Lecture 8: Indexing Techniques

CS3402 Database Systems

Indexes as Access Paths (1/2)

- 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 (2/2)

- 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 much faster than 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

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

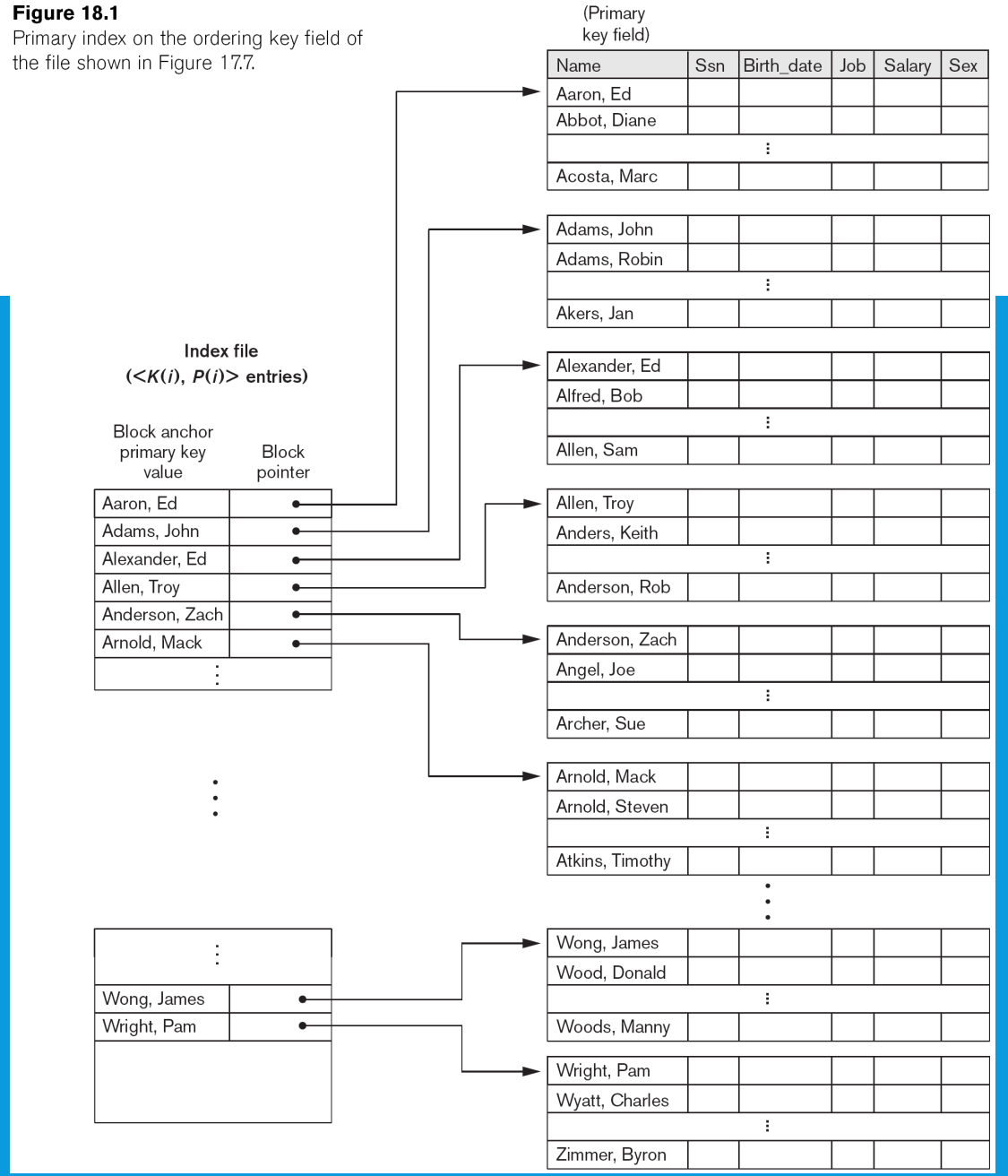


Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.

Primary Index: Example 1

- Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- SSN is the ordering key field that is used to physically order the records on storage
- Suppose that record size fixed and unspanned R = 100 bytes, Block size B = 1,024 bytes, and the number of records r = 30,000 records
- Then, we get:
  - Blocking factor Bfr = B/R = 1,024/100 = 10 records/block
  - Number of file blocks b = r/Bfr = 30,000/10 = 3,000 blocks
  - A binary search on the data file needs approximate $\log_2 3000 = 12$ block accesses

Primary Index: Example 2

- For an index on the SSN field, assume field size $V_{SSN} = 9$ bytes and record pointer size $P_R = 6$ bytes. Then, we get:
  - An index entry size $R_I = V_{SSN} + P_R = 9+6 = 15$ bytes
  - Index blocking factor $Bfr_I = B/R_I = 1,024/15 = 68$ entries/block
  - The total number of index entries = number of blocks = r = 3,000
  - Number of index blocks $b_I = r/Bfr_I = 3,000/68 = 45$ blocks
  - Binary search needs $\log_2 b_I = \log_2 45 = 6$ block accesses
  - In total, 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)

Clustering Index (1/2)

- Defined on an ordered data file that 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

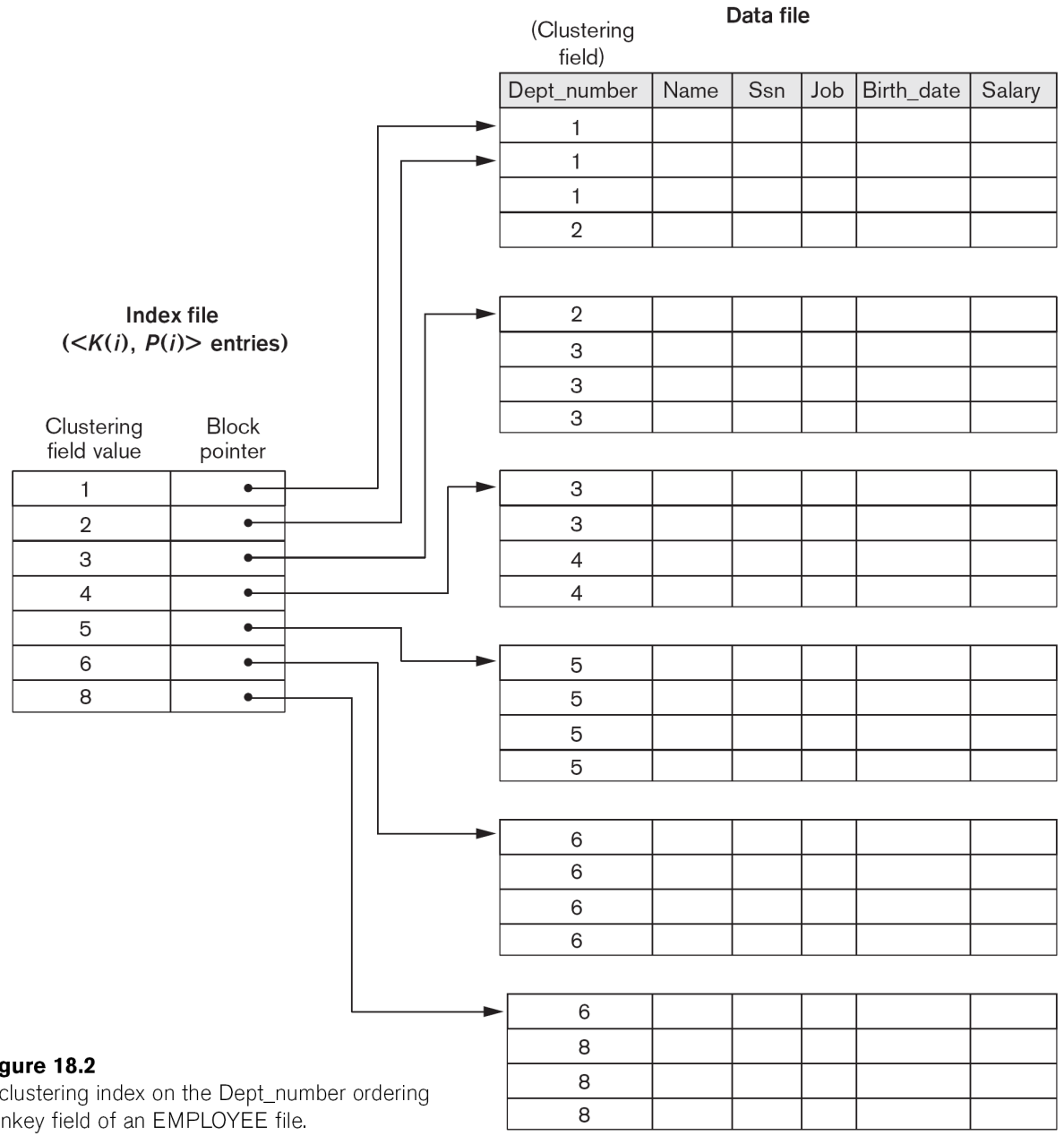
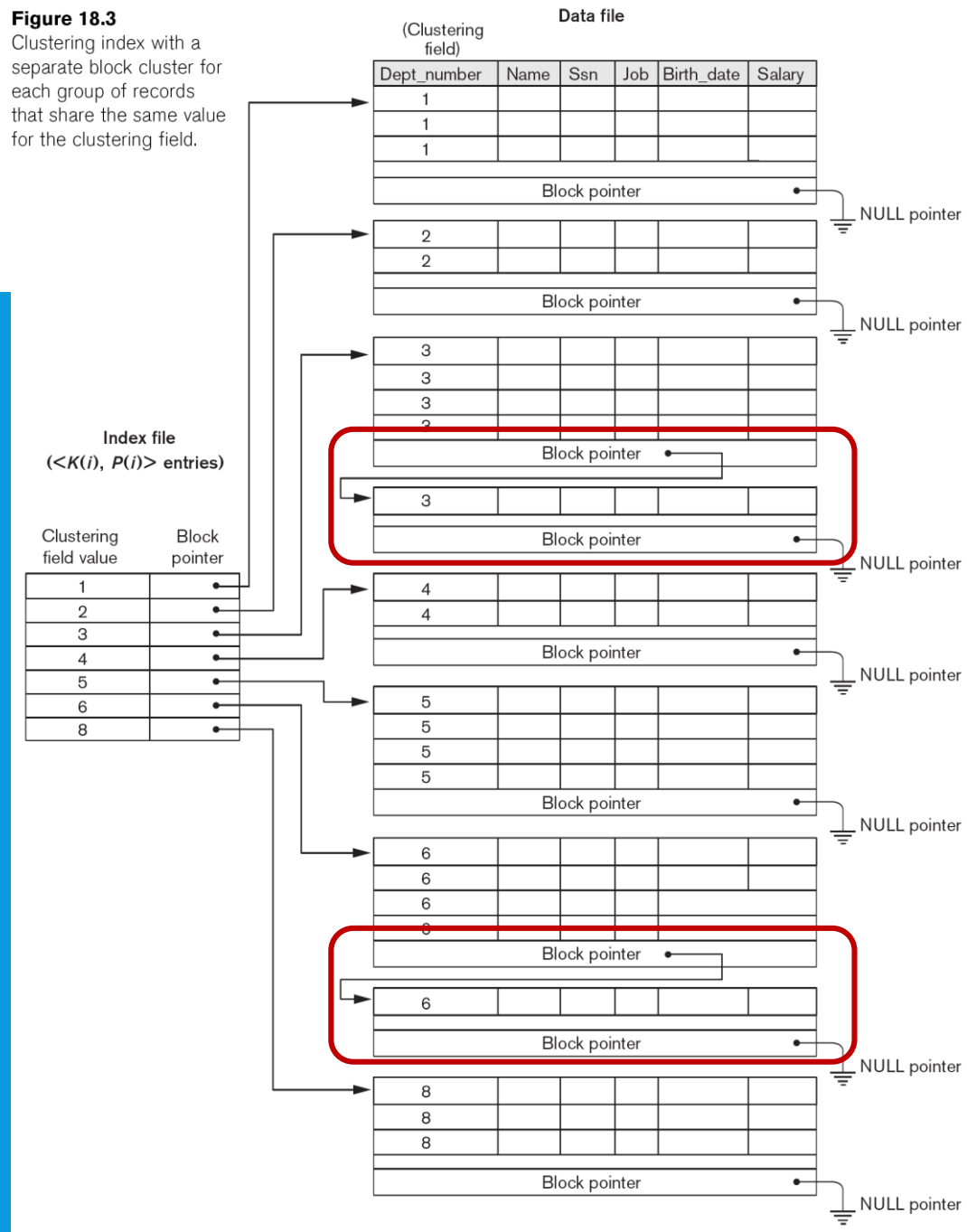


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Clustering Index (2/2)

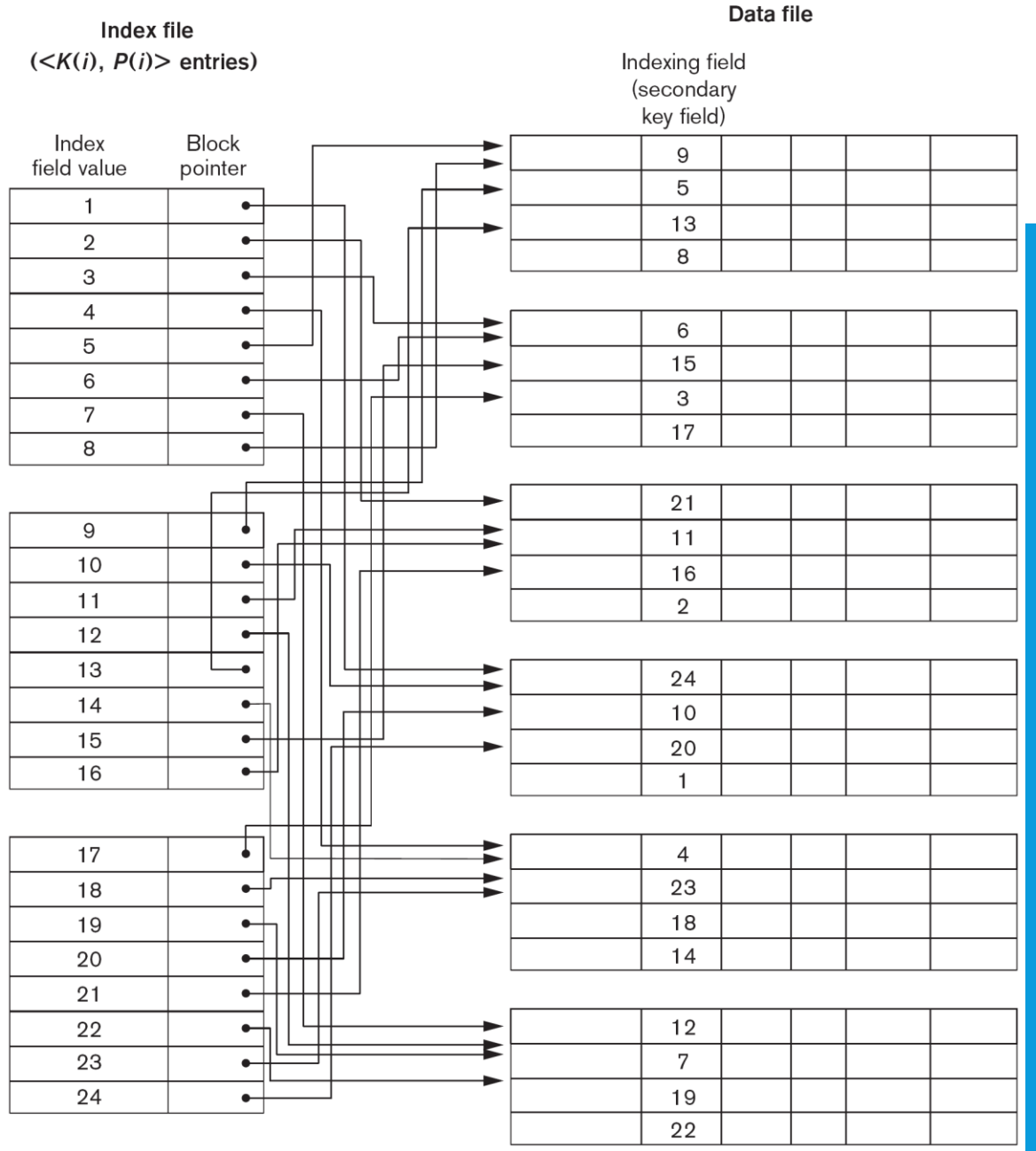
- It is another example of sparse index
  - One index entry points to multiple records (ordered)
- Ordered data file → overflow problem
  - Use overflow pointers or reservation
  - Example: reserving the whole block for handling overflow due to record insertion (anchoring block)



Secondary Index (1/2)

- 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

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



Secondary Index (2/2)

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

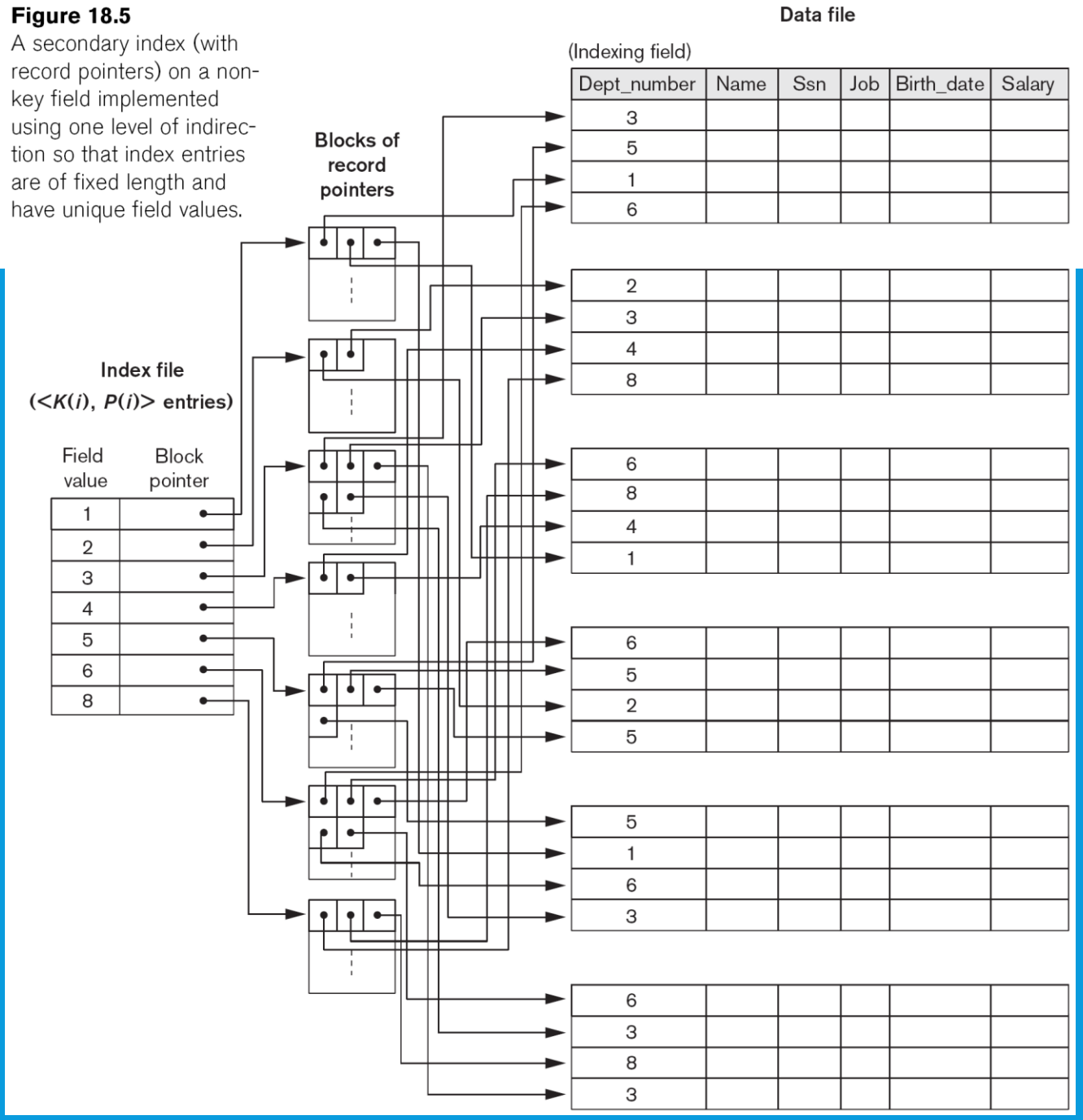


Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Secondary Index: Example 1

- 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/R = 1,024/100 = 10 records/block
  - Number of file blocks b = r/Bfr = 30,000/10 = 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 is b/2 = 3,000/2 = 1,500 block accesses on average

Secondary Index: Example 2

- 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/R_I= 1,024/15= 68 entries/block
  - In a dense index, the total number of index entries = number of records = r = 30,000
  - Number of index blocks b_I = r/ Bfr_I = 30,000/68 = 442 blocks
  - Binary search needs $\log_2 b_I = \log_2 442$ = 9 block accesses
  - To get the required record = 9 + 1 block accesses
  - This is compared to an average linear search cost of:
    - b/2 = 3,000/2= 1,500 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $Log_2 b = \log_2 30{,}000$= 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)

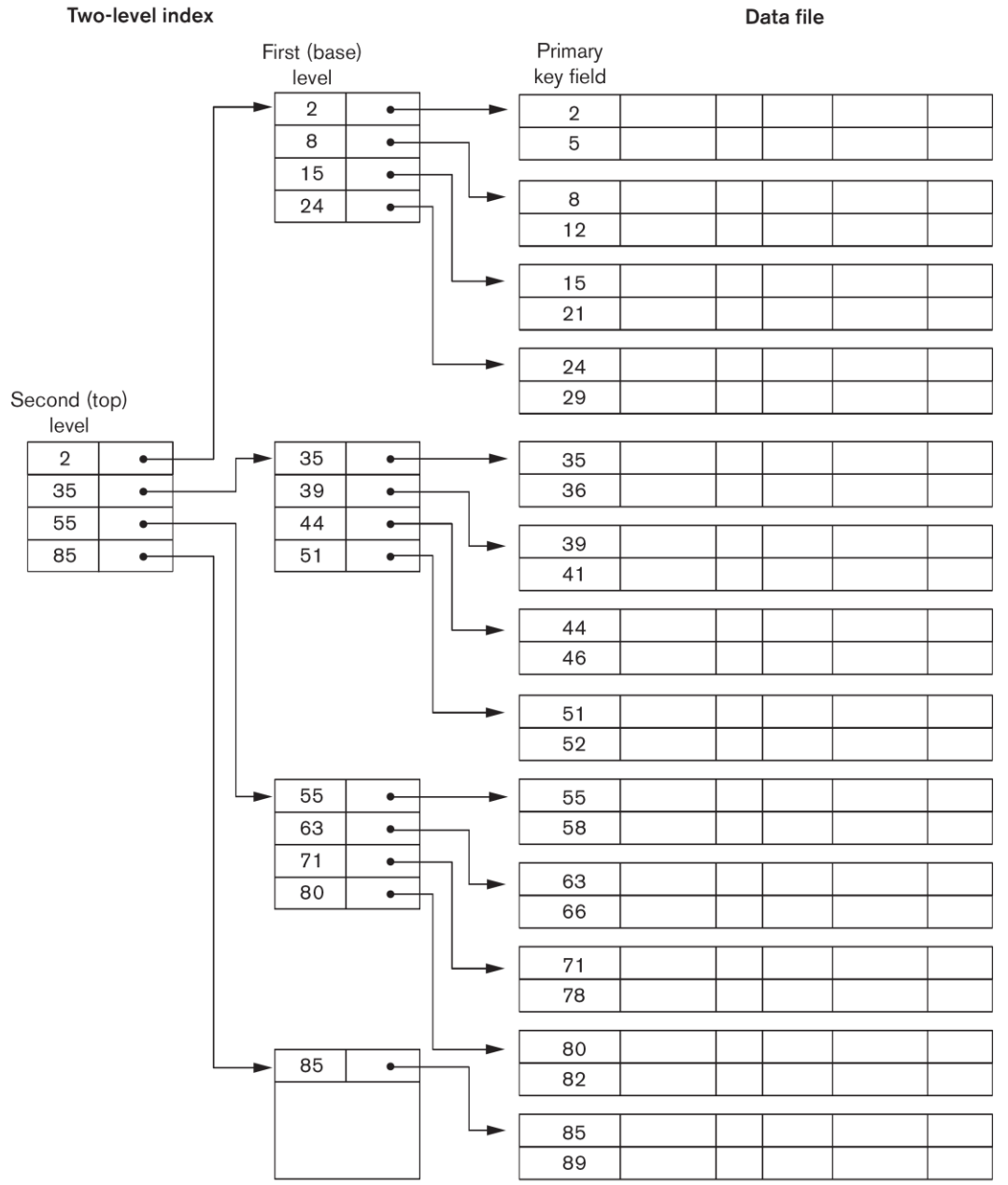
Table 18.2 Properties of Index Types

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

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



Multi-Level Indexes: Example

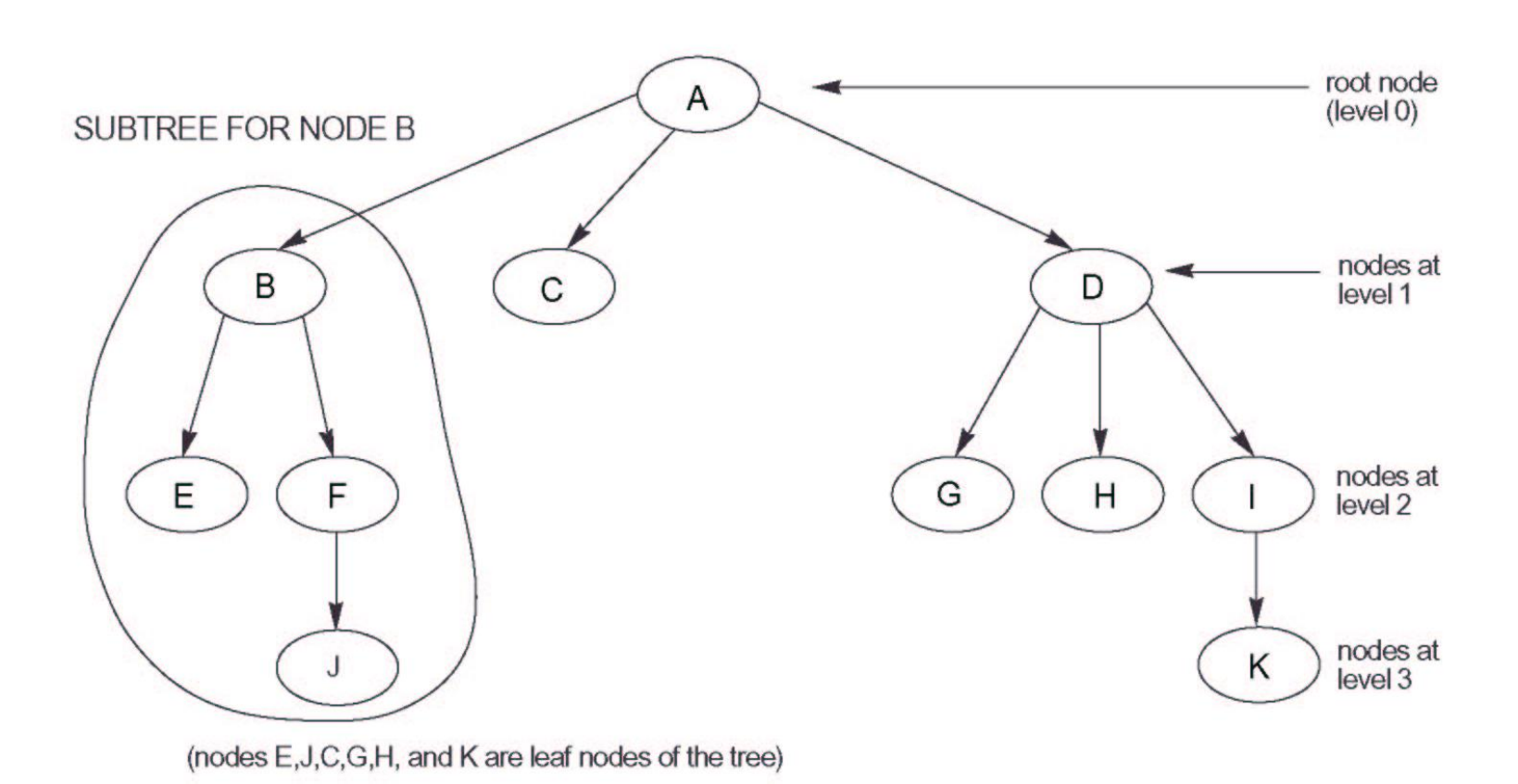
- Suppose the dense secondary index of Example 2 is converted into a multi-level index
  - The index blocking factor (also called fan out, fo) bfr_i = 68 index entries per block
  - The number of first level blocks b_1 =442
  - The number of second-level blocks $b_2 = b_1/fo = 442/68 = 7$ blocks
  - The number of third-level blocks $b_3 = b_2/fo = 7/68 = 1$
  - 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 (1/2)

- 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

Tree Index Structure (2/2)

- A tree data structure that shows an unbalanced tree



Dynamic Multilevel Indexes Using B-Trees and B+-Trees (1/2)

- 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 (2/2)

- 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)

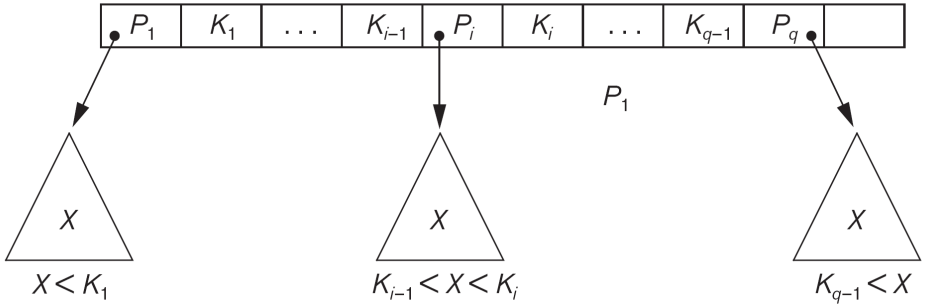
Difference between B-tree and B+-tree

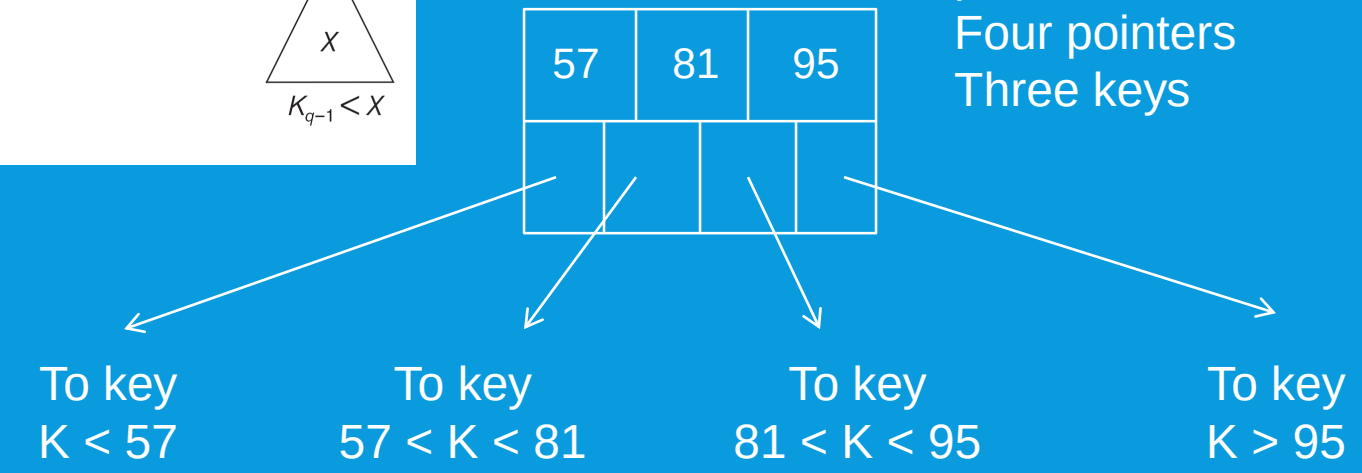
- In a B-tree, pointers to data records exist at all levels of the tree
- In a B^+-tree, all pointers to data records exists at the leaf-level nodes
- A B^+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree since its entry is smaller in size
- The internal nodes of B+-tree contain pointers only

B-tree Index (1/2)

- 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 (2/2)

- 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 $q \leq p$ (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 (i) $K_{i-1} < X < K_i$ for $1 < i < q$; (ii) $X < K_i$ for $i = 1$; and (iii) $K_{i-1} < X$ 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

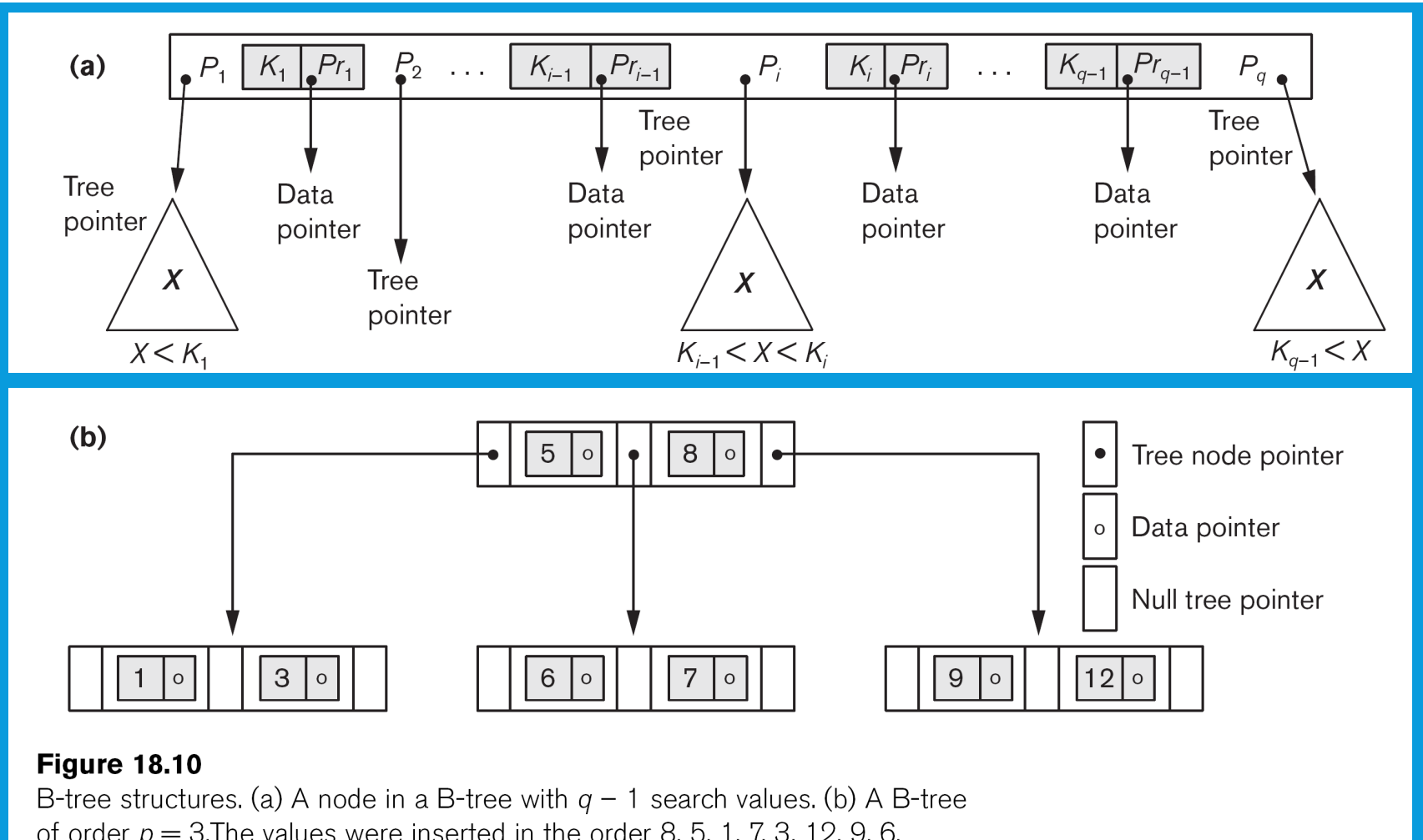


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

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 (i.e., each node has at most 23 pointers)
- 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)

B^+-tree Index (1/2)

- 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)

The Structure of Internal Nodes of a B^+-tree

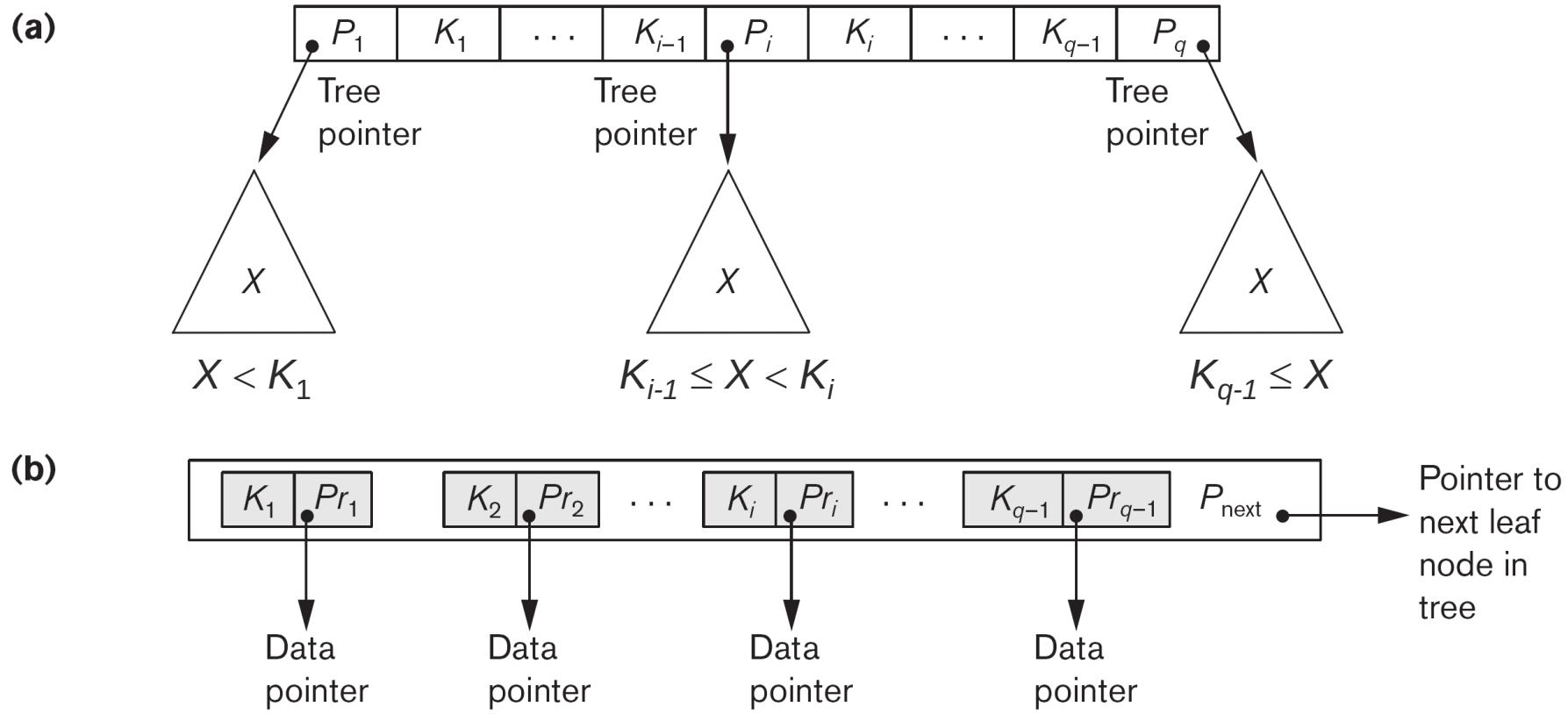
- Each internal node in a B^+-tree is of the form $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$ where $q \leq p$ and P_i is a tree pointer and is also called the tree order
- Each node has at most p tree pointers
- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$, for all search key field values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

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 (or degree) 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 $\lfloor p_{leaf}/2 \rfloor$ data records
- 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 1

- Suppose the search key field is V = 9 bytes, 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 values. These must fit into a single block.
  - $(p * P) + [(p – 1) * V] \leq B$
  - $(p * 6) + [(p – 1) * 9] \leq 512$
  - $15p \leq 512$
  - $p = 34$

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

B^+-tree Index Capacity

- To calculate the approximate number of entries in B^+-tree, we assume each node is 69% full
- On 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

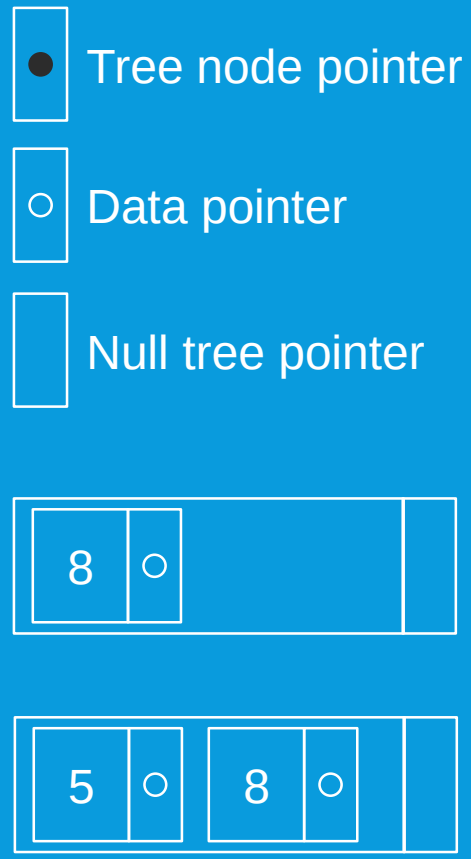
Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Level 3:	12,167 nodes	255,507 data record pointers	

B^+-tree Index Insertion

- Step 1: Descend to the leaf node where the key fits.
- Step 2:
  - (Case 1) If the node has an empty space, insert the key into the node.
  - (Case 2) If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. (Note: $\lceil (n+1)/2 \rceil$-th key is the middle key, hence, the first $\lfloor (n+1)/2 \rfloor$ keys are arranged in the first node and the remaining keys are arranged in the second node.)
    - If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node.
    - If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

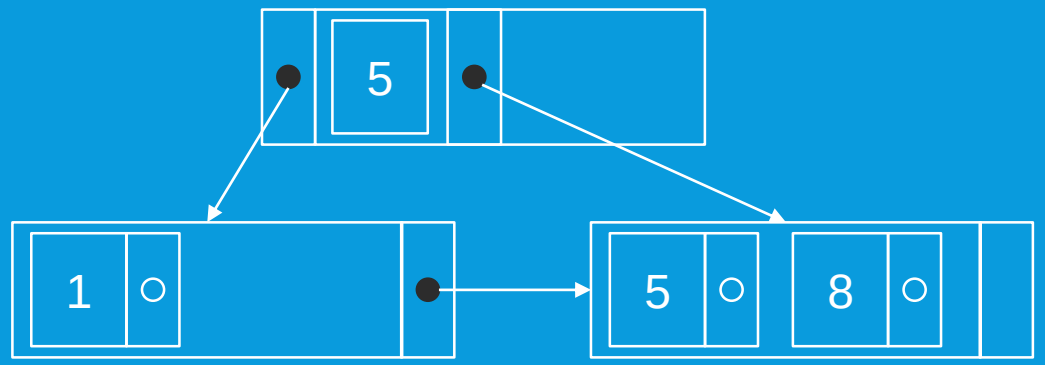
Example of an Insertion in a B$^+$-tree (1/5)

- Insert 8, 5, 1, 7, 3, 12 into an empty B$^+$-tree, in which (i) an internal node can fit three tree node pointers and two key values and (ii) a leaf node can fit two key values with data pointers.
- Insert 8
  - The node has an empty space
- Insert 5
  - The node has an empty space



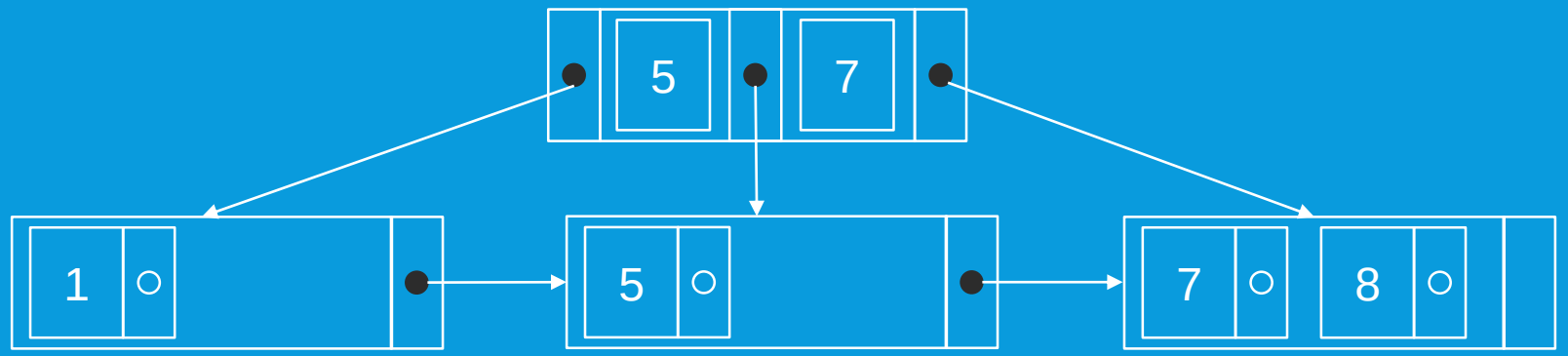
Example of an Insertion in a B$^+$-tree (2/5)

- Insert 1
  - Since the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. Since the node is a leaf, take a copy of the minimum value in the second of these two nodes (i.e., 5) and repeat the insertion algorithm to insert it into the parent node.



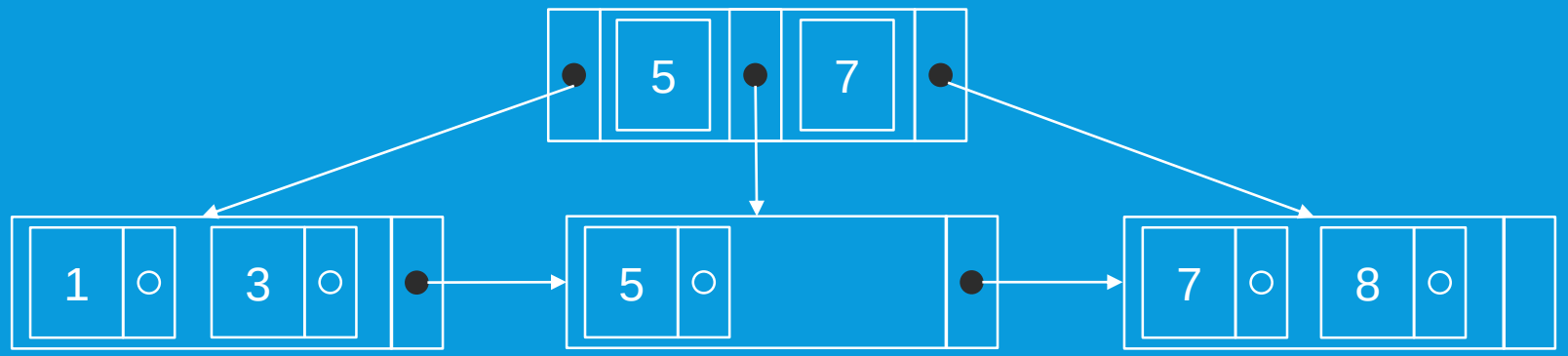
Example of an Insertion in a B$^+$-tree (3/5)

- Insert 7
  - Since the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. Since the node is a leaf, take a copy of the minimum value in the second of these two nodes (i.e., 7) and repeat the insertion algorithm to insert it into the parent node.



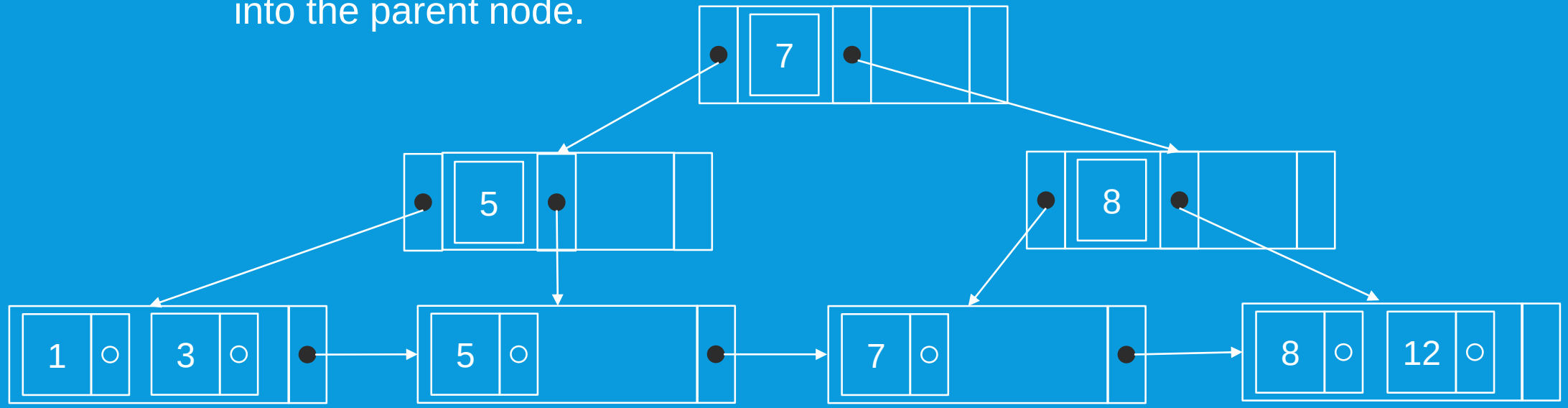
Example of an Insertion in a B$^+$-tree (4/5)

- Insert 3
  - The node has an empty space



Example of an Insertion in a B$^+$-tree (5/5)

- Insert 12
  - Since the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. Since the node is a leaf, take a copy of the minimum value in the second of these two nodes (i.e., 8) and repeat this insertion algorithm to insert it into the parent node. Since the parent node is a non-leaf, exclude the middle value (i.e., 7) during the split and repeat the insertion algorithm to insert 7 into the parent node.

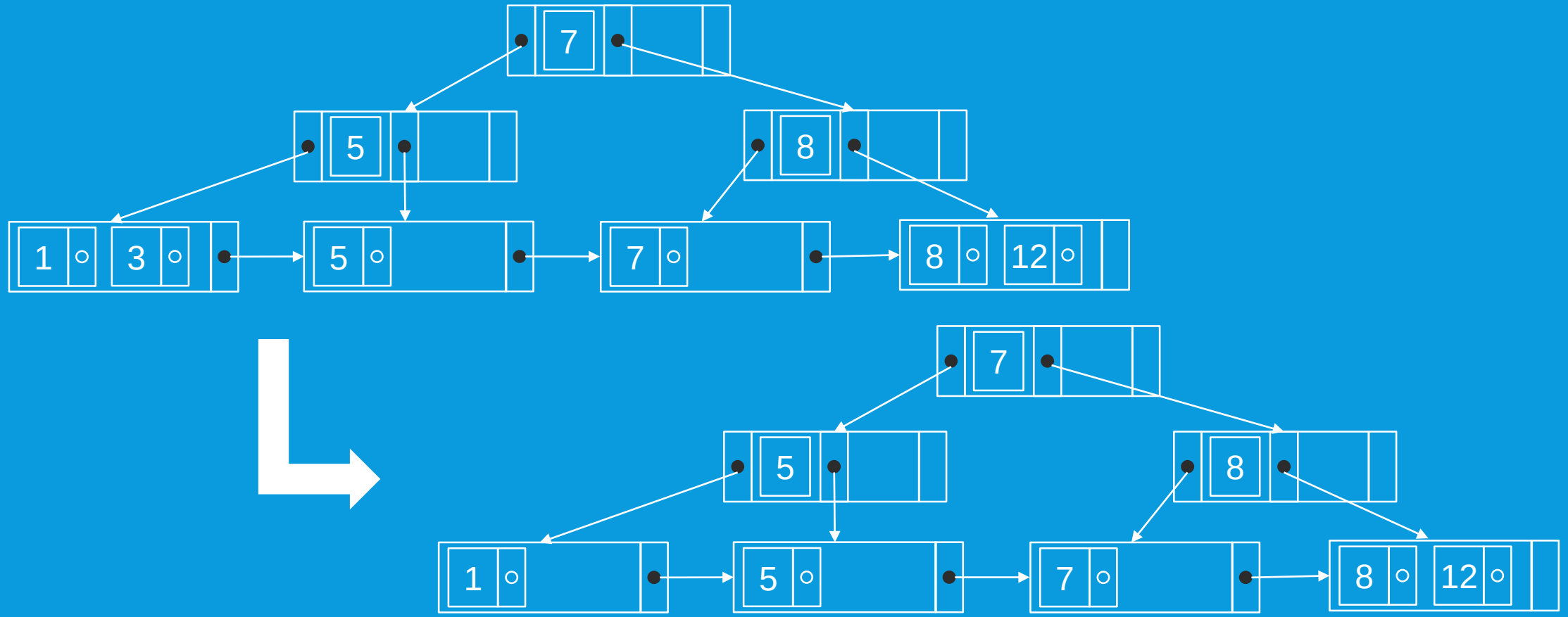


B^+-tree Index Deletion

- Step 1: Descend to the leaf where the key exists.
- Step 2: Remove the required key and associated reference from the node.
  - (Case 1) If the node still has half-full keys, stop.
  - (Case 2) If next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
  - (Case 3) Merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

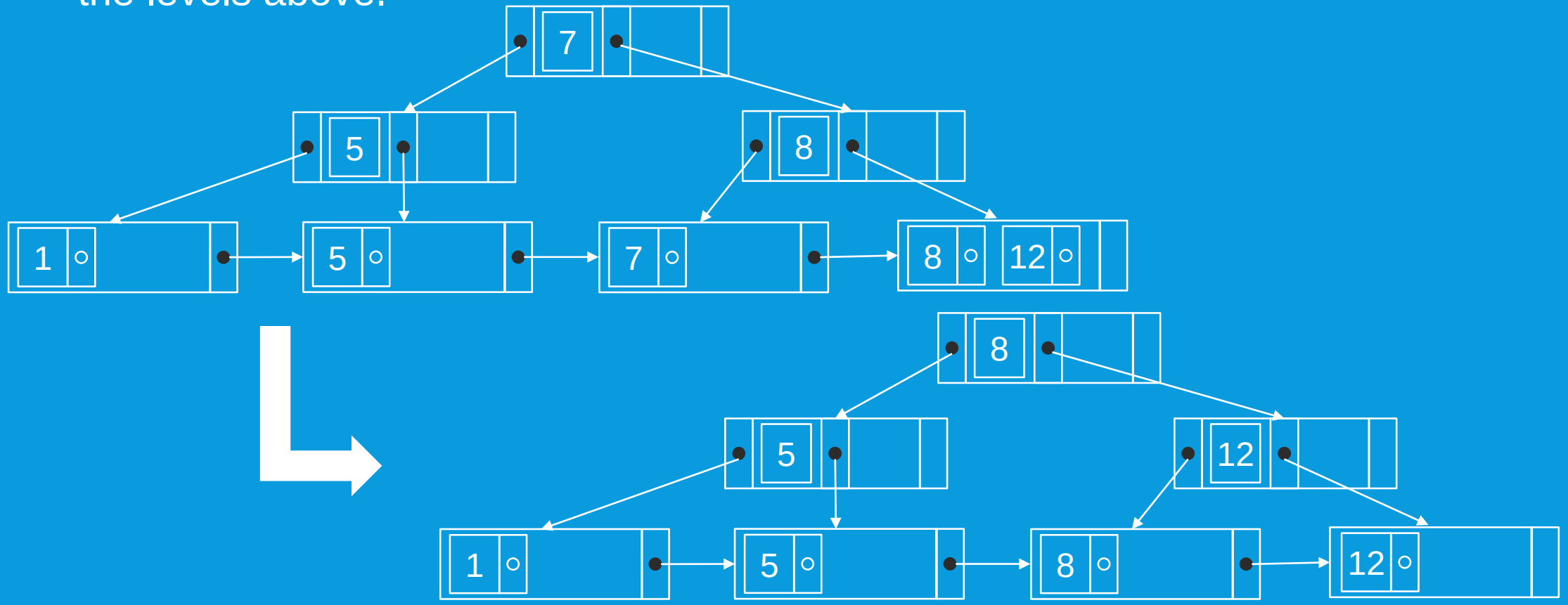
Example of a Deletion in a B$^+$-tree (1/3)

- Delete 3: the node still has half-full keys



Example of a Deletion in a B$^+$-tree (2/3)

- Delete 7: Distribute the keys between the node and the neighbor. Repair the keys in the levels above.



Example of a Deletion in a B$^+$-tree (3/3)

- Delete 12: (1) Merge the node with its sibling, (2) remove the “split key” (i.e., 12) from the parent, (3) merge the parent node with its sibling, and (2) remove the “split key” (i.e., 8) from its parent node (i.e., the root)