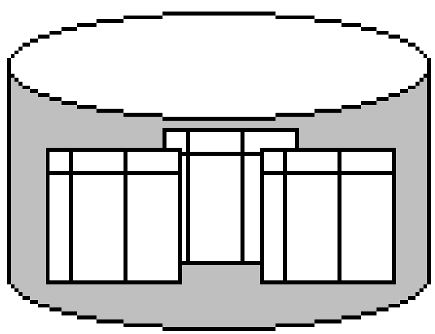
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 3η: Access methods: Hash Indexes

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

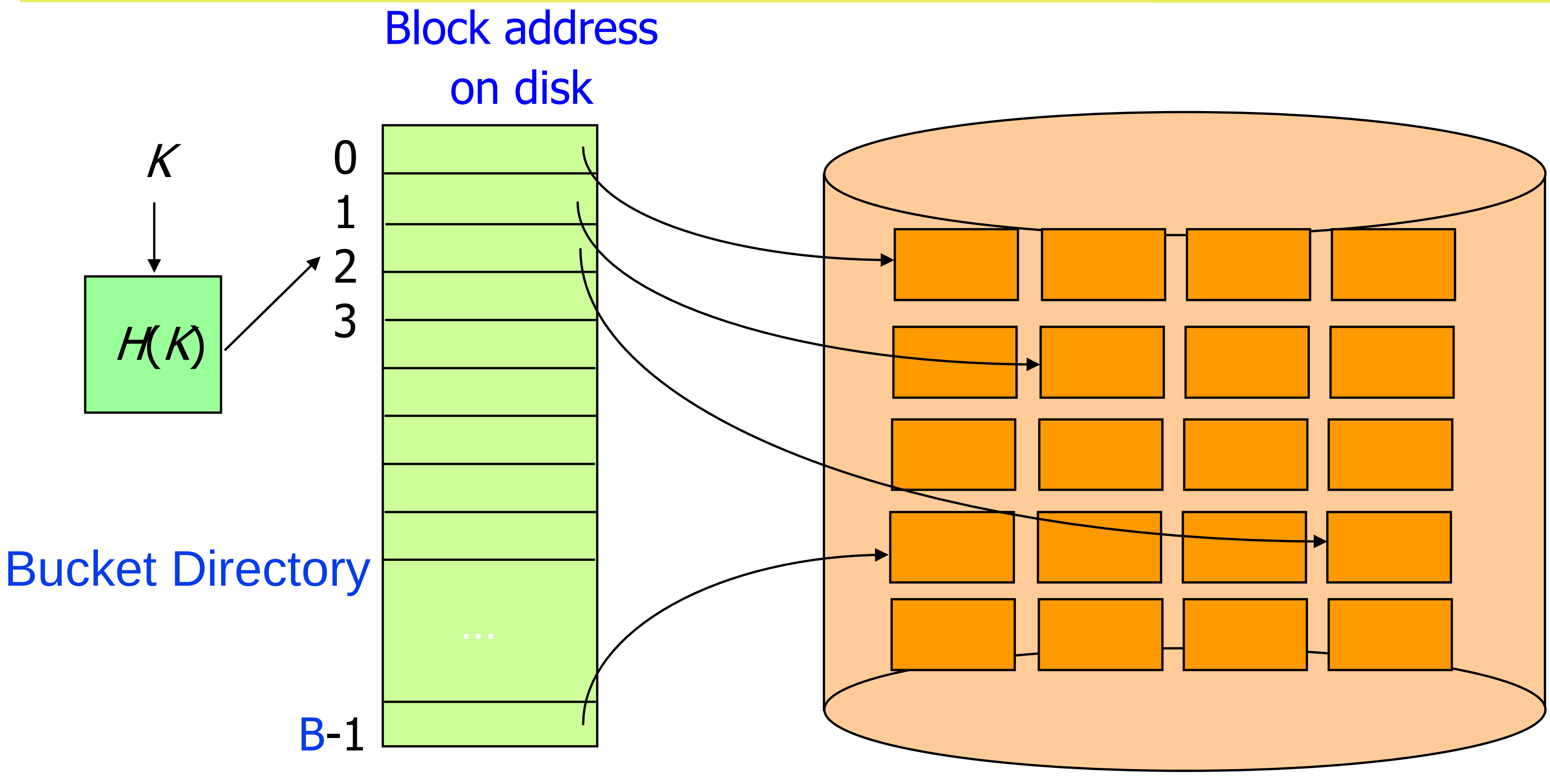
ACCESS METHODS: HASH INDEXES



Hash-based Indexes

- Hash Tables
  - frequently used as main-memory data structures
  - idea: use a hash function **h** taking as input a (hash) key and returning an integer in the range `[0,...,B-1]`; **B** is the number of available buckets
  - organization: a bucket array indexed `[0,...,B-1]` stores the headers of **B** linked lists; if a record has a search key **K**, then it will be stored in the list for the bucket numbered **h(K)**
- Hash Tables in Secondary Storage vs main-memory:
  - Bucket is a unit of storage containing one or more records, typically a disk block
    - Records (or pointers) are placed in the block returned by the hash function
    - Chains of overflow blocks may be added to buckets to accommodate more records
  - Bucket array (directory) consists of block addresses rather than pointers to lists

Hash Tables in Secondary Storage



- Hash file has relative bucket numbers 0 through **B**–1
- Map logical bucket numbers to physical disk block addresses

How do we Choose a Hashing Function ?

- The hash function used should be such that the resulting integer appears to be a random function of the search key
  - thus, buckets will have more or less equal numbers of records and the average time to locate a record will be improved
  - but still one bucket may receive more records than another because some values are more "popular" than others (data skewing)
- Typical hash functions perform computation on the internal binary representation of the search-key
- A function commonly used to hash on integer-valued keys is `K mod B`, which yields an integer in the range 0 to **B**−1
  - **B** is often chosen to be a prime number
  - in certain cases it is convenient to use a power of 2
- For character string keys, characters may be treated as integers (integers are summed up, divided by **B**, and the remainder of the division is taken as the result of the hash function)

Choice of Good Hash Functions

- Desired property: expected number of record keys/bucket is the same for all buckets (uniform distribution) and is relatively small; ideally all buckets consist of only one page
  - B must be large enough to minimize the occurrence of overflow chains
  - B must not be so large that bucket occupancy is small and too much space is wasted
- The worst hash function maps all search-key values to the same bucket
  - this makes access time proportional to the number of search-key values in the file
- Potentially good hash functions: division or multiplication

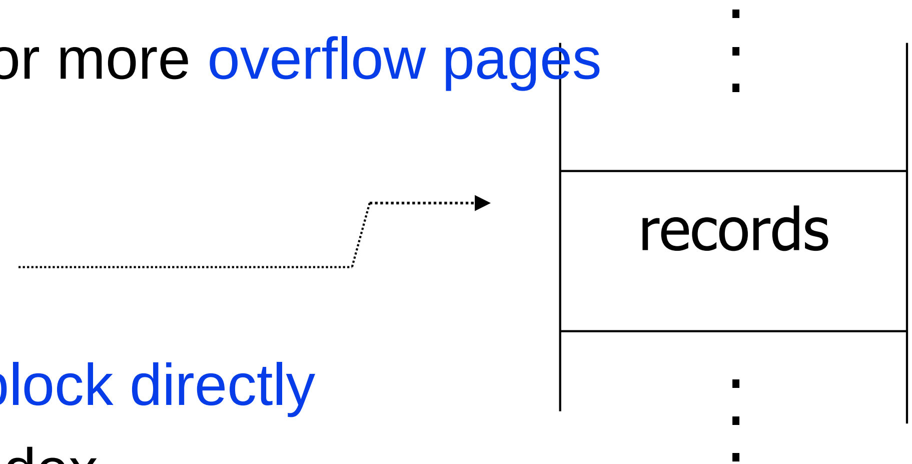
  `h(K) = (a*K + a') mod B,`

  `h(K) = [ fractional-part-of ( K * ϕ ) ] * B,`
  - ϕ: golden ratio (0.6180339887 = (sqrt(5)-1)/2)
  - advantage: B (size of hash table) need not be a prime number but ϕ must be irrational
  - Read Knuth Vol. 3 for more on good hashing functions

Hash Tables in Secondary Storage: Two Alternatives

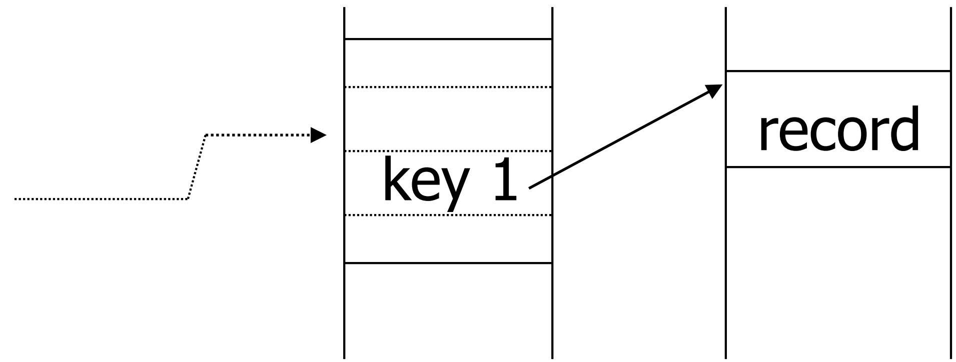
- Index is a collection of buckets
  - Bucket = primary page plus zero or more overflow pages

(1) key -> h(key)



❶ Hash value determines the storage block directly to implement a clustered primary index

(2) key -> h(key)

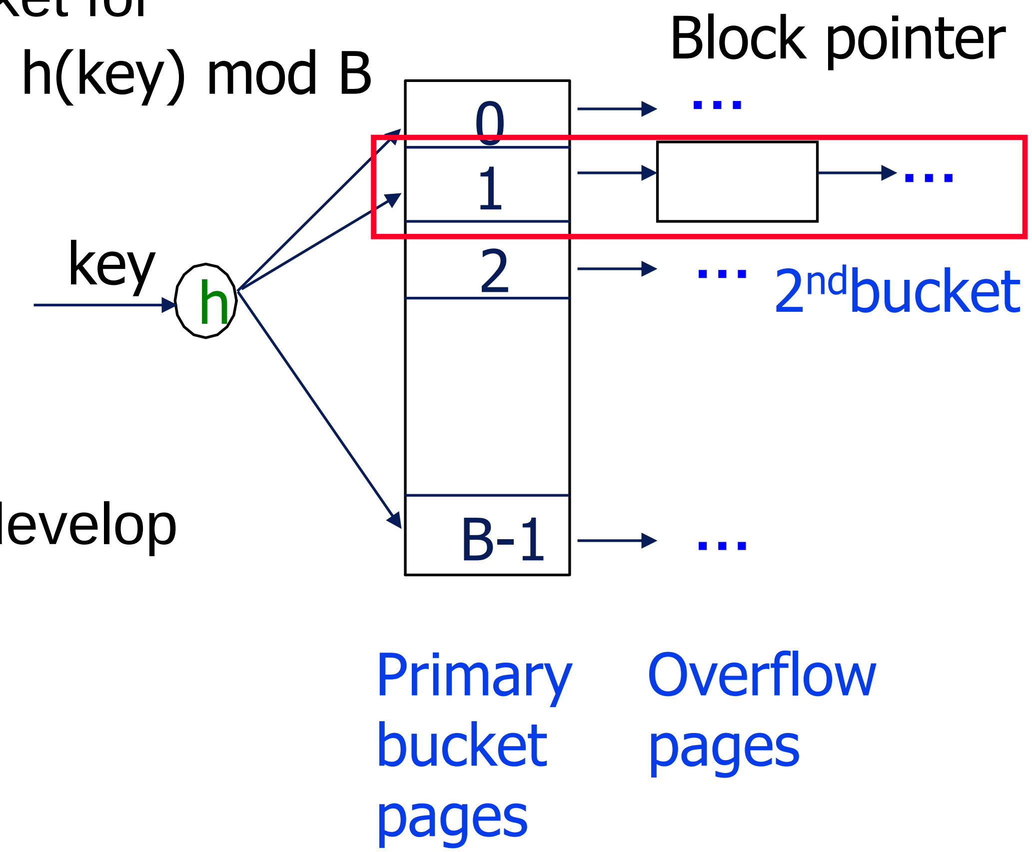


❷ Records located indirectly via index buckets for a secondary index

Index File

Static Hash Tables

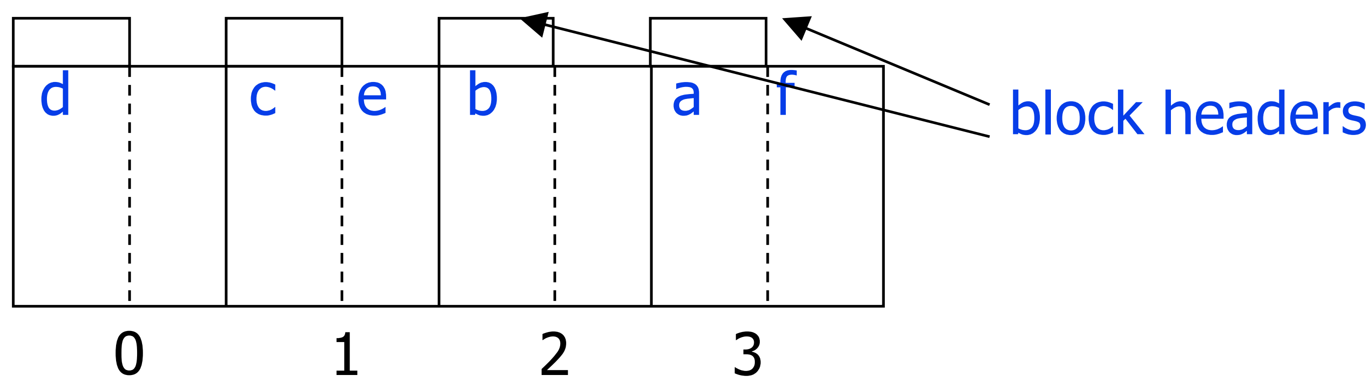
- Buckets contain data entries
  - Buckets of records, not pointers
  - Need to search within the bucket for the record with the key
- One primary page per bucket
  - Fixed, allocated sequentially
  - Additional overflow pages
- Fixed number of buckets
  - File grows
    - Long overflow chains can develop
    - Performance degrade
  - File shrinks
    - Space wastage
- Rehash file periodically



Static Hash Tables

- Each bucket is identified by an address **a**
  - Bucket at address **a** contains all index entries with search key v such that **h(v)=a**
- Assumption: the address of the first block for bucket i is known
  - may use a main-memory directory of pointers to blocks
  - alternatively, the first block for each bucket may be placed in fixed, consecutive locations on disk
- Example: h:{a,b,c,d,e,f} → {0,1,2,3}

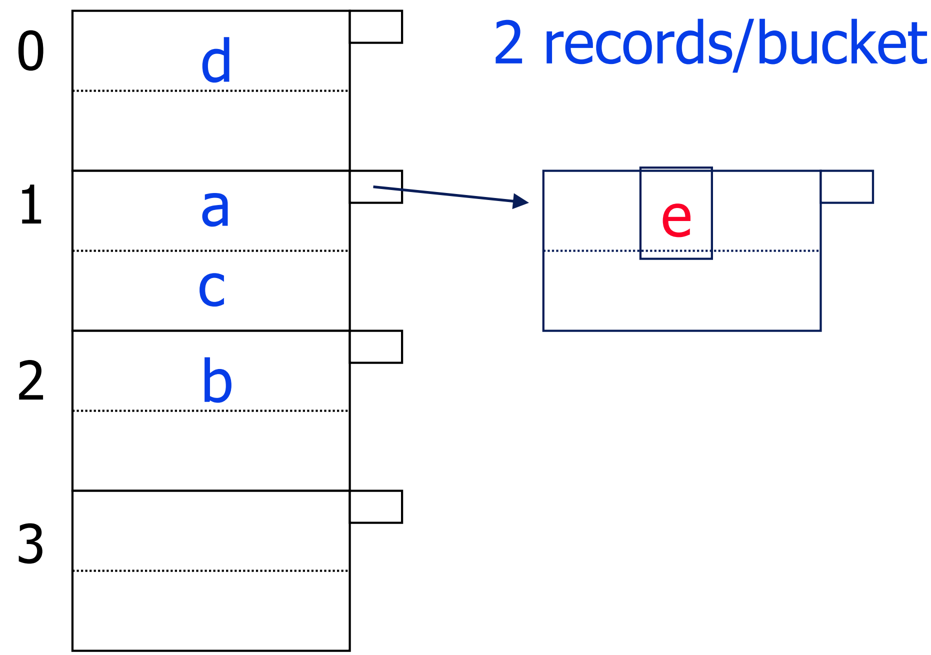
  h(a) = h(f) = 3, h(b) = 2, h(c) = h(e) = 1, h(d) = 0



Inserting into Static Hash Tables

- To insert record with key K:
  - compute h(K)
  - if bucket h(K) has space, insert the record in the block
  - if not, insert it into one of the overflow blocks associated with the bucket
  - if none of them has space, add a new overflow block and store the record there

INSERT:
h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0



Insert: h(e) = 1

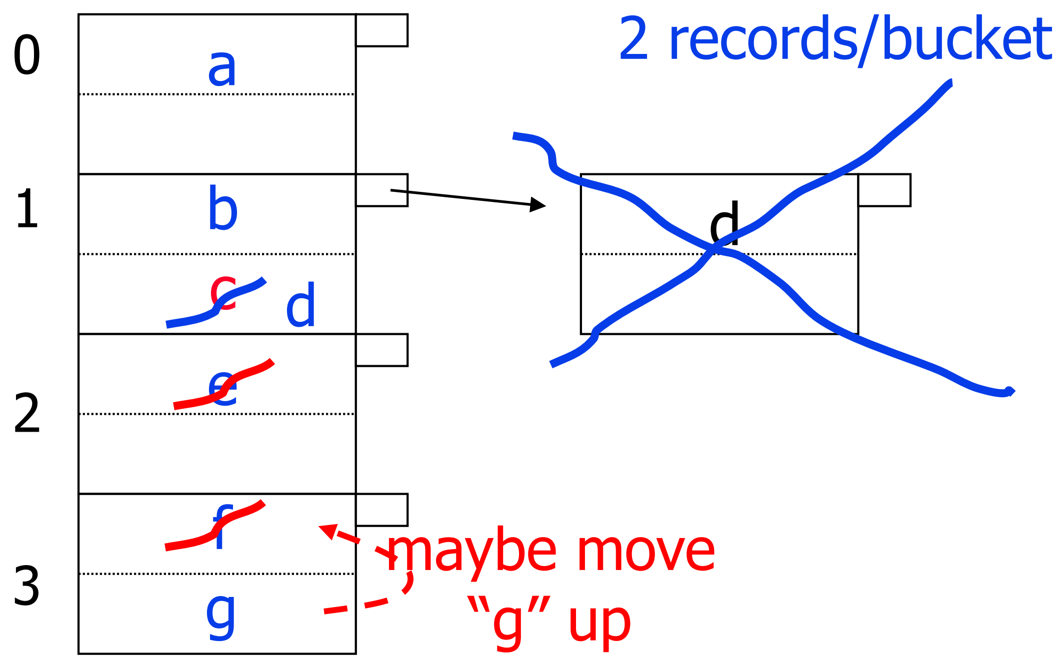
Deleting from Static Hash Tables

- If record with search key K is to be deleted:
  - search bucket h(K) for records with value K
  - delete record(s) found
  - optionally consolidate blocks on a chain after deletion
    - not always a good idea to merge blocks of a chain: repeated insertions / deletions may cause blocks to be created / destroyed at each step

Delete:

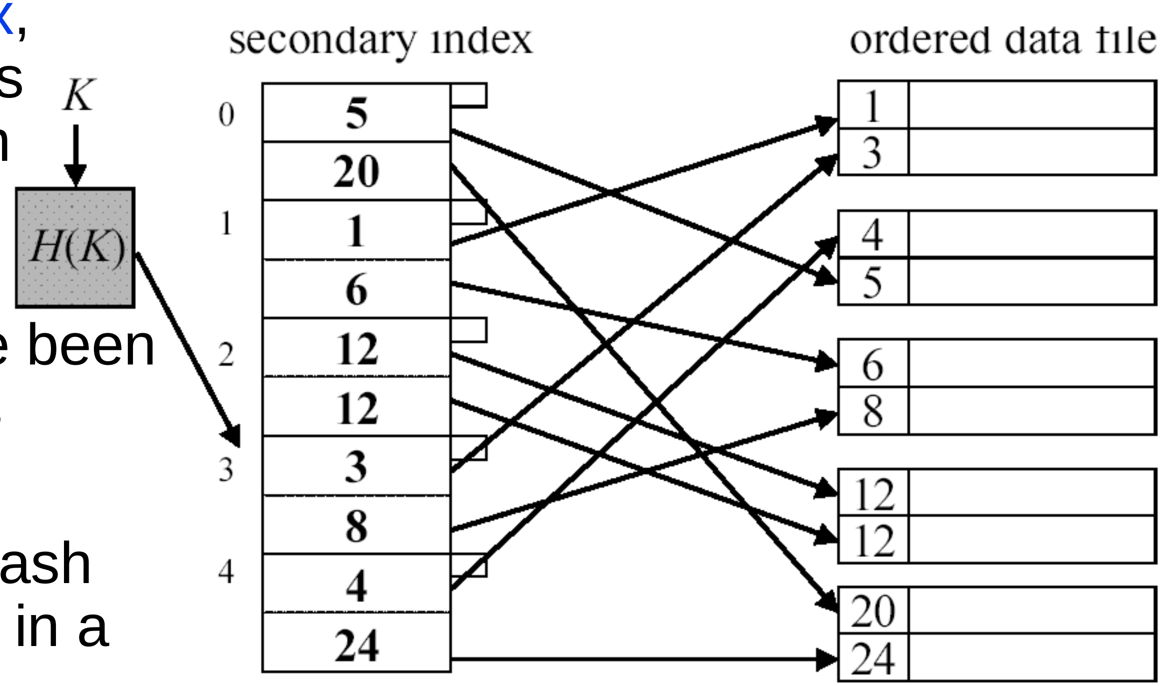
e
f

c



Applications of Hash Tables

- Hashing can be used for both primary and secondary indexes
- For a clustered primary index, no separate index structure is required as the hash function dictates the physical organization of the file
  - The examples so far have been clustered primary indexes
- For a secondary index, the hash function computes a location in a secondary index file
  - This location contains a bucket of physical addresses of records that hash to this logical location



How Full should Buckets be?

- Space utilization:

$$\textit{utilization} = \frac{\text{\# keys used in a bucket}}{\text{total \# keys that fit in a block}}$$

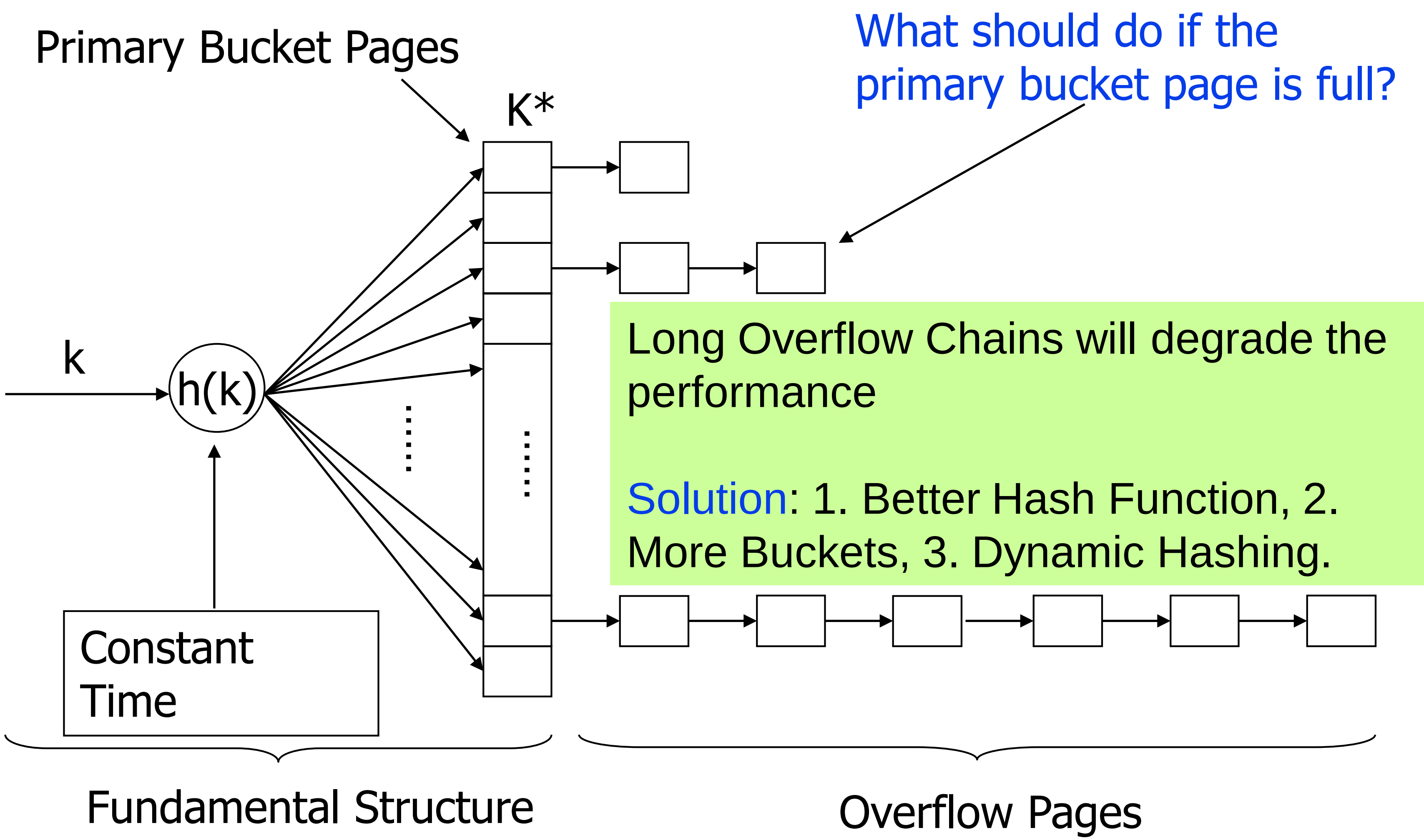
  - If too small: wasting space
  - If too big, overflows significant

- A good space utilization:
  - depends on how good the hash function is
  - Rule of thumb: usually between 50% and 80%
    - If U < 50%, space is underutilized
    - if U > 80%, overflows become significant

Efficiency of Static Hash Tables

- Searching for a key v:
  - ❶ Evaluate *h(v)*
  - ❷ Fetch bucket at *h(v)*
  - ❸ Search bucket
- In an ideal case, we have enough buckets so that each bucket consists of a single block
  - Lookup: one disk I/O
  - Insert / Delete: two disk I/O's
- In such a case, static hash tables (fixed **B**) perform better than simple dense or sparse indexes, or even B+-trees
- Dynamically growing files produce overflow chains, which negate the efficiency of the hashing algorithm
  - many blocks per chain may need to be searched
  - hence, `#blocks/bucket` should be as low as possible

Static Hash Tables



Long Overflow Chains will degrade the performance

Solution: 1. Better Hash Function, 2. More Buckets, 3. Dynamic Hashing.

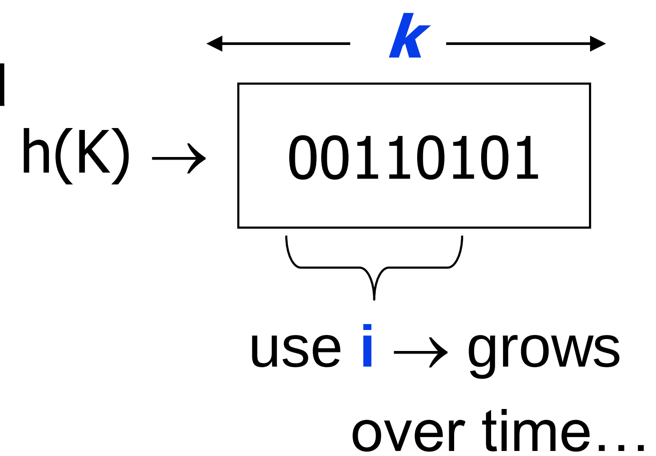
Dynamic Hash Tables

- Bucket (primary page) becomes full
  - Why not re-organize the file by doubling # of buckets?
    - Reading and writing all pages is expensive!
- Idea: the hash file size can grow and shrink "on the fly" in response to the size of the data
  - **B** is allowed to vary
  - Trick lies in how hash function is adjusted!
- Two types of dynamic hashing:
  - Extensible: uses a directory that grows or shrinks depending on the data distribution
    - No overflow buckets
  - Linear: No directory
    - Splits buckets in linear order, uses overflow buckets

Extensible Hashing

- First Idea
  - Use a family of hash functions based on **h**:
    $h_k(key) = h(key) \bmod 2^k$
    - The range of hash function has to be extended to accommodate additional buckets (use a sequence of **k** bits of **h(key)** for some large **k**, e.g., 32)
  - At any given time a unique hash, h_k, is used depending on the number of times buckets have been split
    - bucket addresses use fewer bits (say **i** bits from the beginning of the sequence); hence, directory will have 2^i entries
    - each bucket stores **j**, indicating the number of bits used for placing the records in this block ($j \leq i$)

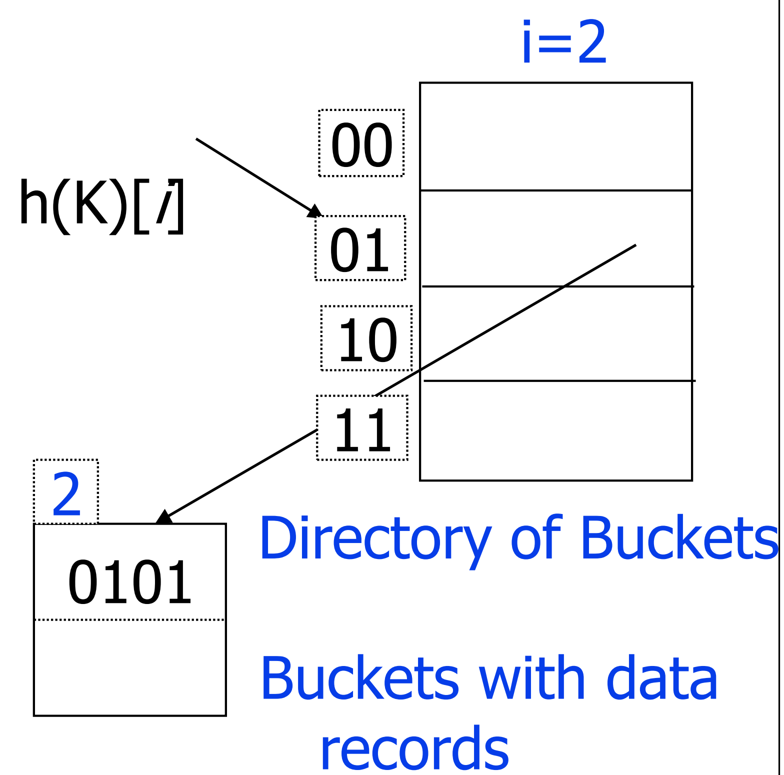
Use **i** of **k** bits output by hash function



Extensible Hashing

- Second Idea:
  - Use directory of pointers to bucket blocks, i.e., introduce a level of indirection
  - The directory's size is a power of 2 and may vary;
    - double # 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
  - Certain buckets may share a block (if the total number of records of the buckets fit in a block)

Use directory of pointers



For simplicity, "0101" represents a record whose key is hashed to "0101"

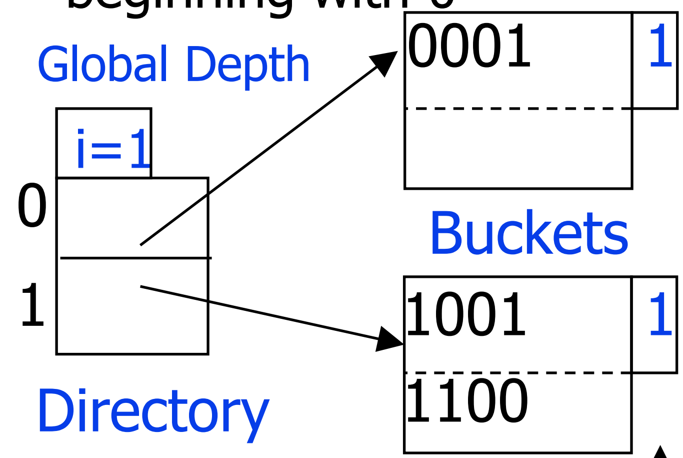
Alternative ❷

Extensible Hashing

- Global depth of directory (i)
  - First i bits of a binary number to tell which bucket an entry belongs to
  - Number of bits needed to express total number of buckets
  - E.g., i=2=>4 buckets; i=3=>8 buckets
  - Increase i by 1 after a directory split
  - Distribute entries across a bucket and its split image based on i th bit
- Local depth of bucket (j)
  - Hash values of data entries in bucket agree on the first j bits
  - Increase j by 1 after a bucket split
  - Assign new j to split image
  - If bucket with local depth j = global depth i is split, then double the directory

- Example: k=4,i=1

Holds records whose key hashes to a sequence beginning with 0

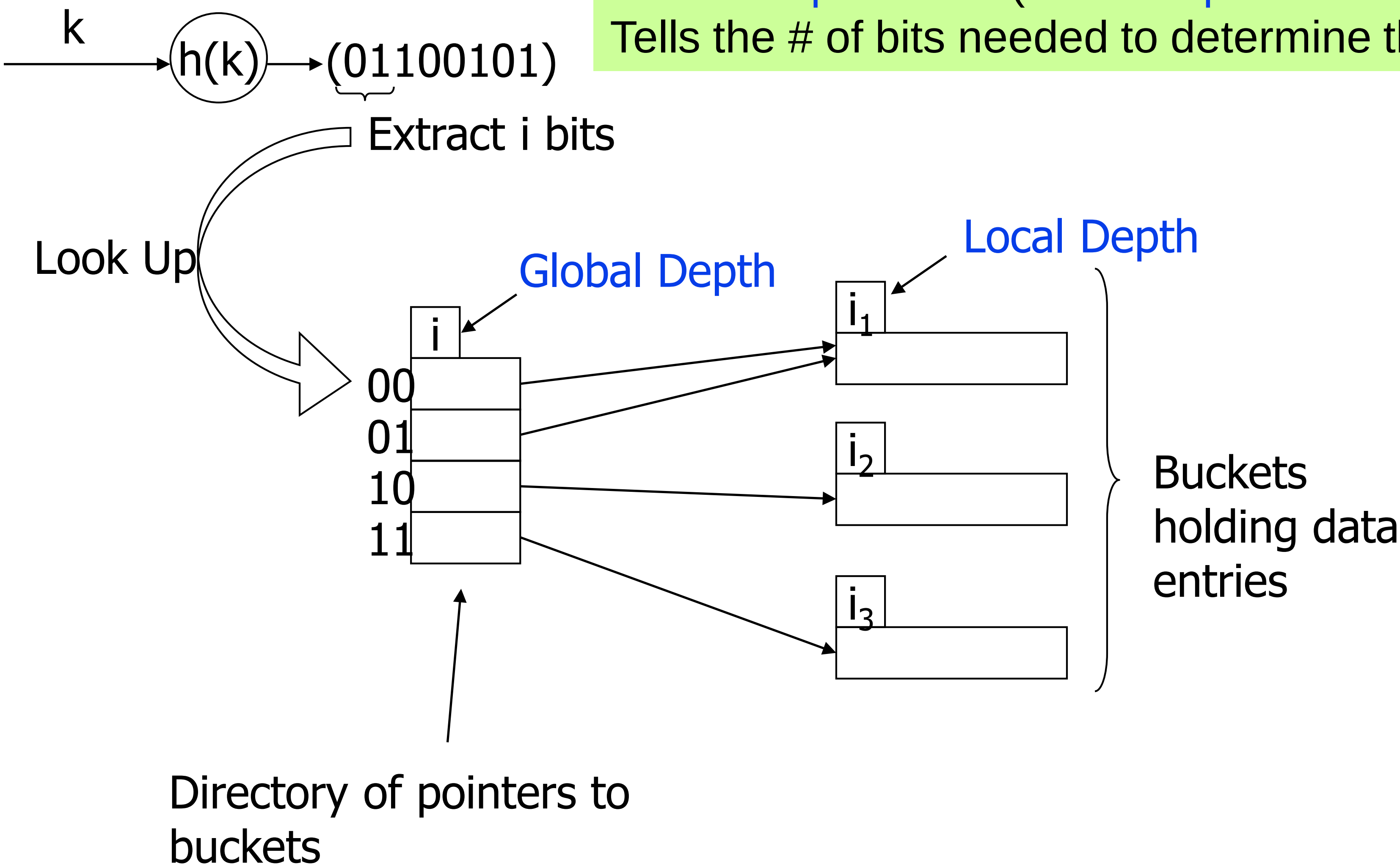


Holds records whose key hashes to a sequence beginning with 1

Local Depth: Number of bits used to determine membership of records in block

Extensible Hashing

Global Depth = max(Local Depth of all buckets)
Tells the # of bits needed to determine the address



Insertion into Extensible Hash Tables

- To insert a record with key **K**:
  - compute **h(K)**; take the first i bits of the sequence returned and follow the pointer in the directory indexed by these i bits (global depth), leading to block **b**
  - if there is room, place record in **b**
  - if not, then, let j be the local depth stored in the block header **b**:
    1. If **j < i** (local < global), then:
       a) Split **b** in two
       b) Distribute records in **b** to the two blocks based on the value of the (j+1)-st bit: those with 0 stay in **b** and those with 1 go to the new block
       c) Update header of each block with the split image j+1
       d) Adjust pointers in directory so that entries point to the correct block depending on their (j+1)-st bit

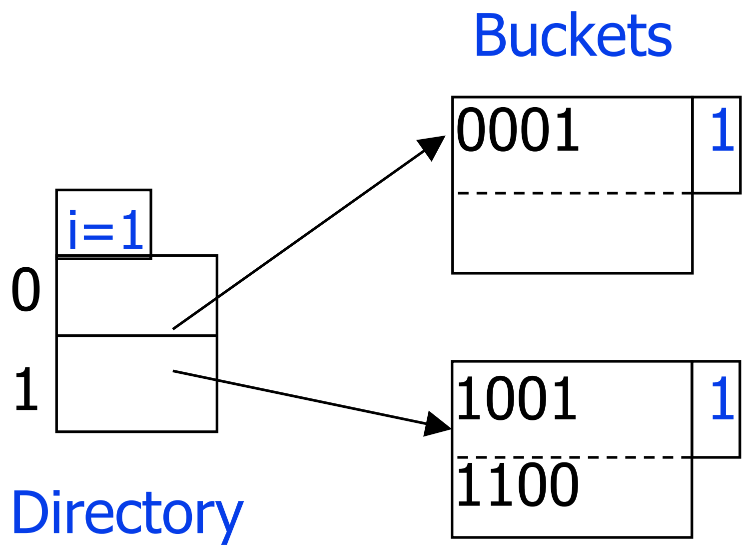
Insertion into Extensible Hash Tables

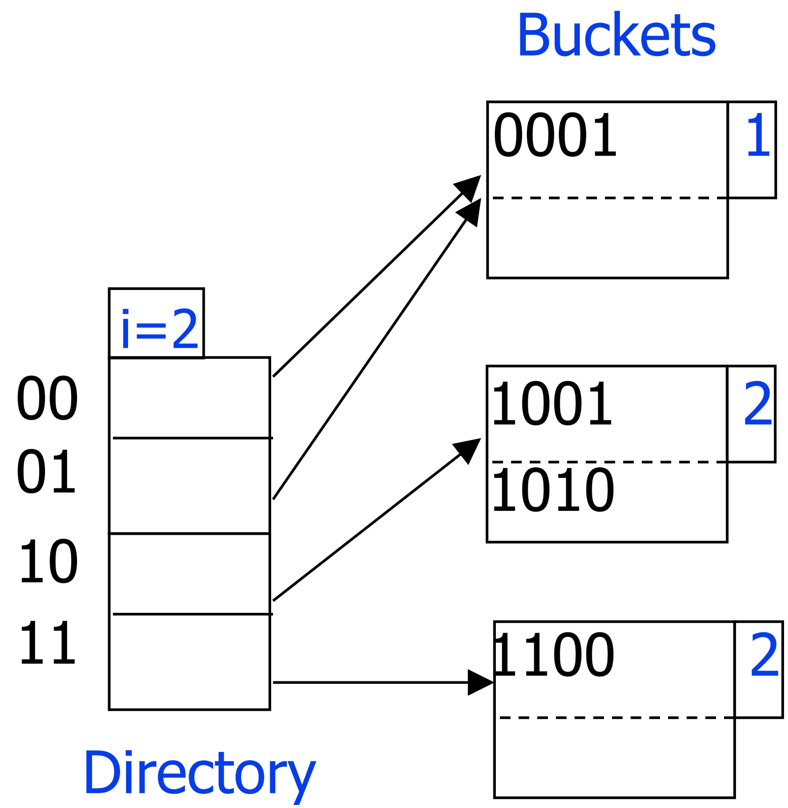
- It may so happen, that examining the `j`-th bit may send all records to one block
  - In this case, the process is repeated with the next higher value of the split image `j`

2. If **`j = i`** (local = global), then:
   a) increment `i` by 1; double the size of the directory so it now has 2^{i+1} entries
   b) let **`w`** be a sequence of `i` bits indexing the entries of the previous directory
      - Then w_0 and w_1 each point to the same block that **`w`** pointed to (entries share the block but the block hasn't changed)
   c) As in case 1, split **`b`** (since **`i>j`**)

Insertion into Extensible Hash Tables

- Example: insert record with value hashing to 1010 in the table

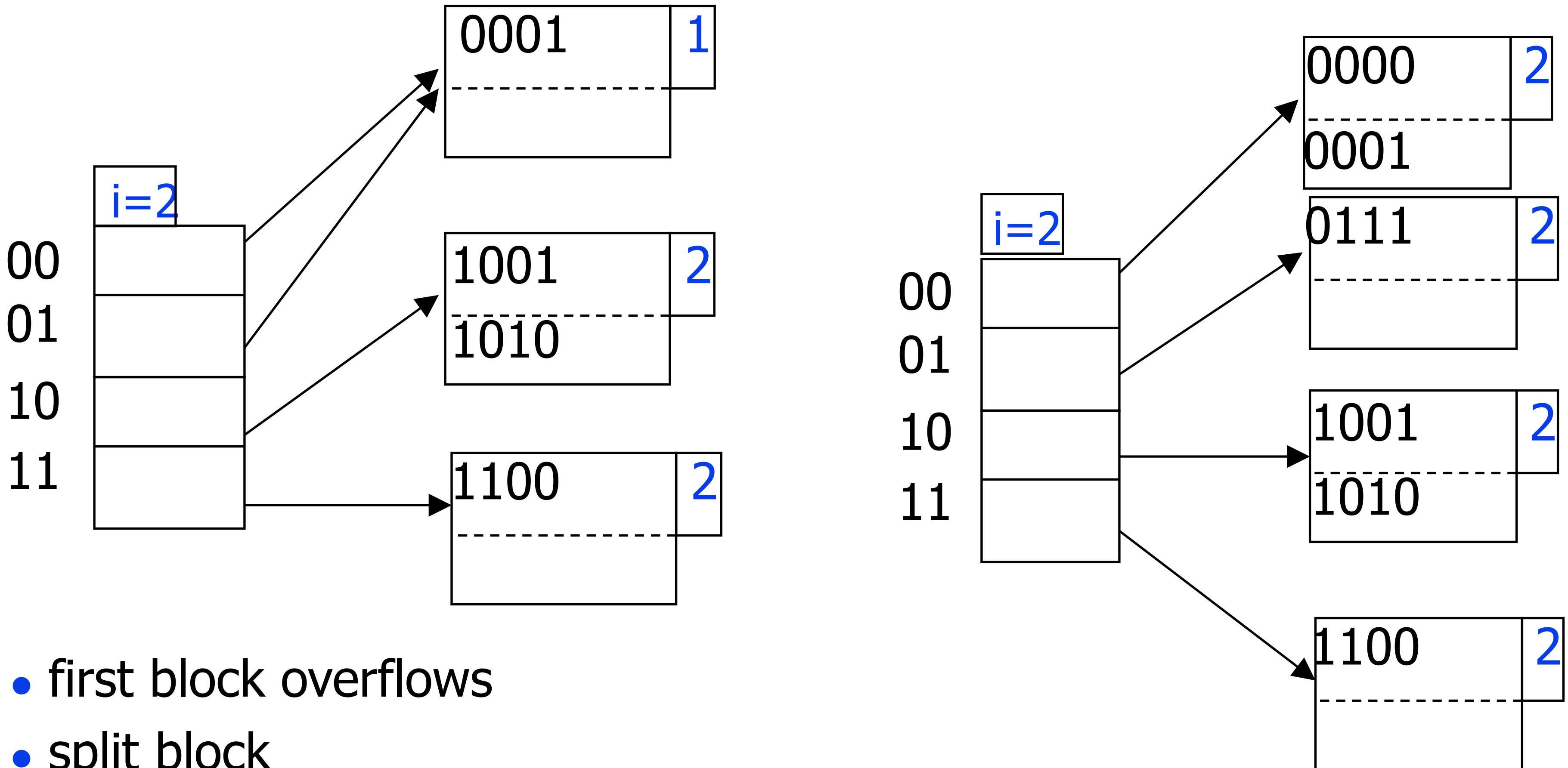




- record must go on the 2nd block
- block is full and must be split
- `j = i = 1; i++`

Insertion into Extensible Hash Tables

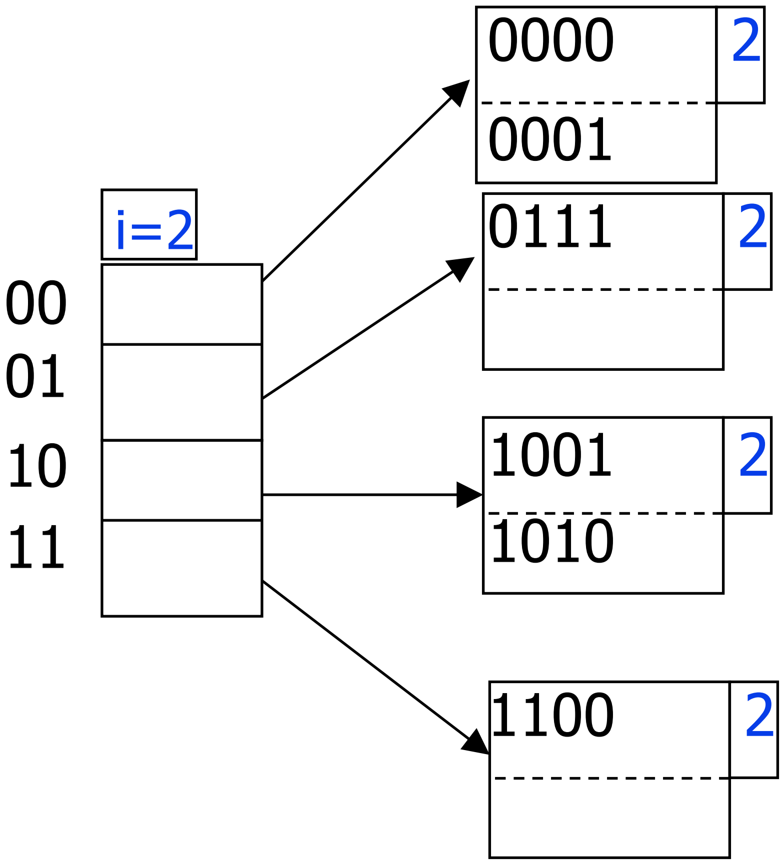
- Example: insert records with values hashing to 0000 and 0111

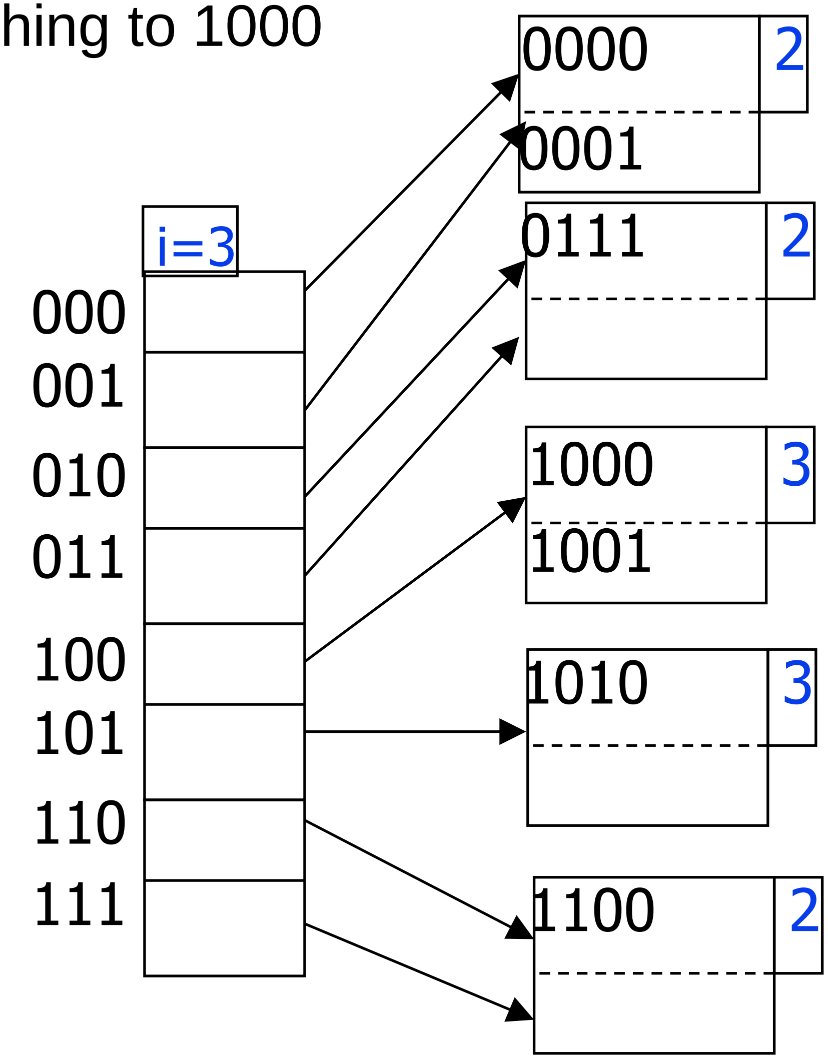


- first block overflows
- split block
- `j < i; j++`

Insertion into Extensible Hash Tables

- Example: insert record with value hashing to 1000





- block for 10 overflows
- `j = i = 2; i++`

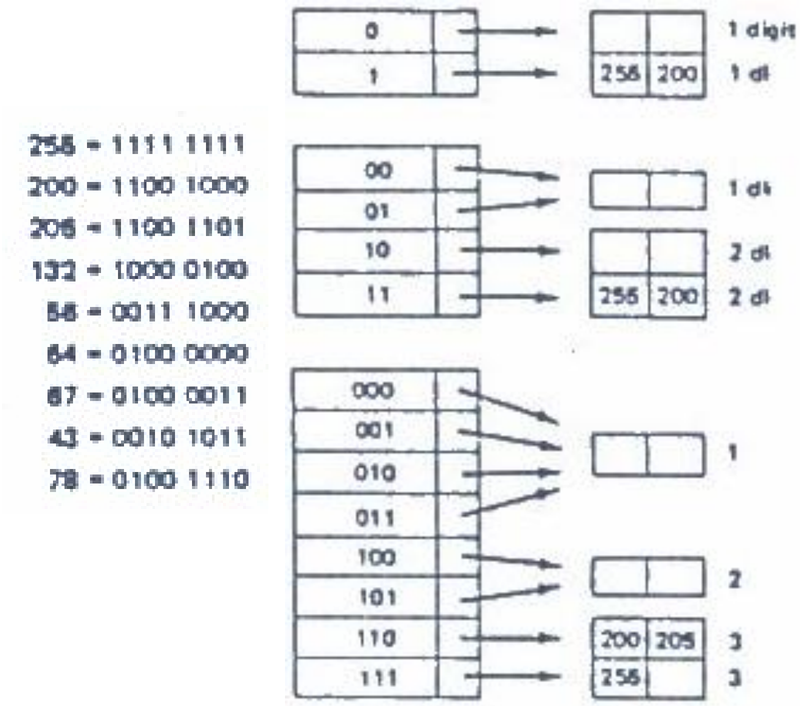
Deletion from Extensible Hash Tables

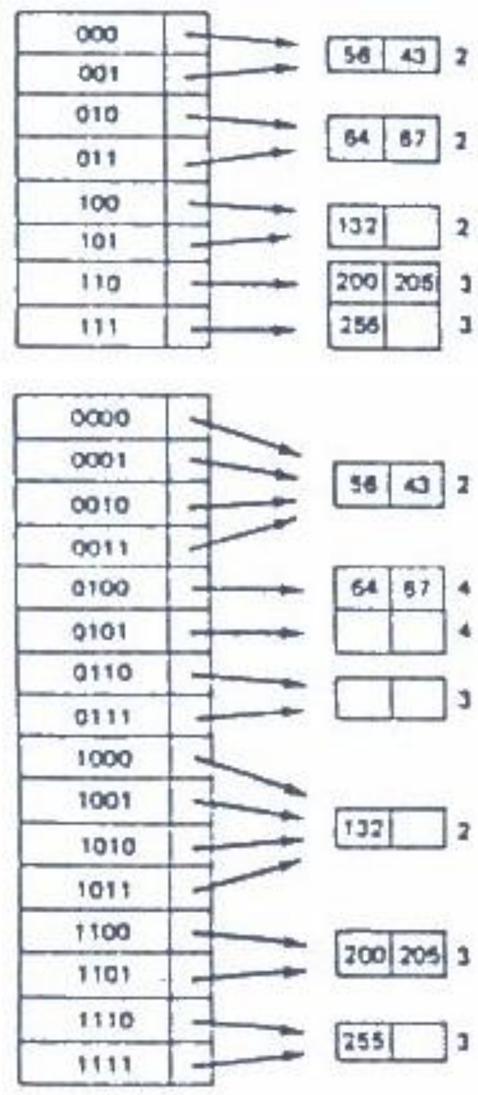
- To delete record with key **K**:
  - remove data entry from bucket
  - if bucket is empty
    - merge bucket with its split image
    - decrease local depth
    - If each directory element points to the same bucket as its split image
      - Halve directory
      - Reduce global depth

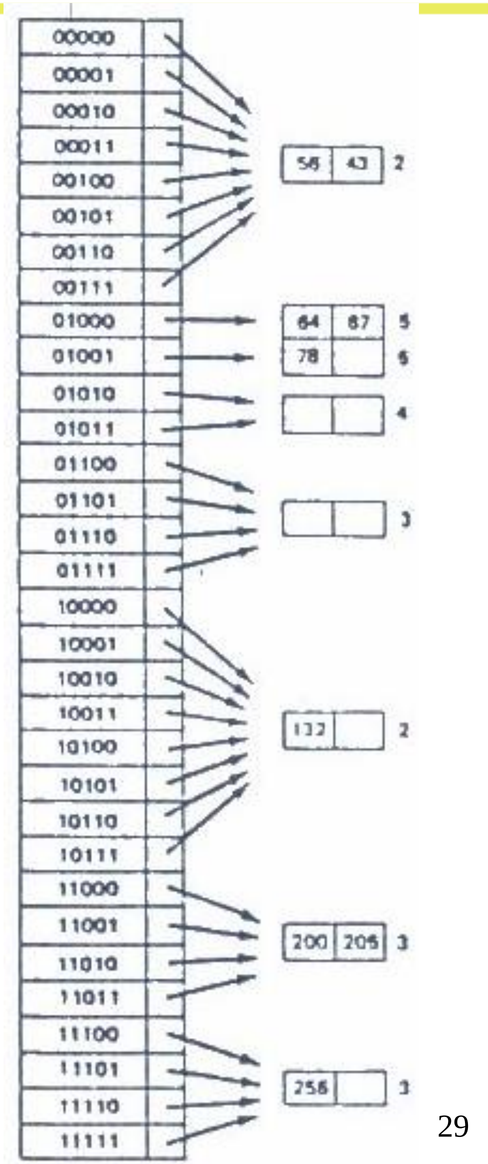
Analysis of Extensible Hash Tables

- With uniform distributed addresses, all the buckets tend to fill up at the same time => split at the same time (Periodic and fluctuating)
  - As buffer fills up to 90%
  - After a concentrated series of splits drops to under 50%
- Space utilization of the bucket
  - R (# of records), b (block size), B (# of Blocks)
  - Utilization = R / b*B
  - Average utilization ~ln 2 (natural logarithm) = 0.69 ==> 69%
    - recall normal B+-tree: 67%, B+-tree with redistribution: 85 %
- Space utilization for the directory
  - How large a directory should we expect to have, given an expected number of keys?
    - estimated directory size = $3.92/b*R^{(1+1/b)}$ Flajolet (1983)
  - Overall Load Factor (LF) of the file below 2 seeks, 75%~80% utilization Litwin (1980)
    - LF: The number of records in the file divided by the number of places for the records in the primary area

A Pathological Case







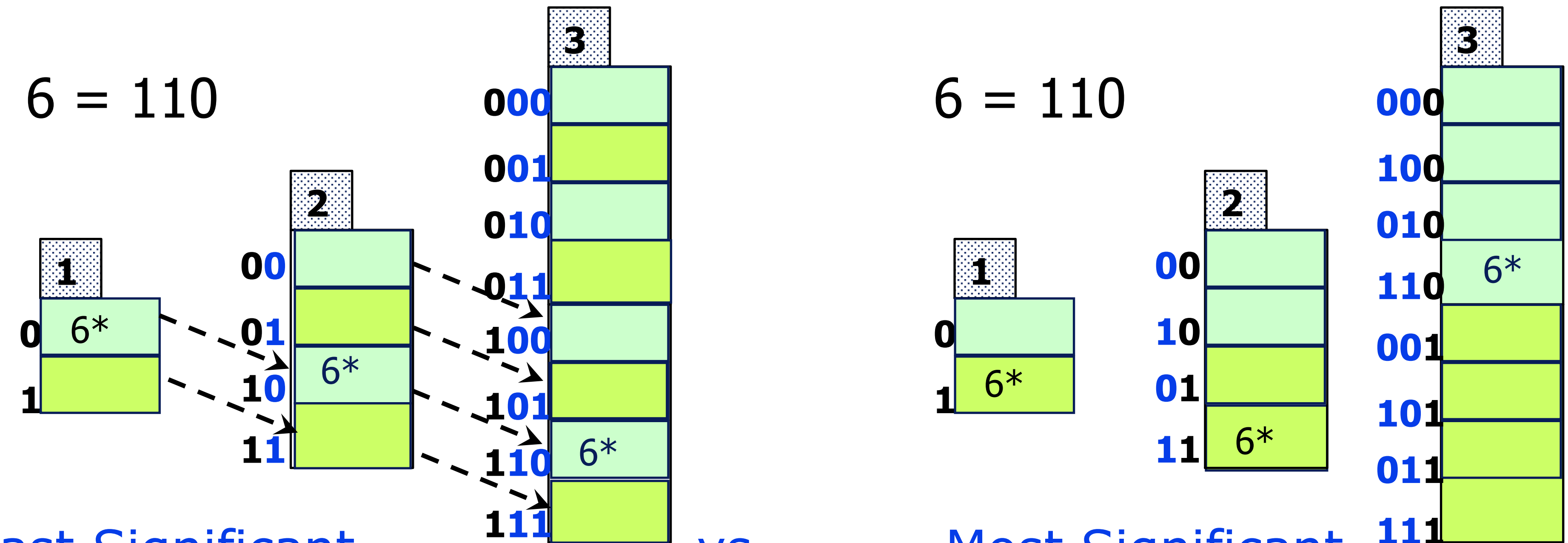
- A pathological case: We need a 32-entry table for 9 records
  - Buckets are labeled with the number of digits used. The bucket factor is 2

Efficiency of Extensible Hash Tables

- Example: 100MB file, 100 bytes/record, 4K pages contains 1.000.000 records (as data entries) and 25.000 directory entries
  - chances are high that directory will fit in memory
    - With 80% utilization the directory size will be
      - 1.2 * 25.000 * 100 = 3.000.000 = 3M
- Pros:
  - equality search can be answered with only one I/O to locate the record
  - if the size of the bucket directory is small enough to be kept in main memory no disk I/O is required to look-up the directory
- Cons:
  - when the size of the bucket directory is doubled (if `i` is already large), it may no longer fit in main memory
  - if the number of records per block is small, block splitting may occur earlier than actually required
- Can we do better? (smoother growth of bucket directories)
  - In Linear Hashing, buckets are split from left to right, regardless of which one overflowed (simple, but it works!!)

Directory Doubling: Least vs. Most Significant Bits

- 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
- Use of least (vs most) significant bits enables efficient directory doubling via copying!
  - doubled by copying it over and `fixing' pointer to split image page

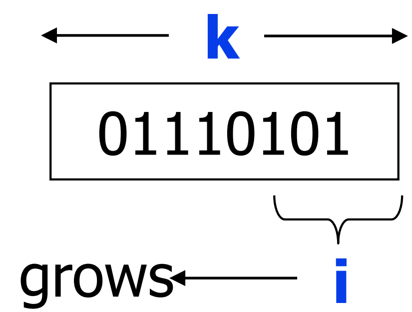


Least Significant vs. Most Significant

Linear Hashing

- Handles the problem of long overflow chains (chaining approach) without using a directory, while it also handles duplicates
- First Idea: Use a family of hash functions h_0, h_1, h_2, ...
  - Initial number of buckets **B**
  - $h_k(key) = h(key) \bmod (\mathbf{B} * 2^k)$
  - **h** is some hash function
    - range is not 0 to **B**-1
  - If $\mathbf{B} = 2^{d0}$, for some d_0, h_i consists of applying **h** and looking at the last d_i bits, where $d_i = d_0 + i$
  - h_{i+1} doubles the range of h_i
    - Eg. **B** = 32 = 2^5, $h_0 = h \bmod 32$, $d_1 = d_0 + 1 = 6$, $h_1 = h \bmod (32 * 2)$

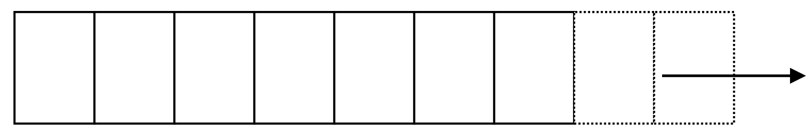
Use **i** low order bits of hash



Linear Hashing

- Second Idea: Split buckets in 'round-robin'
  - Counter `Level` indicate current round number (initially 0)
  - Bucket to split denoted by `Next` (initially bucket 0 or first bucket)
  - $B_{Level} = 2^{Level} B$ = # buckets in file at the beginning of round `Level`
    - # buckets at beginning of round 0, denoted by B_0 is B
  - Round ends when all B_{Level} buckets have been split

The size of the table will grow gradually, but not double in size



Record blocks

Linear Hashing

- During round number `Level`, only hash functions h_{Level} and $h_{Level+1}$ are used, so at any given point in a round, we have:
  - buckets that have been split (0 to `Next`-1)
  - buckets that are yet to be split (`Next` to $B_{level}-1$)
  - buckets created by splits in this round (B_{Level} to $2B_{Level}-1 < B_{level+1}$)

Linear Hashing

- In the middle of a round

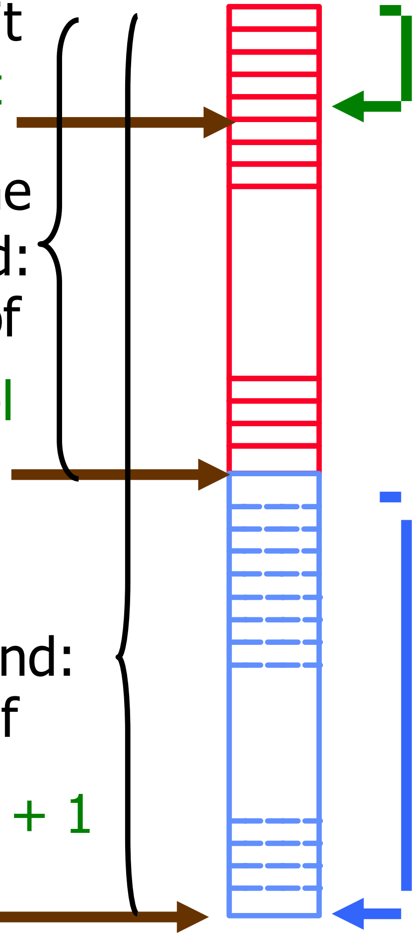
Bucket to be split
Next

Buckets that existed at the beginning of this round: this is the range of h_{Level}

B_{Level}

Buckets available at the beginning of the next round: this is the range of $h_{Level+1}$

$2B_{Level}-1$



Buckets created by split in this round:
If h_{Level} (key) is in this range, must use $h_{Level+1}$(key) to decide if entry is in `split image' bucket

`split image' buckets: created (through splitting of other buckets) in this round

Searching Linear Hash Tables

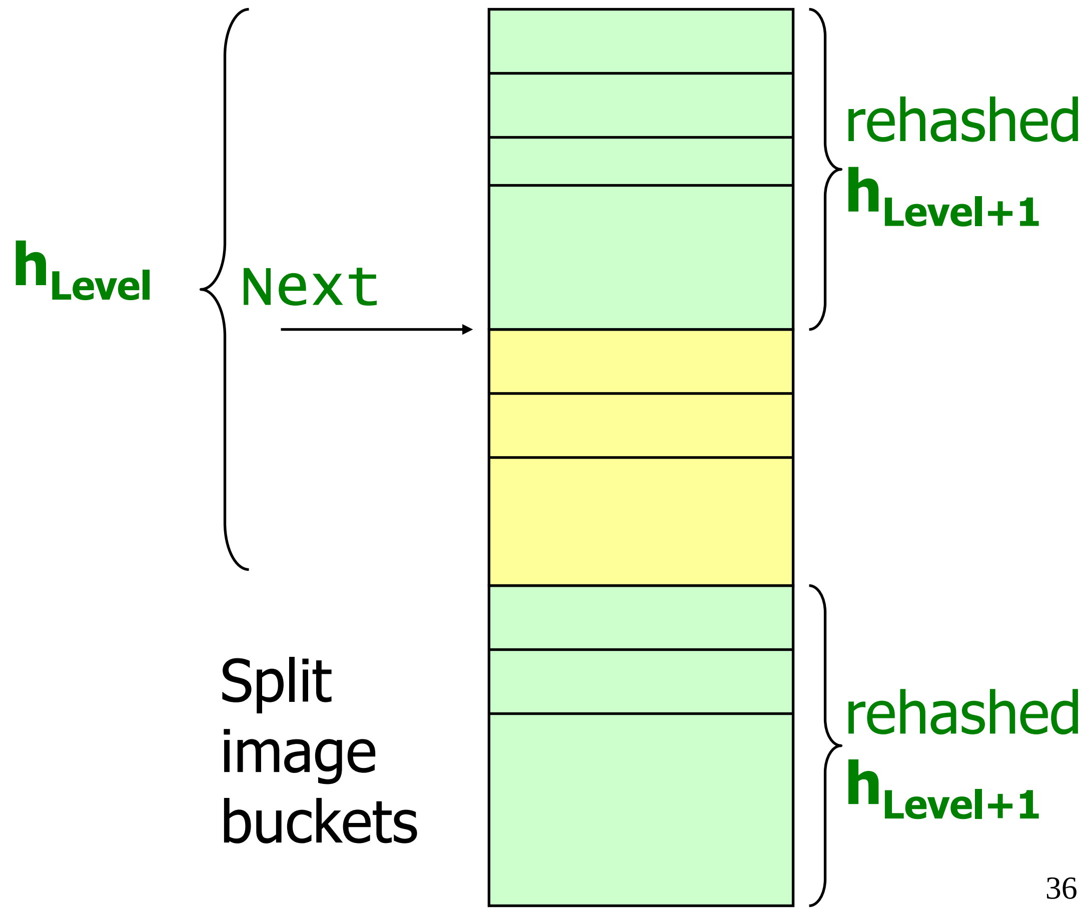
- To find bucket for data entry **K**:

active hash table buckets are those in $\mathbf{h}_{Lavel}$'s range: [0 .. 2^{Level} **B**– 1]

1. Apply hash function $\mathbf{h}_{Level}$ to search key $\mathbf{h}_{Level}(\mathbf{K})$

2. If it leads to an unsplit bucket **b** (bucket Next to $\mathbf{B}_{Level}$)
   - look for **K** there

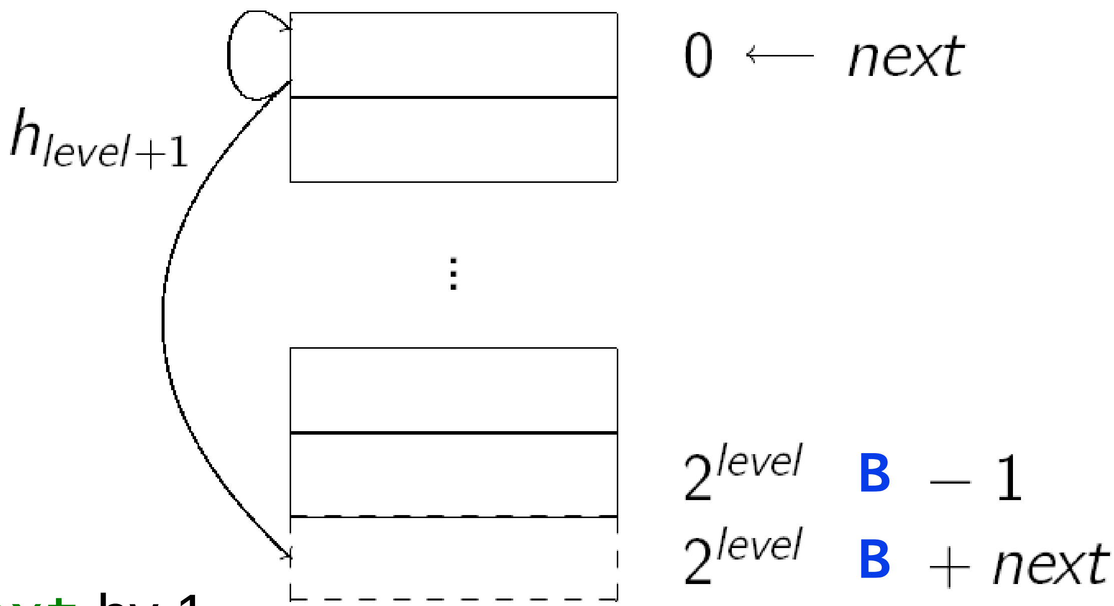
3. If it leads to a split bucket **b** (bucket 0 to Next-1), **K** can be in b or bucket b + B_{Level}
   - apply $\mathbf{h}_{Level+1}(\mathbf{K})$ to find out



Linear Hashing: Bucket Splitting

- Allocate a new bucket, append it to the hash table (its position will be 2^{Level} **B** + Next)

- Re-distribute the entries in bucket Next by rehashing them via $h_{Level+1}$ (some entries remain in bucket Next, some go to bucket **B***2^{Level}+Next)

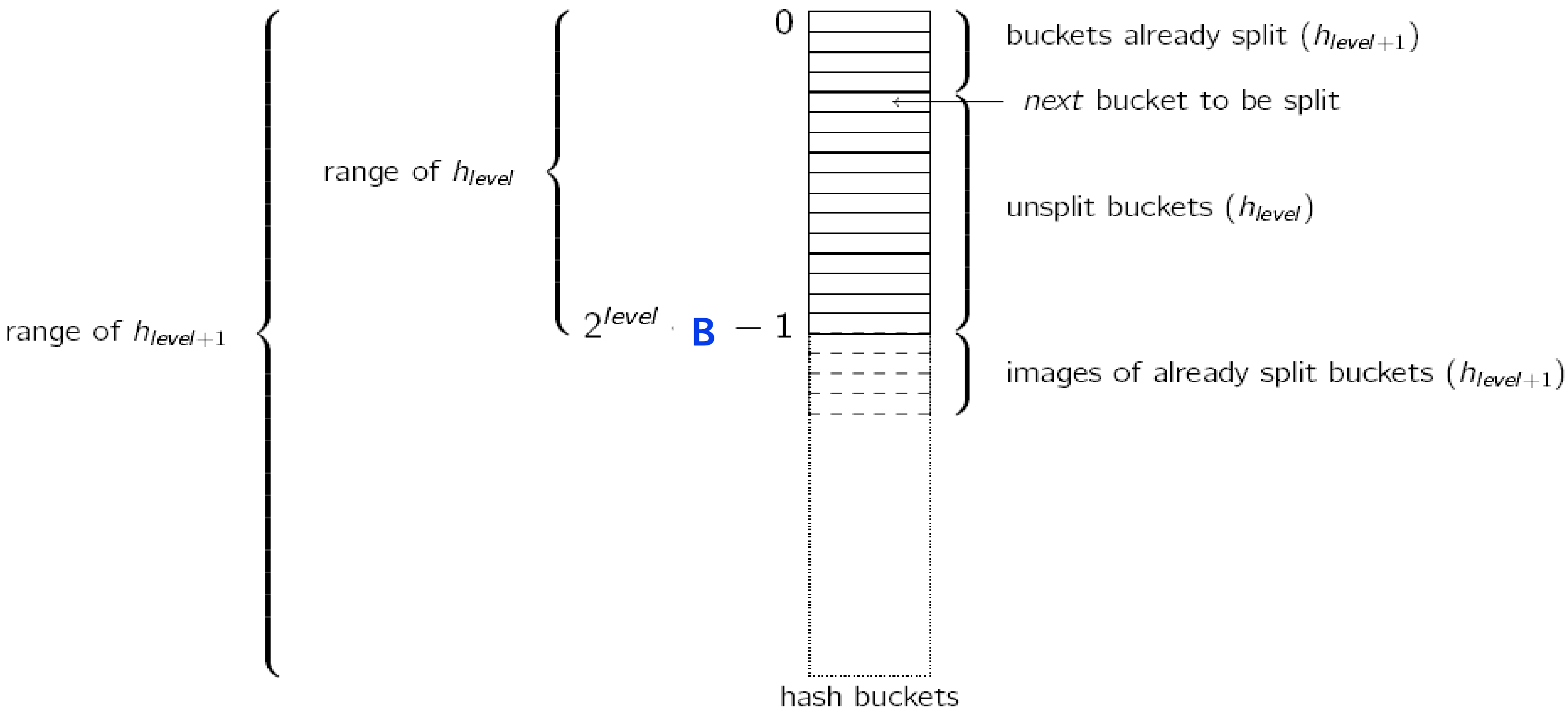


- increment Next by 1

Linear Hashing: Rehashing

- Hashing via h_{Level} has to take care of Next's position:

h_{Level}(key) = { < Next: we hit an already split bucket, rehash
>= Next: we hit a yet unsplit bucket, bucket found



Linear Hashing: Split Rounds

- A bucket split increments `Next` by 1 to mark the next bucket to be split

- How would you propose to handle the situation if `Next` is incremented beyond the last current hash table position, i.e., `Next` $> 2^{Level}B - 1$?
  - If `Next` $> 2^{Level}B - 1$, all buckets in the current hash table are hashed via $h_{Level+1}$ (see previous slides)
  - Linear hashing thus proceeds in a round-robin fashion:
    - If `Next` $> 2^{Level}B - 1$, then
      - increment level by 1,
      - reset next to 0 (start splitting from the hash table top again)

- In general, an overflowing bucket is not split immediately, but—due to round-robin splitting—no later than in the following round

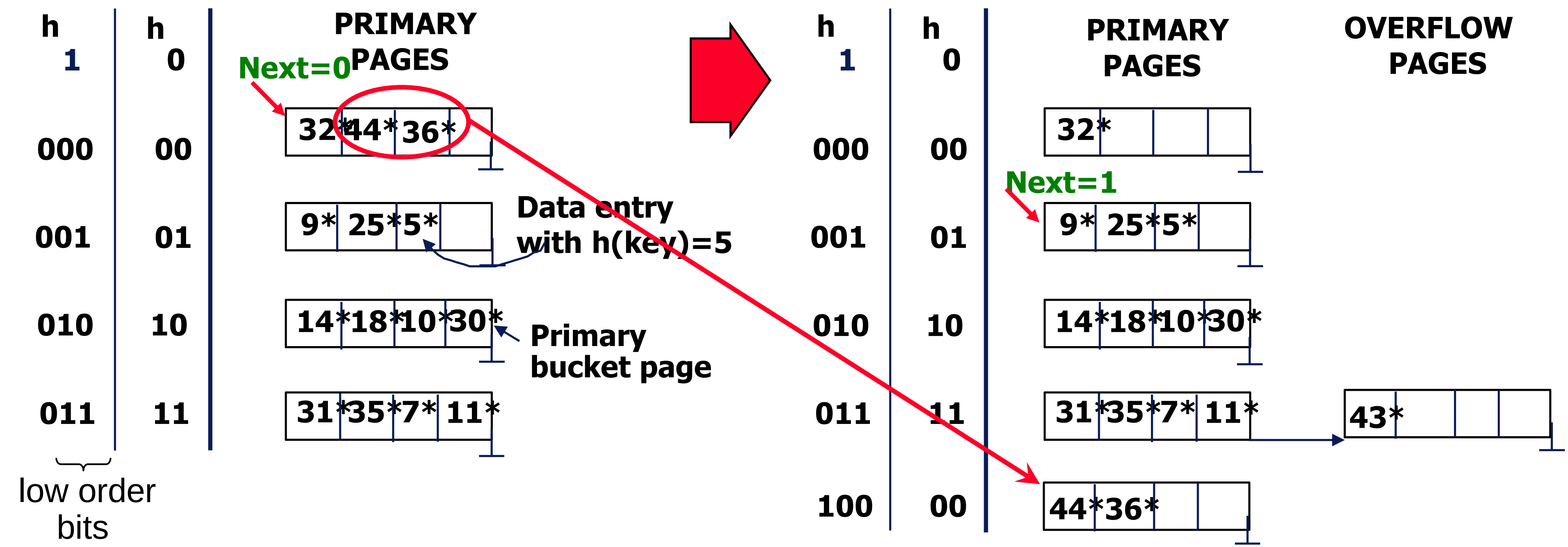
Insertion into Linear Hash Tables

- To insert a data entry **K**:
  - Locate bucket by applying $h_{Level}(K)$ or $h_{Level+1}(K)$
    - If bucket is full
    - Add overflow page and insert data entry
    - (Maybe) Split bucket `Next`
      - Hash function $h_{Level+1}$ redistributes entries between bucket `Next` and bucket `Next` + B_{Level}
      - Increment pointer `Next`
- Buckets are split round-robin, long overflow chains don't occur
  - in general, the spitted bucked is not the bucket which triggered the split!
  - each bucket may use overflow pages
- Choose any criterion to `trigger' split
  - insertions filled a primary bucket page beyond c % capacity,
  - or the overflow chain of a bucket grew longer than p pages, or . . .
- We will examine, in the sequel, the first growth method

Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries

Level=0, B=4, primary bucket capacity = 4 **Level=0, Insert record such that $h0(key)=43=101011_2$**



(This info is for Illustration only!)

(The actual contents of the linear hashed file)

Example: End of a Round

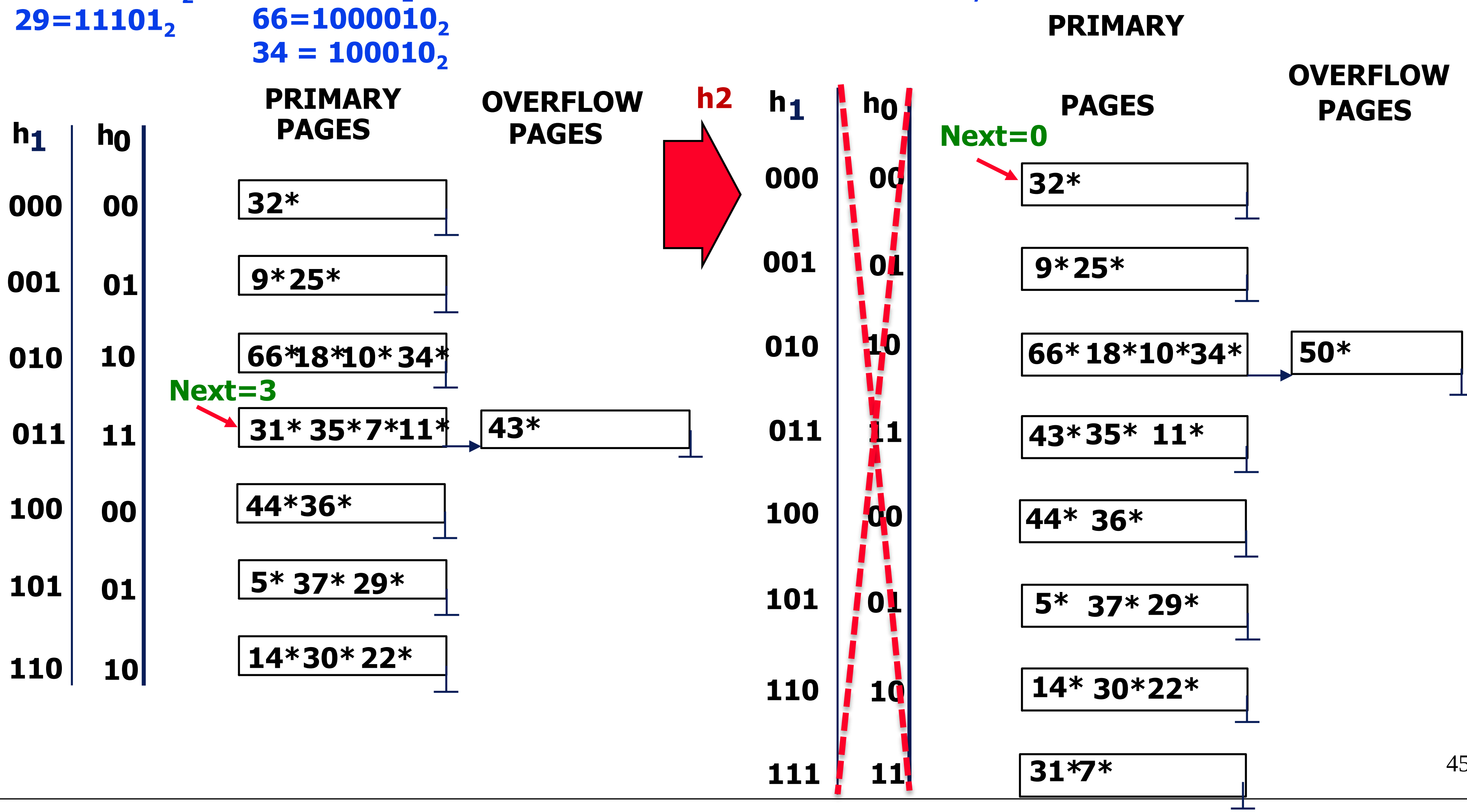
Level=0 , Insert records such that $h_0(k)$

$37=100101_2$ $22=10110_2$

$29=11101_2$ $66=1000010_2$

$34 = 100010_2$

Level=1 , B=8



Analysis of Linear Hashing

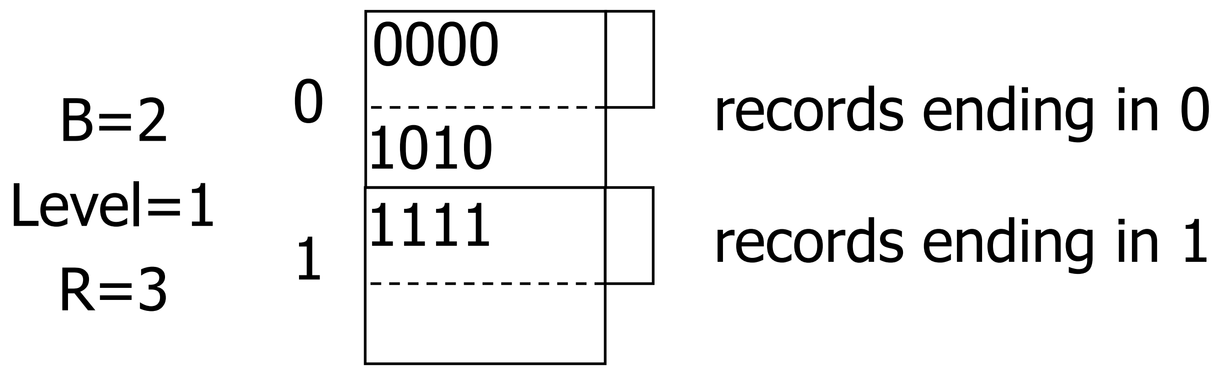
- **B**: the current number of bucket blocks in use (at round number `Level`)
  - As the table grows, the initial # of bucket blocks is increased but it may not always be a power of 2
- **`Level`**: the # bits of **`h(k)`** used to address buckets is $\lfloor \log_2 B \rfloor$ (low-order bits)
- **R**: number of records stored in hash table
  - initially 0 and increased as records are added
- Utilization of the hash table is the number of stored records divided by the number of possible storage places = **R** / **B** * c (#entries/block)
  - ratio **R**/**B** limited so that the average number of records per bucket is a fixed fraction of the number of records that fit in one block
  - overflow blocks are permitted, but the average number of overflow per bucket will be <= 1
- We will use this ratio to determine when to grow or shrink the hash table
  - grow when **R**/**B** >= 90% and shrink when **R**/**B** <= 75%
  - we will use the average value of 85% as the threshold for growth

Insertion into Linear Hash Tables: Alternative Algo

- To insert record with key K:
  - Compute $h(K)=b$ and determine the i-bit rightmost ($b=a_1a_2...a_i$) sequence of $h(K)$ to use as the bucket number
    - If $b<B$ the bucket with number b exists and the record is placed there
    - If $B \leq b < 2^i$, then the bucket does not exist and we have to place the record in bucket $b-2^{i-1}$ (this is the bucket we would obtain if a_1 changed to 0)
      - If the bucket we add has number $1a_2...a_i$, then we split the bucket numbered $0a_2...a_i$ and divide the records depending on their last i bits
    - If $B>2^i$ then i is incremented by 1 (leading 0 is added)
  - Each time an insertion need to be performed, we examine the ratio R/B and if it is too high, we increase B by 1 (R the # of records in hash table)

Linear Hashing Occupancy Example

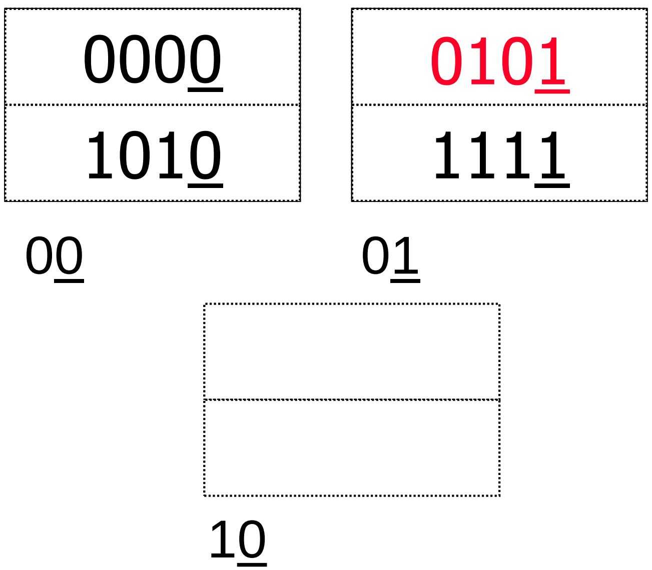
- Example: B=2, Level=1, assume **h** produces 4-bit sequences (k=4)



- If **R** <=1.7**B**, and block holds 2 records then on the average, occupancy c of a bucket does not exceed 85% (i.e., ≤ 1.7/2) of the block capacity

Linear Hashing: Second Example

- Example: 2 keys/block, R=3, B=2, Level =1 assume **h** produces 4-bit sequences (k=4)



- insert 0101
- now R=4 >1.7B (**R**/**B** = 2)
→ B++ Level++
→ get new bucket 10 and distribute keys between buckets 00 and 10

Rule

Rule

If h(k)[i] $= (a_1 \ldots a_i)_2 <$ B, then
look at bucket h(k)[i];
else
look at bucket h(k)[i] $- 2^{i-1} = (0a_2 \ldots a_i)_2$

Linear Hashing: Second Example

-> R=4, B=3, Level =2;

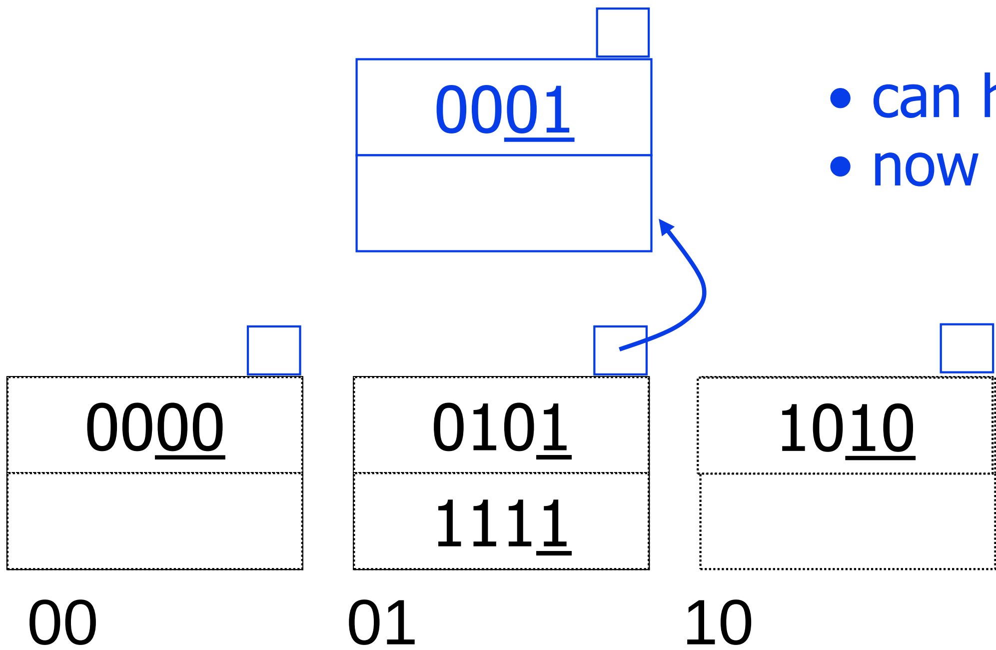
distribute keys between buckets 00 and 10:

00	01	10
0000	0101	1010
~~1010~~	1111	

- now R=4 <1.7B (**R**/**B** = 1.33) !

Linear Hashing: Second Example

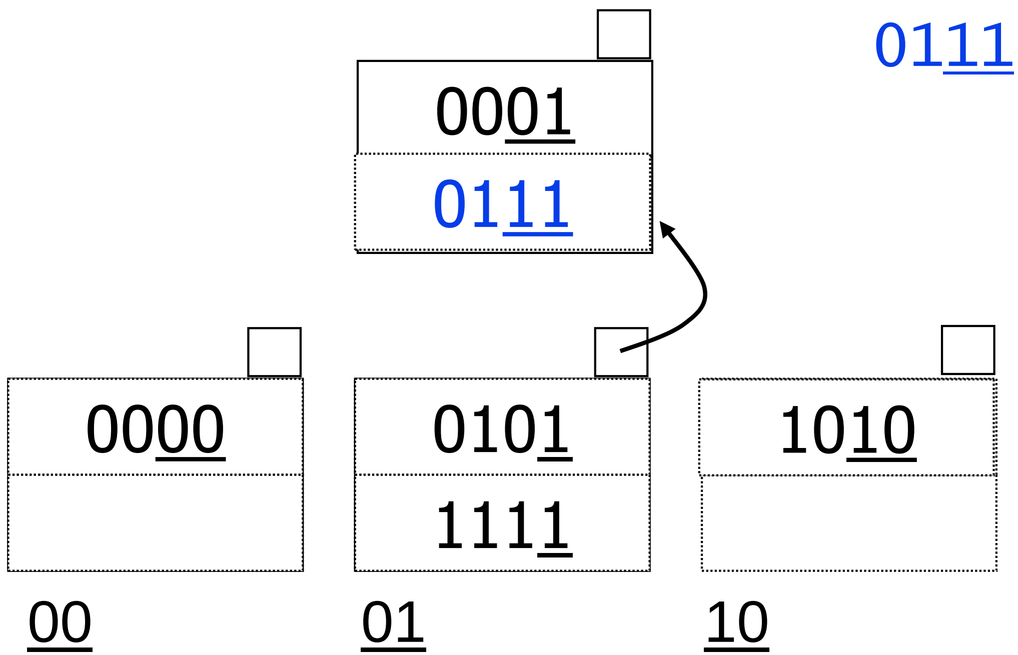
R=4, B=3, Level =2; insert 0001:



- can have overflow chains!
- now **R**/**B** becomes 5/3 < 1.7

Linear Hashing: Second Example

R=5, B=3, Level =2



- insert 0111
- bucket 11 not in use

→ redirect to 01

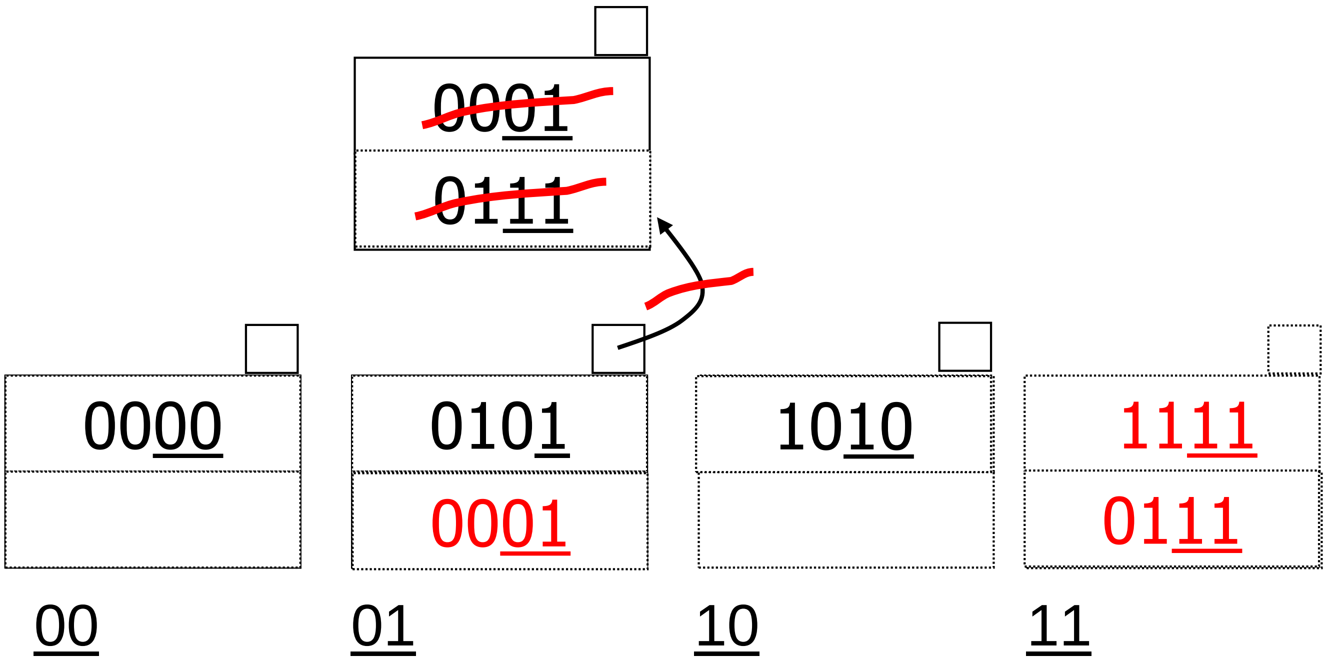
- now R=6 > 1.7B
  → get new bucket 11

Rule

If h(k)[i] = $(a_1 \ldots a_i)_2$ < B, then
look at bucket h(k)[i];
else
look at bucket h(k)[i] – $2^{i-1} = (0a_2 \ldots a_i)_2$

Linear Hashing: Second Example

R=6, B=4, Level =2; distribute keys between 01 and 11

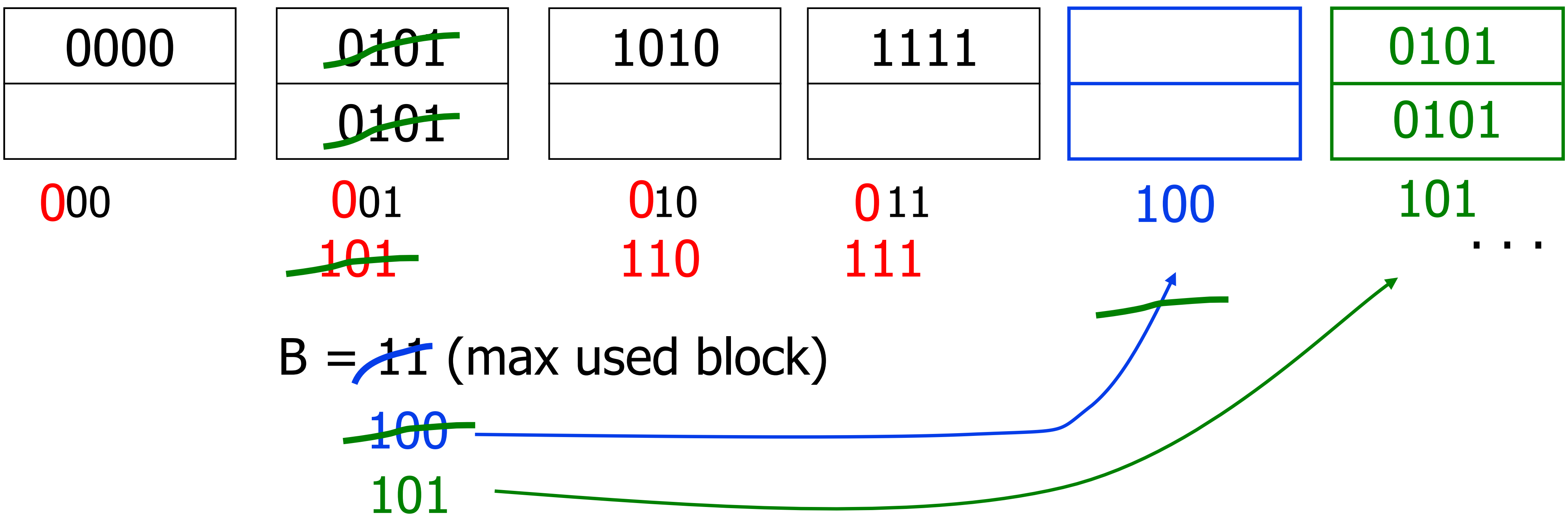


Overflow block no longer needed !

Linear Hashing: Second Example

Example Continued: How to grow beyond this?

```
B++
Level++ (2) 3
```



Linear vs. Extensible Hashing

- Two schemes are quite similar
  - Imagine Linear Hashing (LH) has a directory with elements 0 to **B**-1
  - First split at bucket 0, add directory element **B**
    - Imagine entire directory doubled at this point
    - Elements <1,**B**+1>, <2,**B**+2>, ... are the same
    - Only create directory element **B**, which differs from 0, now
  - Second split at bucket 1, create directory element **B**+1
  - So, directory can double gradually
    - Also, primary bucket pages are created in order
    - If they are allocated in sequence too, then finding bucket **i** is easy
    - Thus, we actually don't need a directory!
- LH is recommended when main storage is at a premium since it requires no directory
  - Particularly useful in a small computer environment
- EH could be useful if sufficient main memory is available to hold the directory
  - Doubling and halving the directory size is expensive

Hashing vs. B+ trees

- In a typical DBMS, you will find support for B+ trees as well as hash-based indexing structures
  - Cost of periodic re-organization
  - Relative frequency of insertions and deletions
  - Is it desirable to optimize average access time at the expense of worst-case access time?
- In a B+ tree, to locate a record with key k means to compare k with other keys k' organized in a (tree-shaped) search data structure
- Hash indexes use the bits of k itself (independent of all other stored records and their keys) to find the location of the associated record
- Hash indexes: best for equality searches, cannot support range searches (also known as scatter storage)
  - True, but they can be used to answer range queries in O(1+Z/B) I/Os, where Z is the number of results (Alstrup et all 2001; Brodal et all 2004)
- Although hash indexes supports multiple attribute keys, cannot support partial key search

References

- Based on slides from:
  - R. Ramakrishnan and J. Gehrke
  - H. Garcia Molina
  - J. Hellerstein
  - L. Mong Li
  - P. Kilpeläinen
  - R. Lawrence
  - M. H. Scholl

Τέλος Ενότητας

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
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

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

Σημείωμα αδειοδότησης

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Διάλεξη 3η: Access methods: Hash IndexesΔιάλεξη 2η: Access methods, File organization, B+Tree**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/