CS3402 : Chapter 8
Files and Hashed Files

Storage Medium for Databases

- Memory hierarchy
  - CPU cache => main memory => flash memory/Phase change memory => magnetic disks/optical disks
  - Slower in access delay but larger in memory size (less expensive)
  - Volatile vs. non-volatile
- Primary storage
  - The storage media that can be operated directly by the CPU
  - Include main memory and cache memory
- Secondary and tertiary storage
  - Slower in access
  - Include magnetic disks, optical disks and flash memory
- A database could be huge in size (several GB or even larger)
  - Need to be resided in secondary/tertiary memory (persistent data and non-volatile storage)

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

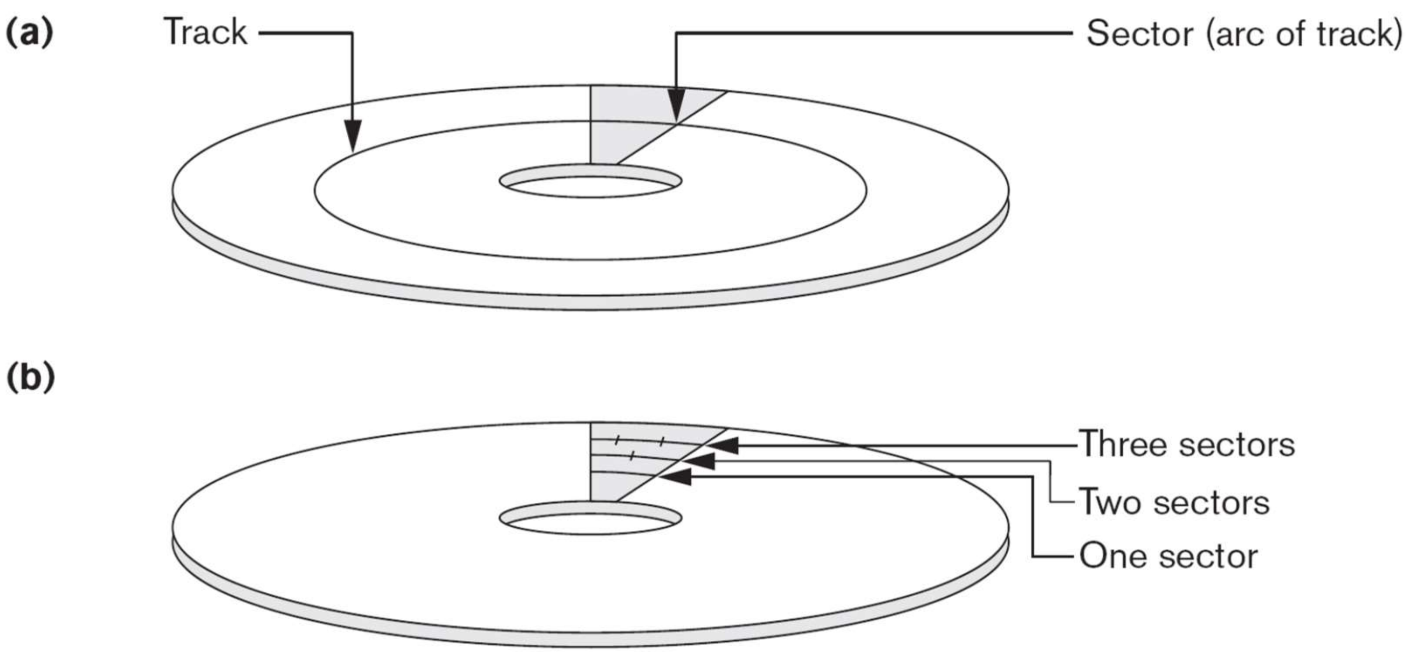
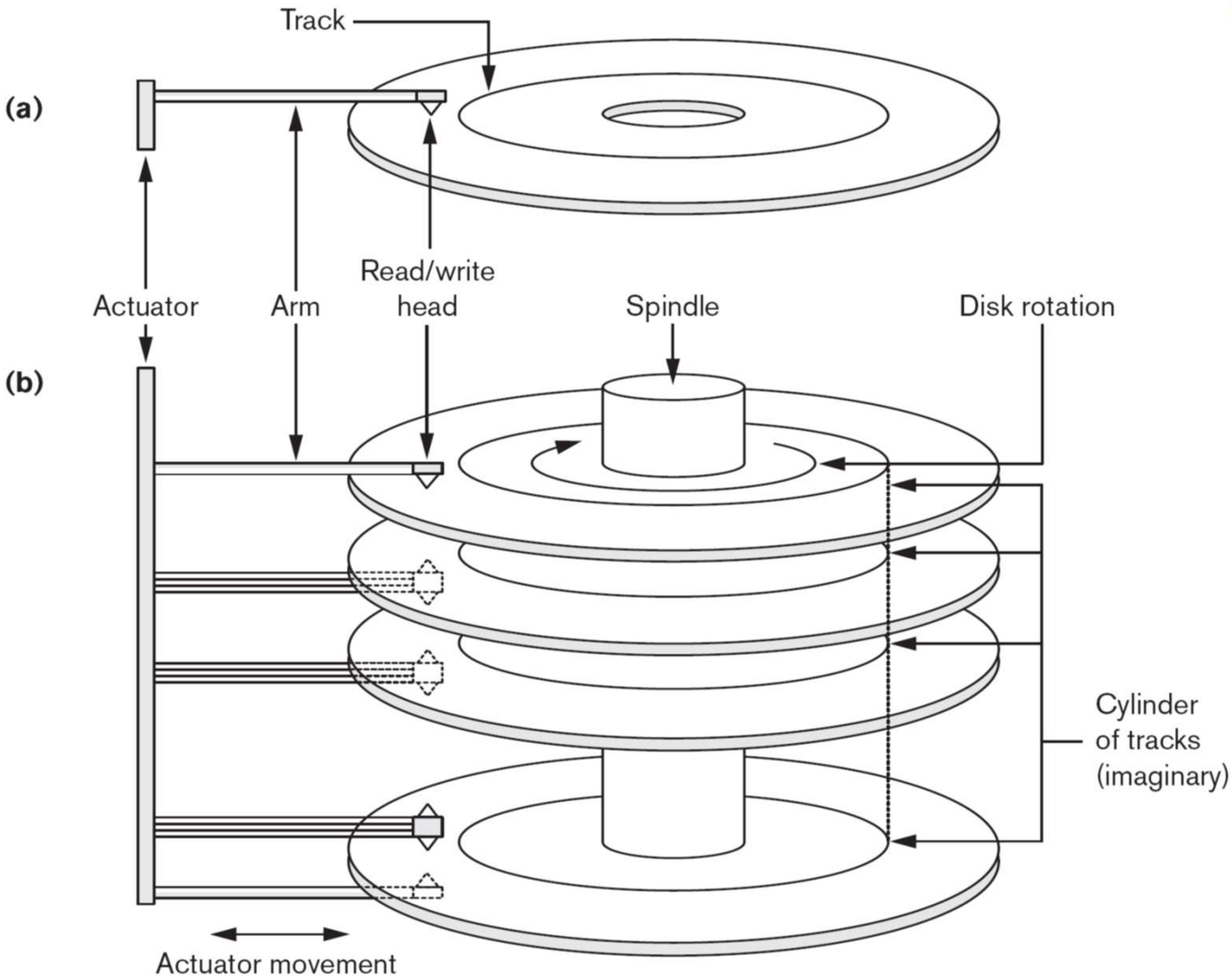


Figure 17.2
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

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

- 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 buffering can be used to speed up the transfer of contiguous disk blocks

Double Buffer to Reduce Access Delay

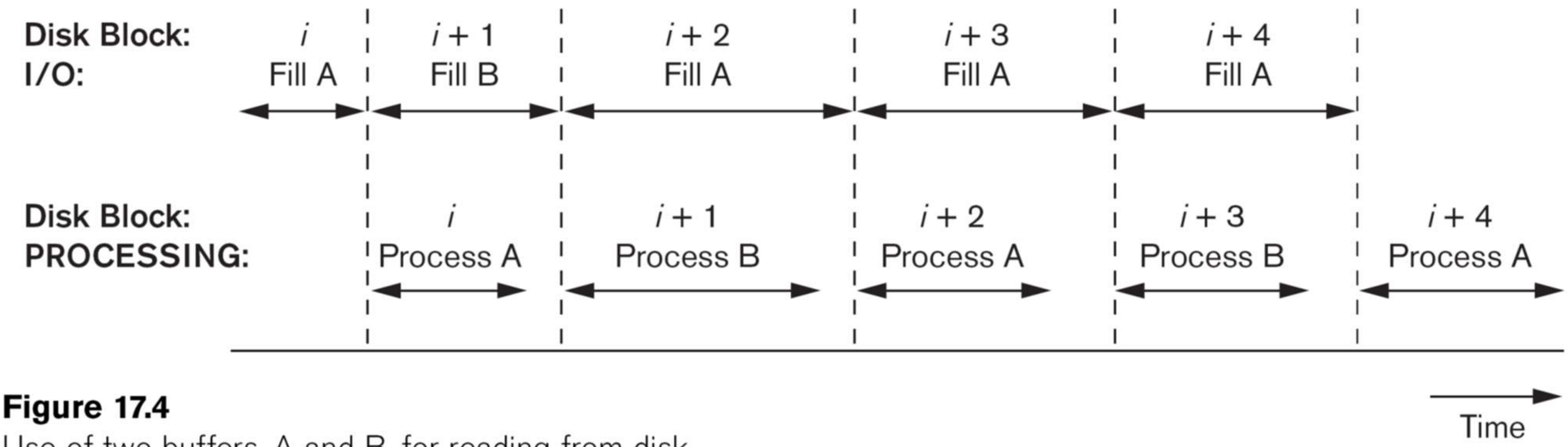
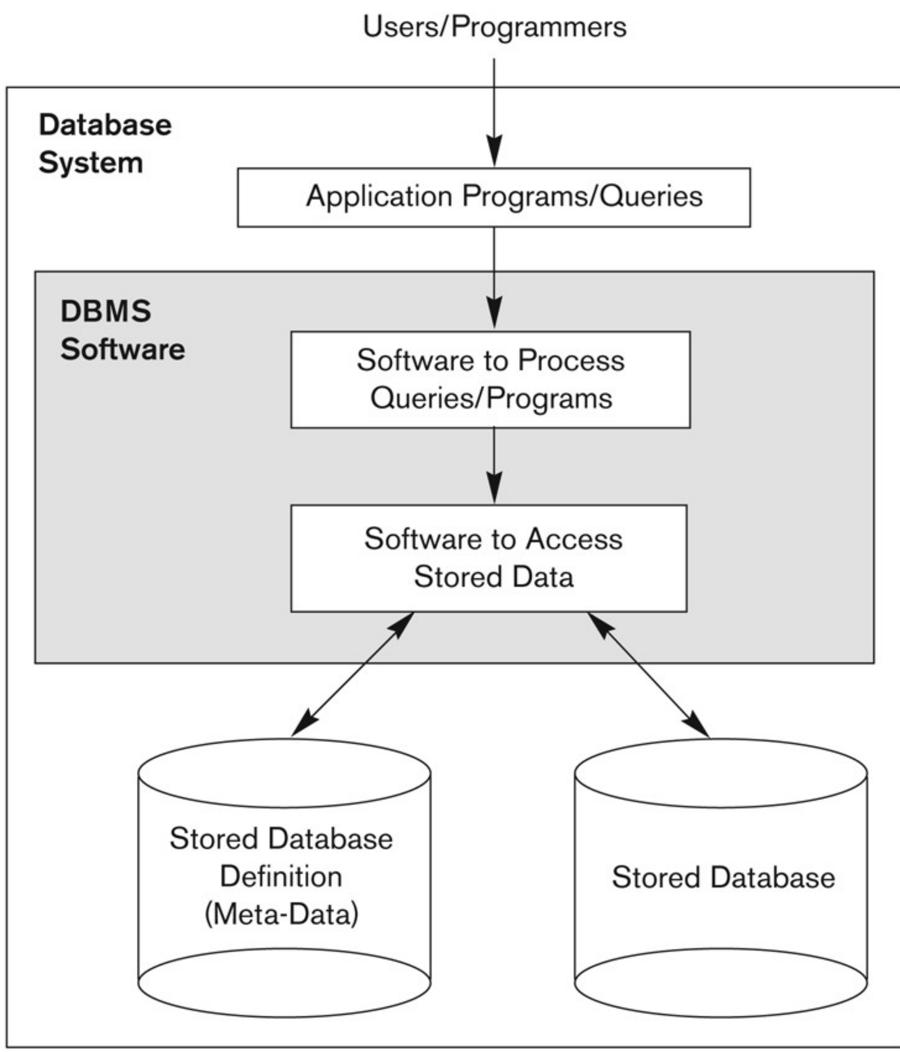


Figure 17.4
Use of two buffers, A and B, for reading from disk.

Database Systems

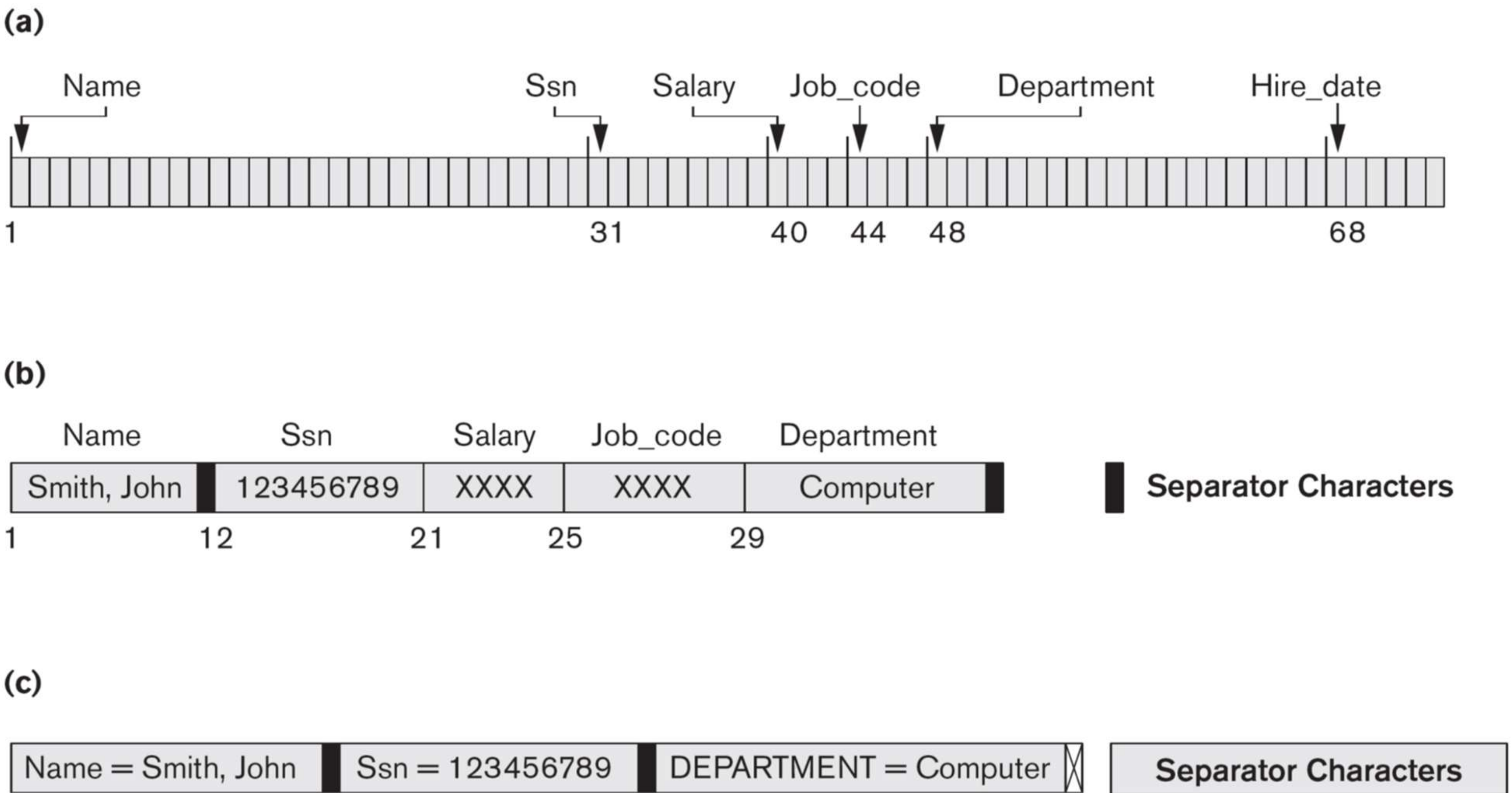


Store the values in files

Figure 1.1
A simplified database system environment.

Records

- Database: data file (records) + meta-data
- Fixed and variable length records
- Records contain fields (attributes)
- Fields may be fixed length or variable length (e.g., Varchar)
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed so that the record can be "parsed"
- 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
- There may be empty space in a block if an integral number of records do not fit into one block



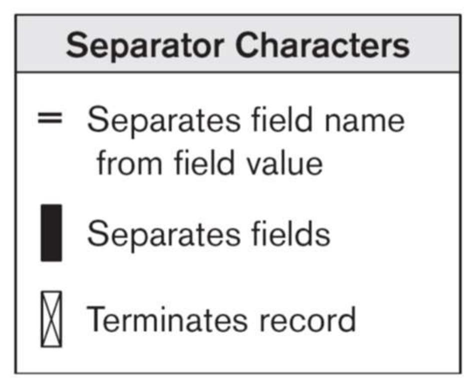


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

Files of Records

- A file (e.g., table) is a *sequence* of records (e.g., tuples), where each record is a collection of data values (fields)
- A file can have fixed-length records or variable-length records
- 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
- File records are stored on disk blocks

Files of Records

- 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 (one by one), linked (using pointers), or indexed (a table to describe their locations)*
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is use
- Files of variable-length records require additional information to be stored in each record, such as separator characters
  - Usually spanned blocking is used with such files

Spanned Vs. Unspanned Files of Records

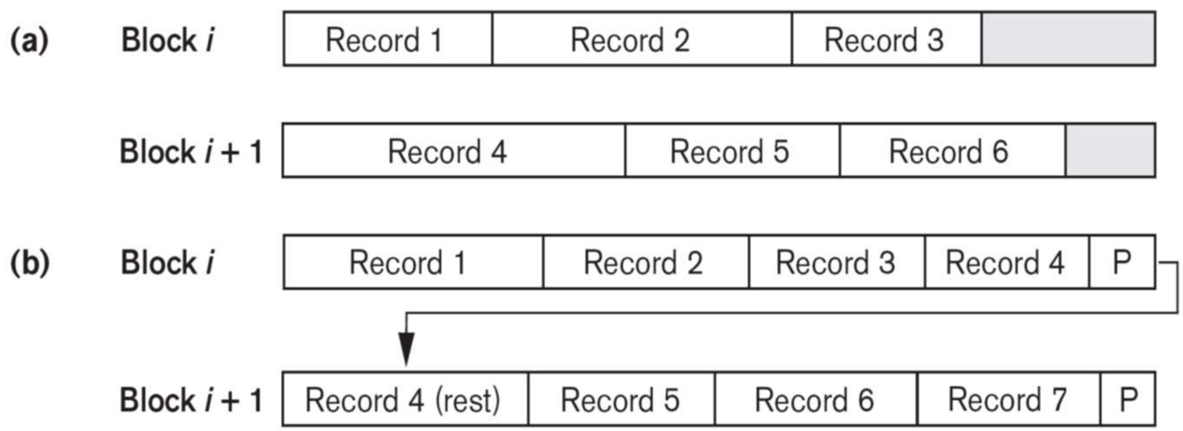


Figure 17.6
Types of record organization.
(a) Unspanned.
(b) Spanned.

Suppose the block size is B. For fixed-length records of size R with B>= R, blocking factor (bfr) = B/R round down
The unused space in each block = B – (bfr x R) bytes

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
  - INSERT: inserts a new record into the file, and makes it the current file record

Typical Operations on Files

- Typical operations on files:
  - DELETE: removes the current file record from the file, usually by marking the record to indicate that it is no longer valid
  - MODIFY: changes the values of some fields of the current file record
  - CLOSE: terminates access to the file
  - REORGANIZE: reorganizes the file records. For example, the records marked “deleted” are physically removed from the file or a new organization of the file records is created
  - READ_ORDERED: reads the file blocks in order of a specific field of 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
- A linear search through the file records is necessary to search for a record
  - This requires reading and searching half the file blocks on average, and is hence quite expensive (n/2)
  - Worst case, all records (n)
- Record insertion is efficient (add to the end)
- Reading the records in order of a particular field requires sorting the file records

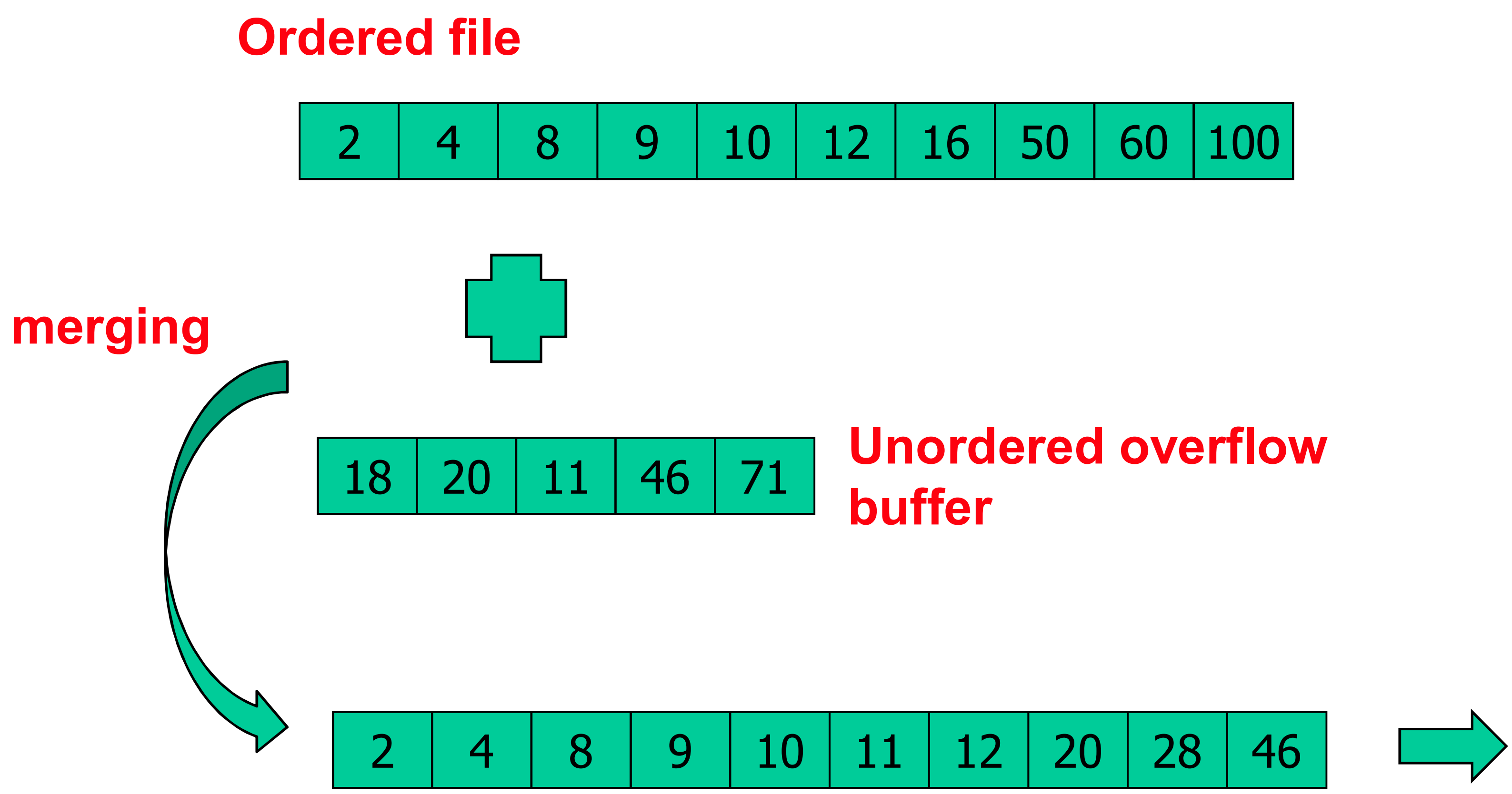
9	16	50	2	10	4	8	12	60	100

Ordered Files

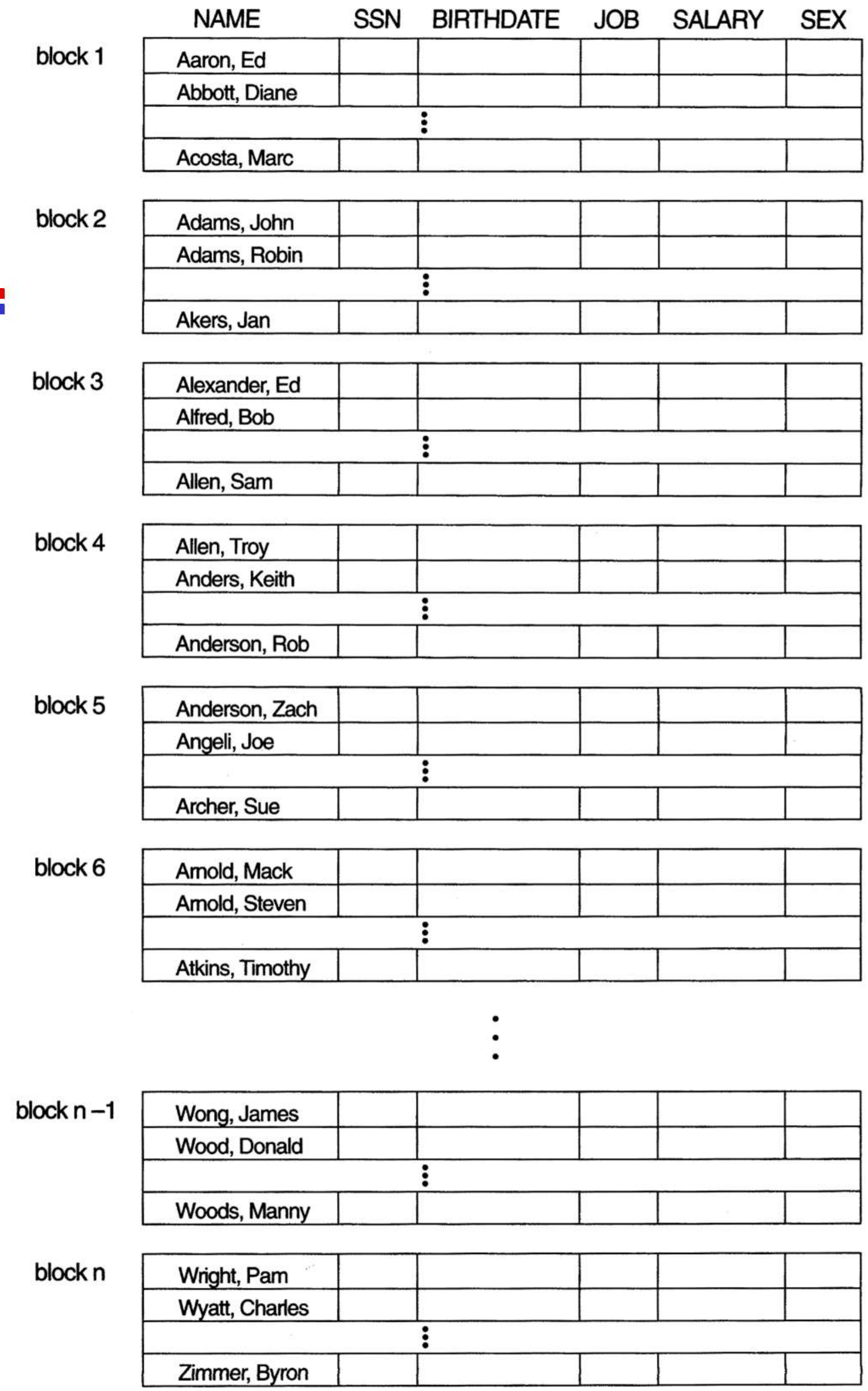
- Also called a sequential file (records are ordered)
- File records are kept sorted by the values of an *ordering field*
- Insertion is expensive: records must be inserted in the correct order
  - It is common to keep a separate unordered *overflow* file for new records to improve insertion efficiency; this is periodically merged with the main ordered file
- A binary search can be used to search for a record on its *ordering field* value
  - This requires reading and searching $\log_2 n$ of the file blocks on the average, an improvement over linear search
- Reading the records in order of the ordering field is quite efficient

2	4	8	9	10	12	16	50	60	100

Ordered Files



Ordered Files

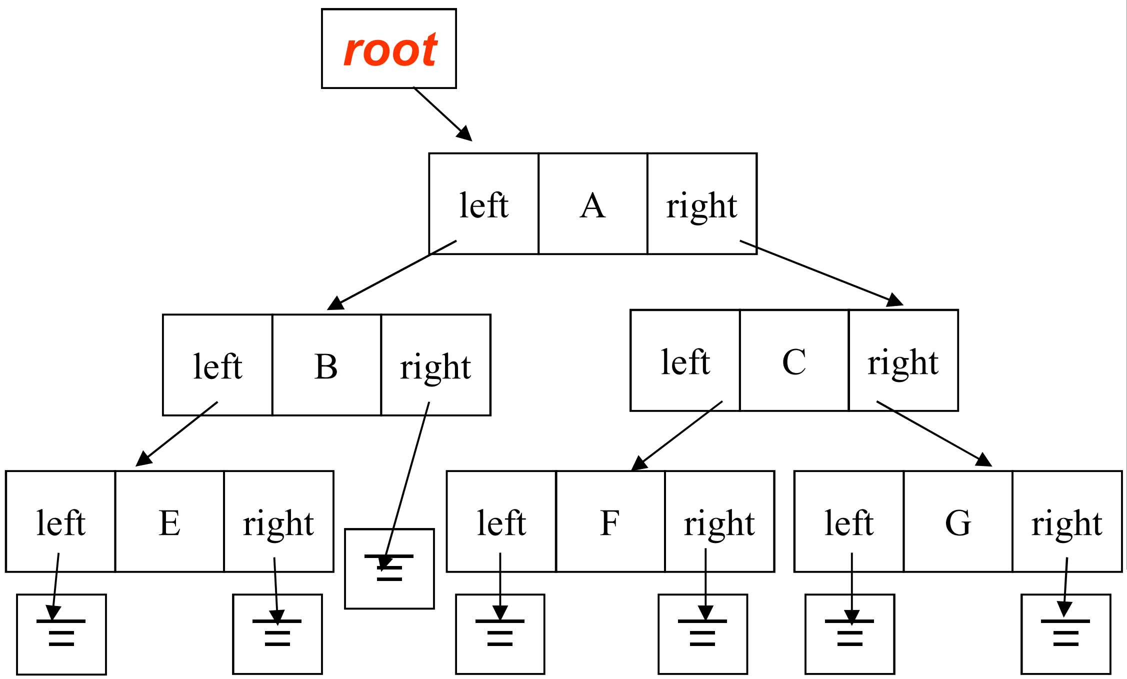


Binary Search

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;

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)

Average Access Times

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

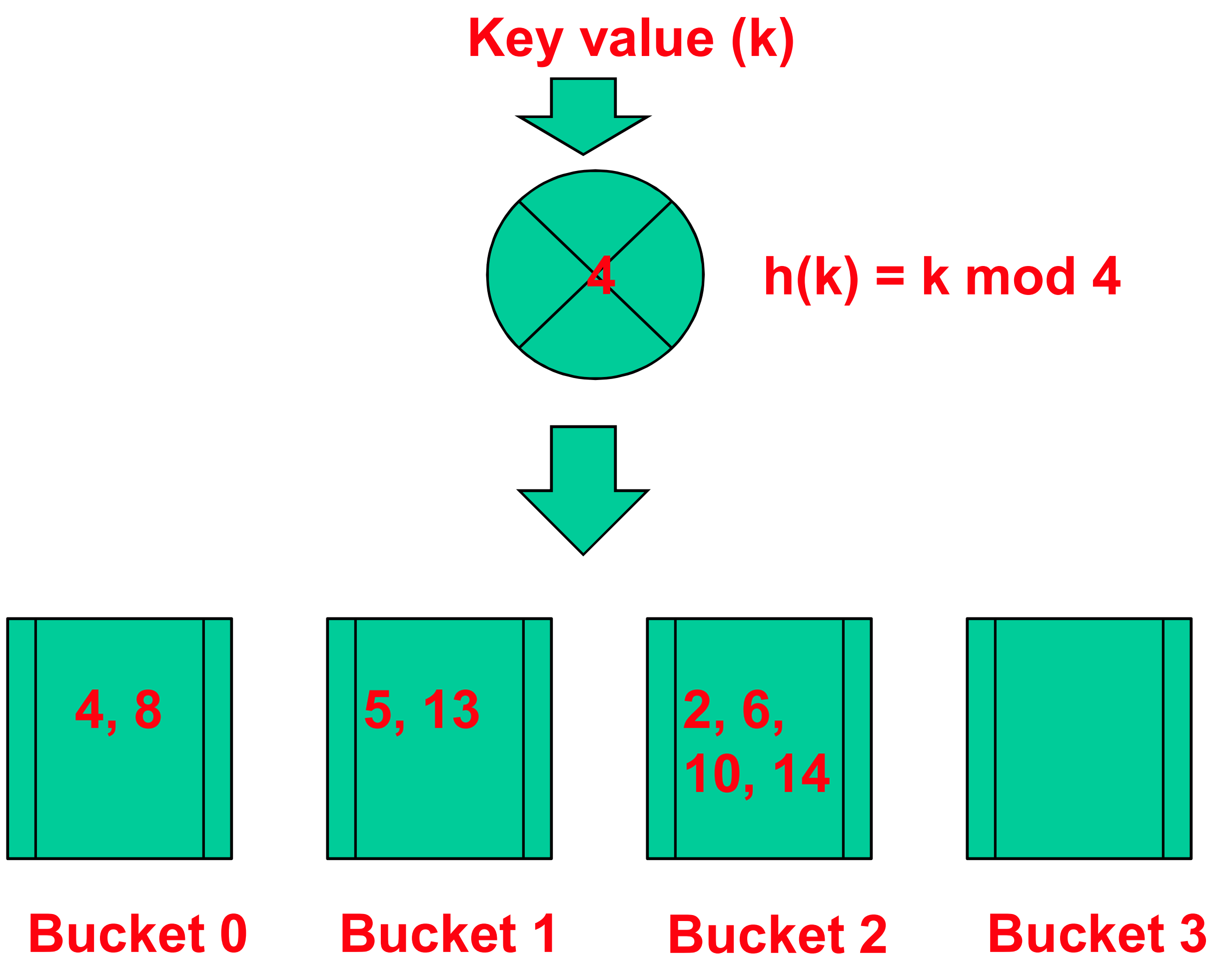
Table 17.2 Average Access Times for a File of b Blocks under Basic File Organizations

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 $bucket_0$, $bucket_1$, ..., $bucket_{M-1}$
- One of the file fields is designated to be the hash key of the file
- Suppose there is a hash function that takes a hash key as an argument to compute an integer in the range 0 to B – 1 where B is the number of buckets
- A bucket array (an array index) from 0 to B – 1 holds the headers of B lists, one for each bucket of the array
- If a record has search key K, we store the record by linking it to the bucket list for the bucket numbered h(K) where h is the hash function
  - h(K) => 0 to B – 1

Hashed Files



External Hashed Files

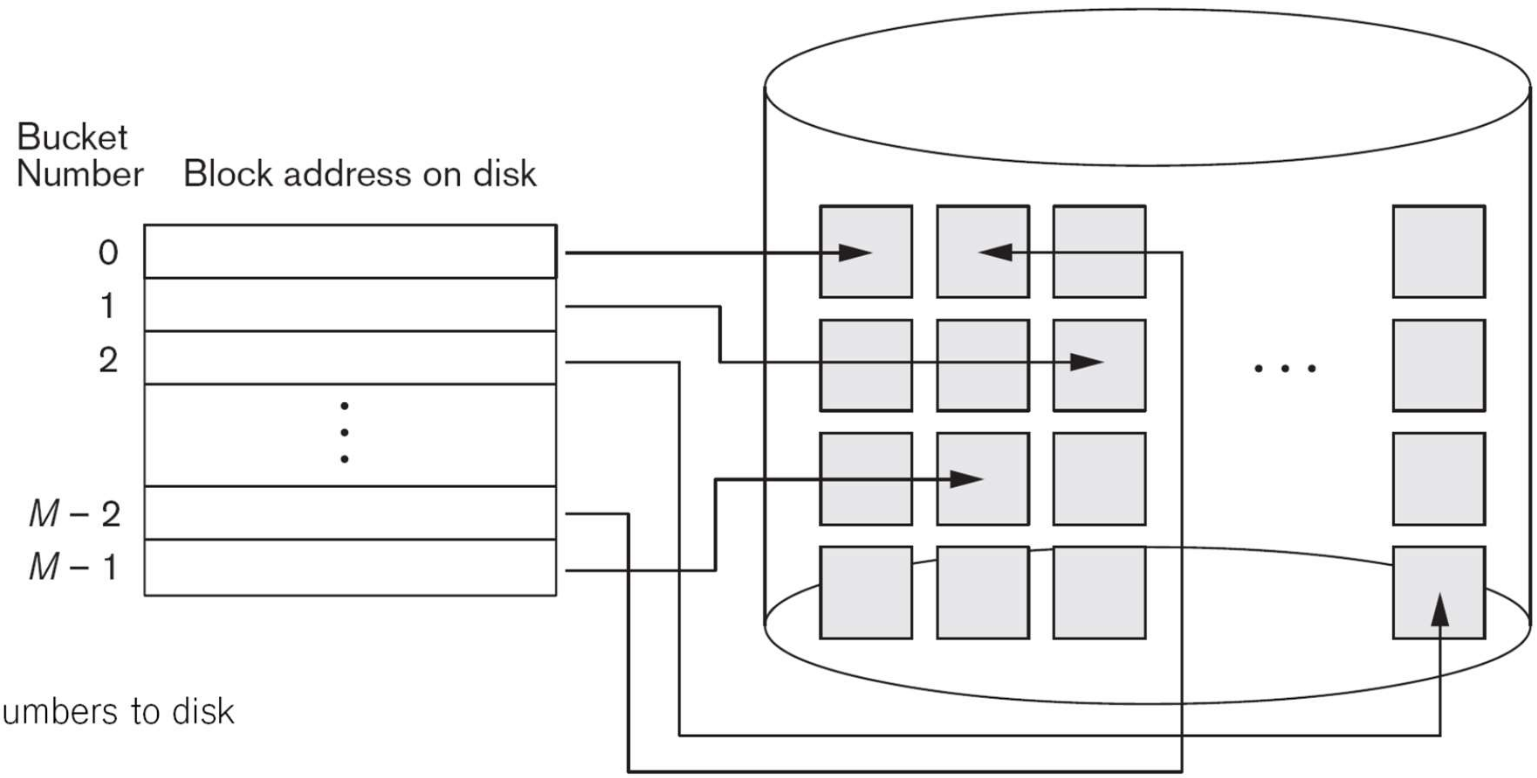
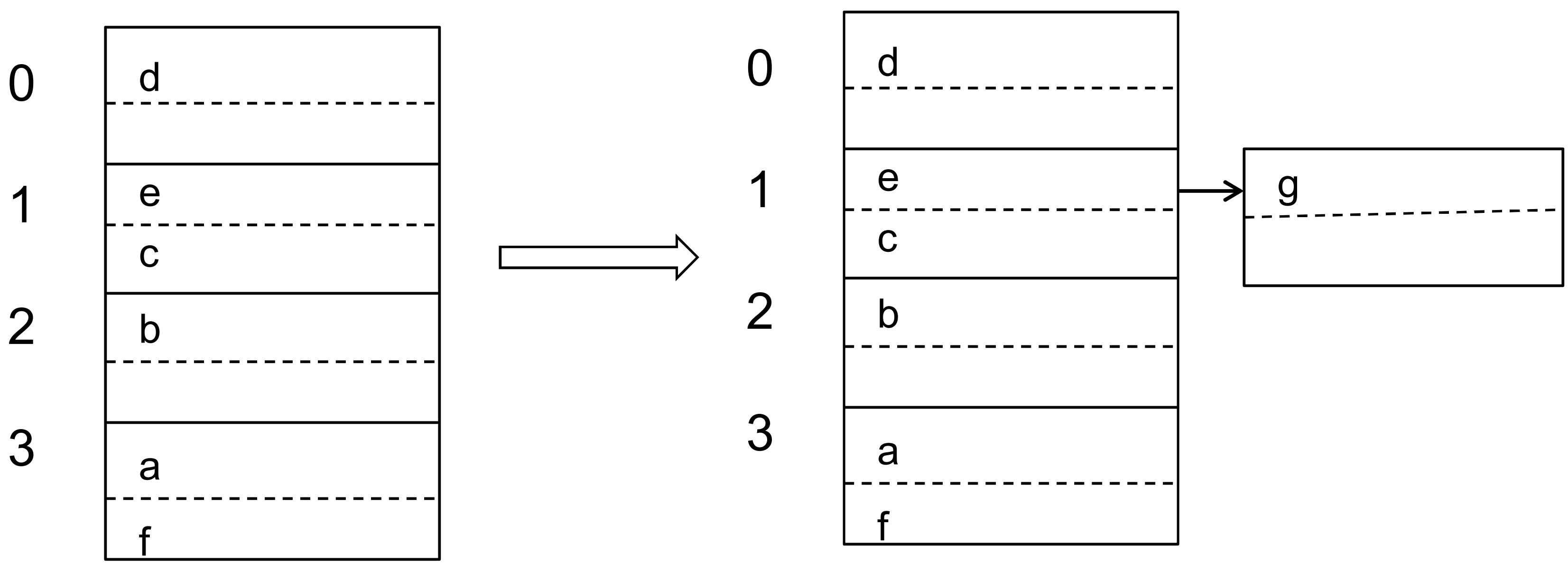


Figure 17.9
Matching bucket numbers to disk block addresses.

Example Hashed Tables

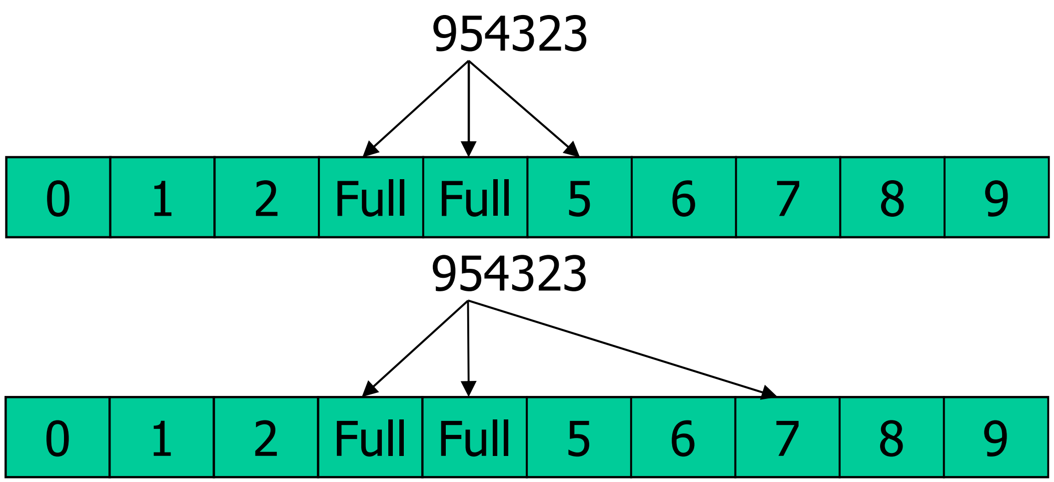
- We assume a block can hold two records and B = 4, i.e., the hash function h returns values from 0 to 3
- Suppose we add to the hash table a record with key g and h(g) = 1. Then we add the new record to the bucket numbered 1
- Collisions occur when a new record hashes to a bucket that is already full



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



Linear Probing

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

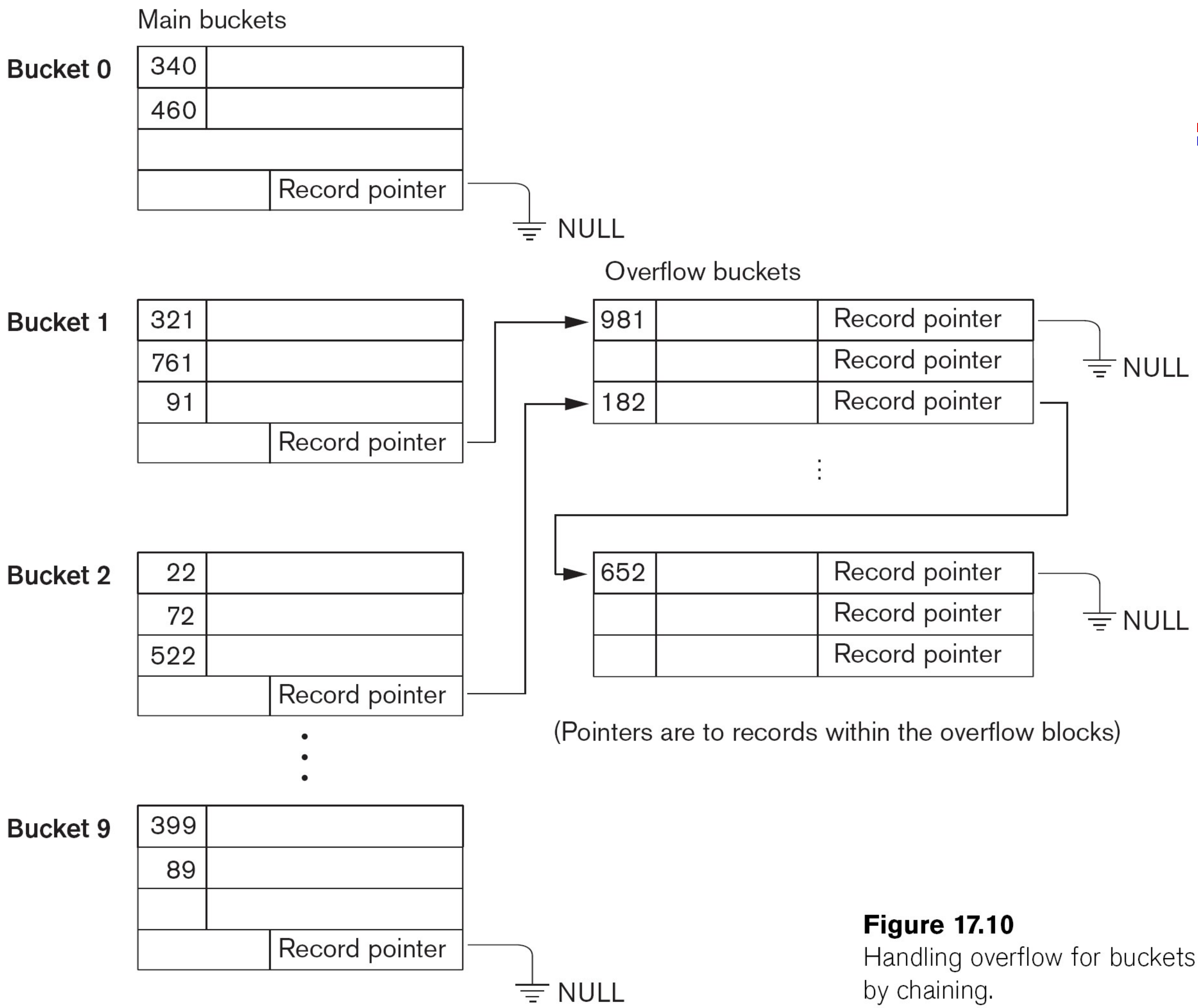


Figure 17.10
Handling overflow for buckets by chaining.

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
  - Ordered access on the hash key is quite inefficient (requires sorting the records)

Extendible and Dynamic Hashing

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

- Both dynamic and extendible hashing use the binary representation (e.g., 1100…) of the hash value h(K) in order to access a directory
  - In dynamic hashing the directory is a binary tree
  - In extendible hashing the directory is an array of size 2^d where d is called the global depth

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 disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks
  - The directory is updated appropriately

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 location d' (called, local depth 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

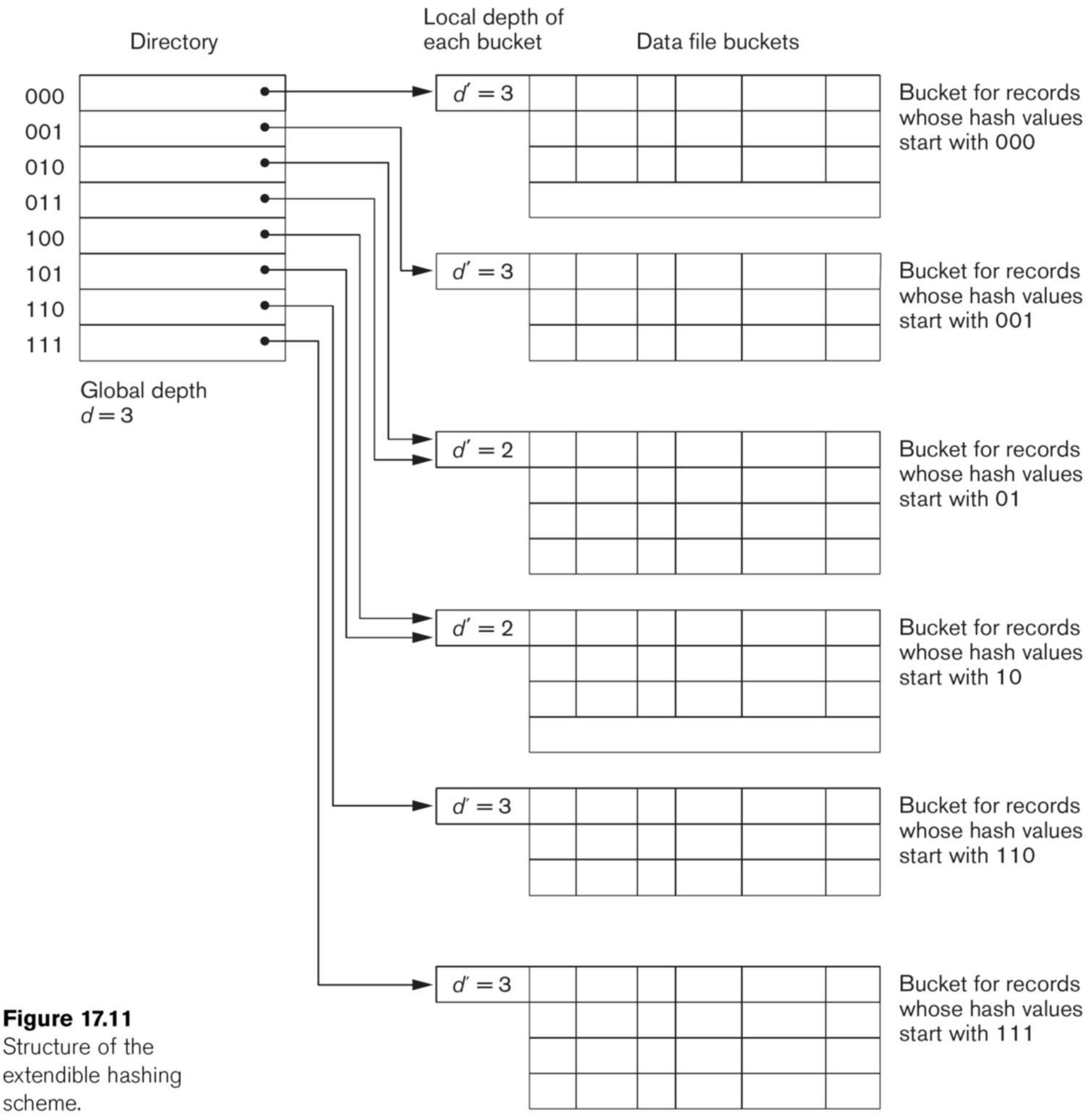


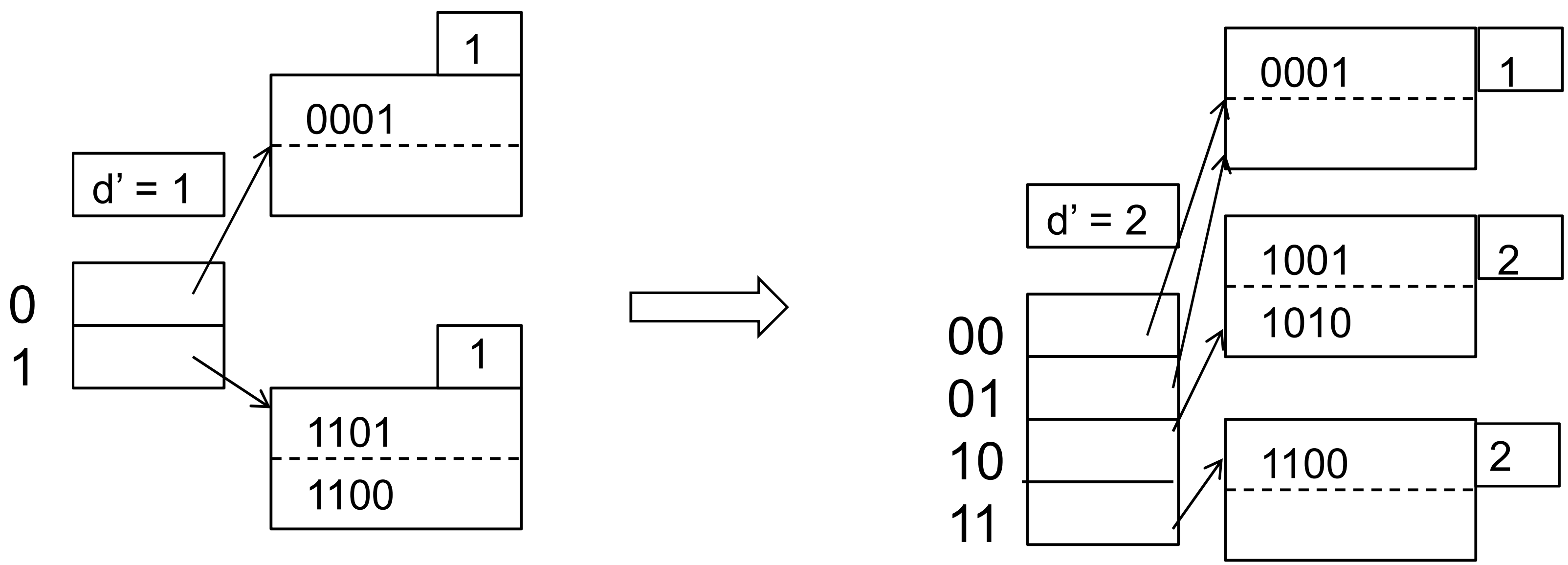
Figure 17.11
Structure of the extendible hashing scheme.

Extendible Hashing Example

- Suppose d = 4, i.e., the hash function produces a sequence of four bits
- At the moment, only one of these bits is used as illustrated by i = 1 (d' = 1) in the box above the bucket array
- The bucket array therefore has only two entries and points to two blocks
  - 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
- 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 by setting d' to 2

Extendible Hashing Example

- The two entries beginning with 0 each point to the block for records whose hashed keys begin with 0 and the block still has the integer 1 in its “nub” to indicate that only the first bit determine membership in the block
- The blocks for records beginning with 1 needs to be split into 10 and 11



Dynamic Hashing

- Dynamic and extendible hashing do not require an overflow area in general

- 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 Hashing

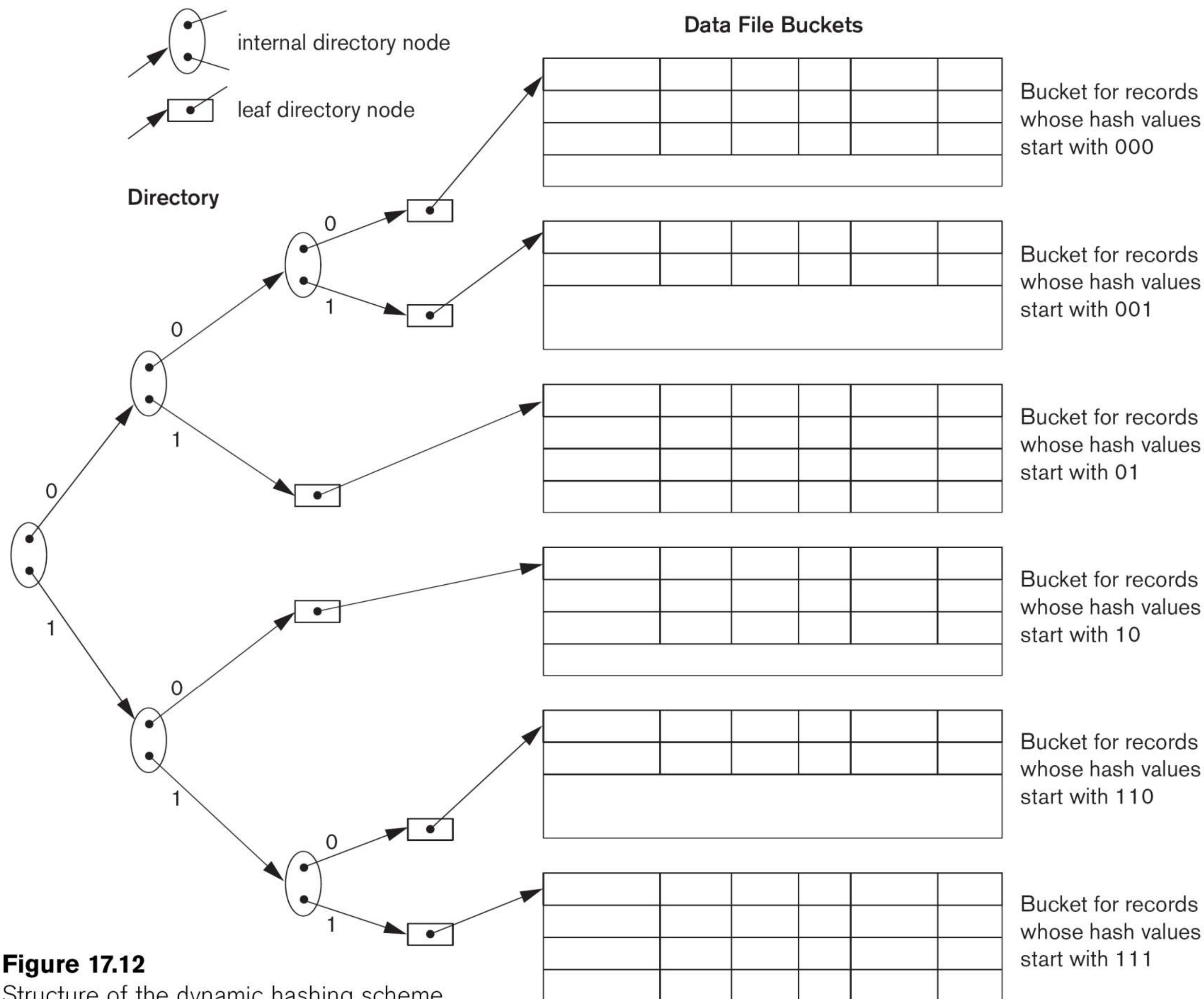


Figure 17.12
Structure of the dynamic hashing scheme.

Reference

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