Chapter 14: Indexing Structures for Files

Chapter Outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees

Indexes

- An index on a file speeds up selections on the search key fields for the index
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - "Search key" is not the same as "key"

    key = minimal set of fields that uniquely identify a tuple

- An index contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given key value k

Primary vs. Secondary Index

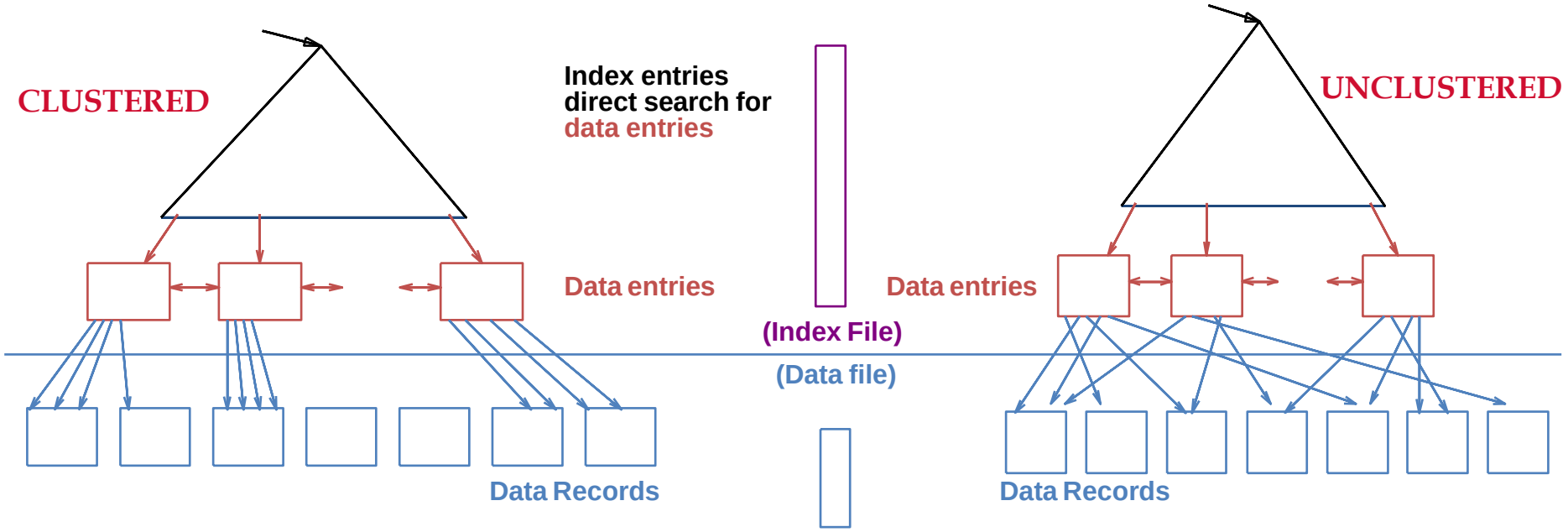
- If search key contains primary key, then called primary index, otherwise secondary
  - Unique index: Search key contains a candidate key
- Duplicate data entries:
  - if they have the same value of search key field k
  - Primary/unique index never has a duplicate
  - Other secondary index can have duplicates

Clustered vs. Unclustered Index

- If order of data records in a file is the same as, or `close to', order of data entries in an index, then clustered, otherwise unclustered
  - A file can be clustered on at most one search key
  - Cost of retrieving data records (range queries) through index varies greatly based on whether index is clustered or not

Clustered vs. Unclustered Index

- To build clustered index, first sort the Heap file
  - with some free space on each page for future inserts
  - Overflow pages may be needed for inserts
  - Thus, data records are `close to', but not identical to, sorted



Types of Indexes

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)

Properties of Index Types

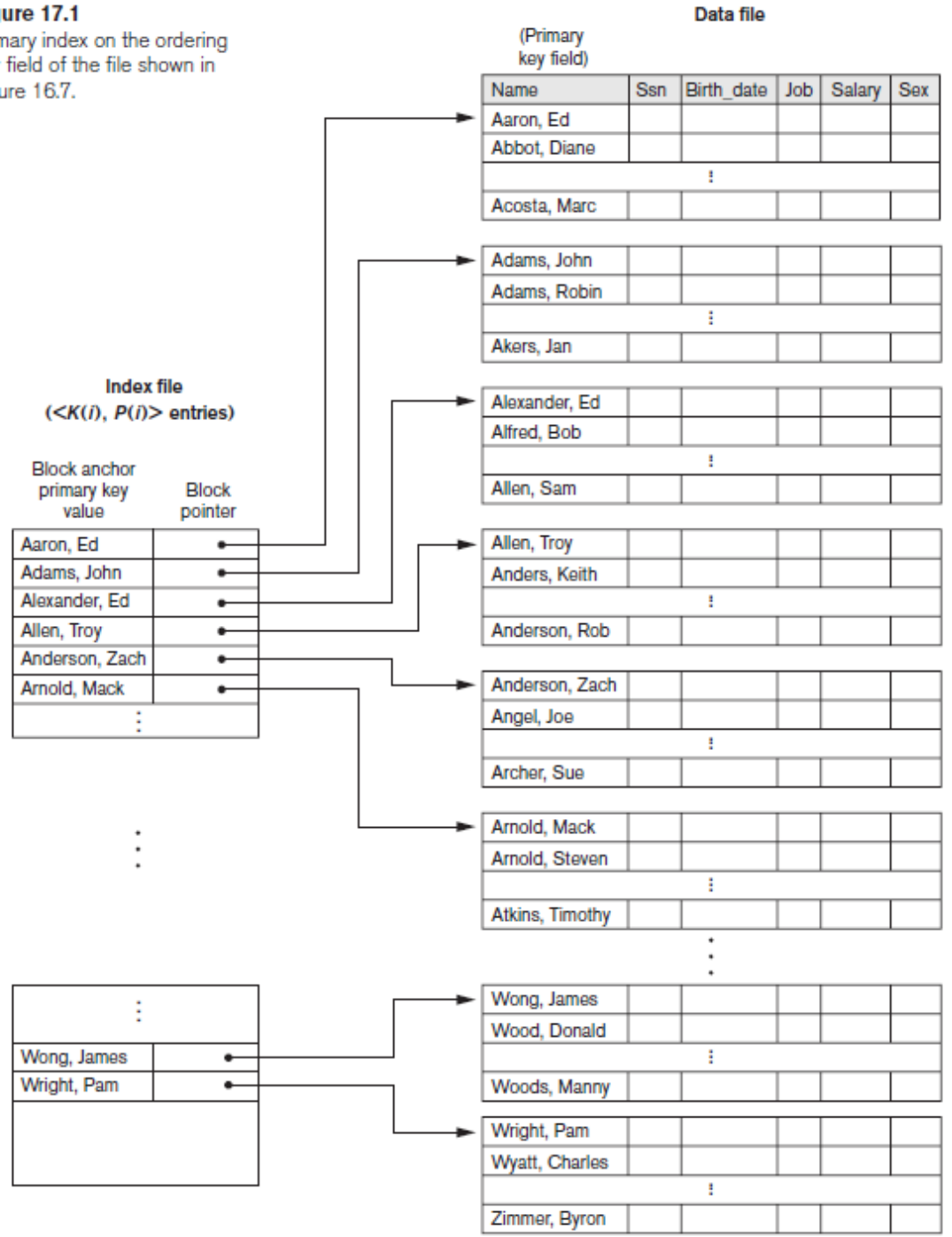
Table 17.2 Properties of Index Types

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 or Nondense	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.

Example of Index

Figure 17.1
Primary index on the ordering key field of the file shown in Figure 16.7.



Example

- Ordered file with records r = 300000, disk block size = B = 4096 bytes and file record length = R = 100 bytes.
- What is the number of I/O operations needed for searching for one record?
- Let ordering key field is 9 bytes, block pointer is 6 bytes; total 15 bytes in each entry of index file.
- What is be the number of I/O operations needed for searching for one record using a primary index file?

Problems with Index

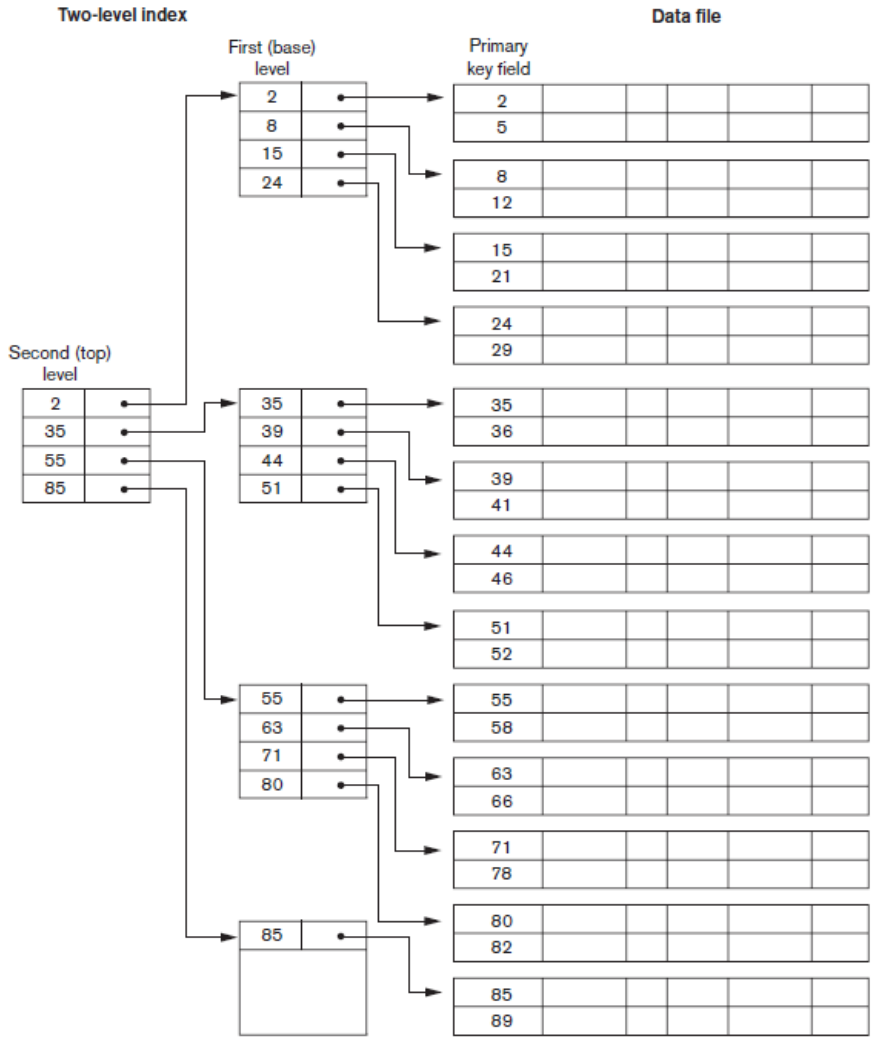
- Insertion
  - Inserting in correct position (make space, change index entries)
  - Move records to make space for new records
  - Move will change anchor records of some blocks
  - Use linked list or overflow records
- Deletion
  - Use delete markers

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

A Two-level Primary Index

Figure 17.6
A two-level primary index resembling ISAM (indexed sequential access method) organization.

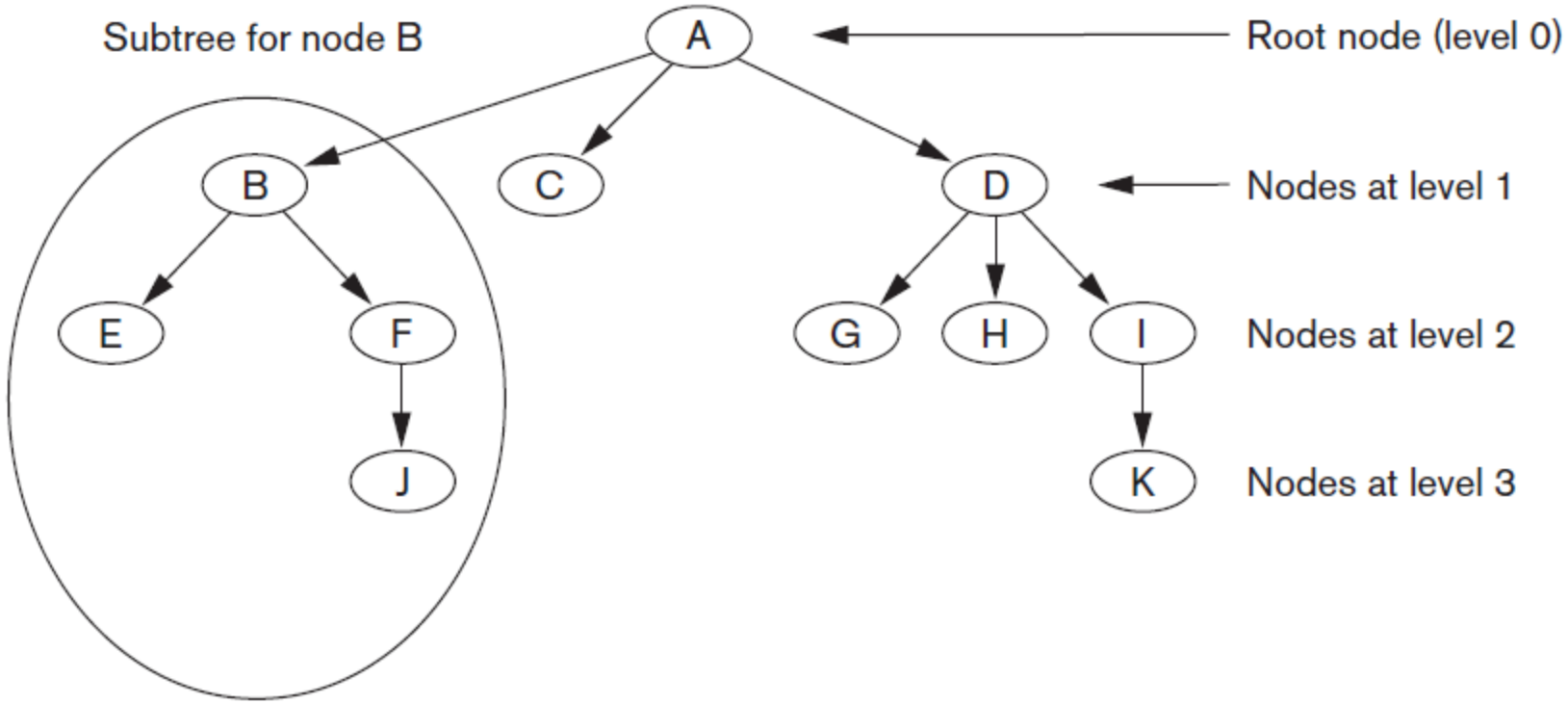


Methods for indexing

- Tree-based
- Hash-based

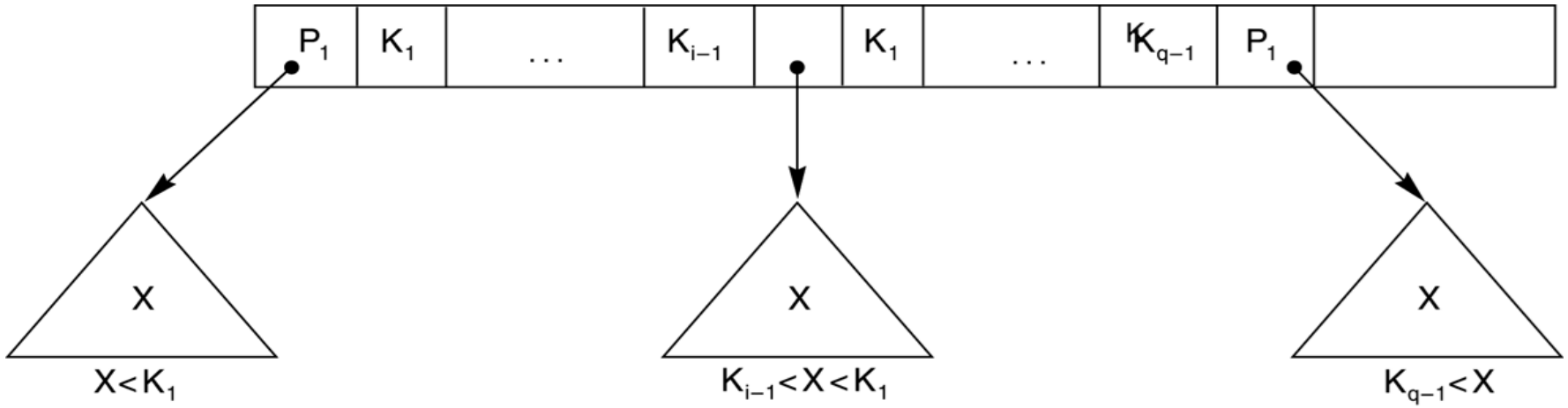
Tree Data Structure

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

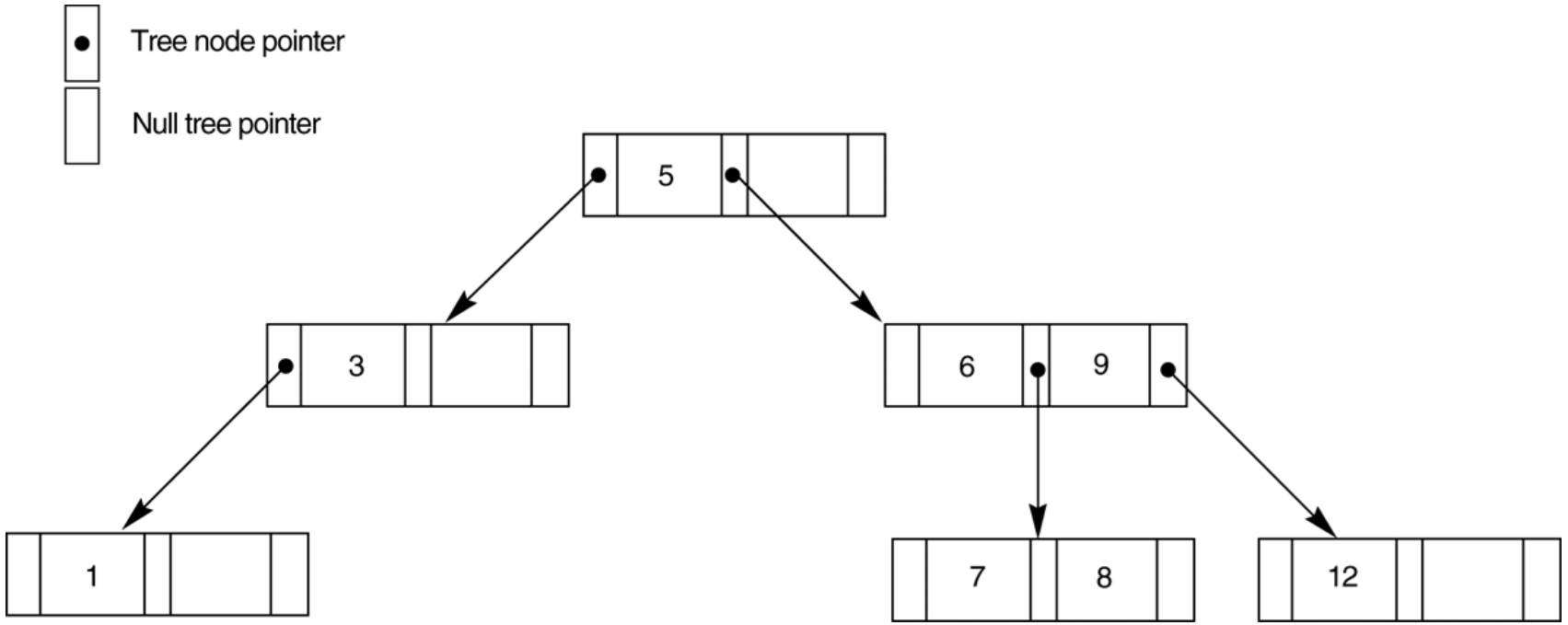


(Nodes E, J, C, G, H, and K are leaf nodes of the tree)

A Node in a Search Tree with Pointers to Subtrees below It



A search tree of order p = 3.



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

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

Dynamic Multilevel Indexes Using B-Trees and B+-Trees (contd.)

- 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 other 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 must be merged with neighboring nodes

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

B-tree Structures

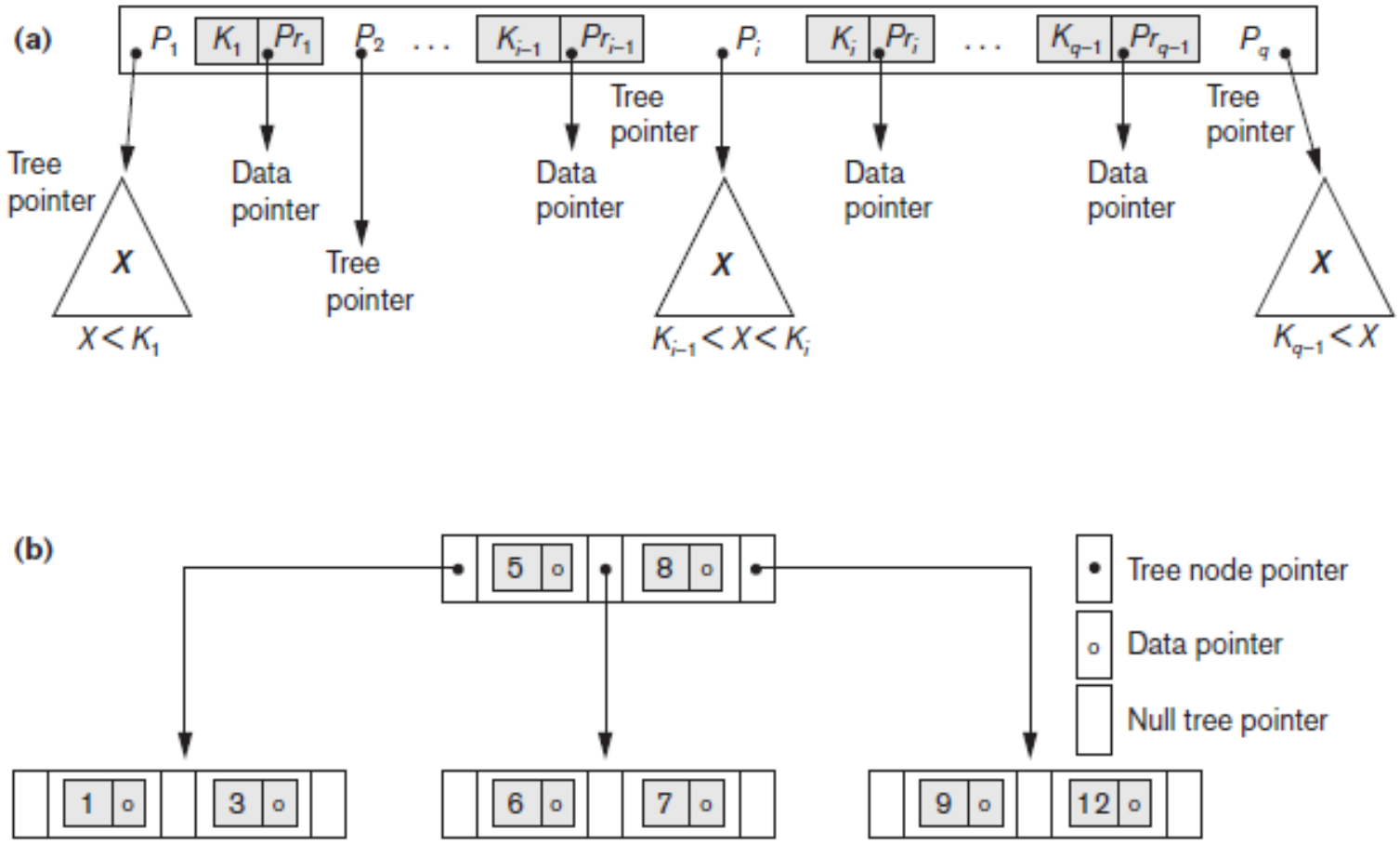


Figure 17.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

The Nodes of a B+-tree

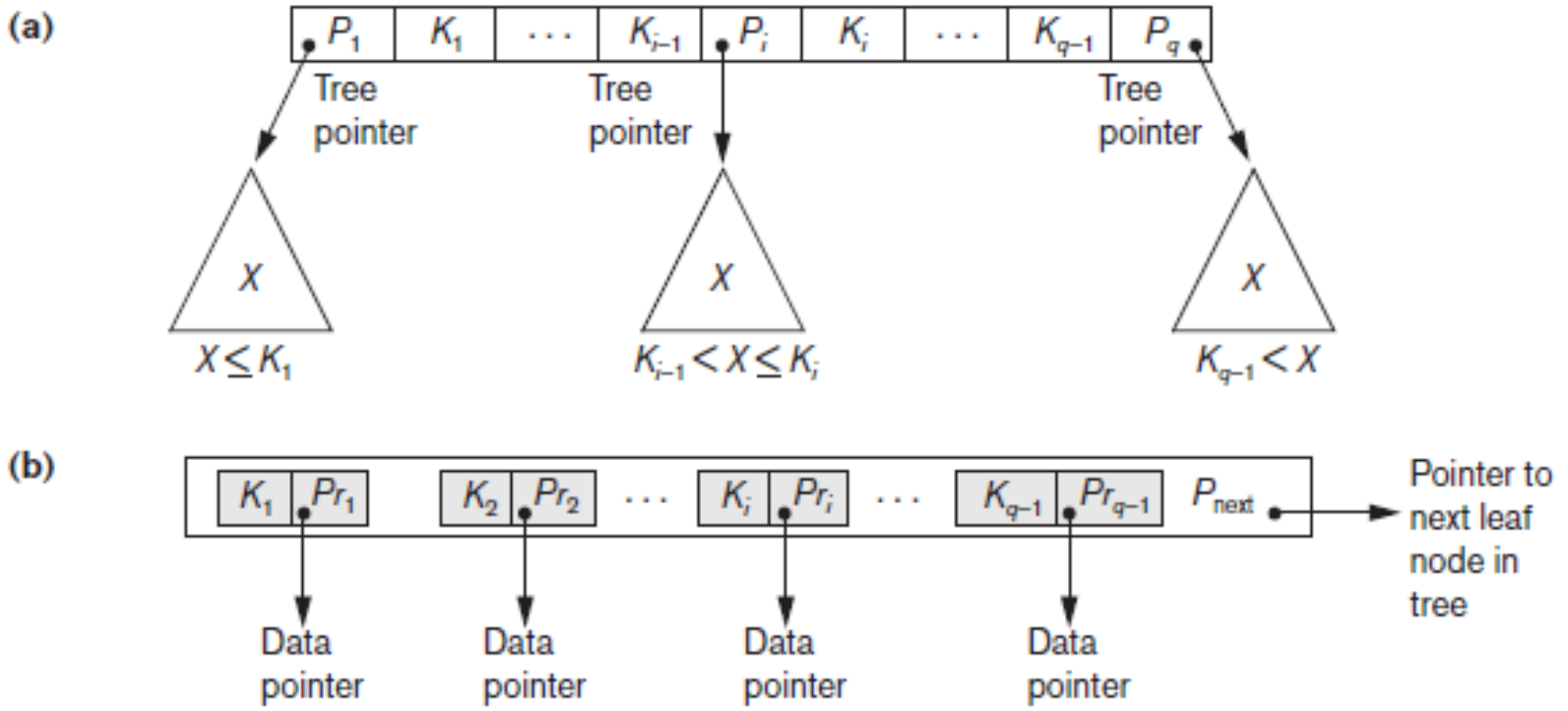
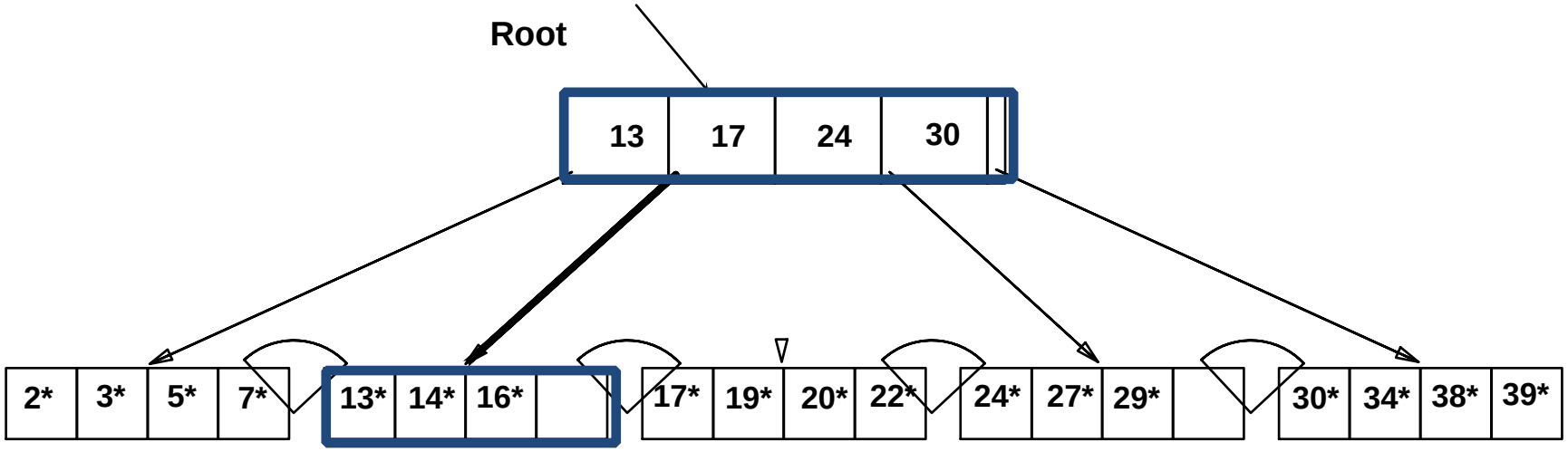


Figure 17.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

B+ Tree Equality Search

Search begins at root, and key comparisons direct it to a leaf.

Search for 15*

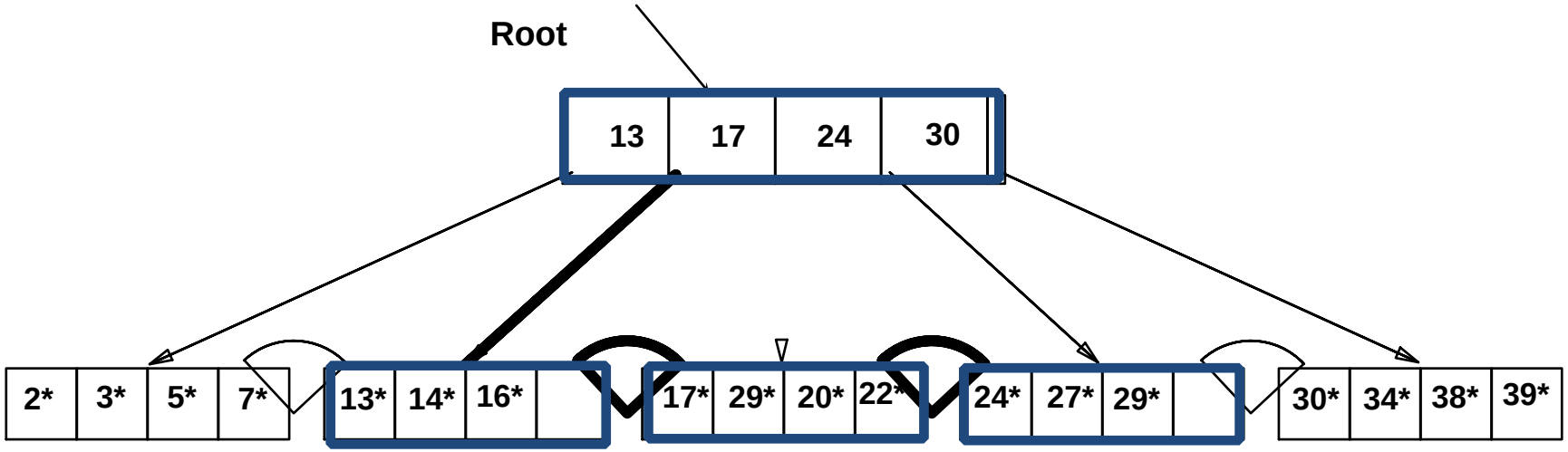


Based on the search for 15, we know it is not in the tree!*

B+ Tree Range Search

Search all records whose ages are in [15,28].

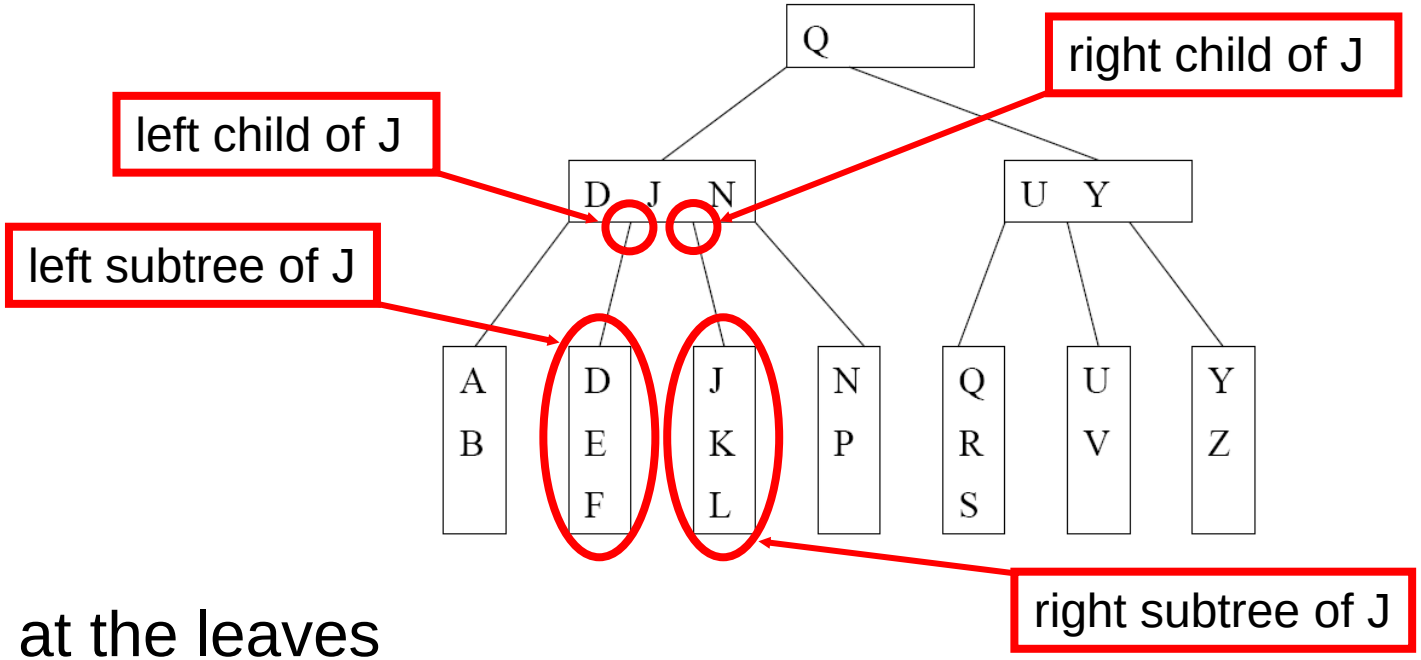
- Equality search 15*.
- Follow sibling pointers.



B+-Tree — Basic Information

- Leaves contain data items
  - all are at the same depth
  - each node has $\lceil q/2 \rceil$ to q data (q also called p_{leaf} or L)
- Internal nodes contain searching keys
  - each node has $\lceil p/2 \rceil$ to p children
  - each node has $\lceil (p/2)-1 \rceil$ to (p-1) searching keys
    - key i is the smallest key in subtree i+1
- Root
  - can be a single leaf, or has 2 to p children

B+-Tree — Example



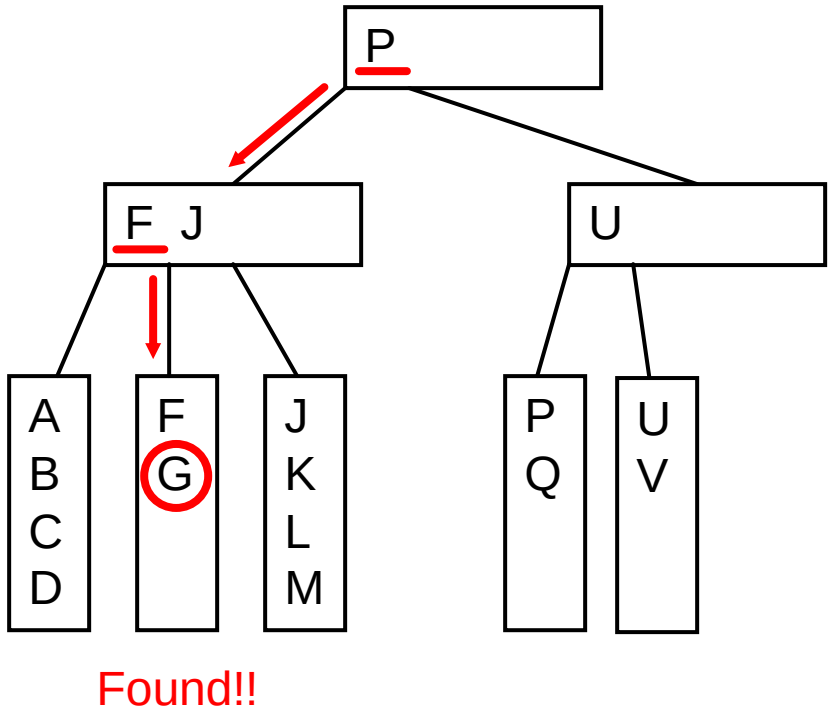
- p = q (L) = 4
- Records are at the leaves
- Node are at least half-full, so that the tree will not degenerate into simple binary tree or even link list
- Left child pointer & right child pointer and also left subtree & right subtree are defined

B+-Tree — In Practical

- Each internal node & leaf has the size of one I/O block of data
  - minimize tree depth and so as no. of disk access
- First one or two levels of the tree are stored in main memory to speed up searching
- Most internal nodes have less than (p-1) searching keys most of the time
  - huge space wastage for main memory, but not a big problem for disk

B+-Tree — Searching Example 1

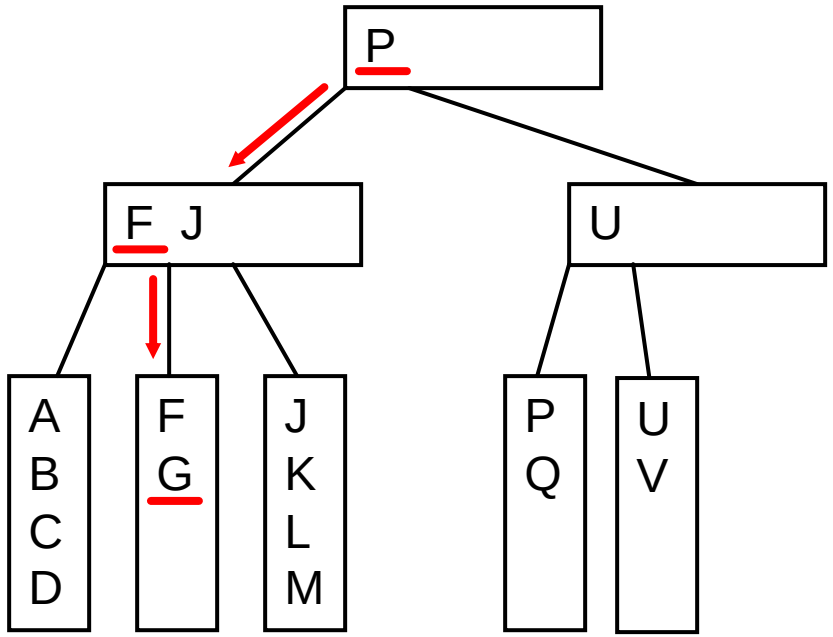
- Search G



p = q = 4

B+-Tree — Searching Example 2

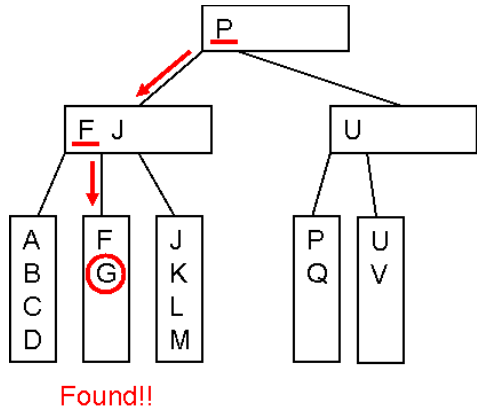
- Search H



Not Found!!

p = q = 4

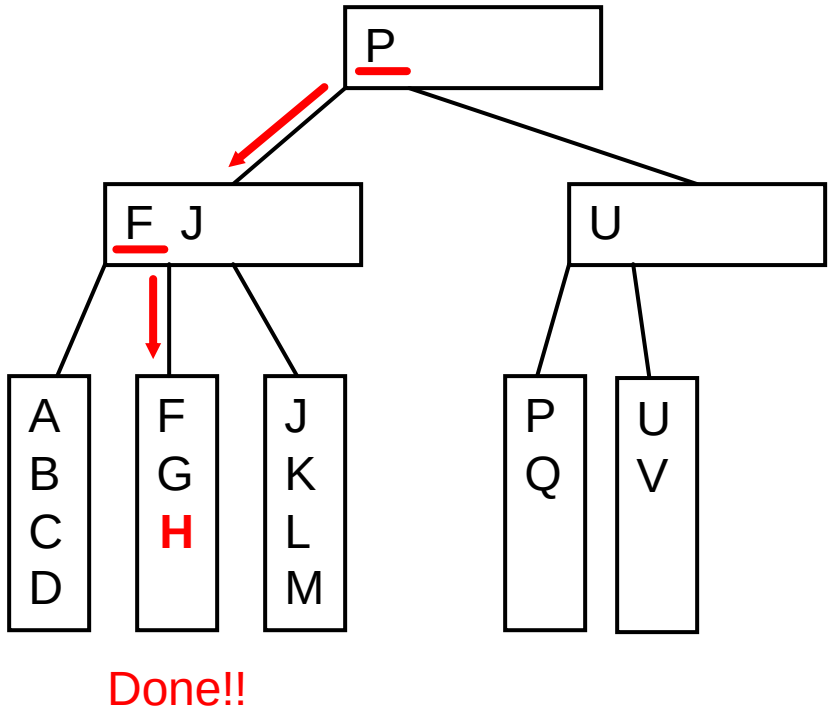
B+-Tree — Searching Algorithm



- Searching KEY:
  - Start from the root
  - If an internal node is reached:
    - Search KEY among the keys in that node
      - linear search or binary search
    - If KEY < smallest key, follow the leftmost child pointer down
    - If KEY >= largest key, follow the rightmost child pointer down
    - If K_i <= KEY < K_j, follow the child pointer between K_i and K_j
  - If a leaf is reached:
    - Search KEY among the keys stored in that leaf
      - linear search or binary search
    - If found, return the corresponding record; otherwise report not found

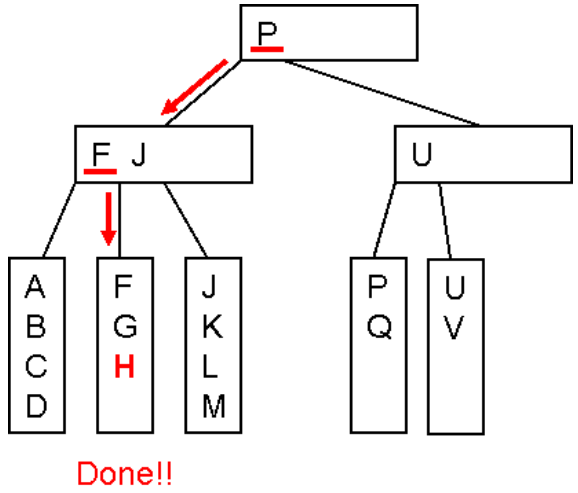
B+-Tree — Insertion Example 1

- Insert H



p = q = 4

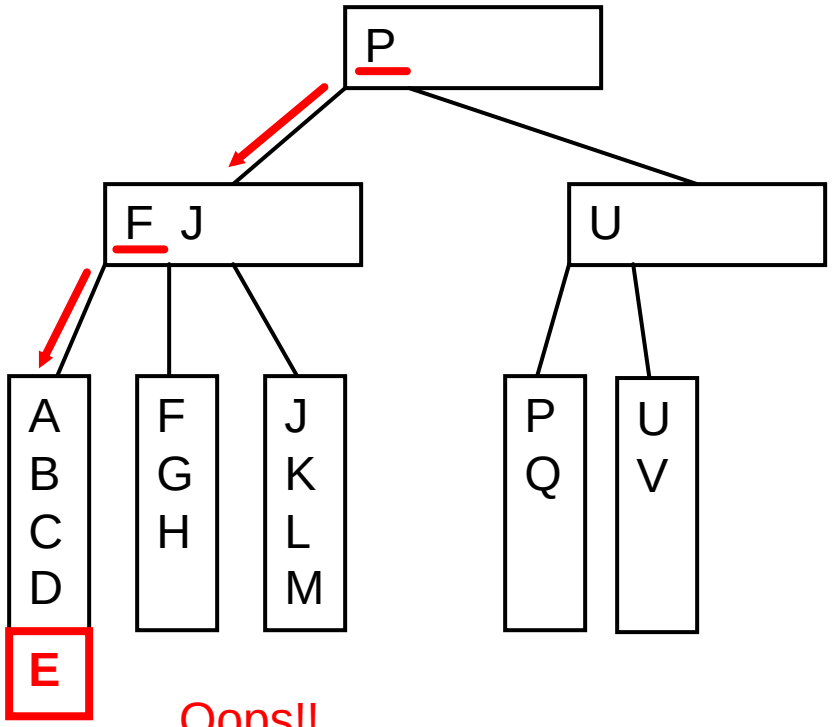
B+-Tree — Insertion Algorithm



- Insert KEY:
  - Search for KEY using search operation
    - we will reach a leaf and get “Not Found”
  - Insert KEY into that leaf
    - If the leaf contains <L keys, just insert KEY into it
    - If the leaf contains L keys, **splitting** is necessary

B+-Tree — Insertion Example 2

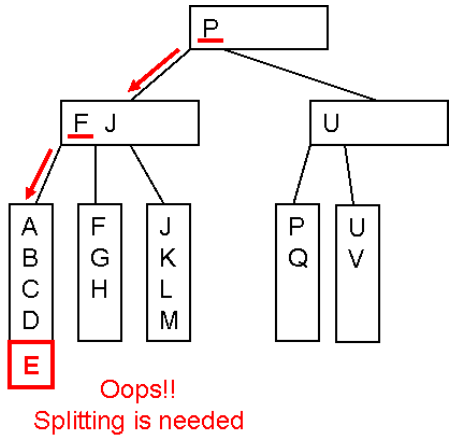
- Insert E



Oops!!
Splitting is needed

p = q = 4

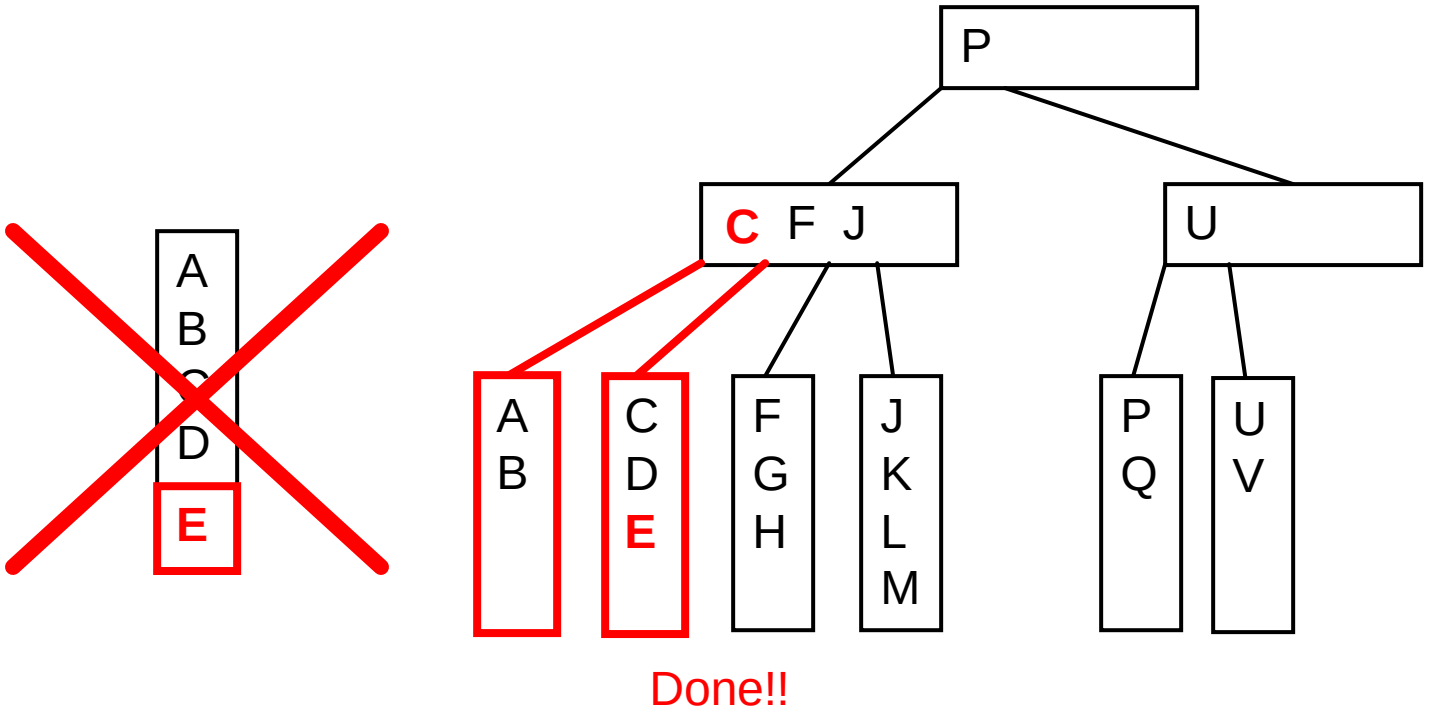
B+-Tree — Insertion Algorithm (Con't)



- Splitting leaf:
  - Cut the node out, insert KEY into it
  - Split it into 2 new leaves L_{left} and L_{right}
    - L_{left} has the $\lfloor (q+1)/2 \rfloor$ smallest keys
    - L_{right} has the remaining $\lceil (q+1)/2 \rceil$ keys
  - Make a copy of the smallest key in L_{right}, say J, to be the parent of L_{left} and L_{right}
  - Insert J, together with L_{left} and L_{right}, into the original parent node

B+-Tree — Insertion Example 2 (Con't)

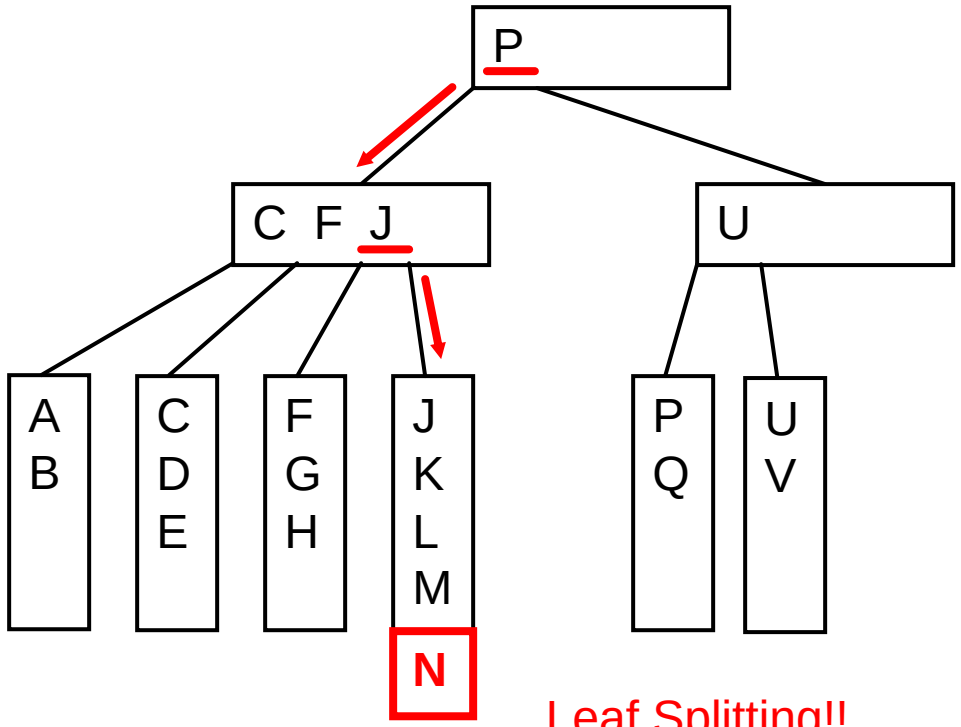
- Insert E



p = q = 4

B+-Tree — Insertion Example 3

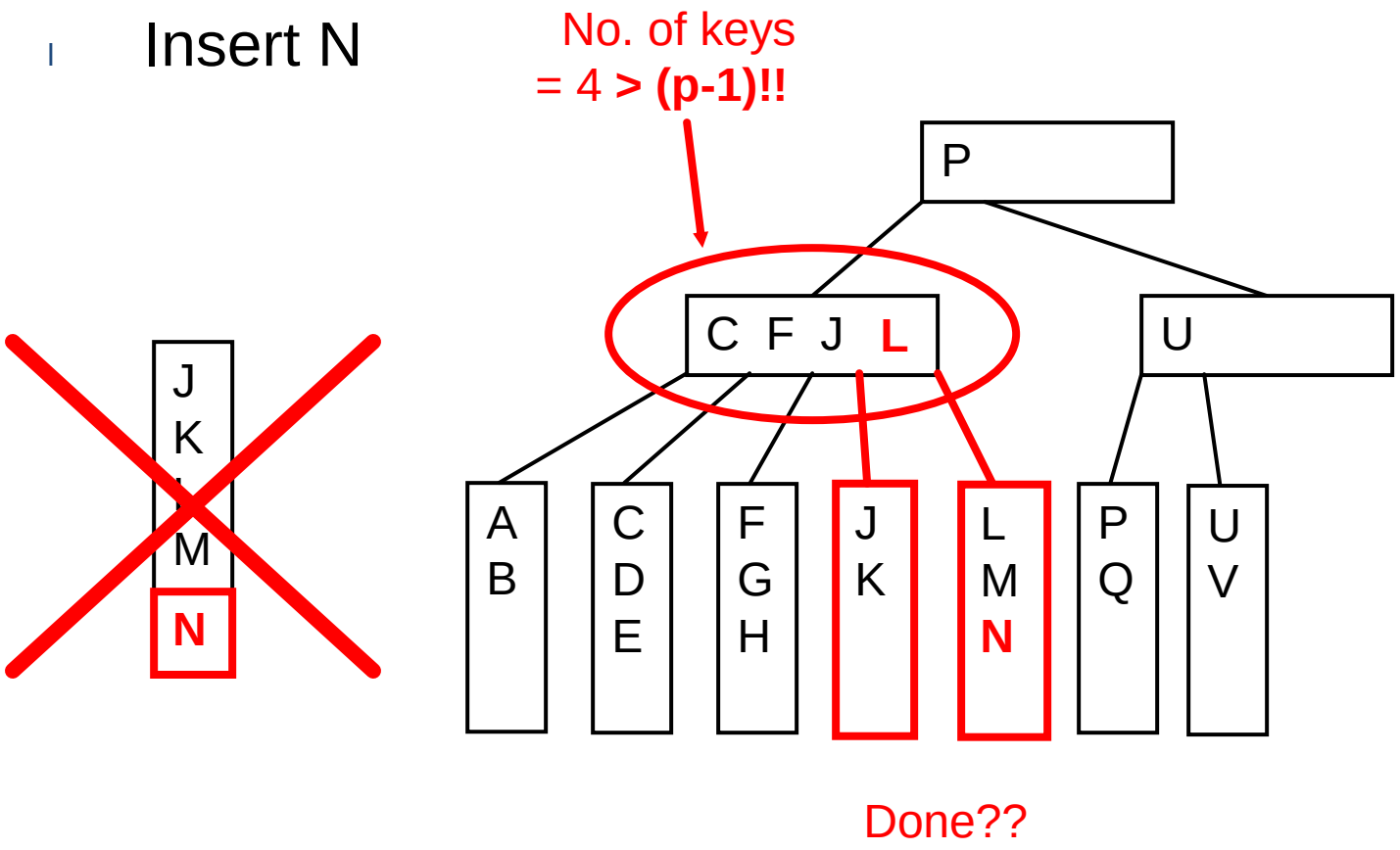
- Insert N



p = q = 4

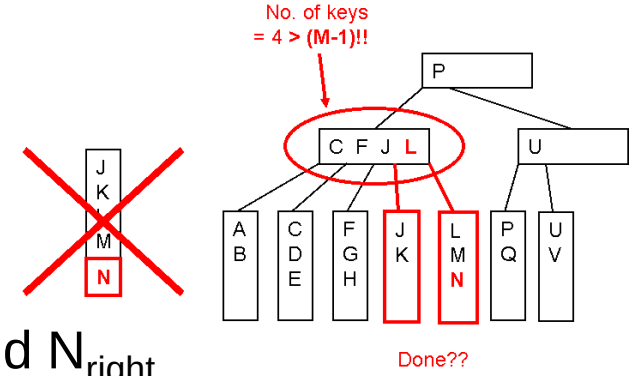
B+-Tree — Insertion Example 3 (Con't)

- Insert N



p = q = 4

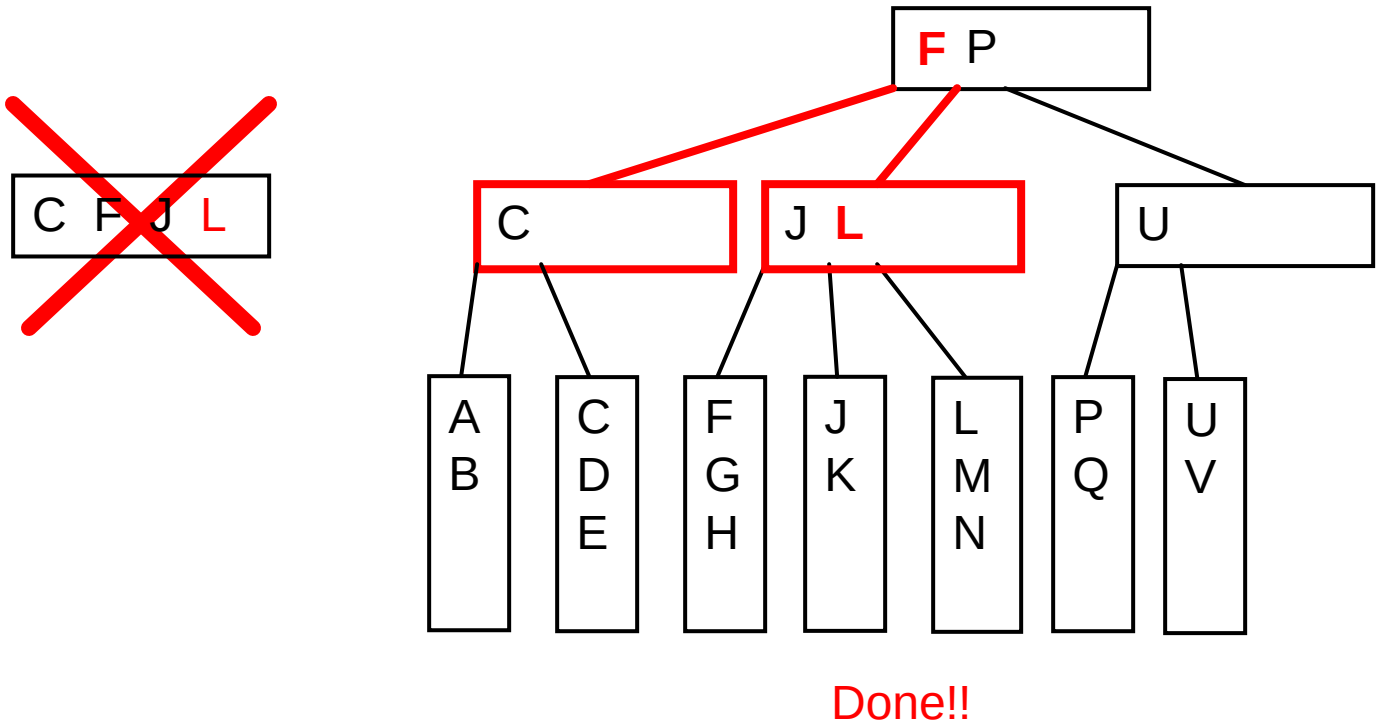
B+-Tree — Insertion Algorithm (Con't)



- Splitting internal node:
  - Cut the node out
  - Split it into 2 new internal nodes N_{left} and N_{right}
    - N_{left} has the smallest ($\lceil p/2 \rceil$-1) keys
    - N_{right} has the largest $\lfloor p/2 \rfloor$ keys
    - Note that the ($\lceil p/2 \rceil$)th key is not in either node!
      - because ($\lceil p/2 \rceil$-1) + ($\lfloor p/2 \rfloor$) = $\lceil p/2 \rceil$ + $\lfloor p/2 \rfloor$ - 1 = p - 1
  - Make the ($\lceil p/2 \rceil$)th key, say J, to be the parent of N_{left} and N_{right}
  - Insert J, together with N_{left} and N_{right}, into the original parent node

B+-Tree — Insertion Example 3 (Con't)

- Insert N



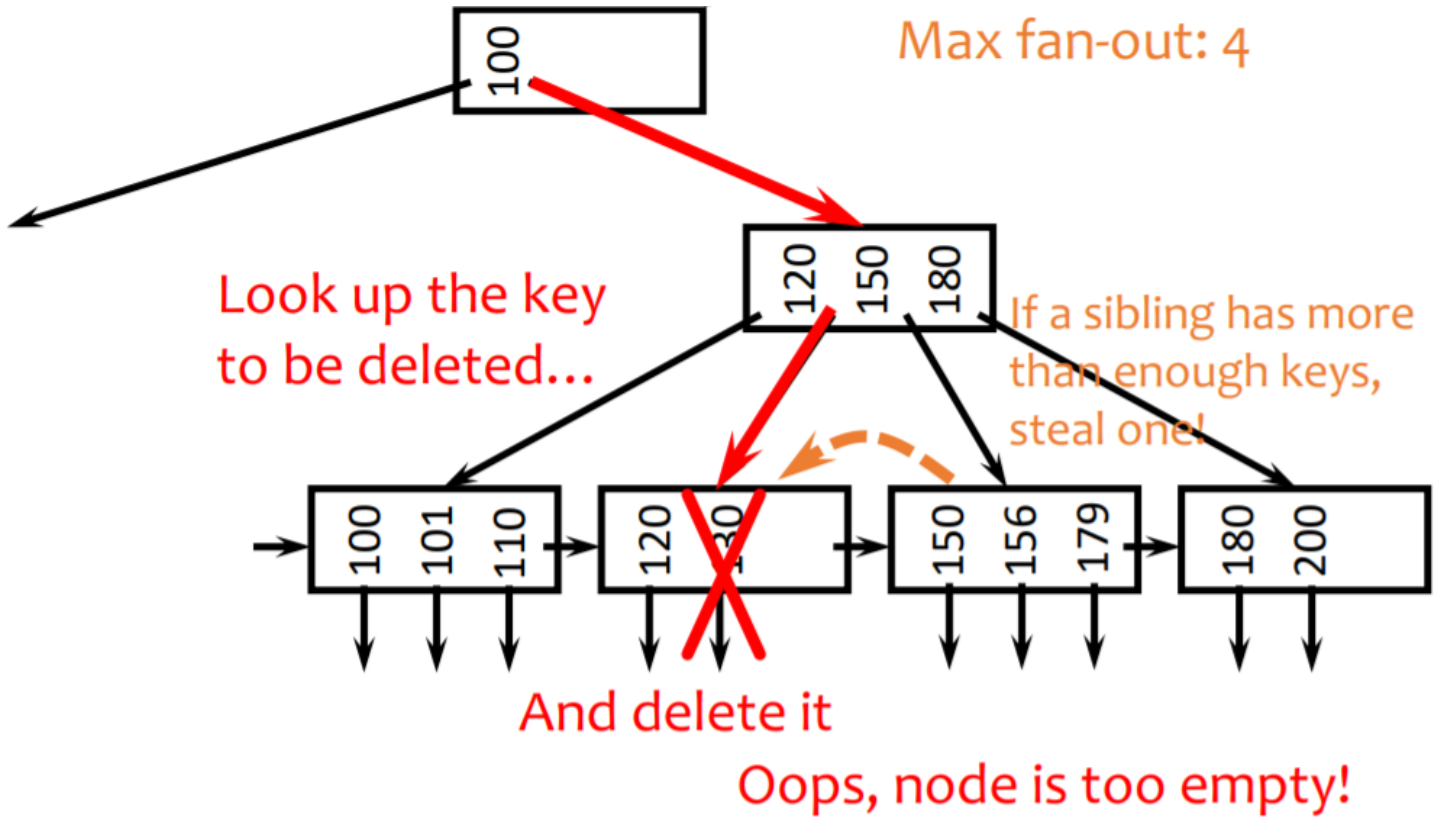
p = q = 4

B+-Tree — Insertion Algorithm (Con’t)

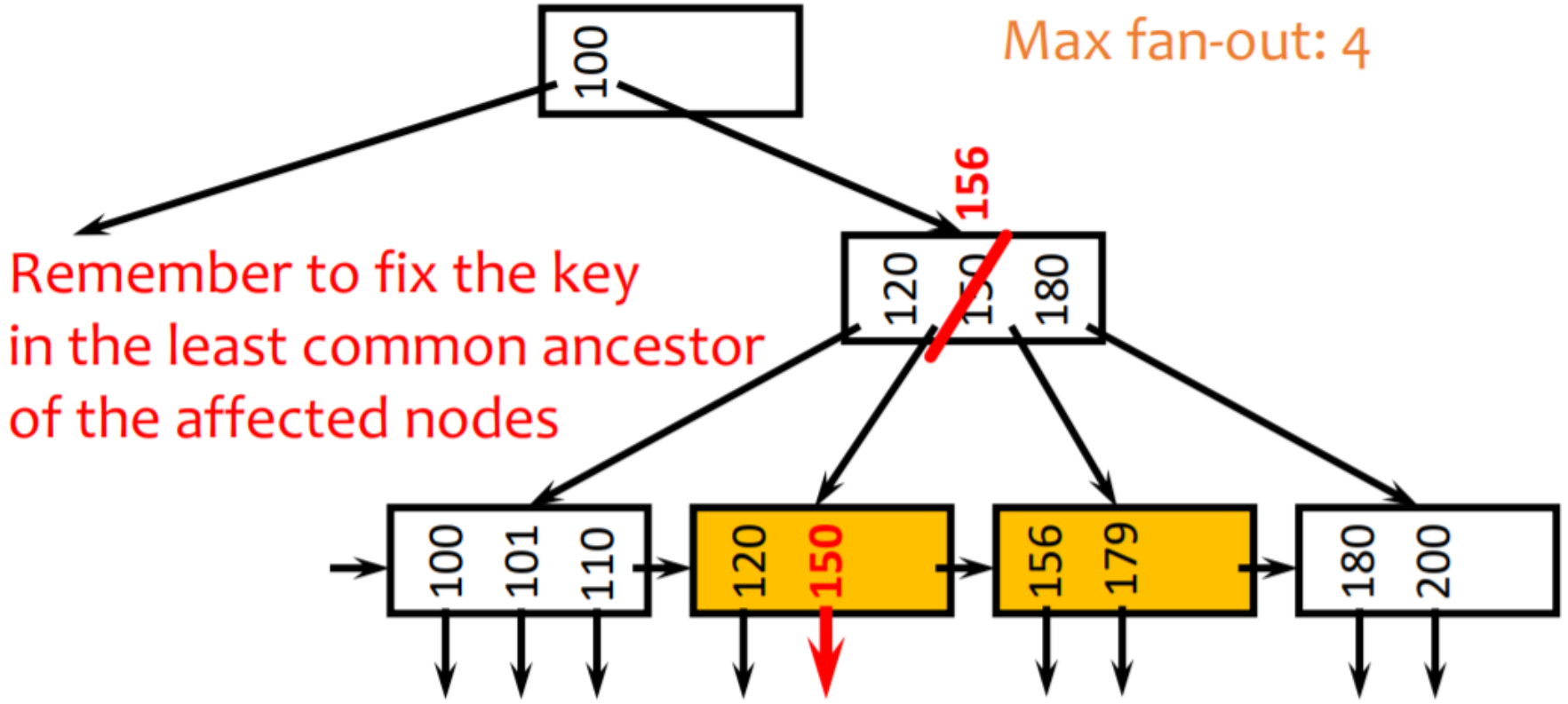
- Splitting root:
  - Follow exactly the same procedure as splitting an internal node
  - J, the parent of N_{left} and N_{right}, is now set to be the root of the tree
    - because the original root is destroyed by splitting
- After splitting the root, the depth of the tree is increased by 1

B+-Tree — Deletion

- Delete a record with search key value 130

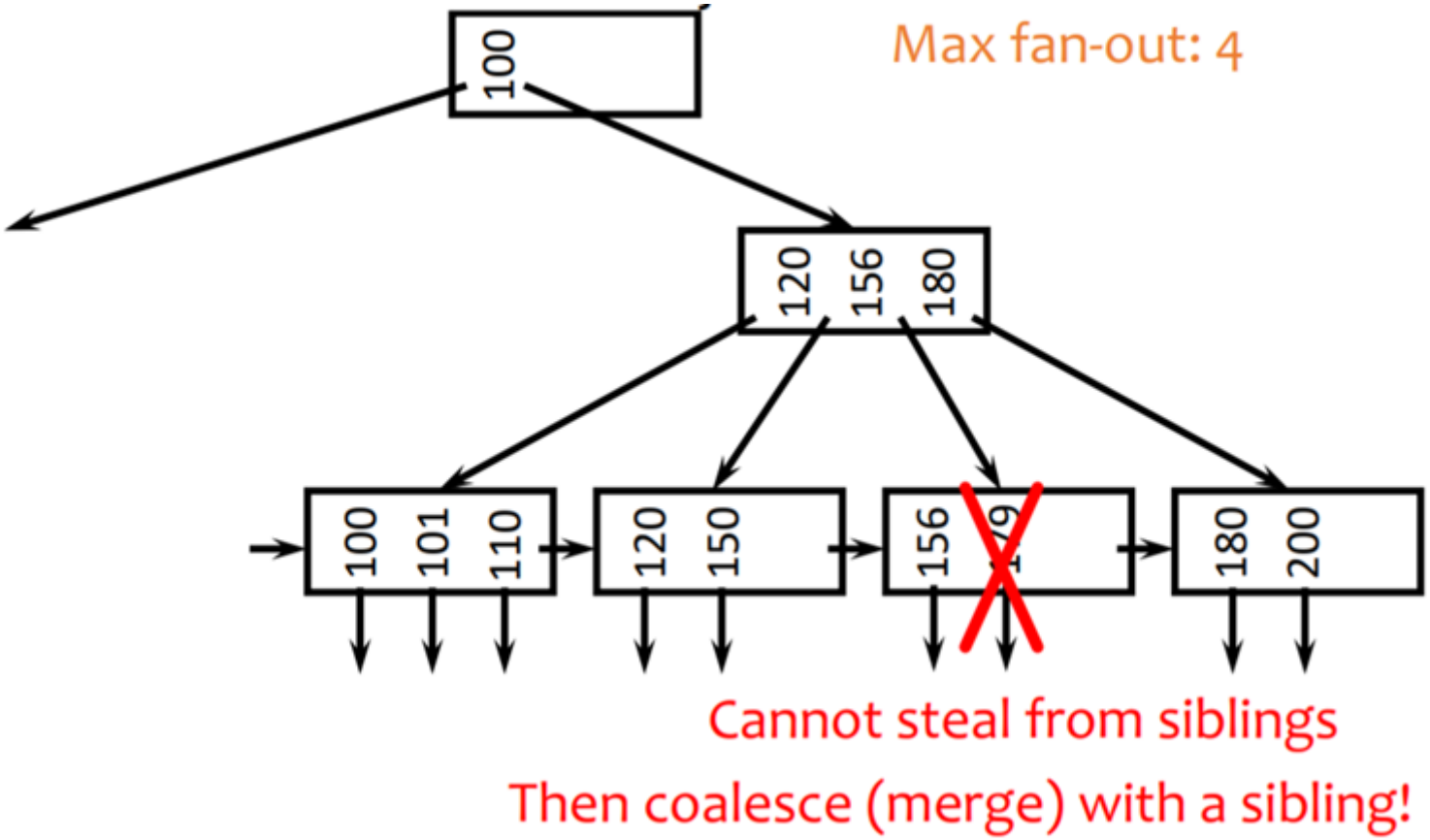


B+-Tree — Deletion (Cont'd)

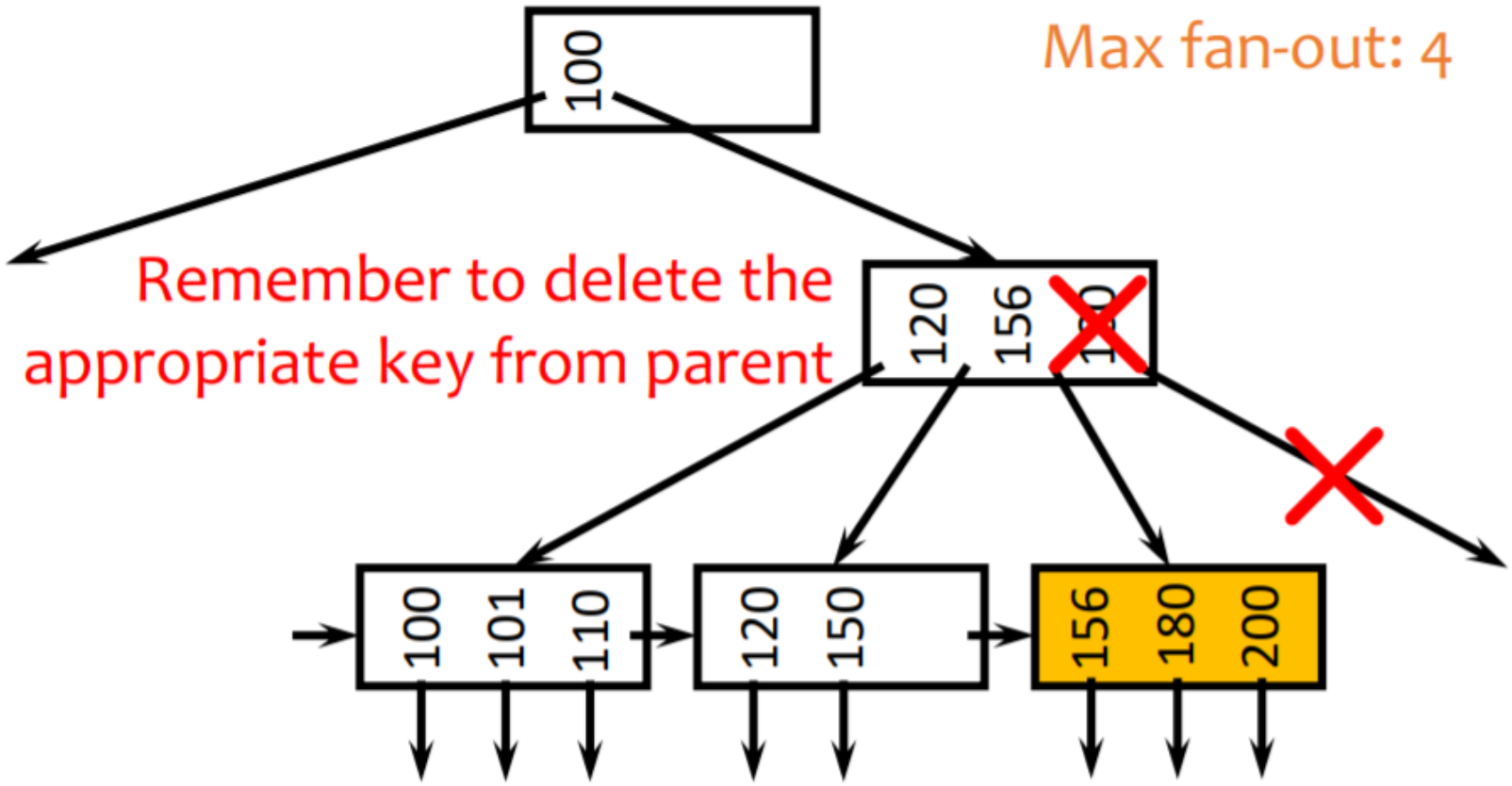


B+-Tree — Deletion (Cont'd)

- Delete a record with search key value 179



B+-Tree — Deletion (Cont'd)



Summary

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees