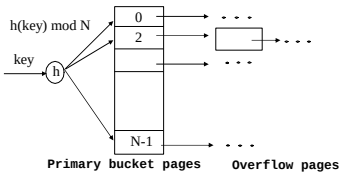
Hash-Based Indexes

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**>
- *Hash-based* indexes are best for *equality selections.*
  - **Provide constant-time searches**
  - ***But* 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 N = bucket to which data entry with key *k* belongs. (N = # of buckets)



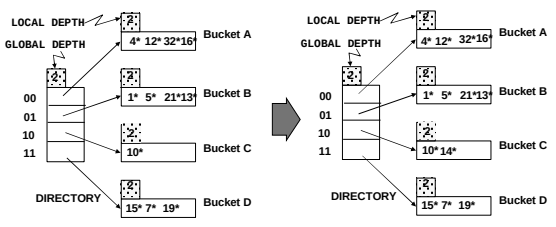
Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... N-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.

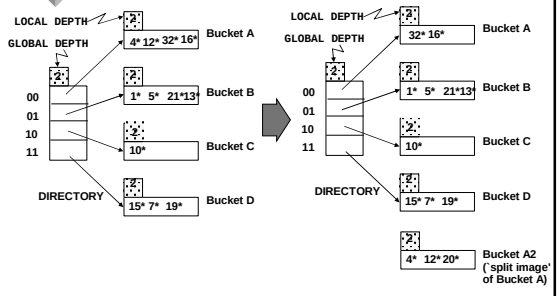
Extendible Hashing

- Main idea: If bucket (primary page) becomes full, why not re-organize file by *doubling* # of buckets?
  - Essentially "splitting" buckets
- But reading and writing all buckets 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.
  - *No overflow pages*!

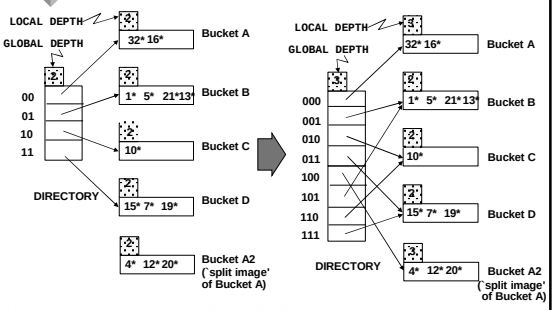
Insert **h**(r)=14



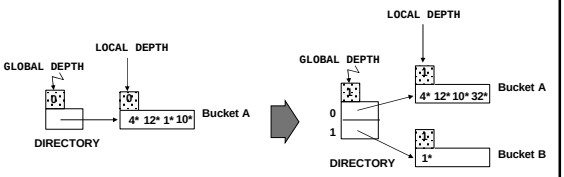
Insert **h**(r)=20



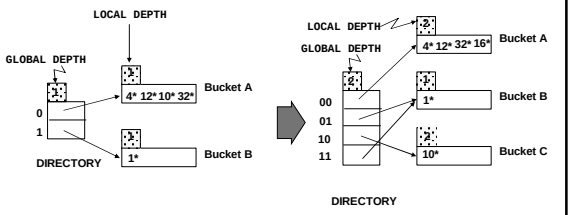
Insert **h**(r)=20



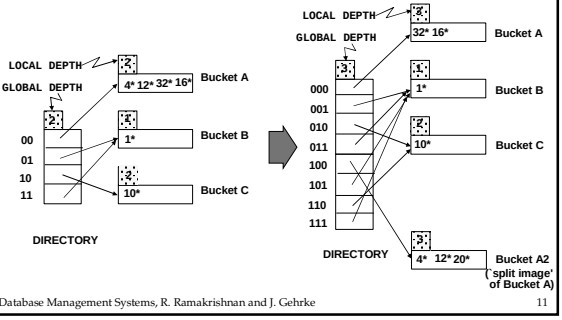
Insert **h**(r)=32



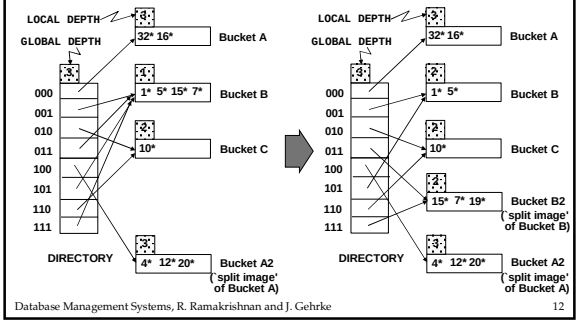
Insert **h**(r)=16



Insert **h**(r)=20



Insert **h**(r)=5, 15, 7, 19



Deletions

- Inverse of insertion
- If removal of data entry makes bucket empty, merge with 'split image'
- If each directory element points to same bucket as its split image, can halve directory

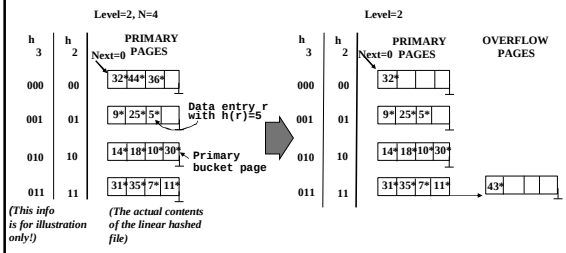
Comments on Extendible Hashing

- 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!
  - When would this happen?

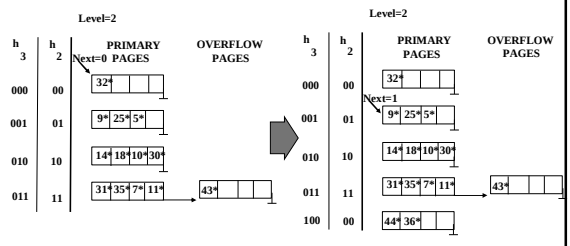
Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing
- LH handles the problem of long overflow chains *without* using a directory, and handles duplicates
- Main idea: split one bucket at a time in *rounds*

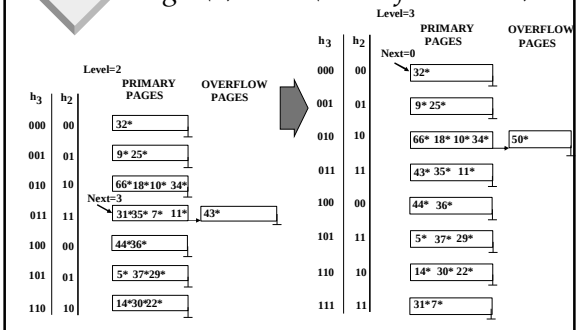
Inserting h(r) = 43



Example (Inserting h(r) = 43)

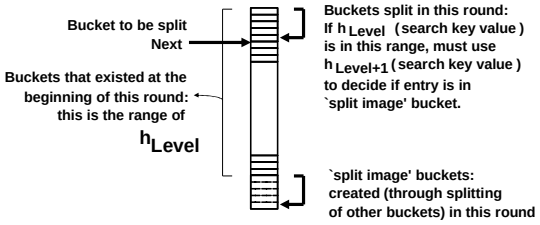


Inserting h(r) = 50 (End of a Round)



Overview of LH File

- In the middle 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 uses directory doubling to avoid overflow pages
  - Duplicates may require overflow pages
- Linear hashing avoids directory by splitting in rounds
  - Naturally handles skew and duplicates
  - Uses overflow buckets (but not very long in practice)