CS660: Intro to Database Systems

Class 8: Hash Indexing

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS660/

Guest Lectures & Midterm

- October 3, Tarikul Islam Papon (Researcher at BU)
  Asymmetry/Concurrency-Aware Bufferpool Manager for Modern Storage Devices

- November 21, George Neville-Neil (Researcher at Yale University)
  Using a SQL engine to manage the internals of an Operating System (tentative title)

- October 22, Review
- October 24, Midterm (in this room)
  - Details about what to study will be posted in Piazza closer to the exam

Project Administrivia

- Common problems with setting up your environment

- Working on a common platform that is easy to deploy and use
  - Using SCC (Shared Computing Cluster of BU)
  - We will soon provide a set of guidelines on how to proceed
  - It will be optional (the requirement is that you upload the code to gradescope)
  - It will help you and us to have a standardized environment
  - We will only support questions about the environment on this platform

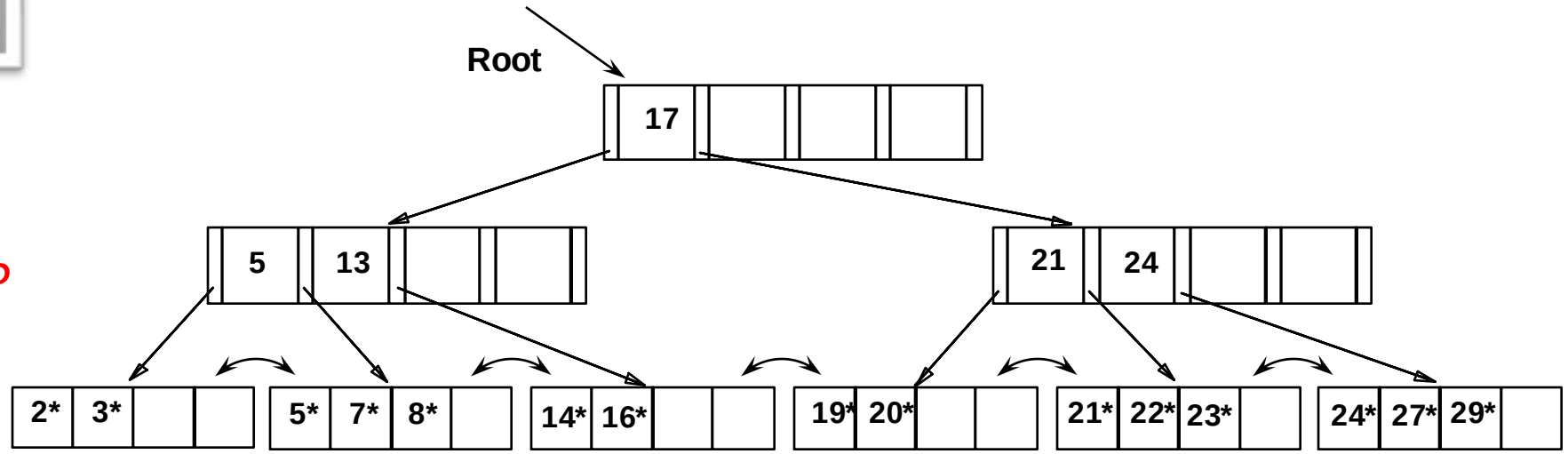
- This will help you and your future colleagues!

Stay tuned!!

Last time: B+ Trees

"It could be said that the world's information is at our fingertips because of B-trees"

Other forms of indexing?



Hash Indexing

Static Hashing

Extendible Hashing

Linear Hashing

Reminder: Alternatives of Data Entries

1. <**k**, entire data record>
2. <**k**, rid of exactly-one-at-a-time matching data record>
3. <**k**, list of rids of matching data records>

Choice is orthogonal to the indexing technique

Hash-based indexes → *equality selections*
Cannot support range searches

Static and dynamic hashing techniques exist

Hash function

(for the purposes of hash indexes)

a **function** that **maps** a **search key** to an **index** between **[0 .. M-1]**

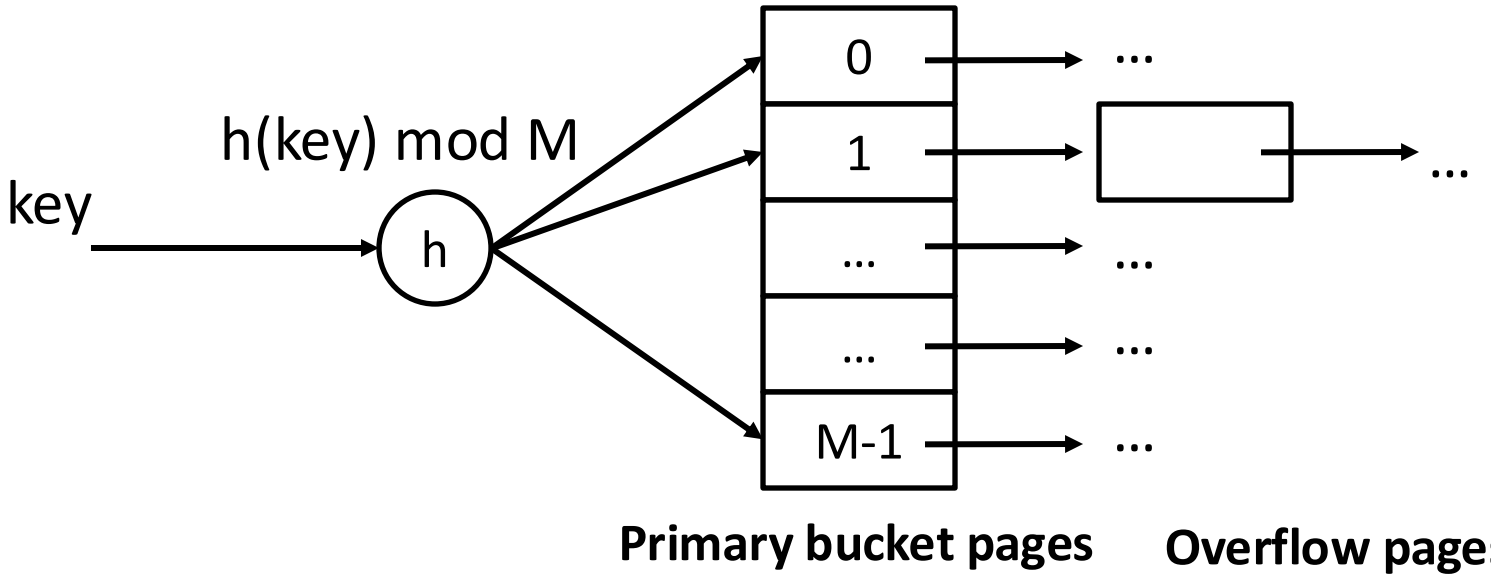
where M is the number of **buckets** (pages) available to our index

- ideally a hash function maps the search keys uniformly in [0, …, M-1]
- in practice simple hash functions are used (fast to compute)
- different keys might be mapped to the same bucket

Static Hashing

#primary bucket pages fixed, allocated sequentially, never de-allocated;
overflow pages if needed

h(k) mod M = bucket to insert data entry with key *k* (M: #buckets)



what if a bucket gets full?

Static Hashing (Contd.)

Buckets contain data entries

Remember, data entries:
<k, record>
<k, rid>
<k, rid-list>

Hash function on *search key* field of record *r*

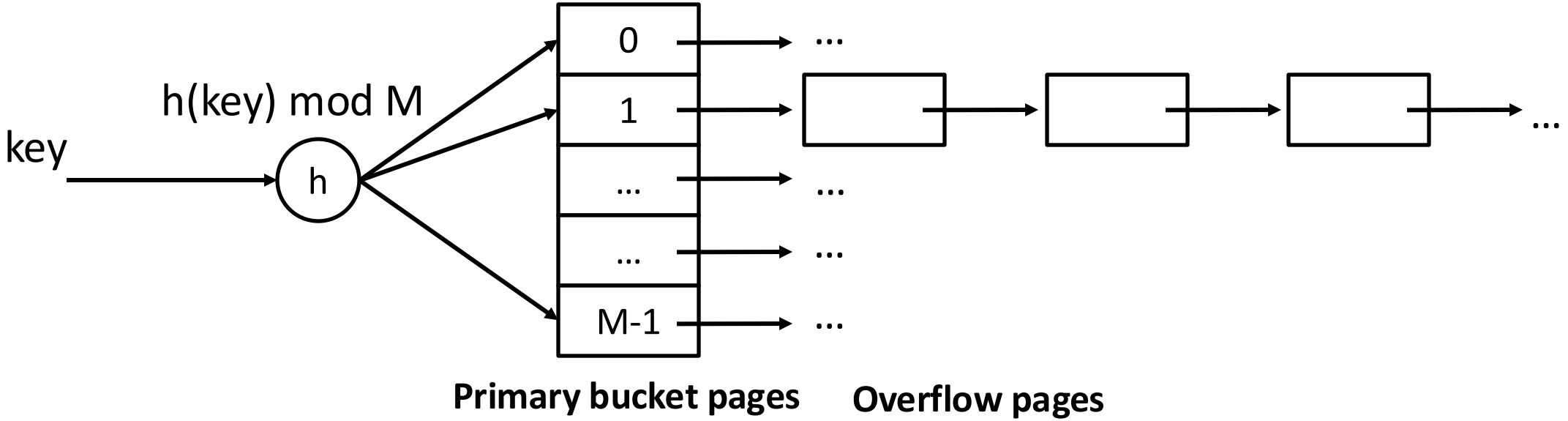
Must distribute values over range 0 ... M-1

What is a good hash function?

h(*key*) = (a * *key* + b) usually works well

a and b are constants; lots known about how to tune **h**

Static Hashing – Problems?



What does that do to performance?

Instead of O(1) we may go as bad as O(N)

Static Hashing – Solutions

Long overflow chains can develop and degrade performance

Ways to solve?

– **Reorganization** (re-hashing) is expensive and may block queries

– ***Extendible* and *Linear Hashing***: Dynamic techniques to fix this problem

Hash Indexing

Static Hashing

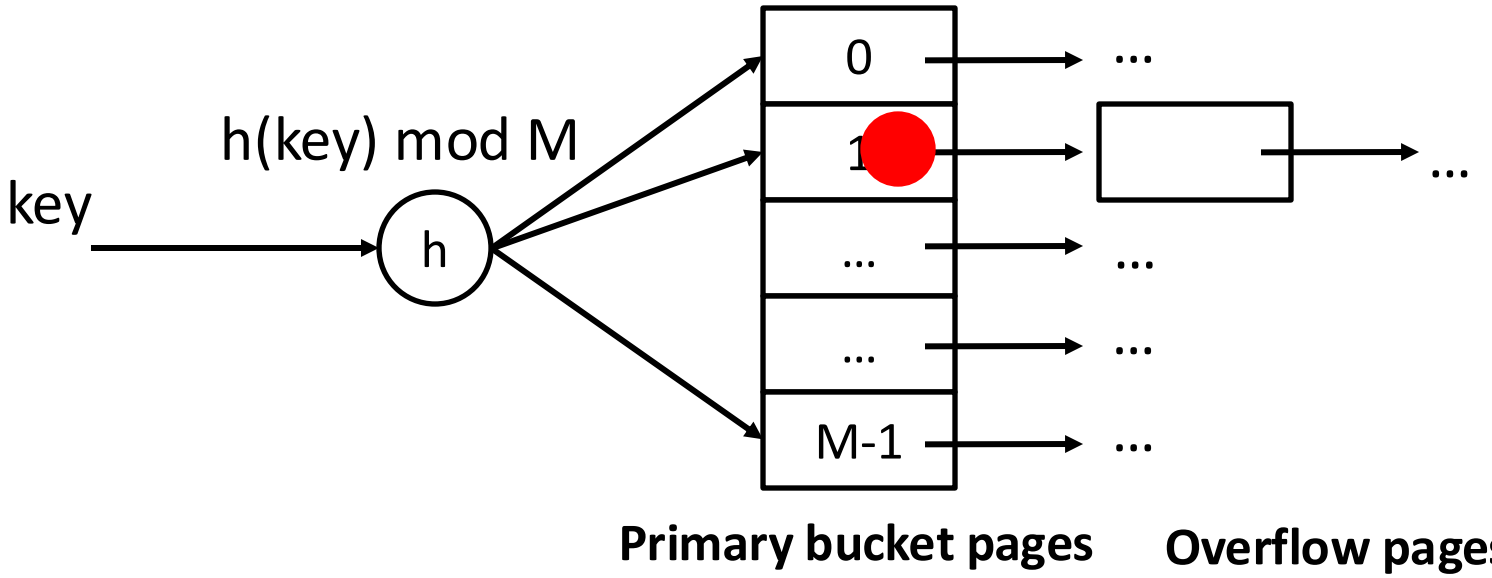
Extendible Hashing

Linear Hashing

Let's start from Static Hashing

What else we can do instead of adding an overflow page?

h(k) mod M = bucket to insert data entry with key *k* (M: #buckets)



Extendible Hashing

Why not **double** the **number** of buckets?

Note that reading and writing all pages is expensive!

how to do this?

Idea:

Use **directory of pointers** to buckets

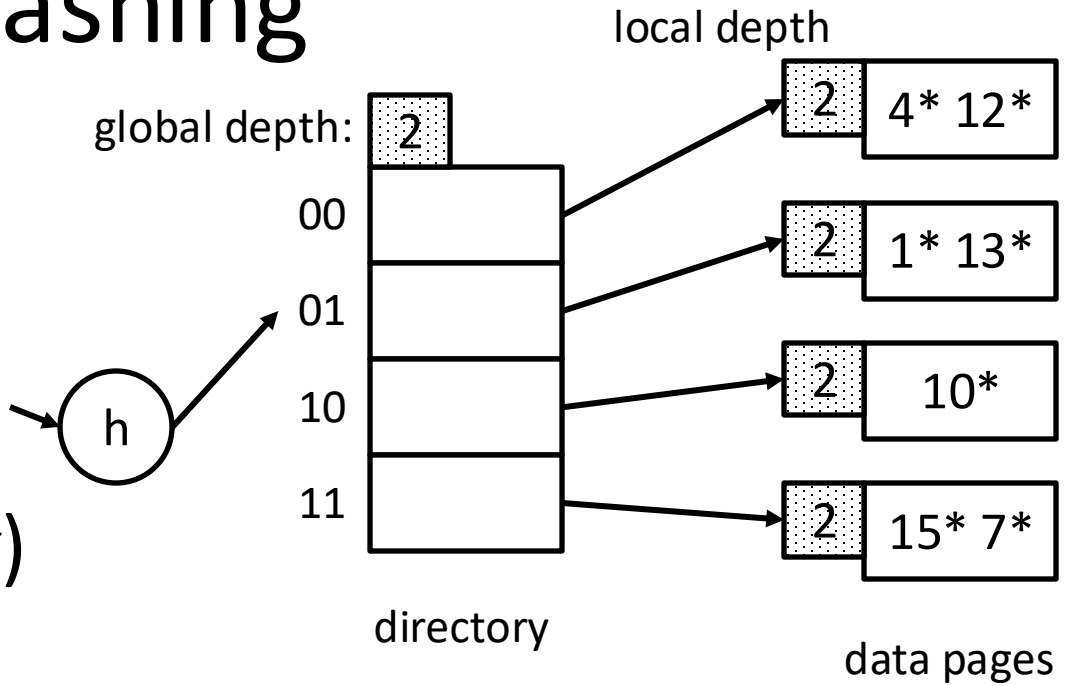
On overflow, **double only the directory** (not the # of buckets)

Why does this help?

- Directory is much smaller than the entire index file
- Only one page of data entries is split
- *No overflow page! (caveat: duplicates w.r.t. the hash function)*

Trick lies in how the hash function is adjusted!

Extendible Hashing



Directory: an array

Search for k:

– Apply hash function h(k)

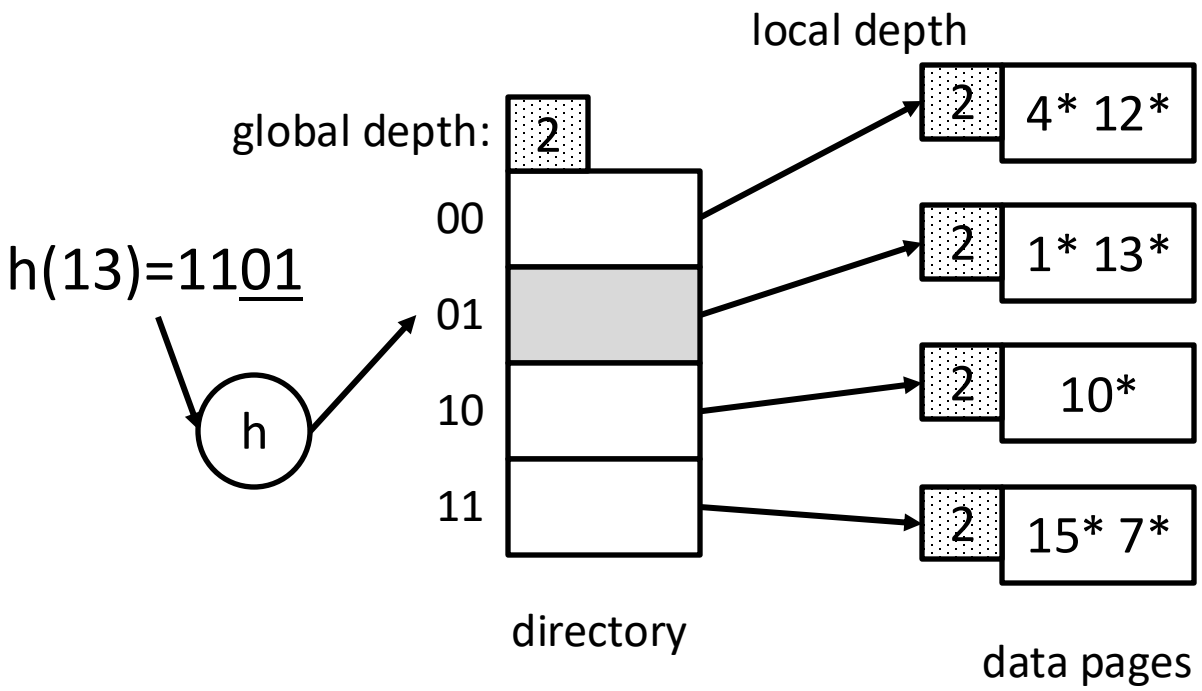
– Take last **global depth** # bits of **h**(*k*)

Insert:

– If the bucket has space, insert, done

– If the bucket is full, **split** it, re-distribute – If necessary, double the directory

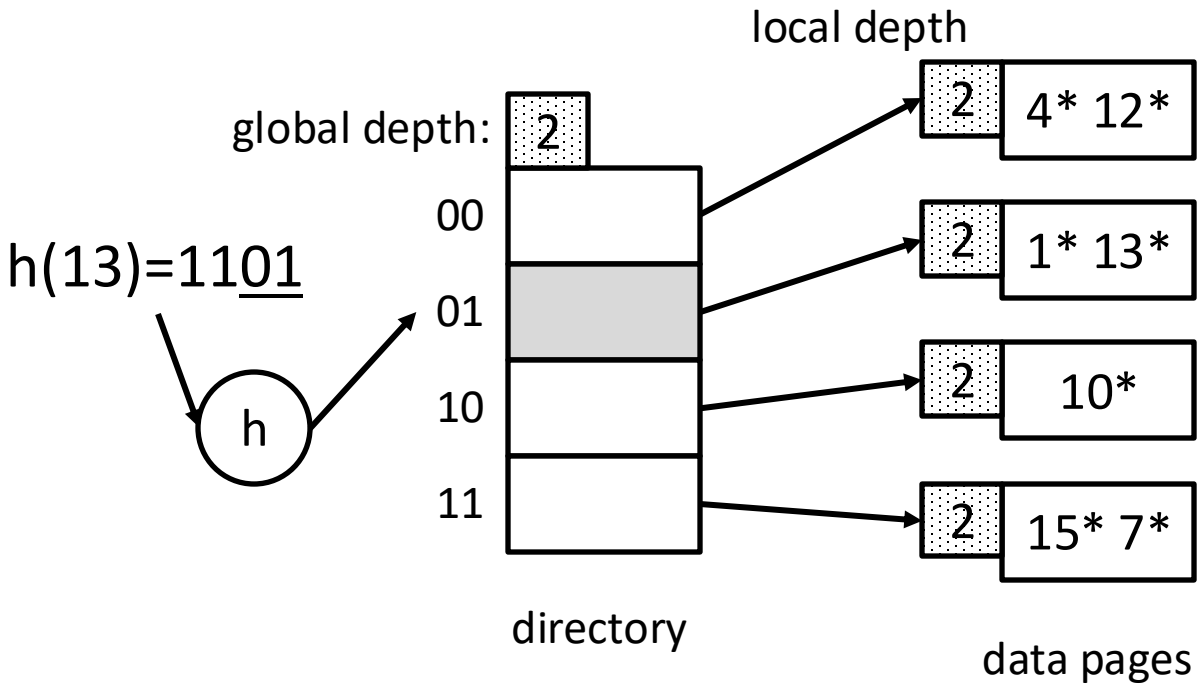
Example



Clarification:

- for ease of presentation, we use h(key)=key

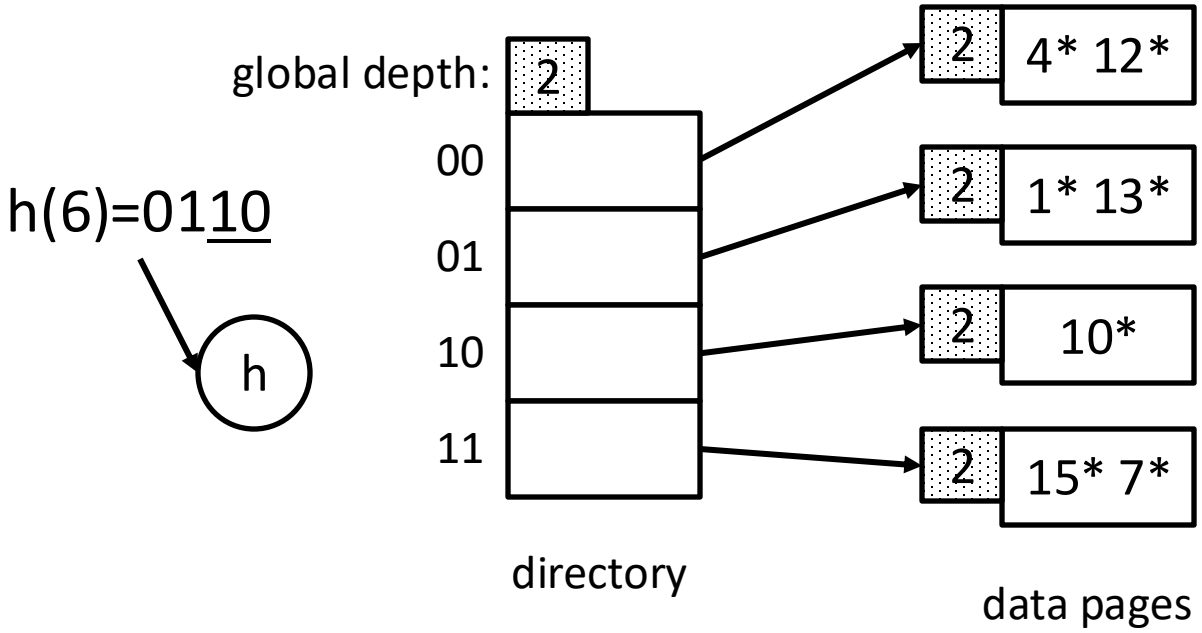
Example



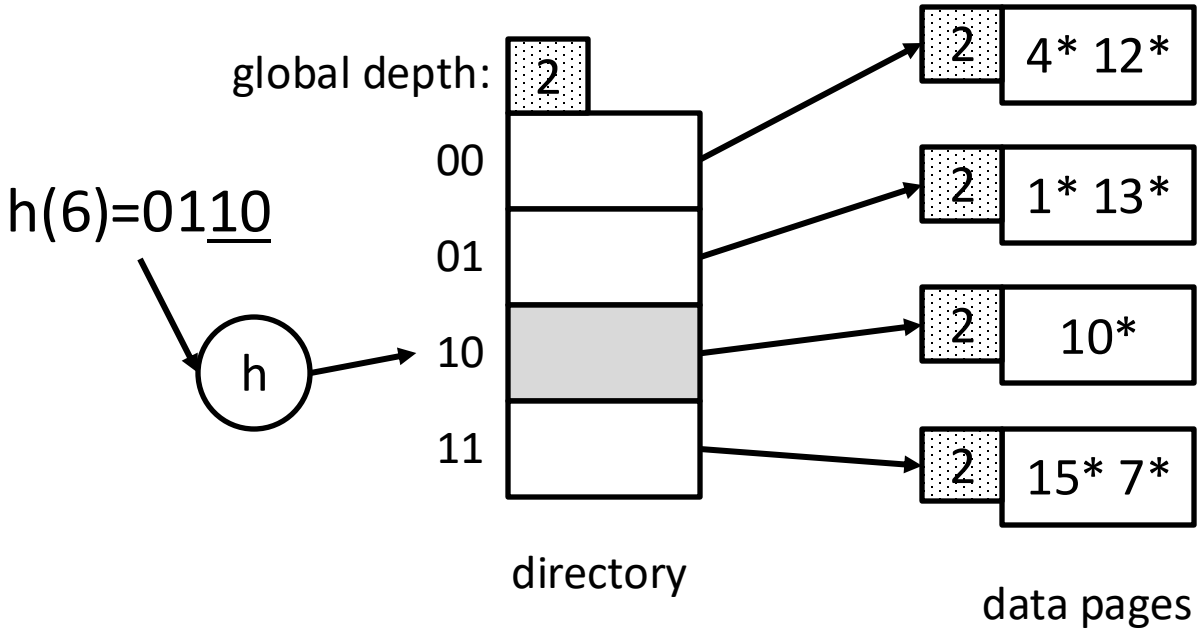
how to calculate global depth?

the last two bits! so: *h(k) mod 4*

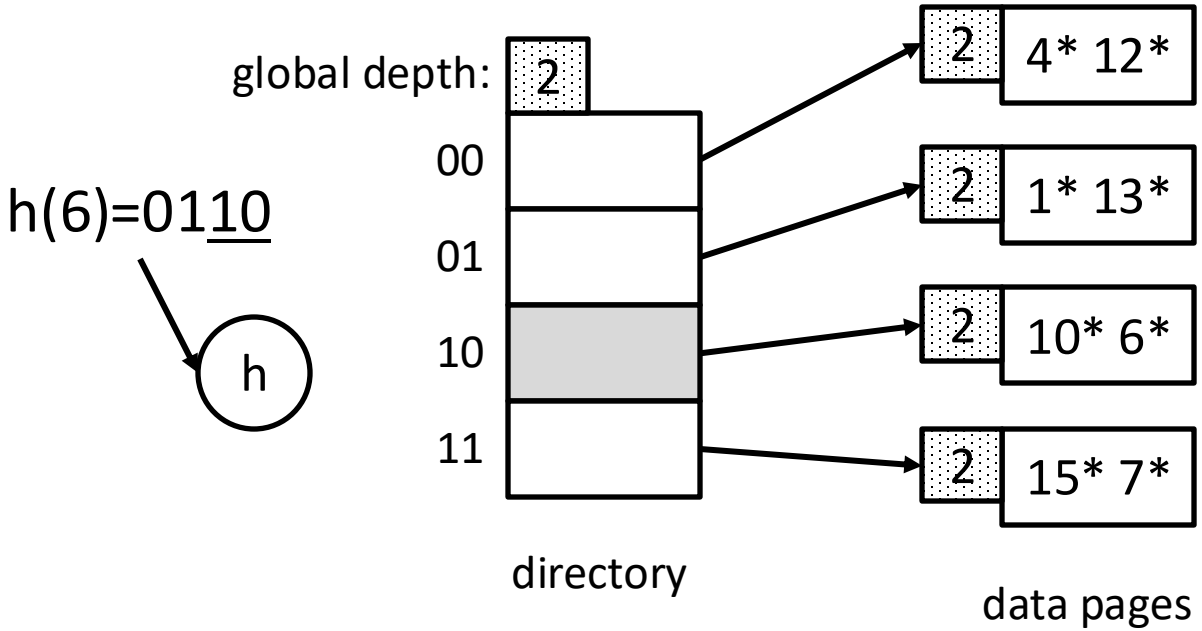
Example: Insert 6



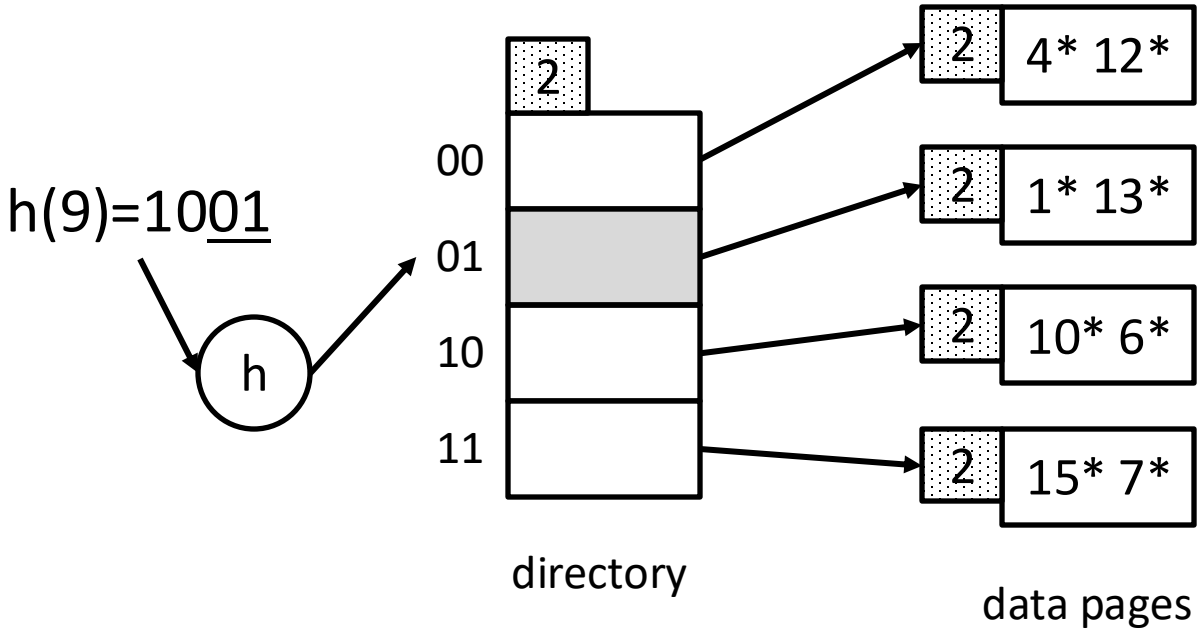
Example: Insert 6



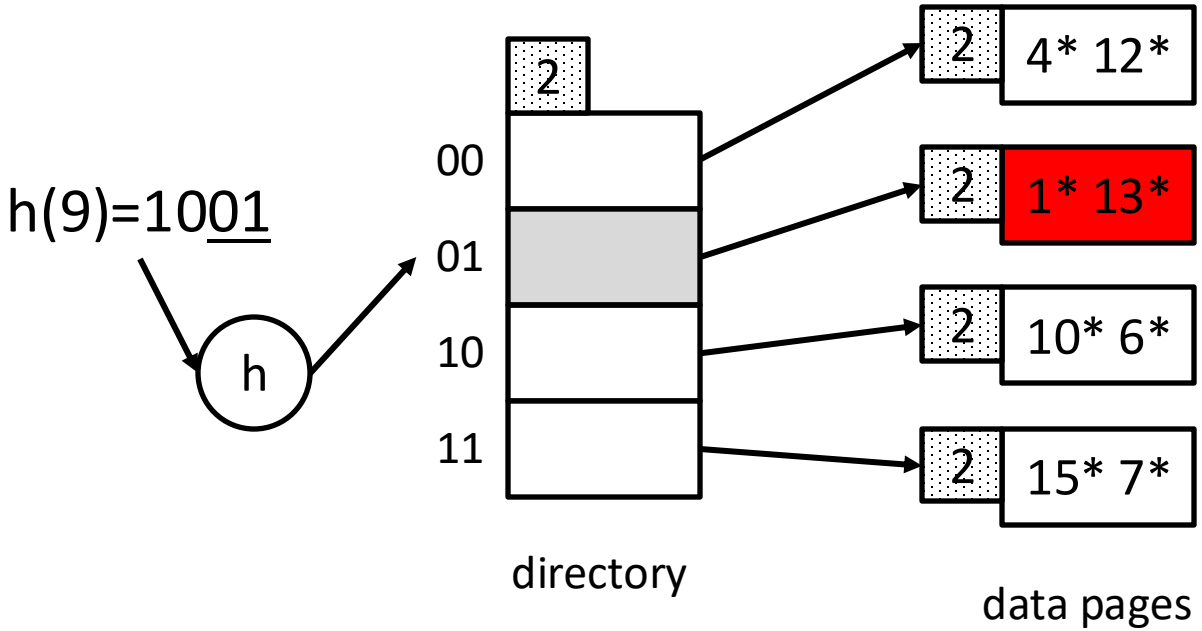
Example: Insert 6



Example 2: Insert 9

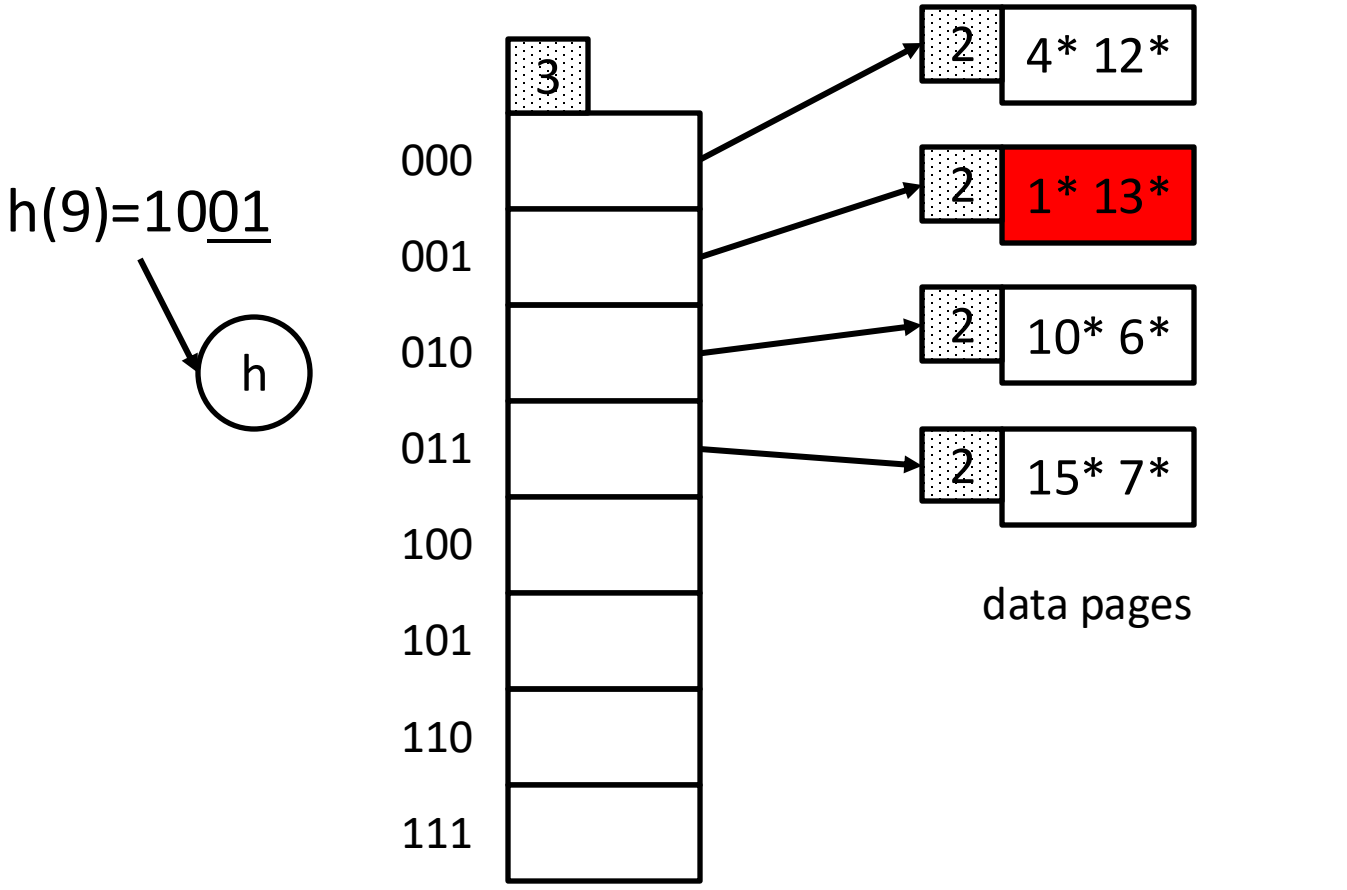


Example 2: Insert 9



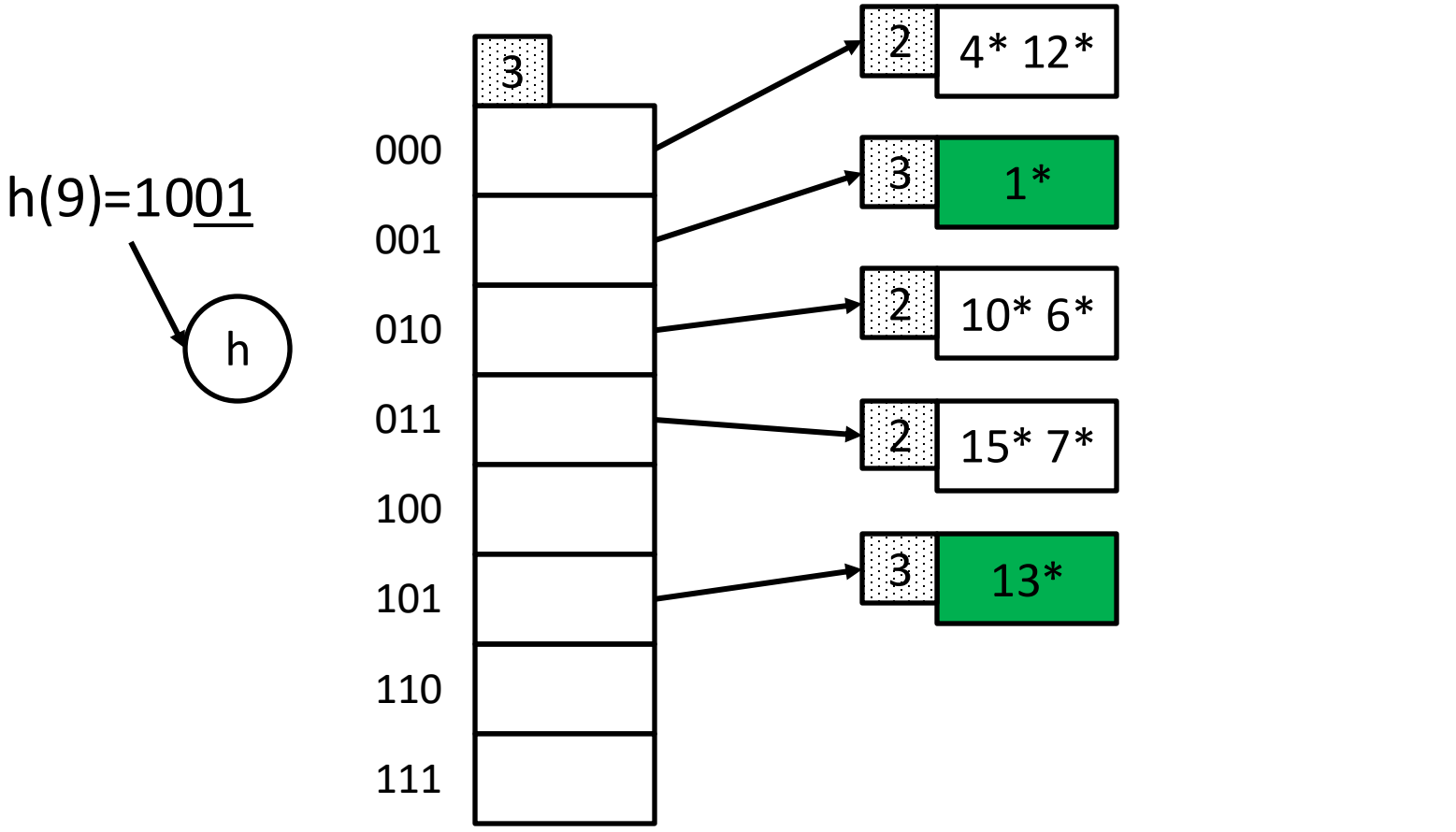
now what??

Example 2: Insert 9



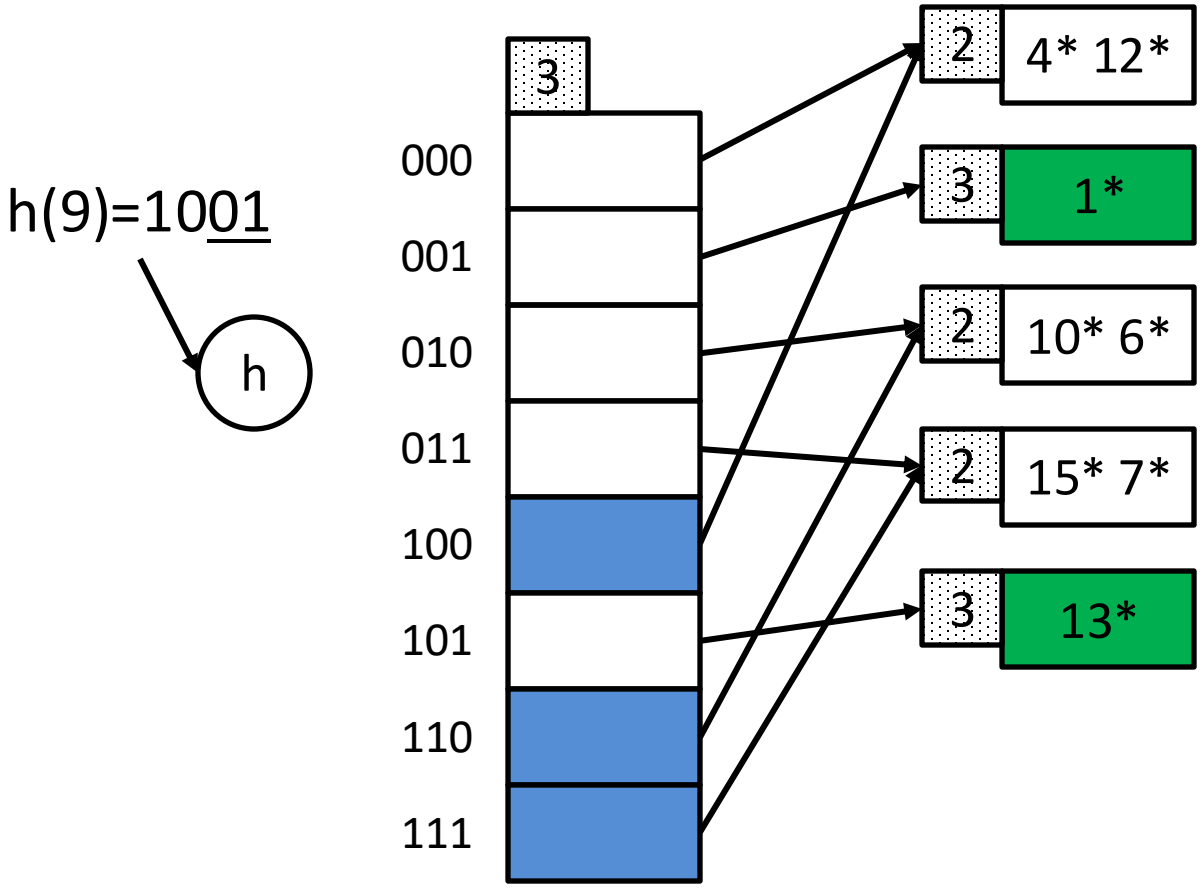
(1) double the directory

Example 2: Insert 9



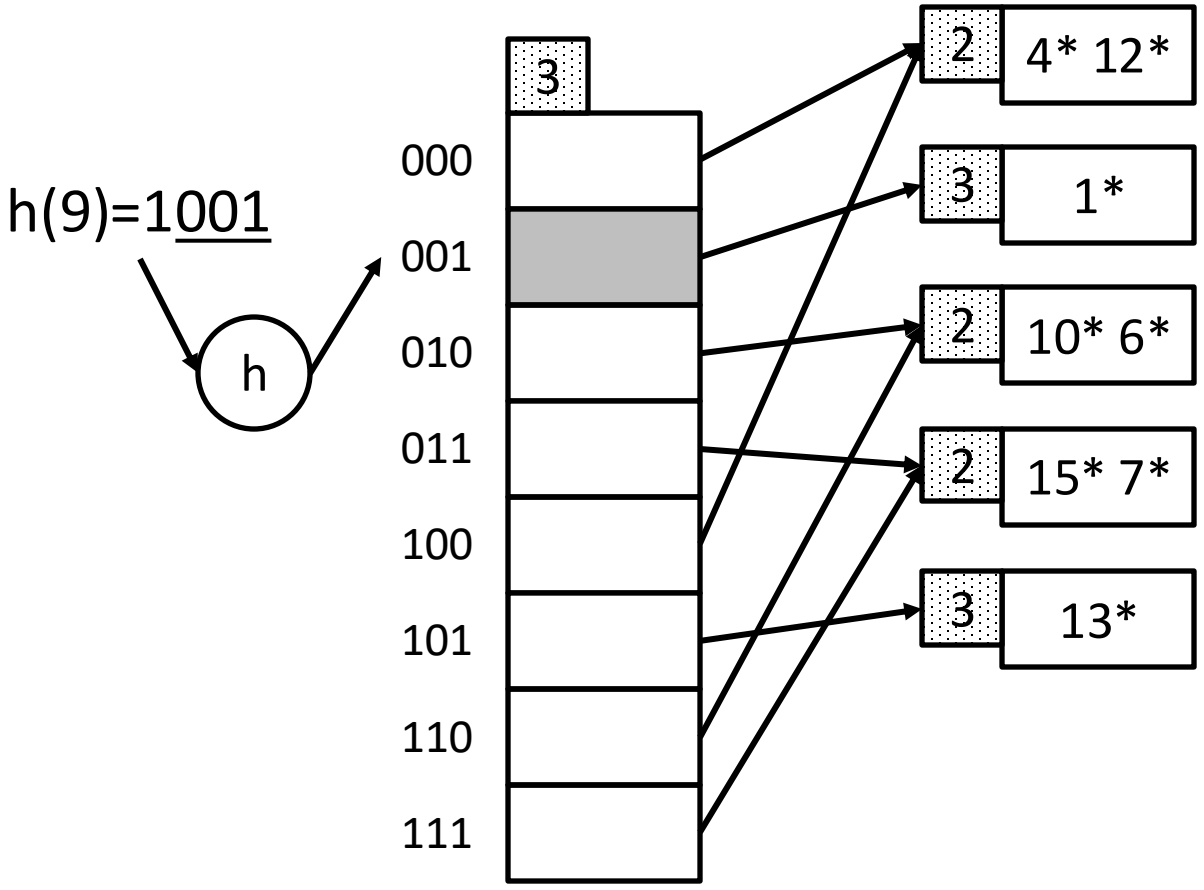
(1) double the directory
(2) re-distribute the split bucket

Example 2: Insert 9

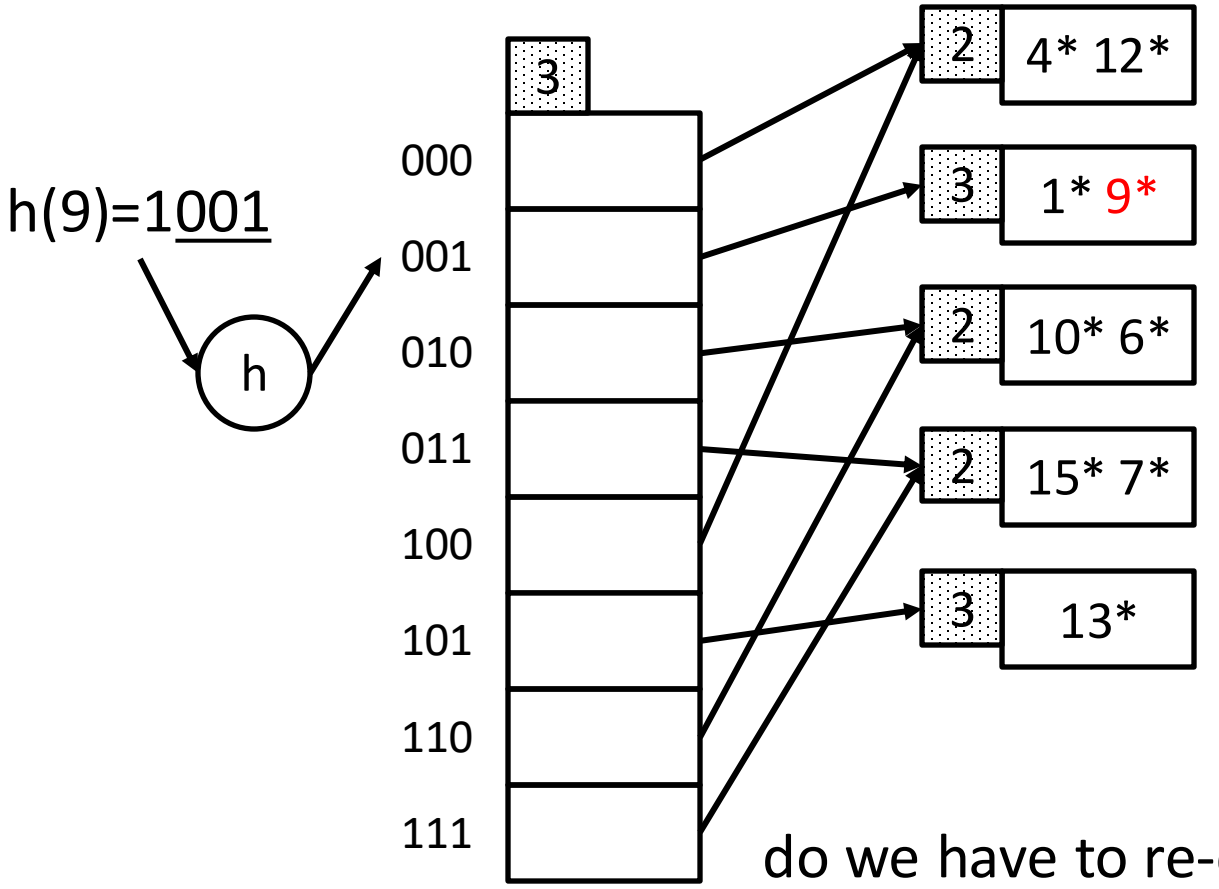


(1) double the directory
(2) re-distribute the split bucket
(3) connect corresponding buckets

Example 2: Insert 9

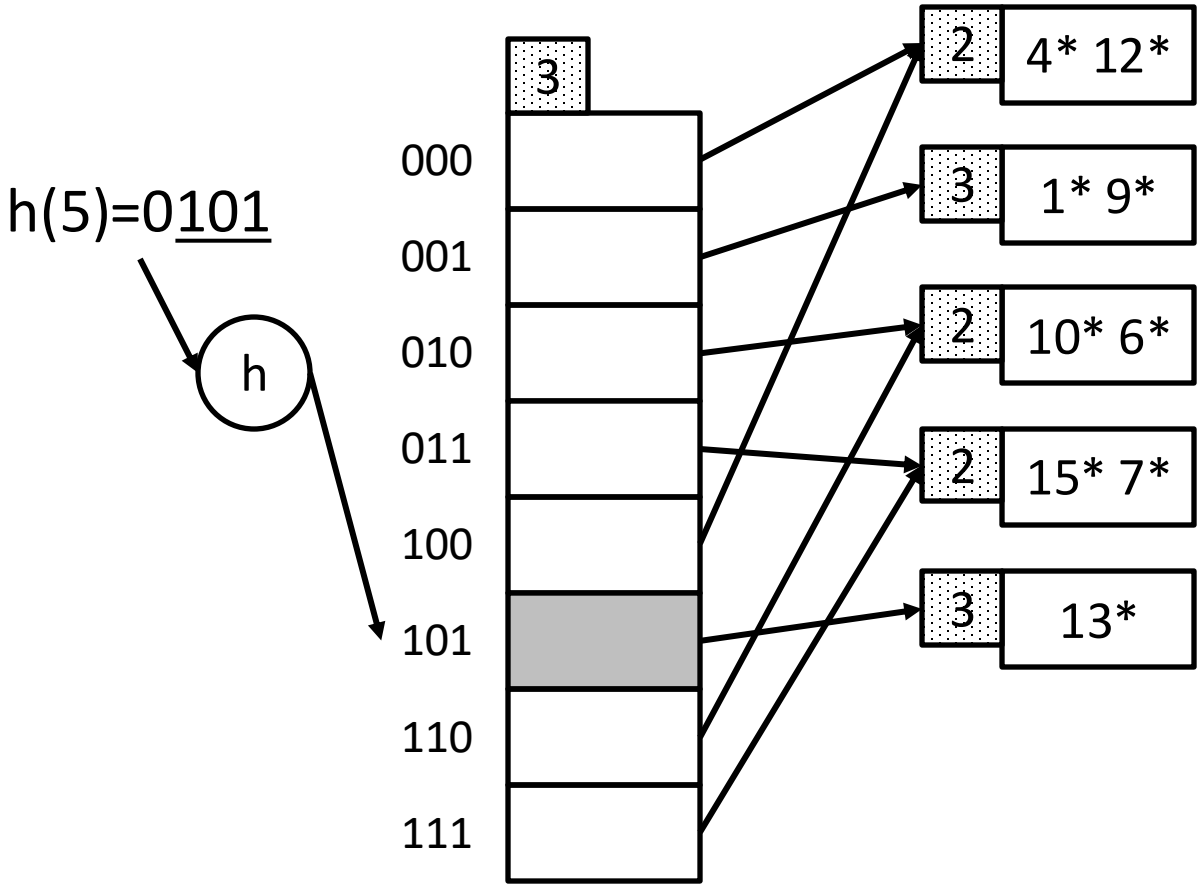


Example 2: Insert 9

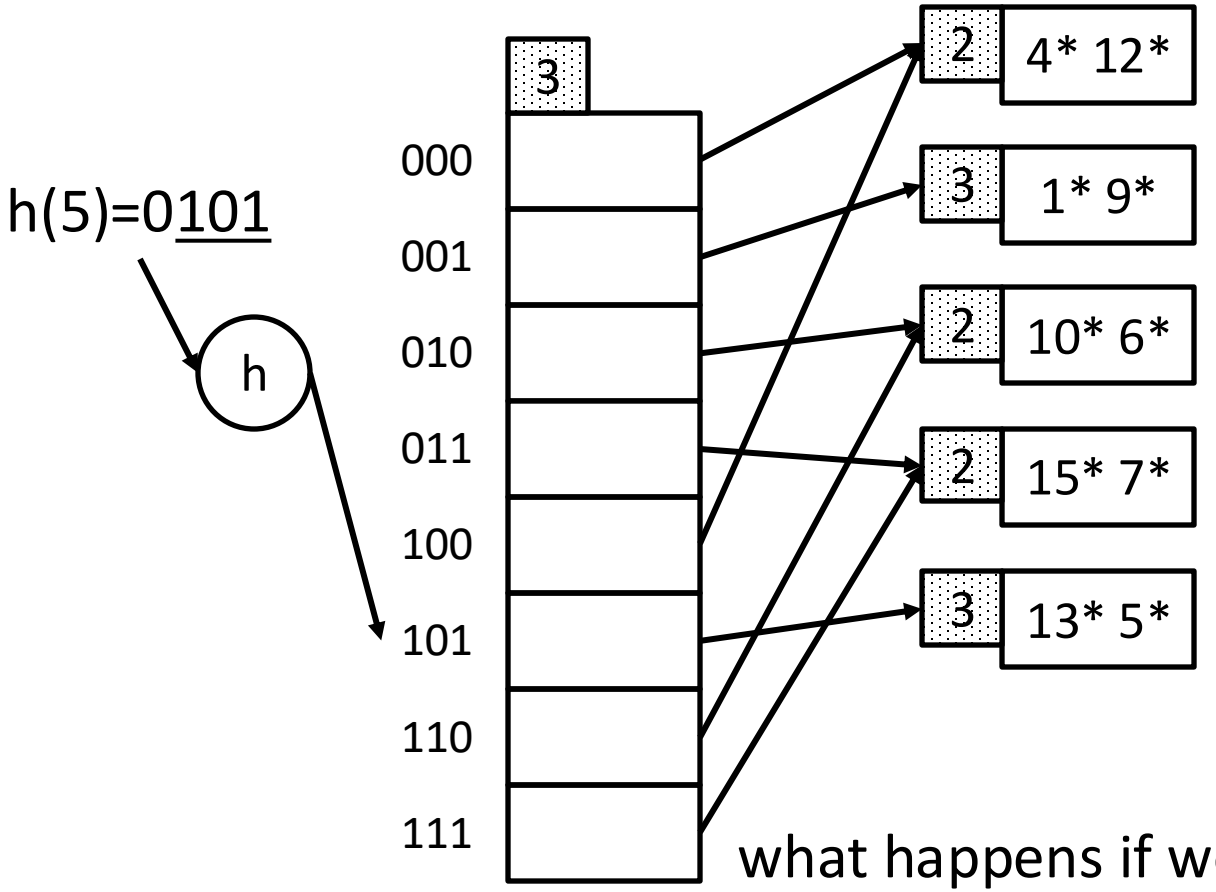


do we have to re-distribute all?

Example 3: Insert 5



Example 3: Insert 5

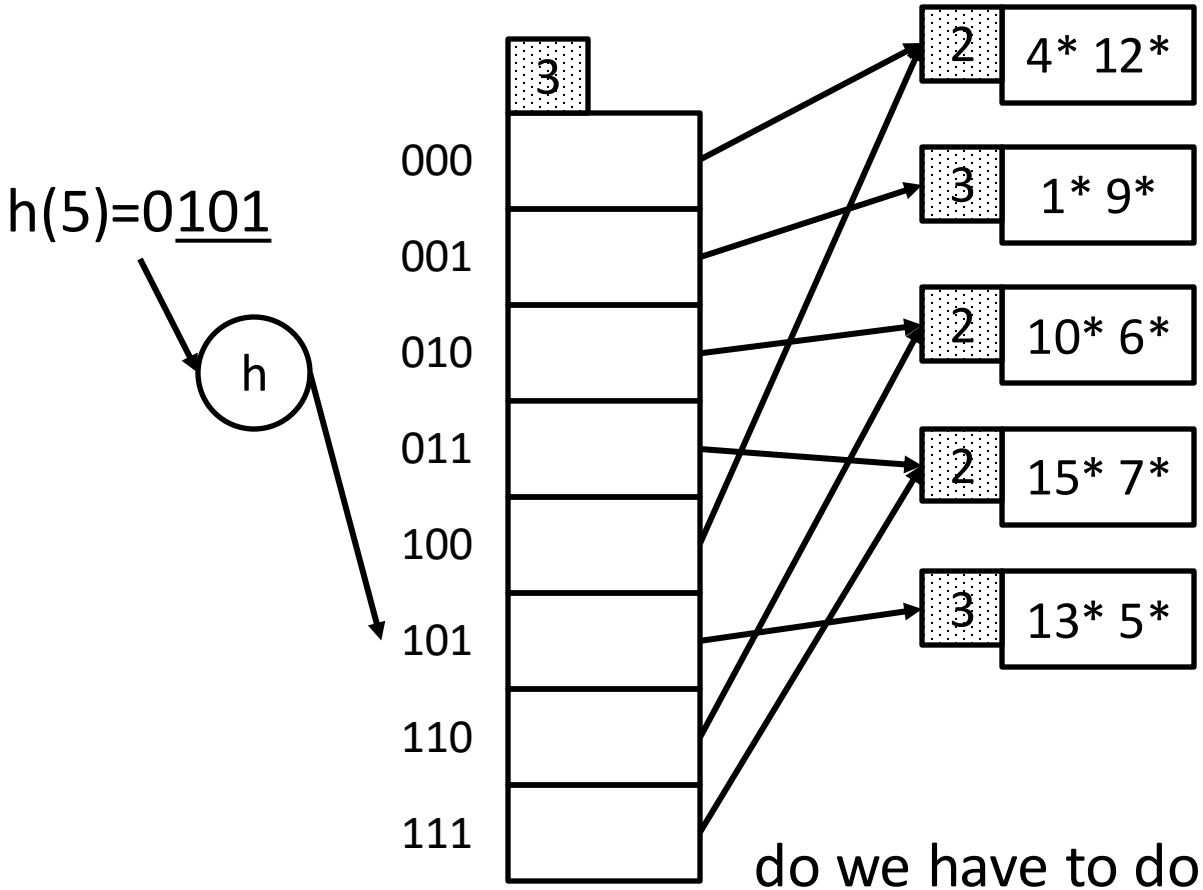


what happens if we want to insert 17 (→ 10001) ?

do we have to re-distribute all?

[17→10001] so, double the dir again!

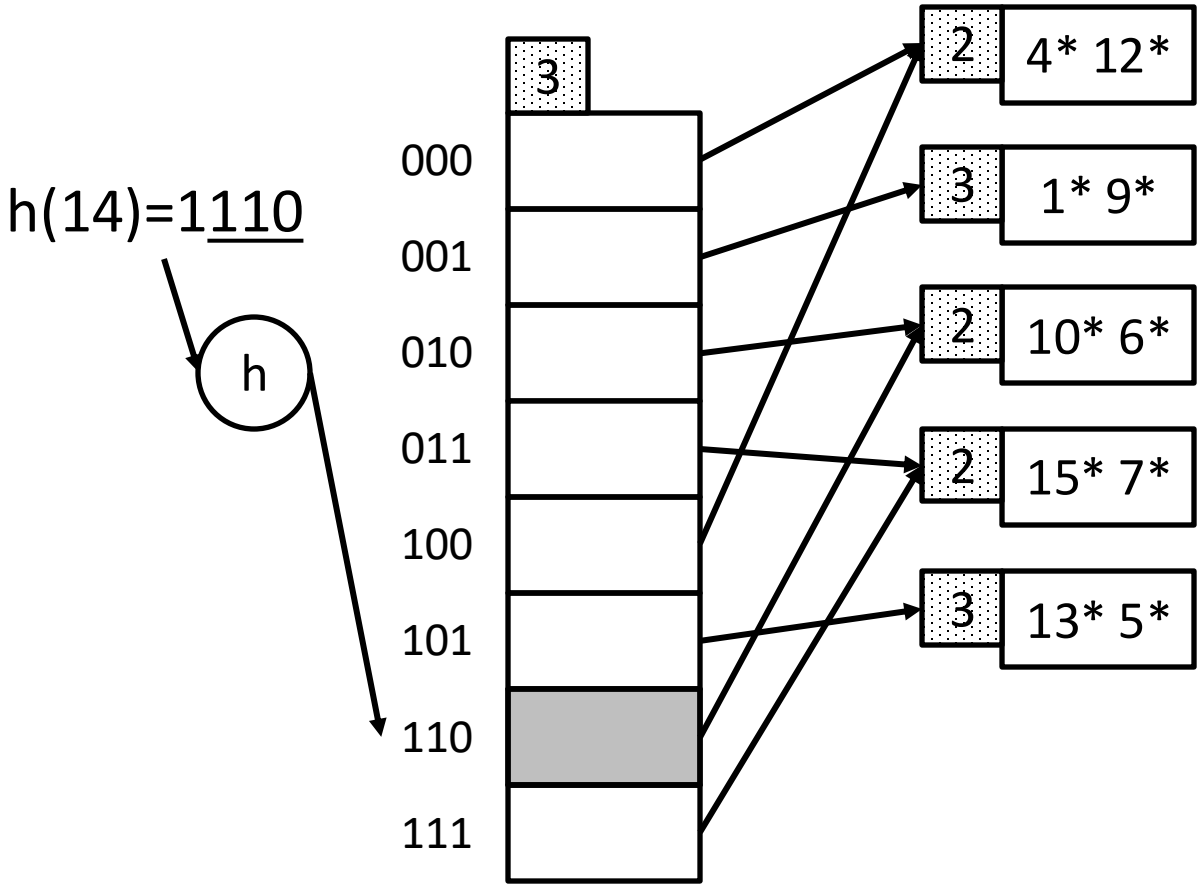
Example 3: Insert 5



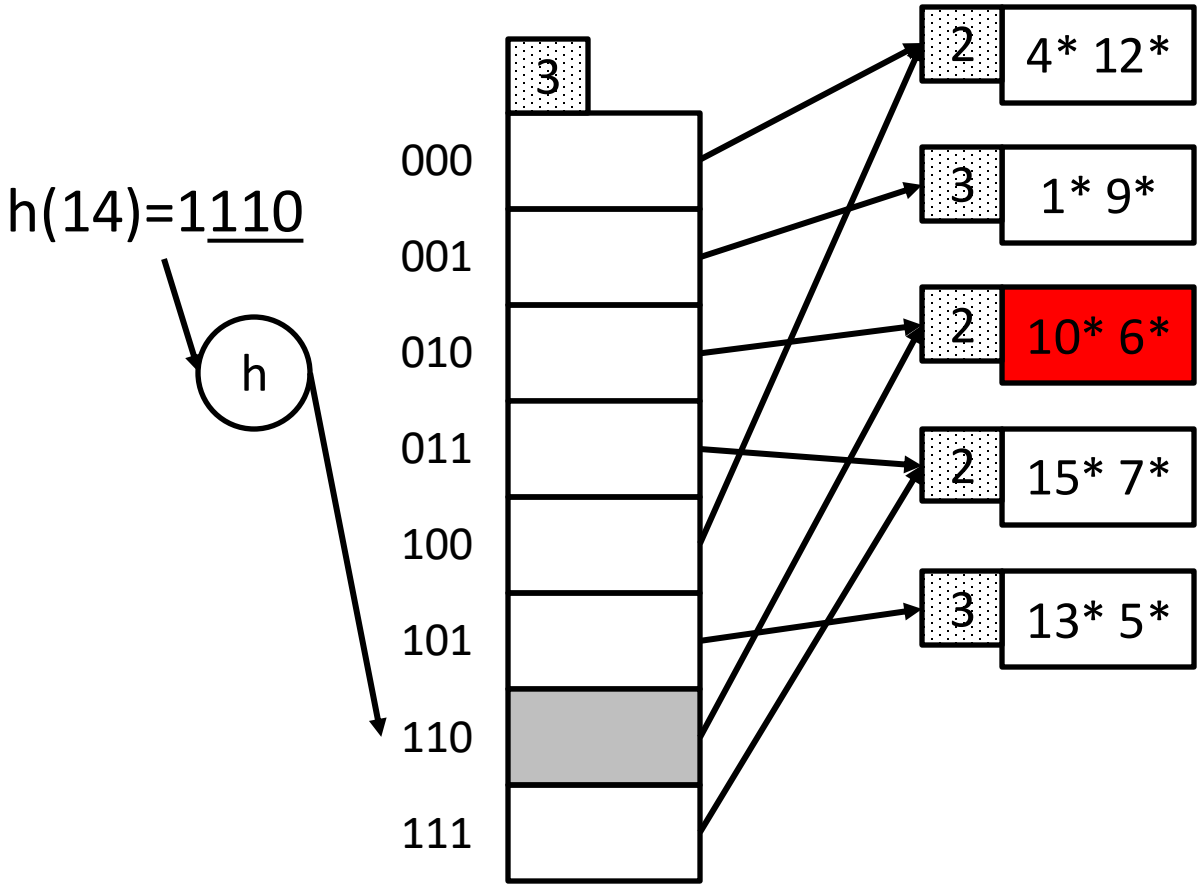
do we have to double the directory every time we split a bucket?

when?

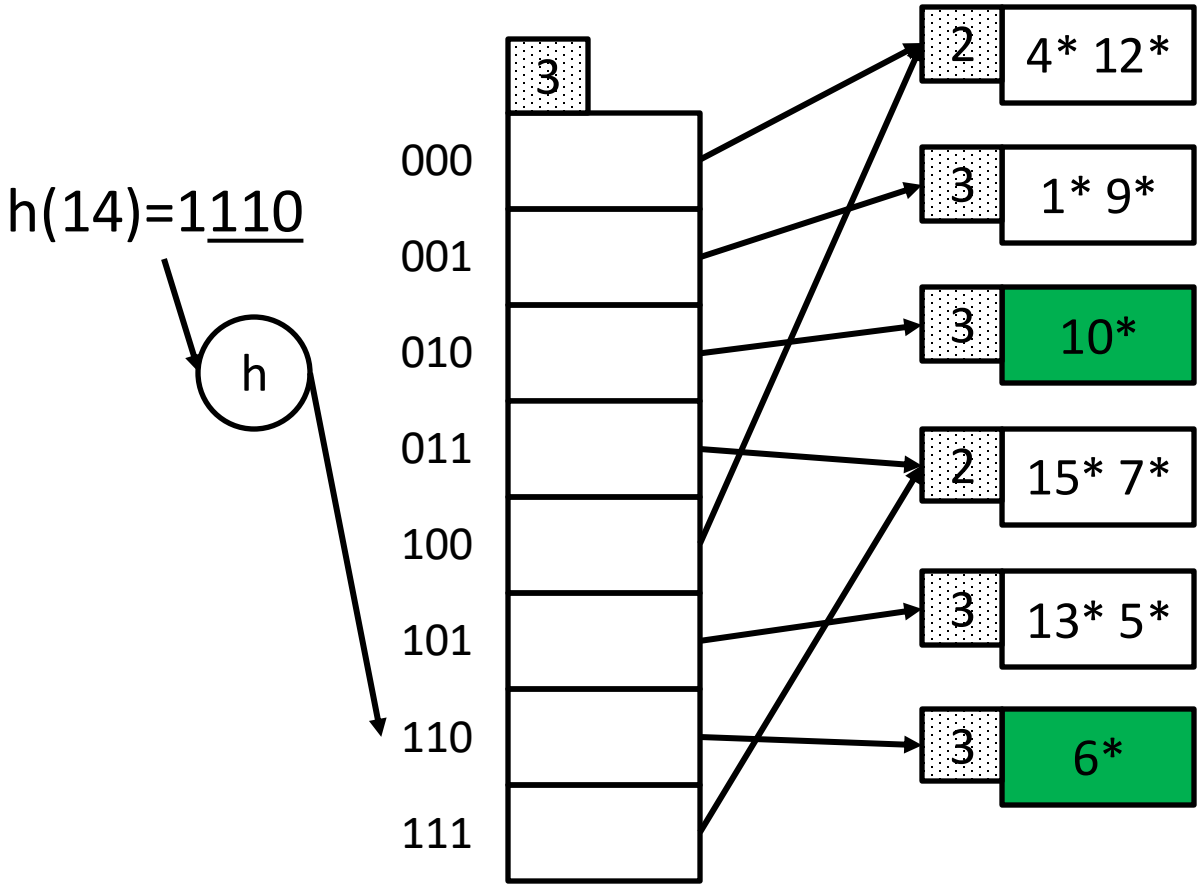
Example 3: Insert 14



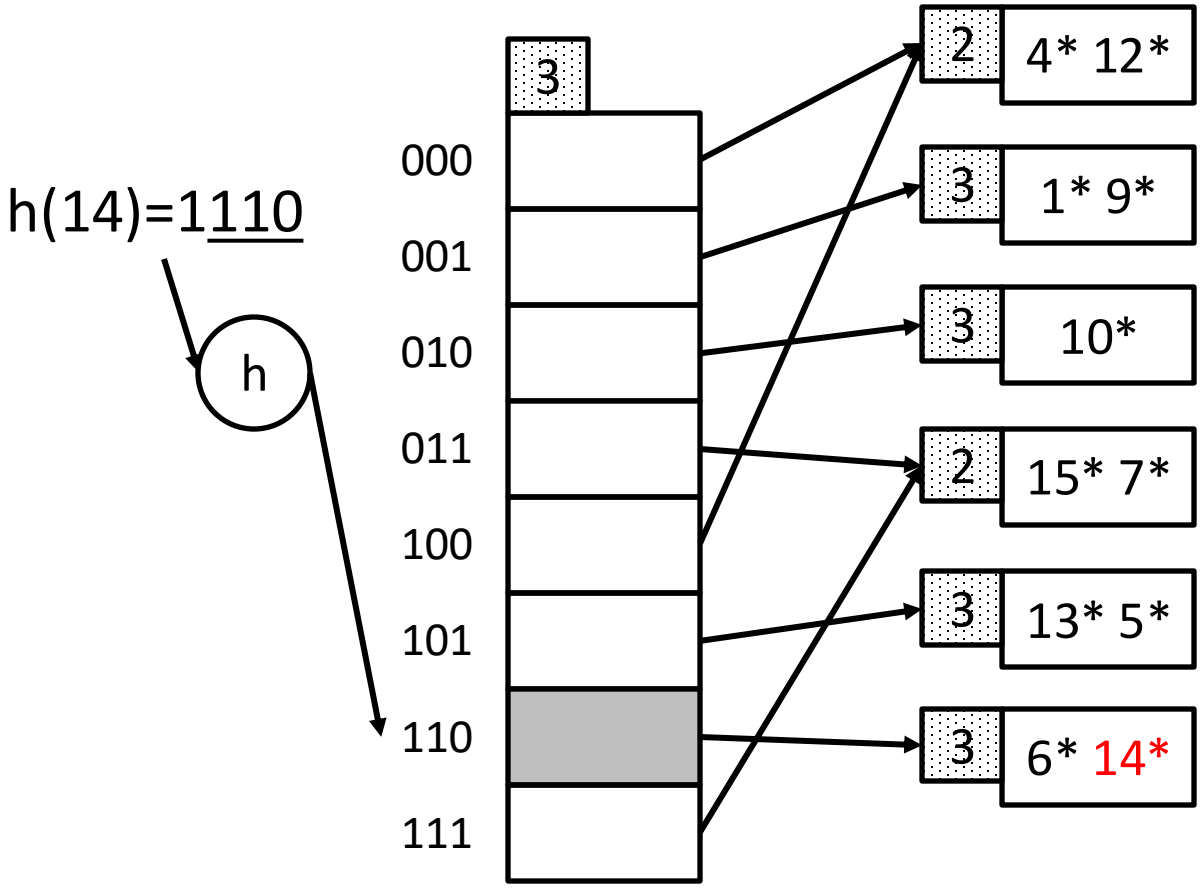
Example 3: Insert 14



Example 3: Insert 14



Example 3: Insert 14



Notes on Extendible Hashing

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, can grow large

Notes on Extendible Hashing

Do we ever need overflow pages?

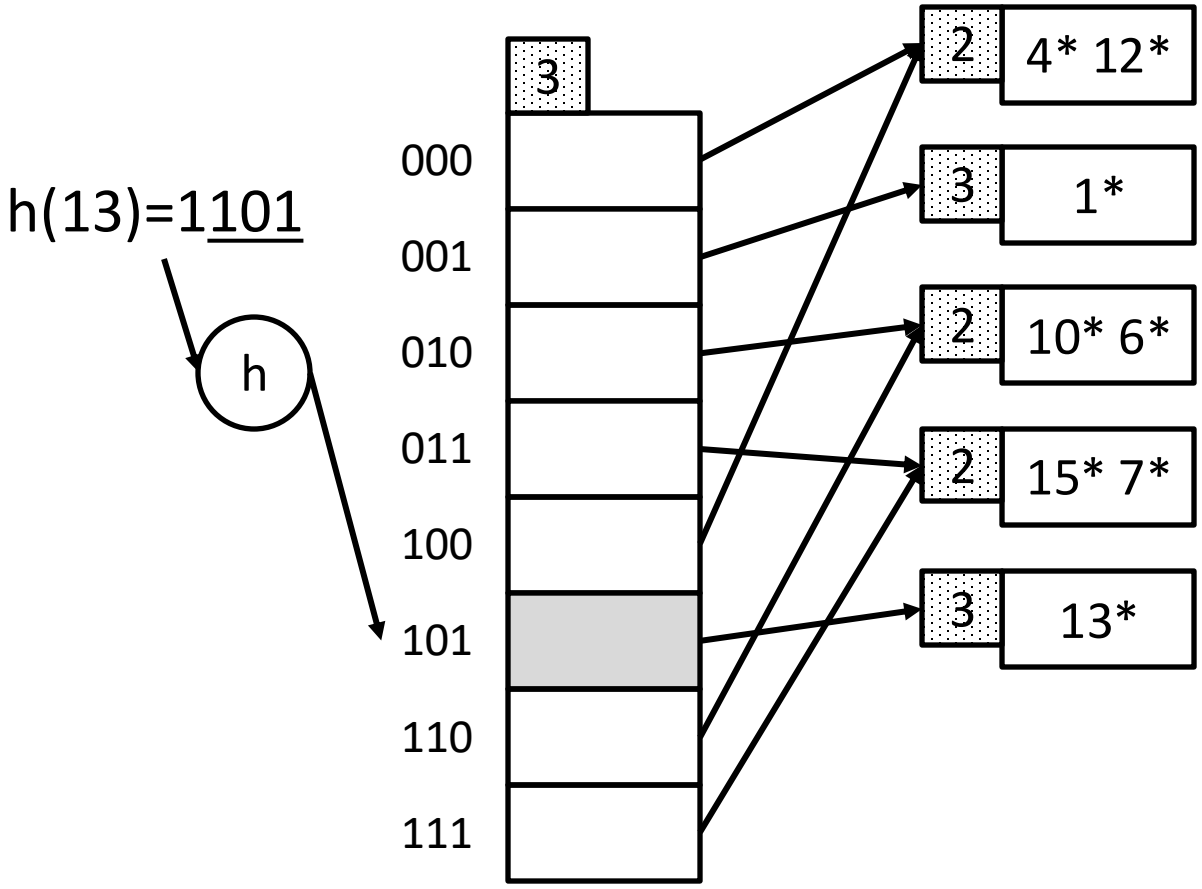
- Multiple entries with same hash value cause problems!

Delete: Reverse of inserts

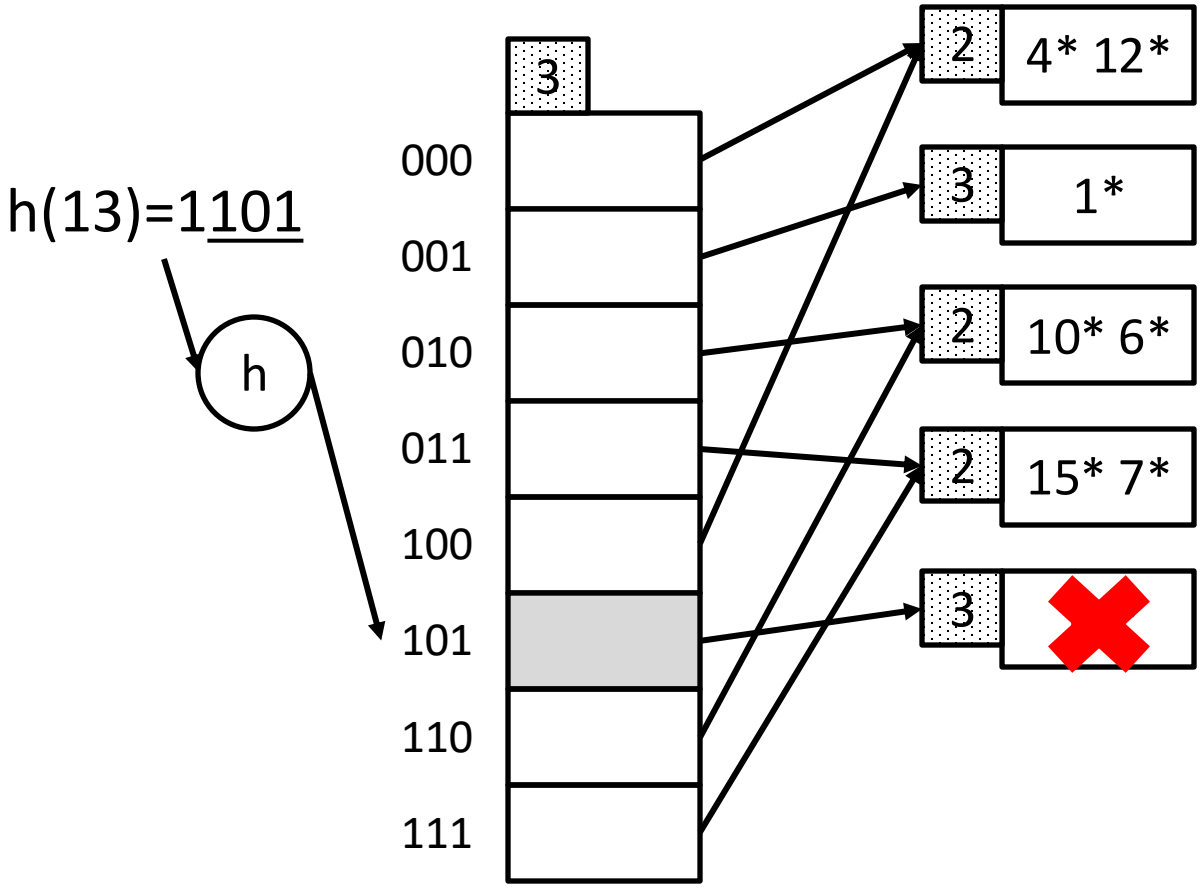
- Can merge with split image
- Can shrink the directory by half. When?

  Each directory element points to same bucket as its split image

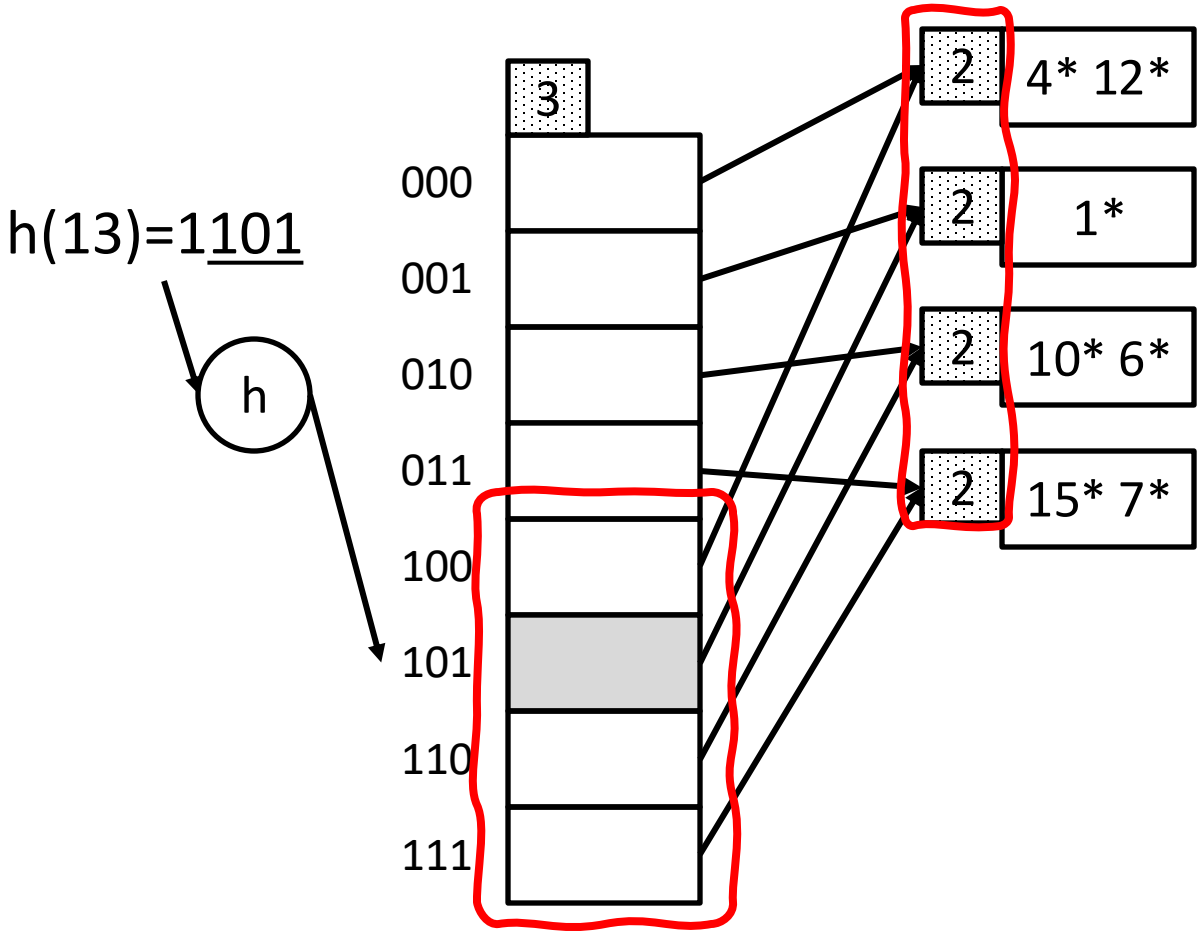
Example: Delete 13



Example: Delete 13



Example: Delete 13



all of the bucket have a smaller local depth than the global one

half of directory points to the bucket of the other half

Notes on Extendible Hashing

Do we ever need overflow pages?

- Multiple entries with same hash value cause problems!

Delete: Reverse of inserts

- Can merge with split image
- Can shrink the directory by half. When?

  Each directory element points to same bucket as its split image
- Is shrinking/merging a good idea?

Hash Indexing

Static Hashing

Extendible Hashing

Linear Hashing

Linear Hashing

another dynamic hashing scheme

LH handles overflow chains without a directory

Idea: Use overflow pages, and split pages in a round-robin fashion

Example

this for information reasons!
it is not really kept.

h_1	h_0	
000	00	4* 8*
001	01	1* 13*
010	10	10*
011	11	15* 7*

Next bucket to split

what happens when we insert 5?

$h_0(5) = 01$

what are the two hash functions?

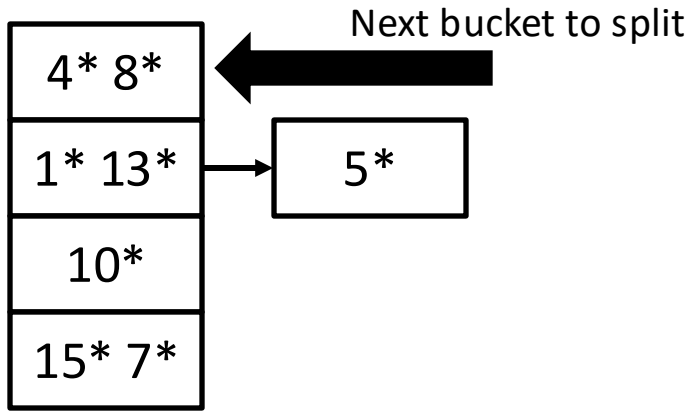
$h_0(key) = key \bmod 4$

$h_1(key) = key \bmod 8$

Example

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11



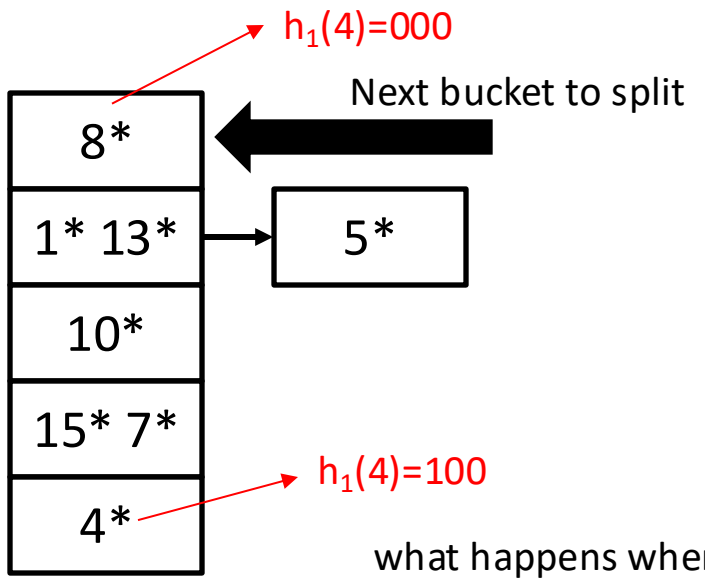
what happens when we insert 5?

(1) 5 goes to an overflow page

Example

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11
100	



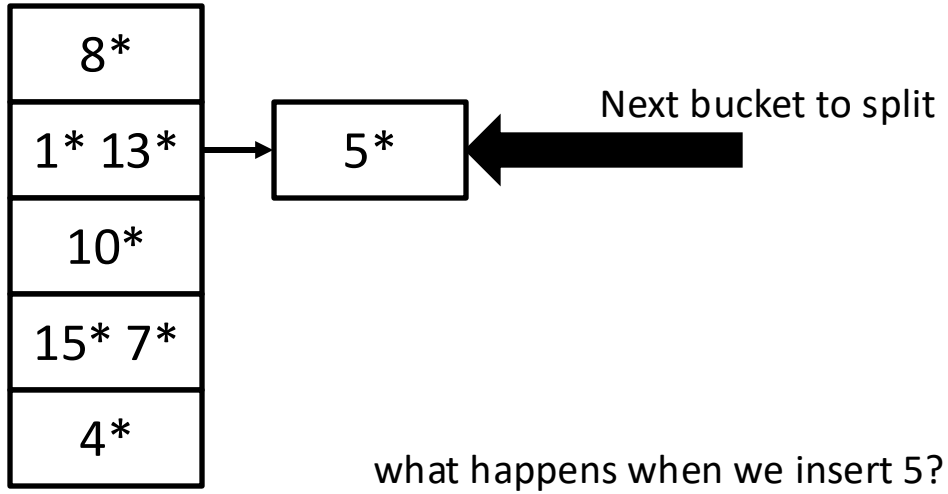
what happens when we insert 5?

(1) 5 goes to an overflow page
(2) we split the ”next” page

Example

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11
100	



what happens when we insert 5?

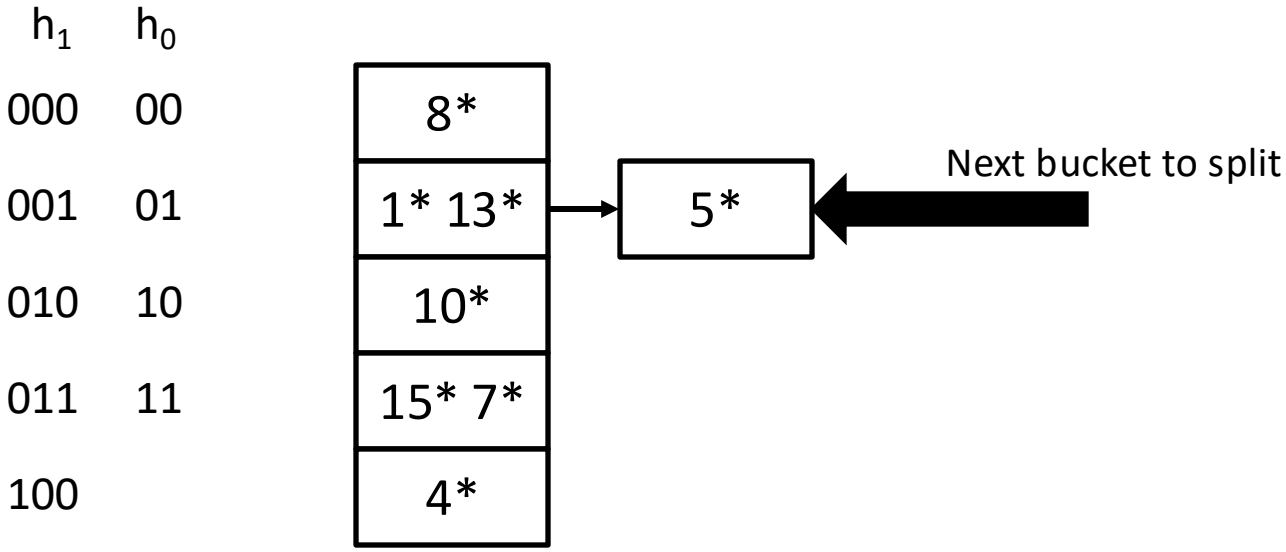
(1) 5 goes to an overflow page
(2) we split the ”next” page
(3) we move the ”next” pointer

More Details on Linear Hashing

- We can have many hash functions h_0 h_1 h_2 …
- if h_i maps data to M buckets, h_{i+1} maps data to 2M buckets
- How to build such a family?
  - pick an initial hash function h(key) that maps to an initial number of *N* buckets
  - $h_i(key)=h(key) \bmod (2^i N)$
- *Rounds* of splitting
  - During round r only hash functions h_r and h_{r+1} are in use
  - When querying during round r
    - we first query with h_r
    - if the resulting bucket is split, we then query with h_{r+1}

Example: Insert 2

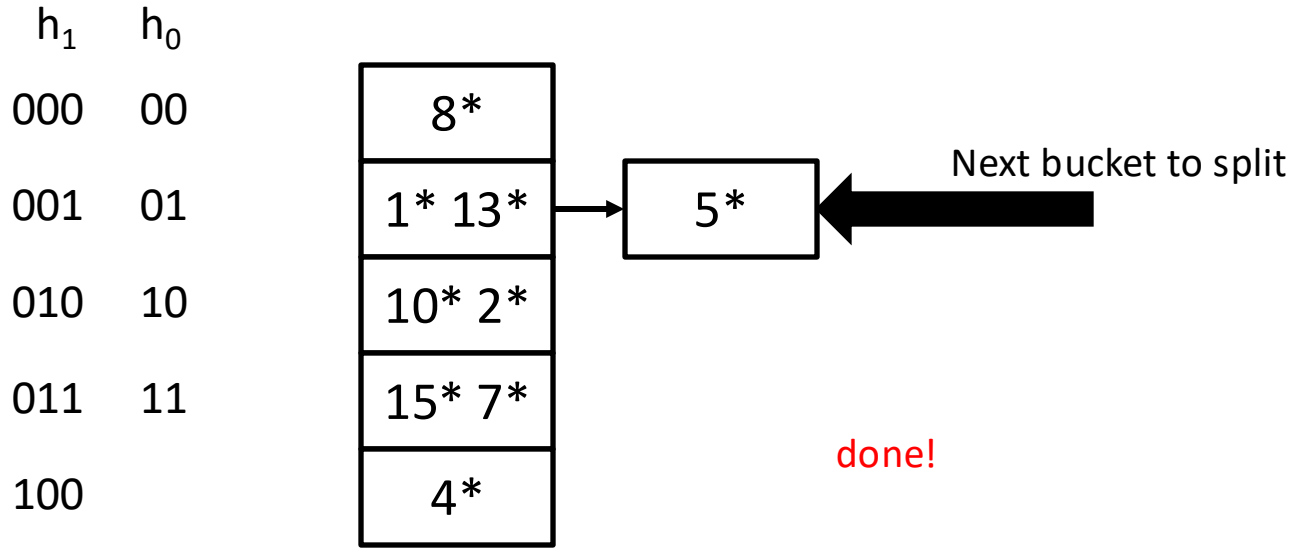
this for information reasons!
it is not really kept.



$h_0(2) = 10$

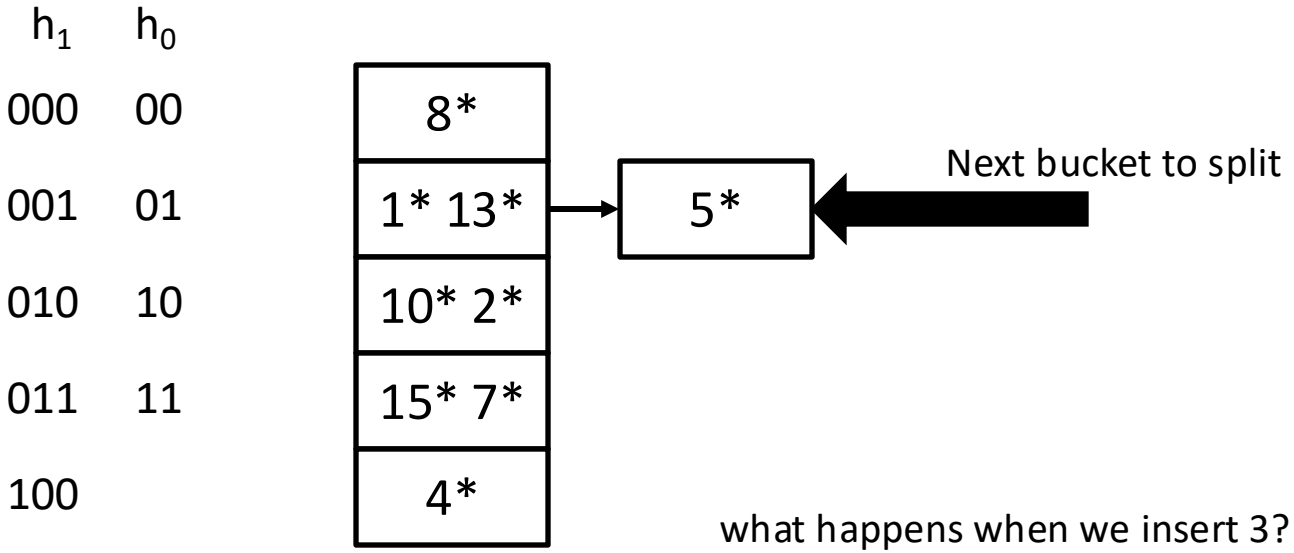
Example: Insert 2

this for information reasons!
it is not really kept.



Example: Insert 3

this for information reasons!
it is not really kept.

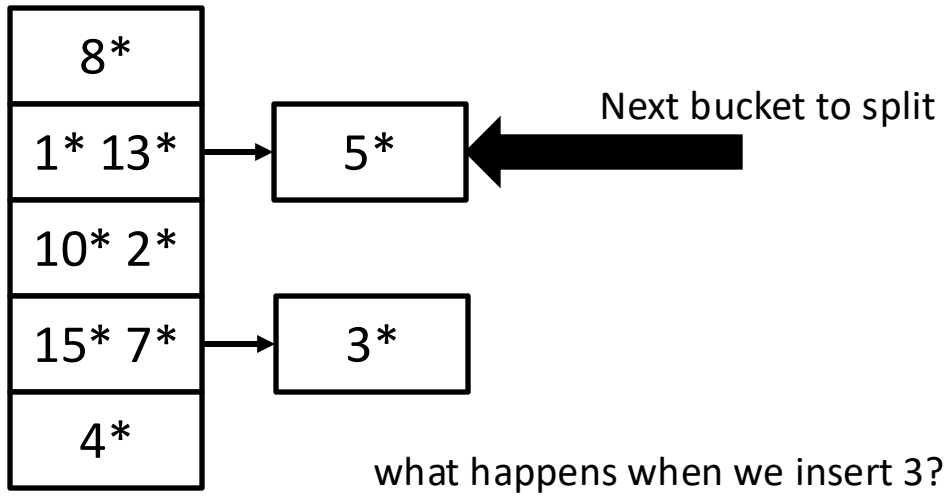


what happens when we insert 3? $h_0(3) = 11$

Example: Insert 3

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11
100	

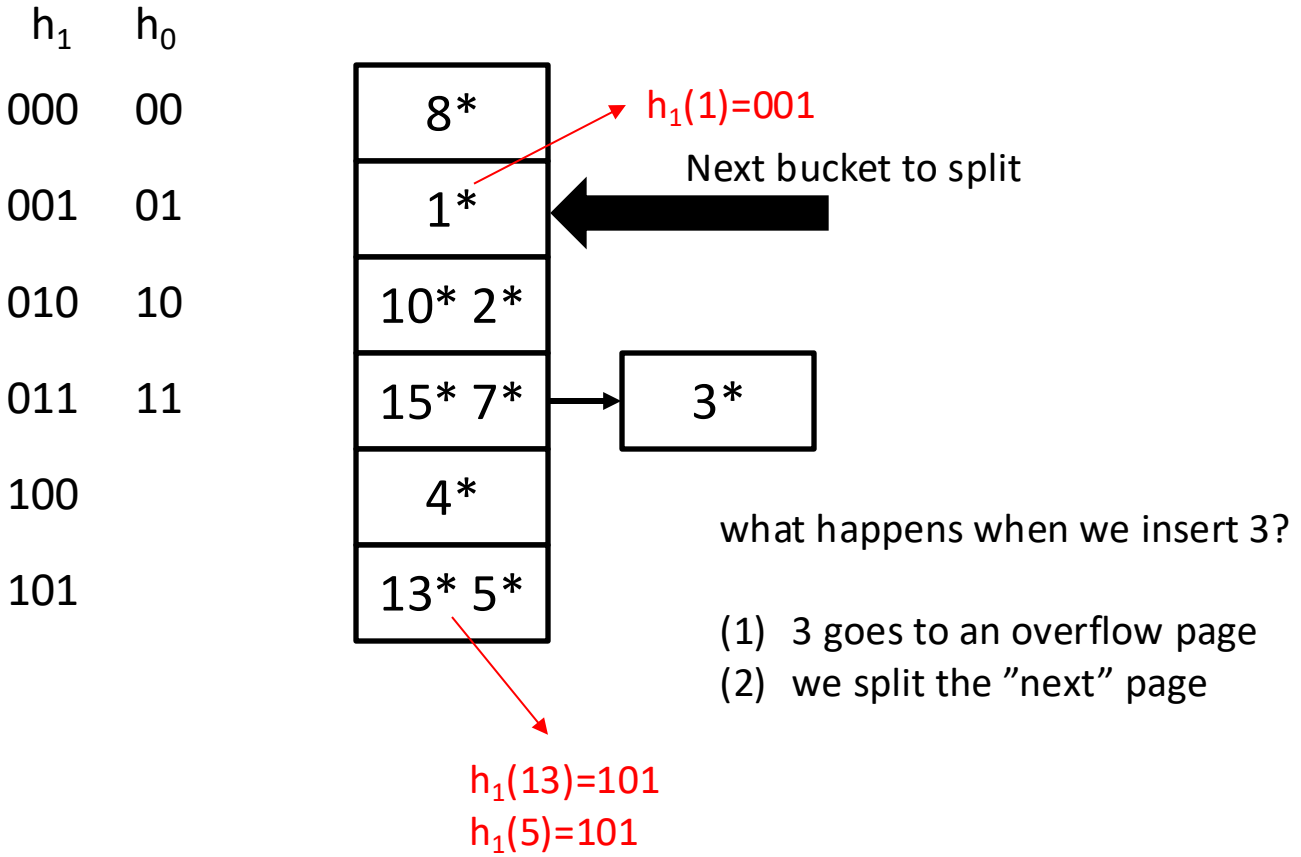


what happens when we insert 3?

(1) 3 goes to an overflow page

Example: Insert 3

this for information reasons!
it is not really kept.



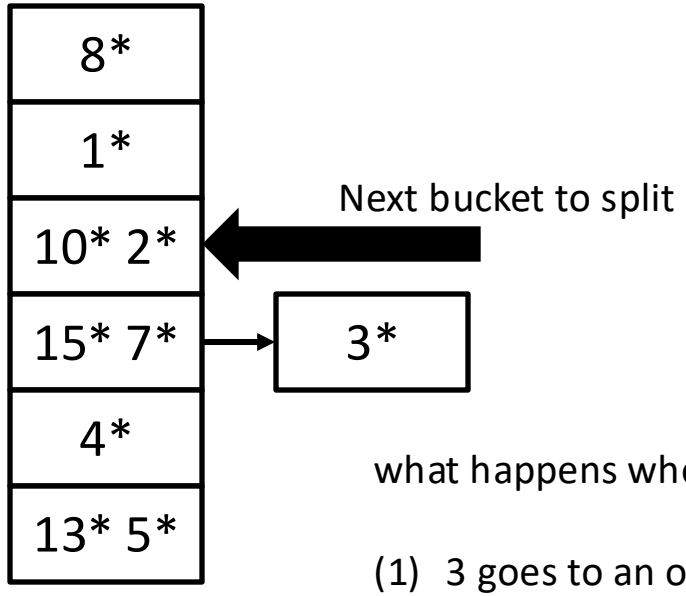
what happens when we insert 3?

(1) 3 goes to an overflow page
(2) we split the ”next” page

Example: Insert 3

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11
100	
101	

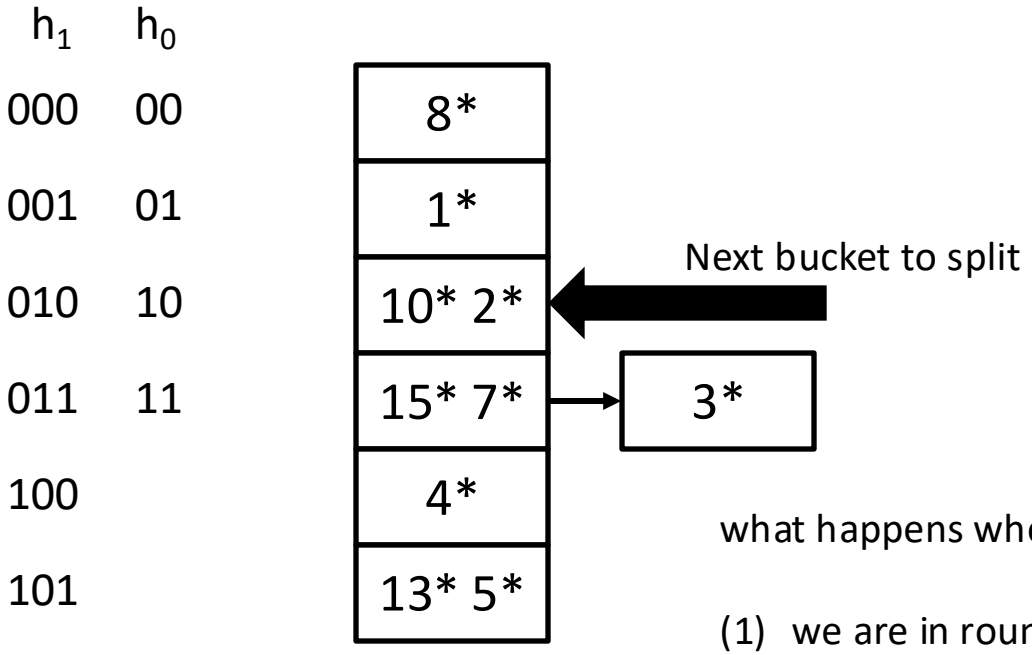


what happens when we insert 3?

(1) 3 goes to an overflow page
(2) we split the ”next” page
(3) we move the ”next” pointer

Example: Query 4

this for information reasons!
it is not really kept.



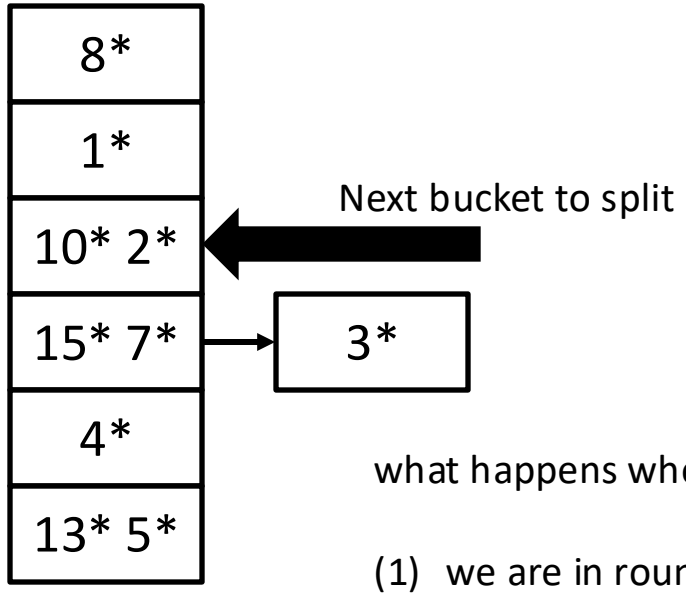
what happens when we query 4?

(1) we are in round 0, so $h_0(4)=00$
(2) bucket 00 does not contain 4 but is split
(3) $h_1(4)=100$
(4) bucket 100 contains 4

Example: Query 19

this for information reasons!
it is not really kept.

h_1	h_0
000	00
001	01
010	10
011	11
100	
101	

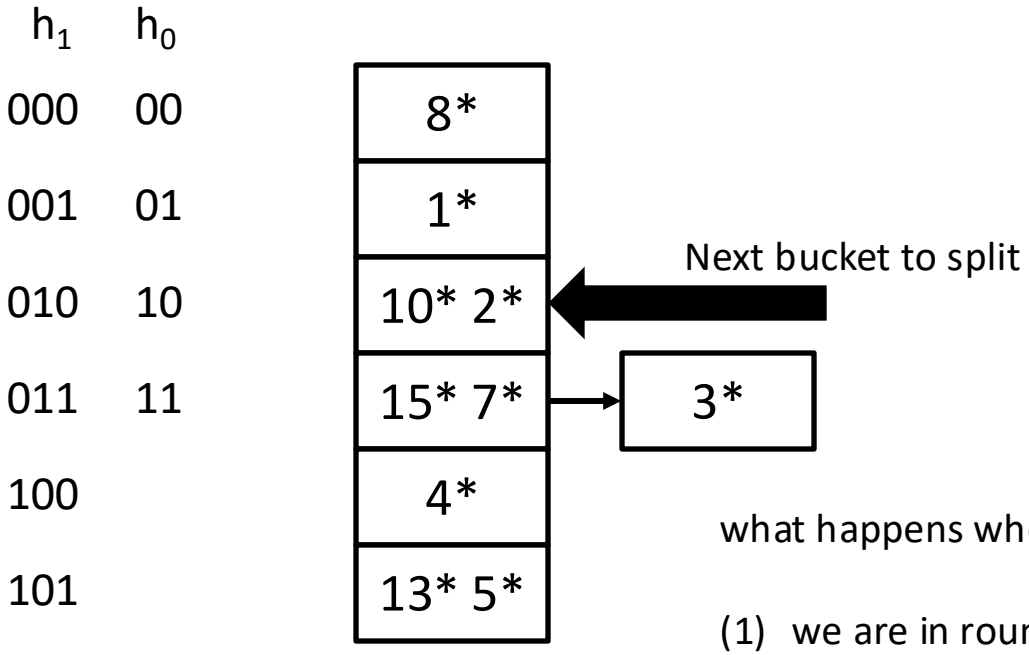


what happens when we query 19?

(1) we are in round 0, so $h_0(19)=11$
(2) bucket 11 does not contain 19 and it is not split
(3) query terminates

Example: Query 1

this for information reasons!
it is not really kept.



what happens when we query 1?

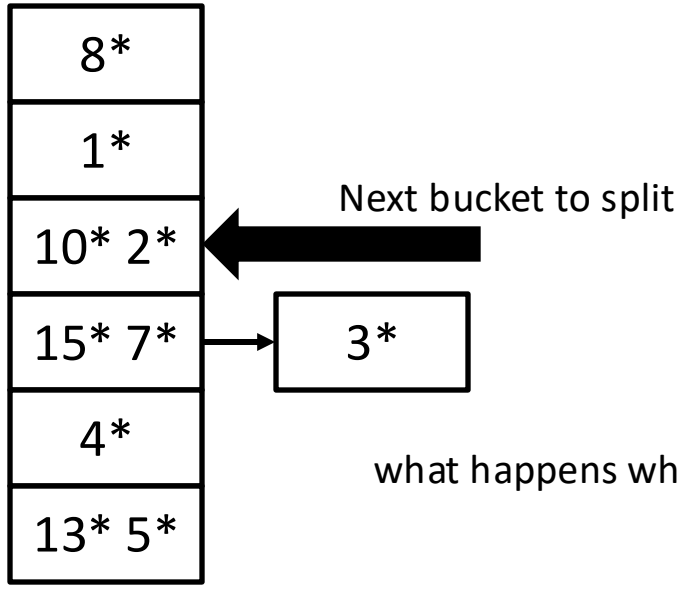
(1) we are in round 0, so $h_0(1)=01$
(2) bucket 01 does contain 1 but it is split
(3) no need to use $h_0(1)$, query terminates

Example: Insert 26

this for information reasons!
it is not really kept.

corner case

h_1	h_0
000	00
001	01
010	10
011	11
100	
101	



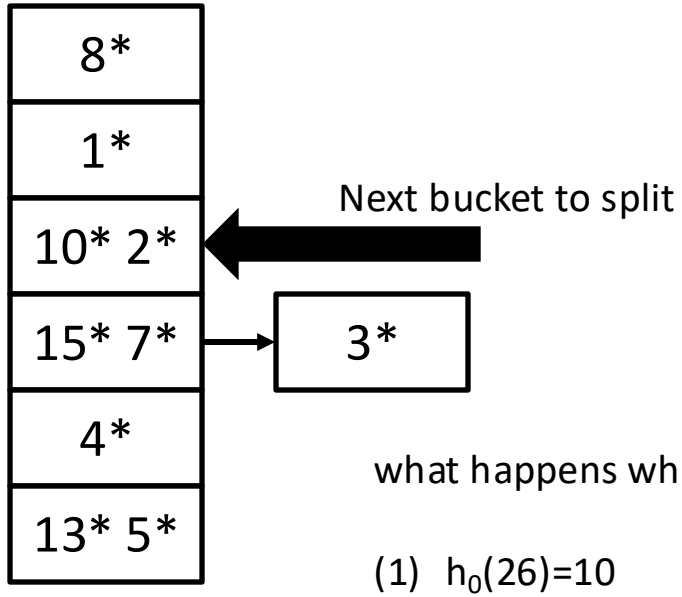
what happens when we insert 26?

Example: Insert 26

this for information reasons!
it is not really kept.

corner case

h_1	h_0
000	00
001	01
010	10
011	11
100	
101	



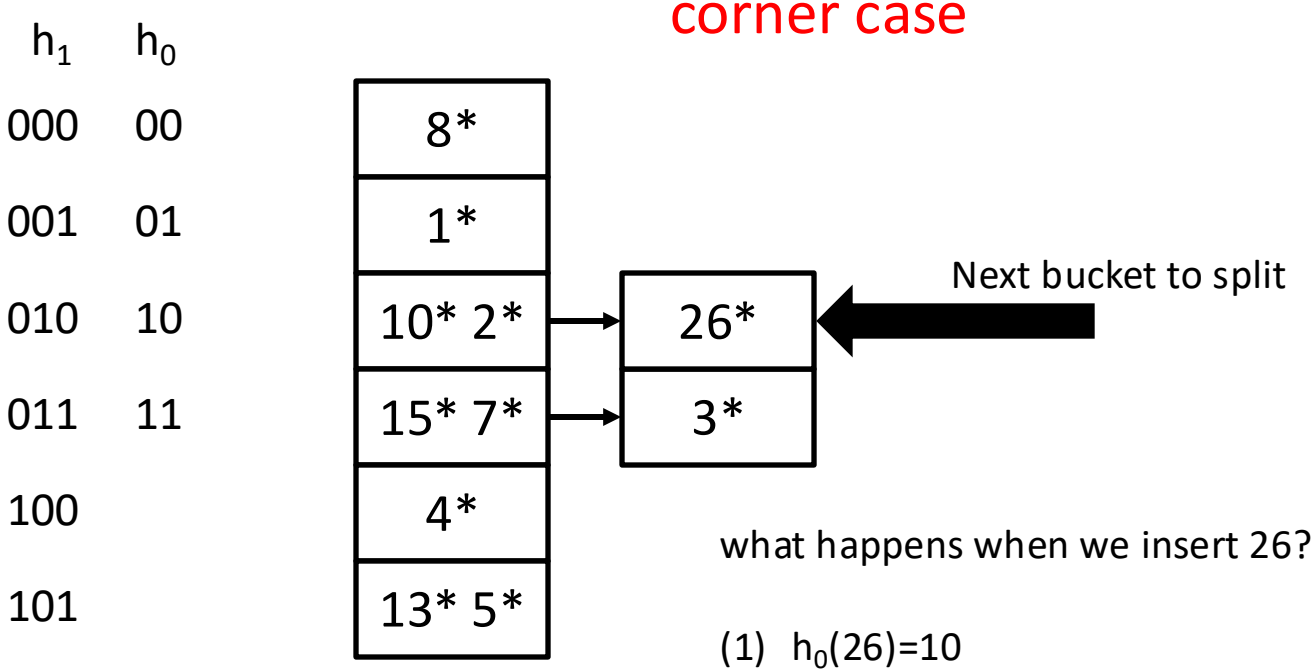
what happens when we insert 26?

(1) $h_0(26)=10$

Example: Insert 26

this for information reasons!
it is not really kept.

corner case

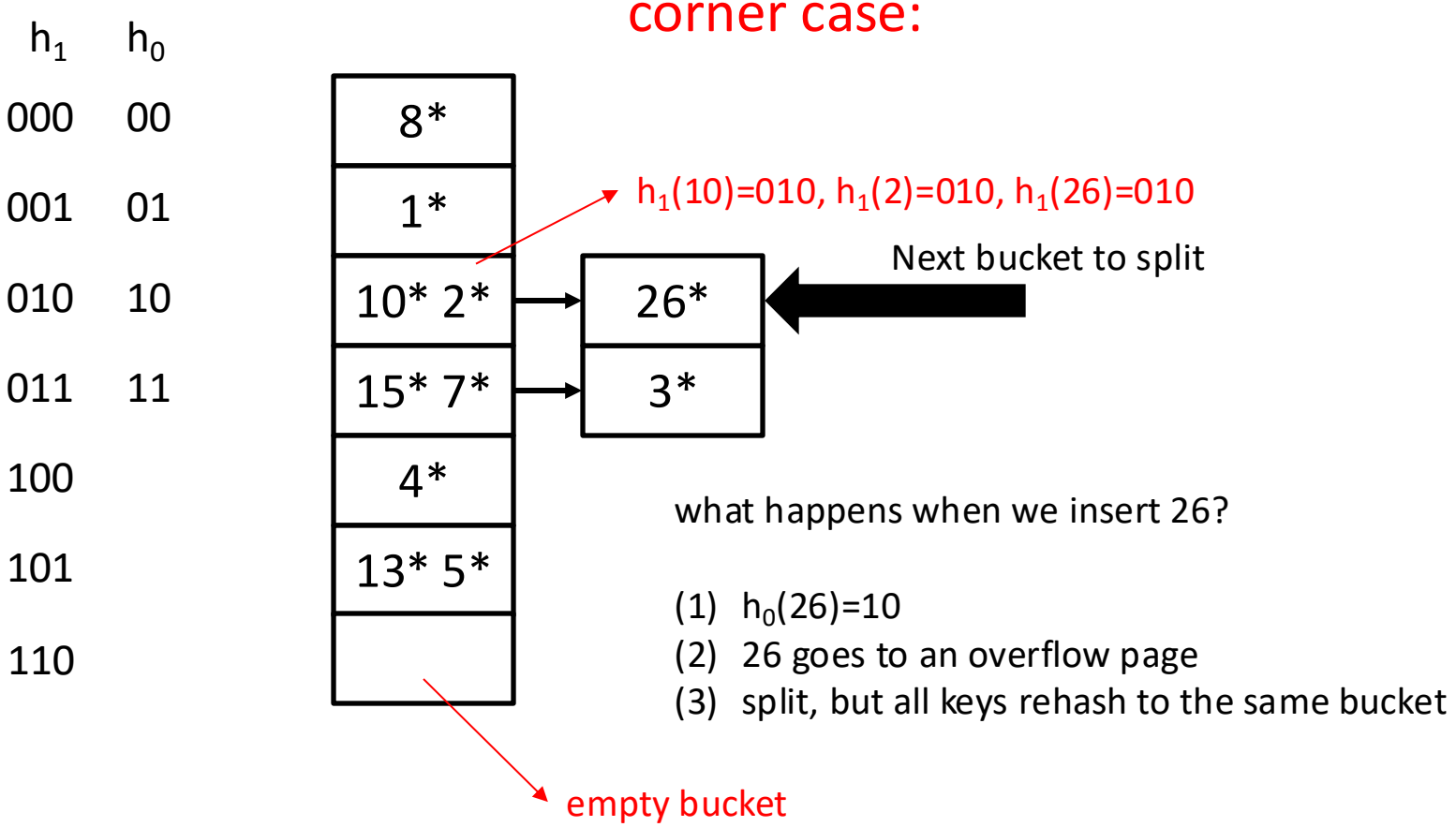


what happens when we insert 26?

(1) $h_0(26)=10$
(2) 26 goes to an overflow page

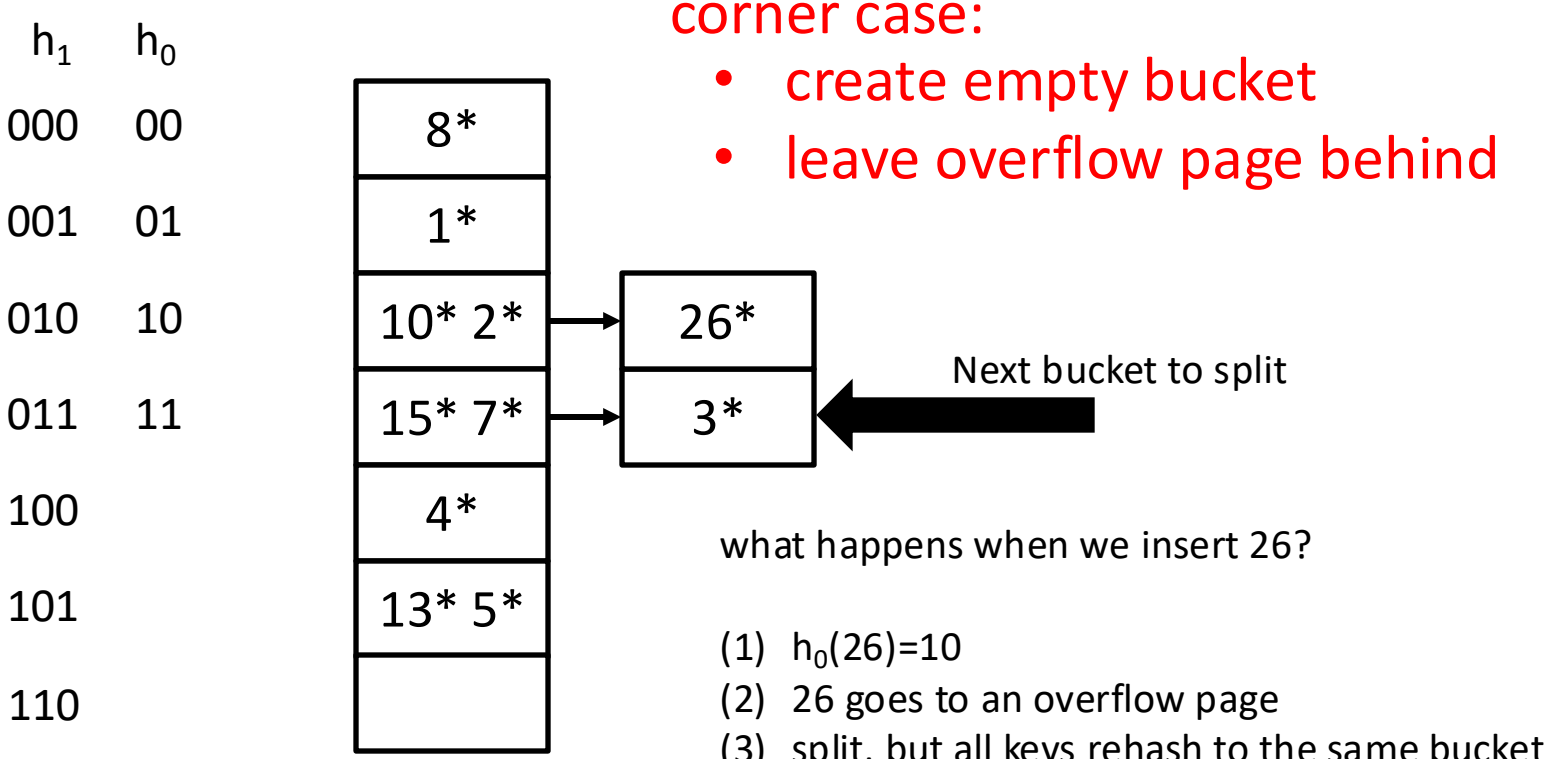
Example: Insert 26

this for information reasons!
it is not really kept.



Example: Insert 26

this for information reasons!
it is not really kept.



corner case:

- create empty bucket
- leave overflow page behind

what happens when we insert 26?

(1) $h_0(26)=10$
(2) 26 goes to an overflow page
(3) split, but all keys rehash to the same bucket
(4) move the ”next” pointer

but, good news: overflown pages will be redistributed in future rounds!

Linear Hashing

h_0, h_1 ,h_2 … can be more general hash functions

when h_0 hits on a split buffer we employ h_1 and we have to look in both buffers

if the second is also split we use h_2 and so on

Benefit: buckets are split round-robin

→ no long chains

Hash Indexing

Hash indexes: best for **equality** searches

Static Hashing can lead to long **overflow chains**

Extendible Hashing

avoids overflow pages by splitting a bucket when full
directory to keep track of buckets
BUT dir. can get too large (>memory) when data is skewed

Linear Hashing

avoids directory by splitting buckets round-robin
uses overflow pages
overflow pages not likely to be long