Huffman coding

In textual data, each character typically takes 8 (ASCII) or 16 (UTF-16) bits. Clearly, we should be able to save some space just by encoding text in a binary format.

Huffman coding is an elegant compression algorithm that combines ideas from RLE (i.e. it exploits frequency information) with binary encoding to save space. Huffman coding was invented by David Huffman in 1952.

Huffman coding

The idea is to generate a binary sequence that represents each character required. This might be the English alphabet, some subset of that, or any collection of symbols. Say we have a 100KB file made up of repetitions of the letters `a` to `f`.

We start by creating a frequency table:

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

Huffman coding

If we use a fixed-length code we can encode this data in about 37.5KB. If we use a variable-length code and assign the shortest code to the most frequently used characters, we can encode it in just 28KB.

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5
Code (fixed-length)	000	001	010	011	100	101
Code (variable-length)	0	101	100	111	1101	1100

Huffman coding

To send this data over the wire or store it in a file, we need to send/store the mapping from code to characters followed by the concatenation of all binary codes as they appear in the unencoded document.

Any sequence of codes must be unambiguous – we can only decode this at the other end if no code is a prefix of any other. If we had used `1` for `c` and `11` for `d`, how would we decode `111`?

Huffman coding

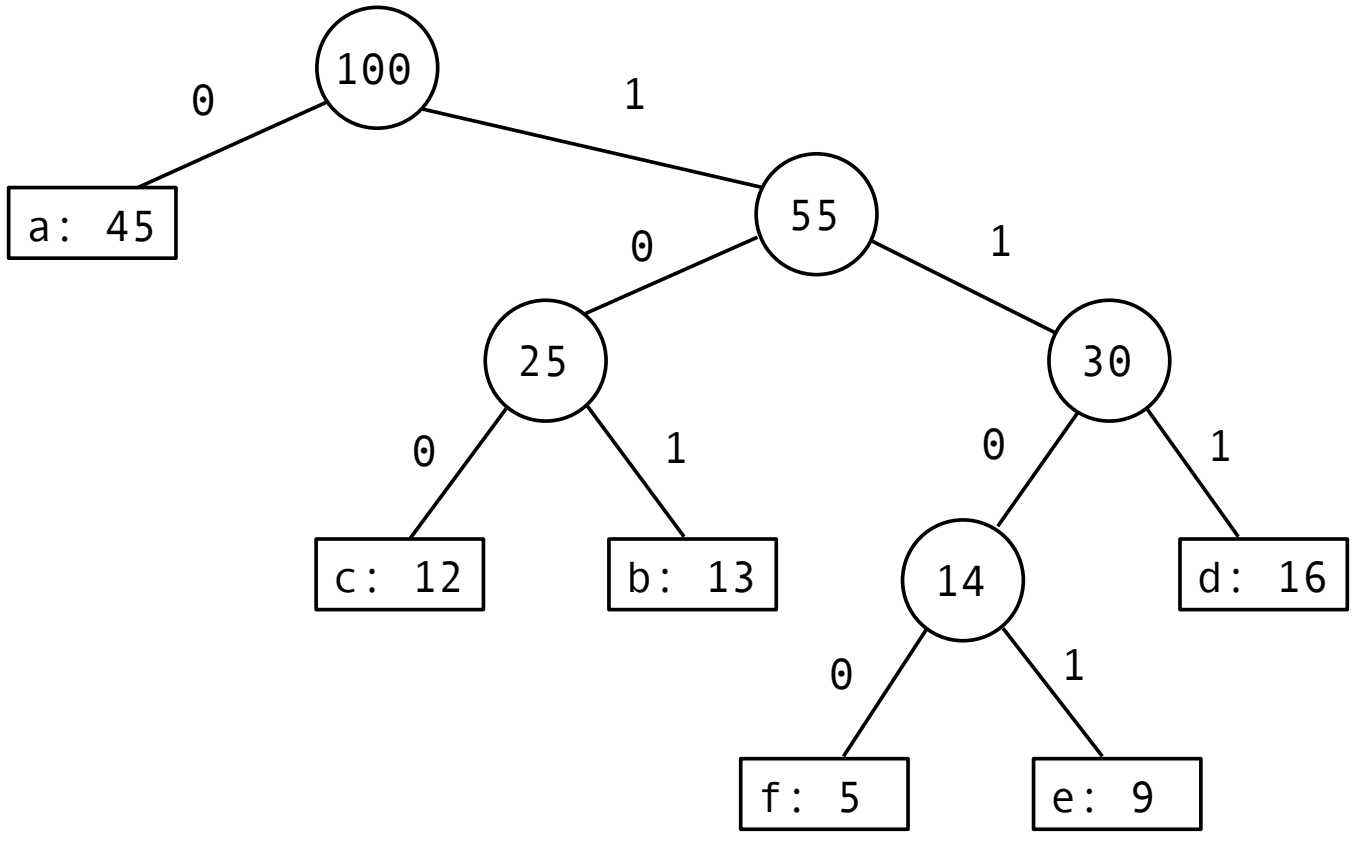
To send this data over the wire or store it in a file, we need to send/store the mapping from code to characters followed by the concatenation of all binary codes as they appear in the unencoded document.

Any sequence of codes must be unambiguous – we can only decode this at the other end if no code is a prefix of any other. If we had used `1` for `c` and `11` for `d`, how would we decode `111`?

	`a`	`b`	`c`	`d`	`e`	`f`
Code (variable-length)	0	101	100	111	1101	1100

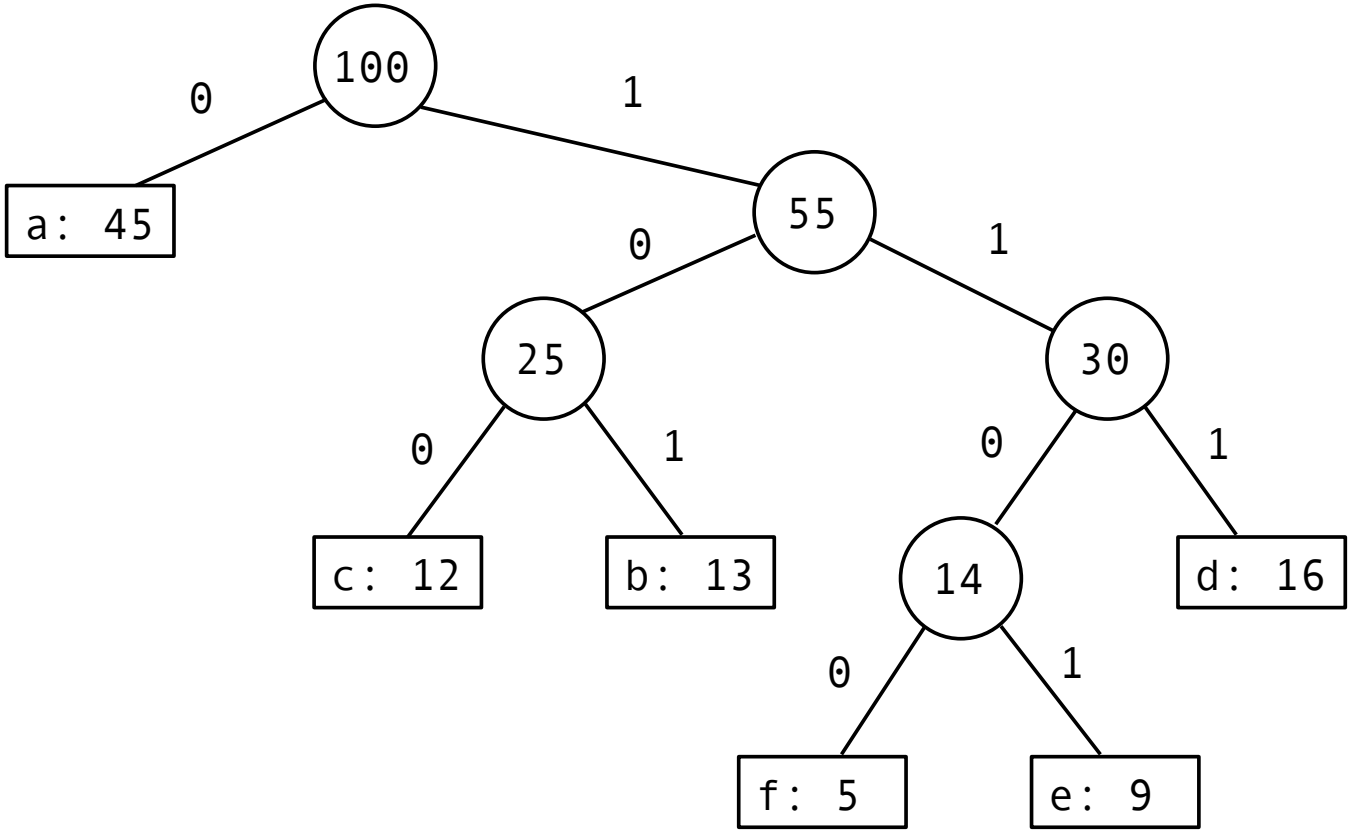
Huffman trees

We can create these variable-length codes using a binary tree (not a search tree). In a Huffman Tree the leaves contain the data, a character and its frequency. Internal nodes are labelled with the combined frequencies of their children.



Huffman trees

To decode data we go start at the root and go left for 0, and right for 1 until we get to a leaf. So, to decode 0101100, we start at the root, read an a, start at the root again, and so on to read abc. An optimal Huffman tree is full.

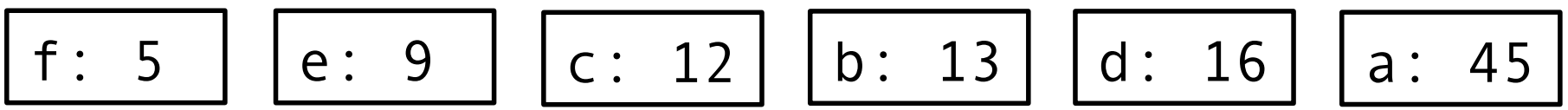


Creating the Huffman coding

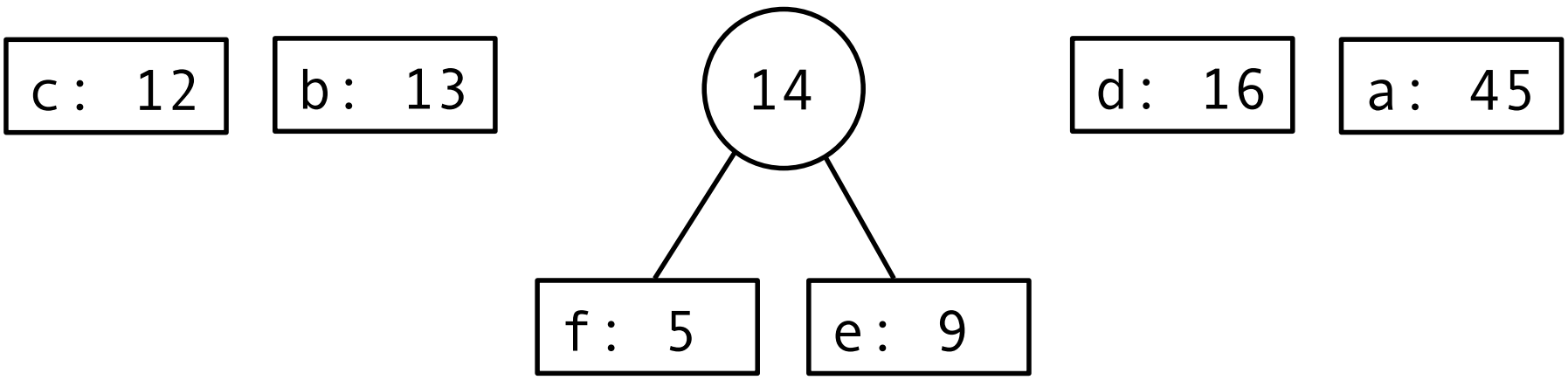
The most elegant part of this scheme is the algorithm used to create the tree:

1. Make a tree node object for each character, with an extra label for its frequency.
2. Put these nodes in a priority queue, where the lowest frequency has highest priority.
3. Repeatedly:
    1. Remove two nodes from the queue and insert them as children to a new node. The char label of the new node is blank and the frequency label is the sum of the labels of its children.
    2. Put the new node back in the queue.
    3. When there is only one item in the queue, that's the Huffman tree.

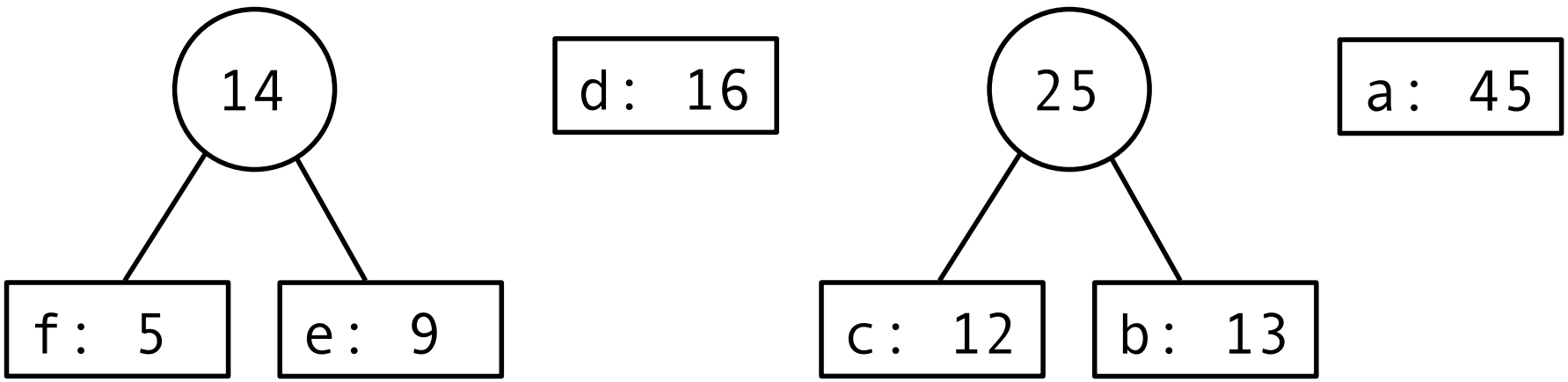
Creating the Huffman coding



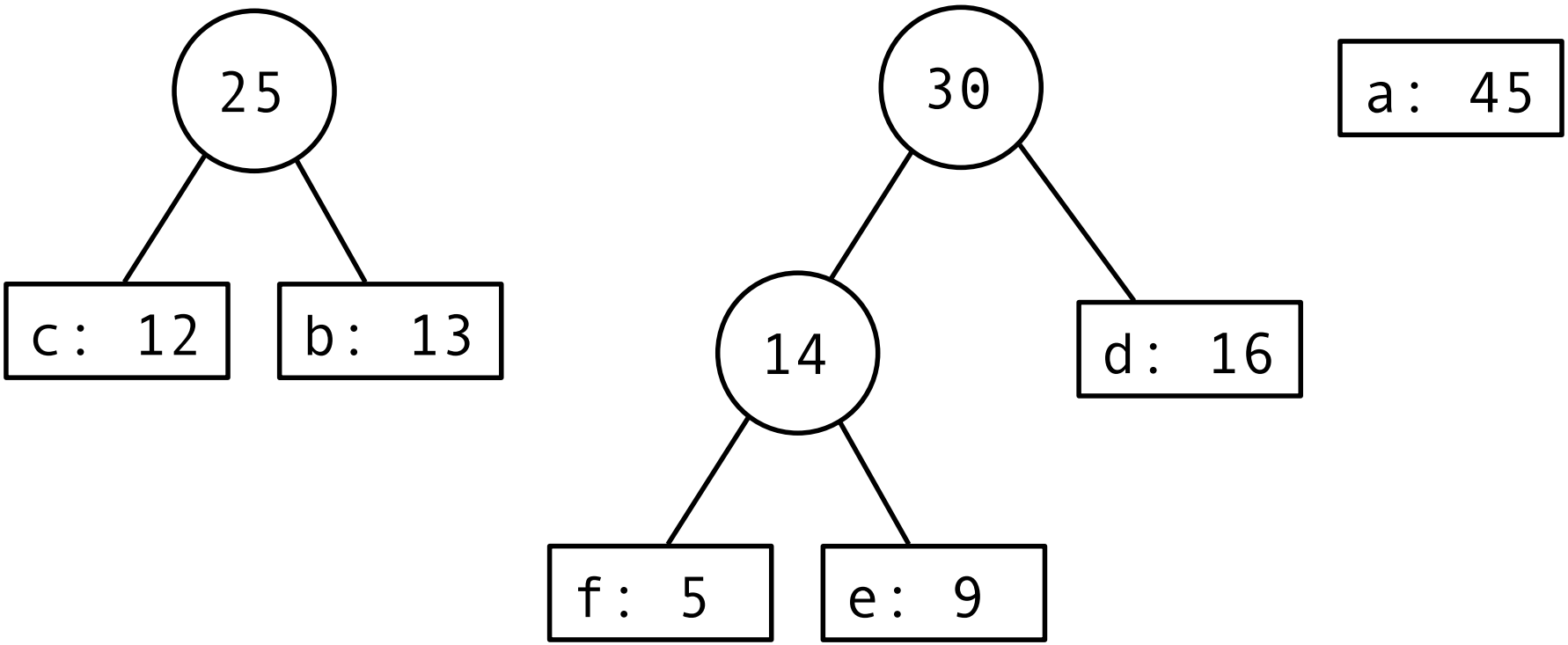
Creating the Huffman coding



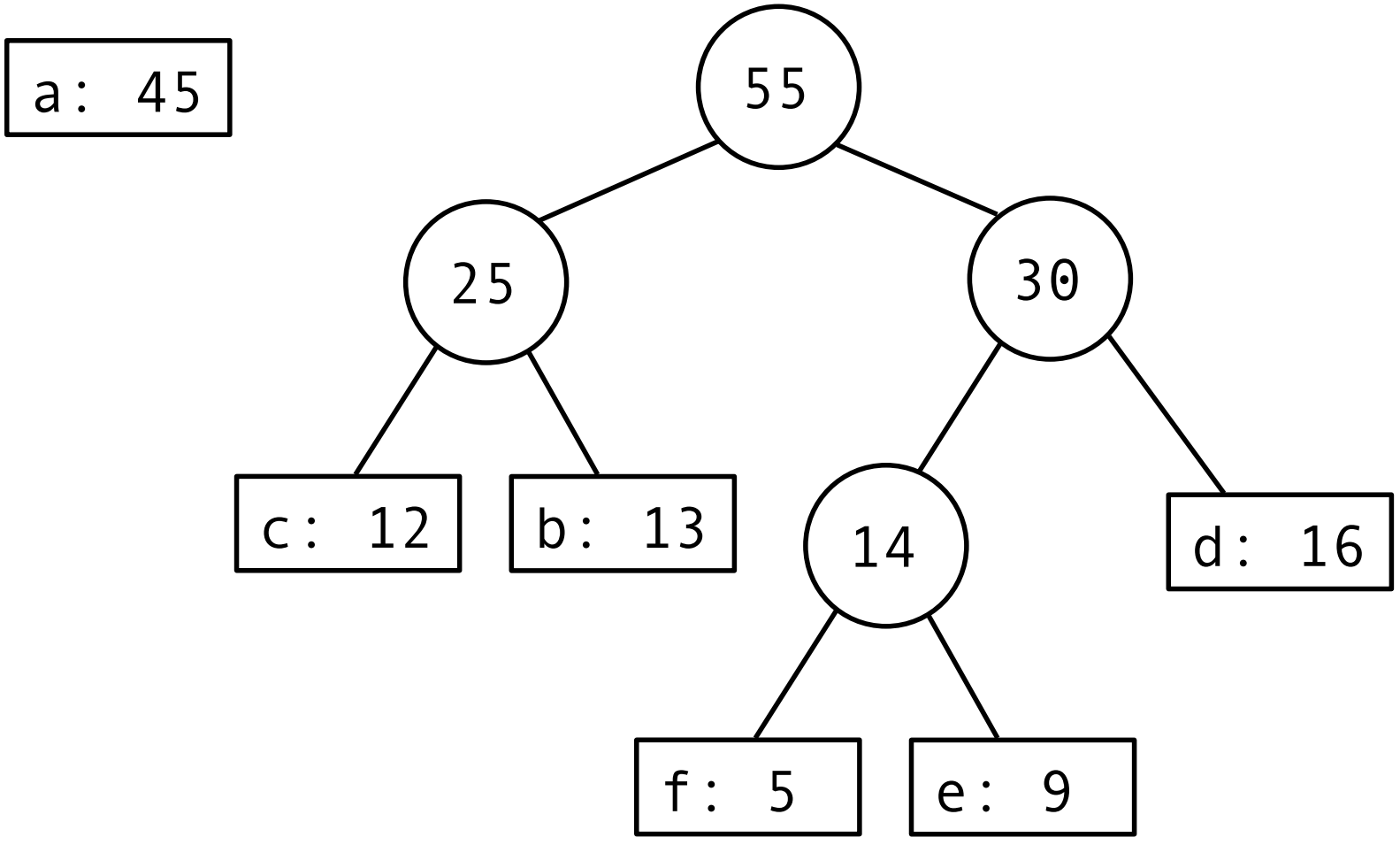
Creating the Huffman coding



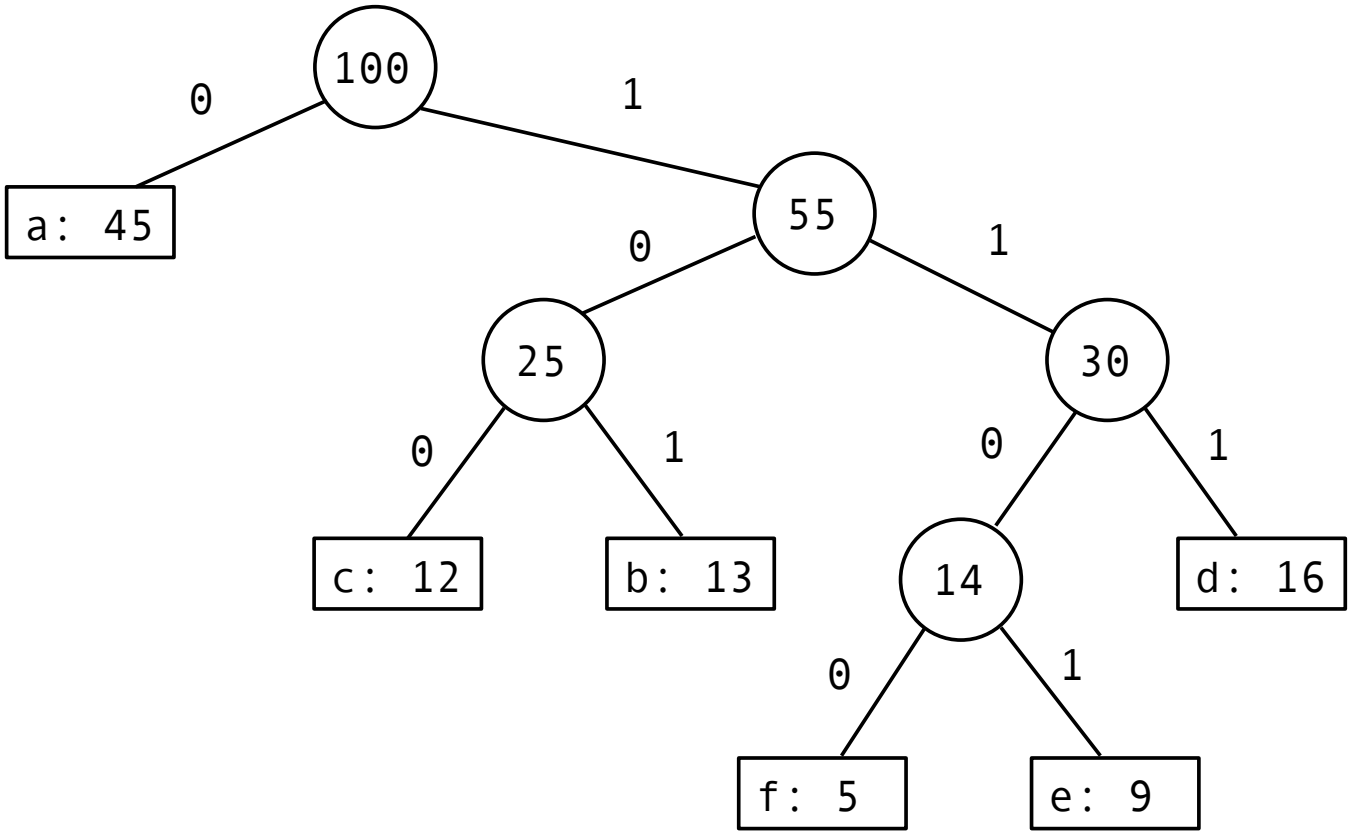
Creating the Huffman coding



Creating the Huffman coding



Creating the Huffman coding



Transmitting the Huffman coding

To use a Huffman coding as part of a compressed file format we begin the file with a "magic number" identifying the format used and the length of the alphabet, A, used by the file. We then store the table as an array in the next $|A|$ bits, followed by one code after another until the EOF character.

This scheme is used by some of the most widely used compressed formats such as GIF and ZIP.