

# Compressione

**Liceo G.B. Brocchi - Bassano del Grappa (VI)**  
**Liceo Scientifico - opzione scienze applicate**  
Giovanni Mazzocchin

# Compressione



# Compressione



# Compressione

- **Ridondanza:** presenza di bit deducibili da altri. È possibile rimuovere i bit ridondanti senza perdere informazione
- **Lossless compression:** i dati originali possono essere completamente recuperati da quelli compressi, senza perdita di informazione
- **Lossy compression:** la compressione è irreversibile. I dati originali non possono essere recuperati completamente

# Run-length encoding (RLE)

- **Run-length encoding** è una tecnica di compressione lossless nella quale una sequenza di byte uguali (*run*) viene codificata nella forma `count + byte`
- Esempio:  
AAAABBBBBBWWWW → 4A6B5W
- Questa tecnica è particolarmente efficiente su dati contenenti molti *run*
- Esercizio: implementare il run-length encoding in Python

# Huffman coding (David A. Huffman, 1952)

- Consideriamo dati costituiti da sequenze di caratteri
- Abbiamo bisogno di una tabella che contenga la frequenza di ciascun carattere all'interno dei dati da comprimere
- **Esempio:**
  - immaginiamo un file di  $1.0E+05$  (100 000) caratteri
  - nel testo compaiono soltanto 6 caratteri diversi:
    - a, b, c, d, e, f
  - a compare 45 000 volte, b 13 000 volte etc...
  - se usassimo un **fixed-length code** avremmo bisogno di 3 bit per ciascun carattere:
    - con 2 bit codificherebbero  $2^2$  caratteri, che è  $< 6$
    - con 3 bit codificherebbero  $2^3$  caratteri, che è  $> 6$

# Huffman coding

- 100 000 caratteri, 3 bit per carattere
- Il file peserà 300 000 bit, ossia 37 500 byte, circa 37 kB

**can we do better?**

# Huffman coding

- Già nel XIX secolo, Samuel Morse codificava le lettere più frequenti della lingua inglese con simboli più brevi, arrivando a produrre un **variable-length code**
- La **Teoria dell'informazione** (argomento dell'ultimo anno) fornisce le basi per comprendere nel dettaglio la compressione. Noi cercheremo di introdurre alcuni concetti intuitivamente



# Huffman coding

- L'obiettivo è generare un *variable-length code* particolare, detto **prefix code**
- In un **prefix code** vale la seguente condizione:
  - per ogni coppia di simboli ( $\mathbf{a_1}$ ,  $\mathbf{a_2}$ ), se il simbolo  $\mathbf{a_1}$  è codificato con la stringa binaria (**codeword**)  $\mathbf{c_1}$ , e il simbolo  $\mathbf{a_2}$  è codificato con la stringa binaria  $\mathbf{c_2}$ , allora  $\mathbf{c_1}$  non è un prefisso di  $\mathbf{c_2}$  e  $\mathbf{c_2}$  non è un prefisso di  $\mathbf{c_1}$
- Vediamo degli esempi

# Huffman coding

	a	b	c	d	e	f
frequency (in thousands)	45	13	12	16	9	5
fixed-length codeword	000	001	010	011	100	101
non-prefix, variable-length codeword	0	01	011	0111	01111	011111
prefix, variable-length codeword	0	101	100	111	1101	1100

# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

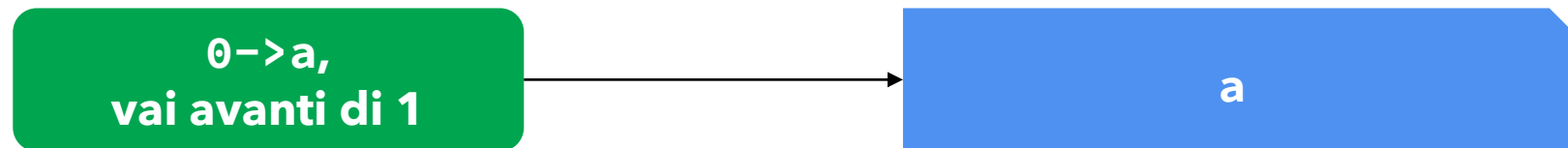
- I prefix code sono molto adatti per essere decodificati
- **Esempio)**  
un decodificatore inizia a leggere la stringa seguente da sinistra:

**001011101**

# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

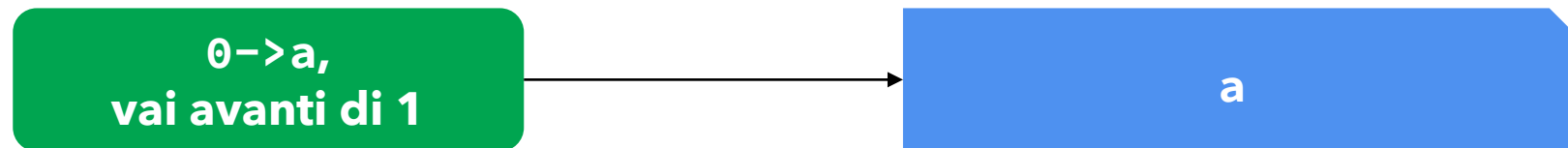
001011101



# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

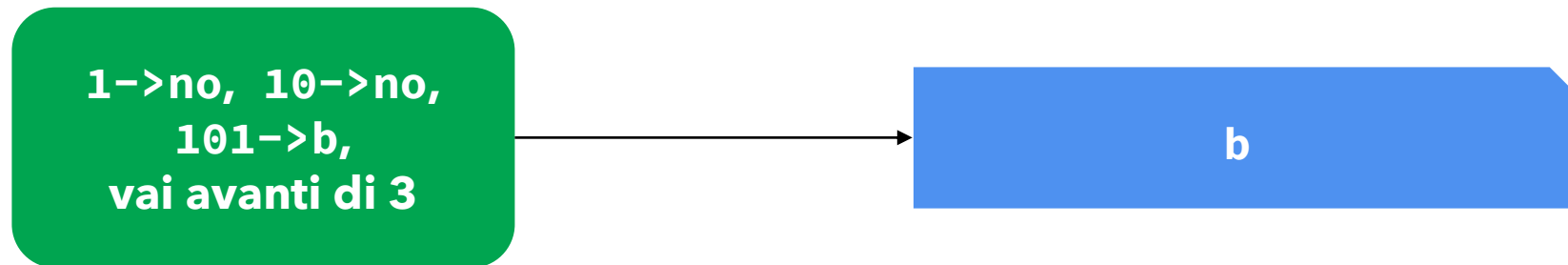
**001011101**



# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

**001011101**



# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

**001011101**

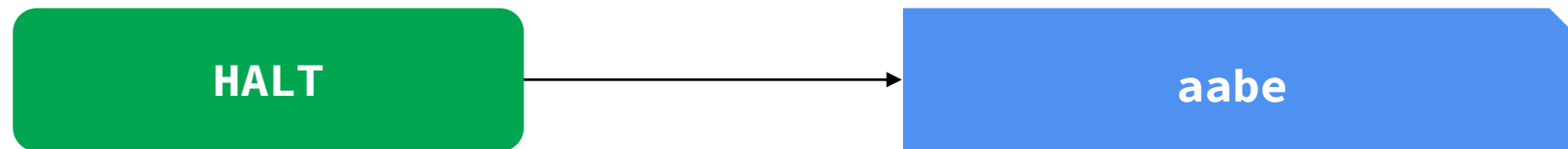
1→no, 11→no,  
110→no,  
1101→e,  
vai avanti di 4

e

# Huffman coding

	a	b	c	d	e	f
prefix, variable-length codeword	0	101	100	111	1101	1100

**001011101**





# L'algoritmo di Huffman

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

L'algoritmo di Huffman permette di generare codici prefissi ottimi

Eccolo descritto in lingua naturale:

**iterare le seguenti operazioni fintantoché ci sono elementi nell'elenco:**

1. ordinare l'elenco in senso ascendente in base alla frequenza (passaggio non necessario)
2. considerare i 2 elementi di frequenza più bassa
3. rimuoverli dall'elenco
4. creare un nodo contenente la somma delle frequenze dei 2 elementi estratti, che ha come figli i 2 elementi estratti
5. inserire il nodo creato nell'elenco, con la frequenza calcolata sopra

# L'algoritmo di Huffman

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

**f: 5**

**e: 9**

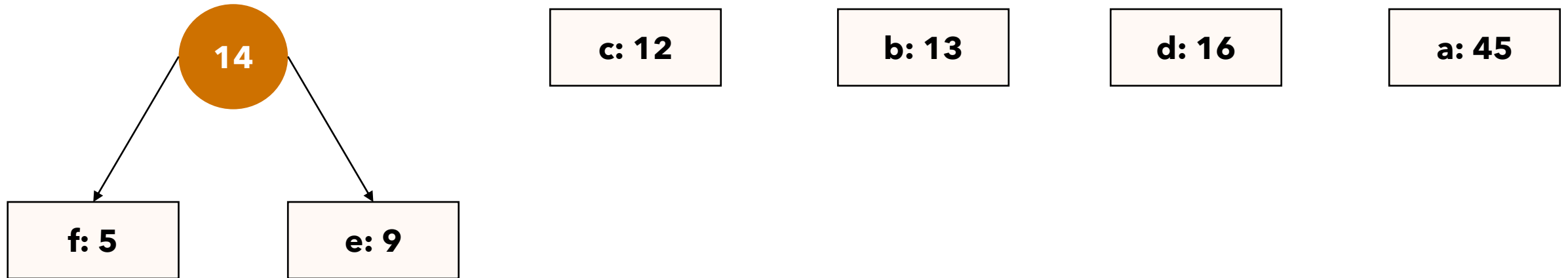
**c: 12**

**b: 13**

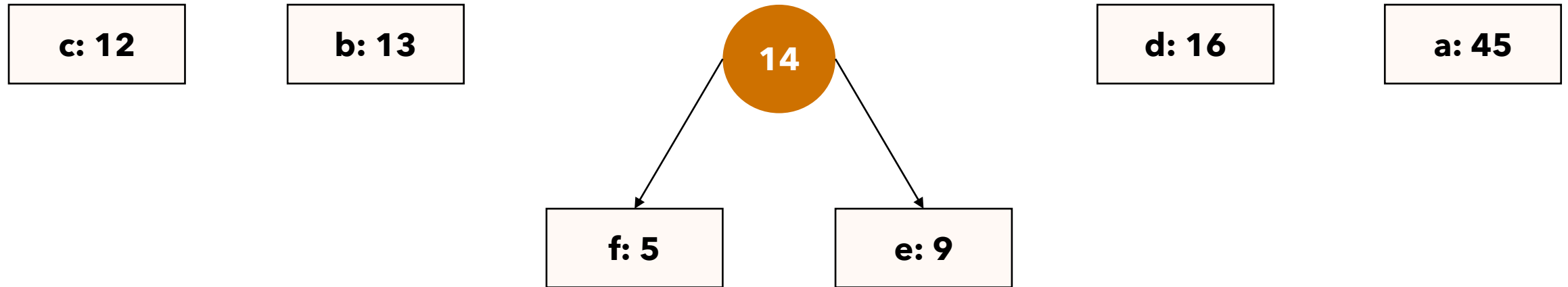
**d: 16**

**a: 45**

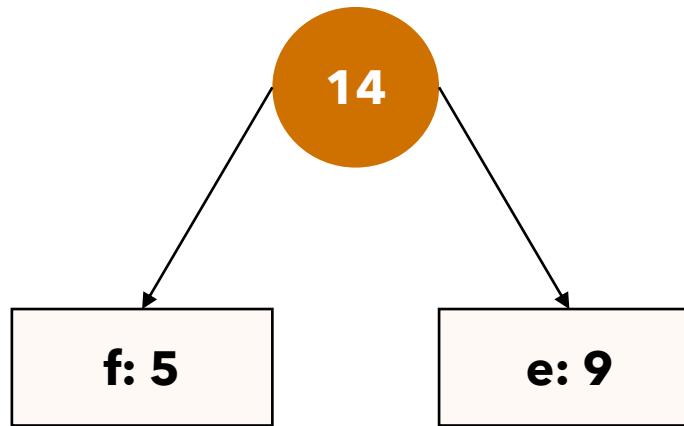
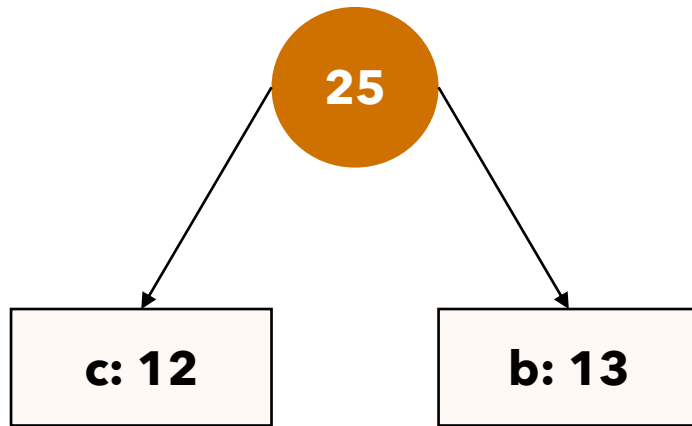
# L'algoritmo di Huffman



# L'algoritmo di Huffman



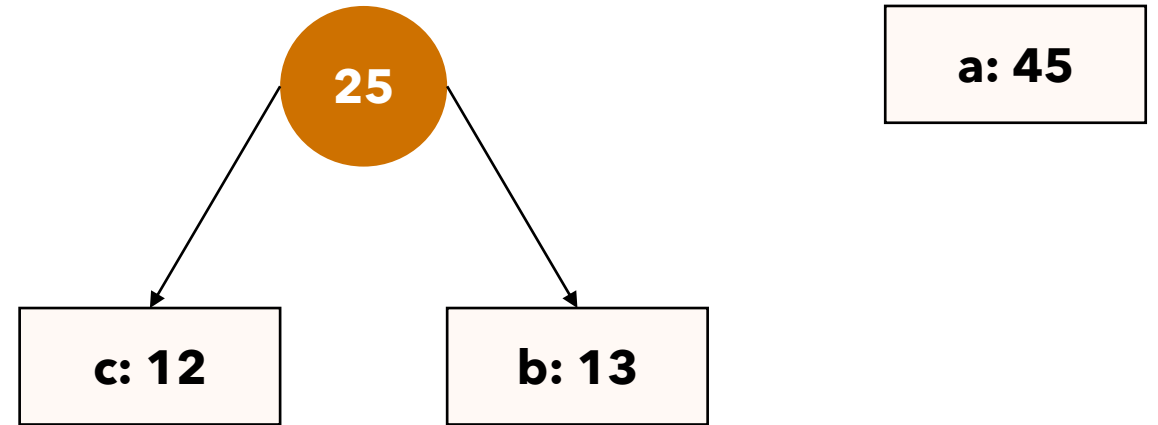
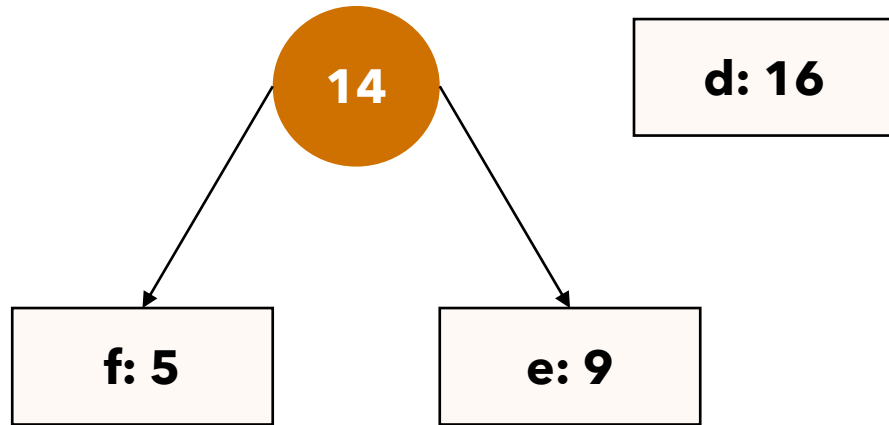
# L'algoritmo di Huffman



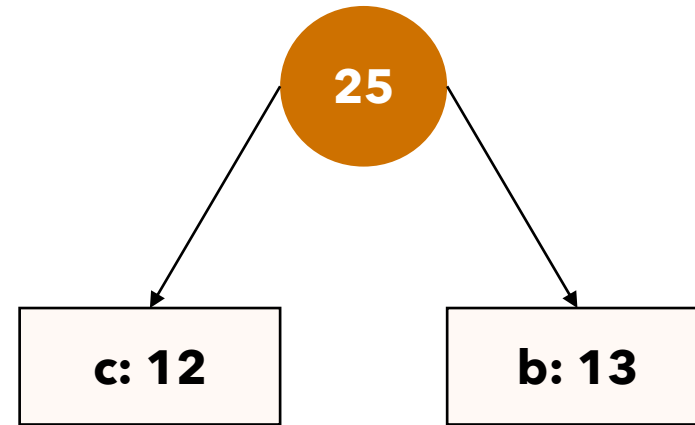
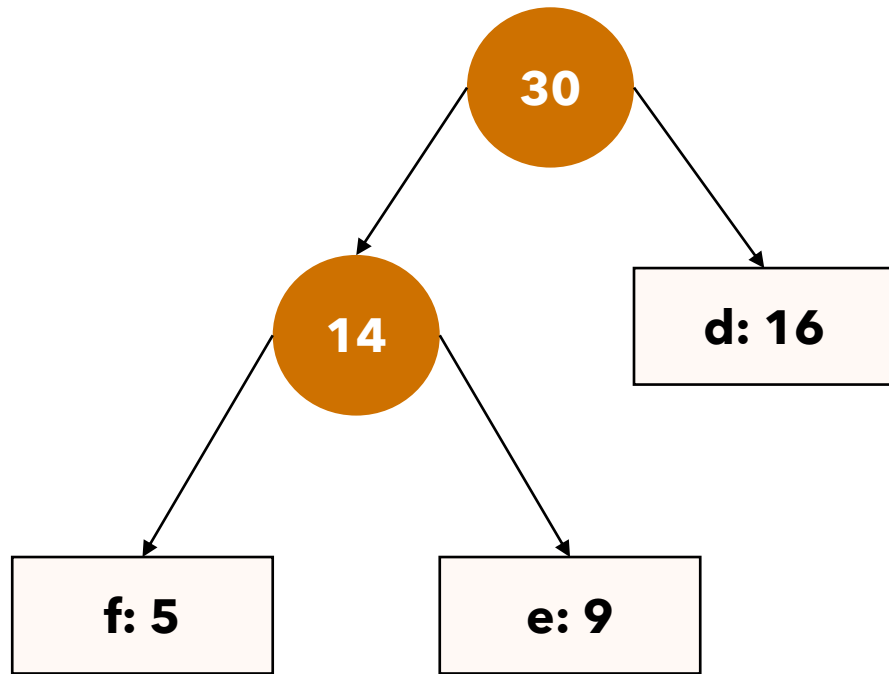
**d: 16**

**a: 45**

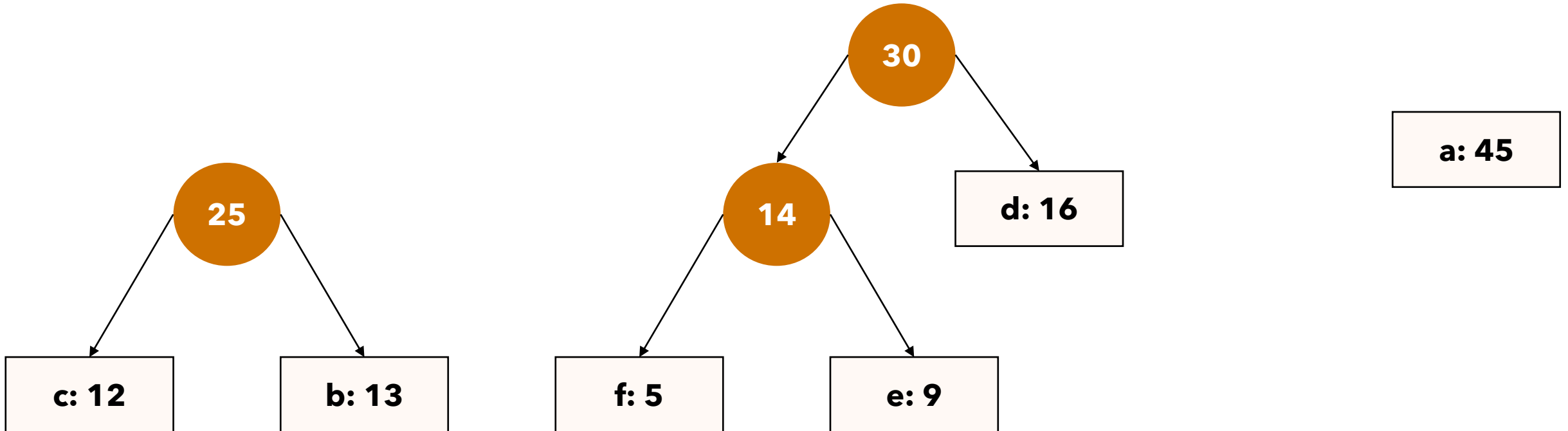
# L'algoritmo di Huffman



# L'algoritmo di Huffman

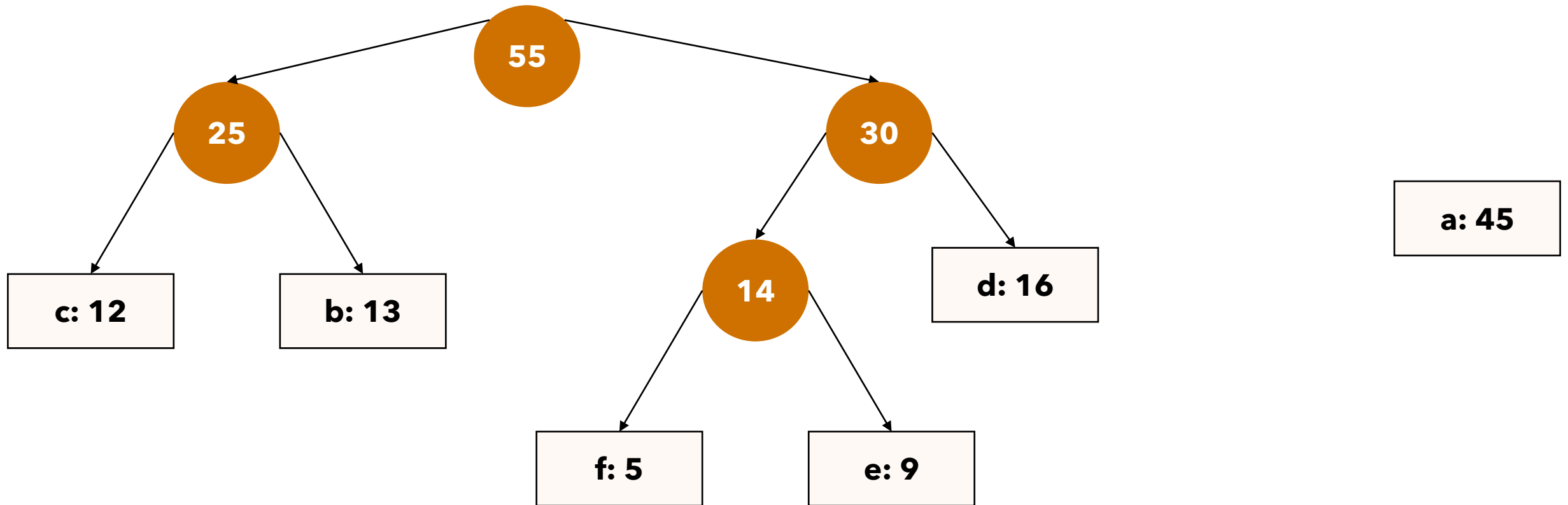


# L'algoritmo di Huffman



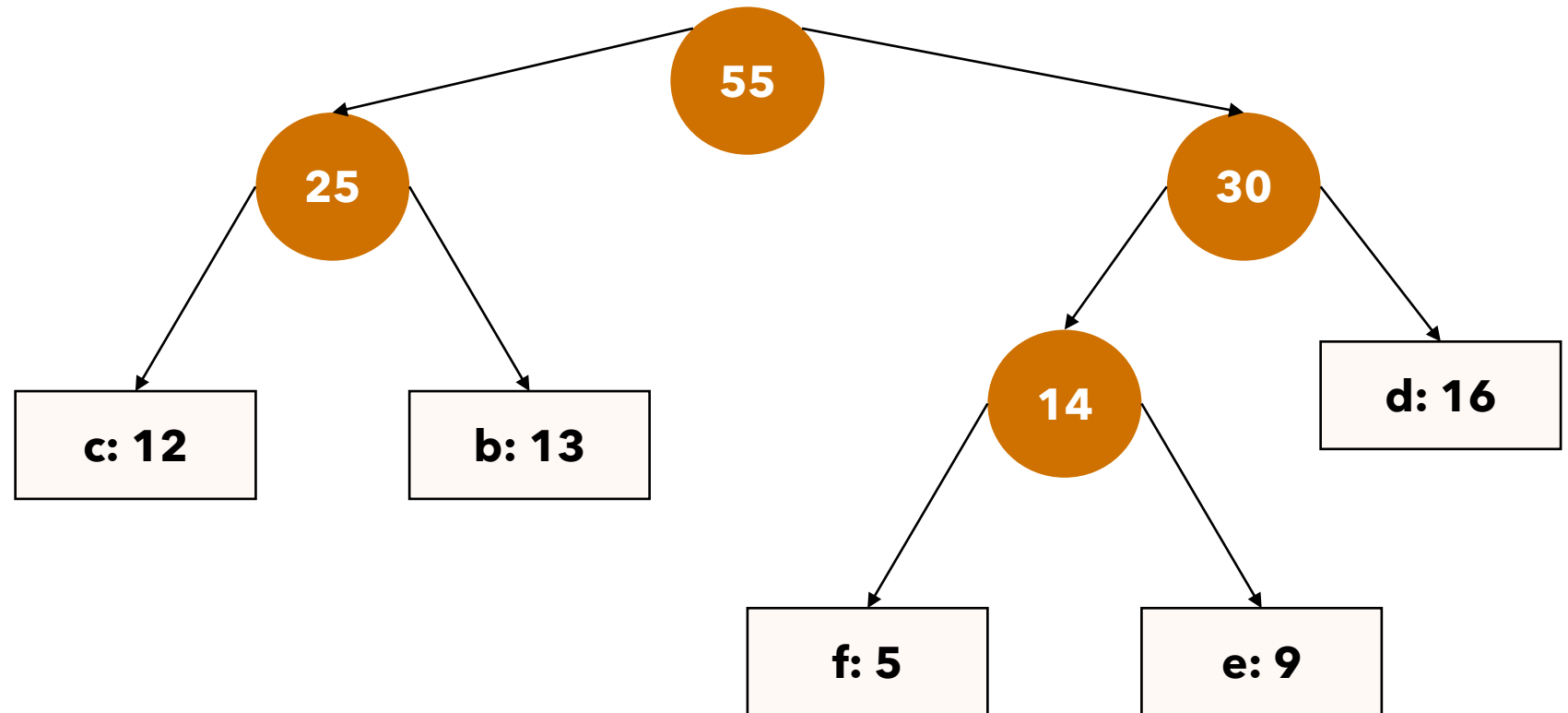


# L'algoritmo di Huffman

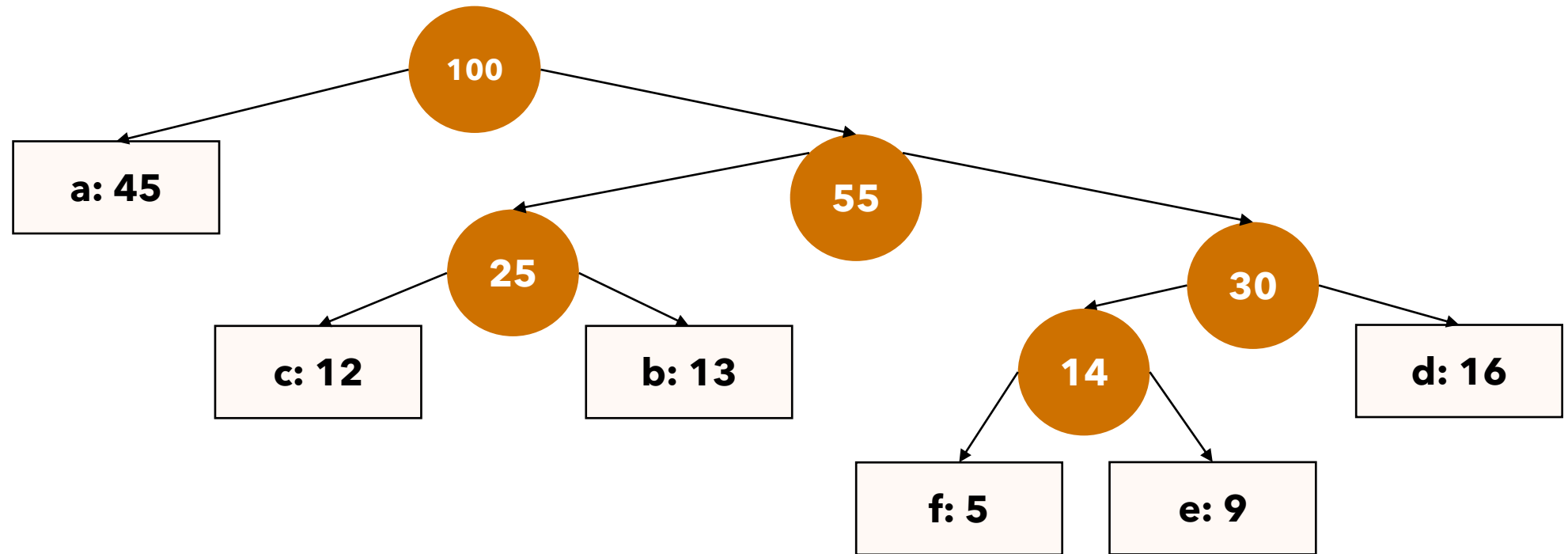


# L'algoritmo di Huffman

**a: 45**



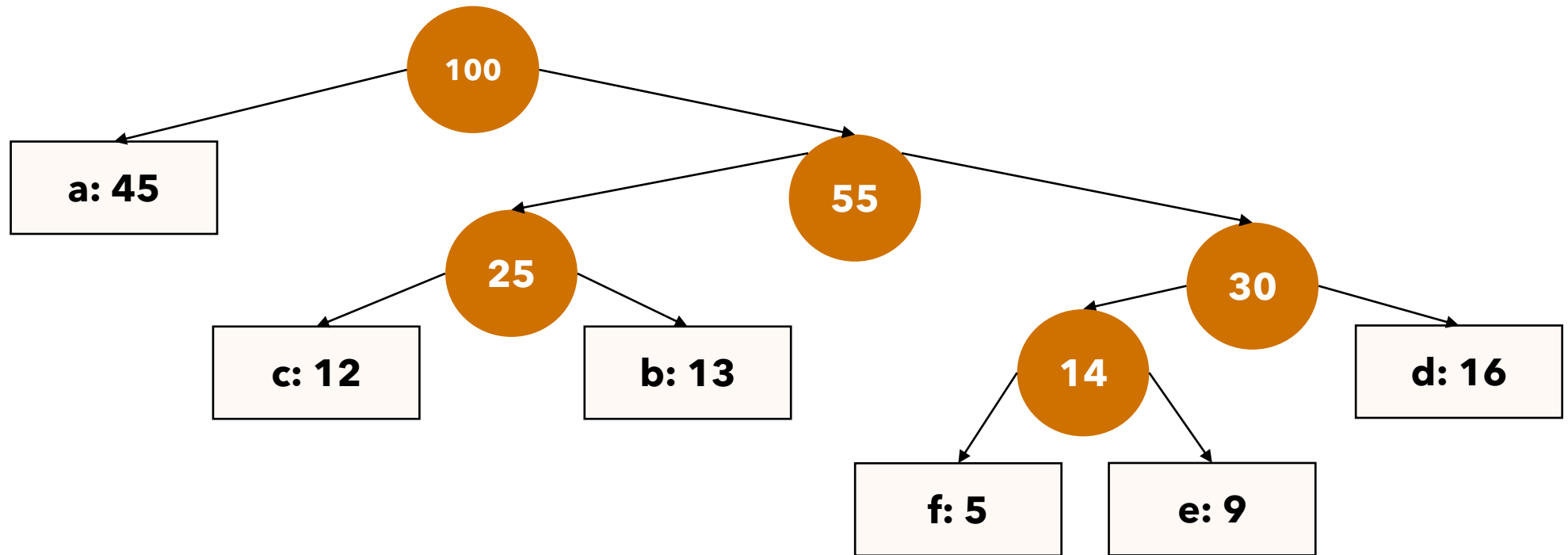
# L'algoritmo di Huffman



# L'algoritmo di Huffman

Il codice di un carattere è prodotto dal percorso dalla radice fino alla foglia corrispondente al carattere stesso. È sufficiente segnare i rami dell'albero così:

- ramo destro: **1**
- ramo sinistro: **0**



# Da vedere/provare a casa

- [CMPRSN \(Compression Overview\) – Computerphile](#)
- [Run-length encoding \(Python\)](#)
- [Huffman algorithm \(Python\)](#)