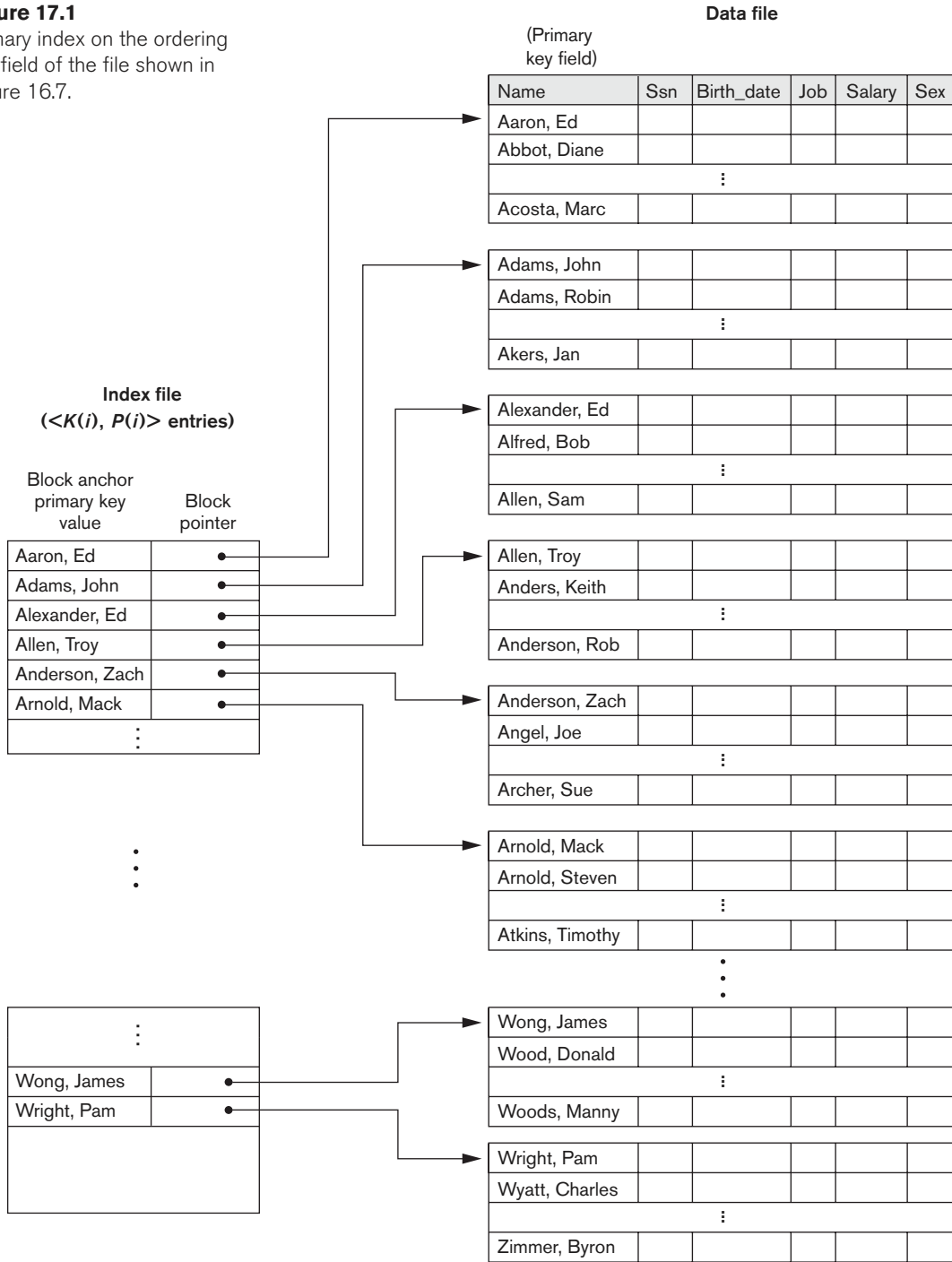


Figure 17.1
 Primary index on the ordering
 key field of the file shown in
 Figure 16.7.



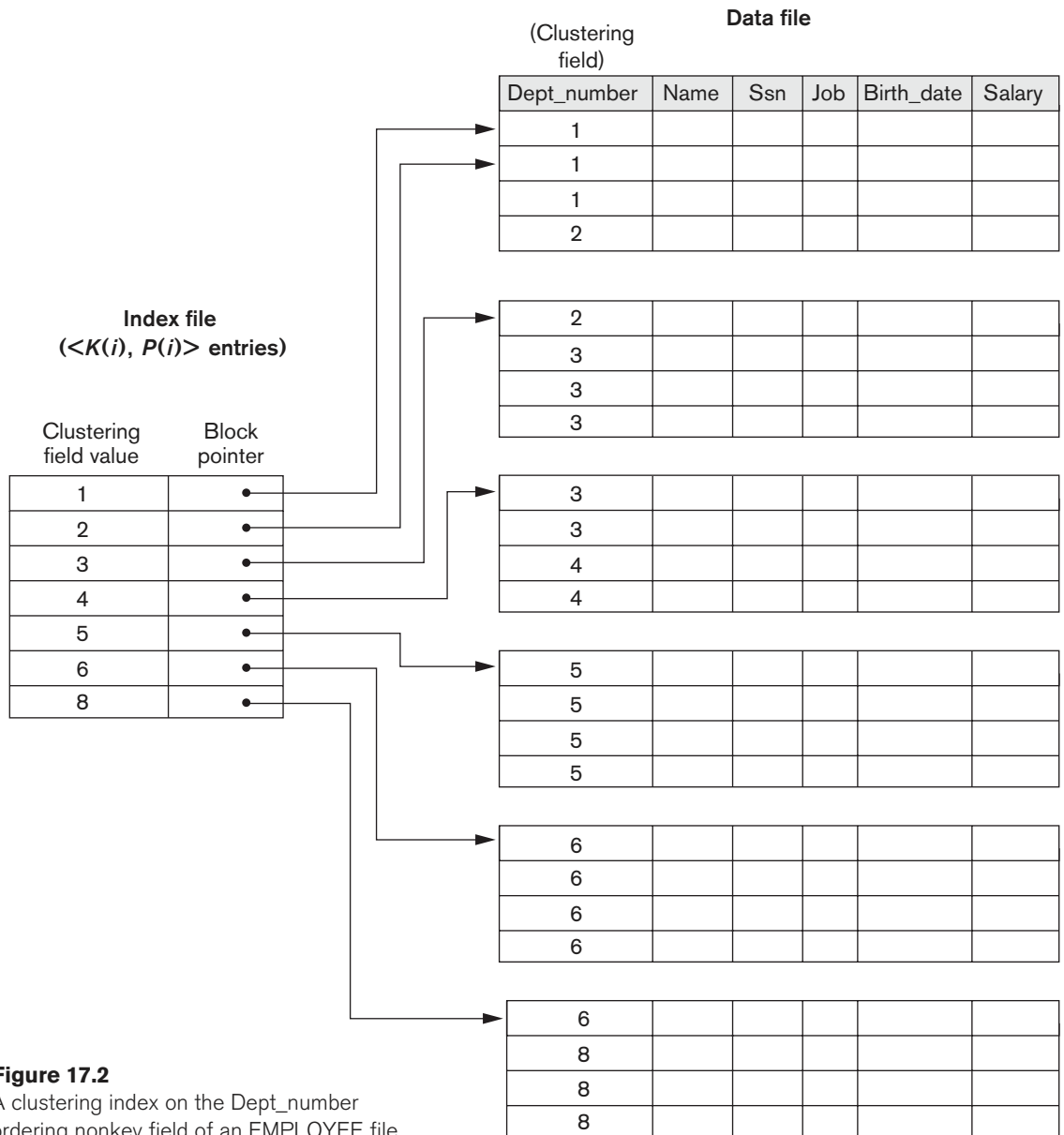


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

There is some similarity between Figures 17.1, 17.2, and 17.3 and Figures 16.11 and 16.12. An index is somewhat similar to dynamic hashing (described in Section 16.8.3) and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

Figure 17.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

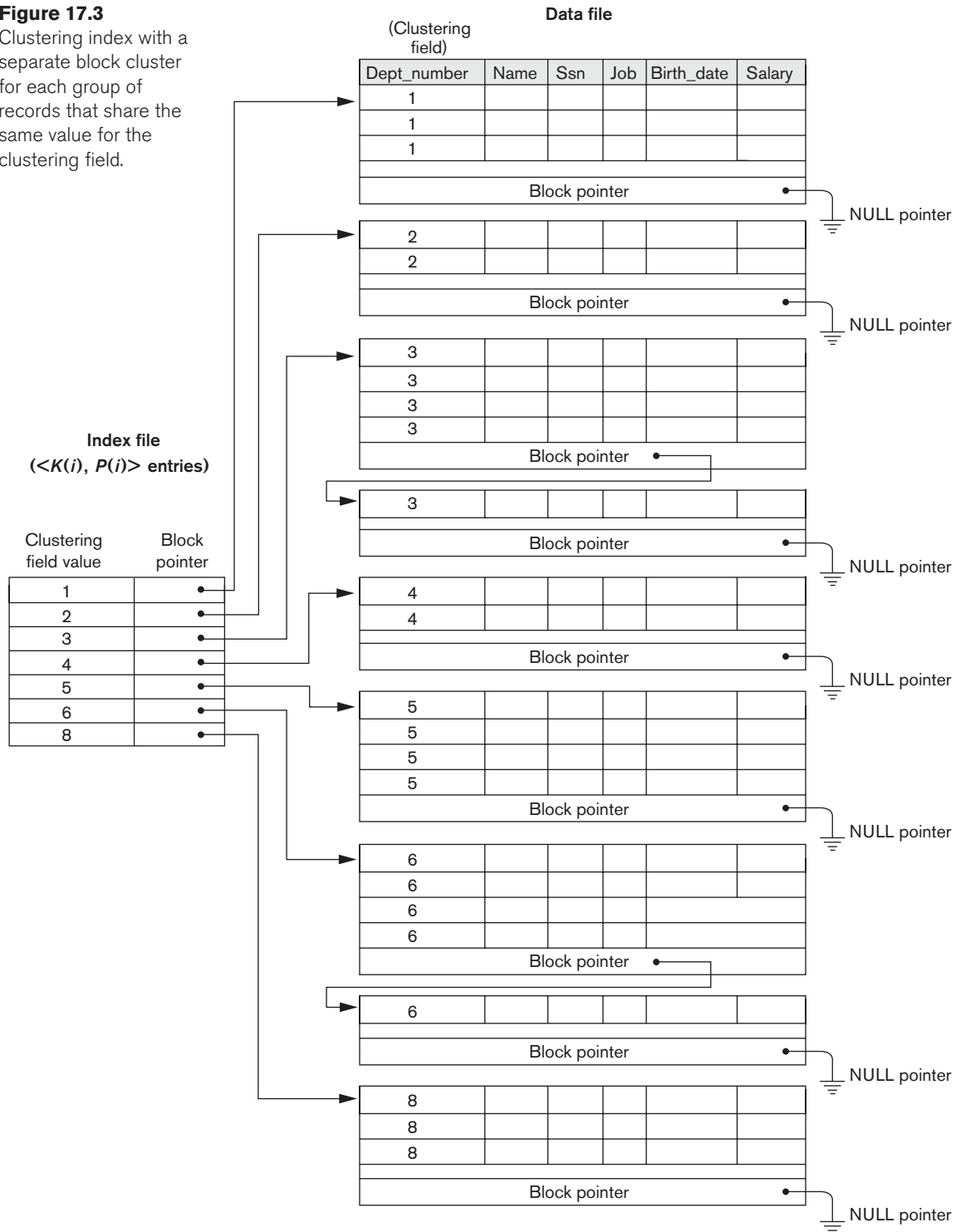


Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no ^a
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records ^b or number of distinct index field values ^c	Dense or Nondense	No

^aYes if every distinct value of the ordering field starts a new block; no otherwise.

^bFor option 1.

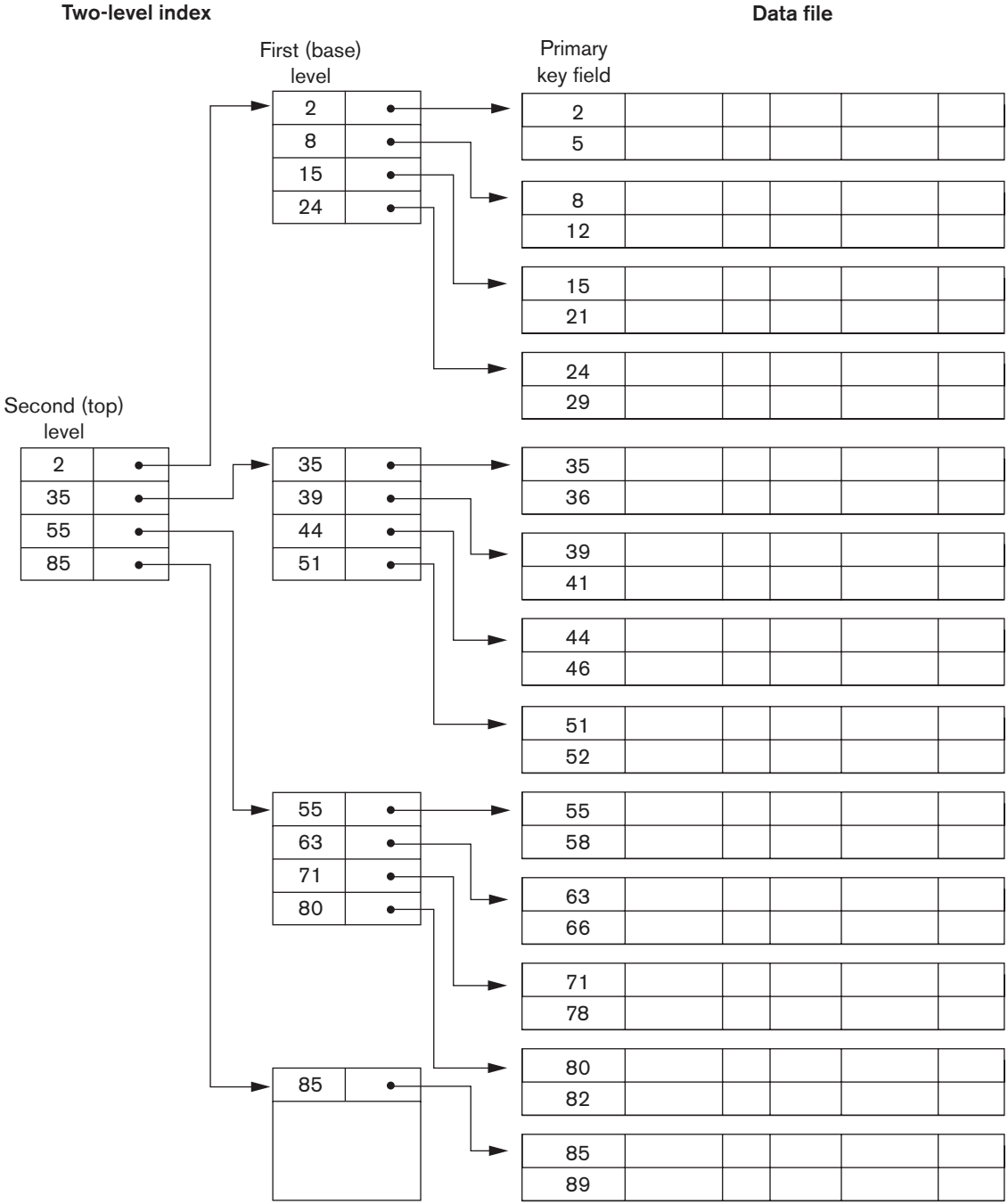
^cFor options 2 and 3.

17.2 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. A binary search requires approximately $(\log_2 b_i)$ block accesses for an index with b_i blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by bfr_i , the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the **fan-out** of the multilevel index, and we will refer to it by the symbol **fo**. Whereas we divide the *record search space* into two halves at each step during a binary search, we divide it n -ways (where $n =$ the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately $(\log_{fo} b_i)$ block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2. Given a blocksize of 4,096, which is most common in today's DBMSs, the fan-out depends on how many (key + block pointer) entries fit within a block. With a 4-byte block pointer (which would accommodate $2^{32} - 1 = 4.2 \times 10^9$ blocks) and a 9-byte key such as SSN, the fan-out comes to 315.

A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a multilevel index, as an *ordered file* with a *distinct value* for each

Figure 17.6
 A two-level primary index resembling ISAM (indexed sequential access method) organization.



higher-capacity indexes. In the DBMSs prevalent in the market today, the common structure used for indexing is B⁺-trees.

17.3.1 Search Trees and B-Trees

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes discussed in Section 17.2 can be thought of as a variation of a search tree; each node in the multilevel index can have as many as fo pointers and fo key values, where fo is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

Search Trees. A search tree is slightly different from a multilevel index. A **search tree of order p** is a tree such that each node contains *at most* $p - 1$ search values and p pointers in the order $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values. All search values are assumed to be unique.⁸ Figure 17.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

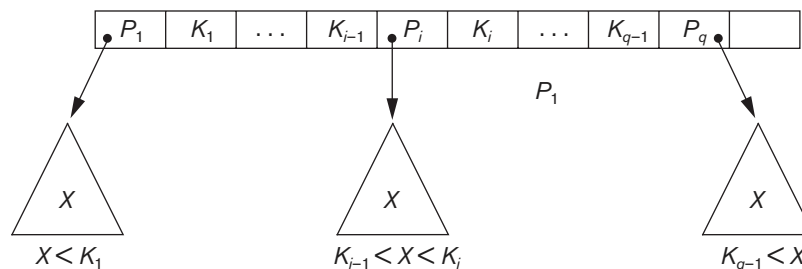
1. Within each node, $K_1 < K_2 < \dots < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i , we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 17.8).

Whenever we search for a value X , we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 17.9 illustrates a search tree of order $p = 3$ and integer search values. Notice that some of the pointers P_i in a node may be NULL pointers.

We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the

Figure 17.8

A node in a search tree with pointers to subtrees below it.



⁸This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.

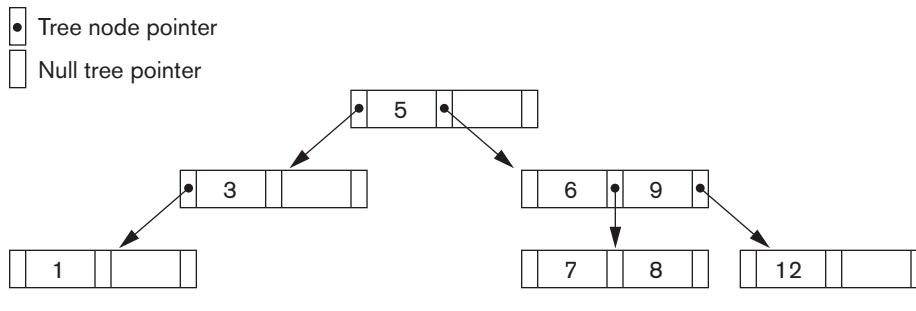


Figure 17.9
A search tree of order $p = 3$.

search field (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level.⁹ The tree in Figure 17.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

- To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels
- To make the search speed uniform, so that the average time to find any random key is roughly the same

Minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees. The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from

⁹The definition of *balanced* is different for binary trees. Balanced binary trees are known as *AVL trees*.