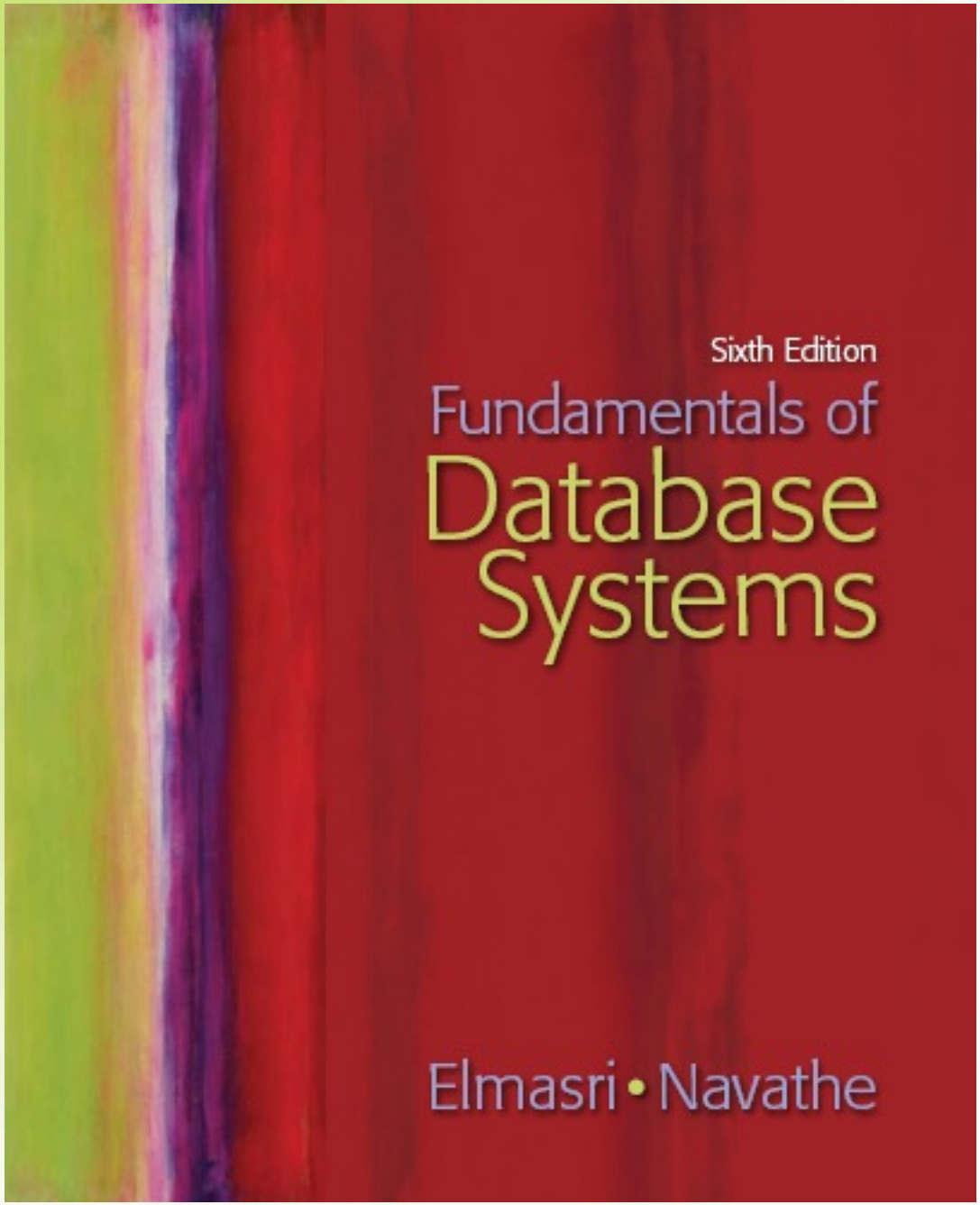
Chapter 18

Indexing Structures for Files



Addison-Wesley
is an imprint of
PEARSON

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries <**field value, pointer to record**>, which is ordered by field value
- The index is called an access path on the field.

Indexes as Access Paths (cont.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values

Indexes as Access Paths (cont.)

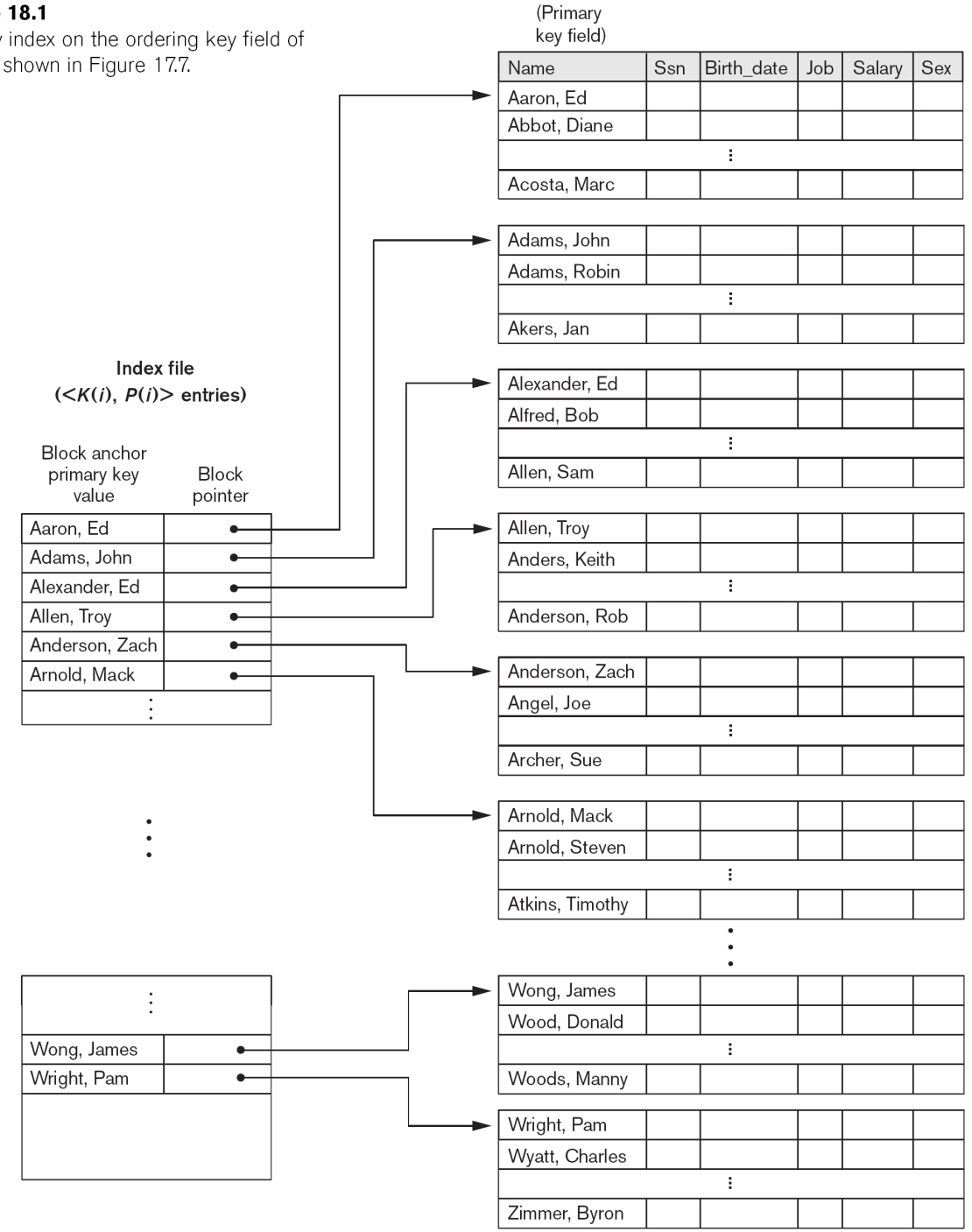
- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that:
  - record size R=150 bytes block size B=512 bytes r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes. Then:
  - index entry size R_I=(V_{SSN}+ P_R)=(9+7)=16 bytes
  - index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ Bfr_I)= (30000/32)= 938 blocks
  - binary search needs $\log_2 bI = \log_2 938$= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b = \log_2 30000$= 15 block accesses

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
  - A similar scheme can use the *last record* in a block.
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

Primary Index on the Ordering Key Field

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



SSTables and LSM-Trees

- Inspired by Google's Bigtable paper
- Log Structured Merge-Trees
- Sorted String Table
  - Sorted log of records (key-value pairs)
  - Merge logs periodically (will also compact)
  - Since records are sorted, index can be sparse
- In-memory balanced search tree (AVL, red-black)
  - Memtable
  - Efficient adding/deleting records maintaining order
  - Dump into new SSTable when size reaches threshold

SSTables and LSM-Trees

- Adding records:
  - Add key-value pair to memtable
- Searching records:
  - First look in memtable
  - If not found, search most recent log
  - If not found search the second most recent log, etc.

Compaction and Merging in SSTables

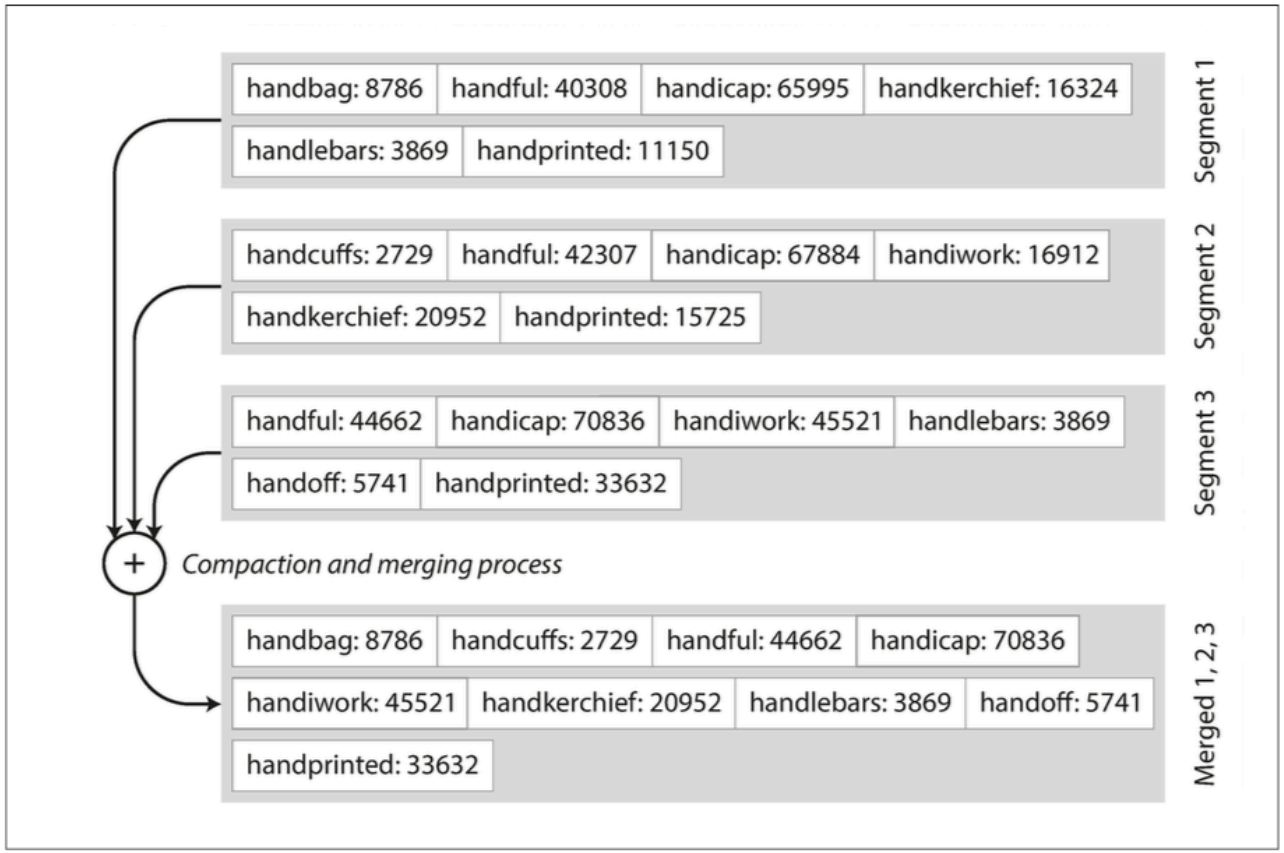


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

SSTable with index

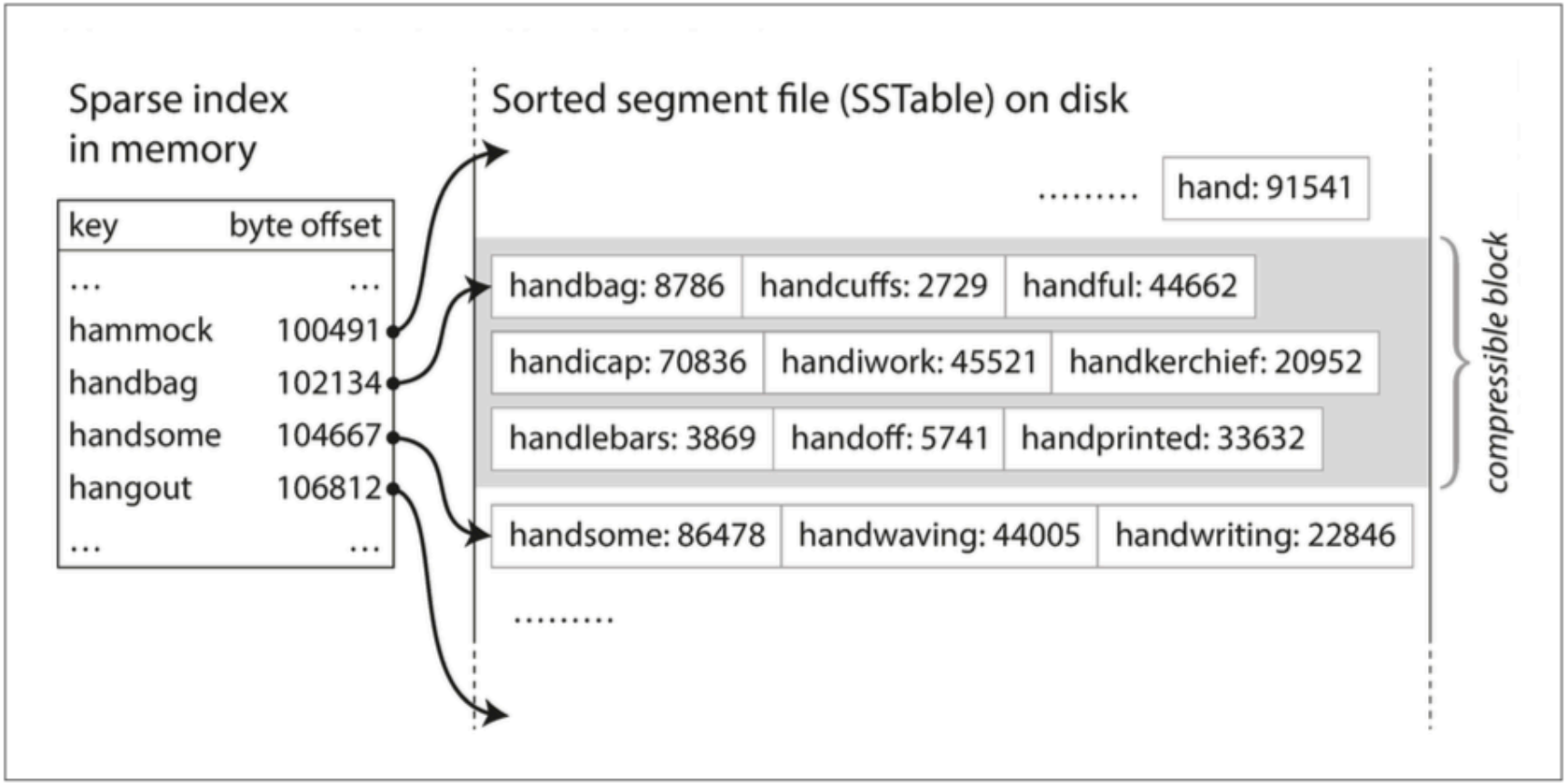


Figure 3-5. An SSTable with an in-memory index.

Types of Single-Level Indexes

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

A Clustering Index Example

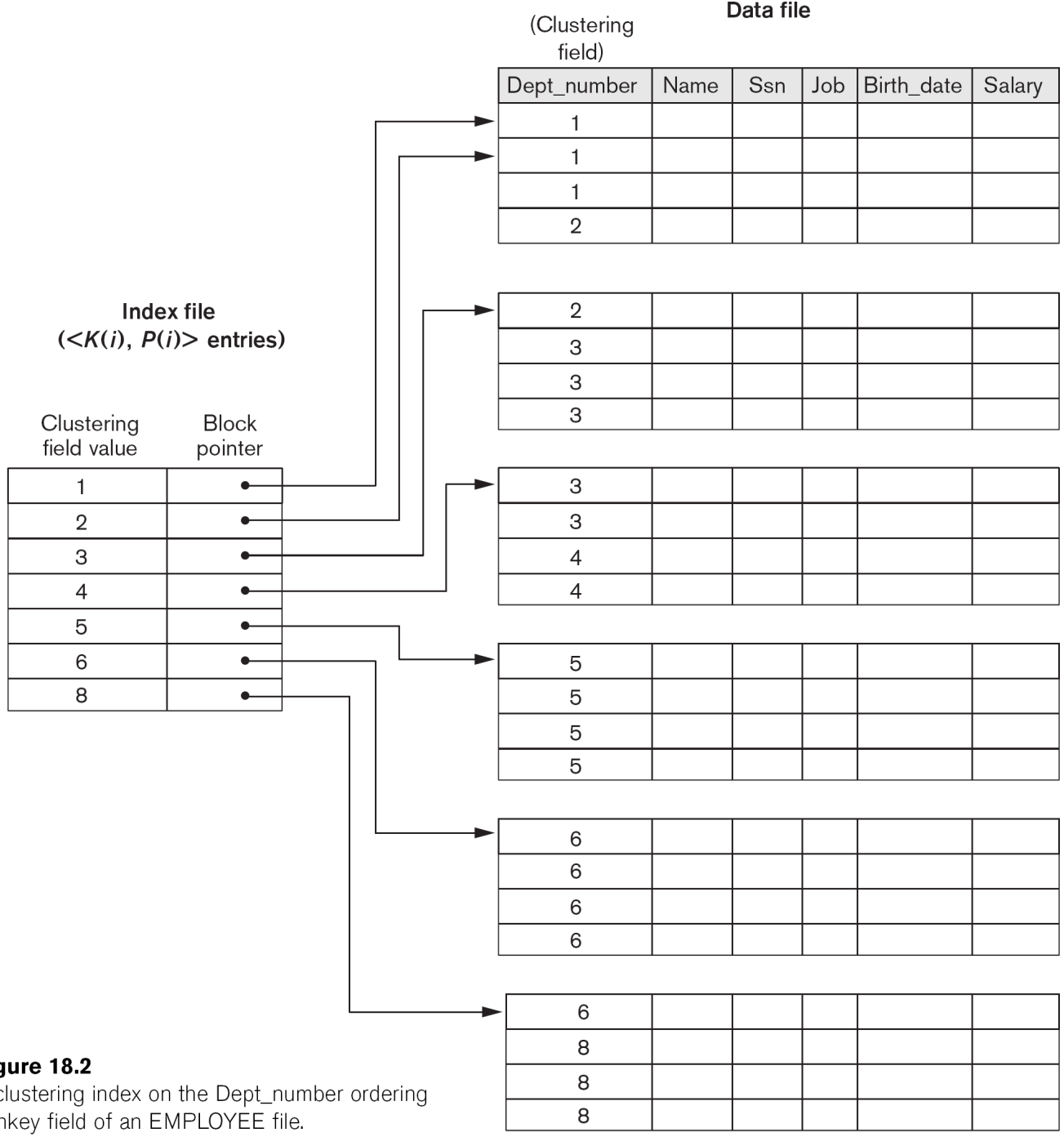
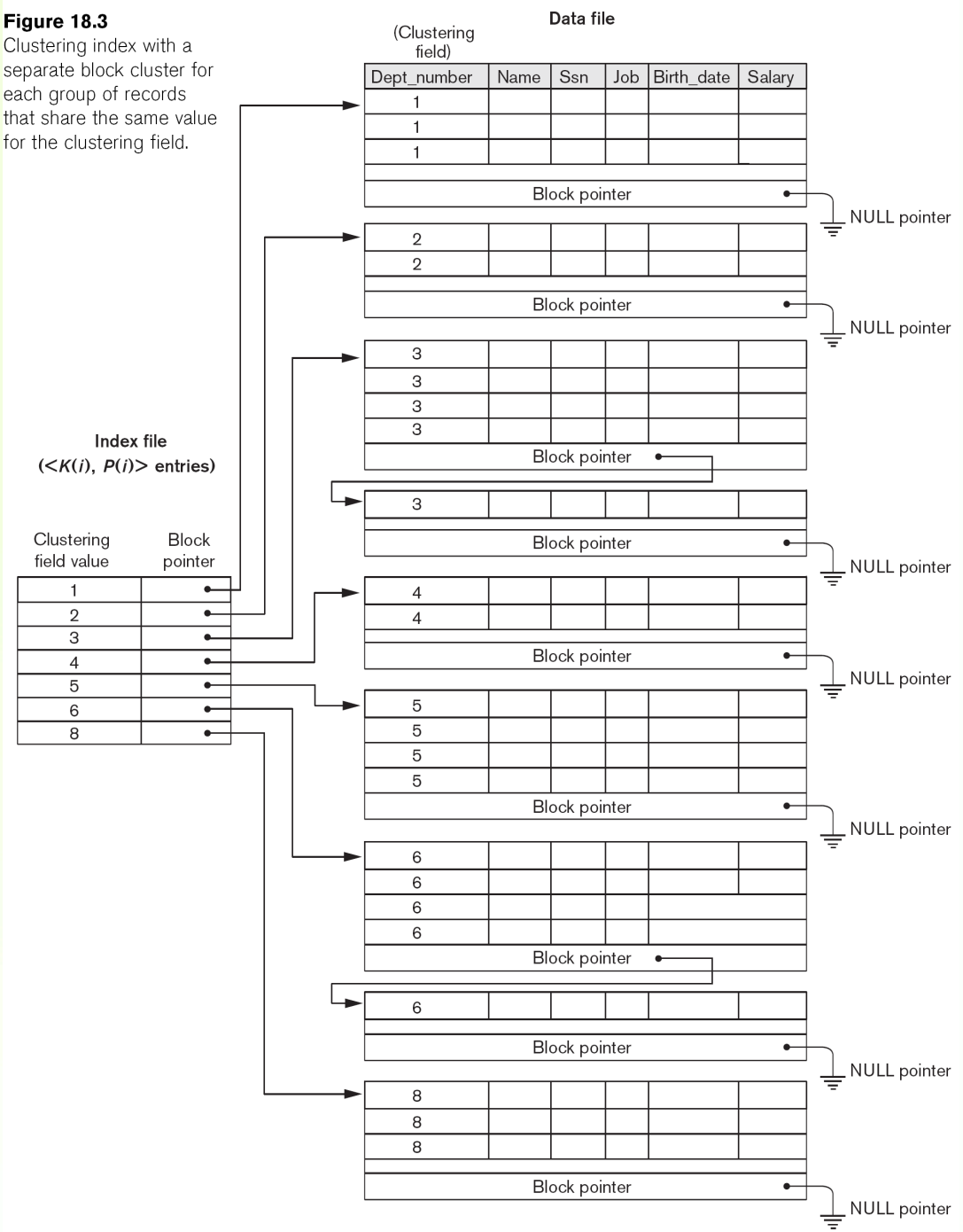


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Another Clustering Index Example

Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

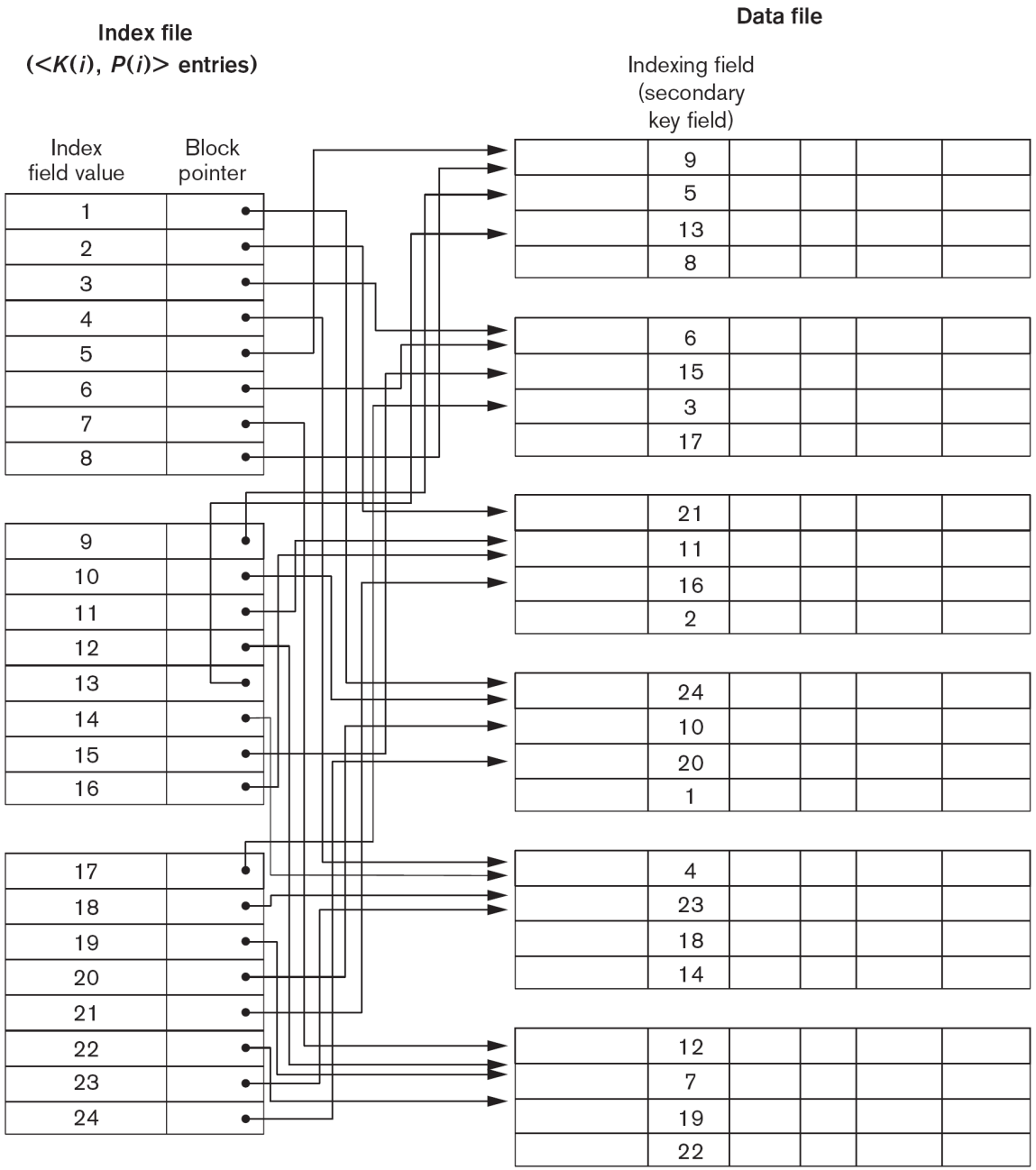


Types of Single-Level Indexes

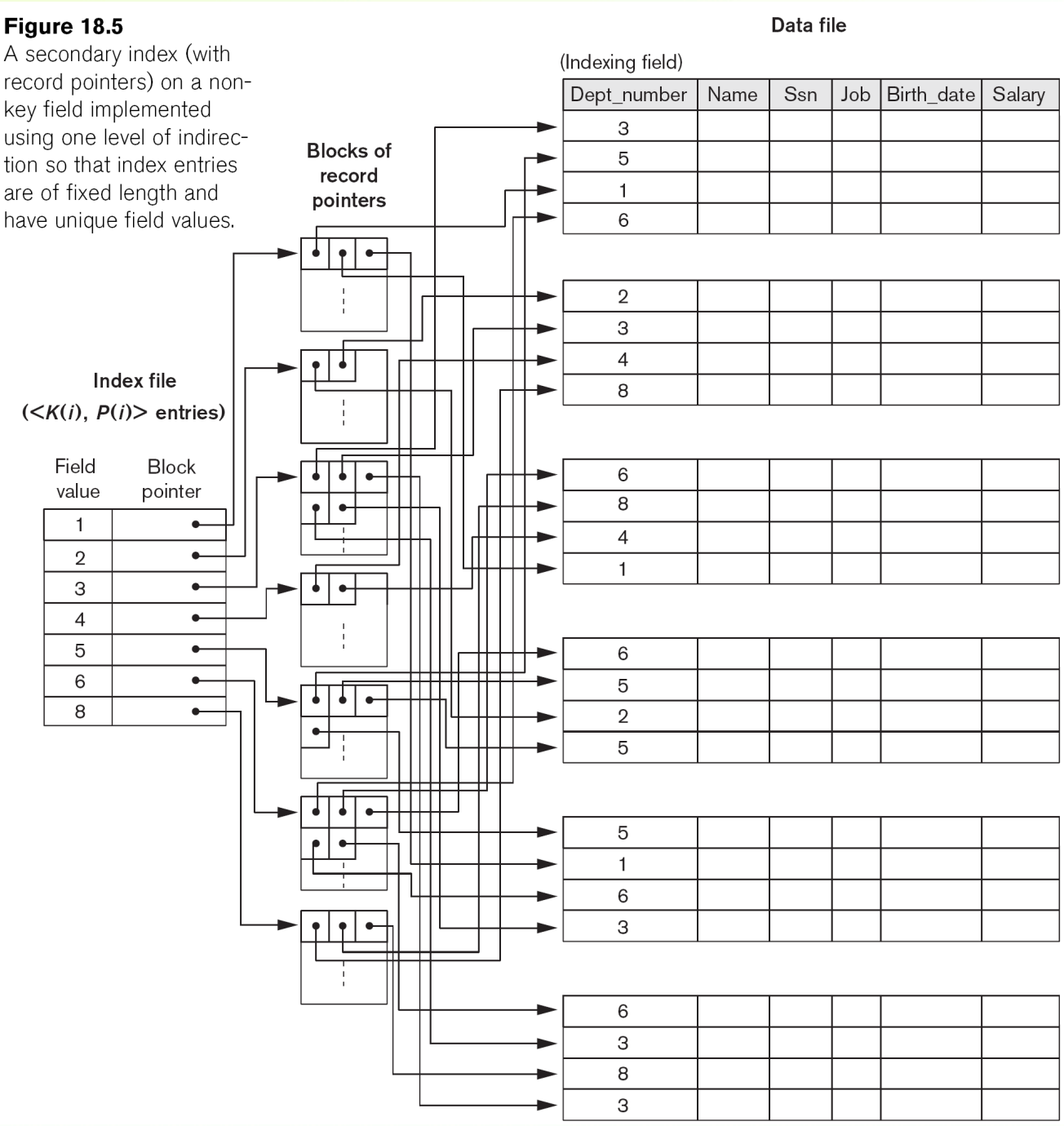
- Secondary Index
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
    - The second field is either a **block** pointer or a record pointer.
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

Example of a Dense Secondary Index

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Example of a Secondary Index



Properties of Index Types

Table 18.2 Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

Clustered vs Non-Clustered

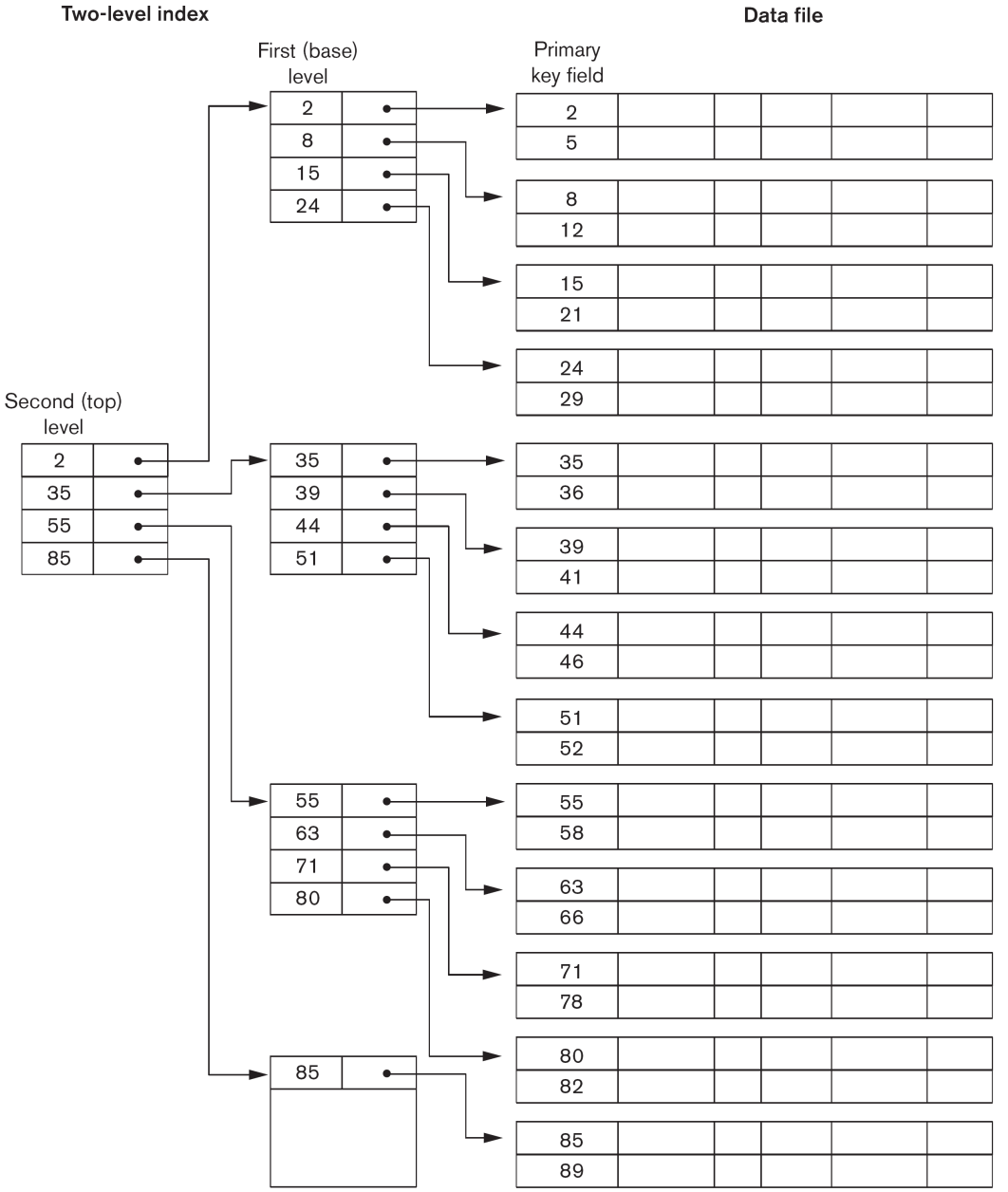
- Clustered Index
  - Type of index where the table records are physically re-ordered to match the index
    - Primary and Clustering
  - Efficient on columns that are searched for a range of values
  - After the row with first value is found using a clustered index, rows with subsequent index values are guaranteed to be physically adjacent
  - Only one per table
- Non-Clustered Index
  - Table records are not physically ordered to match index
    - Secondary
  - Multiple per table possible
- Generally, when a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

A Two-Level Primary Index

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



Multi-Level Indexes

- Such a multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

A Node in a Search Tree with Pointers to Subtrees Below It

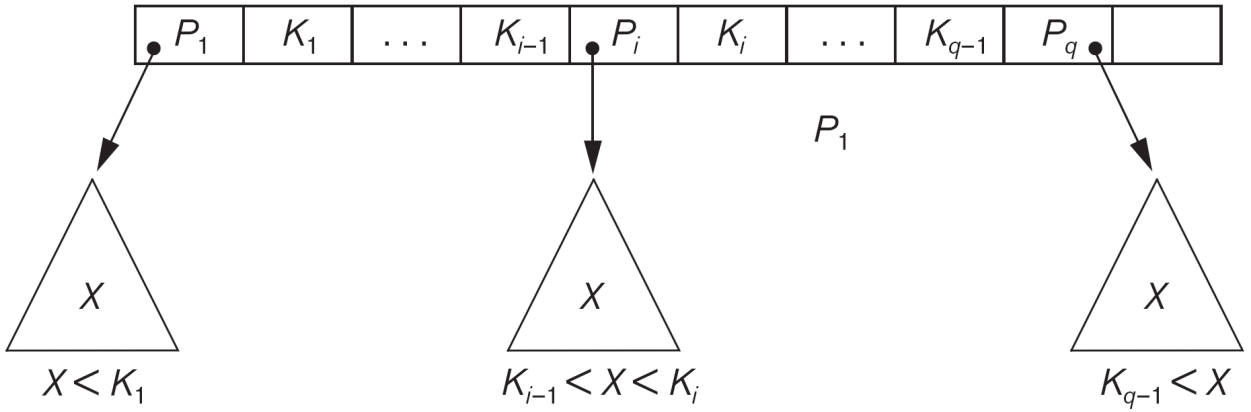
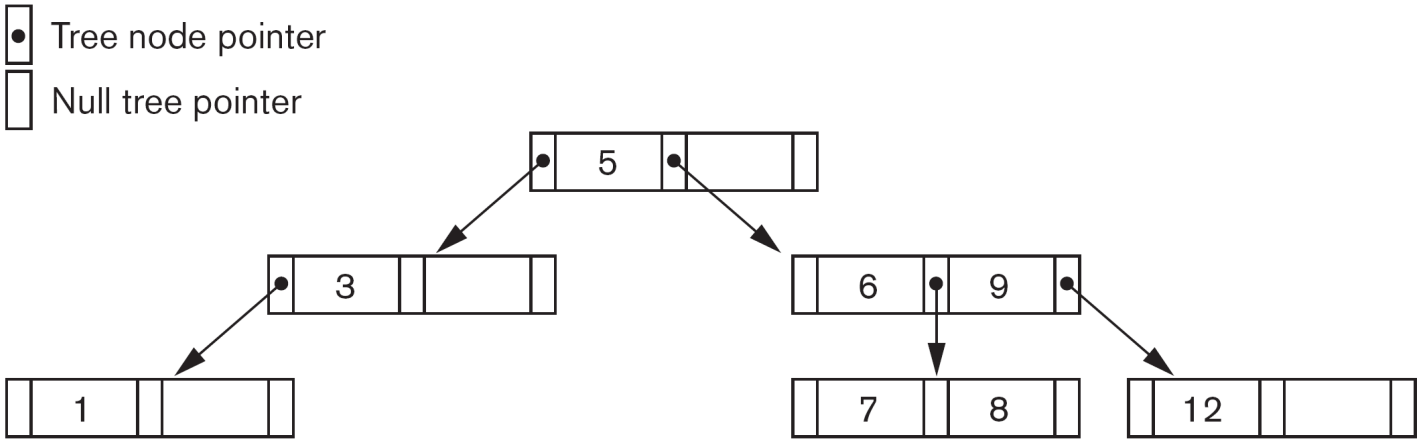


Figure 18.8
A node in a search tree with pointers to subtrees below it.

Figure 18.9
A search tree of order $p = 3$.



Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

Dynamic Multilevel Indexes Using B-Trees and B+-Trees (cont.)

- An insertion into a node that is not full is quite efficient
  - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

B-Trees

- Generalization of a binary search tree
  - A node can have more than two children (variable number of children within a predefined range)
  - During additions and deletions, nodes may be split or merged to stay within range
  - Internal nodes of a B-tree contain a number of keys
    - Keys act as separation values which divide its subtrees
    - If an internal node has 3 child nodes it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.
- Optimized for systems that read and write large blocks of data (good for external memory)

B-Trees

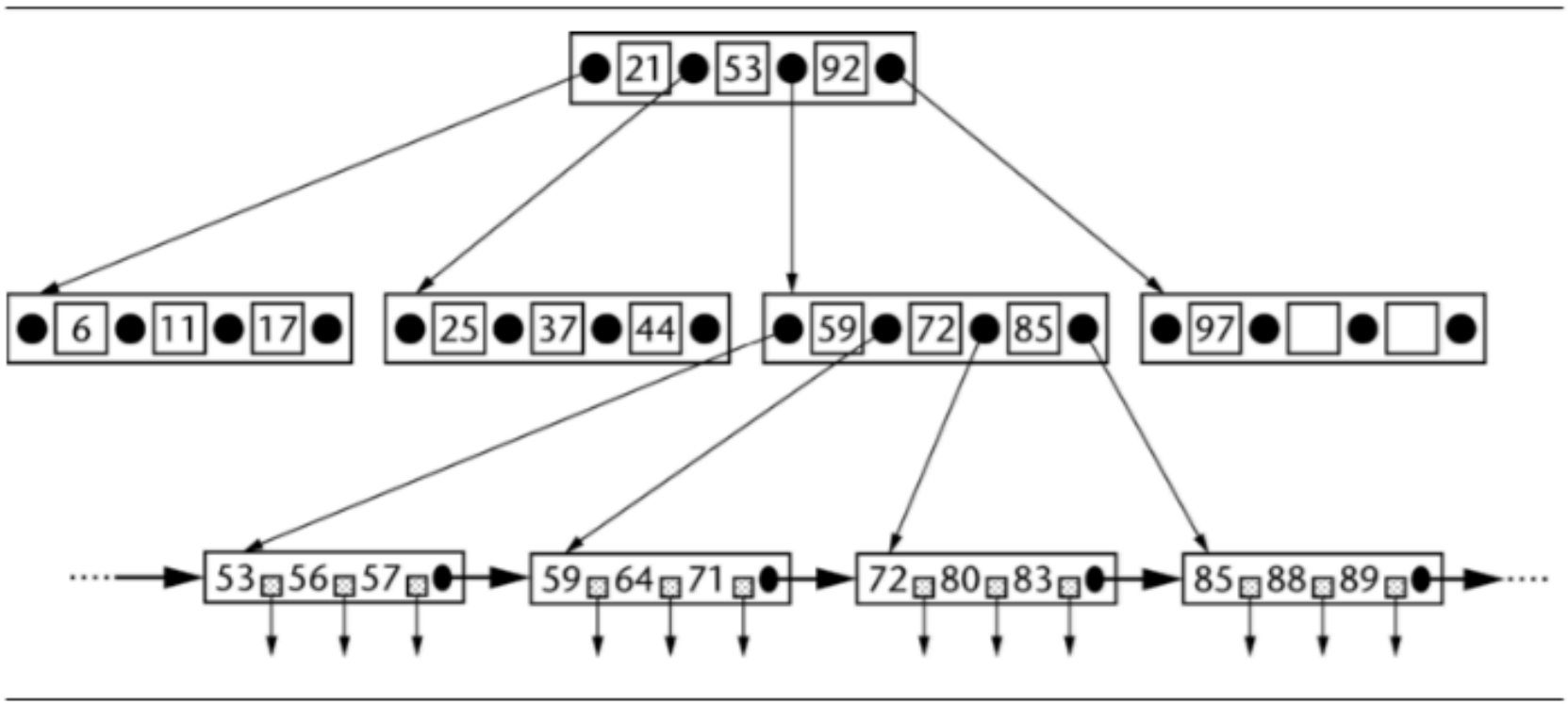


Figure 2.1 B+tree configuration (order four, height three).

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B-tree Structures

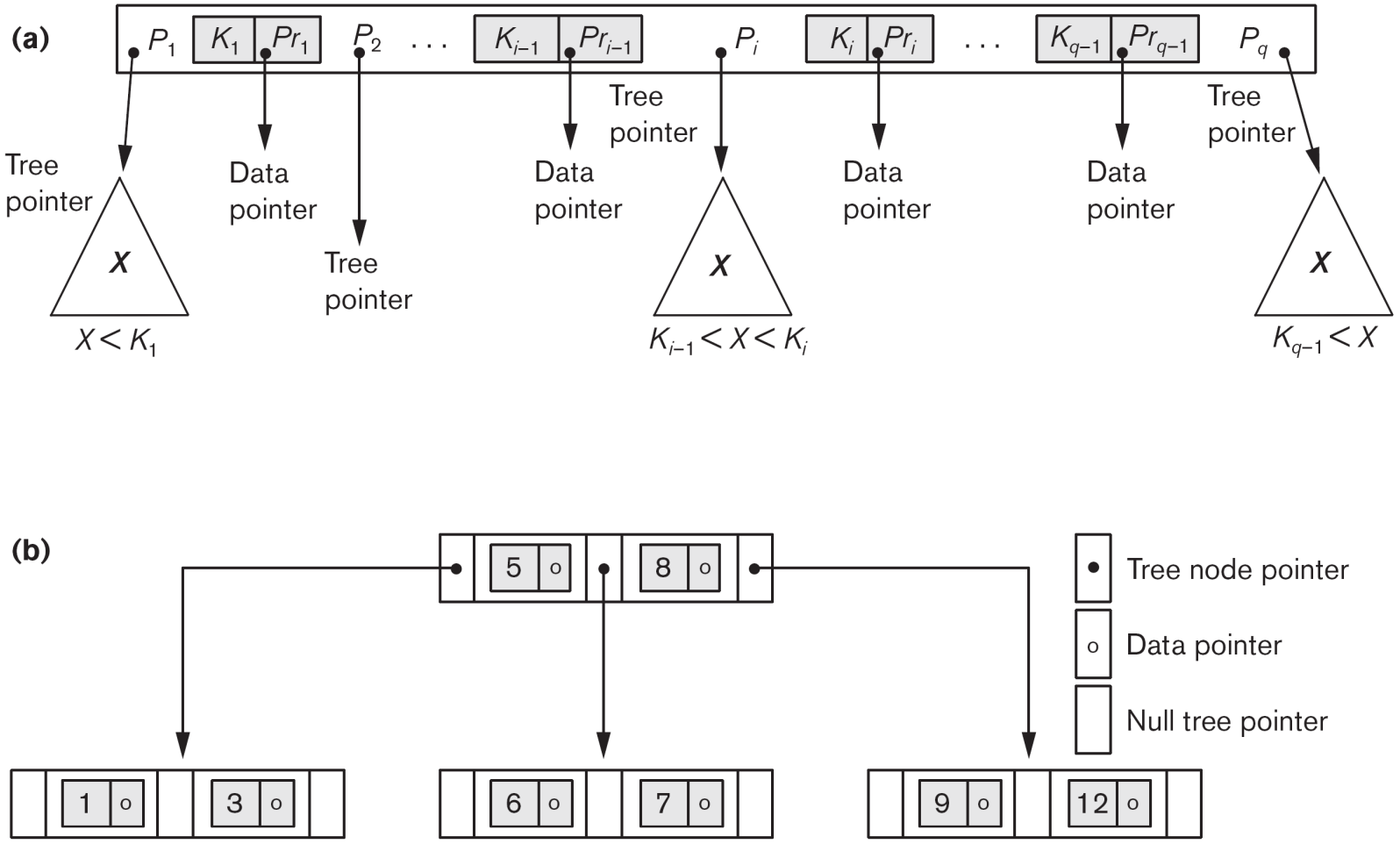
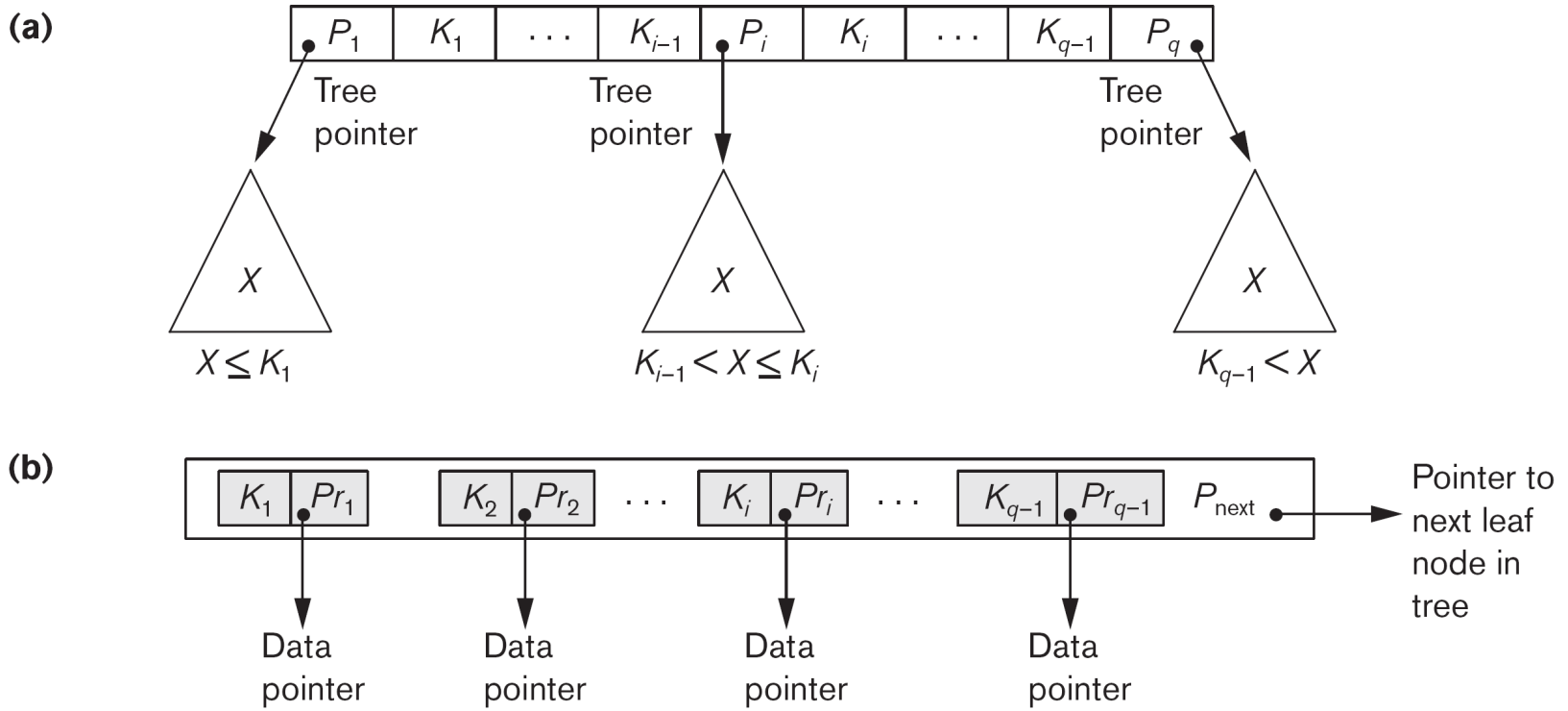


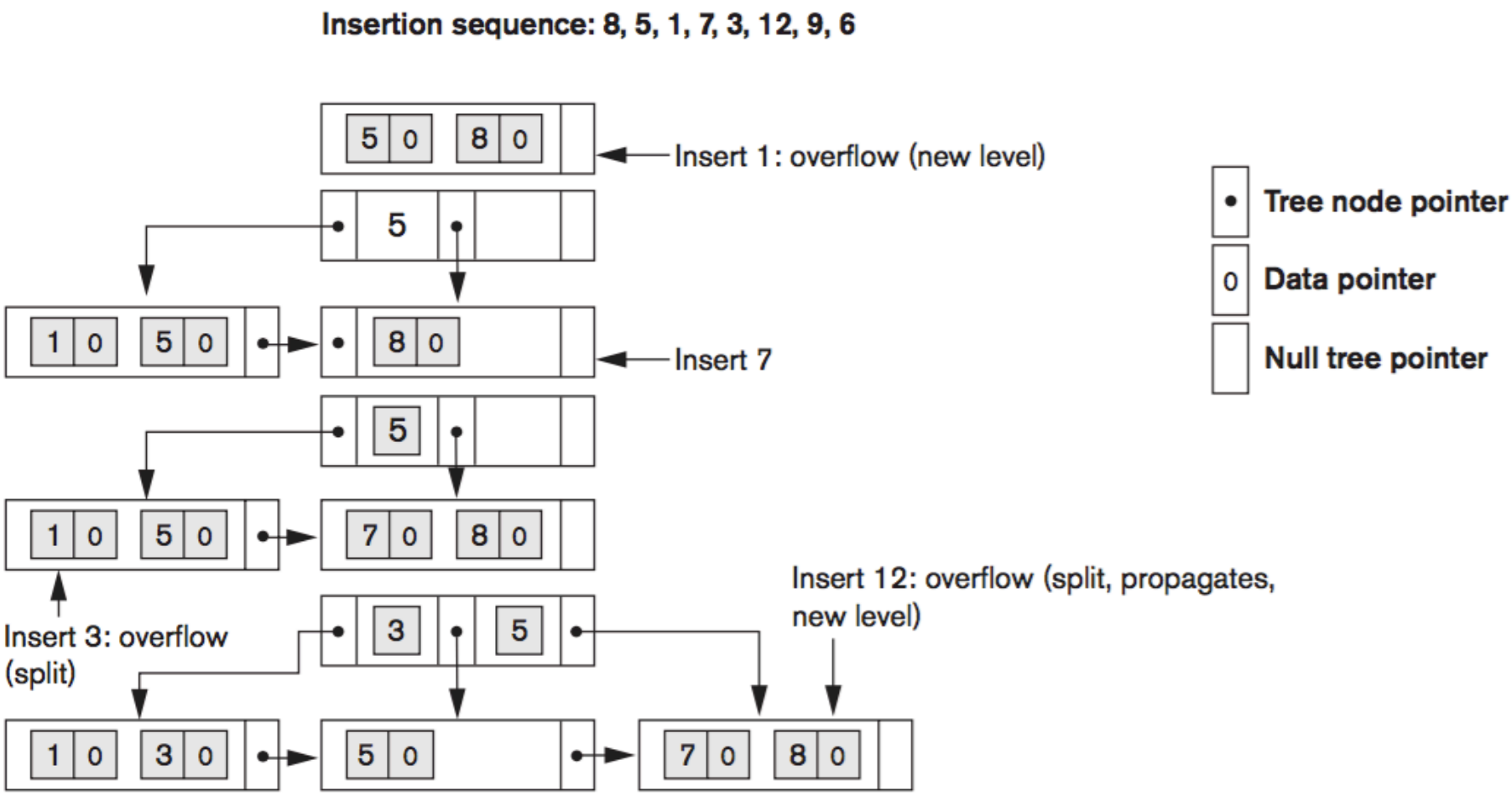
Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

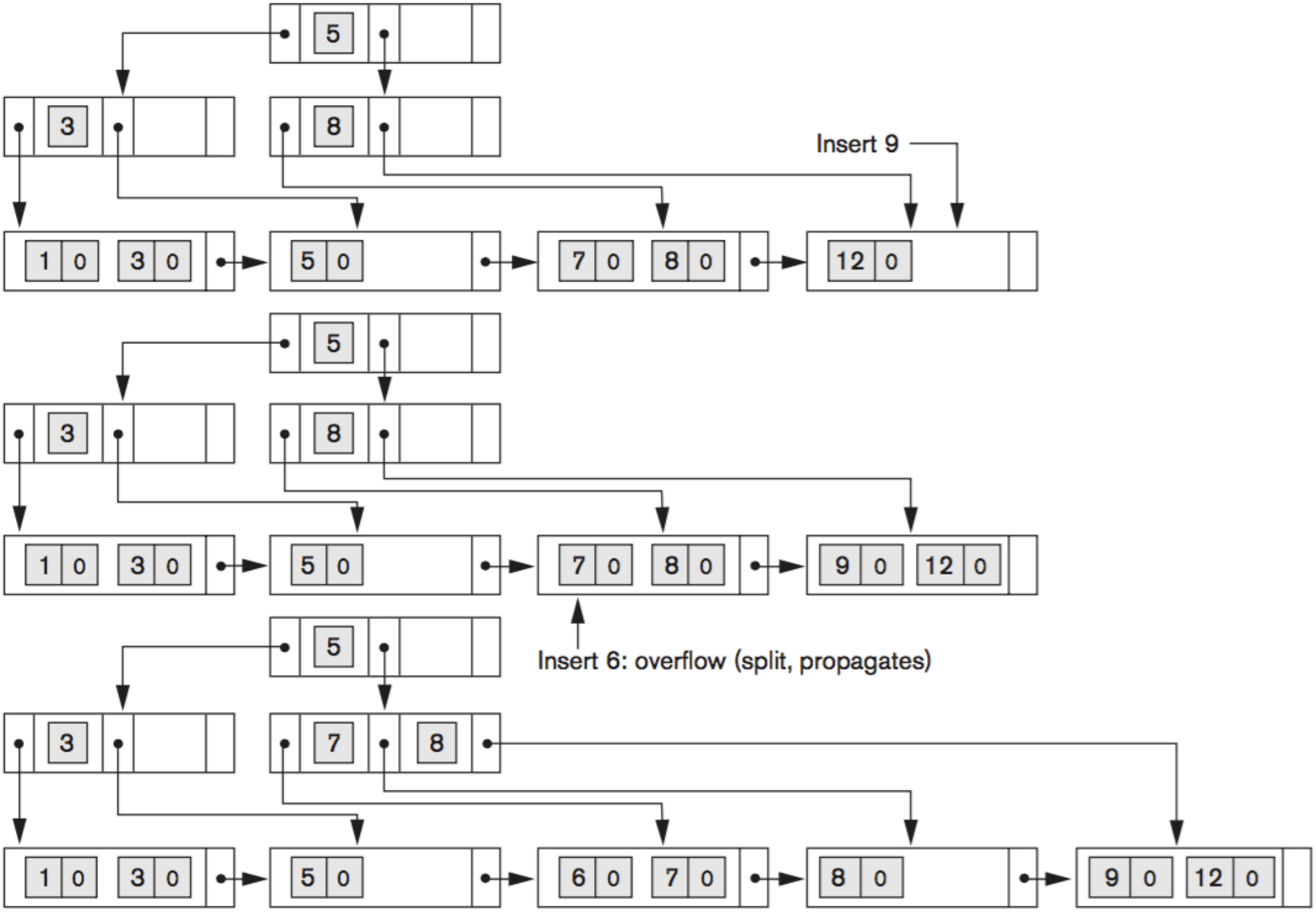
The Nodes of a B+-tree

Figure 18.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values. (b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

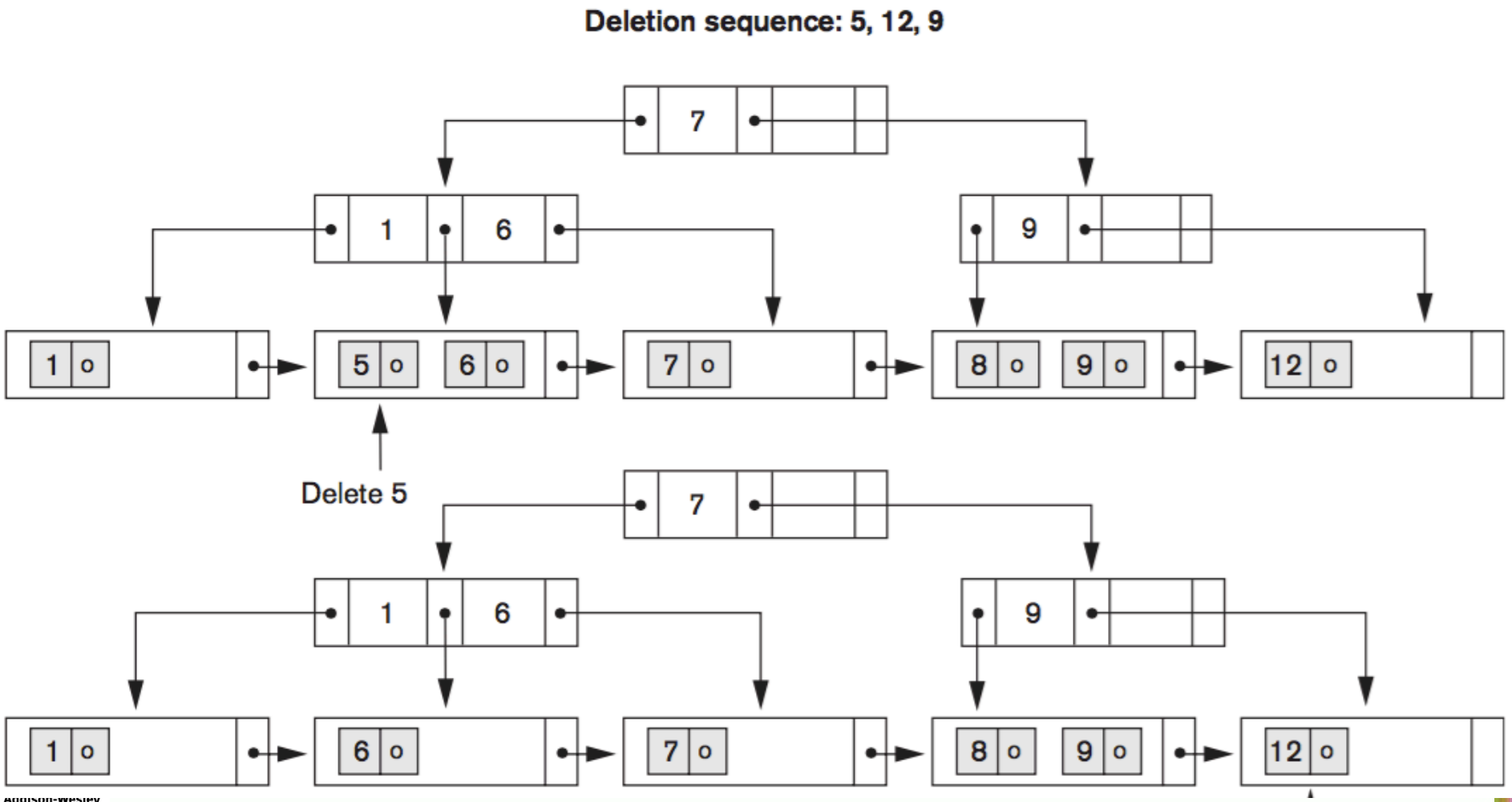


Example of an Insertion in a B+-tree

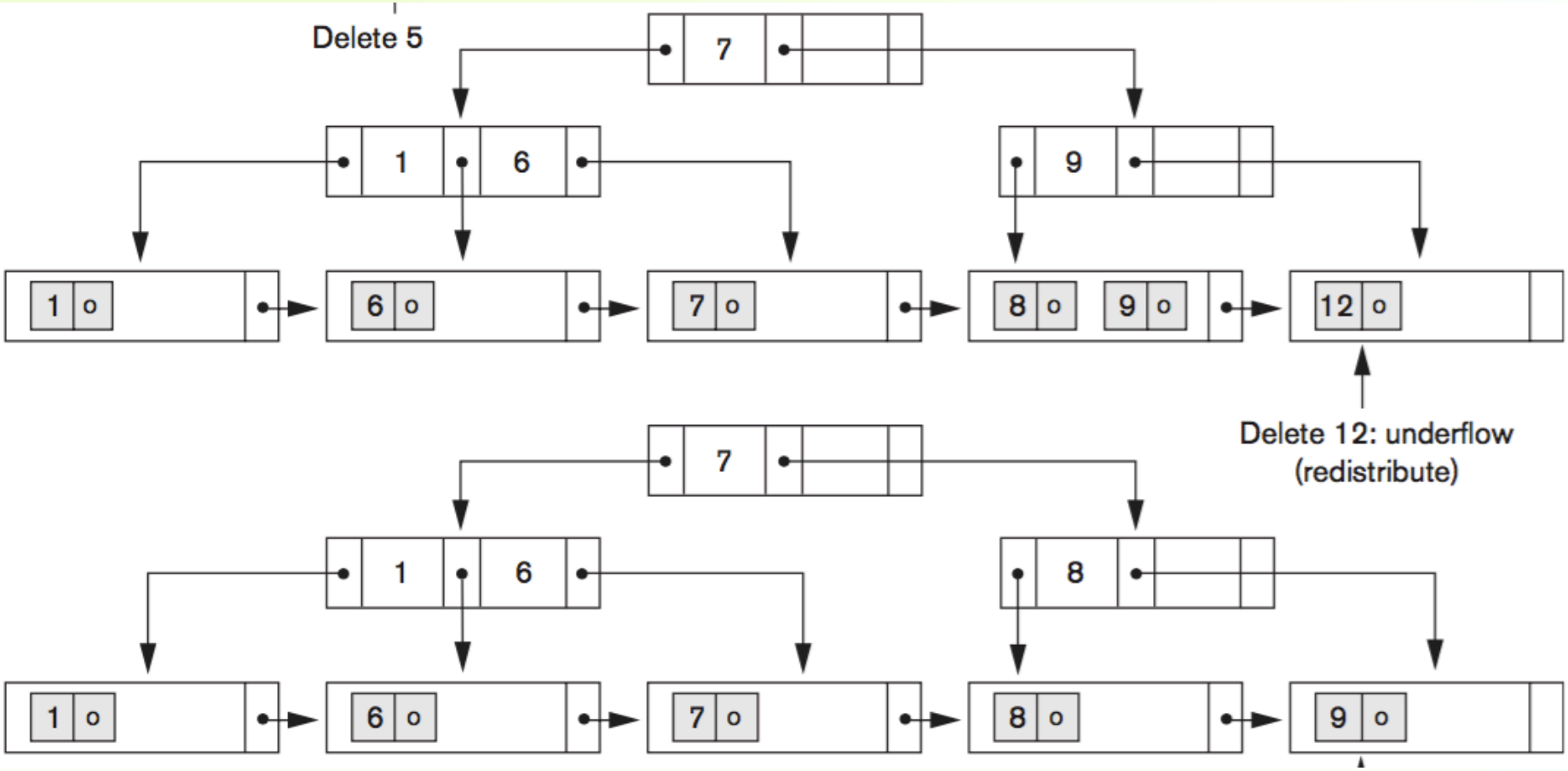




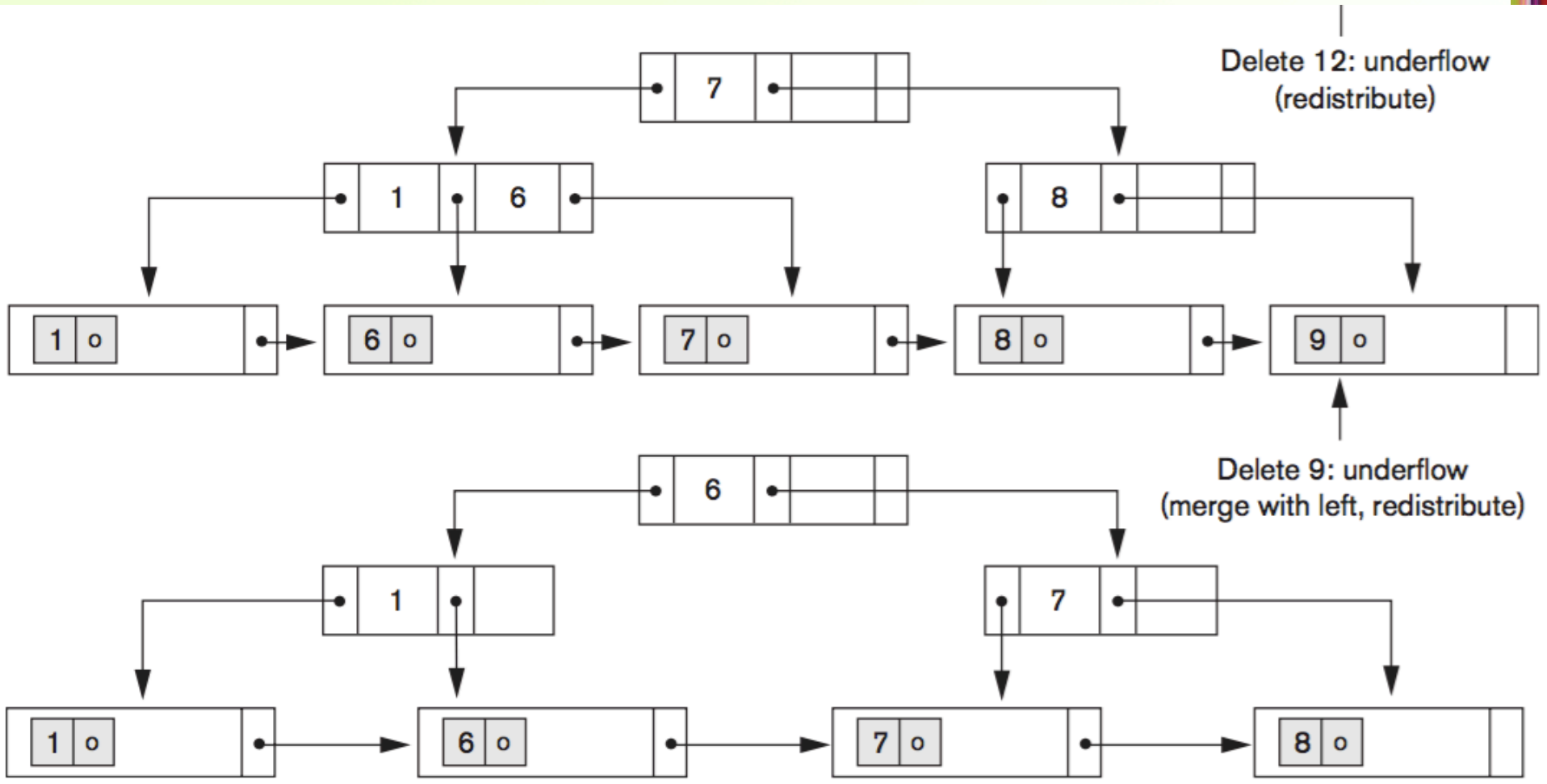
Example of a Deletion in a B+-tree



Example of a Deletion in a B+-tree



Example of a Deletion in a B+-tree



Other Indexing/Storage Structures

- Multi-column indexing
  - Combines several fields into one key by appending one column to another and using the result as an index
  - General way of querying several columns at once
  - Order matters
- Inverted index
  - Useful for word searches in a collection of documents (Full-text search)
- Bitmap index
  - Facilitates querying on multiple keys
- Column oriented storage
  - Particularly useful for data warehouses
  - Instead of storing all column values in same file, split them into files for each column
  - Avoids loading attributes that won't be used in query

Bitmap index

- Typically used with columns containing a small number of distinct possible values
- Built on one particular value of a particular field
- Binary vector representation of all records. Use 1 to indicate a record contains the value, and 0 otherwise

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap Index

- For Sex attribute
  - Sex='M'
    - 10100110
  - Sex='F'
    - 01011001
- For Zipcode attribute
  - Zipcode=94040
    - 10000001
  - Zipcode=30022
    - 01010010
  - Zipcode=19046
    - 00101100
- SELECT ROW_id FROM EMPLOYEE WHERE Sex='M' AND Zipcode=30022
  - 10100110 AND 01010010 = 00000010
  - Result = 6

Inverted Index

- List of all the unique words that appear in a collection of documents
  - For each word, store a list of the documents in which it appears

- Example:
  - The quick brown fox jumped over the lazy dog
  - Quick brown foxes leap over lazy dogs in summer

Design Decisions about Indexing

- **Whether to index an attribute**. The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition.
- **What attribute or attributes to index** on: If multiple attributes from one relation are involved together in several queries, a multiattribute (composite) index is warranted.
- **Whether to set up a clustered index**. At most, one index per table can be a primary or clustering index: If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering.
- **Whether to use a hash index over a tree index**. In general, RDBMSs use B+-trees for indexing. However, ISAM and hash indexes are also provided in some systems. B+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.

Tuning Indexes

- The initial choice of indexes may have to be revised for the following reasons:
  - Certain queries may take too long to run for lack of an index.
  - Certain indexes may not get utilized at all.
  - Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.
- The goal of tuning is to dynamically evaluate the requirements, which sometimes fluctuate seasonally or during different times of the month or week
- Reorganize the indexes and file organizations to yield the best overall performance
- Dropping and building new indexes produces overhead
  - Needs to be justified with boost to performance
- Updating of a table is generally suspended while an index is dropped or created
  - Loss of service must be accounted for
- **Rebuilding the index periodically** may improve performance
  - Most RDBMSs use B+-trees for an index.
    - If there are many deletions on the index key, index pages may contain wasted space
    - Too many insertions may cause overflows in a clustered index that affect performance

Indexes in MySQL

- Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; MEMORY tables also support hash indexes; InnoDB uses inverted lists for FULLTEXT indexes.

Summary

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys