# Huffman Coding

Greedy Algorithms

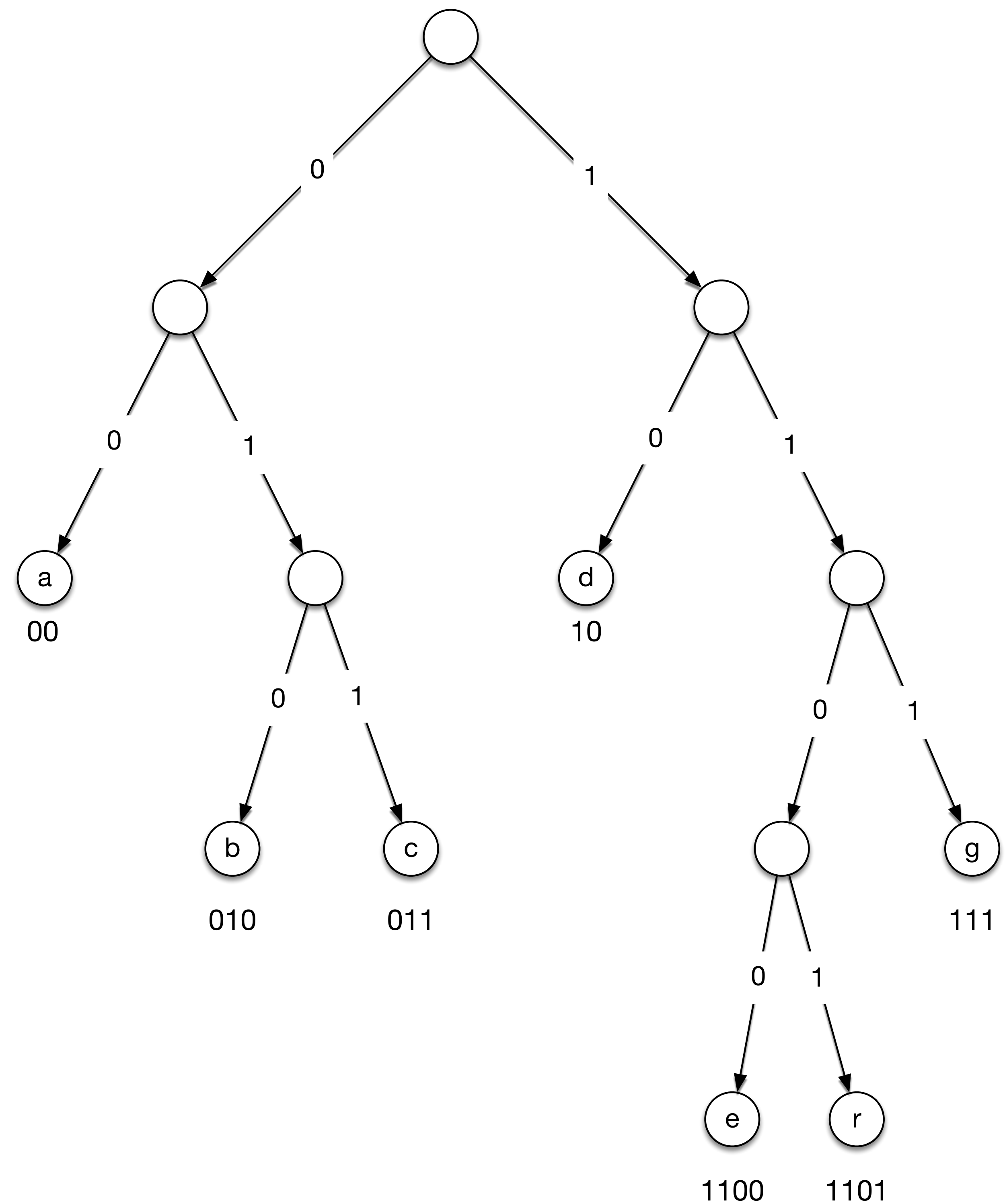
# Prefix-free Codes

- Binary, variable length code
    - Binary: code symbols are 0 and 1
    - Variable length: code words have variable length
- To allow decoding:
    - no prefix of a code word can be part of another code word
    - Otherwise:
        - Cannot decide easily between prefix and complete

00101    00101110101

# Prefix-free Codes

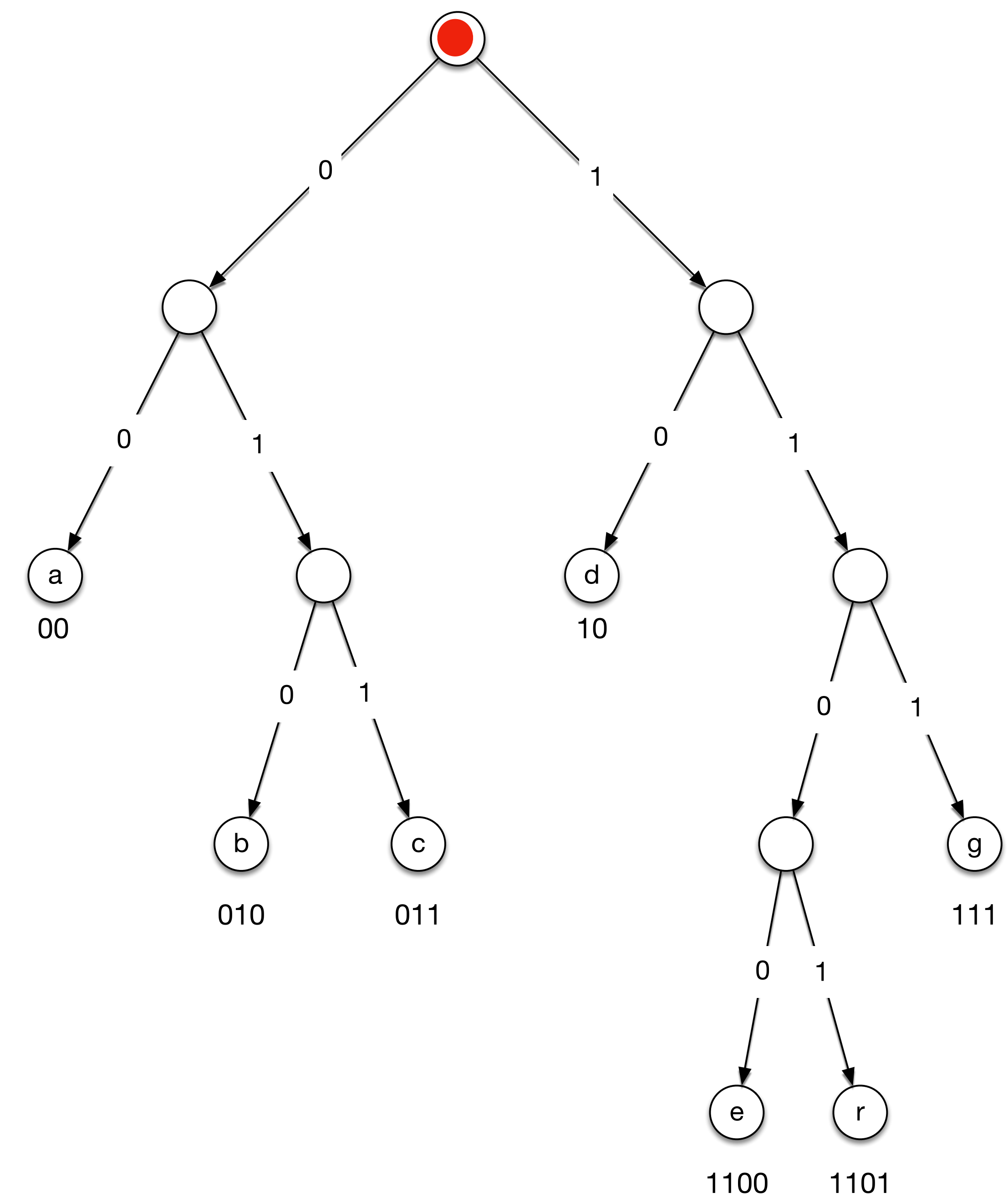
- Prefix codes can be represented as binary trees
  - Left branch is labelled with 0, right with 1
  - Leaves correspond to symbols
  - Path to leaf is code for symbol



# Prefix-free Codes

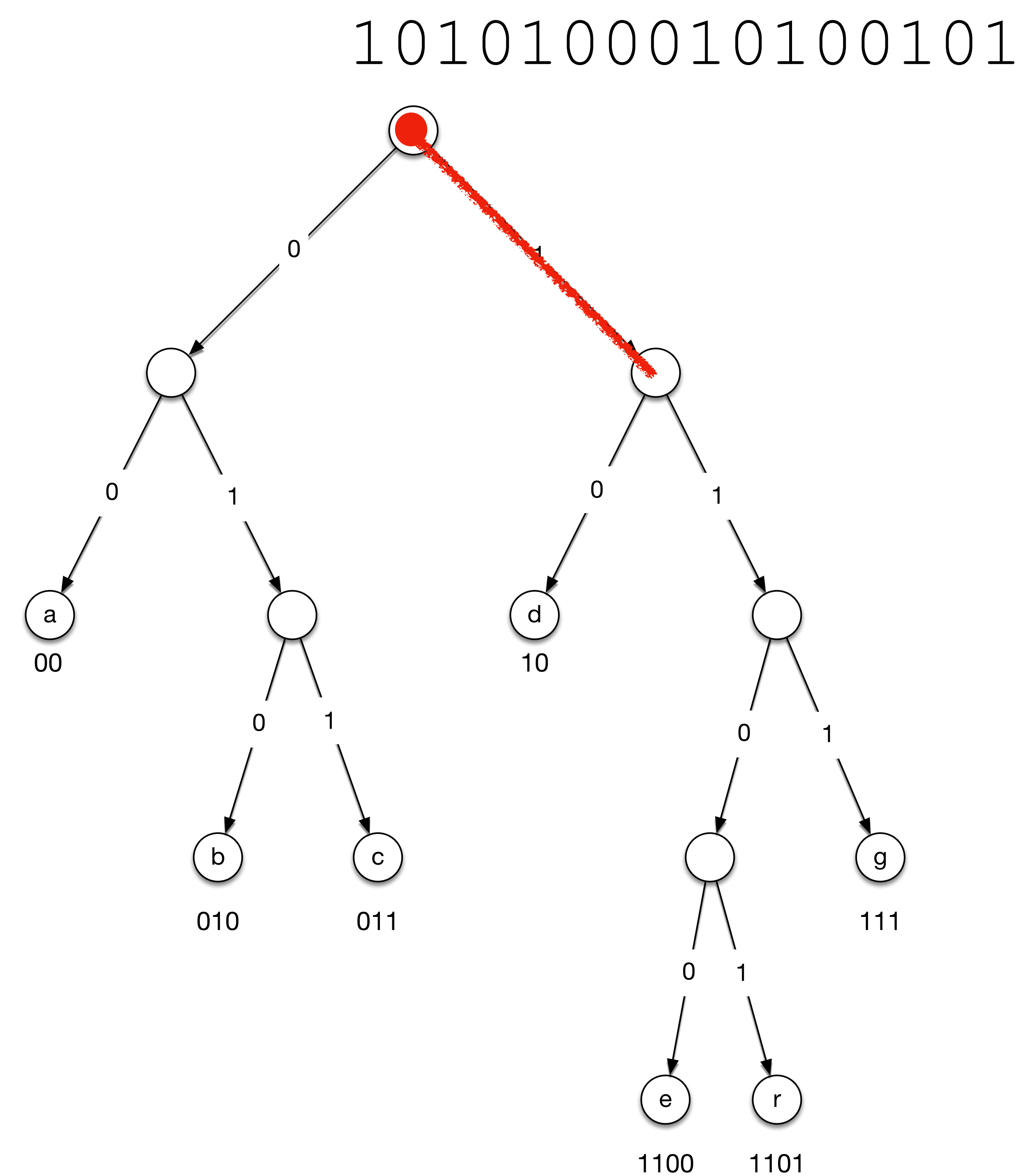
- Start at top

1010100010100101



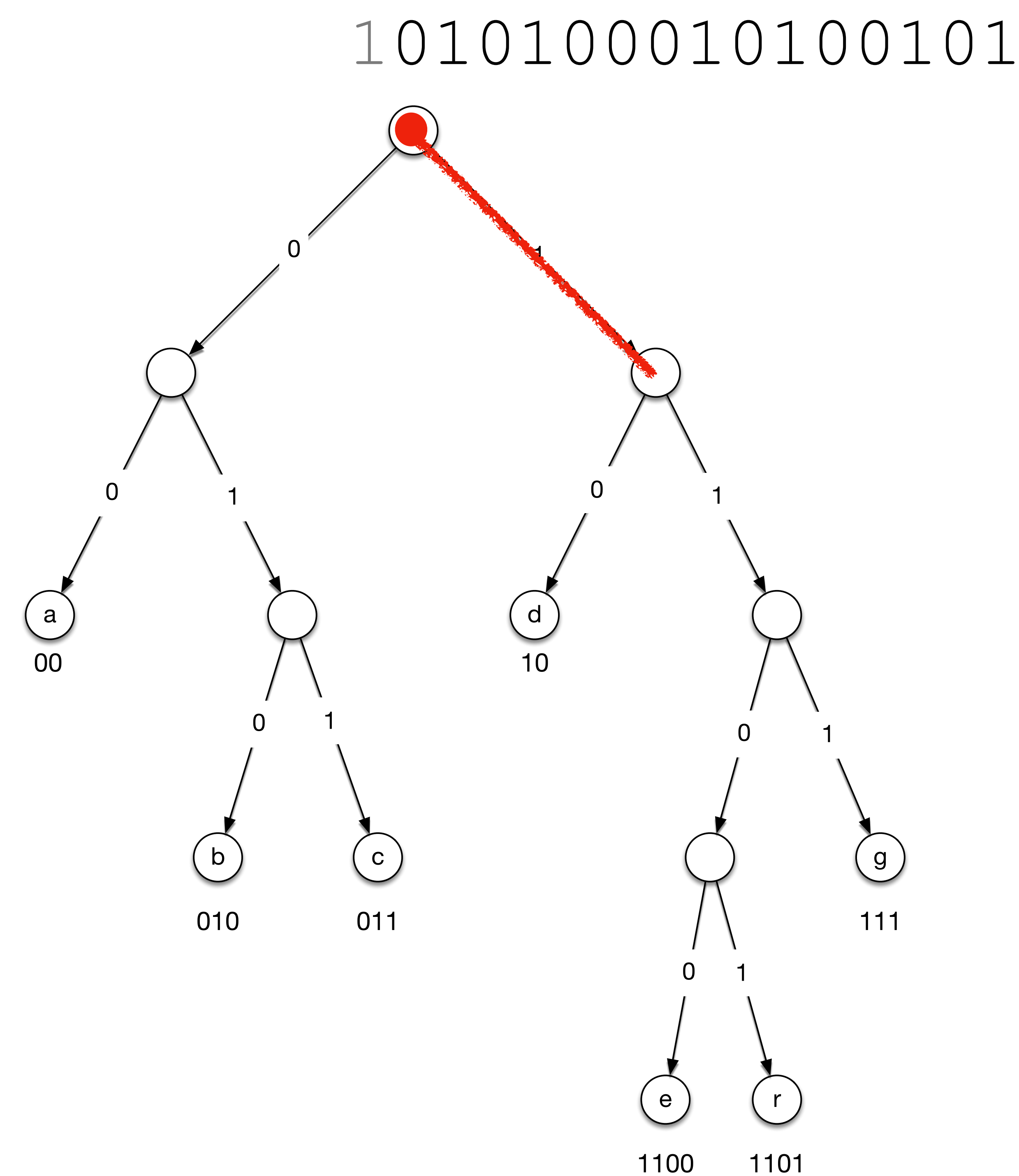
# Prefix-free Codes

- Start at top
- First letter is 1:
  - Go to the right



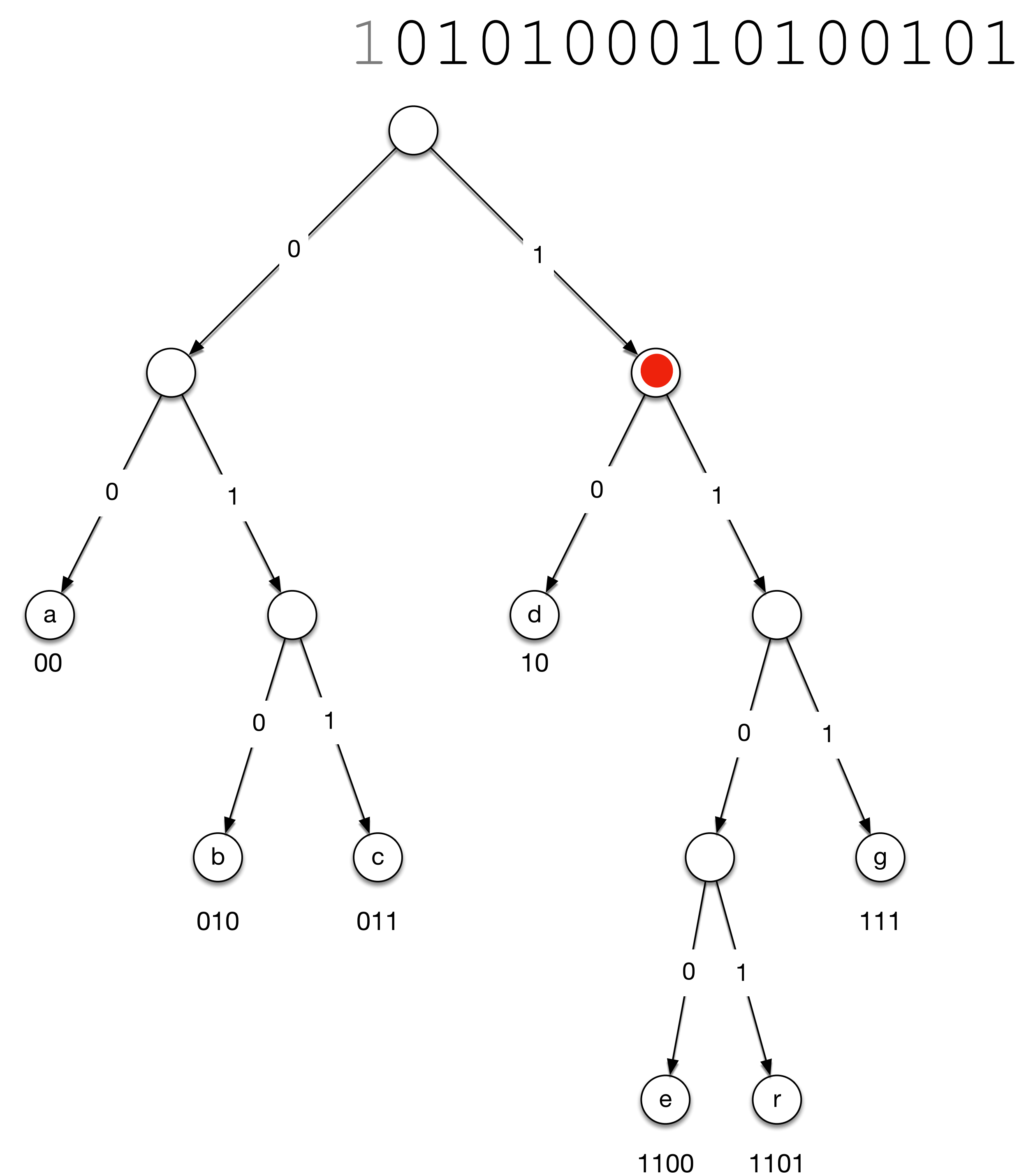
# Prefix-free Codes

- Start at top
- First letter is 1:
    - Go to the right
    - Have processed 1



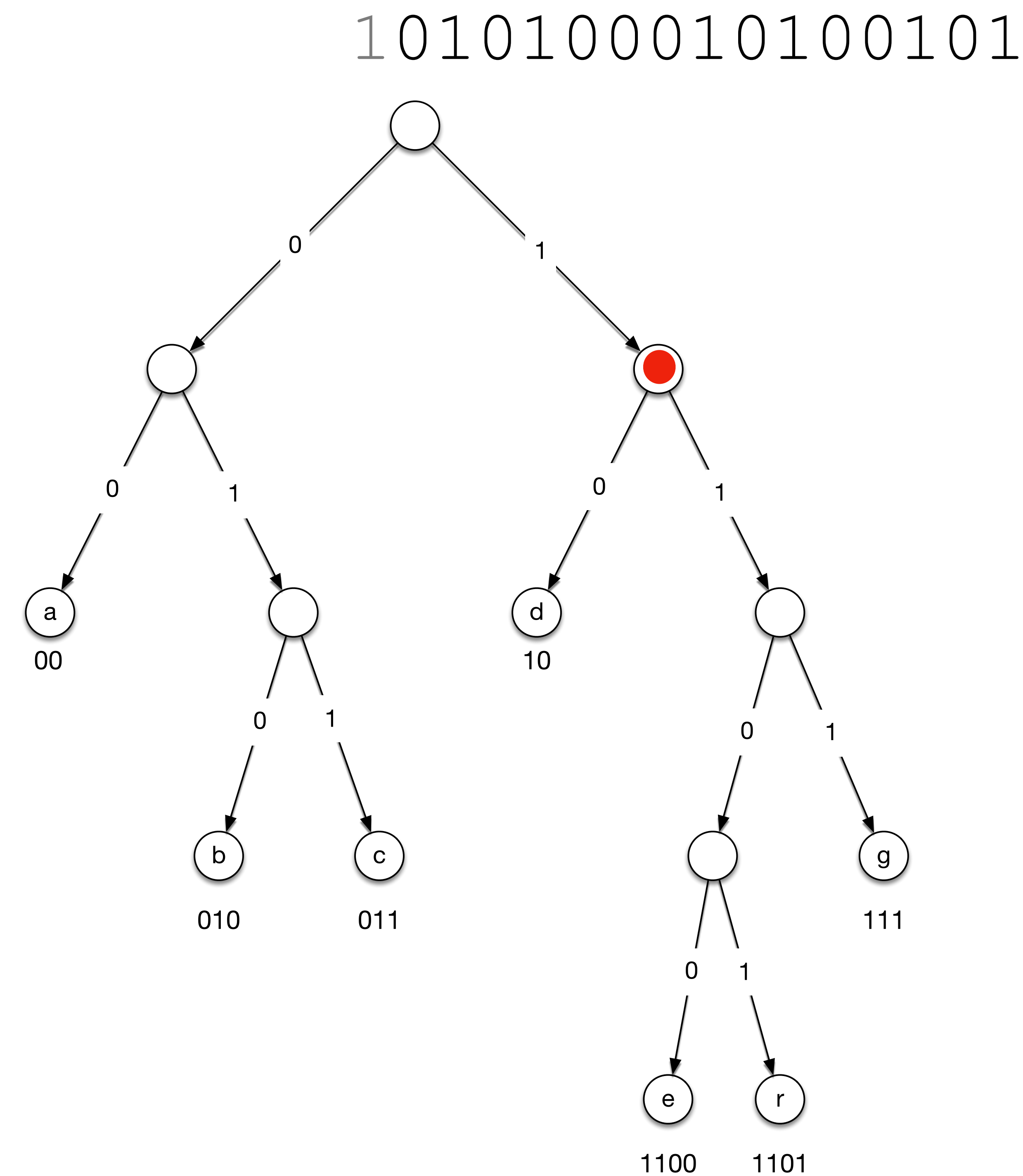
# Prefix-free Codes

- Start at top
- First letter is 1:
    - Go to the right
    - Have processed 1



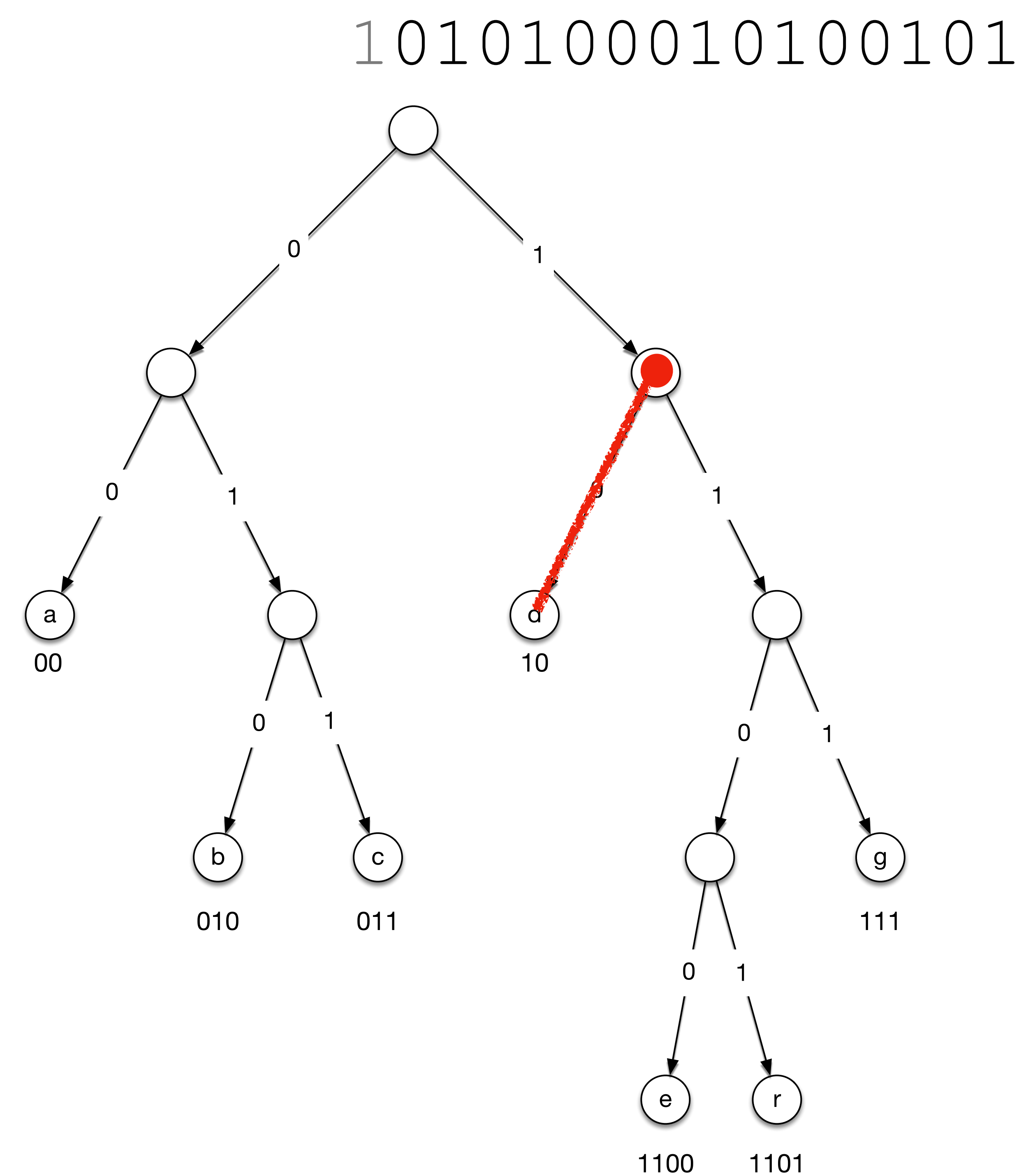
# Prefix-free Codes

- Start at top
- First letter is 1:
  - Go to the right
  - Have processed 1
- Second letter is 0:



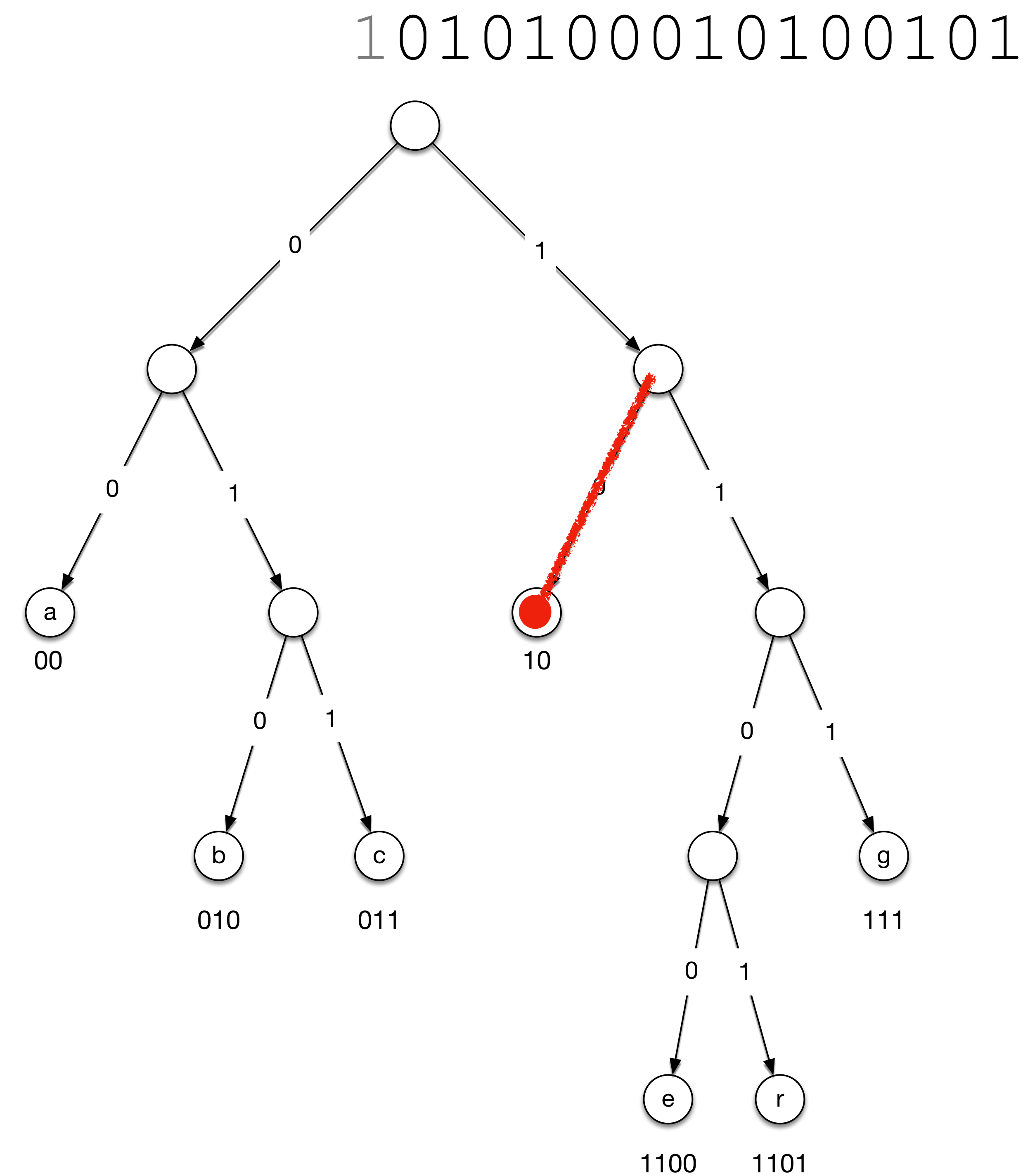
# Prefix-free Codes

- Start at top
- First letter is 1:
  - Go to the right
  - Have processed 1
- Second letter is 0:
  - Go to the left



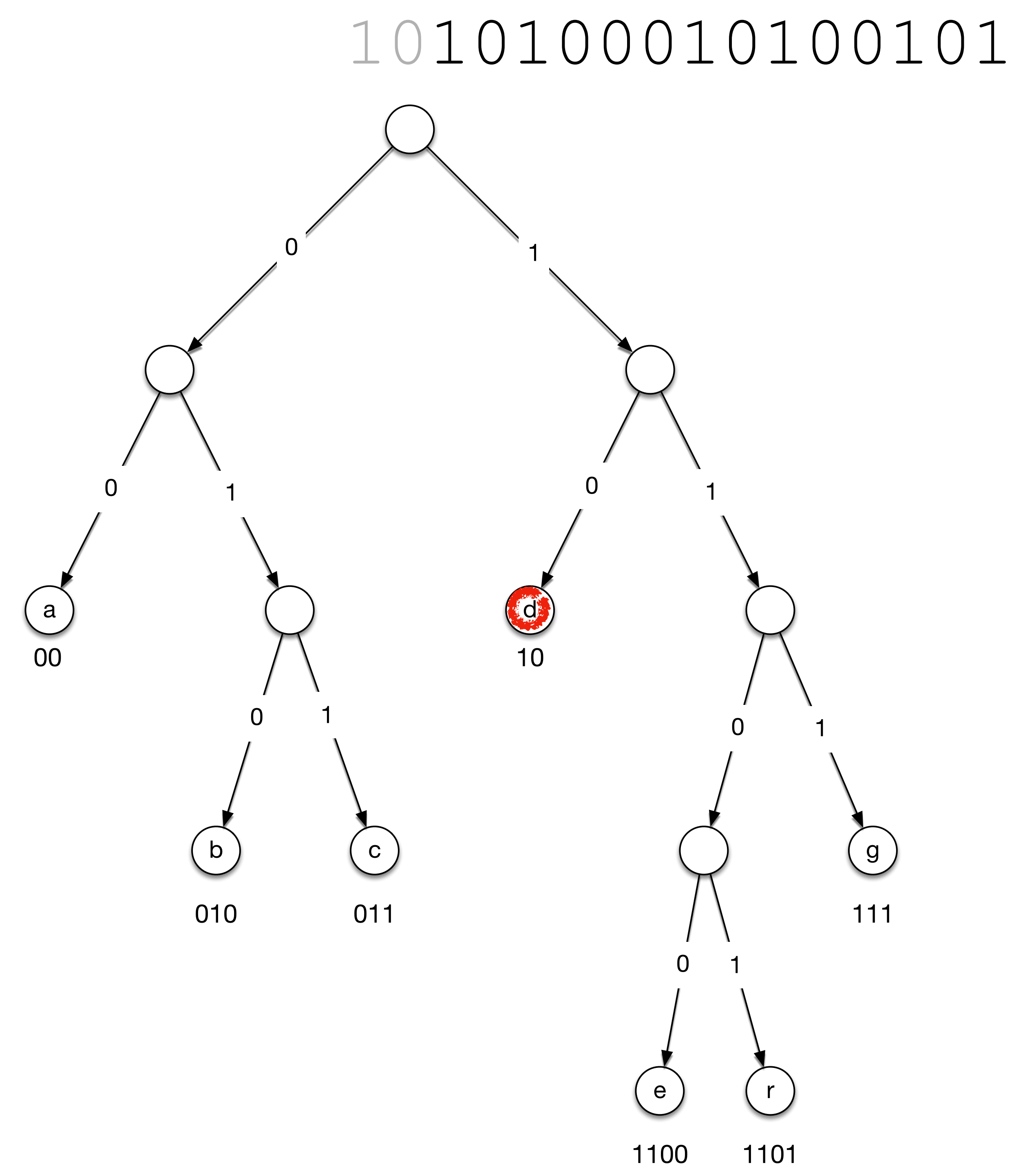
# Prefix-free Codes

- Start at top
- First letter is 1:
  - Go to the right
  - Have processed 1
- Second letter is 0:
  - Go to the left
  - We are in a leaf



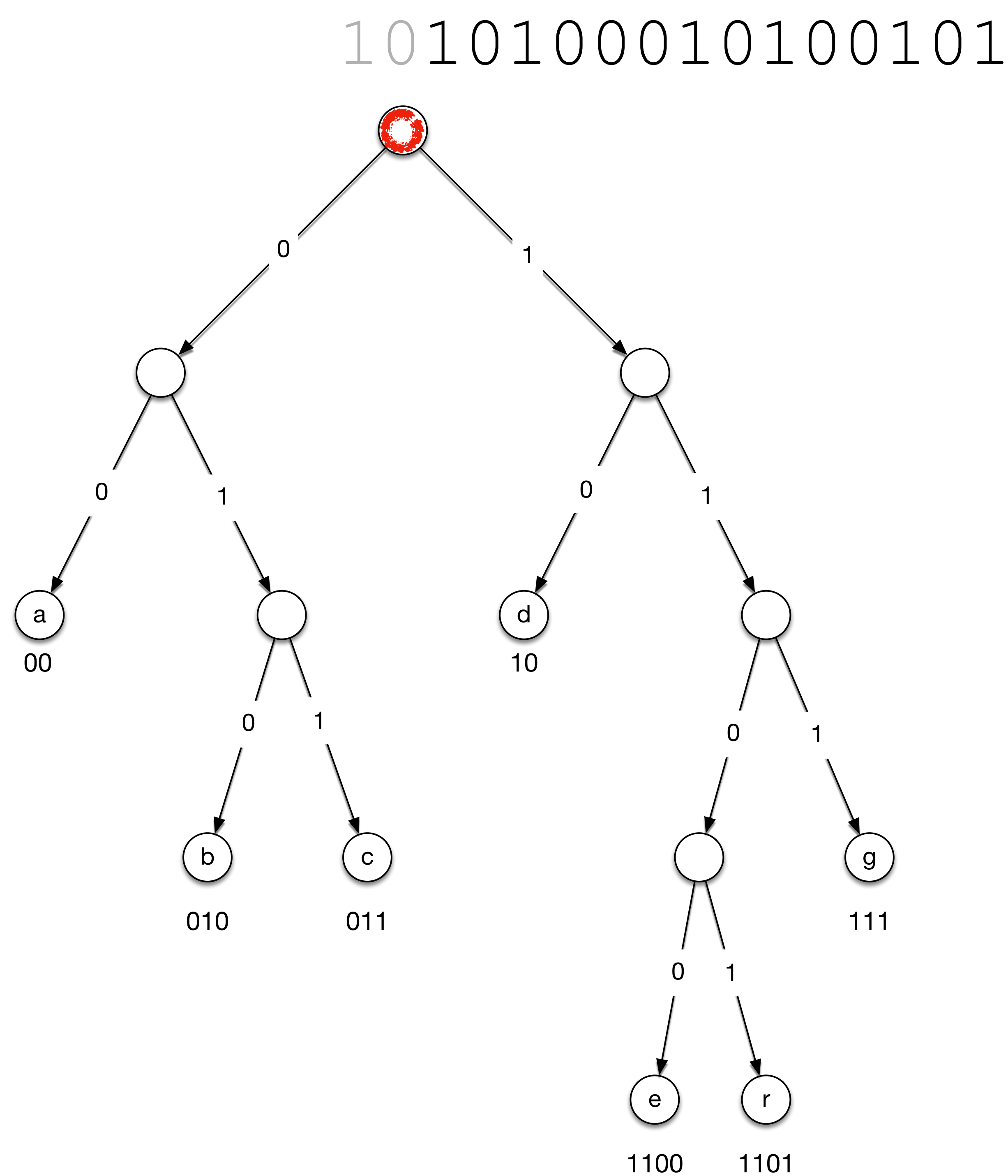
# Prefix-free Codes

- Start at top
- First letter is 1:
    - Go to the right
    - Have processed 1
- Second letter is 0:
    - Go to the left
    - We are in a leaf
- Emit the value of the leaf: d



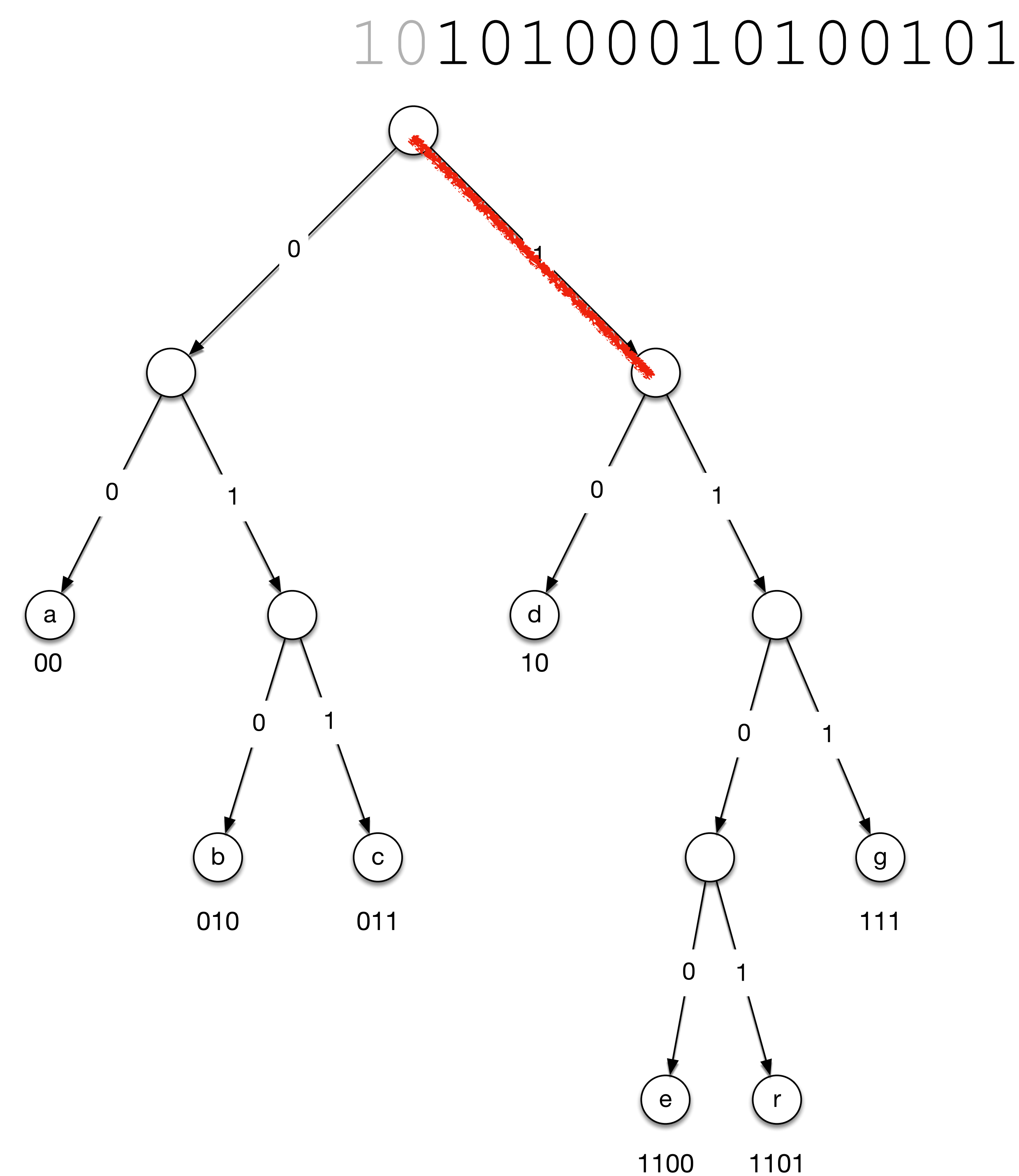
# Prefix-free Codes

- Restart at the top
  - Next letter is 1



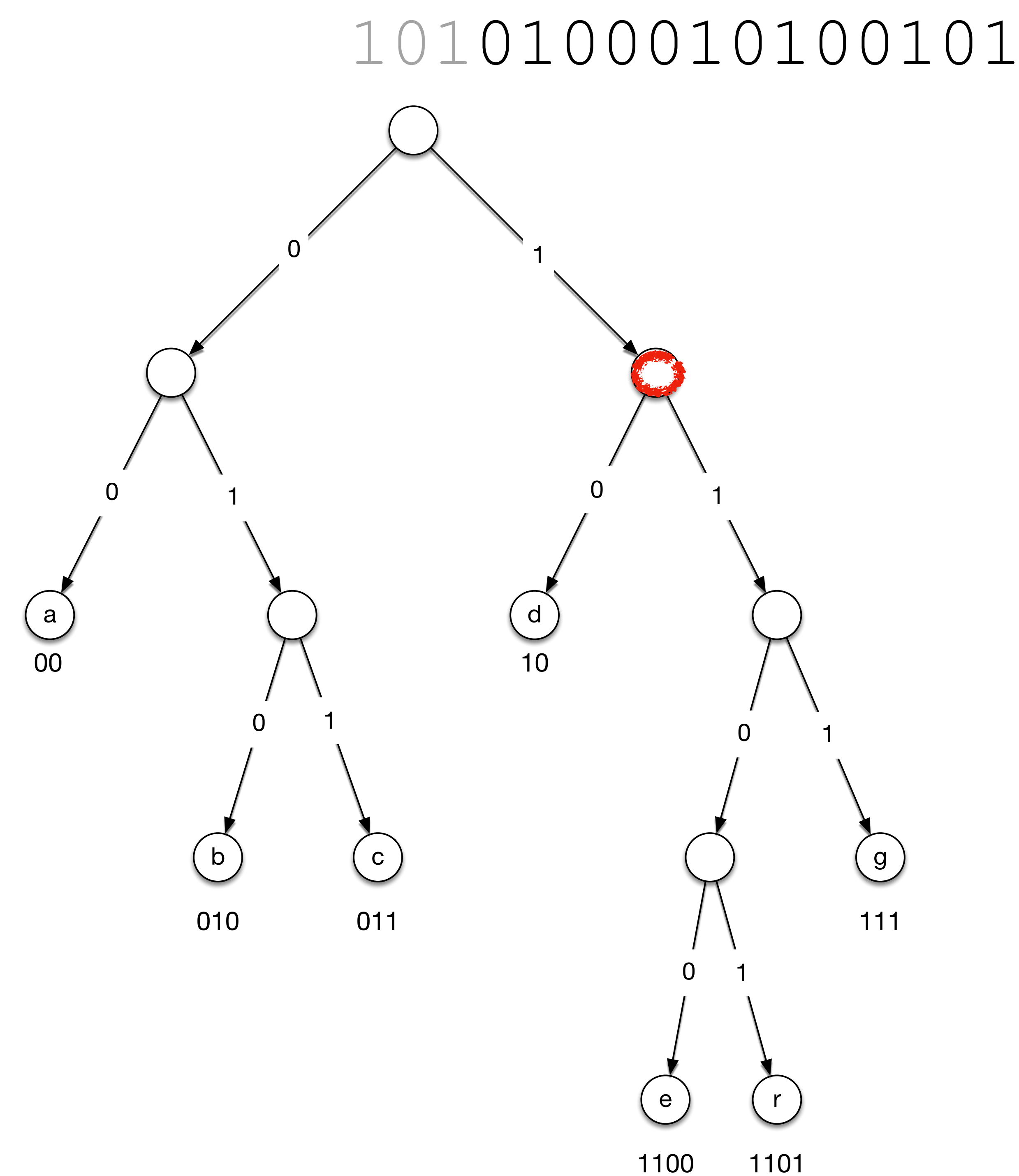
# Prefix-free Codes

- Restart at the top
  - Next letter is 1
  - Go to the right

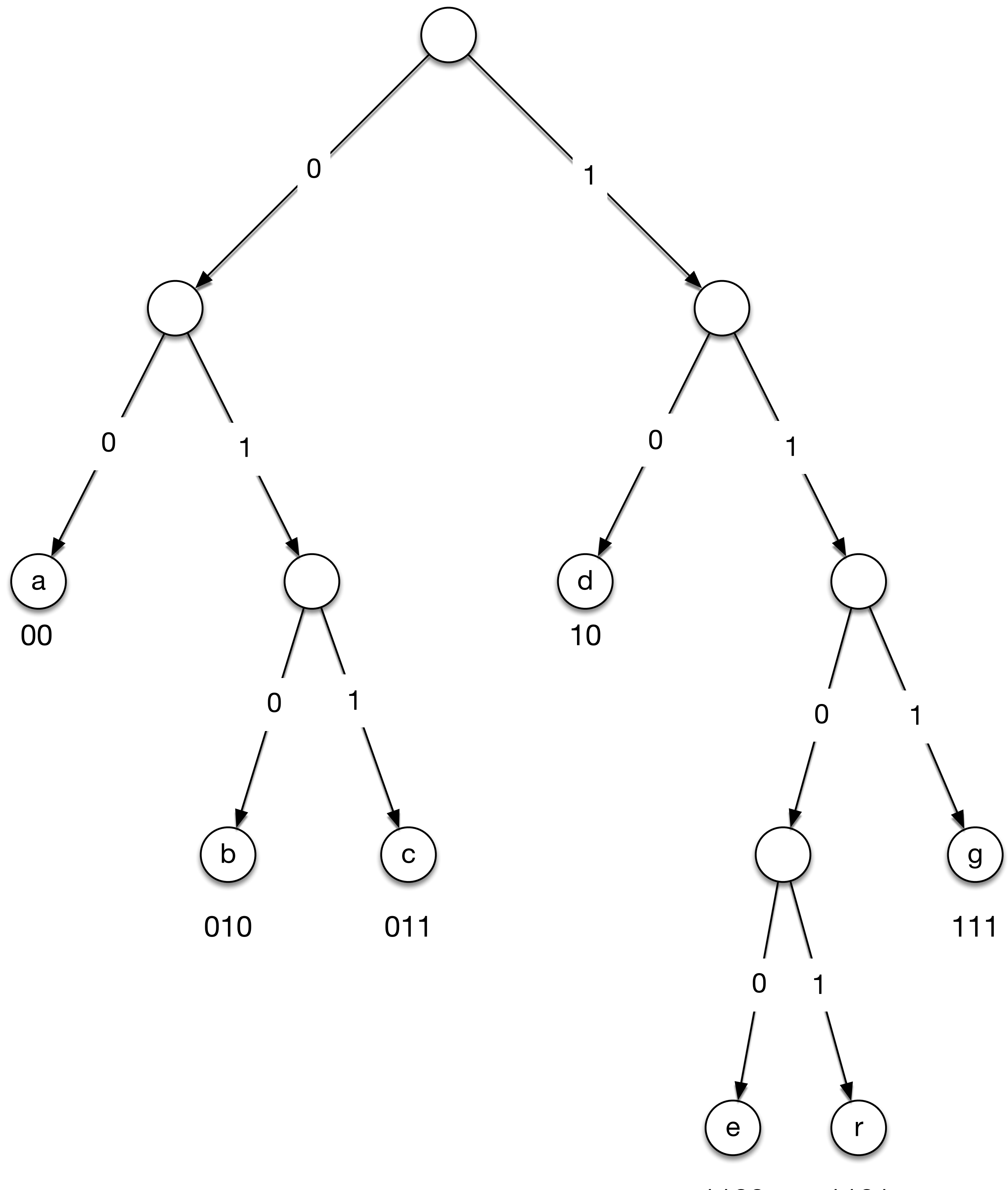


# Prefix-free Codes

- Restart at the top
  - Next letter is 1
  - Go to the right
  - Process repeats

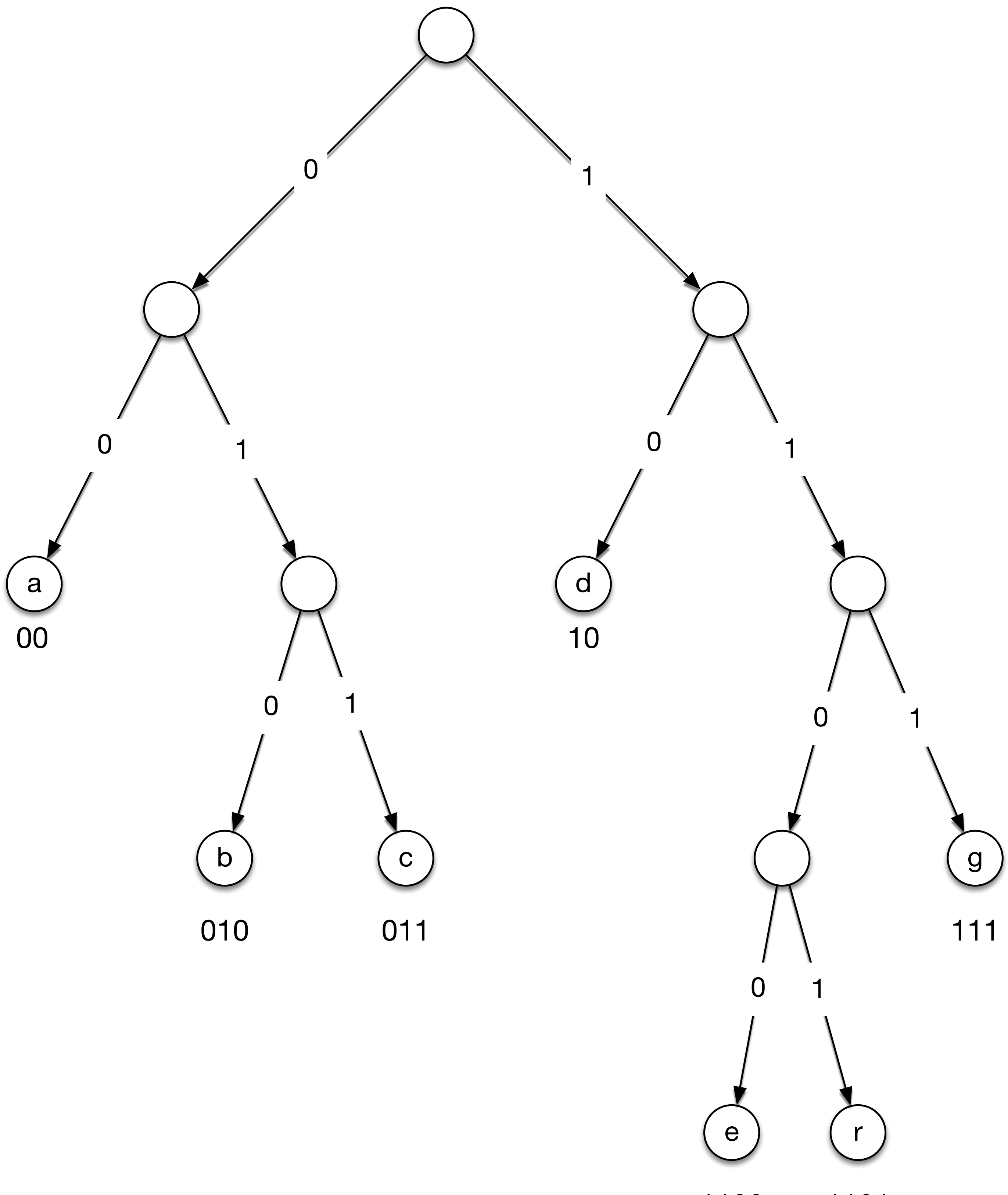


# Prefix-free codes



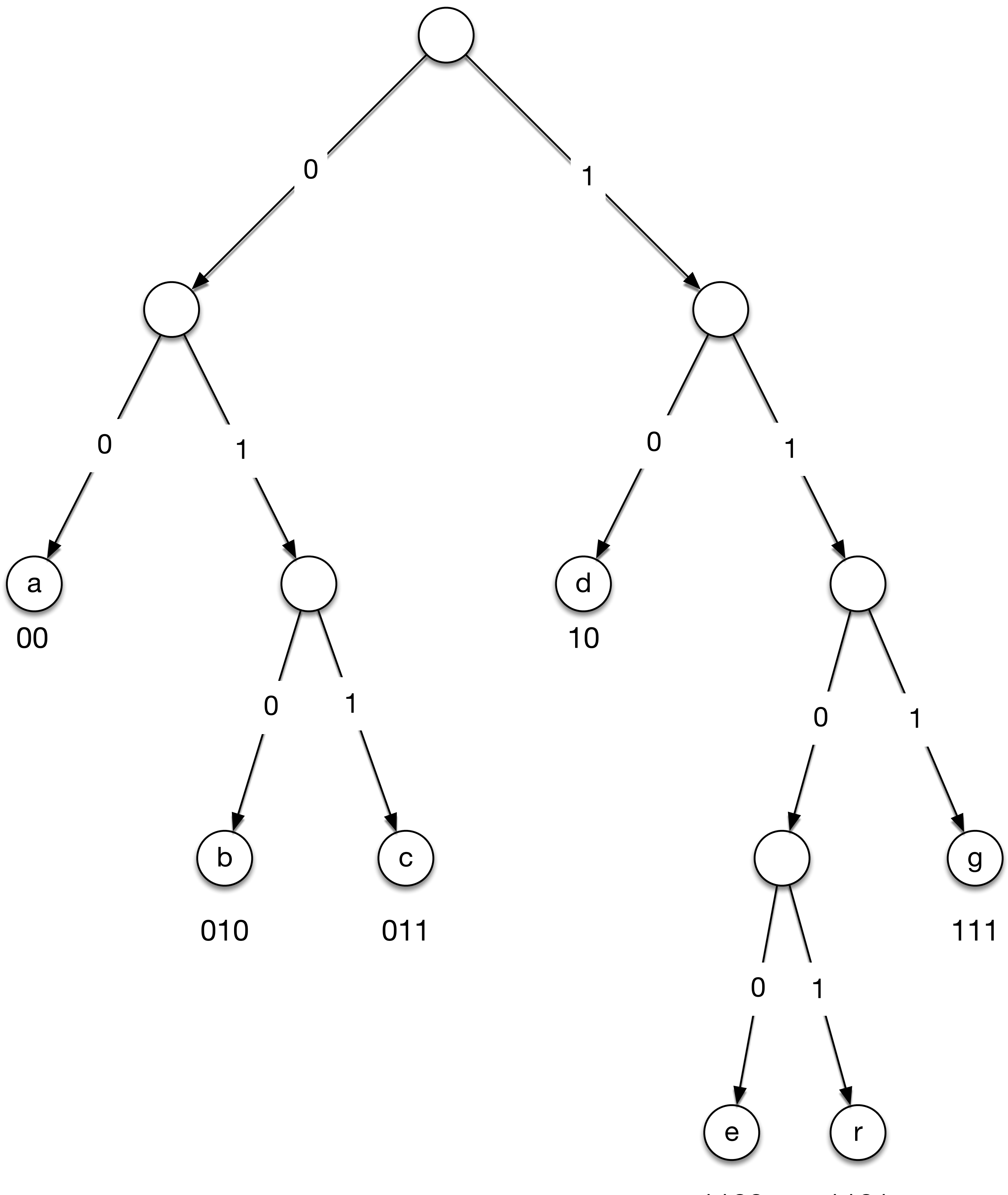

- Decoding 10101000010100101
- Start at top
- Follow bits
  - 10 — d
  - 10 — d
  - 10 — d
  - 00 — a
  - 010 — b
  - 10 — d
  - …

# Prefix-free Codes




- Your turn:
    - 1100111110101101000

# Answer




- Your turn:
    - 11001111010110100
      0
    - 1100 - e
    - 111 - g
    - 1101 - f
    - 011 - c
    - 010 - b
    - 00 - a

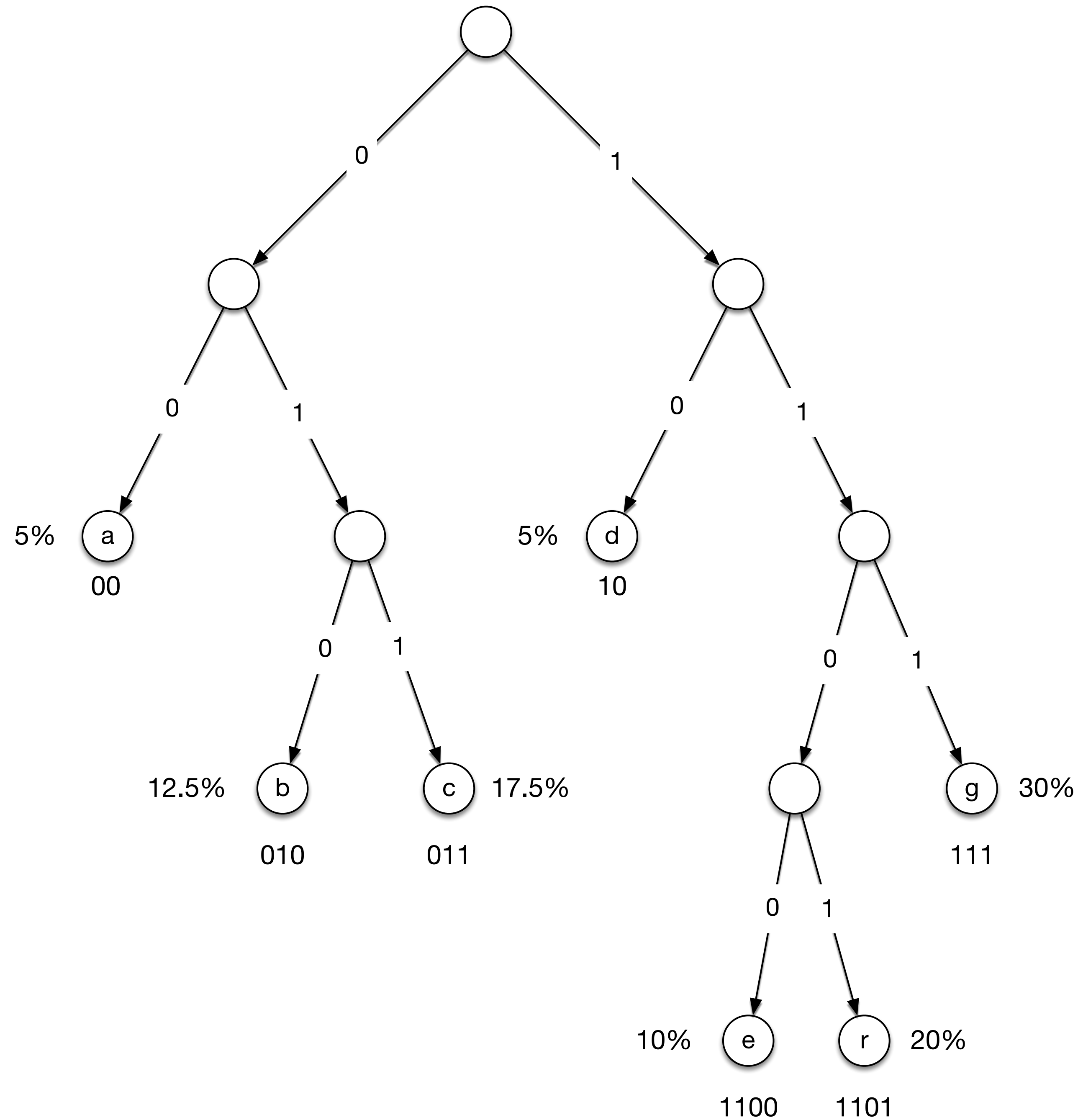
# Huffman Coding

- Obviously, there are many binary trees with a certain number of leaves
- If the symbols appear with different frequencies, then we want to encode frequent ones with short codes and infrequent ones with longer codes
  - **Huffman Coding:**
    - Greedy algorithm to calculate an optimal encoding

# Huffman Coding

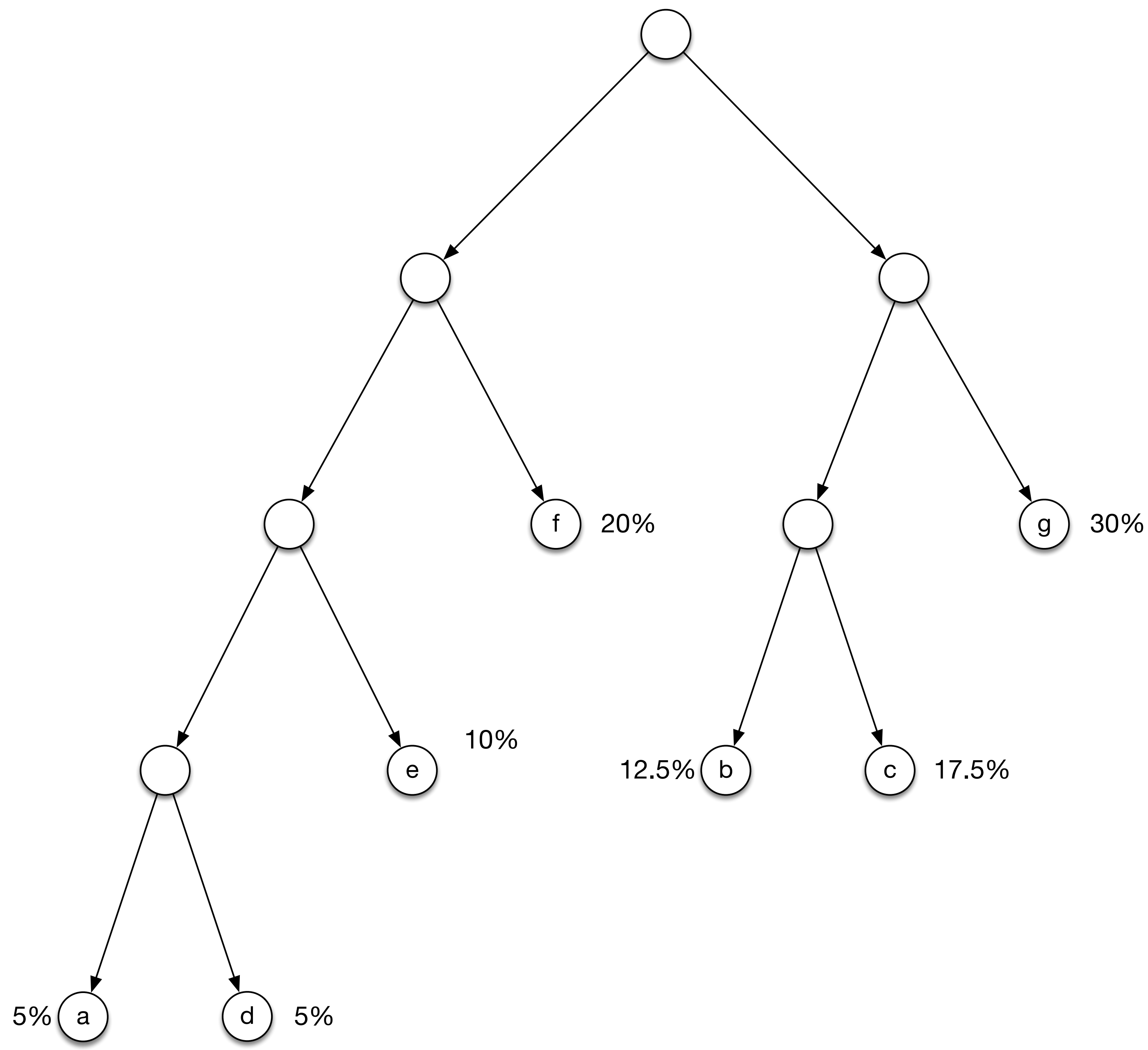
- Measure of goodness
  - Frequency of symbols $f(x)$
  - Depth of corresponding leaf = length of encoding $d(x)$
  - Average Encoding Costs $B = \sum_x f(x) \cdot d(x)$

# Huffman Coding




$$B = 2 \times 0.05 + 3 \times 0.125 + 3 \times 0.175 + 2 \times 0.05 + 4 \times 0.1 + 4 \times 0.2 + 3 \times 0.3 = 3.12$$

# Huffman Coding




$$B = 4 \times 0.05 + 4 \times 0.05 + 3 \times 0.1 + 2 \times 0.2 + 3 \times 0.125 + 3 \times 0.175 + 2 \times 0.3 = 2.6$$

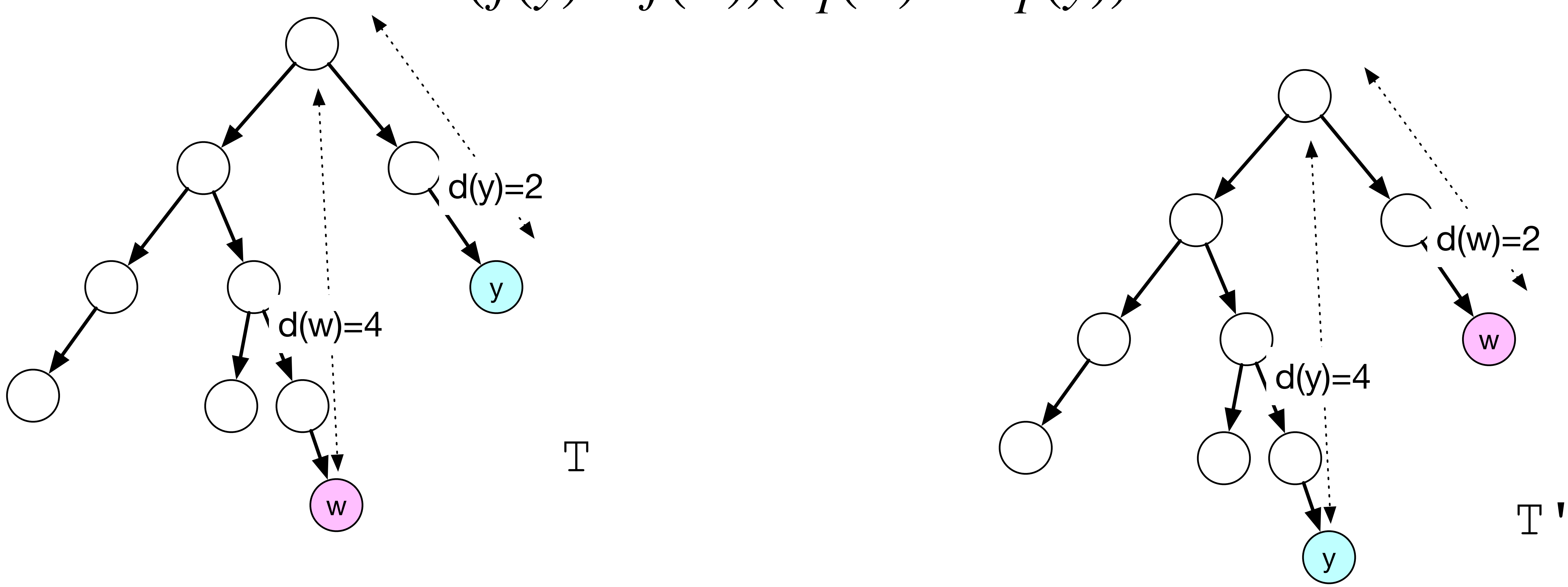
# Huffman Coding

- As we can see, different trees have different expected encoding length

# Huffman Coding

- Let *T* be a binary (encoding) tree and let *T'* be the tree obtained by swapping two leaves y and w.
- Then the difference in the B-values is

$$(f(y) - f(w))(d_T(w) - d_{T'}(y))$$



# Huffman Coding

- Proof:
  - The only difference are the addends corresponding to y and w

$$
\begin{aligned}
& B(T') - B(T) \\
= \; & f(y)d_{T'}(y) + f(w)d_{T'}(w) - f(y)d_T(y) - f(w)d_T(w) \\
= \; & f(y)d_T(w) + f(w)d_T(y) - f(y)d_T(y) - f(w)d_T(w) \\
= \; & \Big(f(y) - f(w)\Big)\Big(d_T(w) - d_T(y)\Big)
\end{aligned}
$$

# Huffman Coding

- What does
  $$B(T') - B(T) = \Big(f(y) - f(w)\Big)\Big(d_T(w) - d_T(y)\Big)$$
  mean?

- If $f(y) > f(w)$ then $y$ better be up higher in the tree or we can gain by swapping

# Huffman Coding

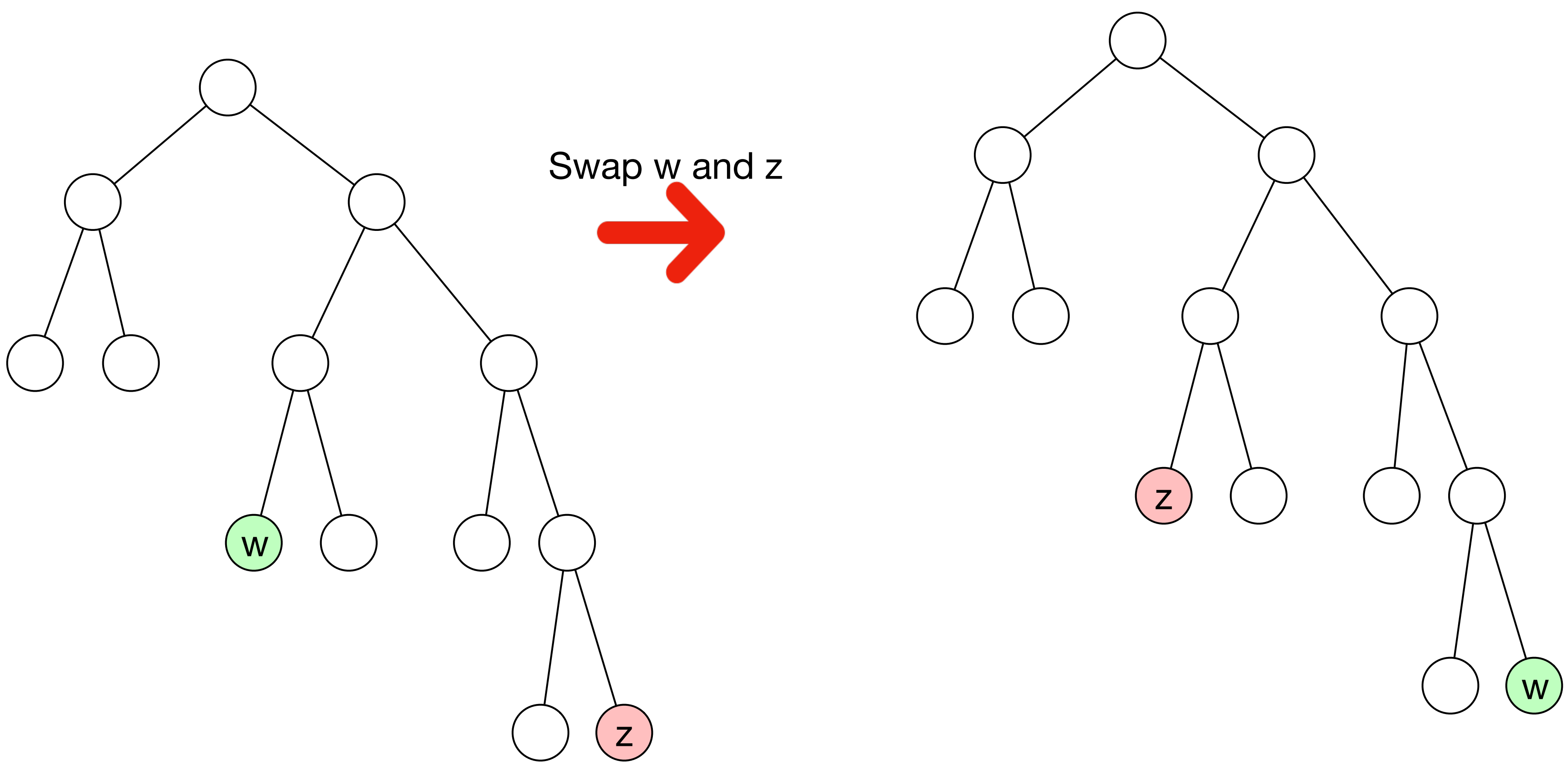
- Lemma: There exists an optimal tree such that the two lowest-frequency symbols are neighbors
  - Furthermore, they have the highest distance from the root

# Huffman Coding

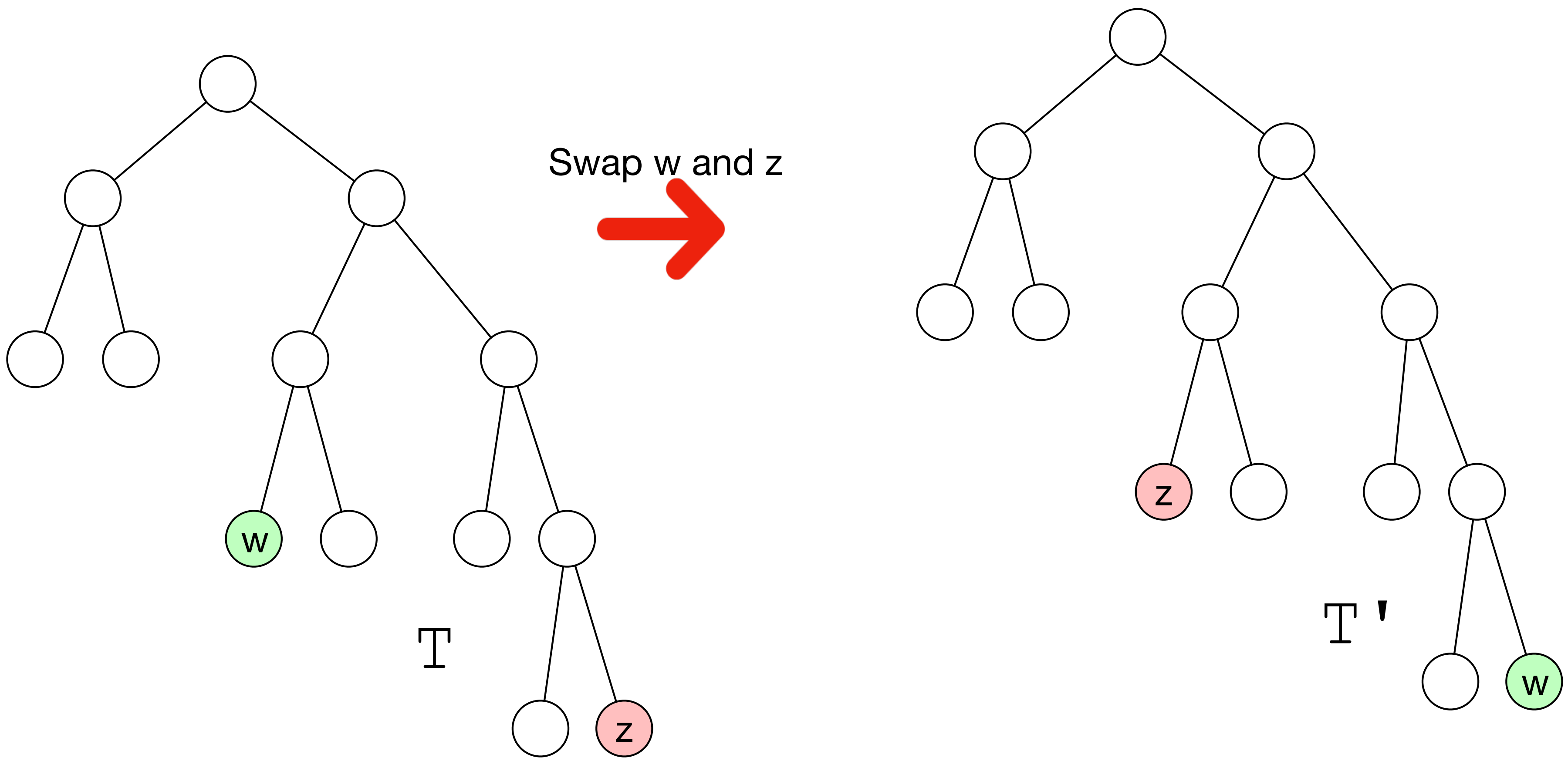
- Proof:
  - Let *y* and *w* be the two symbols with the lowest frequency
  - If there is a tie, take the ones with the biggest depth
  - We are going to show that we can transform the tree into a better (or equally good) one where they are neighbors
  - Assume that $d_T(w) \geq d_T(y)$

# Huffman Coding

- Assume that there is another leaf $z$ at larger distance than $w$
  - $z$ has higher frequency and higher distance from root



# Huffman Coding




- How does the B-value change?
- $B(T') - B(T) = \left(\underbrace{f(z) - f(w)}_{\geq 0}\right)\left(\underbrace{d_T(w) - d_T(z)}_{\leq 0}\right)$
- It goes down, i.e. the new tree is better

# Huffman Coding

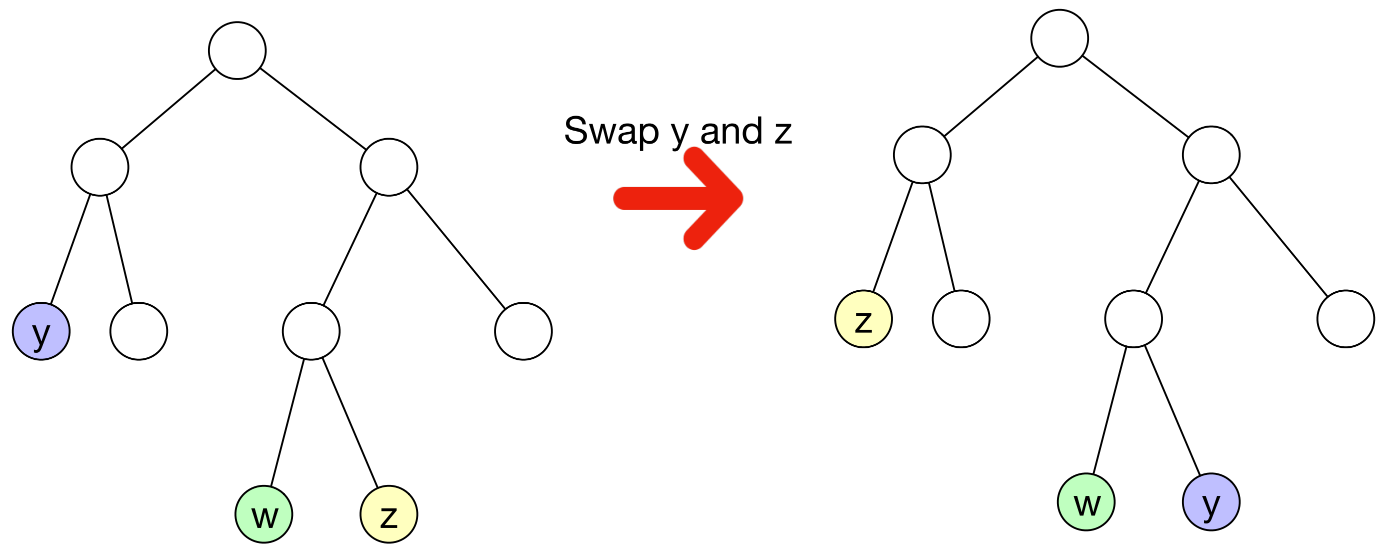
- We now know that we can have a better or equally good tree where $w$ is a leaf at furthest distance from the root
- Case distinctions based on the sibling of $w$
  - $y$ and $w$ are siblings
  - $w$ has another sibling
  - $w$ has no sibling

# Huffman Coding

- Case Distinction:
  - Case 1: *y* and *w* are siblings
    - We are done, this is what we are supposed to show

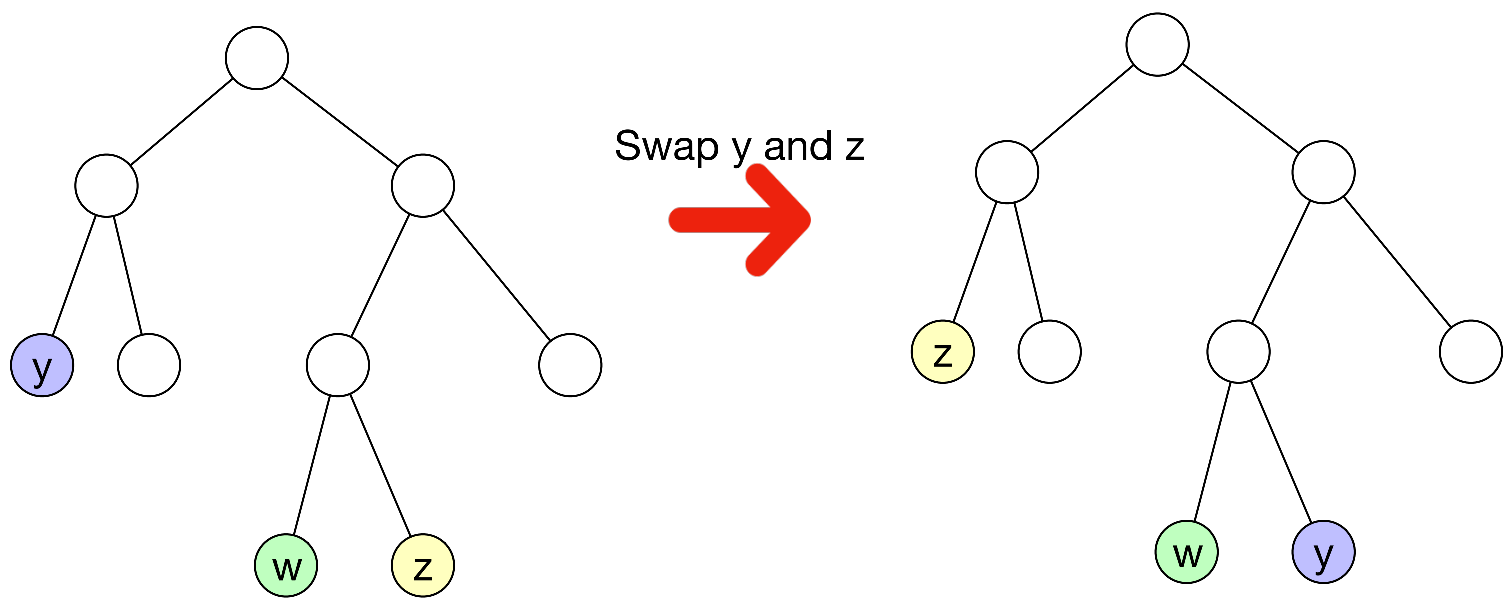
# Huffman Coding

- Case 2: *w* has a sibling *z*
  - Then $f(z) \geq f(y)$ and $d_T(z) = d_T(w) \geq d_T(y)$




$$B(T') - B(T) = \left(f(y) - f(w)\right)\left(d_T(w) - d_T(y)\right)$$

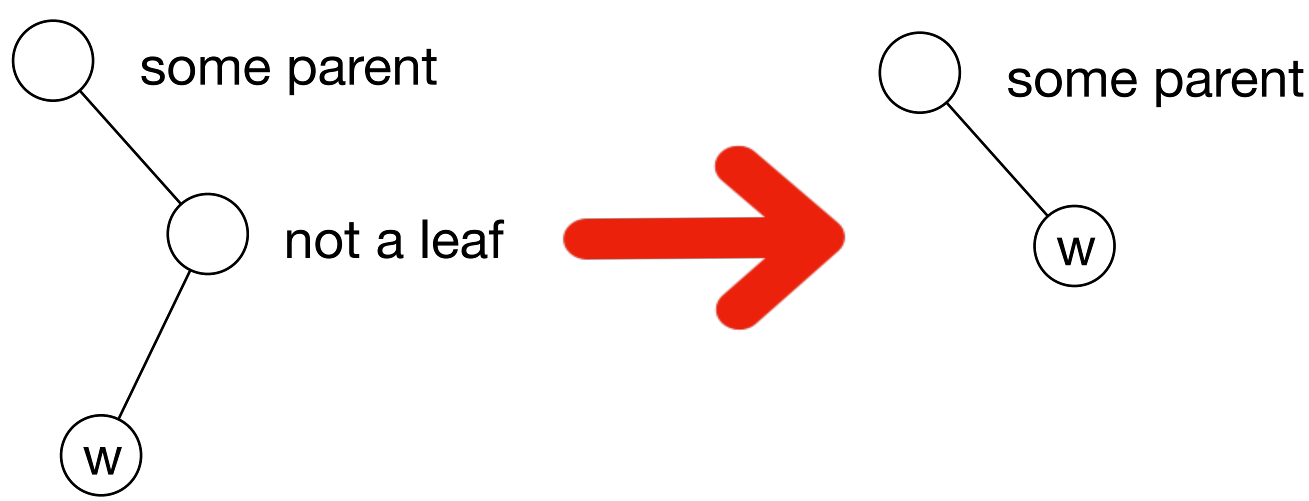
# Huffman Coding




- Since $f(z) \geq f(y)$ and $d_T(z) = d_T(w) \geq d_T(y)$
- If we swap y and z
- $B(T') - B(T) = \Big(\underbrace{f(y) - f(z)}_{\leq 0}\Big)\Big(\underbrace{d_T(z) - d_T(y)}_{\geq 0}\Big)$ is zero or negative
- We are lowering the $B$-value, so we get a better (or equally good) tree

# Huffman Coding

- Case 3:
  - *w* has no sibling
  - Then we can move *w* up and get a better tree
  - The only thing that changes is $d_T(w)$, which becomes lower



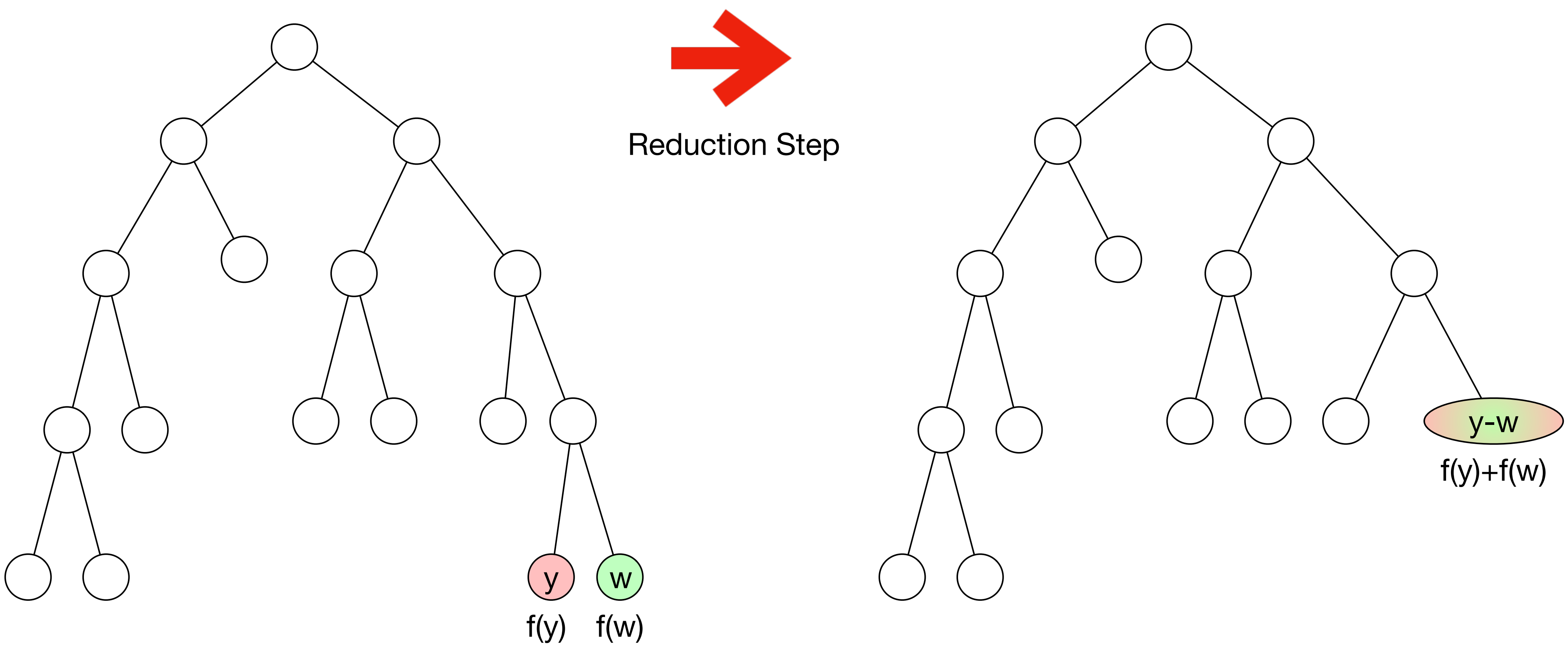

Move w up to get a better tree

# Huffman Coding

- The "Greedy" property
  - A greedy algorithm is a step-by-step algorithm
    - At each step, make an optimal decision based only on the information in the current step
  - In our case:
    - How do we reduce the problem of finding an optimal tree to a simpler one
    - Already know that the two least frequent symbols are siblings in an optimal tree

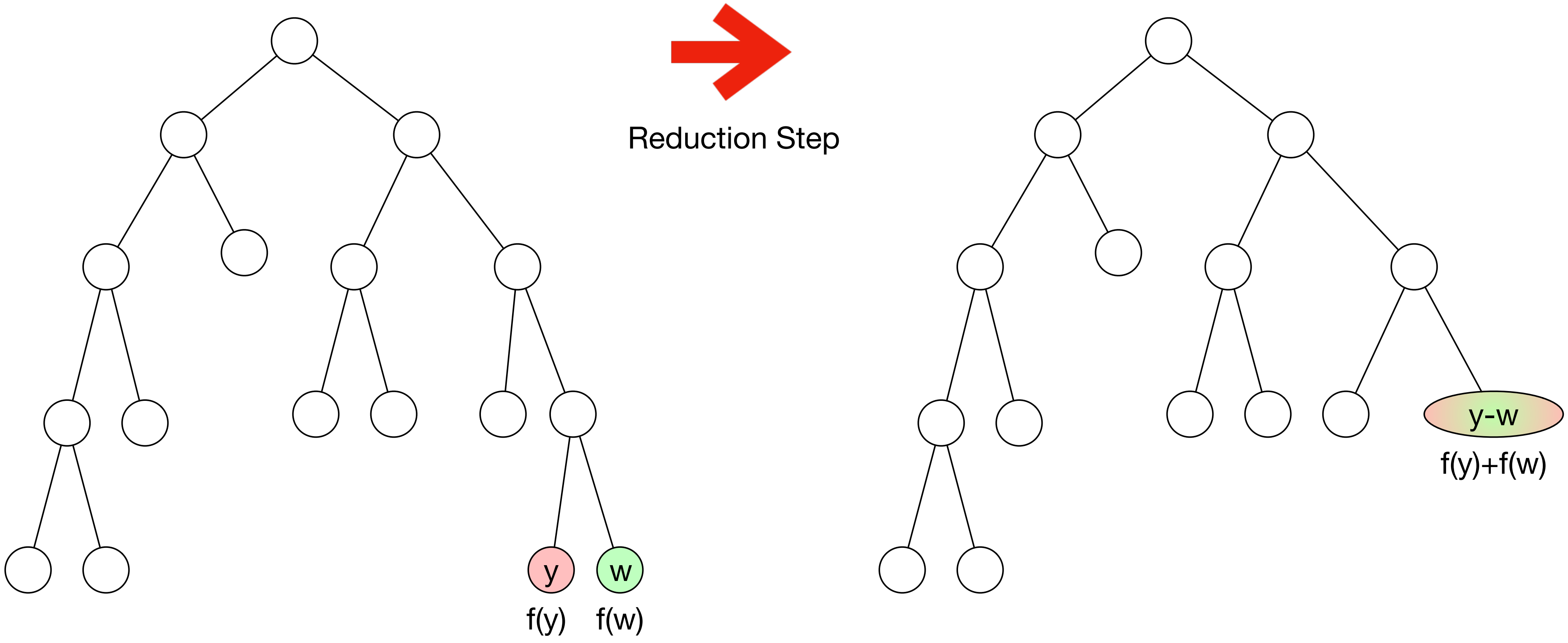
# Huffman Coding

- Reduction step:
  - Merge the two least frequent code symbols



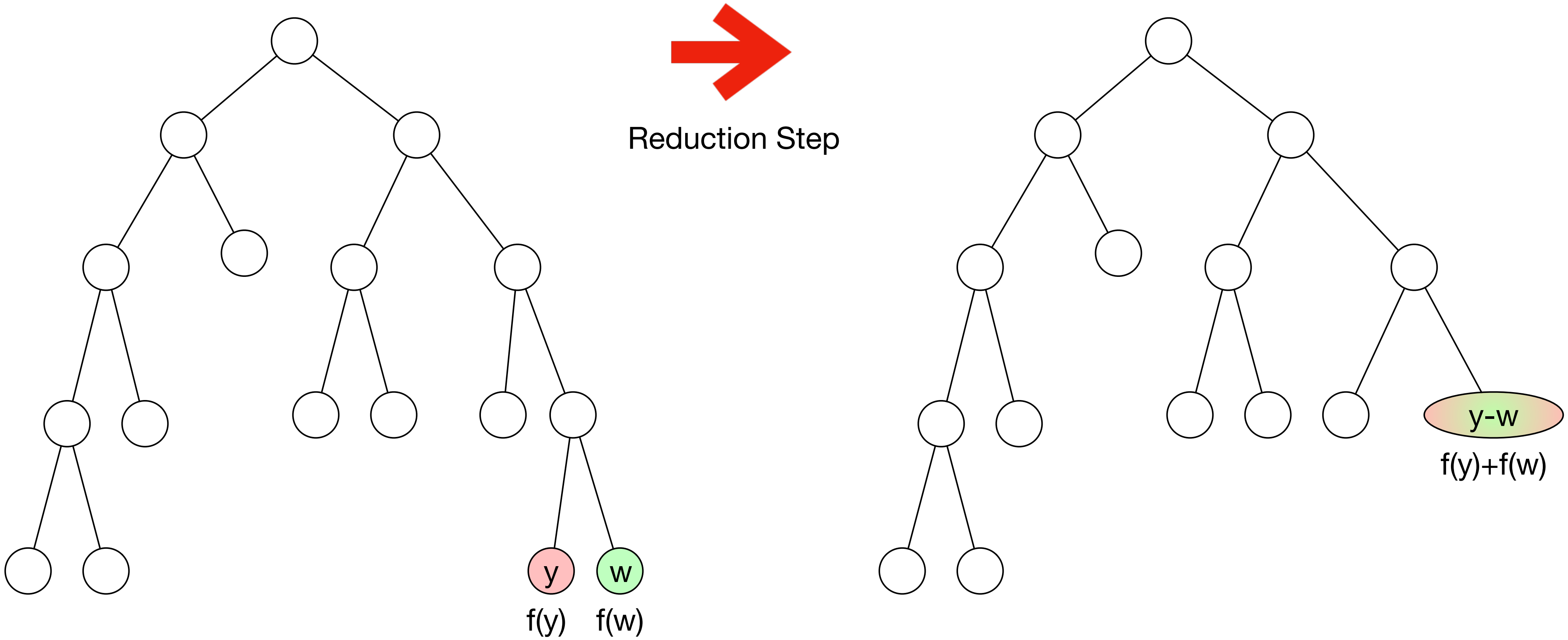
# Huffman Coding

- Create a new 'character' $y\bar{w}$
- Left: alphabet is $\Sigma$ Right: alphabet is $\Sigma - \{y, w\} \cup \{y\bar{w}\}$



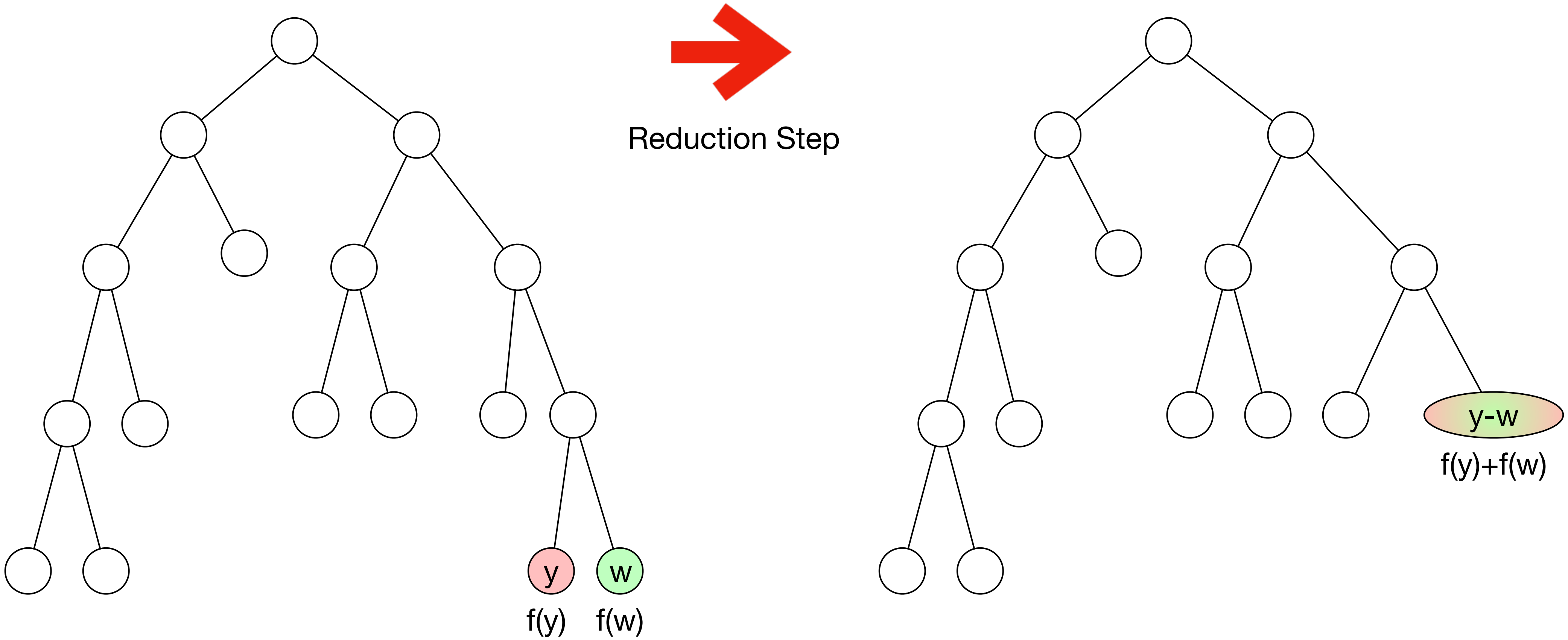
# Huffman Coding

- Create a new 'character' $y\bar{w}$
- Frequency is $f(y\bar{w}) = f(y) + f(w)$



# Huffman Coding

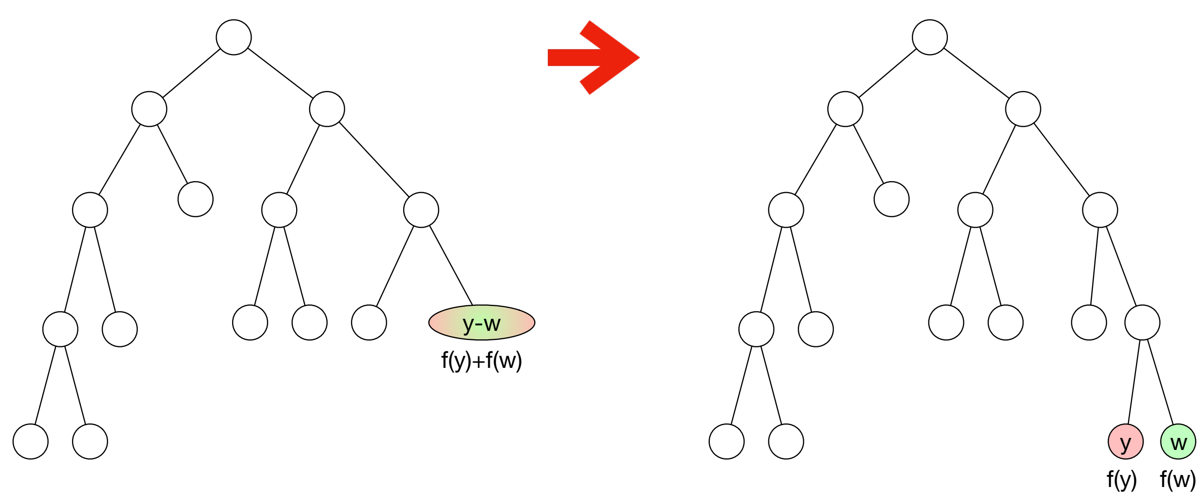
- Everything else stays the same



# Huffman Coding

- Need to show that this step does not counter optimality.

- Lemma: If the tree $T$ obtained on the alphabet $\Sigma - \{y, w\} + \{y\bar{w}\}$ is optimal, then the tree $T'$ replacing the $y\bar{w}$ node with $y$ and $w$ is also optimal



# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

$$B(T') - B(T) = \sum_{c \in \Sigma'} f_{T'}(c) d_{T'}(c) - \sum_{c \in \Sigma} f_T(c) d_T(c)$$

Using the definition

# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

$$B(T') - B(T) = \sum_{c \in \Sigma'} f_{T'}(c) d_{T'}(c) - \sum_{c \in \Sigma} f_T(c) d_T(c)$$

$$= f_{T'}(\overline{yw}) d_{T'}(\overline{yw}) - f_T(y) d_T(y) - f_T(w) d_T(w)$$

We are summing up mostly over the same elements,
so most addends cancel out and this is what it is left

# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

$$B(T') - B(T) = \sum_{c \in \Sigma'} f_{T'}(c) d_{T'}(c) - \sum_{c \in \Sigma} f_T(c) d_T(c)$$

$$= f_{T'}(y\bar{w}) d_{T'}(y\bar{w}) - f_T(y) d_T(y) - f_T(w) d_T(w)$$

$$= f_{T'}(y\bar{w}) d_{T'}(y\bar{w}) - f_T(y) d_T(y) - f_T(w) d_T(y)$$

y and w are siblings and therefore have the same distance from the root

# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

$$B(T') - B(T) = \sum_{c \in \Sigma'} f_{T'}(c) d_{T'}(c) - \sum_{c \in \Sigma} f_T(c) d_T(c)$$

$$= f_{T'}(y\bar{w}) d_{T'}(y\bar{w}) - f_T(y) d_T(y) - f_T(w) d_T(w)$$

$$= f_{T'}(y\bar{w}) d_{T'}(y\bar{w}) - f_T(y) d_T(y) - f_T(w) d_T(y)$$

$$= (f_T(y) + f_T(w))(d_T(y) - 1) - f_T(y) d_T(y) - f_T(w) d_T(y)$$

The combined node is located at a level one up compared to the single nodes for y and w

# Huffman Coding

- Proof:
  - First we calculate the change in the B-values

$$B(T') - B(T) = \sum_{c \in \Sigma'} f_{T'}(c) d_{T'}(c) - \sum_{c \in \Sigma} f_T(c) d_T(c)$$

$$= f_{T'}(\bar{yw}) d_{T'}(\bar{yw}) - f_T(y) d_T(y) - f_T(w) d_T(w)$$

$$= f_{T'}(\bar{yw}) d_{T'}(\bar{yw}) - f_T(y) d_T(y) - f_T(w) d_T(y)$$

$$= (f_T(y) + f_T(w))(d_T(y) - 1) - f_T(y) d_T(y) - f_T(w) d_T(y)$$

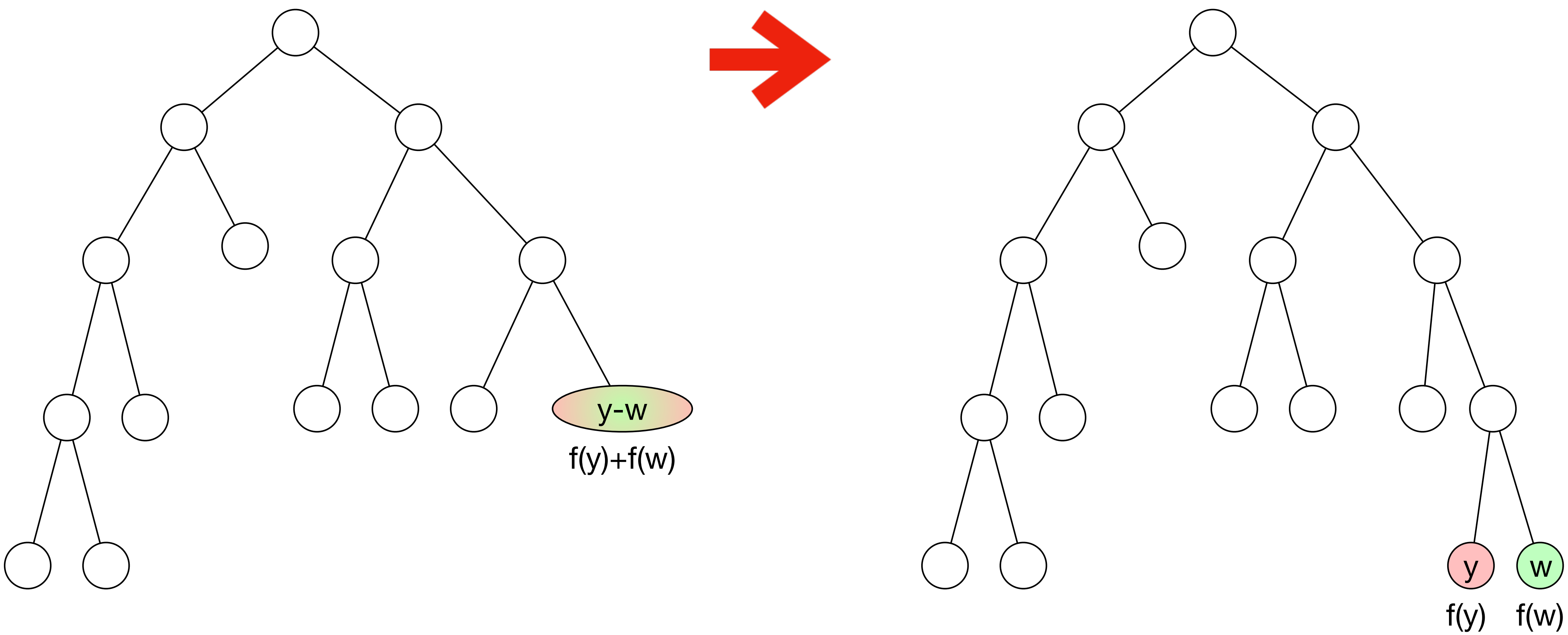
$$= - f_T(y) - f_T(w)$$

# Huffman Coding

- So, by dividing the node $y\bar{w}$ we have to pay a penalty of $f(y) + f(w)$.

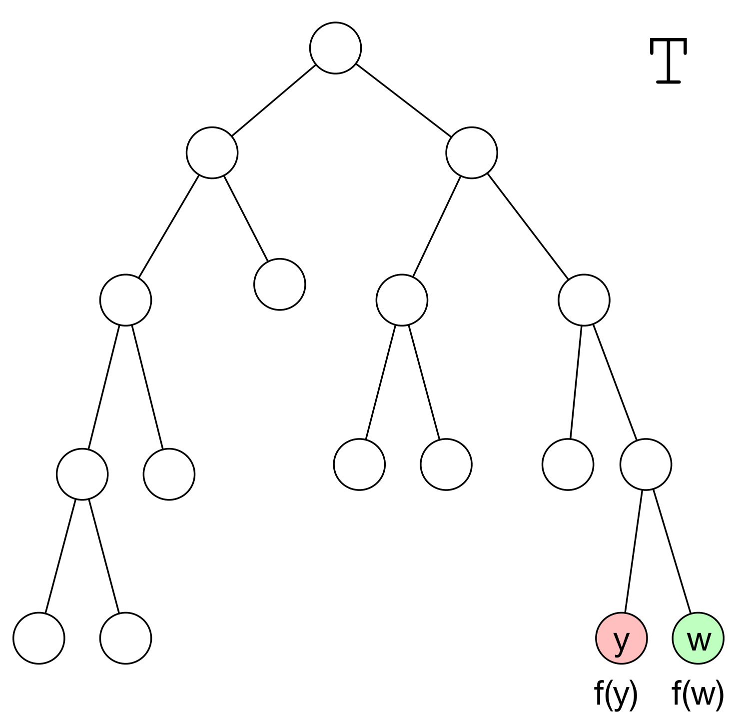
# Huffman Coding

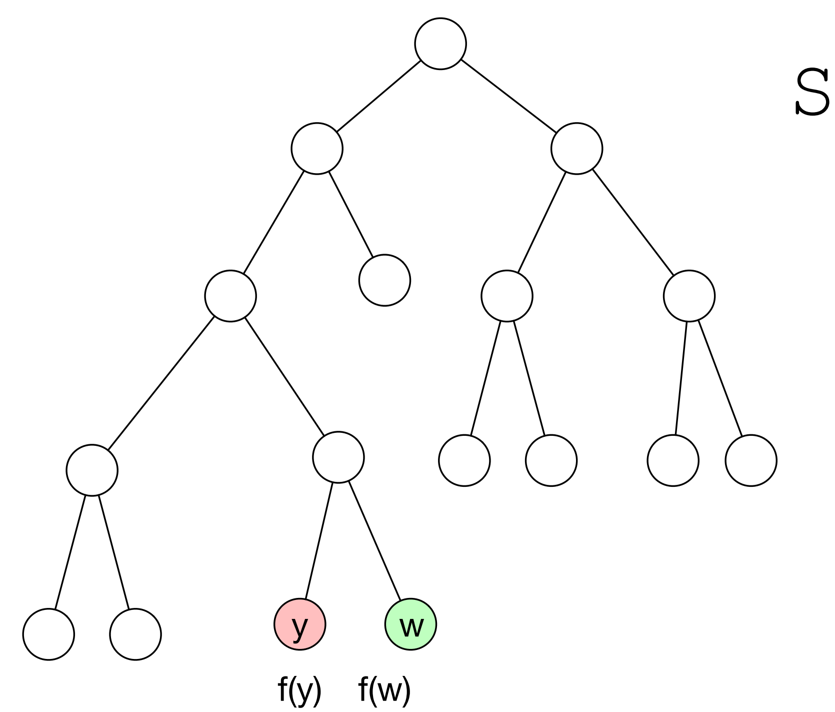
- Now, assume that the left tree is optimal and the right tree is not optimal



# Huffman Coding

- Then there exists a tree $S$ that is better the tree with $y$ and $w$
- We can assume that in this tree, $y$ and $w$ are leave nodes because of the previous lemma

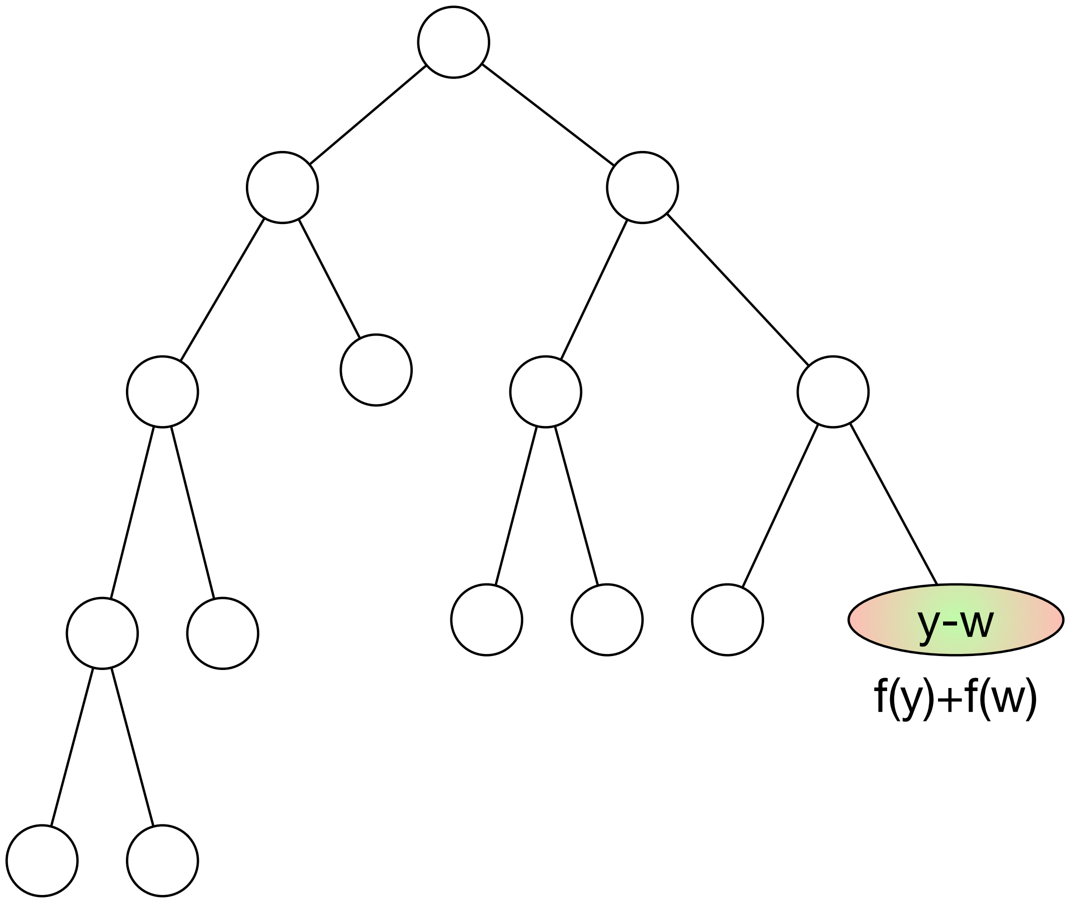


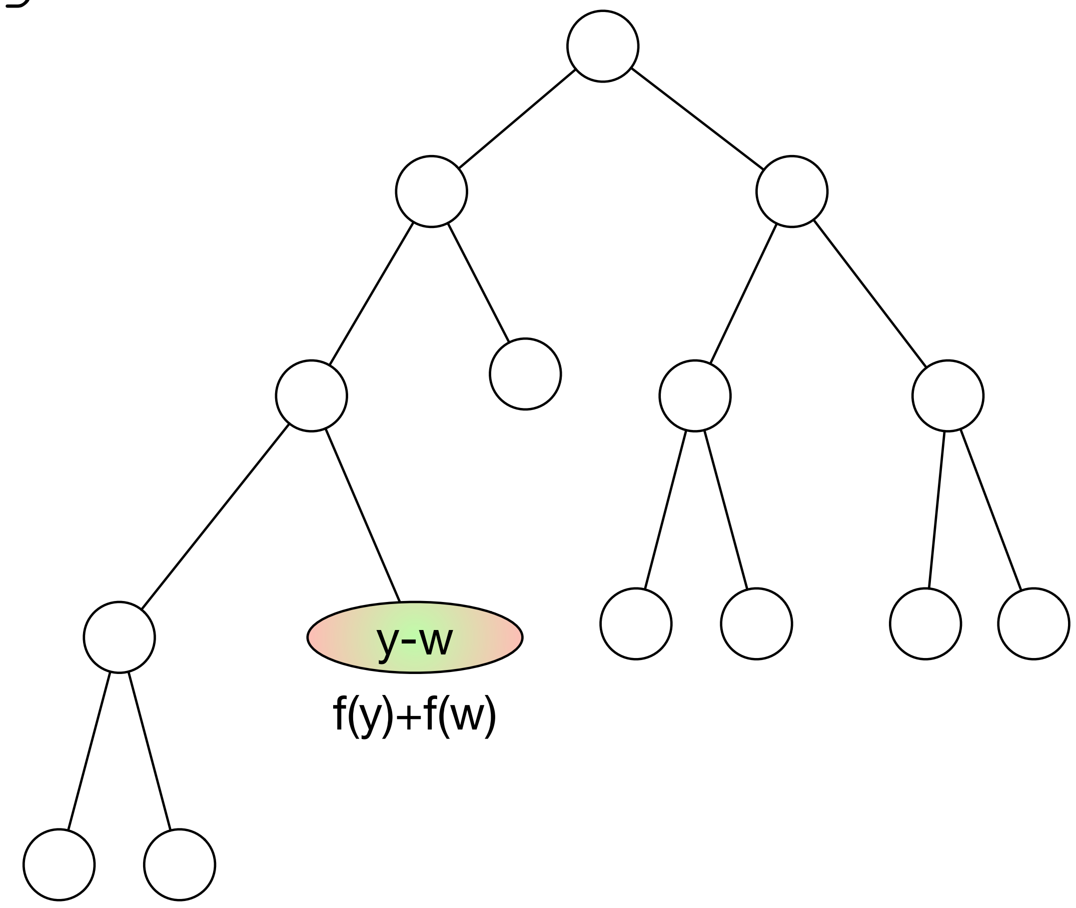
# Huffman Coding

- We now do the same merge step for $S$ and $T$

T after merging

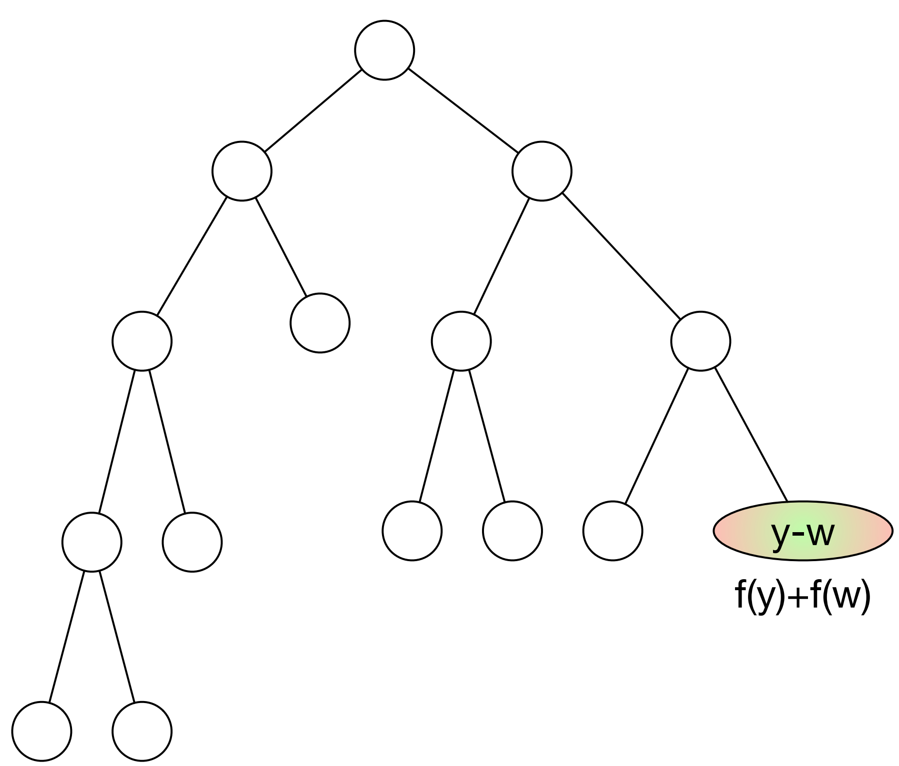


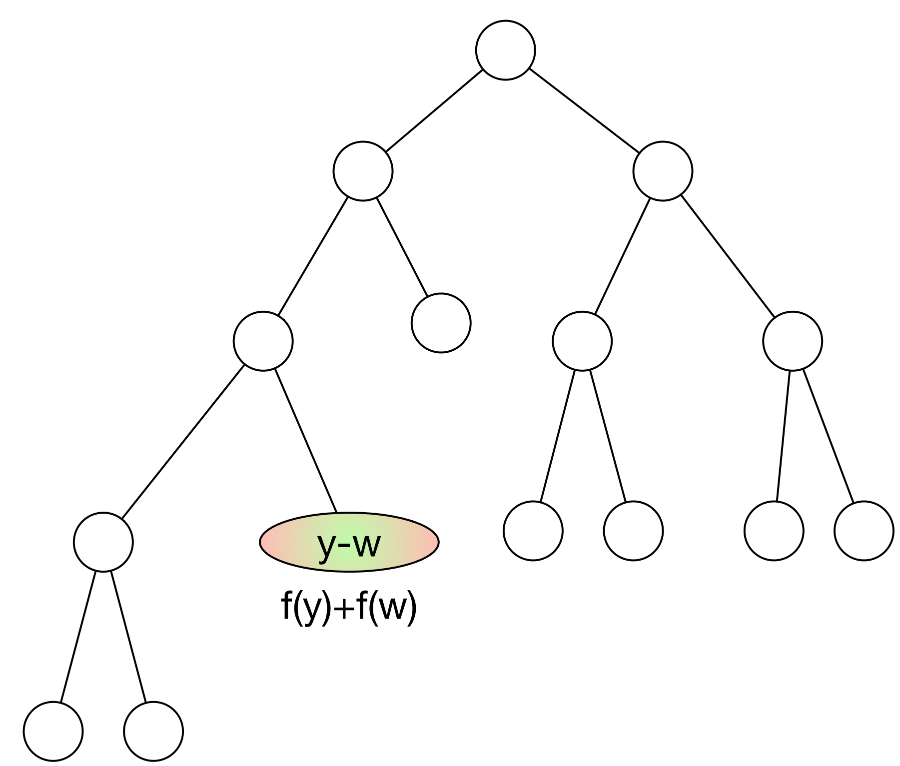

S after



# Huffman Coding

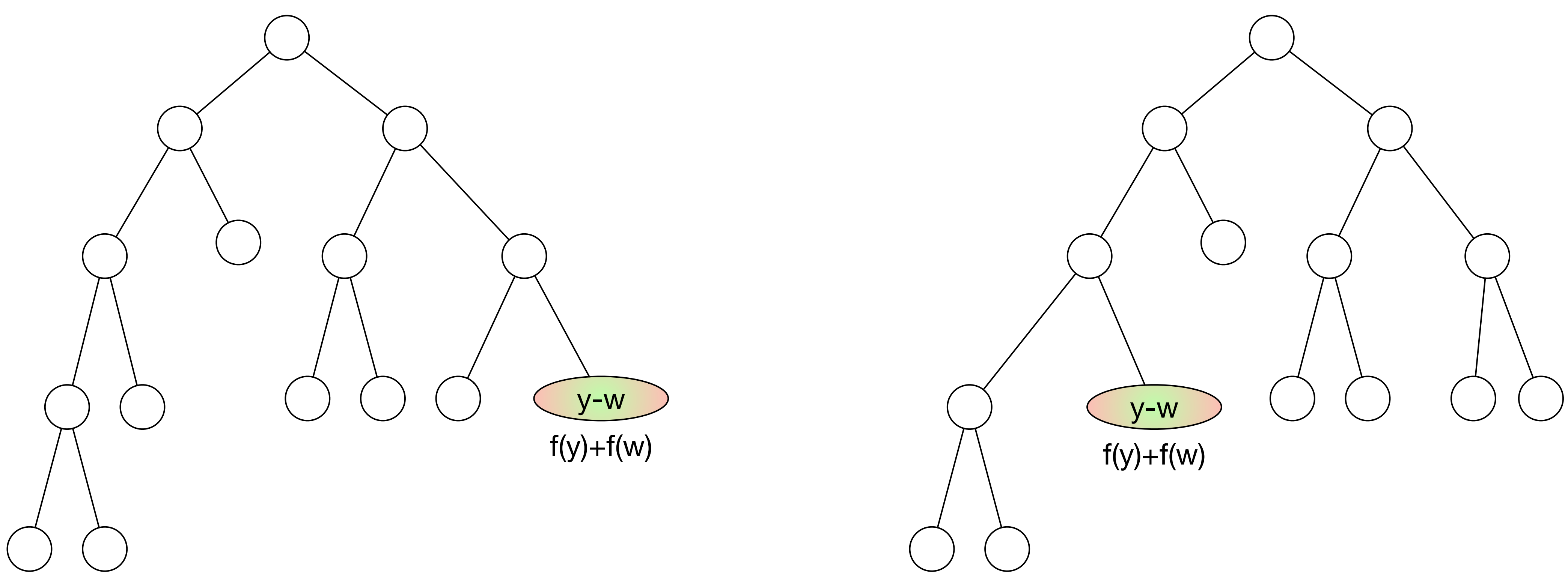
- The B-value for the tree on the right is the B-value of S minus $f(w) + f(y)$
- Which is equal or worse than of the tree on the left



# Huffman Coding

- The B-value for the tree on the right is the B-value of S minus $f(w) + f(y)$
- Which is equal or worse than of the tree on the left
- Which is the B-value of T plus $f(w) + f(y)$

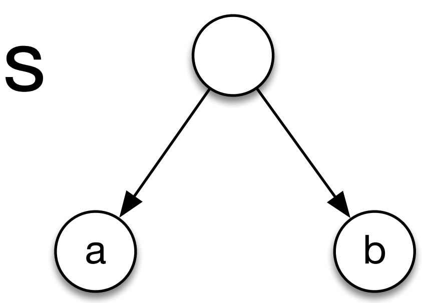


# Huffman Coding

- Thus, S does not have a better B-value

# Huffman Coding

- Huffman's algorithm:
  - If there is only one symbol, create a single node tree
  - Otherwise, select the two most infrequent symbols
  - Combine them with a common ancestor
  - Give the common ancestor the sum of the frequencies
  - Treat the ancestor as a symbol with this frequency
  - Repeat until there is only one symbol



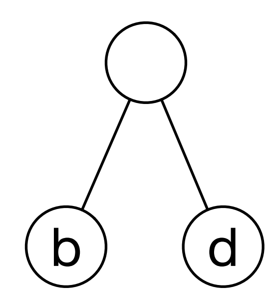
# Huffman Coding

- Example:
  - Absolute frequencies are
    - a — 120
    - b — 29
    - c — 534
    - d — 34
    - e — 2549
    - f — 321
    - g — 45

# Huffman Coding

- Example:
  - Absolute frequencies are
    - a — 120, b — 29, c — 534, d — 34, e — 2549, f — 321, g — 45
  - Combine b and d into (bd)

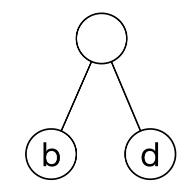
# Huffman Coding

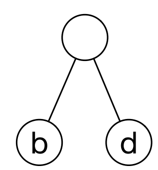
- Example:
  - Absolute frequencies are
    - a — 120, b — 29, c — 534, d — 34, e — 2549, f — 321, g — 45
  - Combine b and d into (bd)
    - a - 120, c - 534, e -2549, f-321, g - 45, 

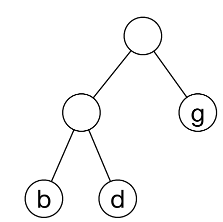
63

```
a — 120, b — 29, c — 534, d — 34, e — 2549,
f — 321, g — 45
```

# Huffman Coding

- Example:
  - a - 120, c - 534, e -2549, f-321, g - 45, 

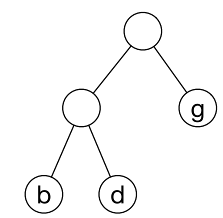
63
  - Combine g and 

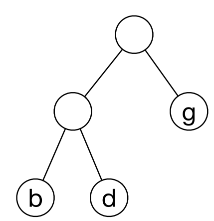
  - a - 120, c - 534, e - 2549, f - 321, 

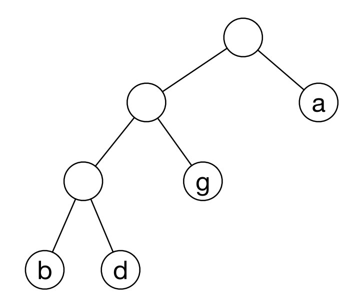
- 108

```
a — 120, b — 29, c — 534, d — 34, e — 2549,
f — 321, g — 45
```

# Huffman Coding

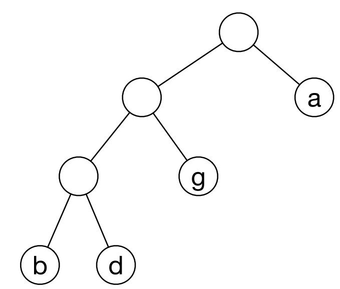
- Example:
  - a - 120, c - 534, e - 2549, f - 321,  - 108

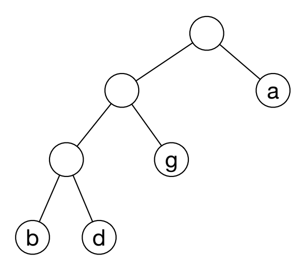
  - Combine a and 

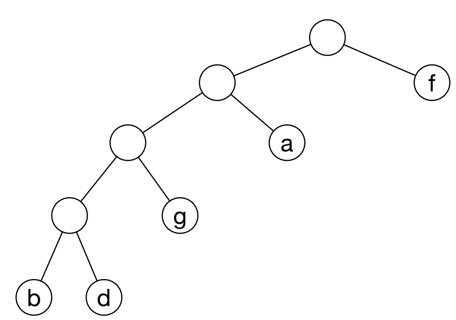
  - Obtain c - 534, e - 2549, f - 321,  - 228


```
a — 120, b — 29, c — 534, d — 34, e — 2549,
f — 321, g — 45
```

# Huffman Coding

- Example:
    - c - 534, e - 2549, f - 321, 

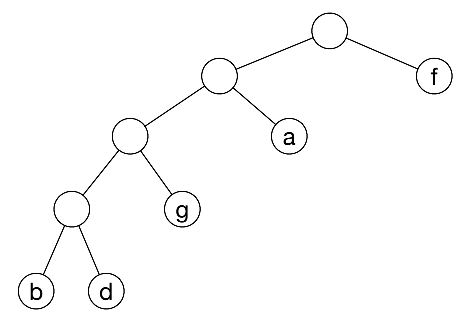
 - 228
    - Combine f and 

    - c - 534, e - 2549, 

 - 549

```
a — 120, b — 29, c — 534, d — 34, e — 2549,
f — 321, g — 45
```

# Huffman Coding

- c - 534, e - 2549,  - 549

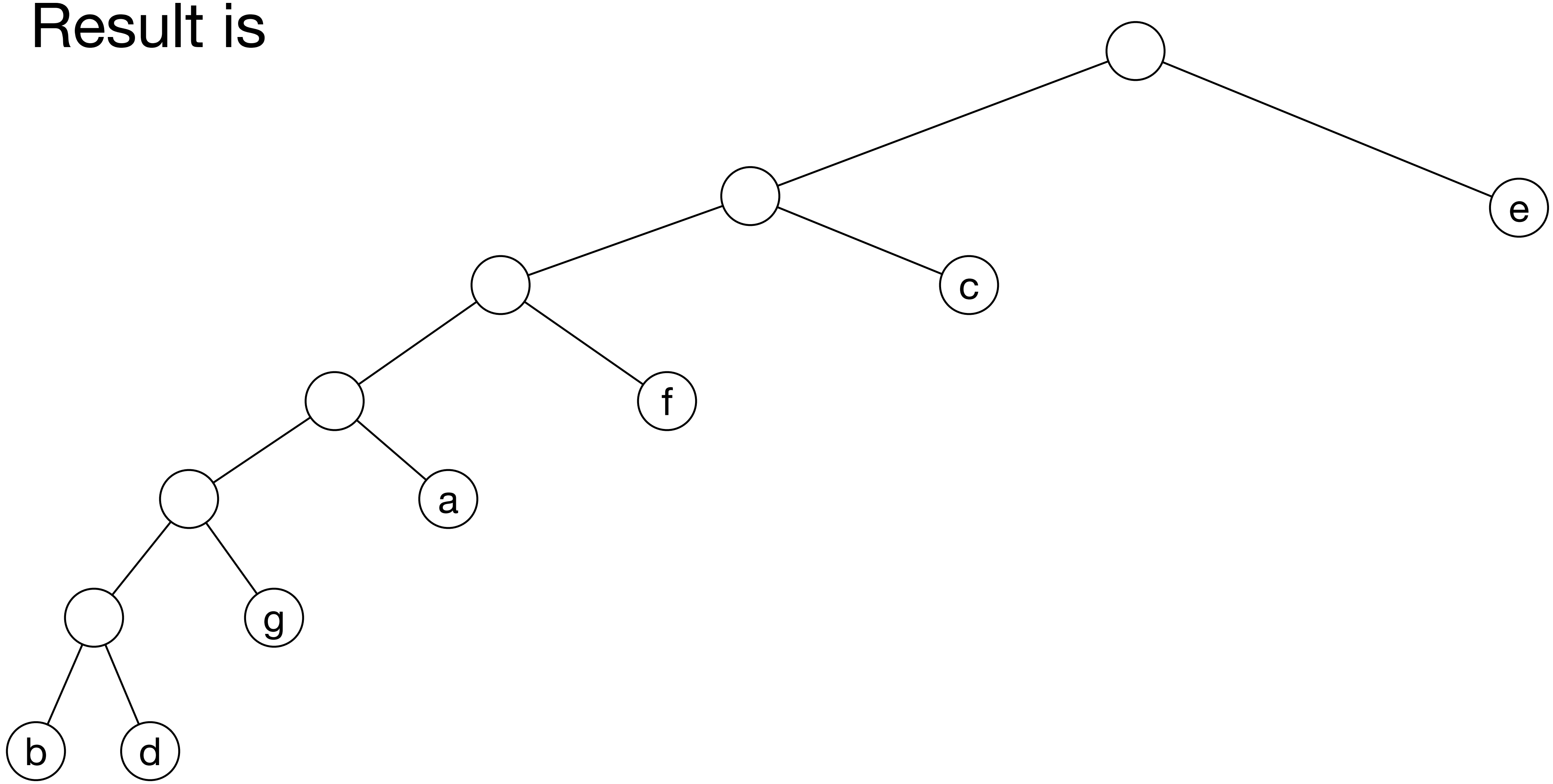


- Combine c with the tree
- Then combine with e

```
a — 120, b — 29, c — 534, d — 34, e — 2549,
f — 321, g — 45
```

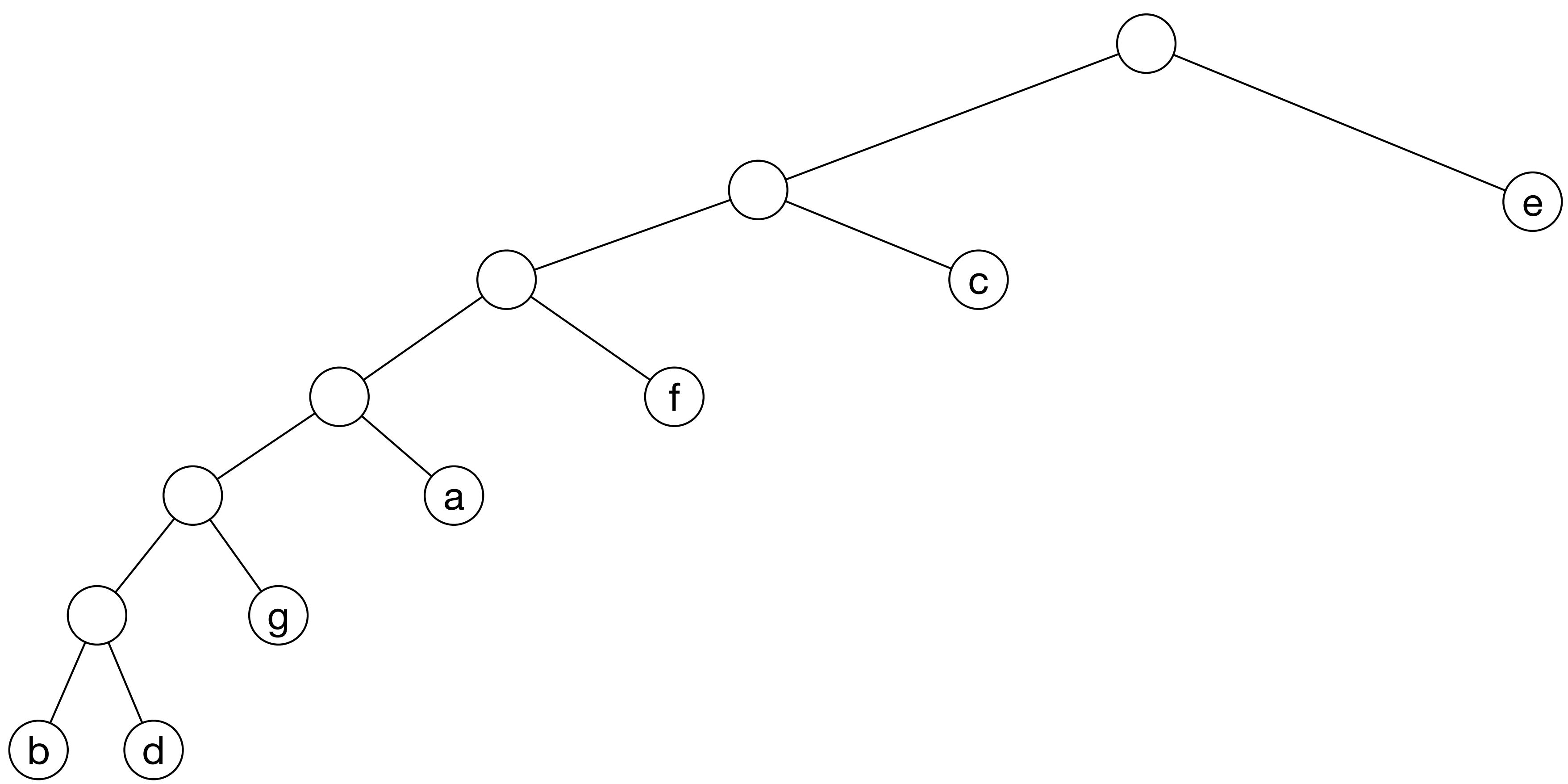
# Huffman Coding

- Result is



# Huffman Coding

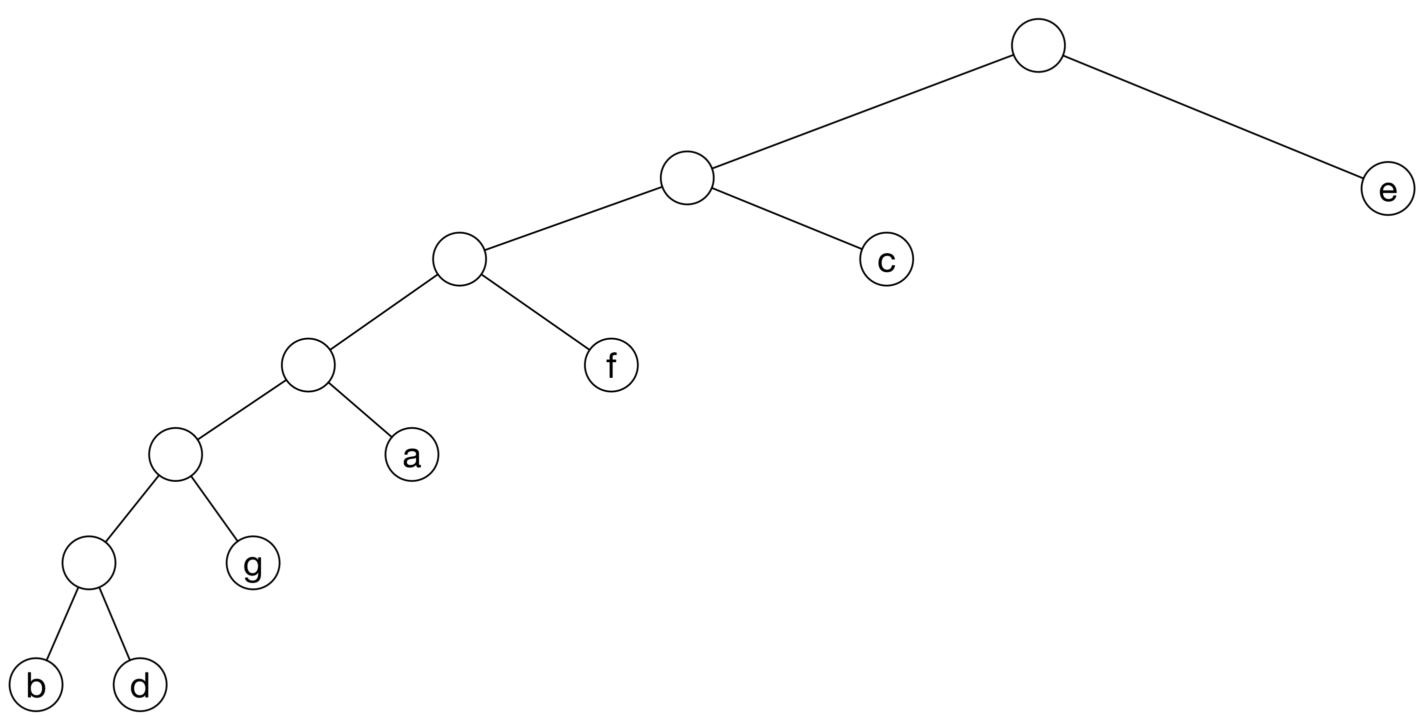
- B-value needs relative frequencies




```
>>> total = 120+29+534+34+2549+321+45
>>> 29/total*6+34/total*6+45/total*5+120/
total*4+321/total*3+534/total*2+2549/total*1
1.5591960352422907
```

# Huffman Coding

- Notice how much choice we have in building this tree
  - We can switch the order of the trees that we put together
  - For this one, the encoding is




```
e - 1
c - 01
f - 001
a - 0001
g - 00001
b - 000000
d - 000001
```

# Huffman Coding

- Try it out yourself
  - a — 0.23
  - e — 0.35
  - i — 0.16
  - o — 0.15
  - u — 0.11

# Huffman Coding

- Solution
  - Have a — 0.23, e — 0.35, i — 0.16, o — 0.15, u — 0.11
  - First combine o and u for 'ou' with frequency  0.26
    - a — 0.23
    - e — 0.35
    - i — 0.16
    - ou — 0.26

# Huffman Coding

- Solution
  - Have a — 0.23, e — 0.35, i — 0.16, ou — 0.26
  - Combine i and a
    - e - 0.35
    - ai - 0.39
    - ou - 0.26

# Huffman Coding

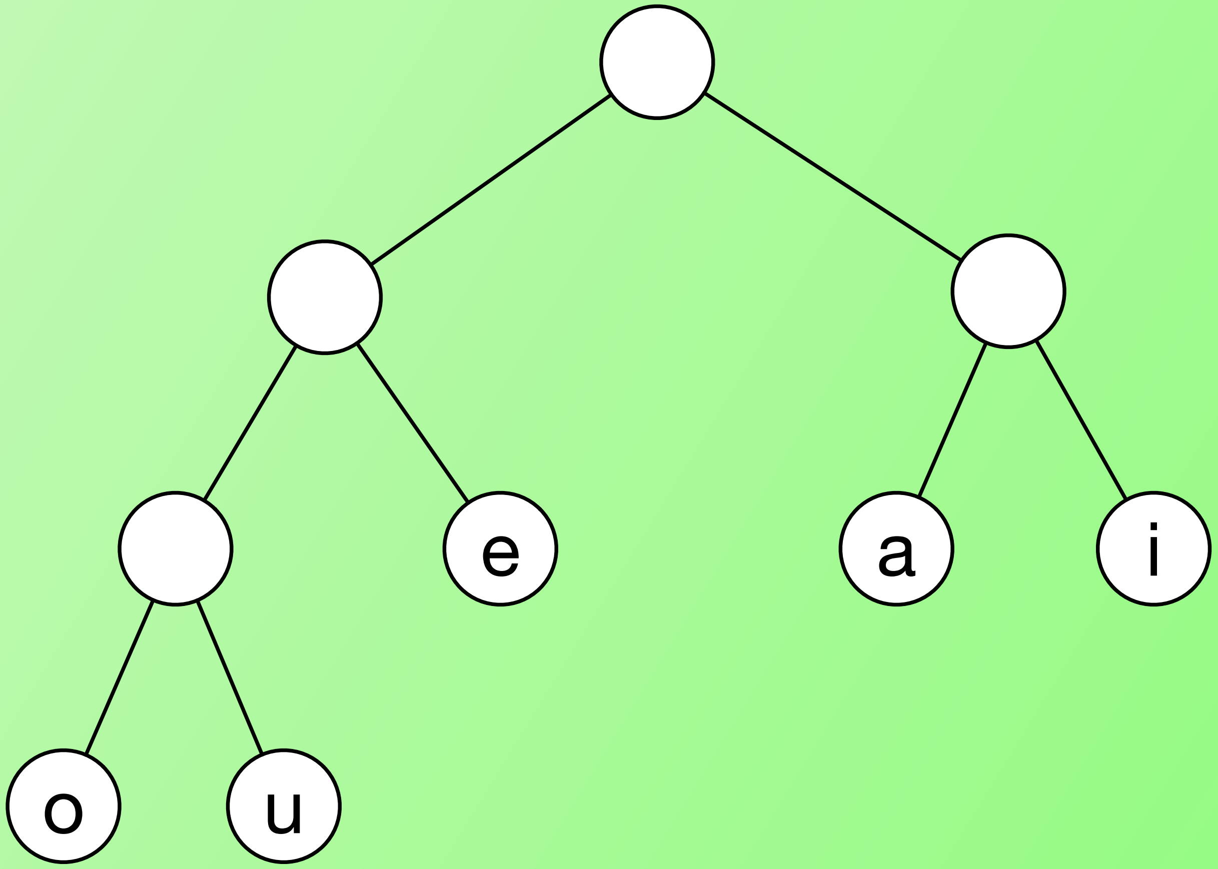
- Solution
  - Have  e - 0.35, ai - 0.39, ou - 0.26
  - Combine ou with e
    - e(ou) 0.61
    - ai 0.39

# Huffman Coding

- Solution
  - Have  e(ou) - 0.61  ai - 0.39
- Combine to get (e(ou)) (ai) with frequency 1.00

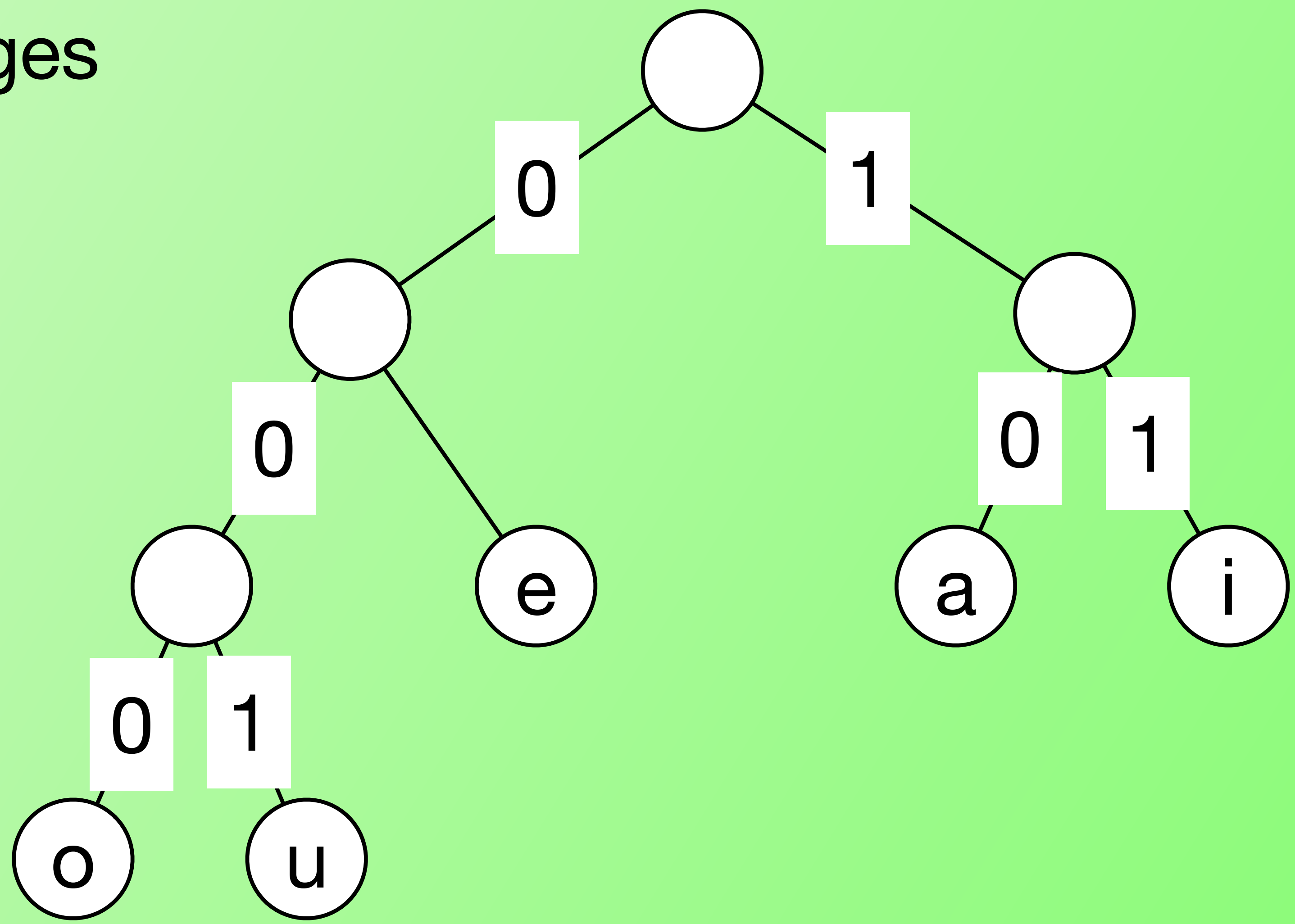
# Huffman Coding

- Solution
  - Have (e(ou)) (ai) with frequency 1.00
  - Translate to tree



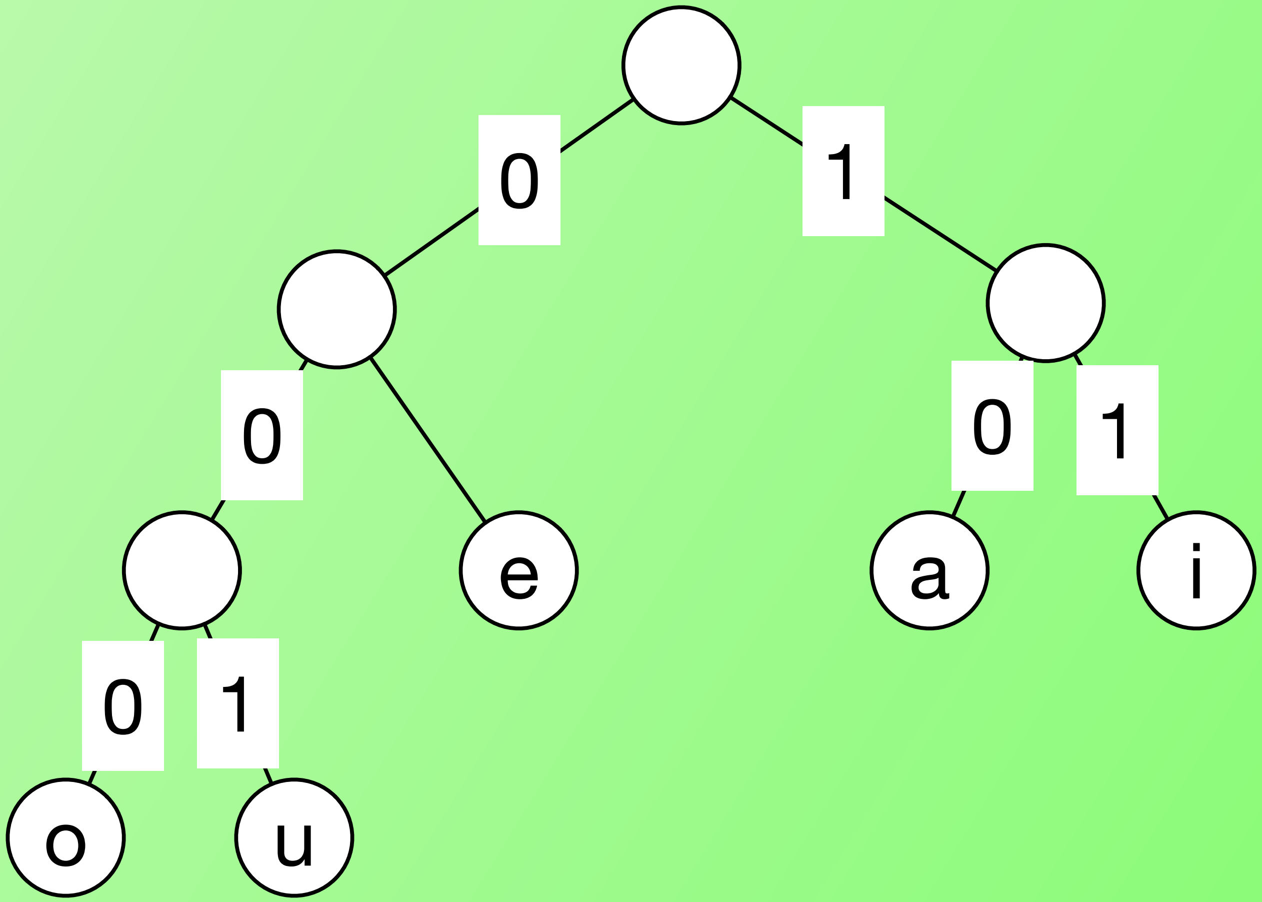
# Huffman Coding

- Solution
  - Label tree edges



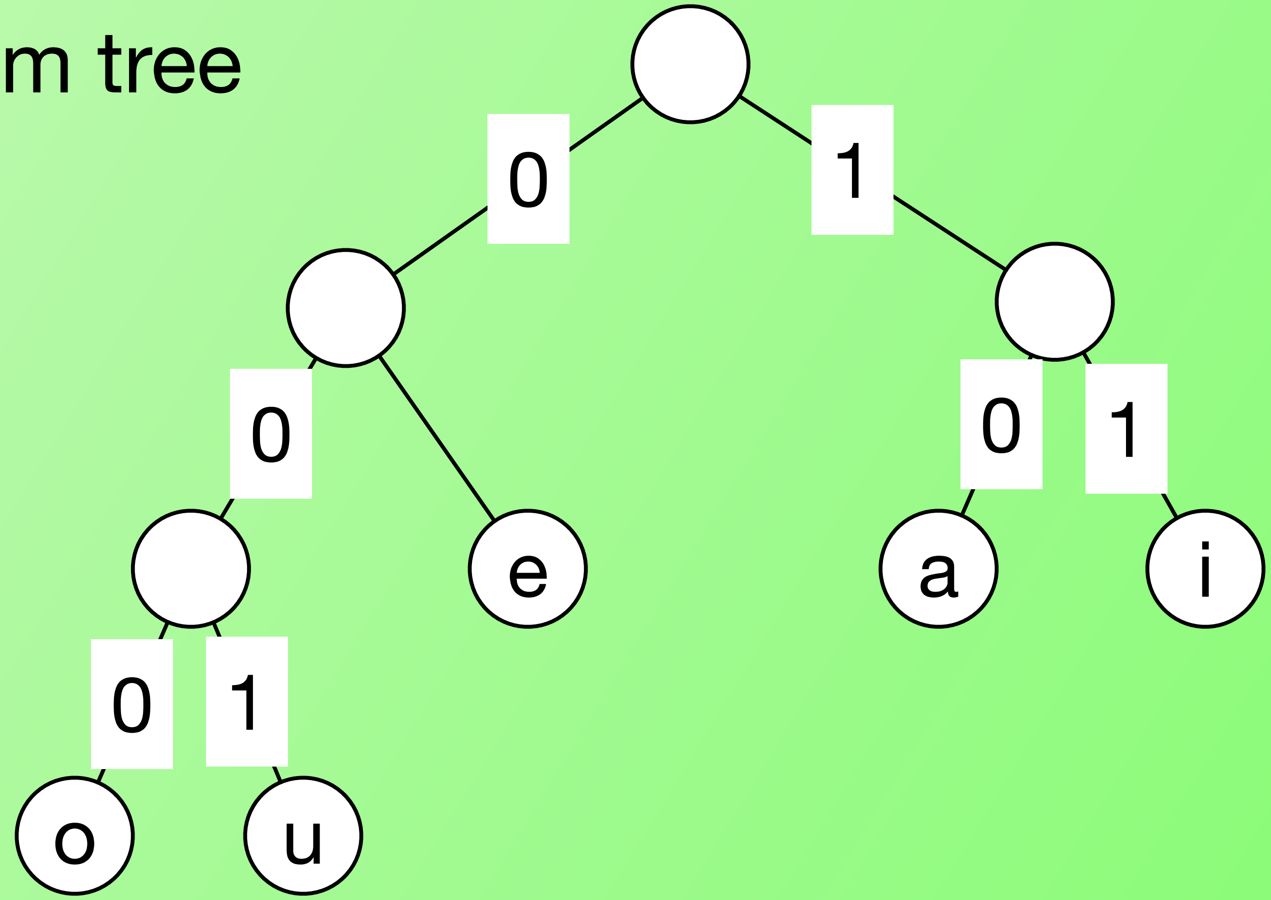
# Huffman Coding

- Solution
  - Read off encoding
  - a — 10
  - e — 01
  - i — 11
  - o — 000
  - u — 001



# Huffman Coding

- Solution
  - Determine B-value from tree
  - a — 0.23
  - e — 0.35
  - i — 0.16
  - o — 0.15
  - u — 0.11




3*0.11+3*0.15+2*0.16+2*0.35+2*0.23=
2.2600000000000002