16.3 The Huffman code problem

Huffman codes are widely used and very effective techniques for data compression.

The principle behind this technique is to design a binary character code for each character, using a variable number of bits to represent each character, so as to reduce the total length of the document.

Example: 100,000-character document using $\{$a,b,c,d,e,f$\}$.

- Using (**fixed-length**) 3 bits per character, the document needs 300,000 bits.
- Using a **variable-length** per character, the document can be stored in 224,000 bits.

16.3 The Huffman code problem

	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

A string “abaabef” then can be encoded as

- 000001000000001100101, using the fixed-length encoding
- 0101001011101100, using the variable-length encoding

16.3 The Huffman code problem

Another example: In the MP3 audio compression scheme, a sound signal is encoded in three steps.

- It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of 44,100 samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44,100 \approx 130$ million measurements.
- Each real-valued sample s_i is quantized: approximated by a nearby number from a finite set Γ. This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.
- The resulting string of length T is encoded in binary, using Huffman encoding.

16.3 The Huffman code problem (continued)

To allow easy, unambiguous decoding, we require the code to be a prefix code: no code is a prefix of another.

A danger with having codewords of different lengths is that the resulting encoding may not be uniquely decipherable.

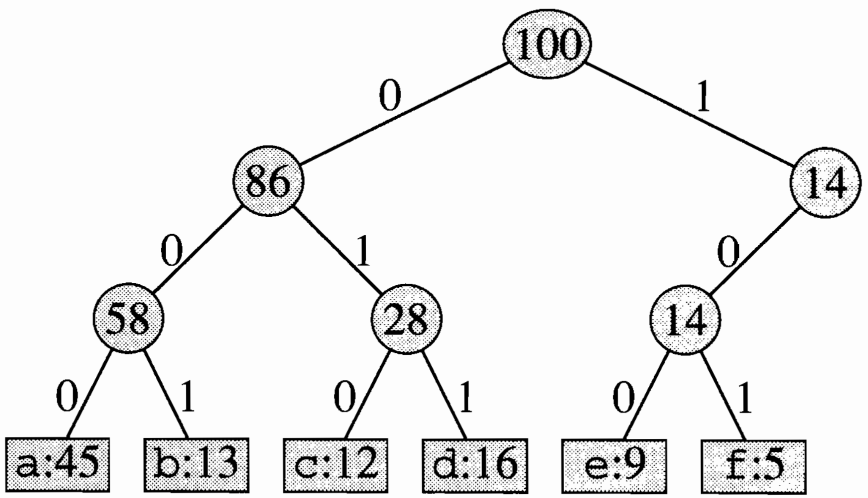
For example, if the codewords are $\{0, 01, 11, 001\}$, the decoding of string like 001 is ambiguous. You can interpret it as 0-01, or 001.

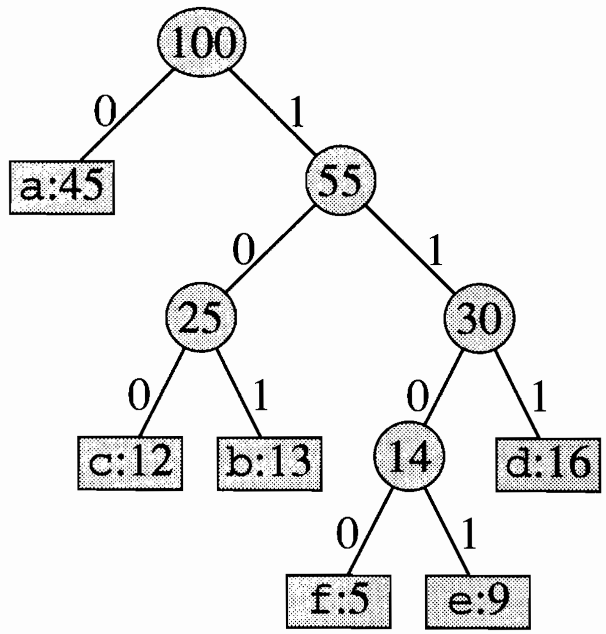
We thus avoid this problem by insisting on the **prefix-free** property: **no codeword can be a prefix of another codeword**, where codeword is a string of digits to represent a character in the encoding.

16.3 The Huffman code problem (continued)

Prefix codes are easily representable as binary trees.

	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





16.3 The Huffman code problem (continued)

Given a character set C and the frequency $f(c)$ of each character $c \in C$ in the document $\mathcal{A}$, the problem is to find a prefix code that minimizes the total length of the document.

The tree representing such an optimal code is called **a Huffman tree T**. We aim to minimize

$$B(T, \mathcal{A}) = \sum_{c \in C} f(c) d_T(c),$$

where

- $d_T(c)$ is the depth of character c in the tree T (numbers of bits used to encode character c), or the length of the codeword of character c.
- $B(T, \mathcal{A})$ is the total number of bits for document $\mathcal{A}$ based on the Huffman tree T.

The problem is to design a tree T to minimize $B(T, \mathcal{A})$.

16.3 The Huffman code problem (continued)

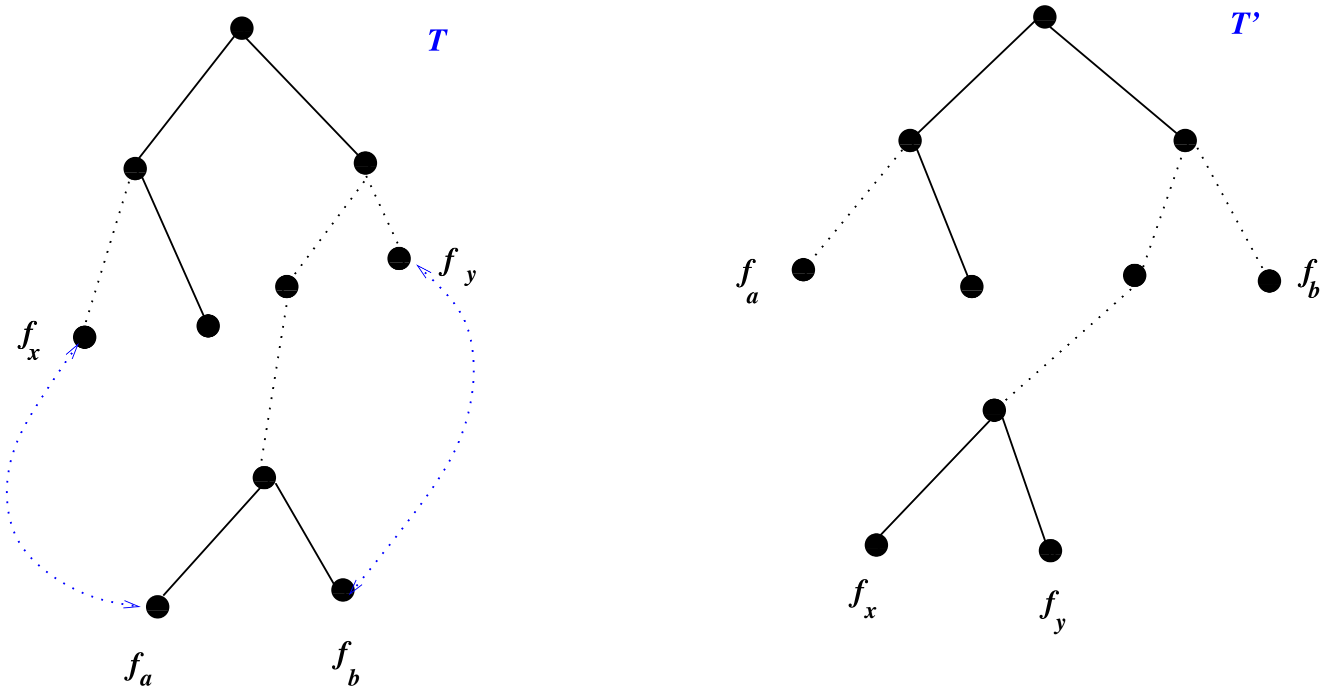
Observations: Let T be a Huffman tree of a character set C.

- There are no nodes with only a single child. Why?
- Rearranging characters within the same level of the tree T does not change $B(T, \mathcal{A})$. Why?
- Swapping a character at a higher level with another less-frequent character reduces $B(T, \mathcal{A})$, where the level of the tree root is the lowest level.

We thus can assume that the two least-frequent characters x and y are siblings at the greatest depth. We have a greedy strategy for the Huffman coding problem:

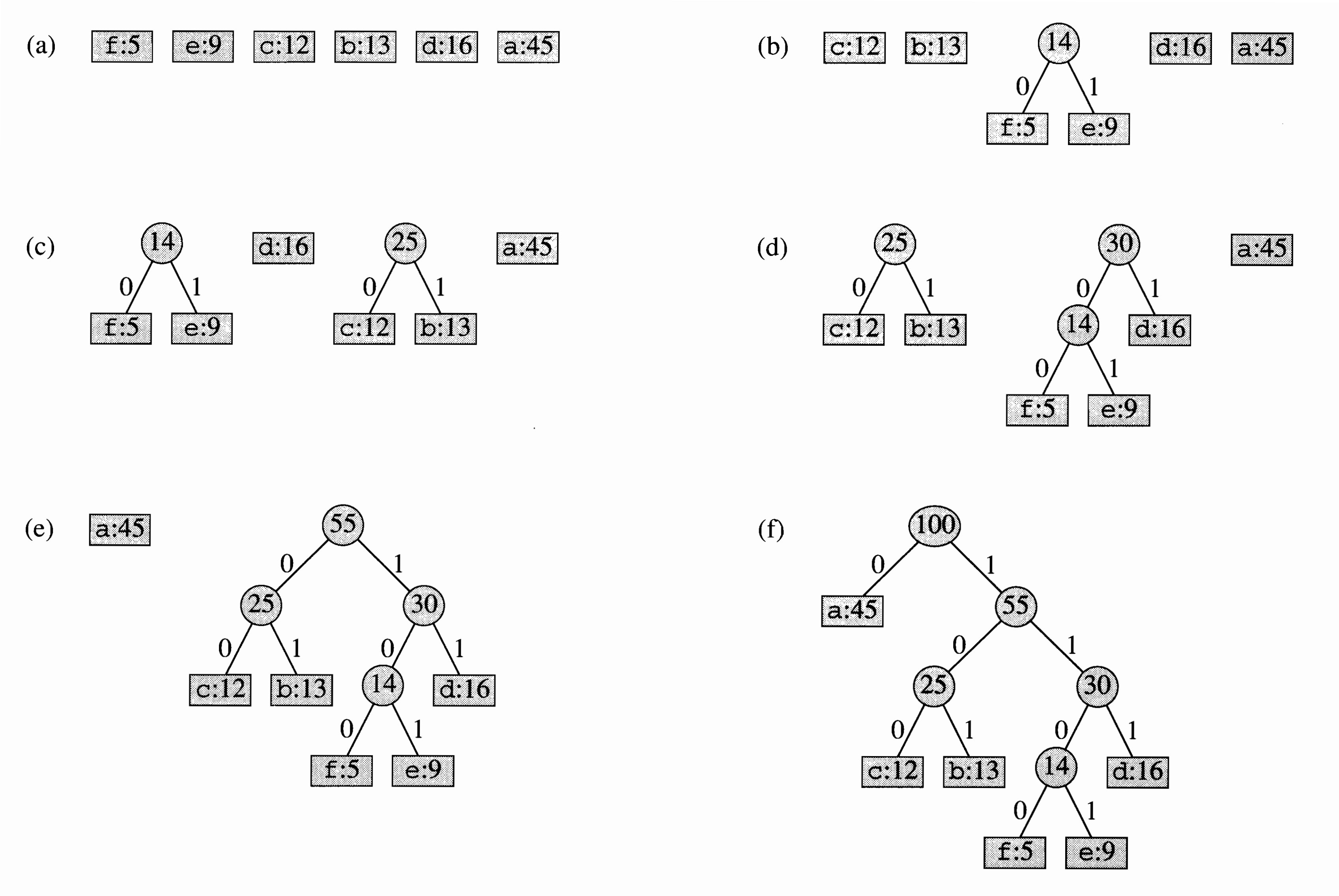
- Replacing x, y and their parent by a leaf representing a new character $\boxed{\text{xy}}$ with frequency $f(\boxed{\text{xy}}) = f(x) + f(y)$ gives a Huffman tree for $C \cup \{\boxed{\text{xy}}\} \setminus \{x, y\}$.

16.3 The Huffman code problem (continued)



$\max\{f_x, f_y\} < \min\{f_a, f_b\}$

16.3 The Huffman code problem (continued)



16.3 The Huffman code problem (continued)

For efficient implementation of a Huffman tree, we can use a priority queue data structure - a minimum heap. It has a set of weighted items as its key value, with fast (such as $O(\log n)$) execution of the operations **Extract_MIN:** find and delete the element with the least weight, and **Insert:** insert an element with a key to the priority queue.

Huffman(C)

$n \leftarrow |C|$;
$Q \leftarrow C$; /* The priority queue */
for $i \leftarrow 1$ to $n-1$
    do $z \leftarrow$ allocate_node();
        $left[z] = x \leftarrow$ Extract_MIN(Q);
        $right[z] = y \leftarrow$ Extract_MIN(Q);
        $f[z] \leftarrow f[x] + f[y]$;
        $Insert(Q, z)$;
return Extract_MIN(Q).

16.3 The Huffman code problem (continued)

Exercise: What is the running time of the greedy algorithm for the construction of a Huffman tree? assuming that there are n characters in a document.