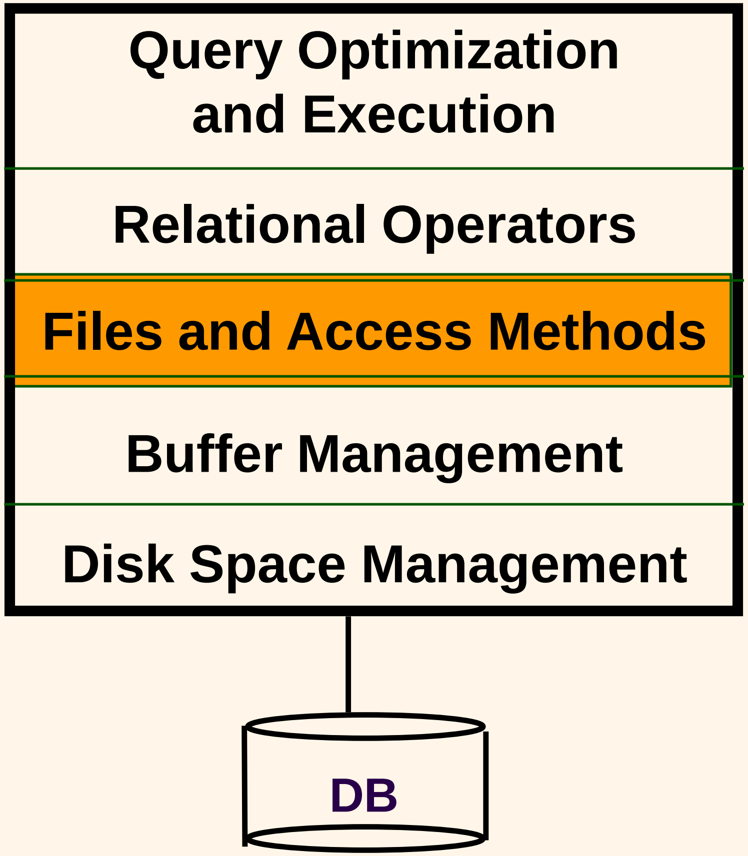
COMP7640
Database Systems & Administration

Hash-based Indexing

Where Are We Now?

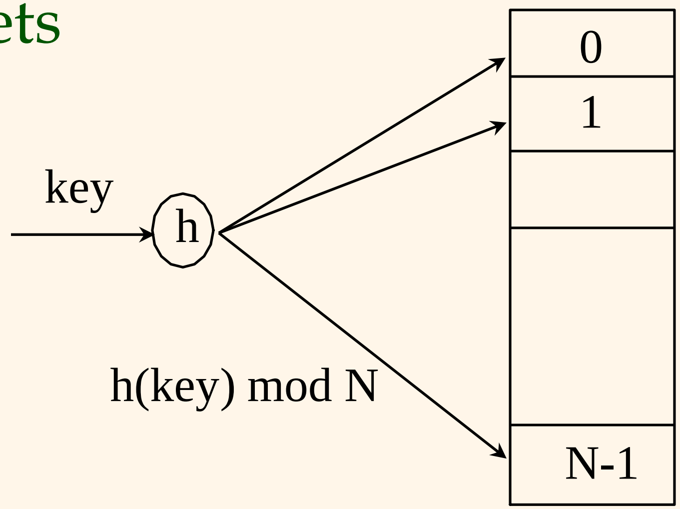


Hash-Based Indexes

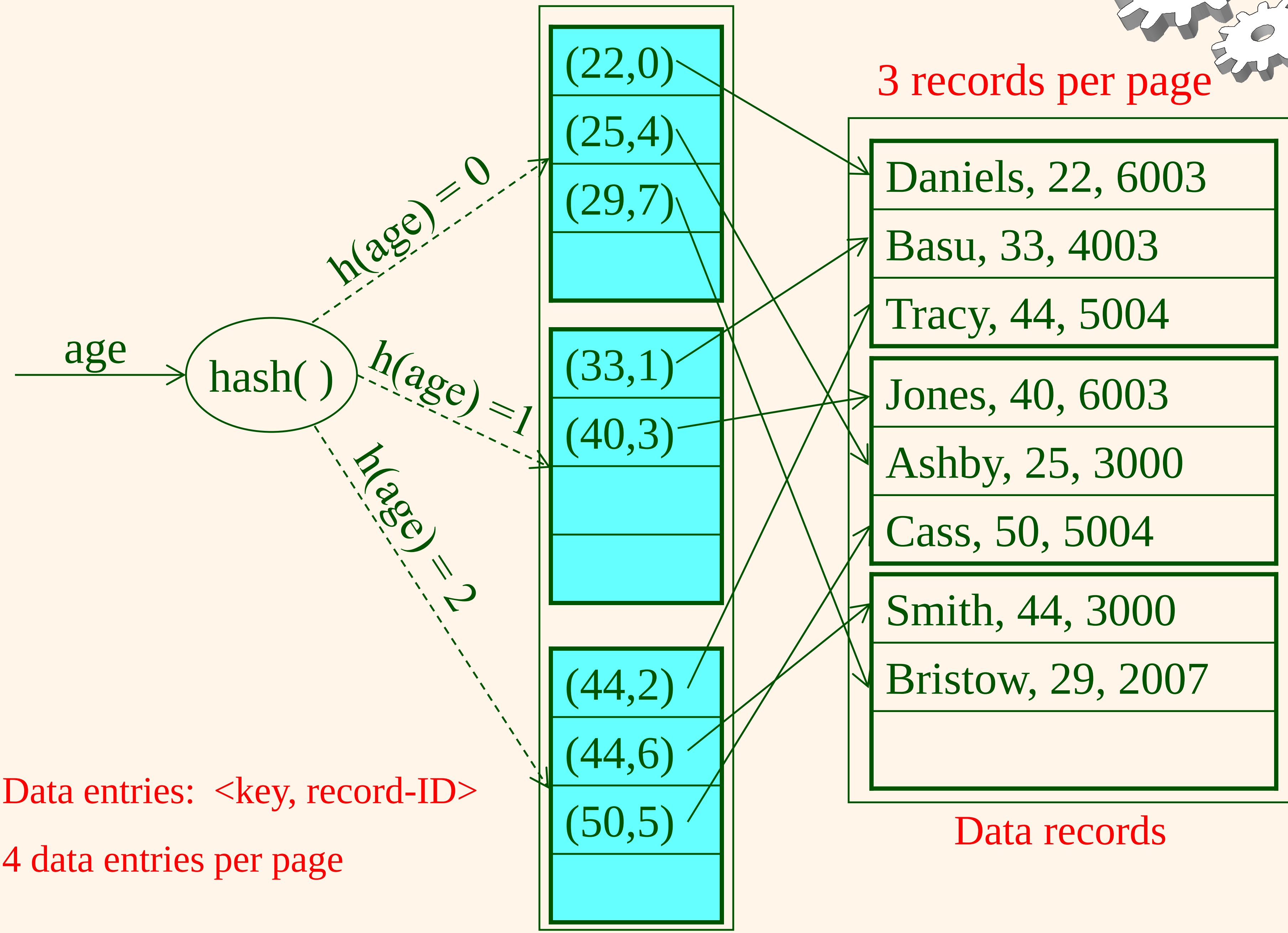
- Best for *equality selections*
- ***Cannot*** support range searches
- Static Hashing
- Dynamic hashing techniques
  - Extendible Hashing
  - Linear Hashing

Hash-Based Indexes

- Hash function
  - works on *search key* field (e.g. age) of record *r*
  - **h(*k*) mod *N*** = bucket to which data entry with key ***k*** belongs (a *bucket* is a page containing data entries)
    - ***N***: the number of buckets in the index
    - **h(*k*) = *a* × *k* + *b*** usually works well, ***a*** and ***b*** are constants
  - Should distribute hashed values *uniformly* over 0 to N-1 buckets



An Example

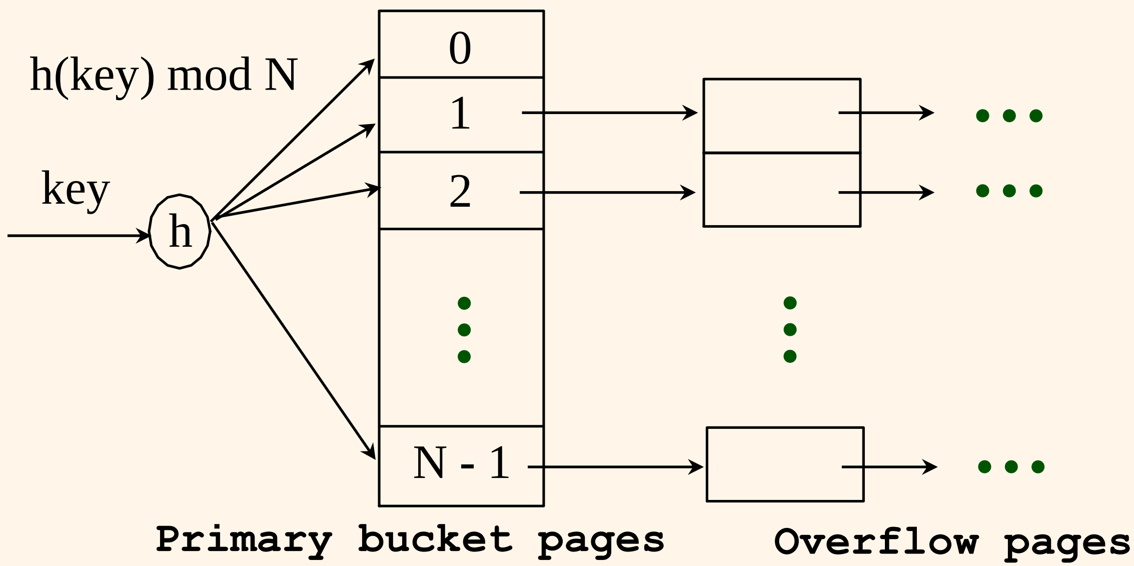


Data entries: <key, record-ID>

4 data entries per page

Static Hashing

- Buckets contain *data entries* (<key value, record-ID>).
- Each bucket takes one fixed primary page.
- Additional overflow pages are allocated if needed.

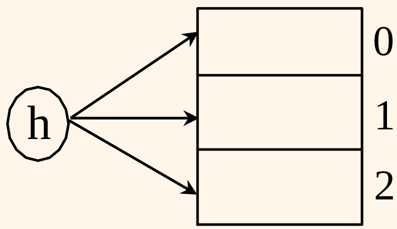


Example

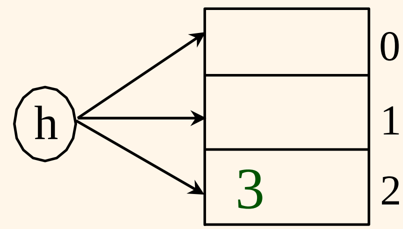
- Each page (bucket) can store at most 8 bytes
  - Let N be the number of *primary buckets*
- Let hash function be $h(k) = a \times k + b$
  - a=1 and b=2, respectively
  - bucket ID=$h(k)$ mod N
- We need to update the hash index based on the following sequence of keys (Each key value is a 4-byte integer)
  - (Insert 3), (Insert 7), (Insert 2), (Insert 12), (Insert 17), (Insert 4), (Insert 10), (Delete 2)

Example *(N=3, Bucket ID=k + 2 mod 3)*

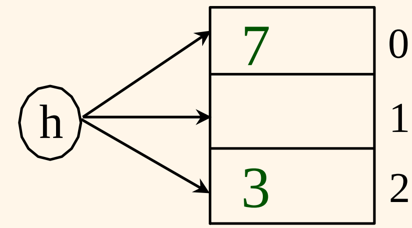
Step 0 (Initially)



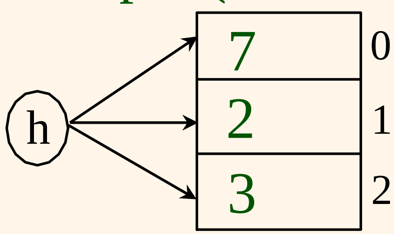
Step 1 (Insert 3)



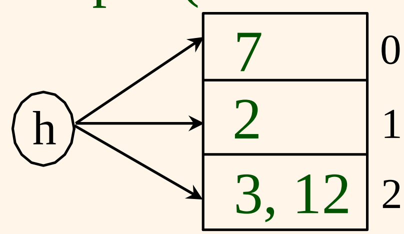
Step 2 (Insert 7)



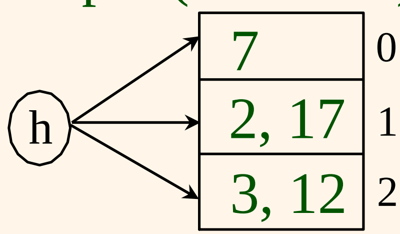
Step 3 (Insert 2)



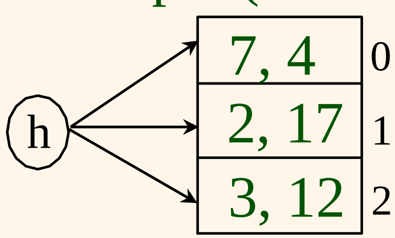
Step 4 (Insert 12)



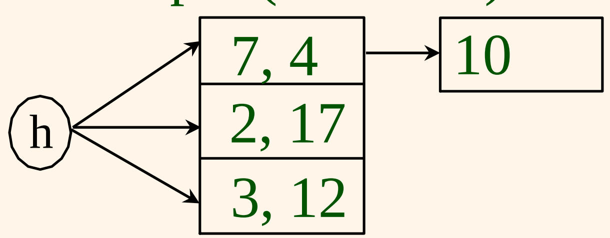
Step 5 (Insert 17)



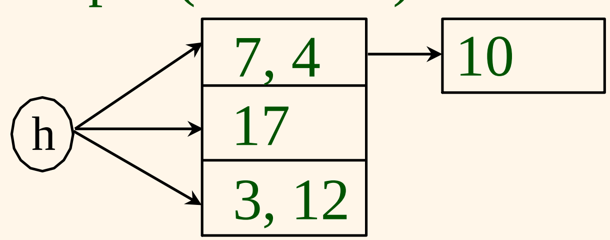
Step 6 (Insert 4)



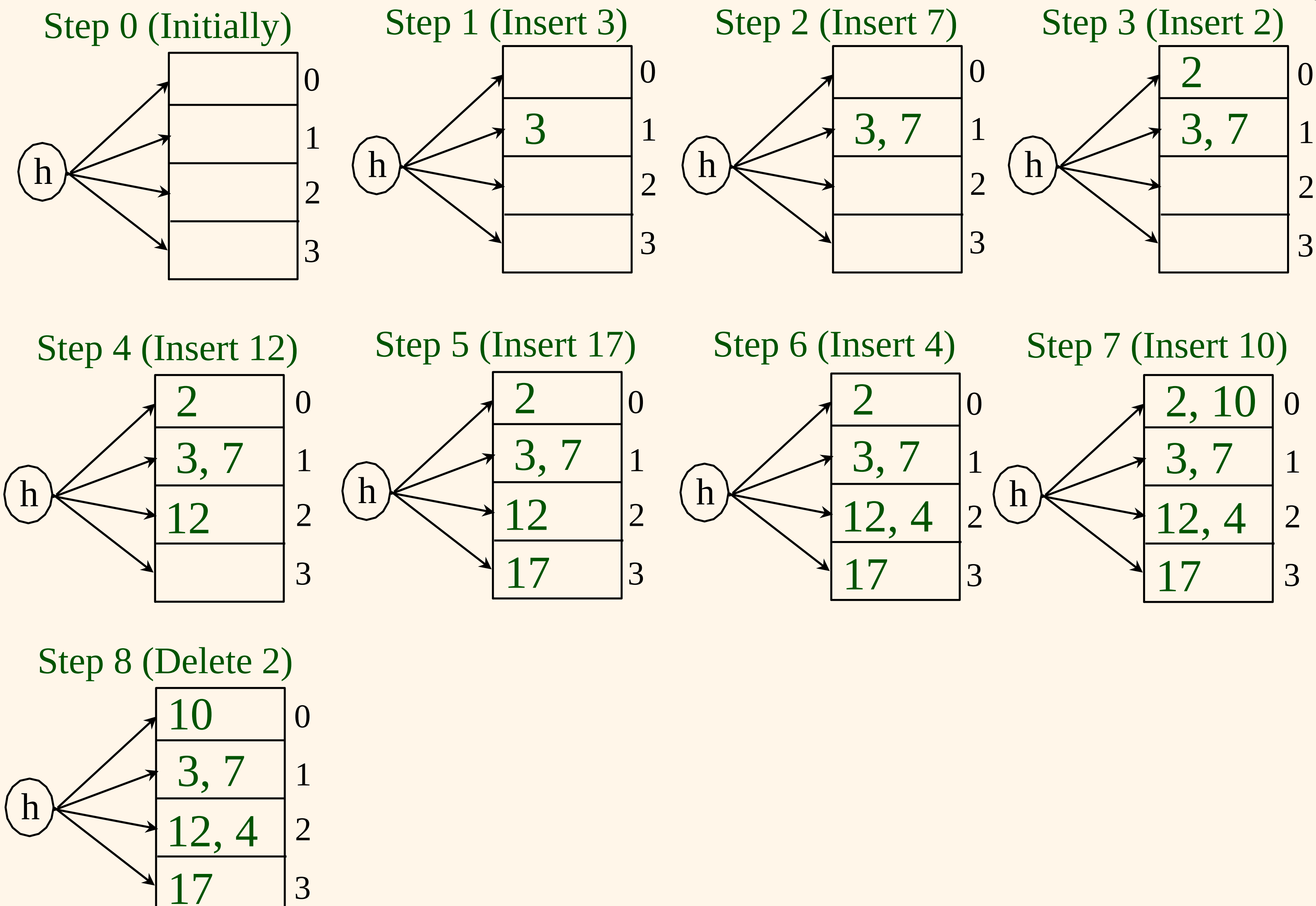
Step 7 (Insert 10)



Step 8 (Delete 2)



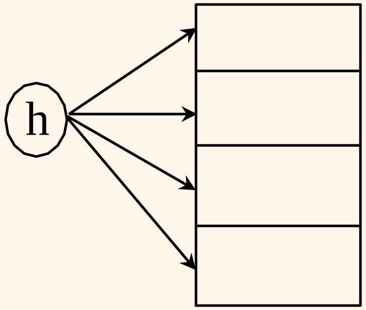
Example (*N*=4, *Bucket ID*=*k* + 2 *mod* 4)



Question 1

- Given a hash function with a = 1, b = 2, and N = 4. Suppose that each page can store at most 7 bytes. Draw the static hashing-based index structure at each step when the following activities occur (4 bytes per integer).

    1. (Insert 3)
    2. (Insert 7)
    3. (Insert 2)
    4. (Insert 12)
    5. (Insert 17)
    6. (Insert 4)
    7. (Insert 10)
    8. (Delete 2)



Static Hashing

- Number of buckets fixed (Hard to determine N)
  - File grows (records are inserted into the relation)
    - Long overflow chains can develop (many overflow pages)
    - Performance degrade
  - File shrinks (records are deleted from the relation)
    - Space wastage (leading to many empty positions)
- To improve
  - Re-build the index by re-hashing files periodically
    - Computationally expensive!
  - *Dynamic hashing techniques*

Extendible Hashing

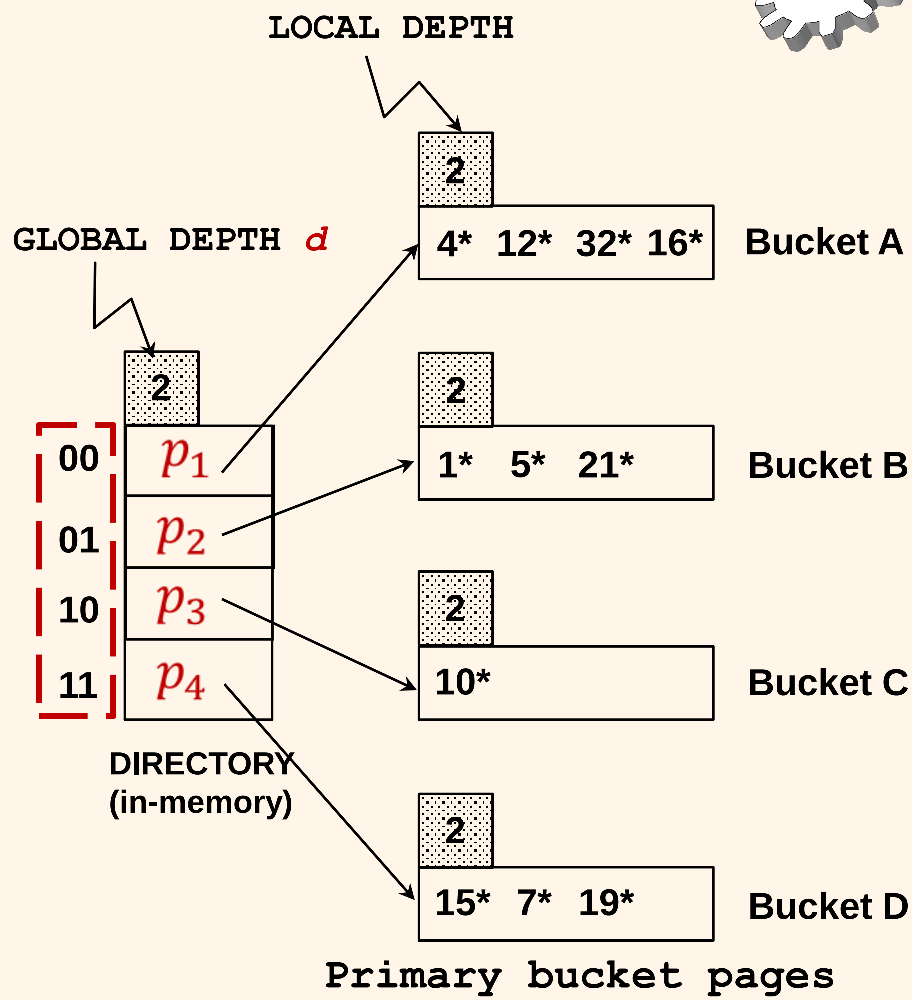
- Bucket (primary page) becomes full
  - Re-organize index by *doubling* the number of buckets
    - Reading and writing all pages is expensive!

- *Idea*: use *directory of pointers to buckets*
  - Double number of buckets by *doubling the directory*
  - Splitting the bucket that *overflowed*
  - Cheaper since directory much *smaller* than index file
  - *No overflow page*!

Trick lies in how hash function is adjusted!

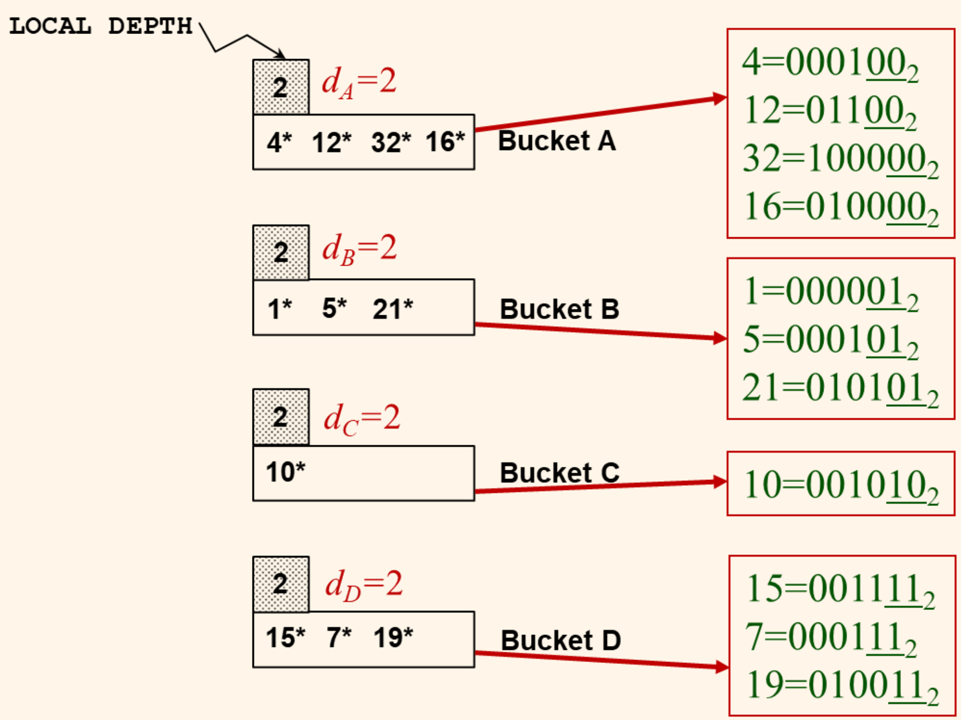
Extendible Hashing

- Maintain a *directory* of pointers $\{p_1, p_2, p_3, \cdots\}$ to buckets
  - The directory has 2^d pointers where d is the *global depth*
- Each pointer p_i in the directory is associated with d bits as its ID
  - E.g., $0=00_2, 1=01_2, 2=10_2, 3=11_2$
- Each pointer in the directory is *only* linked to *one* primary bucket (no overflow pages) but each primary bucket can be linked by *multiple* directory pointers
  - *Directory pointer ID is used as the bucket ID*
  - Link to which one?



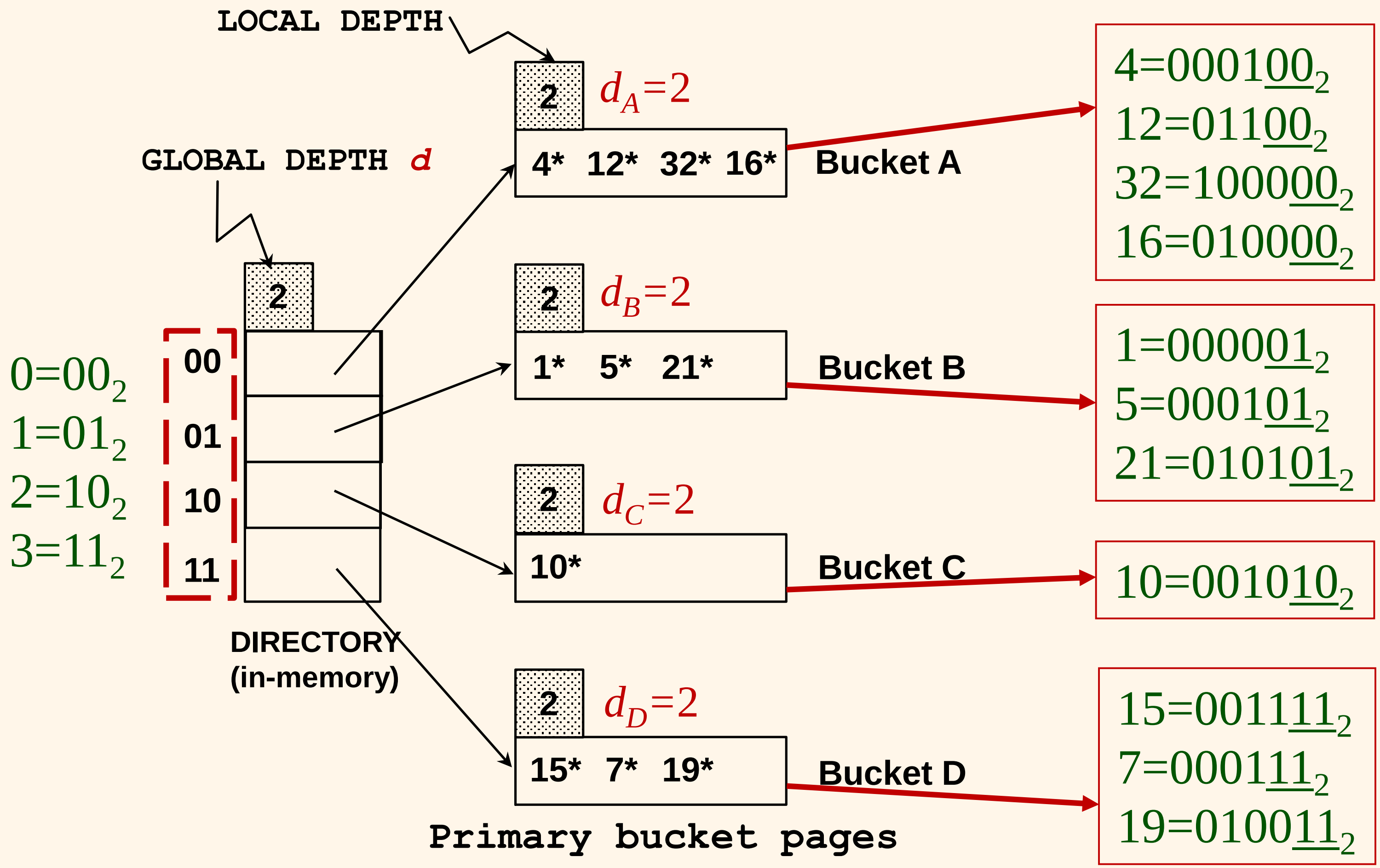
Extendible Hashing (h(k)=k)

- *Local depth* d_A is the actual number of bits that is used for identifying each bucket A
  - Each bucket A has its local depth d_A
  - The last d_A bits of all hashed key values h(k) in bucket A are the same
  - It must be the same or less than the global depth, i.e., $d_A \le d$
  - It determines whether we need to split/merge the bucket

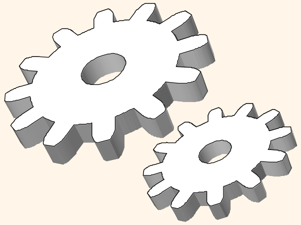


Extendible Hashing (h(k)=k)

❖ The last d_A bits of all hashed key values h(k) in each bucket A are the same, which are equal to the last d_A bits of its corresponding directory pointer's ID



Why using the last d bits

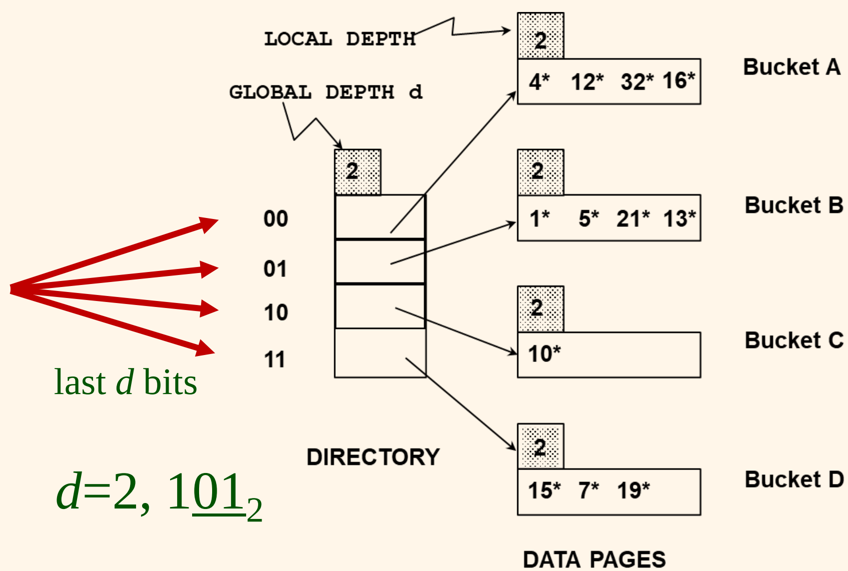


- Recall that bucket-ID = h(k) mod N
- Let N=2^d=# of directory pointers
  - Directory pointer ID = bucket-ID = h(k) mod 2^d
  - The binary format of (h(k) mod 2^d) is equal to the last d bits of h(k)
  - We can take the last d bits of h(k) as its directory pointer ID/bucket ID
- Example: assume h(k)=k+2 and d=2
  - k+2=32=$1000\underline{00}_2$, 32 mod 4=0=$\underline{00}_2$
  - k+2=21=$0101\underline{01}_2$, 21 mod 4=1=$\underline{01}_2$
  - k+2=15=$0011\underline{11}_2$, 15 mod 4=3=$\underline{11}_2$
  - k+2=10=$0010\underline{10}_2$, 10 mod 4=2=$\underline{10}_2$
  - k+2=12=$011\underline{00}_2$, 12 mod 4=0=$\underline{00}_2$

Search with Extendible Hashing

- Given a key value *k*, we need to find the record with this key value
- With the extendible hashing index, it is equivalent to finding bucket containing its corresponding data entry <k, record-ID>
  1) Take last *d* bits of **h**(*k*) and get the directory pointer *p* according to the *d* bits
  2) Follow pointer *p* to get the bucket and the data entry
  3) Get the data record by the record ID in the data entry

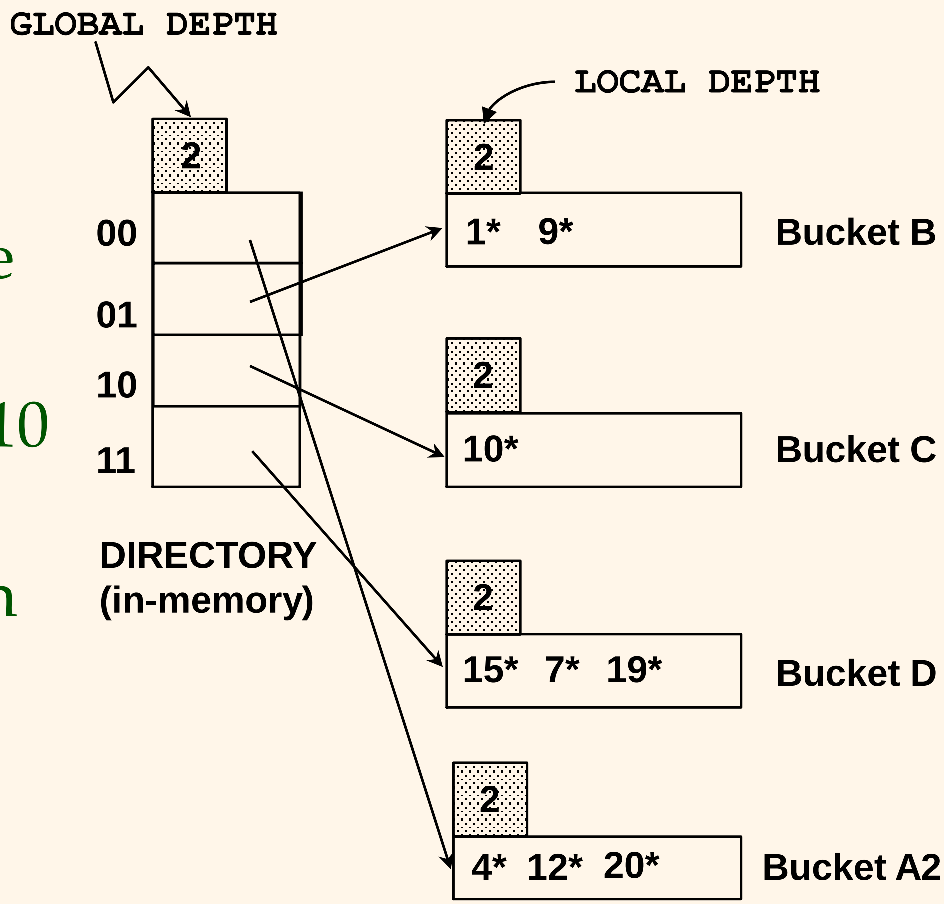
k → **h**(*k*)



5 $\quad \mathbf{h}(k) = k \quad d{=}2,\ 1\underline{01}_2$

Example (Search 10, h(k)=k)*

- 10 can be represented as the binary number 1010_2
- The last d=2 bits of $h(10)$ is 10
- Find the bucket C, which contains the data entry with key=10

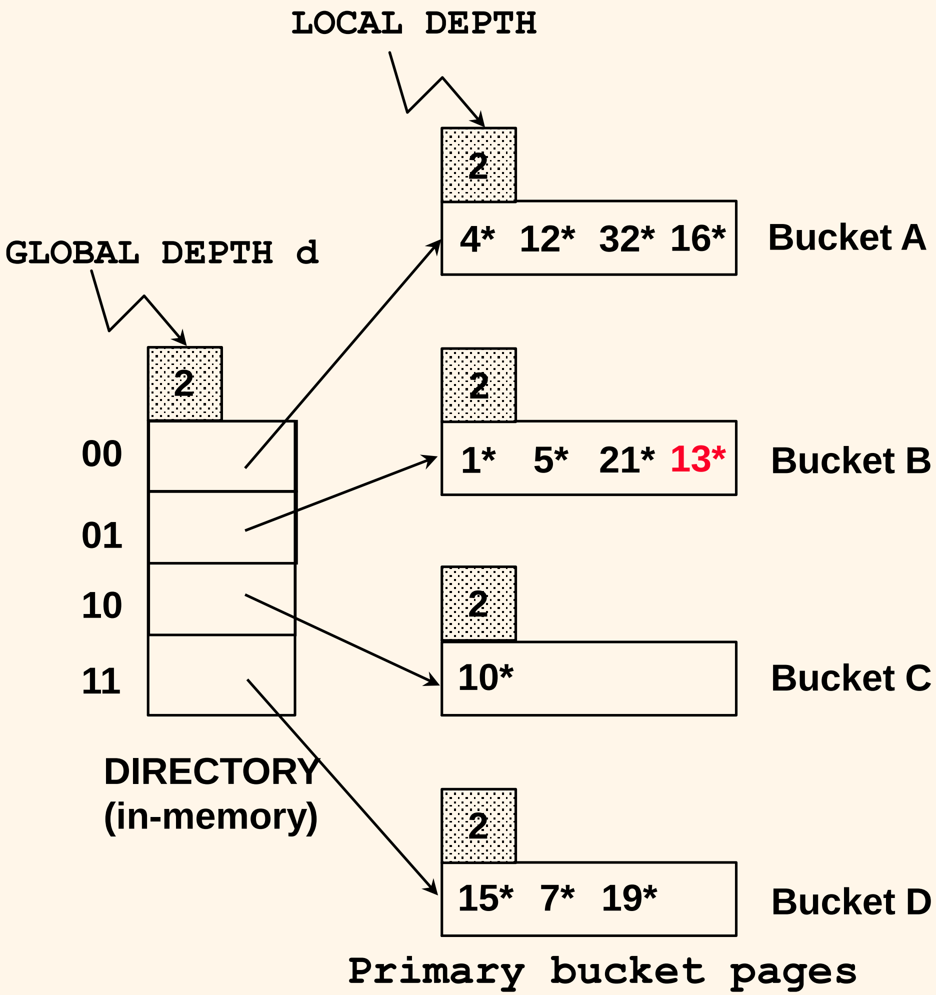


Question 2

- Consider the following numbers v = 18, 25, 37, and 77.
  a) Represent these numbers using the binary format.
  b) Let N = 2. Find v mod N in terms of binary number for each v in a).
  c) Repeat b) for N = 4.
  d) Repeat b) for N = 8.

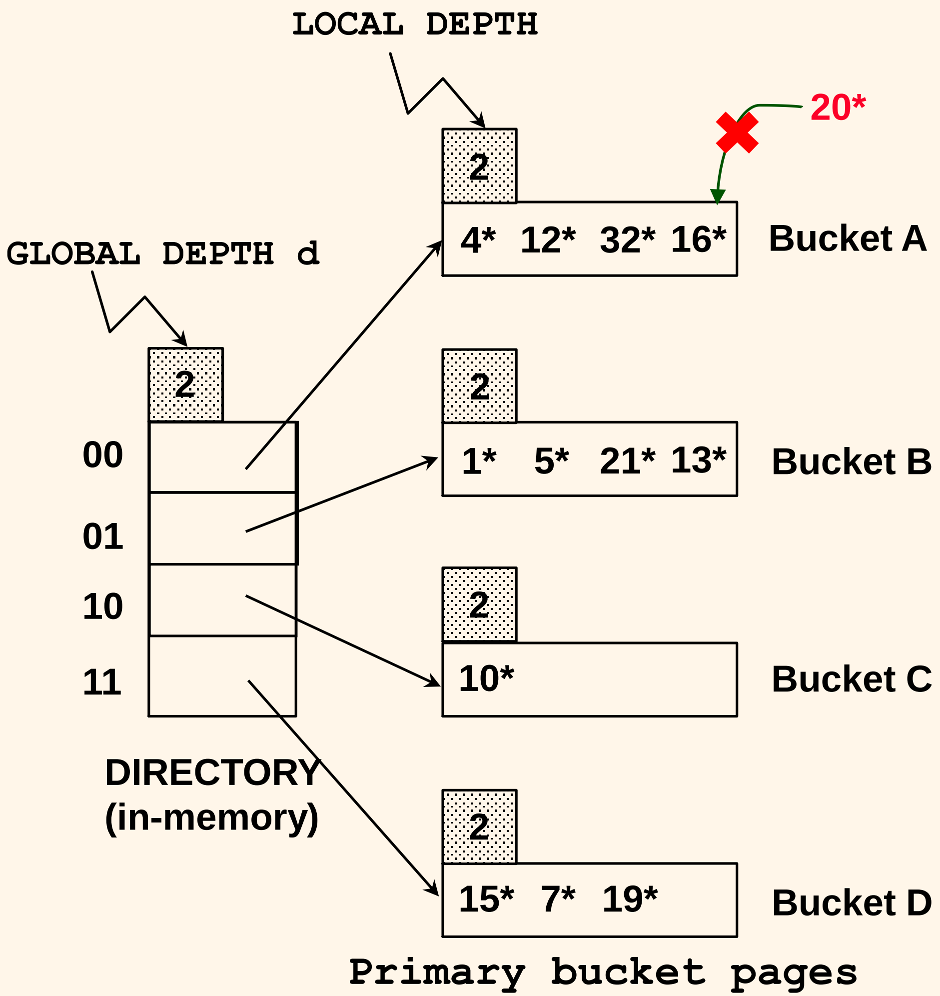
Example (Insert 13, h(k)=k)*

- 13 can be represented as the binary number 1101_2
- The last d=2 bits of h(13) is 01
- Insert 13* in the bucket B since this bucket still has space



Example (Insert 20, h(k)=k)*

- 20 can be represented as the binary number 10100_2
- The last d=2 bits of h(20) is 00
- Bucket A is fully occupied. Cannot directly insert 20* into the bucket A
  - How to solve this?



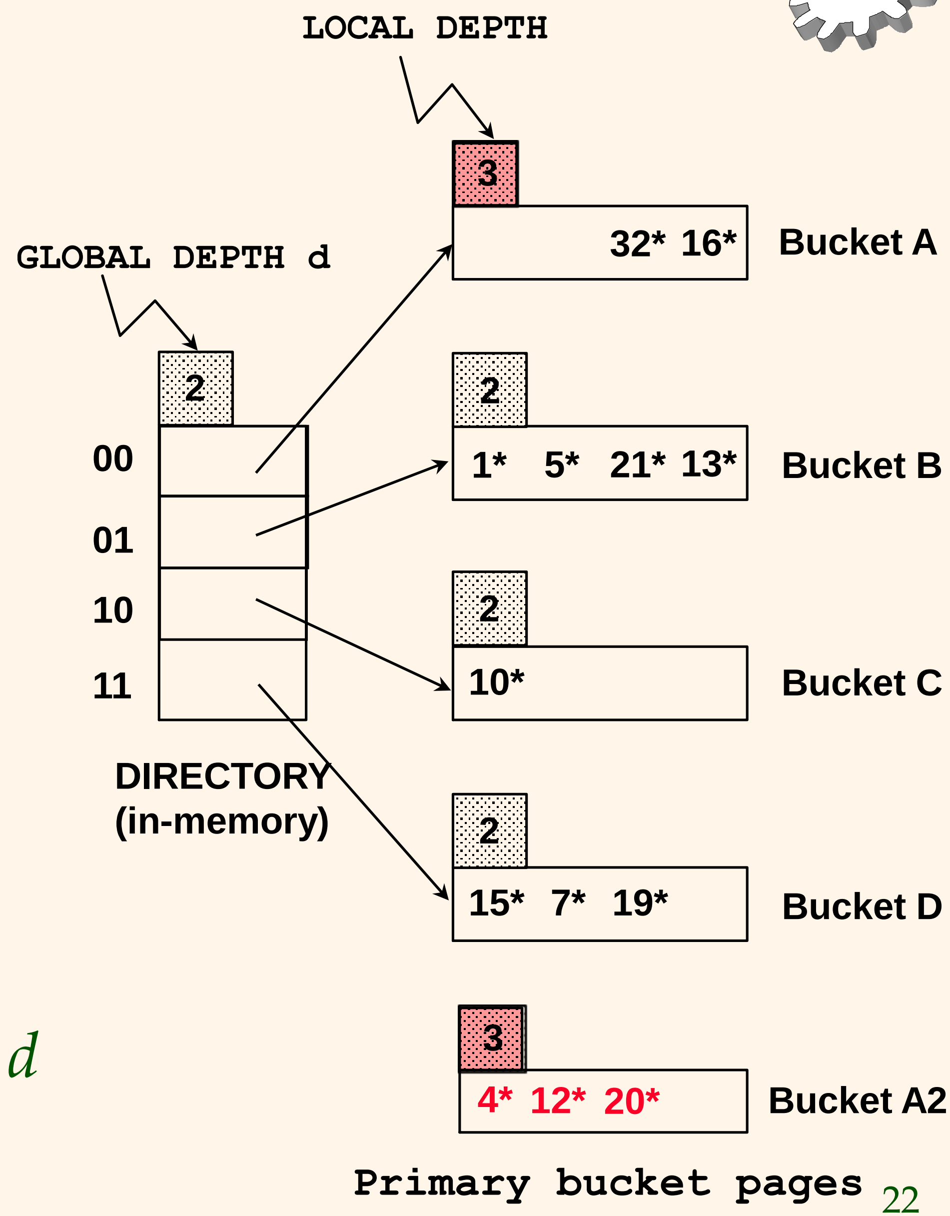
Example (Insert 20, h(k)=k)*

- Split the bucket A
  - Allocate the new page (bucket A2).
  - Redistribute the key values into the bucket A and bucket A2 (**How?**).

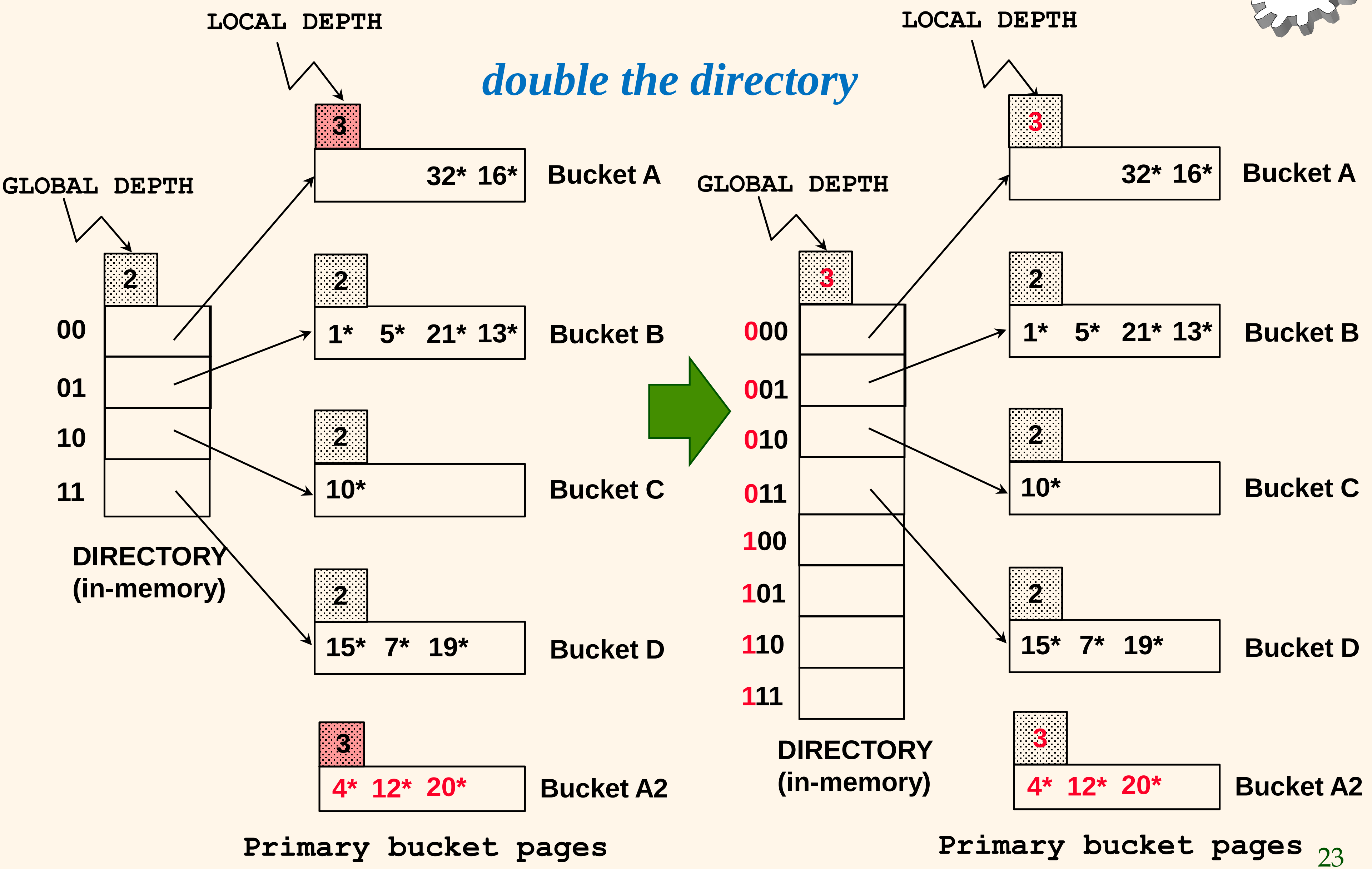
    Using the last d_A+1=3 bits

$32=100\underline{000}_2$ $16=010\underline{000}_2$	$4=000\underline{100}_2$ $12=01\underline{100}_2$ $20=010\underline{100}_2$

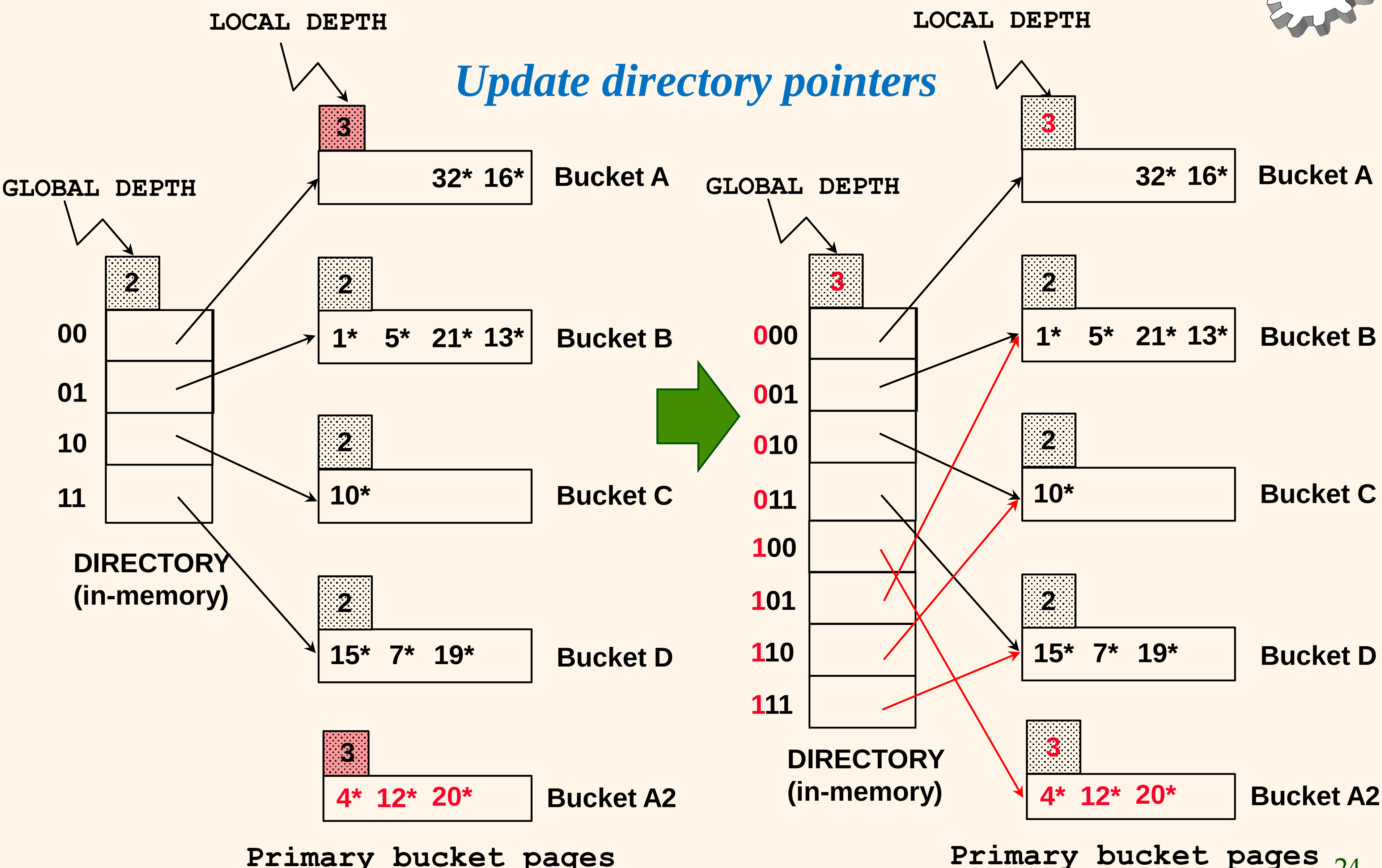
  - Increase the *local depth* of the bucket A and bucket A2 by 1
- Local depth > Global depth *d* (**violate the requirement!**)



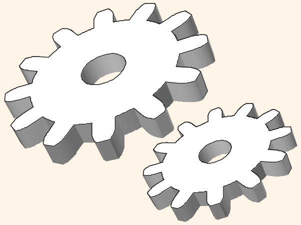
Example (Insert 20*, h(k)=k)



Example (Insert 20*, h(k)=k)



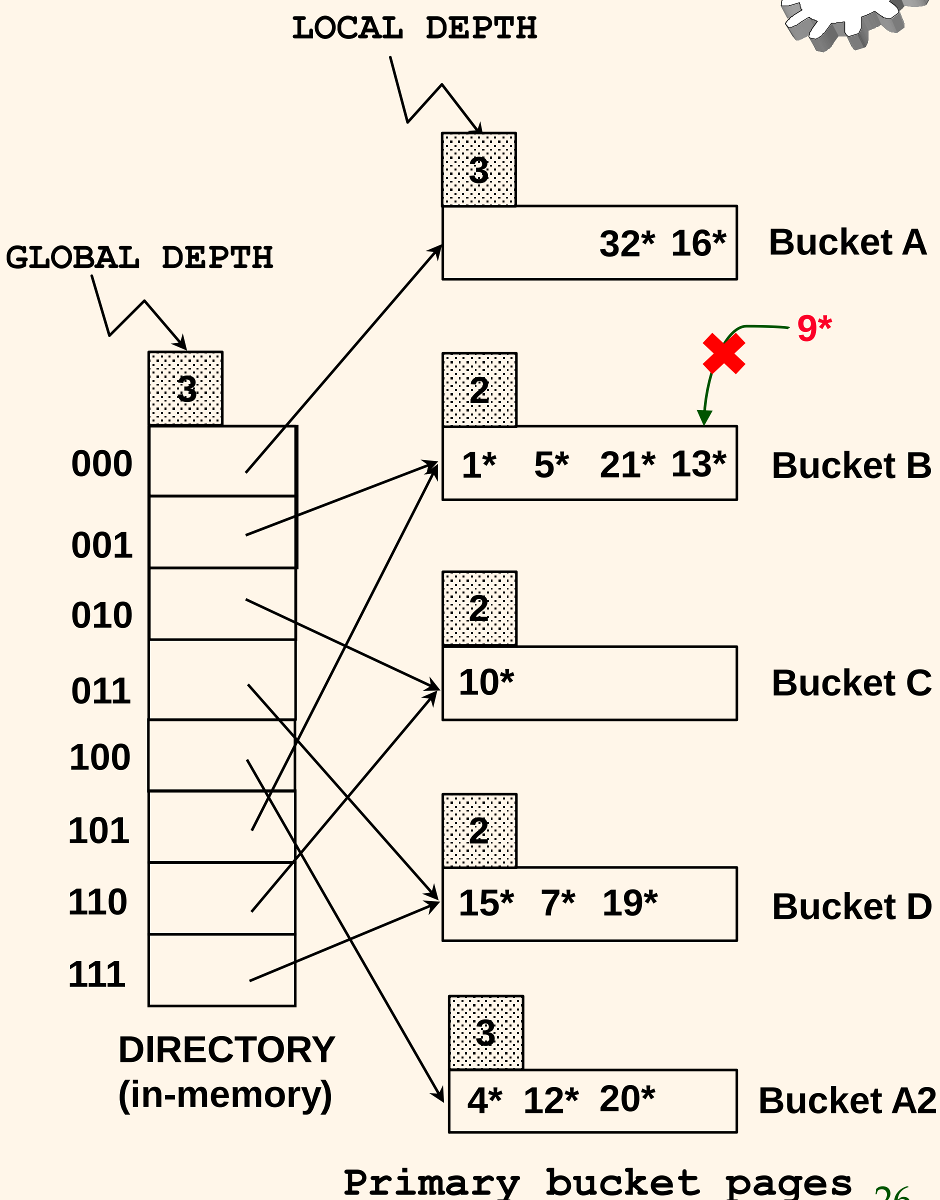
Example (Insert 20, h(k)=k)*

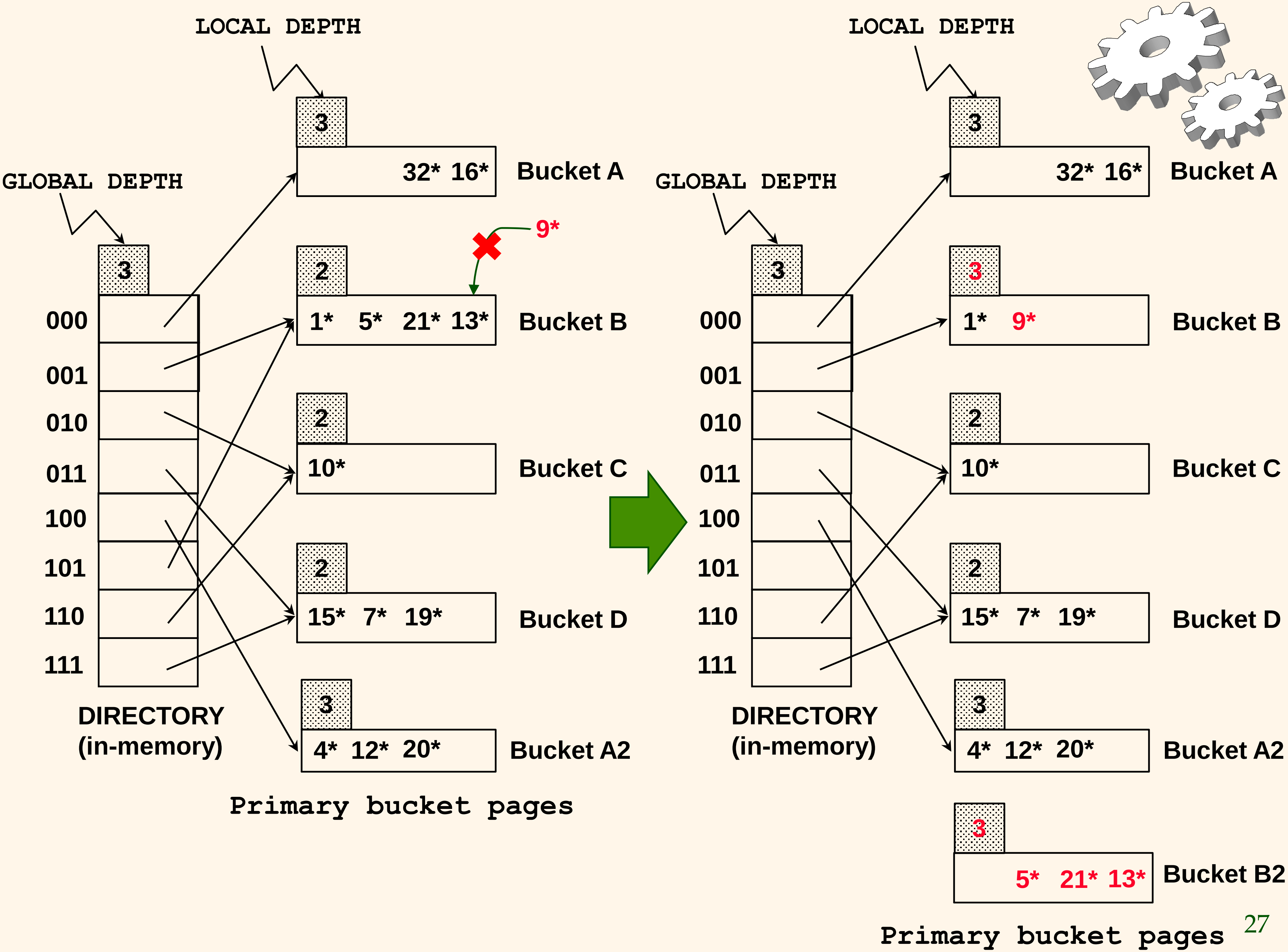


- How to deal with the bucket A which is fully occupied?
  - Split the bucket A into 2 buckets A and A2
  - Redistribute the data entries as per the last d_A+1 bits
  - Increase the local depths of these two buckets by 1, respectively
- How to deal with local depth > global depth *d*?
  - Double the number of pointers in the directory
  - Increase the global depth *d* by 1
  - Link the directory pointers to their corresponding buckets

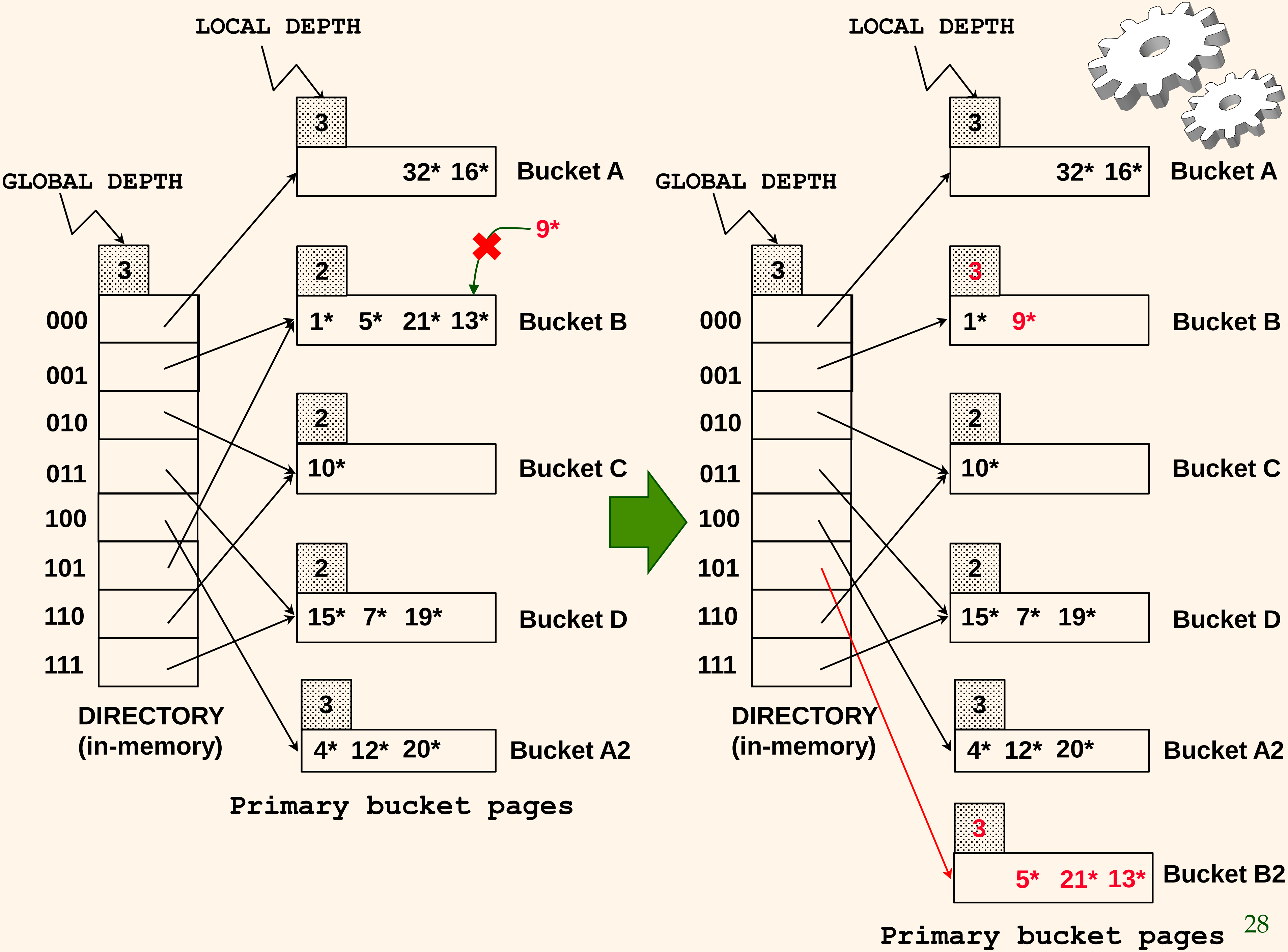
Example (Insert 9*, h(k)=k)

- 9 can be represented as the binary number 1001_2
- The last d=3 bits of h(9) is 001
- Bucket B is fully occupied. Cannot directly insert 9* into the bucket B





LOCAL DEPTH
GLOBAL DEPTH
3
3
32* 16*
Bucket A
2
1* 5* 21* 13*
Bucket B
9*
2
10*
Bucket C
2
15* 7* 19*
Bucket D
3
4* 12* 20*
Bucket A2
000
001
010
011
100
101
110
111
DIRECTORY
(in-memory)
Primary bucket pages
LOCAL DEPTH
GLOBAL DEPTH
3
3
32* 16*
Bucket A
3
1* 9*
Bucket B
2
10*
Bucket C
2
15* 7* 19*
Bucket D
3
4* 12* 20*
Bucket A2
3
5* 21* 13*
Bucket B2
000
001
010
011
100
101
110
111
DIRECTORY
(in-memory)
Primary bucket pages


LOCAL DEPTH
GLOBAL DEPTH
3
3
32* 16*
Bucket A
2
1* 5* 21* 13*
Bucket B
9*
000
001
010
011
100
101
110
111
2
10*
Bucket C
2
15* 7* 19*
Bucket D
3
4* 12* 20*
Bucket A2
DIRECTORY
(in-memory)
Primary bucket pages
LOCAL DEPTH
GLOBAL DEPTH
3
3
32* 16*
Bucket A
3
1* 9*
Bucket B
2
10*
Bucket C
2
15* 7* 19*
Bucket D
3
4* 12* 20*
Bucket A2
3
5* 21* 13*
Bucket B2
DIRECTORY
(in-memory)
Primary bucket pages

Example (Insert 9, h(k)=k)*

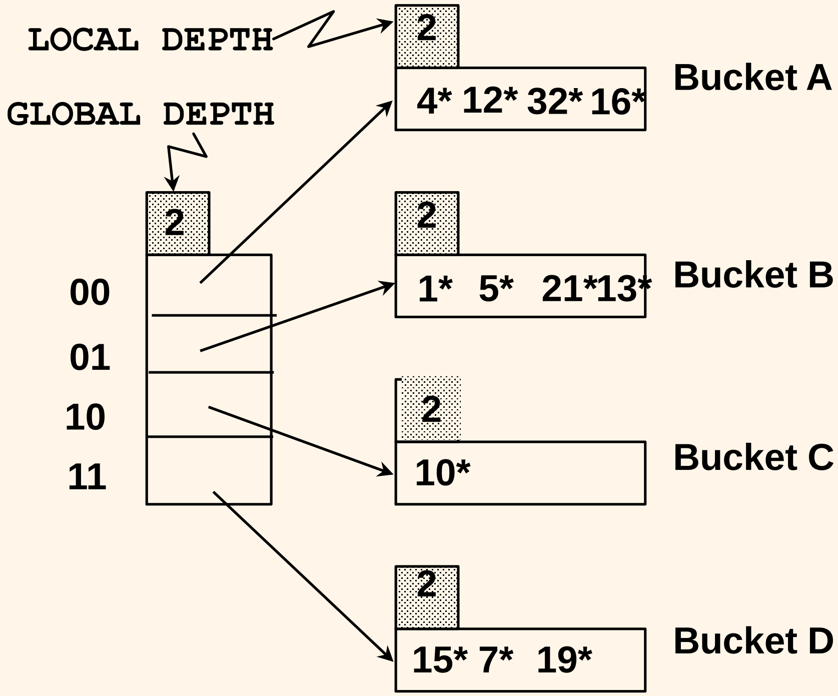
- Local depth ≤ Global depth *d*
  - No need to double the number of pointers in the directory (No need to increase *d*)
- The bucket B is fully occupied (cannot insert new data entries)
  - Split the bucket B into 2 buckets B and B2
  - Redistribute the data entries as per the last d_B+1 bits
  - Link the directory pointer with 101 to the bucket B2
  - Increase the local depths of these two buckets by 1, respectively

Insertion in Extendible Hashing

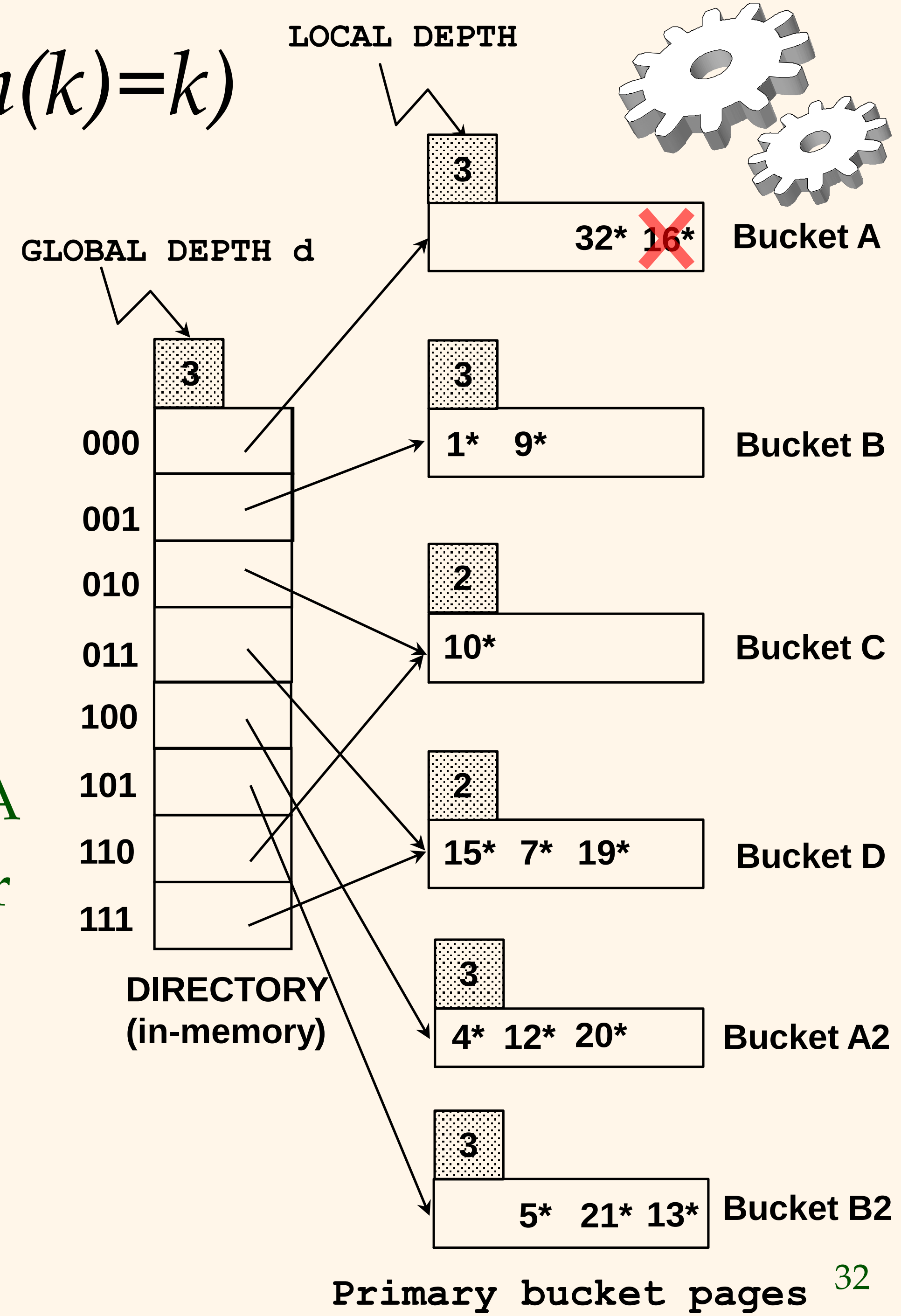
- **Insert**
  - Locate correct bucket (same as Search)
  - If bucket is not full, put data entry there
  - Else
    - Split bucket
    - Increase local depth
    - If local depth > global depth
      - Double directory
      - Increase global depth
    - Fix pointers in the directory

Question 3

- ❖ Given below extendible hashing index, show the updated index after inserting 9* and 20*



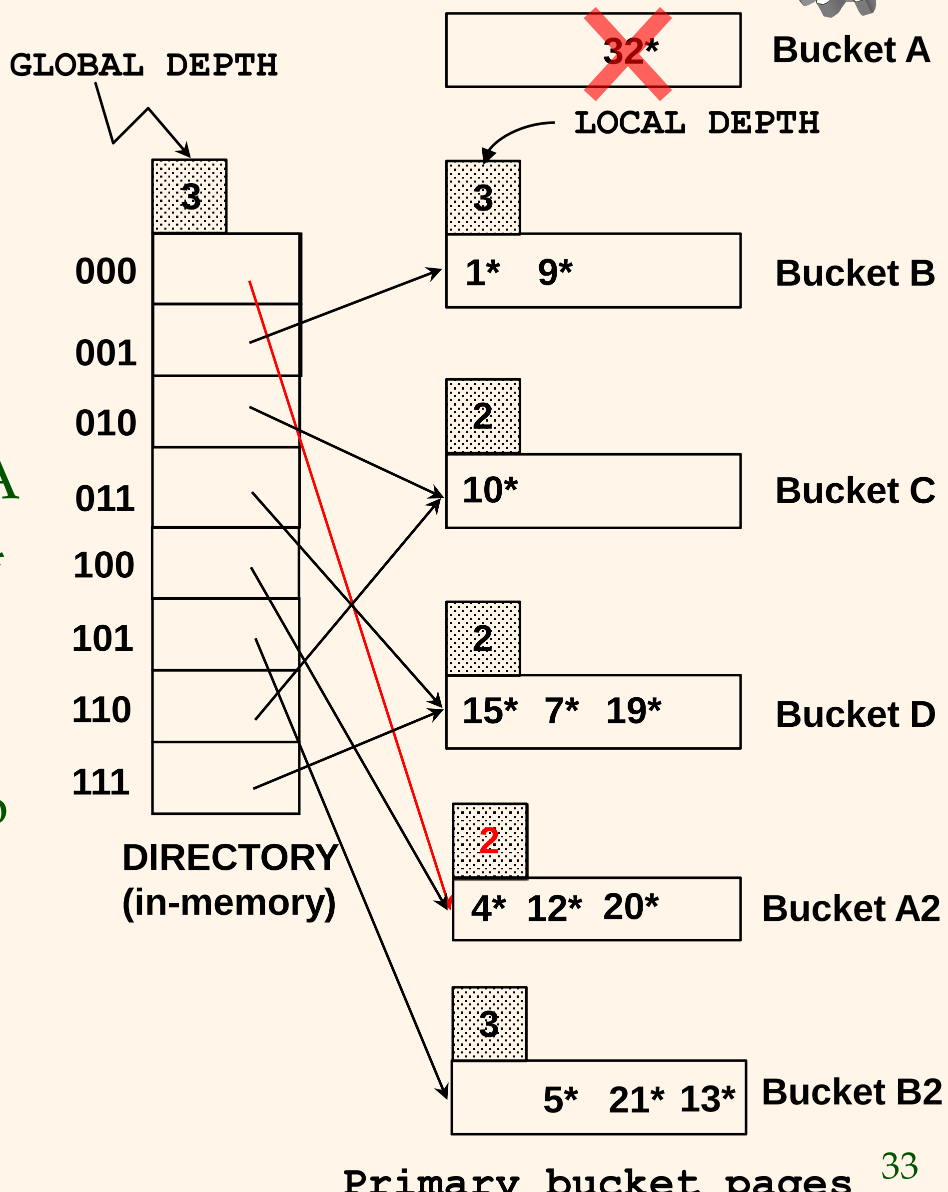
Example (Delete 16, h(k)=k)*



- 16 can be represented as the binary number 10000_2
- The last d=3 bits of $h(16)$ is 000
- Remove 16* in the bucket A
- Bucket A is not empty after 16* is removed

Example (Delete 32, h(k)=k)*

- 32 can be represented as the binary number 100000_2
- The last d=3 bits of $h(32)$ is 000
- Remove 32* in the bucket A
- Bucket A is empty after 32* is removed
  - Remove the bucket A
  - Move the pointer from 000 to bucket A2 (its split image)
  - Decrease the local depth of the bucket A2 by 1



Example (Delete 32, h(k)=k)*

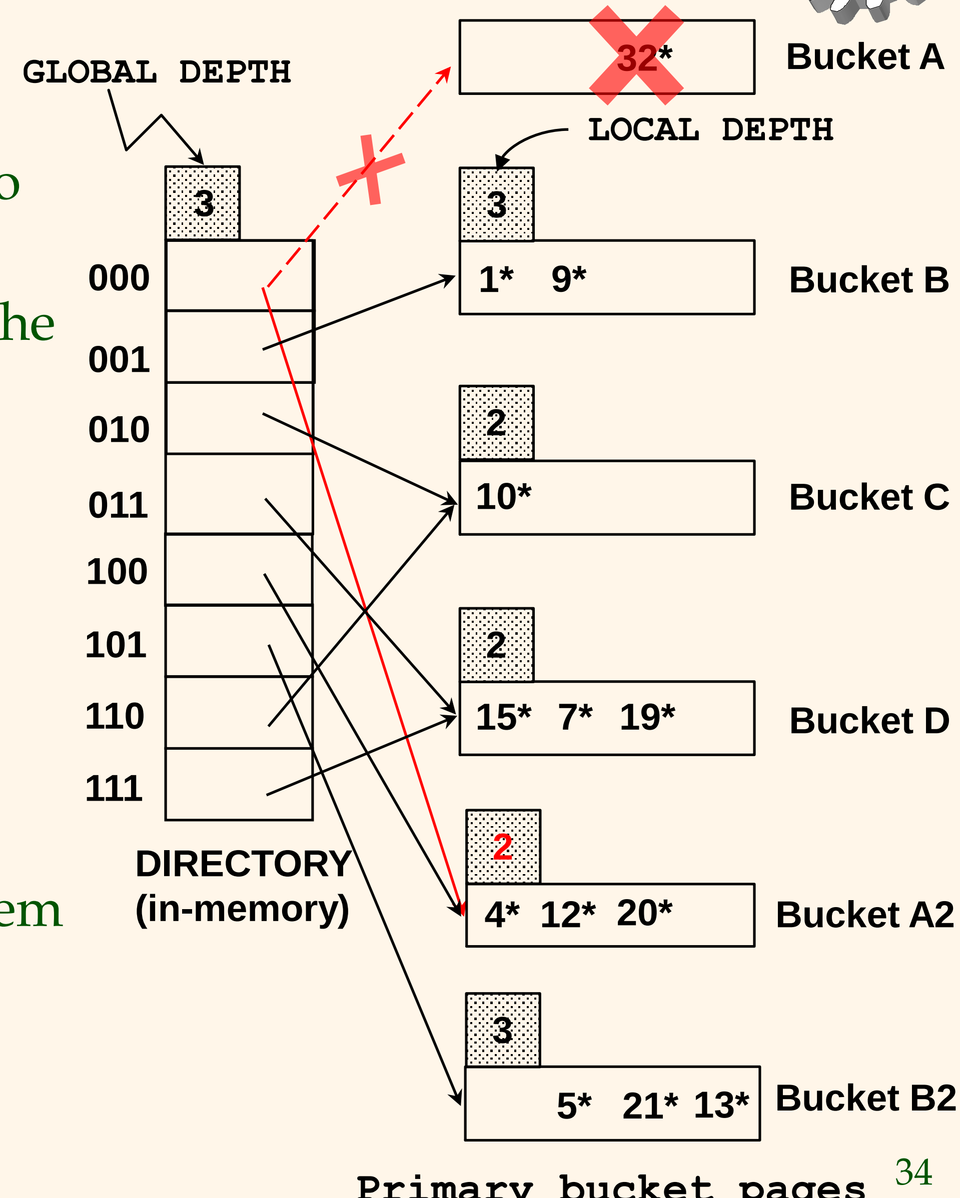
- Why?
  - Move the pointer from 000 to bucket A2 (its split image)
  - Decrease the local depth of the bucket A2 by 1
- Before removing 32*
  - Need 3 bits to locate them

$32=100\underline{000}_2$

$4=000\underline{100}_2$
$12=01\underline{100}_2$
$20=010\underline{100}_2$

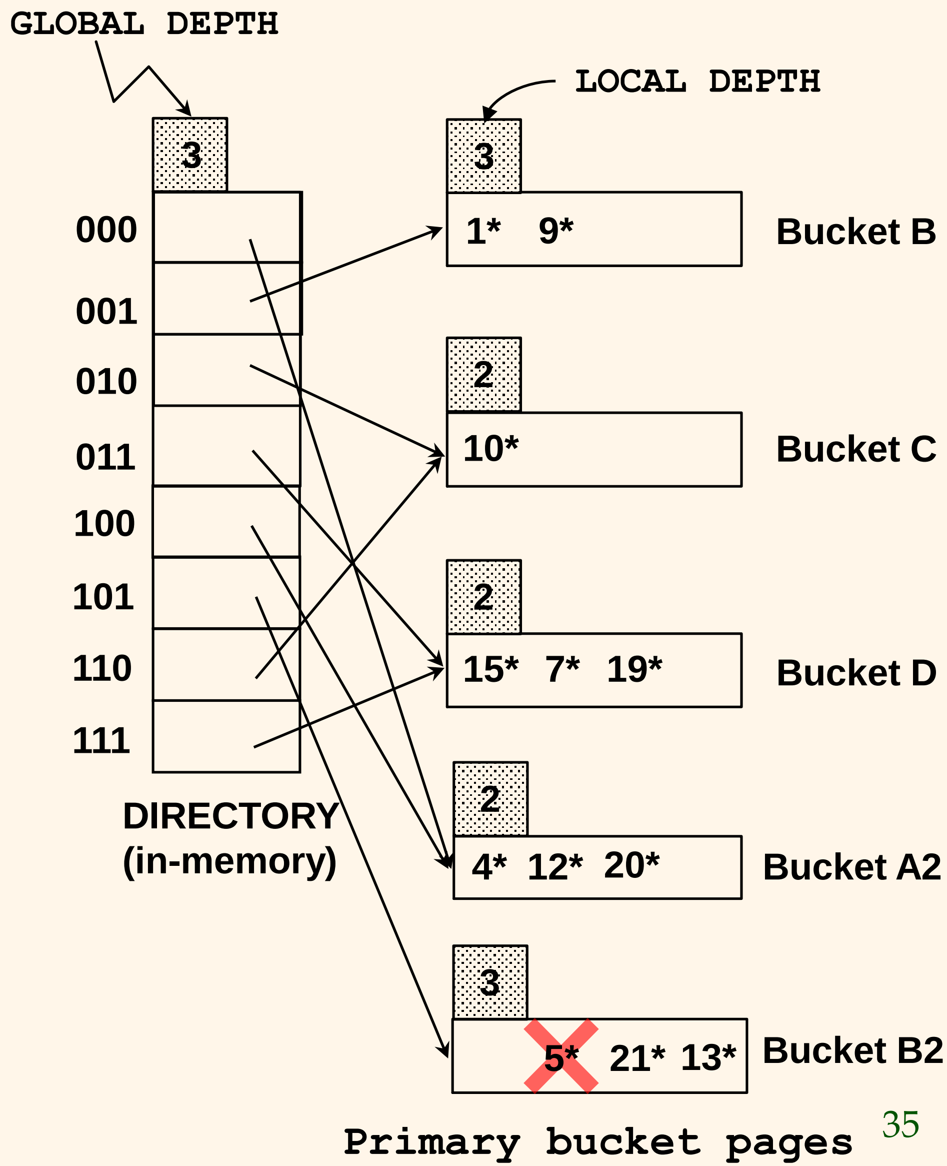
- After removing 32*
  - Only need 2 bits to locate them

$4=0001\underline{00}_2$
$12=011\underline{00}_2$
$20=0101\underline{00}_2$



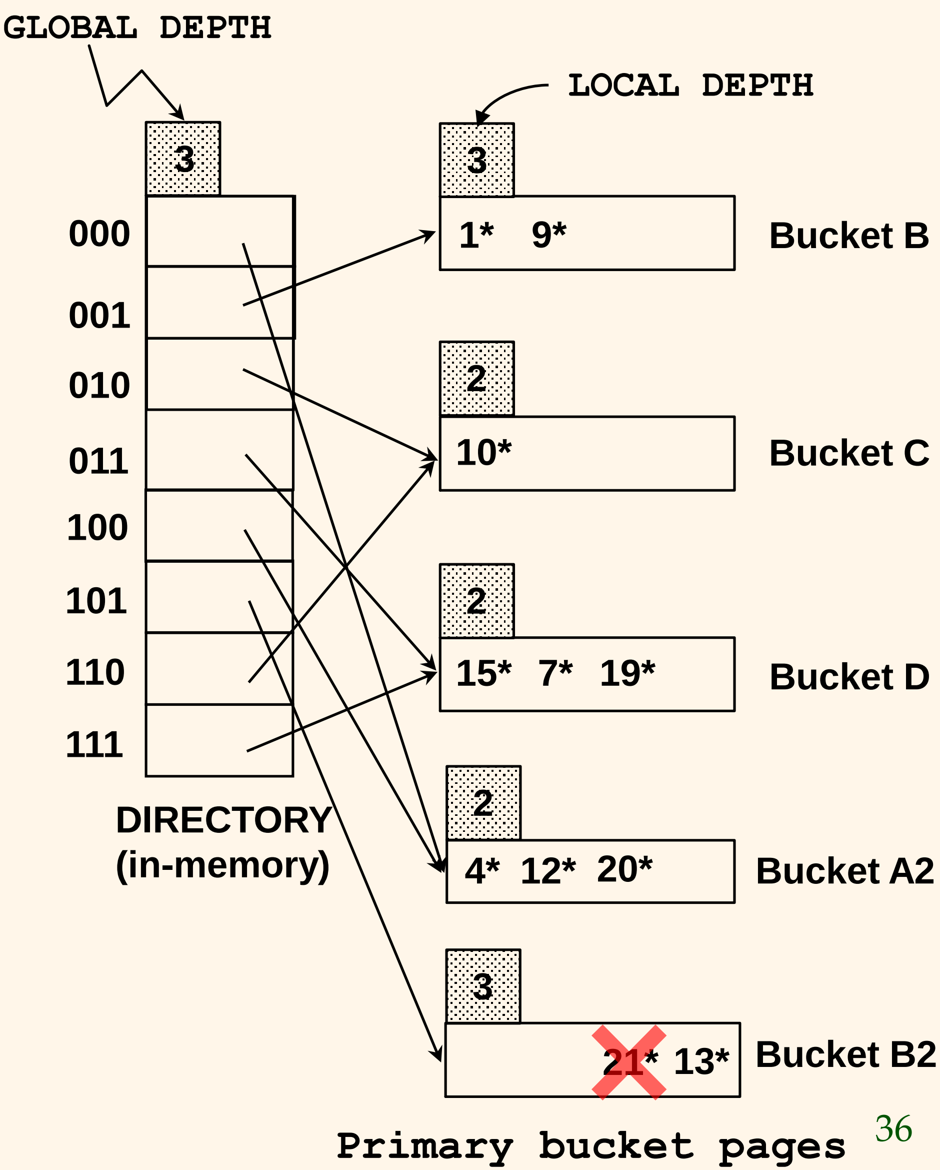
Example (Delete 5, h(k)=k)*

- 5 can be represented as the binary number 101_2
- The last d=3 bits of h(5) is is 101
- Remove 5* in the bucket B2
- Bucket B2 is not empty after 5* is removed



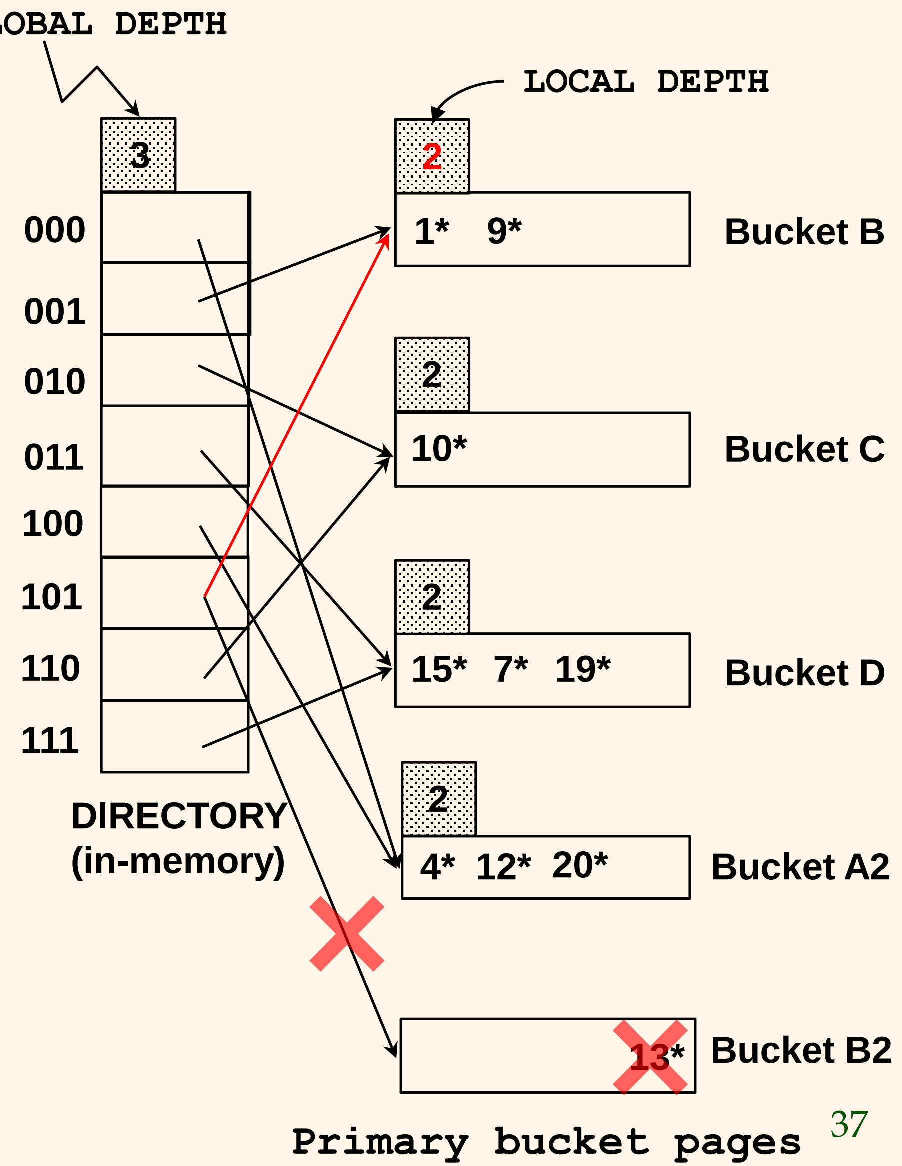
Example (Delete 21*, h(k)=k)

- 21 can be represented as the binary number 10101_2
- The last d=3 bits of $h(21)$ is 101
- Remove 21* in the bucket B2
- Bucket B2 is not empty after 21* is removed



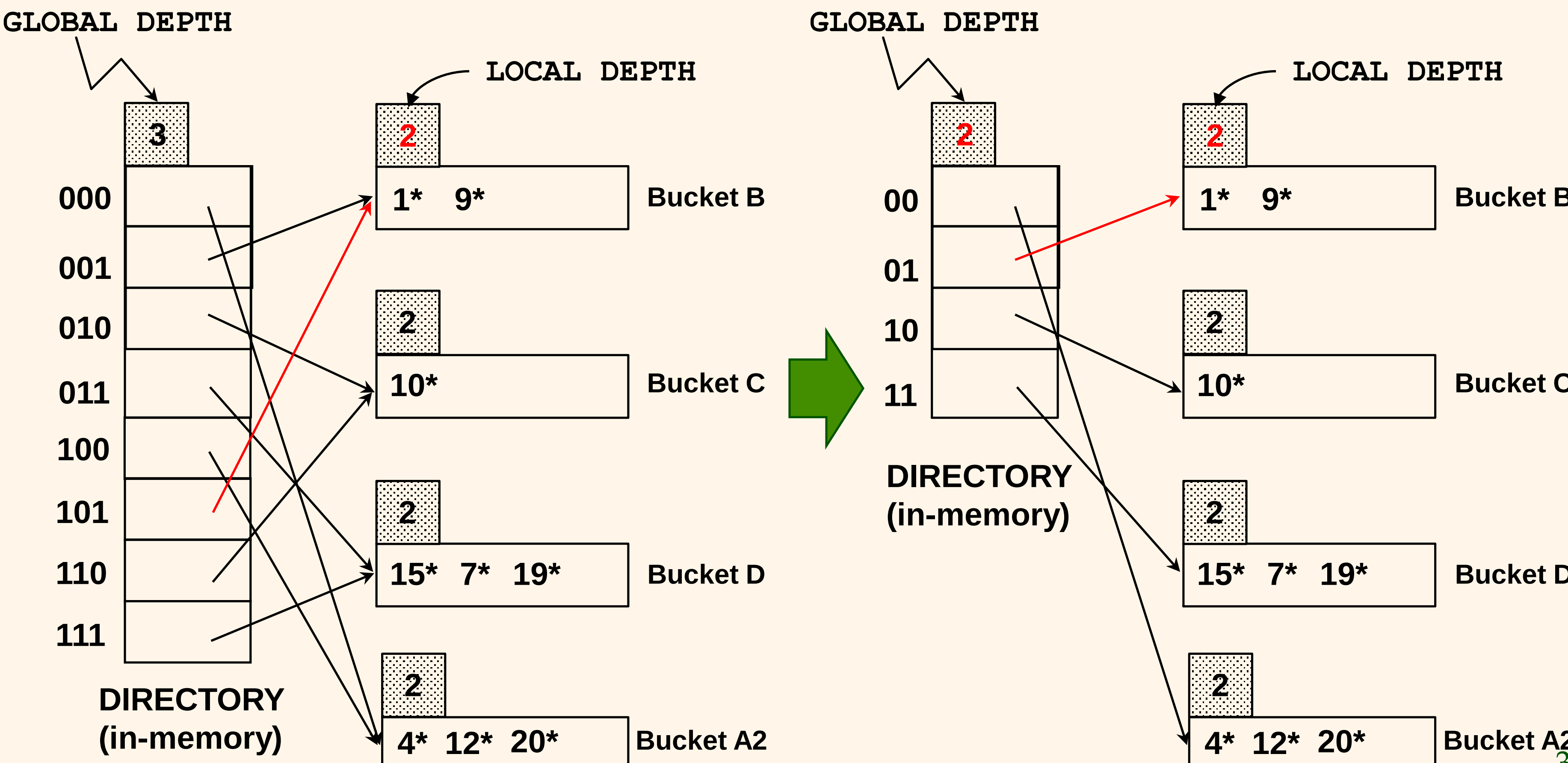
Example (Delete 13, h(k)=k)*

- 13 can be represented as the binary number 1101_2
- The last d=3 bits of h(13) is 101
- Remove 13* in the bucket B2
- Bucket B2 is empty after 13* is removed
  - Remove the bucket B2
  - Move the pointer from 101 to the bucket B (the original bucket)
  - Decrease the local depth of the bucket B by 1
  - And…



Example (Delete 13, h(k)=k)*

- If the local depths of *all buckets* are *d*-1
  - *Halve the size of directory* and *decrease the global depth d by 1*
  - Fix the pointers in the directory



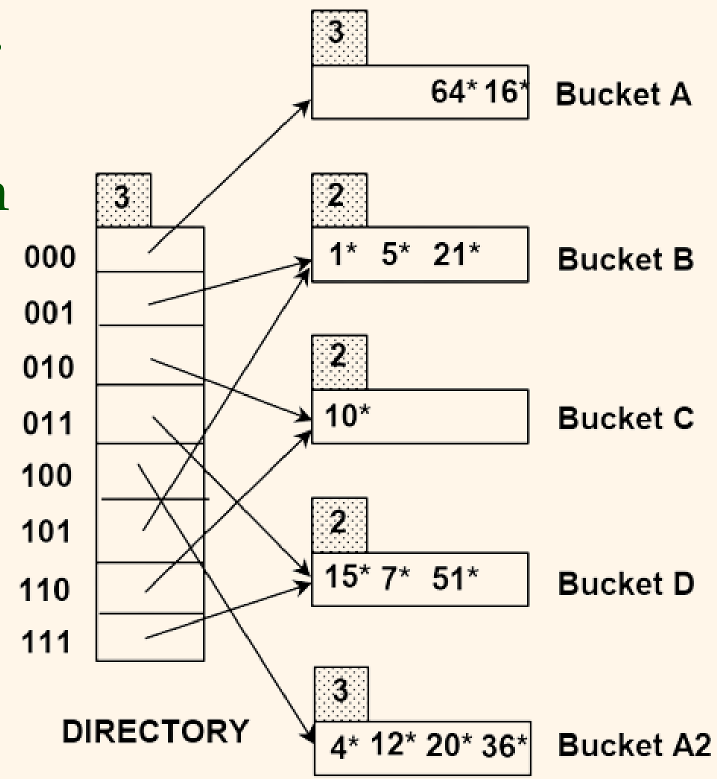
Deletion in Extendible Hashing

- **Delete**
  - Remove data entry from bucket
  - If bucket is empty
    - Merge bucket with its split image (e.g., A and A2, B and B2)
    - Decrease local depth by 1
    - If the local depths of all buckets are *d*-1 (i.e., each directory pointer points to the same bucket as its split image)
      - Halve directory
      - Decrease the global depth *d* by 1
      - Fix the pointers in the directory

Question 4

Hint: looking at the last n bits of a number X is equivalent to looking at the remainder of X mod 2^n.

- Consider the r.h.s. extendible hashing index.
  - a) Show the index after inserting an entry with hash value 68.
  - b) Show the index after inserting entries with hash values 17 & 69 into the *original* index.



Pros of Extendible Hashing

- No need to choose the number of buckets N in advance ☺
- Can lead to the better index structure compared with the static hashing ☺
  - Smaller I/O cost for searching the index structure (there is no long overflow chain)
    - If directory fits in memory, equality search answered with one disk page access; else two.
  - No need to reconstruct the index structure periodically.

Cons of Extendible Hashing

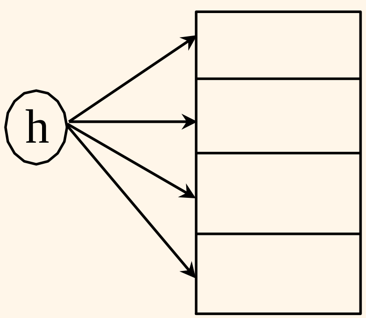
- Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
- Collisions, or multiple entries with same hash value cause problems!

Summary

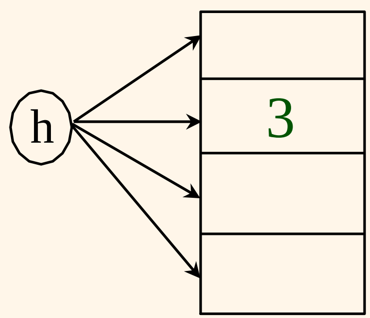
- Hash-based indexes
  - Can support equality queries
  - Cannot support range queries.
- Static Hashing
  - Can lead to long overflow chains.
  - Can result in lots of empty space.
- Extendible Hashing
  - Uses the directory to keep track of all buckets.
  - Does not have overflow pages.

Solution to Question 1

Step 0 (Initial condition)

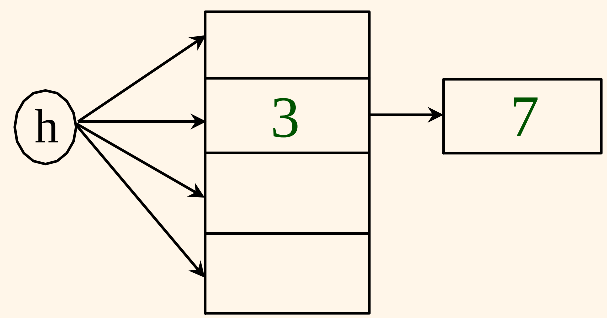


Step 1 (Insert 3)



$(1\times3+2) \bmod 4=1$

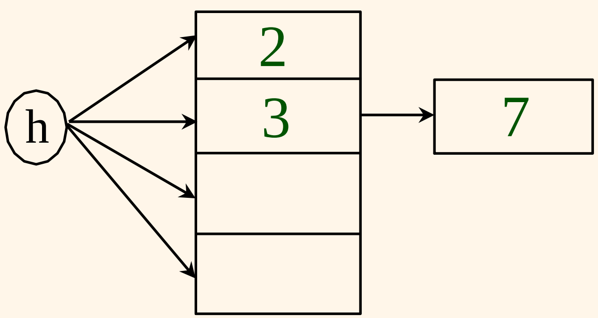
Step 2 (Insert 7)



$(1\times 7+2) \bmod 4=1$

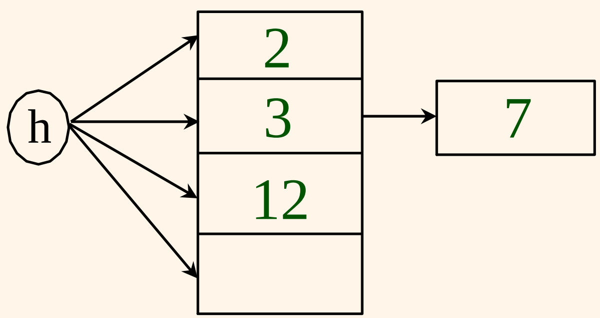
Solution to Question 1

Step 3 (Insert 2)



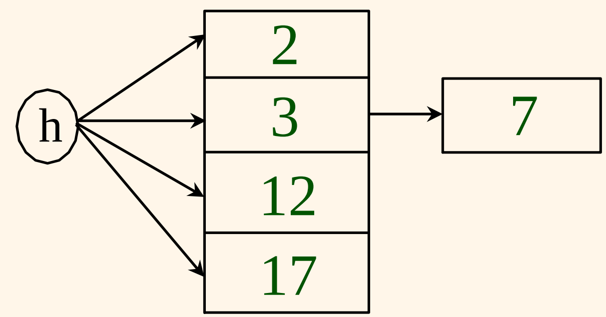
$(1\times 2+2) \bmod 4=0$

Step 4 (Insert 12)



$(1\times 12+2) \bmod 4=2$

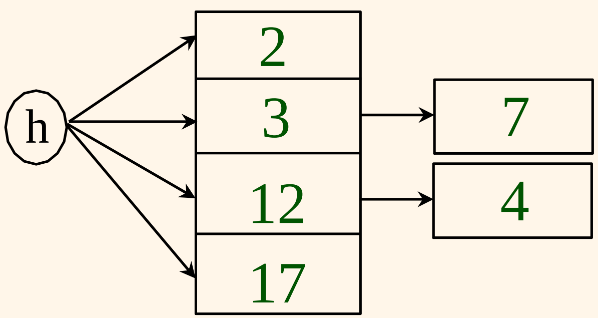
Step 5 (Insert 17)



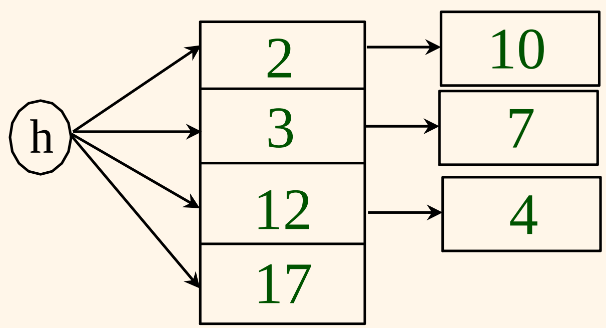
$(1\times 17+2) \bmod 4=3$

Solution to Question 1

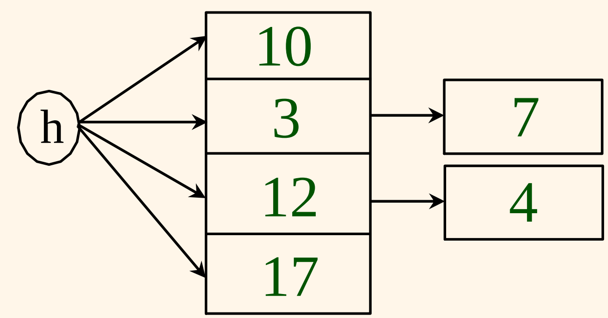
Step 6 (Insert 4)



Step 7 (Insert 10)



Step 8 (Delete 2)



$(1\times 4+2) \bmod 4=2$

$(1\times 10+2) \bmod 4=0$

Solution to Question 2

a) 18: 10010_2, 25: 11001_2, 37: 100101_2, 77: 1001101_2

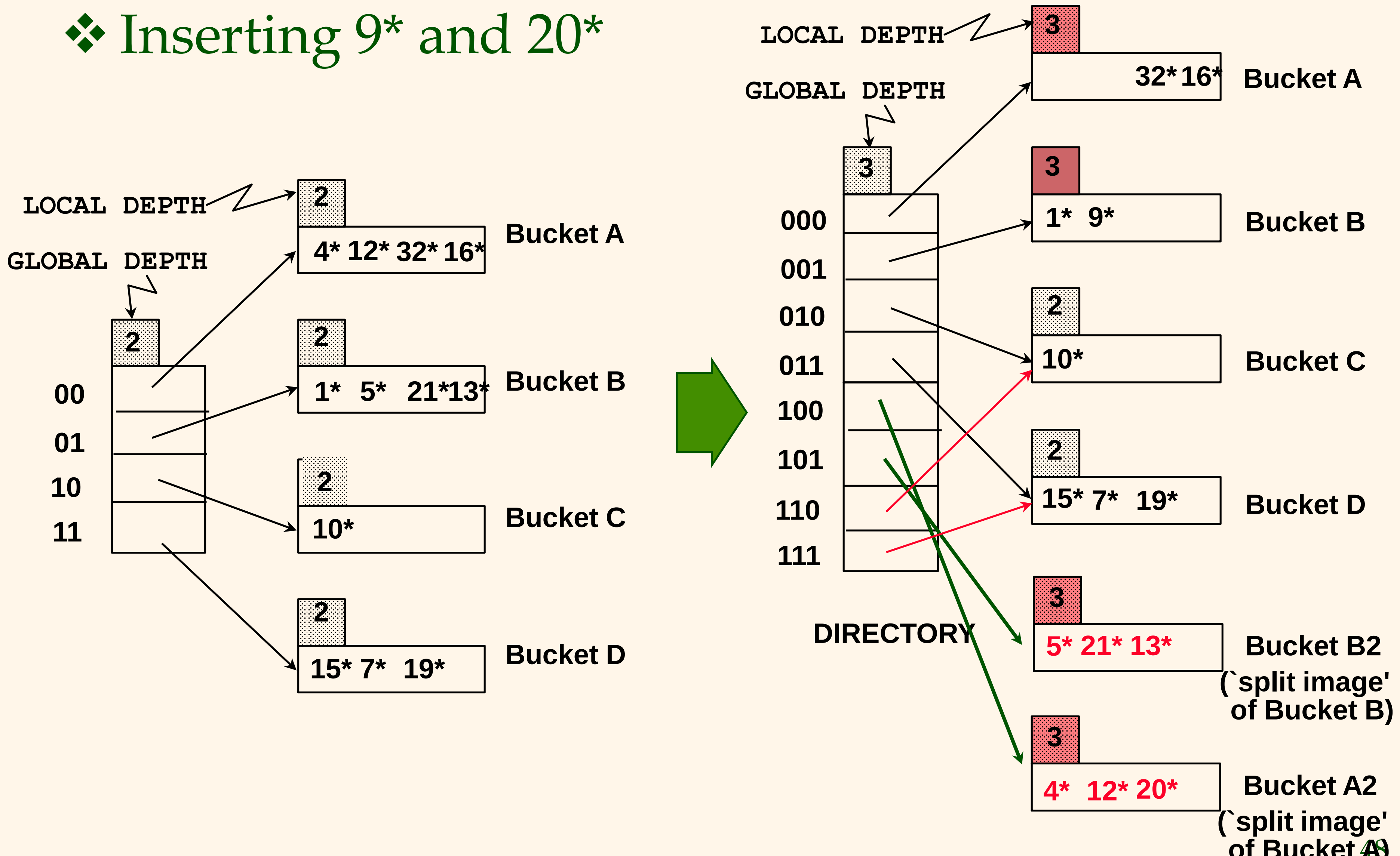
b) 18 mod 2 = 0 , 25 mod 2 = 1,
37 mod 2 = 1, 77 mod 2 = 1

c) 18 mod 4 = 10, 25 mod 4 = 01,
37 mod 4 = 01, 77 mod 4 = 01

d) 18 mod 8 = 010, 25 mod 8 = 001,
37 mod 8 = 101, 77 mod 4 = 101

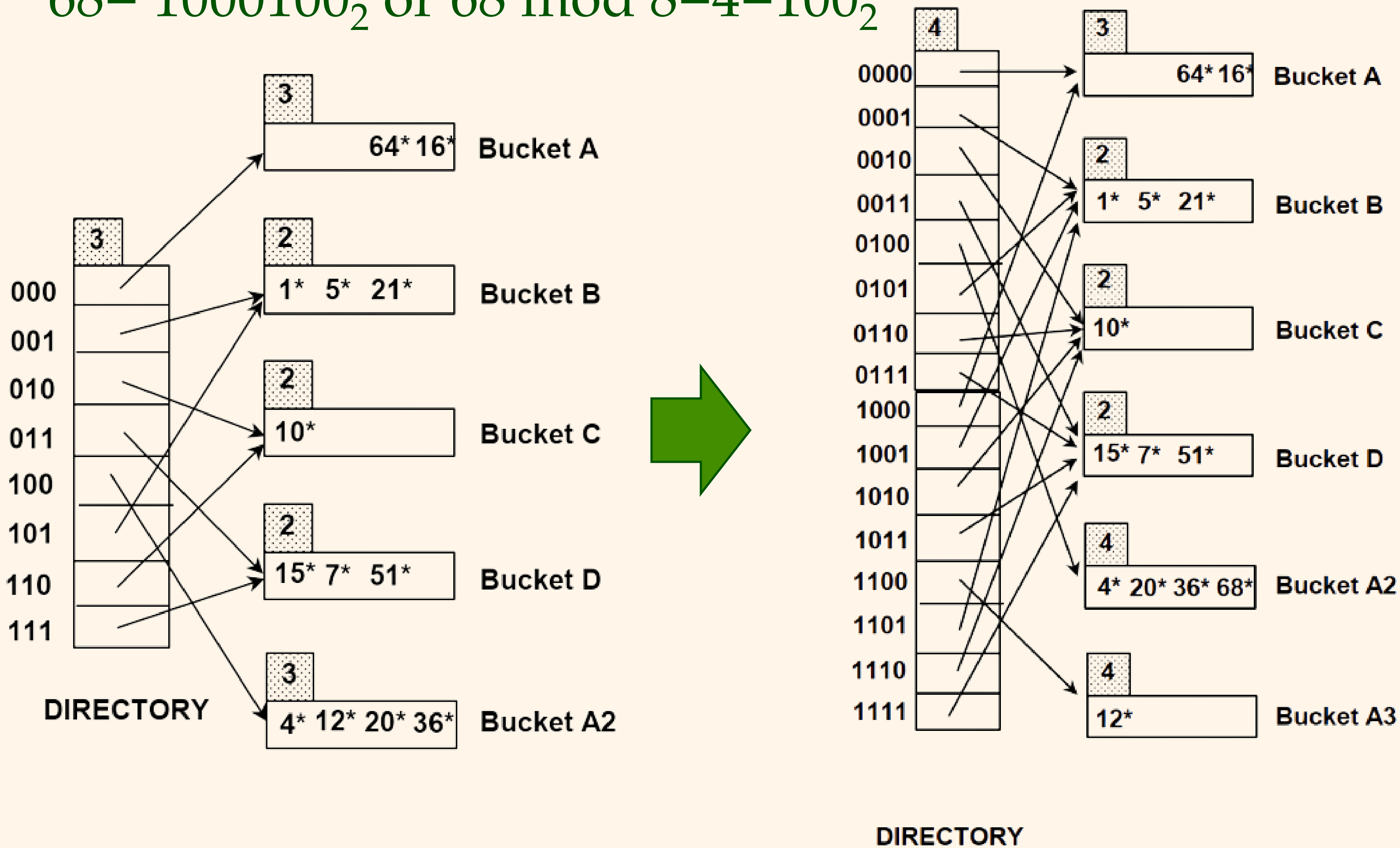
Solution to Question 3

❖ Inserting 9* and 20*



Solution to Question 4

a) The index after inserting an entry with hash value $68 = 1000100_2$ or $68 \bmod 8 = 4 = 100_2$



Solution to Question 4

b) The index after inserting entries with hash values 17= 0010001_2 and 69= 1000101_2 into the original index.

- 17 mod 8 = 1 =001_2, 69 mod 8 = 5 = 101_2