

---

# Chapter 8:

# Data Storage, Indexing Structures for Files

---



---

# Contents

## **1 Data Storage**

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

## **2 Indexing Structures for Files**

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

2.4 Indexes in Oracle

---

---

# Contents

## **1 Data Storage**

### **1.1 Disk Storage Devices**

#### 1.2 Files of Records

#### 1.3 Operations on Files

#### 1.4 Unordered Files & Ordered Files & Hashed Files

#### 1.5 RAID Technology

## **2 Indexing Structures for Files**

### 2.1 Types of Single-level Ordered Indexes

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### 2.4 Indexes in Oracle

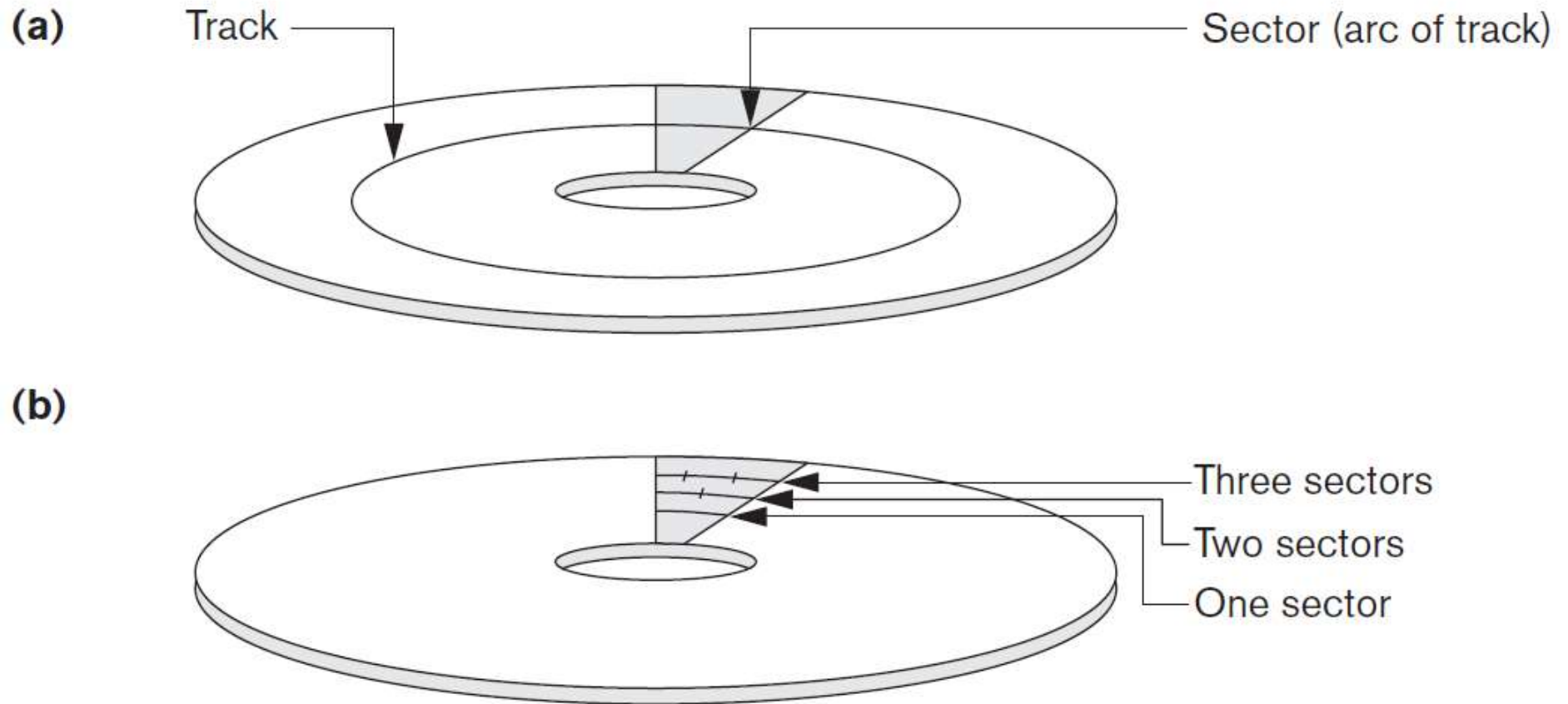
---

---

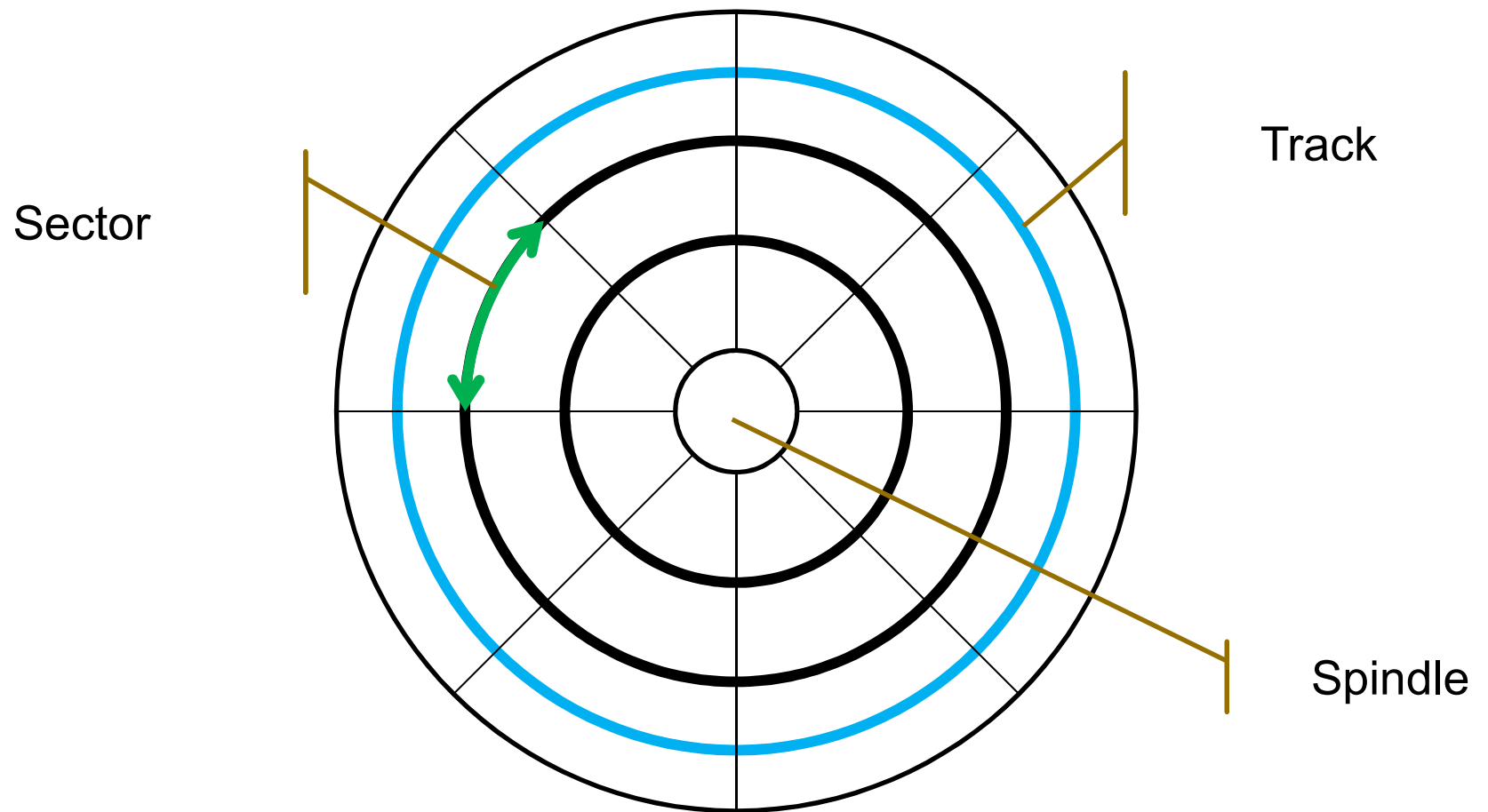
# Disk Storage Devices

- 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.)

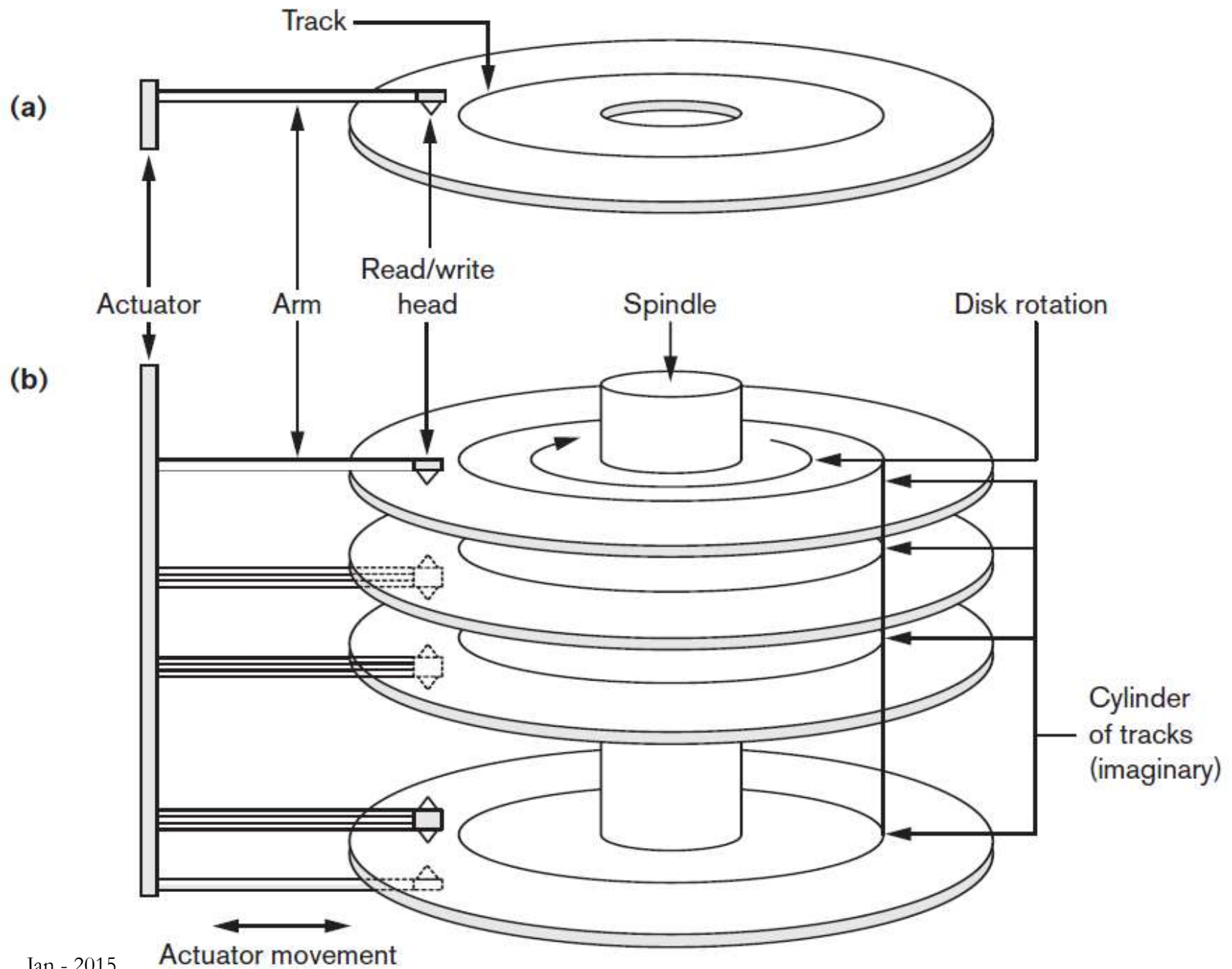
- A track is divided into smaller **blocks** or **sectors**.
  - because a track usually contains a large amount of information .
- A track is divided into **blocks**.
  - The block size  $B$  is fixed for each system.
    - Typical block sizes range from  $B=512$  bytes to  $B=4096$  bytes.
  - Whole 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.



---

# Contents

## **1 Data Storage**

### 1.1 Disk Storage Devices

### **1.2 Files of Records**

### 1.3 Operations on Files

### 1.4 Unordered Files & Ordered Files & Hashed Files

### 1.5 RAID Technology

## **2 Indexing Structures for Files**

### 2.1 Types of Single-level Ordered Indexes

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### 2.4 Indexes in Oracle

---

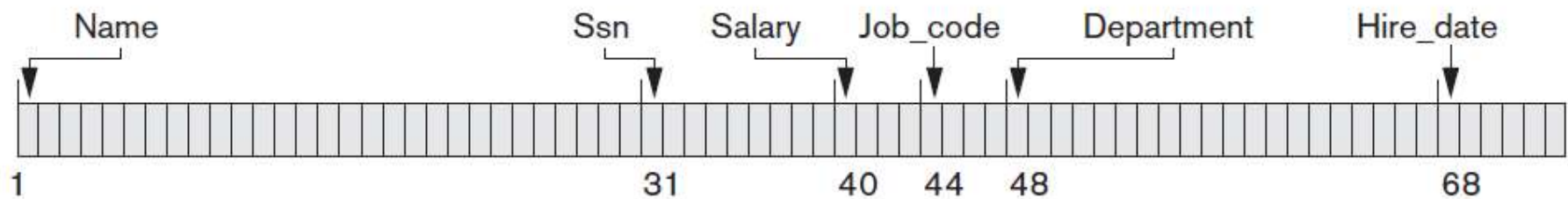
---

# Records

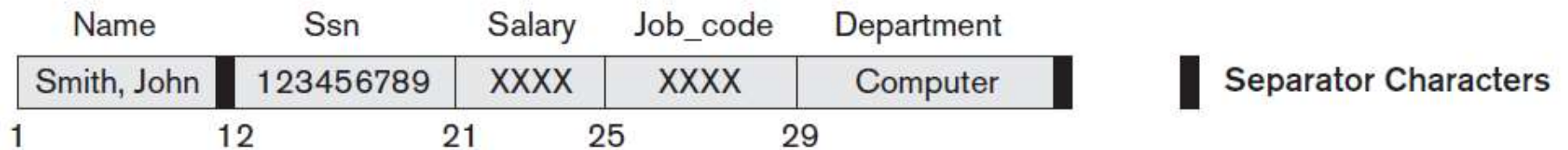
- Fixed and variable length records.
- Records contain fields which have values of a particular type.
  - E.g., amount, date, time, age.
- Fields themselves may be fixed length or variable length.
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed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

# Records (cont.)

(a)



(b)

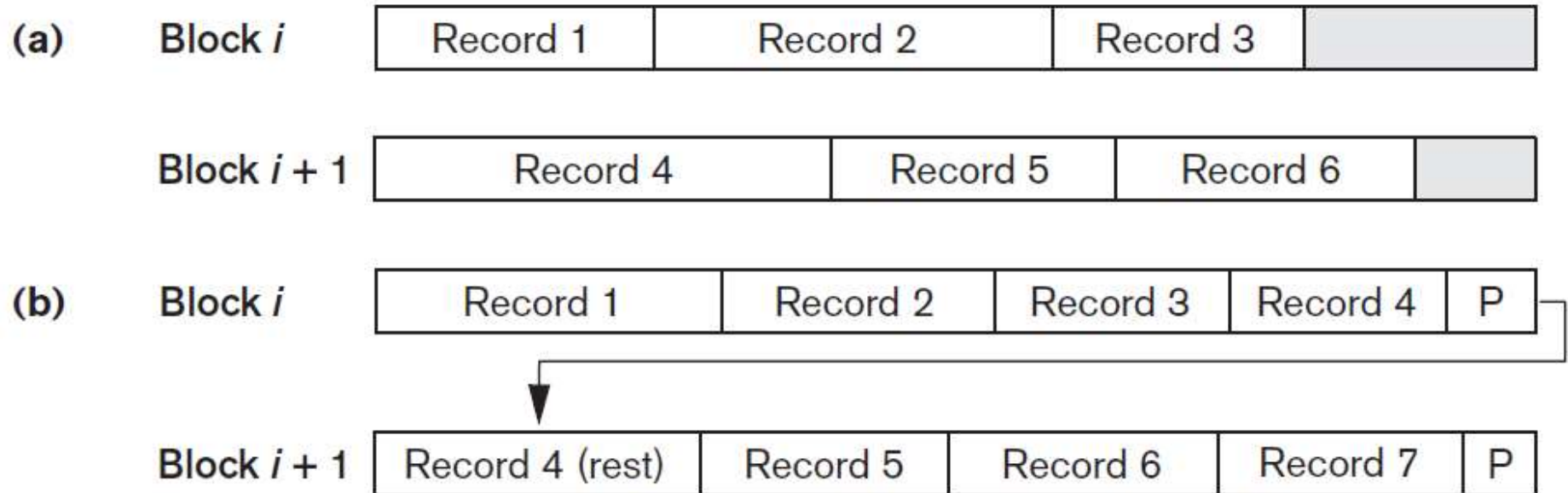


---

# 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.

---

# Contents

## **1 Data Storage**

### 1.1 Disk Storage Devices

### 1.2 Files of Records

### **1.3 Operations on Files**

### 1.4 Unordered Files & Ordered Files & Hashed Files

### 1.5 RAID Technology

## **2 Indexing Structures for Files**

### 2.1 Types of Single-level Ordered Indexes

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### 2.4 Indexes in Oracle

---

---

# 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.

---

# Contents

## **1 Data Storage**

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

**1.4 Unordered Files & Ordered Files & Hashed Files**

1.5 RAID Technology

## **2 Indexing Structures for Files**

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

2.4 Indexes in Oracle

---

---

# 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.)

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					

block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					

block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					

⋮

block n - 1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					

block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					



# 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

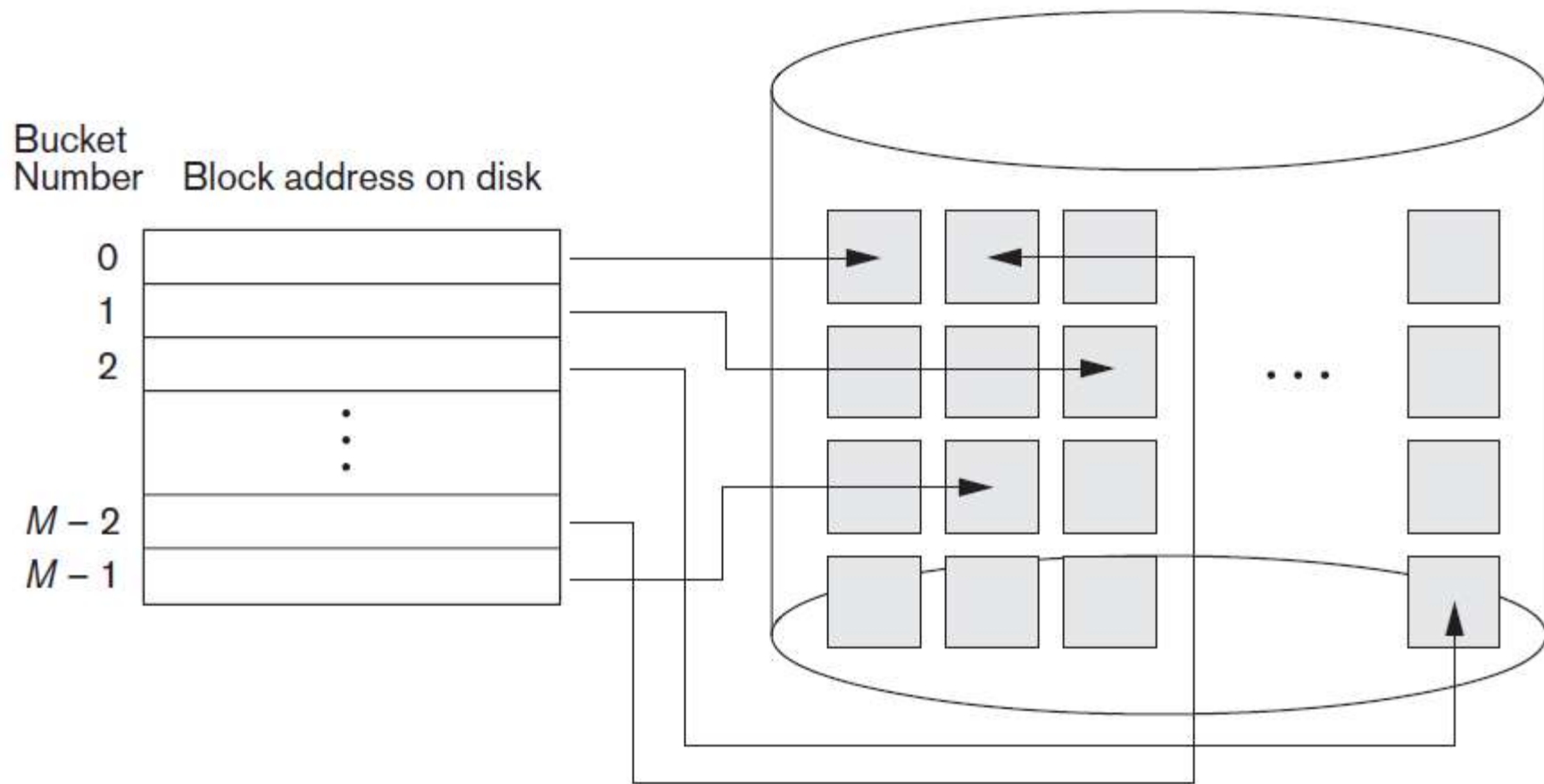
Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

---

# Hashed Files

- Hashing for disk files is called **External Hashing**.
  - The file blocks are divided into M equal-sized **buckets**, numbered  $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$ .
    - 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.

- $h(K) = K \bmod 7$

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

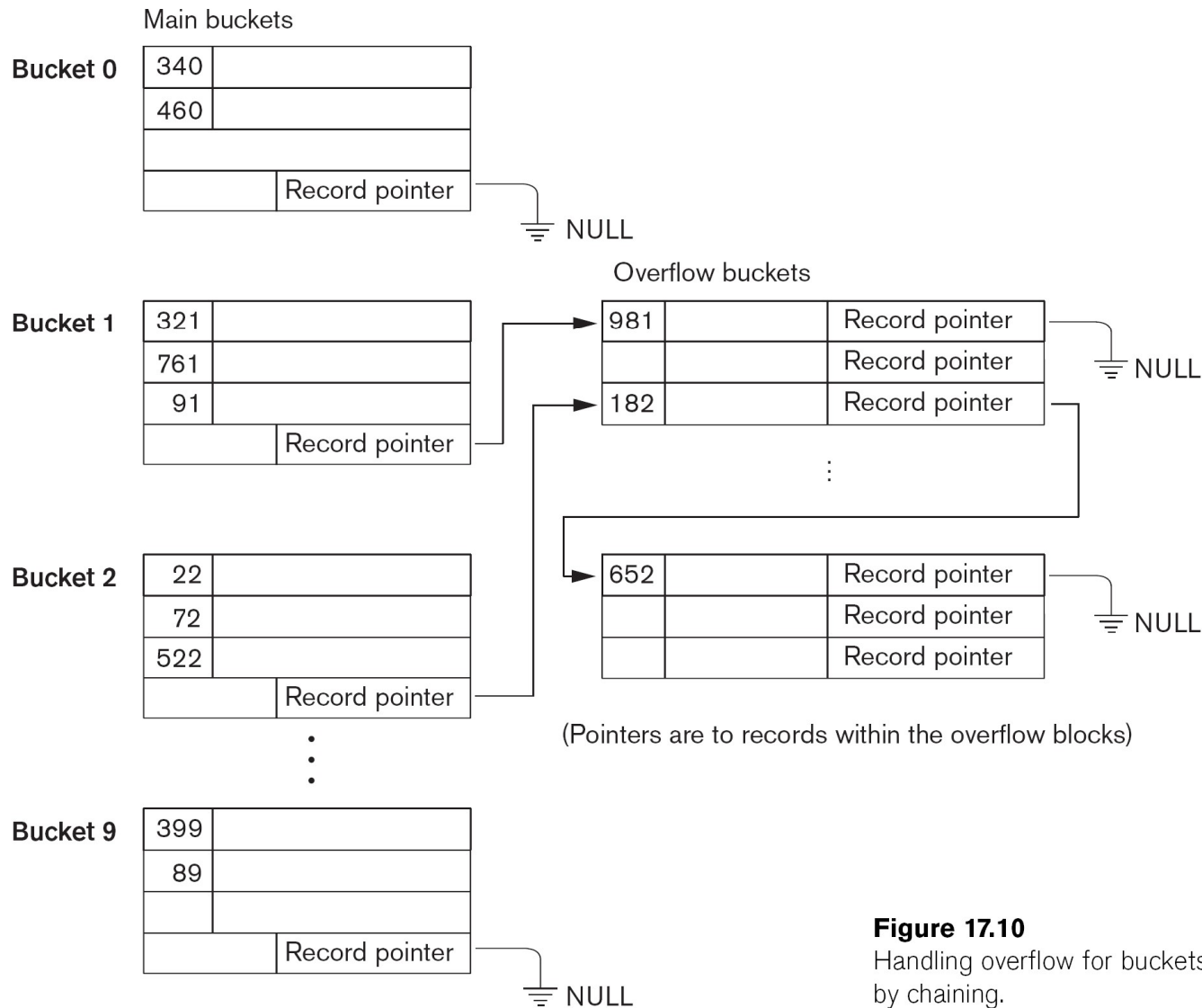
13	1	8	3	11	15	6
----	---	---	---	----	----	---

---

# 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).

---

# Contents

## **1 Data Storage**

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

**1.5 RAID Technology**

## **2 Indexing Structures for Files**

2.1 Types of Single-level Ordered Indexes

2.2 Multilevel Indexes

2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

2.4 Indexes in Oracle

---



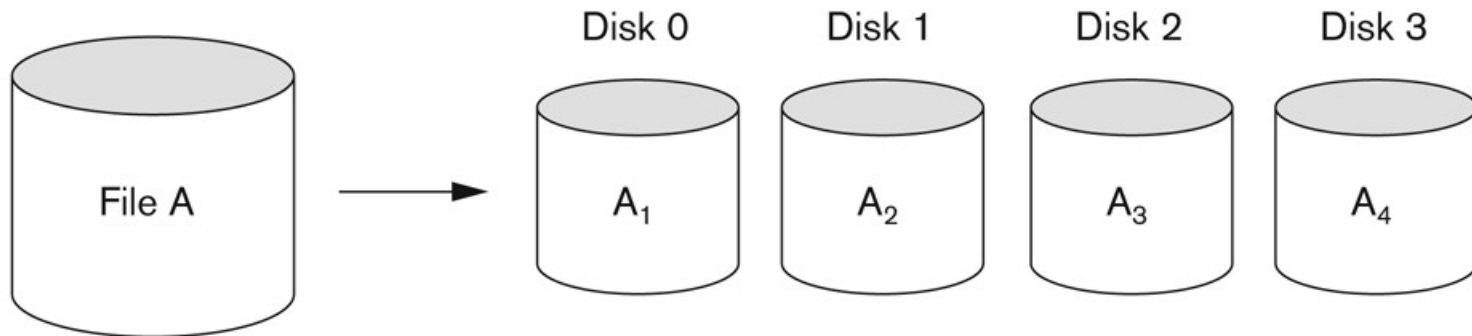
---

# Parallelizing Disk Access using RAID Technology.

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**.
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.

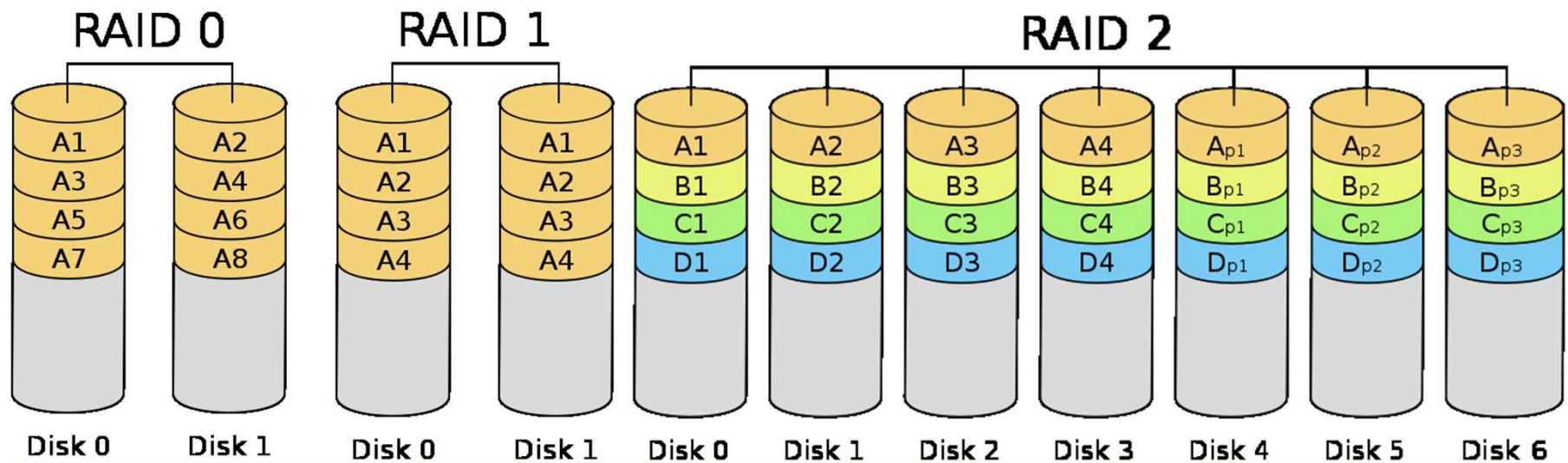
# RAID Technology (cont.)

- A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.
- A concept called **data striping** is used, which utilizes *parallelism* to improve disk performance.
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.



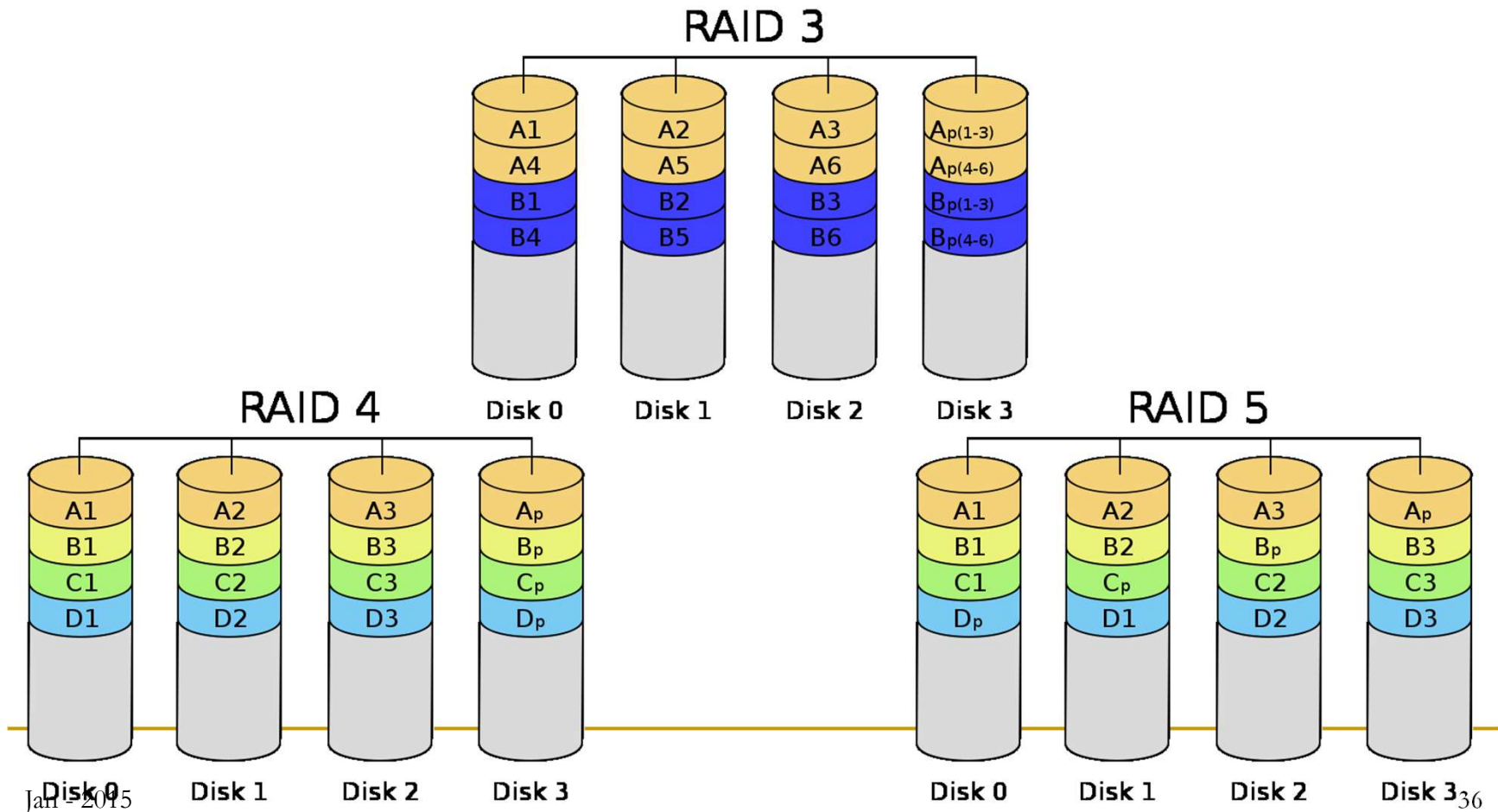
# RAID Technology (cont.)

- Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
  - **Raid level 0** has no redundant data and hence has the best write performance.
  - **Raid level 1** uses mirrored disks.
  - **Raid level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.



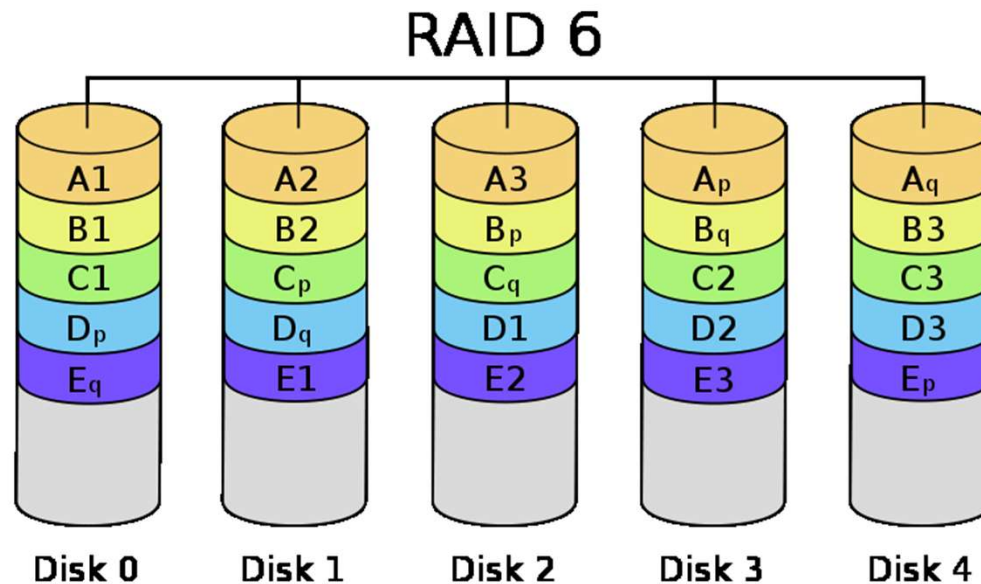
# RAID Technology (cont.)

- ❑ **Raid level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed.
- ❑ **Raid levels 4 and 5** use block-level data striping, with level 5 distributing data and parity information across all disks.



# RAID Technology (cont.)

- ❑ **Raid level 6** applies the so-called  $P + Q$  redundancy scheme using Reed-Solomon codes to protect against up to two disk failures by using just two redundant disks.



---

# Use of RAID Technology (cont.)

- Different raid organizations are being used under different situations:
  - Raid level 1 (mirrored disks) is the easiest for rebuild of a disk from other disks
    - It is used for critical applications like logs.
  - Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
  - Raid level 3 (single parity disks relying on the disk controller to figure out which disk has failed) and level 5 (block-level data striping) are preferred for large volume storage, with level 3 giving higher transfer rates.
  - Most popular uses of the RAID technology currently are: Level 0 (with striping), Level 1 (with mirroring) and Level 5 with an extra drive for parity.
  - Design decisions for RAID include – level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.

---

# Storage Area Networks

- The demand for higher storage has risen considerably in recent times.
- Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing.
- Thus they are moving to a concept of Storage Area Networks (SANs).
  - In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.
- This allows storage systems to be placed at longer distances from the servers and provide different performance and connectivity options.

---

# Storage Area Networks (contd.)

- **Advantages of SANs are:**

- Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches.
- Up to 10km separation between a server and a storage system using appropriate fiber optic cables.
- Better isolation capabilities allowing nondisruptive addition of new peripherals and servers.

- SANs face the problem of combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware.



---

# Contents

---

## **1 Data Storage**

### 1.1 Disk Storage Devices

### 1.2 Files of Records

### 1.3 Operations on Files

### 1.4 Unordered Files & Ordered Files & Hashed Files

### 1.5 RAID Technology

## **2 Indexing Structures for Files**

### **2.1 Types of Single-level Ordered Indexes**

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### 2.4 Indexes in Oracle

---

---

# 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

---

# 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**  
 ( $\langle K(i), P(i) \rangle$  entries)

Primary key value	Block pointer
1	
4	
8	
12	

↓ Primary key field

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  $\lceil \log_2 b \rceil = \lceil (\log_2 7,500) \rceil = 13$  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 file, 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

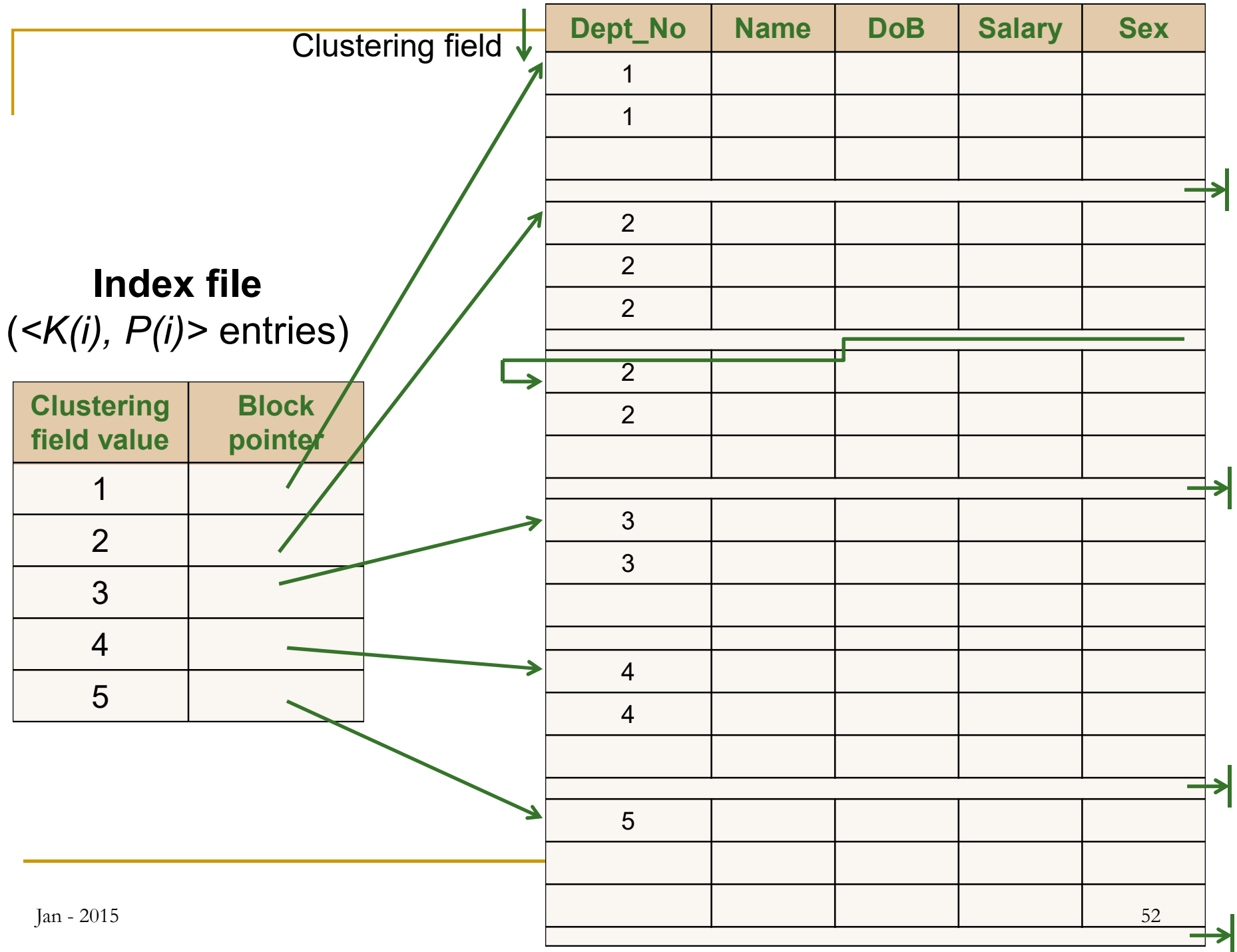
- Defined on an **ordered data file**.
  - The data file is ordered on a *non-key field*.
- One index entry *each distinct value* of the field.
  - The index entry points to the *first data block* that contains records with that field value

↓ Clustering field

**Index file**  
( $\langle K(i), P(i) \rangle$  entries)

Clustering field value	Block pointer
1	
2	
3	
4	
5	

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



---

# 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
- $R_i = 5\text{-byte Zipcode and } 6\text{-byte block pointer}$
- $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/11) \rfloor = 372$  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 ( $\langle K(i), P(i) \rangle$ entries)

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

## Secondary key field

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

...

## Secondary index on key field



---

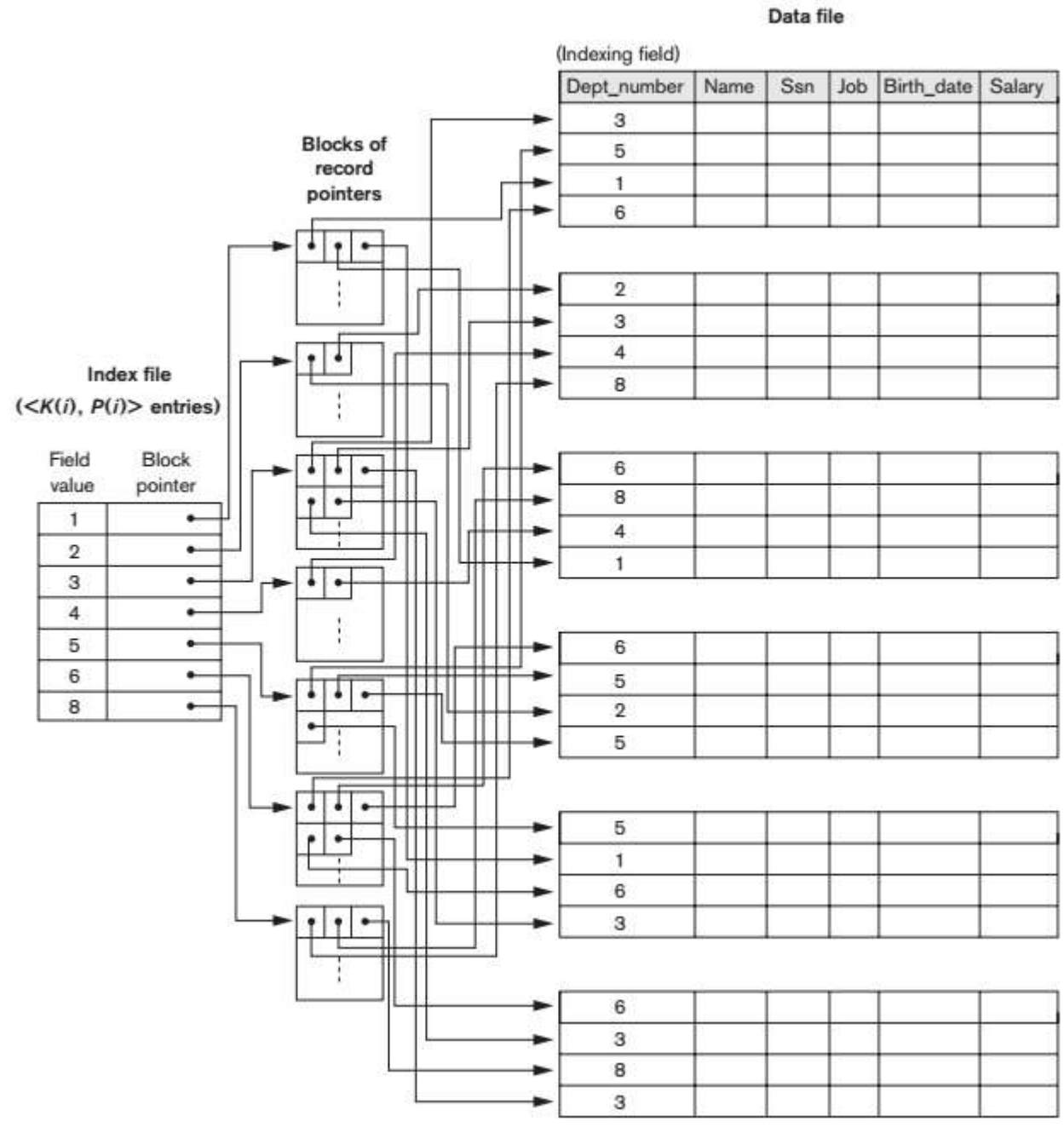
# Secondary index on key field

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

# 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**  $\langle P(i, 1), \dots, P(i, k) \rangle$  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?

## 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:  $11 + 1 = 12$

---

# 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 <sup>a</sup>
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup>	Dense or Nondense	No

<sup>a</sup>Yes if every distinct value of the ordering field starts a new block; no otherwise.

<sup>b</sup>For option 1.

<sup>c</sup>For options 2 and 3.



---

# Contents

---

## **1 Data Storage**

1.1 Disk Storage Devices

1.2 Files of Records

1.3 Operations on Files

1.4 Unordered Files & Ordered Files & Hashed Files

1.5 RAID Technology

## **2 Indexing Structures for Files**

2.1 Types of Single-level Ordered Indexes

**2.2 Multilevel Indexes**

2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

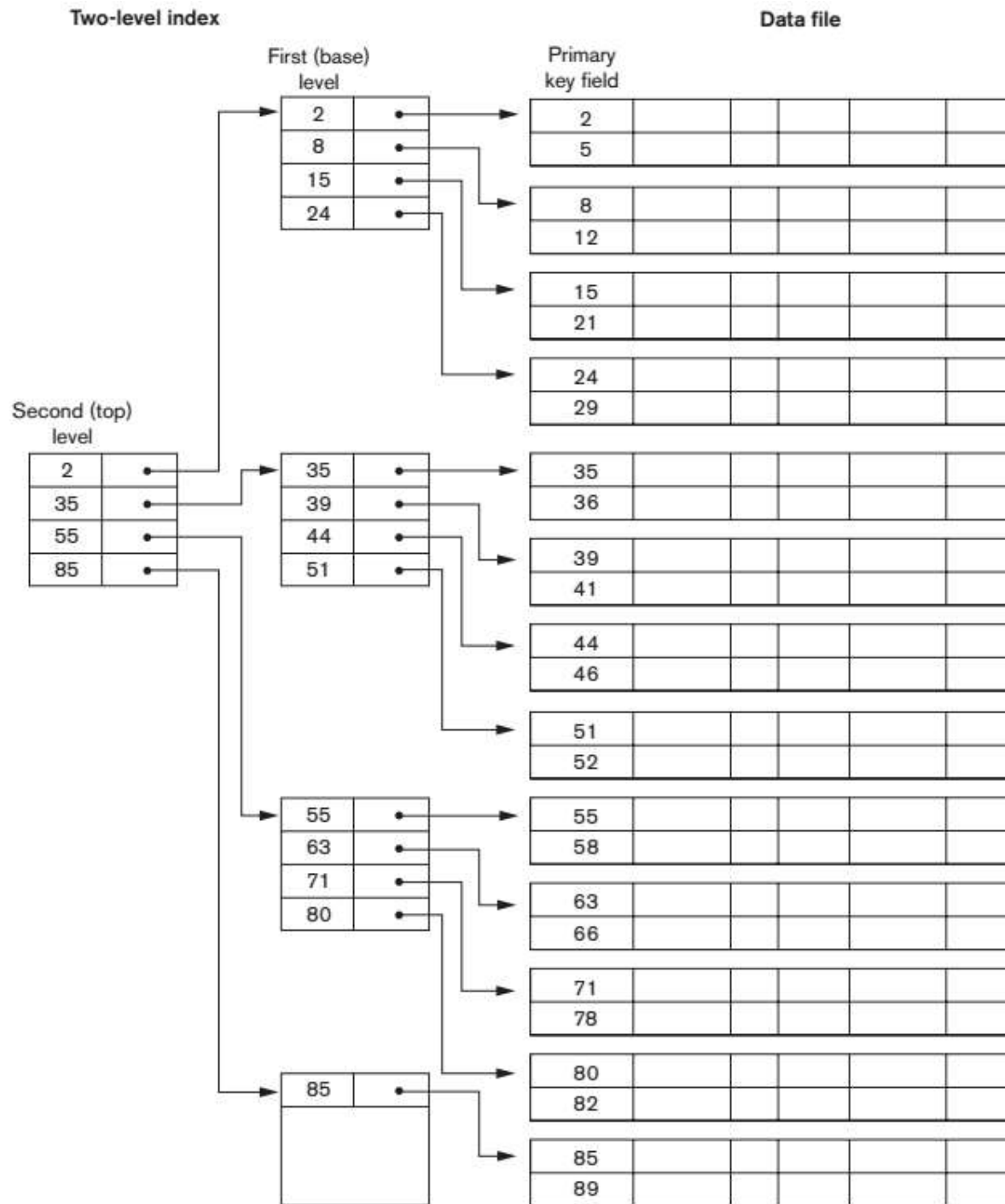
2.4 Indexes in Oracle

---

---

# 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

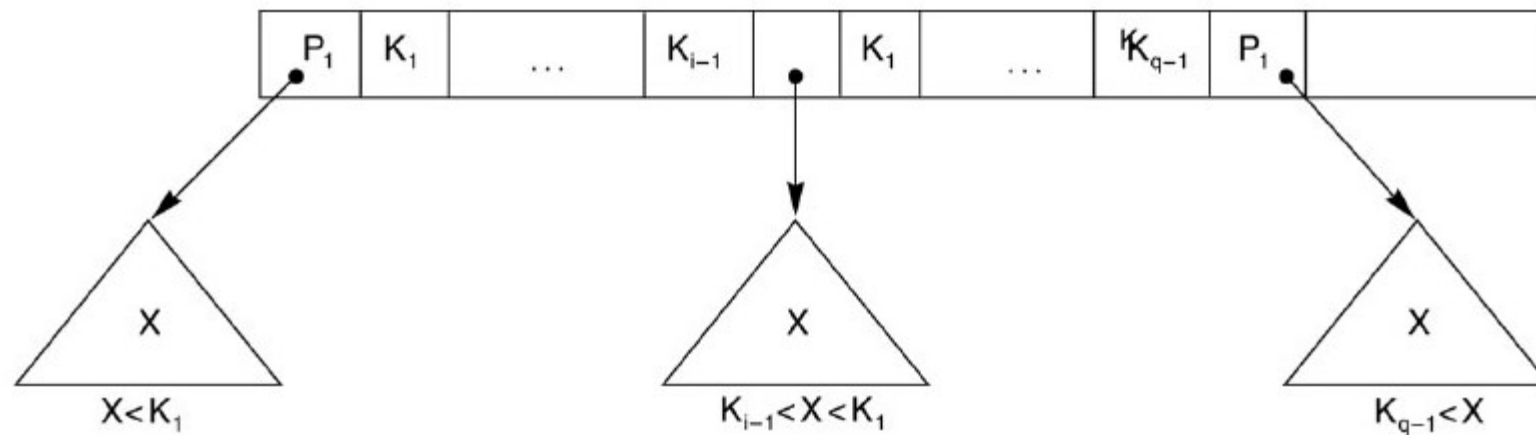
- *From example 3:  $bfr_i = 273$  index entries per block (fo) and  $b_1 = 1,099$  blocks*
  - $b_2 = \lceil (b_1/fo) \rceil = \lceil (1,099/273) \rceil = 5$  blocks
  - $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$  block
- Top level of the index:  $t = 3$
- Total block accesses:  $t +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

---

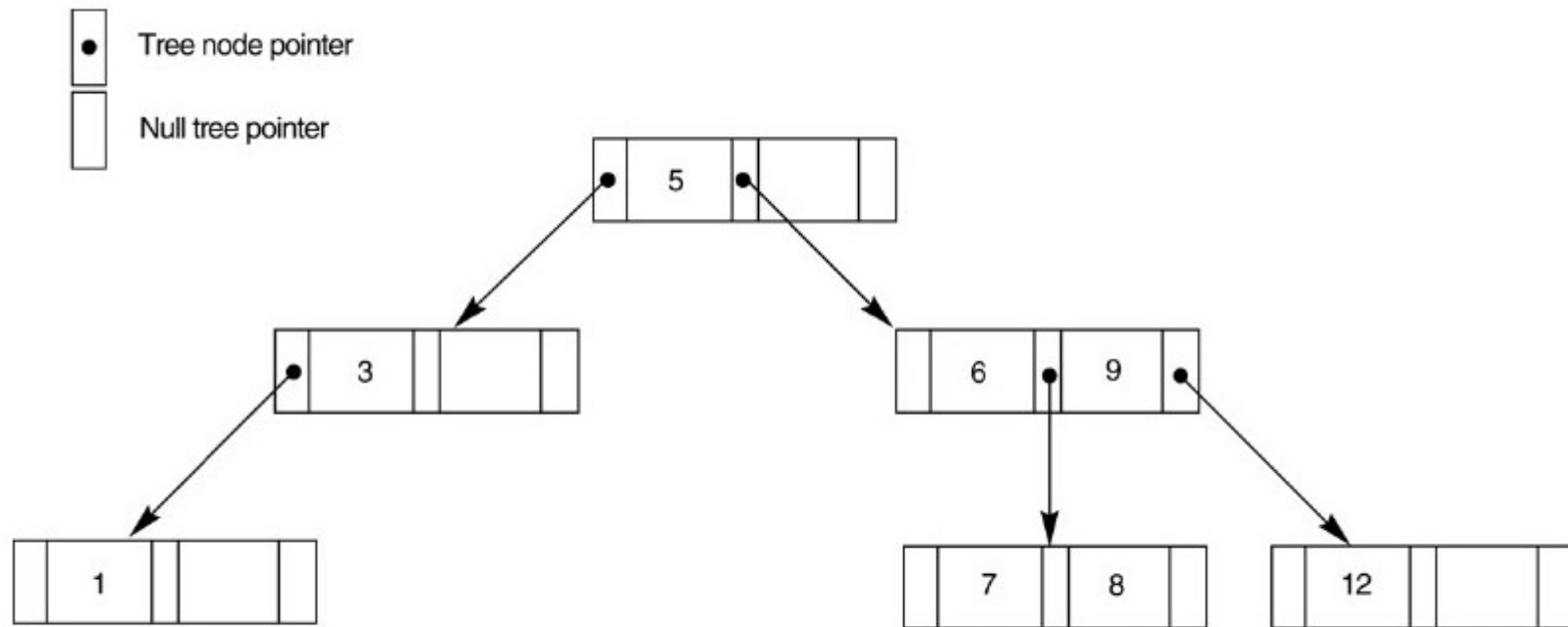
# Multi-Level Indexes

- 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 $p = 3$



---

# Contents

---

## 1 Data Storage

### 1.1 Disk Storage Devices

### 1.2 Files of Records

### 1.3 Operations on Files

### 1.4 Unordered Files & Ordered Files & Hashed Files

### 1.5 RAID Technology

## 2 Indexing Structures for Files

### 2.1 Types of Single-level Ordered Indexes

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### 2.4 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.)

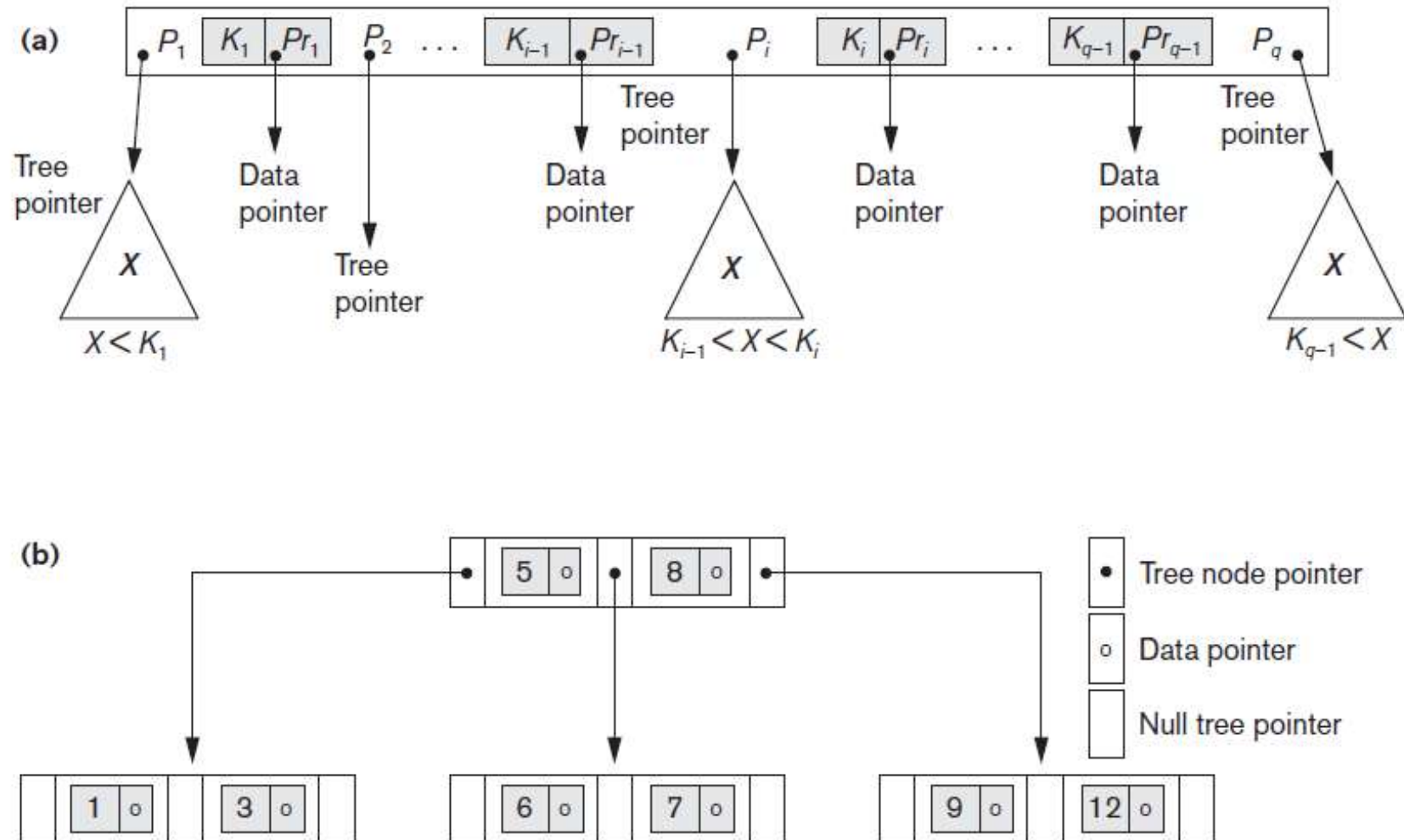
- An insertion into a node that is not full is quite efficient.
  - If a node is full, the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.
- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less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

---

## Difference between B-tree and B<sup>+</sup>-tree

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

# B-tree Structures

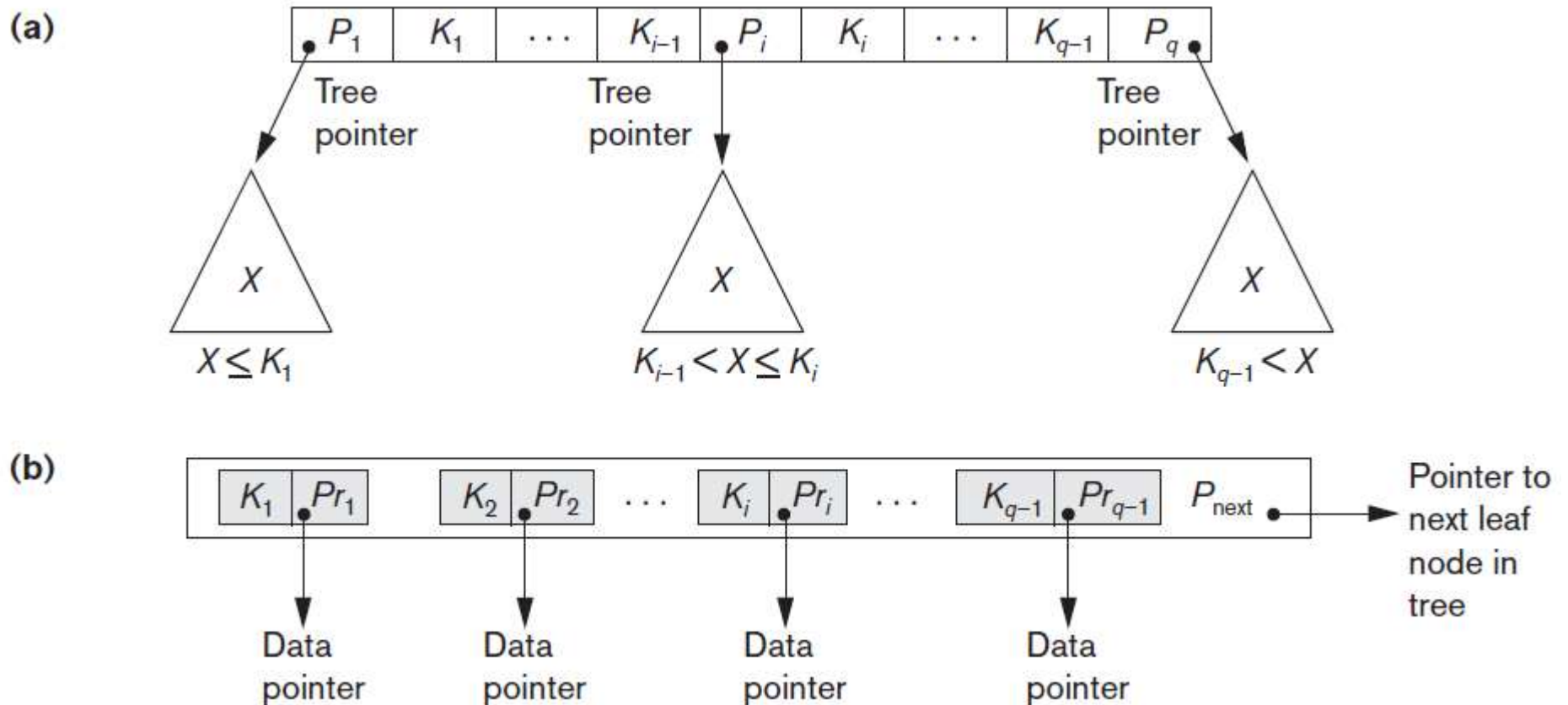


**Figure 18.10**

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.

Jan 2015

# The Nodes of a B<sup>+</sup>-Tree



**Figure 18.11**

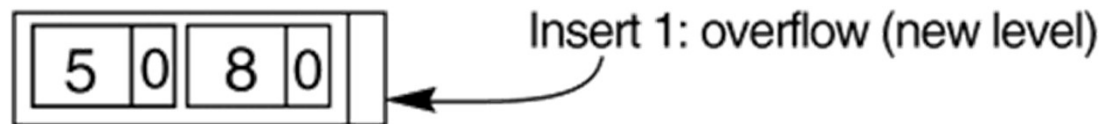
The nodes of a B<sup>+</sup>-tree. (a) Internal node of a B<sup>+</sup>-tree with  $q - 1$  search values. (b) Leaf node of a B<sup>+</sup>-tree with  $q - 1$  search values and  $q - 1$  data pointers.

# Example of insertion in B<sup>+</sup>-Tree

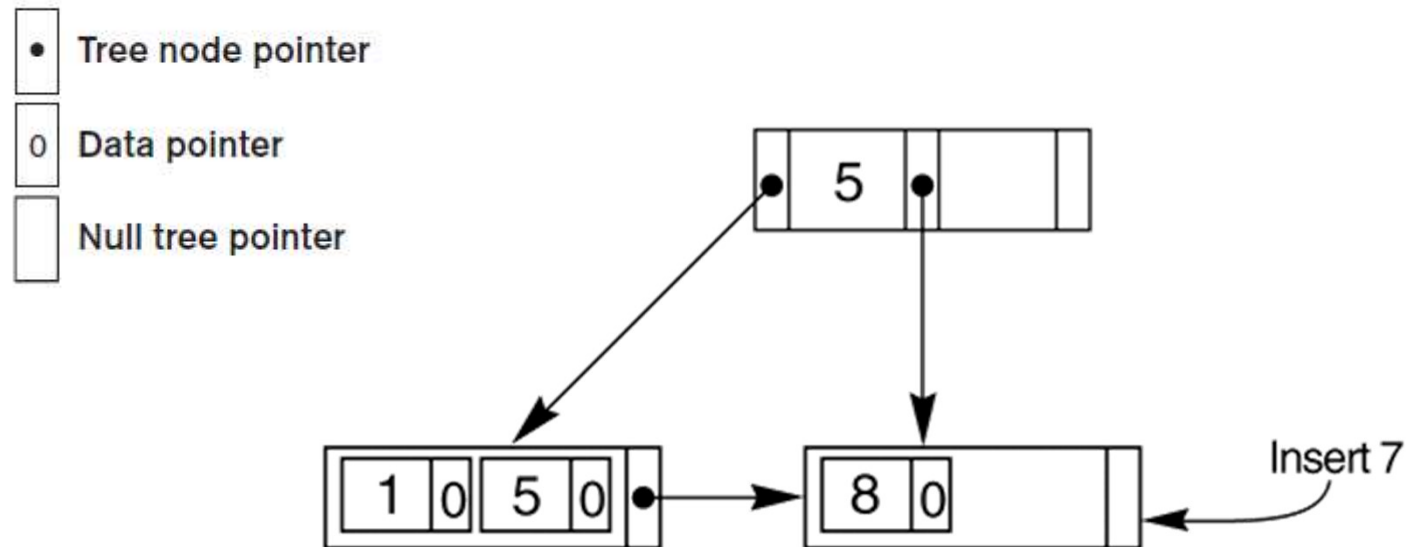
- Tree node pointer
- 0 Data pointer
- Null tree pointer

**p = 3 and p<sub>leaf</sub> = 2**

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

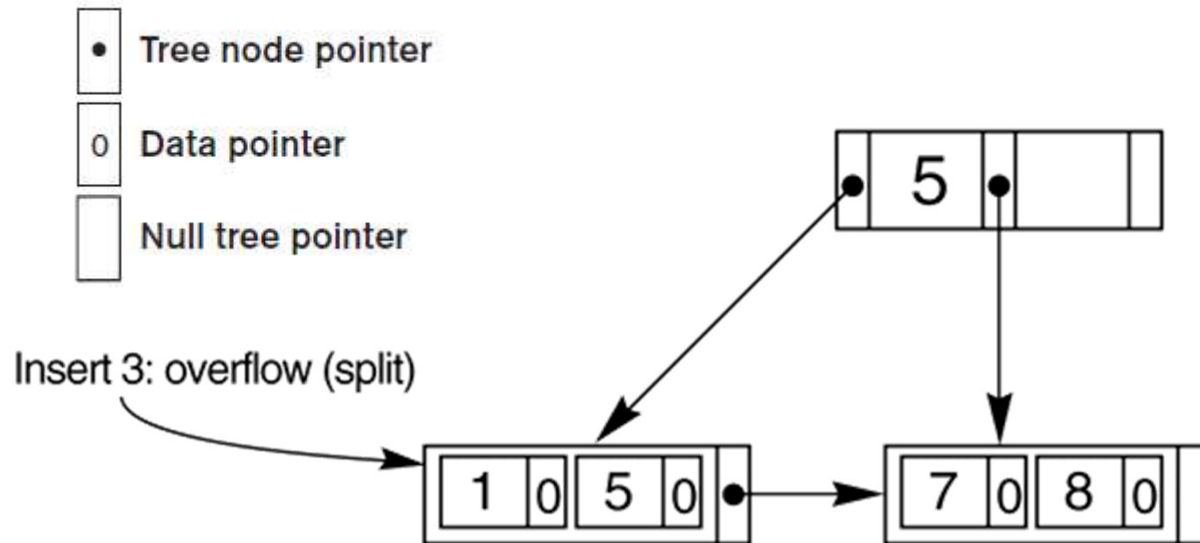


# Example of insertion in B<sup>+</sup>-Tree (cont.)



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

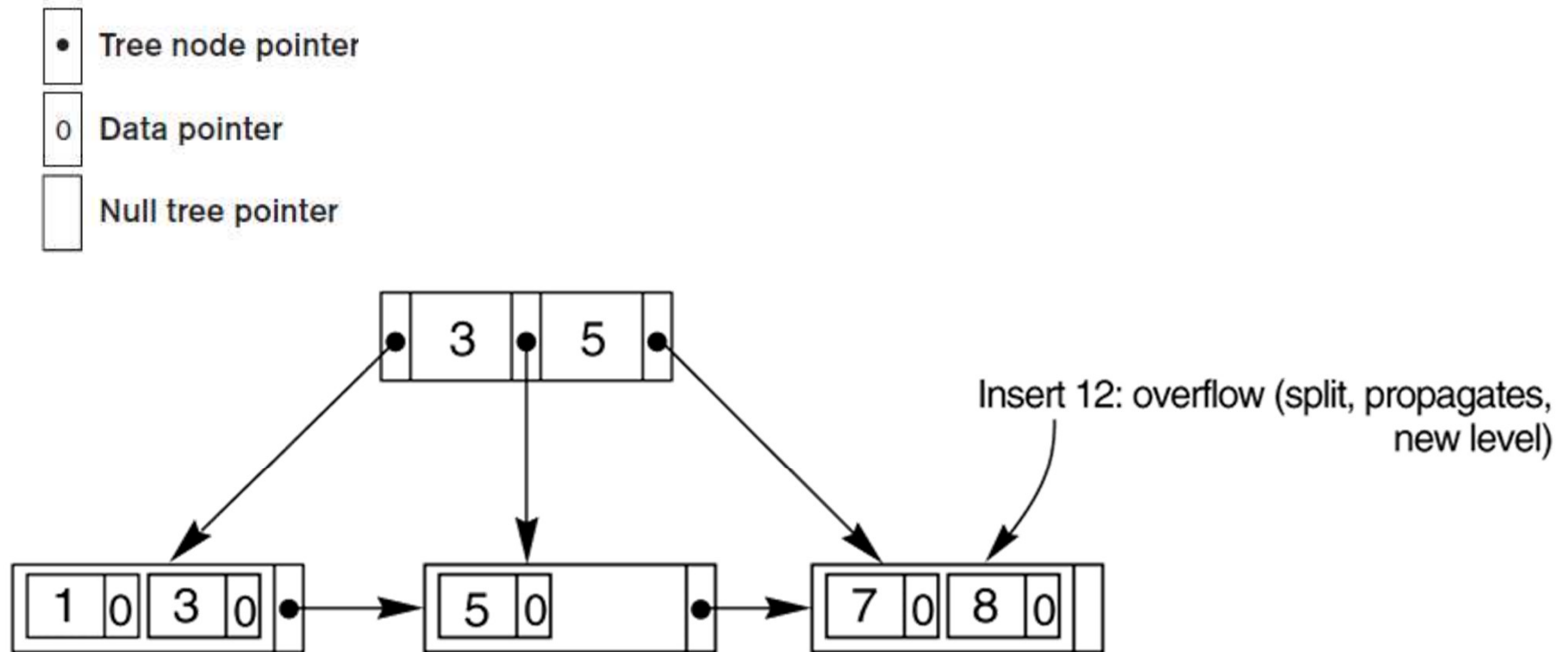
# Example of insertion in B<sup>+</sup>-Tree (cont.)



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

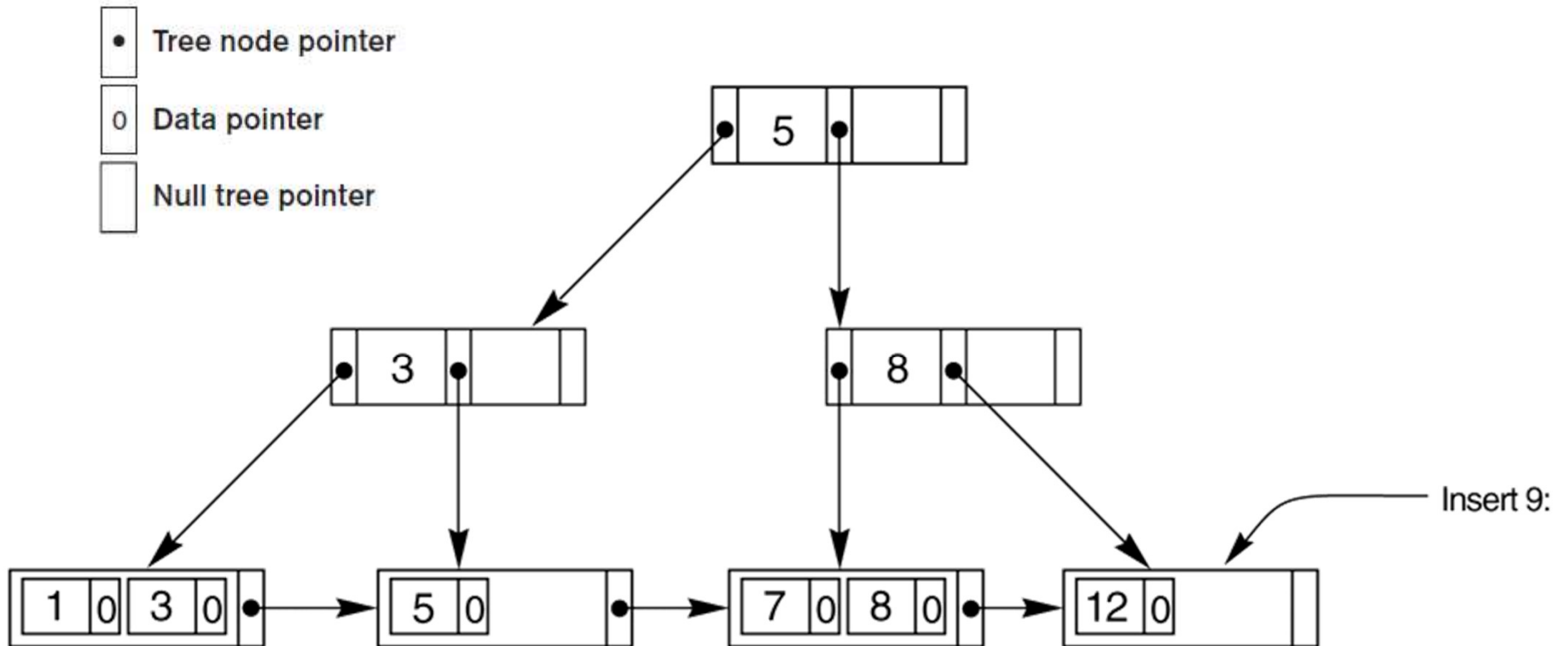


# Example of insertion in B<sup>+</sup>-Tree (cont.)



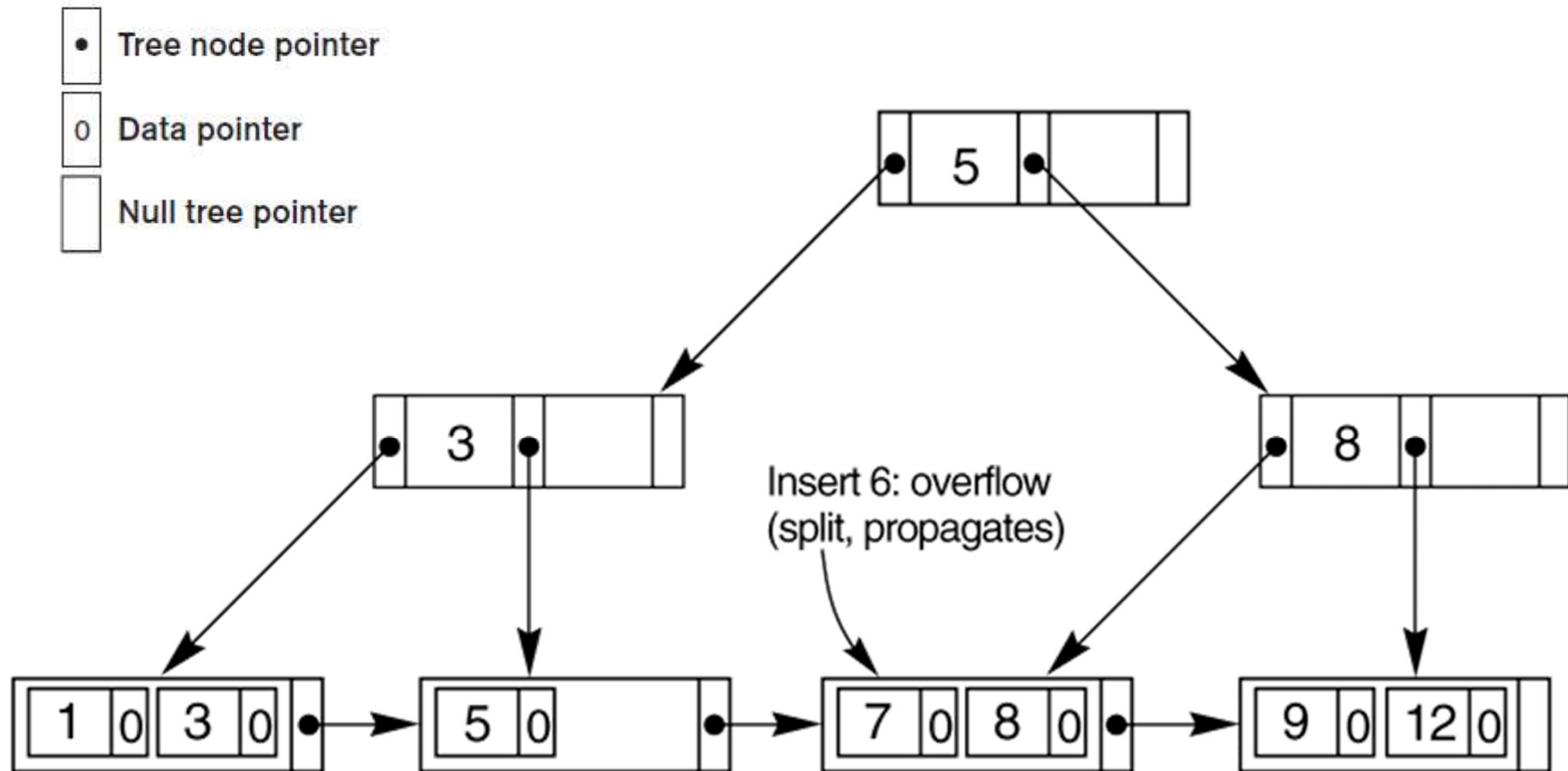
**Insertion Sequence:** 8, 5, 1, 7, 3, 12, 9, 6

# Example of insertion in B<sup>+</sup>-Tree (cont.)



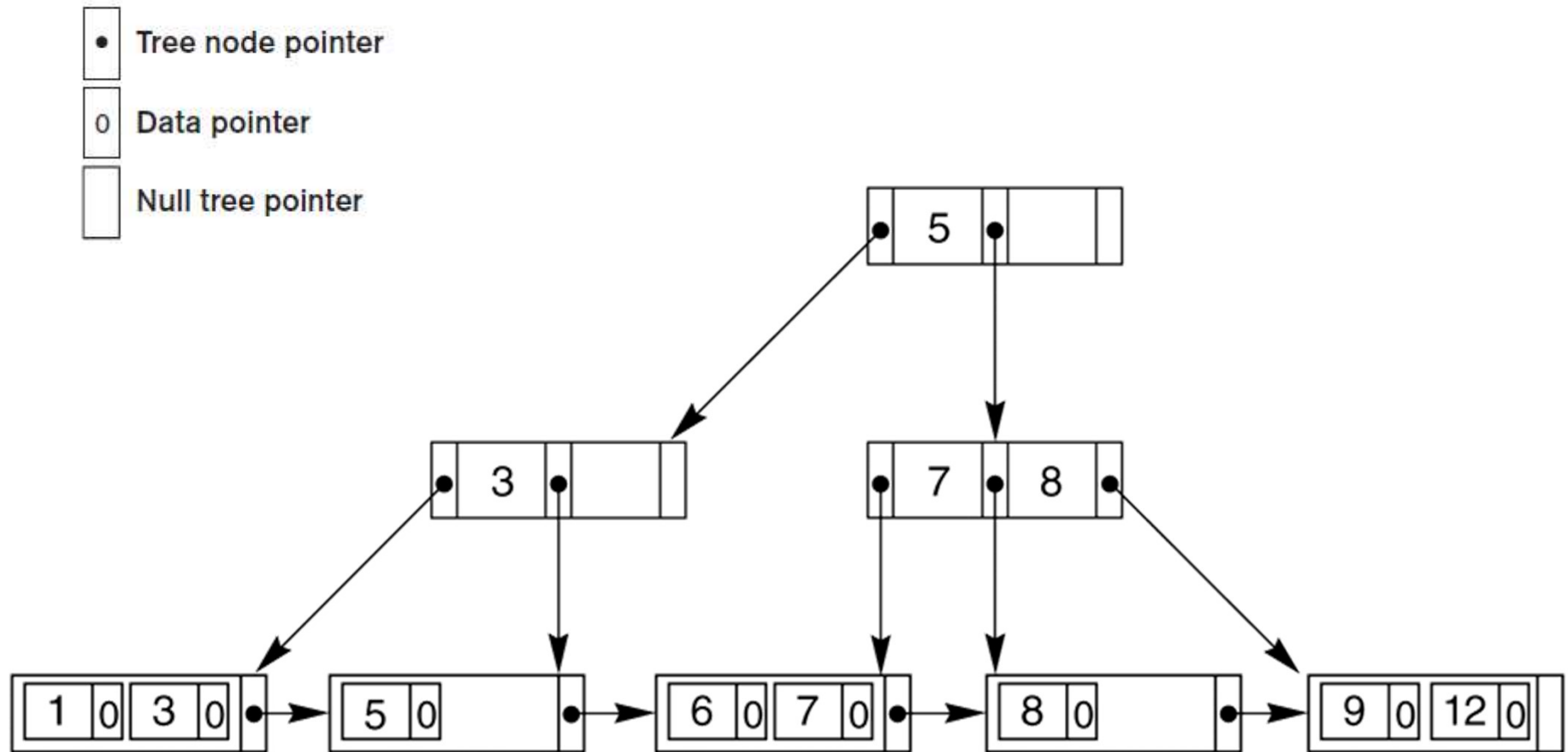
**Insertion Sequence:** 8, 5, 1, 7, 3, 12, 9, 6

# Example of insertion in B<sup>+</sup>-Tree (cont.)



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

# Example of insertion in B<sup>+</sup>-Tree (cont.)



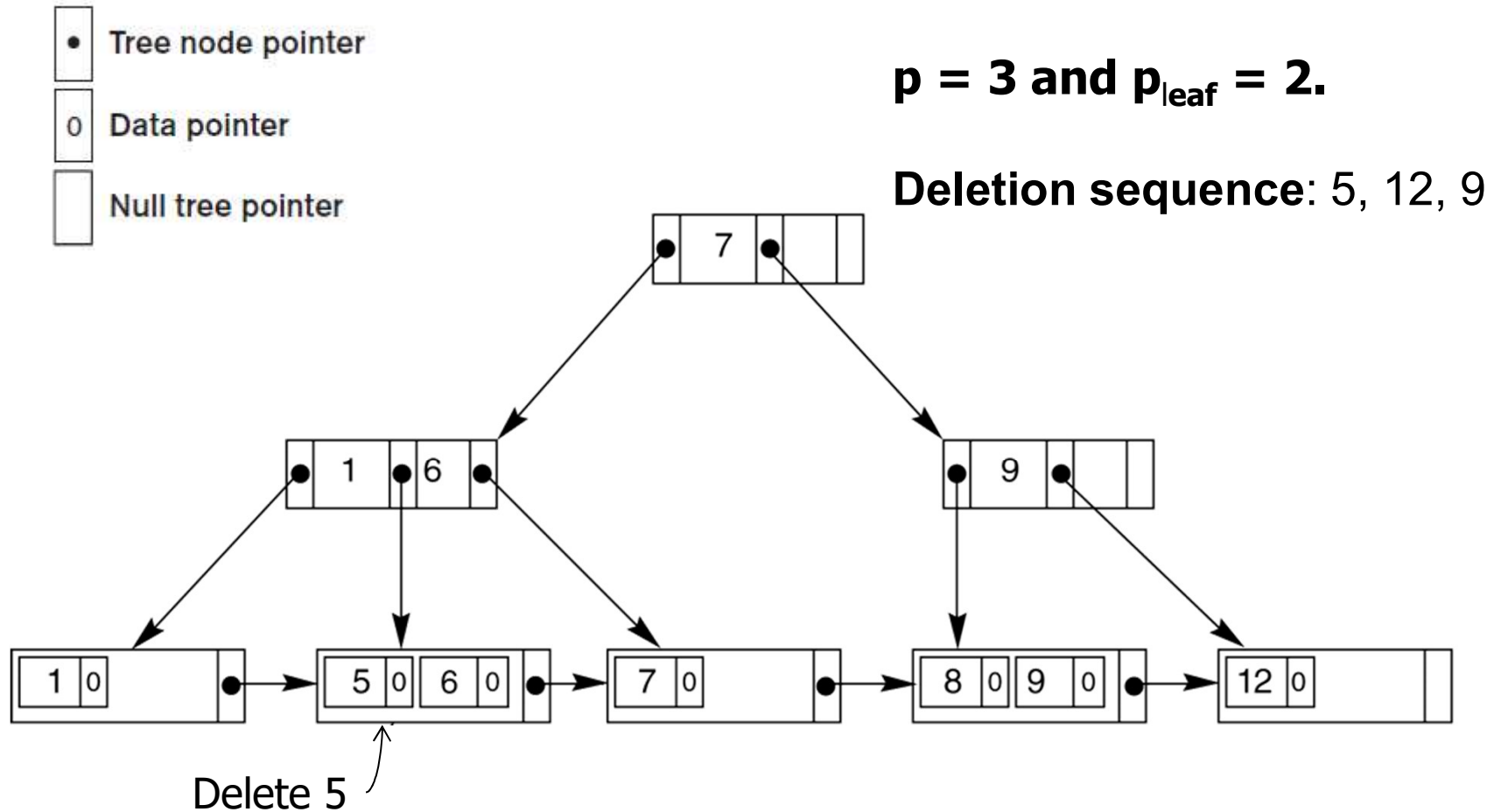
**Insertion Sequence:** 8, 5, 1, 7, 3, 12, 9, 6

---

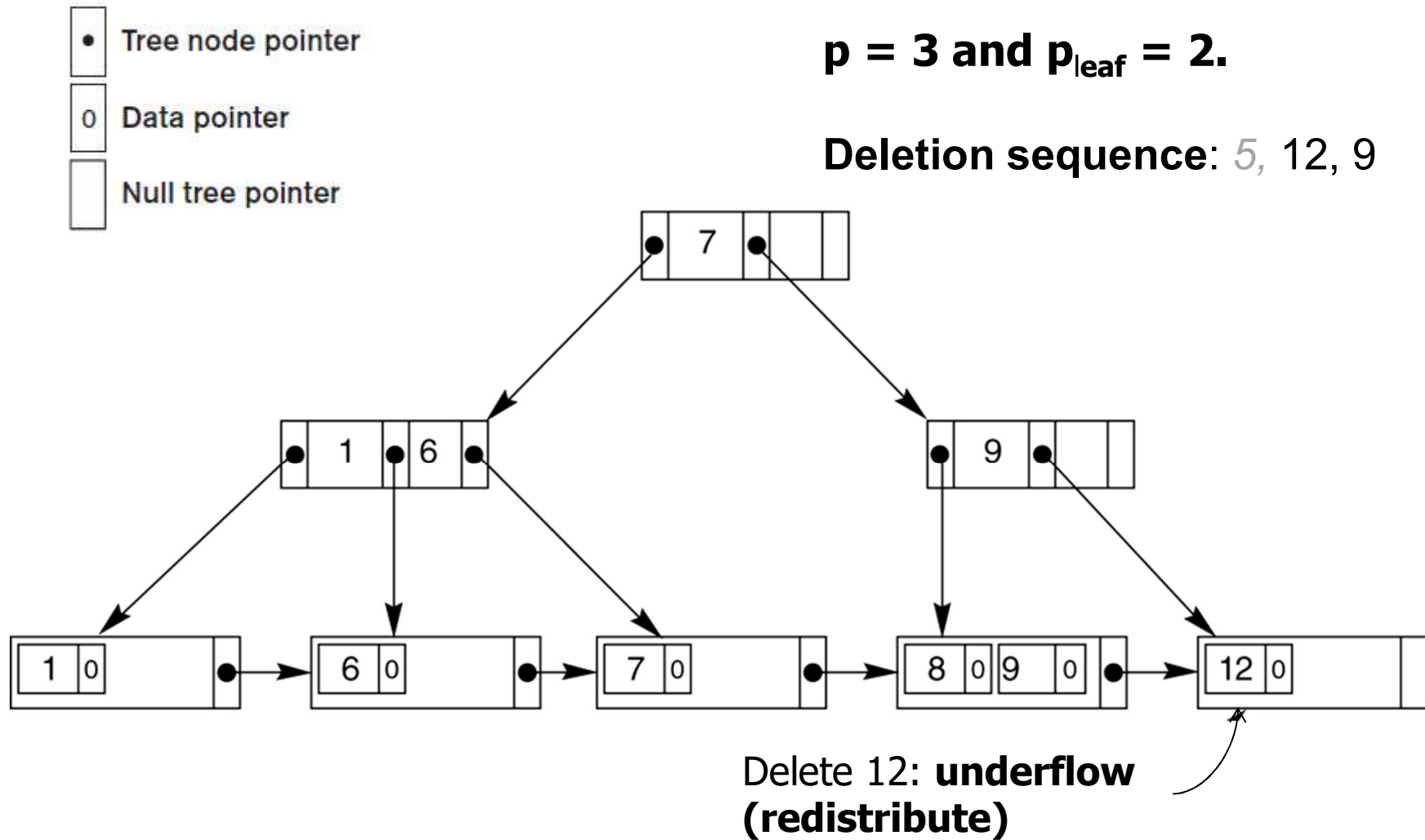
## B<sup>+</sup>-Tree: Delete entry

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has fewer entries than it should,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

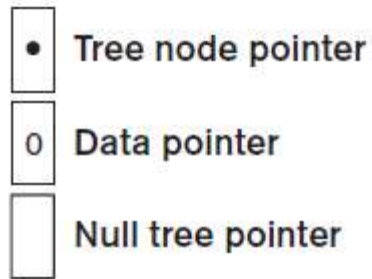
# Example of deletion from B<sup>+</sup>-Tree



# Example of deletion from B<sup>+</sup>-Tree (cont.)

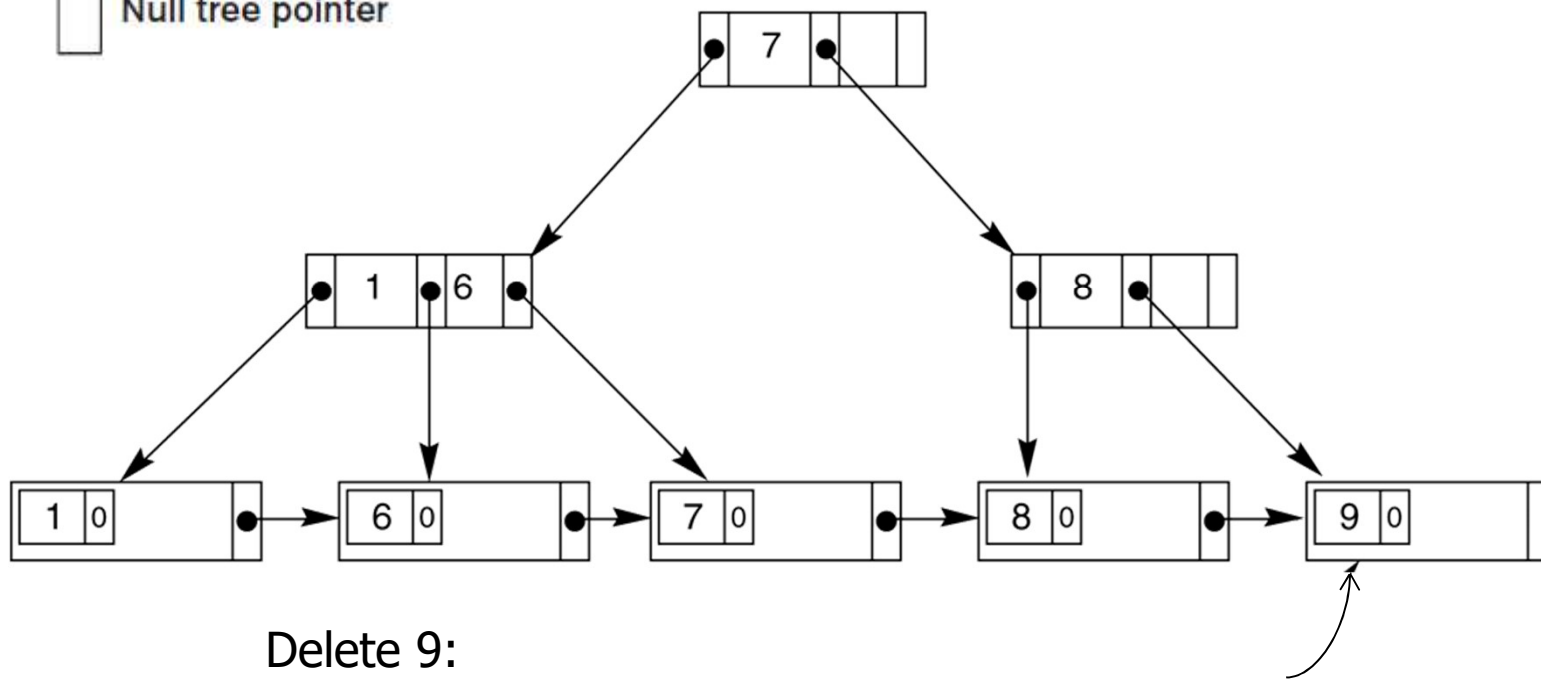


# Example of deletion from B<sup>+</sup>-Tree (cont.)



**p = 3 and p<sub>leaf</sub> = 2.**

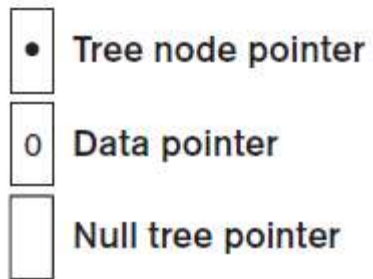
**Deletion sequence: 5, 12, 9**



Delete 9:  
**Underflow (merge with left, redistribute)**

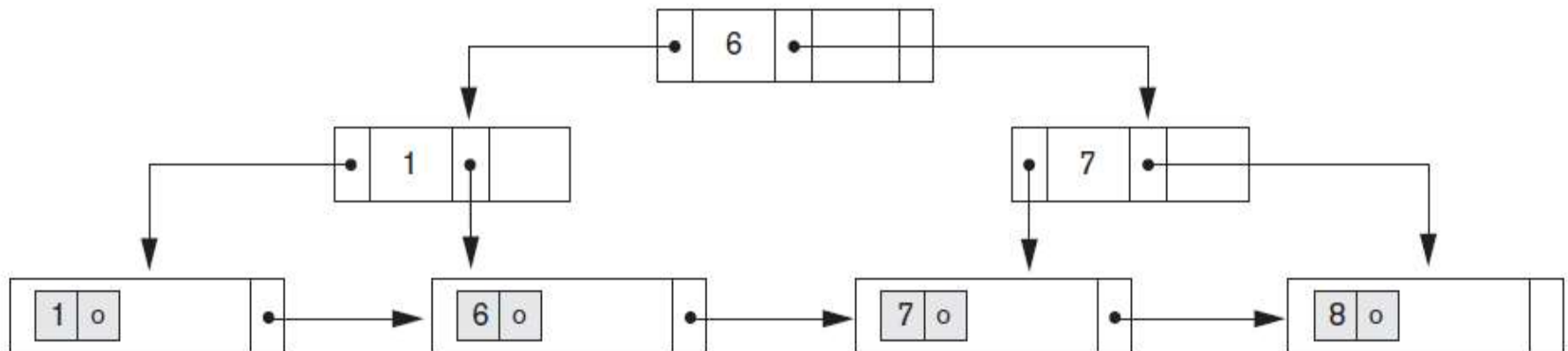


# Example of deletion from B<sup>+</sup>-Tree (cont.)



**p = 3 and p<sub>leaf</sub> = 2.**

**Deletion sequence:** 5, 12, 9



---

# Contents

---

## **1 Data Storage**

### 1.1 Disk Storage Devices

### 1.2 Files of Records

### 1.3 Operations on Files

### 1.4 Unordered Files & Ordered Files & Hashed Files

### 1.5 RAID Technology

## **2 Indexing Structures for Files**

### 2.1 Types of Single-level Ordered Indexes

### 2.2 Multilevel Indexes

### 2.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

### **2.4 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.



---

Q & A