4. Source Encoding Methods

- Called also
  - ***entropy coders***, because the methods try to get close to the entropy (i.e. lower bound of compression).
  - ***statistical coders***, because the methods assume the probability distribution of the source symbols to be given (either statically or dynamically) in the source model.
- The alphabet can be finite or infinite
- Sample methods:
  - Shannon-Fano coding
  - Huffman coding (with variations)
  - Tunstall coding
  - Arithmetic coding (with variations)

4.1. Shannon-Fano code

- First idea: Code length $l_i = \lceil \log_2 p_i \rceil$.
- This satisfies: $H(S) \leq L \leq H(S) + 1$
- Always possible, because Kraft inequality is satisfied:

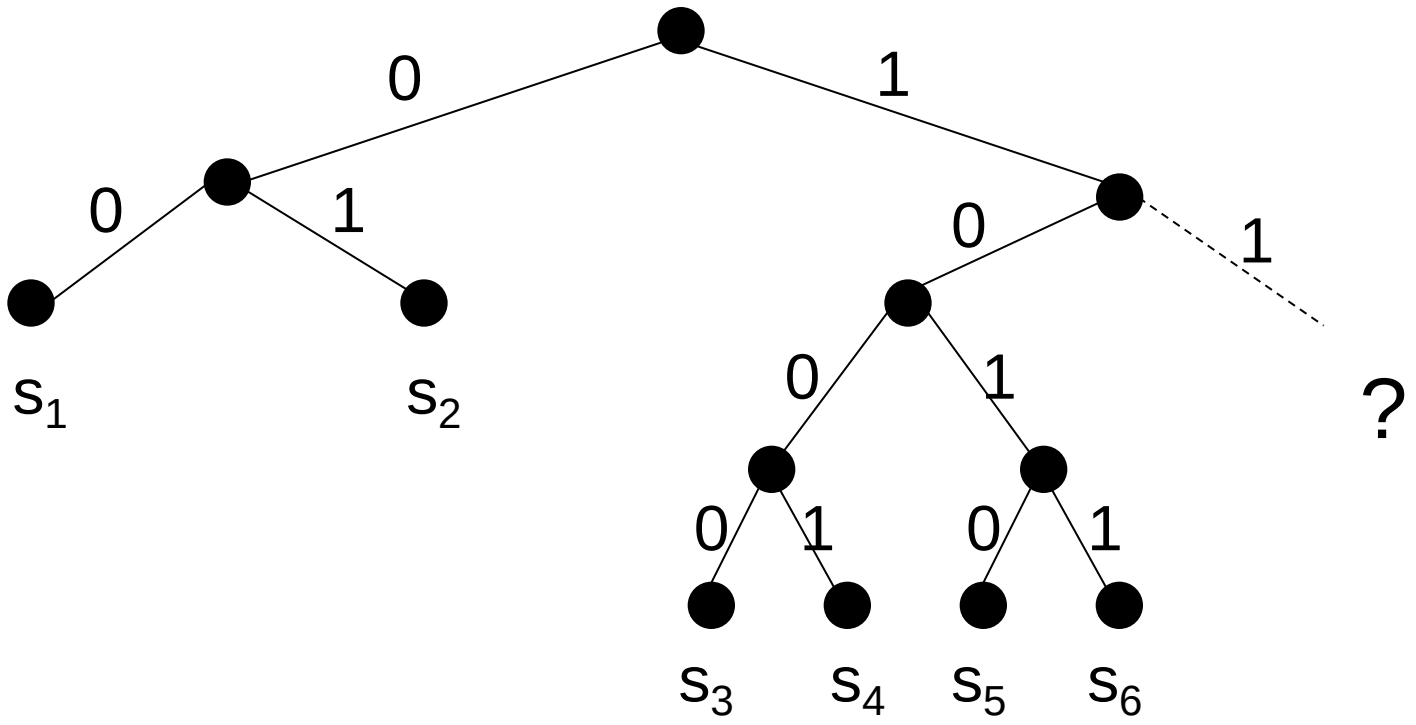
$$l_i \geq \log_2(1/p_i) \Rightarrow p_i \geq 1/2^{l_i} \Rightarrow \sum p_i \geq \sum \frac{1}{2^{l_i}} \Rightarrow 1 \geq \sum \frac{1}{2^{l_i}}$$

Problems:

- The decoding tree may not be complete (succinct).
- How to assign codewords?
- Shannon-Fano method solves these problems by balanced top-down decomposition of the alphabet.

Example

- $p_1 = p_2 = 0.3$: code lengths: $\lceil -\log_2 0.3 \rceil = 2$
- $p_3 = p_4 = p_5 = p_6 = 0.1$: code lengths: $\lceil -\log_2 0.1 \rceil = 4$
- E.g.



Algorithm 4.1.
Shannon-Fano codebook generation

Input: Alphabet $S = \{s_1, ..., s_q\}$, probability distribution $P = \{p_1, ..., p_q\}$, where $p_i \geq p_{i+1}$.

Output: Decoding tree for S.

begin

Create a root vertex r and associate alphabet S with it.

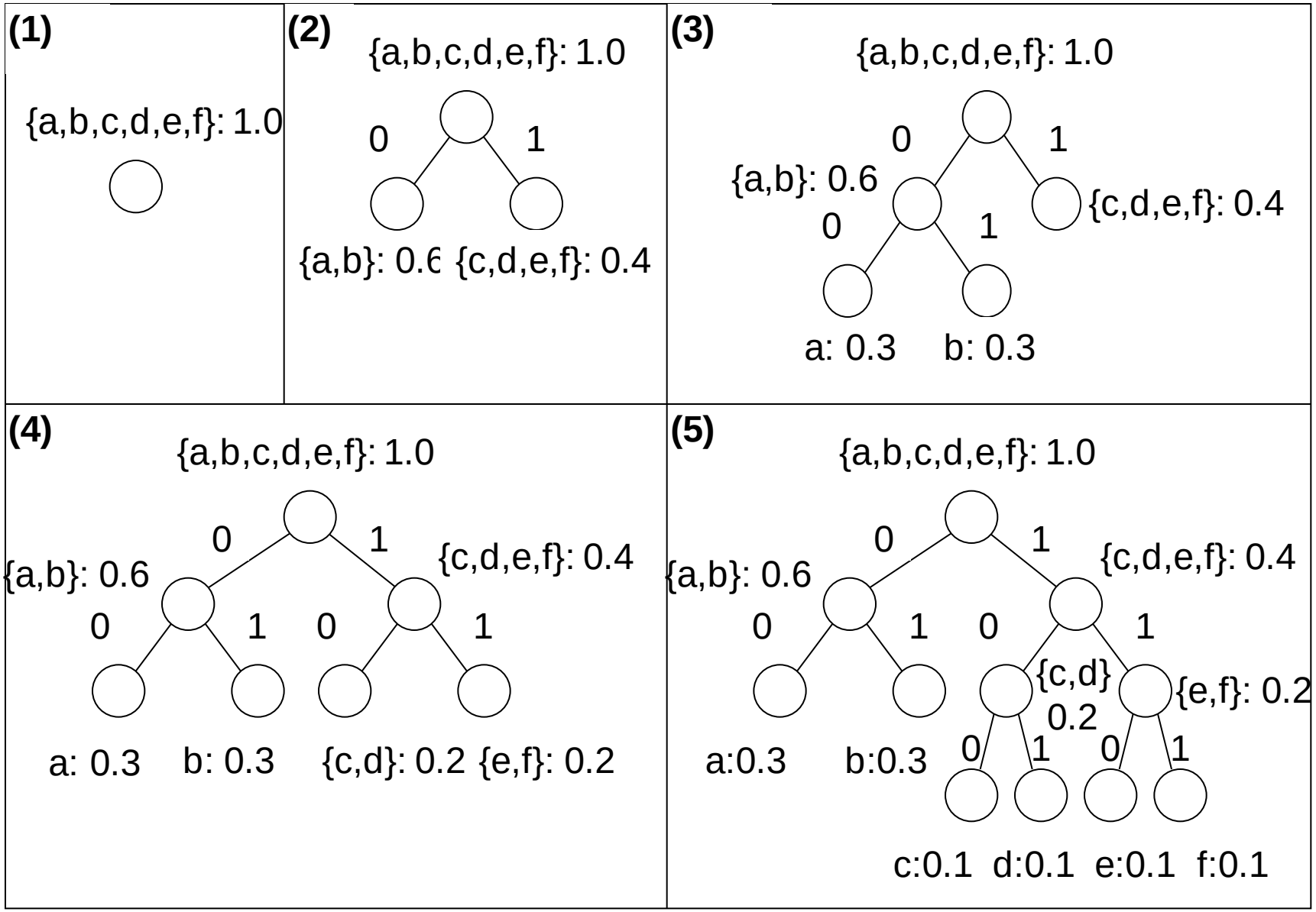
If S has only one symbol **then** return r.

Find j ($\neq 0$ and $\neq q$) such that $\sum_{i=1}^{j} p_i$ and $\sum_{i=j+1}^{q} p_i$ are the closest.

Find decoding trees r_1 and r_2 for the sub-alphabets $\{s_1, ..., s_j\}$ and $\{s_j+1, ..., s_q\}$ recursively and set them to subtrees of r, with labels 0 and 1.

Return the tree rooted by r.

end


(1)
{a,b,c,d,e,f}: 1.0
(2)
{a,b,c,d,e,f}: 1.0
0
1
{a,b}: 0.6
{c,d,e,f}: 0.4
(3)
{a,b,c,d,e,f}: 1.0
0
1
{a,b}: 0.6
{c,d,e,f}: 0.4
0
1
a: 0.3
b: 0.3
(4)
{a,b,c,d,e,f}: 1.0
0
1
{a,b}: 0.6
{c,d,e,f}: 0.4
0
1
0
1
a: 0.3
b: 0.3
{c,d}: 0.2
{e,f}: 0.2
(5)
{a,b,c,d,e,f}: 1.0
0
1
{a,b}: 0.6
{c,d,e,f}: 0.4
0
1
0
1
{c,d}
0.2
{e,f}: 0.2
a:0.3
b:0.3
0
1
0
1
c:0.1
d:0.1
e:0.1
f:0.1

4.2. Huffman code

- Best-known source compression method.
- Builds the tree bottom-up (contrary to Shannon-Fano).

Principles:

- Two least probable symbols appear as lowest-level leaves in the tree, and differ only at the last bit.
- A pair of symbols s_i and s_j can be considered a meta-symbol with probability p_i+p_i.
- Pairwise combining is repeated q-1 times.

Algorithm 4.2. Huffman codebook generation

Input: Alphabet $S = \{s_1, \ldots, s_q\}$, probability distribution $P = \{p_1, \ldots, p_q\}$, where $p_i \geq p_{i+1}$.

Output: Decoding tree for S.

begin

Initialize forest F to contain a one-node tree T_i for each symbol s_i and set $weight(T_i) = p_i$.

while $|F| > 1$ **do**

begin

Let X and Y be two trees with the lowest weights.

Create a binary tree Z, with X and Y as subtrees (equipped with labels 0 and 1).

Set $weight(Z) = weight(X) + weight(Y)$.

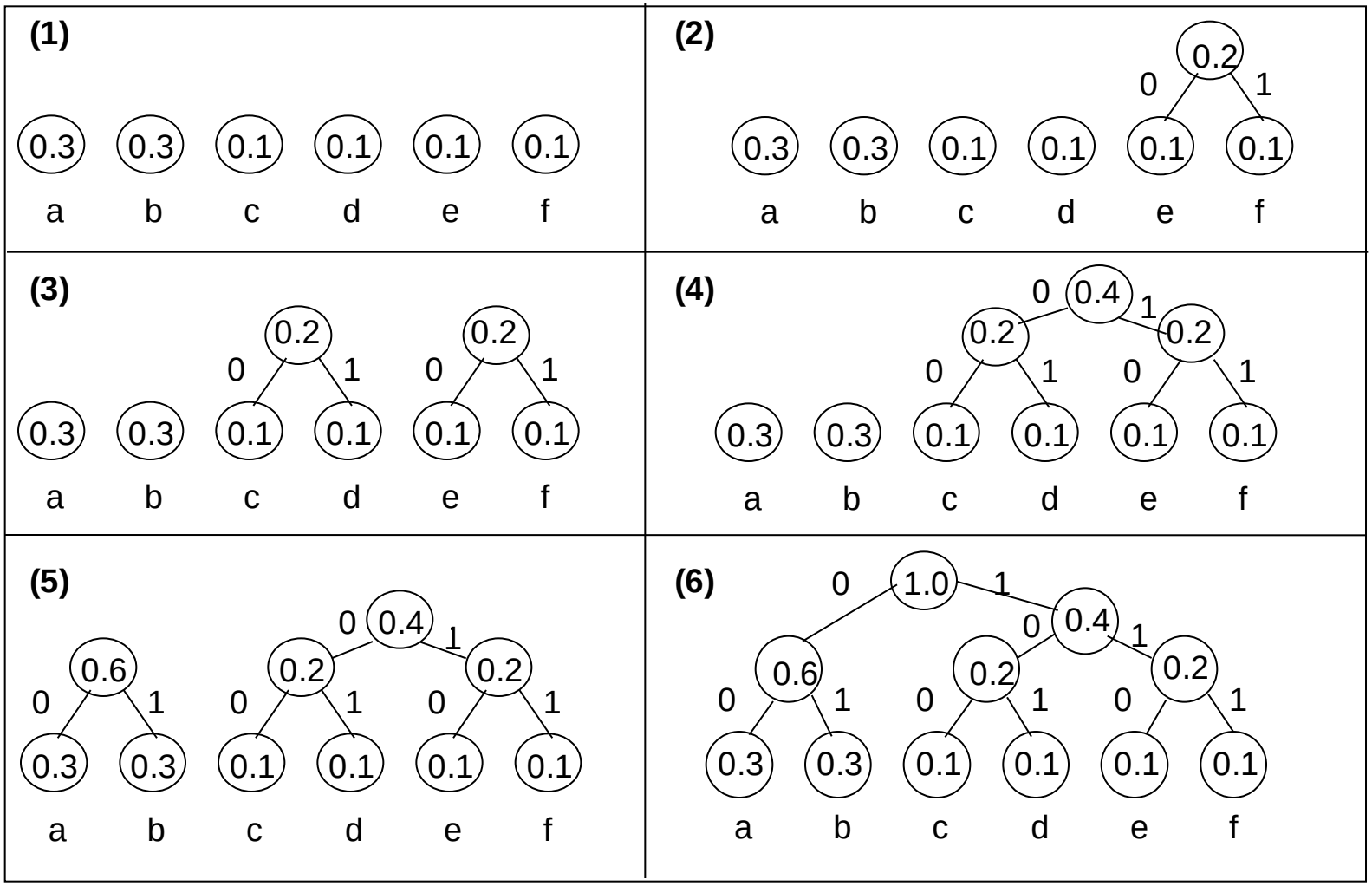
Add Z to forest F and remove X and Y from it.

end

Return the single remaining tree of forest F.

end

Example of Huffman codebook generation



Properties of Huffman code

- Produces an optimal codebook for the alphabet, assuming that the symbols are independent.
- The average code length reaches the lower bound (entropy) if for all i: $p_i = 2^{-k}$ where k is an integer.
- Generally: $H(S) \leq L \leq H(S)+p_1+0.086$, where p_1 is the largest symbol probability.
- The codebook is not unique:
  (1) Equal probabilities can be combined using any tie-break rule.
  (2) Bits 0 and 1 can be assigned to subtrees in either order.

Implementation alternatives of Huffman code

1. Maintain a *min-heap*, ordered by weight;
   the smallest can be extracted from the root.
   The complexity of building the tree: $O(q)$,
   inserting a metasymbol: $O(\log q)$; altogether $O(q \log q)$.

2. Keep the uncombined symbols in a list sorted by weight, and maintain a *queue* of metasymbols.
   - The two smallest weights can be found from these two sequences
   - The new (combined) metasymbol has weight higher than the earlier ones.
   - Complexity: $O(q)$, if the alphabet is already sorted by probability.

Special distributions for Huffman code

- All symbols equally probable, $q = 2^k$, where k is integer: *block code*.
- All symbols equally probable, no k such that $q = 2^k$: *shortened block code*.
- Sum of two smallest probabilities > largest: (shortened) block code.
- Geometric (≈ negative exponential) distribution: $p_i = c \cdot 2^{-i}$: codewords 0, 10, 110 , ..., 111..10, 111..11 (cf. unary code).
- *Zipf distribution*: $p_i \approx c/i$ (symbols s_i sorted by probability): compresses to about 5 bits per character for normal text.

Transmission of the codebook

- Drawback of (static) Huffman coding:
  The codebook must be stored/transmitted to the decoder
- Alternatives:
  - Shape of the tree ($2q$-1 bits) plus leaf symbols from left to right ($q \lceil \log_2 q \rceil$ bits).
  - Lengths of codewords in alphabetic order (using e.g. universal coding of integers); worst case $O(q \log_2 q)$ bits.
  - Counts of different lengths, plus symbols in probability order; space complexity also $O(q \log_2 q)$ bits.

Extended Huffman code

Huffman coding does not work well for:

- Small alphabet
- Skew distribution
- Entropy close to 0, average code length yet ≥ 1.

Solution:

- Extend the alphabet to $S(n)$:
  Take n-grams of symbols as units in coding.
- Effect: larger alphabet (q^n), decreases the largest probability.

Extended Huffman code (cont.)

- Information theory gives:
  $H(S(n)) \leq L(n) \leq H(S(n)) + 1$

- Counted per original symbol:
  $$H(S(n))/n \leq L \leq (H(S(n)) + 1)/n$$
  which gives (by independence assumption):
  $$H(S) \leq L \leq H(S) + 1/n$$

- **Thus**: Average codeword length approaches the entropy.
- **But:** The alphabet size grows exponentially, most of the extended symbols do not appear in messages for large n.
- **Goal:** No explicit tree; codes determined on the fly.

Adaptive Huffman coding

- **Normal Huffman coding**: Two phases, static tree
- **Adaptive compression:** The model (& probability distribution) changes after each symbol; encoder and decoder change their models intact.
- **Naive adaptation:** Build a new Huffman tree after each transmitted symbol, using the current frequencies.
- **Observation:** The structure of the tree changes rather seldom during the evolution of frequencies.
- **Goal:** Determine conditions for changing the tree, and the technique to do it.

Adaptive Huffman coding (cont.)

- **Sibling property:** Each node, except the root, has a sibling (i.e. the binary tree is complete).

- The tree nodes can be listed in non-decreasing order of weight so that each node is adjacent in the list to its sibling.

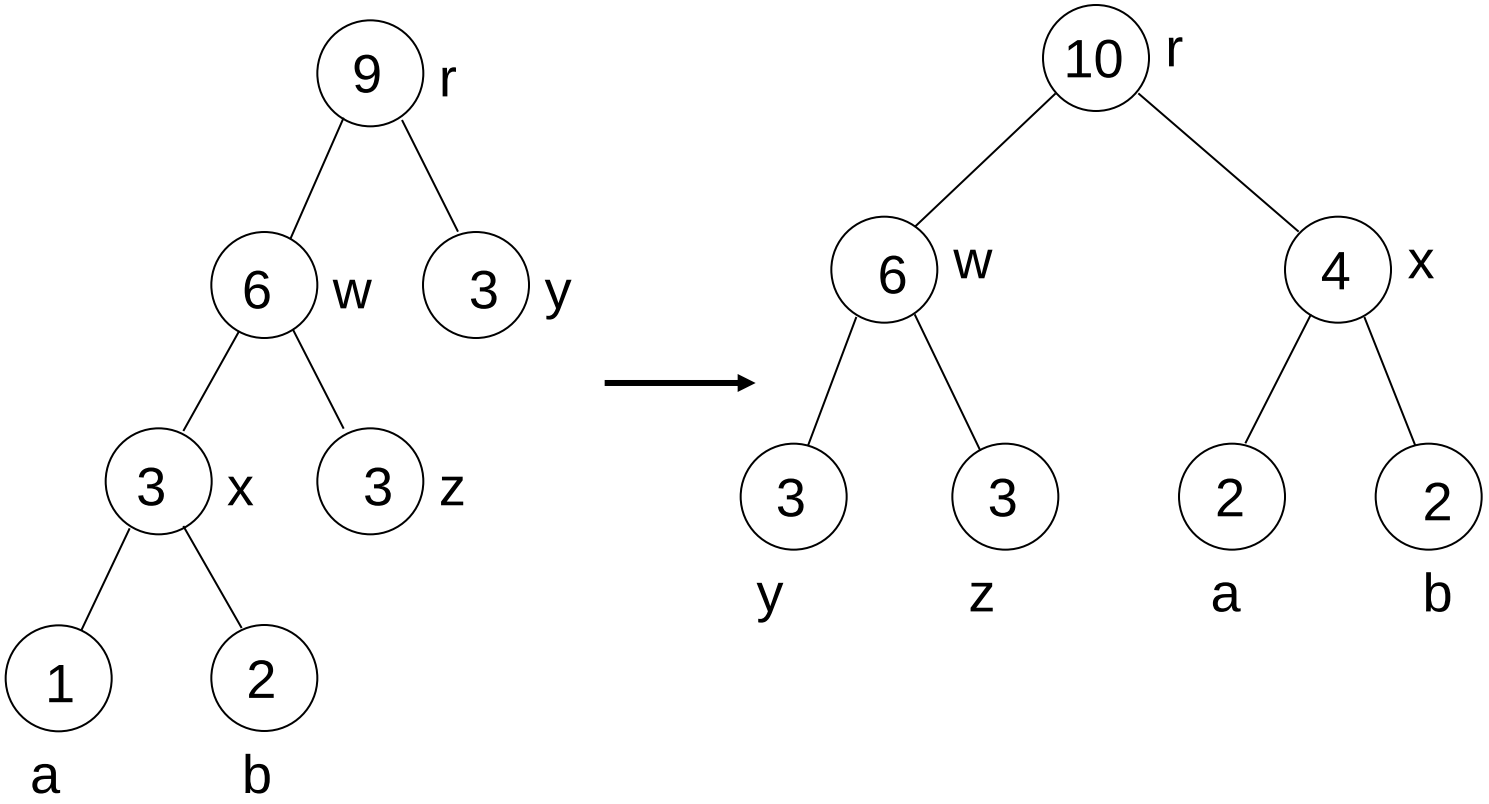
- **Theorem.** A binary tree having weights associated with its nodes, as defined above, is a Huffman tree if and only if it has the sibling property.
  Proof. Skipped.

Implementation of Adaptive Huffman coding

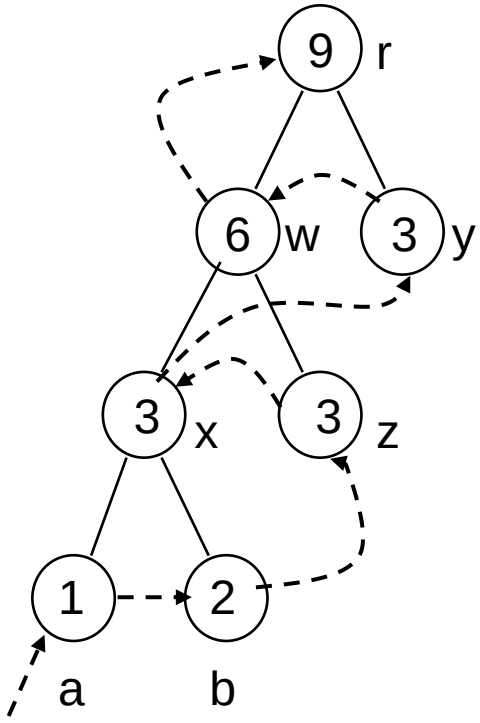
- Start from a balanced tree with weight = 1 for each leaf; the weight of an internal node = sum of child weights.
- Maintain a threaded list of tree nodes in increasing order of weight.
- Nodes of equal weight in the list form a (virtual) *block*.
- After transmitting the next symbol, add one to the weights of nodes on the path from the correct leaf up to the root.
- Increasing a node weight by one may violate the increasing order within the list.
- Swapping of violating node with the *rightmost node in the same block* will recover the order, and maintains the sibling property. Addition of frequencies continues from the new parent.

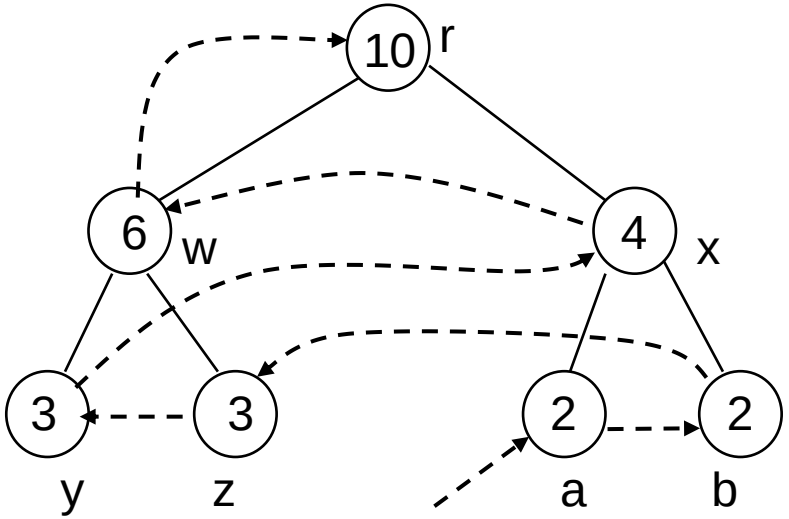
Example of Huffman tree evolution

Increase the weight of ’a’ from 1 to 2:



Example step in adaptive Huffman coding





Notes about Adaptive Huffman coding

Modification:

- Start from an empty alphabet, and a tree with only a *placeholder*.
- At the first occurrence of a symbol, transmit the placeholder code and symbol as such, insert it to the tree by splitting the placeholder node.

Further notes:

- Complexity proportional to the number of output bits.
- Compression power close to static Huffman code.
- Not very flexible in context-dependent modelling.

Canonical Huffman coding

- Goal: effective decoding
- Based on *lengths* of codewords, determined by the normal Huffman algorithm.
- Chooses one of the many possible bit assignments for codewords, e.g.

Symbol	Freq.	Code I	Code II	Code III
a	10	000	111	000
b	11	001	110	001
c	12	100	011	010
d	13	101	010	011
e	22	01	10	10
f	23	11	00	11

Canonical Huffman coding (cont.)

Definition. A Huffman code is any prefix-free assignment of codewords, the lengths of which are equal to the depths of corresponding symbols in a Huffman tree.

Ordering of codeword values:

- From longest to shortest
- Same-length codewords have successive code values
- *k*-bit prefix is smaller than any *k*-bit codeword, i.e. *lexicographic order*

Decoding needs:

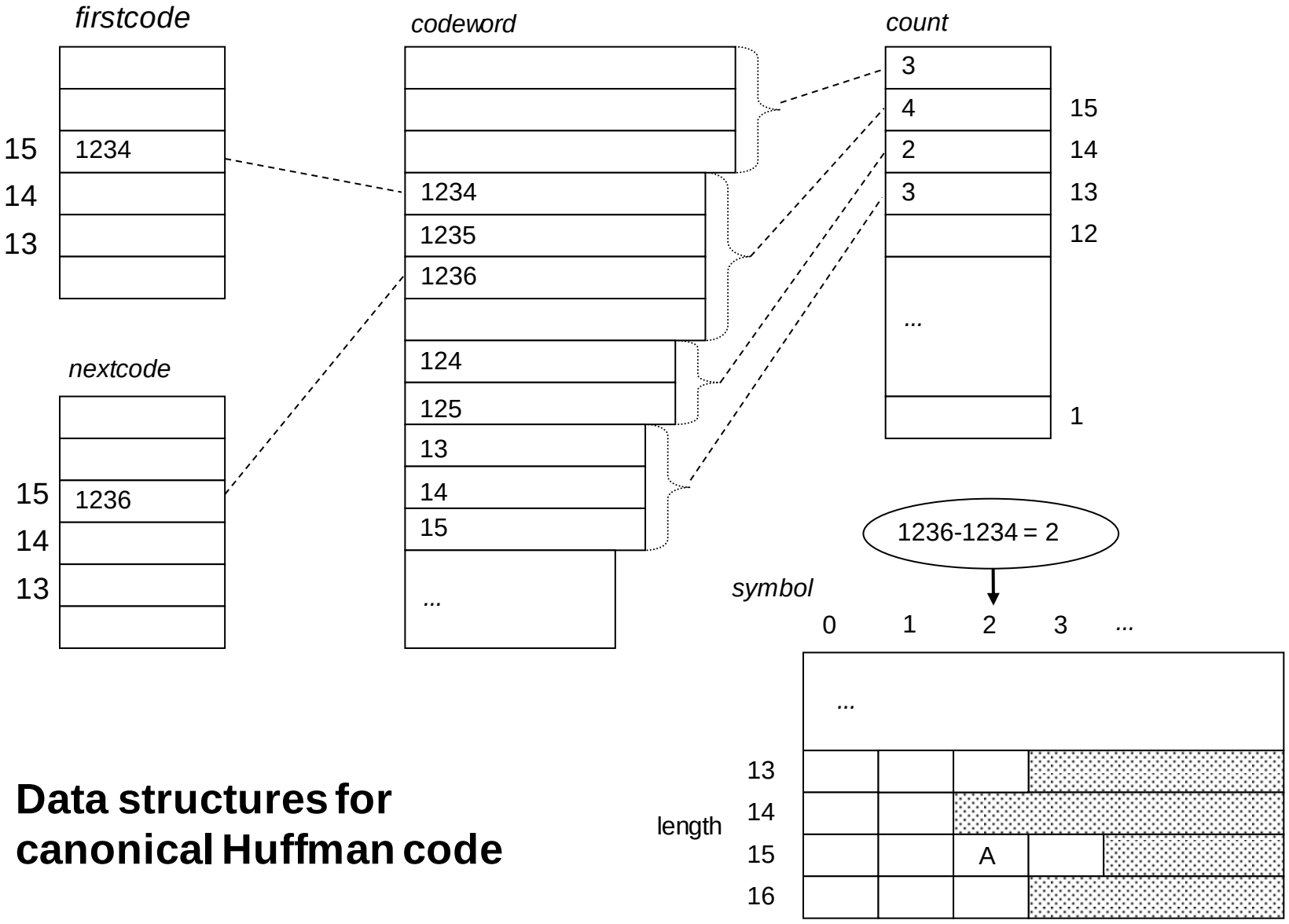
- The first code value for each length.
- The symbol related to the *i*'th value within the same-length codewords.

Algorithm 4.3.:
Assignment of canonical Huffman codewords

Input: Length l_i for each symbol s_i of the alphabet, determined by the Huffman method.

Output: Integer values of codewords assigned to symbols, plus the order number of each symbol within same-length symbols.

```
begin
    Set maxlength := Max{l_i}
    for l := 1 to maxlength do
        Set countl[l] := 0
    for i := 1 to q do
        Set countl[l_i] := countl[l_i] + 1
    Set firstcode[maxlength] := 0
    for l := maxlength – 1 downto 1 do
        Set firstcode[l] := (firstcode[l+1] + countl[l+1] ) / 2
    for l := 1 to maxlength do
        Set nextcode[l] := firstcode[l]
    for i := 1 to q do
    begin
        Set codeword[i] := nextcode[l_i]
        Set symbol[l_i, nextcode[l_i] – firstcode[l_i] ] := i
        Set nextcode[l_i] := nextcode[l_i] + 1
    end
end
```



Data structures for canonical Huffman code

Algorithm 4.4.
Decoding of canonical Huffman code.

Input: The numerical value of the first code for each codeword length, plus the symbol for each order number within the set of codewords of equal length.

Output: Decoded symbol.

begin

Set *value* := **readbit**()

Set *l* := 1

while *value* < *firstcode*[*l*] **do**

begin Set *value* := 2 ∗ *value* + **readbit**()

Set *l* := *l* + 1

end

return *symbol*[*l*, *value* − *firstcode*[*l*]]

end

Properties of canonical Huffman code

- Small amount of memory for the model in decoding: *firstcode* for each different length, and *symbol* table to look up the symbol related to a codeword value.
- Decoding is very fast: no walking in the tree; only a very simple loop for each transmitted bit.

Tunstall coding

- **Goal:** *Variable-length* substrings of the source are encoded to *fixed-length* codewords.
- **Assumption:** Independence of symbols: probability of a string = product of included symbol probabilities.
- **Idea:** For codeword length k, we try to find $\leq 2^k$ approximately equi-probable blocks of symbols.
- **Restrictions:**
  1. It must be possible to parse any message using the selected blocks.
  2. The set of blocks has the *prefix-free* property.
- **(1) and (2) together:** The parsing trie must be a complete *q*'ary tree.

Tunstall's ideas

- Build a parsing trie where each parent-child relationship represents a symbol.
- The symbols on the path from the root to a leaf represent the block which is assigned a codeword.
- Each node has a weight = probability of related path.
- The number of leaves must be $\leq 2^k$.
- Build the trie top-down.
- At each step, extend the leaf having the highest weight with q child nodes, one for each symbol.

Algorithm 4.5: Tunstall codebook generation

- *Input*: Symbols s_i, $i = 1, ..., q$ of the source alphabet S, symbol probabilities p_i, $i = 1, ..., q$, and the length k of codewords to be allocated.

- *Output*: Trie representing the substrings of the extended alphabet, with codewords 0, ..., 2^k-u attached to the leaves ($0 \leq u \leq q - 2$), plus the decoding table.

begin

Initialize the trie with the root and q first-level
nodes, with labels s_1, ... s_q, and weights p_1, ..., p_q.

$n := 2^k - q$ -- Number of remaining codewords

while $n \geq q - 1$ **do**

Find leaf x from the trie having the biggest weight among leaves.

Add q children to x, with labels s_1, ... s_q, and weights
$weight(x) \cdot p_1$, ..., $weight(x) \cdot p_q$.

Set $n := n - q + 1$

end

for each leaf l_i in preorder, $i = 0, 1, 2, ...$ **do**

Assign $codeword(l_i) := i$ (using k bits).

Denote $path(l_i)$ = labels from the root to l_i.

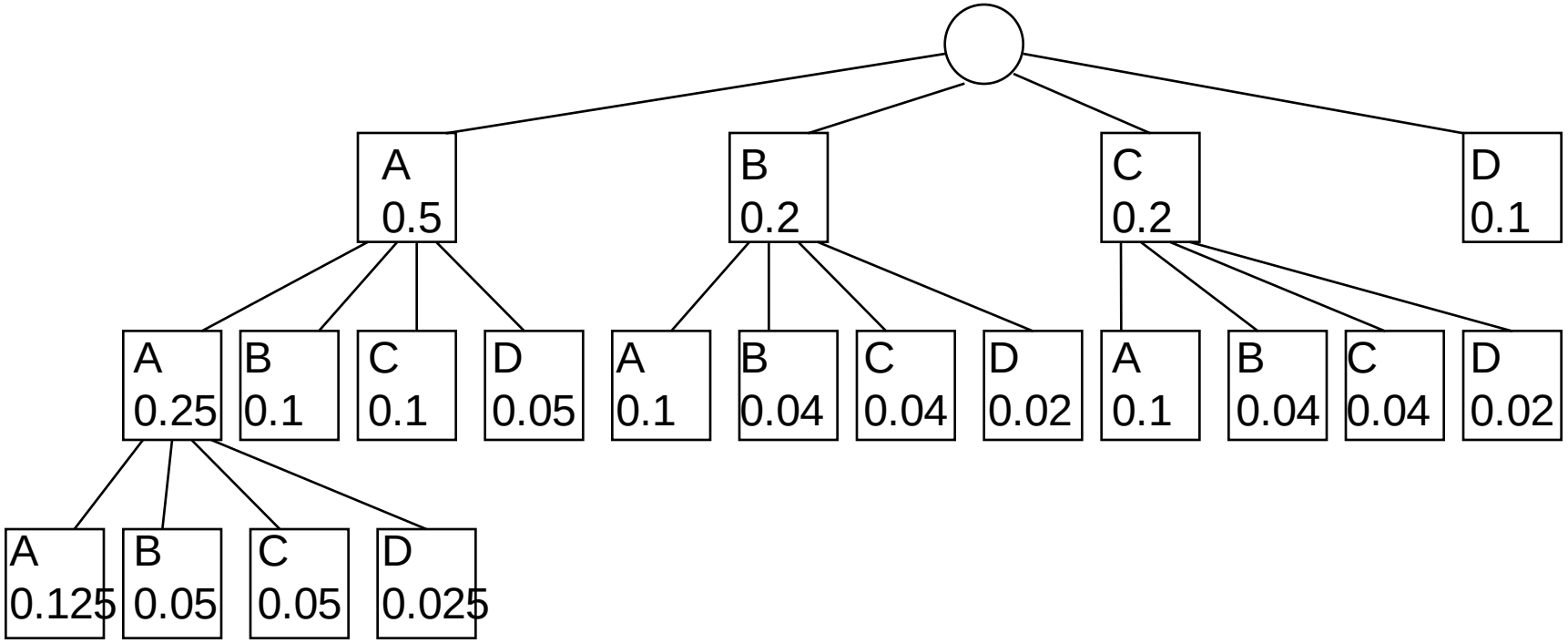
Add pair $(i, path(l_i))$ to the decoding table.

end

end

Tunstall code example:

S = {A, B, C, D}, P = {0.5, 0.2, 0.2, 0.1}, k = 4, 2^k=16



0000 → AAA	0100 → AB	1000 → BB	1100 → CB
0001 → AAB	0101 → AC	1001 → BC	1101 → CC
0010 → AAC	0110 → AD	1010 → BD	1110 → CD
0010 → AAD	0111 → BA	1011 → CA	1111 → D

Properties of Tunstall code

- Number of unused codewords:

$$u = 2^k - (q-1)\left\lfloor \frac{2^k - 1}{q-1} \right\rfloor - 1$$

- Average number of bits per input symbol:

$$\frac{k}{\text{Average path length}} = \frac{k}{\sum_{path \in Trie} (P(path) Length(path))}$$

- Not necessarily optimal