

Access Methods II: Hash-based Indexes

"If you don't find it in the index, look very carefully through the entire catalogue."

-- Sears, Roebuck, and Co.,
Consumer's Guide, 1897

Hash-based Index

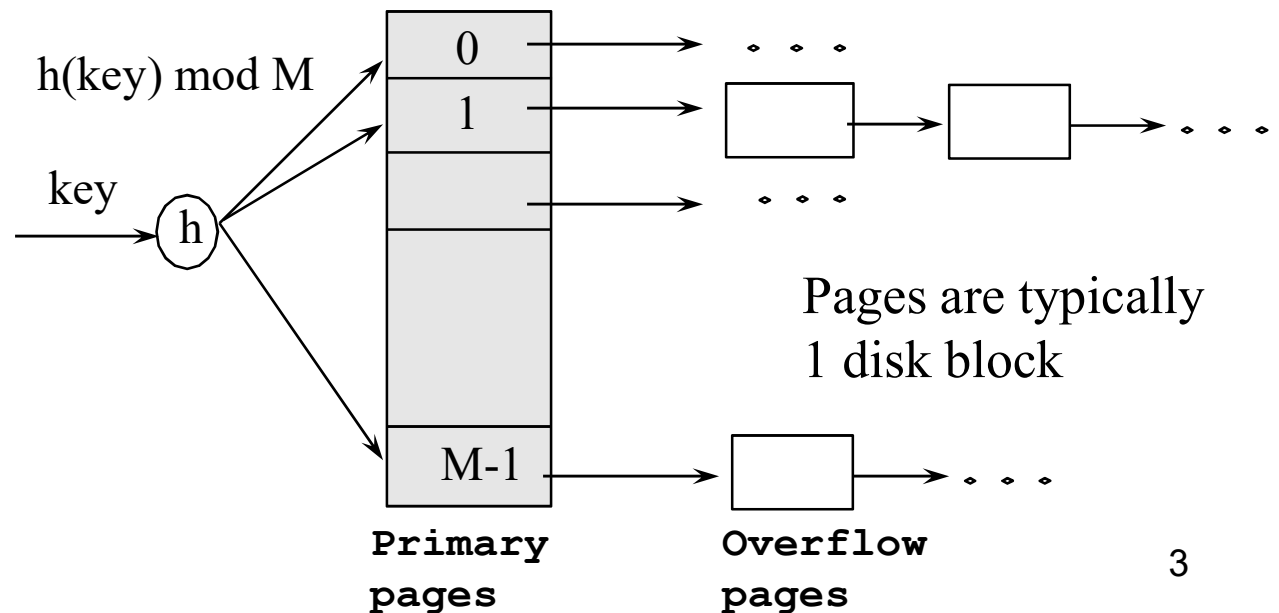
- *Hash-based indexes*
 - Key k , hash function h , $h(k)$ returns the **bucket**/page ID that stores record with key k
 - (Ideally) best for **equality** selections
 - Performance degenerate for **skewed** data distributions
 - **Inefficient for range searches**
 - Depends on hash function used
- Static and dynamic hashing techniques exist

Static Hashing

- Data is stored in M **buckets**: B_0, B_1, \dots, B_{M-1}
 - M is **fixed** at creation time
- Hashing function $h(\cdot)$ is used to determine the bucket to store a record
 - A record with search key k is inserted into bucket B_j , where
$$j = h(k) \bmod M$$
 - Example: hash function of the form $h(\text{key}) = a \cdot \text{key} + b$

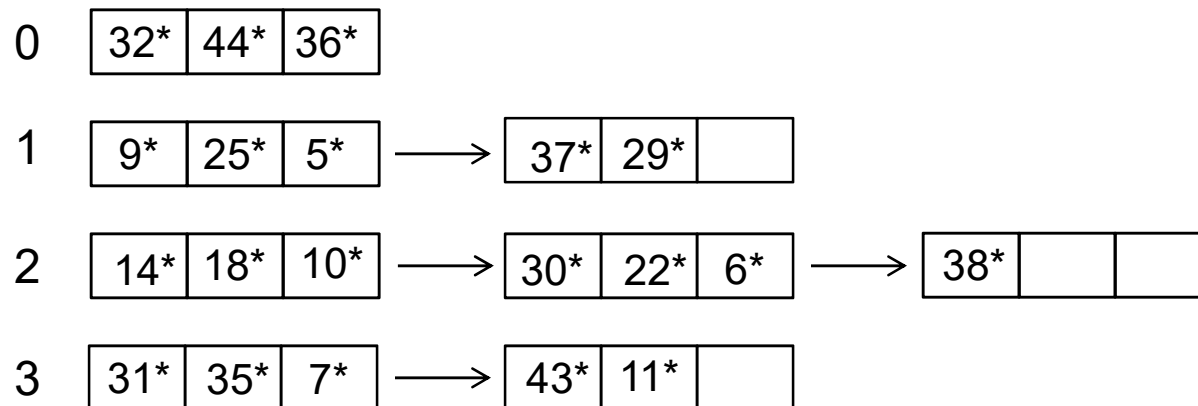
Long overflow chains can develop and degrade performance

- Each bucket consists of one **primary data page** and a chain of zero or more **overflow data pages**
- Primary pages are allocated sequentially, never de-allocated



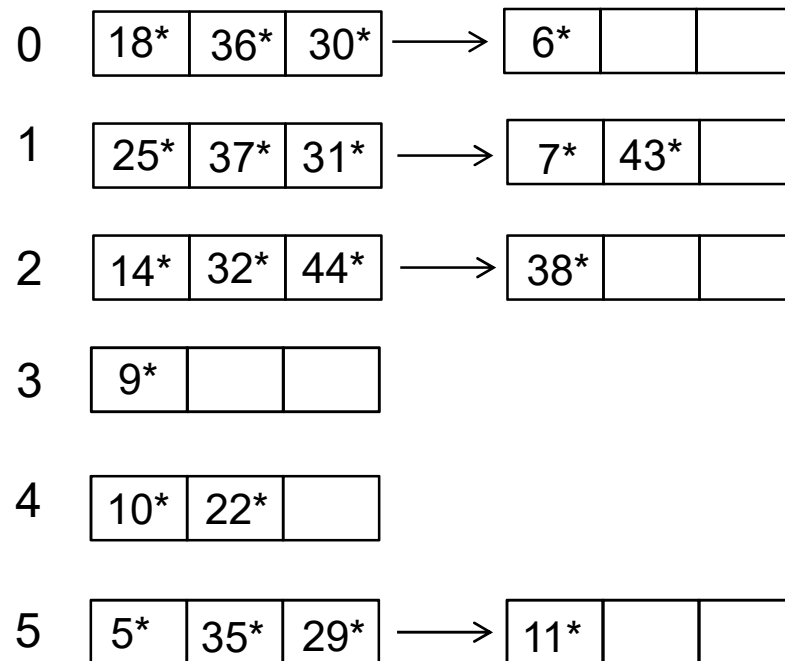
Static Hashing: Example

- k^* denotes a data entry e with key k
- Suppose we have the following keys, and we have a hash index with $M = 4$ buckets, each of which can hold 3 records (hashing function: $j = h(k) = k \bmod 4$)
 - 9, 14, 18, 32, 25, 5, 10, 44, 37, 31, 35, 29, 36, 7, 43, 30, 22, 6, 38, 11



Static Hashing: Example

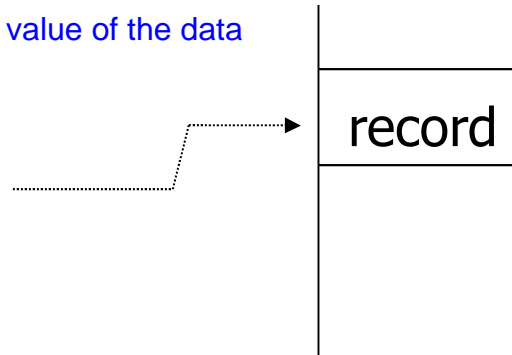
- What if we have $M = 6$ buckets (hashing function: $j = h(k) = k \bmod 6$)?
 - 9, 14, 18, 32, 25, 5, 10, 44, 37, 31, 35, 29, 36, 7, 43, 30, 22, 6, 38, 11



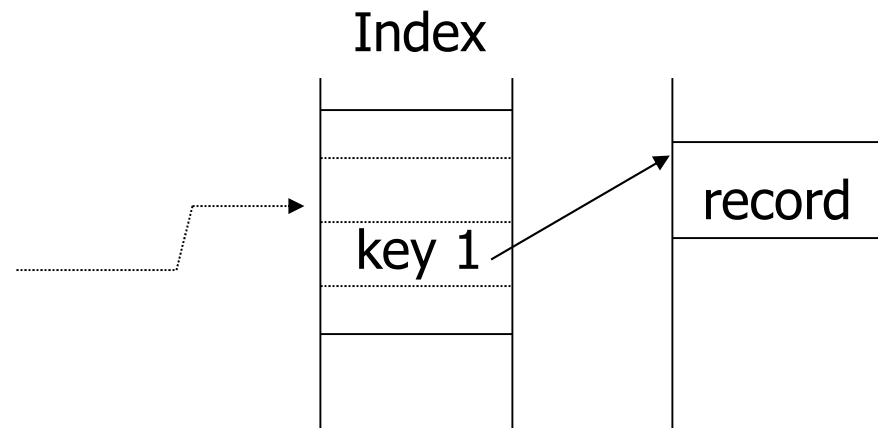
Two alternatives

Format one will actually store the value of the data entry in the bucket

(1) $\text{key} \rightarrow h(\text{key})$



(2) $\text{key} \rightarrow h(\text{key})$



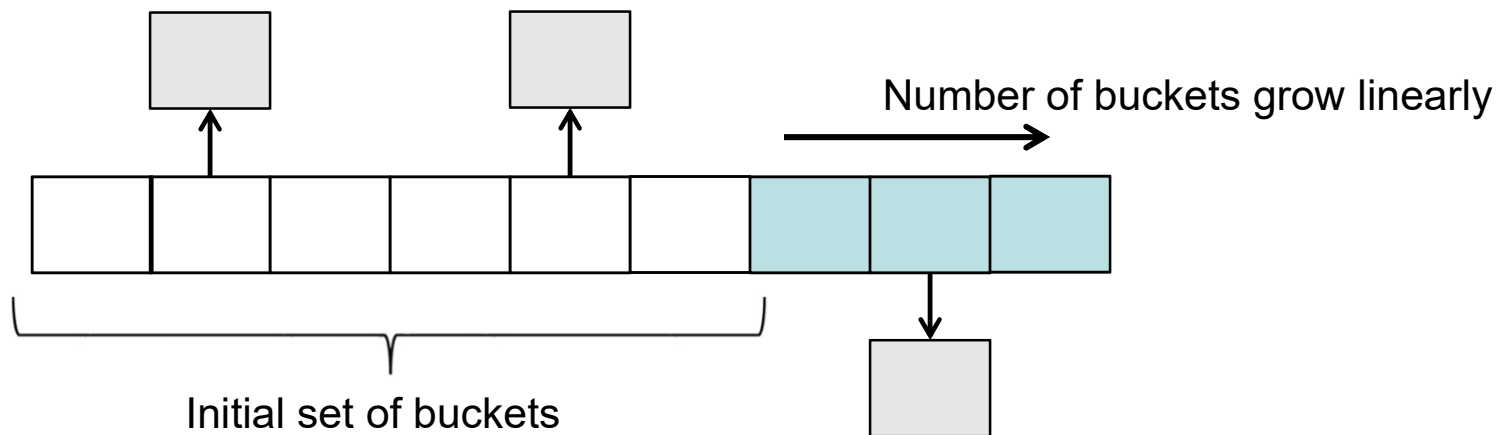
Format 2 will store the key pointer pair = similar to B+ Tree

Static Hashing (Cont.)

- Buckets may contain *data records or pointers*
 - Unless otherwise stated, we assume the **former**
- Hash fn works on *search key* field of record *r*
 - Must distribute values over range 0 ... M-1
 - $h(key) = (a * key + b) \bmod M$ usually works well
 - a and b are constants
 - h has to be tuned for different applications
- **Long overflow chains** can develop and degrade performance
- How to cope with growth?
 - Overflows and **reorganization** (may need to change the hash fn)
 - Dynamic hashing: **Linear Hashing** and **Extendible Hashing**

Dynamic Hashing: Linear Hashing

- Linear Hashing (LH) is a *dynamic* scheme
 - Hash file grows *linearly (one bucket at a time)* by systematic splitting of buckets
- LH handles the problem of long overflow chains
 - Overflow pages are still needed as an overflowed bucket might not be split immediately

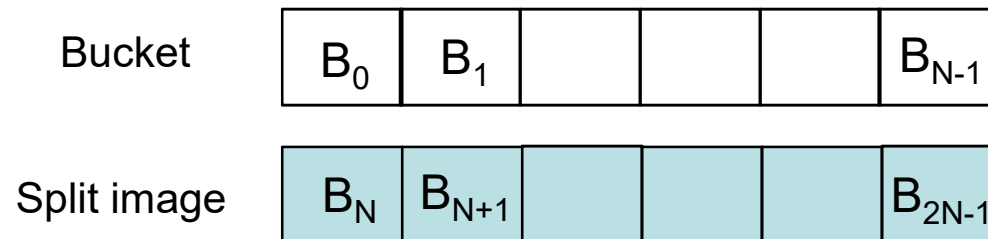


Challenges

- How to split a bucket B_i ?
 - Obtain a new bucket B_j (known as **split image** of B_i)
 - Redistribute entries in B_i between B_i and B_j
 - Issues
 - How to determine j ?
 - How to redistribute entries?
 - How to modify hash function?
- When to split a bucket?

How to determine split image?

- Suppose the initial file size has N buckets: B_0, \dots, B_{N-1}
- Imagine a corresponding set of **virtual** buckets (not created until needed): B_N, \dots, B_{2N-1}



- To split bucket B_i , choose B_{N+i} as its split image (and create B_{N+i})
- File size doubles after one round of splitting (i.e., every bucket has been split)

File doubles after each round ...

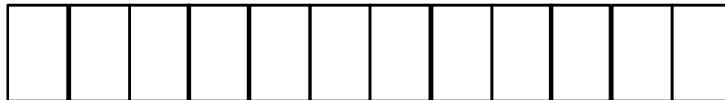
Initial



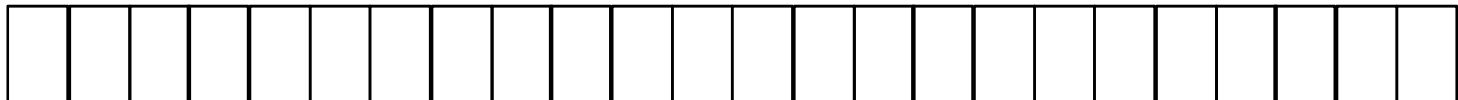
After 1 round



After 2 rounds



After 3 rounds



NOTE:

(a) File grows one bucket at a time

(b) Hash function has to be changed after each round!

How to redistribute entries?

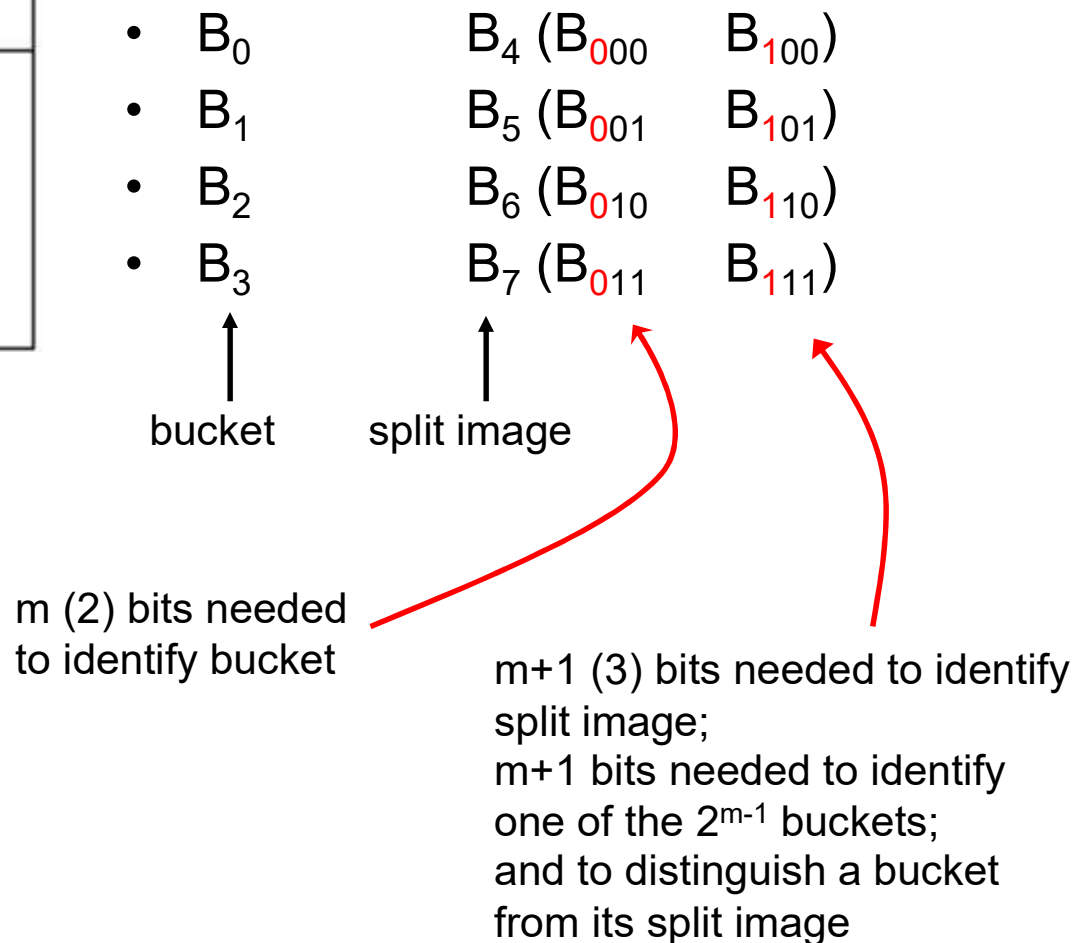
- Let the initial file size be $N_0 = 2^m$ ($m > 1$)
 - Initial file has buckets B_0, \dots, B_{N_0-1}
- Define the initial hash function h_0 as follows:
$$h_0(k) = \text{last } m \text{ bits of } h(k)$$
- $h_0(k) \in [0, N_0 - 1]$
- An entry e belongs to bucket B_i if $h_0(e.\text{key}) = i$
- Example: $m = 2$, $N_0 = 4$, initial hash file has buckets B_0, \dots, B_3

k	h(k)	$h_0(k)$
Alice	$\dots 010$	10
Bob	$\dots 110$	10
Carol	$\dots 111$	11
Dave	$\dots 110$	10

Alice belongs to bucket B_2
Bob belongs to bucket B_2
Carol belongs to bucket B_3
Dave belongs to bucket B_2

How to redistribute entries?

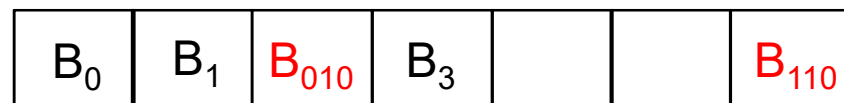
k	h(k)	h ₀ (k)
Alice	...010	10
Bob	...110	10
Carol	...111	11
Dave	...110	10



How to redistribute entries?

k	h(k)	h ₀ (k)
Alice	...010	10
Bob	...110	10
Carol	...111	11
Dave	...110	10

- $B_0 \rightarrow B_0$ or B_4 (B_{000} or B_{100})
- $B_1 \rightarrow B_1$ or B_5 (B_{001} or B_{101})
- $B_2 \rightarrow B_2$ or B_6 (B_{010} or B_{110})
- $B_3 \rightarrow B_3$ or B_7 (B_{011} or B_{111})



 this might create under utilisation

How to redistribute $\{Alice, Bob, Dave\}$ in B_2 ?

Split image of B_2 is B_6

Alice remains in B_2

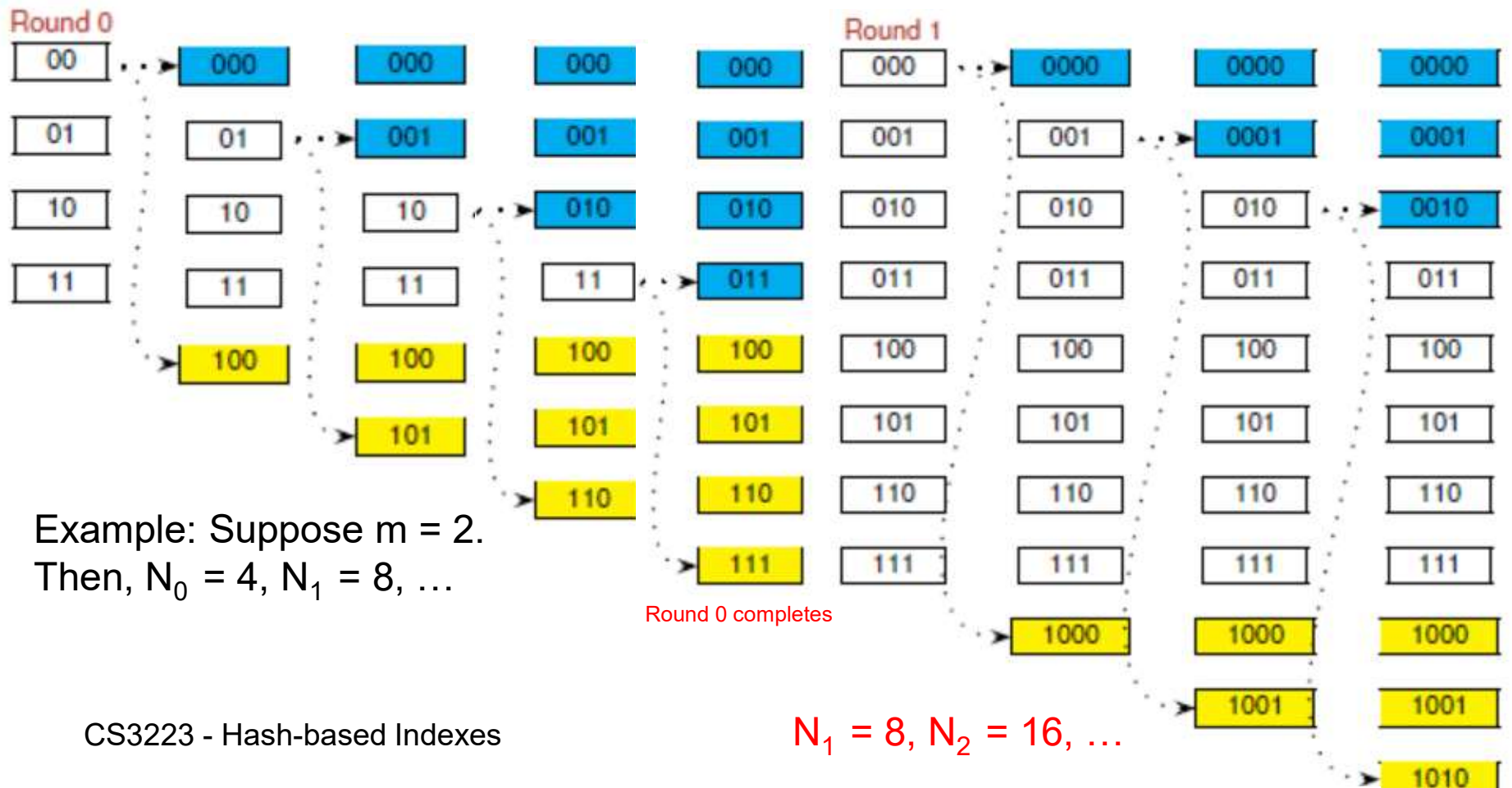
Bob & Dave are moved to B_6

How to redistribute entries?

- To distribute entries in B_i between B_i, \dots, B_{N_0+i}
 - Use the last $(m+1)^{\text{th}}$ bit of $h(e.\text{key})$ to redistribute entry $e \in B_i$
 - If the last $(m+1)^{\text{th}}$ bit of $h(e.\text{key})$ is 0, e remains in B_i
 - Otherwise, move e to B_{N_0+i}
 - If e is moved to B_{N_0+i} , the last $(m+1)$ bits of $h(e.\text{key})$ must be $(=) N_0 + i$

Linear Hashing Example

- To grow file linearly, buckets are split *sequentially*
 - Bucket B_i must be split *before* bucket B_{i+1}
- File size doubles after each round of splitting

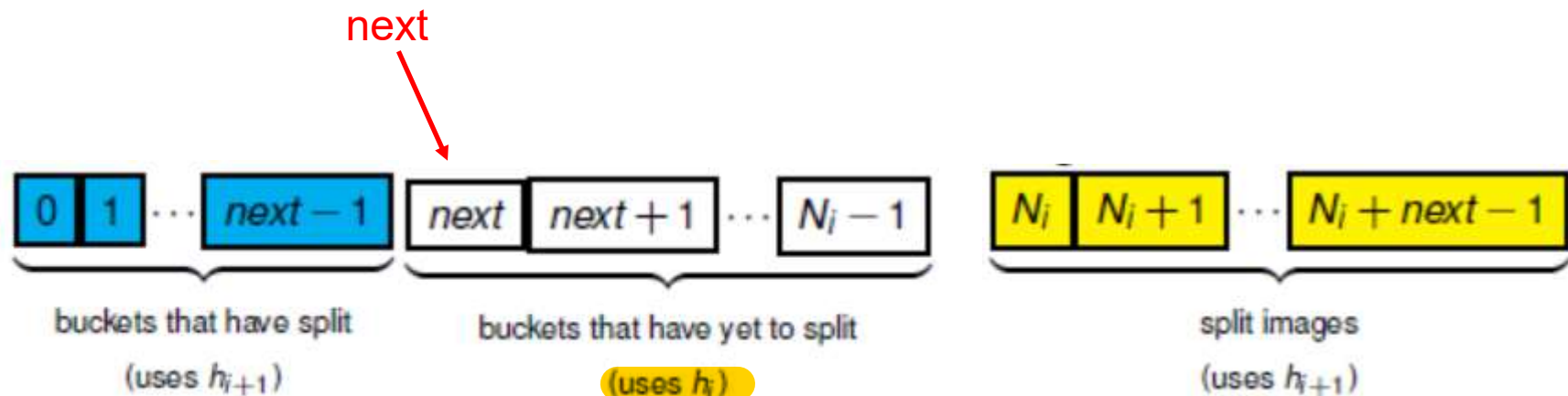


How to modify the hash function?

- Splitting progresses in rounds represented by level = 0, 1, 2, ...
- Let the initial file size be $N = 2^m$ buckets
- At round i
 - Number of buckets at the beginning is $N_i = 2^i N = 2^{m+i}$
 - Number of buckets at the end of round i is $2 N_i$
 - The bucket to be split is chosen in **round robin fashion**
 - **next** = a pointer to the next bucket to be split
 - $\text{next} \in \{0, 1, \dots, N_i - 1\}$
 - When bucket B_{next} is split, its split image is $B_{\text{next} + N_i}$
 - Uses a pair of hash functions: h_i and h_{i+1}
 - $h_{i+1}(k) = \text{last } (m+i) \text{ bits of } h(k)$
 - $h_i(k) = h(k) \bmod N_i$ (**this is essentially the last m bits**)

How to modify the hash function?

- Buckets can be classified into three regions:



Use h_i to first determine if the data is in the original list

- then if it can be found in the area of the "next pointer" where buckets have yet to split then return
- else use the 2nd hash function of h_{i+1} to find if it is in the original list or the split image list

When to split a bucket?

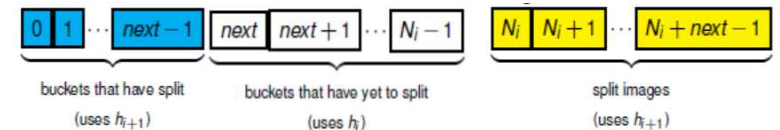
- The time to split the next bucket can be decided with various criteria:
 - Split whenever **some** bucket overflows
 - Split whenever space utilization of file is above some threshold
 - etc
- Overflow pages are still needed since an overflowed bucket might not be split immediately
- Some split pages may not be overloaded
- We shall assume that a ~~bucket~~ is triggered whenever **some bucket overflows**
 - Bucket B_j overflows if an entry is to be inserted into B_j and all the pages in B_j (i.e., primary and overflow pages) are full

So far, ...

- Initially
 - We have 2^m buckets
 - We have 2 hash functions h_i and h_{i+1} (based on last $m+i$ bits and last $m+i+1$ bits)
 - We have a next (*next-bucket-to-split*) pointer; this is pointing at bucket 0
- File grows one bucket at a time
 - Whenever a bucket is split, its records are redistributed using the last $m+i+1$ bits
 - Increment the *next* pointer

Inserting a data entry with search key k

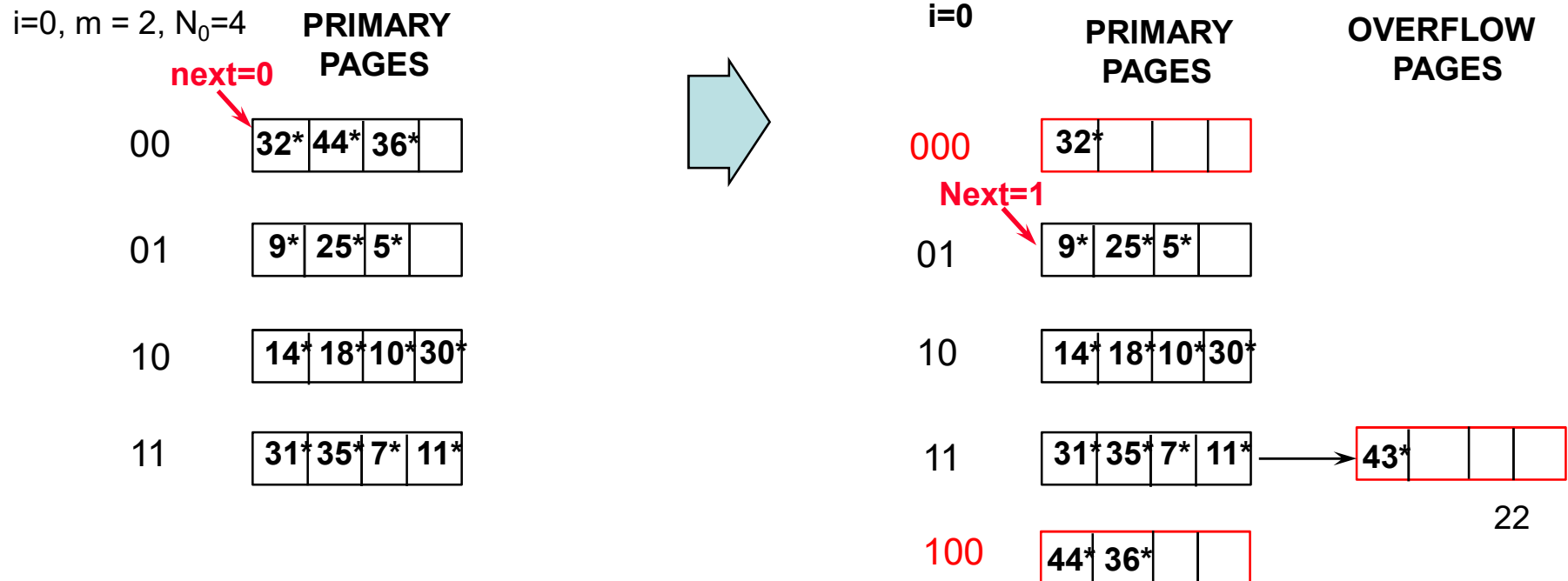
- Find bucket by applying hash function h_i
 - Use $h_i(k)$ if $h_i(k) \geq \text{next}$ (why?)
 - Otherwise, apply h_{i+1}



- If bucket to insert into is full
 - Add overflow page and insert data entry
 - Split $next$ bucket and increment $next$
 - Redistribute entries in B_{next} to B_{next+N_i} using h_{i+1}
 - $next = next + 1$
 - Reset $next$ to bucket 0 when a round completes
- Since buckets are split round-robin, long overflow chains not expected to develop!

Example of Linear Hashing

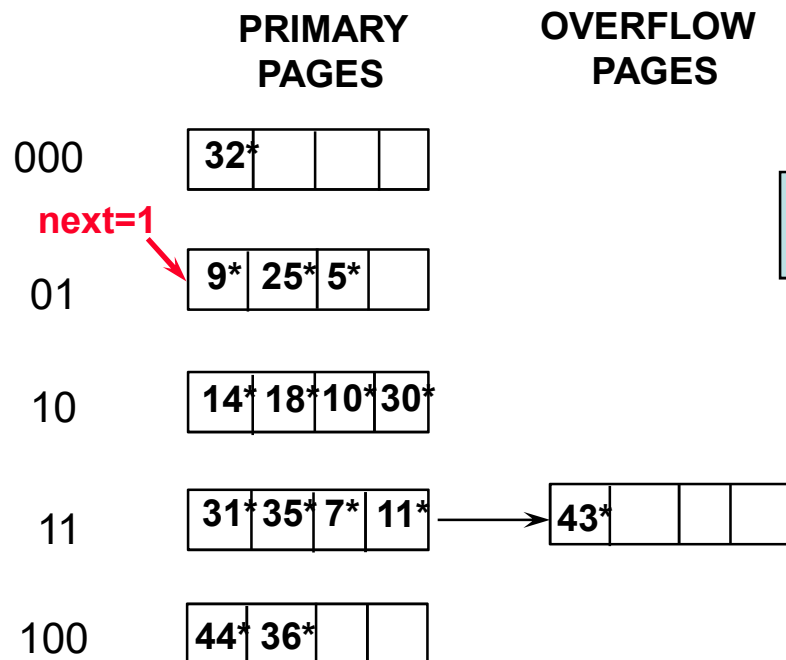
- Recall: On **split**, h_{i+1} is used to **re-distribute** entries
- Insert 43^* (1010**11**)
 - Bucket B_{11} overflows (but we are not splitting it!)
 - Split bucket B_{00} ($32 = 100**0**00$, $36 = 100**1**00$, $44 = 101**1**00$)
 - Increment next to 1
 - Insert 43^* into overflow page



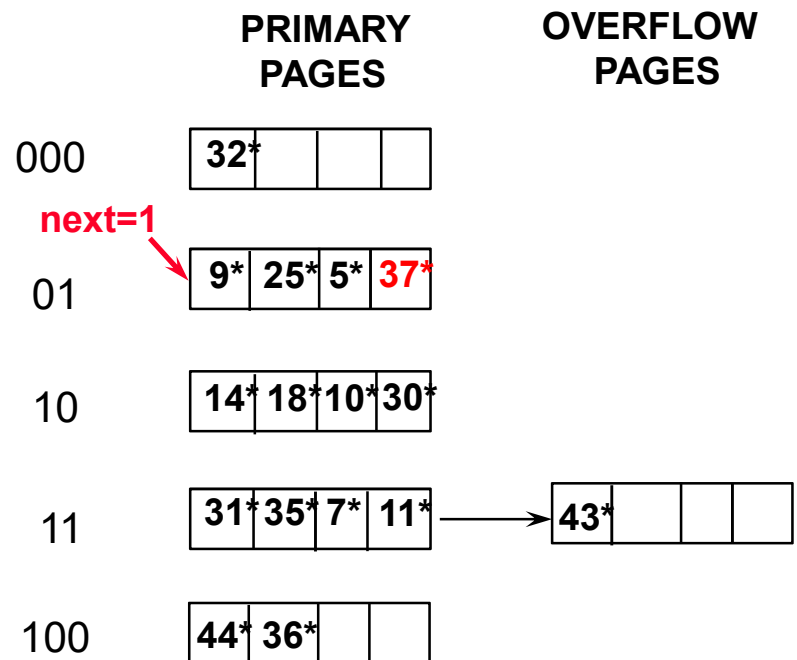
Example of Linear Hashing

- Insert 37^* (100101)
 - No overflow, no splitting
 - Insert 37^* into bucket B_{01}

$i=0, N_0=4$



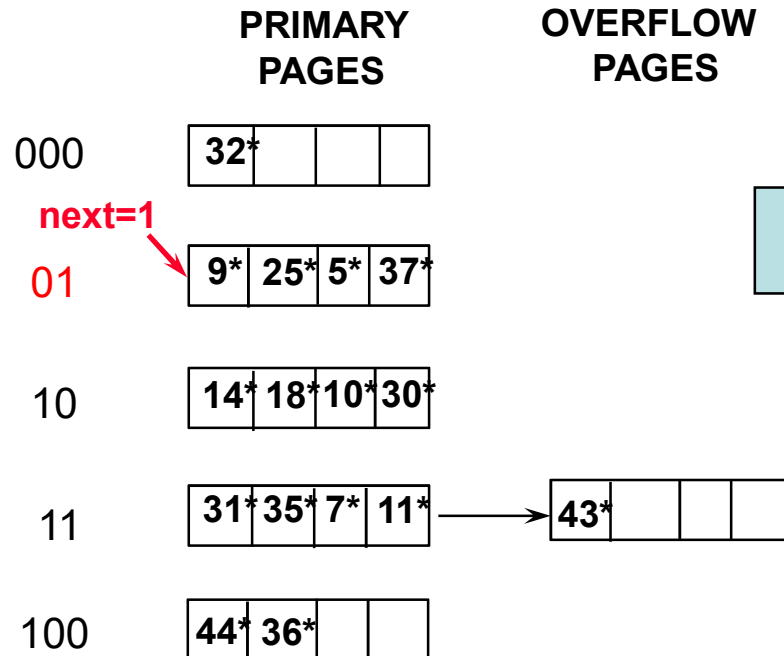
$i=0, N_0=4$



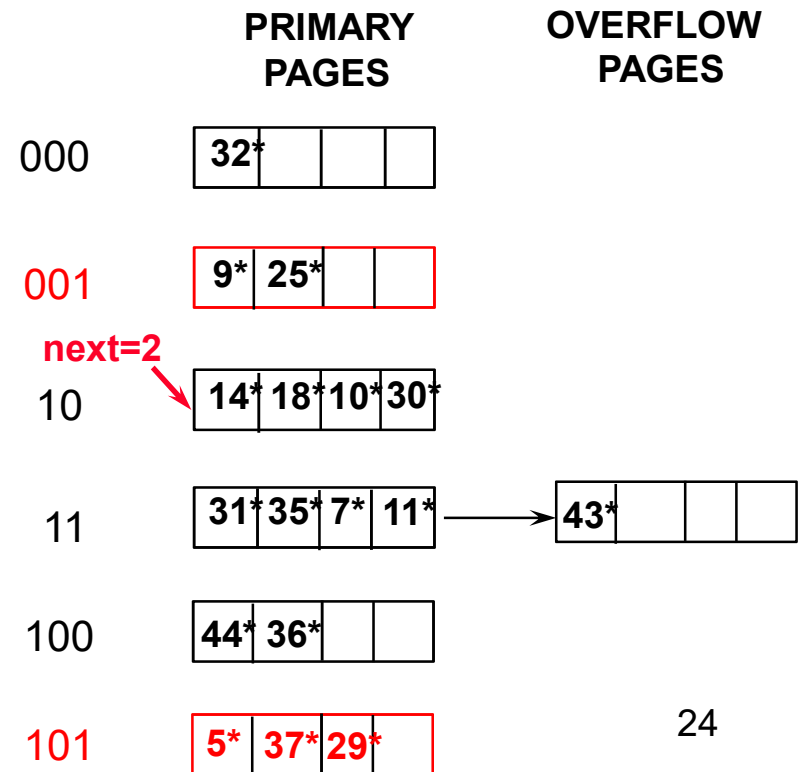
Example of Linear Hashing

- Insert 29^* (0111**01**)
 - Bucket B_{01} overflows
 - Split bucket B_{01} ($5 = 101$, $9 = 1001$, $25 = 11001$, $37 = 100101$)
 - Increment next to 2
 - Insert 29^* into bucket B_{101}

$i=0, N_0=4$



$i=0, N_0=4$

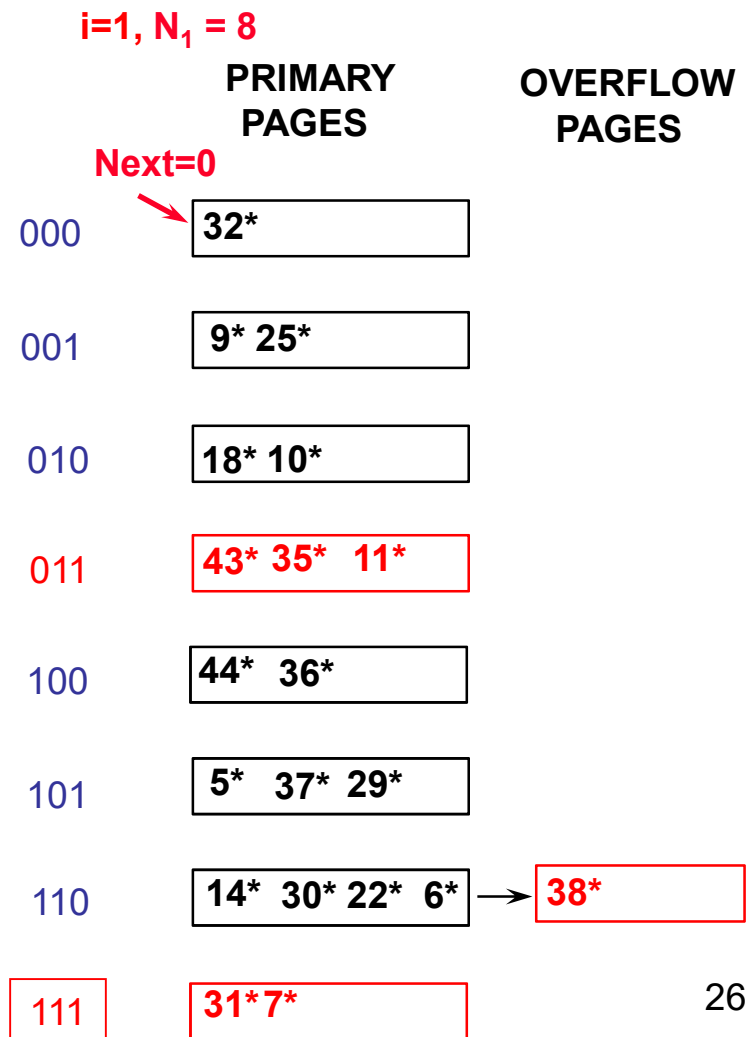
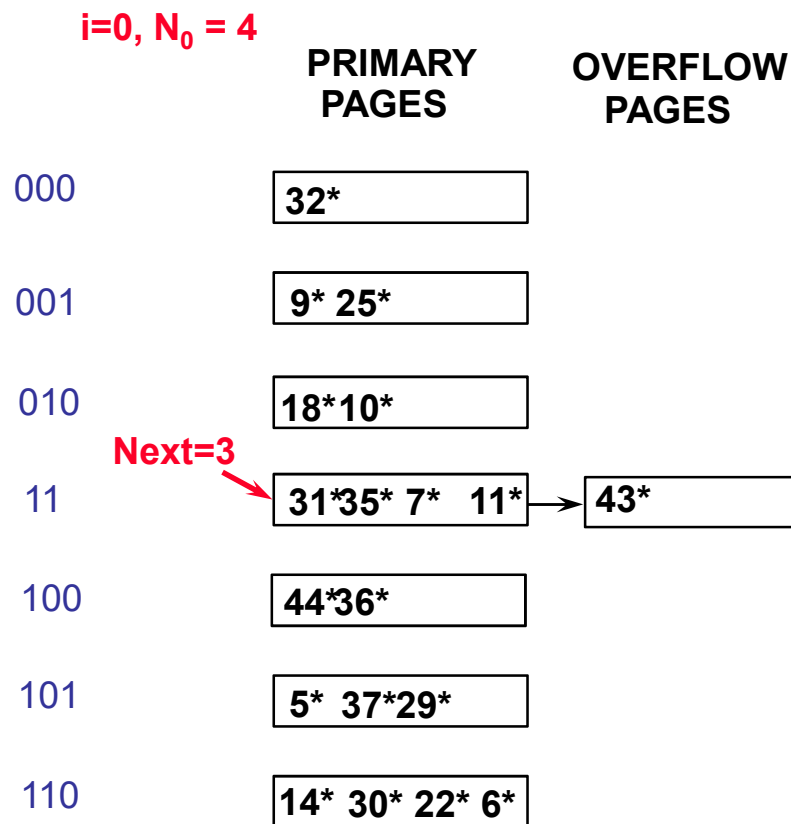


More insertions ...

- Insert 22^* (010110)
 - Bucket B_{10} overflows
 - Split bucket B_{10} ($10 = 1010$, $14 = 1110$, $18 = 10010$, $30 = 11110$)
 - Increment next to 3
 - Insert 22^* into bucket B_{110}
- Insert 6^* (000110)
 - No overflow, no splitting
 - Insert 6^* into bucket B_{110}

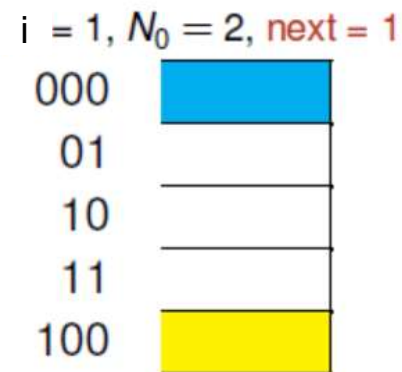
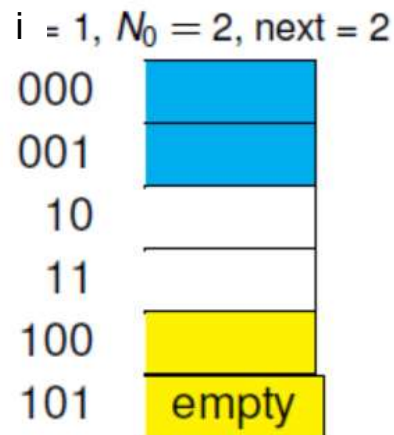
Example: Insert 38* (End of a Round)

- Insert 38* (100**110**)
 - Bucket B_{110} overflows
 - Split bucket B_{11} (7 = **111**, 11 = 1**011**, 31 = 11**111**, 35 = 100**011**, 43 = 101**011**)
 - Increment level to 1; reset next to 0
 - Insert 38* into overflow page



Linear Hashing: Deletion

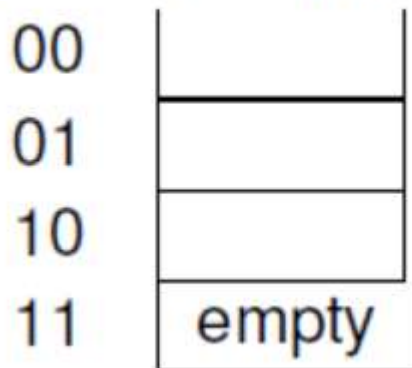
- Locate bucket and delete entry
- If the last bucket $B_{N_i + \text{next} - 1}$ becomes empty, it can be removed
- Case 1: If $\text{next} > 0$
 - Decrement next



Linear Hashing: Deletion

- Case 1: If (next = 0) and ($i > 0$)
 - Update next to point to the last bucket in previous level $B_{ni/2-1}$
 - Decrement level

$i = 1, N_0 = 2, \text{next} = 0$



$i = 0, N_0 = 2, \text{next} = 1$



Linear Hashing: Performance

- One disk I/O unless the bucket has overflow pages
 - On average 1.2 disk I/O for uniform or lowerly skewed data distribution
 - Worst case: I/O cost is linear in the number of data entries
- Poor space utilization with skewed data distribution

Why not simply double the file each time?

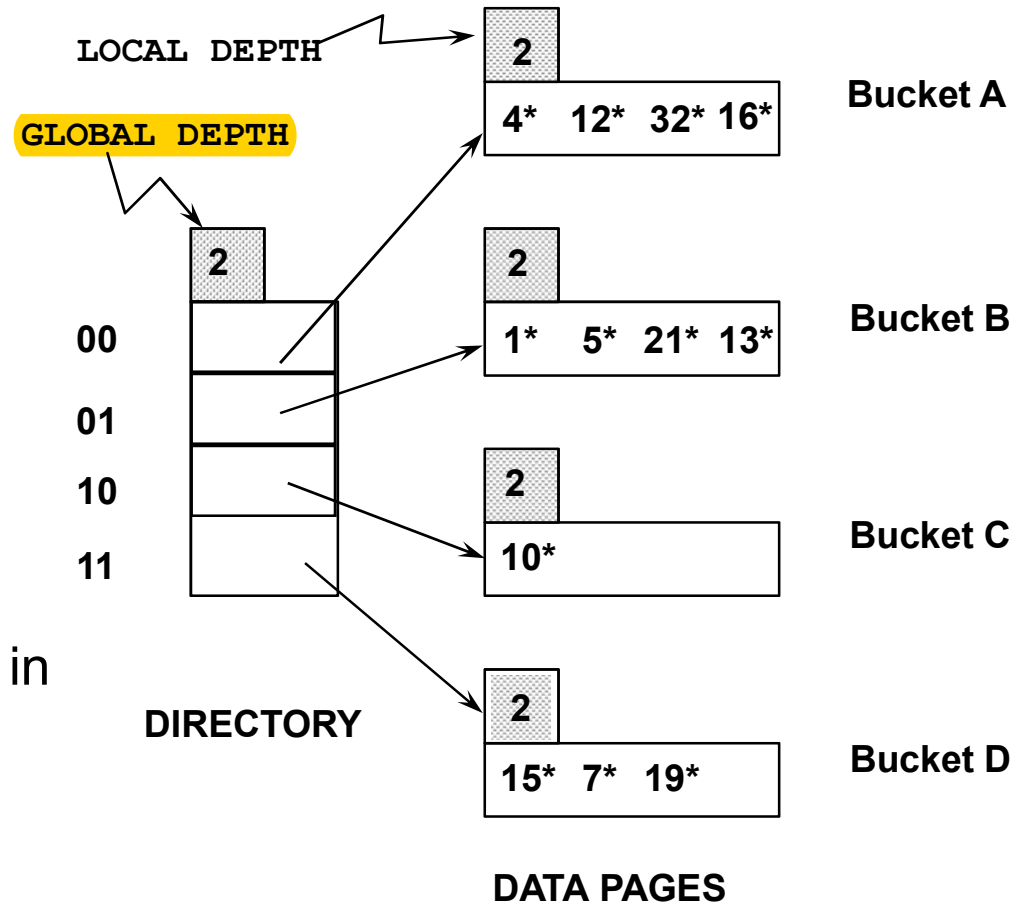
if instead of splitting the buckets 1 by 1, we split by immediately creating all the split images and rehash and split all buckets again - this would essentially just be redoing the entire file again which is inefficient

Dynamic Hashing: Extendible Hashing

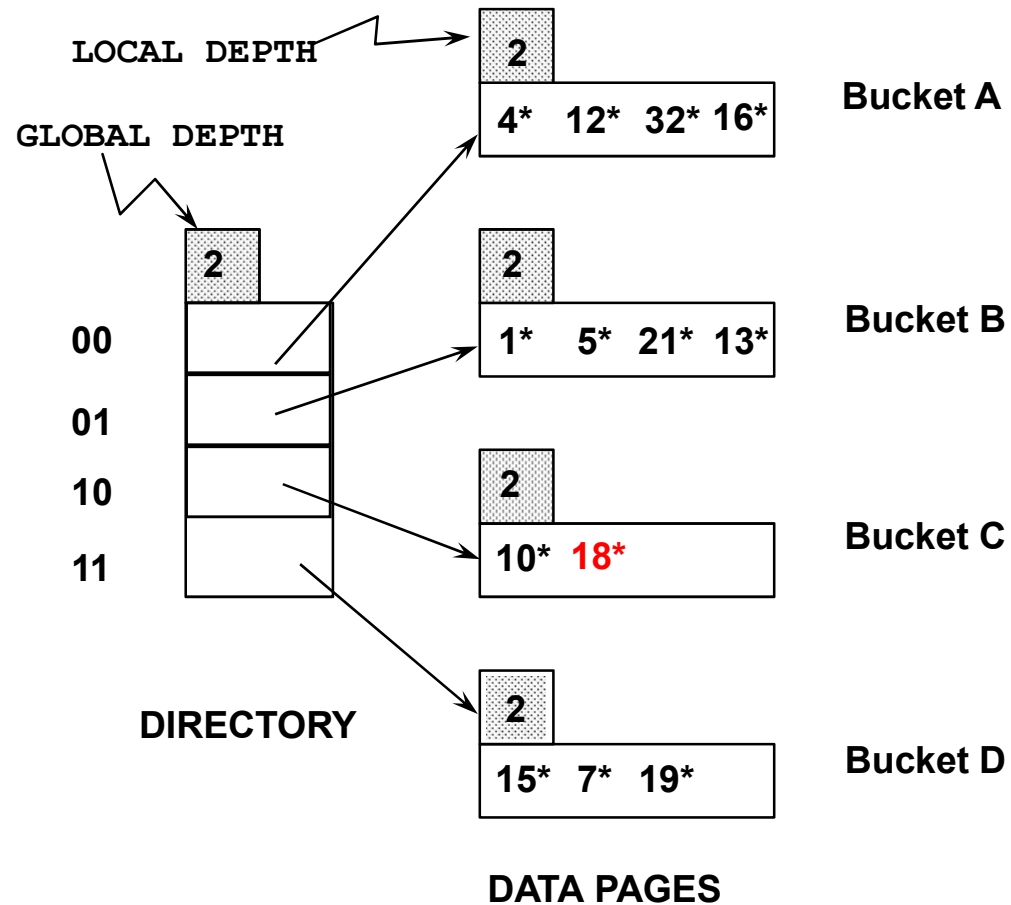
- Handle file growth by *doubling* # of buckets!
 - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory* (not the actual buckets), *splitting just the bucket that overflows*!
 - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split.
No (?) overflow page!
 - Trick lies in how hash function is adjusted!

Example

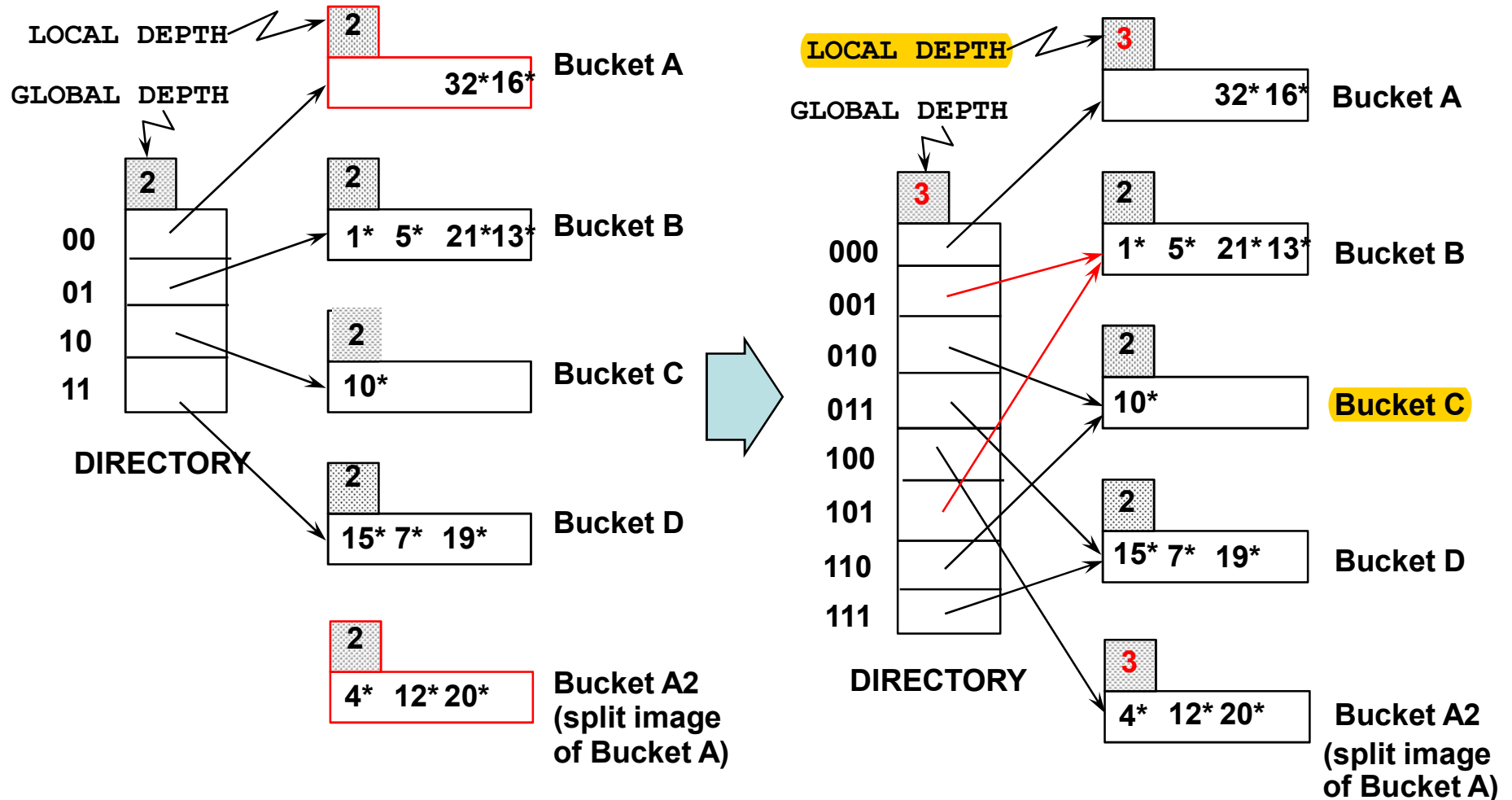
- Directory is array of size 4
- To find bucket for r , take last '*global depth*' # bits of $h(r)$
 - If $h(r) = 5 = \text{binary } 101$, it is in bucket pointed to by 01



Example: Insert 18 (10010)



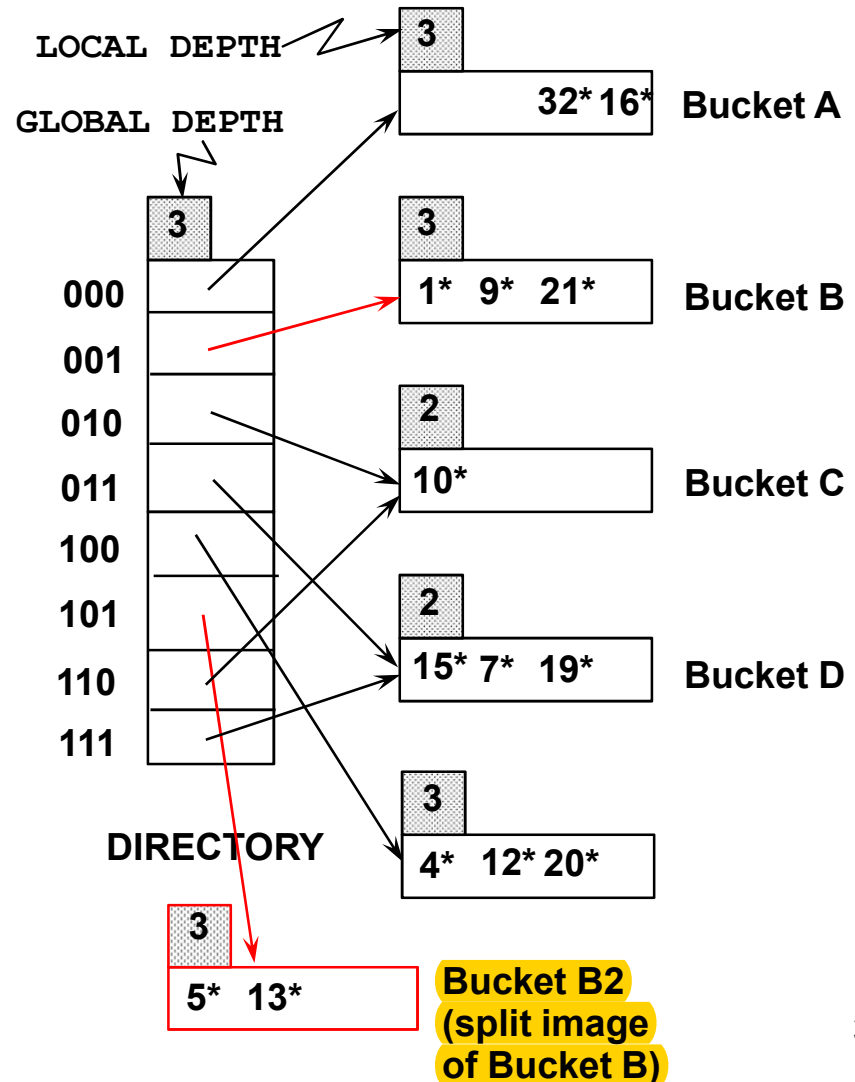
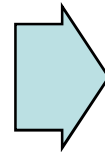
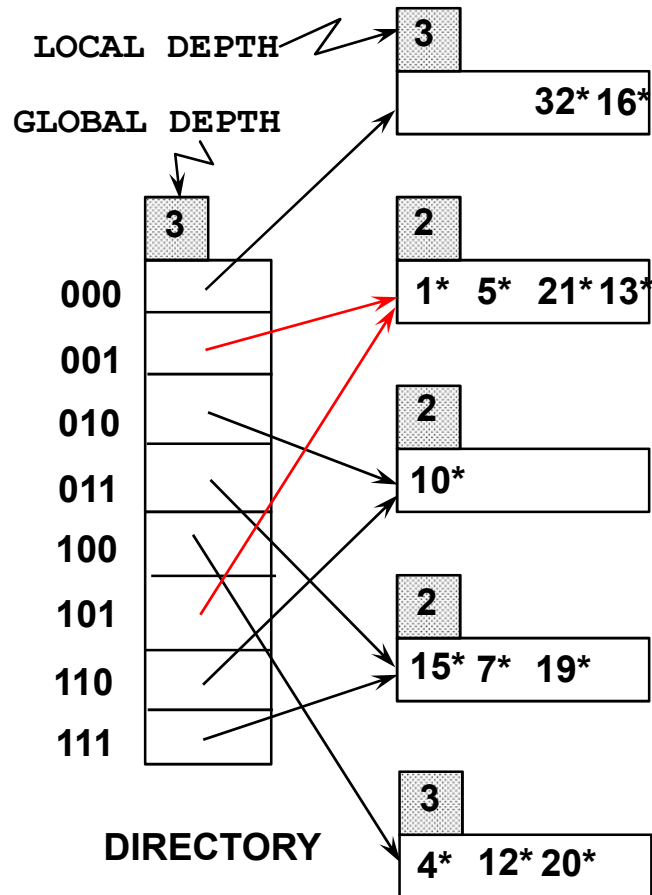
Insert $h(r)=20$ (10100) – Causes *Doubling*



Points to Note

- When does bucket split cause directory doubling?
 - Before insert, local depth of bucket to split = global depth
- 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 an entry belongs to this bucket.


Example: Insert 9 (01001)



Insertion

- If bucket is full, split it (allocate new page, re-distribute)
- If the split bucket has a local depth that is smaller than the global depth
 - we need to **adjust** the directory entries to point to the right buckets
- If the split bucket's local depth is equal to the global depth
 - we need to **double** the directory as well

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
 - 100MB file, 100 bytes/rec, 1,000,000 records, 4K pages.
 - 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
- **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

Summary

- Hash-based indexes: best for equality searches, cannot (?) support range searches
- Static Hashing can lead to long overflow chains
- Dynamic hashing methods such as Linear Hashing and Extendible Hashing avoid overflow pages (and hence speed up equality search cost) by splitting full buckets when new data entries are added