6.3 Balanced Search Trees

In Sections 1.4, 4.5, and 5.3, we discussed the binary search tree—one of the principal data structures for implementing dictionaries. It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it. Note that this transformation from a set to a binary search tree is an example of the representation-change technique. What do we gain by such transformation compared to the straightforward implementation of a dictionary by, say, an array? We gain in the time efficiency of searching, insertion, and deletion, which are all in $\Theta(\log n)$, but only in the average case. In the worst case, these operations are in $\Theta(n)$ because the tree can degenerate into a severely unbalanced one with its height equal to $n - 1$.

Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree—principally, the logarithmic efficiency of the dictionary operations and having the set's elements sorted—but avoids its worst-case degeneracy. They have come up with two approaches.

- The first approach is of the instance-simplification variety: an unbalanced binary search tree is transformed into a balanced one. Because of this, such trees are called ***self-balancing***. Specific implementations of this idea differ by their definition of balance. An ***AVL tree*** requires the difference between the heights of the left and right subtrees of every node never exceed 1. A ***red-black tree*** tolerates the height of one subtree being twice as large as the other subtree of the same node. If an insertion or deletion of a new node creates a tree with a violated balance requirement, the tree is restructured by one of a family of special transformations called ***rotations*** that restore the balance required. In this section, we will discuss only AVL trees. Information about other types of binary search trees that utilize the idea of rebalancing via rotations, including red-black trees and ***splay trees***, can be found in the references [Cor09], [Sed02], and [Tar83].
- The second approach is of the representation-change variety: allow more than one element in a node of a search tree. Specific cases of such trees are ***2-3 trees***, ***2-3-4 trees***, and more general and important ***B-trees***. They differ in the number of elements admissible in a single node of a search tree, but all are perfectly balanced. We discuss the simplest case of such trees, the 2-3 tree, in this section, leaving the discussion of *B*-trees for Chapter 7.

AVL Trees

AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis [Ade62], after whom this data structure is named.

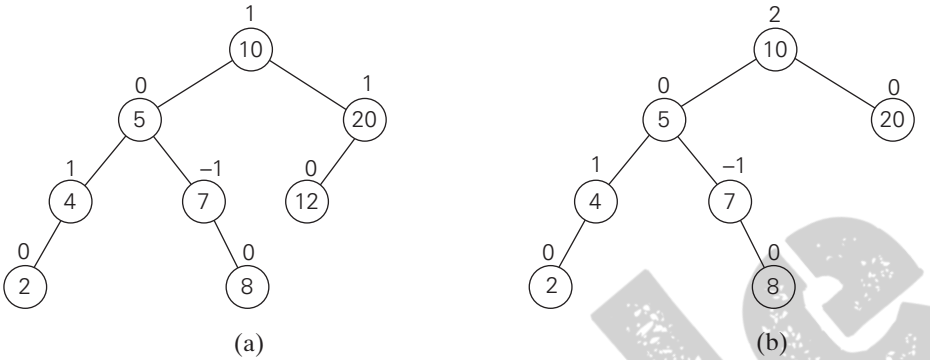


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

DEFINITION An ***AVL tree*** is a binary search tree in which the ***balance factor*** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1. (The height of the empty tree is defined as −1. Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

For example, the binary search tree in Figure 6.2a is an AVL tree but the one in Figure 6.2b is not.

If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A ***rotation*** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or −2. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf. There are only four types of rotations; in fact, two of them are mirror images of the other two. In their simplest form, the four rotations are shown in Figure 6.3.

The first rotation type is called the ***single right rotation***, or ***R-rotation***. (Imagine rotating the edge connecting the root and its left child in the binary tree in Figure 6.3a to the right.) Figure 6.4 presents the single *R*-rotation in its most general form. Note that this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The symmetric ***single left rotation***, or ***L-rotation***, is the mirror image of the single *R*-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion. (You are asked to draw a diagram of the general case of the single *L*-rotation in the exercises.)

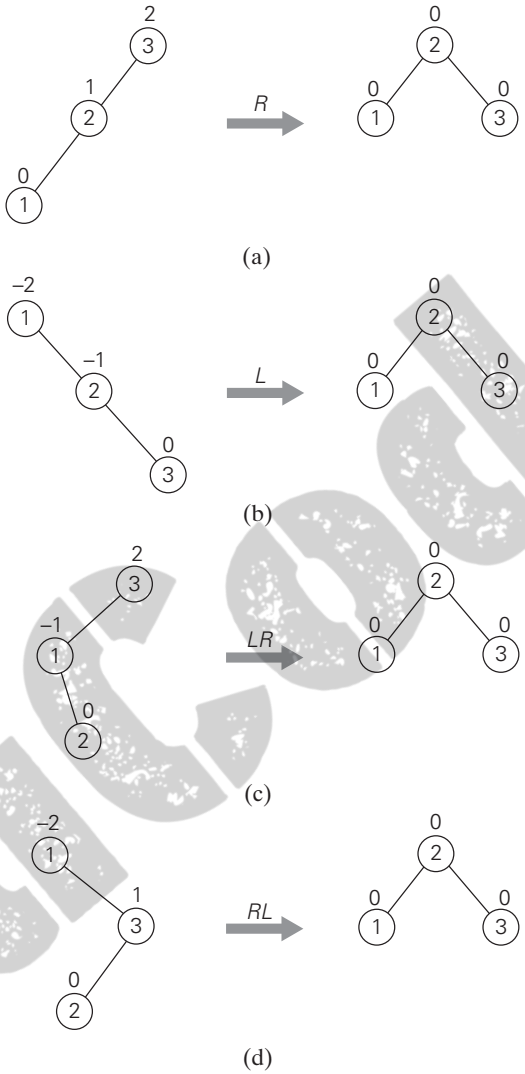


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

The second rotation type is called the ***double left-right rotation*** (***LR-rotation***). It is, in fact, a combination of two rotations: we perform the *L*-rotation of the left subtree of root *r* followed by the *R*-rotation of the new tree rooted at *r* (Figure 6.5). It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

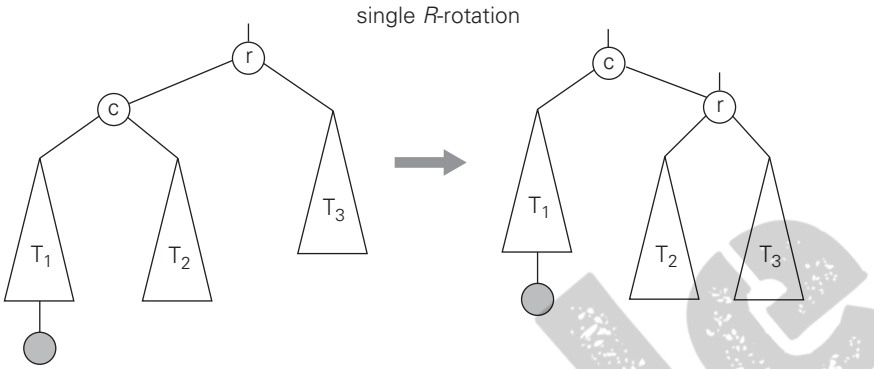


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

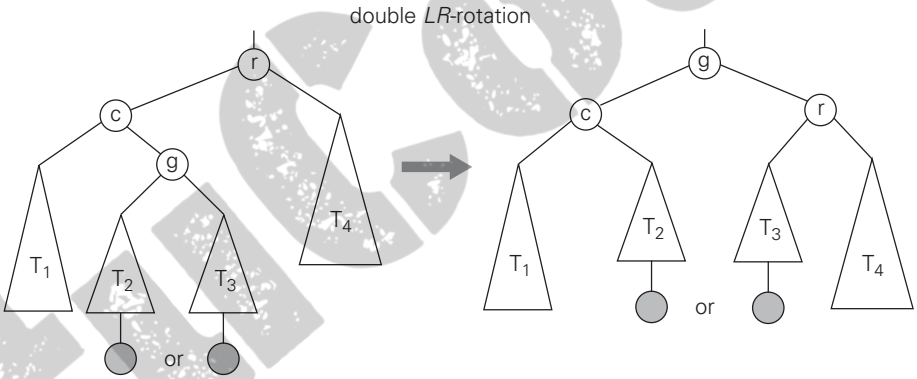


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

The ***double right-left rotation* (*RL-rotation*)** is the mirror image of the double *LR*-rotation and is left for the exercises.

Note that the rotations are not trivial transformations, though fortunately they can be done in constant time. Not only should they guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree. For example, in the initial tree of Figure 6.4, all the keys of subtree T_1 are smaller than c, which is smaller than all the keys of subtree T_2, which are smaller than r, which is smaller than all the keys of subtree T_3. And the same relationships among the key values hold, as they must, for the balanced tree after the rotation.

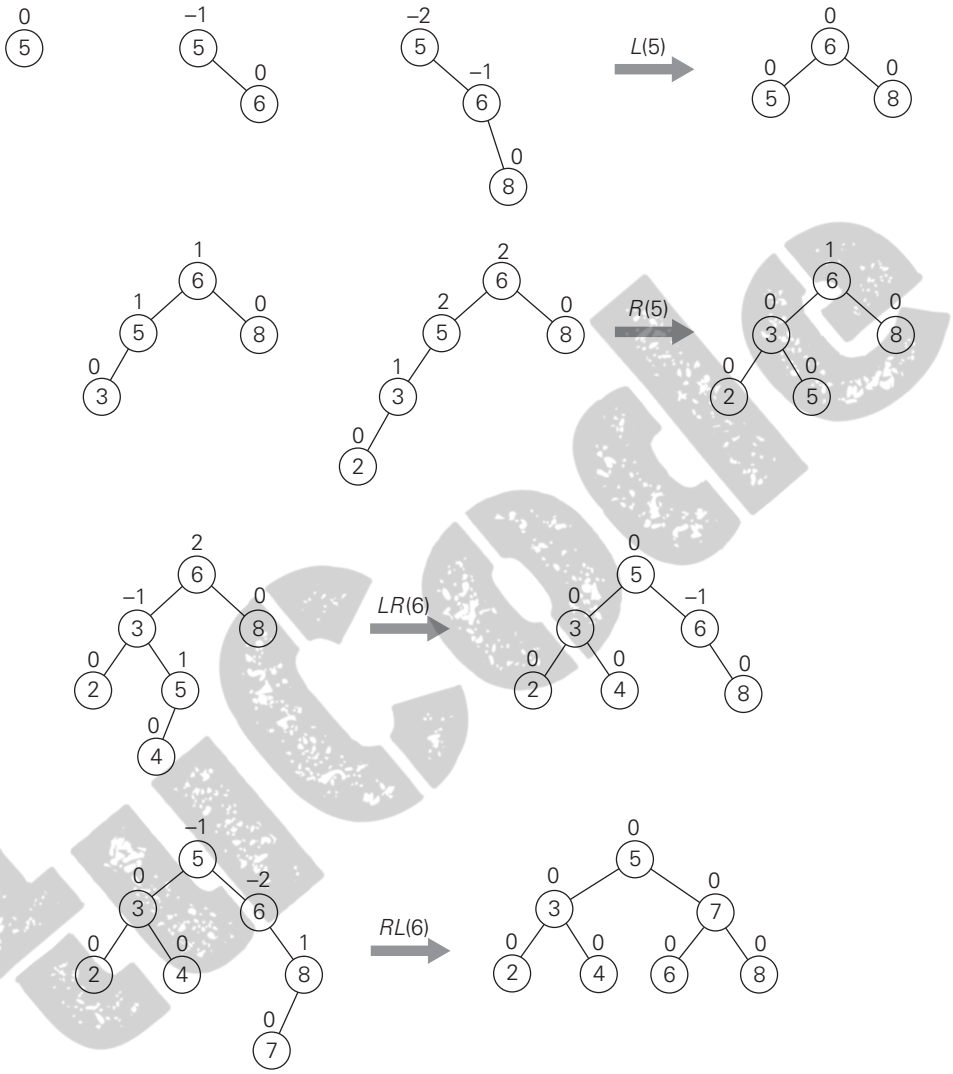


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

An example of constructing an AVL tree for a given list of numbers is shown in Figure 6.6. As you trace the algorithm's operations, keep in mind that if there are several nodes with the ± 2 balance, the rotation is done for the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

How efficient are AVL trees? As with any search tree, the critical characteristic is the tree's height. It turns out that it is bounded both above and below

by logarithmic functions. Specifically, the height h of any AVL tree with n nodes satisfies the inequalities

$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2(n+2) - 1.3277.$$

(These weird-looking constants are round-offs of some irrational numbers related to Fibonacci numbers and the golden ratio—see Section 2.5.)

The inequalities immediately imply that the operations of searching and insertion are $\Theta(\log n)$ in the worst case. Getting an exact formula for the average height of an AVL tree constructed for random lists of keys has proved to be difficult, but it is known from extensive experiments that it is about $1.01 \log_2 n + 0.1$ except when n is small [KnuIII, p. 468]. Thus, searching in an AVL tree requires, on average, almost the same number of comparisons as searching in a sorted array by binary search.

The operation of key deletion in an AVL tree is considerably more difficult than insertion, but fortunately it turns out to be in the same efficiency class as insertion, i.e., logarithmic.

These impressive efficiency characteristics come at a price, however. The drawbacks of AVL trees are frequent rotations and the need to maintain balances for its nodes. These drawbacks have prevented AVL trees from becoming the standard structure for implementing dictionaries. At the same time, their underlying idea—that of rebalancing a binary search tree via rotations—has proved to be very fruitful and has led to discoveries of other interesting variations of the classical binary search tree.

2-3 Trees

As mentioned at the beginning of this section, the second idea of balancing a search tree is to allow more than one key in the same node of such a tree. The simplest implementation of this idea is 2-3 trees, introduced by the U.S. computer scientist John Hopcroft in 1970. A ***2-3 tree*** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes. A ***2-node*** contains a single key K and has two children: the left child serves as the root of a subtree whose keys are less than K, and the right child serves as the root of a subtree whose keys are greater than K. (In other words, a 2-node is the same kind of node we have in the classical binary search tree.) A ***3-node*** contains two ordered keys K_1 and K_2 ($K_1 < K_2$) and has three children. The leftmost child serves as the root of a subtree with keys less than K_1, the middle child serves as the root of a subtree with keys between K_1 and K_2, and the rightmost child serves as the root of a subtree with keys greater than K_2 (Figure 6.7).

The last requirement of the 2-3 tree is that all its leaves must be on the same level. In other words, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf. It is this property that we "buy" by allowing more than one key in the same node of a search tree.

Searching for a given key K in a 2-3 tree is quite straightforward. We start at the root. If the root is a 2-node, we act as if it were a binary search tree: we either stop if K is equal to the root's key or continue the search in the left or right

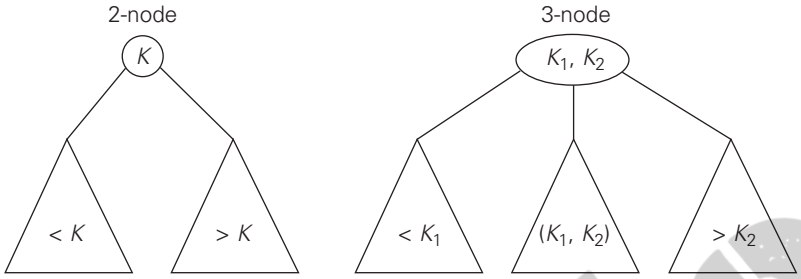


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

subtree if K is, respectively, smaller or larger than the root's key. If the root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if K is equal to one of the root's keys) or in which of the root's three subtrees it needs to be continued.

Inserting a new key in a 2-3 tree is done as follows. First of all, we always insert a new key K in a leaf, except for the empty tree. The appropriate leaf is found by performing a search for K. If the leaf in question is a 2-node, we insert K there as either the first or the second key, depending on whether K is smaller or larger than the node's old key. If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key.) Note that promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

An example of a 2-3 tree construction is given in Figure 6.8.

As for any search tree, the efficiency of the dictionary operations depends on the tree's height. So let us first find an upper bound for it. A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes (such as the final tree in Figure 6.8 for $h = 2$). Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

and hence

$$h \leq \log_2(n + 1) - 1.$$

On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

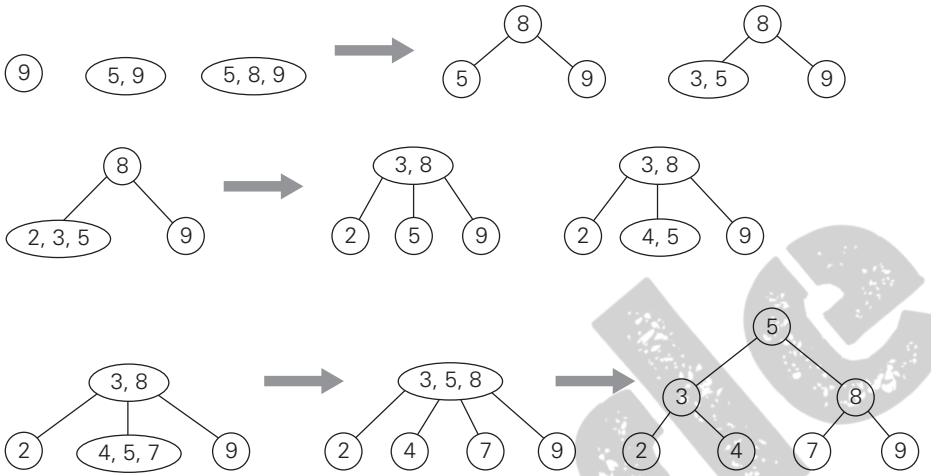


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

and hence

$$h \geq \log_3(n+1) - 1.$$

These lower and upper bounds on height h,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in $\Theta(\log n)$ in both the worst and average case. We consider a very important generalization of 2-3 trees, called B-trees, in Section 7.4.

6.4 Heaps and Heapsort

The data structure called the "heap" is definitely not a disordered pile of items as the word's definition in a standard dictionary might suggest. Rather, it is a clever, partially ordered data structure that is especially suitable for implementing priority queues. Recall that a ***priority queue*** is a multiset of items with an orderable characteristic called an item's ***priority***, with the following operations:

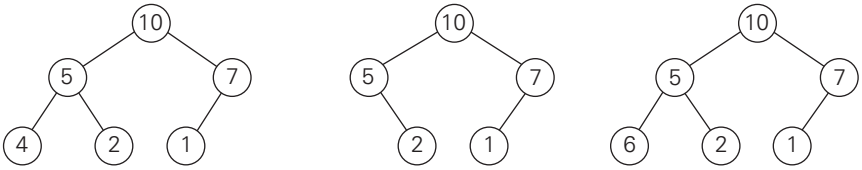


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

It is primarily an efficient implementation of these operations that makes the heap both interesting and useful. Priority queues arise naturally in such applications as scheduling job executions by computer operating systems and traffic management by communication networks. They also arise in several important algorithms, e.g., Prim's algorithm (Section 9.1), Dijkstra's algorithm (Section 9.3), Huffman encoding (Section 9.4), and branch-and-bound applications (Section 12.2). The heap is also the data structure that serves as a cornerstone of a theoretically important sorting algorithm called heapsort. We discuss this algorithm after we define the heap and investigate its basic properties.

Notion of the Heap

DEFINITION A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)[5]

For example, consider the trees of Figure 6.9. The first tree is a heap. The second one is not a heap, because the tree's shape property is violated. And the third one is not a heap, because the parental dominance fails for the node with key 5.

Note that key values in a heap are ordered top down; i.e., a sequence of values on any path from the root to a leaf is decreasing (nonincreasing, if equal keys are allowed). However, there is no left-to-right order in key values; i.e., there is no

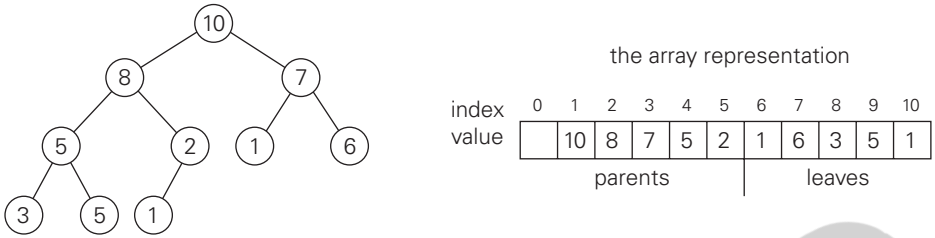


FIGURE 6.10 Heap and its array representation.

relationship among key values for nodes either on the same level of the tree or, more generally, in the left and right subtrees of the same node.

Here is a list of important properties of heaps, which are not difficult to prove (check these properties for the heap of Figure 6.10, as an example).

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor.$$

(Of course, if $2i + 1 > n$, just $H[i] \ge H[2i]$ needs to be satisfied.) While the ideas behind the majority of algorithms dealing with heaps are easier to understand if we think of heaps as binary trees, their actual implementations are usually much simpler and more efficient with arrays.

How can we construct a heap for a given list of keys? There are two principal alternatives for doing this. The first is the ***bottom-up heap construction*** algorithm illustrated in Figure 6.11. It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows. Starting with the last parental node, the algorithm checks whether the parental

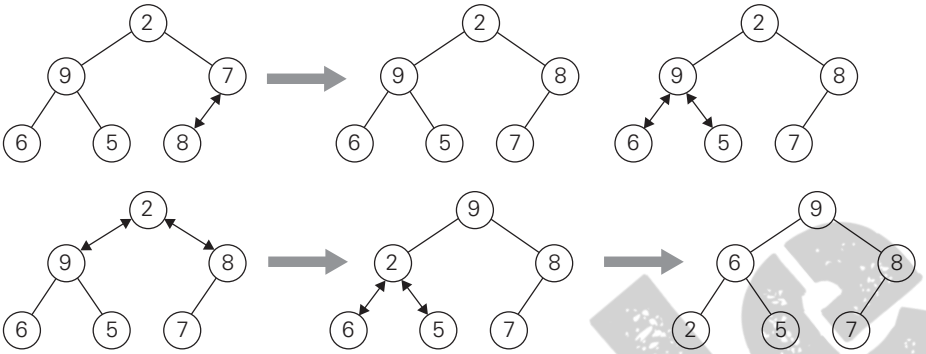


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

dominance holds for the key in this node. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it holds automatically for any key in a leaf.) After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the root of the tree.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of

nodes occurs on each level. Let h be the height of the tree. According to the first property of heaps in the list at the beginning of the section, $h = \lfloor \log_2 n \rfloor$ or just $\lceil \log_2 (n+1) \rceil - 1 = k - 1$ for the specific values of n we are considering. Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$. Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ (see Appendix A) or by mathematical induction on h. Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; some people call it the ***top-down heap construction*** algorithm. So how can we insert a new key K into a heap? First, attach a new node with key K in it after the last leaf of the existing heap. Then sift K up to its appropriate place in the new heap as follows. Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap); otherwise, swap these two keys and compare K with its new parent. This swapping continues until K is not greater than its last parent or it reaches the root (illustrated in Figure 6.12).

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log n)$.

How can we delete an item from a heap? We consider here only the most important case of deleting the root's key, leaving the question about deleting an arbitrary key in a heap for the exercises. (Authors of textbooks like to do such things to their readers, do they not?) Deleting the root's key from a heap can be done with the following algorithm, illustrated in Figure 6.13.

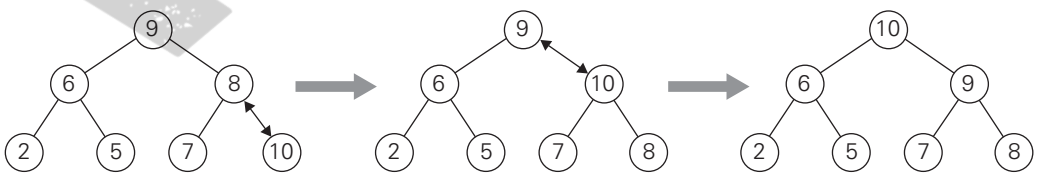


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

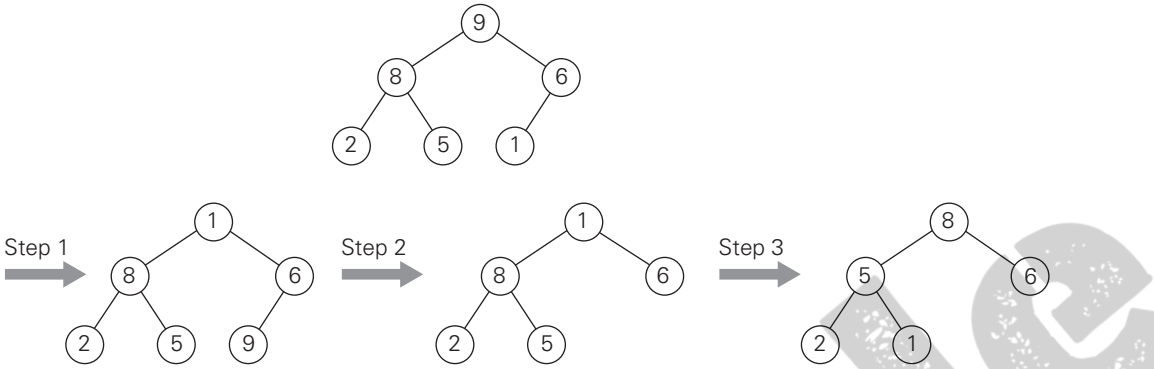


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well.

Heapsort

Now we can describe ***heapsort***—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order. Heapsort is traced on a specific input in Figure 6.14. (The same input as the one

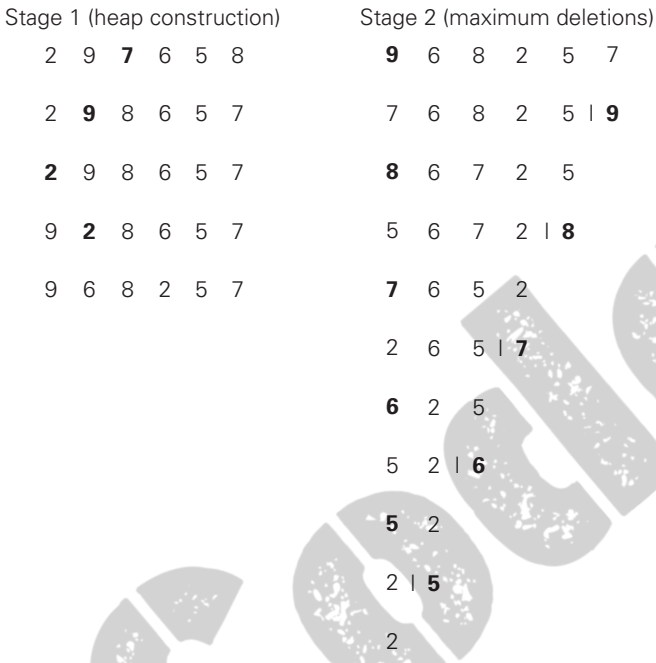


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

in Figure 6.11 is intentionally used so that you can compare the tree and array implementations of the bottom-up heap construction algorithm.)

Since we already know that the heap construction stage of the algorithm is in $O(n)$, we have to investigate just the time efficiency of the second stage. For the number of key comparisons, $C(n)$, needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

This means that $C(n) \in O(n \log n)$ for the second stage of heapsort. For both stages, we get $O(n) + O(n \log n) = O(n \log n)$. A more detailed analysis shows that the time efficiency of heapsort is, in fact, in $\Theta(n \log n)$ in both the worst and average cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort. Unlike the latter, heapsort is in-place, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

7.1 Sorting by Counting

As a first example of applying the input-enhancement technique, we discuss its application to the sorting problem. One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list. This algorithm is called ***comparison-counting sort*** (Figure 7.1).

Array $A[0..5]$		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass $i = 0$	*Count* []	3	0	1	1	0	0
After pass $i = 1$	*Count* []		1	2	2	0	1
After pass $i = 2$	*Count* []			4	3	0	1
After pass $i = 3$	*Count* []				5	0	1
After pass $i = 4$	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

FIGURE 7.1 Example of sorting by comparison counting.

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i] < A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Thus, the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space. On the positive side, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.

The counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values. Assume, for example, that we have to sort a list whose values can be either 1 or 2. Rather than applying a general sorting algorithm, we should be able to take advantage of this additional

information about values to be sorted. Indeed, we can scan the list to compute the number of 1's and the number of 2's in it and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2. More generally, if element values are integers between some lower bound l and upper bound u, we can compute the frequency of each of those values and store them in array $F[0..u-l]$. Then the first $F[0]$ positions in the sorted list must be filled with l, the next $F[1]$ positions with $l+1$, and so on. All this can be done, of course, only if we can overwrite the given elements.

Let us consider a more realistic situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements. Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows. The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$ elements of S, i.e., positions 0 through $F[0]-1$; the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$; and so on. Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as ***distribution counting***.

EXAMPLE Consider sorting the array

13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position $4-1=3$ of the array S that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure 7.2.

	$D[0..2]$			$S[0..5]$					
$A[5] = 12$	1	**4**	6				12		
$A[4] = 12$	1	**3**	6			12			
$A[3] = 13$	1	2	**6**						13
$A[2] = 12$	1	**2**	5		12				
$A[1] = 11$	**1**	1	5	11					
$A[0] = 13$	0	1	**5**					13	

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Here is pseudocode of this algorithm.

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array A. This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered. It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of inputs for which sorting by distribution counting works, in addition to trading space for time.

7.2 Input Enhancement in String Matching

In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching

requires finding an occurrence of a given string of m characters called the ***pattern*** in a longer string of n characters called the ***text***. We discussed the brute-force algorithm for this problem in Section 3.2: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the $O(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $O(n + m)$.

Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best-known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left. Since the latter idea leads to simpler algorithms, it is the only one that we will pursue here. (Note that the Boyer-Moore algorithm starts by aligning the pattern against the beginning characters of the text; if the first trial fails, it shifts the pattern to the right. It is comparisons within a trial that the algorithm does right to left, starting with the last character in the pattern.)

Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool [Hor80]. In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

$$\begin{array}{ccccccc} s_0 & \ldots & & & c & \ldots & s_{n-1} \\ & & \mathtt{B\ A\ R\ B\ E\ R} & & & & \end{array}$$

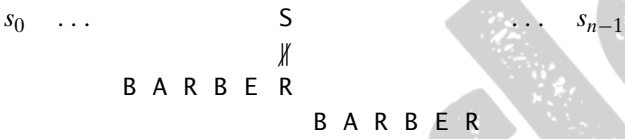
Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.

If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size

of such a shift by looking at the character c of the text that is aligned against the last character of the pattern. This is the case even if character c itself matches its counterpart in the pattern.

In general, the following four possibilities can occur.

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position

always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m-1$ times: for the jth character of the pattern ($0 \le j \le m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

ALGORITHM *ShiftTable*($P[0..m-1]$)

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//        filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

Now, we can summarize the algorithm as follows:

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then

stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is pseudocode of Horspool's algorithm.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])          //generate Table of shifts
i ← m − 1                        //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                        //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                 B A R B E R
```

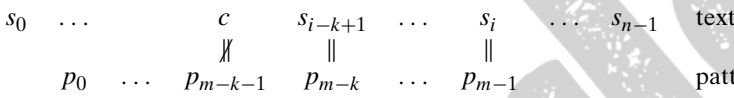
■

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in $O(nm)$ (Problem 4 in this section's exercises). But for random texts, it is in $\Theta(n)$, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.

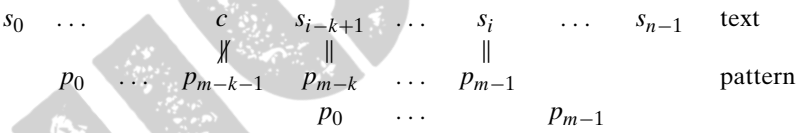
Boyer-Moore Algorithm

Now we outline the Boyer-Moore algorithm itself. If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool's algorithm. Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed as explained earlier.

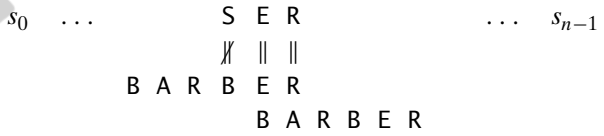
The two algorithms act differently, however, after some positive number k $(0 < k < m)$ of the pattern's characters are matched successfully before a mismatch is encountered:



In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities. The first one is guided by the text's character c that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad-symbol shift***. The reasoning behind this shift is the reasoning we used in Horspool's algorithm. If c is not in the pattern, we shift the pattern to just pass this c in the text. Conveniently, the size of this shift can be computed by the formula $t_1(c) - k$ where $t_1(c)$ is the entry in the precomputed table used by Horspool's algorithm (see above) and k is the number of matched characters:



For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:



The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\mathsf{A}) - 2 = 4 - 2 = 2$ positions:

If $t_1(c) - k \leq 0$, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.

To summarize, the bad-symbol shift d_1 is computed by the Boyer-Moore algorithm either as $t_1(c) - k$ if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

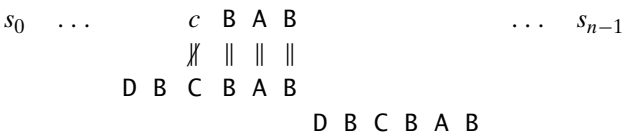
$$d_1 = \max\{t_1(c) - k, 1\}. \quad \textbf{(7.2)}$$

The second type of shift is guided by a successful match of the last $k > 0$ characters of the pattern. We refer to the ending portion of the pattern as its suffix of size k and denote it *suff*(k). Accordingly, we call this type of shift the ***good-suffix shift***. We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, \ldots, m-1$ to fill in the good-suffix shift table.

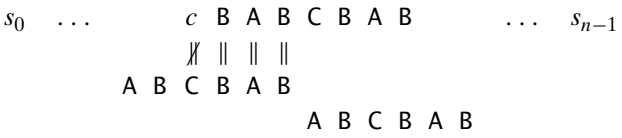
Let us first consider the case when there is another occurrence of *suff*(k) in the pattern or, to be more accurate, there is another occurrence of *suff*(k) not preceded by the same character as in its rightmost occurrence. (It would be useless to shift the pattern to match another occurrence of *suff*(k) preceded by the same character because this would simply repeat a failed trial.) In this case, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of *suff*(k) and its rightmost occurrence. For example, for the pattern ABCBAB, these distances for $k = 1$ and 2 will be 2 and 4, respectively:

k	**pattern**	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4

What is to be done if there is no other occurrence of *suff*(k) not preceded by the same character as in its rightmost occurrence? In most cases, we can shift the pattern by its entire length m. For example, for the pattern DBCBAB and $k = 3$, we can shift the pattern by its entire length of 6 characters:



Unfortunately, shifting the pattern by its entire length when there is no other occurrence of *suff*(k) not preceded by the same character as in its rightmost occurrence is not always correct. For example, for the pattern ABCBAB and $k = 3$, shifting by 6 could miss a matching substring that starts with the text's AB aligned with the last two characters of the pattern:



Note that the shift by 6 is correct for the pattern DBCBAB but not for ABCBAB, because the latter pattern has the same substring AB as its prefix (beginning part of the pattern) and as its suffix (ending part of the pattern). To avoid such an erroneous shift based on a suffix of size k, for which there is no other occurrence in the pattern not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length m. As an example, here is the complete list of the d_2 values—the good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB:

k	**pattern**	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Now we are prepared to summarize the Boyer-Moore algorithm in its entirety.

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases} \tag{7.3}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Shifting by the maximum of the two available shifts when $k > 0$ is quite logical. The two shifts are based on the observations—the first one about a text's mismatched character, and the second one about a matched group of the pattern's rightmost characters—that imply that shifting by less than d_1 and d_2 characters, respectively, cannot lead to aligning the pattern with a matching substring in the text. Since we are interested in shifting the pattern as far as possible without missing a possible matching substring, we take the maximum of these two numbers.

EXAMPLE As a complete example, let us consider searching for the pattern BAOBAB in a text made of English letters and spaces. The bad-symbol table looks as follows:

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

The good-suffix table is filled as follows:

k	**pattern**	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{B}}$AOB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

The actual search for this pattern in the text given in Figure 7.3 proceeds as follows. After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves $t_1(\text{K}) = 6$ from the bad-symbol table and shifts the pattern by $d_1 = \max\{t_1(\text{K}) - 0, 1\} = 6$ positions to the right. The new try successfully matches two pairs of characters. After the failure of the third comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 5$ from the good-suffix table to shift the pattern by $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$. Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.

The next try successfully matches just one pair of B's. After the failure of the next comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 2$ from the good-suffix table to shift

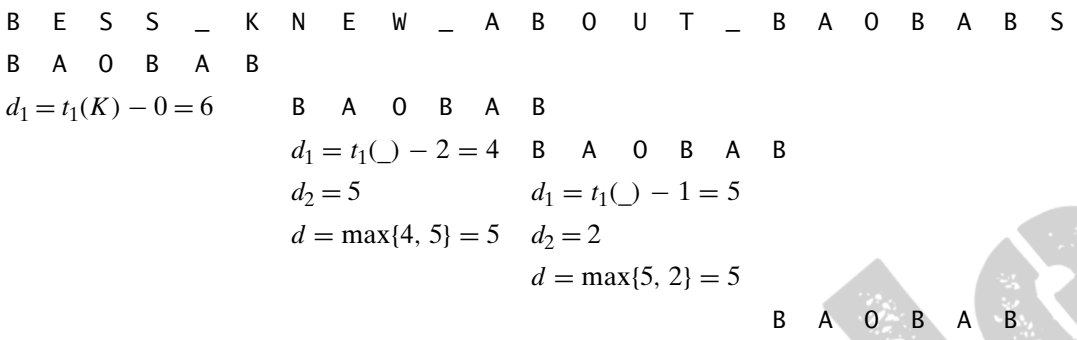


FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm.

the pattern by $\max\{d_1, d_2\} = \max\{6 - 1, 2\} = 5$. Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text. ■

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear. Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern), many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.