Chapter 2

Indexing Structures for Files

Adapted from the slides of “Fundamentals of Database Systems” (Elmasri et al., 2011)

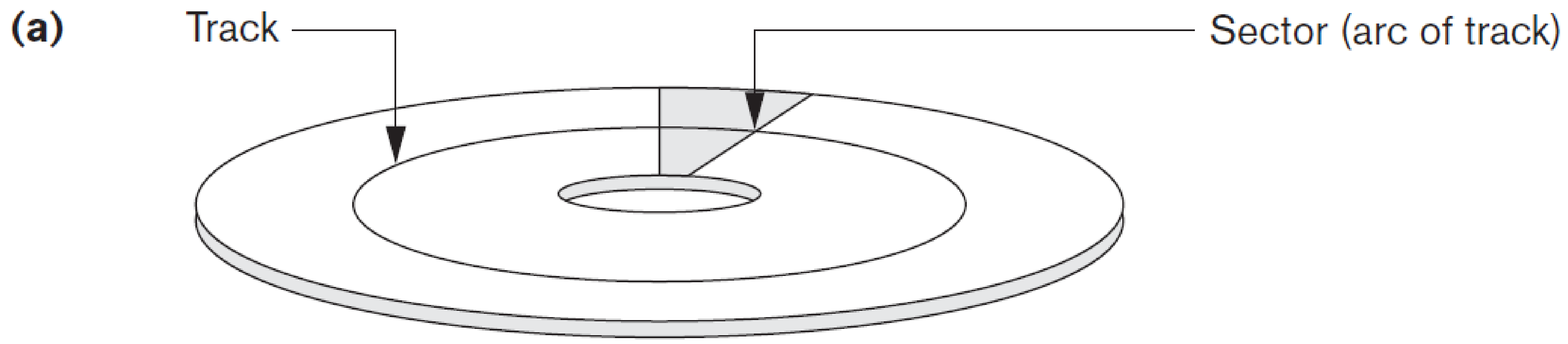
Chapter outline

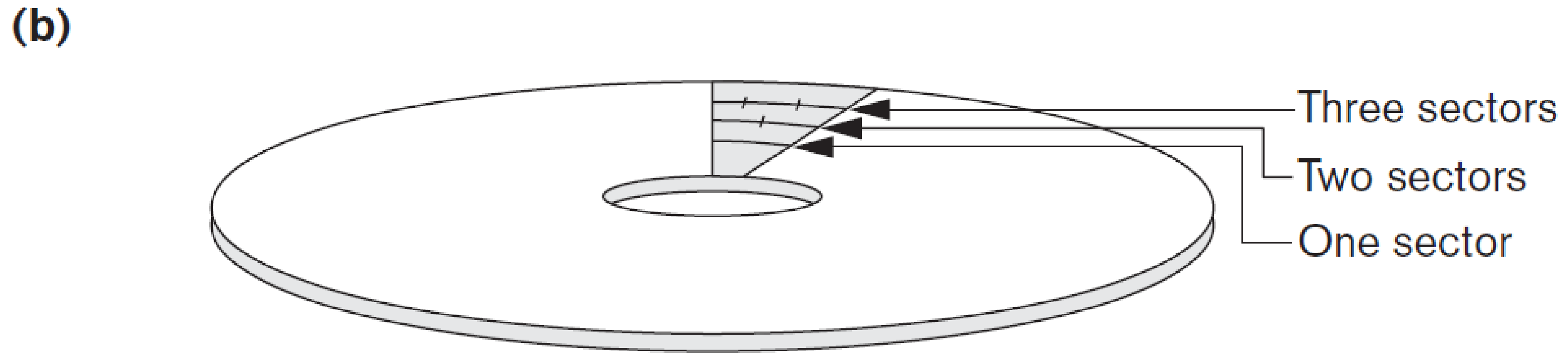
- Data storage
- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B^+-Trees
- Indexes in Oracle

Disk Storage Devices

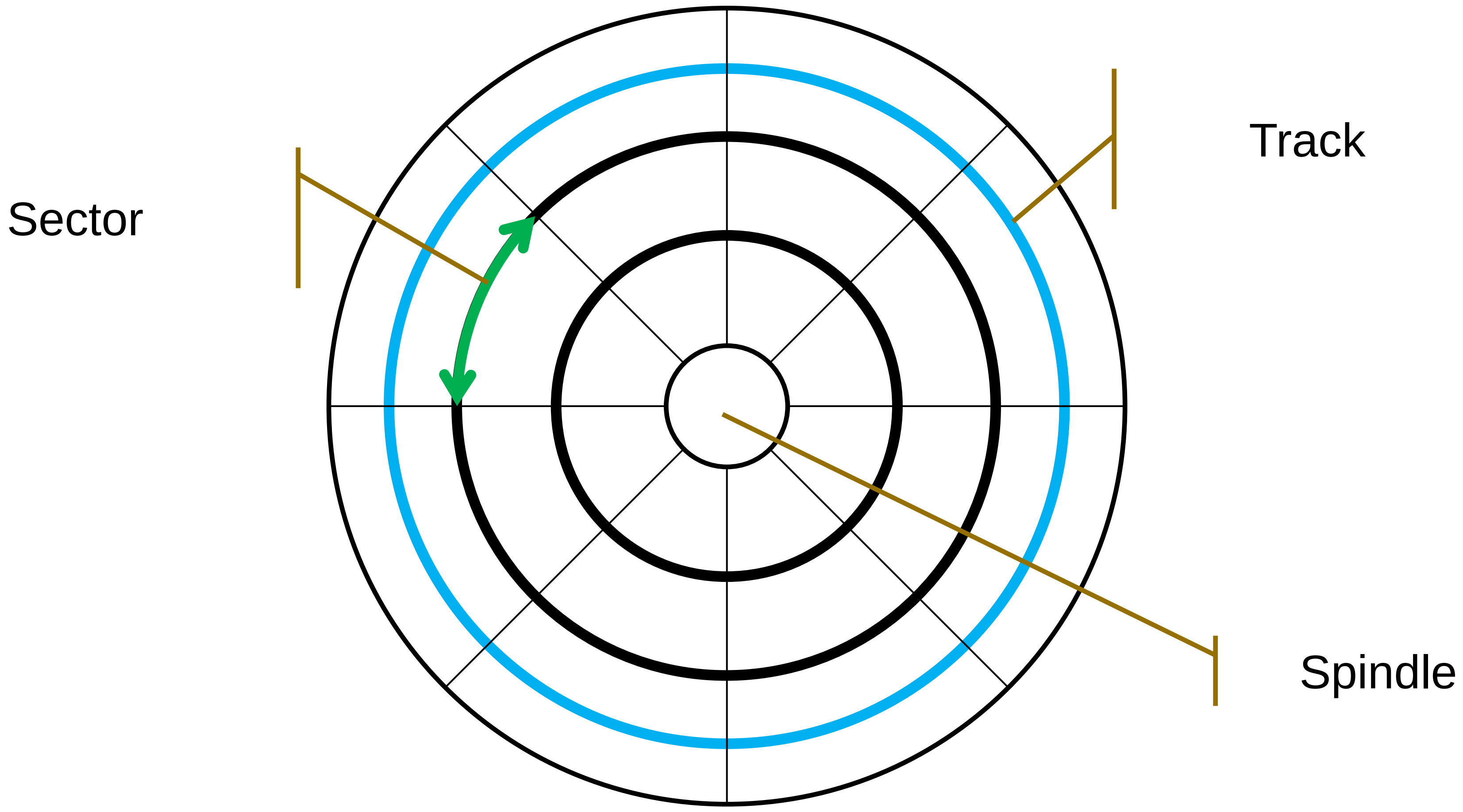
- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic disk surfaces.
- A *disk pack* contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular *tracks* on each disk *surface* .
  -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

Disk Storage Devices (cont.)





Disk Storage Devices (cont.)

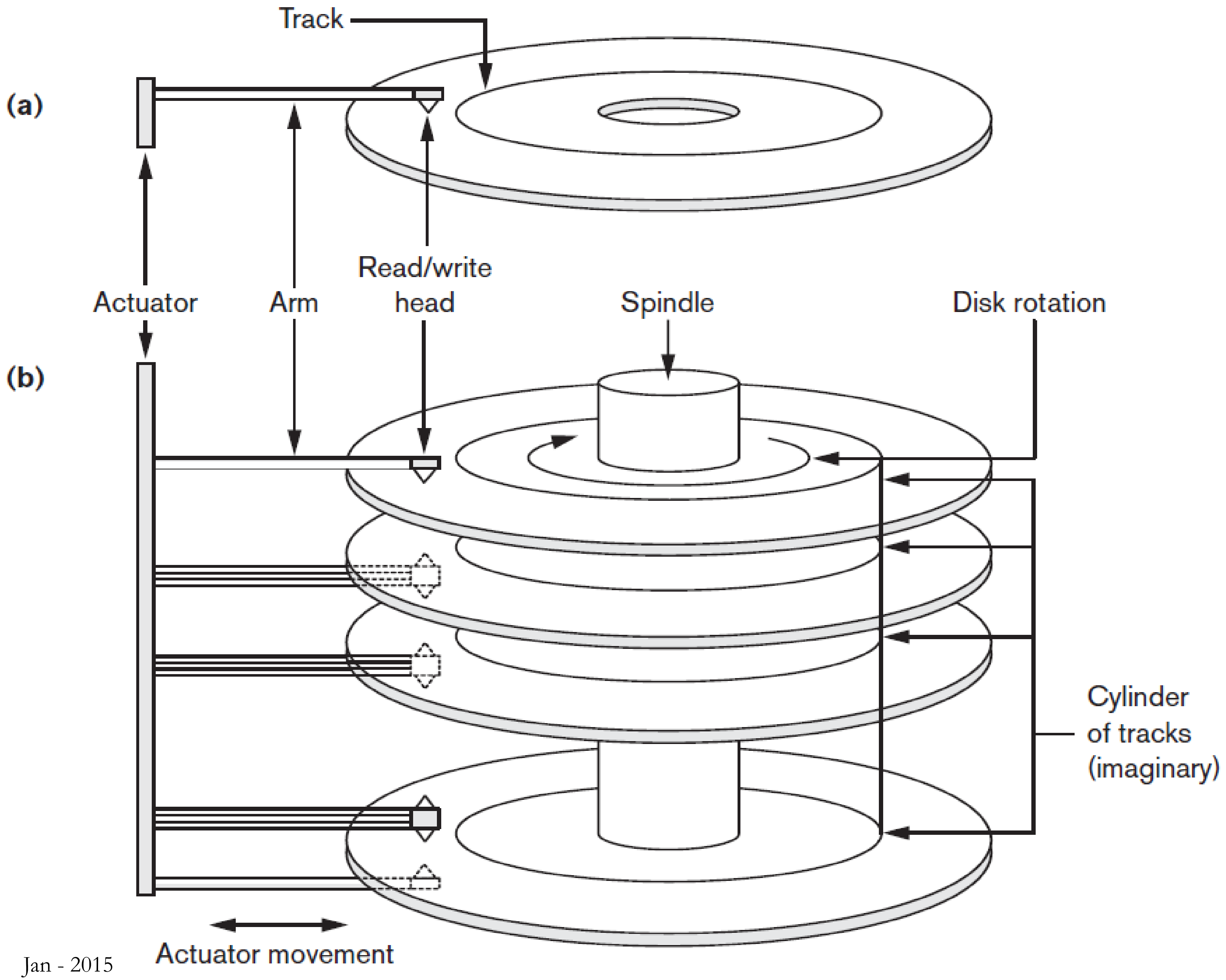


Disk Storage Devices (cont.)

- A track is divided into smaller **blocks** or **sectors.**
  - because a track usually contains a large amount of information .
- A track is divided into **blocks**.
  - 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 (cont.)

- A **read-write head** moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) **rd**.
- *Double buffering* can be used to speed up the transfer of contiguous disk blocks.

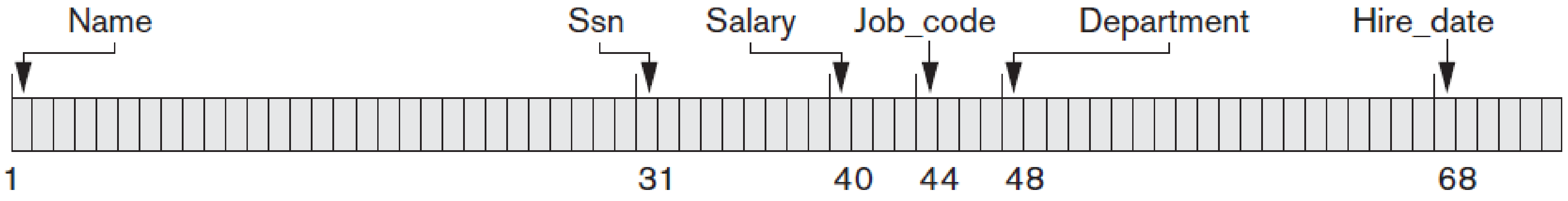

Track
(a)
Actuator
Arm
Read/write head
Spindle
Disk rotation
(b)
Cylinder of tracks (imaginary)
Actuator movement

Records

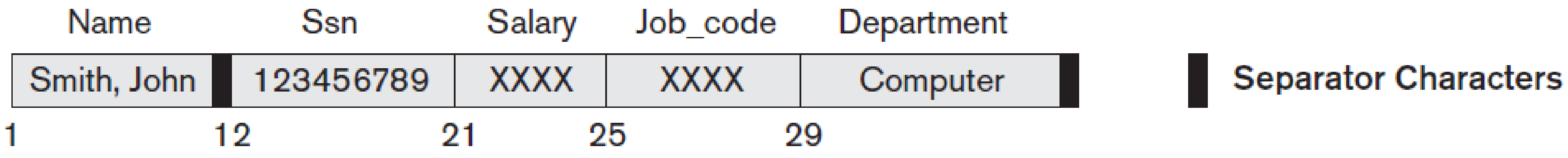
- Fixed and variable length records.
- Records contain fields which have values of a particular type.
  - E.g., amount, date, time, age.
- Fields themselves may be fixed length or variable length.
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed so that the record can be “parsed”.

Records (cont.)

(a)



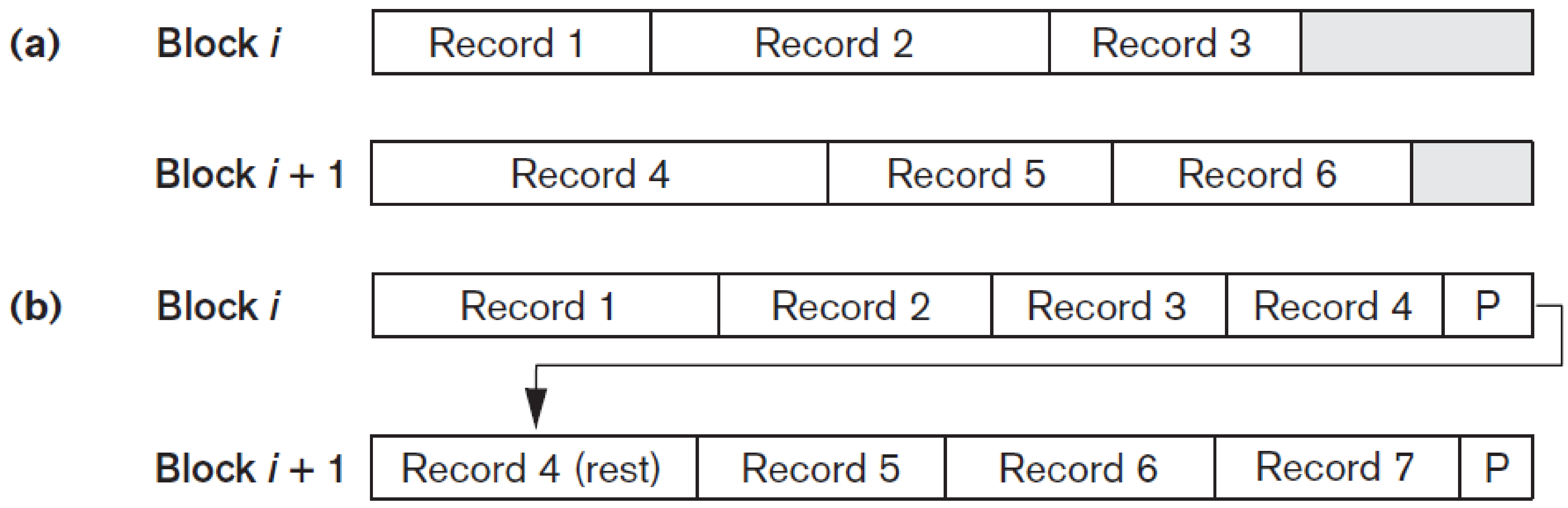
(b)



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.
- **Spanned Records:** refer to records that exceed the size of one or more blocks and hence span a number of blocks.

Blocking (cont.)



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.
- A file can have **fixed-length** records or **variable-length** records.

Files of Records (cont.)

- File records can be **unspanned** or **spanned**:
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.
  - Usually spanned blocking is used with such files.

Operation on Files

Typical file operations include:

- **OPEN:** Reads 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 (cont.)

- **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.

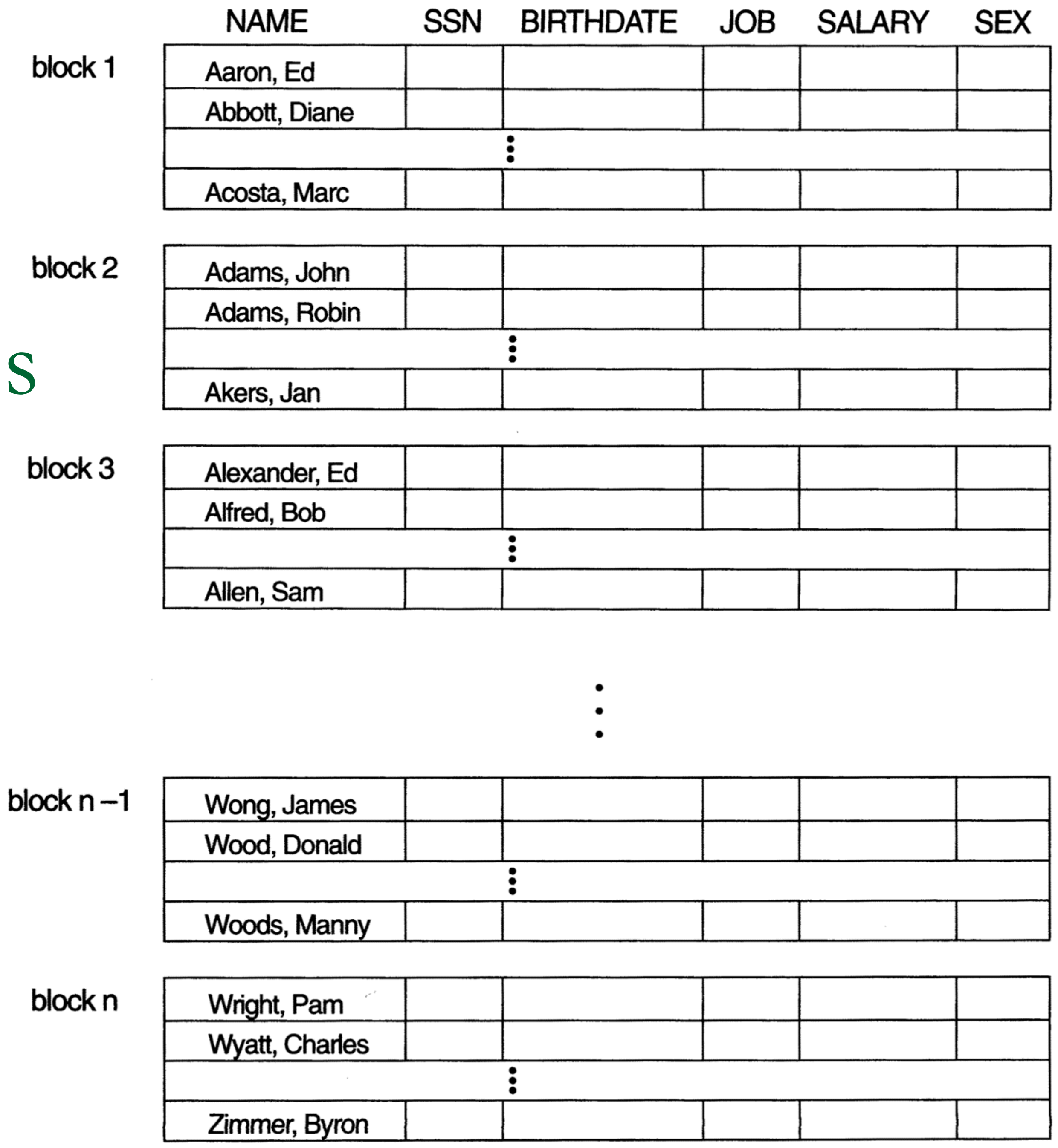
Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

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.
  - It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its *ordering field* value.
  - This requires reading and searching $\log_2$ 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 (cont.)



Average Access Times

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

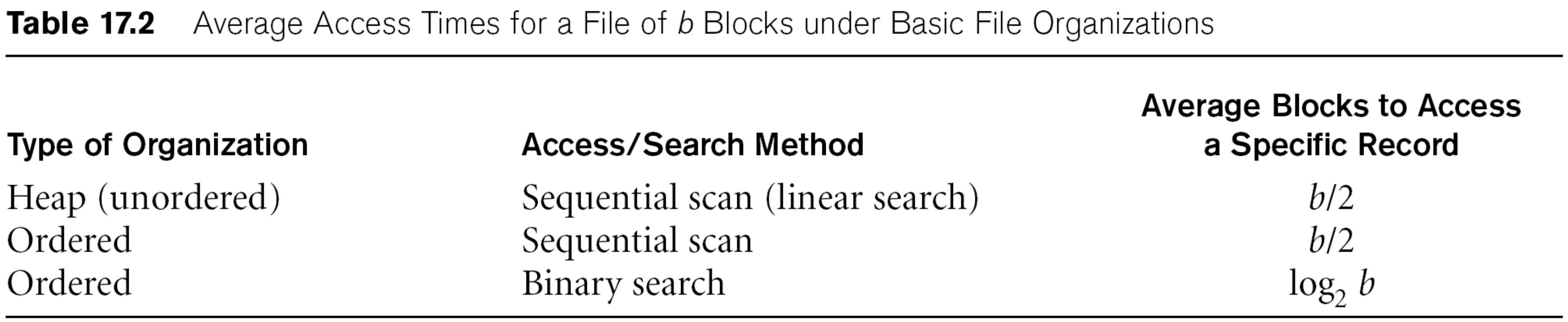


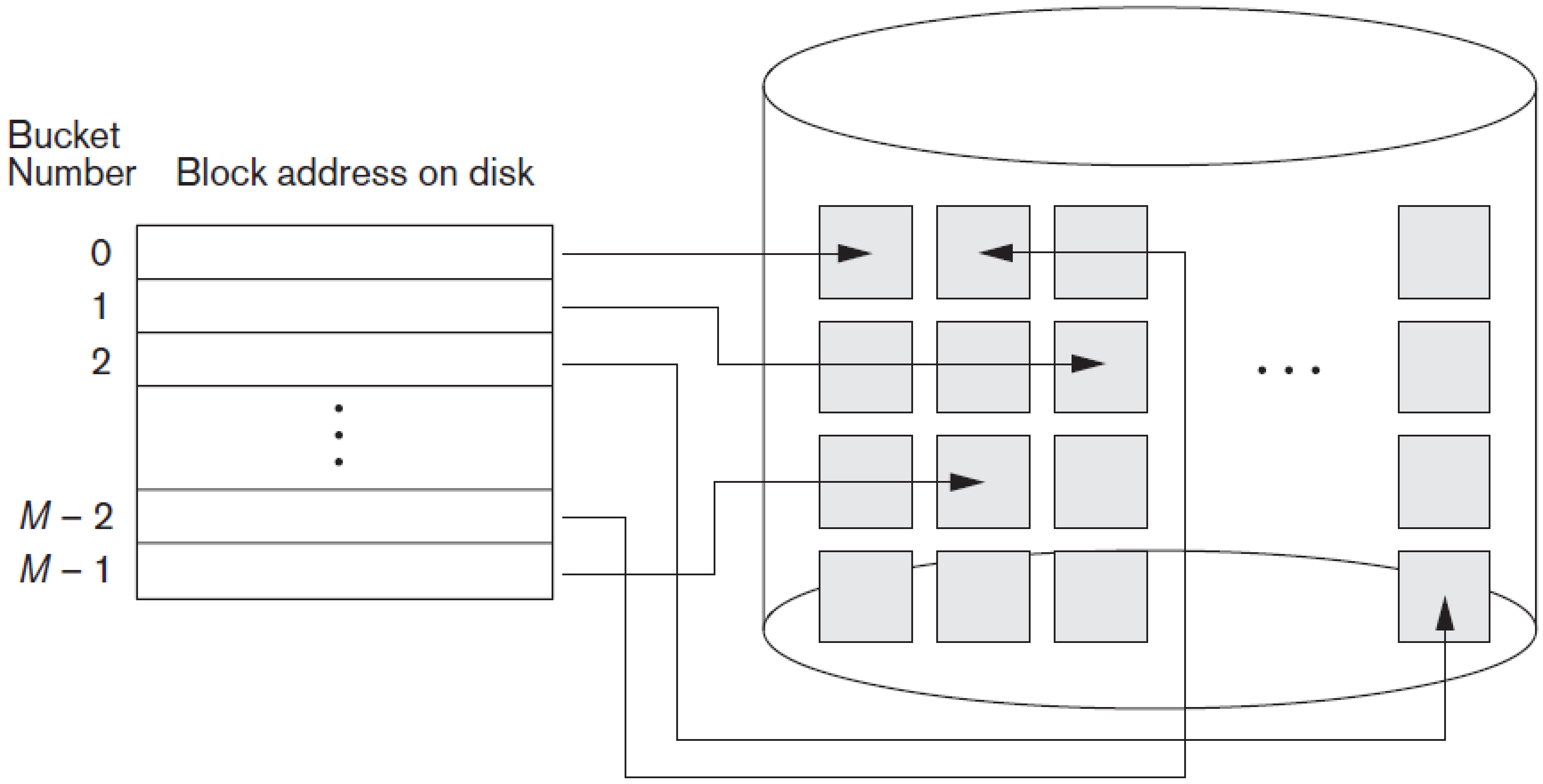
Table 17.2 Average Access Times for a File of *b* Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks 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

- Hashing for disk files is called **External Hashing.**
- The file blocks are divided into M equal-sized **buckets**, numbered bucket_0, bucket_1, ..., bucket_{M-1}.
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- 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.
  - Overflow records that hash to each bucket can be linked together

Hashed Files (cont.)



Hashed Files (cont.)

- 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.
  - h(K) = K mod 7

0	1	2	3	4	5	6
	1		3	11		6

  - Insert 8

	1	8	3	11		6

  - Insert 15

	1	8	3	11	15	6

  - Insert 13

13	1	8	3	11	15	6

Hashed Files (cont.)

- There are numerous methods for collision resolution, including the following:
  - ***Chaining:***
    - Various overflow locations are kept: extending the array with a number of overflow positions.
    - A pointer field is added to each record location.
    - 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.
    - If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Hashed Files (cont.) - Overflow handling

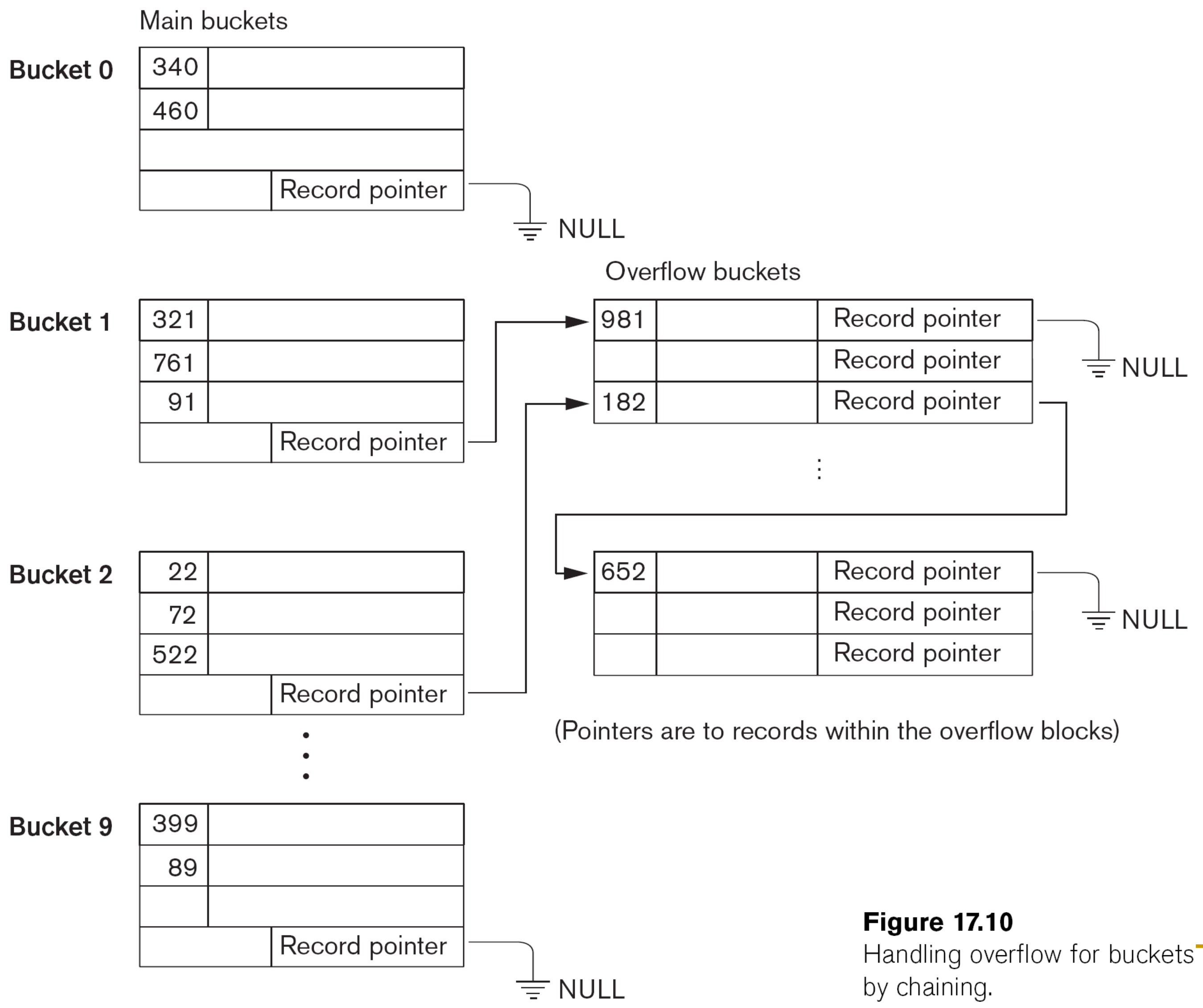


Figure 17.10
Handling overflow for buckets by chaining.

Hashed Files (cont.)

- To reduce overflow records, a hash file is typically kept 70-80% full.
- 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 external hashing:
  - Fixed number of buckets *M* is a problem if the number of records in the file grows or shrinks.
  - Ordered access on the hash key is quite inefficient (requires sorting the records).

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

Example 1: Given the following data file:
EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)

<u>Suppose that:</u>
record size R=150 bytes
block size B=512 bytes
r=30000 records
SSN Field size V_{SSN}=9 bytes, record pointer size P_R=7 bytes

<u>Then, we get:</u>
blocking factor: $bfr = \lfloor B/R \rfloor = \lfloor 512/150 \rfloor$ = 3 records/block
number of blocks needed for the file: $b = \lceil r/bfr \rceil = \lceil 30000/3 \rceil$ = 10000 blocks

For an dense index on the SSN field:
index entry size: $R_I = (V_{SSN} + P_R) = (9+7) = 16$ bytes
index blocking factor $bfr_I = \lfloor B/R_I \rfloor = \lfloor 512/16 \rfloor$ = 32 entries/block
number of blocks for index file: $b_i = \lceil r/bfr_I \rceil = (30000/32) = 938$ blocks
binary search needs $\lceil \log_2 b_i \rceil + 1 = \lceil \log_2 938 \rceil + 1 = 11$ block accesses

This is compared to an average linear search cost of:
(b/2)= 10000/2 = 5000 block accesses
If the file records are ordered, the binary search cost would be:
$\lceil \log_2 b \rceil = \lceil \log_2 10000 \rceil = 13$ block accesses

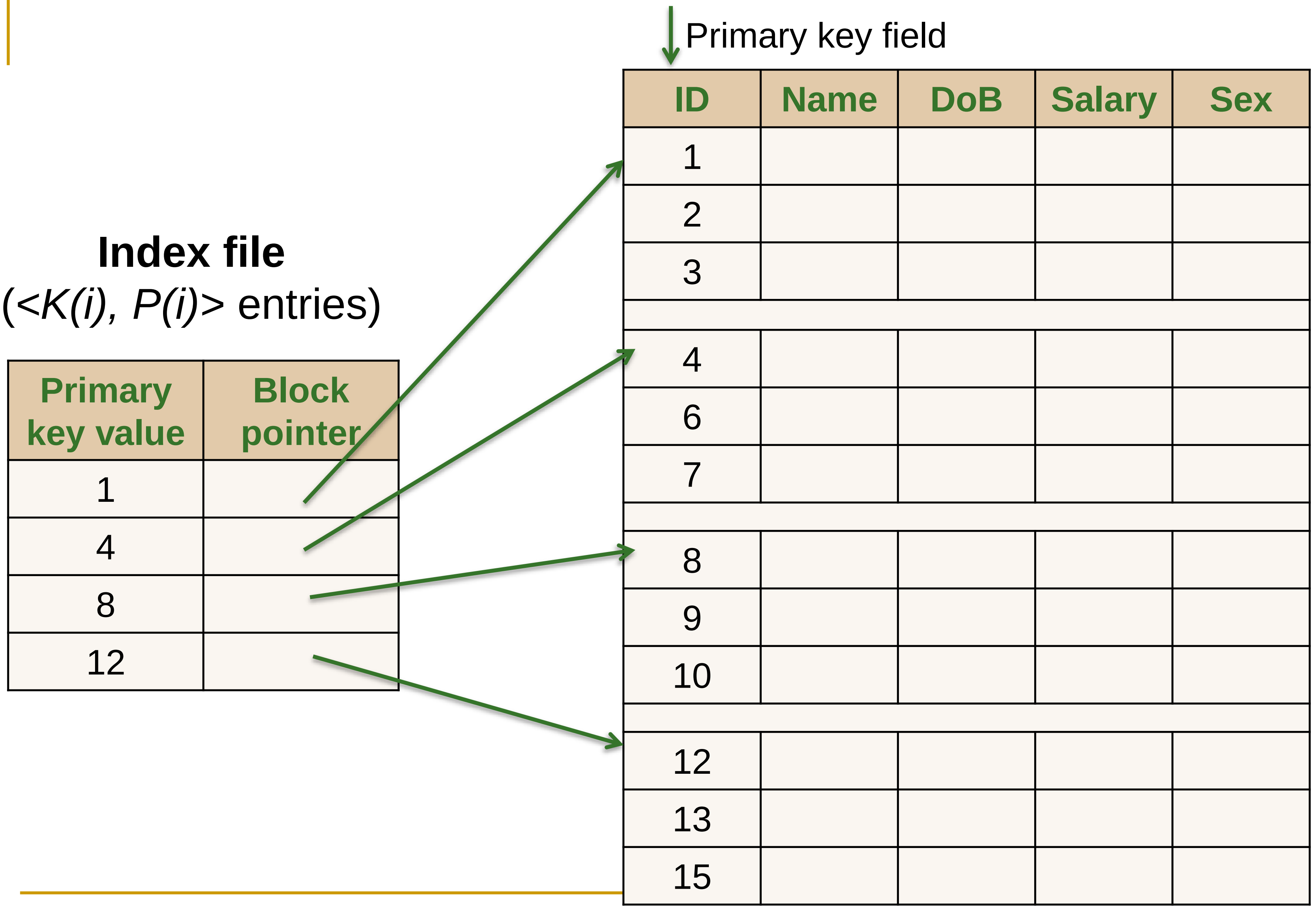
Types of Single-level Ordered Indexes

- Primary Indexes
- Clustering Indexes
- Secondary Indexes

Primary Index

- Defined on an **ordered data file.**
  - The data file is ordered on a ***key field*.**
- One index entry *for each block* in the data file
  - *First record* in the block, which is called the *block anchor*
- A similar scheme can use the *last record* in a block.

Index file
(<*K*(*i*), *P*(*i*)> entries)



Primary key value	Block pointer
1	
4	
8	
12	

ID	Name	DoB	Salary	Sex
1				
2				
3				
4				
6				
7				
8				
9				
10				
12				
13				
15				

Primary Index

- Number of index entries?
  - Number of blocks in data file.
- Dense or Nondense?
  - Nondense
- Search/ Insert/ Update/ Delete?

EXAMPLE 1 (1/2)

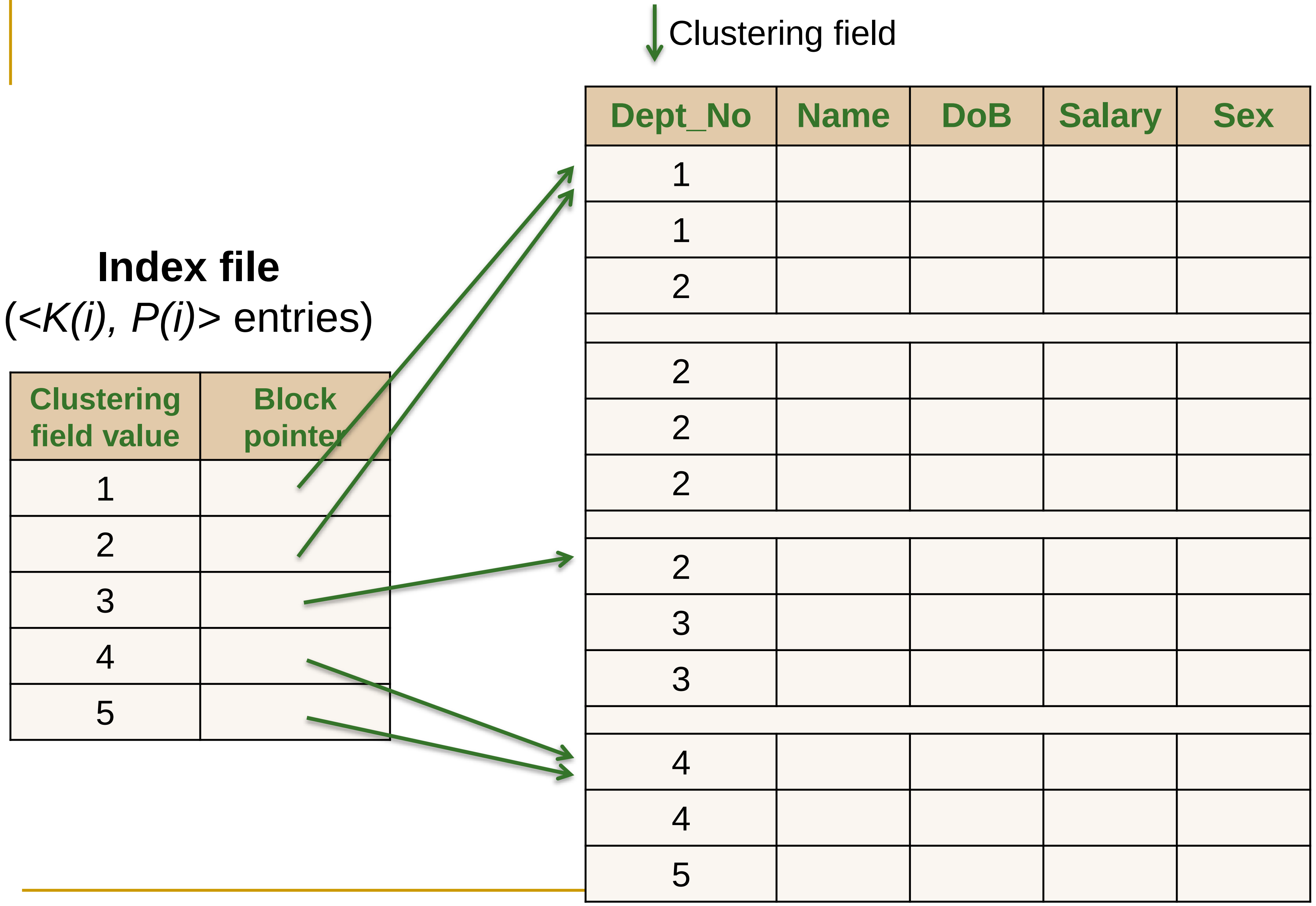
- An ordered file with r = 300,000 records
- A disk with block size B = 4,096 bytes
- File records are of fixed size and are unspanned, with record length R = 100 bytes
- $bfr = \lfloor(B/R)\rfloor = \lfloor(4{,}096/100)\rfloor$ = 40 records per block
- $b = \lceil(r/bfr)\rceil = \lceil(300{,}000/40)\rceil$ = 7,500 blocks
- A binary search on the data file would need approximately $\lceil \log2\ b \rceil = \lceil(\log2\ 7{,}500)\rceil$ = 13 block accesses

EXAMPLE 1 (2/2)

- Suppose that
  - The ordering key field of the file is V = 9 bytes long
  - A block pointer is P = 6 bytes long
- Construct a primary index
  - The size of each index entry is $R_i = (9 + 6) = 15$ bytes
  - The blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block
  - The total number of index entries r_i is equal to the number of blocks in the data fle, which is 7,500
  - The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks
- Searching
  - Binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses
  - A total of 5 + 1 = 6 block accesses

Clustering Index

- Defined on an **ordered data file.**
  - The data file is ordered on a *non-key field.*
- One index entry *each distinct value* of the field.
  - The index entry points to the *first data block* that contains records with that field value



Index file

(<*K*(*i*), *P*(*i*)> entries)

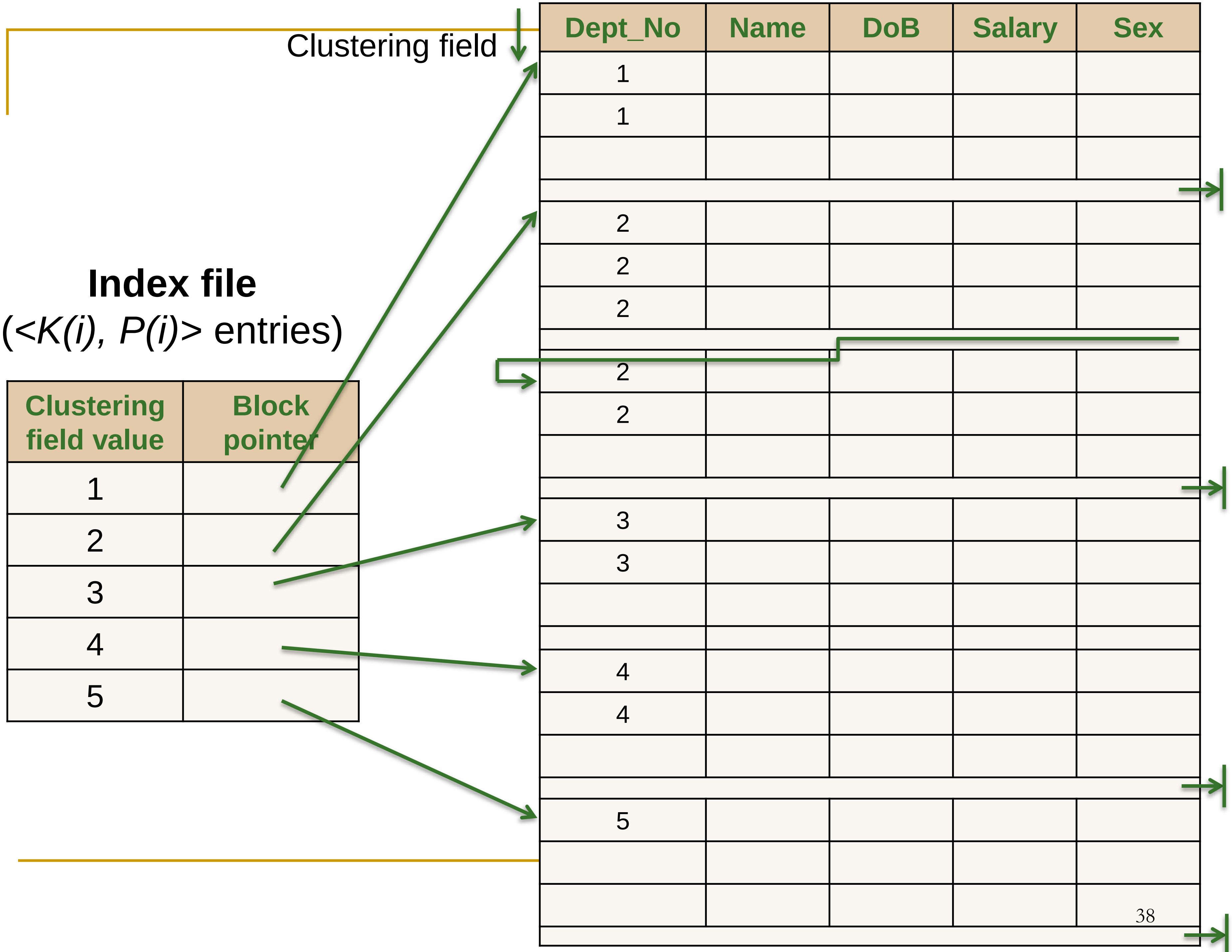
Clustering field value	Block pointer
1	
2	
3	
4	
5	

Clustering field

Dept_No	Name	DoB	Salary	Sex
1				
1				
2				
2				
2				
2				
2				
3				
3				
4				
4				
5				

Index file

(<*K*(*i*), *P*(*i*)> entries)



Clustering Index

- Number of index entries?
  - Number of distinct indexing field values in data file.
- Dense or Nondense?
  - Nondense
- Search/ Insert/ Update/ Delete?
- At most **one primary index or one clustering index but not both.**

EXAMPLE 2

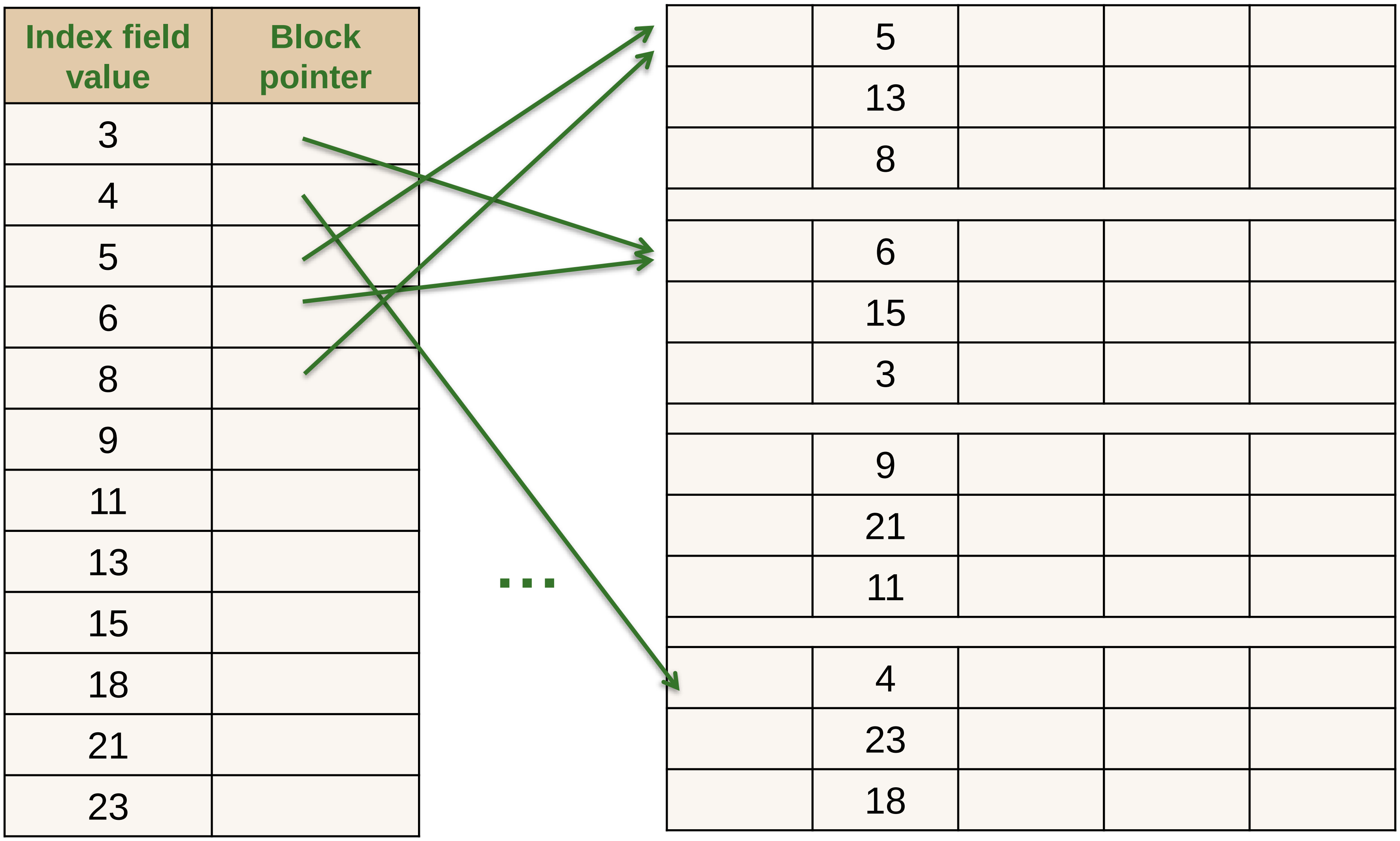
- r = 300,000 records, B = 4,096 bytes
- The file is ordered by the attribute Zipcode and there are 1,000 zip codes in the file
- R_i = 5-byte Zipcode and 6-byte block pointer
- $bfr_i = \lfloor(B/R_i)\rfloor = \lfloor(4{,}096/11)\rfloor$ = 372 index entries per block
- r_i = 1000 index entries of the clustering index
- $b_i = \lceil(r_i/bfr_i)\rceil = \lceil(1{,}000/372)\rceil$ = 3 blocks
- A binary search on the index file would need $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 3)\rceil$ = 2 block accesses

Secondary index

- A secondary index provides a secondary means of accessing a file**.**
  - The data file is unordered on indexing field*.*
- Indexing field:
  - secondary key (unique value)
  - nonkey (duplicate values)
- The index is an ordered file with two fields:
  - The first field: *indexing field.*
  - The second field: *block* pointer or *record* pointer.
- There can be ***many*** secondary indexes for the same file.

Index file
(<*K(i)*, *P(i)*> entries)

Secondary
key field



Index field value	Block pointer
3	
4	
5	
6	
8	
9	
11	
13	
15	
18	
21	
23	

	5			
	13			
	8			
	6			
	15			
	3			
	9			
	21			
	11			
	4			
	23			
	18			

Secondary index on key field

Secondary index on key field

- Number of index entries?
  - Number of record in data file
- Dense or Nondense?
  - Dense
- Search/ Insert/ Update/ Delete?

EXAMPLE 3

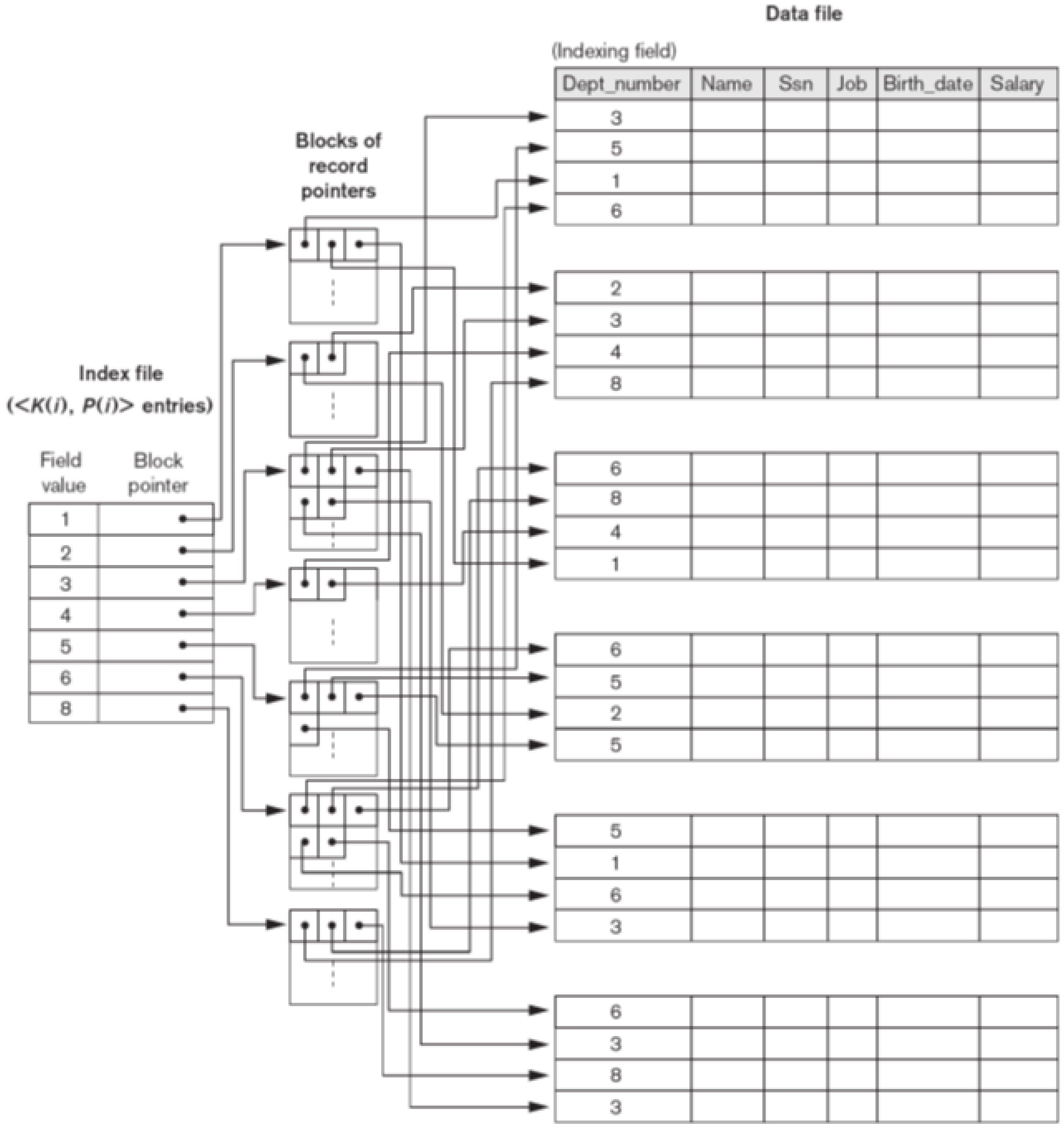
- r = 300,000, R = 100 bytes, B = 4,096 bytes, b = 7,500 blocks
- A non-ordering key field of the file that is V = 9 bytes long
- A linear search on the file would require $b/2$ = 7,500/2 = 3,750 block accesses on the average
- A secondary index
  - A block pointer is P = 6 bytes long, so each index entry is R_i = (9 + 6) = 15 bytes
  - $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor$ = 273 index entries per block
  - $b_i = \lceil (r/bfr_i) \rceil = \lceil (300{,}000/273) \rceil$ = 1,099 blocks
  - A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 1{,}099) \rceil$ = 11 block accesses
  - Total block accesses: 11 + 1 = 12

Secondary index on non-key field

- **Discussion:** Structure of Secondary index on non-key field?
- Option 1: include **duplicate index entries** with the same *K*(*i*) value - one for each record.
- Option 2: keep a **list of pointers** <*P*(*i*, 1), ..., *P*(*i*, *k*)> in the index entry for *K*(*i*).
- Option 3:
  - more commonly used.
  - one entry for each *distinct index field value* + *an* **extra level of indirection** to handle the multiple pointers.

- Secondary Index on non-key field:

  option 3



Secondary index on nonkey field

- Number of index entries?
    - Number of records in data file
    - Number of distinct index field values
- Dense or Nondense?
    - Dense/ nondense
- Search/ Insert/ Update/ Delete?

Summary of Single-level indexes

- Ordered file on indexing field?
  - Primary index
  - Clustering index
- Indexing field is Key?
  - Primary index
  - Secondary index
- Indexing field is not Key?
  - Clustering index
  - Secondary index

Summary of Single-level indexes

- Dense index?
  - Secondary index

- Nondense index?
  - Primary index
  - Clustering index
  - Secondary index

Summary of Single-level indexes

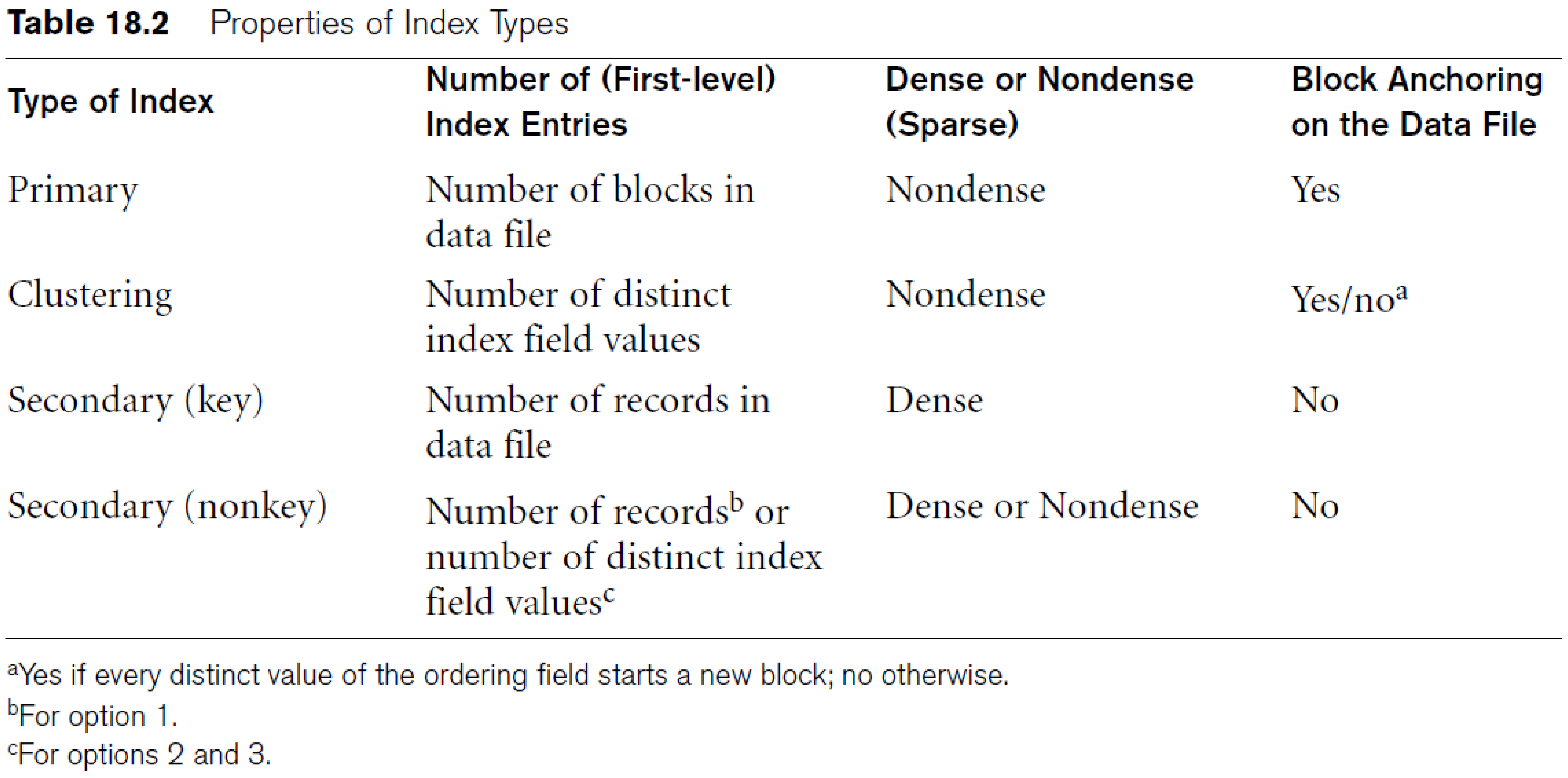


Table 18.2 Properties of Index Types

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

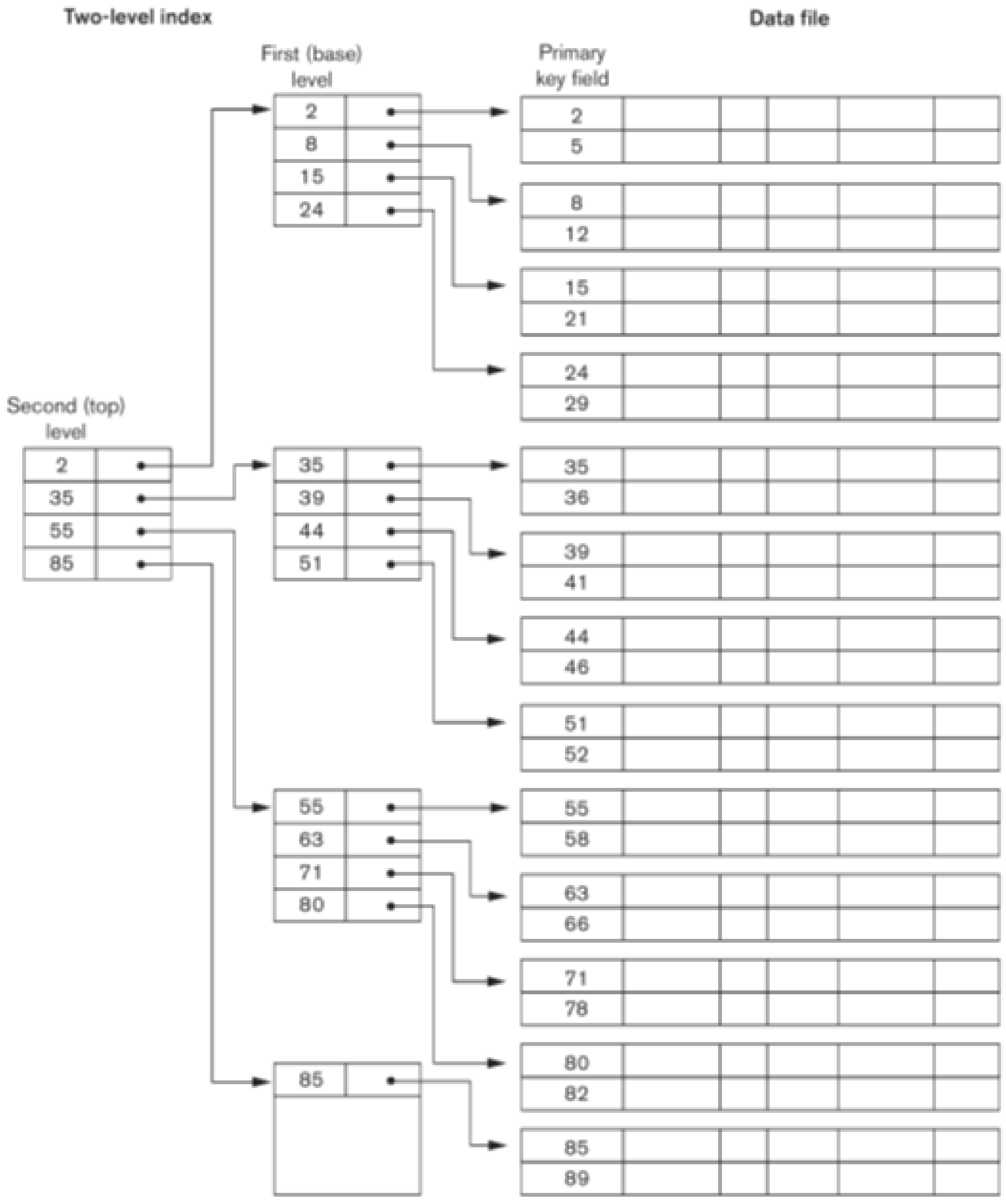
[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

Chapter outline

- Types of Single-level Ordered Indexes
    - Primary Indexes
    - Clustering Indexes
    - Secondary Indexes
- **Multilevel Indexes**
- Dynamic Multilevel Indexes Using B-Trees and B^+-Trees
- Indexes in Oracle

Multi-Level Indexes

- Because a single-level index is an ordered file, we can **create a primary index *to the index itself***.
  - 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 resembling ISAM (Indexed Sequential Access Method) organization.

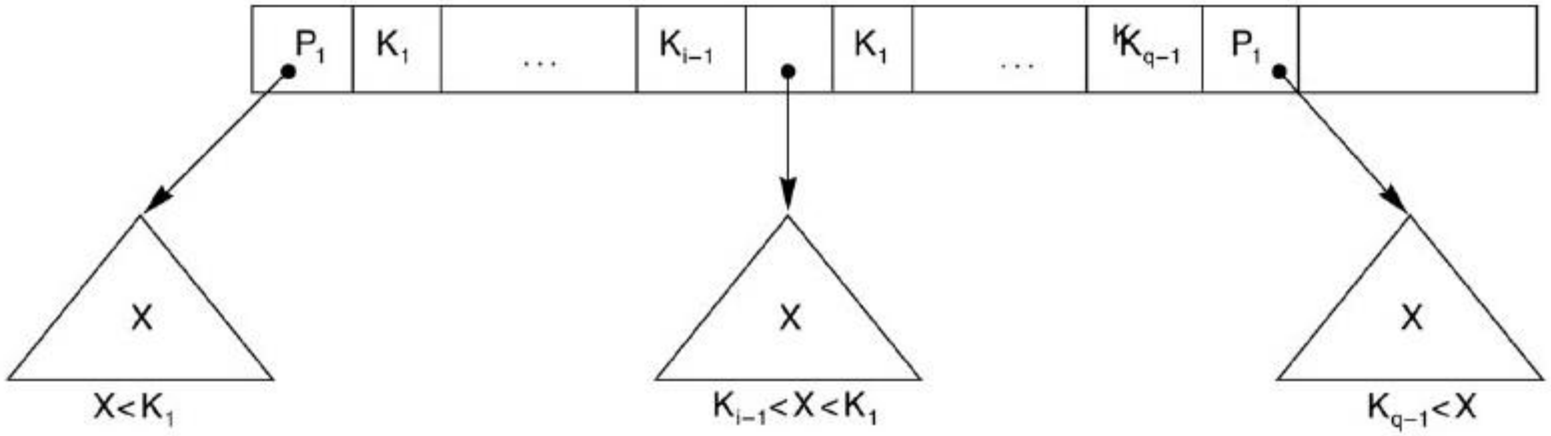
EXAMPLE 4

- *From example 3: bfr_i* = 273 index entries per block (fo) and b_1 = 1,099 blocks
  - $b_2 = \lceil (b_1/fo) \rceil = \lceil (1{,}099/273) \rceil = 5$ blocks
  - $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$ block
- Top level of the index: t = 3
- Total block accesses*:* t + 1 = 3 + 1 = 4 block accesses

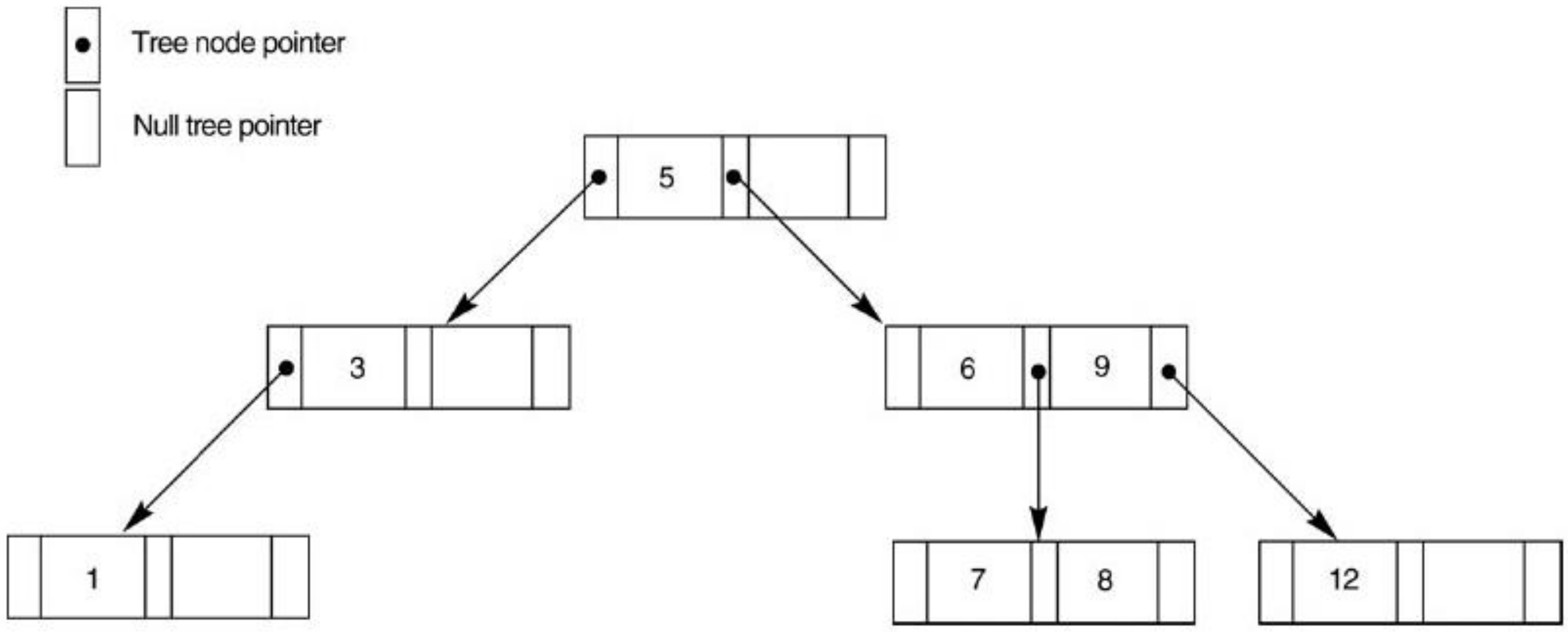
Multi-Level Indexes

- 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



A search tree of order p = 3



Chapter outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- **Dynamic Multilevel Indexes Using B-Trees and B$^+$-Trees**
- Indexes in Oracle

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.

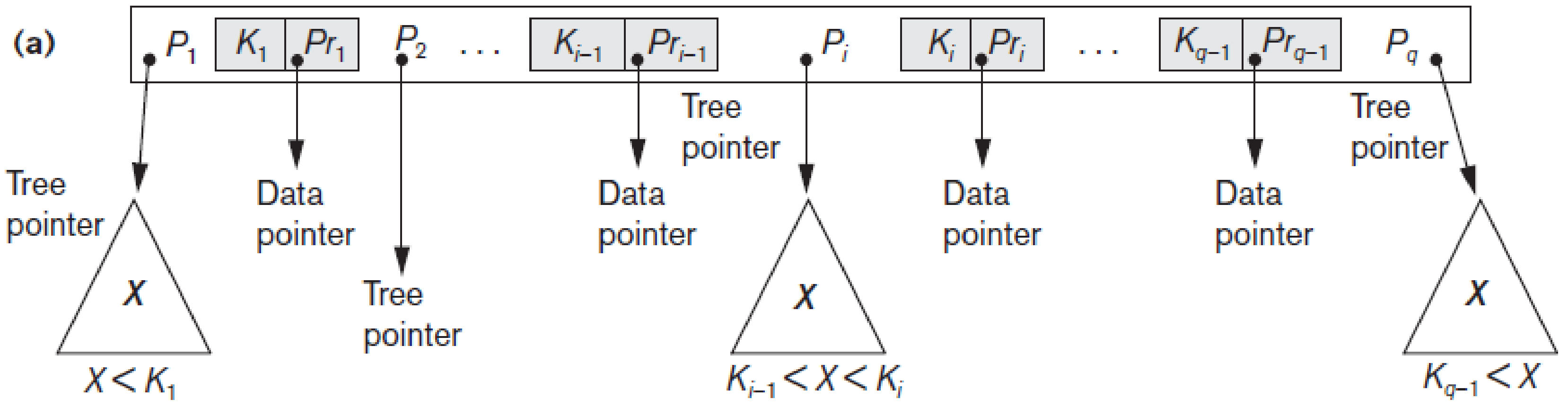
Dynamic Multilevel Indexes Using B-Trees and B^+-Trees (cont.)

- 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.

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 exist 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



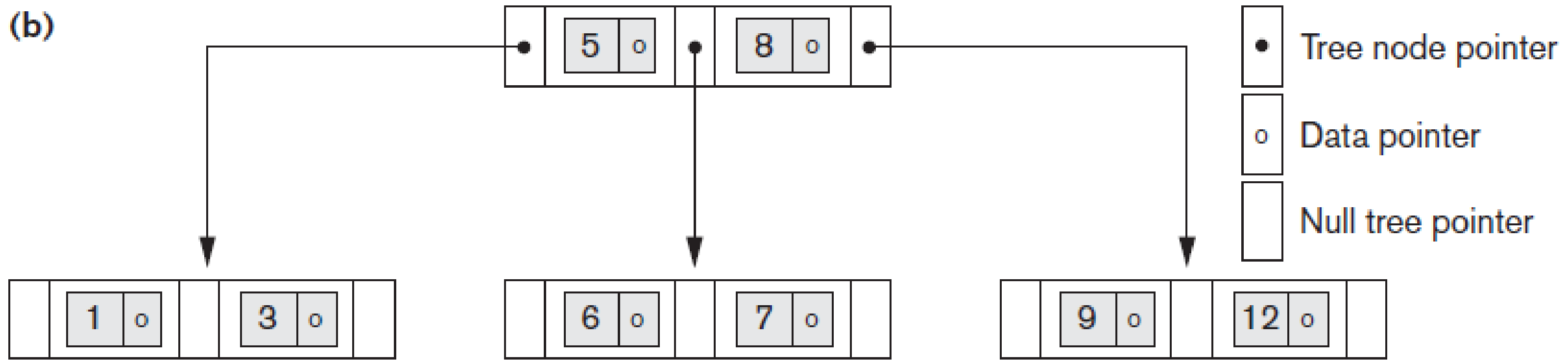


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

The Nodes of a B^+-Tree

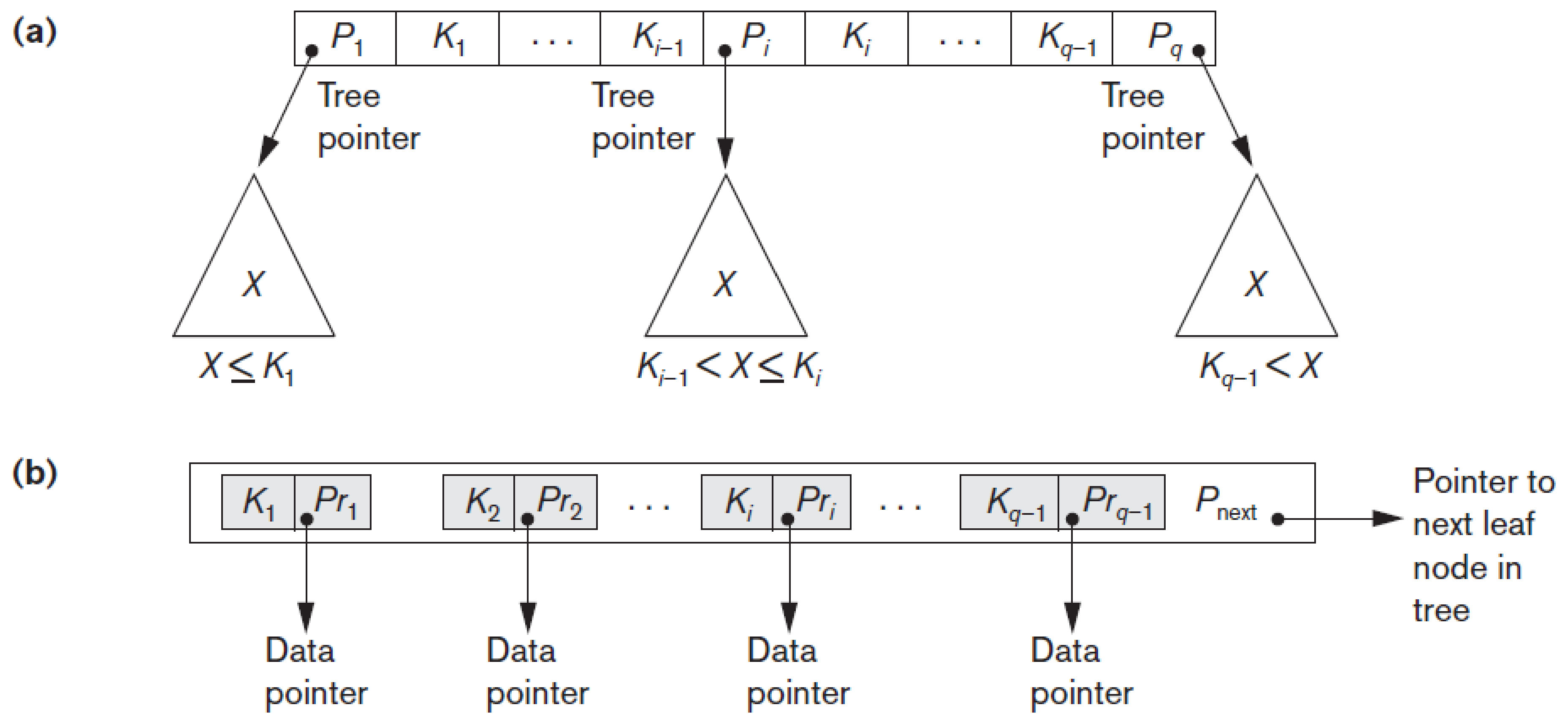


Figure 18.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values.
(b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

The Nodes of a B$^+$-Tree (cont.)

- A B$^+$-Tree of order p and p_{leaf}:
  - Each internal node:
    - Has at most p tree pointers.
    - Except the root, has at least $\lceil (^{p}/_{2}) \rceil$ tree pointer.
  - An Internal node with q pointers , q ≤ p, has q – 1 search values.
  - Each leaf node:
    - Has at most p_{leaf} data pointers.
    - has at least $\lceil (^{p_{leaf}}/_{2}) \rceil$

EXAMPLE 5: Suppose the search field is **V = 9 bytes** long, the disk block size is **B = 512** bytes, a record (data) pointer is **P_t = 7** bytes, and a block pointer is **P = 6** bytes. Each B-tree node can have at *most* p tree pointers, p – 1 data pointers, and p – 1 search key field values. These must fit into a single disk block if each B-tree node is to correspond to a disk block. Hence, we must have:

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

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

$$(22*p) \leq 528$$

We can choose to be a large value that satisfies the above inequality, which gives p = 23 (p = 24 is not chosen because of additional information).

EXAMPLE 6: Suppose that search field of Example 2 is a non-ordering key field, and we construct a B-Tree on this field. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have:

$$p * 0.69 = 23 * 0.69$$

Or approximately 16 pointers and, hence, 15 search key field values. The average fan-out fo = 16. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Level	Nodes	Index entries	Pointers
Root:	1 node	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	

At each level, we calculated the number of entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size, pointer size, and search key field size, a two-level B-tree holds 3840+240+15=**4095** entries on the average; a three-level B-tree holds **65,535** entries on the average.

23*0.69 = 15.87 ≈ 16

- **EXAMPLE 7**: Calculate the order of a B^+-tree.
- Suppose that the search key field is **V=9 bytes** long, the block size is **B=512bytes**, a record pointer **is P_r=7bytes**, and a block pointer is **P=6byte**s, as in Example 3. An internal node of the B^+-tree can have up to p tree pointers and p-1 search field values; these must fit into a single block. Hence, we have:

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

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

$$\Leftrightarrow \quad 15*p \leq 512$$

- We can choose p to be the largest value satisfying the above inequality, which give **p = 34**.
- This is larger than the value of 23 for the B-Tree, resulting in a larger fan-out and more entries in each internal node of a B^+-Tree than in the corresponding B-Tree.

EXAMPLE 7 (cont.)

- The leaf nodes of B$^+$-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

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

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

$$\Leftrightarrow (16 * p_{leaf}) \leq 506$$

- If follows that each leaf node hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

- **EXAMPLE 8**: Suppose that we construct a B^+-Tree on the field of Example 4. To calculate that approximate number of entries of the B^+-Tree, we assume that each node is 69 percent full. On the average, each internal node will be have 34*0.69 ≈ 23.46 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold $0.69*p_{leaf} = 0.69*31 \approx 21.39$ or approximately 21 data record pointers. A B^+-tree will have the following average number of entries at each level:

Level	**Nodes**	**Index entries**	**Pointers**
Root	1 nodes	22 entries	23 pointers
Level 1	23	23*22 = 506	23^2=529 pointers
Level 2	529	529*22 = 11,638	23^3=12,167 pointers
Leaf level	12,167	12,167 *21 = 255,507	

- For the block size, pointer size, and search field size given above, a three-level B^+-tree holds up to **255,507** record pointers, on the average. Compare this to the **65,535** entries for the corresponding B-tree in Example 3.

B$^+$-Tree: Insert entry

- Insert new entry at leaf node.
- If leaf node is full: overflows and must be split.
  - Create a new node.
  - The first $j = \lceil ((p_{leaf} + 1)/2) \rceil$ entries are kept in the original node.
  - The remaining entries are moved to the new node.
  - The jth search value is replicated in the parent internal node in the correct sequence.
  - An extra pointer to the new node is created in the parent.

B^+-Tree: Insert entry (cont.)

- If the parent internal node is full: overflow and must be split.
  - The jth $(j = \lfloor((p+1)/2)\rfloor)$ search value is move to the parent.
  - The first j – 1 entries are kept.
  - The remaining entries (from j+1 to the end) is hold in a new internal node.
- This splitting can propagate all the way to create a new root node

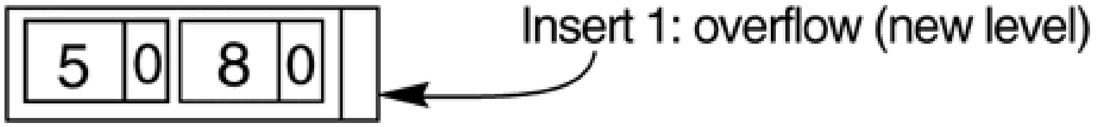
  → new level for the B^+-tree

Example of insertion in B^+-tree

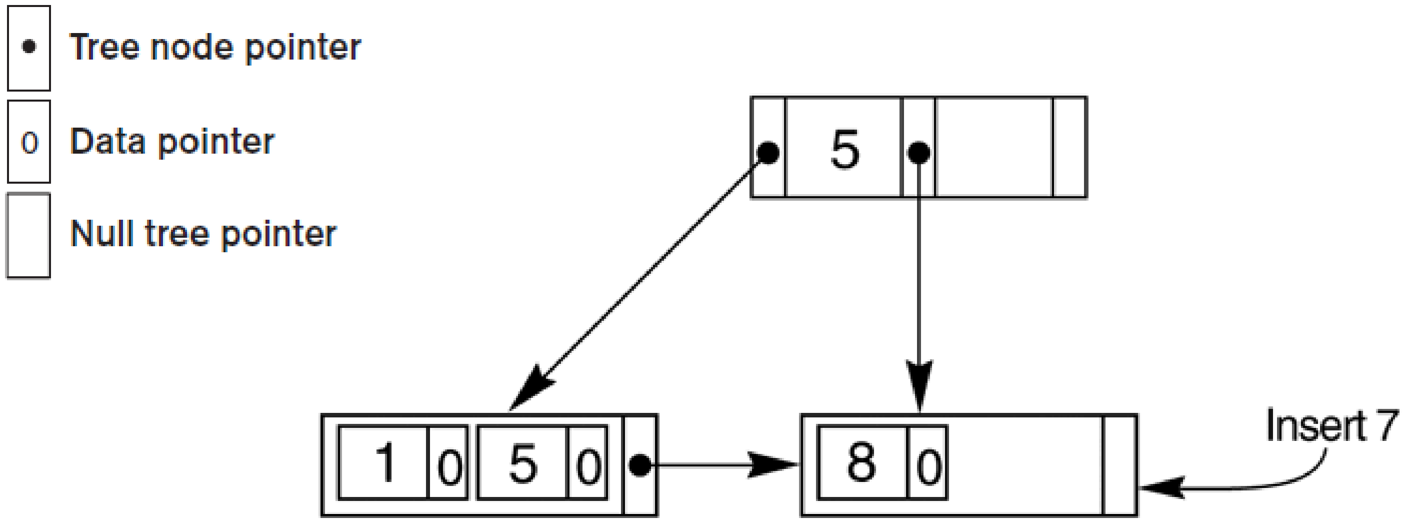
●	Tree node pointer
0	Data pointer
☐	Null tree pointer

p = 3 and p_{leaf} = 2

Insertion Sequence: 8, 5, 1, 7, 3, 12, 9, 6



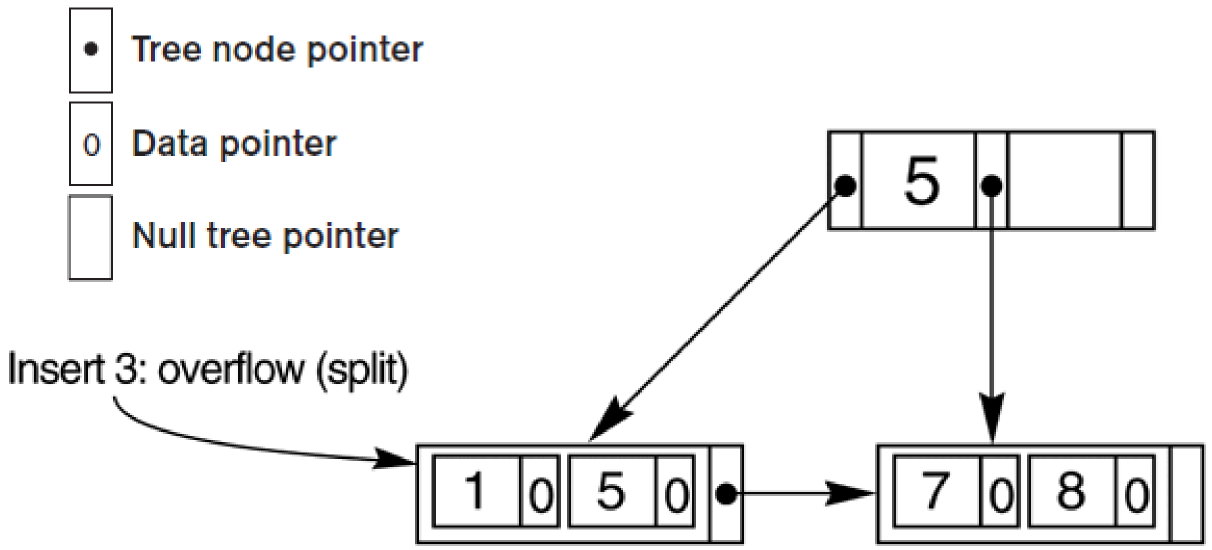
Example of insertion in B^+-tree (cont.)



p = 3 and p_{leaf} = 2

Insertion Sequence: *8, 5, 1,* 7, 3, 12, 9, 6

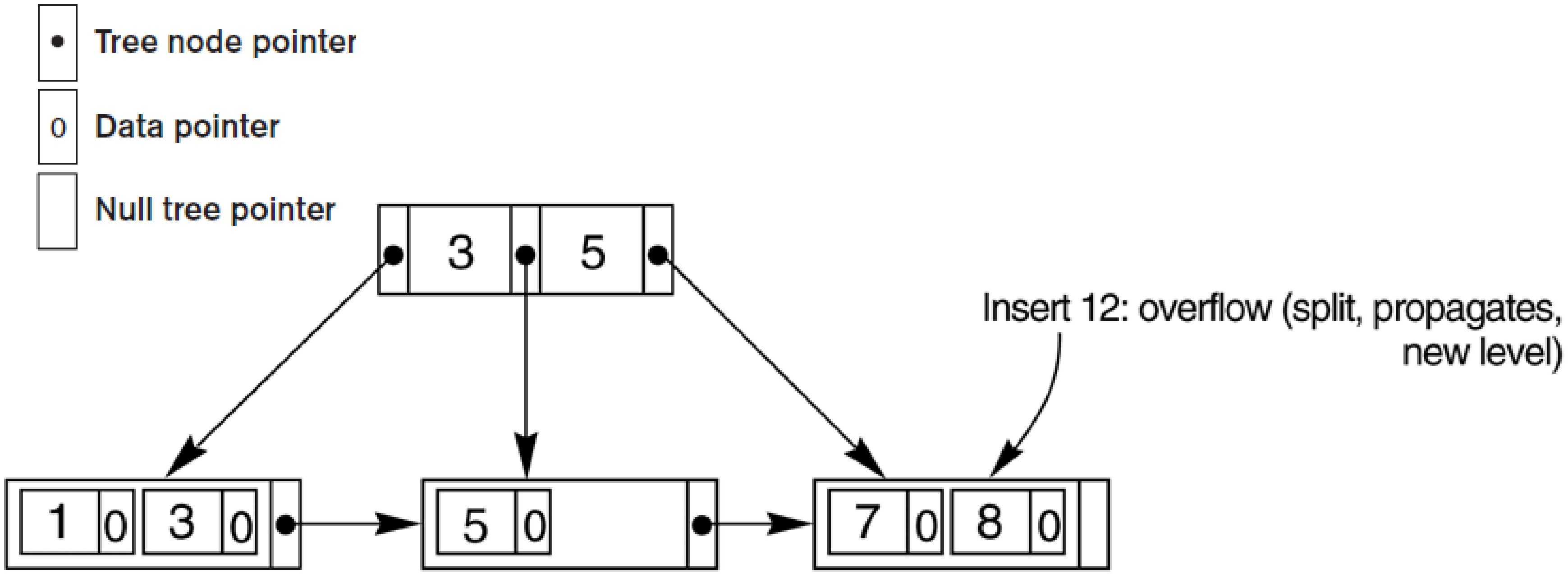
Example of insertion in B$^+$-tree (cont.)



p = 3 and p_{leaf} = 2

Insertion Sequence: *8, 5, 1, 7,* 3, 12, 9, 6

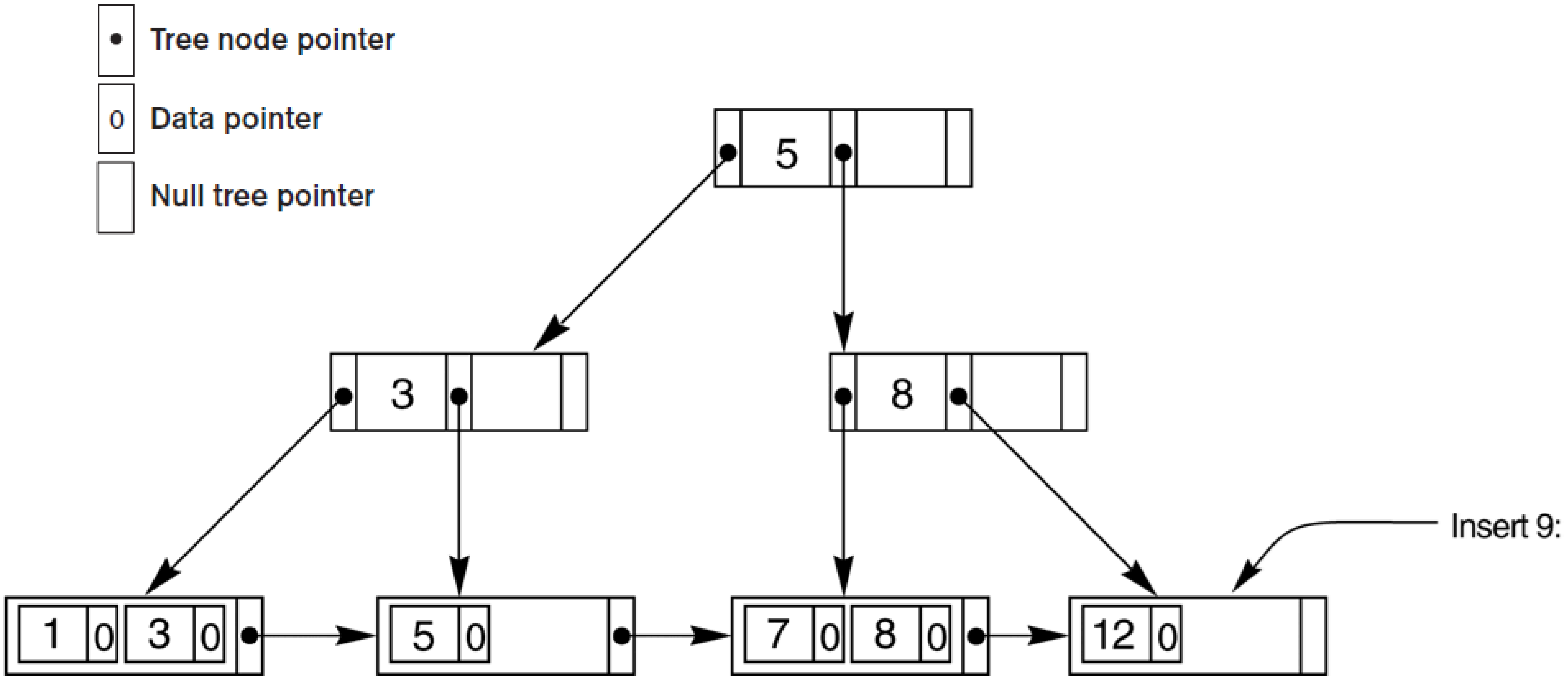
Example of insertion in B$^+$-tree (cont.)



p = 3 and p_{leaf} = 2

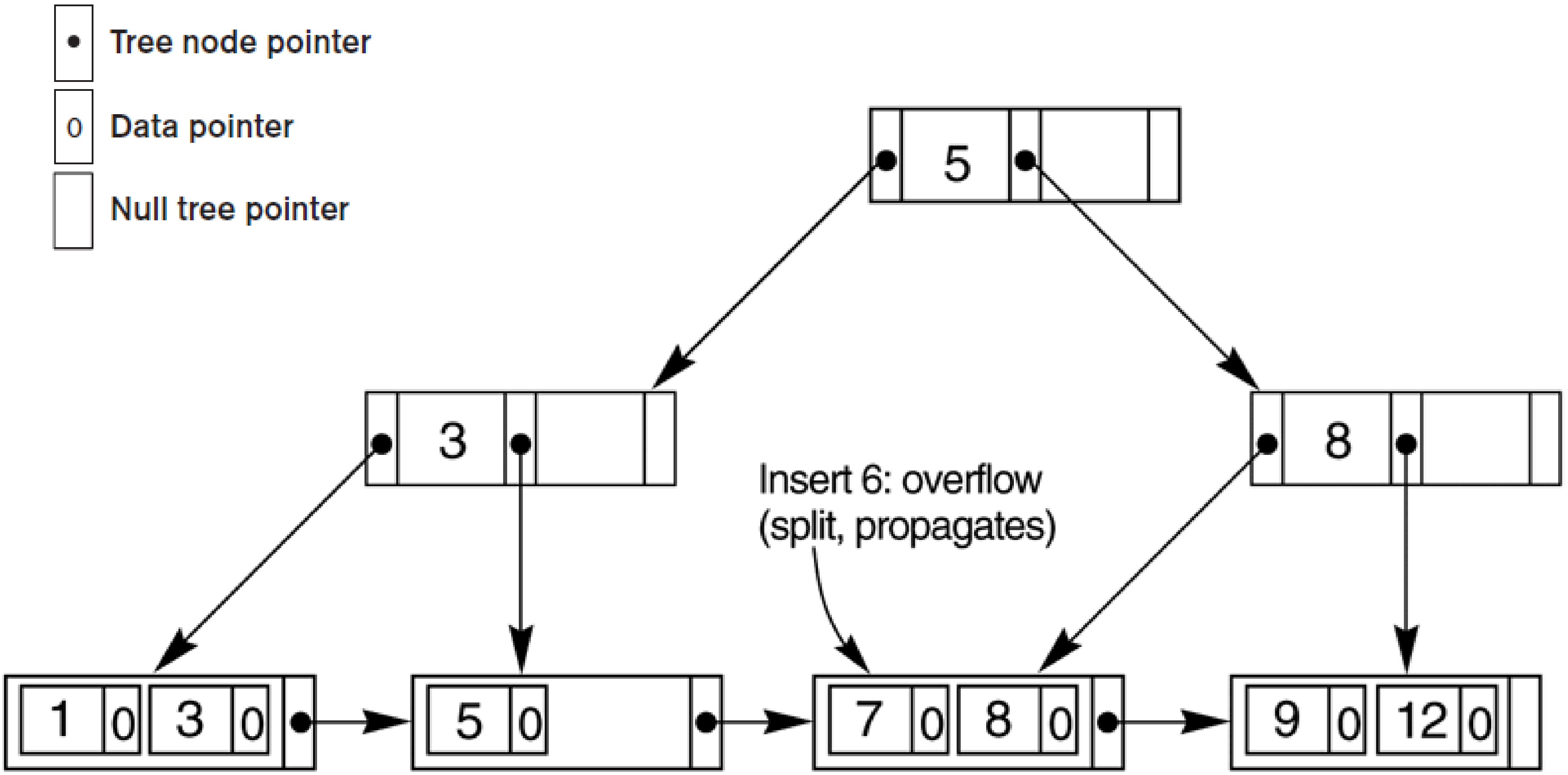
Insertion Sequence: *8, 5, 1, 7, 3,* 12, 9, 6

Example of insertion in B^+-tree (cont.)



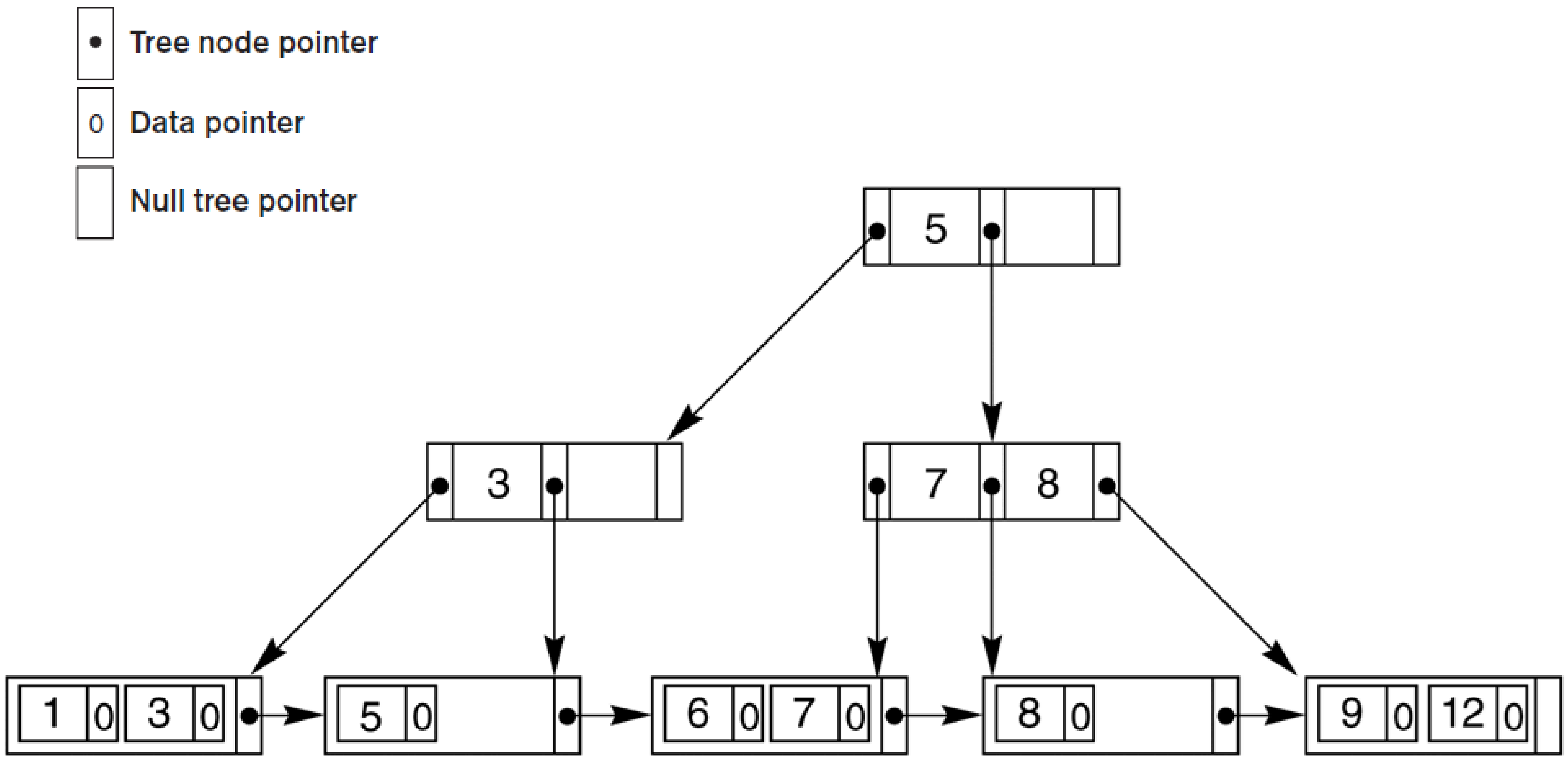
p = 3 and p_{leaf} = 2 **Insertion Sequence:** *8, 5, 1, 7, 3, 12,* 9, 6

Example of insertion in B$^+$-tree (cont.)



p = 3 and p_{leaf} = 2 **Insertion Sequence:** *8, 5, 1, 7, 3, 12, 9,* 6

Example of insertion in B^+-tree (cont.)



$p = 3$ and $p_{leaf} = 2$ **Insertion Sequence:** *8, 5, 1, 7, 3, 12, 9, 6*

B^+-Tree: Delete entry

- Remove the entry from the leaf node.
- If it happens to occur in an internal node:
  - Remove.
  - The value to its left in the leaf node must replace it in the internal node.
- Deletion may cause underflow in leaf node:
  - Try to find a sibling leaf node – a leaf node directly to the left or to the right of the node with underflow.
  - Redistribute the entries among the node and its siblings. (Common method: The left sibling first and the right sibling later)
  - If redistribution fails, the node is merged with its sibling.
  - If merge occurred, must delete entry (pointing to node and sibling) from parent node.

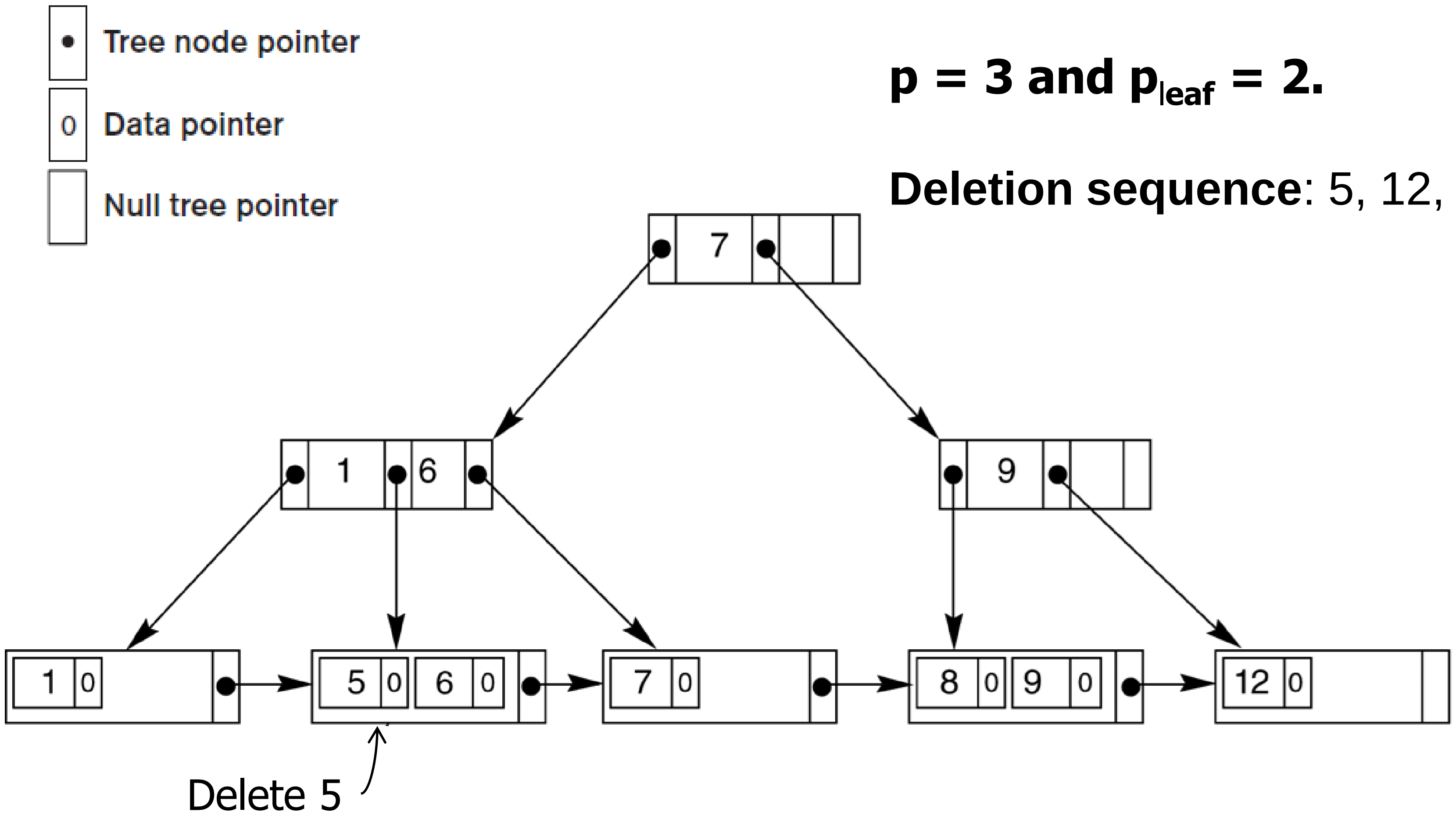
B^+-Tree: Delete entry (cont.)

- If an internal node is underflow:
  - Redistribute the entries among the node, its siblings and entry pointing to node and sibling of parent node .
  - If redistribution fails, the node is merged with its sibling and the entry pointing to node and sibling of parent node .
  - If merge occurred, must delete entry pointing to node and sibling from parent node.
  - If the root node is empty → the merged node becomes the new root node.
- Merge could propagate to root, reduce the tree levels.

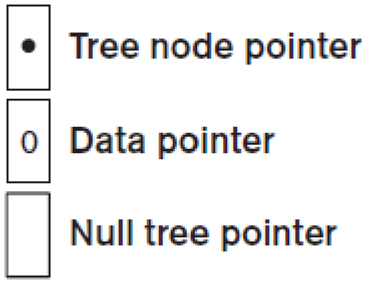
Example of deletion from B$^+$-tree.

p = 3 and p_{leaf} = 2.

Deletion sequence: 5, 12, 9

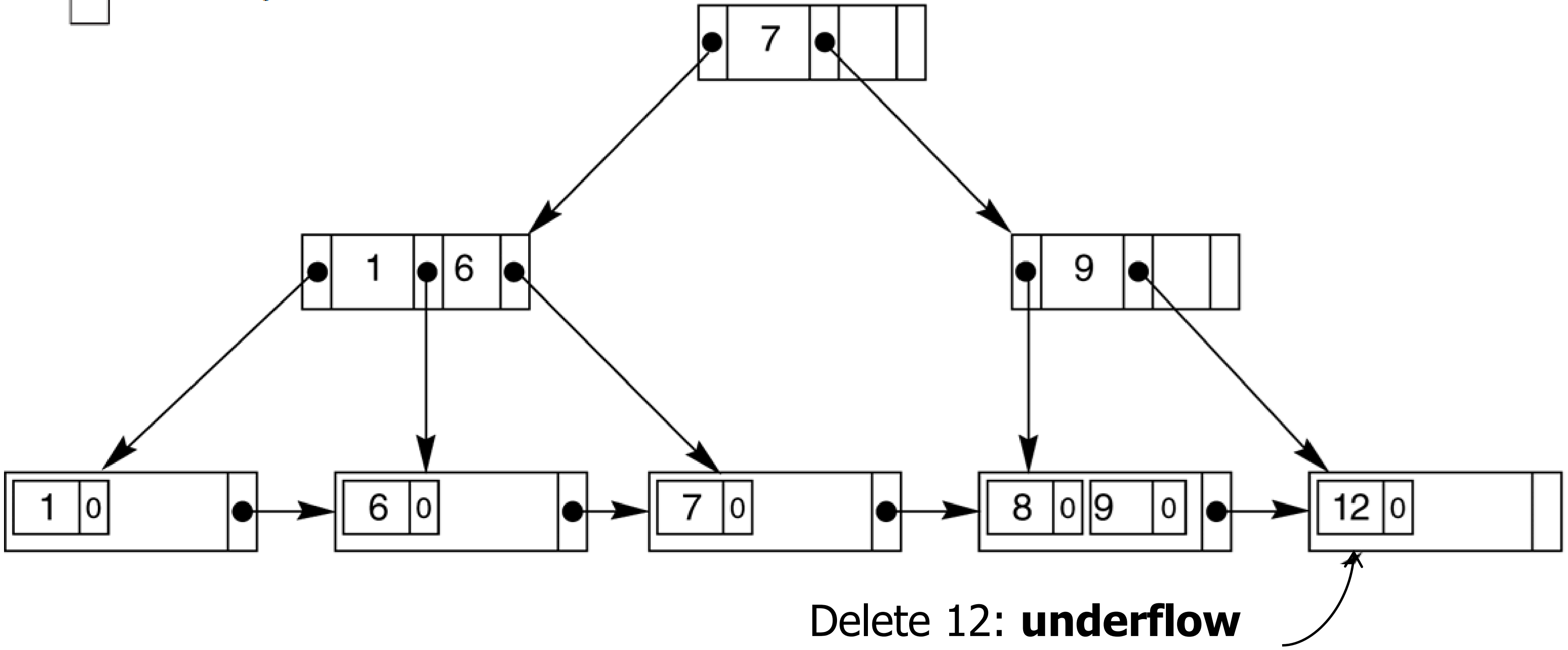


Example of deletion from B$^+$-tree (cont.)



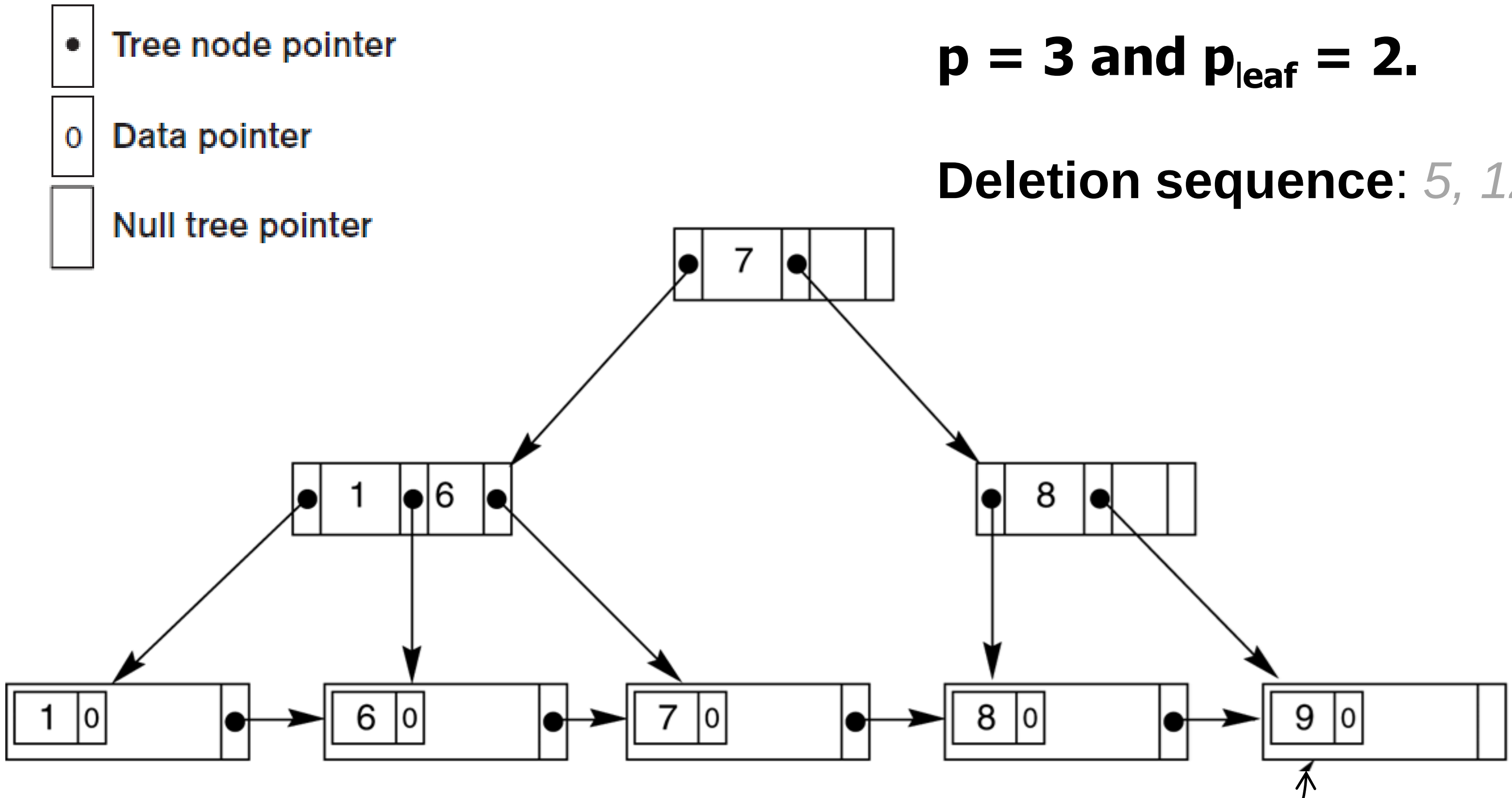
P = 3 and p$_{leaf}$ = 2.

Deletion sequence: 5, 12, 9



Delete 12: **underflow (redistribute)**

Example of deletion from B$^+$-tree (cont.)



p = 3 and p_{leaf} = 2.

Deletion sequence: *5, 12,* 9

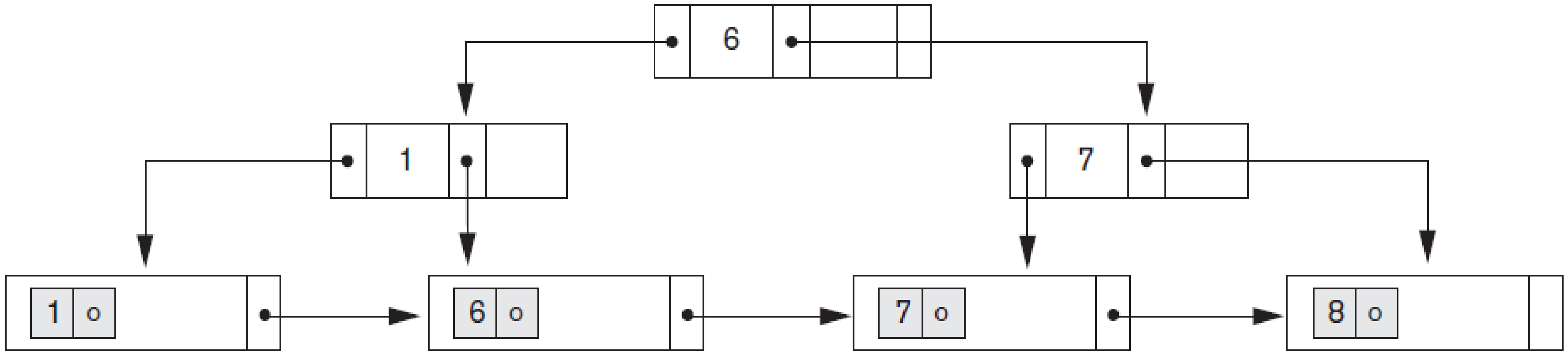
Delete 9:
Underflow (merge with left, redistribute)

Example of deletion from B+-tree (cont.)

●	Tree node pointer
o	Data pointer
☐	Null tree pointer

p = 3 and p_{leaf} = 2.

Deletion sequence: *5, 12, 9*



Notes & Suggestions

- [1], chapter 18:
  - Index on Multiple Keys
  - Other Types of Indexes
- Search, Insertion and Deletion with B-Trees.

Chapter outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- **Indexes in Oracle**

Types of Indexes

- B-tree indexes: standard index type
  - *Index-organized tables:* the data is itself the index.
  - *Reverse key indexes:* the bytes of the index key are reversed. For example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index over many blocks.
  - *Descending indexes:* This type of index stores data on a particular column or columns in descending order.
  - *B-tree cluster indexes:* is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key.

Types of Indexes (cont.)

- *Bitmap and bitmap join indexes:* an index entry uses a bitmap to point to multiple rows. A bitmap join index is a bitmap index for the join of two or more tables.
- *Function-based indexes:*
  - Includes columns that are either transformed by a function, such as the UPPER function, or included in an expression.
  - B-tree or bitmap indexes can be function-based.
- *Application domain indexes:* customized index specific to an application.

Creating Indexes

- Simple create index syntax:

  CREATE [UNIQUE | BITMAP] INDEX [schema.] <index_name>

  ON [schema.] <table_name> (column [**ASC | DESC**] [, column [**ASC | DESC**]] ...)

  [**REVERSE**];

Example of creating indexes

- **CREATE INDEX** ord_customer_ix **ON** ORDERS (customer_id);
- **CREATE INDEX** emp_name_dpt_ix ON HR.EMPLOYEES(last_name **ASC**, department_id **DESC**);
- **CREATE BITMAP INDEX** emp_gender_idx **ON** EMPLOYEES (Sex);
- **CREATE BITMAP INDEX** emp_bm_idx **ON** EMPLOYEES (JOBS.job_title) **FROM** EMPLOYEES, JOBS **WHERE** EMPLOYEES.job_id = JOBS.job_id;

Example of creating indexes (cont.)

Function-Based Indexes:

- **CREATE INDEX** emp_fname_uppercase_idx
  ON EMPLOYEES (**UPPER**(first_name));
- **SELECT** First_name, Lname
  FROM Employee **WHERE** UPPER(Lname)= "SMITH";
- **CREATE INDEX** emp_total_sal_idx
  ON EMPLOYEES (salary + (salary * commission_pct));
- **SELECT** First_name, Lname
  FROM Employee
  WHERE ((Salary*Commission_pct) + Salary) > 15000;

Guidelines for creating indexes

- Primary and unique keys *automatically have indexes*, but you might want to create an index on a foreign key.
- Create an index on any column that the query uses to join tables.
- Create an index on any column from which you search for particular values on a regular basis.
- Create an index on columns that are commonly used in ORDER BY clauses.
- Ensure that the disk and update maintenance overhead an index introduces will not be too high.

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 in Oracle

Review questions

1) Define the following terms: indexing field, primary key field, clustering field, secondary key field, block anchor, dense index, and nondense (sparse) index.
2) What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?
3) Why can we have at most one primary or clustering index on a file, but several secondary indexes?
4) How does multilevel indexing improve the efficiency of searching an index file?
5) What is the order p of a B-tree? Describe the structure of B-tree nodes.
6) What is the order p of a B+-tree? Describe the structure of both internal and leaf nodes of a B+-tree.
7) How does a B-tree differ from a B+-tree? Why is a B+-tree usually preferred as an access structure to a data file?

Type	Capacity*	Access Time	Max Bandwidth	Commodity Prices (2014)**
Main Memory- RAM	4GB–1TB	30ns	35GB/sec	$100–$20K
Flash Memory- SSD	64 GB–1TB	50µs	750MB/sec	$50–$600
Flash Memory- USB stick	4GB–512GB	100µs	50MB/sec	$2–$200
Magnetic Disk	400 GB–8TB	10ms	200MB/sec	$70–$500
Optical Storage	50GB–100GB	180ms	72MB/sec	$100
Magnetic Tape	2.5TB–8.5TB	10s–80s	40–250MB/sec	$2.5K–$30K
Tape jukebox	25TB–2,100,000TB	10s–80s	250MB/sec–1.2PB/sec	$3K–$1M+