# Hash-Based Indexes

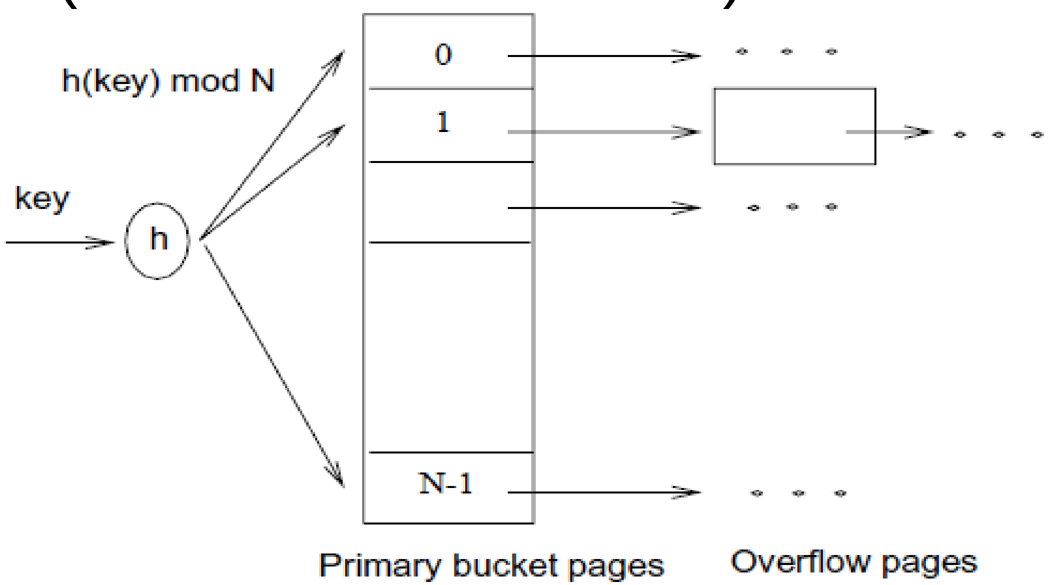
Jianlin Feng
School of Software
SUN YAT-SEN UNIVERSITY

# 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>
  - Choice orthogonal to the indexing technique
- 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

- The number of primary pages is fixed.
- Primary pages are allocated sequentially, never de-allocated;
  - overflow pages if needed.
- **h(*k*) *mod N* =** bucket to which data entry with key k belongs. (*N* = number of buckets)



# Static Hashing (Contd.)

- Buckets contain *data entries.*
- Hash function 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

- 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, double the number of buckets by doubling the directory,*
  - *splitting just the bucket that overflowed!*

# Extendible Hashing (Contd.)

- Directory is much smaller than file, so doubling it is much cheaper.
- Only one page of data entries is split. *No overflow page!*
- Trick lies in how hash function is adjusted!

# Extendible Hashing Equals Balanced Radix Search Trees (1)

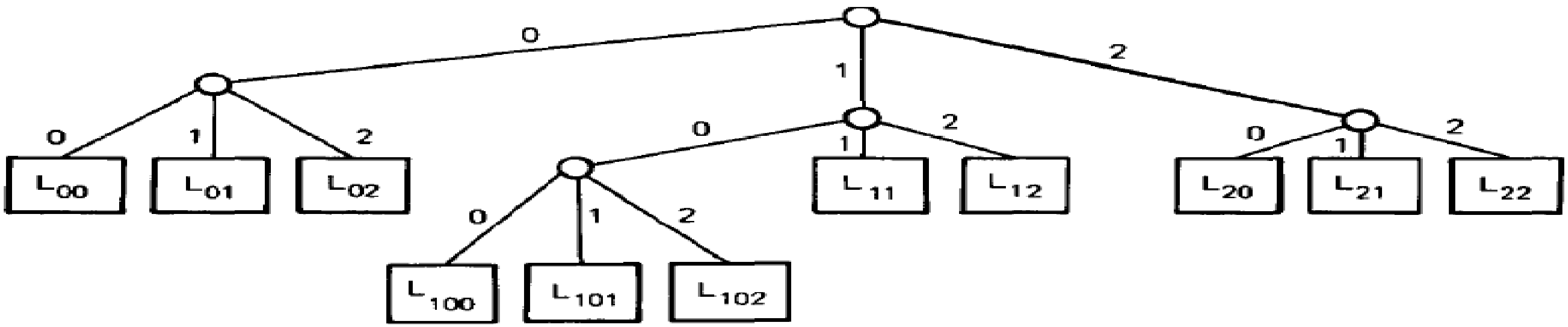



Fig. 1. A radix search tree

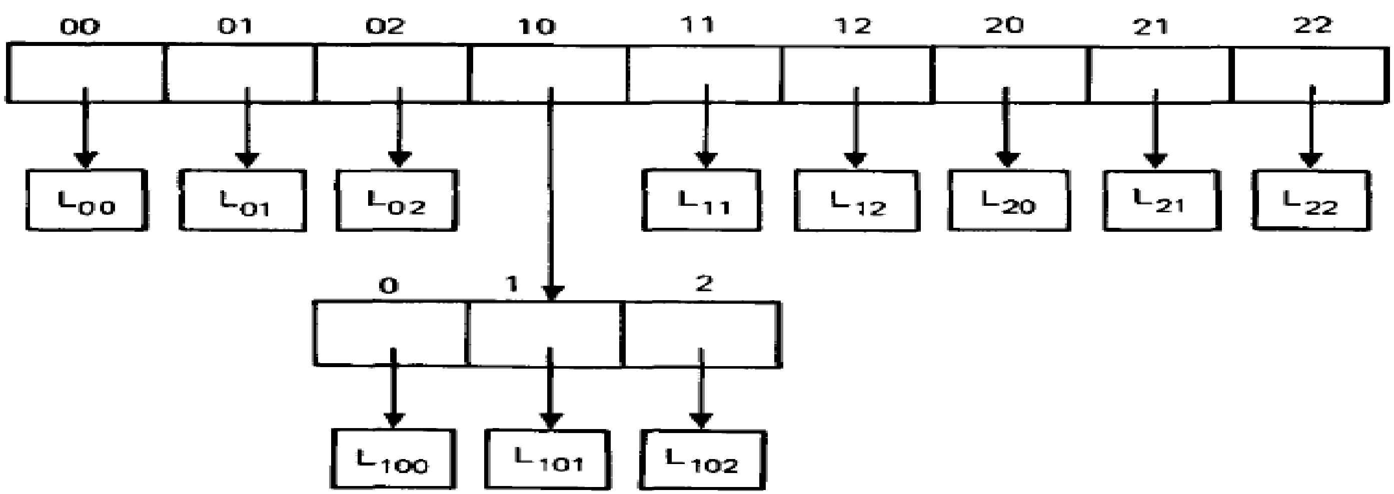



Fig. 2. Radix search tree with two levels compressed into one

# Extendible Hashing Equals Balanced Radix Search Trees (2)

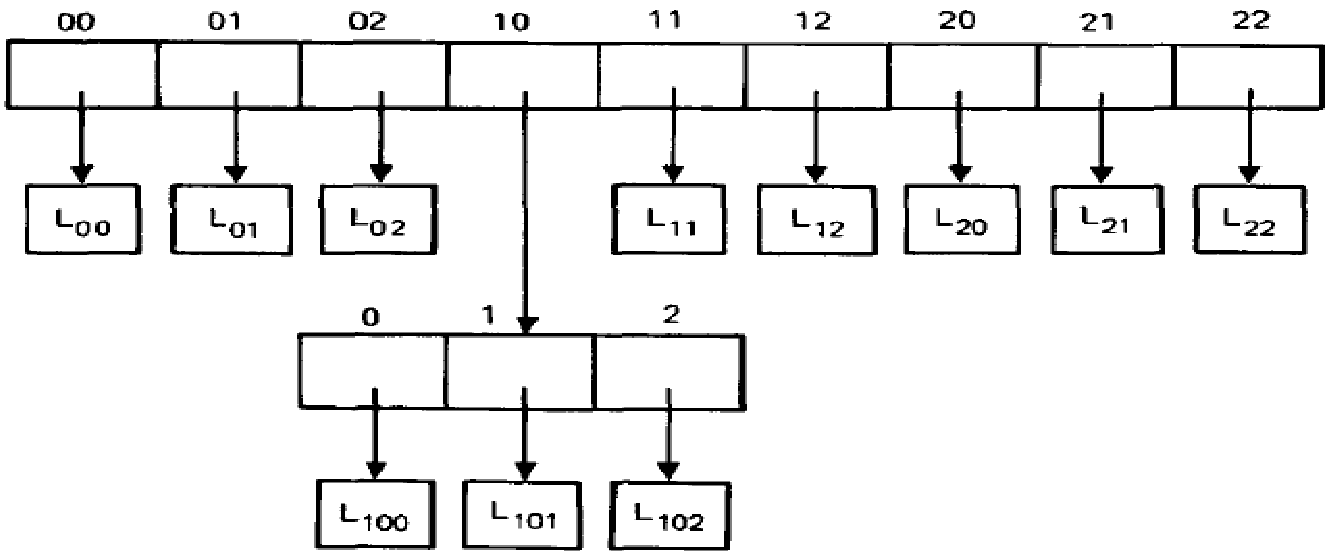



Fig. 2. Radix search tree with two levels compressed into one

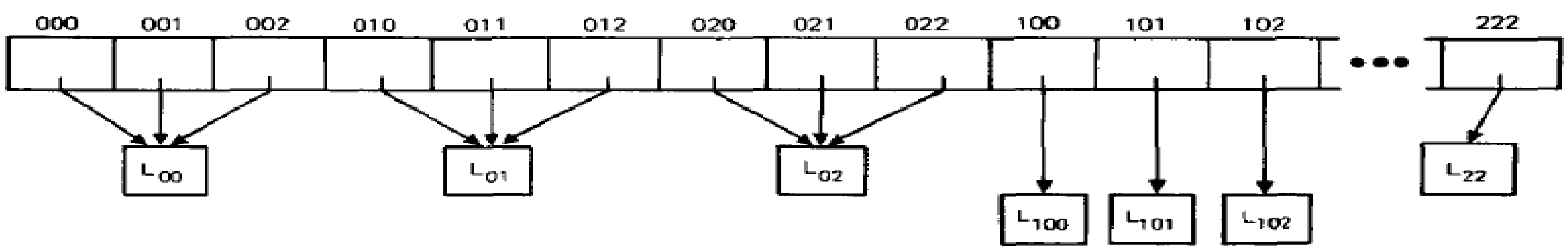
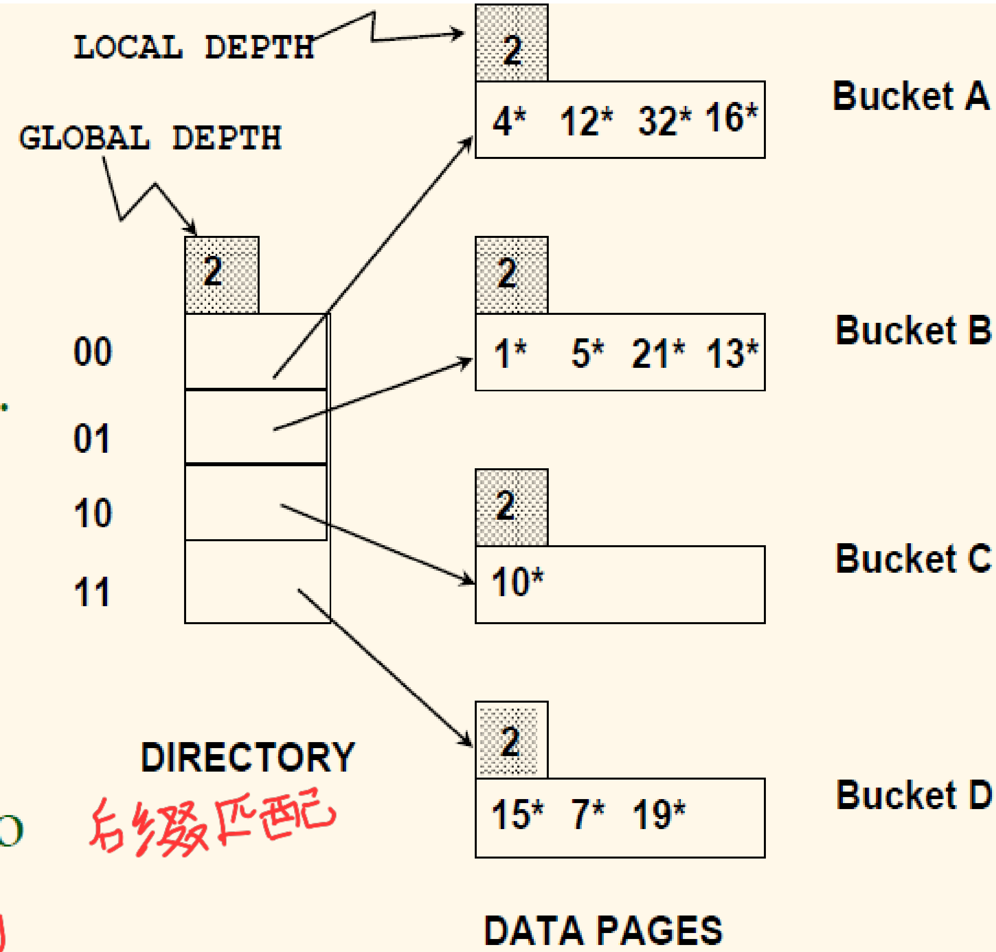



Fig. 3. Degenerate radix search tree

# *Example*

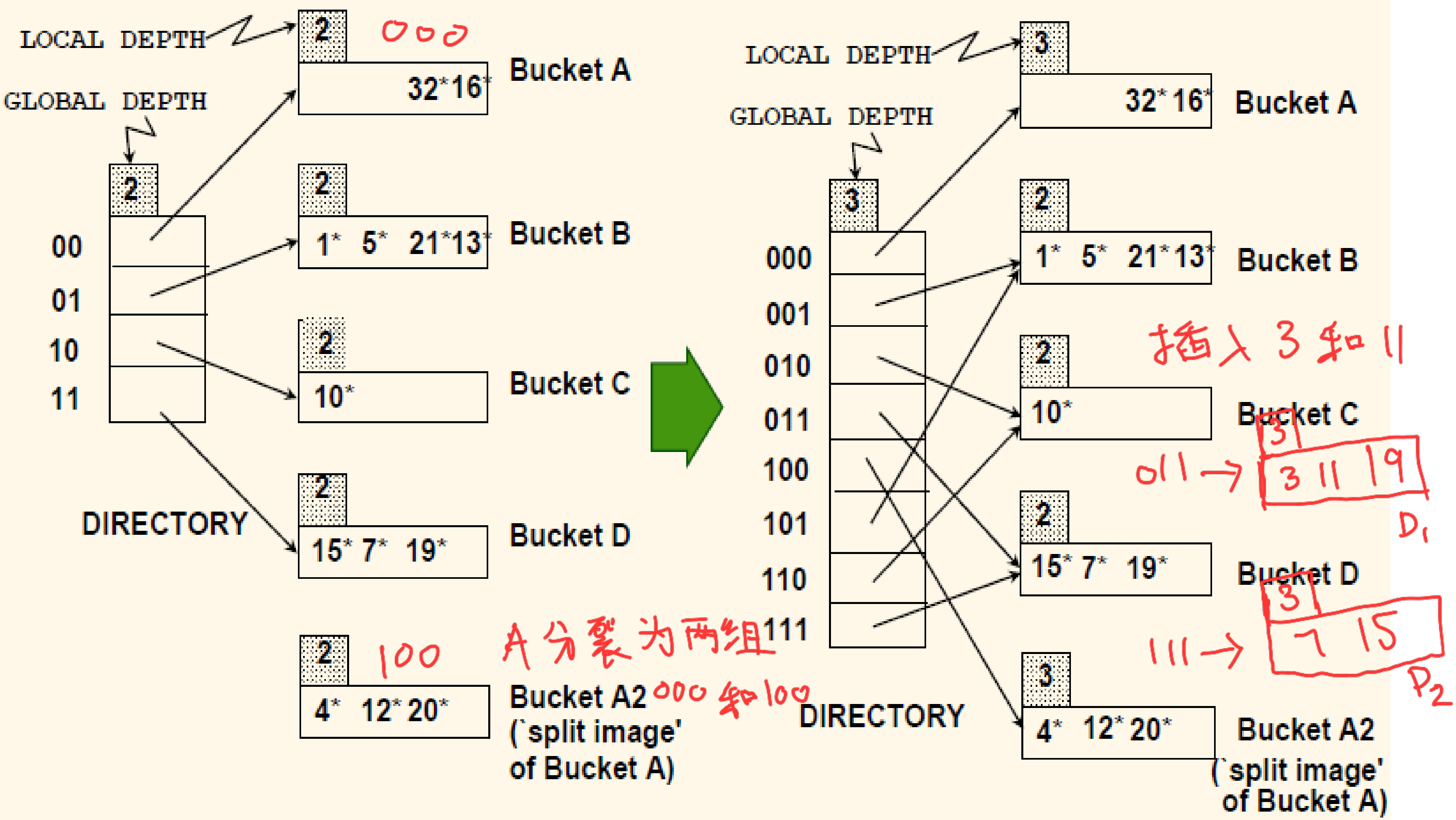



- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

分裂

- **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)*

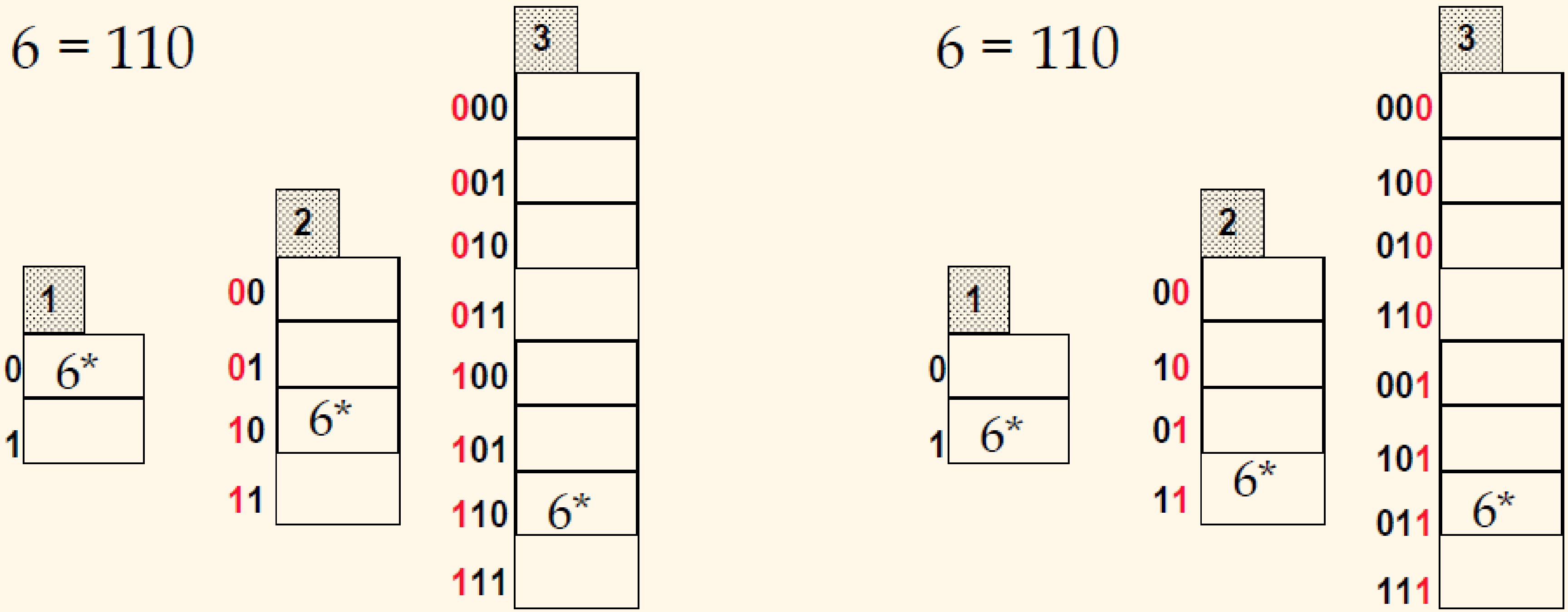


# Points to Note

- 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 number of bits needed to tell which bucket an entry belongs to.*
  - *Local depth of a bucket: number 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 Doubling

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.
  - 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.

# 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, and handles duplicates.
  - What problem will duplicates cause in Extendible Hashing?

# The Idea of Linear Hashing

- Use a family of hash functions $h_0$, $h_1$, $h_2$, ..., where $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
  - $\mathbf{h_i}(key) = h(key) \bmod (2^i N)$; *N = initial # buckets*
  - *h is some hash function (range is not 0 to N-1)*
  - *If $N = 2^{d0}$, for some d0, $h_i$ consists of applying h and looking at the last di bits, where di = d0 + i.*

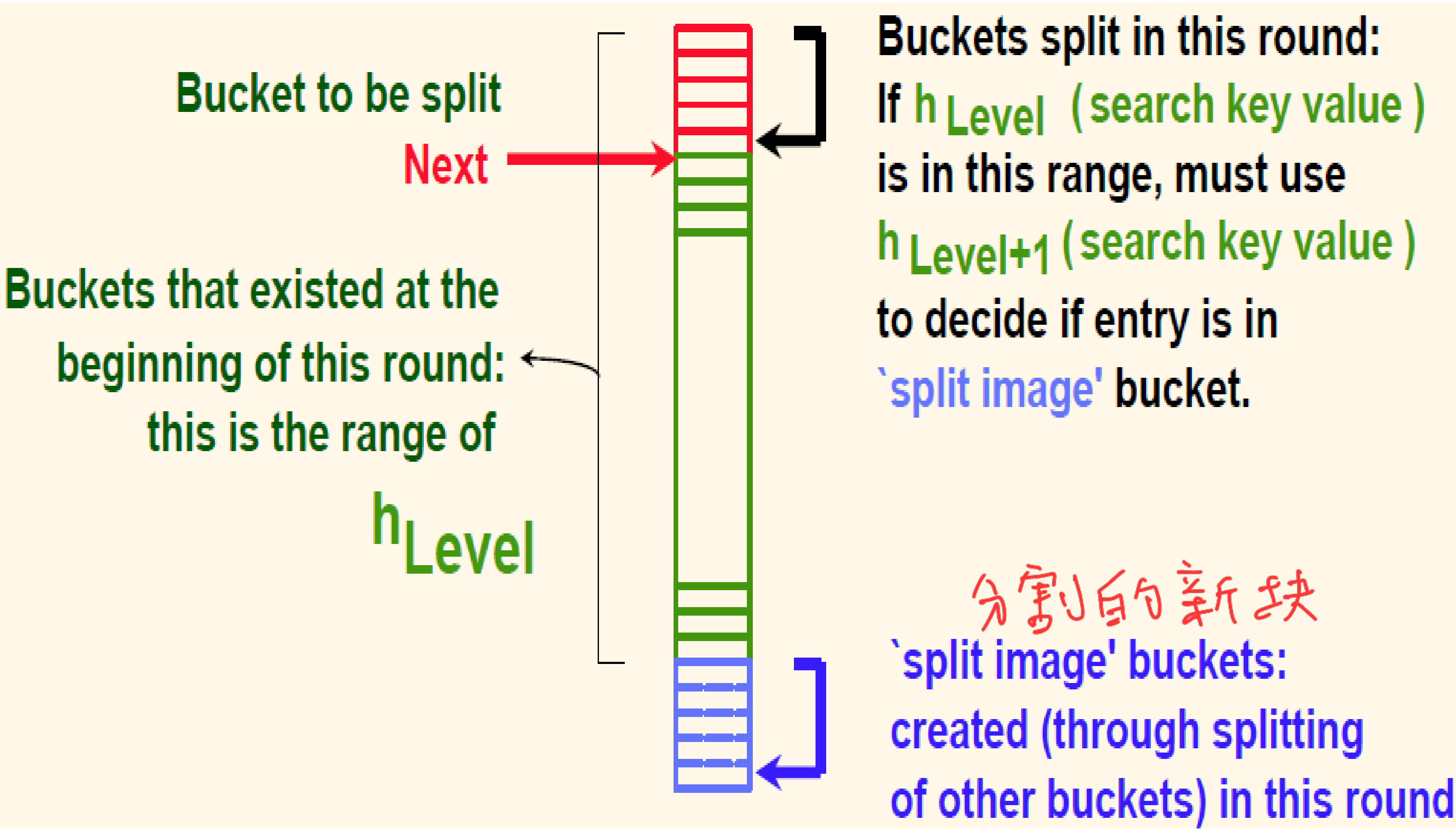
# The Idea of Linear Hashing (Contd.)

- 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.

    0~Next-1 已拆分　　Next 后未拆分
  - Buckets 0 to *Next-1* have been split; *Next* to $N_R$ yet to be split.
  - Current round number is *Level*.

# Overview of LH File:

## in the Middle of the *Level*–th Round



# Search in Linear Hashing

- 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 $h_{Level}$ / $h_{Level+1}$:
  - 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.

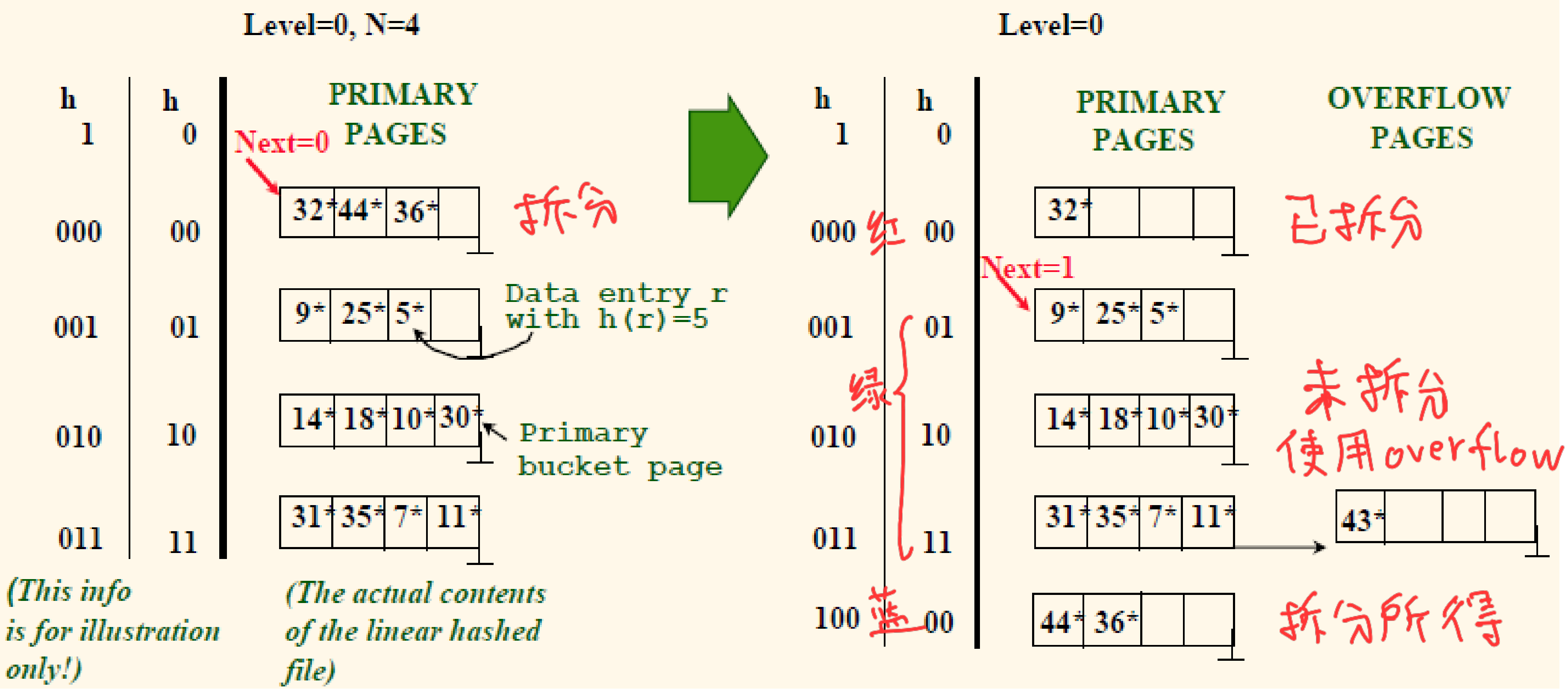
# Bucket Split

- 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;
  - the split image is therefore bucket number $b+N_{Level}$.
  - *Next* ← *Next* + 1.

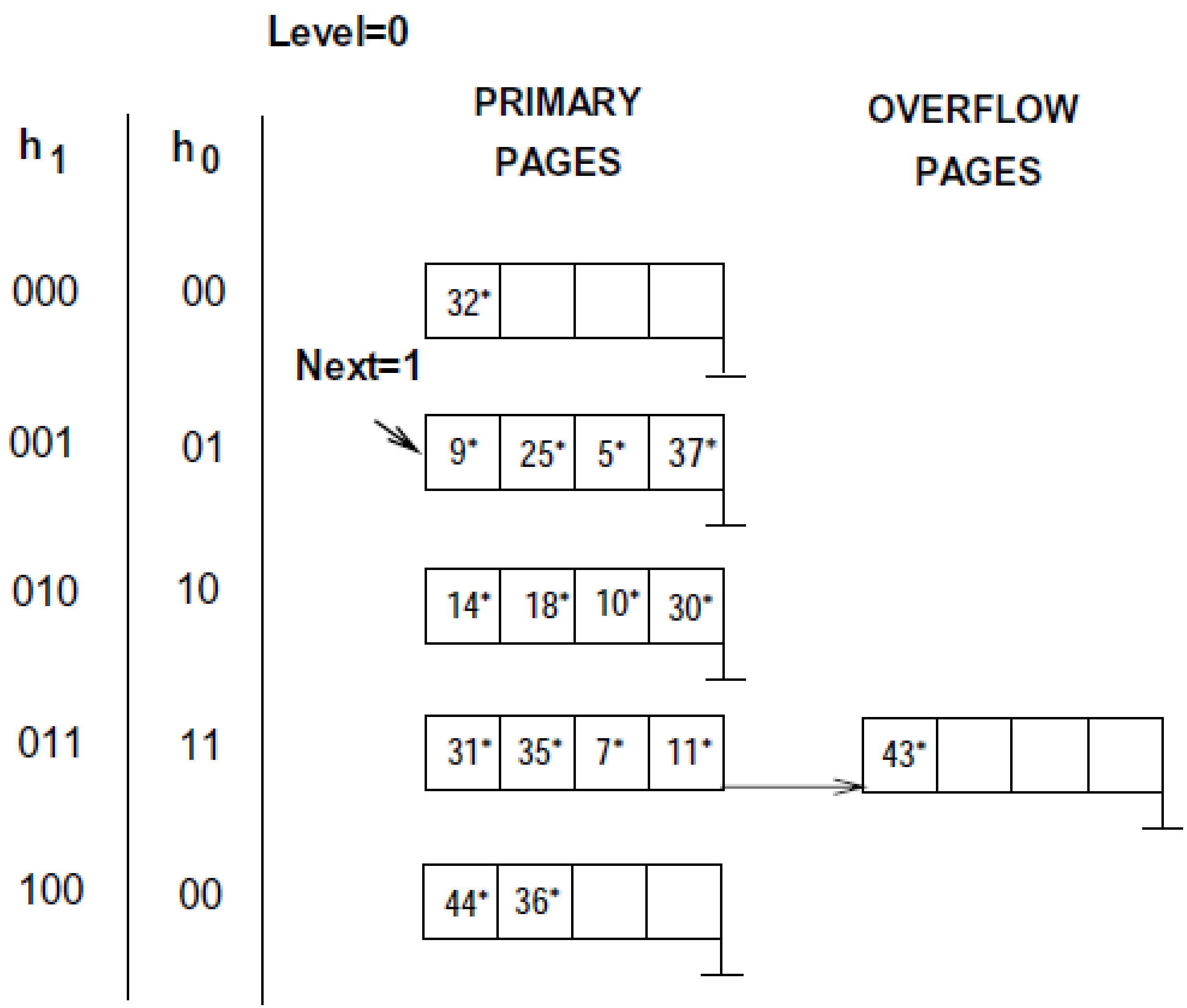
# Example of Linear Hashing

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

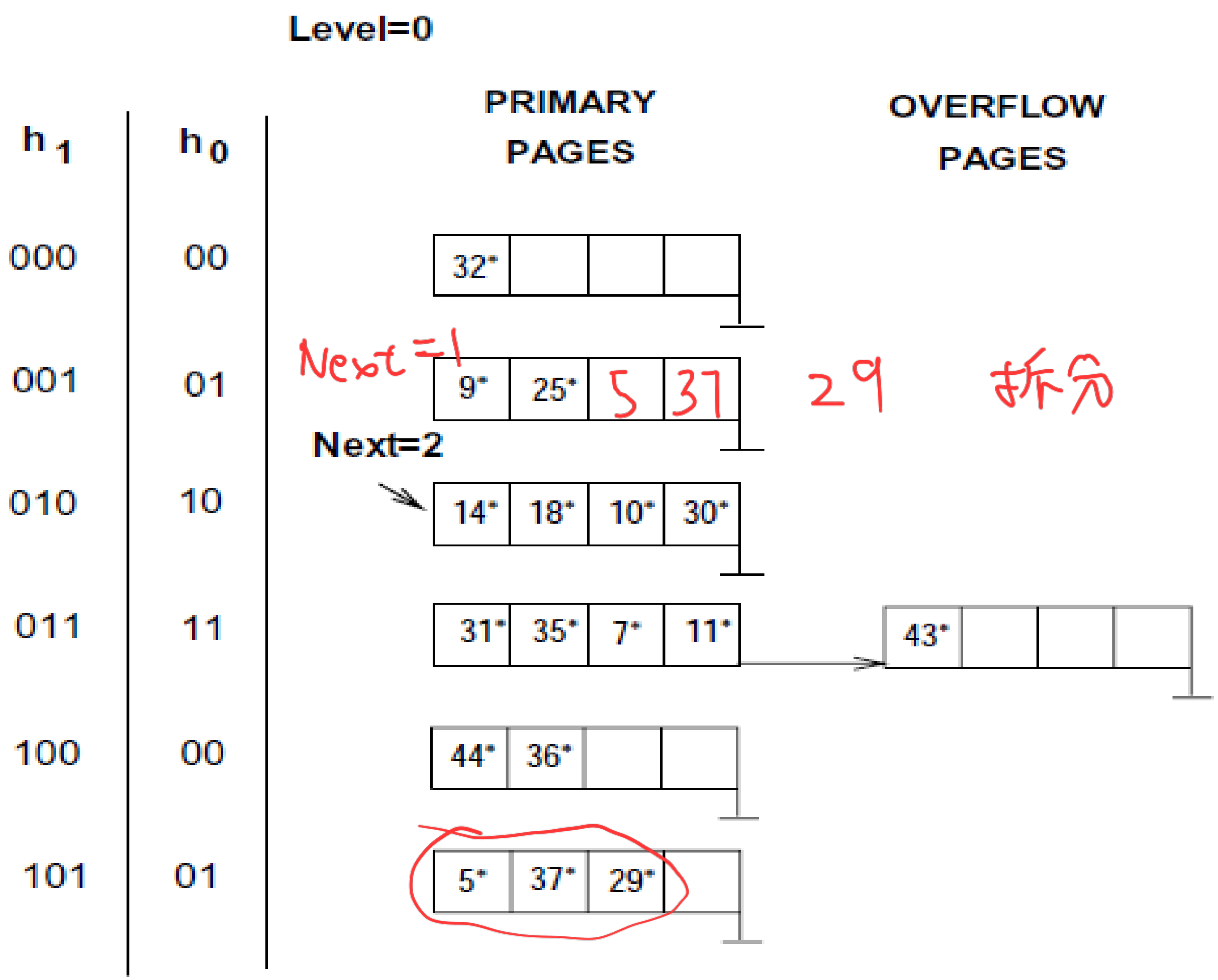
Insert data entry of 43*



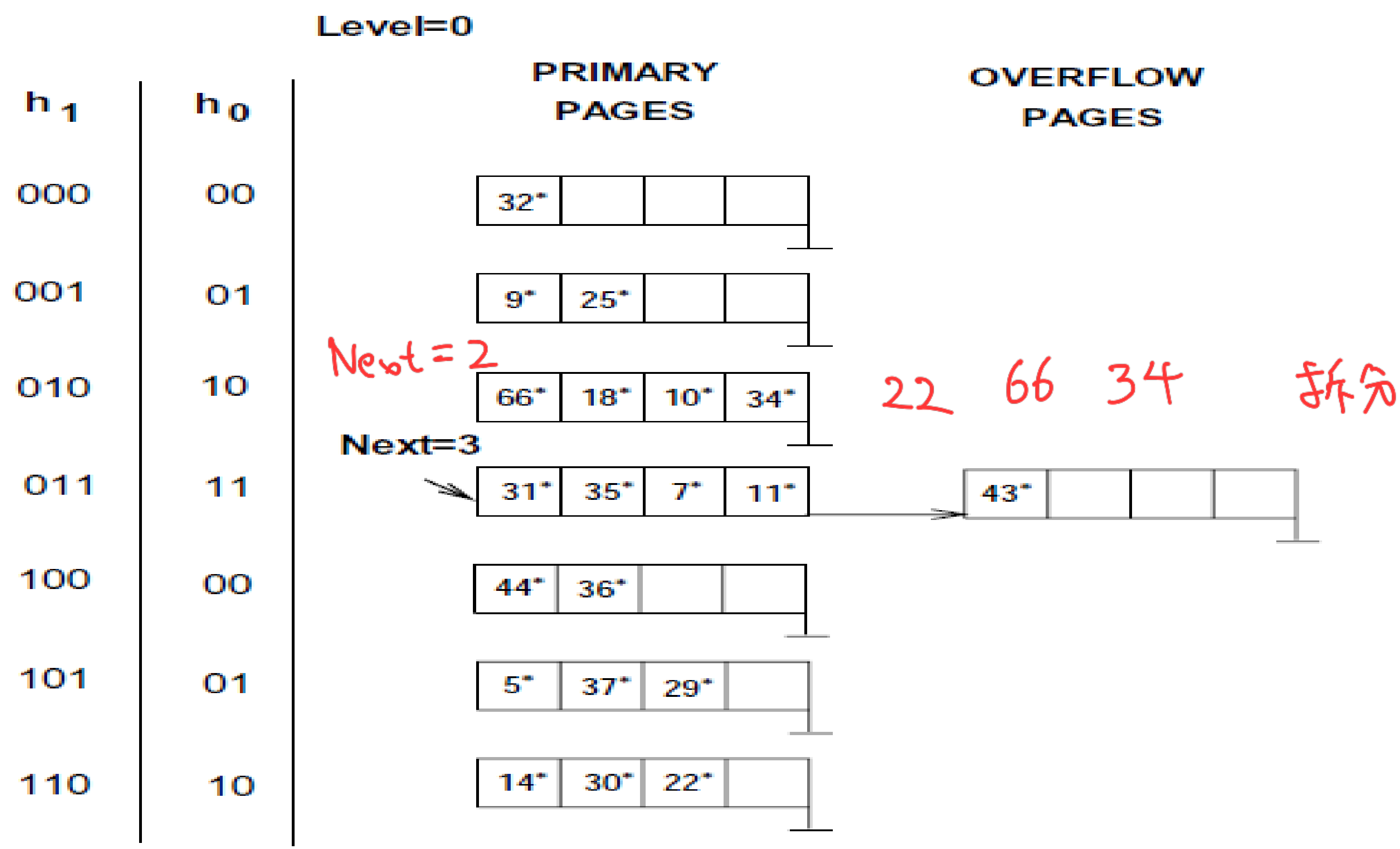
# After Inserting Data Entry of 37*



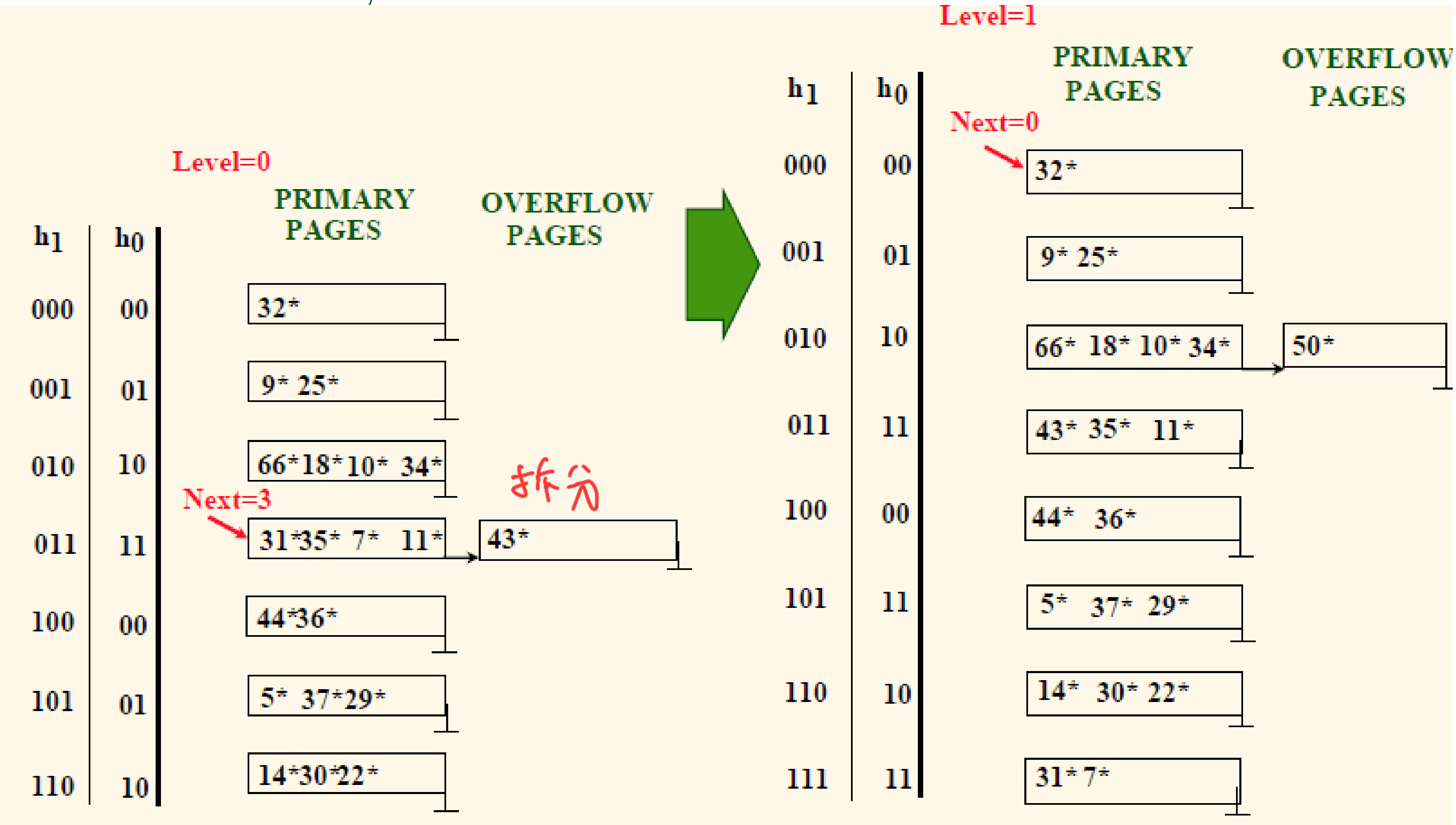
# After Inserting Data Entry of 29*



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



# 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*.
  - Imagine directory being doubled at this point, but elements <1,*N*+1>, *<2,N+2>, ... are the same. So, we can avoid copying elements from 1 to N*-1.
  - We process subsequent splits in the same way,
  - 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.