ICOM 6005 – Database Management Systems Design

Dr. Manuel Rodríguez-Martínez
Electrical and Computer Engineering Department

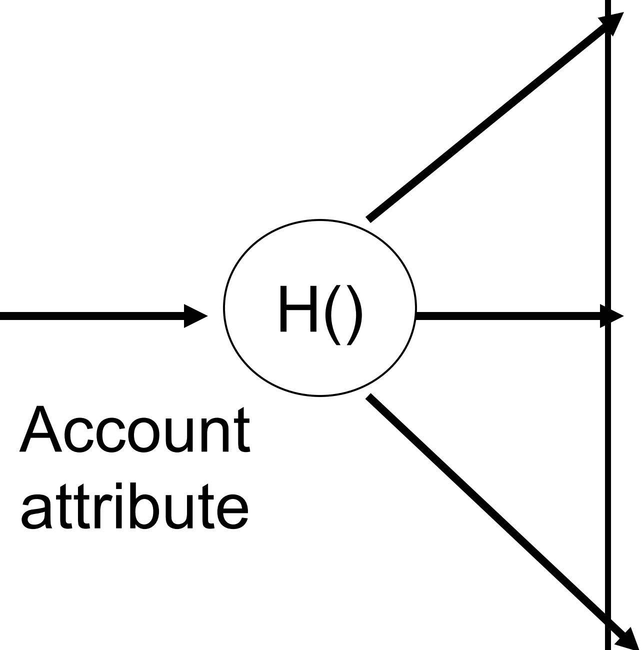
Hash-Based Indexing

- Read
  - New Book: Chapter 11
- Hash methods can be used for index files to support efficient searches by equality
  - Often require 1 to 2 I/O operation
- Three type of hashing schemes
  - Static Hashing
  - Extensible Hashing
  - Linear Hashing
- In practice, commercial DBMS use hashing indexing for temporary calculations
  - Aggregation and joins
  - Tree-based indices are use as actual indices on relations

General hashing approach

- Hash File for a relation R has N pages
  - Each page is called a bucket
  - Buckets are numbered from 0 to N – 1
  - If a bucket gets full, an overflow page must be chained to it
- Each record t in R has a search k
  - Can be made out of 1 or more attributes
  - Ex. Studens(sid, name, login, age, gpa)
    - Search key: age attribute
- A hash function is used to map the search key k of a record t in R to bucket number [0, N-1]
  - Hash function should distribute records uniformly
  - Record is searched inside the bucket

Hash Index (clustered)

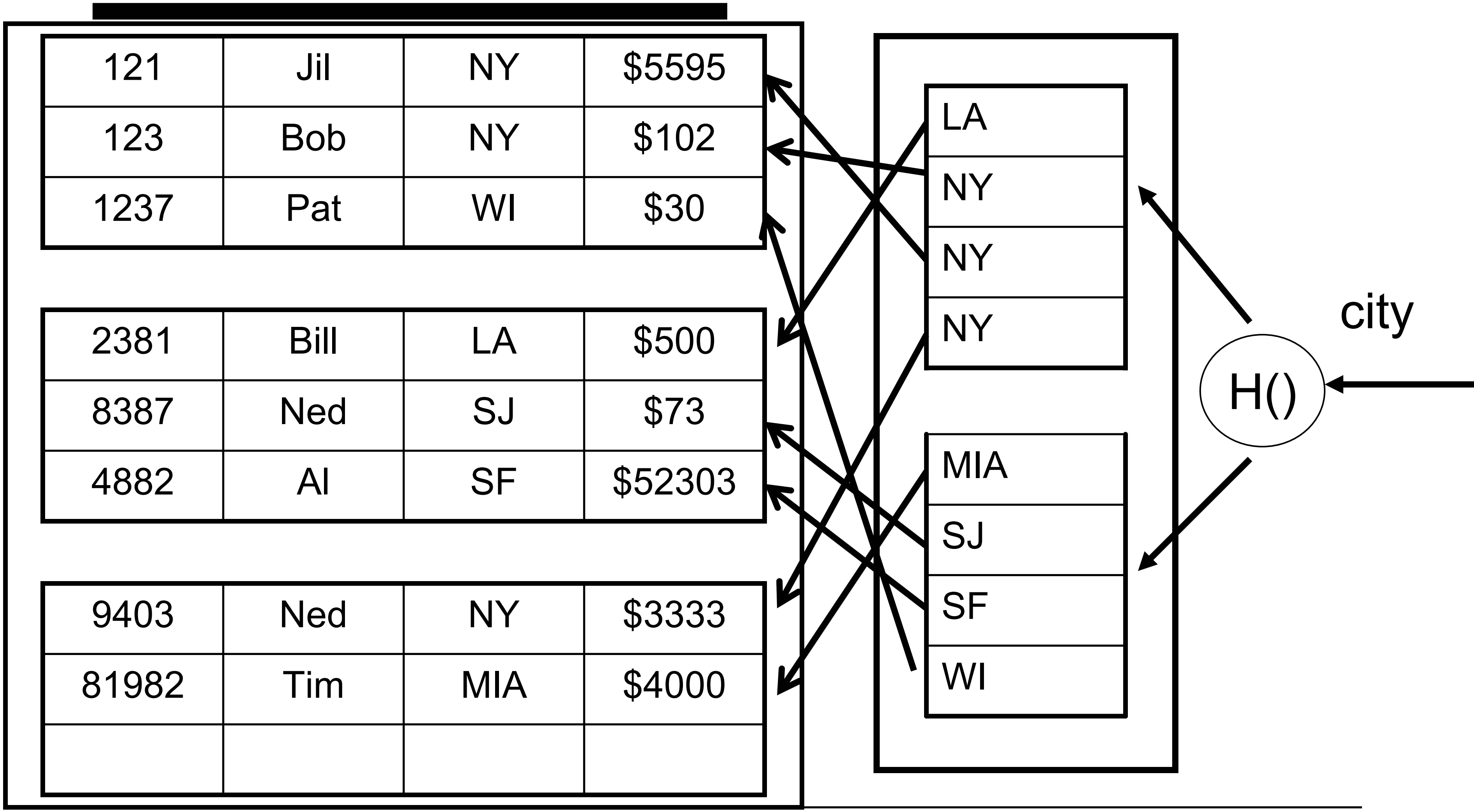


121	Jil	NY	$5595
123	Bob	NY	$102
1237	Pat	WI	$30

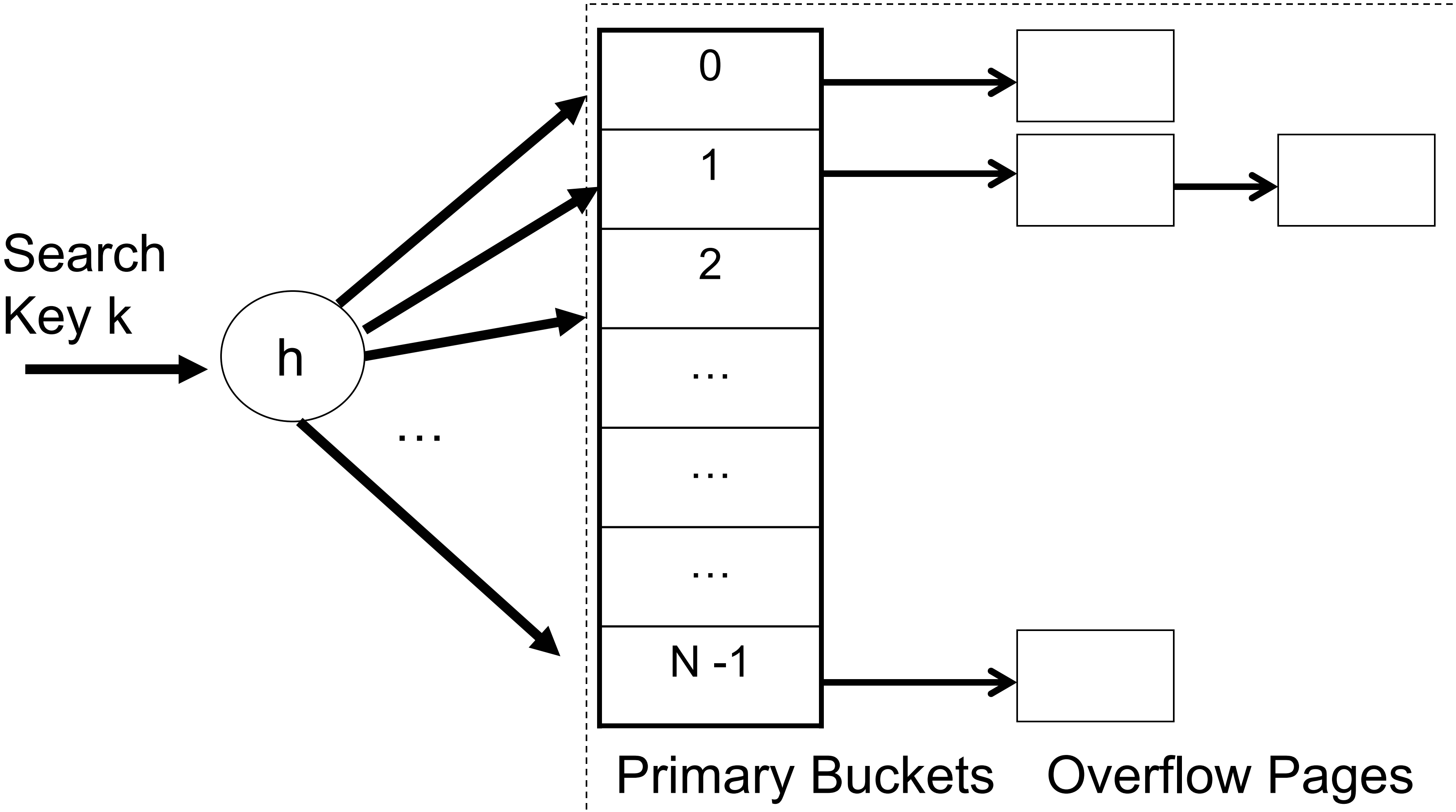
2381	Bill	LA	$500
8387	Ned	SJ	$73
4882	Al	SF	$52303

9403	Ned	NY	$3333
81982	Tim	MIA	$4000

Hash Index (Unclustered)



Static Hashing: Scheme



Static Hashing: Issues

- Number of primary buckets is **fixed** at file creation
- Hash function maps key to a bucket number
- Typical hash function
  - H(k) = a*k + b
  - Bucket number = h(k) mod N
- Key can be int or char
  - Char – each character is mapped to ASCII, and all value are added to get an integer
  - Parameters a and b are choose to tune the distribution of values (i.e. need to play with this values to get them right …)
- When a primary bucket get full, need to create an overflow page and chain it to primary bucket.

Static hashing: Operations

- **Search** for key value k:
  - Hash k to find the bucket, call this bucket B
  - Search records in B to kind the one(s) with key k
  - If records are found
    - **clustered**, the data record is there: Cost: 1 I/O
    - **unclustered**, need to fetch the actual data page: Cost 2 I/Os
  - If records are not found, need to search in overflow pages (if there are any)
    - **Clustered**: Cost: (1 + number of pages searched) * I/O
    - **Unclustered**: Cost: (2 + number of pages searched) * I/O
- The more overflow page you have, the worst the performance get
  - Need to keep overflow pages to 1 or 2, but rarely gets done!

Static Hashing: Operations (cont.)

- Insert (or Update) record with key k
  - Hash k to find bucket, call this bucket B
  - If bucket has room
    - **clustered**, write data record there: Cost: 2 I/Os
      - Read page, then write it back updated
    - **unclustered**, write record to actual data page: Cost 4 I/Os
  - If bucket is full, write to overflow page (create one if needed)
    - **Clustered**: Cost: (2 + number of pages searched) * I/O
    - **Unclustered**: Cost: (4 + number of pages searched) * I/O
- Delete costs are the same, since we need to write page back to disk
- Again, overflow pages make performance bad as the number of records increases

Extensible hashing

- Allows the number of buckets to grow or shrink
- Hash function hashes to slots in a directory
  - Slots store the page id of the bucket
  - Directory can be kept in buffer pool
  - Directory can have hundreds or thousand of slots to buckets
- When a bucket gets full
  - Create a new bucket and split records between the new and full bucket
    - Redistributes the data
    - Hash function still works!!!
  - Overflow page is need only if you have many duplicate records

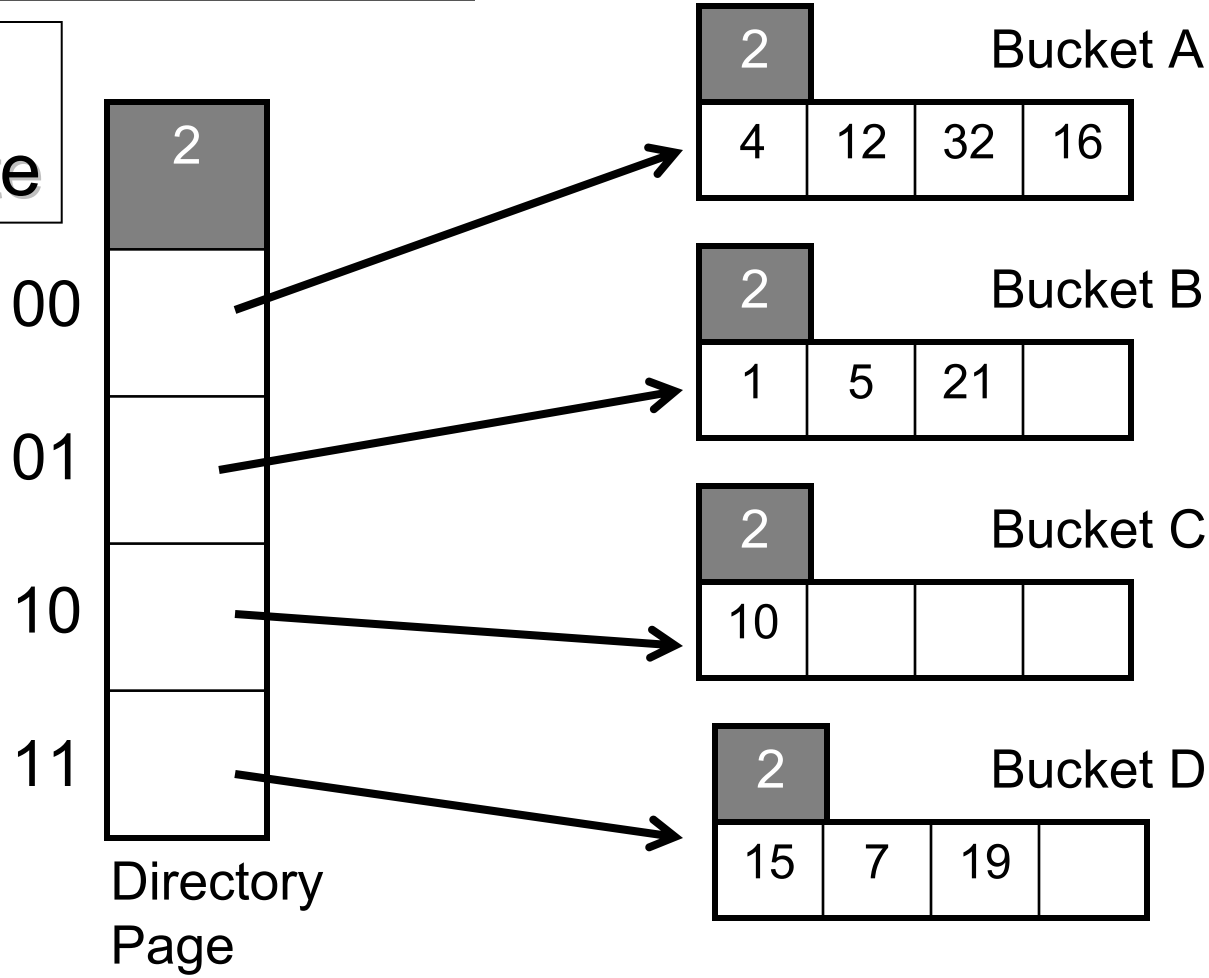
Binary Pattern Hashing Technique

- Hash function will map search key to binary pattern
  - Ex h(3.40) = 00110011
- Last *d* bits in the pattern are taken as bucket number!
  - Ex. If *d* = 2, then h(3.40) = 00110011 will yield bucket number 11, which is 4 in binary
    - Thus, 3.40 goes to bucket 4
- The number of *d* of bits used to hash the search key is called the depth
- Two types of depths
  - Bucket depth
    - Number of bits need to hash value to a given bucket
  - File depth
    - Largest depth of any bucket

Example Extensible Hashing Index

Hashing on an int attribute

H(4) = 100
d = 2
gives slot 00.
For value 4
Bucket is
then found
from the slot.

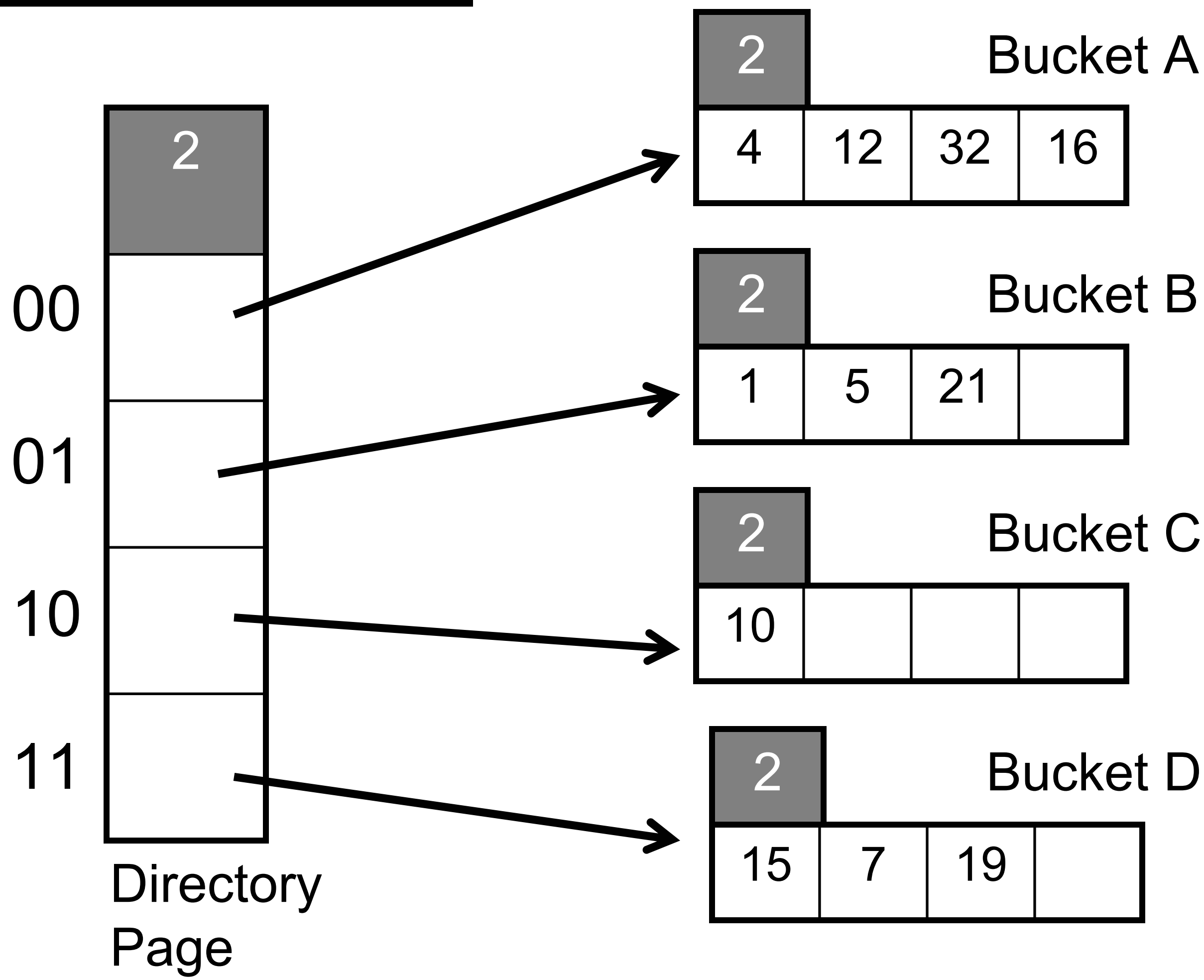


The use of the depth

- Depth tells us the number of bits that we need to use to pick a bucket
  - Ex. H(4) = 100, d = 2, tell us to use 00 to identify slot. This would be slot 00.
- Directory has a global depth
  - Used to hash key to proper slot
- Each bucket has a local depth
  - Used when bucket need to be slipt
- Let us see what happens when we need to insert the value 20 into the hash index
  - H(20) = 10100, d =2

The issue of a full bucket

H(20) = 10100
d = 2
gives slot 00.
This bucket is
Full



Splitting a bucket

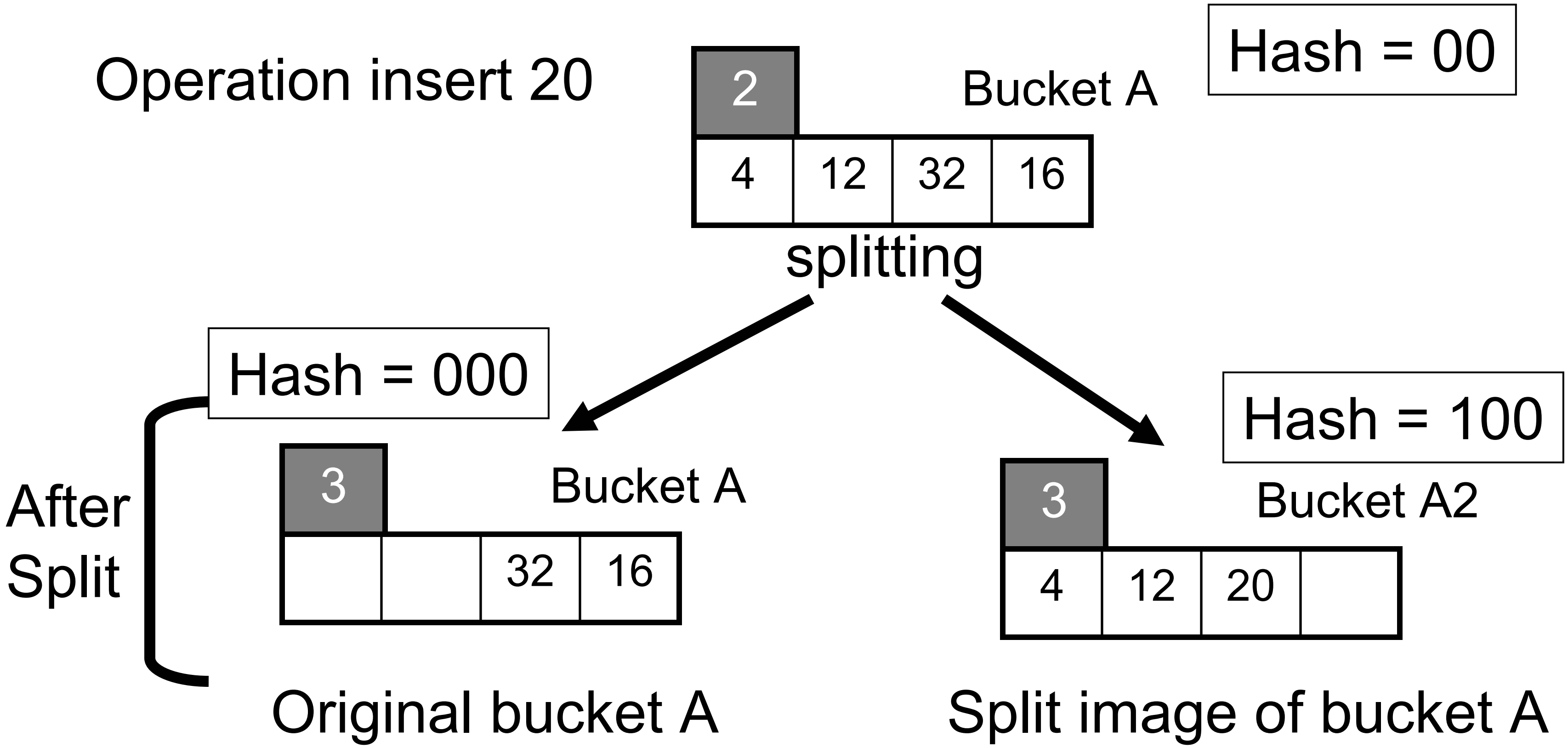
- A full bucket gets split into two buckets
  - Their directory slots are called corresponding elements
- These buckets have the same hash value at the current depth *d*
- But at depth *d* + 1, they differ by 1 bit
  - one has a 1 at bit position *d* + 1
  - The other has a 0 at bit position *d* + 1
- Example:
  - Bucket A is splint into two buckets: bucket A and bucket A2
  - Bucket A , *d* = 2, has value 00, but at *d* = 3 becomes 000
  - Bucket A2, *d* = 2, has value 00, but at *d* = 3 becomes 100

Splitting a bucket (cont.)

- The values in the original bucket A and the new value to be inserted get distributed into buckets A and A2.
- The hash function now increment the local depth of the bucket to be $d + 1$
- Now, the keys are hashed to buckets using $d + 1$ bits
- Recall that bucket A had: 4, 12, 32, 16, and $d = 2$
  – We wanted to insert 20
- Now d becomes 3, we get the following hashing:
  – 4 = 100 H(4) = 100
  – 12 = 1100 H(12) = 100
  – 32 = 100000 H(32) = 000
  – 16 = 10000 H(16) = 000
  – 20 = 10100 H(20) = 100

Corresponding elements & buckets

Operation insert 20

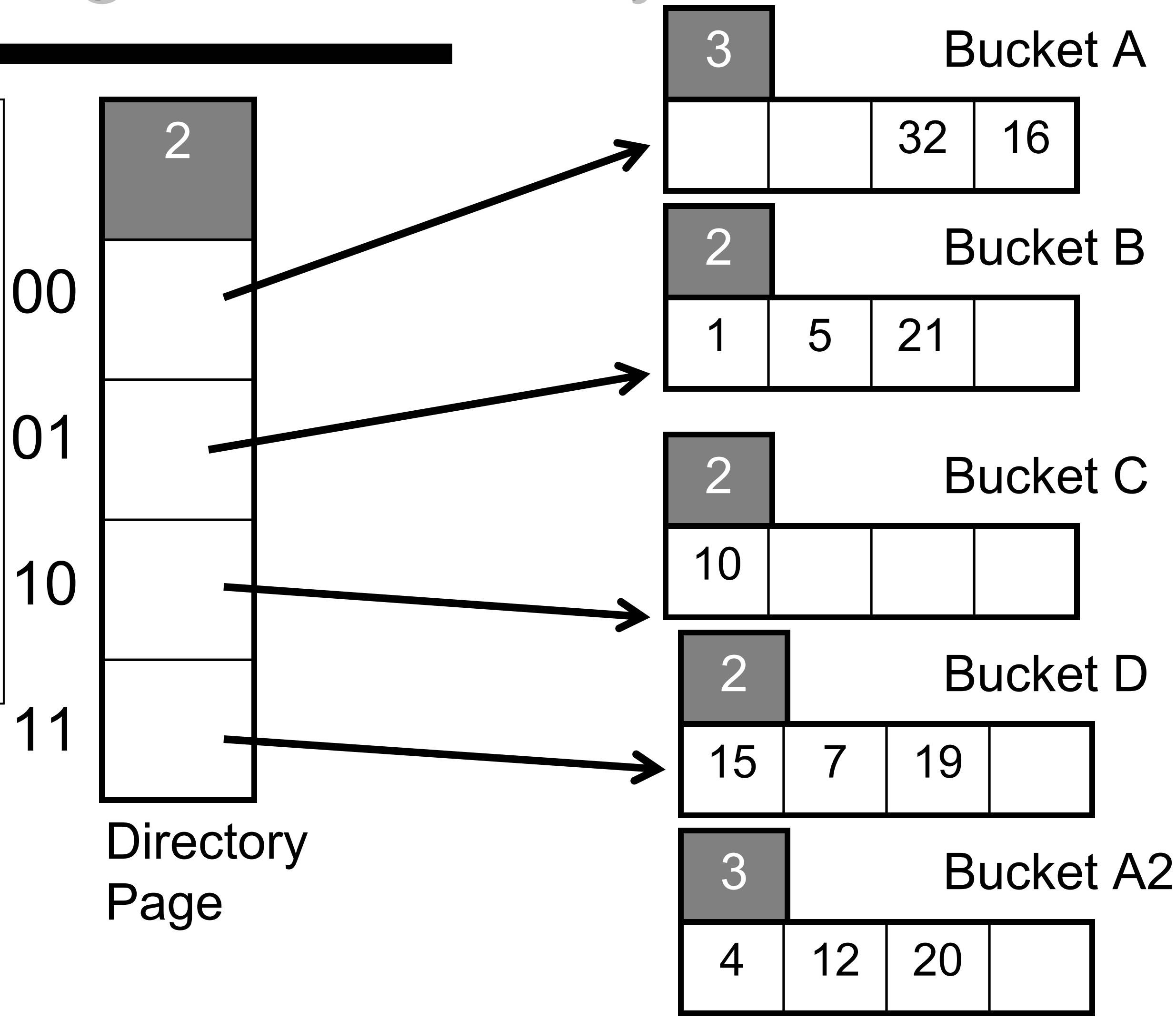


Local depth is changed from 2 to 3
Need 3 bits for hashing

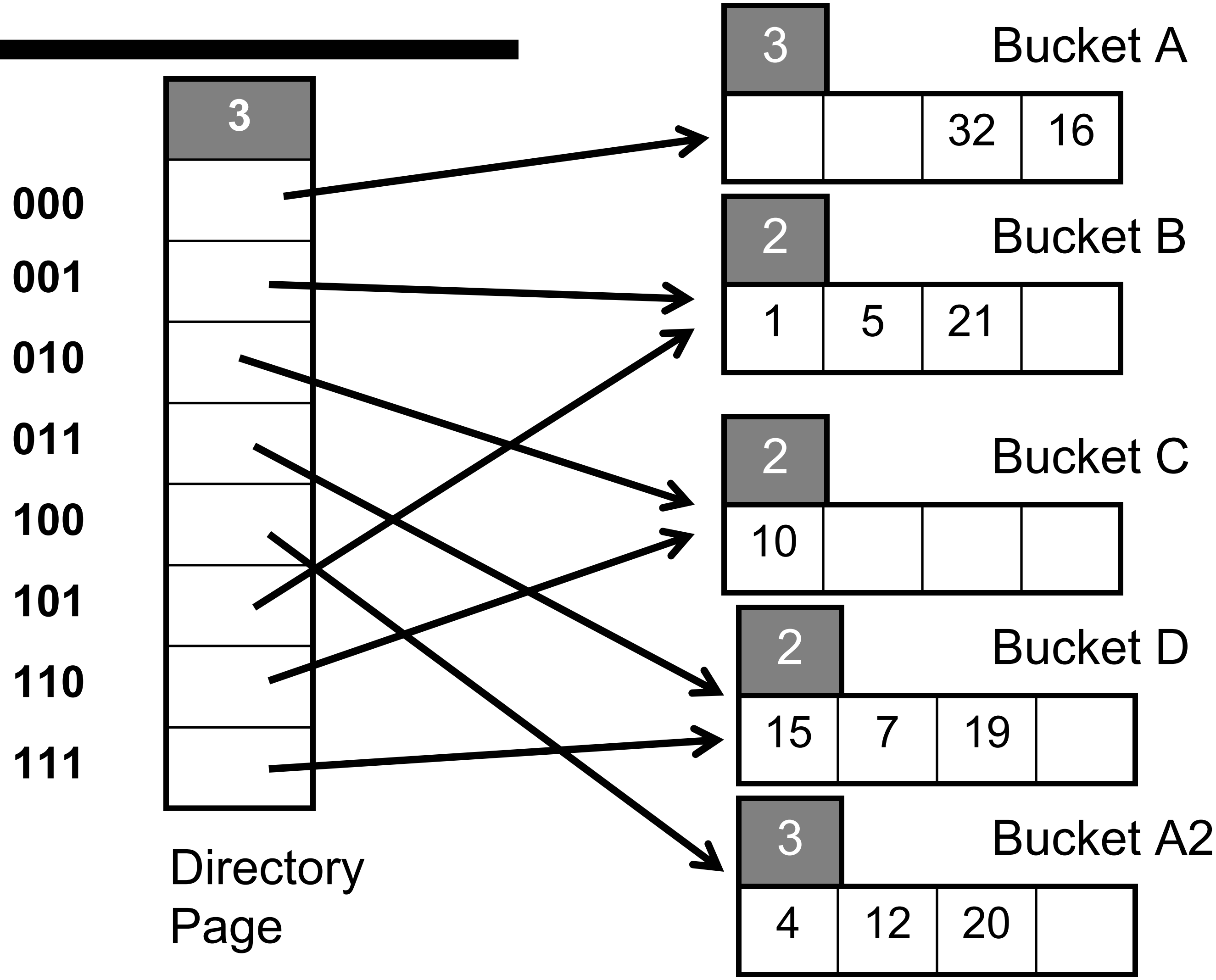
Expanding the Directory

Number of slots is not enough.

Need to double size of the directory and increase Global depth



Expanded Hash Index



Some Issues

- Some corresponding elements point to the same bucket
  – This means the bucket has not been split
- Not all splits operations cause the directory to be double.
- Each bucket has a local depth
  – If depth of bucket = global depth – 1, then splitting this bucket will not cause a doubling in directory
- Doubling only occurs when
  – Bucket is full and cannot fit another insertion
  – Bucket has same local depth as global depth

Cost estimates for operations

- Assume directory is in buffer pool
- Search for equality
  - Clustered – 1 I/O
  - Un-clustered – 2 I/O
- Erase
  - Clustered – 2 I/Os
  - Unclustered – 4 I/Os
- Insert (No splitting)
  - Clustered – 2 I/Os
  - Unclustered – 4 I/Os
- Insert (Splitting)
  - Clustered – 4 I/Os
  - Unclustered – 8 I/Os
- Overflow pages will be needed when you have lots of values with the same search key k

Tradeoffs of Extensible Hashing

- Advantages
  - Can gracefully adapt to insertion and deletions
  - Limits the number of overflow pages
  - Hash function is easy to implement
    - No need for complex prime number computations
- Disadvantages
  - Directory can grow large when we have billions of records
  - Also, when we have skewed data distributions
    - Lots of values go to same bucket
    - Lots of empty buckets, a few one have all the data
    - Overflow pages due to **collisions** (values that hash to same bucket)
  - Too much doubly in the size of the directory

Linear Hashing

- Dynamic hashing technique
  - No need for directory
  - Limits overflow pages due to collisions
  - Splitting of buckets is done in a more lazy fashion
- Idea is to have a family of hash functions
  - $h_0, h_1, h_2, \ldots$
  - Each function has a range twice as big as the predecesor
  - If h_i maps to M buckets, h_{i+1} maps to *2M* buckets
  - This is used when more buckets are needed
    - Switch from current h_i to h_{i+1} if we need to grow number of buckets beyond current M (we double number of buckets to *2M*)

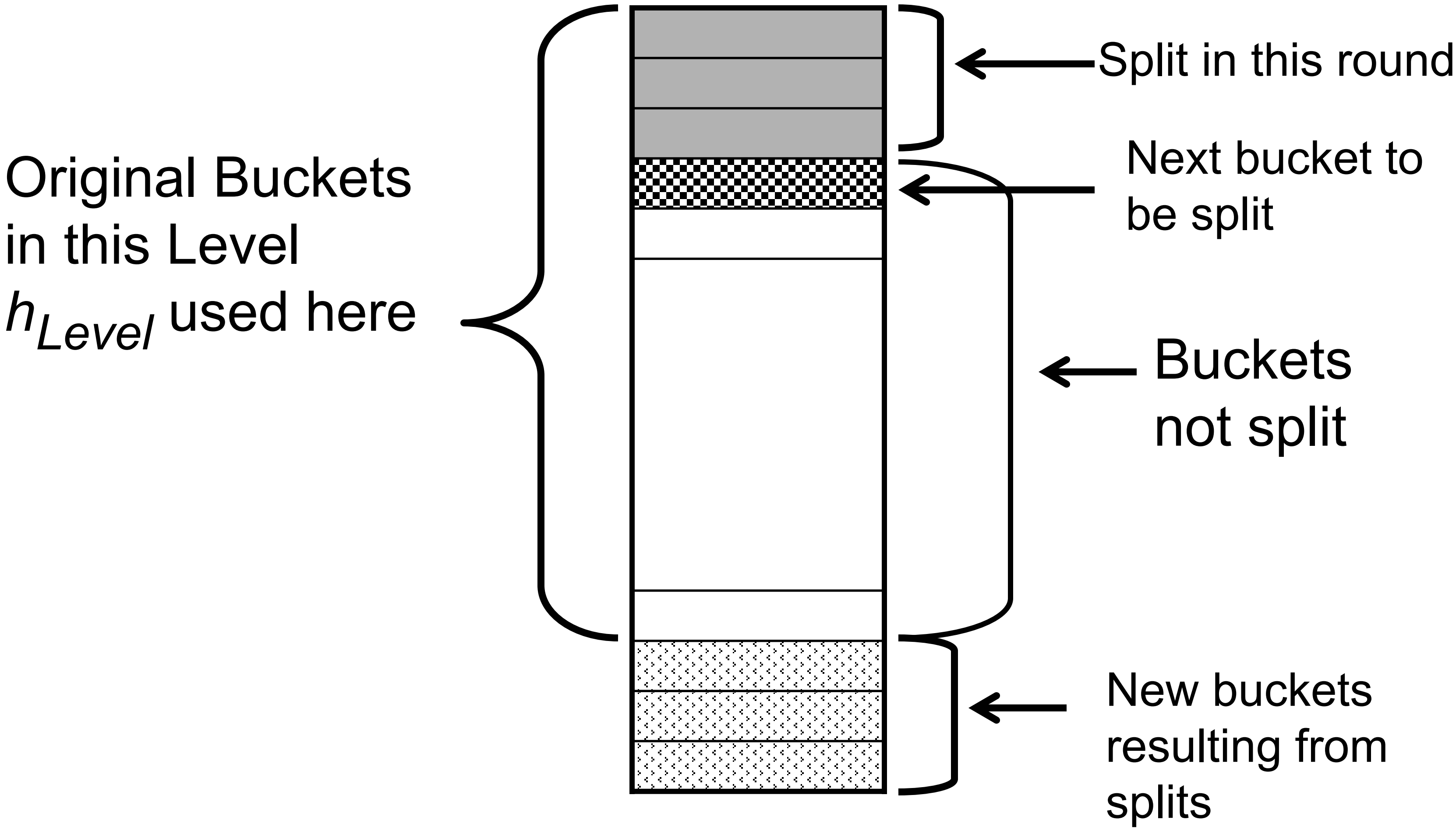
Building the family of hash functions

- General form is
  - $h_i\,(key) = h(key)\ mod\ (2^i N)$
  - h(key) acts as the base function
  - h(key) is the same as for extensible hashing
    - Looks as the bit pattern in the value
- If N is a power of 2, and d_0 is the number of bit to represent N, then d_i gives the number of bits used by function h_i
  - $d_i = d_0 + i$

General Scheme

- Hash index file as an associated round number
  - Called *Level*
- At round number *Level* we use hash functions
  - h_{Level} and $h_{Level + 1}$
- We keep track of the next bucket to be split
  - Buckets are split in a round robin fashion
    - Every bucket eventually gets splits …
- Index file has tree types of buckets
  - Buckets that were split in this round
  - Buckets that are yet to be split
  - Buckets created by splits in this round

Organization of Index Hash File



Example scenario

h1	h0					
000	00	32	44	36		Next=0
001	01	9	25	5		
010	10	14	18	10	30	
011	11	31	35	7	11	