Chapter 7: Storage Management

Part 2: Index Structures

Zhaonian Zou

Massive Data Computing Research Center
School of Computer Science and Technology
Harbin Institute of Technology, China
Email: znzou@hit.edu.cn

Spring 2019

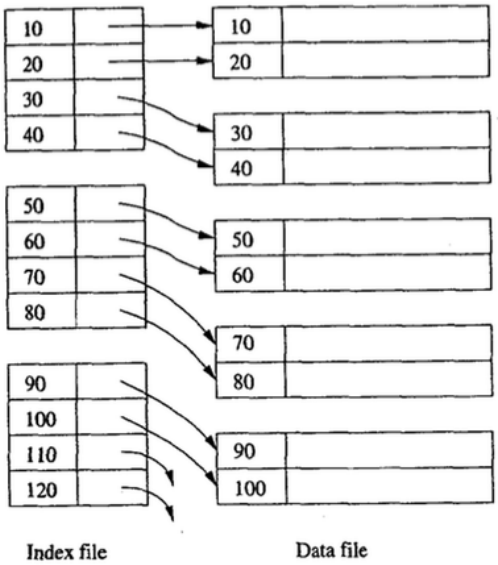
Outline[1]

[1] Updated on April 2, 2019

7.5 Indexes on Sequential Files

Index Structures (索引结构)[2]

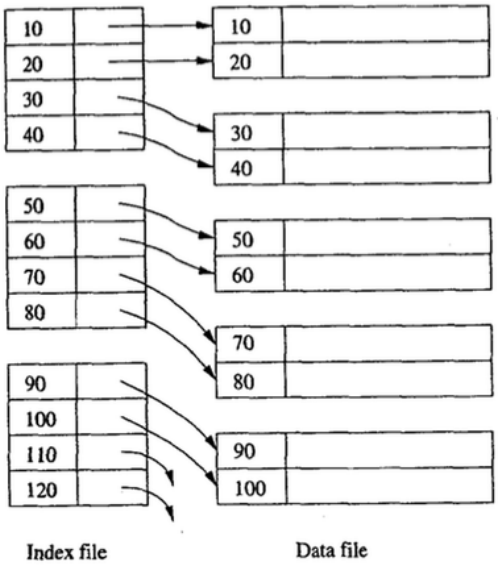
- The data file (数据文件) stores sorted data records
- The index file (索引文件) stores key-pointer pairs corresponding to the data records or the data blocks



[2]Please refer to Chapter 6 for more details on the concepts and the design of indexes

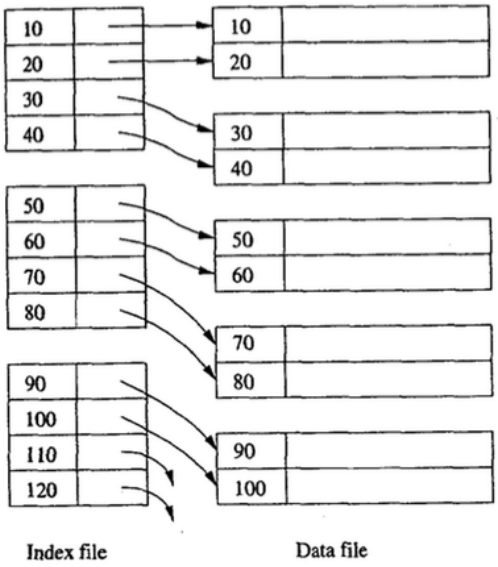
Index Structures (Cont'd)

- # of index blocks < # of data blocks (Why?)
- We can use binary search to find a key K in the index file



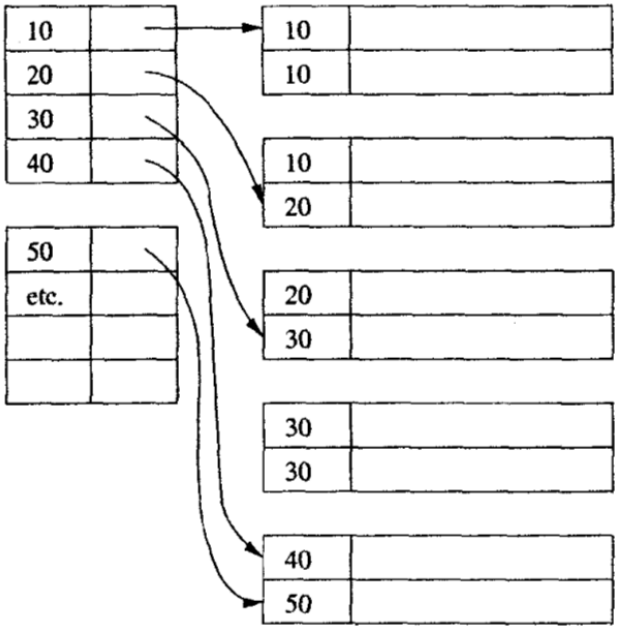
Dense Indexes (稠密索引) with Unique Search Keys

- The data records are sorted by the search key
- There is a key-pointer entry in the index file for every record of the data file (1条记录⟷1个索引项)
- The index entries are also sorted by the search key



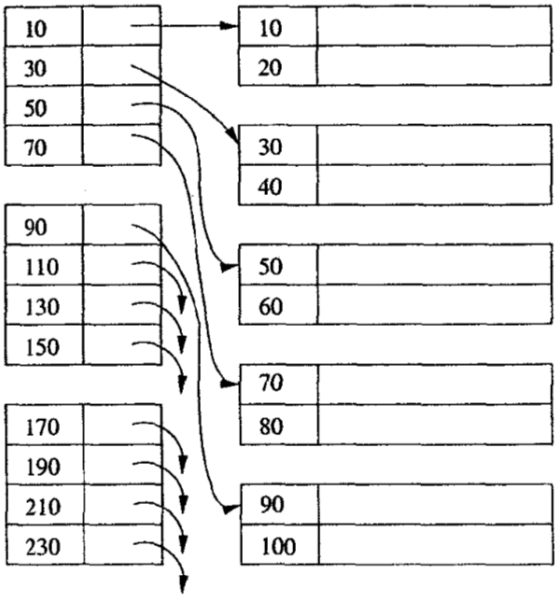
Dense Indexes with Duplicate Search Keys

- There is a key-pointer entry in the index file for each search key K (1个键值⟷1个索引项)
- This pointer points to the first of the records with K



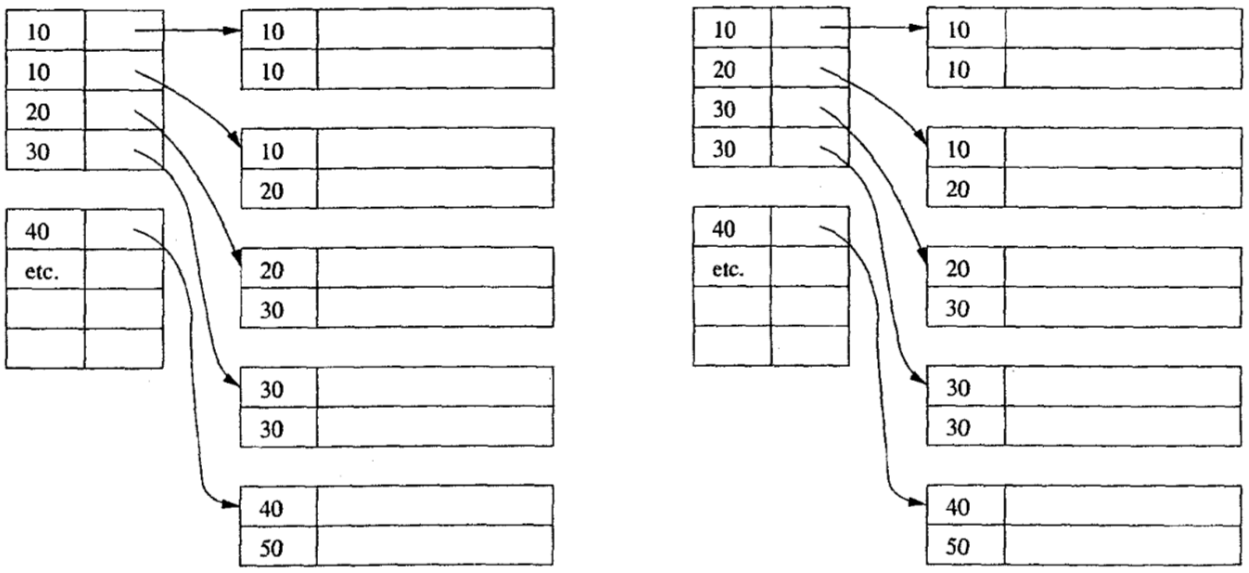
Sparse Indexes (稀疏索引) with Unique Search Keys

- The data records are sorted by the search key
- There is a key-pointer entry in the index file for every data block. The key is for the first record on the block. (1个块⟷1个索引项)
- The index entries are also sorted by the search key



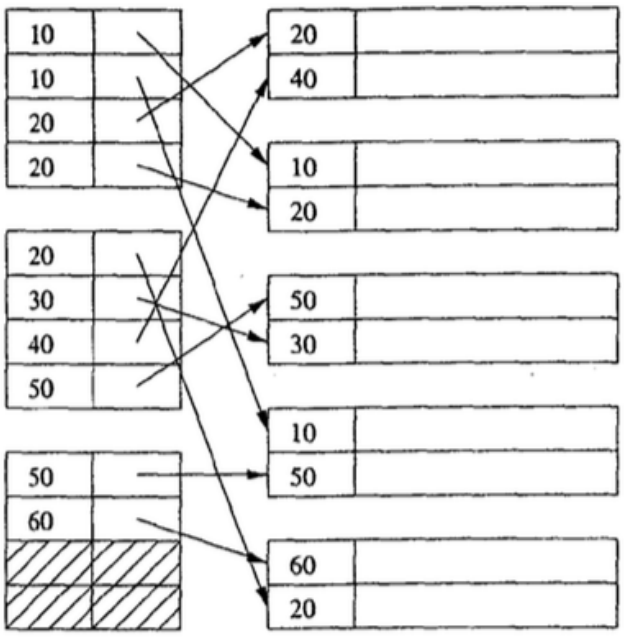
Sparse Indexes with Duplicate Search Keys

- There is a key-pointer pair in the index file for each data block (1个块⟷1个索引项)
- The key is the lowest search key or the lowest new search key on the block



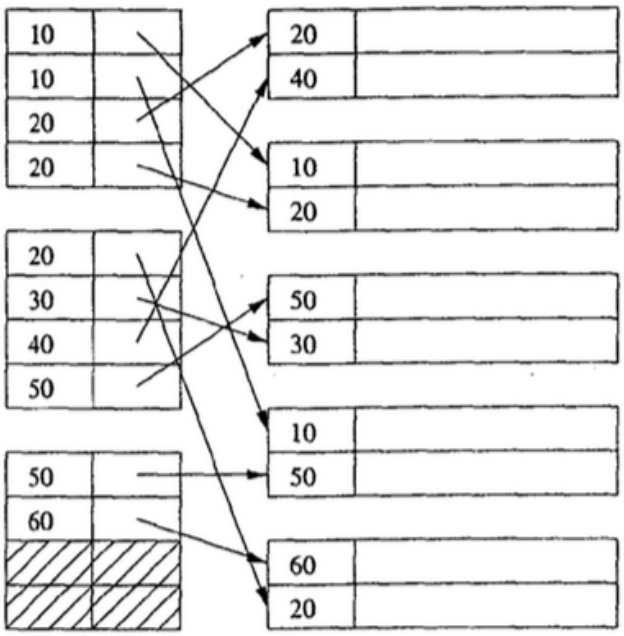
Secondary Indexes (二级索引)

- The data file is sorted by the attributes other than the search key
- The key-pointer entries in the index file are sorted by the search key
- A secondary index must be dense (Why?)



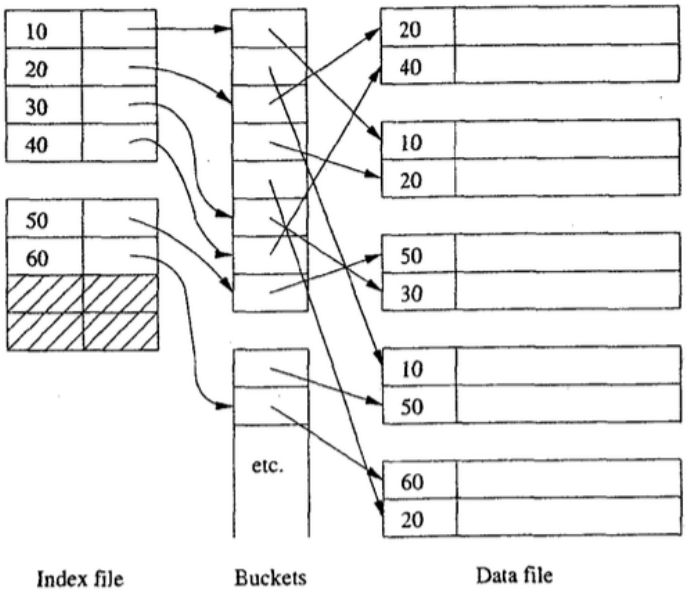
Secondary Indexes (Cont'd)

- If a search-key value appears n times in the data file, then the value is written n times in the index file



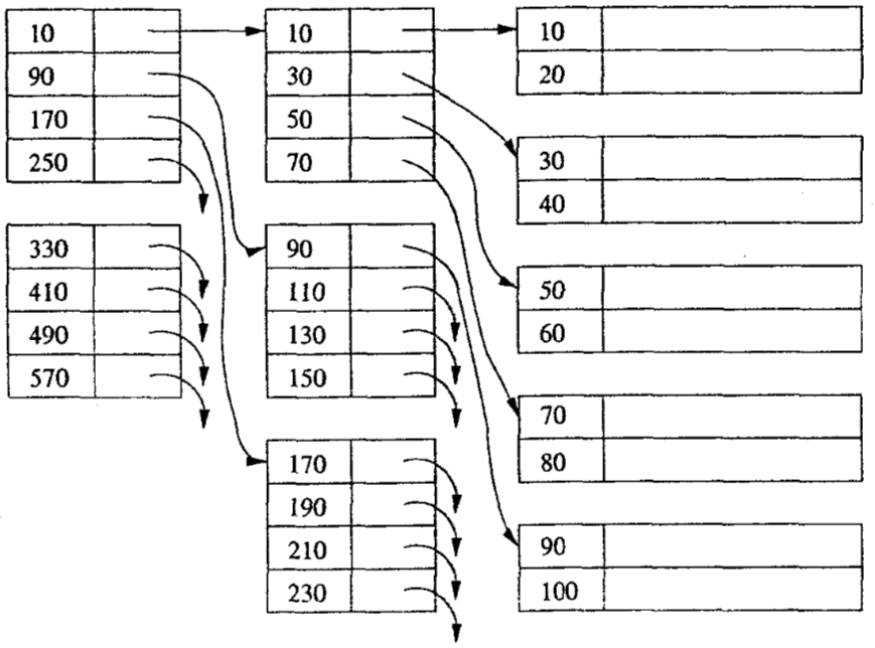
Secondary Indexes (Cont'd)

- If a search-key value appears n times in the data file, then the value is written n times in the index file
- A convenient way to avoid repeating values is to use a level of indirection called buckets



Multilevel Indexes (多层索引)

- Put an index on an index
- The first-level index can be either sparse or dense
- The second and higher levels must be sparse (Why?)

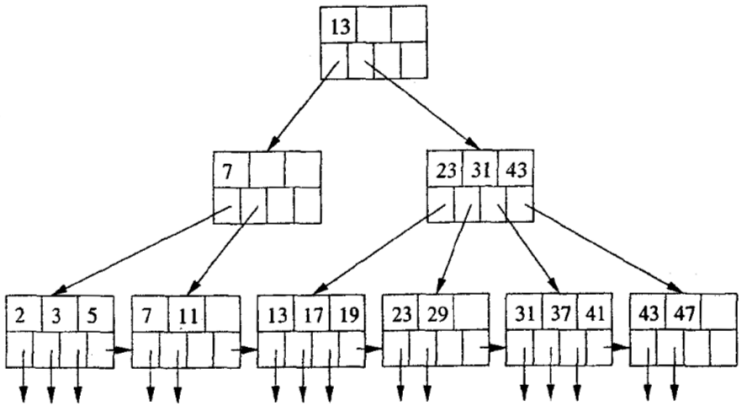


7.6 Tree-based Index Structures

7.6 Tree-based Index Structures
B+ Trees

B+ Trees

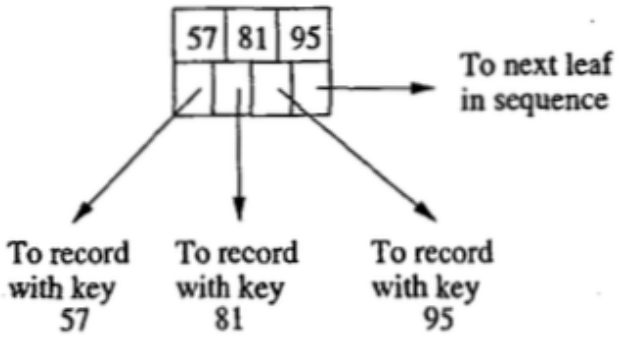
- A B+ tree is balanced (平衡树)
- 1 node = 1 disk block
- Each node contains n search-key values and $n+1$ pointers, where n is as large as possible (多叉树)
- A B+ tree generally consists of three types of nodes
  - The root (根节点)
  - Interior nodes (内节点)
  - Leaf nodes (叶节点)



B+ Tree Nodes

Leaf Nodes (叶节点)

- (串接) The last pointer points to the next leaf to the right
- (半满至全满) At least $\lfloor \frac{n+1}{2} \rfloor$ pointers except the last one are used and point to data records
- The ith pointer, if used, points to a record with the ith key
- (有序) All keys in the leaf nodes are ordered in increasing order of the search key



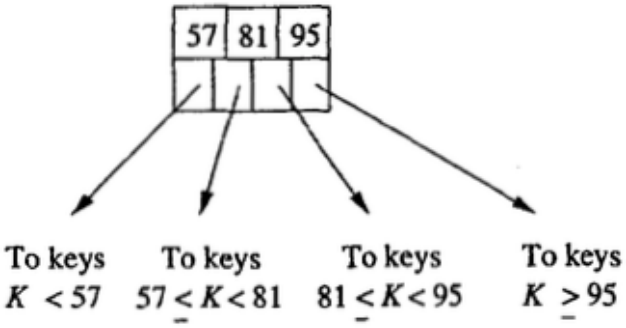
B+ Tree Nodes (Cont'd)

The Root (根节点)

- (至少有2个儿子) There are at least two used pointers

Interior Nodes (内节点)

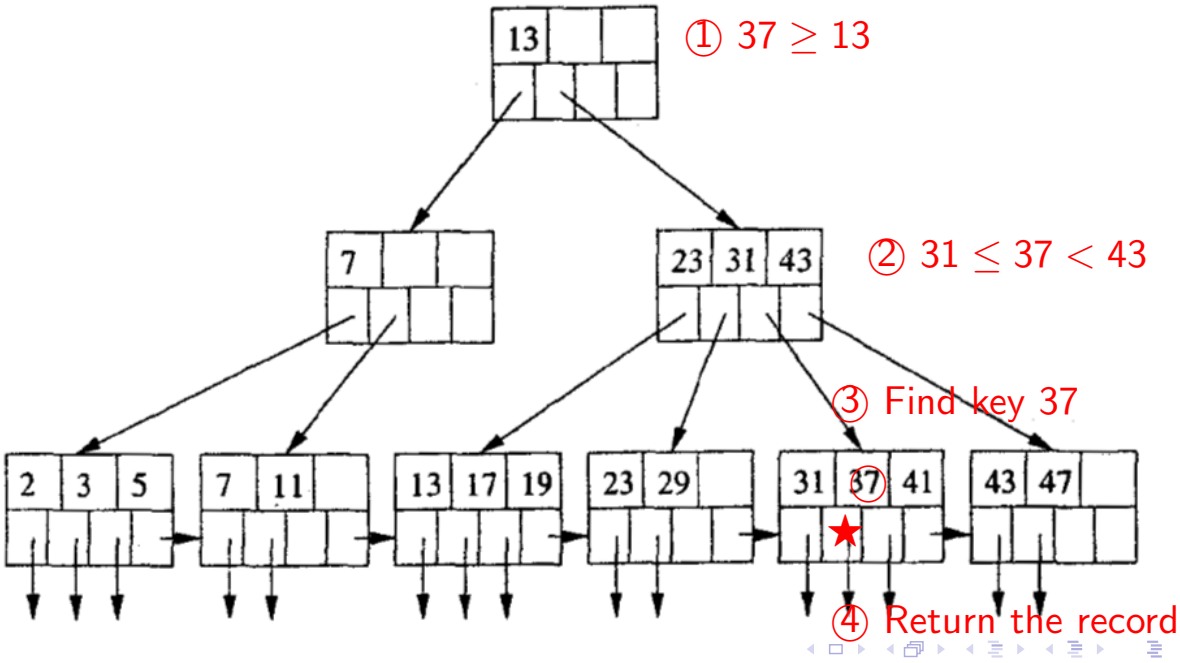
- (半满至全满) At least $\lceil \frac{n+1}{2} \rceil$ pointers are actually used



Lookup in B+ Trees

Example (Lookup)

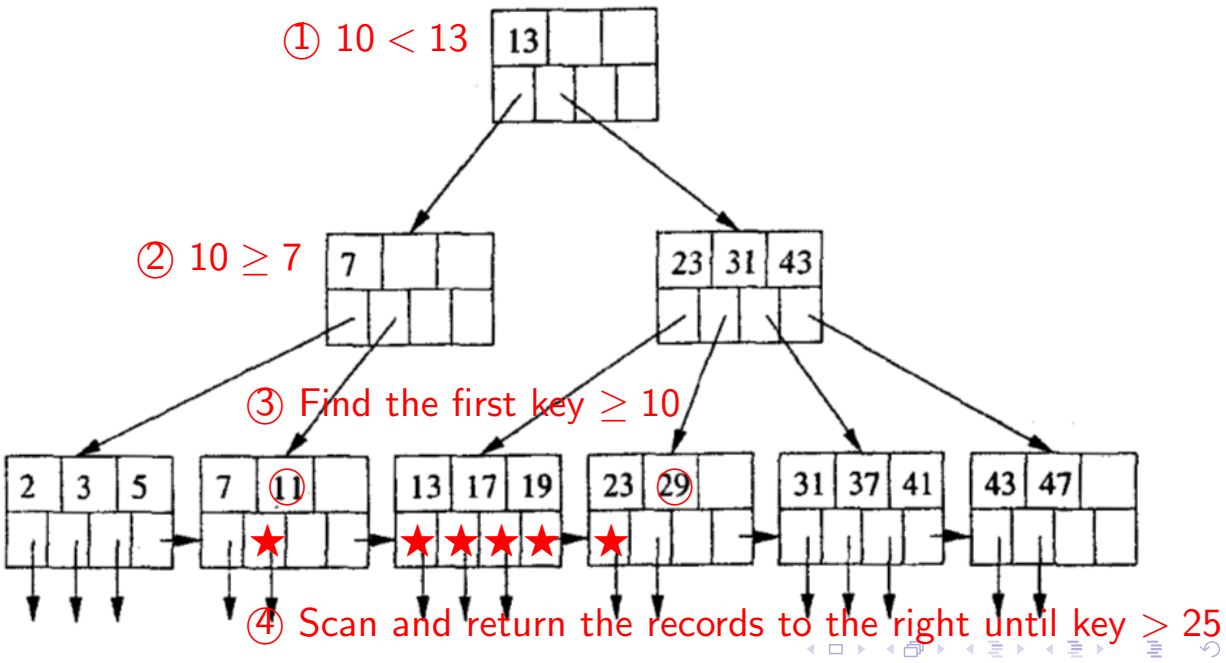
Find a record with search key 37



Range Queries in B+ Trees

Example (Range Queries)

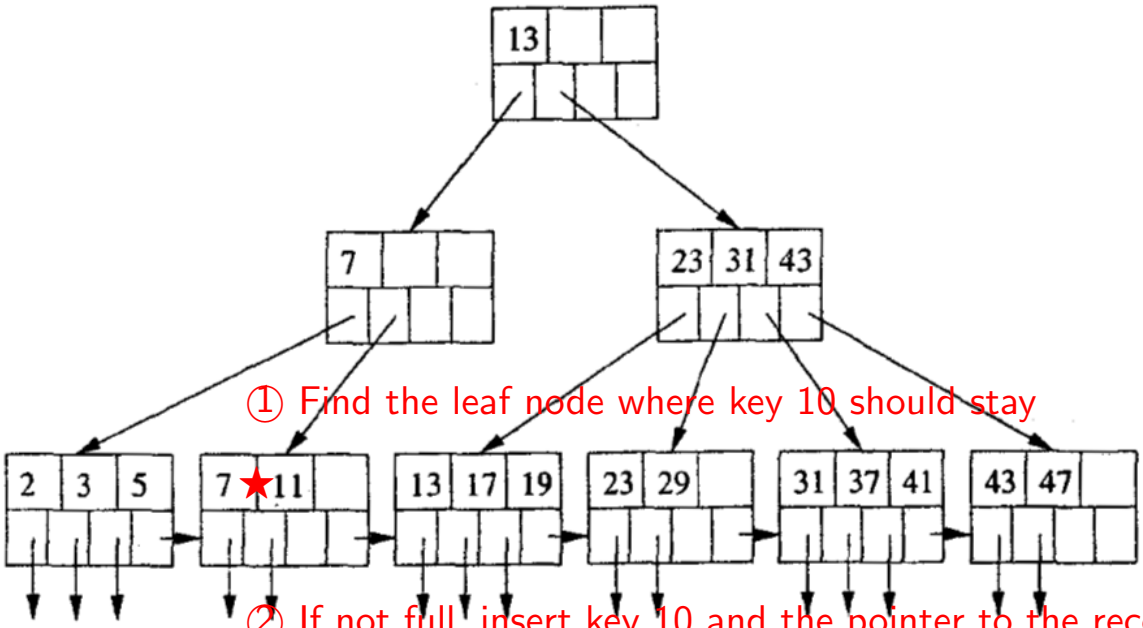
Find records with search keys in $[10, 25]$



Insertion into B+ Trees without Splitting

Example (Insertion without Splitting)

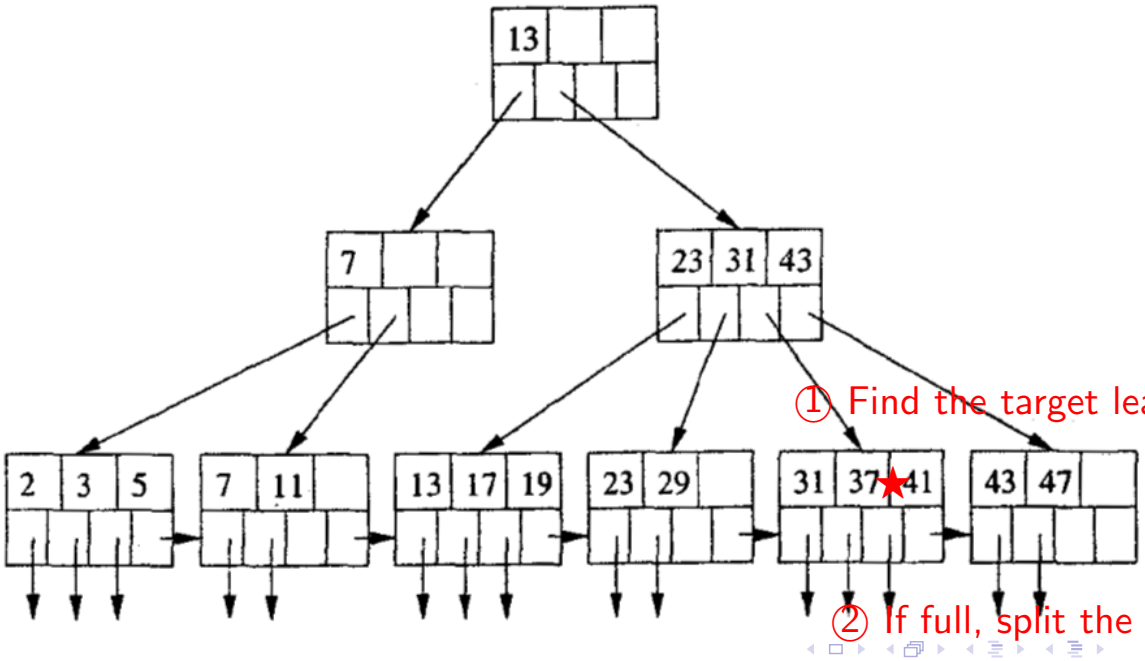
Insert a record with search key 10



Insertion into B+ Trees with Splitting

Example (Insertion with Splitting, Step 1)

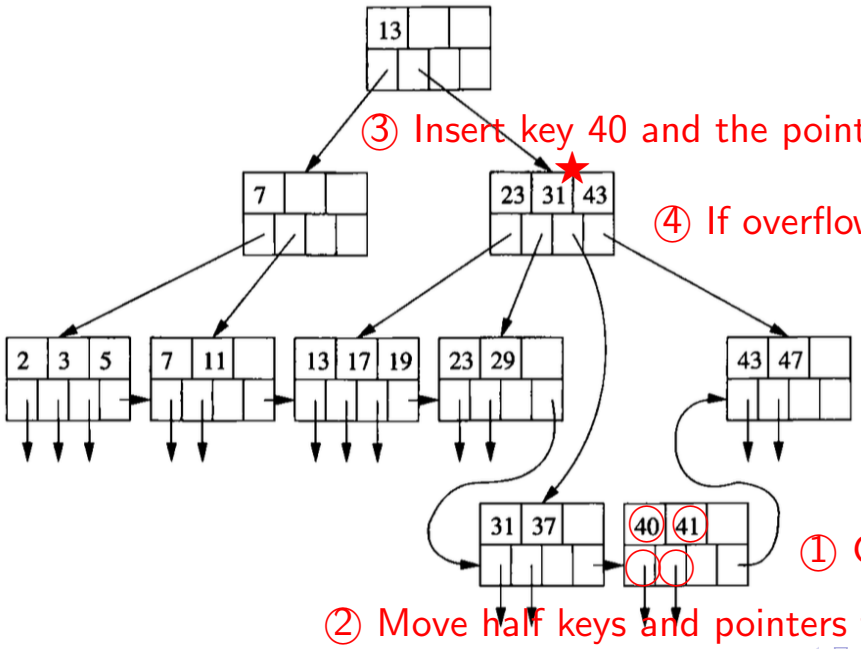
Insert a record with search key 40



Insertion into B+ Trees with Splitting (Cont'd)

Example (Insertion with Splitting, Step 2)

Insert a record with search key 40



③ Insert key 40 and the pointer to the new leaf node

④ If overflow, split this node

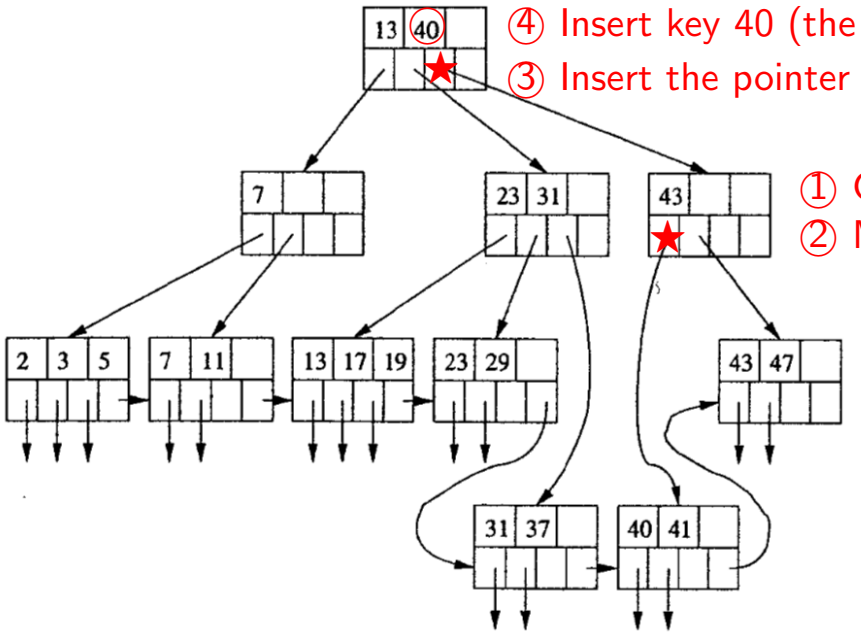
① Create a new leaf node

② Move half keys and pointers to the new leaf node

Insertion into B+ Trees with Splitting (Cont'd)

Example (Insertion with Splitting, Step 3)

Insert a record with search key 40



④ Insert key 40 (the min key in the subtree)

③ Insert the pointer to the new node

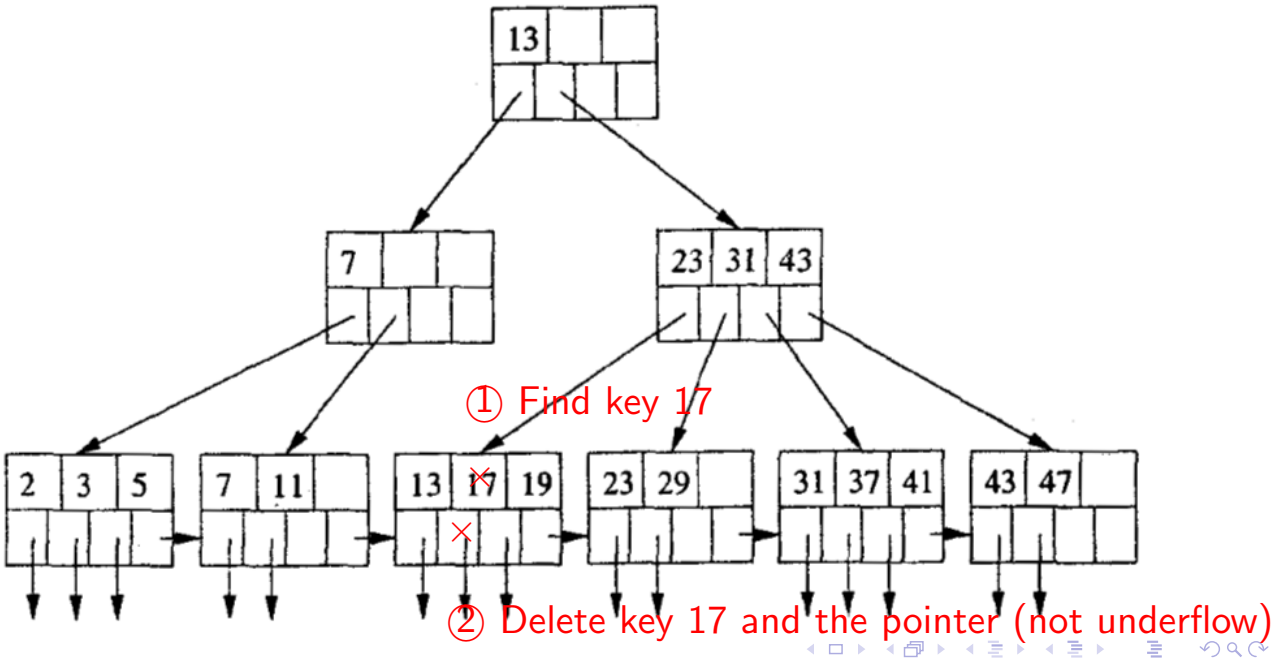
① Create a new node

② Move half pointers here

Deleting Non-minimum Key in Leaf Node

Example (Deleting Non-minimum Key in Leaf Node)

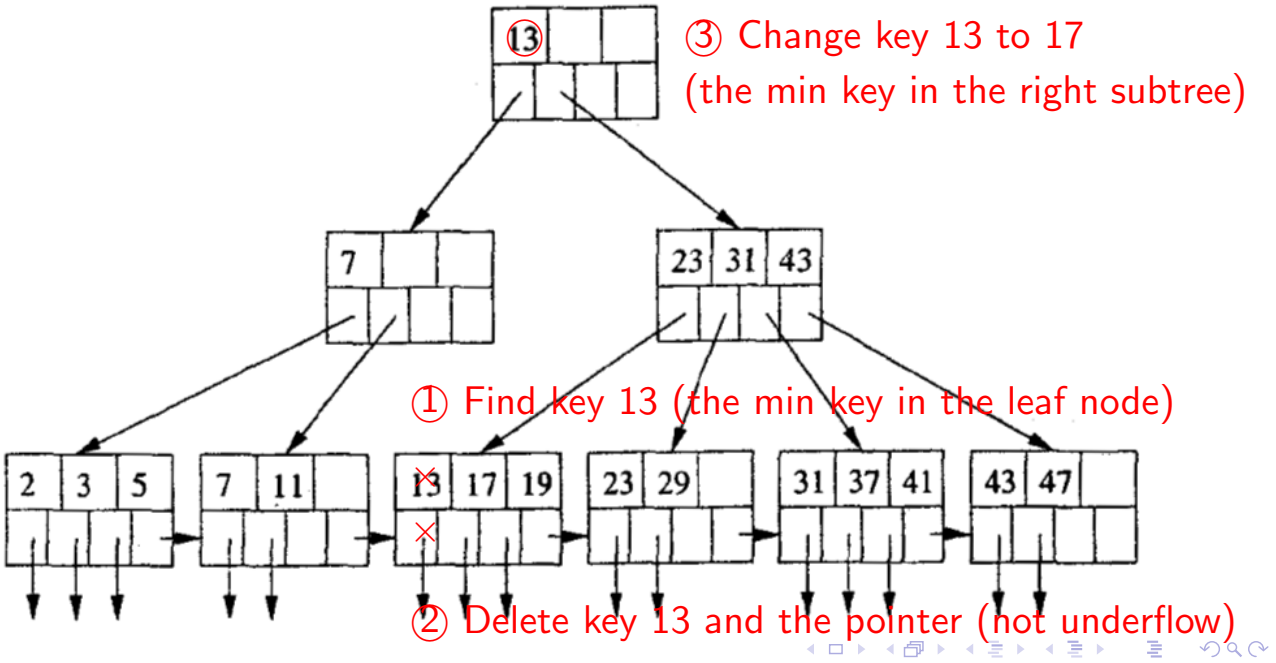
Delete a record with search key 17



Deleting the Minimum Key in Leaf Node

Example (Deleting the Minimum Key in Leaf Node)

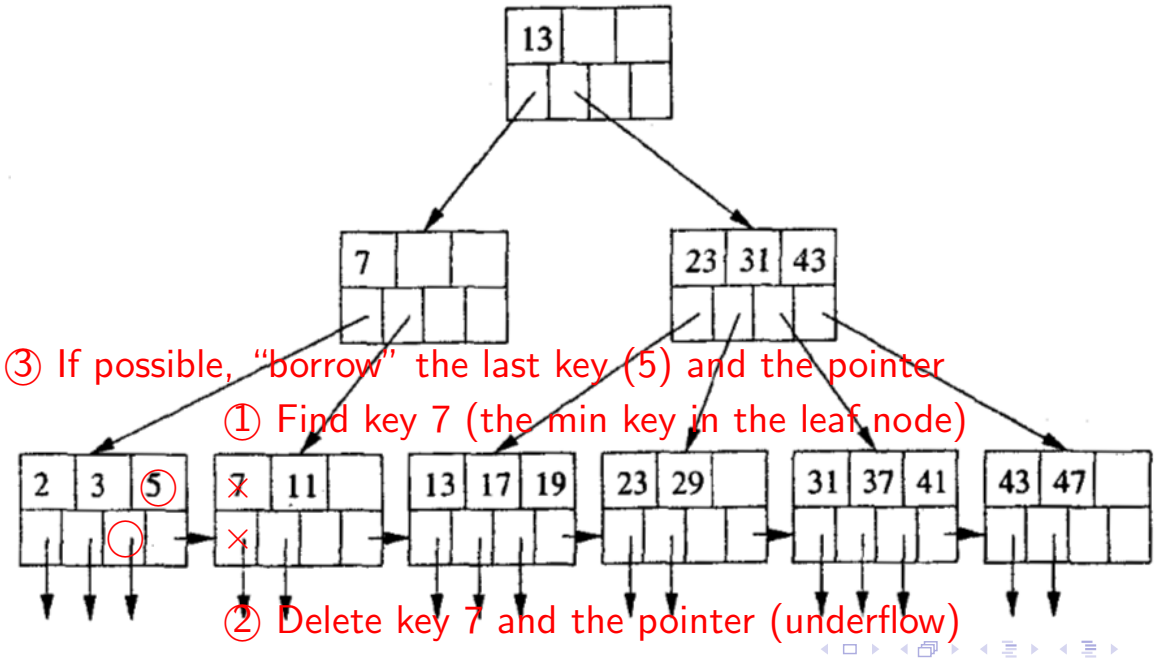
Delete a record with search key 13



Deletion from B+ Trees with Merge

Example (Deletion with Merge)

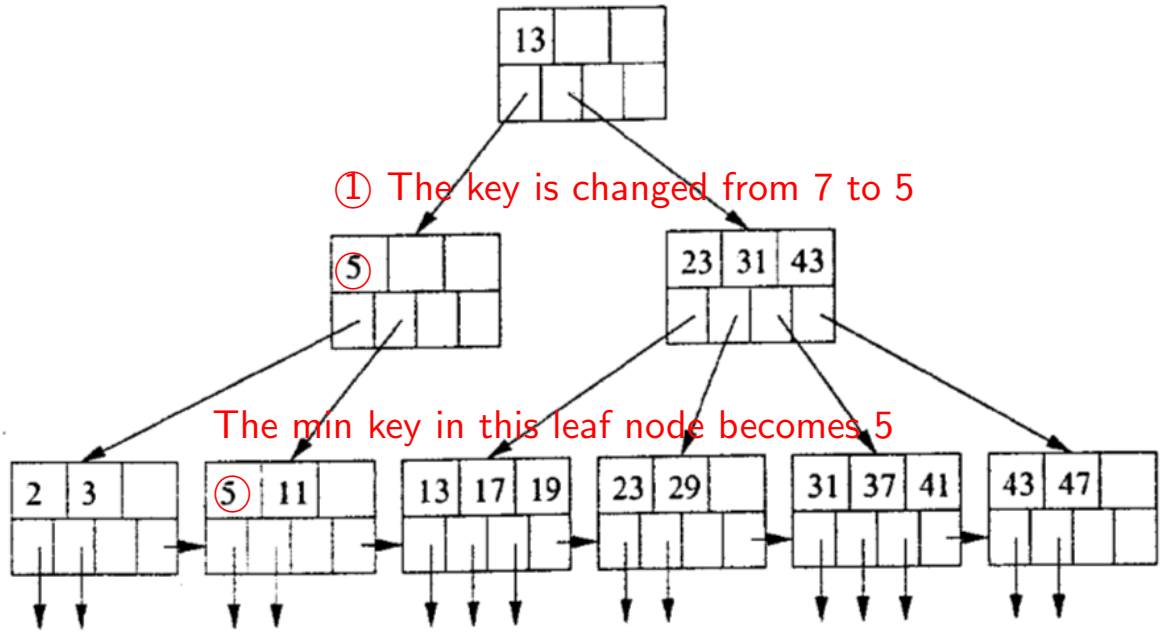
Delete a record with search key 7



Deletion from B+ Trees with Merge (Cont'd)

Example (Deletion with Merge (Cont'd))

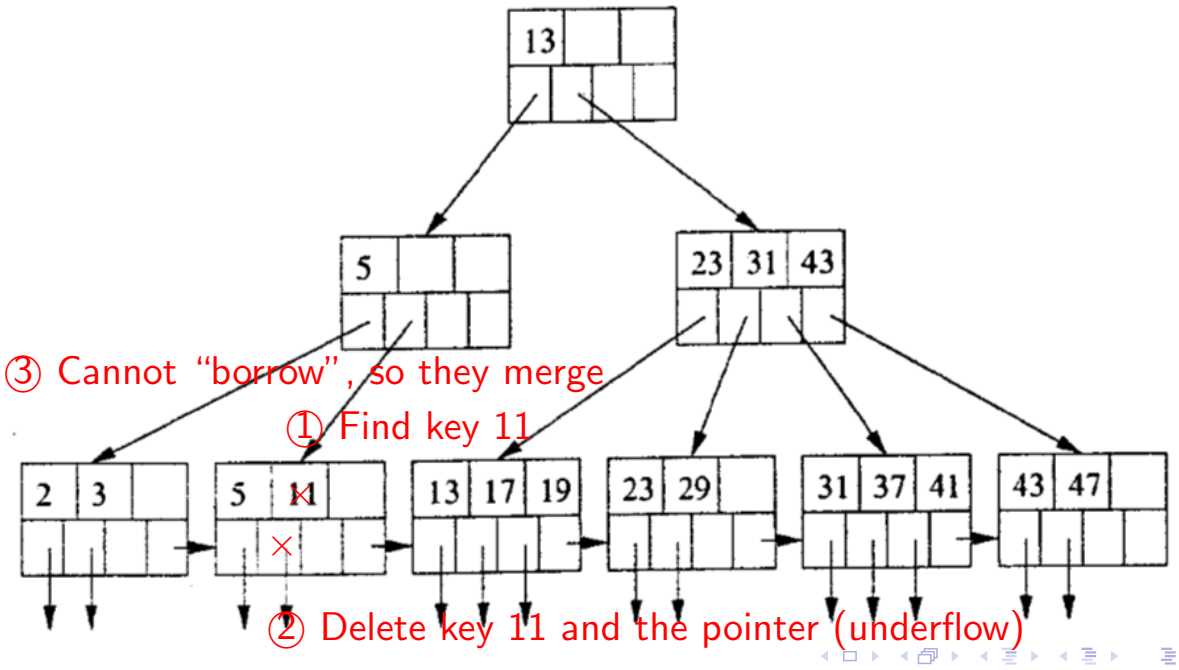
Delete a record with search key 7



Deletion from B+ Trees with Merge (Cont'd)

Example (Deletion with Merge)

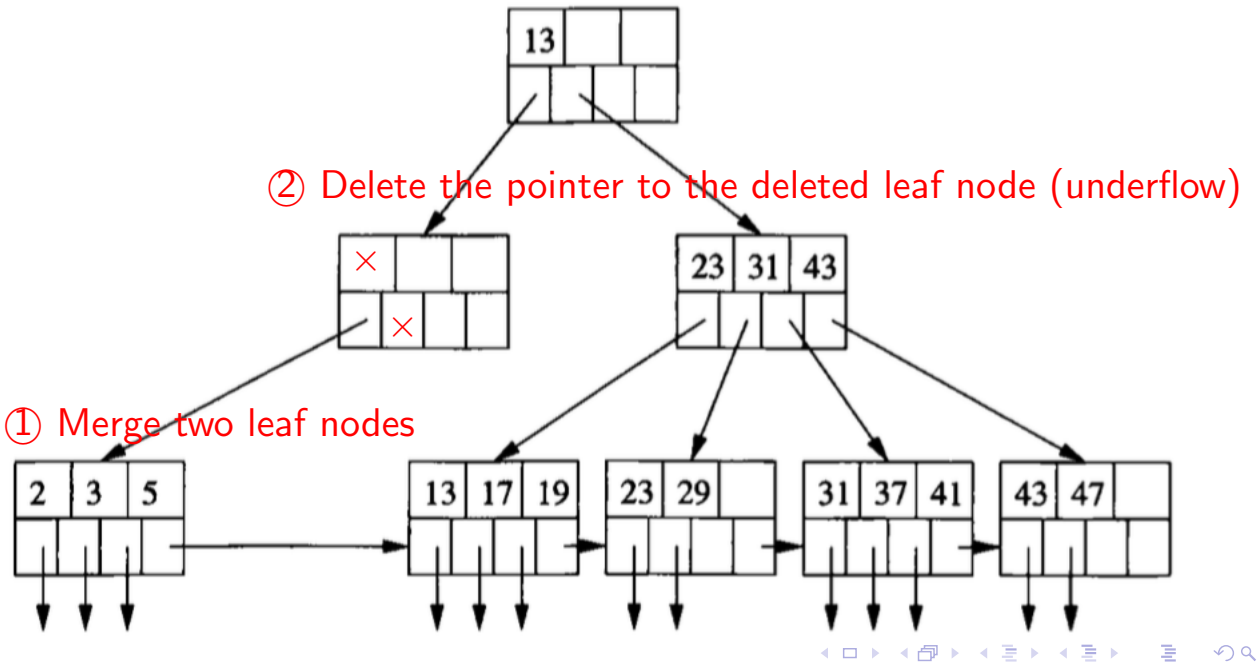
Delete a record with search key 11



Deletion from B+ Trees with Merge (Cont'd)

Example (Deletion with Merge (Cont'd))

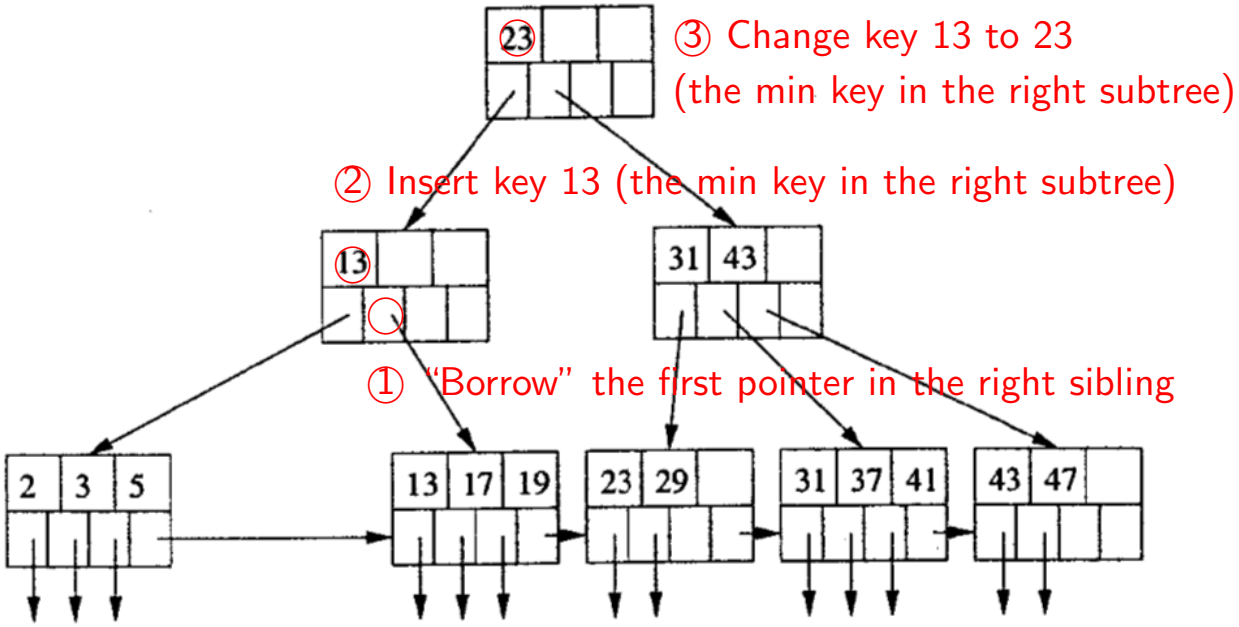
Delete a record with search key 11



Deletion from B+ Trees with Merge (Cont'd)

Example (Deletion with Merge (Cont'd))

Delete a record with search key 11



Key Compression in B+ Trees

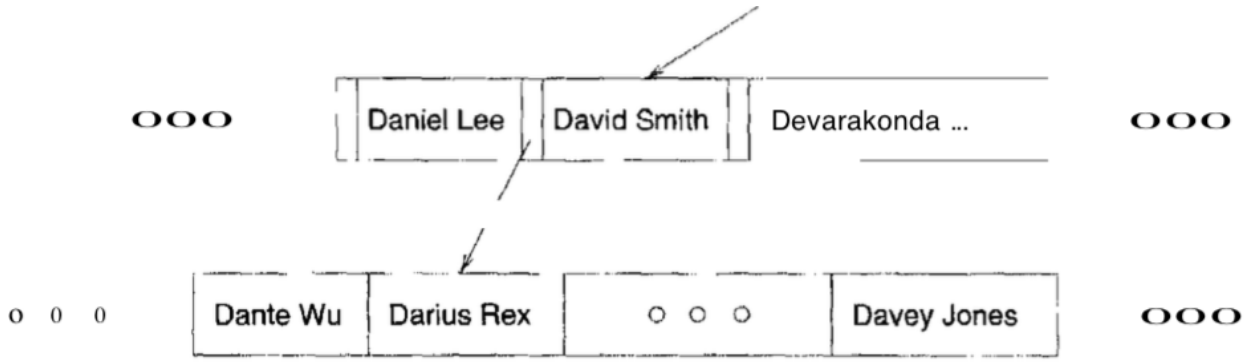
Why Compressing Search Key Values?

- The number of disk I/Os to retrieve a data entry in a B+ tree = the height of the tree $\approx \log_{fan\_out}$(# of data entries)
- The fan-out (扇出) of the tree is the number of index entries fit on a page, which is determined by the size of index entries
- The size of an index entry depends primarily on the size of the search key value
- Search key values are very long $\Longrightarrow$ the fan-out is low $\Longrightarrow$ the tree is high $\Longrightarrow$ the query time is long

Key Compression in B+ Trees (Cont'd)

Basic Idea

- Search key values in index entries are used only to direct traffic to the appropriate leaf
- We need not store search key values in their entirety in index entries



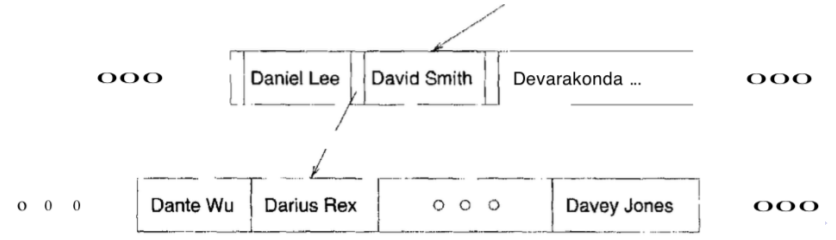
Prefix Key Compression (前缀压缩)

- The prefix key compression technique is supported in many commercial implementations of B+ trees
- 前缀压缩≠前缀索引(第6章)

Example (Prefix Key Compression)

The search key value "David Smith" is compressed into "Davi"

- "Davi" is larger than the largest key value, "Davey Jones", in the subtree to the left of "David Smith"
- "Davi" is smaller than the smallest key value in the subtree to the right of "David Smith"
- "Davi" is the shortest among such prefix of "David Smith"



Bulk-Loading a B+ Tree

Definition (Bulk-loading (批量加载))

Creating a B+ tree on an existing collection of data records

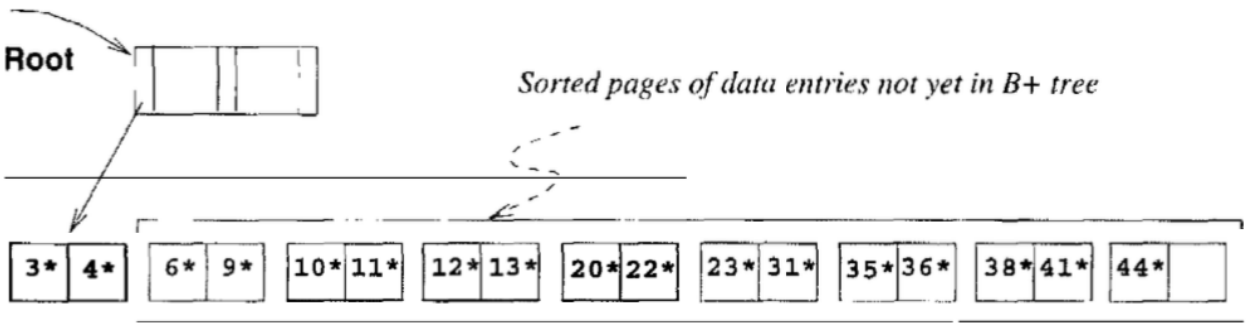
Top-Down Approach

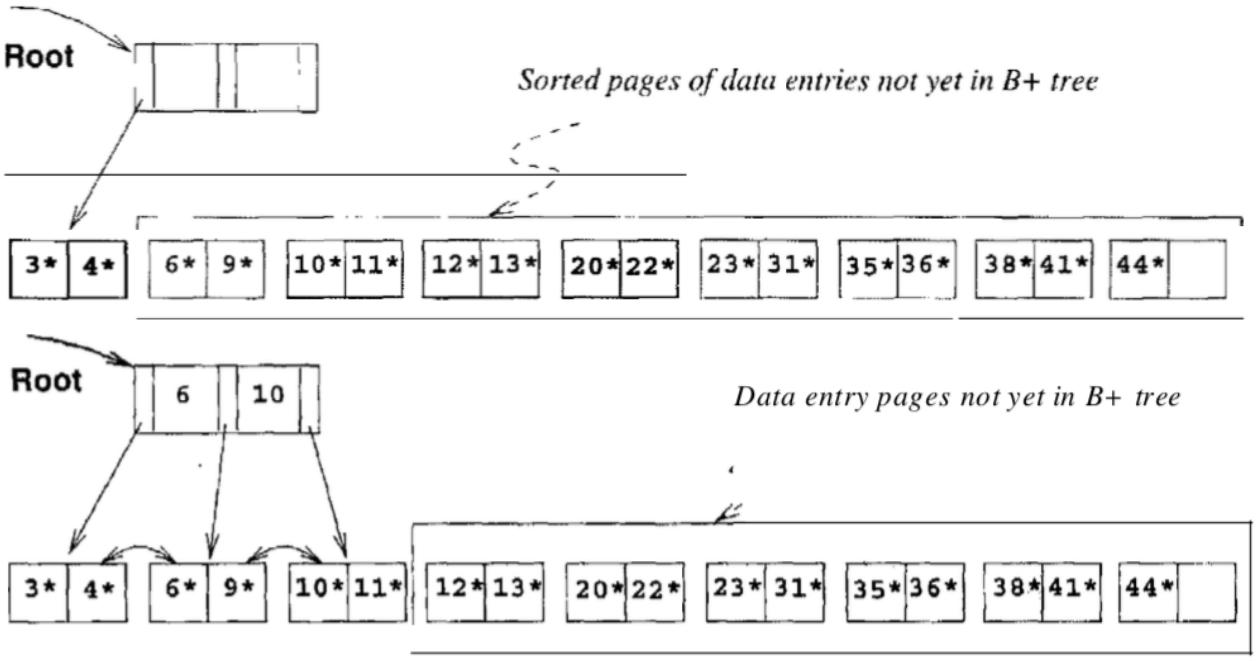
- Method: Insert records one at a time
- Limitation: This approach is likely to be quite expensive because each entry requires to start from the root and go down to the appropriate leaf page

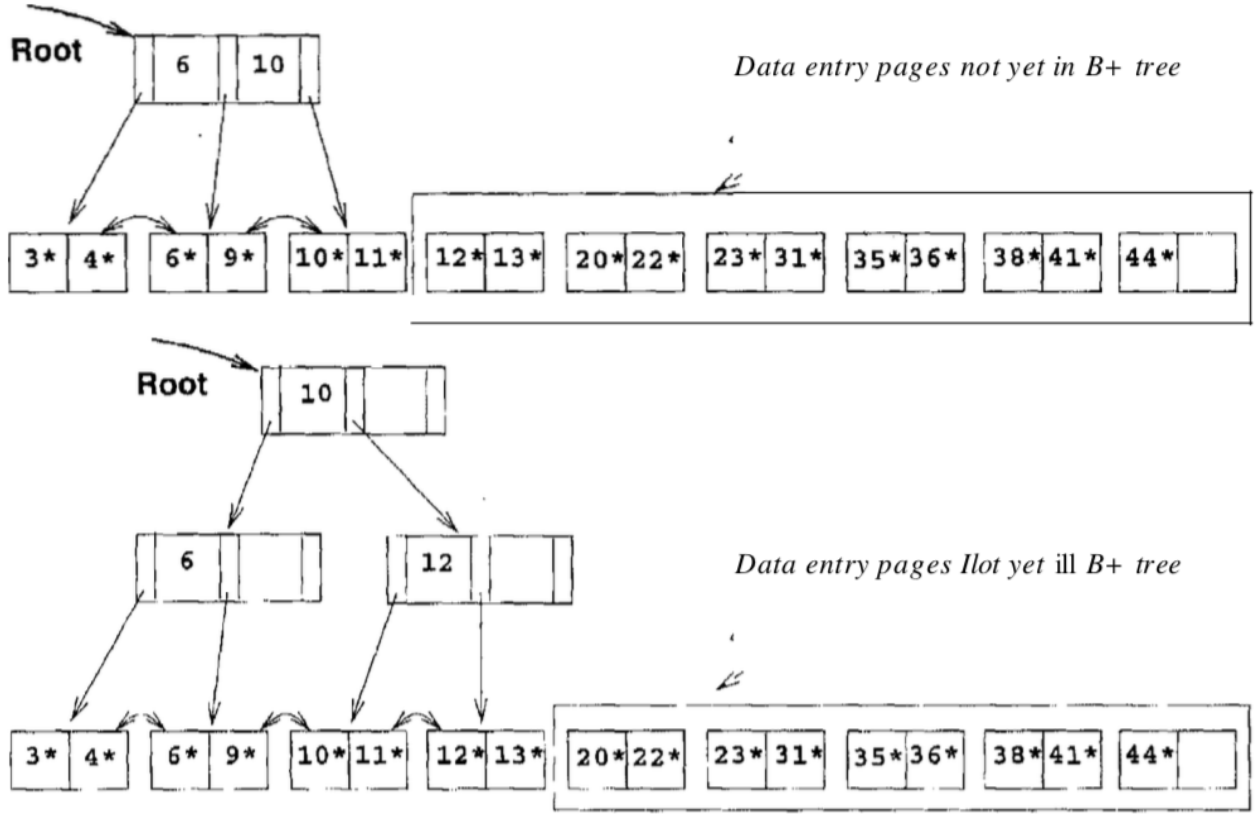
Bulk-Loading a B+ Tree (Cont'd)

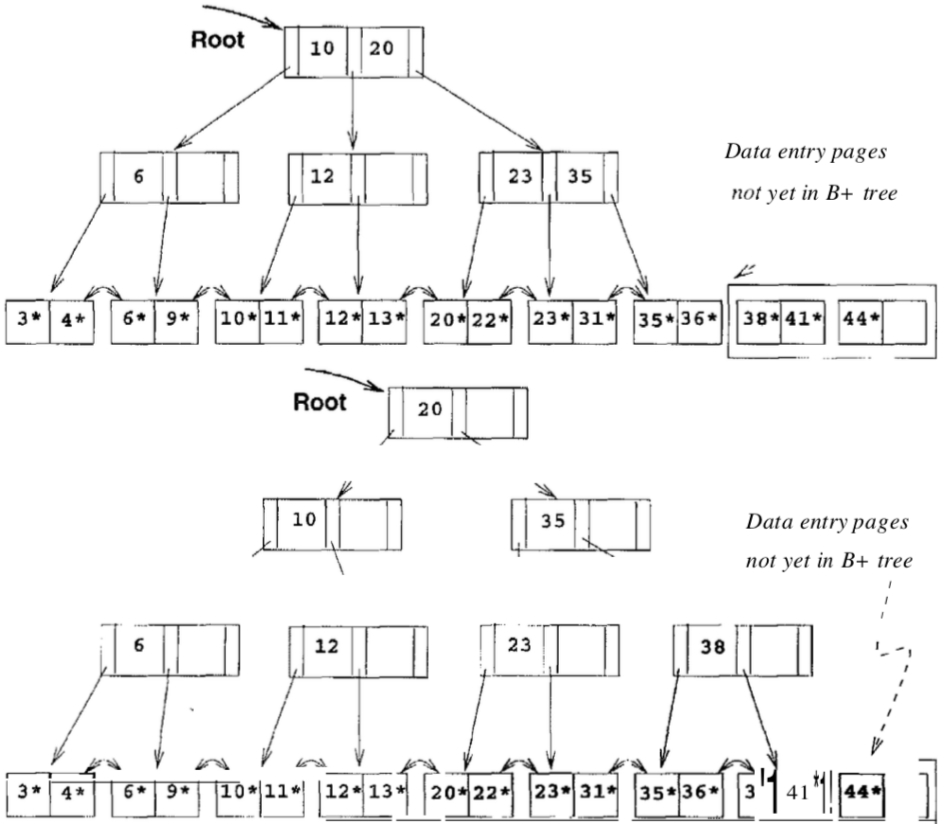
Bottom-Up Approach

1. Sort the entries to be inserted into the B+ tree according to the search key
2. Allocate an empty page to serve as the root and insert a pointer to the first page of sorted entries into it
3. Entries for the leaf pages are always inserted into the right-most index page just above the leaf level. When that page fills up, it is split




Root
Sorted pages of data entries not yet in B+ tree
3* 4*
6* 9*
10* 11*
12* 13*
20* 22*
23* 31*
35* 36*
38* 41*
44*
Root
6
10
Data entry pages not yet in B+ tree
3* 4*
6* 9*
10* 11*
12* 13*
20* 22*
23* 31*
35* 36*
38* 41*
44*


Root
6
10
Data entry pages not yet in B+ tree
3* 4*
6* 9*
10* 11*
12* 13*
20* 22*
23* 31*
35* 36*
38* 41*
44*
Root
10
6
12
Data entry pages Ilot yet ill B+ tree
3* 4*
6* 9*
10* 11*
12* 13*
20* 22*
23* 31*
35* 36*
38* 41*
44*

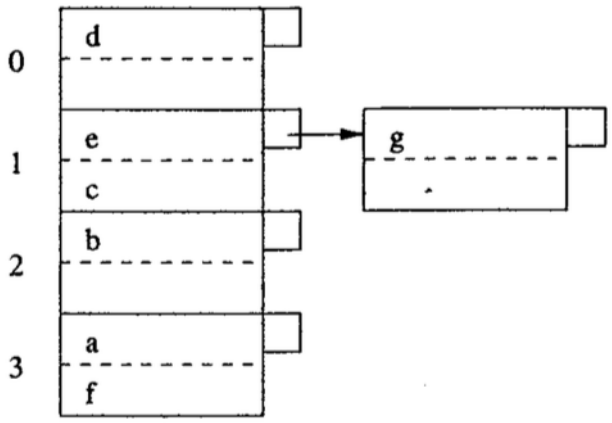


Splits occur only on the right-most path from the root to the leaf level

7.7 Hash-based Index Structures

Secondary-Storage Hash Tables

- The bucket array consists of blocks, rather than pointers to the headers of lists
- A record with search key K is put in the block for the bucket numbered $h(K)$, where h is the hash function
- If a bucket overflows, then a chain of overflow blocks can be added to the bucket to hold more records
- Operations: lookup (查找), insertion (插入), deletion (删除)



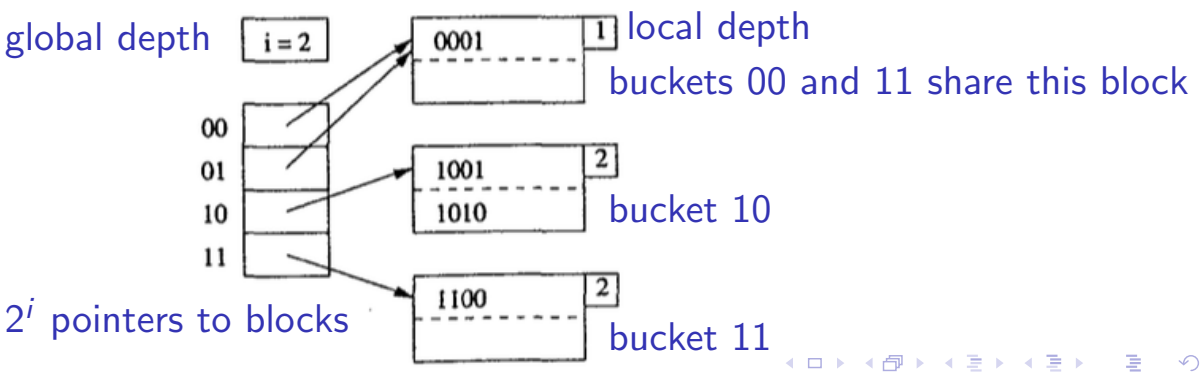
Types of Secondary-Storage Hash Tables

- Static Hash Tables (静态哈希表): the number of buckets never changes
- Dynamic Hash Tables (动态哈希表): the number of buckets is allowed to vary so that there is about one block per bucket
  - Extensible hash tables (可扩展哈希表)
  - Linear hash tables (线性哈希表)

7.7 Hash-based Index Structures
Extensible Hash Tables

Extensible Hash Tables (可扩展哈希表)

- For each key K, $h(K)$ consists of k bits
- A record with search key K is placed in the bucket numbered by the first i bits of $h(K)$ (i is called the global depth)
- Certain buckets can share a block if the total number of records in those buckets can fit in the block
- There is an array of 2^i pointers to blocks
- The number j (local depth) appearing in the nub of each block indicates how many bits of $h(K)$ is used to determine membership of records in this block. We must have local depth $j \leq$ global depth i

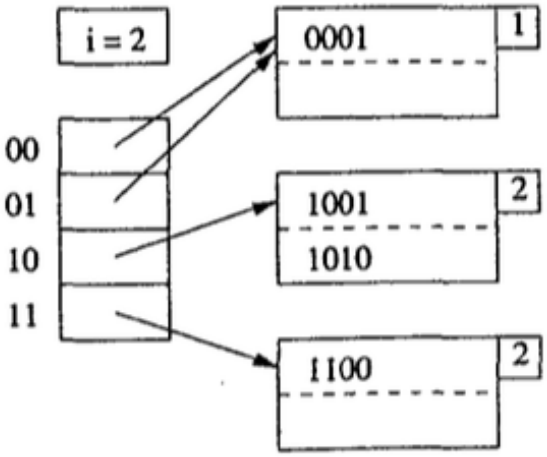


Lookup in Extensible Hash Tables

Example (Lookup)

Find a record whose key hashes to 1010

1. The global depth is 2. The record could be found in the bucket numbered 10.
2. Find the record in the block of bucket 10

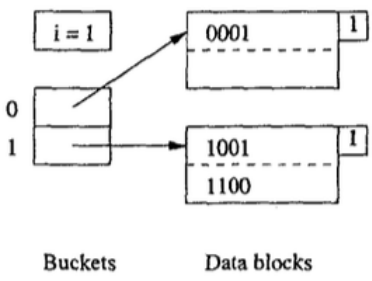


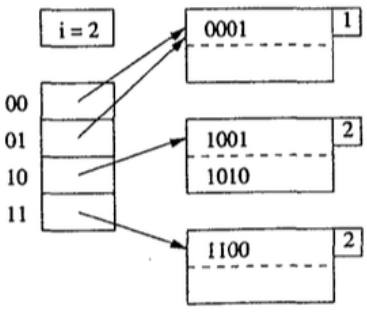
Insertion into Extensible Hash Tables

Example (Insertion)

Insert a record whose key hashes to 1010

1. The global depth is currently 1. The record should be inserted into bucket 1.
2. After insertion, bucket 1 will overflow, so the global depth is increased by 1 (the number of buckets is doubled)
3. Create a new block for bucket 11 and set pointers to the blocks
4. Redistribute the records in bucket 1 to buckets 10 and 11. Insert the record to bucket 10
5. Increase the local depth of buckets 10 and 11 by 1



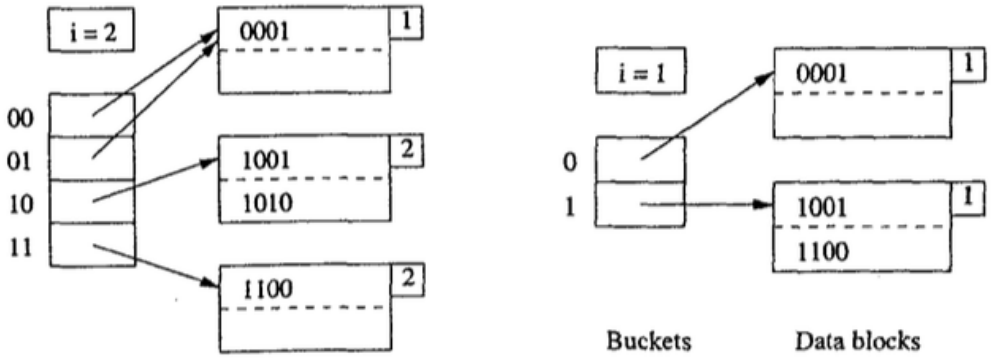


Deletion from Extensible Hash Tables

- The deleted data entry is located and removed
- If the deletion leaves the bucket empty, it can be merged with its split image, although this step is often omitted in practice
- We can halve the directory and reduce the global depth, although this step is not necessary for correctness

Example (Deletion)

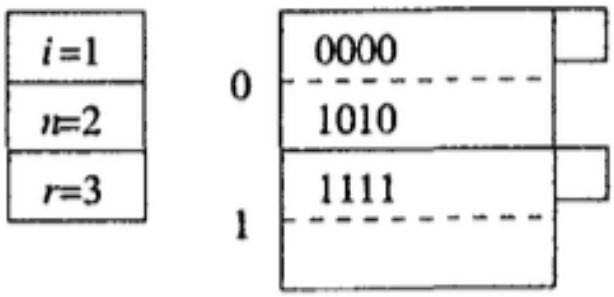
Delete a record whose key hashes to 1010



7.7 Hash-based Index Structures
Linear Hash Tables

Linear Hash Tables (线性哈希表)

- n: the current number of buckets
- r: the current number of records
- A record with search key K is placed in the bucket determined by the last i bits of $h(K)$
  - m: the integer represented by the last i bits of $h(K)$
  - If $m < n$, the record is placed in bucket m
  - If $n \leq m < 2^i$, the record is placed in bucket $m - 2^{i-1}$
- It is required that $r/bn \leq \theta$, where b is the maximum number of records a block can contain, and $0 < \theta < 1$ is a threshold (e.g., $\theta = 85\%$)

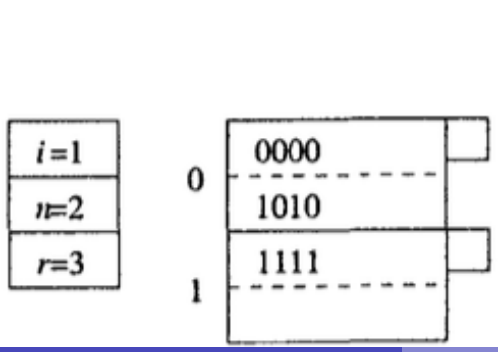


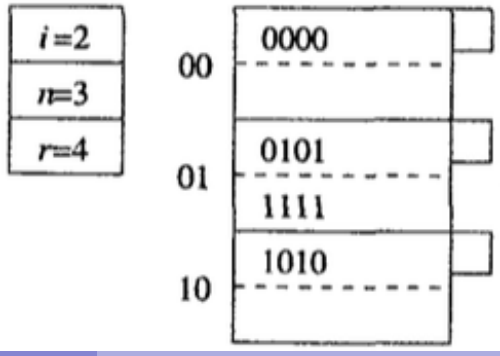
Insertion into Linear Hash Tables

1. A record with search key K is placed in the bucket determined by the last i bits of $h(K)$
   - m: the integer represented by the last i bits of $h(K)$
   - If $m < n$, the record is placed in bucket m
   - If $n \leq m < 2^i$, the record is placed in bucket $m - 2^{i-1}$
2. If $r/bn > \theta$, we add a new bucket numbered $n + 1$
3. If $n + 1 \geq 2^{i-1}$, split the bucket numbered $n + 1 - 2^{i-1}$

Example (Insertion)

Insert a record whose key hashes to 0101 ($\theta = 0.85$)

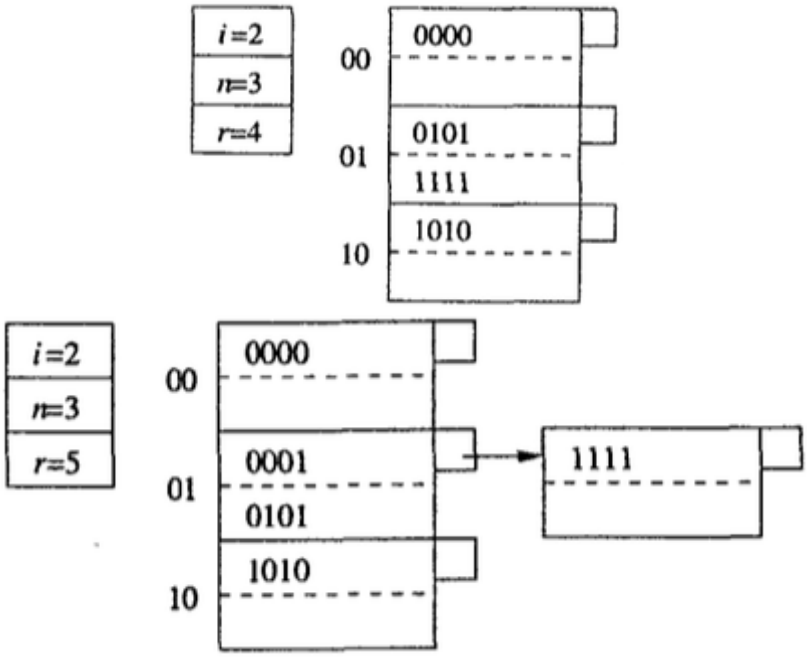




Insertion into Linear Hash Tables (Cont'd)

Example (Insertion)

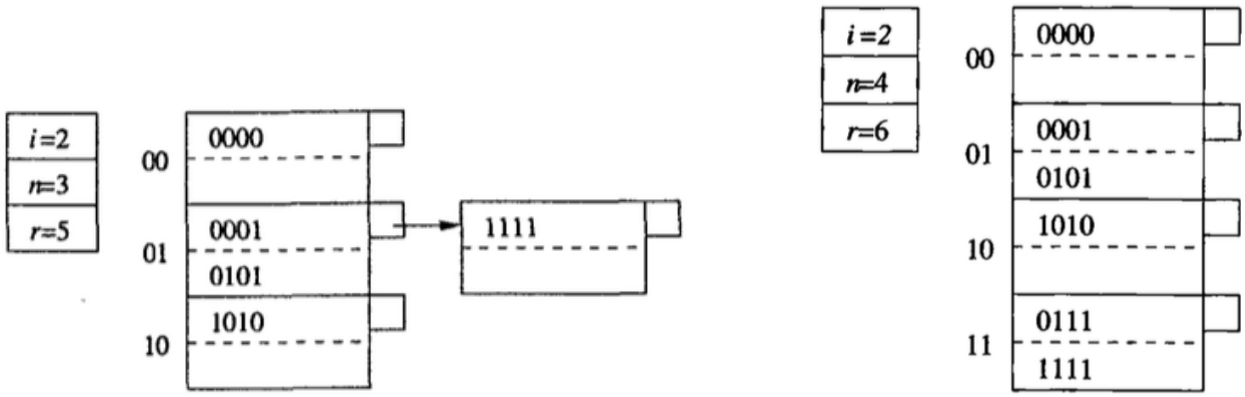
Insert a record whose key hashes to 0001 ($\theta = 0.85$)



Insertion into Linear Hash Tables (Cont'd)

Example (Insertion)

Insert a record whose key hashes to 0111 ($\theta = 0.85$)



Deletion from Linear Hash Tables

Deletion is essentially the inverse of insertion

Summary

1. 7.5 Indexes on Sequential Files
2. 7.6 Tree-based Index Structures
   - B+ Trees
3. 7.7 Hash-based Index Structures
   - Extensible Hash Tables
   - Linear Hash Tables