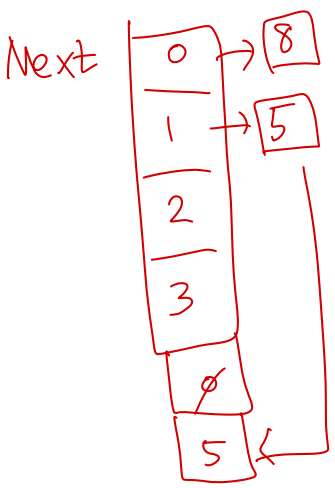



# Database Systems
# Lecture13 – Chapter 14: Indexing

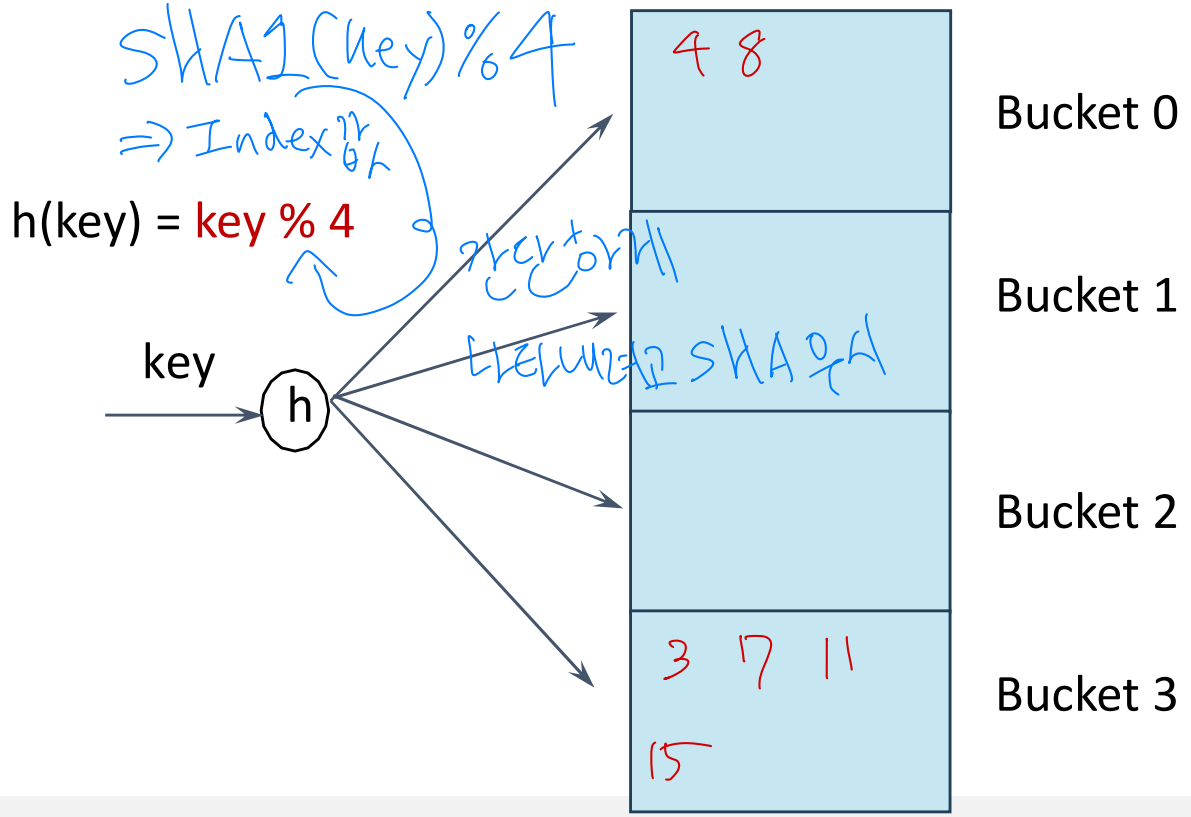
Beomseok Nam (남범석)
bnam@skku.edu

# Static Hashing

- **Bucket**: Storage unit holding index entries
  - (typically 4KB disk block)
- **Hash Function (h)**: Maps search keys → bucket addresses
- Different keys may map to the same bucket (**collisions**)
  - Entire bucket needs to be searched **sequentially**
- **Hash Index**: Buckets store ***pointers to records***
- **Hash File** : Buckets store ***records***

# Static Hashing

- Fixed-size array of buckets (e.g., HT = new Bucket[size]; )
- h(key) → get bucket index in array
  - Often uses a **cryptographic hash function** (e.g., SHA-256) followed by modulo: H(key) { return SHA256(key) % size; }
  - Let's assume H(key) = key % size for simplicity



# Handling of Bucket Overflows

- Bucket overflow occurs because of Hash Collision

[1] Insufficient buckets

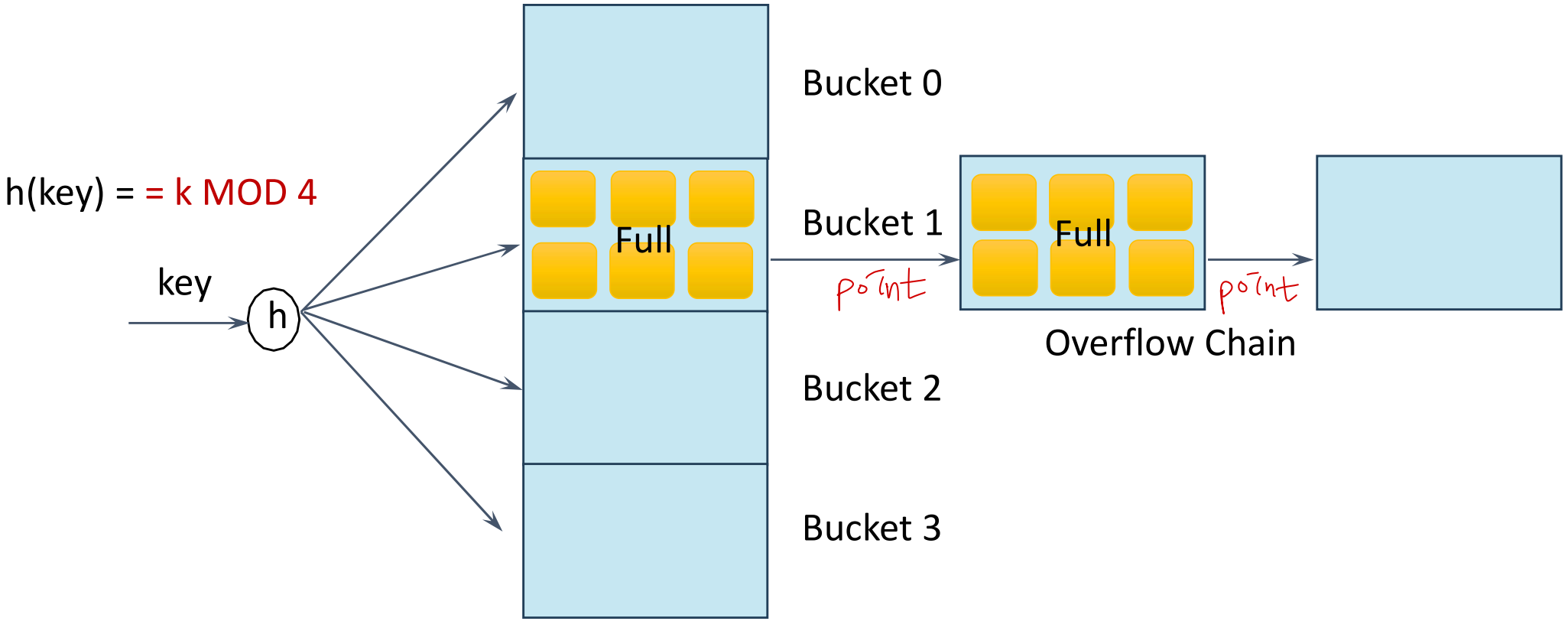
- This type of bucket overflow cannot be eliminated

[2] Skew in distribution of records.

- Skew occurs due to two reasons:
  - Hash function produces skewed distribution of keys
- Overflow is handled by either
  - **Overflow Chain**
    or
  - **Closed Hashing (Open Addressing)**

# Handling of Bucket Overflows

- **Overflow Chain** (open Hashing)
  - Overflow buckets are chained together in a linked list.



# Handling of Bucket Overflows (Closed Hashing)

- **Closed Hashing (a.k.a Open Addressing)**
  - ***Linear probing***: Use the next bucket (in cyclic order) that has space.

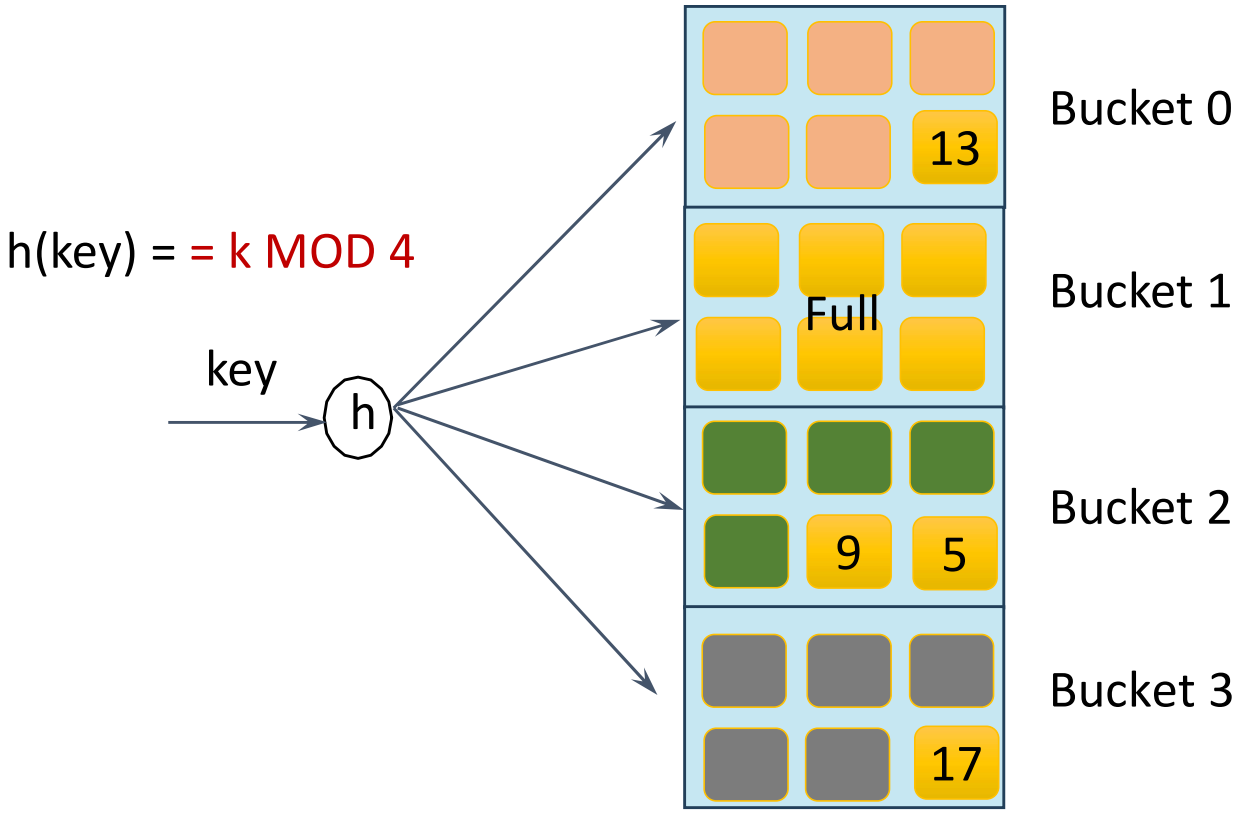
Next Free space의 위치 ⇒ O(B)

Bucket수

원래 Index가 full이면

다음거 → 다음거 → 찾다보면

Worst = B




Problem: Worst Search Complexity = O(N). → Limit Probing Distance.

# Example of Hash File

- Hash file of *instructor* file, using *dept_name* as key
  - E.g. h(Music) = 1
  - h(History) = 2
  - h(Physics) = 3
  - h(Elec. Eng.) = 3

bucket 0

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

bucket 1

| | | | |
|---|---|---|---|
| 15151 | Mozart | Music | 40000 |
| | | | |
| | | | |
| | | | |

bucket 2

| | | | |
|---|---|---|---|
| 32343 | El Said | History | 80000 |
| 58583 | Califieri | History | 60000 |
| | | | |
| | | | |

bucket 3

| | | | |
|---|---|---|---|
| 22222 | Einstein | Physics | 95000 |
| 33456 | Gold | Physics | 87000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| | | | |

bucket 4

| | | | |
|---|---|---|---|
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| | | | |
| | | | |

bucket 5

| | | | |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| | | | |
| | | | |
| | | | |

bucket 6

| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| | | | |

bucket 7

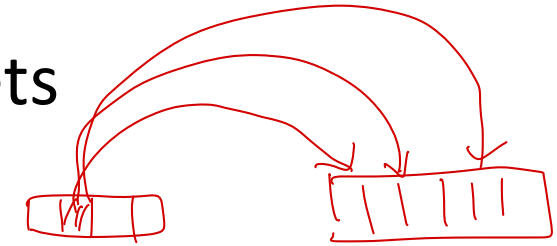
| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

The given hash function fails to provide a uniform distribution.

→ Static Hashing is a Bad Choice

# Deficiencies of Static Hashing

- Static Hashing → Fixed-size array of buckets



- ***Databases grow or shrink with time***.
- Solution: **Rehashing**
  - periodic re-organization using a new hash function
  - Expensive, disrupts normal operations

Stop

- Better solution: *allow the number of buckets to be modified dynamically.*

# Dynamic Hashing

- *Trick lies in how hash function is adjusted!*

- **Extendable Hashing**
  - Designed for **disk-based** hash tables
  - Buckets may be **shared by multiple hash values**
  - Uses a **directory (array)** of pointers to buckets
    - *Idea*: Add a level of indirection!

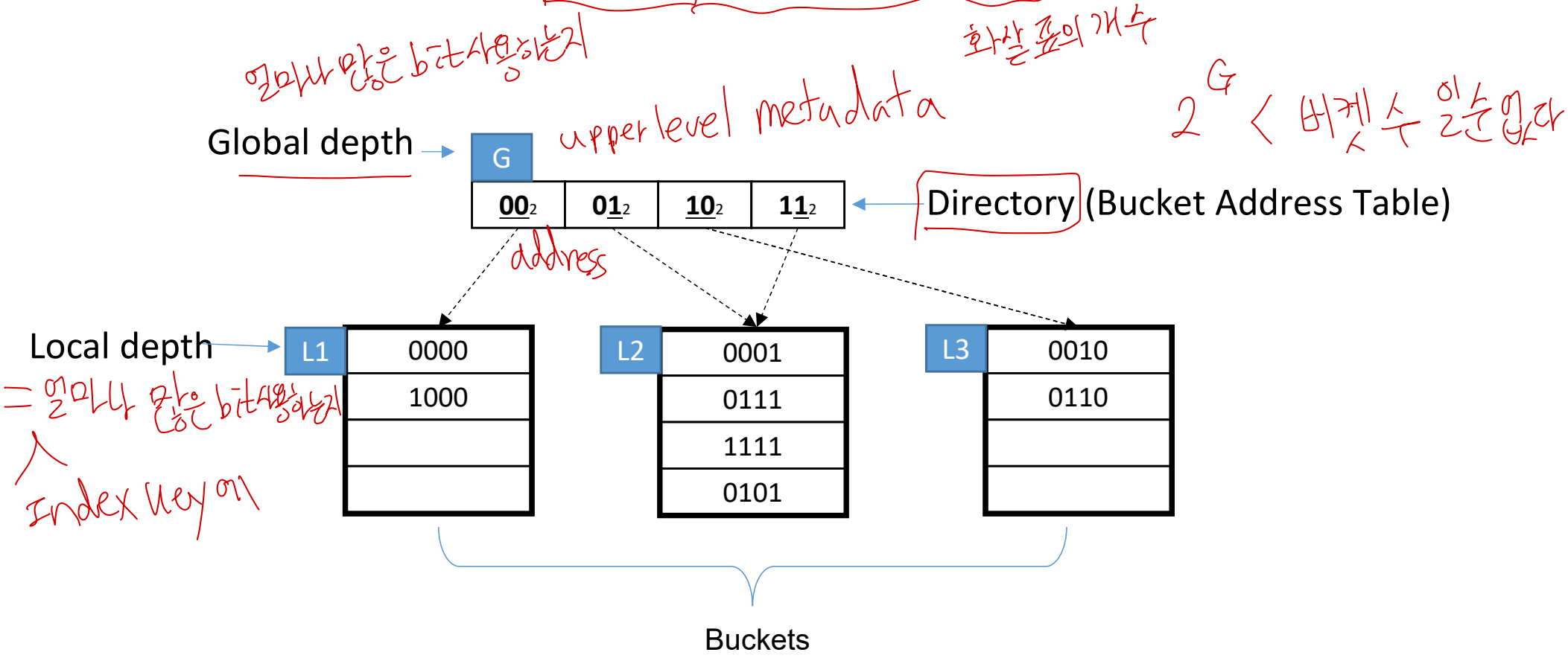
- **Linear Hashing**
  - Do rehashing in an incremental manner

# Extendable Hashing

- Use **prefix (or postfix)** bits of the hash key
- Let ***Global Depth*** **G = # of bits used**
  - Number of buckets <= $2^G$
  - **Initially, G = 0**, grows/shrinks with DB size
- Buckets do not need to be stored contiguously on disk.
  - Need Directory (i.e., Bucket Address Table)
  - directory size = $2^G$
- Dynamic Behavior
  - **Multiple directory entries** may point to the **same bucket**
  - ***Local Depth*****:** # of bits used by each bucket
  - Buckets split/merge as data changes

# Extendable Hash Structure

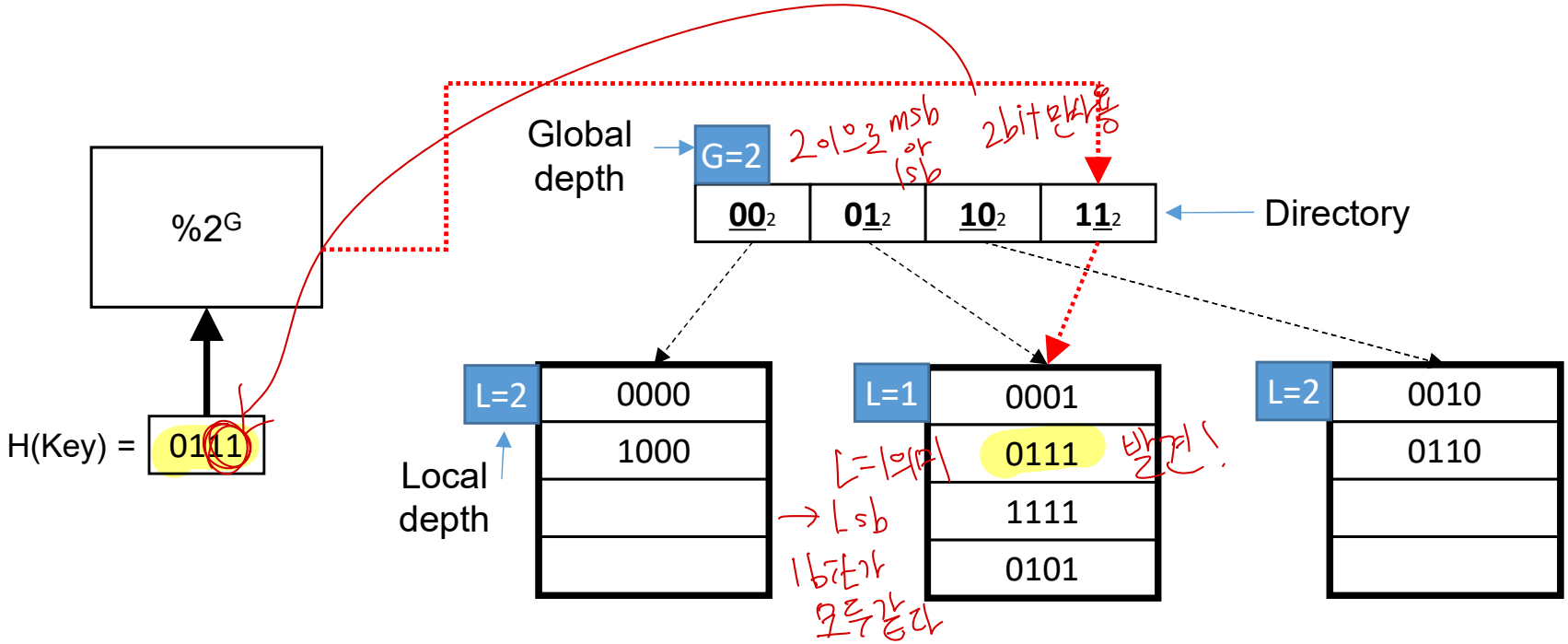
- ***Global Depth*** **G = # of bits used** to look up the directory
- ***Local Depth*** **L = # of bits used** to index keys in each bucket
- A bucket is shared by $2^{(G-L)}$ directory entries




In this example, *L1* = *L3* = *G* = *2*, whereas *L2* = *G* – 1 = 1

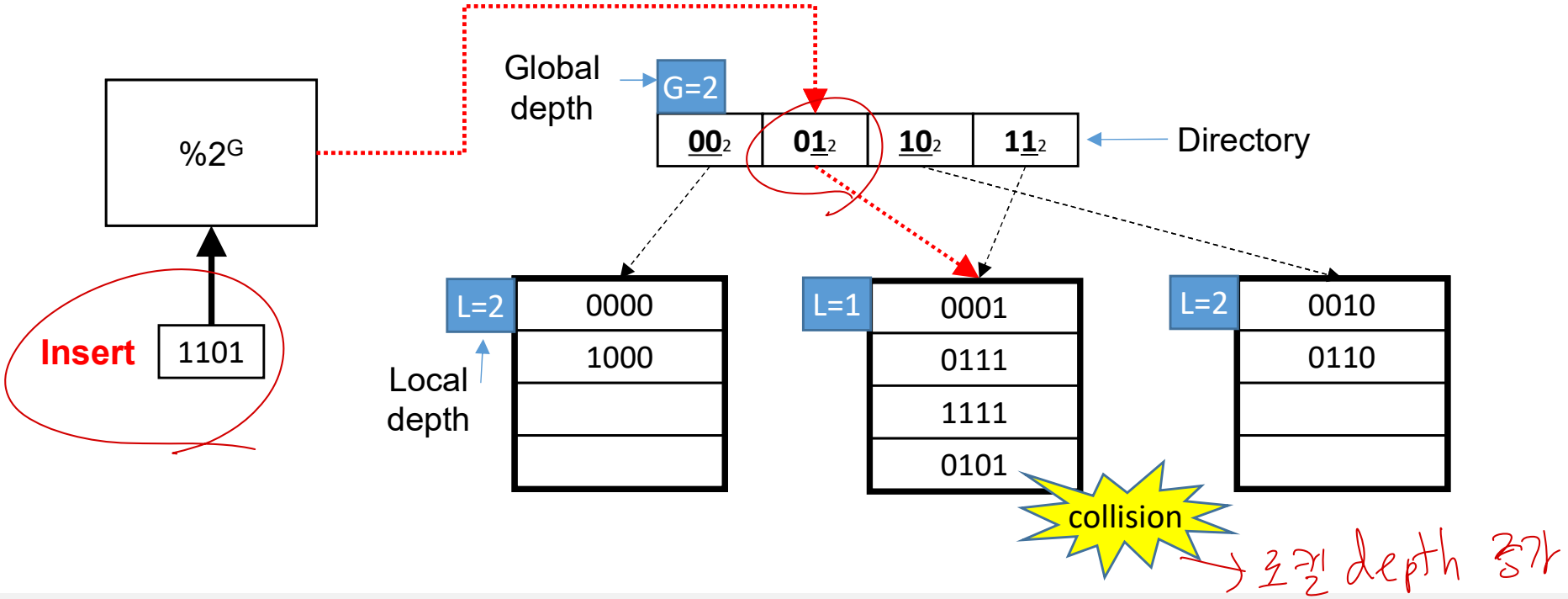
# Extendable Hashing – How it Works

- Directory is an array of size 4 ($2^2$), so 2 bits are needed.
- We will use postfix in the examples
  - global depth = # of least significant bits of h(K);
- If h(K) = 5 = binary 01**01**, it is in bucket pointed to by 01.
- If h(K) = 7 = binary 01**11**, it is in bucket pointed to by 11



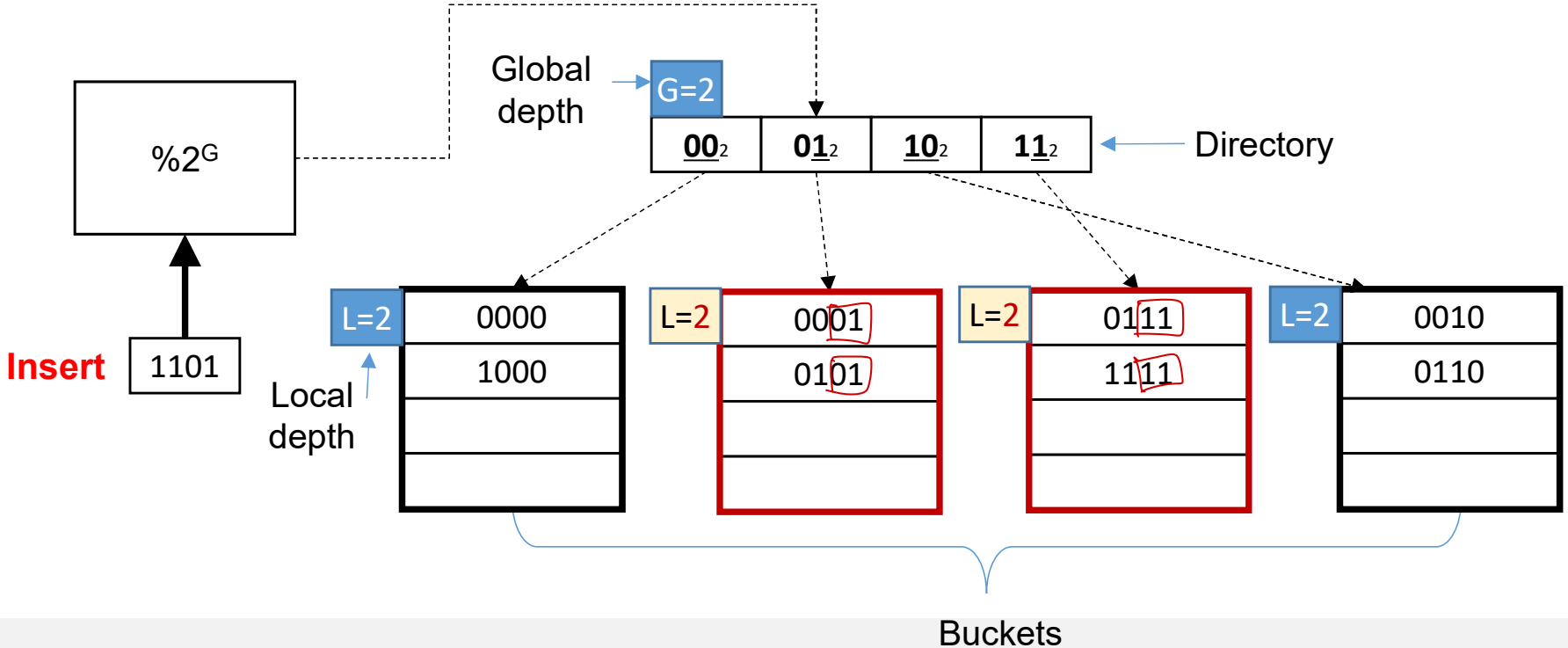
# Extendable Hashing – How it Works

- Bucket Split
  - if G> *L* (more than one pointer to bucket)
    - allocate a new bucket, and set *L* = *L+1*
    - Update the directory to point to the new bucket
    - move records in the overflow bucket to the new bucket
    - Further splitting is required if the bucket is still full



# Extendable Hashing – How it Works

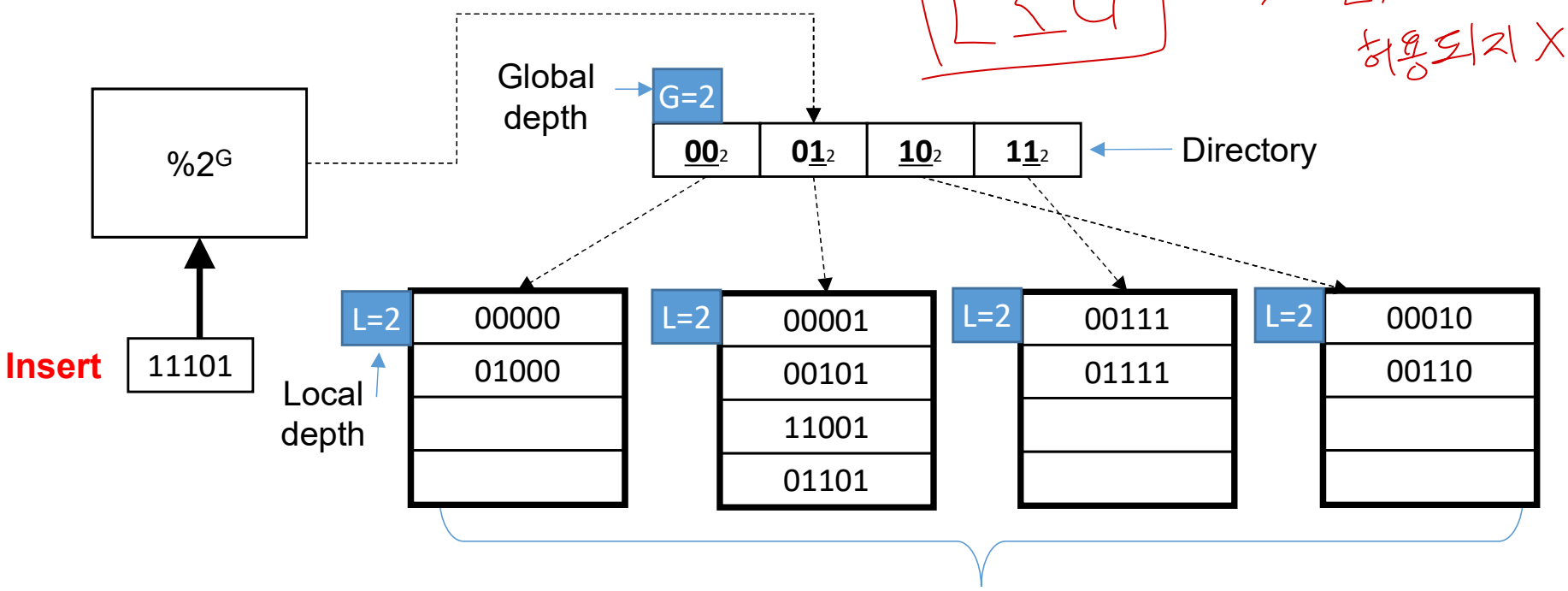
- Bucket Split
  - if G> *L* (more than one pointer to bucket)
    - allocate a new bucket, and set *L* = *L+1*
    - Update the directory to point to the new bucket
    - move records in the overflow bucket to the new bucket
    - Further splitting is required if the bucket is still full



# Extendable Hashing – How it Works

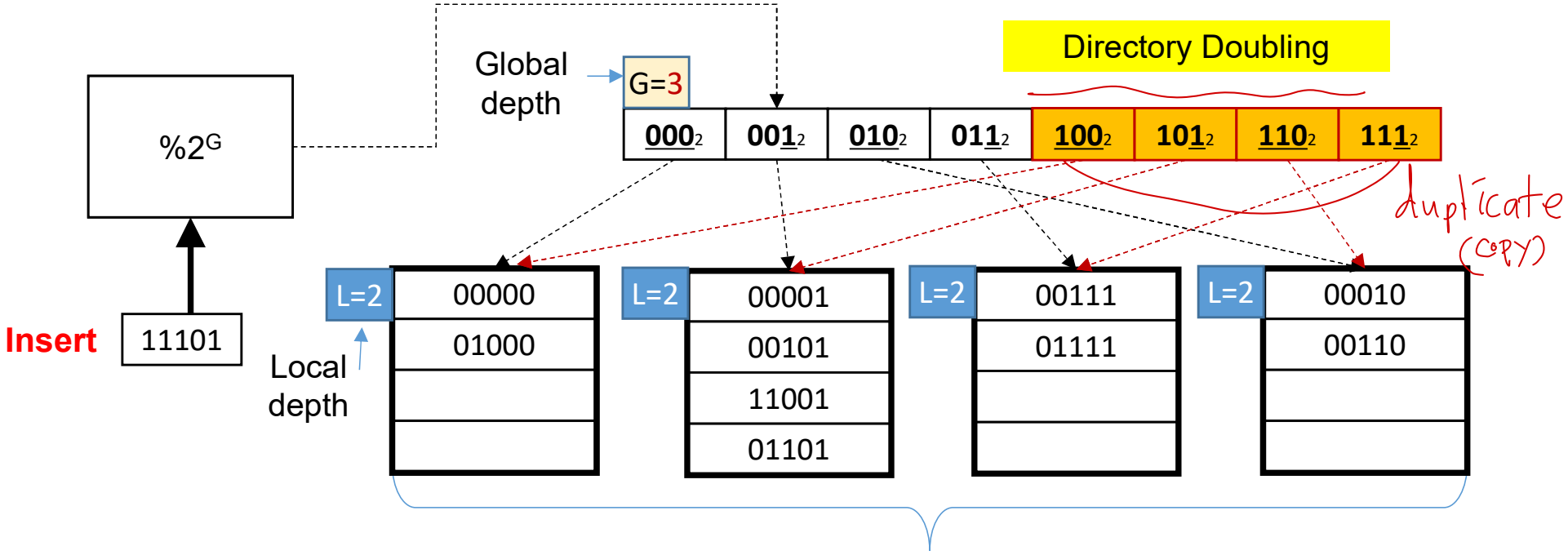
- Bucket Split → ***Directory Doubling***
  - If *G* = *L* (only one pointer to bucket)
    - increment *G* and double the size of directory.
    - Now *G*> *L*.
    - replace each entry by two entries that point to the same bucket.

항상 L<G → L>G되려고하는것은 허용되지 X



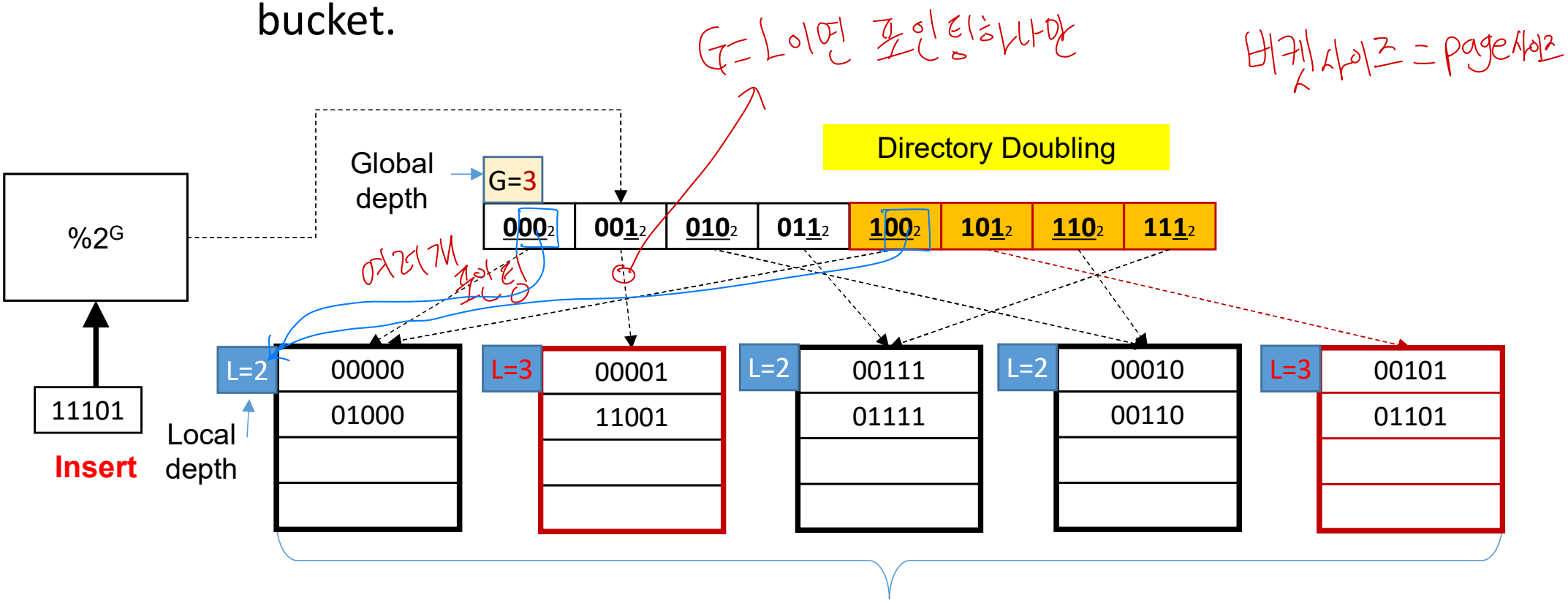
# Extendable Hashing – How it Works

- Bucket Split → ***Directory Doubling***
  - If *G* = *L* (only one pointer to bucket)
    - increment *G* and double the size of directory.
    - Now *G*> *L*.
    - replace each entry by two entries that point to the same bucket.



# Extendable Hashing – How it Works

- Bucket Split → ***Directory Doubling***
  - If *G* = *L* (only one pointer to bucket)
    - increment *G* and double the size of directory.
    - Now *G*> *L*.
    - replace each entry by two entries that point to the same bucket.



# Extendable Hashing - Pros and Cons

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find records
  - Directory doubling is expensive
  - Directory can become large
    - Solution: $B^+$-tree structure for bucket address table

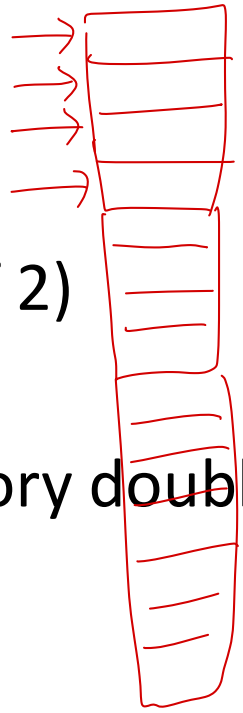
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory
  - At the cost of more bucket overflows

# Linear Hashing – a lazy approach

- **No directory** → uses temporary overflow pages
- **Round-robin splitting**:
  - split the bucket pointed by Next, then advance Next

bucket address 계산 ?

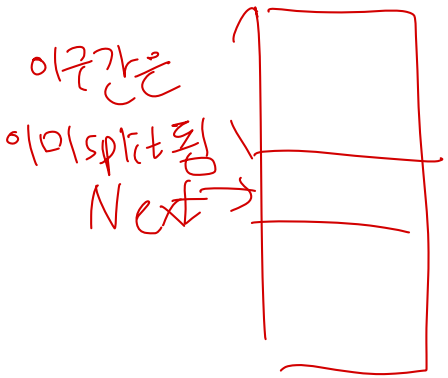
- Use a ***family of hash functions*** $h_0, h_1, h_2, \ldots$
- $h_{Level}(key) = h(key) \bmod (2^{Level} N)$
  - N = initial # buckets (must be a power of 2)
  - h is some hash function
- $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)



다차면 늘리고!

# Linear Hashing - algorithm

- Algorithm works in `rounds'.
- **Level** = Current round number




- At start of a round
  - Total buckets = N * $2^{Level}$ buckets (denoted as $N_{Level}$ )
  - Buckets 0 to *Next* – *1* → already split
  - *Next* to $N_{Level}$ – 1 → not yet split

- End of round when *Next* = $N_{Level}$
  - Move to the next round:

```
Level++;
Next = 0;
```

# Linear Hashing - Insert

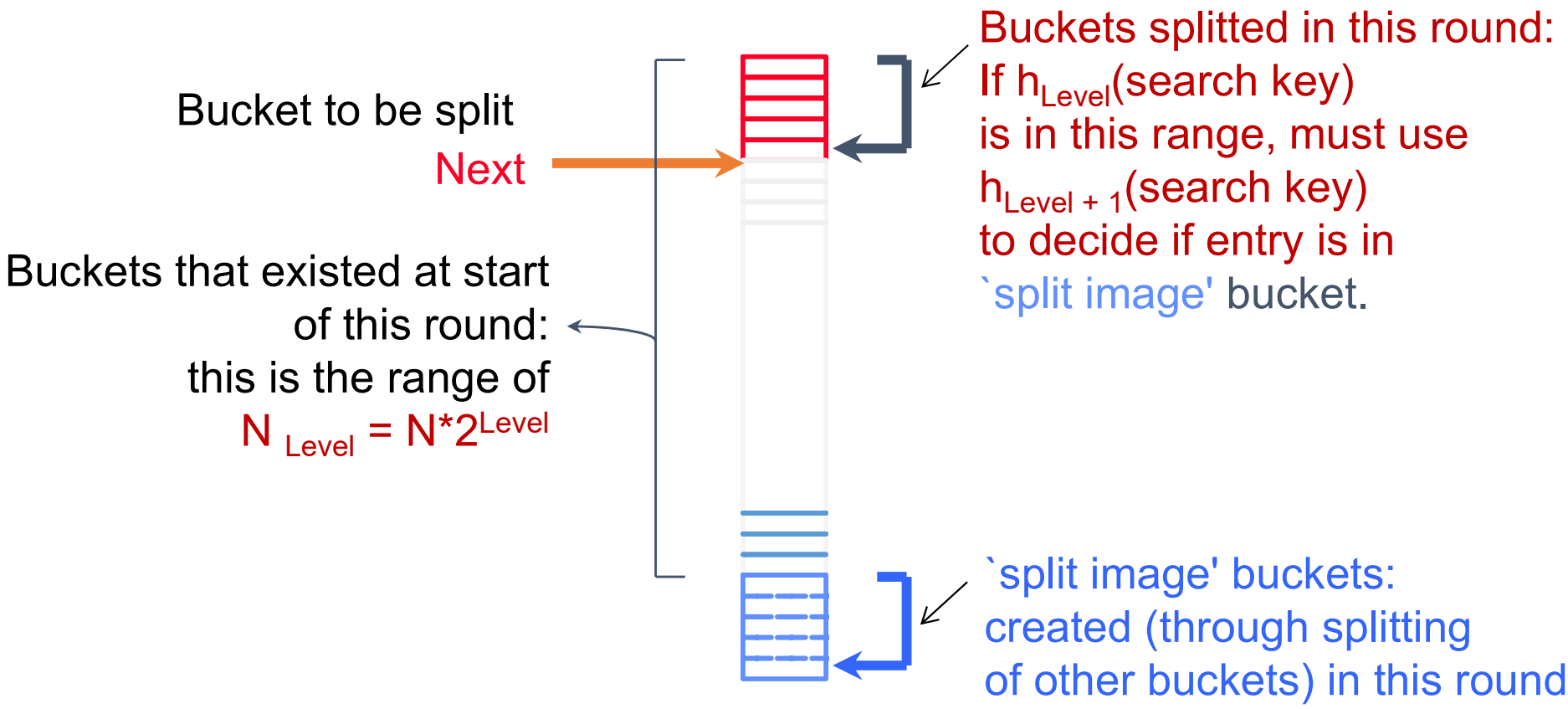
- Find target bucket using current hash ($h_{Level}$)
- If there's space → insert and done.
- If no space:
  - Add a temporary overflow page and insert there.
  - Then, split the bucket pointed by *Next* bucket.
    - *This is likely NOT the bucket that just overflew!!!*
  - Increment *Next*.
  - After split, use **$h_{Level+1}$** to re-distribute entries.

- ⚠️ Buckets are split **in round-robin**, not by need → **Long overflow chains** may develop!

O(N)
↳ overflow chain bucket 수

# Linear Hashing – How it Works

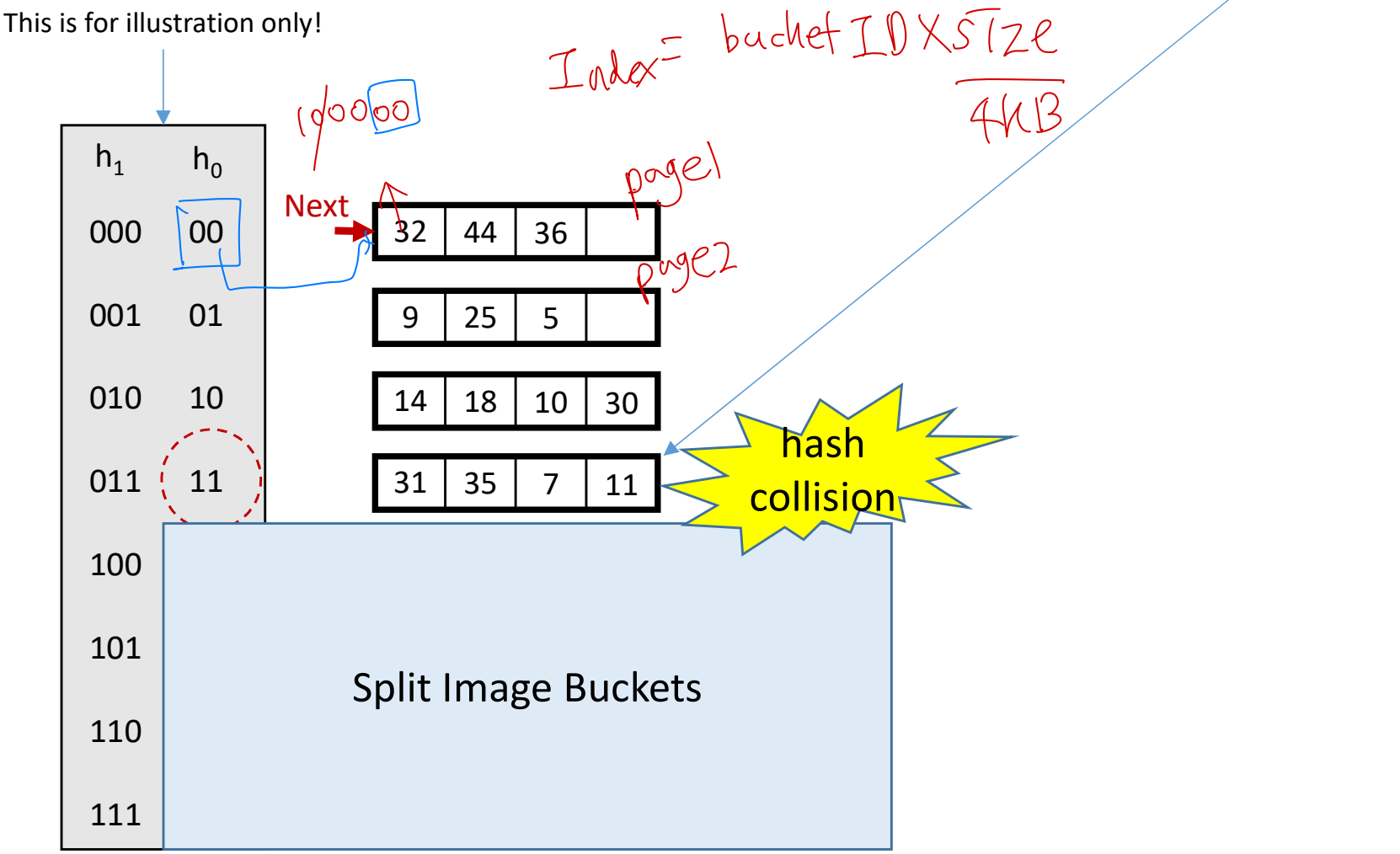
- Overview



# Linear Hashing – How it Works

- Insert 43 ($101011_{(2)}$) (Level 0, N=4)

$43\%N = 11_{(2)}$

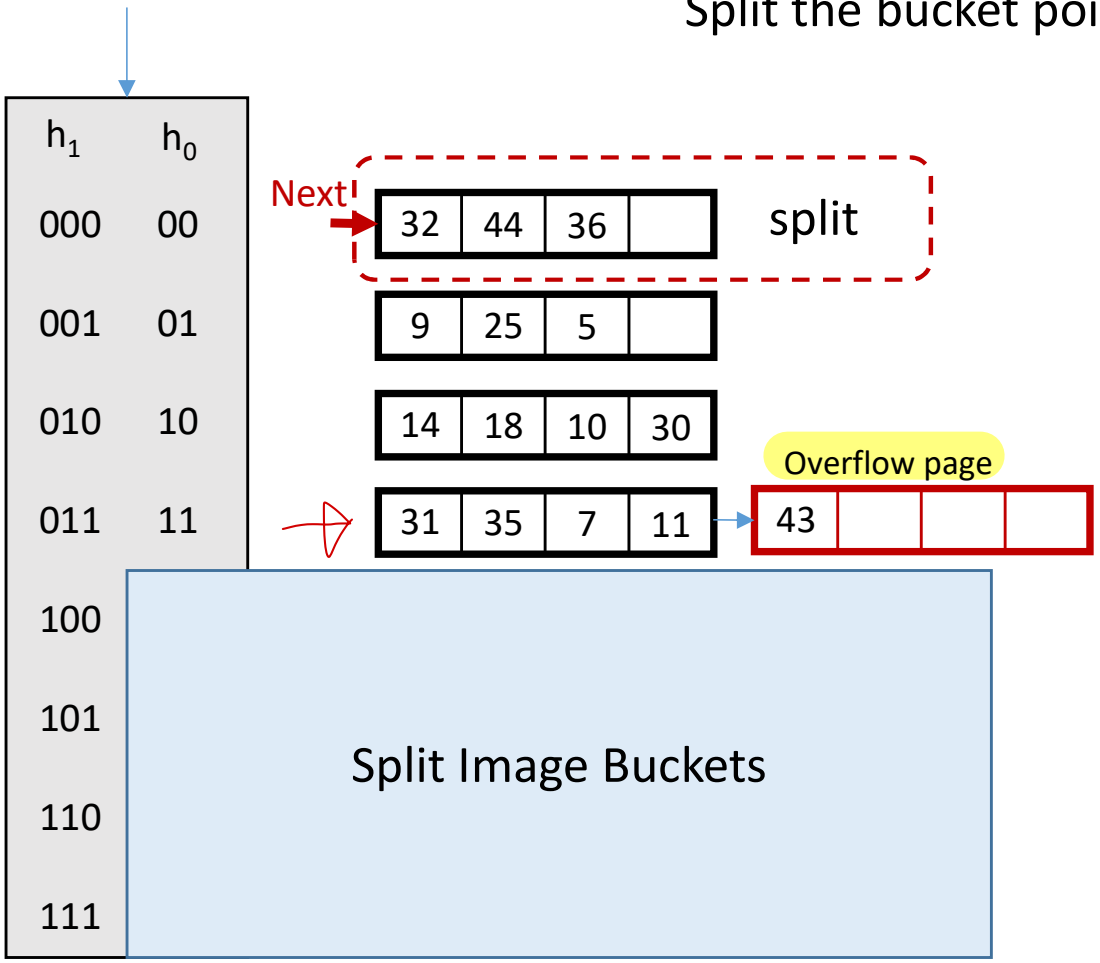


# Linear Hashing – How it Works

- Insert 43 ($101011_{(2)}$) (Level 0, N=4)  $43\%N = 11_{(2)}$

This is for illustration only!

Split the bucket pointed by Next



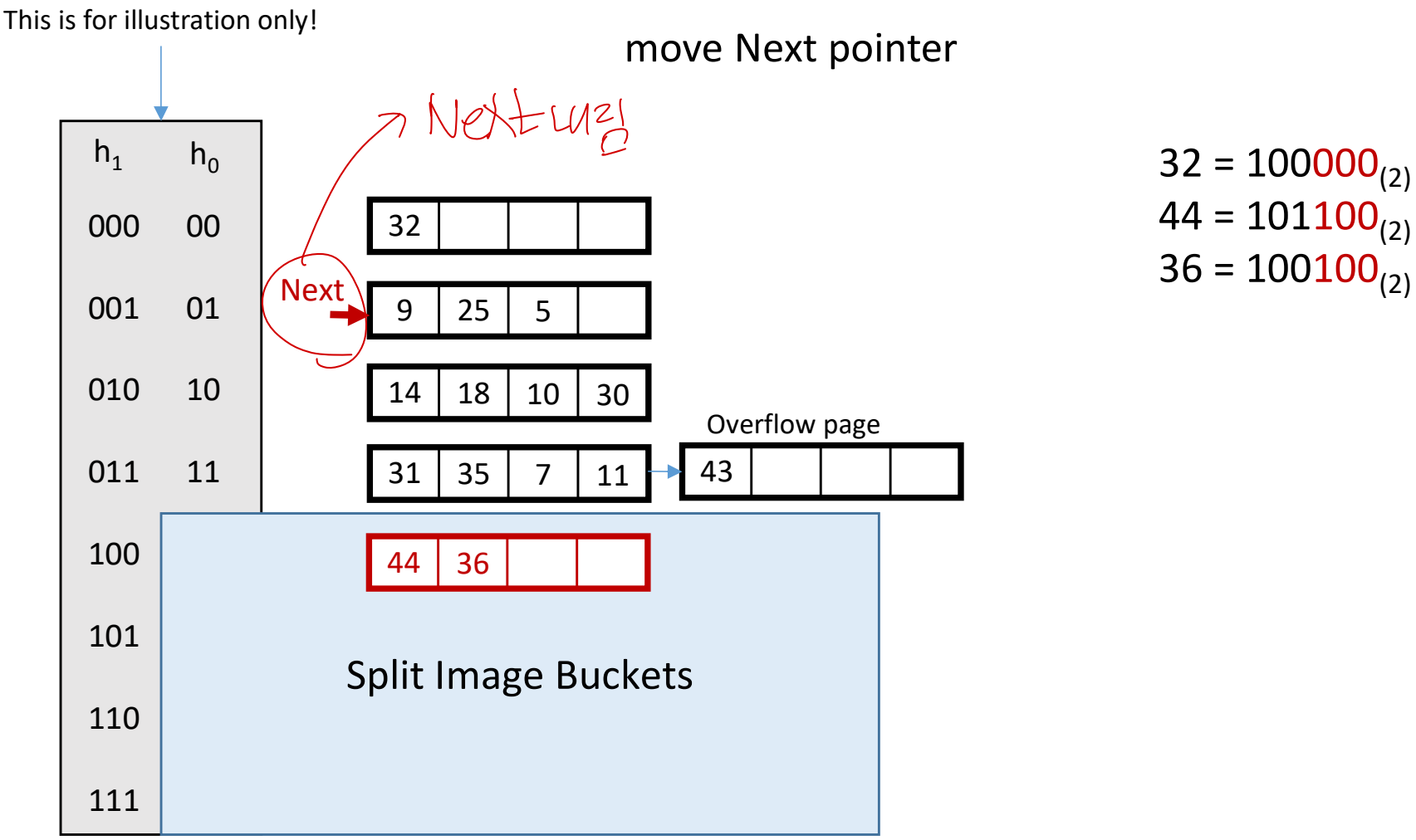

$32 = 100000_{(2)}$
$44 = 101100_{(2)}$
$36 = 100100_{(2)}$

collision 발생 → Next pointer 가리키는 버켓 split

# Linear Hashing – How it Works

- Insert 43 ($101011_{(2)}$) (Level 0, N=4)

$43\%N = 11_{(2)}$

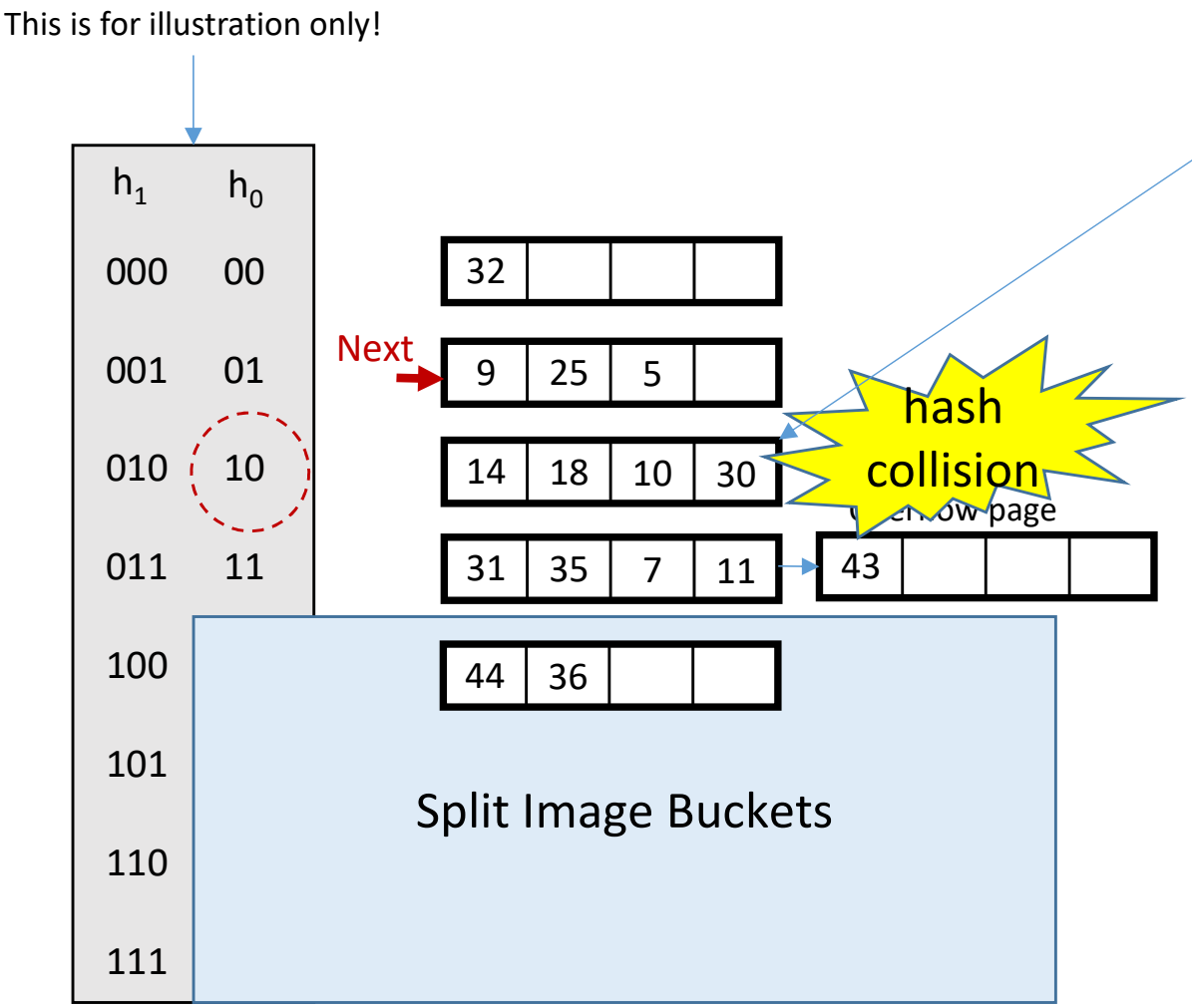



$32 = 100000_{(2)}$
$44 = 101100_{(2)}$
$36 = 100100_{(2)}$

# Linear Hashing – How it Works

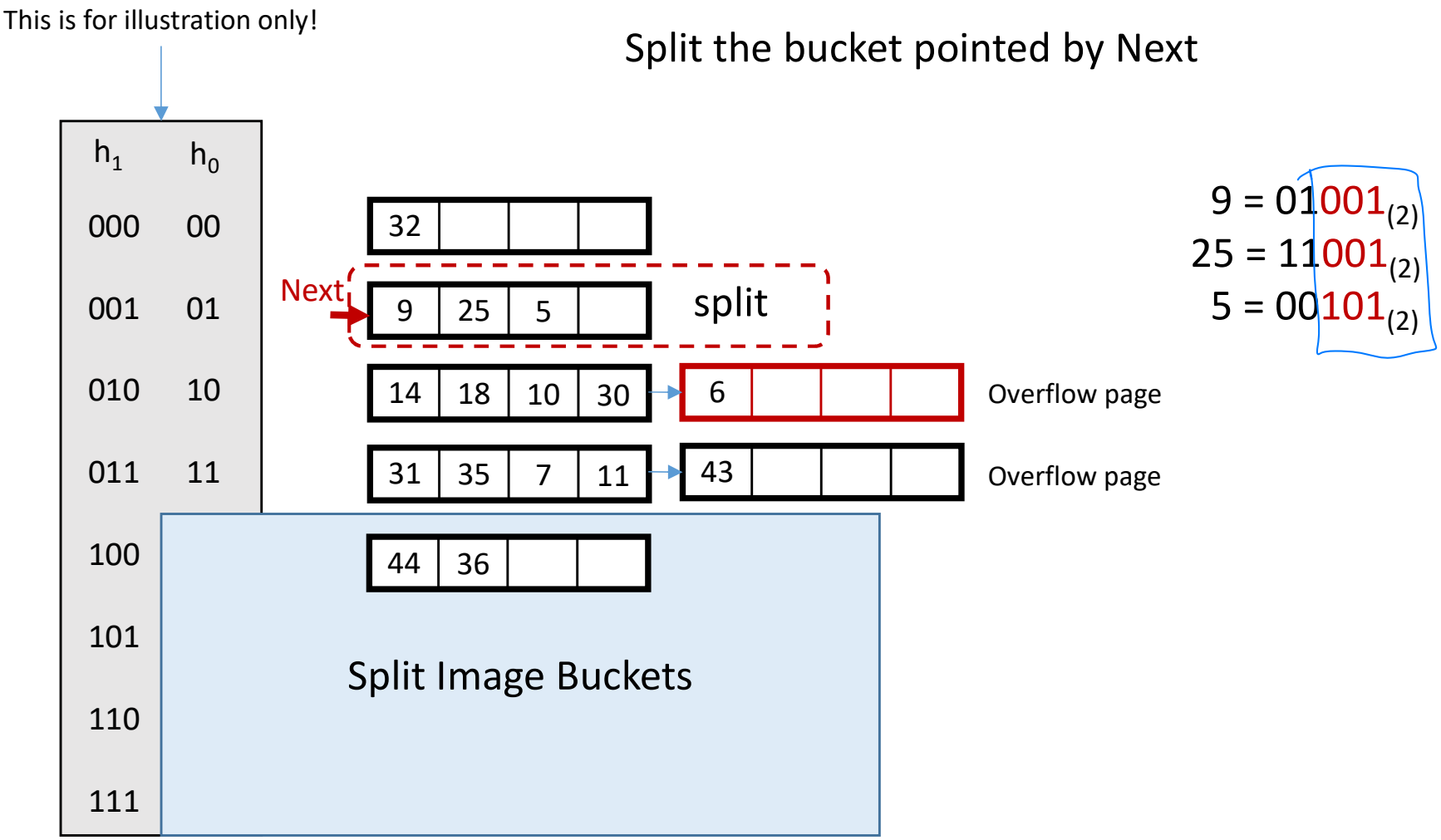
- Insert 6 ($000110_{(2)}$) (Level 0, N=4)

$6\%N = 10_{(2)}$



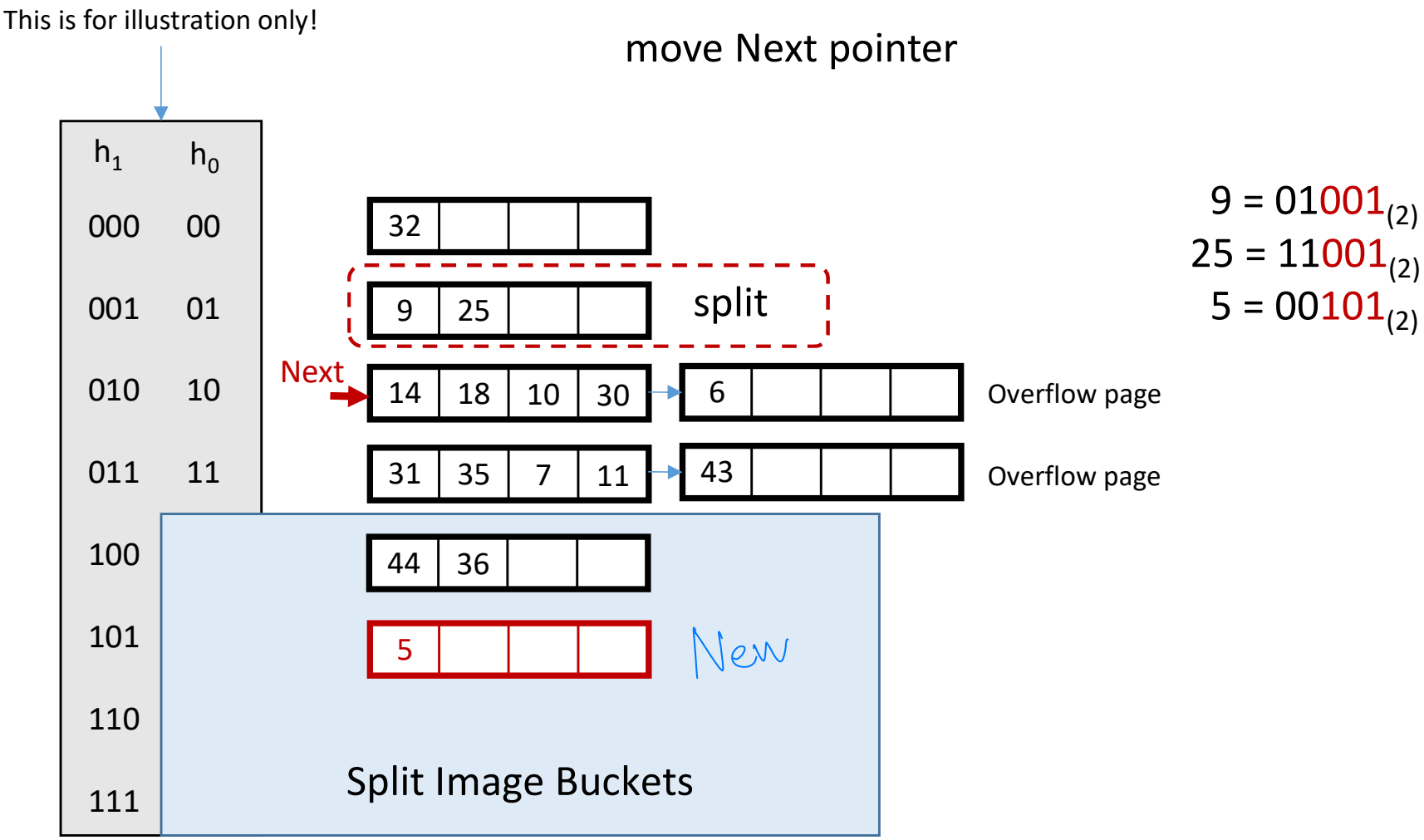
# Linear Hashing – How it Works

- Insert 6 ($000110_{(2)}$) (Level 0, N=4)  $6\%N = 10_{(2)}$



# Linear Hashing – How it Works

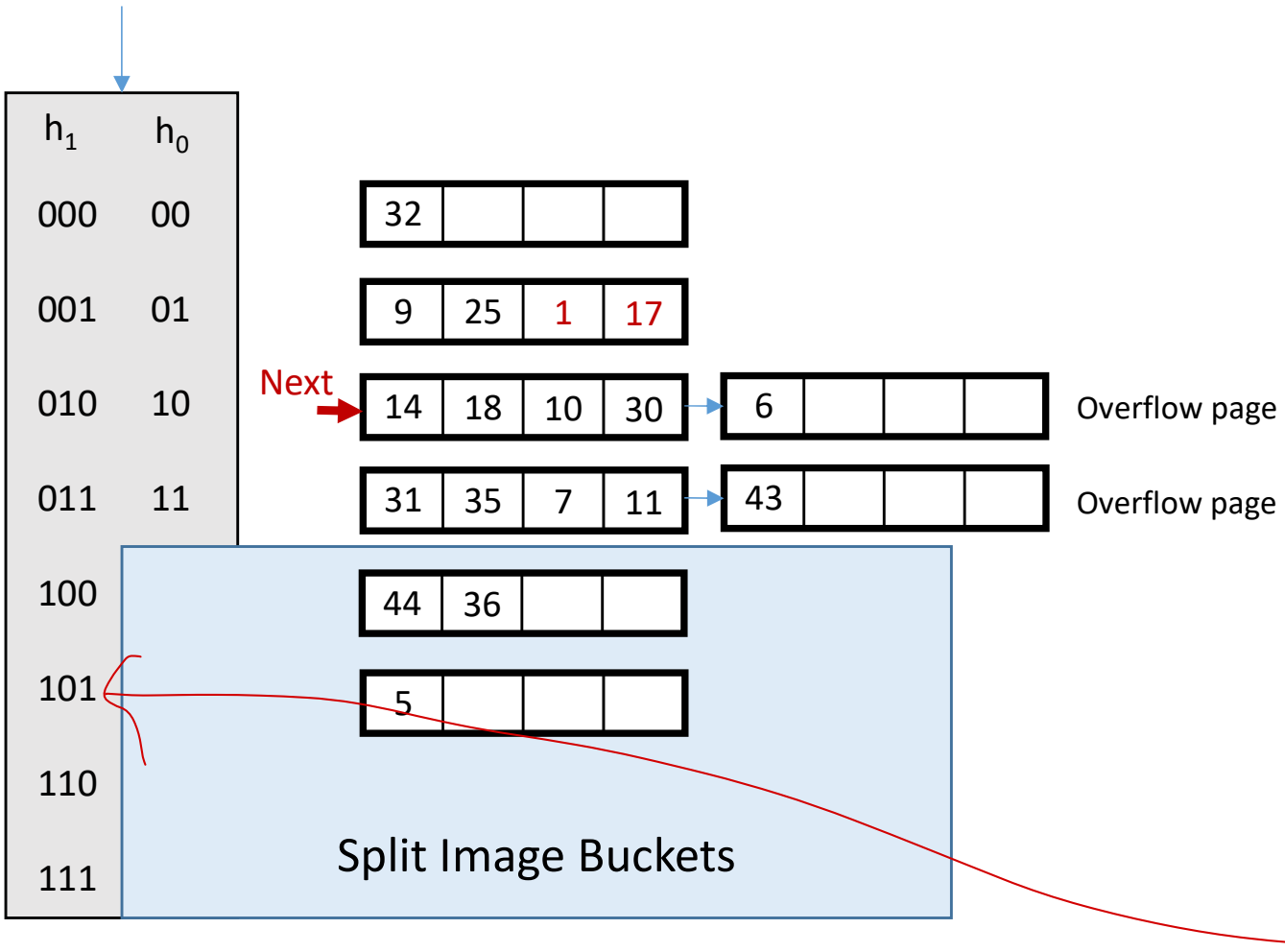
- Insert 6 ($000110_{(2)}$) (Level 0, N=4)  $6\%N = 10_{(2)}$



# Linear Hashing – How it Works

- Insert 1 & 17 ($000001_{(2)}$, $010001_{(2)}$) (Level , N=4)  $1\%N = 01_{(2)}$

This is for illustration only!




25 = 11001 → 바뀜

Next=10

2bit확인 01 < 10

→ previous bucketor

3bit을확인해서

→ New(split된)버킷을

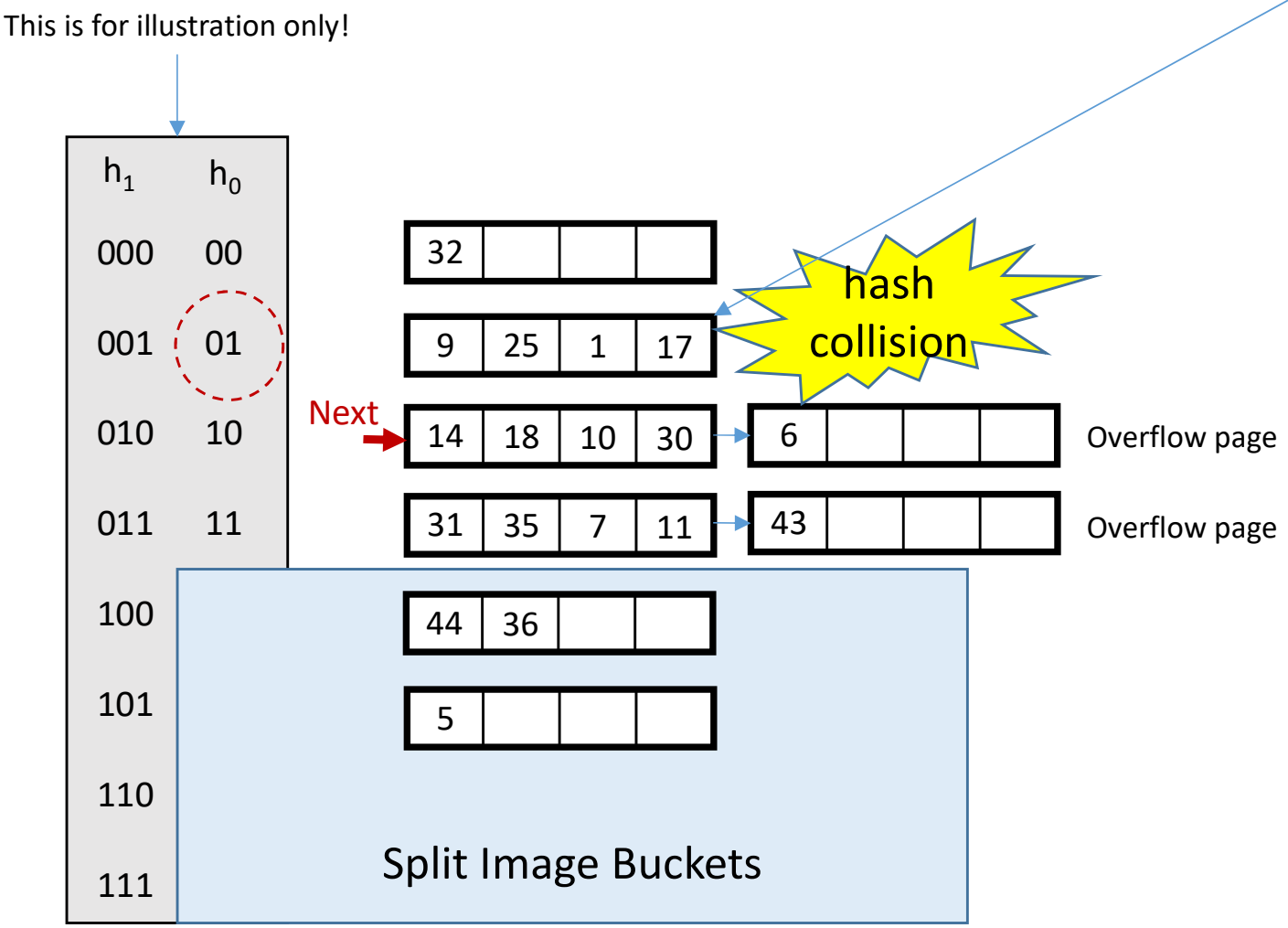
찾아야함

29 = 11101

# Linear Hashing – How it Works

- Insert 33 ($100001_{(2)}$) (Level 0, N=4)

$33\%N = 01_{(2)}$

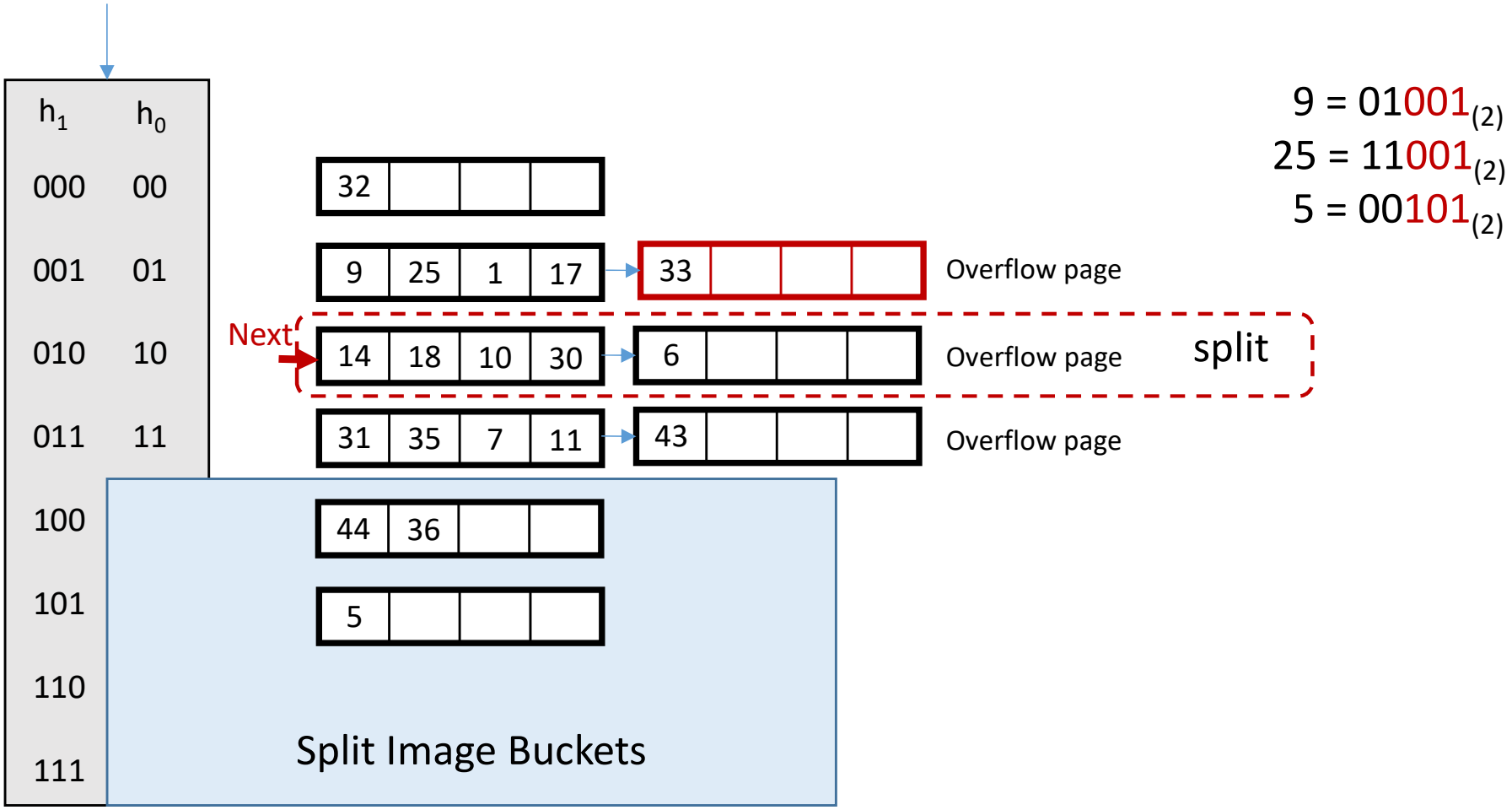


# Linear Hashing – How it Works

- Insert 33 ($100001_{(2)}$) (Level 0, N=4)

$33\%N = 01_{(2)}$

This is for illustration only!




$9 = 01001_{(2)}$

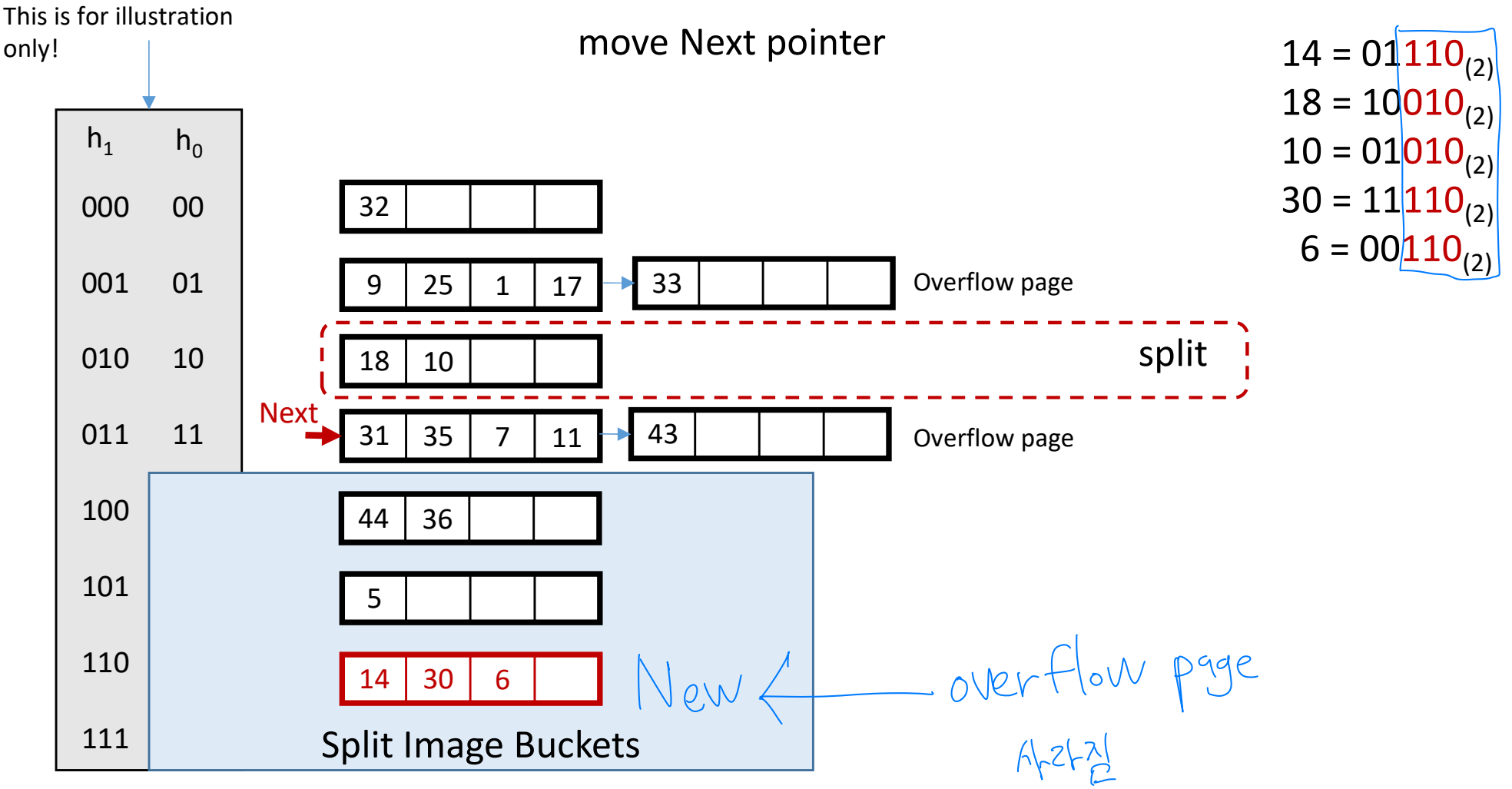
$25 = 11001_{(2)}$

$5 = 00101_{(2)}$

# Linear Hashing – How it Works

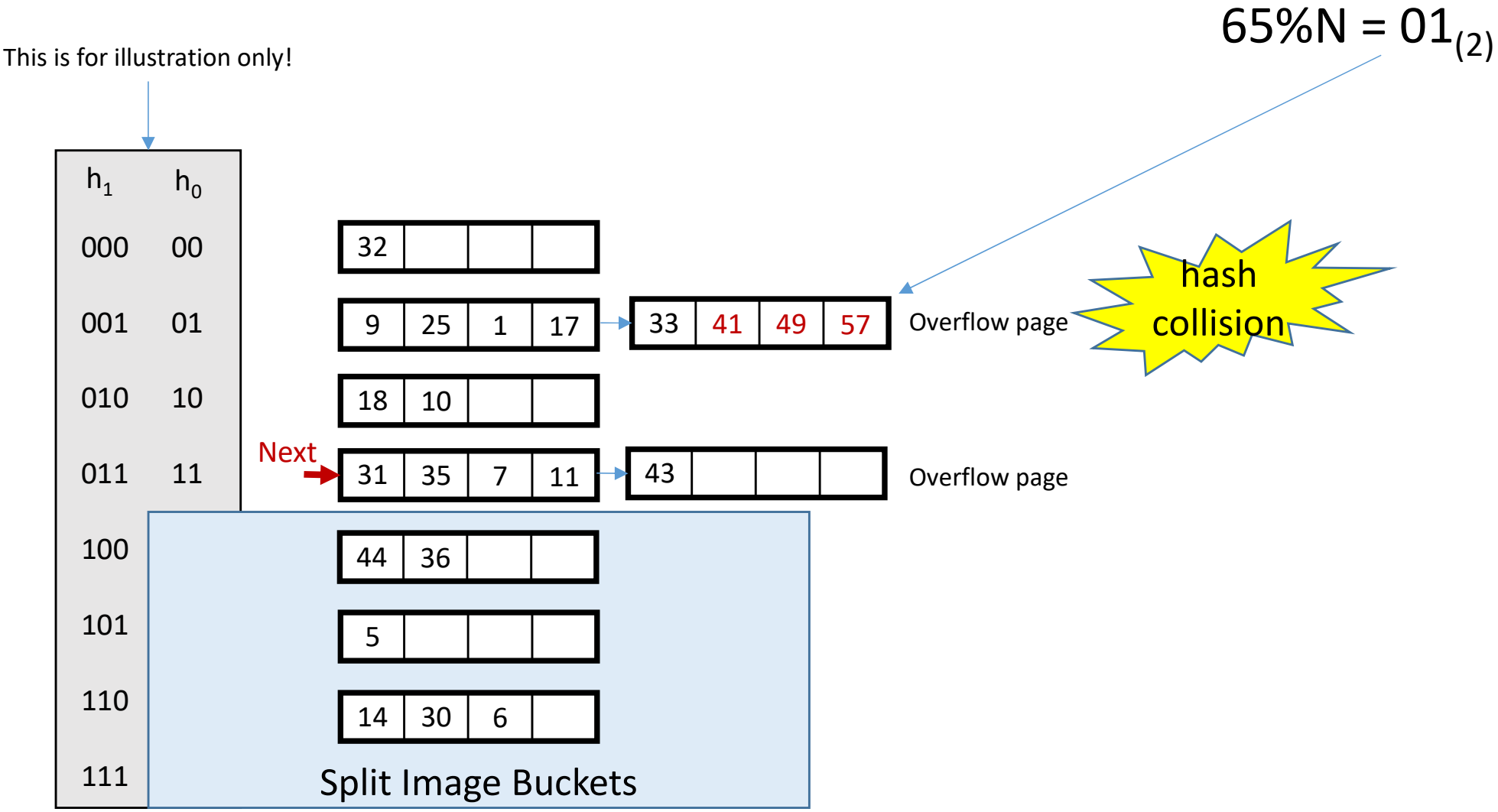
- Insert 33 ($100001_{(2)}$) (Level 0, N=4)

$33\%N = 01_{(2)}$



# Linear Hashing – How it Works

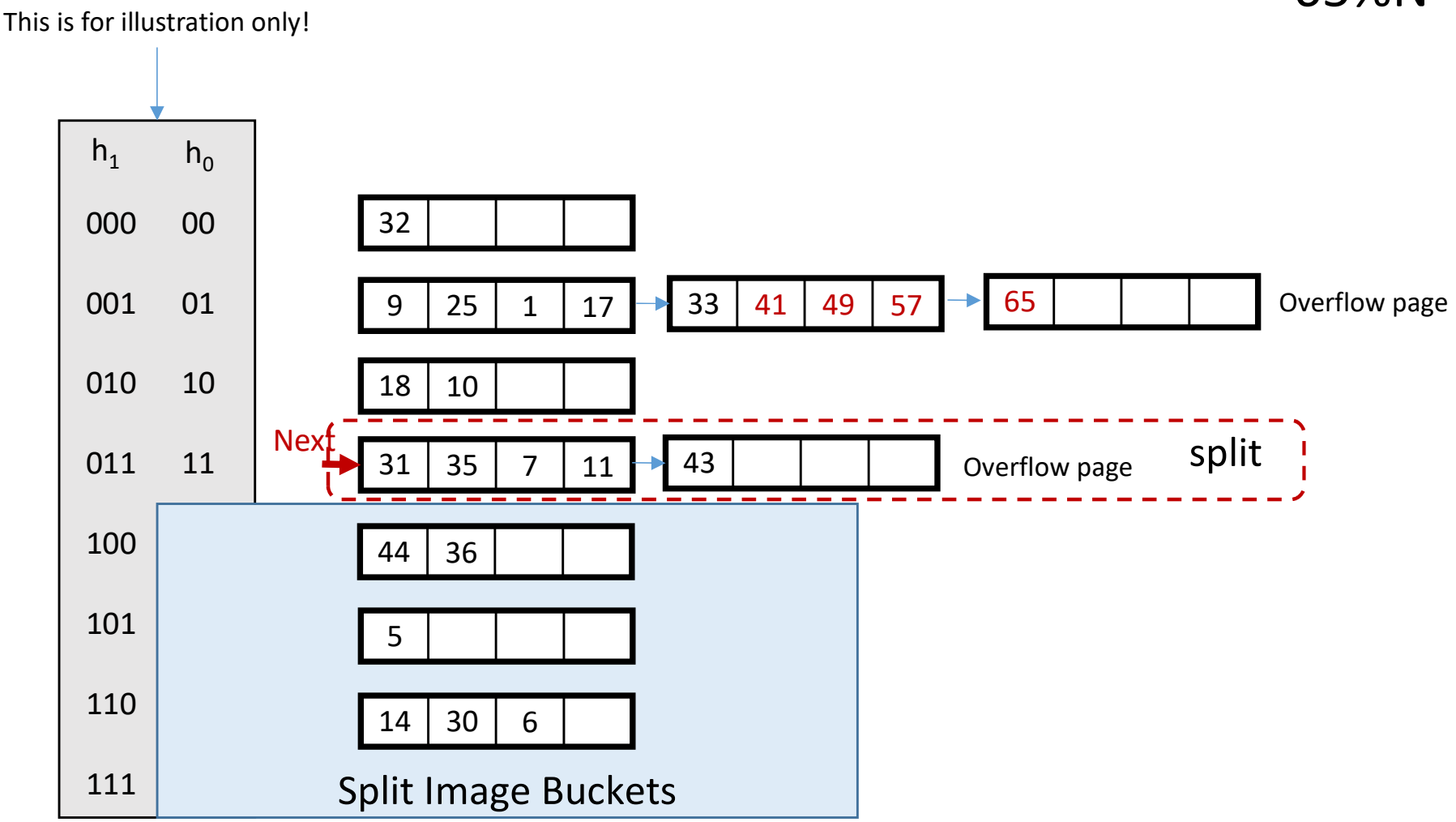
- Insert 41, 49, 57, 65 ($101001_{(2)}$, $110001_{(2)}$, $111001_{(2)}$, $1000001_{(2)}$)



# Linear Hashing – How it Works

- Insert 41, 49, 57, 65 ($101001_{(2)}$, $110001_{(2)}$, $111001_{(2)}$, $1000001_{(2)}$)

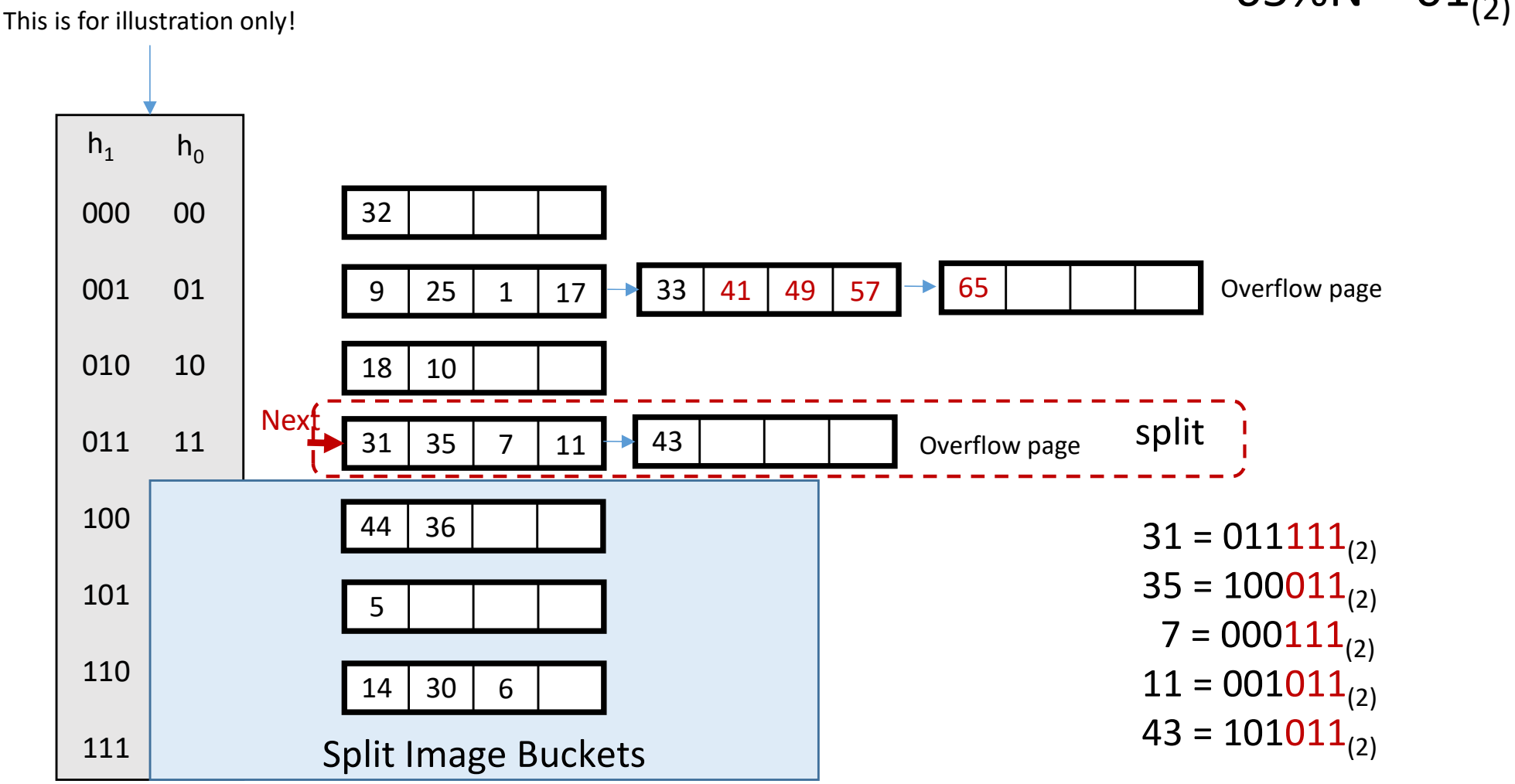
$65\%N = 01_{(2)}$



# Linear Hashing – How it Works

- Insert 41, 49, 57, 65 ($101001_{(2)}$, $110001_{(2)}$, $111001_{(2)}$, $1000001_{(2)}$)

$65\%N = 01_{(2)}$

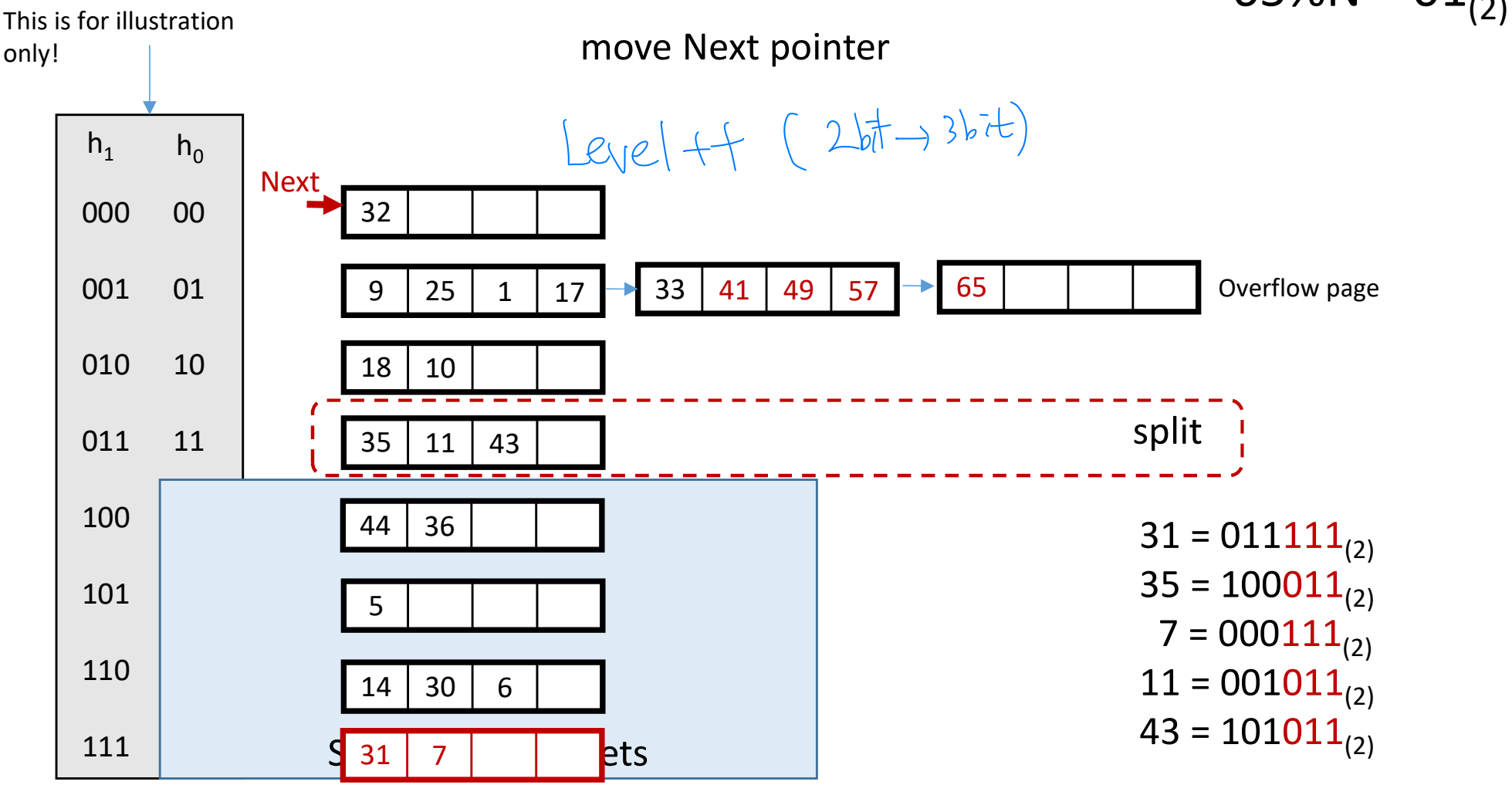



$31 = 011111_{(2)}$
$35 = 100011_{(2)}$
$7 = 000111_{(2)}$
$11 = 001011_{(2)}$
$43 = 101011_{(2)}$

# Linear Hashing – How it Works

- Insert 41, 49, 57, 65 ($101001_{(2)}$, $110001_{(2)}$, $111001_{(2)}$, $1000001_{(2)}$)

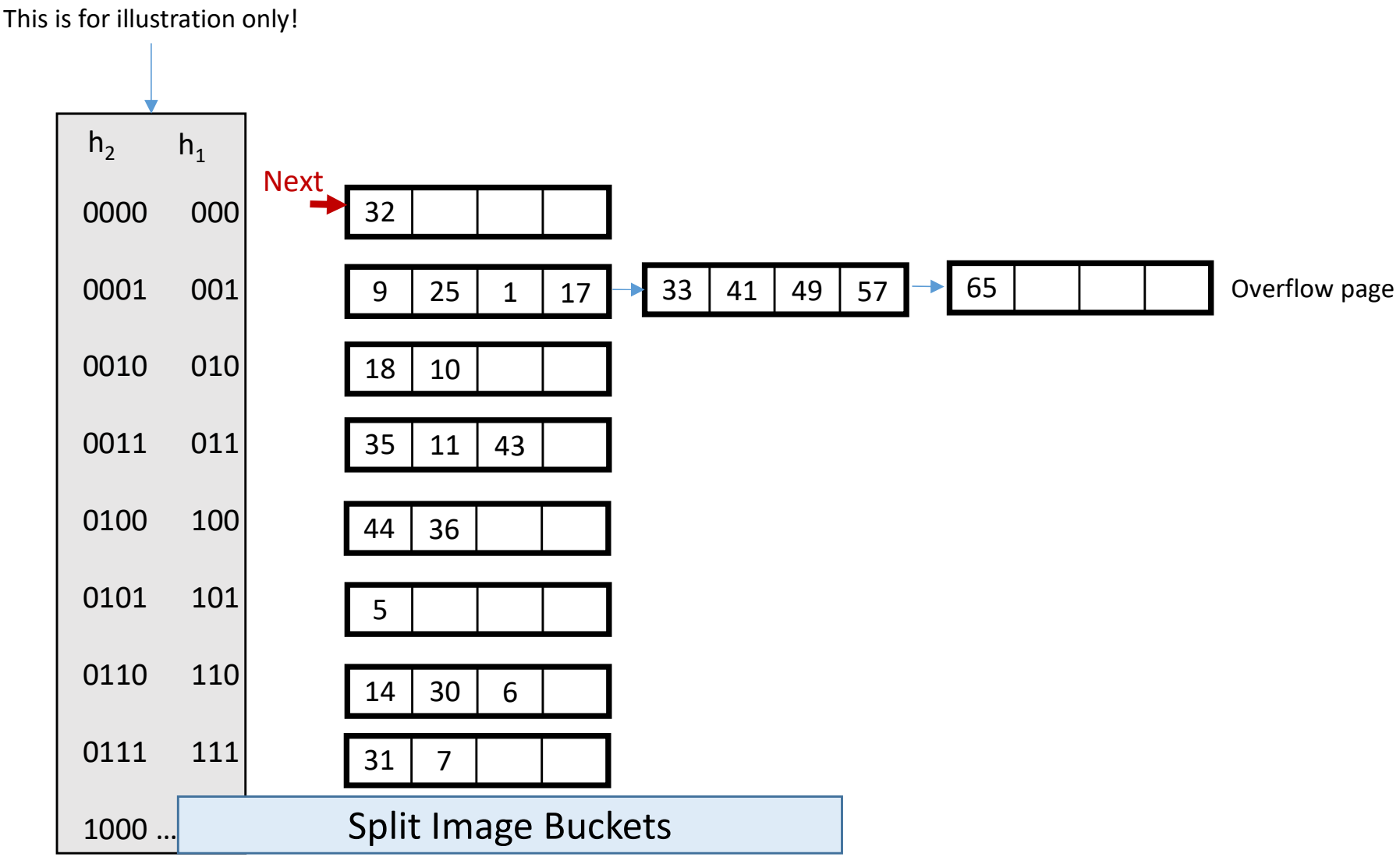
$65\%N = 01_{(2)}$




$31 = 011111_{(2)}$
$35 = 100011_{(2)}$
$7 = 000111_{(2)}$
$11 = 001011_{(2)}$
$43 = 101011_{(2)}$

# Linear Hashing – How it Works

- Level 1, N=8



# Linear Hashing Search Algorithm

- Compute $h_{Level}(key)$ to find a target bucket for key

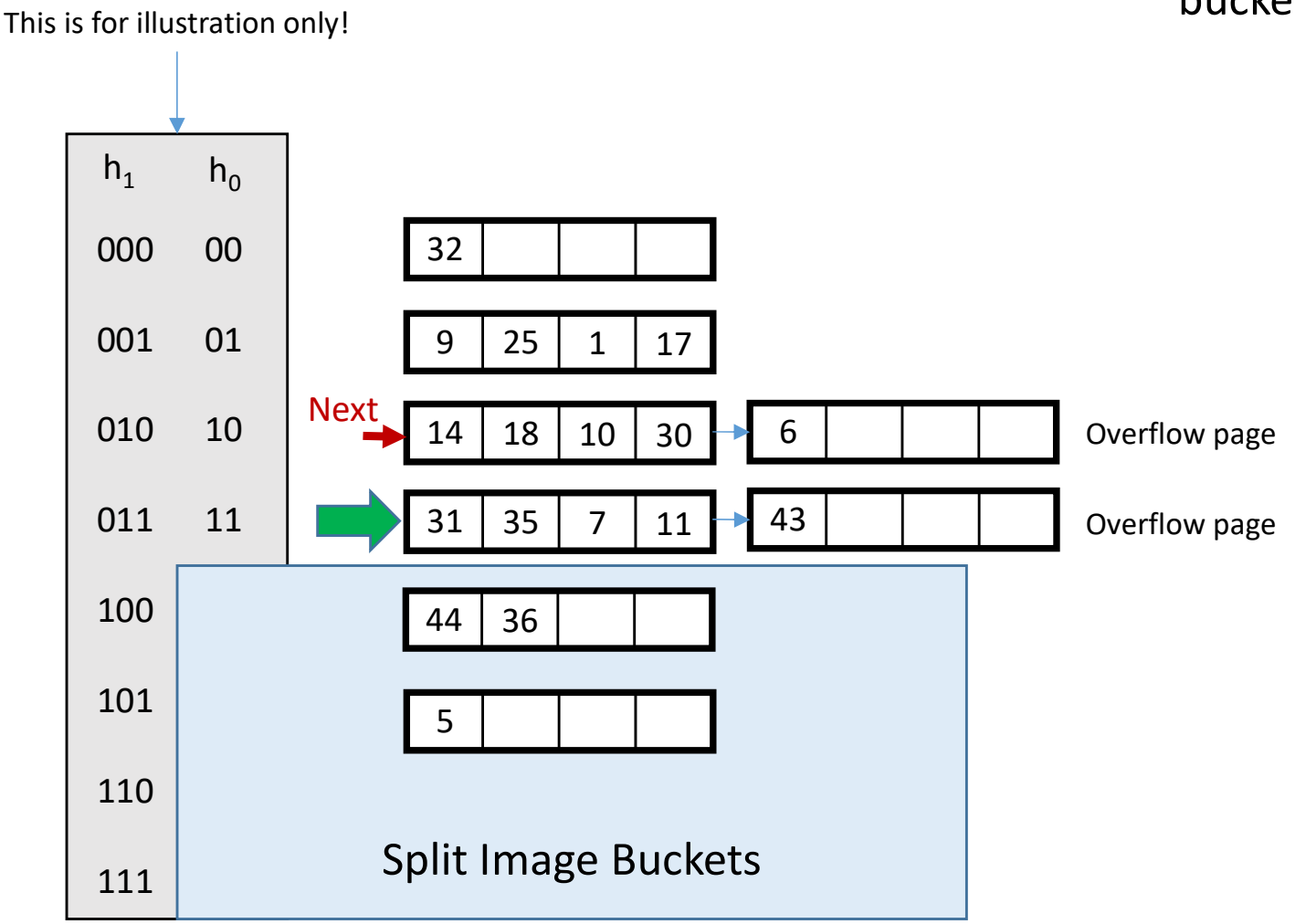
- If $\mathbf{h}_{Level}(r)$ >= Next → after
  - Bucket that has not been split this round
  - key belongs to that bucket for sure
- Else → previous (이미 스플릿됨) → additional bit 확인!
  2bit → 3bit
  - Buckets has already been split
  - Must recompute $\mathbf{h}_{Level+1}(r)$ or $\mathbf{h}_{Level}(r) + N_{Level}$
  - Key will go to either
    $\mathbf{h}_{Level}(r)$ or
    $\mathbf{h}_{Level}(r) + N_{Level}$

# Linear Hashing – How it Works

- Search 7 ($00111_{(2)}$) (Level , N=4)

$7\%N = 11_{(2)}$ >= Next

bucket $11_{(2)}$ has not split yet



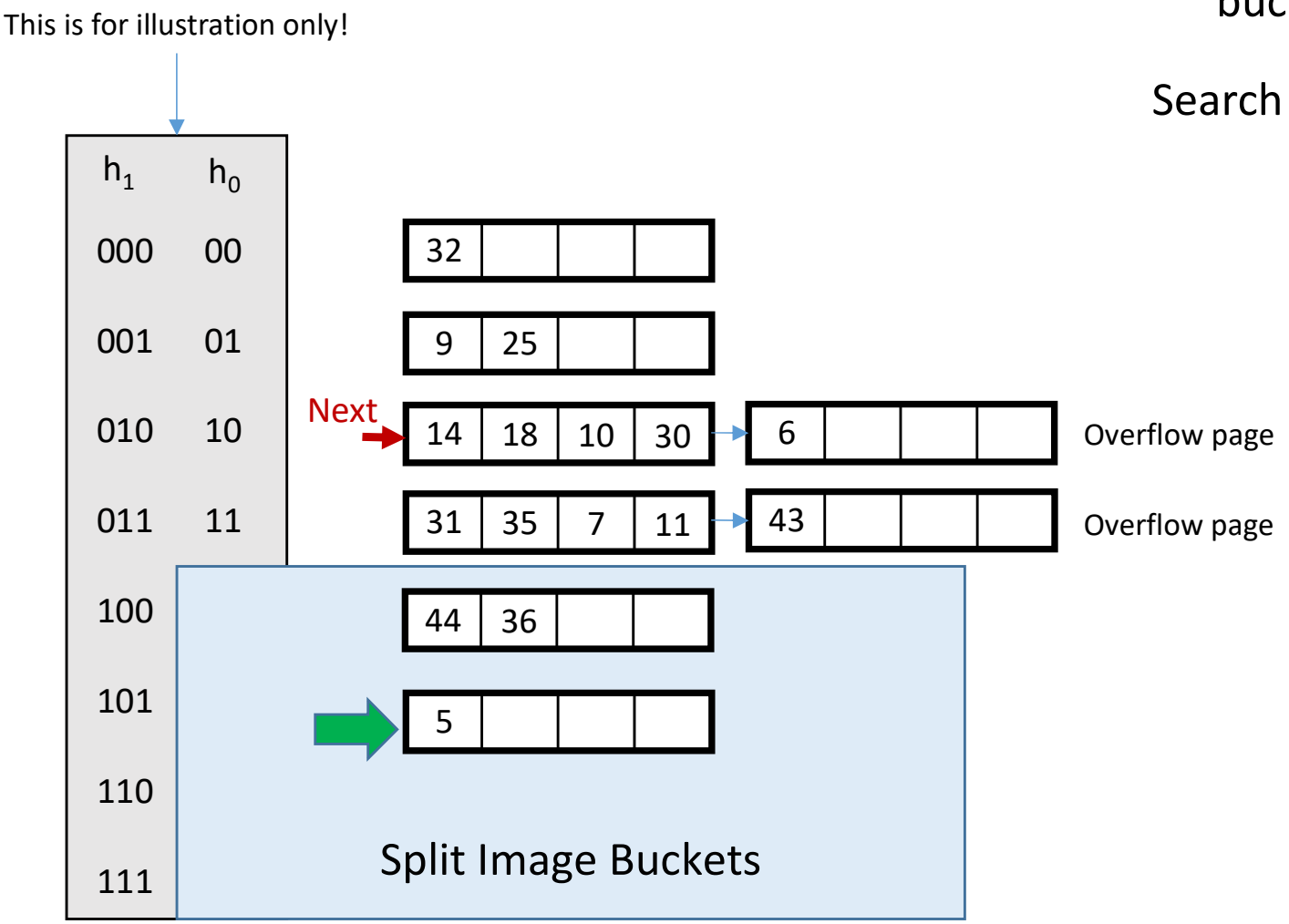
# Linear Hashing – How it Works

- Search 5 ($000101_{(2)}$) (Level , N=4)

$5\%N = 01_{(2)} < \text{Next}$

bucket $01_{(2)}$ has split

Search bucket $5\%(2^{level}N) = 101_{(2)}$



# Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better for point query
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B+-trees

# Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  **select** *ID*

  **from** *instructor*

  **where** *dept_name* = “Finance” **and** *salary* = 80000

Index → 1번째 조건 만족 체크 → 2번째 체크

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
  3. Take intersection of both sets

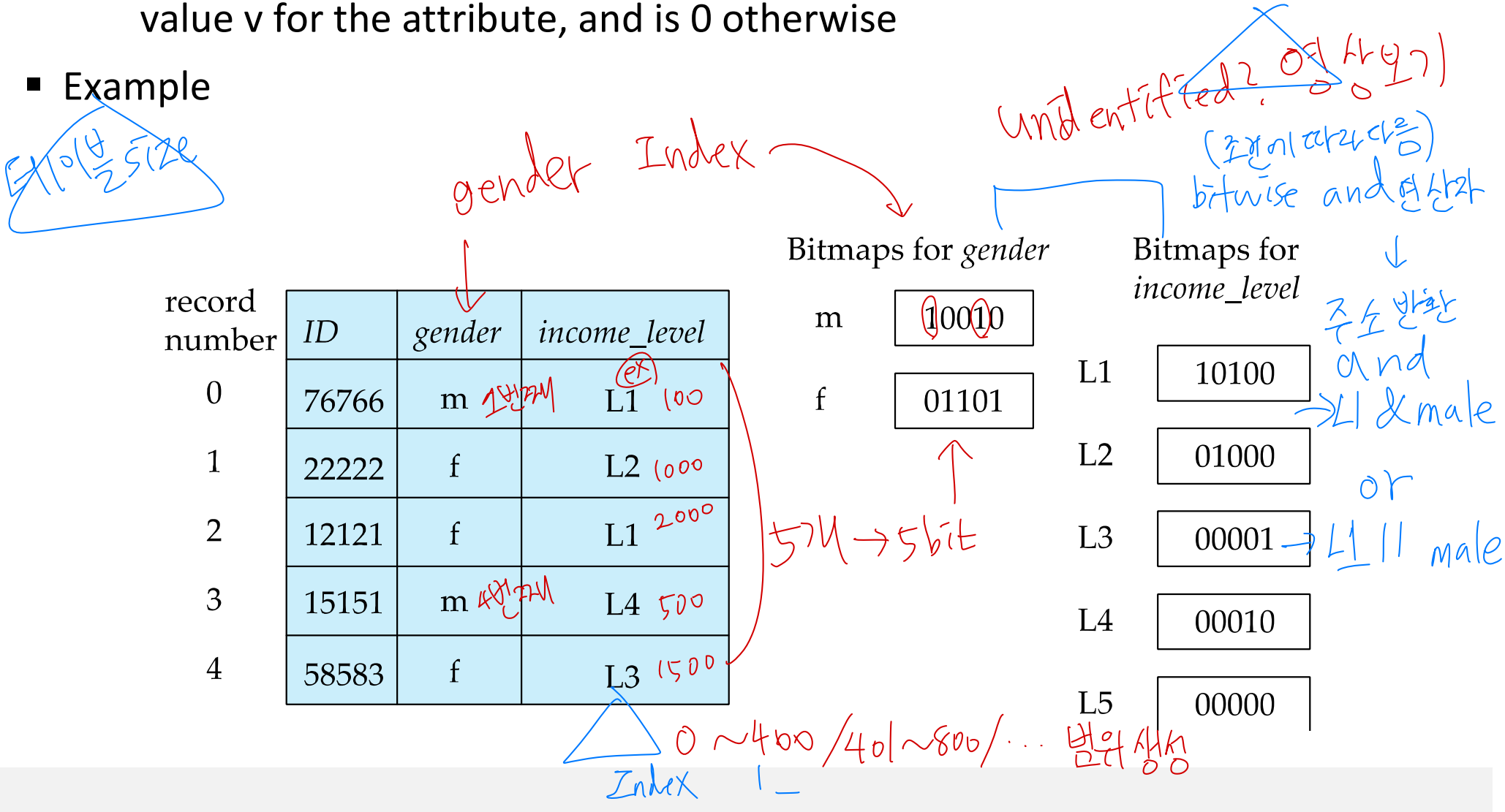
고정상

# Multiple-Key Access and Bitmap Index

- Bitmap indices are designed for efficient querying on multiple keys
- Records are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

# Bitmap Index

- a bitmap index has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- Example



# Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

bitwise 연산

# Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g., if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size

# Creation of Indices

- Example

  **create index** *takes_pk*

  **on** *takes* (*ID,course_ID, year, semester, section*)

  **drop index** *takes_pk*

- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
- Some database also create indices on foreign key attributes
  - Index on foreign key would be useful for this query:
    - *takes* ⋈ $\sigma_{name='Shankar'}$ (*student*)
- Indices can greatly speed up lookups, but impose cost on updates