Hash-Based Indexes

Chapter 11

Index Design Space

Organization Structure for k*

- Tree-based

  (+) Range, equality search
- Hash-based

  (+) Equality search
  - Static hashing
  - Extensible hashing
  - Linear hashing

Data Entry (k*) Contents

1. Actual Data record

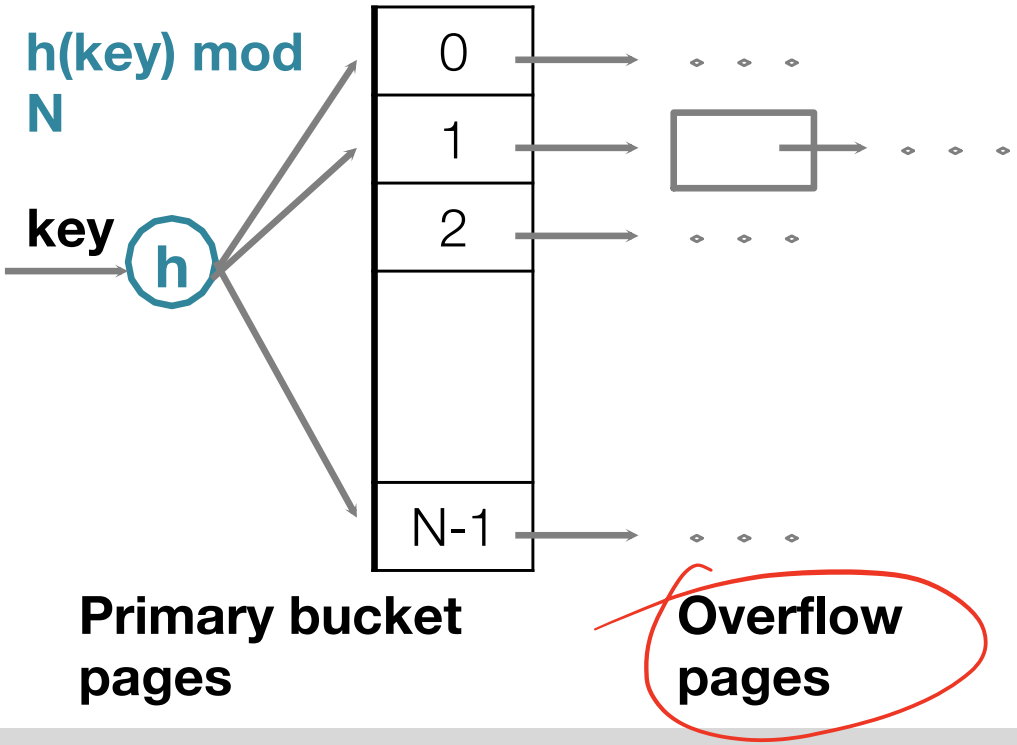
   index = file
2. <k, rid>

   actual records in a diff file
3. <k, list of rids>

Static Hashing

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

bucket size is a page



Question??

The maximum length of overflow chain with static hashing, for a data set of d elements is

A. 1

B. Log d

C. d ✓

D. Infinity (there is no maximum)

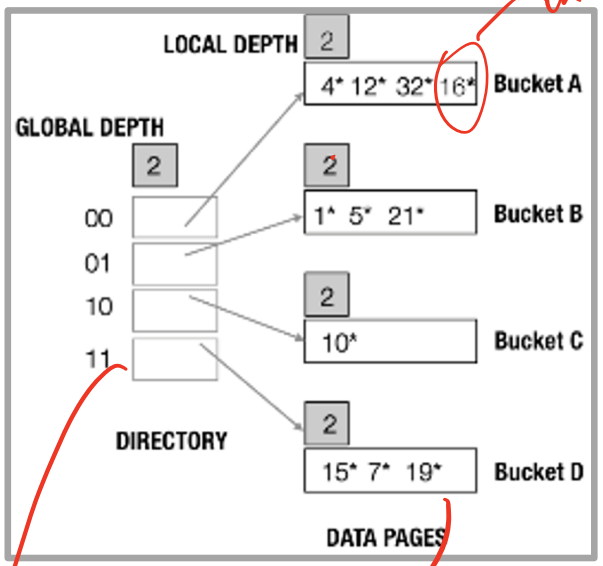
Static Hashing (Contd.)

- Static hashing with fixed # N of buckets can be problematic
  - Long overflow chains can develop (and hurt performance!)
- Might consider periodically doubling N and "rehashing" file
  - Slow when that happens: Entire file has to be read and written
  - Index unavailable while rehashing
- **Dynamic hashing** fixes above problems. Two dynamic hashing techniques: ✗ double N without rehashing
  - Extensible hashing
  - Linear Hashing

Extensible Hashing

each bucket is one page in size

- Main Idea: Use a directory (array) of N = 2^d pointers to buckets, where *d* is called global depth d=2 here
- Each bucket has a local depth, initially equal to *d*
- *Invariant*: local depth ≤ global depth.
- Search for k:
  - Compute hash function h(k) — an integer
  - Take last global depth # bits of **h**(*k*) and use them to index into the directory to find the bucket and retrieve the data entry.



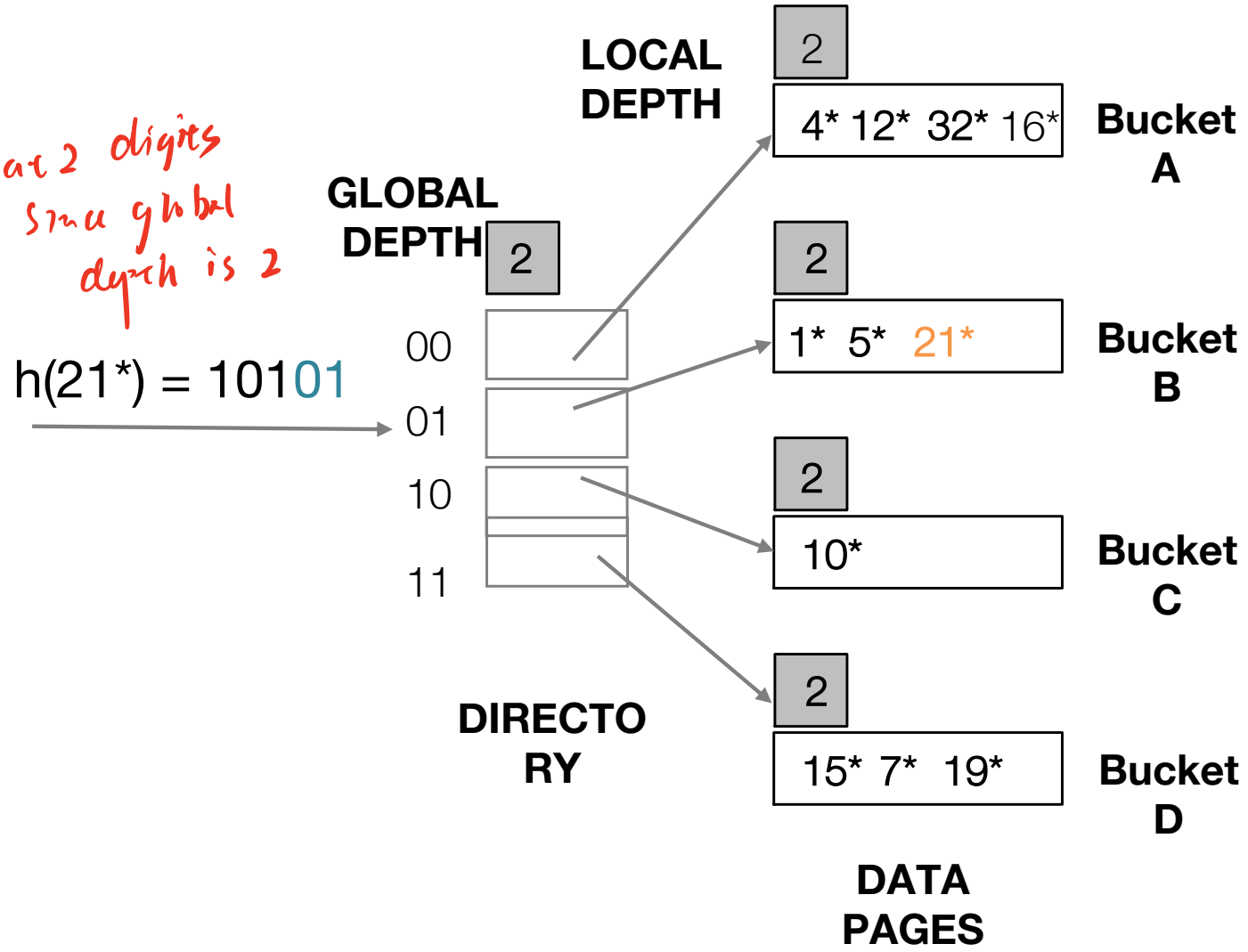
one bucket is a page

2^d buckets

Example: Search

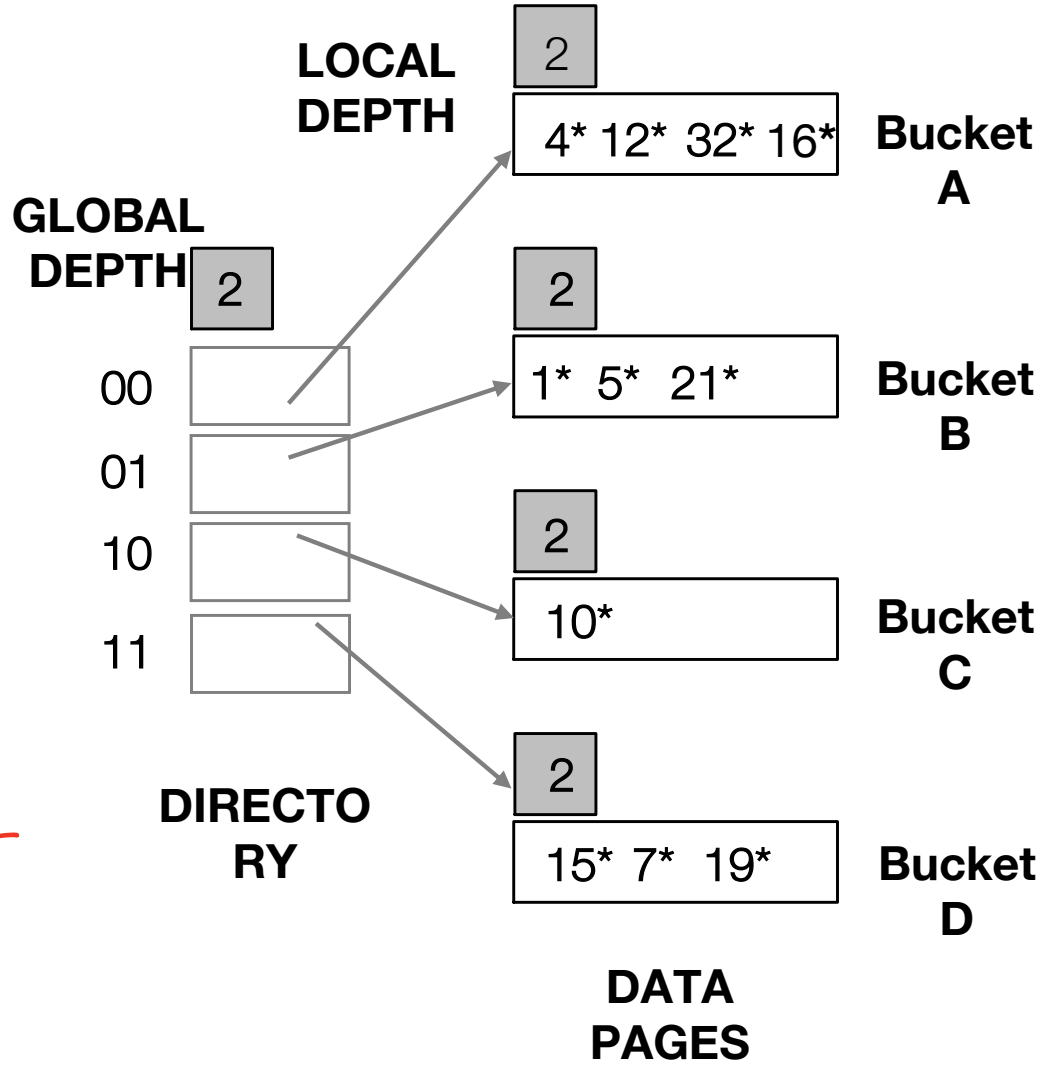
- Find 21*
- Suppose h(21*) = 10101

look at 2 digits since global depth is 2



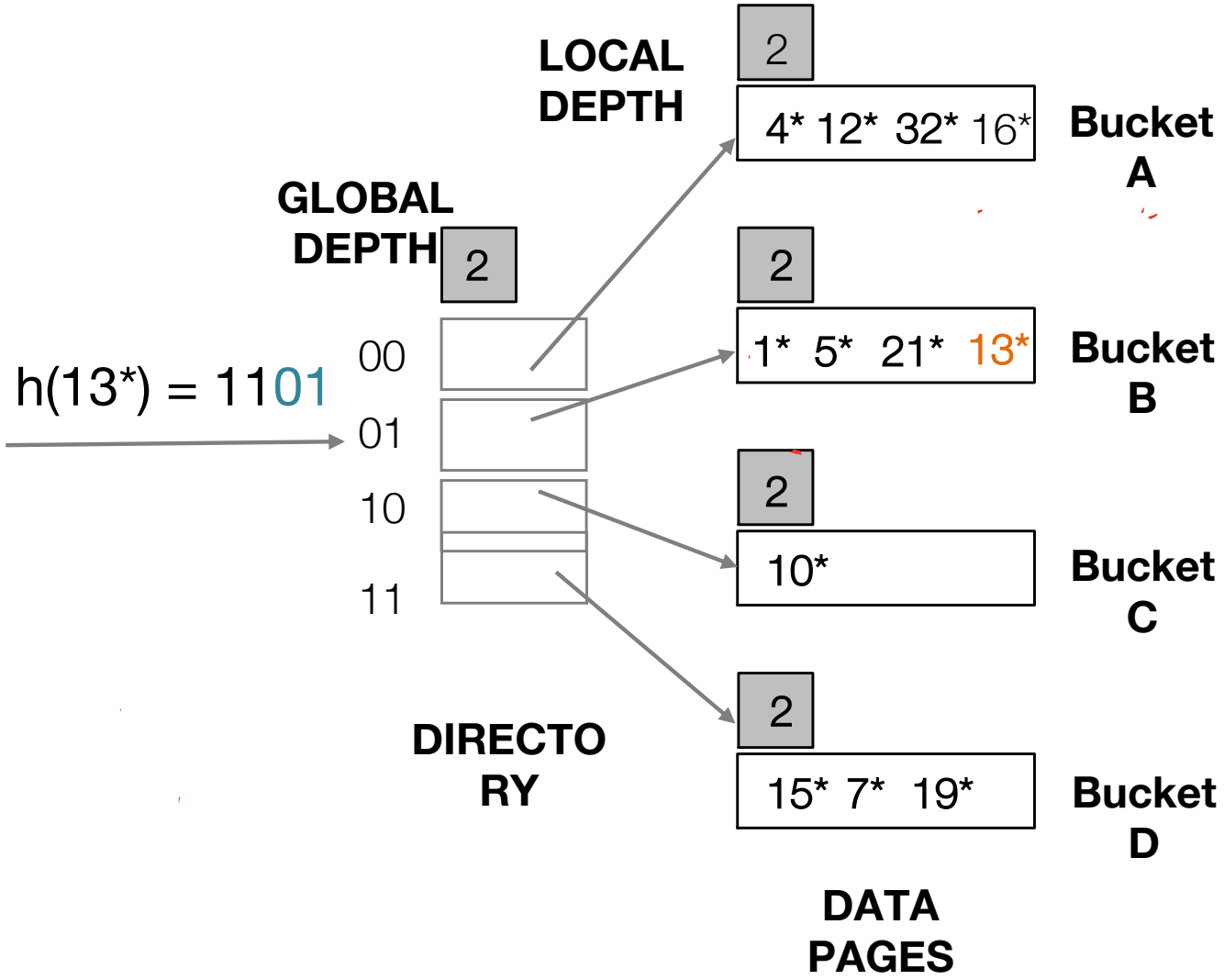
Extensible Hashing: Insert

- Insert k:
  - Take last global depth # bits of **h**(*k*)
  - If bucket has space, insert, done
  - If bucket if full:
    - Redistribute the values in the overflowed bucket into the two buckets using an extra bit from the index, incrementing their local depth.
    - If buckets' local depth exceeds global depth, restore invariant: double the directory size and increment global depth. Fix pointers.

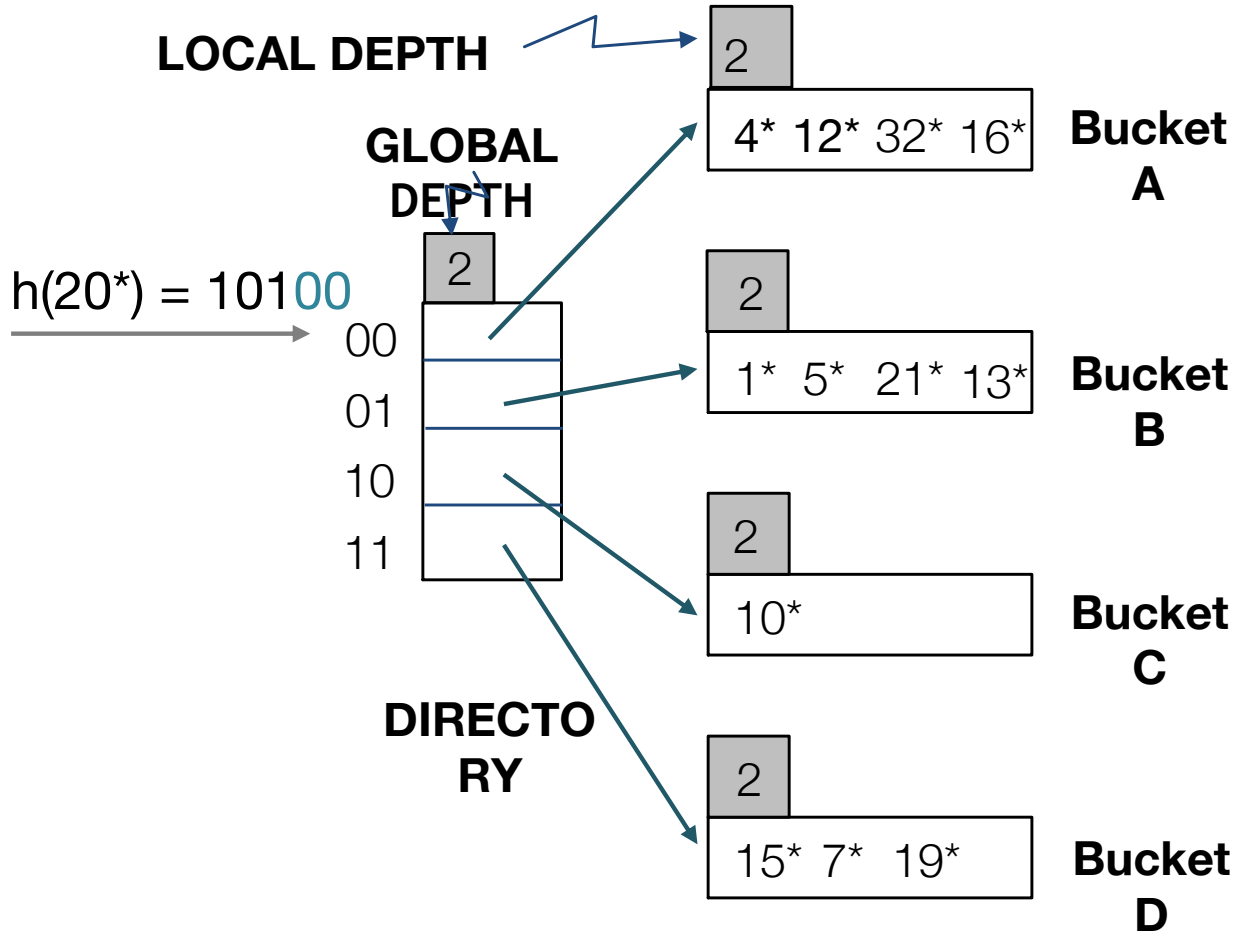


Example: Insertion into a free bucket

- Insert 13*
- Suppose
  h(13*) = 1101



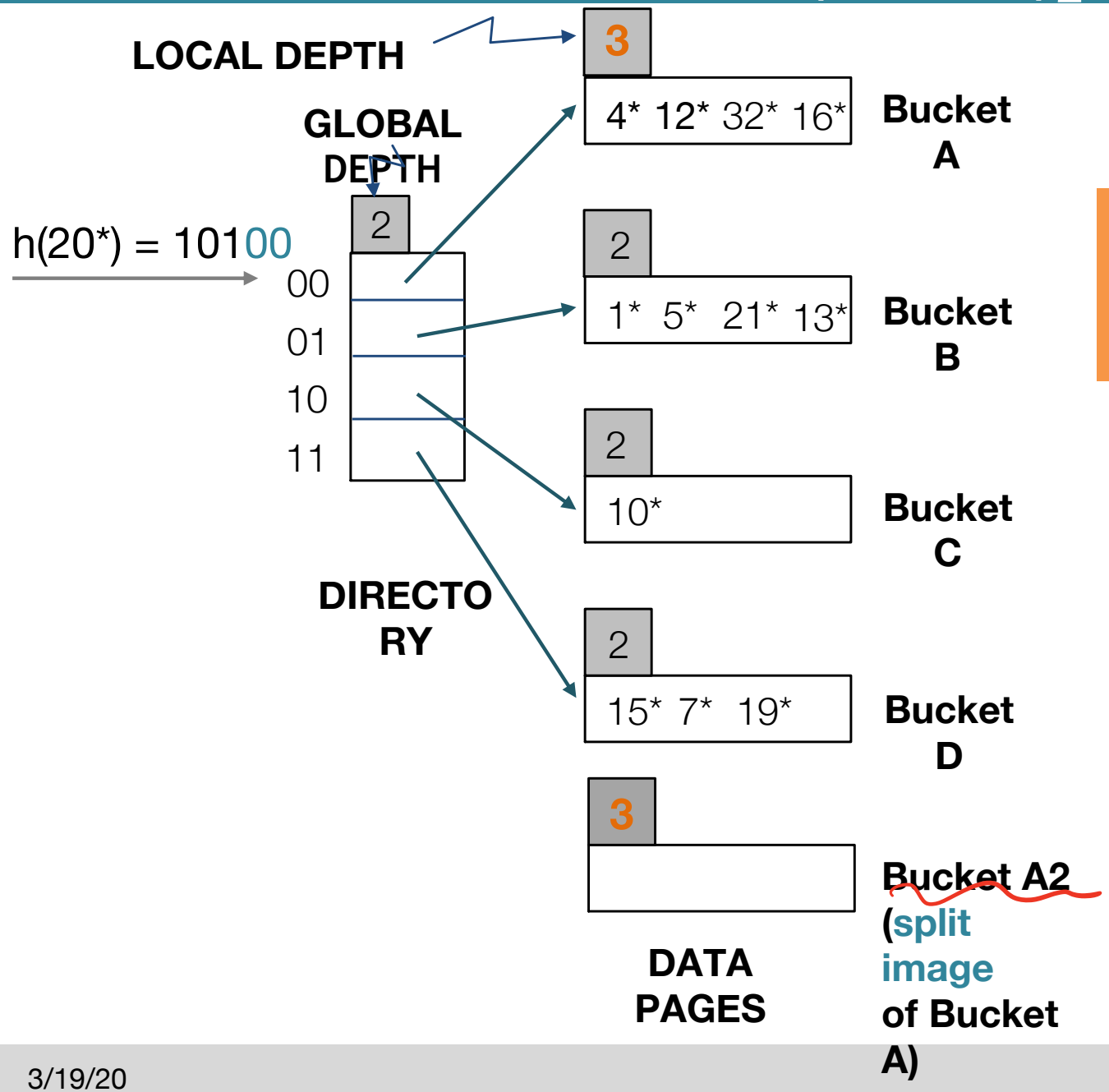
Example: Insertion into a full bucket
Insert 20 = $(10100)_2$



Bucket A is full! No space.

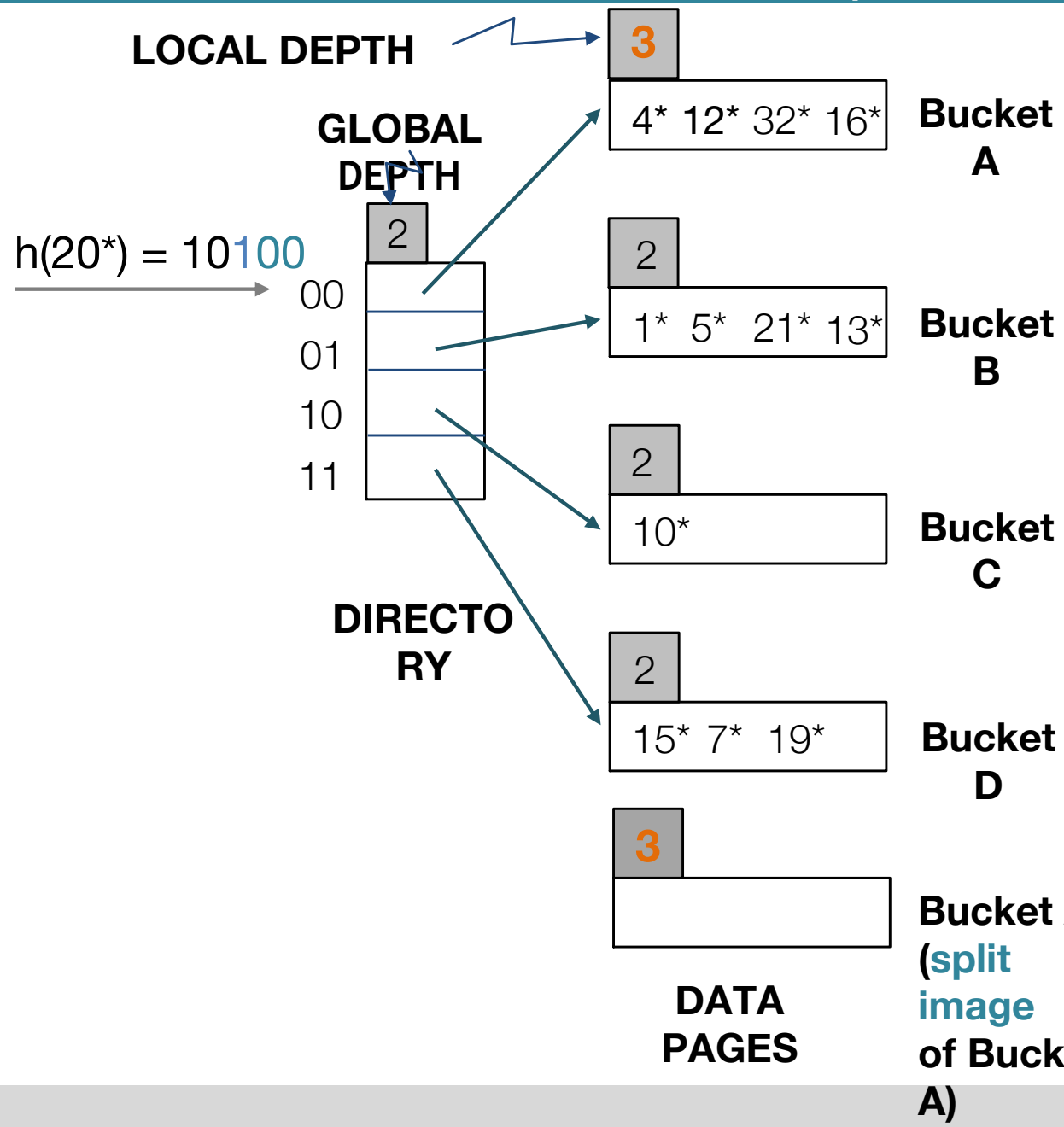
Bucket A is full! No space. Next step: Split Bucket A, incrementing its local depth

Example: Insertion into a full bucket
Insert 20 = $(10100)_2$



Split full bucket, Increment local depth

Example: Insertion into a full bucket
Insert 20 = $(10100)_2$

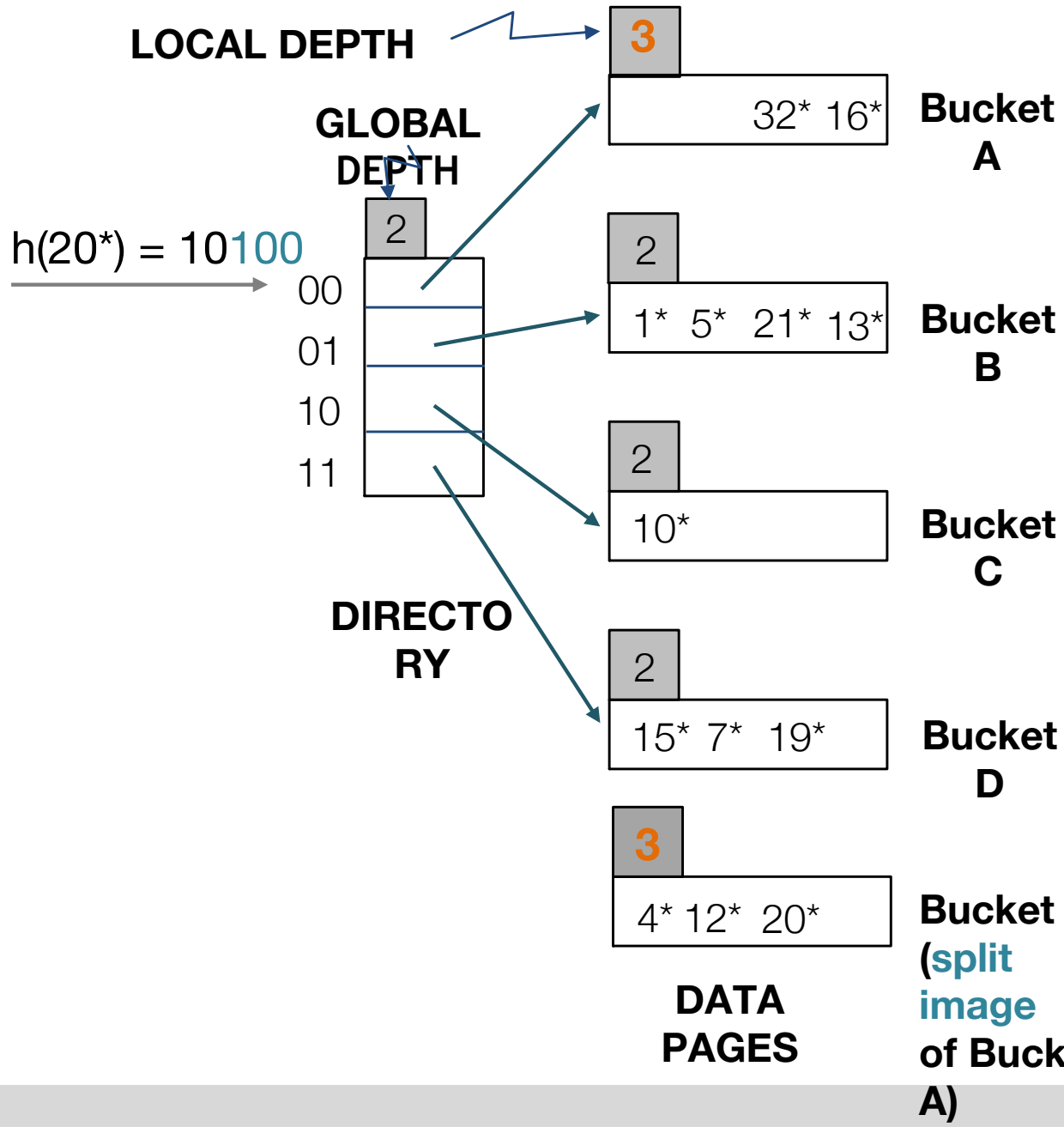


$32=(100000)_2$
$16=(10000)_2$

Redistribute entries of full backet and inserted value using the new local depth

$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Example: Insertion into a full bucket
Insert 20 = $(10100)_2$

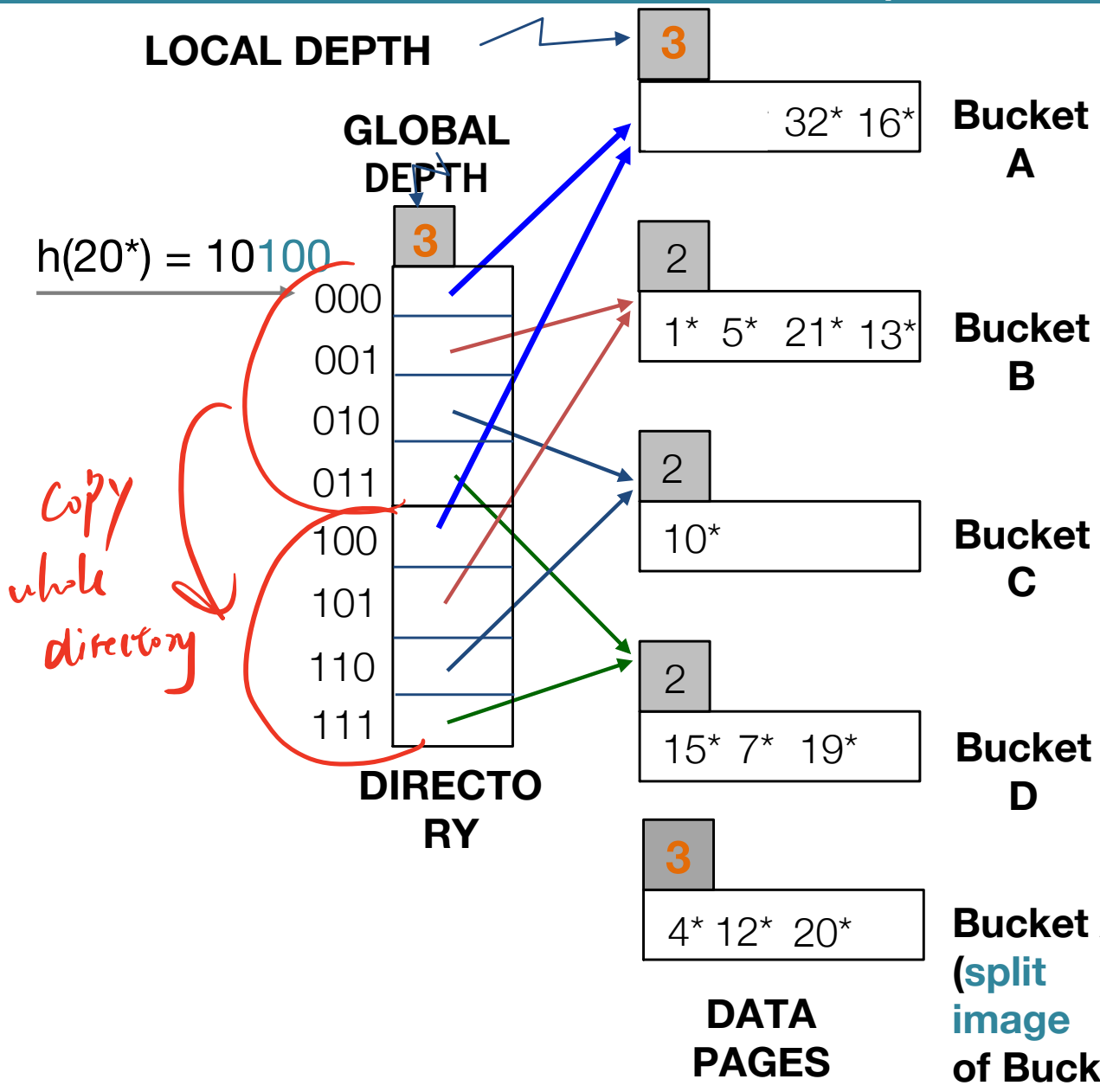


$32=(100000)_2$
$16=(10000)_2$

Values redistributed. But, invariant is violated! Fix that by doubling the directory

$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Example: Insertion into a full bucket
Insert 20 = $(10100)_2$



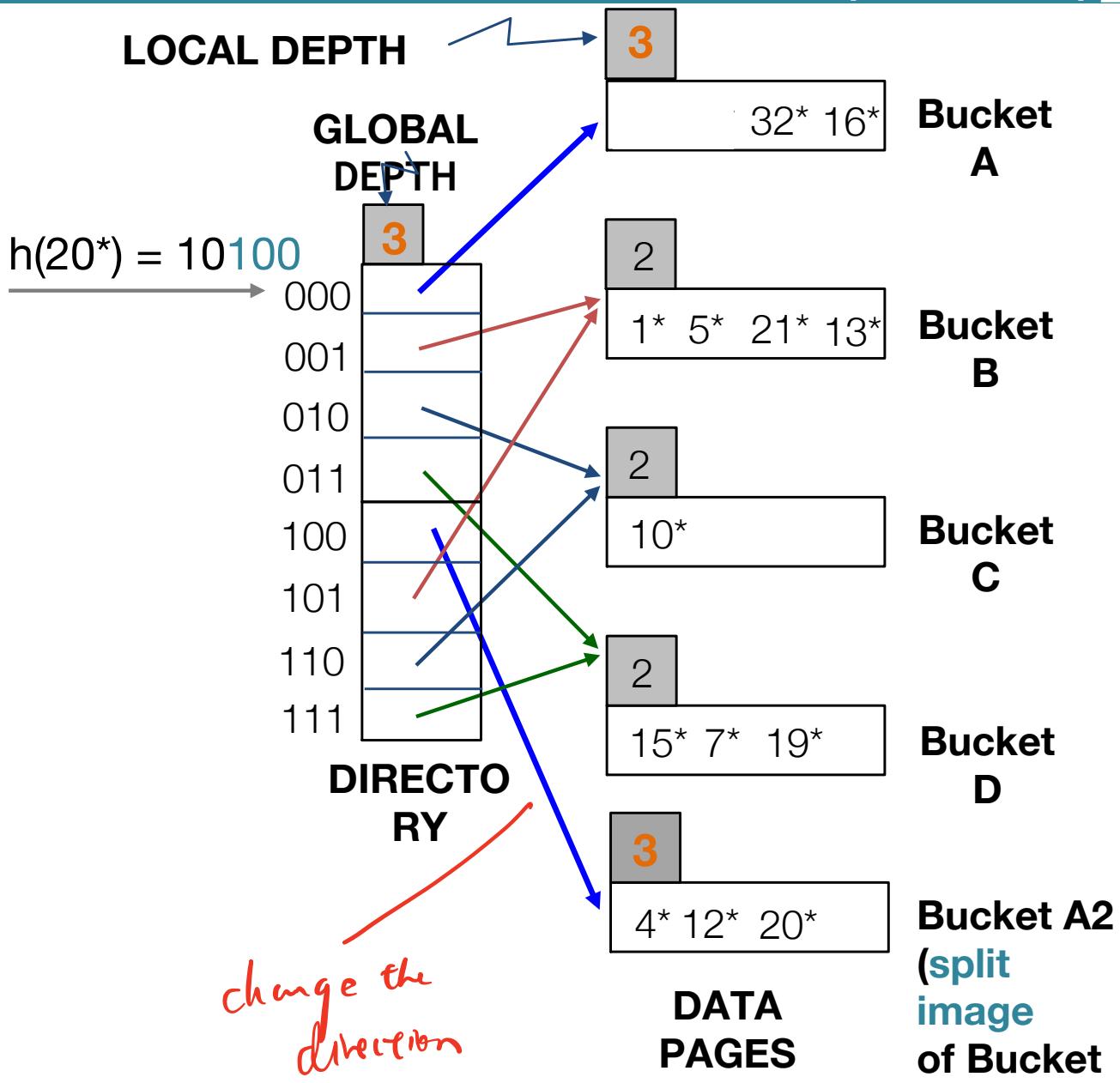
$32=(100000)_2$
$16=(10000)_2$

Notice that splitting a bucket only requires doubling the directory when new LD > old GD

Directory doubled by *copying it over*

$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Example: Insertion into a full bucket
Insert 20 = $(10100)_2$

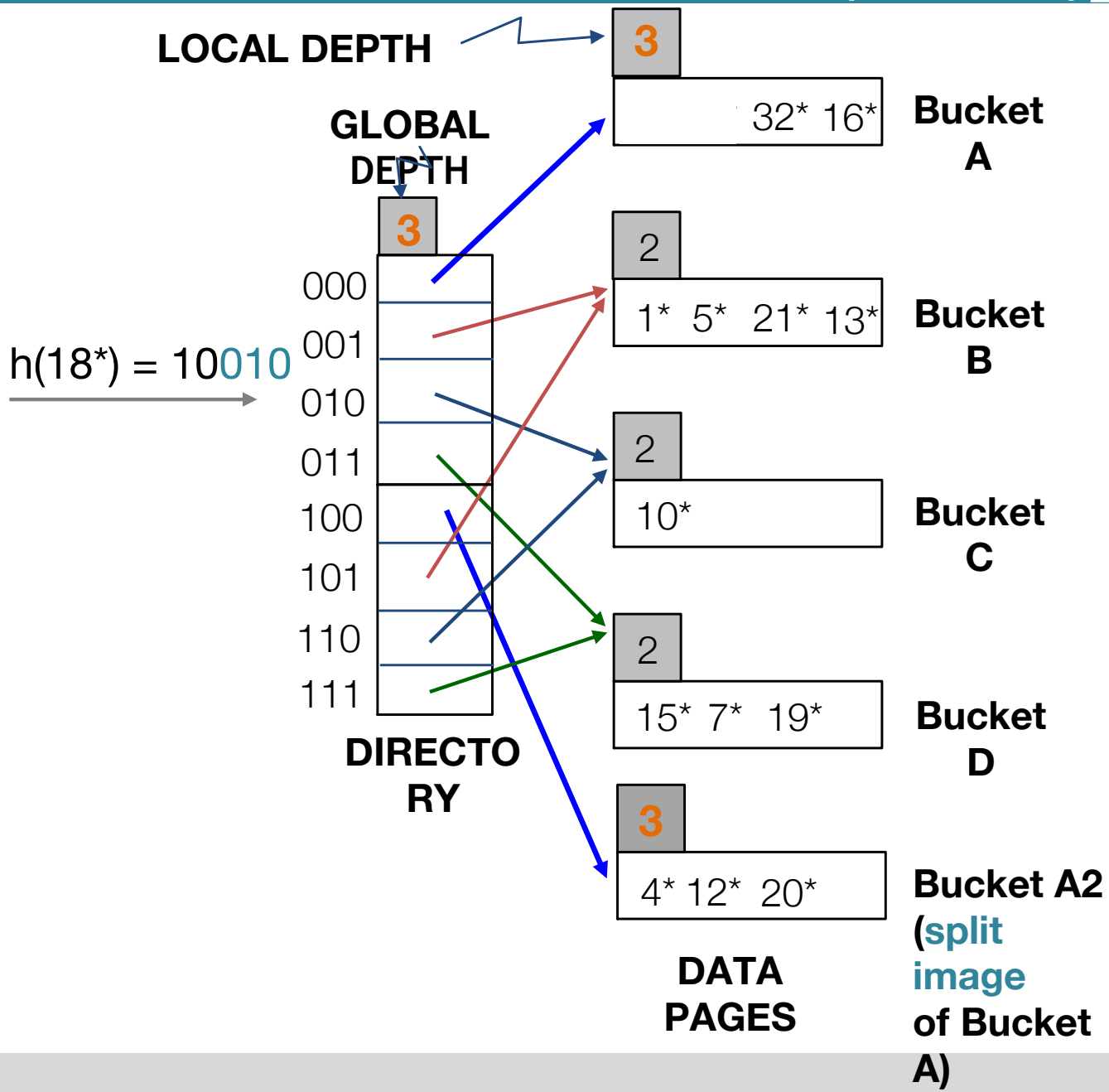


$32=(100000)_2$
$16=(10000)_2$

Fix the pointer for the split image bucket

$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Example: Insertion into a free bucket
Insert 18 = $(10010)_2$



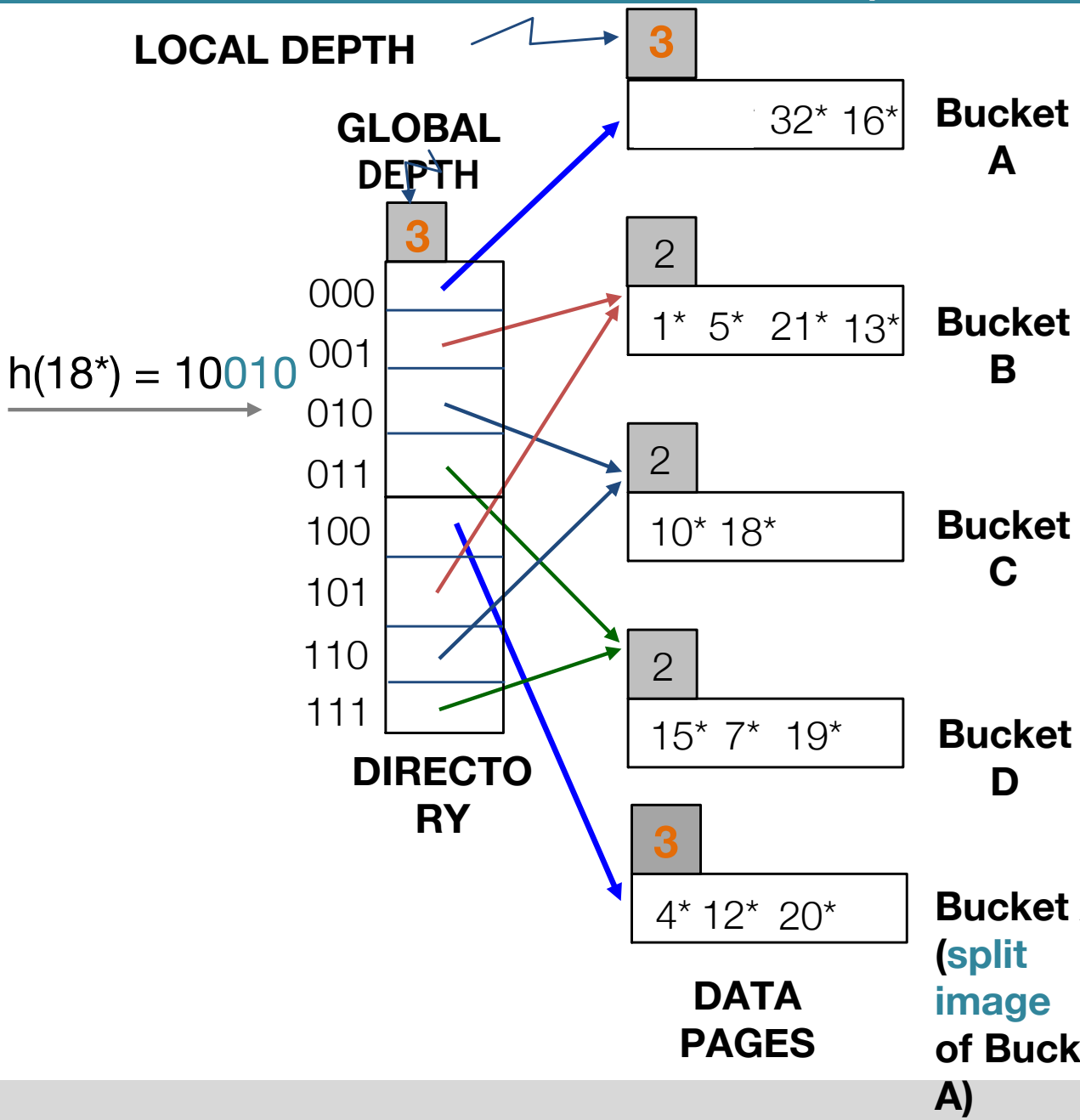
$32=(100000)_2$
$16=(10000)_2$

Use last 3 bits to hash

global depth

$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Example: Insertion into a free bucket
Insert 18 = $(10010)_2$



$32=(100000)_2$
$16=(10000)_2$

Use last 3 bits to hash

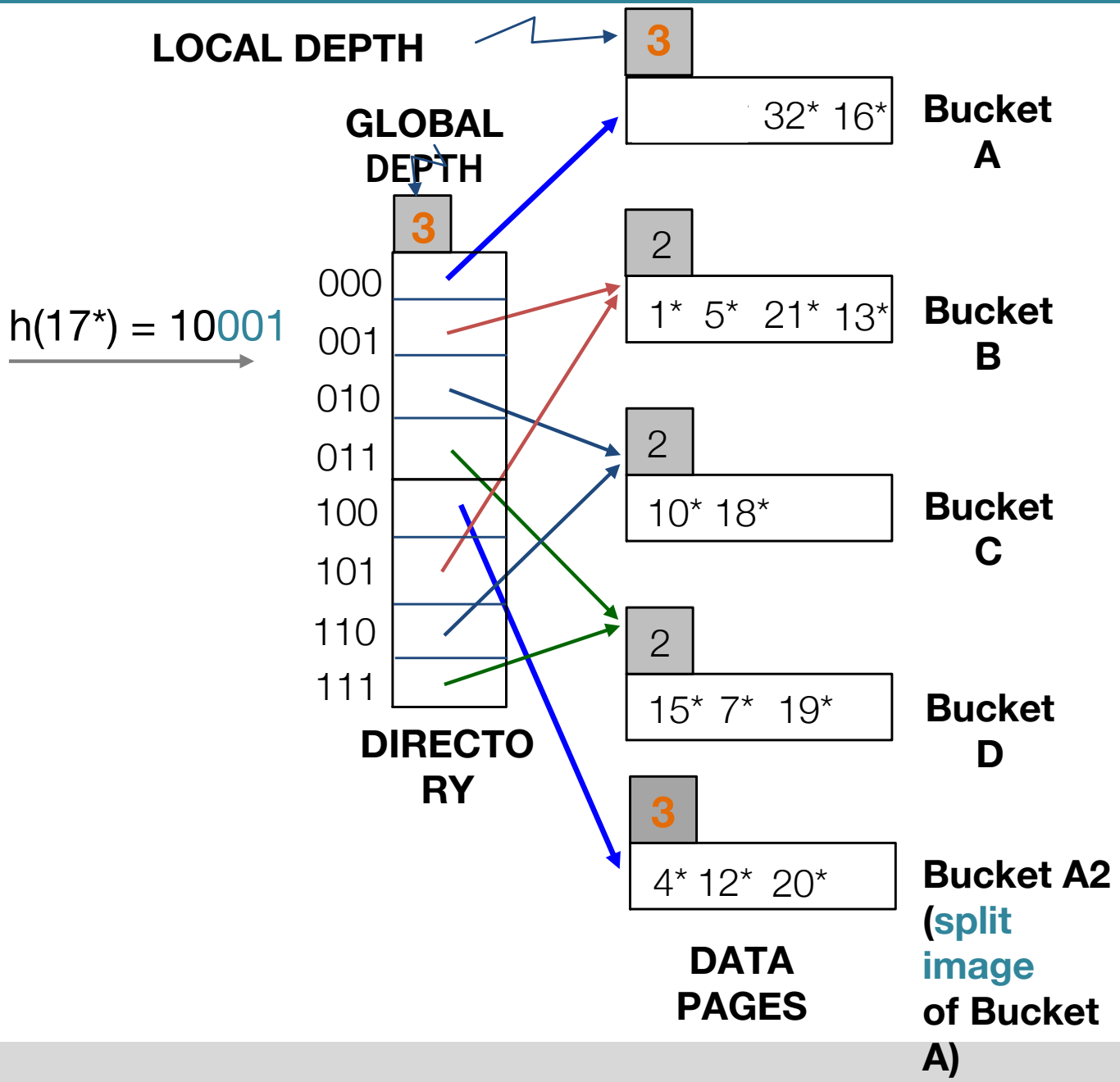
$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$

Question??

Local depth is

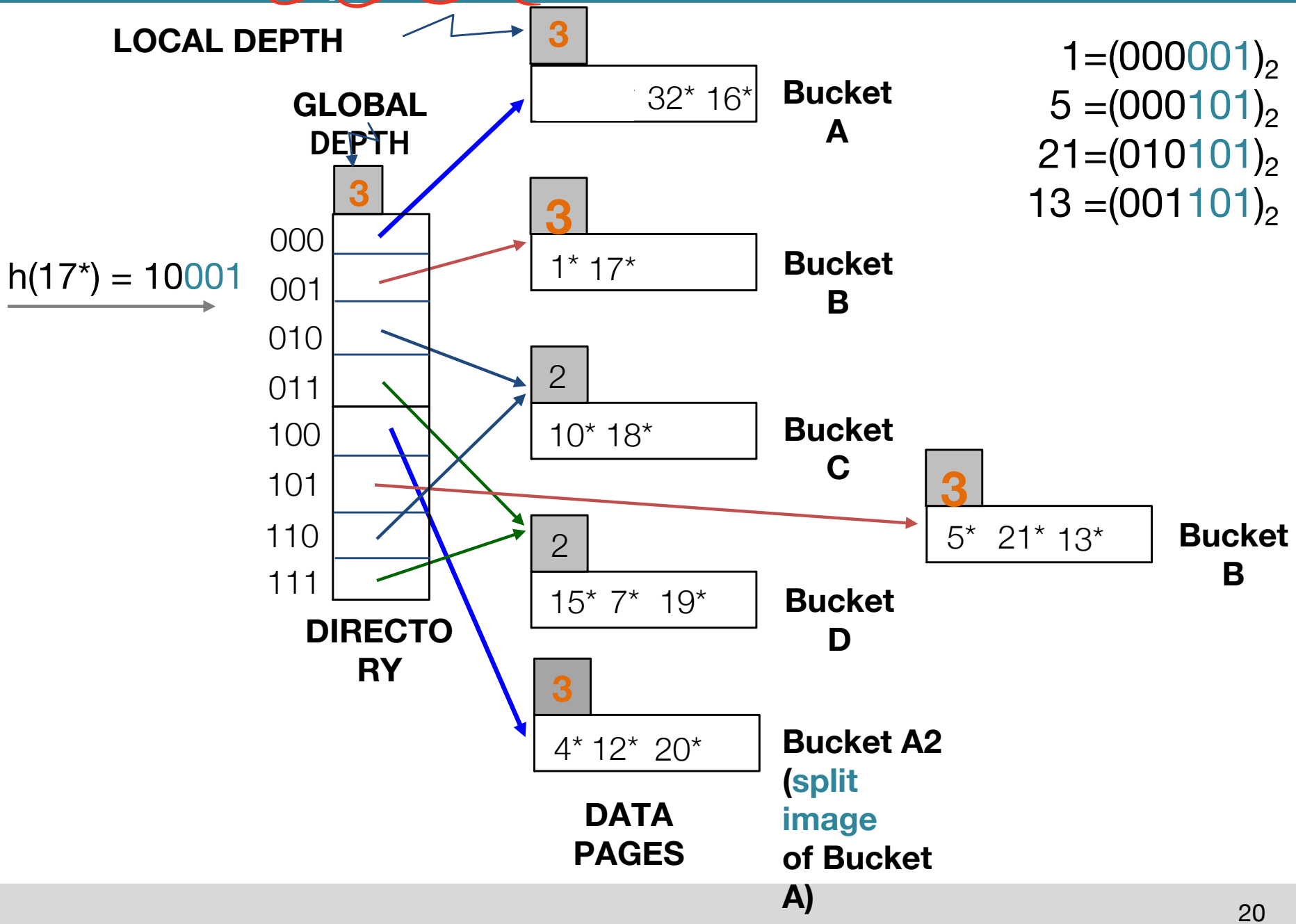
A. Always less than or equal to global depth 
B. Always greater than or equal to global depth
C. Can be less than, greater than, or equal to global depth
D. None of the above

Question: What will happen if 17 is inserted?



$1=(000001)_2$
$5=(000101)_2$
$21=(010101)_2$
$13=(001101)_2$

Answer: Split only. No directory doubling is required since invariant is OK



Comments on Extensible Hashing

one page look up

- How many disk accesses for equality search?
  - One if directory fits in memory, else two
- Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large

lots of with low local depth
but a few with high local depth

Question??

Extensible hashing still needs overflow pages

A. True

B. False

split and redistribute

but what if all entries are the same?

Answers

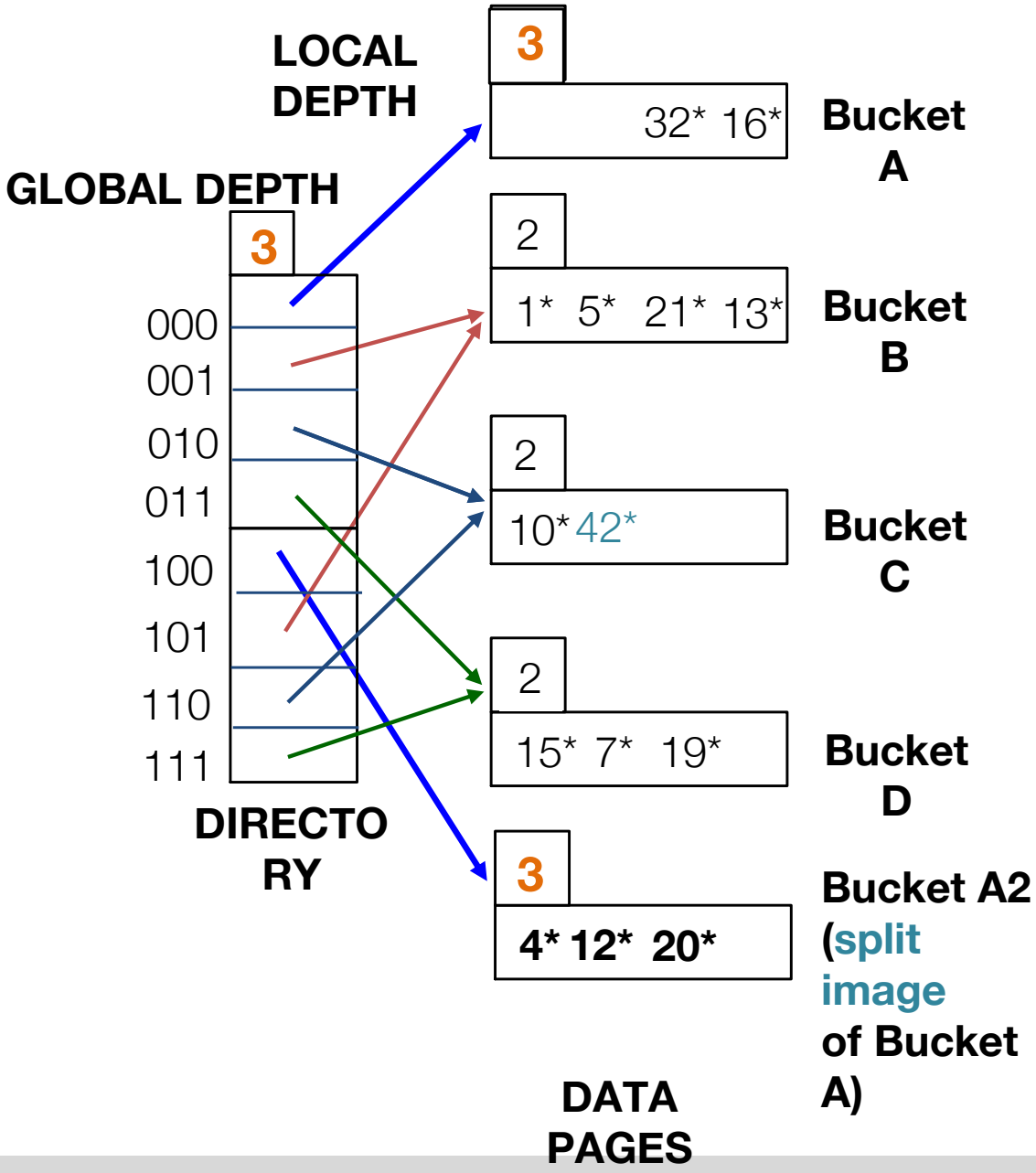
- Multiple entries with same hash value cause problems => We could still need overflow pages

2 insert page count: one read (1) one write (1) (no split)

3 (with split): read (1), insert, create new copy, write two pages (2)

- Delete: Reverse of inserts – see textbook

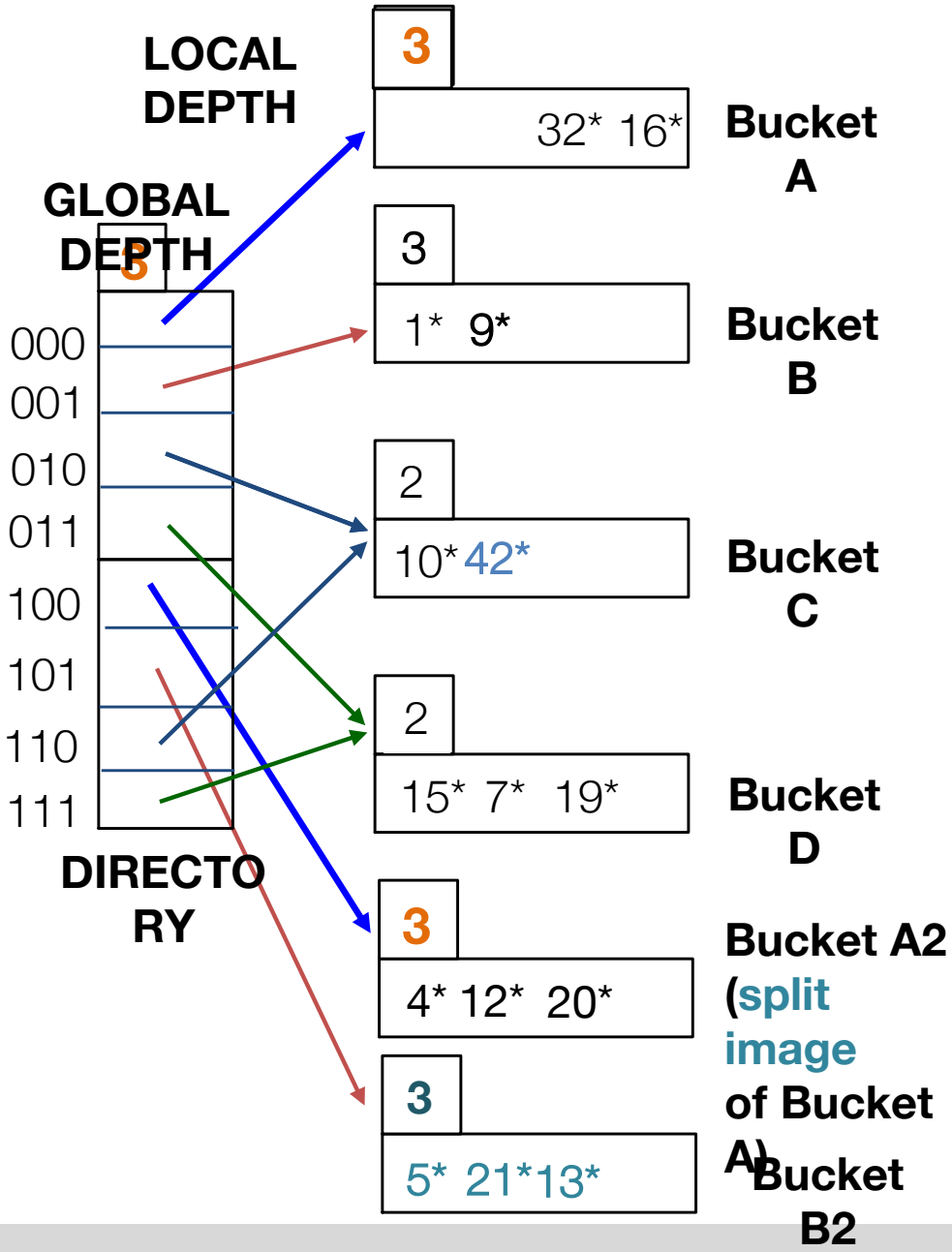
Extensible Hashing Exercise: Insert 42, 9



$42=(101010)_2$

$9=(1001)_2$

Answer: Insert 42, 9



$42=(101010)_2$

$9=(1001)_2$

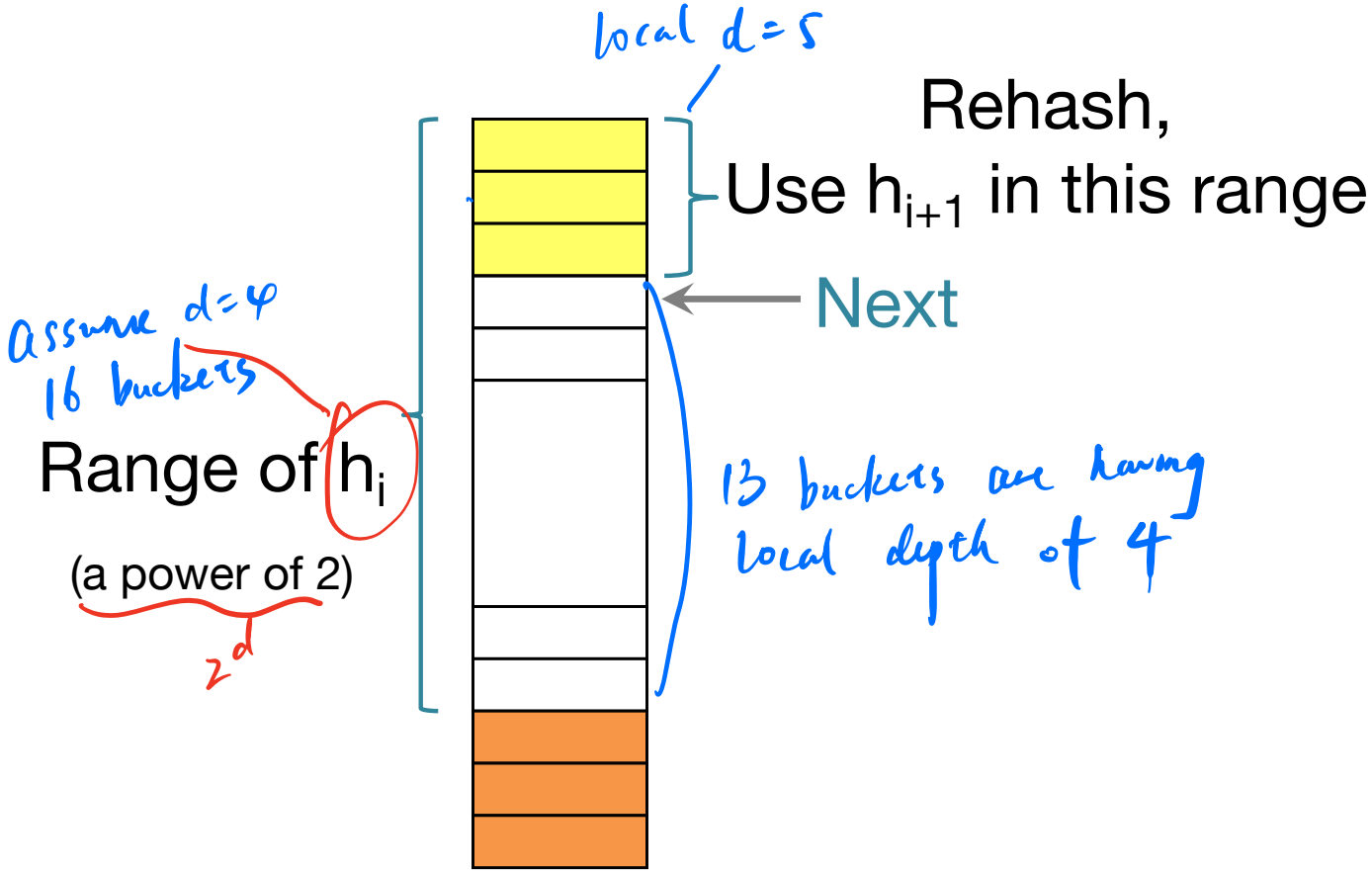
2nd technique: Linear Hashing

assume a good hash fcn distribute things evenly

- Overflow pages are allowed
- Buckets are split in a deterministic order (a Next pointer is maintained), rather than the overflowing bucket
- Upon an overflow anywhere, next bucket in the deterministic order is split.
- This means that overflow chains can occur
- Eventually, all buckets will split, making the overflow chains rare

2nd Technique: Linear Hashing

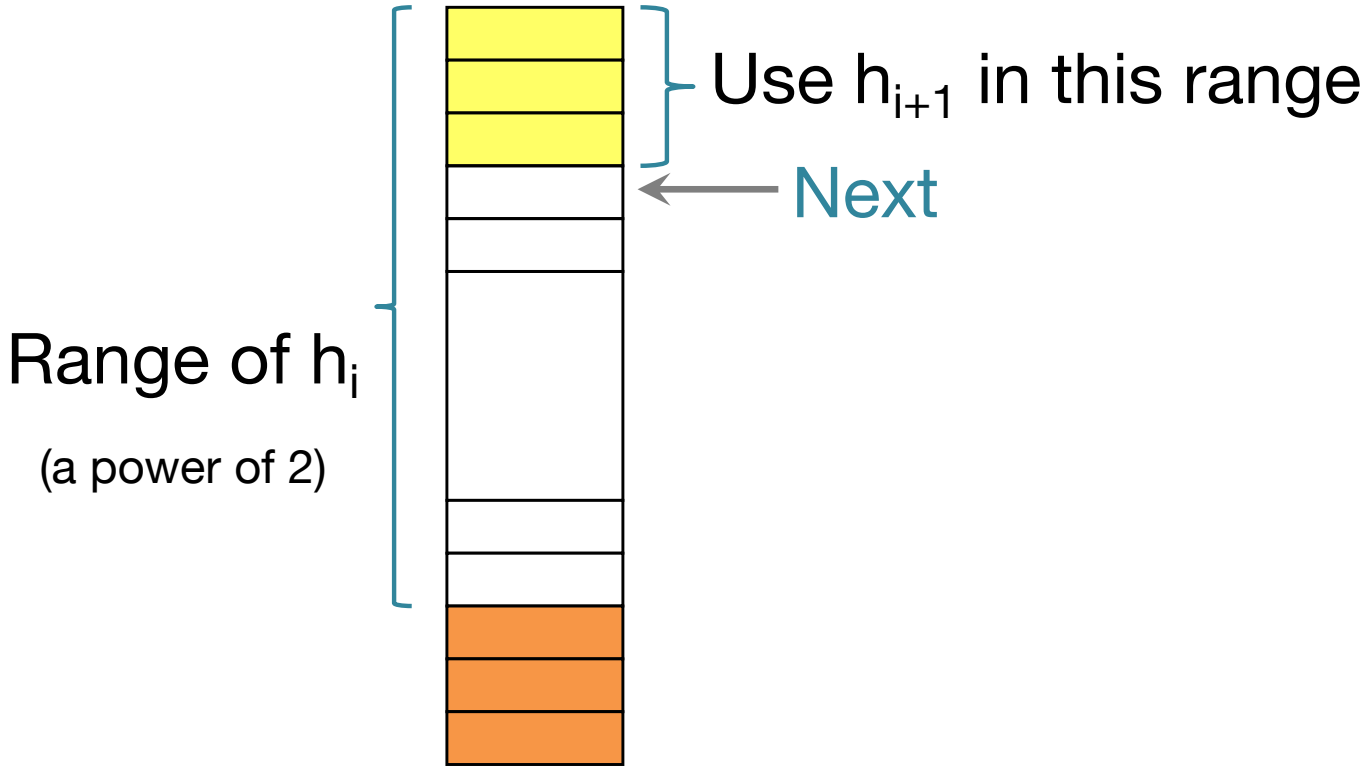
- Works in rounds
- Grows more gradually



Yellow and Orange are split image buckets.

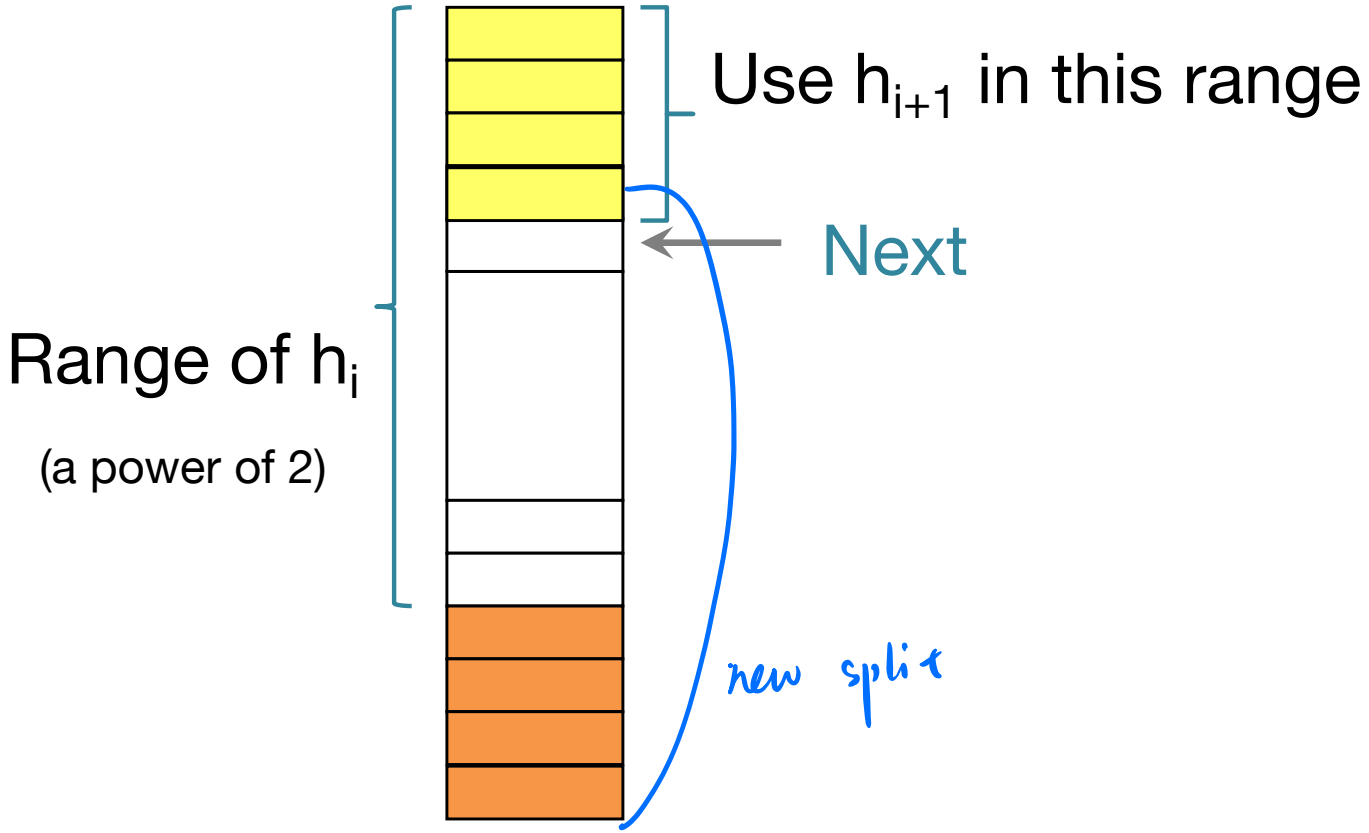
Next identifies the bucket to be split next.

Buckets During Round *i*



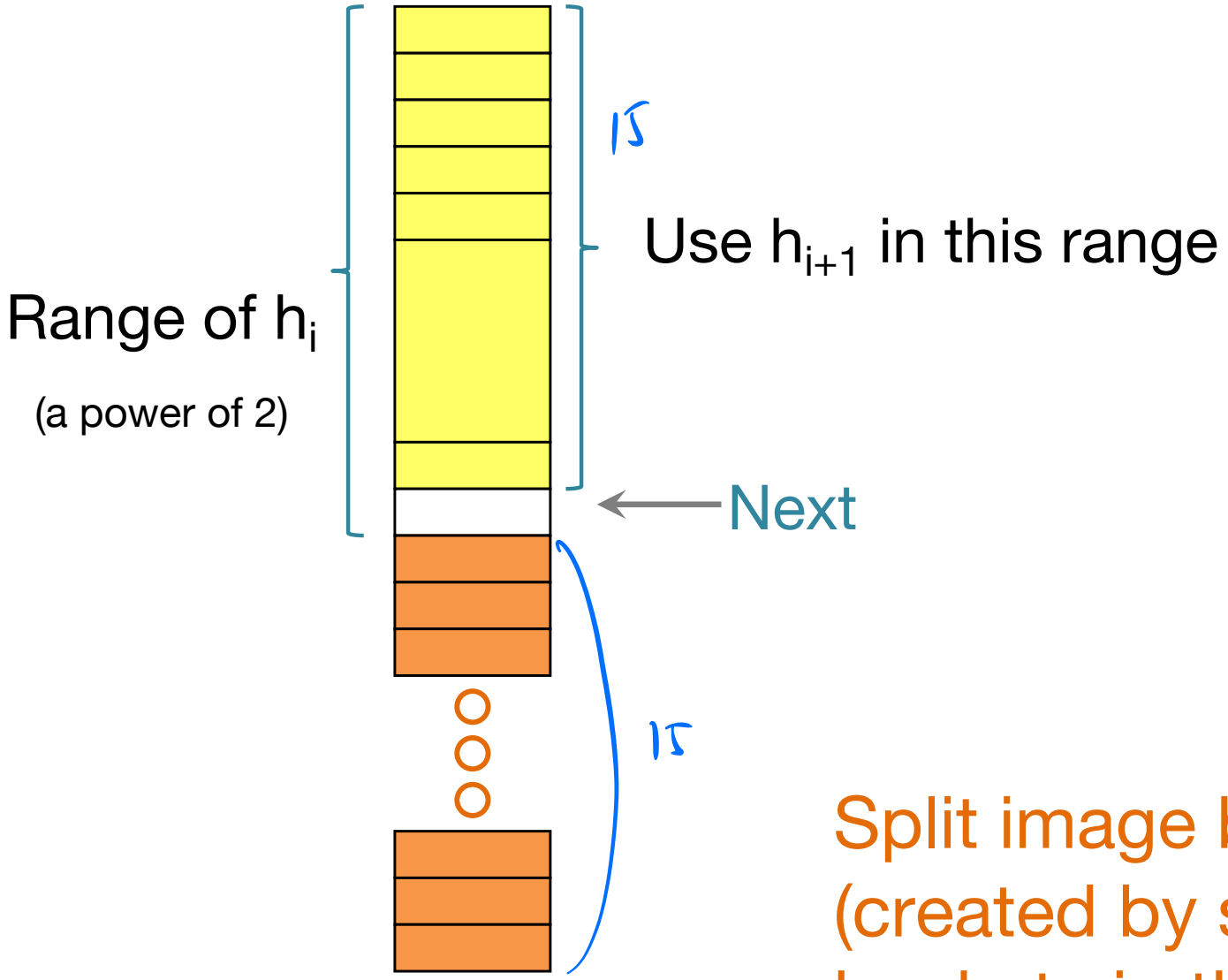
Split image buckets (created by splitting other buckets in this round). h_{i+1} uses an extra bit over h_i

Upon an overflow anywhere



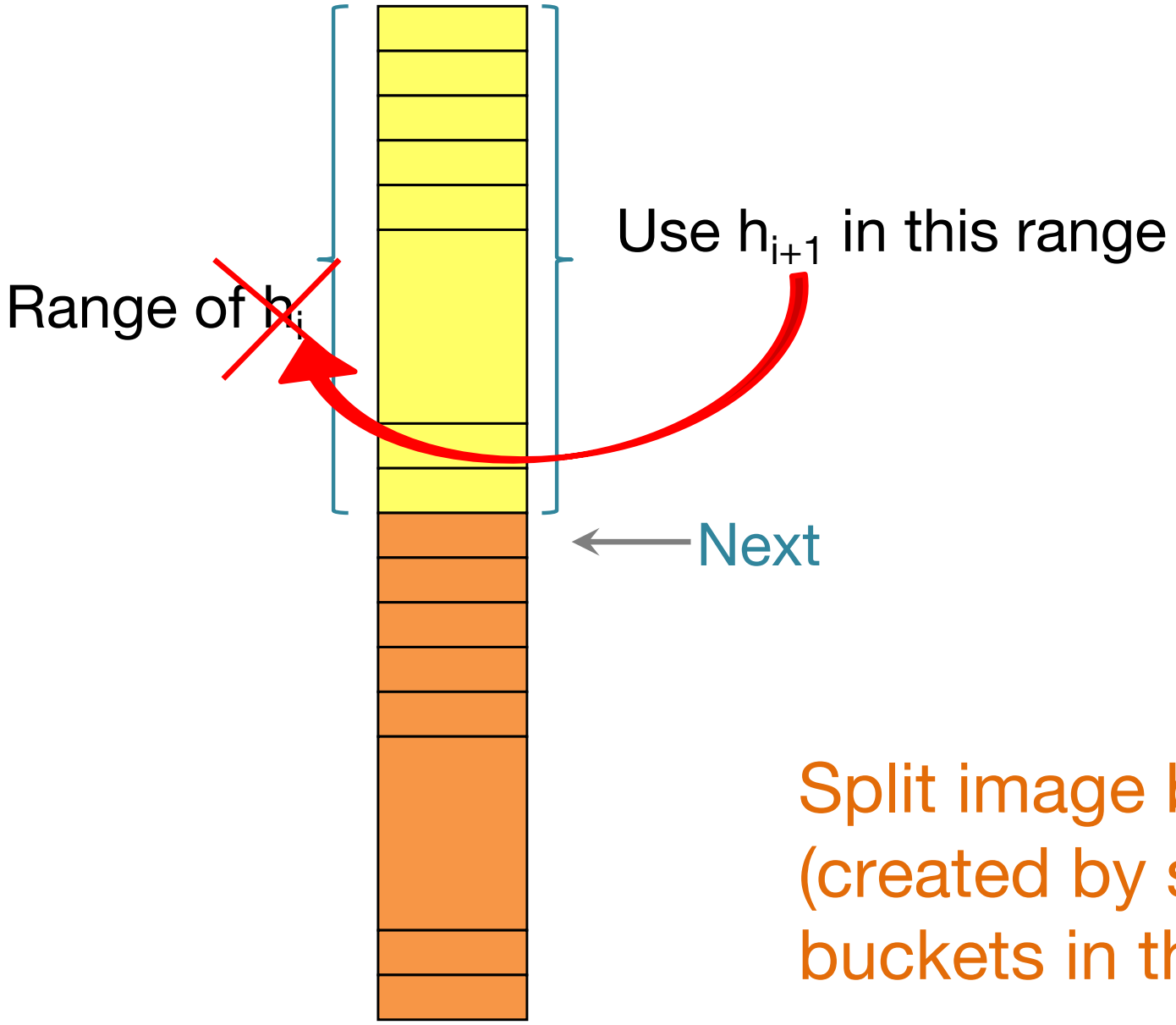
Add an image bucket, advance Next.

Overflow: Split next bucket, advance Next



Split image buckets (created by splitting other buckets in this round)

End of Round *i* = Start of Round *i+1*



Split image buckets (created by splitting other buckets in this round)

End of Round *i* = Start of Round *i+1*

turn everything white, move "next" to beginning

16 → 32

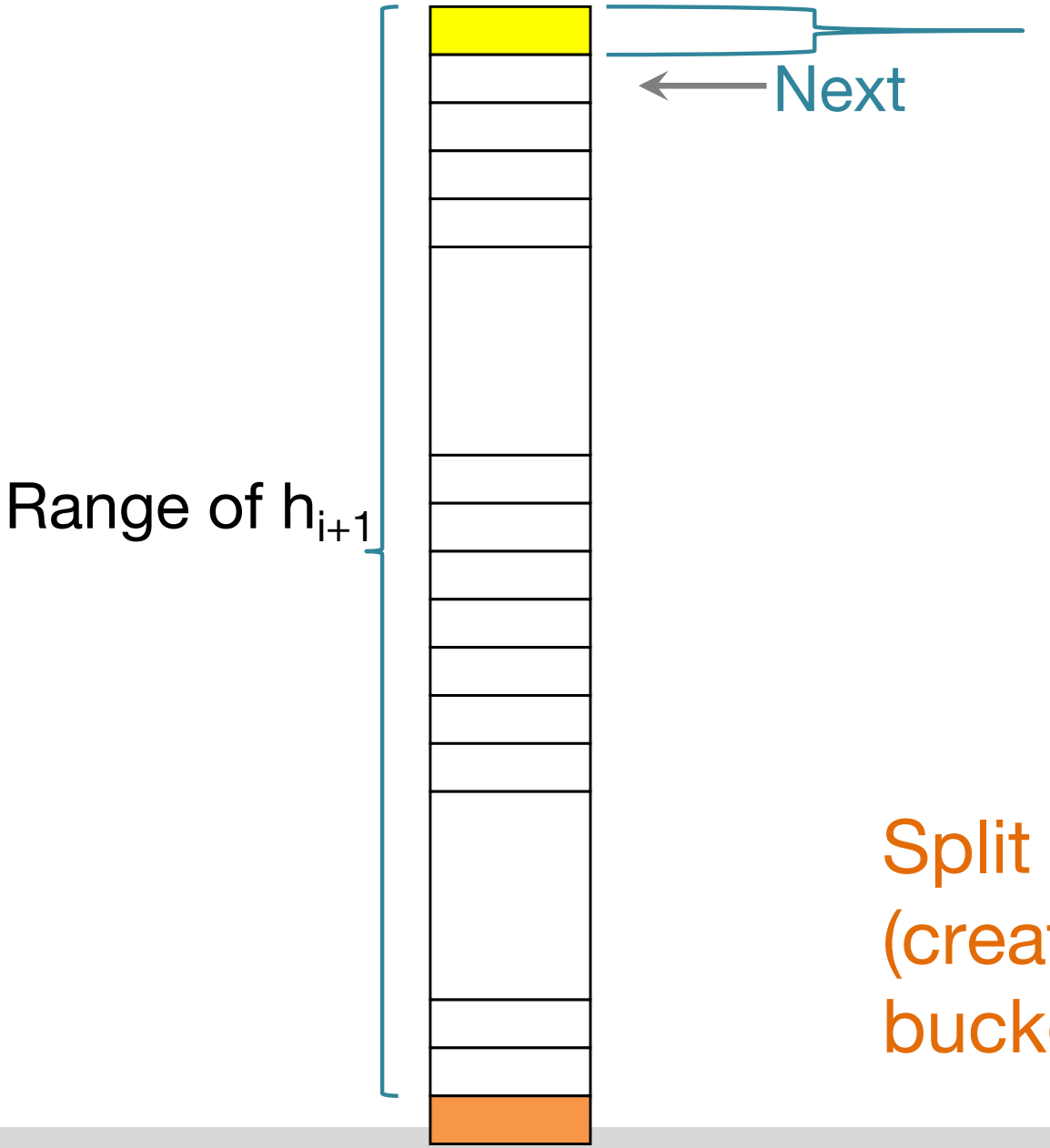
Size = 32 now

Range of h_{i+1}

Next

Split image buckets (created by splitting other buckets in this round)

Round *i+1* after the First Split



Split image buckets (created by splitting other buckets in this round)

Linear Hashing: Details

- Splitting proceeds in rounds, starting from Level 0
  - During round *Level*, **only** h_{Level} and $h_{Level+1}$ are in use
- Variables
  - Level: Initialized to 0 (like global depth)
  - Next: Pointer to the bucket next to be split
- At the beginning of round # Level,
  - # buckets (some virtual) in the file = $N_{Level} = N * 2^{Level}$

where N is initial number of buckets

Level is not # of bits in extensible hashing

N: base directory size to start with, never start with 1

Question??

In Linear Hashing, you always split the page pointed to by *Next*, even if the split is caused by an overflow to a different page.

A. True

B. False

Linear Hashing: Hash Functions

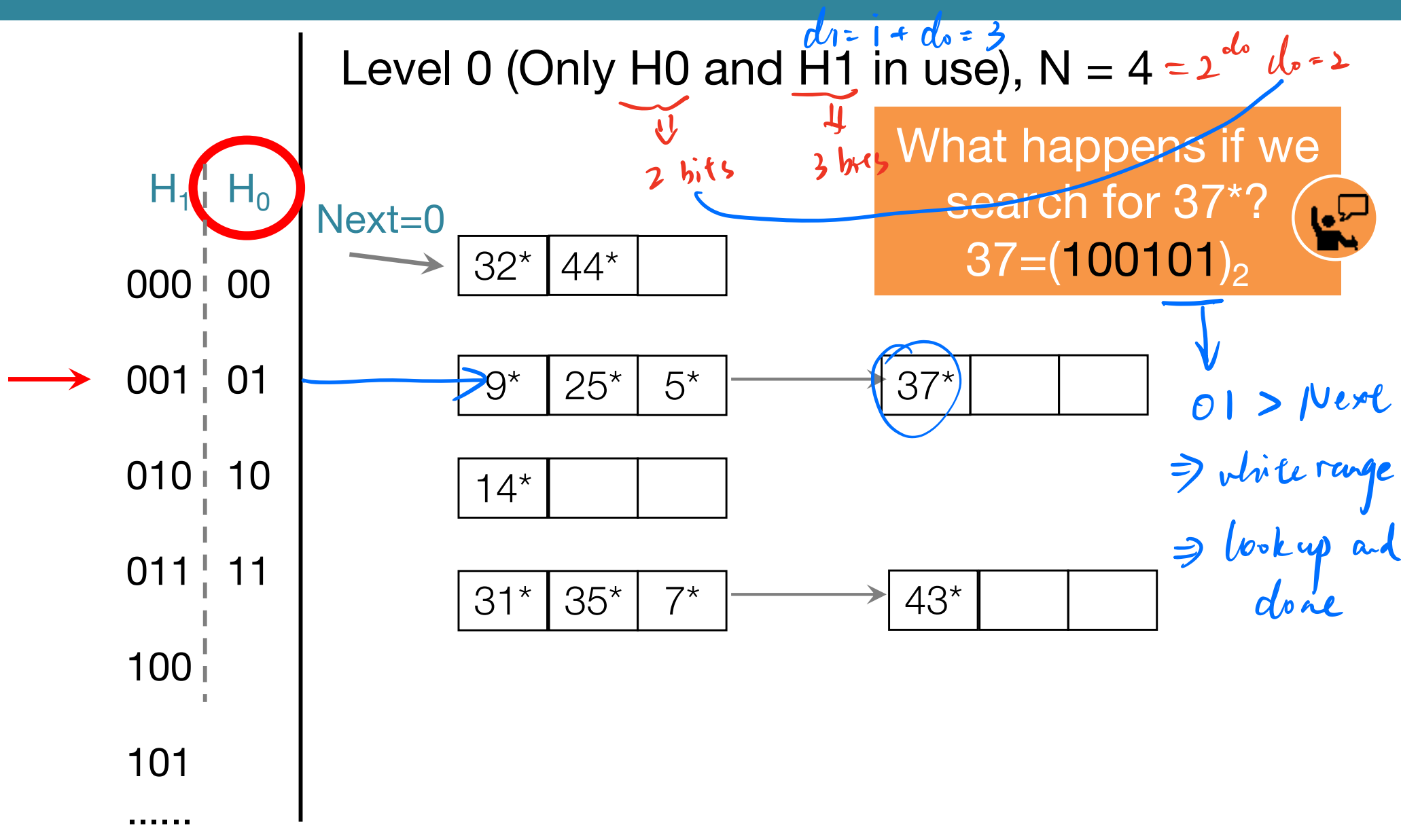
- Use a family of hash functions: H_0, H_1, H_2, ...
  - H_{i+1} doubles the range of H_i (similar to directory doubling)
  - Hash family typically obtained by choosing a hash function *h*( ) and initial number of buckets $N=2^{d_0}$
  - To compute $H_i(value)$, apply hash function *h*( ), and look at last d_i bits of the result where $d_i = d_0 + i$
  - In other words: $H_i(value) = h(value)\ mod\ (2^i N)$

Linear Hashing Algorithm

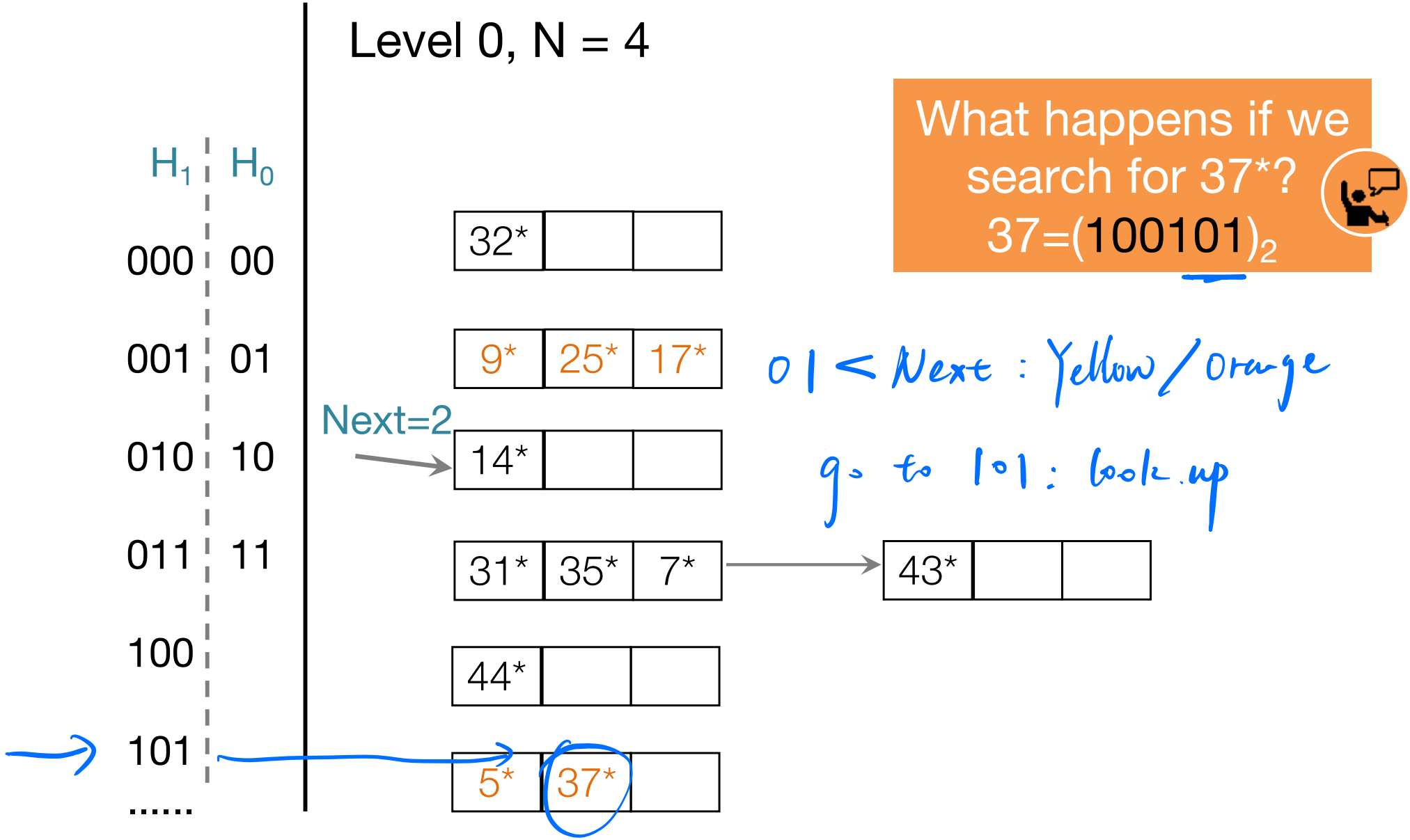
Search:

- Find bucket by applying $\mathbf{H}_{Level}$ *first.*
- Apply $\mathbf{H}_{Level+1}$ if the value returned by $\mathbf{H}_{Level}$ < Next
- Search for the item in the bucket (possibly a chain of pages)

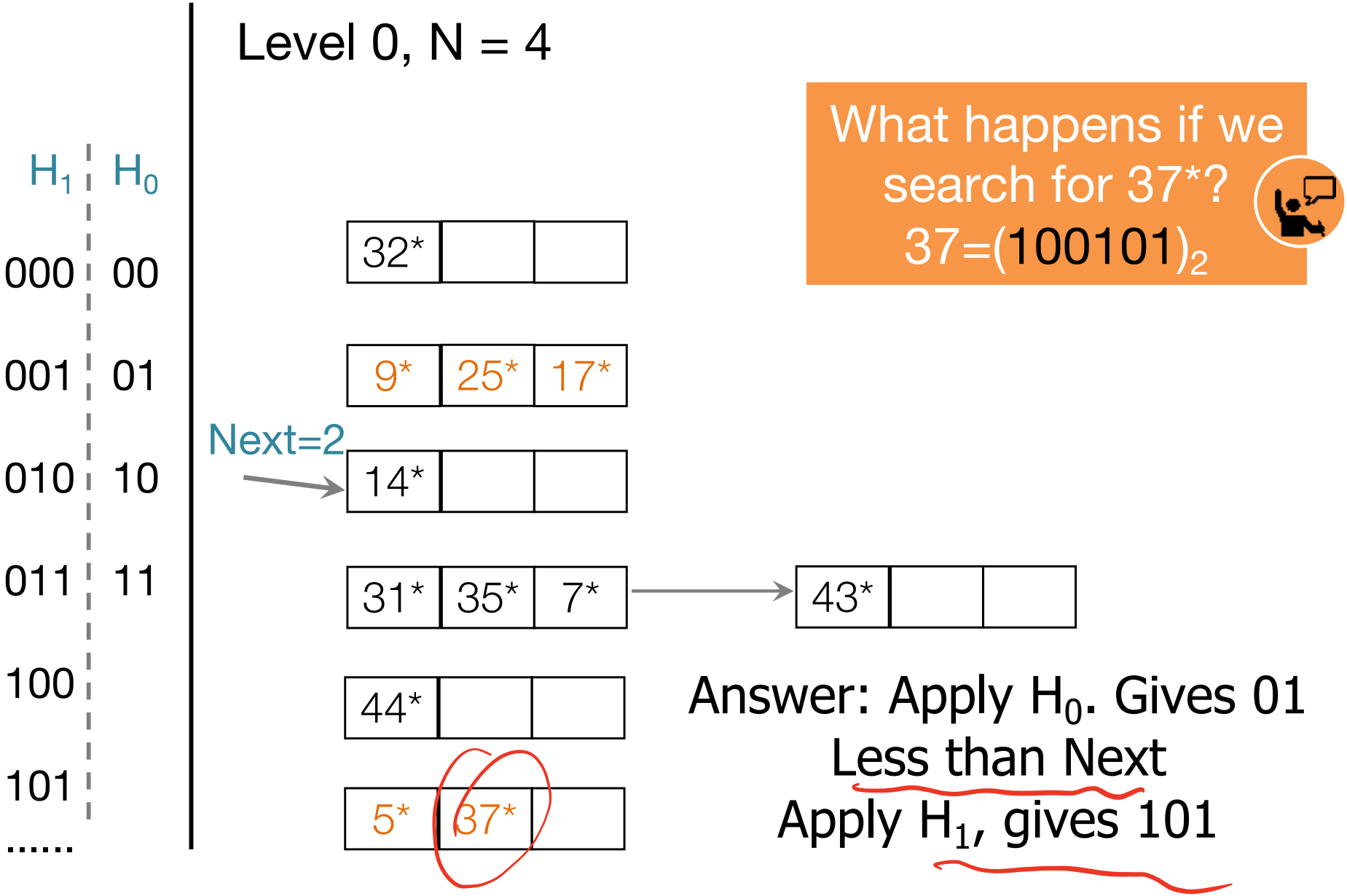
Linear Hashing Example #1



Linear Hashing Search Example #2



Linear Hashing Search Example #2



Linear Hashing -- Insert

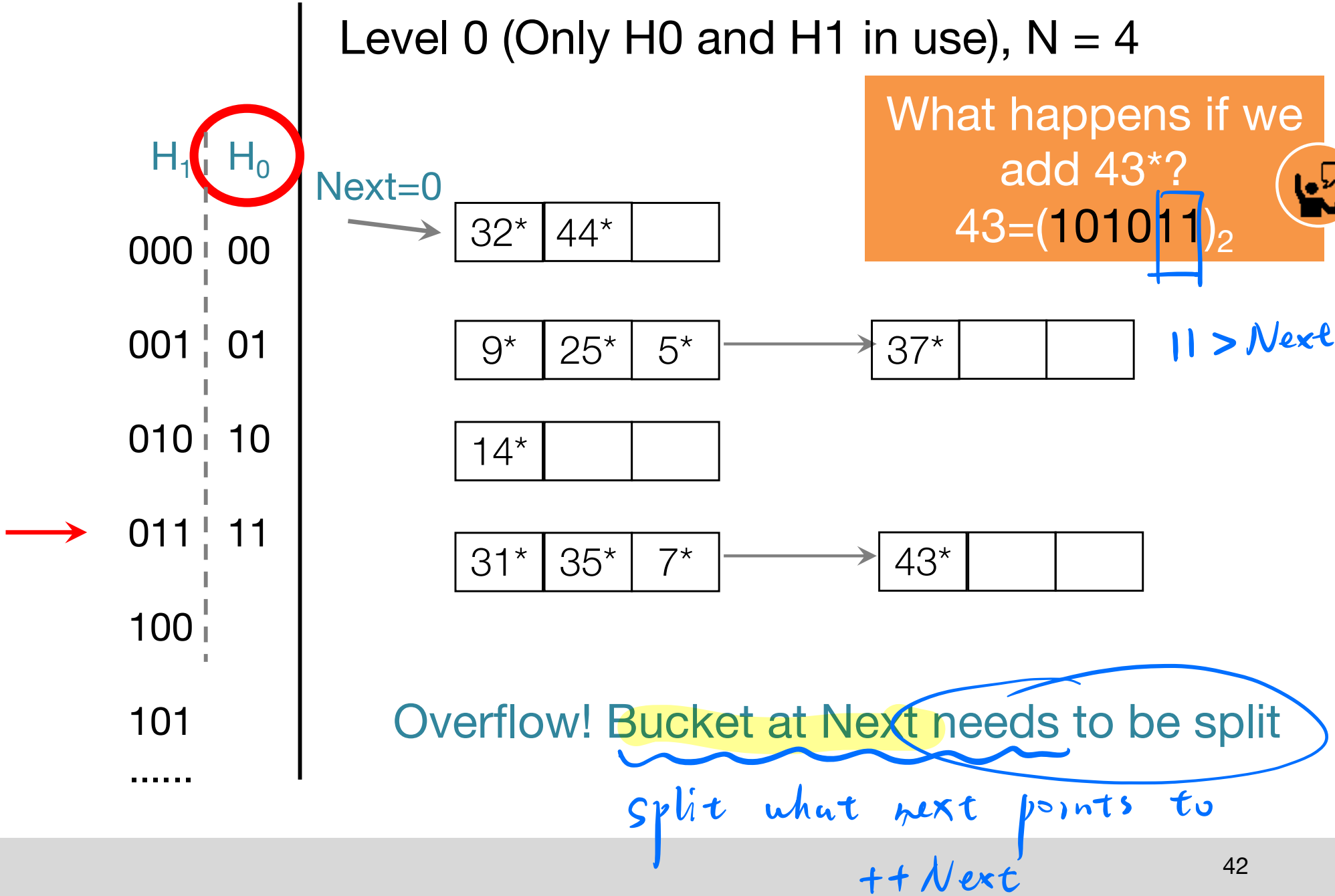
- Find bucket by applying h_{Level}
  - Apply $h_{Level+1}$ if the value returned by h_{Level} < Next
- If bucket being inserted into is not full, insert normally. DONE
- Else, full page.
  - Add overflow page to the bucket and insert. (see special case #1)
  - Split *Next* bucket using $h_{Level+1}$ and increment Next (see special case #2)

Special case #1: If full bucket is Next, split first. (You may not need to also add an overflow page in the original bucket)

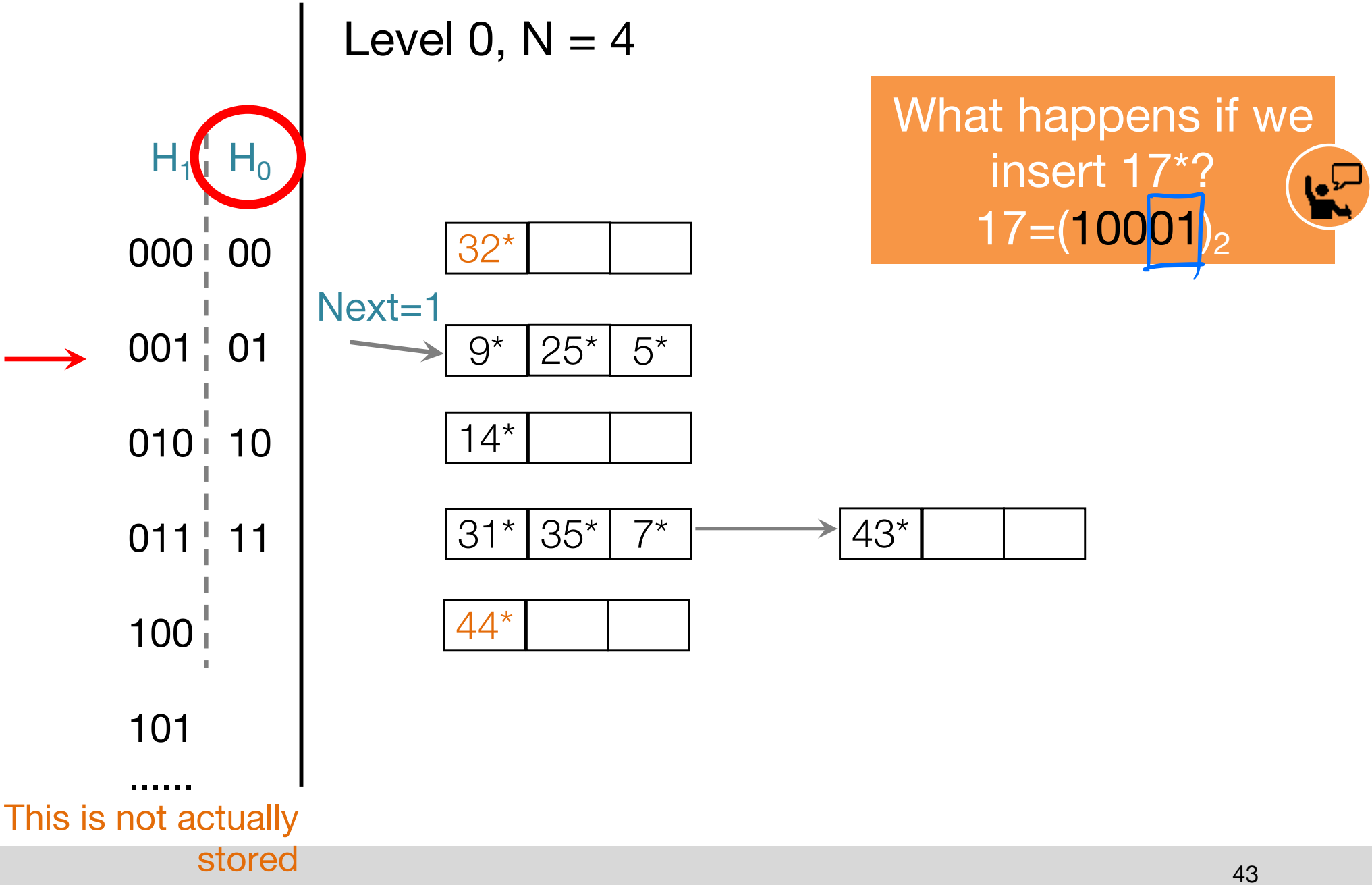
Special case #2: If Next = N_{Level} – 1 and a split is triggered

1. Split the bucket Next (i.e. the last bucket before the split image buckets)
2. Reset Next = 0
3. Start next round. Level = Level + 1

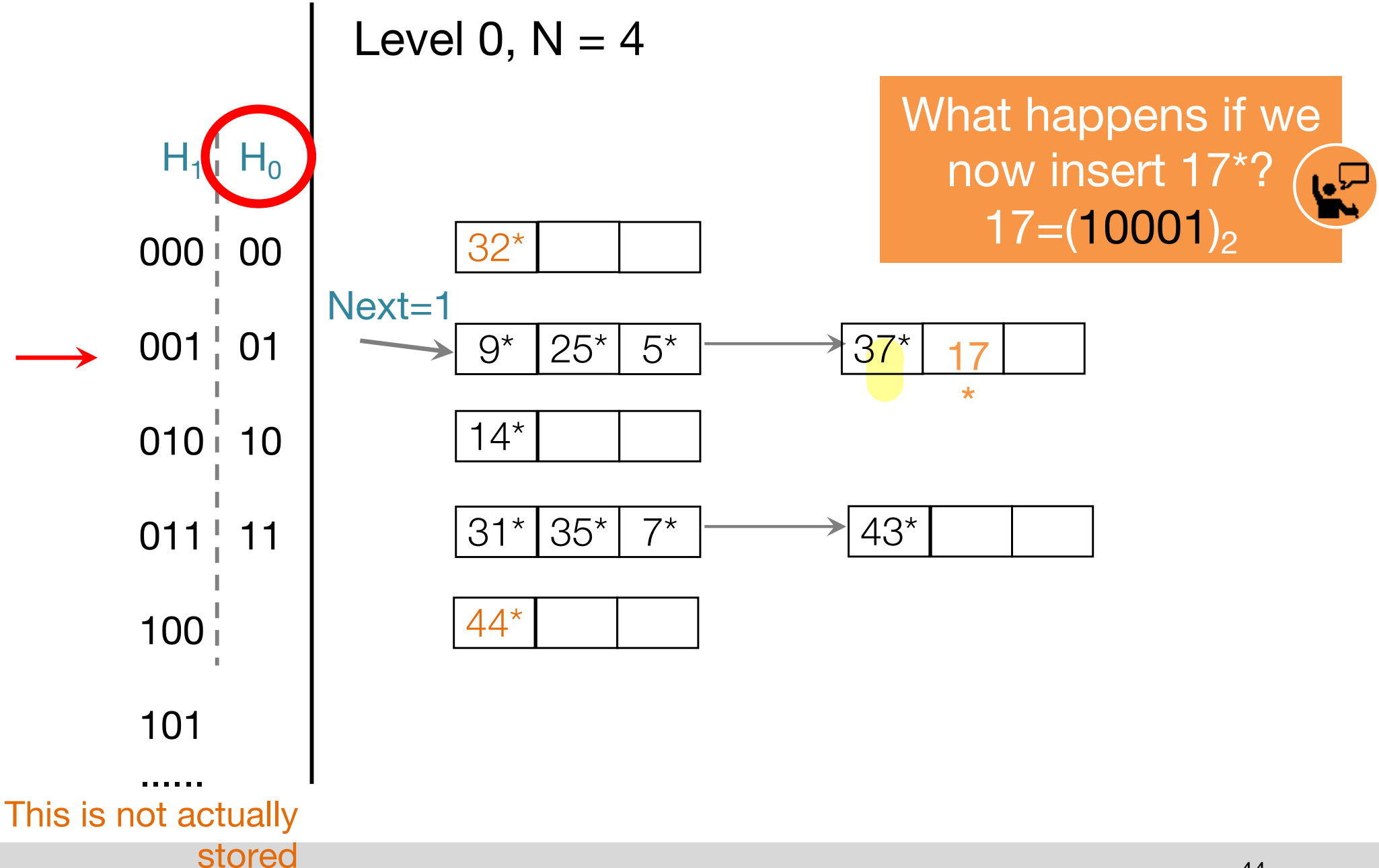
Linear Hashing Example



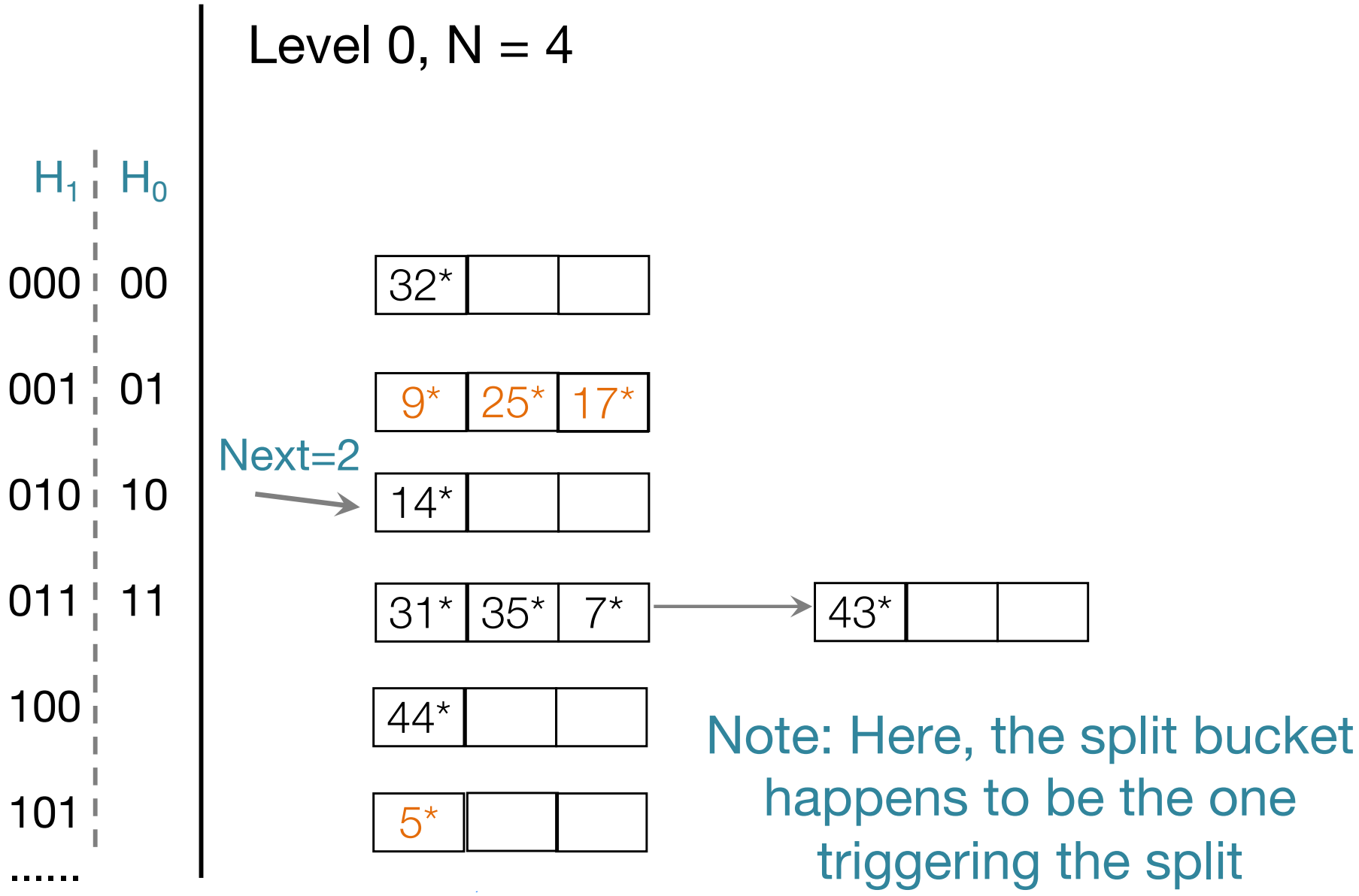
Linear Hashing Example



Wrong Answer



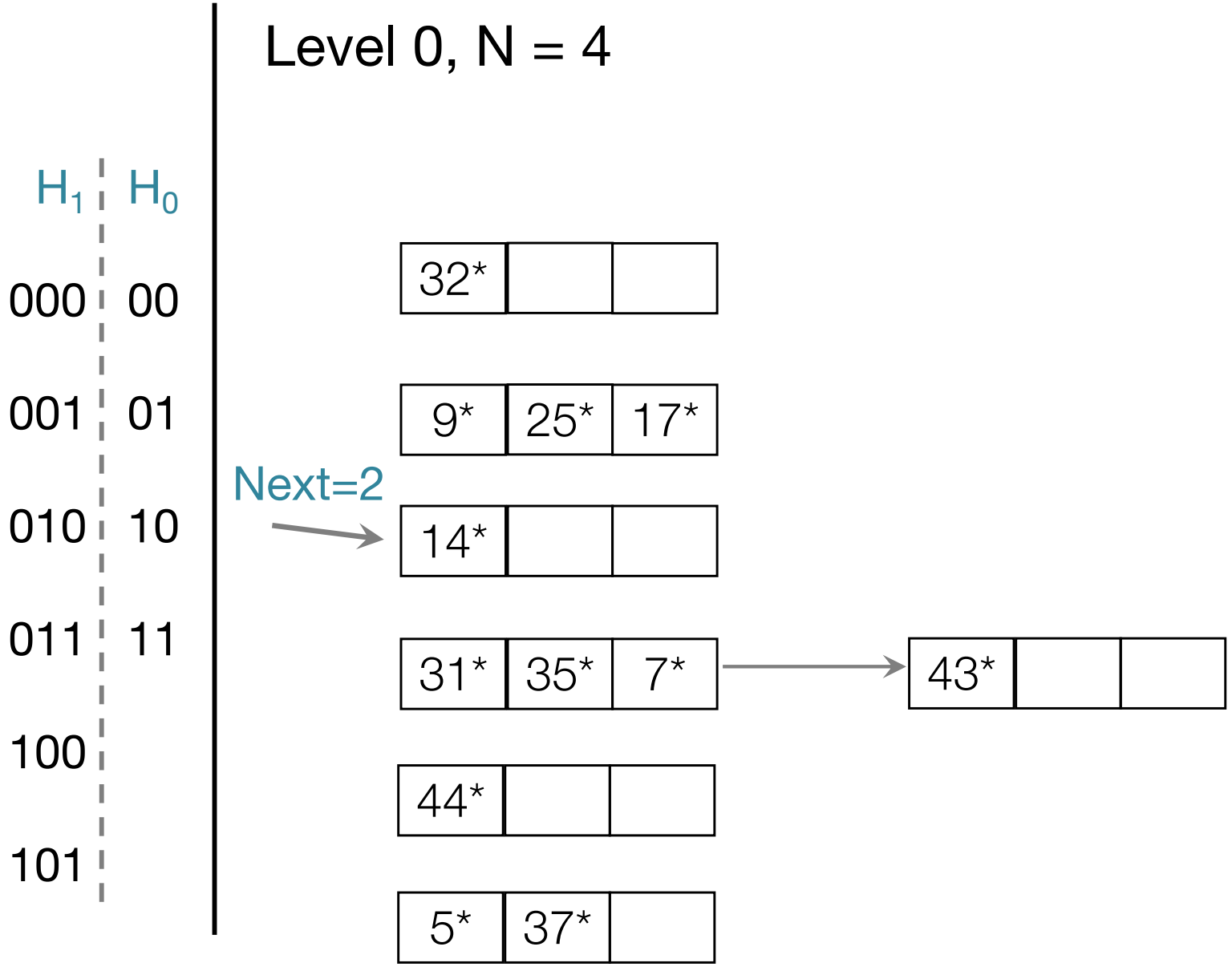
Correct Answer



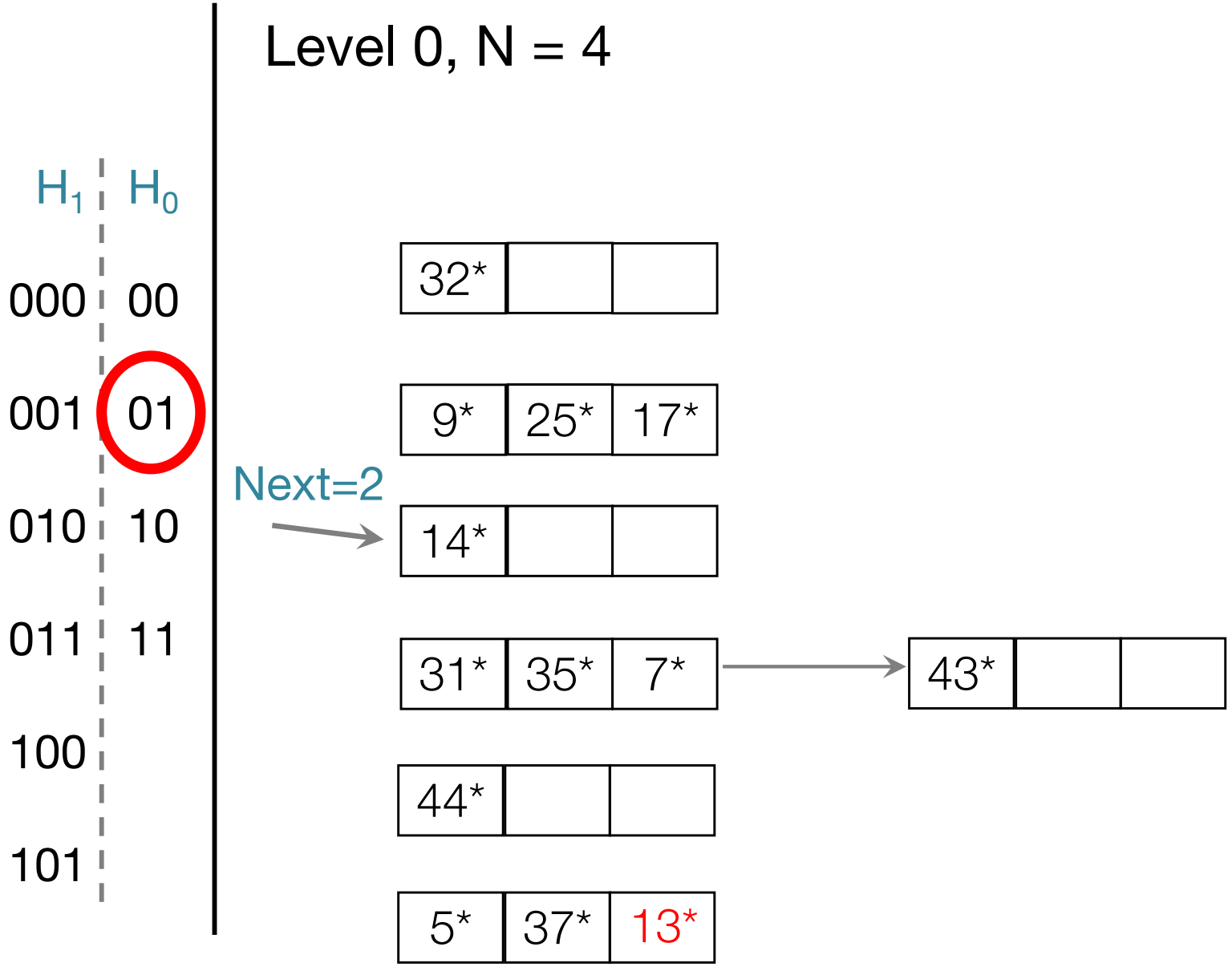
Linear Hashing (Contd.)

- Can choose any criterion to trigger split
  1. e.g., split on an overflow (as in example)
  2. e.g., space utilization on the page > 90%
- Since buckets are split round-robin, long overflow chains don't develop!
- Deletes: see textbook

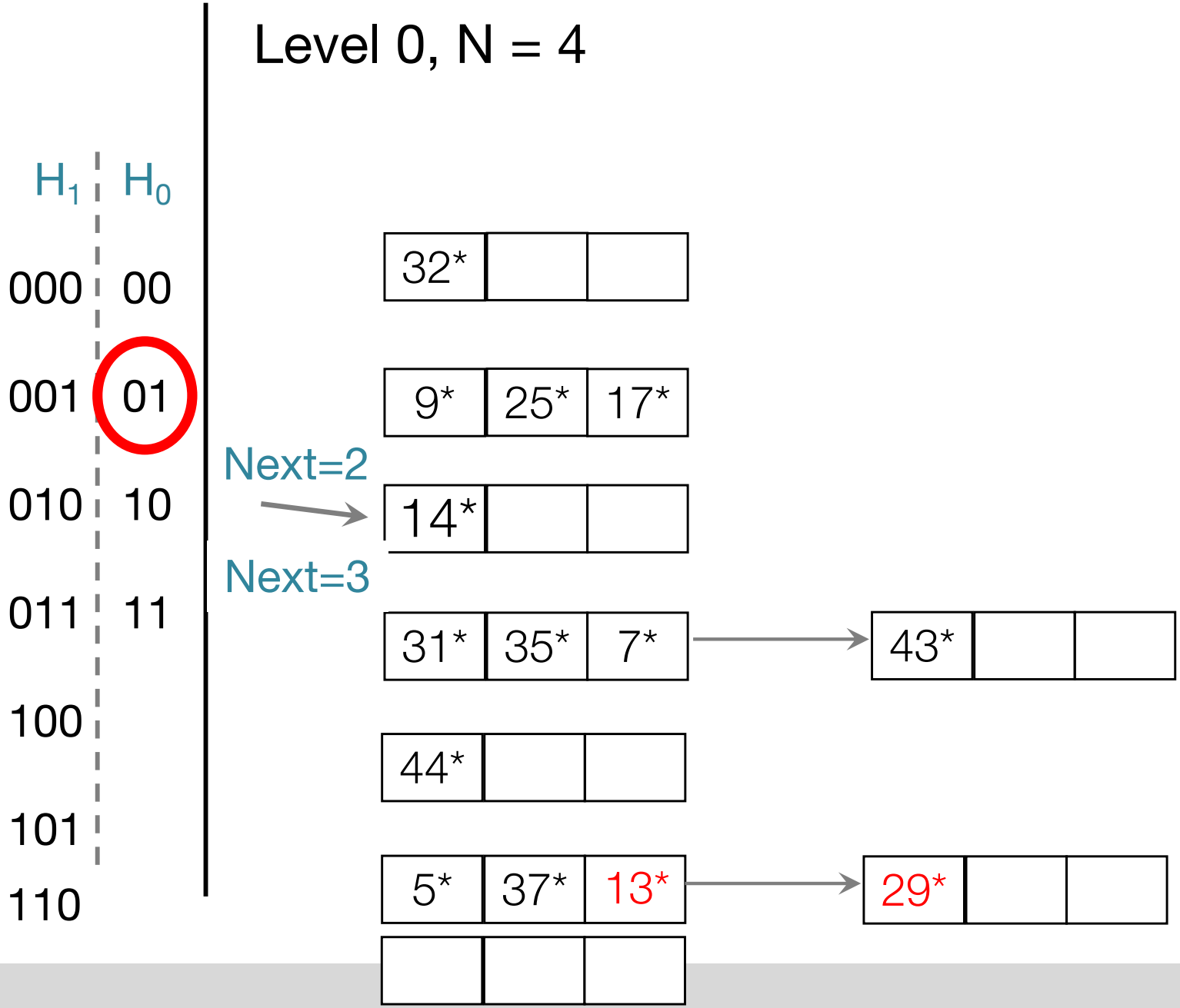
Exercise: Insert 13=$(1101)_2$ & 29=$(11101)_2$



Exercise: Insert 13=$(1101)_2$ & 29=$(11101)_2$



Exercise: Insert 13=$(1101)_2$ & 29=$(11101)_2$



Summary

- Discussed 3 kinds of hash-based indexes
- Static Hashing can lead to long overflow chains
- Extensible Hashing
  - Directory to keep track of buckets, doubles periodically
  - Always splits the “right” bucket
- Linear Hashing
  - Split buckets round-robin, and use overflow pages
  - Space utilization could be lower than Extensible Hashing

Optional Exercises

11.1, 11.3, 11.7, 11.9