Chapter 18 ***Indexing Structures for Files***

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries <field value, pointer to record>, which is ordered by field value
- The index is called an access path on the field.

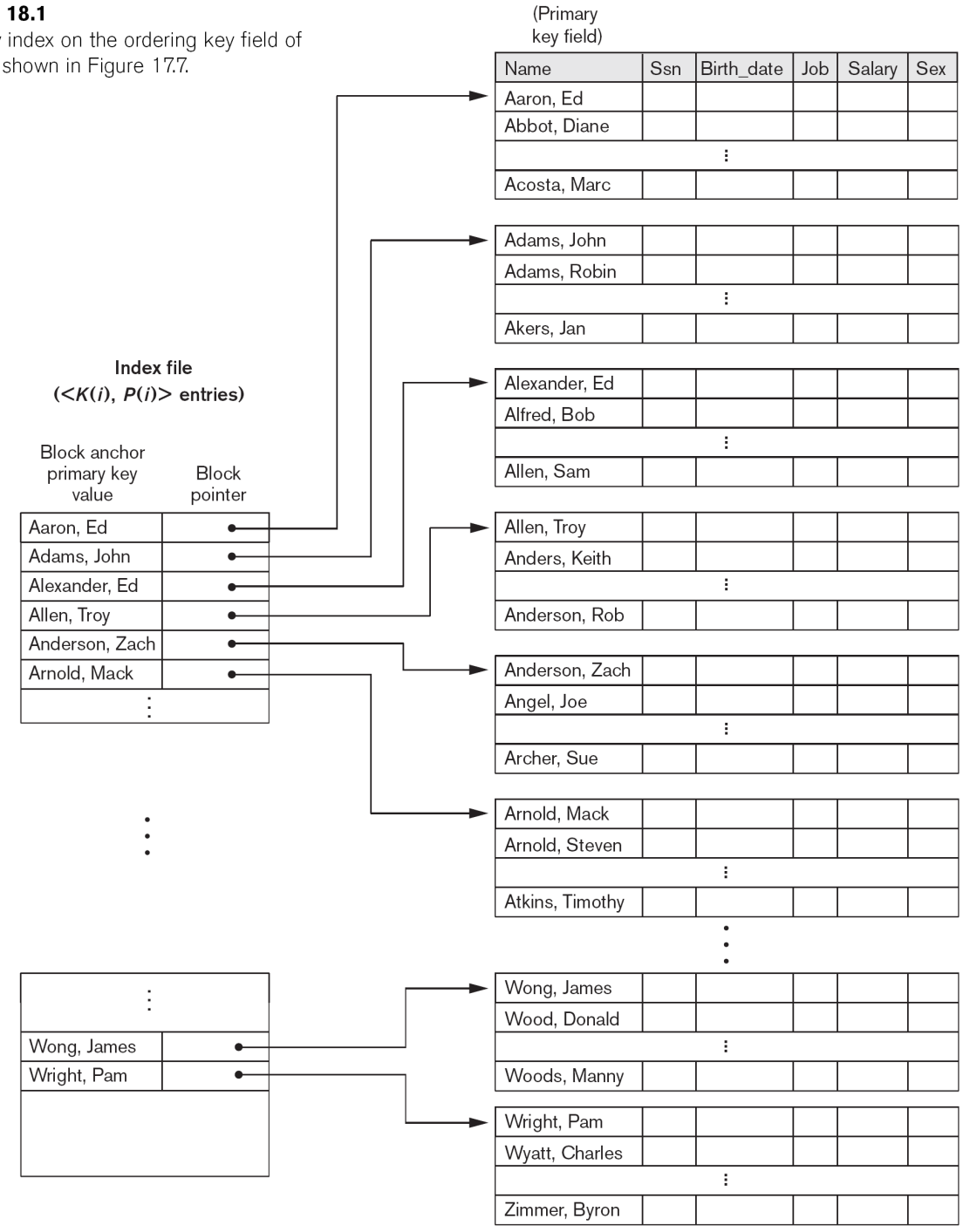
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A dense index has an index entry for every search key value (and hence every record) in the data file.
  - A sparse (or nondense) index, on the other hand, has index entries for only some of the search values

- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that:
  - record size R=150 bytes block size B=512 bytes r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes. Then:
  - index entry size R_I=(V_{SSN}+ P_R)=(9+7)=16 bytes
  - index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ Bfr_I)= (30000/32)= 938 blocks
  - binary search needs $\log_2$bI= $\log_2$938= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2$b= $\log_2$30000= 15 block accesses

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a key field
  - Includes one index entry for each block in the data file; the index entry has the key field value for the first record in the block, which is called the block anchor
  - A similar scheme can use the last record in a block.
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a non-key field unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry for each distinct value of the field; the index entry points to the first data block that contains records with that field value.

- It is another example of nondense index where Insertion and Deletion is relatively straightforward with a clustering index.

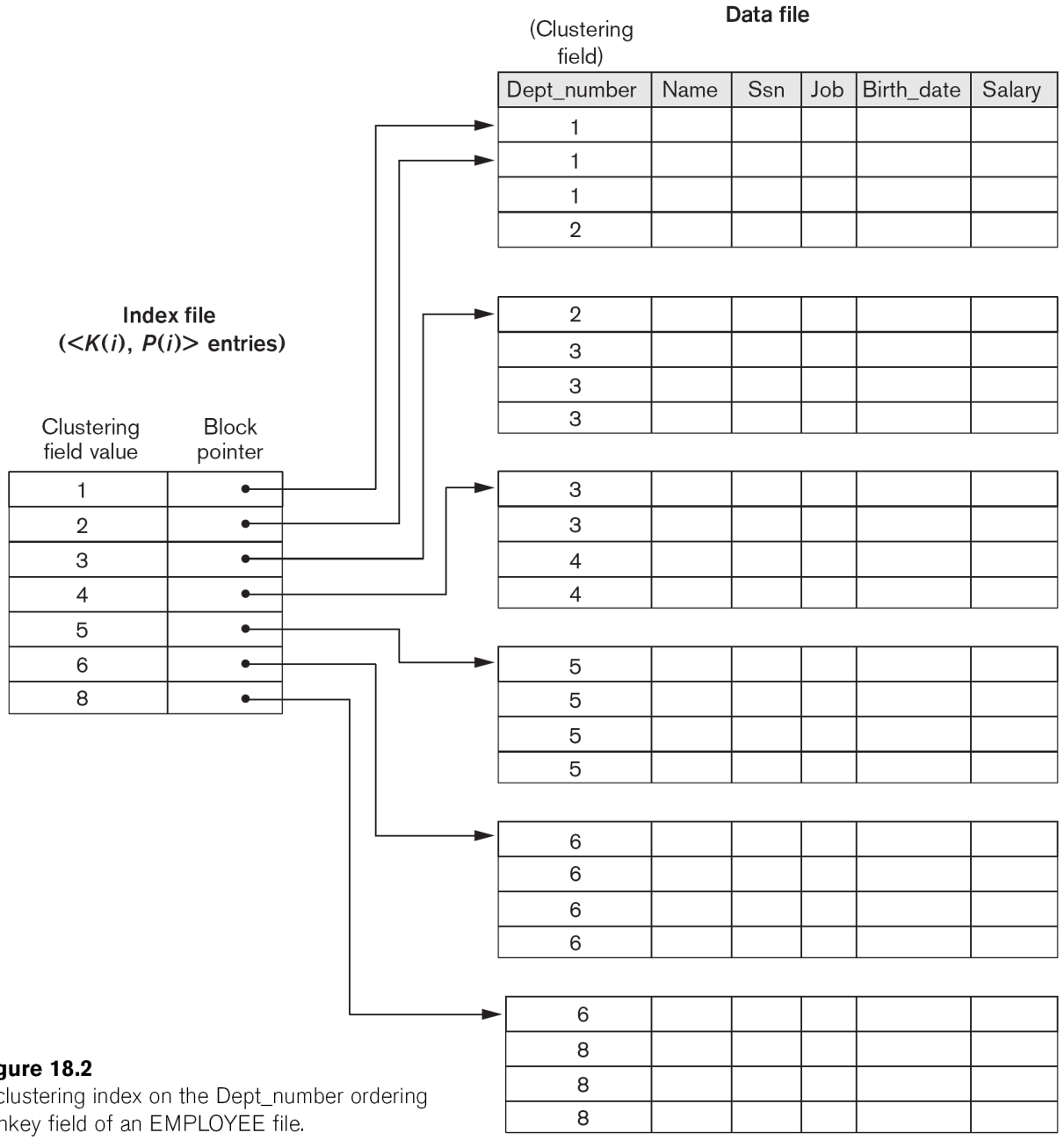
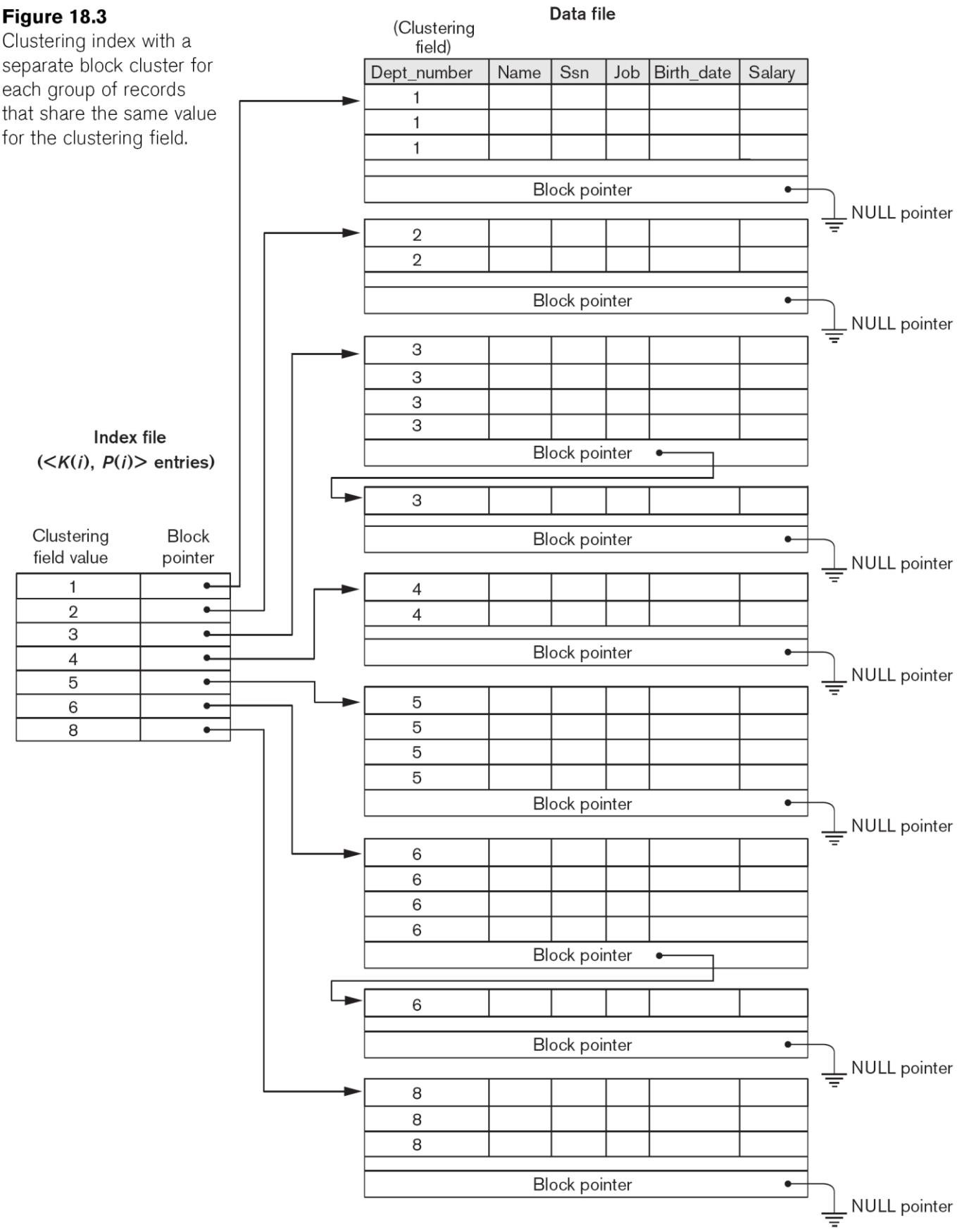


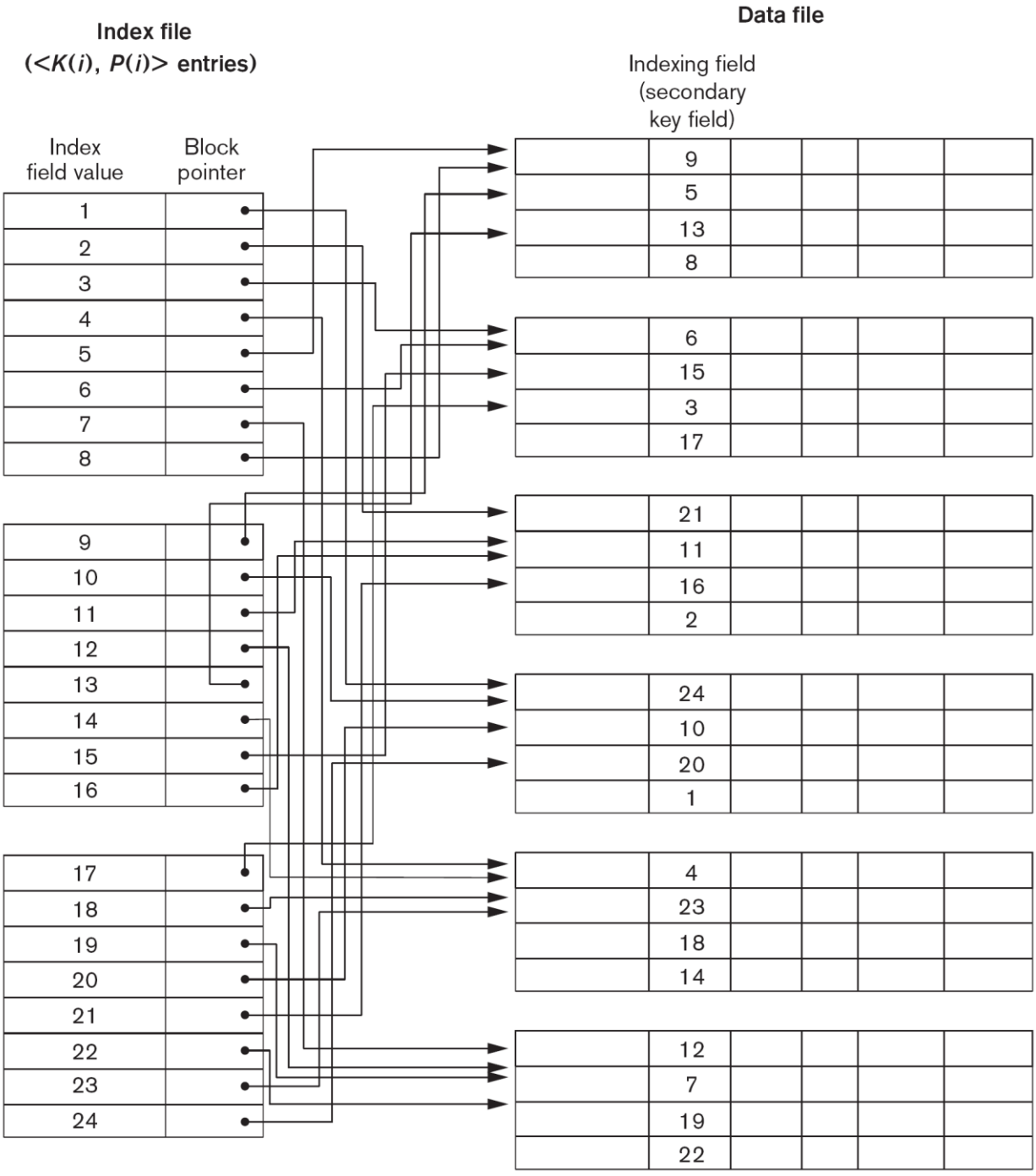
Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



- Secondary Index
    - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
    - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
    - The index is an ordered file with two fields.
        - The first field is of the same data type as some non-ordering field of the data file that is an indexing field.
        - The second field is either a block pointer or a record pointer.
        - There can be many secondary indexes (and hence, indexing fields) for the same file.
    - Includes one entry for each record in the data file; hence, it is a dense index

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Properties of Index Types

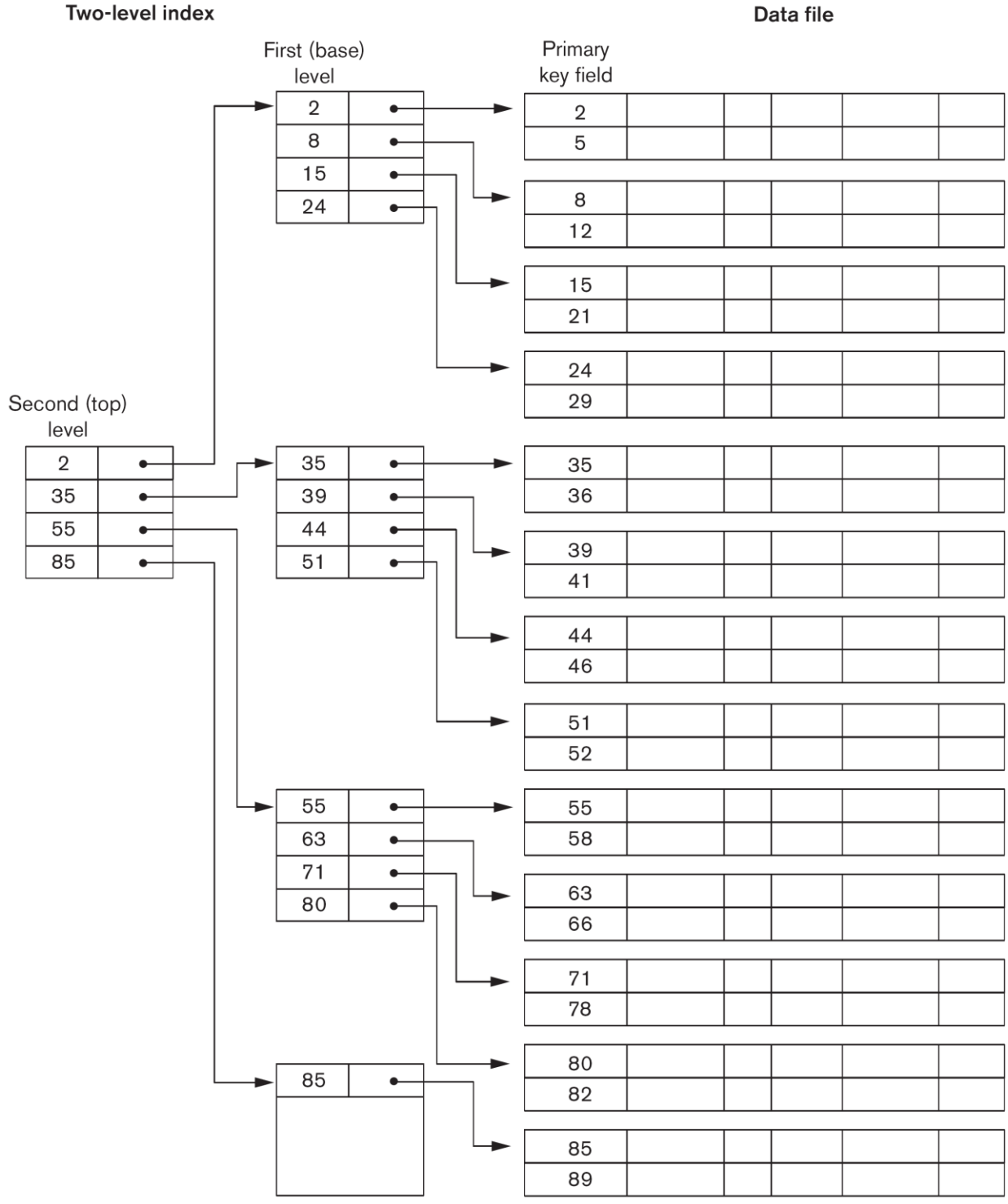
Table 18.2 Properties of Index Types

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

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index to the index itself;
    - In this case, the original index file is called the first-level index and the index to the index is called the second-level index.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the top level fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



- Such a multi-level index is a form of *search tree*
    - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

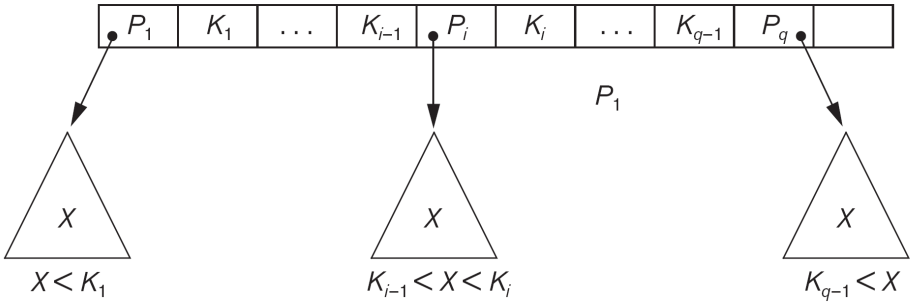


Figure 18.8
A node in a search tree with pointers to subtrees below it.

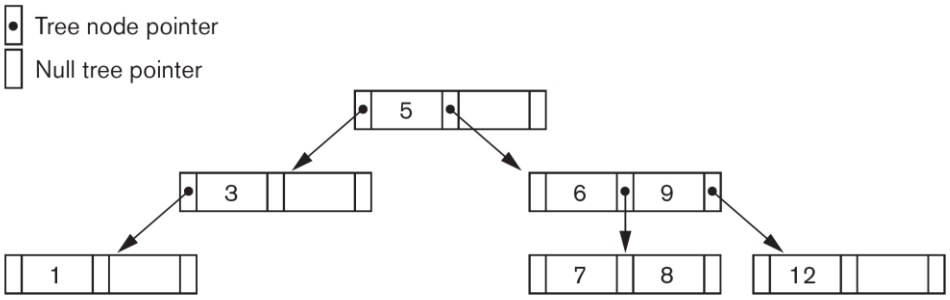


Figure 18.9
A search tree of order $p = 3$.

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion an deletion problem
    - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full
- An insertion into a node that is not full is quite efficient
    - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes
- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B+ Tree Examples

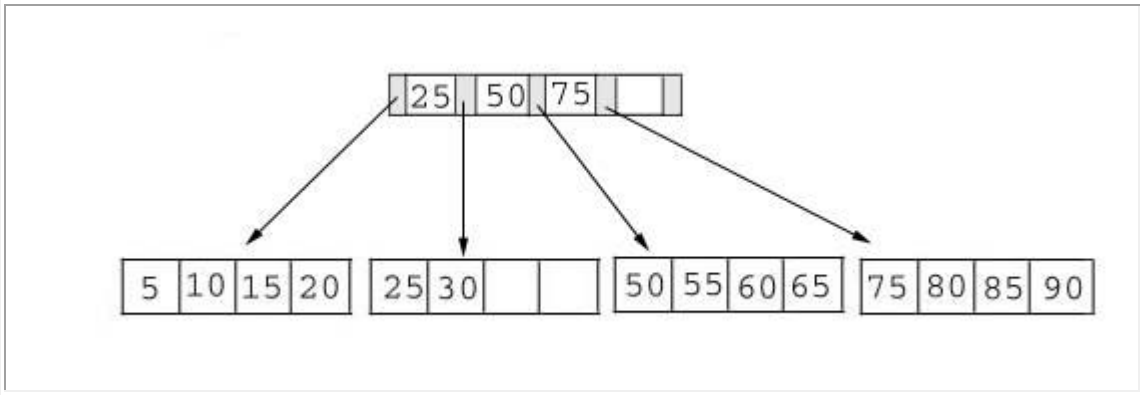
B+ Trees A B+ Tree combines features of ISAM and B Trees. It contains index pages and data pages. The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always index pages. These features are similar to ISAM. Unlike ISAM, overflow pages are not used in B+ trees. The index pages in a B+ tree are constructed through the process of inserting and deleting records. Thus, B+ trees grow and contract like their B Tree counterparts. The contents and the number of index pages reflects this growth and shrinkage.

B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for any B+ or B tree. As our example we use the smallest page structure. This means that our B+ tree conforms to the following guidelines.

Number of Keys/page	4
Number of Pointers/page	5
Fill Factor	50%
Minimum Keys in each page	2

As this table indicates each page must have a minimum of two keys. The root page may violate this rule. The following table shows a B+ tree. As the example illustrates this tree does not have a full index page. (We have room for one more key and pointer in the root page.) In addition, one of the data pages contains empty slots.

B+ Tree with four keys



Adding Records to a B+ Tree

The key value determines a record's placement in a B+ tree. The leaf pages are maintained in sequential order AND a doubly linked list (not shown) connects each leaf page with its sibling page(s). This doubly linked list speeds data movement as the pages grow and contract.
We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action in the insert algorithm. The scenarios are:

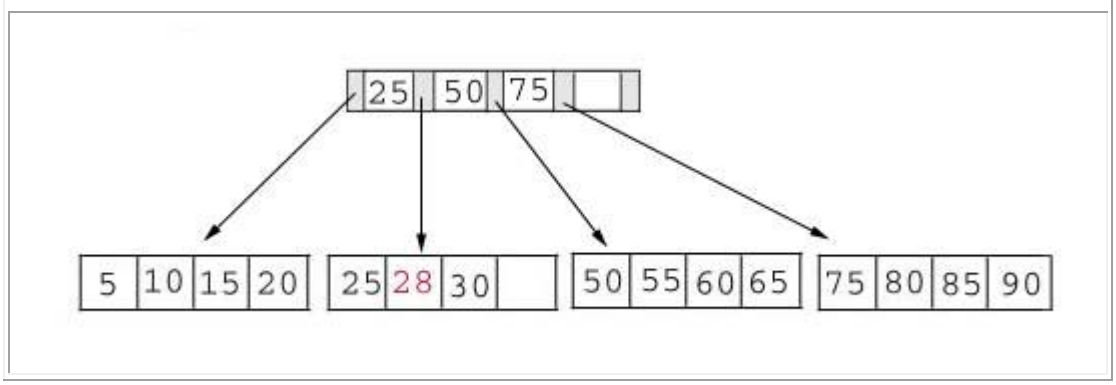
The `insert` algorithm for B+ Trees

Leaf Page Full	**Index Page FULL**	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

Illustrations of the insert algorithm

The following examples illustrate each of the **insert** scenarios. We begin with the simplest scenario: inserting a record into a leaf page that is not full. Since only the leaf node containing 25 and 30 contains expansion room, we're going to insert a record with a key value of 28 into the B+ tree. The following figures show the result of this addition.

Add Record with Key 28



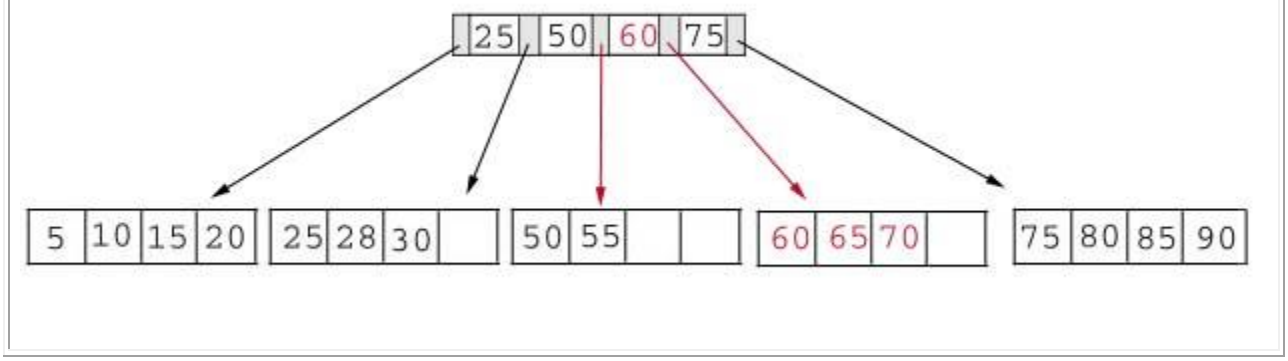
Adding a record when the leaf page is full but the index page is not

Next, we're going to insert a record with a key value of 70 into our B+ tree. This record should go in the leaf page containing 50, 55, 60, and 65. Unfortunately this page is full. This means that we must split the page as follows:

Left Leaf Page	Right Leaf Page
50 55	60 65 70

The middle key of 60 is placed in the index page between 50 and 75.
The following table shows the B+ tree after the addition of 70.

Add Record with Key 70



Adding a record when both the leaf page and the index page are full

As our last example, we're going to add a record containing a key value of 95 to our B+ tree. This record belongs in the page containing 75, 80, 85, and 90. Since this page is full we split it into two pages:

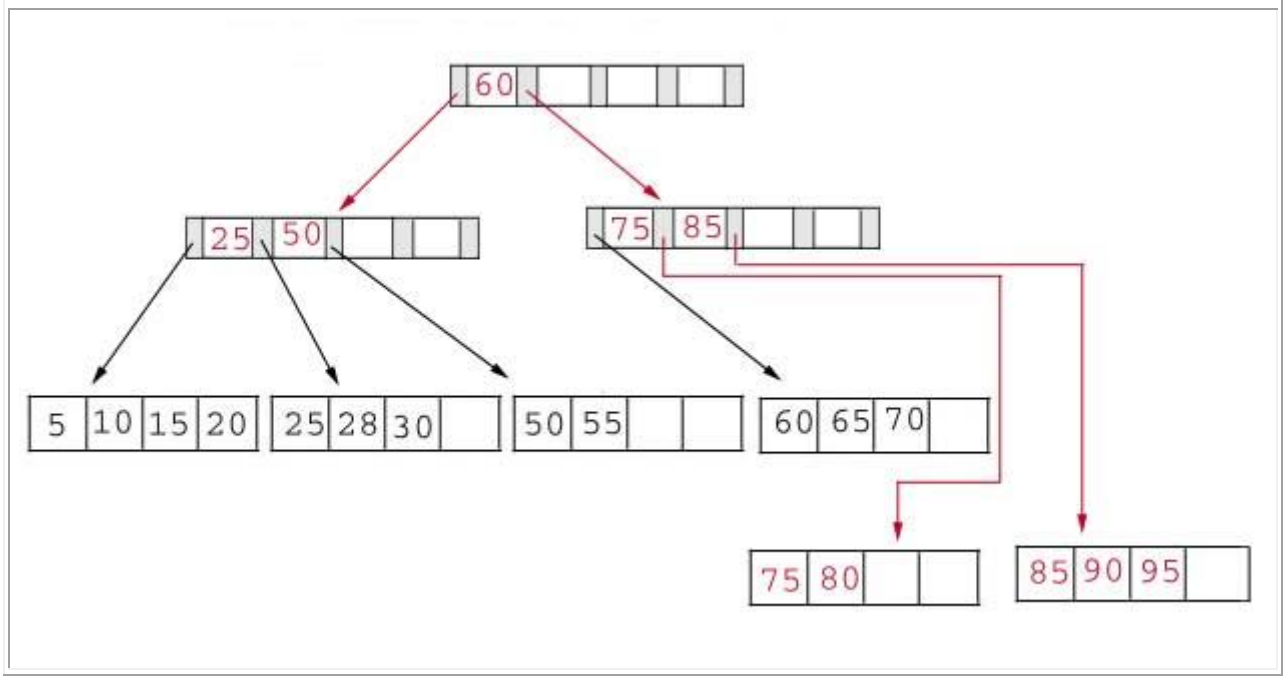
Left Leaf Page	Right Leaf Page
75 80	85 90 95

The middle key, 85, rises to the index page. Unfortunately, the index page is also full, so we split the index page:

Left Index Page	Right Index Page	New Index Page
25 50	75 85	60

The following table illustrates the addition of the record containing 95 to the B+ tree.

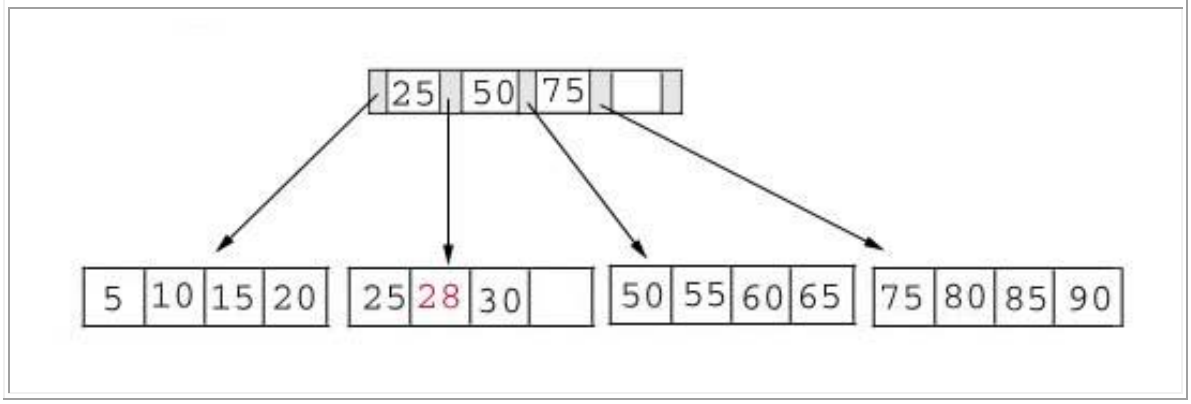
Add Record with Key 95



Rotation

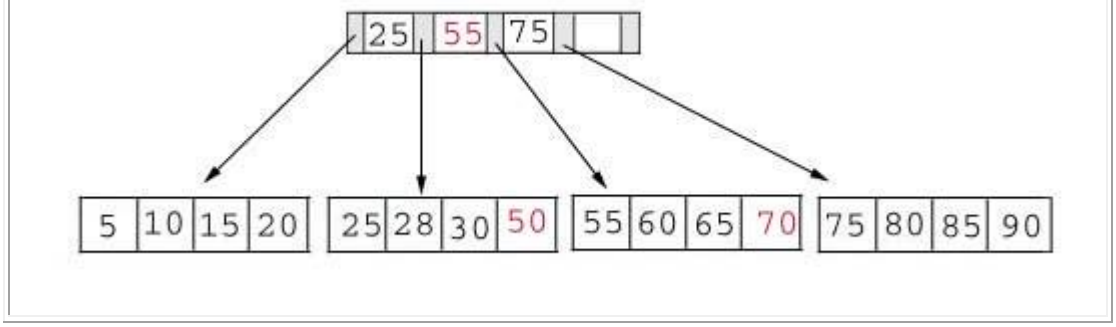
B+ trees can incorporate rotation to reduce the number of page splits. A rotation occurs when a leaf page is full, but one of its sibling pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting the indices as necessary. Typically, the left sibling is checked first (if it exists) and then the right sibling.
As an example, consider the B+ tree before the addition of the record containing a key of 70. As previously stated this record belongs in the leaf node containing 50 55 60 65. Notice that this node is full, but its left sibling is not.

Add Record with Key 28



Using rotation we shift the record with the lowest key to its sibling. Since this key appeared in the index page we also modify the index page. The new B+ tree appears in the following table.

Illustration of Rotation



Deleting Keys from a B+ tree

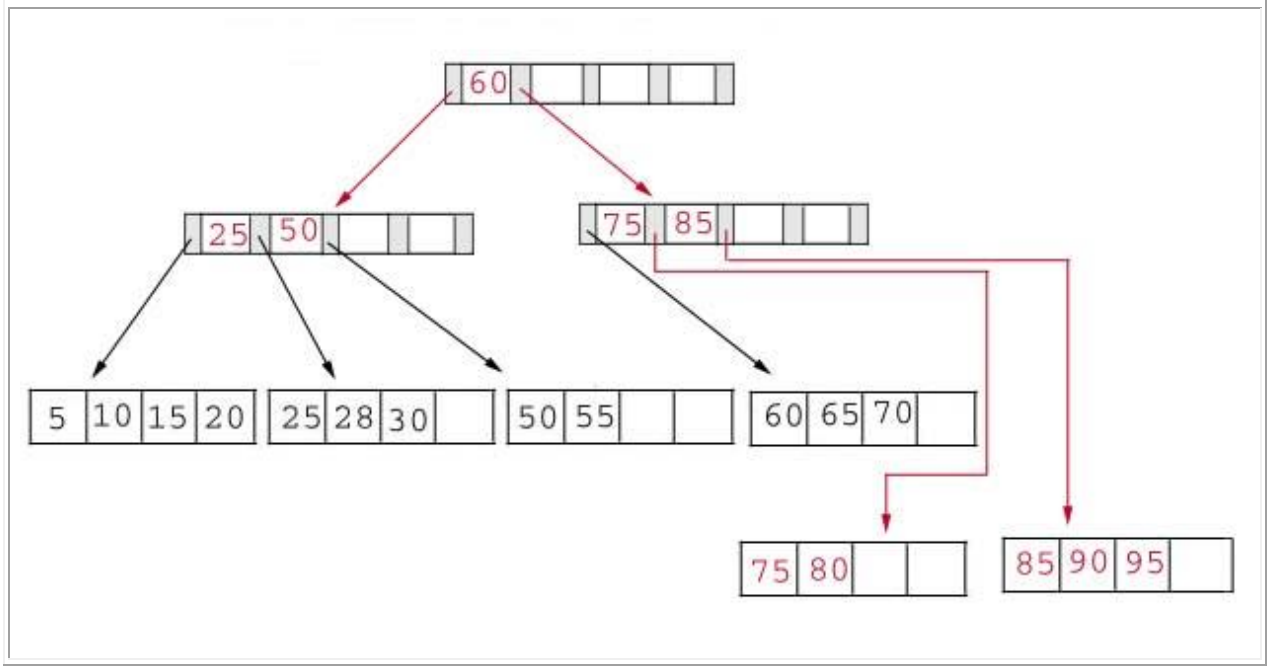
We must consider three scenarios when we delete a record from a B+ tree. Each scenario causes a different action in the delete algorithm. The scenarios are:

The `delete` algorithm for B+ Trees

Leaf Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

As our example, we consider the B+ tree after we added 95 as a key. As a refresher this tree is printed in the following table.

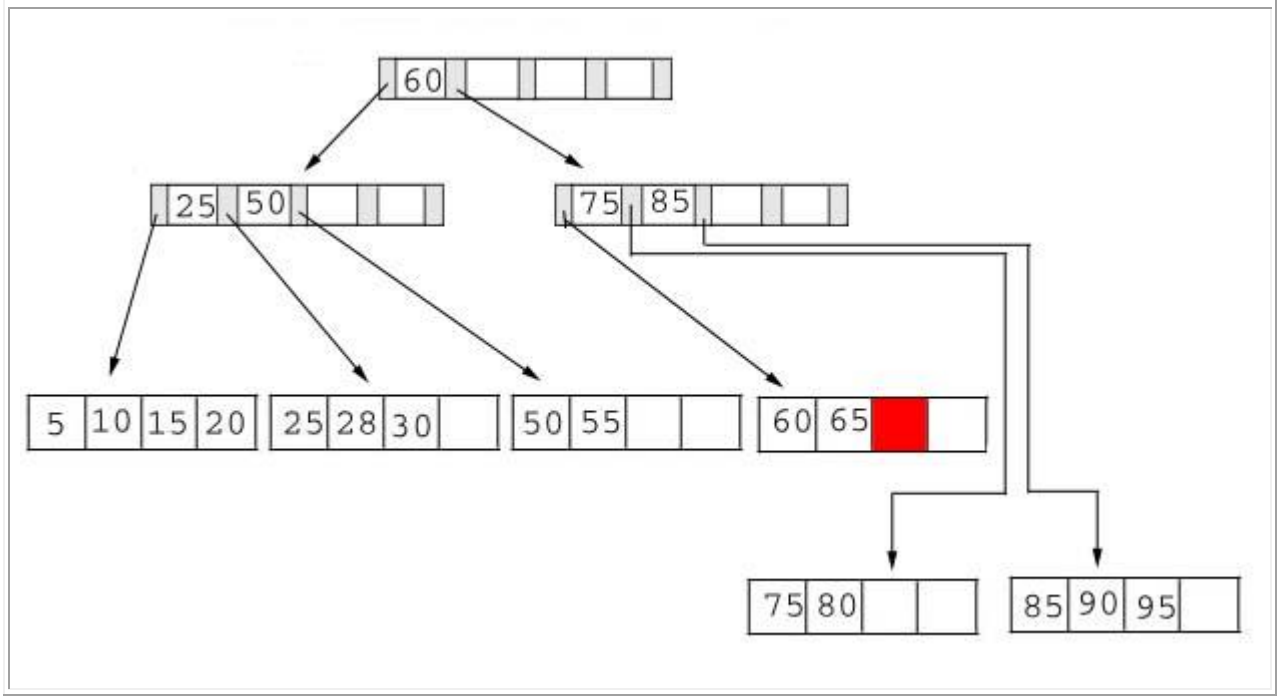
Add Record with Key 95



Delete 70 from the B+ Tree

We begin by deleting the record with key 70 from the B+ tree. This record is in a leaf page containing 60, 65 and 70. This page will contain 2 records after the deletion. Since our fill factor is 50% or (2 records) we simply delete 70 from the leaf node. The following table shows the B+ tree after the deletion.

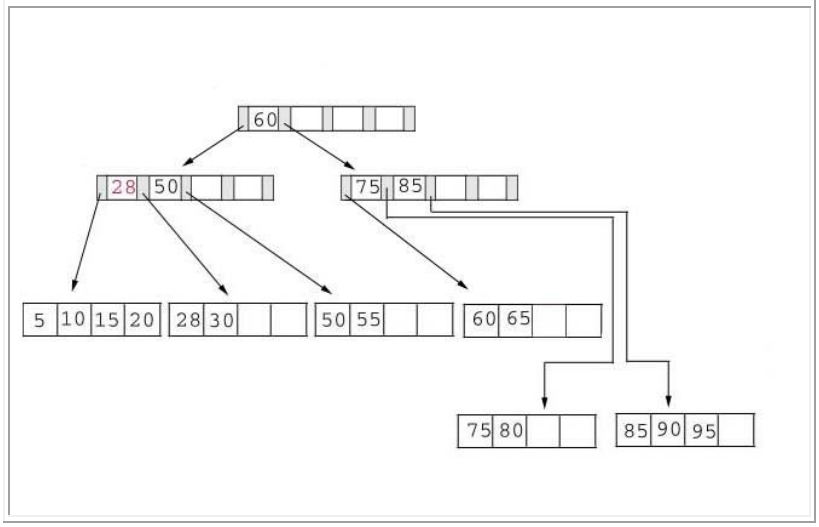
Delete Record with Key 70



Delete 25 from the B+ tree

Next, we delete the record containing 25 from the B+ tree. This record is found in the leaf node containing 25, 28, and 30. The fill factor will be 50% after the deletion; however, 25 appears in the index page. Thus, when we delete 25 we must replace it with 28 in the index page. The following table shows the B+ tree after this deletion.

Delete Record with Key 25



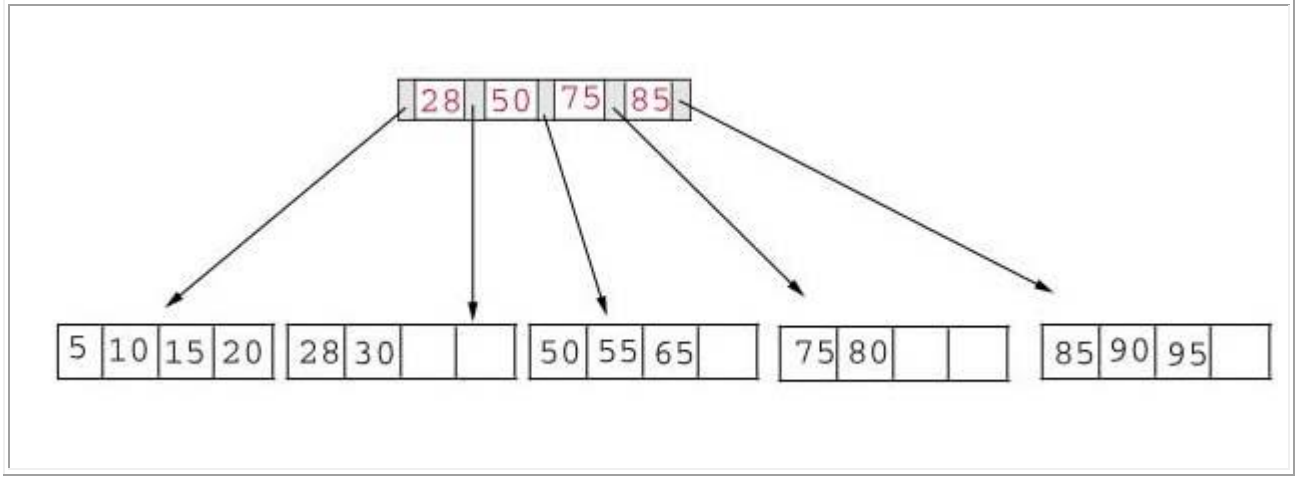
Delete 60 from the B+ tree

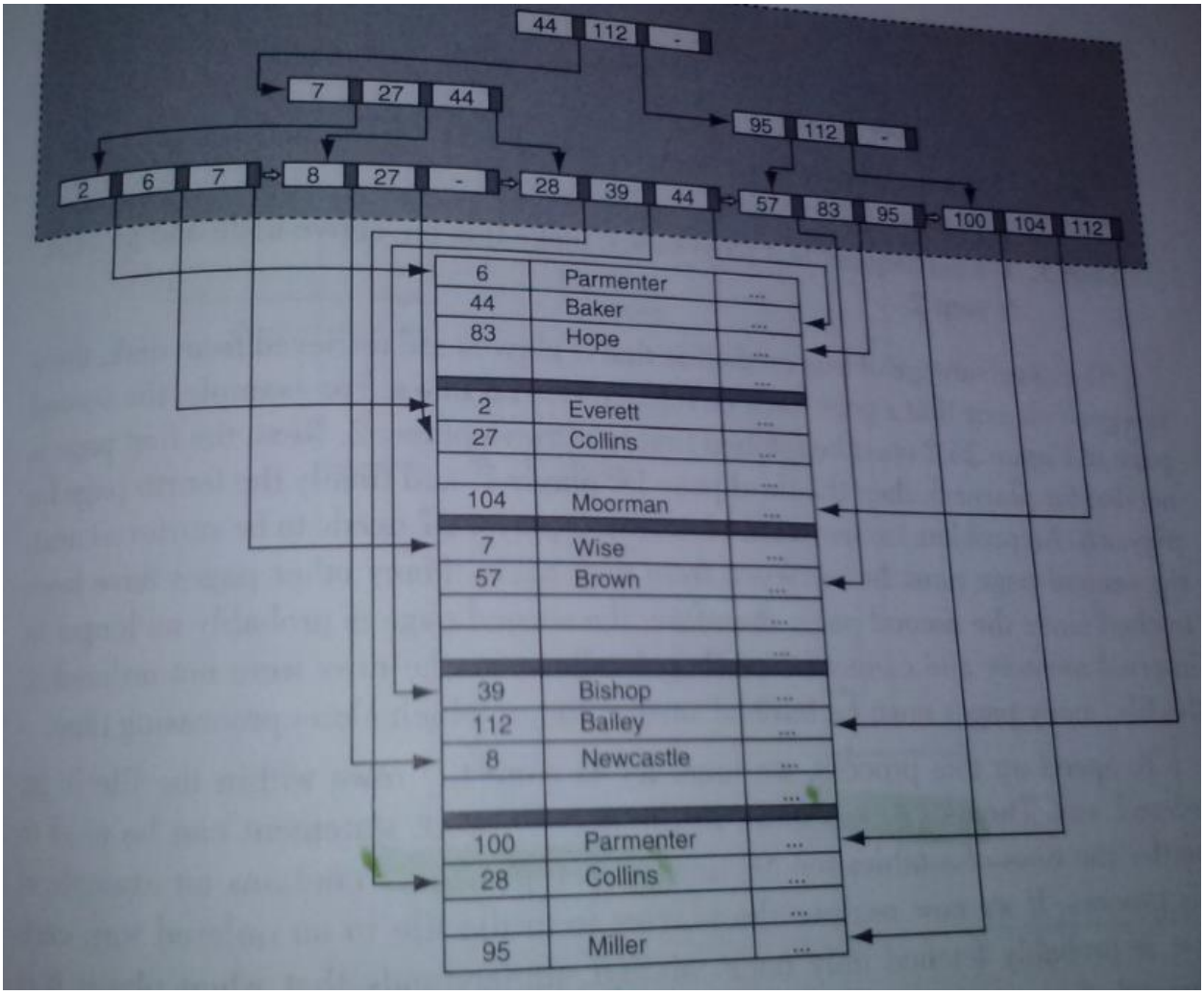
In this example, we're going to delete 60 from the B+ tree. This deletion is interesting for several reasons:

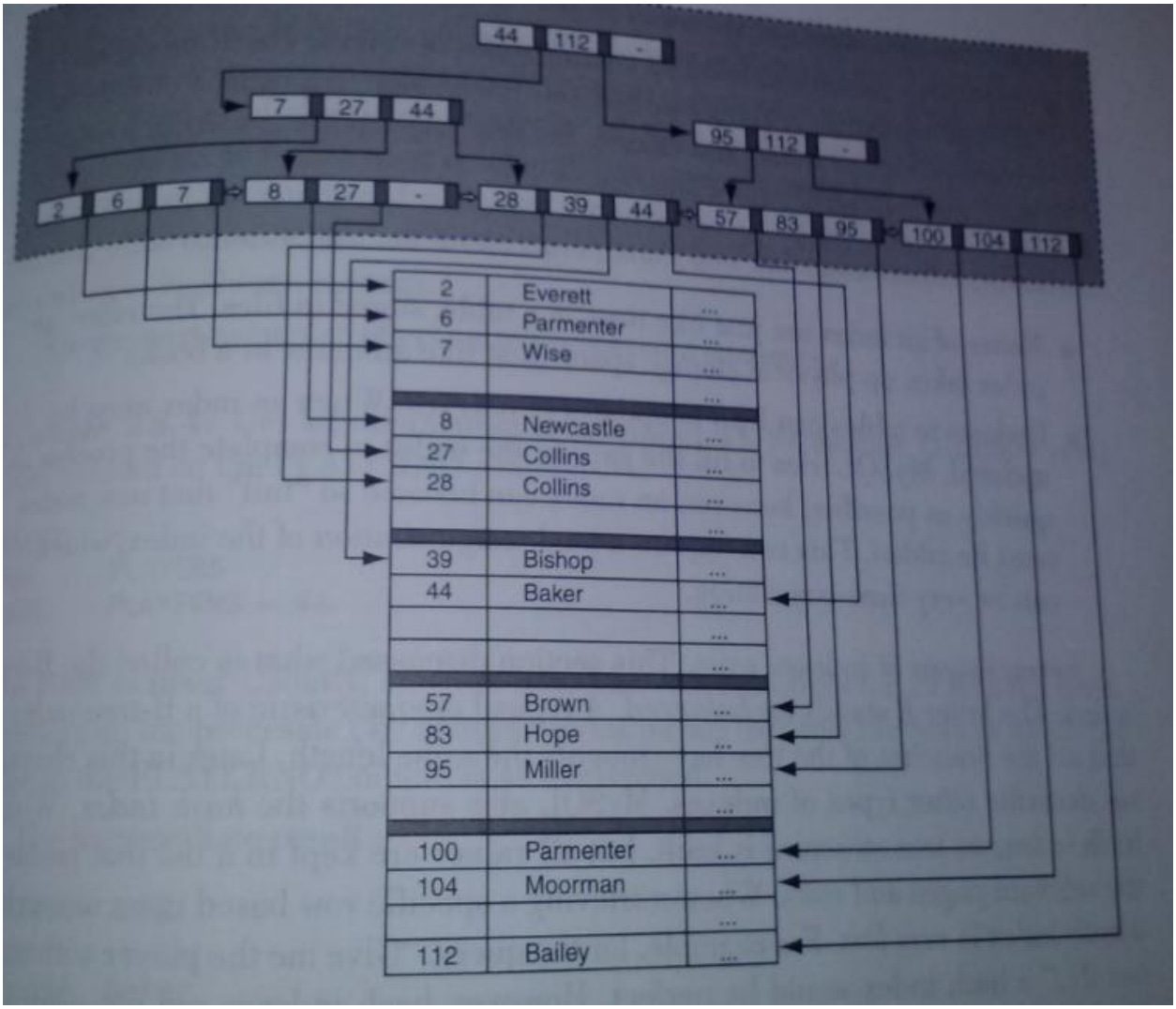
1. The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.
2. With recombined pages, the index page will be reduced by one key. Hence, it will also fall below the fill factor. Thus, we must combine index pages.
3. Sixty appears as the only key in the root index page. Obviously, it will be removed with the deletion.

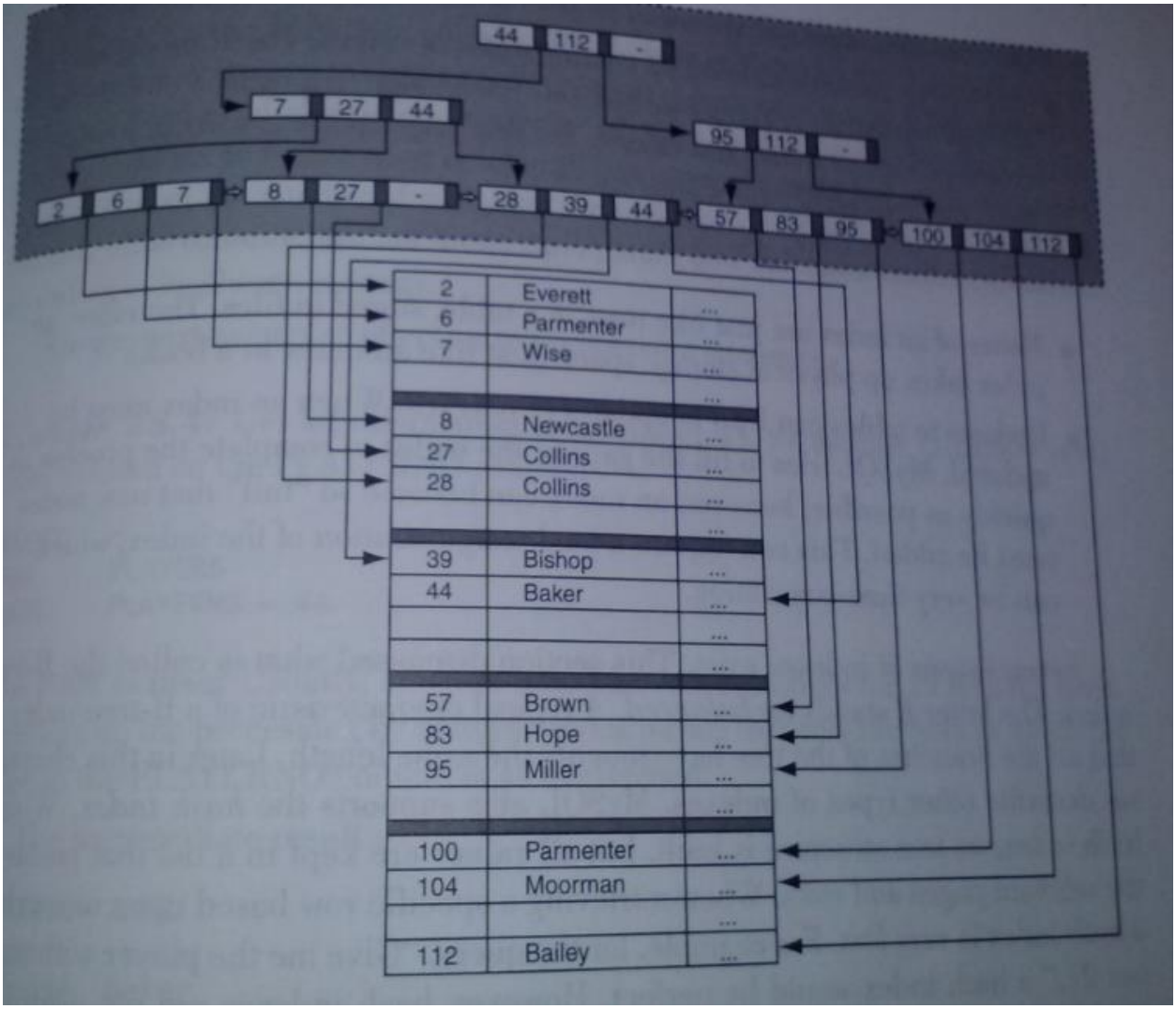
The following table shows the B+ tree after the deletion of 60. Notice that the tree contains a single index page.

Delete Record with Key 60




44 112 -
7 27 44
95 112 -
2 6 7
8 27 -
28 39 44
57 83 95
100 104 112
6 Parmenter
44 Baker
83 Hope
2 Everett
27 Collins
104 Moorman
7 Wise
57 Brown
39 Bishop
112 Bailey
8 Newcastle
100 Parmenter
28 Collins
95 Miller


44 112 -
7 27 44
95 112 -
2 6 7
8 27 -
28 39 44
57 83 95
100 104 112
2 Everett
6 Parmenter
7 Wise
8 Newcastle
27 Collins
28 Collins
39 Bishop
44 Baker
57 Brown
83 Hope
95 Miller
100 Parmenter
104 Moorman
112 Bailey


44 112 -
7 27 44
95 112 -
2 6 7
8 27 -
28 39 44
57 83 95
100 104 112
2 Everett
6 Parmenter
7 Wise
8 Newcastle
27 Collins
28 Collins
39 Bishop
44 Baker
57 Brown
83 Hope
95 Miller
100 Parmenter
104 Moorman
112 Bailey