Lecture 7: Disk Storage, Basic File Structures, and Hashing

CS3402 Database Systems

Storage Hierarchy (1/3)

- Primary storage
  - This category includes storage media that can be operated on directly by the computer's central processing unit (CPU), such as the computer's main memory and smaller but faster cache memories.
  - Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than demanded by typical enterprise-level databases. The contents of main memory are lost in case of power failure or a system crash.
- Secondary storage
  - The primary choice of storage medium for online storage of enterprise databases has been magnetic disks. However, flash memories are becoming a common medium of choice for storing moderate amounts of permanent data. When used as a substitute for a disk drive, such memory is called a solid-state drive (SSD).

Storage Hierarchy (2/3)

- Tertiary storage
  - Optical disks (CD-ROMs, DVDs, and other similar storage media) and tapes are removable media used in today's systems as offline storage for archiving databases and hence come under the category called tertiary storage.
  - These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices.
- Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

Storage Hierarchy (3/3)

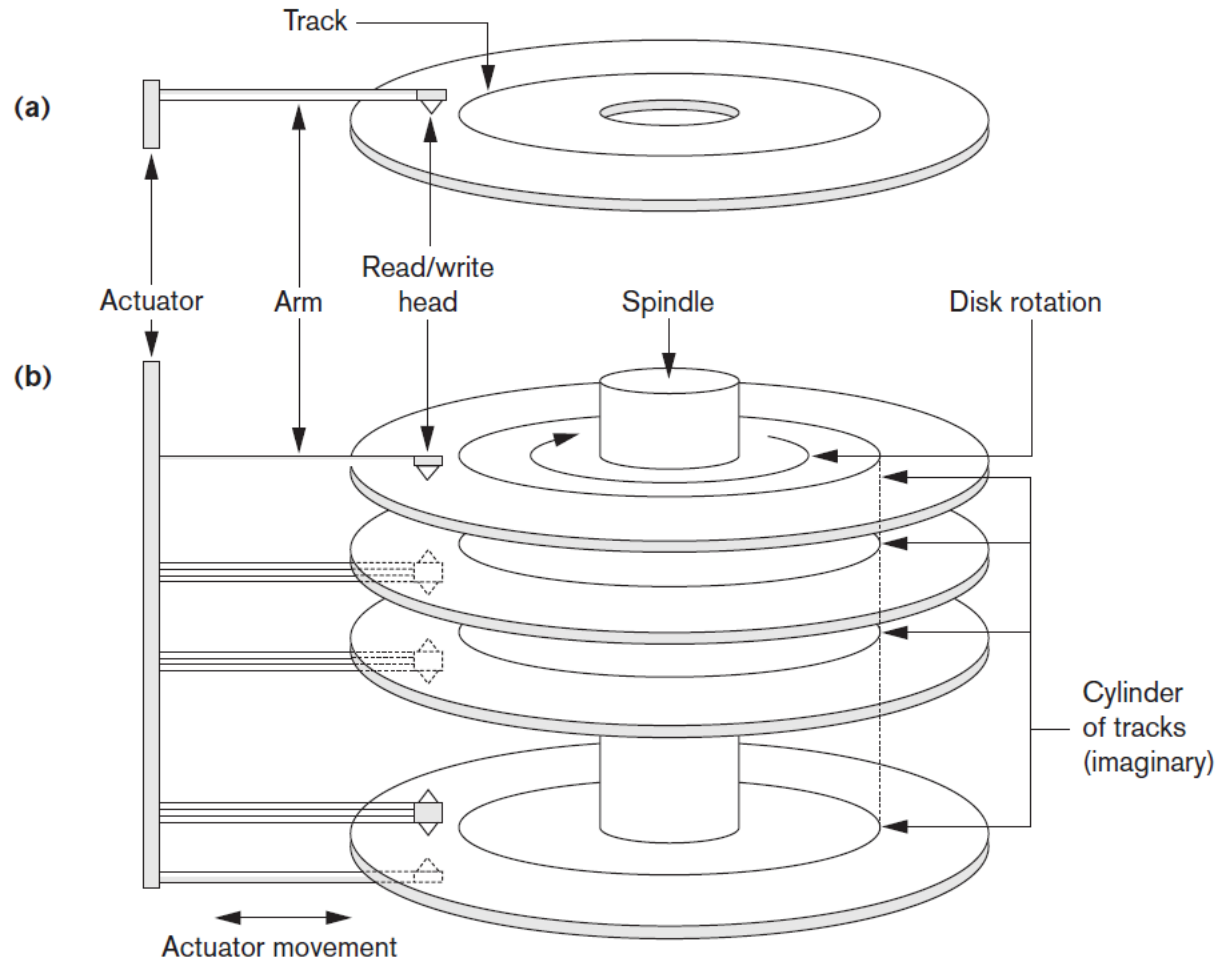
- At the primary storage level, the memory hierarchy includes, at the most expensive end, cache memory, which is a static RAM (random access memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining.
- The next level of primary storage is DRAM (dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called main memory. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility and lower speed compared with static RAM.
- At the secondary and tertiary storage level, the hierarchy includes magnetic disks; mass storage in the form of CD-ROM (compact disk–read-only memory) and DVD (digital video disk or digital versatile disk) devices; and finally tapes at the least expensive end of the hierarchy.

Hardware Description of Disk Devices (1/3)

- Magnetic disks are used for storing large amounts of data. The device that holds the disks is referred to as a **hard disk drive**, or **HDD**. The most basic unit of data on the disk is a single **bit** of information. By magnetizing an area on a disk in certain ways, one can make that area represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into **bytes** (or **characters**).
- Byte sizes are 8 bits is the most common. We assume that one character is stored in a single byte, and we use the terms byte and character interchangeably.
- The **capacity** of a disk is the number of bytes it can store, which is usually very large.

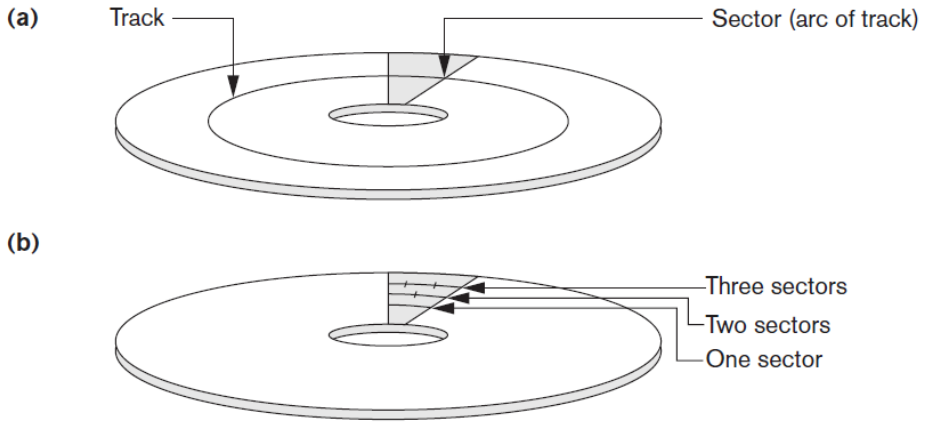
Hardware Description of Disk Devices (2/3)

- All disks are made of magnetic material shaped as a thin circular disk, as shown in (a).
- To increase storage capacity, disks are assembled into a **disk pack**, as shown in (b), which may include many disks and therefore many surfaces.
- Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are called a **cylinder** because of the shape they would form if connected in space.



Hardware Description of Disk Devices (3/3)

- A track usually contains a large amount of information, it is divided into smaller blocks or sectors.
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed. One type of sector organization, as shown in (a), calls a portion of a track that subtends a fixed angle at the center a sector.
- The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 8192 bytes.
- **Disk → Track → Sectors → Blocks**



Disk Access Delay (1/2)

- To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the **seek time**. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msec on servers.
- Following that, there is another delay—called the **rotational delay** or **latency**—while the beginning of the desired block rotates into position under the read/write head. It depends on the rpm of the disk. For example, at 15,000 rpm, the time per rotation is 4 msec and the average rotational delay is the time per half revolution, or 2 msec. At 10,000 rpm the average rotational delay increases to 3 msec.
- Finally, some additional time is needed to transfer the data; this is called the **block transfer time**.
- **Disk access delay = seek time + rotational delay + block transfer time**

Disk Access Delay (2/2)

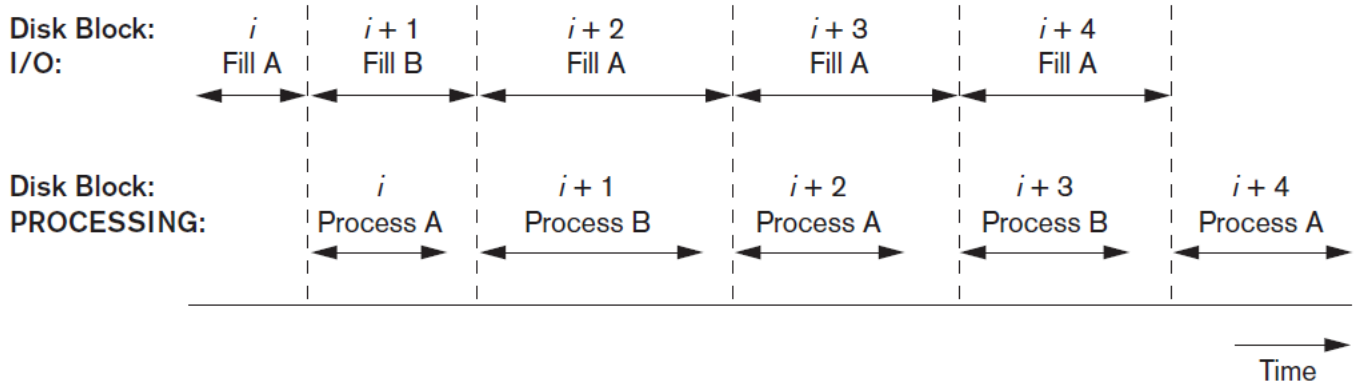
- The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred.
- The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on a disk.

Buffering of Blocks (1/2)

- When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer.
- While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Buffering of Blocks (2/2)

- Reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer.
- The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering**.



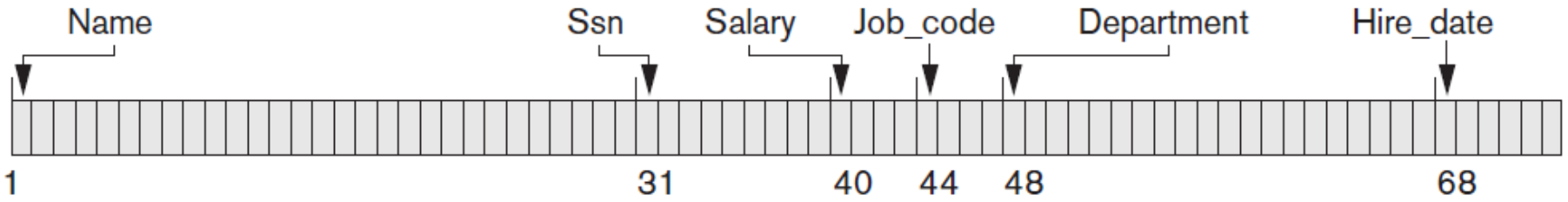
Use of two buffers, A and B, for reading from disk.

Files, Fixed-Length Records, and Variable-Length Records (1/4)

- A **file** is a sequence of records. In many cases, all records in a file are of the same record type.
- If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records.**
- If different records in the file have different sizes, the file is said to be made up of **variable-length records**.
  - The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the Name field of EMPLOYEE can be a variable-length field.
  - The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields**).

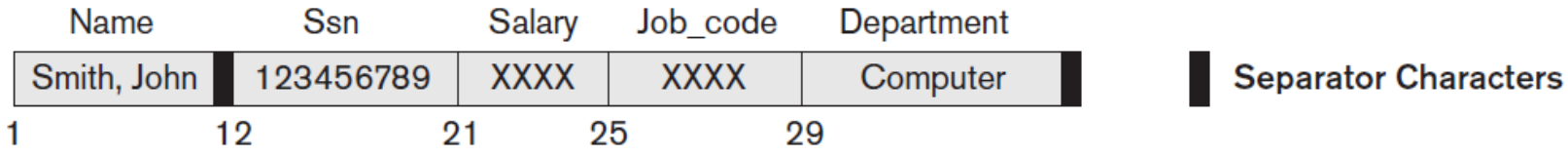
Files, Fixed-Length Records, and Variable-Length Records (2/4)

- The fixed-length EMPLOYEE records have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files.
- Space is wasted when certain records do not have values for all the physical spaces provided in each record.



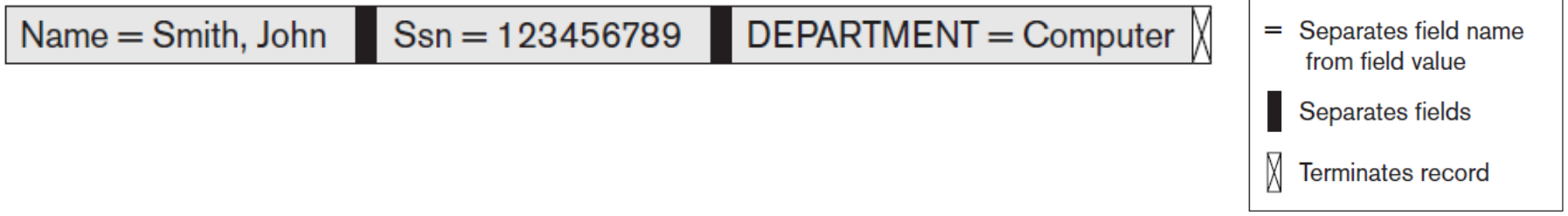
Files, Fixed-Length Records, and Variable-Length Records (3/4)

- For variable-length fields, each record has a value for each field, but we do not know the exact length of some field values.
- To determine the bytes within a particular record that represent each field, we can use special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields, or we can store the length in bytes of the field in the record, preceding the field value.



Files, Fixed-Length Records, and Variable-Length Records (4/4)

- A file of records with optional fields can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values
- Multiple types of separator characters could be used, although we could use the same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field.
- A more practical option is to assign a short **field type** code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs.

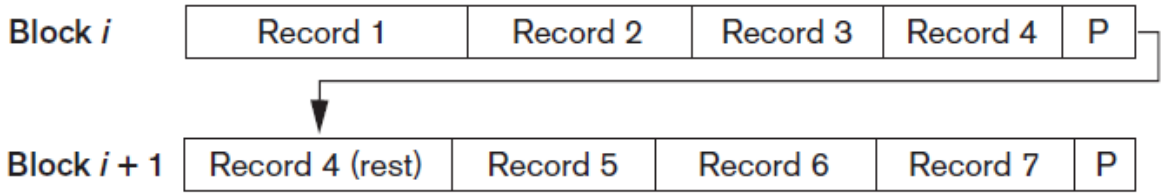


Record Blocking and Spanned vs. Unspanned Records (1/3)

- The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.
- When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one 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. The value bfr is called the **blocking factor** for the file.
- In general, R may not divide B exactly, so we have some unused space in each block equal to: $B - (bfr * R)$ bytes

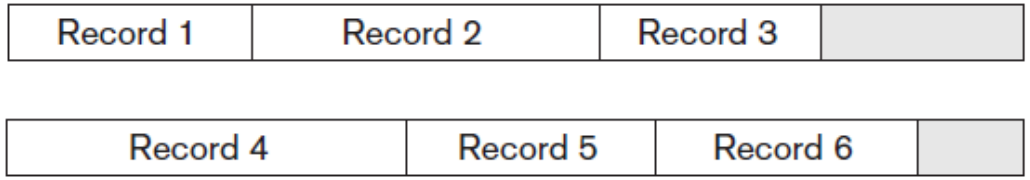
Record Blocking and Spanned vs. Unspanned Records (2/3)

- To utilize this unused space, we can store part of a record on one block and the rest on another. A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk.
- This organization is called **spanned** because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization.



Record Blocking and Spanned vs. Unspanned Records (3/3)

- If records are not allowed to cross block boundaries, the organization is called **unspanned**. This is used with fixed-length records having B > R because it makes each record start at a known location in the block, simplifying record processing.



- For variable-length records, either a spanned or an unspanned organization can be used. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block.

Files of Unordered Records (Heap Files) (1/2)

- In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap file.**
- Inserting a new record is very efficient. The last disk block of the file is copied into a buffer, the new record is added, and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header.

Files of Unordered Records (Heap Files) (2/2)

- However, searching for a record using any search condition involves a **linear search** through the file block by block—an expensive procedure.
  - If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record.
  - For a file of b blocks, this requires searching (b/2) blocks, on average.
  - If no records or several records satisfy the search condition, the program must read and search all b blocks in the file, i.e., worst-case.
- To delete a record, a program must first find its block.

Files of Ordered Records (Sorted Files) (1/3)

- We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field**. This leads to an **ordered** or **sequential** file.
- If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is called the **ordering key** for the file.

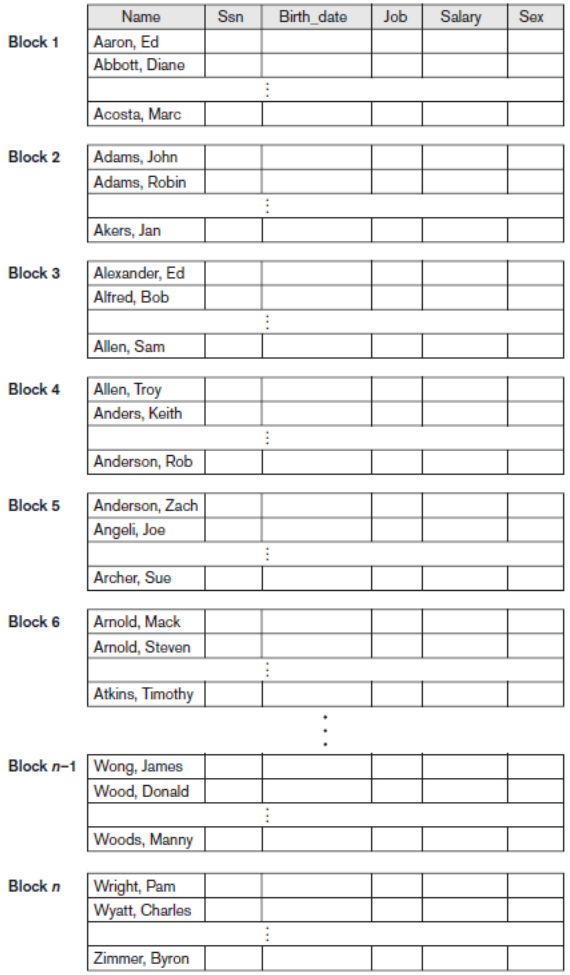


Figure 16.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

Files of Ordered Records (Sorted Files) (2/3)

- Ordered records have some advantages over unordered files.
  - Reading the records in order of the ordering key values becomes extremely efficient because no sorting is required. The search condition may be of the type < key = value>, or a range condition such as < value1 < key < value2>.
  - Finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one (unless the current record is the last one in the block).
  - Using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time.

Files of Ordered Records (Sorted Files) (3/3)

- A **binary search** for disk files can be done on the blocks rather than on the records.
- Suppose that the file has b blocks numbered 1, 2, …, b; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K.
- Assuming that disk addresses of the file blocks are available in the file header, a binary search usually accesses log2(b) blocks, whether the record is found or not—an improvement over linear searches, where, on the average, (b/2) blocks are accessed when the record is found and b blocks are accessed when the record is not found.

Average Access Time

- The average access time in block accesses to find a specific record in a file with b blocks.

Table 16.3 Average Access Times for a File of *b* Blocks under Basic File Organizations

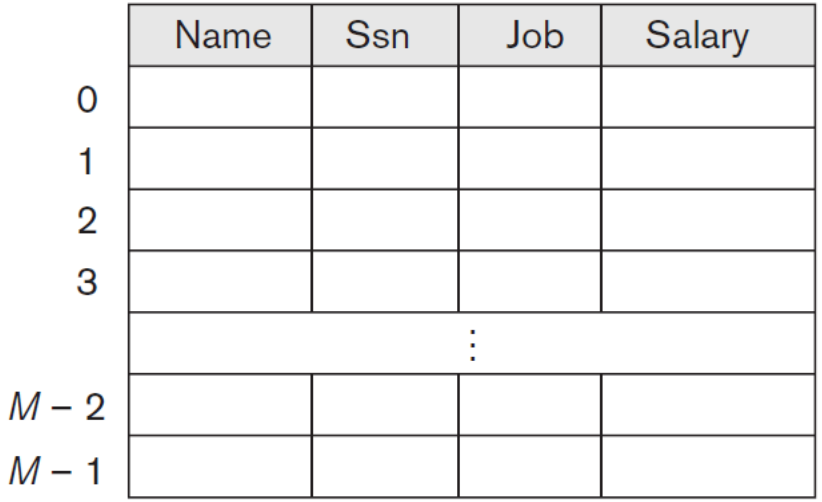
Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

Hash Files

- Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions. This organization is usually called a **hash file**.
- The search condition must be an equality condition on a single field, called the **hash field**. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**.
- The idea behind hashing is to provide a function h, called a **hash function**, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

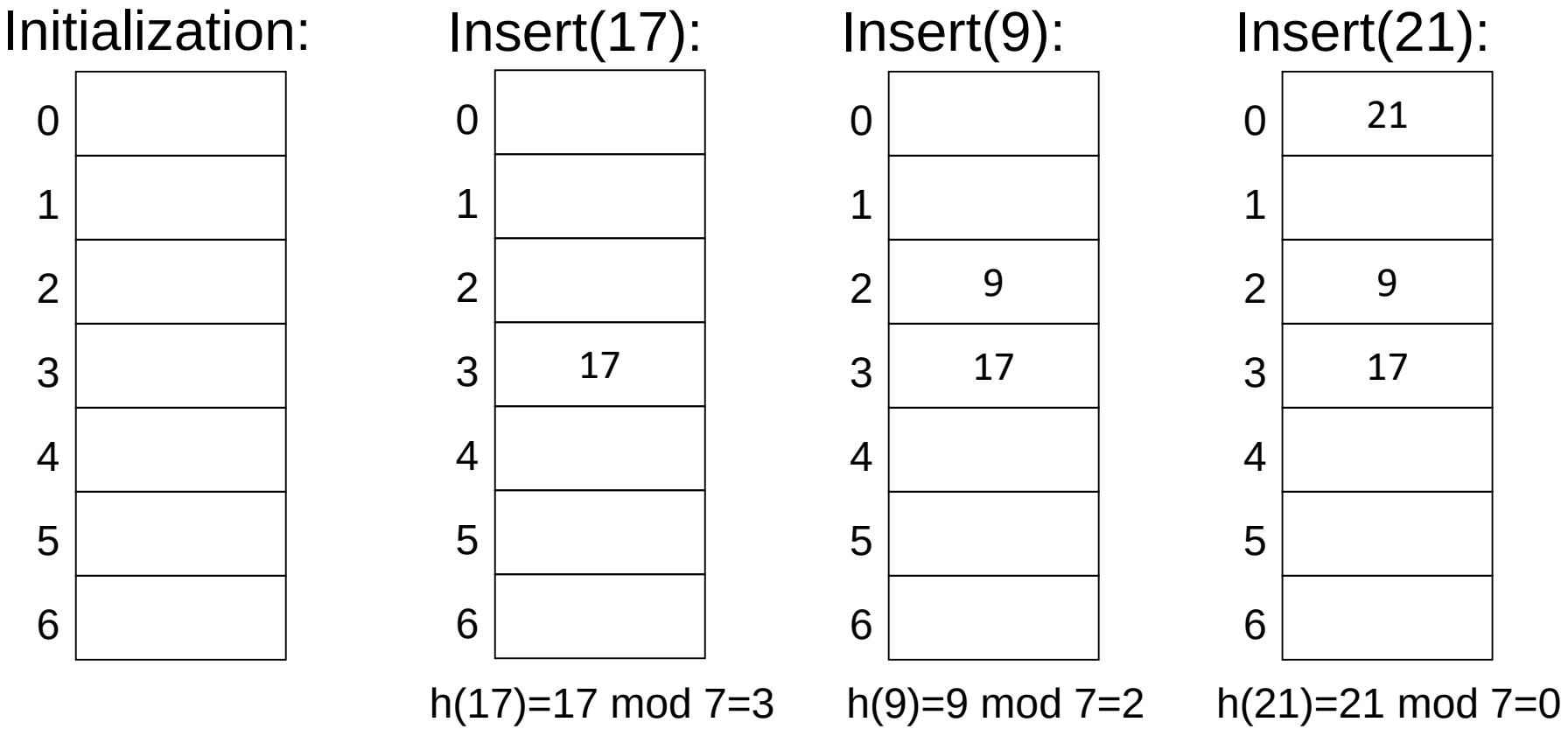
Internal Hashing (1/3)

- Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field.
- For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to M – 1; then we have M **slots** whose addresses correspond to the array indexes.
- We choose a hash function that transforms the hash field value into an integer between 0 and M − 1. One common hash function is the h(K) = K **mod** M function, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.



Internal Hashing (2/3)

- Example: assume keys are non-negative integers
- Choose M=7, h(x) = x mod 7



Internal Hashing (3/3)

- Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values.
- A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called **collision resolution**.

Three Methods for Collision Resolution (1/2)

- **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. Additionally, 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. A linked list of overflow records for each hash address

Three Methods for Collision Resolution (2/2)

- **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. Note that the series of hash functions are used in the same order for retrieval.
- Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest. Deletion algorithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail.

External Hashing (1/3)

- Hashing for disk files is called **external hashing**. To suit the characteristics of disk storage, the target address space is made of **buckets**, each of which holds multiple records.
- A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number rather than assigning an absolute block address to the bucket.
- A table maintained in the file header converts the bucket number into the corresponding disk block address

External Hashing (2/3)

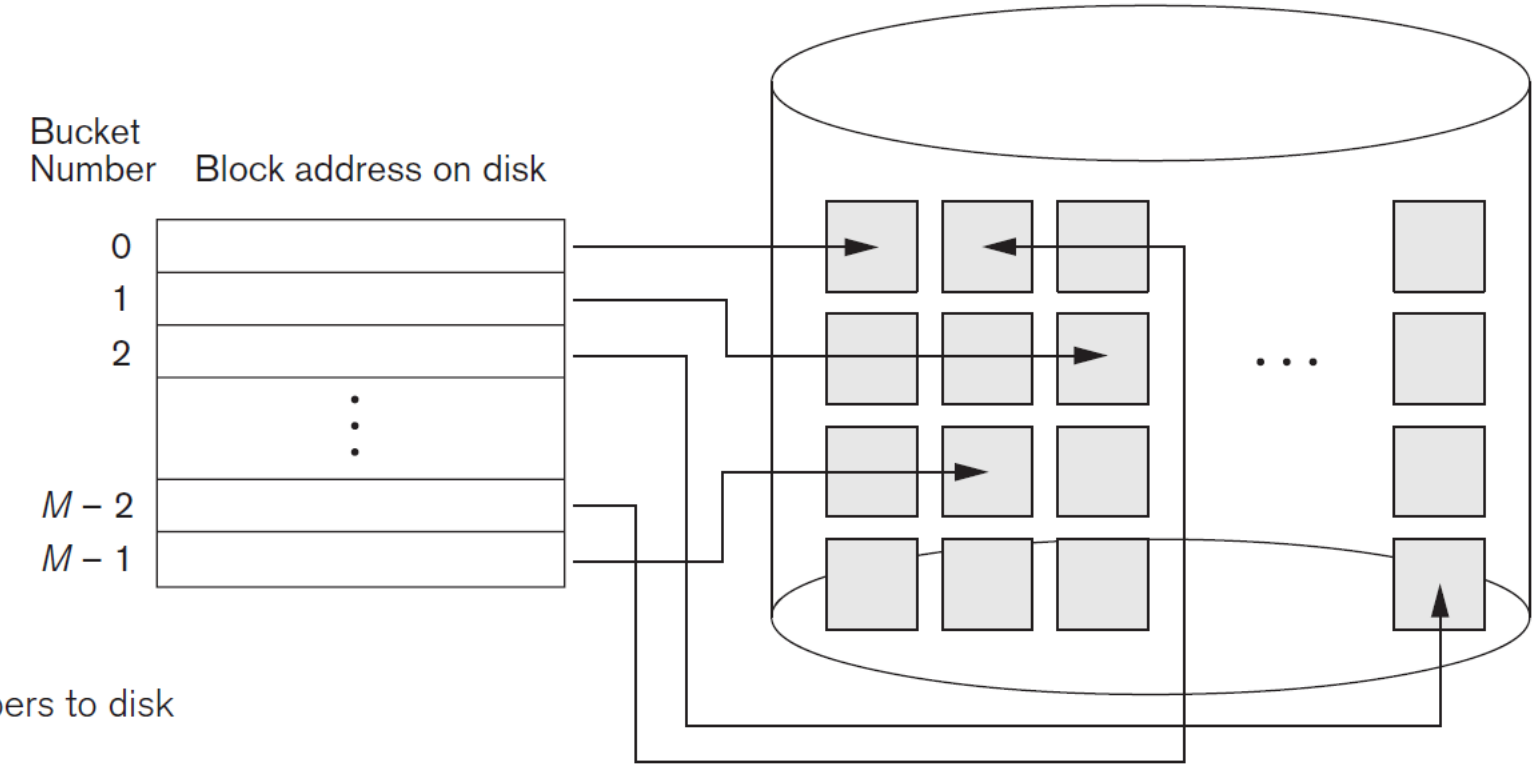


Figure 16.9
Matching bucket numbers to disk block addresses.

External Hashing (3/3)

- Collision problem: a bucket is filled to capacity and a new record being inserted hashes to that bucket.
- We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket. The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block.

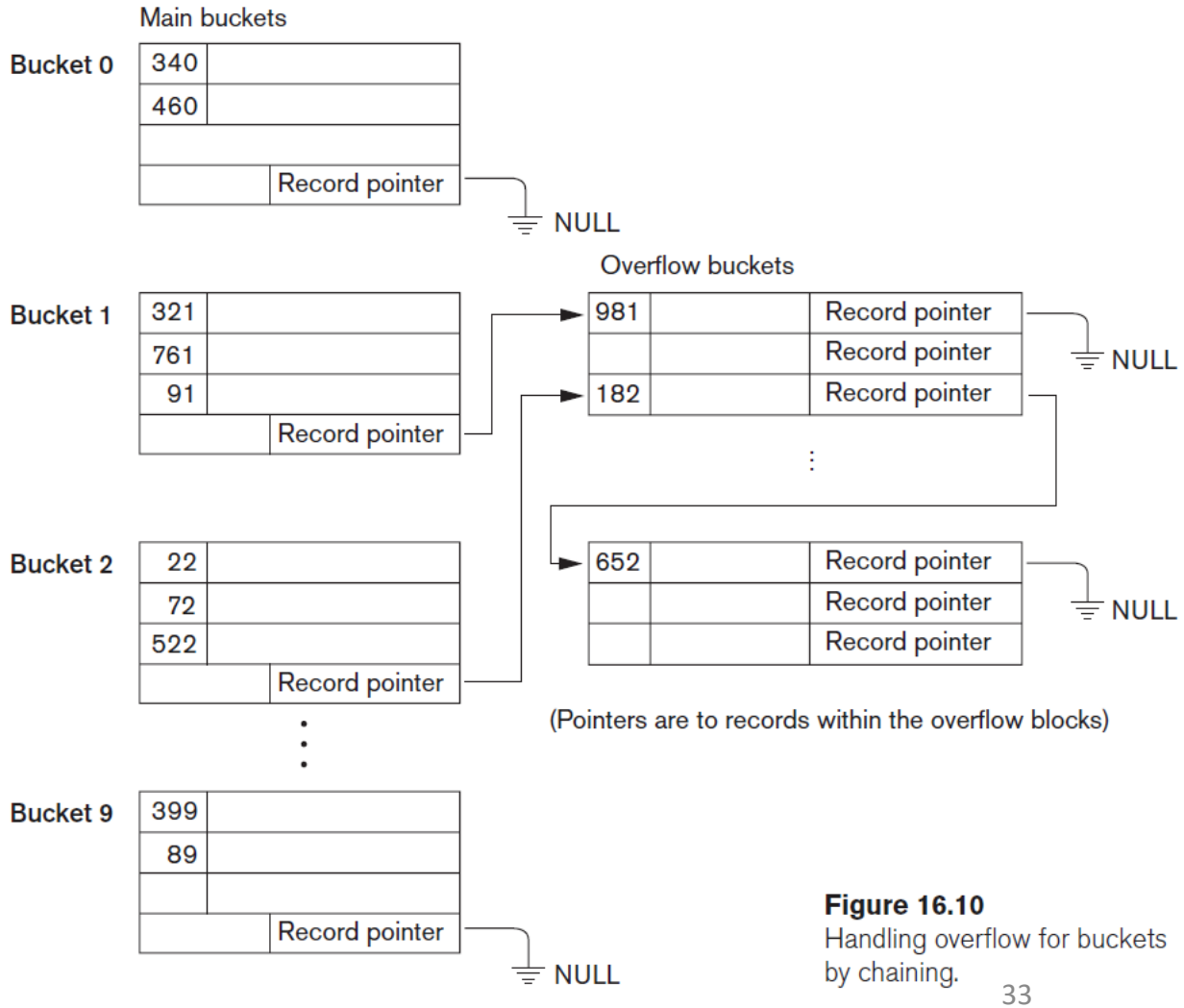


Figure 16.10
Handling overflow for buckets by chaining.

Static Hashing

- The hashing scheme described so far is called **static hashing** because a fixed number of buckets M is allocated. A major drawback of the static hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation.
- Two schemes attempt to remedy this situation
  - Extendible hashing
  - Dynamic hashing

Extendible Hashing (1/4)

- An array of 2^d bucket addresses—is maintained, where d is called the **global depth** of the directory. The integer value corresponding to the first (highorder) 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 in which the corresponding records are stored.
- A **local depth** d′—stored with each bucket—specifies the number of bits on which the bucket contents are based.

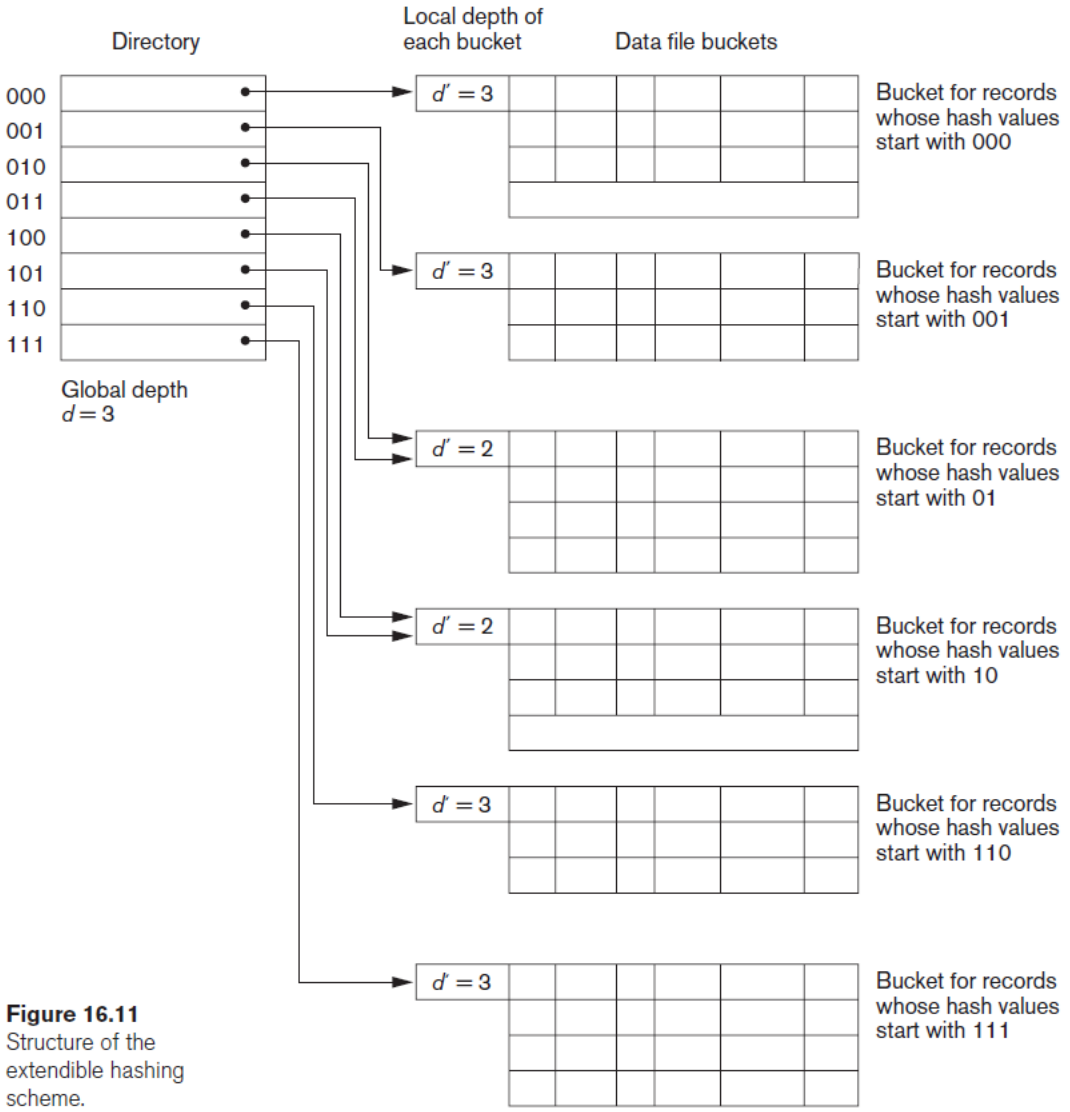


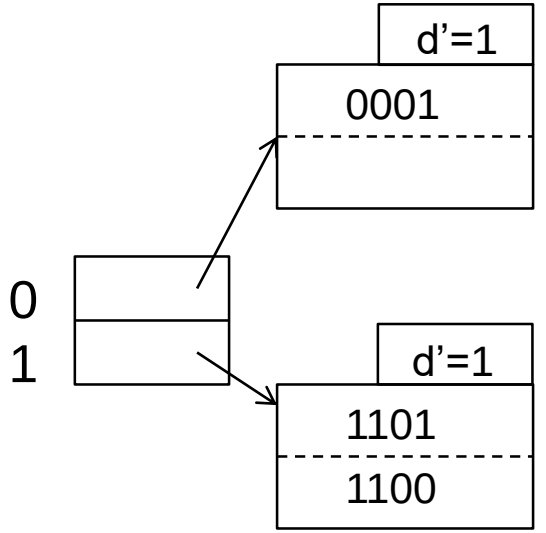
Figure 16.11
Structure of the extendible hashing scheme.

Extendible Hashing (2/4)

- The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array.
- Doubling is needed if a bucket, whose local depth d′ is equal to the global depth d, overflows. Halving occurs if d > d′ for all the buckets after some deletions occur.
- Most record retrievals require two block accesses—one to the directory and the other to the bucket.

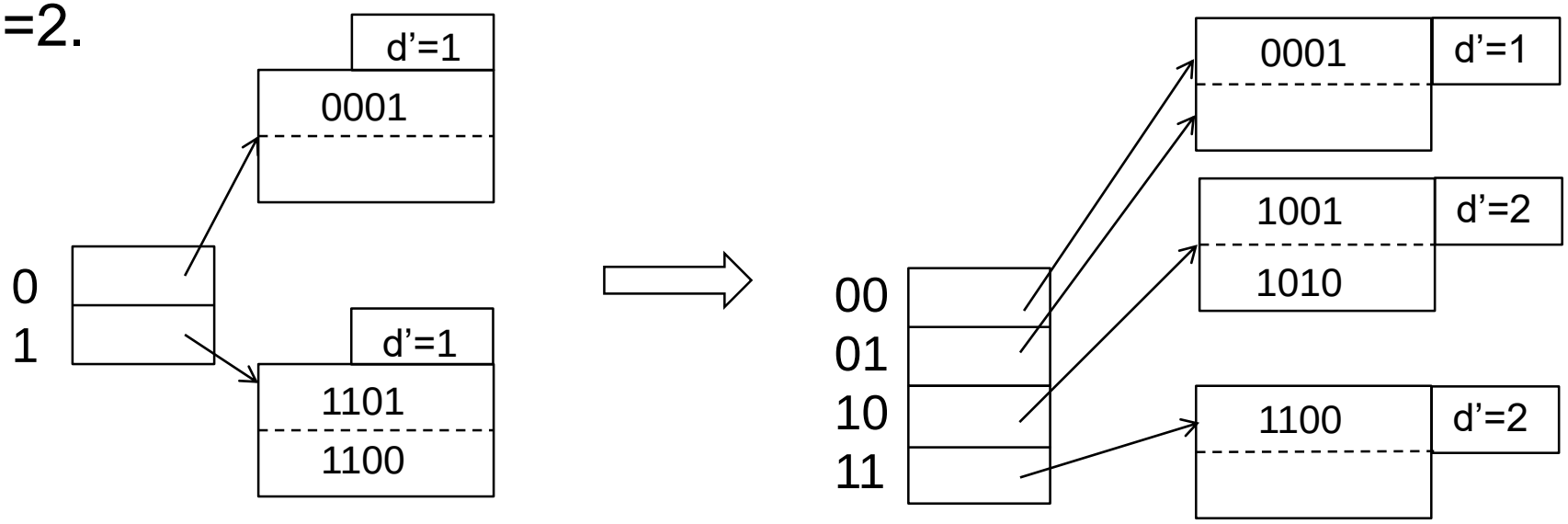
Extendible Hashing (3/4)

- Suppose d=4, i.e., the hash function produces a sequence of four bits
- Now, 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



Extendible Hashing (4/4)

- However, the second block is full. It needs to be split by setting d' to 2.
- The two entries beginning with 0 each point to the block for records whose hashed keys begin with 0 and the block still has d'=1 to indicate that only the first bit determines membership in the block.
- The blocks for records beginning with 1 needs to be split into 10 and 11, i.e., d'=2.



Dynamic Hashing (1/2)

- The addresses of the buckets were either the n high-order bits or n − 1 high-order bits, depending on the total number of keys belonging to the respective bucket.
- The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. Whereas extendible hashing uses the notion of global depth (high-order d bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth d − 1, dynamic hashing maintains a tree-structured directory with two types of nodes:
  - Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
  - Leaf nodes—these hold a pointer to the actual bucket with records.

Dynamic Hashing (2/2)

- An example of the dynamic hashing shows (“000”, “001”, “110”, and “111”) with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets (“01” and “10”) are shown with high-order 2-bit addresses (corresponding to the local depth of 2).
- The latter two are the result of collapsing the “010” and “011” into “01” and collapsing “100” and “101” into “10”.
- Note that the directory nodes are used implicitly to determine the “global” and “local” depths of buckets in dynamic hashing.

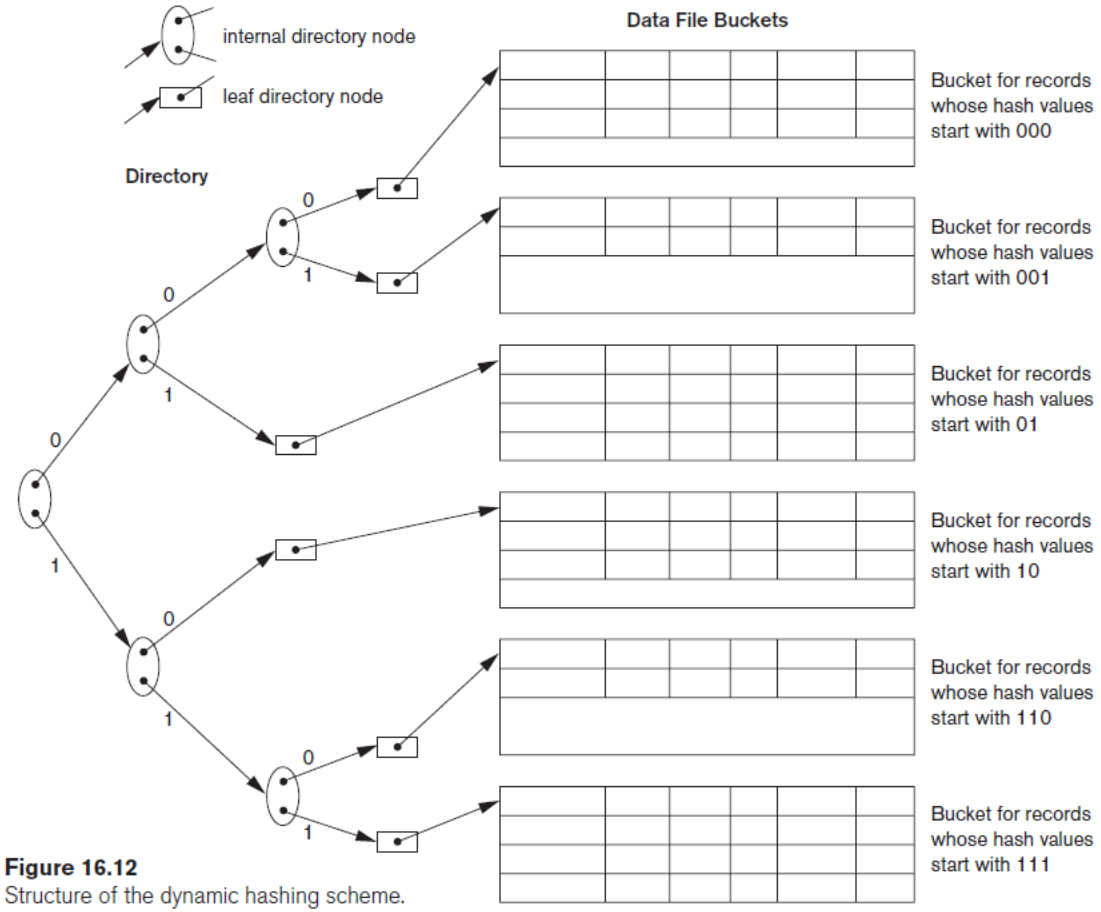


Figure 16.12
Structure of the dynamic hashing scheme.