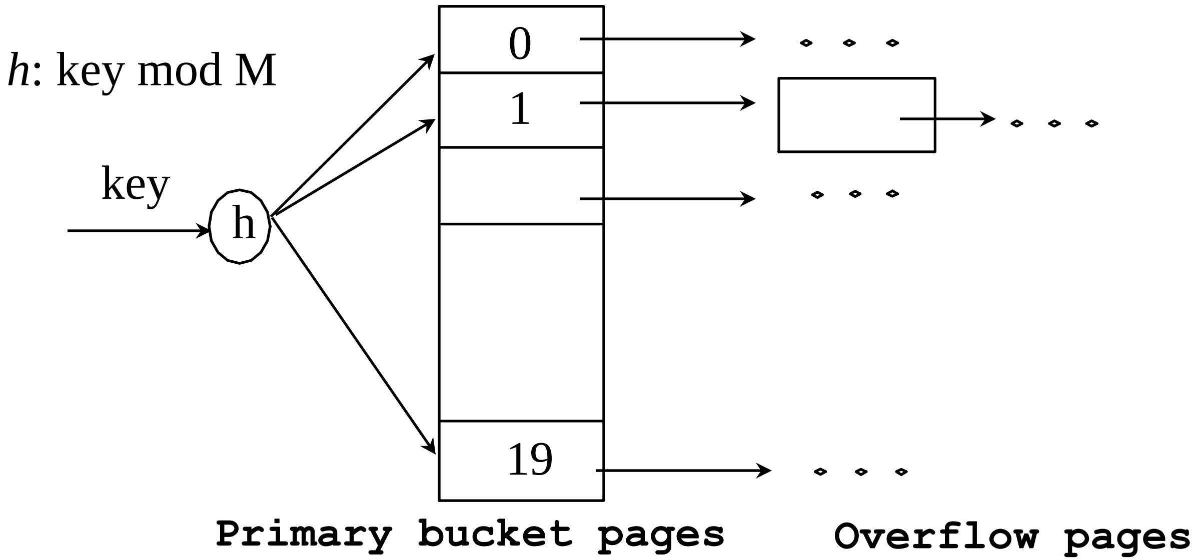
Hash-based Indexes

Introduction

- *As for any index, 3 alternatives for data entries* **k***
  - Data entries are kept in ***buckets*** (an abstract term)
  - Each bucket is a collection of one primary page and zero or more overflow pages
  - Given a search key value, k, we can find the bucket where the data entry k* is stored as follows:
    - Use a function, called *hash function*, denoted as *h*
    - The value of *h*(k) is the address for the desired bucket (i.e, the address of a bucket is represented by the address of its primary page)
    - *h*(k) should distribute the search key values uniformly over the collection of buckets
- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.
- Static and dynamic hashing techniques exist

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- A simple hash function can be: **h**(k) = f(k) mod M where M = # of buckets and f(k) = a × k + b
- Example: f(k) = k. Let M = 20. Thus h(k) = k mod 20
  - Assume each page contains two entries
  - For k = 1, 21, 41, one of them must go to overflow page



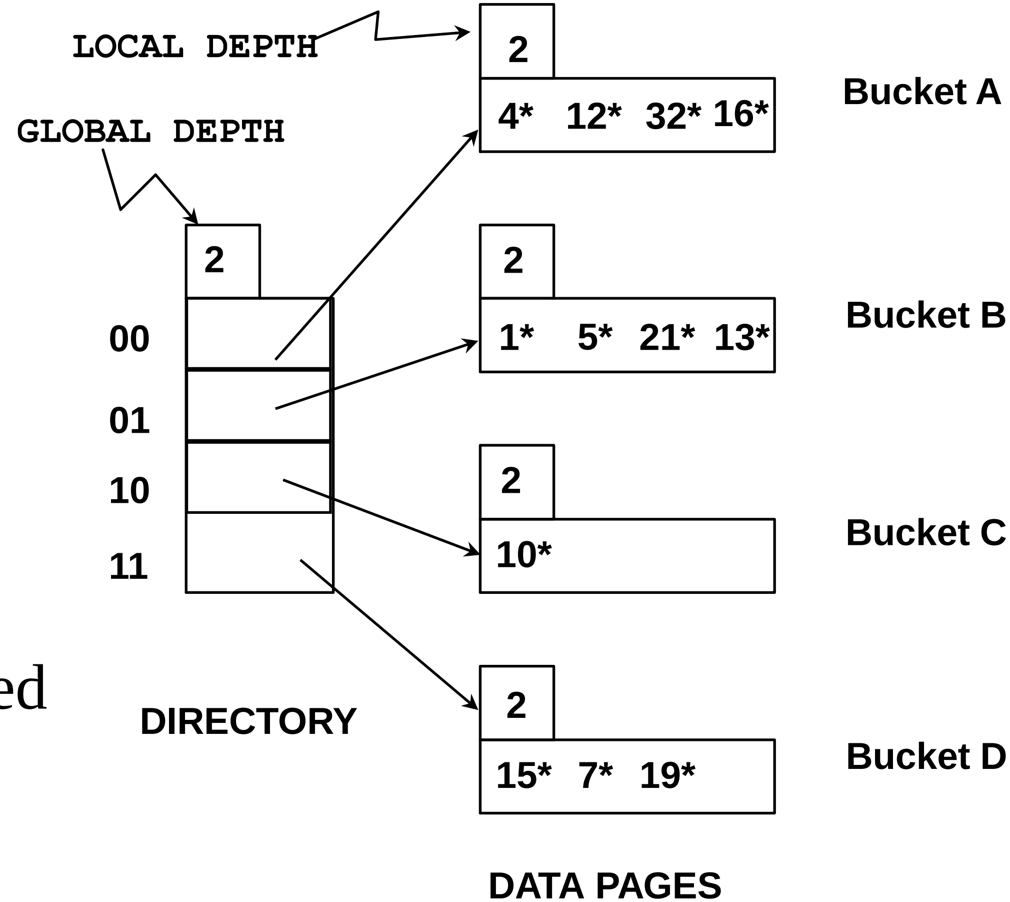
Static Hashing (Cont.)

- Buckets contain *data entries*.
- Hash function must distribute values over range of 0 ... M-1.
- It should be random and uniform.
  - If search key values greatly outnumber M, then many different key values may be hashed to the same bucket
  - Consider what happens if M = 20 and there are 1000 different search key values:
    - At least one bucket contains 50 values.
    - If the size of a page is 2, then that bucket contains 25 pages: 1 primary and 24 overflow pages
  - Therefore, long overflow chains can develop and degrade performance.
  - *Extendible hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Must re-hash all data entries to the right buckets
  - Example: assume hash function h(k) = k mod M
    - For M = 4, entries 3* and 7* both in bucket 3 (3 mod 4 = 7 mod 4 = 3)
    - But for M = 8, entry 7* will be in bucket 7
  - Can we only re-hash those values that have changed addresses?
    - Difficulties: without re-hashing all the values, we don't know which values keep the old addresses and which get new addresses
  - Reading and writing all pages are expensive!
  - Question: how do we add more buckets, but only re-hash a few data entries?
  - Answer: use a level of indirection, ***directory of pointers to buckets***

Example

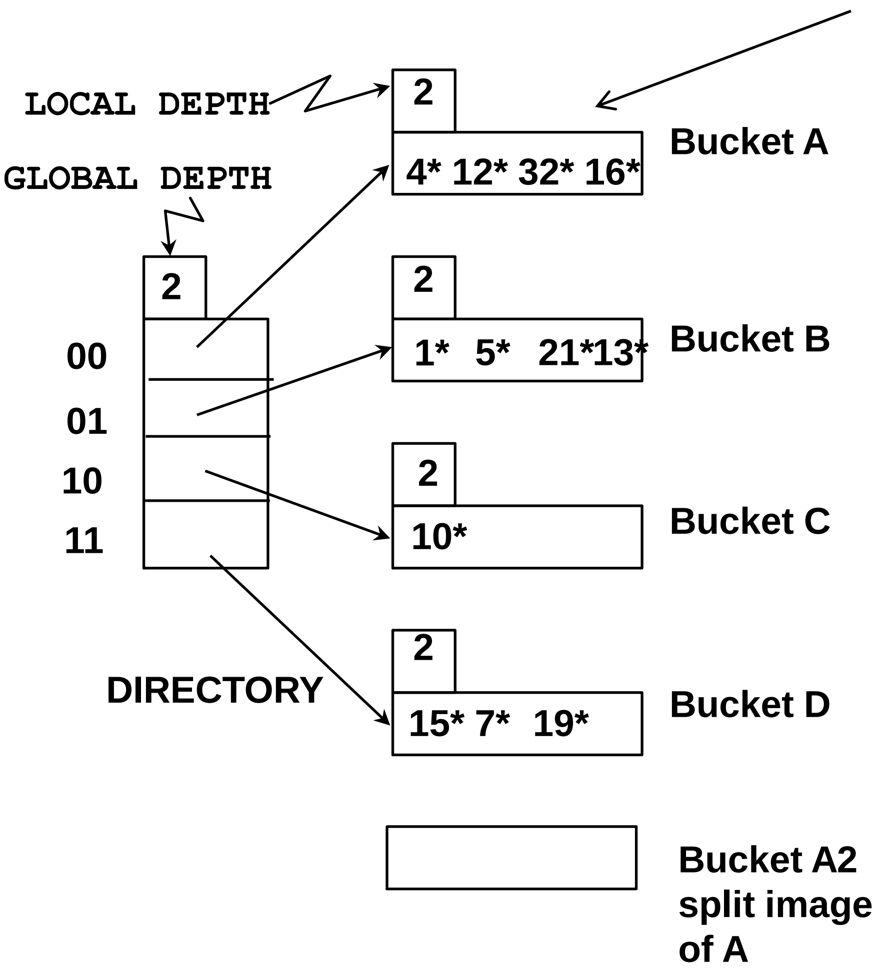


- **h**(r) = *r mod 32*
- Directory is array of size 4.
- To find bucket for r, take last \`*global depth*' # bits of **h**(r)
  - If r = 5, **h**(r) = 5 = binary 101, 5* is in bucket pointed to by 01.

❖ **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.)

Example (cont.): Insert 20*



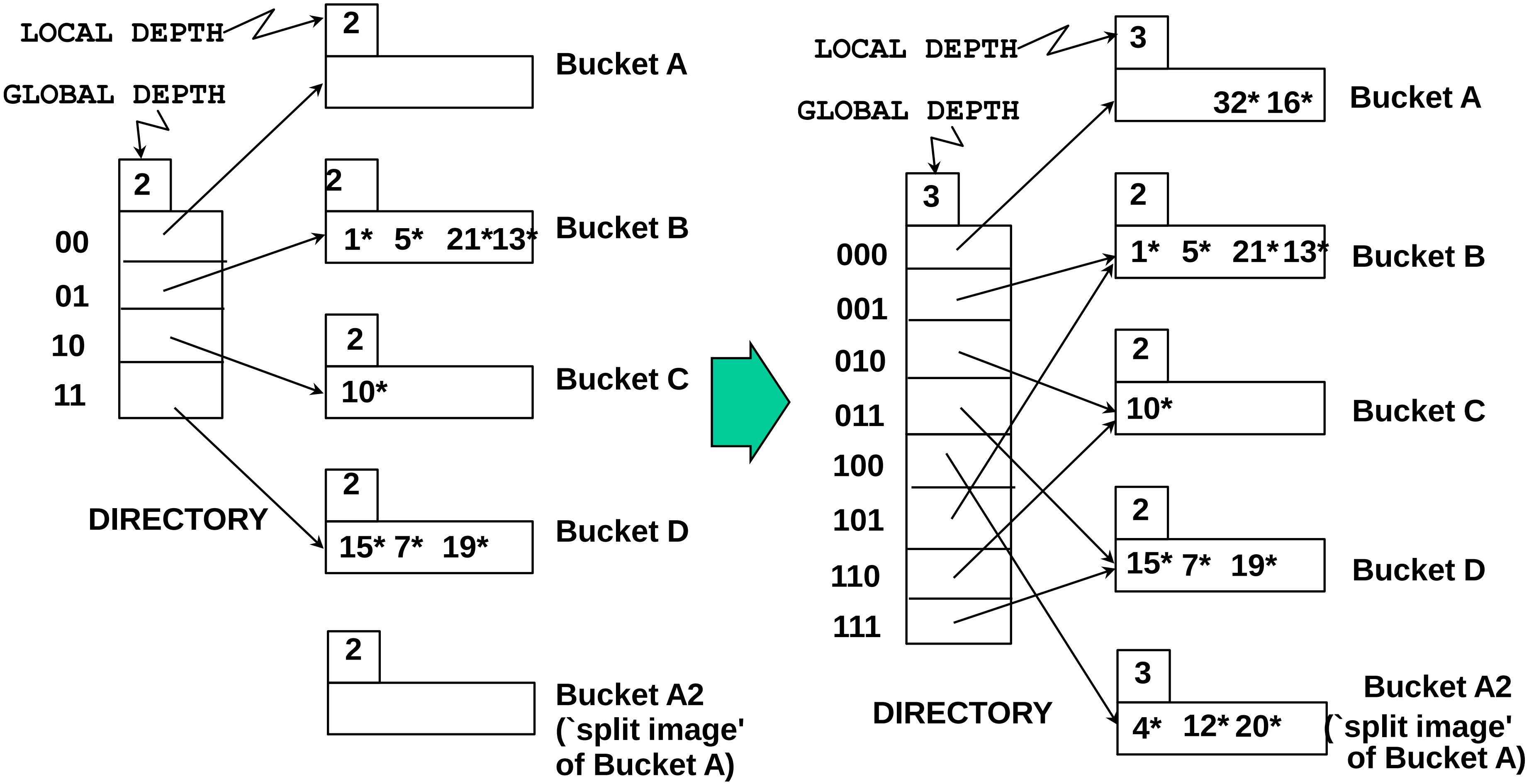
Insert 20*, causing overflow, we do the following:

- Split bucket A to A &A2
- for the five data entries, 4*, 12*,32*,16*,20*, if for any r* its 3rd bit in h(r) is 1, then move it to A2:

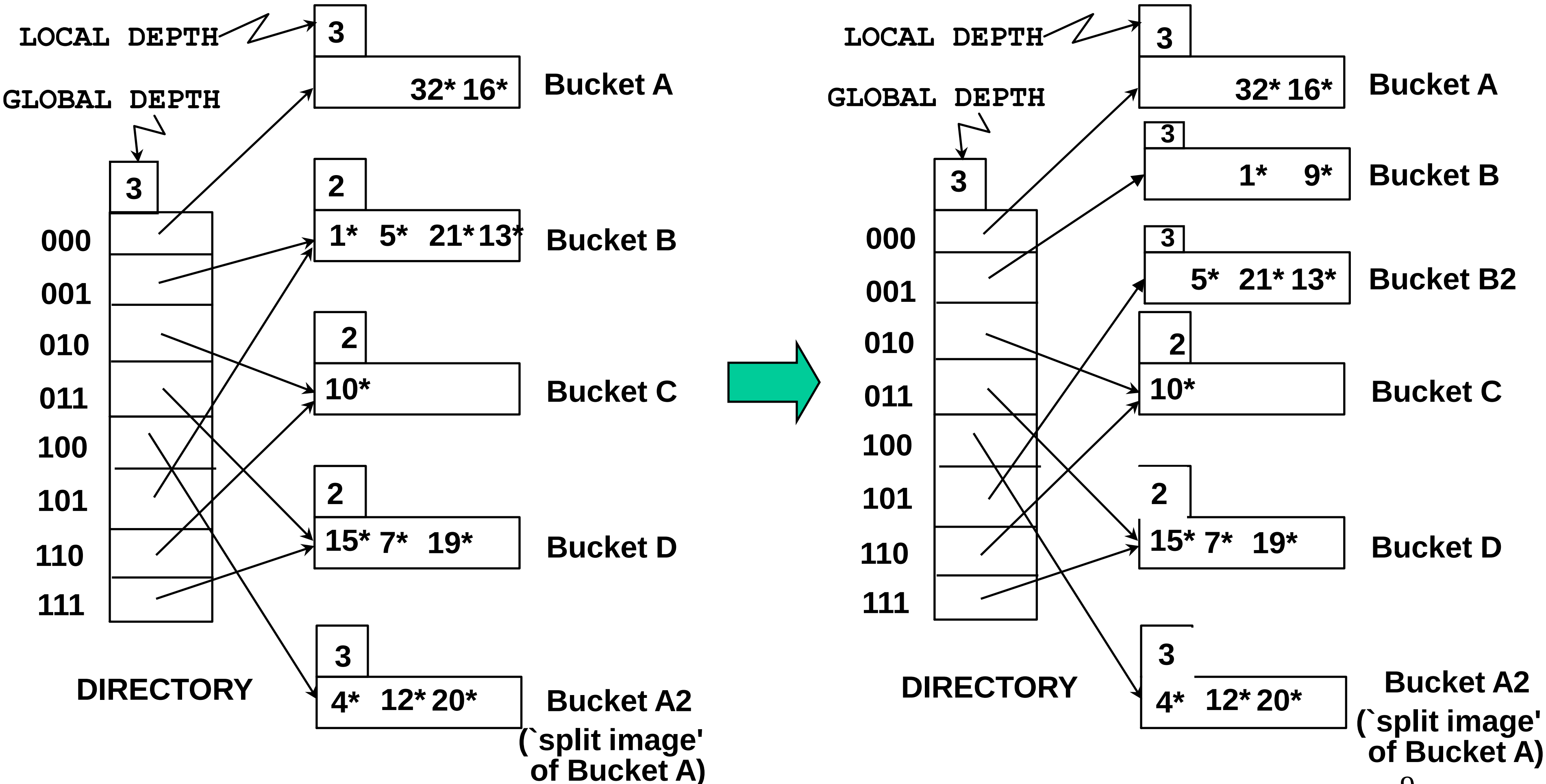
h(4) = 000100
h(12) = 001100
h(32) = 100000
h(16) = 010000
h(20) = 010100

Insert h(r)=20 (Causes Doubling)

h(16)=010000
h(32)=000000
h(20)=010100
h(12)=001100
h(4) =000100



Insert 9 (Does Not Cause Doubling): h(9) = 01001

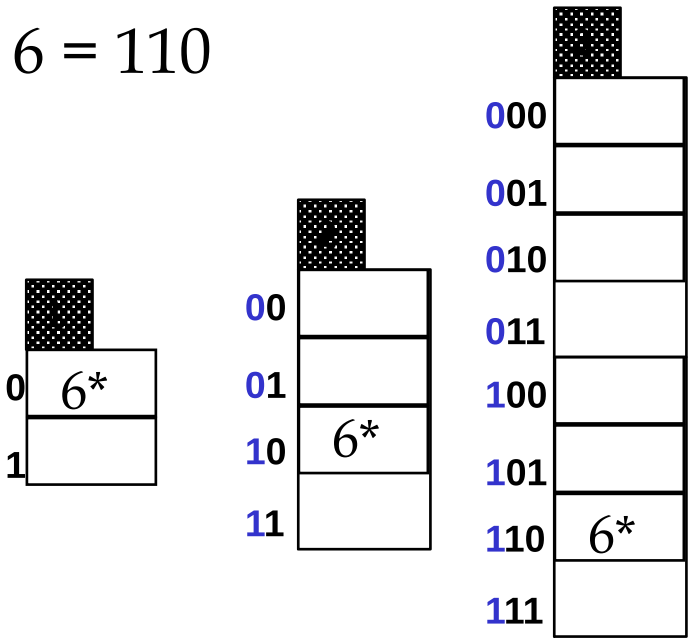


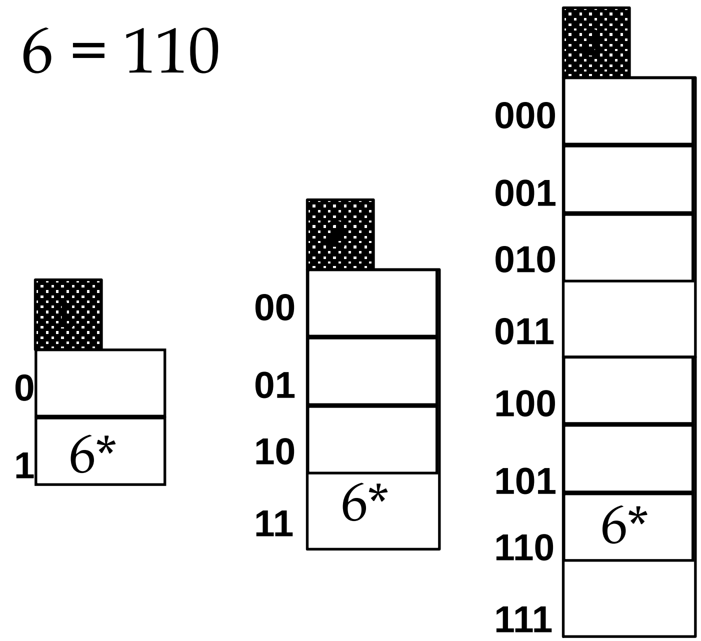
Points to Note

- h(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 # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if the directory need doubling. How?
- When does bucket split cause directory doubling?
  - Before insertion, *local depth* of bucket <= *global depth*.
  - After insertion, if overflow, generate split image, and increment the *local depth*
  - If this causes *local depth* > *global depth*, then directory is doubled, and at the same time increment *global depth*
  - Doubling directory is done by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Directory Doubling

Why use least significant bits in directory?
⇔ Allows for doubling via copying!





Least Significant vs. Most Significant

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed (e.g., a large number of search key values all are hashed to the same bucket), directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to the same bucket as its split image, can halve directory.

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. (*Duplicates may require overflow pages*.)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.