# Huffman Coding

CLRS 16.3

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3. That is, only 6 different characters appear, and the character a occurs 45,000 times.

| | 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 |

# How to represent such a file of information?

- *A **binary character code*** (or ***code*** for short) in which each character is represented by a unique binary string, which we call a ***codeword***.
- A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.

| | 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 |

- If we use a ***fixed-length code***, we need 3 bits to represent 6 characters: a = 000, b = 001, . . . , f = 101. This method requires 300,000 bits to code the entire file. Can we do better?

- A ***variable-length code*** requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file.

We code the 3-character file abc as 0.101.100 = 0101100, where "." denotes concatenation.

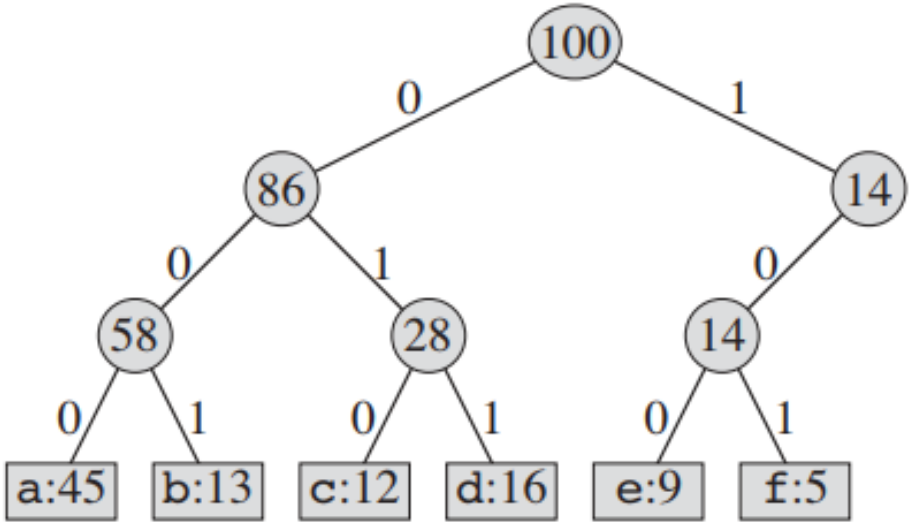
# Prefix codes

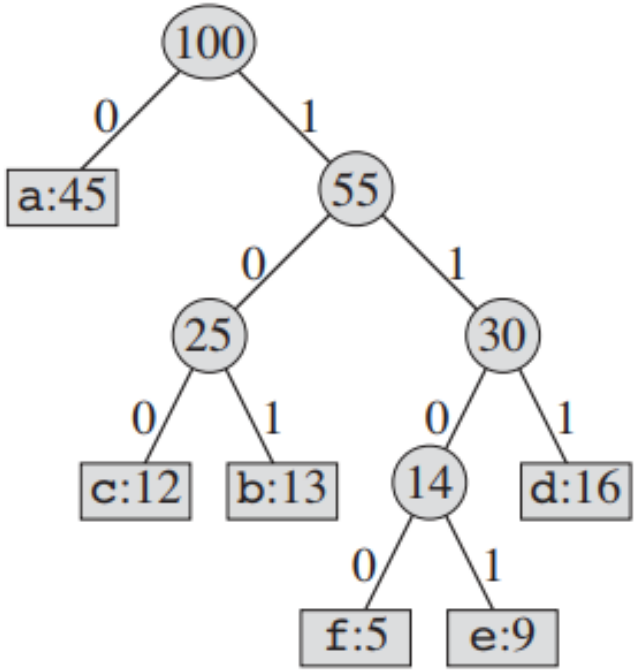
Codes in which no codeword is also a prefix of some other codeword -- are called ***prefix codes.***

| ***Not prefix codes*** | ***Prefix codes*** |
|---|---|
| ***A = 011*** | ***A = 011*** |
| ***B = 01*** | ***B = 11*** |

- **A prefix code can always achieve the optimal data compression among any character code**, and so we suffer no loss of generality by restricting our attention to prefix codes.
- Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

# Decoding


100
0
1
86
14
0
1
0
58
28
14
0
1
0
1
0
1
a:45
b:13
c:12
d:16
e:9
f:5


100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

# Decoding

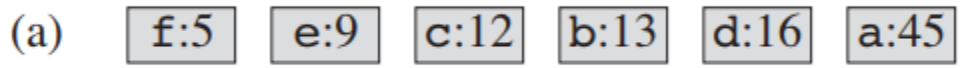
- The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation.
- We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.” Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.
- An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children.

# Huffman Code

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

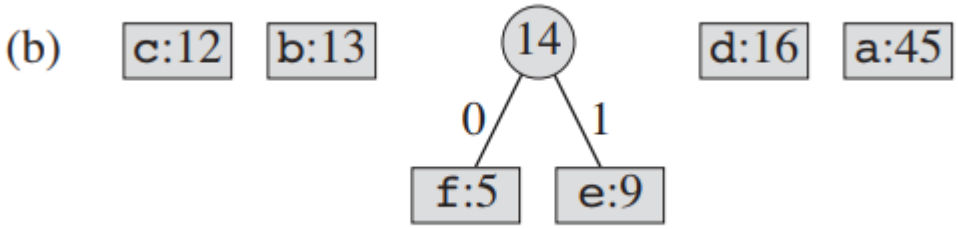
# Simulation

Each part shows the contents of the queue sorted into increasing order by frequency.

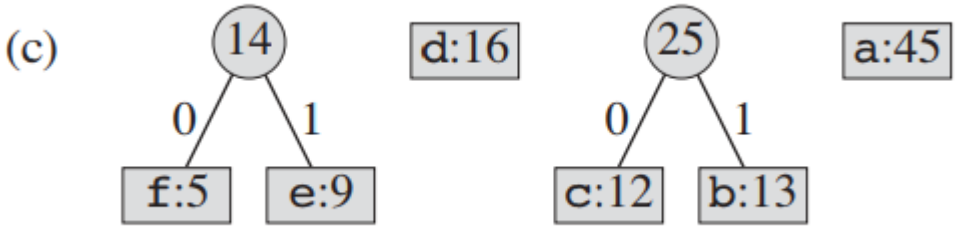


# Simulation

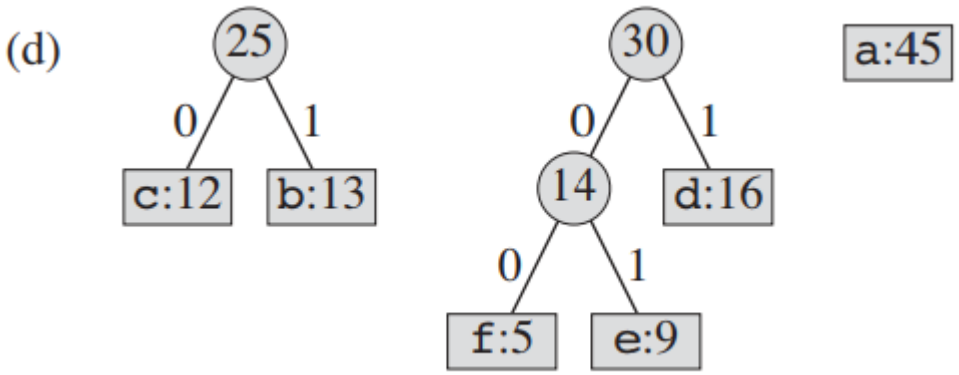
At each step, the two trees with lowest frequencies are merged.



# Simulation

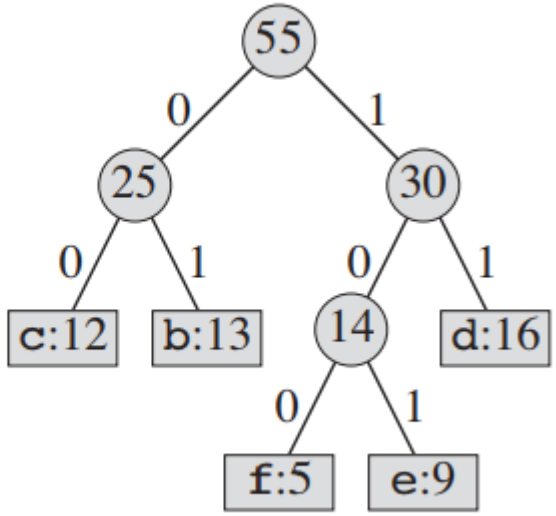


# Simulation

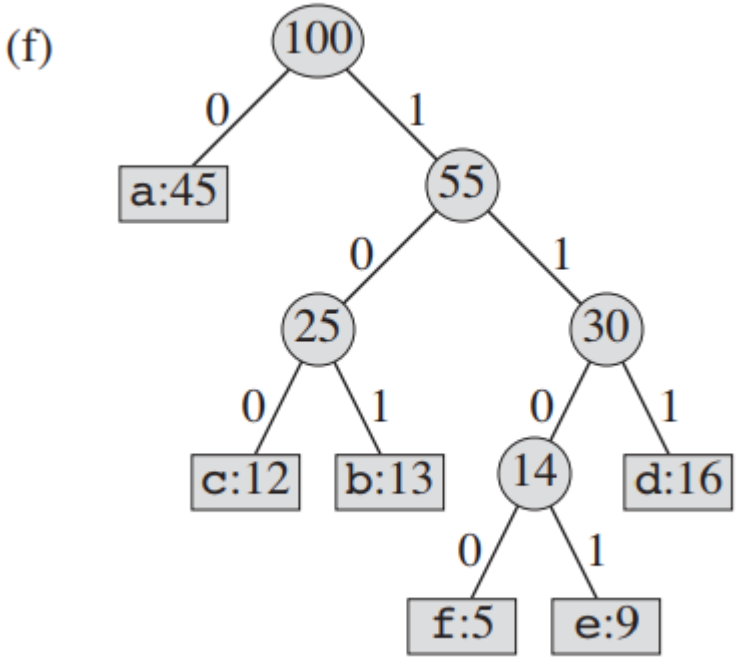


# Simulation

(e) a:45



# Simulation



# Runtime Analysis

To analyze the running time of Huffman's algorithm, we assume that $Q$ is implemented as a binary min-heap (see Chapter 6). For a set $C$ of $n$ characters, we can initialize $Q$ in line 2 in $O(n)$ time using the BUILD-MIN-HEAP procedure discussed in Section 6.3. The **for** loop in lines 3–8 executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of $n$ characters is $O(n \lg n)$. We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree (see Chapter 20).