pages
- Solution: add more buckets (i.e., increase "N")?
  - Okay, but need to rehash everything!
  - *Doubling* # of buckets makes rehashing easier, just use one more bit to differentiate 2N buckets
- Two dynamic approaches:
  - Extendible hashing
  - Linear hashing

Extendible Hashing

- Idea: add level of ***indirection***!
- Use a **directory to point to buckets**
- "Double" # of buckets by ***doubling the directory***
  - Directory much smaller than file, so doubling it is much cheaper (might fit in RAM)
  - When want to "split" a bucket, double the directory
  - Allocate new page *only for the split bucket*



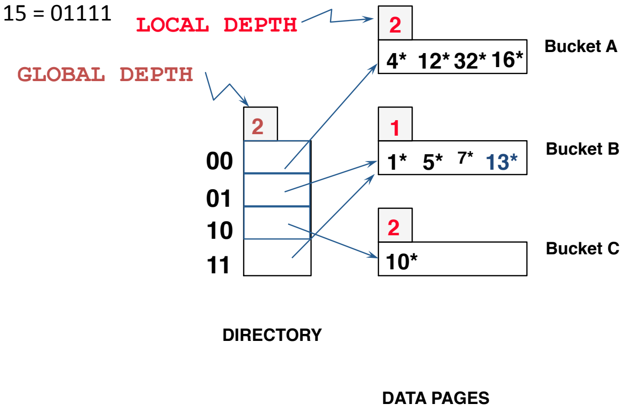
Handling Inserts

- Use **global depth** to look up bucket in directory
- If there's room, put data entry there.

- Else, if bucket is full, *split* it:
  - increment local depth of original page
  - allocate new page with new local depth
  - re-distribute records from original page
  - double directory *if necessary* (when **local > global**)
  - add entry for the new page to the directory

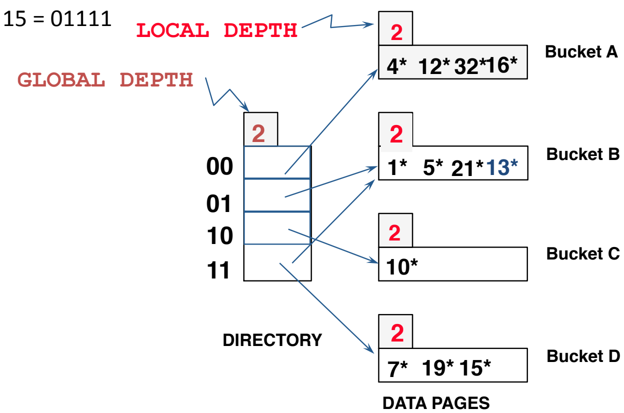
Example: Insert 21*,19*, 15* (before picture)

- 21 = 10101
- 19 = 10011
- 15 = 01111

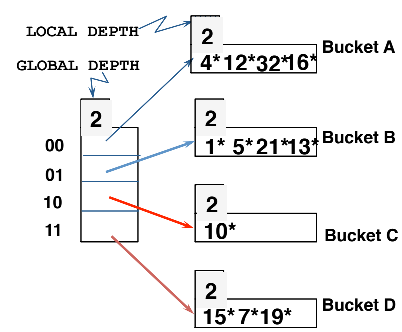


Example: Insert 21*,19*, 15*

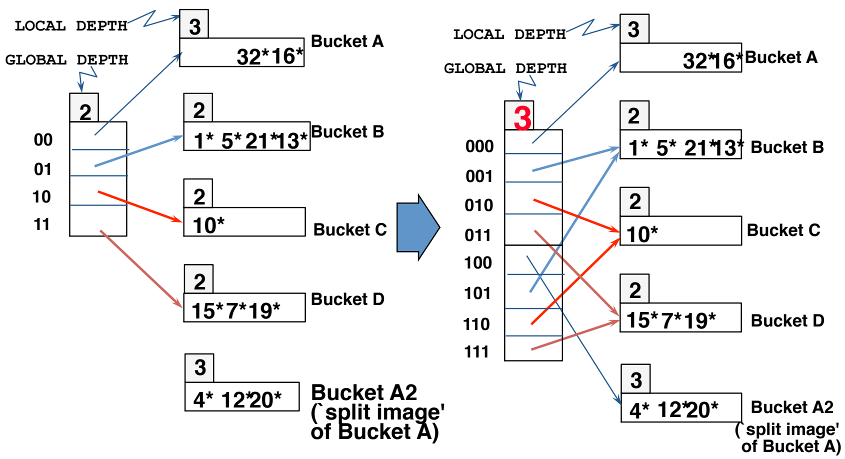
- 21 = 10101
- 19 = 10011
- 15 = 01111



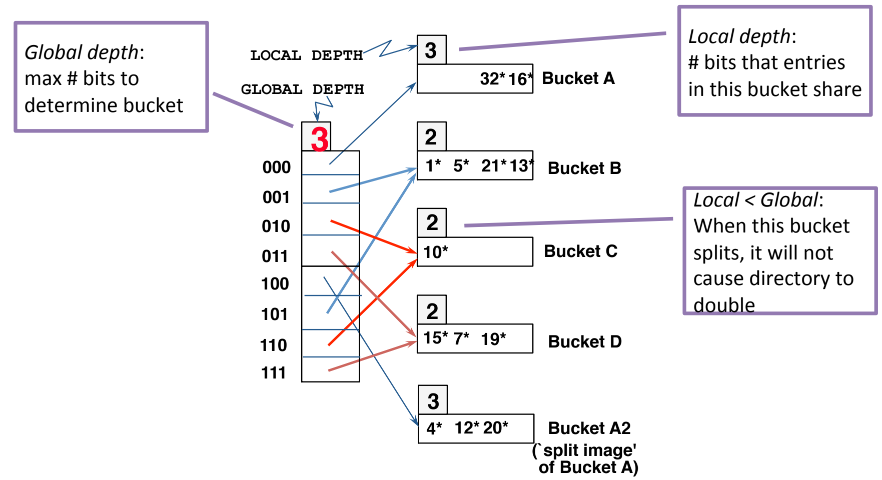
Insert 20* (10100): Causes Doubling (before picture)



Insert 20* (10100): Causes Doubling



Local vs. Global Depth



Extendible Hashing: Comments

- If directory fits in memory, equality search answered with **one disk access**; else two
- Avoids overflow pages (*besides those needed for duplicates/collisions*)

Delete:

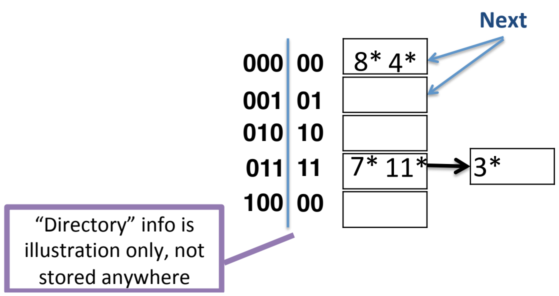
- If removal of data entry makes bucket empty, can be merged with `split image'
- If each directory element points to same bucket as its split image, can halve directory.

Linear Hashing – a Lazier Approach

- Issues with Extendible
  – Completion of an insertion can take a while if it caused a split… **have to move data around**
- Linear Hashing:
  – Idea: decouple ***what is split*** from ***the action that triggers a split***
  – A dynamic hashing scheme that handles the problem of long overflow chains **without using a directory**

Linear Hashing Example

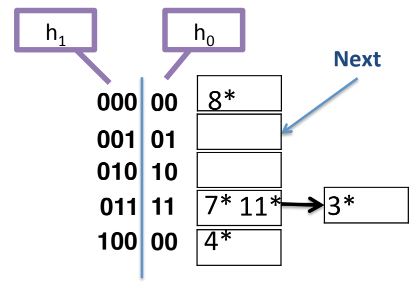
- Avoids directory by:
  - using ***temporary*** overflow pages and choosing the bucket that is split in a *round-robin* fashion.
  - For example, when *any* bucket overflows: split the bucket that is currently pointed to by the "*Next*" pointer and then increment that pointer to the next bucket.



Insert 3* ?

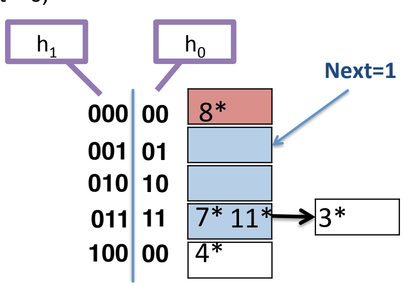
Linear Hashing – The Main Idea

- Use a family of functions h_0, h_1, h_2, ...
- h_i = *hashed key* mod(2^iN)
  - N = initial # buckets (a power of 2)
  - h_{i+1} doubles the range of h_i (similar to directory doubling in extendible hashing)
- *Note*: at a given time, could be "using" two functions: one function for buckets that have been split vs. ones that haven't



Linear Hashing (Contd.)

- Algorithm proceeds in rounds. Current round number is *Level*
  - There are N_{Level} = N * 2^{Level} buckets at the beginning of a round (so N_0 = N)
  - Round ends when all initial buckets in the round have been split (i.e., round ends after splitting bucket *Next* = N_{level}-*1*).
  - The level determines which hash function to use
- To start next round:
  Level++;
  Next = 0;



State of bucket splits:
Buckets *0* to *Next-1* are split
Buckets *Next* to N_{level}-1 are not

Linear Hashing Search Algorithm

To find bucket for data entry *k*, first find $h_{Level}(k)$.

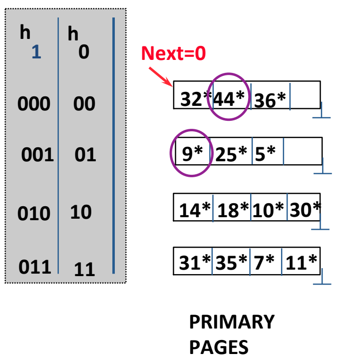
Then:

If $\mathbf{h}_{Level}(k)$ >= Next (i.e., $\mathbf{h}_{Level}(k)$ is a bucket that hasn't been split this round) then *k* belongs in that bucket for sure.

Else, *k* could belong to bucket $\mathbf{h}_{Level}(k)$ ***or*** bucket $\mathbf{h}_{Level}(k) + N_{Level}$, must apply $\mathbf{h}_{Level+1}(k)$ to find out

Example: Search
44 (11100), 9 (01001)

Level=0, Next=0, N=4



$h_{Level}(key) = key \bmod (2^{Level}N)$

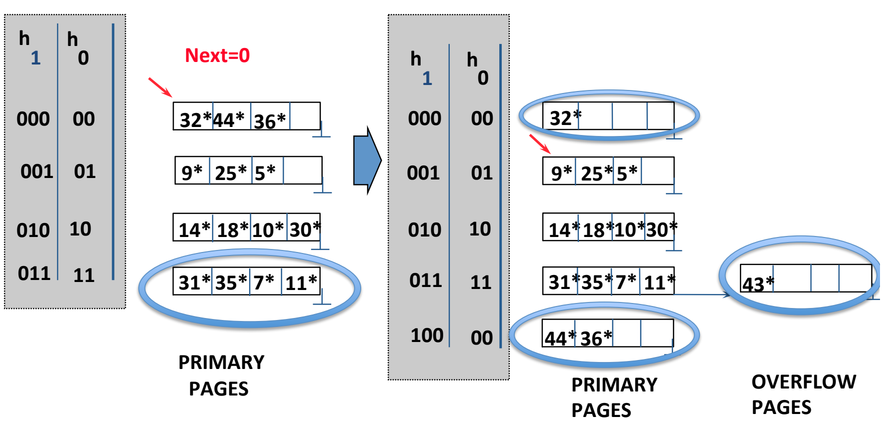
Linear Hashing - Insert

- Find appropriate bucket, if fits, then DONE.
- Else, if no room:
  - Add overflow page and insert data entry.
  - Split *Next* bucket and increment *Next*.
    - This is likely NOT the bucket being inserted to!
    - To split a bucket, create a new bucket and use **$h_{Level+1}$** to re-distribute entries.
- Since buckets are split round-robin, long overflow chains don't develop!

Example: Insert 43 (101011)

Level=0, Next = 0
N=4

Level=0, Next=1

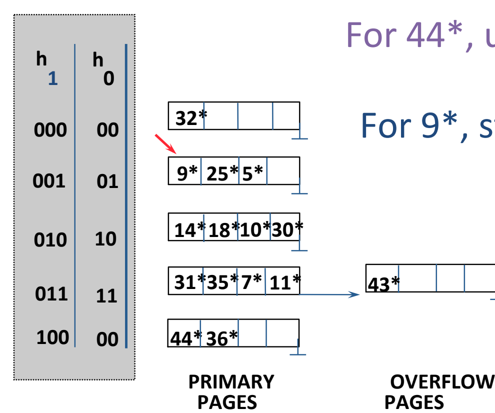


Example: Search
44 (11100), 9 (01001)

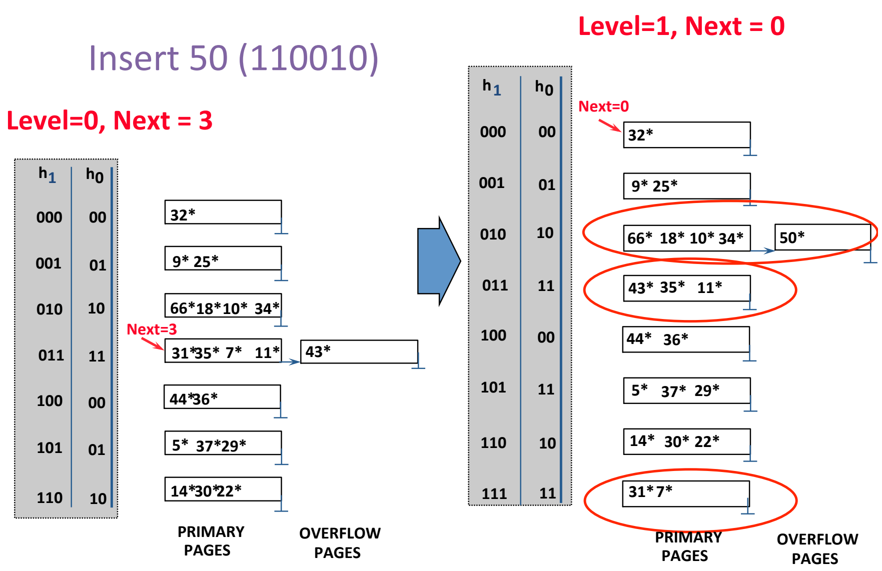
Level=0, Next = 1, N=4

For 44*, use h_1

For 9*, still use h_0



Example: End of a Round



Extendible vs. Linear

- Extendible
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large
- Linear
  - Amount of storage space used 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

Exercise 5: Trees vs. Hashes

Relations:

Professors(**pid,** name, phone)
Clubs(**name**, advisorId, motto)

Query:

```
SELECT C.name, P.name, P.phone
FROM Clubs C, Professors P
WHERE C.advisorId = P.pid;
```

JOIN algorithm:

```
for each page of Clubs
 for each tuple on that page
    probe index on Professors.pid to find matching advisorId
    // extract necessary fields, etc.
```

Which of these two possible indexes on Professors.*pid* would result in fewer I/Os when evaluating the JOIN?

(a). A **B+Tree index** with four levels. Only the root node stays in the buffer pool. **Per Clubs tuple: 3 I/Os to get leaf page, another 1 I/O to fetch record**

(b). An **Extendible hash index**. The directory fits in memory and there are no overflow pages. **Per Clubs tuple: 1 I/O for bucket, 1 I/O for record**