Multiway Trees

7

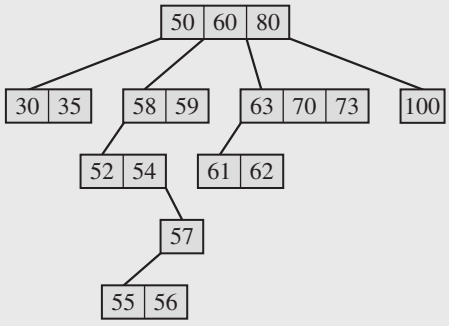
At the beginning of the preceding chapter, a general definition of a tree was given, but the thrust of that chapter was binary trees, in particular, binary search trees. A tree was defined as either an empty structure or a structure whose children are disjoint trees $t_1, \ldots, t_m$. According to this 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.*

In a more useful version of a multiway tree, an order is imposed on the keys residing in each node. A *multiway search tree of order m,* or an *m-way search tree,* is a multiway tree in which

1. Each node has m children and $m - 1$ keys.
2. The keys in each node are in ascending order.
3. The keys in the first i children are smaller than the ith key.
4. The keys in the last $m - i$ children are larger than the ith key.

The m-way search trees play the same role among m-way trees that binary search trees play among binary trees, and they are used for the same purpose: fast information retrieval and update. The problems they cause are similar. The tree in Figure 7.1 is a 4-way tree in which 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. Constructing such trees requires a more careful approach.

FIGURE 7.1 A 4-way tree.



7.1 The Family of B-Trees

The basic unit of I/O operations associated with a disk is a block. When information is read from a disk, the entire block containing this information is read into memory, and when information is stored on a disk, an entire block is written to the disk. Each time information is requested from a disk, this information has to be located on the disk, the head has to be positioned above the part of the disk where the information resides, and the disk has to be spun so that the entire block passes underneath the head to be transferred to memory. This means that there are several time components for data access:

$$\textit{access time} = \textit{seek time} + \textit{rotational delay (latency)} + \textit{transfer time}$$

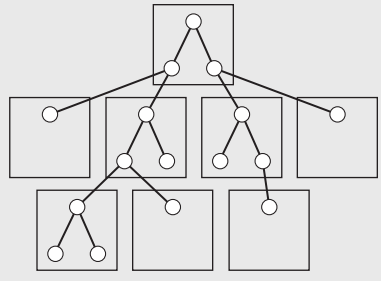
This process is extremely slow compared to transferring information within memory. The first component, *seek time,* is particularly slow because it depends on the mechanical movement of the disk head to position the head at the correct track of the disk. *Latency* is the time required to position the head above the correct block, and on the average, it is equal to the time needed to make one-half of a revolution. For example, the time needed to transfer 5KB (kilobytes) from a disk requiring 40 ms (milliseconds) to locate a track, making 3,000 revolutions per minute and with a data transfer rate of 1,000KB per second, is

$$\textit{access time} = 40 \text{ ms} + 10 \text{ ms} + 5 \text{ ms} = 55 \text{ ms}$$

This example indicates that transferring information to and from the disk is on the order of milliseconds. On the other hand, the CPU processes data on the order of microseconds, 1,000 times faster, or on the order of nanoseconds, 1 million times faster, or even faster. We can see that processing information on secondary storage can significantly decrease the speed of a program.

If a program constantly uses information stored in secondary storage, the characteristics of this storage have to be taken into account when designing the program. For example, a binary search tree can be spread over many different blocks on a disk, as in Figure 7.2, so that an average of two blocks have to be accessed. When the tree is used frequently in a program, these accesses can significantly slow down the execution time of the program. Also, inserting and deleting keys in this tree require many block accesses. The binary search tree, which is such an efficient tool when it resides entirely in memory, turns out to be an encumbrance. In the context of secondary storage, its otherwise good performance counts very little because the constant accessing of disk blocks that this method causes severely hampers this performance.

FIGURE 7.2 Nodes of a binary tree can be located in different blocks on a disk.



It is also better to access a large amount of data at one time than to jump from one position on the disk to another to transfer small portions of data. For example, if 10KB have to be transferred, then using the characteristics of the disk given earlier, we see that

$$\textit{access time} = 40 \text{ ms} + 10 \text{ ms} + 10 \text{ ms} = 60 \text{ ms}$$

However, if this information is stored in two 5KB pieces, then

$$\textit{access time} = 2 \cdot (40 \text{ ms} + 10 \text{ ms} + 5 \text{ ms}) = 110 \text{ ms}$$

which is nearly twice as long as in the previous case. The reason is that each disk access is very costly; if possible, the data should be organized to minimize the number of accesses.

7.1.1 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.

A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage. One important property of B-trees is the size

of each node, which can be made as large as the size of a block. The number of keys in one node can vary depending on the sizes of the keys, organization of the data (are only keys kept in the nodes or entire records?), and of course, on the size of a block. Block size varies for each system. It can be 512 bytes, 4KB, or more; block size is the size of each node of a B-tree. The amount of information stored in one node of the B-tree can be rather large.

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 nonroot and each nonleaf 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.[1]

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

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, as in

```
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>;
};
```

Usually, m is large (50–500) so that information stored in one page or block of secondary storage can fit into one node. Figure 7.3a 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, possibly variant records (unions). In these cases, the array `keys` is an array of objects, each having a unique identifier field (such as the identifying code in Figure 7.3a) and an address of the entire record on secondary storage, as in

[1] In this definition, the order of a B-tree specifies the *maximum* number of children. Sometimes nodes of a B-tree of order m are defined as having k keys and $k + 1$ pointers where $m \leq k \leq 2m$, which specifies the *minimum* number of children.

FIGURE 7.3 One node of a B-tree of order 7 (a) without and (b) with an additional indirection.

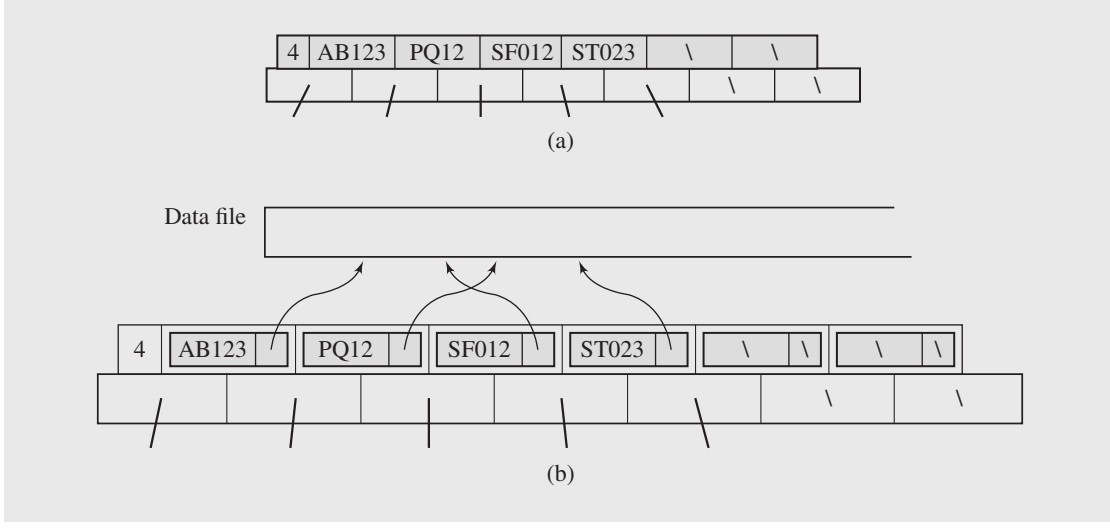
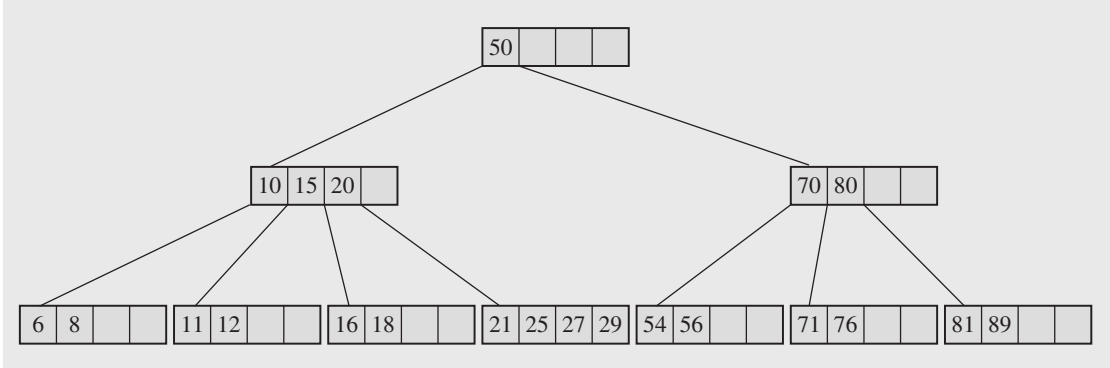


Figure 7.3b. If the contents of one such node also reside in secondary storage, each key access would require two secondary storage accesses. In the long run, this is better than keeping the entire records in the nodes, because in this case, the nodes can hold a very small number of such records. The resulting B-tree is much deeper, and search paths through it are much longer than in a B-tree with the addresses of records.

From now on, B-trees will be shown in an abbreviated form without explicitly indicating `keyTally` or the pointer fields, as in Figure 7.4.

FIGURE 7.4 A B-tree of order 5 shown in an abbreviated form.



Searching in a B-Tree

An algorithm for finding a key in a B-tree is simple, and is coded as follows:

```
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;
}
```

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

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

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

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

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. If the root can be kept in memory at all times, this number can be reduced to only three seeks into secondary storage.

Inserting a Key into a B-Tree

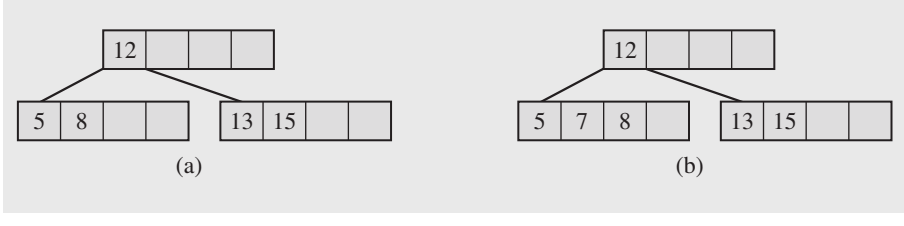
Both the insertion and deletion operations appear to be somewhat challenging if we remember that all leaves have to be at the last level. Not even balanced binary trees require that. Implementing insertion becomes easier when the strategy of building a tree is changed. When inserting a node into a binary search tree, the tree is always built from top to bottom, resulting in unbalanced trees. If the first incoming key is the smallest, then this key is put in the root, and the root does not have a left subtree unless special provisions are made to balance the tree.

But a tree can be built from the bottom up so that the root is an entity always in flux, and only at the end of all insertions can we know for sure the contents of the root. This strategy is applied to inserting keys into B-trees. In this process, 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.

To approach the problem more systematically, there are three common situations encountered when inserting a key into a B-tree.

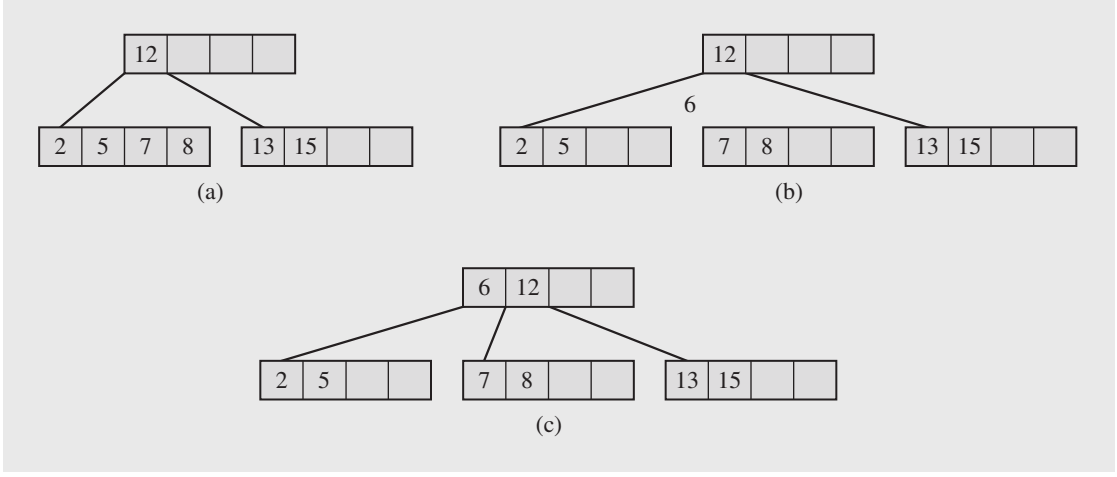
1. A key is placed in a leaf that still has some room, as in Figure 7.5. In a B-tree of order 5, a new key, 7, is placed in a leaf, preserving the order of the keys in the leaf so that key 8 must be shifted to the right by one position.

FIGURE 7.5 A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells.



2. The leaf in which a key should be placed is full, as in Figure 7.6. 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.
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 7.7a, 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 7.7b). So the parent is split

FIGURE 7.6 Inserting the number 6 into a full leaf.



(Figure 7.7c), 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 7.7d). It should be obvious that it is the only case in which the B-tree increases in height.

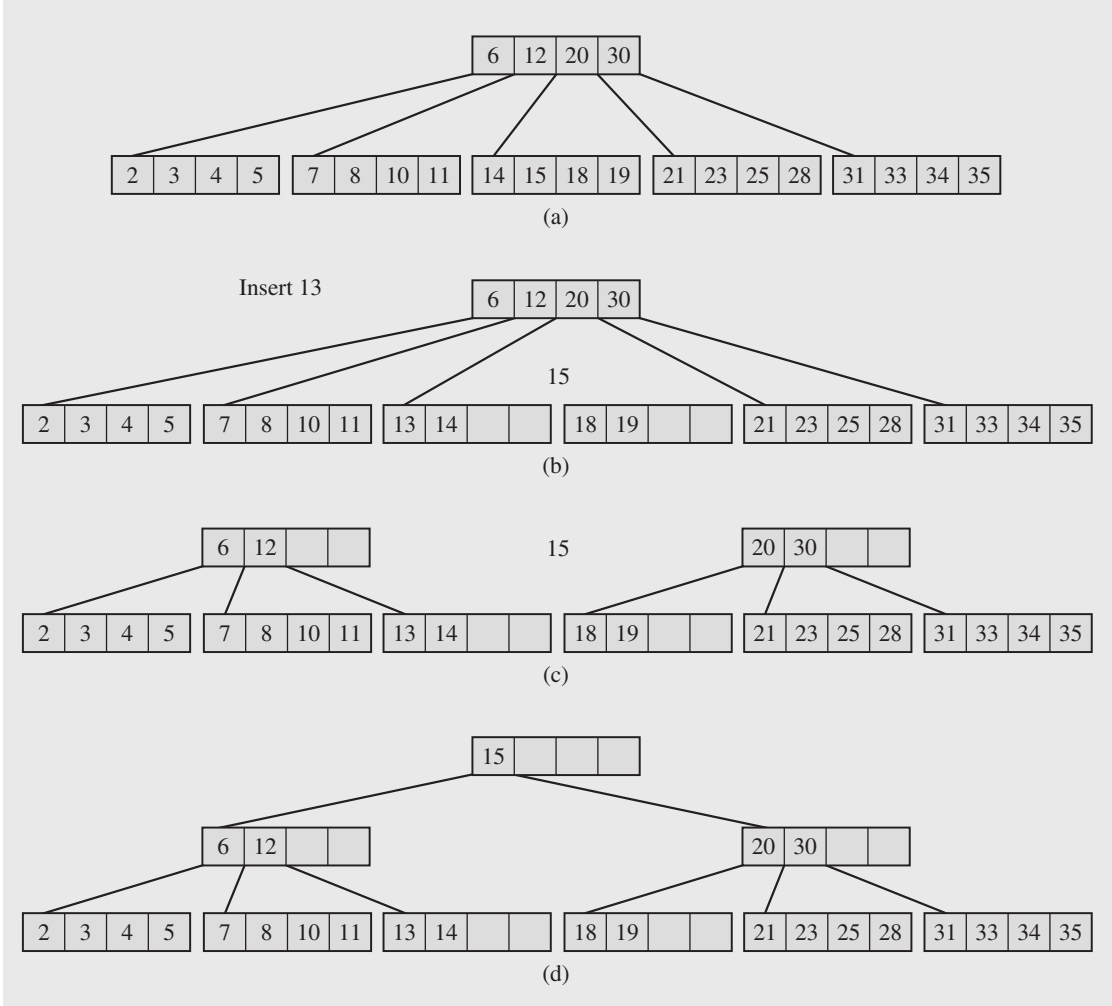
An algorithm for inserting keys in B-trees follows:

```
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;
```

Figure 7.8 shows the growth of a B-tree of order 5 in the course of inserting new keys. Note that at all times the tree is perfectly balanced.

A variation of this insertion strategy uses *presplitting:* when a search is made from the top down for a particular key, each visited node that is already full is split. In this way, no split has to be propagated upward.

FIGURE 7.7 Inserting the number 13 into a full leaf.



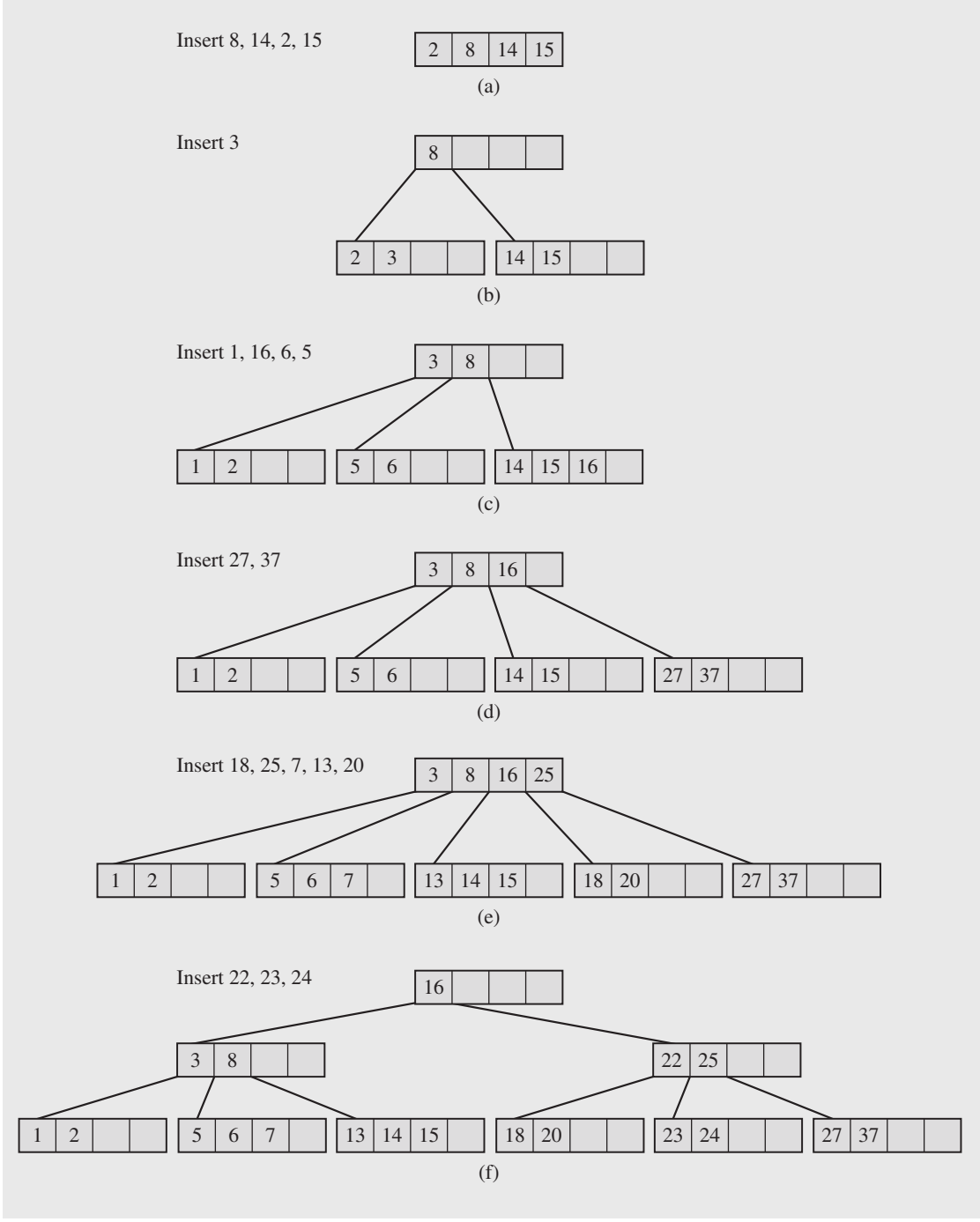
How often are node splits expected to occur? A split of the root node of a B-tree creates two new nodes. All other splits add only one more node to the B-tree. During the construction of a B-tree of p nodes, $p - h$ splits have to be performed, where h is the height of the B-tree. Also, in a B-tree of p nodes, there are at least

$$1 + (\lceil m/2 \rceil - 1)(p - 1)$$

keys. The rate of splits with respect to the number of keys in the B-tree can be given by

$$\frac{p - h}{1 + (\lceil m/2 \rceil - 1)(p - 1)}$$

FIGURE 7.8 Building a B-tree of order 5 with the `BTreeInsert()` algorithm.



After dividing the numerator and denominator by $p - h$ and observing that $\frac{1}{p-h} \to 0$ and $\frac{p-1}{p-h} \to 1$ with the increase of p, the average probability of a split is

$$\frac{1}{\lceil m/2 \rceil - 1}$$

For example, for $m = 10$, this probability is equal to .25; for $m = 100$, it is .02; and for $m = 1{,}000$, it is .002, and expectedly so: the larger the capacity of one node, the less frequently splits occur.

Deleting a Key from a B-Tree

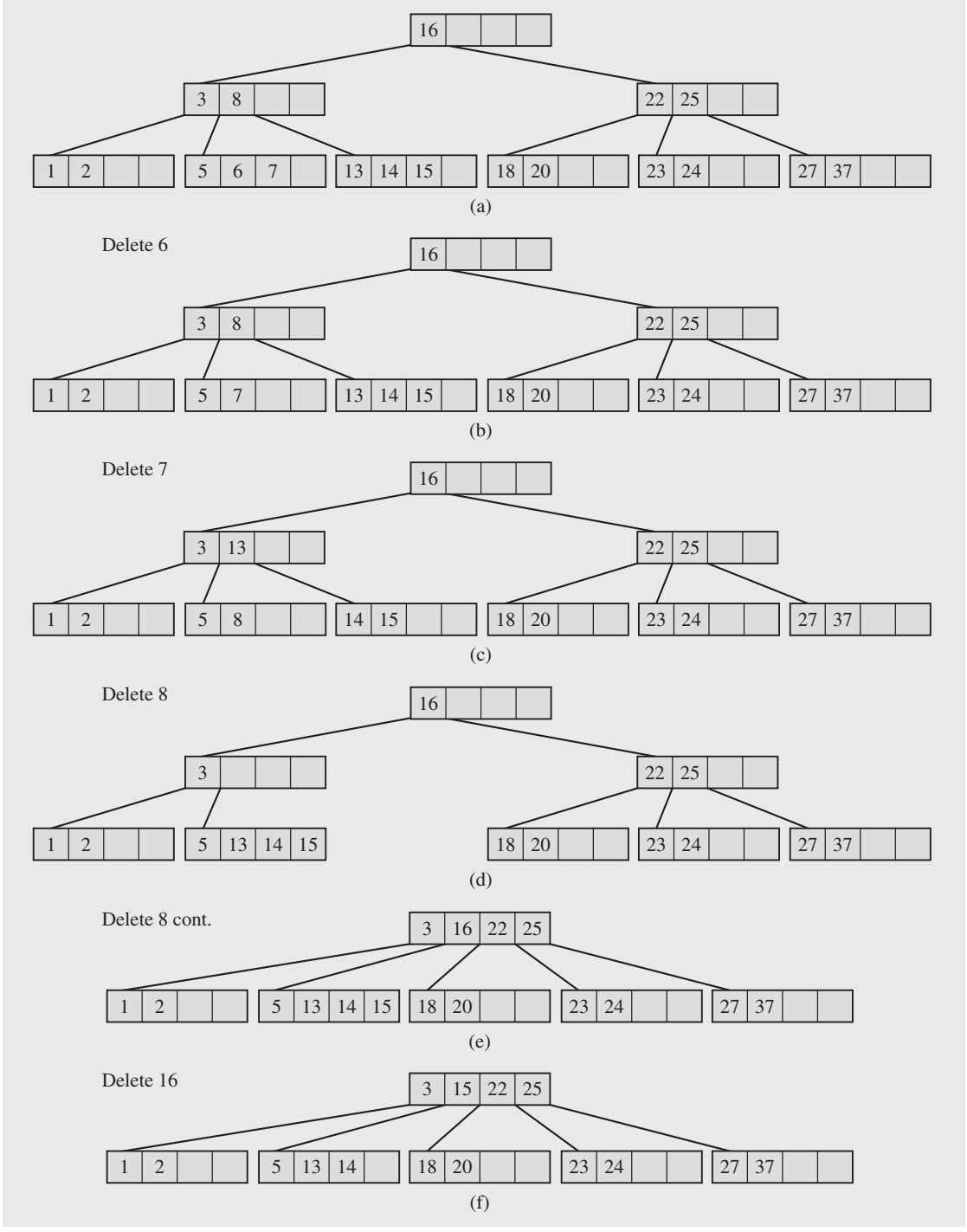
Deletion is to a great extent a reversal of insertion, although it has more special cases. Care has to be taken to avoid allowing any node to be less than half full after a deletion. This means that nodes sometimes have to be merged.

In deletion, there are two main cases: deleting a key from a leaf and deleting a key from a nonleaf node. In the latter case, we will use a procedure similar to `deleteByCopying()` used for binary search trees (Section 6.6).

1. 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 (see Figures 7.9a–b), this is the inverse of insertion's case 1.
- 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 (see Figures 7.9b–c).
  - 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 (see Figures 7.9c–d). 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.
    - 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 (see Figures 7.9c–e). This is the inverse of insertion's case 3.

FIGURE 7.9 Deleting keys from a B-tree.



2. Deleting a key from a nonleaf. 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 (see Figures 7.9e–f).

   Here is the deletion algorithm:

```
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;
```

B-trees, according to their definition, are guaranteed to be at least 50% full, so it may happen that 50% of space is basically wasted. How often does this happen? If it happens too often, then the definition must be reconsidered or some other restrictions imposed on this B-tree. Analyses and simulations, however, indicate that after a series of numerous random insertions and deletions, the B-tree is approximately 69% full (Yao 1978), after which the changes in the percentage of occupied cells are very small. But it is very unlikely that the B-tree will ever be filled to the brim, so some additional stipulations are in order.

7.1.2 B*-Trees

Because each node of a B-tree represents a block of secondary memory, accessing one node means one access of secondary memory, which is expensive compared to accessing keys in the node residing in primary memory. Therefore, the fewer nodes that are created, the better.