Data Structures

October 19

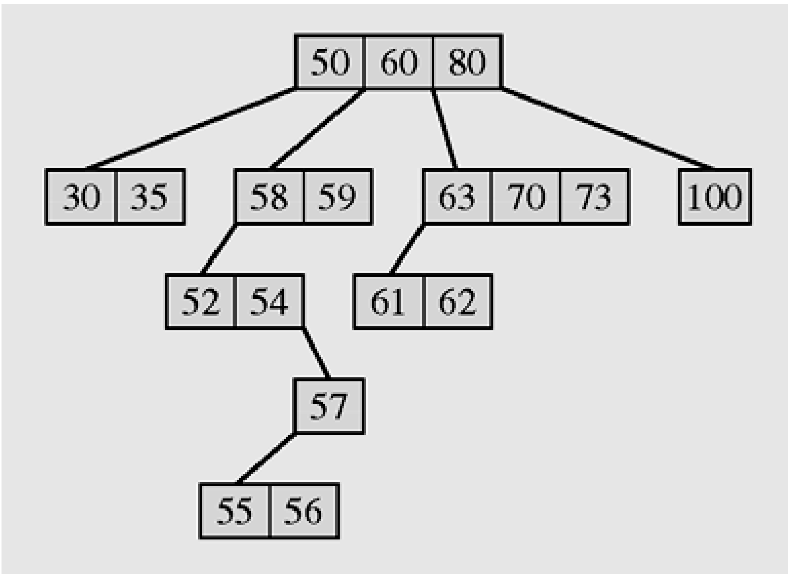
Topics

- Multi-way trees

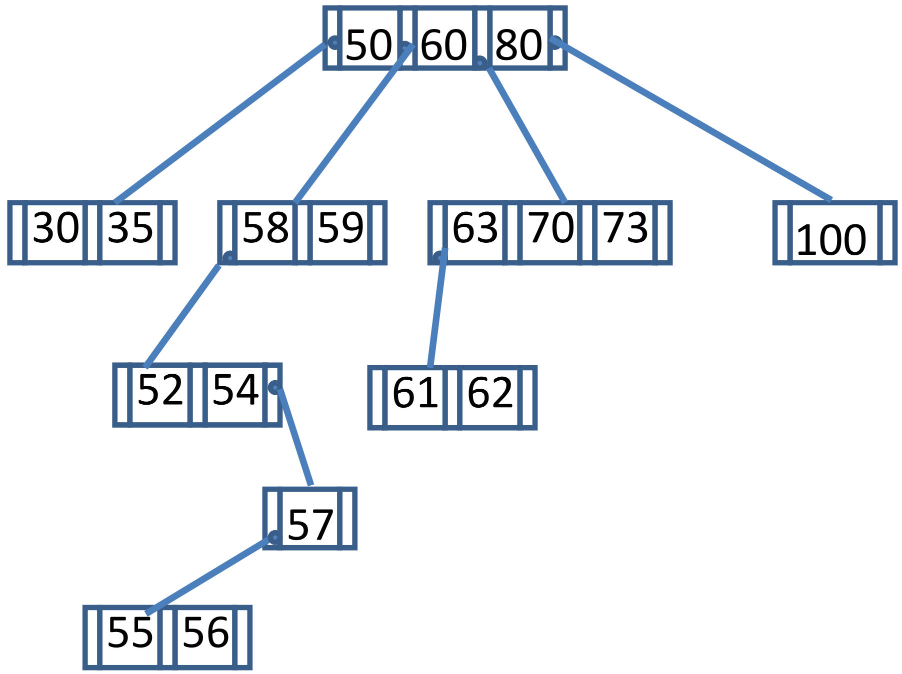
Multiway Trees

- A **multiway search tree of order m**, or an **m-way search tree**, is a multiway tree in which:
  - Each node has (at most) m children and (at most) $m - 1$ keys
  - The keys in each node are in ascending order
  - The keys in the first i children are smaller than the ith key
  - The keys in the last $m - i$ children are larger than the ith key

Multiway Search Trees (continued)

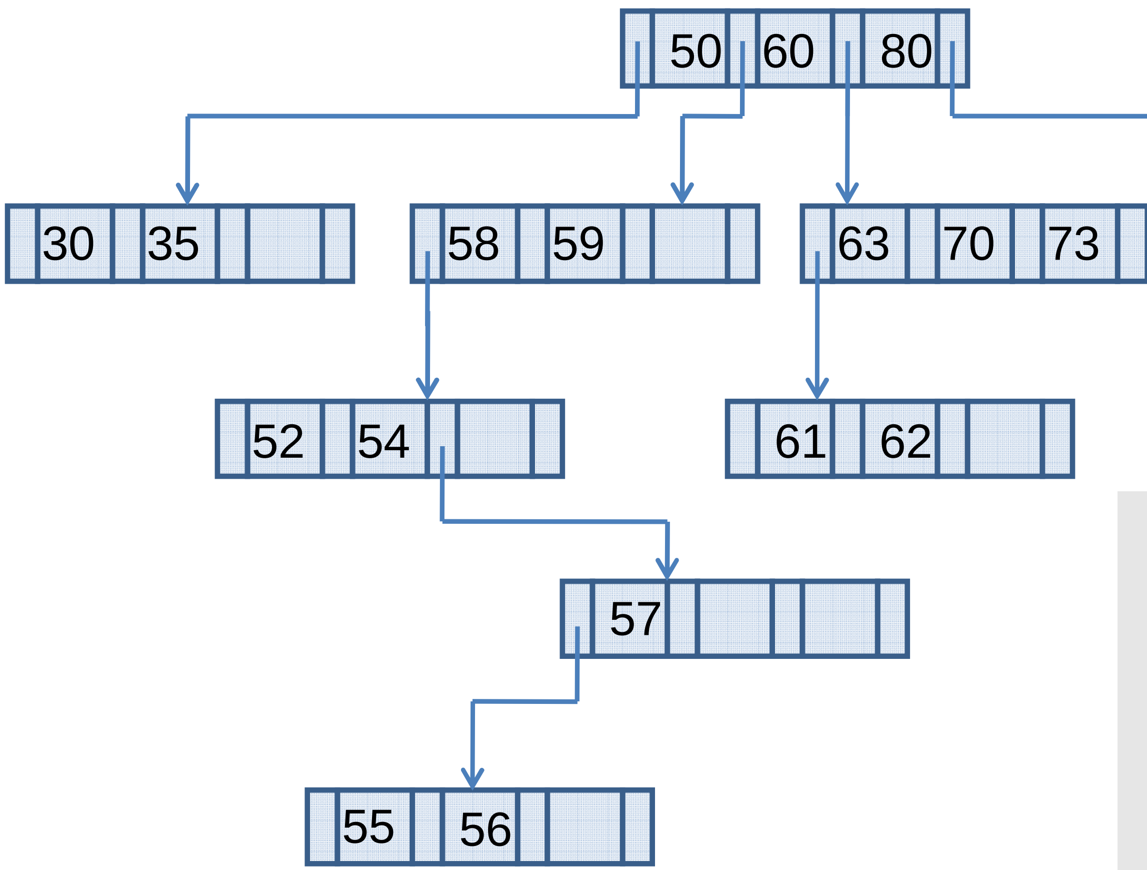


A 4-way tree (book notation)



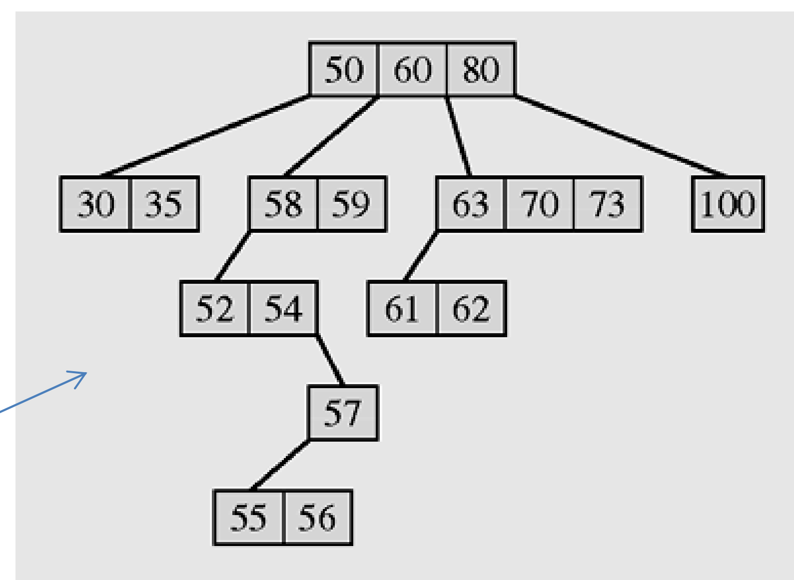
The same 4-way tree in a different notation

Multiway Search Trees (continued)



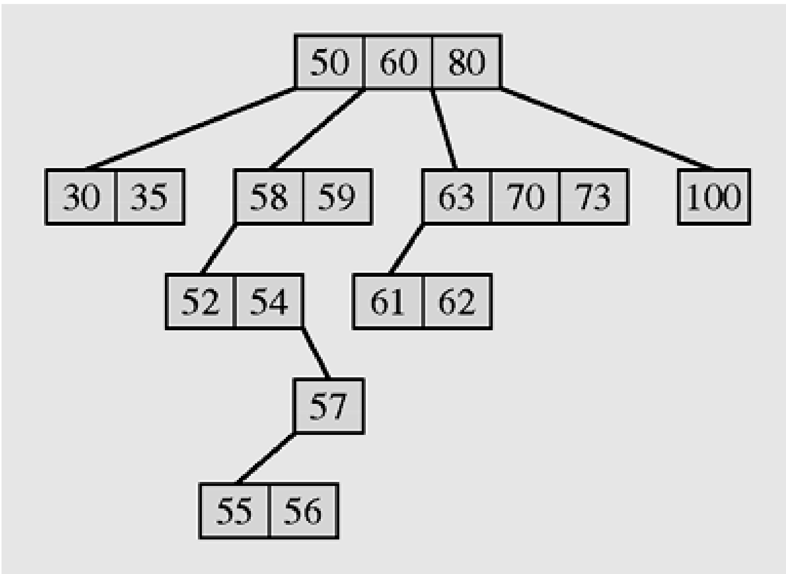
The same 4-way tree in a third notation

Is **not** a 4-way B-tree:
for instance the node
with one key (100) should have two children; will
introduce B-trees later

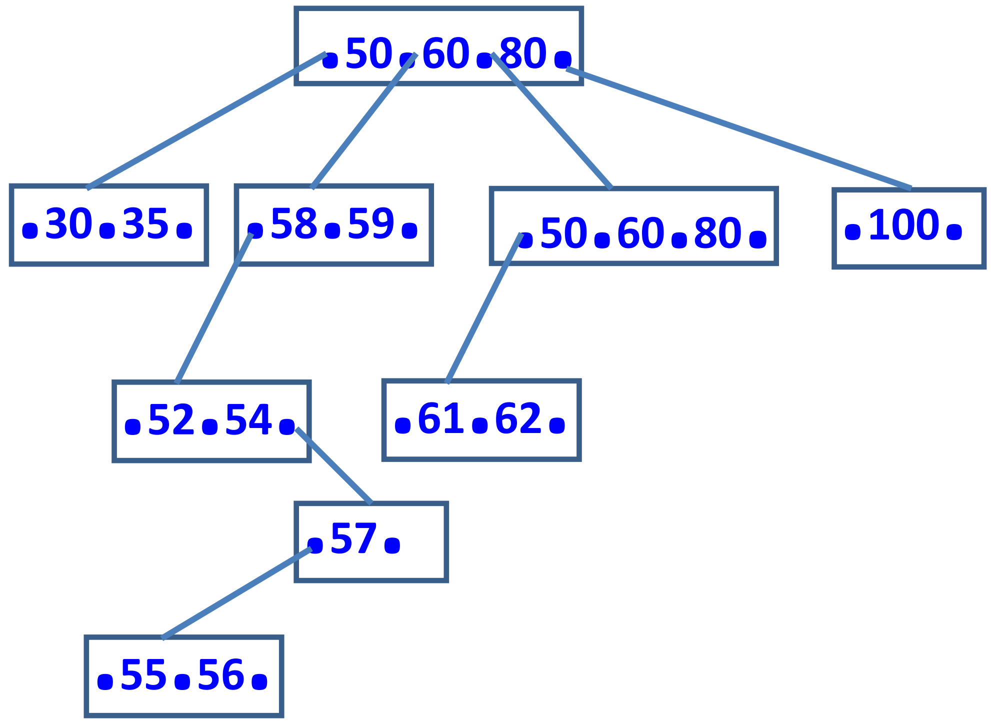


A 4-way tree (book notation)

Multiway Search Trees (continued)



A 4-way tree (book notation)



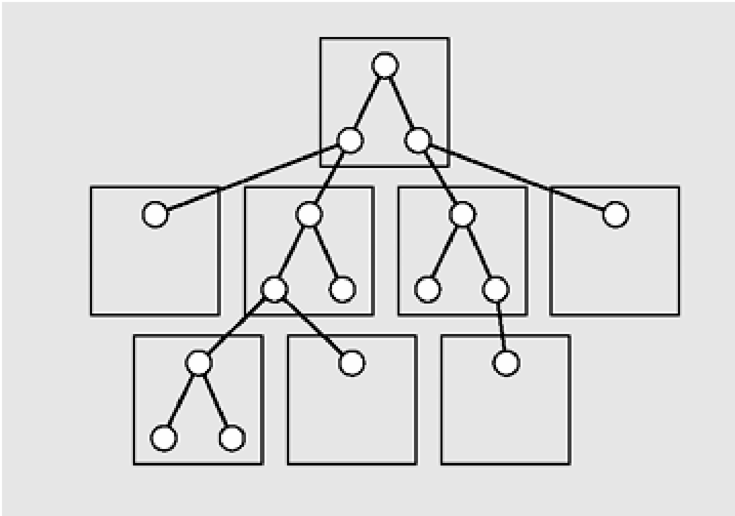
The same 4-way tree in a fourth notation

The Family of B-Trees

access time = seek time + rotational delay (latency) + transfer time

- **Seek time** depends on the mechanical movement of the disk head to position the head at the correct track of the disk
- **Latency** is the time required to position the head above the correct block and is equal to the time needed to make one-half of a revolution
- No matter how you go about it the upshot is:
- Roughly 100 disk accesses per second correspond to 500,000,000 instructions per second (*order of magnitude difference 10^6*)

The Family of B-Trees (continued)



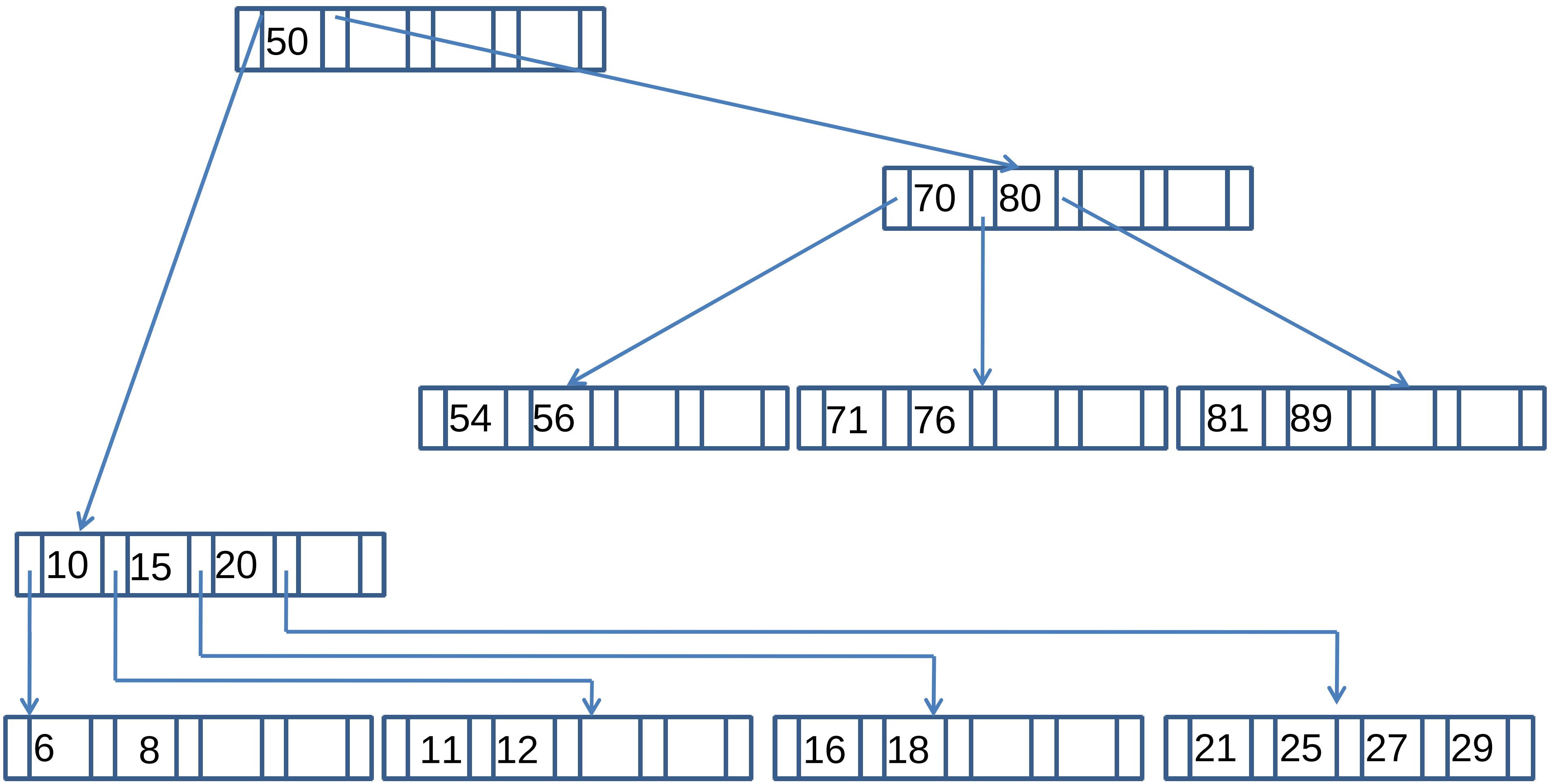
Nodes of a binary tree can be located in different blocks on a disk

Definition: B-Tree of Order *m*

- A multiway search tree of order m
- The root has at least two children, unless it is a leaf
- Each nonroot and each nonleaf nodes holds **k-1** keys and **k** pointers to subtrees, where ***ceiling*(m/2) ≤ k ≤ m**
- All leaves are on the same level
- (NB *ceiling*(m/2) = (m+1) *div* 2); usual notation for ceiling: $\lceil m/2 \rceil$

B-Tree of Order *m*, slightly more precise definition

- A multiway tree of order **m** (m ≥ 3)
- The root has at least two children, unless it is a leaf
- Each nonroot and each nonleaf nodes holds **k-1** keys and **k** pointers to subtrees (*more abstractly: has* ***k*** *children*), where **ceiling(m/2) ≤ k ≤ m** (we emphasize: *each nonleaf node with k-1 keys has precisely k children*)
- All leaves are on the same level

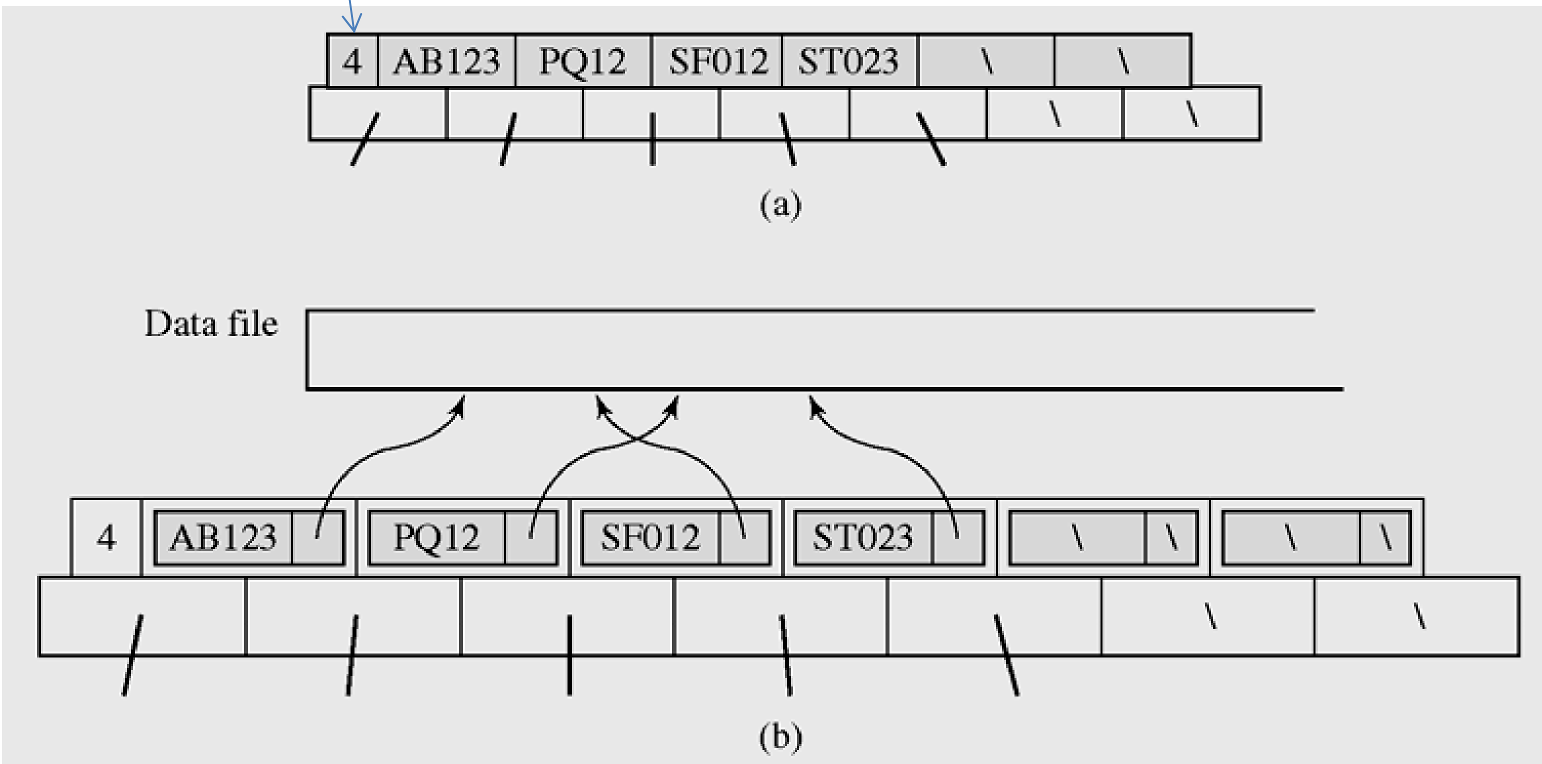


Example of B-Tree of order 5

NB the node with keys 10,15,20 and the node with keys 70, 80 are on the same level;
Of course the same is true for the children of these two nodes. (Key Tallies are omitted)

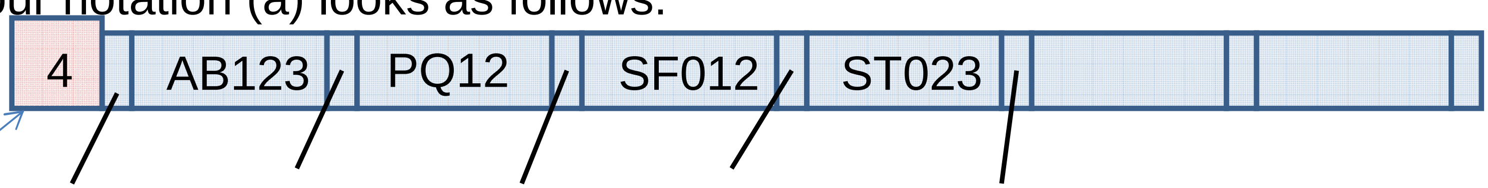
B-Trees

Key tally



One node of a B-tree of order 7 (a) without and (b) with an additional indirection

In our notation (a) looks as follows:



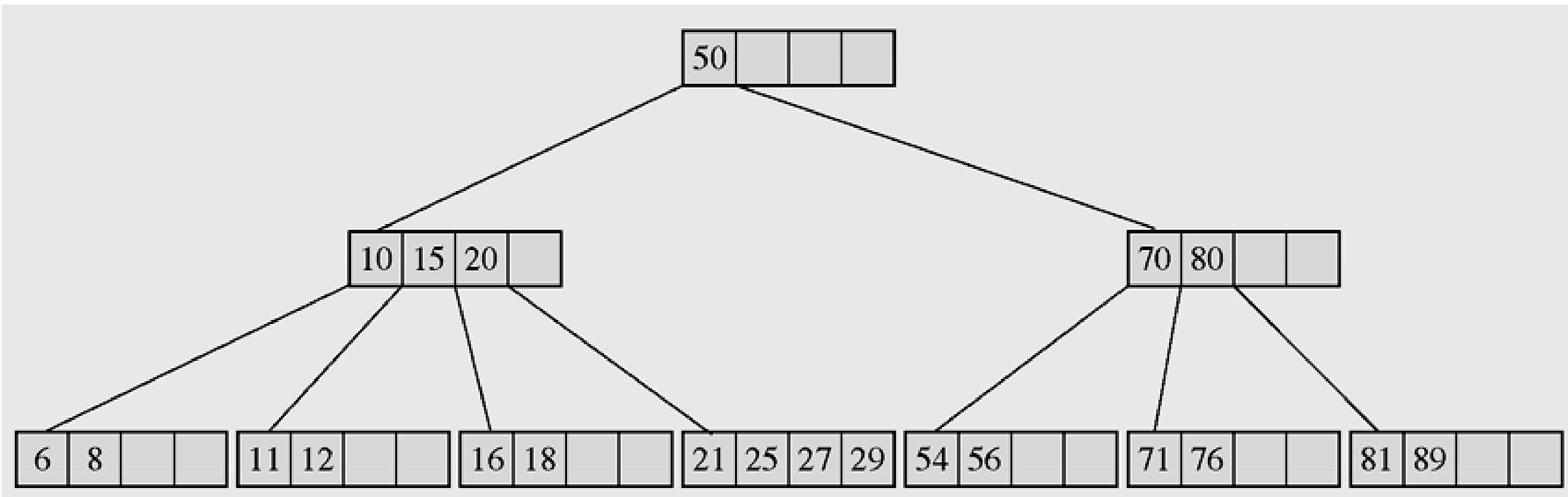
Key tally

Discussion of additional indirection

Height h of B-trees of order m:

- **$h \le \log_q ((n+1)/2) + 1$, n is the number of keys, q = *ceiling*(m/2)**
- Discussion and derivation of this formula

B-Trees (continued)

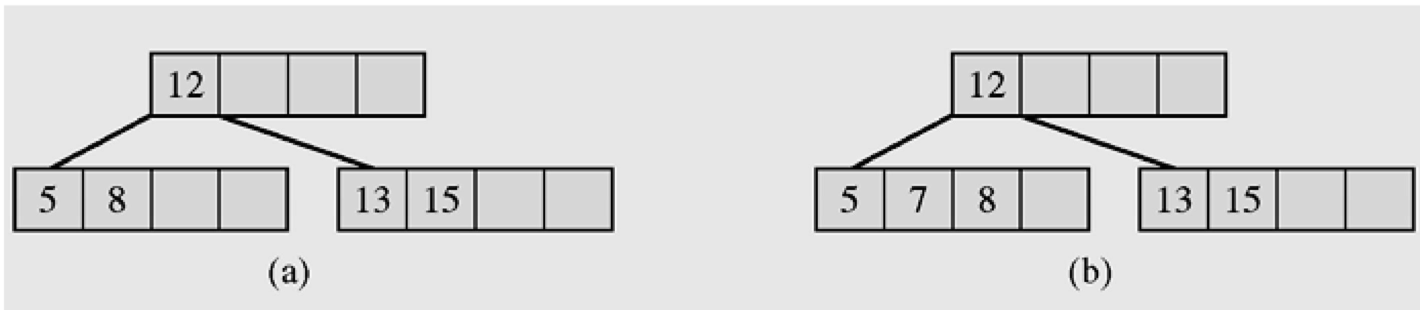


A B-tree of order 5 shown in an abbreviated form

Inserting a Key into a B-Tree

- There are three common situations encountered when inserting a key into a B-tree:
  - A key is placed in a leaf that still has some room
  - The leaf in which a key should be placed is full
  - If the root of the B-tree is full then a new root and a new sibling of the existing root have to be created

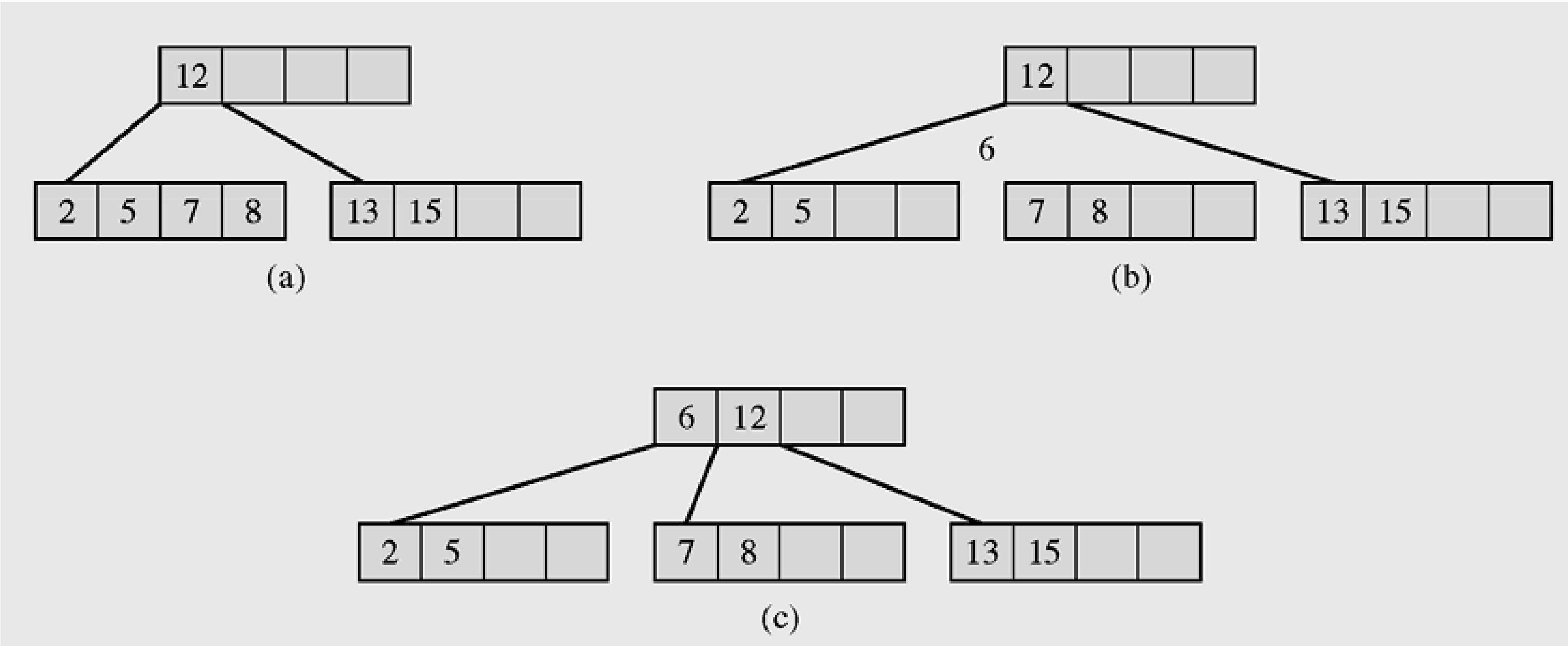
Inserting a Key into a B-Tree of order 5 (continued)



A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells

ceiling(5/2)-1= 2 ≤ (#of keys) ≤ 4 = 5-1
ceiling(5/2) = 3 ≤ (#of children) ≤ 5

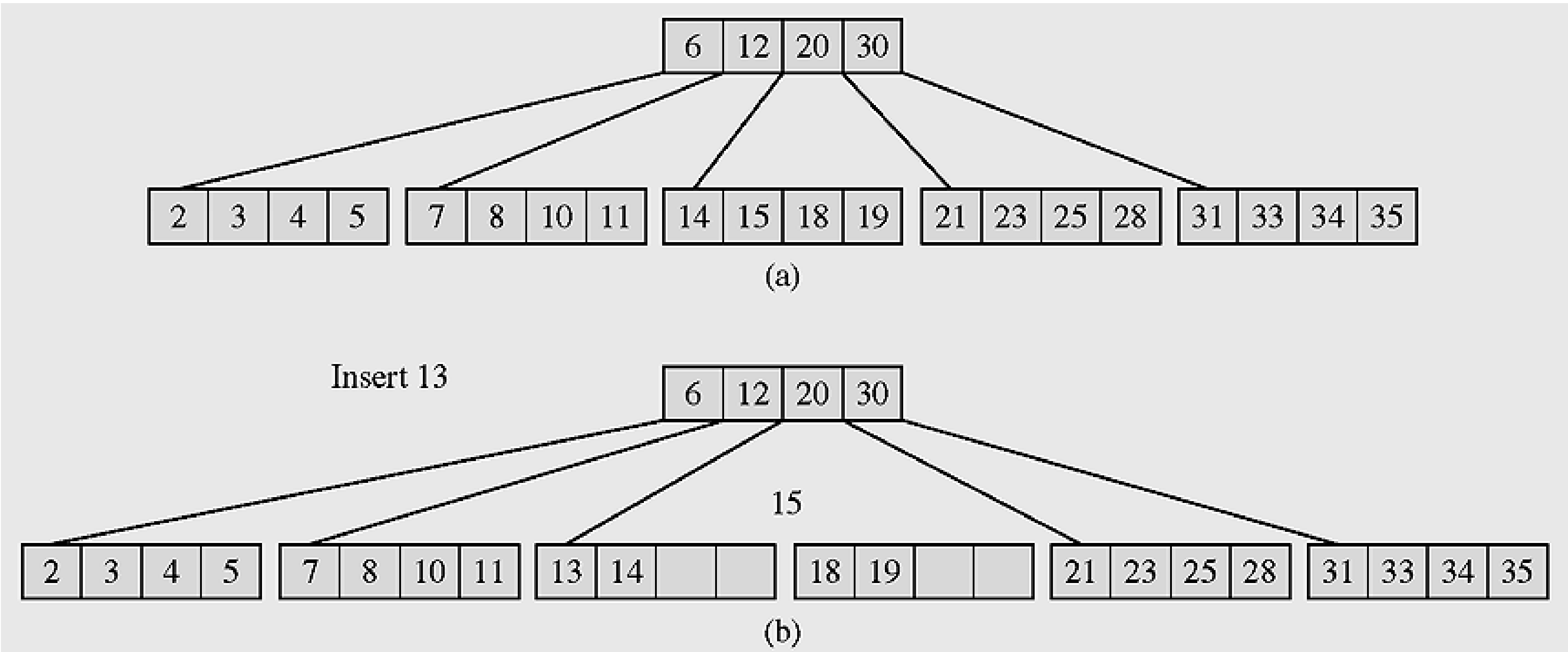
Inserting a Key into a B-Tree of order 5 (continued)



Inserting the number 6 into a full leaf

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

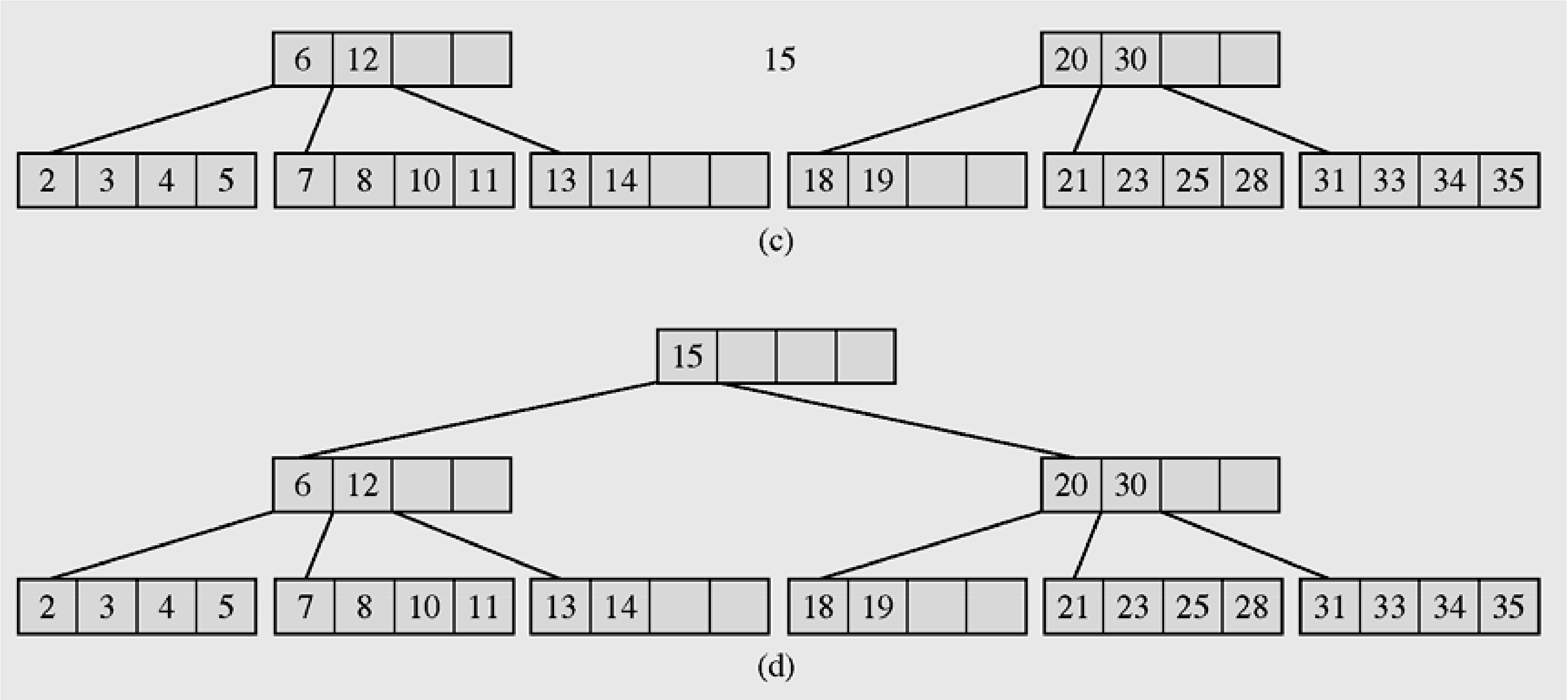
Inserting a Key into a B-Tree of order 5 (continued)



Inserting the number 13 into a full leaf

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

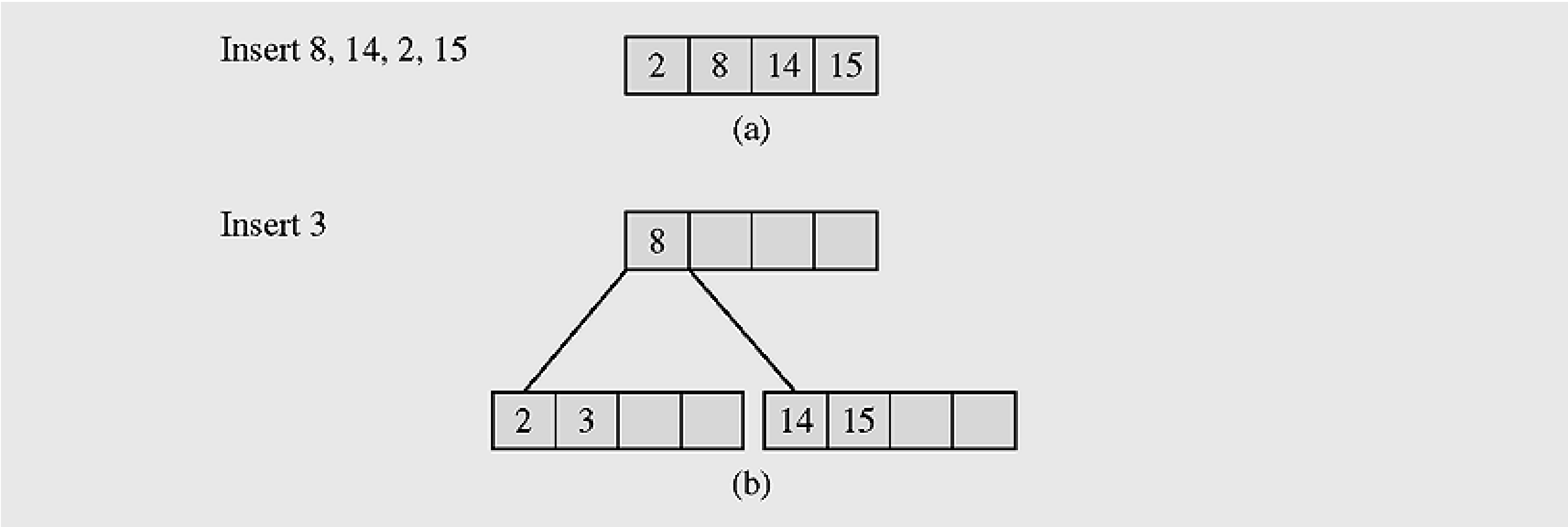
Inserting a Key into a B-Tree of order 5 (continued)



Inserting the number 13 into a full leaf (continued)

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

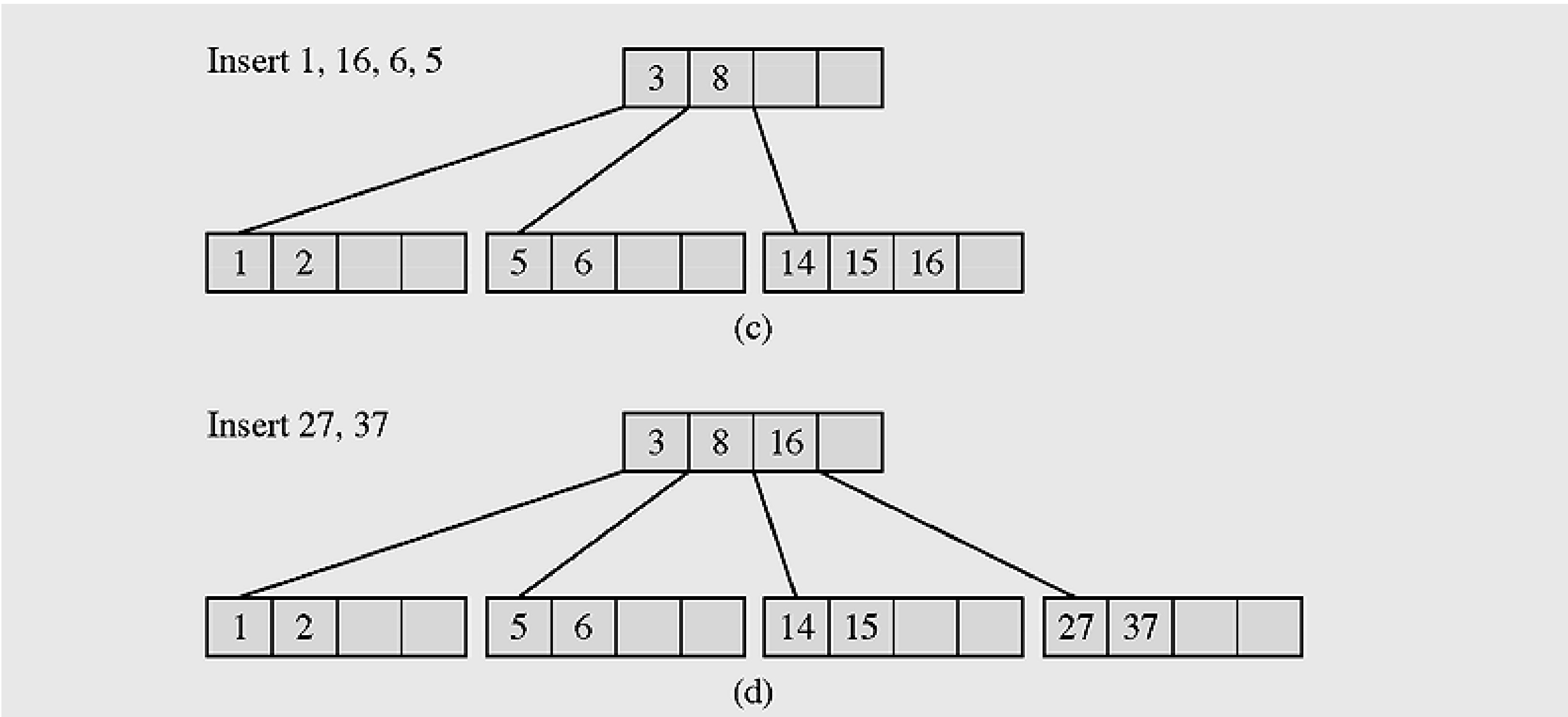
Inserting a Key into a B-Tree of order 5 (continued)



Building a B-tree of order 5 with the `BTreeInsert()` algorithm

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

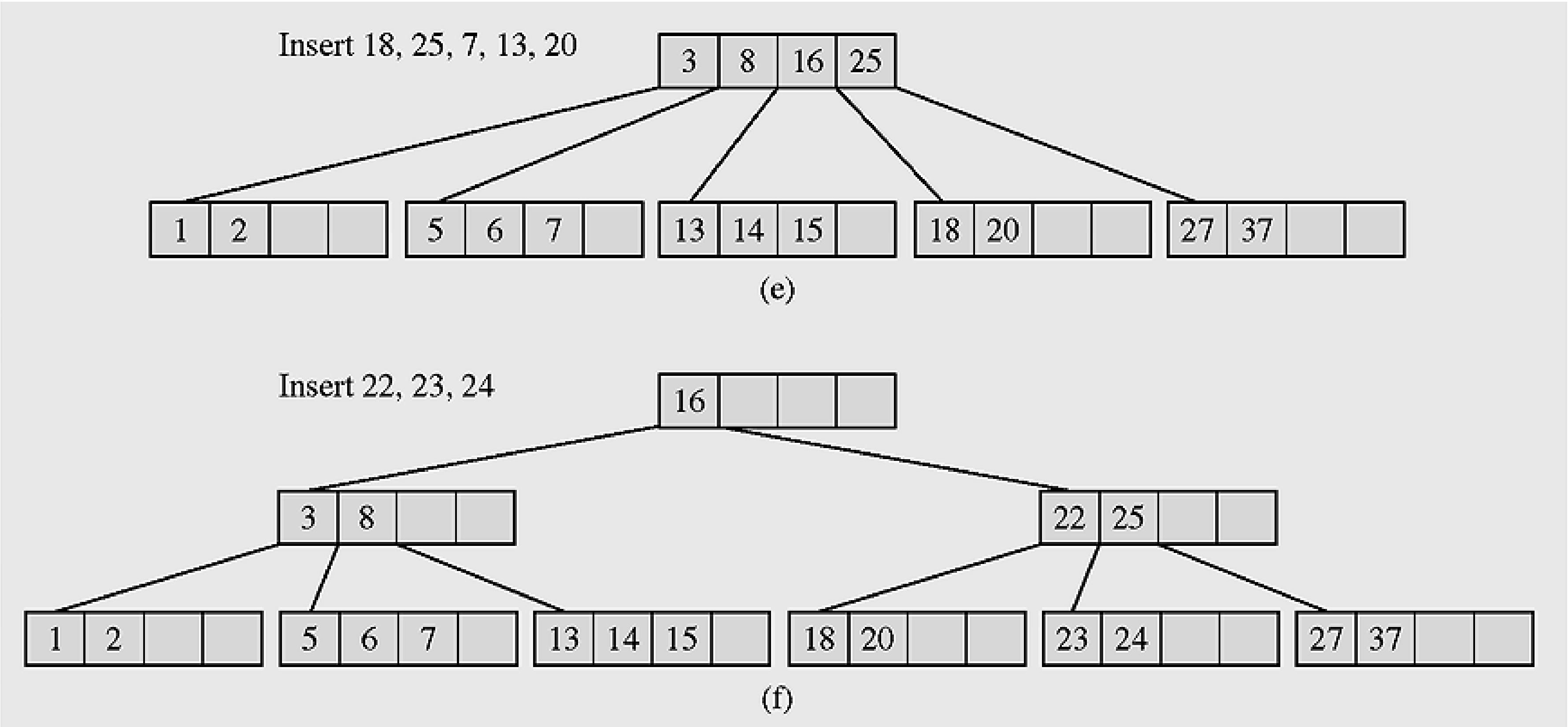
Inserting a Key into a B-Tree of order 5 (continued)



Building a B-tree of order 5 with the `BTreeInsert()` algorithm (continued)

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

Inserting a Key into a B-Tree of order 5 (continued)



Building a B-tree of order 5 with the `BTreeInsert()` algorithm (continued)

2 ≤ (#of keys) ≤ 4
3 ≤ (#of children) ≤ 5

Inserting a Key into a B-Tree

- See page 306 in Drozdek for the algorithm in words
- See page 307 for its implementation in C++

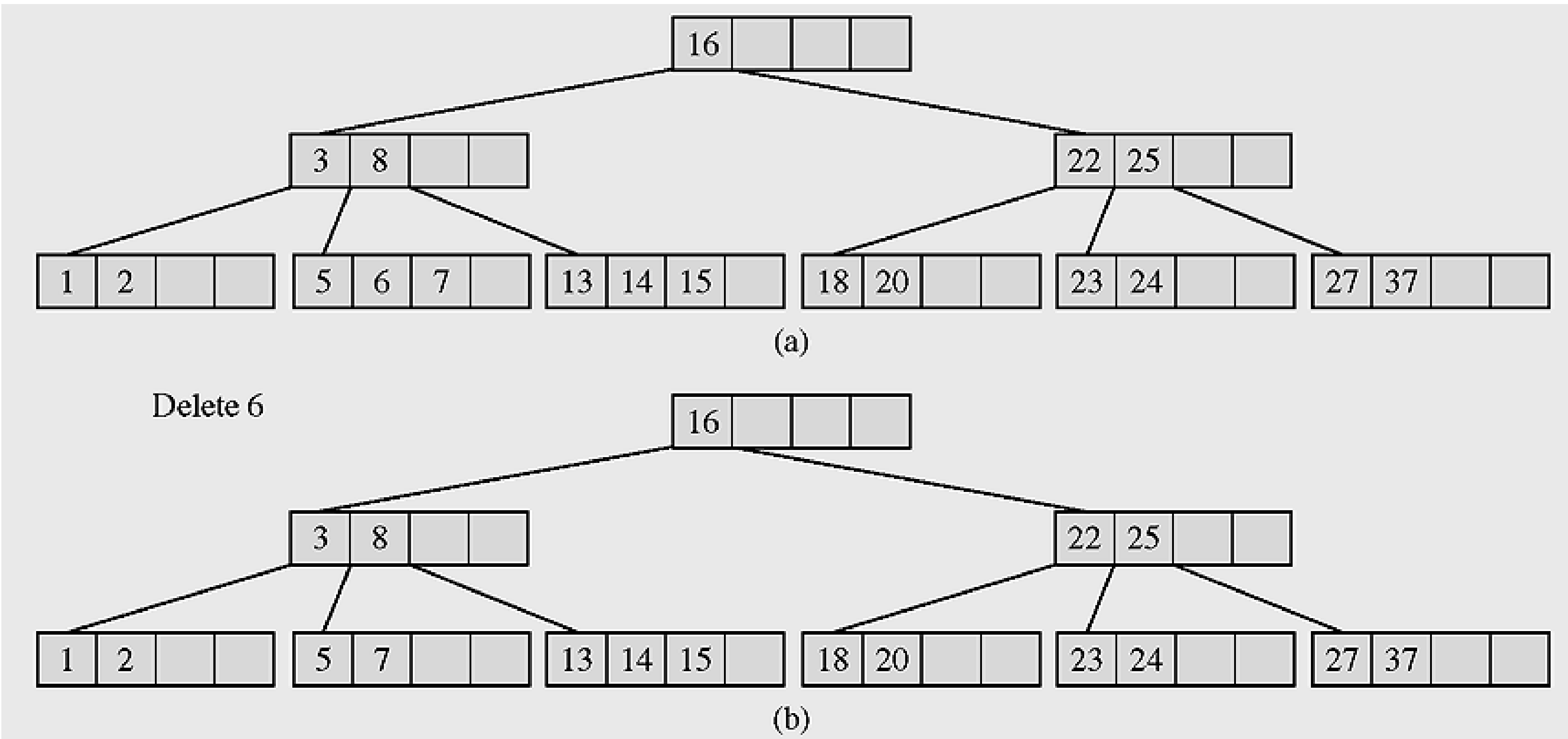
Inserting a Key into a B-Tree of order m

- Find a leaf where the key should be placed according to the search tree property
- In case this leaf has fewer than m-1 the key is placed in order in this leaf
- If the leaf overflowed (i.e., it contains m-1 keys) it is split in two. The median key is moved to the father; there one continues in the same fashion.

Deleting a Key from a B-Tree

- Avoid allowing any node to be less than half full after a deletion
- In deletion, there are two main cases:
  - Deleting a key from a leaf
  - Deleting a key from a nonleaf node

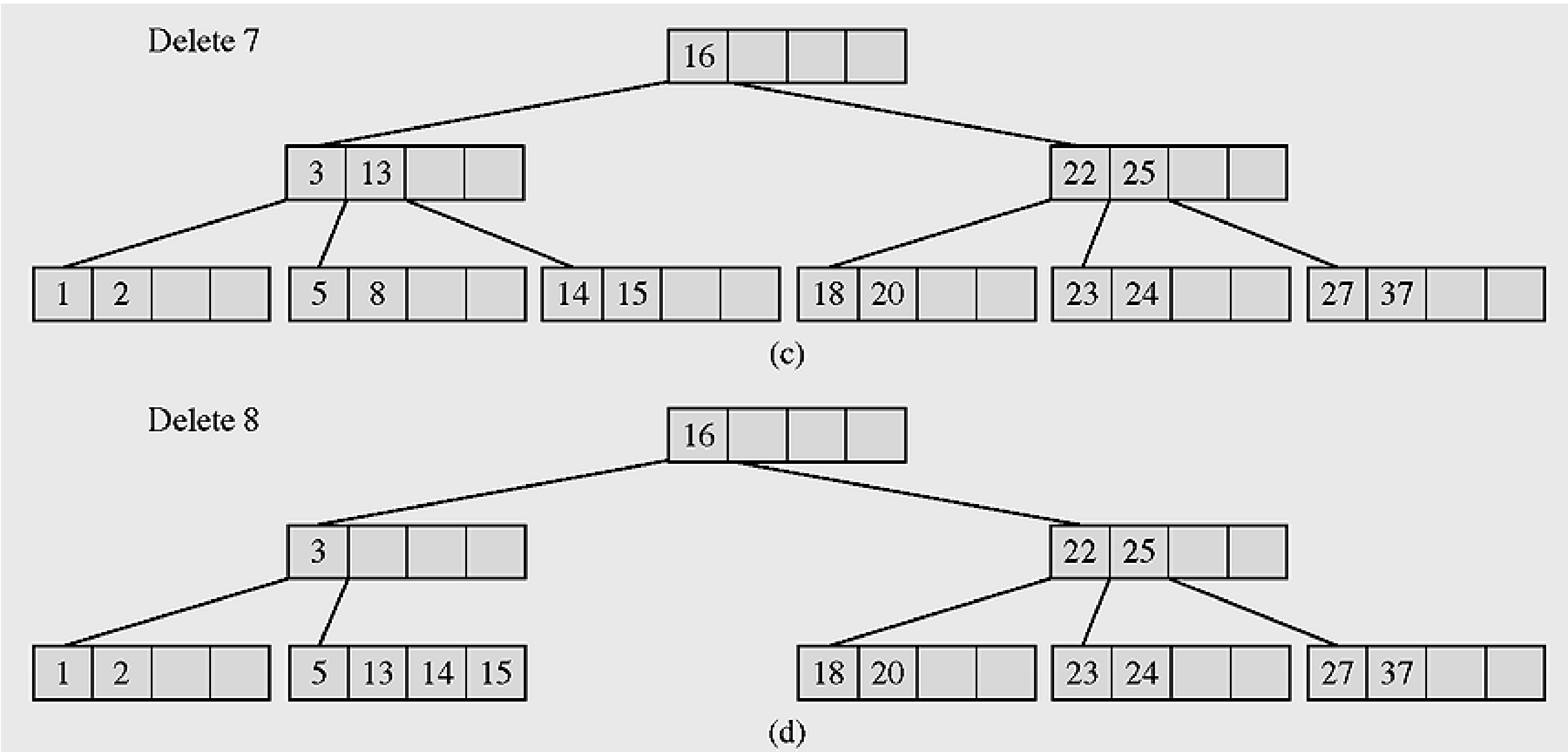
Deleting a Key from a B-Tree of order 5 (continued)



Deleting keys from a B-tree

$2 \leq$ (#of keys) ≤ 4
$3 \leq$ (#of children) ≤ 5

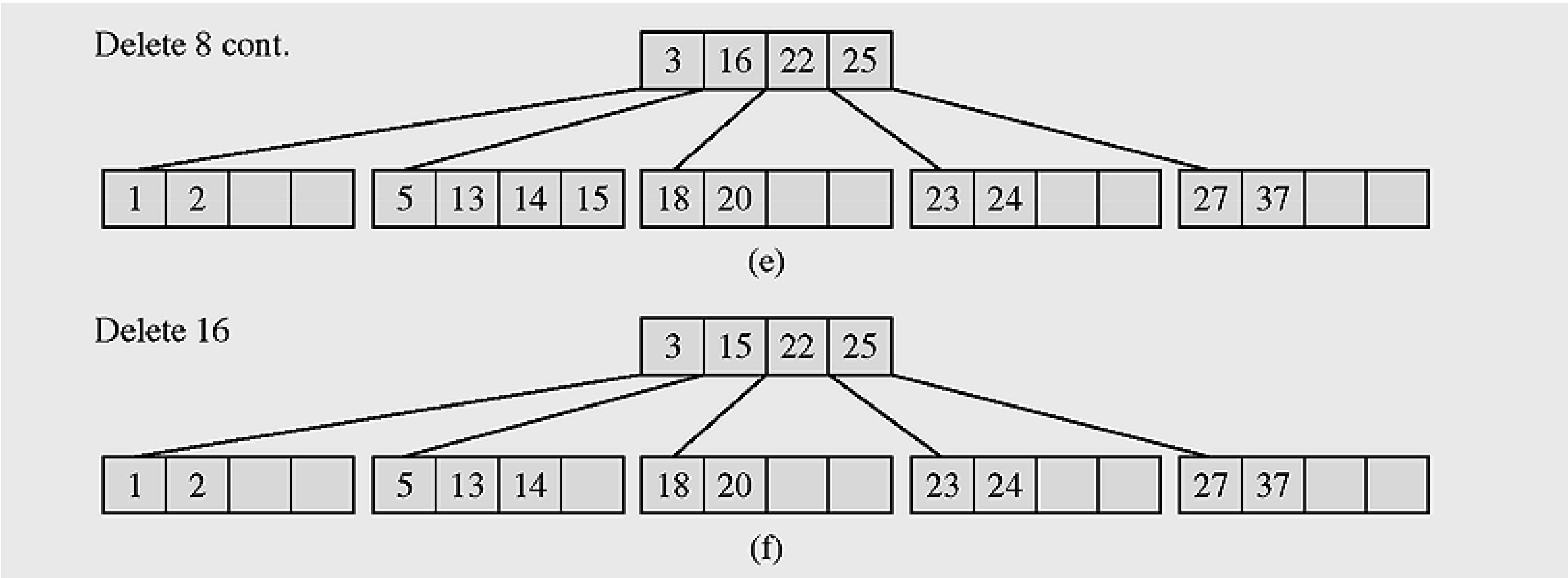
Deleting a Key from a B-Tree of order 5 (continued)



Deleting keys from a B-tree (continued)

$2 \le$ (#of keys) ≤ 4
$3 \le$ (#of children) ≤ 5

Deleting a Key from a B-Tree of order 5 (continued)



Deleting keys from a B-tree (continued)

2 ≤ (#of keys) ≤ 4

3 ≤ (#of children) ≤ 5

Deleting a key from a B-tree of order m

- If the key is not a leaf swap it with the biggest smaller key present in the B-tree – this predecessor will be in a leaf.
- Remove the key from the leaf.
- If the node from which the key is removed is not underflowing (i.e., contains at least *ceiling*(m/2)-1 keys), then we are done.
- If the node from which the key is deleted underflows, then try each of the following:
  a. Try to borrow a key of the immediate left brother (in case the node has a left brother)
  b. If no success in a): Try to borrow a key of the immediate right brother (if the node has a right brother)
  c. If no success in b): merge the node with its immediate left brother (if the node has a left brother
  d. If no success in c): merge the node with the immediate right brother (if you come this far, there will always be a right brother)
  e. Merging two brothers will cause the father to have one less key; now we apply a-d to the father node