Hash-Based Indexing

Tree Indexing Summary

- Static and dynamic data structures
  - ISAM and B+ trees
- Speed up both range and equality searches
- B+ trees very widely used in practice
- ISAM trees can be useful if dataset relatively static (not a lot of overflow pages)
  - Because index is static, no need to lock index pages

Indexing using Hashing

- *Hash-based* indexes are for *equality selections*. ***Cannot*** support range searches.

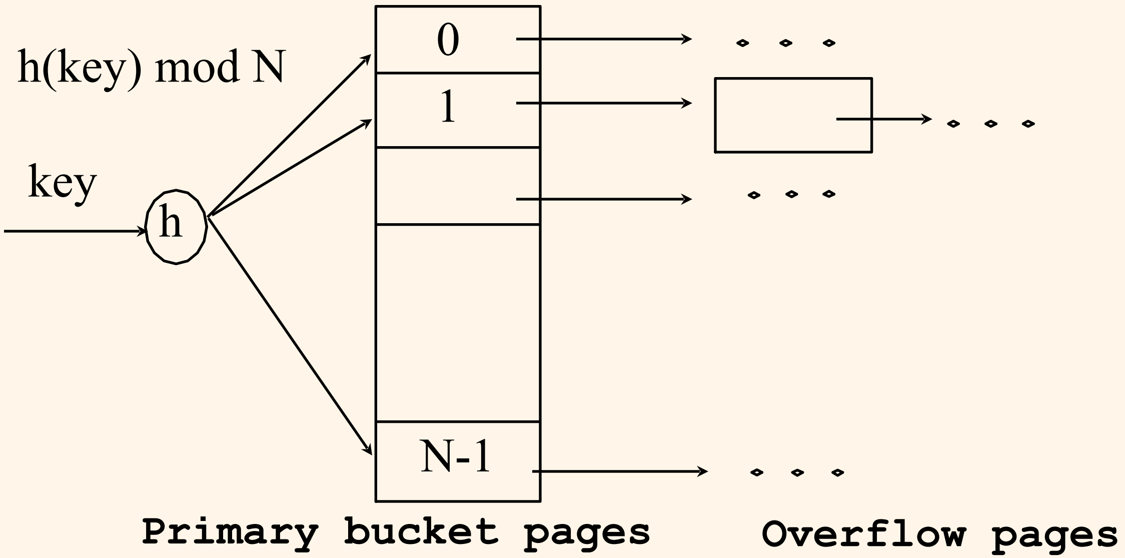
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Indexing Using Hashing

- *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**>

Static Hashing

- For each key k, compute some *hash function* h(k)
- **h**(*k*) mod N = bucket to which data entry with key *k* belongs. (N= # of buckets)



Static Hashing

- Hash fn works on *search key* field of record *r*.
  - $h(k)$ *mod* N must distribute values over range 0 ... N-1.
  - $h(k) = (a * k + b)$ usually works well.
  - a and b are constants that can be used to tune h

Static Hashing

- Primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed
  - If no overflow pages, lookup just one disk I/O
  - Overflow pages degrade performance

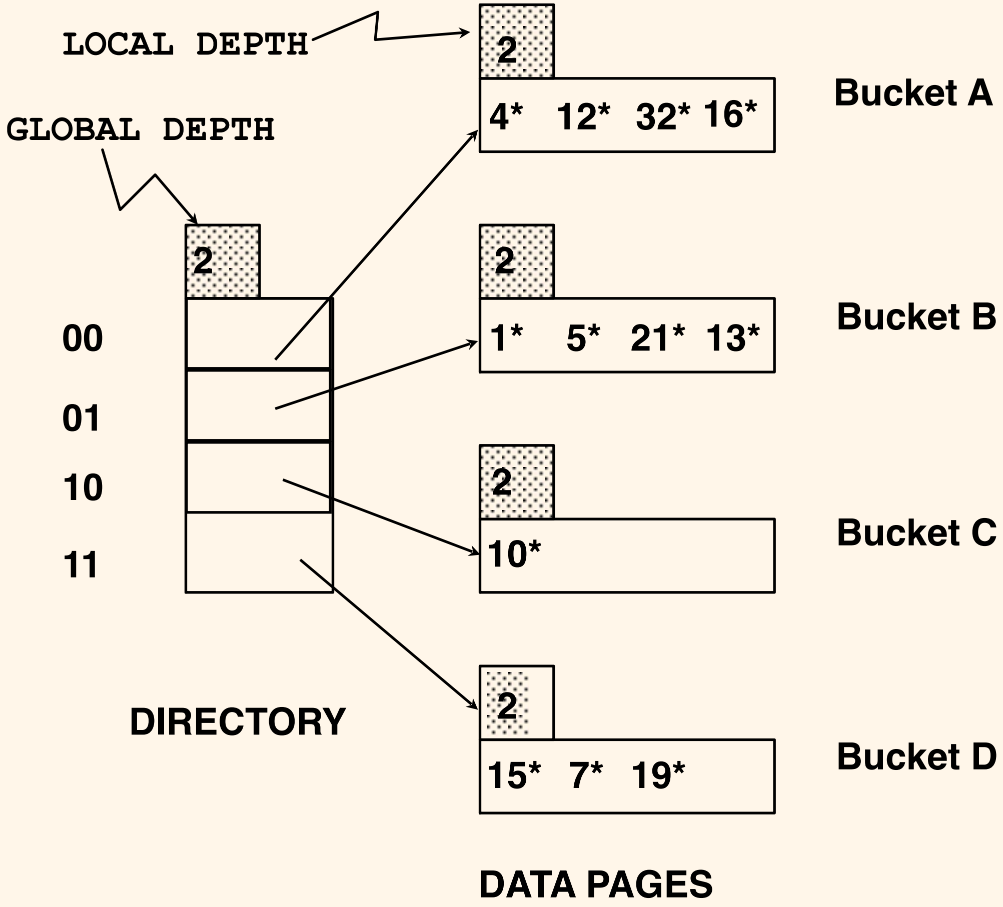
Static Hashing

- If too many overflow pages, can **rehash** (reorganize into more buckets)
  - E.g. double the number of buckets
- Problems:
  - Takes time if index is big
  - Index cannot be used during rehashing

Extendible Hashing

- Better idea: split only the bucket that has overflowed
- Keep a *directory* of pointers to buckets, so searches for the old bucket are properly directed to the two new ones
  - When bucket is split, only directory needs to be adjusted, and this is pretty small

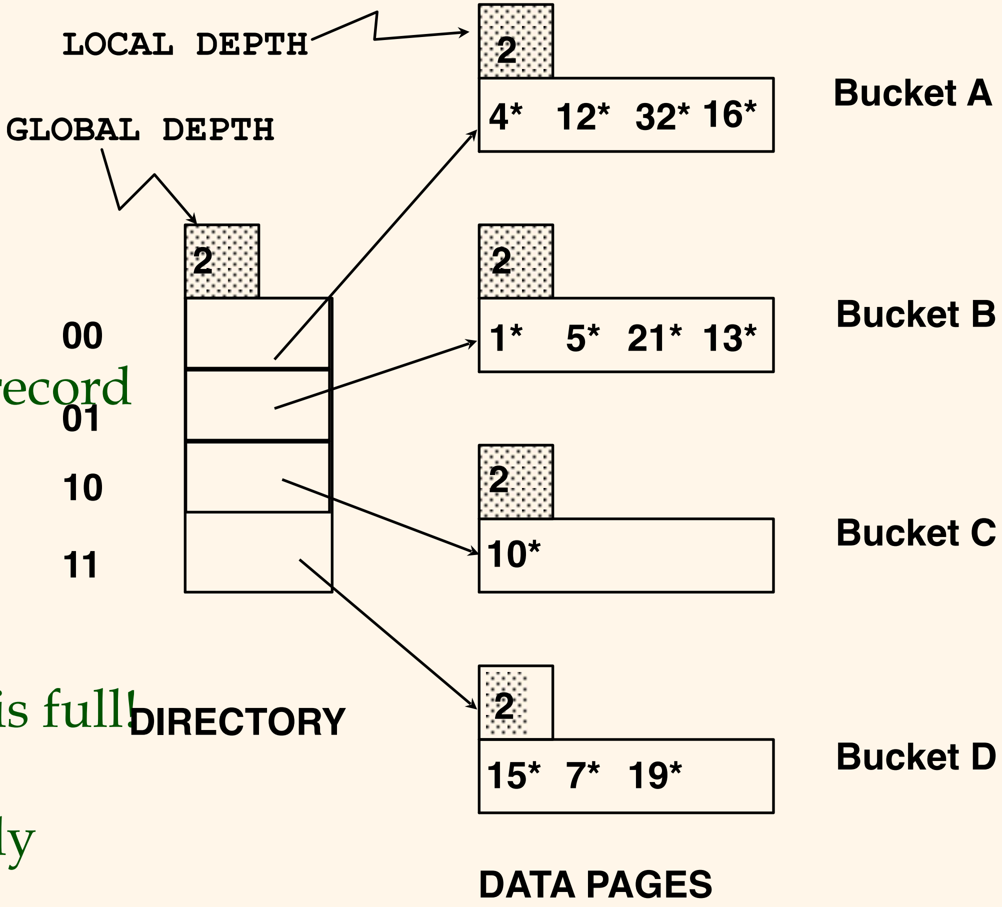
Example



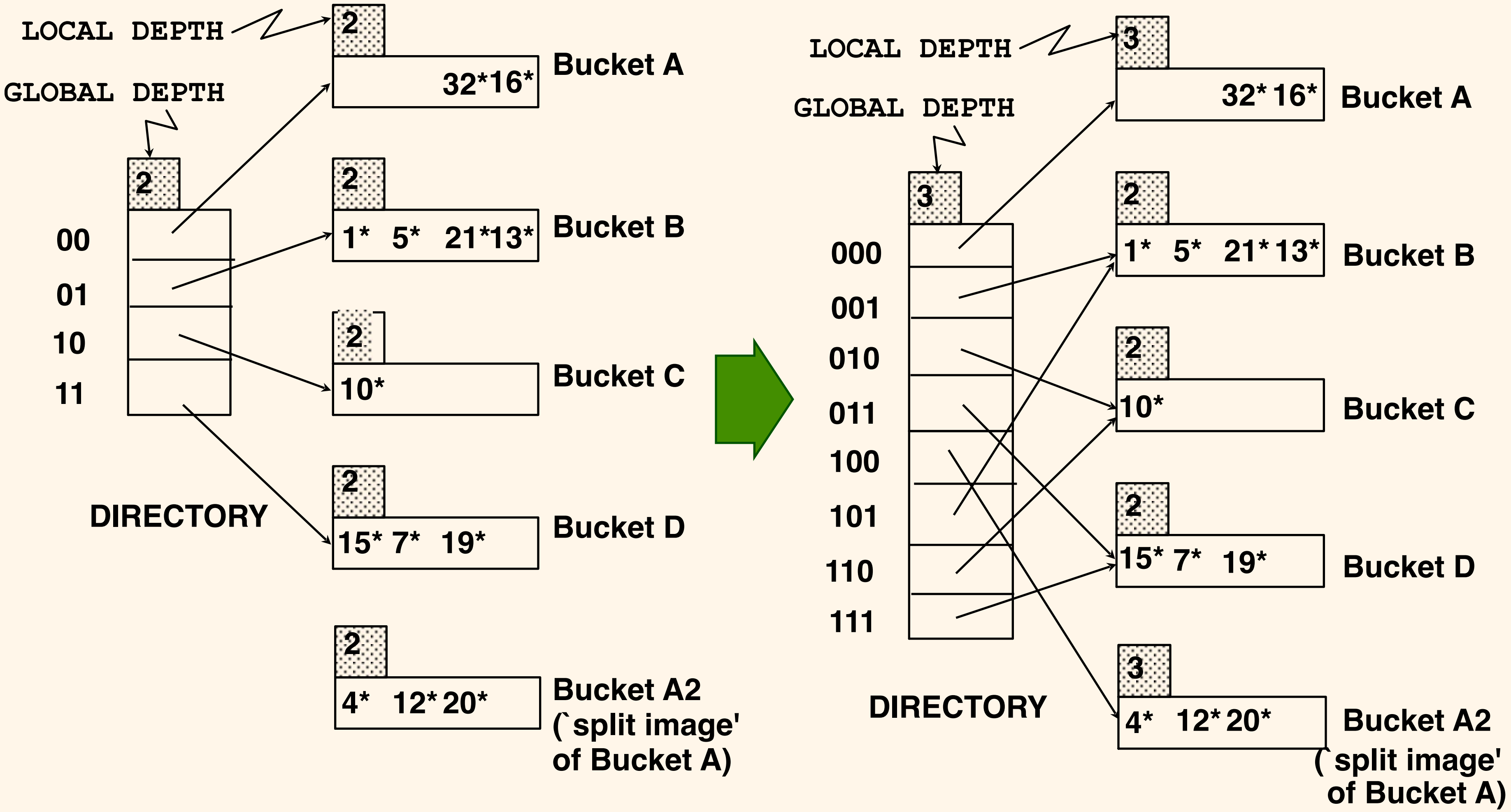
- Notation: 1* = data entry with *hash value 1*
- To find bucket for record with key k, look at last 2 bits of h(k)
- E.g. if h(k) = 14 i.e. 1110, goes into 3rd bucket

Example

- Now suppose want to insert record with h(k) = 20
- 20 in binary is 10100
- Belongs in first bucket which is full!
- Let's split that bucket (and only that one) into 2
- How to distribute entries among new buckets? Look at 3rd bit from the right!



Insert ***h***(*r*)=20 *(Causes Doubling)*



Global and local depth

- *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.
  - Bucket C has local depth 2, so all entries will share the same last 2 bits but may have a different 3rd-to-last one!

Points to Note

- Splitting does not always require directory doubling!
  - E.g. if we now insert lots of values into Bucket C and need to split it, no need to double directory
- When does bucket split cause directory doubling?
  - When its local depth pre-split was equal to the global depth!

What about deletions?

- 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.

- In practice may omit this step and leave index as is
  - Relatively common for other indexes too e.g. B+ trees
  - Justification: deletes typically less frequent than inserts

Performance of Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
- Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.

Performance of Extendible Hashing

- Doubling the directory is a "stop the world" operation and may still take a while
- May still need overflow pages if we have hash collisions (two data entries/search keys have *same hash value*)

Linear Hashing

- Another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH is similar to EH but does not use a directory
  - Allows some (more) overflow pages instead

Linear Hashing

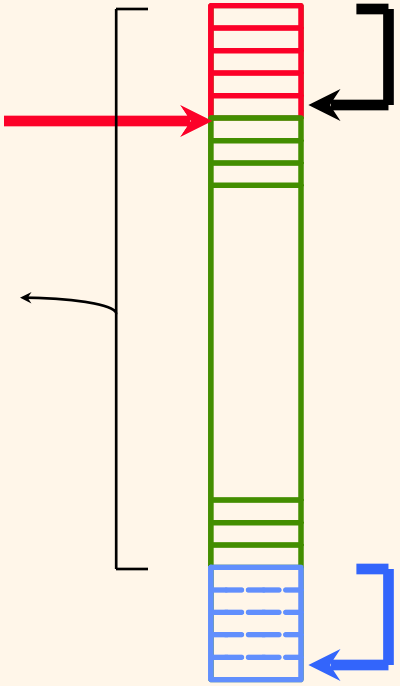
- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
- Do not necessarily split the bucket that has overflowed, but rather the bucket whose "turn" it is.

Linear Hashing

- Splitting proceeds in rounds. Round ends when all N_R initial (for round R) buckets are split.
- Current round number is *Level*.

Overview of LH File

- In the middle of a round.


Bucket to be split
Next
Buckets that existed at the beginning of this round:
Buckets split in this round:
`split image' buckets: created (through splitting of other buckets) in this round

The big question

- How do we quickly find the right bucket for search key k?
- After all, that's the whole reason we are doing hashing…
- Linear hashing: have TWO hash functions in play at all times

Hash function families

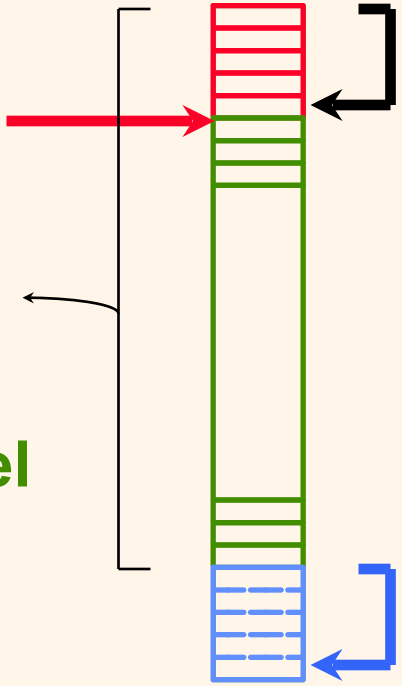
- Use a family of hash functions: h_0, h_1, h_2
- Range of h_{i+1} is twice the range of h_i
- Can generate them from some "starter" function h that maps search keys to integers
- Say initial number of buckets is $N = 2^d$
- Can take h_0 as last *d* bits of h(k)
- And h_i is last *d*+*i* bits of h(k)
- E.g. suppose N = 4…

Why we don't need a directory

Bucket to be split

Next

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



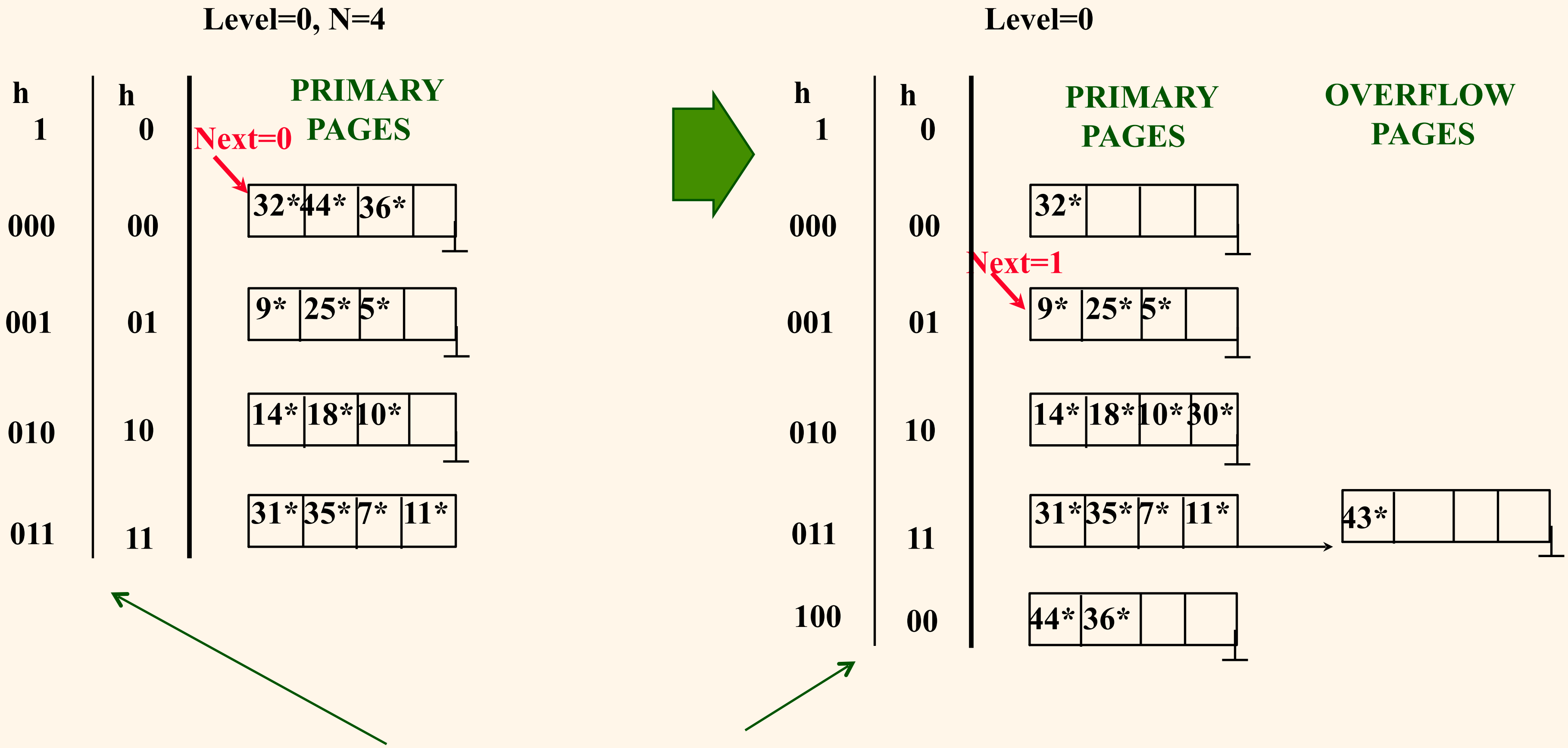
Buckets split in this round: If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in `split image' bucket.

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

Why we don't need a directory

- For search, need to know which *Level* (round) we are in.
- To find bucket for data entry *r*, find $\mathbf{h}_{Level}(r)$:
  - If *Next* <= $\mathbf{h}_{Level}(r)$ <= 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.

*Example – inserting 43**



Level=0, N=4

h_1	h_0	PRIMARY PAGES
000	00	32* 44* 36*
001	01	9* 25* 5*
010	10	14* 18* 10*
011	11	31* 35* 7* 11*

Next=0

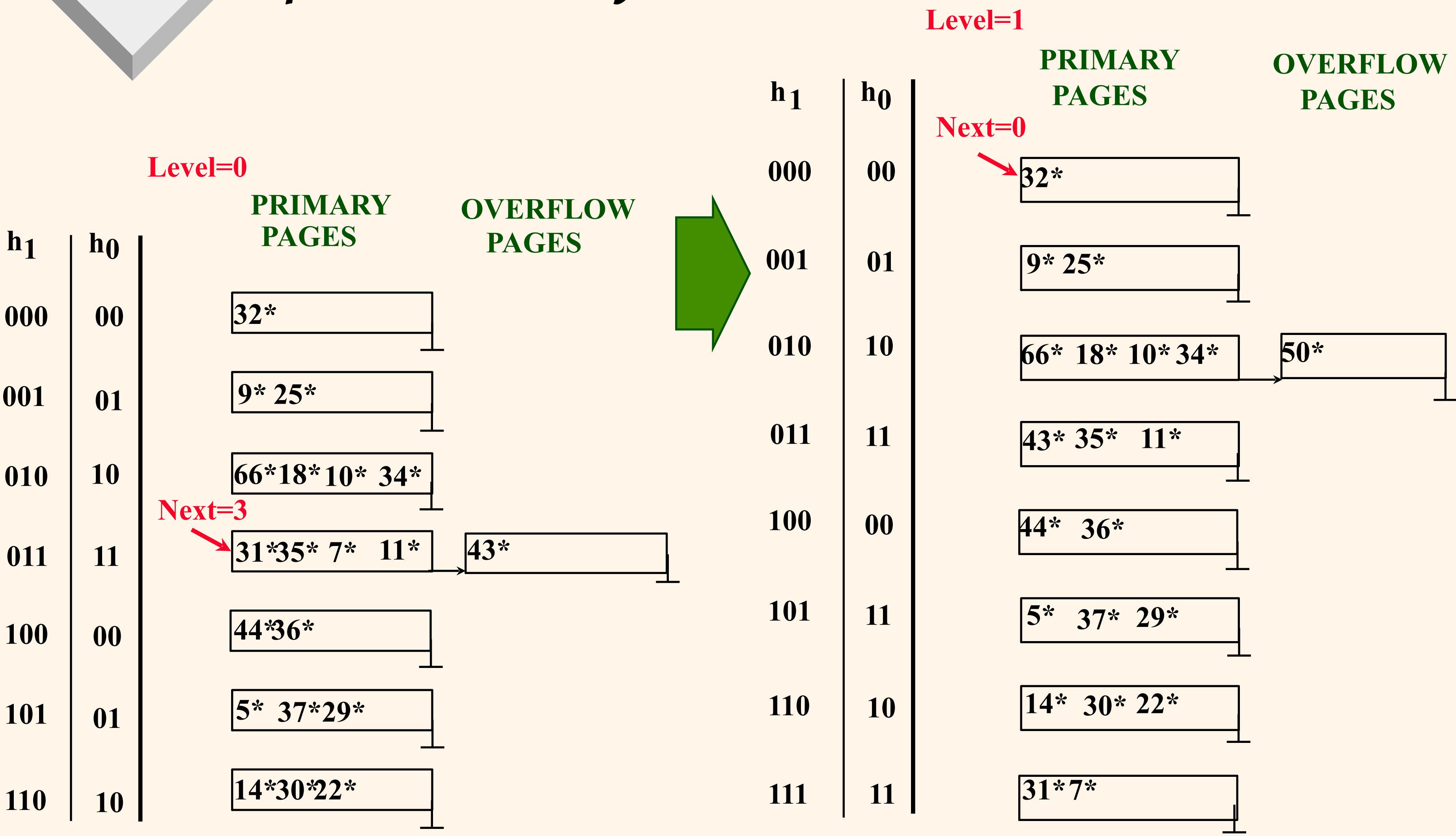
Level=0

h_1	h_0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25* 5*	
010	10	14* 18* 10* 30*	
011	11	31* 35* 7* 11*	43*
100	00	44* 36*	

Next=1

Note this is NOT a directory

*Example: End of a Round, insert 50**



Level=0

Next=3

h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66*18*10* 34*	
011	11	31*35* 7* 11*	43*
100	00	44*36*	
101	01	5* 37*29*	
110	10	14*30*22*	

Level=1

Next=0

h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	50*
011	11	43* 35* 11*	
100	00	44* 36*	
101	11	5* 37* 29*	
110	10	14* 30* 22*	
111	11	31* 7*	

Linear Hashing Continued

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ or $\mathbf{h}_{Level+1}$:
  - 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, eg. desired occupancy
- Similarities/differences to doubling directory in EH

Linear Hashing Continued

- **Delete:** can be implemented as reverse of insert, or just ignored (leave index as is)

- Cost of lookup: 1 I/O if primary bucket pages consecutive on disk

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.

Summary

- Extendible Hashing 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.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - 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.