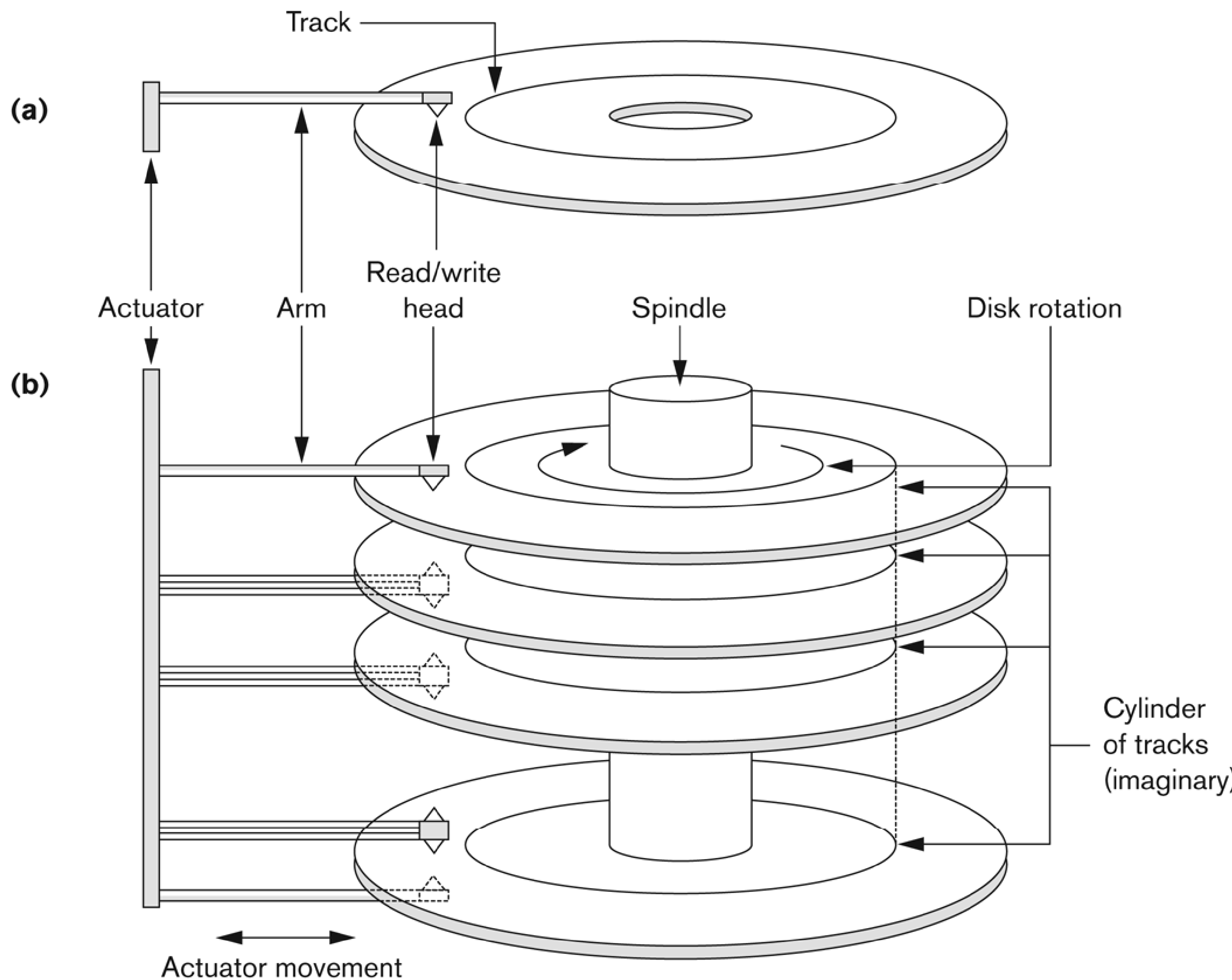
Disk Storage Devices

- 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 or more

Figure 13.1
(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.



Disk Storage Devices (contd.)

- A track is divided into smaller **blocks** or **sectors**
  - because it usually contains a large amount of information
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.
  - One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector.
- 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 (contd.)

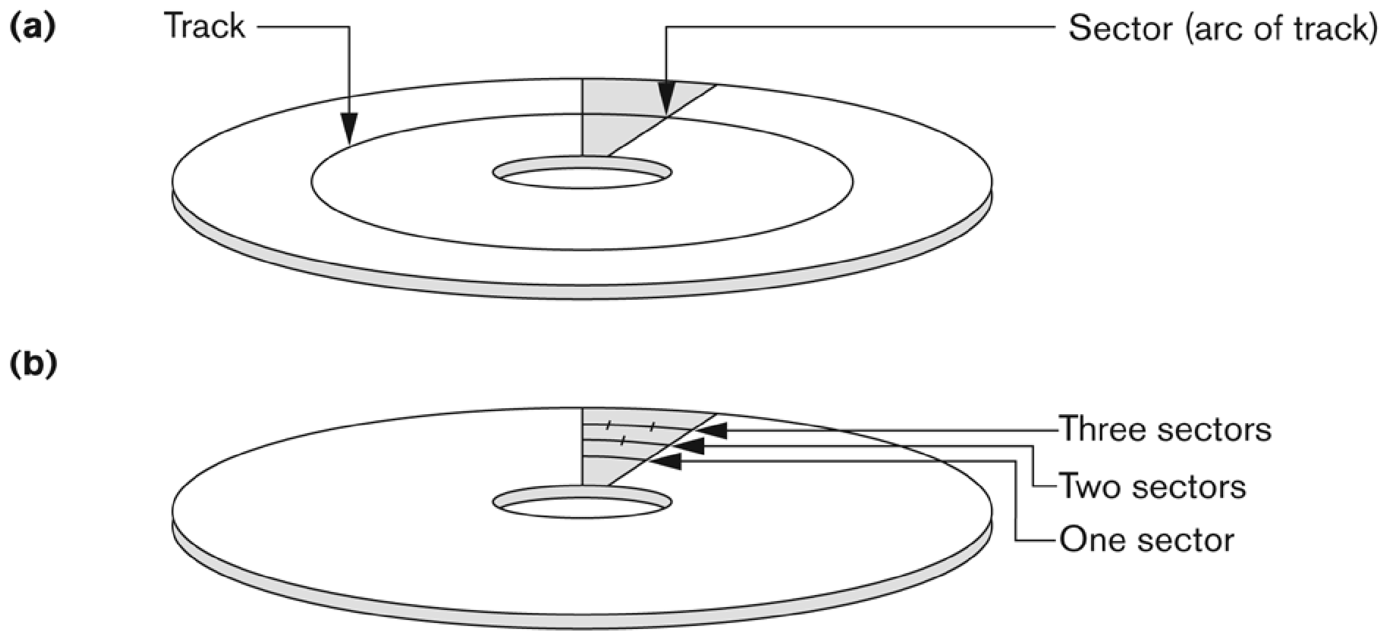


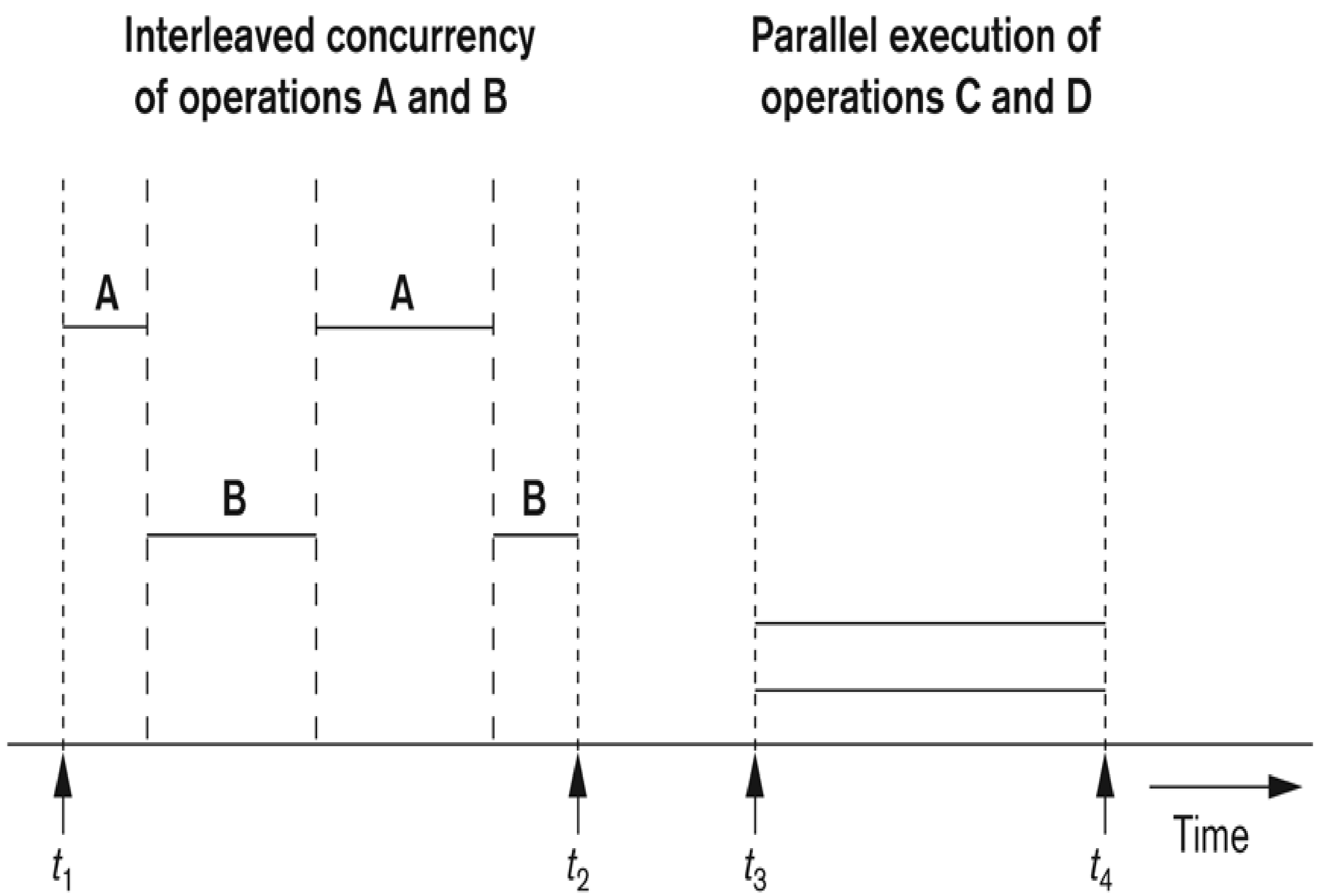
Figure 13.2
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

Disk Storage Devices (contd.)

- 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.

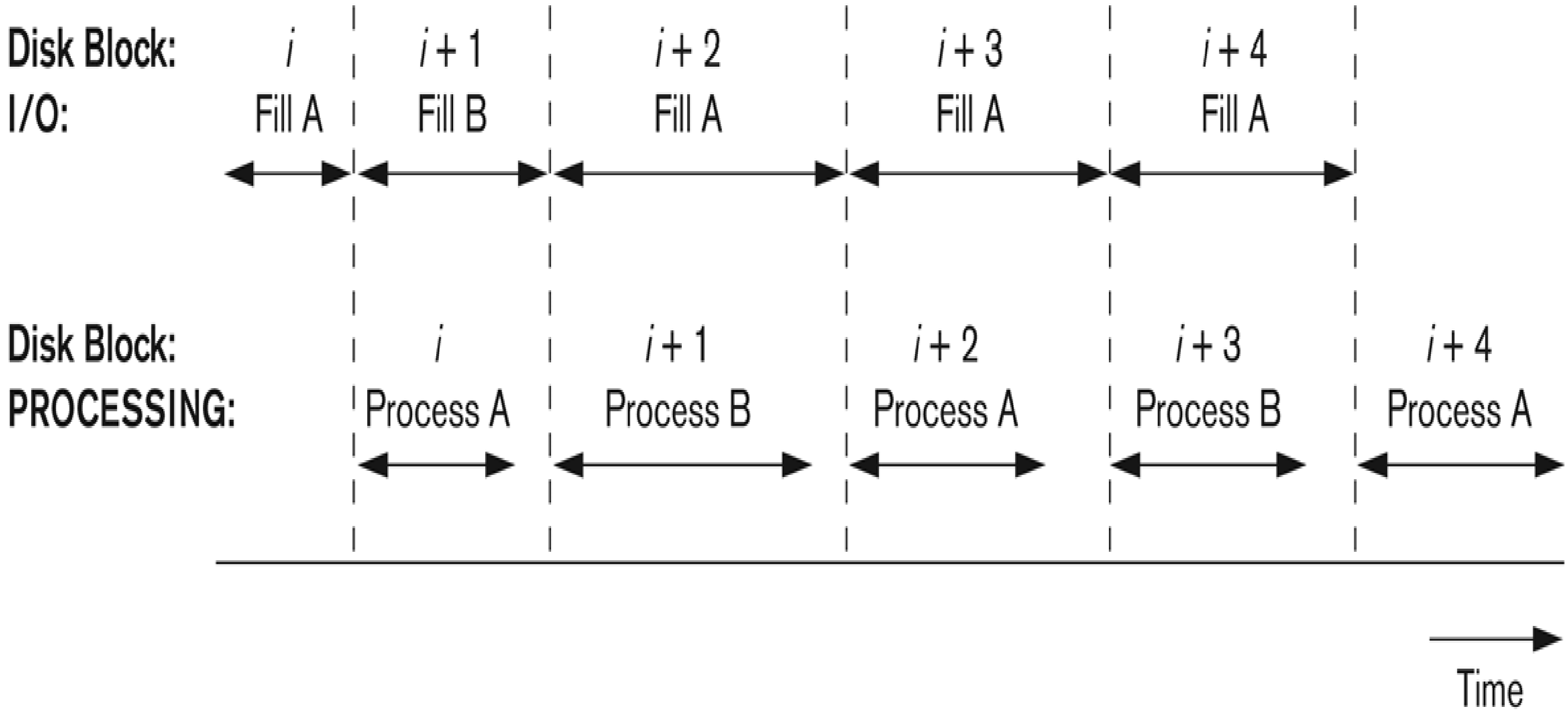
Buffering of blocks I

Figure 13.3
Interleaved concurrency versus parallel execution.



Buffering of blocks II

Figure 13.4
Use of two buffers, A and B, for reading from disk.



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."

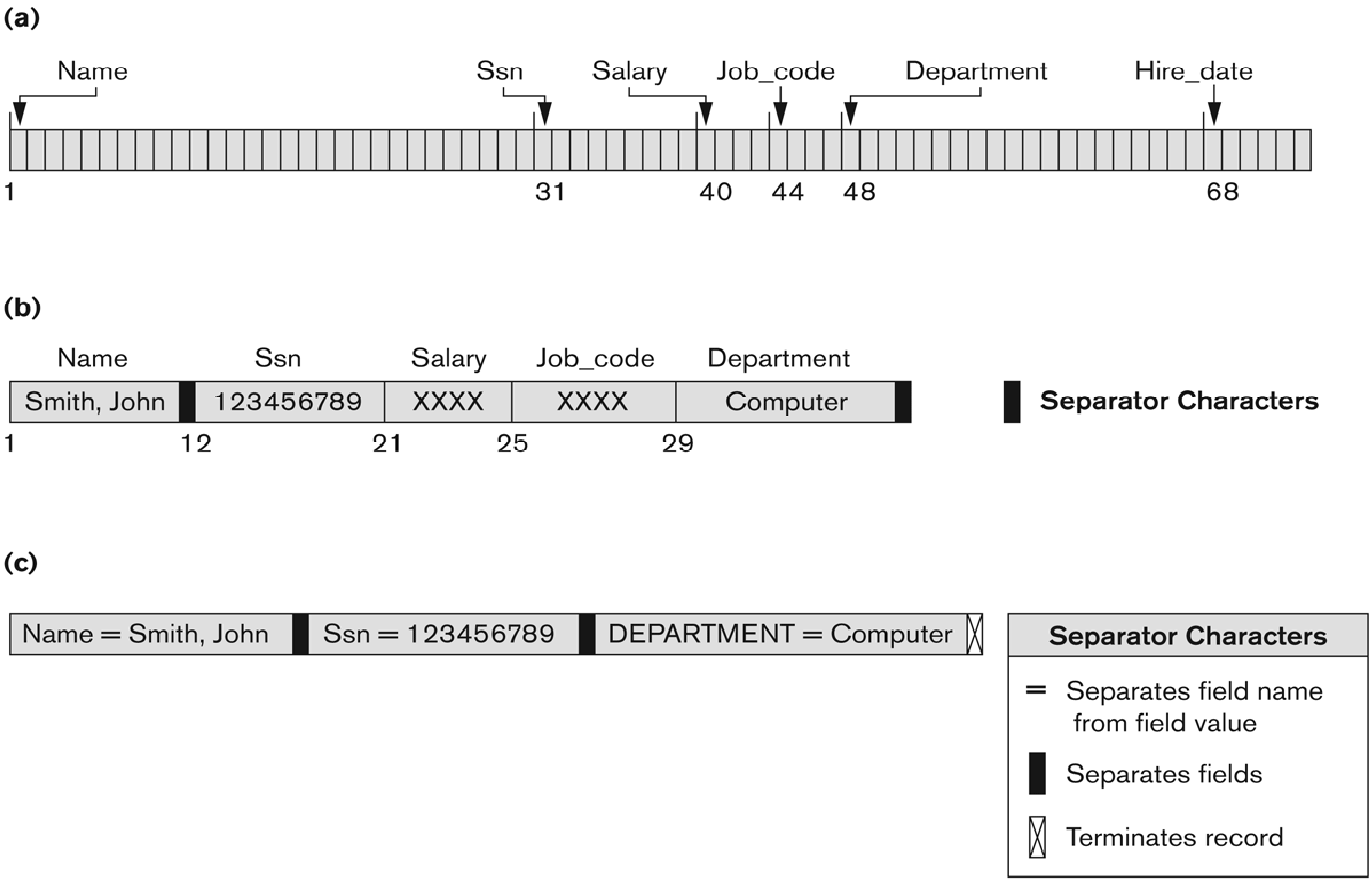


Figure 13.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

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

Files of Records

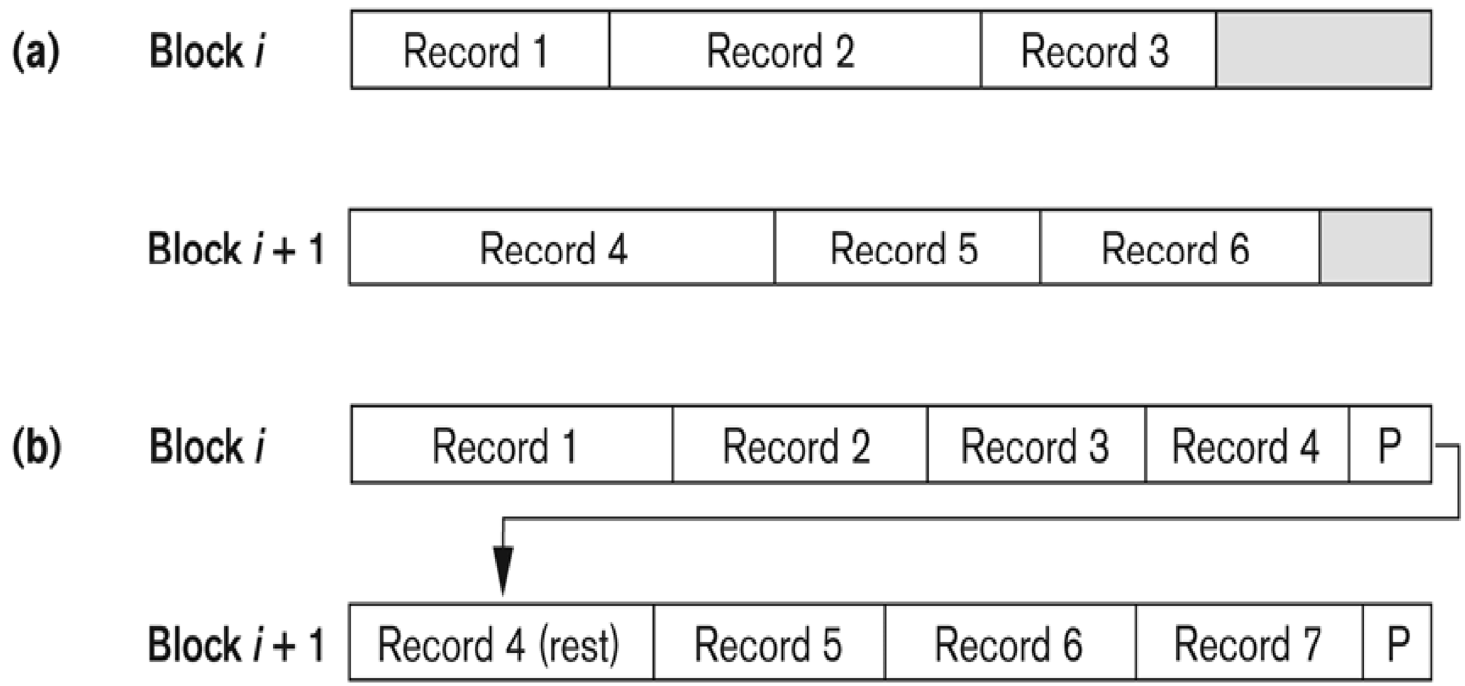
- A **file** is a *sequence* of records, where each record is a collection of data values (or data items).
- A **file descriptor** (or **file header**) includes information that describes the file, such as the *field names* and their *data types*, and the addresses of the file blocks on disk.
- Records 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 (contd.)

- 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.

Figure 13.6

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



Operation on Files

- Typical file operations include:
  - **OPEN**: Readies 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 & 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.
  - **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.

	Name	Ssn	Birth_date	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					

Figure 13.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

Algorithm 13.1. Binary Search on an Ordering Key of a Disk File

```
l ← 1; u ← b; (* b is the number of file blocks *)
while (u ≥ l ) do
  begin i ← (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 ← i – 1
  else if K > (ordering key field value of the last record in block i )
     then l ← 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;
```

Average Access Times

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

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

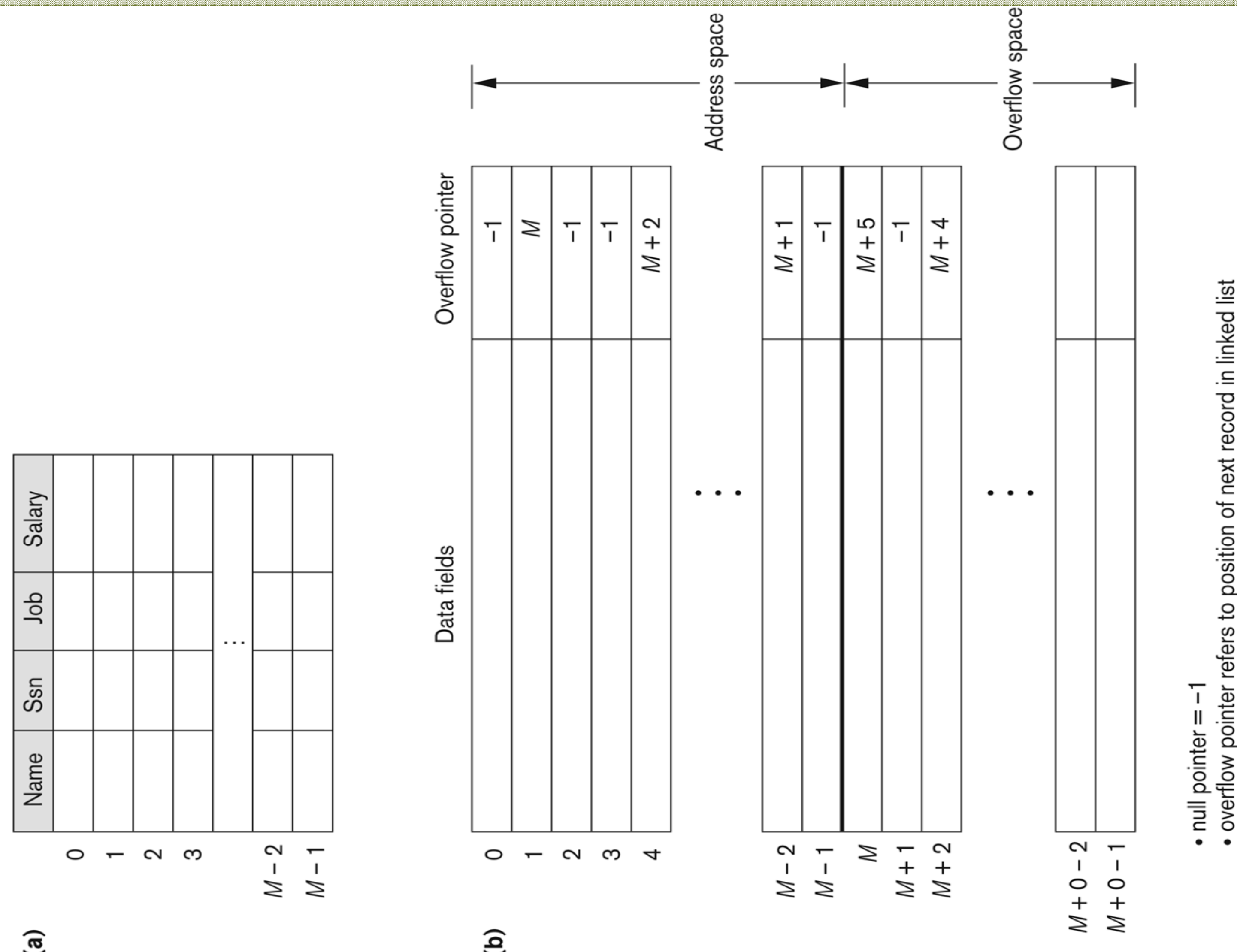
TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

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.

INTERNAL HASHING

Figure 13.8
Internal hashing data structures. (a) Array of *M* positions for use in internal hashing.
(b) Collision resolution by chaining records.



Hashed Files (contd.)

- 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.
  - **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 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.

Algorithm 13.2. Two simple hashing algorithms. (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

(a)

```
temp ← 1;
for i ← 1 to 20 do temp ← temp * code(K[i ] ) mod M ;
hash_address ← temp mod M;
```

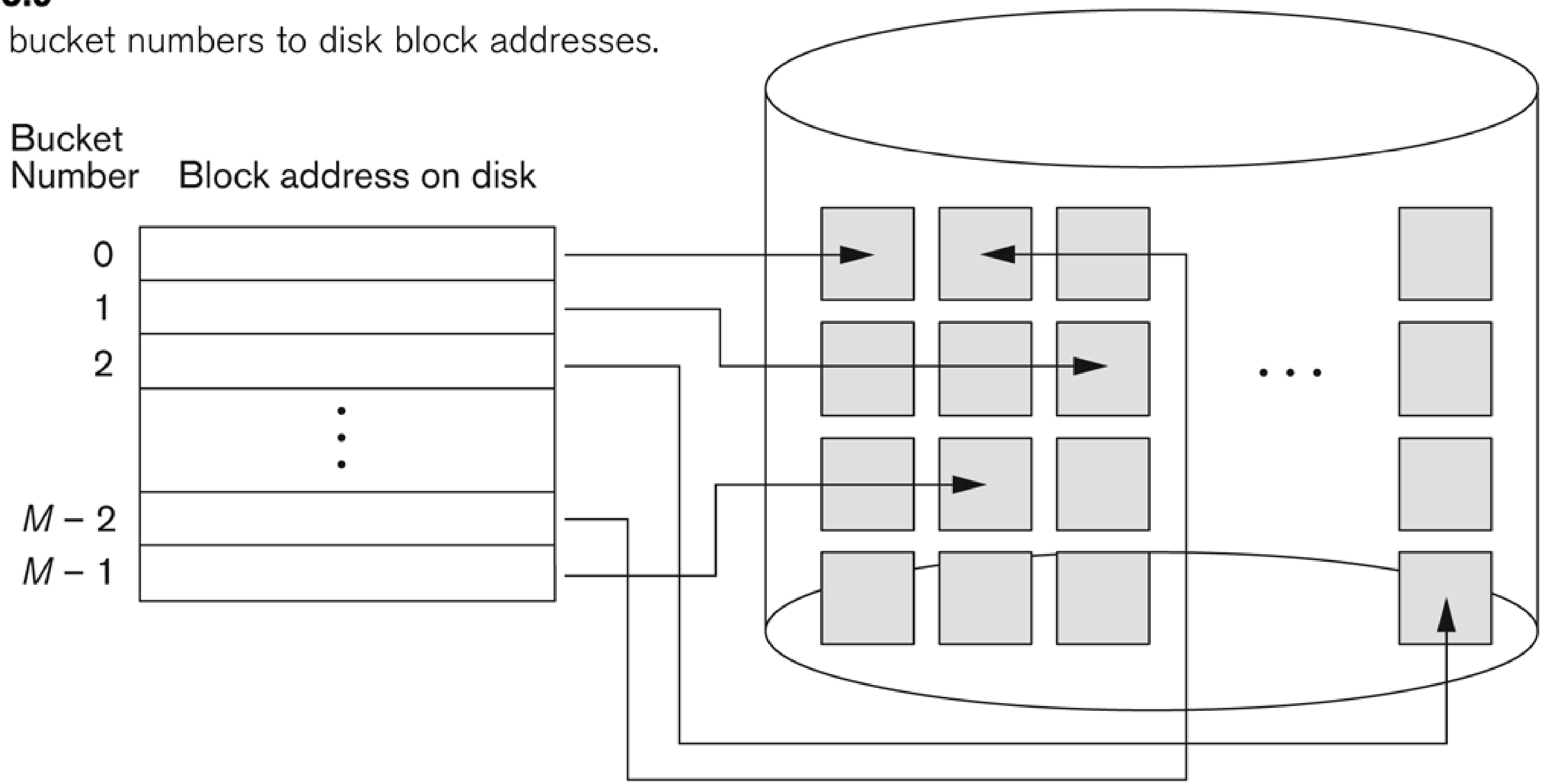
(b)

```
i ← hash_address(K); a ← i;
if location i is occupied
  then begin i ← (i + 1) mod M;
        while (i ≠ a) and location i is occupied
          do i ← (i + 1) mod M;

        if (i = a) then all positions are full
        else new_hash_address ← i;
        end;
```

External Hashing for Disk Files

Figure 13.9
Matching bucket numbers to disk block addresses.



External Hashing for Disk 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).

Hashed Files - Overflow handling

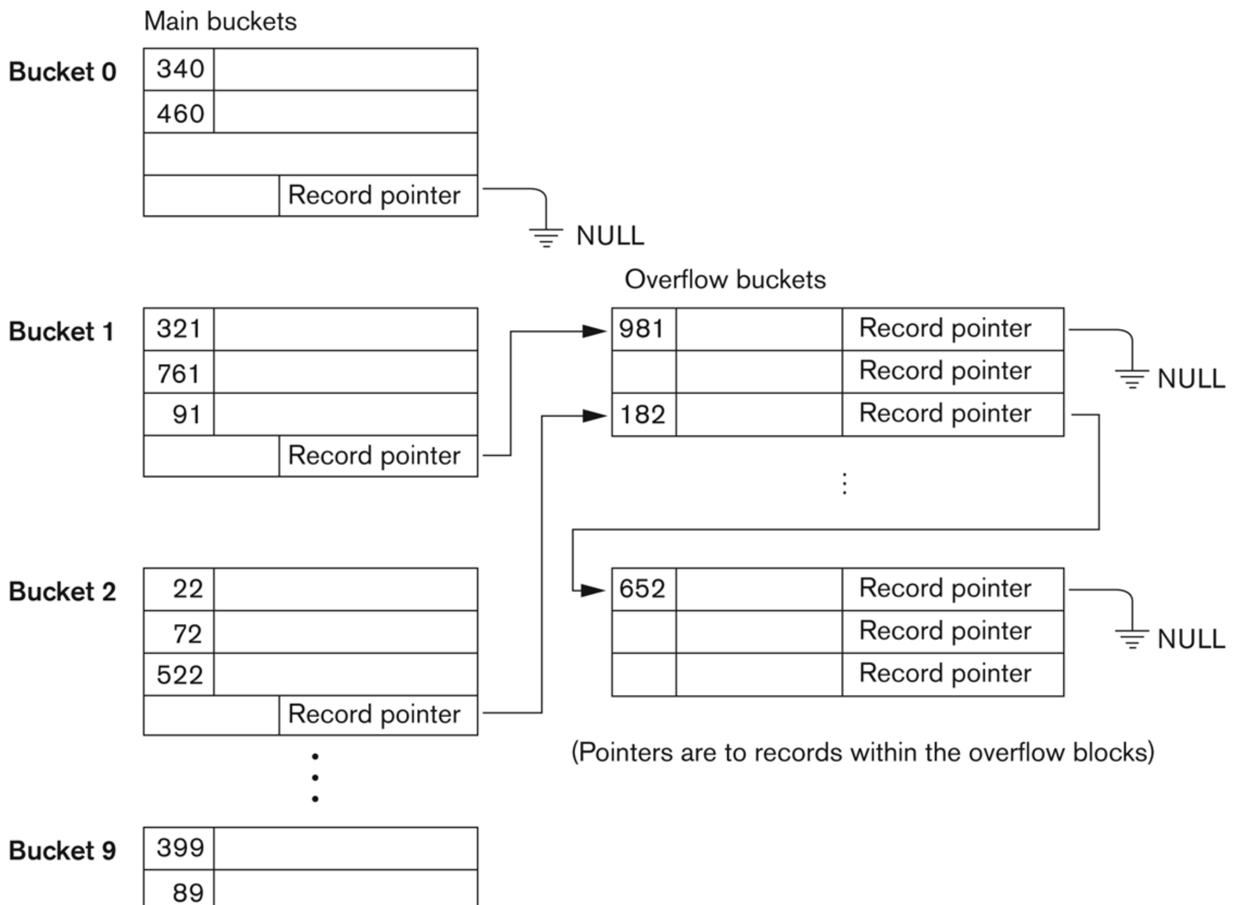


Figure 13.10
Handling overflow for buckets by chaining.

Dynamic And Extendible Hashed Files

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
  - These techniques include the following: **dynamic hashing, extendible hashing**, and **linear hashing.**
- Both dynamic and extendible hashing use the **binary representation** 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**.

Dynamic And Extendible Hashing (contd.)

- 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.
- Dynamic and extendible hashing do not require an overflow area.
- Linear hashing does require an overflow area but does not use a directory.
  - Blocks are split in *linear order* as the file expands.

Extendible Hashing

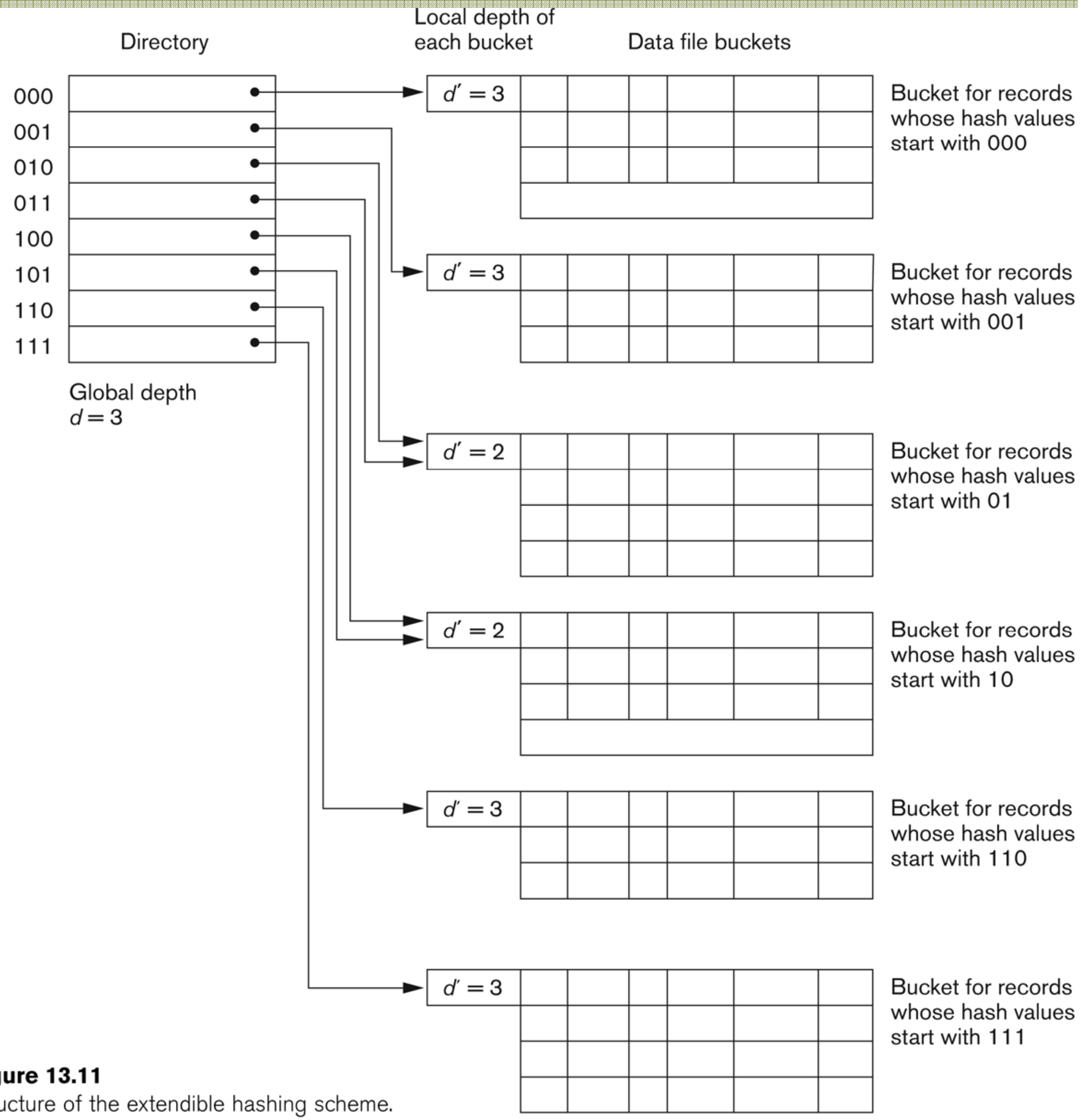


Figure 13.11
Structure of the extendible hashing scheme.

Linear Hashing

Algorithm 13.3. The Search Procedure for Linear Hashing

if $n = 0$

then $m \leftarrow h_j(K)$ (* m is the hash value of record with hash key K *)

else **begin**

$m \leftarrow h_j(K)$;

if $m < n$ then $m \leftarrow h_{j+1}(K)$

end;

search the bucket whose hash value is m (and its overflow, if any);