Data Storage, Indexing

Dr. Jeevani Goonetillake

File Organization and Storage Structures

Primary Storage (Main Memory)

- Fast
- Volatile
- Expensive

Secondary Storage (Files in disks or tapes)

- Non-Volatile

Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A *disk pack* contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular *tracks* on each disk *surface*. Track capacities vary typically from 4 to 50 Kbytes.

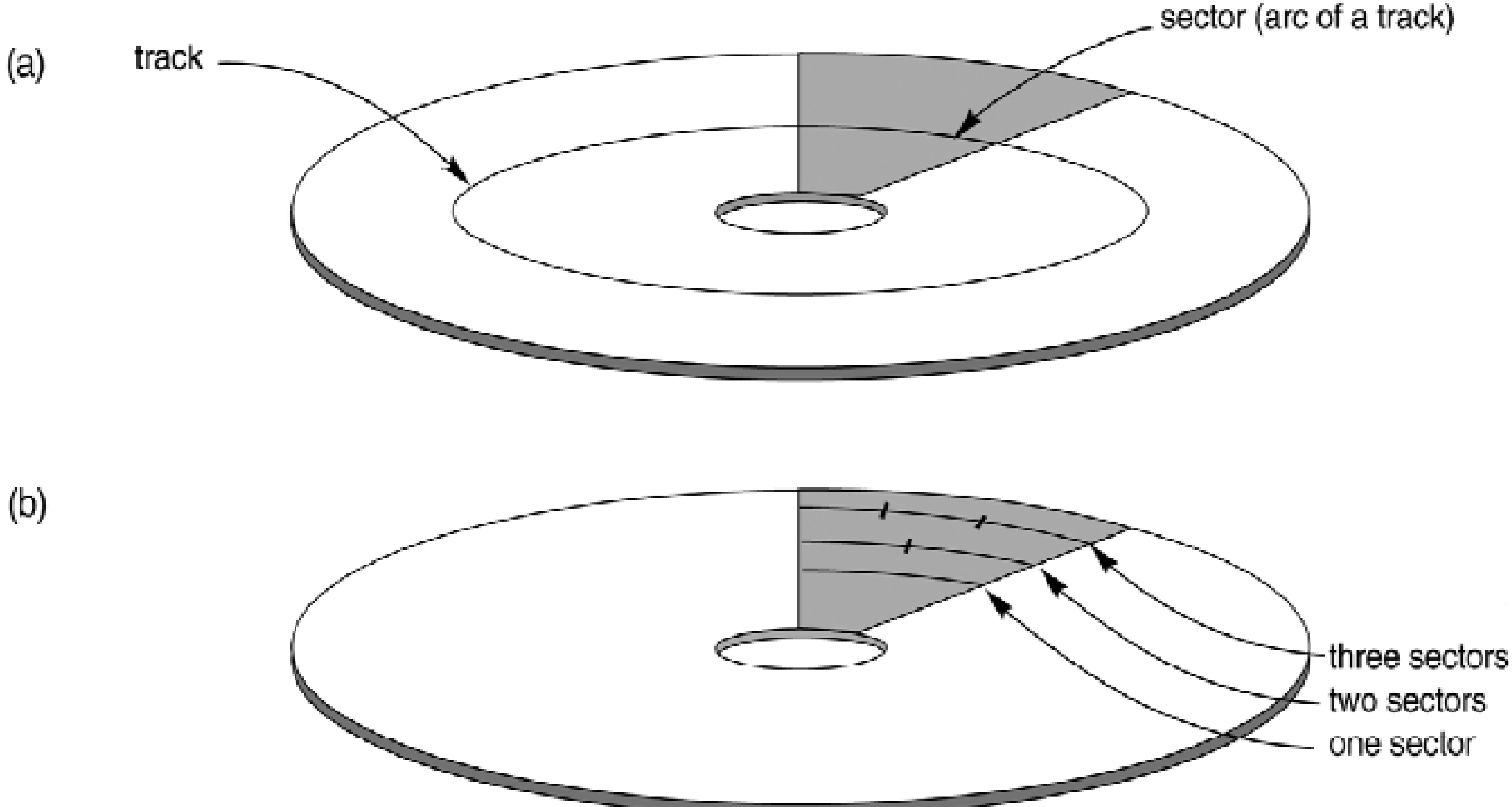
Disk Storage Devices

- Since a track usually contains a large amount of information, it is divided into smaller *blocks* or *sectors*.

- The block size B is fixed for each system.

- Typical block sizes range from B=512 bytes to B=4096 bytes. Whole blocks are transferred between disk and main memory for processing.

Disk Storage Devices



Disk Storage Devices

- A *read-write* head moves to the track that contains the block to be transferred.
- Disk rotation moves the block under the readwrite head for reading or writing.
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.

Blocking

- Blocking: refers to storing a number of records in one block on the disk.
- Blocking factor (*bfr*) refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.

Files of Records

- A file is a *sequence* of records, where each record is a collection of data values (or data items).

- A *file descriptor* (or *file header*) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.

- Records are stored on disk blocks. The *blocking factor bfr* for a file is the (average) number of file records stored in a disk block.

Operation on Files

- **OPEN:** Readies the file for access, and associates a pointer that will refer to a *current* file record at each point in time.
- **FIND:** Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT:** Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ:** Reads the current file record into a program variable.
- **INSERT:** Inserts a new record into the file, and makes it the current file record.

Operation on Files

- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY:** Changes the values of some fields of the current file record.
- **CLOSE:** Terminates access to the file.
- **REORGANIZE:** Reorganizes the file records. For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ_ORDERED:** Read the file blocks in order of a specific field of the file.

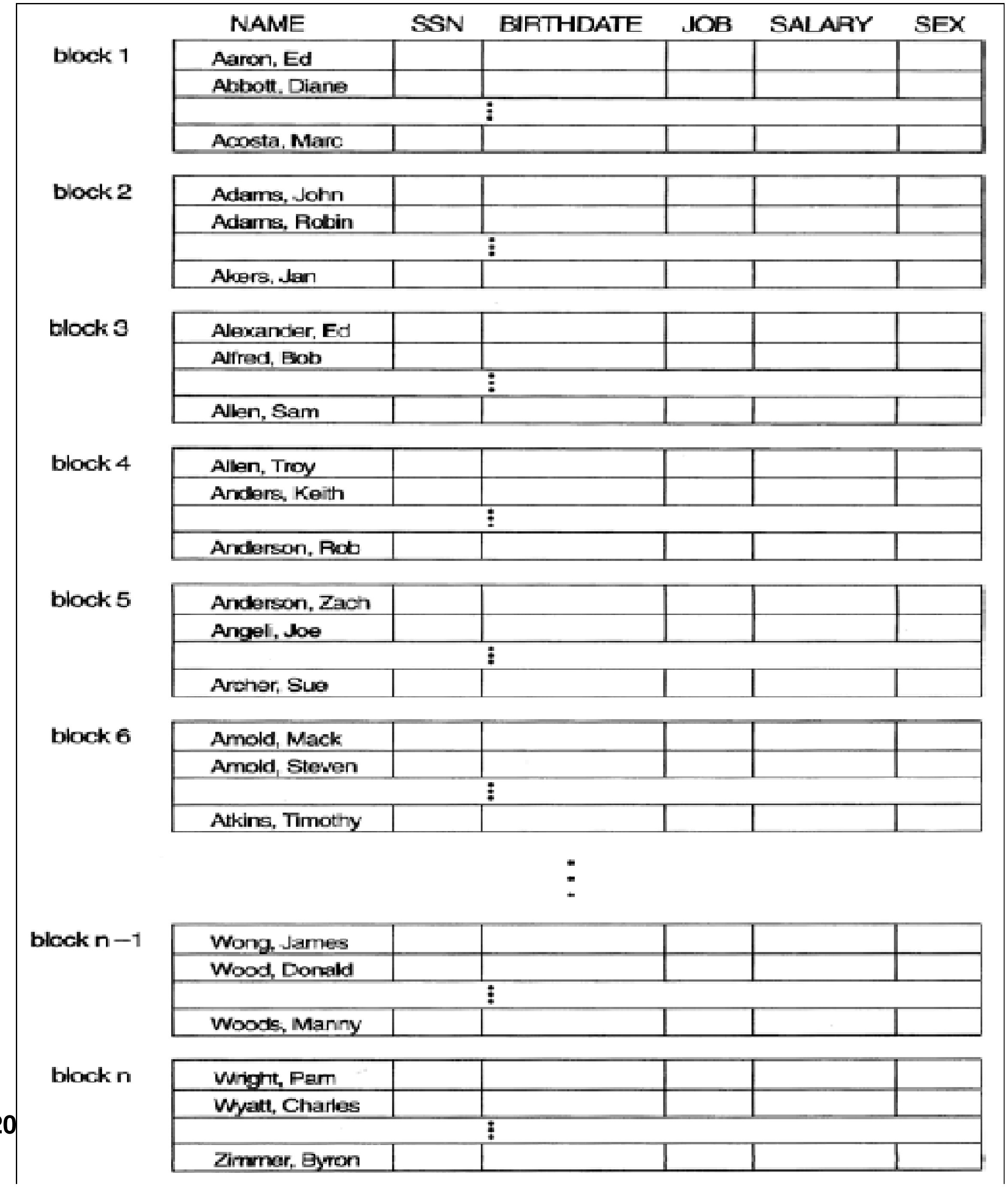
Unordered Files

- Also called a *heap* or a *pile* file.
- New records are inserted at the end of the file.
- To search for a record, a *linear search* through the file records is necessary. This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- To delete a record, the record is marked as deleted. Space is reclaimed during periodical reoganization.

Ordered Files

- Also called a *sequential file*.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the *correct order*.
- A *binary search* can be used to search for a record on its *ordering field value*. This requires reading and searching log2 of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Ordered Files



Average Access Times

The following table shows the average access time to access a specific record for a given type of file

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

Hashed Files

- The file blocks are divided into M equal-sized *buckets*, numbered bucket0, bucket1, ..., bucket M-1.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the *hashing function*.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full. An overflow file is kept for storing such records.

Hashed Files

- There are numerous methods for collision resolution, including the following:

Open addressing: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.

Chaining: A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.

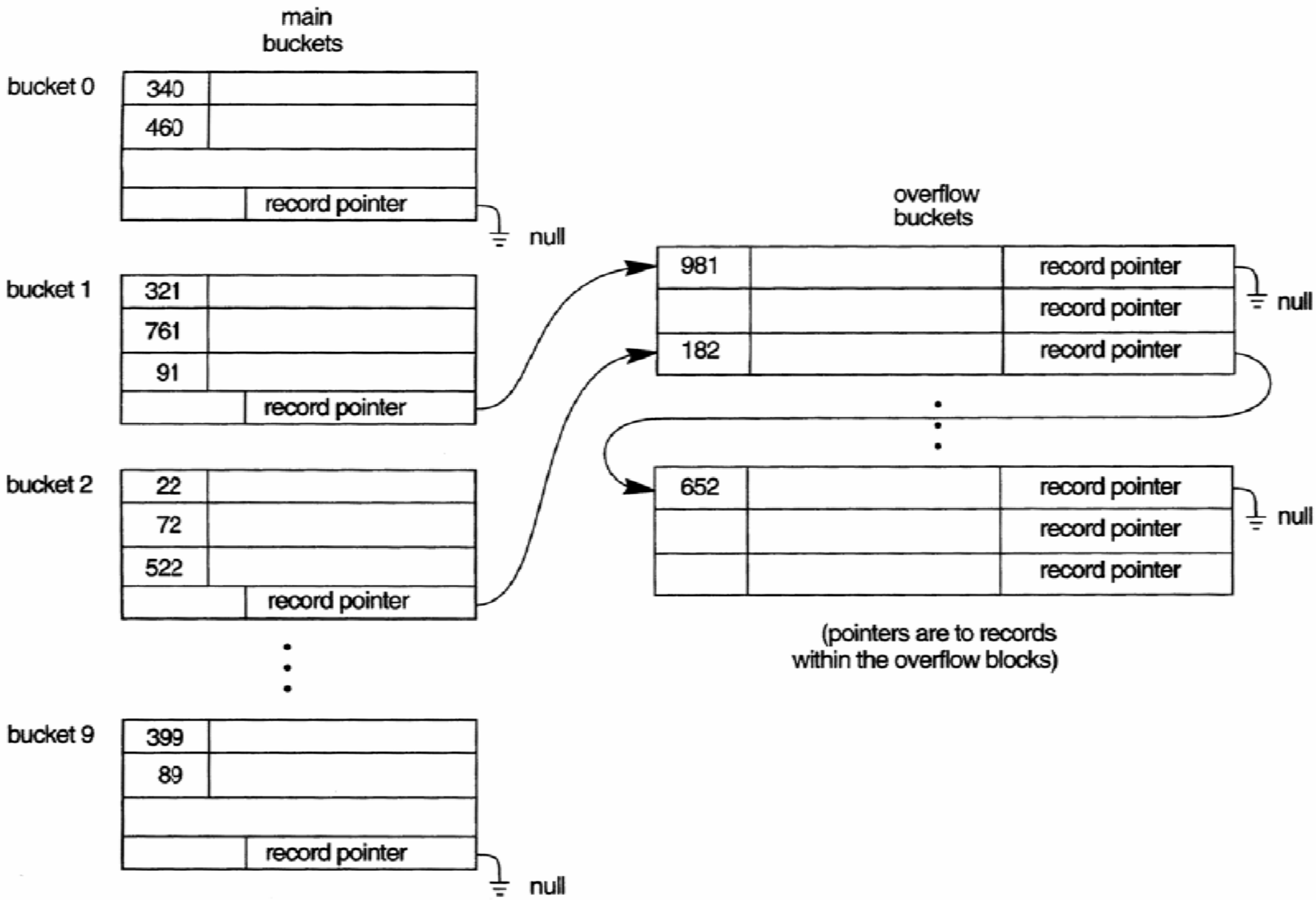
Multiple hashing: The program applies a second hash function if the first results in a collision.

Hashed Files

- The hash function h should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of *static* hashing:

  Fixed number of buckets M is a problem if the number of records in the file grows or shrinks.

Hashed Files



Hashed Files Limitation

- **Inappropriate for some retrievals:**
  based on pattern matching
  eg. Find all students with ID like 98xxxxxx.
- Involving ranges of values
  eg. Find all students from 50100000 to 50199999.
- Based on a field other than
  the hash field

Student ID	Tutorial	Grade
50195255	T01	A
50194525	T02	A
98076543	T01	A+

Indexes

- Index: A data structure that allows particular records in a file to be located more quickly

  ~ Index in a book

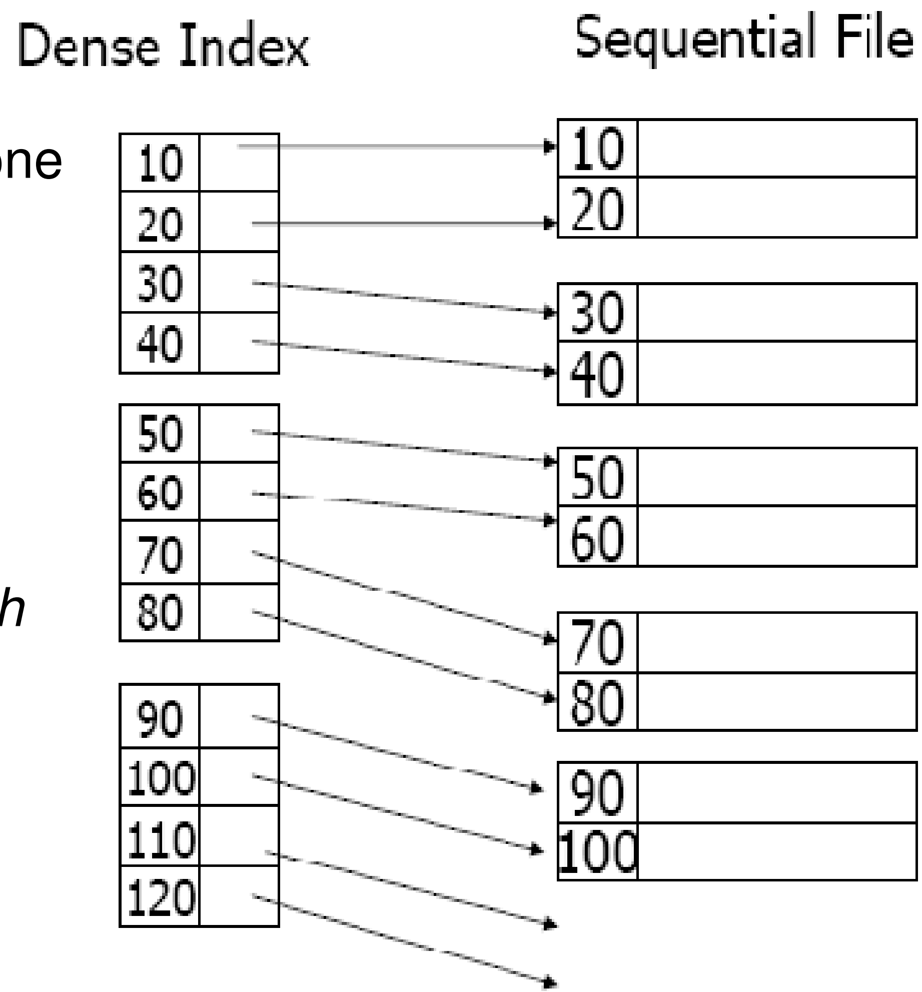
- An index can be sparse or dense:
  - Sparse: record for only some of the search key values (eg. Staff Ids: CS001, EE001, MA001). Applicable to ordered data files only.
  - Dense: record for every search key value. (eg. Staff Ids: CS001, CS002, .. CS089, EE001, EE002, ..)

Indexes

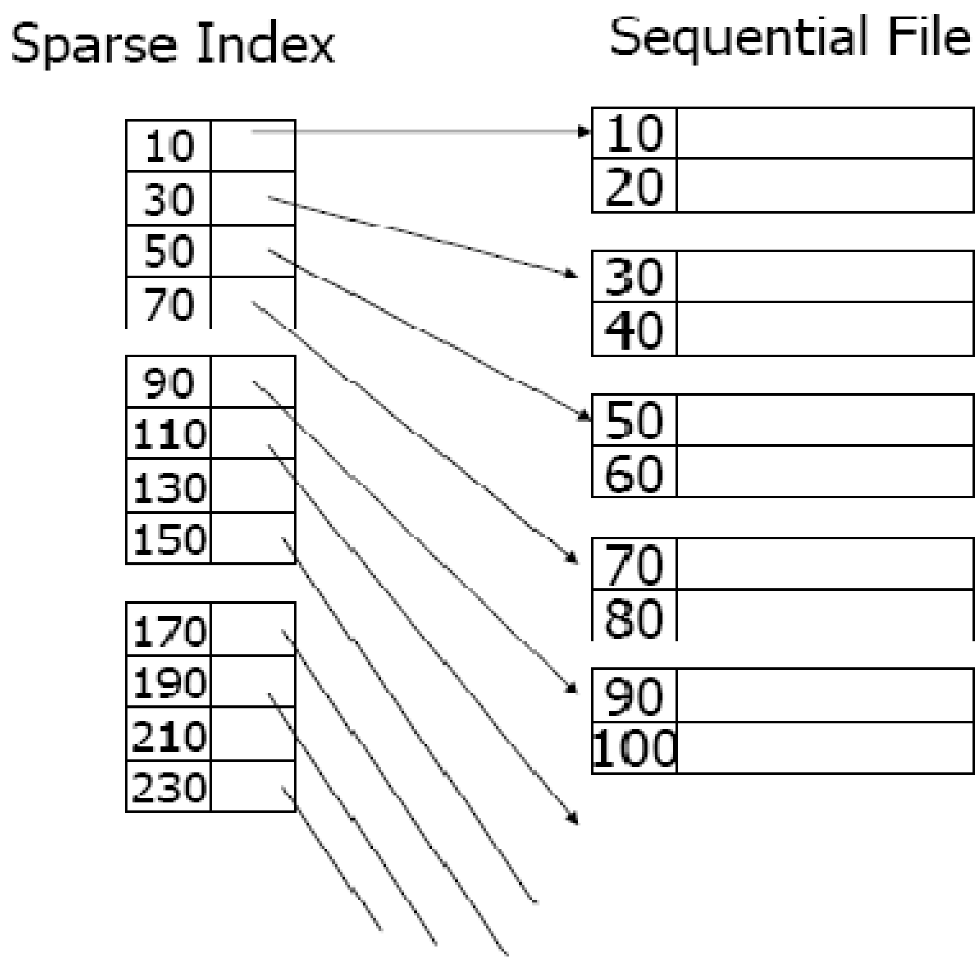
- **Data file**: a file containing the logical records
- **Index file**: a file containing the index records
- **Indexing field**: the field used to order the index records in the index file

Dense Index

–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.



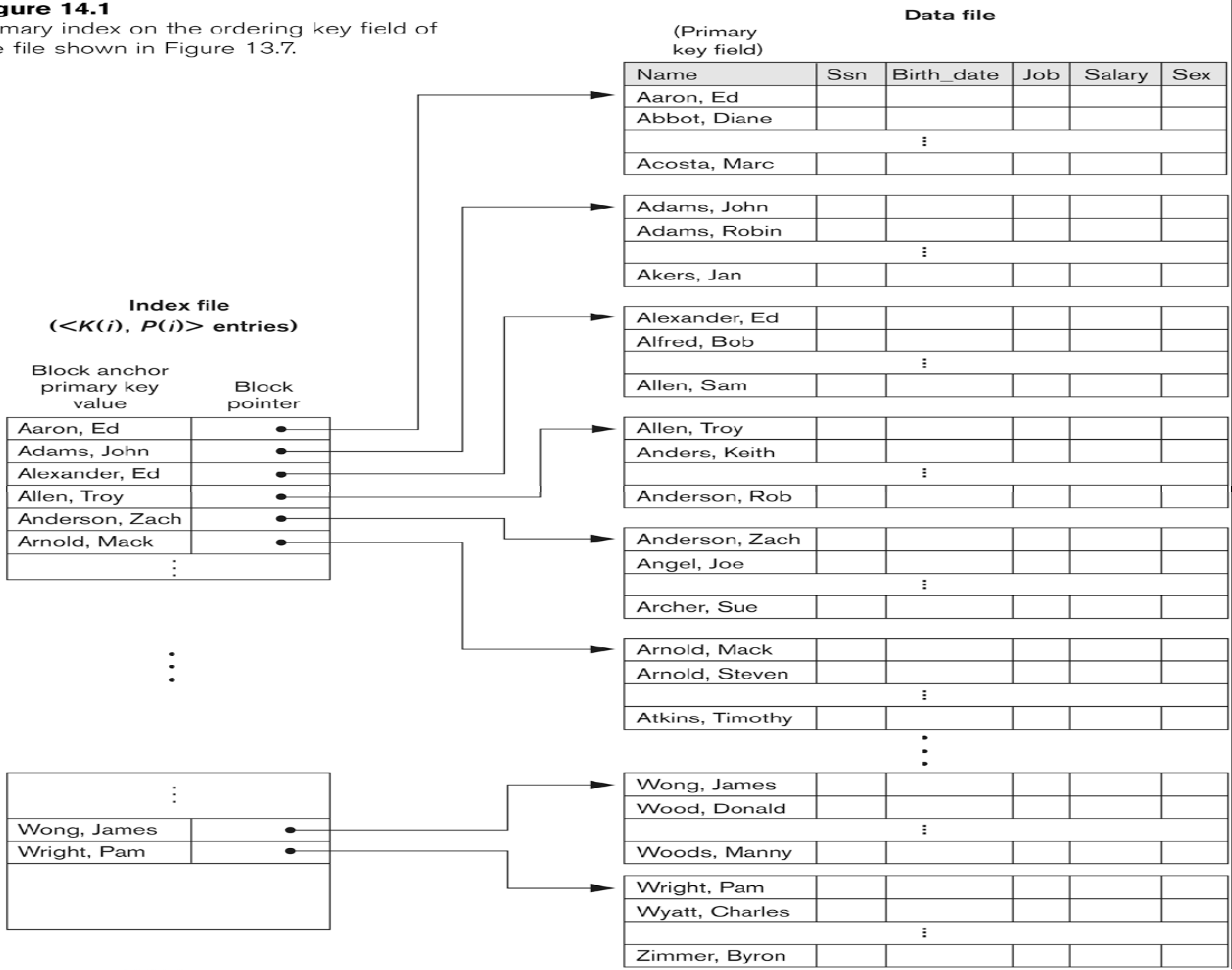
Sparse Index



Primary Index

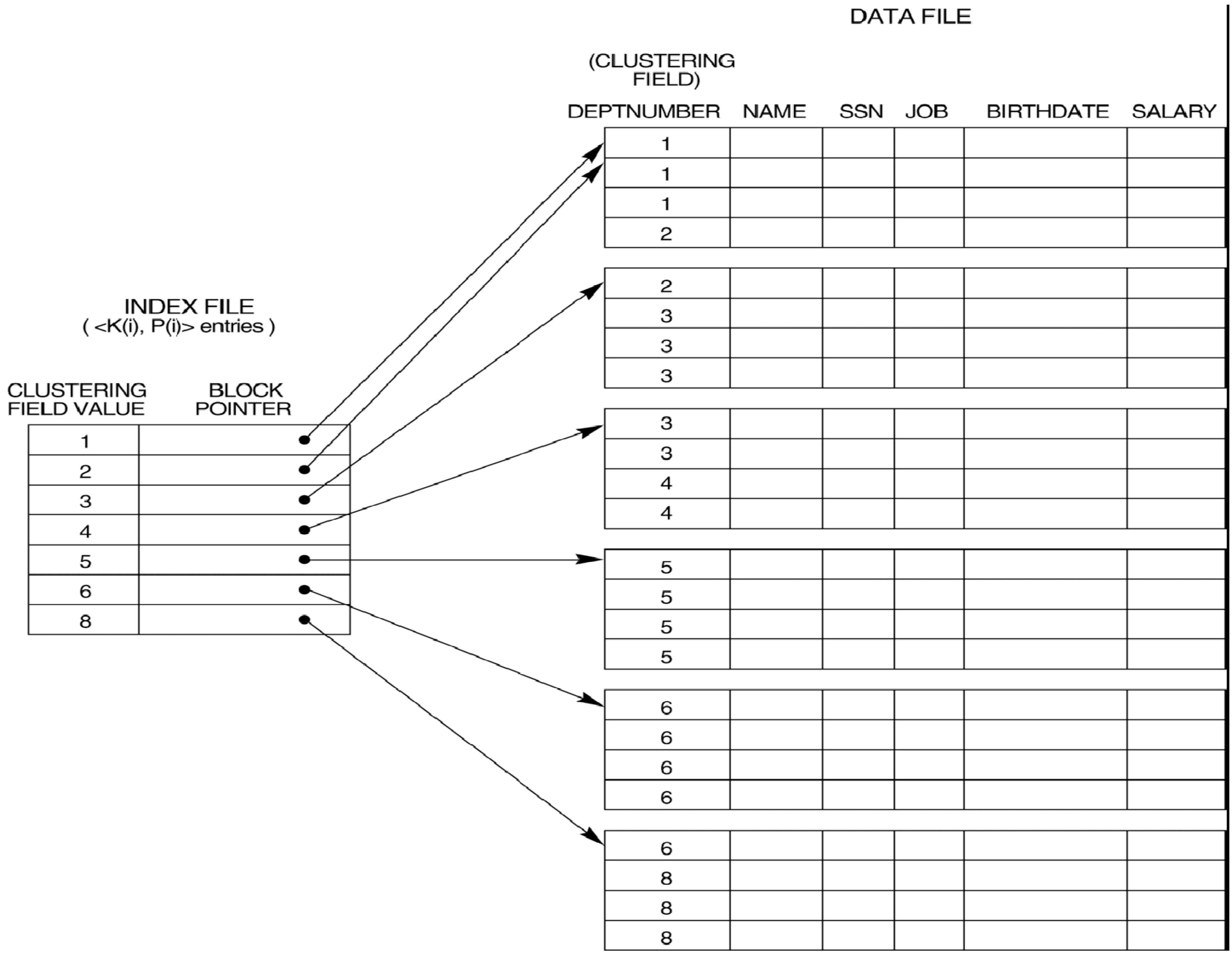
- Defined on an ordered data file.
- The data file is ordered on a **key field.**
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor.*
- A primary index is a nondense (sparse) index, 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.

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.



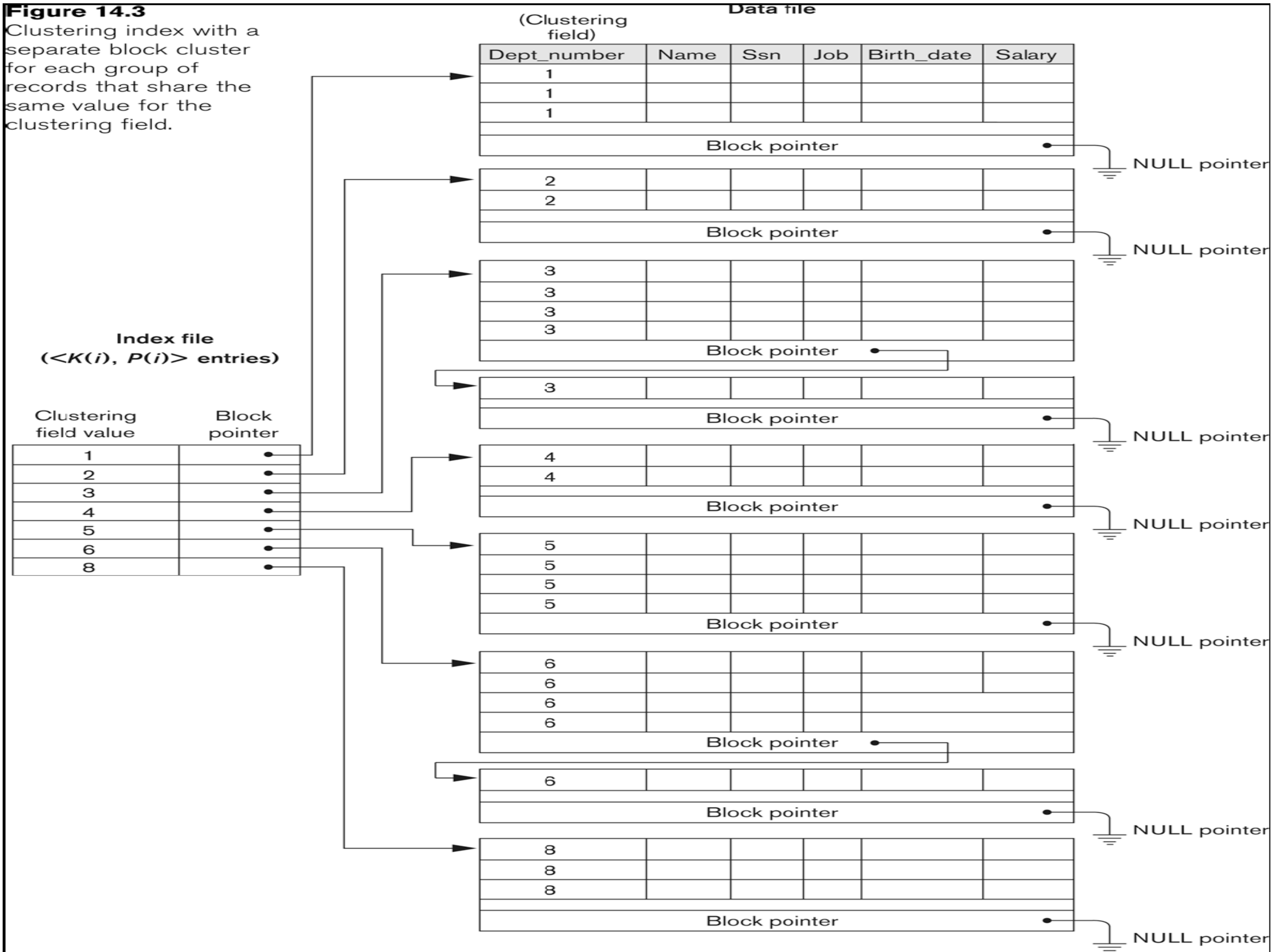
Clustering Index

- 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.



DATA FILE
(CLUSTERING FIELD)
DEPTNUMBER
NAME
SSN
JOB
BIRTHDATE
SALARY
INDEX FILE
(<K(i), P(i)> entries)
CLUSTERING FIELD VALUE
BLOCK POINTER
1
2
3
4
5
6
8

Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Secondary Index

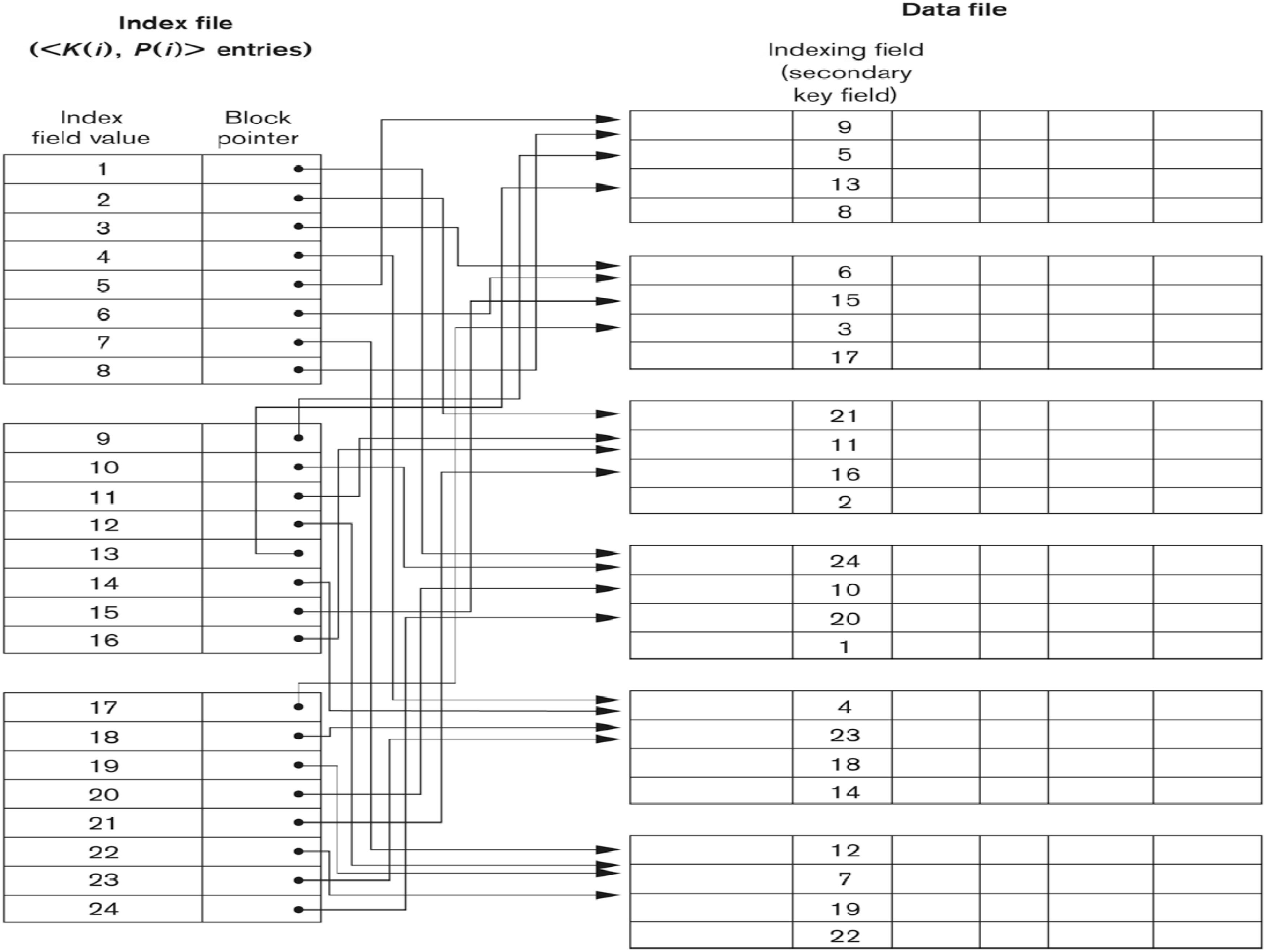
- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- The index is an ordered file with two fields.
- 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.

Secondary Index

- 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.*

Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



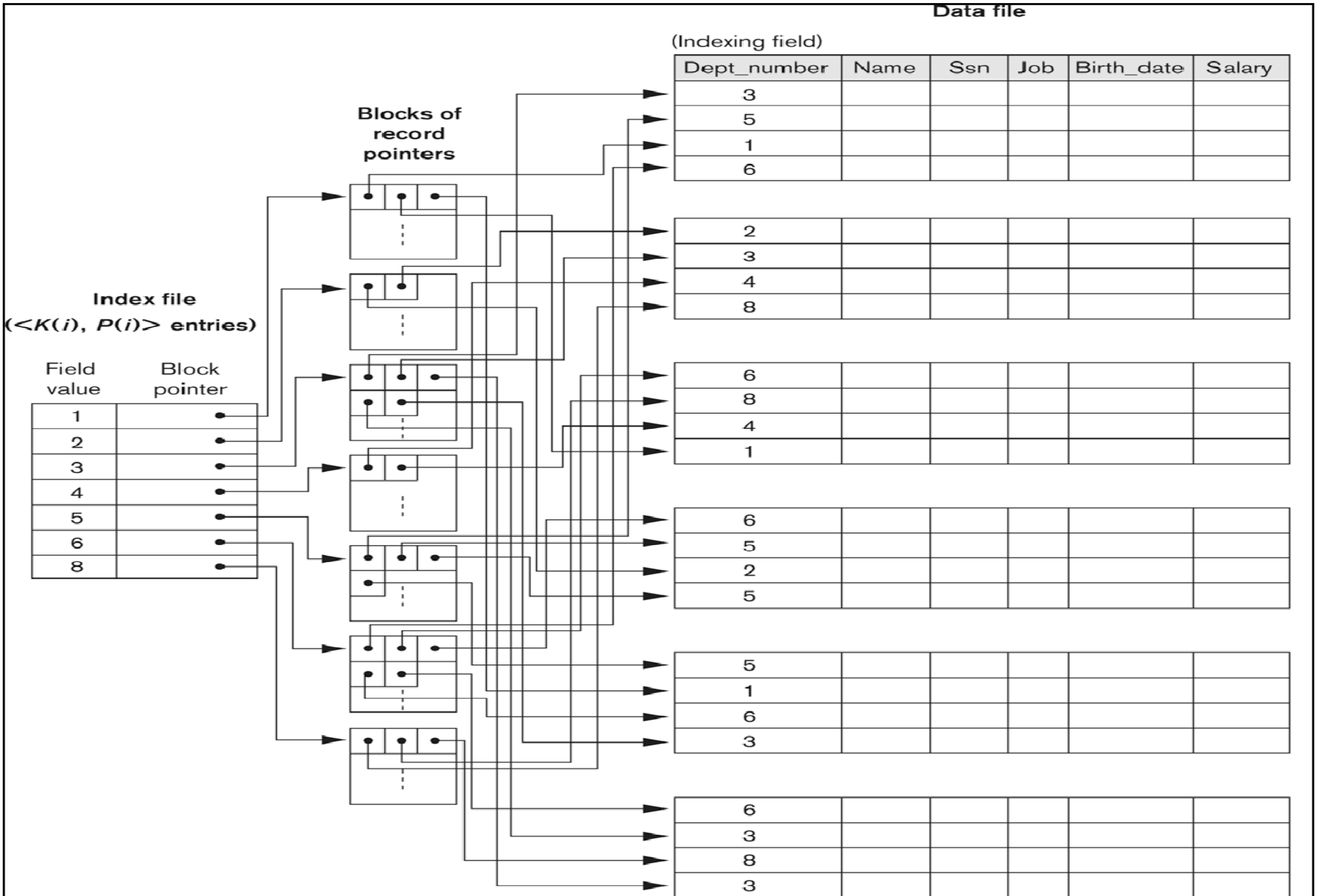


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	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

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

[c]For options 2 and 3.

Multi-Level Indexes

- Since 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.

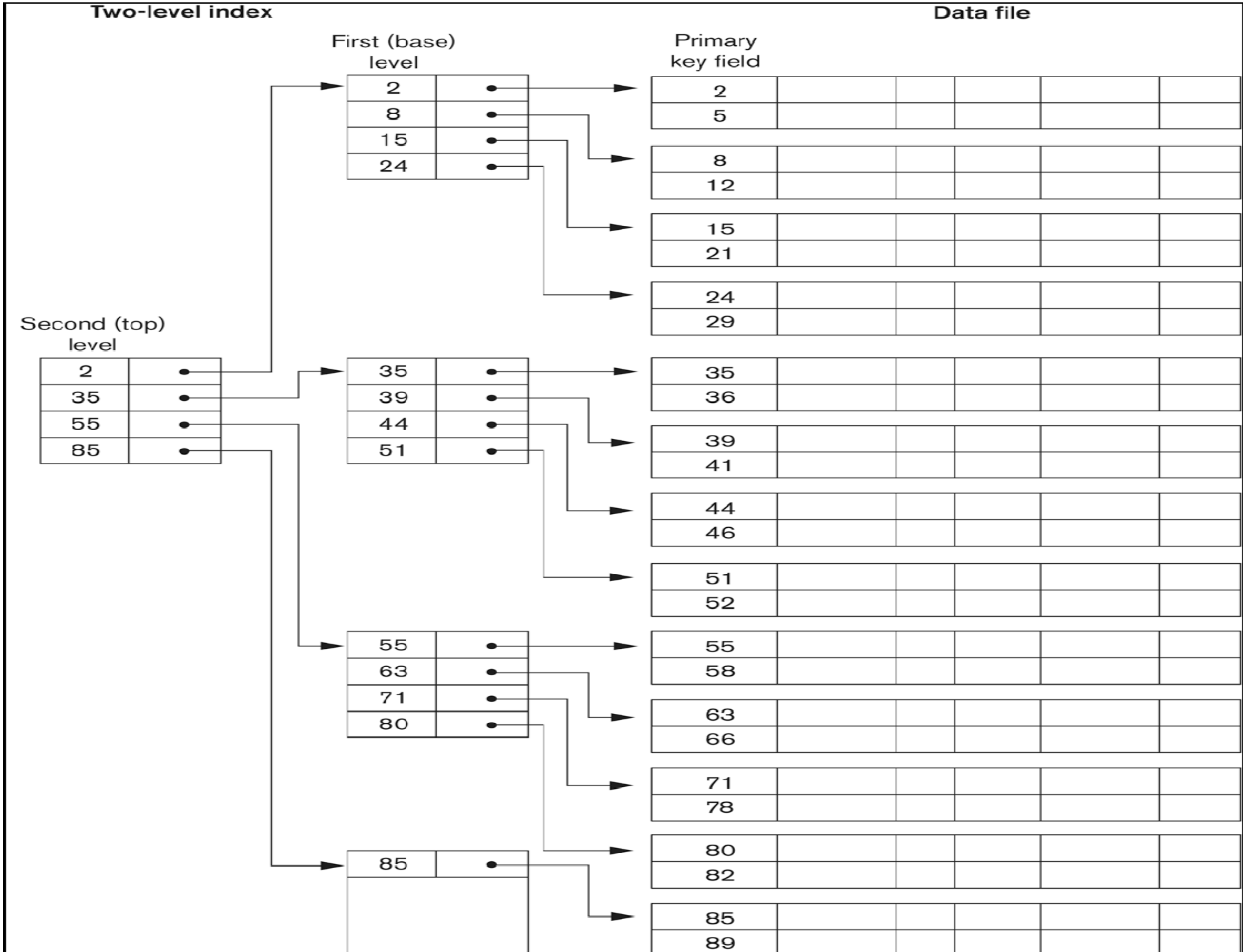


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

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*.

Dynamic Multilevel Indexes Using B+-Trees

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

Dynamic Multilevel Indexes Using B+-Trees

- 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

Dynamic Multilevel Indexes Using B+-Trees

- 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+ tree

The structure of the ***internal nodes*** of a B+ tree of order p is as follows:

- Each internal node is of the form

$<P_1, K_1, P_2, K_2 \ldots\ldots, K_{q-1}, P_{q-1}, P_q>$

where $q \le p$. Each P_i is a tree pointer.

- Within each node $K_1 < K_2 < \ldots . < K_{q-1}$
- Each node has at most p tree pointers.
- Each node with q tree pointers, $q \le p$, has q-1 search key field values.

B+ tree

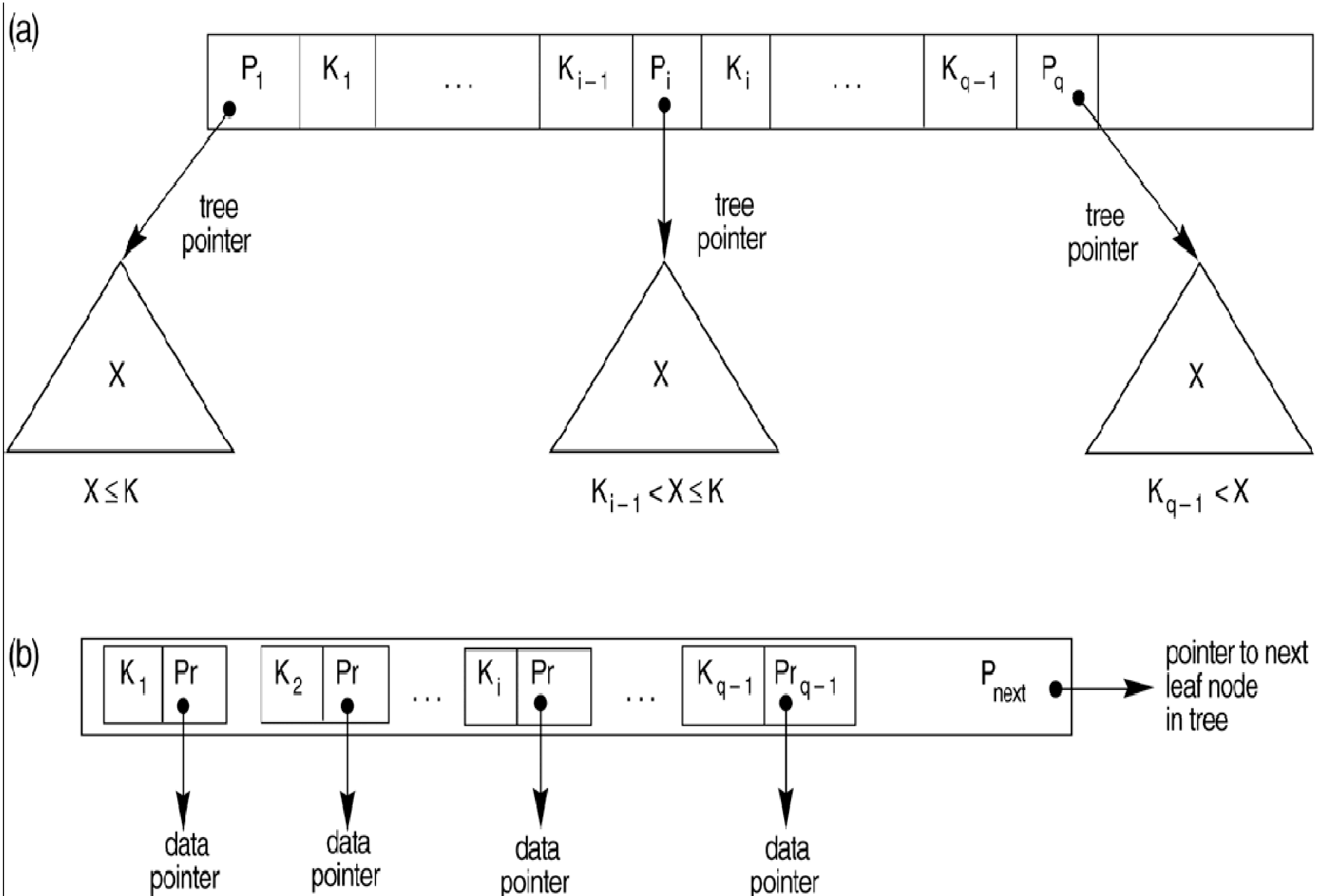
The structure of the ***leaf nodes*** of a B+ tree of order p is as follows:

- Each leaf node is of the form

$<K_1,Pr_1>,<K_2,Pr_2>,\ldots\ldots,<K_{q-1},Pr_{q-1}>,P_{next}>$

where $q \leq p$. Each Pr_i is a data pointer. P_{next} points to the next leaf node of the B+ tree.

- Within each node $K_1 < K_2 < \ldots.<K_{q-1}$
- All leaf nodes are at the same level.



(a)
P1
K1
. . .
Ki–1
Pi
Ki
. . .
Kq–1
Pq
tree pointer
tree pointer
tree pointer
X
X
X
X ≤ K
Ki–1 < X ≤ K
Kq–1 < X
(b)
K1
Pr
K2
Pr
. . .
Ki
Pr
. . .
Kq–1
Prq–1
Pnext
pointer to next leaf node in tree
data pointer
data pointer
data pointer
data pointer

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.

Figure 14.12
An example of insertion in a B+-tree with $p = 3$ and $p_{leaf} = 2$.