MULTIWAY TREES

Dr. Megha Ummat

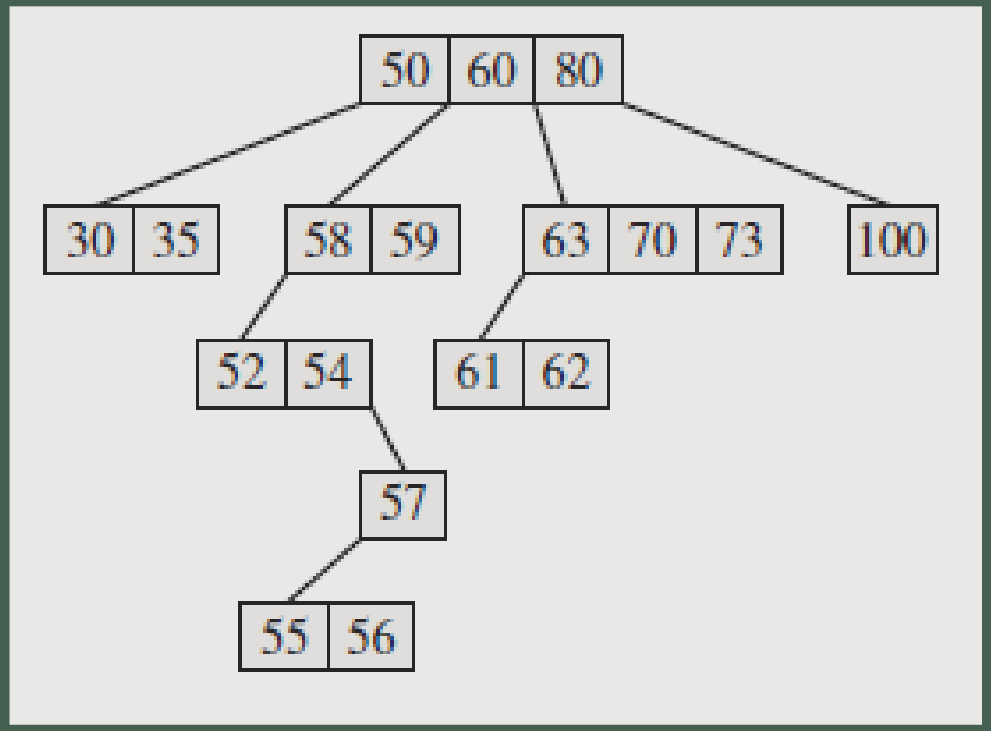
Multiway Tree

- A tree was defined as either an empty structure or a structure whose children are disjoint trees $t_1, \ldots, t_m$.
- According to this definition, each node of this kind of tree can have more than two children. This tree is called a *multiway tree of order m*, or an *m-way tree*.

Multiway Tree

- A *multiway search tree of order m*, or an *m-way search tree*, is a multiway tree in which
  - Each node has *m* children and *m* – 1 keys.
  - The keys in each node are in ascending order.
  - The keys in the first *i* children are smaller than the i^{th} key.
  - The keys in the last *m* – *i* children are larger than the i^{th} key.
- M-way search trees are used for fast information retrieval and update.

Multiway Tree



Example of 4-way Tree

- Accessing the keys can require a different number of tests for different keys: the number 35 can be found in the second node tested, and 55 is in the fifth node checked.
- The tree, therefore, suffers from a known malaise: it is unbalanced.
- This problem is of particular importance if we want to use trees to process data on secondary storage such as disks or tapes where each access is costly.

B-Trees

- In database programs where most information is stored on disks or tapes, the time penalty for accessing secondary storage can be significantly reduced by the proper choice of data structures.
- *B-trees* (Bayer and McCreight 1972) are one such approach.
- The number of keys in one node can vary depending on the sizes of the keys, organization of the data, and the size of a block.
- The amount of information stored in one node of the B-tree can be rather large.

B-Tree

- A *B-tree of order m* is a multiway search tree with the following properties:

  1. The root has at least two subtrees unless it is a leaf.

  2. Each non-root and each non-leaf node holds $k - 1$ keys and k pointers to subtrees where $\lceil m/2 \rceil \leq k \leq m$.

  3. Each leaf node holds $k - 1$ keys where $\lceil m/2 \rceil \leq k \leq m$.

  4. All leaves are on the same level.

- According to these conditions, a B-tree is always at least half full, has few levels, and is perfectly balanced.

B-Tree

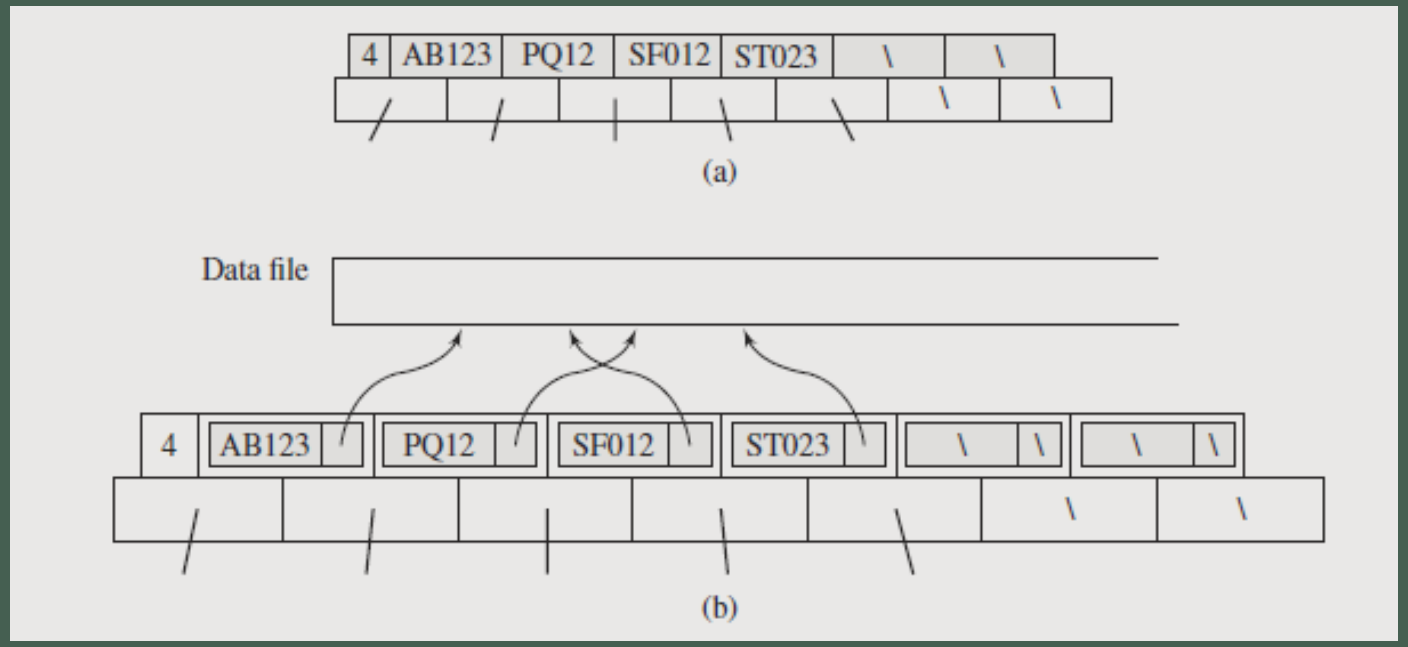
- A node of a B-tree is usually implemented as a class containing an array of *m* – 1 cells for keys, an *m*-cell array of pointers to other nodes, and possibly other information facilitating tree maintenance, such as the number of keys in a node and a leaf/nonleaf flag.

```
template <class T, int M>
class BTreeNode {
public:
    BTreeNode();
    BTreeNode(const T&);
private:
    bool leaf;
    int keyTally;
    T keys[M-1];
    BTreeNode *pointers[M];
    friend BTree<T,M>;
};
```

B-Tree

- Figure (a) contains an example of a B-tree of order 7 that stores codes for some items.
- In this B-tree, the keys appear to be the only objects of interest. In most cases, however, such codes would only be fields of larger structures.
- In these cases, the array keys is an array of objects, each having a unique identifier field (such as the identifying code in Figure (a) and an address of the entire record on secondary storage, as in Figure (b).

B-Tree

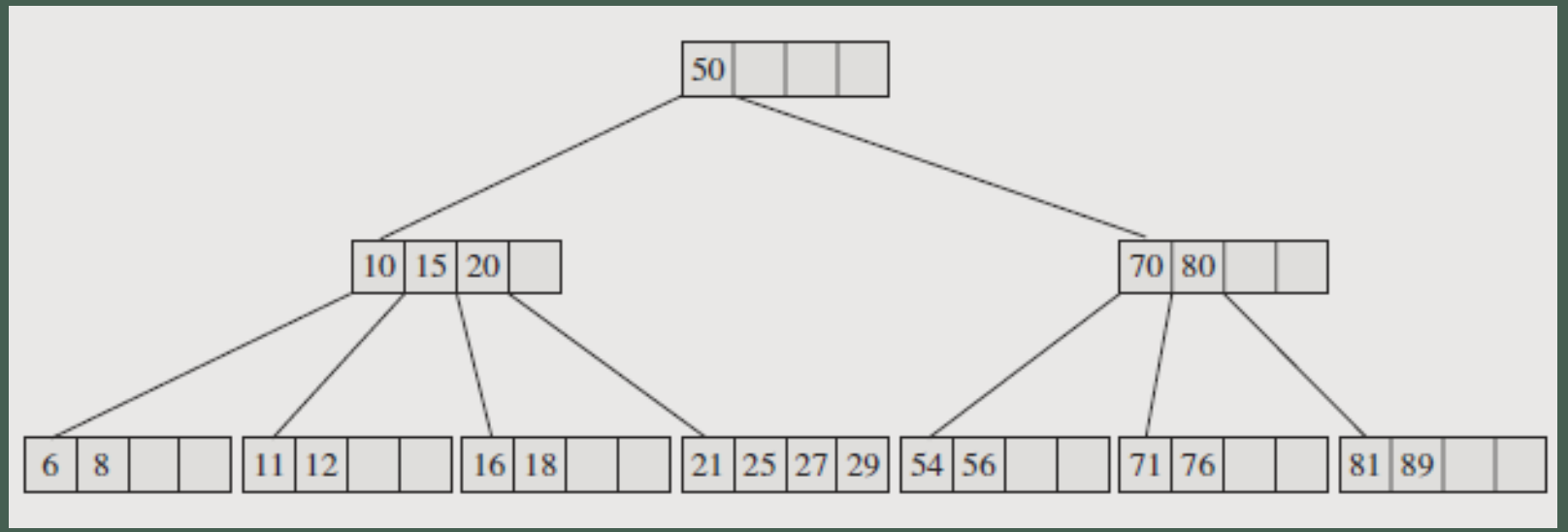

4
AB123
PQ12
SF012
ST023
\
\
(a)
Data file
4
AB123
PQ12
SF012
ST023
\
\
\
\
\
\
(b)

B-Tree

- If the contents of one such node also reside in secondary storage, each key access would require two secondary storage accesses.
- In the long run, this is better than keeping the entire records in the nodes, because in this case, the nodes can hold a very small number of such records.
- The resulting B-tree is much deeper, and search paths through it are much longer than in a B-tree with the addresses of records.

B-Tree

- From now on, B-trees will be shown in an abbreviated form without explicitly indicating keyTally or the pointer fields.



Searching a B-Tree

```
BTreeNode *BTreeSearch(keyType K, BTreeNode *node){
    if (node != 0) {
        for (i=1; i <= node->keyTally && node->keys[i-1] < K; i++);
        if (i > node->keyTally || node->keys[i-1] > K)
            return BTreeSearch(K,node->pointers[i-1]);
        else return node;
    }
    else return 0;
}
```

Searching a B-Tree

- The worst case of searching is when a B-tree has the smallest allowable number of pointers per nonroot node, $q = \lceil m/2 \rceil$, and the search has to reach a leaf (for either a successful or an unsuccessful search).
- In this case, in a B-tree of height h, there are

$$
\begin{aligned}
&1 \text{ key in the root} + \\
&2(q-1) \text{ keys on the second level} + \\
&2q(q-1) \text{ keys on the third level} + \\
&2q^2(q-1) \text{ keys on the fourth level} + \\
&\vdots \\
&2q^{h-2}(q-1) \text{ keys in the leaves (level } h) = \\
&1 + \left(\sum_{i=0}^{h-2} 2q^i \right)(q-1) \text{ keys in the B-tree}
\end{aligned}
$$

Searching a B-Tree

With the formula for the sum of the first n elements in a geometric progression,

$$\sum_{i=0}^{n} q^i = \frac{q^{n+1} - 1}{q - 1}$$

the number of keys in the worst-case B-tree can be expressed as

$$1 + 2(q-1)\left(\sum_{i=0}^{h-2} q^i\right) = 1 + 2(q-1)\left(\frac{q^{h-1} - 1}{q - 1}\right) = -1 + 2q^{h-1}$$

The relation between the number n of keys in any B-tree and the height of the B-tree is then expressed as

$$n \geq -1 + 2q^{h-1}$$

Solving this inequality for the height h results in

$$h \leq \log_q \frac{n+1}{2} + 1$$

Searching a B-Tree

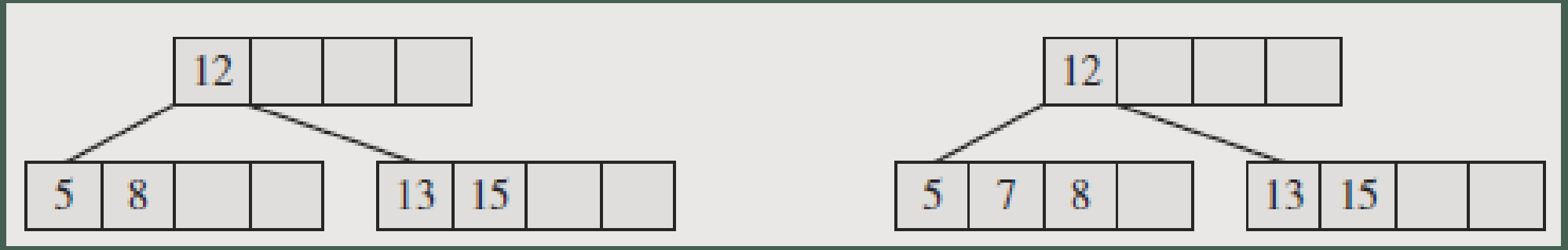
- This means that for a sufficiently large order m, the height is small even for a large number of keys stored in the B-tree.
- For example, if $m = 200$ and $n = 2{,}000{,}000$, then $h \leq 4$; in the worst case, finding a key in this B-tree requires four seeks.

Inserting a key in B-Tree

- Given an incoming key, we go directly to a leaf and place it there, if there is room.
- When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent.
- If the parent is full, the process is repeated until the root is reached and a new root created.

Inserting a key in B-Tree: Case 1

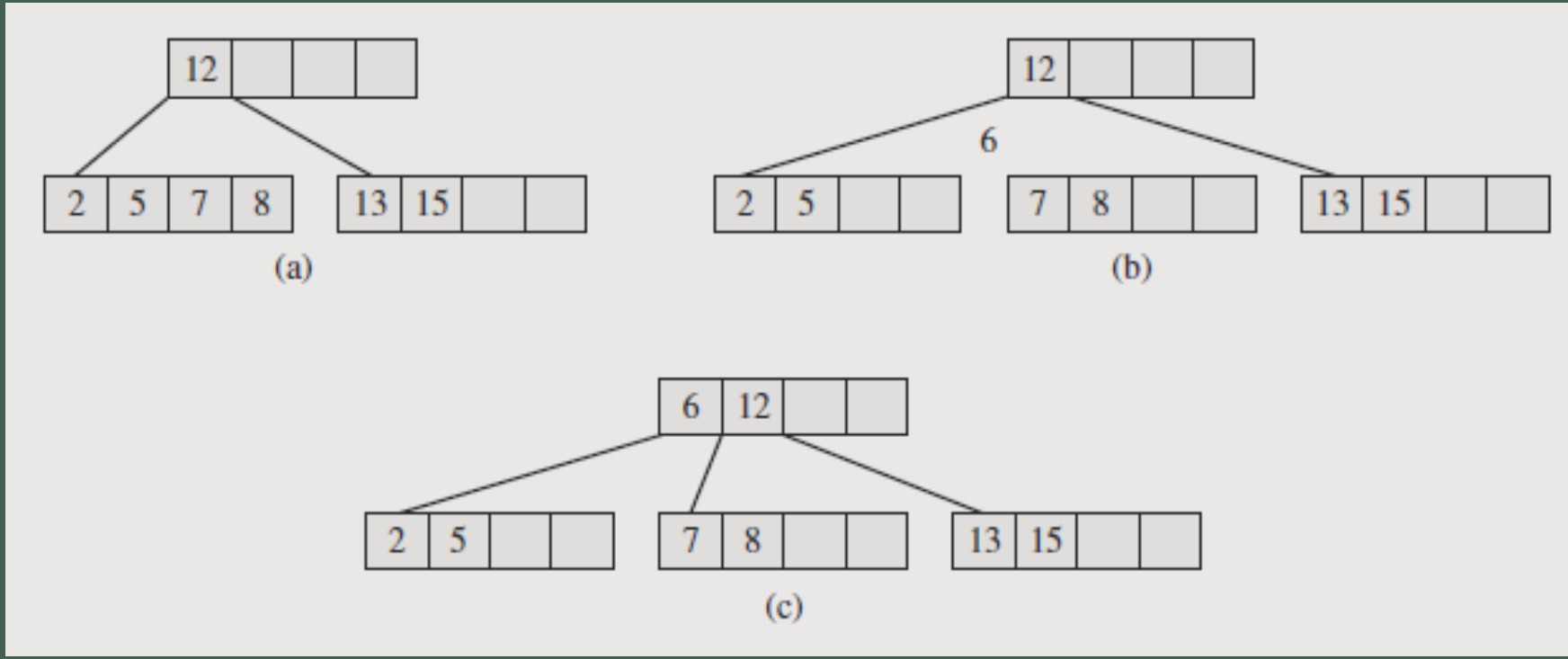
- A key is placed in a leaf that still has some room, as in Figure. In a B-tree of order 5, a new key, 7, is placed in a leaf, preserving the order of the keys in the leaf so that key 8 must be shifted to the right by one position.



Inserting a key in B-Tree: Case 2

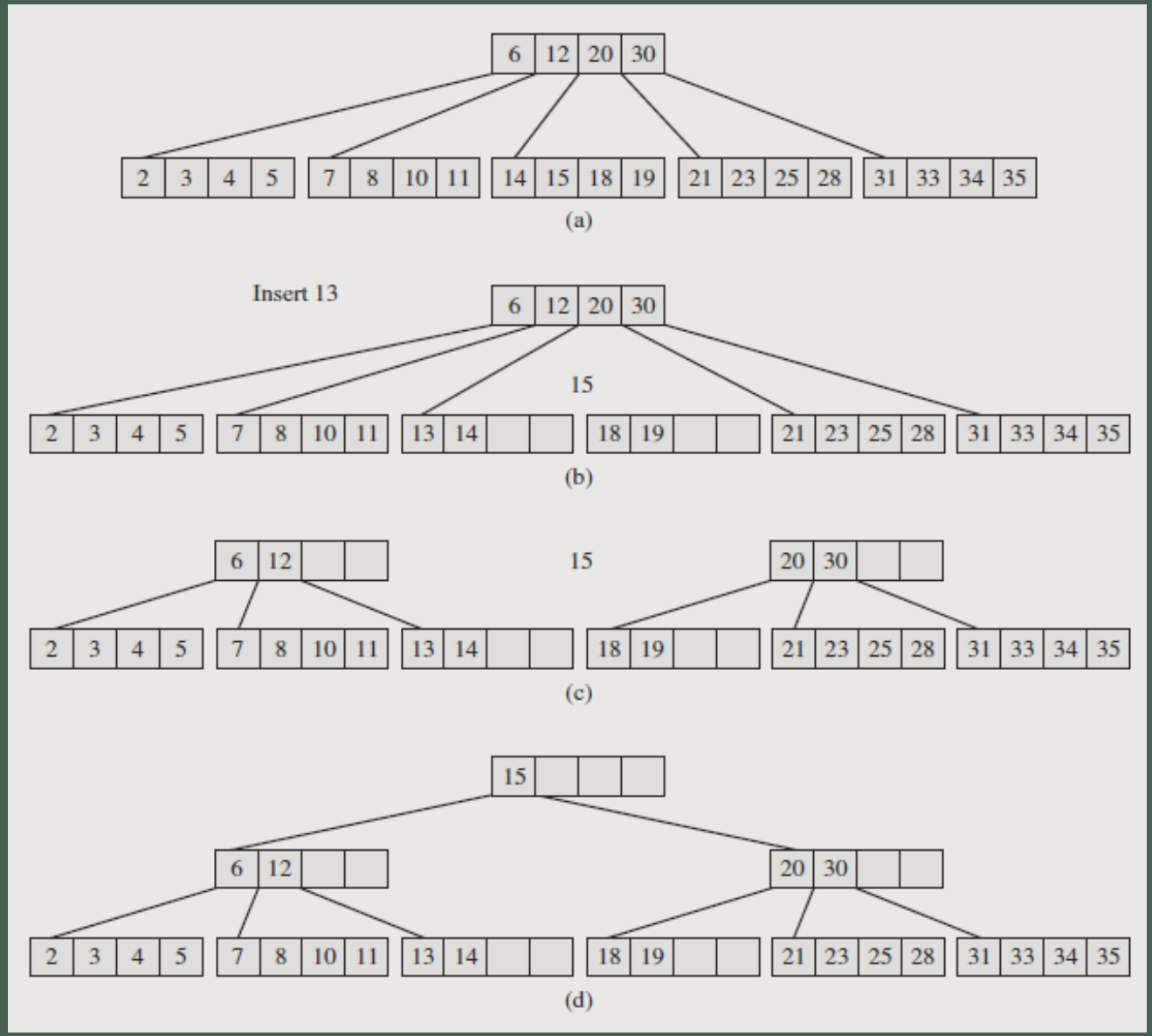
- The leaf in which a key should be placed is full.
- In this case, the leaf is *split*, creating a new leaf, and half of the keys are moved from the full leaf to the new leaf. But the new leaf has to be incorporated into the B-tree.
- The middle key is moved to the parent, and a pointer to the new leaf is placed in the parent as well.
- The same procedure can be repeated for each internal node of the B-tree so that each such split adds one more node to the B-tree.
- Moreover, such a split guarantees that each leaf never has less than $\lceil m/2 \rceil - 1$ keys.

Inserting a key in B-Tree



Inserting a key in B-Tree: Case 3

- A special case arises if the root of the B-tree is full. In this case, a new root and a new sibling of the existing root have to be created.
- This split results in two new nodes in the B-tree.
- For example, after inserting the key 13 in the third leaf in Figure (a), the leaf is split (as in case 2), a new leaf is created, and the key 15 is about to be moved to the parent, but the parent has no room for it (Figure (b)).
- So the parent is split (Figure (c)), but now two B-trees have to be combined into one. This is achieved by creating a new root and moving the middle key to it (Figure (d)).
- It should be obvious that it is the only case in which the B-tree increases in height.


6 12 20 30
2 3 4 5
7 8 10 11
14 15 18 19
21 23 25 28
31 33 34 35
(a)
Insert 13
6 12 20 30
15
2 3 4 5
7 8 10 11
13 14
18 19
21 23 25 28
31 33 34 35
(b)
6 12
15
20 30
2 3 4 5
7 8 10 11
13 14
18 19
21 23 25 28
31 33 34 35
(c)
15
6 12
20 30
2 3 4 5
7 8 10 11
13 14
18 19
21 23 25 28
31 33 34 35
(d)

Inserting a key in B-Tree

```
BTreeInsert(K)
  find a leaf node to insert K;
   while (true)
      find a proper position in array keys for K;
      if node is not full
         insert K and increment keyTally;
         return;
      else split node into node1 and node2;// node1 = node, node2 is new;
         distribute keys and pointers evenly between node1 and node2 and
         initialize properly their keyTally's;
         K = middle key;
         if node was the root
            create a new root as parent of node1 and node2;
            put K and pointers to node1 and node2 in the root, and set its keyTally to 1;
            return;
         else node = its parent;// and now process the node's parent;
```

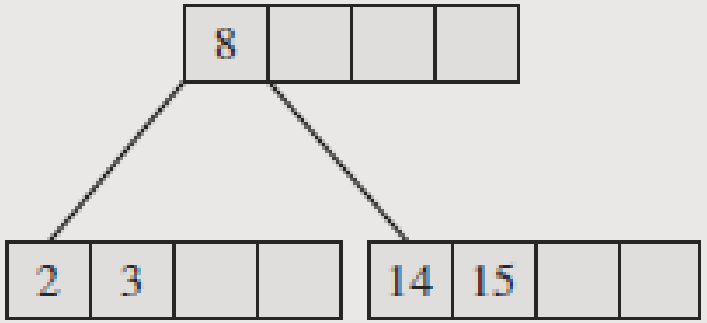
Exercise

- Insert 8, 14, 2, 15, 3, 1, 16, 6, 5, 27, 37, 18, 25, 7, 13, 20, 22, 23, 24 into a B-tree of order 5.

Insert 8, 14, 2, 15

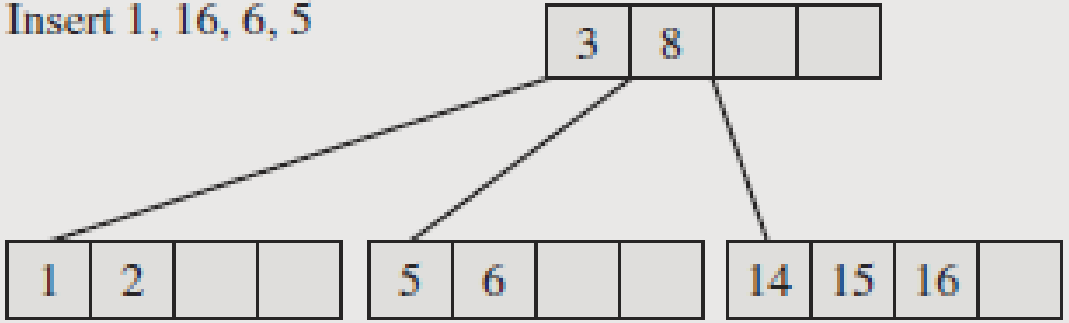
(a)

Insert 3

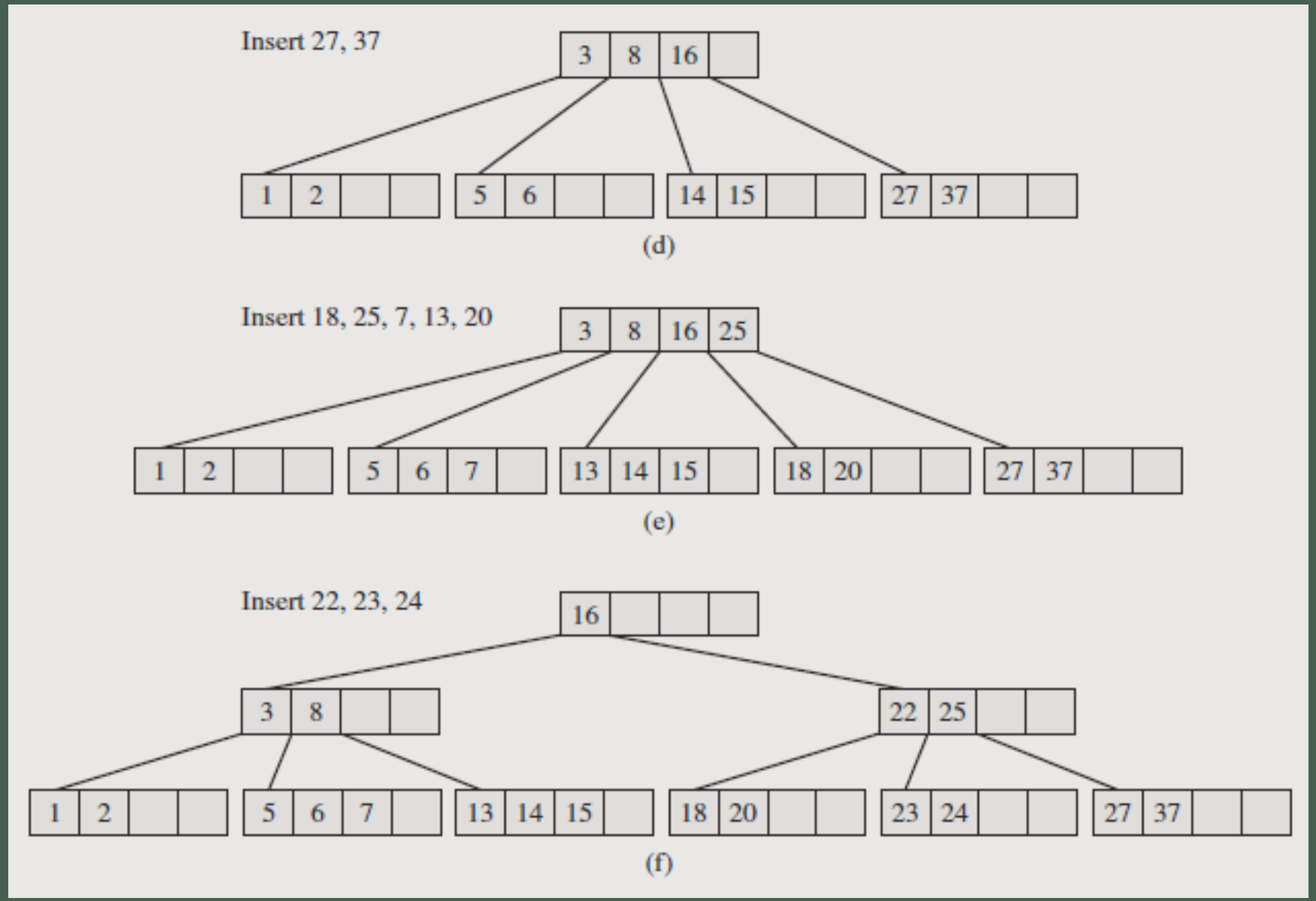


(b)

Insert 1, 16, 6, 5



(c)

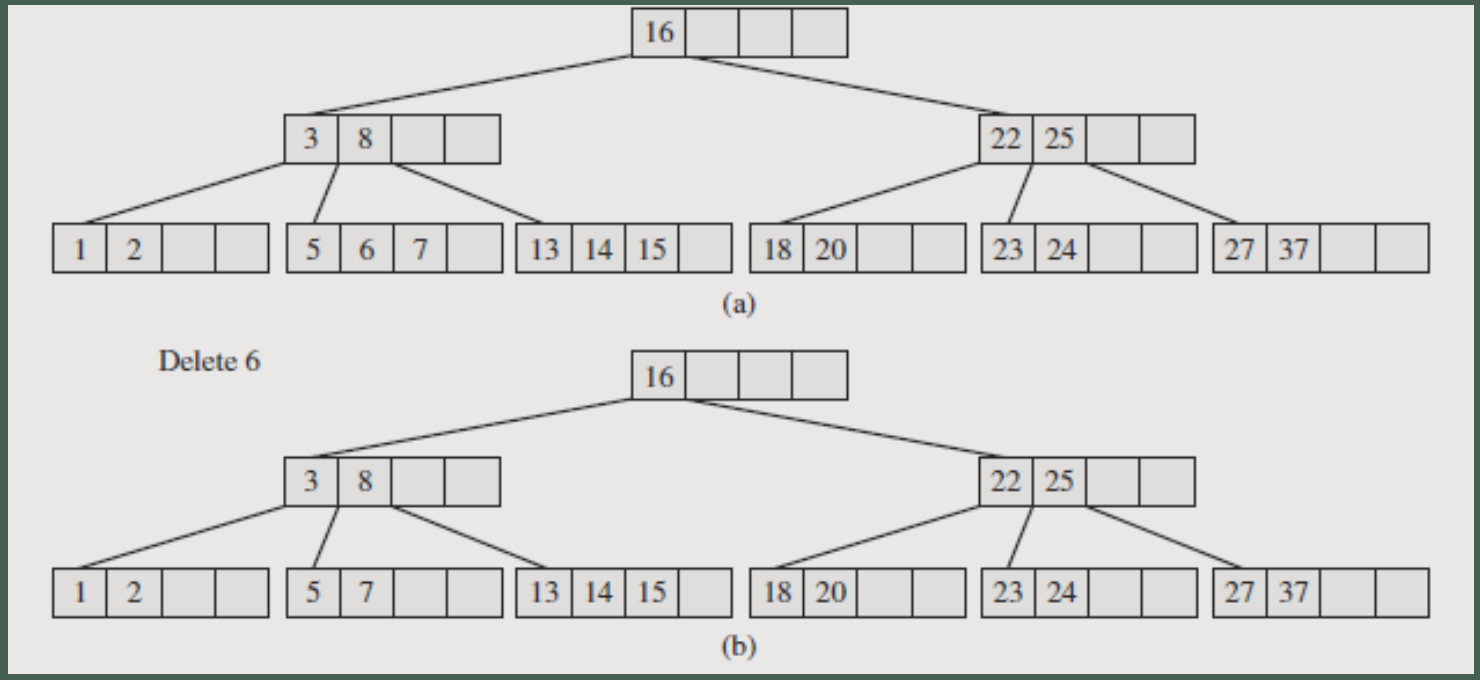

Insert 27, 37
3 8 16
1 2
5 6
14 15
27 37
(d)
Insert 18, 25, 7, 13, 20
3 8 16 25
1 2
5 6 7
13 14 15
18 20
27 37
(e)
Insert 22, 23, 24
16
3 8
22 25
1 2
5 6 7
13 14 15
18 20
23 24
27 37
(f)

Deleting a key from B-Tree

- Deletion is to a great extent a reversal of insertion, although it has more special cases.
- Care has to be taken to avoid allowing any node to be less than half full after a deletion. This means that nodes sometimes have to be merged.
- In deletion, there are two main cases:
  - deleting a key from a leaf and
  - Deleting a key from a nonleaf node (we will use a procedure similar to deleteByCopying()used for binary search trees).

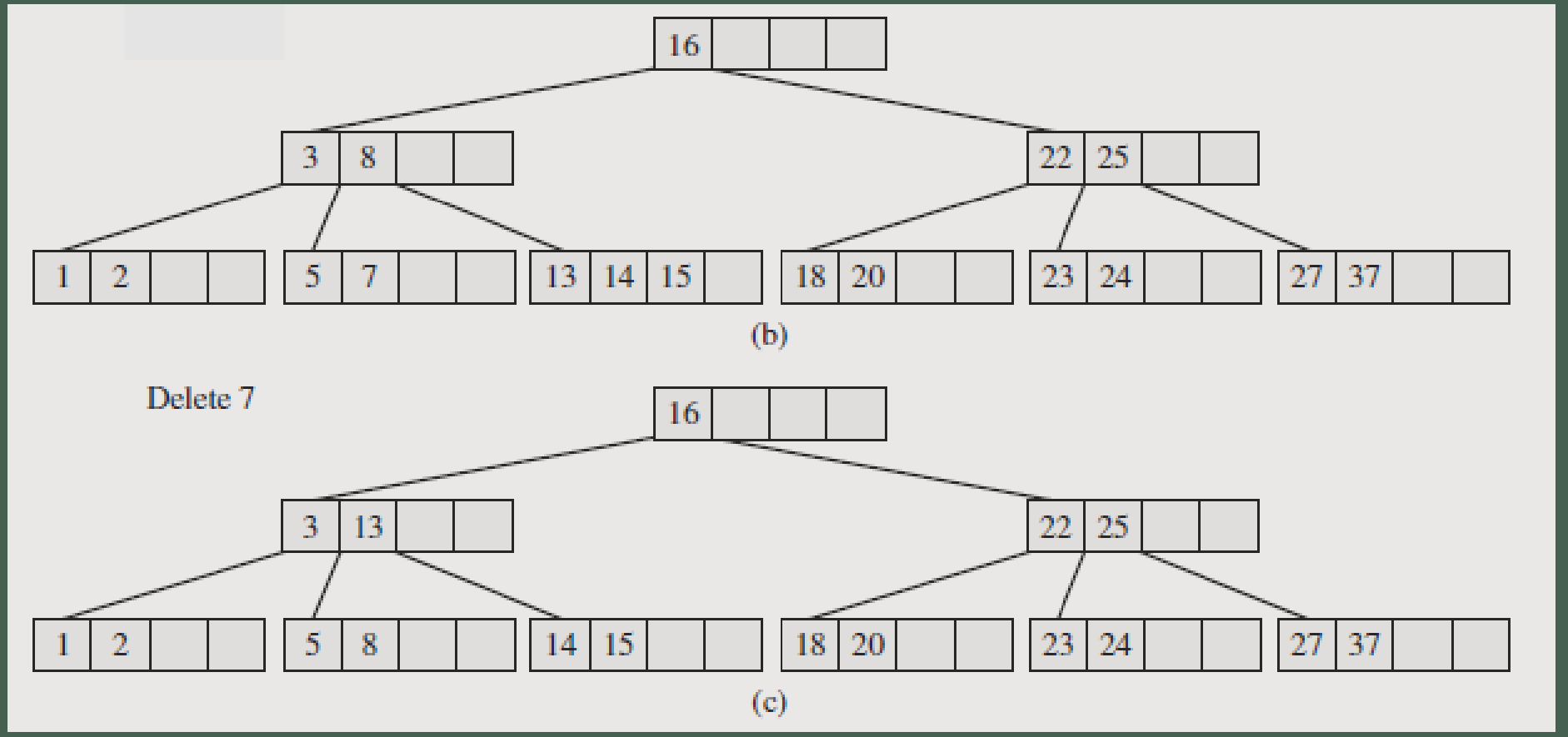
Deleting a Key from a Leaf

- 1.1: If, after deleting a key *K*, the leaf is at least half full and only keys greater than *K* are moved to the left to fill the hole, this is the inverse of insertion's case 1.



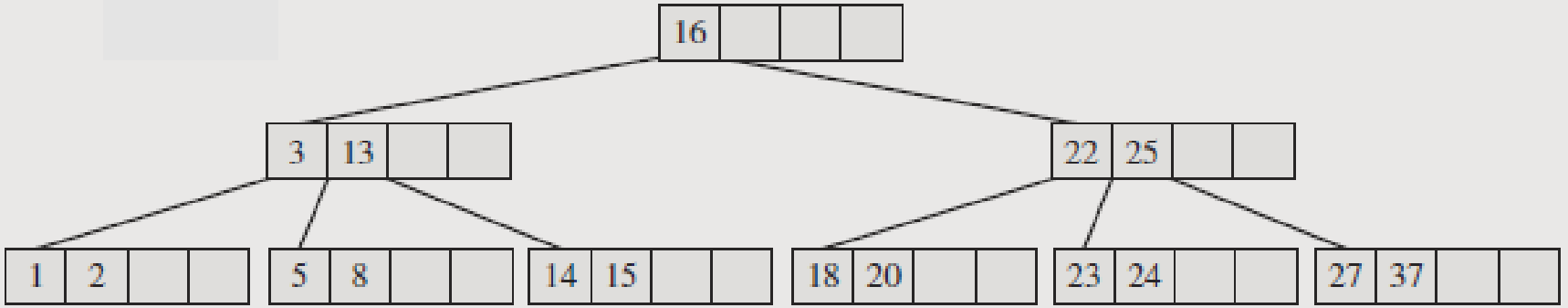
Deleting a Key from a Leaf

- 1.2: If, after deleting *K*, the number of keys in the leaf is less than $\lceil m/2 \rceil - 1$, causing an *underflow:*
- 1.2.1 If there is a left or right sibling with the number of keys exceeding the minimal $\lceil m/2 \rceil - 1$, then all keys from this leaf and this sibling are *redistributed* between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.


16
3
8
22
25
1
2
5
7
13
14
15
18
20
23
24
27
37
(b)
Delete 7
16
3
13
22
25
1
2
5
8
14
15
18
20
23
24
27
37
(c)

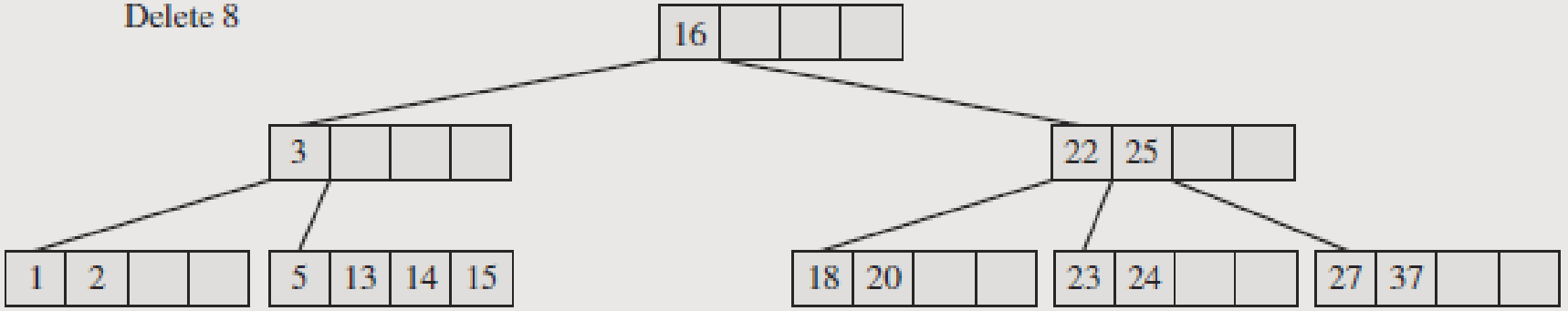
Deleting a Key from a Leaf

- **1.2.2** If the leaf underflows and the number of keys in its siblings is $\lceil m/2 \rceil - 1$, then the leaf and a sibling are *merged;* the keys from the leaf, from its sibling, and the separating key from the parent are all put in the leaf, and the sibling node is discarded.
- The keys in the parent are moved if a hole appears. This can initiate a chain of operations if the parent underflows.
- The parent is now treated as though it were a leaf, and either step 1.2.2 is repeated until step 1.2.1 can be executed or the root of the tree has been reached. This is the inverse of insertion's case 2.



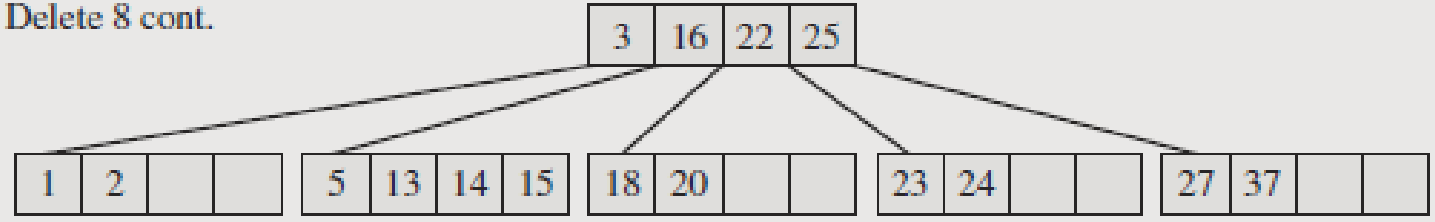
(c)

Delete 8



(d)

Delete 8 cont.



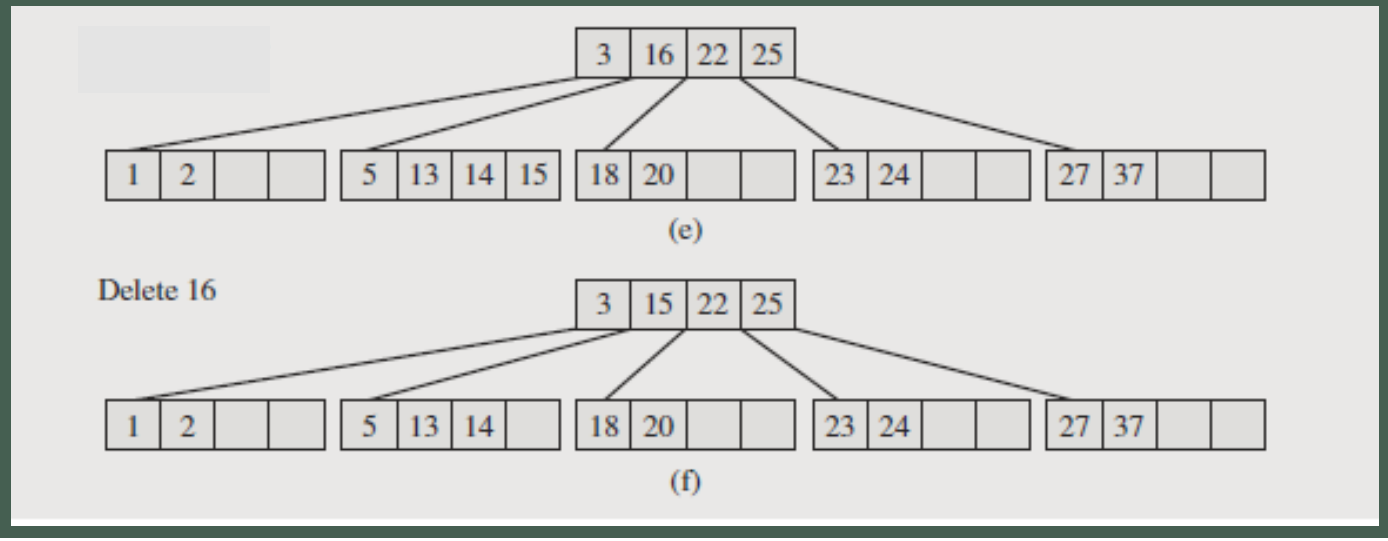
(e)

Deleting a Key from a Leaf

- 1.2.2.1 A particular case results in merging a leaf or nonleaf with its sibling when its parent is the root with only one key.
- In this case, the keys from the node and its sibling, along with the only key of the root, are put in the node, which becomes a new root, and both the sibling and the old root nodes are discarded. This is the only case when two nodes disappear at one time. Also, the height of the tree is decreased by one (shown in the previous slide).
- This is the inverse of insertion’s case 3.

Deleting a Key from Non-leaf

- This may lead to problems with tree reorganization. Therefore, deletion from a nonleaf node is reduced to deleting a key from a leaf.
- The key to be deleted is replaced by its immediate predecessor (the successor could also be used), which can only be found in a leaf. This successor key is deleted from the leaf, which brings us to the preceding case 1.



```
BTreeDelete(K)
    node = BTreeSearch(K,root);
    if (node != null)
        if node is not a leaf
            find a leaf with the closest predecessor S of K;
            copy S over K in node;
            node = the leaf containing S;
            delete S from node;
        else delete K from node;
        while (1)
            if node does not underflow
                return;
            else if there is a sibling of node with enough keys
                redistribute keys between node and its sibling;
                return;
            else if node's parent is the root
                if the parent has only one key
                    merge node, its sibling, and the parent to form a new root;
                else merge node and its sibling;
                return;
            else merge node and its sibling;
                node = its parent;
```