Data Structures

November 16

Objectives

Discuss the following topics:

- **Data Compression and Huffman Codes**

Data Compression and Huffman Codes

- Encoding symbols using bits
  - Ultimately computers operate on sequences of bits → need encoding schemes that take a rich alphabet and converts into bits
- Simplest scheme: use a *fixed* number of bits for each symbol in the alphabet, and then just concatenate bit strings for each symbol in the text.
- Example of encoding with fixed number of bits: 26 letters, space (to separate words), comma, period, question mark, exclamation mark, and apostrophe; total 32 symbols, can do with 5 bits.

Data Compression and Huffman Codes

- Ascii works similarly
- Can you do better for the case of 32 symbols? Can't do with 4 bits – can only distinguish 16 items
- BUT: maybe over large stretches of text we can spend on ***average*** less than 5 bits per symbol? e, t, a, o, i, n get to be use much more frequently in English than q,j,x,z (more than an order of magnitude)
- Could use a small number of bits for the frequent letters, and a large number of bits for the less frequent ones – hopefully on average less than 5 bits on a long string of text
- Reduce average number of bits per letter: fundamental problem in Datacompression

Data Compression and Huffman Codes

- How do you construct an encoding which in *optimal* way takes advantage of *nonuniform* frequencies of letters?
- Discussion of message transfer
- Morse Code
- Prefix Codes
- Ambiguity in Morse code: pairs of letters where the bit string that encodes one letter is a prefix of the bit string that encodes another
- Map letters to bitstrings so no encoding is prefix of any other

Data Compression and Huffman Codes

- A prefix code for a set S of letters is function γ that maps each letter $x \in S$ to some sequence of zeros and ones such that for different $y, x \in S$, the sequence γ(x) is not a prefix of γ(y).
- Works: consider a text consisting of a sequence of letters $x_1x_2x_3\ldots x_n$. Convert this to a sequence of bits by simply encoding each letter using γ and then concatenating this: $\gamma(x_1)\,\gamma(x_2)\,\gamma(x_3)\ldots\,\gamma(x_n)$.

Data Compression and Huffman Codes

- Decoding?
  - Scan bits from left to right
  - As soon as you see enough bits to match encoding of some letter, output this letter as the first letter of the text -- must be correct! No shorter or longer prefix could encode any other letter
  - Delete corresponding set of bits from the front of the message and iterate
- No need for pauses etc

Data Compression and Huffman Codes

- For prefix code we have
  - Each code word corresponds to exactly one symbol
  - No look ahead is required
- Example 1: S = {A, B, C}
  - Code1: $\gamma_1(A)=1$; $\gamma_1(B)=0$; $\gamma_1(C)=10$.
  - Code2: $\gamma_2(A)=1$; $\gamma_2(B)=00$; $\gamma_2(C)=10$.
  - Code3: $\gamma_3(A)=11$; $\gamma_3(B)=10$; $\gamma_3(C)=01$.
- Requirements:
- Each code word corresponds to exactly one symbol
- No look ahead is required
- Length of a codeword for a given symbol m_j should not exceed the length of the codeword of a less probable symbol m_i: $P(m_i) \leq P(m_j) \Rightarrow \text{length}(m_i) \geq \text{length}(m_i)$

Data Compression and Huffman Codes

- Length of a codeword for a given symbol m_j should not exceed the length of the codeword of a less probable symbol m_i: $P(m_i) \leq P(m_j) \Rightarrow \text{length}(m_i) \geq \text{length}(m_j)$
- In optimal encoding system, no unused short codewords either as stand-alone encodings or as prefixes of longer codewords (means longer codewords were created unnecessarily)
- Example: 01,000,001,100,101 for a certain set of 5 symbols: 11 is not used; → optimal coding: 01,10,11,000,001.

Data Compression and Huffman Codes

- Another example of prefix code: S={a,b,c,d,e}.γ_1(a)=11; γ_1(b)=01; γ_1(c)=001; γ_1(d)=10; γ_1(e)=000; cecab → 0010000011101

If recipient knows γ_1, then can decode unambiguously.

- ***Towards Optimal Prefix Codes*** Suppose for each $x \in S$ there is a frequency f_x, representing the fraction of the letter in the text that are equal to x. (Total number of letters in the text is n, then nf_x of these letters equal x). We have: $\sum_{x \in S} f_x = 1$.
- Encoding length of a given text using encoding γ ($|\gamma(x)|$ denotes the length of $\gamma(x)$):
- **encoding length=$\sum_{x \in S} nf_x |\gamma(x)| = n\sum_{x \in S} f_x |\gamma(x)|$.**
- **average number of bits per letter = $\sum_{x \in S} f_x |\gamma(x)|$.**

Data Compression and Huffman Codes

- Our earlier example of prefix code: S={a,b,c,d,e}.γ_1(a)=11; γ_1(b)=01; γ_1(c)=001; γ_1(d)=10; γ_1(e)=000;

Their frequencies: f_a=0.32; f_b=0.25; f_c=0.20; f_d=0.18; f_e=0.05;

Average number of bits per letter using prefix code γ_1 is:

0.32·2+ 0.25·2 +0.20·3+ 0.18·2+ 0.05·3 = 2.25

γ_1 is not the best:

γ_2(a)=11; γ_2(b)=10; γ_2(c)=01; γ_2(d)=001; γ_2(e)=000; is better

0.32·2+ 0.25·2 +0.20·2+ 0.18·3+ 0.05·3 = 2.23

Data Compression and Huffman Codes

Problem:

- Given an alphabet S and a set of frequencies for the letters, produce a prefix code γ such that average number of bits

$$ABL(\gamma) = \sum_{x \in S} f_x \, |\gamma(x)|$$

is as small as possible.

- Such a code is called *optimal*

Data Compression and Huffman Codes

Representing prefix codes using binary trees

Consider a (rooted) binary tree T such that the number of leaves is equal to the size of the alphabet S, label each leaf with a distinct letter in S.

Such a T naturally describes a prefix code: for each letter $x \in S$ follow the path from the root to the leaf labeled with x; each time you go from a node to its left child write down a 0 and in case you go to the right, write down a 1.

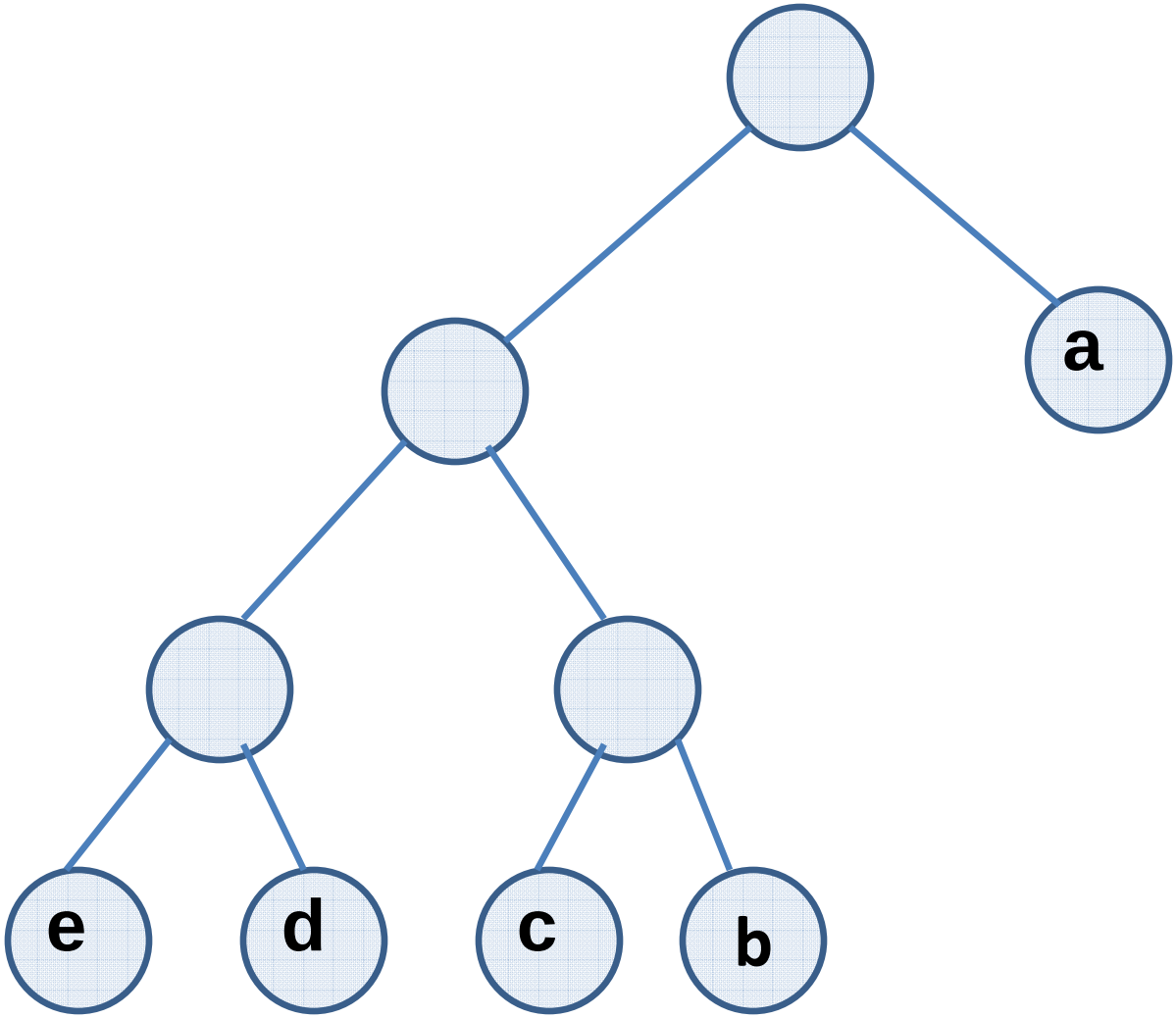
The encoding of S constructed from T is a prefix code

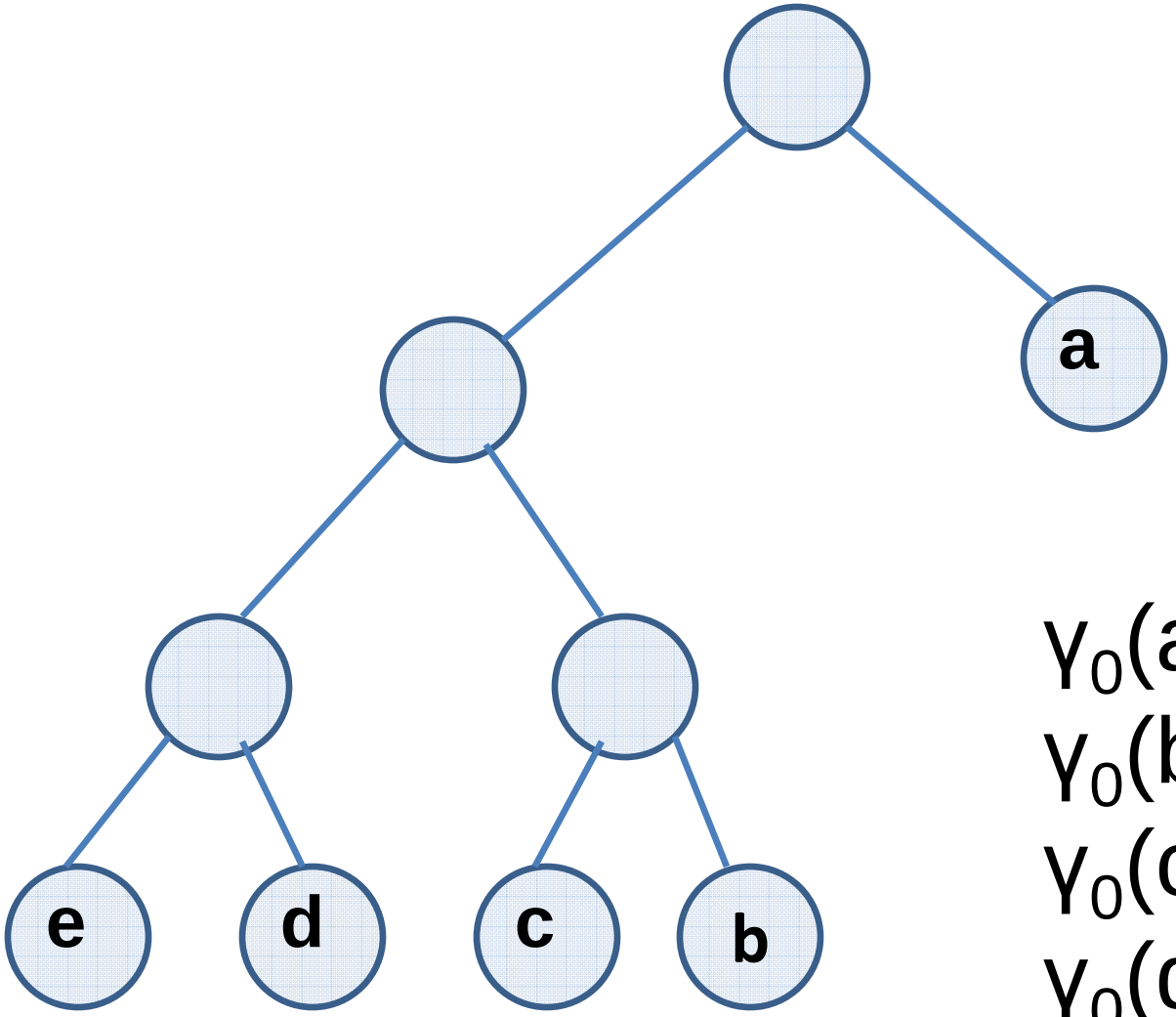
Data Compression and Huffman Codes

Representing prefix codes using binary trees

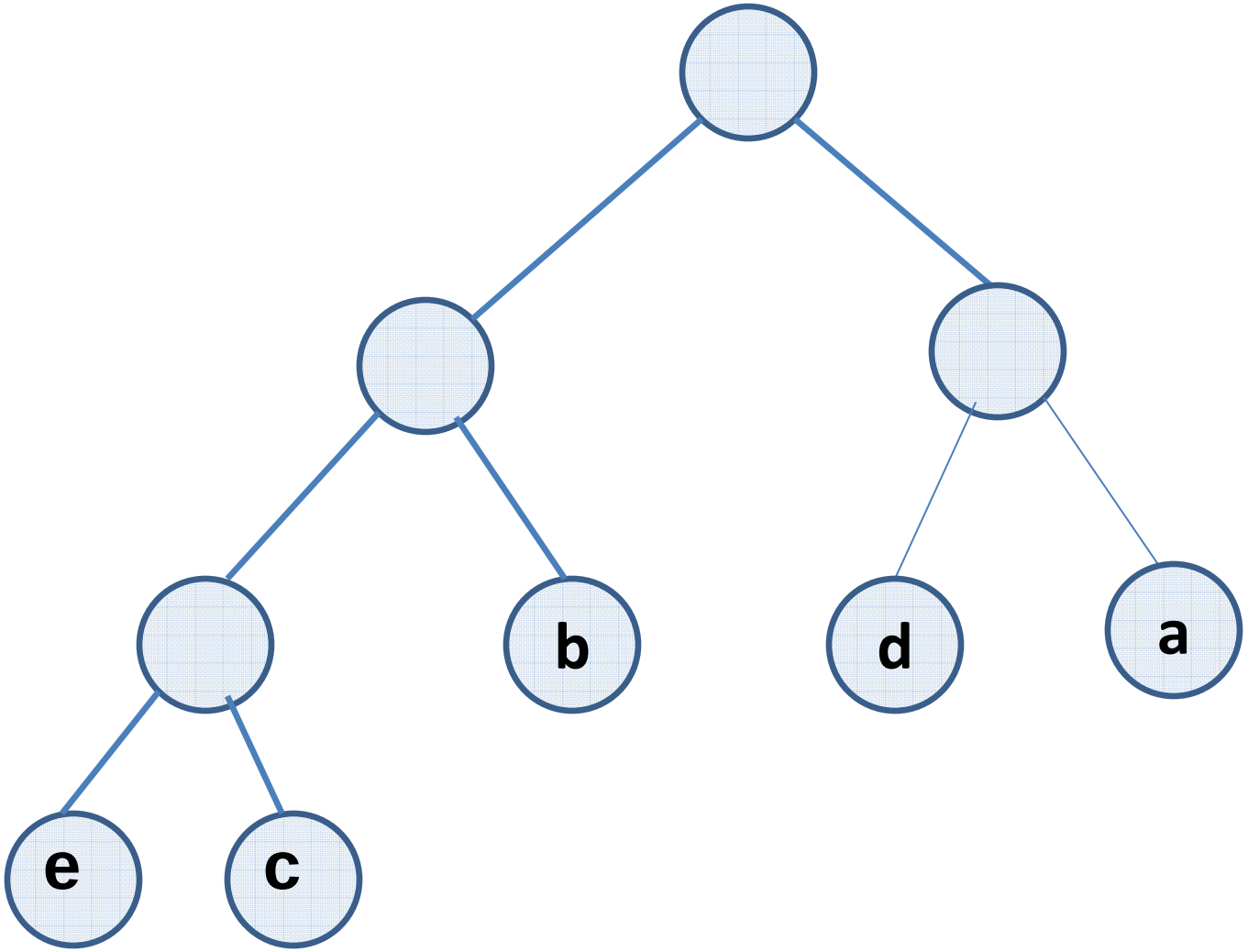
Can also go in the other direction: given a prefix code γ we can build a binary tree T which "stores" the prefix code recursively as follows:

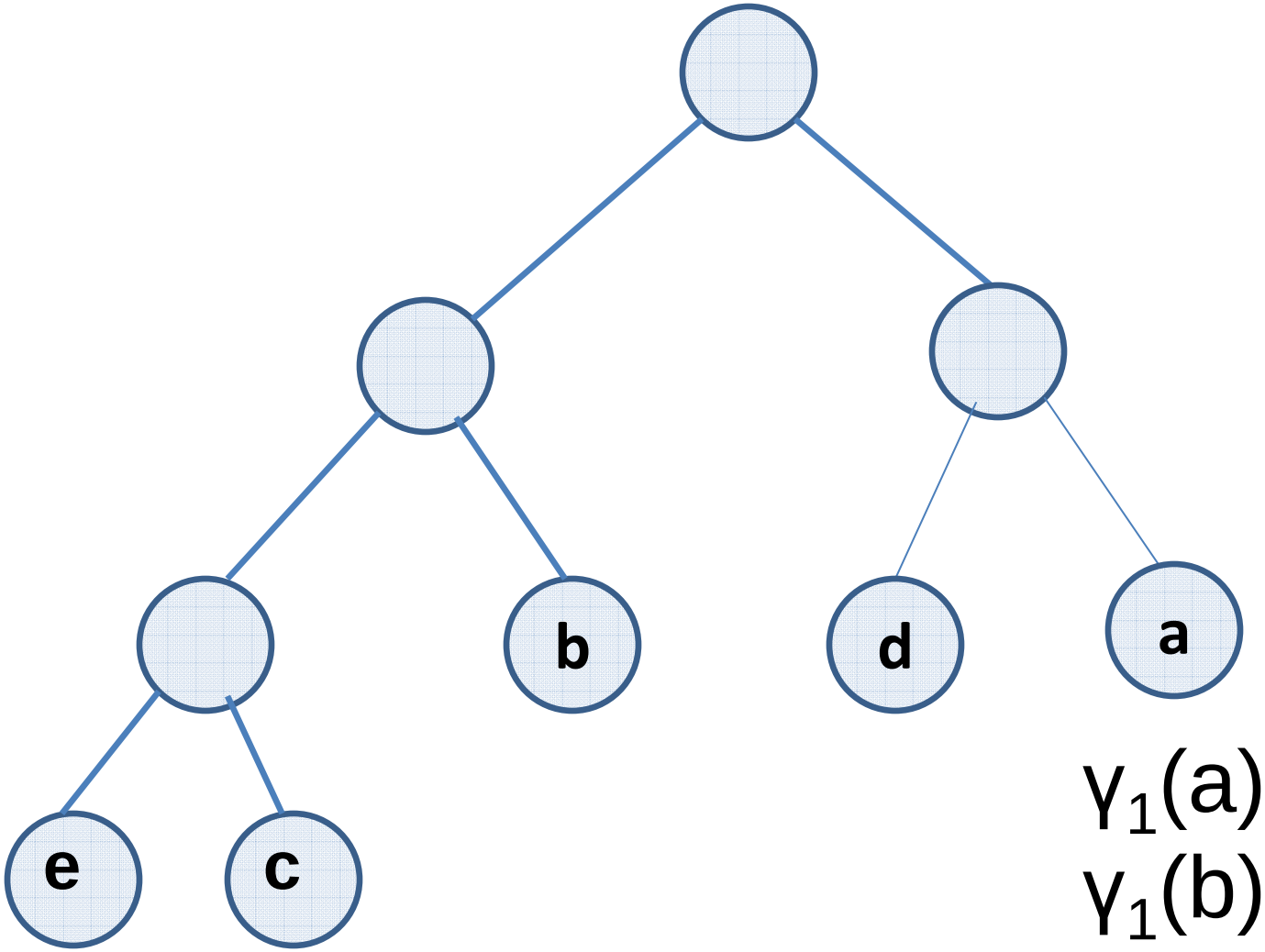
We start with a root; all letters $x \in S$ whose encodings start with a 0 will be leaves in the left subtree of the root; all letters $y \in S$ whose encodings start with a 1 will be leaves in the right subtree of the root;


a
e
d
c
b

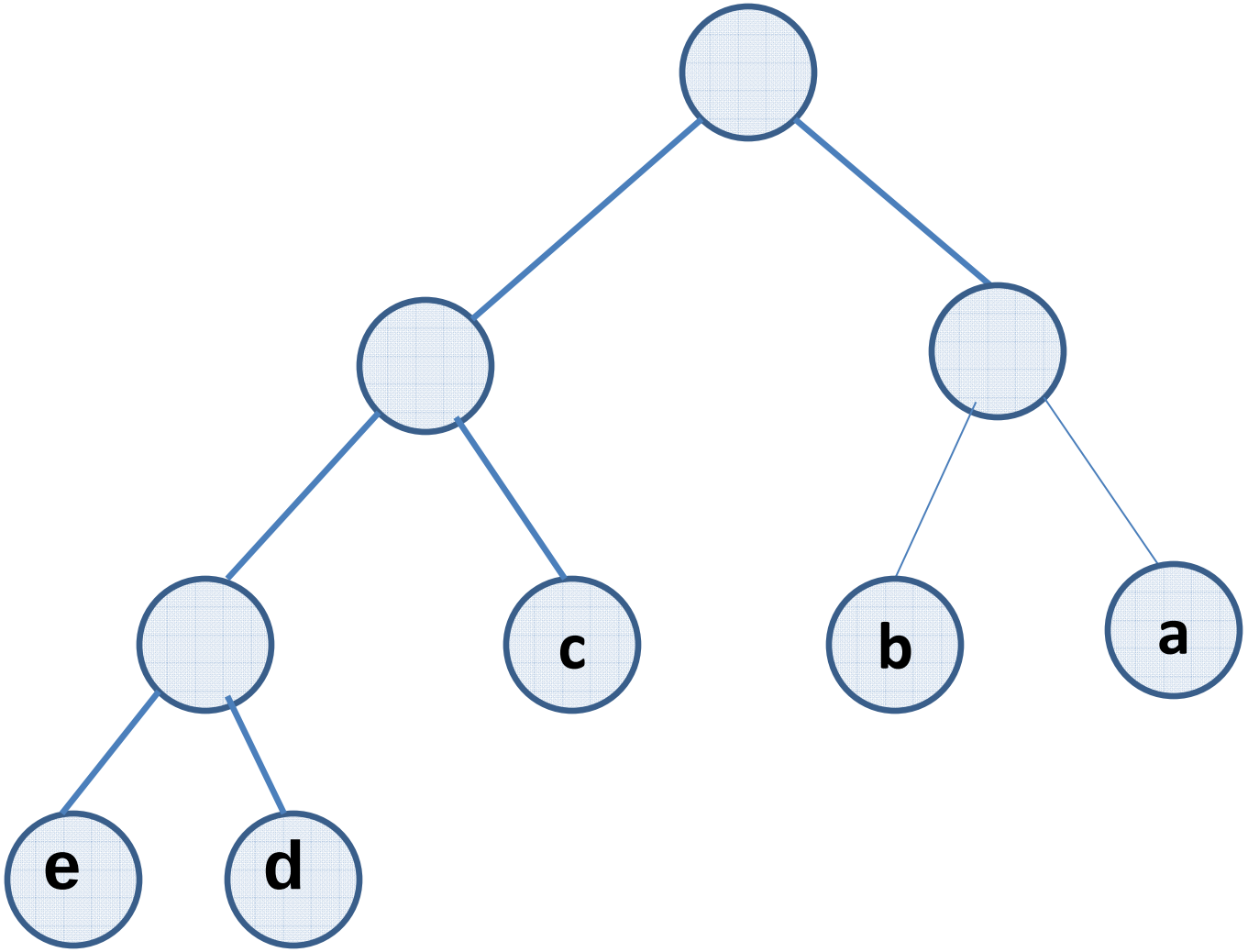


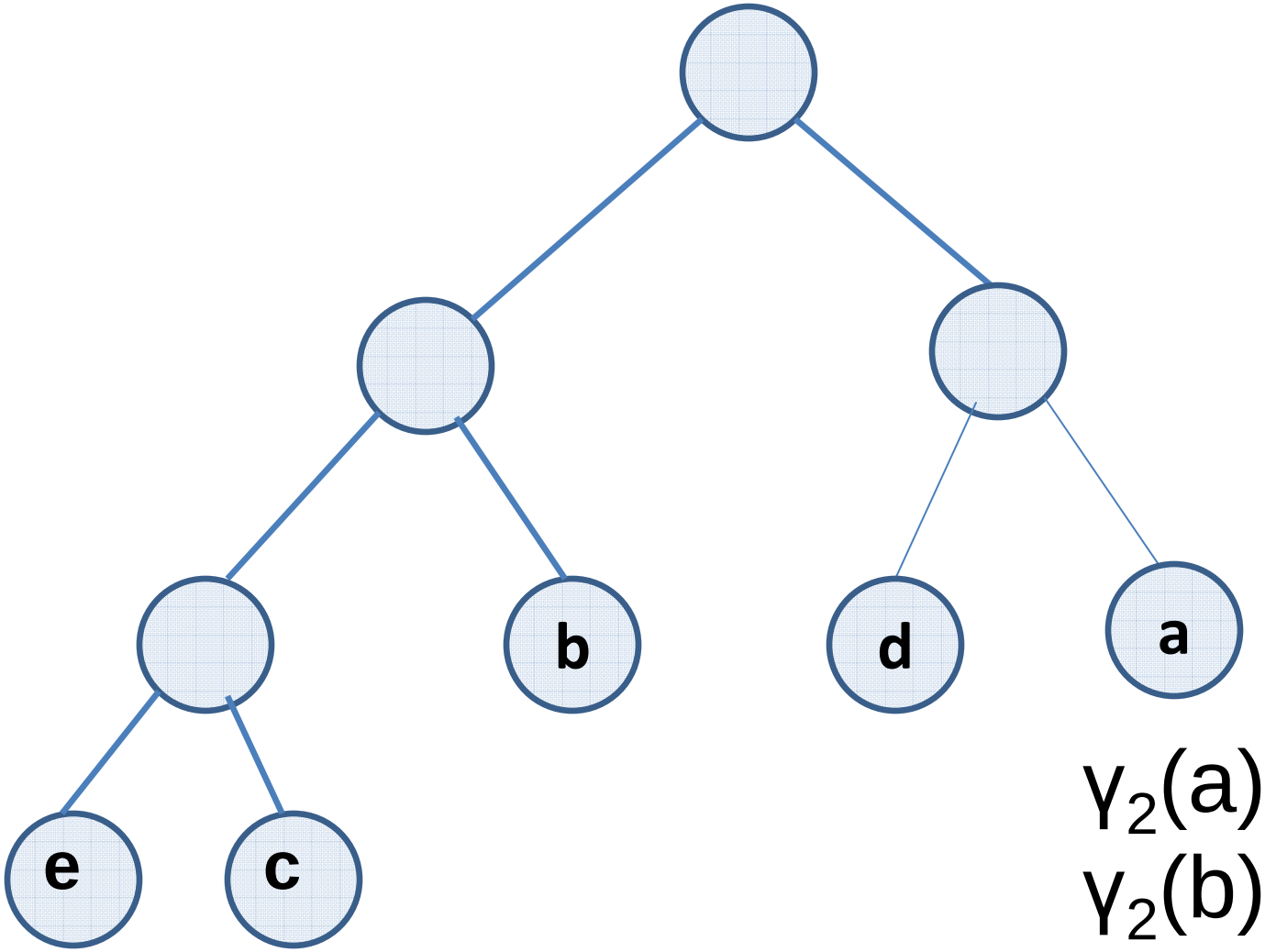
$\gamma_0(a)=1;$
$\gamma_0(b)=011;$
$\gamma_0(c)=010;$
$\gamma_0(d)=001;$
$\gamma_0(e)=000;$


e
c
b
d
a



$\gamma_1(a)=11;$
$\gamma_1(b)=01;$
$\gamma_1(c)=001;$
$\gamma_1(d)=10;$
$\gamma_1(e)=000;$


e
d
c
b
a



$\gamma_2(a)=11;$
$\gamma_2(b)=10;$
$\gamma_2(c)=01;$
$\gamma_2(d)=001;$
$\gamma_2(e)=000;$

Data Compression and Huffman Codes

- The search for an optimal prefix code can be viewed as the search for a binary tree T, together with a labelling of the leaves of T, that minimizes the average number of bits per letter.
- Note: the length of encoding of a letter $x \in S$ is the length of the path from the root to the leaf labeled with x
- Length of the path from root to leaf v is called the depth of the leaf, notation: $\text{depth}_T(v)$; ((recall $\text{depth}_T(v)$ = (level of v) – 1))

Data Compression and Huffman Codes

- Searching for: labeled binary tree that minimizes the weighted average of the depths of the leaves, where the average is weighted by the frequencies of the letters that label the leaves: $\sum_{x \in S} f_x$ **depth$_T$(x),** denote this quantity by **ABL(T)**

- Claim: **The binary tree corresponding to the optimal prefix code is full.** (full = each node has two children)

Data Compression and Huffman Codes

- Discussion of Shannon-Fano codes
- Example: S={a,b,c,d,e} frequencies: f_a=0.32; f_b=0.25; f_c=0.20; f_d=0.18; f_e=0.05;
- Resulting code is γ_1 (which we know not to be optimal)

Data Compression and Huffman Codes

- Claim: Suppose T* is a binary tree corresponding to an optimal prefix code. Let u and v be leaves of T* such that depth(u) < depth(v). Suppose further that leaf u is labeled with $y \in S$ and leaf v is labeled with $x \in S$. Then $f_y \geq f_z$.
- If somebody gave you the structure of T* without the labeling, you would be able to label it in an optimal way

Data Compression and Huffman Codes

- Claim: Consider T* is a binary tree corresponding to an optimal prefix code. Let v be a leaf in T* whose depth is as large as possible. Leaf v has a parent u (we exclude the trivial case of alphabets with one letter), we know that T* is full binary tree , u has another child w. **This child, w, is a leaf of T*.** (we will refer to w and v as siblings)
- Claim: There is an optimal prefix code, with corresponding tree T*, in which the lowest-frequency letters are assigned to leaves that are siblings in T*

Data Compression and Huffman Codes

An Algorithm to Construct an Optimal Prefix Code:

- Suppose y* and z* are letters in S with the two lowest frequencies (can break ties arbitrarily). Previous claim tells where y* and z* go (can go) in an optimal solution: they end up as sibling leaves below a common parent.

This common parent acts like a “meta-letter” whose frequencey is the sum of the frequencies of y* and z*

Data Compression and Huffman Codes

An Algorithm to Construct an Optimal Prefix Code:

- Algo: replace y* and z* with with this meta-letter, obtaining an alphabet which is one letter smaller. We recursively find prefix code for the smaller alphabet, and then “unwrap” the metaletter back into y* and z* to obtain prefix code for S:

Data Compression and Huffman Codes

To construct a prefix code for an alphabet S with given frequencies:

if S has two letters then

encode one letter using 0 and the other using 1

else

let y* and z* be the two lowest-frequency letters

form a new alphabet S' by deleting y* and z* and replacing them with a new letter ω of frequency $f_{y^*}+f_{z^*}$.

recursively construct a prefix code γ' for S', with tree T'

Define a prefix code for S as follows:

Start with T'

Take the leaf ω and add two children below it labeled y* and z*

endif

Data Compression and Huffman Codes

- Claim: $ABL(T') = ABL(T) - f_{\omega}$.
- Claim: the Huffman code for a given alphabet achieves the minimum number of bits per letter of any prefix code.
- Implementation: without thought it is $O(k^2)$ where k is the number of letters in the alphabet; using priority queues we get $O(k \log(k))$.
- extensions