CES 512 Feb 6

Goals:

- Complete Chapter 3 (from Lecture 1)
- Heap (Chapter 7)
  - Definition
  - Algorithms for
    - *Insert*
    - *DeleteMin*
    - *Heap-sort*
    - *Build-heap*
- Some applications of heap

Summary of Lecture 1:

Overview of algorithm design process:

- numerical algorithms
- non-numerical algorithms

Insertion sorting: has a best case performance of O(n) and the worst-case performance of $O(n^2)$.

Example of a max-heap – its two representations

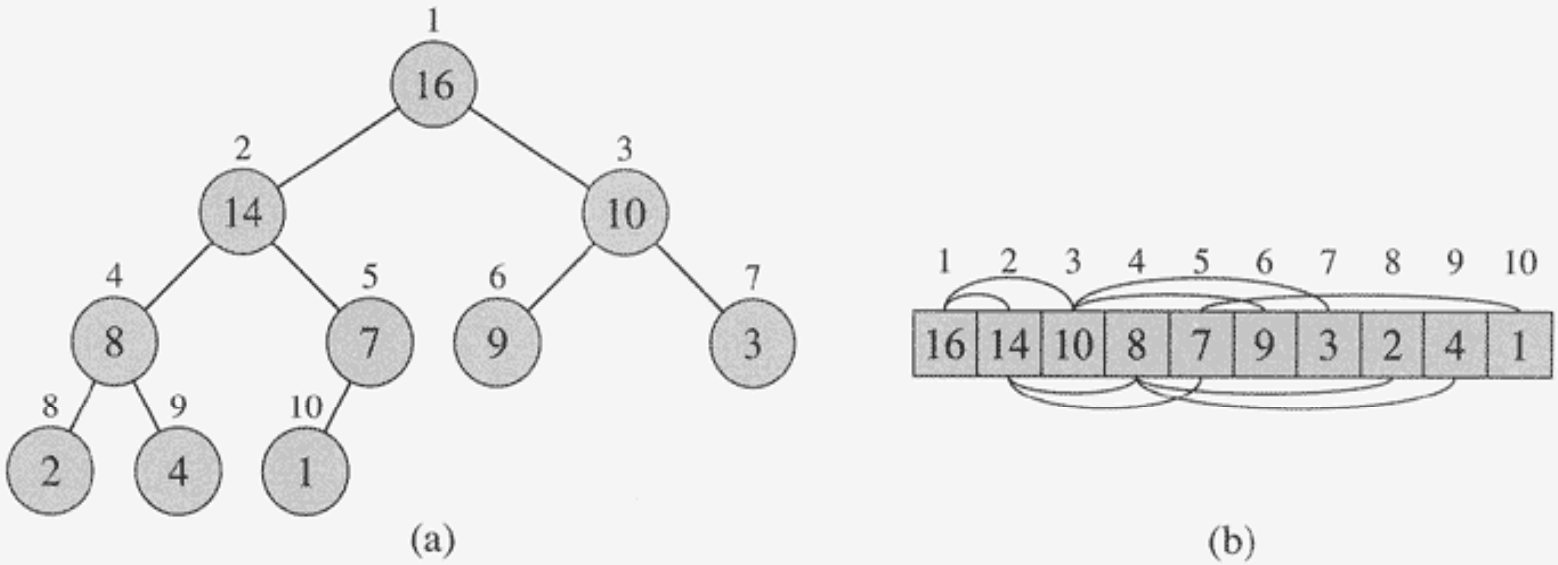
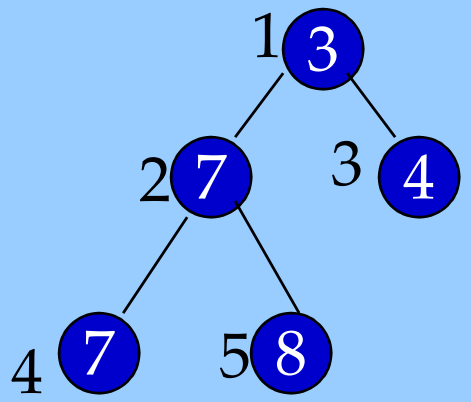


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Heap Sort: Based on a clever data structure called a heap.

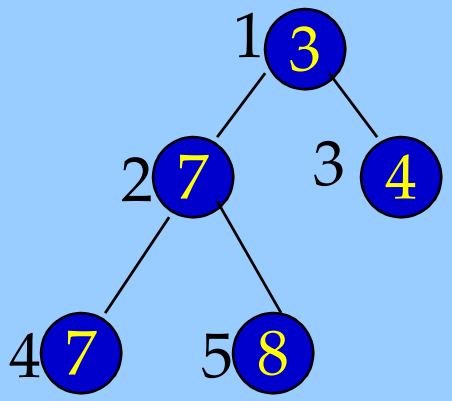
A heap is an array in which the keys will always be stored in index 1 to k for some k. Such an array is called a heap if when you view the heap as a full binary tree, the parent key is always <= the child key.



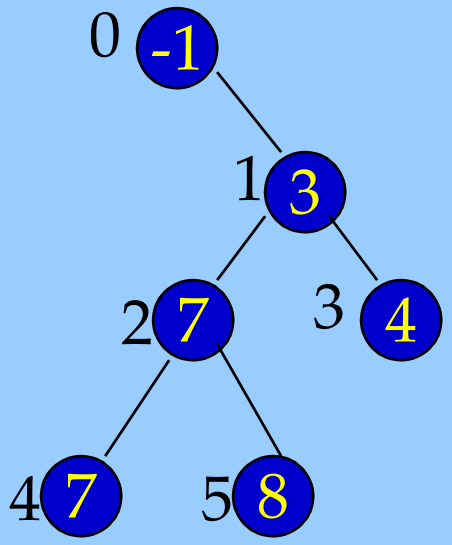
A	1	2	3	4	5
	3	7	4	7	8

```
class Heap
   {int[]
      HeapElements;
    int Size}
```

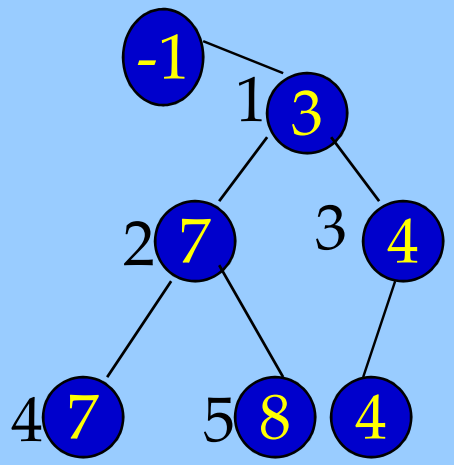
Insert 2 into the heap shown below:



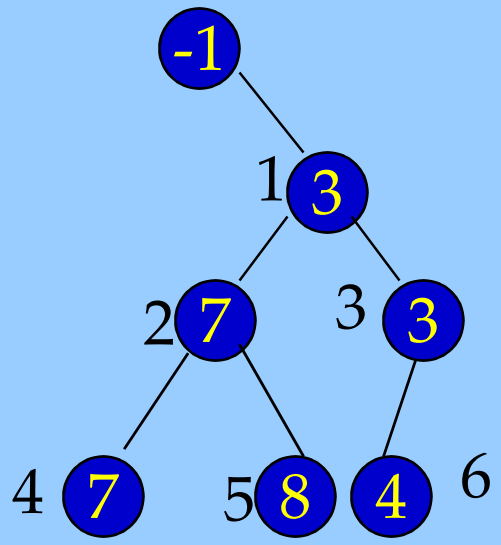
heap with sentinel looks as follows:



Size = 6, so initially, j= 6; Since H[j/2]= 4 > 2, H[3] is copied to H[6]. Thus the heap now looks as follows:



The new value of j=3. Since H[j/2] =H[1] = 3 is also greater than 2, H[1] copied to H[3]. Now the heap looks as in the next slide.

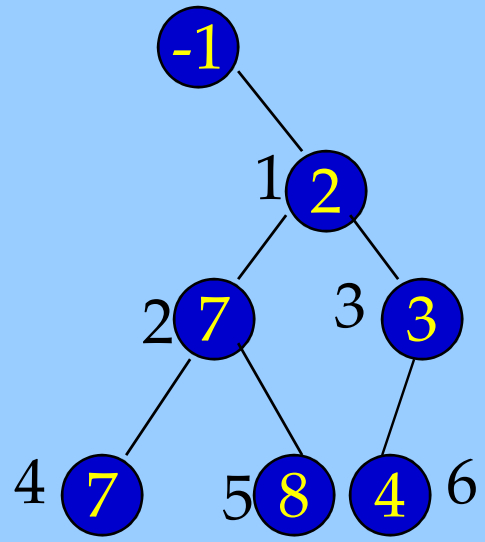


The new value of j = [3/2] = 1.

Now, H[j/2] = H[0] -1 < 2 so the iteration stops.

The last line sets H[j] = H[1] = 2.

The final heap looks as follows:



- An obvious but important fact:

  HeapElements[1] contains the smallest key.

Heap can support the following three operations very fast, i.e. in time *O (log n)* where *n* is the size of the heap.

- void Insert(int x)
- int DeleteMin( )

- Informal description of the algorithm for Insert(x):

  Find a place for x along the path from the leaf with *index* *Size*/2 and the root (*index* = *1*).

More formal description:

```
Size++;
for (int j=Size; HeapElements[j/2] > x;
          j/=2)
  HeapElements[j]=HeapElements[j/2];
HeapElements[j] = x;
```

- Although the code is extremely simple, it is tricky. Test it with some examples.

- What happens if the key inserted is the smallest?

- Another fundamental property of a heap:

  The length of the path from any leaf to the root is exactly [log (n+1)]. (Here height is defined as the number of nodes in any path from the root to a leaf.)

Conclusion: **`HeapInsert`** performs $O(\log n)$ basic operations (comparisons and data movements) to insert a key into a heap of size n in the worst-case.

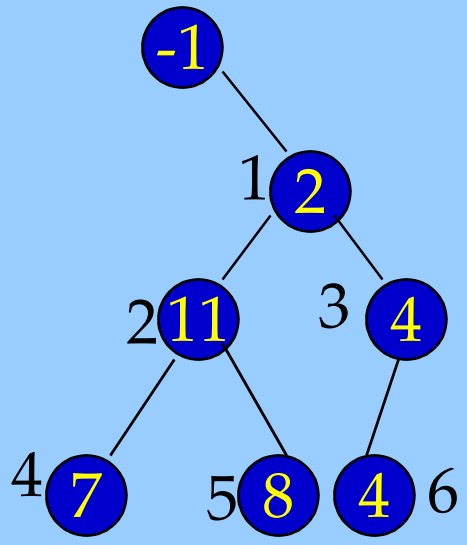
How to perform **`DeleteMin()?`**

- Remove the key stored in H[1] and store it in *Temp* (to be returned at the end of the procedure).

- Then, tentatively move the key in H[Size] into H[1] and call **`Heapify(1).`**

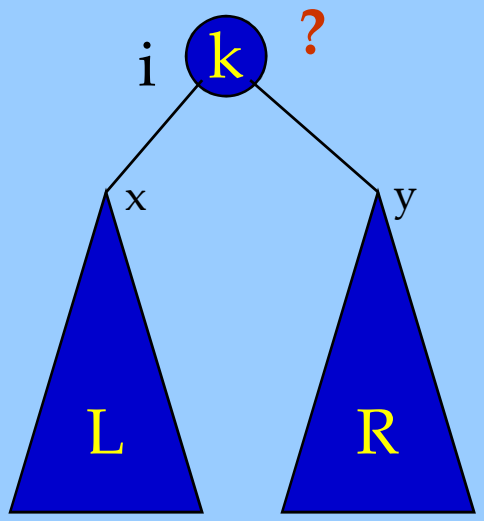
`void Heapify(i)` performs the following task:
Adjust(i) will make the tree rooted at i into a heap ***given that the subtrees rooted at 2i and 2i+1 are heaps.***

Example:



Here, the subtrees rooted at 4 and 5 are heaps, but the subtree rooted at 2 is not. A call `Heapify(2)` should make the subtree rooted at 2 into a heap.

Outline of Adjust(i) as a recursive method:

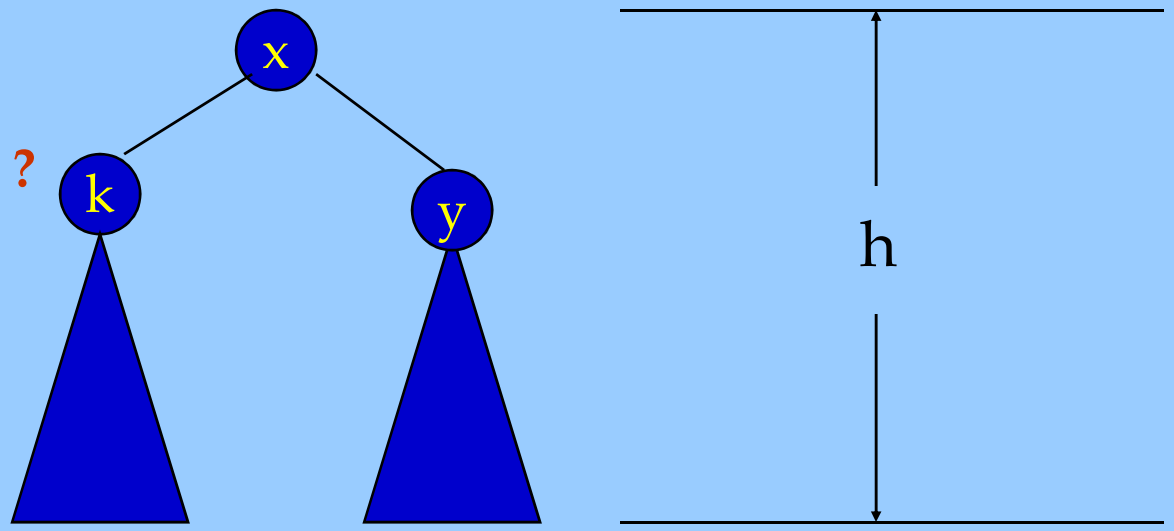


Note:
Root of L has index 2i and the root of R has index (2i+1).
Suppose H[2i] = x, H[2i+1]= y.

- Case 1: (k < x) && (k < y) In this case, the tree rooted at is a heap. So, the procedure terminates.

- Case 2: (k > x) && (y > x). Thus, x is the minimum of the three keys k, x and y. Suppose we swap the keys k and x. Now the problem reduces to Adjust(2i).

- Case 3: y = min {k, x, y}. In this case, swap k and y; Adjust(2i+1).



Key observation: In order to perform **`Heapify(i),`** we perform 2 comparisons at two comparisons at each level, and if the height of the tree is h, the total number of operations is at most 2h. Since h <= log (n+1), the total number of comparisons performed is O(log n).

Number of data movements is O(log n) as well (since at most 3 data movements per level).

Max-Heapify pseudo-code

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4      then largest ← l
 5      else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7      then largest ← r
 8  if largest ≠ i
 9      then exchange A[i] ↔ A[largest]
10           MAX-HEAPIFY(A, largest)
```

What we have seen thus far:

- *insert* into a heap.
- *Max-Heapify* to make the subtree rooted at i into a heap, given that subtrees rooted at 2*i and 2*i+1 are heaps.
- *deletemin* from a heap.
- number of key comparisons (worst-case)
  - *insert* : [log n]
  - *DeleteMin* : 2 *[log n]
- number of data movements (worst-case)
  - *insert*: [log n]
  - *DeleteMin:* [log n]

Heap Sorting

Input: Array A[1 .. n] of keys.

Output: The array A[1..n] with keys arranged in descending order.

1) Initialize the heap A[1]; let k = 2;

2) while (k <= n)

{A.insert(A[k]); ++k;} n x O (log n) = O (n log n)

3) k = n;

while (k >= 2)

A[k] = A.deletemin( ); n x O (log n) = O (n log n)

Build-heap

Given an array A of numbers (completely unordered). We want to convert A into a heap. We will need this at the start after we have read in the numbers from input etc.

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        do MAX-HEAPIFY(A, i)
```

Example

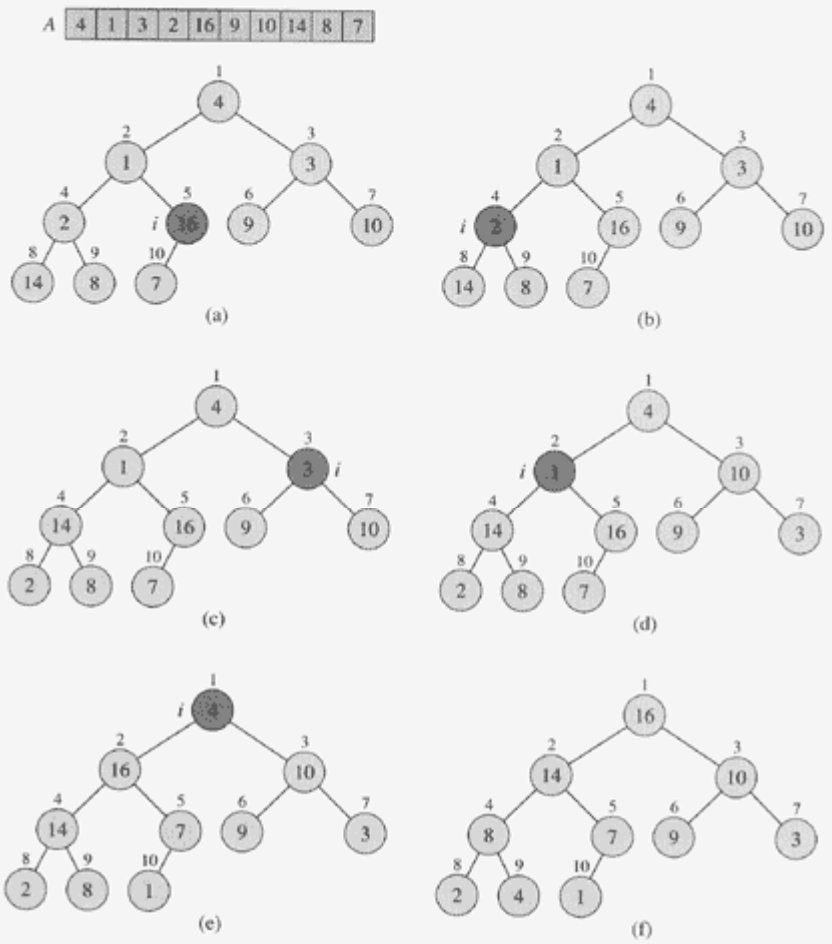


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

What is the time complexity of this algorithm?

Analysis

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)
- ***Tighter analysis:*** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, and most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

Some applications of priority queue

Huffman coding:

Problem: Given a text over a finite alphabet S, we want to encode each symbol using a string over {0,1} so that the total length of the encoded text is minimized?

Example: abracadabra S = {a, b, c, d, r}

```
a -> 01, b -> 001, c-> 110, d -> 101, r-> 111
```

Coded text:

```
01 001 110 01 110 01 101 01 001 111 01
```

Prefix code: No code word is a prefix of any other.

Block code vs. variable-length code

- block code is a prefix code. (Example: ASCII, unicode etc.)

- variable length code.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

Tree representation of the code

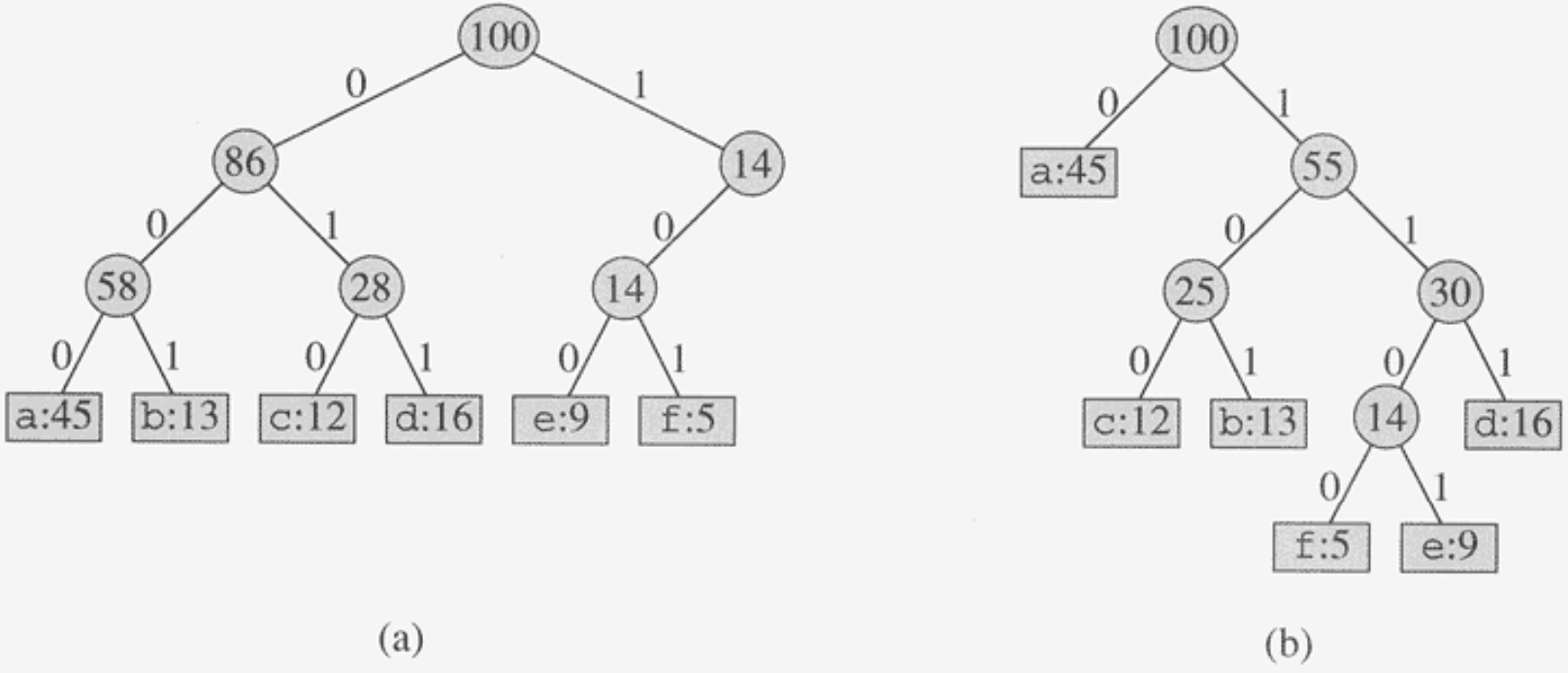


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . . . , f = 101. **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.

Huffman coding algorithm

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

Example

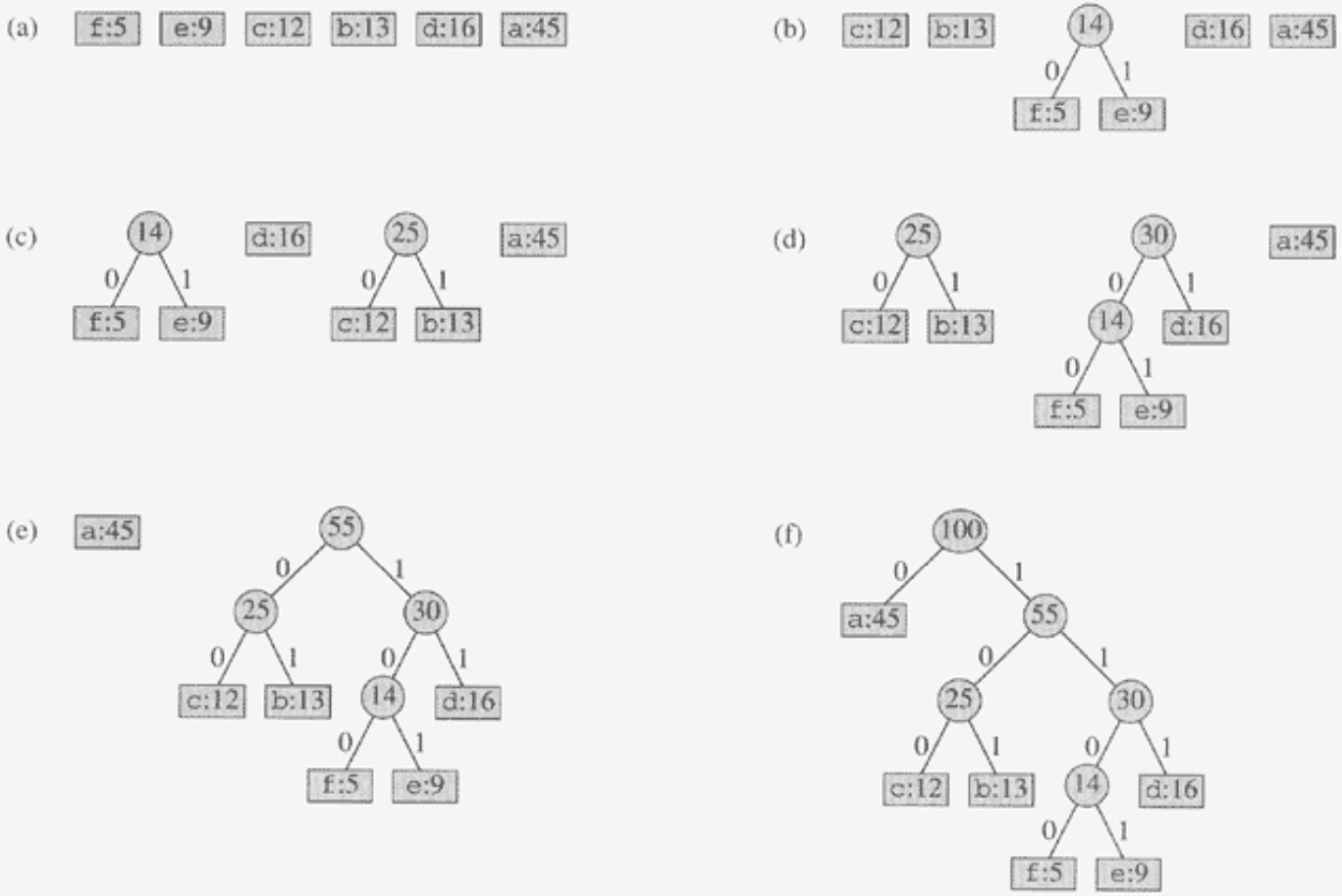


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.