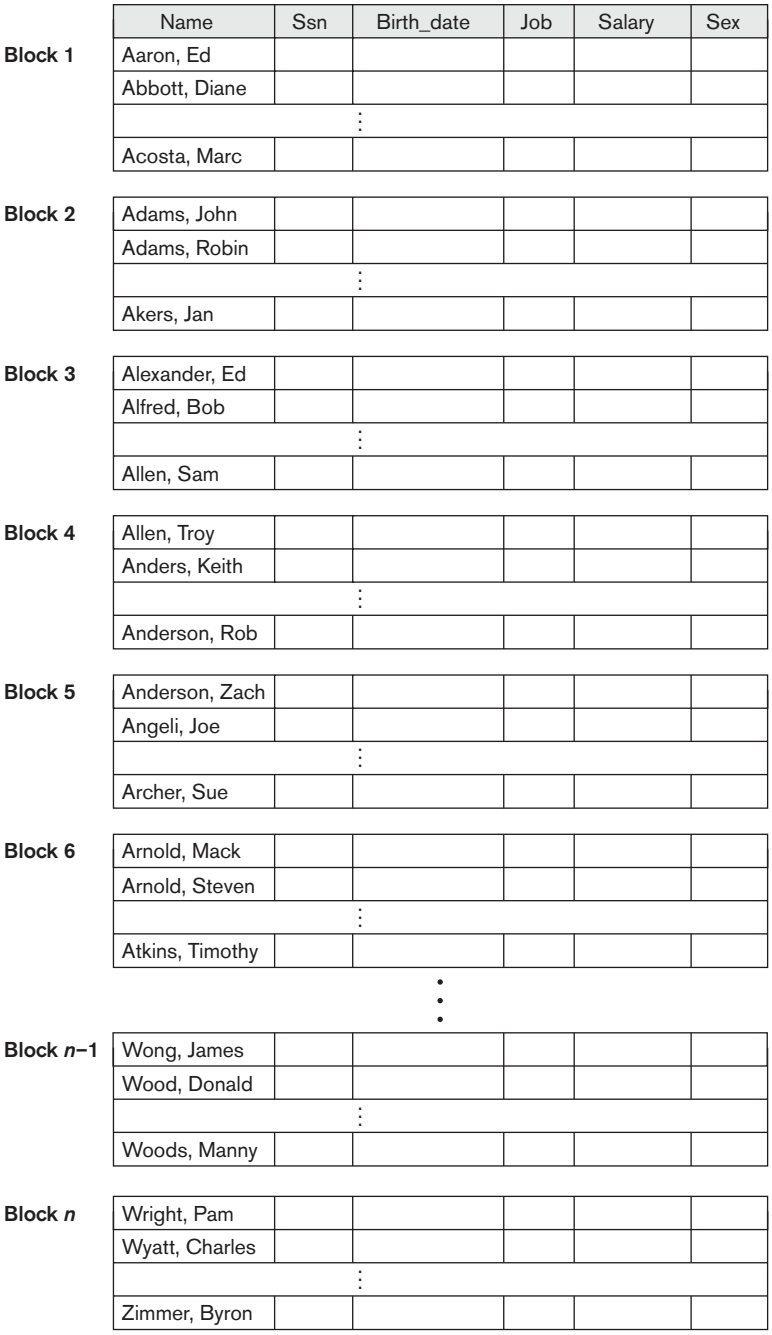

Figure 16.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

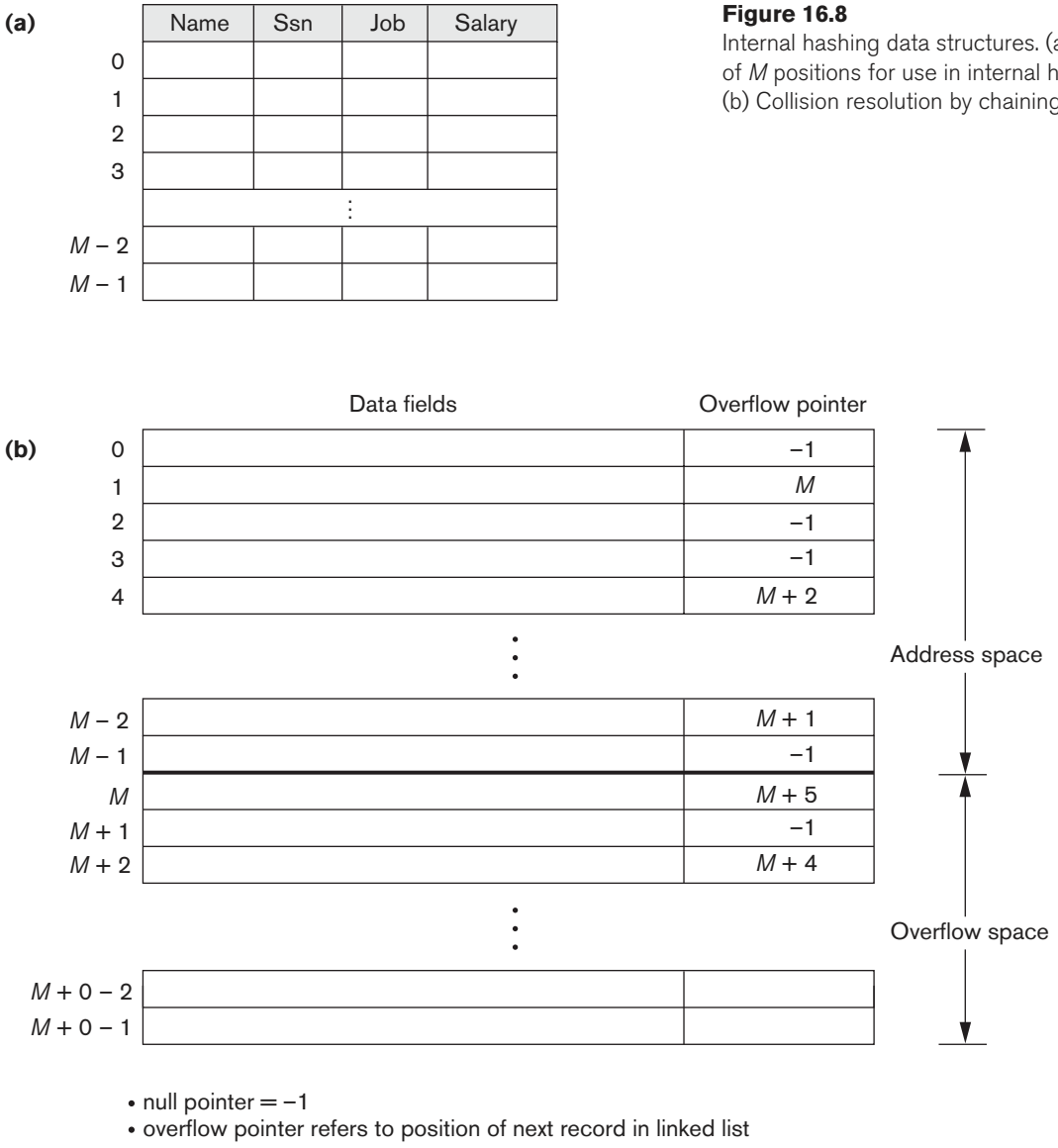

Figure 16.8
Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.

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. For a hash field whose data type is a string of 20 characters, Algorithm 16.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type *K: array* [1..20] *of char* (in Pascal) or *char K*[20] (in C).

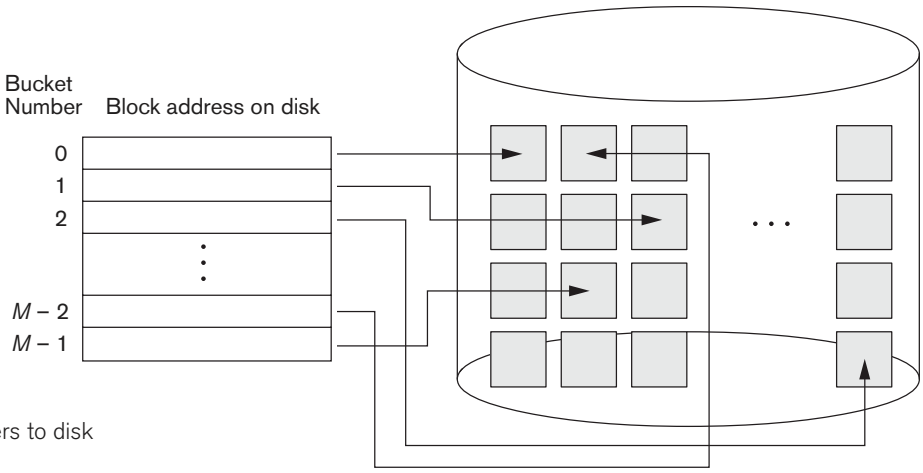

Figure 16.9
Matching bucket numbers to disk block addresses.

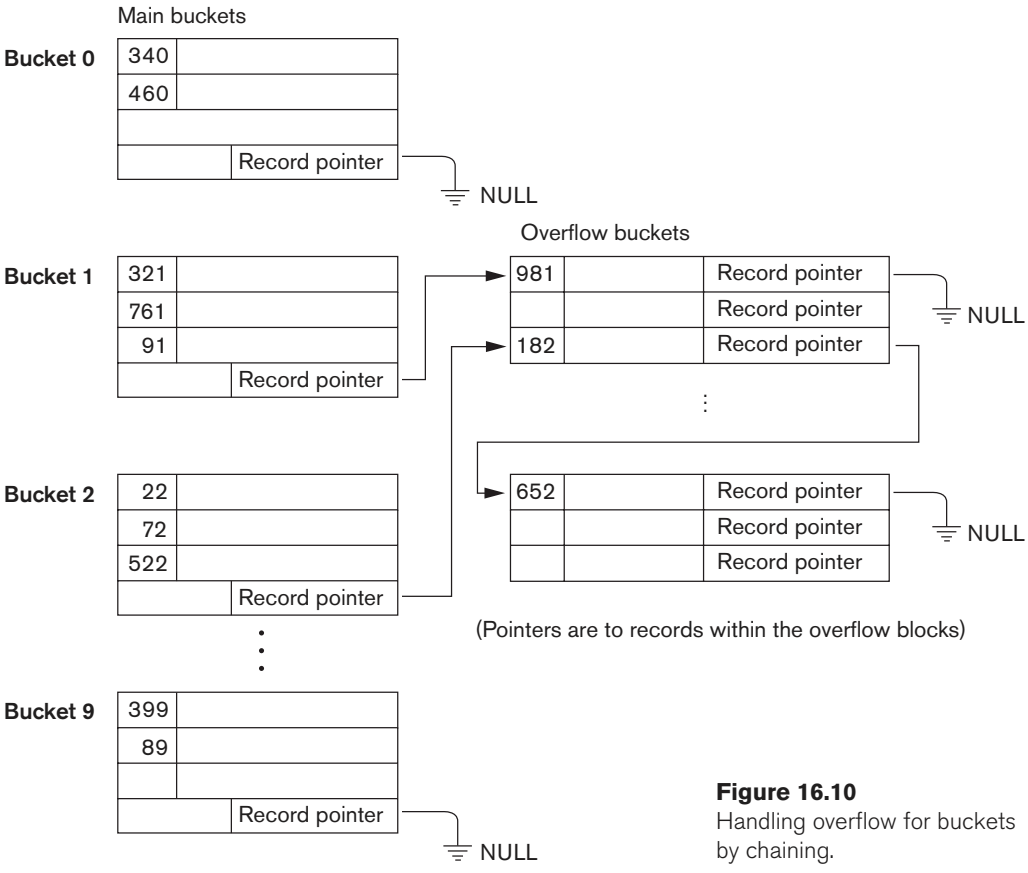

Figure 16.10
Handling overflow for buckets by chaining.

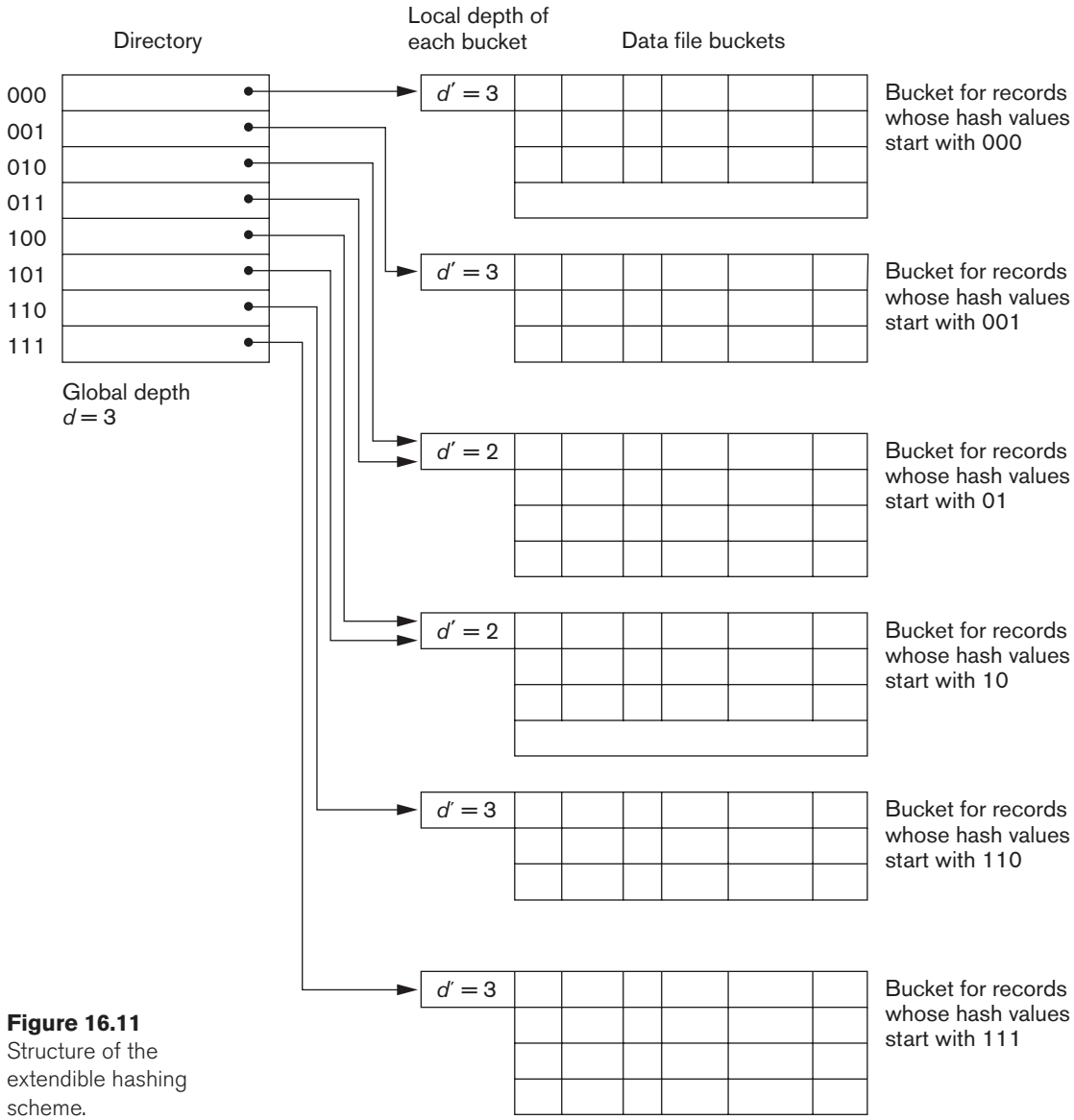

Figure 16.11
Structure of the extendible hashing scheme.

directory is also split into two locations, both of which have the same pointer value as did the original location.

The main advantage of extendible hashing that makes it attractive is that the *performance of the file does not degrade as the file grows*, as opposed to static external hashing, where collisions increase and the corresponding chaining effectively increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated

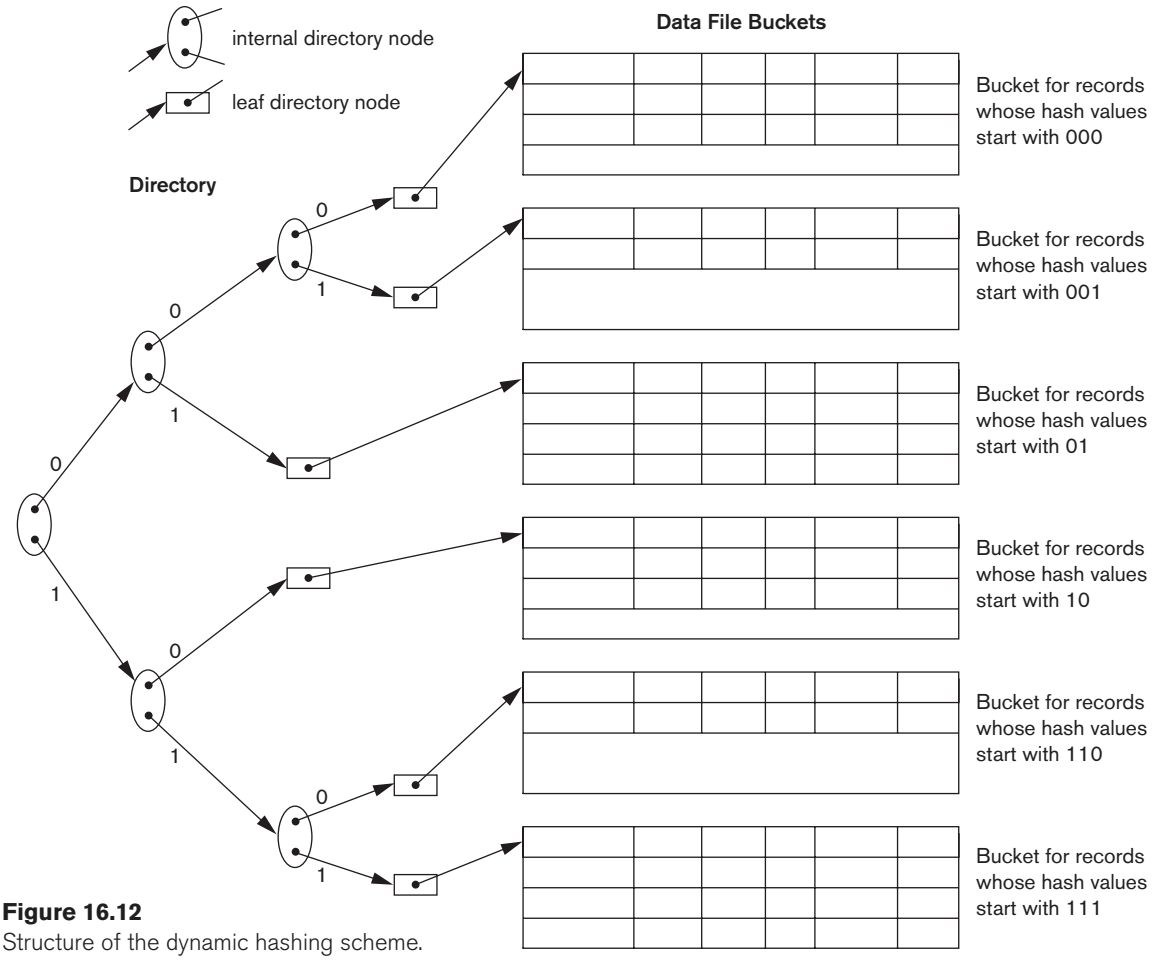

Figure 16.12
Structure of the dynamic hashing scheme.

based on h_i will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.

As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, … . If enough overflows occur, all the original file buckets 0, 1, … , $M - 1$ will have been split, so the file now has $2M$ instead of M buckets, and all buckets use the hash function h_{i+1}. Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a *delayed split* of their buckets. There is no directory; only a value n—which is initially set to 0 and is incremented by 1 whenever a split occurs—is needed to determine which buckets have been split. To retrieve a record with hash key value K, first apply the function h_i to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split. Initially, $n = 0$, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

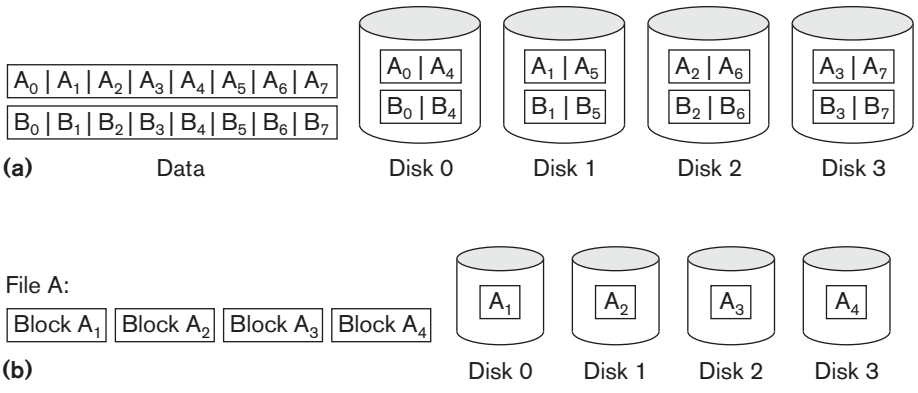

Figure 16.13
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.

disks using parity or some other error-correction code, reliability can be improved. In Sections 16.10.1 and 16.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 16.10.3 discusses RAID organizations and levels.

16.10.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTBF (mean time between failures) of a disk drive is assumed to be 200,000 hours or about 22.8 years (for the disk drive in Table 16.1 called Seagate Enterprise Performance 10K HDD, it is 1.4 million hours), the MTBF for a bank of 100 disk drives becomes only 2,000 hours or 83.3 days (for a bank of 1,000 Seagate Enterprise Performance 10K HDD disks it would be 1,400 hours or 58.33 days). Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

One technique for introducing redundancy is called **mirroring** or **shadowing**. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours; then the mean time to data loss of a mirrored disk system using 100 disks with MTBF of 200,000 hours each is $(200{,}000)^2/(2 * 24) = 8.33 * 10^8$ hours, which is 95,028 years.[18] Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.

[18]The formulas for MTBF calculations appear in Chen et al. (1994).