

Huffman Codes

- Widely used technique for data compression
- Assume the data to be a sequence of characters
- Looking for an effective way of storing the data
- ***Binary character code***
 - Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- 3 bits needed
- $a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 100$, $f = 101$
- Requires: $100,000 \cdot 3 = 300,000$ bits

Huffman Codes

- Idea:
 - Use the frequencies of occurrence of characters to build a optimal way of representing each character

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- Assign short codewords to frequent characters and long codewords to infrequent characters
- $a = 0$, $b = 101$, $c = 100$, $d = 111$, $e = 1101$, $f = 1100$
- $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000$
 $= 224,000$ bits

Prefix Codes

- Prefix codes:
 - Codes for which no codeword is also a prefix of some other codeword
 - Better name would be “prefix-free codes”
- We can achieve optimal data compression using prefix codes
 - We will restrict our attention to prefix codes

Encoding with Binary Character Codes

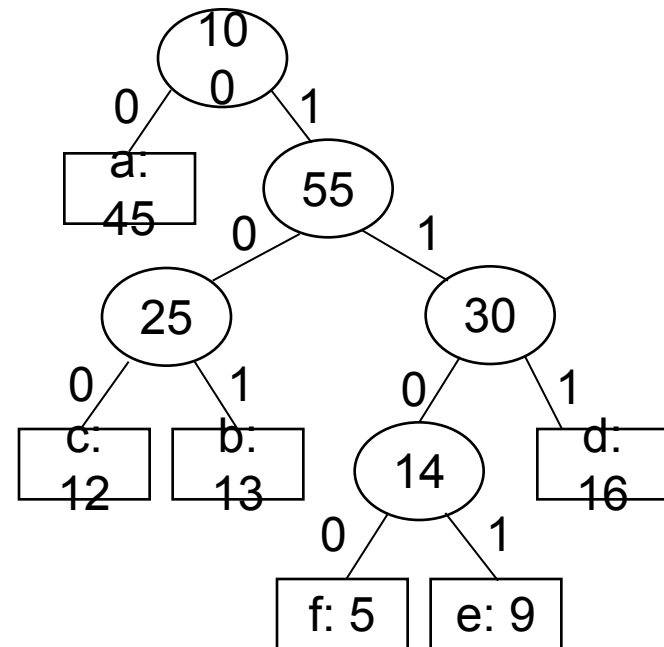
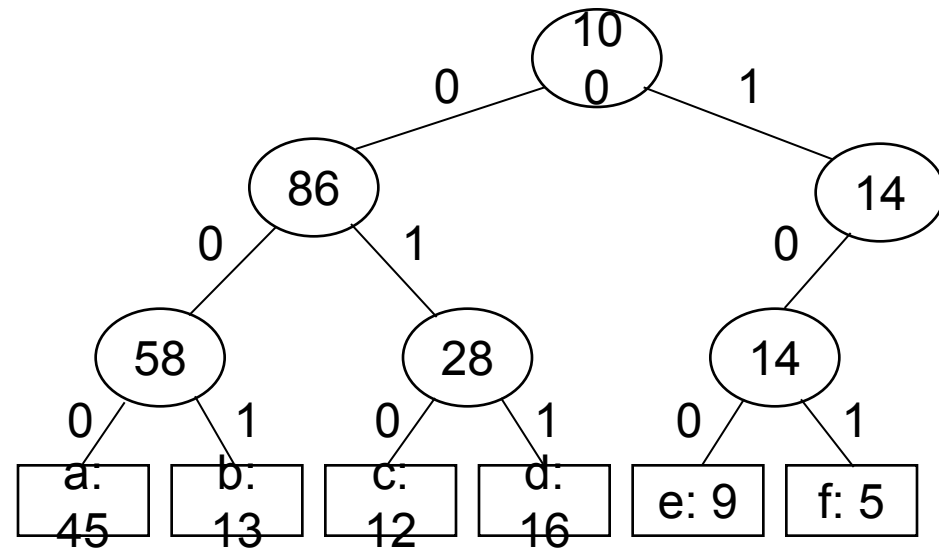
- Encoding
 - Concatenate the codewords representing each character in the file
- *E.g.:*
 - $a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100$
 - $abc = 0 \cdot 101 \cdot 100 = 0101100$

Decoding with Binary Character Codes

- Prefix codes simplify decoding
 - No codeword is a prefix of another \Rightarrow the codeword that begins an encoded file is unambiguous
- Approach
 - Identify the initial codeword
 - Translate it back to the original character
 - Repeat the process on the remainder of the file
- *E.g.:*
 - $a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100$
 - $001011101 = \begin{array}{ccc} 0 & \cdot & \cdot & \cdot \\ & 0 & 101 & 1101 \end{array} = \text{aabe}$

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
 - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
 - Length of the path from root to the character leaf (depth of node)



Optimal Codes

- An optimal code is always represented by a **full binary tree**
 - Every non-leaf has two children
 - Fixed-length code is not optimal, variable-length is
- How many bits are required to encode a file?
 - Let \mathcal{C} be the alphabet of characters
 - Let $f(c)$ be the frequency of character c
 - Let $d_T(c)$ be the depth of c 's leaf in the tree T corresponding to a prefix code

$$B(T) = \sum_{c \in \mathcal{C}} f(c) d_T(c) \quad \text{the cost of tree } T$$

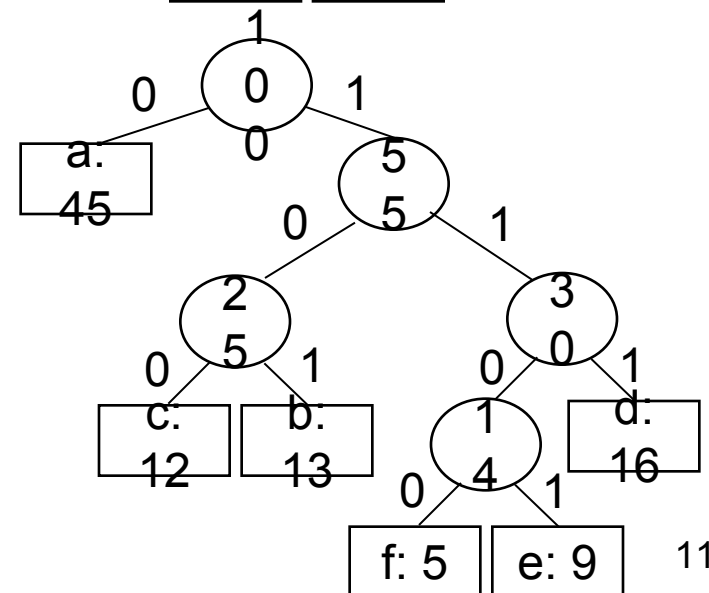
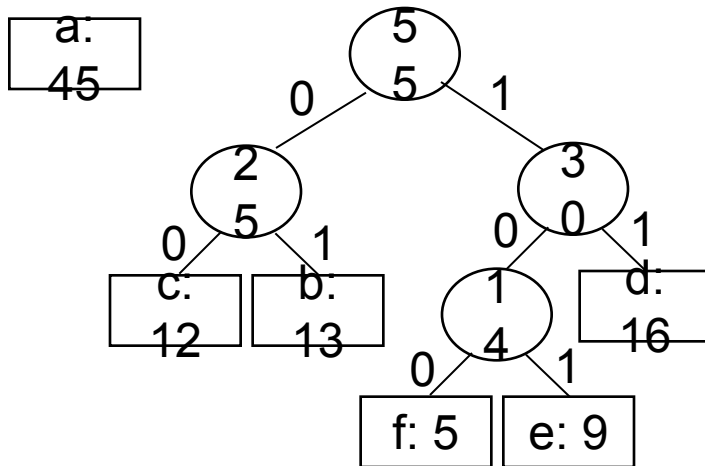
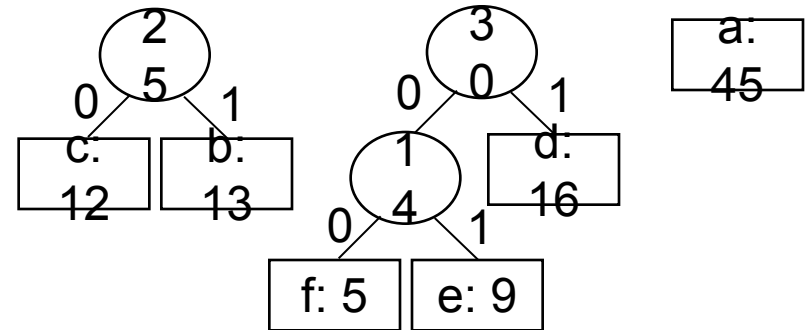
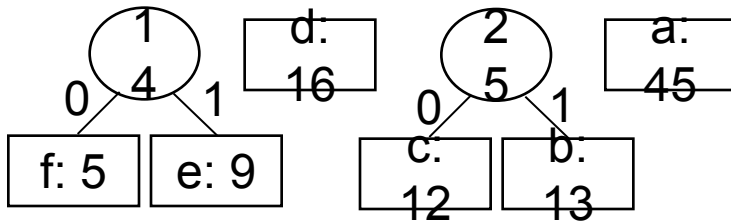
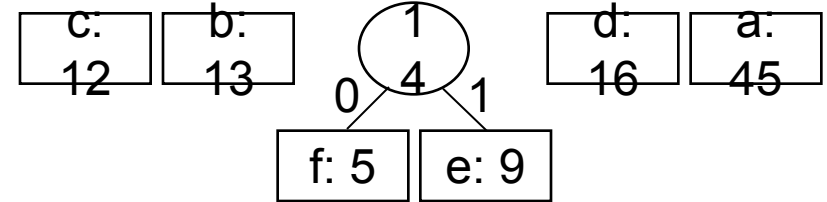
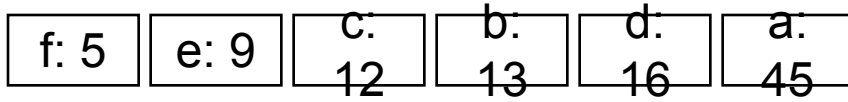
Constructing a Huffman Code

- A greedy algorithm that constructs an optimal prefix code called a **Huffman code**
- Assume that:
 - \mathcal{C} is a set of n characters
 - Each character has a frequency $f(c)$
 - The tree T is built in a bottom up manner
- Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45
------	------	----------	----------	----------	----------

 - Start with a set of $|\mathcal{C}|$ leaves
 - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
 - Use a min-priority queue Q , keyed on f to identify the two least frequent objects

Example



Building a Huffman Code

Alg.: HUFFMAN(\mathcal{C})

Running time: $O(n \lg n)$

1. $n \leftarrow |\mathcal{C}|$
 2. $Q \leftarrow \mathcal{C}$ $\longleftarrow O(n)$
 3. **for** $i \leftarrow 1$ **to** $n - 1$
 4. **do** allocate a new node z
 5. $\text{left}[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
 6. $\text{right}[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
 7. $f[z] \leftarrow f[x] + f[y]$
 8. INSERT (Q, z)
 9. **return** EXTRACT-MIN(Q)
- $\left. \begin{array}{l} \text{lines 4-8} \end{array} \right\} O(n \lg n)$