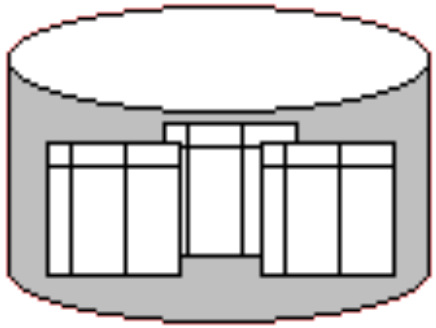
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Φροντιστήριο 4: Tutorial on Indexing part 2 - Hash-based Indexing

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

TUTORIAL ON INDEXING PART 2:
HASH-BASED INDEXING



Simple Hashing Algorithm

- Hash function: a function h(K) transforms a key K into an address
- Example: Take first two characters of the name, multiply their ASCII representations
- Difference between hashing/indexing:
  - Addresses generated appear to be random
  - Two different keys may be transformed to the same address
- Ideal: Should spread out records as uniformly as possible over the range of addresses
- Does the previous sample ensure this? Let's improve it
- Step 1: Represent the key in numerical form

L	O	W	E	L	L	<		BLANKS			>
76	79	87	69	76	76	32	32	32	32	32	32

  - Using more parts of a key – key differences cause hash value differences
  - Extra processing time: significant?

Simple Hashing Algorithm

Step 2 : Fold & Add

- Chop off with two ASCII numbers each:
  76 | 79 | 87 | 69 | 76 | 76 | 32 | 32 | 32

Need to care: overflow. Define a ceiling value

Decide a prime number < 32767. Eg: 19937

7679	+	8769	→	16448	→	16448 mod 19 937
16448	+	7676	→	24124	→	24124 mod 19 937
4187	+	3232	→	7419	→	7419 mod 19 937
7419	+	3232	→	10651	→	10651 mod 19 937
10651	+	3232	→	13883	→	13883 mod 19 937

Division by a prime number usually produces more random distribution

Step 3: Divide by the Size of the Address Space:

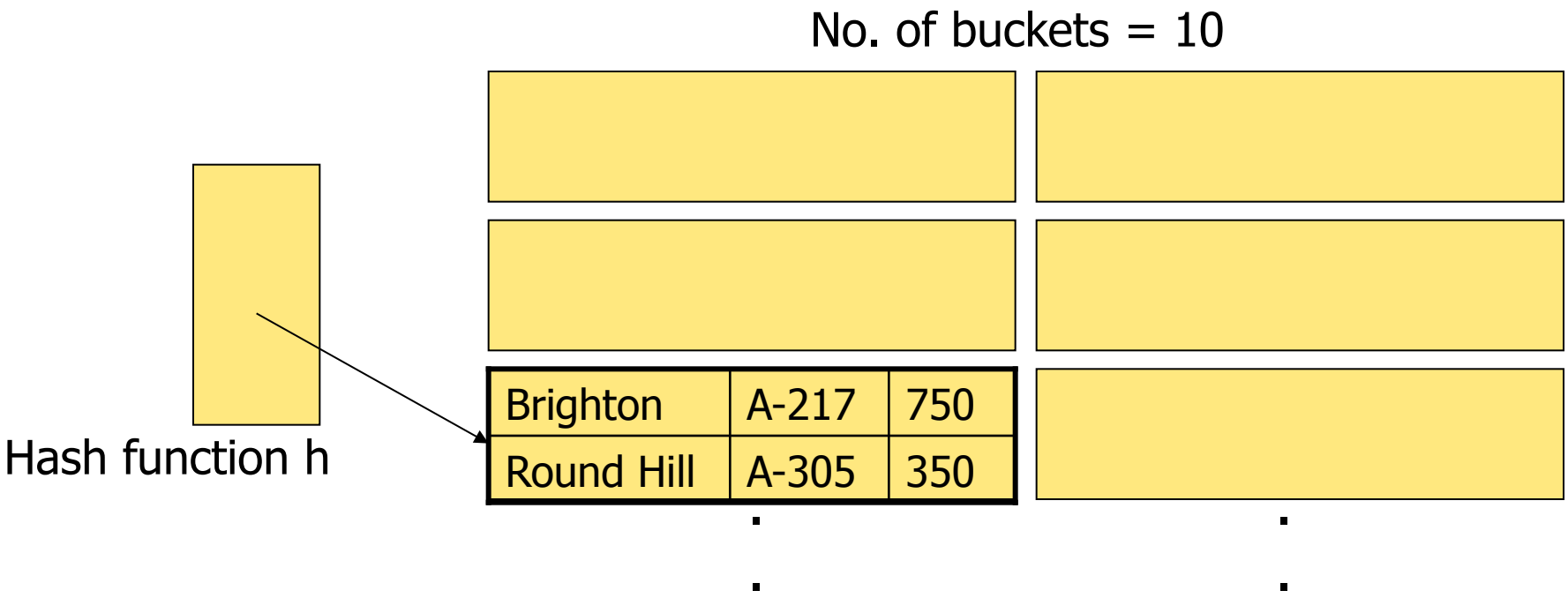
- Shrink the size of the number within the range of address records
- a = s mod n
- s = 13883 , File size 100
- a will be 83. Advice : Decide file size as 101. Why?

Hashing

- Buckets:
  - Set up an area to keep the records: Primary area
  - Divide primary area into buckets
  - Issue: Non-uniform distribution of the records
  - More than one key may give same hash value: Collision
- Separate Chaining:
  - New record with the same hash value has been added, and the corresponding bucket is full. How to solve?
  - Add the address of a new bucket in a special area called overflow area
  - Keep some overflow area in each cylinder

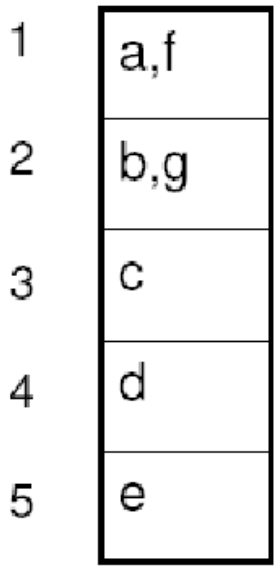
Static Hashing

- Static Hashing
  - A hash function h maps a search-key value K to an address of a bucket
  - Commonly used hash function: K mod n_B where n_B is the # of buckets
  - E.g. h(Brighton) = (2+18+9+7+8+20+15+14) mod 10 = 93 mod 10 = 3

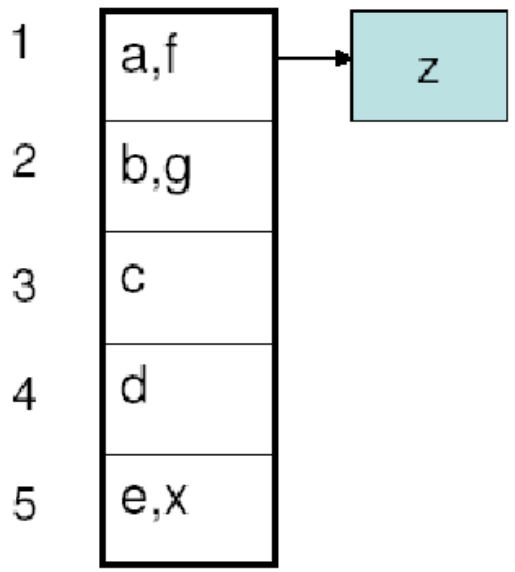


Static Hashing

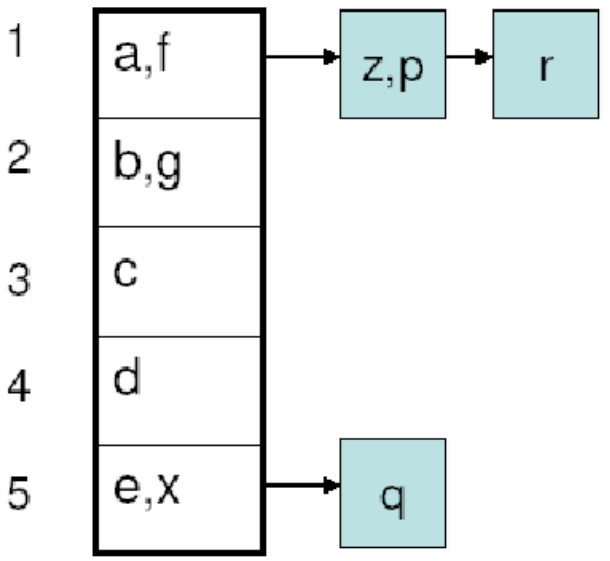
- Assume 2 data entries per bucket and we have 5 buckets
- Insert key values a,b,c,d,e,f,g where h(a)=1, h(b)=2, h(c)=3, …h(g)=2



Insert z,x where
h(z)=1 and h(x)=5



Insert p,q,r where h(p)=1,
h(q)=5 and h(r)=1



Static Hashing

- Clustered Hash Index :
  - The index structure and its buckets are represented as a file (say file.hash)
  - The relation is stored in file.hash (i.e., each entry in file.hash corresponds to a record in the relation)
  - Assuming no duplicates: the record can be accessed in 1 I/O
- Non-clustered Hash Index:
  - The index structure and its buckets are represented as a file (say file.hash)
  - The relation remains intact
  - Each entry in file.hash has the following format: (search-key value, RID)
  - Assuming no duplicates: the record can be accessed in 2 I/O

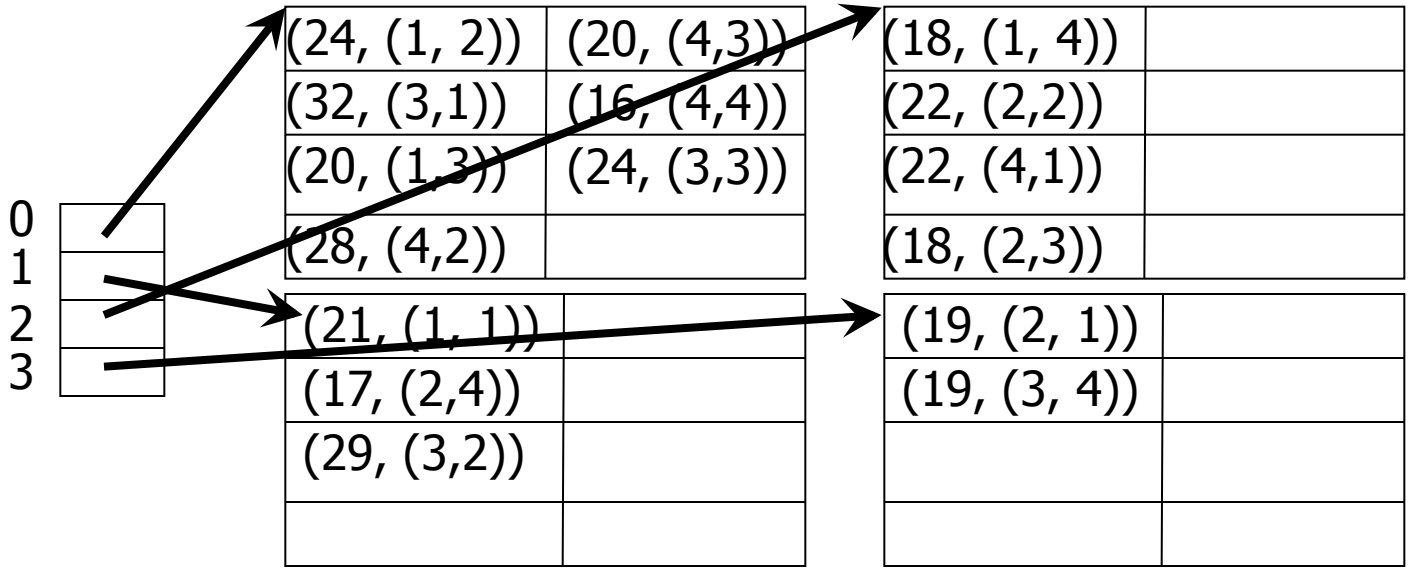
Static Hashing Example

- Assume a student table: `Student(name, age, gpa, major)`
  - tuples(`Student`) = 16
  - pages(`Student`) = 4

Bob, 21, 3.7, CS	Kane, 19, 3.8, ME	Louis, 32, 4, LS	Chris, 22, 3.9, CS
Mary, 24, 3, ECE	Lam, 22, 2.8, ME	Martha, 29, 3.8, CS	Chad, 28, 2.3, LS
Tom, 20, 3.2, EE	Chang, 18, 2.5, CS	James, 24, 3.1, ME	Leila, 20, 3.5, LS
Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE	Pat, 19, 2.8, EE	Shideh, 16, 4, CS

Non-Clustered Hash Index Example

- A non-clustered hash index on the age attribute with 4 buckets,
- h(age) = age mod B



Bob, 21, 3.7, CS	Kane, 19, 3.8, ME	Louis, 32, 4, LS	Chris, 22, 3.9, CS
Mary, 24, 3, ECE	Lam, 22, 2.8, ME	Martha, 29, 3.8, CS	Chad, 28, 2.3, LS
Tom, 20, 3.2, EE	Chang, 18, 2.5, CS	James, 24, 3.1, ME	Leila, 20, 3.5, LS
Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE	Pat, 19, 2.8, EE	Shideh, 16, 4, CS

Non-Clustered Hash Index Example

- A non-clustered hash index on the age attribute with 4 buckets,
- h(age) = age mod B
- Pointers are page-ids

0	500
1	1001
2	706
3	101

1001

(21, (1, 1))	
(17, (2,4))	
(29, (3,2))	

500

(24, (1, 2))	(20, (4,3))
(32, (3,1))	(16, (4,4))
(20, (1,3))	(24, (3,3))
(28, (4,2))	

101

(19, (2, 1))	
(19, (3, 4))	

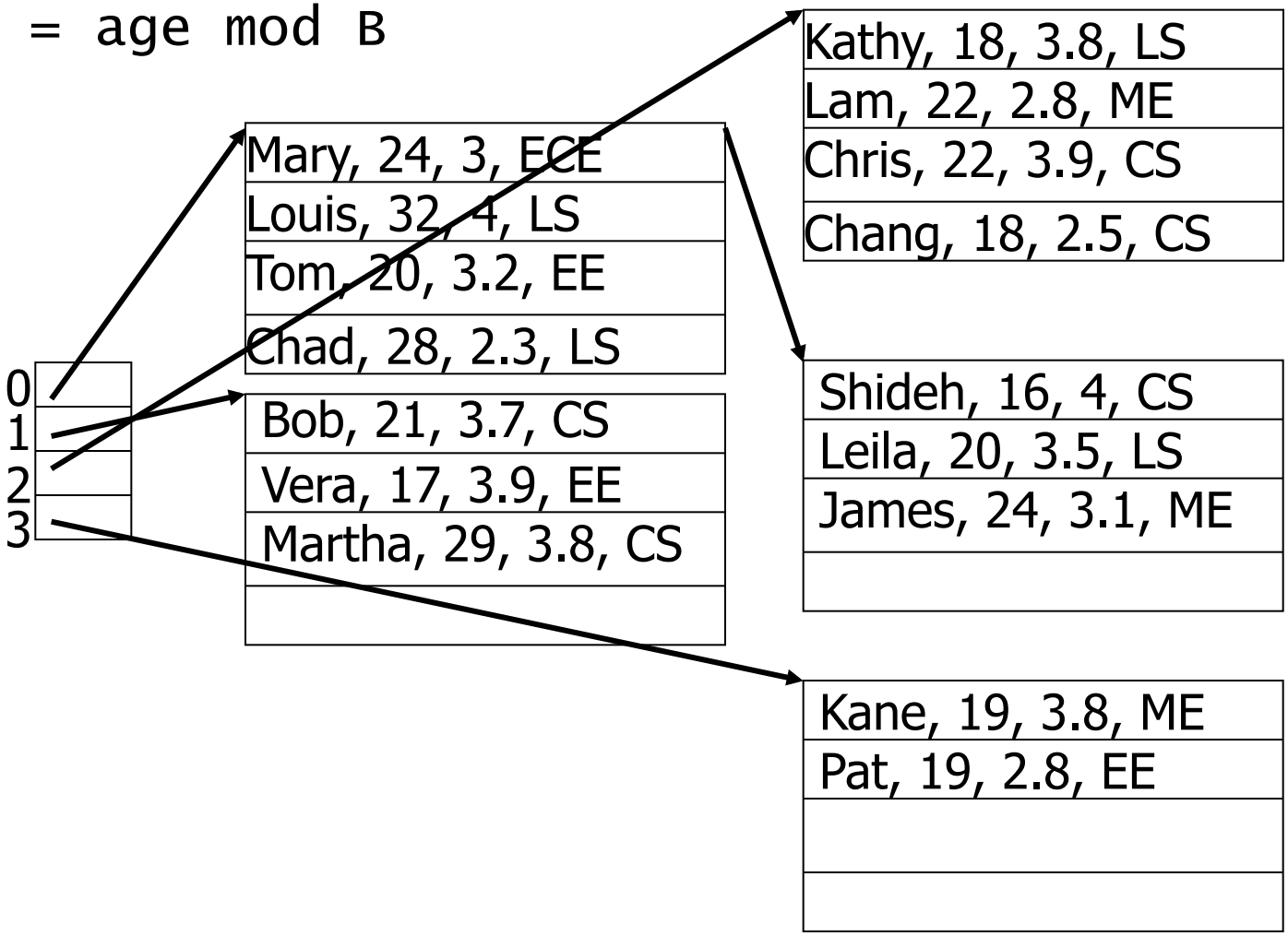
706

(18, (1, 4))	
(22, (2,2))	
(22, (4,1))	
(18, (2,3))	

Bob, 21, 3.7, CS	Kane, 19, 3.8, ME	Louis, 32, 4, LS	Chris, 22, 3.9, CS
Mary, 24, 3, ECE	Lam, 22, 2.8, ME	Martha, 29, 3.8, CS	Chad, 28, 2.3, LS
Tom, 20, 3.2, EE	Chang, 18, 2.5, CS	James, 24, 3.1, ME	Leila, 20, 3.5, LS
Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE	Pat, 19, 2.8, EE	Shideh, 16, 4, CS

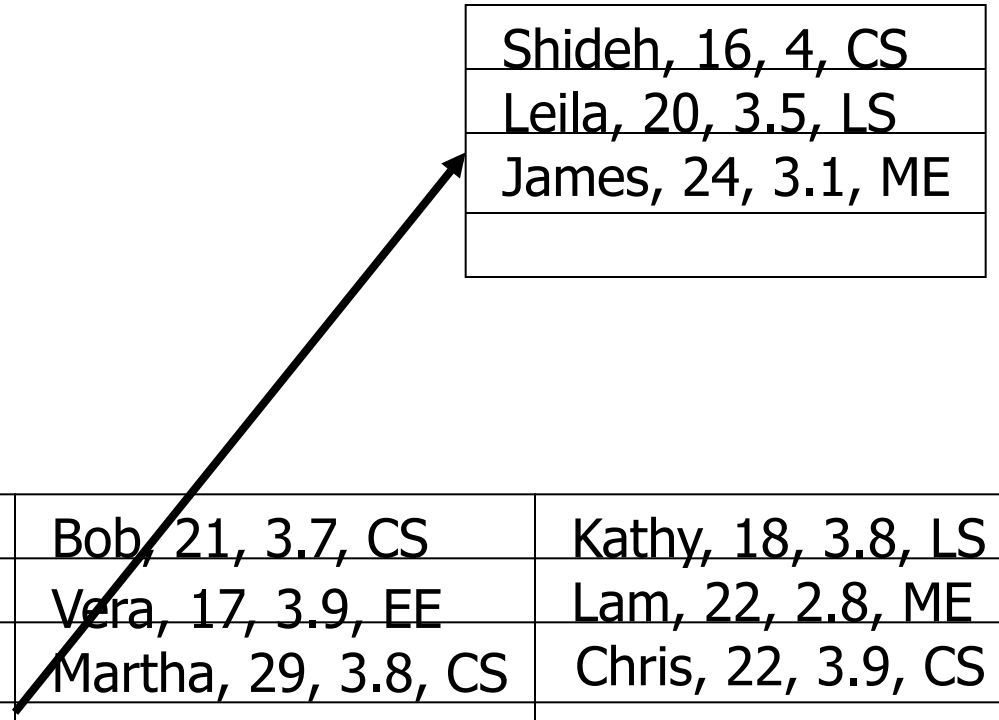
Clustered Hash Index Example

- A clustered hash index on the age attribute with 4 buckets,
- h(age) = age mod B



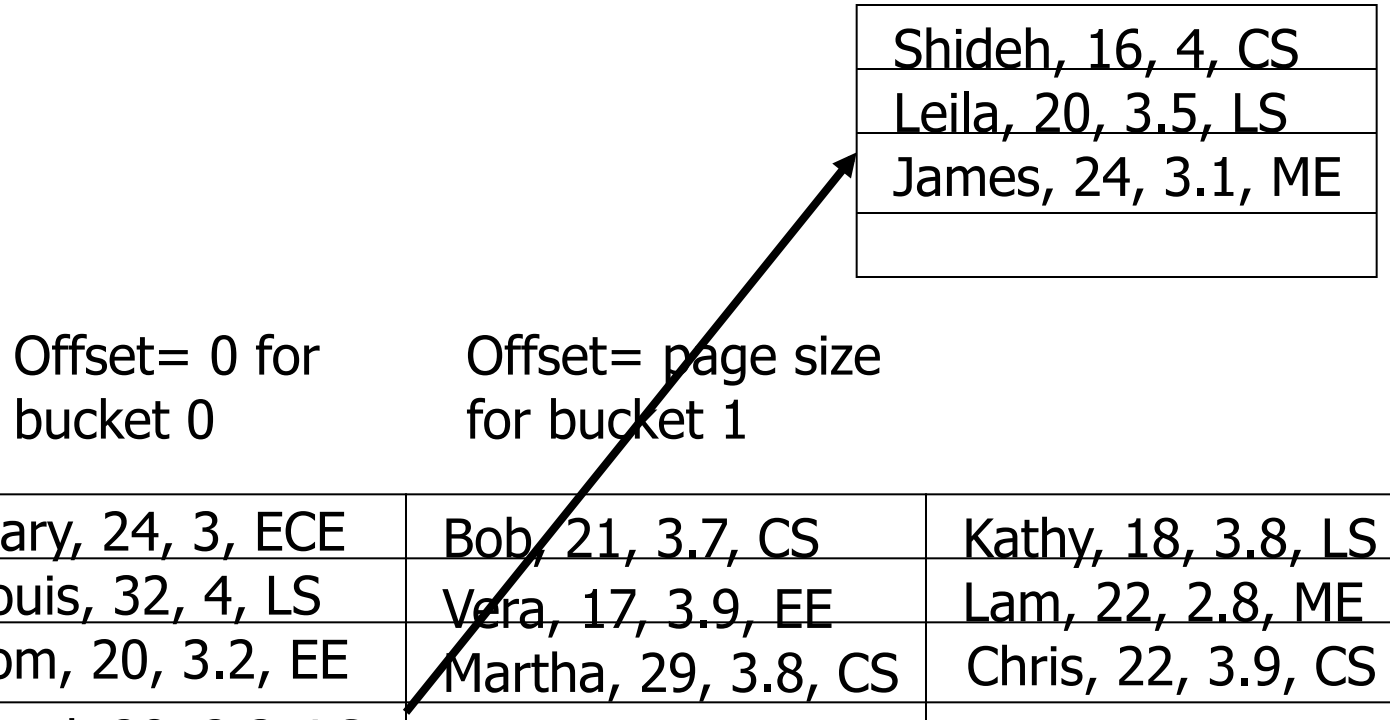
Clustered Hash Index Example: Sequential Layout

- A clustered hash index on the age attribute with 4 buckets,
- `h(age) = age mod 4`
- When the number of buckets is known in advance, the system may assume a sequentially laid file to eliminate the need for the hash directory



Clustered Hash Index Example: Sequential Layout

- A clustered hash index on the age attribute with 4 buckets,
- `h(age) = age mod 4`
- When the number of buckets is known in advance, the system may assume a sequentially laid file to eliminate the need for the hash directory



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B bucket addresses
  - Databases grow with time
    - If initial number of buckets is too small, performance will degrade due to too many overflows
    - If initial number of buckets is set to a high value, anticipating a large file size at some point in the future, significant amount of space will be wasted initially
  - If database shrinks, again space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically

Load Factor

- The Load Factor:
  - Lf: The number of records in the file divided by the number of places for the records in the primary area
  - Primary area should be 70 % to 90 % full
- Lf = R / (B * c)
  - Lf → Load factor
  - R → # of records in the file
  - c → # of records which one bucket can hold
  - B → # of buckets (blocks) in the primary area
- How the choice of Lf & c can affect time Tf for fetching a record?

  Fetching Using Buckets & Chains

  As long as there is no overflow,

  Tf = s + r + dtt

  s → average seek time

  r → average rotational latency

  dtt → time to transfer the data in one bucket

Fetching using Buckets & Chains

- Result: If the distribution of values is even, and primary areas are chosen big enough, hashing is very efficient
  - Contradiction: Larger bucket size also means a larger dtt!
  - Each access to a bucket takes longer
- Handling overflow buckets: Same size as primary buckets or smaller
- To sum up:
  - Performance can be enhanced by the choice of bucket size and of load factor
  - Space utilization: Policy for handling overflow
  - If the file size does not grow so much, hashing can give excellent performance & space utilization

Database Growth

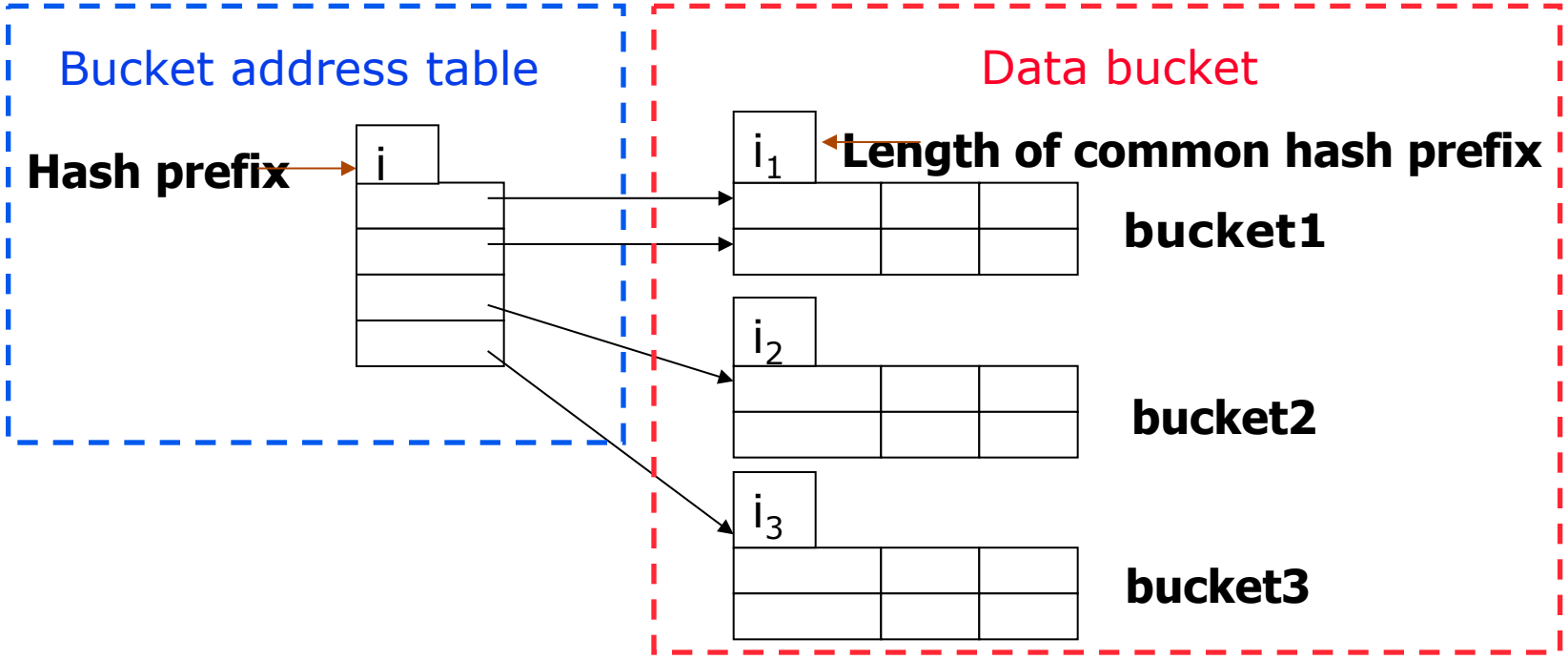
- Consider a database that grows too much. Will buckets suffice?
- x → average overflow chain length
- Tf (successful) = s + r + dtt + (x/2) * (s + r + dtt)
  - One access to get the primary bucket
  - On average, traverse half of the overflow chain
- Tf (unsuccessful) = s + r + dtt + x * (s + r + dtt)
  - One access to get the primary bucket
  - Traverse the whole overflow chain
- Deletion:
  - Merely mark the deleted record; when the whole hash table is reorganized, marked records are not copied to the new table
  - Replace the deleted record with the last record; de-allocate empty buckets
    - Cost : Td = Tf (unsuccessful) + 2r

Database Growth

- Insertion:
  - New insertions are at the end of the chain
  - Assume no new buckets are necessary
    - Ti = Tf (unsuccessful) + 2r (A fetch and a rotation of the disk)
- Sequential Operations:
  - Hashing is useless for sequential operations. Why?
  - Given than n sequential reads are to be processed: Tx = n * Tf
- If the growth of a file can be predicted, hashing with buckets is excellent
  - Range queries? No!
  - Mixture of sequential operations, range searches, and individual searches: Prefer B+ trees
- File grows considerably: Reorganization is necessary
  - Extendible Hashing
  - Linear Hashing
  - No file reorganization is required

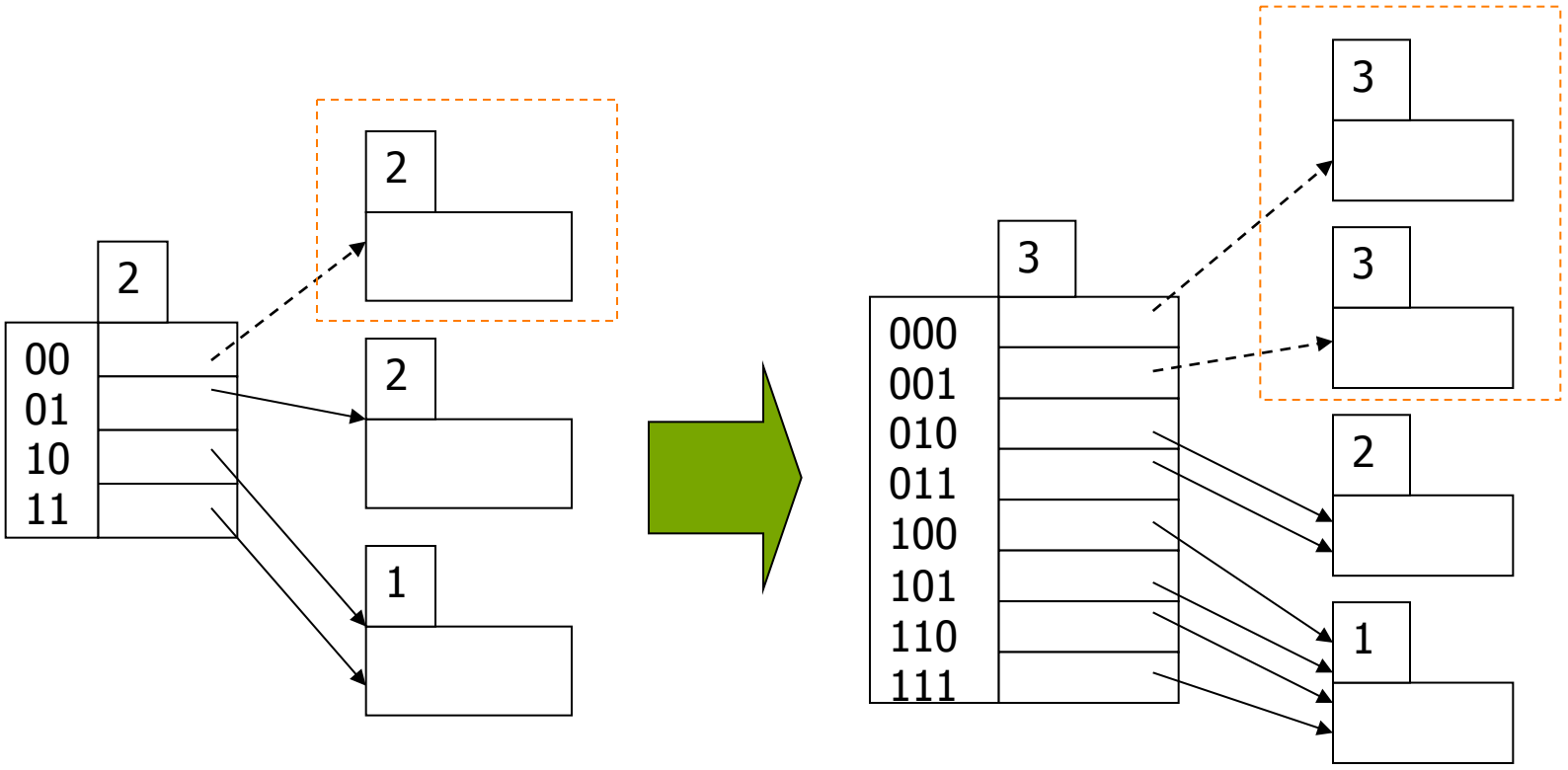
Extendible Hashing (EH)

- Extendible Hashing
  - Hash function returns b bits
  - Only the prefix (or suffix) i bits are used to hash the item
  - There are 2^i entries in the bucket address table
  - Let i_j be the length of the common hash prefix for data bucket j, there are $2^{(i-i_j)}$ entries in the bucket address table that point to j



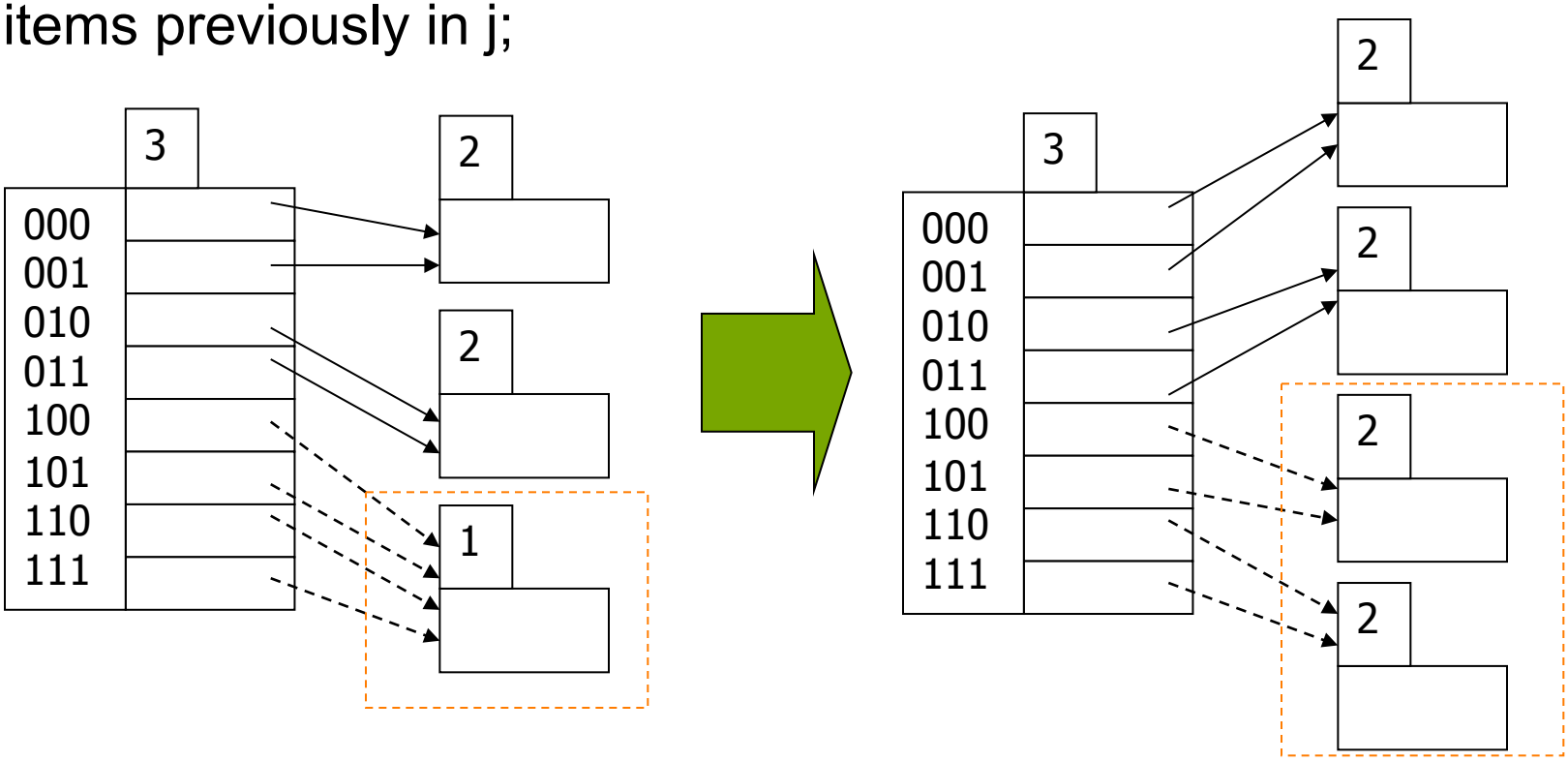
Splitting

- Splitting - Case 1: i_j=i
  - Only one entry in bucket address table points to data bucket j
  - i++; Split data bucket j to j, z; i_j=i_z=i; Rehash all items previously in j;



Splitting (cont.)

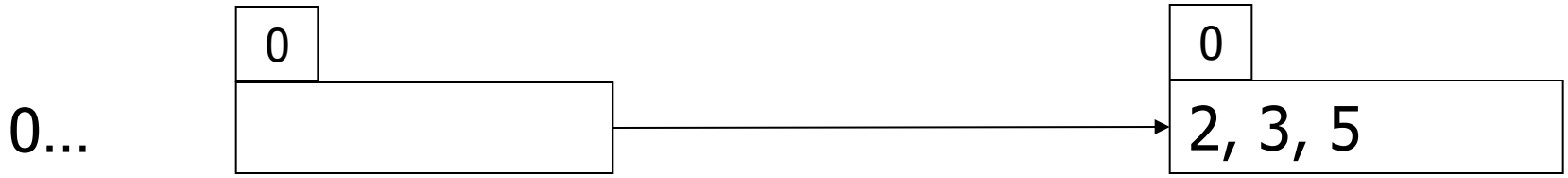
- Splitting - Case 2: $i_j < i$
  - More than one entries in bucket address table point to data bucket j
  - Split data bucket j to j, z; $i_j = i_z = i_j + 1$; Adjust the pointers that previously pointed to j, so that they point to j and z; Rehash all items previously in j;



Extendible Hashing (EH): First Example

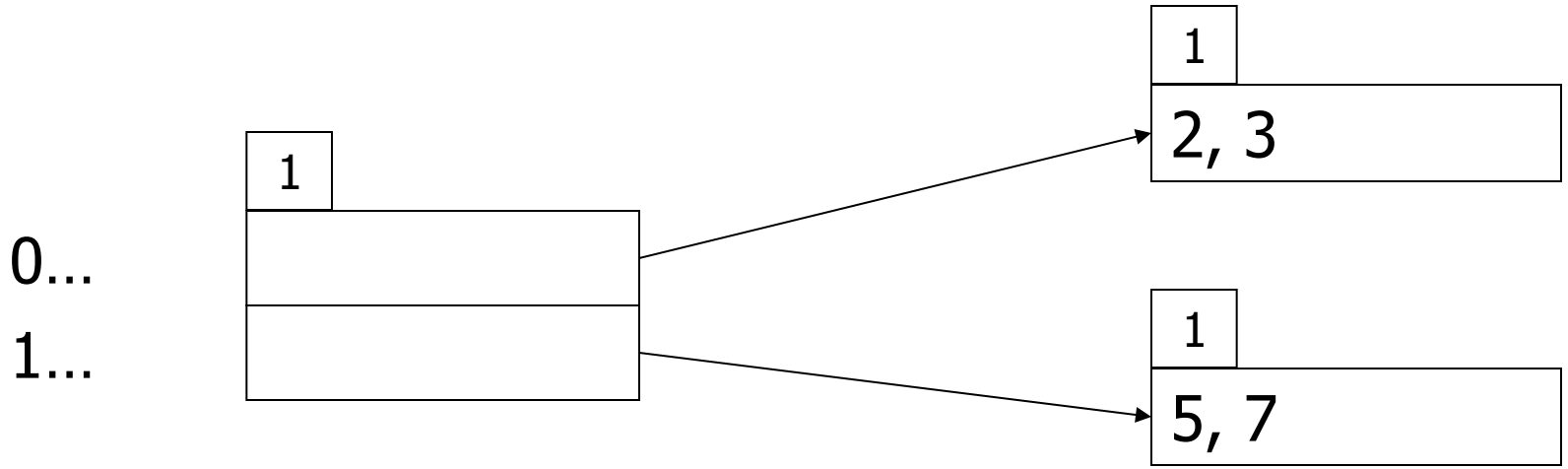
- Show the extensible hash structure if the hash function is `h(x)=x mod 8` and buckets can hold three records
- In this example, the i prefix bits are used

- Insert 2, 3, 5



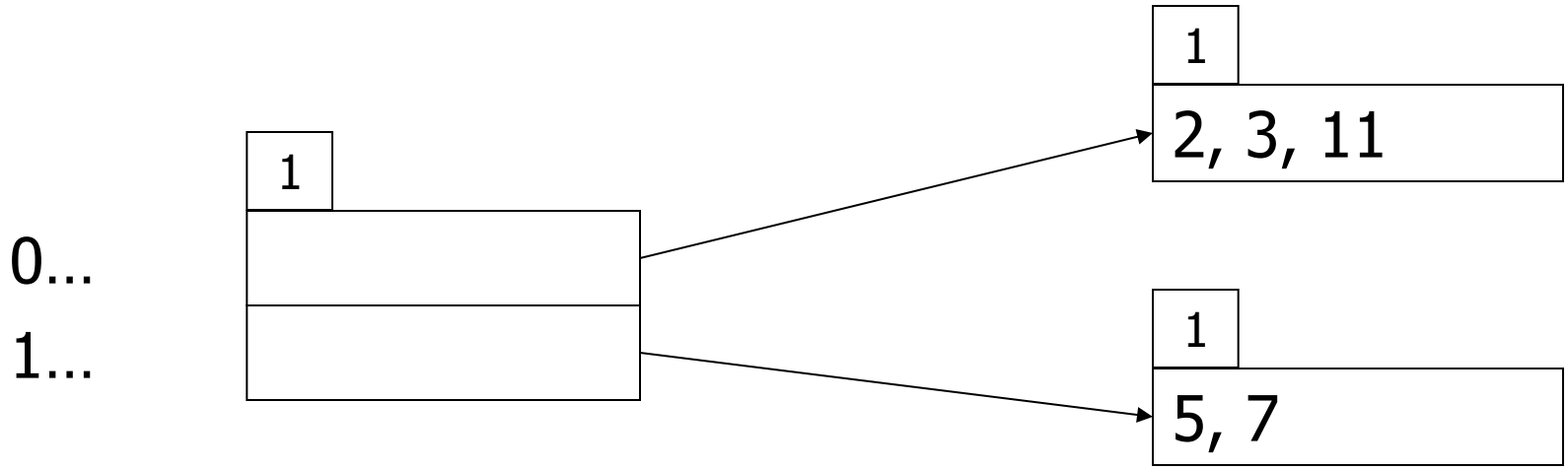
Next: Insert 7

Insert 7



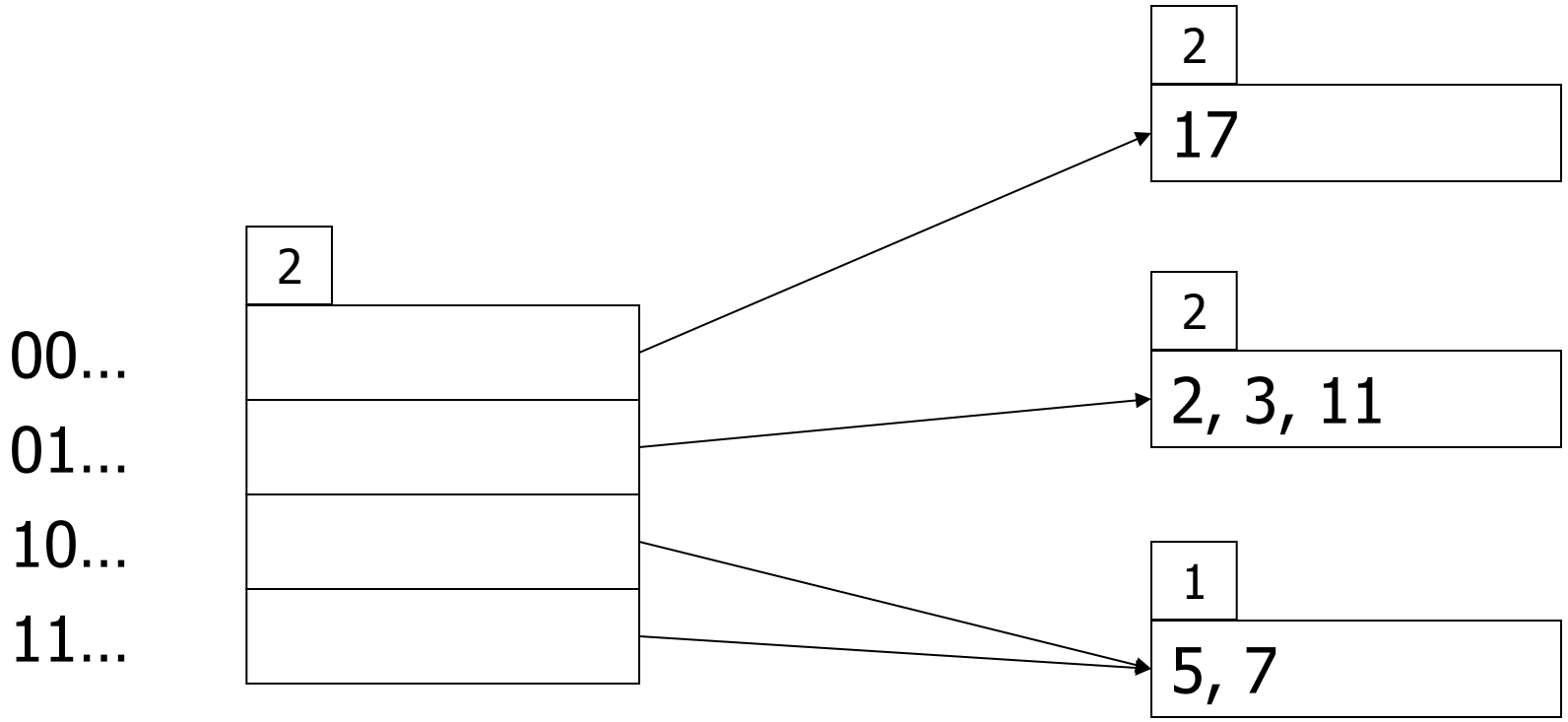
Next: Insert 11

Insert 11



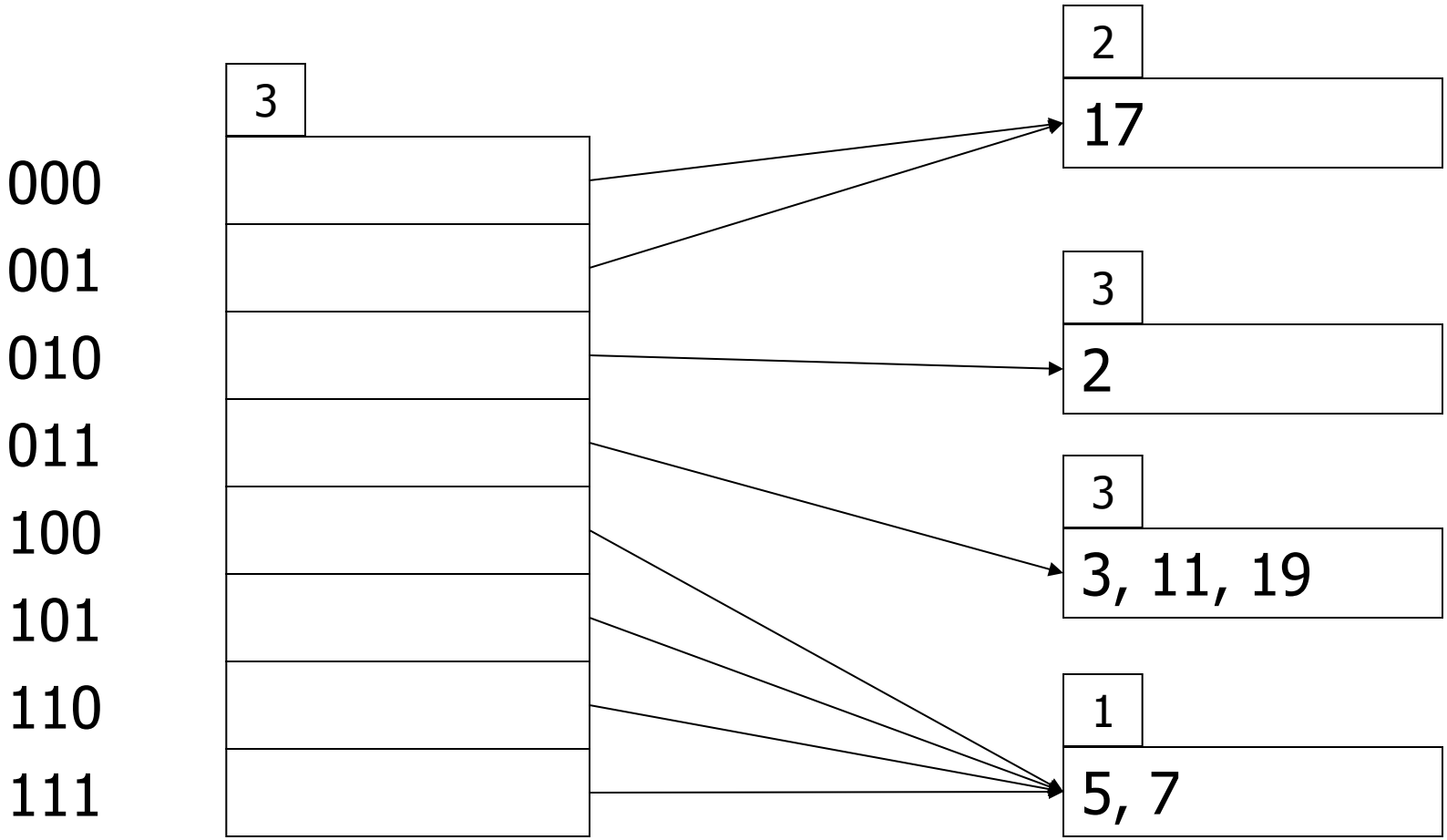
Next: Insert 17

Insert 17



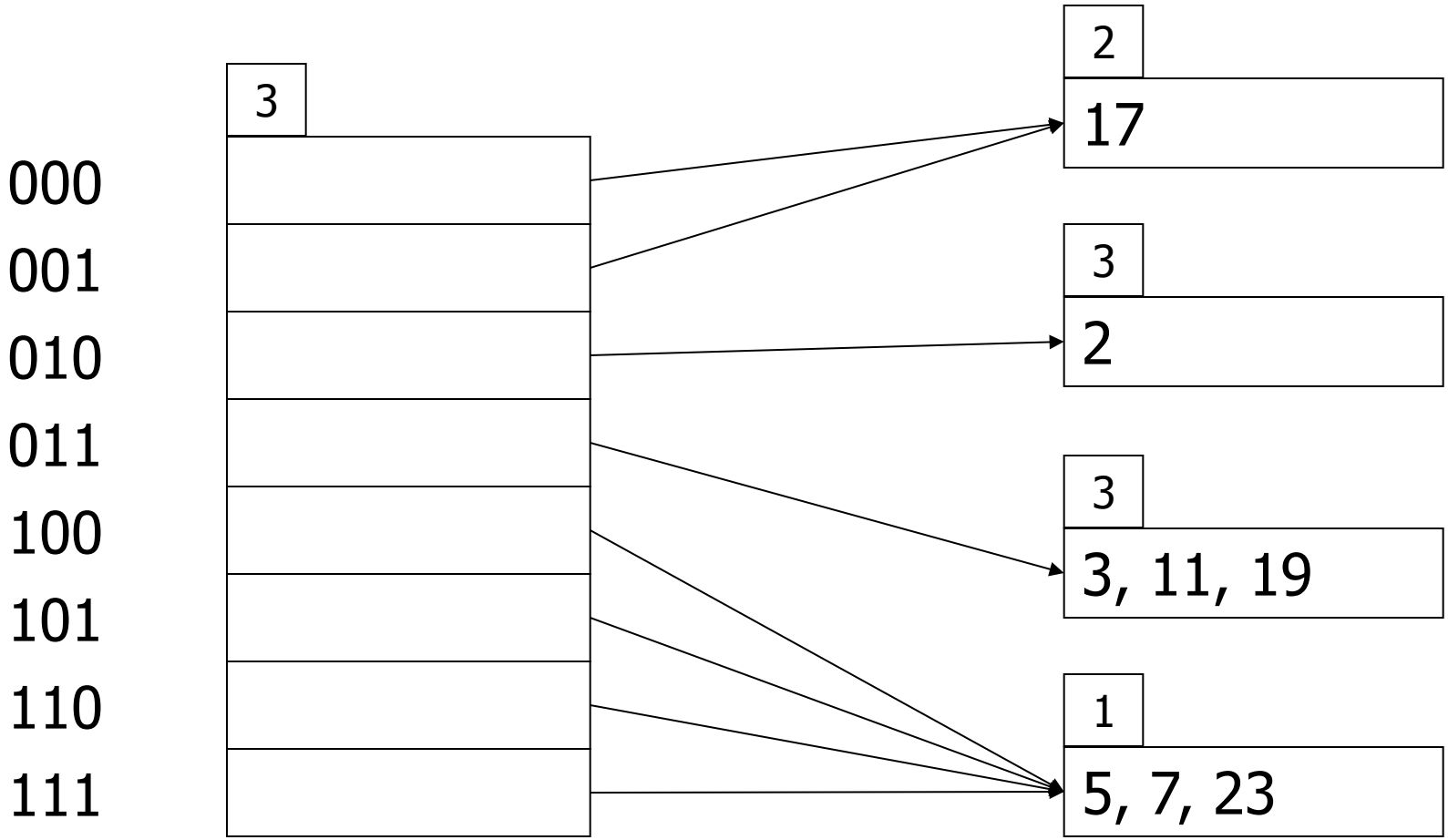
Next: Insert 19

Insert 19



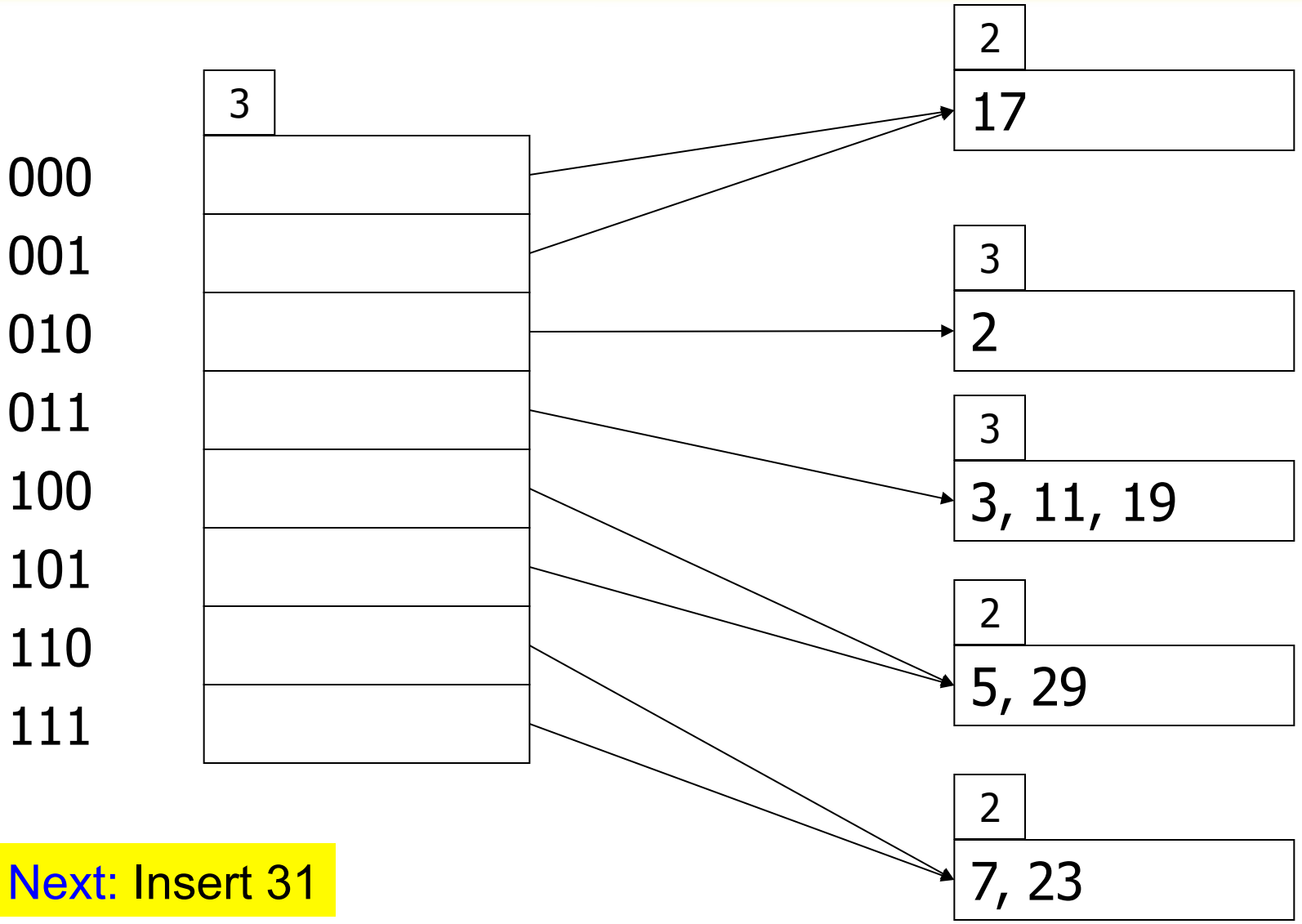
Next: Insert 23

Insert 23



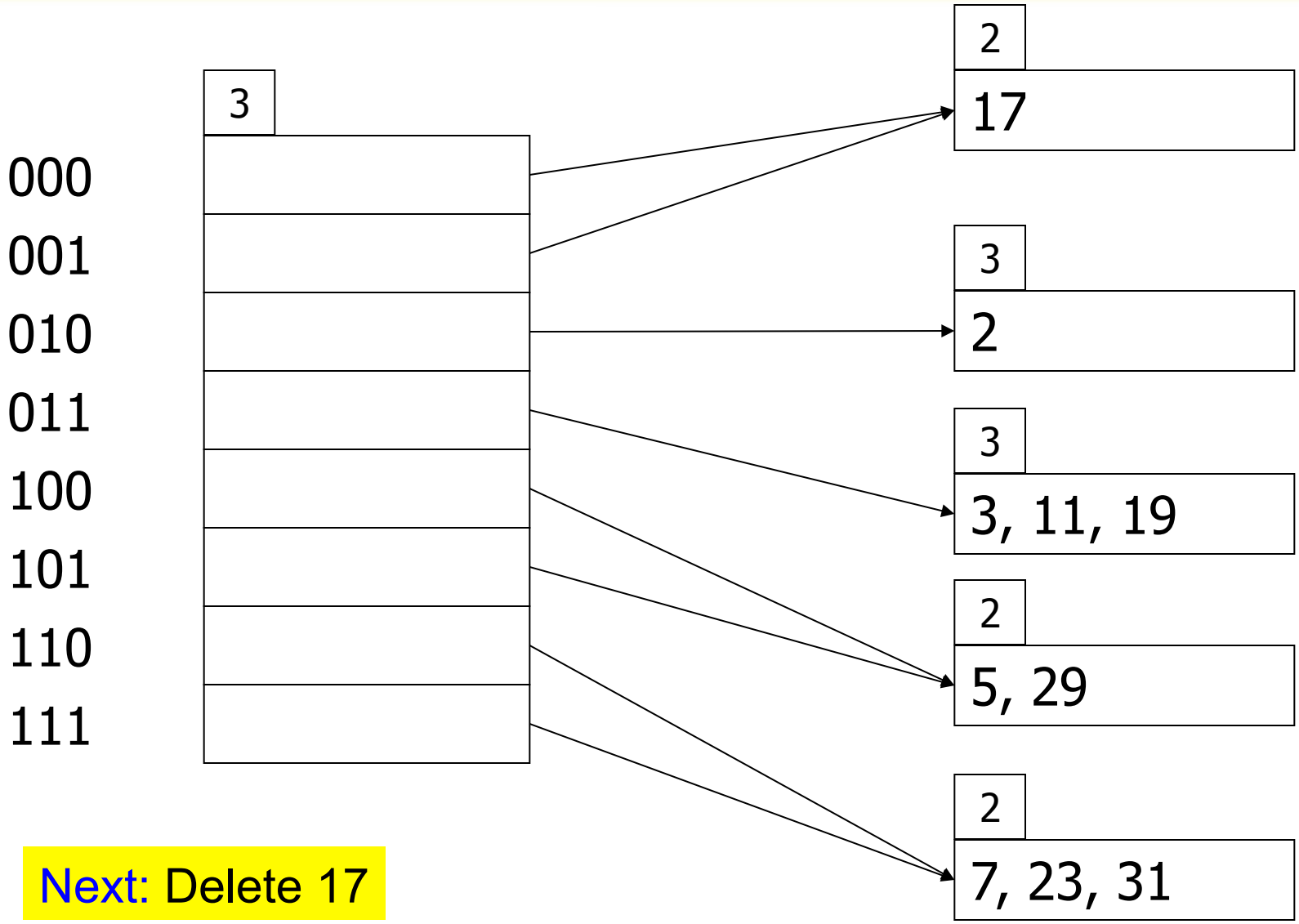
Next: Insert 29

Insert 29



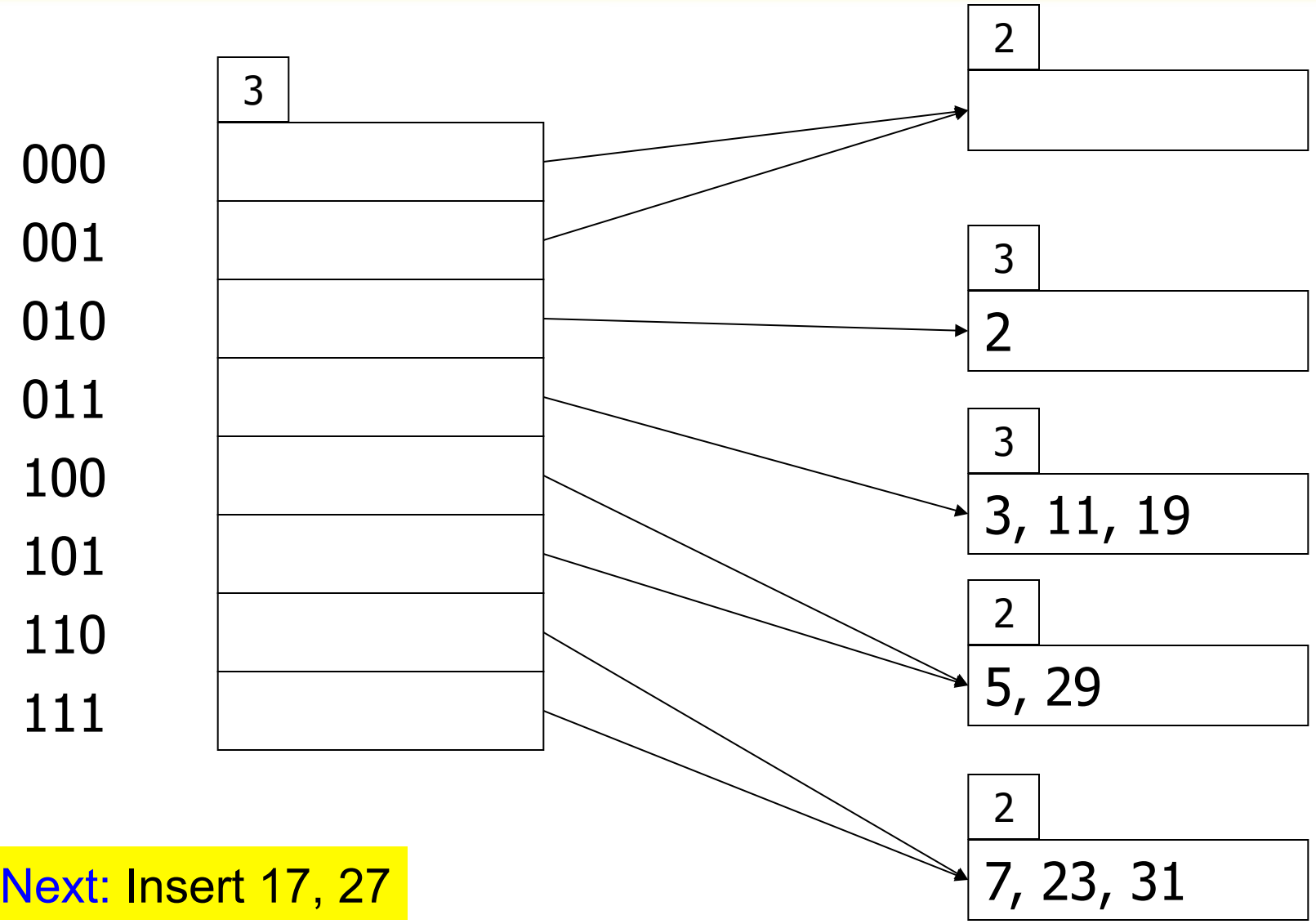
Next: Insert 31

Insert 31



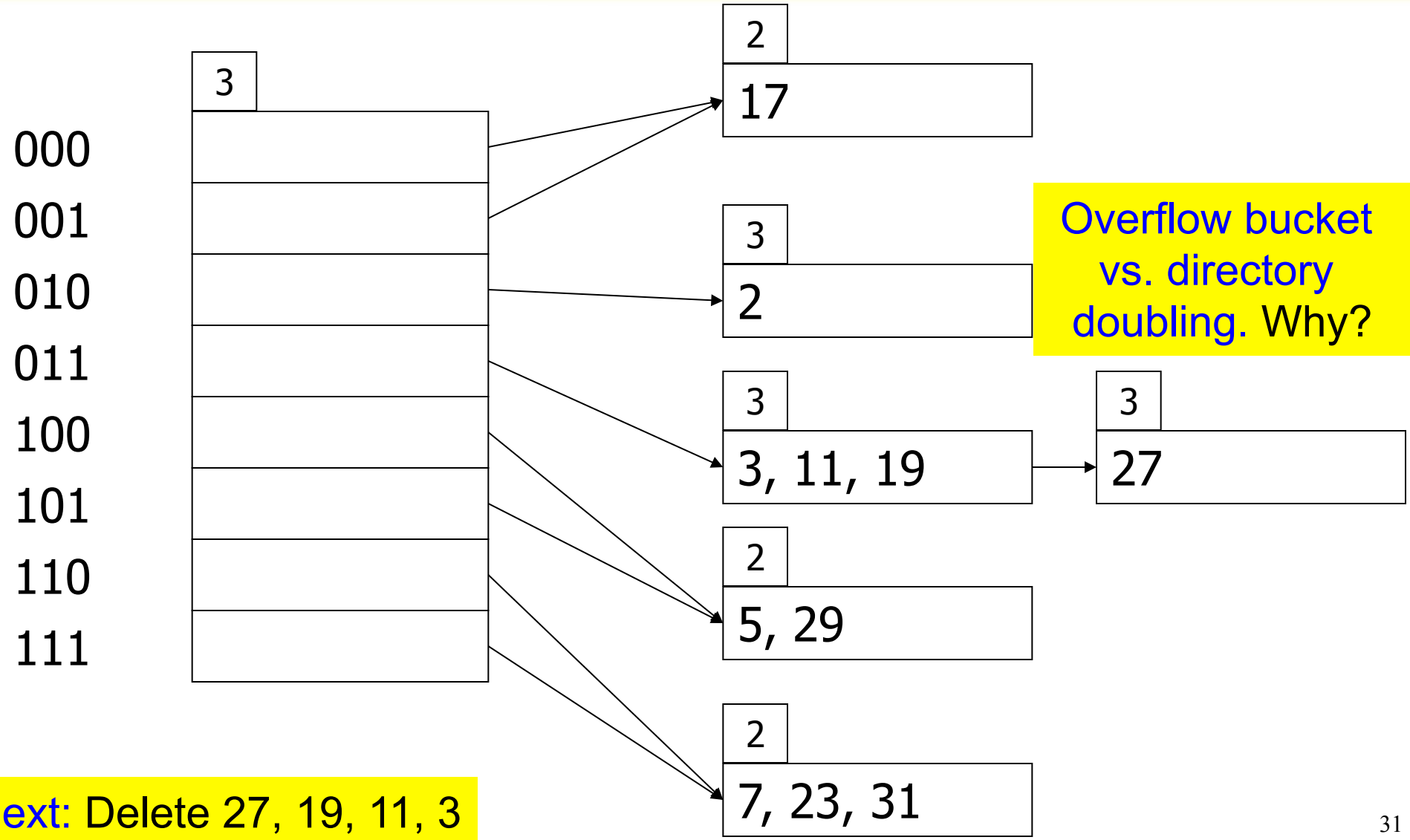
Next: Delete 17

Delete 17



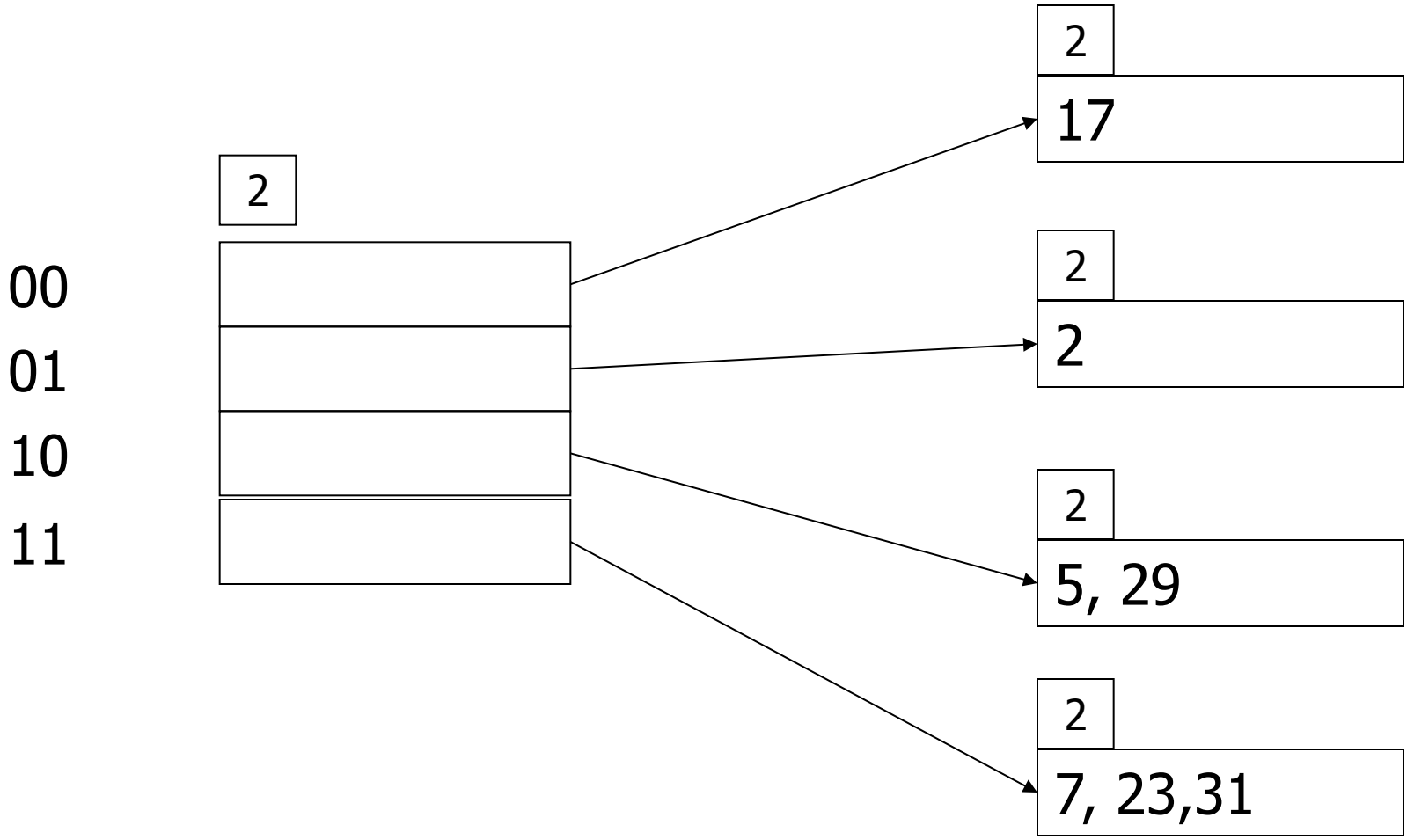
Next: Insert 17, 27

Insert 17, 27

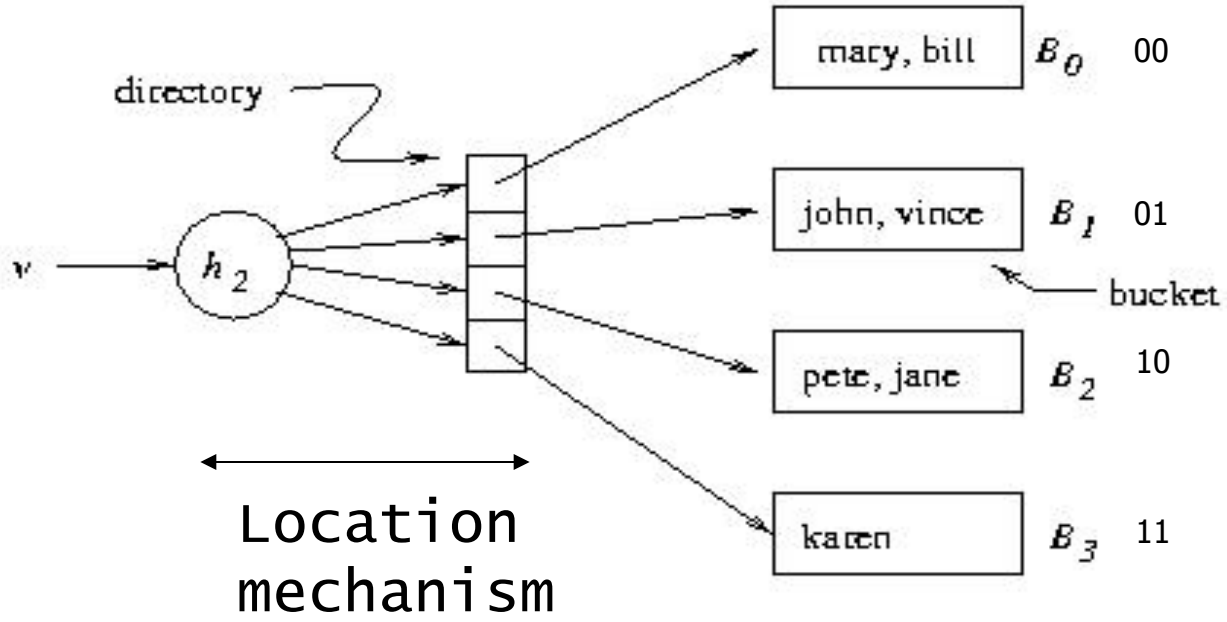


Next: Delete 27, 19, 11, 3

Delete 27, 19, 11, 3 (reduce to 2-level)



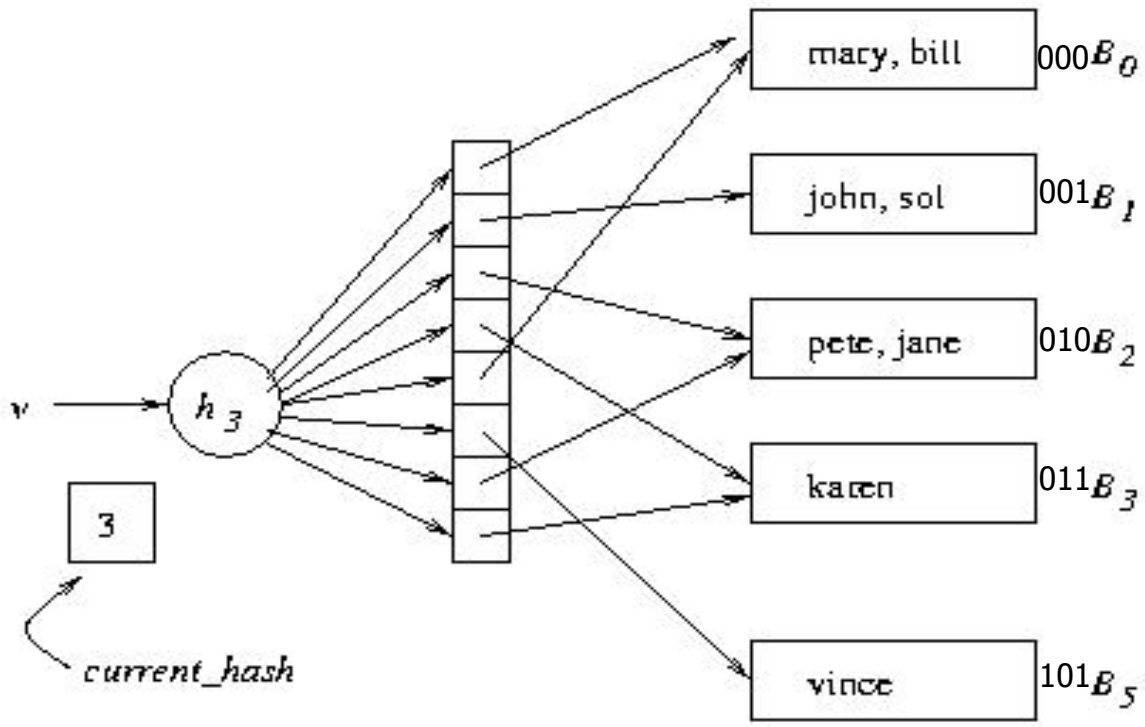
Extendible Hashing (EH): Second Example



v	h(v)
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111

- Extensible hashing uses a directory (level of indirection) to accommodate family of hash functions. In this example, the i suffix bits are used
- Suppose next action is to insert sol, where h(sol) = 10001
- Problem: This causes overflow in B_1

Extendible Hashing (EH): Second Example

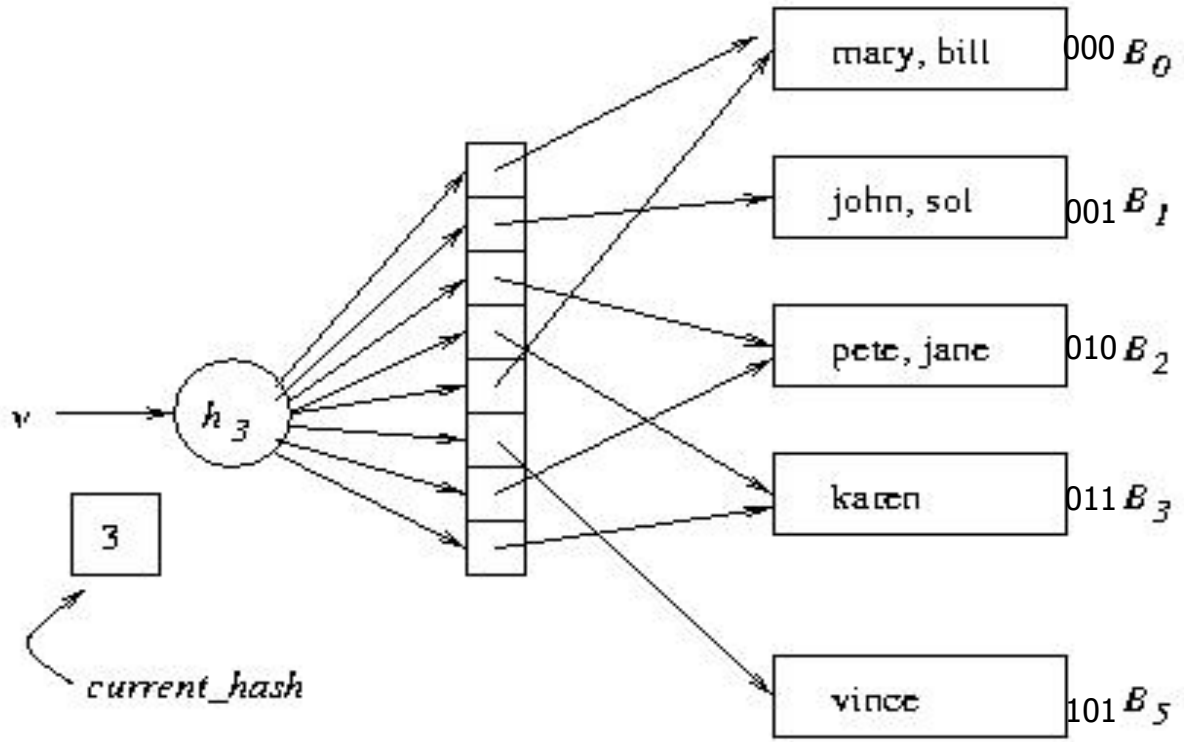


Solution:

1. Switch to h_3
2. Concatenate copy of old directory to new directory
3. Split overflowed bucket, *B*, into *B* and *B′*, dividing entries in *B* between the two using h_3
4. Pointer to *B* in directory copy replaced by pointer to *B′*

- Note: Except for *B′*, pointers in directory copy refer to original buckets
  - *current_hash* identifies current hash function

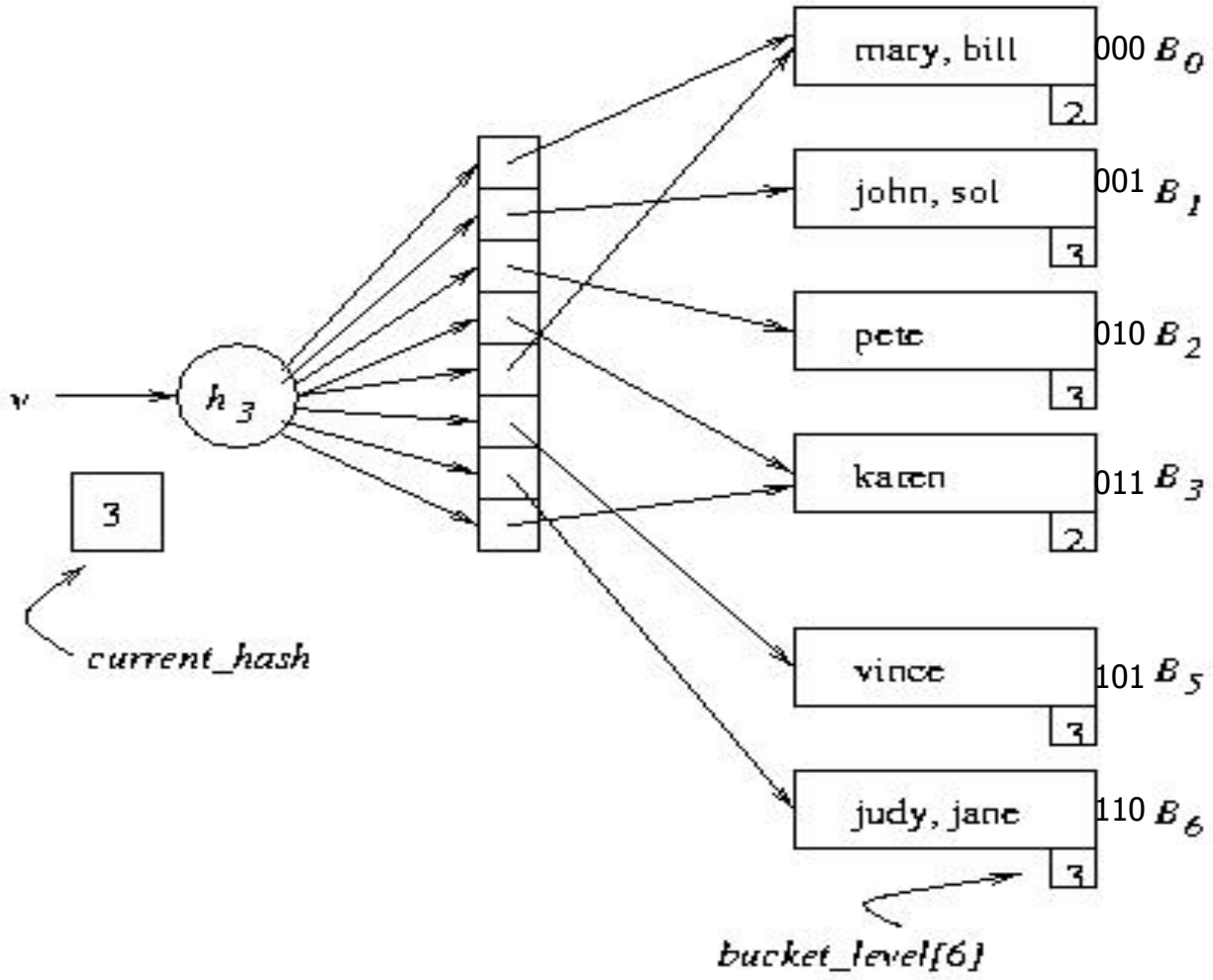
Extendible Hashing (EH): Second Example



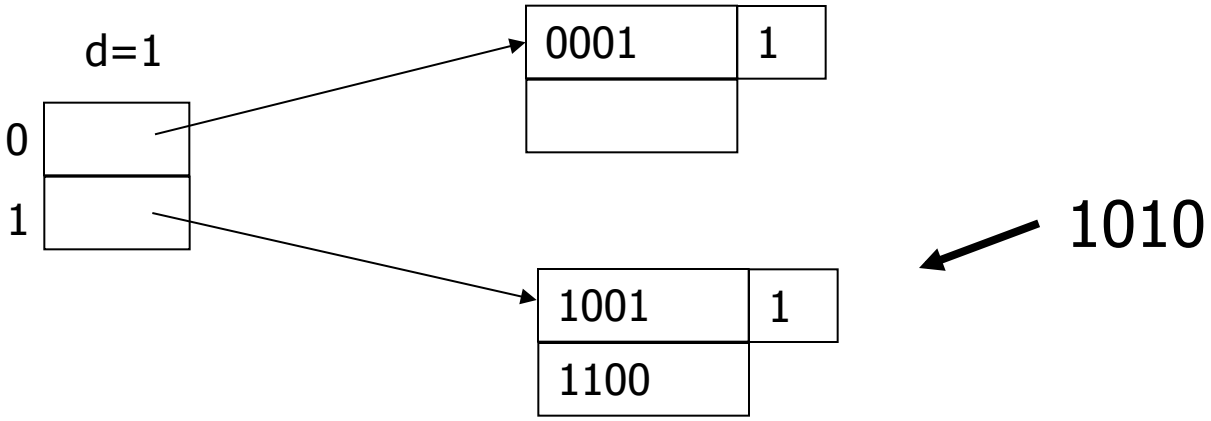
Next action: Insert judy, where h(judy)=00110
B_2 overflows, but directory need not be extended

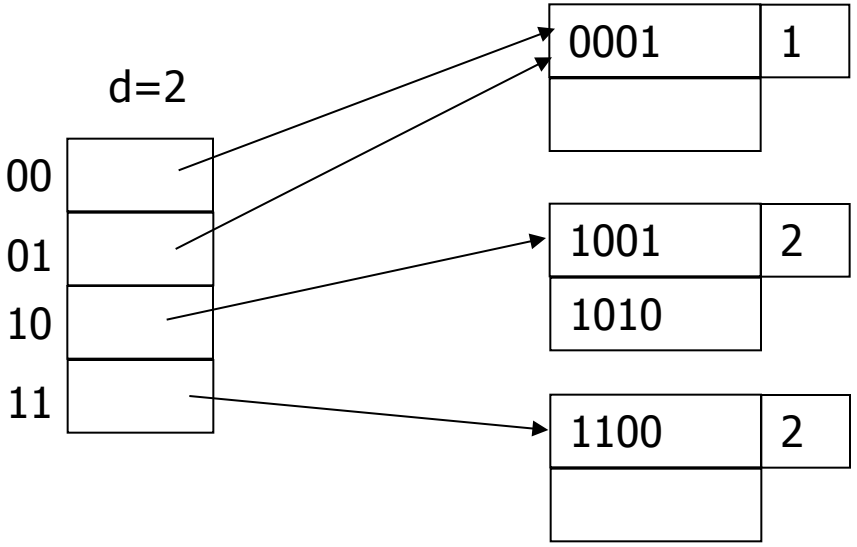
- Problem: When B_i overflows, we need a mechanism for deciding whether the directory has to be doubled
- Solution: *bucket_level[i]* records the number of times B_i has been split
  - If *current_hash* > *bucket_level[i],* do not enlarge directory

Extendible Hashing (EH): Second Example

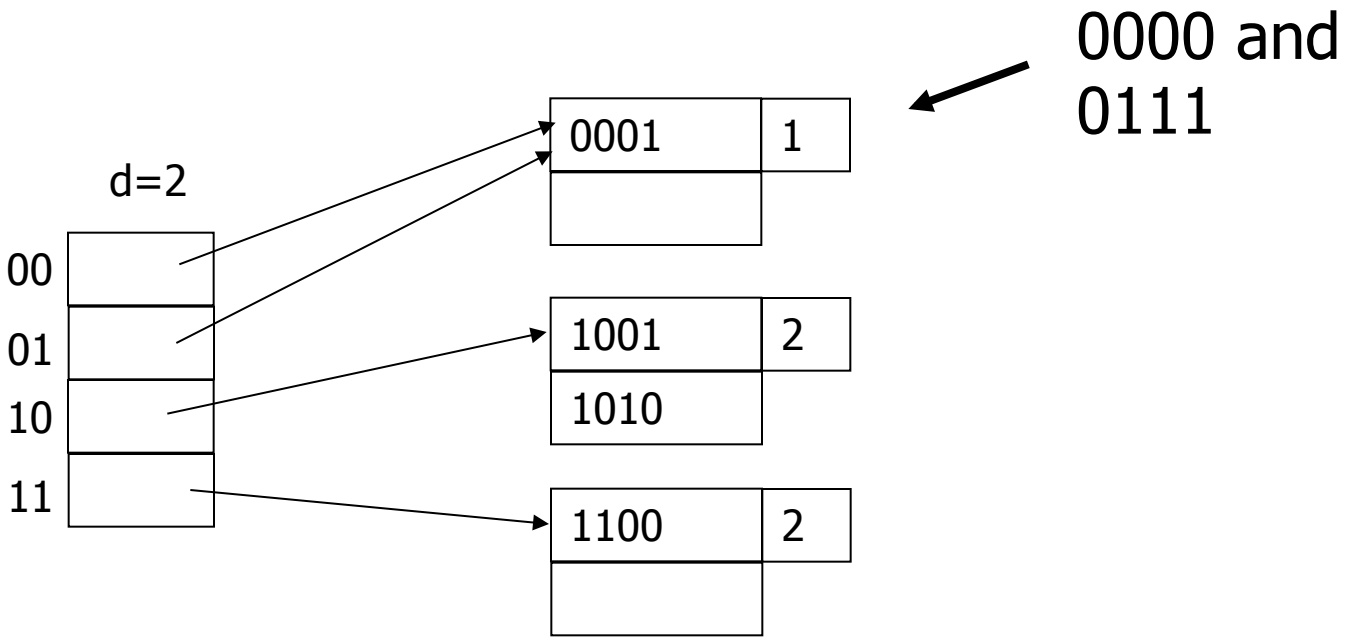


Extendible Hashing (EH): Third Example

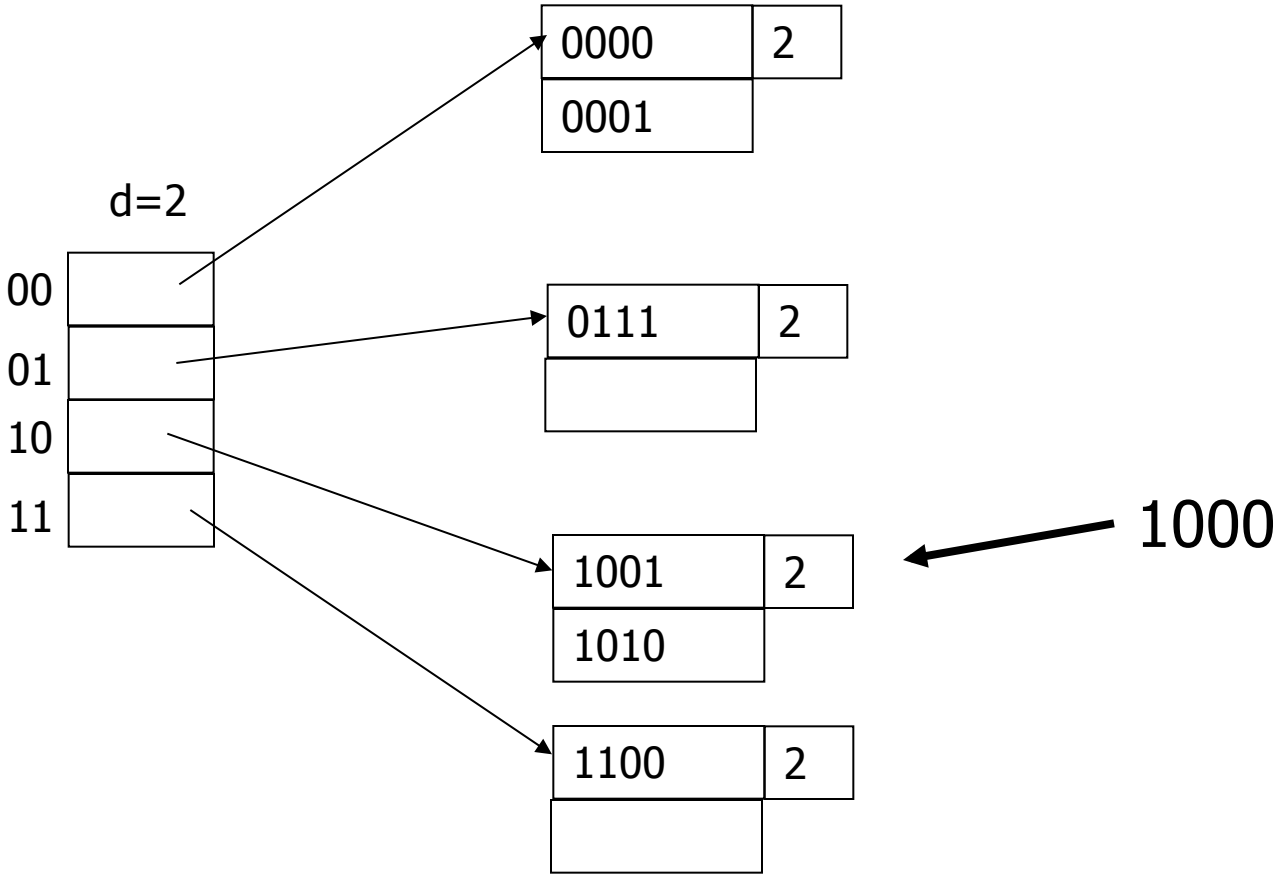




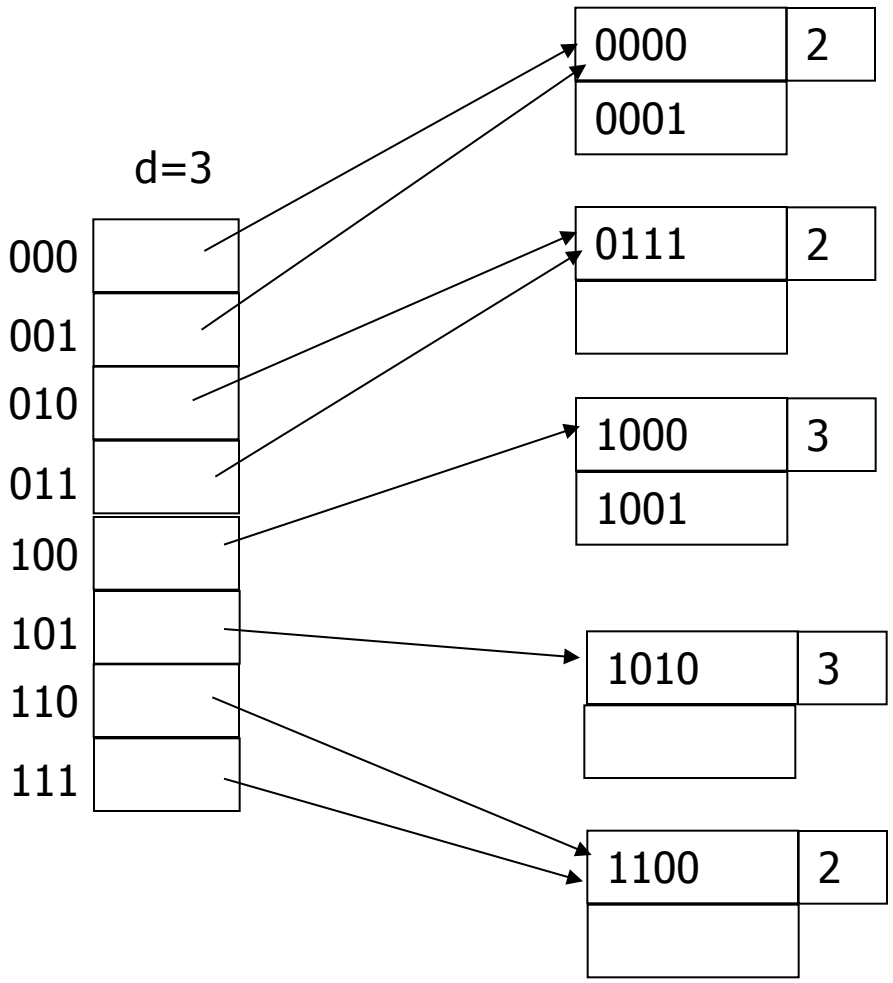
Extendible Hashing (EH): Third Example



Extendible Hashing (EH): Third Example



Extendible Hashing (EH): Third Example

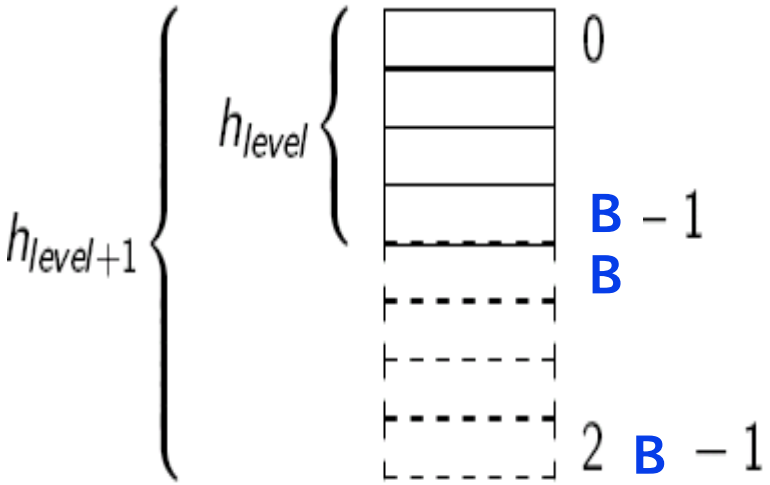


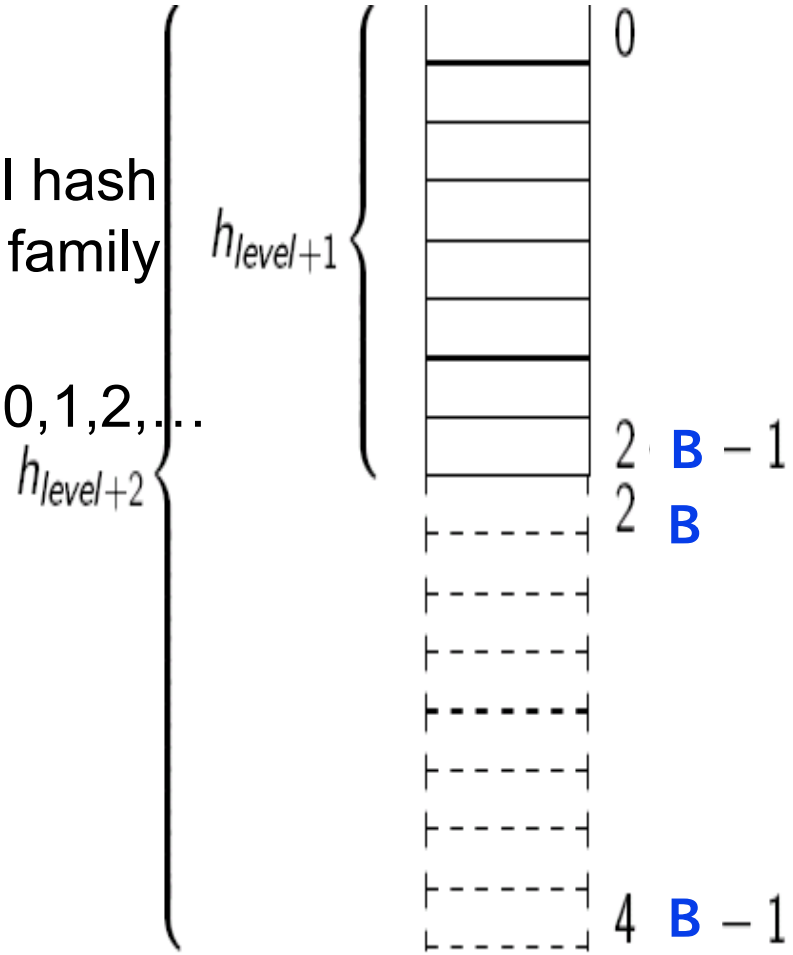
Linear Hashing (LH)

- Proposed by Litwin in 1981, linear hashing can, just like extendible hashing, adapt its underlying data structure to record insertions and deletions:
  - Linear hashing does not need a hash directory in addition to the actual hash table buckets,
  - Maintains a constant load factor; Incrementally add new buckets to the primary area; Lf = R / (B * c) B → # of buckets, c → #records per block
  - . . . but linear hashing may perform bad if the key distribution in the data file is skewed
- The core idea behind linear hashing is to use an ordered family of hash functions, h_0, h_1, h_2, . . . (traditionally the subscript is called the hash function's `level`);
  - Calculate a hash number for a given `level`
  - Calculate boundary value: c * Lf
  - Place the record according to its last k digits
  - In case an expand is required, split an existing bucket using its k+1 last digits of the hash number
- Note: Boundary value & current value of k should be kept on the file header

Linear Hashing (LH): Hash Function Family

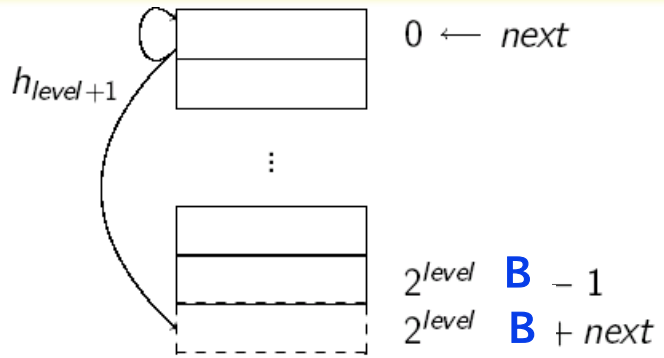
- We design the family so that the range of $h_{level+1}$ is twice as large as the range of h_{level} (for level = 0,1,2, ...) as depicted below
  - Here, h_{level} has range [0 . . . B − 1] (i.e., a range of size B)
- Given an initial hash function h and an initial hash table size B, one approach to define such a family of hash functions h_0, h_1, h_2, . . .would be
  - $h_{level}(k) = h(k) \bmod (2^{level} \cdot B)$ for level=0,1,2,...





Linear Hashing (LH): Basic Scheme

- Start with `level` = 0, `next` = 0
- The current hash function in use for searches (insertions /deletions) is h_{level}, active hash table buckets are those in h_{level}'s range: [0 . . . 2^{level}.B – 1]



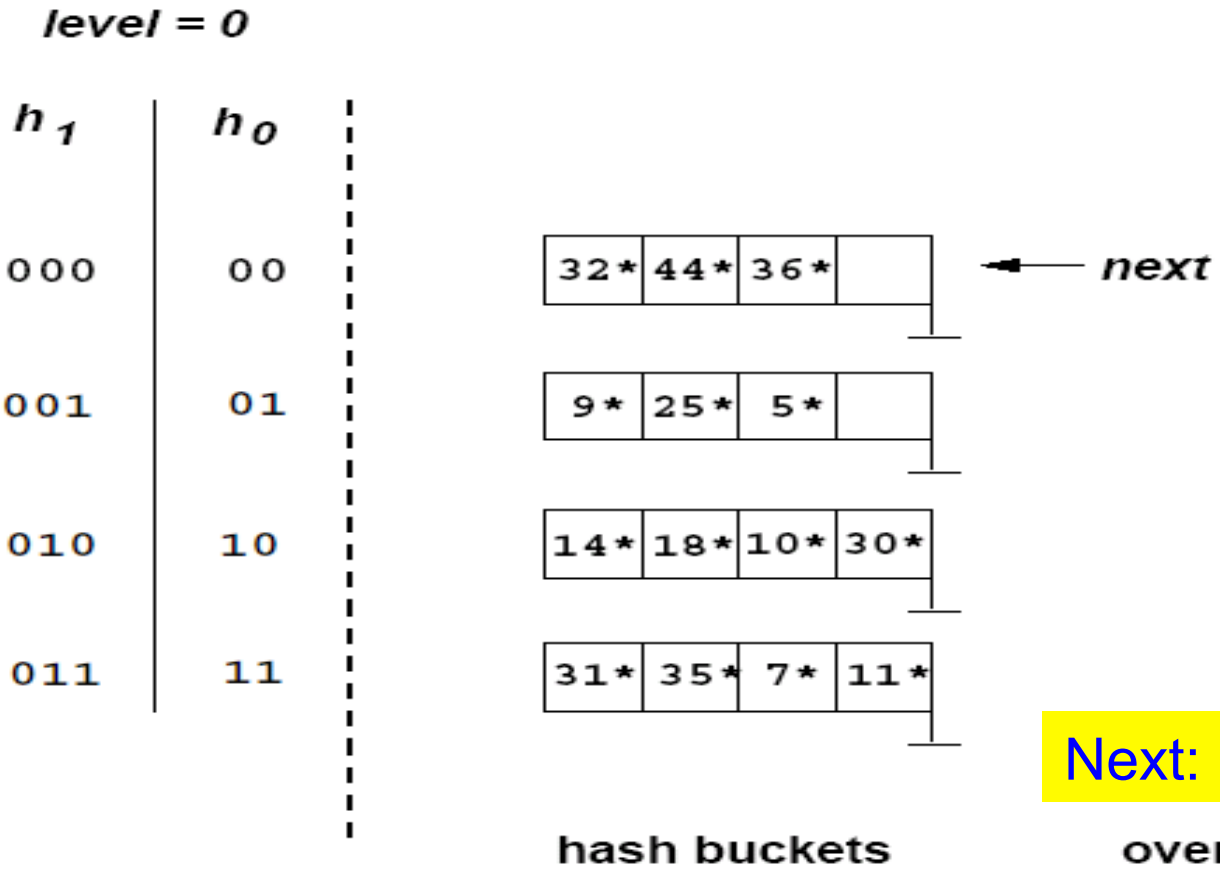
- Whenever we realize that the current hash table overflows, e.g.,
  - insertions filled a primary bucket beyond c % capacity,
  - or the overflow chain of a bucket grew longer than p pages, or . . .

we split the bucket at hash table position `next` (in general, this is not the bucket which triggered the split!)

1. Allocate a new bucket, append to it the hash table (its position will be 2^{level}.B+`next`),
2. Redistribute the entries in bucket next by rehashing them via $h_{level+1}$ (some entries remain in bucket next, some go to bucket 2^{level}.B +`next`),
3. Increment `next` by 1

Linear Hashing (LH): First Example

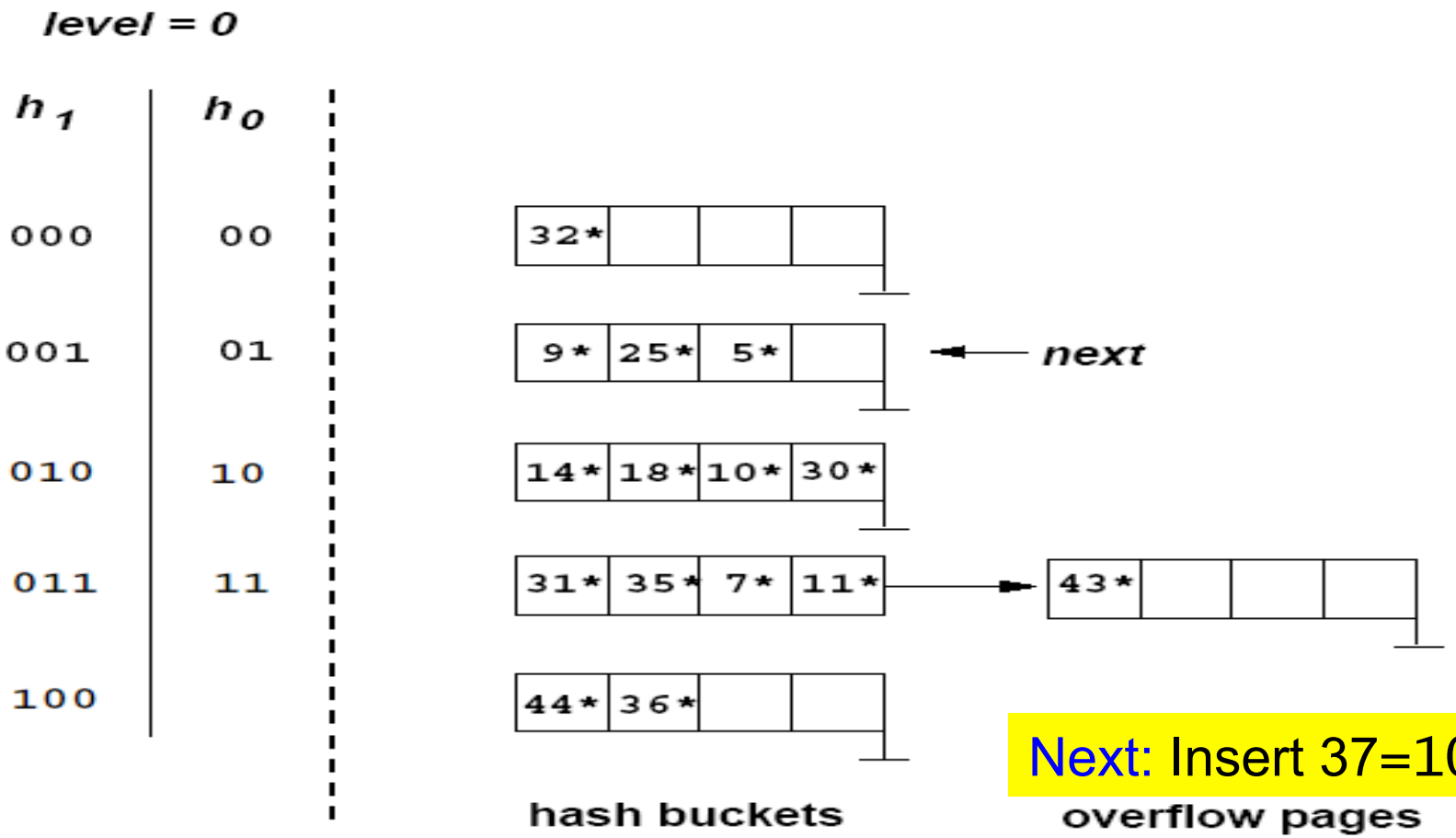
- Linear hash table: primary bucket capacity of 4, initial hash table size N = 4, level = 0, next = 0
  - Split criterion: allocation of a page in an overflow chain



Next: Insert 43=101011_2

Insert 43

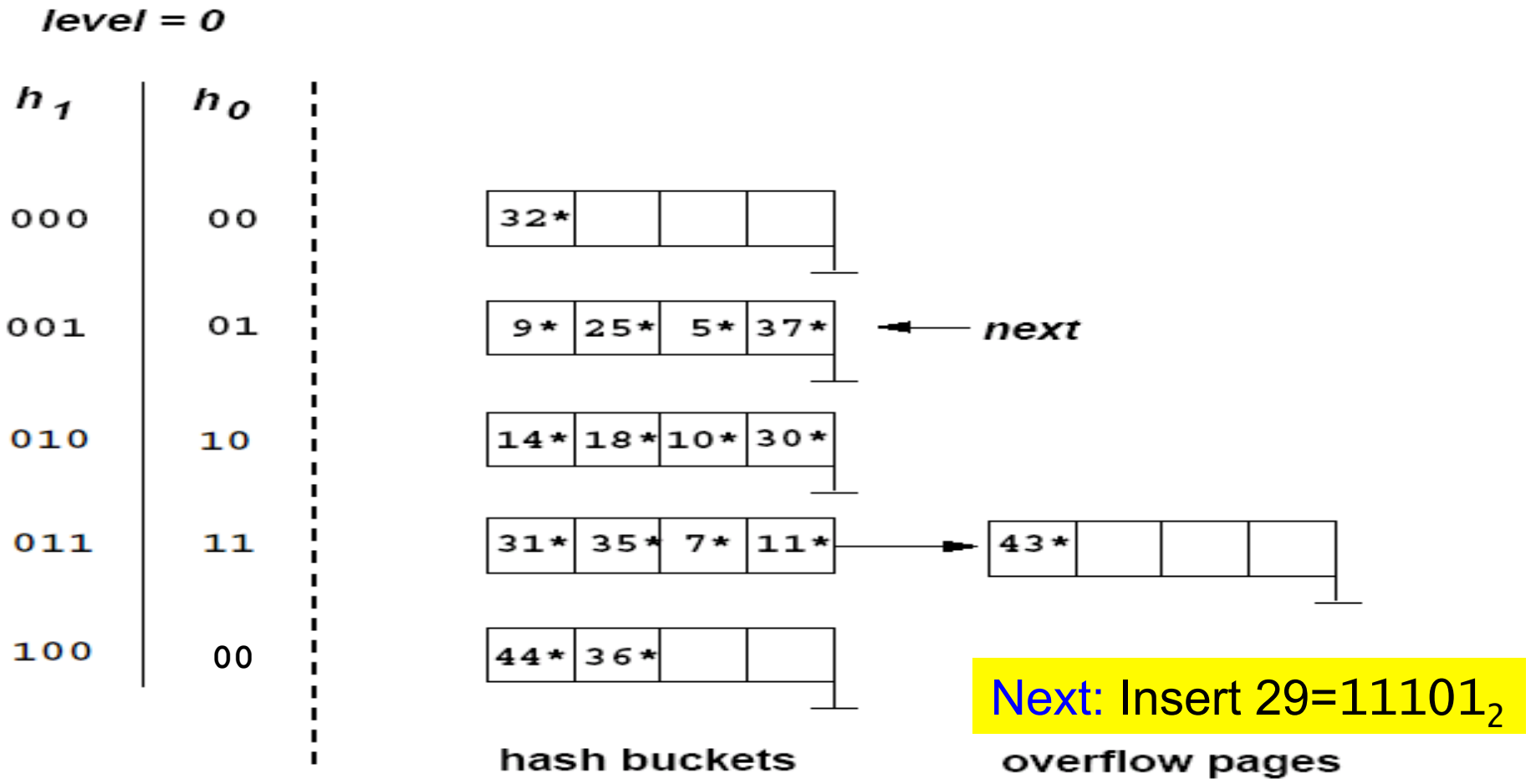
- Insert record with key k such that $h_0(k) = 43 = 101011_2$:



Next: Insert $37=100101_2$

Insert 37

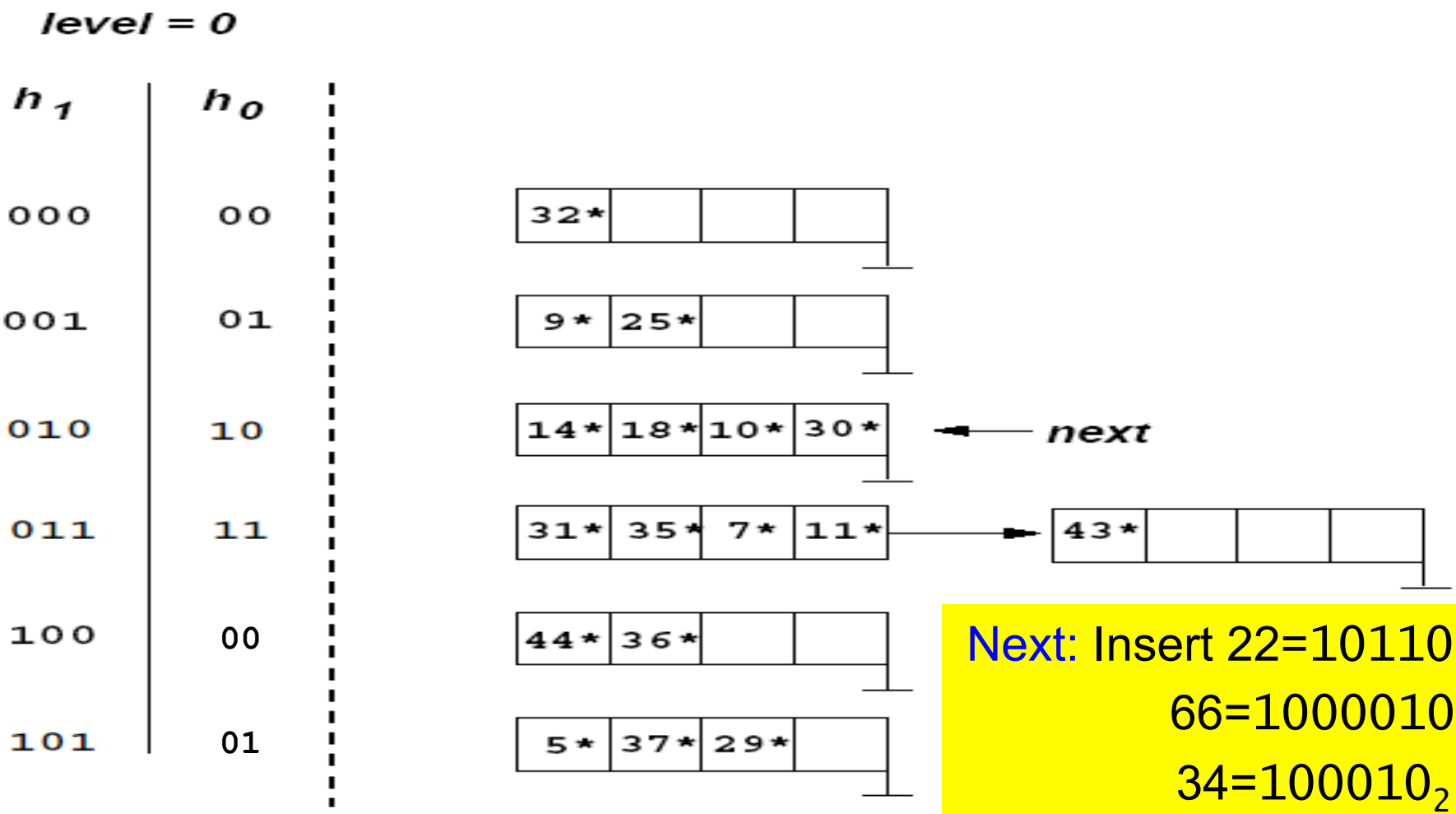
- Insert record with key k such that $h_0(k) = 37 = 100101_2$:



Next: Insert $29=11101_2$

Insert 29

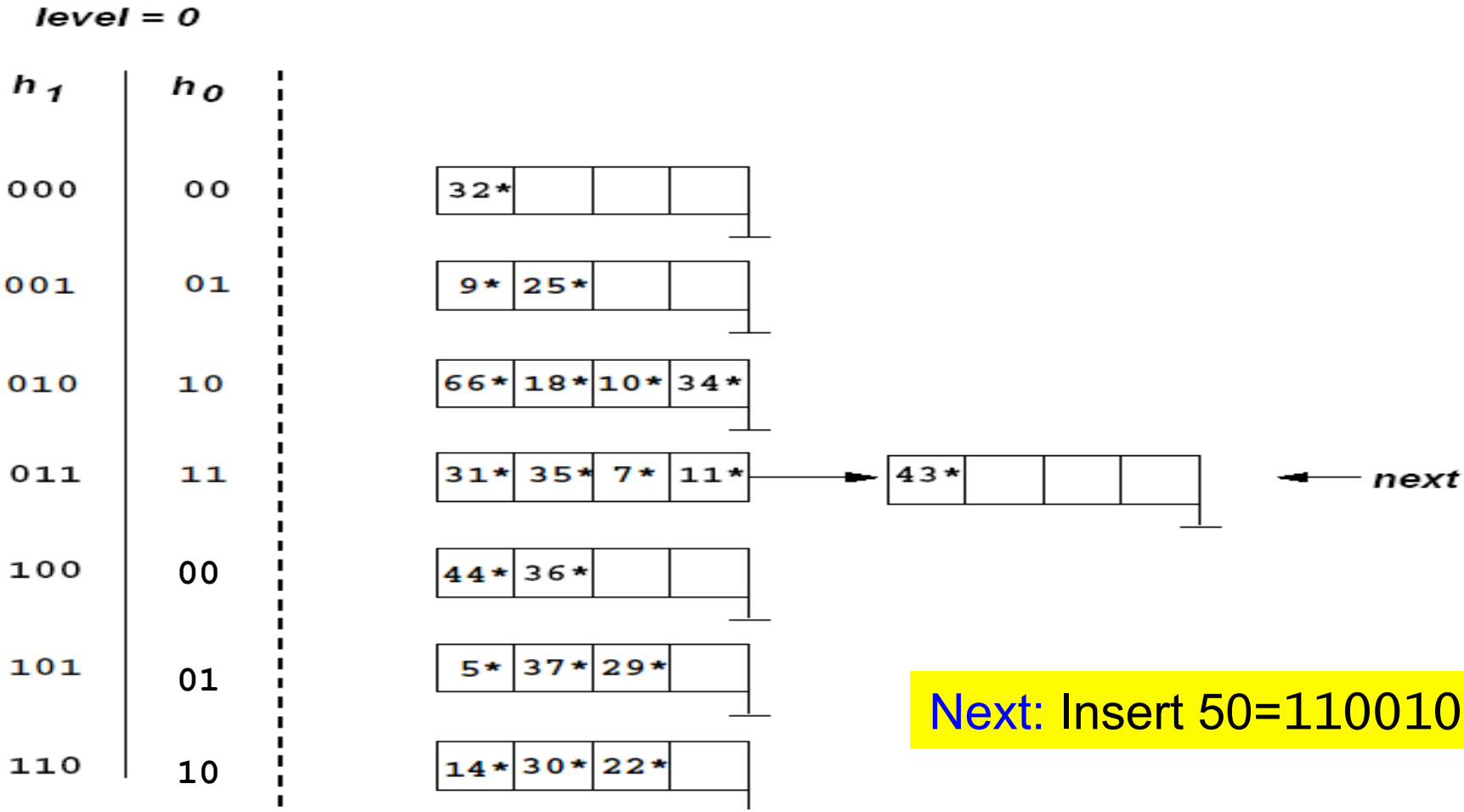
- Insert record with key k such that $h_0(k) = 29 = 11101_2$:



Next: Insert $22=10110_2$, $66=1000010_2$, $34=100010_2$

Insert 22, 36, 34

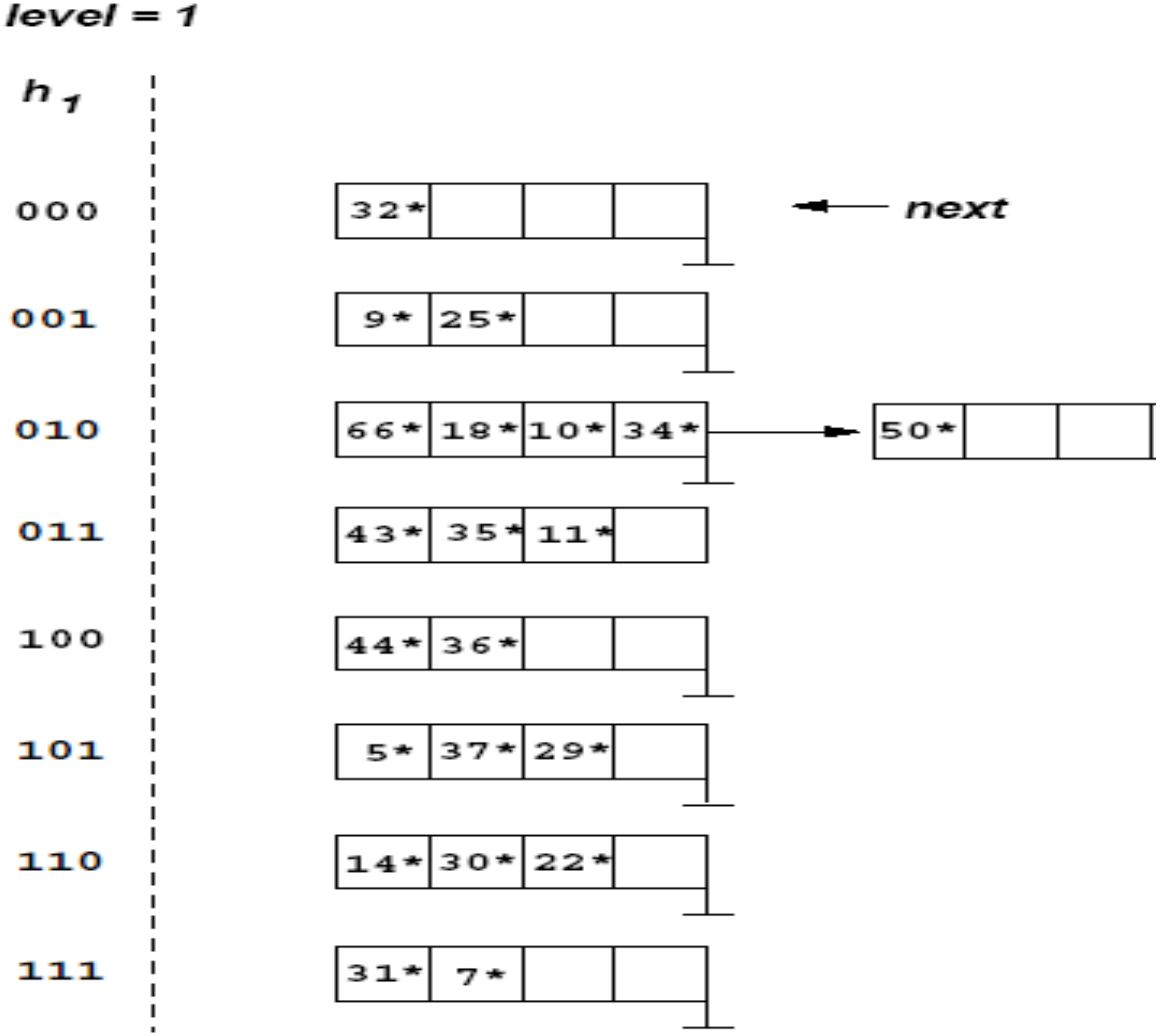
- Insert 3 records with keys k such that $h_0(k) = 22 = 10110_2$ ($66 = 1000010_2$, $34 = 100010_2$):



Next: Insert $50=110010_2$

Insert 50

- Insert record with key k such that $h_0(k) = 50 = 110010_2$:



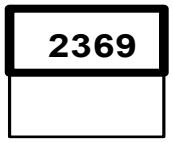
Linear Hashing (LH): Second Example (Alter. Algo)

i ≡ level

n ≡ next

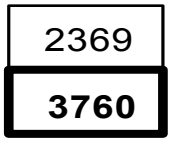
1 - Insert 2369

$h0(K) = K \bmod 2^0$
$h1(K) = K \bmod 2^1$
i=0, n=0



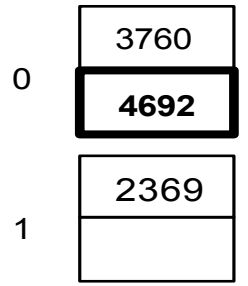
2 - Insert 3760

$h0(K) = K \bmod 2^0$
$h1(K) = K \bmod 2^1$
i=0, n=0



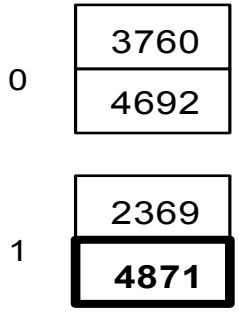
3 - Insert 4692

$h1(K) = K \bmod 2^1$
$h2(K) = K \bmod 2^2$
i=1, n=0



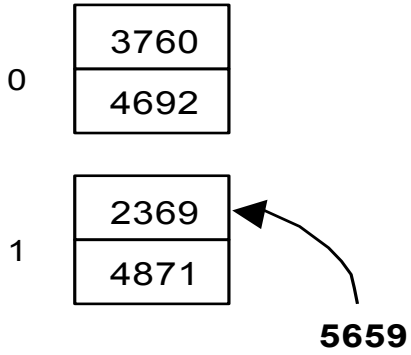
4 - Insert 4871

$h1(K) = K \bmod 2^1$
$h2(K) = K \bmod 2^2$
i=1, n=0



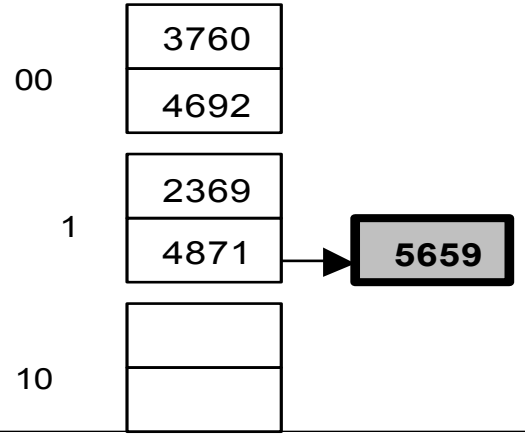
5a - Insert 5659

$h1(K) = K \bmod 2^1$
$h2(K) = K \bmod 2^2$
i=1, n=0



5b - Insert 5659

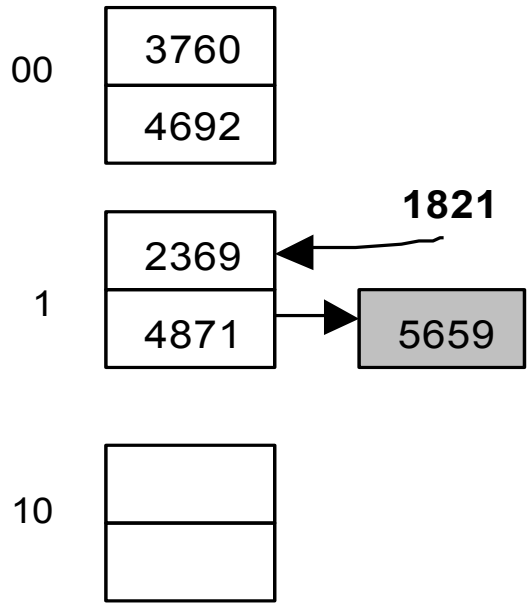
$h1(K) = K \bmod 2^1$
$h2(K) = K \bmod 2^2$
i=1, n=1



Linear Hashing (LH): Second Example

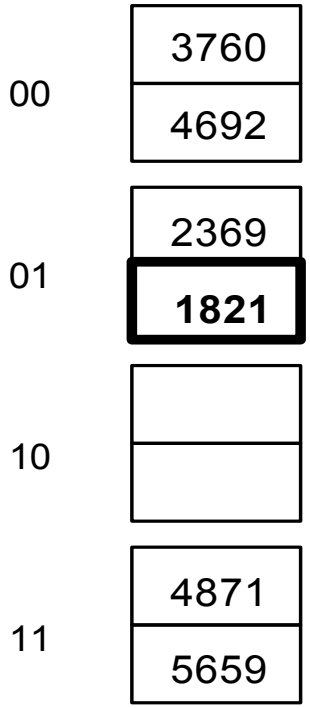
6a - Insert 1821

$h1(K) = K \bmod 2^1$
$h2(K) = K \bmod 2^2$
i=1, n=1



6b - Insert 1821

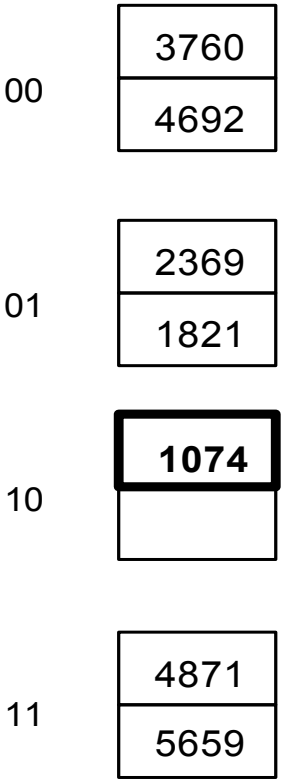
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=0



Linear Hashing (LH): Second Example

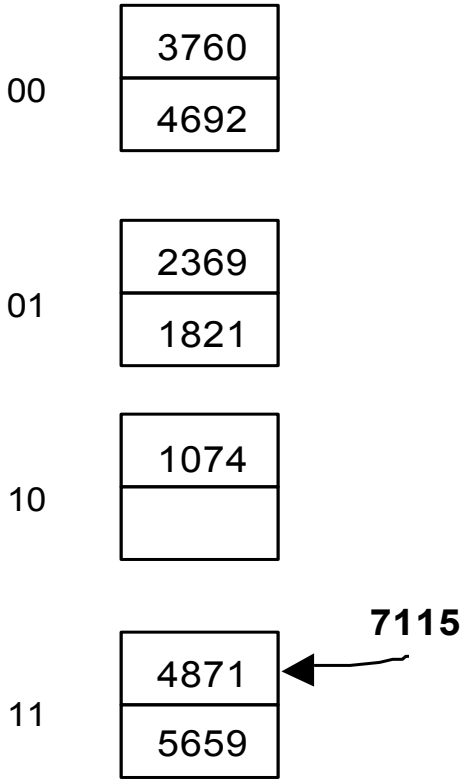
7 - Insert 1074

$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=0



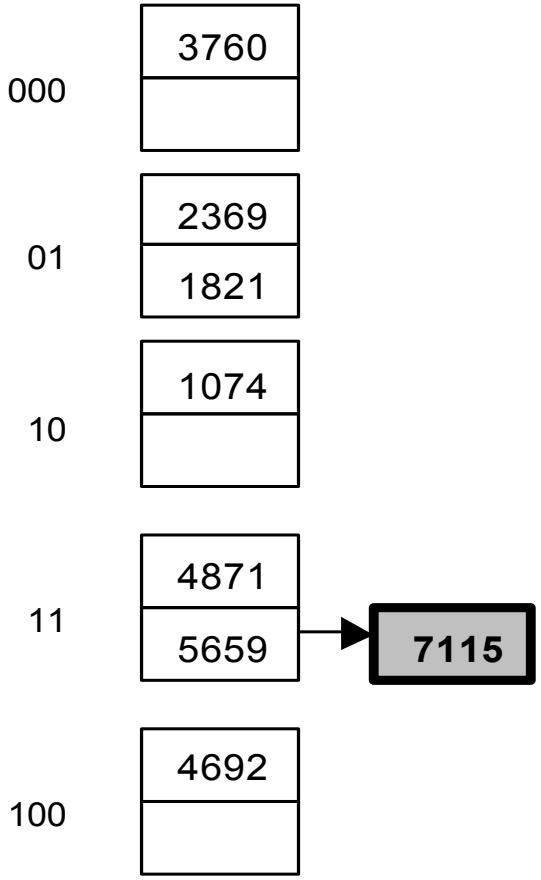
8a - Insert 7115

$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=0



8b - Insert 7115

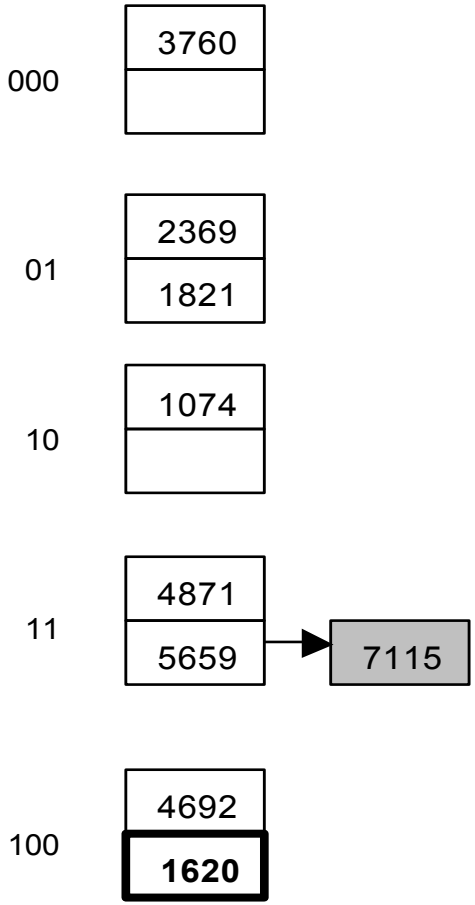
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=1



Linear Hashing (LH): Second Example

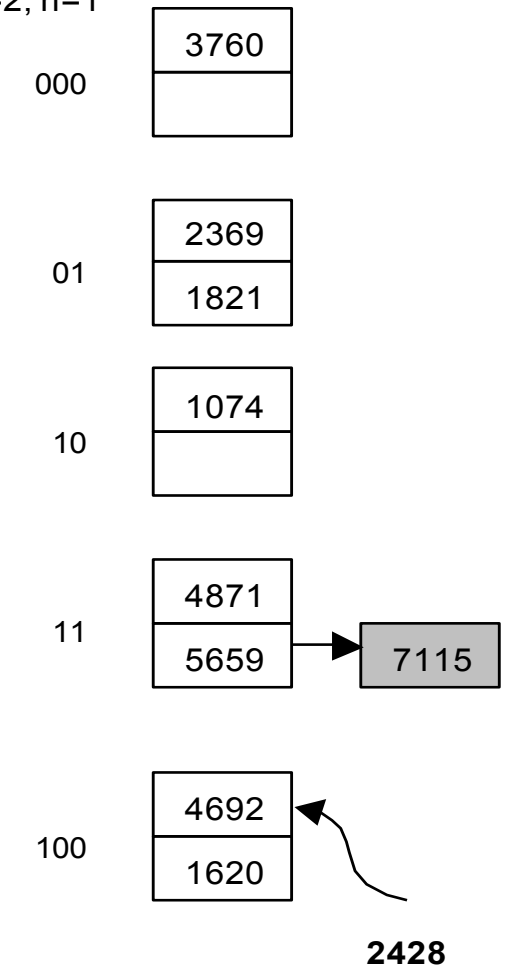
9 - Insert 1620

$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=1



10a - Insert 2428

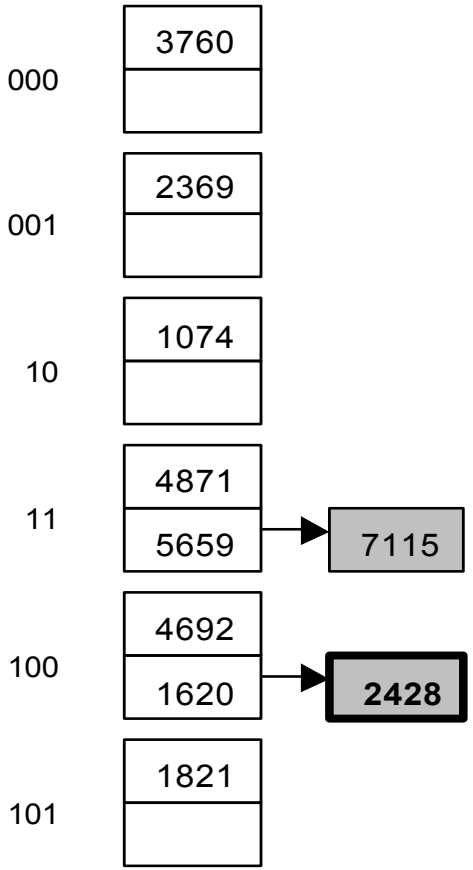
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=1



Linear Hashing (LH): Second Example

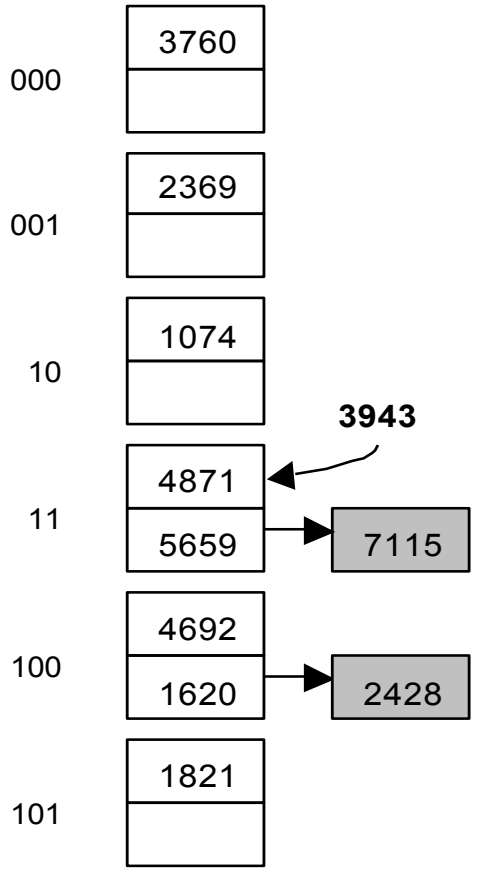
10b - Insert 2428

$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=2



11a - Insert 3943

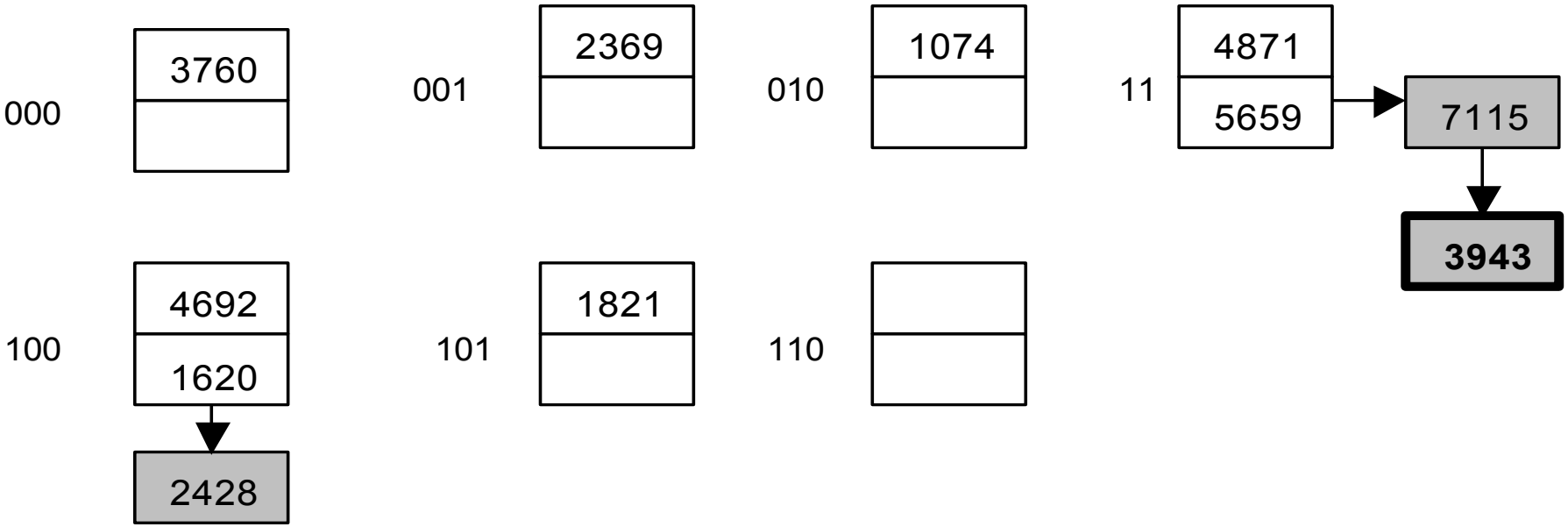
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=2



Linear Hashing (LH): Second Example

11b - Insert 3943

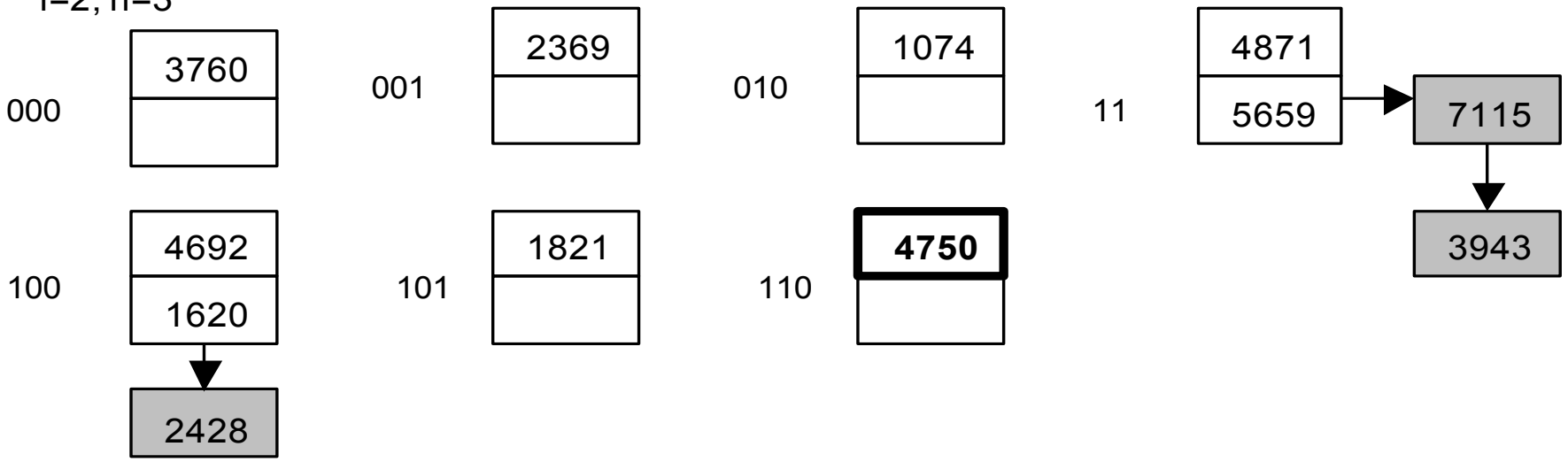
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=3



Linear Hashing (LH): Second Example

12 - Insert 4750

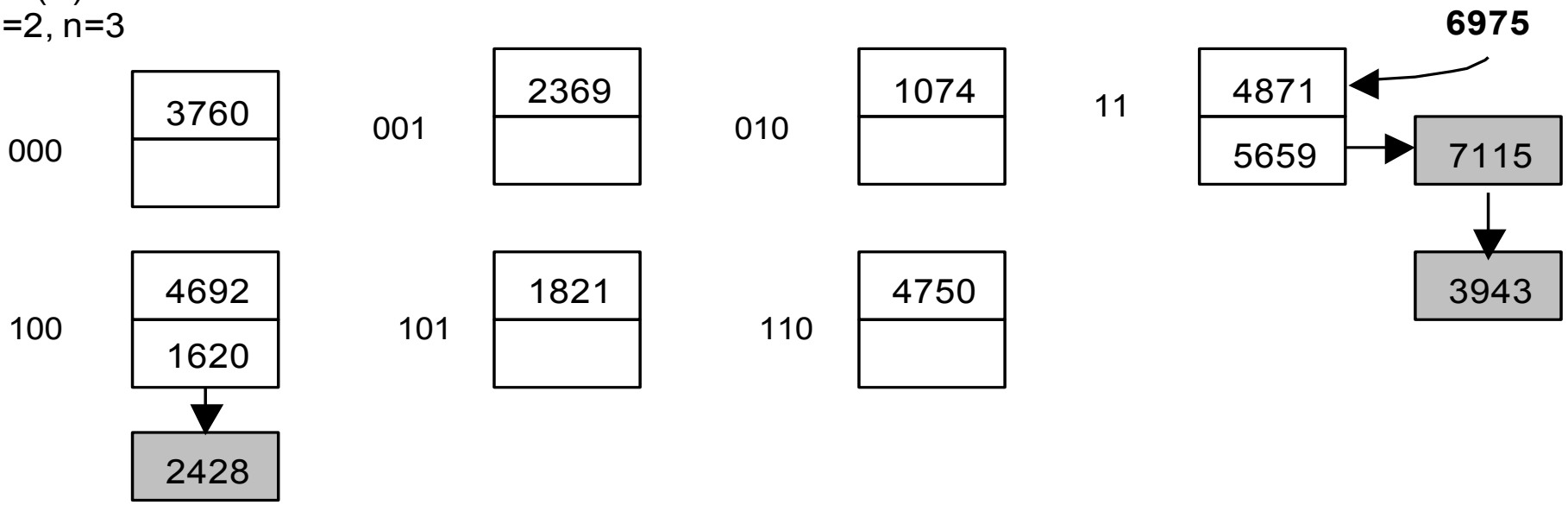
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=3



Linear Hashing (LH): Second Example

13a - Insert 6975

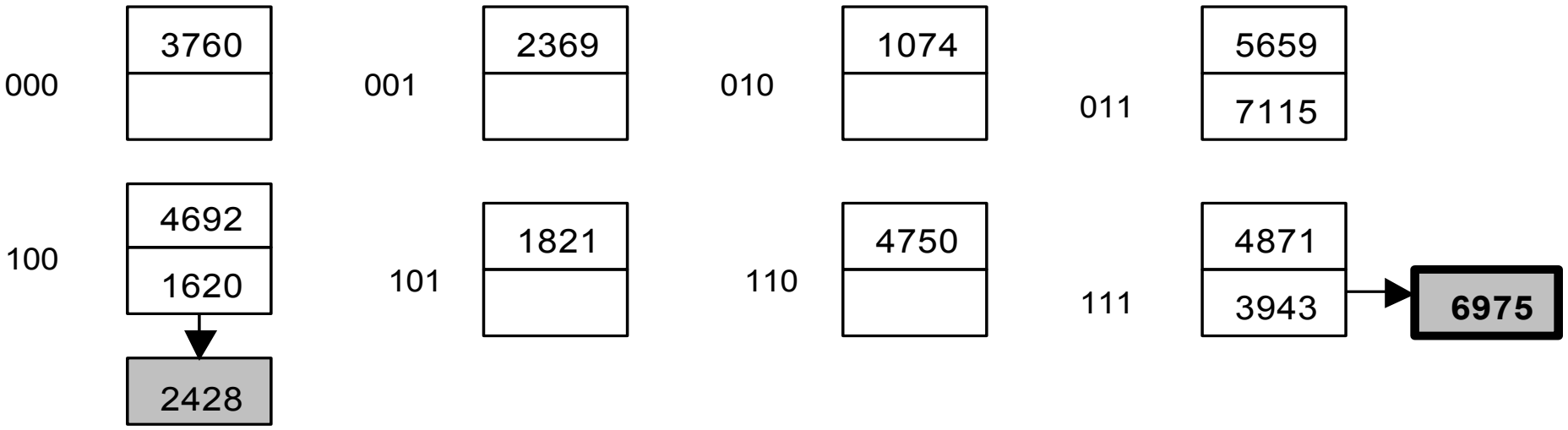
$h2(K) = K \bmod 2^2$
$h3(K) = K \bmod 2^3$
i=2, n=3



Linear Hashing (LH): Second Example

13b - Insert 6975

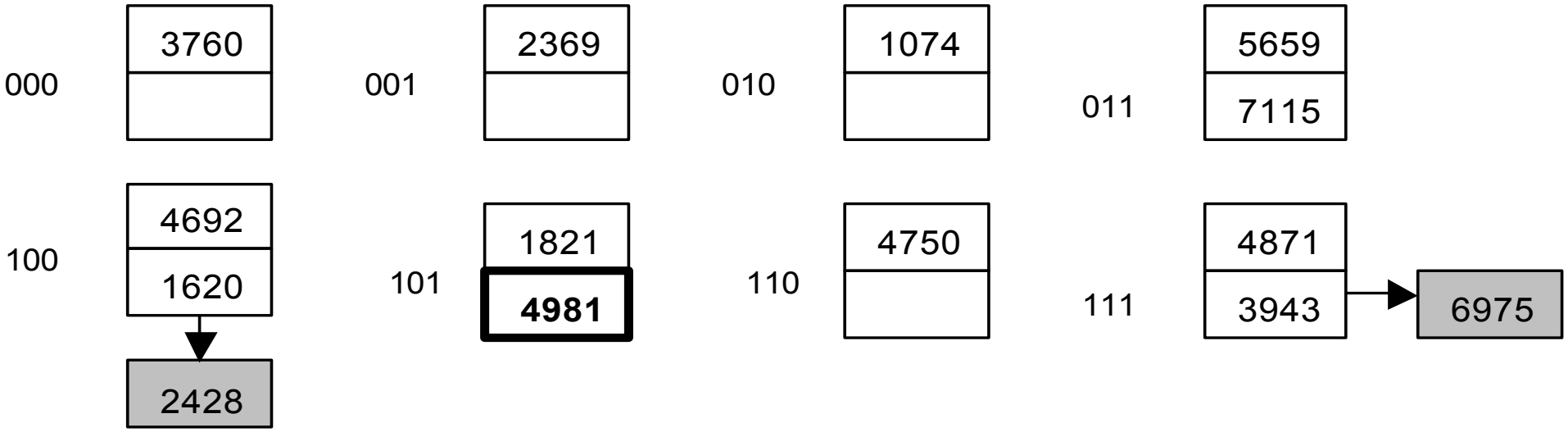
$h3(K) = K \bmod 2^3$
$h4(K) = K \bmod 2^4$
i=3, n=0



Linear Hashing (LH): Second Example

14 - Insert 4981

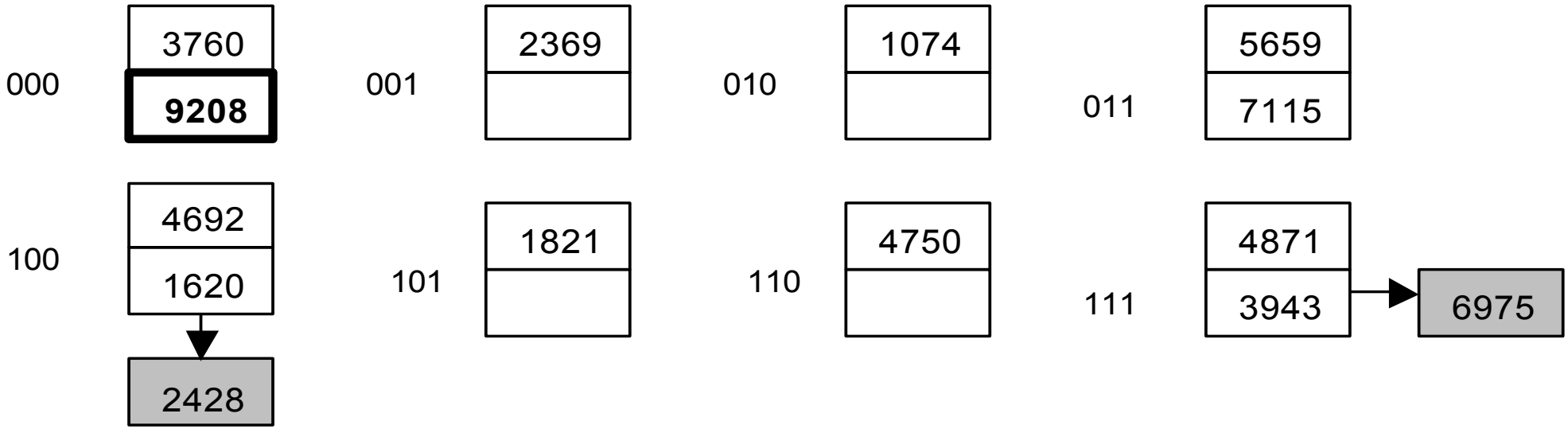
$h3(K) = K \bmod 2^3$
$h4(K) = K \bmod 2^4$
i=3, n=0



Linear Hashing (LH): Second Example

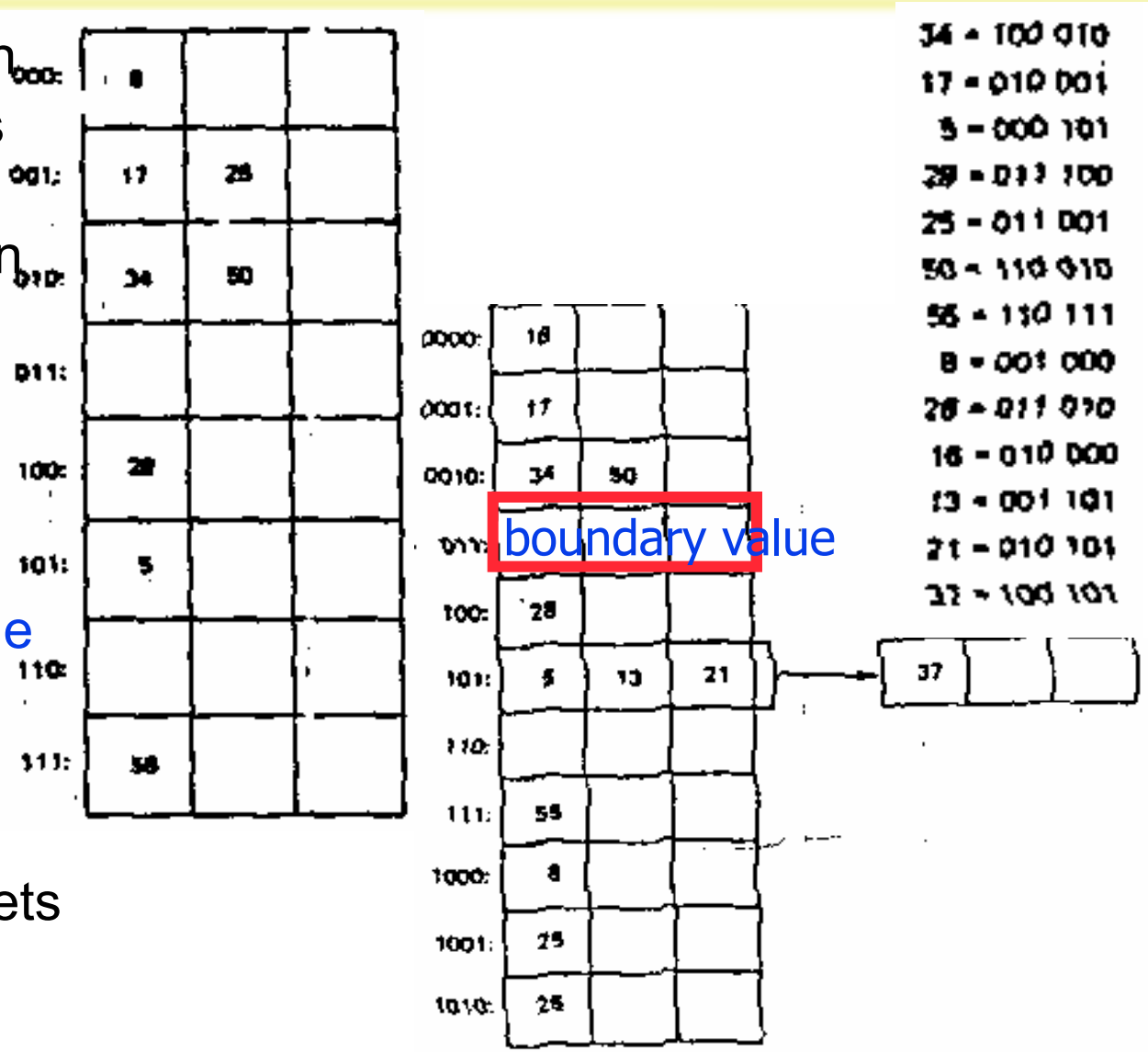
15 - Insert 9208

$h3(K) = K \bmod 2^3$
$h4(K) = K \bmod 2^4$
i=3, n=0



Linear Hashing (LH): Third Example (Alter. Algo)

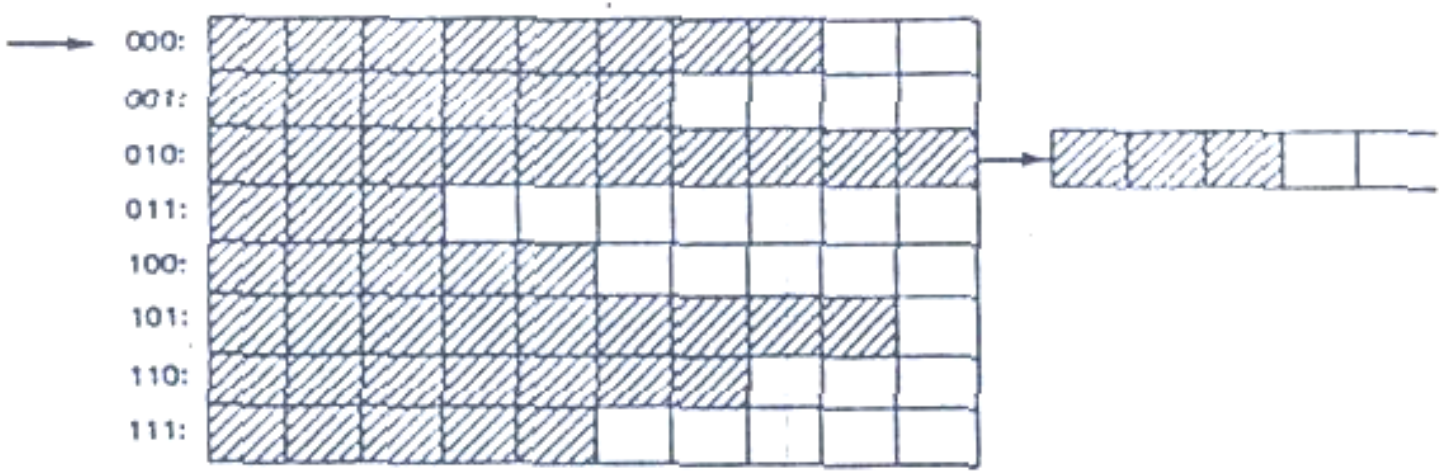
- Calculate hash function
- Look at the last k digits of the hash function
  - If k digit is less than boundary value, then k+1
  - Follow overflow chains, as with traditional hashing
- If k=3 & boundary value is 011
- 1000=8; use 4 digits
- 1101=13; use 3 digits
- Comment: Some buckets used twice as others; fetch time decreases



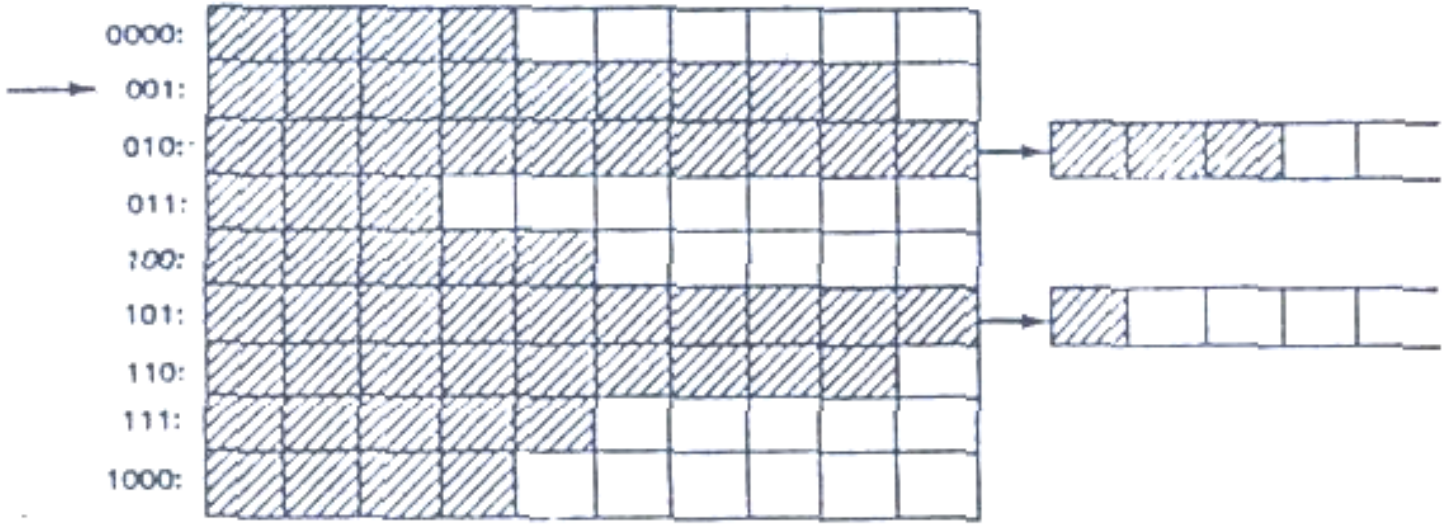
LH Third Example: Insertion and Load Factor

- Search for the correct bucket in which to place
  - If the bucket is full, allocate a new overflow bucket, address it & insert record
  - If there are Lf * c records more than needed add one more bucket to the primary area
  - Distribute the records relating to new bucket
  - For each insertion, add 1 to the boundary value
- Example:
  - Initially 8 primary areas
  - Load factor of 70 %
  - Bucket factor of 10
  - Initially 56 records
  - Add a bucket for each 0,7 * 10 = 7 record insertions

LH Third Example: Insertion and Load Factor



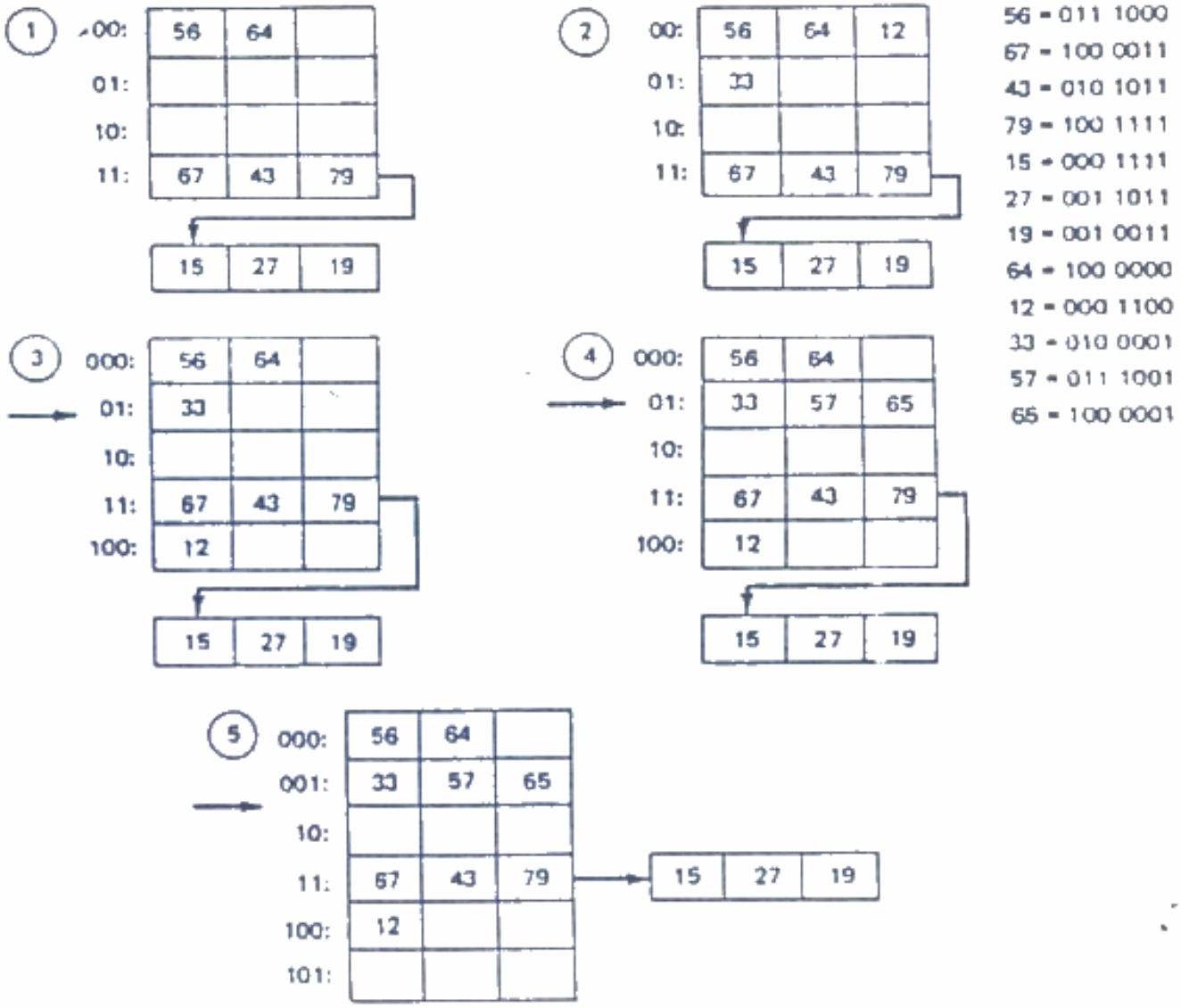
There are 58 records above. Now add 7 more records: 3 whose hash value ends in 001, 2 in 101 and 2 in 110. Then add one bucket to the primary area.



LH Third Example: Insertion and Load Factor

- Bucket factor is 3
- Load factor is 67 %
- Started with 4 buckets
- Make eight insertions
- See figure on the next slide:
- Overflow and table expansions are independent events
- Overflow doesn't trigger table expansion. Neither does the opposite!
- Last words:
- `Ti= Tf(unsuccessful) + 2r + (1/c * (s+r+dtt))`

LH Third Example: Insertion and Load Factor



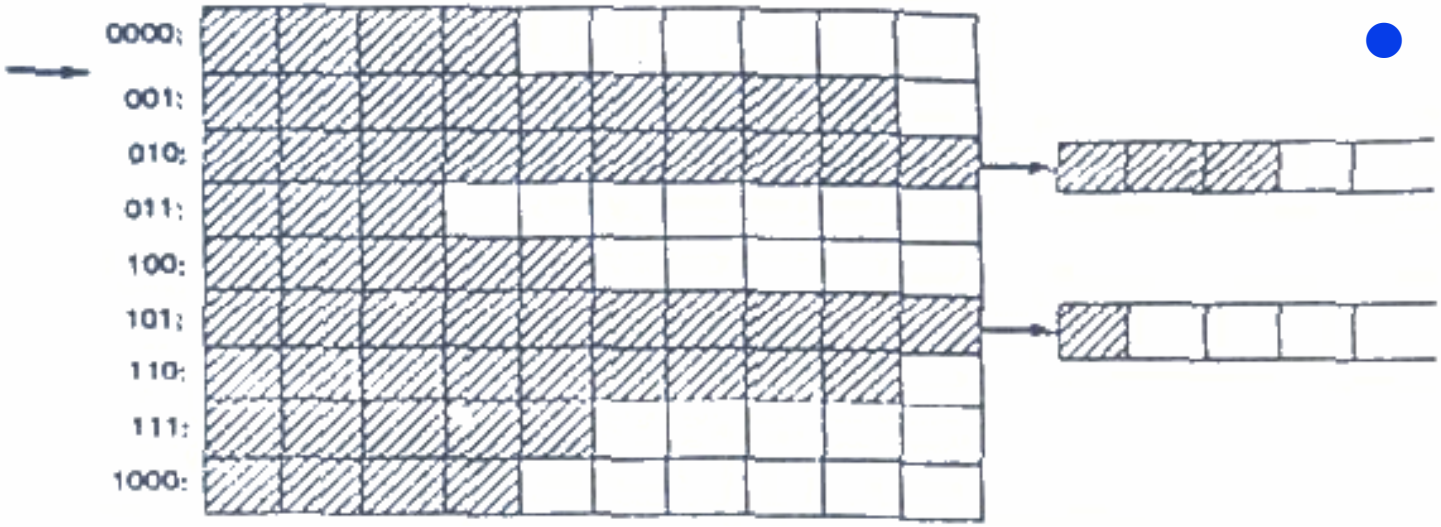
- A sample problem in linear hashing: We begin with 8 records in 4 primary area buckets
- The load factor is 67% (2/3), and the bucket factor is 3
- After the initial loading, whenever we insert 2 records, we expand the table by 1 bucket

LH Third Example: Insertion and Load Factor

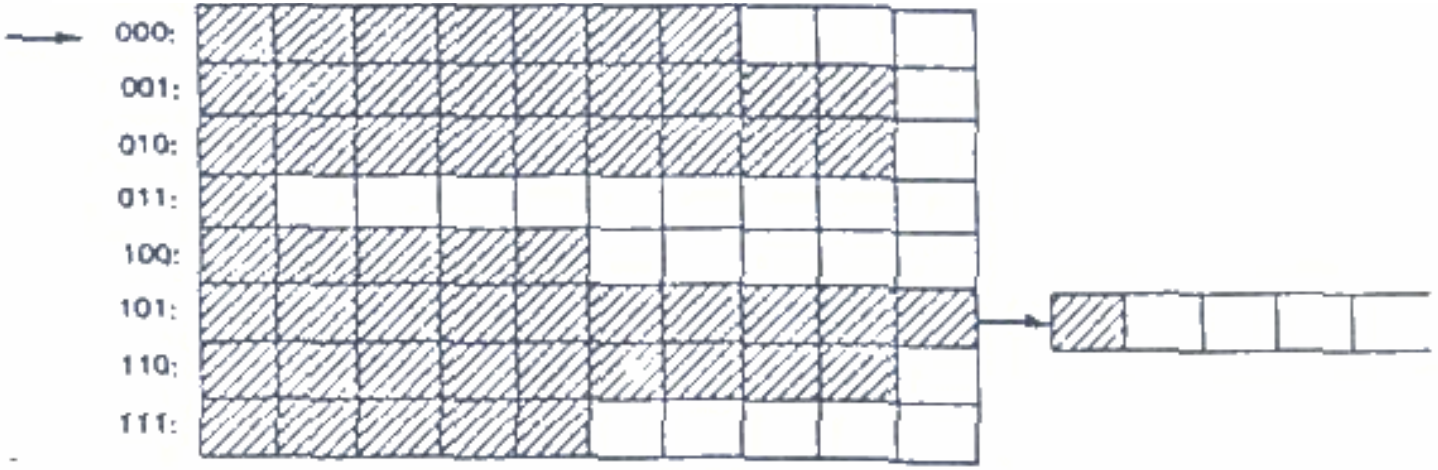
- Replace the deleted record with the last record in the chain
- If the last overflow is empty, de-allocate it
- When `c * Lf` is less than # needed, decrement primary area by one bucket

- Exactly opposite of Insertion
- Occasionally, last bucket in the chain is de-allocated
- `Td = Tf(unsuccessful) + 2r`

LH Third Example: Deletion and Load Factor



Now delete 4 records from 010, deallocating the chain. Also delete 2 records from 011 to 1 from 1000. Then contract the table.



- Linear hashing table contraction: To keep the load factor constant, the hash table can be contracted by 1 bucket every time there are Lf*c records less than are needed for the fixed load factor desired. On the right, where the load factor is 70% and the bucket factor is 10, a bucket is removed from the primary area whenever there are 7 records less than needed for a Lf of 70%

Constructing a Hash Table for Given Data

- There exists a file, we want to set up hashing
- Construction time is high:
- `R * (r + s + dtt + 2r )`
  - One access fetch and a modify for each record
- Option 1:
  - Read in the file and calculate hash values for each record
  - Sort by hash value
- Option 2:
  - Partition the space of hash values
  - Load some partition of the file, do hashing, write back disk…

Hash-Based Indexing Summary

- 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)
  - If directory fits in memory, equality search answered with one disk access; else two
  - Directory grows in spurts, can become large for skewed hash distributions
    - Collisions, or multiple entries with same hash value cause problems!
- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages
  - Overflow pages not likely to be long
  - Duplicates handled easily
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on “dense” data areas
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization
- For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed!

Hash-Based Indexing Summary

- Hashing is the best file structure for single-record fetches (e.g., ATM, airline reservations):
  - Insertion & deletions are very fast
  - There are more insertions than deletions
  - Reorganization schemes: Linear & Extendible Hashing
- Linear Hashing:
  - As new records are added, new buckets are appended to the primary area
  - Load factor is kept constant
- Extendible Hashing:
  - After index reaches a predetermined size, primary area buckets double in size
  - First few digits of the hash number give the location of the index entry in memory
  - Next few determine the address of the data bucket
  - Subsequent digits yield the offset of the block

Hash-Based Indexing Summary

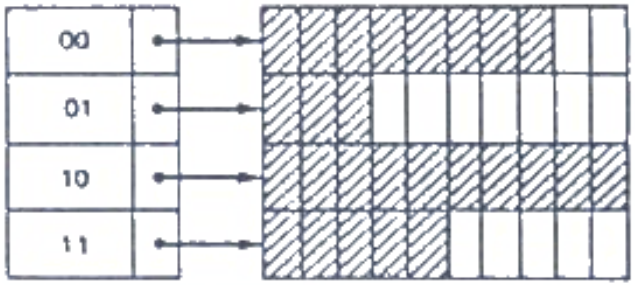
- No hashing method is convenient for sequential operations or for range searches
  - B+ trees are recommended especially for range searches

- Formulas:
  - Load factor is calculated by: `Lf = R / (B * c)`
  - Single record fetch without overflow: `Tf = s + r + dtt`
  - If there are overflows with an average # of chains x:
    - `Tf(successful)   = ( 1 + x/2 ) * ( s + r + dtt )`
    - `Tf(unsuccessful) = ( 1 + x )   * ( s + r + dtt )`
  - For insertion and deletion we use
    - `Ti = Td = Tf(unsuccessful) + 2r`

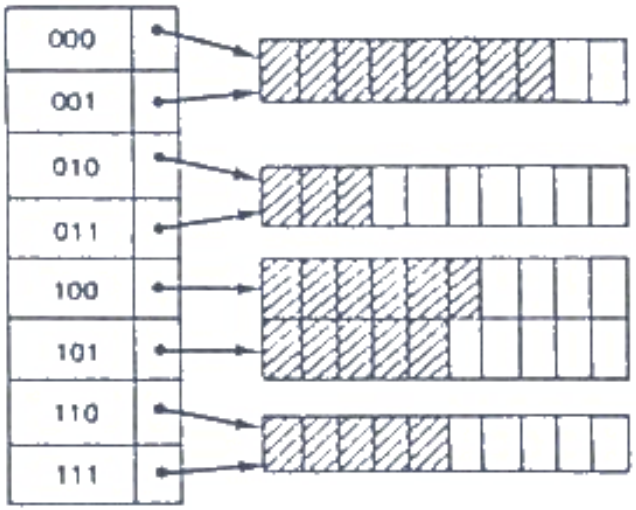
Overall Indexing Summary

- Index properties
  - Integrated vs. Separate Storage Structure
  - Clustered vs. Unclustered, Dense vs. Sparse, Primary vs. Secondary
  - Multi-level Indices
  - Handle duplicate values differently depending on the index characteristics
- Tree-structured indexes – B+ trees
  - Dynamic structure
  - Inserts/deletes leave tree height-balanced, cost = $\log_F N$
  - High fan-out (F), depth rarely more than 3 or 4
- Hash-based indexes
  - Static Hashing
    - Long overflow chains
  - Extendible Hashing
    - Directory to keep track of buckets, doubles periodically
    - Avoids overflow pages by splitting a full bucket when a new data entry is to be added to it
  - Linear Hashing
    - Avoids directory by splitting buckets round-robin, and using overflow pages

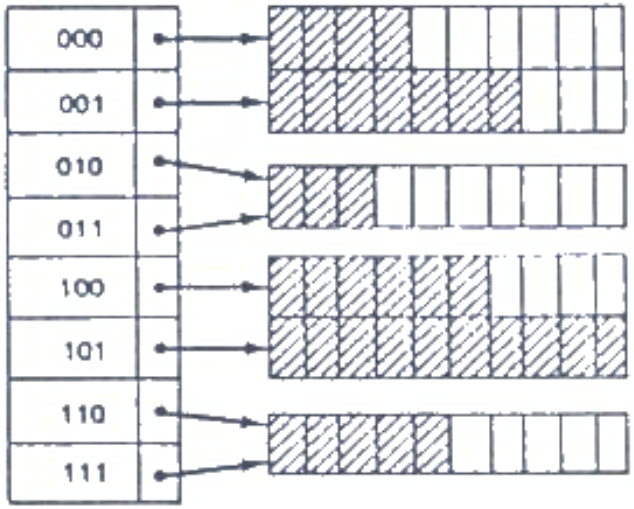
Extendible Hashing



(a) Overflow in bucket 10
table doubles; add one bucket.



(b) Add three records to the first bucket; overflow does not expand table.



(c) Structure after second overflow.

Figure 6.9. Extendible hashing. When a bucket overflows, the index must sometimes be doubled. Sometimes, as in the second overflow, it is necessary only to allocate a new bucket and change an index entry.

Τέλος Ενότητας

BY NC ND

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Χρηματοδότηση

•Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

•Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

•Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημειώματα

Σημείωμα αδειοδότησης

•Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση, Όχι Παράγωγο Έργο 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

•Ως **Μη Εμπορική** ορίζεται η χρήση:

–που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο

–που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο

–που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

•Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

.

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Φροντιστήριο 4: Tutorial on Indexing part 2 - Hash-based Indexing**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/