CS3402 : Chapter 9
Indexing Techniques

Indexes as Access Paths

- An index is an auxiliary file to provides fast access to a record in a data file (index file + data file)
- Without an index, we may use the sequential search (for unordered file), i.e., to examine the record one by one to find the required record stored in disk (from beginning to end of the file until the required record is found)
- An index is an ordered sequence of entries <field value, pointer to record>, ordered by the field value
- The pointer provides the address (location or physical block) of the required record (in the data file) in disk storage
- An index is usually specified on one field (called key field which may not be a key) of the data file (although it could be specified on several fields)

Indexes as Access Paths

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
  - The biggest delay in data retrieval is the disk retrieval time
  - The size of each index entry is much smaller than the size of a record
  - Reading the index from disk is faster than reading data/records file from disk (smaller number of disk blocks)
  - Searching data/records in the main memory is very fast << the disk access

- Instead of searching the data file sequentially, we search the index to get the address of the required records
- A binary search on the index yields (ordered) a pointer to the file record

Types of Single Level Indexes

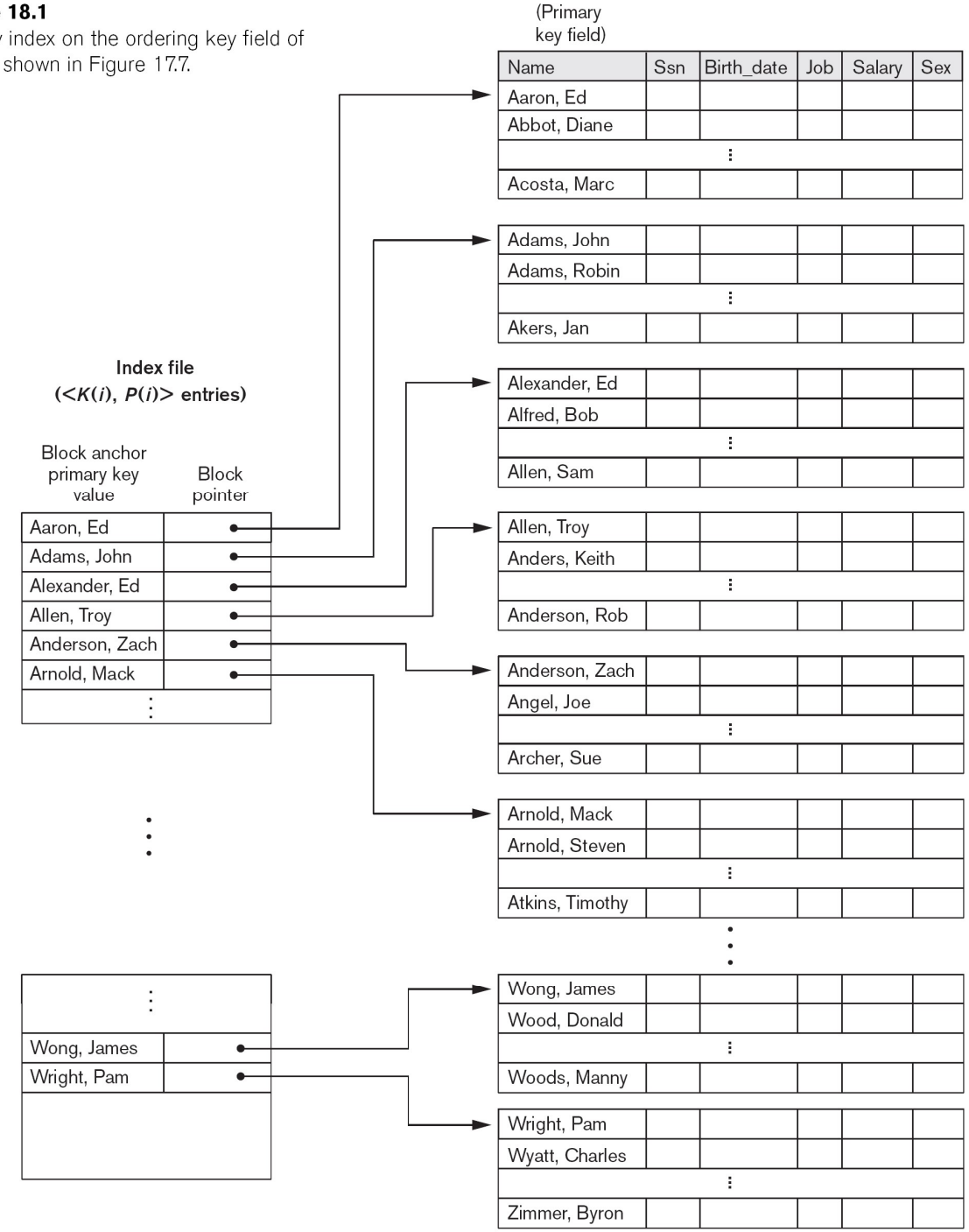
- Single level index: one index points to one data record
- Primary index (key field)
  - Specified on the ordering key field of an ordered file of records
- Cluster index (non-key field)
  - The ordering field of the index is not a key field and numerous records in the data file can have the same value for the ordering field (repeating value attributes)
- Secondary index (non-ordering field)
  - The index field is specified on any non-ordering field of a data file
- Dense or sparse indexes
  - A dense index has an index entry for every search key value (and hence every record) in the data file. Thus larger index size
  - A sparse index, on the other hand, has index entries for only some of the search values. Thus smaller index size

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file (of a specific key)
    - The data file is physically ordered on a key field (non-repeating value attribute)
  - One index entry *for each block* in the data file
    - A primary index is a sparse index (small index)
    - The number of entries is much smaller than the number of records
    - The index entry has the key field value for the *first record* in the block, which is called the *block anchor*
    - Searching subsequent records in the block is easy once the block is in the main memory

Primary Index on the ~~Ordering Key~~ Field

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



Searching an Index Example

- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- SSN is the ordering key field
  - An ordering key is used to physically order the records on storage
- Suppose that:
  - Record size fixed and unspanned R=100 bytes
  - Block size B=1024 bytes
  - Number of record r=30,000 records
- Then, we get:
  - Blocking factor Bfr= B div R = 1024 div 100= 10 records/block
  - Number of file blocks b= (r/Bfr)= (30000/10)= 3,000 blocks
  - A binary search on the data file need approximate $\log_2 3000 = 12$ block accesses

Searching an Index Example

- For an index on the SSN field, assume the field size V_{SSN}=9 bytes and assume the record pointer size P_R=6 bytes. Then:
  - An index entry size R_I=(V_{SSN}+ P_R)=(9+6)=15 bytes
  - Index blocking factor Bfr_I= B div R_I= 1024 div 15= 68 entries/block
  - The total number of index entries = number of blocks = 3000
  - Number of index blocks b= (r/ Bfr_I)= (3000/68)= 45 blocks
  - Binary search needs $\log_2$bI= $\log_2$45= 6 block accesses
  - To get the required record = 6 + 1 block accesses

Note: Once a block is stored in the main memory, the searching cost of the data/records within the block can be ignored (very fast)

Types of Single-Level Indexes

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field (repeating values)*
  - One index entry for each distinct value of the field
  - A clustering index is also an ordered file with two fields
    - <clustering field, disk block pointer>
  - The pointer points to the first block in the data file that has a record with that value for its clustering field
  - It is another example of sparse index
    - One index entry points to multiple records (ordered)
  - Ordered data file => overflow problem
    - Use overflow pointers or reservation

A Clustering Index Example

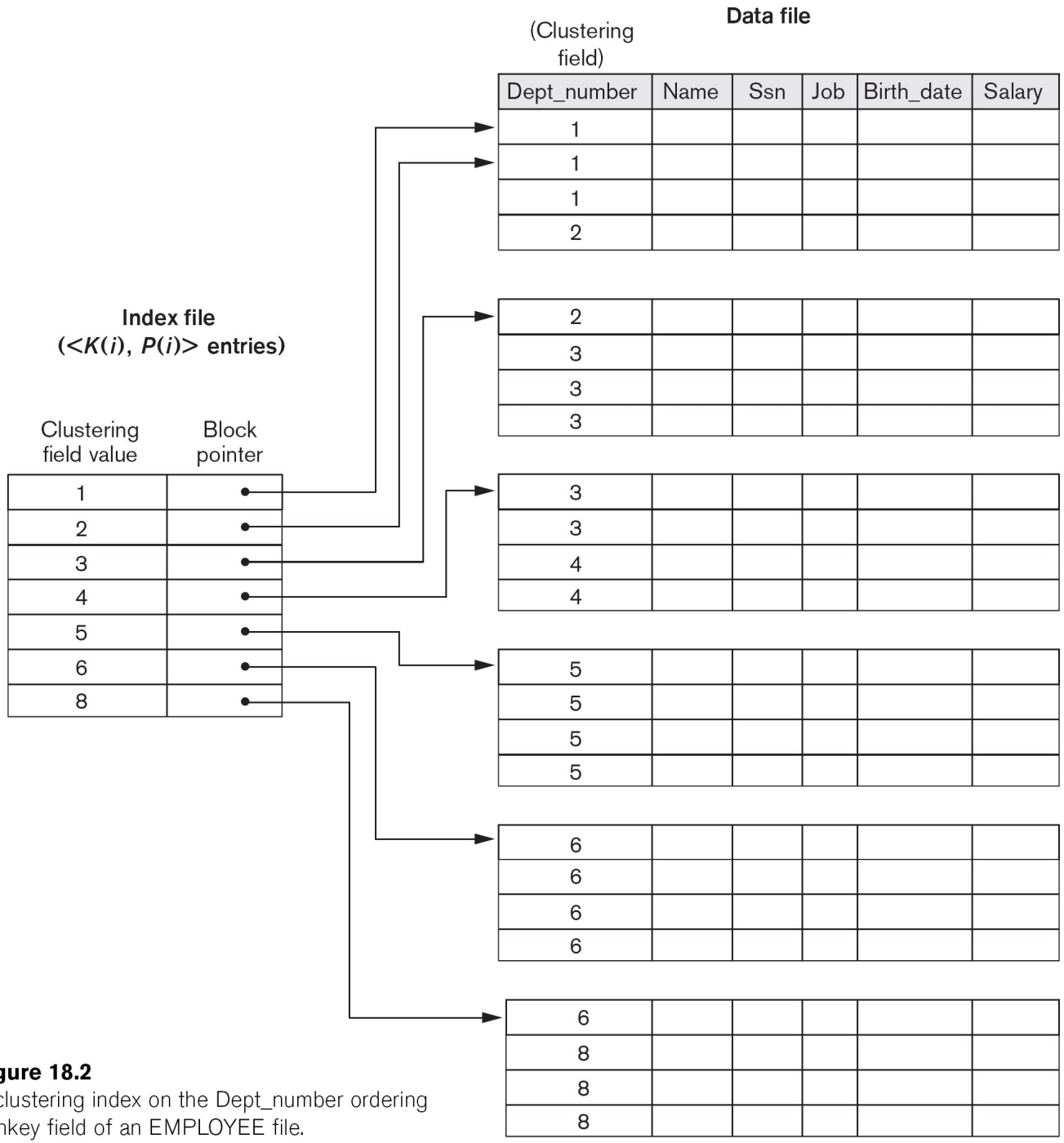
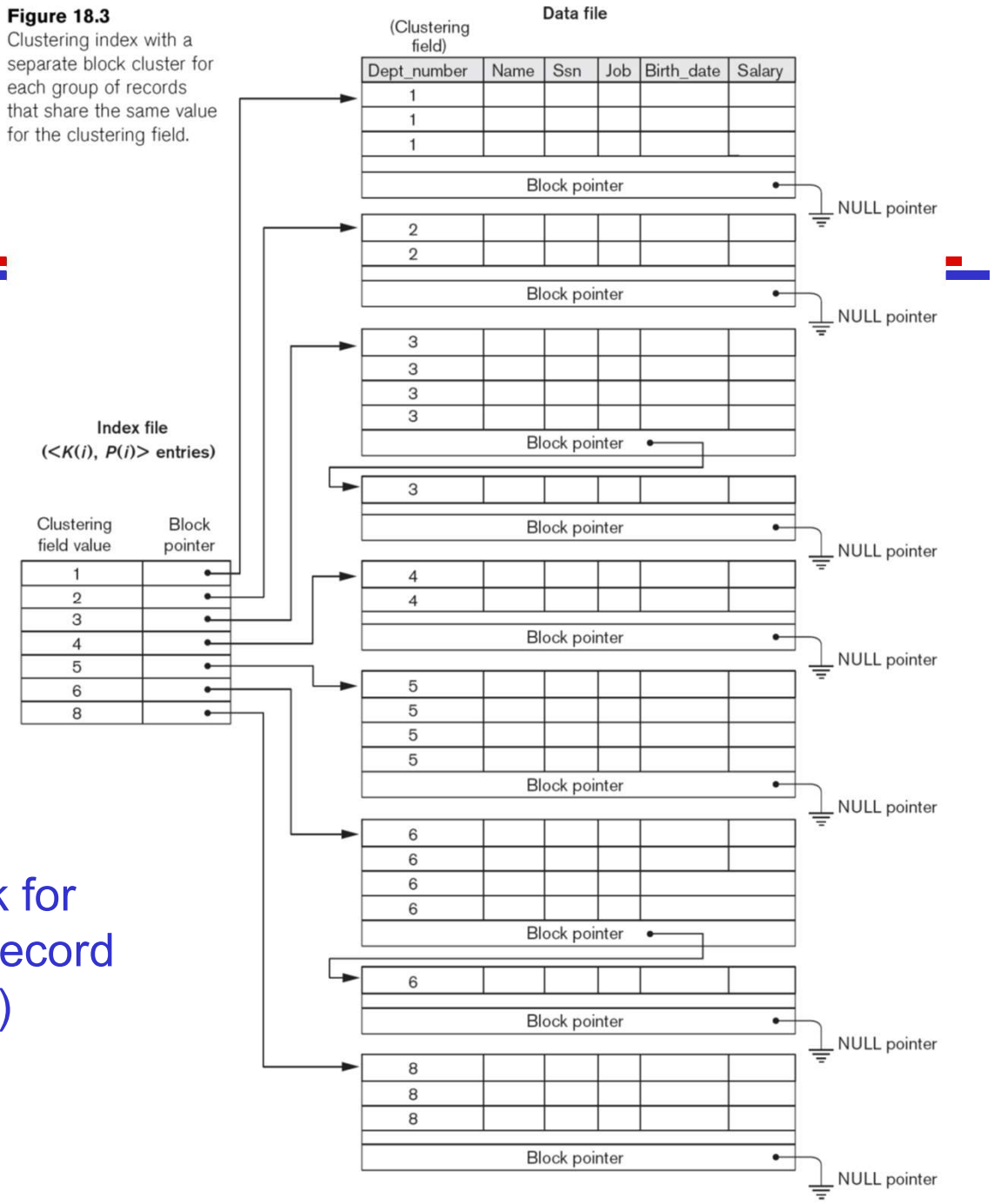


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Another Clustering Index Example



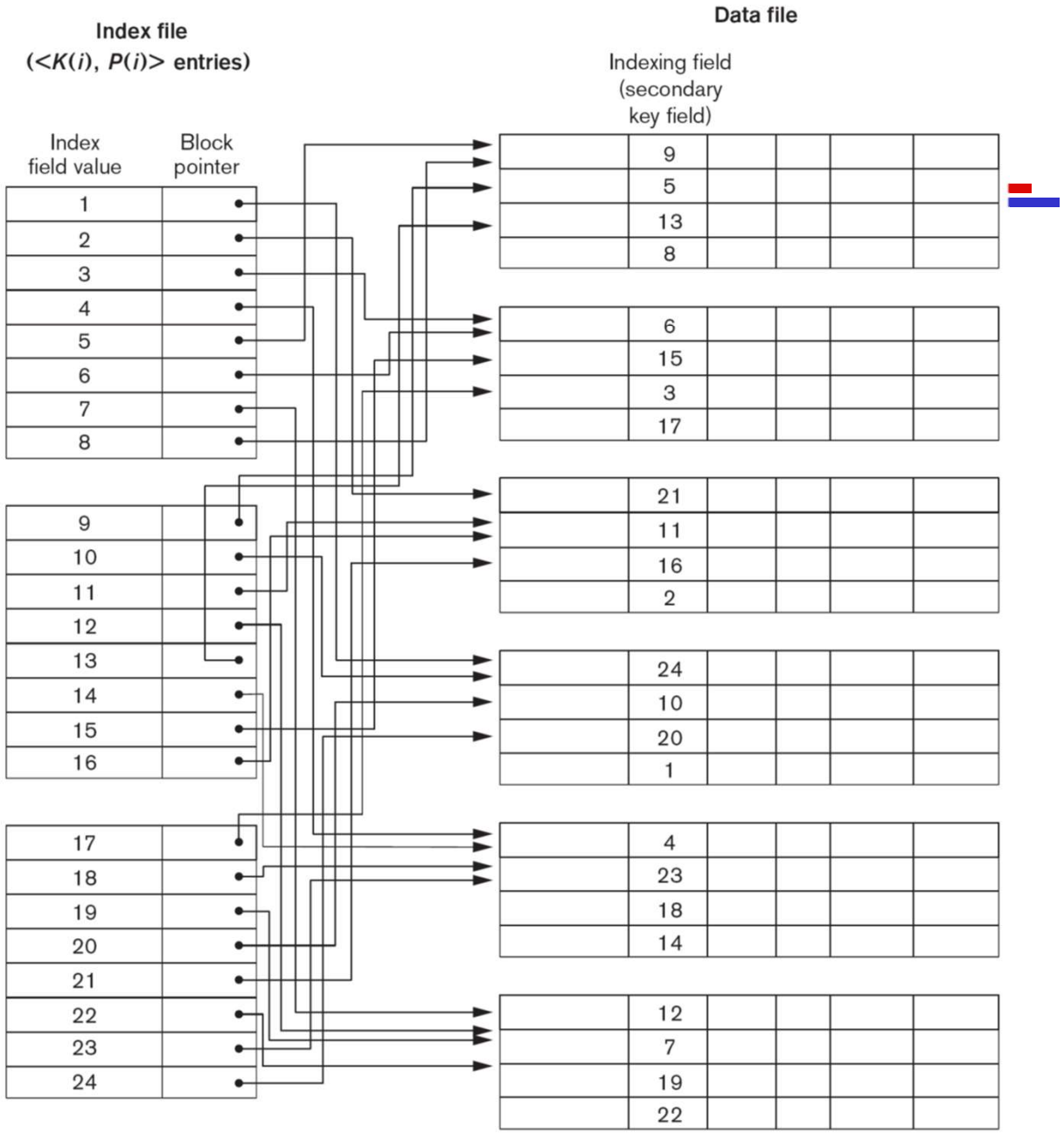
Reserving the whole block for handling overflow due to record insertion (anchoring block)

Types of Single-Level Indexes

- 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 key field or a non-key field
  - The index is an ordered file with two fields
    - An indexing field + a block pointer or a record pointer
    - There can be *several* secondary indexes (and hence, indexing fields) for the same file
  - Block pointers => sparse index
    - Each index entry has multiple index pointers pointing to the records with the same value
  - One entry for each record in the data file => *dense index*
    - Each record has one pointer

Example of a Dense Secondary Index

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



Example of a Secondary Index

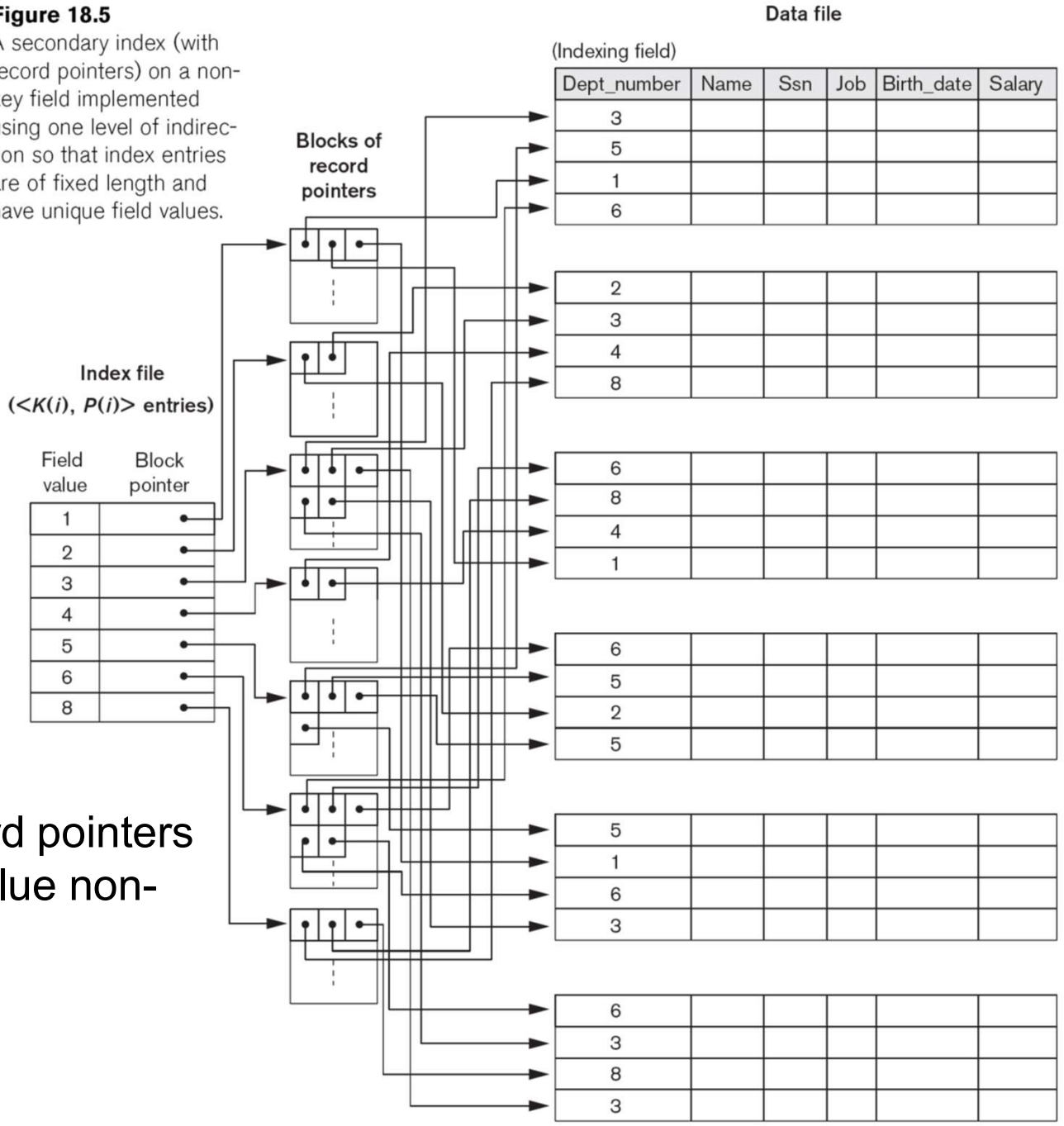


Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Using blocks of record pointers for handling same value non-key records

Secondary Index Example

- Suppose:
  - Record size fixed and unspanned R=100 bytes
  - Block size B=1024 bytes
  - Number of record r=30,000 records

- Then, we get:
  - Blocking factor Bfr= B div R = 1024 div 100= 10 records/block
  - Number of file blocks b= (r/Bfr)= (30000/10)= 3,000 blocks
  - We want to search for a record with a specific value for the secondary key (length 9 bytes)
  - Without the secondary index, a linear search on the data file requires on average b/2 = 3000/2 = 1,500 block accesses on average

Secondary Index Example

- Suppose we construct a secondary index on a non-ordering key field of the data file
  - An index entry size $R_I=(V+ P_R)=(9+6)=15$ bytes
  - Index blocking factor $Bfr_I= B \text{ div } R_I= 1024 \text{ div } 15= 68$ entries/block
  - In a dense index, the total number of index entries = number of records = 30000
  - Number of index blocks $b= (r/ Bfr_I)= (30000/68)= 442$ blocks
  - Binary search needs $\log_2 b= \log_2 442= 9$ block accesses
  - To get the required record = 9 + 1 block accesses
  - This is compared to an average linear search cost of:
    - (b/2) = 3000/2= 1500 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\text{Log}_2 b = \log_2 30{,}000=$ 12 block accesses

Properties of Index Types

Table 18.1 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 18.2 Properties of Index Types

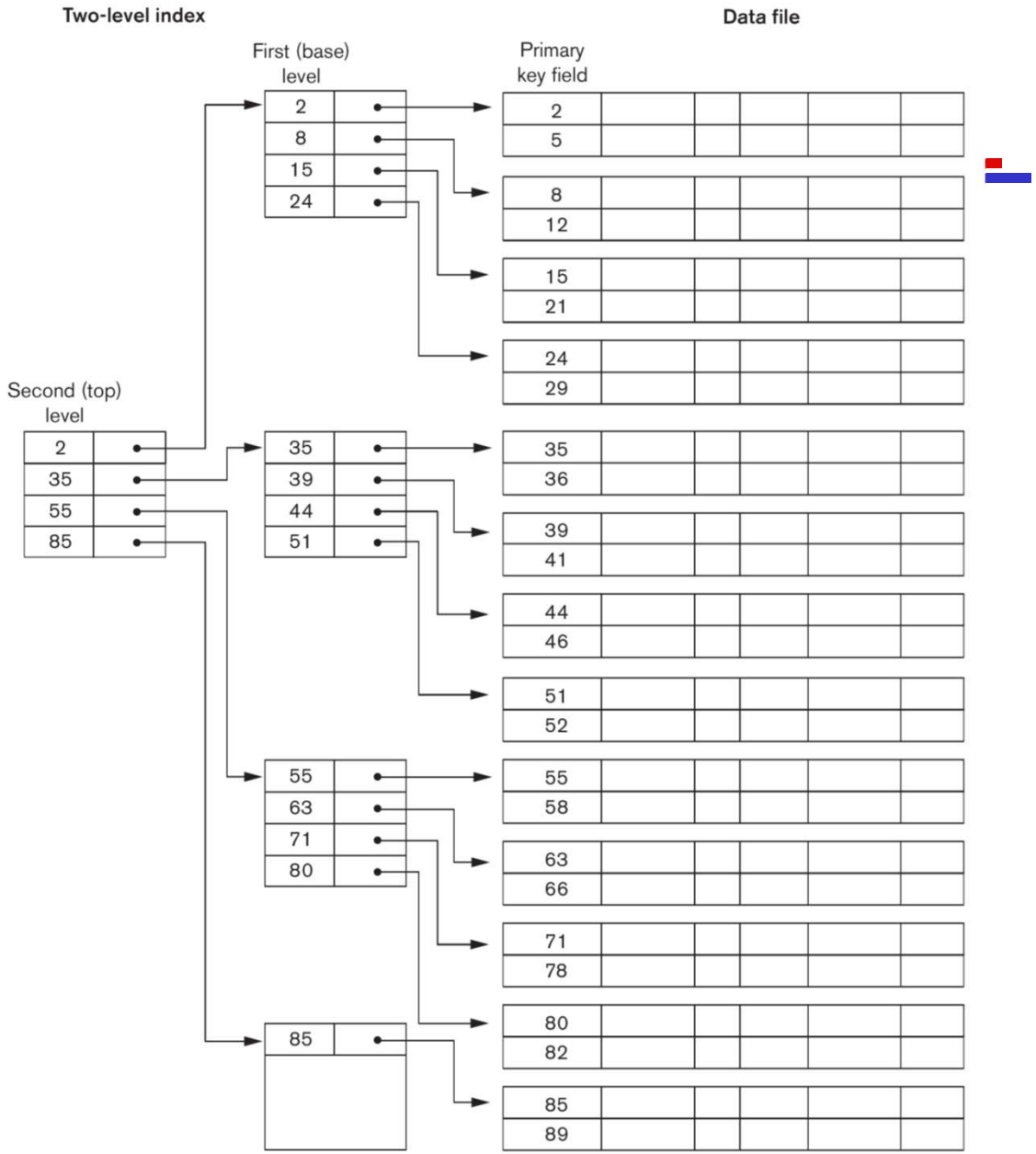
Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)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

A Two-Level Primary Index

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



Multi-Level Indexes Example

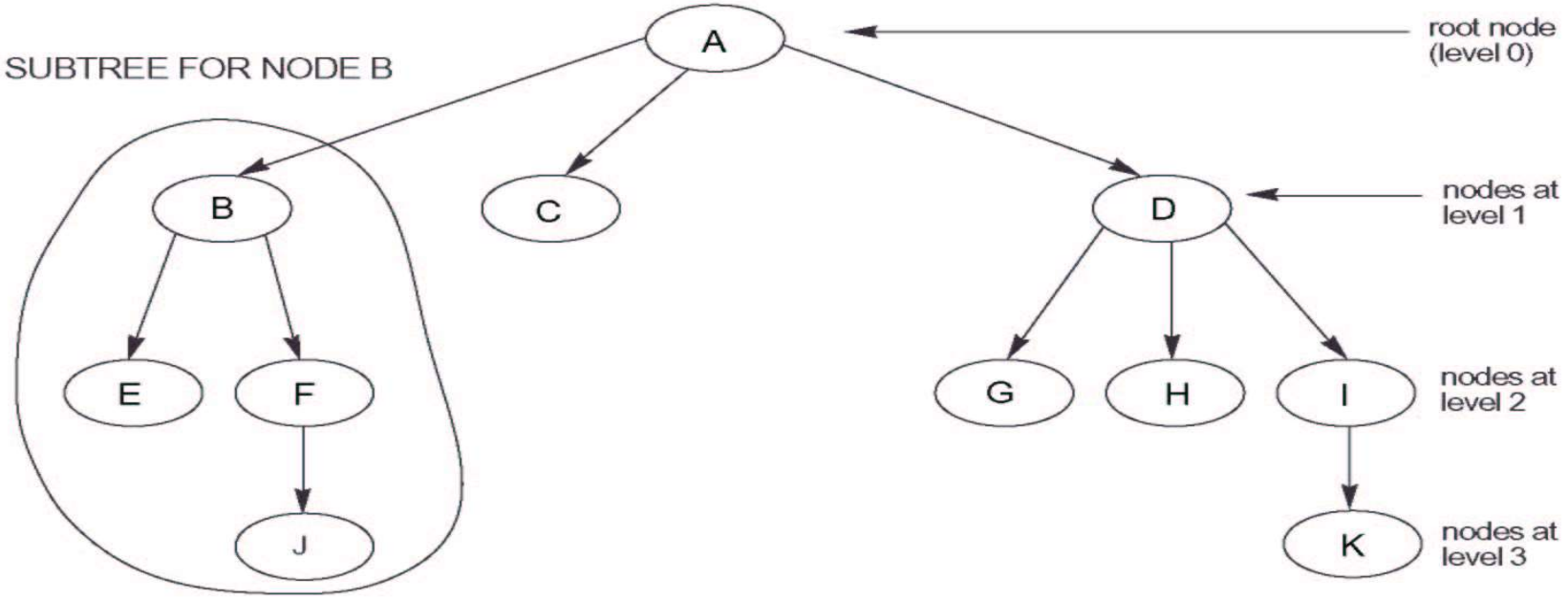
- Suppose the dense secondary index of Example 2 is converted into a multi-level index
  - The index blocking factor (also called fan out, fo) bfr_i = 68 index entries per block
  - The number of first level blocks b1 =442
  - The no. of second-level blocks b2 = b1/fo = 442/68 = 7 blocks
  - The no. of third-level blocks b3 = b2/fo = 7/68 = 1
  - Hence the third-level is the top level of the index t =3
  - To access a record, access one block at each level plus one block from the data file = t + 1 = 3 + 1 = 4 block accesses

Tree Index Structure

- A multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe performance problem because every level of the index is an *ordered file*
- We may use a B-/B$^+$-tree index to resolve the problem (reservation of nodes)
- A tree is formed of multi-level nodes
- Except the root node, each node in the tree has one parent node and zero or more child node
- A node that does not have any child nodes is called a leaf node
- A non-leaf node is called internal node
- A sub-tree of a node consists of the node and all its descendant
- If the leaf nodes are at different levels, the tree is called unbalanced

Multi-Level Indexes

Figure 6.7 A tree data structure that shows an unbalanced tree.



(nodes E,J,C,G,H, and K are leaf nodes of the tree)

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

- Most multi-level indexes use B-tree or B^+-tree data structures for solving the insertion and deletion problem
  - They reserves spaces in each tree node (disk block) to allow for new index entries
- In B-Tree and B^+-Tree, each node corresponds to a disk block
- Each node is kept between half-full and completely full
  - Need to restructure the tree in case of node overflow (insertion when full) or underflow (deletion when half full)
  - Less than half-full, wastes of space and the tree will have many levels (higher retrieval cost)

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

- 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 higher 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 may merge with neighboring nodes (may propagate to higher levels)

Difference between B-tree and B+-tree

- 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 since its entry is smaller in size
- The internal nodes of B+-tree contains pointers only

B-tree Index

- B-tree organizes its nodes into a tree
- It is balanced (B) as all paths from the root to a leaf node have the same length

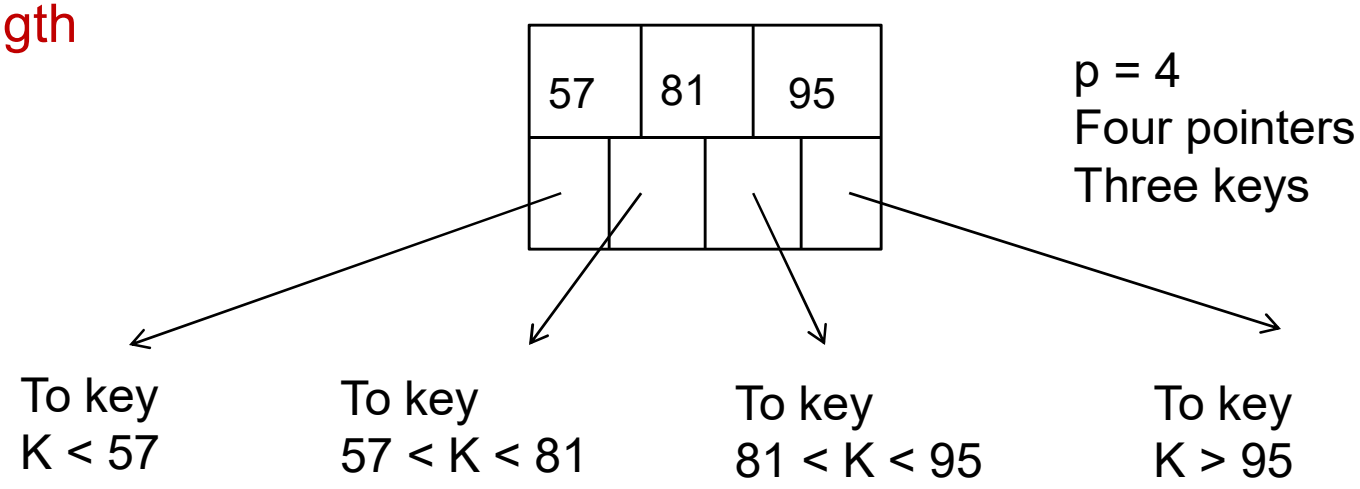
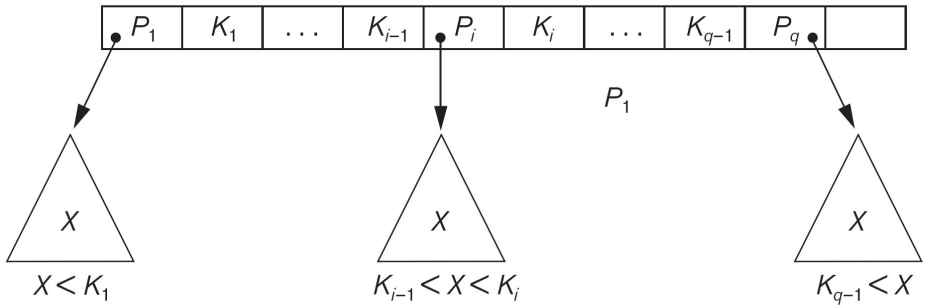


Figure 18.8
A node in a search tree with pointers to subtrees below it.



B-tree Index

- Each internal node in a B-tree is of the form $<P_1, <K_1,Pr_1>, P_2, <K_2,Pr_2>, \ldots, <K_{q-1},Pr_{q-1}>,P_q>$ where $q <= p$ (each node has at most p tree pointers)
- Each P_i is a tree pointer to another node in the B-tree
- Each Pr_i is a data pointer points to the record whose search key field value is equal to K_i
- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
- For all search key field values X in the subtree pointed 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 $i = q$
- Each node has at most p tree pointers (order of the tree)
- Each node except the root and leaf nodes, has at least p/2 tree pointers
- All leaf nodes are at the same level (balanced tree) and have the same structure as internal nodes except that all of their tree pointers of P_i are NULL

B-tree Structures

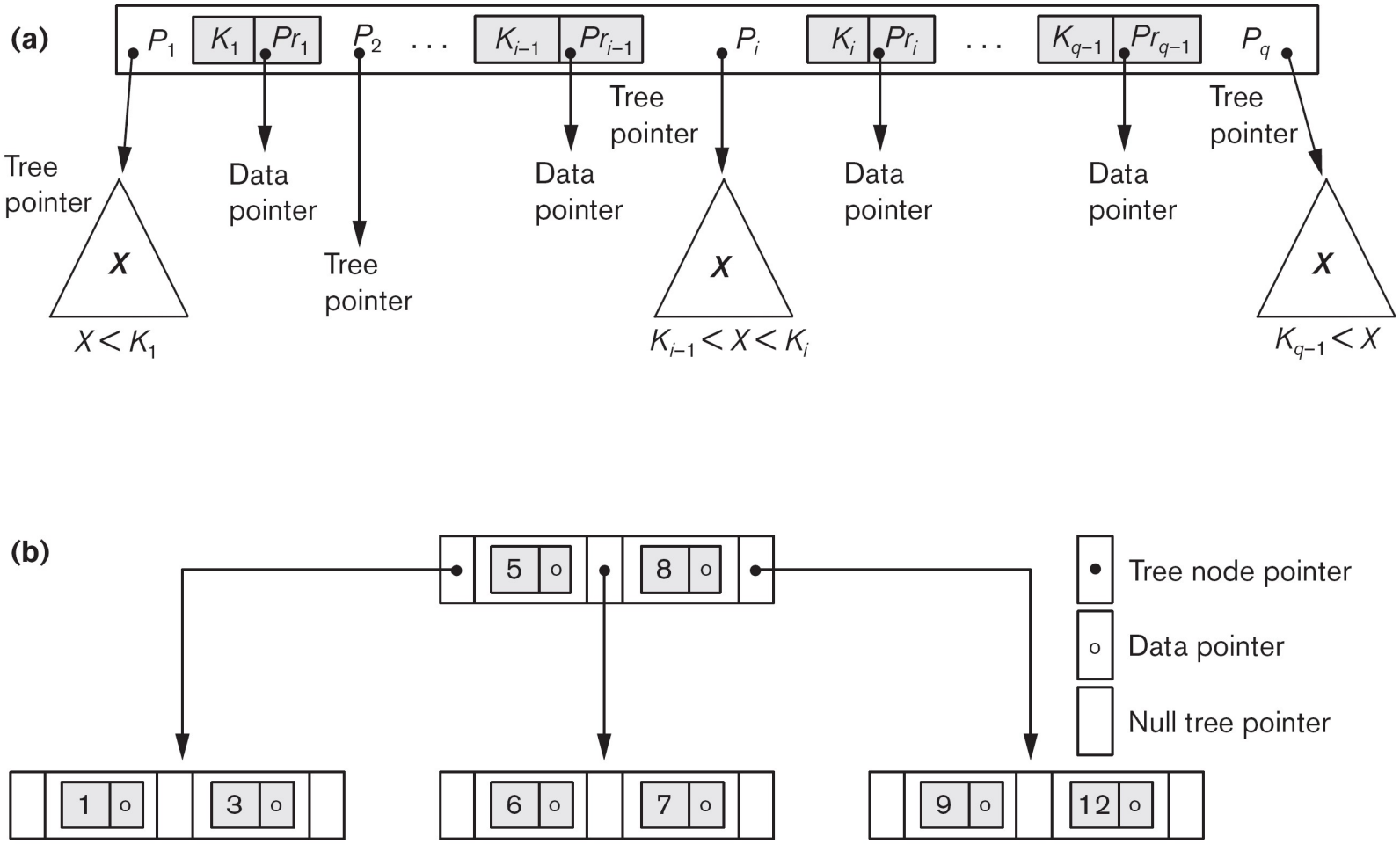


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

B-tree Index Insertion

- Each B-tree node can have at most q tree pointers, p – 1 data pointers, and search key values
- Insertion
  - B-tree starts with a single root node at level 0
  - Once the root is full with p – 1 search key values and we attempt to insert another entry into the tree, the root node splits into two nodes at level 1
  - Only the middle value is kept in the root node and the rest of the values are split evenly between the other two nodes
  - When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes
  - If the parent node is full, it is also split

B-tree Index Capacity

- Suppose the search field is a non-ordering key field and we construct a B-tree on this field with p = 23
- Assume that each node is 69% full, i.e., 0.69 * 23 = 16 pointers (fan-out)
  - Root: 1 node 15 key entries 16 pointers
  - Level 1: 16 nodes 240 key entries 256 pointers
  - Level 2: 256 nodes 3840 key entries 4096 pointers
  - Level 3: 4096 nodes 61,440 key entries
- Key field: all search values K are unique
- Non-key field: the entry pointer may point to a block or a cluster of blocks that contain the pointers to the file records (repeating value)
- What will be the maximum number of key values if the order of the tree is 4 (p) and it has 3 levels?

B^+-tree Index

- In a B^+-tree, data pointers are stored only at the leaf nodes of the tree
- The pointers in internal nodes are tree pointers to blocks that are tree nodes
- The pointers in leaf nodes are data pointers to the data file records
- The leaf nodes of a B^+-tree are usually linked to provide ordered access on the search field to the records
- Because entries in the internal nodes of a B+-tree does not include data pointers, more entries (tree pointers) can be packed into an internal node and thus fewer levels (higher capacity)

B^+-tree Index

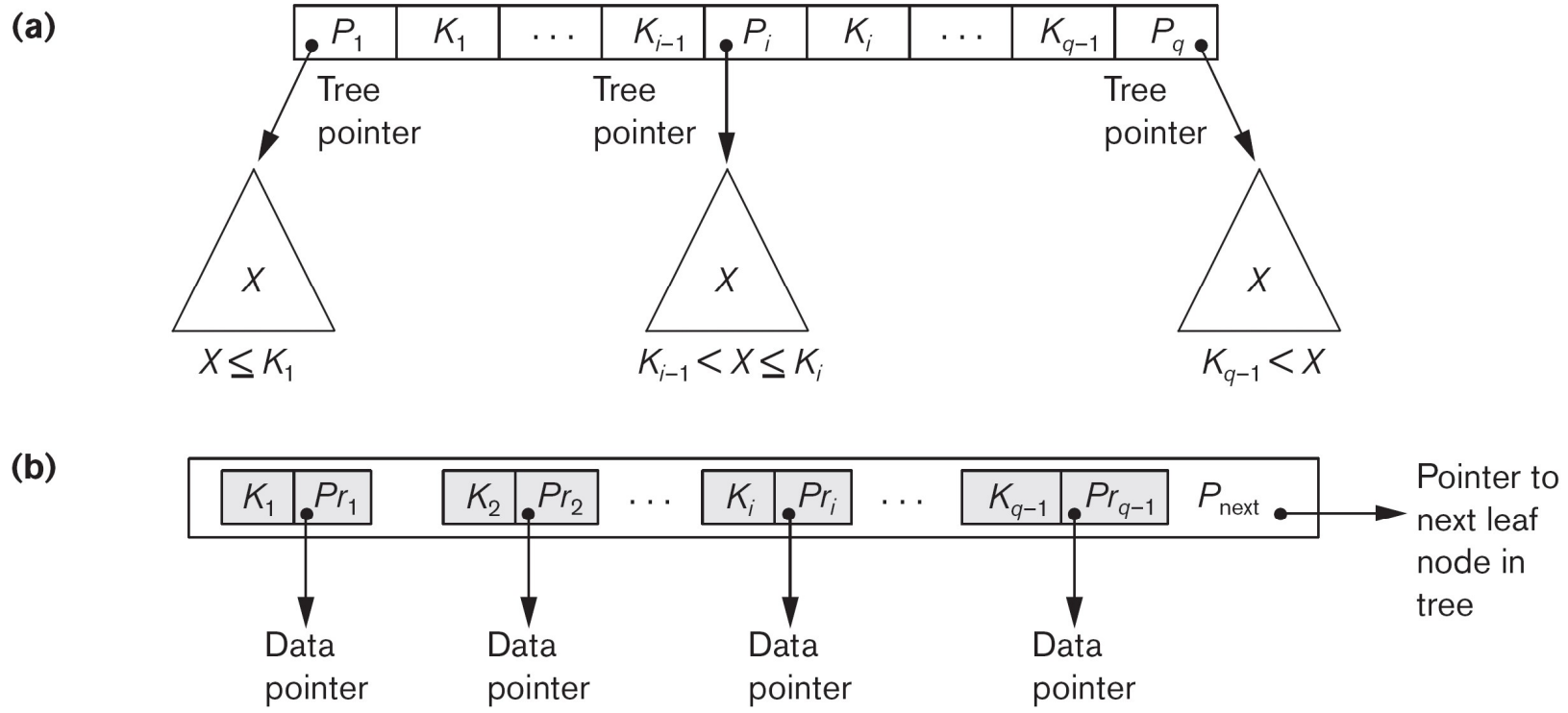
- The structure of internal nodes of a B+-tree
  - Each internal node in a B^+-tree is of the form $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$ where $q <= p$ and P_i is a tree pointer and is also called the tree order
  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  - For all search key field values X in the subtree pointed by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$
  - Each node has at most p tree pointers
  - Each node except the root and leaf nodes has at least p/2 tree pointers

B^+-tree Index

- The structure of the leaf nodes of a B+-tree
  - Each leaf node is of the form
  - $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$ and Pr_i is a data pointer and P_{next} points to the next leaf node of the tree
  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  - The maximum number of data pointers in a leaf node is the tree order minus 1
  - Each Pr_i is a data pointer points to the record whose search field is K_i
  - Each leaf node has at least p/2 values
  - All leaf nodes are at the same level

The Nodes of a B+-tree

Figure 18.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values.
(b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.



B^+-tree Index Example

- Suppose the search key field is V = 9 bytes and the block size is B = 512 bytes, a record pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes
- An internal node of B^+-tree can have up to *p* tree pointers and *p-1* search field value. These must fit into a single block
  - (p * P) + ((p – 1) * V) <= B
  - (p * 6) + (p – 1) * 9) <= 512
  - 15p <= 512
  - p = 34

- The order p_{leaf} for the leaf node
  - (p_{leaf} * (Pr + V)) + P <= B
  - (p_{leaf} * (7 + 9)) + 6 <= 512
  - 16 * p_{leaf} <= 506
  - p_{leaf} = 31

B^+-tree Index Capacity

- To calculate the approximate number of entries in B+-tree, we assume each node is 69% full
- One average each internal node will have 34 * 0.69 = 23 pointers and 22 key values
- Each leaf node on average will hold 0.69 * P_{leaf} = 0.69 * 31 = 21 data record pointers
  - Root: 1 node 22 key entries 23 pointers
  - Level 1: 23 nodes 506 key entries 529 pointers
  - Level 2: 529 nodes 11,638 key entries 12,167 pointers
  - Level 3: 12,167 nodes 255,507 data record pointers

B^+-tree Index Insertion

- To find a place for the new key in the appropriate leaf, we put it there if there is room

- If a leaf node is full and new entry is inserted there, the node overflows
  - We split the leaf into two and divide the keys between the two new nodes so each is half full
  - The first $j = (p_{leaf} + 1)/2$ entries in the original node are kept while the remaining entries are moved to a new leaf node

- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level if there is room insert it. Otherwise split the parent node and continue up the tree

B^+-tree Index Insertion

- The jth search value is replicated in the parent internal node and an extra pointer to the new node is created in the parent
- Note that every key value must exist at the leaf level
- Every value appearing in an internal node also appears as the (rightmost/leftmost) value in the leaf level of the subtree pointed at by the tree pointer to the left of the value
- If we try to insert into the root and there is no room, we split the root into two nodes and create a new root at the next higher level

Example of an Insertion in a B+-tree

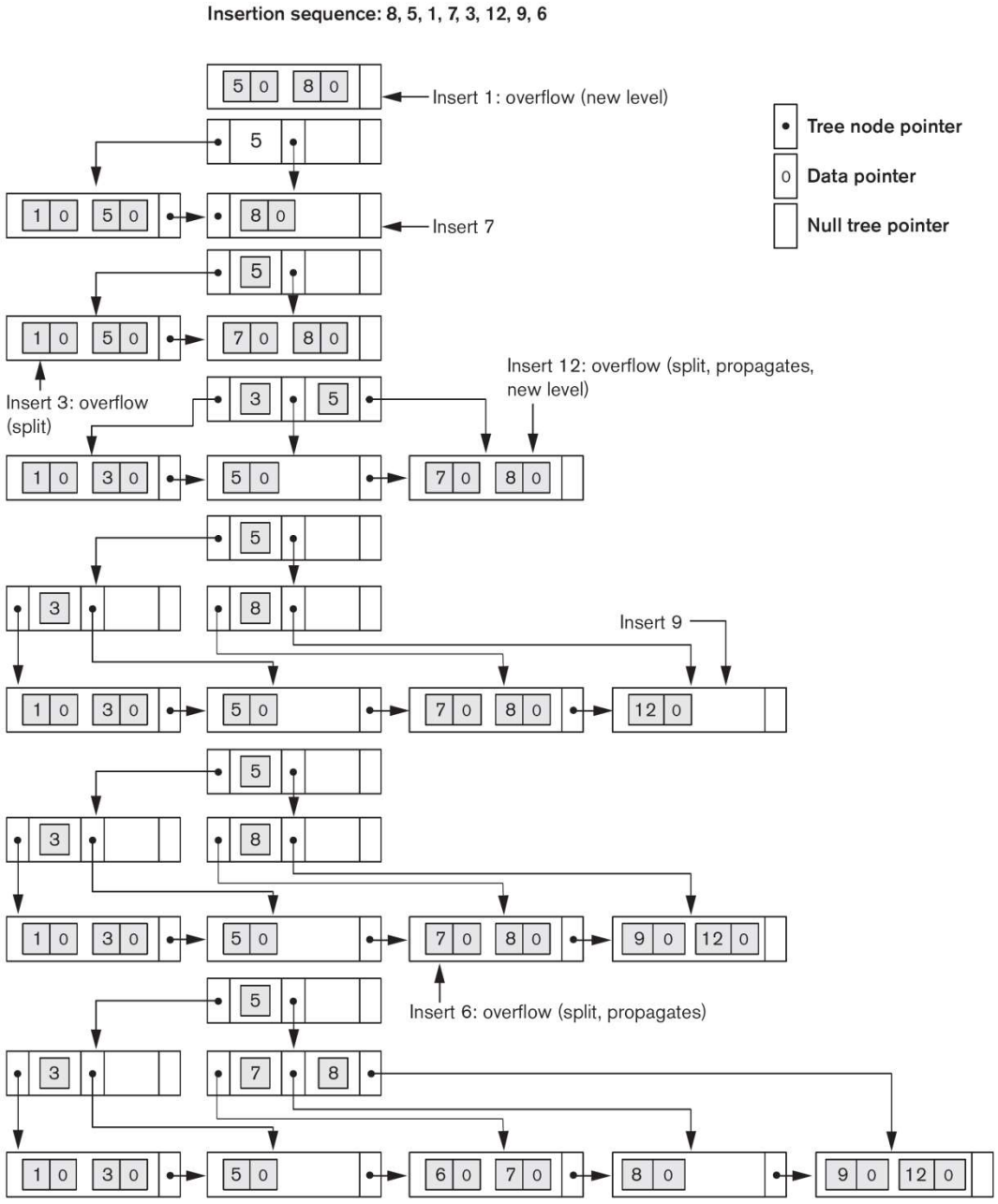
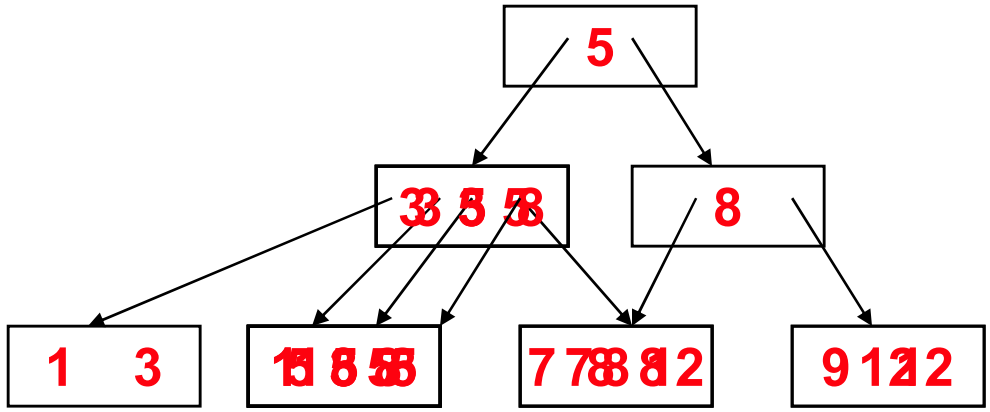


Figure 18.12
An example of insertion in a B+-tree with $p = 3$ and $p_{leaf} = 2$.

Animation

8, 5, 1, 7, 3, 12, 9



B^+-tree Index Deletion

- When an entry is deleted, it is always removed from the leaf node
- If it happens to occur in an internal node, it must also be removed and the value to its left in the leaf node must replace it in the internal node
- Deletion may cause underflow. Then, we try to find a sibling leaf node and redistribute the entries among the node and its sibling so that both are at least half full
- Otherwise, the node is merged with its sibling and the number of leaf node is reduced

Example of a Deletion in a B+-tree

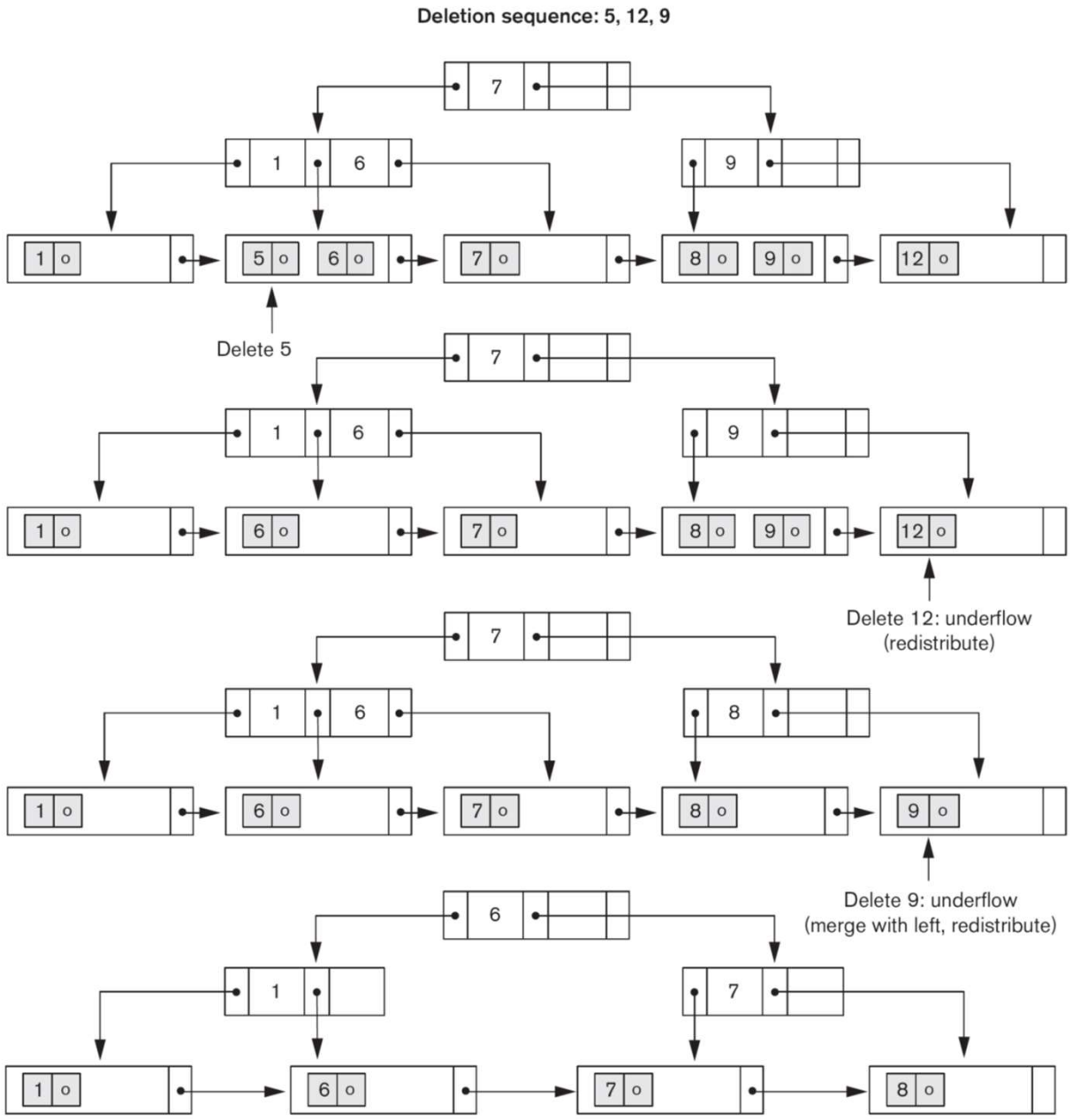


Figure 18.13
An example of deletion from a B+-tree.

References

- 6e
  - *Chapter 16, pages 565-598*
  - *Chapter 17, pages 613-642*