COP 3540: Introduction to Database Structures

Fall 2017

Hash-Based Indexing

M. Rathod

Hash-Based Indexing

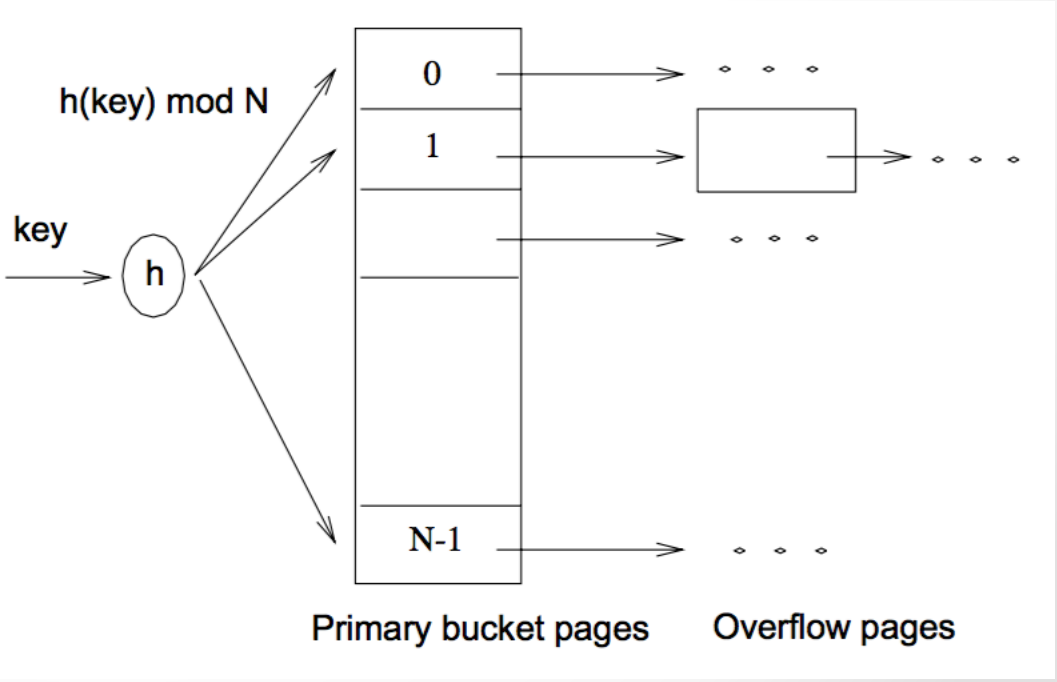
- Hash-based indexing is best for equality searches
- Do not support range searches

We will look at:

- Static Hashing
- Extendible Hashing
- Linear Hashing

Static Hashing

- pages with data are viewed as a collection of **buckets** (0 to *N* -1)
- one **primary page** with **overflow** pages
- buckets contain data entries
- **hash function h** is applied when searching for data entry to identify bucket



Static Hashing

Insert:

- use hash function to identify the correct bucket and put entry
- if no space, allocate a new overflow page and add the page to the overflow chain of the bucket

Delete:

- use hash function to identify the correct bucket, locate the data entry, remove it
- if data entry is the last in an overflow page, page is removed and added to a list of *free pages*

Static Hashing

Hash function:

- distributes values in the domain of the search field uniformly over the collection of buckets
- for N buckets, numbered 0 through $N - 1$
  function h is $h(value) = (a * value + b)$
- bucket identified is $h(value) \bmod N$
- constants a and b can be chosen to 'tune' the hash function

Static Hashing

- #of buckets is known, primary pages can be stored on successive disk pages

- hence:
  - one disk I/O for search
  - two I/Os (read and write the page) for insert and delete operations

- # of buckets are fixed
  - if file shrinks, space is wasted
  - file grows, long overflow chains can develop, performance deteriorates

Static Hashing

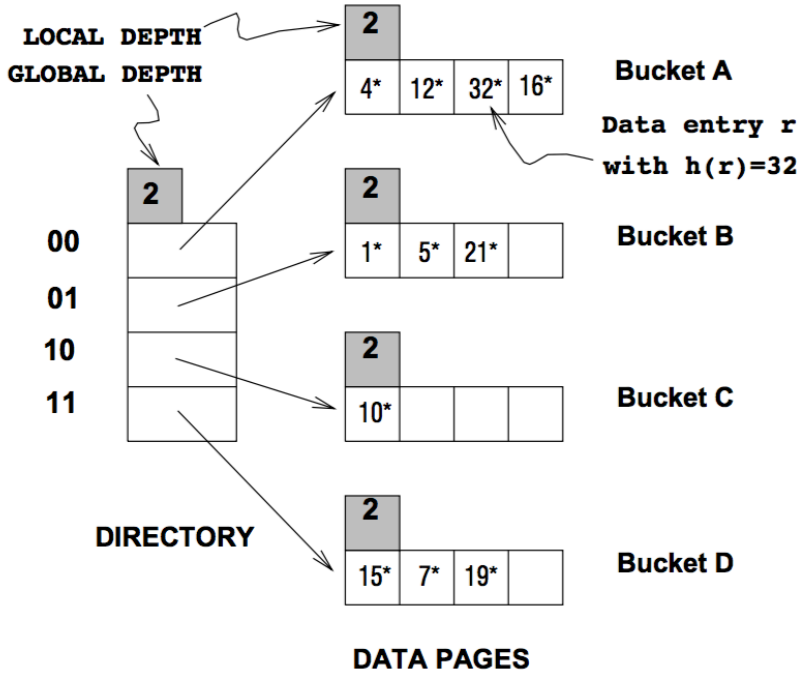
- #of buckets is known, primary pages can be stored on successive disk pages

- hence:
  - one disk I/O for search
  - two I/Os (read and write the page) for insert and delete operations

- # of buckets are fixed
  - if file shrinks, space is wasted
  - file grows, long overflow chains can develop, performance deteriorates

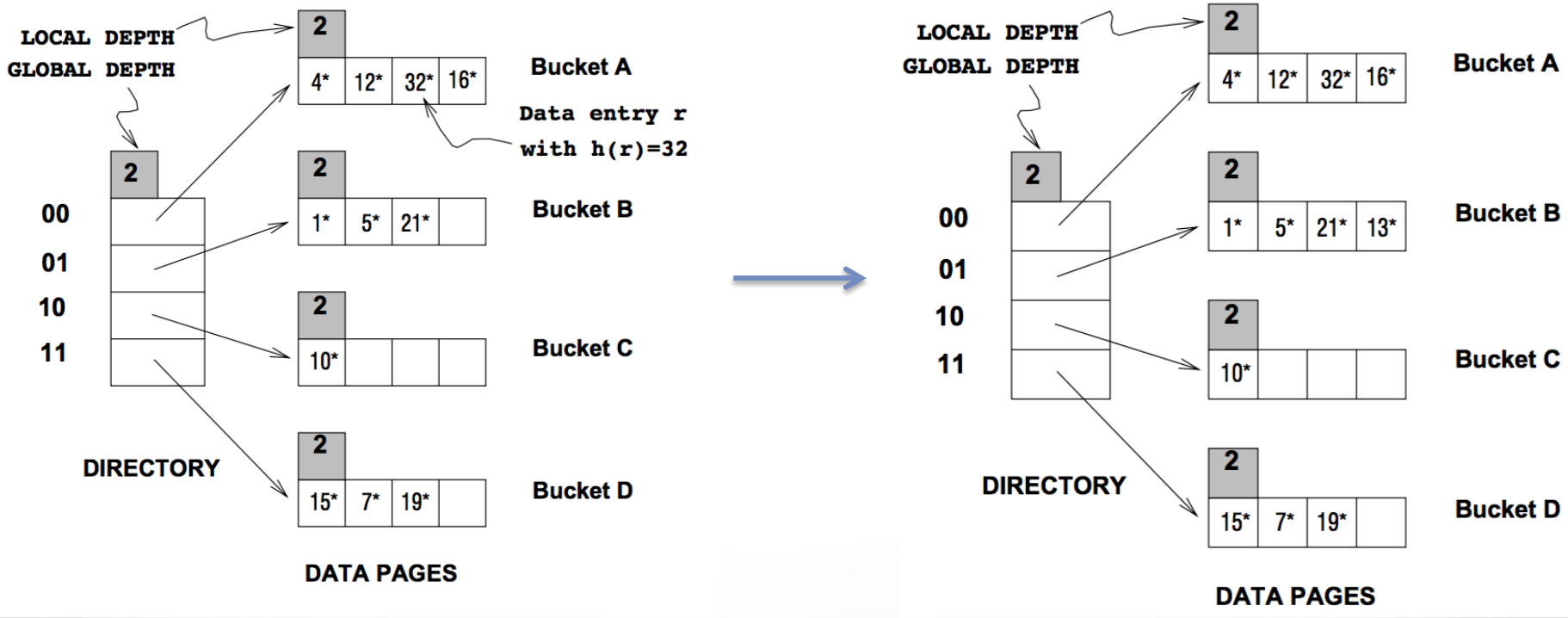
Extendible Hashing

- directory array is of size 4
- each element is a pointer to a bucket
- hash function is applied - last two bits of its binary representation gets number between 0 and 3
- pointer in this position gives us the bucket
- each bucket has 4 data entries
- locating a data entry with hash value 5 (binary 101): look at directory element 01 and follow the pointer to the data page



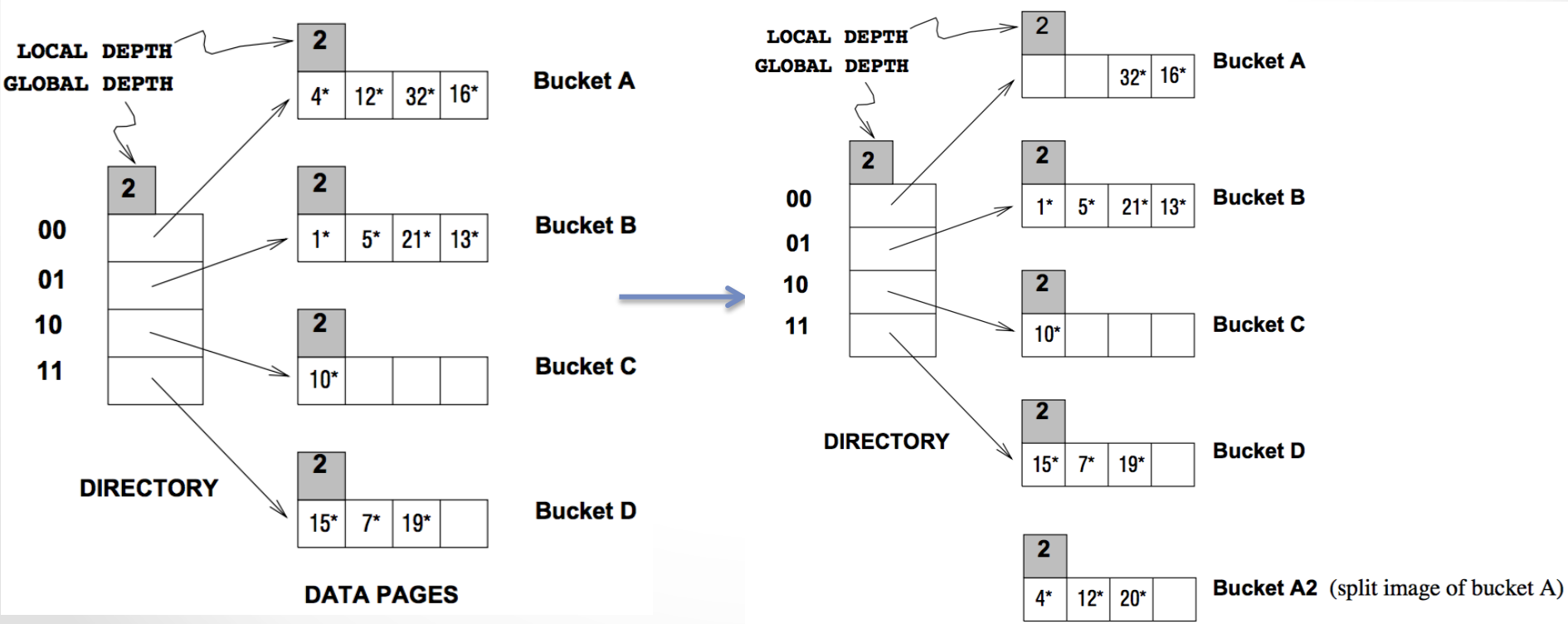
Extendible Hashing

- Insert: data entry with hash value 13*
- look at directory element 01 (bucket B) and go to the page containing data entries 1*, 5*, and 21*
- page has space for an additional data entry



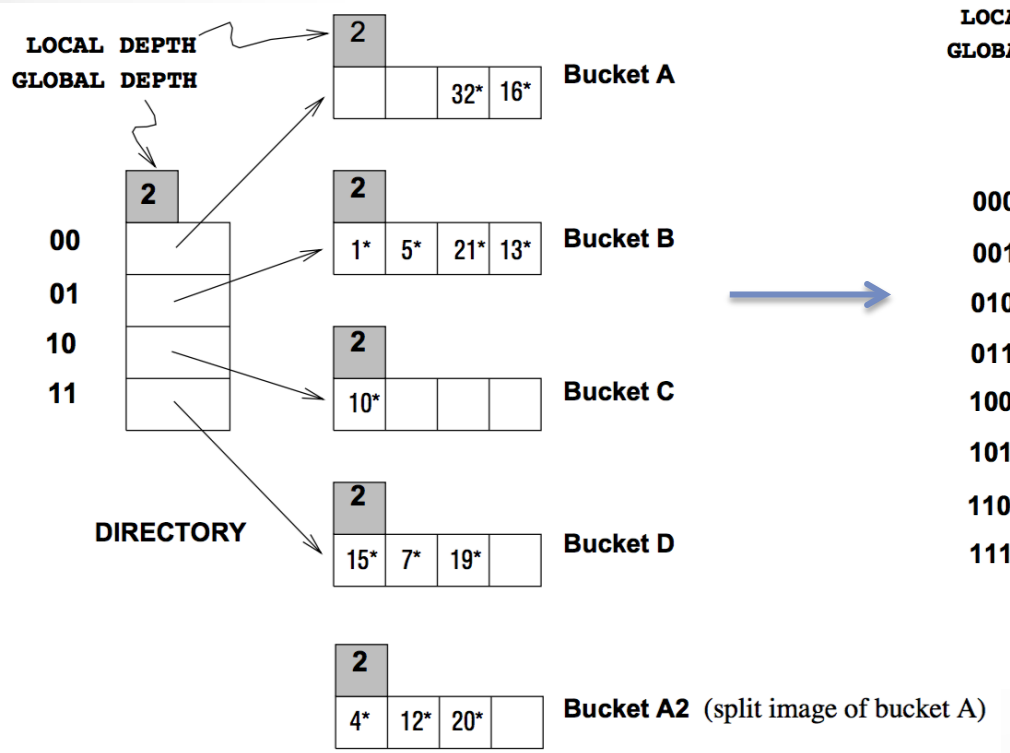
Extendible Hashing

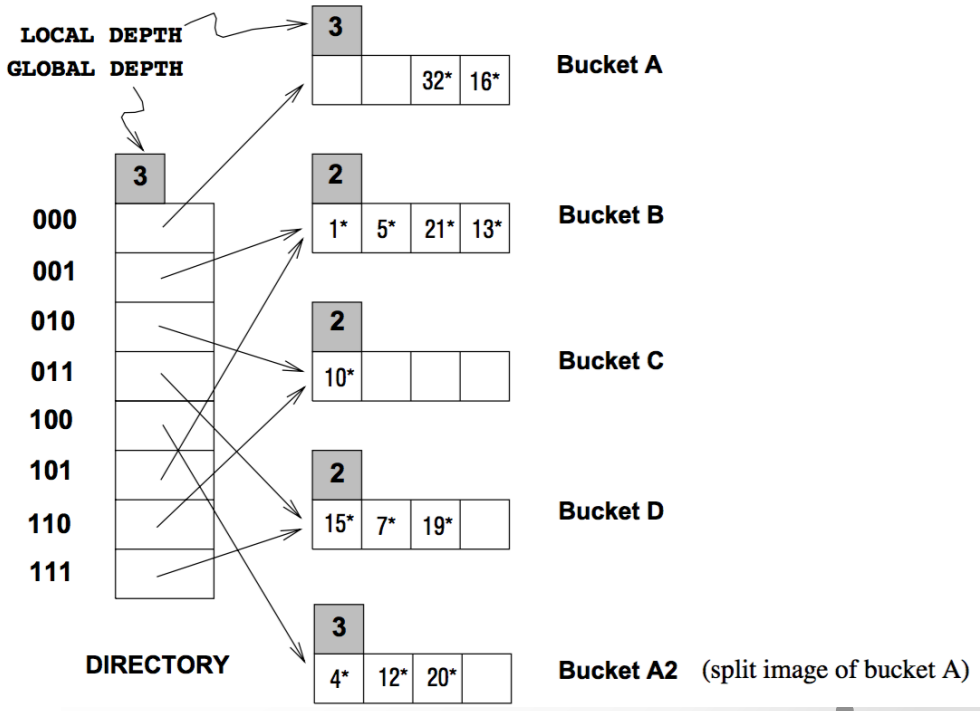
- Insert: data entry 20* (binary 10100)
- for directory element 00, bucket A is already full
- first split the bucket by allocating a new bucket
- redistribute the contents by considering the last three bits of *h(r)*



Extendible Hashing

- Insert: data entry 20* (binary 10100)
- issue is three bits need to discriminate between A & A2
- solution is to double the directory
- apply a hash function *h* as a binary number and to interpret the last *d* bits (global depth)

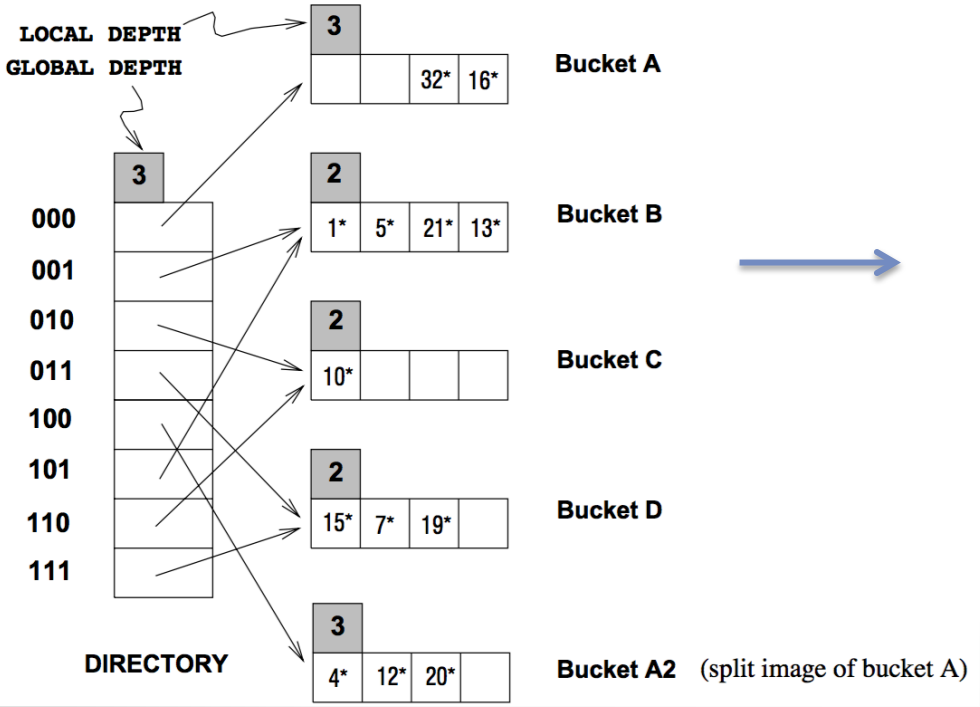


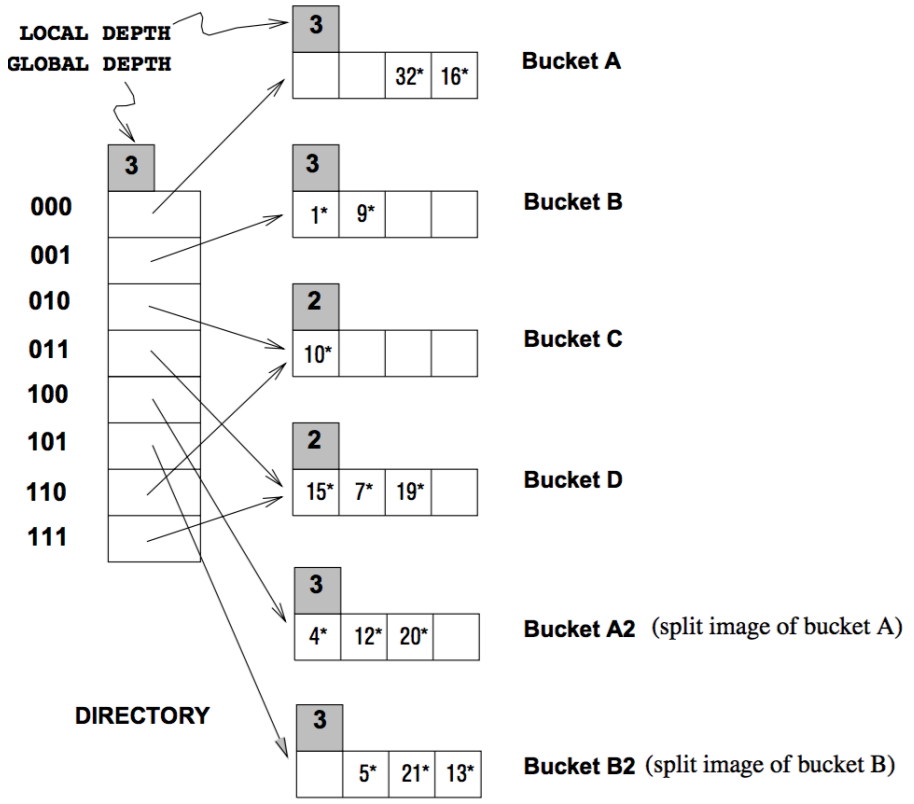


Extendible Hashing

- Insert: data entry 9* (binary 1001) in bucket B (full)
- split the bucket and use directory elements 001 and 101 to point to the bucket and its split image
- when a bucket whose **local depth = global depth** is split, the directory must be doubled
- bucket B local depth= 2 and

global depth = 3





Extendible Hashing

- initially, all local depths are equal to the global depth
- increment the global depth by 1 each time the directory doubles
- when a bucket is split (regardless if directory doubles), increment local depth of the split bucket by 1, assign same local depth to its split image
- first d bits (the most significant bits) instead of the last d (least significant bits) can also be used
- typical example: a 100 MB file with 100 bytes per data entry and a page size of 4 KB contains 1,000,000 data entries and only about 25,000 elements in the directory (each page/bucket contains roughly 40 data entries, one directory element per bucket)
- **skewed data distribution** - distribution of hash values of search field values is very 'bursty' or non-uniform
- **collisions**: data entries with the same hash value must be handled specially - we need overflow pages

Linear Hashing

- dynamic hashing technique
- no directory
- deals naturally with collisions
- offers flexibility with the timing of bucket splits (trade off slightly greater overflow chains for higher average space utilization)
- overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing for skewed data distribution

Linear Hashing

- utilizes a family of hash functions h0, h1, h2, ..., where function's range is twice that of its predecessor
- h_i maps a data entry into one of M buckets, h_{i+1} maps a data entry into one of 2M buckets
- choose a hash function *h* and an initial number *N* of buckets, and defining $h_i(value) = h(value)\ mod\ (2^iN)$
- **rounds** of splitting explains this best

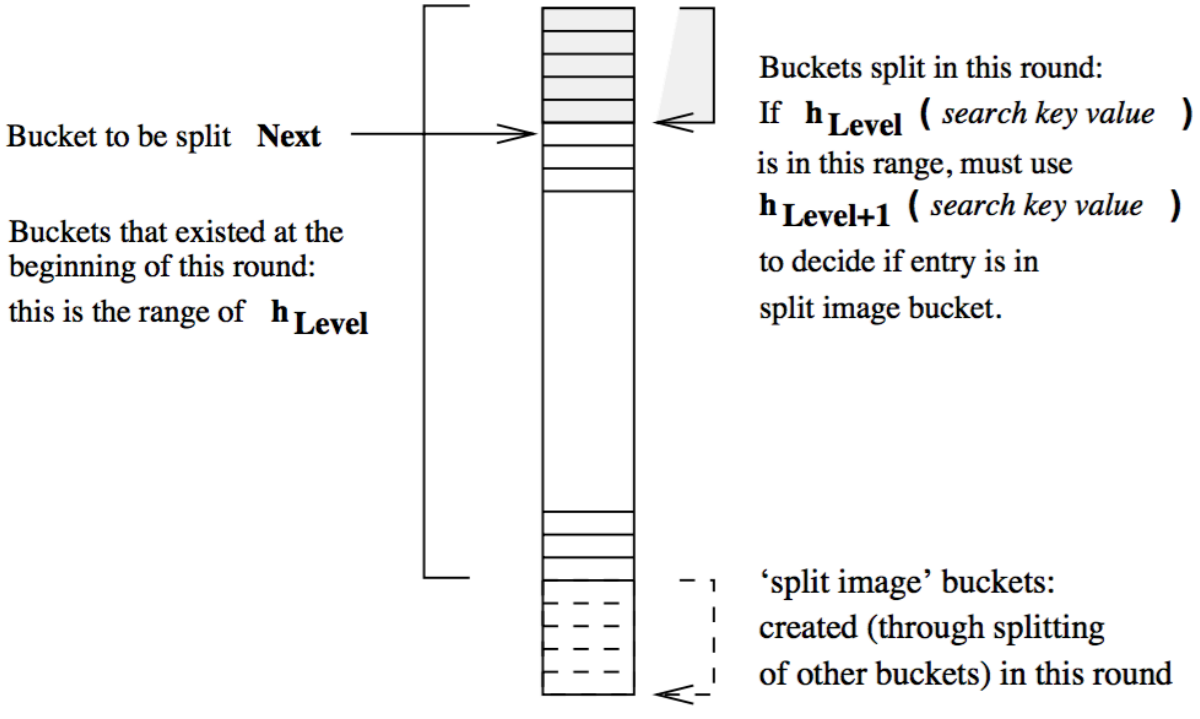
Linear Hashing

rounds of splitting explains this best

- during round number *Level*, only hash functions h_{Level} and $h_{Level+1}$ are in use
- buckets in the file at the beginning of the round are split (doubling the number of buckets)

we have:

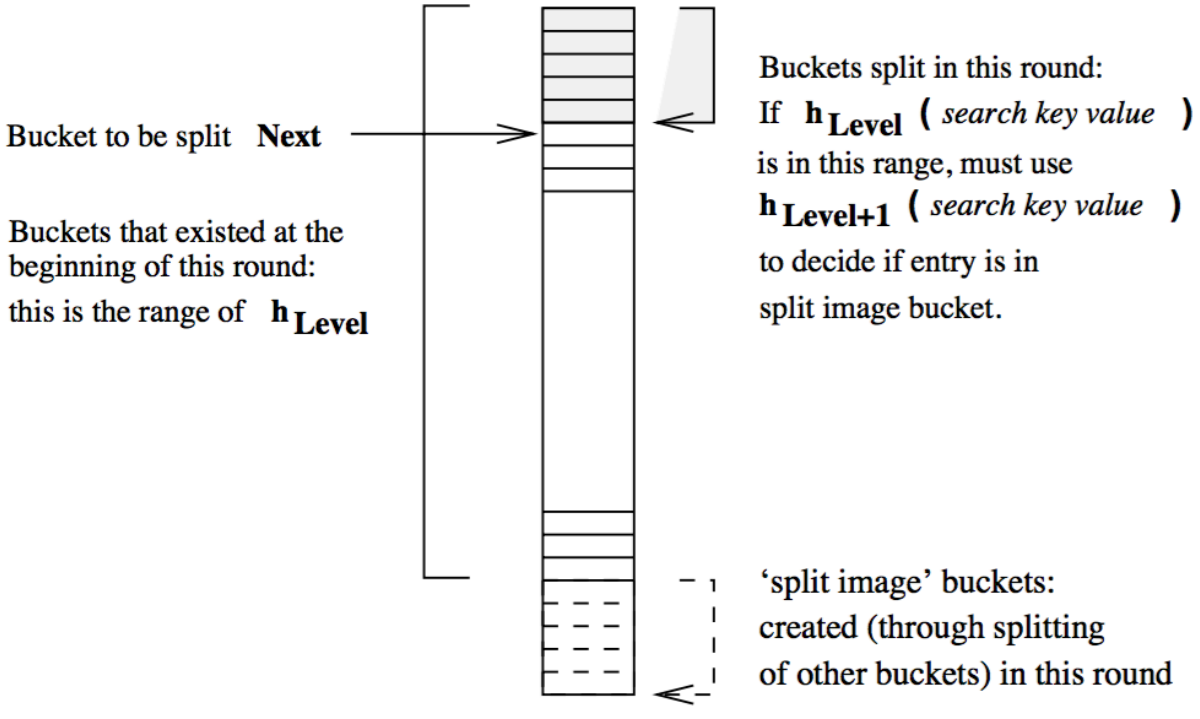
- buckets that have been split
- buckets that are yet to be split
- buckets created by splits in this round
-



Linear Hashing

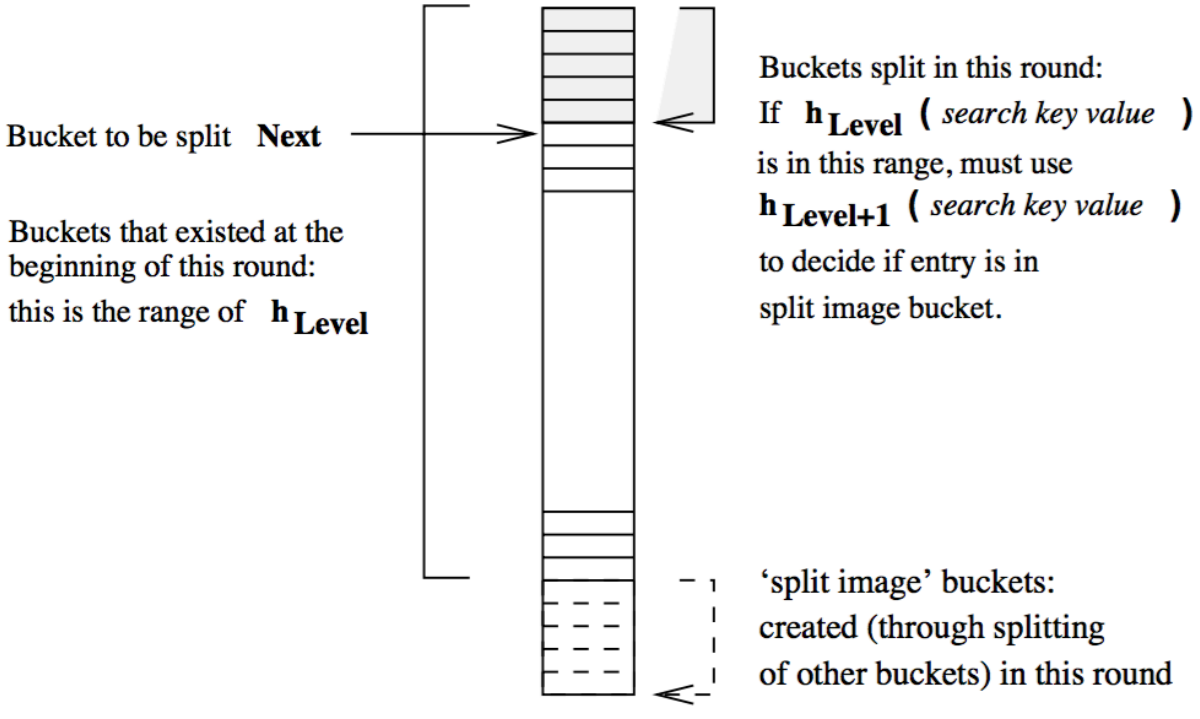
apply hash function h_{Level}

- if this leads to one of the unsplit buckets, look there
- i f it leads to one of the split buckets, the entry may be there or it may have been moved to the new bucket, apply $_{hLevel+1}$ to determine which bucket



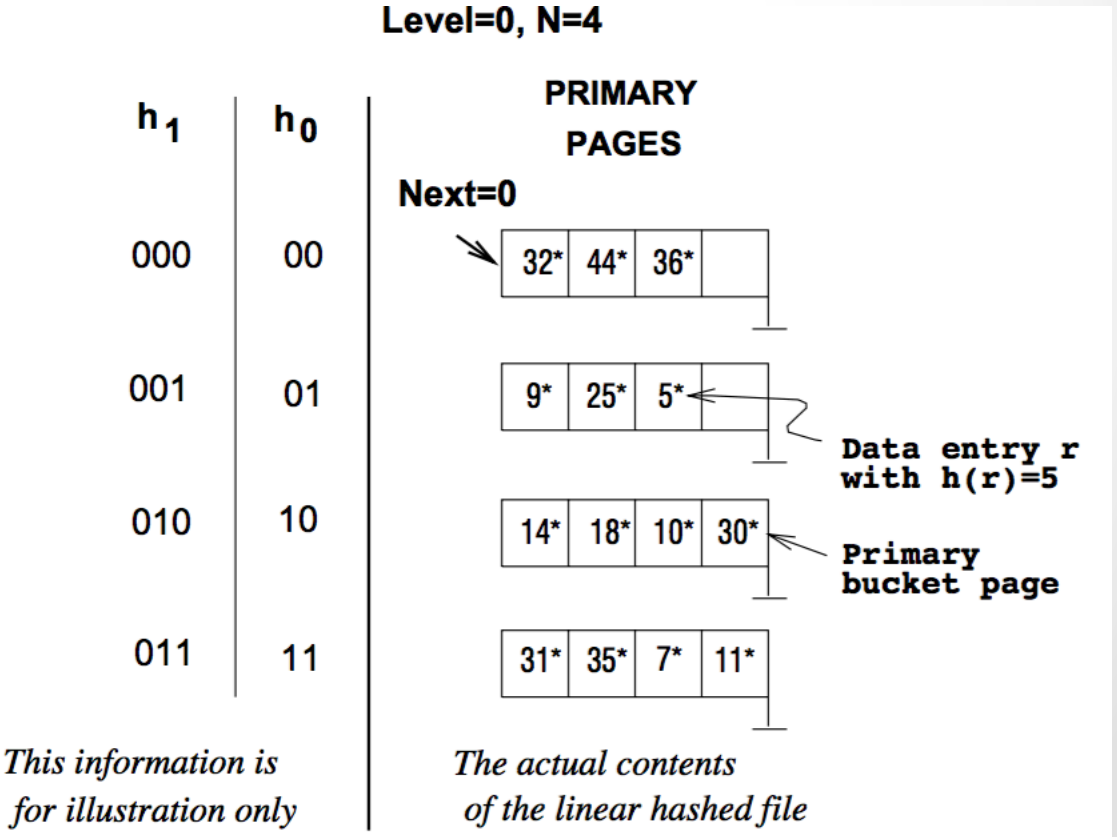
Linear Hashing

- when an insert triggers a split, the bucket into which the data entry is inserted is not necessarily the bucket that is split
- overflow page is added to store the newly inserted data entry
- bucket to split is chosen **in round-robin fashion**
- all buckets are eventually split - redistributing the data entries in overflow chains before the chains get to be more than one or two pages long



Linear Hashing

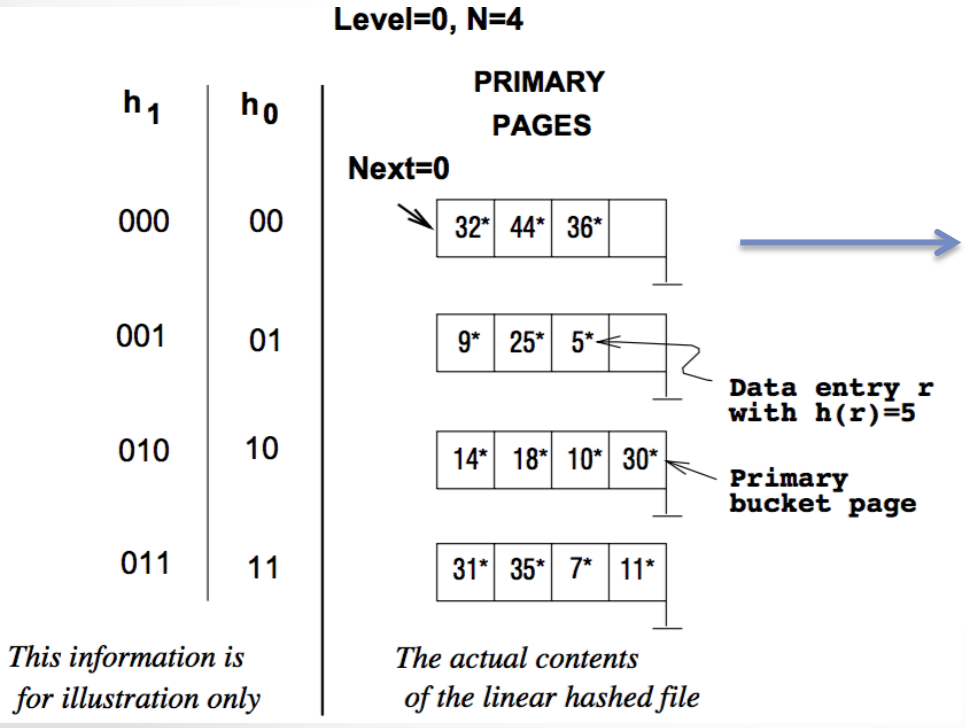
- Level is initialized to 0 (indicates the current round)
- bucket to split is denoted by Next
- number of buckets in the file by N_{Level}.
- $N_{Level} = N * 2^{Level}$

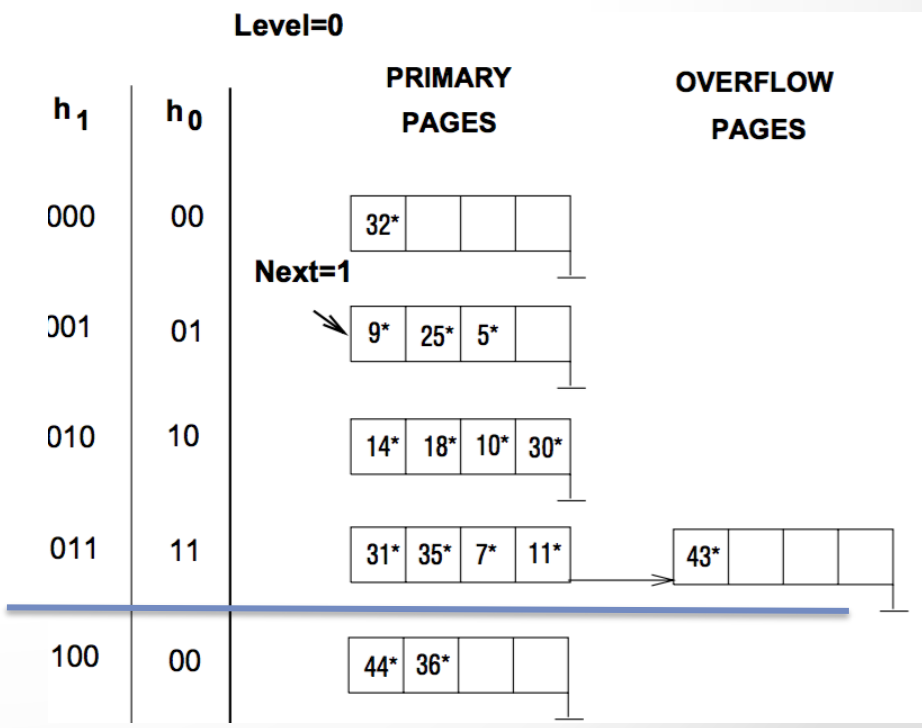


Linear Hashing

Inserting data entry 43*

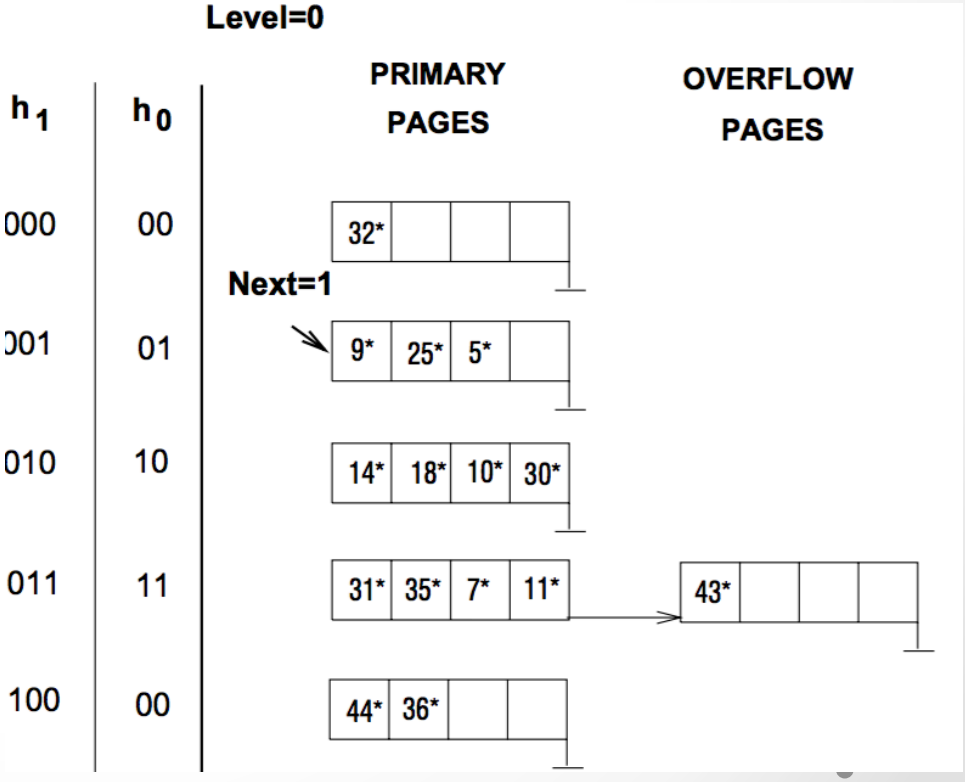
- split whenever a new overflow page is added
- when a split is triggered, *Next* bucket is split
- hash function $h_{Level+1}$ redistributes entries between this bucket & split image
- split image is bucket number b + N_{Level}
- Next is incremented by 1





Linear Hashing

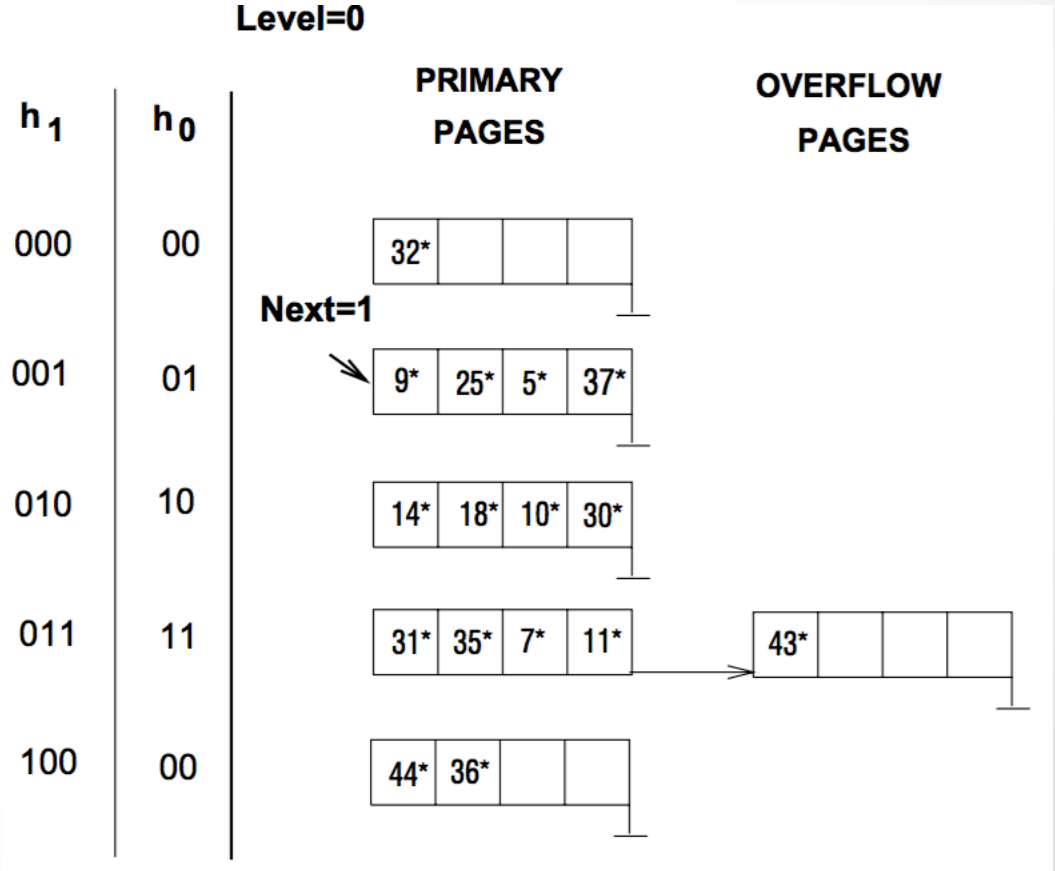
- h0(32) and h0(44) are both 0 (binary 00)
- Next is currently equal to 1 (a bucket that has been split)
- we apply h_1
- $h_1(32) = 0$ (binary 000)
- $h_1(44) = 4$ (binary 100)
- 32 belongs in bucket A
- 44 belongs in its split image, bucket A2



Linear Hashing

Inserting data entry 37*

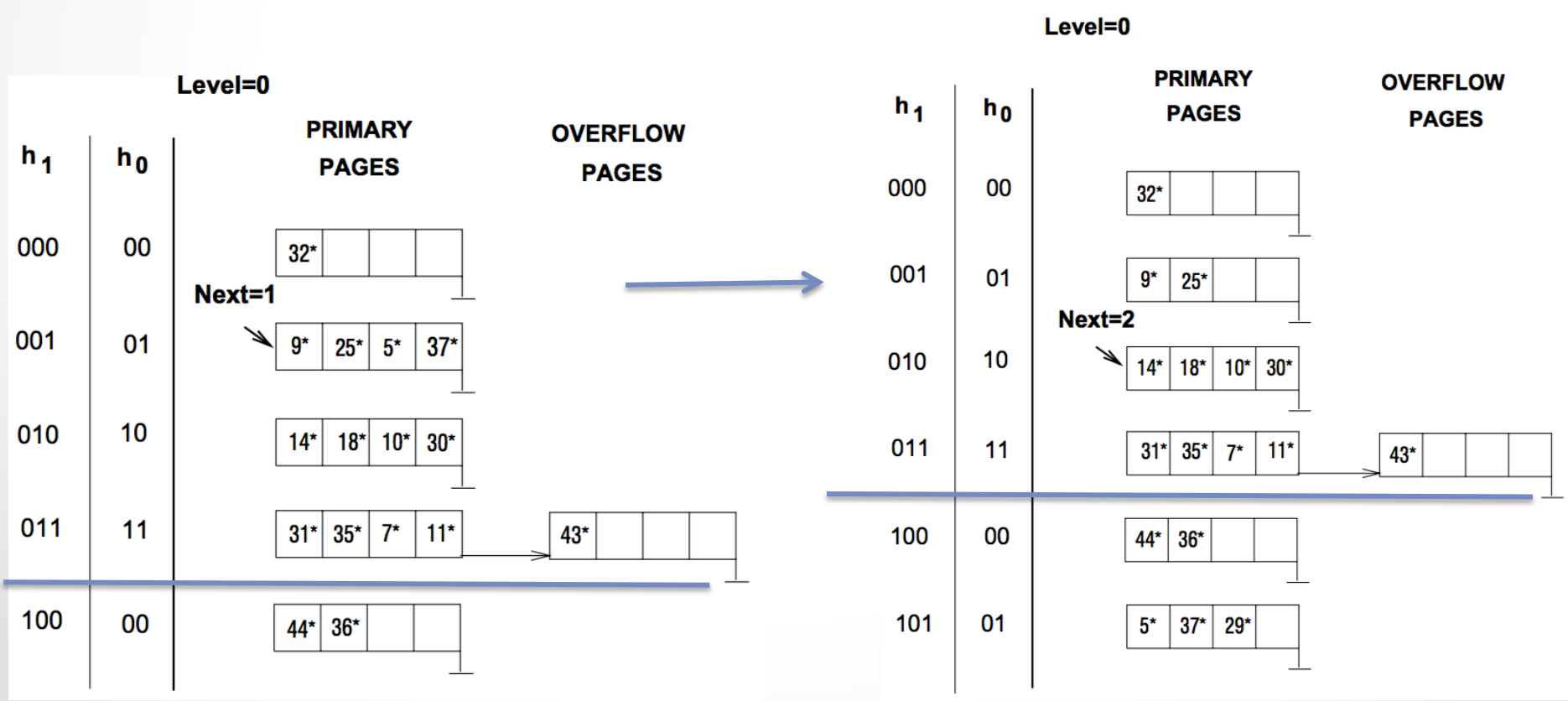
- no split
- appropriate bucket has space for the new data entry



Linear Hashing

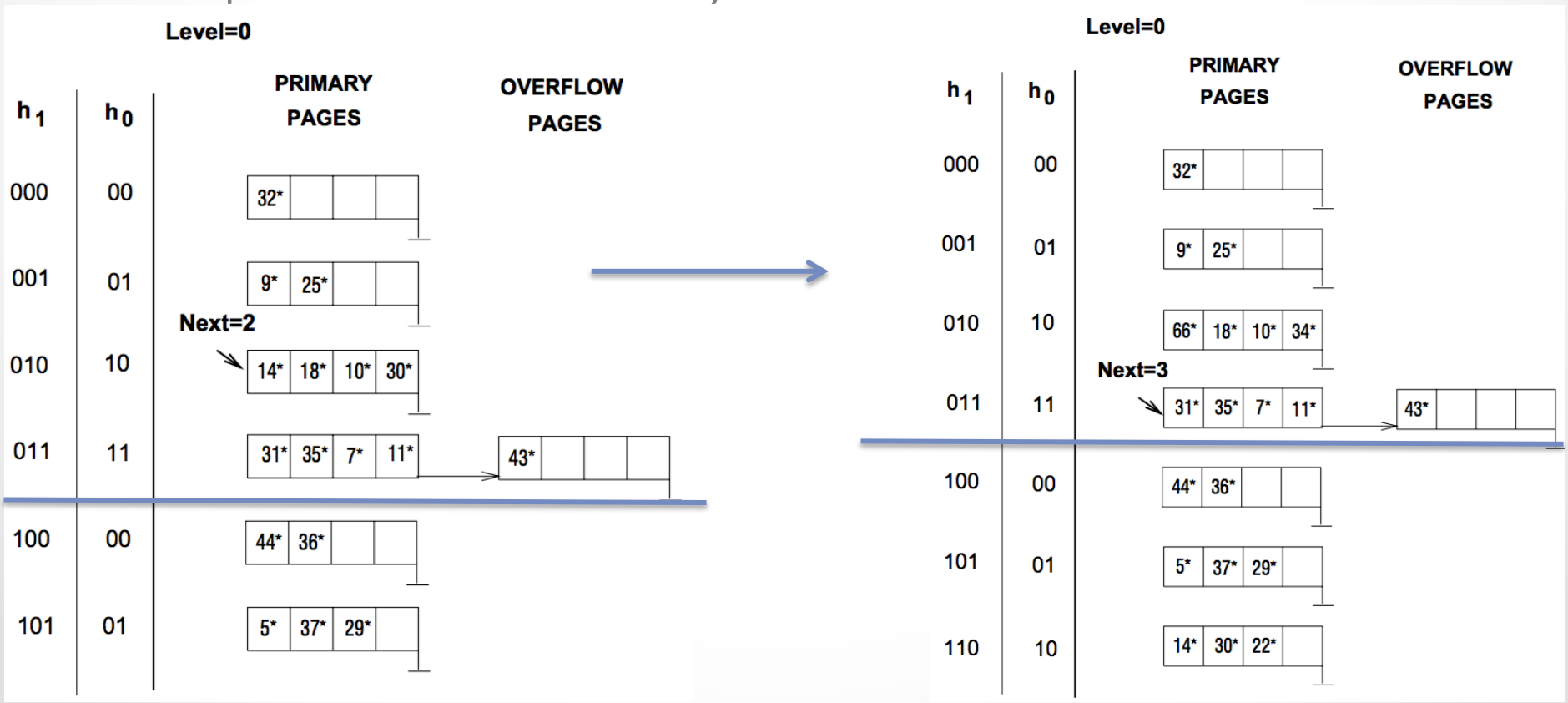
Inserting data entry 29*

- bucket pointed to by Next is full
- split is triggered but no need for a new overflow bucket
- entries are redistributed between 001 and 101



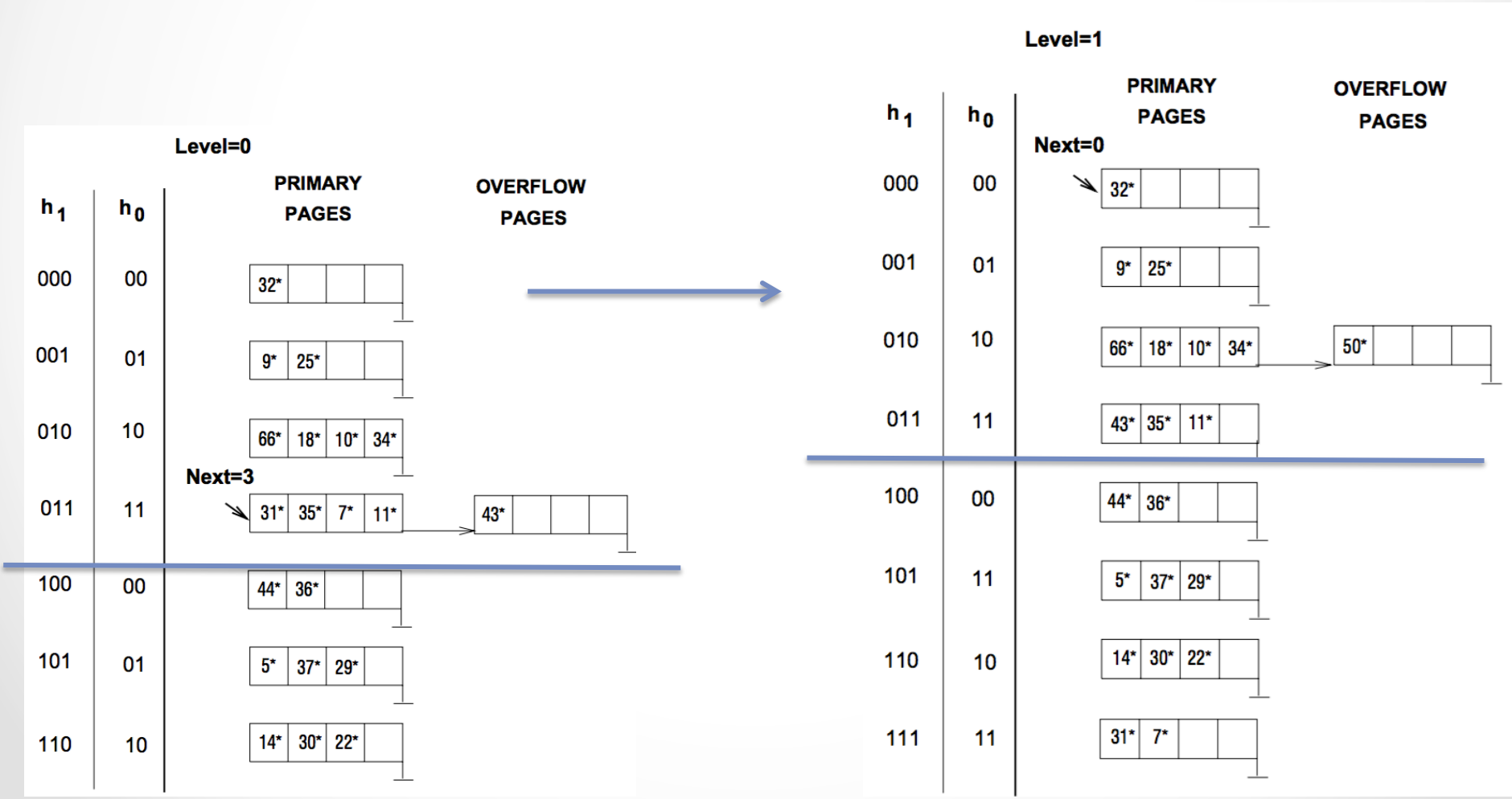
Linear Hashing

- Inserting records with h(r)=22, 66, and 34
- 22* - no overflow, are redistributed between 010 and 110
- 66* - space, insert directly into 010
- 34* - space, insert directly into 010



Linear Hashing

- insert data entry 50*
- 50* causes a split that leads to incrementing *Level*



Linear Hashing

Deletion:

- inverse of insertion
- if last bucket in the file is empty
  - remove
  - Next is decremented.
- if Next is 0 and the last bucket becomes empty,
  - Next is made to point to bucket (M/2) – 1 (M is the current number of buckets)
  - Level is decremented
  - empty bucket is removed
- we can combine the last bucket with its split image even when it is not empty (merge)
- merge is based on:
  - file occupancy of the file
  - done to improve space utilization

FAU

SUMMARY

- Hash-based indexes are designed for equality queries
- A **hashing function** is applied to a search field value and returns a **bucket** number
- bucket number corresponds to a page on disk
- **Static Hashing** index has a fixed number of primary buckets
- **Extendible Hashing** is a dynamic index structure introduces a level of indirection in the form of a directory
- **Linear Hashing** avoids a directory by splitting the buckets in a round-robin fashion
- Insertions can trigger bucket splits, but buckets are split sequentially in order