CHAPTER 10
SEARCH STRUCTURES

10.1 Optimal binary search trees

- We introduce binary search trees in Chapter 5, and in this section we look at the construction of these search trees for a static set of identifiers
  - That is, we make no additions to or deletions from the tree; we only perform searches.
- We begin by examining the correspondence between a binary search tree and the binary search function.
  - For example, a binary search on the list (**do, if , while**) is equivalent to using the function (Program 5.17) on the binary search tree of Figure 10.1.
  - Although this is a full binary tree, it may not be an optimal binary search tree for this list if the identifiers are searched for with different frequency. (program 5.17, Fig 10.1)

Problem: Given 每個 key 的 access freq., 怎樣的binary search tree 所需的平均 access 次數，是最佳的?

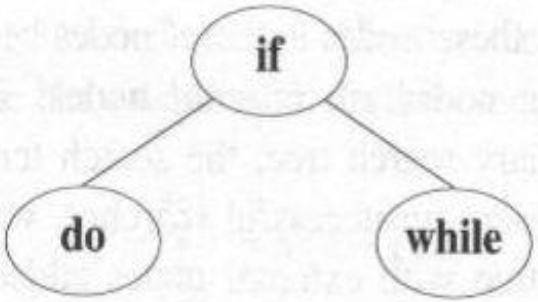


Figure 10.1: Binary search tree corresponding to a binary search on the list (**do, if, while**)

– To find an optimal binary search tree for a given static list, we must first decide on a cost measure for search tree.
  - Assume that we wish to search for an identifier at level *k* of a binary search tree using the *search2* function. We know that *search2* makes *k* iterations of the **while** loop.
  - Generally, the number of iteration of this loop equals the level number of the identifier we seek. Since the **while** loop determines the computing time of the search, it is reasonable to use the level number of a node as its cost.
– Consider the two search trees of Figure 10.2
  - The second binary tree has a better worst case search time than the first tree.
  - If we search for each identifier with equal probability, the average number of comparisons for successful search is 2.4. The comparisons for second tree is 2.2. Thus, the second tree also has better average behavior. (Fig 10.2)

設 **access freq. of all nodes are equal.**
則 **access "if",** 左圖需 **3** 次；右圖需 **2** 次
平均，左圖 **2.4 access/key;** 右圖 **2.2 access/key**

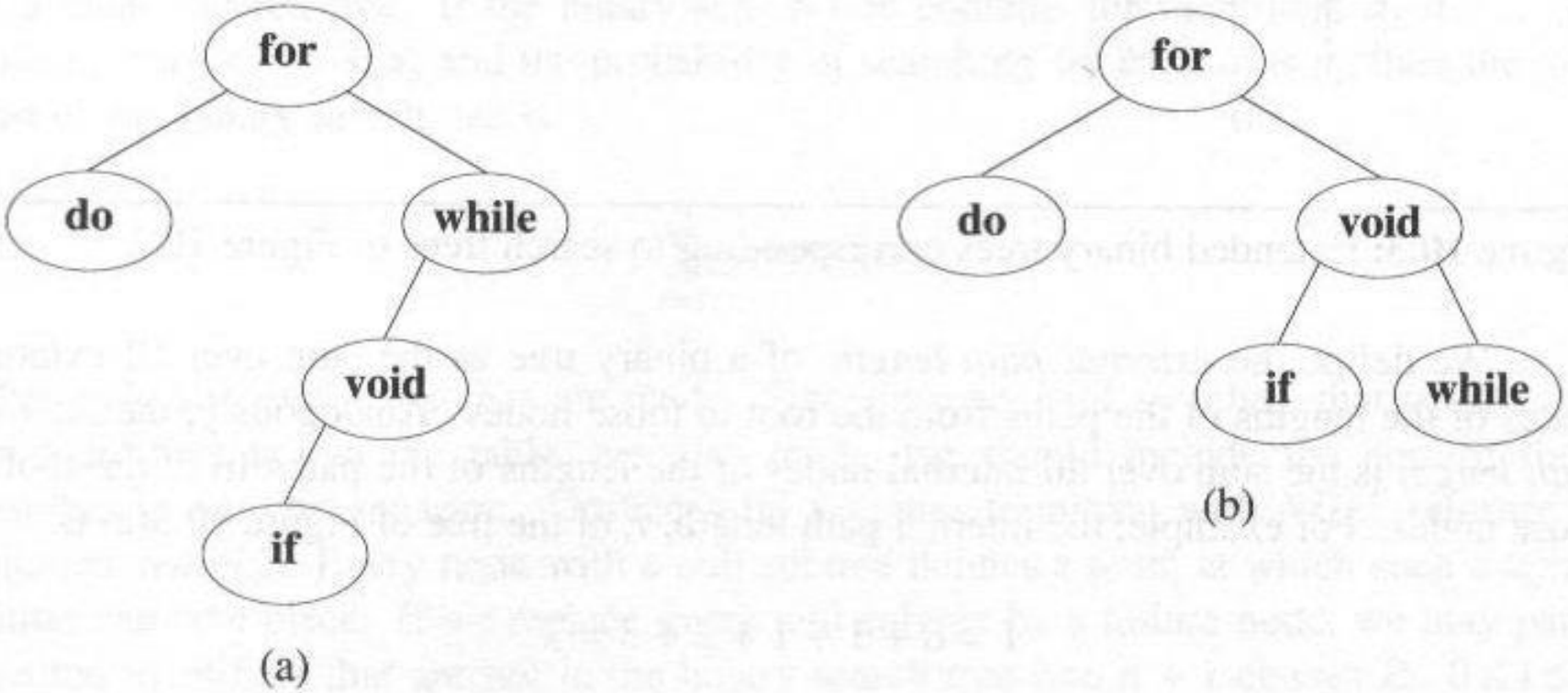


Figure 10.2: Two possible binary search trees

– In evaluating binary search trees, it is useful to add a special square node at every place there is a null links.
  - We call these nodes *external nodes (or failure nodes)*. The remaining nodes are *internal nodes.*
  - A binary tree with external nodes added is an *extended binary tree*.

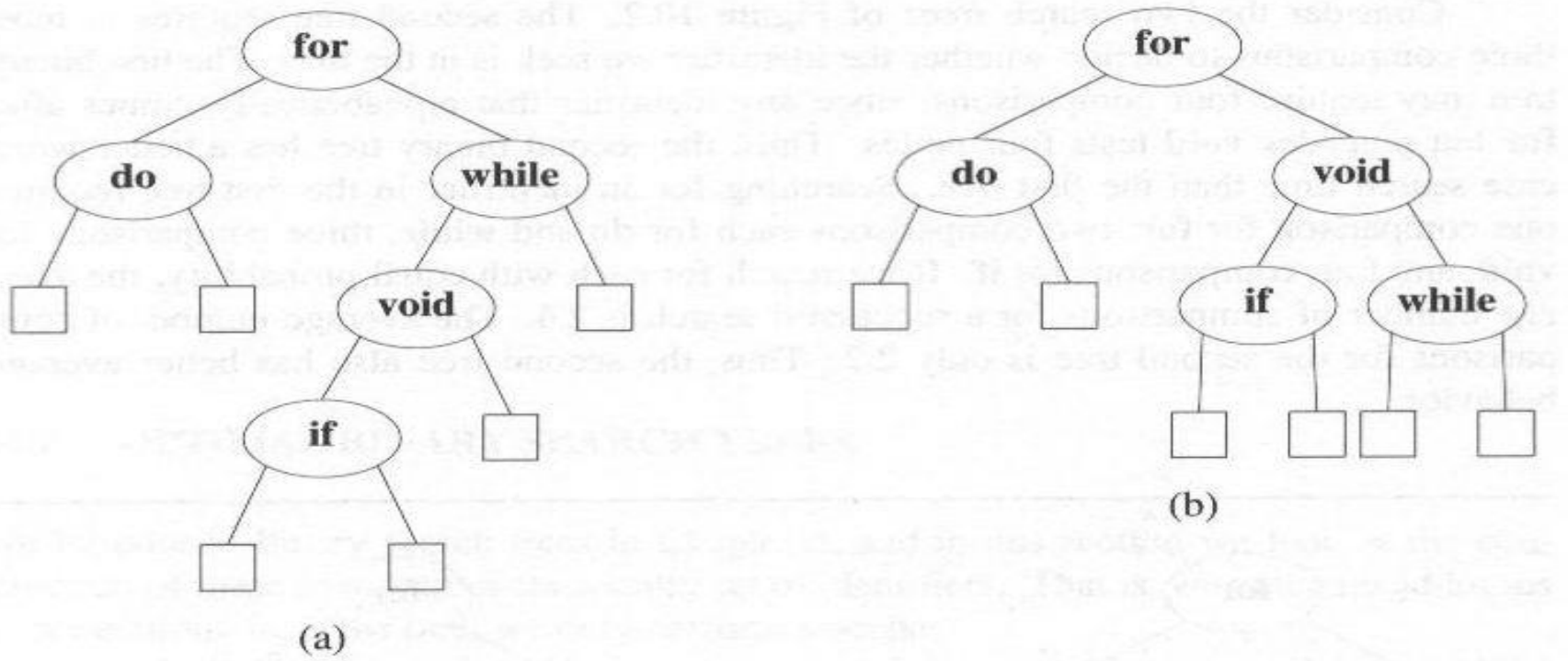


Figure 10.3: Extended binary trees corresponding to search trees of Figure 10.2

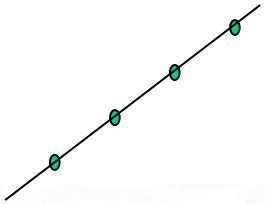
- We define the *external path length* of a binary tree as the sum overall external nodes of the lengths of the paths from the root to those nodes.
- Analogously, the *internal path length* is the sum over all internal nodes of the lengths of the paths from the root to those nodes
  - For example, the internal path length, *I*, of the tree of Figure 10.3(a) is: (到每一個 internal node 之路徑長度總和)

    $$I = 0 + 1 + 1 + 2 + 3 = 7$$

  - Its external path length, *E*, is : (到每一個 external node 之路徑長度總和)

    $$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

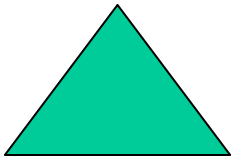
– The internal and external path length of a binary tree with n internal nodes are related by the formula $E = I + 2n$.
  - Hence, binary trees with the maximum E also have maximum I.
– What are the maximum and minimum possible values for I over all binary trees with n internal nodes?
  - The worst case occurs when the tree is skewed, that is, the tree has a depth of n.



$$I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

(I 的最大值)

- The best case occurs when we have as many internal nodes as close to the root as possible. One tree with minimal internal path length is the complete binary tree that we see that the distance of node i from the root is $\lfloor \log_2 i \rfloor$. Hence the smallest value for I is:



$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

(I 的最小值)

– If the binary search tree contains the identifiers a_1, a_2, …, a_n with $a_1 < a_2 < \ldots < a_n$ and the probability of searching for each a_i is p_i , then the total cost of any binary search tree is: (when only successful searches are made)

$$\sum_{i=1}^{n} p_i \cdot \text{level}(a_i)$$

(平均成功的 access 到 interal node 所需之次數)

– It is easy to see that for all identifiers in a particular class, E_i, the search terminates at the same failure node; it terminates at different failure nodes for identifiers in different classes.

– We may number the failure nodes form 0 to n with i being the failure node for class E_i, $0 \le i \le n$. If q_i is the probability that the identifier we are searching for is in E_i, then the cost of the failure node is:

$$\sum_{i=0}^{n} q_i \cdot (\text{level(failure node } i) - 1)$$

(每次沒找到 key 所需的 access 次數；－1 是因為不需要到達 failure node)

– Therefore, the total cost of a binary search tree is (i.e., objective function):

$$\sum_{i=1}^{n} p_i \cdot \text{level}(a_i) + \sum_{i=0}^{n} q_i \cdot (\text{level (failure node } i\text{)} - 1)$$

– An optimal binary search tree for the identifier set $a_1, \ldots, a_n$ is one that minimizes the above eq. over all possible binary search trees for this identifier set.

- Since all searches must terminate either successfully or unsuccessfully, we have

Subject to: $$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

以下頁的圖為例:

Example 10.1: Figure 10.4 shows the possible binary search trees for the identifier set (a_1, a_2, a_3) = (**do**, **if**, **while**). If we search for the identifiers with equal probabilities, $p_i = a_j = 1/7$ for all i and j, we have:

cost (tree a) = 15/7; cost (tree b) = 13/7 ⟸
cost (tree c) = 15/7; cost (tree d) = 15/7
cost (tree e) = 15/7

As expected, tree b is optimal. However, with p_1 =.5, p_2 = .1, p_3 = .05, q_0 = .15, q_1 = .1, q_2 = .05, and q_3 = .05, we have:

cost (tree a) = 2.65; cost (tree b) = 1.9
⟹ cost (tree c) = 1.5; cost (tree d) = 2.05
cost (tree e) = 1.6

Tree c is optimal with this assignment of p's and q's. □

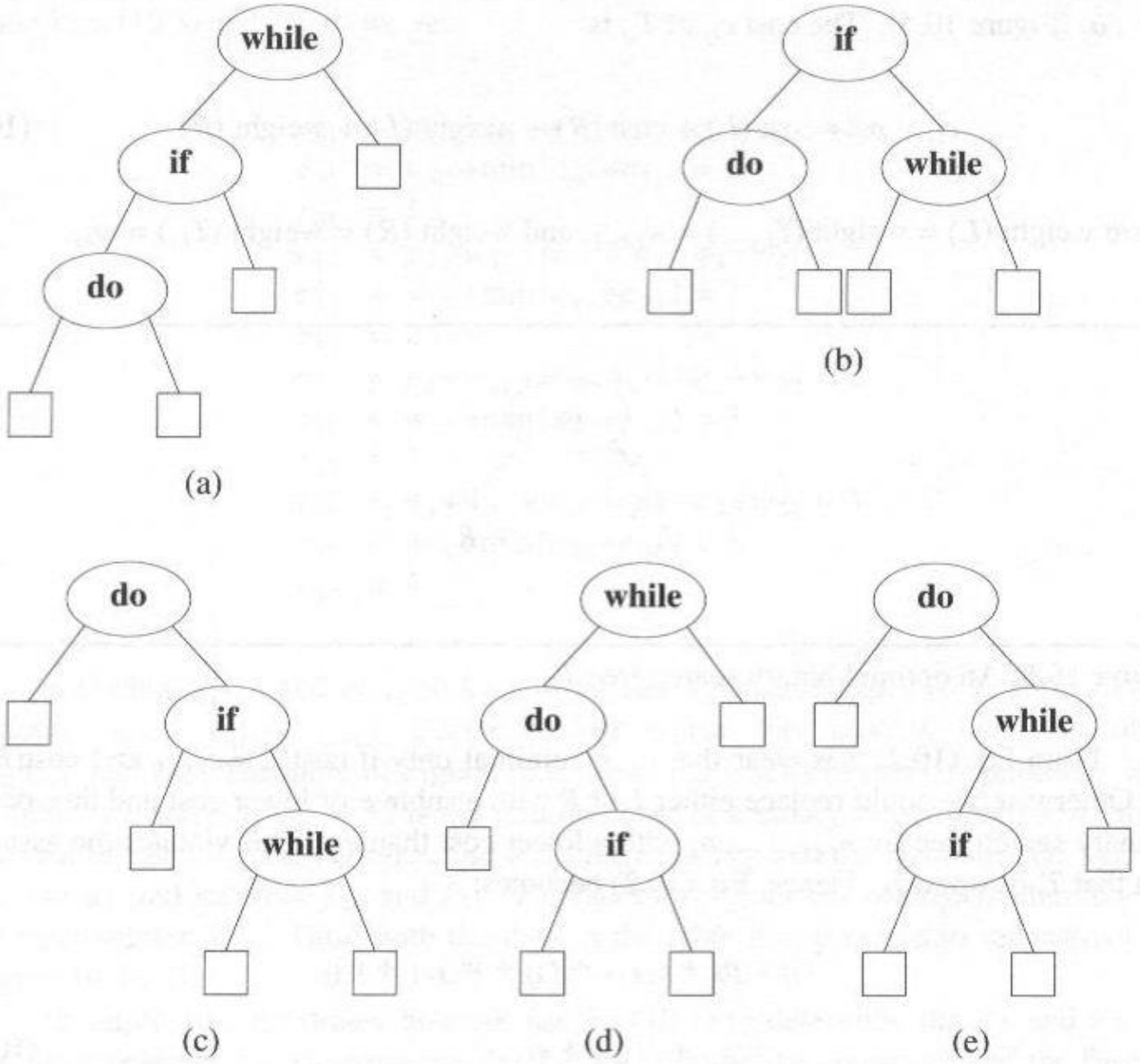


Figure 10.4: Binary search trees with three identifiers

(Dynamic Programming 作法 (供參考))

- How do we determine the optimal tree from all the possible binary search tree for a given set of identifiers?
  - We could proceed as in Example 10.1 and explicitly generate all possible binary search trees. Thus, we would compute the cost of each such tree and determine the optimal tree.
  - This brute force algorithm impractical for large values of *n*.
    - We can determine the cost of each of the binary search trees in O(n) time for an n node tree.
    - If *N(n)* is the total number of distinct binary search trees with *n* identifiers, the complexity of the algorithm is O(*n N*(*n*)).
    - From Section 5.10, we know that N(n) = O($4^n/n^{3/2}$).
    - The high complexity make this brute force algorithm impractical for large values of *n*.

– We can find a fairly efficient algorithm by making some observations about the properties of optimal binary search trees.

- Let $a_1 < a_2 < \ldots < a_n$ be the n identifiers represented in a binary search tree.
- Let T_{ij} denote an optimal binary search tree for $a_{i+1}, \ldots, a_j$, $i < j$. T_{ii} is an empty tree for $0 \le i \le n$ and T_{ij} is not defined for $i > j$.
- Let c_{ij} denote the cost of the search tree T_{ij}. By definition c_{ii} is 0.
- Let r_{ij} denote the root T_{ij} and let

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (q_k + p_k)$$

  denote the weight of T_{ij}. By definition, $r_{ii} = 0$ and $w_{ii} = q_i$, $0 \le i \le n$.
- T_{0n} is an optimal binary search for $a_1, \ldots, a_n$. Its cost is c_{0n}, its weight is w_{0n}, and its root is r_{0n}.

- If T_{ij} is an optimal binary search tree for $a_{i+1}, \ldots, a_j$ and $r_{ij} = k$, then k satisfies the inequality $i < k \leq j$. T has two subtrees L and R. L is the left subtree and contains the identifiers $a_{i+1}, \ldots, a_{k-1}$ and R is the subtree and contains the identifiers $a_{k+1}, \ldots, a_j$ (Figure 10.5).
- The cost c_{ij} of T_{ij} is
- $c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

- From Eq.(10.2) it is clear that c_{ij} is minimal only if cost(L) = $c_{i,k-1}$ and cost(R) = c_{kj} . Otherwise we could replace either L or R with a subtree of lower cost and thus obtain a binary search free for $a_{i+1}, \ldots, a_j$ with a lower cost than c_{ij}. This violates the assumption that T_{ij} is optimal.
- Hence, Eq. (10.2) becomes:
- $$c_{ij} = p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj}$$
  $$= w_{ij} + c_{i,k-1} + c_{kj}$$
- Since T_{ij} is optimal, it follows form Eq. (10.3) that $r_{ij} = k$ is such that

$$w_{ij} + c_{i,k-1} + c_{kj} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$

or

$$c_{i,k-1} + c_{kj} = \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$$

- Equation (10.4) show us how to obtain T_{0n} and c_{0n}, starting from the knowledge that $T_{ii} = \phi$ and $c_{ii} = 0$.
- **Example 10.2:** Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ (**do**, **for**, **void**, **while**). Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$. (We have multiplied the original p's and q's by 16 for convenience.) Initially, $w_{i,i} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \le i \le 4$. Using Eqs. (10.3) and (10.4) we get:

$$
\begin{aligned}
w_{01} &= p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8 \\
c_{01} &= w_{01} + \min\{c_{00} + c_{11}\} = 8 \\
r_{01} &= 1 \\
w_{12} &= p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7 \\
c_{12} &= w_{12} + \min\{c_{11} + c_{22}\} = 7 \\
r_{12} &= 2 \\
w_{23} &= p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3 \\
c_{23} &= w_{23} + \min\{c_{22} + c_{33}\} = 3 \\
r_{23} &= 3 \\
w_{34} &= p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3 \\
c_{34} &= w_{34} + \min\{c_{33} + c_{44}\} = 3 \\
r_{34} &= 4
\end{aligned}
$$

Knowing $w_{i,i+1}$ and $c_{i,i+1}$, $0 \le i < 4$ we can again use Eqs. (10.3) and (10.4) to compute $w_{i,i+2}$, $c_{i,i+2}$, $r_{i,i+2}$, $0 \le i < 3$. We repeat this process until we obtain w_{04}, c_{04}, and r_{04}. The table of Figure 10.6 shows the results of this computation. From the table, we see that $c_{04} = 32$ is the minimal cost of a binary search tree for a_1 to a_4. The root of tree T_{04} is a_2. Hence, the left subtree is T_{01} and the right subtree T_{24}. T_{01} has root a_1 and subtrees T_{00} and T_{11}. T_{24} has root a_3; its left subtree is therefore T_{22} and right subtree T_{34}. Thus, with the data in the table it is possible to reconstruct T_{04} (Figure 10.7). □

$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Computation is carried out row-wise from row 0 to row 4

Figure 10.6: Computation of c_{04} and r_{04}.

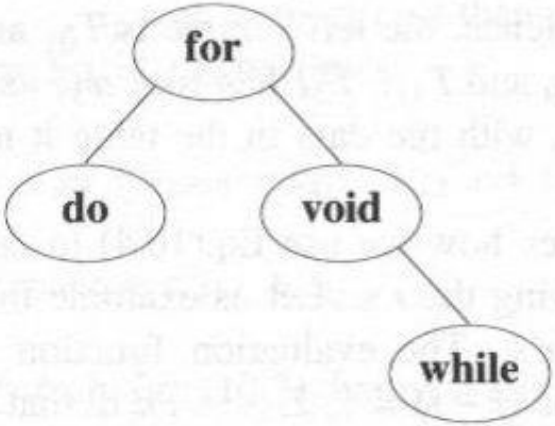


Figure 10.7: Optimal search tree for Example 10.2

- The total time to evaluate all the c_{ij}'s and r_{ij}'s is:

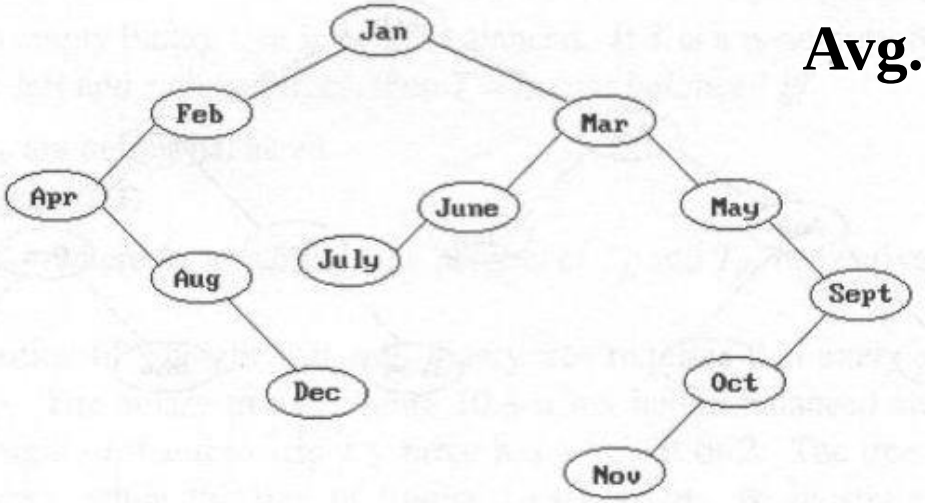
$$\sum_{m=1}^{n}(nm - m^2) = O(n^3)$$

- Actually we can do better than this by using a result attributed to D. E. Knuth.
  - He states that we can find the optimal l in Eq.(10.4) by limiting the search to the range $r_{i,j-1} \leq l \leq r_{i+1,j}$.
  - In this case, the computing time becomes $O(n^2)$ (Exercise 3)

10.2 AVL trees

– We also may maintain dynamic tables as binary search trees.

- In Chapter 5, we discussed how to insert elements into and delete them from binary search trees.
- Figure 10.8 shows the binary search tree obtained by entering the months *January* to *December*, in that order, into an initially empty binary search tree.
-



Avg. acc/key =3.5

Figure 10.8: Binary search tree obtained for the months of the year

- More discussion with Figure 10.8
  - The maximum number of comparisons needed to search for any identifier in the tree of Figure 10.8 is six (for *November*).
  - The average number of comparisons is = 42/12 = 3.5.
- If we enter the months into the tree in the order *July*, *February*, *May*, *August*, *January*, *March*, *October*, *April*, *December*, *June*, *November*, and *September*, we obtain the tree of Figure 10.9.
  - This tree is well balanced and does not have any paths to leaf nodes that are much longer than others.
  - In contrast, the tree of Figure 10.8 has six nodes on the path from the root to November, but only tree nodes on the path from the root to April.

- In Figure 10.9, the maximum number of comparisons needed to search for any identifier is now four, and the average number of comparisons is 37/12 ≈ 3.1.
- In addition, all the intermediate trees created during the construction of the tree of Figure 10.9 are also well balanced.

– Suppose that we now enter the months into an initially empty tree in alphabetical order. The tree degenerates into the chain shown in Figure 10.10.
  - The maximum search time is now 12 identifiers comparisons and the average is 6.5.
  - Thus, in the worst case, binary search trees correspond to sequential searching in an ordered list.
  - However, when we enter the identifiers in a random order, the tree tends to be balanced as in Figure 10.9.

- If all permutations are equally probable, then we can prove that the average search and insertion time is O(log*n*) for *n* node binary search tree.
- (Figure 10.9, 10.10)

Avg. acc/key =3.1

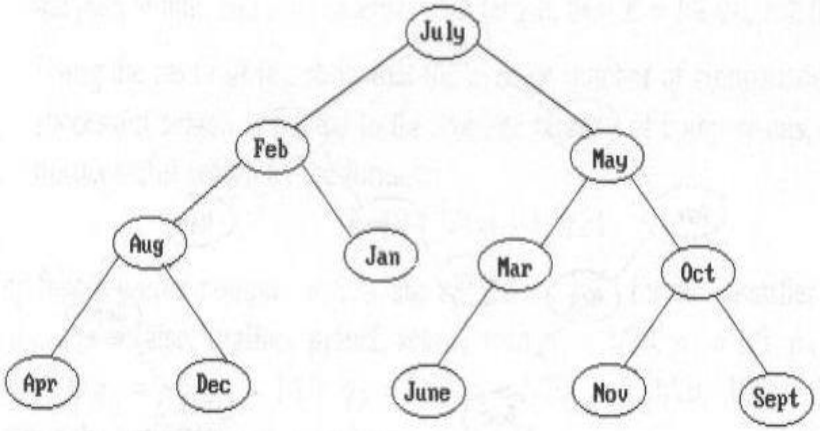


Figure 10.9: A balanced tree for the months of the year

Avg. acc/key =6.5

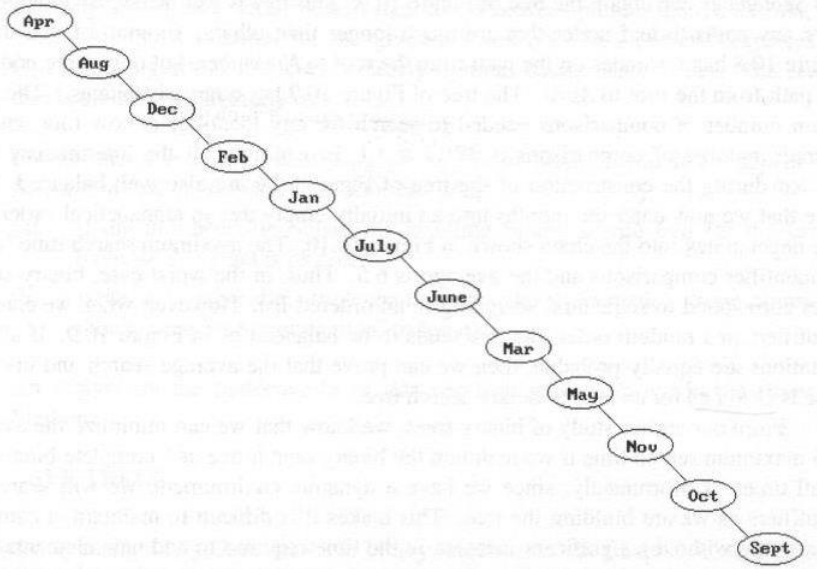


Figure 10.10: Degenerate binary search tree

– From our earlier study of binary trees, we know that we can minimize the average and maximum search time if we maintain the binary search tree as a complete binary tree at all times.
  - Unfortunately, since we have a dynamic environment, it is hard to achieve the objective if we maintain a complete binary tree without a significant increase in the time required to add new elements.
  - The increased time arises because, in some cases, we may have to restructure the entire tree to accommodate a new entry.
  - However, we can maintain a balanced tree that ensures that the average and worst case search time for a tree with n nodes is $O(\log n)$.

– In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of the subtrees.
  - Since the trees are balanced, we can perform dynamic retrievals in O(log*n*) time for a tree with n nodes. We also can enter an element into the tree, or delete an element form it, in O(log*n*) time. The resulting tree remain height balanced.
  - We refer to the trees introduced by Adelson-Velskii and Landis as *AVL trees*. As with binary trees, we may define AVL tree recursively.

– **Definition:**

An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is *height balanced iff*

1. T_L and T_R are height balanced, and
2. $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

– The definition of a height balanced binary tree requires that every subtree also be height balanced.

- Example:
  – The binary tree of Figure 10.8 is not height balanced.
  – The binary tree of Figure 10.9 is height balanced.
  – The binary tree of Figure 10.10 is not height balanced.

– To illustrate the processes involved in maintaining a height balanced binary search tree, let us construct such a tree for the months of the year.
  - This time we will insert the months into the tree in the order *March*, *May*, *November*, *August*, *April*, *January*, *December*, *July*, *February*, *June*, *October*, and *September*.
  - Figure 10.11 shows the tree as it grows, and the restructuring involved in keeping it balanced.
  - The numbers by each node represent the difference in heights between the left and right subtrees of that node. We refer to this as the balance factor of the node.

– **Definition:**

The balance factor, *BF(T)*, of a node, *T*, in a binary tree is defined as h_L - h_R, where h_L and h_R are, respectively, the heights of the left and right subtrees of *T*. For any node *T* in an AVL tree *BF(T)* = -1, 0, or 1.

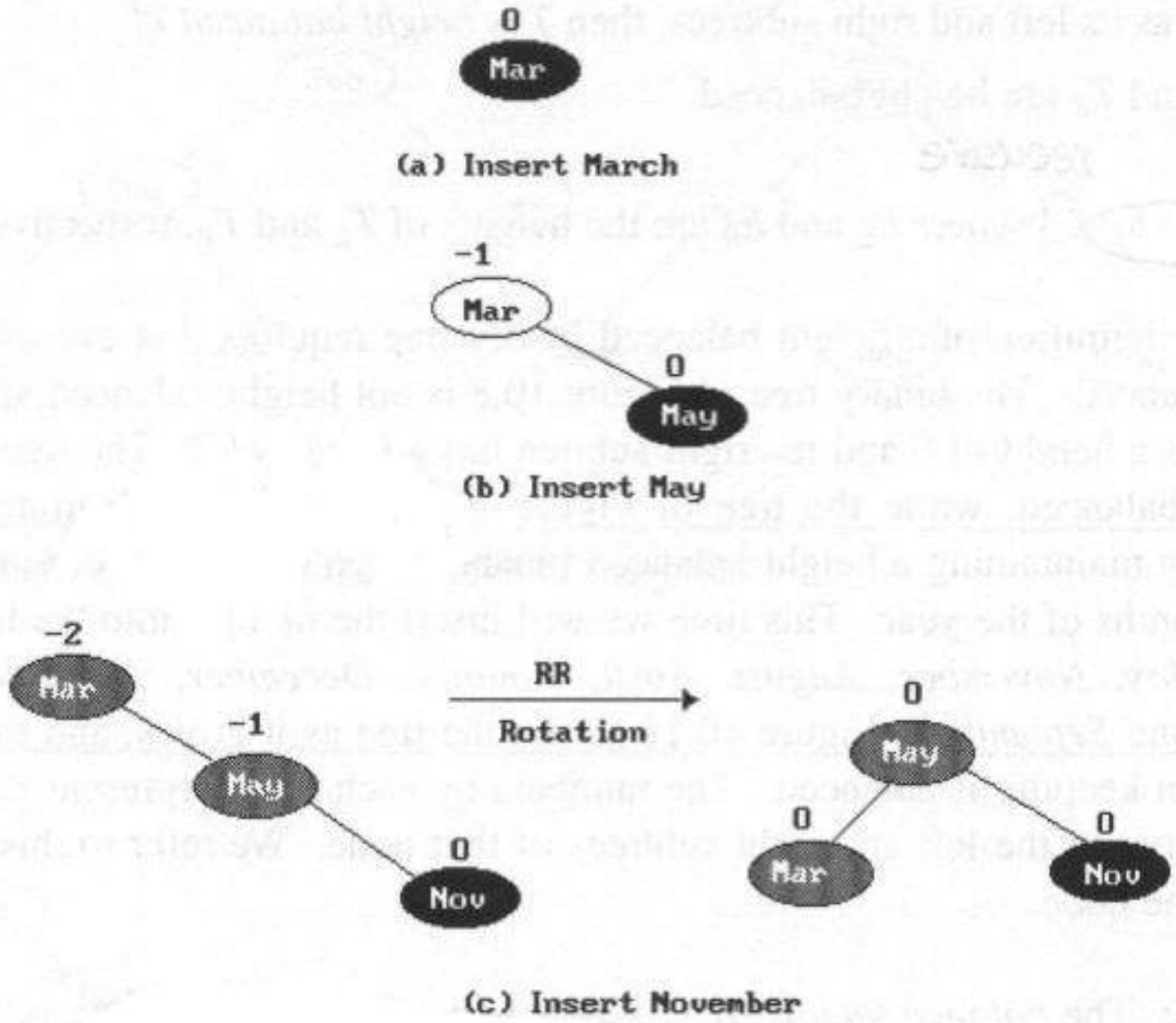


Figure 10.11: Insertion into an AVL tree

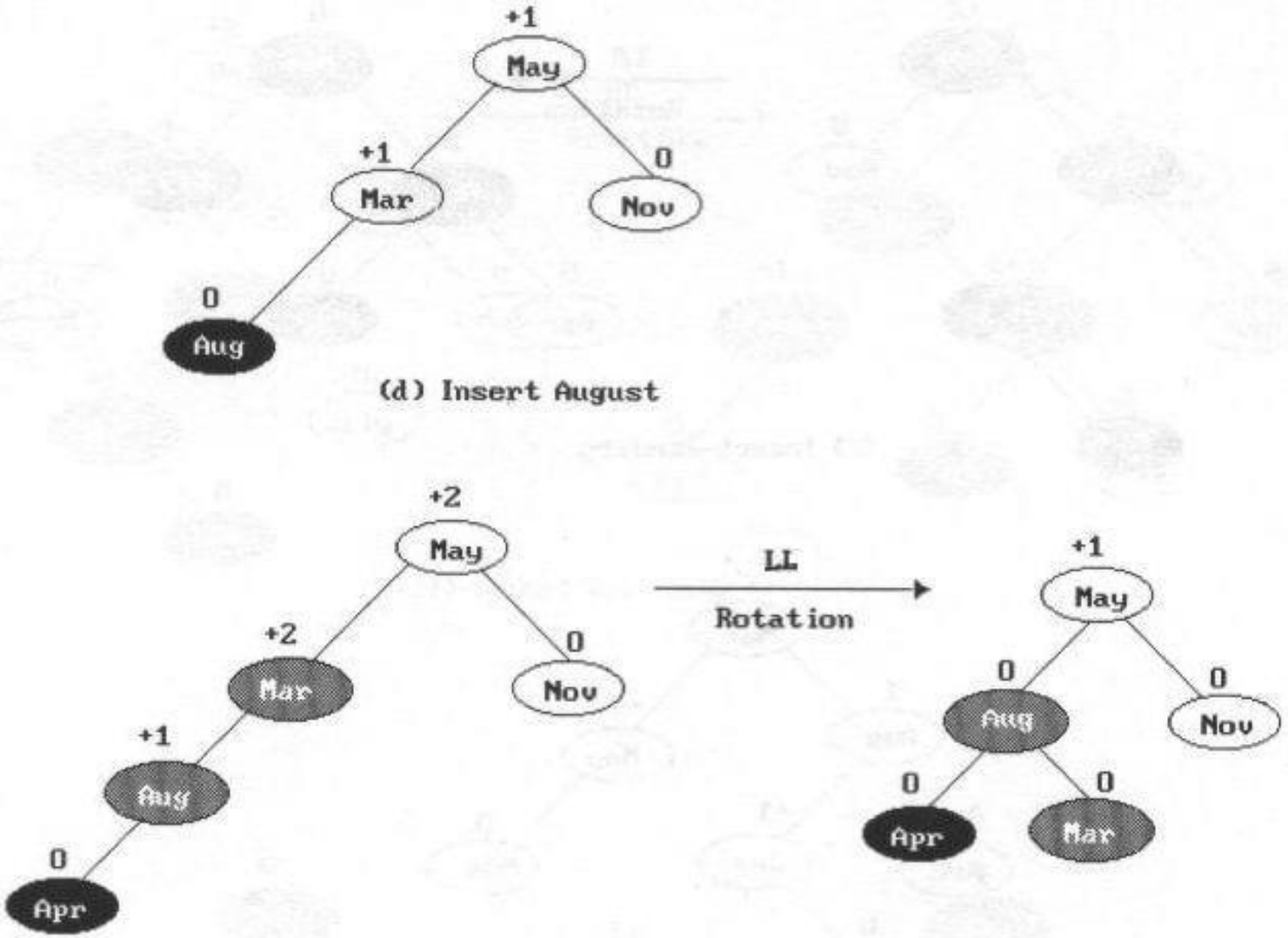
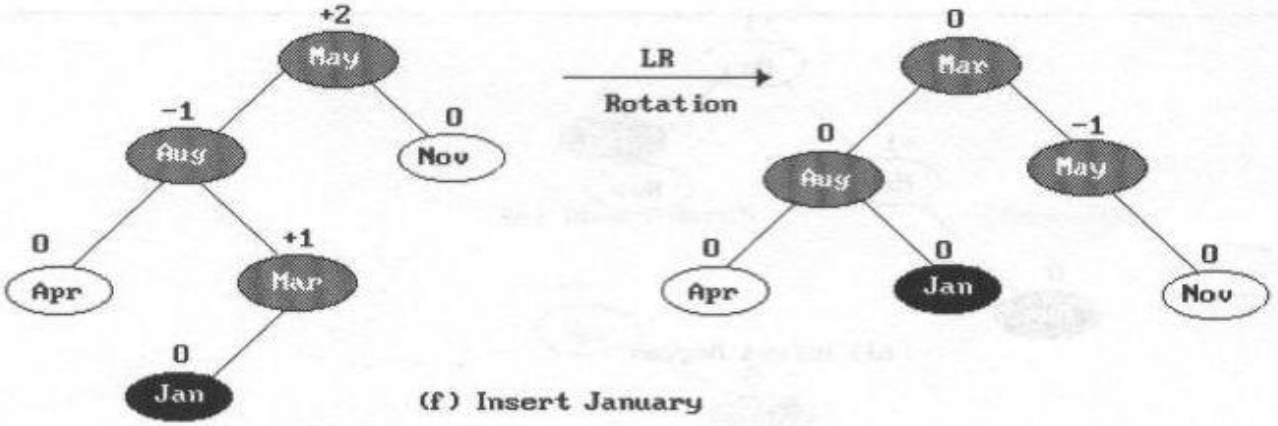
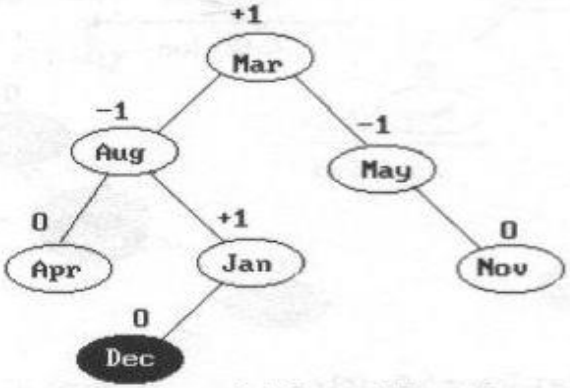


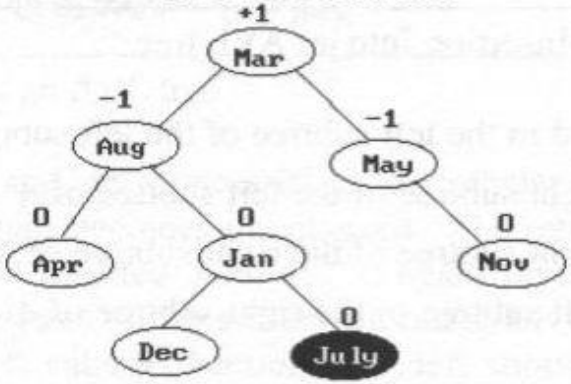
Figure 10.11 (continued): Insertion into an AVL tree



(f) Insert January

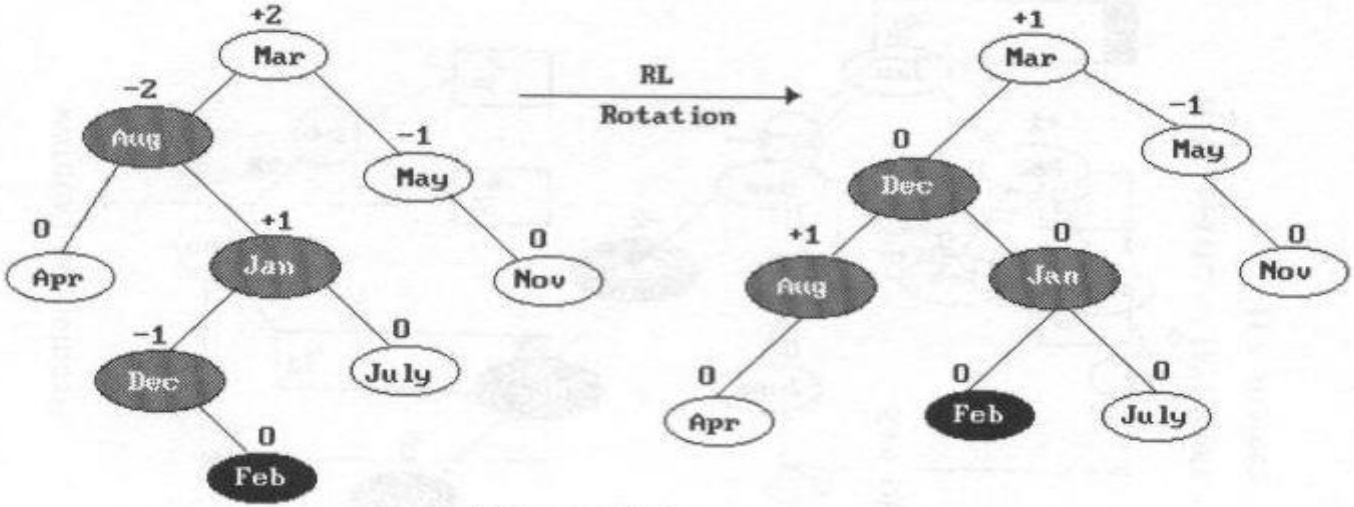


(g) Insert December

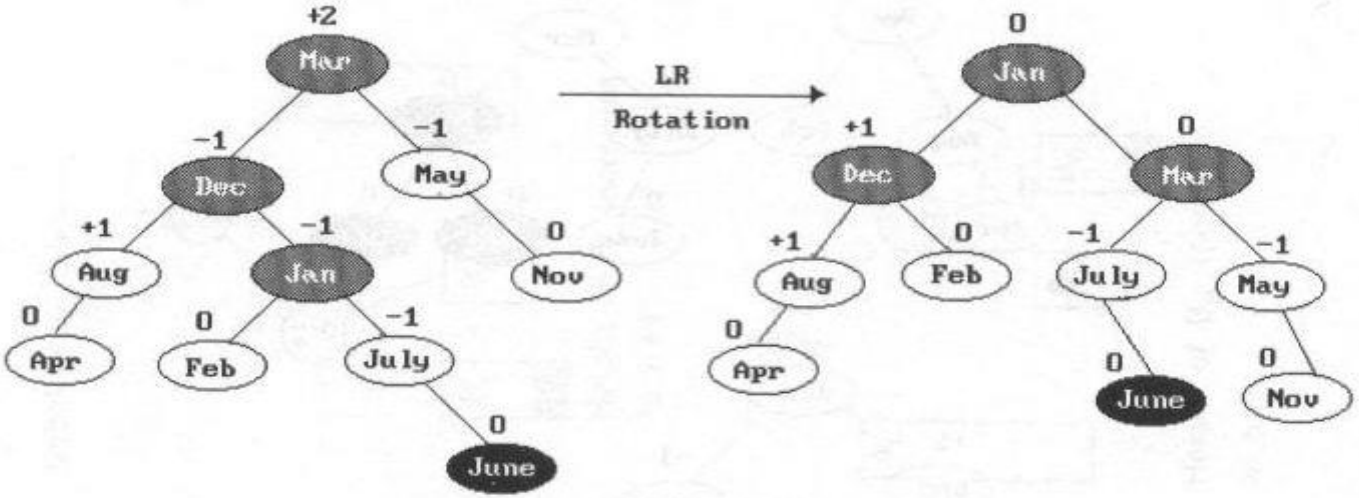


(h) Insert July

Figure 10.11 (continued): Insertion into an AVL tree

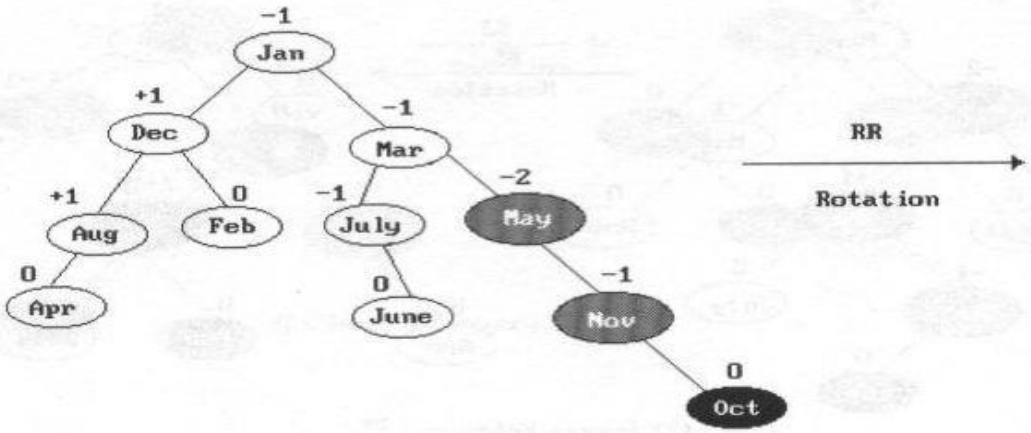


(i) Insert February

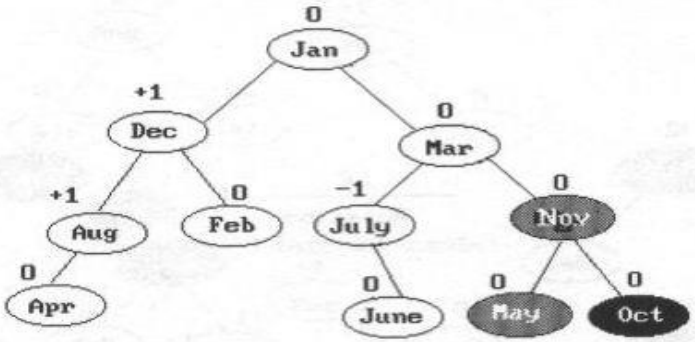


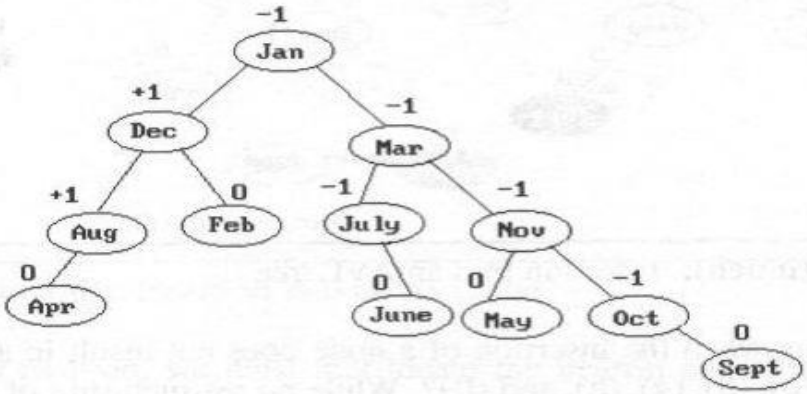
(j) Insert June

Figure 10.11 (continued): Insertion into an AVL tree



(k) Insert October

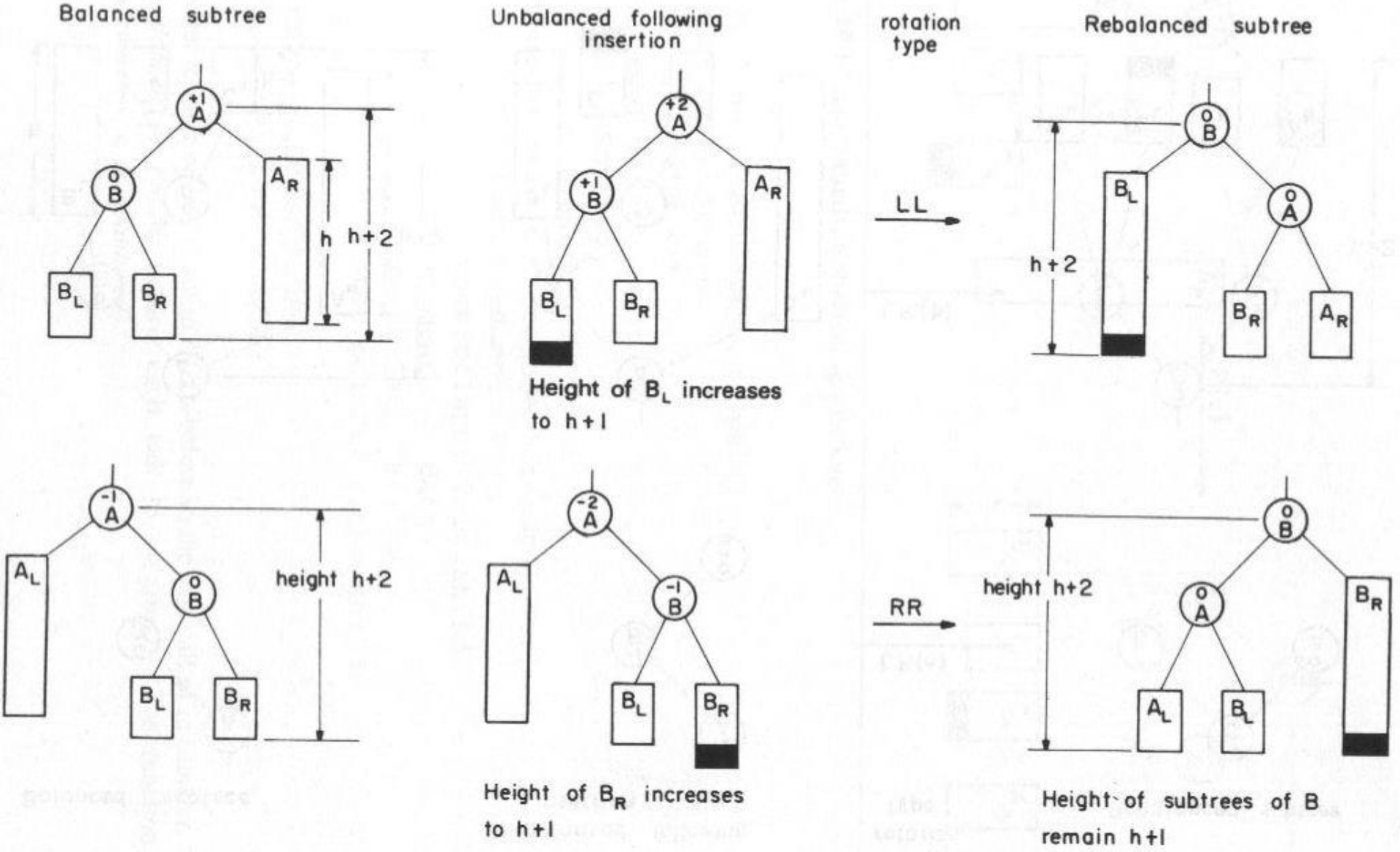


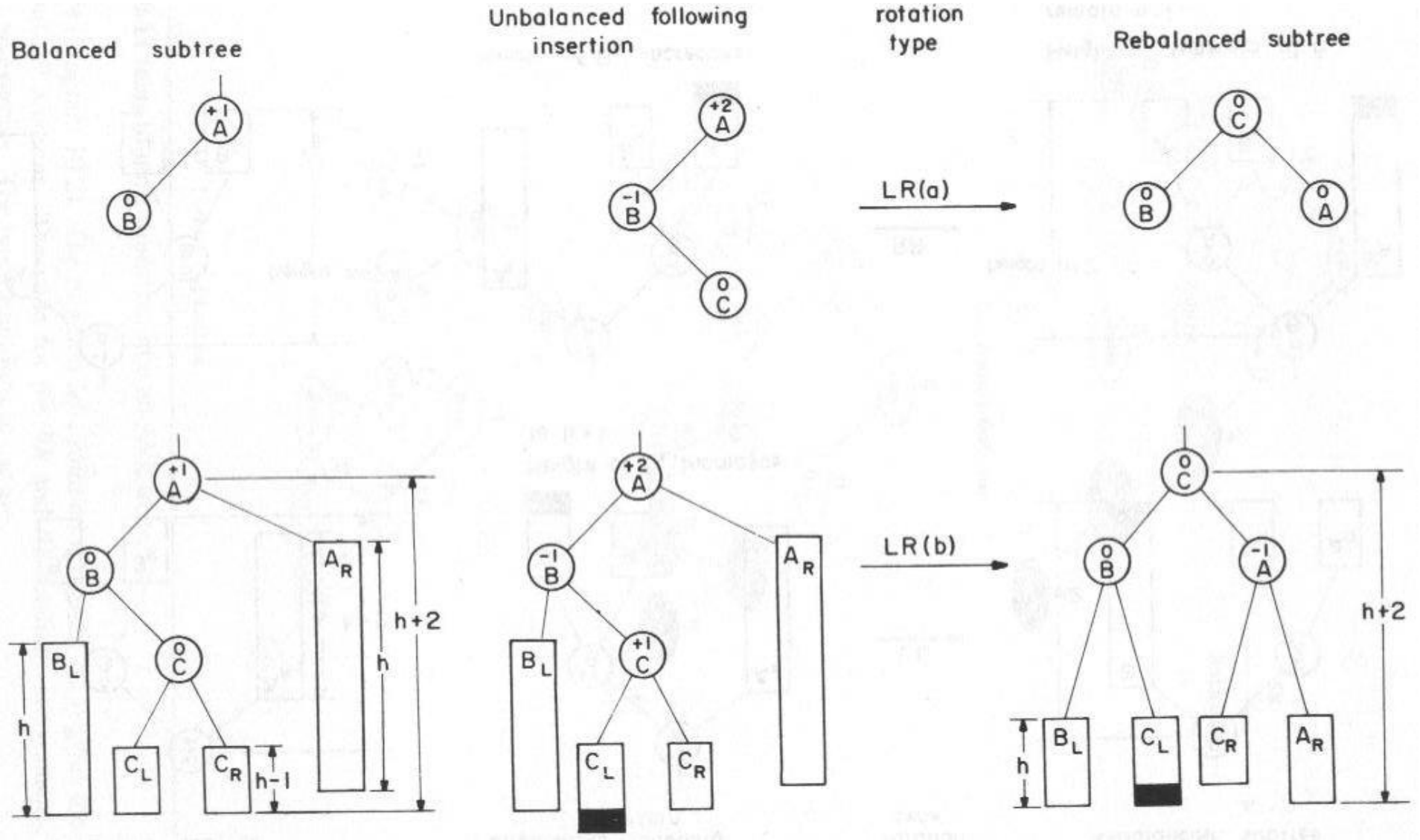


(l) Insert September

Figure 10.11 (continued): Insertion into an AVL tree

– In the preceding example we saw that the addition of a node to a balanced binary search tree could unbalance it.
– We carried out the rebalancing using four different kinds of rotations: *LL*, *RR*, *LR*, and *RL* (Figure 10.11 (e), (c), (f), and (i), respectively).
  • *LL* and *RR* are symmetric as are *LR* and *RL*.
– These rotations are characterized by the nearest ancestor, *A*, of the inserted node, *Y*, whose balance factor becomes ±2. We characterize the rotation type as follows:
  1. *LL*: new node *Y* is inserted in the left subtree of the left subtree of *A*.
  2. *LR*: *Y* is inserted in the right subtree of the left subtree of *A*
  3. *RR*: *Y* is inserted in the right subtree of the right subtree of *A*
  4. *RL*: *Y* is inserted in the left subtree of the right subtree of *A*
  • *(Figure 10.12, p.491 C definition, program 10.2, 10.3)*


Balanced subtree
Unbalanced following insertion
rotation type
Rebalanced subtree
+1 A
0 B
A_R
B_L
B_R
h
h+2
+2 A
+1 B
LL
0 B
0 A
h+2
Height of B_L increases to h+1
-1 A
A_L
0 B
height h+2
-2 A
-1 B
RR
height h+2
Height of B_R increases to h+1
Height of subtrees of B remain h+1


Balanced subtree
Unbalanced following insertion
rotation type
Rebalanced subtree
+1 A
0 B
+2 A
-1 B
0 C
LR(a)
0 C
0 B
0 A
+1 A
0 B
0 C
A_R
B_L
C_L
C_R
h
h+2
h-1
+2 A
-1 B
+1 C
LR(b)
0 C
0 B
-1 A

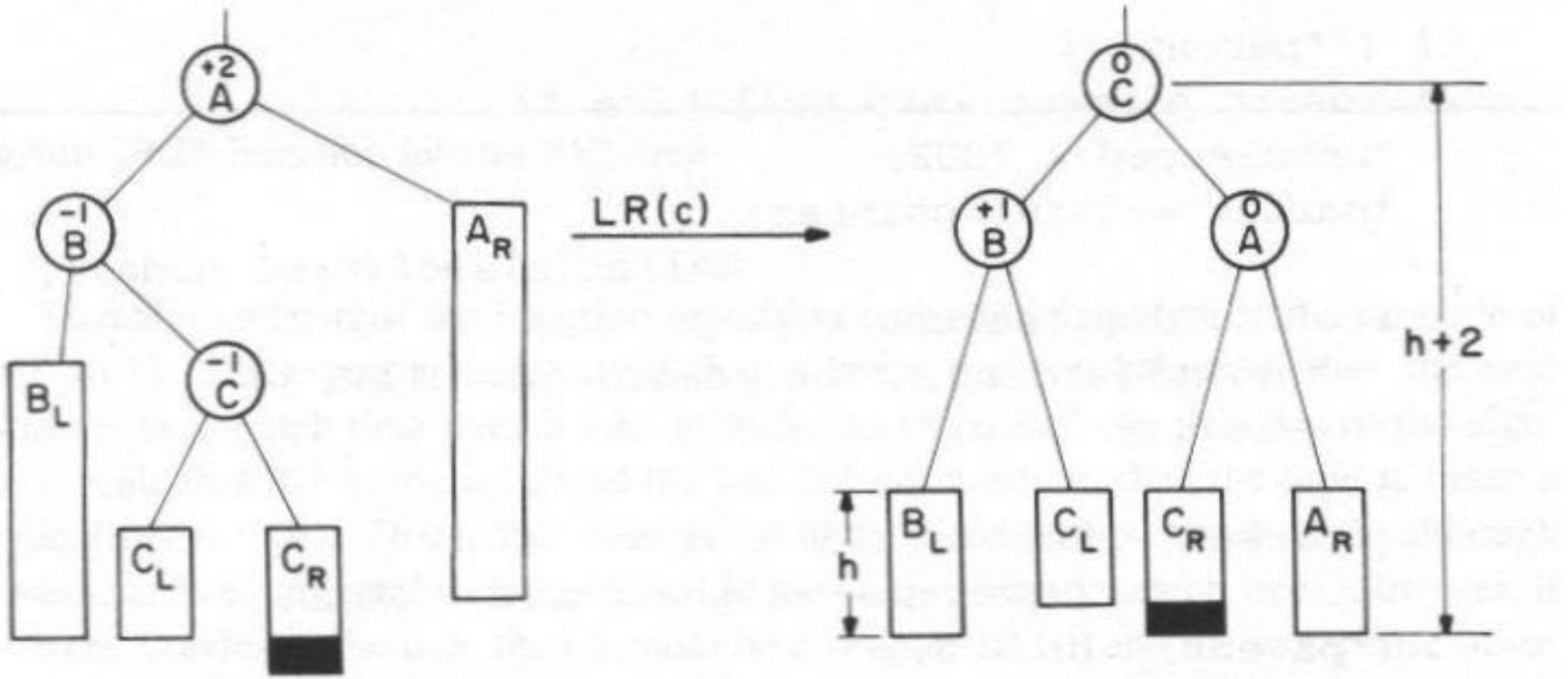


Figure 10.12 (continued): Rebalancing rotations

– **Complexity:**

- In the case of binary search trees, if there were n nodes in the tree, then h (the height of tree) could be be n and the worst case insertion time would be $O(n)$.
- In the case of AVL trees, since h is at most $(\log n)$, the worst case insertion time is $O(\log n)$.

– Figure 10.13 compares the worst case times of certain operations on sorted sequential lists, sorted linked lists, and AVL trees.

•

Operation	Sequential list	Linked list	AVL tree
Search for x	$O(\log n)$	$O(n)$	$O(\log n)$
Search for kth item	$O(1)$	$O(k)$	$O(\log n)$
Delete x	$O(n)$	$O(1)^1$	$O(\log n)$
Delete kth item	$O(n-k)$	$O(k)$	$O(\log n)$
Insert x	$O(n)$	$O(1)^2$	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

1. Doubly linked list and position of x known.
2. If position for insertion is known.

Figure 10.13: Comparison of various structures

10.3 Two-three trees (2-3 trees)

- Definition and properties
  - By considering search tree of degree greater than 2, we can arrive at tree structures for which the insertion and deletion algorithms are simpler than that for AVL trees.
    - These algorithms have O(log*n*) complexity.
  - The tree structure we consider is called a 2-3 tree.
    - This name reflects the fact that each internal node in a 2-3 tree has degree two or three.
    - A degree two node is called a 2-*node* while a degree three node is called a 3-*node*.

Definition:

A 2-3 tree is a search tree that is either empty or satisfies the following properties:

1. Each internal node is either a 2-node or a 3-node. A 2-node has one element while a 3-node has two elements.
2. Let *left_child* and *middle_child* denote the children of a 2-node. Let *data_l* be the element in this node and let *data_l.key* be its key. All elements in the 2-3 subtree with the root *left_child* have key less than *data_l.key* , while all elements in the 2-3 subtree with root *middle_child* have key greater than *data_l.key*.
3. Let *left_child*, *middle_child* and *right_child* denote the children of a 3-node. Let *data_l* and *data_r* be the two elements in this node. Then, *data_l.key* < *data_r.key*; all keys in the 2-3 subtree with root *left_child* are less than *data_l.key*; all keys in the 2-3 subtree with root *middle_child* are less than *data_r.key* and greater than *data_l.key*; and all keys in the 2-3 subtree with root *right_child* are greater than *data_r.key*.
4. All external nodes are at the same level.

– An example 2-3 tree is given in Figure 10.14.

–

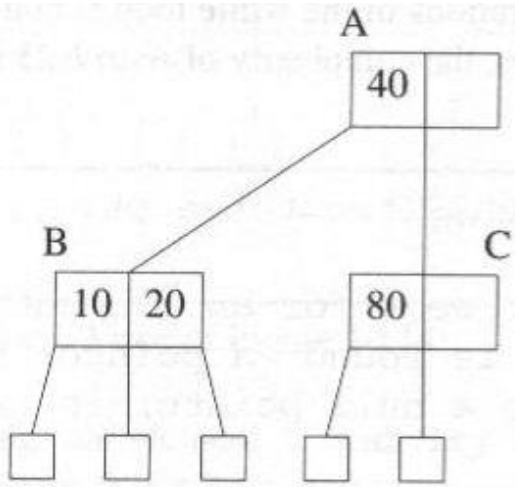


Figure 10.14: An example 2-3 tree

– The number of elements in a 2-3 tree with height h is between 2^h -1 and 3^h -1.

– The height of a 2-3 tree with n elements is between $\lceil \log_3(n+1) \rceil$ and $\lceil \log_2(n+1) \rceil$.

- Searching a 2-3 tree
  - Searching a 2-3 tree, *t*, for a node that contains an element with key *x*.
    - The search function uses a *compare* functions that compare a key, *x*, with the keys in a given node *p*. it returns the value 1, 2, 3, or 4, respectively, depending on whether *x* is less than the first key, between the first and second keys, greater than the second key, or equal to one of the keys in *p*.
    - The complexity of *search23* is O(log*n*).

- Insertion into a 2-3 tree

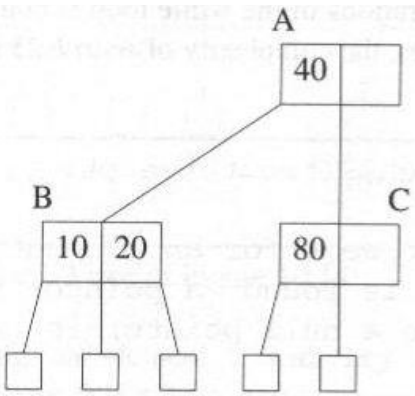


Figure 10.14: An example 2-3 tree

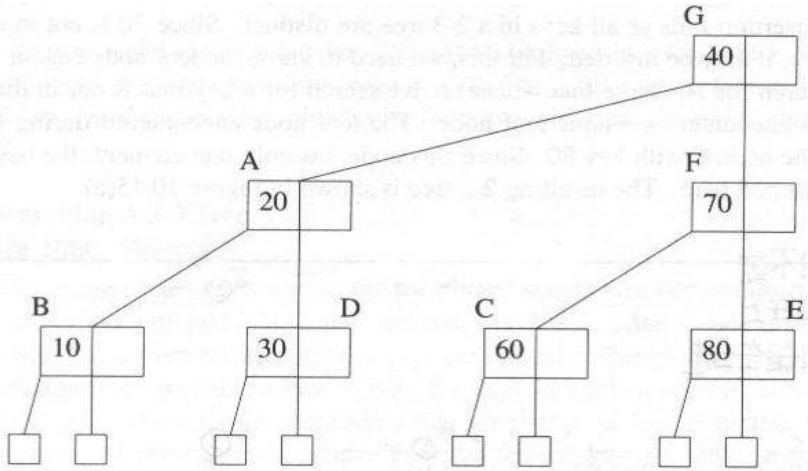
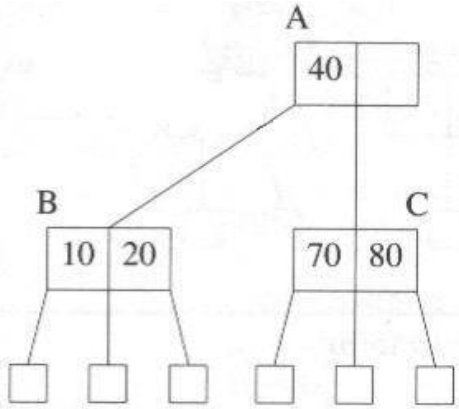
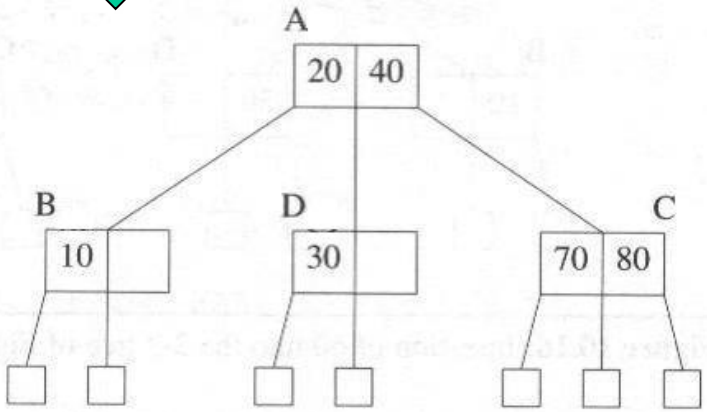


Figure 10.16: Insertion of 60 into the 2-3 tree of Figure 10.15(b)



(a) 70 inserted

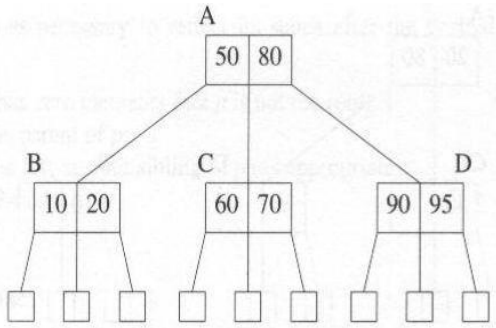


(b) 30 inserted

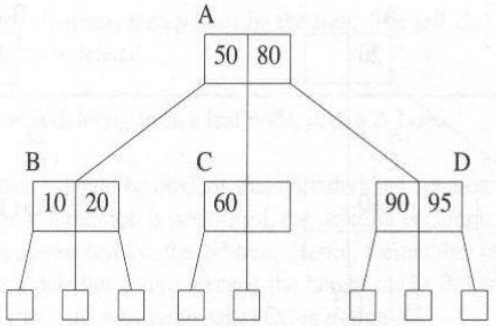
Figure 10.15: Insertion into the 2-3 tree of Figure 10.14

- **Analysis of insertion:**
  - Insertion into a 2-3 tree with n elements takes $O(\log n)$ time.

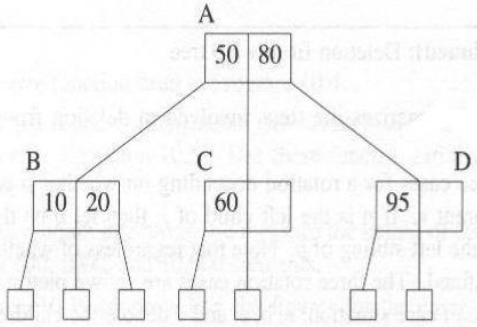
- Deletion from a 2-3 tree
  - In case we are deleting an element that is not in a leaf node, then we transform this into a deletion from a leaf node by replacing the deleted element by a suitable element that is in a leaf.
  - In a general situation we may use either the element with largest key in the subtree on the left or the element with smallest key in the subtree on the right of the element being deleted.
  - Henceforth, we consider only the case of deletion from a leaf node.
    - (figure 10.17)
    - (program 10.6)



(a) Initial 2-3 tree

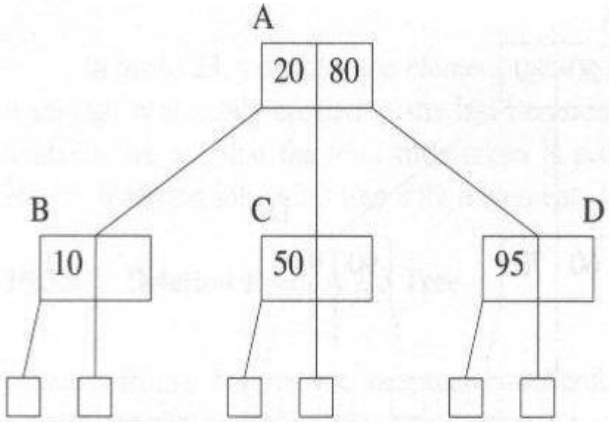


(b) 70 deleted

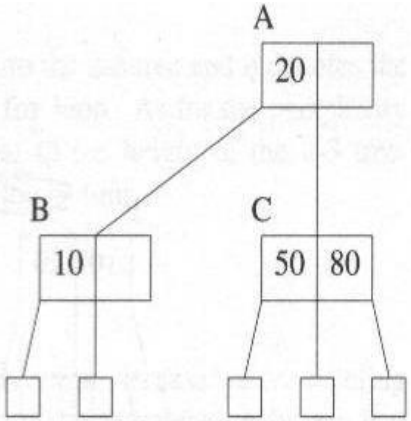


(c) 90 deleted

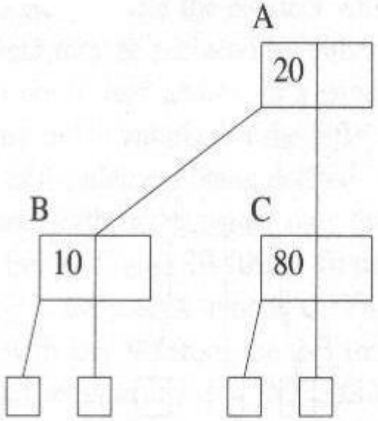
Figure 10.17: Deletion from a 2-3 tree



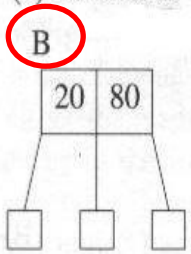
(d) 60 deleted



(e) 95 deleted



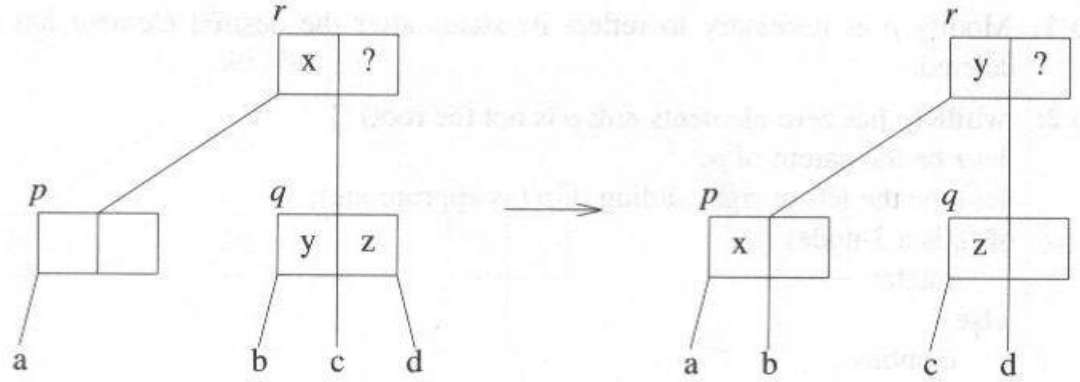
(f) 50 deleted



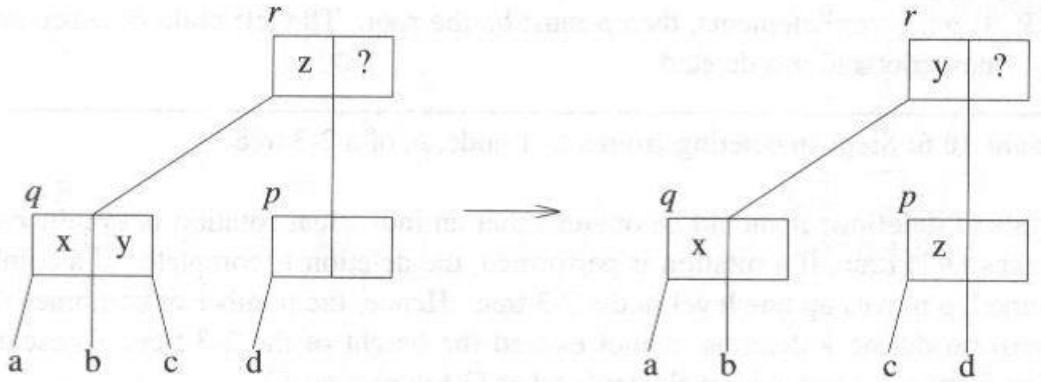
(g) 10 deleted

Figure 10.17 (continued): Deletion from a 2-3 tree

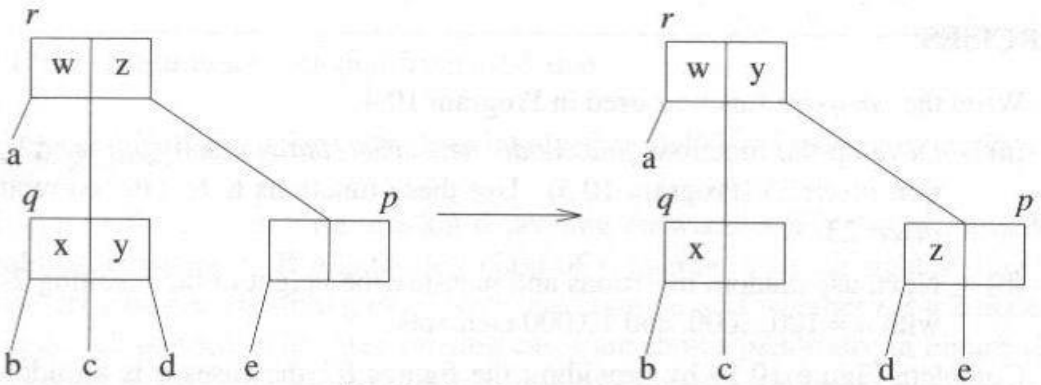
– There are three cases for a rotation depending on whether *p* is the left, middle, or right child of it parent *r*.
  - If *p* is the left child of *r*, then let *q* be the right sibling of *p*. Otherwise, let *q* be the left sibling of *p*.
  - Note that regardless of whether *r* is a 2-node or a 3-node, *q* is well defined.
  - The three rotation cases are shown pictorially in Figure 10.18. A "?" denotes a don't care situation. a, b, c, and d denote the children of nodes.
  - (Figure 10.18)



(a) p is the left child of r



(b) p is the middle child of r



(c) p is the right child of r

Figure 10.18: The three cases for rotation in a 2-3 tree

– Figure 10.19 shows the two cases for a combine when *p* is the left child of *r*.

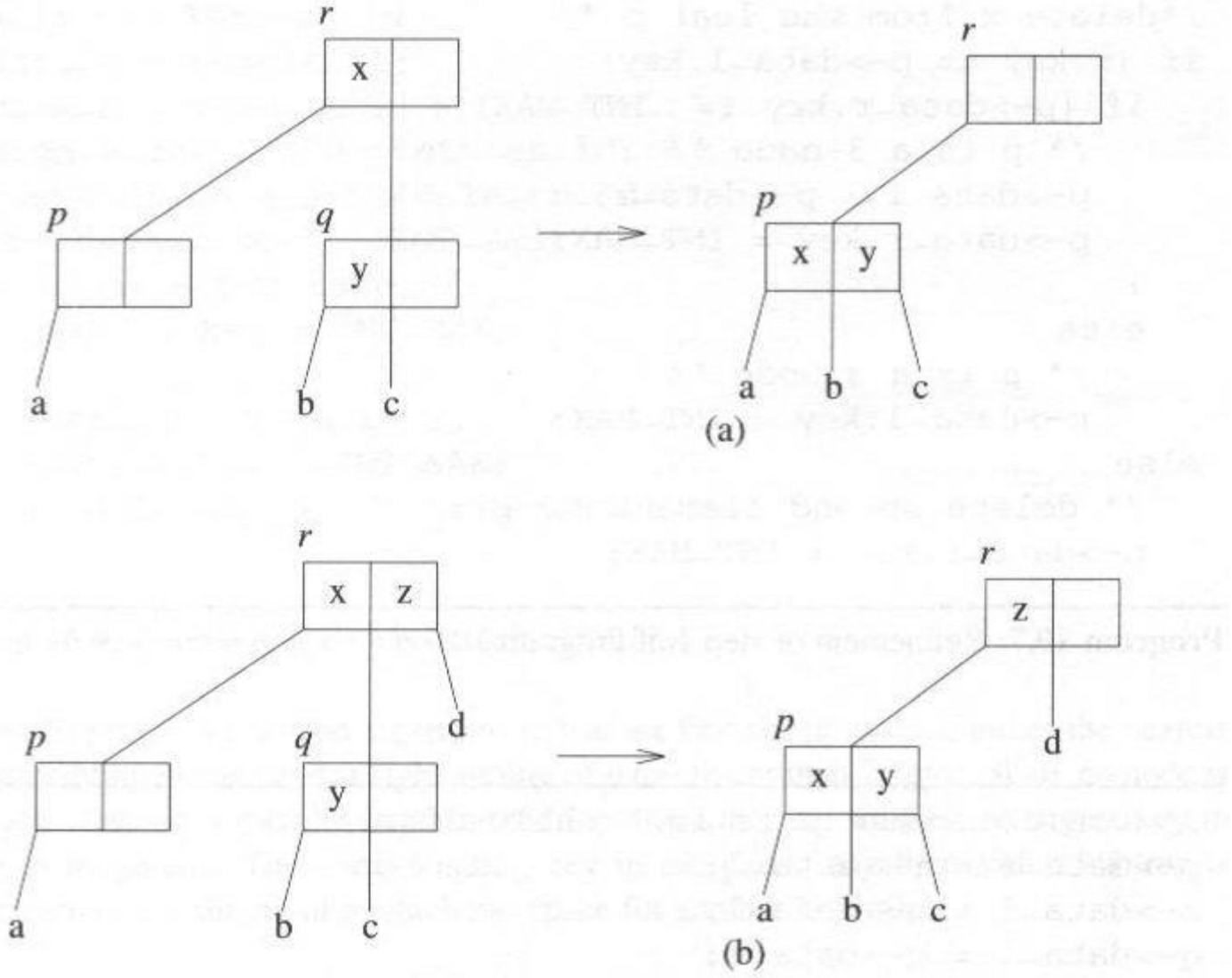


Figure 10.19: Combining in a 2-3 tree when *p* is a the left child of *r*

- **Analysis of deletion:**
  - Deletion from a 2-3 tree with n elements takes O(log n) time.

10.4 Two-three-four trees (2-3-4 trees)

- Definition and properties
  - A 2-3-4 tree extends a 2-3 tree so that 4-nodes are also permitted (4-nodes may have up to four children).

Definition:

A *2-3-4 tree* is a search tree that is either empty or satisfies the following properties:

1. Each internal node is either a 2-, 3- or 4-node. A 2-node has one element, a 3-node has two elements, and a 4-node has three elements.
2. Let *left_child* and *left_mid_child* denote the children of a 2-node. Let *data_l* be the element in this node and let *data_l.key* be its key. All elements in the 2-3-4 subtree with the root *left_child* have key less than *data_l.key* , while all elements in the 2-3-4 subtree with root *left_mid_child* have key greater than *data_l.key*.
3. Let *left_child*, *left_mid_child* and *right_mid_child* denote the children of a 3-node. Let *data_l* and *data_m* be the two elements in this node. Then, *data_l.key* < *data_m.key*; all keys in the 2-3-4 subtree with root *left_child* are less than *data_l.key*; all keys in the 2-3-4 subtree with root *left_mid_child* are less than *data_m.key* and greater than *data_l.key*; and all keys in the 2-3-4 subtree with root *right_mid_child* are greater than *data_m.key*.

4. Let *left_child*, *left_mid_child*, *right_mid_child*, and *right_child* denote the children of a 4-node. Let *data_l*, *data_m*, and *data_r* be the three elements in this node. Then, *data_l.key* < *data_m.key* < *data_r.key*; all keys in the 2-3-4 subtree with root *left_child* are less than *data_l.key*; all keys in the 2-3-4 subtree with root *left_mid_child* are less than *data_m.key* and greater than *data_l.key*; and all keys in the 2-3-4 subtree with root *right_mid_child* are greater than *data_m.key* but less than *data_r.key*; and all keys in the 2-3-4 subtree with root *right_child* are greater than *data_r.key*.
5. All external nodes are at the same level.

– An example 2-3-4 tree is given in Figure 10.20.

–

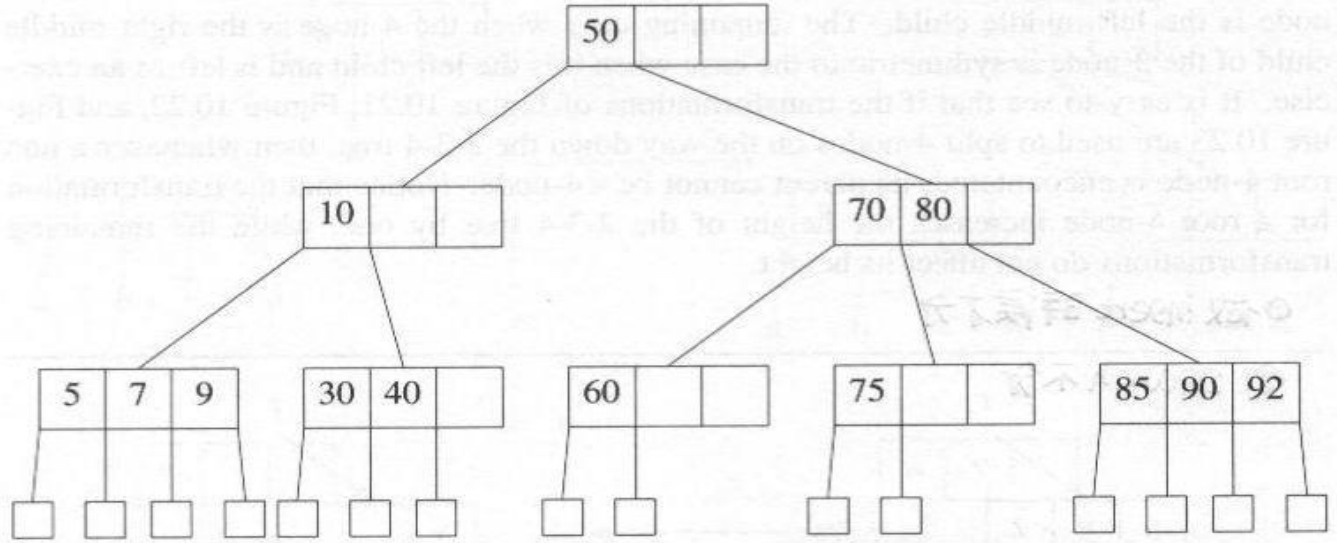


Figure 10.20: An example 2-3-4 tree

– The number of elements in a 2-3-4 tree with height h is between $2^h - 1$ and $4^h - 1$.

– The height of a 2-3 tree with n elements is between $\lceil \log_4(n+1) \rceil$ and $\lceil \log_2(n+1) \rceil$.

- An advantage 2-3-4 trees have over 2-3 trees is that we may insert an element into, or delete an element from, a 2-3-4 tree by a single root to leaf pass.
  - The same operations on a 2-3 tree require a forward root to a leaf pass followed by a backward leaf to root pass.
  - As a result, the corresponding 2-3-4 tree algorithms are similar.
- More interestingly, we can efficiently represent a 2-3-4 tree as a binary tree called a red-black tree.
  - As we shall see in the next section, red-black trees utilize space more efficiently than either 2-3 or 2-3-4 trees.
  - In this section, we shall see how we can insert elements into, and delete them from, a 2-3-4 tree by making a single top-down root to leaf pass over the tree.

- Insertion into a 2-3-4 tree
  - If the leaf node into which the element is to be inserted is a 4-node, then this node splits and a backward leaf to root pass is initiated.
  - This backward pass terminates when either a 2- or 3-node is encountered or when the root is split.
  - To avoid the backward leaf to root pass, we split 4-nodes on the way down the tree.
  - As the result, the leaf node into which the insertion is to be made is guaranteed to be a 2- or 3-node. The element to be inserted may be added to this node without any further node splitting.

- There are essentially three different situations to consider for a 4-node:
  - it is the root of the 2-3-4 tree (Figure 10.21)
  - its parent is a 2-node (Figure 10.22)
  - its parent is a 3-node (Figure 10.23 and Figure 10.24)
- It is easy to see that if the transformations of Figure 10.21, Figure 10.22, and Figure 10.23 are used to split 4-nodes on the way down the 2-3-4 tree, then whenever a non root 4-node is encountered, its parent cannot be a 4-node.
- Notice that the transformation for a root 4-node increase the height of the 2-3-4 tree by one, while the remaining transformations do not affect its height.
- (Figure 10.21-24)

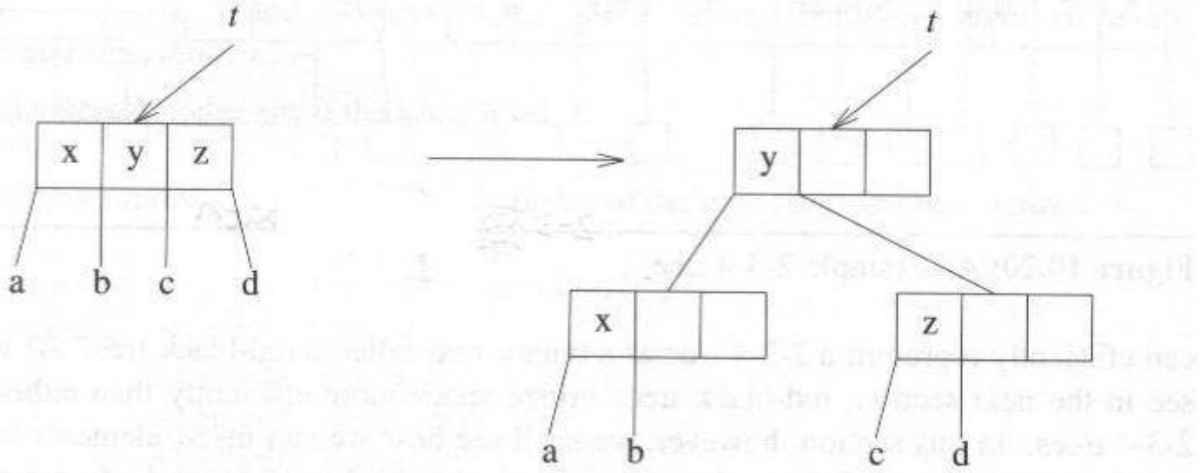


Figure 10.21: Transformation when the 4-node is the root

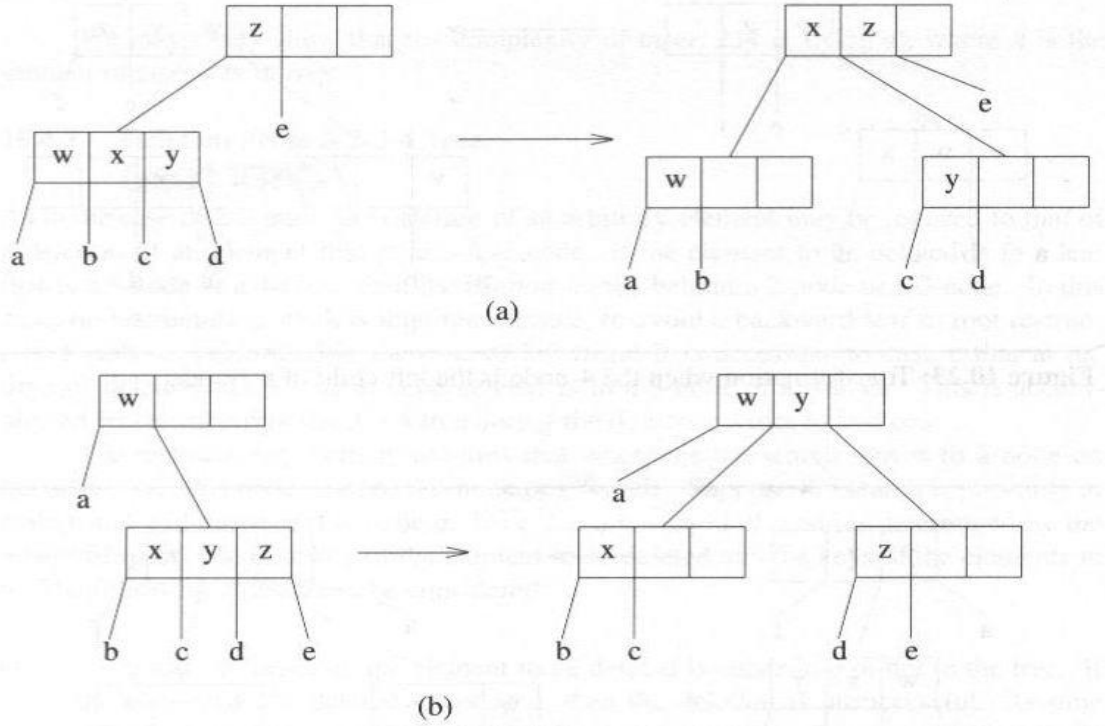


Figure 10.22: Transformation when the 4-node is the child of a 2-node

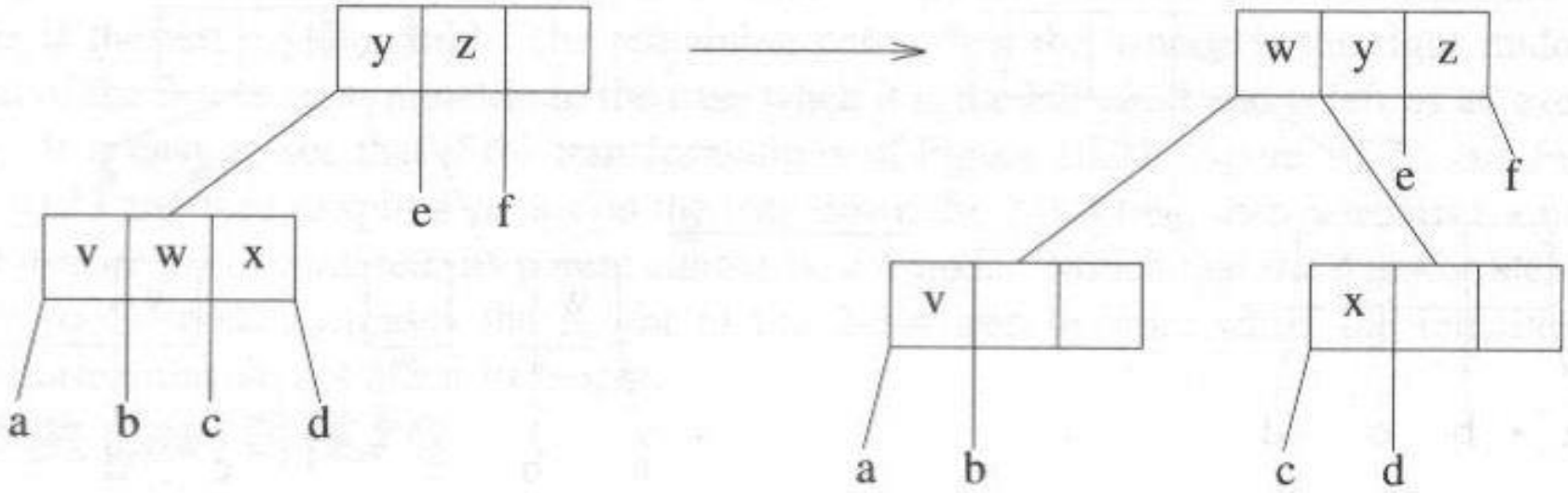


Figure 10.23: Transformation when the 4-node is the left child of a 3-node

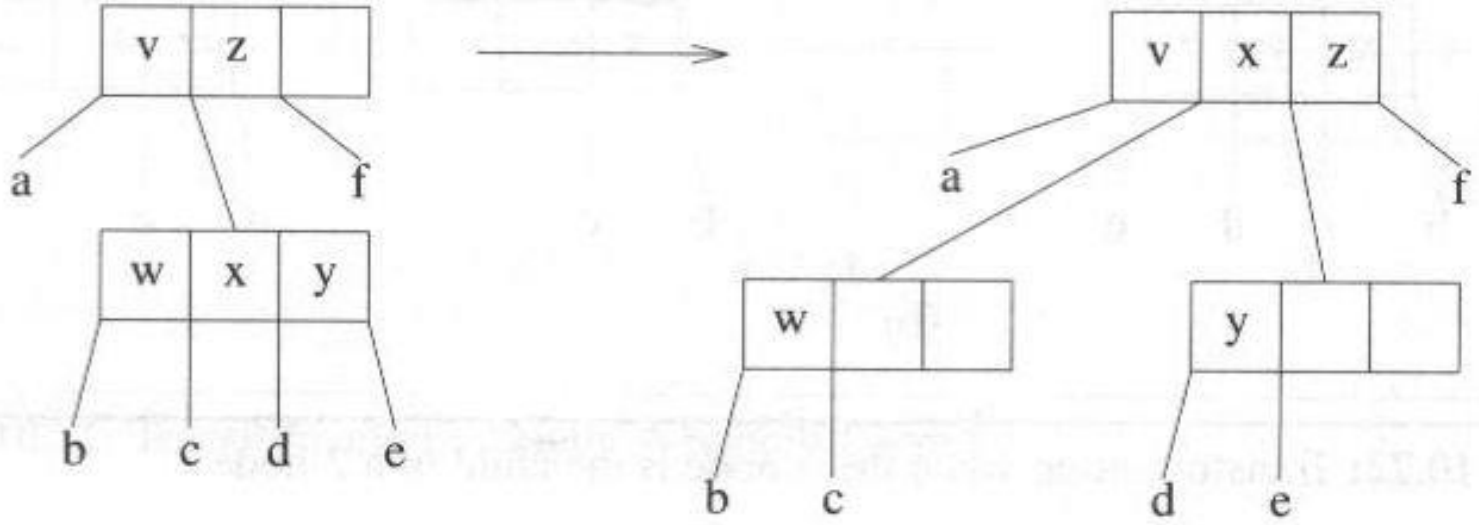


Figure 10.24: Transformation when the 4-node is the left middle child of a 3-node

- Deletion from a 2-3-4 tree
  - As in the case of 2-3 trees, the deletion of an arbitrary element may be reduced to that of a deletion of an element that is in a leaf node.
  - If the element to be deleted is in a leaf that is a 3-node or a 4-node, then its deletion leaves behind a 2-node or a 3-node. In this case, no restructuring work is required.
  - Hence, to avoid a backward leaf to root restructuring path (as performed in the case of 2-3 trees) it is necessary to ensure that at the time of deletion, the element to be deleted is in a 3-node or a 4-node.
    - This is accomplished by restructuring the 2-3-4 tree during the downward root to leaf pass.

– The restructuring strategy requires that when ever the search moves to a node on the next level, this node must be a 3-node or a 4-node.
  - Suppose the search is presently at node *p* and will move next to node *q*.
  - Note that *q* is a child of *p* and is determined by the relationship between the key of the element to be deleted and the keys of the elements in *p*.
– The following cases are to be considered:

  p is a leaf. In this case, the element to be deleted is either in *p* or not in the tree. If the element to be deleted is not in *p*, then the deletion is unsuccessful. Assume this is not the case. By the nature of the restructuring process, *p* can be a 2-node only if it is also the root. The deletion results in an empty tree.

q is not a 2-node. In this case, the search moves to *q* and no restructuring is need.

q is a 2-node and its nearest sibling *r* is also a 2 node (if *q* is the left child of *p*, then its nearest sibling is the left middle child of *p*; otherwise, the nearest sibling is its left sibling). Now, if *p* is a 2-node, it must be the root and we perform the transformation of Figure 10.21 in reverse. That is, *p*, *q*, and *r*, are combined to form a 4-node and the height of the tree decreases by 1. If *p* is a 3-node or a 4-node, then we perform, in reverse, the 4-node splitting transformation for the corresponding case (Figures 10.22 through 10.24).

q is a 2-node and its nearest sibling *r* is a 3-node. In this case, we perform the transformation of Figure 10.25. This figure only show the transformations for the case when *q* is the left child of a 3-node *p*. The case when *q* is the left middle child , right middle child, or right child and when *p* is a 2-node (In this case *p* is the root) or a 4-node are similar.

q is a 2-node and it nearest sibling *r* is a 4-node. This is similar to the case when *r* is a 3-node.

- (Figure 10.21, 22-24, 25)

- root

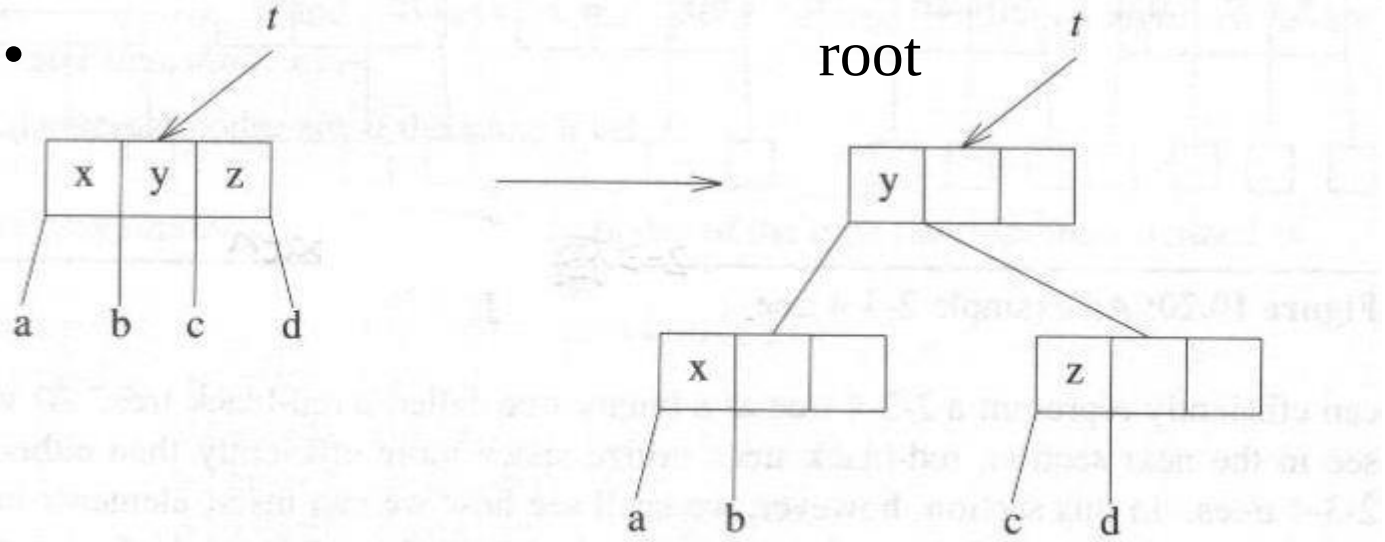


Figure 10.21: Transformation when the 4-node is the root

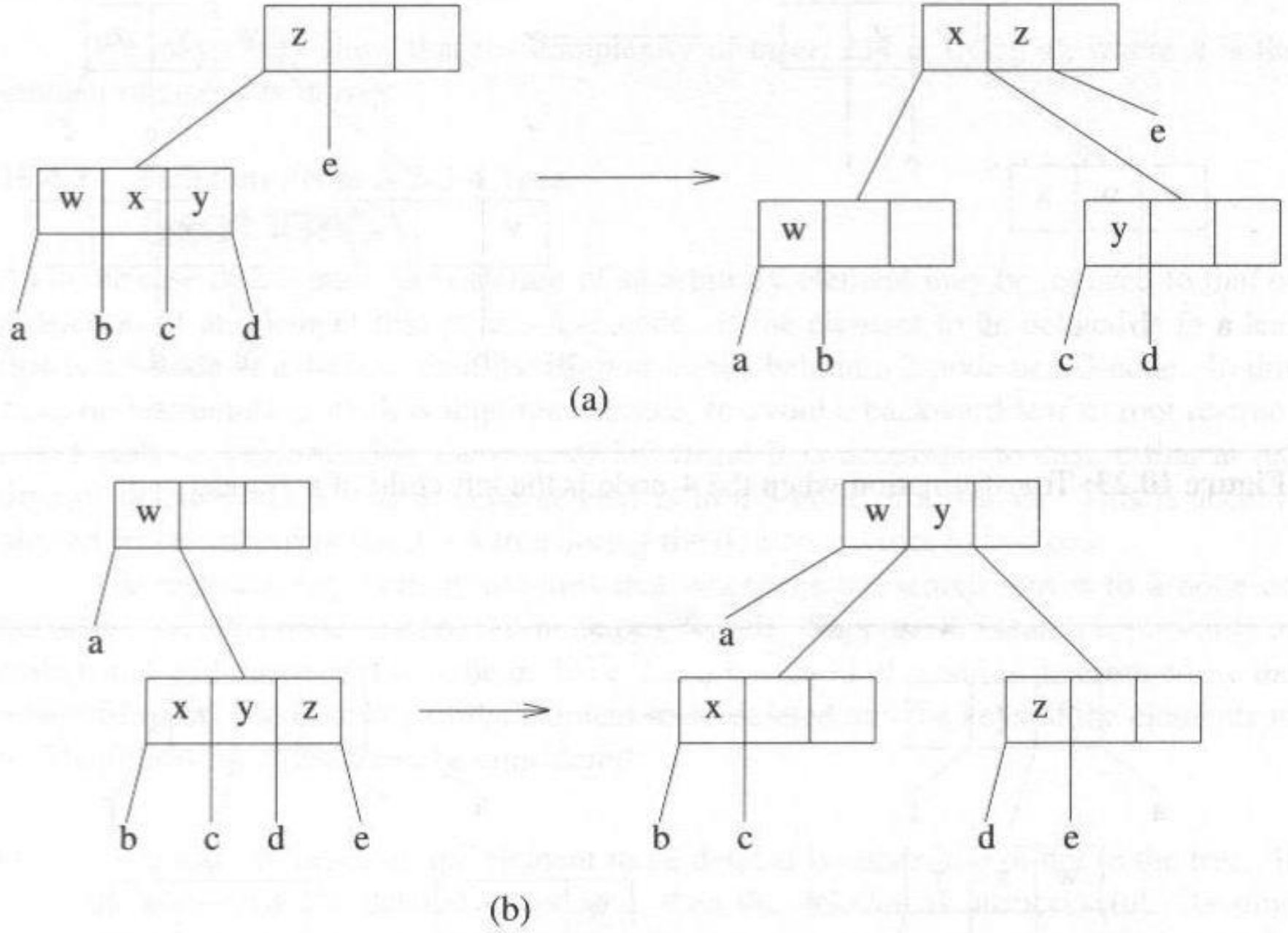


Figure 10.22: Transformation when the 4-node is the child of a 2-node

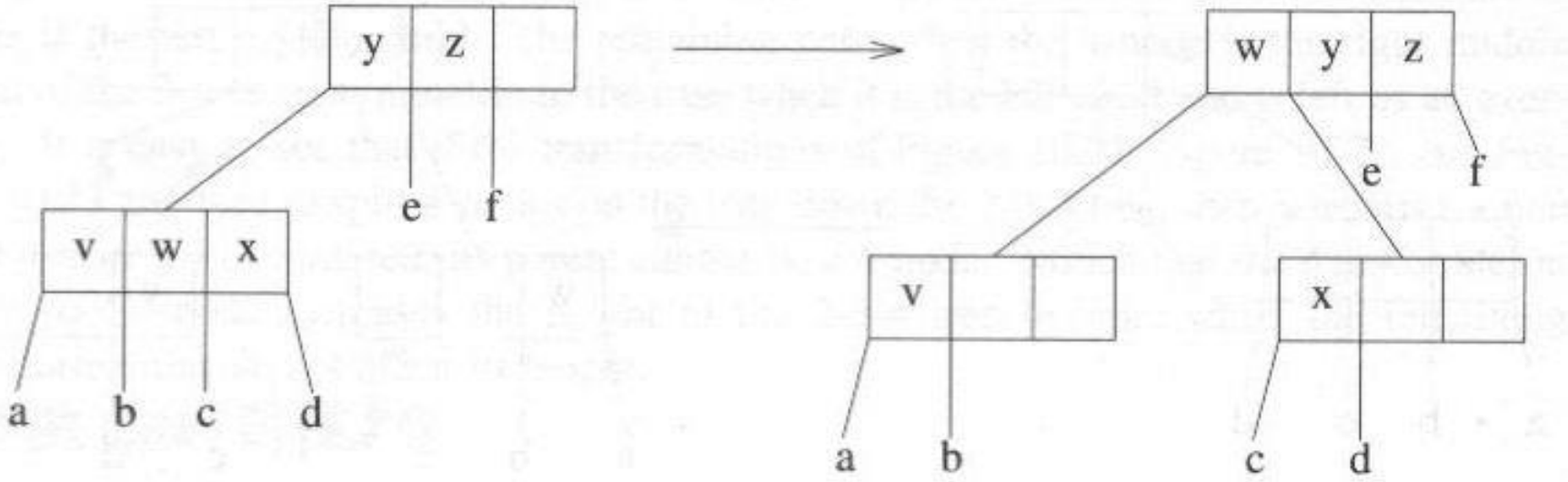


Figure 10.23: Transformation when the 4-node is the left child of a 3-node

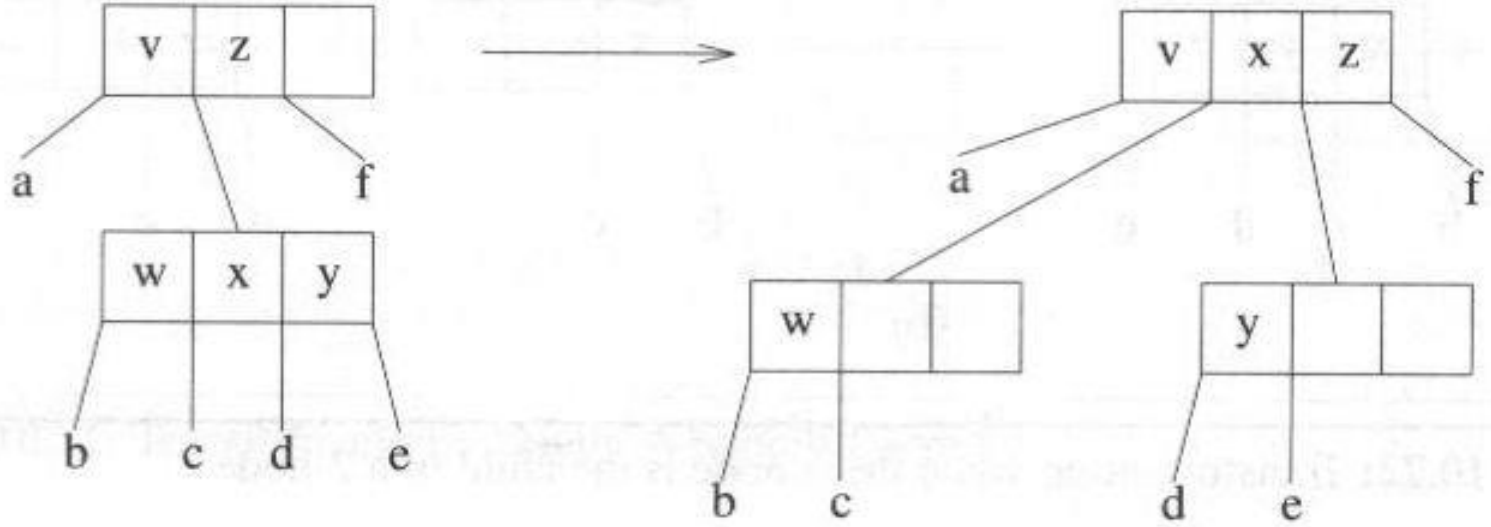
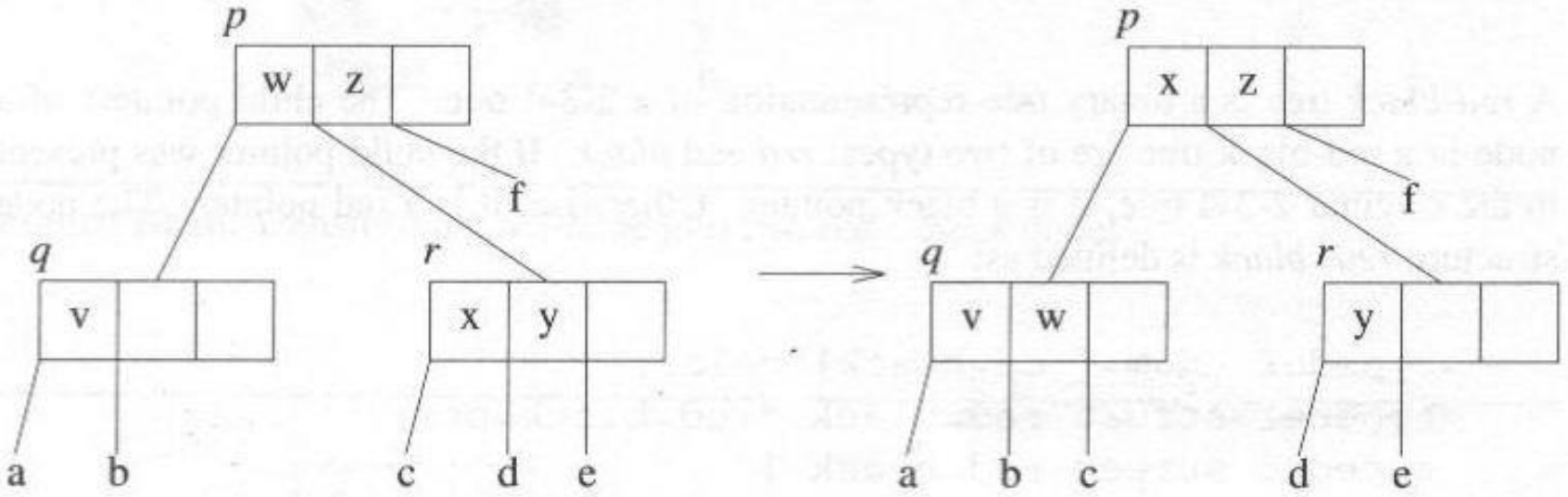
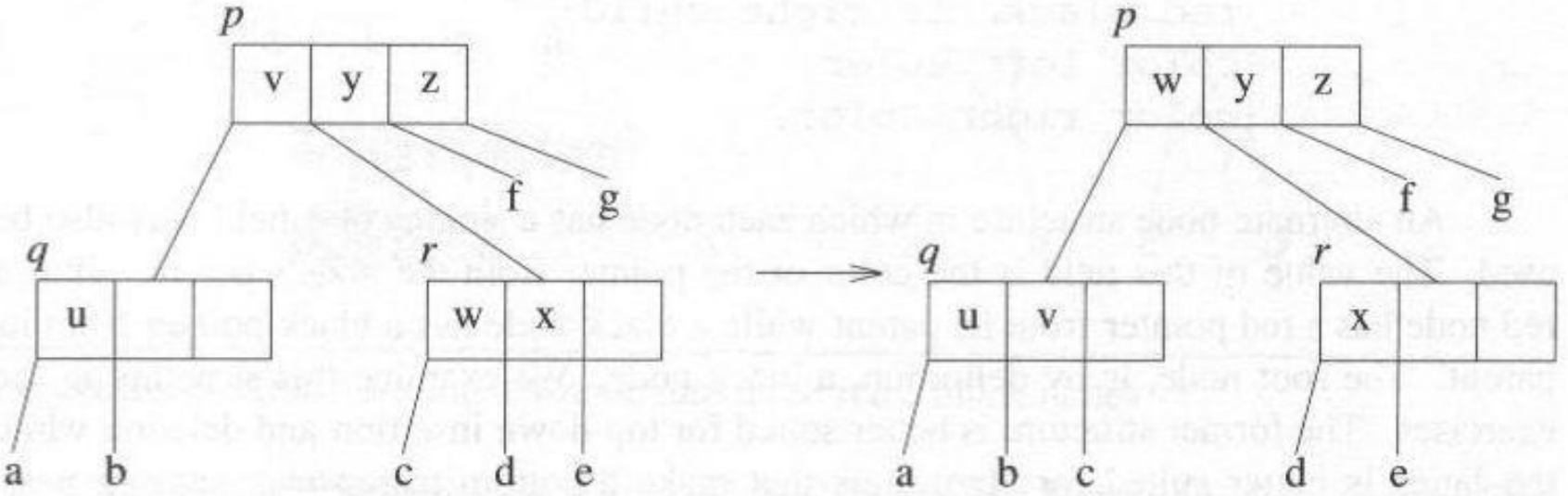


Figure 10.24: Transformation when the 4-node is the left middle child of a 3-node



(a) q is the left child of a 3-node



(b) q is the left child of a 4-node

Figure 10.25: Deletion transformation when the nearest sibling is a 3-node

10.5 Red-black Trees

- Definition and properties
  - A *red-black* tree is a binary tree representation of a 2-3-4 tree.
    - The child pointers of a node in a red-black tree are of two types: *red* and *black*.
    - If the child pointer was present in the original 2-3-4 tree, it is a black pointer. Otherwise, it is a red pointer.

- An alternate node structure in which each node has a single color field may also be used.
  - The value of this field is the color of the pointer from the node's parent.
    - Thus a red node has a red pointer from its parent while a black node has a black pointer from its parent.
  - The root node, is by definition, a black node.
- The former structure is better suited for top down insertion and deletion while the latter is better suited for algorithms that make a bottom to top restructuring pass.
- When drawing a red-black tree, we shall use a solid line to represent a black pointer and a broken one to represent a red pointer.

– We transform a 2-3-4 tree using nodes of type two34 into red-black trees as follows:

  We represent a 2-node, *p*, by a *red_black* node, *q*, with both its color fields black and *data* = *data_l*; *q->left_child* = *p-> left_child*, and *q->right_child* = *p->left_mid_child*.

  A 3-node *p* is represented by two *red_black* nodes connected by a red pointer. There are two ways in which this may be done (see Figure 10.26, color fields are not shown).

  A 4-node is represented by a three *red_black* nodes one of which is connected to the remaining two by red pointers (see Figure 10.27).

– The red-black tree representation of the 2-3-4 tree of Figure 10.20 is given in Figure 10.28.

  • (Figure 10.26,27,20,28)

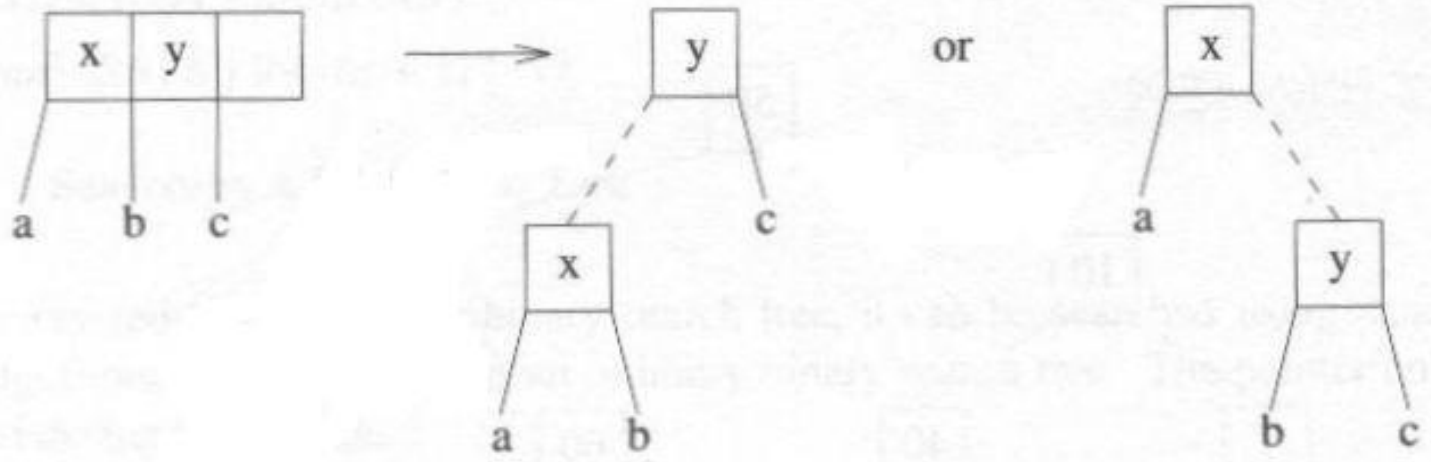


Figure 10.26: Transforming a 3-node into two *red_black* nodes

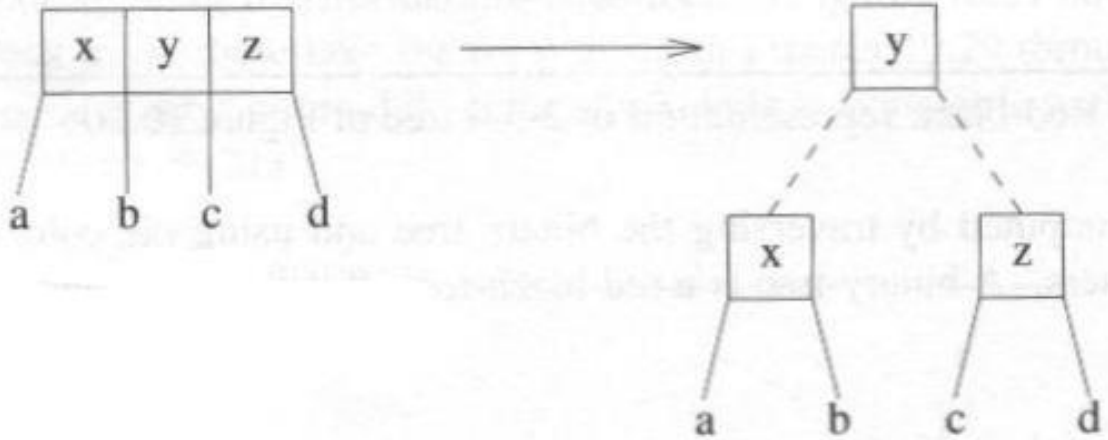


Figure 10.27: Transforming a 4-node into three *red_black* nodes

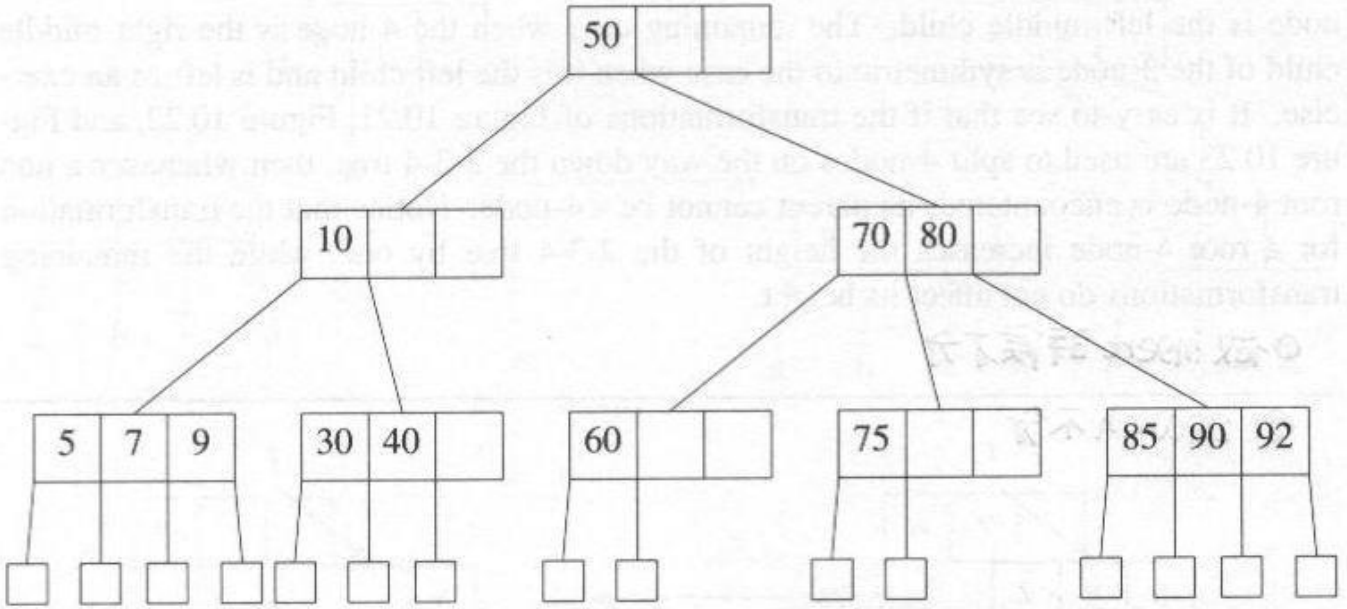


Figure 10.20: An example 2-3-4 tree

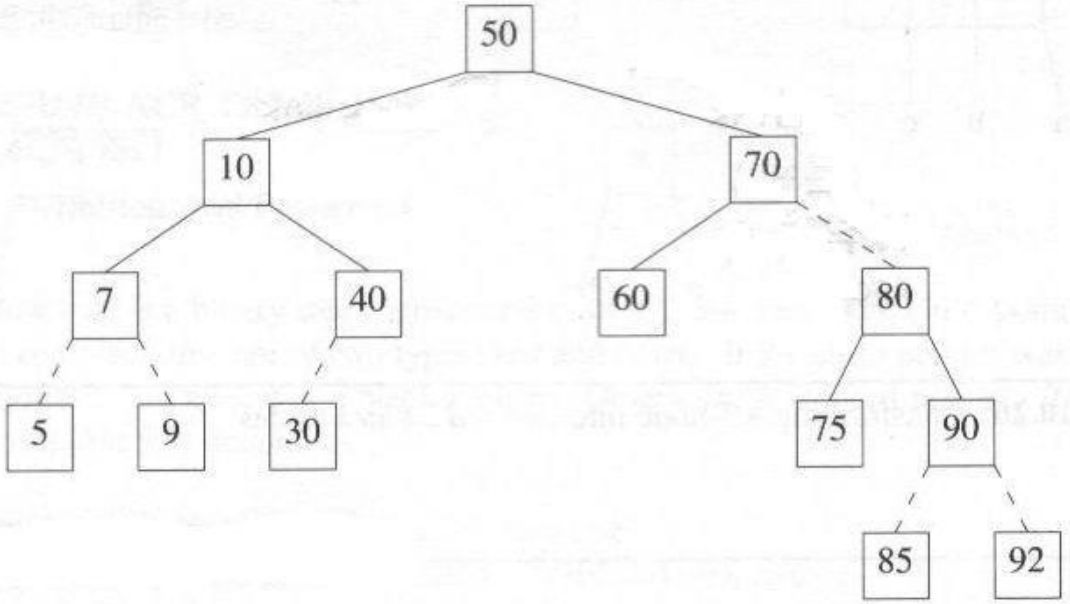


Figure 10.28: Red-black representation of 2-3-4 tree of Figure 10.20

– One may verify that a red-black tree satisfies the following properties:

(P1) It is a binary search tree.

(P2) Every root to external node path has the same number of black links (this follows from the fact that all external nodes of the original 2-3-4 tree are on the same level and black pointers represent original pointers).

(P3) No root to external node path has two or more consecutive red pointers (this follows from the nature of the transformation of Figure 10.26 and Figure 10.27).

– An alternated definition of red-black trees is possible.

- In this, we associate a rank (指原來2-3-4 tree 的level) with each node x in the tree. This value is not explicitly stored in each node.
- If the rank of the root (also called rank of the tree) is known, then the rank of every other node can be computed by traversing the binary tree and using the color information of the nodes/pointers.
- A binary tree is a red-black tree iff it satisfies the following properties:

  (Q1) It is a binary search tree.

  (Q2) The rank of each external node is 0.

  (Q3) Every internal node that is the parent of an external node has rank 1.

  (Q4) For every node x that has a parent $p(x)$, $rank(x) \leq rank(p(x)) \leq rank(x) + 1$.

  (Q5) For every node x that has a grandparent $gp(x)$, $rank(x) < rank(gp(x))$.

- Intuitively, each node x of a 2-3-4 tree T is represented by a collection of nodes in its corresponding red-black tree.
  - All nodes in this collection have a rank equal to $height(T) - level(x) + 1$.
  - So, each time there is a rank change in a path from the root of red-black tree, there is a level change in the corresponding 2-3-4 tree.
  - Black pointers go from a node of a certain rank to one whose rank is one less while red pointers connect two nodes of the same rank.

– **Lemma 10.1**

Every red-black tree RB with n (internal) nodes satisfies the following:

(1) $height\ (RB) \leq 2\lceil \log_2(n+1) \rceil$

(2) $height\ (RB) \leq 2 rank\ (RB)$

(3) $rank\ (RB) \leq \lceil \log_2(n+1) \rceil$

- Searching a red-black tree
  - Since every red-black tree is a binary search tree, it can be searched using exactly the same algorithm as used to search an ordinary binary search tree. The pointer colors are not used during this search.

- Top down insertion
  - An insertion can be carried out in one of the two ways: top down and bottom up.
    - In a top down insertion a single root to leaf pass is made over the red-black tree.
    - A bottom up insertion makes both a root to leaf and a leaf to root pass.
  - To make a top down insertion, we use the 4-node splitting transformations described in Figures 10.21 through 10.24.
    - In terms of red-black trees, these take the form given in Figure 10.29 through 10.32.
    - The case when 4-node is the right middle child of a 3-node is symmetric to the case when it is the left child (Figure 10.31). (Figure 10.21-24, 29-32)

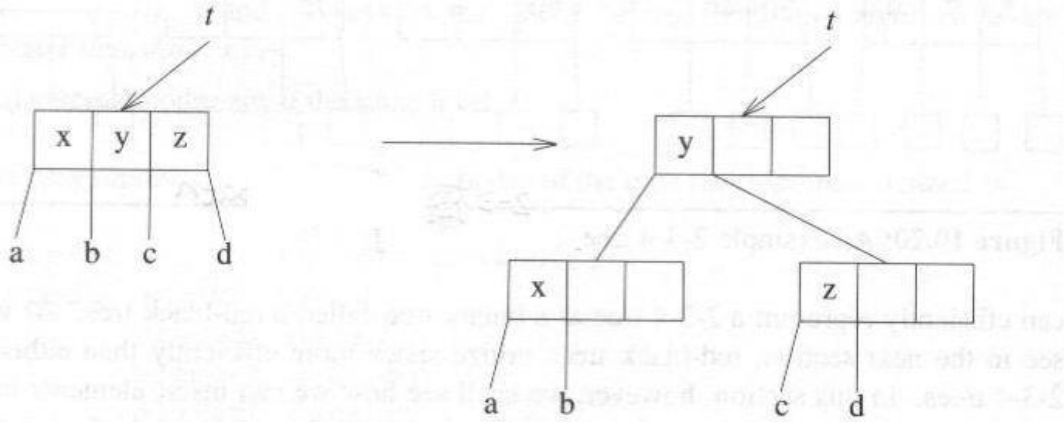


Figure 10.21: Transformation when the 4-node is the root

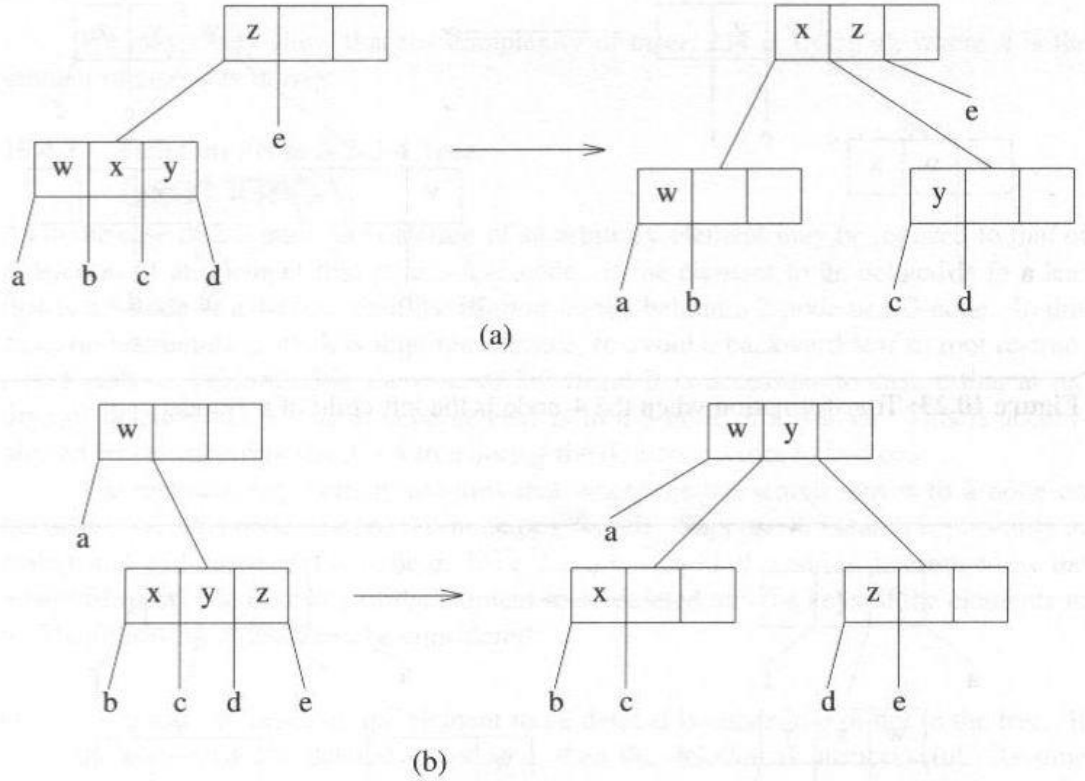


Figure 10.22: Transformation when the 4-node is the child of a 2-node

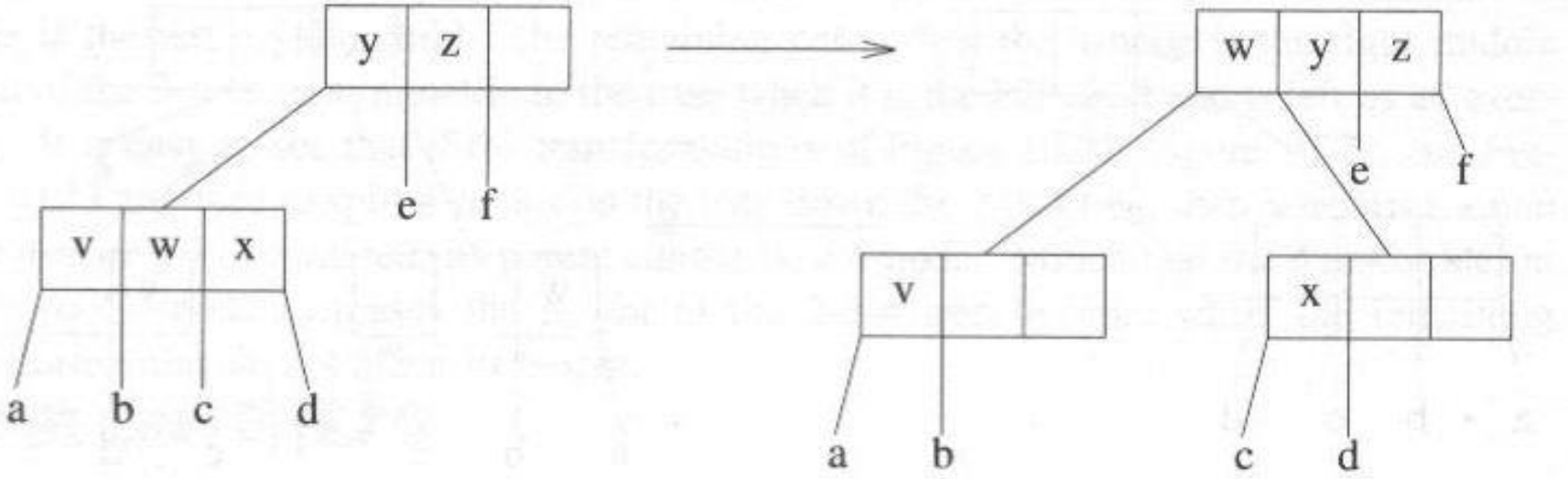


Figure 10.23: Transformation when the 4-node is the left child of a 3-node

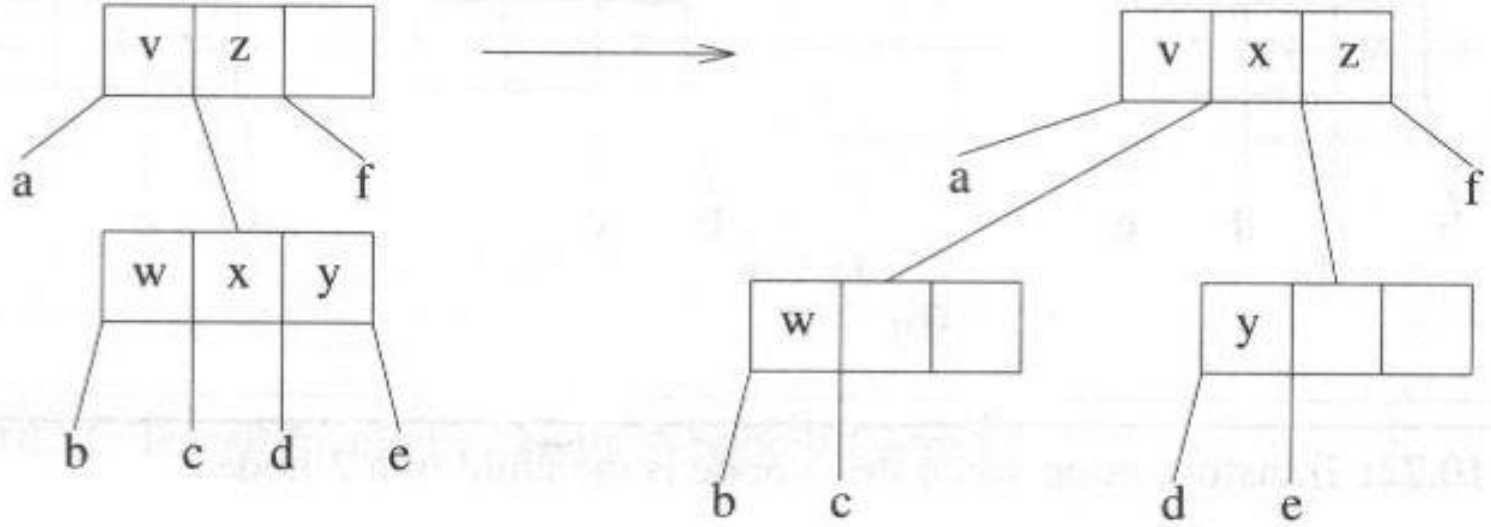


Figure 10.24: Transformation when the 4-node is the left middle child of a 3-node

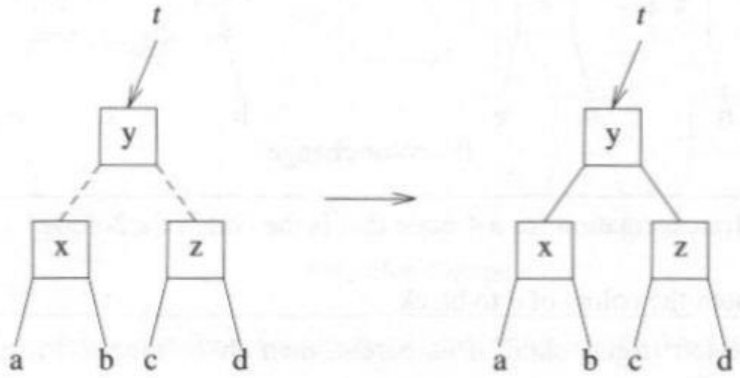
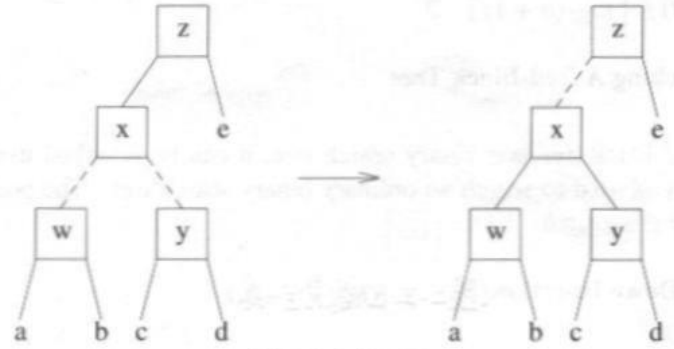


Figure 10.29: Transformation for a root 4-node



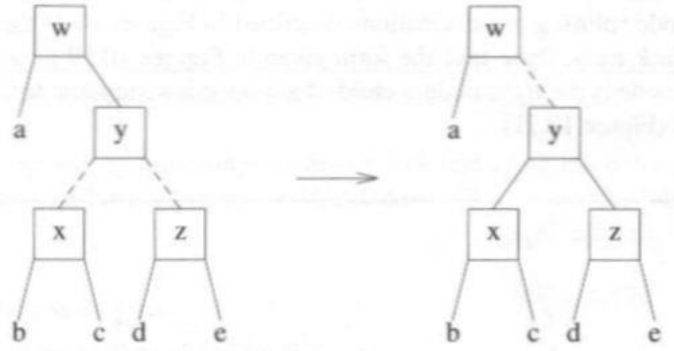


Figure 10.30: Transformation for a 4-node that is the child of a 2-node

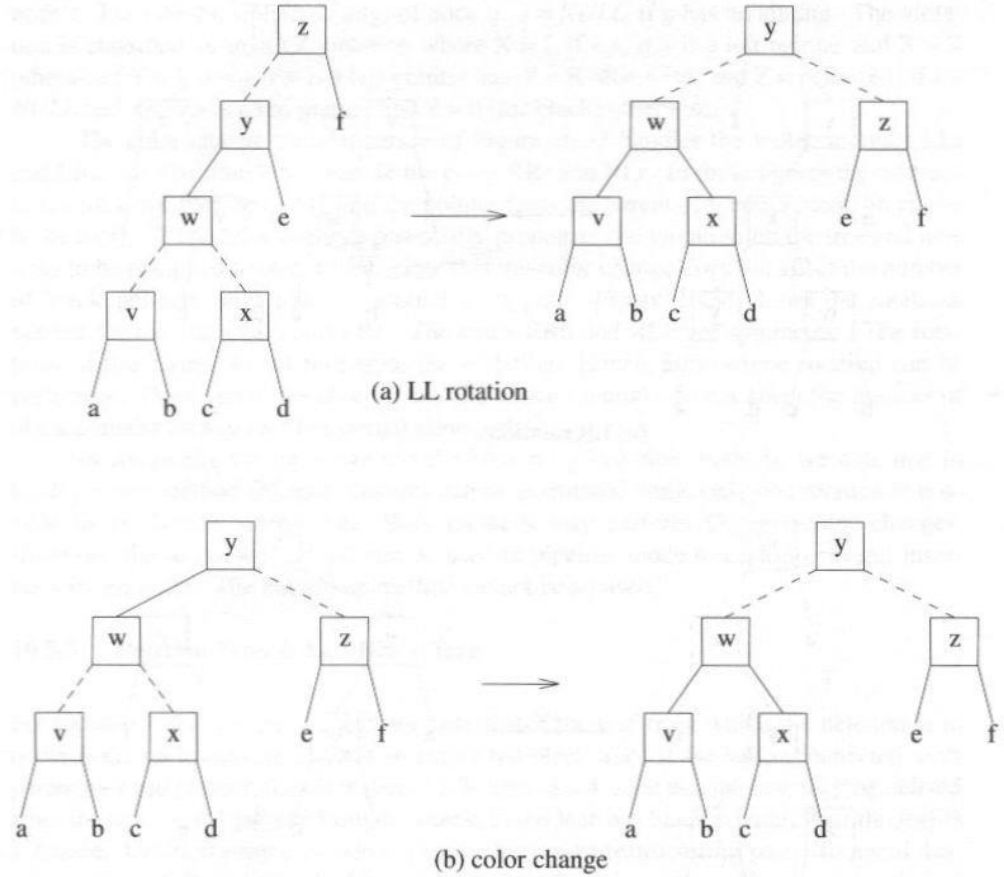


Figure 10.31: Transformation for a 4-node that is the left child of a 3-node

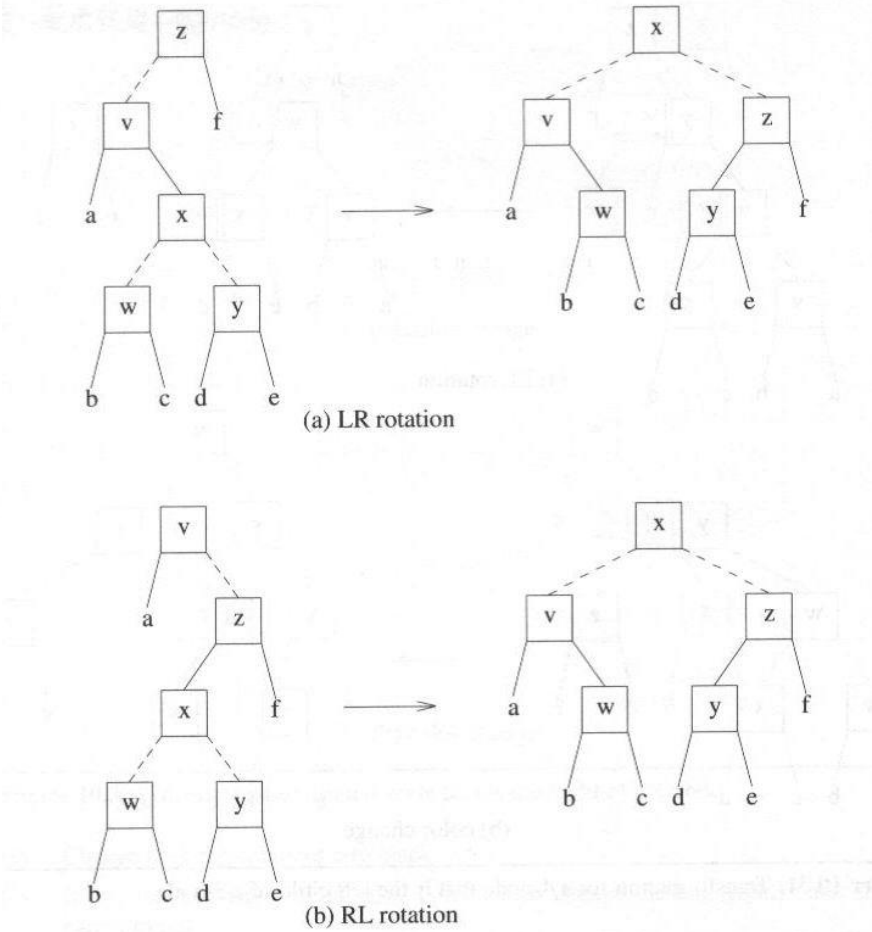


Figure 10.32: Transformation for a 4-node that is the left middle child of a 3-node

- We can detect a 4-node by simply looking for nodes *q* for which both color fields are red.
  - Such nodes together with their two children form a 4-node.
  - When such a *q* is detected, the transformations of Figures 10.29 through 10.32 are accomplished as below:
    - Change both colors of *q* to black.
    - If *q* is the left (right) child of its parent, then change the left (right) color of its parent to red.
    - If we now have two consecutive red pointers, then one is from the grandparent, *gp*, of *q* to the parent, *p* of *q* and the other from *p* to *q*. Let the direction of the first of these be *X* and that of the second be *Y*. We shall use L (R) to denote a left (right) direction. *XY* = LL, LR, and RL in the case of Figures 10.31(a), 10.32(a), and 10.32(b), respectively. For the case symmetric to Figure 10.31(a) that arises when the 4-node is a right middle child of a 3-node, *XY* = RR. A rotation similar to that performed in AVL trees is needed. We describe the rotation for the case *XY* = LL. Now, node *p* takes the place previously occupied by *pp*; the right child of *p* become the left child of *pp* and *pp* becomes the right child of *p*.

– It is interesting to note that when the 4-node to be split is a root or the child of a 2-node or that of "nicely" oriented 3-node (as in Figure 10.31(b)), color changes suffice. Pointers need to be changed only when the 4-node is the child of a 3-node that is not "nicely" oriented (as in Figure 10.31(a) and 10.32).

- Deletion from a red-black tree
  - For top down deletion from a leaf, we note that if the leaf from which the deletion is to occur is the root, then the result is an empty red-black tree.
  - If the leaf is connected to its parent by a red pointer, then it is part of a 3-node or a 4-node and the leaf may be delete from the root.
  - If the pointer from the parent to the leaf is a black pointer, then the leaf is a 2-node.
    - Deletion from a 2-node requires a backward restructuring pass.
    - To avoid this, we ensure that the deletion leaf has a red pointer from its parent.

- This is accomplished by using the insertion transformations in the reverse direction together with red-black transformations corresponding to the 2-3-4 deletion transformations (3) and (4) (*q* is a 2-node whose nearest sibling is a 3- or 4-node), and a 3-node transformation that switches from one 3-node representation to the other as necessary to ensure that the search for the element to be deleted moves down a red pointer.

– Since most of the insertion and deletion transformations can be accomplished by color changes and require no pointer changes or data shifts, these operations actually take less time using red black trees than when a 2-3-4 tree is represented using nodes of type *two*34.

10.6 B-trees

- Definition of *m*-way search trees
  - The balanced search trees that we have studied so far (AVL trees, 2-3 trees, 2-3-4 trees, and red-black trees) allow us to search, insert, and delete entries from a table in O(log*n*) time, where *n* is the number of entries in the table.
    - These structures are well suited to applications in which the table is small enough to be accommodated in internal memory.
  - However, when the table is too large for this, these structures do not have good performance. This is because we must now retrieve the nodes of the search tree structure form a disk.

- Since the time required for a disk access is significantly more than that for an internal memory access, we seek structures that would reduce the number of disk accesses.
- We shall use the term *index* to refer to a symbol table that resident on a disk.
  - The symbol table maybe assume to be too large to be accommodated in the internal memory of the target computer.
  - To obtain better performance, we shall use search trees whose degree is quite large.

Definition:

A *m-way search tree*, is either empty or satisfies the following properties;

The root has at most m subtrees and has the structure:

$$n, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_n, A_n)$$

where the A_i, $0 \leq i \leq n < m$ are pointers to subtrees and the K_i, $0 \leq i \leq n < m$ are key values.

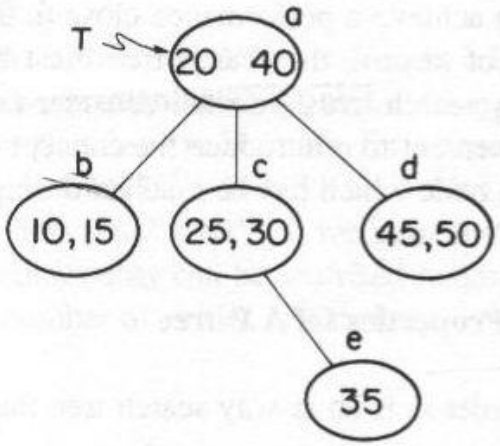
$K_i < K_{i+1}$, $1 \leq i < n$.

All key values in the subtree A_i are less than K_{i+1} and gretater than K_i, $0 < i < n$.

All key values in the subtree A_n are greater than K_n and those in A_0 are less than K_1.

The subtrees A_i, $0 \leq i \leq n$ are also m-way search trees.

– AVL trees are 2-way search trees, 2-3 trees are 3-way search tree, and 2-3-4 trees are 4-way search trees.
  - Of course, there are 2-way search trees that are not AVL trees, 3-way search trees that are not 2-3 trees, and 4-way search trees that are not 2-3-4 trees.
  - A 3-way search tree that is not a 2-3 tree is shown in Figure 10.35
  -



node	schematic format
a	2,b,(20,c),(40,d)
b	2,0,(10,0),(15,0)
c	2,0,(25,0),(30,e)
d	2,0,(45,0),(50,0)
e	1,0,(35,0)

Figure 10.35: Example of a 3-way search tree that is not a 2-3 tree

- Searching an m-way search tree
  - Suppose we wish to search the m-way search tree T for the key value x.
    - Assume that T resides on a disk. We begin by retrieving the root node from the disk.
    - Assume that this node has the structure given in the definition of an m-way search tree. For convenience, assume that $K_0 = -\infty$ and $K_{n+1} = +\infty$.
    - By searching the keys of the root, we determine i such that $K_i \leq x < K_{i+1}$.
      - If $x = K_i$, then the search is complete.
      - If $x \neq K_i$, then from the definition of an m-way search, it follows that if x is in the tree, it must be in subtree A_i.
    - So, we retrieve the root of this subtree from the disk and proceed to search it. This process continues until we either find x or we have determined that x is not in the tree.

- When the number of keys in the node being searched is small, a sequential search is used. When this number is large, a binary search may be used.

– In a tree of degree m and height h the maximum number of nodes is $\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m-1)$.
  - Since each node has at most m - 1 keys, the maximum number of keys in an m-way tree index of height h is m^h - 1.
  - For a binary tree with h = 3 this figure is 7. For a 200-way tree with h =3 w have m^h -1 = $8*10^6$ -1.

– Clearly, the potentials of high order search tree are much greater than those of low order search tree.
  - To achieve a performance close to that of the best m-way search trees for a given number of keys n, the search tree must be balanced.
  - The particular variety of balanced m-way search trees we shall consider her is known as a B-tree.

- Definition and properties of a B-tree

Definition:

A B-tree of order m is an m-way search tree that is either empty or satisfies the following properties:

The root node has at least 2 children

All nodes other than the root node and failure nodes have at least $\lceil m/2 \rceil$ children.

All failure nodes are at the same level.

– Observe that

- a 2-3 tree is a B-tree of order 3 and a 2-3-4 tree is a B-tree of order 4.
- All B-trees of order 2 are full binary trees.
- B-trees of order 2 exist only when the number of key value is $2^k - 1$ for some k.
- for any given number of keys and any m, $m > 2$, there is a B-tree of order m that contains this many keys.

– Number of key values in a B-tree

- A B-tree of order m in which all failure nodes are at the same level l+1 has at most m^l - 1 keys.
- What is the minimum number, N, of keys in such a B-tree?
  - From the definition of a B-tree we know that if l > 1, the root node has at least two children. Hence, three are at least two nodes at level 2.
  - Each of these nodes must have at least $\lceil m/2 \rceil$ children. Thus, there are at least $2\lceil m/2 \rceil$ nodes at level 3.
  - At level 4 there must be at least $2\lceil m/2 \rceil^2$ nodes, and continuing this argument, we see that there are at least $2\lceil m/2 \rceil^{l-2}$ nodes at level l when $l > 1$. All of these nodes are nonfailure nodes.
  - If the key values in the tree are $K_1, K_2, \ldots, K_N$ and $K_i < K_{i+1}$, $1 \leq i < N$, then the number of failure nodes is $N + 1$.
  - This results in N + 1different nodes that one could reach while searching for a key value x that is not in the B-tree.
  - Therefore, we have,
  - $$\begin{aligned} N + 1 &= \text{number of failure nodes} \\ &= \text{number of nodes at level } l + 1 \\ &\geq 2\lceil m/2 \rceil^{l-1} \end{aligned}$$

- This in turn implies that if there are N key values in a B-tree of order m, then all non failure nodes are at levels less than or equal to l, $l \leq \log_{\lceil m/2 \rceil}\{(N + 1)/2\} + 1$
  - The maximum number of accesses that have to be made for a search is l.
  - Using a B-tree of order $m = 200$, and index with $N \leq 2*10^6 - 2$ will have $l \leq \log_{100}\{(N + 1)/2\} + 1$.
  - Since l is integer, we obtain $l \leq 3$. For $n \leq 2*10^8 - 2$ we get $l \leq 4$.
- Thus, the use of a high order B-tree results in a tree index that can be searched making a very small number of disk accesses even when the number of entries is very large.

– Choice of *m*

- B-trees of high order are desirable since they result in a reduction in the number of disk accesses needed to search an index.
- If the index has *N* entries, then a B-tree of order $m = N + 1$ would have only one level.
  - This choice of *m* clearly is not reasonable, since by assumption the index is too large to fit in internal memory.
  - Consequently, the single node representing the index cannot be read into memory and processed.
- In arriving at a reasonable choice for *m*, we must keep in mind that we are really interested in minimizing the total amount of time needed to search the B-tree for a value *x*. This **cost** has two components,
  - the time for reading in the node from the disk
  - the time needed to search this node (in memory) for the key.

- Assume that each node of a B-tree of order m is of a fixed size.
- Let K_i be α characters long and the two pointers associated with each key are 2β characters long.
- Then the size of a node is approximately $m(\alpha+2\ \beta)$ characters.
- Hence, the time t_i required to read in a node is

$$t_i = t_s + t_l + m(\alpha+2\ \beta)\ t_c$$
$$= a + bm$$

where

$a = t_s + t_l$ = seek time + latency time

$b = (\alpha+2\ \beta)\ t_c$ and t_c = transmission time per char

- If binary search is used to search each node of the B-tree, then the internal processing time per node is $c \log_2 m + d$ for some constant c and d. the total processing time per node is thus

$$\tau = a + bm + c \log_2 m + d$$

- For an index with N entries, the number of levels, l, is bounded by:

$$l \leq \log_{\lceil m/2 \rceil} \{(N+1)/2\} + 1$$

$$\leq f \frac{\log_2 \{(N+1)/2\}}{\log_2 m} \text{ for some constant } f$$

- The maximum search time is $= \tau * l =$

$$\text{maximum search time} = g \left\{ \frac{a+d}{\log_2 m} + \frac{bm}{\log_2 m} + c \right\} \text{ seconds}$$

where $g = f * \log_2 \{(N+1)/2\}$.

- We therefore desire a value of m that minimizes the maximum search time **(代入實際值).**
  - If the disk drive available has a t_s = 1/100 sec and t_l = 1/40 sec, then a = 0.035 sec. Since d will typically be a few microseconds, we may ignore it in comparison with a.
  - Hence, $a + d \approx a$ =0.35 sec.
  - If each key value is at most six characters long and that each A_i and B_i is three characters long, $\alpha = 6$ and $\beta = 3$. If the transmission rate t_c is $5*10^{-6}$ sec/charac, then $b = (\alpha + 2\beta)t_c = 6*10^{-5}$ sec. The formula for the maximum search time now becomes:

$$g\left\{\frac{35}{\log_2 m}+\frac{0.06m}{\log_2 m}+1000c\right\}\text{ milliseconds}$$

- This function is tabulated in Figure 10.36 and plotted in Figure 10.37.
  - It is evident that there is a wide range of values of m for which nearly optimal performance is achieved. This corresponds to the almost flat region $m \in [50, 400]$.
  - In the case the lowest value of m in this region results in a node size greater than the allowable capacity of an input buffer, the value of m will be determined by the buffer size.
  -

m	Search time (sec)
2	35.12
4	17.62
8	11.83
16	8.99
32	7.38
64	6.47
128	6.10
256	6.30
512	7.30
1024	9.64
2048	14.35
4096	23.40
8192	40.50

Figure 10.36: Values of $(35 + .06m)/\log_2 m$

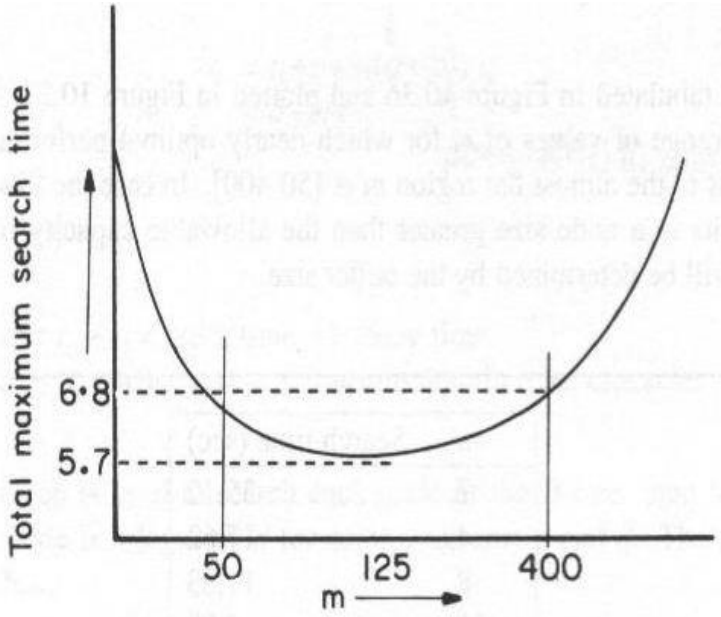


Figure 10.37: Plot of $(35+.06m)/\log_2 m$

- Insertion into a B-tree
  - The algorithm to insert a new key into a B-tree is a generalization of the two pass insertion algorithm for 2-3 trees.
    - While, for $m > 3$, we could also generalize the top-down insertion algorithm described for 2-3-4 trees, this is not desirable as this algorithm splits many nodes and each time we change a node, it has to be written to disk. This increases the number of disk accesses.
  - The insertion algorithm for B-trees of order *m* first performs a search to determine the leaf node, *p*, into which the new key is to be inserted.
    - If the insertion of the new key into *p* results in *p* having m keys, the node *p* is split. Otherwise, the new *p* is written to the disk and the insertion is complete.

- To split the node, assume that following the insertion of the new key, p has the format:

  $m, A_0, (K_1, A_1), \ldots, (K_m, A_m)$, and $K_i < K_{i+1}$, $1 \leq i < m$

- The node is split into two nodes p and q with the following formats:

  node p: $\lceil m/2 \rceil - 1, A_0, (K_1, A_1), \ldots, (K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

  node q: $m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}), \ldots, (K_m, A_m)$

- The remaining key $K_{\lceil m/2 \rceil}$ and a pointer to the new node q form a tuple $(K_{\lceil m/2 \rceil}, q)$. This is to be insert to the parent of p.

– As in the case of 2-3 trees, inserting into the parent may require us to split the parent and this splitting process can propagate all the way up to the root.

  - When the root splits, a new root with a single key is created and the height of the B-tree increases by one.

- **Analysis of B-tree insertion:**
  - The average number of disk accesses is only $l + 2s + 1 < l + 101/99 \approx l + 1$.
    - h is the height of the B-tree.
    - The average number of splits, s, may determined as blow

      s = (total number of splits)/N

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

      $< 1/(\lceil m/2 \rceil - 1)$
    - p is the number of non failure nodes in the final B-tree with N entries.

- Deletion from a B-tree
  - The deletion algorithm for B-trees is also a generalization of the deletion algorithm for 2-3 trees.
    - The deletion of x from a nonleaf node can be transformed into a deletion from a leaf.
  - There are four cases when deleting from a leaf node *p*.

    p is also the root. If the root is left with at least one key, the changed root is written to disk and we are done. Otherwise, the B-tree is empty following the deletion. In the remaining cases, *p* is not root.

    Following the deletion, *p*, has at least $\lceil m/2 \rceil - 1$ keys. The modified leaf is written to disk and we are done.

p has $\lceil m/2 \rceil - 2$ keys and its nearest sibling, q, has at least $\lceil m/2 \rceil$.

- To determine this, we examine only one of the at most two nearest siblings that p may have.
- p is deficient as it has one less than the minimum number of keys required. q has more keys than the minimum required.
  - » As in the case of a 2-3 tree, a rotation is performed.
  - » In this rotation, the number of keys in q decreases by one while the number in p increases by one.
  - » As the result, neither p nor q are deficient following the rotation. The rotation leaves behind a valid B-tree.
- Let r be the parent of p and q. If q is the nearest right sibling of p, then let i be such that K_i is the ith key in r, all keys in p are less than K_i, and all those in q are greater than K_i.
- For the rotation, K_i is replaced by the first (i.e., the smallest) key in q, K_i becomes the rightmost key in p, and the leftmost subtree of q become the right most subtree of p.
- The changed nodes p, q, and r are written to disk and the deletion is complete. The case when q is the nearest left sibling of p is similar.

p has $\lceil m/2 \rceil - 2$ keys while q has at least $\lceil m/2 \rceil - 1$.

- So, p is deficient and q has the minimum number of keys permissible for a non root node.
- Now, Nodes p and q and the key K_i are combined to form a single node.
  - » The combined node has $(\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 = 2\lceil m/2 \rceil - 2 \leq m - 1$ keys which will at most fill the node.
  - » The combined node is written to disk.
- The combining operation reduces the number of keys in the parent node r by one.
- If the parent does not become deficient, the changed parent is written to disk and we are done. Otherwise, if the deficient parent is the root, it is discarded as it has no key.
- If the deficient parent is not the root, it has exactly $\lceil m/2 \rceil - 2$ keys.
  - » To remove this deficiency, we first attempt a rotation with one of r's nearest siblings.
  - » If this isn't possible, a combine is done.
  - » This process of combining can continue up the B-tree only until the children of root are combine.

- **Analysis of B-tree deletion:**
  - For a B-tree of height h, h disk accesses are made to find the node from which the key is to be deleted and to transform the deletion to that from a leaf.
  - The total number of disk accesses is $3h - 1$.

B-tree Insertion/Deletion Examples

在以下的 foils 裡:

我們先將 B-tree 的 insertion/deletion 動作列出；

後面再附上這些動作連同文字的部份，以及 B*-tree 和 B^+-tree 的內容。

FIGURE 7.6 Inserting the number 6 into a full leaf.

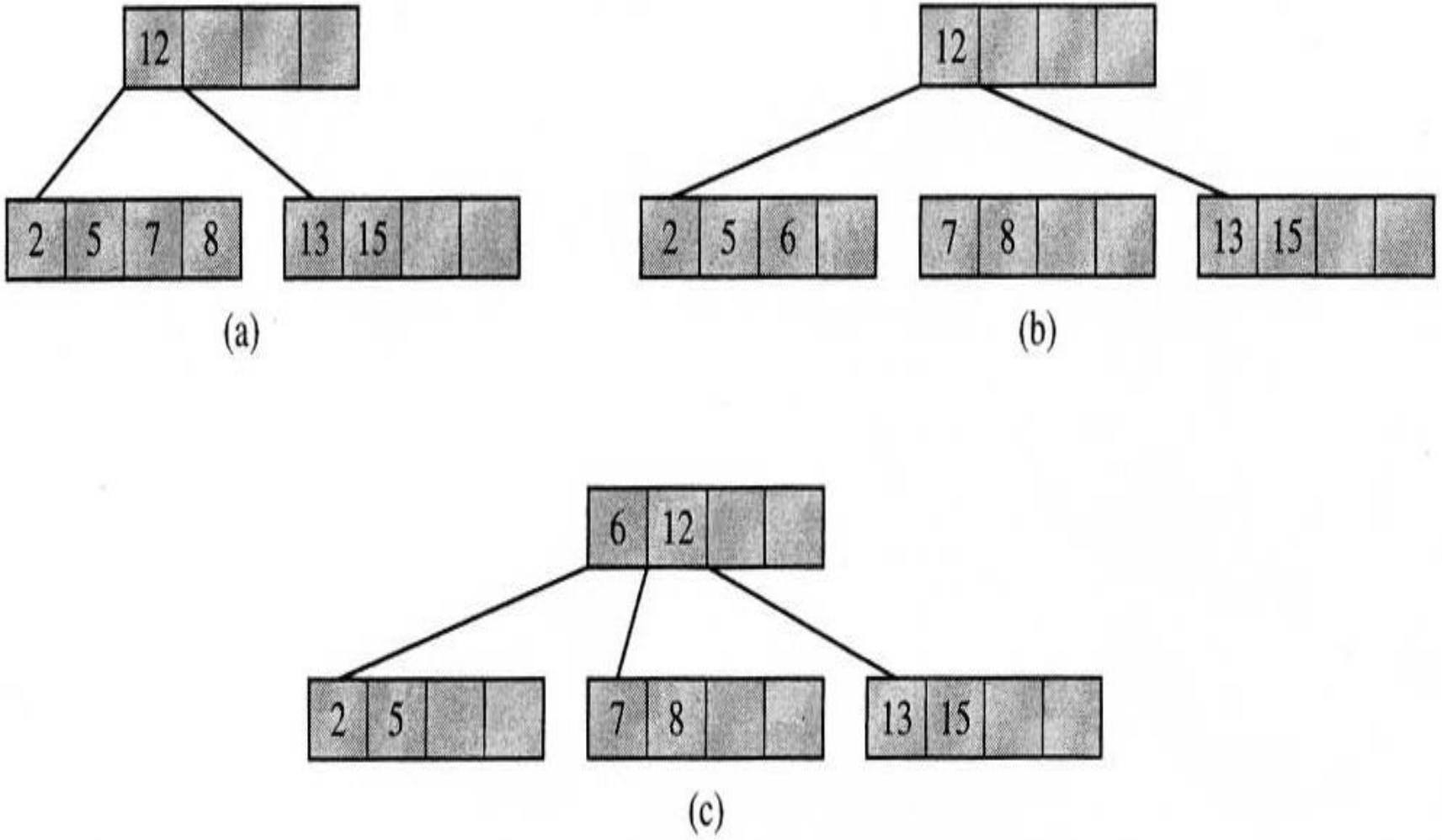


FIGURE **7-7** Inserting the number 13 into a full leaf.

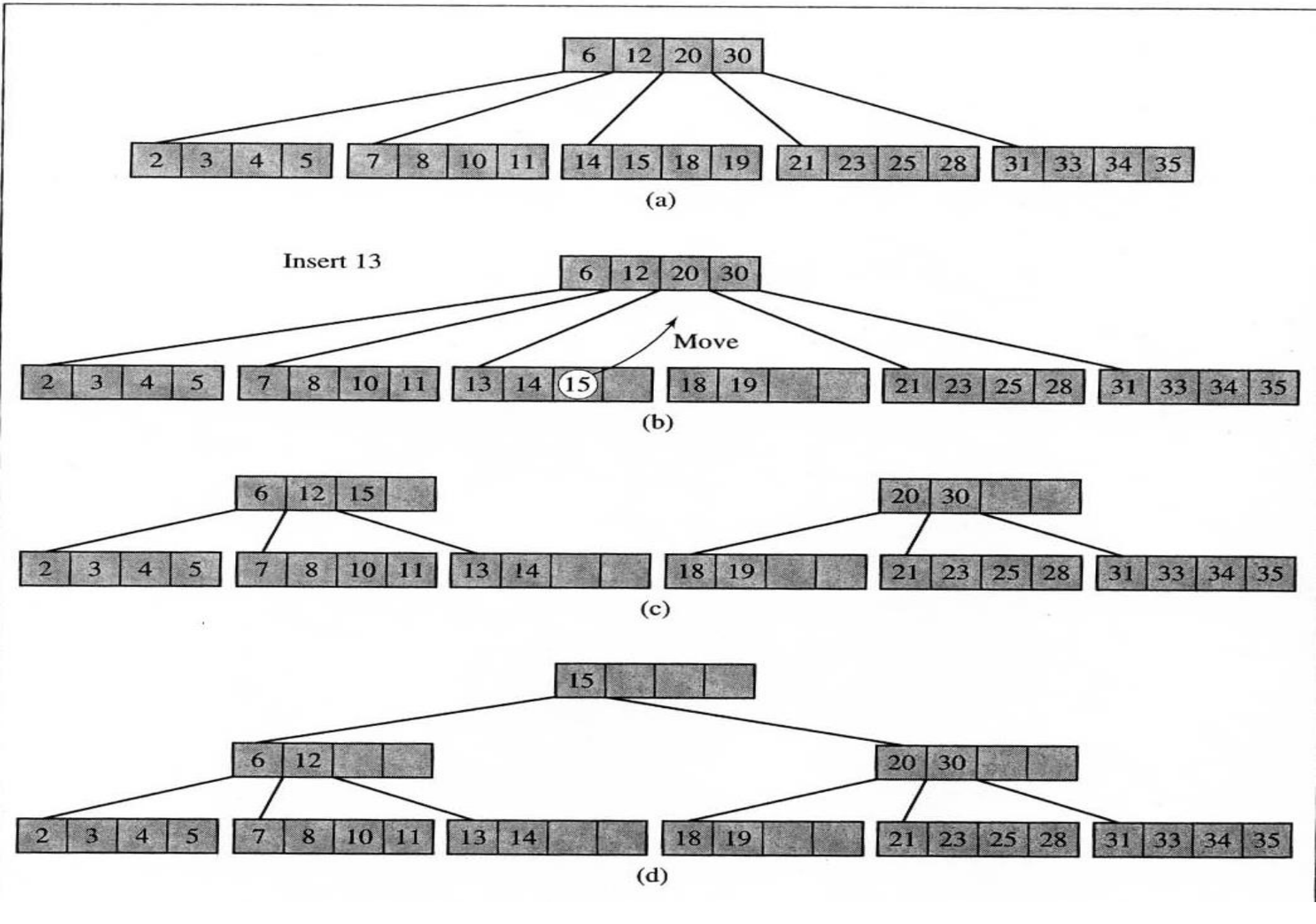


FIGURE 7.8 Building a B-tree of order 5 with the `BTreeInsert()` algorithm.

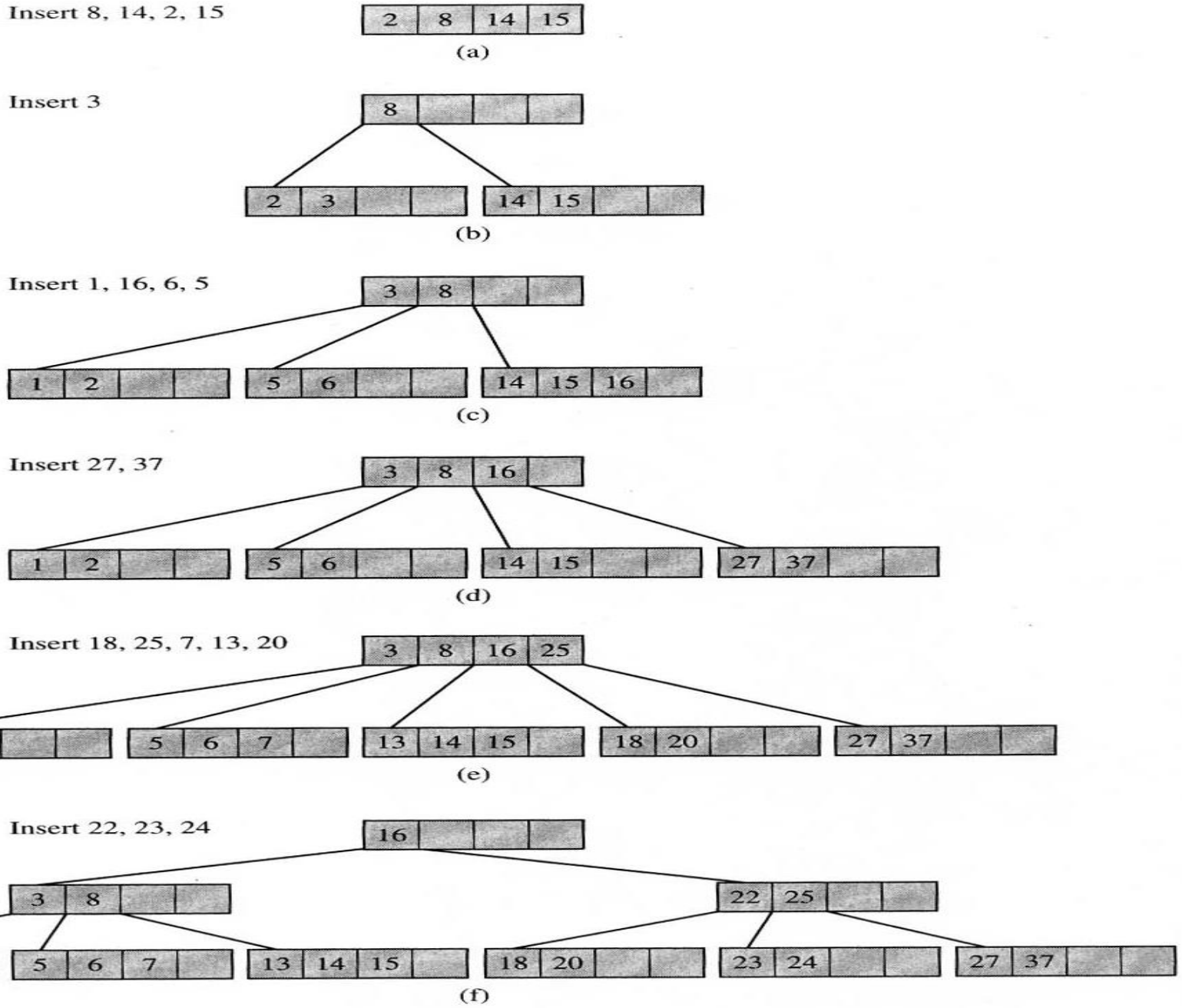
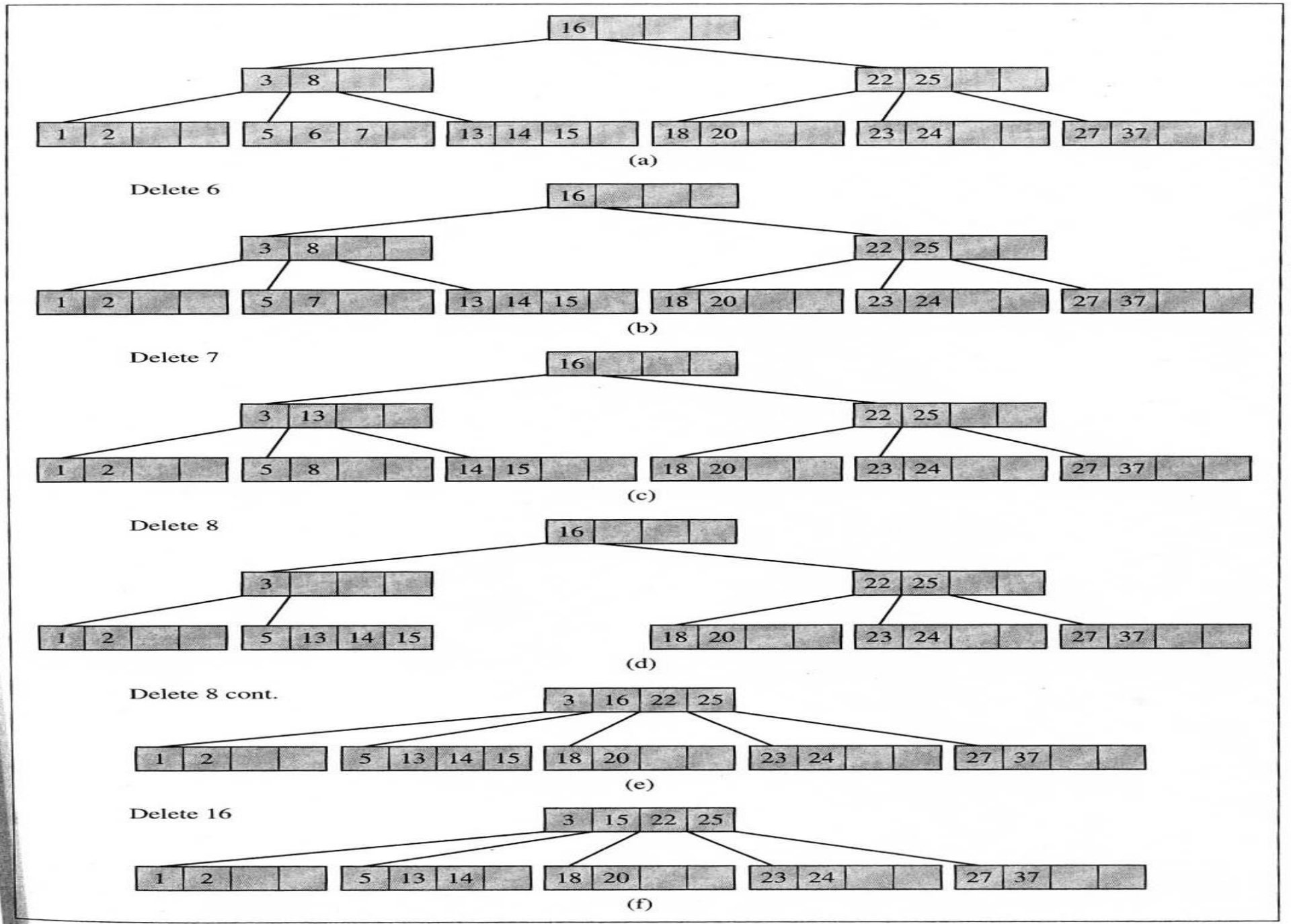


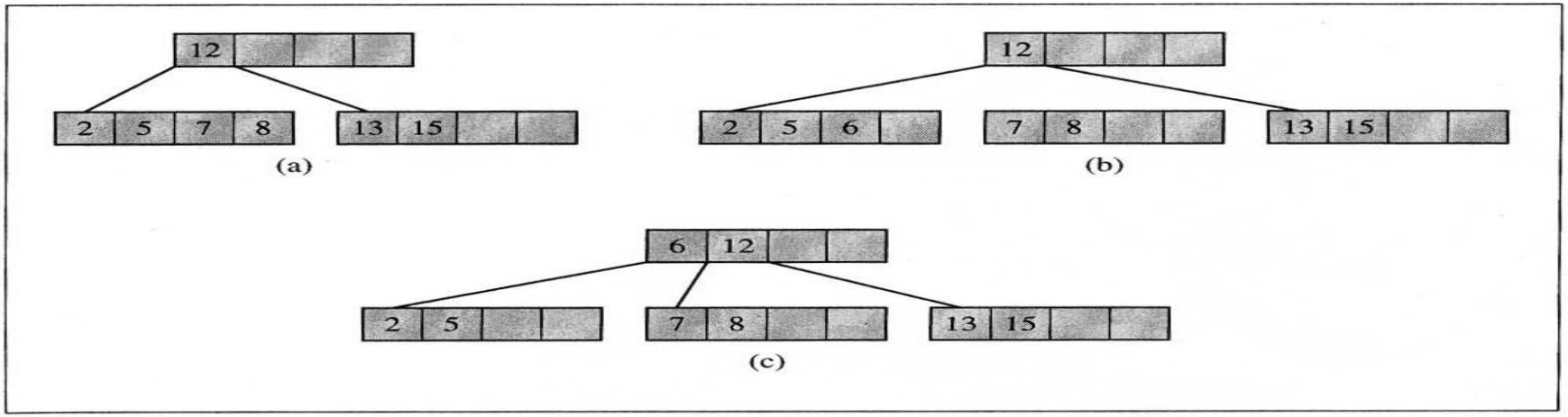
FIGURE 7.9 Deleting keys from a B-tree.



(附件)

以下是前面 B-tree insertion/deletion 例子的原圖 (含文字的部份) 和 B*-tree 及 B^+-tree 的部份。

FIGURE 7.6 Inserting the number 6 into a full leaf.



now two B-trees have to be combined into one. This is achieved by creating a new root and moving the last key from the old root to it (7.7d). It should be obvious that it is the only case in which the B-tree increases in height.

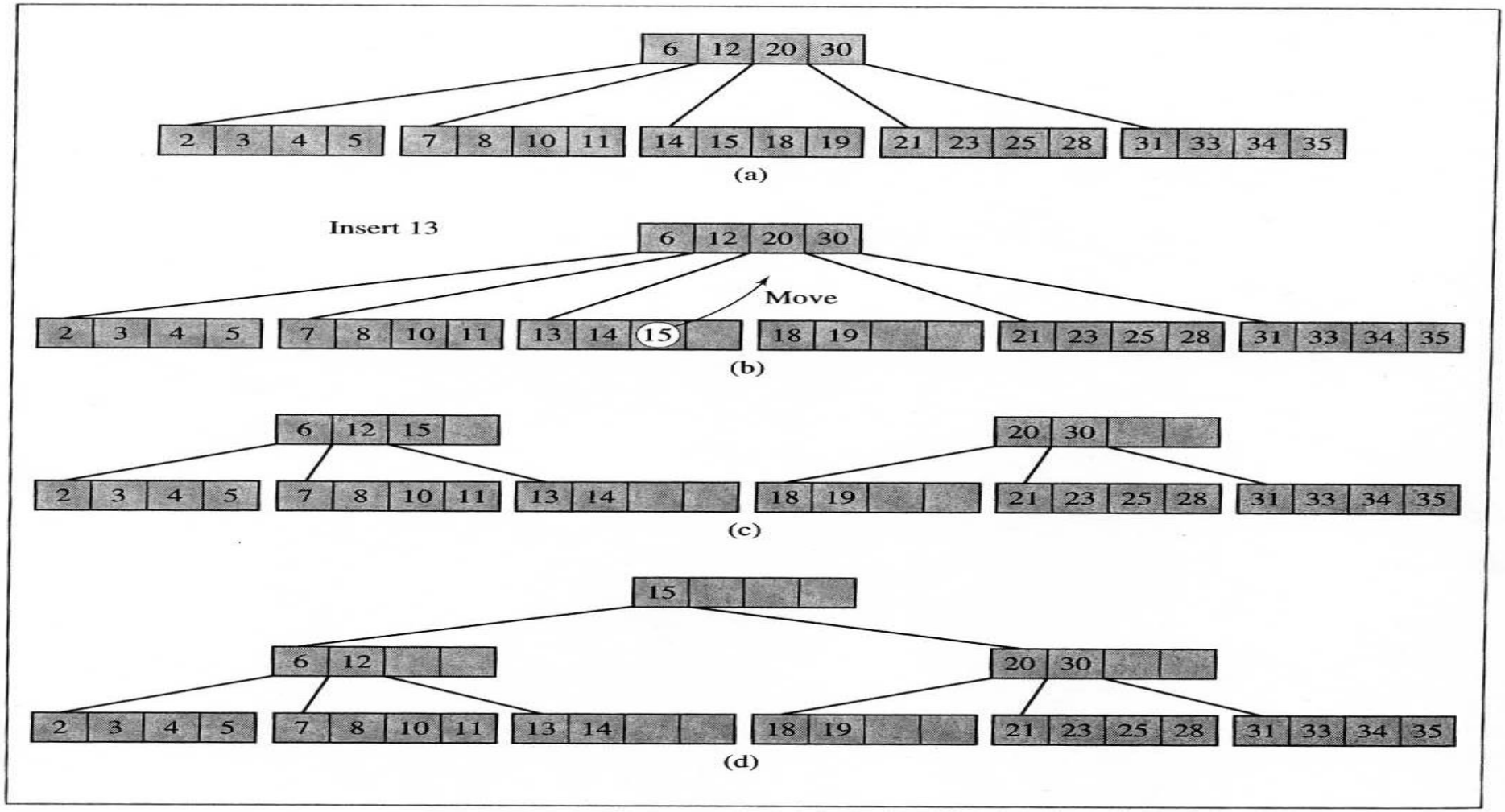
An algorithm for inserting keys in B-trees follows:

```
BTreeInsert (K)
  find a leaf node to insert K;
   while (1)
      find a proper position in array keys for K;
      if node is not full
         insert K and increment keyTally;
         return;
      else split node in node1 and node2;// node1 = node, node2 is new;
         distribute keys and pointers evenly between node1 and node2 and
         initialize properly their keyTally's;
         K = the last key of node1;
         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.



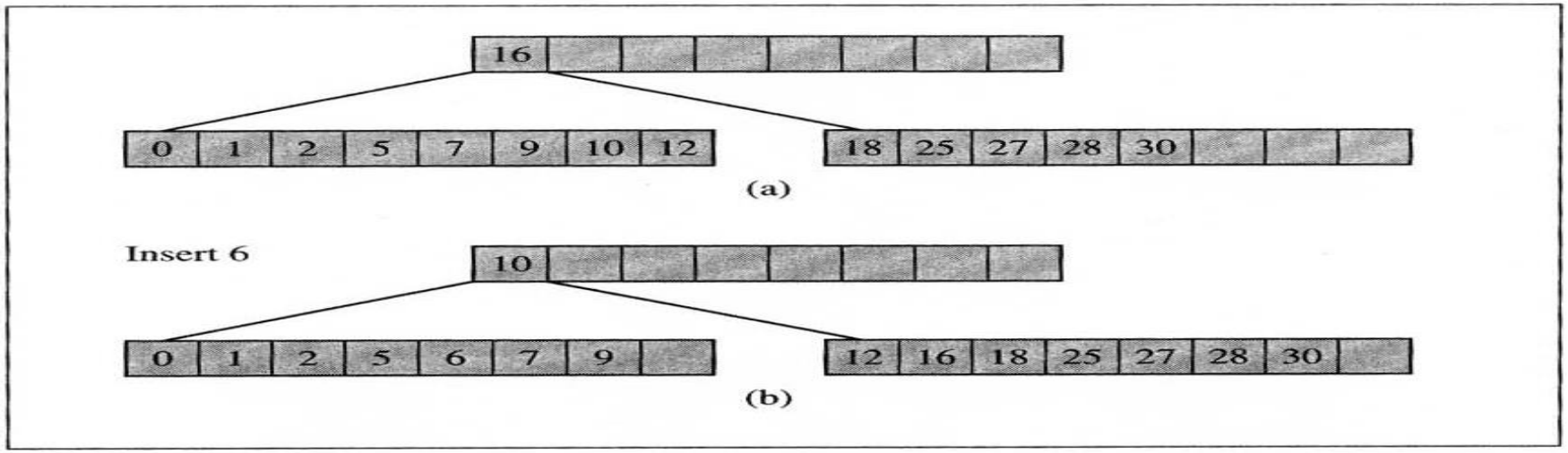
How often are node splits expected to occur? A split of the root node of a B-tree creates two new nodes. All other splits add only one more node to the B-tree. During the construction of a B-tree of p nodes, $p - h$ splits have to be performed, where h is the height of the B-tree. Also, in a B-tree of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

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

Overflow in a B*-tree is circumvented by redistributing keys between an overflowing node and its sibling.



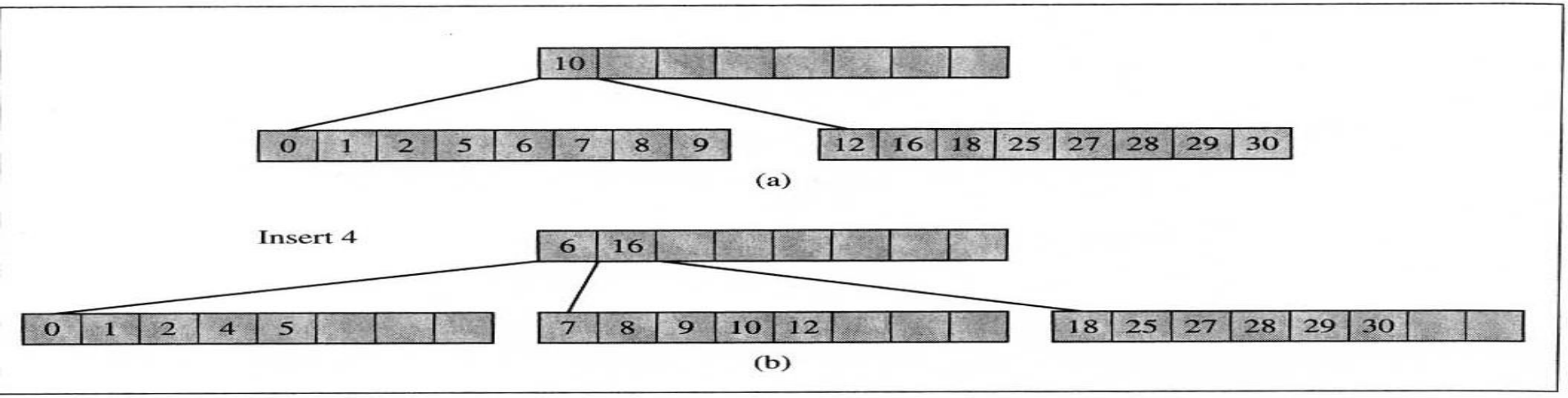
A B^*-tree is a variant of the B-tree introduced by Donald Knuth and named by Douglas Comer. In a B*-tree, all nodes except the root are required to be at least two-thirds full, not just half full as in a B-tree. More precisely, the number of keys in all nonroot nodes in a B-tree of order m is now k for $\lfloor \frac{2m-1}{3} \rfloor \leq k \leq m-1$. The frequency of node splitting is decreased by delaying a split, and when the time comes, by splitting two nodes into three, not one into two. The average utilization of B*-tree is 81% (Leung, 1984).

A split in a B*-tree is delayed by attempting to redistribute the keys between a node and its sibling when the node overflows. Figure 7.10 contains an example of a B*-tree of order 9. The key 6 is to be inserted into the left node, which is already full. Instead of splitting the node, all keys from this node and its sibling are evenly divided and the median key, key 10, is put into the parent. Notice that this not only evenly divides the keys, but also the free spaces so that the node which was full is now able to accommodate one more key.

If the sibling is also full, a split occurs: One new node is created, the keys from the node and its sibling (along with the separating key from the parent) are evenly divided among three nodes, and two separating keys are put into the parent (see Figure 7.11). All three nodes participating in the split are guaranteed to be two-thirds full.

Note that, as may be expected, this increase of a *fill factor* can be done in a variety of ways, and some database systems allow the user to choose a fill factor between .5 and 1. In particular, a B-tree whose nodes are required to be at least 75% full is called a B**-tree (McCreight 1977). The latter suggests a generalization: A B^n-tree is a B-tree whose nodes are required to be $\frac{n+1}{n+2}$ full.

FIGURE 7.11 If a node and its sibling are both full in a B*-tree, a split occurs: A new node is created and keys are distributed between three nodes.



7.1.3 B+-Trees

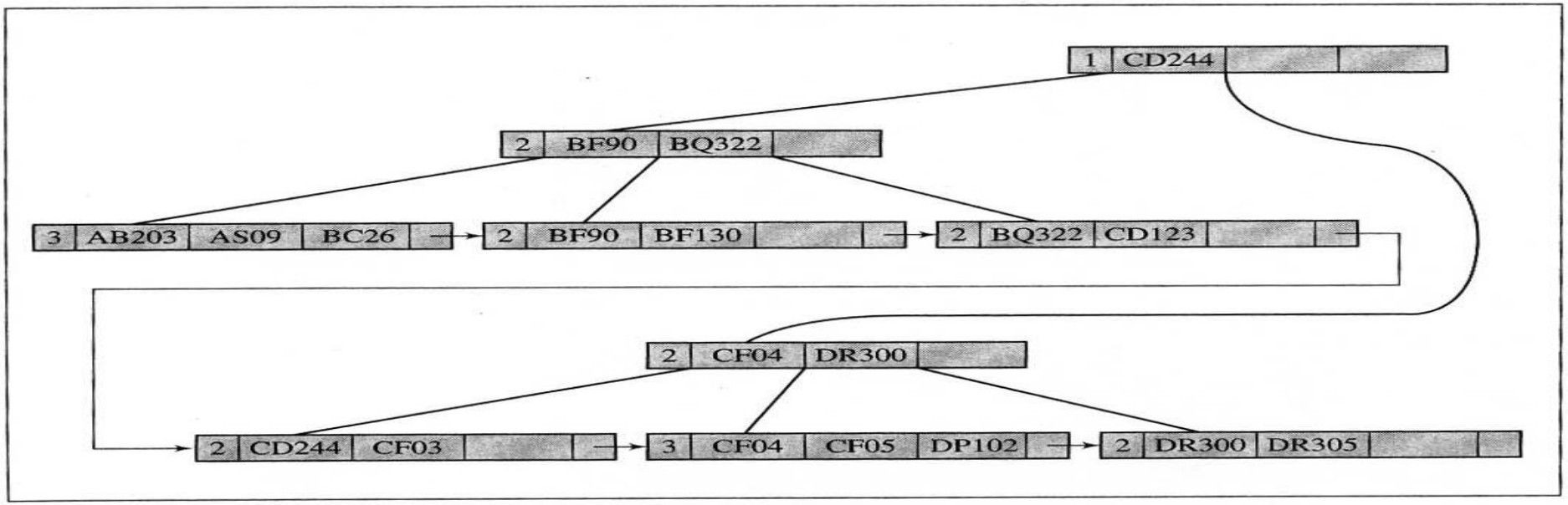
Since one node of a B-tree represents one secondary memory page or block, the passage from one node to another requires a time-consuming page change. Therefore, we would like to make as few node accesses as possible. What happens if we request that all the keys in the B-tree be printed in ascending order? An inorder tree traversal can be used which is easy to implement, but for nonterminal nodes, only one key is displayed at a time and then another page has to be accessed. Therefore, we would like to enhance B-trees to allow us to access data sequentially in a faster manner than using inorder traversal. A *B+-tree* offers a solution (Wedekind 1974).[2]

In a B-tree, references to data are made from any node of the tree, but in a B+-tree, these references are made only from the leaves. The internal nodes of a B+-tree are indexes for fast access of data; this part of the tree is called an *index set*. The leaves have a different structure than other nodes of the B+-tree, and usually they are linked sequentially to form a *sequence set* so that scanning this list of leaves results in data given in ascending order. Hence, a B+-tree is truly a B plus tree: It is an index implemented as a regular B-tree plus a linked list of data. Figure 7.12 contains an example of a B+-tree. Note that internal nodes store keys, pointers, and a key count. Leaves store keys, references to records in a data file associated with the keys, and pointers to the next leaf.

Operations on B+-trees are not very different from operations on B-trees. Inserting a key into a leaf which still has some room requires putting the keys of this leaf in

[2]Wedekind, who considered these trees to be only "a slight variation" of B-trees, called them B*-trees.

FIGURE **7.12** An example of a B^+-tree of order 4.



order. No changes are made in the index set. If a key is inserted into a full leaf, the leaf is split, the new leaf node is included in the sequence set, keys are distributed evenly between the old and the new leaves, and the first key from the new node is copied (not moved, as in a B-tree) to the parent. If the parent is not full, this may require local reorganization of the keys of the parent (see Figure 7.13). If the parent is full, the splitting process is performed the same way as in B-trees. After all, the index set is a B-tree.

Deleting a key from a leaf leading to no underflow requires putting the remaining keys in order. No changes are made to the index set. In particular, if a key which occurs only in a leaf is deleted, then it is simply deleted from the leaf but can remain in the internal node. The reason is that it still serves as a proper guide when navigating down the B^+-tree because it still properly separates keys between two adjacent children even if the separator itself does not occur in either of the children. The deletion of key 6 from the tree in Figure 7.13b results in the tree in Figure 7.14a. Note that the number 6 is not deleted from an internal node.

When the deletion of a key from a leaf causes an underflow, then either the keys from this leaf and the keys of a sibling are redistributed between this leaf and its sibling or the leaf is deleted and the remaining keys are included in its sibling. Figure 7.14b illustrates the latter case. After deleting the number 2, an underflow occurs and two leaves are combined to form one leaf. The first key from the right sibling of the node remaining after merging is copied to the parent node, and keys in the parent are put in order. Both these operations require updating the separator in the parent. Also, removing a leaf may trigger merges in the index set.

FIGURE **7.13** An attempt to insert the number 6 in the first leaf of a B^+-tree.

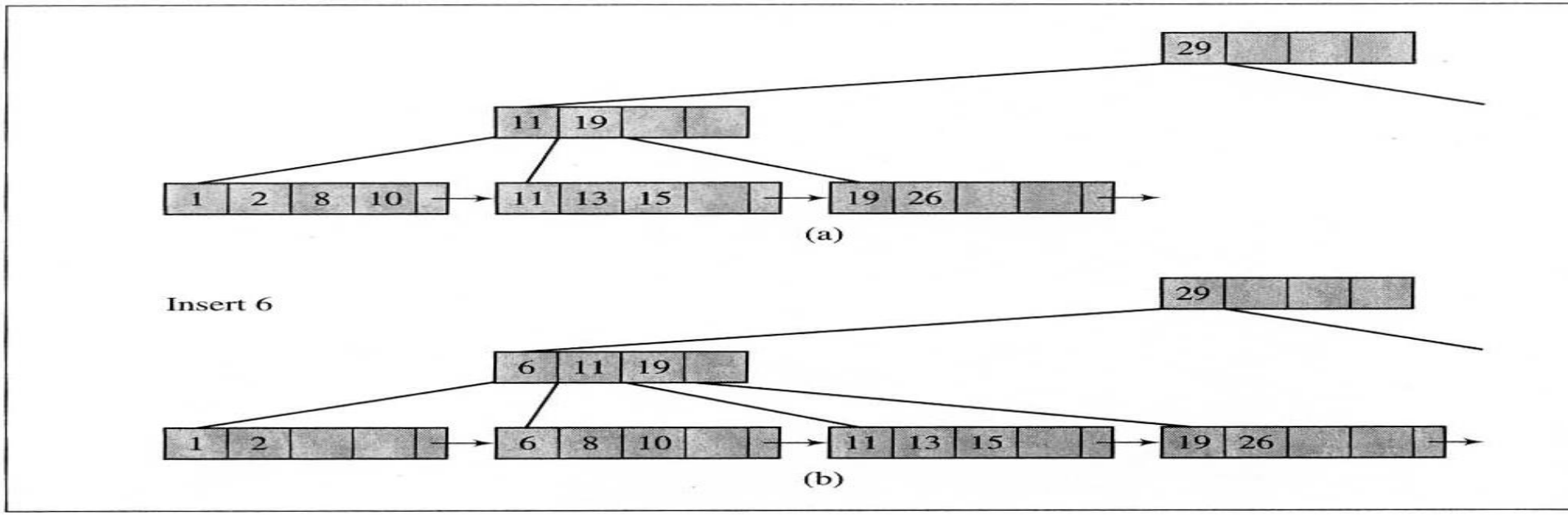
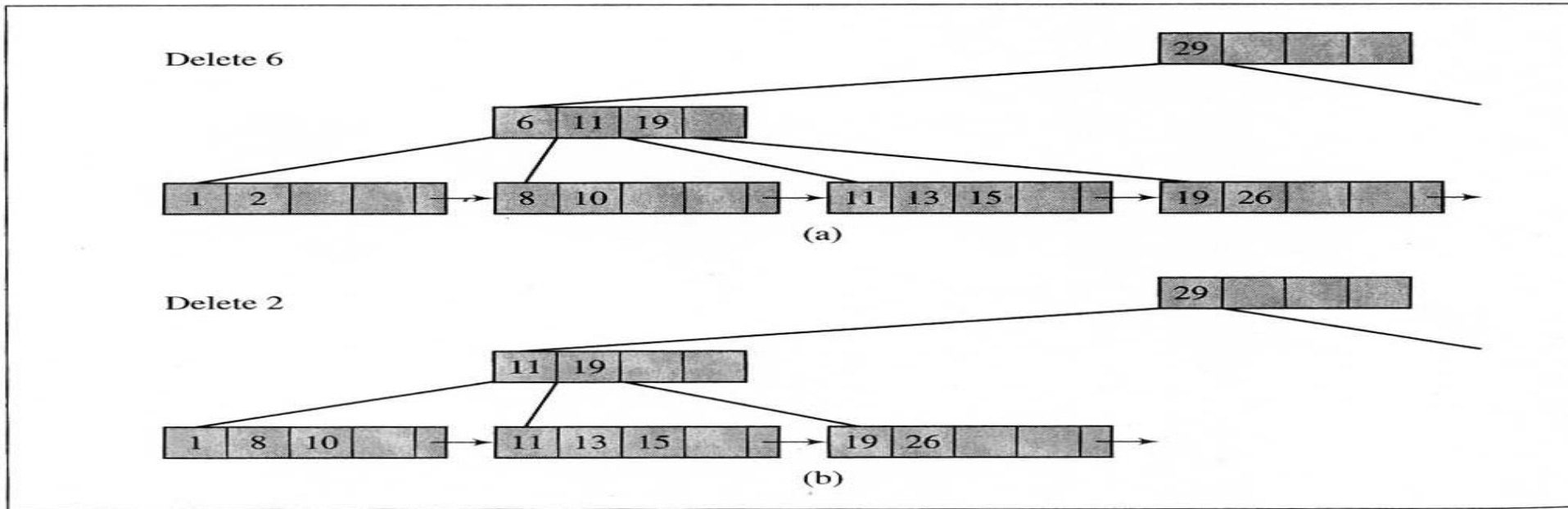


FIGURE **7.14** Actions after deleting the number 6 from the B^+-tree in Figure 7.13b.



- Variable size key values
  - With a node format of the form $n, A_0, (K_1, A_1), \ldots, (K_n, A_n)$, the first problem created by the use of variable size key values, K_i, is that a binary search can no longer be carried out.
    - Since, given the location of the first tuple (K_1, A_1) and n, we cannot easily determine K_n or even the location of $K_{(1+n)/2}$.
    - When the range of key value size is small, it is best to allocate enough space for the largest size key value.
    - When the range in sizes is large, storage my be waste and another node format may become better, i.e., the format

      $$n, A_0, \alpha_1, \alpha_2, \ldots, \alpha_n, (K_1, A_1), \ldots, (K_n, A_n)$$

      where α_i is the address of K_i in internal memory, i.e., K_i = memory(α_i) .
    - In this case, a binary search of the node can still be made.

- The use of variable size nodes is not recommended
  - Since this would require a more complex storage management system.
  - More importantly, the use of variable size nodes would result in degraded performance during insertion, as an insertion into a node would require us to request a larger node to accommodated the new value being inserted.
- Nodes of a fixed size should be used.
  - The size should be such as to allow for at least m - 1 key values of the largest size. During insertions, however, we can relax the requirement that each have $\leq m$ -1 key values. Instead, a node will be allowed to hold as many values as can fit into it and will contain at least $\lceil m/2 \rceil$ - 1 values.
  - Another possibility is to use some kind of key sampling scheme to reduce the key value size so as not to exceed some predetermined size, d.