SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (2/6)

- indexing
  - one- and multidimensional
  - tree-structured
  - partition-based indexing
  - bulk-loading
  - main-memory indexing
  - hash-indexes
  - differential indexing
  - read-optimized indexing
  - write-optimized indexing
  - data warehouse indexing
  - text indexing: inverted files
  - (flash-indexing)

One-dimensional Indexes.

Hash-Indexes.

Motivation

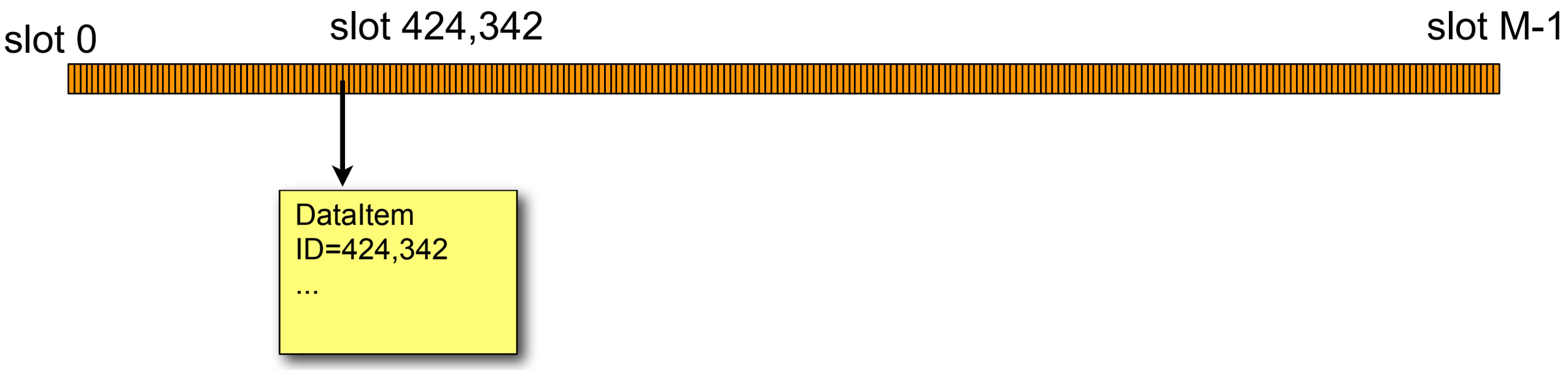
- works if range queries are not required
- otherwise hash-indexes do **not** make sense
- extremely efficient: usually O(1)
- also used as temporary index structures during query processing:
  - hash join (simple and Grace)
  - aggregation
  - duplicate elimination
  - semi-joins

Problem

- given
  - a key domain 0, ..., M-1
  - M is potentially large
  - point queries asking for data items having a key in 0, ..., M-1
- Example
  - What is the address of the student having ID 424342?

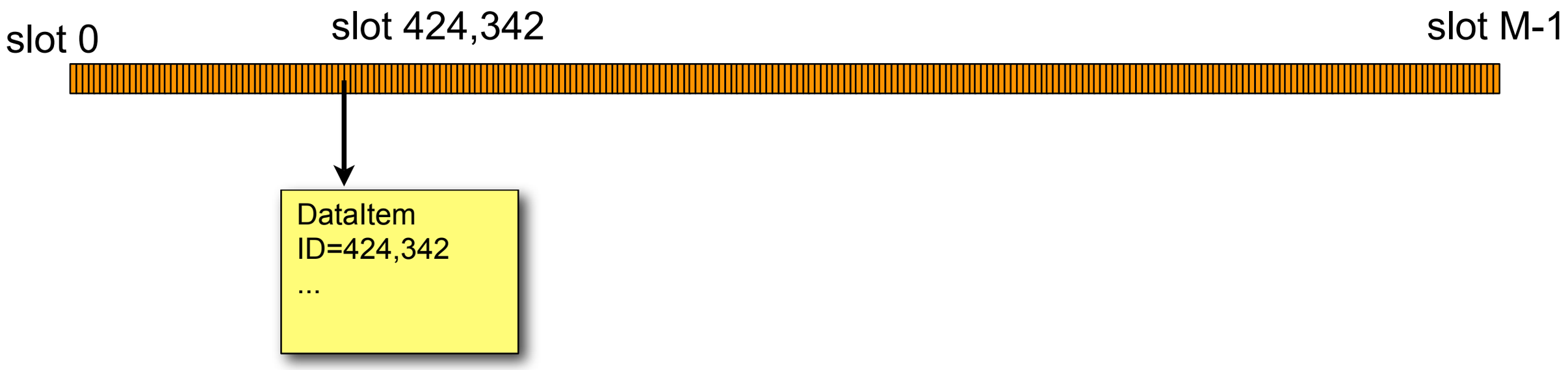
Naive Solution

- create an array a of size M:
  a = new Object[M]();
- for each array slot we keep a reference to the data item
- the reference requires at least 4 Bytes
  (depending on compiler and 32/64 bit, it may be even more)
- thus we need at least 400 MB just to allocate the **empty** array



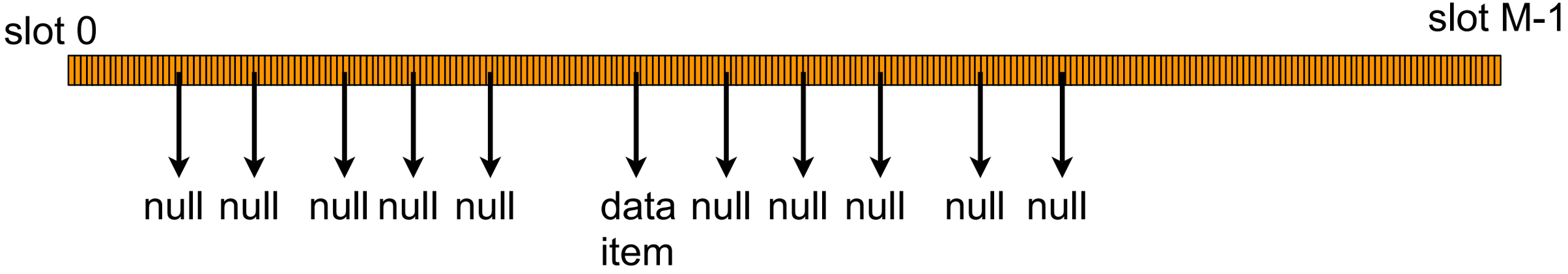
Naive Solution: Methods

- insert(Key key, DataItem x):
  - a[key] = x;
- delete(Key key):
  - a[key] = null;
- pointQuery(Key key):
  - return a[key];



Naive Solution: Space Usage

- for example, let‘s assume that M = 100,000,000
- let‘s assume that the size of the dataset N = 100,000
- => M/N = 1,000
- thus only 1 out of 1,000 array slots will be occupied
- all other array slots will contain null pointers!!
- this is a huge waste of space



Static Hashing.

Core Idea of Hashing

- try to keep the efficiency of the naive array solution
- however: reduce the size of the array
- pick an M‘ << M
- for instance M‘ = 200,000 instead of M=100,000,000
- allocate an array of size M‘
- transform any incoming key from the range 0, ..., M-1 to the range 0, ..., M‘-1
- this is achieved by a so-called **hash function** h(key)
- Design goals for hash functions:
  - distribute incoming keys evenly over key range 0, ..., M‘-1
  - avoid collisions
  - collision: different key from 0,..., M-1 maps to same key in 0, ..., M‘-1

Hash Functions: Division Method

- h(key) = **key mod M‘**
- Example: M‘ = 12, key = 100, then h(key) = 4
- only a single division operation => very fast
- Note:
  - certain values of M‘ should be avoided
  - M‘ should not be a power of 2
  - if M‘ = 2^p, then h(key) is just the p lowest bits of k
  - if all low-order p-bit patterns are equally likely, this is fine
  - however if the pattern is unknow, we should use a different hash function
  - very often a prime not too close to an exact power of 2 is a good choice
- in practice: too simple, not good enough

Hash Functions: Folding

- split binary representation of key into components
- re-combine these components using adding, shifting, multiplication, etc.
- usually fast when using only bit shiftinf and masking
- should not use too many multiplications
- Example:
  - h(key) = ((key **mod** 1000) + (key **div** 1000)) mod M‘
  - M‘ = 300 key = 424,342,
  - h(key) = (342 + 424) mod 300 = 166
- some similarity to a pseudo-random number generator...
- good method in practice as cheap, yet good distribution

Hash Functions: Random

- use a pseudo-random generator to generate a hash
- use key as a seed for that generator
- Example:
  - h(key) = {
    - Random rand = new Random(key);
    - return random.nextInt() % M‘;
  - }
  - M‘ = 300 key = 424,342 , rand.nextInt() = 1,605,688,069
  - h(key) = 1,605,688,069 % 300 = 169

seed for random engine

- Why does this work? Aren‘t we using **random** numbers?
- in practice: may be OK for external memory hashing (dominated by I/O cost)
- however too expensive for main memory method

Perfect Hashing vs. Collision Handling

- In some cases we may be able to come up with a so-called **perfect hash function**
- a hash function is called perfect if it does not produce any collisions
- highly depends on key domain whether we can come up with such a function
- Example:
  - public bus system in Saarbrücken
  - all bus lines seem to have at least three digits starting with a 1..
  - 102, 124, 170, etc
  - But are there 170 different bus lines in Saarbrücken?
  - h(key) = (key-100) mod M‘; h(102)= 2, h(124) = 24, h(170) = 70
  - in general: we have to live with collisions

Hash Collisions

- we have to be prepared to handle collisions
- for instance: recall the ‘bad‘ hash function h(key) = **key mod M‘**
- let key_x = x + y*M‘ having x in 0, ..., M‘-1, y>0
- then h($key_{x,y}$) = x for any y
- Example:
  - M‘=300, x=42
  - $key_{42,y}$ = 42+y*300 => h($key_{42,y}$)=42
  - h(42)=h(342)=h(642)=h(942)=...= 42
- This also means: if we lookup a hash-table with a key and find something in array slot a[h(key)], this does **not** mean we found the key.
- We need a final comparison with the entire key to decide.

How to fix Collisions? Overflow Chains!

- in practice an array slot will not point to a single data item
- but possibly to a list of data items
- Variant 1:
  - each array slot points to list of data items
  - this list is called an overflow chain
- Variant 2:
  - array slot points to data item
  - array slot keeps additional pointer to possible overflow chain
- in practice: better solutions exist

Linear Probing

- assume existing hash function h(key)
- create a new hash function

  h‘(key) = (h(key) + i) mod M‘

  where i = 0,1,..., M‘-1
- Intuition
  - first probe the usual array slot h(key)
  - if that is occupied, probe the next slot h(key)+1, h(key)+2, ...
  - until we reach the end of the array, then we start from array slot 0
- has to be considered for inserting and querying
- suffers from **primary clustering**
  (not be confused with clustering in indexes!)
- long runs of occupied slots build up increasing average search time

Quadratic Probing

- assume existing hash function h(key)
- create a new hash function

$$h'(key, i) = (h(key) + c_1 * i + c_2 * i^2) \bmod M'$$

where i = 0,1,..., M'-1
and c_1 and $c_2 \neq 0$

- Intuition
  - first probe the usual array slot h(key)
  - later probes depend in a quadratic manner on the probe number i
- choice of c_1, c_2, and M' are constrained
- details: see Cormen book

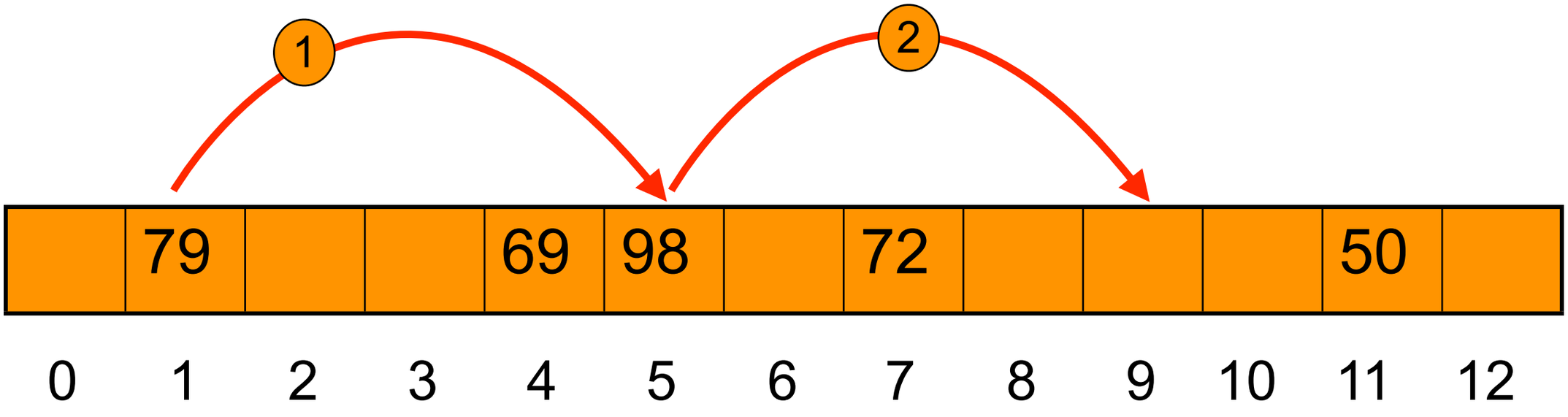
Double Hashing

- use two hash functions

  $h'(key,i) = (h(key) + i * h_2(key)) \bmod M'$.

- Intuition
  - first probe the usual array slot h(key)
  - later probes are offset from previous positions by the amount $h_2(key)$ modulo M‘
- unlike the case of linear or quadratic probing, probe sequence here depends in two ways upon the key
- initial offset position, offset, or both may vary
- improves over linear and quadratic probing
- details: Cormen book

Double Hashing Example: Insertion

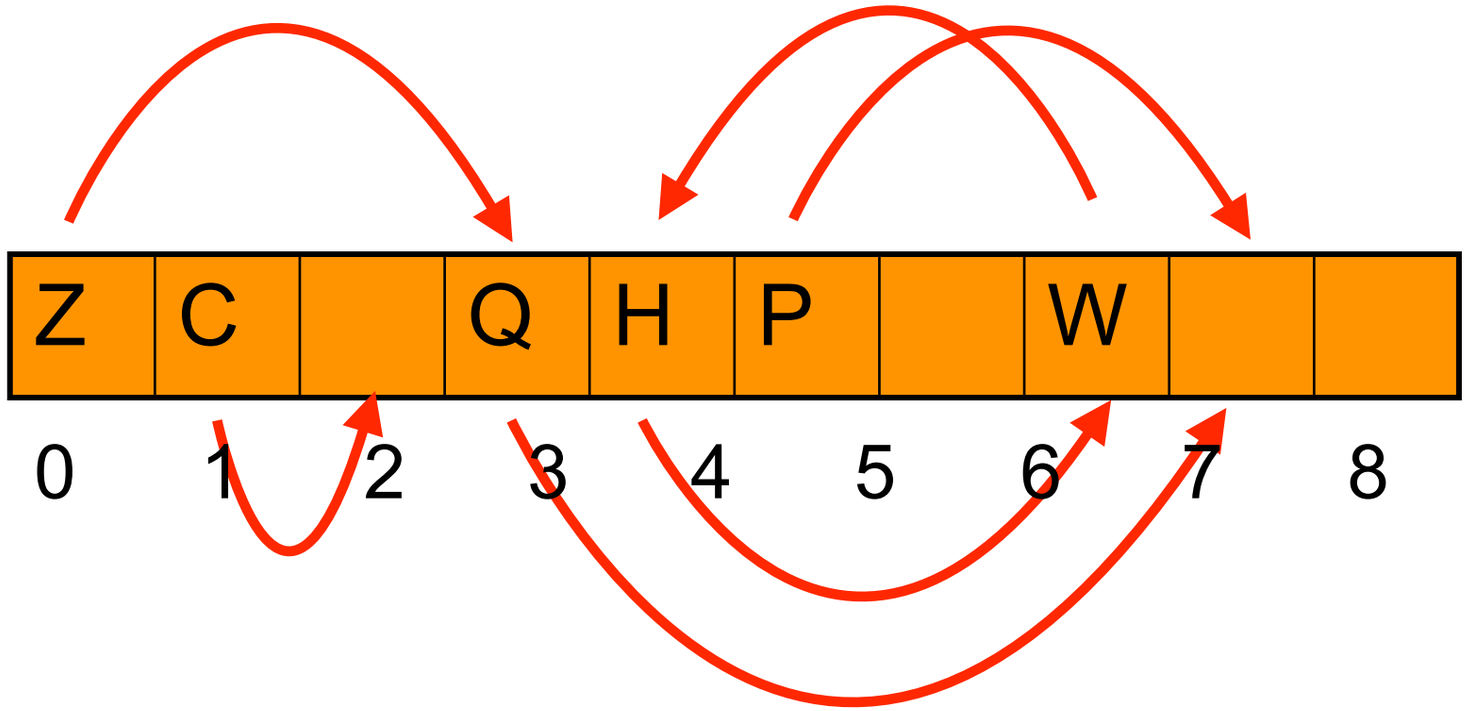


- we will try to insert the key 14, M‘=13
- h(key) = key mod 13, h_2(key) = 1+ (key mod 11)
- h‘(key,i) = (h(key) + i*h_2(key)) mod 13‘.
- 14 ≡1 (mod 13) and 14≡ 3 (mod 1)
- first probe at slot 1 -> already full
- second probe at slot 1+1+3 = 5 -> already full
- third probe at slot 1+2*4 = 9 -> empty -> insert

Cuckoo Hashing

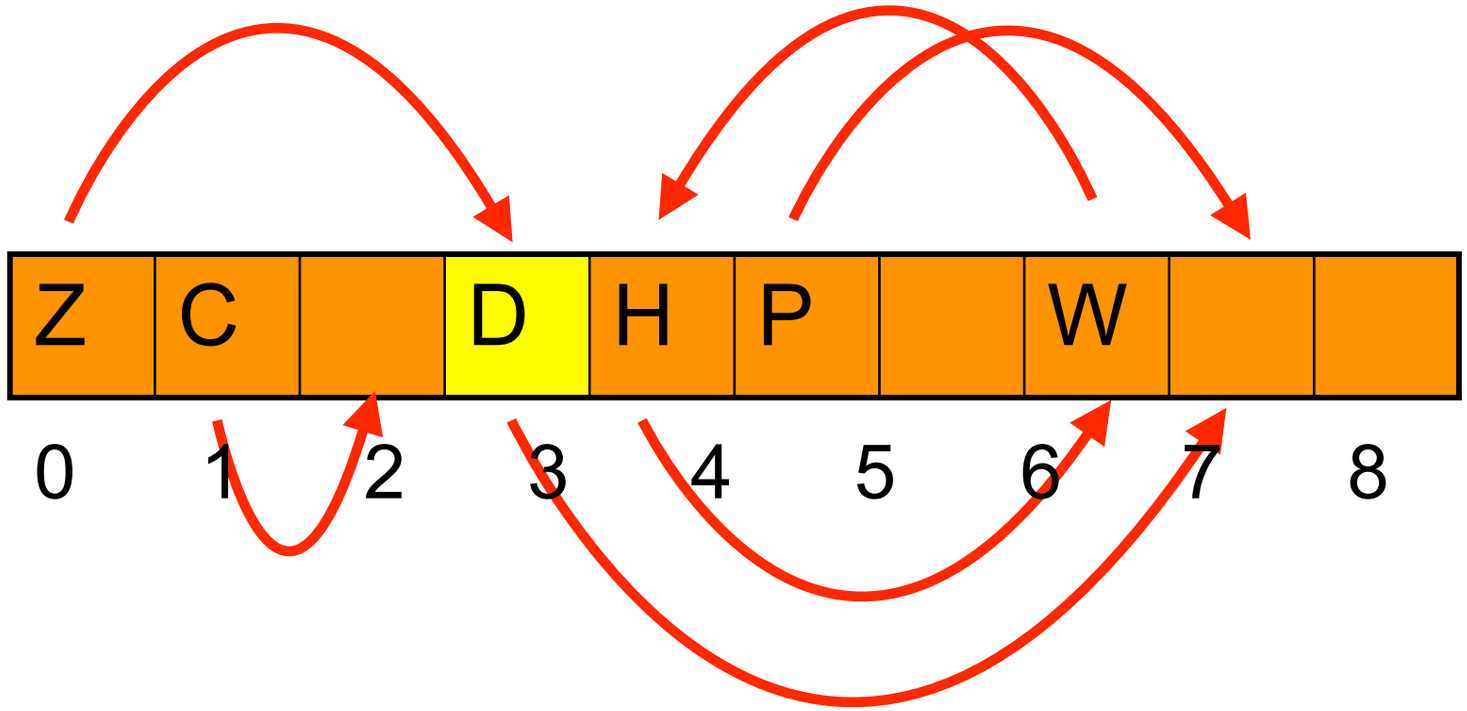
- double hashing variant
- two hash functions h_1(key) and h_2(key)
- idea: instead of looking for a free slot, throw out the current occupant
- analogy: cuckoo looking for a nest will throw out other bird‘s kids to make room for its own kids
- the throwing out idea is applied recursively.
- Literature: R. Pagh and F. Rodler, Cuckoo Hashing, Proceedings of European Symposium on Algorithms, 2001

Cuckoo Hashing



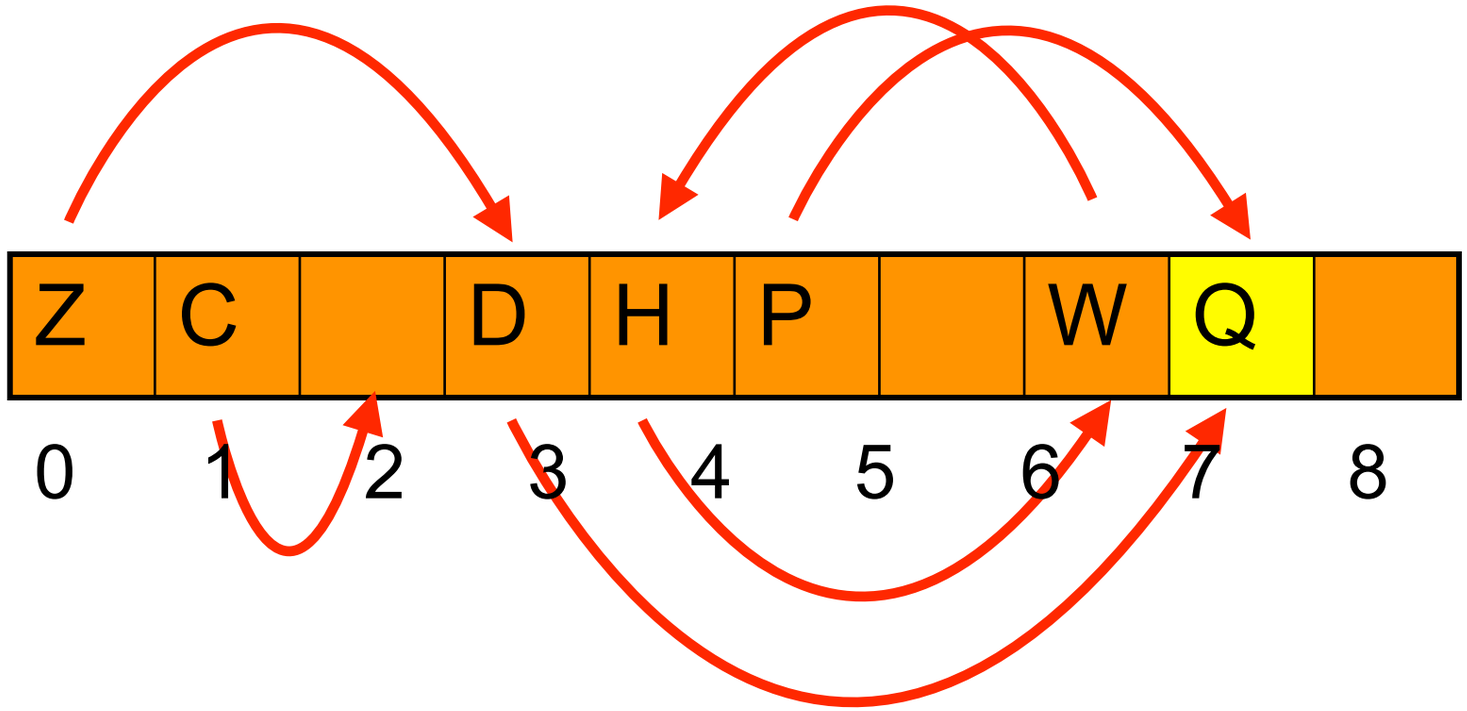
- assume we want to insert data item D
- $h_1(D)=3$
- however slot 3 contains Q => replace Q by D
- but what do we do with Q?

Cuckoo Hashing



- compute $h_1(Q)=3$ => same as current slot (not always!)
- => compute $h_2(Q) = 7$
- slot 7 is free => insert Q

Cuckoo Hashing



- Done.

Cuckoo Insertion Algorithm

```
procedure insert(key){
    if (a[h1(key)] == key or a[h2(key)] == key )
        return;                                          // HT already contains key
    else {
        pos := h1(key);                                  // compute slot returned by h1
        loop n times{                                    // loop required to break cycles
            currentItem = a[pos];                        // handle to entry at slot <pos>
            a[pos] = key;                                // we insert the element anyway
            if (currentItem == null) then                // was the slot unoccupied?
                return;                                  // done.
            else                                         // continue with currentItem
                if (pos == h1(currentItem) ) then        // hash of replaced item points to pos?
                    pos = h2(currentItem)                // use second hash function
                else
                    pos = h1(currentItem)                // use first hash function
        }
        rehash();                                        // rehash if looping n times fails
        insert(key);                                     // then insert key recursively
    }
}
```

Main Memory

- cuckoo hashing has very good space utilization (95-98%)
- however in practice for large hash tables less efficient for probes than traditional techniques
- requires additional improvements to make it fast
- core ideas:
  - use multiple entries per bucket again
  - combine with SIMD instructions
- Literature:
  - M. Zukowski et al. Architecture-conscious hashing. In DAMON 2006
  - Kenneth A. Ross: Efficient Hash Probes on Modern Processors. ICDE 2007: 1297-1301

How to Fix Collisions? Rehash!

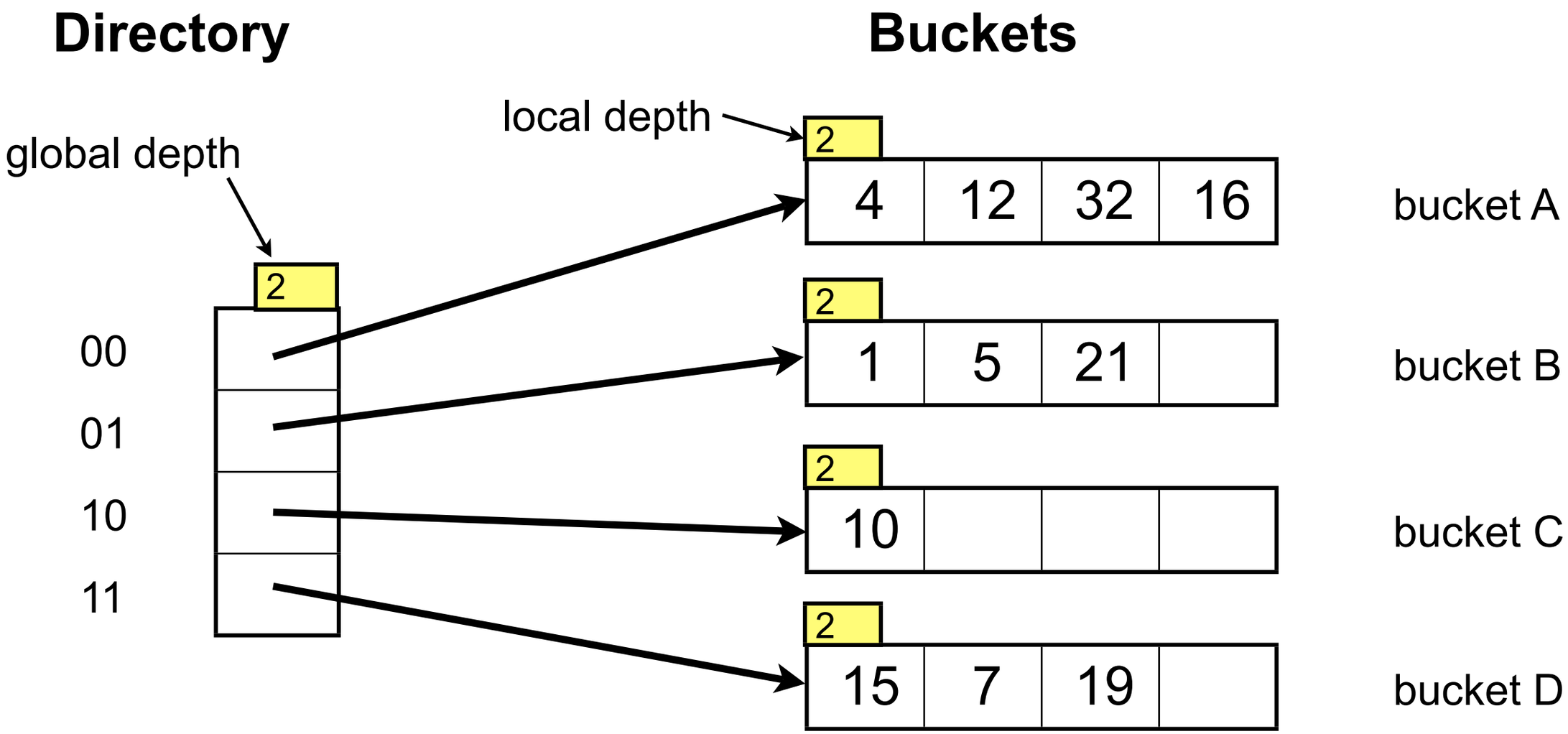
- still probing variants may not be efficient (or even successful) if hash table becomes too occupied
- then it may pay off to rehash
- rehash
  - pick an M“ > M‘
  - create new hash table with M“
  - re-insert all entries from old hash table into new hash table
- in other words
  - throw old hash table away
  - insert everything into new hash table
- may be very too expensive in practice (though linear)
- however sometimes the only choice
- other options: split hash table dynamically...

Dynamic Hashing.

Extendible Hashing

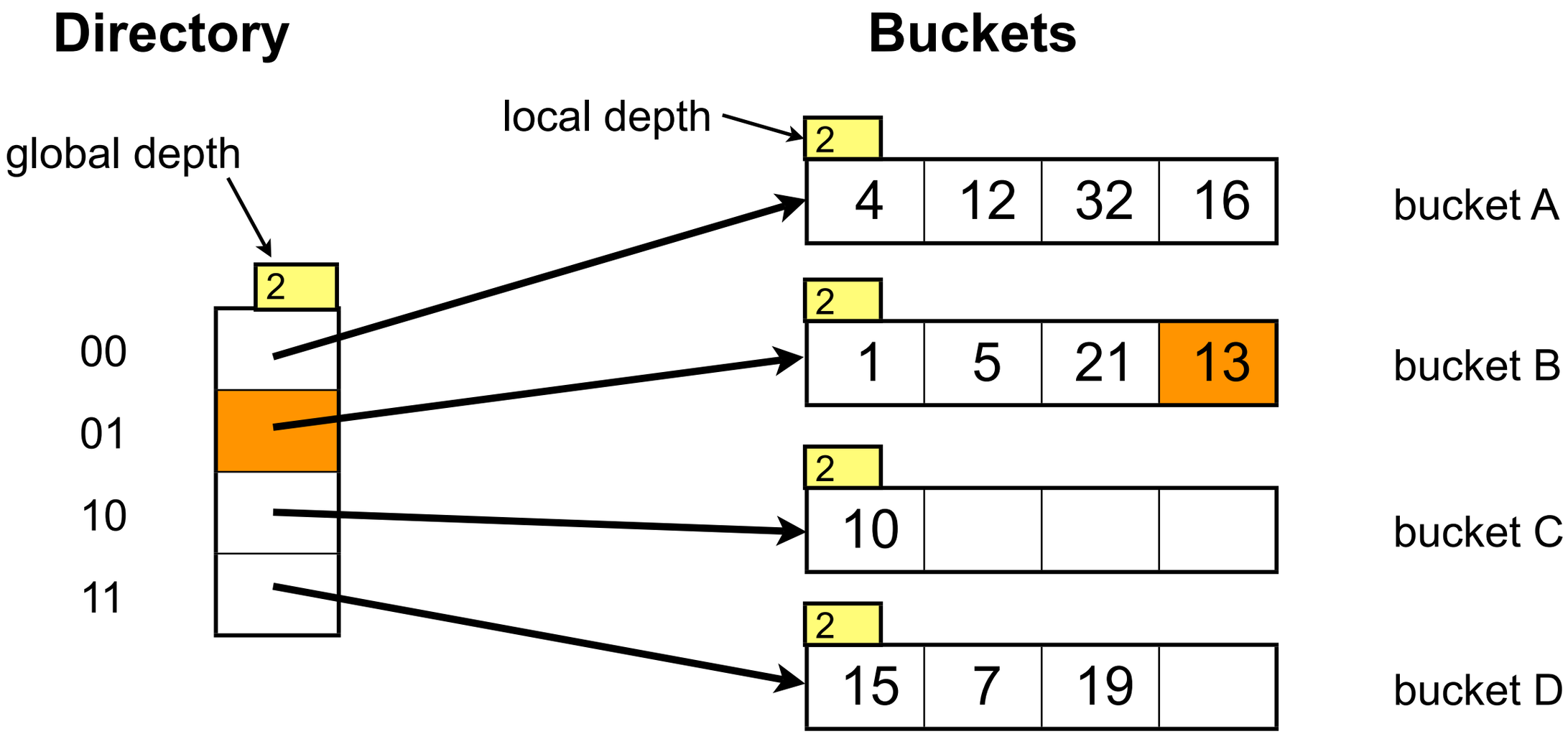
- problem: bucket becomes full
- why not reorganize by doubling number of buckets?
- idea: use directory of pointers to buckets
- doubling number of buckets then means:
  - double the directory => cheap
  - only double bucket that overflows => also cheap
  - directory much smaller than buckets => we do not pay much
  - trick lies in how hash function is adjusted

Extendible Hashing Example



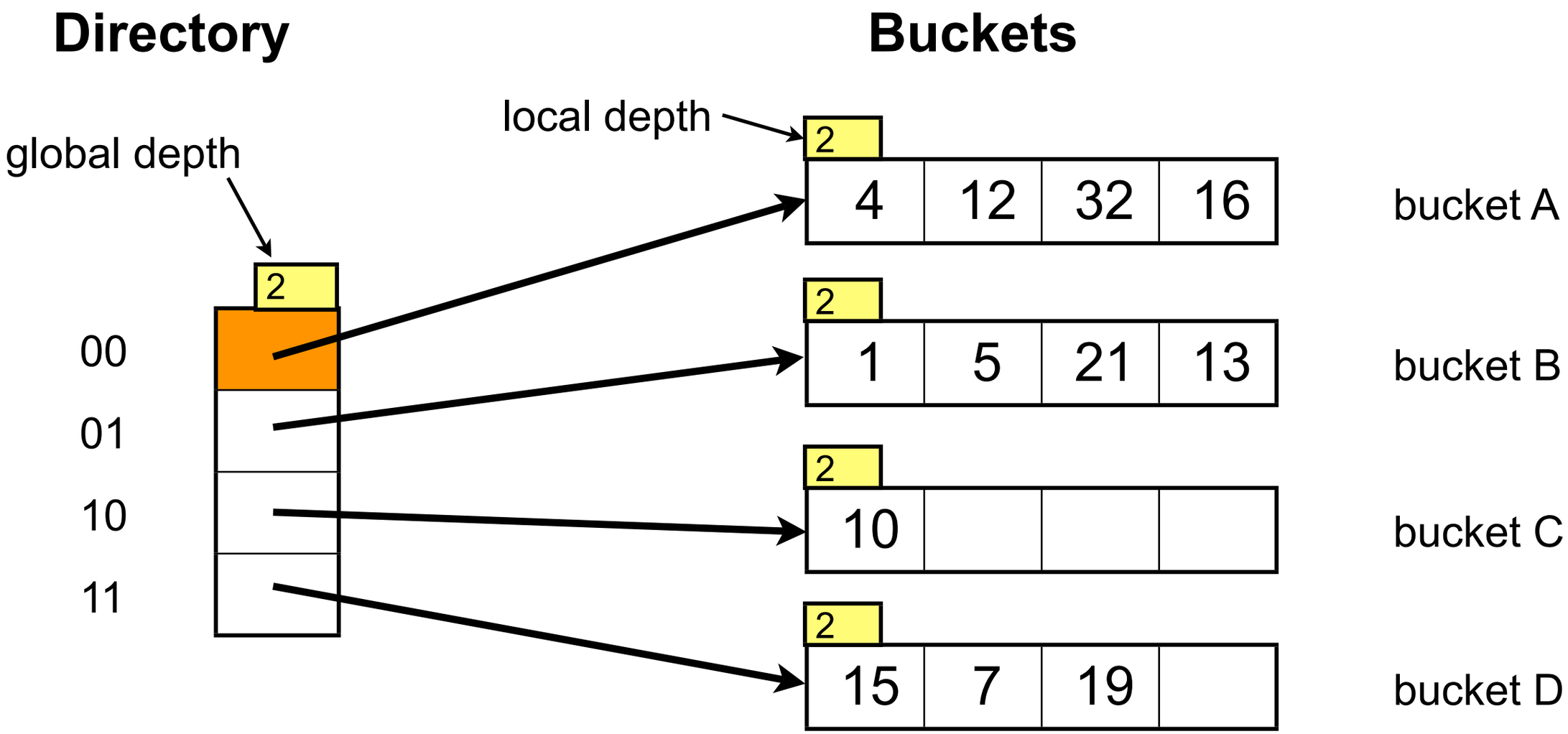
- any hash function, however we take only the last two bits of the hash to determine bucket
- here each bucket may hold 4 data entries

Insert 13



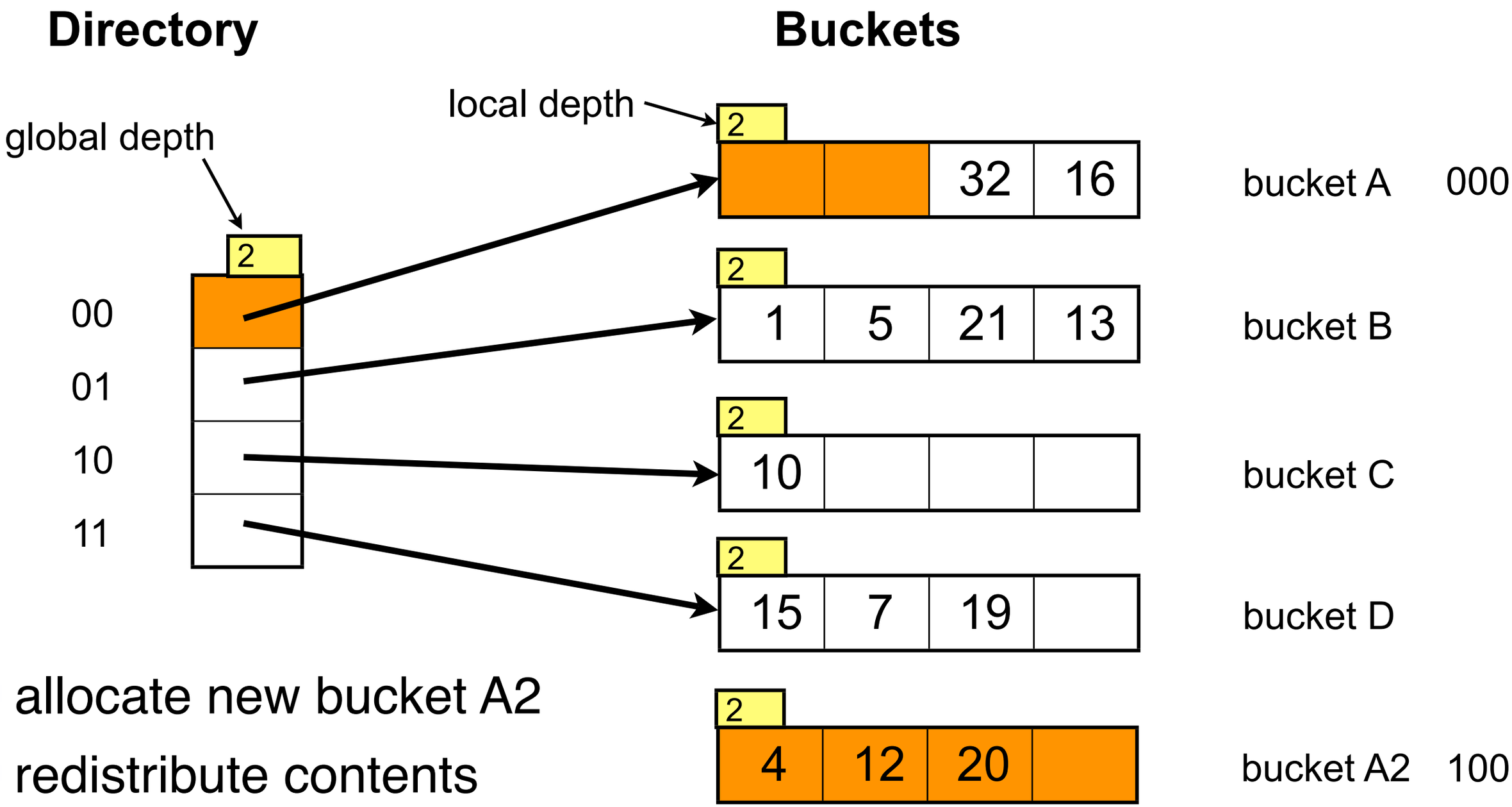
- $13 = 1101_2$ => suffix 01_2 => use bucket B
- bucket B has a free slot => insert => done.

Insert 20



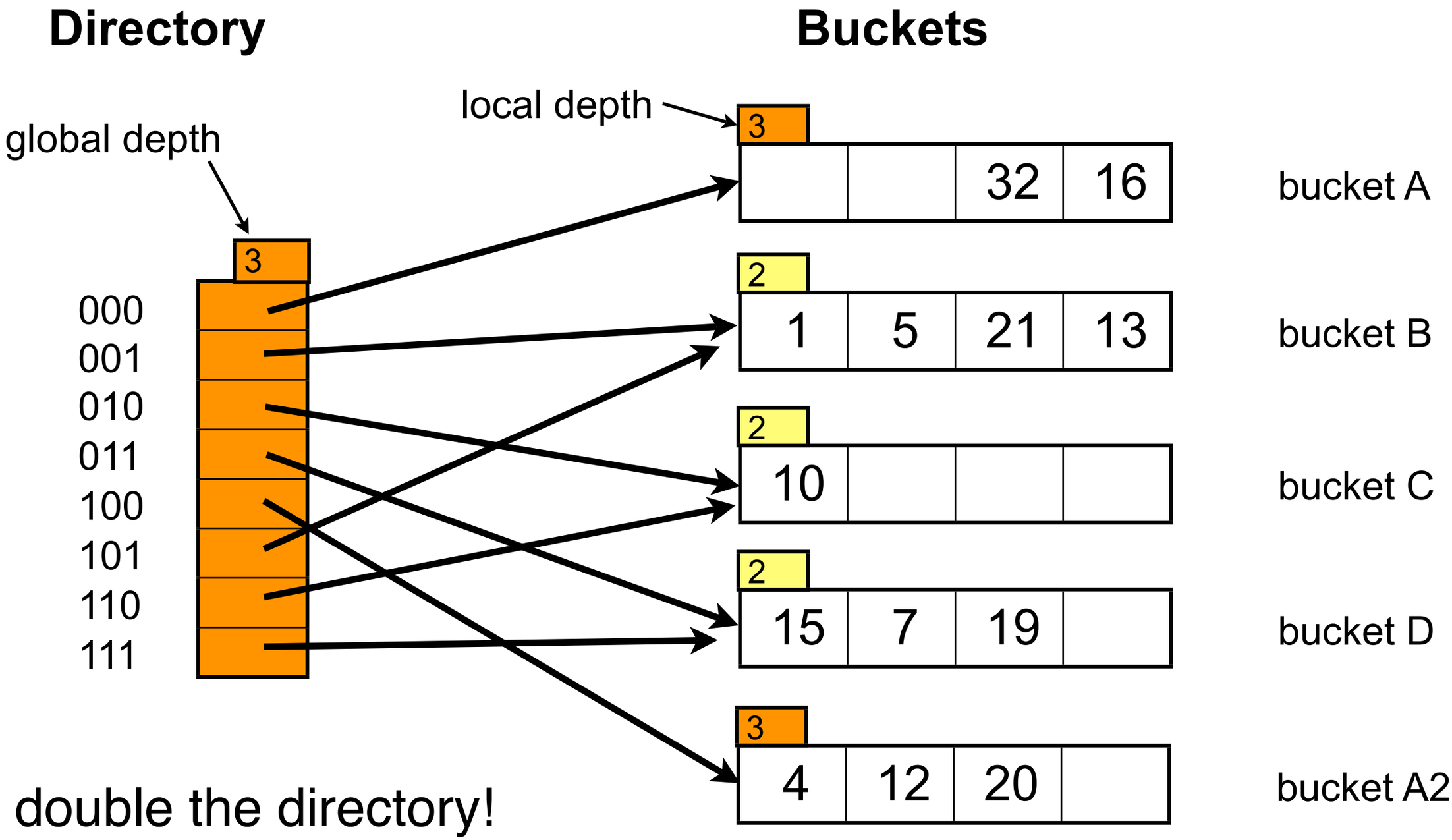
- 20 = 10100_2 => suffix 00_2 => use bucket A
- bucket A is already full => we first need to split the bucket!

Splitting Bucket A (Step 1)



- allocate new bucket A2
- redistribute contents
  => consider last **three** bits

Splitting Bucket A (Step 2)

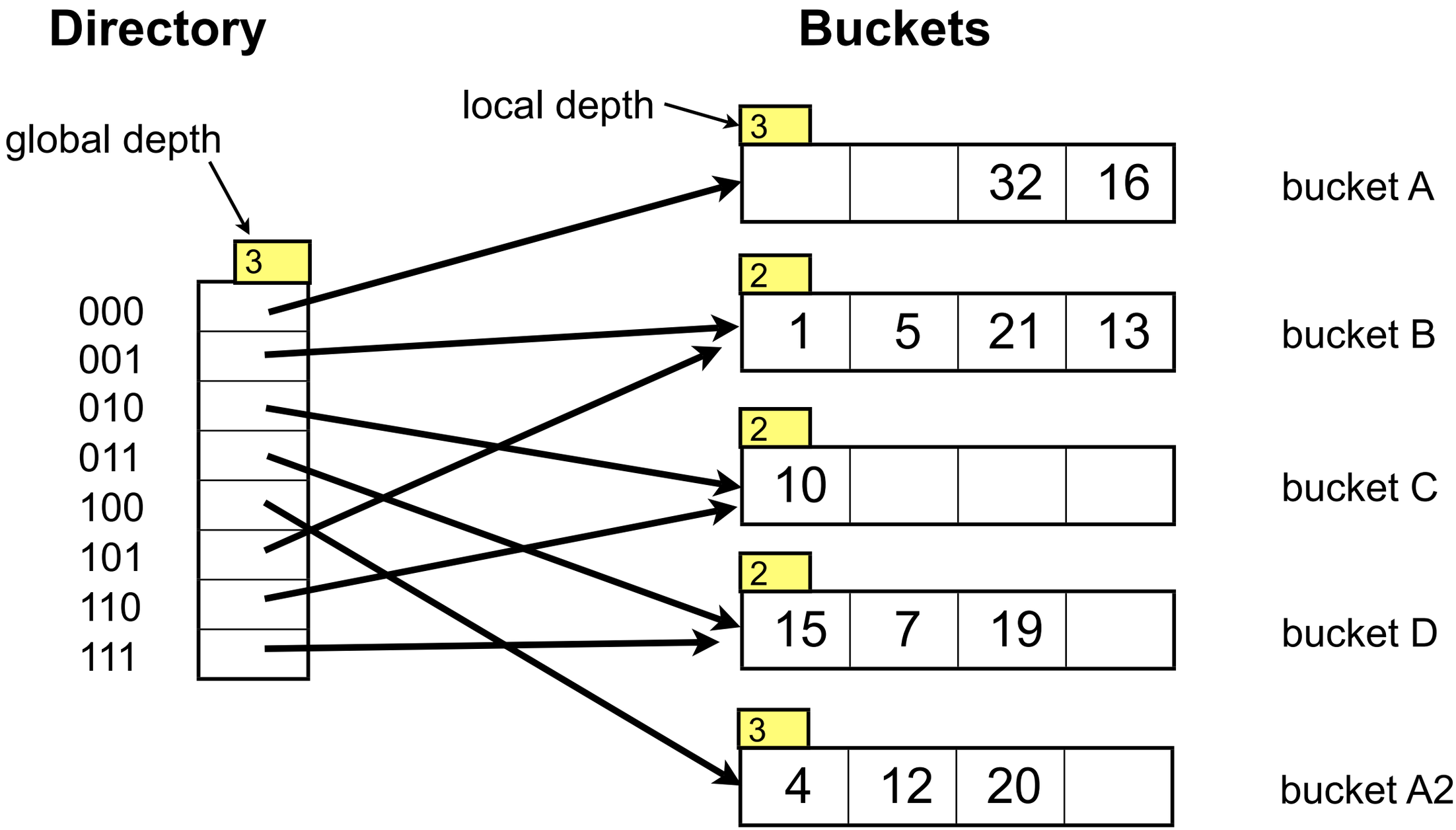


- double the directory!
- global depth := 3 => always use three bits from hash

When to Double the Directory?

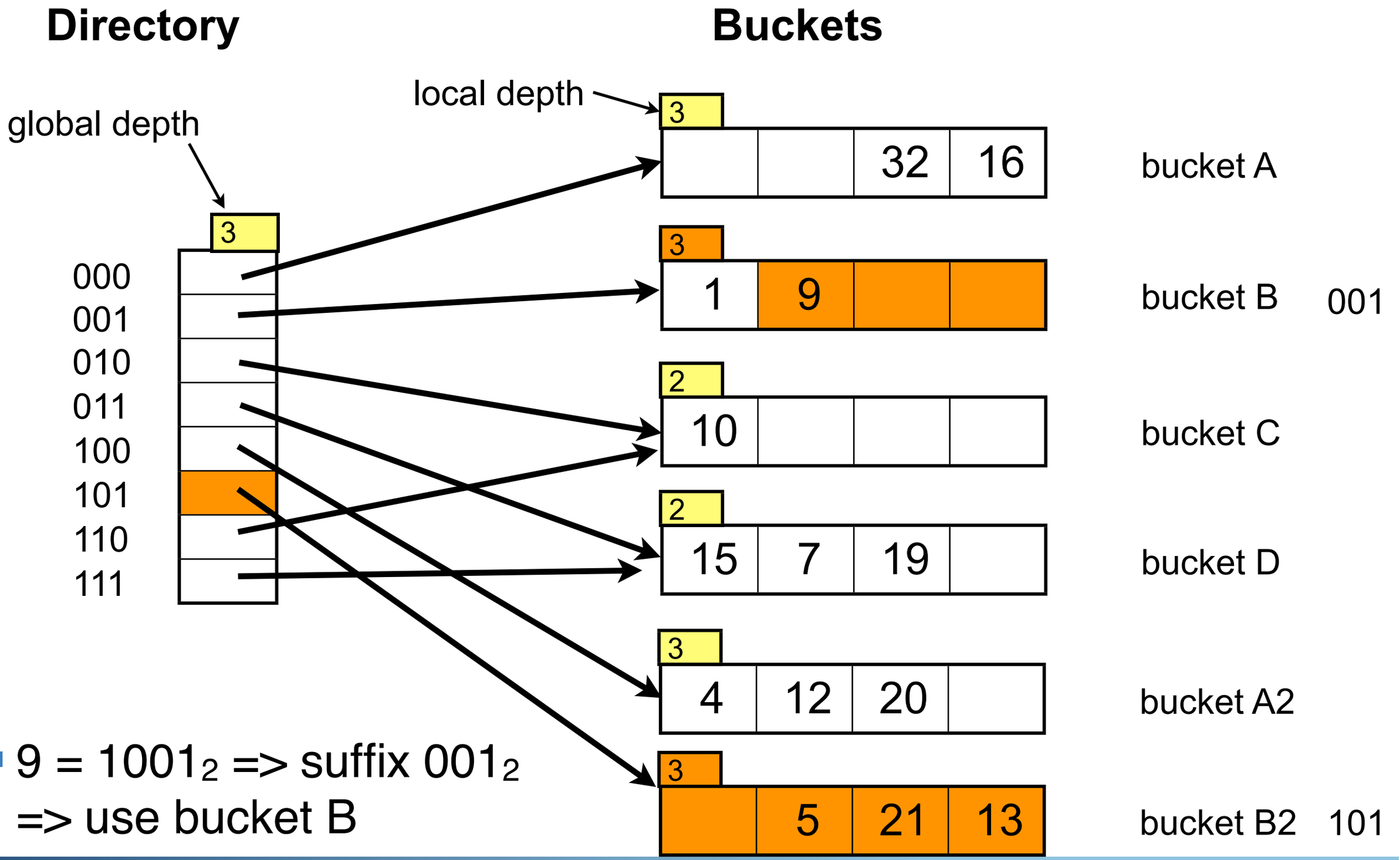
- double directory if
  - bucket determined by current directory is full
  - and local depth of that bucket == global depth
- otherwise we simply double the bucket and redirect one pointer in the directory
- for example let‘s insert 9 = 1001_2

Insert 9 (Step 1)



- $9 = 1001_2$ => suffix 001_2
  => use bucket B

Insert 9 (Step 2)



- $9 = 1001_2$ => suffix 001_2
  => use bucket B

Discussion

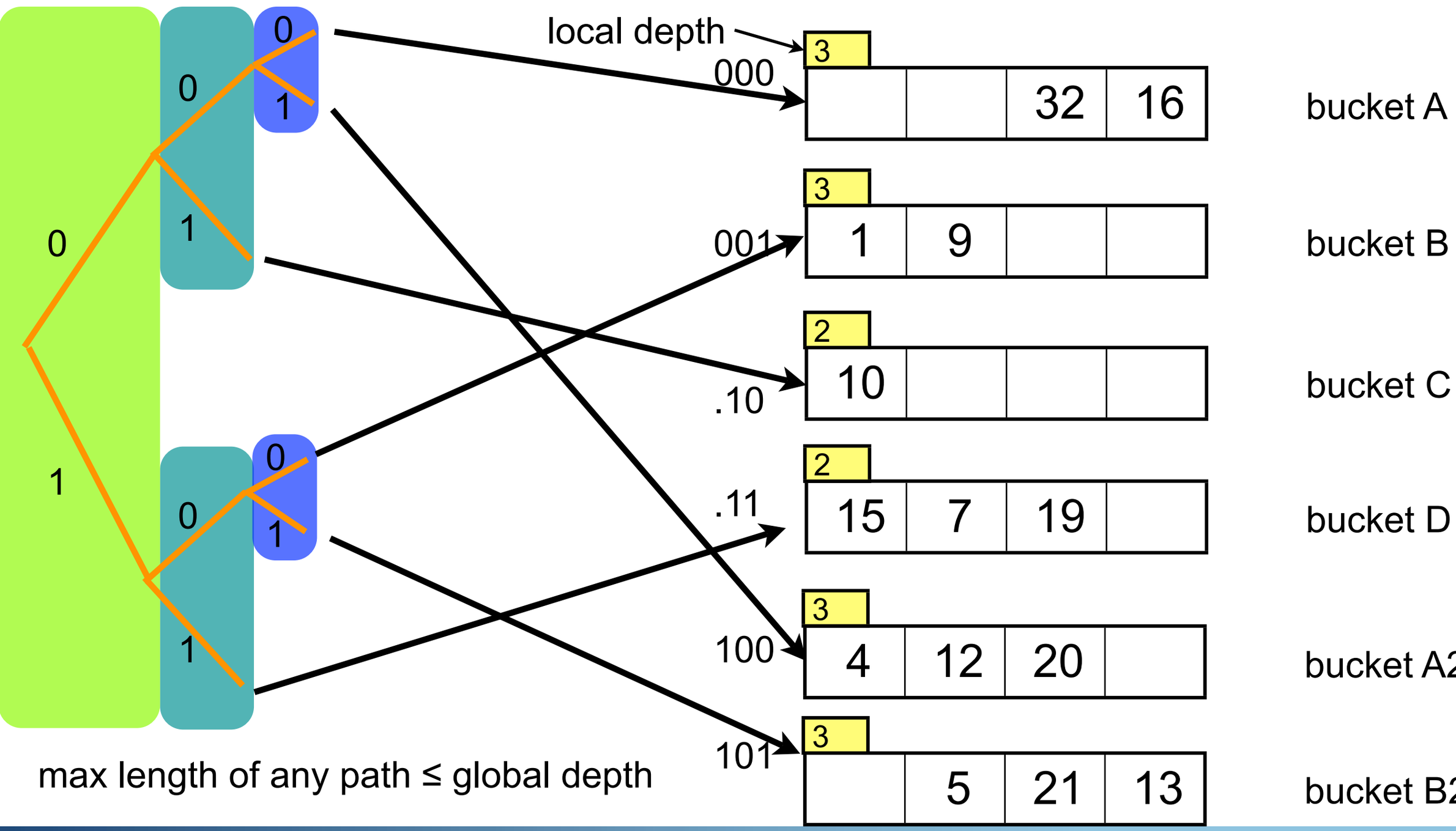
- bucket split leads to increase in local depth
- only if local depth greater than global depth => double directory => increase global depth
- deletes: could also merge buckets to decrease local depth
- could even decrease global depth
- however often omitted in practice
- entire approach sensitive to skewed distributions
- if distribution of **hash values of keys** (not distribution of keys themselves) is skewed,
- => directory may become large

Discuss the difference!

Another View on Extendible Hashing

- actually the directory may be considered a linear representation of a digital tree!!
- **logically** the directory corresponds to a digital tree
- **physically** the directory uses a linearized digital trie of granularity “global depth“
- therefore: extendible hashing is a **hybrid** of a tree index and a hash index
- see also excellent discussion in Härder&Rahm book
- we will come back to these analogies when discussing multi-dimensional indexing/z-codes

Tree Representation of Directory



max length of any path ≤ global depth

External Memory Hashing Variants

- keep directory in main memory
- keep buckets (=one page) on external memory
- thus we expect on disk I/O per access (search or insert)
- Example
  - 100MB file
  - 100 bytes per data item => 1M data items
  - page size 4KB
  - => about 40 data items per page/bucket
  - we need one directory entry per bucket
  - => 1M/40 = 25,000 directory entries
  - assume pointer (or offset in file) requires 4 bytes
  - than we need only 100KB of main memory
  - instead of 100MB
  - factor 1,000 better

Linear Hashing

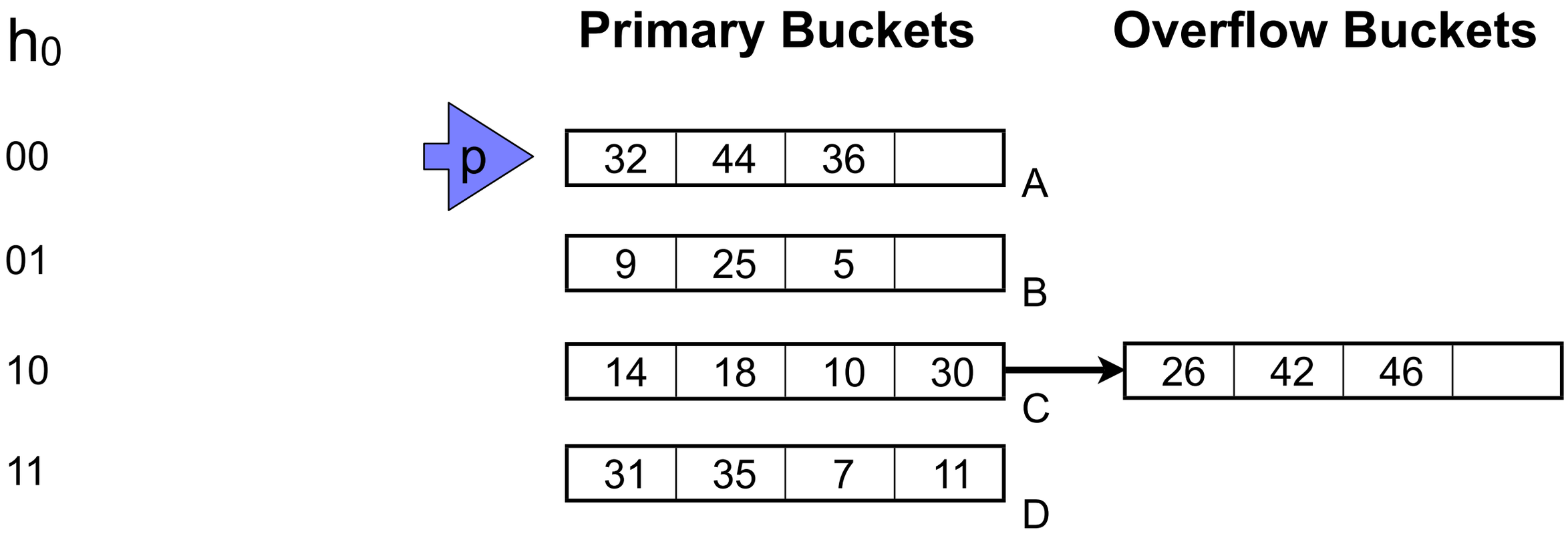
- another dynamic hashing variant
- similar to extendible hashing, however: **no** directory
- uses family of hash functions h_0, h_1, h_2, ...
- $h_{Level}(key) = h(key) \bmod (2^i N)$, N = initial number of buckets
- h is some hash function, range is **not** 0, ..., N-1
- $h_{Level+1}$ doubles the range of h_{Level}
- this is similar to directory doubling
- buckets may use overflow chains

Core Algorithmic Idea

- start with pointer p=0 pointing to bucket 0
- monitor global occupancy δ of hash index
- whenever occupancy δ > 80%
  - perform next split of bucket p
  - append new bucket to end of bucket list
  - redistribute entries of old bucket to new bucket
  - increase p by one
- first round ends if all initial buckets 0,..., N-1 have been split
- this corresponds to a directory doubling
- however: in contrast to extendible hashing, directory does not double in one step but **gradually**
- if all buckets have been doubled => next round

Linear Hashing Example

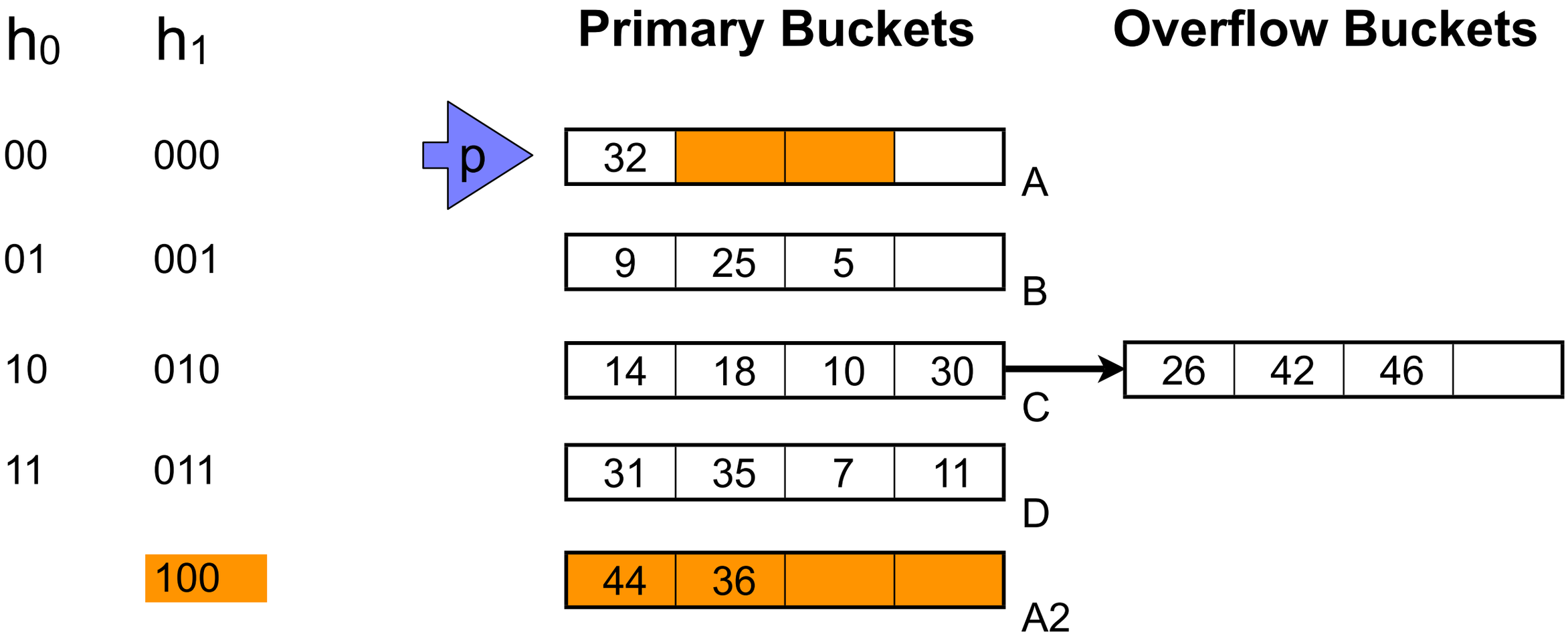
Level=0, N=4



- 4 buckets, Level=0
- start of first round
- p points to bucket A
- occupancy condition triggers a split

Linear Hashing Example

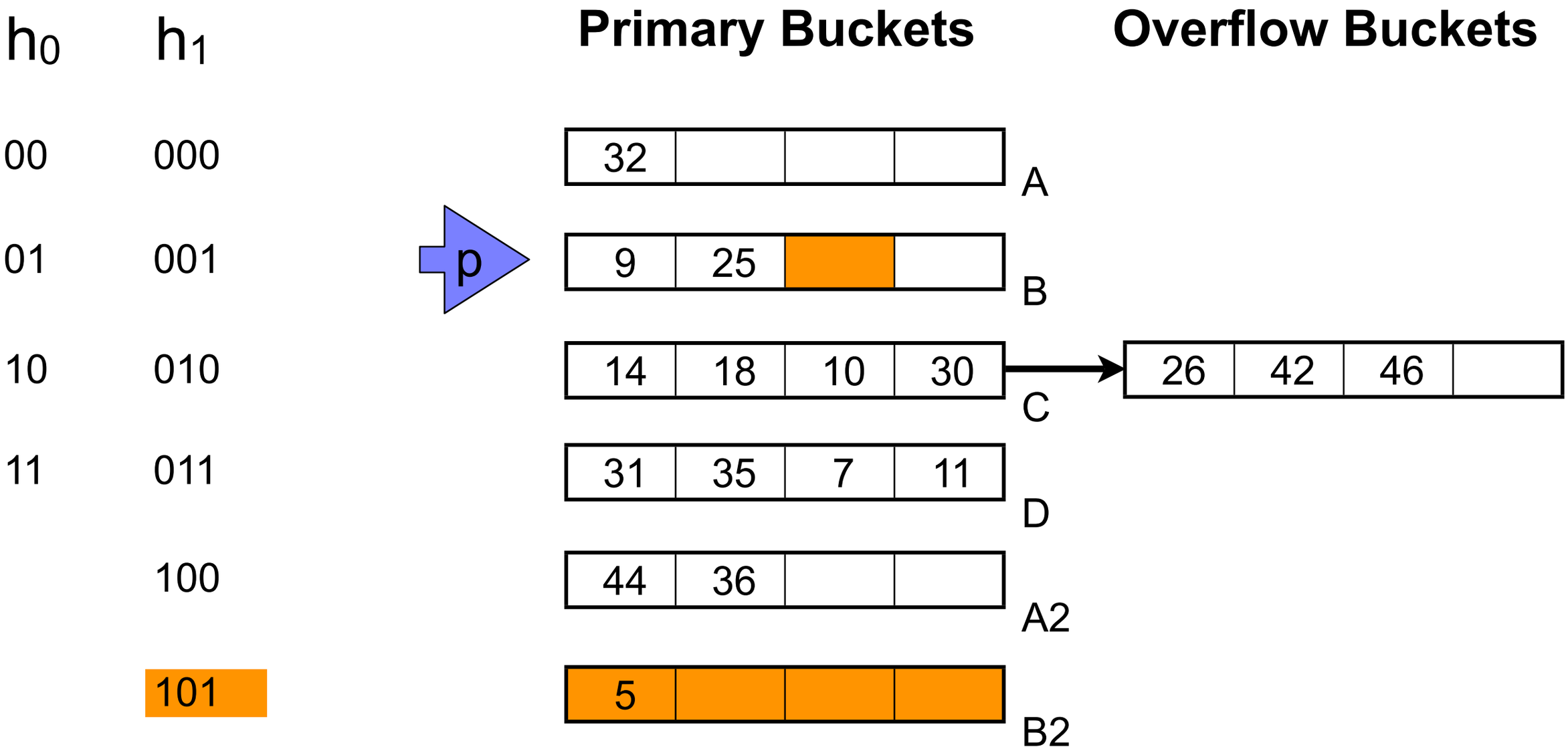
Level=0, N=4



- bucket A is split
- contents are redistributed to buckets A and A2
- 32 stays in bucket A; 44 and 36 move to bucket A2
- redistribution based on h_1!

Linear Hashing Example

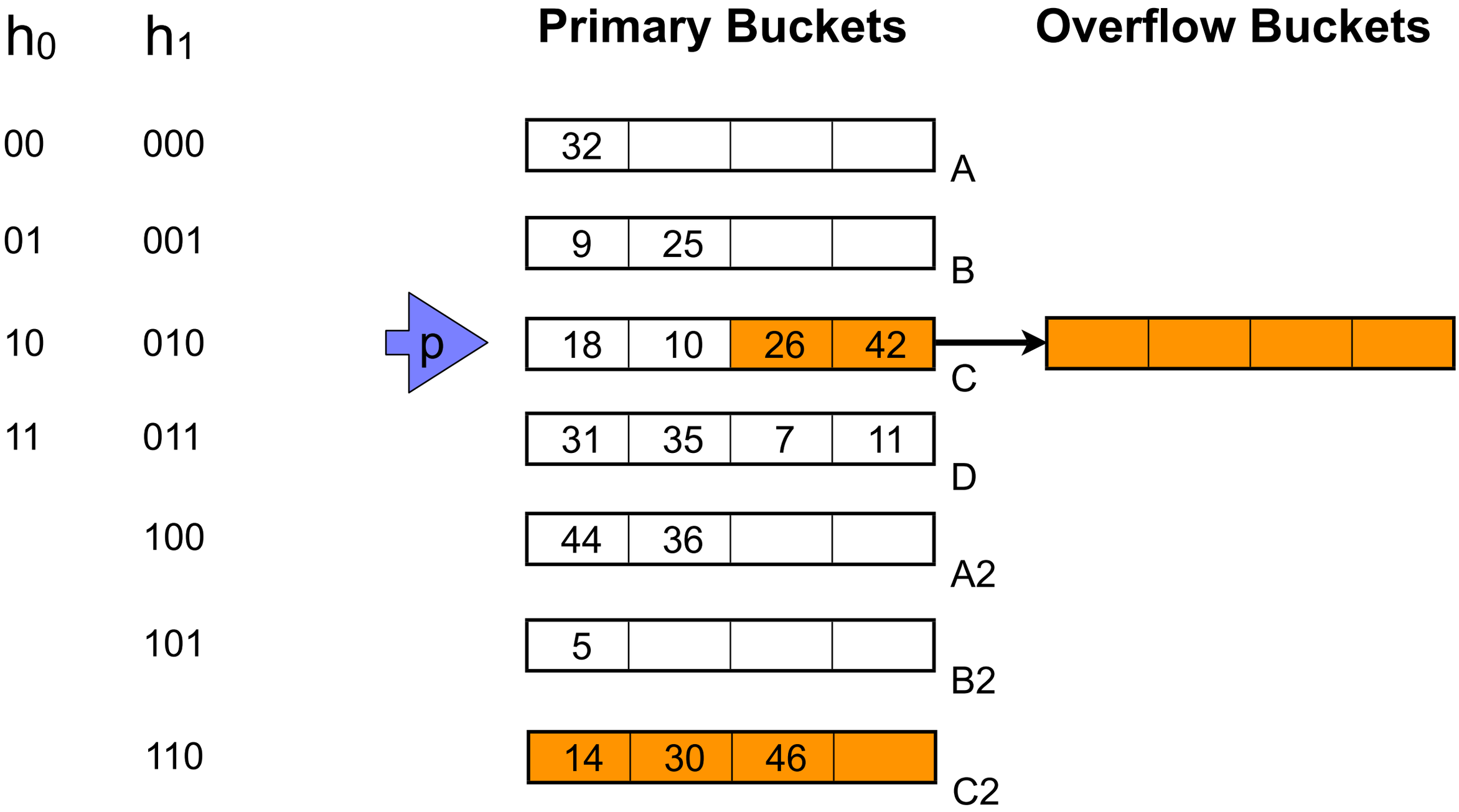
Level=0, N=4



- next split: bucket B is split
- contents are redistributed to buckets B and B2
- redistribution based on h_1!

Linear Hashing Example

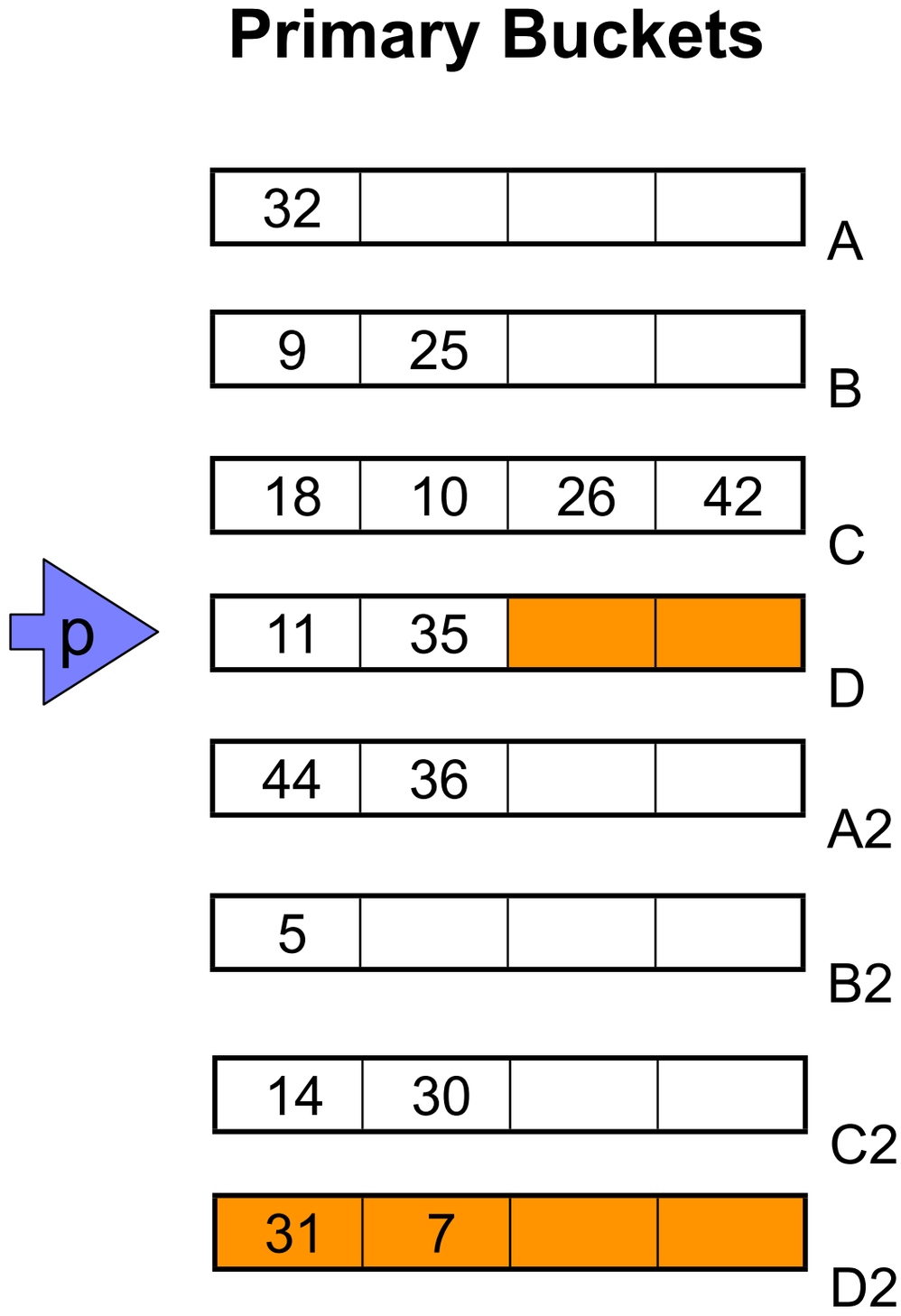
Level=0, N=4



End of First Round

Level=0, N=4

h_0	h_1
00	000
01	001
10	010
11	011
	100
	101
	110
	111



Find(key)

- Assume Level=0
- if $h_0(key) \geq p$ => $h_0(key)$ computes the correct bucket
- Why? buckets $h_0(key) \geq p$ have **not** been split yet
- else if $h_0(key) < p$ => use $h_1(key)$ to compute the correct bucket
- Why? buckets $h_0(key) < p$ **have been split already** in the current round!
- in general:
  - h := $h_{Level}(key)$
  - if (h < p) THEN h := $h_{Level+1}(key)$

Bitmap Indexes

- nice relationship here
- bucket = one bit
- array representation without hash -> uncompressed bitmap
- smaller array with hash -> **bloom filter**
- everything else is the same!
- bloom filter also used as a signature
- bloom filter may be shipped to other places
- Example: parallel join processing