CS3402 : Chapter 9
Indexing Techniques

Indexes as Access Paths

- 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.

  - Without an index, we may use the sequential search (for unordered file or non-order attribute in ordered file)

  - Index are additional files on disk that provide secondary access paths, which provide fast access the records without affecting the physical placement of records in the primary data file on disk.

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)
- An index is an ordered sequence of entries <field value, pointer to record>, ordered by the field value
- The pointer provides the address of the record with the corresponding field value in the data file
- An index is usually specified on one field 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: index entry directly points to data record
- Primary index (ordering key field)
  - Specified on the ordering key field of an ordered file of records
- Cluster index (ordering 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 record in the data file. Thus larger index size
  - A sparse index, on the other hand, has index entries for only some records. 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 an ordered file with two fields<key field, block pointer>
    - 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
    - A primary index is a sparse index (small index)
    - The number of entries is much smaller than the number of records

Primary Index on the Ordering Key Field

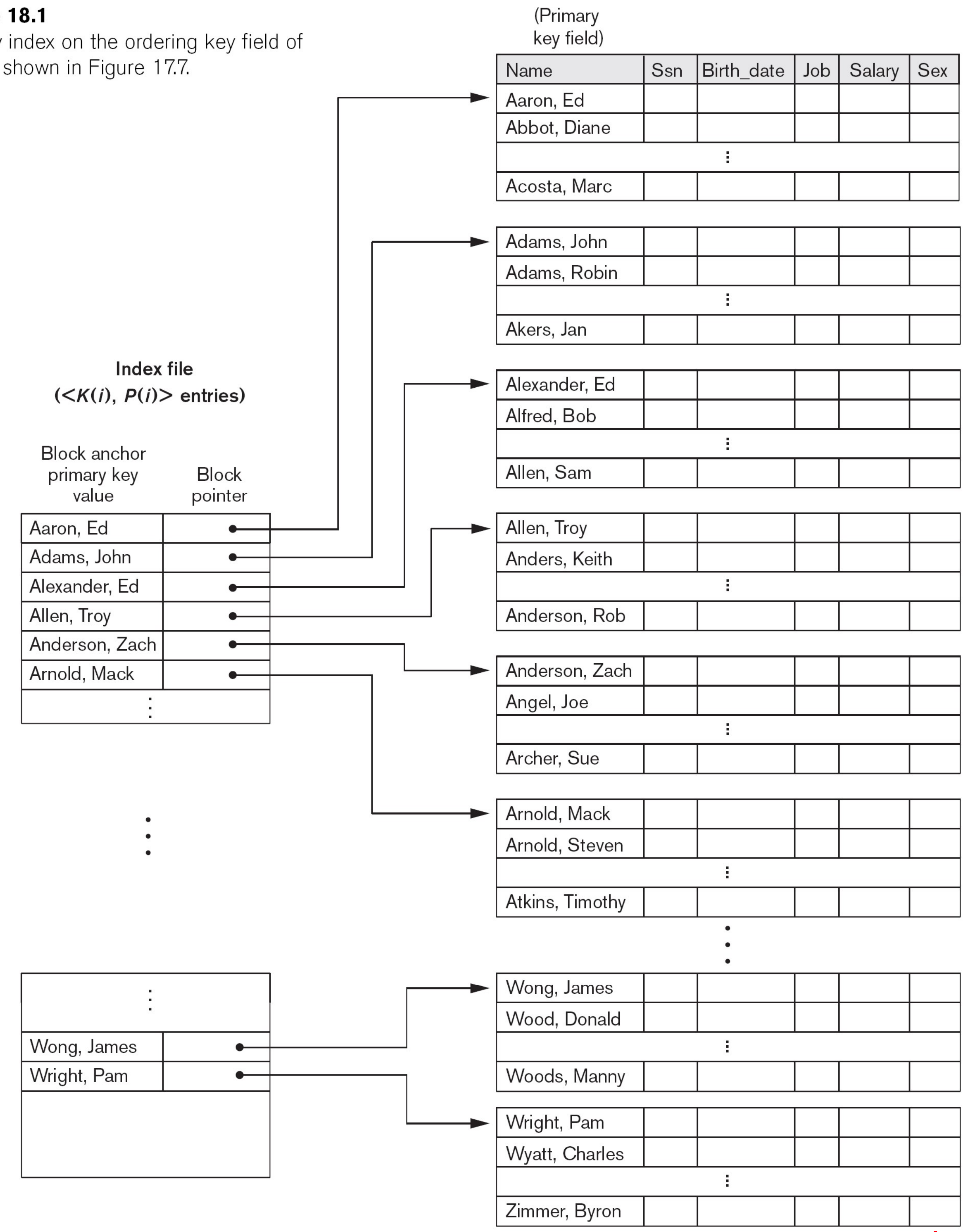


Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.

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 block 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 bI= (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, 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

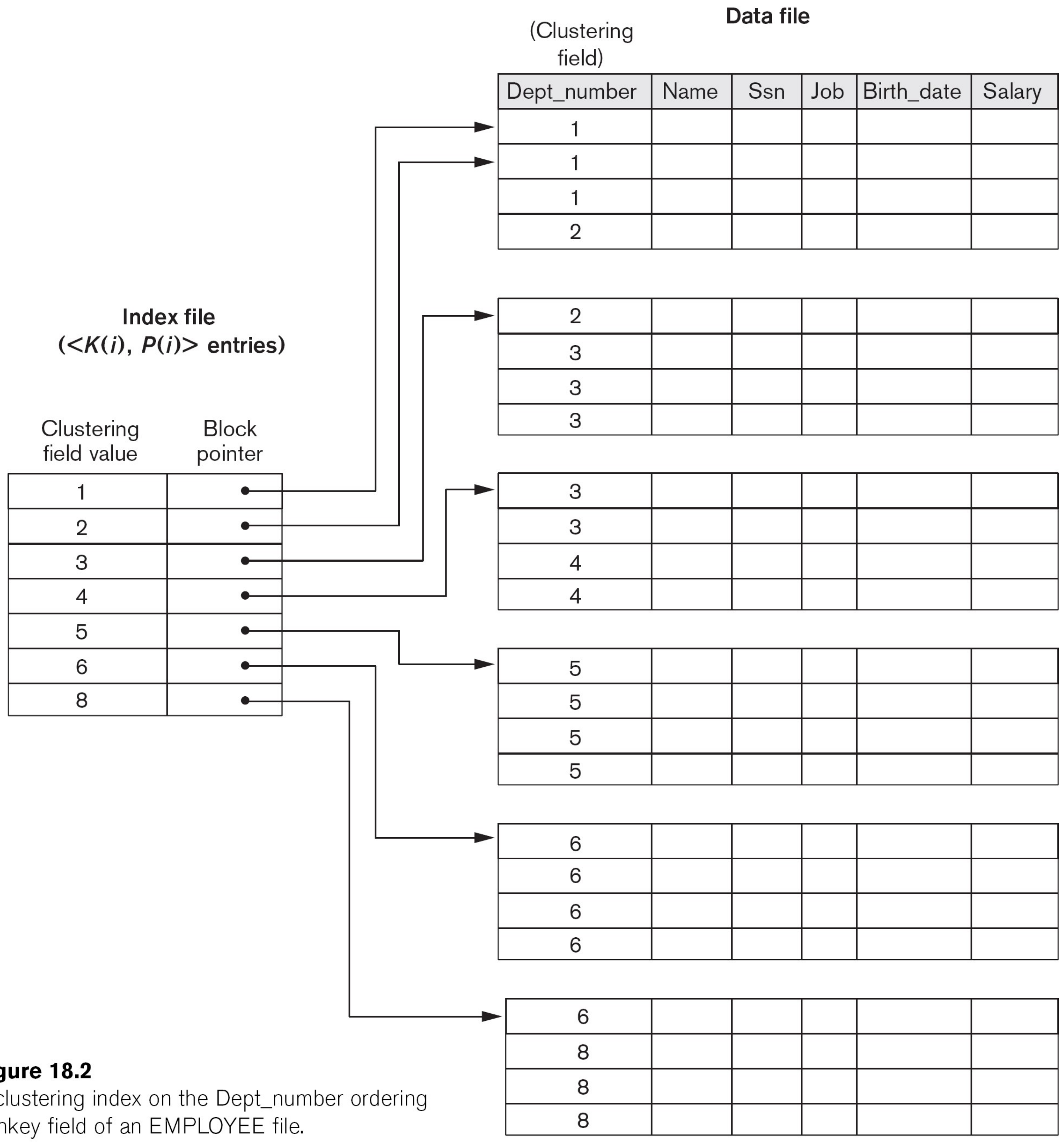
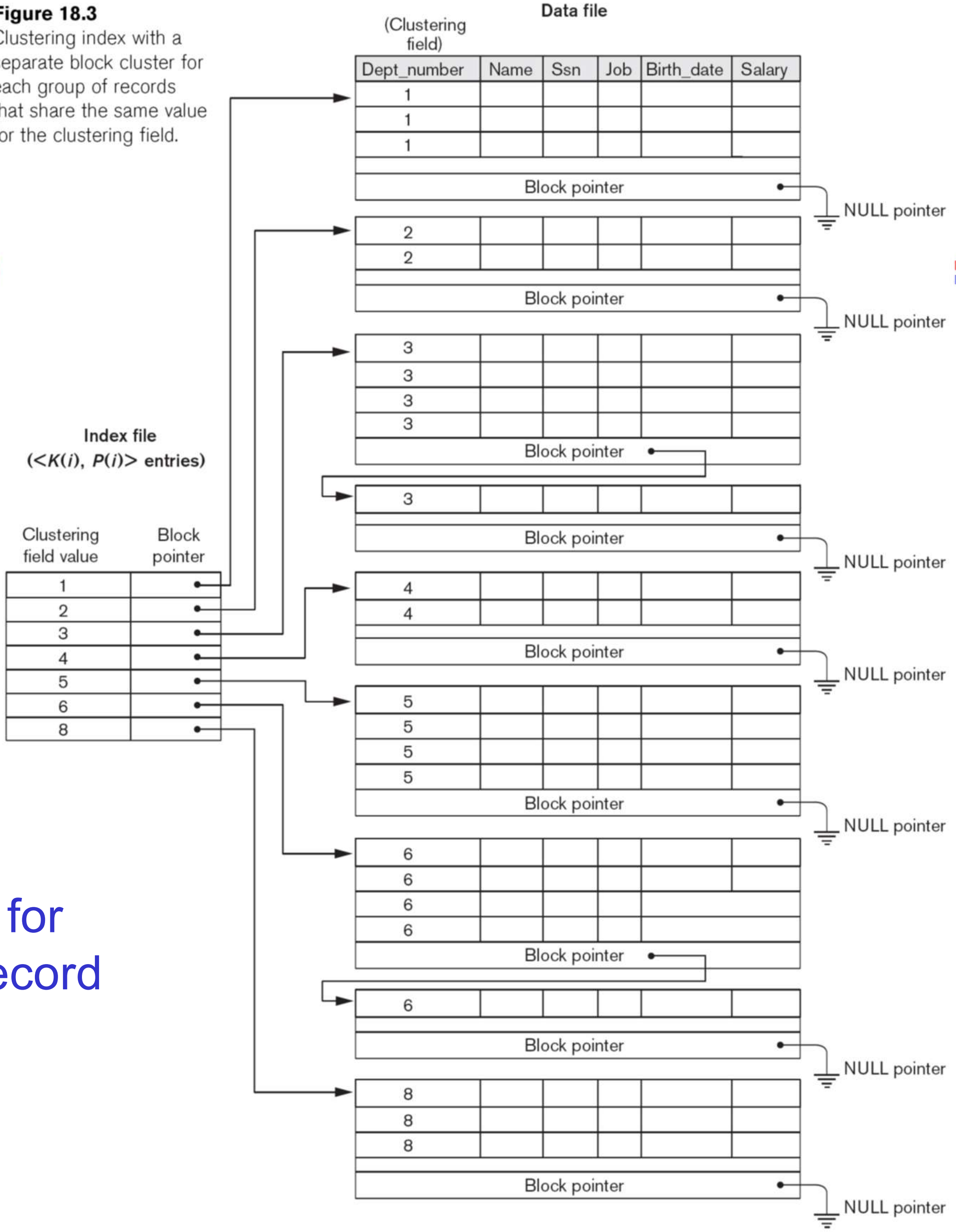


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Another Clustering Index Example

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



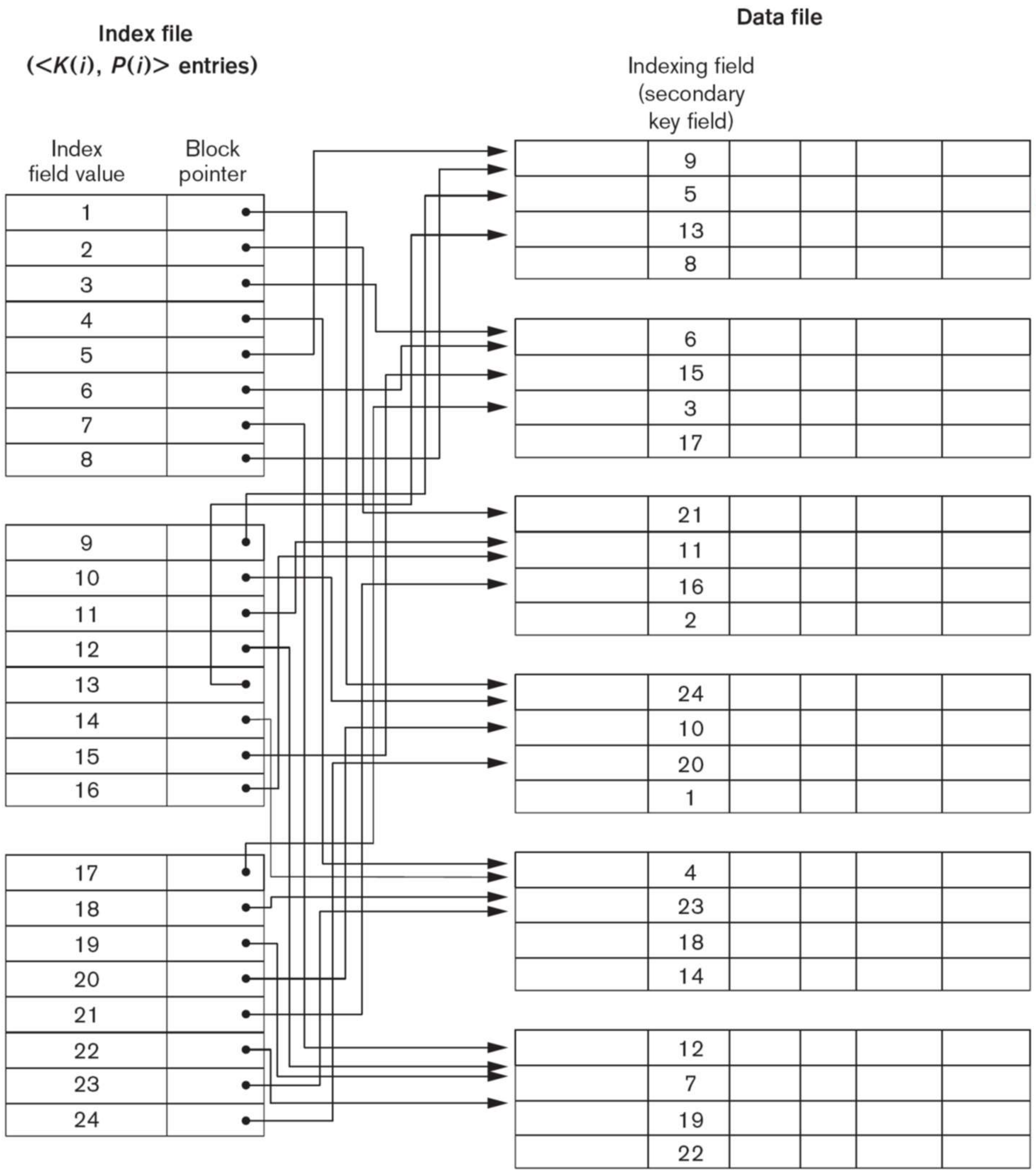
Reserving the whole block for handling overflow due to record insertion (anchoring block)

Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a secondary way to access the data file for which *some primary access already exists*
  - There can be *several* secondary indexes (and hence, indexing fields) for the same file
  - The secondary index may be on a key field or a non-key field but non-ordering field
  - The index is an ordered file with two fields
    - An indexing field value + a block pointer or a record pointer
  - Normally secondary index is *dense index*
    - Each record has one pointer

Secondary Index on Key Field

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



Secondary Index on non-Key Field

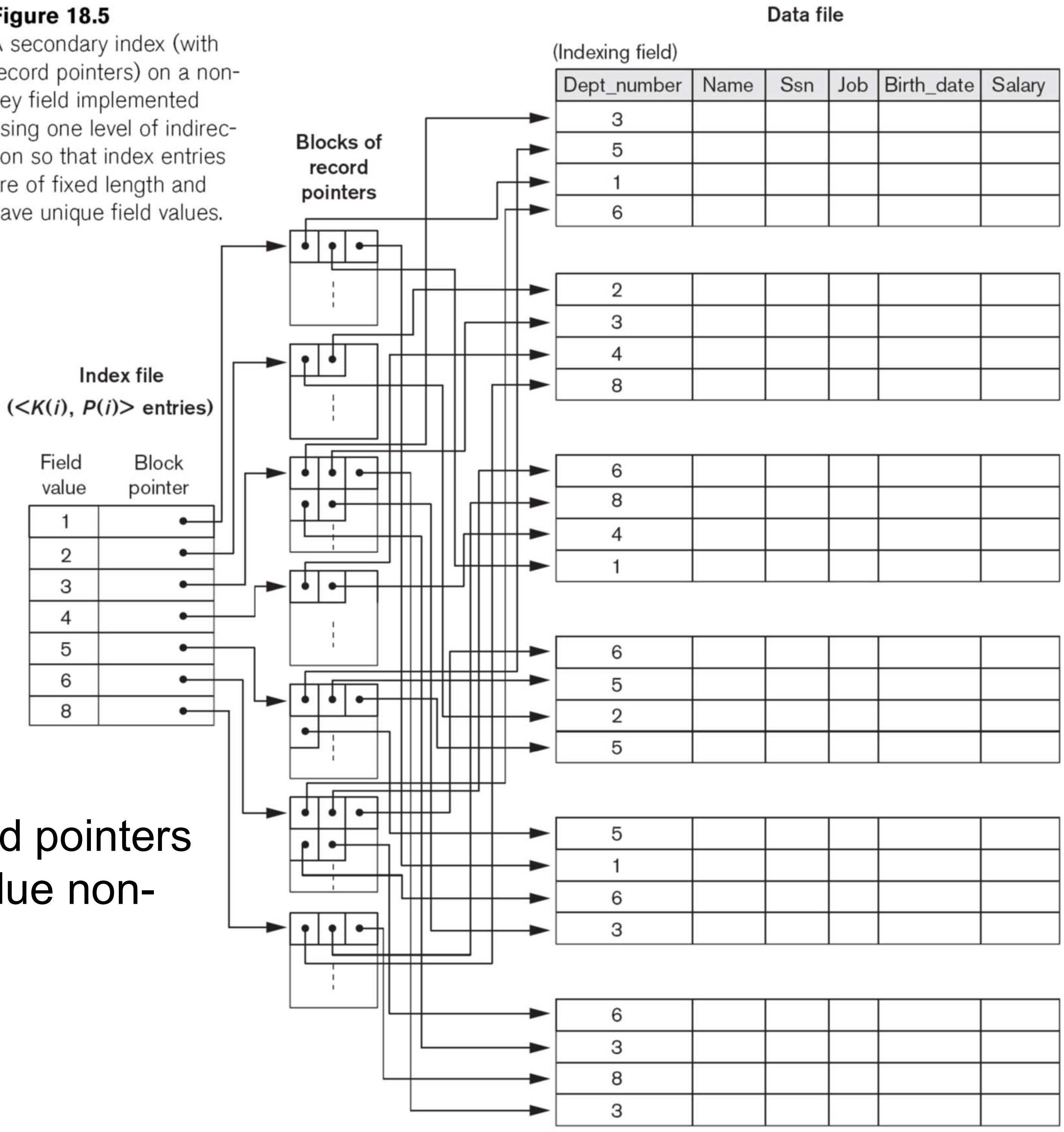


Figure 18.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

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 on a non-ordering key field (length 9 bytes)
  - Without the secondary index, a linear search on the data file requires 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. The block pointer is 6 bytes.
  - An index entry size R_I=(V+ P_R)=(9+6)=15 bytes
  - Index blocking factor Bfr_I= B div R_I= 1024 div 15= 68 entries/block
  - In a dense index, the total number of index entries = number of records = 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) = 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

Summary of single-level index

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

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

Table 17.2 Properties of Index Types

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

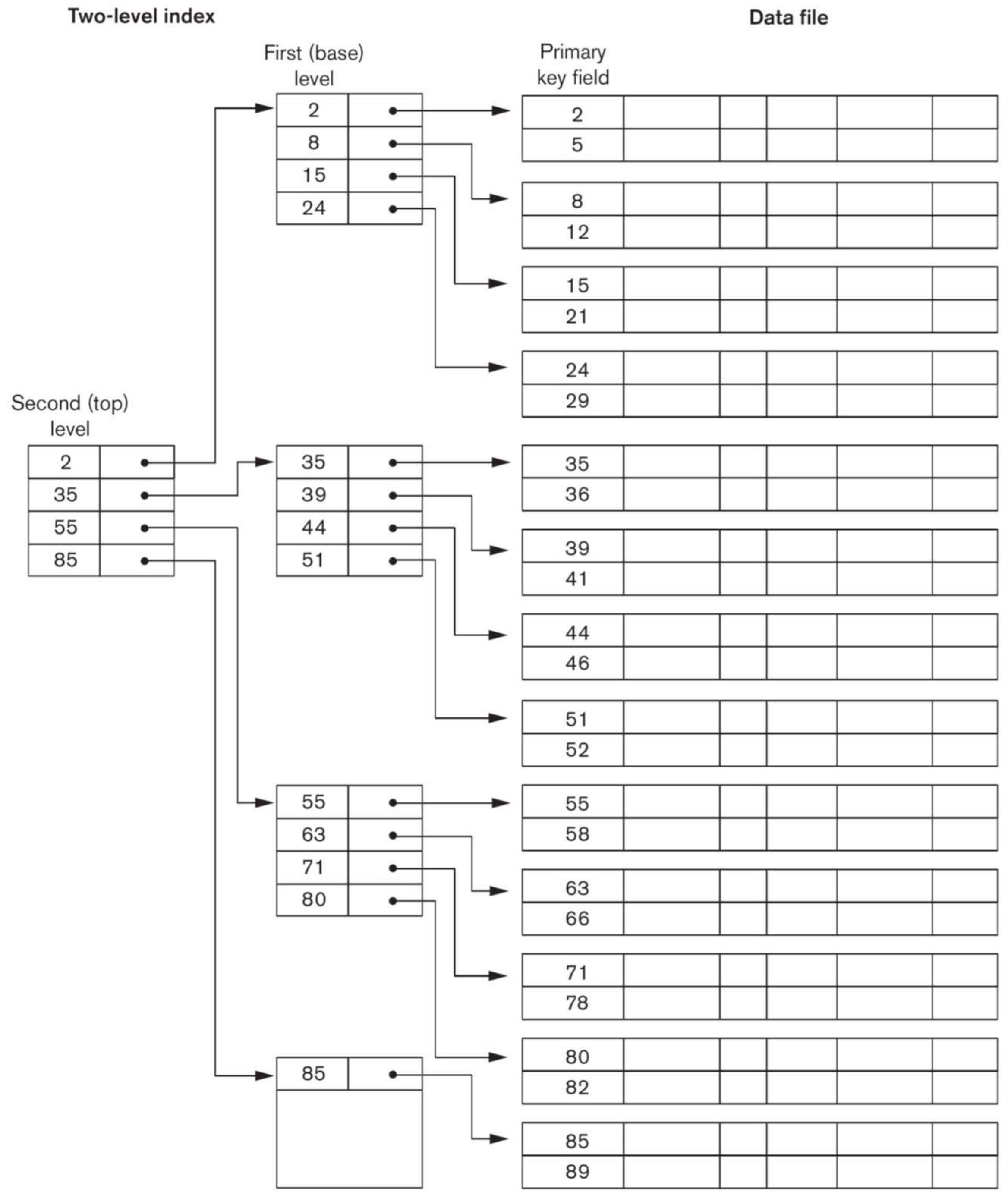
[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

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 single-level index (primary, secondary, clustering) as long as the single-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

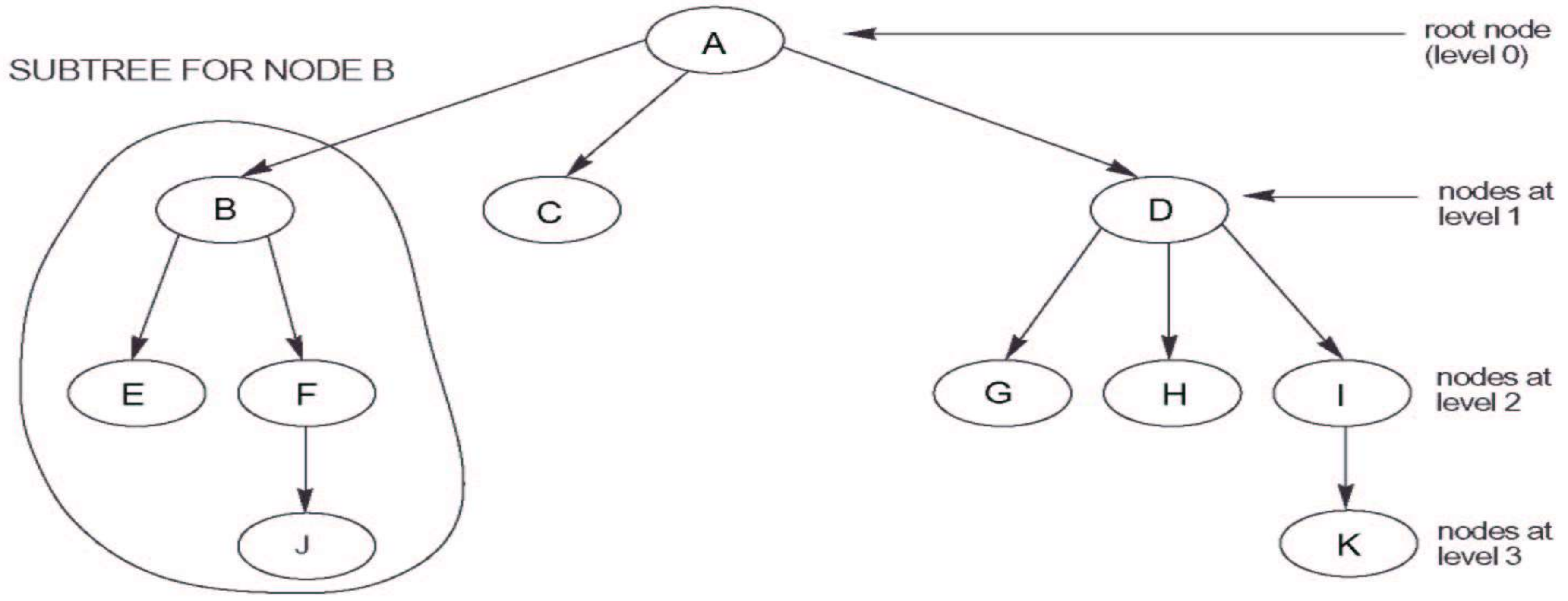
- Suppose the dense secondary index example is converted into a multi-level index
  - The index blocking factor (also called fan out, fo) bfr_i = 68 index entries per block
  - The number of first level blocks b1 =3000/68=442
  - The no. of second-level blocks b2 = b1/fo = 442/68 = 7 blocks
  - The no. 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 dynamic 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)

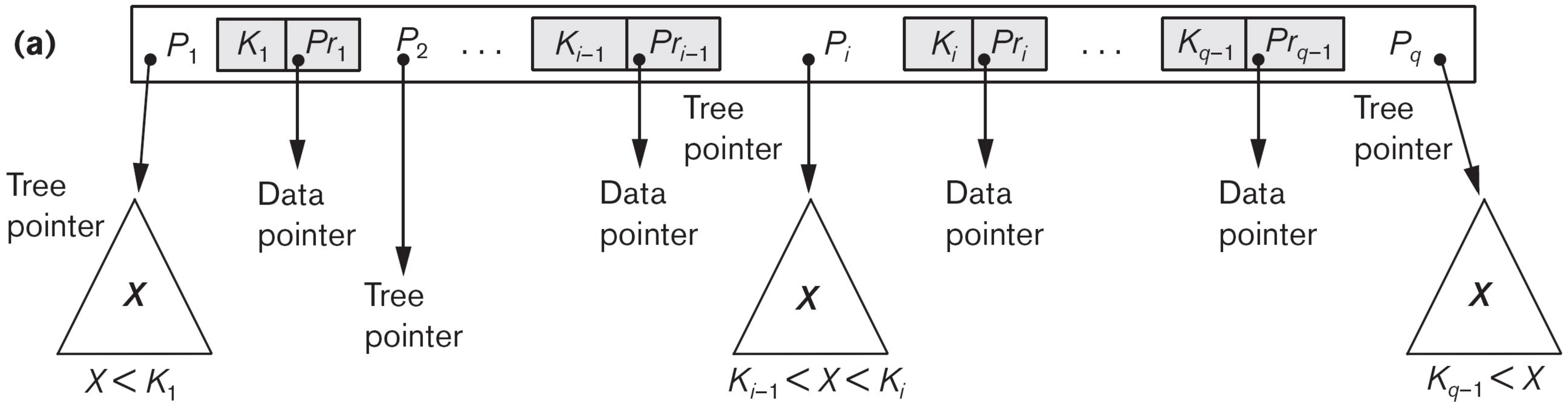
Difference between B-tree and B+-tree

- 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

B-tree Index

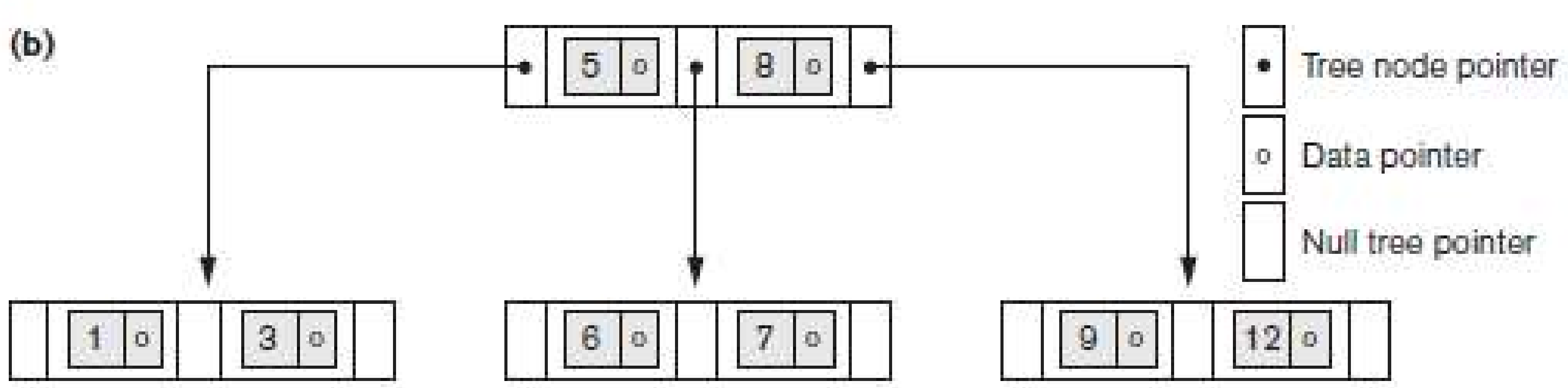
- B-tree organizes its nodes into a tree
- It is balanced (B) as all paths from the root to a leaf node have the same length
- Each internal node in a B-tree is of the form $<P_1, <K_1,Pr_1>, P_2, <K_2,Pr_2>, \ldots, <K_{q-1},Pr_{q-1}>,P_q>$
- Each P_i is a tree pointer to another node in the next level of B-tree
- Each Pr_i is a data pointer points to the record whose search key field value is equal to K_i

(a)



B-tree Index

- 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 q tree pointers, q-1 keys. q is the order of the tree. (full)
- Each node except for the root, has at least $\lceil q/2 \rceil$ tree pointers, $\lceil q/2 \rceil - 1$ keys. (half-full)
- 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

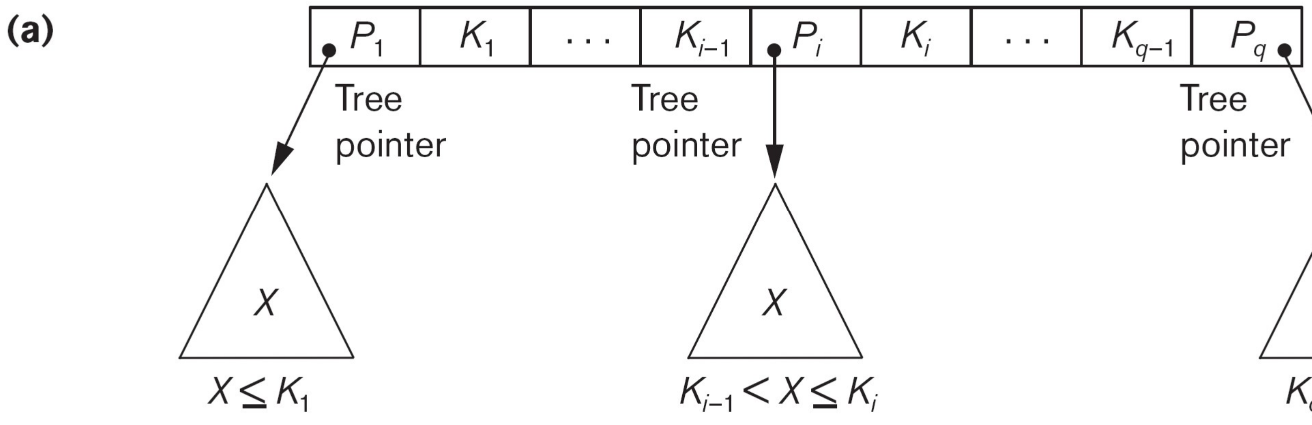


B^+-tree Index

- 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)

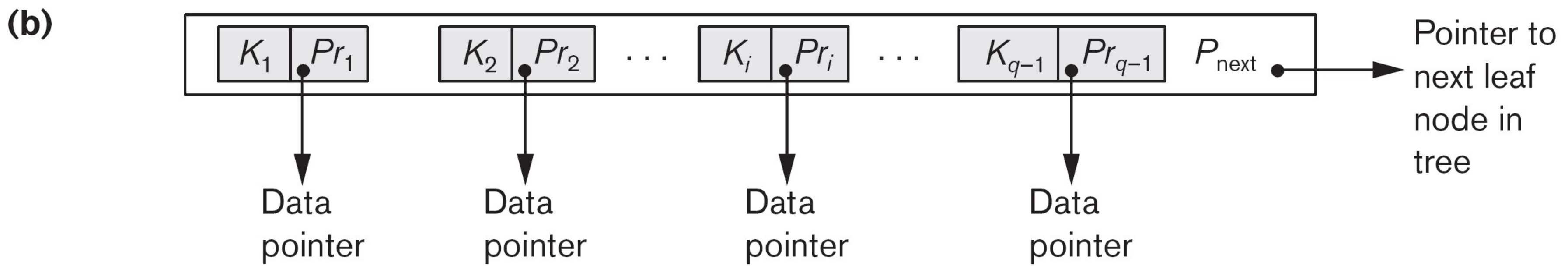
B^+-tree Index

- The structure of internal nodes of a B+-tree
  - Each internal node in a B^+-tree is of the form $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$ and P_i is a tree pointer.
  - 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 internal node has at most q tree pointers, q-1 keys. q is the order of the tree. (full)
  - Each internal node except root node has at least $\lceil q/2 \rceil$ tree pointers $\lceil q/2 \rceil$ -1 keys. (half-full)

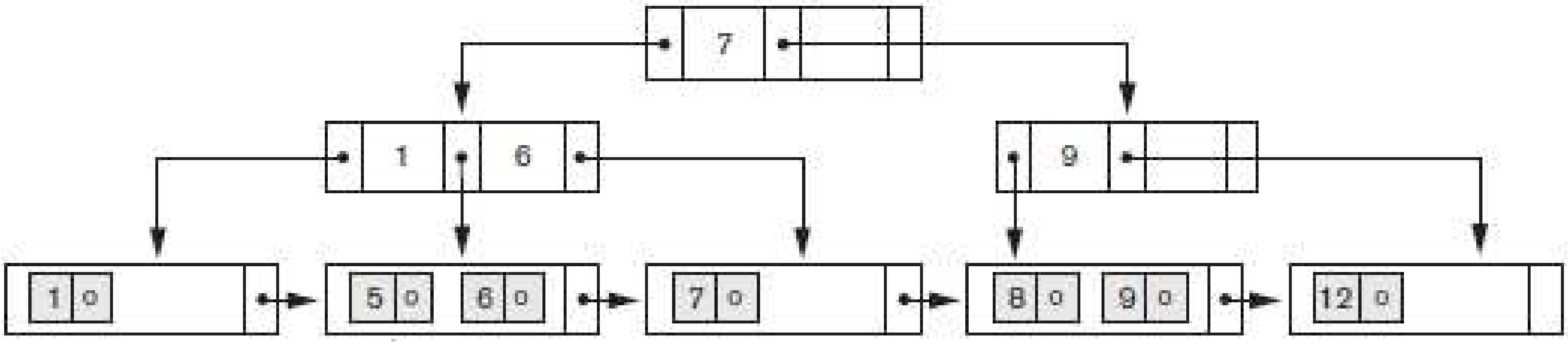


B^+-tree Index

- 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, $K_1 < K_2 < \ldots < K_{q-1}$
  - Each leaf node has at most q pointers, $q-1$ keys (full), and at least $\lceil q/2 \rceil$ pointers, $\lceil q/2 \rceil - 1$ keys (half-full), q is the order of leaf node.
  - All leaf nodes are at the same level



B$^+$-tree Index



B^+-tree Index Example

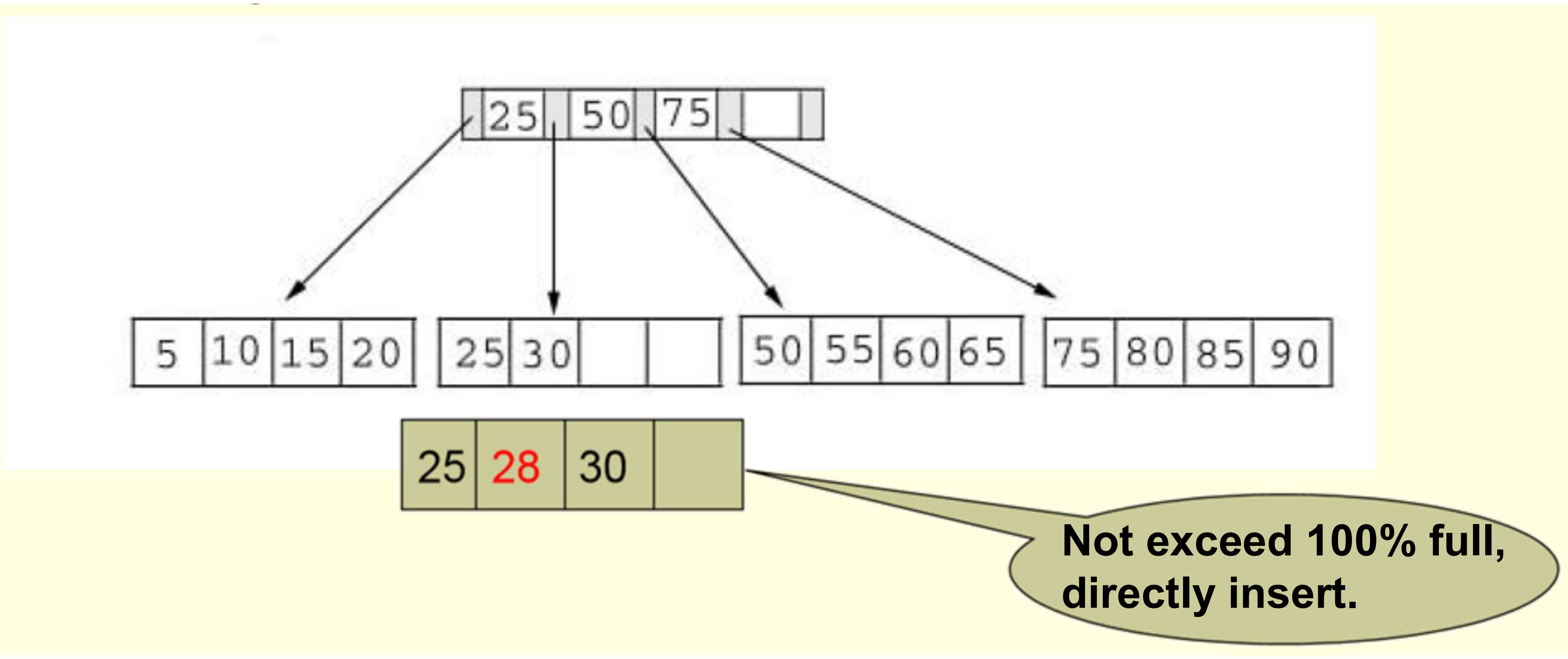
- Suppose the search key field is V = 9 bytes and the block size is B = 512 bytes, a data pointer is Pr = 8 bytes, and a tree pointer is P = 6 bytes
- An internal node of B^+-tree can have up to *q* tree pointers and *q-1* search field values. These must fit into a single block
  - $(q * P) + ((q – 1) * V) <= B$
  - $(q * 6) + (q – 1) * 9) <= 512$
  - $15q <= 512+9$
  - $q <= 34$
- The order q_{leaf} for the leaf node
  - $((q_{leaf} -1)* (Pr + V)) + P <= B$
  - $((q_{leaf} -1)* (8 + 9)) + 6 <= 512$
  - $17 * (q_{leaf} -1)<= 506$
  - $q_{leaf} <= 30$

B^+-tree Index Insertion

- Step1: 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 by the middle key value, distributing the keys evenly between the two nodes, so each node is half full.
    - (Case 2a for leaf node) If the node is a leaf, take a copy of the middle key value, and repeat step 2 to insert it into the parent node.
    - (Case 2b for internal node) If the node is a non-leaf, exclude the middle key value during the split and repeat step 2 to insert this excluded value into the parent node.

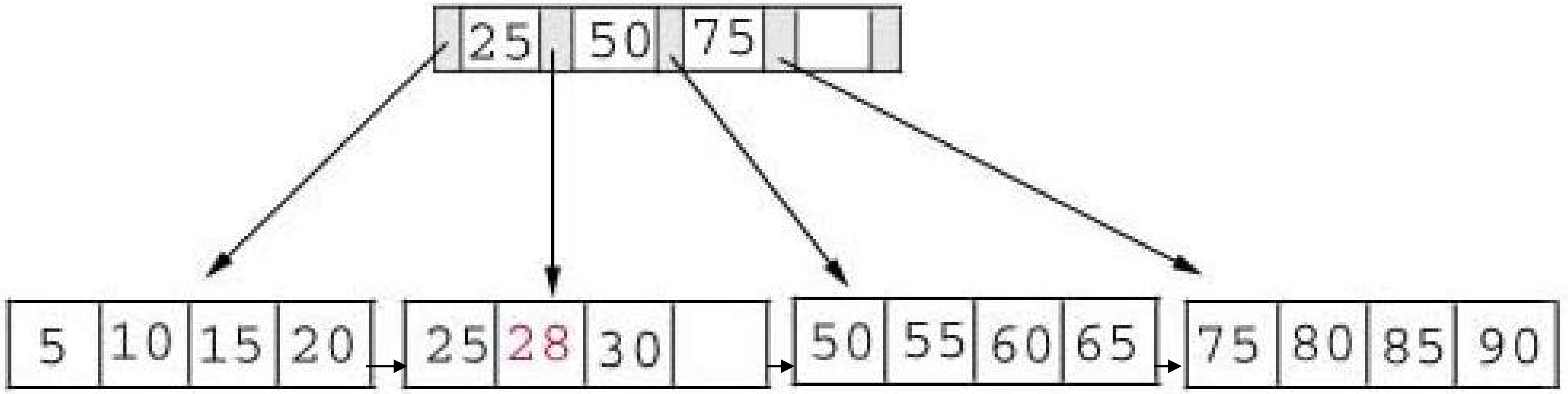
B⁺-tree Index Insertion

Example #1: insert 28 into the below tree



B^+-tree Index Insertion

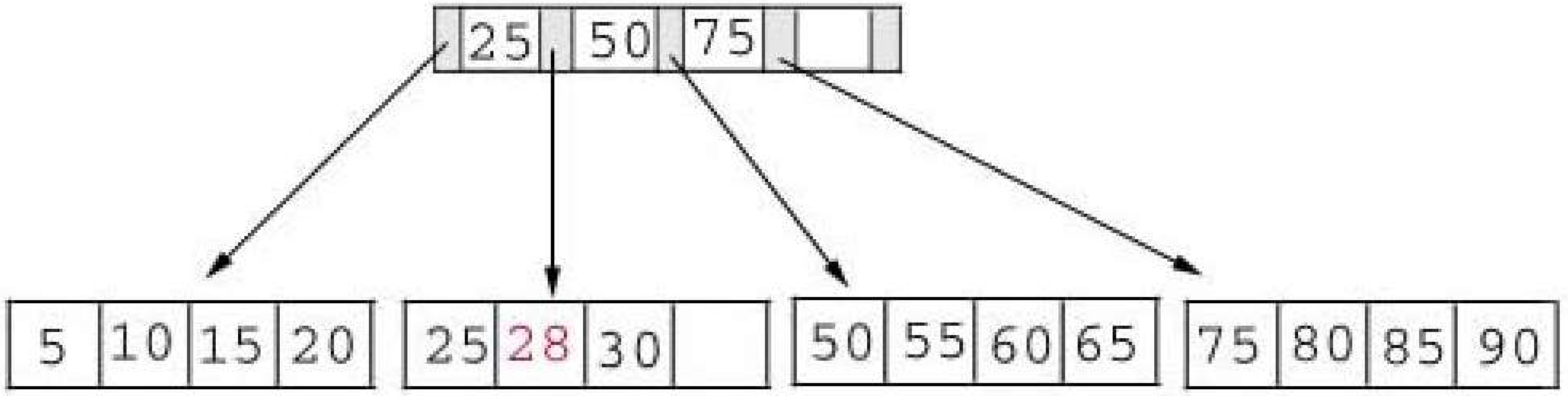
Example #1: insert 28 into the below tree



Result: insert 28 into the appropriate leaf node

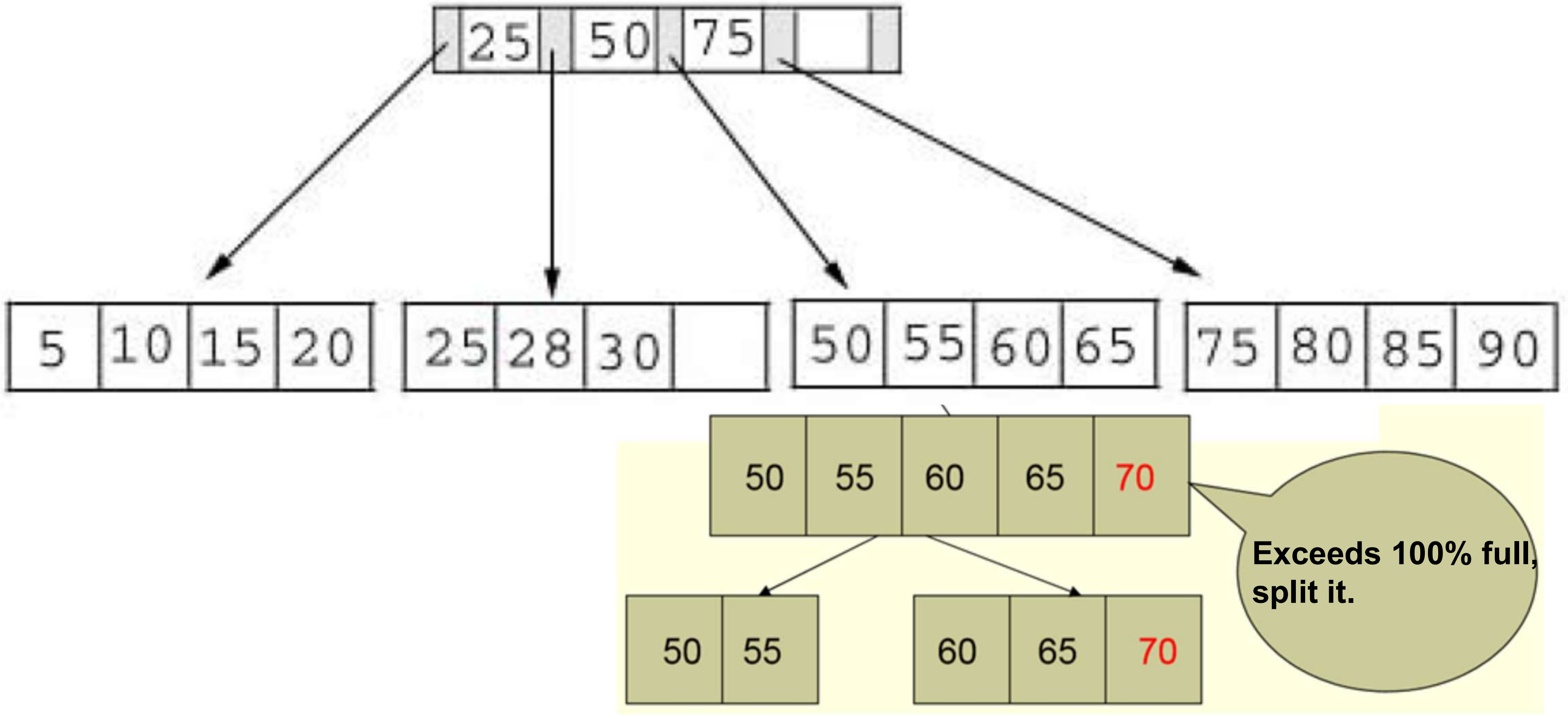
B+-tree Index Insertion

Example #2: insert 70 into the below tree



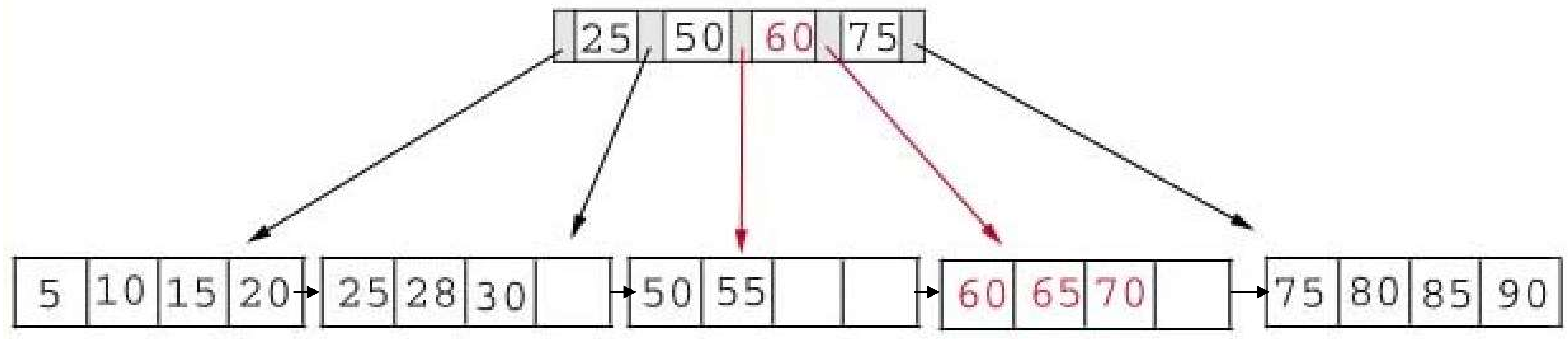
B^+-tree Index Insertion

Example #2: insert 70 into the below tree



B^+-tree Index Insertion

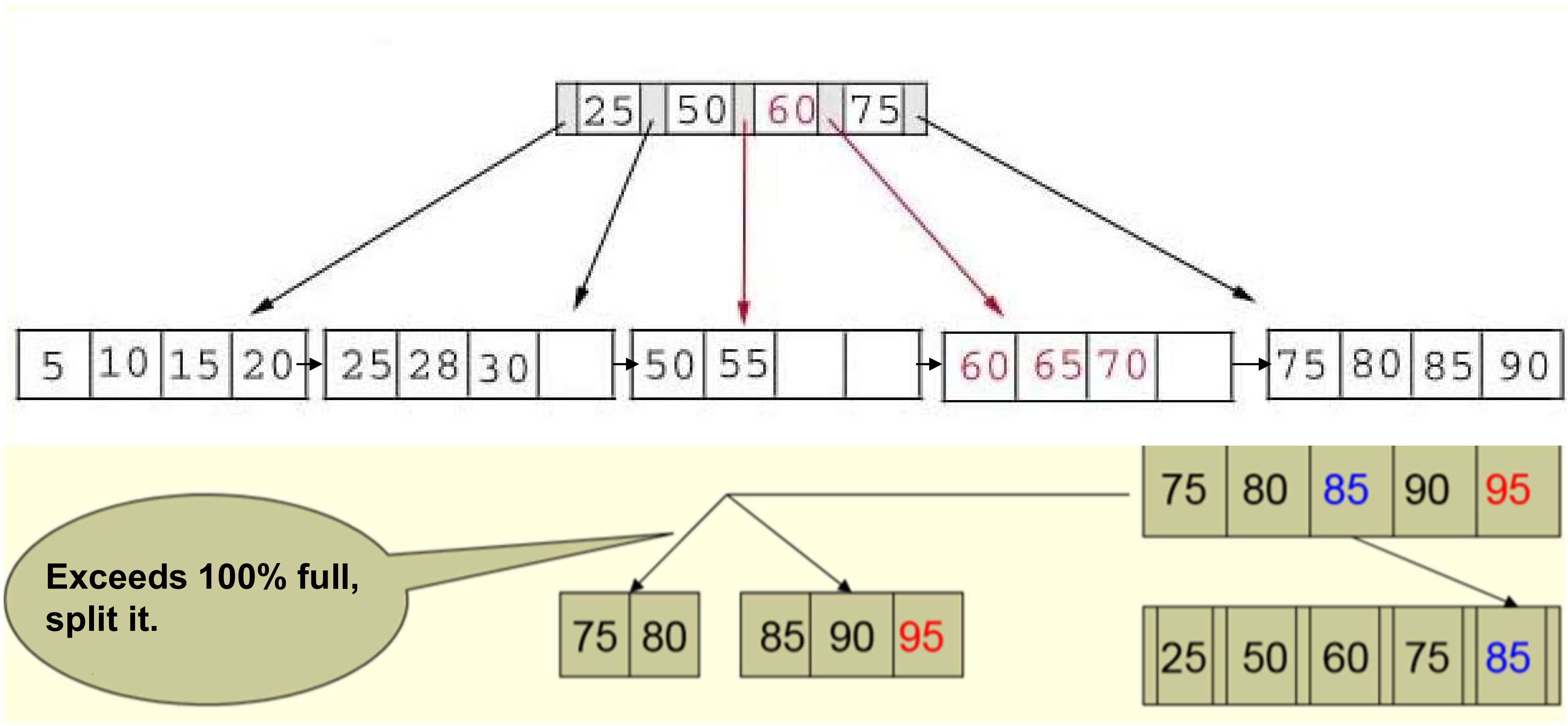
Example #2: insert 70 into the below tree



Result: split leaf node from the middle, chose the middle key 60, and place it into the parent node.

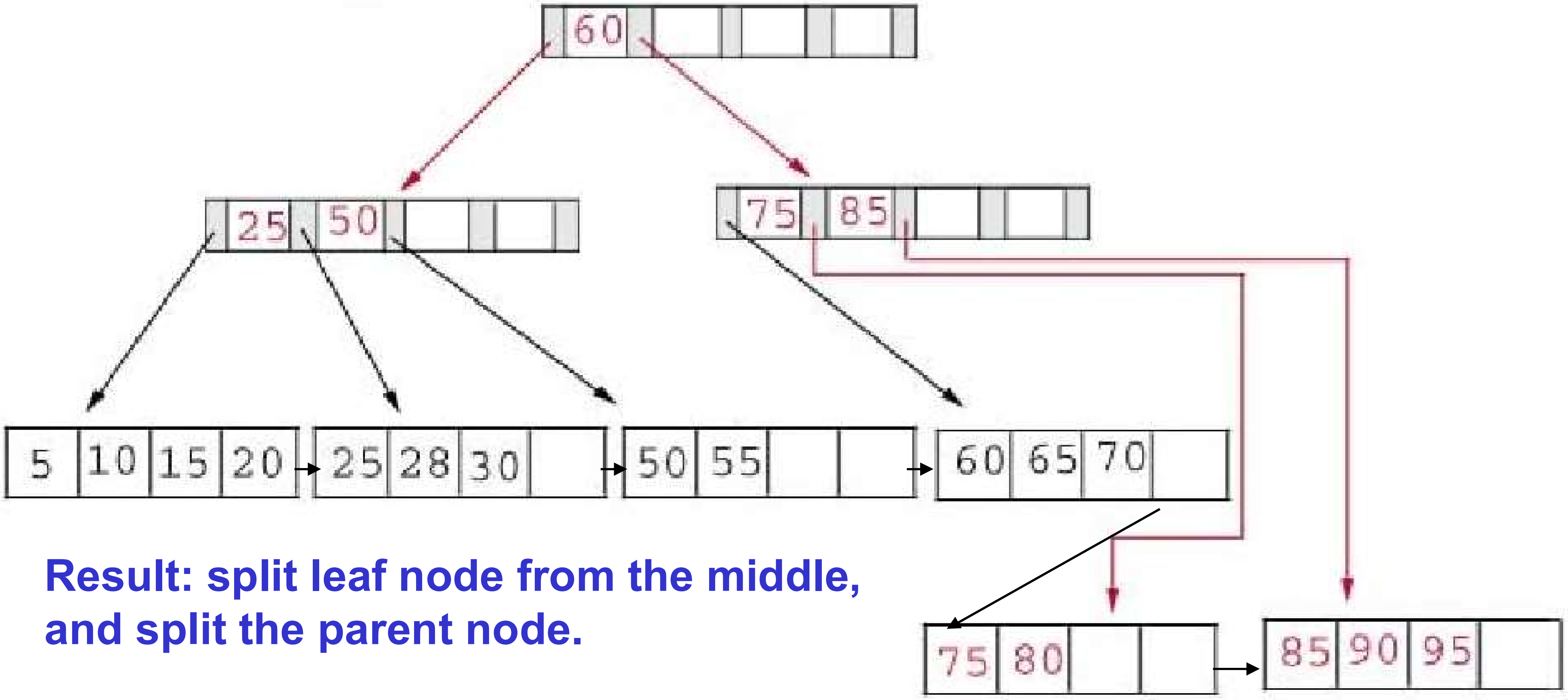
B+-tree Index Insertion

Example #3: insert 95 into the below tree



B^+-tree Index Insertion

Example #3: insert 95 into the below tree



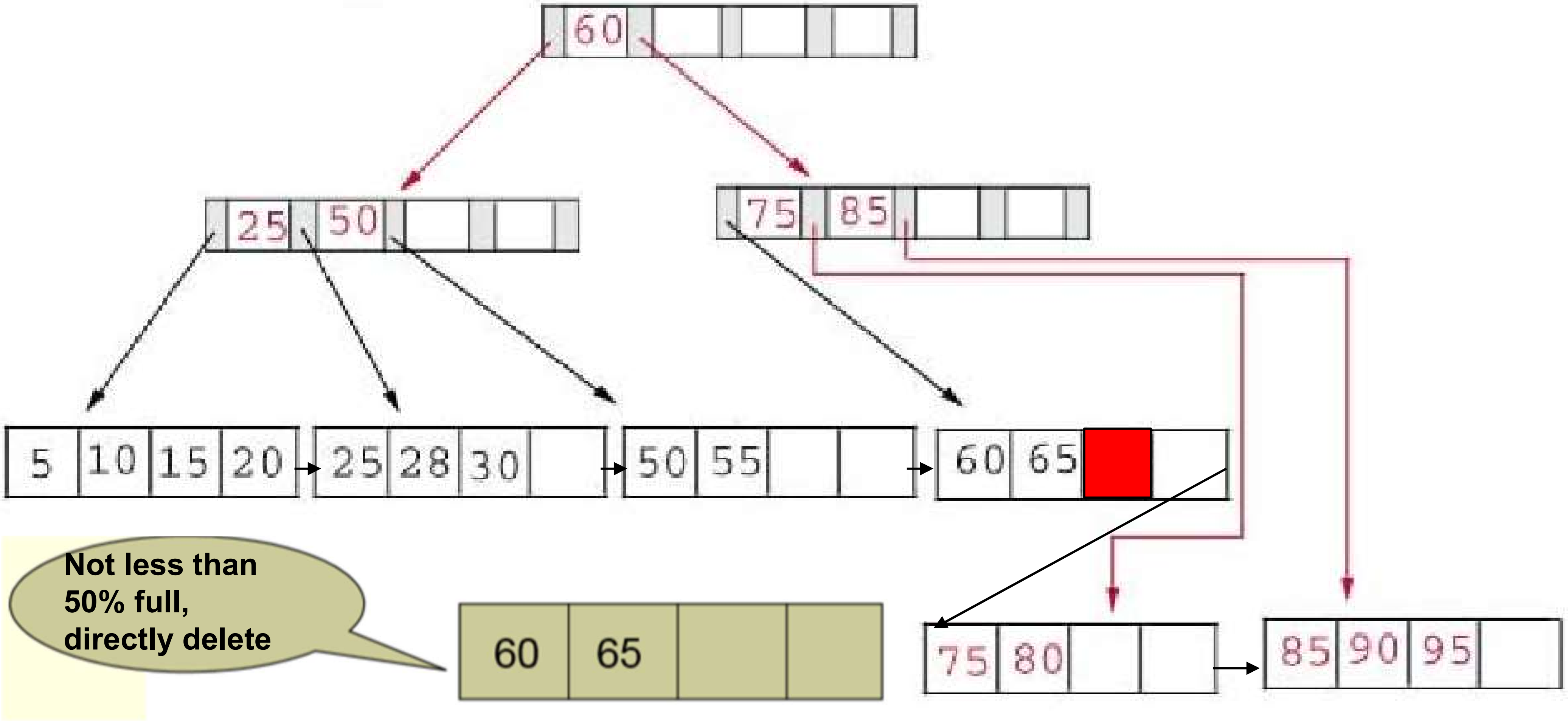
Result: split leaf node from the middle, and split the parent node.

B^+-tree Index Deletion

- Step1: Descend to the leaf node where the key fits
- Step 2: Remove the required key and associated reference from the node.
  - (Case 1): If the node still has half-full keys, repair the keys in parent node to reflect the change in child node if necessary and stop.
  - (Case 2): If the node is less than half-full, but left or right sibling node is more than half-full, redistribute the keys between this node and its sibling. Repair the keys in the level above to represent that these nodes now have a different “split point” between them.
  - (Case 3): If the node is less than half-full, and left and right sibling node are just half-full, merge the node with its sibling. Repeat step 2 to delete the unnecessary key in its parent.

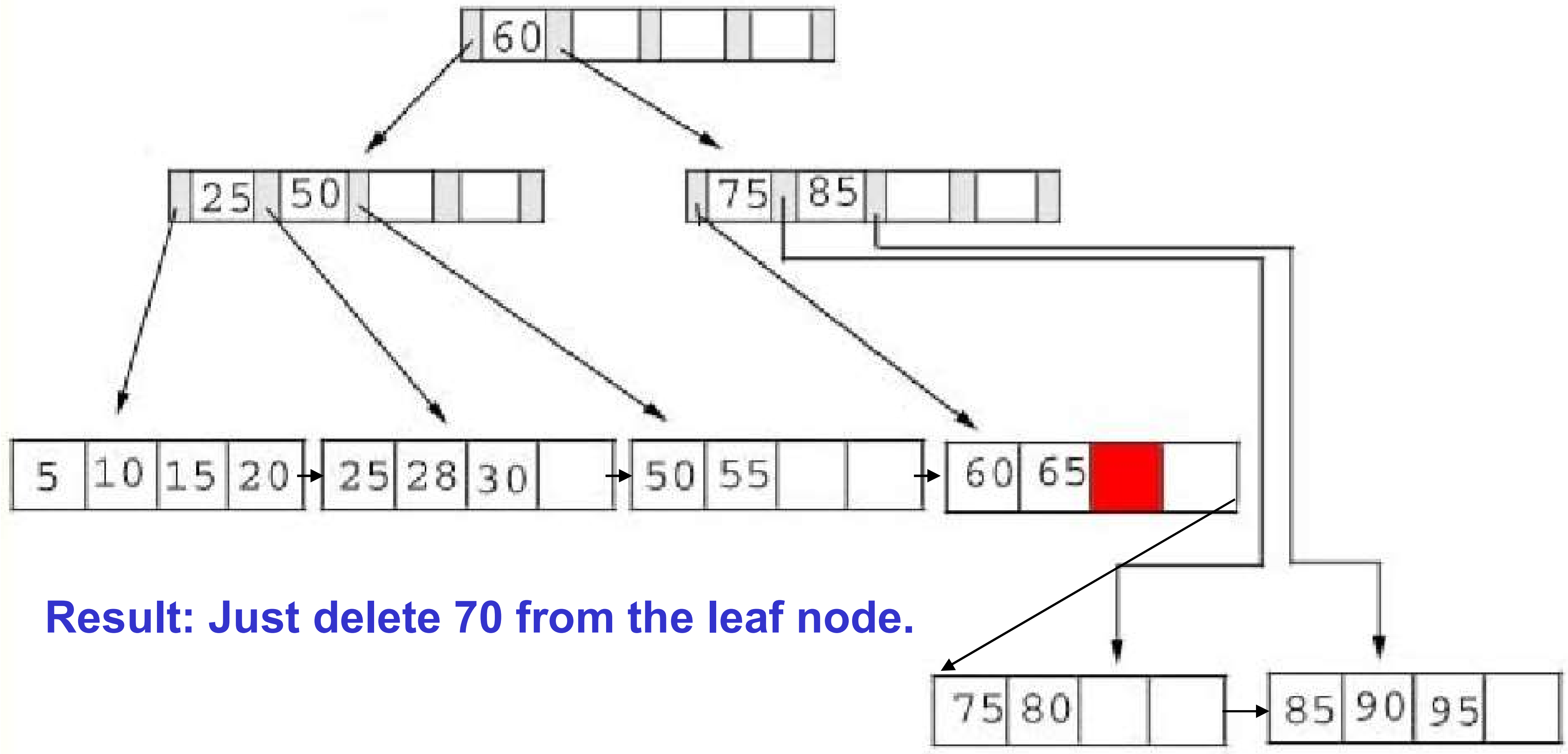
B^+-tree Index Deletion

Example #1: Delete 70 from the below tree



B^+-tree Index Deletion

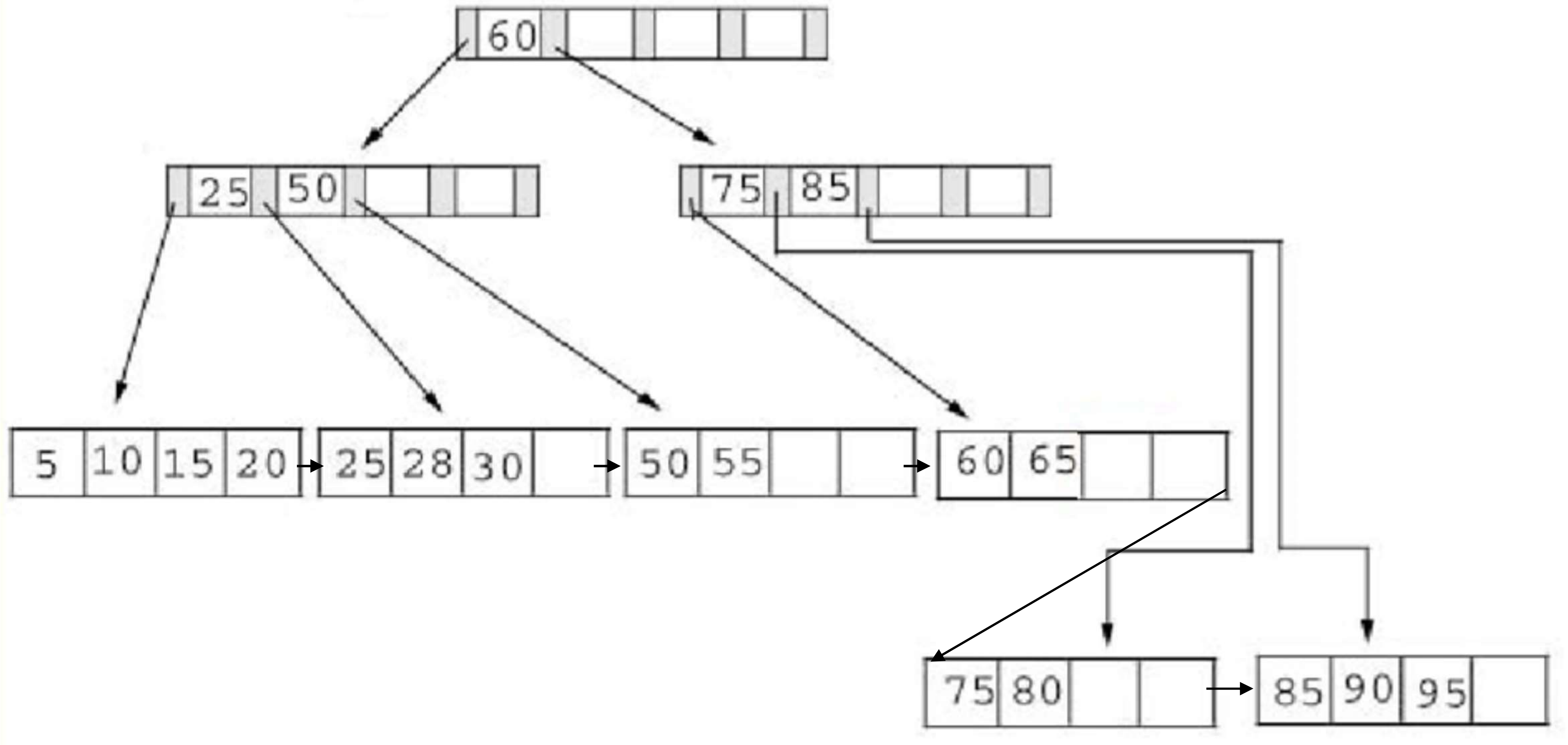
Example #1: Delete 70 from the below tree



Result: Just delete 70 from the leaf node.

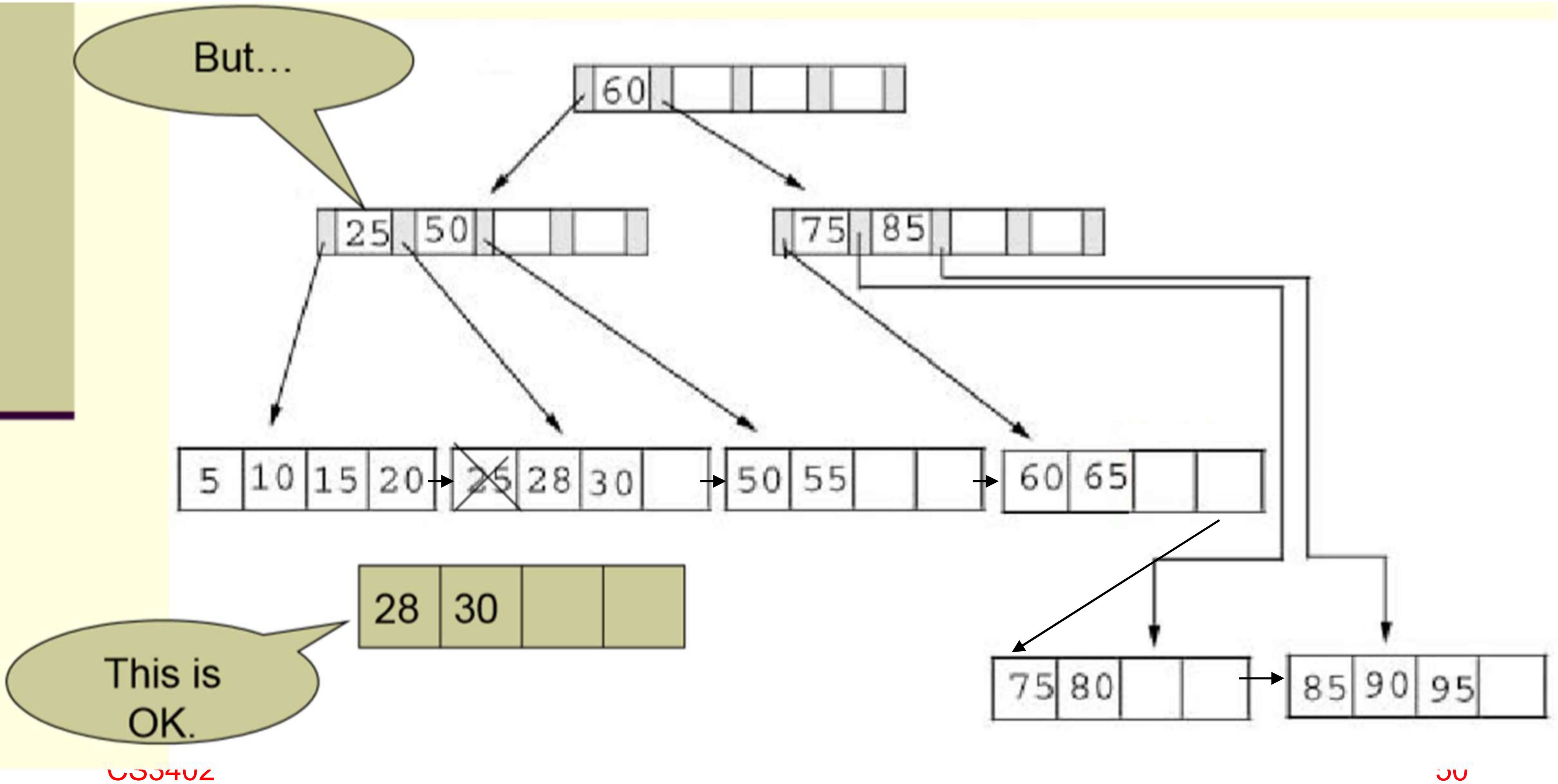
B+-tree Index Deletion

Example #2: Delete 25 from the below tree



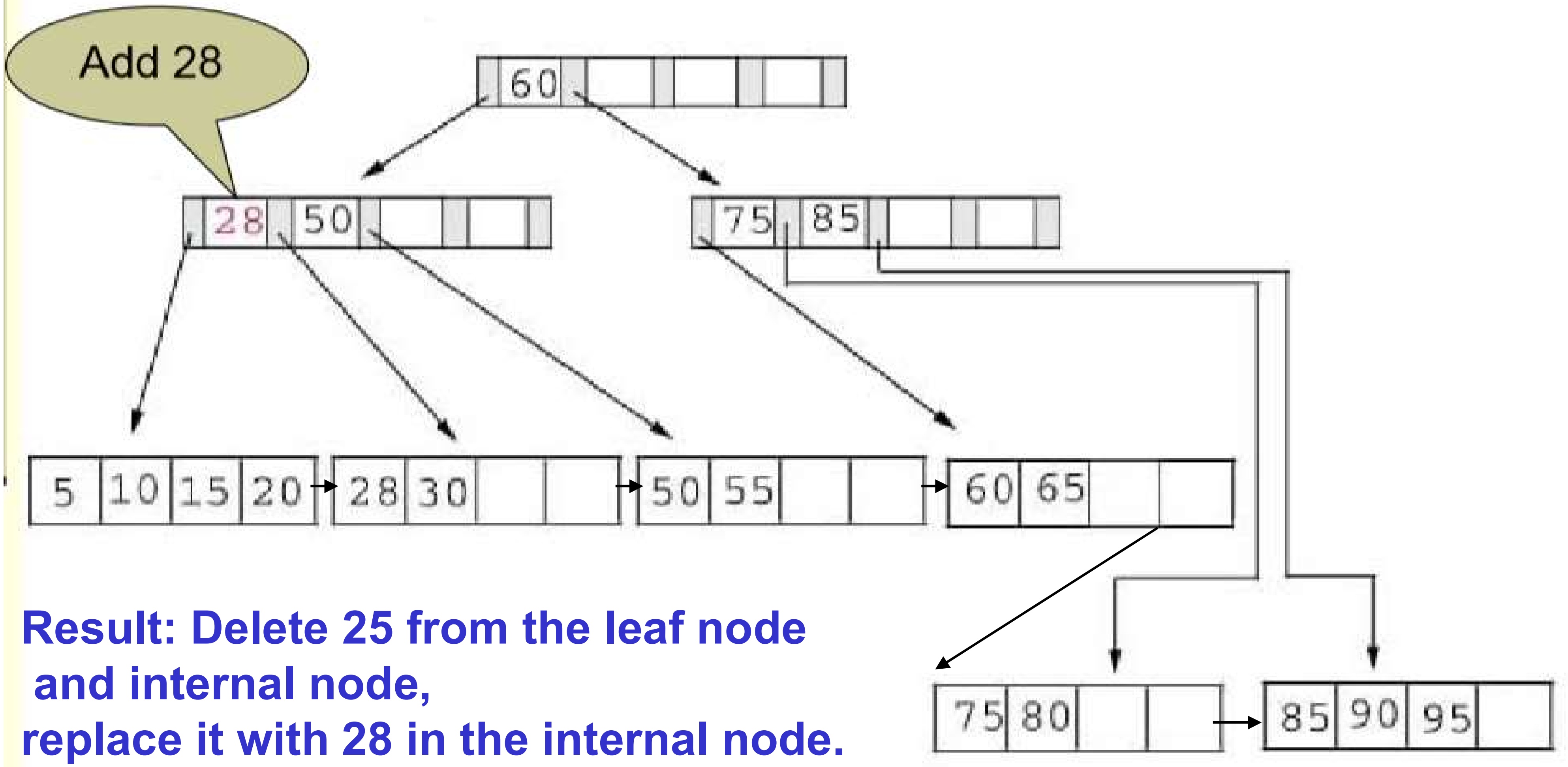
B+-tree Index Deletion

Example #2: Delete 25 from the below tree



B+-tree Index Deletion

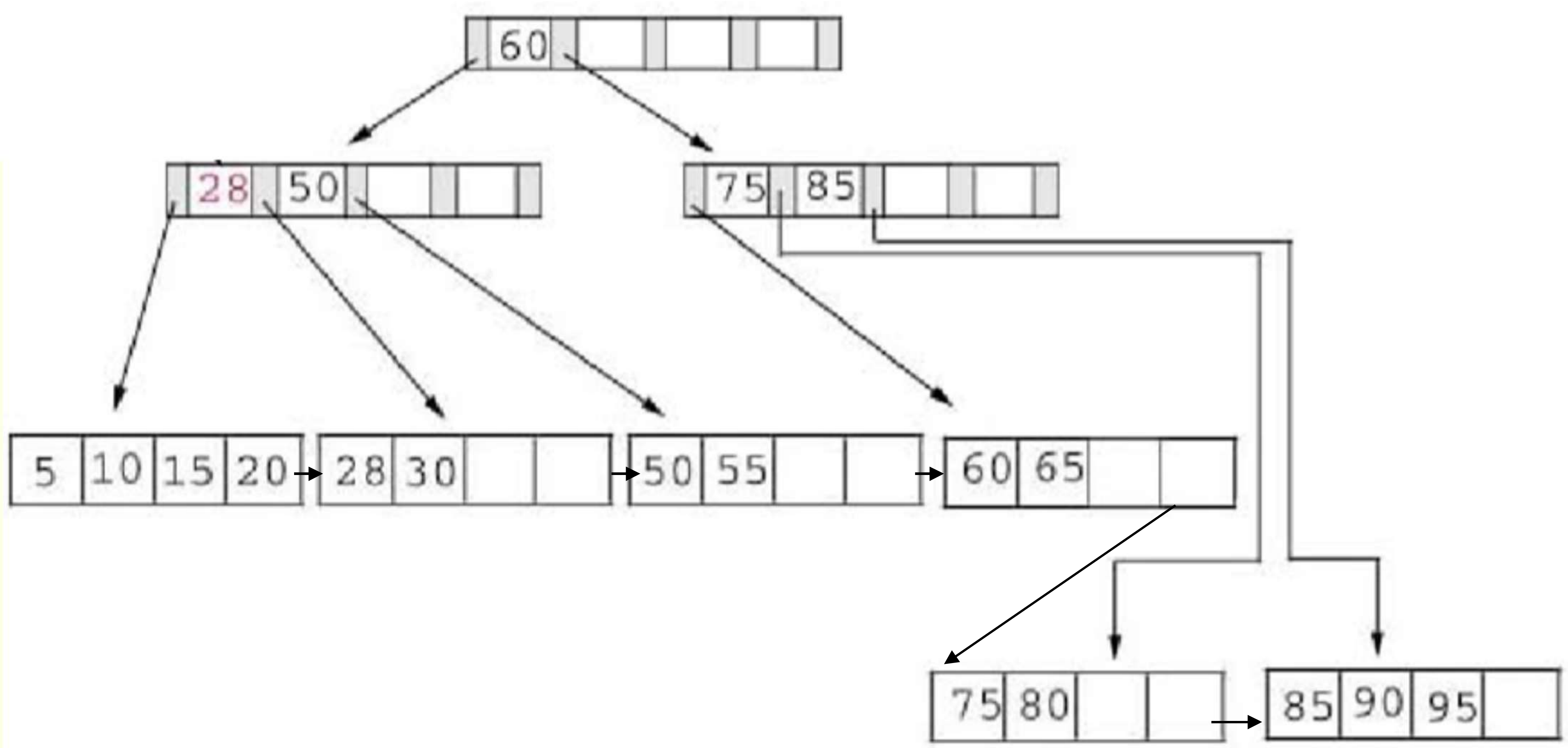
Example #2: Delete 25 from the below tree



Result: Delete 25 from the leaf node and internal node, replace it with 28 in the internal node.

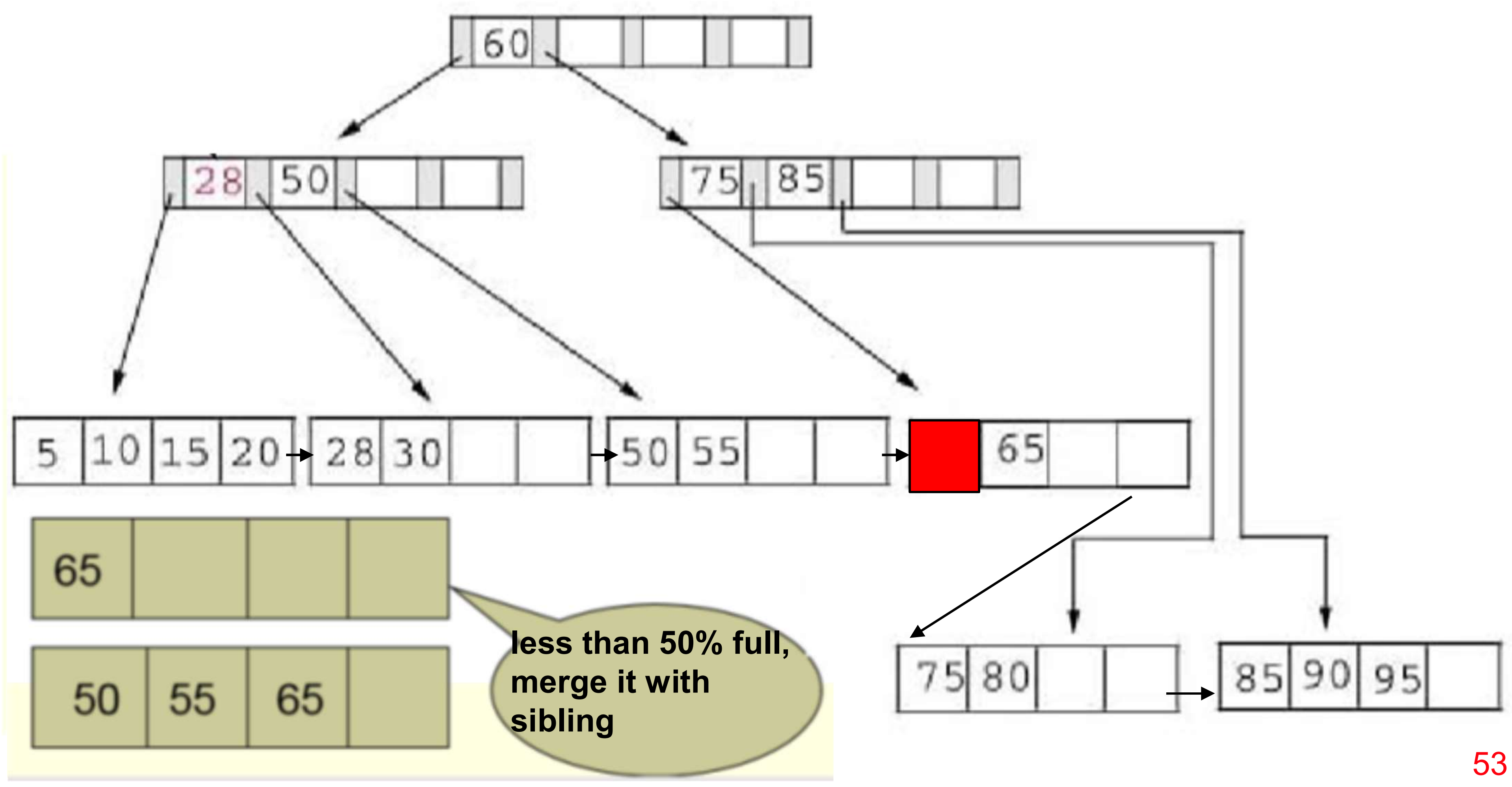
B+-tree Index Deletion

Example #3: Delete 60 from the below tree



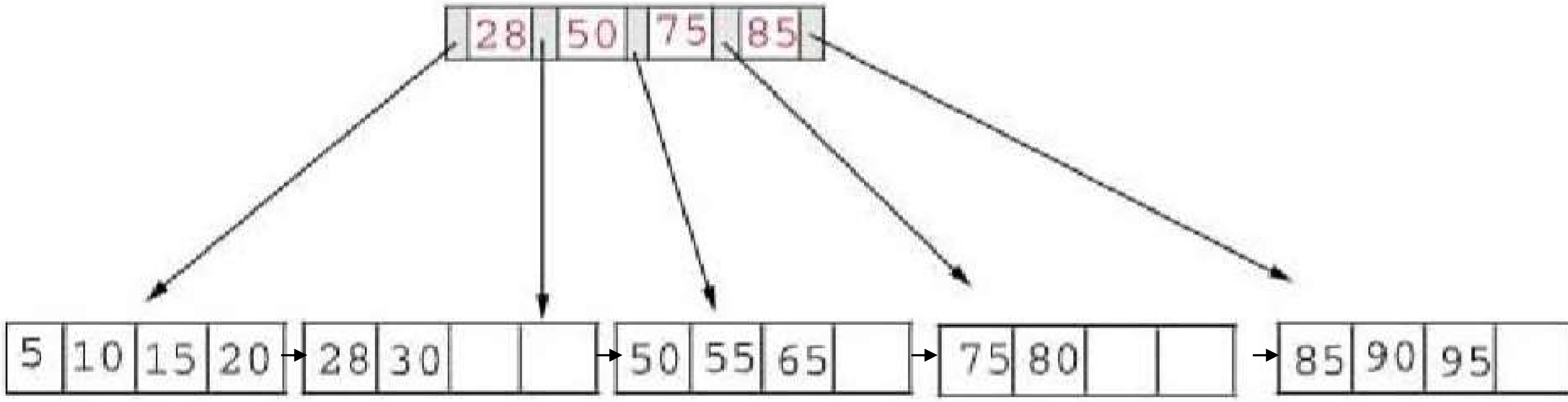
B⁺-tree Index Deletion

Example #3: Delete 60 from the below tree



B+-tree Index Deletion

Example #3: Delete 60 from the below tree



Result: Delete 60 from the leaf node, combine leaf nodes and then internal nodes

Defining Index in SQL

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ])
[ CLUSTER ];
```

```
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER;
```

- The optional keyword CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute.
- The optional keyword UNIQUE is used when we constrain index field to be the key field.
- The value for <order> can be either ASC (ascending) or DESC (descending). The default is ASC.

- DROP INDEX <index name>;

References

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