Huffman codes

Problem Statement

- Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string

Problem Statement

- 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.

	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 we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

Options to represent such a file of information

- Consider problem of designing a binary character code
- Each character is represented by a unique binary string, which we call a codeword
- 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

Variable–Length Code

- Better than a fixed–length code
- Giving frequent characters short codewords and infrequent characters long code–words

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 we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

- Figure 16.3 shows such a code; here the 1–bit string 0 represents a, and the 4–bit string 1100 represents f

Variable–Length Code

- This code requires (45 . 1 + 13 . 3 + 12 . 3 + 16 . 3 + 9 . 4 + 5 . 4). 1,000 = 224,000 bits to represent the file, a savings of approximately 25%

Variable–Length Code

- Better than a fixed–length code
- Giving frequent characters short codewords and infrequent characters long code–words

Prefix codes

- We consider here only codes in which no codeword is also a prefix of some other codeword.
- Such codes are called prefix codes

Encoding

- Simple for any binary character code;
- We just concatenate the codewords representing each character of the file
- For example, with the variable–length prefix code of Figure 16.3, we code the 3–character file abc as 0.101.100 = 0101100, where "." denotes concatenation.

Decoding

- 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.
- We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file

Decoding

- In our example, the string 001011101 parses uniquely as 0 . 0 . 101 . 1101, which decodes to aabe.
- 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.

Decoding

- 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.”

Fixed vs Variable Length Code

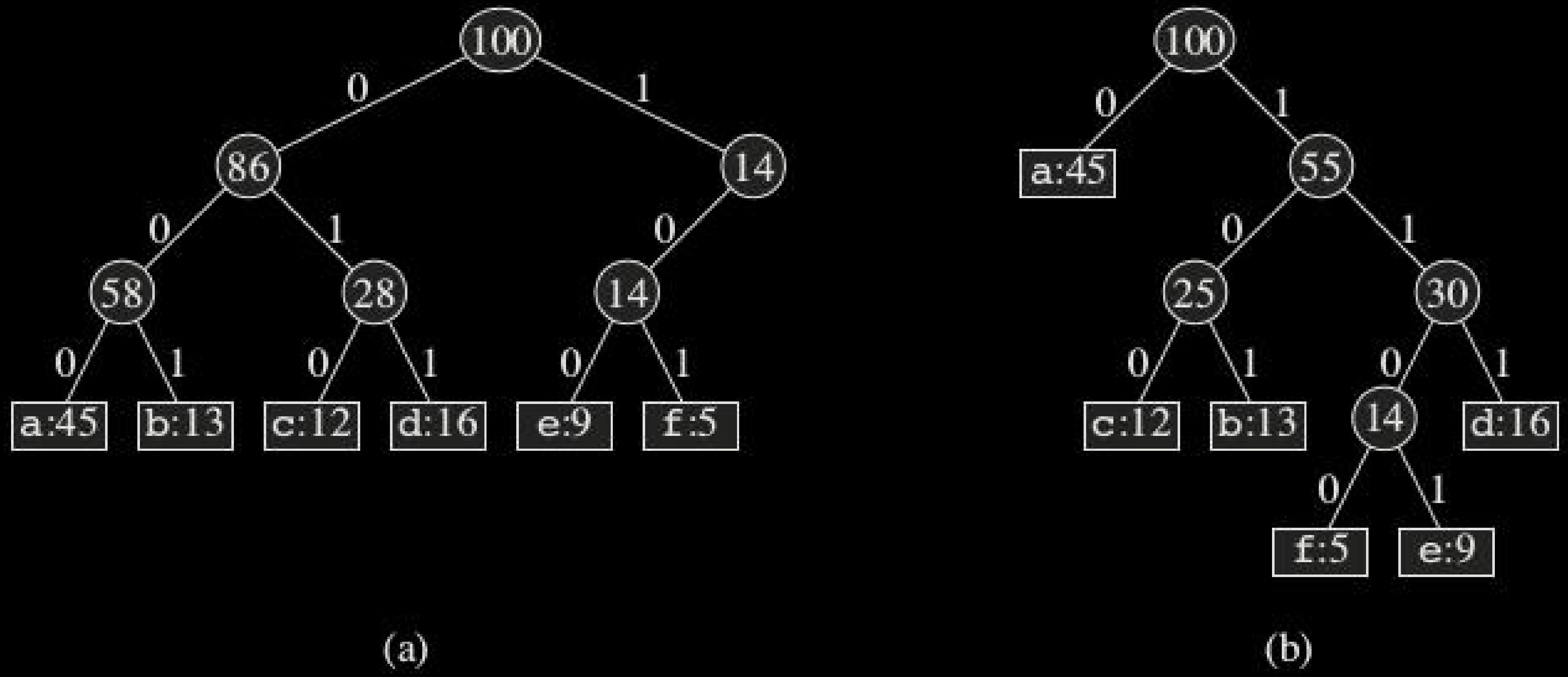


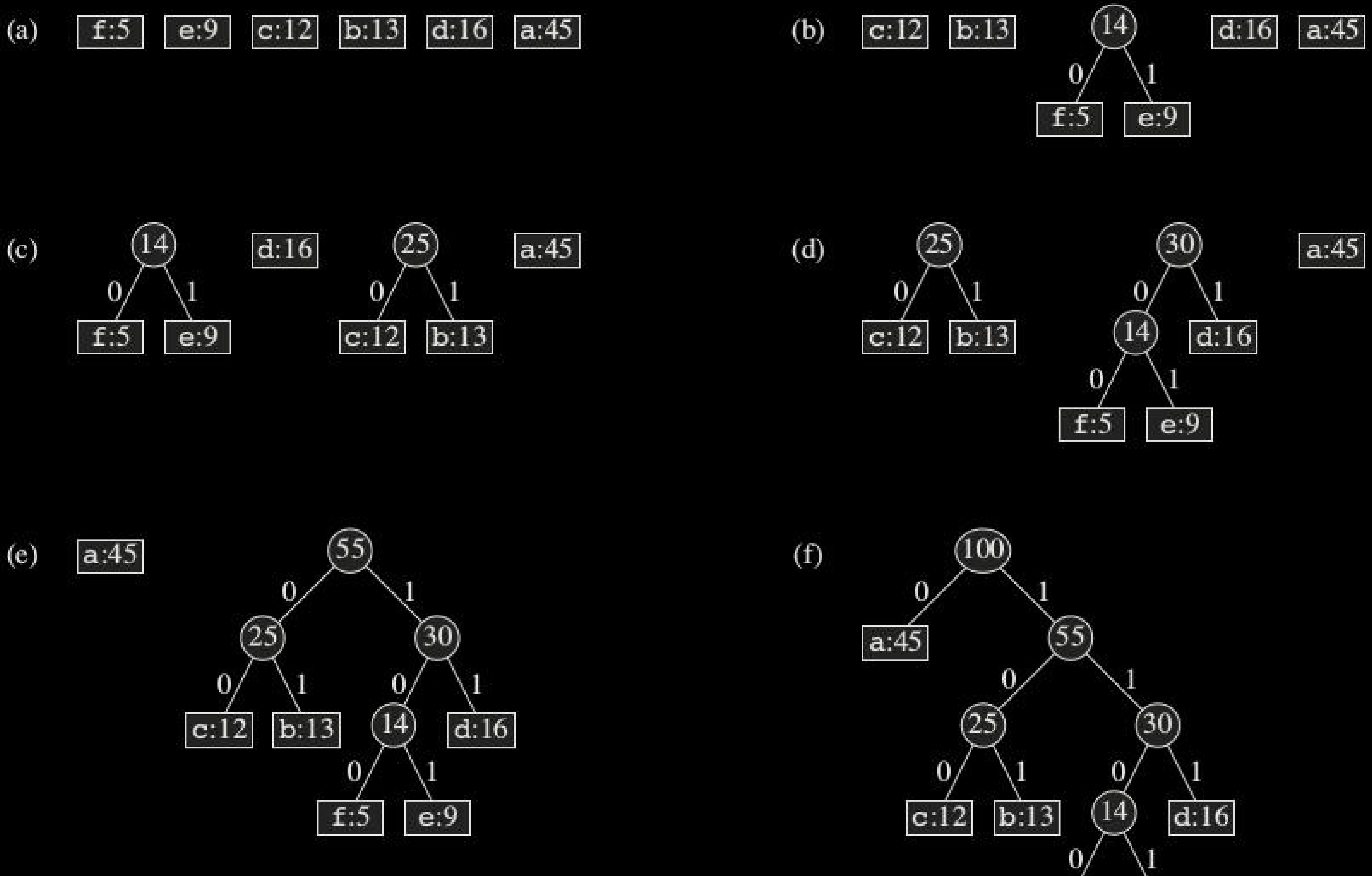
Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, ...,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

Decoding

- An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children
- Fixed–length code in our example is not optimal since its tree, shown in Figure 16.4(a), is not a full binary tree: it contains codewords beginning 10. . . , but none beginning 11

Pseudocode

- C is a set of n characters
- Each character c ∈ C is an object with an attribute c.freq giving its frequency
- algorithm builds the tree T corresponding to the optimal code in a bottom–up manner
- Begins with a set of |C| leaves and performs a sequence of |C| – 1 "merging" operations to create the final tree
- Algorithm uses a min–priority queue Q, keyed on the freq attribute, to identify two least–frequent objects to merge together



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

Correctness of Huffman's algorithm

- To prove that greedy algorithm HUFFMAN is correct ST:
- Problem of determining an optimal prefix code exhibits the greedy–choice
- Optimal–substructure properties

Lemma to show that Greedy–Choice Property holds

- Let C be an alphabet in which each character c ∈ C has frequency c.freq.
- Let x and y be two characters in C having the lowest frequencies.
- Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Lemma to show that Greedy–Choice Property holds

- Idea of the proof is to take the tree T representing an arbitrary optimal prefix code
- Modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree

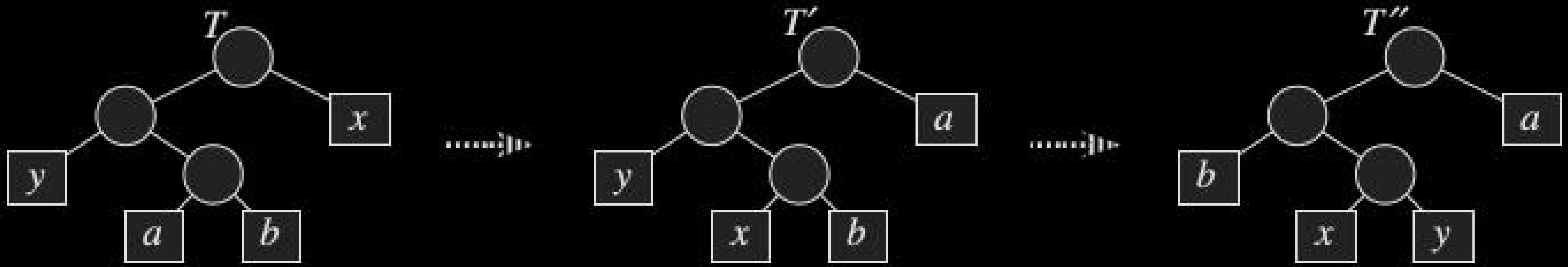


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.