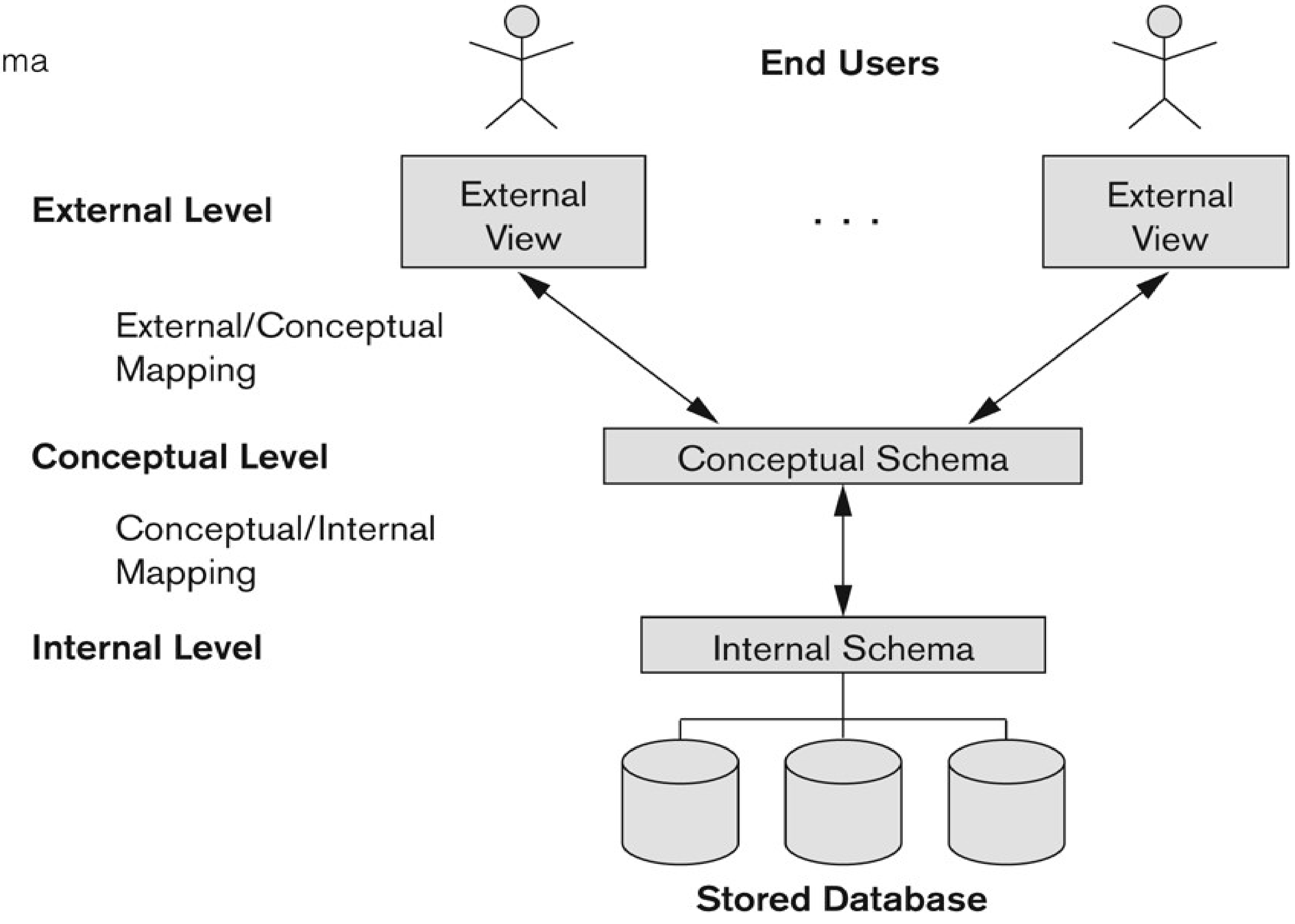
CS3402 : Chapter 8
Files and Hashed Files

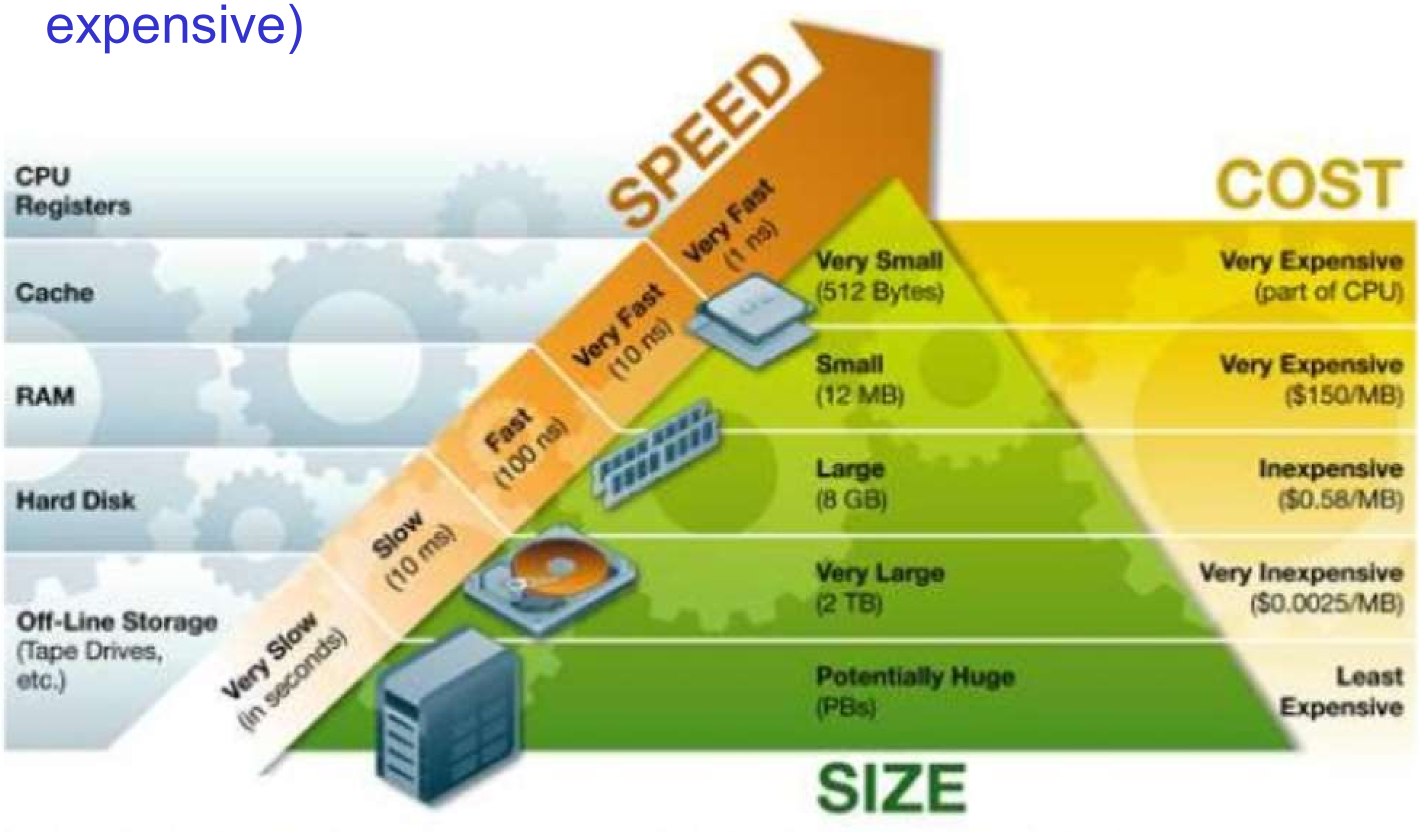
Data Abstraction: 3-level architecture

Figure 2.2
The three-schema architecture.



Storage Medium for Databases

- Memory hierarchy
  - CPU register->CPU cache => RAM=> magnetic disks/optical disks=>offline storage
  - Slower in access delay but larger in memory size (less expensive)



Memory Hierarchy

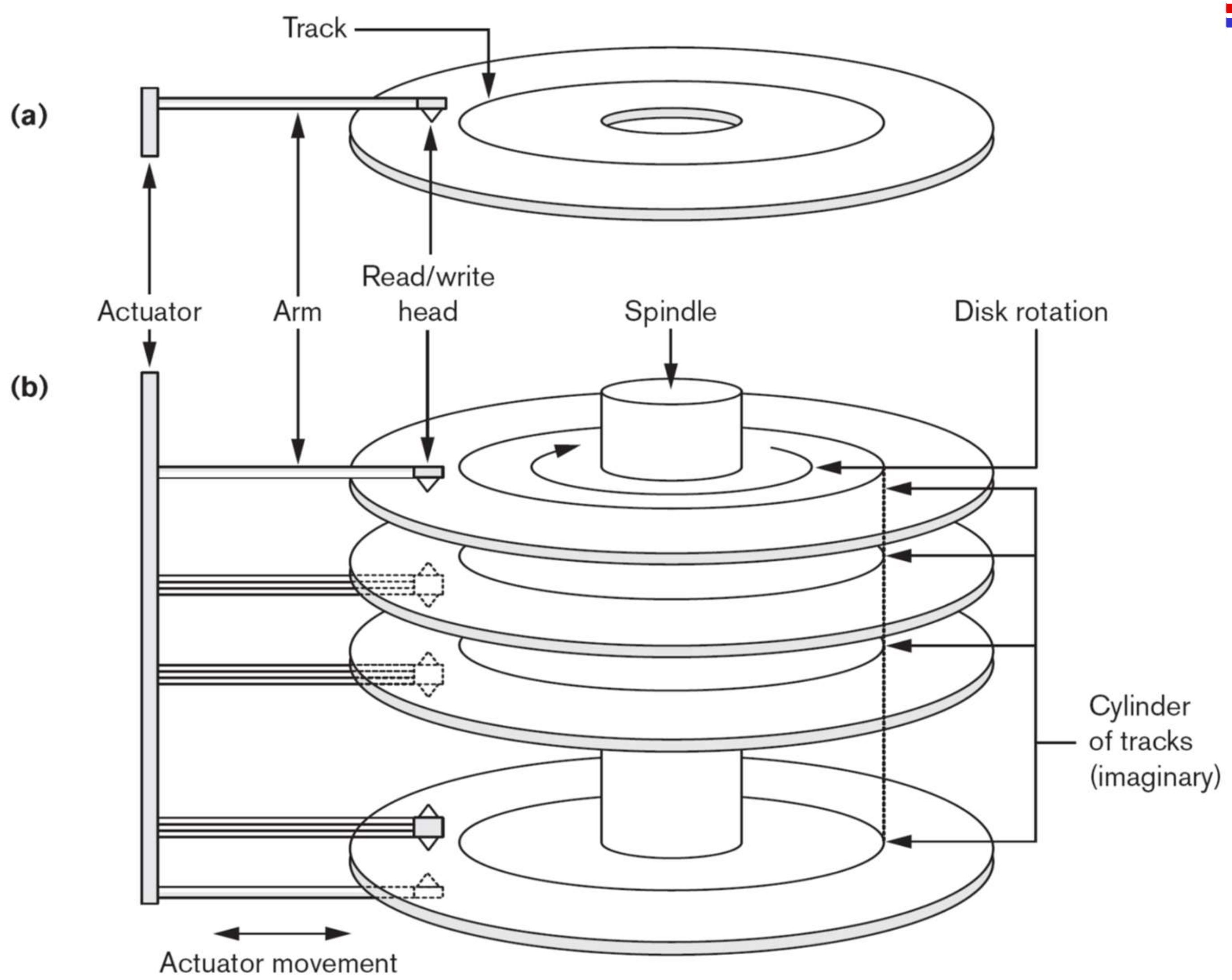
- Primary storage
  - The storage media that can be operated directly by the CPU.
  - Include register, cache, RAM.
- Secondary storage
  - Cannot be operated directly by the CPU, lower speed in access
  - Include magnetic disks, SSD and flash memory (U disk).
- Tertiary storage
  - also cannot be operated directly by the CPU, even slower
  - removable media used as offline storage, include Optical disks (CD-ROMs, DVDs) and magnetic tapes.
- A database could be huge in size (several GB or even larger)
- Need to be resided in secondary or Tertiary memory

Disk Storage Devices

- Preferred secondary storage device for high storage capacity and low cost.
- Data are 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 or more
- A track is divided into fixed size sectors and then into blocks
  - Typical block sizes range from B=512 bytes to B=4096 bytes
  - Whole blocks are transferred between disk and main memory for processing
- Track -> sectors -> blocks

Disk Storage Devices

Figure 17.1
(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.



Disk Storage Devices

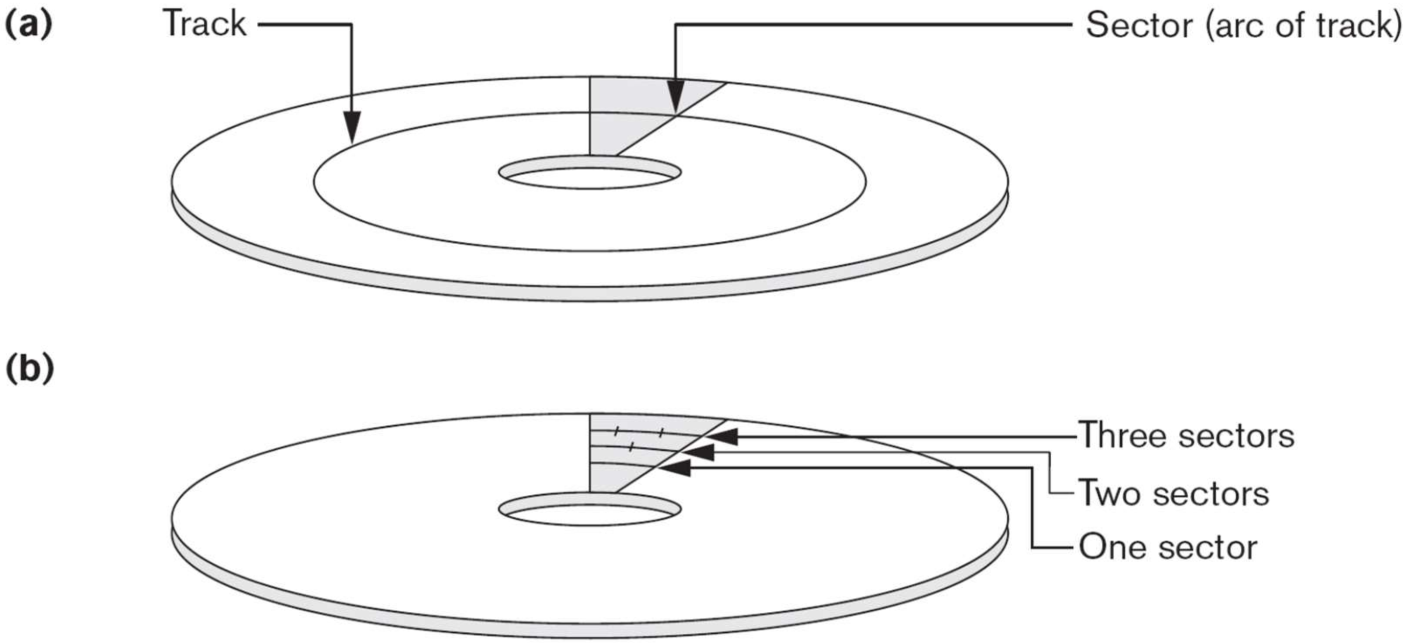


Figure 17.2
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

Disk Storage Devices

- A read-write head moves to the track that contains the block to be transferred
  - Disk rotation moves the block under the read-write head for reading or writing
- To access a physical disk block:
  - to the identified track number (seek time, e.g., 3 to 8ms)
  - the block number (within the cylinder) (rotational delay, e.g., 2ms)
  - and get the block data (transfer delay)
- Disk access delay = seek time + rotational delay + transfer delay

- Reading or writing a disk block is time consuming because of the seek time and rotational delay (latency)

Double Buffer to Reduce Access Delay

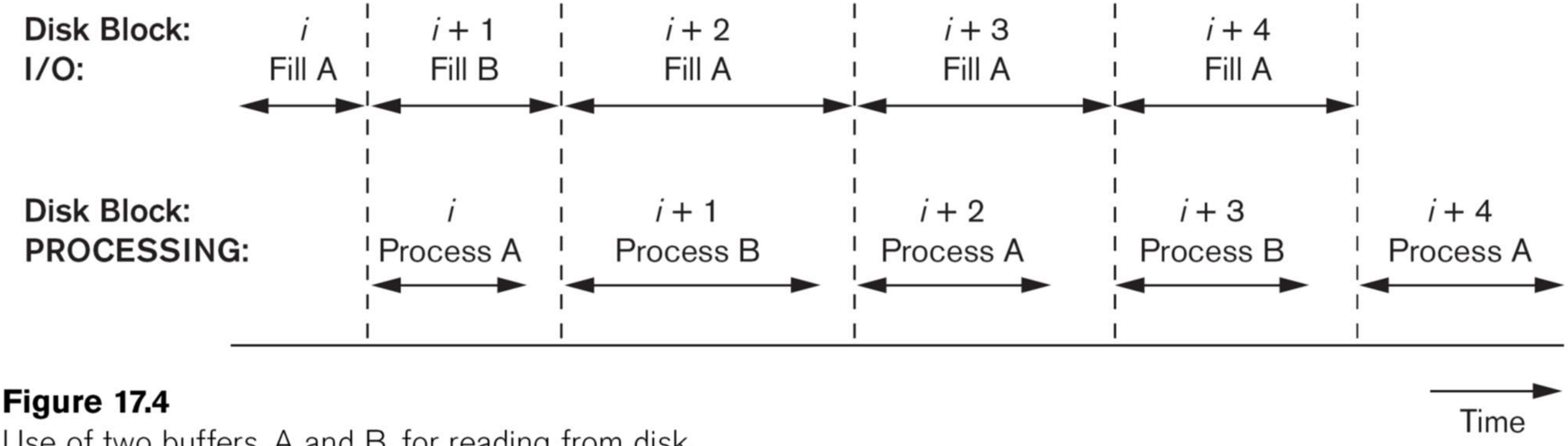


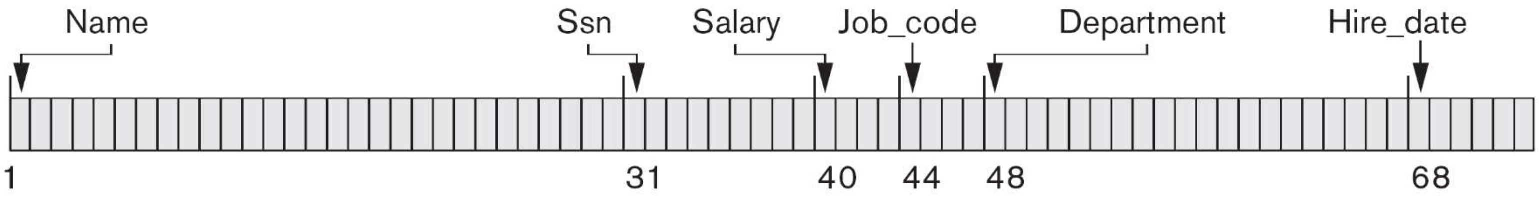
Figure 17.4
Use of two buffers, A and B, for reading from disk.

Files of Records

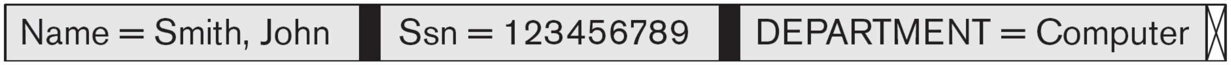
- Database: data file (records) + meta-data
- A data file (e.g., table) is a *sequence* of records (e.g., tuples), where each record is a collection of data values (fields).
- A file descriptor (or file header) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk

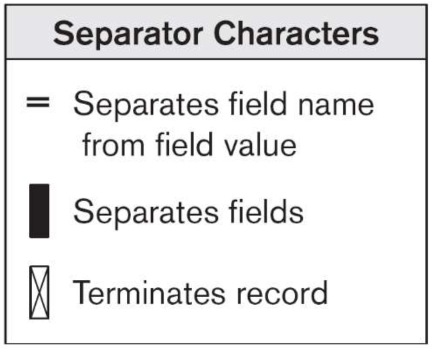
Records

- Fixed length records:
  - Every record in the file has exactly the same size (in bytes)



- Variable length records :
  - If different records in the file have different sizes
  - Separator characters or length fields are needed so that the record can be "parsed"





Separator Characters	
=	Separates field name from field value
▮	Separates fields
☒	Terminates record

Files of Records

- When the block size is larger than the record size, each block will contain numerous records.

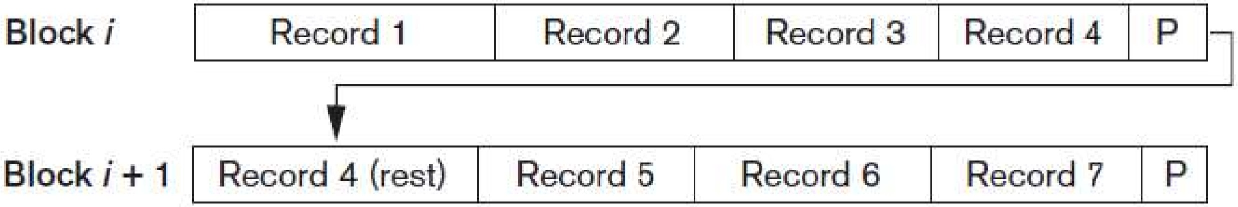
- Blocking: Refers to storing a number of records into one block on the disk

- Blocking factor (bfr) refers to the number of records per block
- Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $bfr=\lfloor B/R \rfloor$ records per block, where the $\lfloor x \rfloor$(floor function) rounds down the number x to an integer.

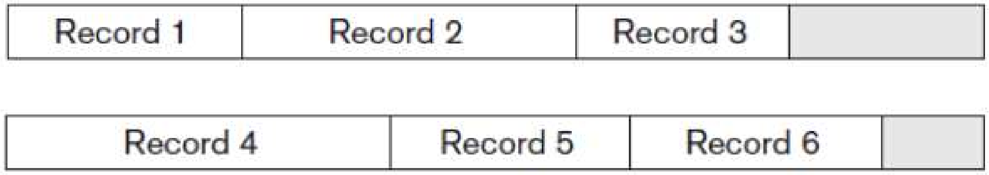
- There may be empty space in a block if an integral number of records do not fit into one block: $B-\lfloor B/R \rfloor *R$

Files of Records

- File records can be unspanned or spanned
  - Spanned: a record can be stored in more than one block



  - Unspanned: no record can span two blocks



- In a file of fixed-length records, unspanned blocking is usually used.
- For variable-length records, either a spanned or an unspanned organization can be used. it is advantageous to use spanning to reduce the lost space in each block.

Typical Operations on Files

- Typical operations on files:
  - OPEN: makes the file ready 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
  - READ_ORDERED: reads the file blocks in order of a specific field of the file

Typical Operations on Files

- Typical operations on files:
  - INSERT: inserts a new record into the file, and makes it the current file record
  - DELETE: removes the current file record from the file, usually by marking the record to indicate that it is no longer valid
  - 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
  - MODIFY: changes the values of some fields of the current file record
  - CLOSE: terminates access to the file

Unordered Files

- Also called a heap file (records are unordered)

- New records are inserted at the end of the file
  - Arranged in their insertion sequence
  - record insertion is efficient (add to the end)

- A linear search through the file records is necessary to search for a record
  - For a file of b blocks, this requires reading and searching half the file blocks on average, and is hence quite expensive (b/2)
  - Worst case, all blocks (b)

9	16	50	2	10	4	8	12	60	100

Ordered Files

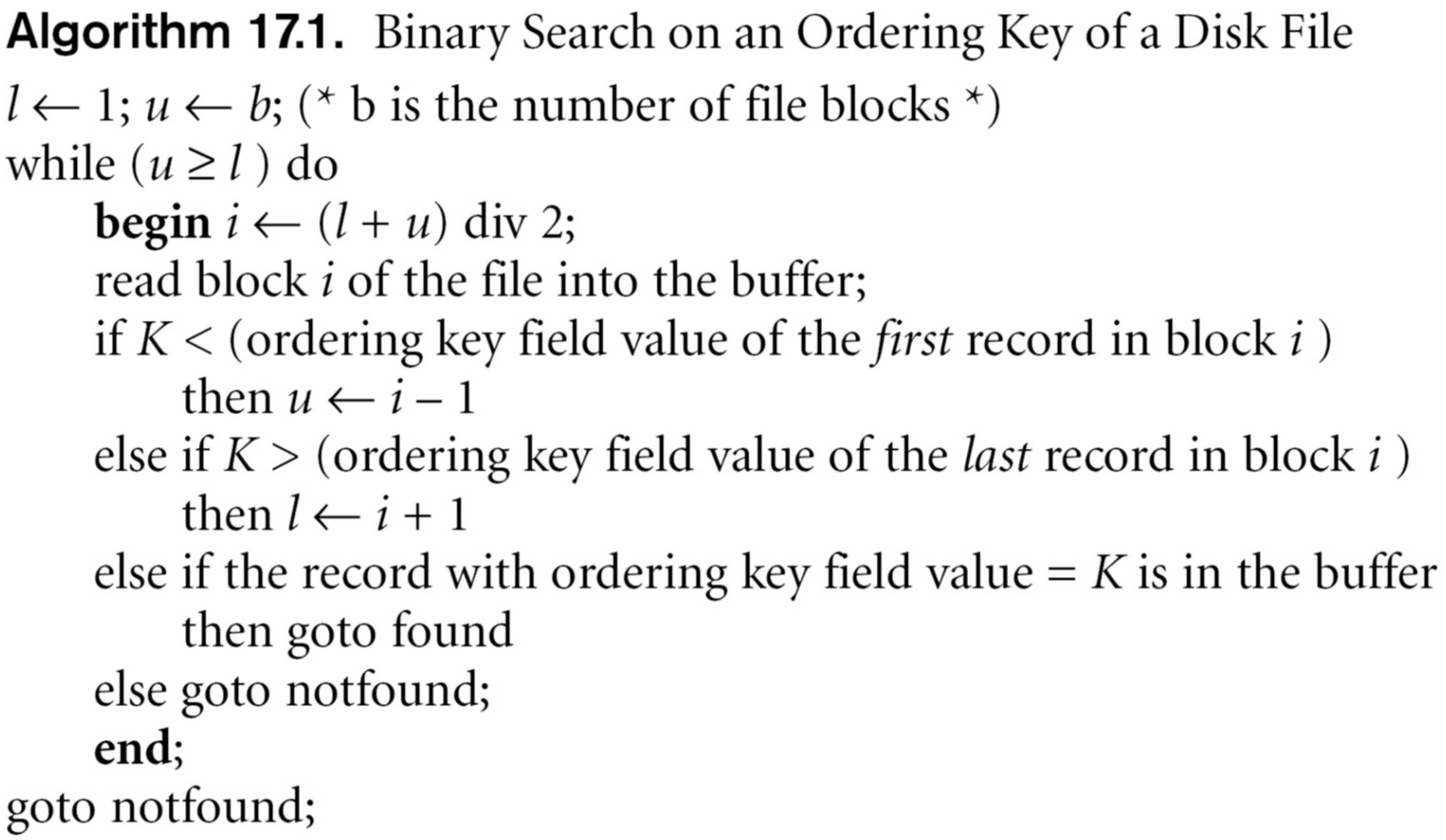
- Also called a sequential file (records are ordered)
- File records are kept sorted by the values of an *ordering field*

- Reading the records in order of the ordering field is quite efficient
- A binary search can be used to search for a record on its *ordering field* value
  - This requires reading and searching $\log_2 b$ of the file blocks on the average, an improvement over linear search

- Insertion is expensive: records must be inserted in the correct order
  - It is common to keep a separate unordered *overflow* file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.

2	4	8	9	10	12	16	50	60	100

Binary Search

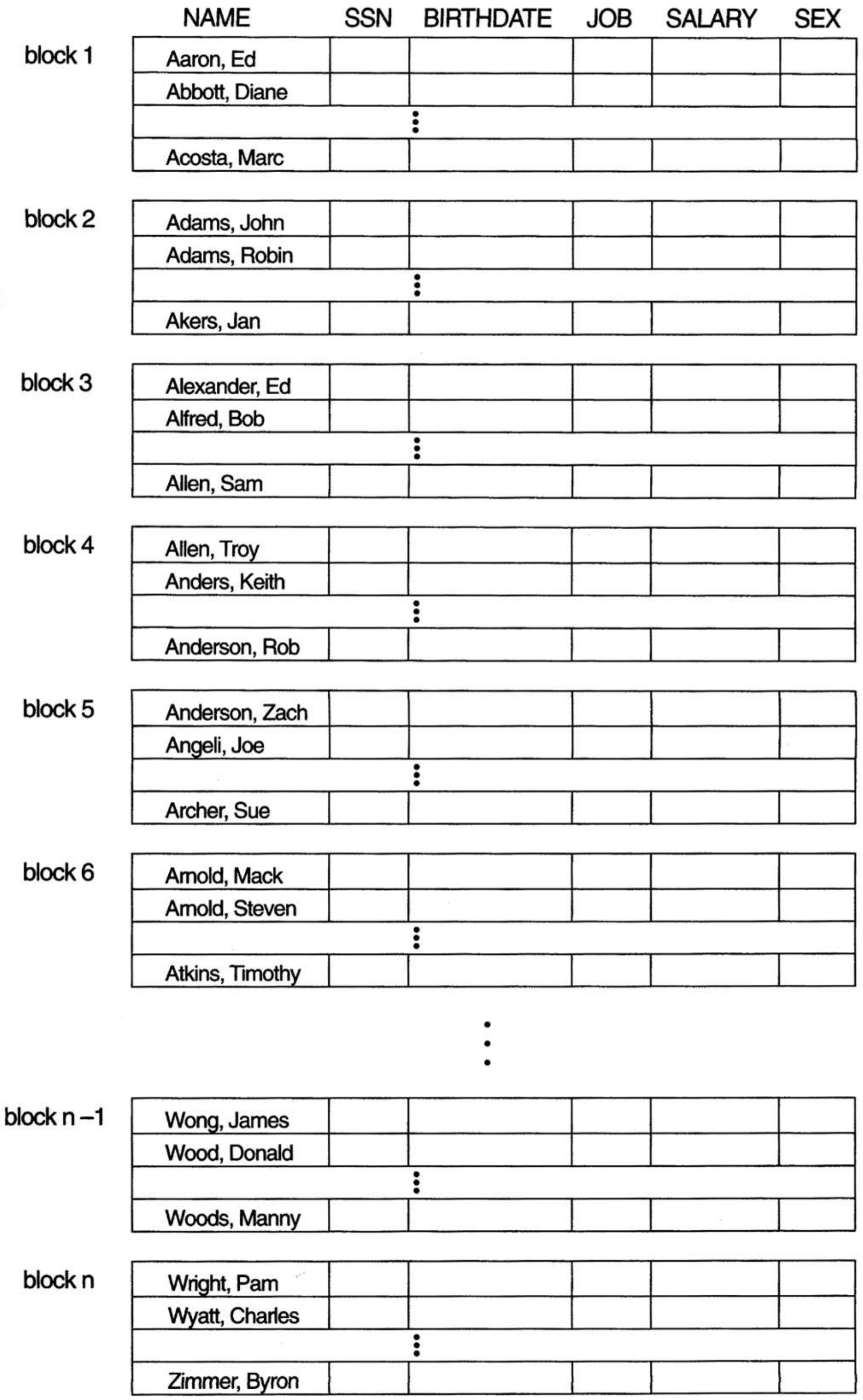


Algorithm 17.1. Binary Search on an Ordering Key of a Disk File

```
$l \leftarrow 1$; $u \leftarrow b$; (* b is the number of file blocks *)
while ($u \geq l$ ) do
    begin $i \leftarrow (l + u)$ div 2;
    read block $i$ of the file into the buffer;
    if $K <$ (ordering key field value of the first record in block $i$ )
        then $u \leftarrow i - 1$
    else if $K >$ (ordering key field value of the last record in block $i$ )
        then $l \leftarrow i + 1$
    else if the record with ordering key field value = $K$ is in the buffer
        then goto found
    else goto notfound;
    end;
goto notfound;
```

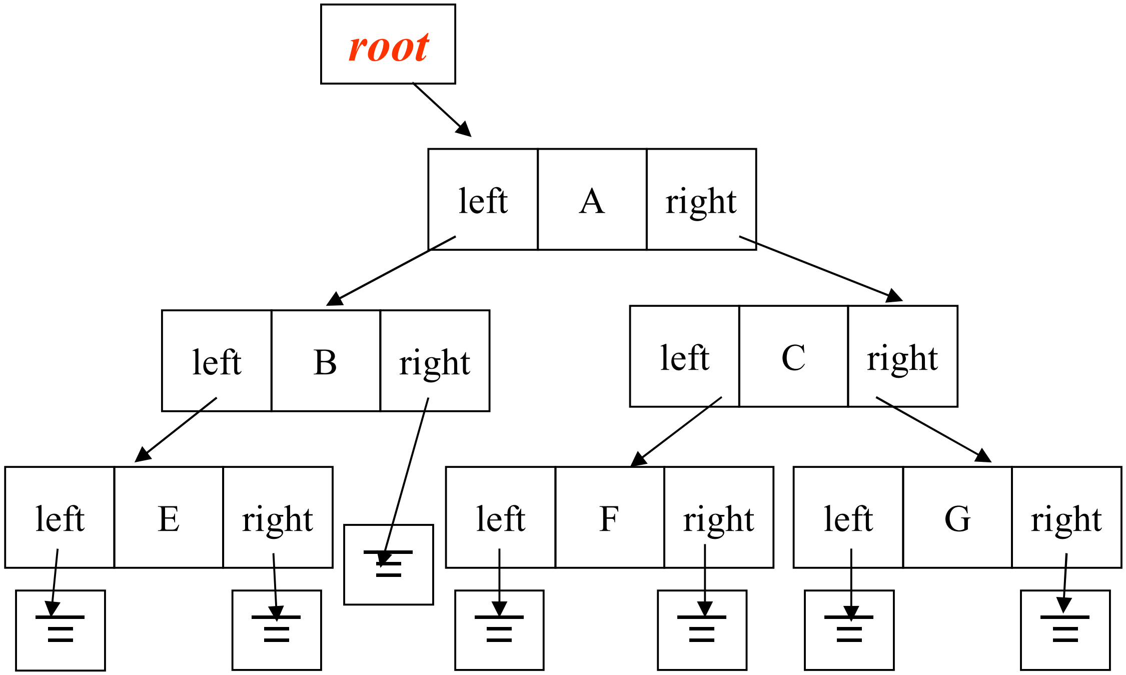
Ordered Files

A **binary search** for disk files can be done on the blocks rather than on the records, because a block is read together.



	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 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
⋮						
block n –1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

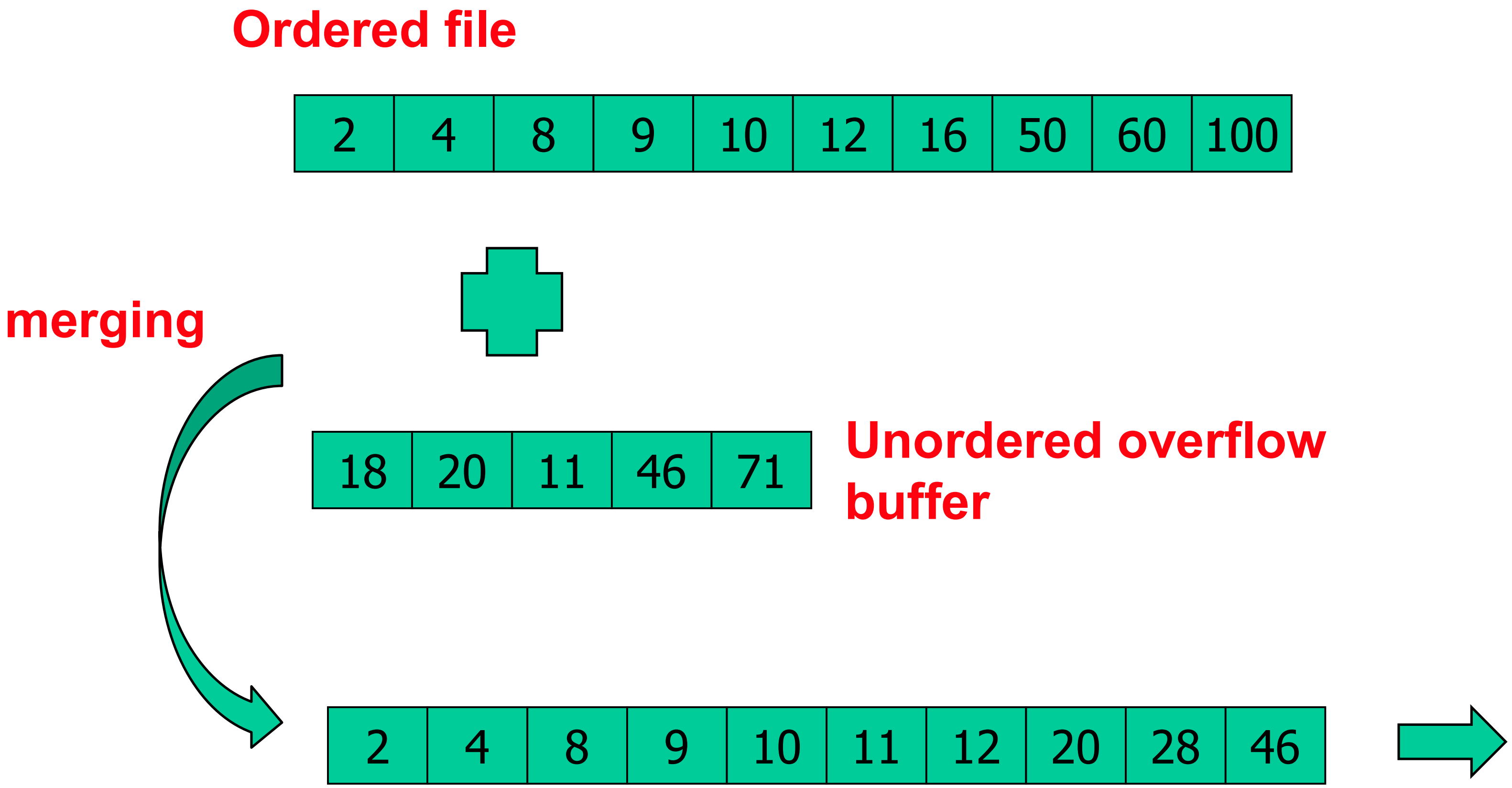
Binary Tree Data Structures



```
class TreeNode
{
private:
        int info;
        TreeNode* left;
        TreeNode* right;
};
class Mytree
{
private:
        TreeNode* root;
}
```

- *Binary Tree*: every node has two links (pointers)

Ordered Files



Ordered file

2	4	8	9	10	12	16	50	60	100

merging

Unordered overflow buffer

18	20	11	46	71

2	4	8	9	10	11	12	20	28	46

Average Access Times

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

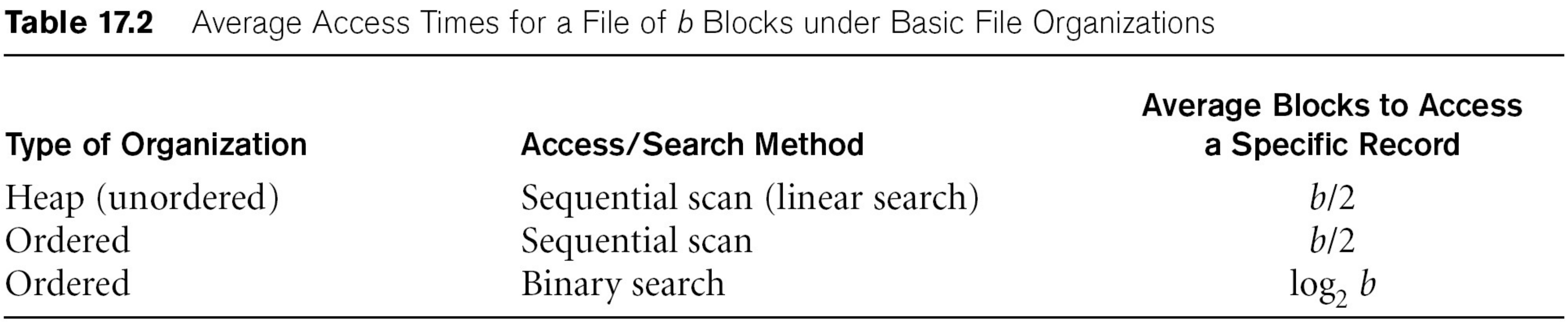


Table 17.2 Average Access Times for a File of *b* Blocks under Basic File Organizations

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 (files on disk)

- The file blocks are divided into M equal-sized buckets, numbered bucket_0, bucket_1, ..., $\text{bucket}_{M-1.}$ A bucket is either one disk block or a cluster of contiguous disk blocks.

- One of the file fields is designated to be the hash key of the file
- Suppose there is a hash function h(K) that takes a hash key K as an input to compute an integer in the range 0 to B – 1 where B is the number of buckets
  - h(K) => 0 to B – 1

- A table maintained in the file header converts the bucket number into the corresponding disk block address.

Hashed Files

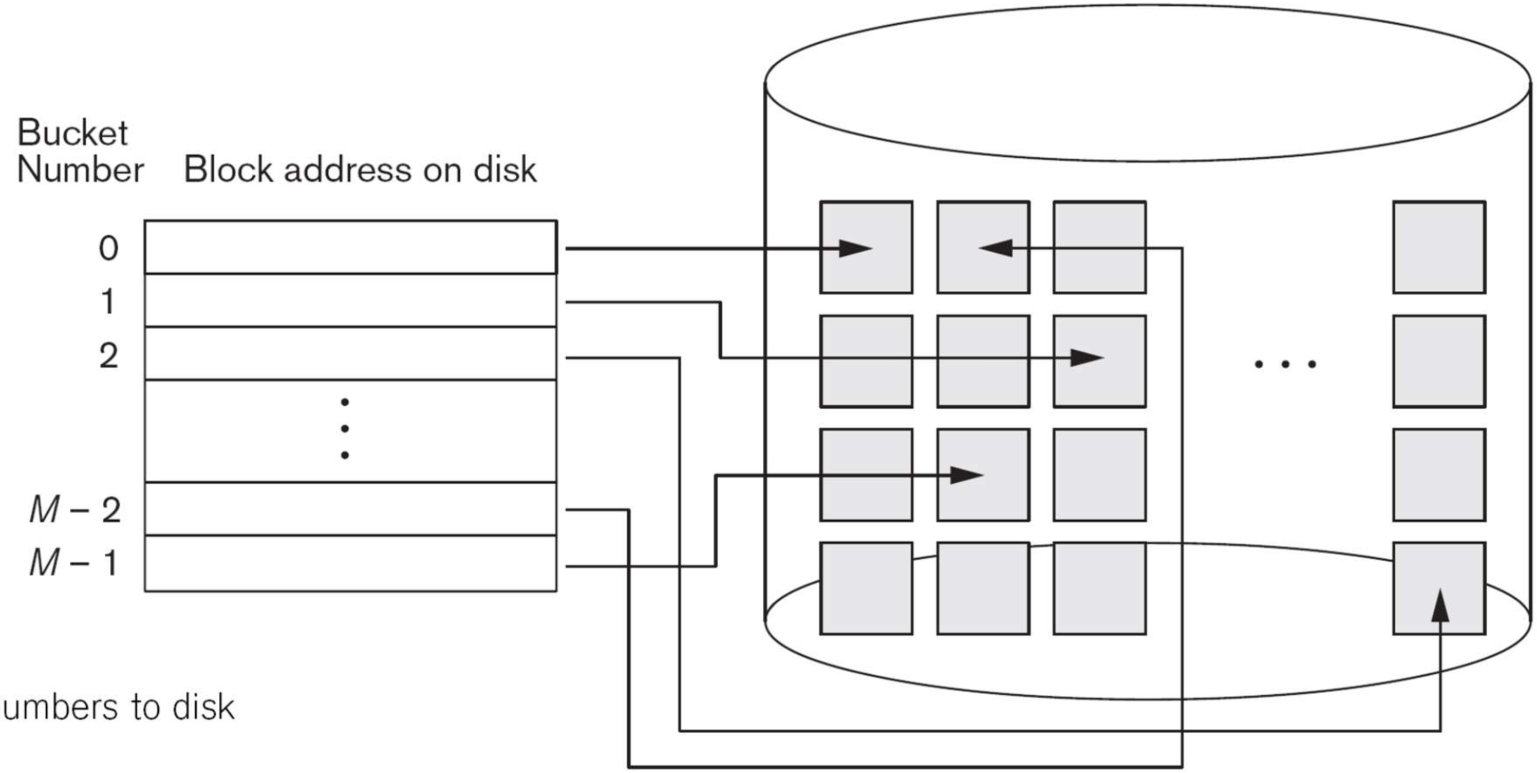
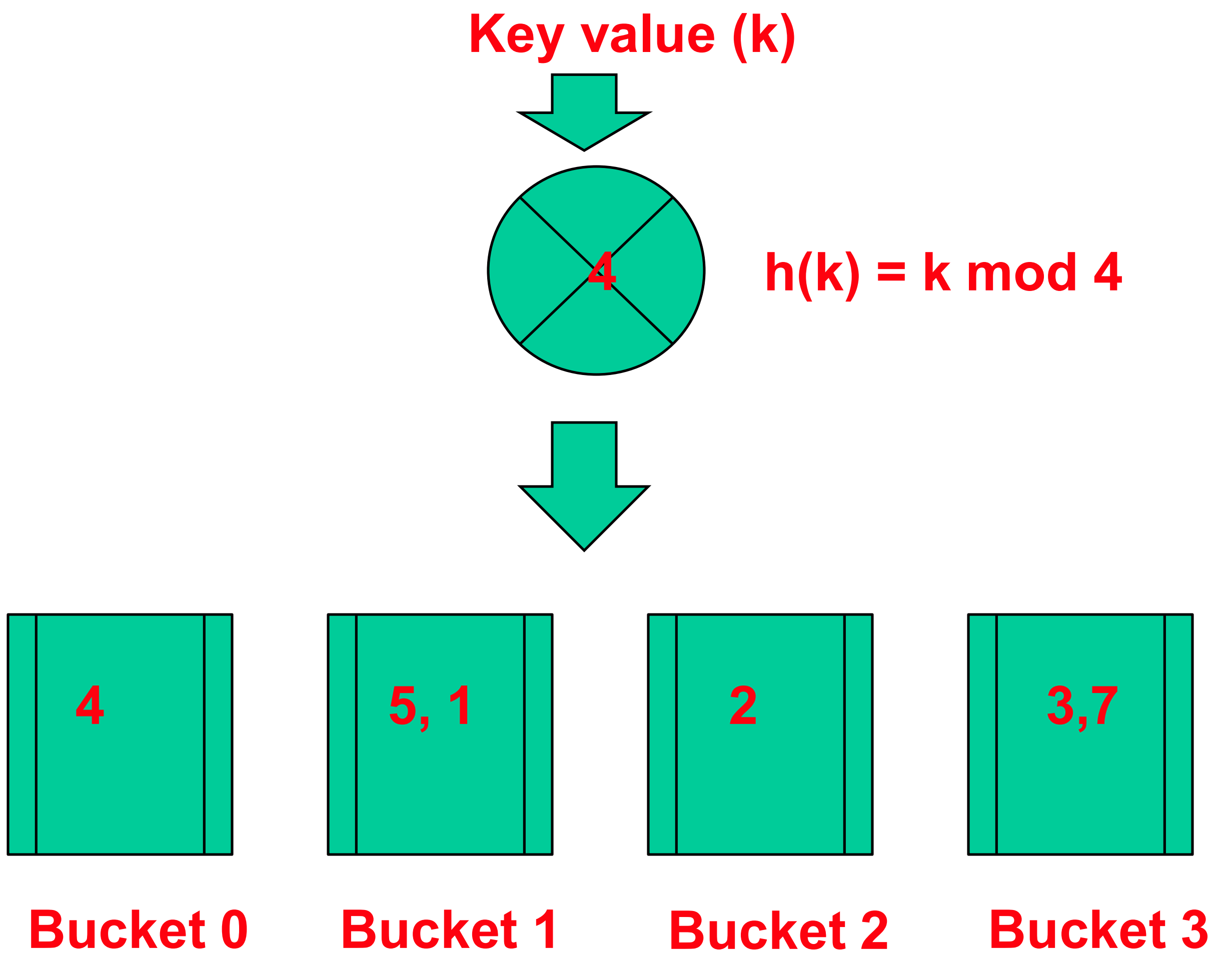


Figure 17.9
Matching bucket numbers to disk block addresses.

Hashed Files



Key value (k)

4

h(k) = k mod 4

4

5, 1

2

3,7

Bucket 0

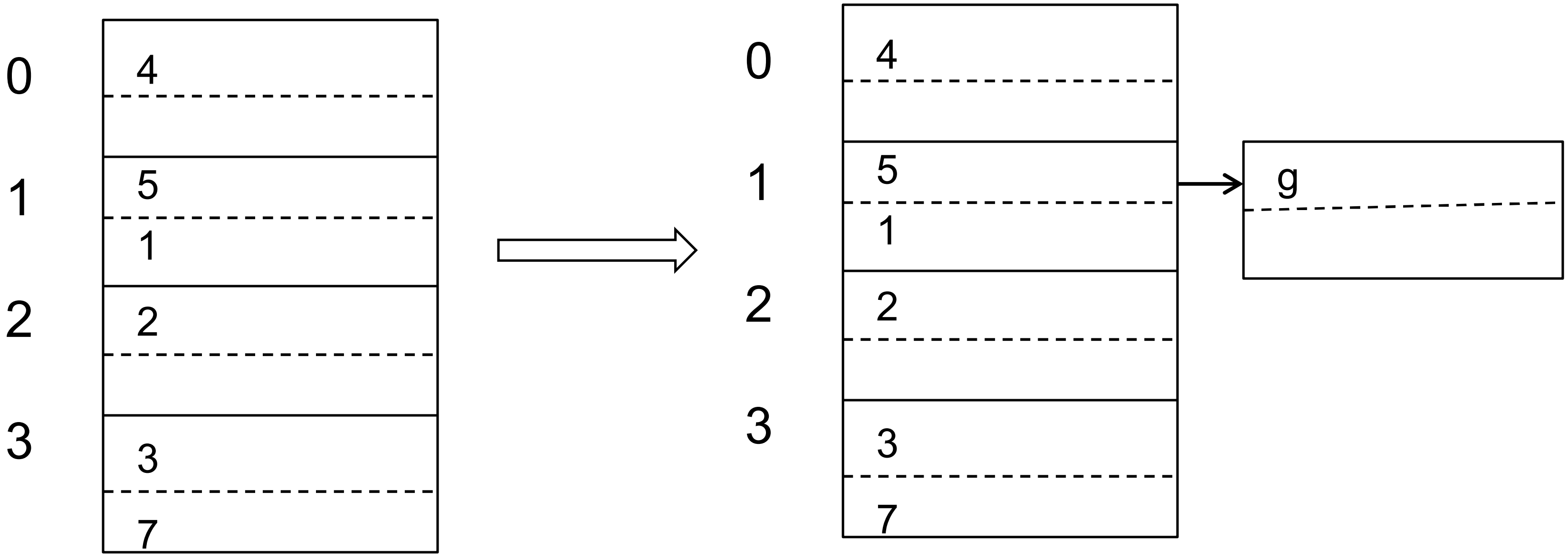
Bucket 1

Bucket 2

Bucket 3

Hashed Files – Collision Resolution

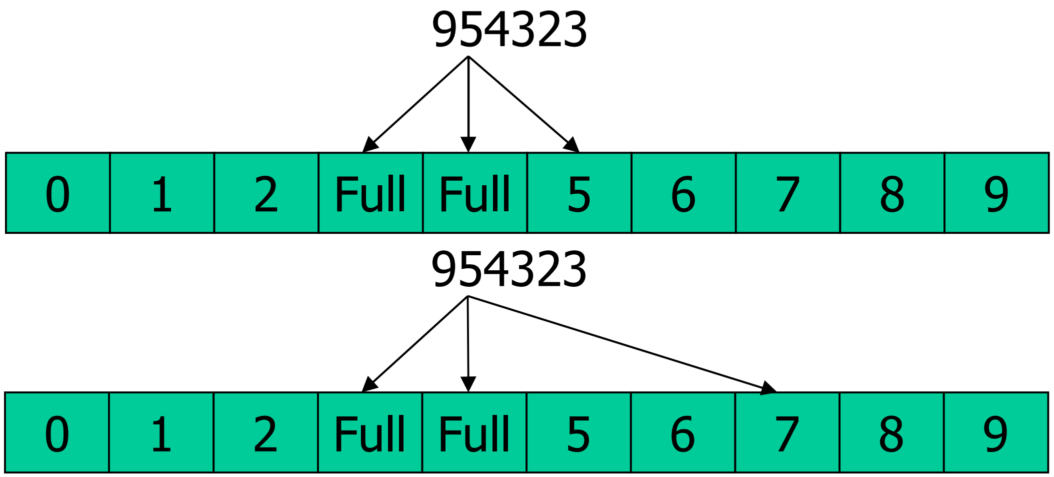
- We assume a block can hold two records and B = 4, i.e., the hash function h(k)=k mod 4 returns values from 0 to 3
- Suppose we add to the hash table a record with key 9 and h(9) = 1. Then we add the new record to the bucket numbered 1
- Collisions occur when a new record hashes to a bucket that is already full



Hashed Files – Collision Resolution

- There are numerous methods for collision resolution:
  - 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.
    - *Linear Probing*
      - If collide, try Bucket_id+1, Bucket_id+2, …, Bucket_id+n
    - *Quadratic Probing*
      - If collide, try Bucket_id+1, Bucket_id+4,…, Bucket_id+n^2
  - What are the differences in performance?

Hashed Files – Collision Resolution



954323

0	1	2	Full	Full	5	6	7	8	9

Linear Probing

954323

0	1	2	Full	Full	5	6	7	8	9

Quadratic Probing

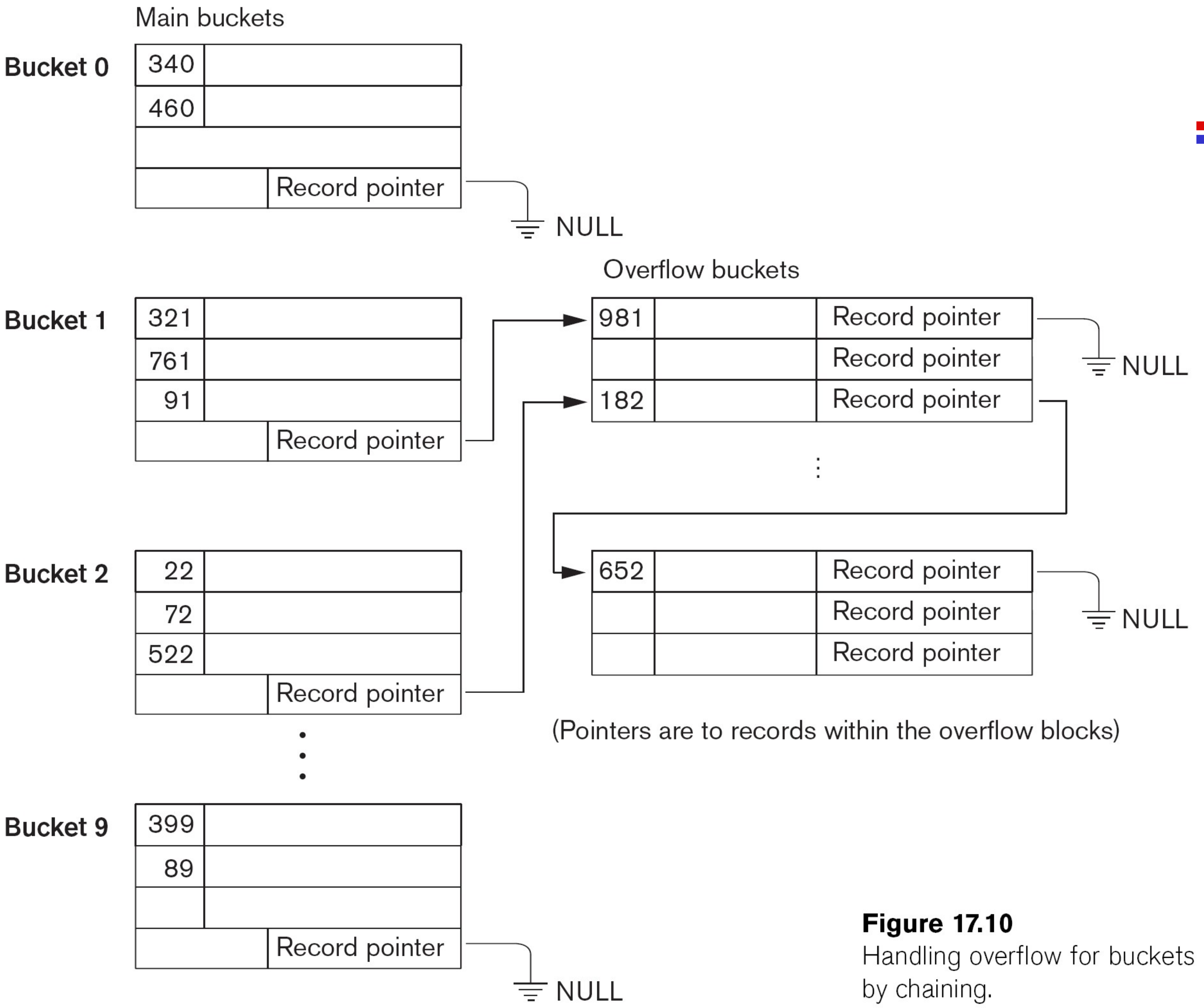
Assume h(x)=x%10 (take the last digit), and every slot is a bucket

Hashed Files Collision Resolution

- Chaining:
  - For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions
  - In addition, a pointer field is added to each bucket
  - A collision is resolved by placing the new record in an unused overflow bucket and setting the pointer of the occupied hash address bucket to the address of that overflow bucket

- 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 - Overflow Handling



(Pointers are to records within the overflow blocks)

Figure 17.10
Handling overflow for buckets by chaining.

Hashed Files

- To reduce the number of overflow, 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
  - Searching overflow records are more expensive
- 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

Extendible and Dynamic Hashing

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are extended to allow dynamic growth and shrinking of the number of buckets.
  - These techniques include the following: dynamic hashing and extendible hashing

- Both dynamic and extendible hashing use an access structure of the hash value h(K) , which is called directory
  - In dynamic hashing the directory is a binary tree
  - In extendible hashing the directory is an array

Extendible and Dynamic Hashing

- The directories can be stored on disk, and they expand or shrink dynamically
  - Directory entries point to the disk blocks that contain the stored records
- An insertion in a bucket that is full causes the bucket to split into two buckets and the records are redistributed among the two buckets
  - The directory is updated appropriately
  - Dynamic and extendible hashing do not require an overflow area in general

Extendible Hashing

- A directory consisting of an array of 2^d bucket addresses is maintained
- d is called the global depth of the directory
- The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry and the address in that entry determines the bucket storing the records
- A local depth d' stored with each bucket specifies the number of bits on which the bucket contents are based.
- The value of d' can be increased or decreased by one at a time to handle overflow or underflow respectively

Extendible Hashing

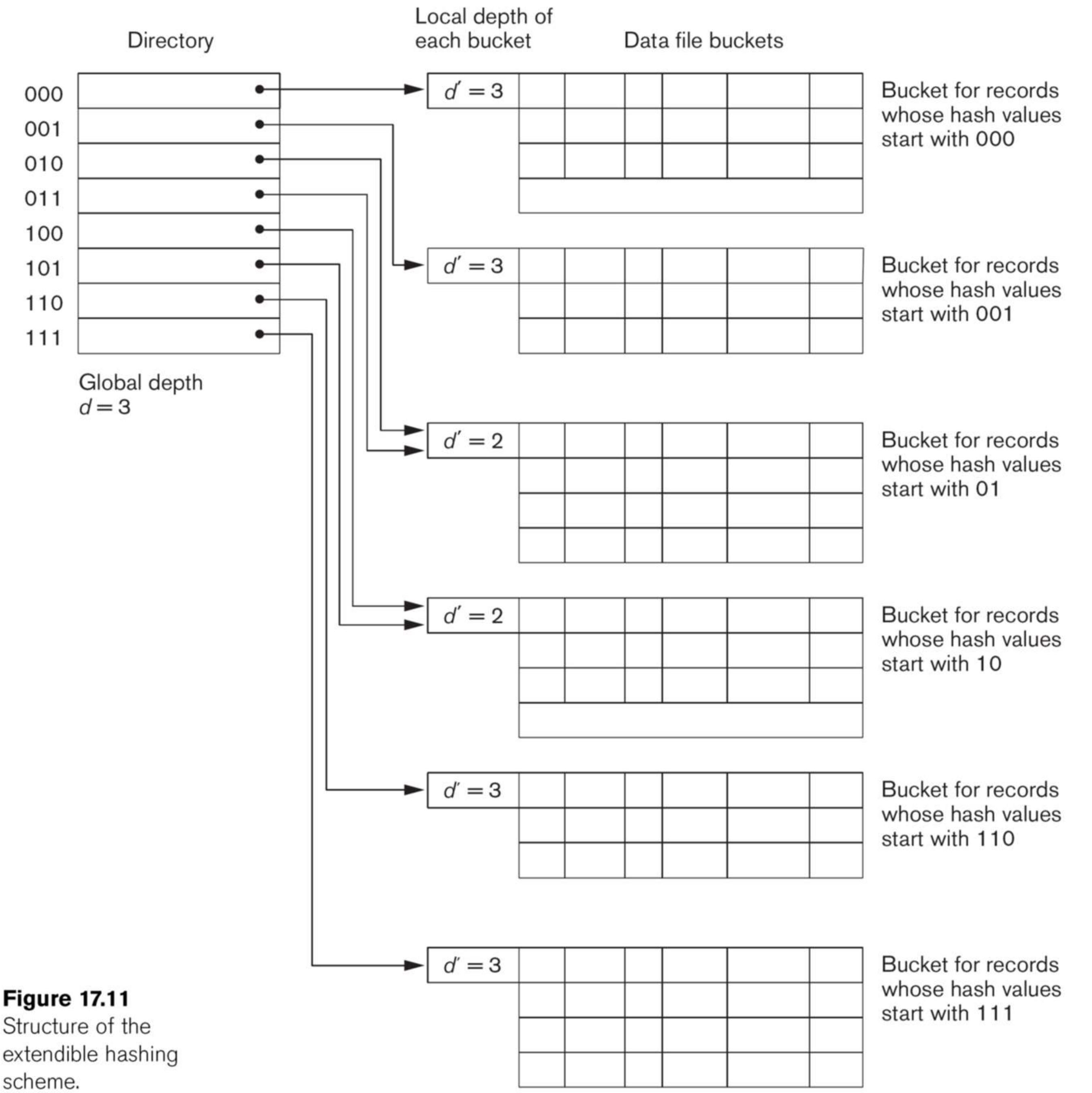
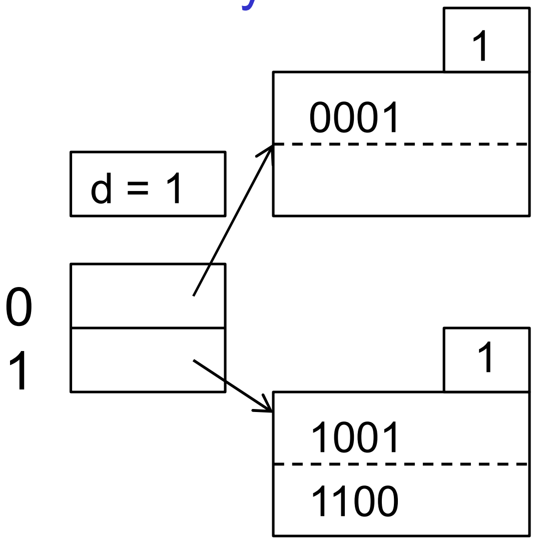

Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth
$d = 3$
$d' = 3$
Bucket for records whose hash values start with 000
$d' = 3$
Bucket for records whose hash values start with 001
$d' = 2$
Bucket for records whose hash values start with 01
$d' = 2$
Bucket for records whose hash values start with 10
$d' = 3$
Bucket for records whose hash values start with 110
$d' = 3$
Bucket for records whose hash values start with 111

Figure 17.11
Structure of the extendible hashing scheme.

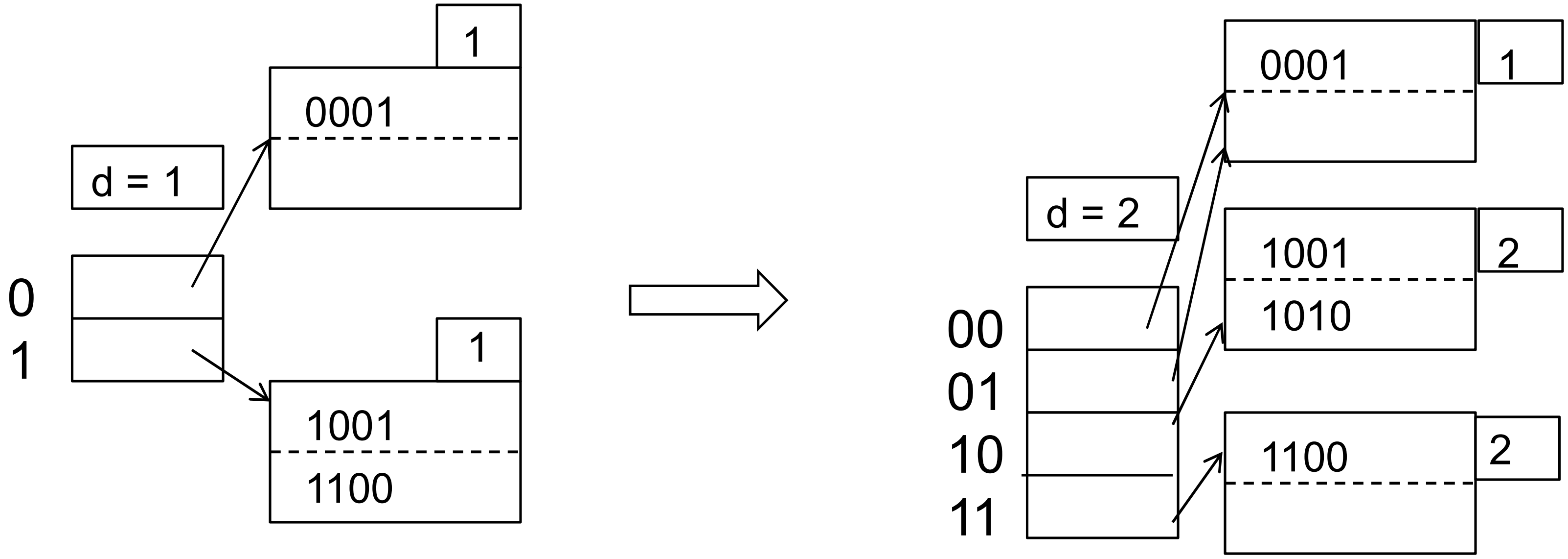
Extendible Hashing Example

- Suppose the hash function produces a sequence of four bits
- At the beginning, only one bit is used as illustrated by d =1
- The bucket array therefore has only two entries and points to two blocks. Each block can hold at most two records.
  - The first holds all the current records whose search keys hash to a bit sequence beginning with 0
  - The second holds all those whose search keys hash to a sequence beginning with 1



Extendible Hashing Example

- Suppose we insert a record whose key hash to the sequence 1010
- Since the first bit is 1, it belongs in the second block. However, the second block is full. It needs to be split into two buckets 10 and 11. The records are redistributed in these two buckets.
- The local depth and global depth are increased.



Dynamic Hashing

- Dynamic hashing maintains tree-structured directory with two types of nodes
  - Internal nodes that have two pointers: the left pointer corresponding to the 0 bit (in the hash address) and a right pointer corresponding to the 1 bit
  - Leaf nodes: these hold a pointer to the actual bucket with records

Dynamic Hashin

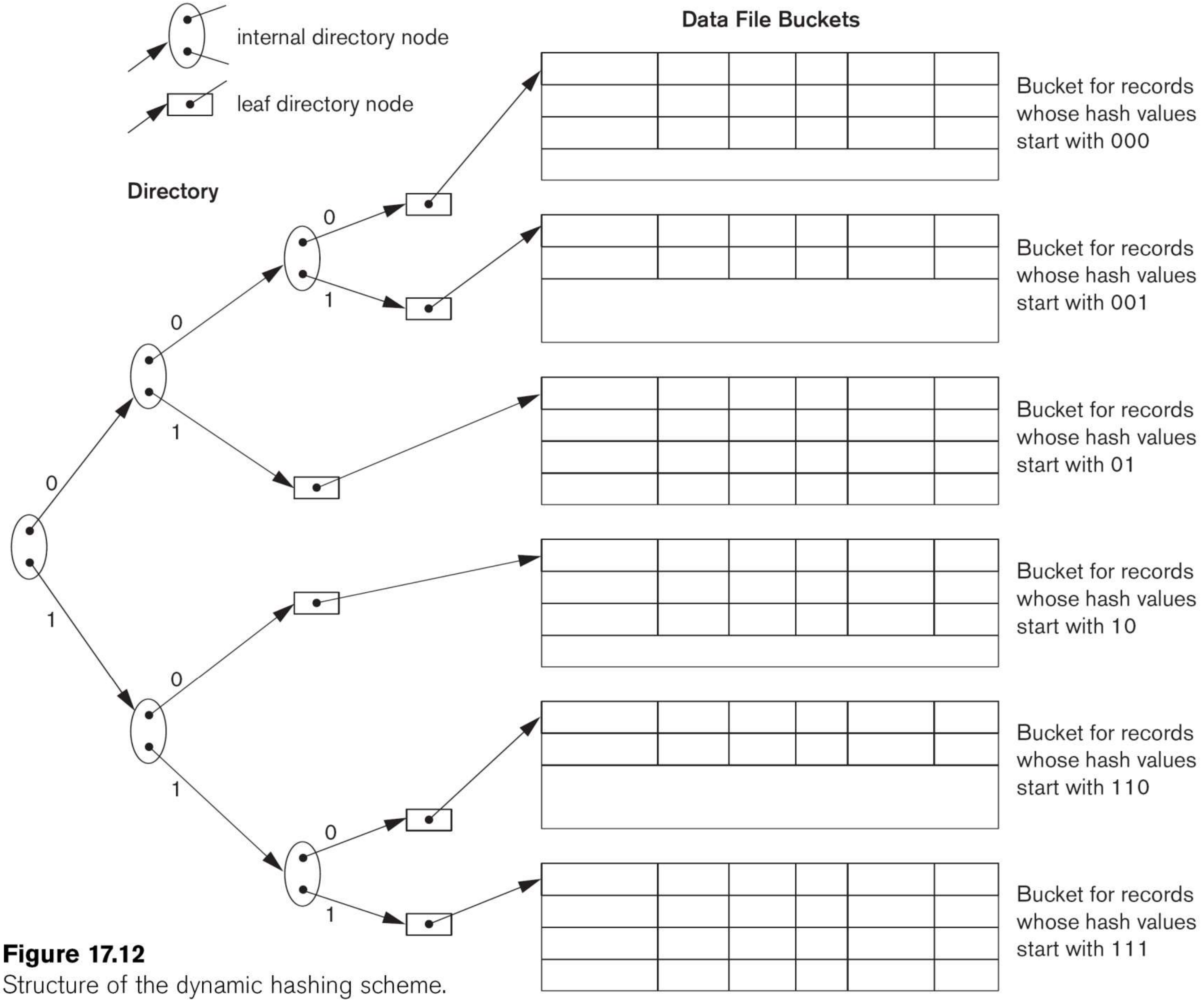

internal directory node
leaf directory node
Directory
Data File Buckets
0
1
0
1
0
1
0
1
0
1
Bucket for records whose hash values start with 000
Bucket for records whose hash values start with 001
Bucket for records whose hash values start with 01
Bucket for records whose hash values start with 10
Bucket for records whose hash values start with 110
Bucket for records whose hash values start with 111

Figure 17.12
Structure of the dynamic hashing scheme.

Reference

- 6e
  - *Ch. 16, pages 565-598*