MODULE-3

Module-3

TRANSFORM-AND-CONQUER: Balanced Search Trees, Heaps and Heapsort.

SPACE-TIME TRADEOFFS: Sorting by Counting: Comparison counting sort, Input Enhancement in String Matching: Horspool's Algorithm.

Chapter 6 (Sections 6.3,6.4), Chapter 7 (Sections 7.1,7.2)

Balanced Search Trees

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.

AVL Trees

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

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.)

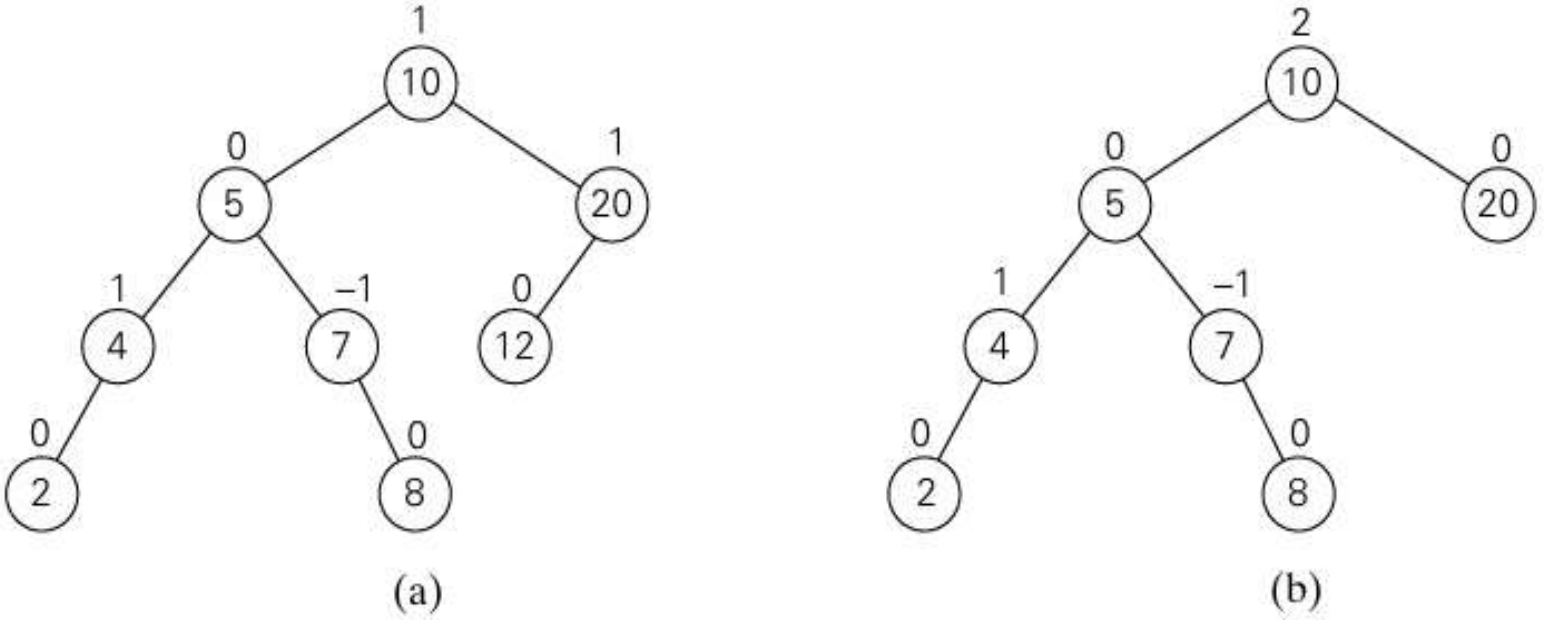


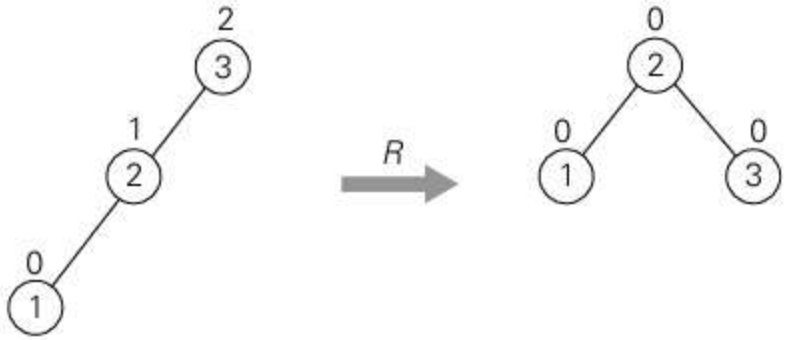
FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

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 Figure6.3.

1.Single right rotation,or R-rotation

✓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 at the right.) Figure6.4 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.



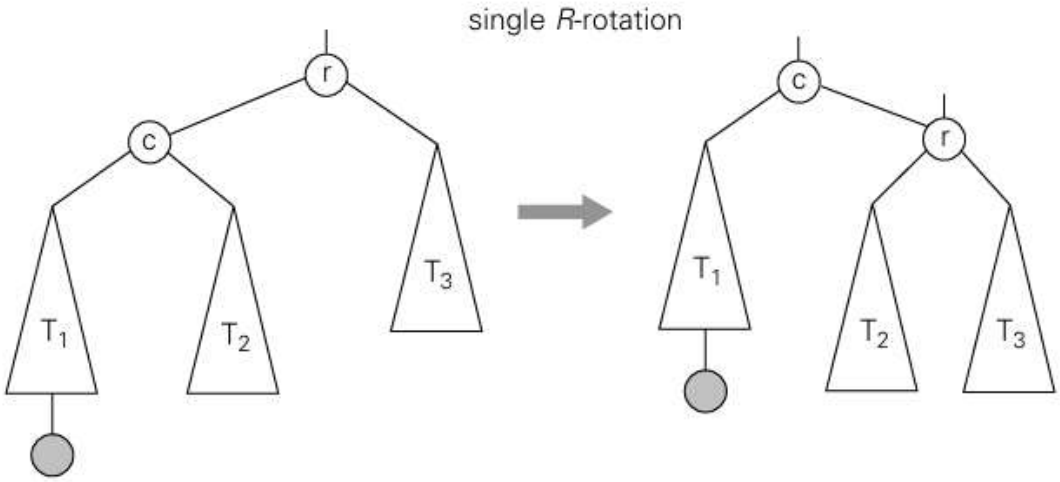
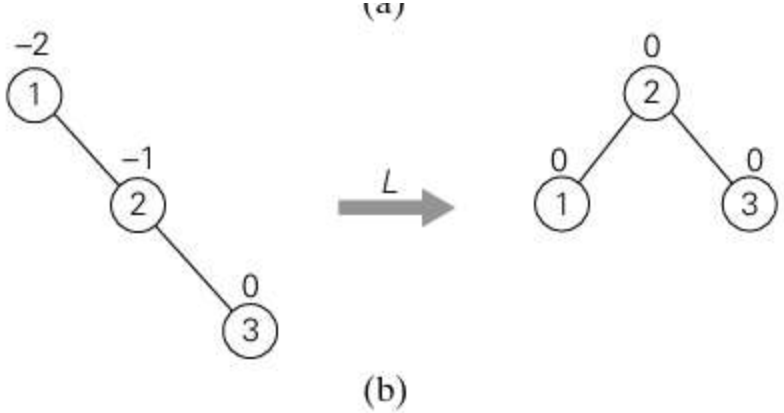


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

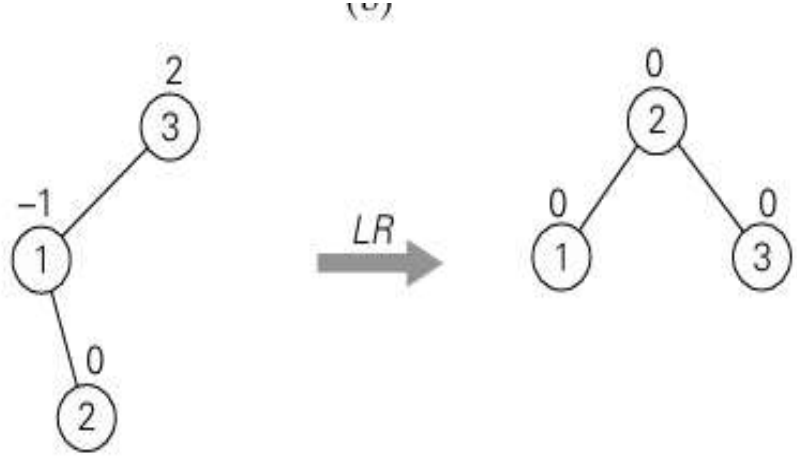
2.Single left rotation, or L-rotation

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.



3.Double left-right rotation (LR 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. 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.



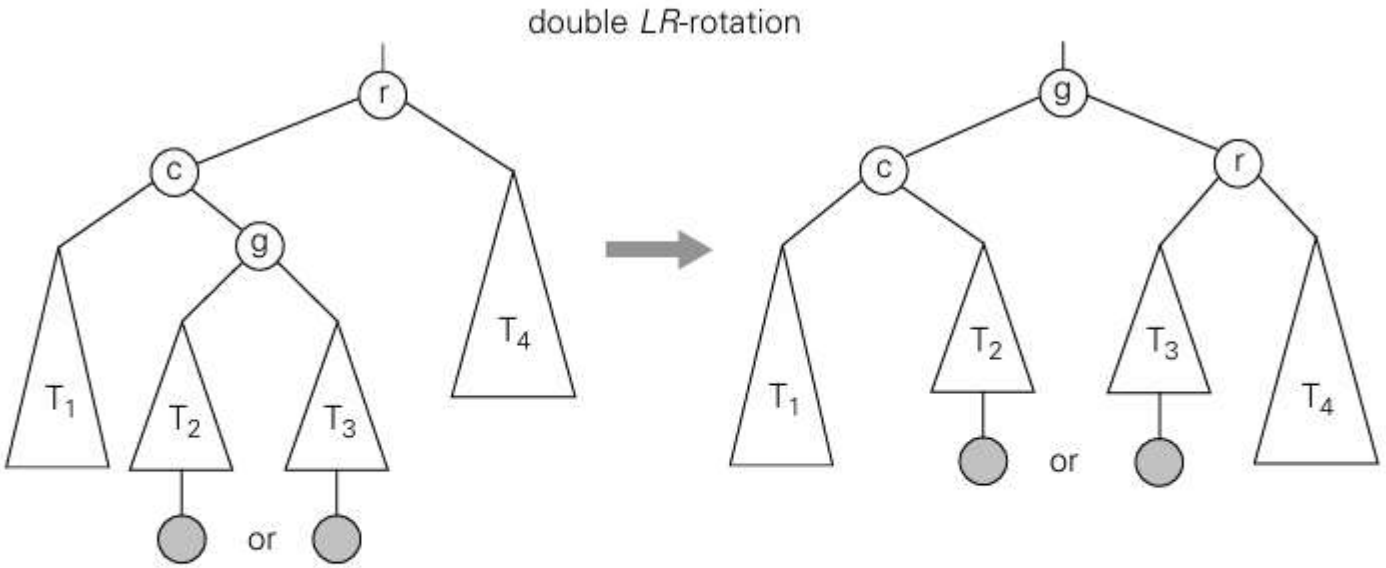


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

4.Double right-left rotation(RL-rotation)

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

2-3 Trees

✓ 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.

✓A 3-node contains two ordered keys K1 and K2 (K1<K2) and has three children. The leftmost child serves as the root of a subtree with keys less than K1, the middle child serves as the root of a subtree with keys between K1 and K2, and the rightmost child serves as the root of a subtree with keys greater than K2

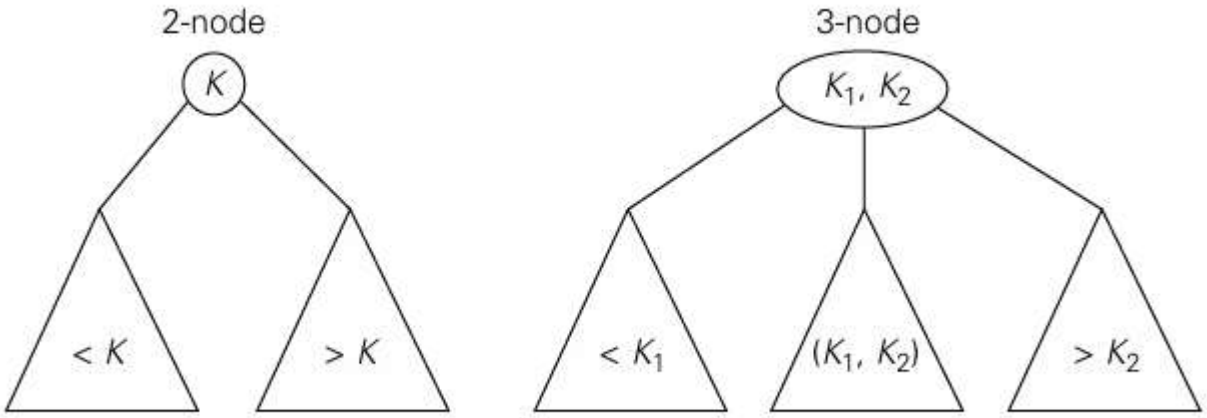


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

✓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

Properties of 2-3 tree

- Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
- Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children.
- Data is stored in sorted order.
- It is a balanced tree.
- All the leaf nodes are at same level.
- Each node can either be leaf, 2 node, or 3 node.
- Always insertion is done at leaf.

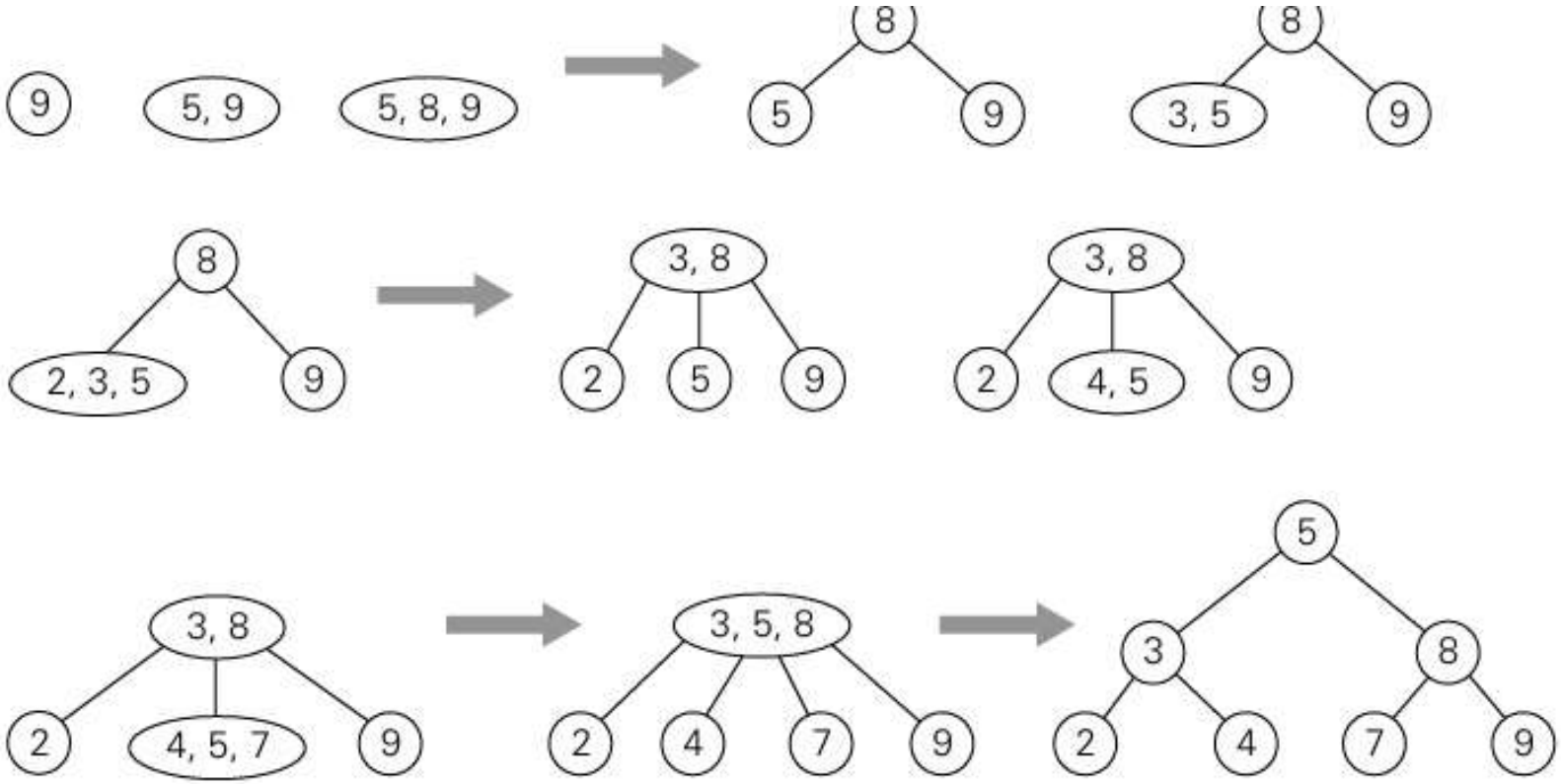


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

Heaps and Heapsort

The data structure called the “heap” is a clever, partially ordered data structure that is especially suitable for implementing priority queues.

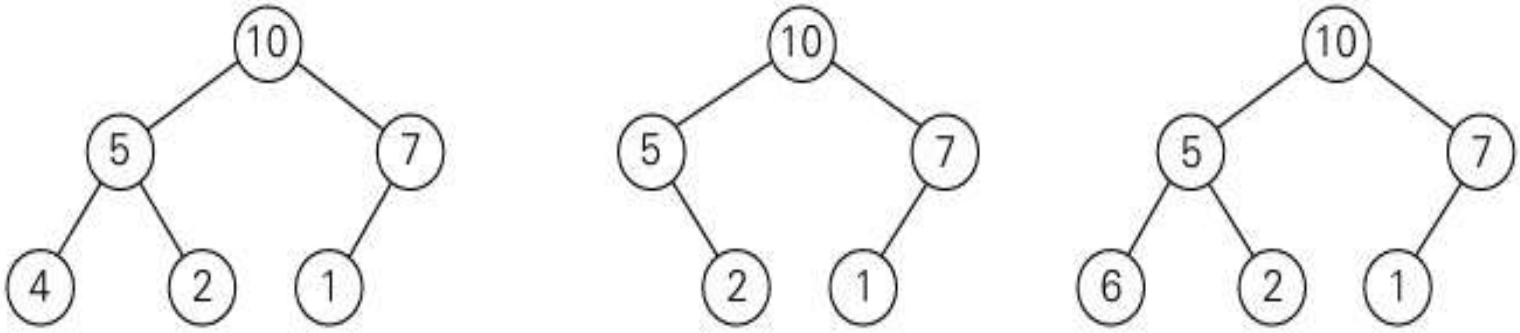


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

Applications:

✓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, Dijkstra's algorithm, Huffman encoding, and branch-and-bound applications.

Heaps and Heapsort

The data structure called the “heap” is a clever, partially ordered data structure that is especially suitable for implementing priority queues.

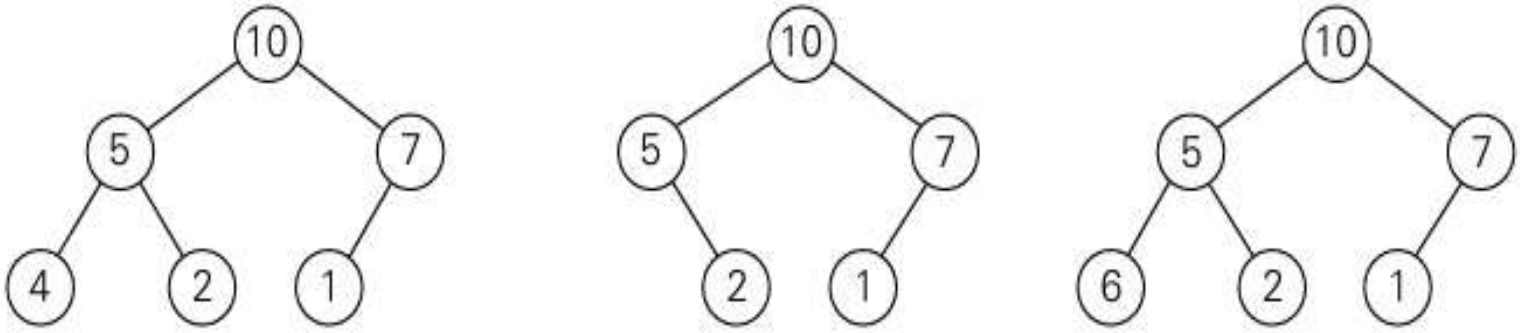


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

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 auto matically satisfied for all leaves.)

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

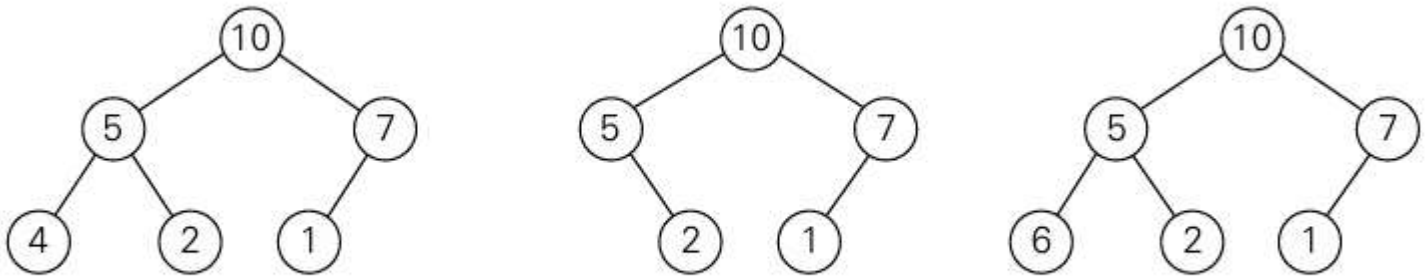


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

✓Note that key values in a heap are ordered topdown;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 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.

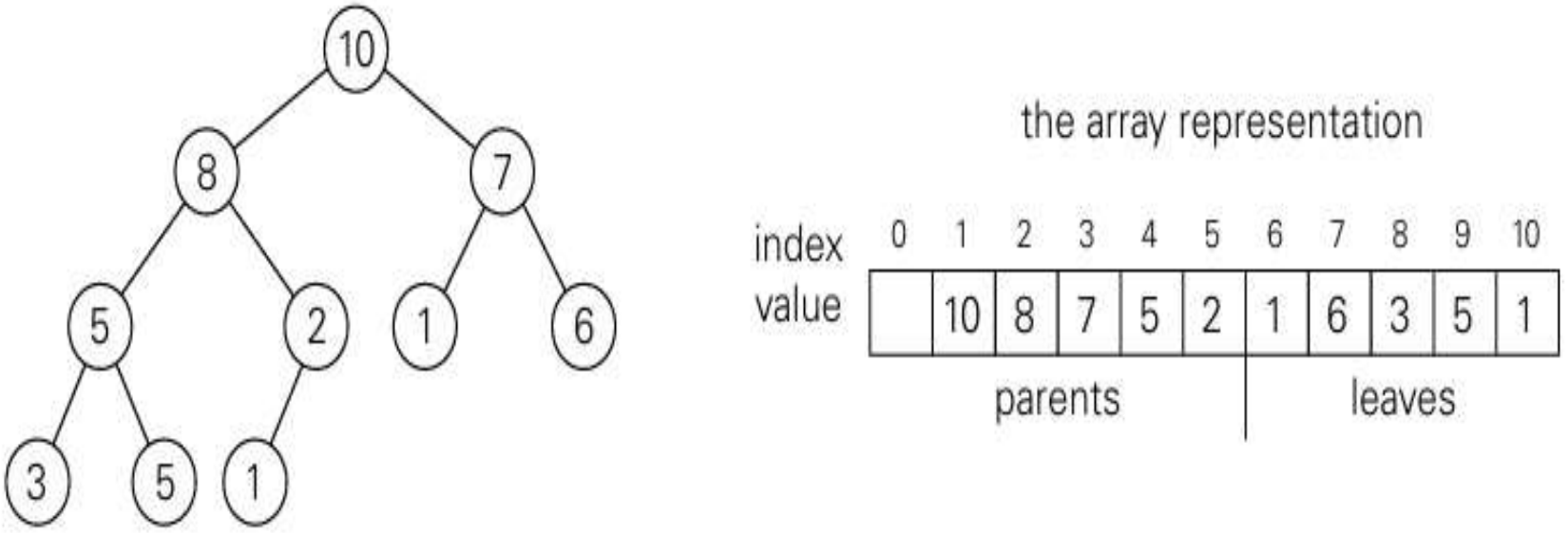


FIGURE 6.10 Heap and its array representation.

A list of important properties of heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\log_2 n$.
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 n/2 positions of the array, while the leaf keys will occupy the last n/2 positions;

b. The children of a key in the array's parental position i($1 \leq i \leq n/2$) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i($2 \leq i \leq n$) will be in position i/2 .

Constructing a heap for a given list of keys

There are two principal alternatives for doing this.

1.The first is the bottom-up heap construction algorithm

2.The first is the topdown-up heap construction algorithm

The bottom-up heap construction algorithm.

✓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 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.

✓ 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.

✓This process continues until the parental dominance for K is satisfied.

✓ 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.

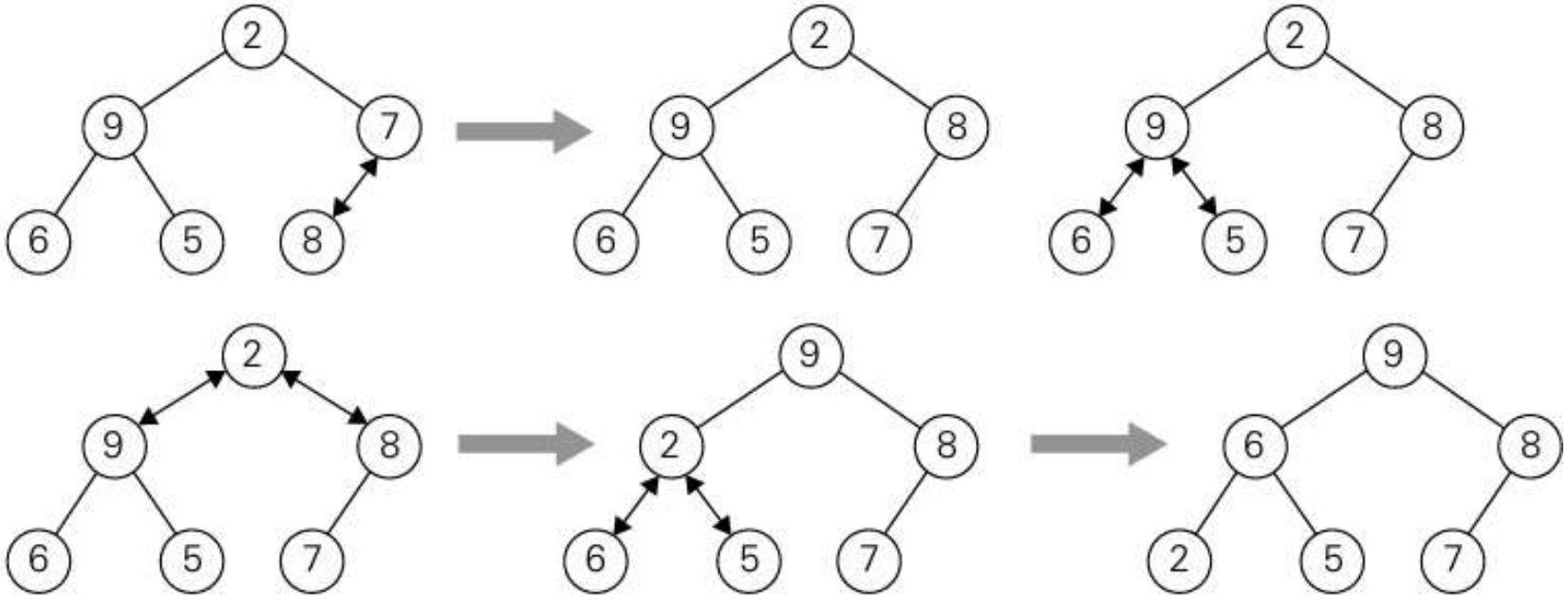


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

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
```

Top-down heap construction algorithm.

✓ 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 .

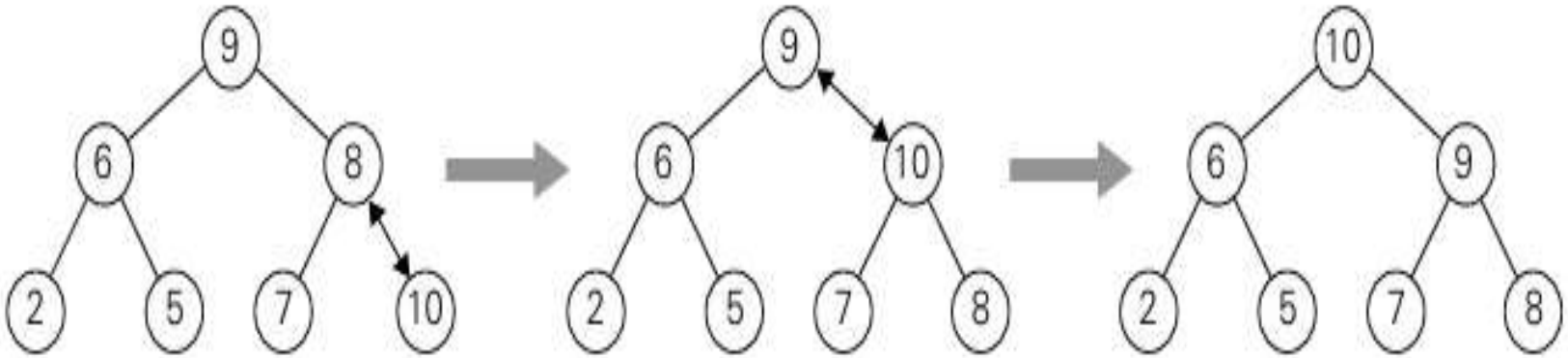


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Deleting the root's key from a heap can be done with the following algorithm

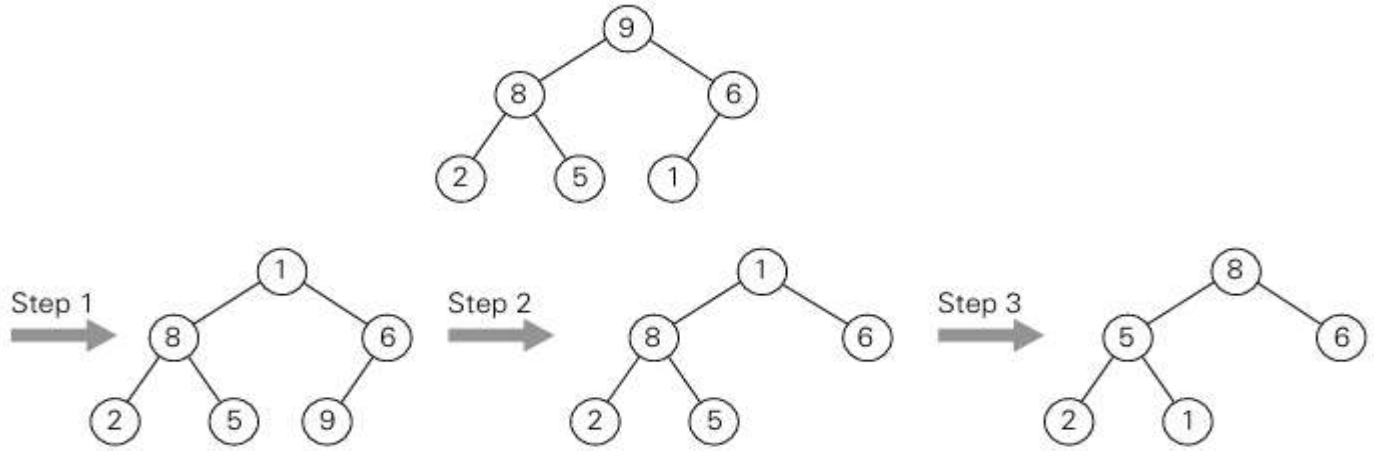


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

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.

Heapsort

Now we can describe heapsort—an interesting sorting algorithm discovered by J. W. J. Williams . 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.

SPACE-TIME TRADEOFFS:

Sorting by Counting: Comparison counting sort

✓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

Example:-

✓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

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
```

Input Enhancement in String Matching

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

$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

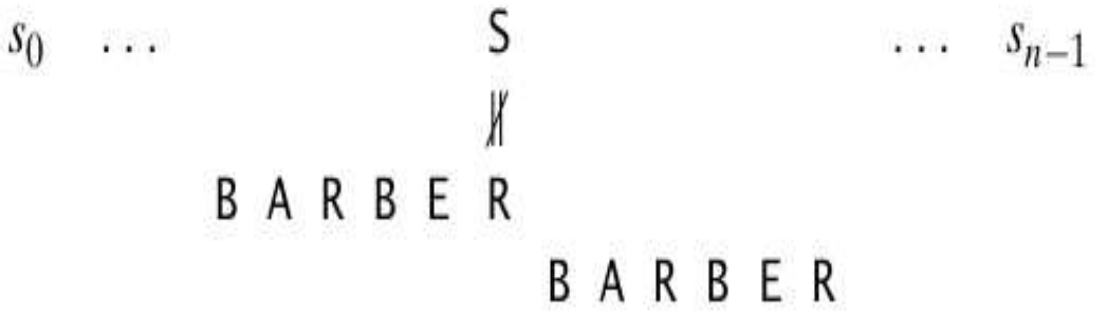
B A R B E R

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.

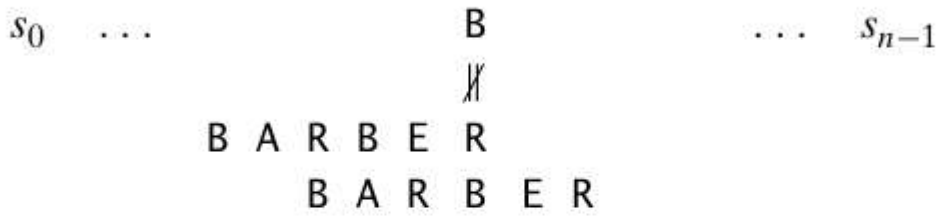
If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size of such a shift by looking at the character c of the text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

L E A D E R

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:

s_0 . . . A R . . . s_{n-1}

R E O R D E R

R E O R D E R

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

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 $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

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

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.