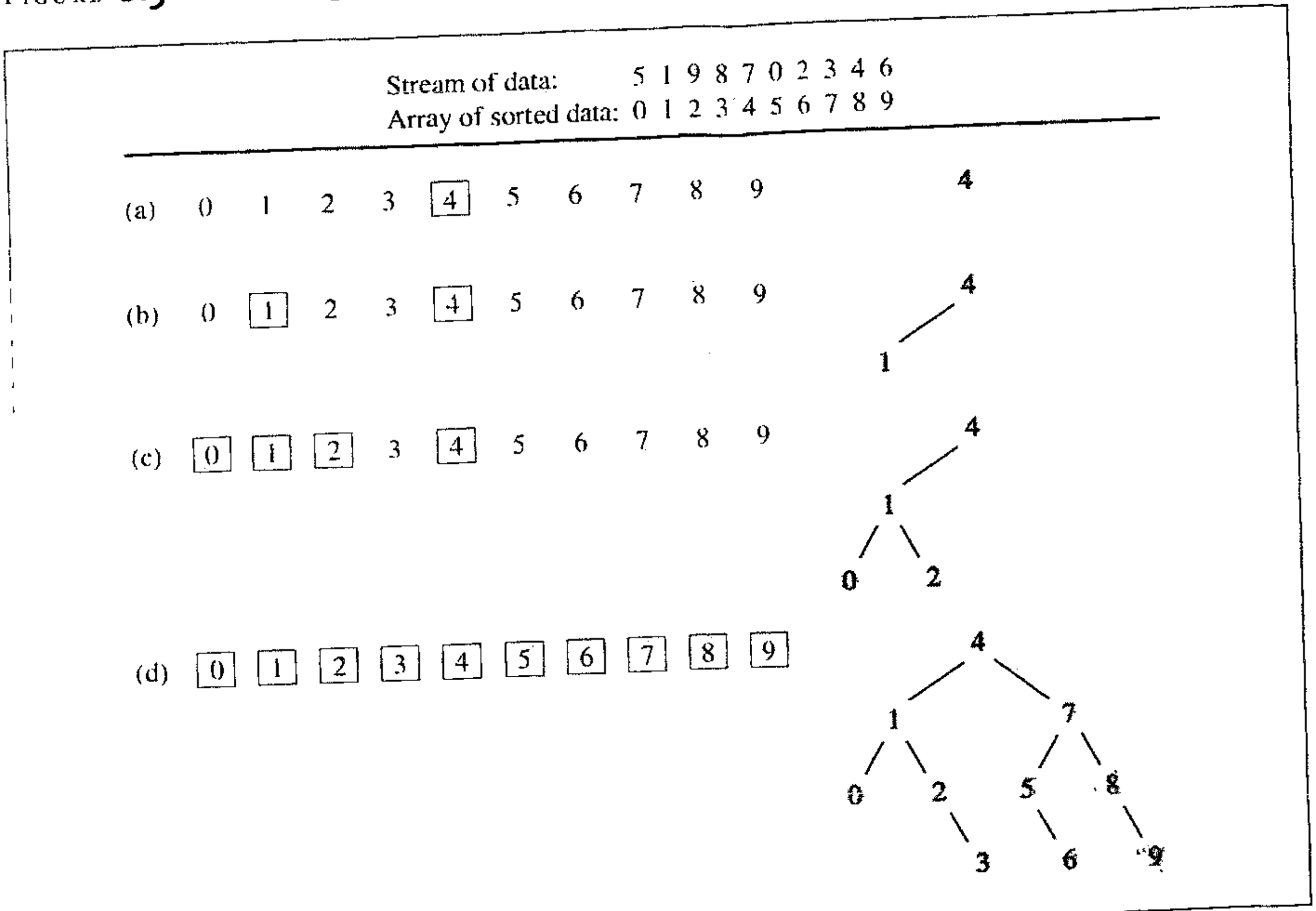
FIGURE 6.36 Creating a binary search tree from an ordered array.


Stream of data: 5 1 9 8 7 0 2 3 4 6
Array of sorted data: 0 1 2 3 4 5 6 7 8 9
(a)
(b)
(c)
(d)

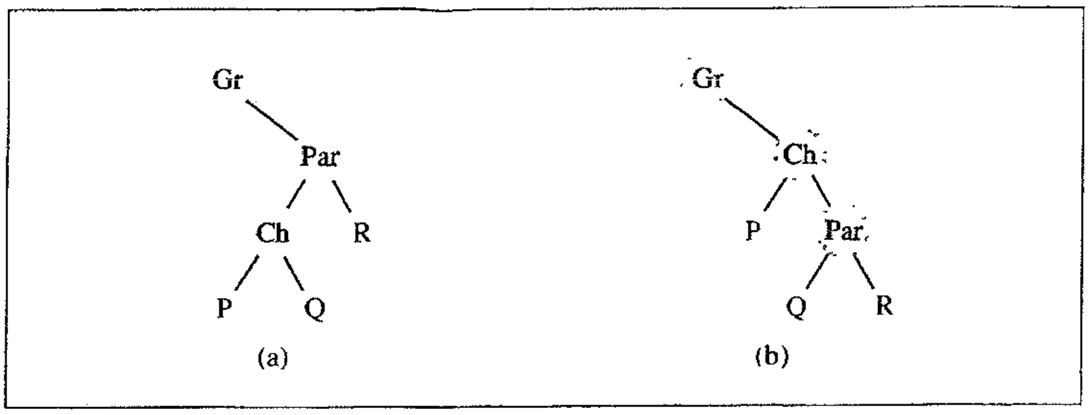
6.7.1 The DSW Algorithm

The algorithm discussed in the previous section was somewhat inefficient in that it required an additional array which needed to be sorted before the construction of a perfectly balanced tree began. To avoid sorting, it required deconstructing and then reconstructing the tree, which is inefficient except for relatively small trees. There are, however, algorithms that require little additional storage for intermediate variables and use no sorting procedure. The very elegant DSW algorithm was devised by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.

The building block for tree transformations in this algorithm is the *rotation*. There are two types of rotation, left and right, which are symmetrical to one another. The right rotation of the node `Ch` about its parent `Par` is performed according to the following algorithm:

```
rotateRight (Gr, Par, Ch)
  if Par is not the root of the tree // i.e., if Gr is not null
     grandparent Gr of child Ch becomes Ch's parent by replacing Par;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

FIGURE 6.37 Right rotation of child Ch about parent Par.



The steps involved in this compound operation are shown in Figure 6.37. The third step is the core of the rotation, when `Par`, the parent node of child `Ch`, becomes the child of `Ch`, when the roles of a parent and its child change. However, this exchange of roles cannot affect the principal property of the tree, namely, that it is a search tree. The first and the second steps of `rotateRight()` are needed to ensure that, after the rotation, the tree remains a search tree.

Basically, the DSW algorithm transfigures an arbitrary binary search tree into a linked listlike tree called a *backbone* or *vine*. Then this elongated tree is transformed in a series of passes into a perfectly balanced tree by repeatedly rotating every second node of the backbone about its parent.

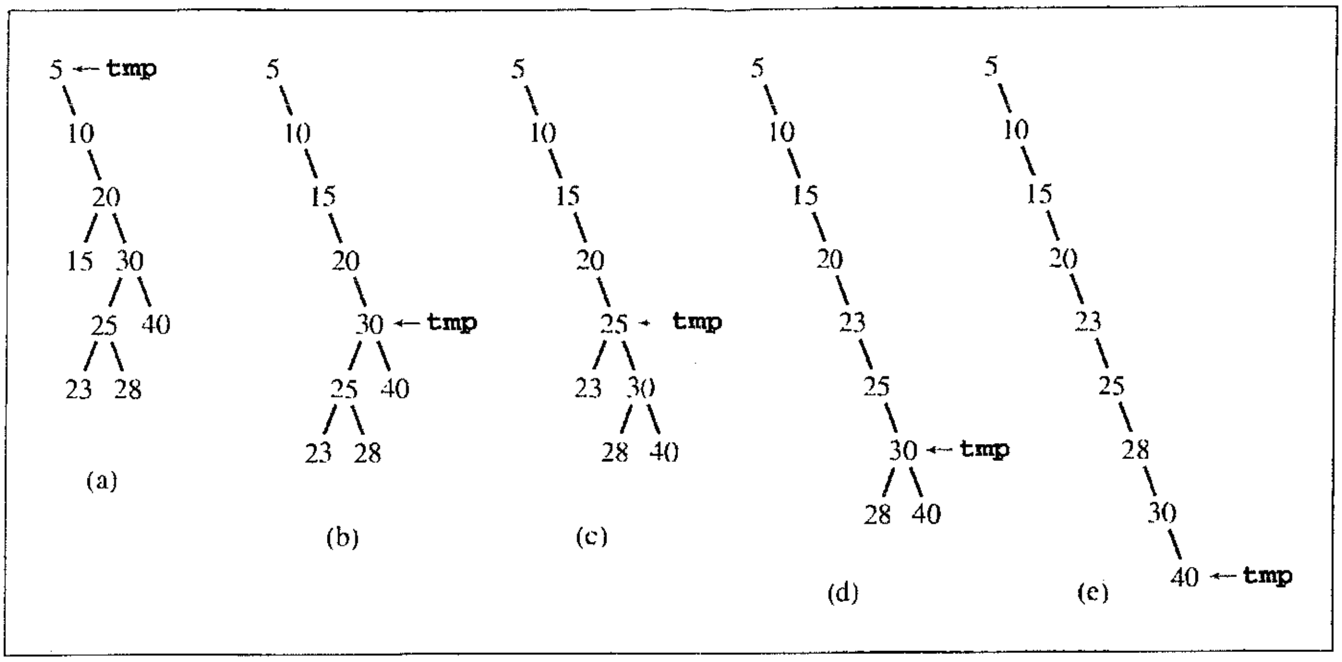
In the first phase, a backbone is created using the following routine:

```
createBackbone(root, n)
  tmp = root;
  while (tmp != 0)
    if tmp has a left child
        rotate this child about tmp; // hence the left child
                                     // becomes parent of tmp;
        set tmp to the child which just became parent;
    else set tmp to its right child;
```

This algorithm is illustrated in Figure 6.38. Note that a rotation requires knowledge about the parent of `tmp`, so another pointer has to be maintained when implementing the algorithm.

In the best case, when the tree is already a backbone, the `while` loop is executed n times and no rotation is performed. In the worst case, when the root does not have a right child, the `while` loop executes $2n - 1$ times with $n - 1$ rotations performed, where n is the number of nodes in the tree; that is, the run time of the first phase is $O(n)$. In this case, for each node except the one with the smallest value, the left child of `tmp` is rotated about `tmp`. After all rotations are finished, `tmp` points to the root, and after n iterations, it descends down the backbone to become null.

FIGURE 6.38 Transforming a binary search tree into a backbone.



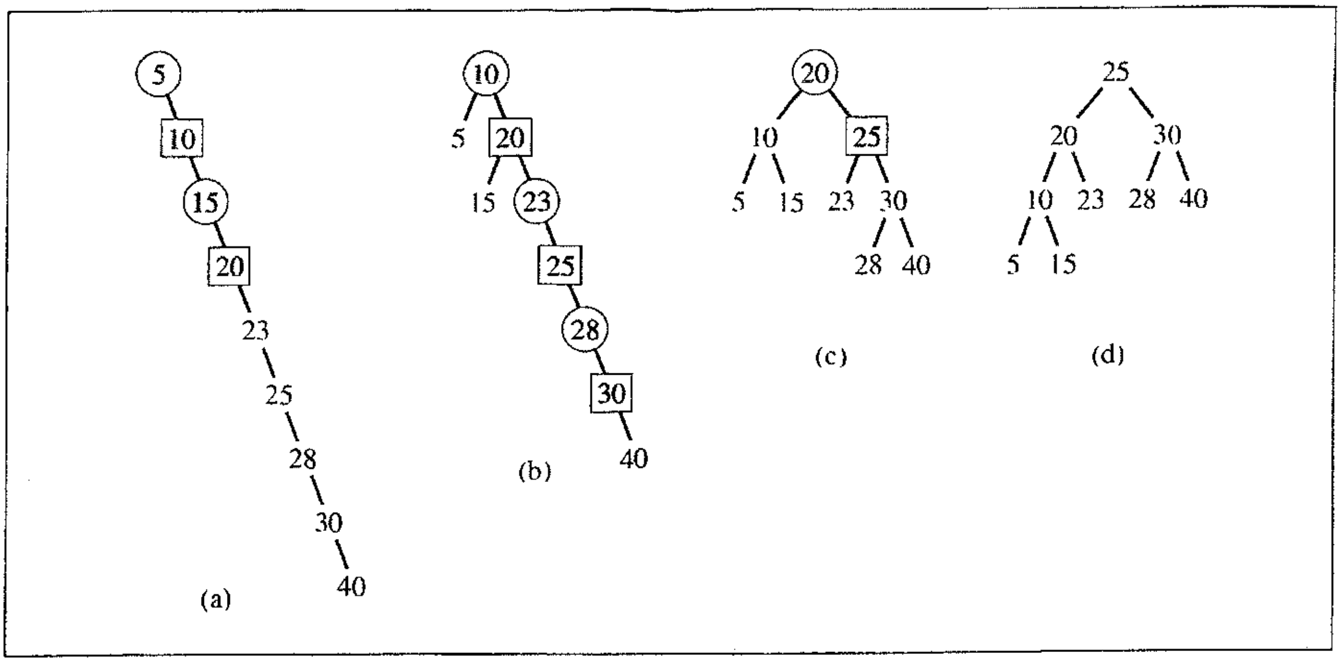
In the second phase, the backbone is transformed into a tree, but this time, the tree is perfectly balanced by having leaves only on two adjacent levels. In each pass down the backbone, every second node down to a certain point is rotated about its parent. The first pass is used to account for the difference between the number n of nodes in the current tree and the number $2^{\lfloor \lg(n+1) \rfloor} - 1$ of nodes in the closest complete binary tree where $\lfloor x \rfloor$ is the closest integer less than x. That is, the overflowing nodes are treated separately.

```
createPerfectTree(n)
  m = $2^{\lfloor lg(n+1) \rfloor}-1$;
  make n-m rotations starting from the top of backbone;
  while (m > 1)
    m = m/2;
    make m rotations starting from the top of backbone;
```

Figure 6.39 contains an example. The backbone in Figure 6.38e has nine nodes and is preprocessed by one pass outside the loop to be transformed into the backbone shown in Figure 6.39b. Now, two passes are executed. In each backbone, the nodes to be promoted by one level by left rotations are shown as squares; their parents, about which they are rotated, are circles.

To compute the complexity of the tree building phase, observe that the number of iterations performed by the `while` loop equals

FIGURE 6.39 Transforming a backbone into a perfectly balanced tree.



$$(2^{\lg(m+1)-1}-1)+\cdots+15+7+3+1=\sum_{i=1}^{\lg(m+1)-1}(2^i-1)=m-\lg(m+1)$$

The number of rotations can now be given by the formula

$$n-m+(m-\lg(m+1))=n-\lg(m+1)=n-\lfloor\lg(n+1)\rfloor$$

that is, the number of rotations is $O(n)$. Because creating a backbone also required at most $O(n)$ rotations, the cost of global rebalancing with the DSW algorithm is optimal in terms of time because it grows linearly with n and requires a very small and fixed amount of storage.

6.7.2 AVL Trees

The previous two sections discussed algorithms which rebalanced the tree globally; each and every node could have been involved in rebalancing either by moving data from nodes or by reassigning new values to pointers. Tree rebalancing, however, can be performed locally if only a portion of the tree is affected when changes are required after an element is inserted into or deleted from the tree. One classical method has been proposed by Adel'son-Vel'skii and Landis, which is commemorated in the name of the tree modified with this method: the AVL tree.

An *AVL tree* (originally called an *admissible tree*) is one in which the height of left and right subtrees of every node differ by at most one. For example, all the trees in Figure 6.40 are AVL trees. Numbers in the nodes indicate the *balance factors* which are the differences between the heights of the left and right subtrees. A balance factor is