Hash-Based Indexes

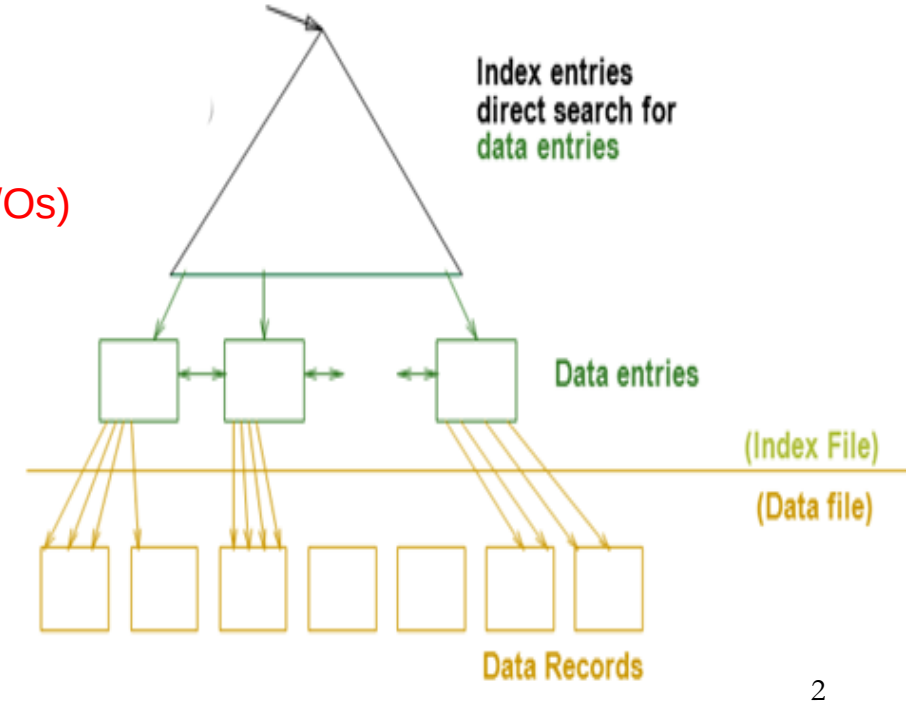
SUN YAT-SEN UNIVERSITY

Review

- 索引技术概述
  - 可以为关系建立索引，都是文件
  - 索引文件由两部份组成
    1. 数据项部分
       - Data Entry(数据项) ⟺ data record（数据记录）
    2. 引导部份
       - 树索引技术

         Cost = $\log_F N$　　(2~3 I/Os)
       - Hash索引

         1~2 I/Os

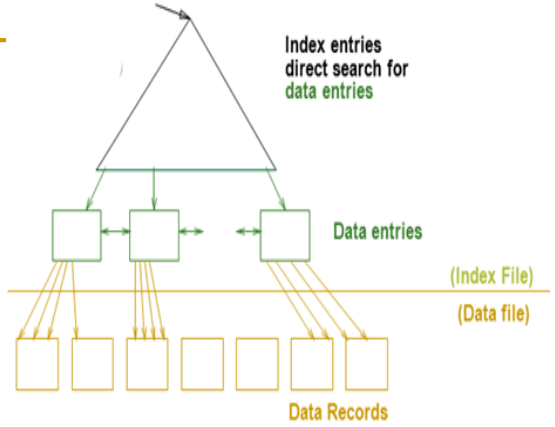


Introduction

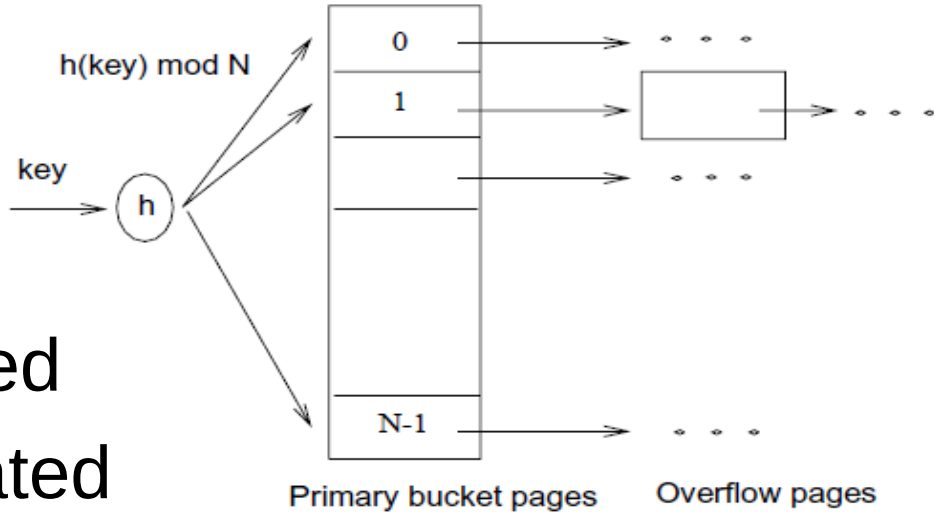
- As for any index, 3 alternatives for data entries k*:
  - Data record with key value k
  - <k, rid of data record with search key valuek>
  - <k, list of rids of data records with search key k>

- Hash-based indexes are best for equality selections. Cannot support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing(静态哈希)

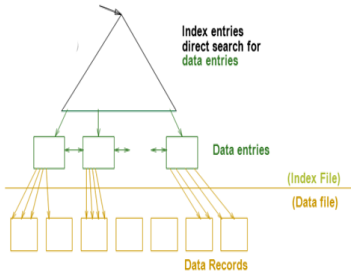


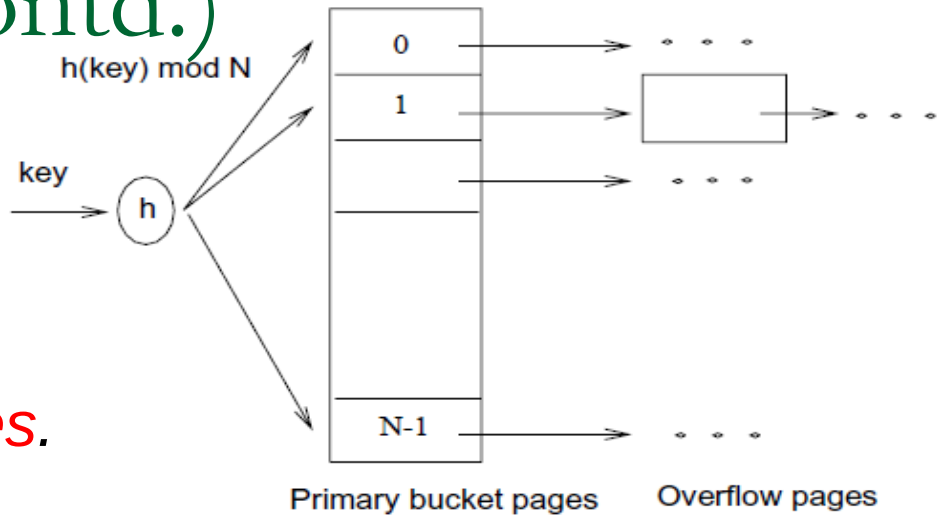
- 静态哈希索引由一系列桶(bucket)组成
  - 每个桶依次编号为0、1、…、N-1
  - 每个桶有一个主页(primary page),也可能有一些溢出页(overflow pages)



- 静态性
  - Number of buckets is fixed
  - Primary pages are allocated sequentially, never deallocated;
    - overflow pages if needed.

Static Hashing (Contd.)

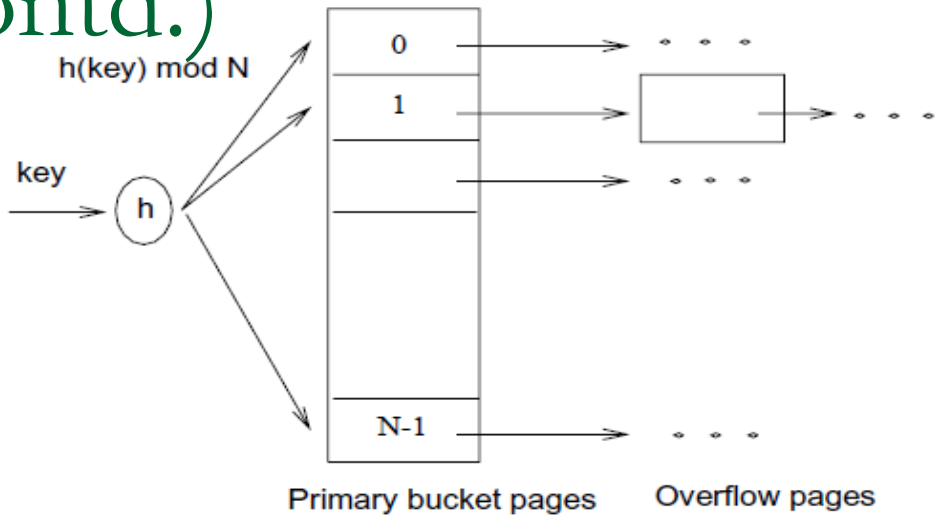




- Buckets contain *data entries*.
- 定位所属的桶?

  h(*k*) *mod N* = bucket to which data entry with key k belongs.
  - *N* = number of buckets
  - Hash function(哈希函数) 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;

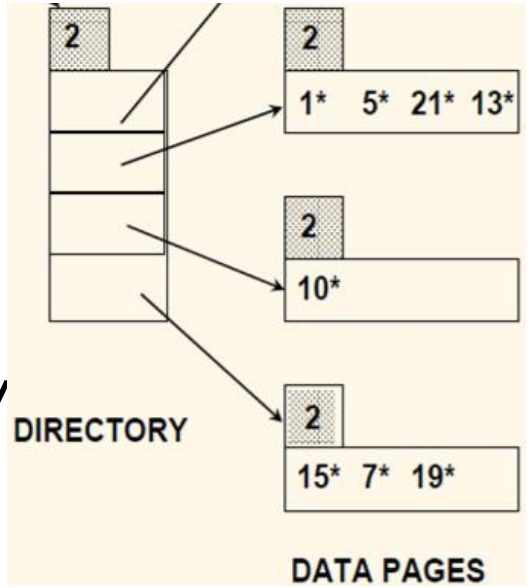
Static Hashing (Contd.)



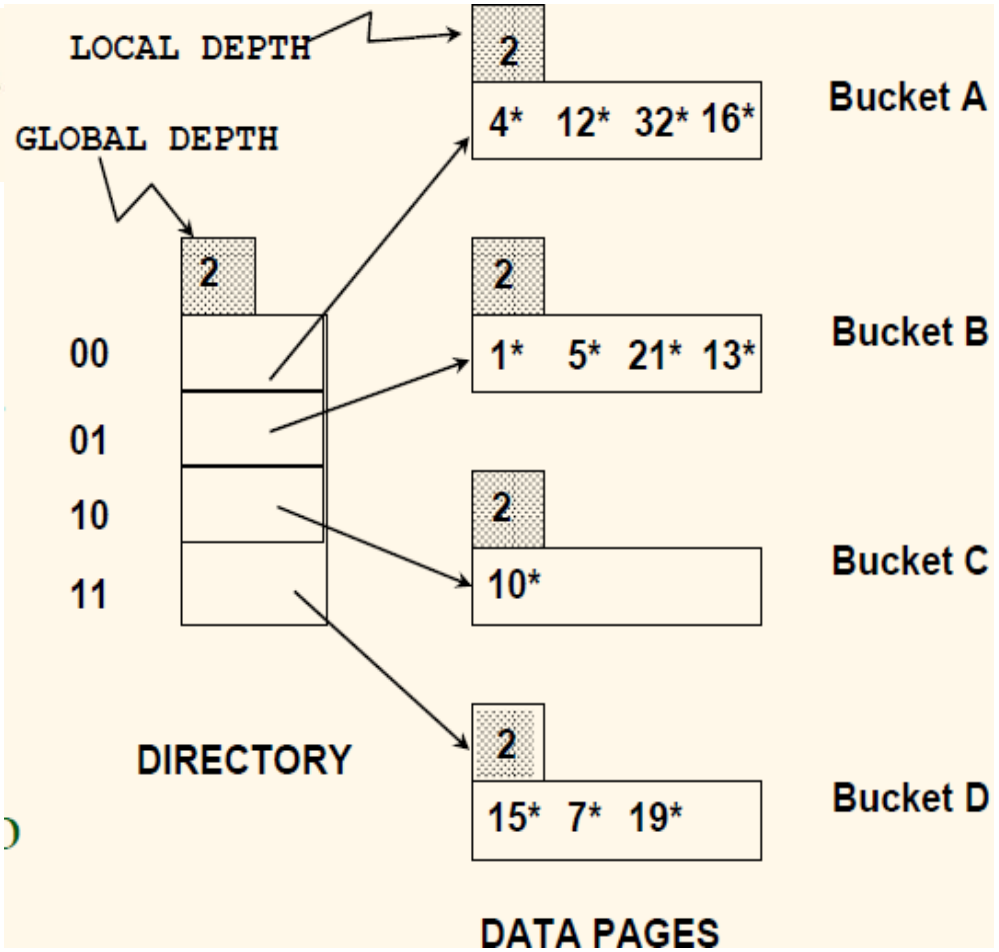
- Long overflow chains(长溢出链) can develop and degrade performance.
  - *Extendible and Linear Hashing: Dynamic techniques to fix this problem.*

Extendible Hashing(可扩展哈希)

- Situation: Bucket (primary page) becomes full.
  - Why not re-organize file by *doubling the number of buckets?* Reading and writing all pages is expensive!
- *Idea of Extendible Hashing:*
  - *Use directory of pointers to buckets(桶指针目录)*
    - *double the number of buckets by doubling the directory*
    - Directory is much smaller than file, so doubling it is much cheaper.
  - *splitting just the bucket that overflow*
    - *No overflow page!*



Example



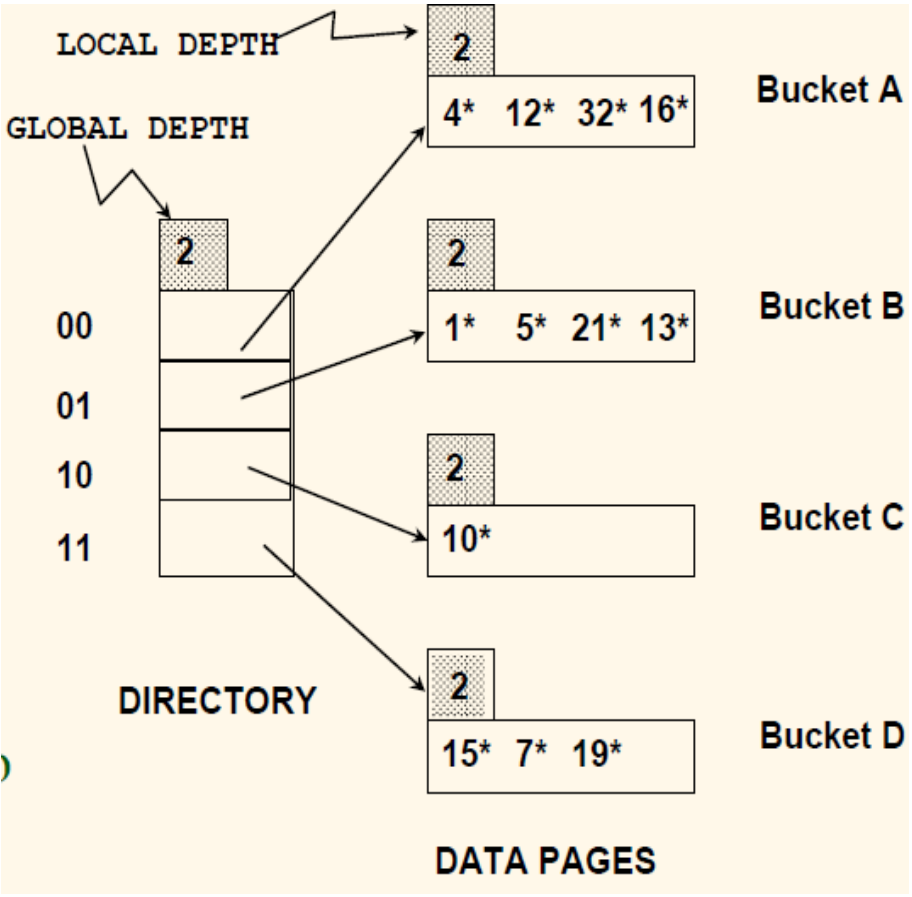
- Directory is array of size 4.
- To find bucket for r，take last 'global depth'（全局深度） #bits of h(r)； we denote r by h(r) .
  - If h(r) = 5 = binary 101，it is in bucket pointed to by 01.

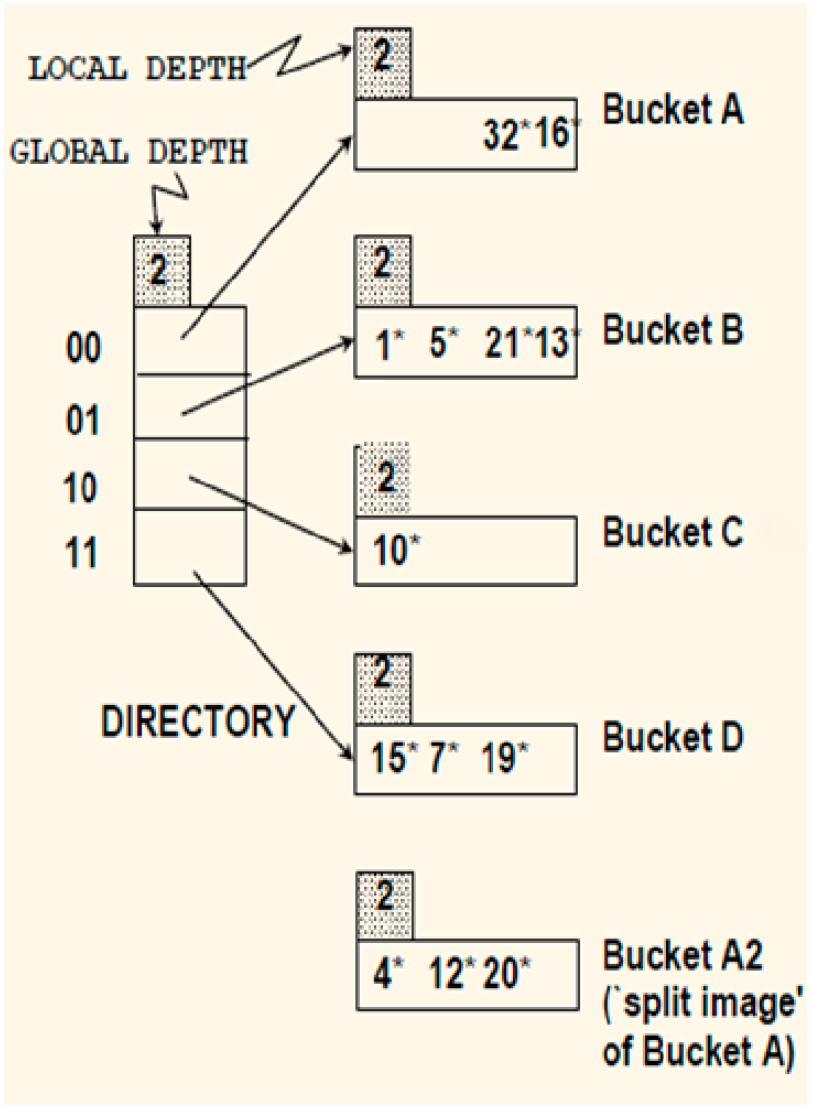
- Insert：If bucket is full， split it(allocate new page， re-distribute) .
  - If necessary， double the directory.
  - As will see， splitting a bucket does not always require doubling
    - we can tell by comparing global depth with local depth for the split bucket.

Insert h(r)=20(Causes Doubling)

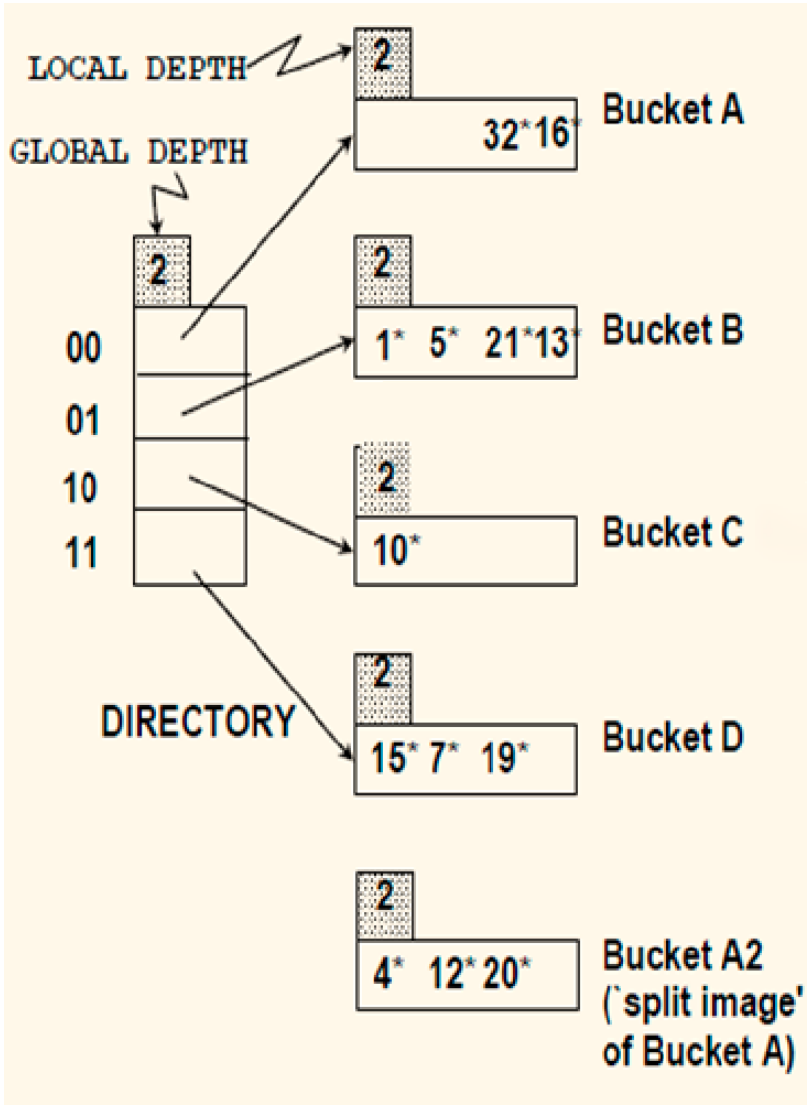
10100

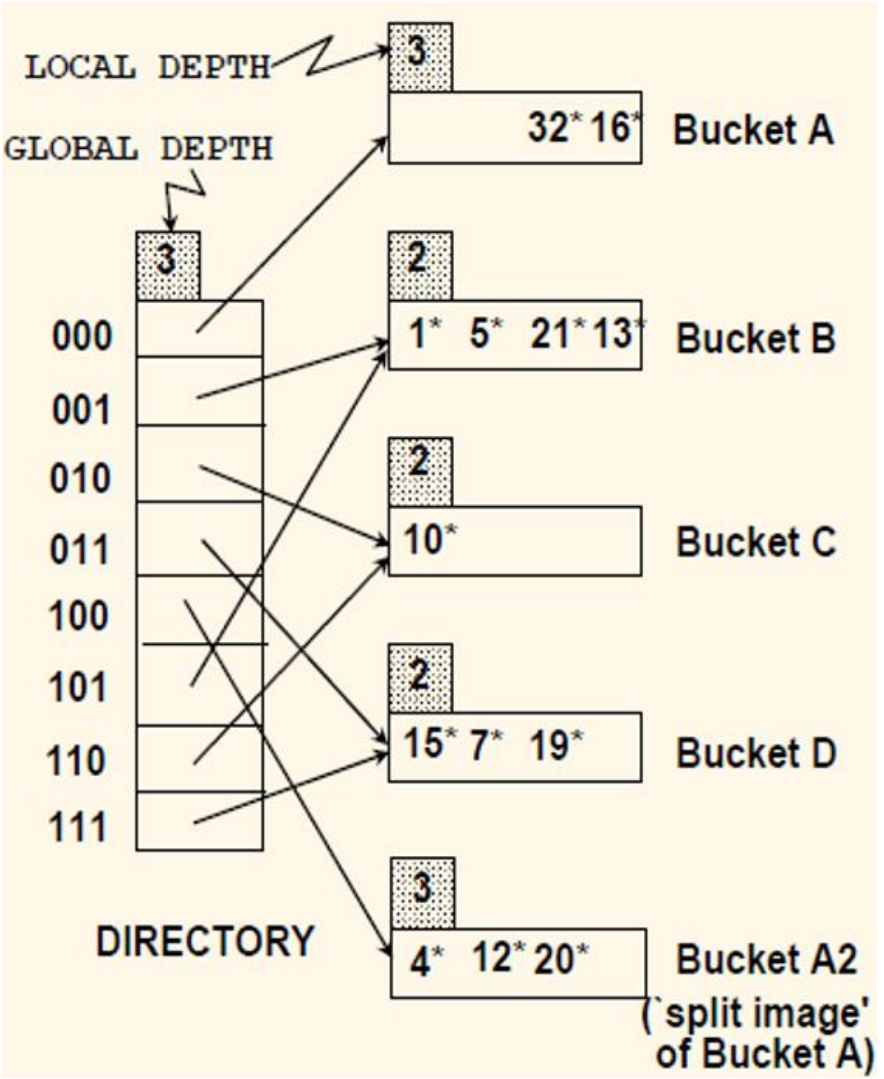
If bucket is full, split it
(allocate new page, re-distribute)





Insert h(r)=20 (Causes Doubling) 10100

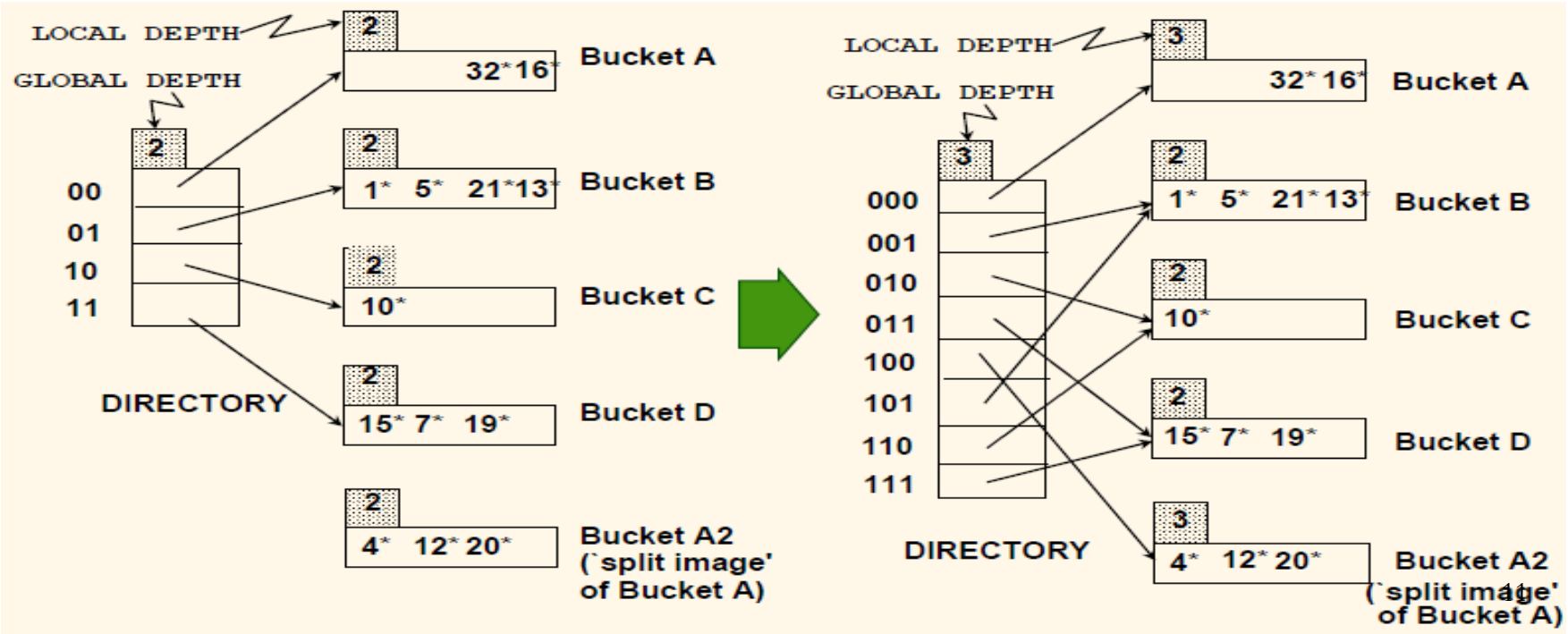




目录加倍→（1、空间；2、复制；3、映像指针） → 全局深度++→ 局部深度

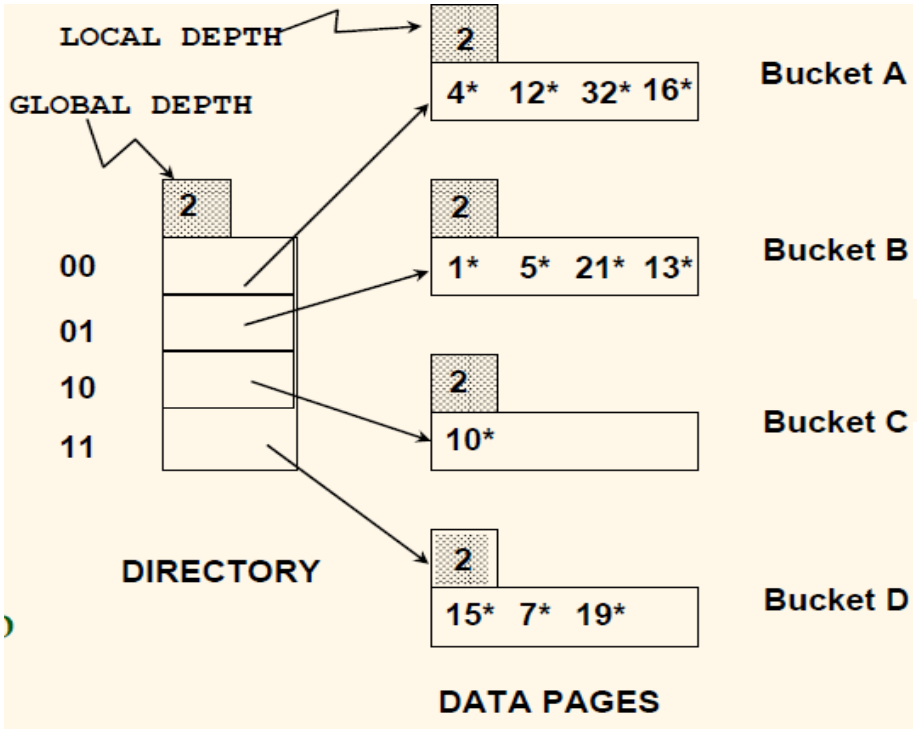
Points to Note(Contd.)

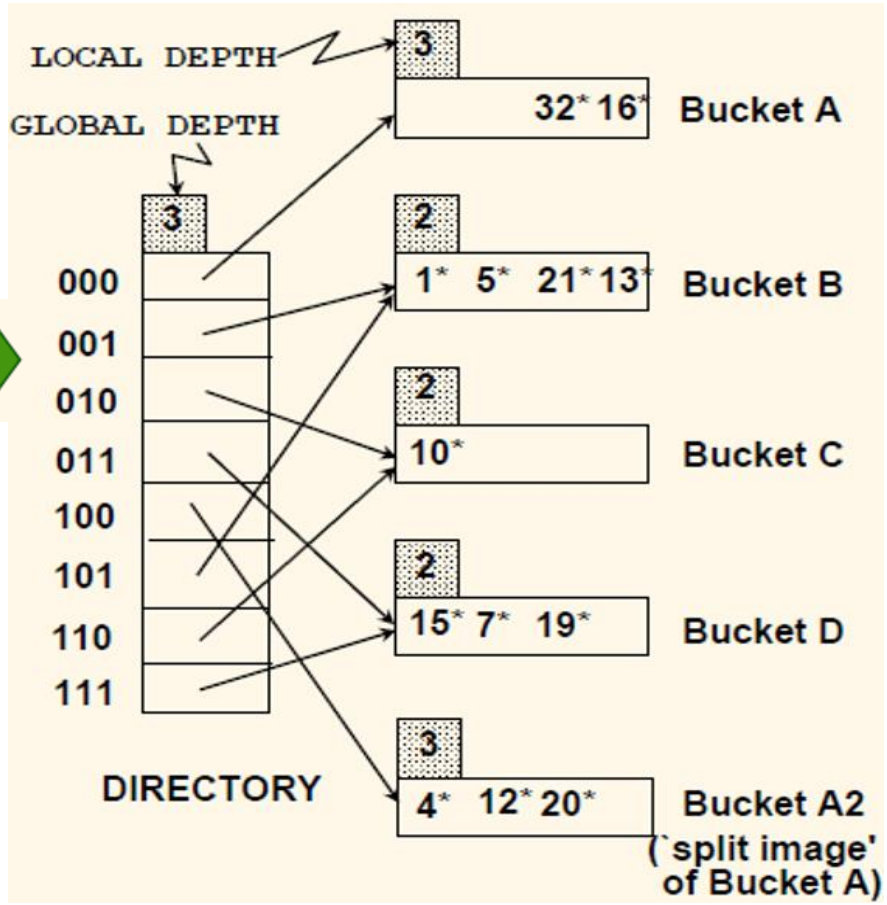
- 20 = binary 10100.
  - Last 2 bits (00) tell us r belongs in A or A2.
  - Last 3 bits needed to tell which.
- *Global depth of directory（目录的全局深度）: Max number of bits needed to tell which bucket an entry belongs to.*
- *Local depth of a bucket（桶的局部深度）: number of bits used to determine if an entry belongs to this bucket.*



Points to Note

- When does bucket split cause directory doubling?
  - Before insert, *local depth of bucket = global depth. Insert causes local depth to become > global depth.*





低位扩展哈希与高位扩展哈希

将哈希值映射到桶号的两种方案

低位方案： 低几位

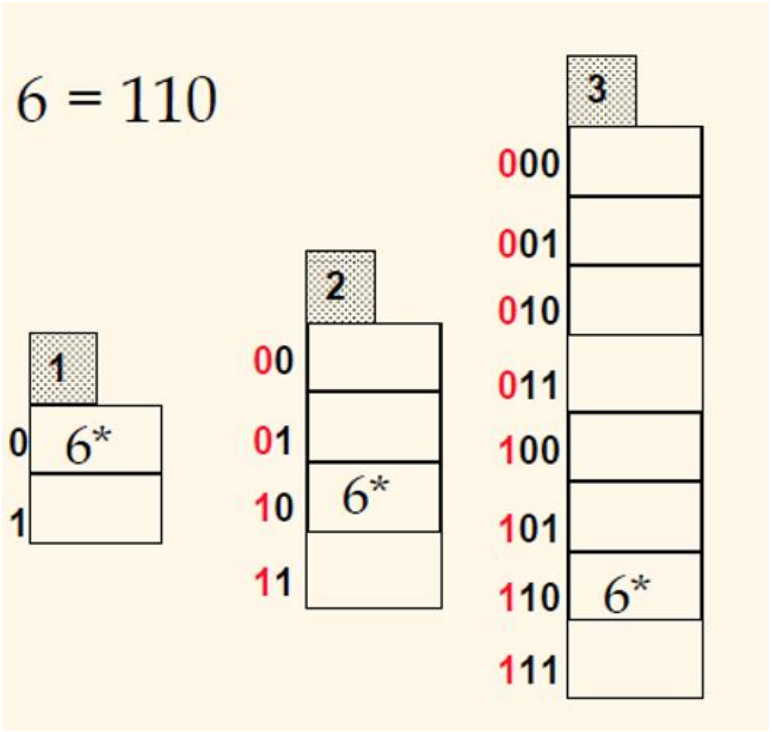
低位 → 高位

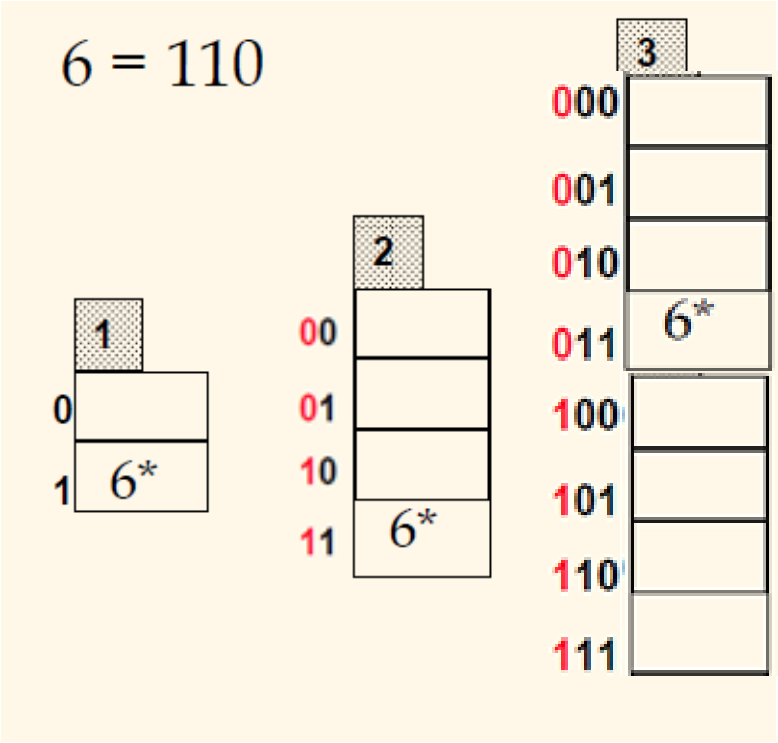
高位方案： 高几位（约定总位数）

高位 → 低位（反转？）

Why use least significant bits in directory?

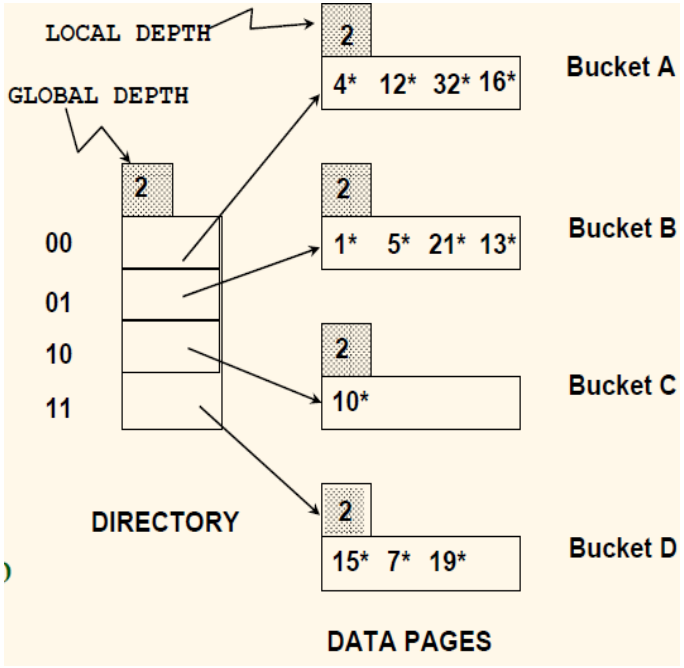
⇔ Allows for doubling via copying!





Least Significant VS Most Significant

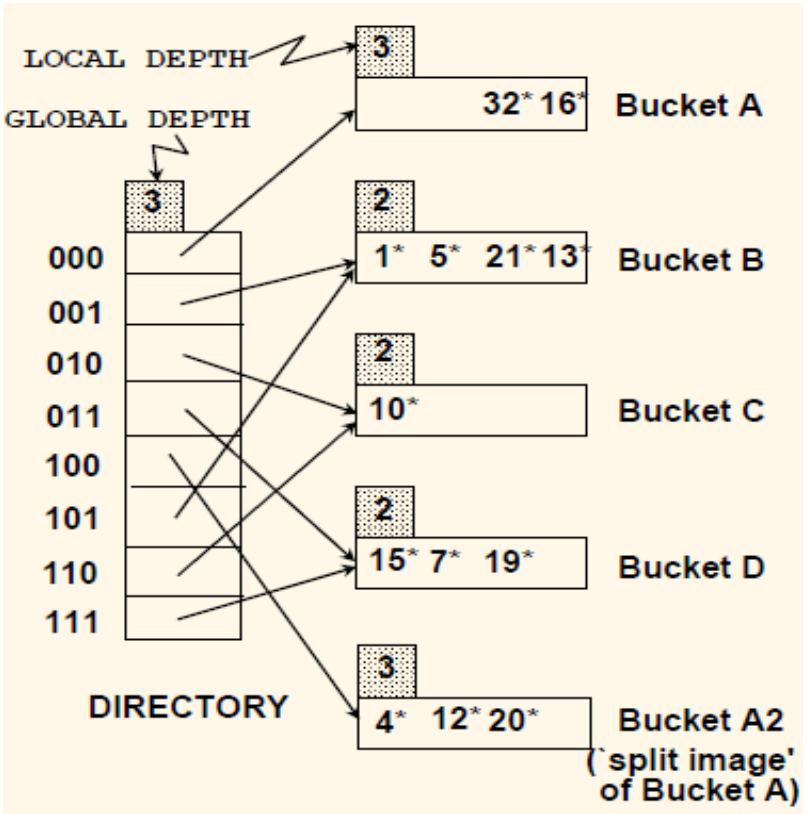
Equality Search in Extendible Hashing

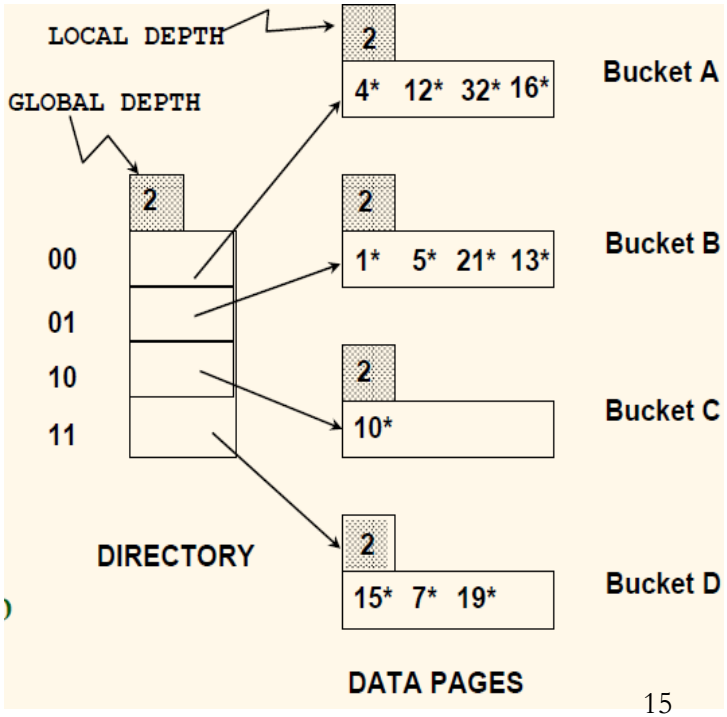


- If directory fits in memory, equality search answered with one disk access; else two.
- chances are high that directory will fit in memory.
- 如: 对于 100MB file
  - 若 100 bytes/rec, 则 contains 1,000,000 records (as data entries)
  - 若 4K/page,则 4K/100 =40数据项/桶,
    从而 1,000,000/40 = 25,000 （桶，即directory elements）
  - 若 10 bytes/目录项, 则 25,000*10/1000 ≈ 250KB

Delete in Extendible Hashing

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





Linear Hashing (LH)-线性哈希

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.

- LH handles the problem of long overflow chains without using a directory by using overflow pages.

The Idea of Linear Hashing

- Use a family of hash functions h_0, h_1, h_2, ...

- $\mathbf{h_i(}$*key*$\mathbf{) =}$ *h*$\mathbf{(}$*key*$\mathbf{)\ mod\ (2^iN)}$
  - ***h** is some hash function (range is **not** 0 to N-1)*
  - ***N = initial # buckets,通常, N = 2^{d0}, for some d0***
  - ***h_i consists of applying **h** and looking at **the last di bits**, where **di = d0 + i**.***
    - h_{i+1} doubles the range of h_i (similar to directory doubling)

The Idea of Linear Hashing (Contd.)

- 桶分裂方式: choosing bucket to split round-robin (循环分裂) – 双重循环

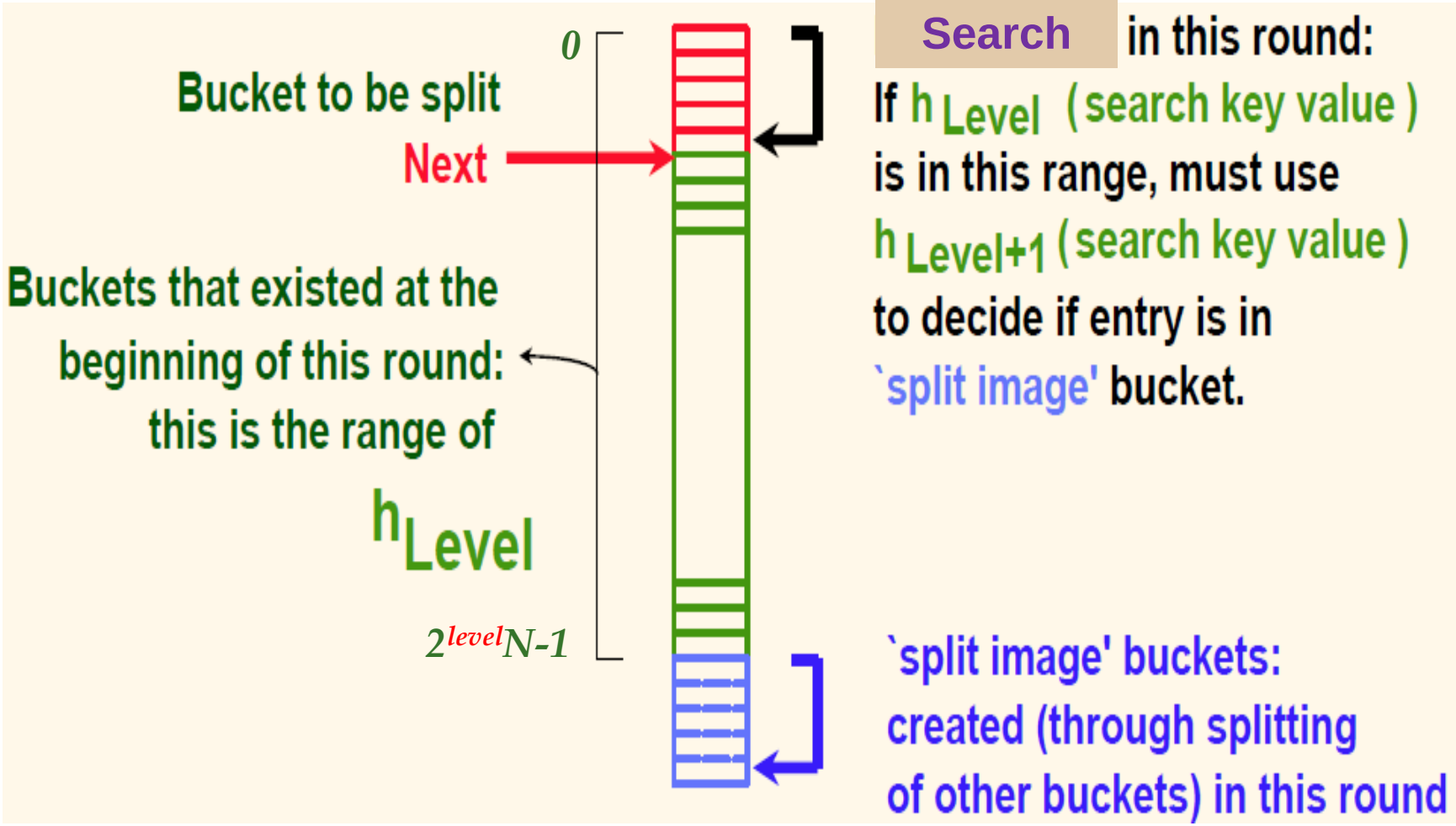
  $h_0, h_1, h_2, \ldots,$

  - **外循环:** 循环分裂级别逐渐递增: 0,1,2,…
    - Current round number is *Level*. (初值为0)
  - **内循环:** 进行循环分裂。在第*Level*级,
    - 初始桶数: $2^{level}N$
    - 桶0,1,2,…,$2^{level}N-1$ 逐个分裂
      - *Next*指向将要分裂的桶
      - Buckets 0 to *Next-1* have been split;
      - *Next* to $2^{level}N-1$ yet to be split.
    - 循环结束后的桶数: $2^{level+1}N$。则进入第 *Level+1* 级循环

Overview of LH File:

$h_0, h_1, h_2, \ldots,$

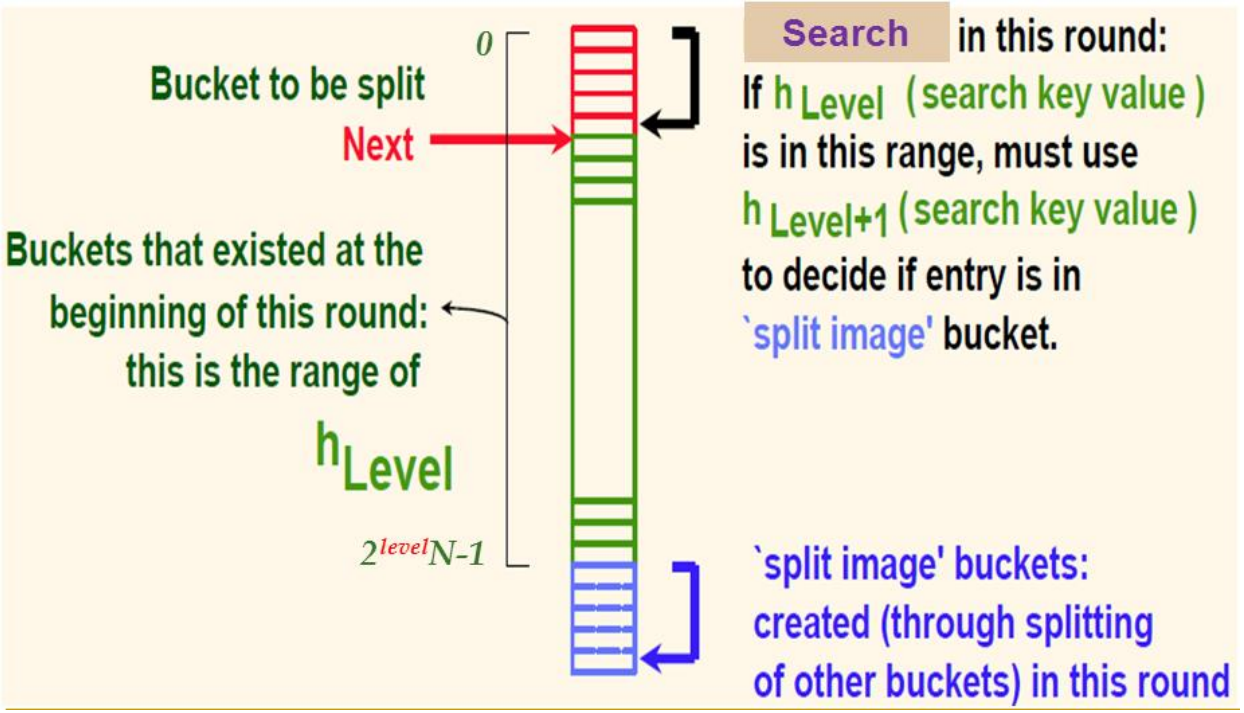
in the Middle of the *Level* –th Round



Search in Linear Hashing

$h_0, h_1, h_2, \ldots,$

- To find bucket for data entry *r*, *find* $\boldsymbol{h_{Level}(r)}$:
  - If $h_{Level}(r)$ in range `*Next to* N_R', *r* belongs here.
  - Else, *r* could belong to bucket $h_{Level}(r)$ or bucket $h_{Level}(r) + N_R$; must apply $h_{Level+1}(r)$ to find out.

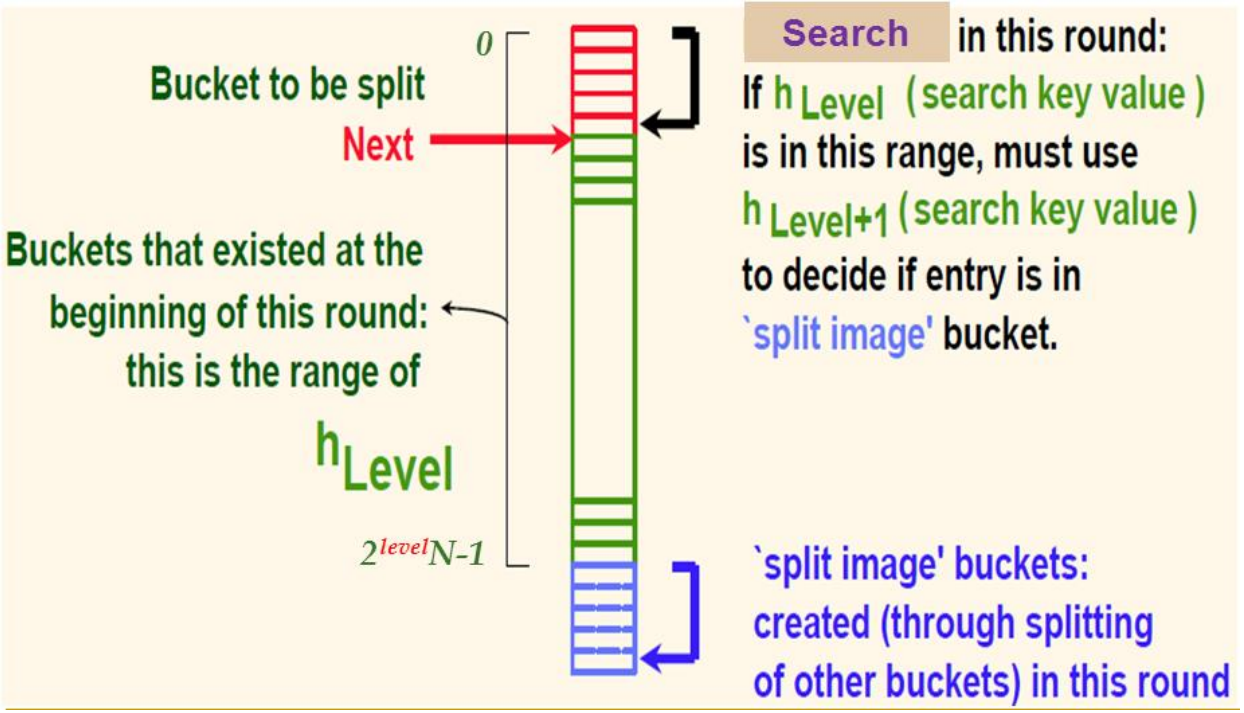


Inserting a Data Entry in LH

- Find bucket by applying $h_{Level} / h_{Level+1}$:

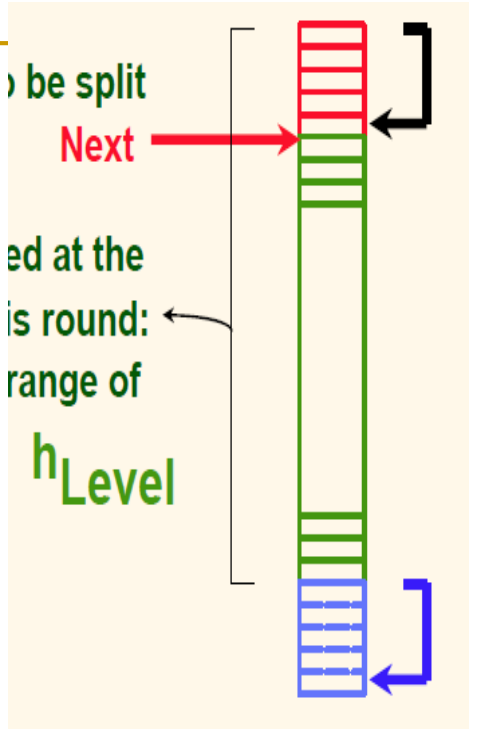
  $h_0, h_1, h_2, \ldots,$

  - If the bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (Maybe) split *Next* bucket and increment *Next*.
  - Else simply insert the data entry into the bucket.



Bucket Split

$h_0, h_1, h_2, \ldots,$



- A split can be triggered by
  - the addition of a new overflow page
  - conditions such as space utilization
- Whenever a split is triggered,
  - the *Next* bucket is split,
    - and hash function $h_{Level+1}$ redistributes entries between this bucket (say bucket number b) and its split image(bucket number $b+N_{Level}$)
  - *Next* ← *Next* + 1.

Example of Linear Hashing

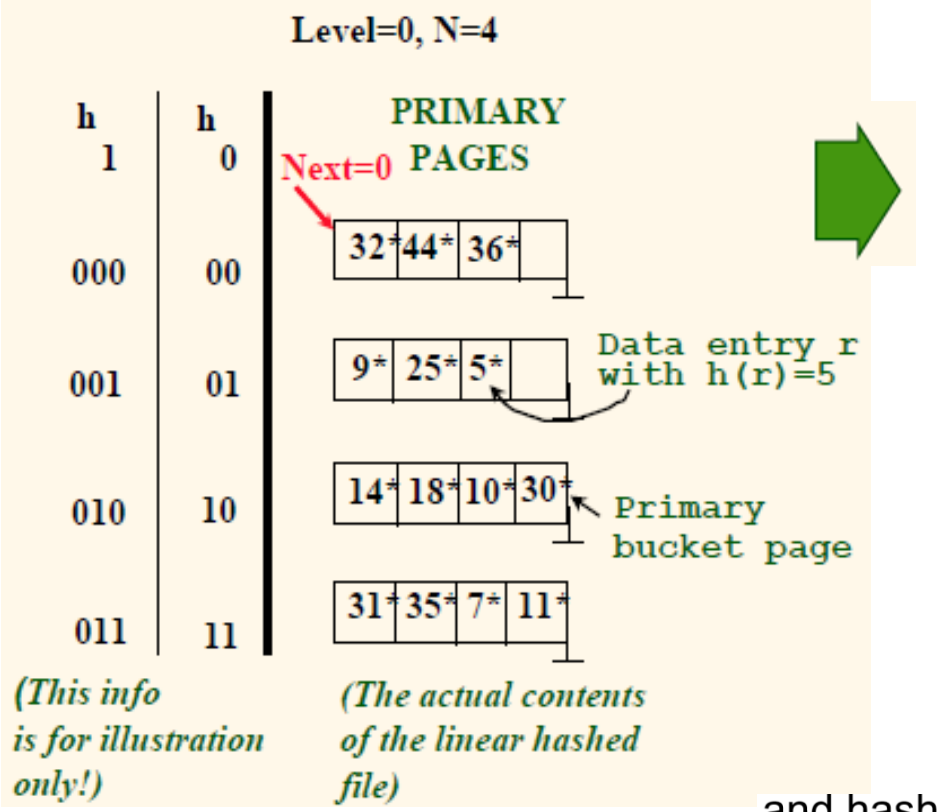
$h_0, h_1, h_2, \ldots,$

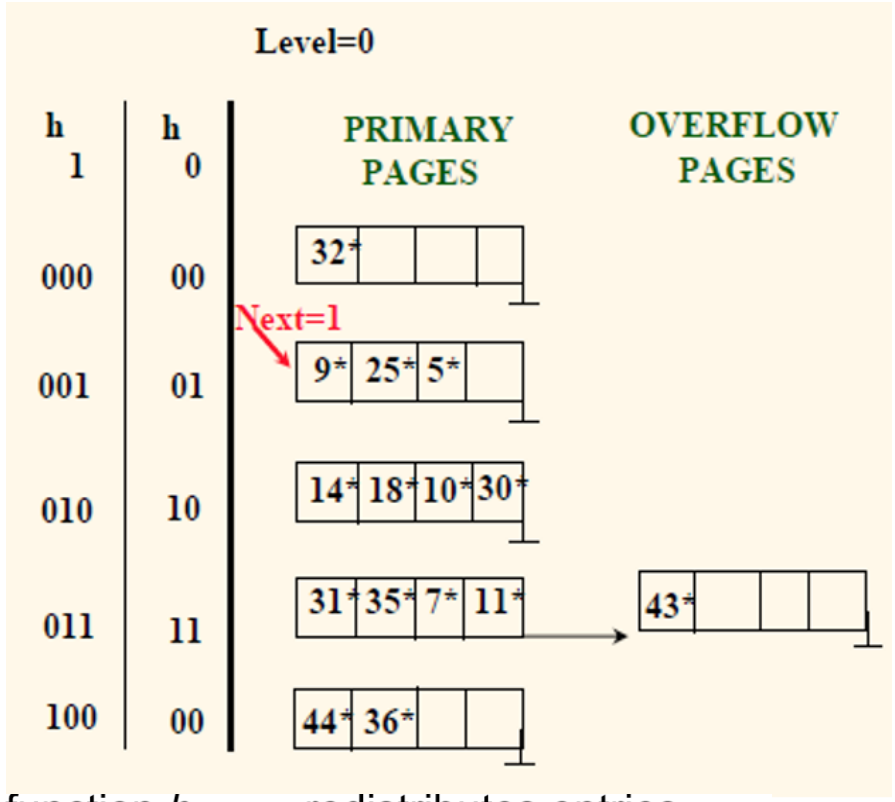
$\mathbf{h_i}(key) = h(key) \bmod (2^i N)$

- On split, $\mathbf{h}_{Level+1}$ is used to re-distribute entries.

Insert data entry of 43*

101011

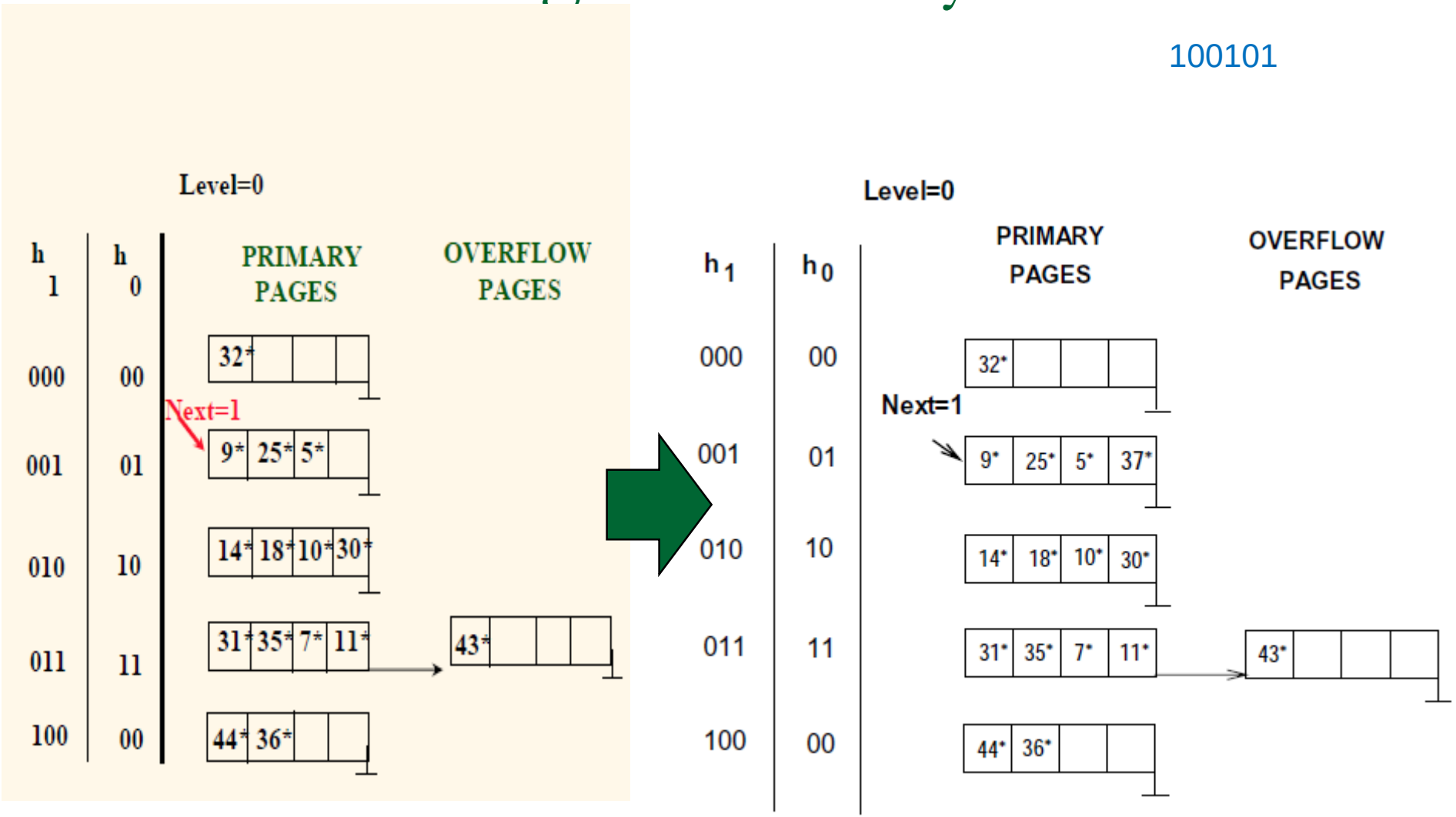




and hash function $h_{Level+1}$ redistributes entries between this bucket (say bucket number b) and its split image(bucket number $b+N_{Level}$)

After Inserting Data Entry of 37*

100101

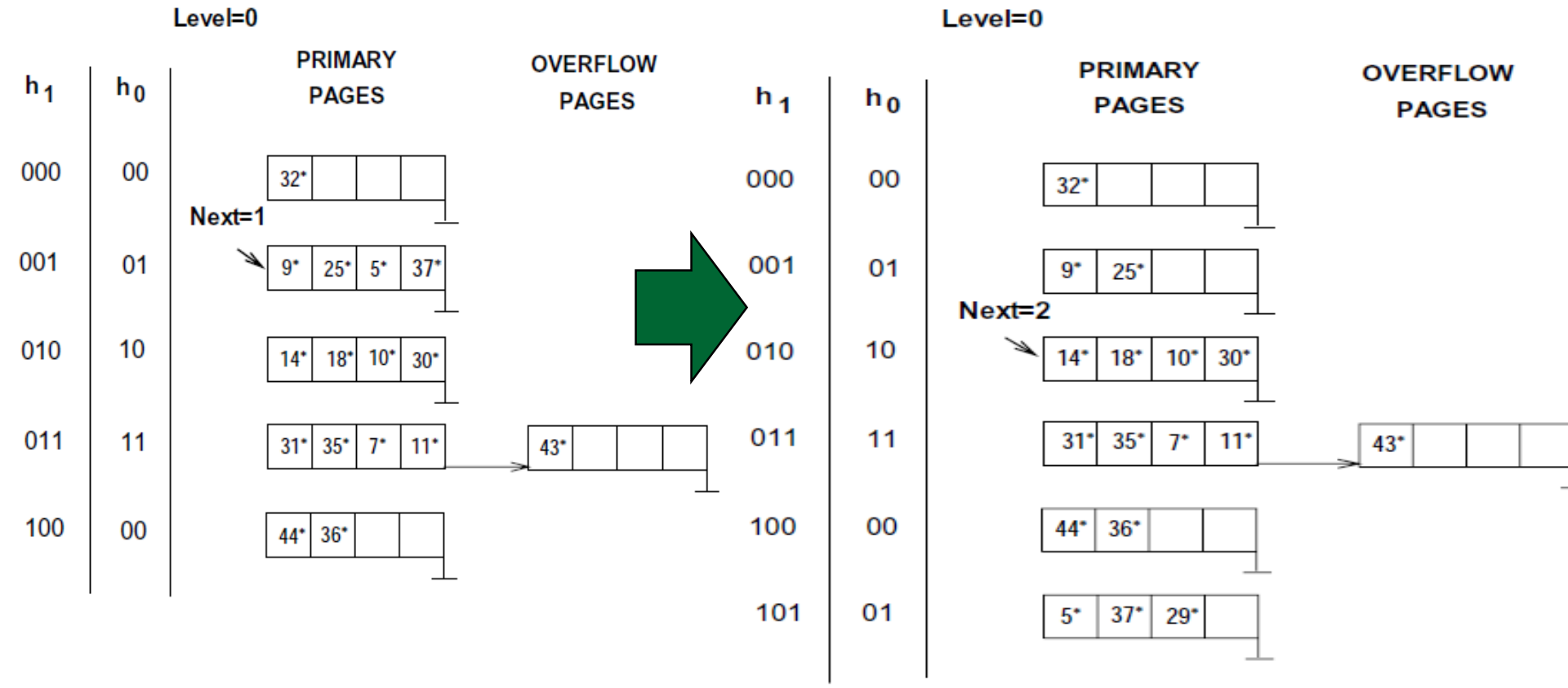


$h_0, h_1, h_2, \ldots,$

After Inserting Data Entry of 29*

11101

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

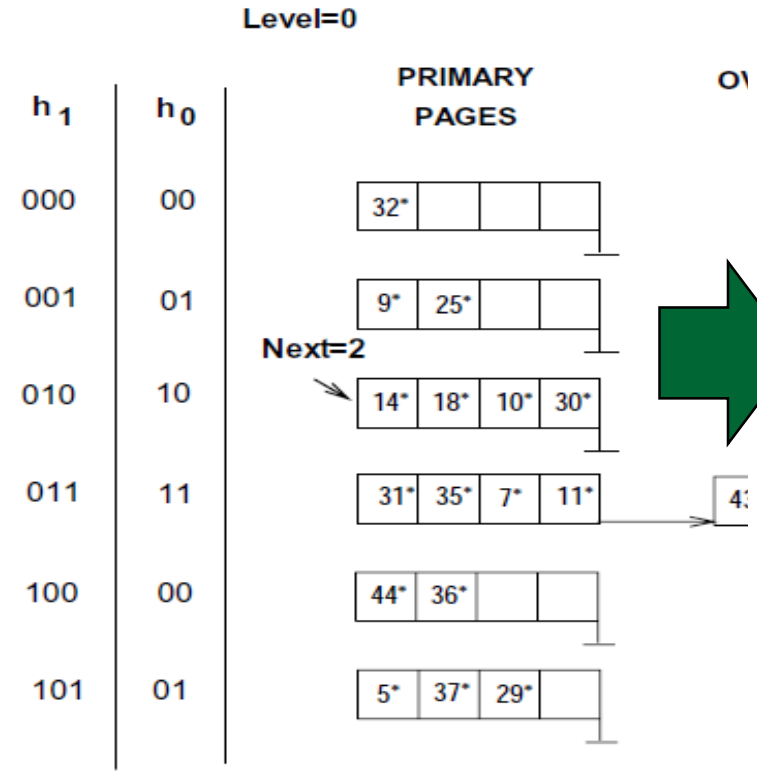


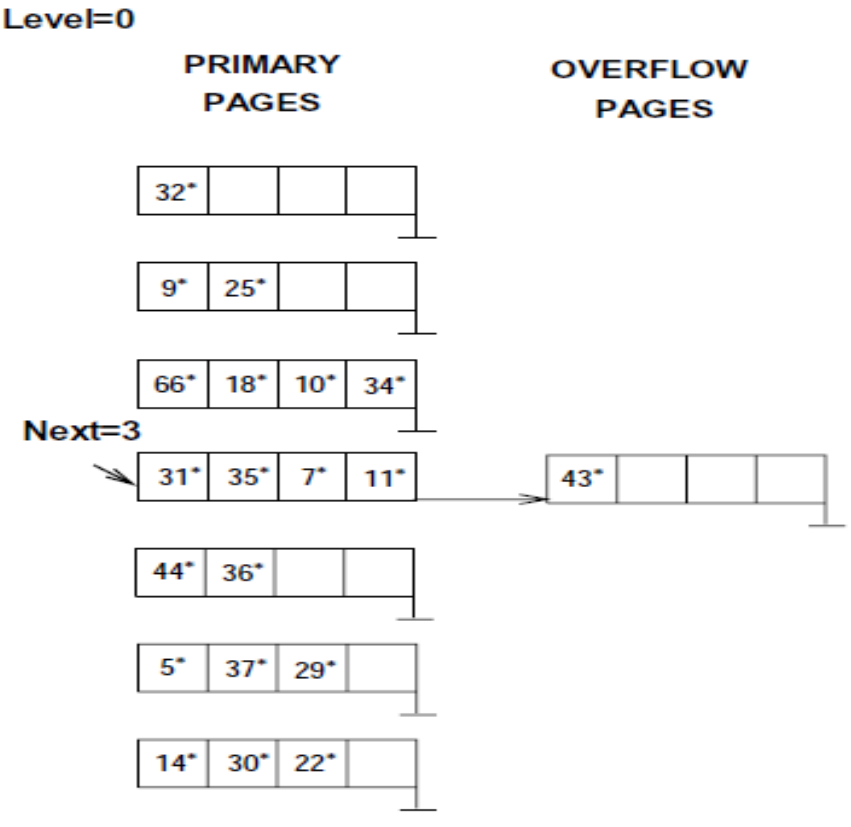
$h_0, h_1, h_2, \ldots,$

After Inserting Data Entries of 22*, 66* and 34*

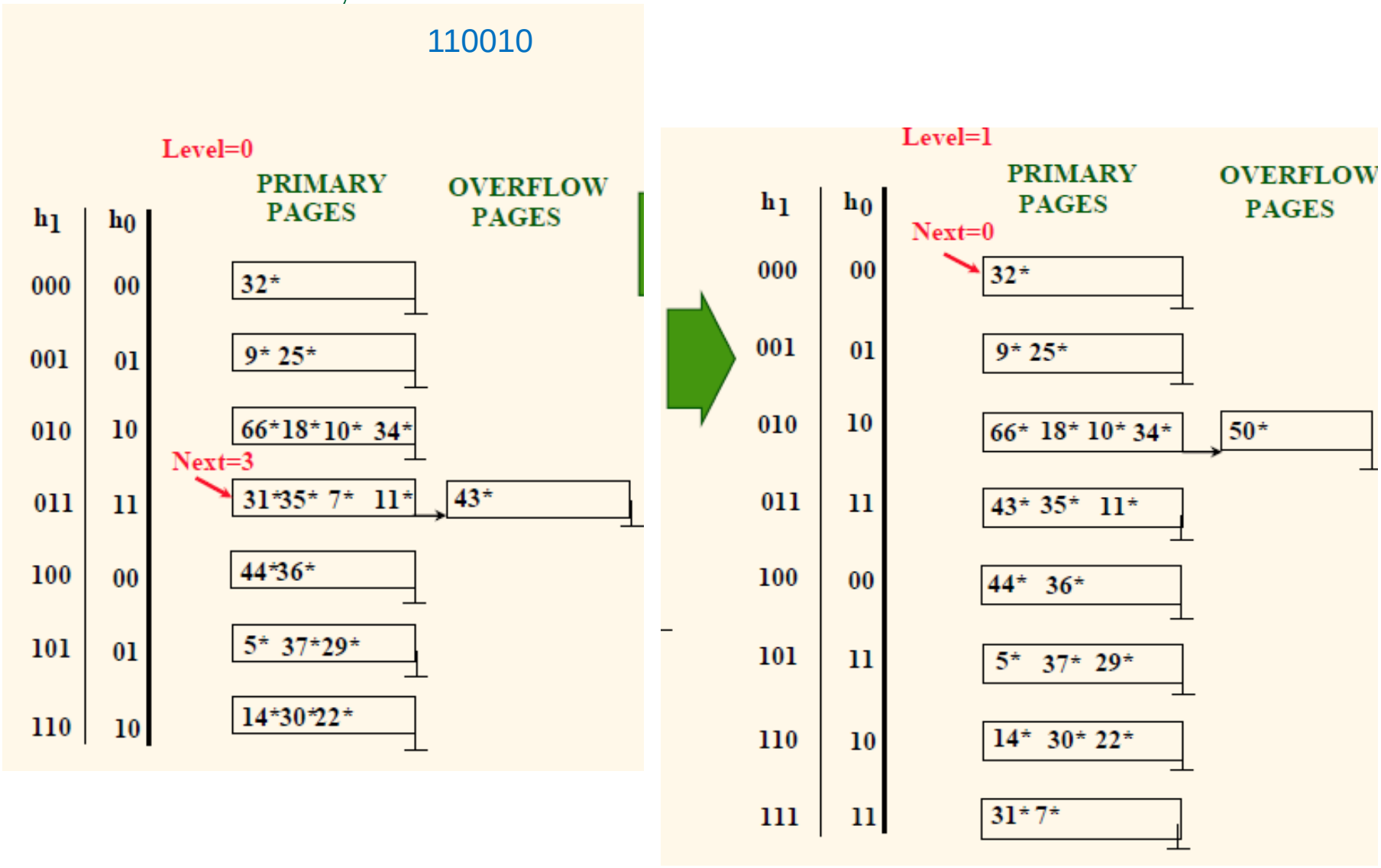
10110

1000010 100010





Example: End of a Round, After Inserting Data Entry 50*.



Extendible VS. Linear Hashing

- Imagine that we also have a directory in LH with elements 0 to *N*-1.
  - The first split is at bucket 0, and so we add directory element *N*.
  - We process subsequent splits in the same way,即 <1,*N*+1>, <*2,N*+2>, ... ,<*N-1,2N-1*>
  - And at the end of the round, all the orginal *N* buckets are split, and the directory is doubled in size.
- i.e., LH doubles the imaginary directory gradually.

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

- 要求: 掌握可扩展和线性 Hash 索引的构建方法, 即插入与删除