COMP 3311
DATABASE MANAGEMENT SYSTEMS

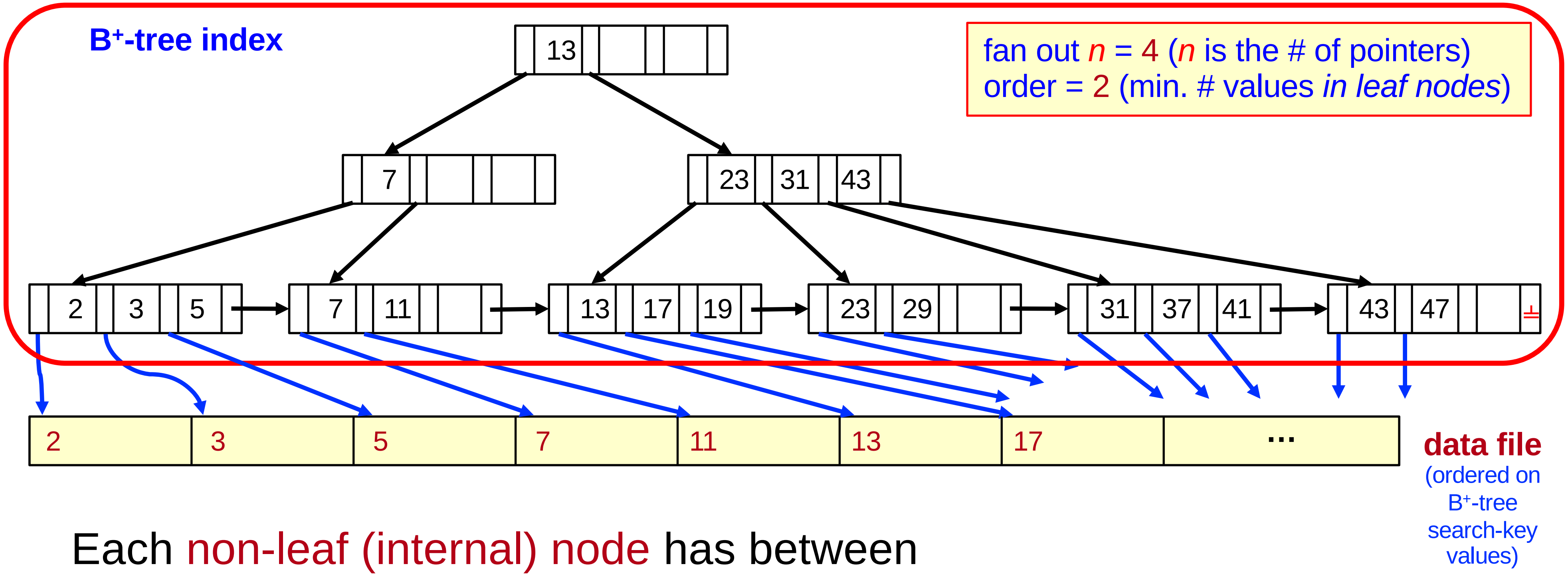
LECTURE 13
INDEXING: B+-TREE INDEX

B+-TREE INDEX

- **Disadvantage of index-sequential files**
  - Performance degrades as the file grows due to overflow pages.
  - Periodic reorganization of the entire file is required.
- **Advantage of B+-tree index files**
  - Balanced tree ⟹ every path from root to leaves is the same length.
  - Automatically reorganizes itself with small, local changes in the presence of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- **Disadvantage of B+-trees**
  - Extra insertion, deletion and space overhead.

The advantages of B+-trees far outweigh their disadvantages. They are used extensively in all commercial products.

B+-TREE: STRUCTURE



fan out n = 4 (n is the # of pointers)
order = 2 (min. # values *in leaf nodes*)

Each non-leaf (internal) node has between

- $\lceil n/2 \rceil$ and n pointers ($\lceil n/2 \rceil$-1 (min) and n-1 (max) values).
  - **In the example:** between 2 and 4 pointers (1 and 3 values).

Special cases for root:
if non-leaf - min 1 value.
if leaf - min 0 values.

Each leaf node is at least half full, i.e., has between

- $\lceil (n\text{-}1)/2 \rceil$ +1 and n pointers, e.g., ($\lceil (n\text{-}1)/2 \rceil$ (min) and n-1 (max) values).
  - **In the example:** between 3 and 4 pointers (2 and 3 values).

B+-TREE: STRUCTURE (CONT'D)

- **Balanced tree**
  - All paths from the root node to the leaf nodes are the same length.
- **Fan-out**
  - The maximum number of pointers/children in each node, denoted n.
- **B+-tree order**
  - The value $\lceil (n-1)/2 \rceil$ is called the order and corresponds to the minimum number of values in a leaf node.
- **Special cases**
  - If the root is not a leaf, it has at least 2 children (i.e., it has at least 1 value).
  - If the root is a leaf (i.e., there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-TREE: NON-LEAF (INTERNAL) NODES

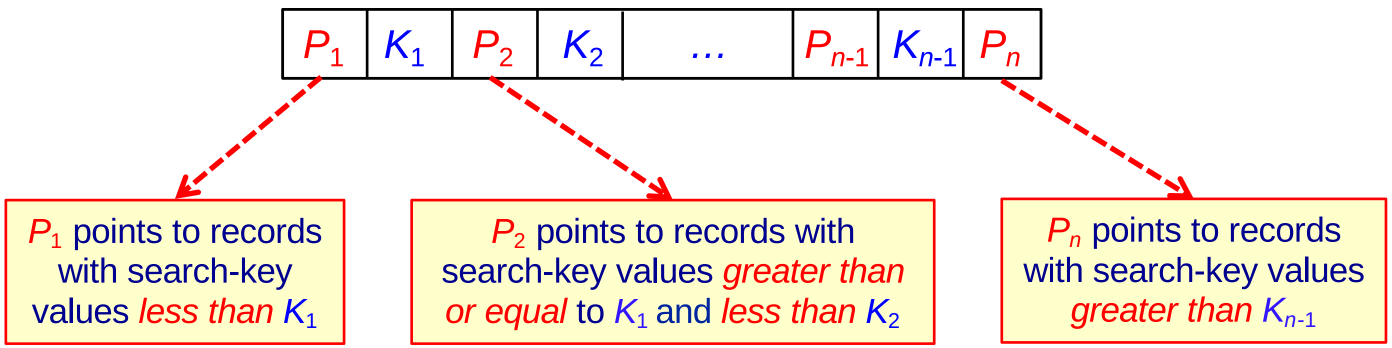
- Non-leaf (internal) nodes form a multi-level, sparse index on the leaf nodes (i.e., only some search-key values are present).

- For a non-leaf node with *n* pointers:

First pointer P_1: All the search-key values in the subtree to which P_1 points are *less than* K_1.

Internal pointer: For $2 \leq i \leq n-1$, all the search-key values in the subtree to which P_i points have values *greater than or equal to* K_{i-1} and *less than* K_i.

Last pointer P_n: All the search-key values in the subtree to which P_n points are *greater than* K_{n-1}.

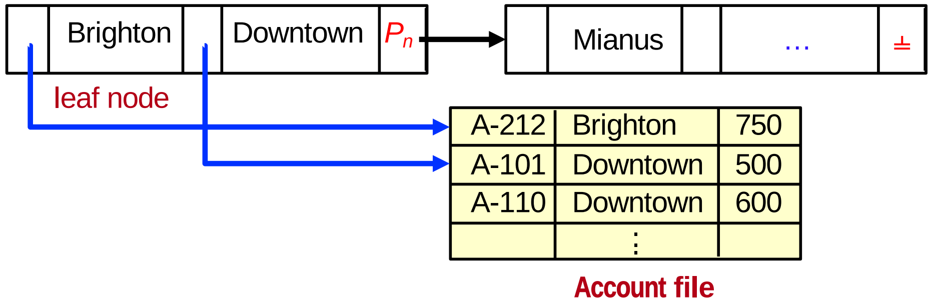


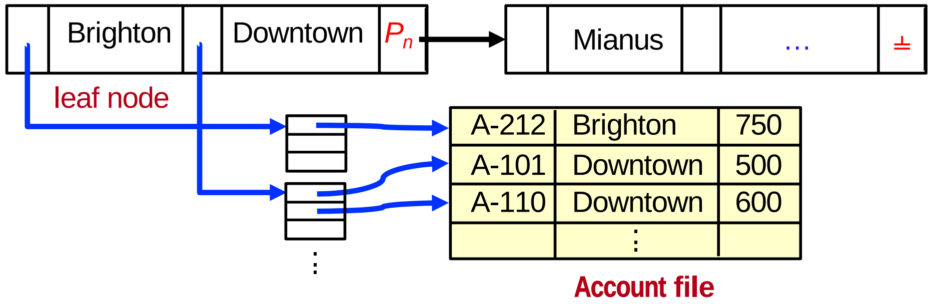
B⁺-TREE: LEAF NODES

- For $i = 1, 2, \ldots, n–1$, pointer P_i *either* points
  - to a file record with search-key value K_i,

    or

    to a "bucket" of pointers each of which points to a file record that has search-key value K_i.





- Search-key values in a node are kept in sorted order.
  - If L_i, and L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key value.
- The last pointer in a node, P_n, points to the next leaf node in the search-key order (right sibling node) or contains an end symbol.

B^+-TREE: OBSERVATIONS

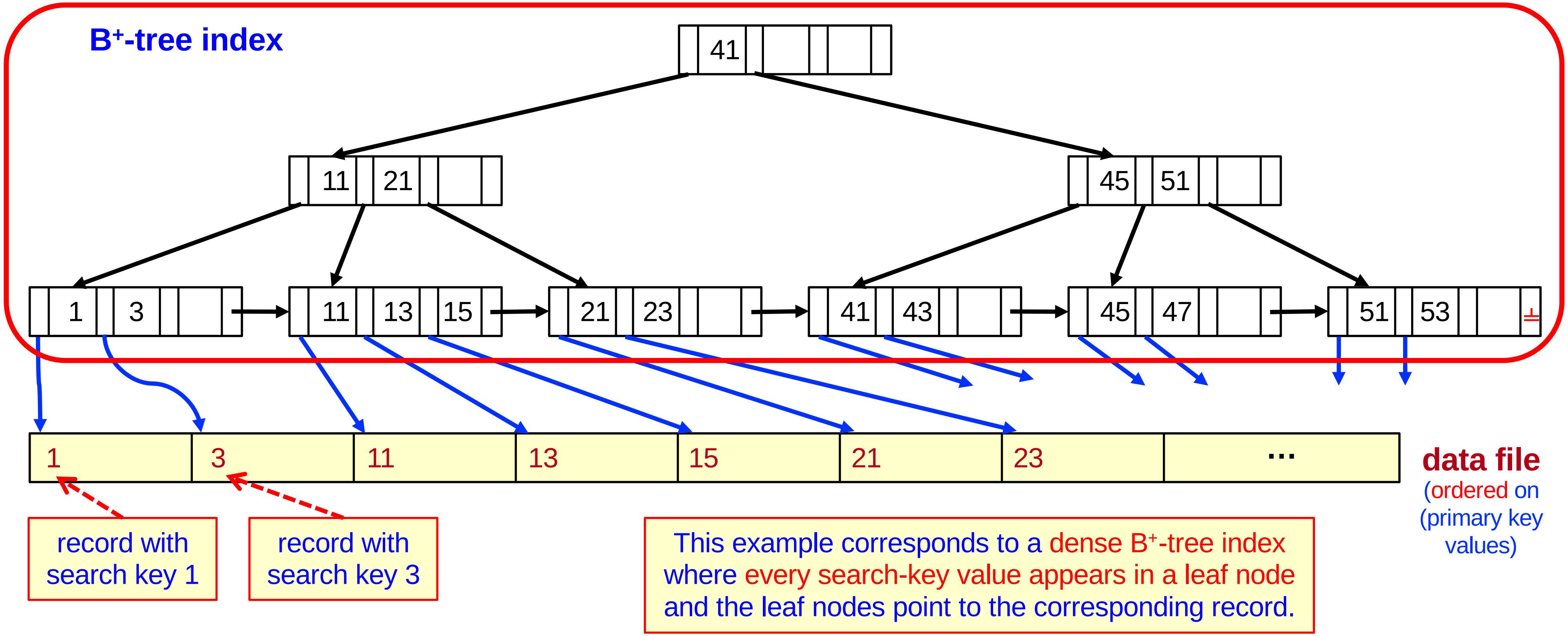
- Since the inter-node connections are given by pointers, the "close" pages need not be "physically" close (i.e., no need for sequential storage).

- The non-leaf levels of the B^+-tree form a sparse index (i.e., not every search-key value is present in the index).

- Since a B^+-tree contains a relatively small number of levels (logarithmic in the size of the data file), search can be conducted efficiently.

- Insertions and deletions to the data file can be handled efficiently, as the index can be restructured in logarithmic time.

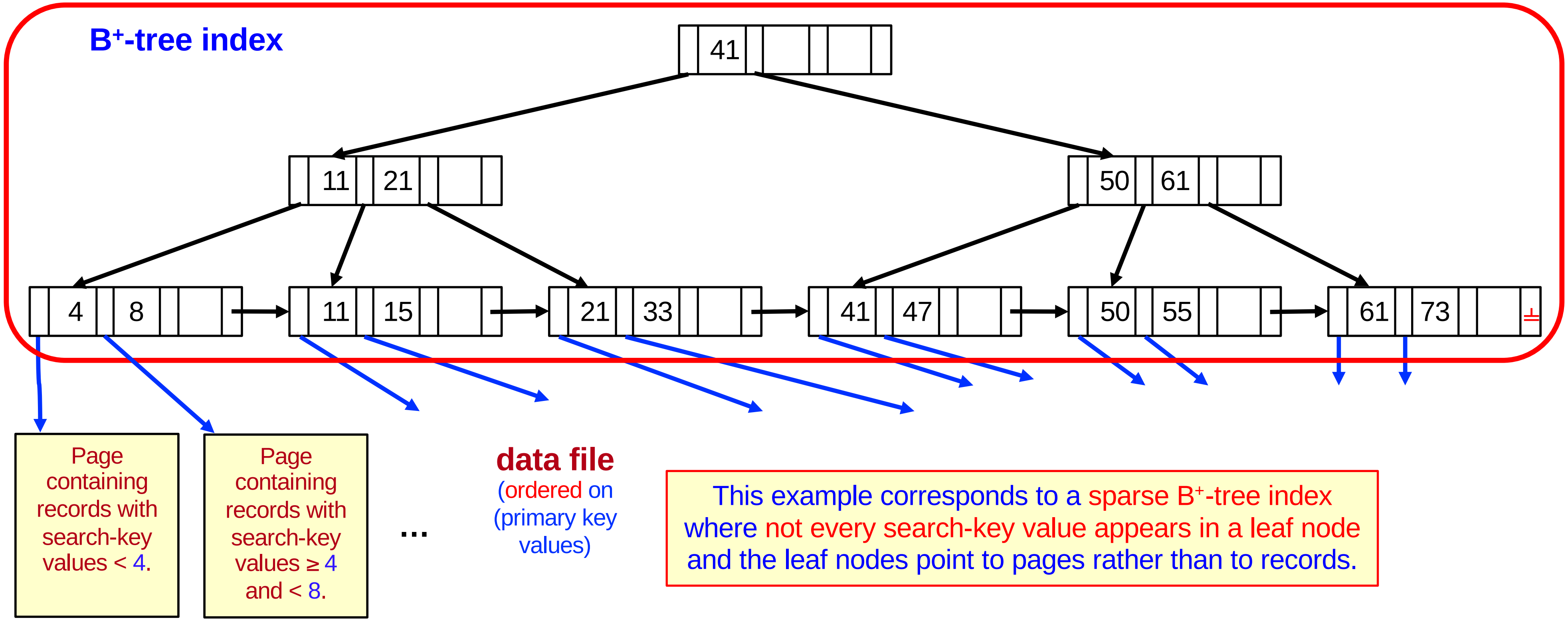
EXAMPLE CLUSTERING B+-TREE ON PRIMARY KEY



Leaf nodes: contain all the primary key values (dense) and point to the record in the file with that value.

Data file: records are ordered by the primary key.

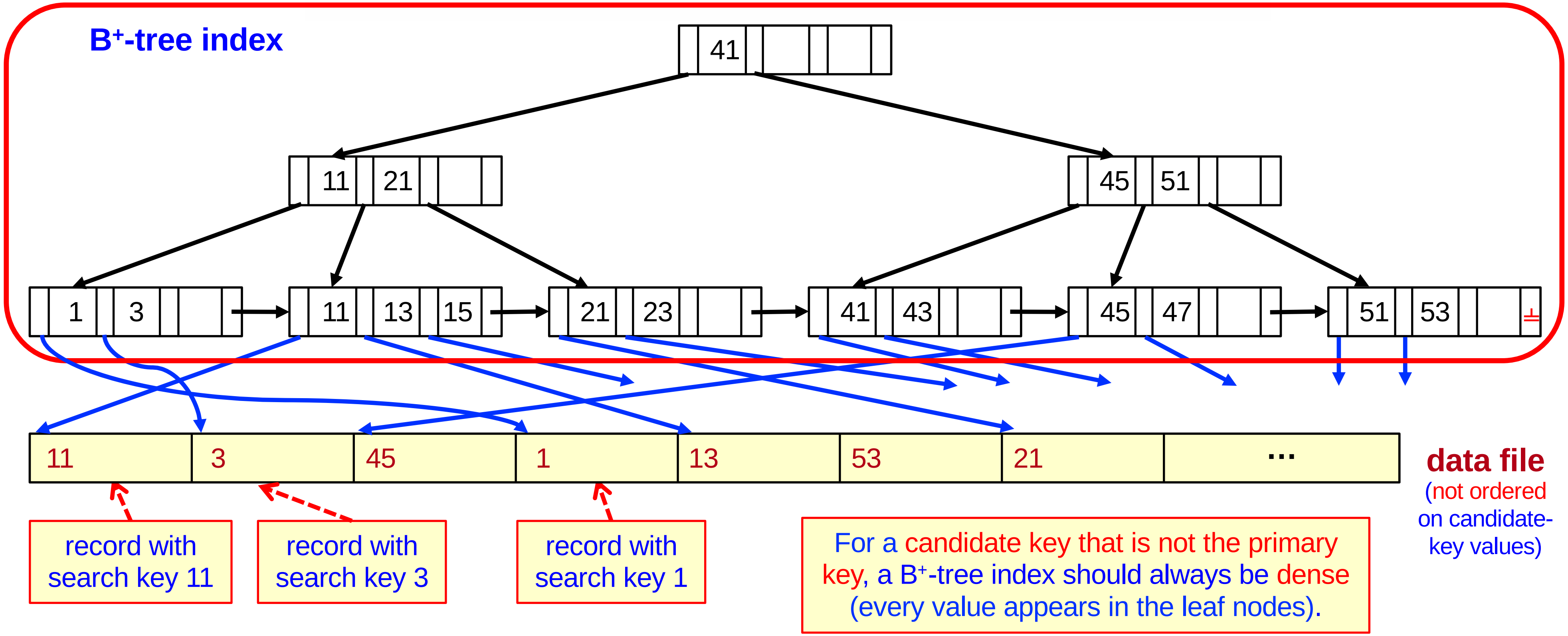
EXAMPLE CLUSTERING B+-TREE ON PRIMARY KEY



Leaf nodes: contain only some of the primary key values (sparse) and point to a page with several records.

Data file: records are ordered by the primary key.

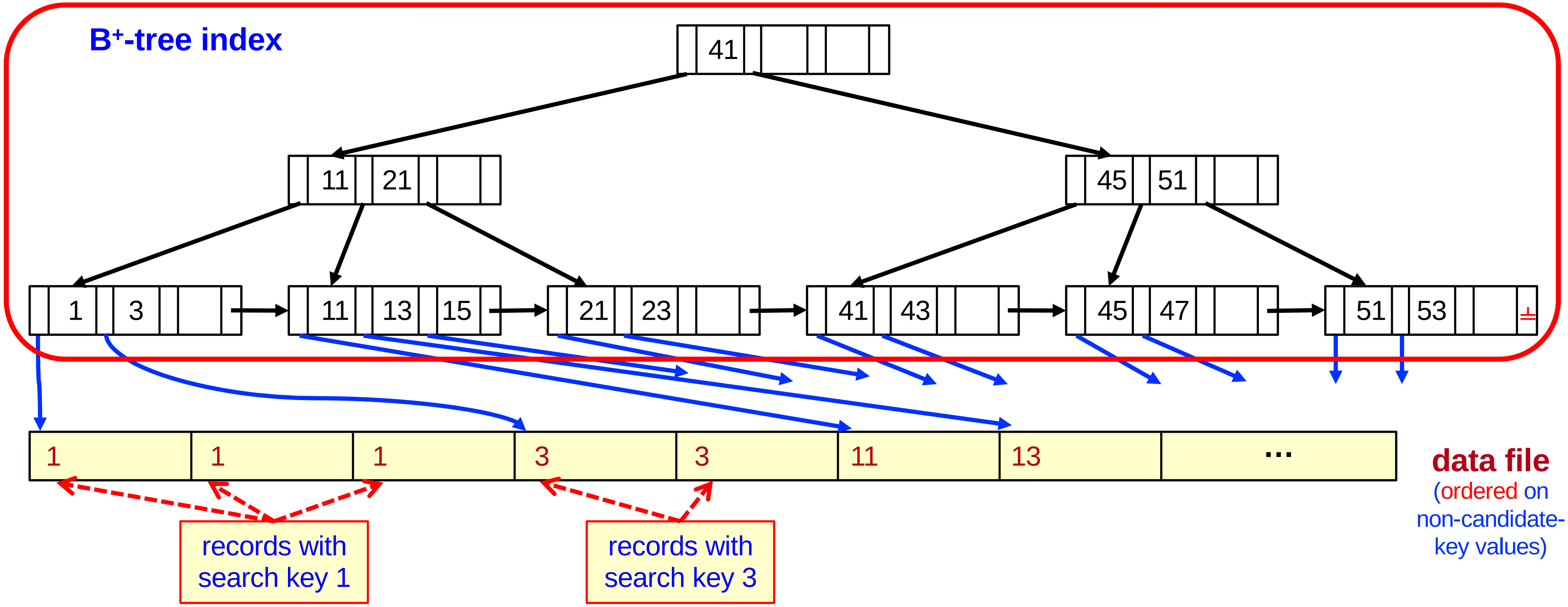
EXAMPLE NON-CLUSTERING/SECONDARY B⁺-TREE ON CANDIDATE KEY



Leaf nodes: contain all the candidate-key values (dense) and point to the record in the file with that value.

Data file: records are not ordered by the candidate key.

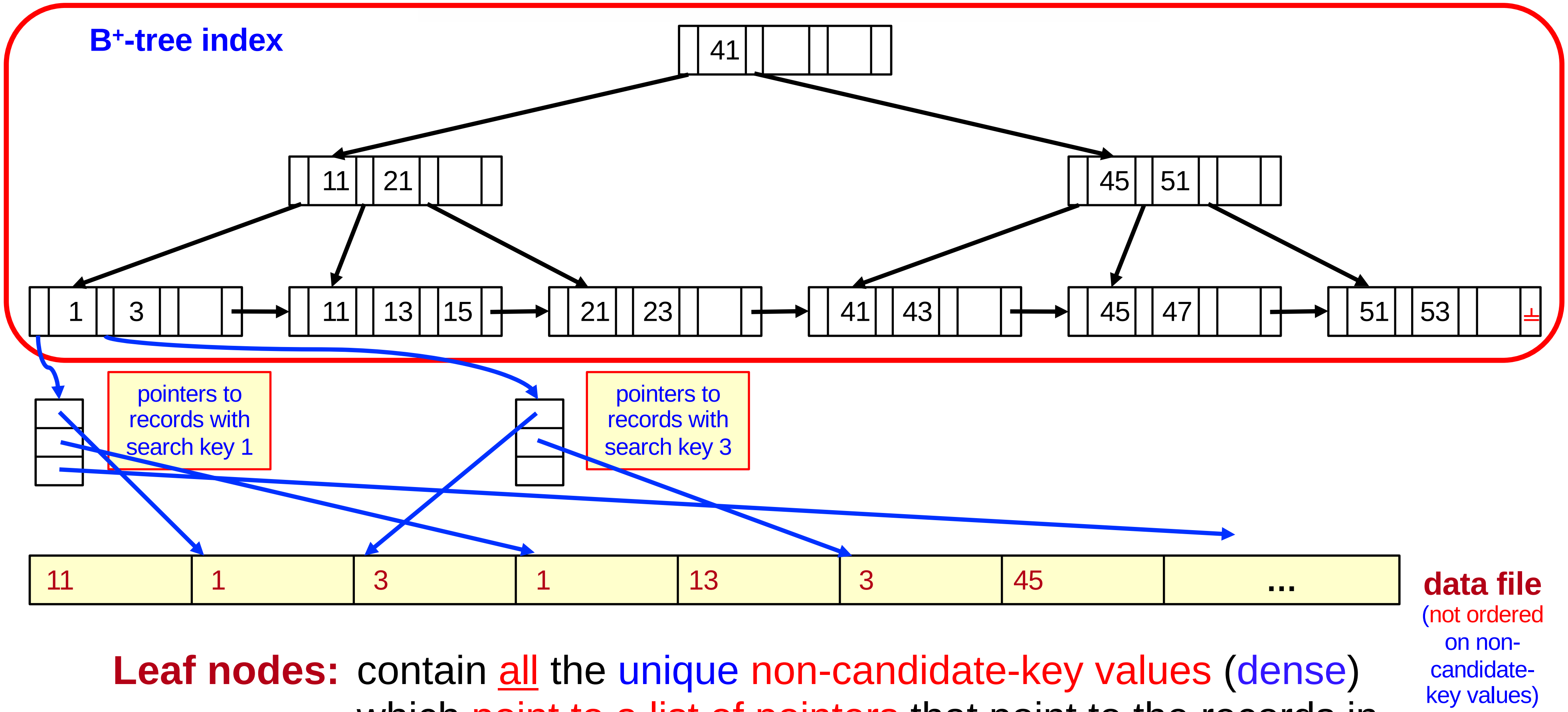
EXAMPLE CLUSTERING B⁺-TREE ON NON-CANDIDATE KEY



Leaf nodes: contain all the unique non-candidate-key values (dense) and point to the first record in the file with that value.

Data file: records are ordered by the non-candidate key, which may be duplicated in different records.

EXAMPLE NON-CLUSTERING B⁺-TREE ON NON-CANDIDATE KEY



Leaf nodes: contain all the unique non-candidate-key values (dense) which point to a list of pointers that point to the records in the file with that value.

Data file: records are not ordered by the non-candidate key.

B+-TREE: QUERY

- Find all records with a search-key value of v.

1. Start at the root node.
   - If there is an entry with search-key value $K_j = v$, *follow pointer* P_{j+1}.
   - Otherwise, if $v < K_{n-1}$ (there are n pointers in the node, i.e., v is not larger than all values in the node) *follow pointer* P_j, where K_j is the smallest search-key value > v.
   - Otherwise, if $v \geq K_{n-1}$, follow P_n to the child node.
2. If the node reached by following the pointer in step 1 is not a leaf node, repeat step 1 on the node, and follow the corresponding pointer.
3. Reached a leaf node.
   - If for some i, key $K_i = v$ follow pointer P_i to the desired record or list of pointers to records.
   - Else no record with search-key value v exists.

B+-TREE: QUERY (CONT'D)

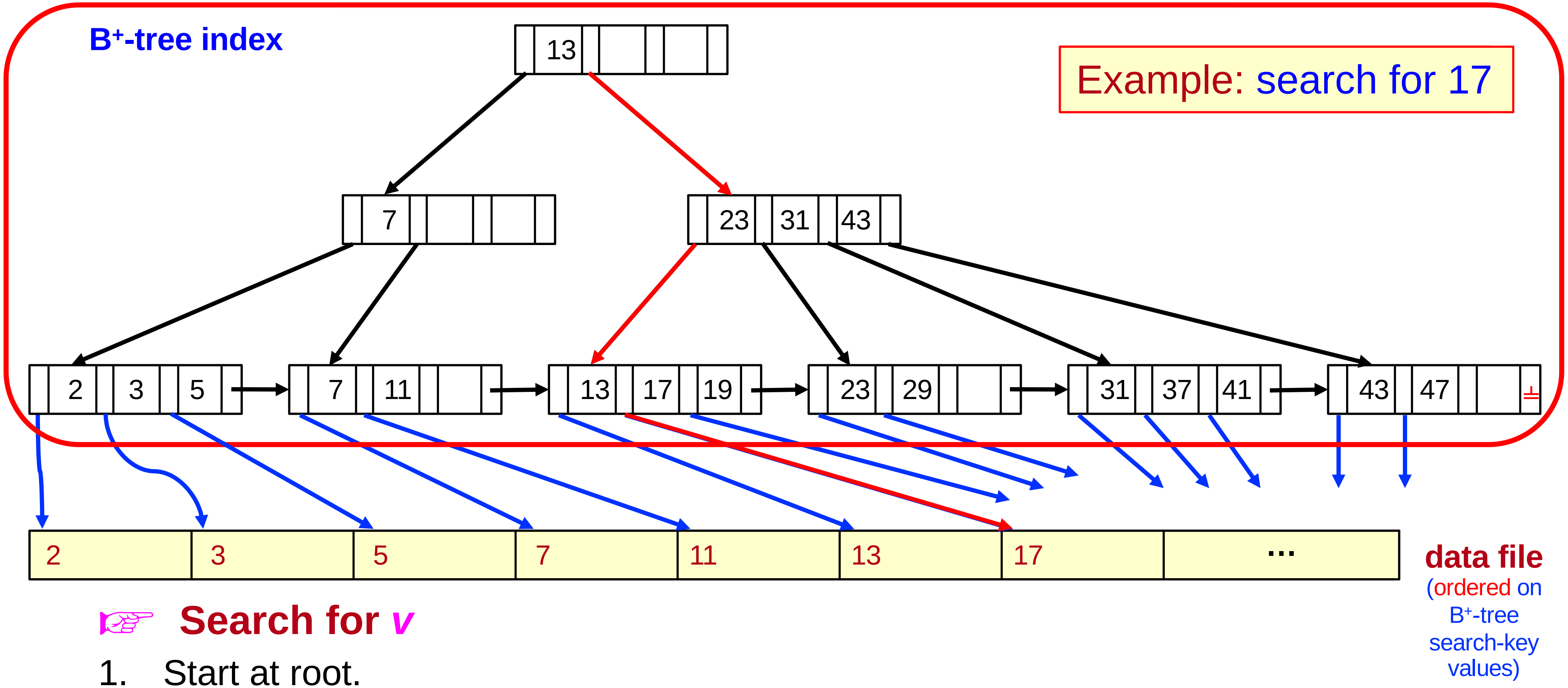
- In processing a query, a path is traversed in the tree from the root to some leaf node.

- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
  - Assuming each node holds the *minimum* number of values.

- A node is generally the same size as a disk page, typically 4KB, and n is typically around 100 (assuming 40 bytes per index entry).

- With 1 million search-key values and n = 100, at most $\log_{50}(1{,}000{,}000)$ = 4 nodes are accessed in a search.

B+-TREE: QUERY EXAMPLE



☞ Search for *v*

1. Start at root.
2. Find K_i such that $K_i > v$; *follow left pointer*; if no $K_i > v$, *follow last pointer* in node.
3. At each node, repeat step 2 until a leaf node is reached.
4. At leaf node:
   - If *v* is found, *follow left pointer* to the record or list of pointers to records.
   - Else, if *v* is not found, then no record with the search-key value *v* exists.

B+-TREE UPDATE

- When a record is inserted or deleted from a table, indexes on the table must be updated.

- Updates to a record can be modeled as a deletion of the old record followed by an insertion of the updated record.

☞ **Only need to consider B+-tree insertion and deletion operations.**

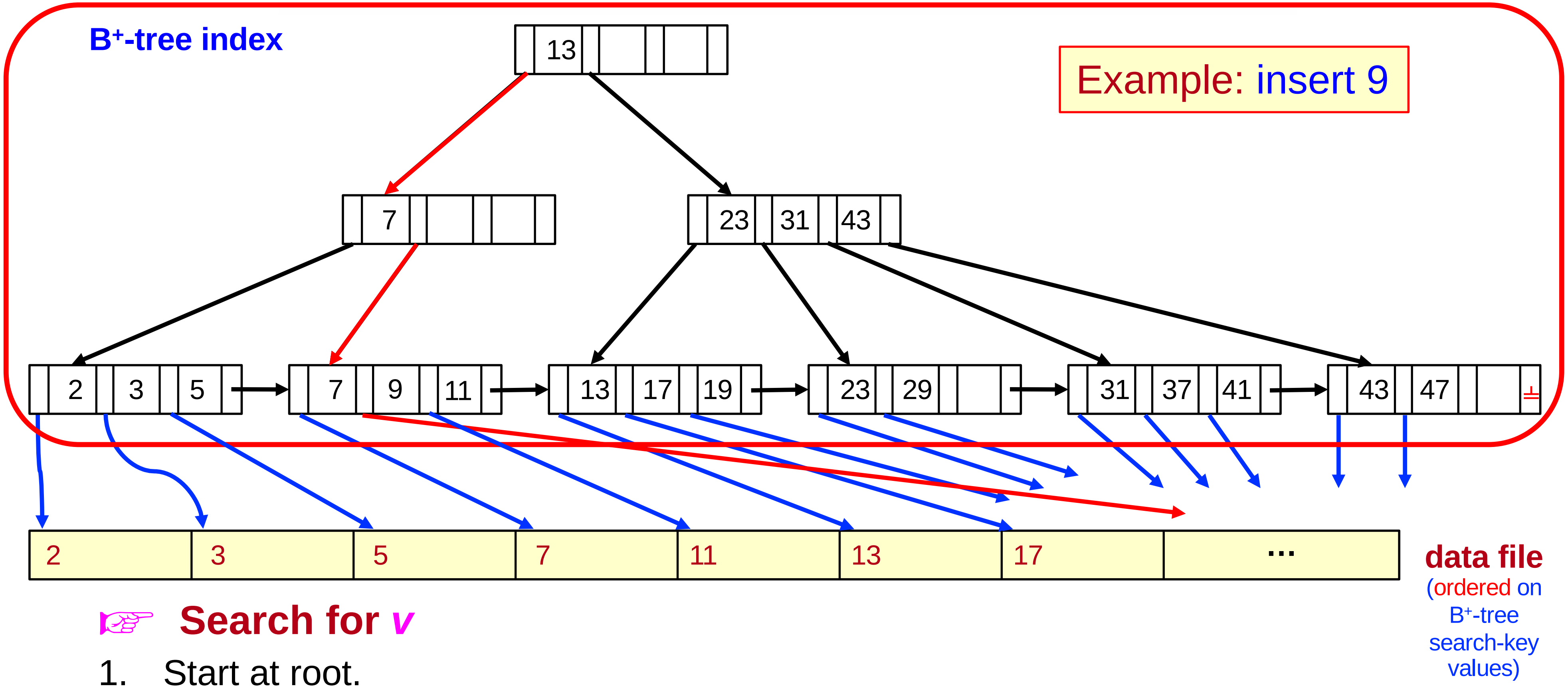
- To maintain the B+-tree properties, it may be necessary to
  - split a node when doing an insertion (node overfull).
  - coalesce (combine) nodes when doing a deletion (node underfull).

☞ **Must ensure that balance is preserved when doing an update.**

B⁺-TREE INSERTION

- Use query strategy to find the leaf node *L* where the value belongs.
- Put the value into *L* (a leaf node).
  - If *L* has enough space, *done*!
  - Else (*leaf node overfull*), must *split* *L* (into *L* and a new node leaf node *L'*).
    - Place first $\lceil n/2 \rceil$ values into *L* , copy up value at $\lceil n/2 \rceil+1$ and place values starting at $\lceil n/2 \rceil+1$ up to and including the last value into *L'*.
    - **Copy up:** Insert an index entry (value at $\lceil n/2 \rceil+1$, pointer to *L'*) into the correct position in the (non-leaf) parent of *L*.
- Splits can happen recursively.
  - To split an index (*internal*) node *N* (into *N* and a new internal node *N'*).
    - Place first $\lceil n/2 \rceil$ -1 values into *N*, push up value at $\lceil n/2 \rceil$ and place values starting at $\lceil n/2 \rceil+1$ up to and including the last value into *N'*.
    - **Push up:** Insert an index entry (value at $\lceil n/2 \rceil$, pointer to *N'*) into the correct position in the parent of *N*.
- Splits "grow" tree; a root split increases the height.
  - B^+-tree growth: The tree gets *wider* or *one level taller at the top*.

B+-TREE: INSERTION EXAMPLE



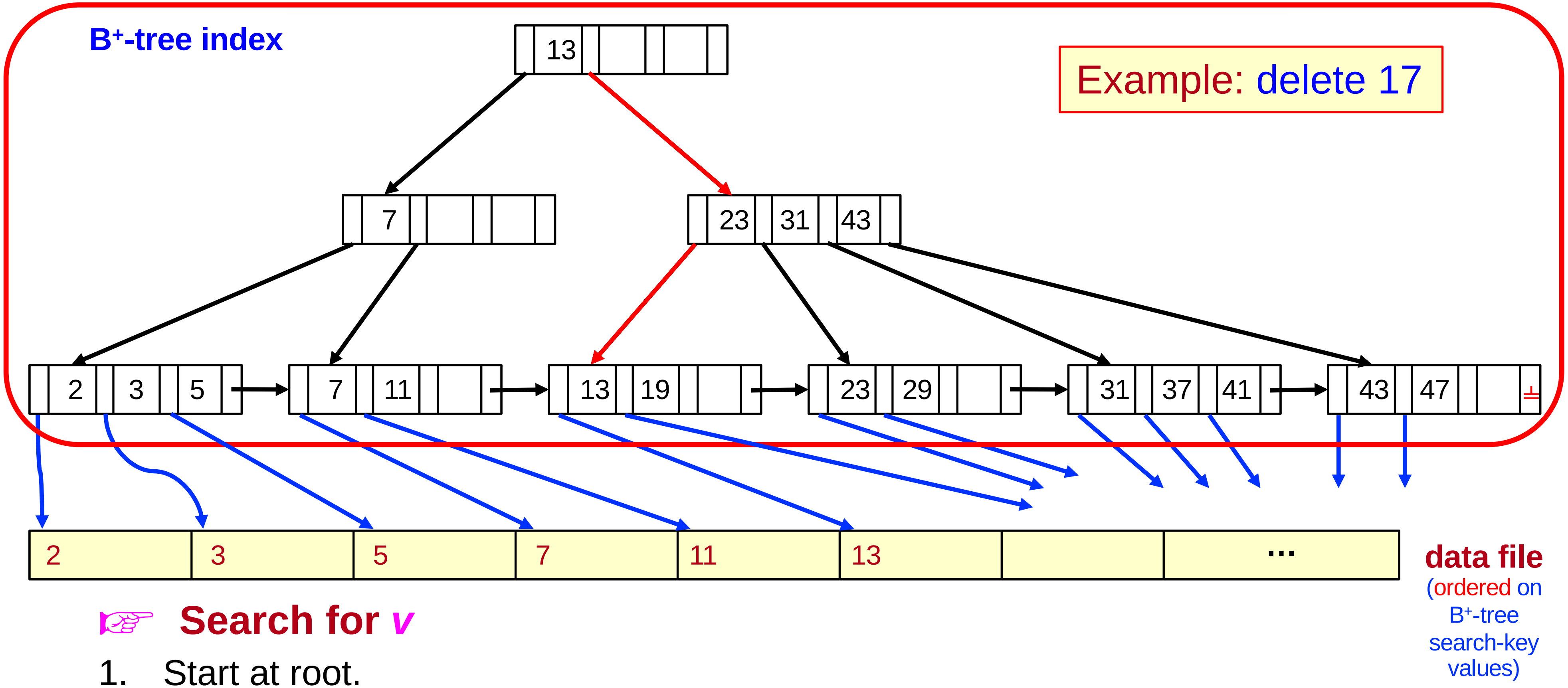
☞ **Search for *v***

1. Start at root.
2. Find K_i such that $K_i > v$; *follow left pointer*; if no $K_i > v$, *follow last pointer* in node.
3. At each node, repeat step 2 until a leaf node is reached.
4. At leaf node:
   - If enough space, *insert in node in order*.
   - Else, (*leaf node overfull*), must *split*.

B⁺-TREE DELETION

- Use query strategy to find the leaf node *L* where the value belongs.
- Remove the leaf node entry.
  - If *L* has at least $\lceil (n\text{-}1)/2 \rceil$ values, *done*!
  - Else *L* has less than $\lceil (n\text{-}1)/2 \rceil$ values (*leaf node underfull*),
    - Try to re-distribute by borrowing values from a *sibling node* (an adjacent node to the right or the left *under the same parent*).
    - If re-distribution fails, *merge* *L* and a sibling.
- If a merge occurred, delete the entry (pointing to *L* or the sibling) from parent (non-leaf node) of *L*.
- A merge could propagate to the root, decreasing the height of the tree.
  - For non-leaf (internal) nodes, redistribution requires that a node have at least $\lceil n/2 \rceil$ pointers.
  - If this criterion cannot be met by re-distribution, then nodes must be merged.

B+-TREE: DELETION EXAMPLE



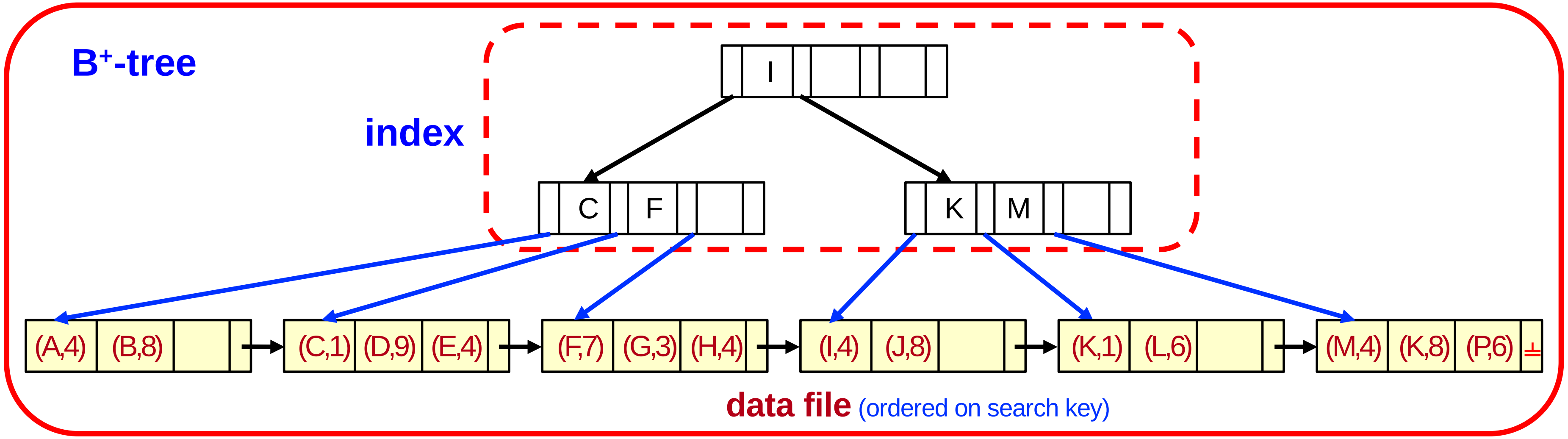
☞ Search for *v*

1. Start at root.
2. Find K_i such that $K_i > v$; *follow left pointer*; if no $K_i > v$, *follow last pointer* in node.
3. At each node, repeat step 2 until a leaf node is reached.
4. At leaf node:
   - If *L* has at least $\lceil(n\text{-}1)/2\rceil$ values, *done*!.
   - Else, *L* has less than $\lceil(n\text{-}1)/2\rceil$ values (*leaf node underfull*).

B^+-TREE FILE ORGANIZATION

- The index file degradation problem is solved by using B^+-tree indexes.
- The data file degradation problem can be solved by using B^+-tree file organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B^+-Tree File Organization (cont'd)

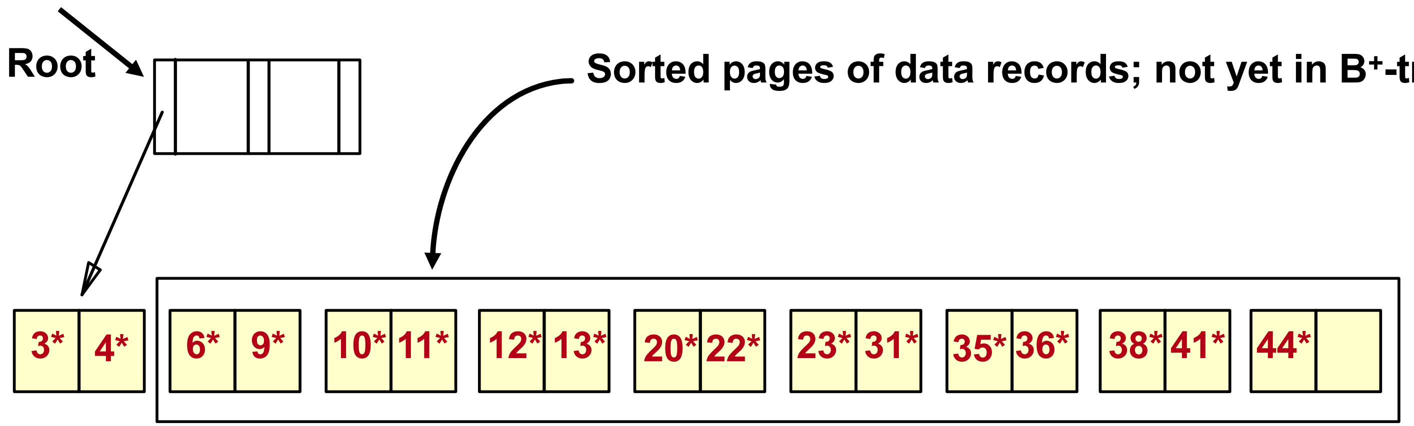


Example of B^+-tree File Organization

- In a B^+-tree file organization, good space utilization is important since records use more space than pointers.
- To improve space utilization, we can involve more sibling nodes in redistribution during splits and merges.
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries.

B+-TREE BULK LOADING

- Creating a B+-tree by repeatedly inserting records is very slow and can be costly (i.e., need to read and write leaf nodes).
- Bulk Loading can be done much more efficiently.
- Initialization: Sort all data entries (using external sorting), insert pointer to first (leaf) page in a new (root) page.
- Only need to write each leaf node once; never have to read it.



B+-TREE BULK LOADING: EXAMPLE

- Index entries for leaf pages are always entered into the right-most index page just above the leaf level. When this fills up, it splits. (A split may go up right-most path to the root.)

- Much faster than repeated inserts!