Lecture 8: Indexing Techniques

CS3402 Database Systems

Introduction (1/2)

- In this lecture, we assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations that were described in last lecture.
- We will describe additional auxiliary **access structures** called **indexes.**
  - The index structures are used to speed up the retrieval of records in response to certain search conditions.
  - They are additional files on disk that provide **secondary access paths**, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk.
  - They enable efficient access to records based on the **indexing fields** that are used to construct the index.

Introduction (2/2)

- Basically, any field of the file can be used to create an index, and multiple indexes on different fields—as well as indexes on multiple fields—can be constructed on the same file.
- A variety of indexes are possible; each of them uses a particular data structure to speed up the search.
- To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located.
- The most prevalent types of indexes are based on ordered files (single-level indexes) and use tree data structures (multilevel indexes, B^+-trees) to organize the index.

Types of Single-Level Ordered Indexes

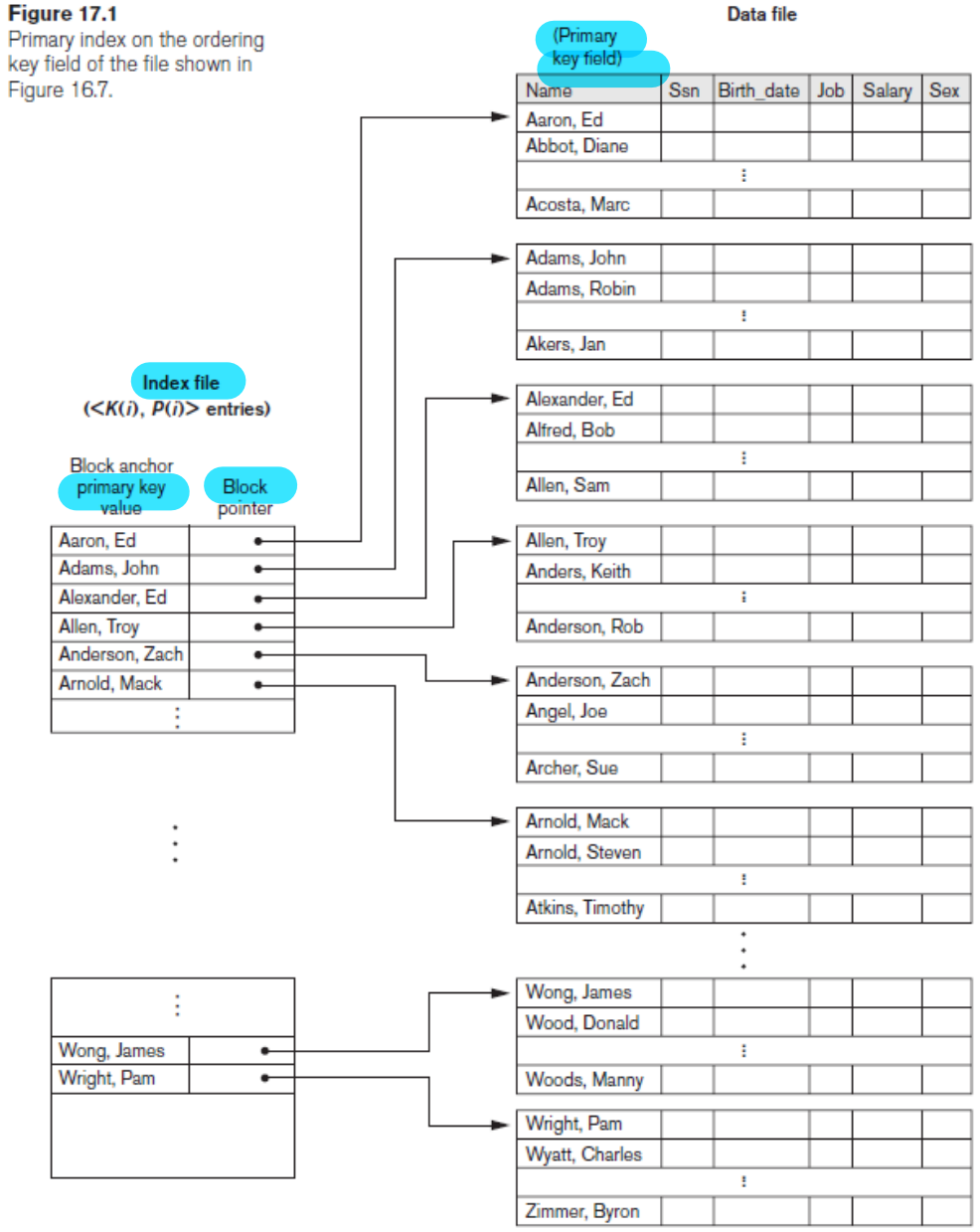
- A **primary index** is specified on the ordering key field of an **ordered file** of records. An ordering key field is used to physically order the file records on disk, and every record has a unique value for that field.
- If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field—another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case.
- Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, but not both.
- A third type of index, called a **secondary index**, can be specified on any nonordering field of a file. A data file can have several secondary indexes in addition to its primary access method.

Primary Indexes (1/6)

- A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file.
  - The first field is of the same data type as the ordering key field—called the **primary key**—of the data file.
  - The second field is a pointer to a disk block (a block address).
- There is one **index entry** (or **index record**) in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values.

Primary Indexes (2/6)

- To create a primary index on the ordered file using the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique).
- Each entry in the index has a Name value and a pointer. The first three index entries are as follows:
  - <K(1) = (Aaron, Ed), P(1) = address of block 1>
  - <K(2) = (Adams, John), P(2) = address of block 2>
  - <K(3) = (Alexander, Ed), P(3) = address of block 3>
- The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.



Primary Indexes (3/6)

- 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. A sparse index has fewer entries than the number of records in the file.
- Thus, 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 (or every record).

Primary Indexes (4/6)

- The index file for a primary index occupies a much smaller space than does the data file, for two reasons.
  - First, there are fewer index entries than there are records in the data file.
  - Second, each index entry is typically smaller in size than a data record because it has only two fields, both of which tend to be short in size; consequently, more index entries than data records can fit in one block.
- Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file.
- If the primary index file contains only b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

Primary Indexes (5/6)

- For example, suppose that we have an ordered file with $r = 300,000$ records stored on a disk with block size $B = 4,096$ bytes. File records are of fixed size and are unspanned, with record length $R = 100$ bytes.
- The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (4,096/100) \rfloor = 40$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (300,000/40) \rceil = 7,500$ blocks.
- A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 7,500) \rceil = 13$ block accesses.

Primary Indexes (6/6)

- Suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.
- The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/Ri) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block.
- The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks.
- To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses.
- To search for a record using the index, we need one additional block access to the data file for a total of 5 + 1 = 6 block accesses (i.e., an improvement over binary search on the data file, which required 13 disk block accesses)

Clustering Indexes (1/4)

- If file records are physically ordered on a nonkey field—which does not have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file.**
- We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field.

Clustering Indexes (2/4)

- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer.
- There is one entry in the clustering index for each distinct value of the clustering field, and it contains the value and a pointer to the first block in the data file that has a record with that value for its clustering field.

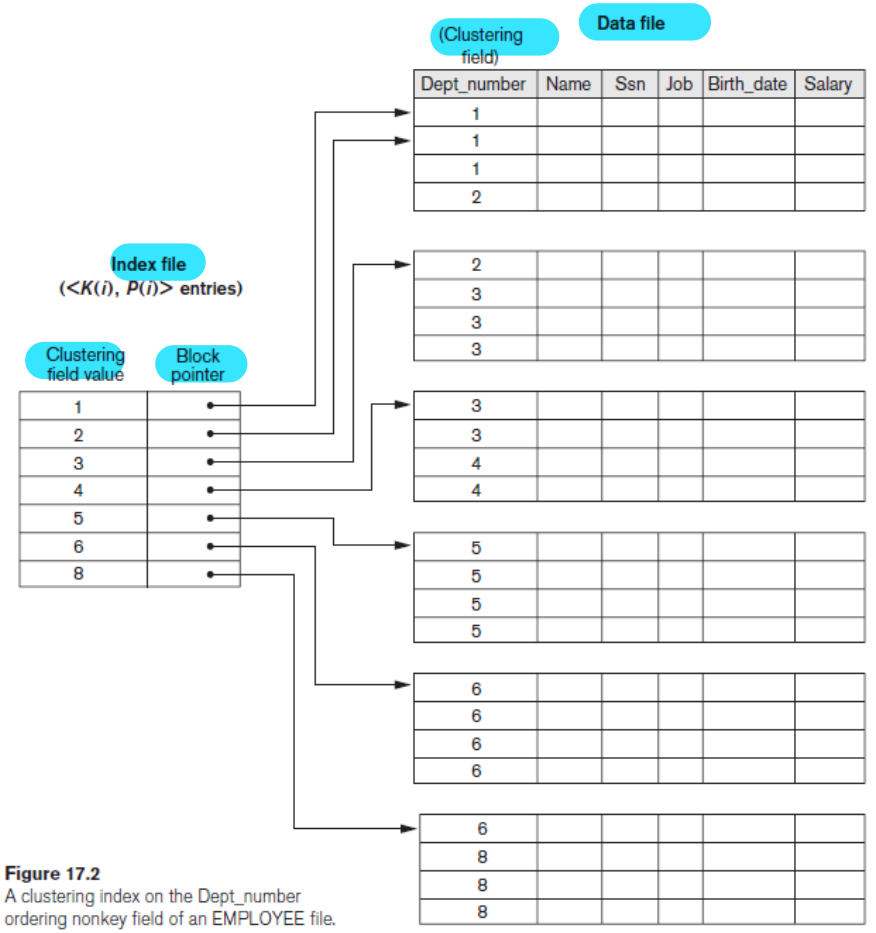
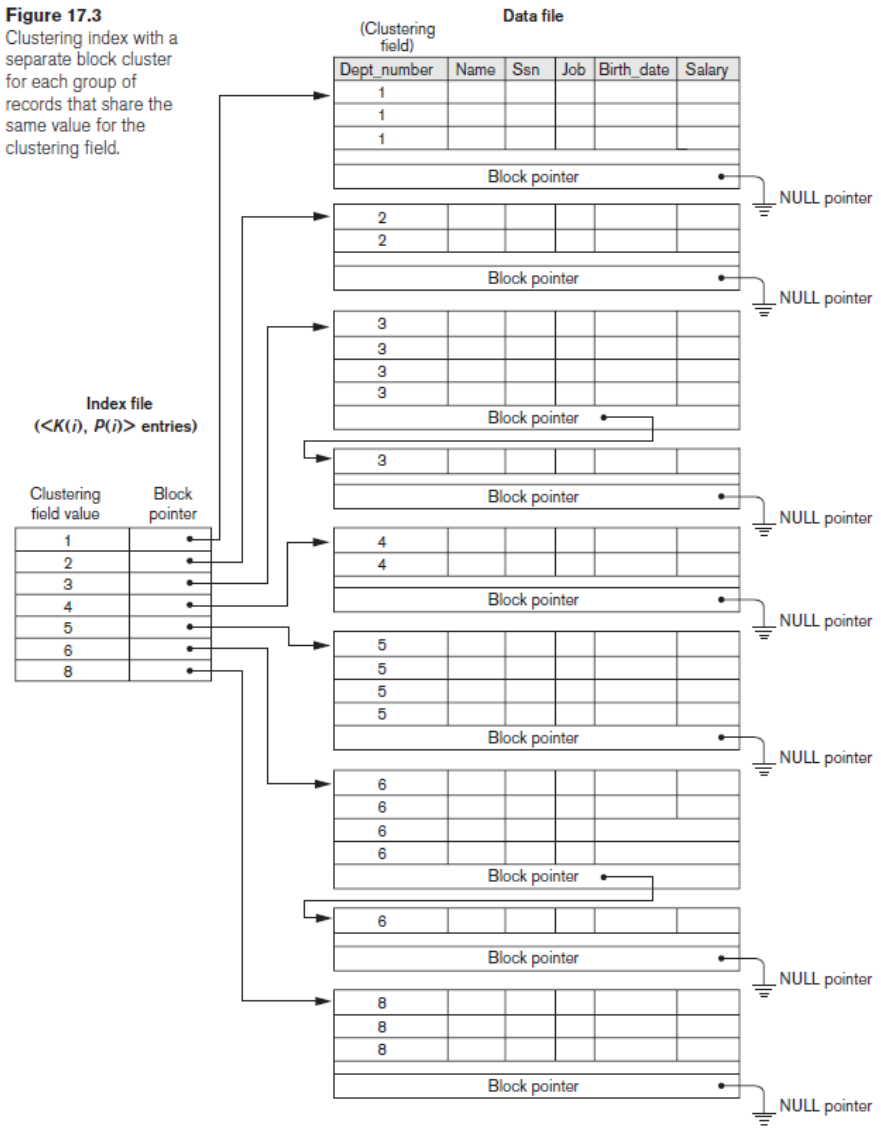


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Clustering Indexes (3/4)

- Notice that record insertion and deletion still cause problems because the data records are physically ordered.
- To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field; all records with that value are placed in the block (or block cluster).
- This makes insertion and deletion relatively straightforward.
- Use overflow blocks to handle the overflow problem.



Clustering Indexes (4/4)

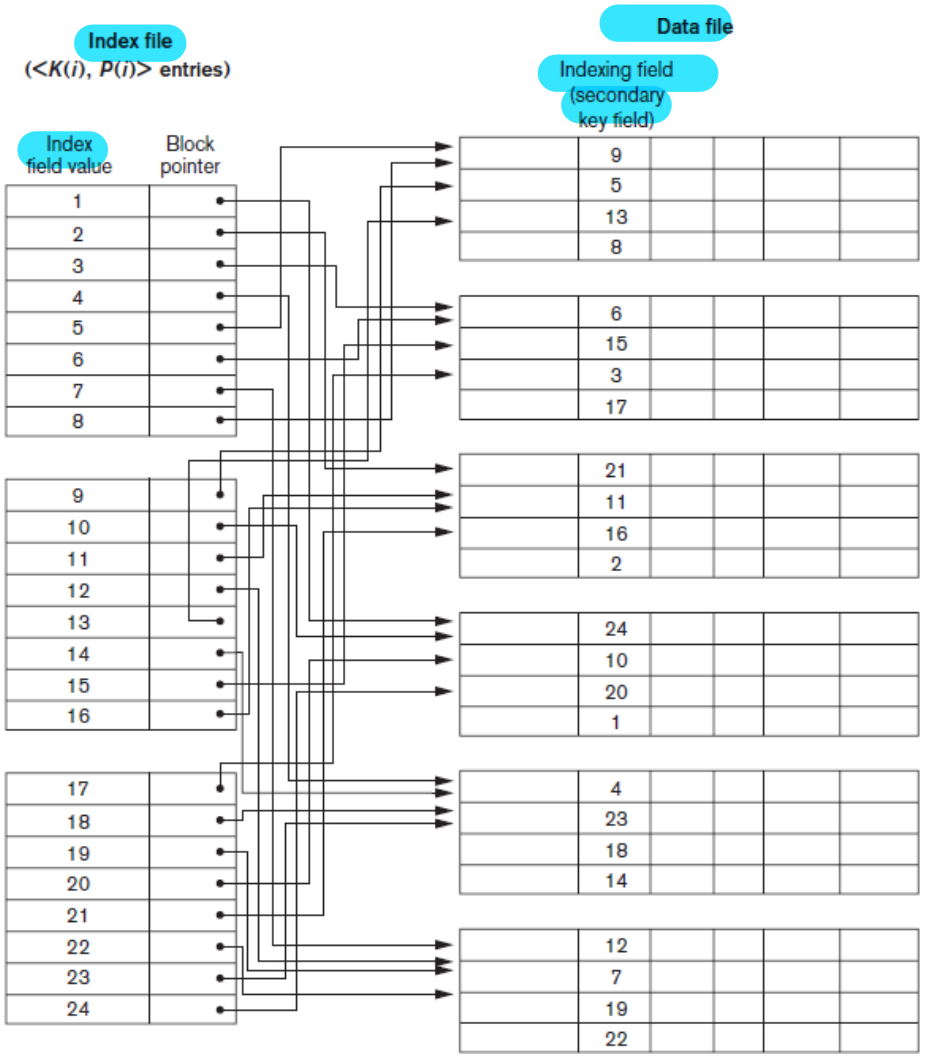
- For example, suppose that we consider the same ordered file with r = 300,000 records stored on a disk with block size B = 4,096 bytes.
- Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.)
- The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/11) \rfloor = 372$ index entries per block.
- The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1,000/372) \rceil = 3$ blocks.
- To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$ block accesses.

Secondary Indexes (1/5)

- A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed.
- Many secondary indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field.
- The index is again an ordered file with two fields.
  - The first field is of the same data type as some nonordering field of the data file that is an **indexing field**.
  - The second field is either a block pointer or a record pointer.

Secondary Indexes (2/5)

- First we consider **a secondary index access structure** **on a** **key (unique) field** that has a distinct value for every record.
- The entries are **ordered** by value of the first field, so we can perform a binary search. Because the records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors.
- That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index.
- In the example, a secondary index in which the pointers in the index entries are block pointers, not record pointers.



Secondary Indexes (3/5)

- Consider the file with r = 300,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 4,096 bytes.
- The value of bfr is $\lfloor (B/R) \rfloor = \lfloor (4{,}096/100) \rfloor = 40$ records per block. The file has $b = \lceil (r/bfr) \rceil = \lceil (300{,}000/40) \rceil = 7{,}500$ blocks.
- Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is V = 9 bytes long. Without the secondary index, to do a linear search on the file would require b/2 = 7,500/2 = 3,750 block accesses on the average.

Secondary Indexes (4/5)

V = 9 bytes long

- Suppose that we construct a secondary index on that nonordering key field of the file. A block pointer is P = 6 bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/15) \rfloor = 273$ index entries per block.
- In a dense secondary index such as this, the total number of index entries r_i is equal to the number of records in the data file, which is 300,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (300,000/273) \rceil = 1,099$ blocks.
- A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 1,099) \rceil = 11$ block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 11 + 1 = 12 block accesses—a vast improvement over the 3,750 block accesses needed on the average for a linear search.

Secondary Indexes (5/5)

- **We can also create a secondary index on a nonkey, nonordering field of a file.** In this case, numerous records in the data file can have the same value for the indexing field.
- An extra level of indirection is created to handle the multiple pointers. In this scheme, the second field in an index entry is a pointer to a disk block, which contains a set of record pointers; each record pointer in that disk block points to one of the data file records with value for the indexing field.
- If some value occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.

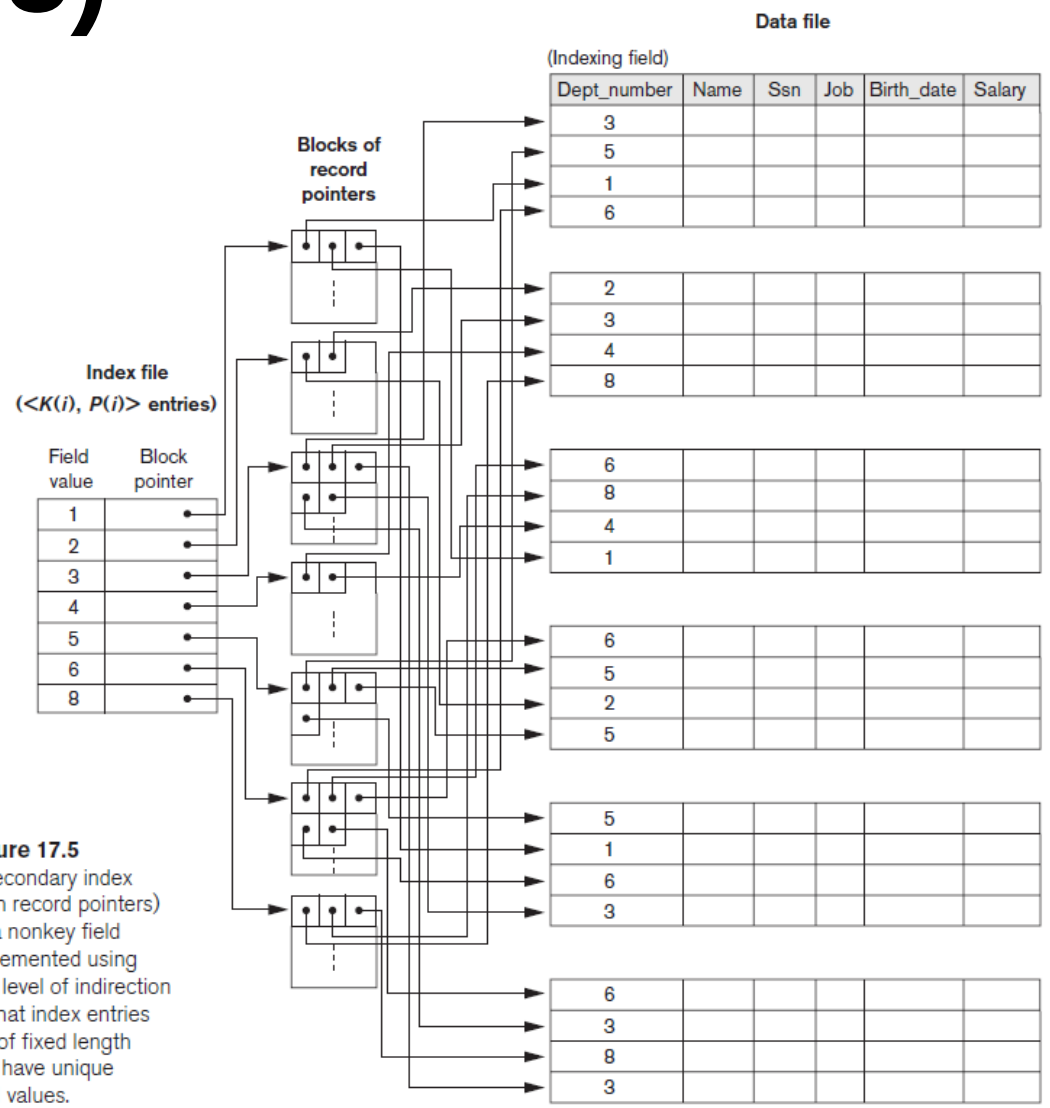


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Multilevel Indexes (1/7)

- 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**.

Multilevel Indexes (2/7)

- 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.
- 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 $2^{32} - 1 = 4.2 * 10^9$ blocks) and a 9-byte key such as SSN, the fan-out comes to 315. 4096/13

Multilevel Indexes (3/7)

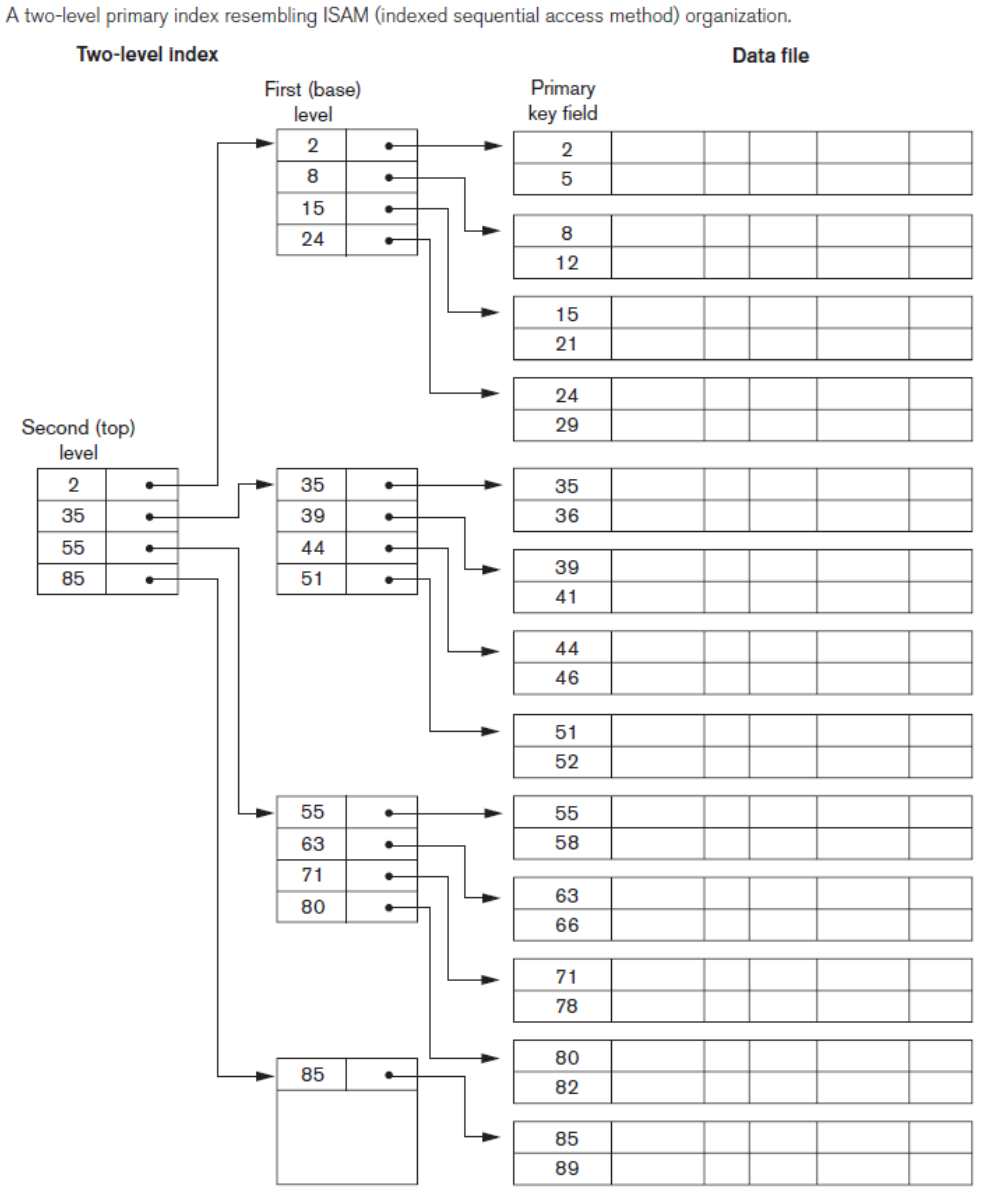
- A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a multilevel index, as an ordered file with a distinct value for each key field in the index entry.
- Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index.
- Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level. The blocking factor bfr_i for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address.
- If the first level has r_1 entries, and the blocking factor—which is also the fan-out—for the index is $bfr_i = fo$, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

Multilevel Indexes (4/7)

- We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is $r_3 = \lceil (r_2/fo) \rceil$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block.
- We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the *t*-th level is called the **top** index level.

Multilevel Indexes (5/7)

- The multilevel scheme described in this lecture can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has distinct values for the key field and fixed-length entries.
- Here shows an example of a multilevel index built over a primary index.



Multilevel Indexes (6/7)

- Suppose that the dense secondary index (slide #18) is converted into a multilevel index.
- We calculated the index blocking factor $bfr_i = 273$ index entries per block, which is also the fan-out fo for the multilevel index; the number of first-level blocks $b_1 = 1{,}099$ blocks was also calculated. 300e3/273
- The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (1{,}099/273) \rceil = 5$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. height
- To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to previous example (slide #18), where 12 block accesses were needed when a single-level index and binary search were used.

Multilevel Indexes (7/7)

- A multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value.
- We are still faced with the problems of dealing with index insertions and deletions, because all index levels are physically ordered files.
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks.

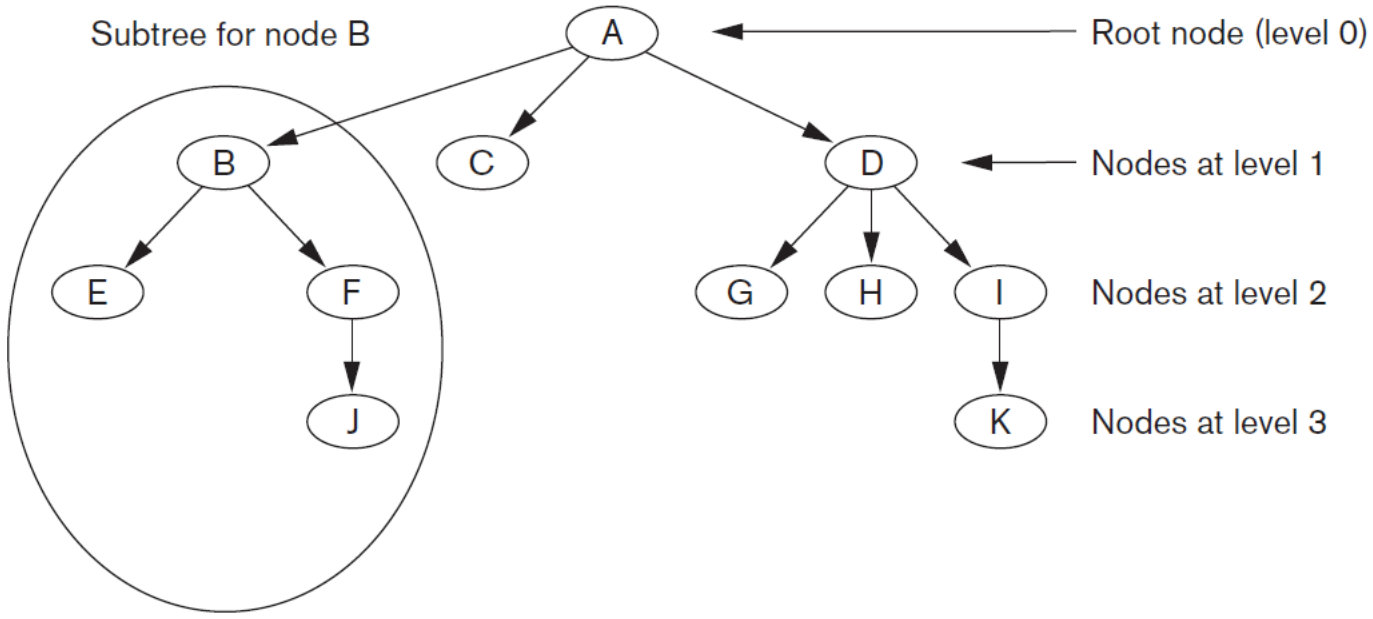
Dynamic Multilevel Indexes Using B-Trees and B+-Trees (1/2)

- B-trees and B^+-trees are special cases of the well-known search data structure known as a **tree**.
- A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node.
- The **level** of a node is always one more than the level of its parent, with the level of the root node being zero.
- A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node n and the subtrees of all the child nodes of n.

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

- The root node is A, and its child nodes are B, C, and D.
- Nodes E, J, C, G, H, and K are leaf nodes.
- Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

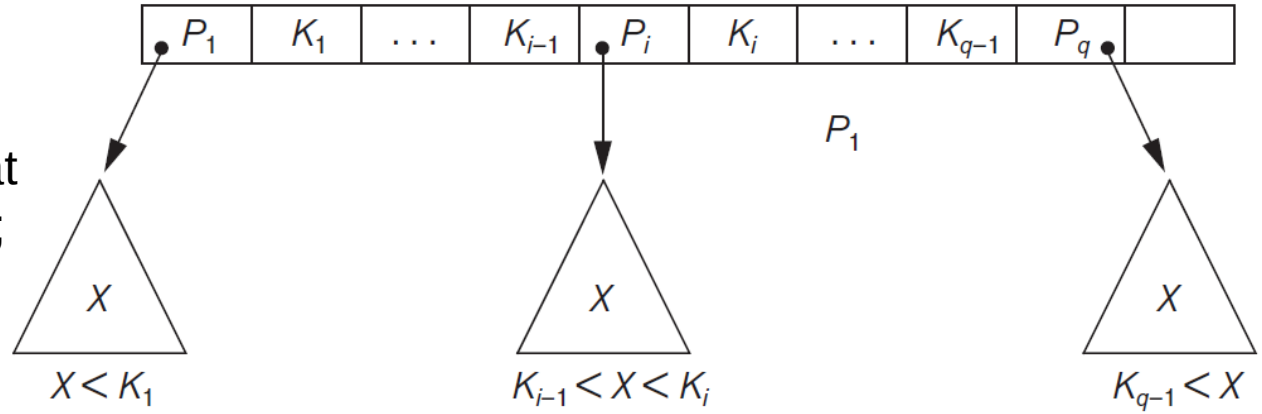
Figure 17.7
A tree data structure that shows an unbalanced tree.



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

Search Trees and B-Trees (1/3)

- 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 <P_1, K_1, P_2, K_2, … , P_{q-1}, K_{q-1}, P_q>, 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.
- Two constraints must hold at all times on the search tree:
  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
  - 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 $i = q$.



Search Trees and B-Trees (2/3)

- This example 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.

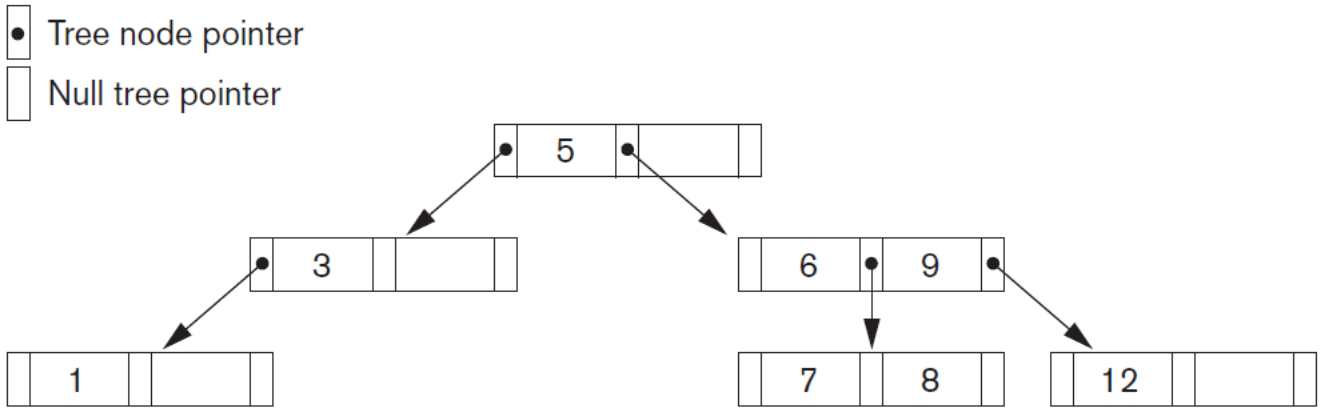


Figure 17.9
A search tree of order $p = 3$.

Search Trees and B-Trees (3/3)

- 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.
  - 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.

B-Trees (1/3)

- The B-tree has additional constraints that ensure that the tree is always balanced. More formally, a **B-tree of order** p, when used as an access structure on a key field to search for records in a data file, can be defined as follows:
  - Each internal node in the 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 P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
  - Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
  - For all search key field values X in the subtree pointed at by P_i (the i-th subtree), 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.
  - Each node, except the root and leaf nodes, has at least $\lfloor (p/2) \rfloor$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
  - A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).
  - All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers P_i are NULL.

B-Trees (2/3)

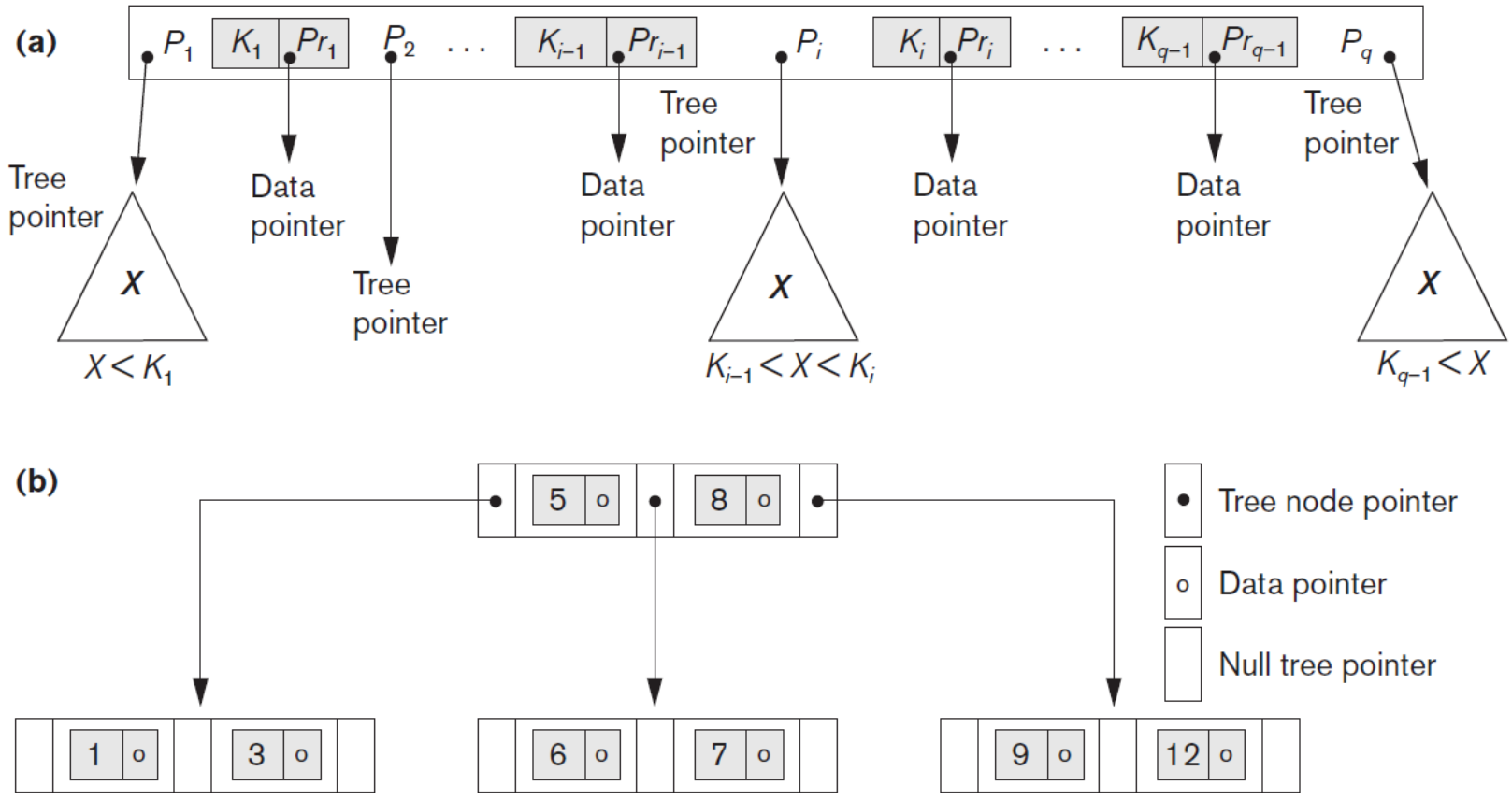


Figure 17.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

B-Trees (3/3)

- Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with p = 23. Assume that each node of the B-tree is 69% full. Each node, on the average, will have p * 0.69 = 23 * 0.69 or approximately 16 pointers and, hence, 15 search key field values. The **average fan-out** fo = 16.

p-1

- We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries (240x15)	256 pointers (240+16)
Level 2:	256 nodes	3,840 key entries (256x15)	4,096 pointers (3840+256)
Level 3:	4,096 nodes	61,440 key entries (4096x15)	

- At each level, we calculated the number of key entries by multiplying the total number of pointers at the previous level by 15.

B^+-Trees (1/9)

- Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B^+-tree**.
- In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B^+-tree, data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.
- In the B^+-tree, The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

B^+-Trees (2/9)

- The leaf nodes of the B^+-tree are usually linked to provide ordered access on the search field to the records.
- These leaf nodes are similar to the first (base) level of an index.
- Internal nodes of the B^+-tree correspond to the other levels of a multilevel index.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B^+-tree to guide the search.

B^+-Trees (3/9)

- The structure of the internal nodes of a B^+-tree of order p is as follows:
- Each internal node is of the form $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$ where and each P_i is a **tree pointer**.
  - Within eq ≤ p ach internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
  - For all search field values X in the subtree pointed at 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 internal node has at most p tree pointers.
  - Each internal node, except the root, has at least $\lfloor (p/2) \rfloor$ **tree pointers**. The root node has at least two tree pointers if it is an internal node.
  - An internal node with q pointers, q ≤ p, has q − 1 search field values.

B^+-Trees (4/9)

- The structure of the leaf nodes of a B^+-tree of order p is as follows:
  - Each leaf node is of the form $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$, where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next leaf node of the B^+-tree.
  - Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}$, $q \leq p$.
  - Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
  - Each leaf node has at least $\lfloor (p-1)/2 \rfloor$ **values**.
  - All leaf nodes are at the same level.

B+-Trees (5/9)

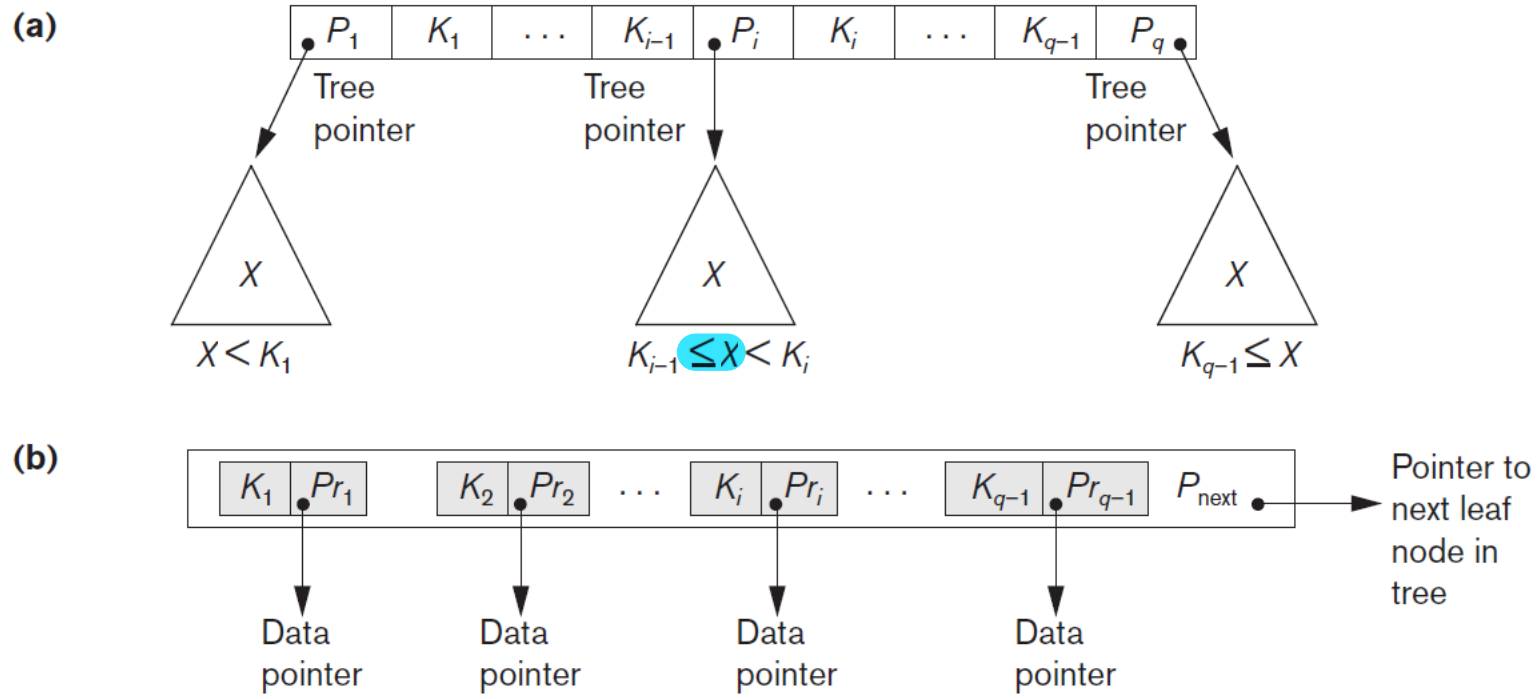


Figure 17.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

B^+-Trees (6/9)

- The pointers in internal nodes are tree pointers to blocks that are tree nodes, whereas the pointers in leaf nodes are data pointers to the data file records or blocks—except for the P_{next} pointer, which is a tree pointer to the next leaf node.
- By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field.
- For a B^+-tree on a nonkey field, an extra level of indirection is needed similar to the one shown for the secondary index, so the Pr pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file.

B^+-Trees (7/9)

- To calculate the order p of a B^+-tree, suppose that the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record pointer is Pr = 7 bytes, and a block pointer/tree pointer is P = 6 bytes.
- An internal node of the B+-tree can have up to p tree pointers and p − 1 search field values; these must fit into a single block. Hence, we have:

  $(p * P) + ((p - 1) * V) \leq B$

  $(p * 6) + ((p - 1) * 9) \leq 512$

  $(15 * p) \leq 521$

- We can choose p to be the largest value satisfying the above inequality, which gives p = 34. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree.

B^+-Trees (8/9)

- The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the number of data record pointers p_{leaf} for the leaf nodes can be calculated as follows:

  $(p_{leaf} * (Pr + V)) + P \leq B$

  $(p_{leaf} * (7 + 9)) + 6 \leq 512$

  $(16 * p_{leaf}) \leq 506$

- It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

B^+-Trees (9/9)

- Suppose that we construct a B^+-tree. To calculate the approximate number of entries in the B^+-tree, we assume that each node is 69% full.
- On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * p_{leaf} = 0.69 * 31 or approximately 21 data record pointers. A B^+-tree will have the following average number of entries at each level:

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	

- Thus, in this example, a three-level B^+-tree holds up to 255,507 record pointers, with the average 69% occupancy of nodes. Compare this to the 65,535 entries for the corresponding B-tree in the previous example.

B^+-Trees: Insertion (1/6)

- 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. $n = p_{leaf}$ for a leaf node and $n = p$ for an internal node)
    - (Case 2a) 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.
    - (Case 2b) If the node is a non-leaf, exclude the middle key value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

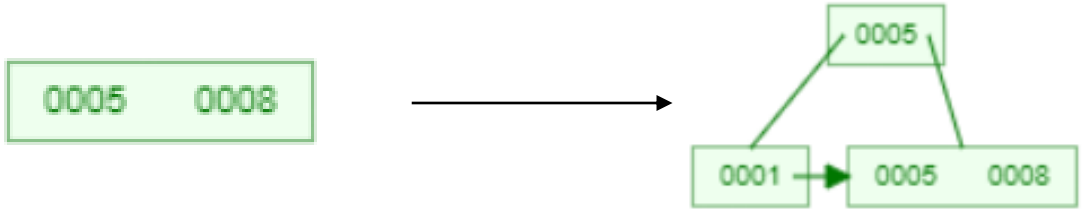
B$^+$-Trees: Insertion (2/6)

- 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 (Case 1, the root node can fit two key values)
- Insert 5
  - The node has an empty space (Case 1, the root node can fit two key values)

B$^+$tree algorithm visualization: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

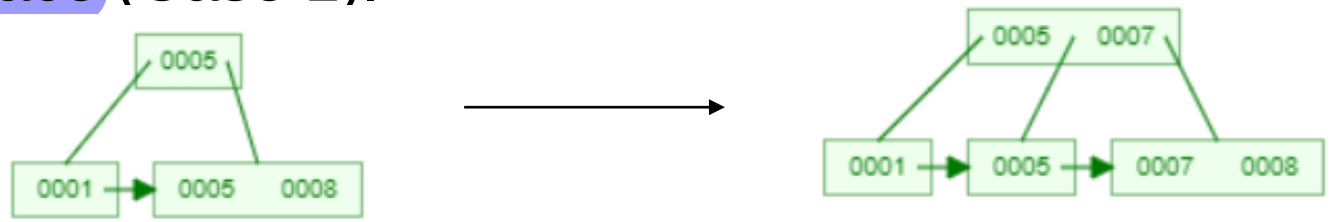
B$^+$-Trees: Insertion (3/6)

- Insert 1
  - Since the root node is already full (Case 2), split it into two nodes, distributing the keys evenly between the two nodes. The first $\lfloor (p_{leaf} + 1)/2 \rfloor = \lfloor (2+1)/2 \rfloor = 1$ key is arranged to the first node.
  - Since the node is a leaf (Case 2a), take a copy of the minimum value in the second node (i.e., 5) and repeat the insertion algorithm to insert it into the parent node. The parent node has an empty space (Case 1).



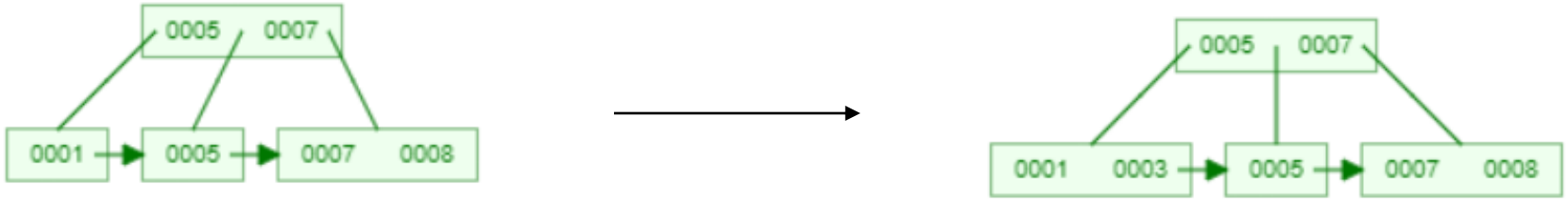
B+-Trees: Insertion (4/6)

- Insert 7
  - Since the node is already full (Case 2), split it into two nodes, distributing the keys evenly between the two nodes. The first key is arranged to the first node.
  - Since the node is a leaf (Case 2a), 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. The parent node has an empty space (Case 1).



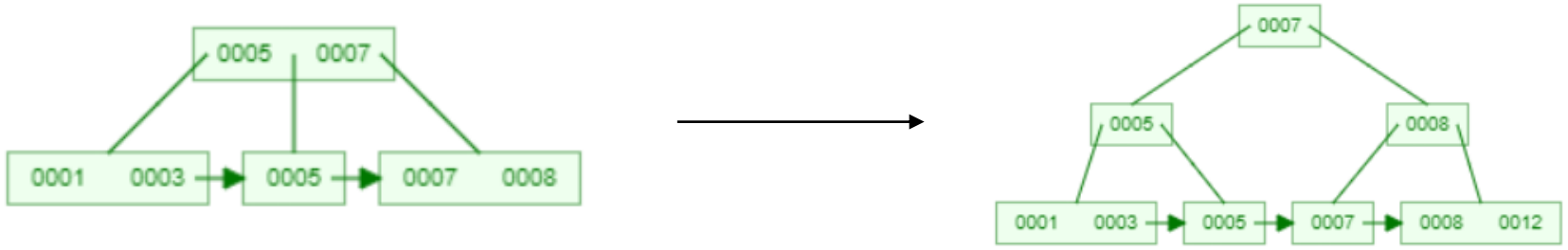
B+-Trees: Insertion (5/6)

- Insert 3
  - The node has an empty space (Case 1)



B+-Trees: Insertion (6/6)

- Insert 12
  - Since the node is already full (Case 2), split it into two nodes, distributing the keys evenly between the two nodes. The first key is arranged to the first node.
  - Since the node is a leaf (Case 2a), 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 and full (Case 2b), exclude the middle value (i.e., 7) during the split and repeat the insertion algorithm to insert 7 into the parent node.

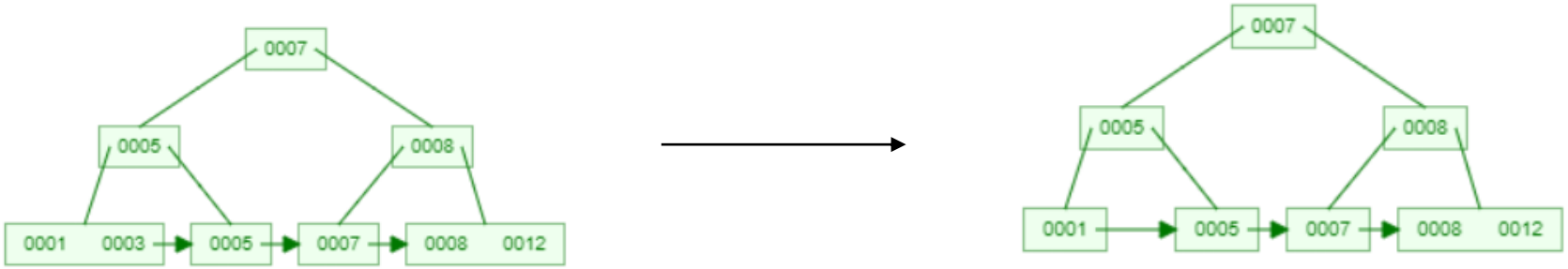


B^+-Trees: Deletion (1/4)

- 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).

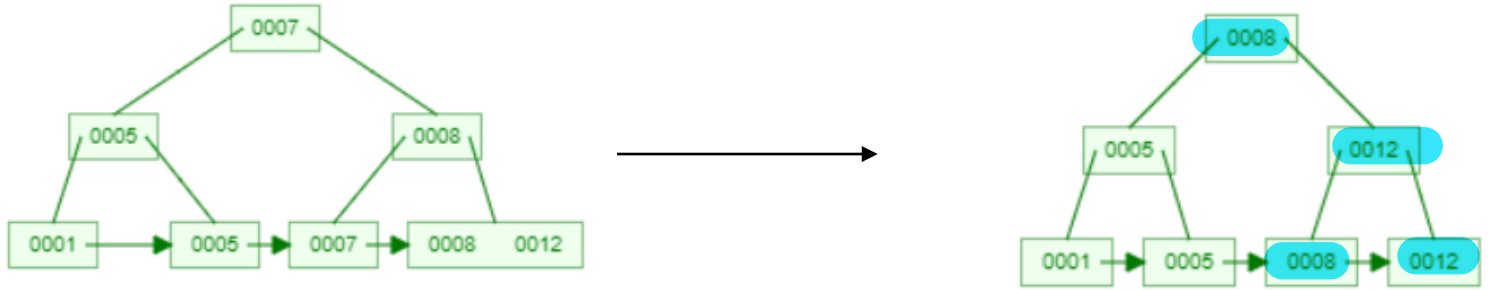
B+-Trees: Deletion (2/4)

- Delete 3
  - The node still has half-full keys (Case 1)



B+-Trees: Deletion (3/4)

- Delete 7
  - The sibling has more than necessary keys (Case 2). Distribute the keys between the node and the sibling (i.e., move 8 to the node).
  - The new "split point" is 12 at the upper level.
  - The new "split point" is 8 at the root.



B+-Trees: Deletion (4/4)

- Delete 12
  - Merge the node with its sibling (Case 3)
  - Remove the “split key” (i.e., 12) from the parent node
  - This node is a non-leaf, we need to incorporate the “split key” from the parent into our merging.