Access Methods II: Hash-based Indexes

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

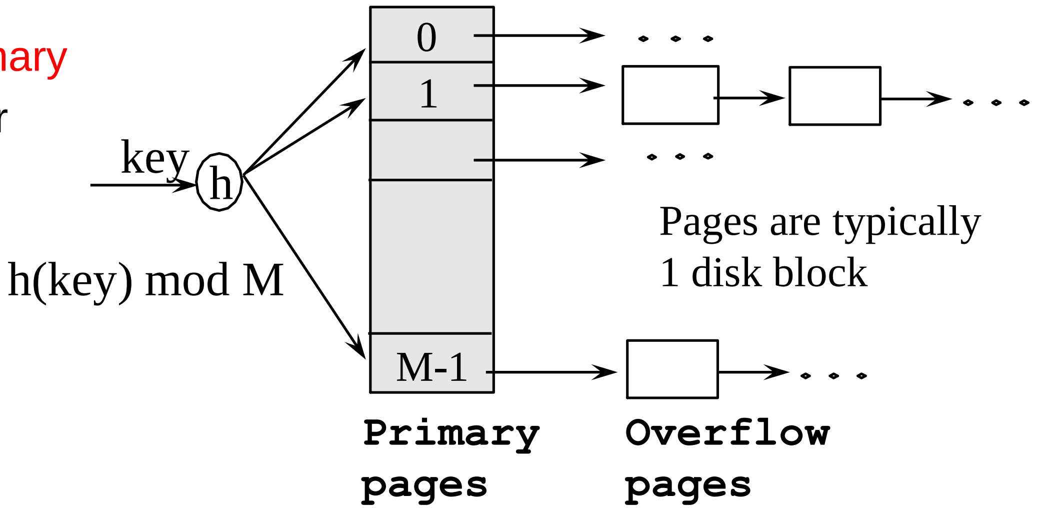
-- Sears, Roebuck, and Co., Consumer's Guide, 1897

Hash-based Index

- *Hash-based* indexes
  - Data (or data entries) are stored in buckets
  - 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

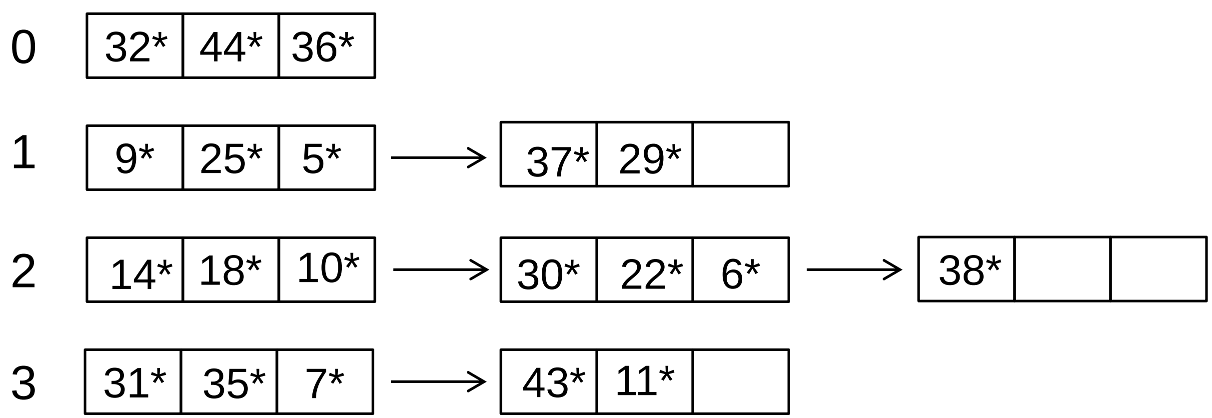
- Data is stored in M buckets: $B_0, B_1, \ldots, B_{M-1}$
  - M is *fixed* at creation time
  - Each bucket consists of one primary data page and a chain of zero or more overflow data pages
  - Primary pages are allocated sequentially, never de-allocated



- Hashing function h(.) 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(key) = a.key + b
    - a and b are constants
    - h has to be tuned for different applications
  - Long overflow chains can develop and degrade performance

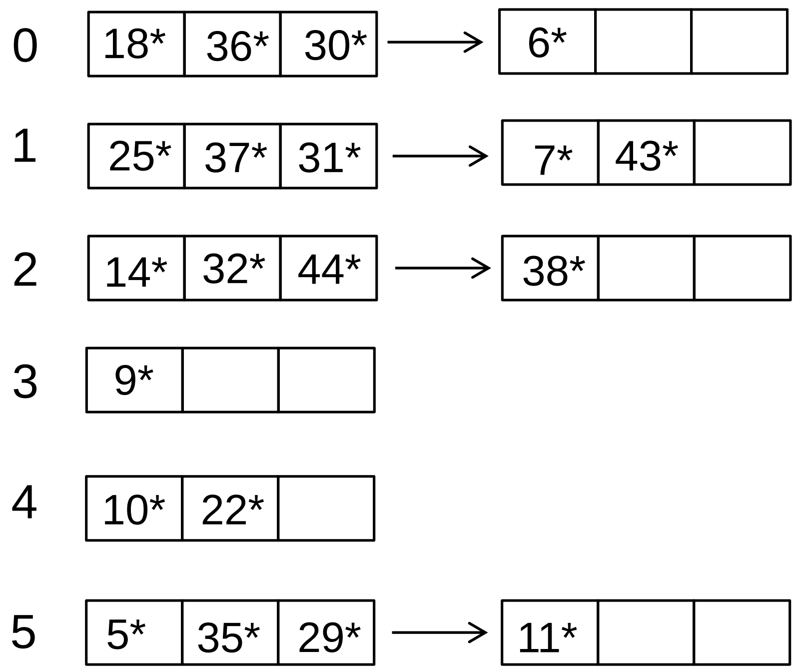
Static Hashing: Example

- 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 mod 4)
  - 9, 14, 18, 32, 25, 5, 10, 44, 37, 31, 35, 29, 36, 7, 43, 30, 22, 6, 38, 11



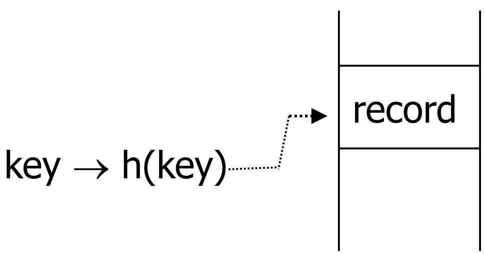
Static Hashing: Example

- 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

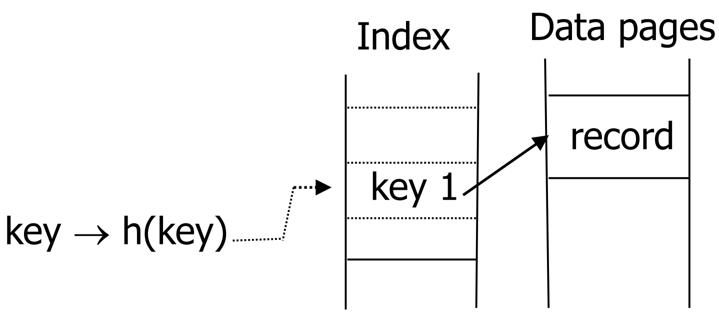


Static Hashing (Cont.)

- Buckets may contain *data records (Format 1) or pointers (Format 2)*
  - Unless otherwise stated, we assume the former (Format 1)



Format 1: Buckets contain the data records

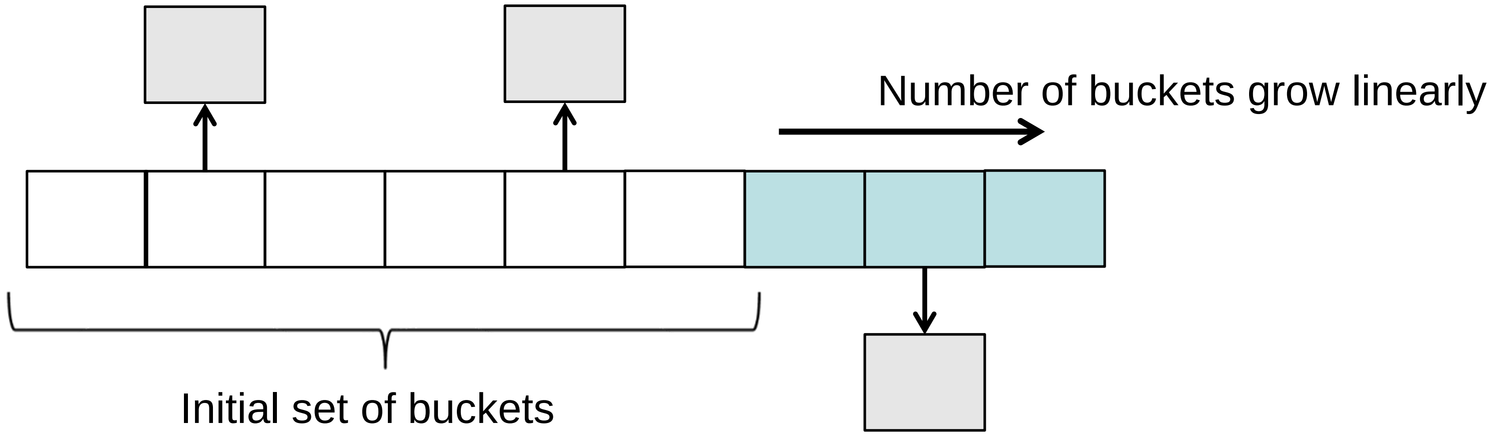


Format 2: Buckets contain the (key, pointer)-pairs

- 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?

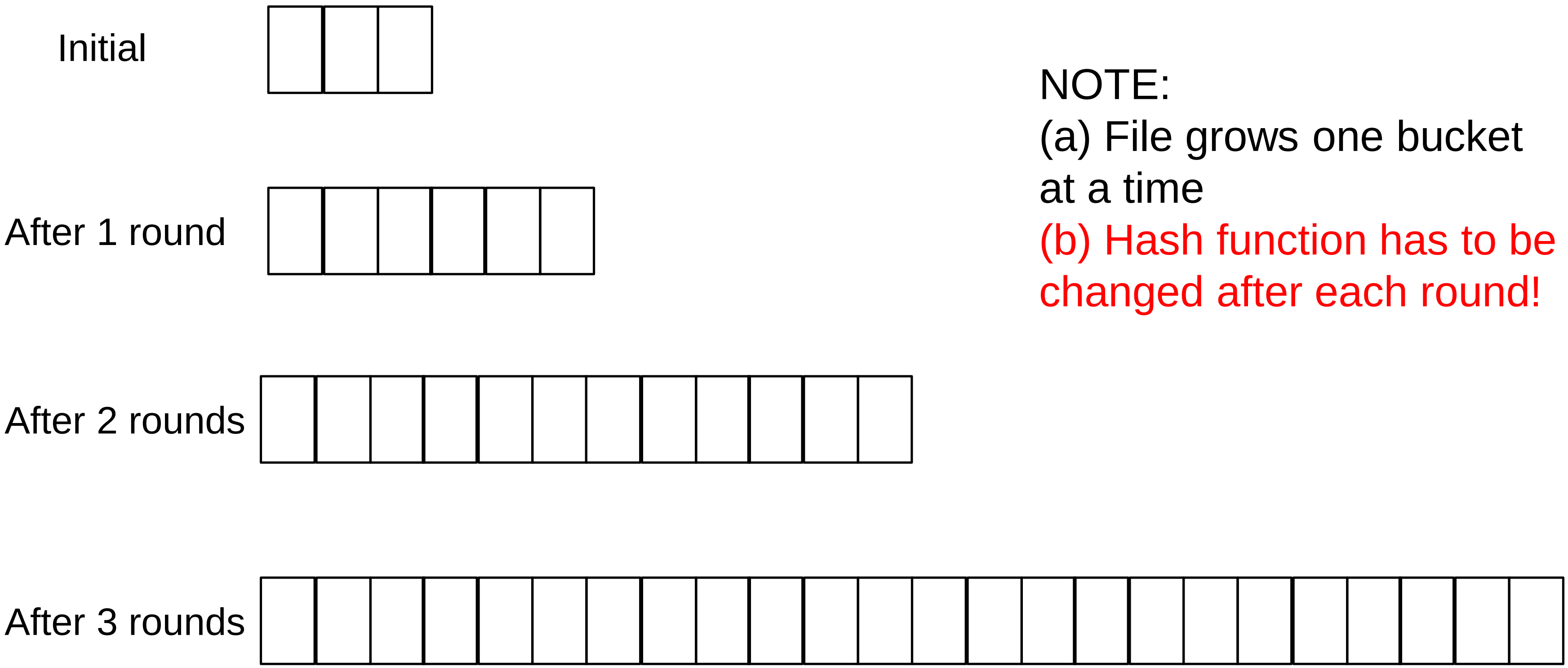
How to determine split image?

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

Bucket	B_0	B_1				B_{N-1}
Split image	B_N	B_{N+1}				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 …



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 $B_0, \ldots, B_{N0-1}$
- 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 $h_0(e.key) = i$
- Example: $m = 2$, $N_0 = 4$, initial hash file has buckets $B_0, \ldots, B_3$

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

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

bucket

split image

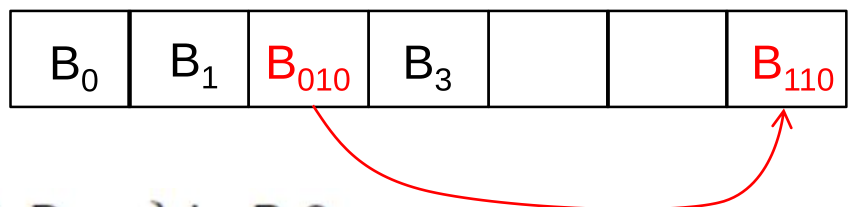
m (2) bits needed
to identify bucket

m+1 (3) bits needed to identify
split image;
(m+1)th bit needed to distinguish
a bucket from its split image

How to redistribute entries?

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



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

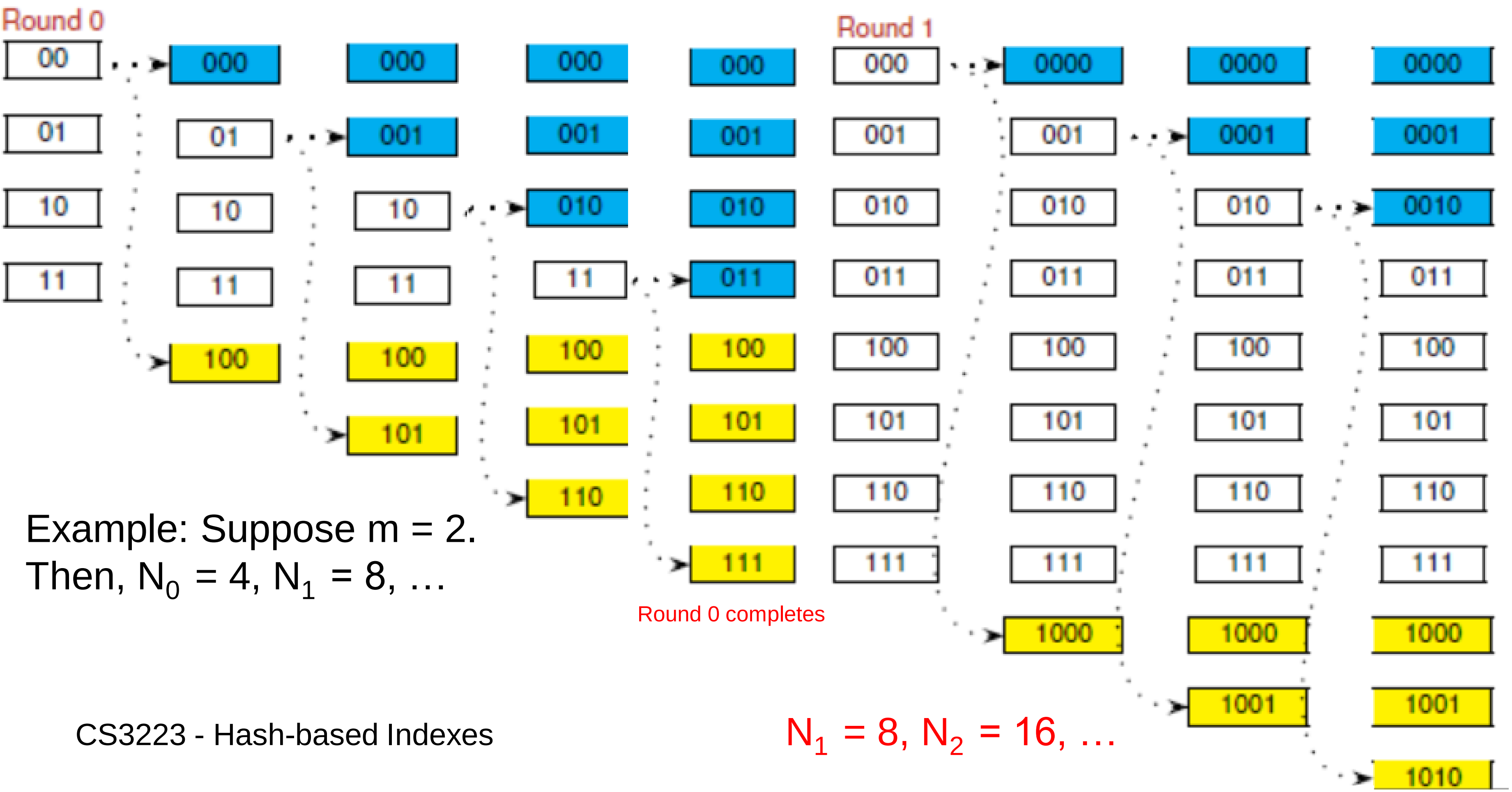
Think of it as re-hashing using 3 bits instead of 2

How to redistribute entries?

- To distribute entries in B_i between B_i, …, B_{N0+i}
  - Use the last $(m+1)^{th}$ bit of h(e.key) to redistribute entry $e \in B_i$
    - If the last $(m+1)^{th}$ bit of h(e.key) is 0, e remains in B_i
    - Otherwise, move e to B_{N0+i}
  - If e is moved to B_{N0+i}, the last (m+1) bits of h(e.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



Example: Suppose m = 2.
Then, $N_0 = 4$, $N_1 = 8$, …

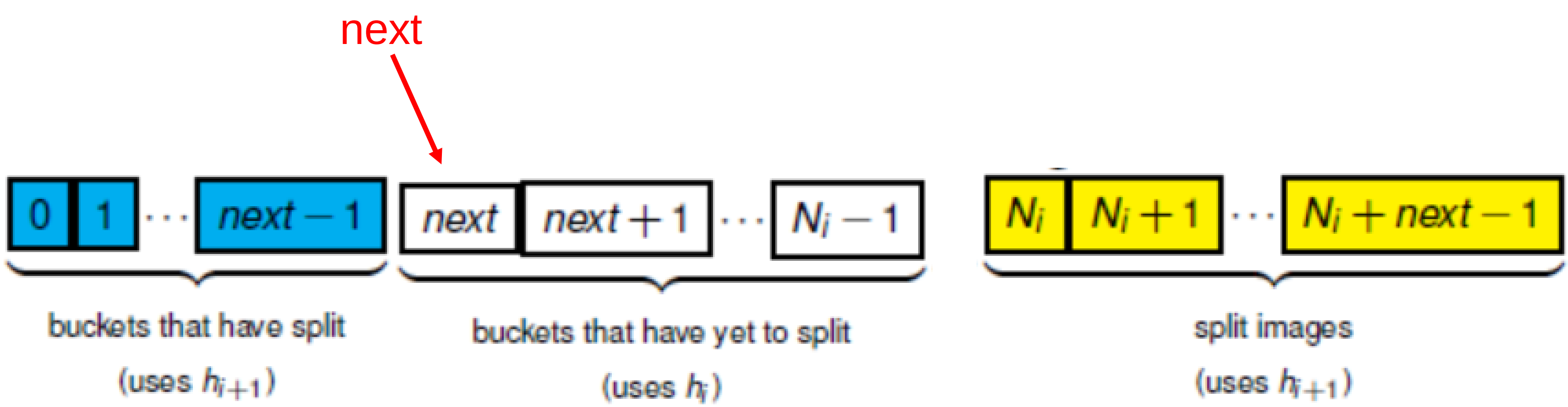
$N_1 = 8$, $N_2 = 16$, …

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
    - next $\in \{0, 1, \dots, N_i - 1\}$
    - When bucket B_{next} is split, its split image is $B_{next+Ni}$
  - Uses a pair of hash functions: h_i and h_{i+1}
    - $h_{i+1}(k)$ = last (m+i) 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:



When to split a bucket?

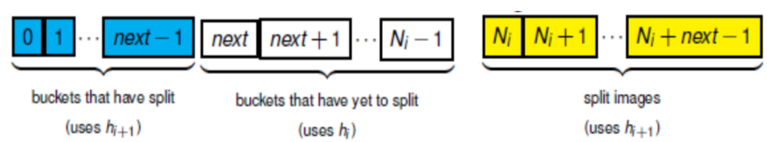
- The time to split the next bucket can be decided with various criteria:
  - Split whenever some bucket/page overflows
  - Split whenever space utilization of file is above some threshold
  - etc
- Overflow pages are still needed since an overflowed bucket might not be split immediately
- Some split 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

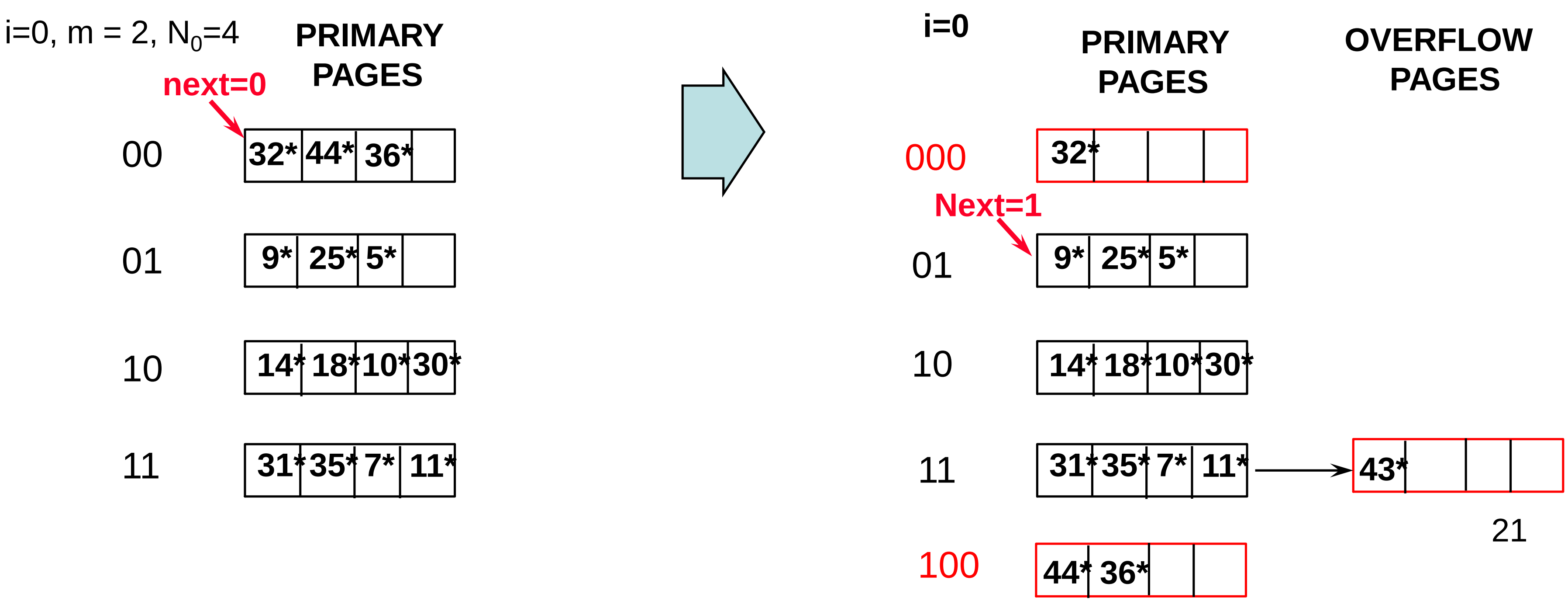
- Find bucket by applying hash function $\mathbf{h}_i$
  - Use $\mathbf{h}_i$ (k) if $\mathbf{h}_i$ (k) ≥ next (why?)
  - Otherwise, apply $\mathbf{h}_{i+1}$



- 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 $B_{next+Ni}$ using $\mathbf{h}_{i+1}$
    - *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!

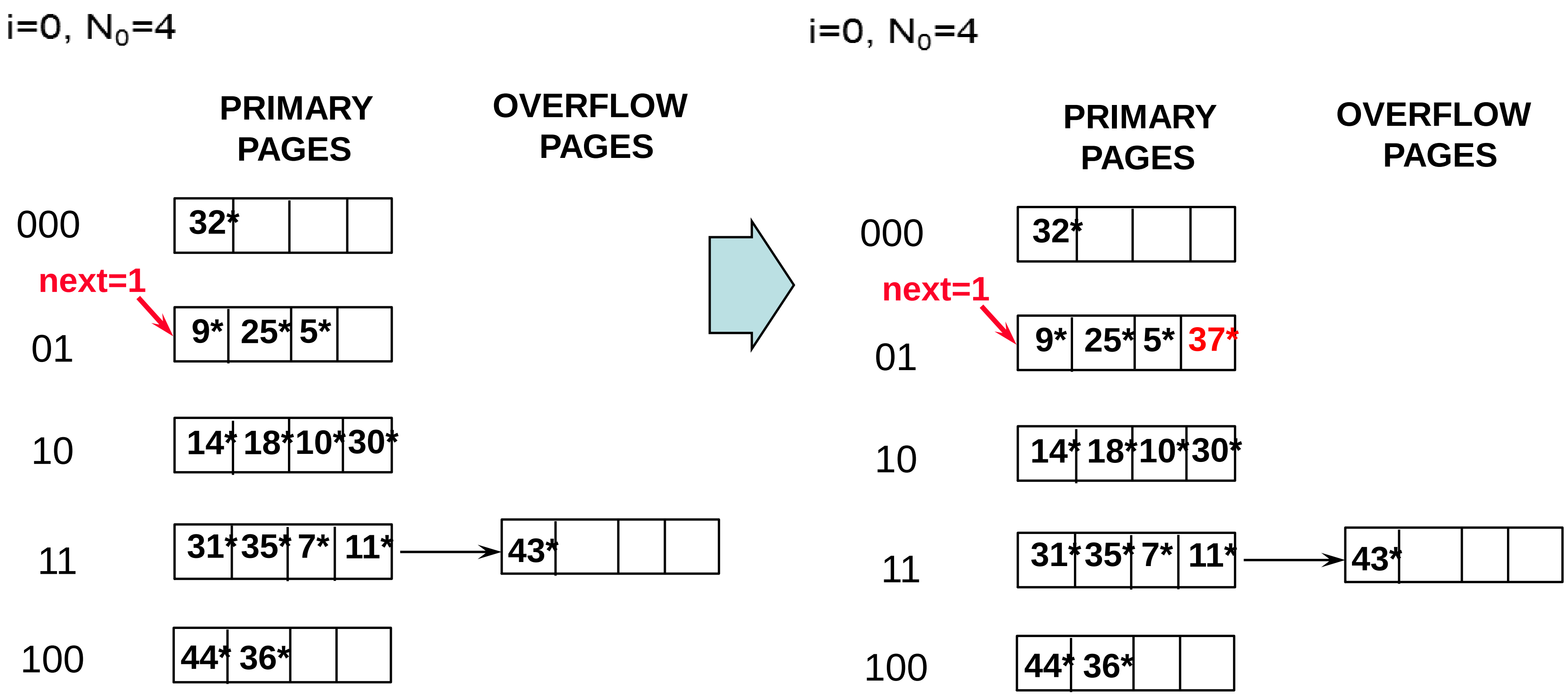
Example of Linear Hashing

- Recall: On split, $\mathbf{h}_{i+1}$ is used to re-distribute entries
- Insert 43* (101011)
  - Bucket B_{11} overflows (but we are not splitting it!)
  - Split bucket B_{00} (32 = 100000, 36 = 100100, 44 = 101100)
  - Increment next to 1
  - Insert 43* into overflow page



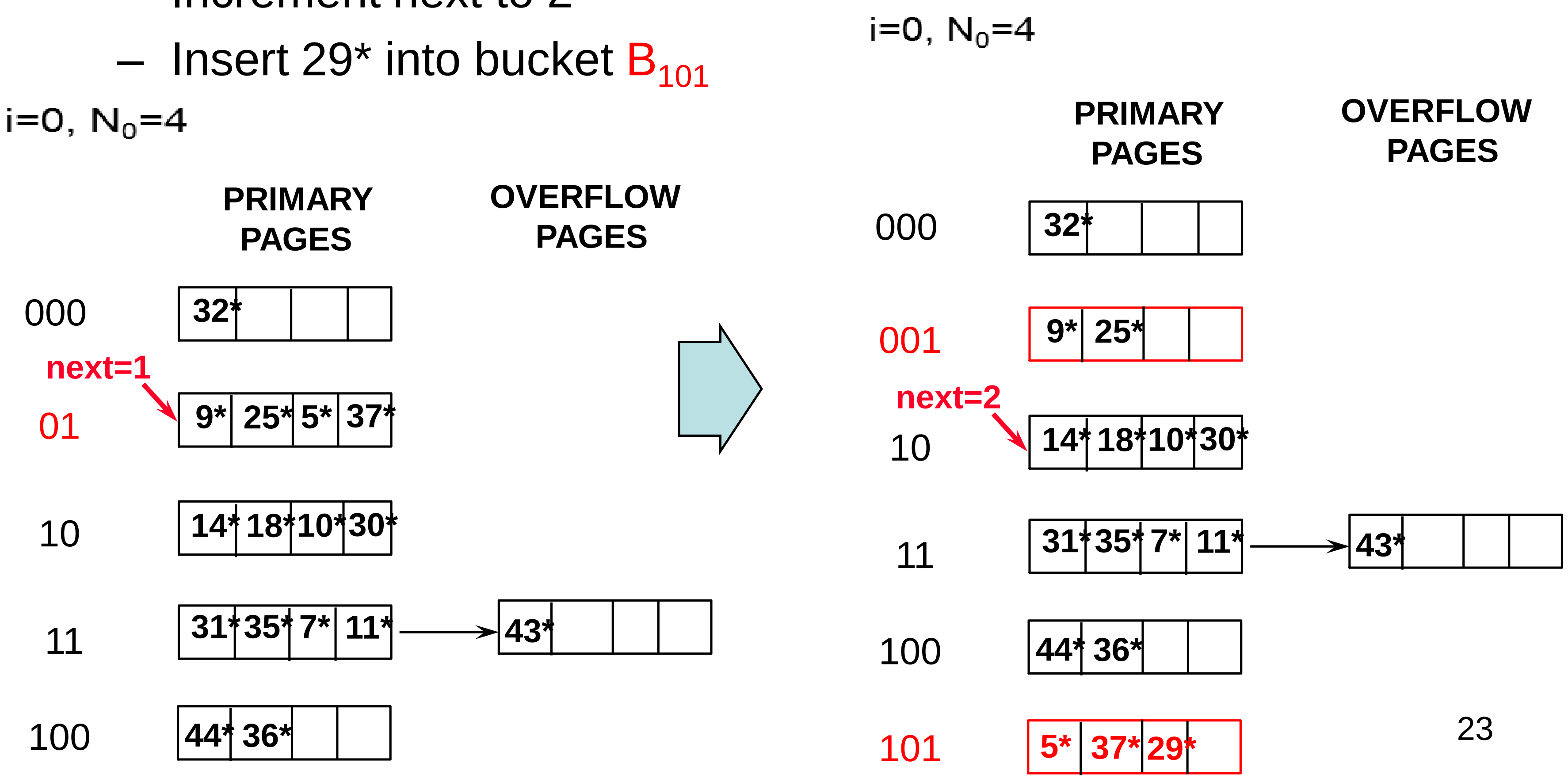
Example of Linear Hashing

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



Example of Linear Hashing

- Insert 29* (011101)
  - 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}



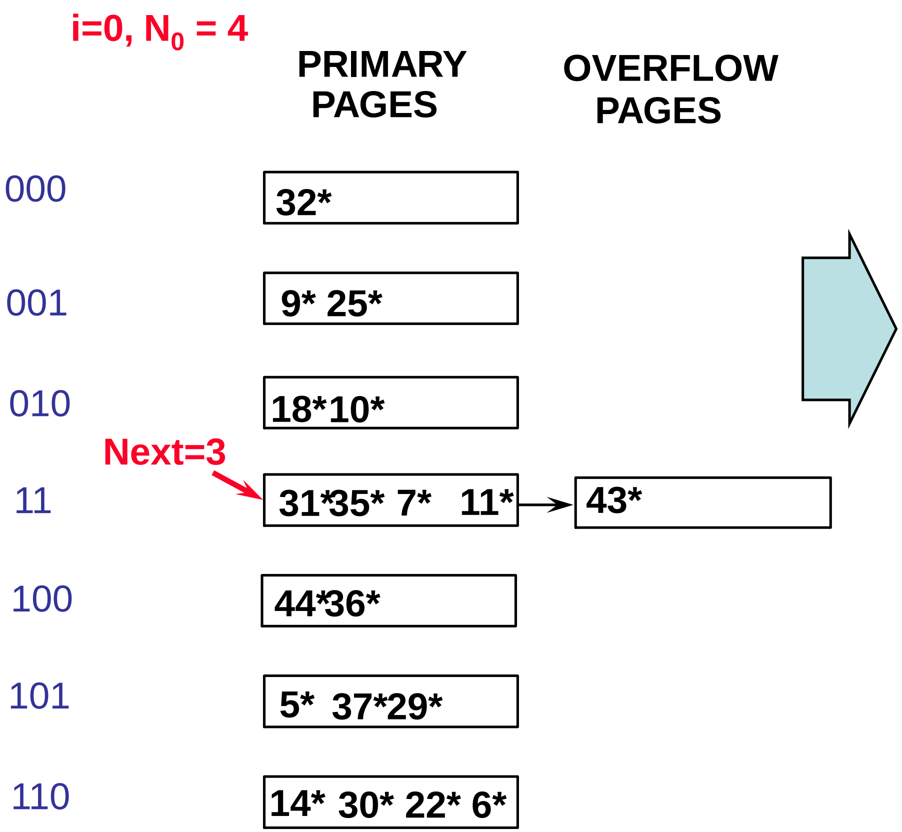
More insertions …

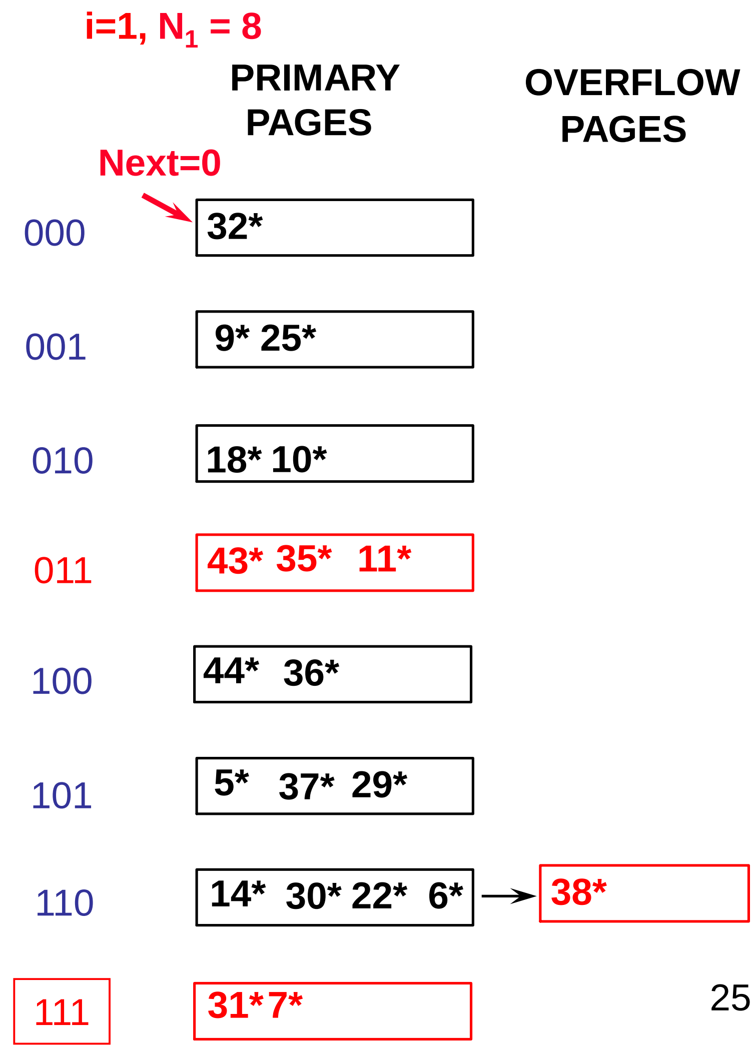
- Insert 22* (010110)
  - Bucket B_{10} overflows
  - Split bucket B_{10} (10 = 1010, 14 = 1110, 18 = 10010, 30 = 11110)
  - Increment next to 3
  - Insert 22* into bucket B_{110}

- Insert 6* (000110)
  - No overflow, no splitting
  - Insert 6* into bucket B_{110}

Example: Insert 38* (End of a Round)

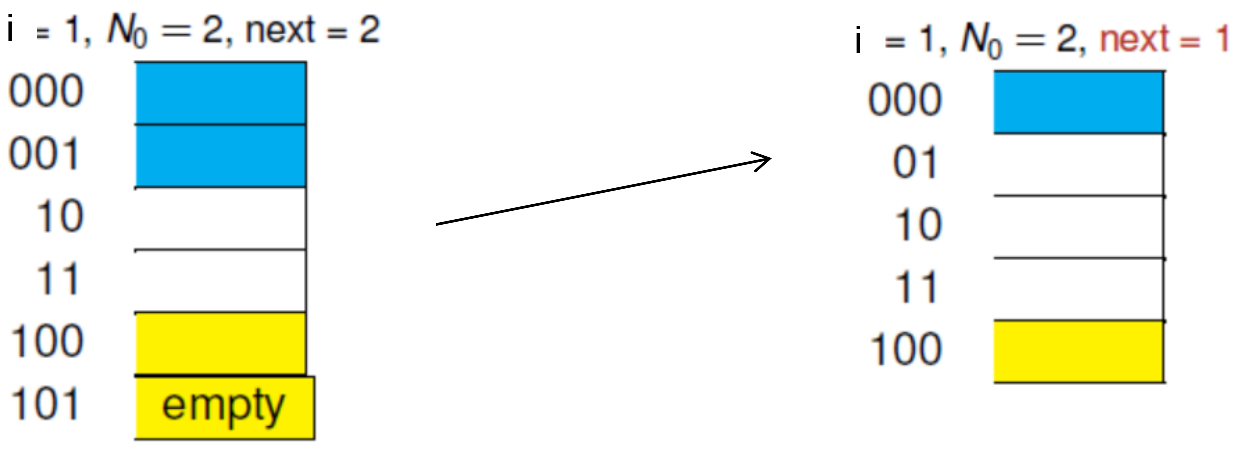
- Insert 38* (100110)
  - Bucket B_{110} overflows
  - Split bucket B_{11} (7 = 111, 11 = 1011, 31 = 11111, 35 = 100011, 43 = 101011)
  - Increment level to 1; reset next to 0
  - Insert 38* into overflow page





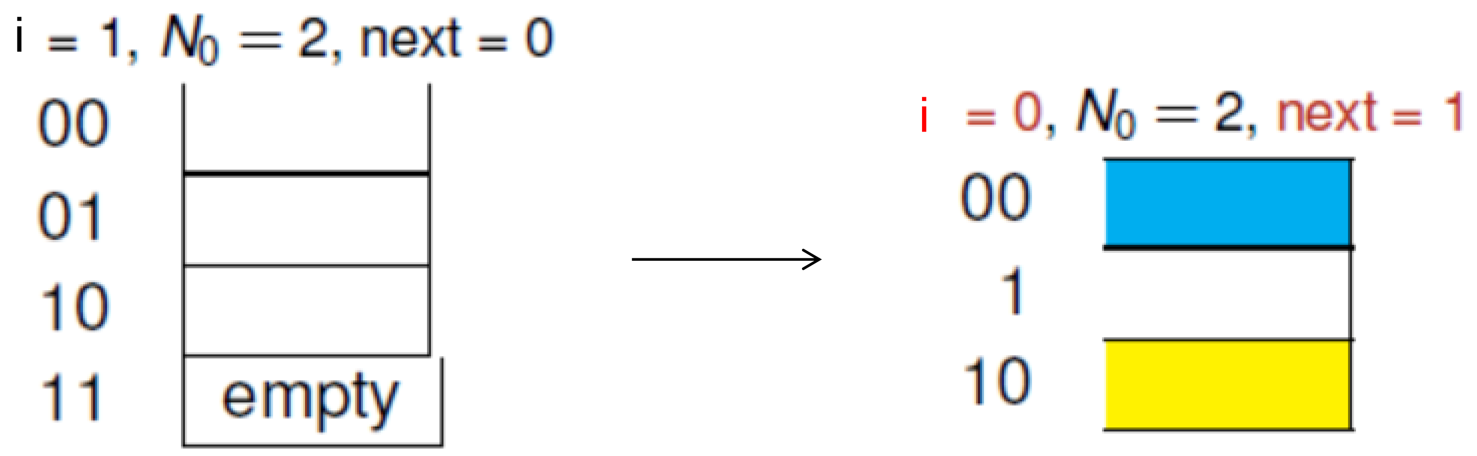
Linear Hashing: Deletion

- Locate bucket and delete entry
- If the last bucket $B_{Ni + next - 1}$ becomes empty, it can be removed
- Case 1: If 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



Linear Hashing: Performance

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

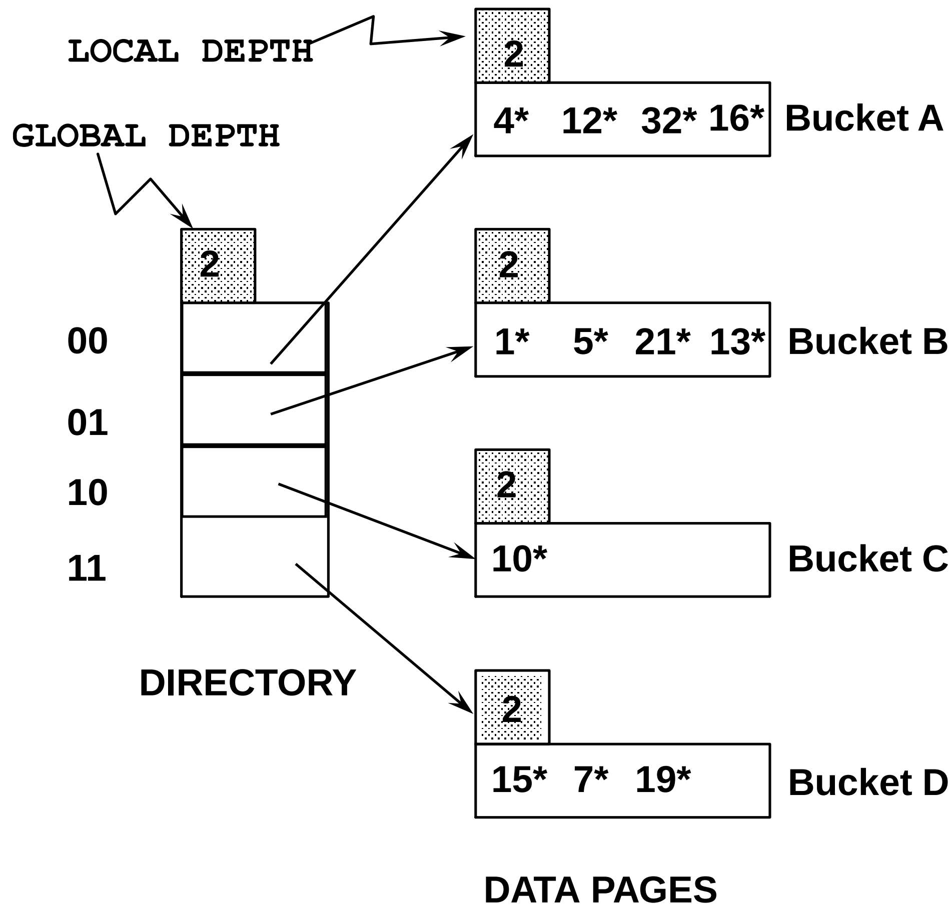
Why not simply double the file each time?

Dynamic Hashing: Extendible Hashing

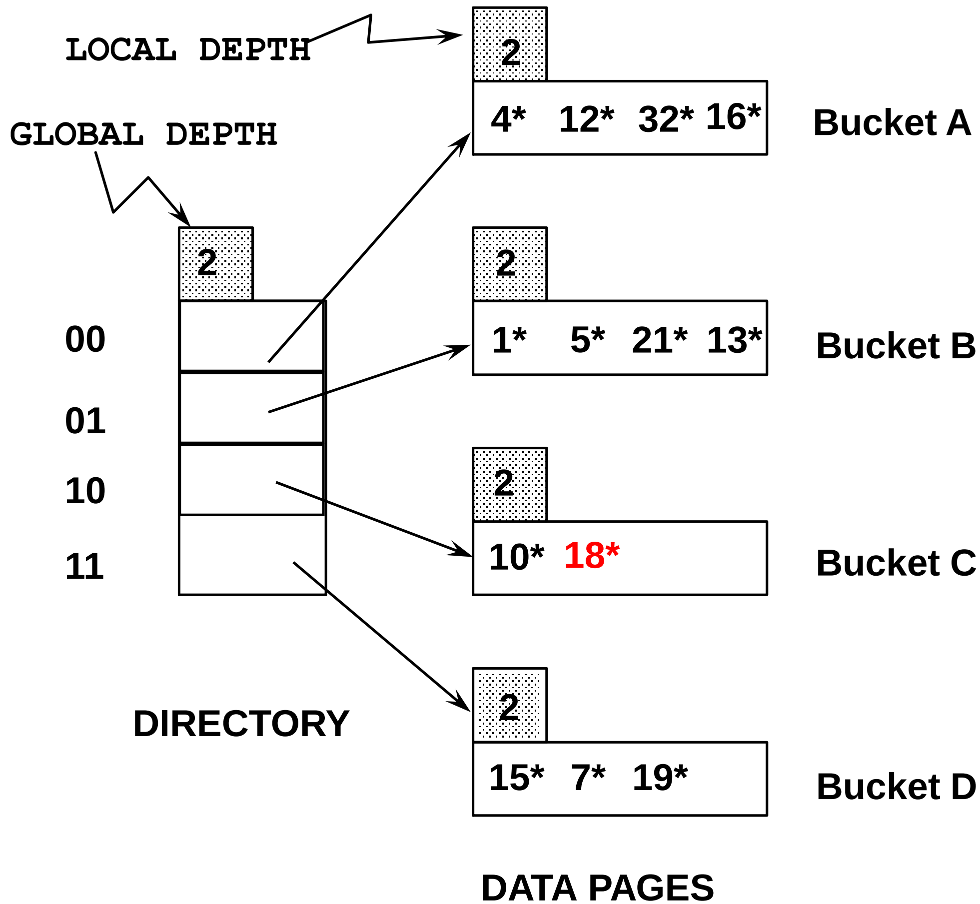
- Handle file growth by *doubling* # of buckets!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory (not the actual buckets), splitting just the bucket that overflows*!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page 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 = binary 101, it is in bucket pointed to by 01

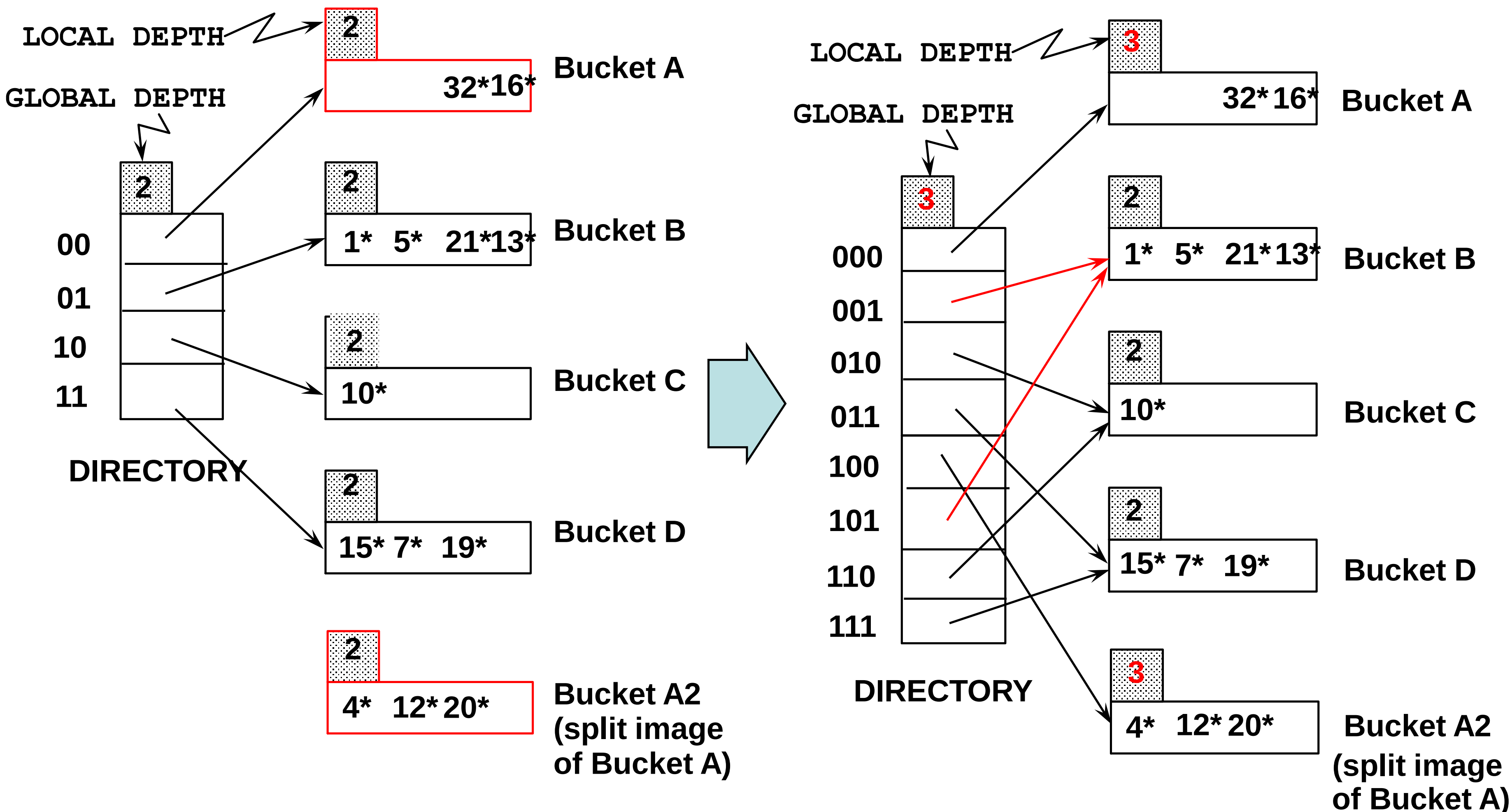


Example: Insert 18 (10010)



What about 20 (10100)?

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

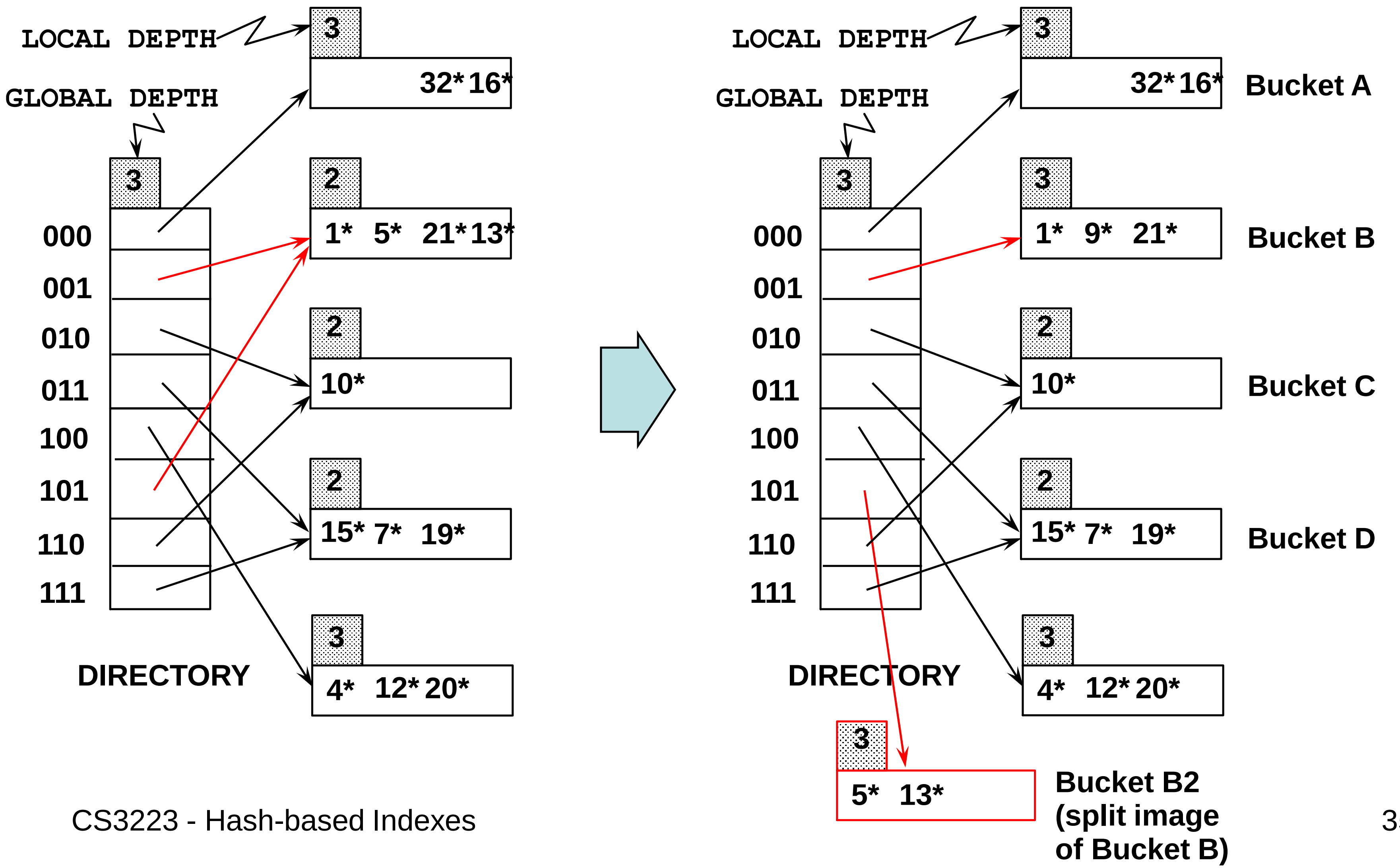


Points to Note

- 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