Advanced Database Management Systems

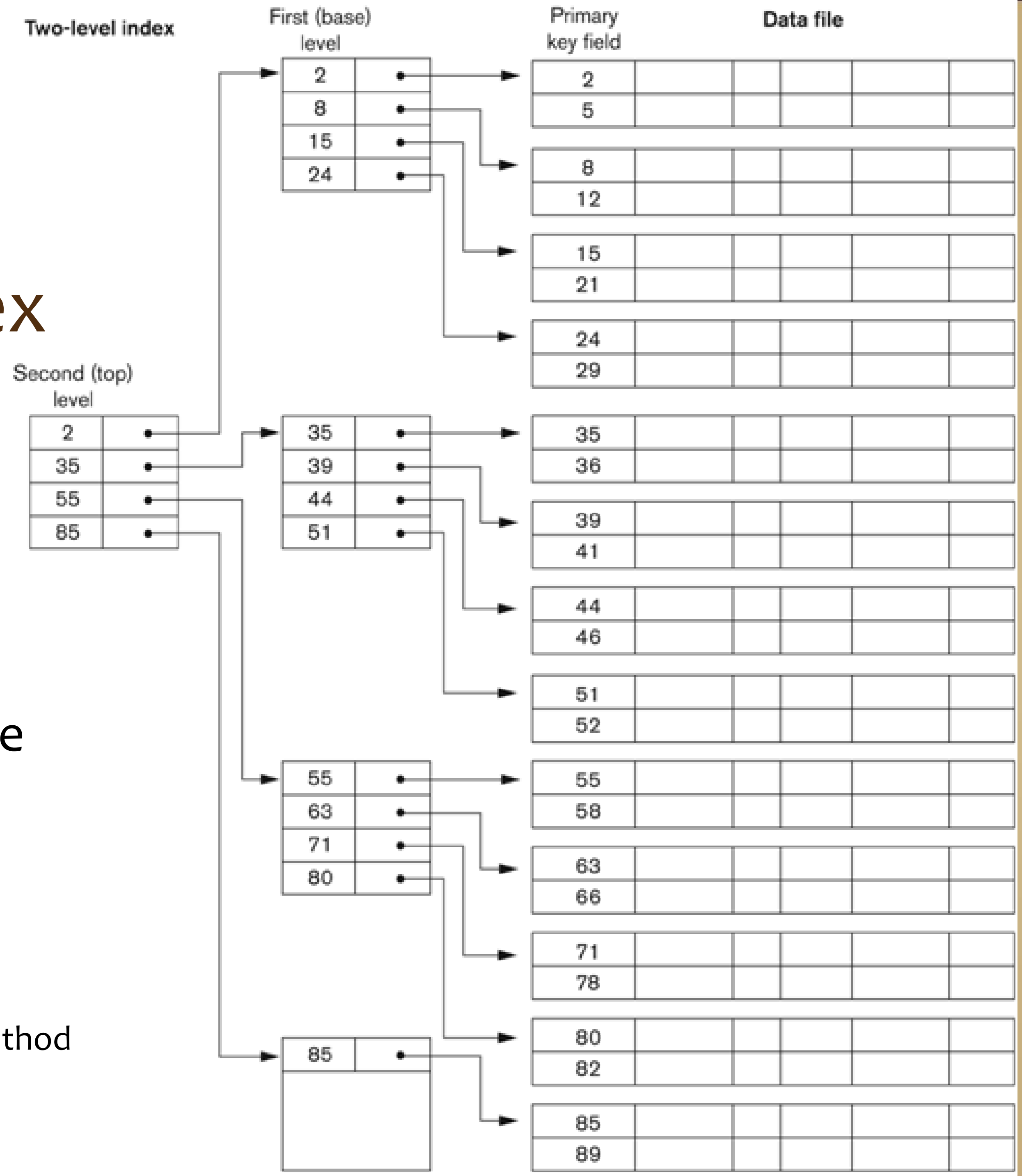
Lecture 16

Dynamic Indexes : Sections 14.3

Multi-Level Indexes

- Since 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
  - Create additional levels 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)

Two-level, static, primary index



This is similar to the ISAM organization used in early IBM systems

Index Sequential Access Method

Schematic view of multilevel index

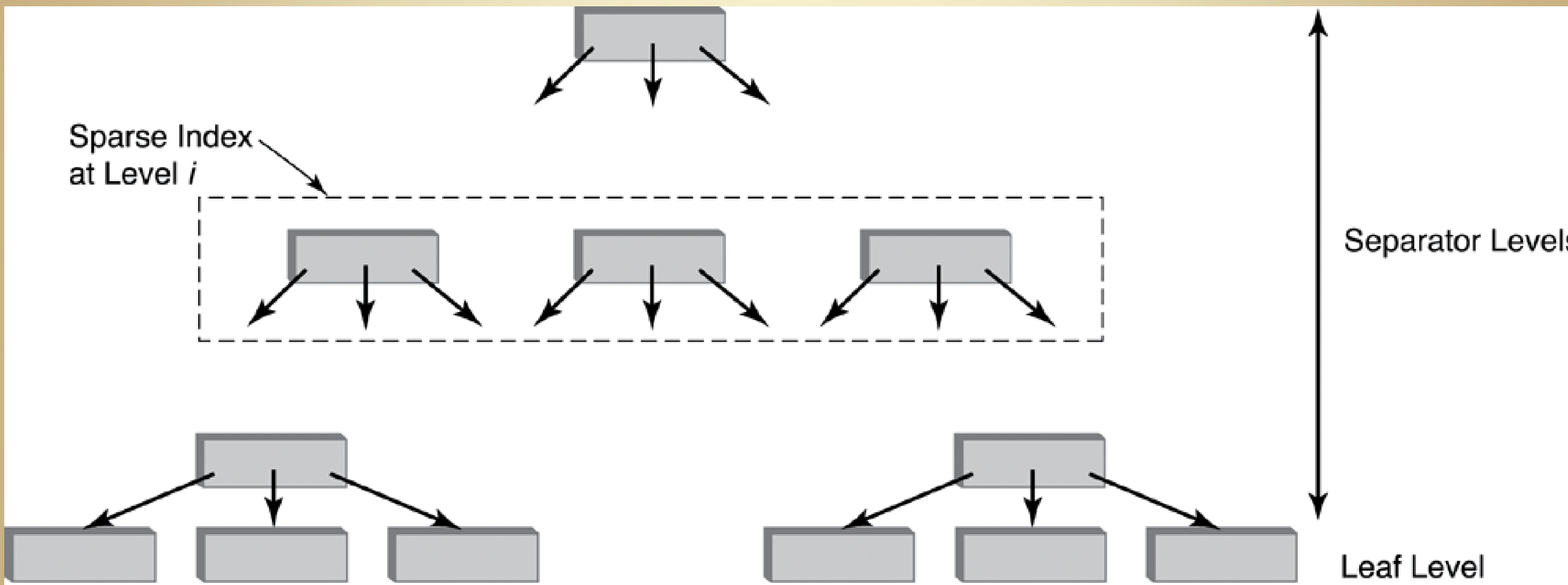
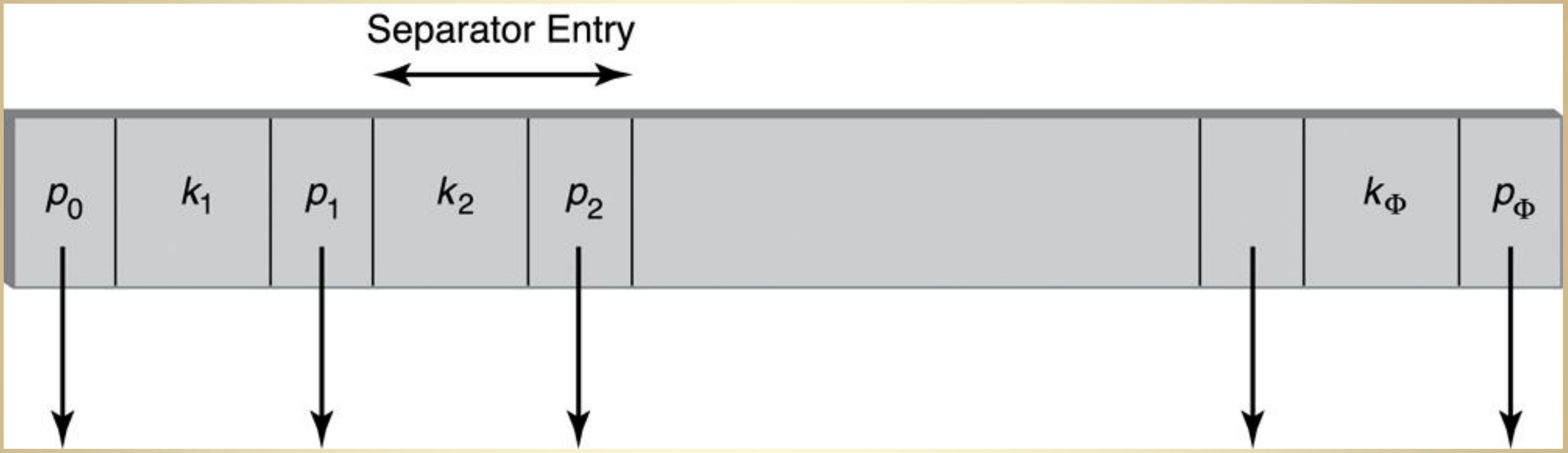
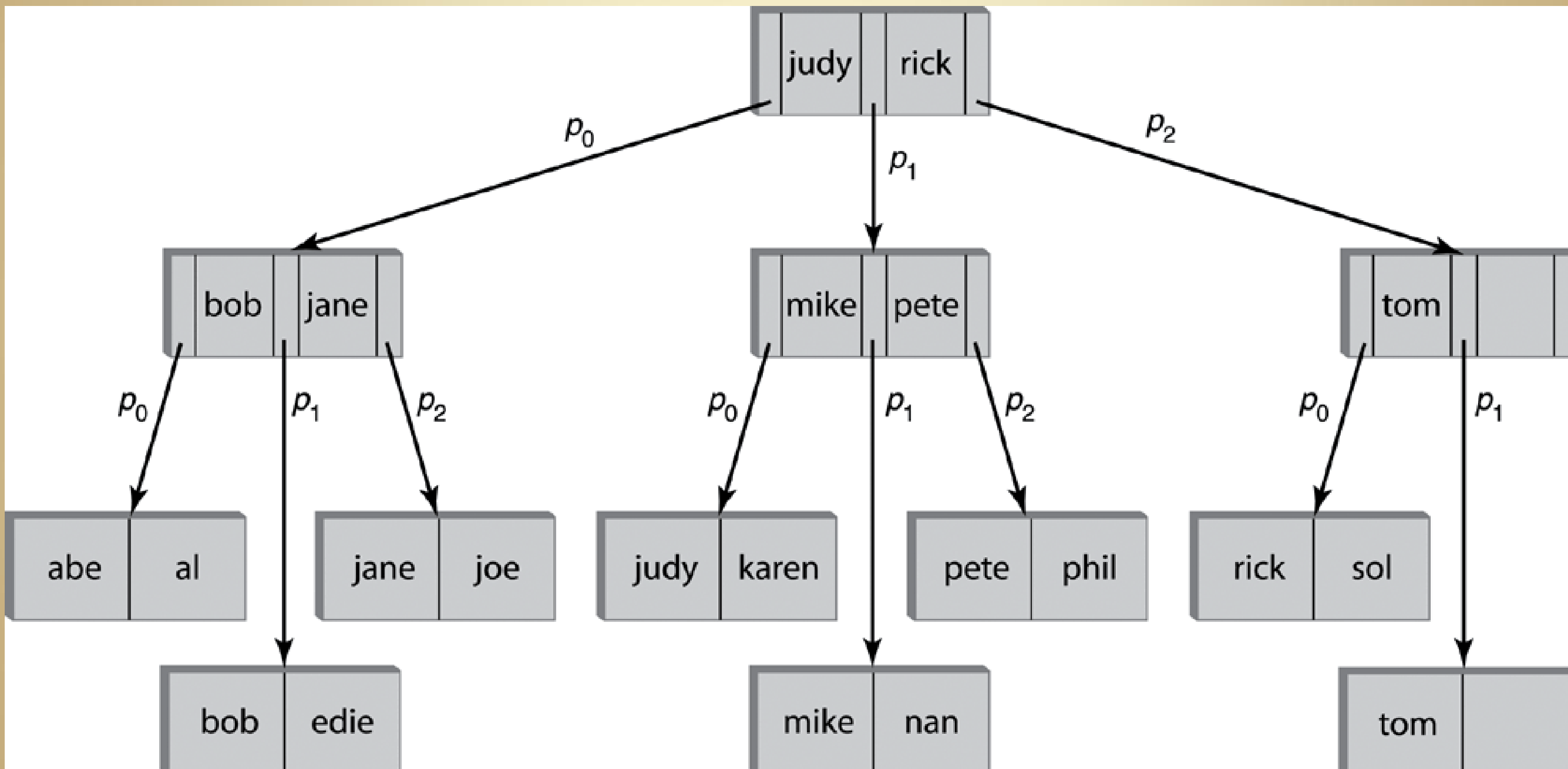


Figure 9.15 Page at a separator level in an ISAM index.



Example of an ISAM index



Dynamic Indexes

- Previous indexes are *static*
  - built up level by level from a data instance (always balanced)
  - must be rebuilt if the record set changes
- Multi-level indexes are a form of *search tree*
  - for static indexes, insertion and deletion of new index entries is a severe problem, since every level of the index is an *ordered file*.

Dynamic Indexes

- *Dynamic indexes* are modified as the record set changes

- *Dynamic indexes* are *balanced search tree* structures
  - tree nodes are sized to match block size
  - nodes are not kept in contiguous blocks
  - nodes are left partially filled to avoid extensive modification to the tree when records are inserted
  - insert and delete algorithms are designed to keep tree balanced
  - balanced = all leaf nodes at same level

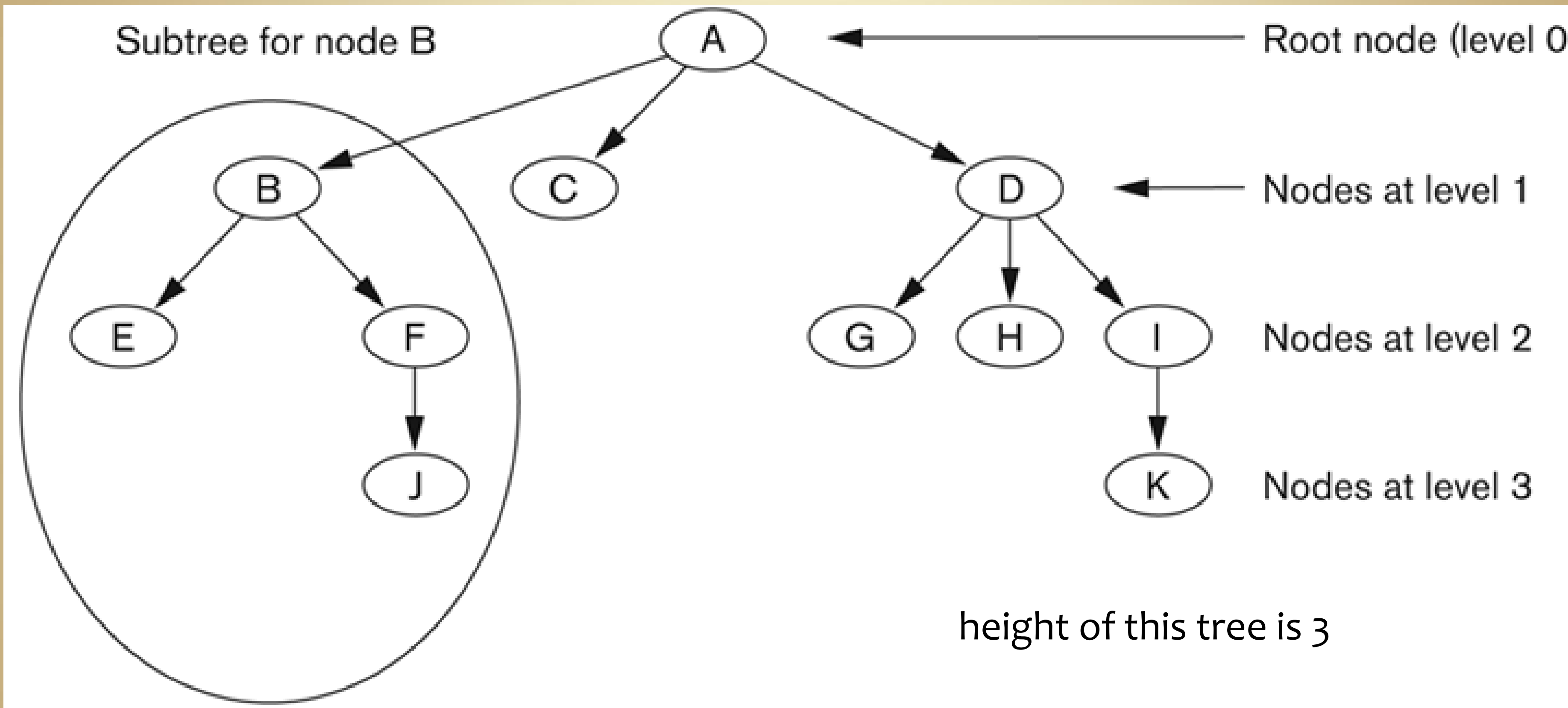
Trees: Review

- A tree is a rooted, directed, acyclic graph

- every node has 0-1 parents and 0-*p* children
  - *p* is the order or fan-out of the tree
- *root* node has no parent: there is exactly one root
- *leaf* nodes have no children
- *interior* nodes have at least one child
- *height* of a tree is length of longest path
  - (from root to some leaf)

- binary trees (order 2) are typical
  for in-memory algorithms
- order of disk-based trees is selected
  to match node size to disk block size

An Unbalanced Tree

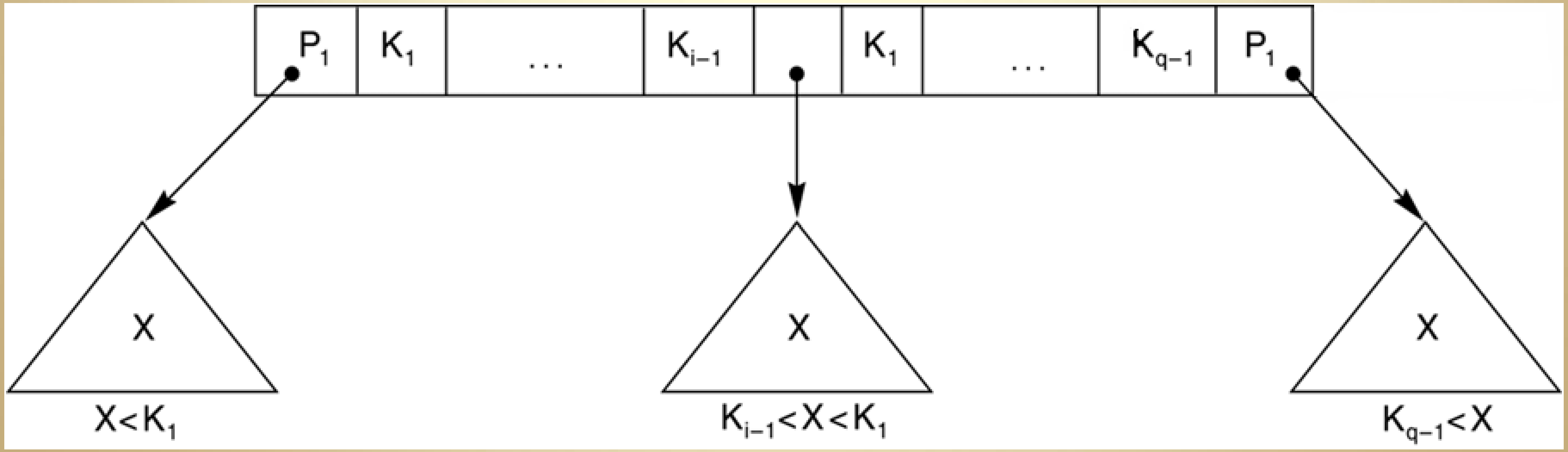


balanced tree = path length from root to any leaf node is the same

Search Tree: Basic Structure

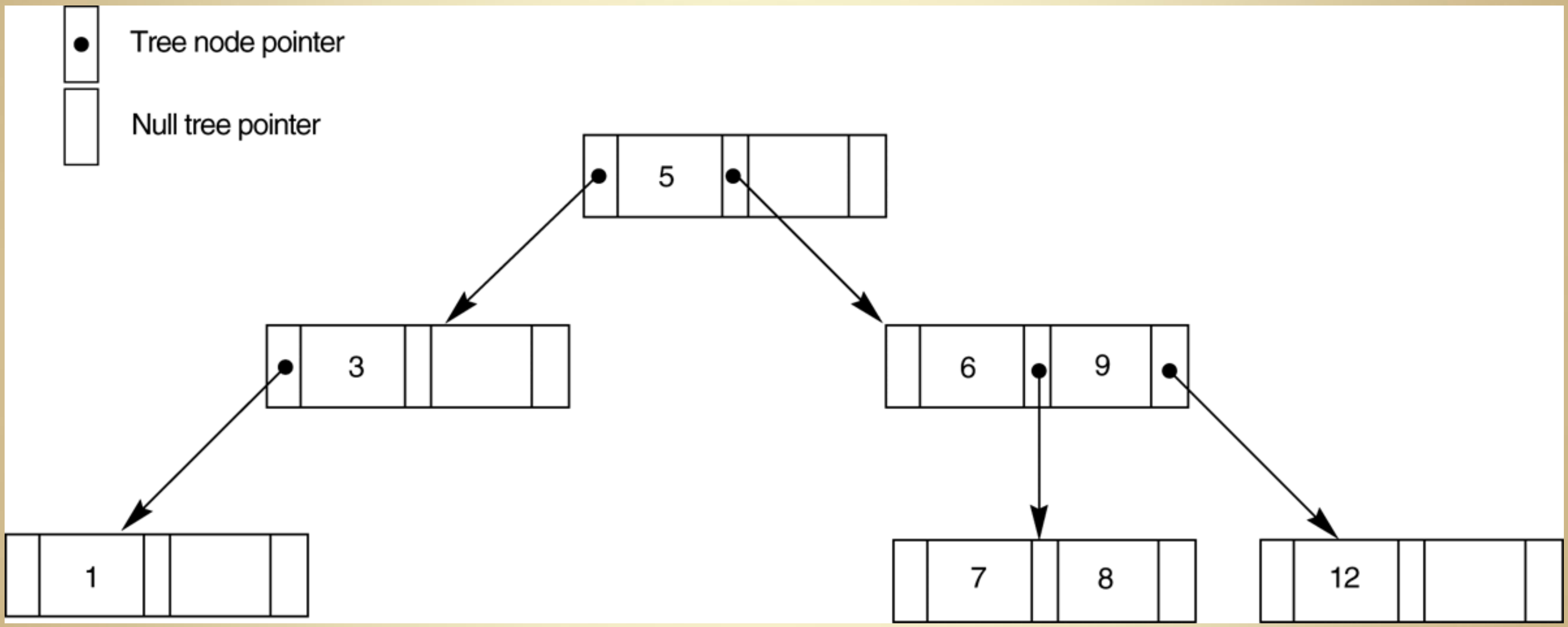
- Disk based trees have a large fan-out (order)
- Maximize order such that nodes fit in disk blocks/pages

- Following node has order *q*



A node with order *q*, will have *q-1* separator values

Search Tree, Order = 3



Is this a balanced tree?

B-Trees and B+-Trees

- Most multi-level indexes
  use *B-tree* or *B⁺-tree* data structures
  - Efficient insertion and deletion
  - Each tree node corresponds to a disk block
  - Update algorithms maintain balanced tree
  - Nodes are kept between half-full and completely full
    to allow for new index entries

Updates to B-Trees & B+-Trees

- Insertion into a node that is not full is quite efficient
- Insertion into a full node
  causes a split into two nodes
  - Splitting may propagate to other tree levels

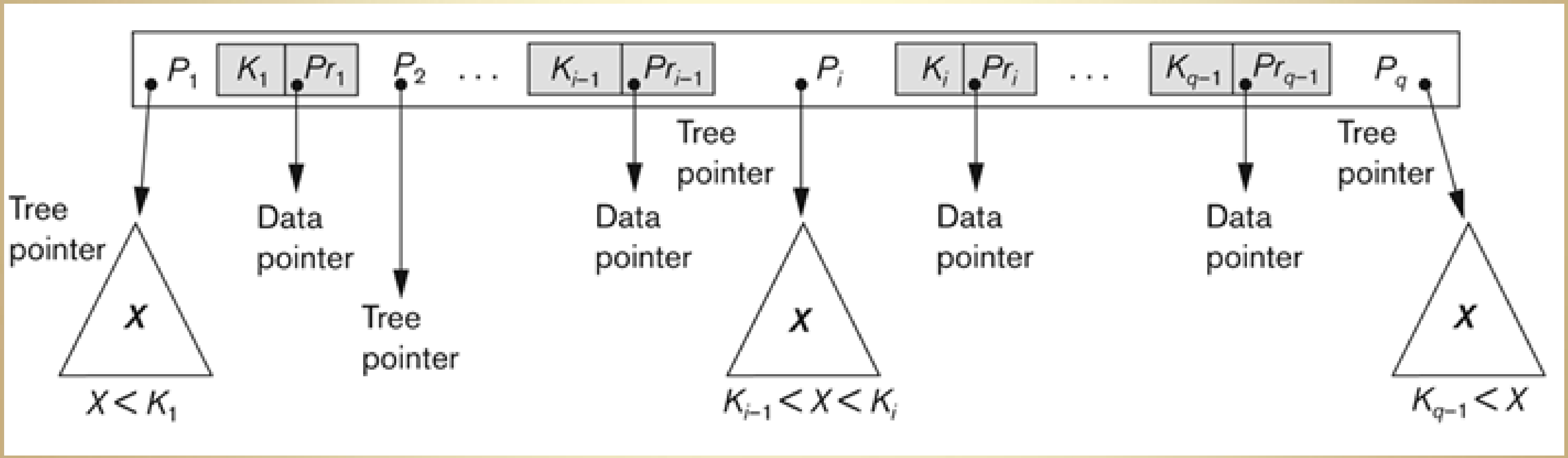
- Deletion into from a node that
  is more than half full is quite efficient
- When deletion causes a node to become
  less than half full, it must be merged with
  neighboring nodes
  - merging may propagate to other tree levels

B-trees

- B-trees have a single node type
  - interior and leaf nodes have the same structure
  - interior nodes have keys,
    subtree pointers and data pointers
  - leaf nodes have keys and data pointers,
    all subtree pointers are NULL
  - **key values appear in exactly one node**

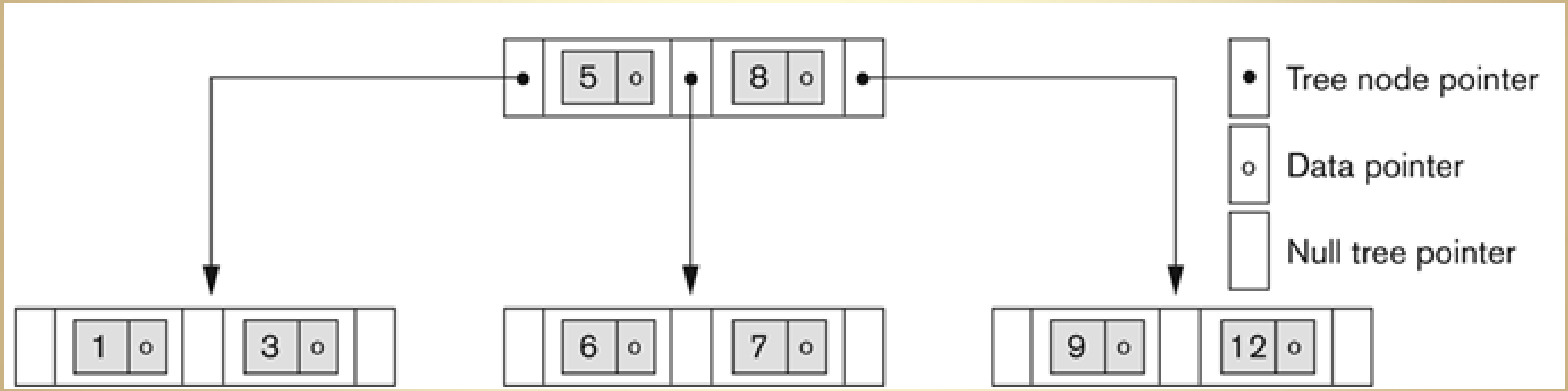
B-tree Structure (order *q*)

B-tree node with *q-1* search values and *q* pointers



Example B-tree

A B-tree of order 3



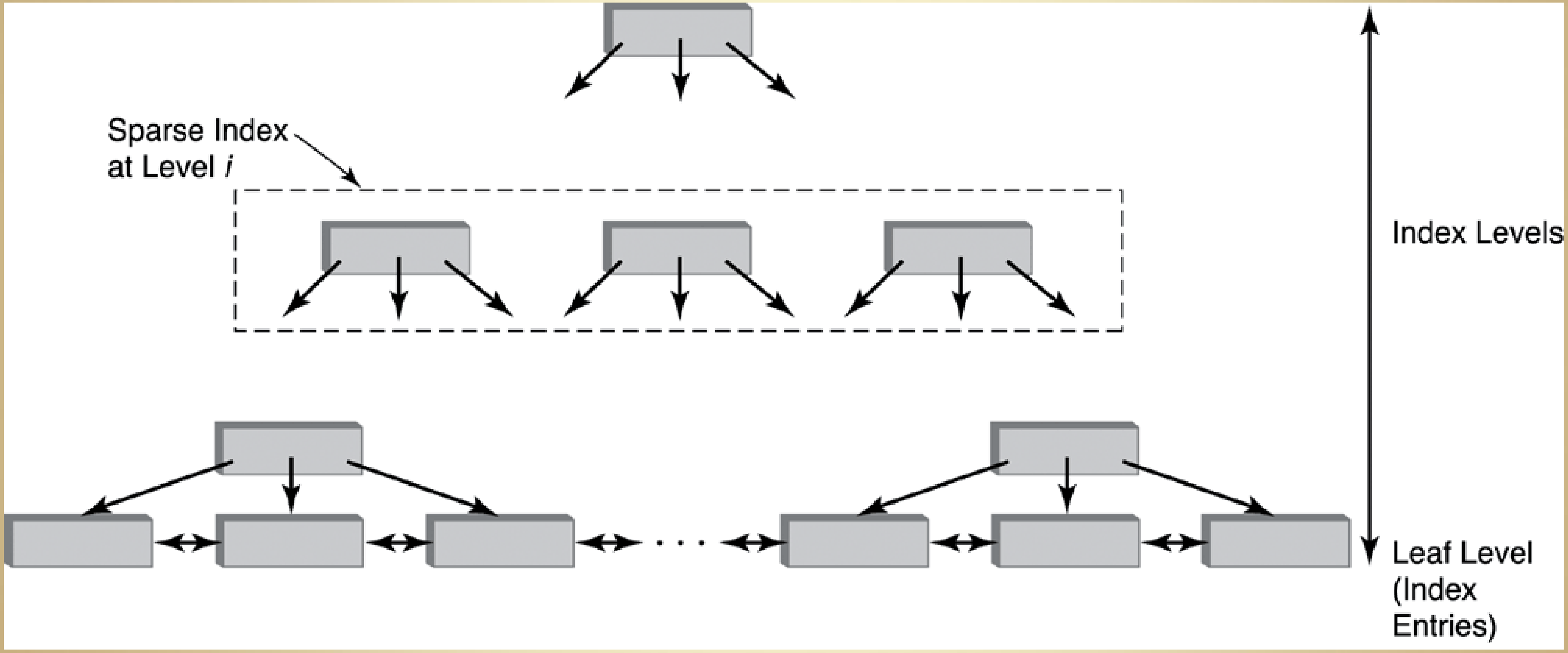
B-tree, B^+-tree Comparison

- B-tree:
  - pointers to data records exist at all levels of the tree
  - each index value appears in exactly one node
- B^+-tree:
  - pointers to data records exists only in the leaf nodes
  - each index value appears in exactly one leaf node
  - some index values also appear in interior nodes

B^+-Trees

- Support equality and range searches, multi-attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure
- Responds to dynamic changes in the table
- B^+-trees have two kinds of nodes
  - *interior* nodes contain keys and *subtree* pointers
  - *leaf* nodes contain keys and *data* pointers
  - key values may appear in multiple nodes

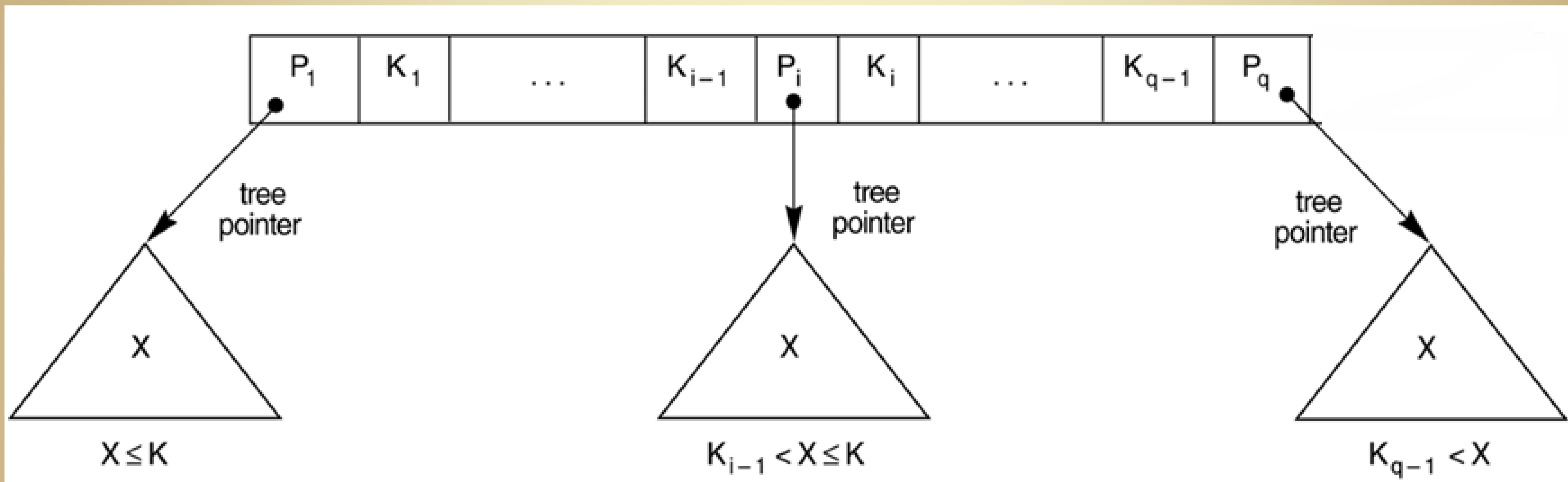
B+ Tree Structure



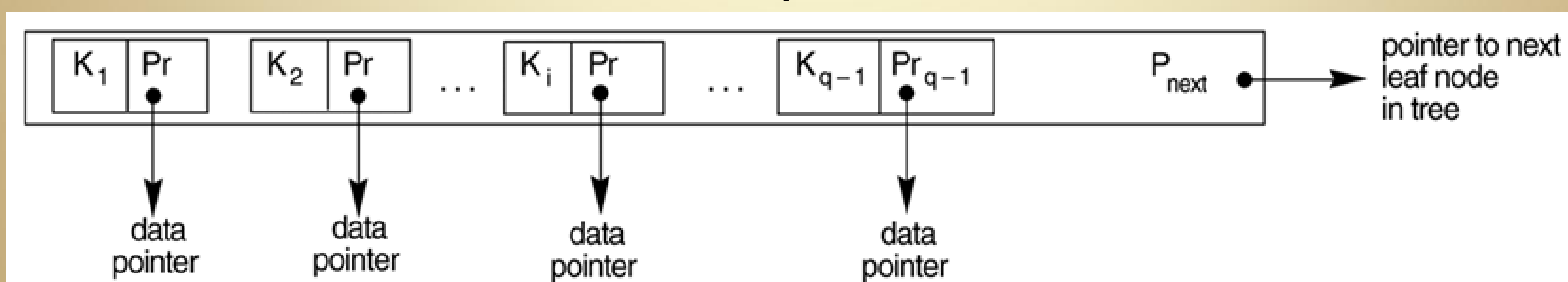
– Leaf level is an ordered linked list of index entries
– All data pointers are in leaves, interior nodes have sub-tree pointers
– Sibling pointers support range searches

B⁺-tree Nodes (Order = *q*)

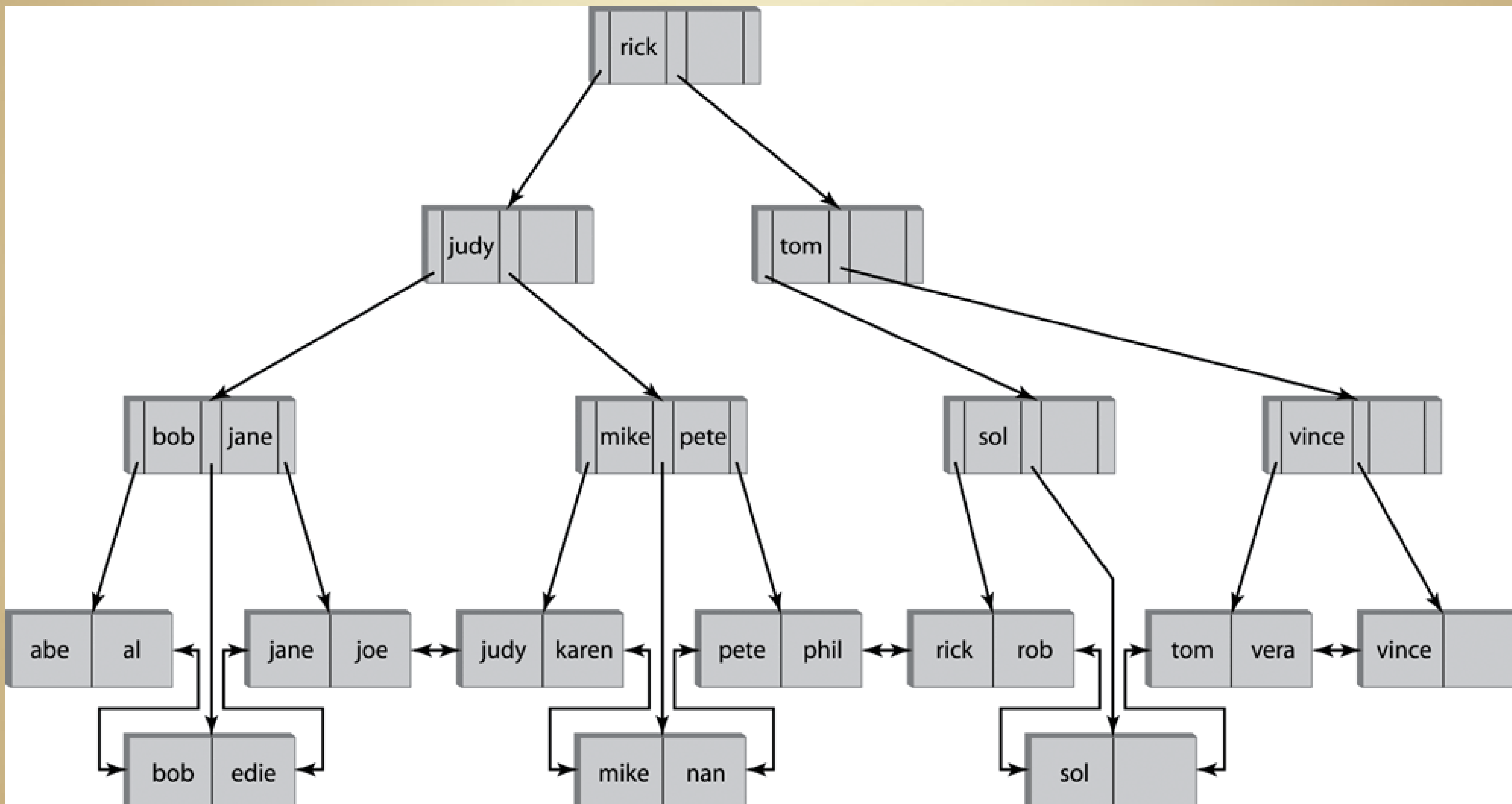
Internal node: *q–1* search values, *q* subtree pointers



Leaf node: *q–1* search values, *q–1* data pointers,
1 or 2 next node pointers



Example B+-tree (order 3)



not shown: each entry in leaf level has a pointer to the data file

Order of a B+-tree

- Let B = block size, P = block pointer size and V = index value size.

$$(p-1)V + pP < B$$
$$p(V+P) - V < B$$
$$p(V+P) < B + V$$

$$p < \frac{B+V}{V+P}$$

$$\boxed{p = \left\lfloor \frac{B+V}{V+P} \right\rfloor}$$

- B+-tree nodes are deliberately kept partially filled
  - $q < p$ is the *effective fan-out* of a node
  - q is typically defined as some *fill factor*, for example: $q = 0.7p$

B^+-tree Size

- To estimate the size of a B^+-tree:
  - determine *p*, from block size, pointer size and key size
  - determine *q*, from *p* and desired fill factor
  - determine number of leaf nodes, r_{leaf}, using *q-1* and number of data records (*r*)
  - determine number of interior nodes, r_i, at each level using number of blocks in lower level (r_{i-1}) and *q*
  - When $r_i = 1$, you've reached the root. The tree height is $h = i$

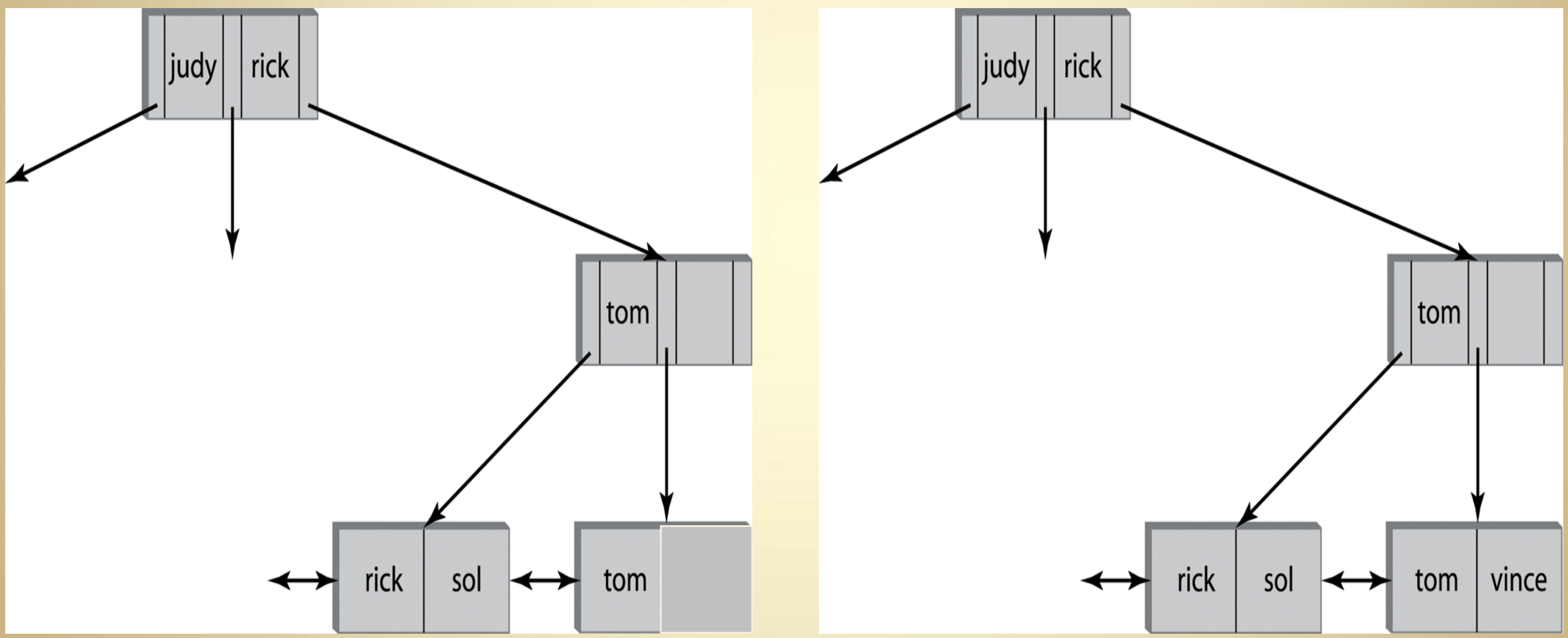
$$r_0 = r_{leaf} = \left\lceil \frac{r}{q-1} \right\rceil$$

$$r_i = \left\lceil \frac{r_{i-1}}{q} \right\rceil$$

  - Blocks required to store the tree is $\sum_{i=0}^{h} r_i$

B+-tree Insertion Example

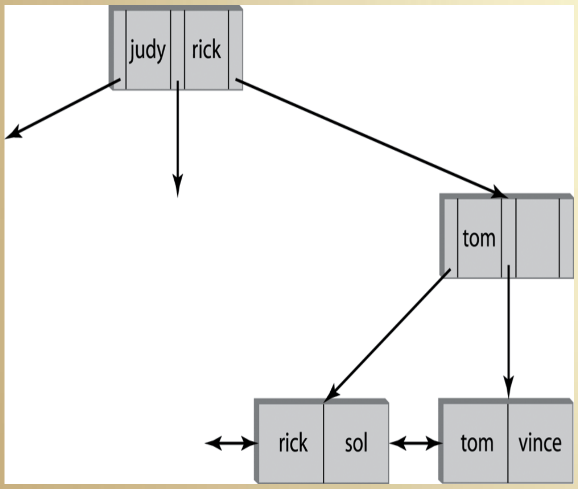
insert “vince”

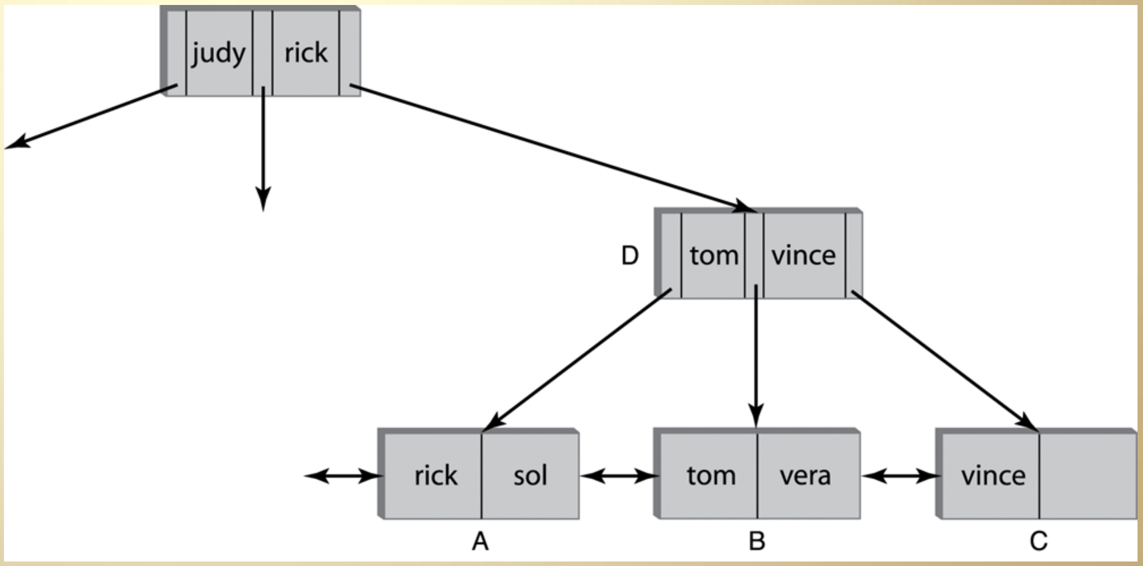


B+-tree Insertion Example

Insert “vera”: Since there is no room in leaf page:

1. Create new leaf, C
2. Distribute index entries between B and C (maintain order)
3. Adjust parent node D

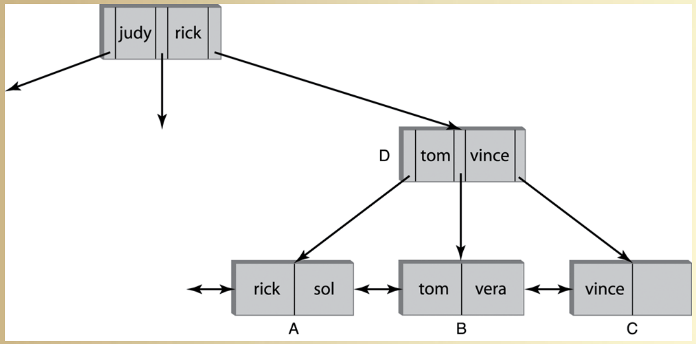


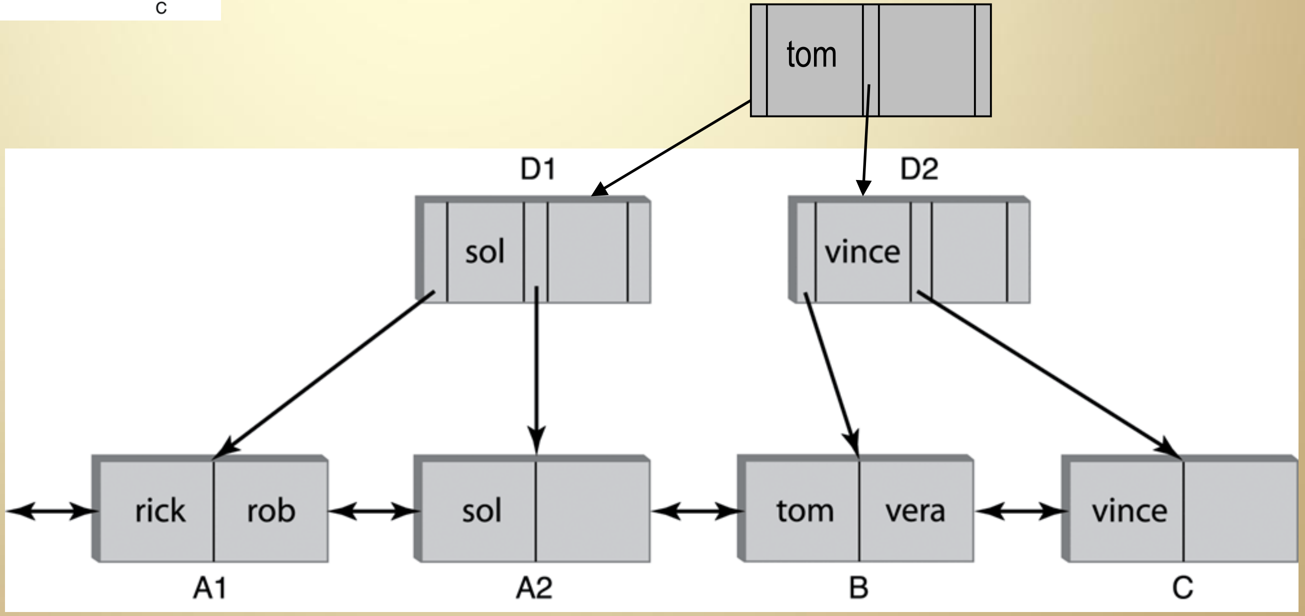


B+-tree Insertion Example

Insert "rob". Since there is no room in leaf A:

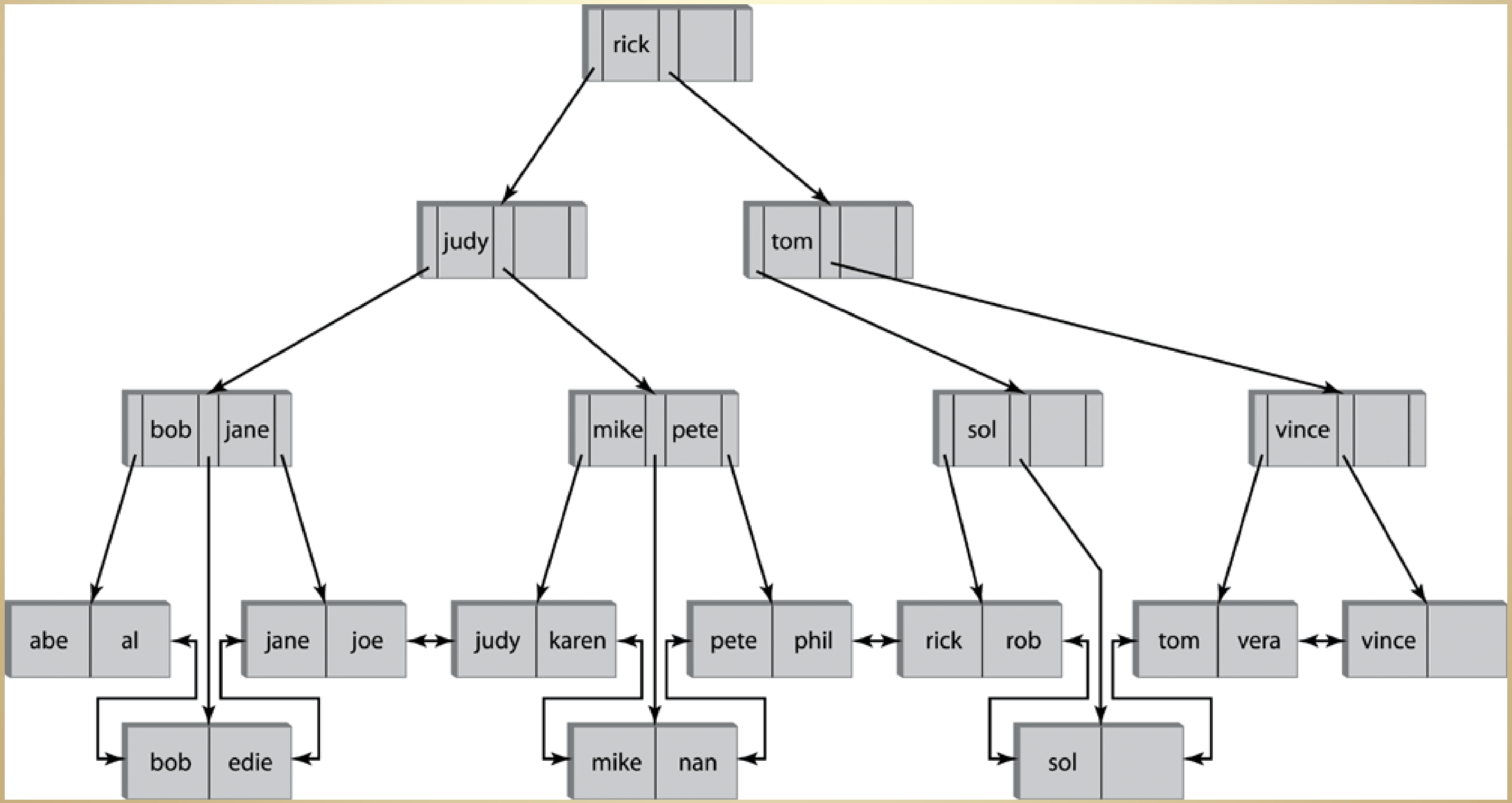
1. Split A into A1 and A2 and distribute index entries between the two (maintain order)
2. Split parent D into D1 and D2 to make room for additional pointer
3. Add new level to connect D1 and D2



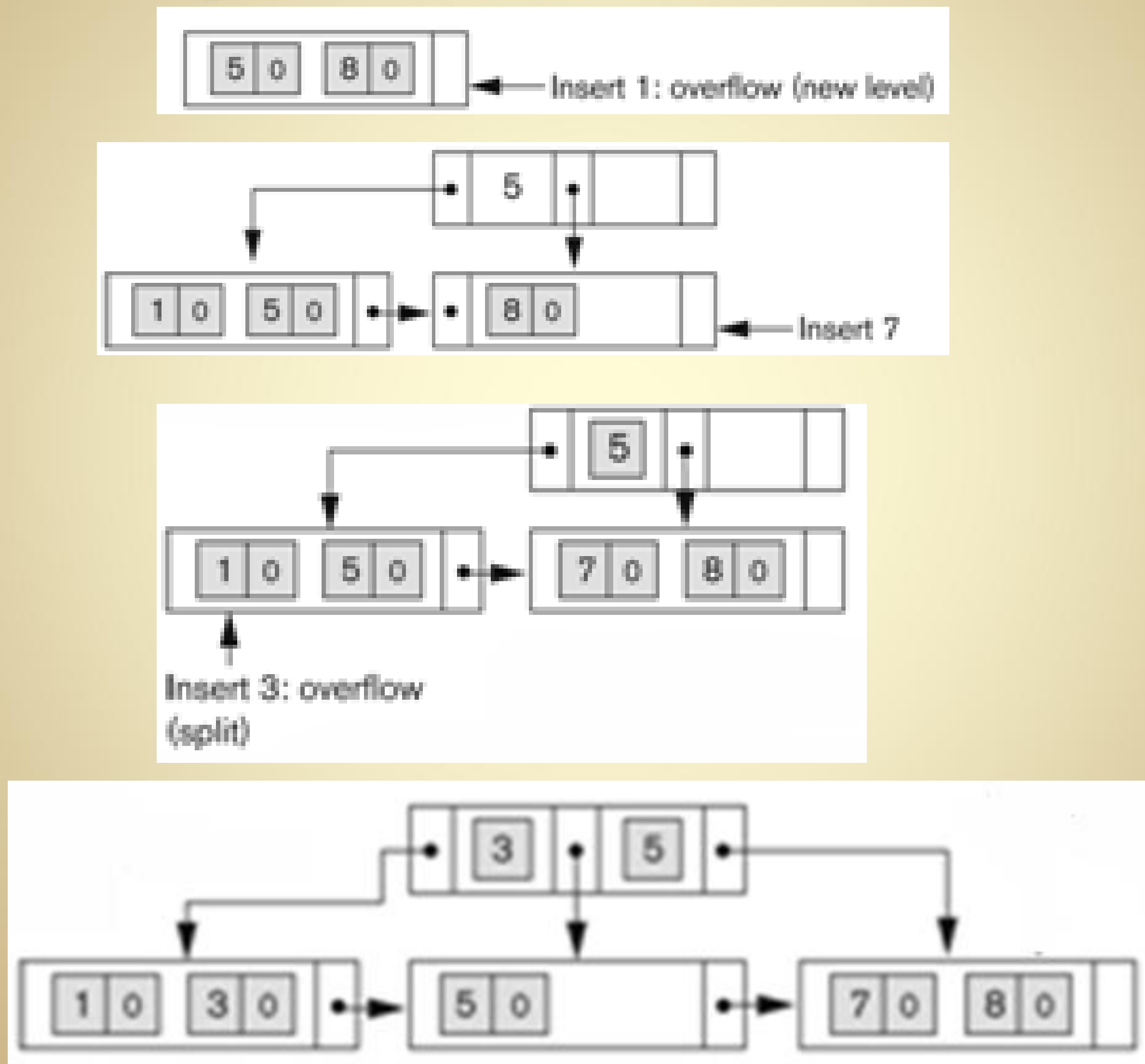


B⁺-tree Insertion: Split Propagation

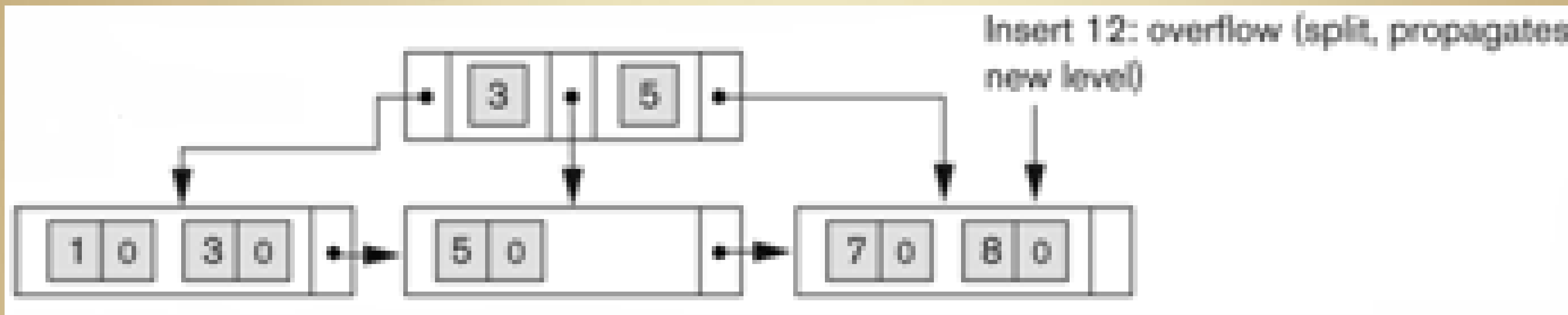
- When splitting an interior node, push a separator up
- Repeat process at next level
- Height of tree increases by one

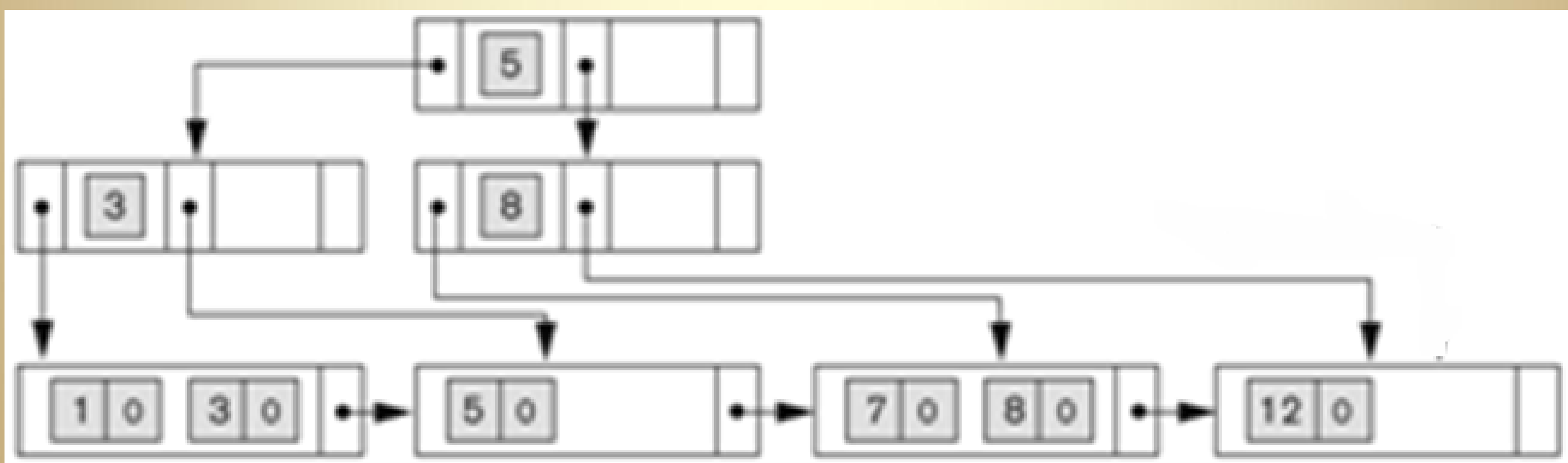


Example: B+-tree Insertion

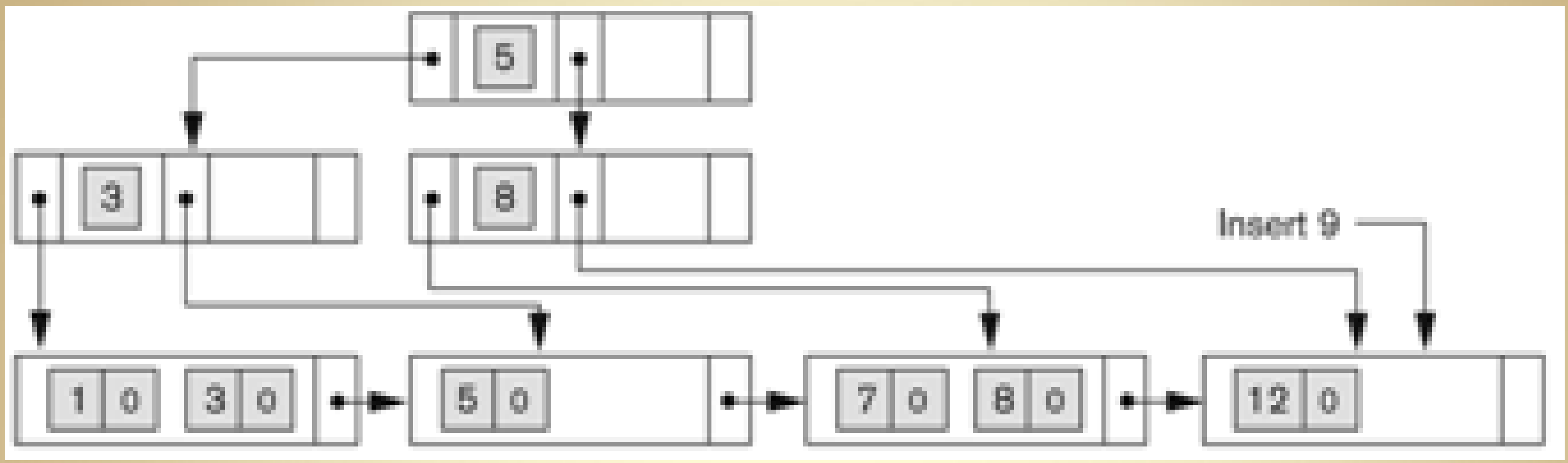


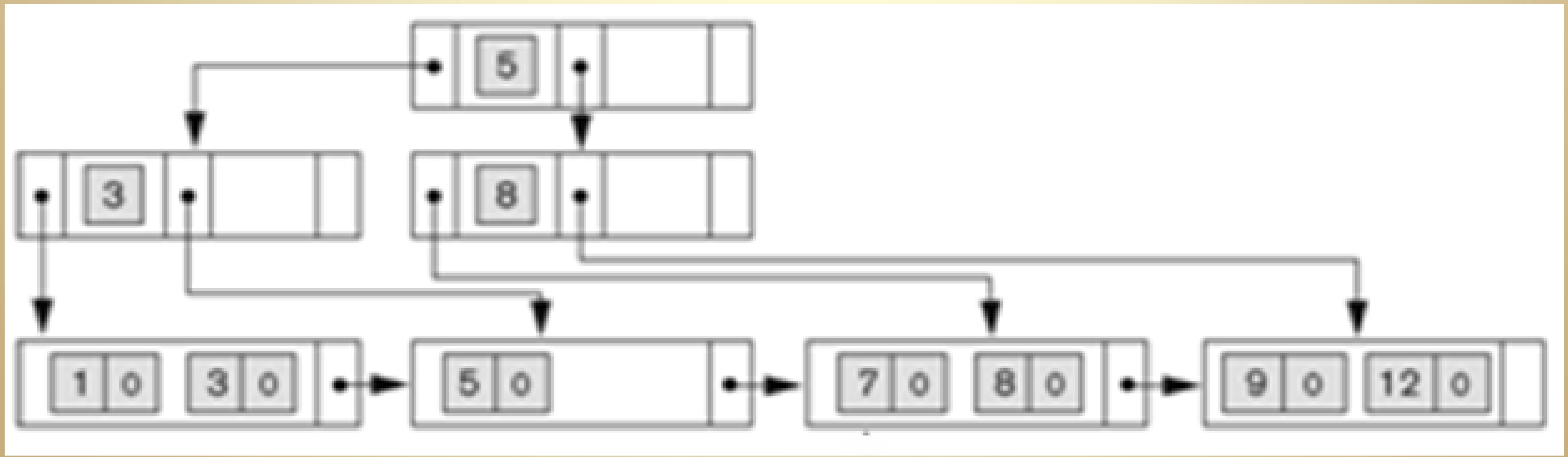
Example: B⁺-tree Insertion



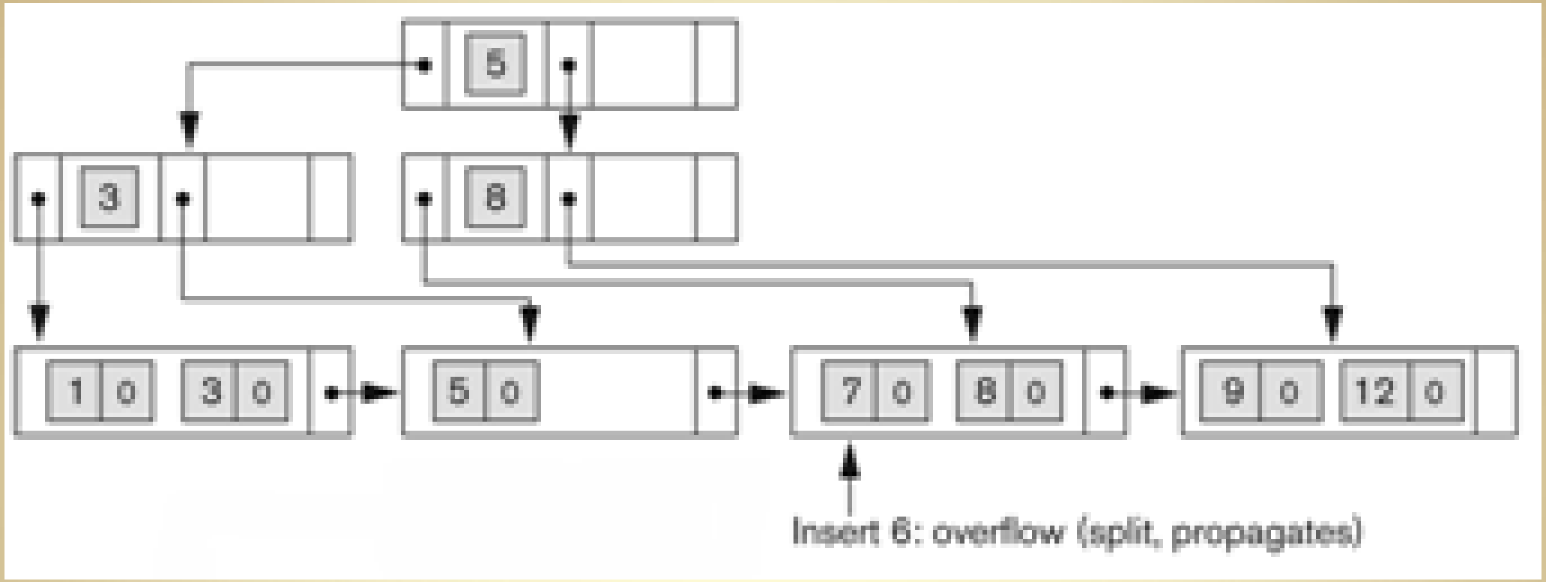


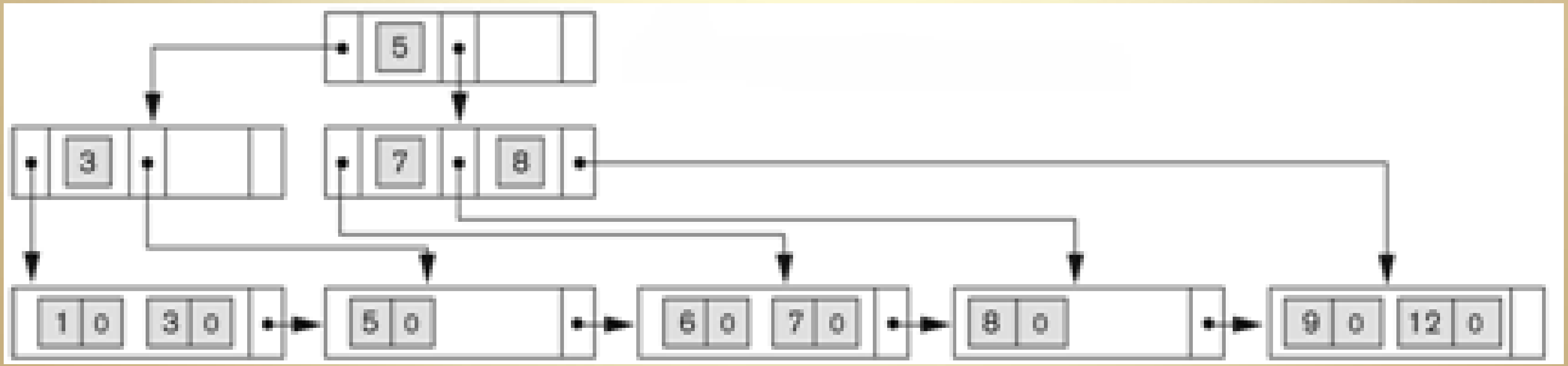
Example: B+-tree Insertion



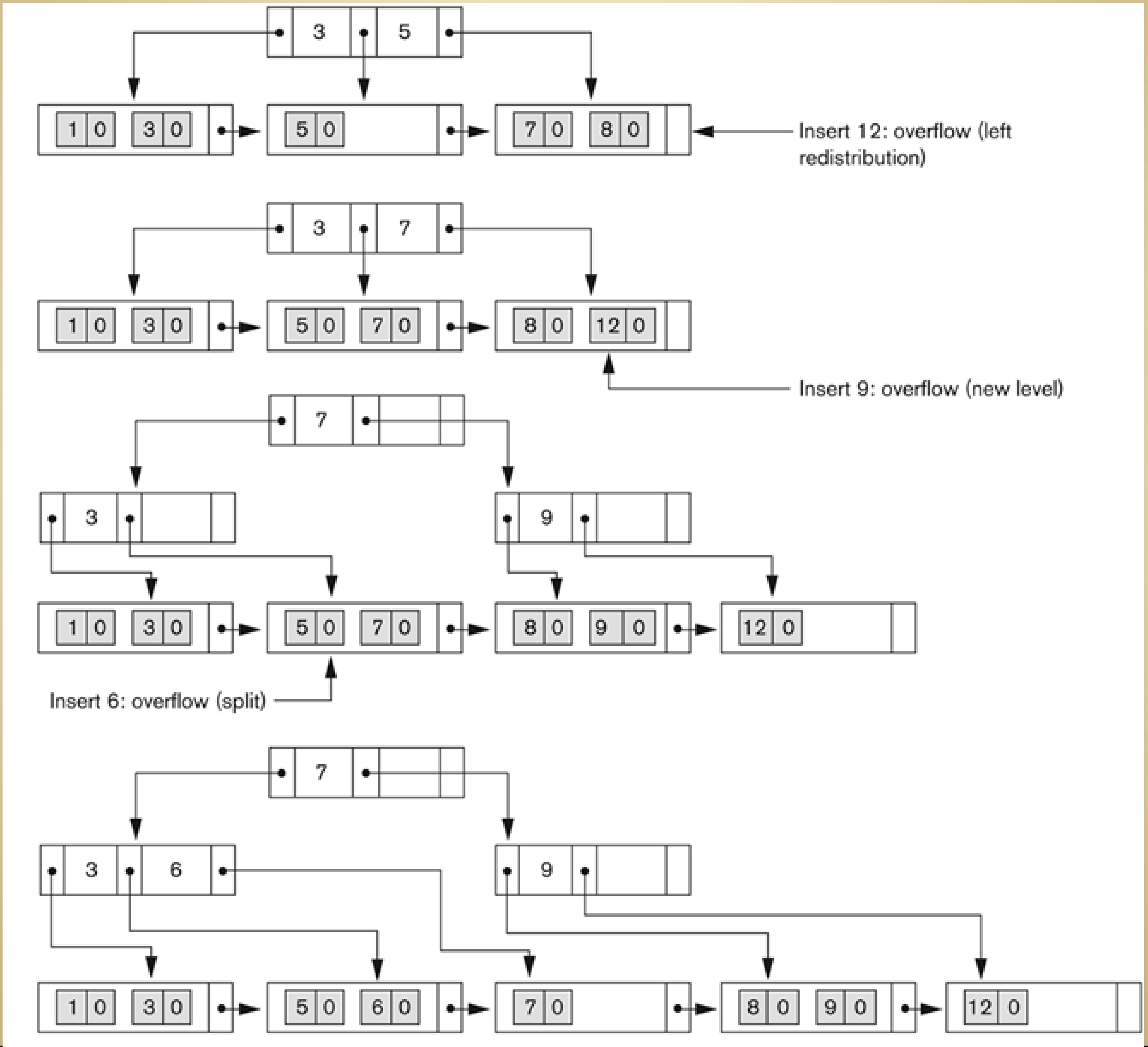


Example: B+-tree Insertion





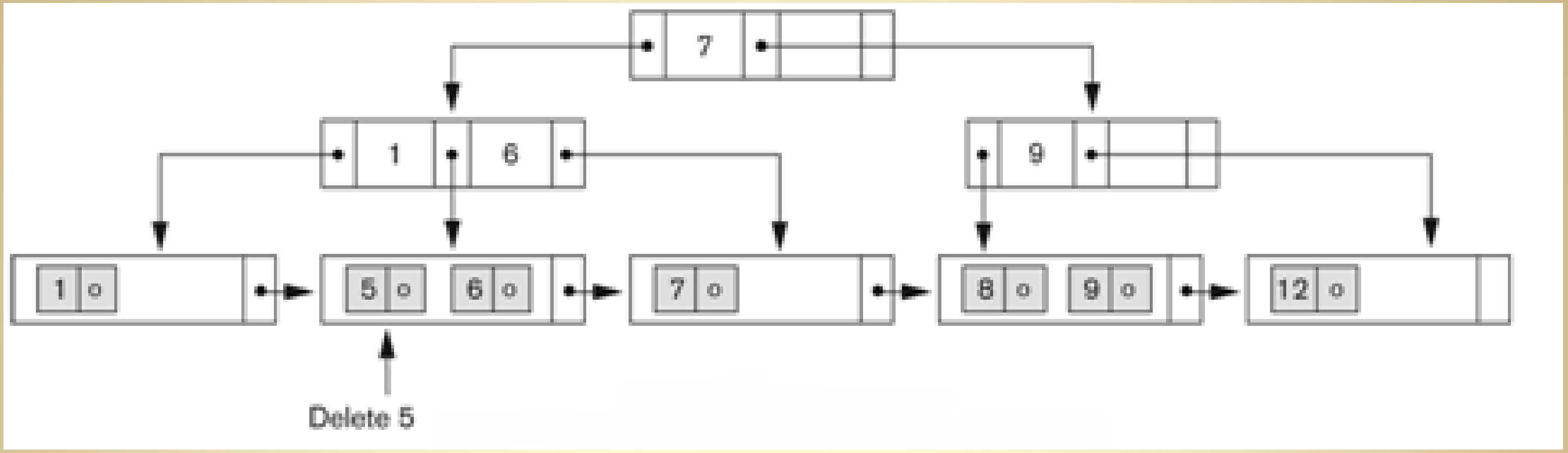
Example: B+-tree Insertion

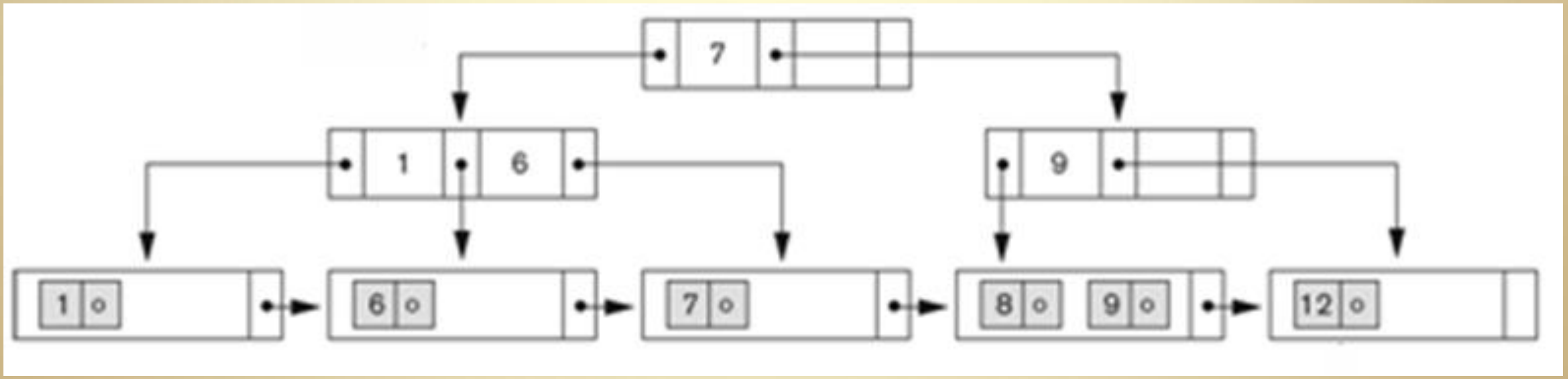


Handling Deletions

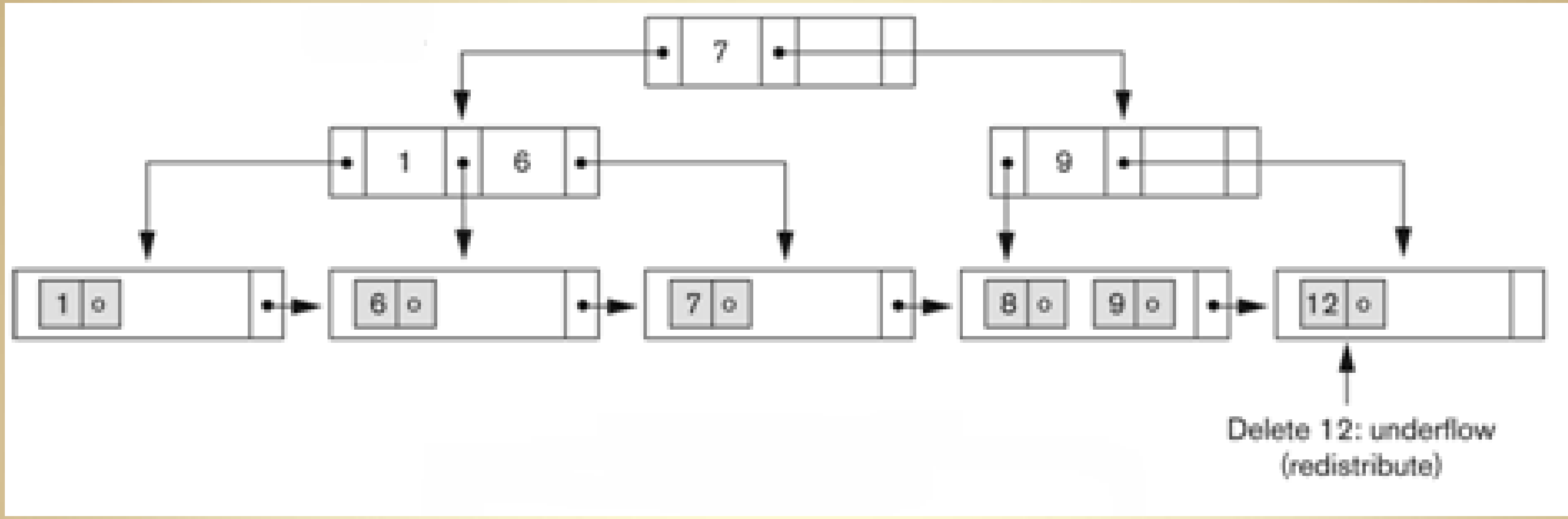
- Deletion can cause page to have fewer than q entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
- In practice, tables generally grow, and merge algorithm is often not implemented
  - Reconstruct tree to compact it

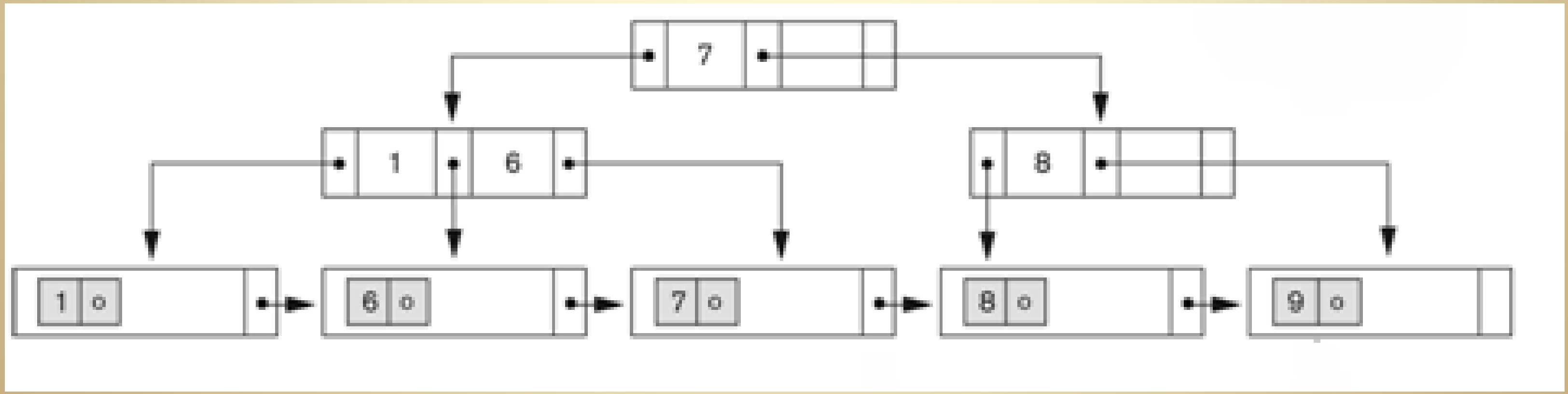
Example: B$^+$-tree Deletion



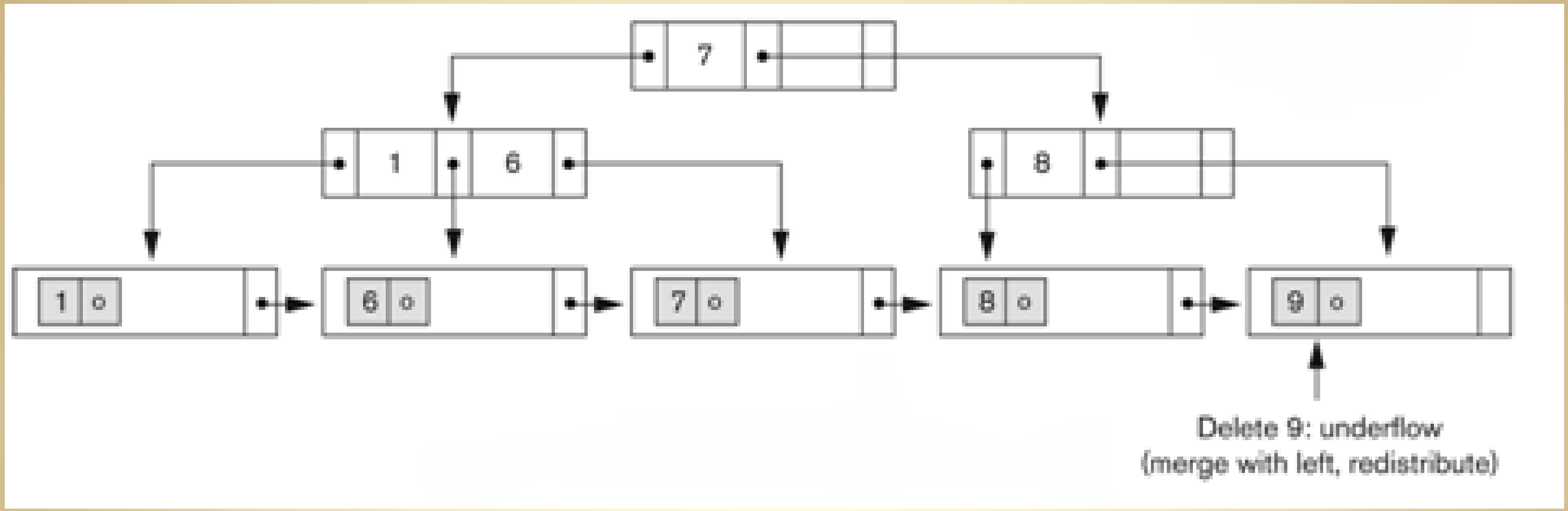


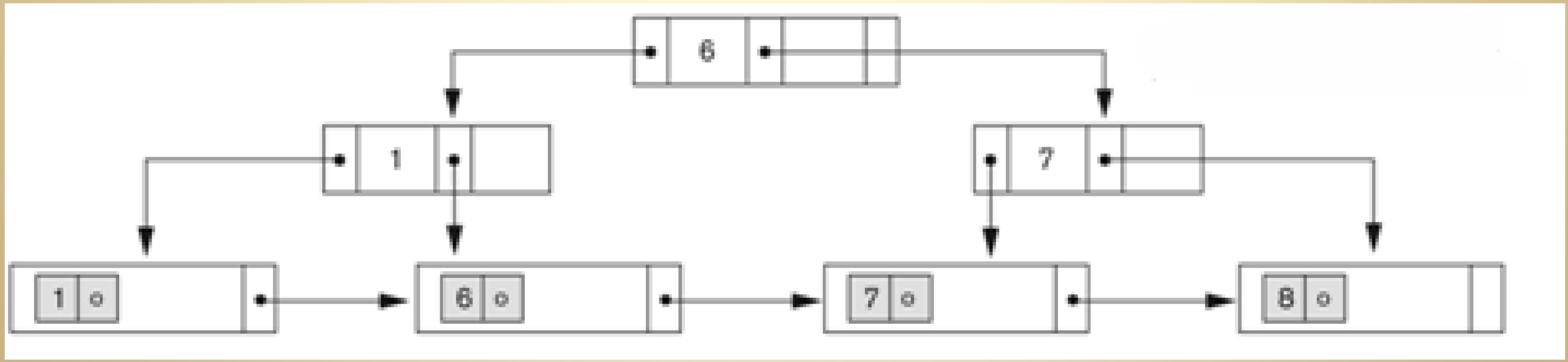
Example: B^+-tree Deletion





Example: B$^+$-tree Deletion





REFERENCES

- Figures on slides 4, 5, 6, 20, 22, 25-28 borrowed from *Database Systems:*
  An Application-Oriented Approach (2nd *ed)*
  by Kifer, Bernstein and Lewis, Addison-Wesley, 2005

- Remaining figures from Elmasri/Navathe