

COURSE 10

Hash-Based Files

Hash files (Direct file organization)

- Hashing is a method of determining the address of a record based on the value of one of its fields (usually key value)
- **Ideal case:** function which calculates the address where each record is to be stored:
$$h: \{K_1, K_2, \dots, K_n\} \rightarrow A, h(K_i) = \text{memory add. of } i^{\text{th}} \text{ record}$$

(K_i is the key value of the i^{th} record, A is the set of disk addresses)
- Hard to define such functions:
 - all possible key values must be known from the beginning
 - for big files is almost impossible to maintain bijectivity

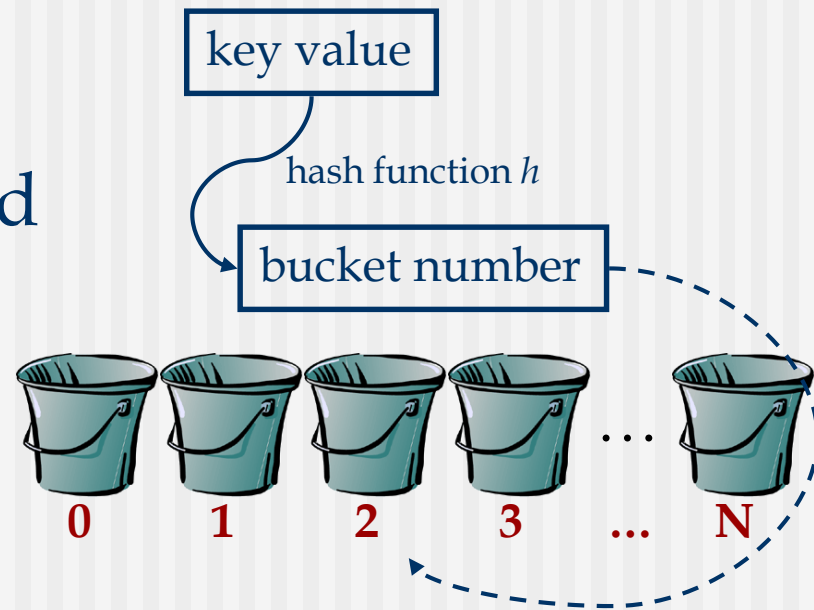
Buckets

- Solution - allow collisions:

$$h(K_i) = h(K_j), \text{ for } i \neq j \quad (h - \text{'hashing function'})$$

- Records with key values K_i and K_j are synonyms

- All synonym records are stored into a “*bucket*” which starts at the address returned by h . A “*bucket*” is a unit of storage containing one or more records (a bucket is typically a disk block).



Main Concern When Using Hashed Files

- **Distribution problem** - once we have chosen the hash algorithm, we have no control over the distribution of records in mass storage.
- **Clustering problem** - majority of records are placed in the same bucket and the rest of buckets contain almost no records.
- **Overflow problem** - unless the buckets are extremely large, overflow may occur.

Defining Hashing Functions

- Requirements for a 'good' hashing function:
 - fast evaluation
 - minimizes number of collisions (evenly dispersion of records over addresses space)
- Assume insert records into 41 buckets: the probability of placing the 1st record to an empty bucket is $41/41$, the 2nd is $40/41$, the 3rd is $39/41$ and so on. The probability of placing 8 records into 8 empty buckets is
$$(41/41)(40/41)(39/41)....(34/41) = 0.482$$

Less than 50%!!!

Defining Hashing Functions (cont)

- Approaches:
 - Division method
 - Mid-square method
 - Folding method
 - Multiplication method

- Typical hash functions perform computation on the internal *binary representation* of the search-key (for example, for a string search-key, the binary representations of all the characters in the string could be added and this sum could be the parameter of the hash function).

Defining Hashing Functions (cont)

■ Division method:

- Simply define $h(k) = k \bmod N$
- This guarantees the range of $h(k)$ to be $[0 \dots N-1]$
- Choosing $N = 2^d$ for some d , only the least d bits of k will be effectively considered
- **Prime** numbers were found to work best for N .

■ Mid-square method:

- Square the key and take some digits from the middle

■ Folding method:

- Divide the number in half and combine the two halves (e.g. add them together)

Defining Hashing Functions (cont)

■ Multiplication method

- Extract the fractional part of $Z * k$ (for a specific Z) and multiply by hash table size N (N is arbitrary here):

$$h(k) = \lfloor N * (Z * k - \lfloor Z * k \rfloor) \rfloor = \lfloor N * \{Z * k\} \rfloor$$

- Best results are obtained for

$$Z = (\sqrt{5} - 1)/2 = 0.61803... \text{ or } Z = (3 - \sqrt{5})/2 = 0.38196...$$

- For $Z = Z' / 2^w$ and $N = 2^d$ (w : number of bits in a CPU word)

$$h(k) = \text{msb}_d(Z' * k)$$

where $\text{msb}_d(x)$ denotes the d most significant bits of x !

(example: 42 has binary rep. 101010, $\text{msb}_3(42) = 5$ – bin. 101)

- Theorem. Given an irrational number x and placing $\{x\}, \{2x\}, \dots, \{nx\}$ on $[0,1]$ segment at most 3 different lengths of resulted $n+1$ segments are obtained. Also, the next value, $\{(n+1)x\}$, will be placed on one of the biggest segments

Example of Hashing Functions

- Key value 'Toyota'

- Concatenate together the alphabetic position of 1st and 2nd character: Toyota \Rightarrow

20	15
----	----

- Functions to hash-up the integer

- Division method: mod by 97 $\rightarrow 2015 \bmod 97 = 75$

- Mid-square method: $2015^2 = 4060225$ (take 2 middle dig)

- Folding method: $2015 \rightarrow 20+15 = 35$

- Multiplication method: $\lfloor 99 * \{2015 * 0.61803\} \rfloor = 32$

- Why not use 2015 as hashed value?

- 4 digits \rightarrow 10000 possible values! \rightarrow table will be largely empty

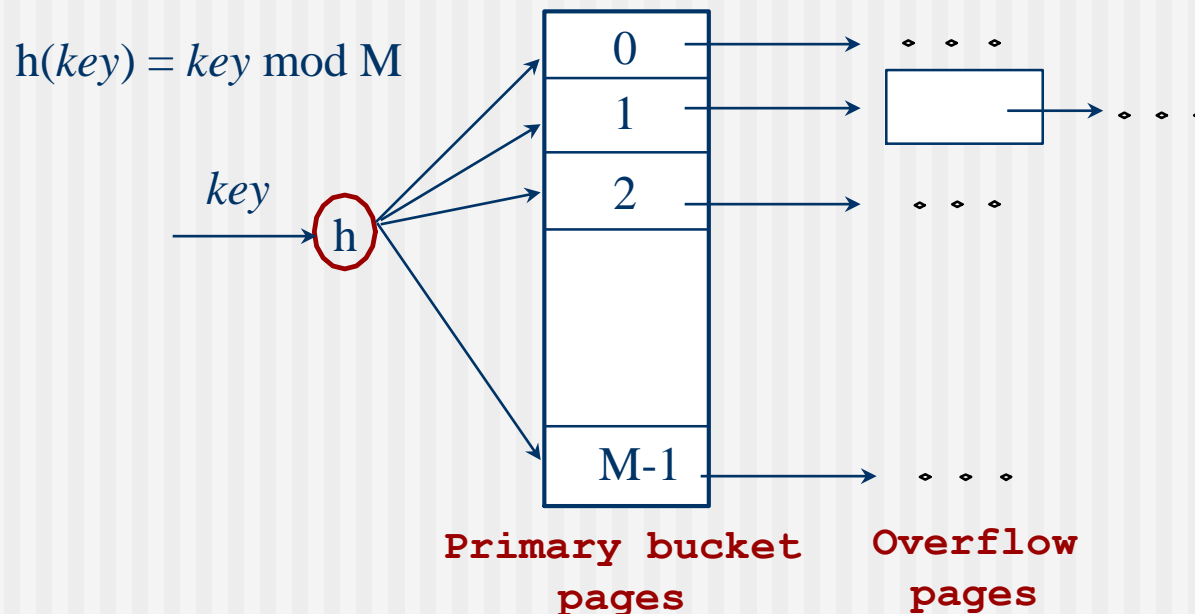
- our sample needs only 100 slots \rightarrow we might overflow it

Collision Handling

- Open addressing.
- Store new record in an unsorted overflow area.
- Apply a second hash function to the hash key to get a “second choice” address.
- Store pointers instead of records. At the hash address will be stored:
 - All the pointers of synonym records - **bucket** of addresses.
 - The pointer to the first record. As part of the first record, store a pointer to the second synonym record, and so on - **linked list** addresses.

Static Hashing

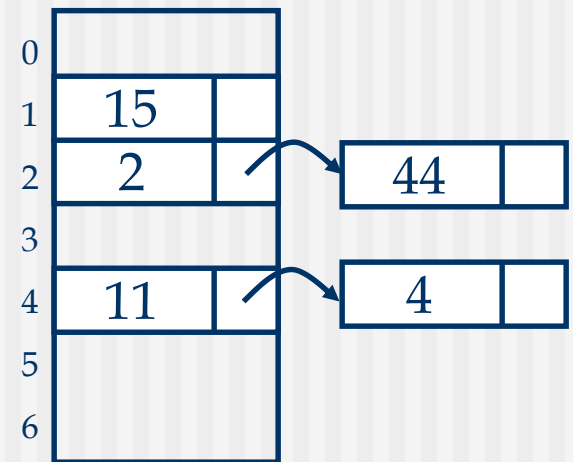
- Number of primary pages (or slots) is fixed, allocated sequentially, never de-allocated; some variants use overflow pages (slots) if needed.
- $h(k) = k \bmod M$ = bucket to which data entry with key k belongs. (M = number of buckets)



Static Hashing: Independent Lists

- All synonyms are stored into a specific linked list
- The hashed file contains a list of M data entries, each is the head of a list of synonyms
- The order of synonyms in list can be:
 - the order of insertions
 - decreasing order of searching frequency
 - increasing order of key value (searching without success stops earlier)

k	$h(k) = k \bmod 7$
11	4
2	2
44	2
4	4
15	1

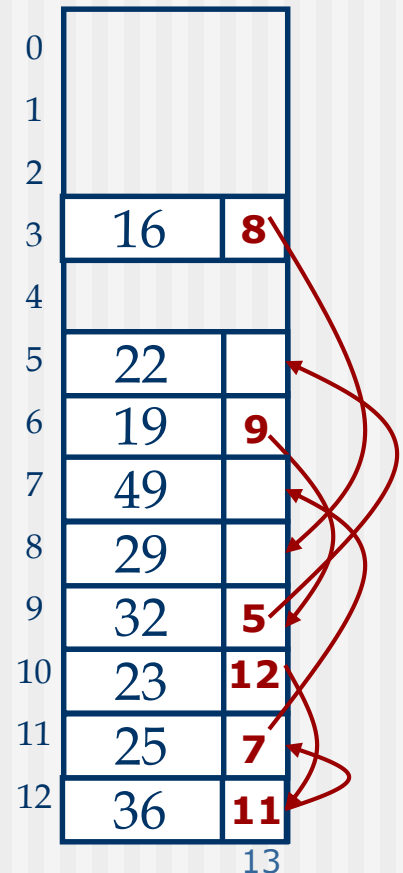


Static Hashing: Interleaved Lists

- Overflow pages (or slots) not used
- Insert a new record with key value K:
 - If the slot at $h(K)$ address is empty: store the record
 - If the slot where the record must be stored is full:
 - occupy the first empty slot (searching from bottom to top of file)
 - insert the occupied slot at the end of the list which contains the slot referred by $h(K)$

Example:

k	$h(k) = k \bmod 13$
16	3
23	10
36	10
25	12
19	6
32	6
29	3
49	10
22	9



Static Hashing: Interleaved Lists (cont)

- Deleting a record with key value K :
 - If the slot from $h(K)$ is empty: error message
 - If the slot from $h(K)$ is full:
 1. remove the record
 2. search, following the link, a record r with $h(K_r) = h(K)$
 - if such record is found
 - move it to current slot.
 - repeat step 2 for the new empty slot
 - otherwise, continue with step 3
 3. copy the link of the empty slot in the previous element in list (if exist)

Static Hashing: Interleaved Lists (cont)

- Example for deleting records: delete record with key value 23.

0		
1		
2		
3	16	8
4		
5	22	
6	19	9
7	49	
8	29	
9	32	5
10	36	12
11		
12	25	7

k	$h(k) = k \bmod 13$
16	3
23	10
36	10
25	12
19	6
32	6
29	3
49	10
22	9

Static Hashing: Open Addressing

- The file contains only data entries (no link slot)
- Insert a new record with key value K:
 - If the slot at $h(K)$ address is empty: store the record
 - If the slot where a record must be stored is full search for an empty slot at $h(K)+1, h(K)+2, \dots, M-1, 0, \dots, h(K)-1$ addresses.
- Good for 75% occupancy

Example:

k	$h(k) = k \bmod 13$
5	5
21	8
24	11
22	9
23	10
34	8
35	9

0	35
1	
2	
3	
4	
5	5
6	
7	
8	21
9	22
10	23
11	24
12	34

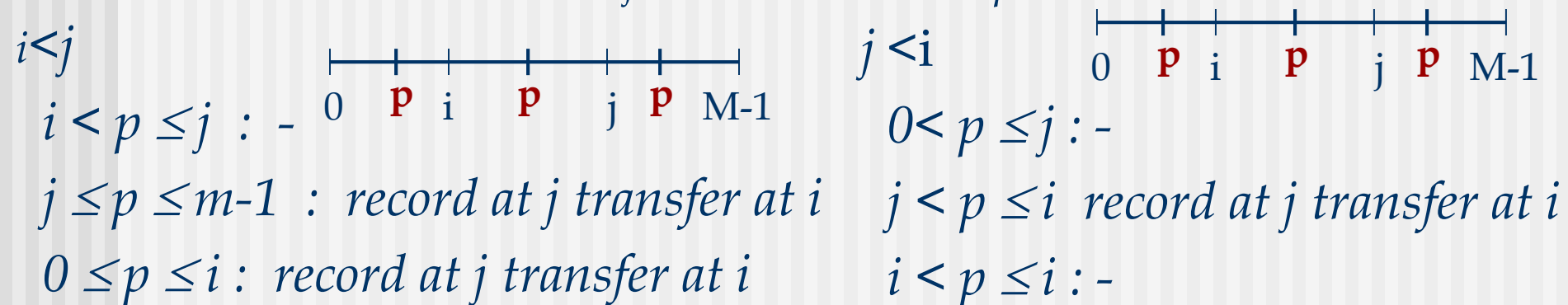
Static Hashing: Open Addressing (cont)

■ Deleting a record with key value K:

A. Replace the key value K with a special code or character. (searching and inserting algorithms must be adjusted in order to interpret special values)

B. Delete the desired record and switch remaining records. Consider i , j and p memory addresses so that:

- i is the address of deleted record,
- no empty slots between i and j
- the record stored at j must be stored at p



Conclusions on Static Hashing

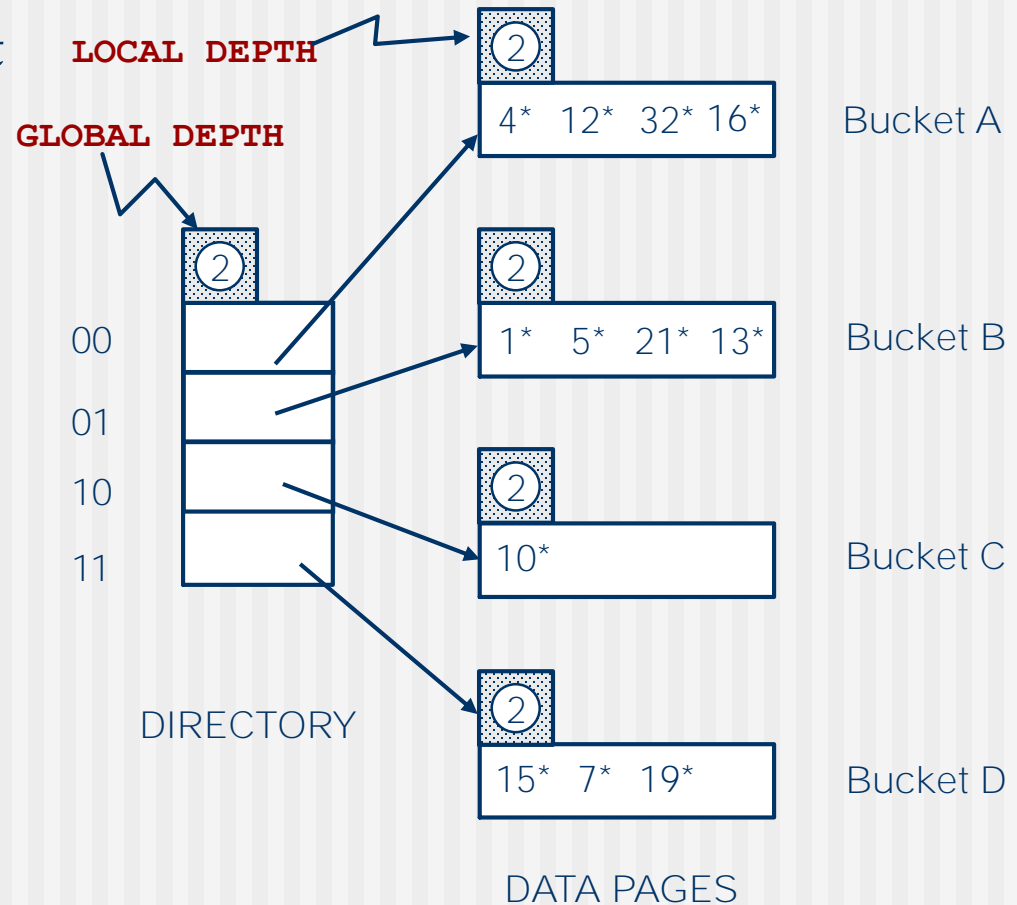
- Buckets contain *data entries*.
- Hash function works on *search key* field of record r . Must distribute values over range $0 \dots M-1$.
- If the underlying data file grows, the development of overflow chains spoils the otherwise predictable hash I/O behavior (1-2 I/Os).
- Similarly, if the file shrinks significantly, the static hash table may be a waste of space (data entry slots in the primary buckets remain unallocated).
- *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

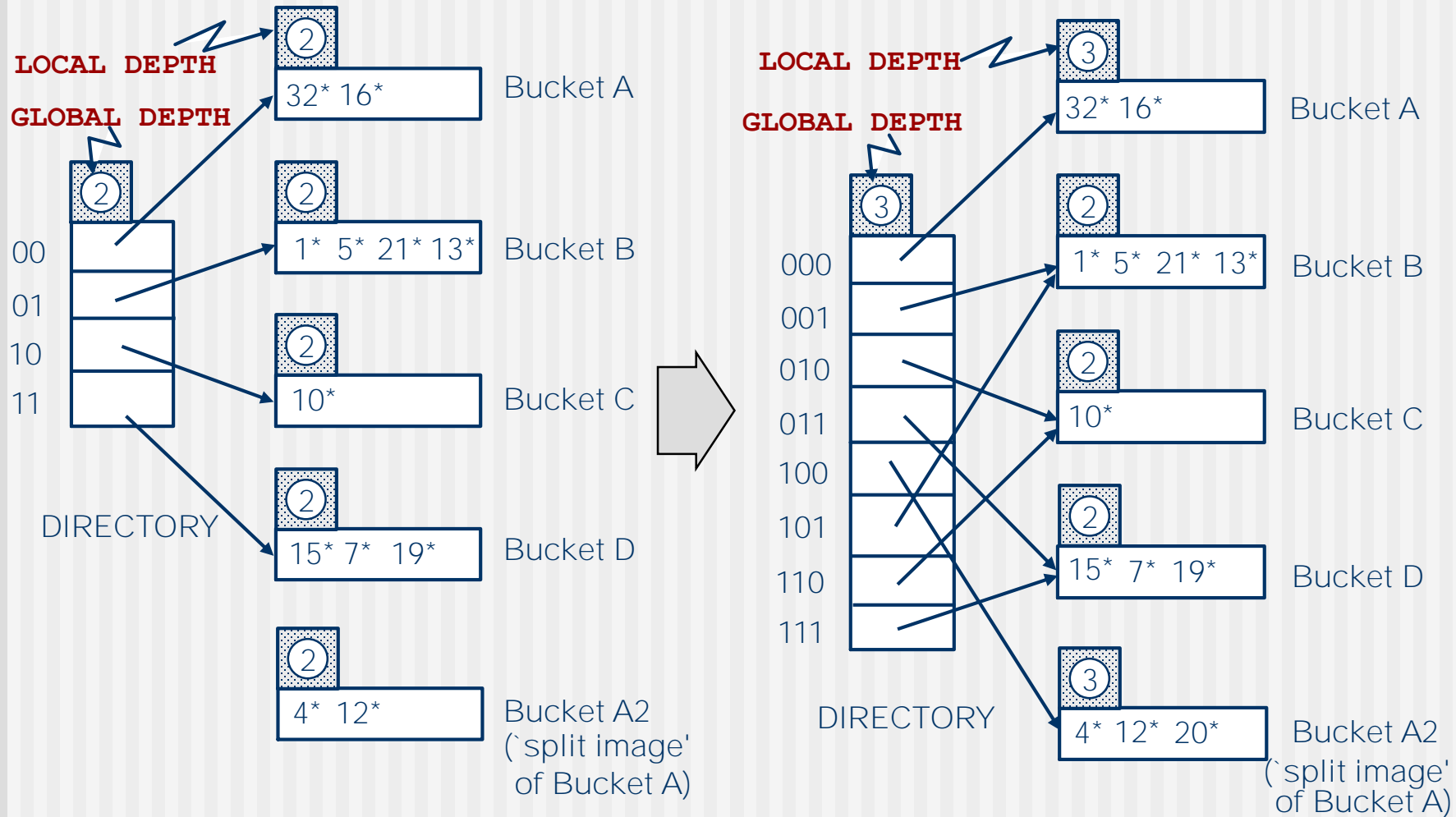
- Situation: Bucket (primary page) becomes full. Why not reorganize file by *doubling* number of buckets?
 - Reading and writing all pages is expensive!
 - Idea: Use *directory of pointers to buckets*, double number of buckets by *doubling the directory*, splitting just the bucket that overflowed!
 - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page!*
 - Trick lies in how hash function is adjusted!

Example

- Directory is array of size 4.
- To find bucket for r , take last '*global depth*' number of bits of $h(r)$; we denote r by $h(r)$.
 - If $h(r) = 5 = \text{binary } 101$, it is in bucket pointed to by 01.
- **Insert:** If bucket is full, *split* it (*alloc. 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.)



Insert k: $h(k) = 20 \rightarrow$ Doubling

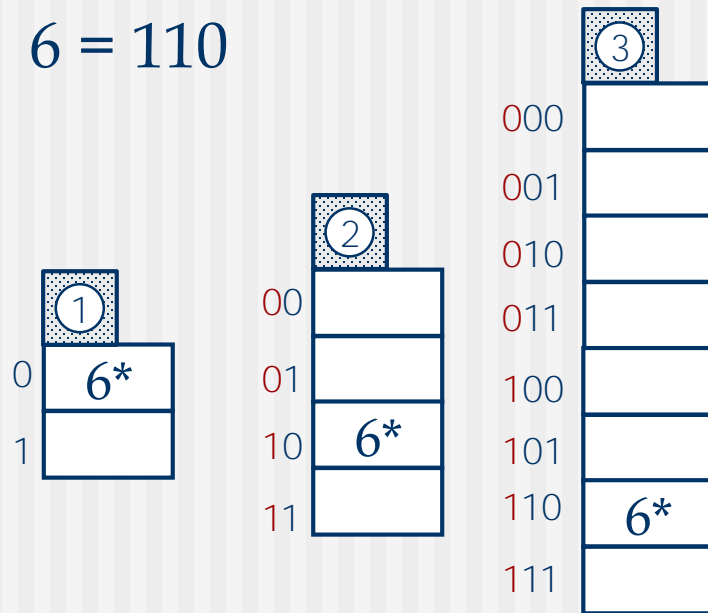


Extendible Hashing

- Inserting $h(k) = 20$ (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 number of bits needed to tell which bucket an entry belongs to.
 - *Local depth of a bucket*: number of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
 - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become $>$ *global depth*; directory is doubled by *copying it over* and 'fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

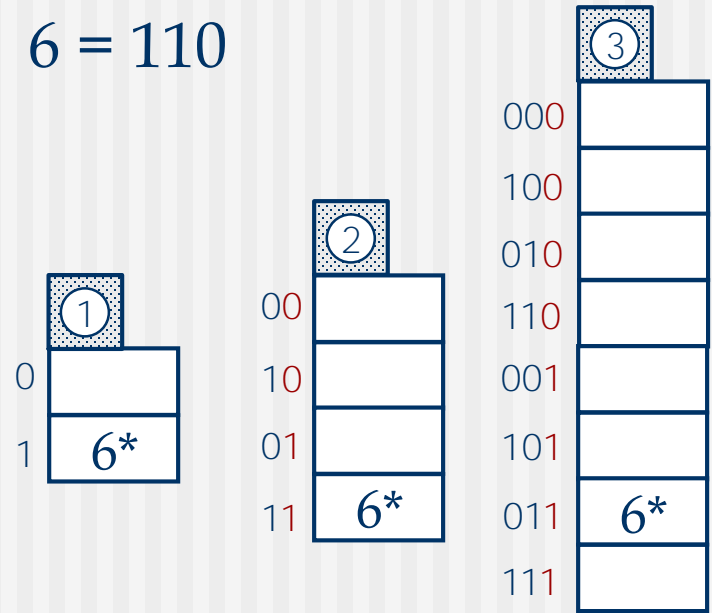
Directory Doubling

- Using least significant bits in directory → 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.
 - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
 - Directory grows in spurts, and, if the distribution of *hash values* is skewed, 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 same bucket as its split image, can halve directory.

Evaluation of Hash-Based Files

■ Advantages:

- Very fast access from information you are likely to already have.
- Best for *equality selections*, e.g.

SELECT * FROM R WHERE A = k

while the underlying data file - for relation R - grows or shrinks, we can answer such an equality query using a single I/O operation.

Evaluation of Hash-Based Files (cont)

■ Disadvantages:

- There can only be one hash arrangement based on one hash key. The other forms of access must be done by indexing.
- The sequential order of the records is unlikely to be meaningful in any way, although there is work being done on order-preserving hash algorithms that keep records in the logical order of their hash key.
- There are likely to be gaps of empty spaces of various sizes in the file – it won't be consistently loaded.
- Cannot support range searches.
- Does not support retrieval based on other fields than *exact* hash field
- Not recommended when hash field is frequently updated