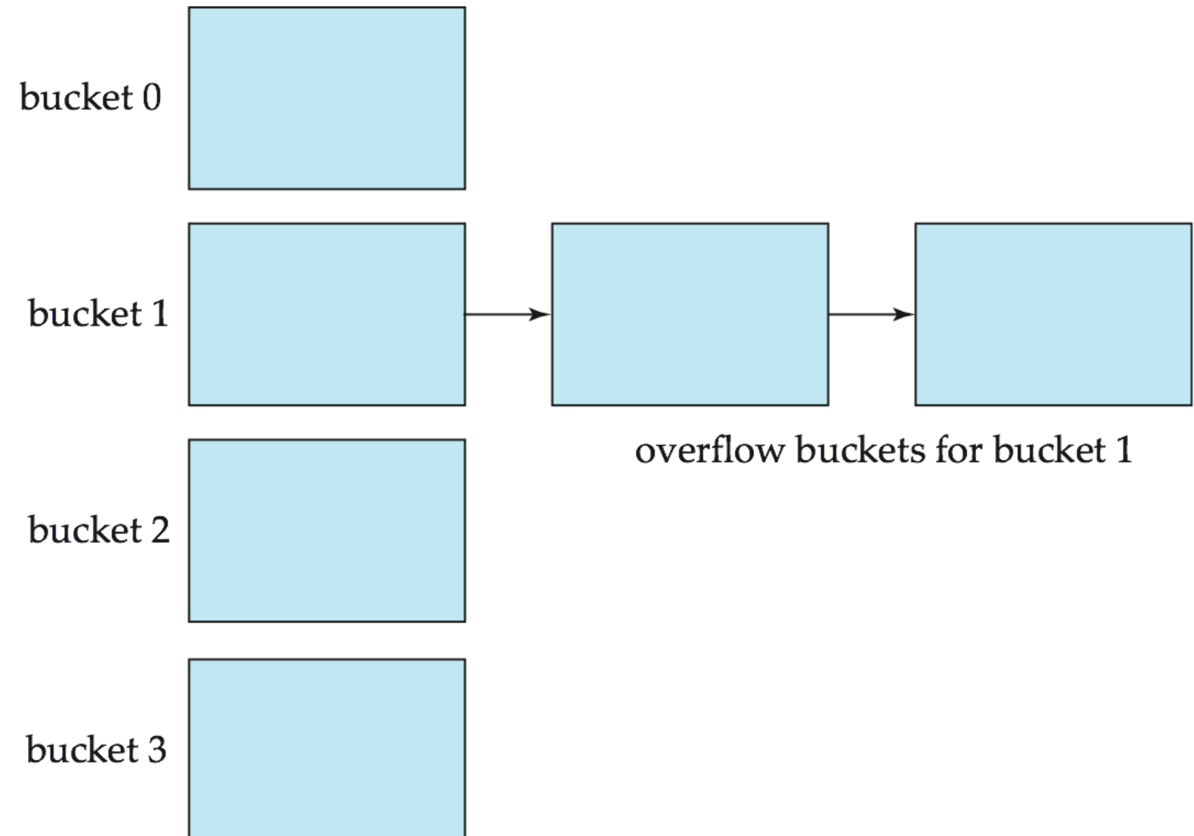
BBM371- Data Management

Lecture 7: Hash-based Indexing

15.11.2018

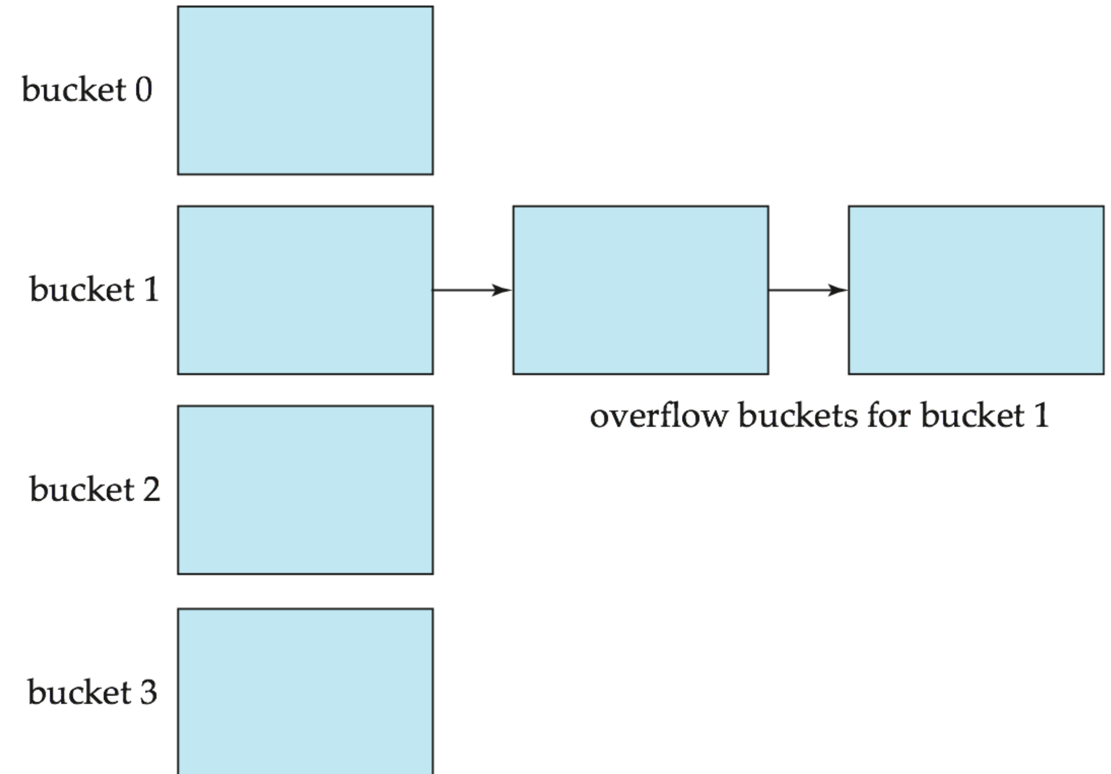
Static Hashing

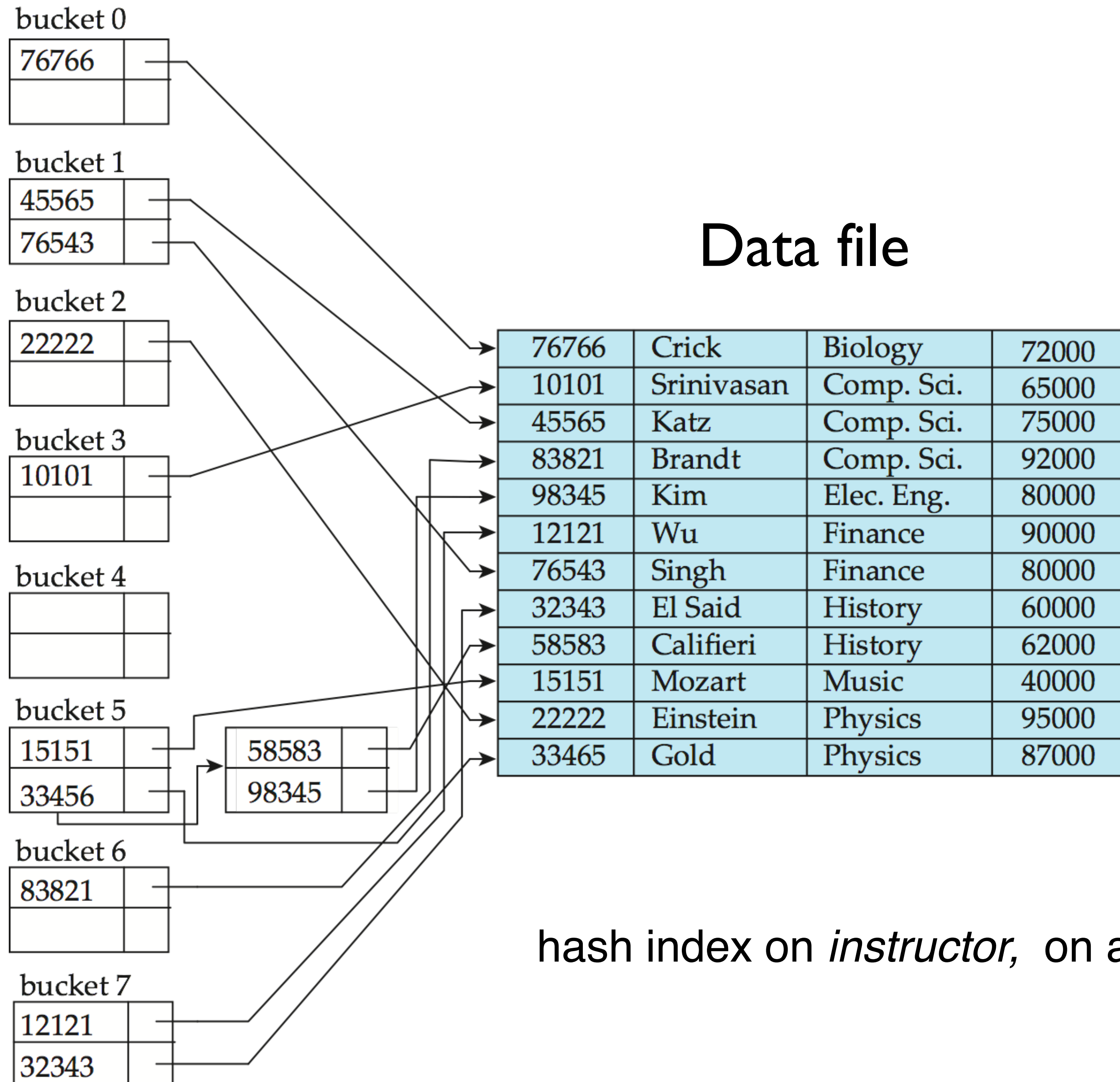
- Hash tables in the memory can be used in secondary storage as well
- 0 to B-1 indexes where B is the number of Buckets
  - Buckets is the block or blocks associated with an index
- Records are stored in buckets indexed with h(k)
  - k: search key of record, h(k) is the has of search key



Static Hashing

- If a bucket has too many records, a chain of overflow blocks can be added
- We can find the first block of each bucket with index i (without a disk access):
  - We can have an array in main memory with pointers to blocks
  - We have continous allocation of buckets and buckets are fixed length





76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor*, on attribute *ID*

Static Hashing Insert

- Insert(record r, search key k)
- If Bucket h(k) has space;
  - Insert r to the first available block (can be either overflow or first block)
- All blocks for bucket h(k) are filled
  - Add a new overflow block to the chain
  - Add record to new block

Static Hashing Delete

- Delete(all records with search key k)
- Read first block of h(k) bucket
- Search for the records with key k, any found is deleted
  - So we must search overflow blocks as well
- Can move records to empty spaces in earlier blocks.
- Optionally, when an overflow block is not needed anymore remove the block
  - Must avoid oscillations; frequent additions and removal to the same bucket can cause frequent block allocation and release.

Efficiency of Static Hashing

- Ideally if there are enough buckets;
  - Most records fit on one block of a bucket
  - Insertion or deletion takes only two disk I/O
- If file grows then long chains of blocks needed for buckets
- Solution: increase the number of buckets B dynamically
- Increasing B requires re-indexing from scratch
  - Cannot allow long and expensive operations for normal use of databases.

Dynamic Hashing

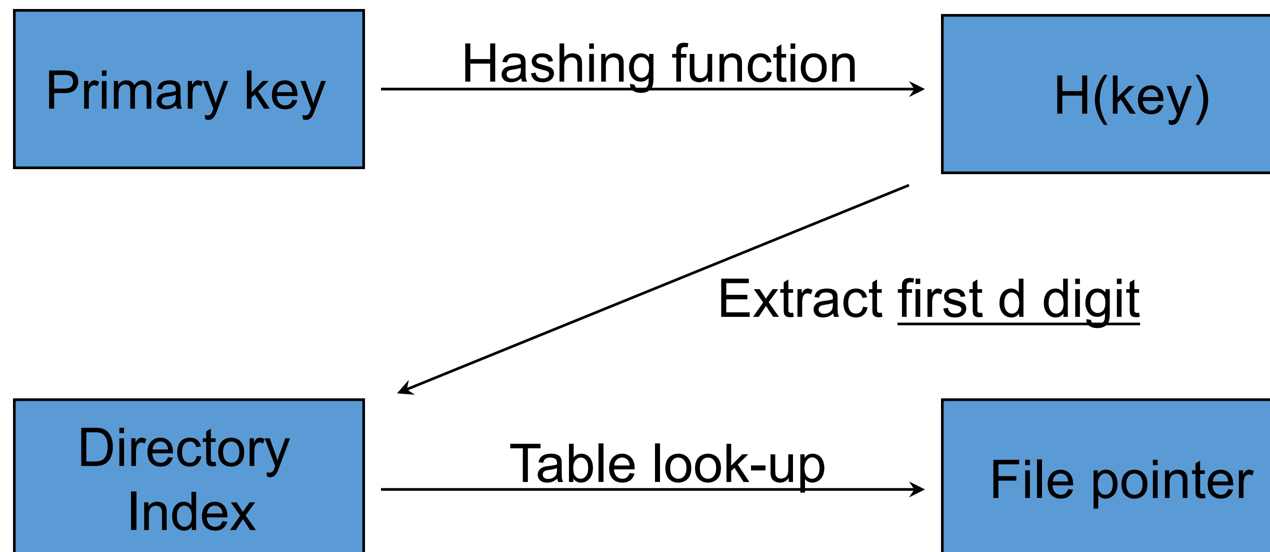
- Dynamic = Changing number of Buckets B dynamically
- Two methods
  - Extendible (or Extensible) Hashing: Grow B by doubling it
  - Linear Hashing: Grow B by incrementing it by 1
- To save storage space both methods can choose to shrink B dynamically
  - Must avoid oscillations when removes and additions are both common.

Extendible Hashing

- *Idea*: Use *directory of pointers to buckets,*
- double # of buckets B by *doubling the directory*, splitting just the bucket that overflowed!
- Directory much smaller than file, so doubling it is much cheaper.
- Only one page of data entries is split. *No overflow blocks*
- Trick lies in how hash function is adjusted!

Extendible Hashing Overview

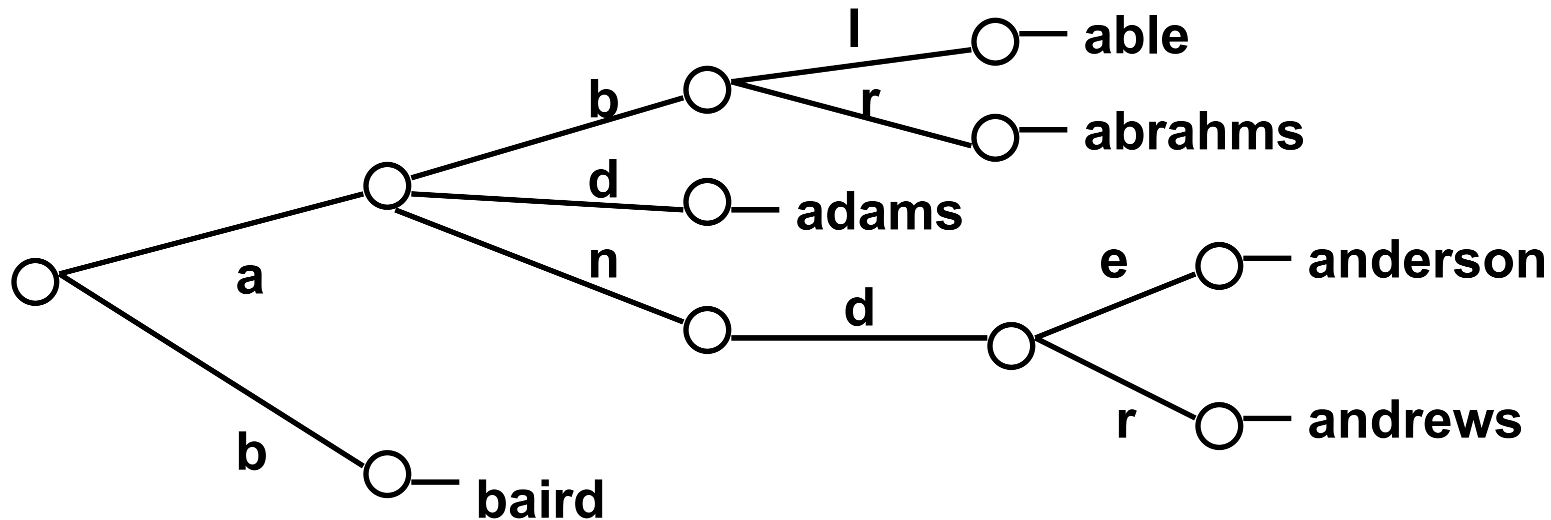
- Extendible Hashing



Hash Function Similar to Tries

- Idea from Tries file (radix searching)
  - The branching factor of the tree is equal to the # of alternative symbols in each position of the key

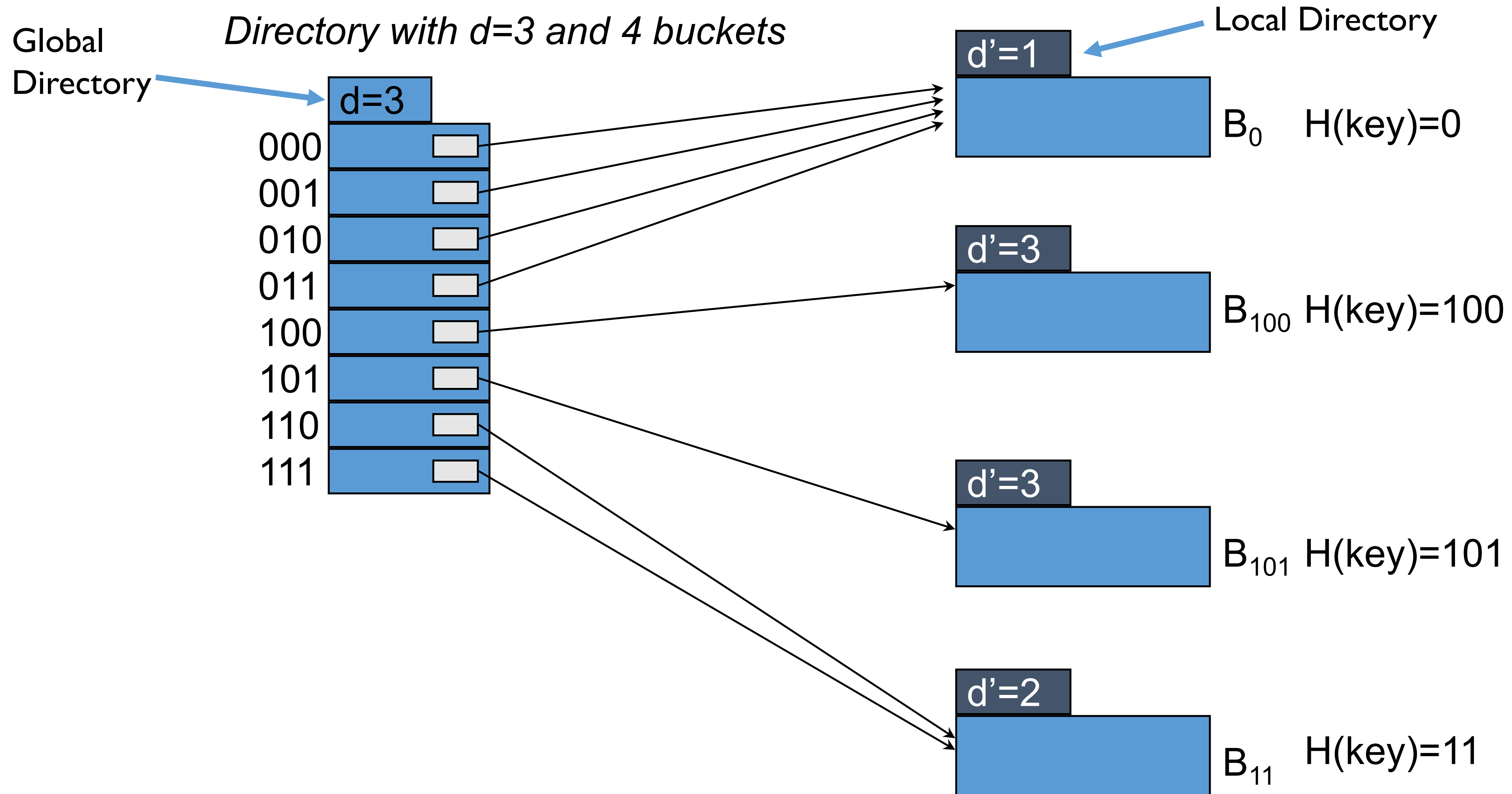
    e.g.) Radix 26 trie - able, abrahms, adams, anderson, adnrews, baird
  - *Use the first d characters for branching*



Extendible Hashing

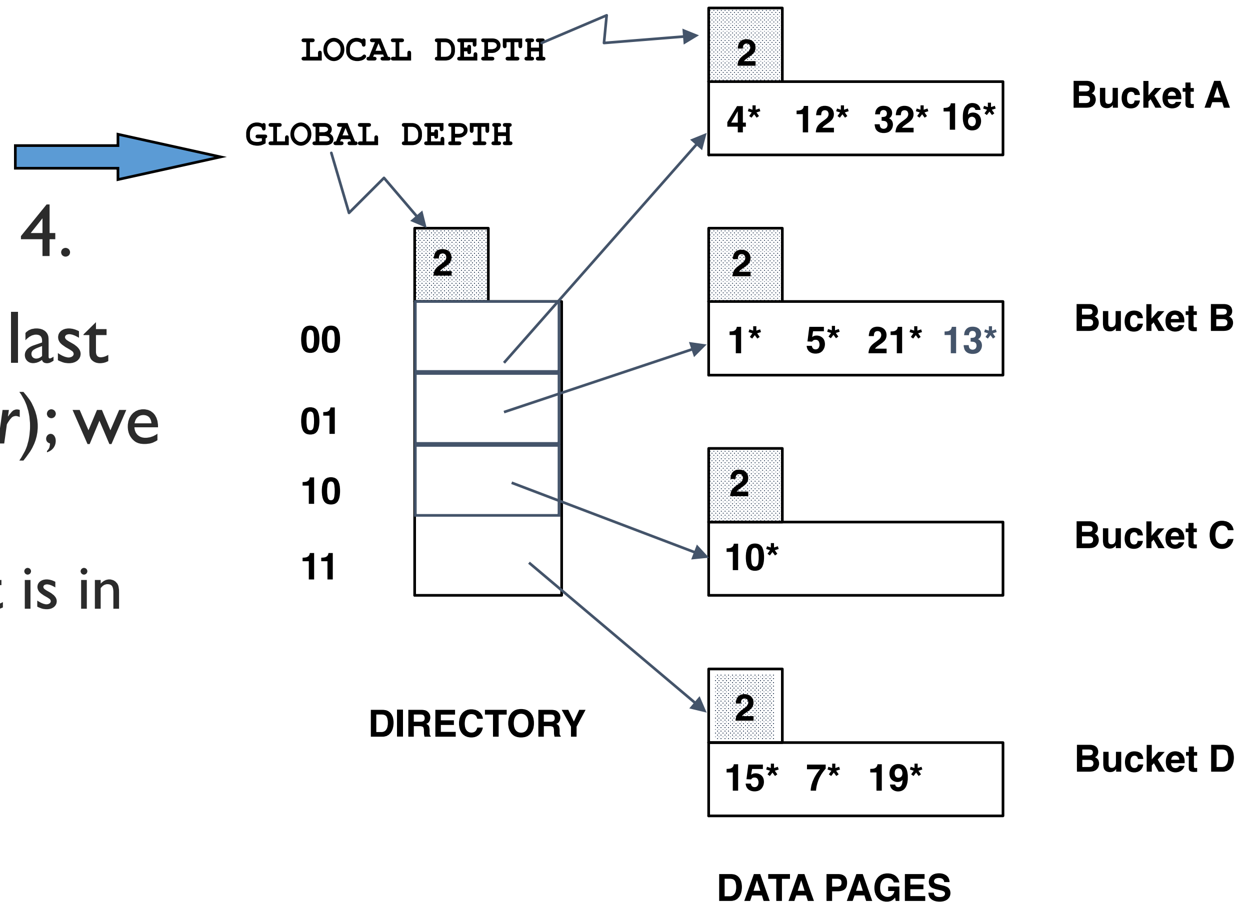
- h(k) maps keys to a fixed address space, with size the largest prime less than a power of 2 ($65531 < 2^{16}$)
- File pointers point to blocks of records known as buckets, where an entire bucket is read by one physical data transfer, buckets may be added to or removed from the file dynamically
- The d bits are used as an index in a directory array containing 2^d entries, which usually resides in primary memory
- The value d, the directory size(2^d), and the number of buckets change automatically as the file expands and contracts

Extendible Hashing Example



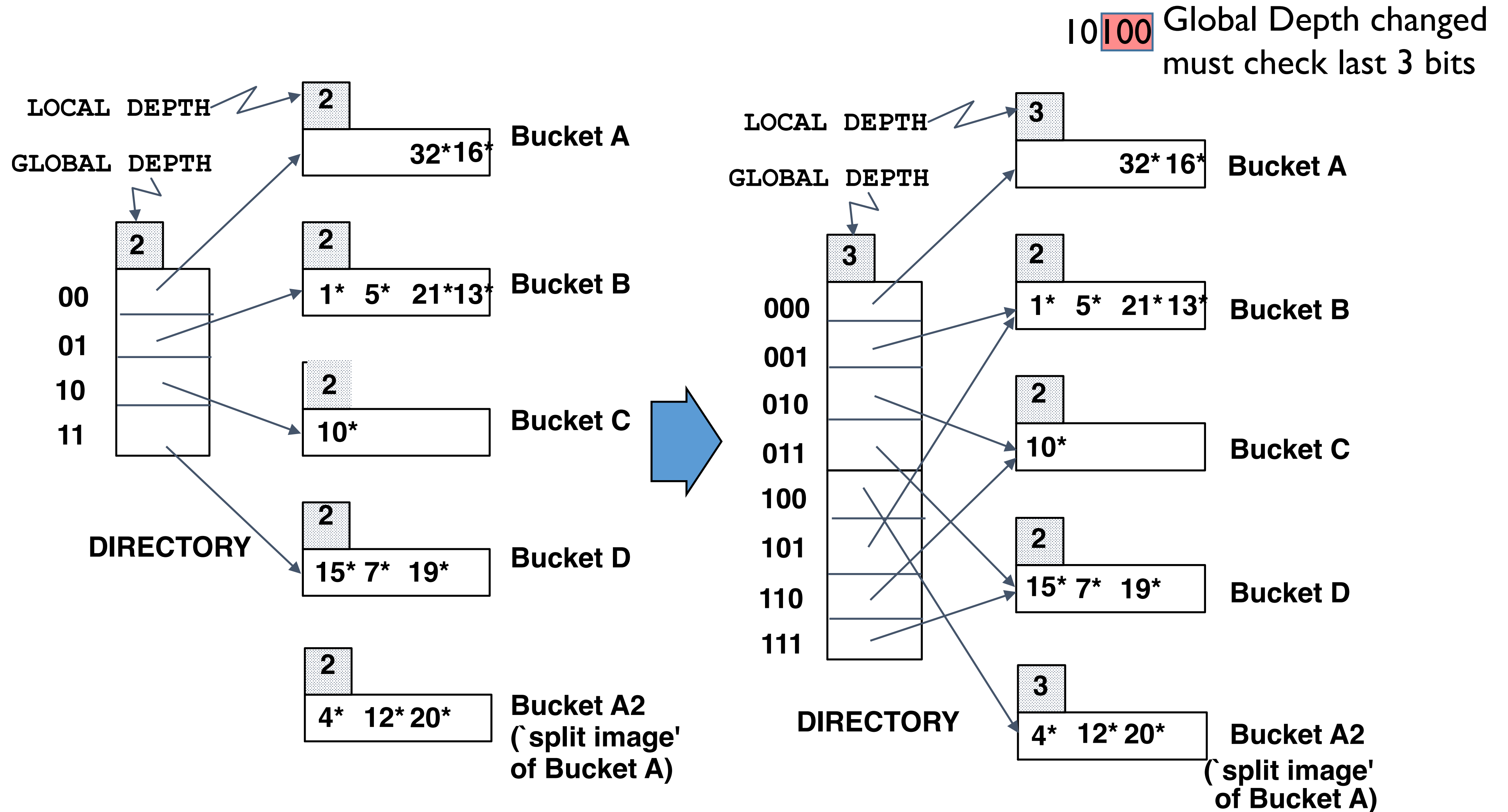
Example

- 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. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

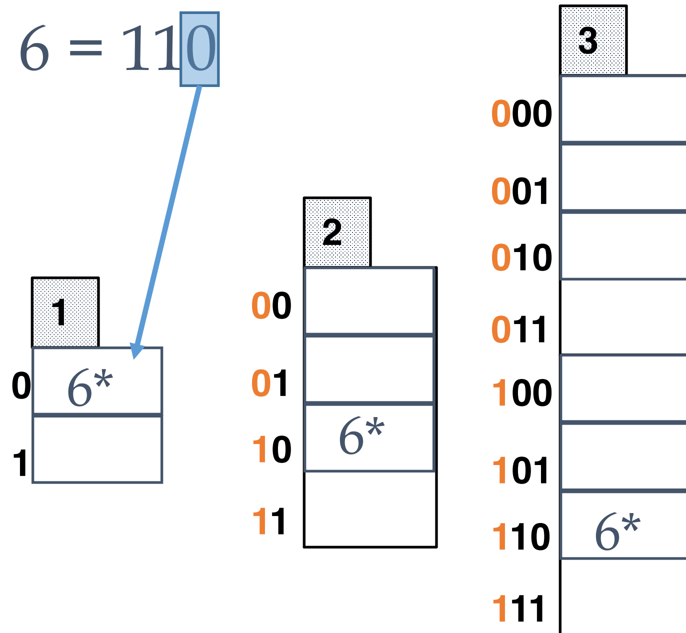
Insert h(r)=20 (Causes Doubling)

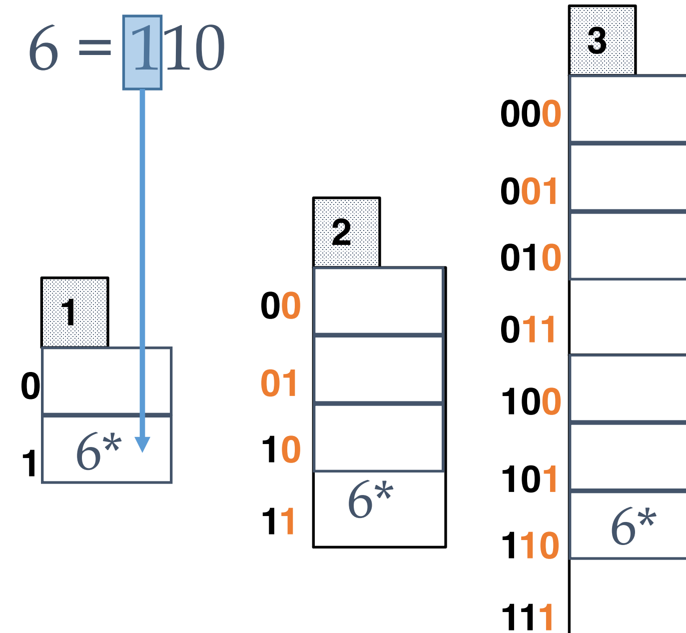


Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **3** 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!)

Directory Doubling



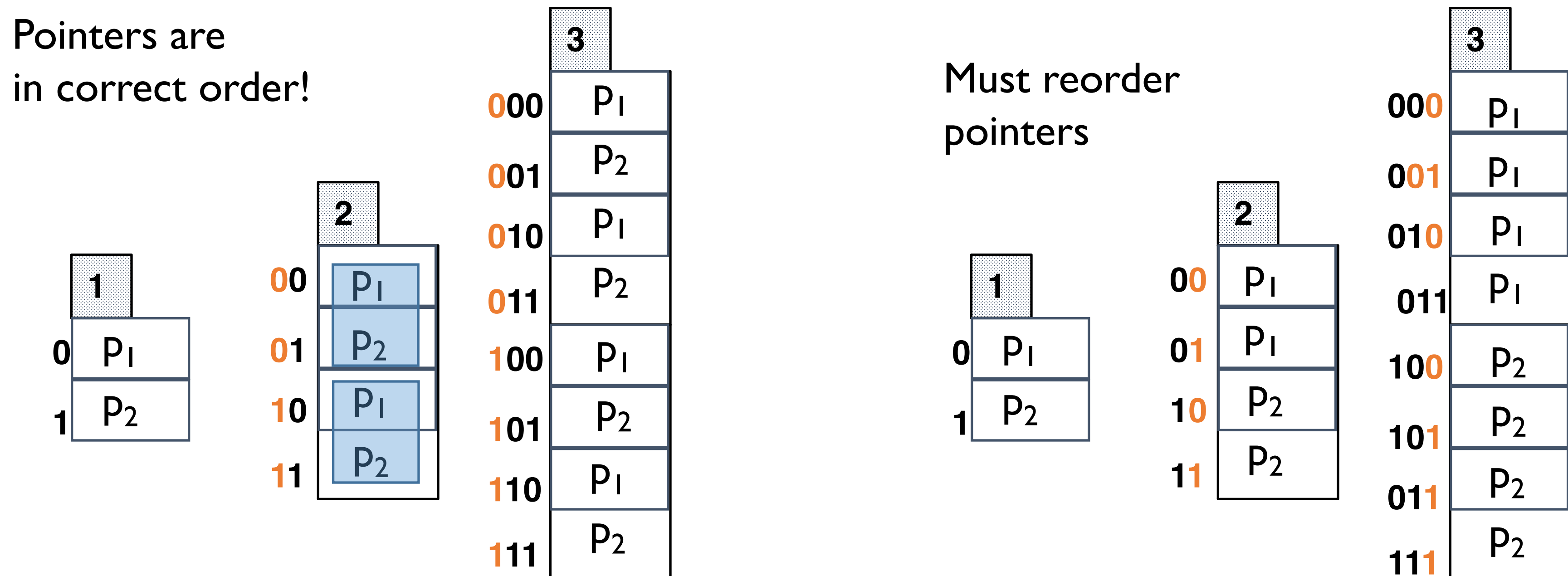


Least Significant vs. Most Significant

Directory Doubling

Why use least significant bits in directory?

→Allows for doubling via copying!



Least Significant vs. Most Significant

Extendible Hashing Adv.

- Can dynamically adjust B as needed
- Directory only stores pointers to buckets, can be small
  - Do we need to store bucket index i?
- Doubling directory can be done easily by copying blocks twice

Extendible Hashing Disadv.

- Need additional Access to directory
  - But if it is small we can keep it in memory
- Must store local directory
- Can double directory unnecessarily if all keys are mapped to same bucket
  - Edge case? How many buckets will be formed if all records map to 0000000
- If directory is larger than memory will require multiple disk access
  - Doubling will be more expensive
- Buckets are allocated in different times, hard to get continuous areas.

Linear Hashing

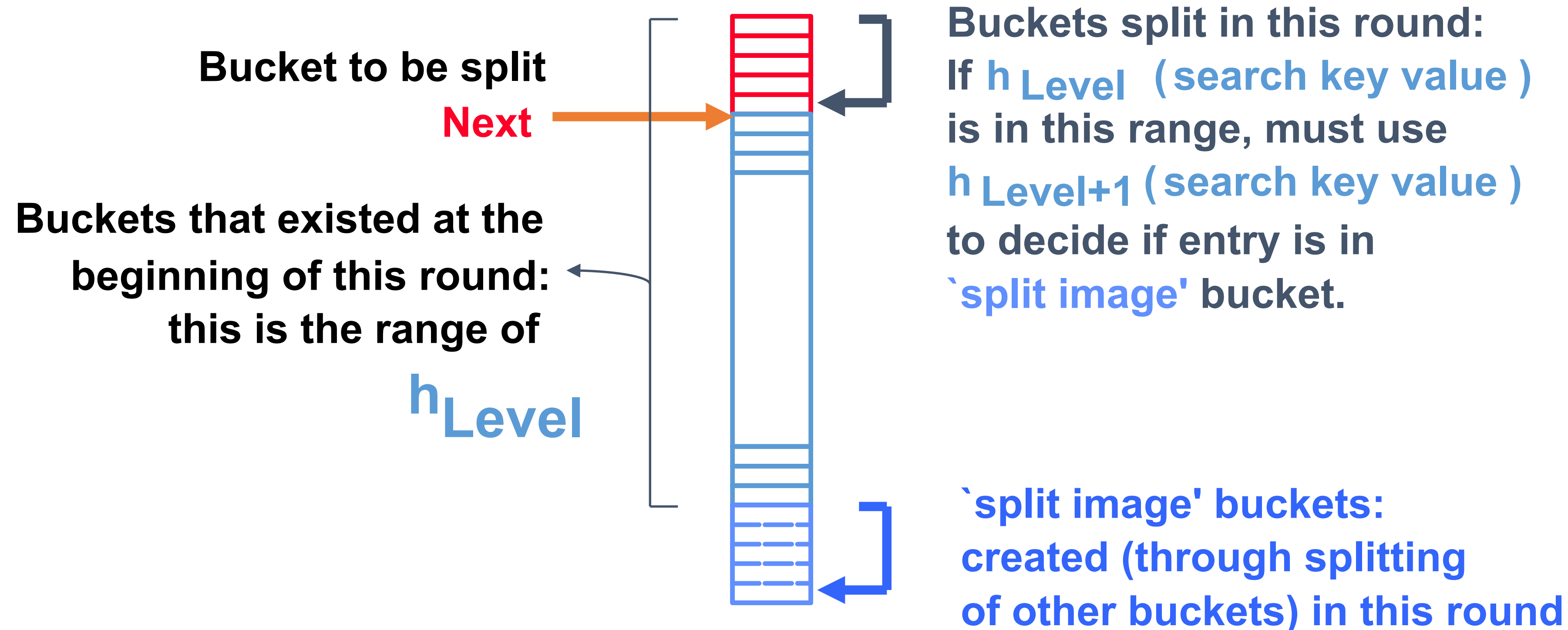
- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory,
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; N = initial # buckets
  - **h** is some hash function
  - 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 using overflow pages, and choosing bucket to split round-robin.
  - 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.



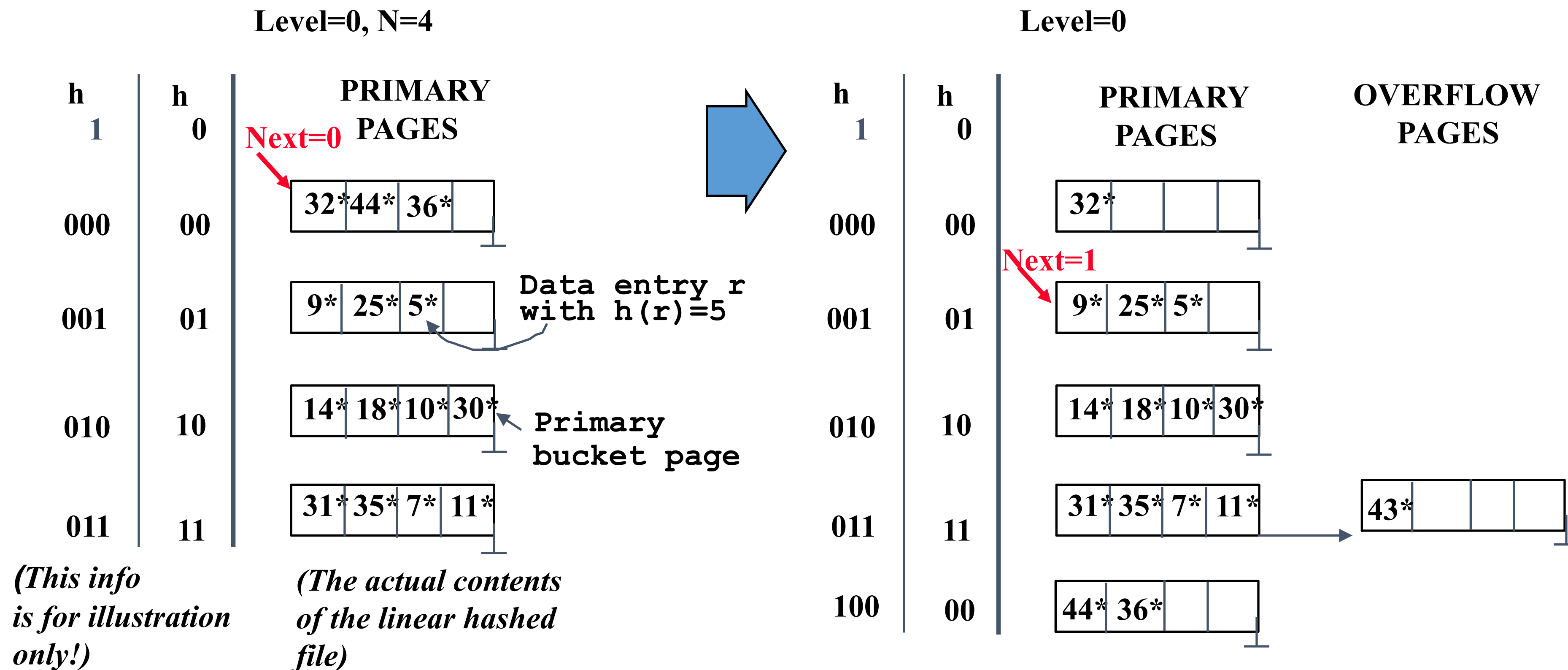
Linear Hashing (Contd.)

- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$(r) or $\mathbf{h}_{Level+1}$(r):
  - 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.
- Since buckets are split round-robin, long overflow chains are avoided
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

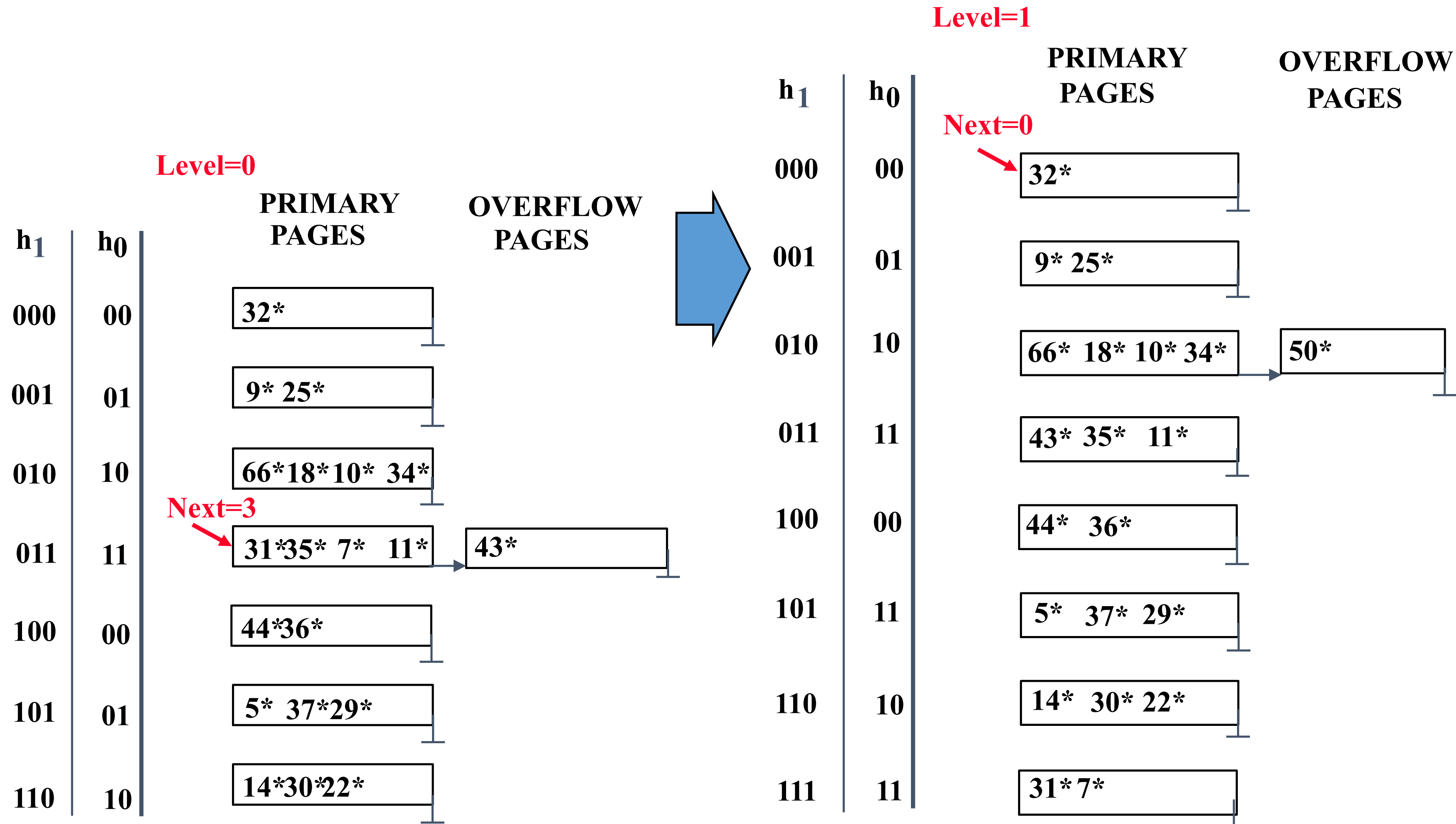
Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries.

insert 43 → adds overflow → triggers split → increments Next



Example: 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 directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be 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!

Indexing vs Hashing

- Hashing good for probes given key

e.g.,

```
SELECT ...
FROM R
WHERE R.A = 5
```

Indexing vs Hashing

- INDEXING (Including B Trees) good for

  Range Searches:

  e.g., SELECT

  FROM R

  WHERE R.A > 5