Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 2: Indexing Structures for Files

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

- Overall Introduction to Database Management Systems
- Chapter 1. Disk Storage and Basic File Structures
- **Chapter 2. Indexing Structures for Files**
- Chapter 3. Algorithms for Query Processing and Optimization
- Chapter 4. Introduction to Transaction Processing Concepts and Theory
- Chapter 5. Concurrency Control Techniques
- Chapter 6. Database Recovery Techniques

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

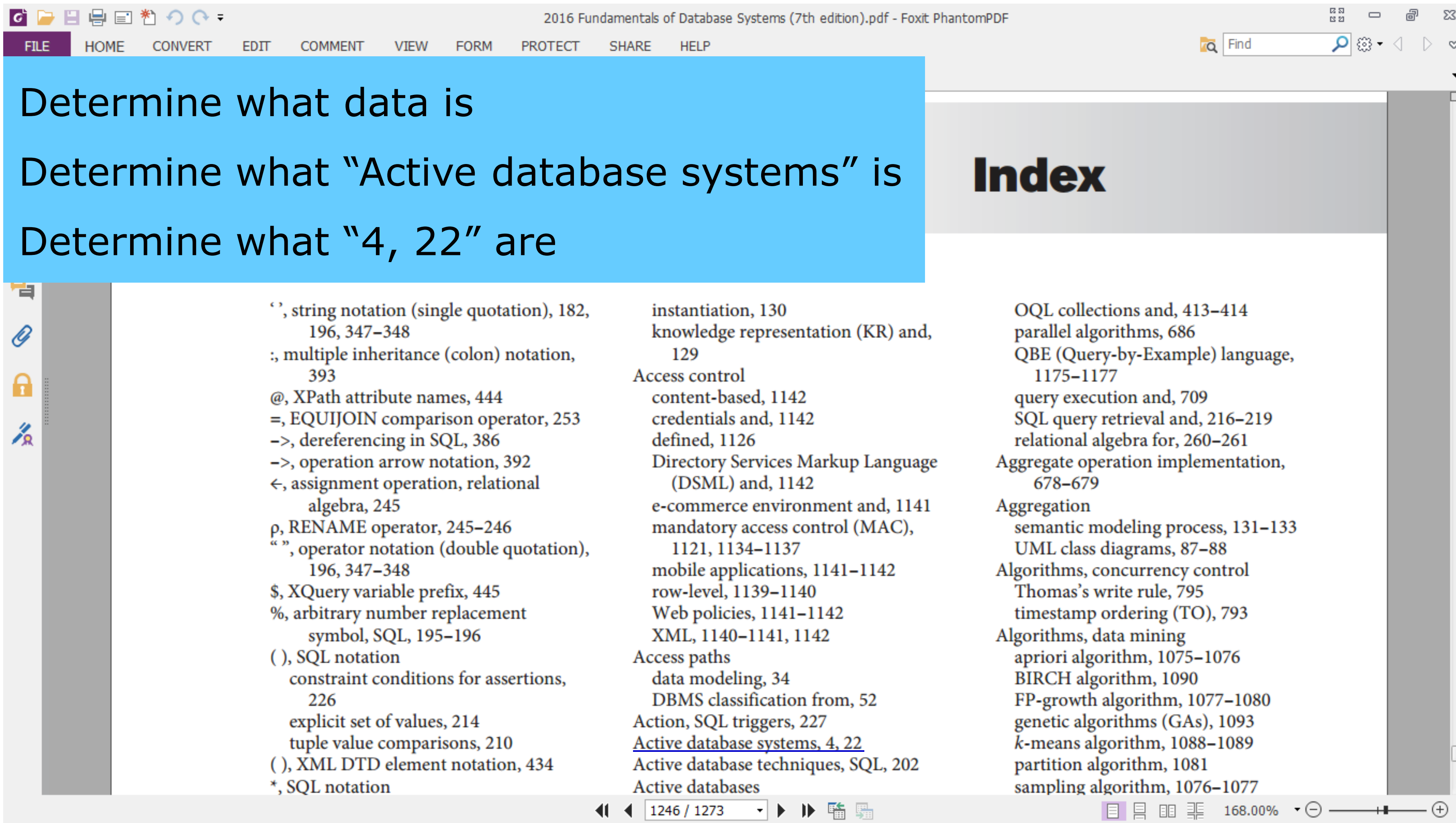
- 2.1. Types of Single-level Ordered Indexes
- 2.2. Multilevel Indexes
- 2.3. Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- 2.4. Indexes on Multiple Keys
- 2.5. Other File Indexes

Indexing …

Determine what data is

Determine what “Active database systems” is

Determine what “4, 22” are



Index

The index section in [1]

Have you ever used this section in any book?

Indexing …

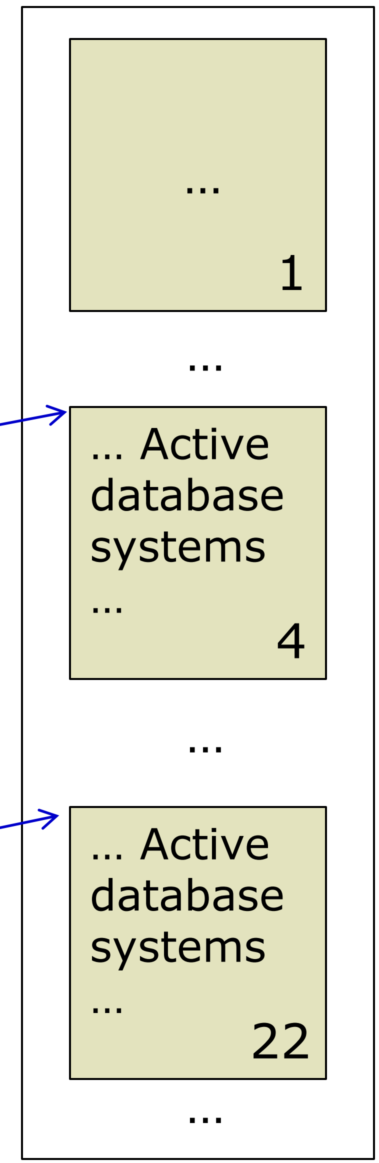
Index

Value	Page No.
‘’, string notation, …	182, 196, 347-348
…	…
Action, SQL triggers	227
Active database systems	4, 22
Active database techniques, SQL	202
…	…

Ordered indexed values

Linking values

(addressable)



Book content

Indexing …

- Assumption: a data file exists with some primary organization such as the unordered, ordered, or hashed organization.
- Indexes are *additional* auxiliary access structures of a data file.
  - Role: ***secondary* access paths**, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk
  - Purpose: speed up the retrieval of records in response to *certain search conditions*
  - Management: *additional* ordered files on disk

2.1. Types of Single-level Ordered Indexes

- A single-level ordered index is an access structure defined on a field of a file (or multiple fields of a file).
  - This index is a file including many entries. Each entry is <**Field value**, **Pointer**(s)>.
    - Field values: able to be ordered.
    - Pointer(s): record pointers or block pointers to the data file.
  - The field is called an **indexing field**.
  - The index file is **ordered** with the field values.
  - **Binary search** is applied on the index file with the conditions =, >, <, ≥, ≤, *between* on the indexing field.

2.1. Types of Single-level Ordered Indexes

Single-level ordered index file
(bfr_i = 4 records/block)

Field Value	(Block) Pointer
2	•
10	•
18	•
23	•
35	•

Data file of employee records
(bfr = 3 records/block)

SSN	Name	...	Dept
2	F	...	2
5	AA	...	5
8	U	...	4
10	C	...	2
11	L	...	2
15	EG	...	1
18	B	...	4
19	FL	...	5
21	N	...	5
23	Y	...	5
28	D	...	1
34	O	...	2
35	NM	...	3

SSN: indexing field

2.1. Types of Single-level Ordered Indexes

- The index file usually occupies considerably less disk blocks than the data file because the number of the entries in the index file is much smaller.
- A binary search on the index file yields a pointer to the file record.
- Indexes are characterized as dense or sparse.
  - A **dense index** has an index entry for *every field value* (and hence every record) in the data file.
  - A **sparse** (or **non-dense**) **index** has index entries for only some field values.
    - The previous example is a *non-dense* index.

2.1. Types of Single-level Ordered Indexes

- Types of ordered indexes
  - A **primary index** is specified on the *ordering key field* of an **ordered file** of records.
  - A **clustering index** is specified on the *ordering non-key field* of an **ordered file** of records.
  - A **secondary index** is specified on any *non-ordering field* of a file of records.
  - → A file can have *at most one* physical ordering field, so it can have *at most one* primary index or one clustering index, *but not both*.
  - → A data file can have *several* secondary indexes in addition to its primary access method.

2.1. Types of Single-level Ordered Indexes

- Primary indexes
  - An ordered file whose records are of fixed length with two fields:
    - The first field is of the same data type as the ordering key field—called the **primary key**—of the data file.
    - The second field is a pointer to a disk block.
  - There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values: $<K(i), P(i)>$.
  - The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.

Primary Indexes

- Non-dense index
- Block anchor

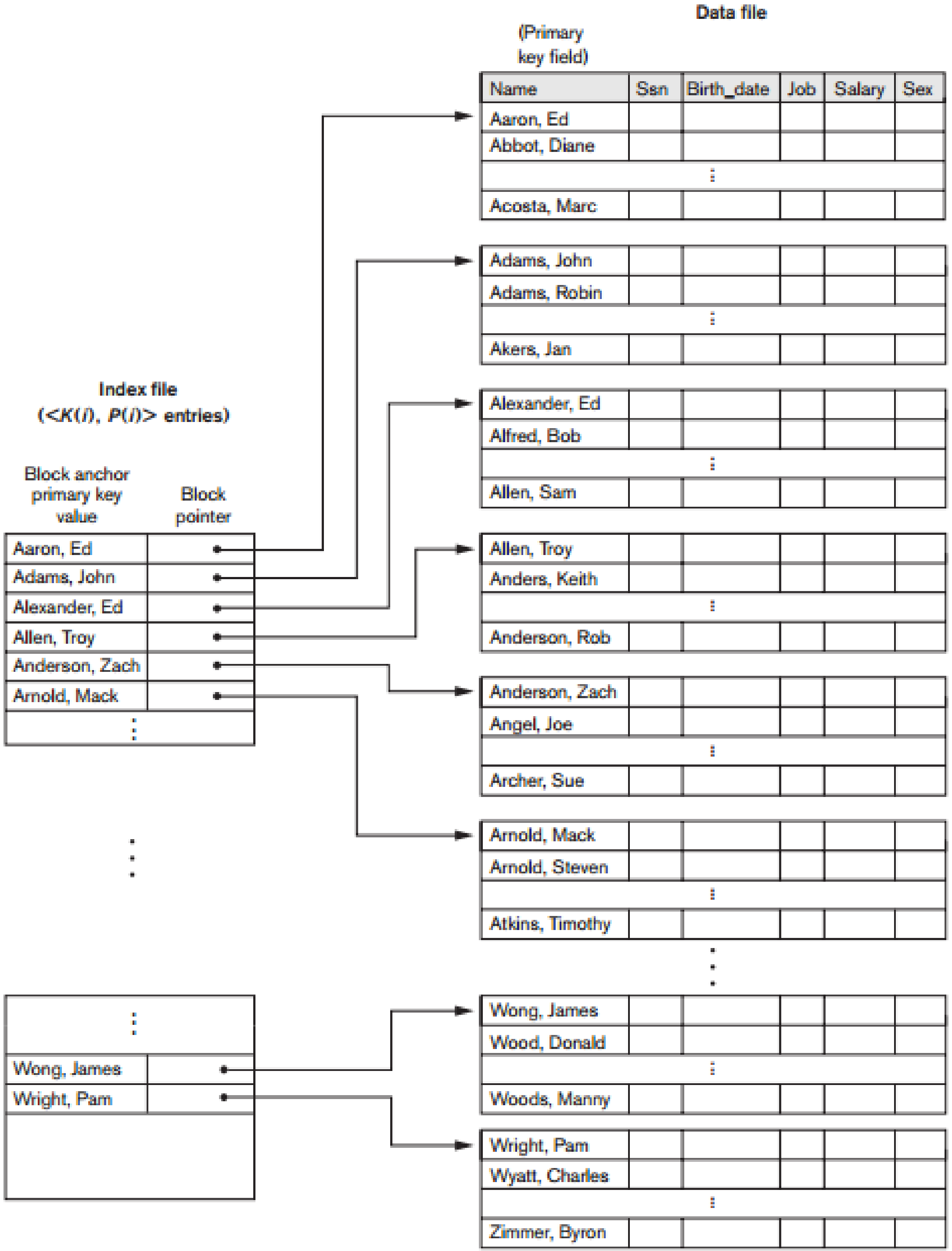


Figure 17.1, pp. 604, [1]

2.1. Types of Single-level Ordered Indexes

- Primary indexes
  - A primary index is a nondense (sparse) index.
    - Why?
  - The index file for a primary index occupies a much smaller space than does the data file.
    - Why?
  - Given the value *K* of its primary key field, a binary search is used on the index file to find the appropriate index entry *i*, and then retrieve the data file block whose address is *P*(*i*).
  - A major problem with a primary index is insertion and deletion of records.
    - Why and how to solve it?

Primary indexes

Example 1: given the following data file with the ordering key field SSN
EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)

record size R=150 bytes
block size B=512 bytes
number of records r=30,000 records

blocking factor bfr= B div R= $\lfloor B/R \rfloor$ = 512 div 150= 3 records/block
number of file blocks b = $\lceil r/bfr \rceil$= $\lceil 30{,}000/3 \rceil$ = 10,000 blocks

For a primary index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the block pointer size P_B=6 bytes.

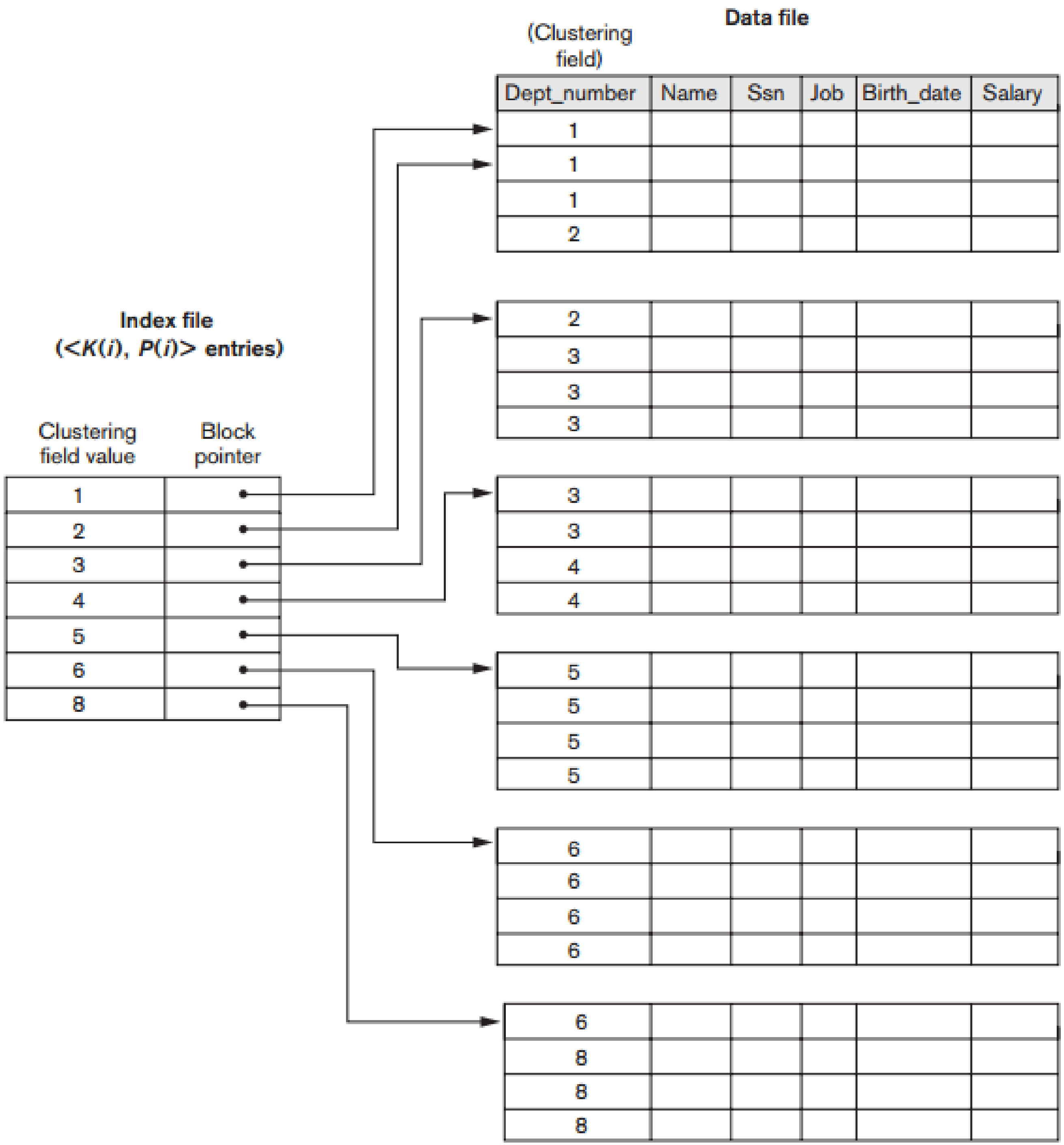
index entry size R_i=V_{SSN}+ P_B=9+6=15 bytes
index blocking factor bfr_i= B div R_i= 512 div 15= 34 entries/block
number of index entries r_i = number of file blocks b = 10,000 entries
number of index blocks b_i= $\lceil r_i/ bfr_i \rceil$ = $\lceil 10{,}000/34 \rceil$ = 295 blocks
binary search on the index needs $\lceil \log_2 b_i \rceil$ = $\lceil \log_2 295 \rceil$ = 9 block accesses
one extra block access to retrieve the record from the data file
The total search cost via the index is: 9 + 1 = 10 block accesses

This is compared to an average linear search cost directly on the data file:
$\lceil b/2 \rceil$ = $\lceil 10{,}000/2 \rceil$ = 5,000 block accesses
Because the file records are ordered, the binary search cost would be:
$\lceil \log_2 b \rceil$ = $\lceil \log_2 10{,}000 \rceil$ = 13 block accesses

2.1. Types of Single-level Ordered Indexes

- Clustering indexes
  - Also defined on an ordered data file with an ordering *non-key* field.
  - Each 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 *non-dense* index.
  - Record insertion and deletion cause problems.
    - Why?
  - To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field.
    - All records with that value are placed in the block (or block cluster).

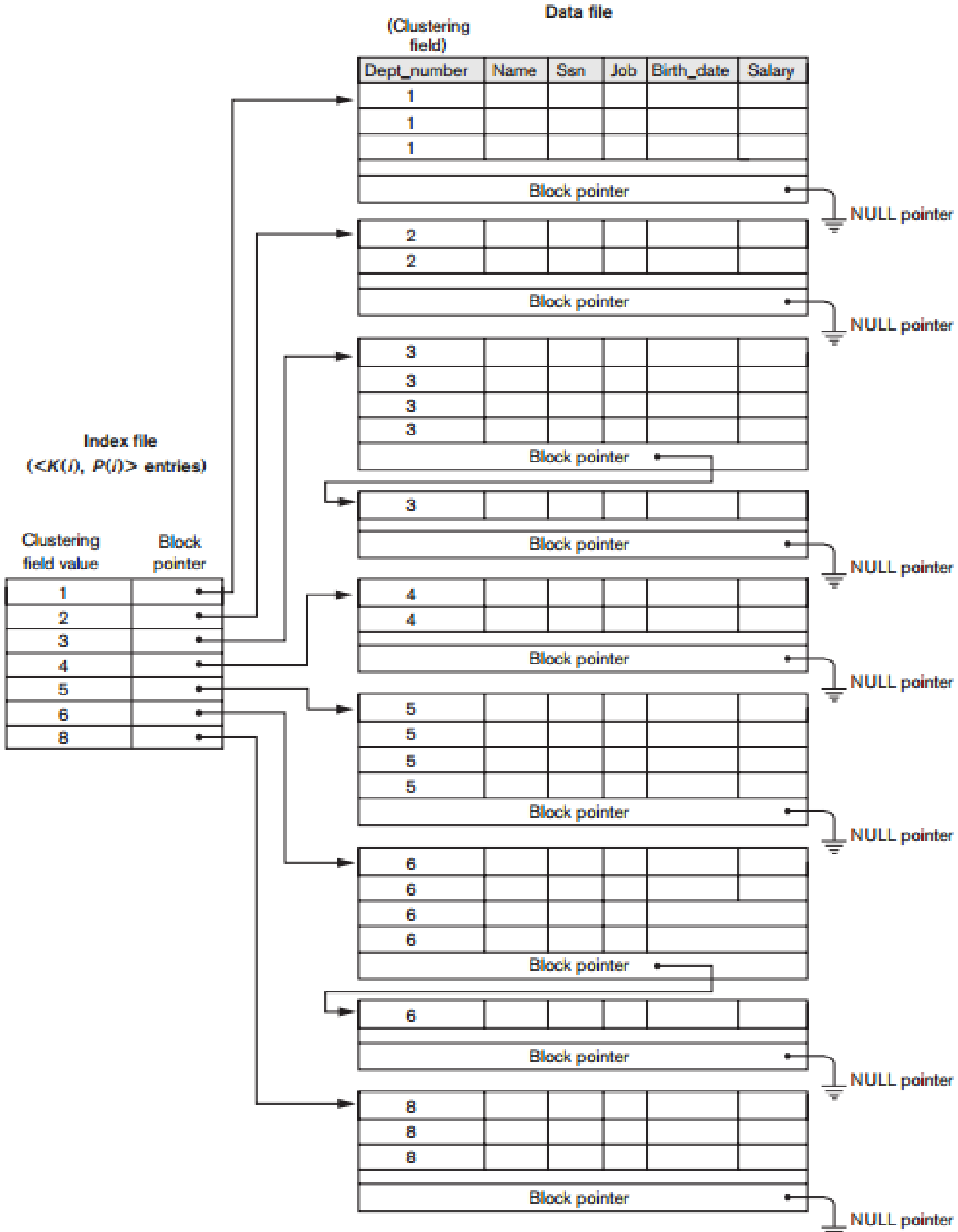
Clustering indexes



- Non-dense
- No block anchor

A clustering index on the Dept_number *ordering nonkey* field of an EMPLOYEE file

Figure 17.2, pp. 607, [1]



Clustering indexes

- Non-dense
- Block anchor

A clustering index on the Dept_number *ordering nonkey* field of an EMPLOYEE file:

a separate block cluster for each group of records that share the same value for the clustering field

Figure 17.3, pp. 608, [1]

Clustering indexes

Example 2: given the data file with the ordering non-key field DEPT_NUMBER
EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ..., DEPT_NUMBER)

record size R=150 bytes; block size B=512 bytes
number of records r=30,000 records
It is assumed that there are *125 distinct values of the DEPT_NUMBER field* and *even distribution* across DEPT_NUMBER values.

blocking factor bfr= B div R= $\lfloor B/R \rfloor$ = 512 div 150= 3 records/block
number of file blocks b = $\lceil r/bfr \rceil$= $\lceil 30{,}000/3 \rceil$ = 10,000 blocks

For a clustering index on DEPT_NUMBER, the field size V_{DEPT_NUMBER}=4 bytes, assume the block pointer size P_B=6 bytes.

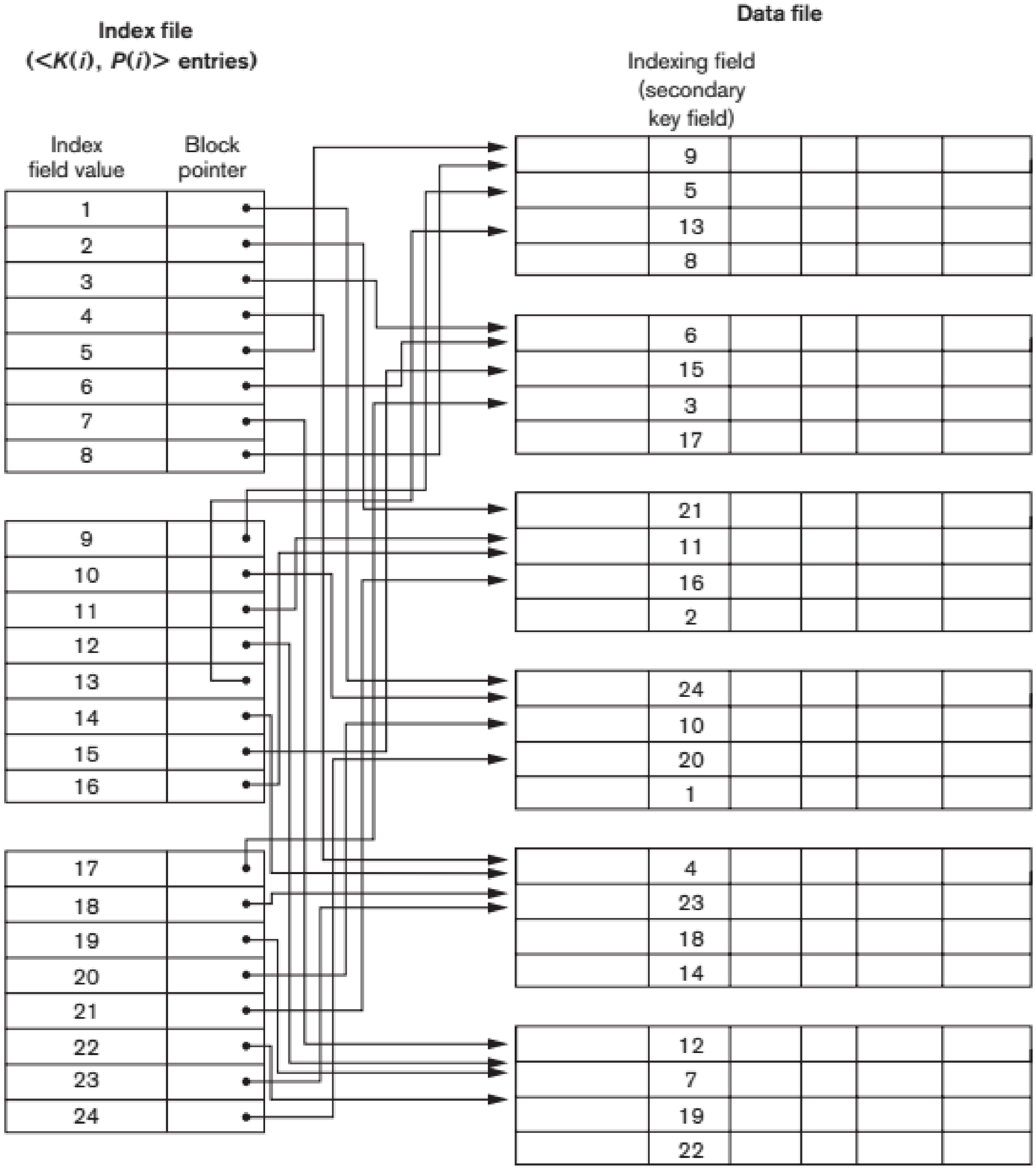
index entry size $R_i=V_{DEPT\_NUMBER}+P_B=4+6=10$ bytes
index blocking factor bfr_i= B div R_i= 512 div 10= 51 entries/block
number of index entries r_i = number of distinct values = 125 entries
number of index blocks $b_i= \lceil r_i/ bfr_i \rceil = \lceil 125/51 \rceil$ = 3 blocks
binary search on the index needs $\lceil \log_2 b_i \rceil = \lceil \log_2 3 \rceil$ = 2 block accesses
one extra block access to retrieve the ***first*** *record from the data file*
The total search cost via the index is: 2 + 1 = 3 block accesses

This is compared to an average linear search cost directly on the data file:
$\lceil b/2 \rceil = \lceil 10{,}000/2 \rceil$ = 5,000 block accesses
Because the file records are ordered, the binary search cost would be:
$\lceil \log_2 b \rceil = \lceil \log_2 10{,}000 \rceil$ = 13 block accesses

2.1. Types of Single-level Ordered Indexes

- Secondary indexes
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - a secondary index may be defined on a field which is a candidate key and has a unique value in every record, or a nonkey with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some *nonordering field* of the data file that is an *indexing field.*
    - The second field is either a *block* pointer or a *record* pointer.
  - A secondary index is a *dense index*.
    - Why?

Secondary indexes



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

→ The secondary index provides a ***logical ordering*** of the data file.

Figure 17.4, pp. 610, [1]

Secondary indexes

<u>Example 3</u>: given the following data file with the non-ordering key field SSN
EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)

record size R=150 bytes
block size B=512 bytes
number of records r=30,000 records

blocking factor bfr= B div R= $\lfloor B/R \rfloor$ = 512 div 150= 3 records/block
number of file blocks b = $\lceil r/bfr \rceil$= $\lceil 30{,}000/3 \rceil$ = 10,000 blocks

For a primary index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes.

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 entries r_i = number of file records r = 30,000 entries
number of index blocks $b_i = \lceil r_i / bfr_i \rceil = \lceil 30{,}000/32 \rceil$ = 938 blocks
binary search on the index needs $\lceil \log_2 b_i \rceil = \lceil \log_2 938 \rceil$ = 10 block accesses
one extra block access to retrieve the record from the data file
The total search cost via the index is: 10 + 1 = 11 block accesses

Because the data file is not ordered according to the values of SSN, an average linear search is done directly on the data file with the cost:
$\lceil b/2 \rceil = \lceil 10{,}000/2 \rceil$ = 5,000 block accesses

2.1. Types of Single-level Ordered Indexes

- Secondary indexes
  - A secondary index defined on a *non-ordering key* field has the number of index entries equal to the number of records in the data file.
  - A secondary index defined on a *non-ordering non-key* field can be implemented in three ways:
    - (1). Include duplicate index entries with the same *K*(*i*) value —one for each record
    - (2). Use variable-length records for the index entries, with a repeating field for the pointer
      - A list of pointers <*P*(*i*, 1), … , *P*(*i*, *k*)> in the index entry for *K*(*i*)—one pointer to each block that contains a record whose indexing field value equals *K*(*i*)
    - (3). Keep the index entries at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers

Secondary indexes

Index file

Field value	Record pointer
1	•
1	•
1	•
2	•
2	•
3	...
3	...
3	...
3	...
3	...
4	...
4	...
5	...
5	...
5	...
5	...
6	...
6	...
6	...
6	...
6	...
8	•
8	•
8	•

Data file

(Indexing field)

Dept_number	Name	Ssn	Job	Birth_date	Salary
3					
5					
1					
6					

2					
3					
4					
8					

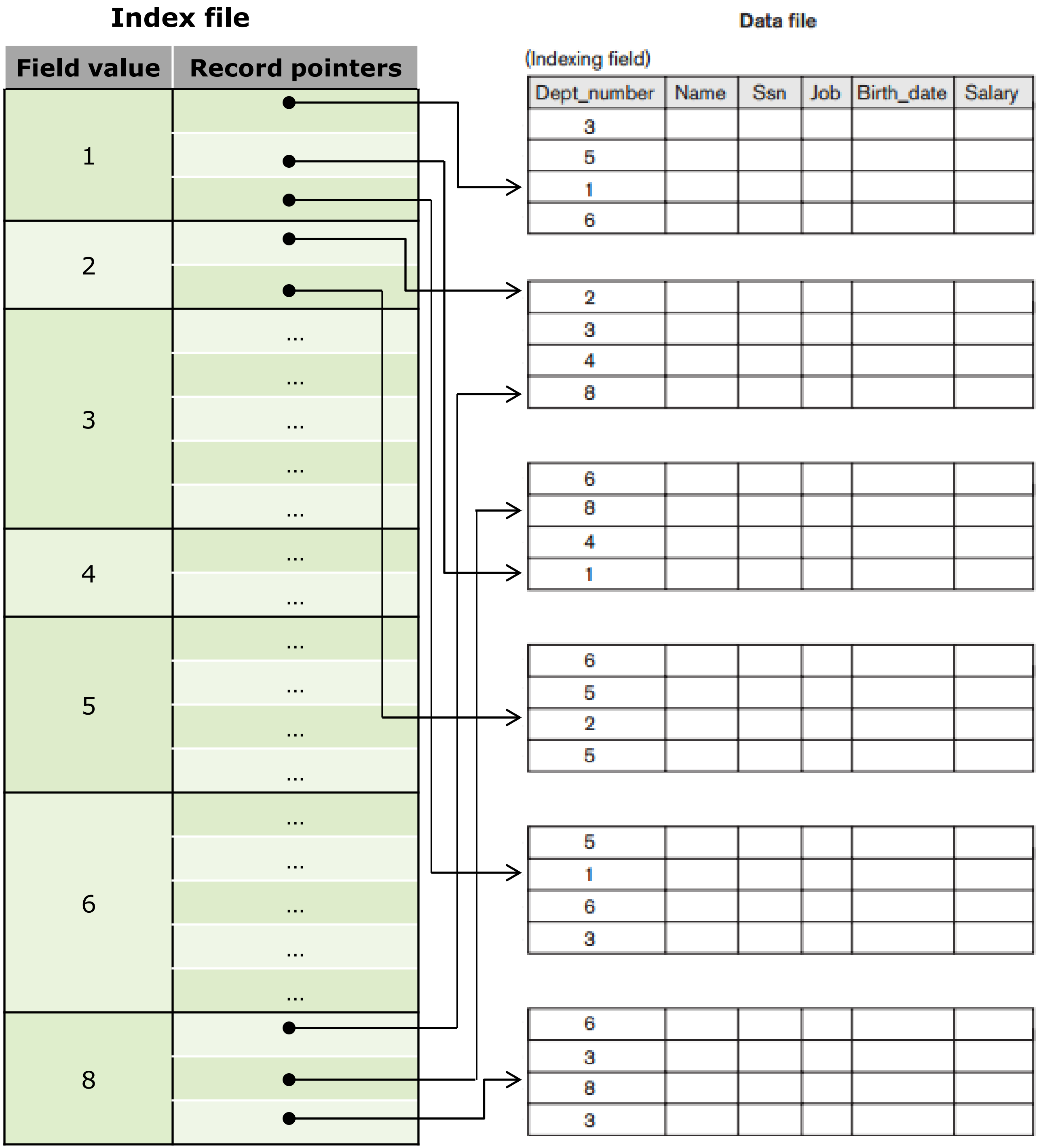
6					
8					
4					
1					

6					
5					
2					
5					

5					
1					
6					
3					

6					
3					
8					
3					

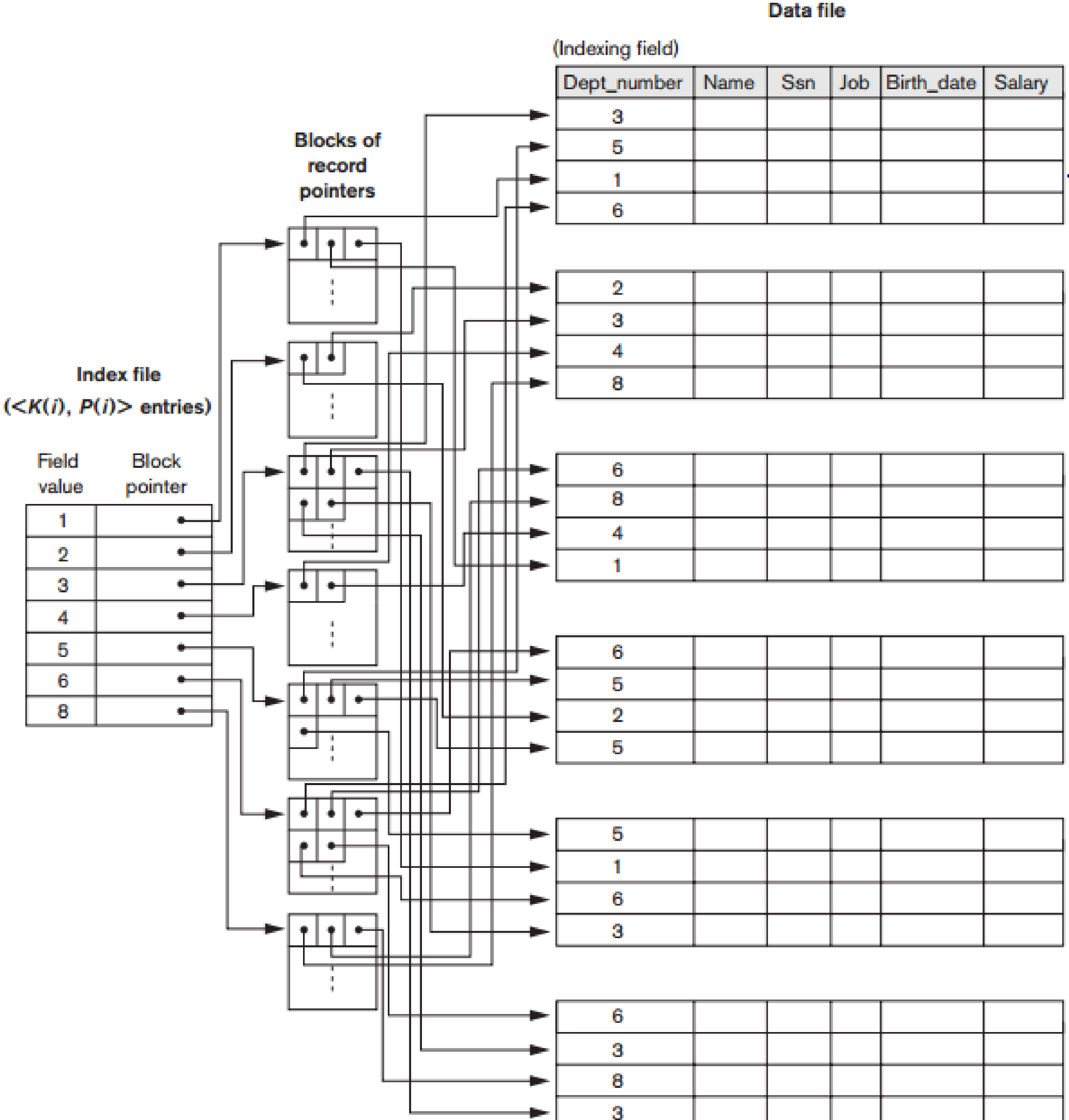
A secondary index (with record pointers) on a nonkey field implemented using ***option (1)***



Secondary indexes

A secondary index (with record pointers) on a nonkey field implemented using ***option (2)***

Secondary indexes



A secondary index (with record pointers) on a nonkey field implemented using one level of indirection in ***option (3)*** so that index entries are of fixed length and have unique field values.

Figure 17.5, pp. 612, [1]

Secondary indexes using implementation with option (3)

Example 4: given the file with the non-ordering non-key field DEPT_NUMBER EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ..., DEPT_NUMBER)

record size R=150 bytes; block size B=512 bytes

number of records r=30,000 records

It is assumed that there are *125 distinct values of the DEPT_NUMBER field* and *even distribution* across DEPT_NUMBER values.

blocking factor bfr= B div R= $\lfloor B/R \rfloor$ = 512 div 150= 3 records/block

number of file blocks b = $\lceil r/bfr \rceil$= $\lceil 30{,}000/3 \rceil$ = 10,000 blocks

For a secondary index on DEPT_NUMBER, the field size V_{DEPT_NUMBER}=4 bytes, assume the block pointer size P_B=6 bytes, the record pointer size P_R=7 bytes.

index entry size $R_i = V_{DEPT\_NUMBER} + P_B = 4+6 = 10$ bytes

index blocking factor bfr_i= B div R_i= 512 div 10= 51 entries/block

number of index entries r_i = number of distinct values = 125 entries

number of index blocks $b_i = \lceil r_i/\ bfr_i \rceil = \lceil 125/51 \rceil$ = 3 blocks

index blocking factor at the indirection level $bfr_{ii} = \lfloor B/P_R \rfloor = \lfloor 512/7 \rfloor$ = 73 pointers/block

number of index entries r_{ii} per distinct value at the indirection level = number of record pointers per distinct value of DEPT_NUMBER = $\lceil 30{,}000/125 \rceil$ = 240 pointers

number of index blocks per distinct value at the indirection level $b_{ii} = \lceil r_{ii}/bfr_{ii} \rceil = \lceil 240/73 \rceil$ = 4 blocks

Secondary indexes using implementation with option (3)

Example 4: given the file with the non-ordering non-key field DEPT_NUMBER EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ..., DEPT_NUMBER)

record size R=150 bytes; block size B=512 bytes

number of records r=30,000 records

It is assumed that there are *125 distinct values of the DEPT_NUMBER field* and *even distribution* across DEPT_NUMBER values.

To retrieve the ***first record*** from the data file, given a DEPT_NUMBER value:

binary search on the index needs $\lceil \log_2 b_i \rceil = \lceil \log_2 3 \rceil = 2$ block accesses

*one extra block access to have access to the **indirection level***

*one extra block access to retrieve the **first** record from the data file*

The total search cost via the index is: 2 + 1 + 1 = 4 block accesses

To retrieve ***all*** the records with the indirection level and *even distribution*, given a DEPT_NUMBER value:

binary search on the index needs $\lceil \log_2 b_i \rceil = \lceil \log_2 3 \rceil = 2$ block accesses

number of block accesses to the blocks in the indirection level: 4 block accesses,

number of block accesses to the data file: $\lceil 30{,}000/125 \rceil = 240$ block accesses

The total cost to retrieve all the records = 2 + 4 + 240 = 246 block accesses

2.1. Types of Single-level Ordered Indexes

Table 17.1, pp. 613, [1]

Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	**Index Field Not Used for Physical Ordering of the File**
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

2.1. Types of Single-level Ordered Indexes

Table 17.2, pp. 613, [1]

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

[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.

2.2. Multilevel Indexes

- Because a single-level index is an *ordered* file, we can create another *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 the third, fourth, ..., top level until all entries of the *top level* fit in one disk block.
- A multilevel index can be created for any type of the first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block.

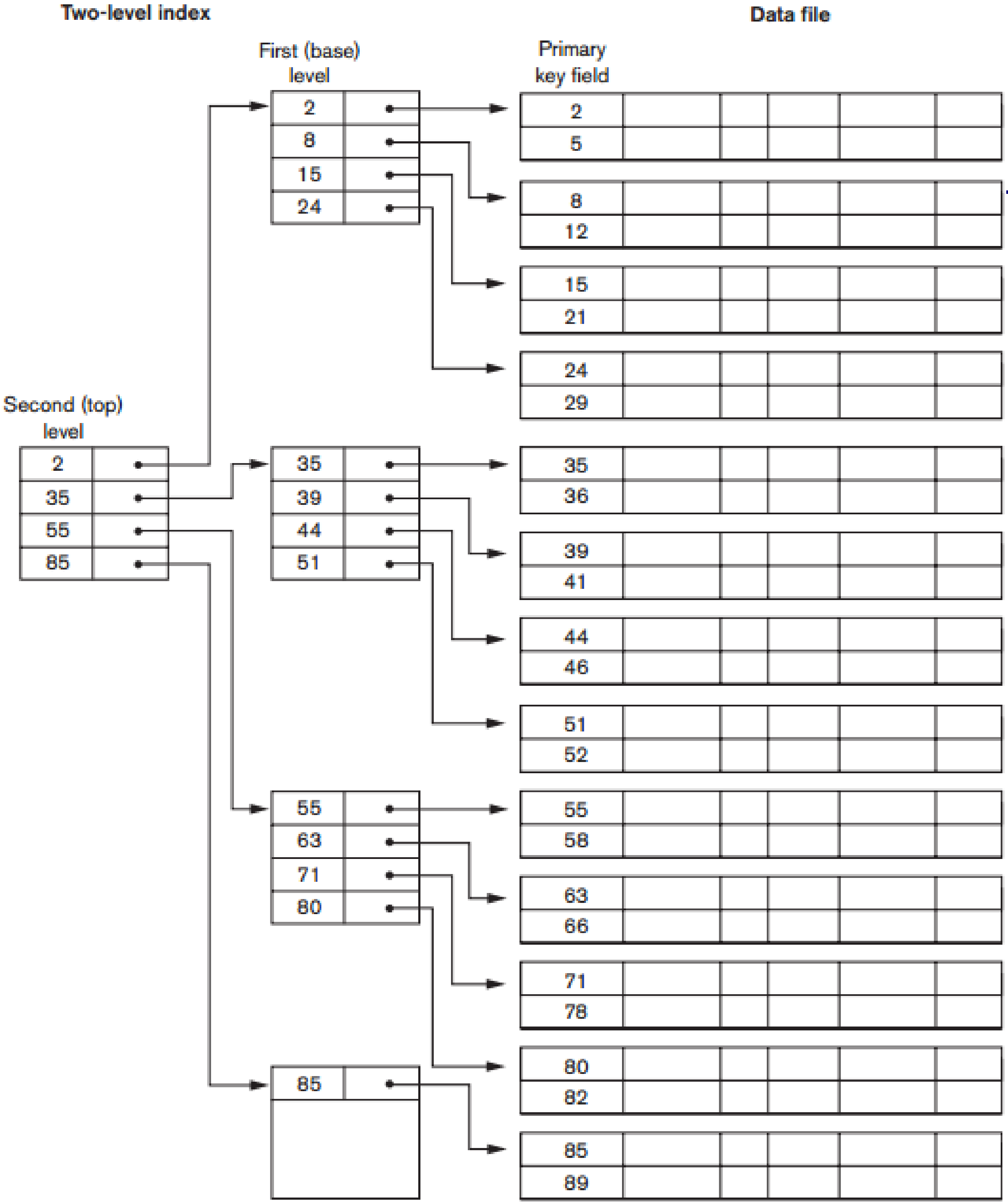
2.2. Multilevel Indexes

- A binary search is applied to *the single-level ordered index file* to locate pointers to a disk block or to a record (or records) in the file having a specific index field value.
  - A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks because each step reduces the part of the index file that we continue to search by a factor of 2.
- For much faster search, a **multilevel index** reduces the part of the index that we continue to search by bfr_i, the blocking factor for the index, called the **fan-out** *fo*.

2.2. Multilevel Indexes

- We divide the *record search space* into two halves at each step during a *binary search*.
- We divide the record search space *n*-ways (where n = the *fan-out*) at each search step using the *multilevel index*.
- Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2.
  - In most cases, the fan-out is much larger than 2.

Multilevel Indexes



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

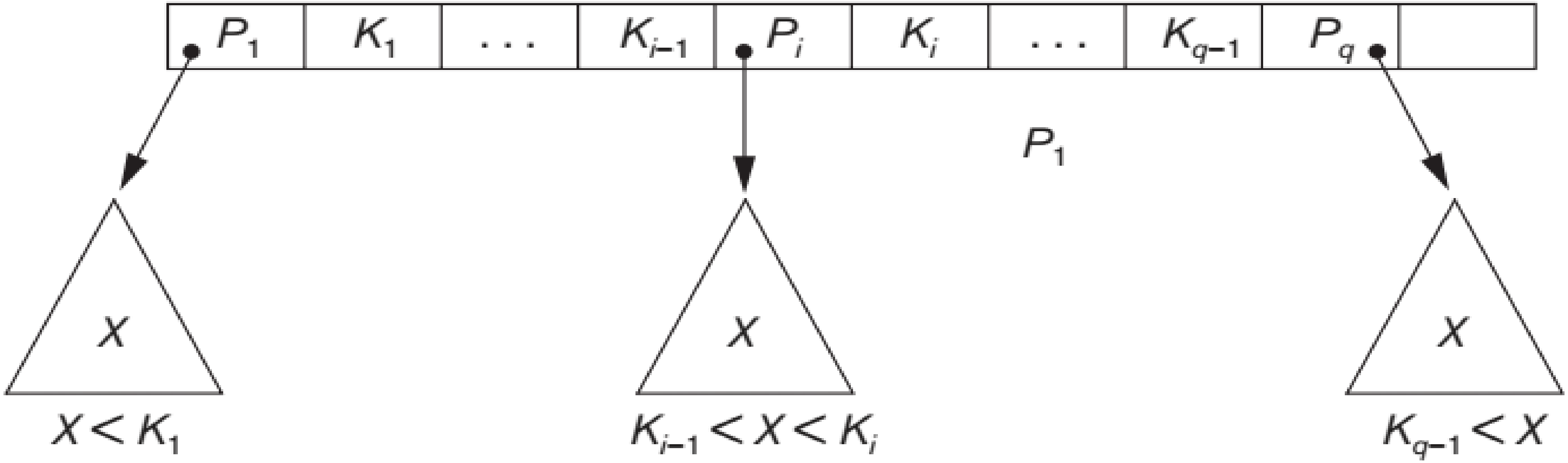
Figure 17.6, pp. 615, [1]

2.2. Multilevel Indexes

- Example 5: Generate a multilevel index on the single-level secondary index on the non-ordering key field SSN in Example 3.
  - The index blocking factor bfr_i = 32 index entries/block
    - This is the fan-out *fo* of the multilevel index.
  - The number of the first-level index blocks b_1 = 938 blocks
  - → Create other levels until we reach the top level with one index block
  - The number of the second-level index blocks $b_2 = \lceil b_1/fo \rceil = \lceil 938/32 \rceil$ = 30 blocks
  - The number of the third-level index blocks = $b_3 = \lceil b_2/fo \rceil = \lceil 30/32 \rceil$ = 1 block
  - → The third-level index is the top one.
  - → Equality search for a record with a given SSN via the multilevel index costs: 3 *(index levels)* + 1 *(data)* = 4 block accesses.
  - → Equality search for a record with a given SSN via the single-level index with binary search costs: 11 block accesses.

2.2. Multilevel Indexes

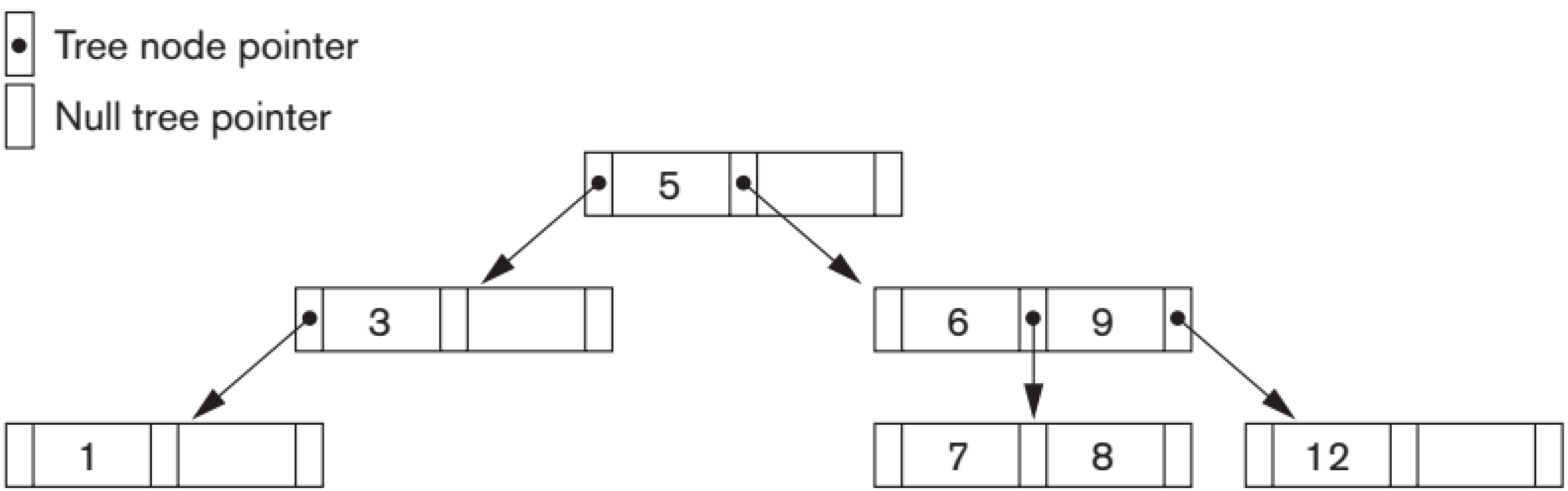
- A multilevel index is a form of a *search tree*; however, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file.*
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that *leaves some space in each of its blocks* for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks *when the data file grows and shrinks*.



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

q ≤ p where p is the tree order.

Figure 17.8, pp. 618, [1]



A search tree of order p = 3

Figure 17.9, pp. 619, [1]

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

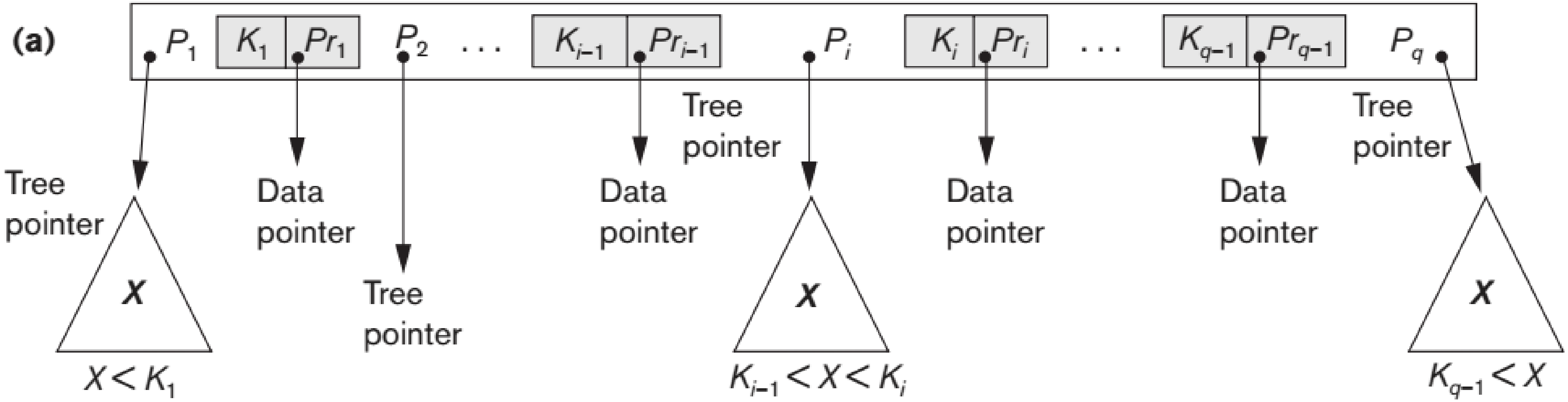
- B-trees and B+-trees are special cases of the **balanced** search tree structure.
- A **tree** is formed of **nodes**.
  - Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent.
  - A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node.
  - The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.

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

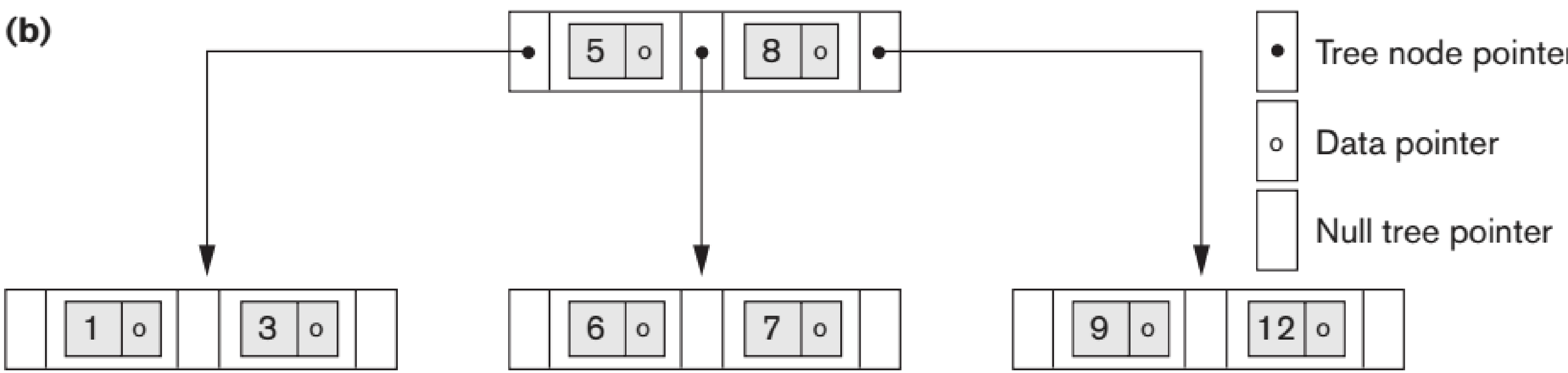
- B-Tree and B+-Tree
  - Each node is stored in a disk block.
  - Each node is kept between 50 and 100 percent full.
  - All the leaf nodes are at the same level.
- In B-tree, pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure.
- B+-tree is a variation of B-tree.
  - Pointers to the data blocks are stored only in leaf nodes.
  - The leaf nodes are usually linked to each other.

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

- All the nodes in B-tree are of the same structure.
- In B+-tree, the structure of internal nodes is different from that of leaf nodes.
  - For the same data file, B+-tree has fewer levels.
  - For the same levels, B+-tree is a higher-capacity index.
- B+-tree is a common structure used for indexing in current DBMSs.



(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.

B-tree structures

Figure 17.10, pp. 621, [1]

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

- A **B-tree of order** *p*, when used as an access structure on a *key field* to search for records in a data file, can be defined:

  1. Each internal node in the B-tree is of the form $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$ where $q \le p$. Each P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).

  2. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.

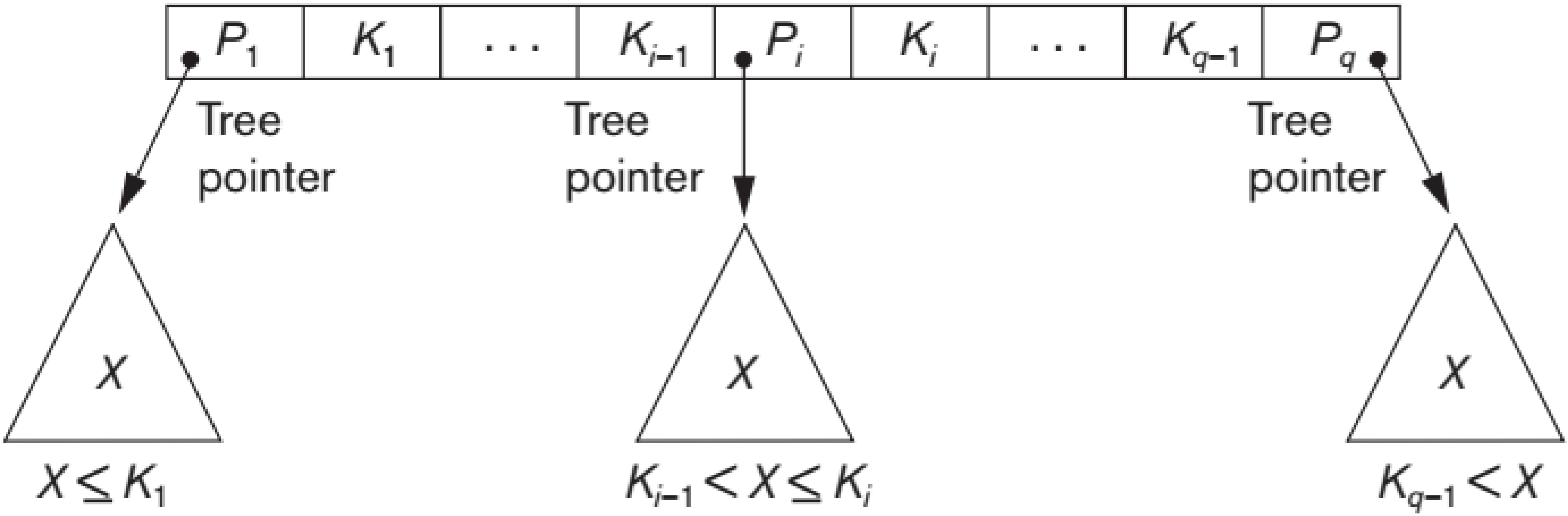
  3. For all search key field values X in the subtree pointed at by P_i, we have: $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$

  4. Each node has at most *p* tree pointers.

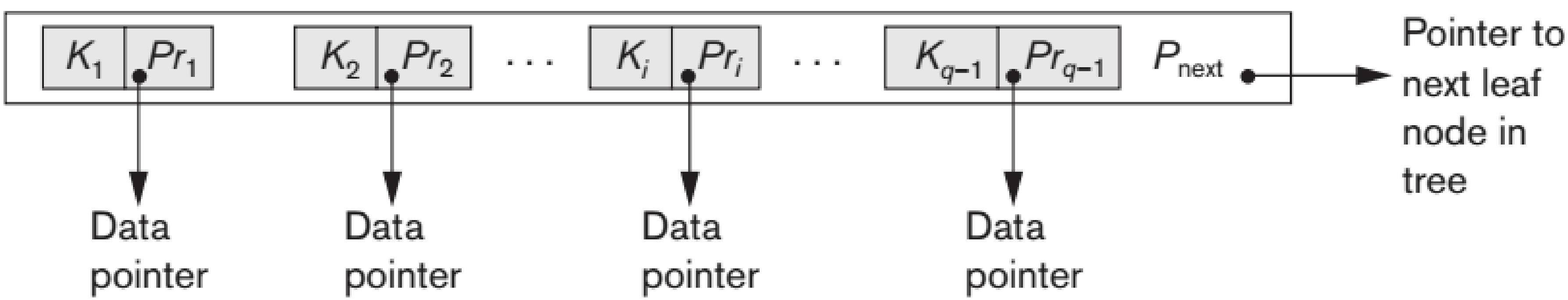
  5. Each node, except the root and leaf nodes, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

  6. A node with q tree pointers, $q \le p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).

  7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.



(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.

The nodes of a B+-tree

Figure 17.11, pp. 623, [1]

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

- The structure of the *internal nodes* of a B+-tree of order p:

  1. Each internal node is of the form
  $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$
  where $q \leq p$ and each P_i is a **tree pointer**.

  2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.

  3. For all search values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

  4. Each internal node has at most p tree pointers.

  5. Each internal node, except the root, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.

  6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

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

- The structure of the *leaf nodes* of a B+-tree of order p_{leaf}:

  1. Each leaf node is of the form
  $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots , <K_{q-1}, Pr_{q-1}>, P_{next}>$
  where $q \le p_{leaf}$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node*.

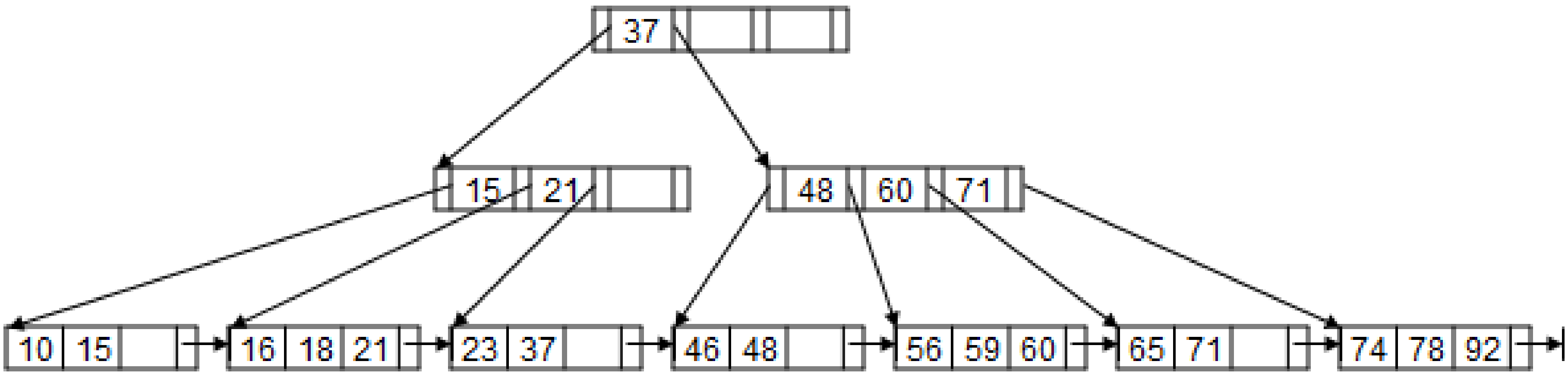
  2. Within each leaf node, $K_1 \le K_2 \ldots , K_{q-1}, q \le p_{leaf}$.

  3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).

  4. Each leaf node has at least $\lceil p_{leaf}/2 \rceil$ values.

  5. All leaf nodes are at the same level.

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



A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Insertion sequence:

23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16

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

- In the following examples, the capacity index of B-tree and that of B+-tree are examined with the same number of levels.
- Given the parameters:
  - Disk block size B = 512 bytes
  - Search field size V = 9 bytes
  - Block pointer P = 6 bytes
  - Record (data) pointer P_r = 7 bytes
  - Each node in both B-tree and B+-tree is 69% full.

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

- Consider a B-tree: order *p* and 69% full nodes
  - Each node have at most *p* tree pointers, *p*-1 data pointers, and *p*-1 search values.
  - Each node must fit into a single disk block.
  - Using the aforesaid parameters, *p* is calculated:

$p*P + (p-1)*(P_r+V) \le B$

$p*6 + (p-1)*(7+9) \le 512$

$22*p \le 528$

$p \le 24$

<u>Selected</u>: p = 23

<u>Reason</u>: **p=23 instead of p=24** because a node contains additional information needed for manipulating the tree, such as the number of entries *q* in the node and a pointer to the parent node.

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

- Consider a B-tree: order *p* and 69% full nodes
  - On the average, each node has:
    - p*0.69 = 23*0.69 = 15.87 ≈ 16
      - → 16 pointers and 15 search values
  - The **average fan-out** *fo* = 16.
  - The **capacity** of a *three-level* B-tree is:

Root:	1 node	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3,840 entries	4,096 pointers
Level 3:	4,096 nodes	61,440 entries	

→ The total number of index entries

= 15 + 240 + 3,840 + 61,440

= 65,535 entries

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

- Consider a B+-tree: orders p, p_{leaf}, and also 69% full nodes
  - Each internal node have up to p tree pointers and p-1 search values, fitting in a single block.
  - Each leaf node has the same number of values and pointers, except that the pointers are data pointers (record pointers) and a next block pointer.

$p*P + (p-1)*V \leq B$

$p*6 + (p-1)*9 \leq 512$

$15*p \leq 521$

<u>Selected</u>: $p = 34$

$p_{leaf}*(P_r+V) + P \leq B$

$p_{leaf}*(7+9) + 6 \leq 512$

$16*p_{leaf} \leq 506$

<u>Selected</u>: $p_{leaf} = 31$

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

- Consider a B+-tree: orders p, p_{leaf}, and also 69% full nodes
  - Each internal node has: 34*0.69 ≈ 23 pointers.
  - Each leaf node has: 31*0.69 ≈ 21 data pointers.
  - **Fan-out** *fo* = 23 and 21 for internal and leaf nodes
  - The **capacity** of a three-level B+-tree:
    - **Root**: 1 node 22 entries 23 pointers
    - **Level 1**: 23 nodes 506 entries 529 pointers
    - **Level 2**: 529 nodes 11,638 entries 12,167 pointers
    - **Leaf level**: 12,167 nodes 255,507 data pointers
    - → The total number of index entries = 255,507 entries

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

- Given the same parameters, a three-level B-tree and a three-level B+-tree are obtained:

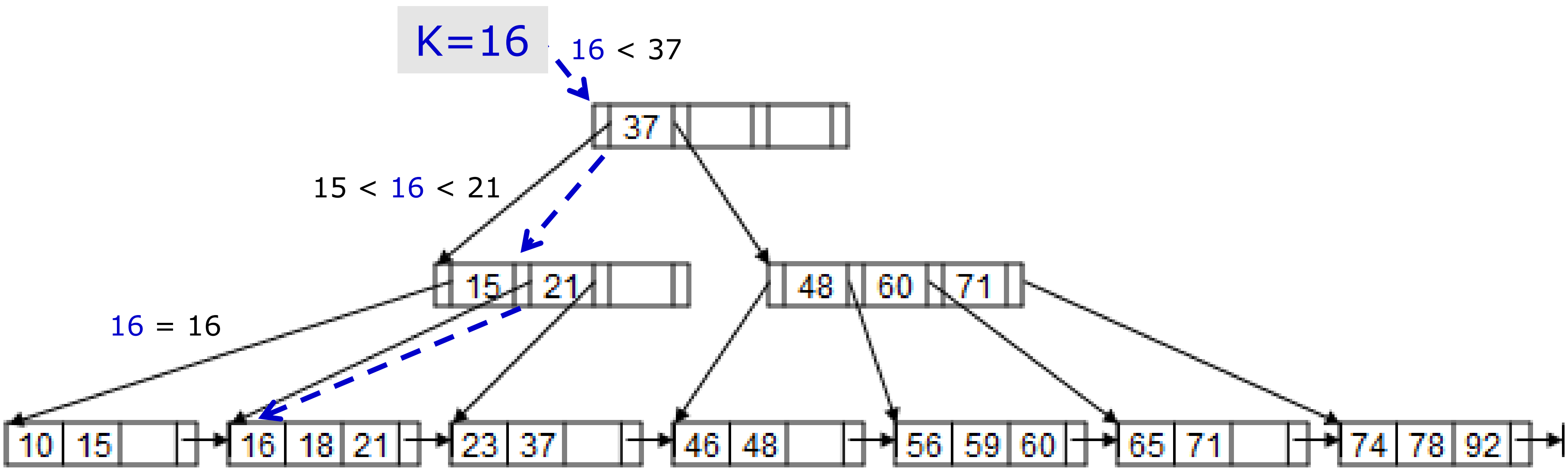
	B-tree	B+-tree
Order	$p = 23$	$p = 34$ $p_{leaf} = 31$
Fan-out (69% full)	16	Internal node: 23 Leaf node: 21
Capacity	65,535 entries	255,507 entries
Equality search	2-5 block accesses	5 block accesses

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

- Search for a Record with Search Key Field Value *K*, using B+-tree (Algorithm 17.2, pp. 625-626, [1])

$n \leftarrow$ block containing root node of B^+-tree;
read block n;
while (n is not a leaf node of the B^+-tree) do
    begin
    $q \leftarrow$ number of tree pointers in node n;
    if $K \leq n.K_1$ (*$n.K_i$ refers to the ith search field value in node n*)
        then $n \leftarrow n.P_1$ (*$n.P_i$ refers to the ith tree pointer in node n*)
        else if $K > n.K_{q-1}$
            then $n \leftarrow n.P_q$
            else **begin**
                search node n for an entry i such that $n.K_{i-1} < K \leq n.K_i$;
                    $n \leftarrow n.P_i$
                    end;
    read block n
    end;
search block n for entry (K_i, Pr_i) with $K = K_i$; (* search leaf node *)
if found
    then read data file block with address Pr_i and retrieve record
    else the record with search field value K is not in the data file;

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



A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Search for 16, 62:

16: left 37, right 15, 16 *(found)*

62: right 37, right 60, left 65 *(check data)*

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

- Search using B-trees and B+-trees
  - Search conditions on indexing attributes
    - =, <, >, ≤, ≥, between, MINIMUM value, MAXIMUM value
  - Search results
    - Zero, one, or many data records
  - Search cost
    - B-trees
      - From 1 to (1 + the number of tree levels) + data accesses
    - B+-trees
      - 1 (root level) + the number of tree levels + data accesses
- Logically ordering for a data file

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

- An insertion into a node that is not full is quite efficient.
- If a node is full, the insertion causes "*overflow*".
  - A split of the current node into two nodes is done.
  - 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 "*underflow*", i.e. a node becomes less than half full, it might be merged with neighboring nodes.

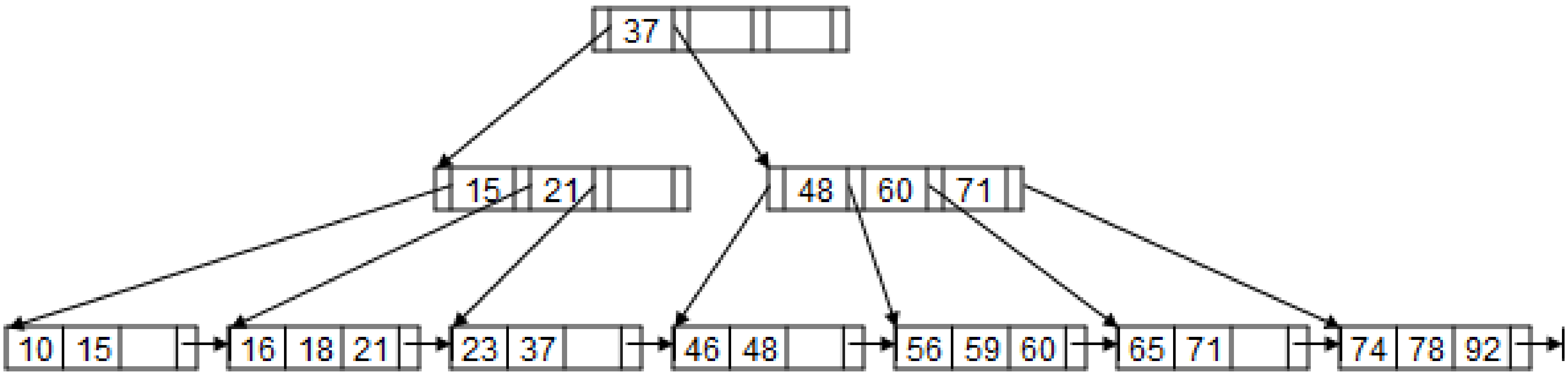
2.3. Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Insert a Record with Search Key Field Value *K* in a B+-Tree of Orders p, p_{leaf}
  - Search a leaf node n in which (K, P_r) is inserted
  - If the leaf node n is not full, then insert (K, P_r).
  - Otherwise, the node *overflows* and must be split.
    - The first $j = \lceil (pleaf + 1)/2 \rceil$ entries in the original node are kept there, and the remaining entries are moved to a new leaf node.
    - The j^{th} search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent.
    - These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split.

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

- Insert a Record with Search Key Field Value *K* in a B+-Tree of Orders p, p_{leaf}
  - Search a leaf node n in which (K, P_r) is inserted
  - If the leaf node n is not full, then insert (K, P_r).
  - Otherwise, the node *overflows* and must be split.
    - If the parent internal node is full, the new value will cause it to overflow also, so it must be split.
      - The entries up to P_j—the j^{th} tree pointer after inserting the new value and pointer, where $j = \lfloor (p + 1)/2 \rfloor$—are kept, whereas the j^{th} search value is moved to the parent, **not replicated**.
      - A new internal node will hold the entries from P_{j+1} to the end of the entries in the node.
      - This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree.

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

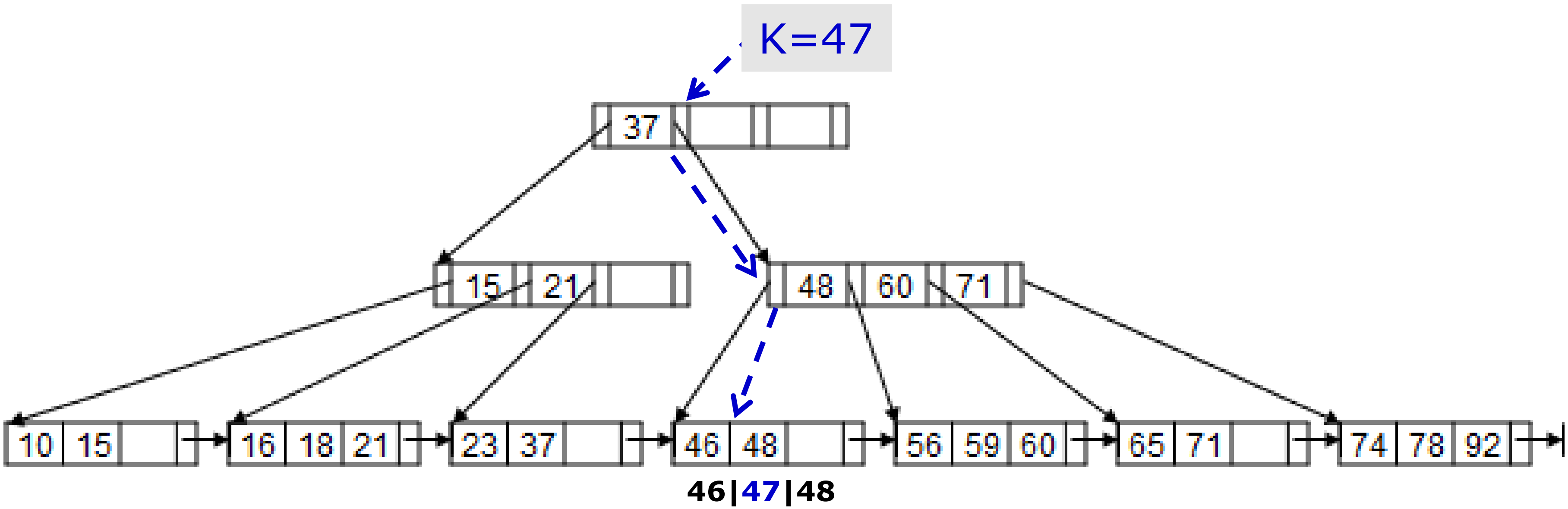


A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Insertion: 47, 58

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

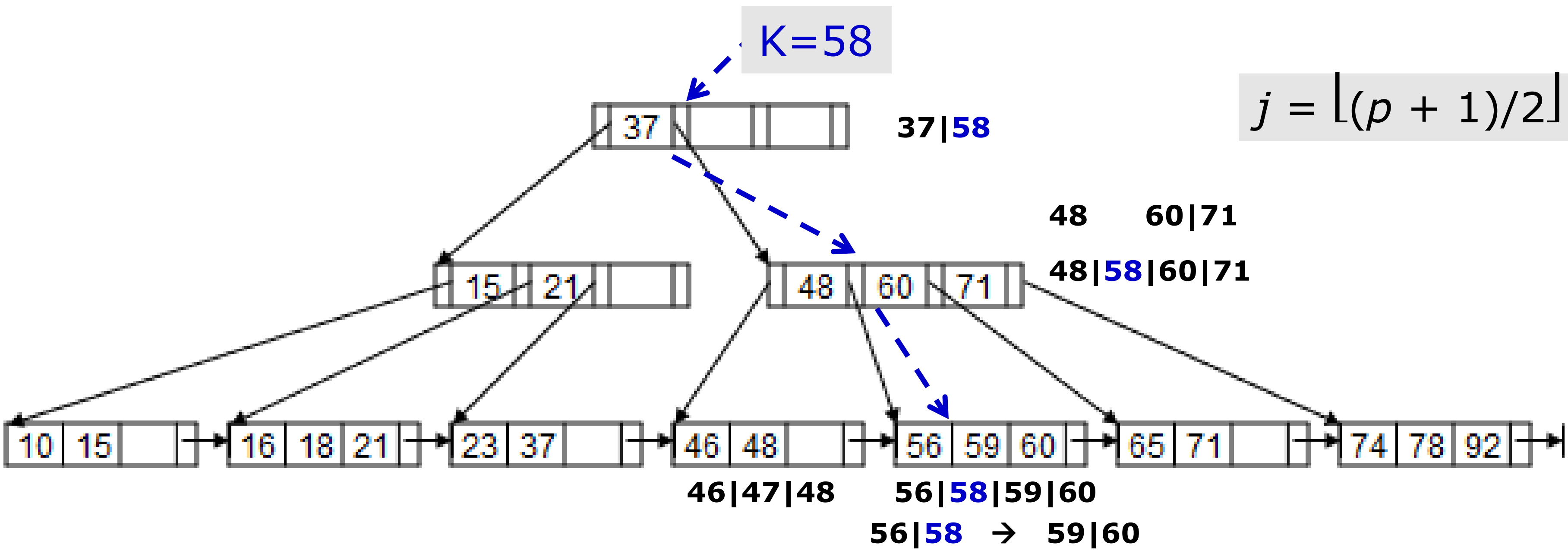


A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Insertion: 47, 58

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



$j = \lfloor (p + 1)/2 \rfloor$

$j = \lceil (p_{\text{leaf}} + 1)/2 \rceil$

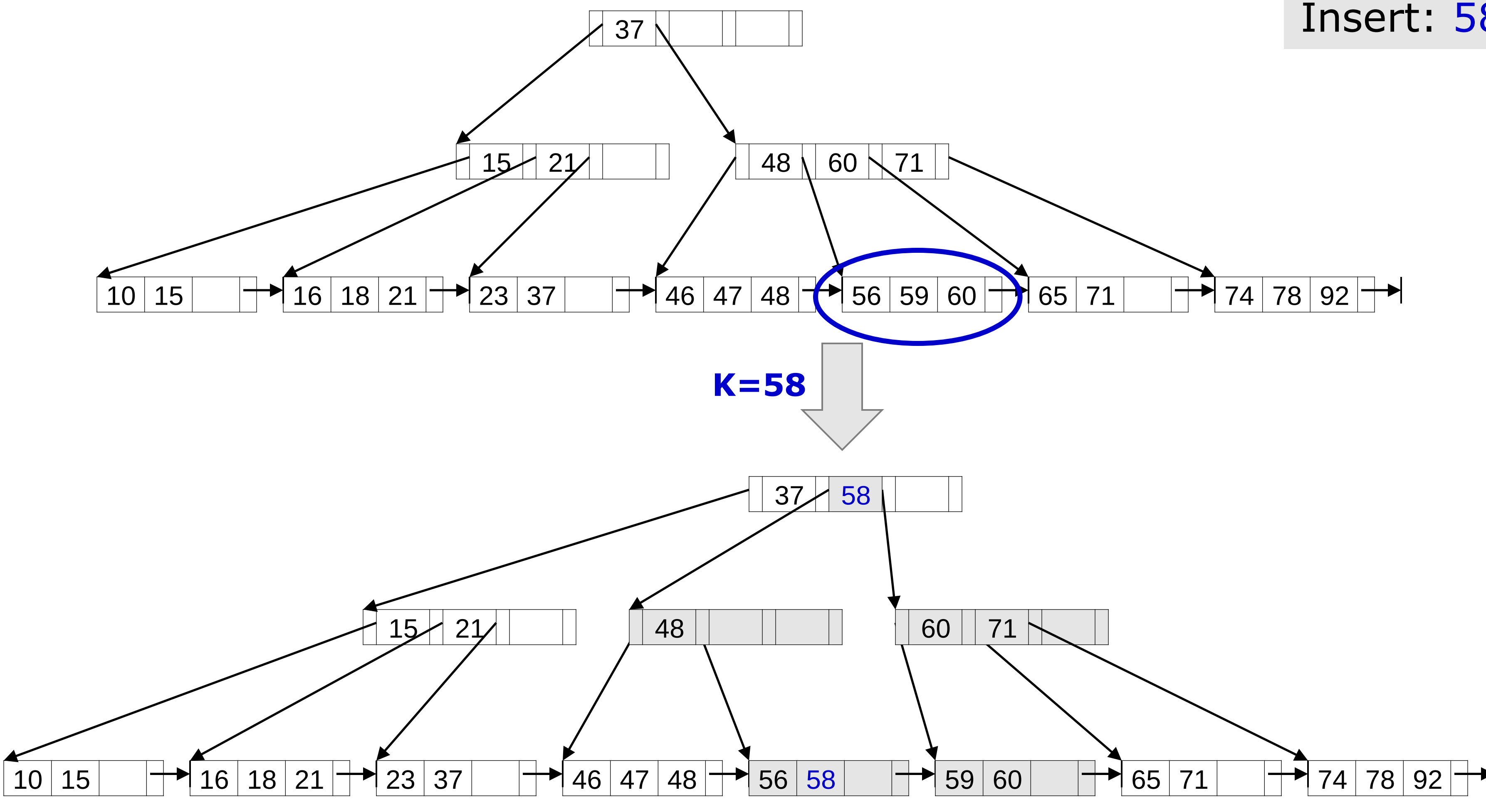
A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Insertion: 47, 58

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

Insert: 58



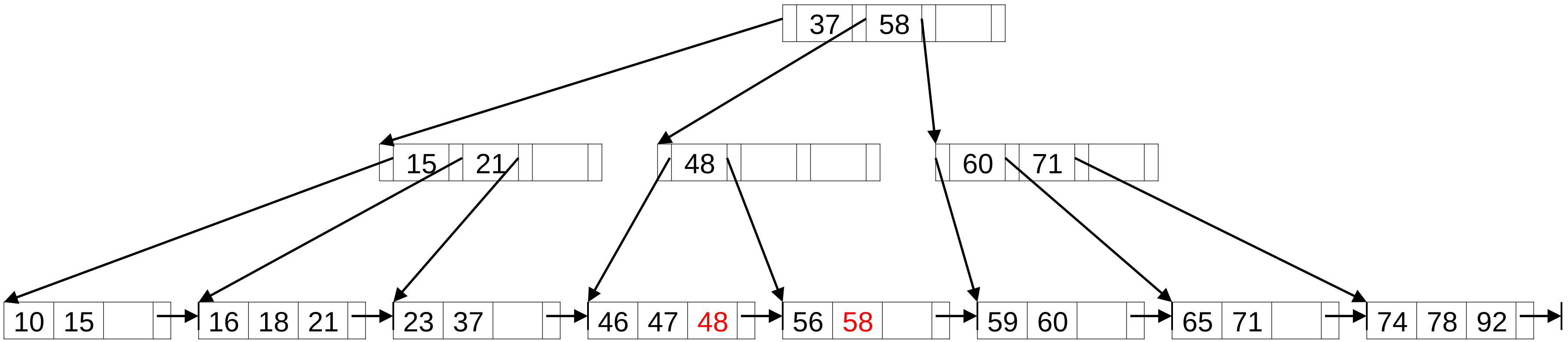
2.3. Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Delete a Record with Search Key Field Value *K* in a B+-Tree of Orders p, p_{leaf}
  - Search a leaf node n from which (K, P_r) is removed
  - If found, it is always removed from the leaf level.
  - If it happens to occur in an internal node, it must also be removed from there.
    - *The value to its left* in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree.
  - Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required.

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

- Delete a Record with Search Key Field Value *K* in a B+-Tree of Orders p, p_{leaf}
  - Deletion may cause **underflow**.
    - Find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full.
    - Otherwise, the node is merged with its siblings and the number of leaf nodes is reduced.
    - A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If also not possible, the three nodes are merged into two leaf nodes.
    - Underflow may propagate to **internal** nodes and reduce the tree levels.

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



A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

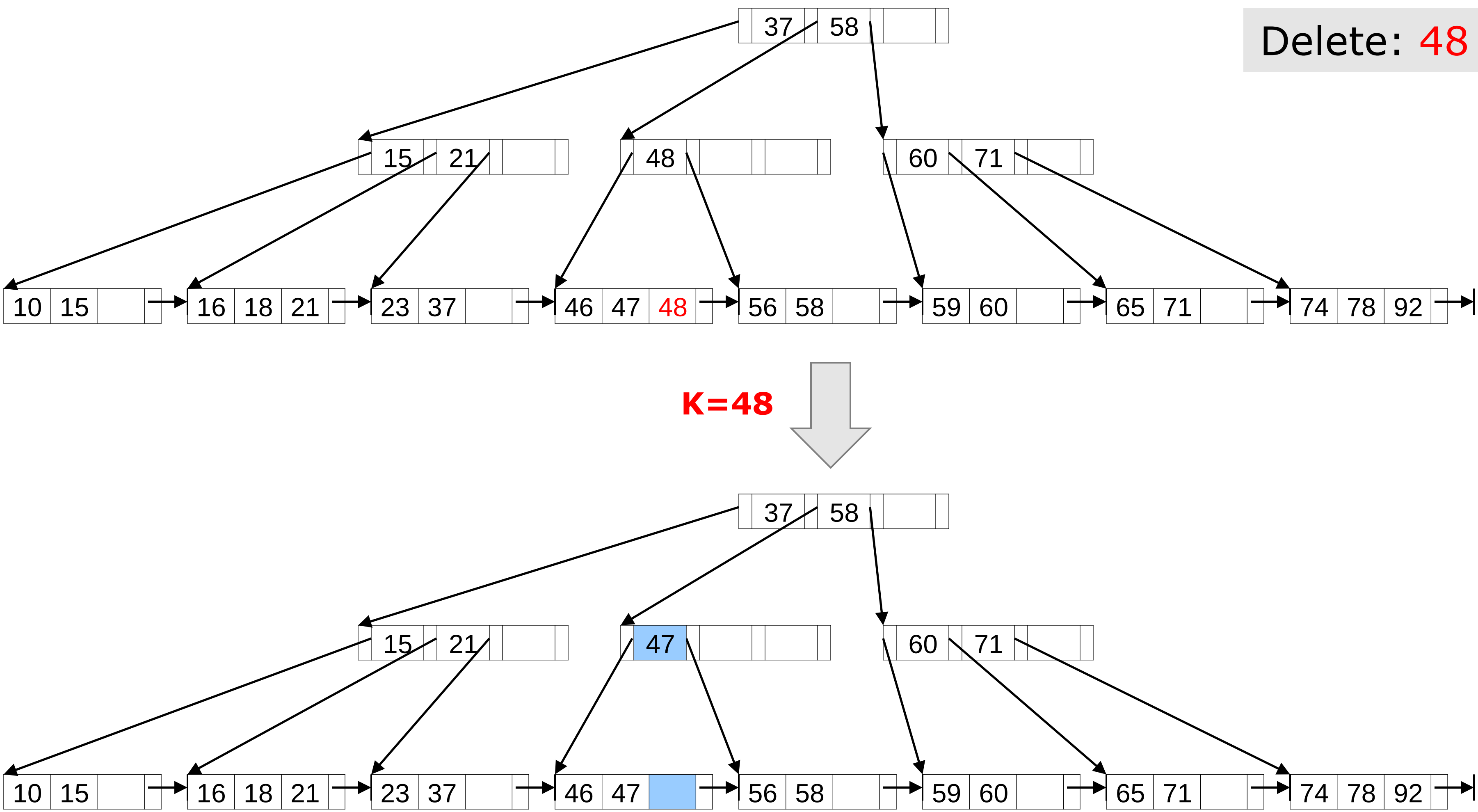
For simplicity, data pointers are not included in the leaf nodes.

Underflow: number of tree pointers < $\lceil p/2 \rceil$ = 2

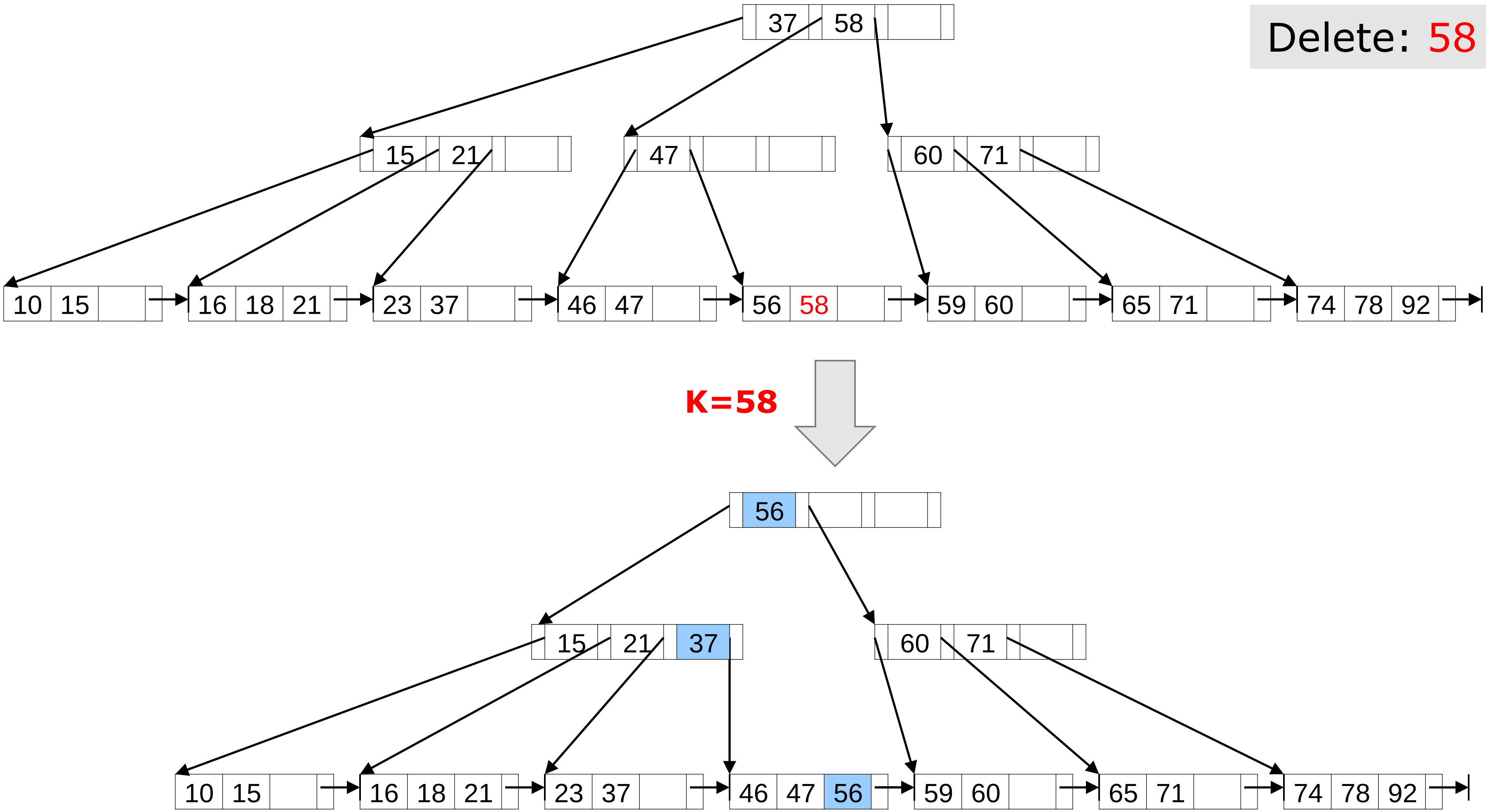
number of data pointers < $\lceil p_{leaf}/2 \rceil$=2

Deletion: 48, 58

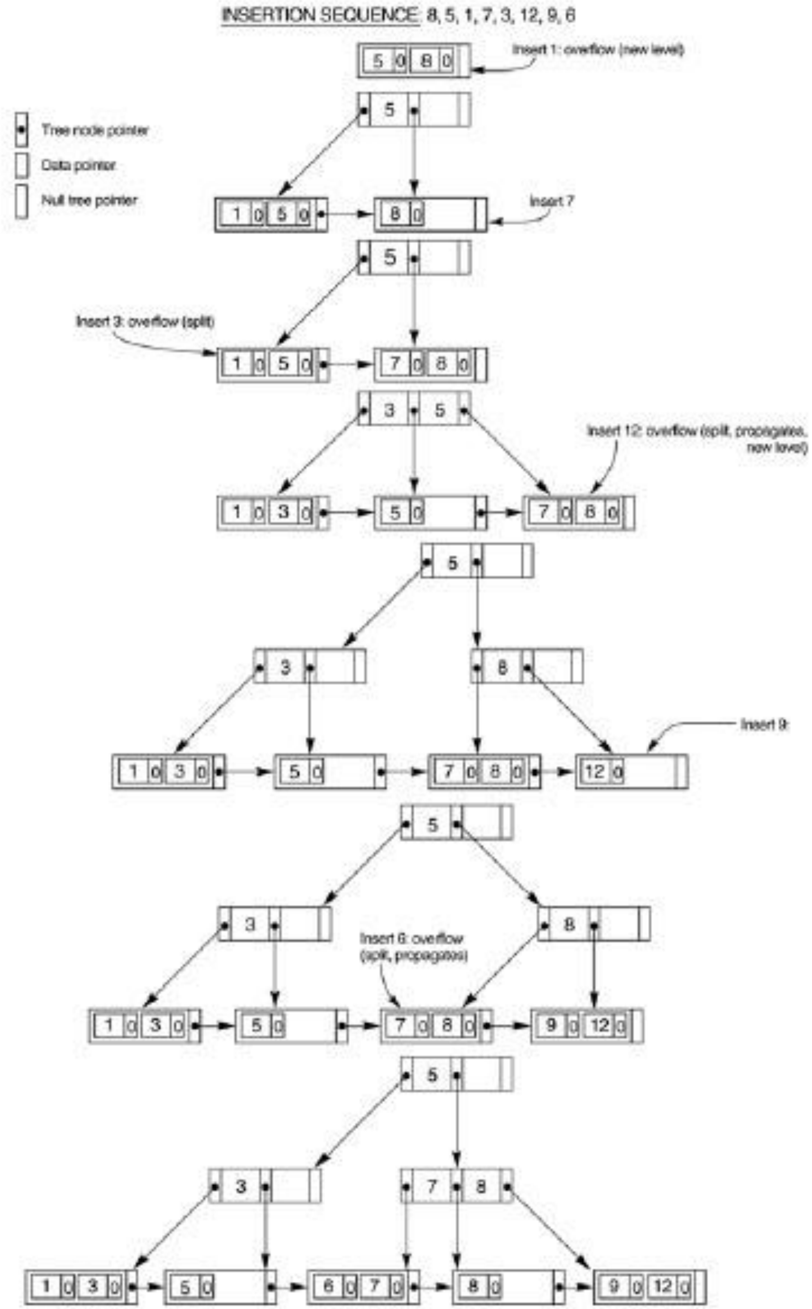
2.3. Dynamic Multilevel Indexes Using B-Trees and B+-Trees



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



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

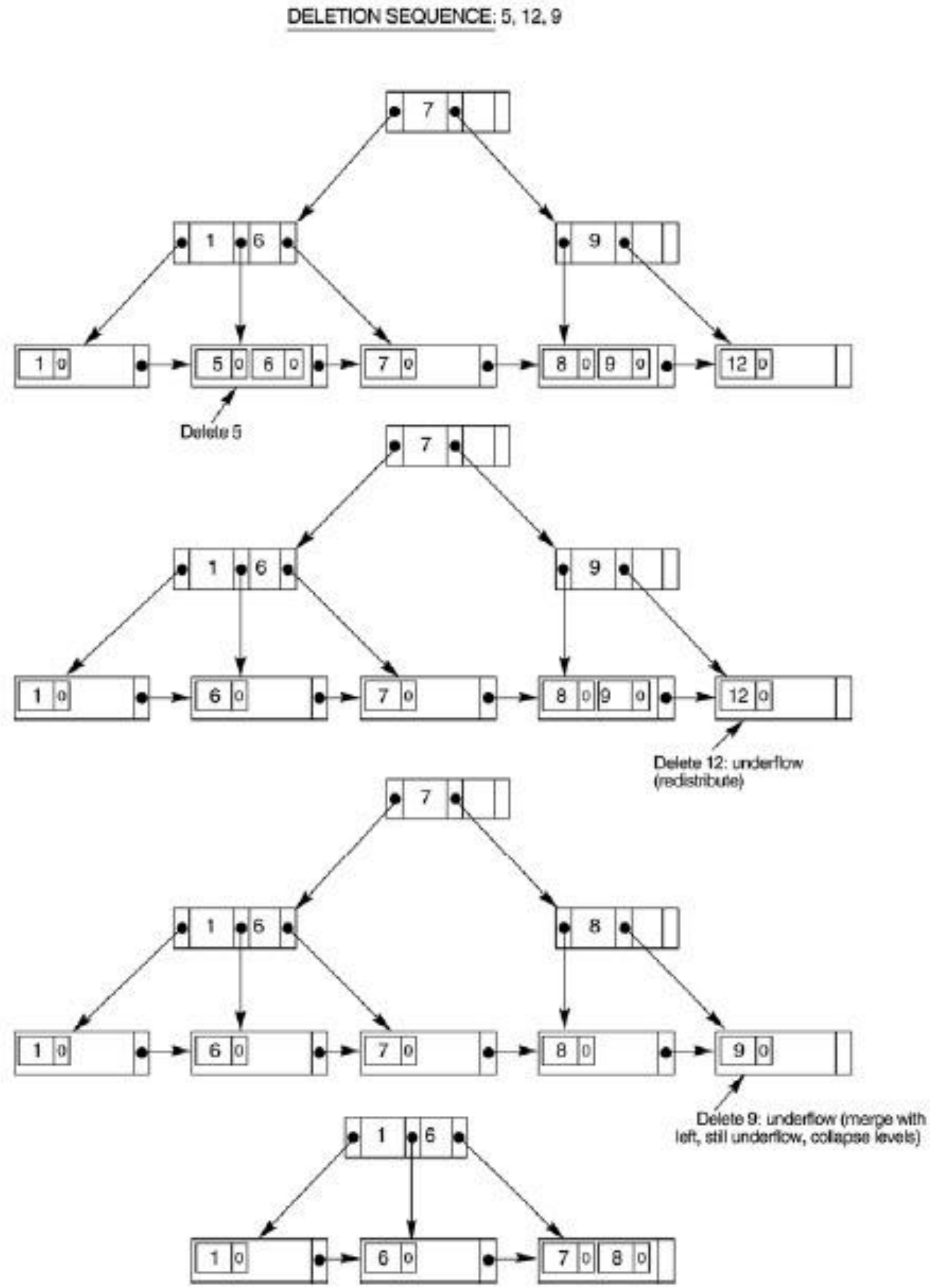


Original insertion:

8, 5, 1, 7, 3, 12, 9, 6

Update this B+-tree with *more insertions*:

15, 4, 10



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

Original deletion:

5, 12, 9

Update this B+-tree with *more deletions*:

6, 1

2.4. Indexes on Multiple Keys

- In many retrieval and update requests, multiple attributes are involved.
- If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.
- if an index is created on attributes $<A_1, A_2, \ldots, A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots, v_n>$.
- A lexicographic ordering of these tuple values establishes an order on this composite search key.
- An index on a composite key of n attributes works similarly to any index discussed so far.

2.5. Other File Indexes

- Hash indexes
  - The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization.
- Bitmap indexes
  - A bitmap index is built on **one particular value** of a field (the column in a table) with respect to all the rows (records) and is an array of bits.
- Function-based indexes
  - In Oracle, an index such that the value that results from applying a function (expression) on a field or some fields becomes the key to the index

2.5. Other File Indexes

- Hash indexes
  - The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization.
    - access structures similar to indexes, based on *hashing*
  - Support for *equality searches on the hash field*

Hash-based indexing

a *hashing function*: the sum of the digits of Emp_id modulo 10

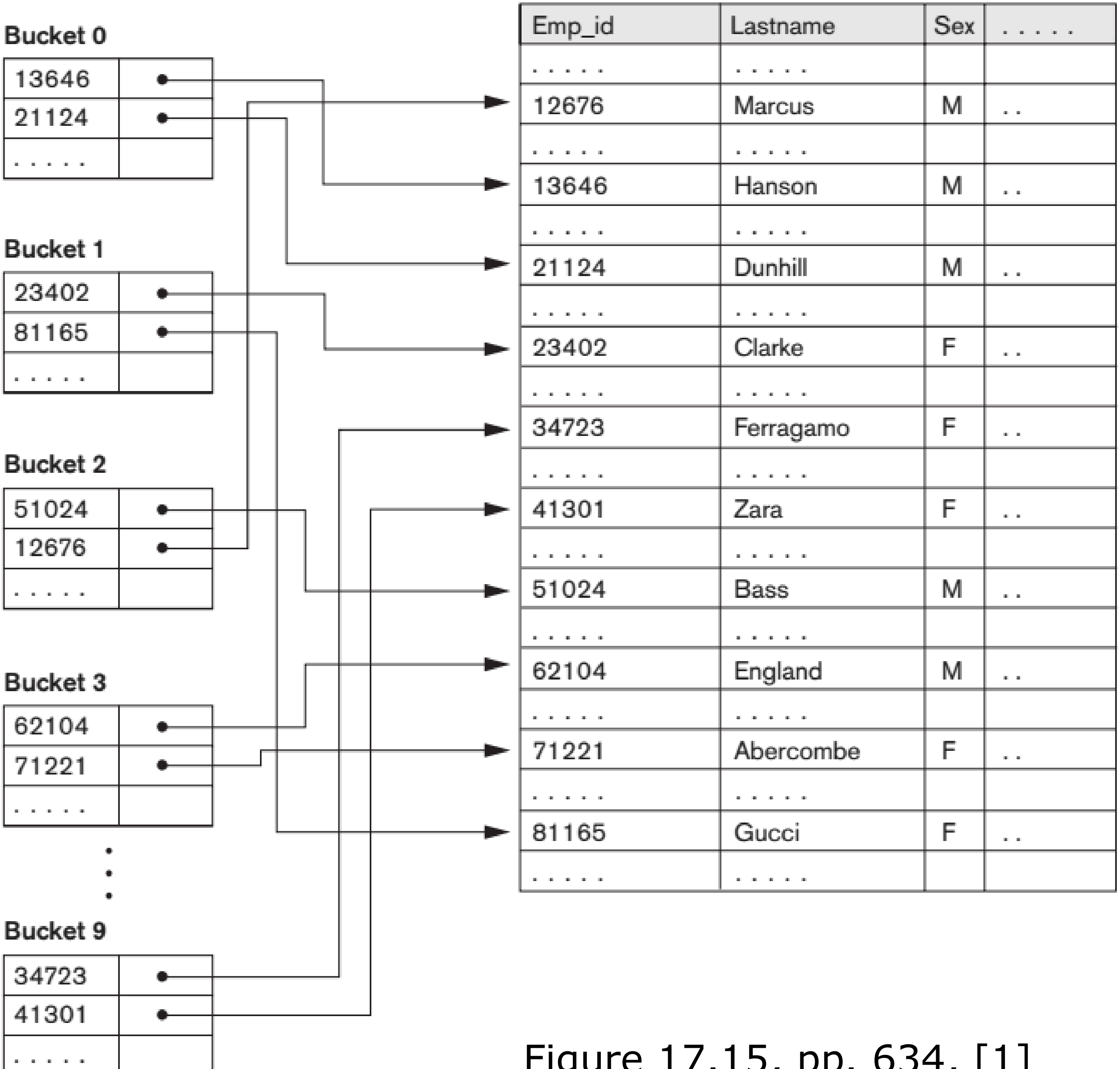


Figure 17.15, pp. 634, [1]

2.5. Other File Indexes

- Bitmap indexes
  - A bitmap index is built on **one particular value** of a field (the column in a table) with respect to all the rows (records) and is an array of bits.
    - Each bit in the bitmap corresponds to a row. If the bit is set, then the row contains the key value.
  - In a bitmap index, each indexing field value is associated with pointers to multiple rows.
  - Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion.
    - The number of distinct values of the indexed field is small compared to the number of rows.
    - The indexed table is either read-only or not subject to significant modification by DML statements.

2.5. Other File Indexes

- Bitmap indexes – Adapted examples in Oracle - E25789-01

customers Table

cust_id	cust_last_name	cust_marital_status	cust_gender
1	Kessel		M
2	Koch		F
3	Emmerson		M
4	Hardy		M
5	Gowen		M
6	Charles	single	F
7	Ingram	single	F

Sample bitmaps

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1
single	0	0	0	0	0	1	1
divorced	0	0	0	0	0	0	0

2.5. Other File Indexes

- Bitmap indexes – Adapted examples in Oracle - E25789-01

SELECT COUNT(*)
FROM customers
WHERE cust_gender = 'F'
AND cust_marital_status **IN** ('single', 'divorced');

The resulting bitmap to access the table

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1
single	0	0	0	0	0	1	1
divorced	0	0	0	0	0	0	0
single **or** divorced, **and** F	0	0	0	0	0	1	1

2.5. Other File Indexes

- Function-based indexes
  - The use of any function on a column prevents the index defined on that column from being used.
    - *Indexes are only used with some specific search conditions on indexed columns.*
  - In Oracle, a function-based index is an index such that the value that results from applying some function (expression) on a field or a collection of fields becomes the key to the index.
    - A function-based index can be either a B-tree or a bitmap index.

2.5. Other File Indexes

- Function-based indexes – Examples, pp. 637-638, [1]

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));

SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH"


CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));

SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;


CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

2.5. Other File Indexes

- Common statements for index creation

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ] ;

CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;
```

- UNIQUE is used to guarantee that no two rows of a table have duplicate values in the key column or column.
- CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute.
- Specifying CLUSTER on a key (unique) attribute would create some variation of a primary index, whereas specifying CLUSTER on a nonkey (nonunique) attribute would create some variation of a clustering index.

Summary

- Indexes: additional access structures
  - Created on one or many fields of a data file, called indexing fields
    - Ordering key field => Primary indexes
    - Ordering non-key field => Clustering indexes
    - Non-ordering field => Secondary indexes
  - Also stored in index files on disk
  - Single-level vs. Multilevel indexes
    - Dynamic multilevel index structures: B-tree, B+-tree
  - Support certain search conditions
    - =, >, >=, <, <=, and “between” on indexing fields

Chapter 2: Indexing Structures for Files

Check for understandings

- **2.1.** What are indexes? Give at least three examples.
- **2.2.** What are primary, secondary, and clustering indexes? Give at least one example for each.
- **2.3.** Compare primary, secondary, and clustering indexes with each other. Which are dense and which are not? Explain the characteristics in their corresponding data file that make them dense or sparse.

Check for understandings

- **2.4.** Why can at most one primary or clustering index created on a data file, but zero or many secondary indexes? Give an example to demonstrate your answer.
- **2.5.** Distinguish between single-level indexes and multilevel indexes. Give an example to demonstrate your answer.
- **2.6.** Describe B-tree and B+-tree when they are used as secondary access structures for a data file. Distinguish between B-tree and B+-tree. Give an example for each structure.

Check for understandings

- **2.7**. Given a data file of 15 employee records in 8 blocks with *bfr* = 2 records/block:
  - These records are in ascending order of SSN values.
  - SSN is a primary key.
  - Name is a candidate key.
  - Dept is a foreign key.
- Build a **B-tree** index on each field: SSN, Name, Dept, given order p = 3. What is each index called?

SSN	Name	...	Dept
2	F	...	2
5	AA	...	5
8	U	...	4
10	C	...	2
11	L	...	2
15	EG	...	1
18	B	...	4
19	FL	...	5
21	N	...	5
23	Y	...	5
28	D	...	1
34	O	...	2
35	NM	...	3
36	T	...	5
40	P	...	7

Check for understandings

- **2.8**. Given a data file of 15 employee records in 8 blocks with *bfr* = 2 records/block:
  - These records are in ascending order of SSN values.
  - SSN is a primary key.
  - Name is a candidate key.
  - Dept is a foreign key.
- Build a **B+-tree** index on each field: SSN, Name, Dept, given orders p_{leaf} = 2 and q = 3. What is each index called?

SSN	Name	...	Dept
2	F	...	2
5	AA	...	5
8	U	...	4
10	C	...	2
11	L	...	2
15	EG	...	1
18	B	...	4
19	FL	...	5
21	N	...	5
23	Y	...	5
28	D	...	1
34	O	...	2
35	NM	...	3
36	T	...	5
40	P	...	7

Check for understandings

- **2.9**. Given a data file of 15 employee records in 8 blocks with *bfr* = 2 records/block:
  - These records are in ascending order of SSN values.
  - SSN is a primary key.
  - Name is a candidate key.
  - Dept is a foreign key.

SSN	Name	...	Dept
2	F	...	2
5	AA	...	5
8	U	...	4
10	C	...	2
11	L	...	2
15	EG	...	1
18	B	...	4
19	FL	...	5
21	N	...	5
23	Y	...	5
28	D	...	1
34	O	...	2
35	NM	...	3
36	T	...	5
40	P	...	7

- How are those trees updated if departments 2 and 7 are merged to be 10?

Check for understandings

- **2.10**. Given a data file of 15 employee records in 8 blocks with *bfr* = 2 records/block:
  - These records are in ascending order of SSN values.
  - SSN is a primary key.
  - Name is a candidate key.
  - Dept is a foreign key.

- How are those trees updated if the records of employees NM and L are removed?

SSN	Name	…	Dept
2	F	…	2
5	AA	…	5
8	U	…	4
10	C	…	2
11	L	…	2
15	EG	…	1
18	B	…	4
19	FL	…	5
21	N	…	5
23	Y	…	5
28	D	…	1
34	O	…	2
35	NM	…	3
36	T	…	5
40	P	…	7

Check for understandings

- **2.11**. Given a data file of 15 employee records in 8 blocks with *bfr* = 2 records/block:
  - These records are in ascending order of SSN values.
  - SSN is a primary key.
  - Name is a candidate key.
  - Dept is a foreign key.
- How are those trees updated if the records of employees 18 and 11 are removed?

SSN	Name	…	Dept
2	F	…	2
5	AA	…	5
8	U	…	4
10	C	…	2
11	L	…	2
15	EG	…	1
18	B	…	4
19	FL	…	5
21	N	…	5
23	Y	…	5
28	D	…	1
34	O	…	2
35	NM	…	3
36	T	…	5
40	P	…	7