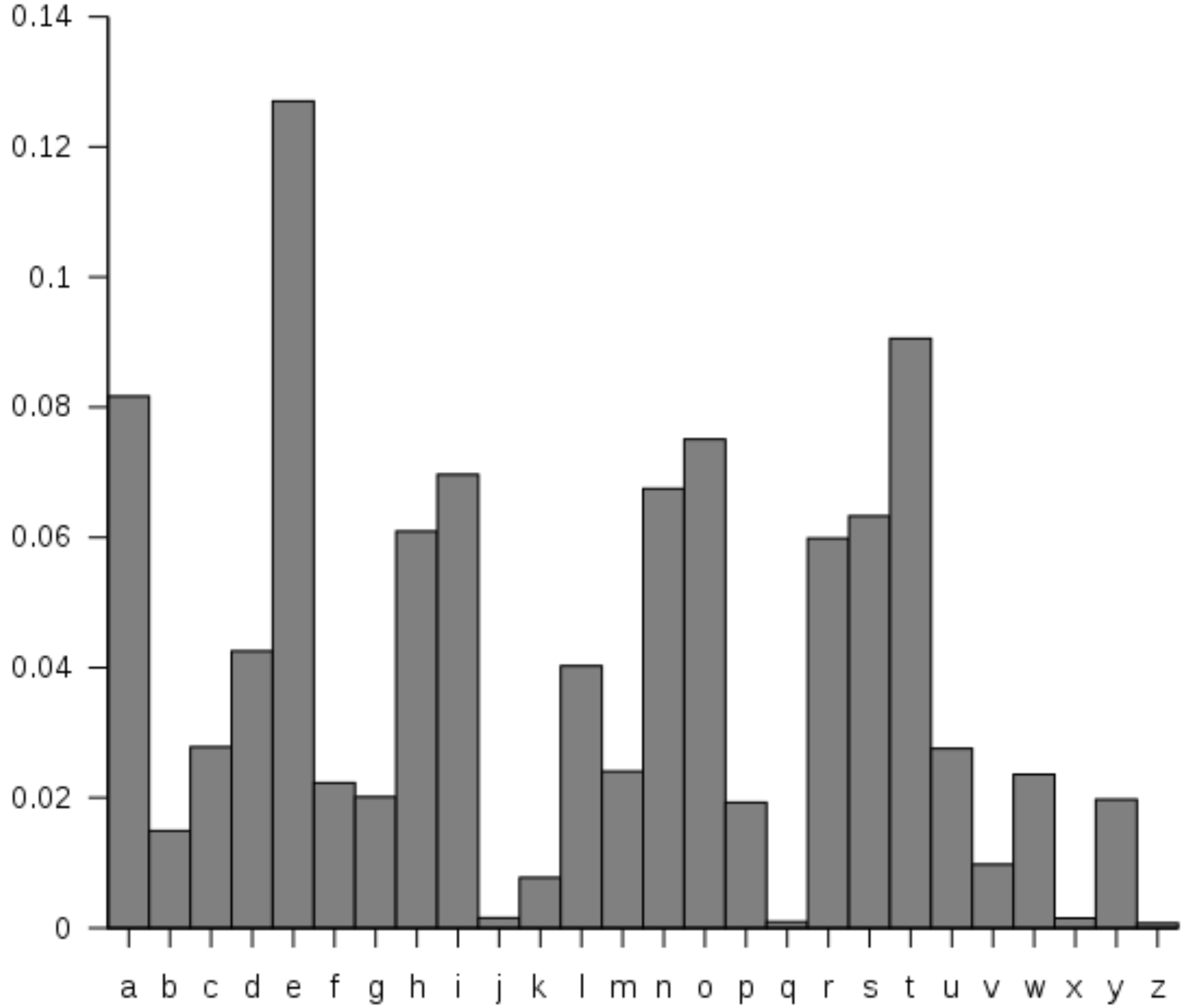
Huffman Encoding

The coolest algorithm *ever*

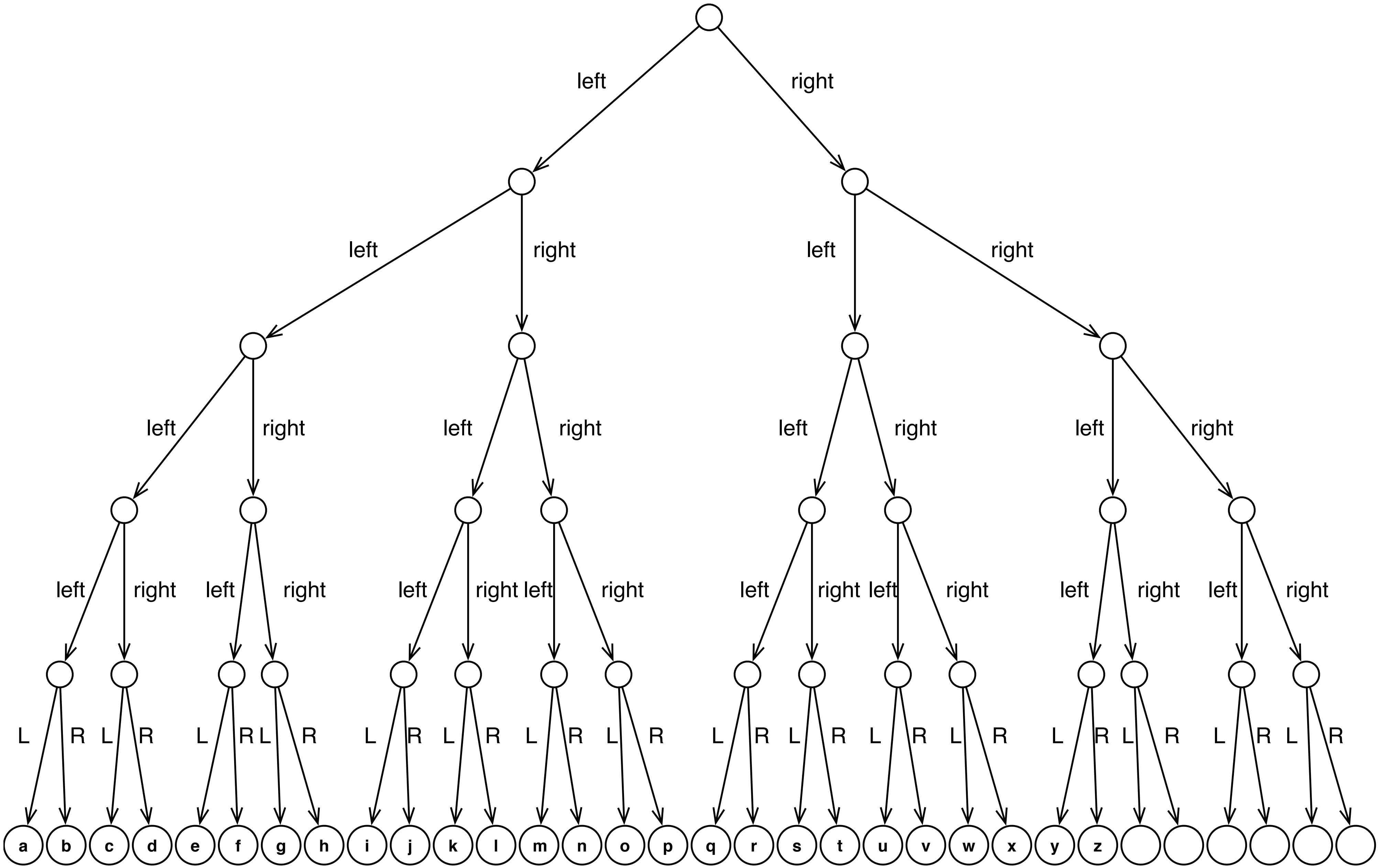
Say we want to devise a system for representing symbolic data using numbers. Maybe A is 1, B is 2, and so on. If we want to represent the 52 upper and lower case letters used in English, plus some other symbols like spaces, commas and periods, we might have 60. Since we represent data on computers using binary digits we might as well use 64 so we use 8 bits, which is 1 byte.

Using this scheme, we can encode the entire paragraph above using 3016 bits (377 bytes).

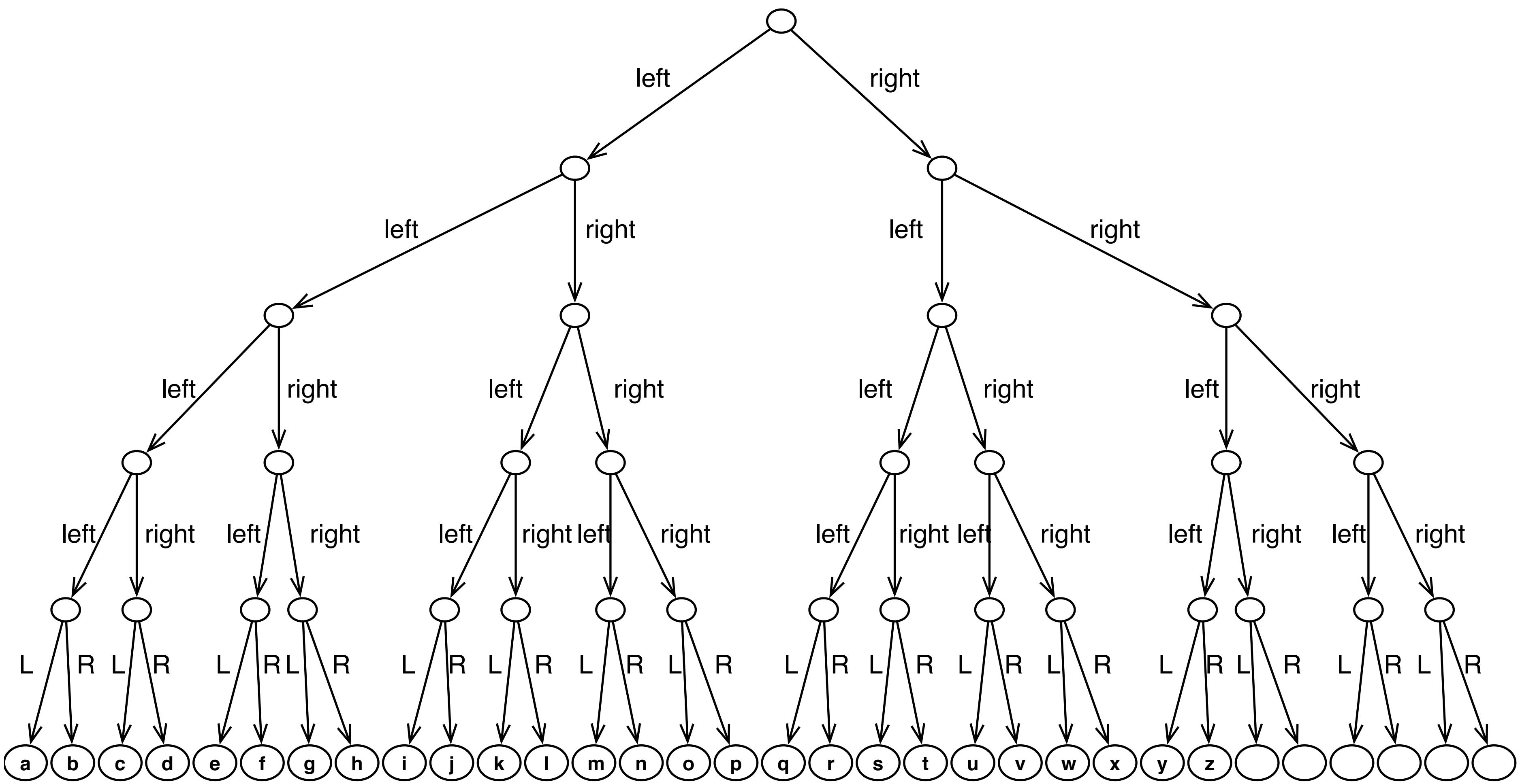
But we could compress this data so it it uses much less memory. Huffman encoding is one way to do this. It is also a really fun algorithm.



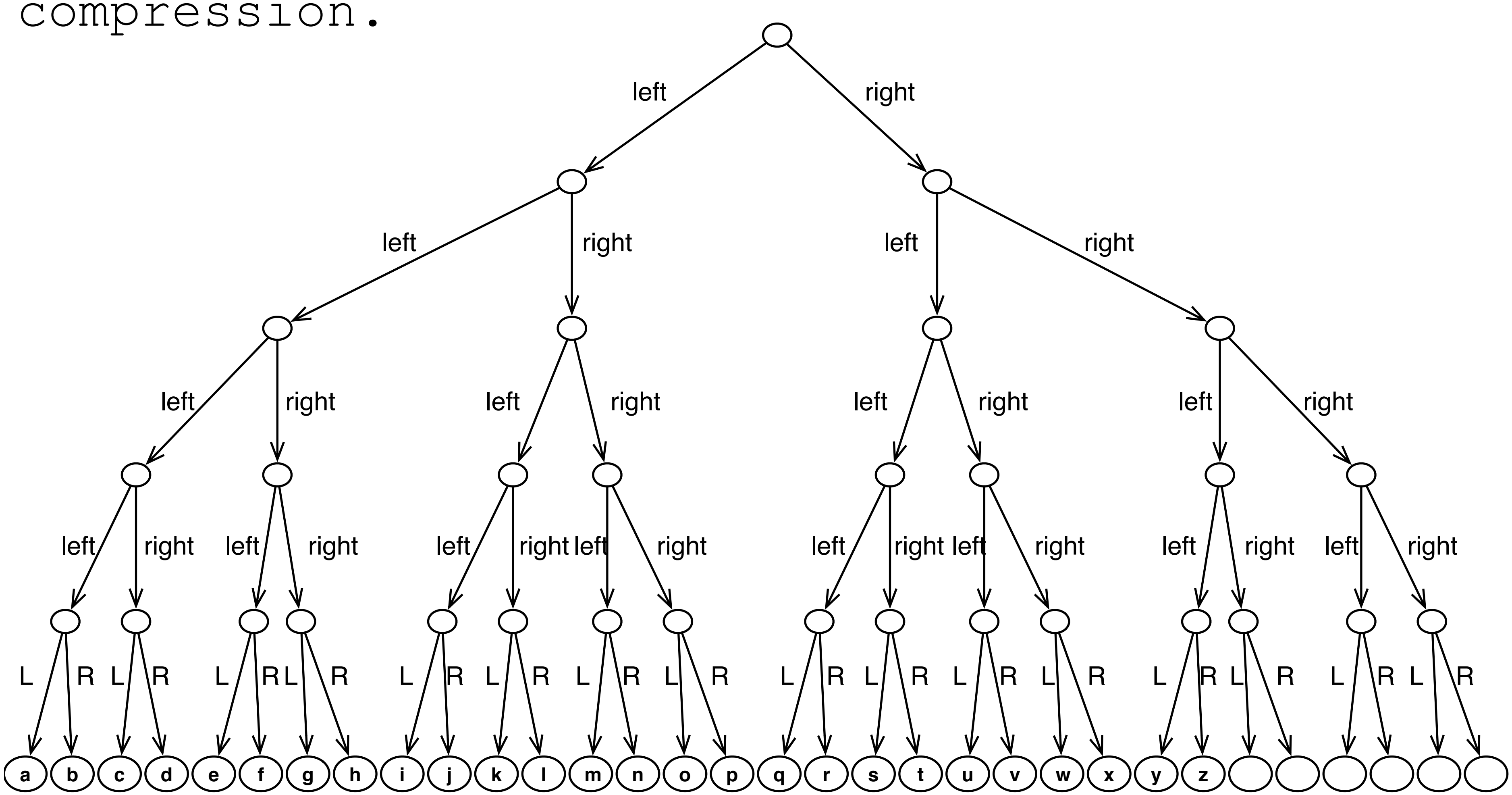
Here's the distribution of letters used in English, according to some guy on Wikipedia. This doesn't include punctuation, or space.


left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
left
right
L
R
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z

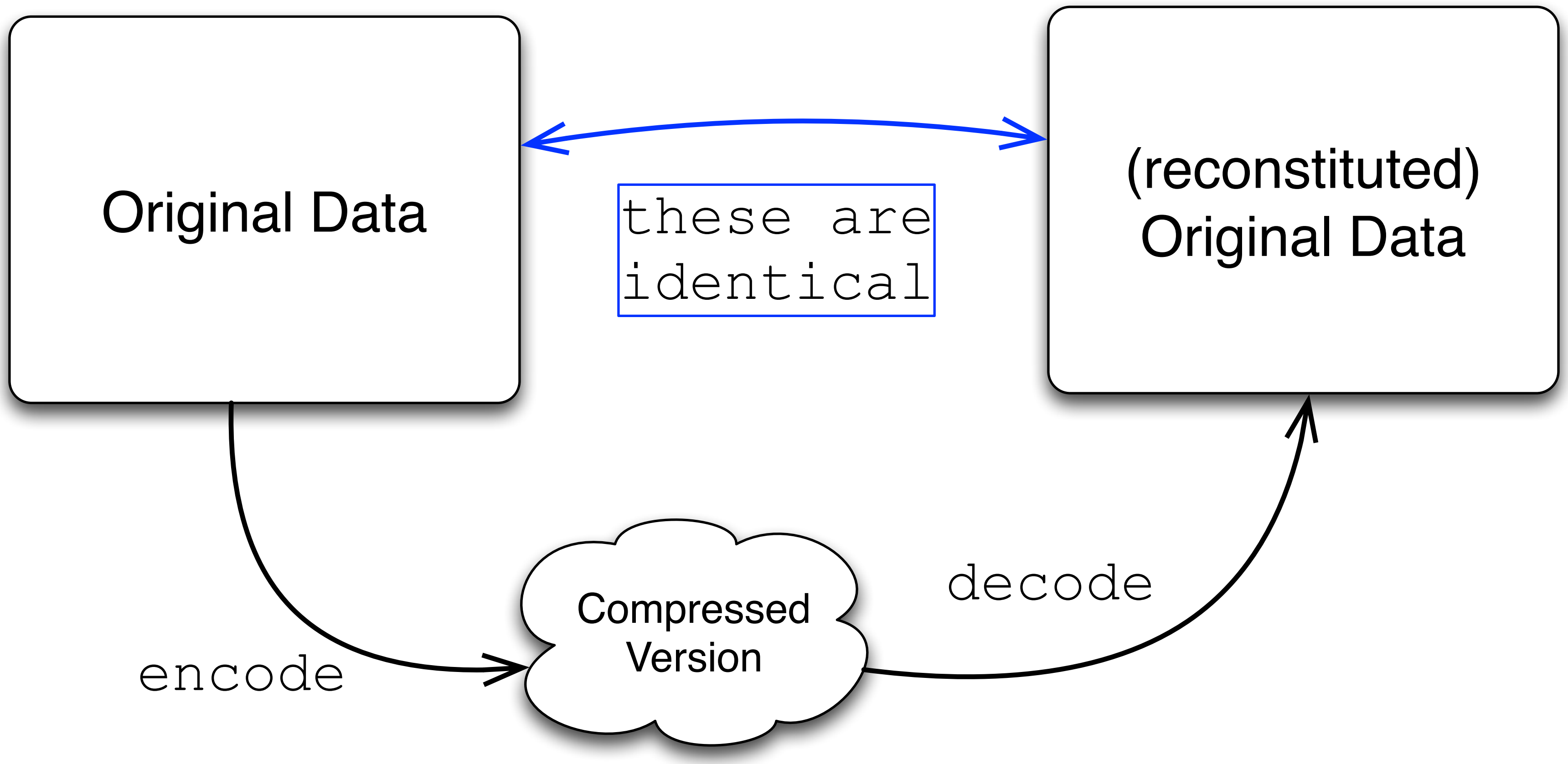
A naïve way of encoding the letters of the alphabet is to put all the data on the leaf layer. Five bits gives you 32 possible leaf nodes; some of which are not used. This means each symbol requires the same number of bits to identify it.



What if we could use a variable number of bits to represent different symbols? E.g. 3 bits for 'e' since it is so common, but 7 bits for 'z' which is rare? This would mean the entire collection of data requires less space. This is the goal with data compression.



The goal with lossless compression is to find an alternate encoding so we can compress and reconstruct the data without losing any information.



Step One: Create a Frequency Table

To do this, scan a corpus of text (or other symbolic data) and create a map of characters with their relative frequency.

corpus

THE VARIABLE MAN

BY PHILIP K. DICK

ILLUSTRATED BY EBEL

He fixed things--clocks, refrigerators, vidsenders and destinies. But he had no business in the future, where the calculators could not handle him. He was Earth's only hope--and its sure failure!

Security Commissioner Reinhart rapidly climbed the front steps and entered the Council building. Council guards stepped quickly aside and he entered the familiar place of great whirring machines. His thin face rapt, eyes alight with emotion, Reinhart gazed intently up at the central SRB computer, studying its reading.

"Straight gain for the last quarter," observed Kaplan, the lab organizer. He grinned proudly, as if personally responsible. "Not bad, Commissioner."

(and so on)

symbol frequency

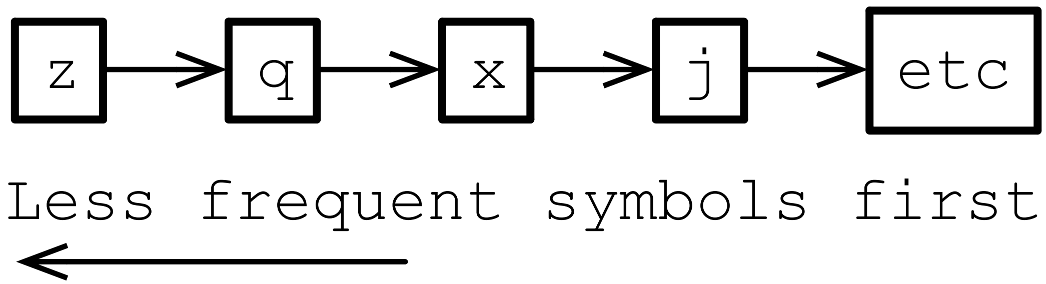
a: 81
b: 14
c: 27
d: 42
…
y: 3
z: 2
#: 1
-: 1

Step Two: Create a Priority Queue

The frequency table gives a mapping of symbols to their relative frequencies. Use this data to create a priority queue where smaller frequencies have higher priority.

<u>symbol frequency</u>

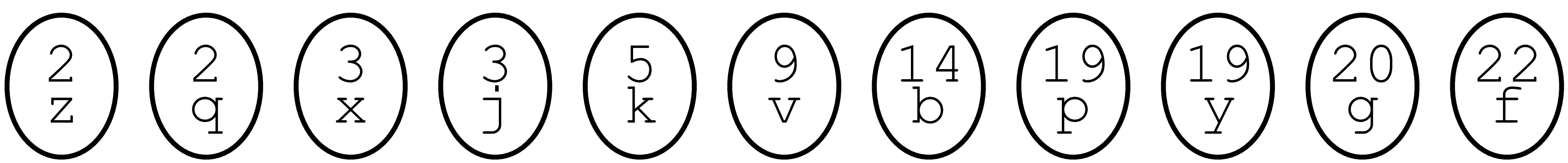
a: 81
b: 14
c: 27
d: 42
…
y: 3
z: 2



Step Three: Greedily Build A Huffman Tree

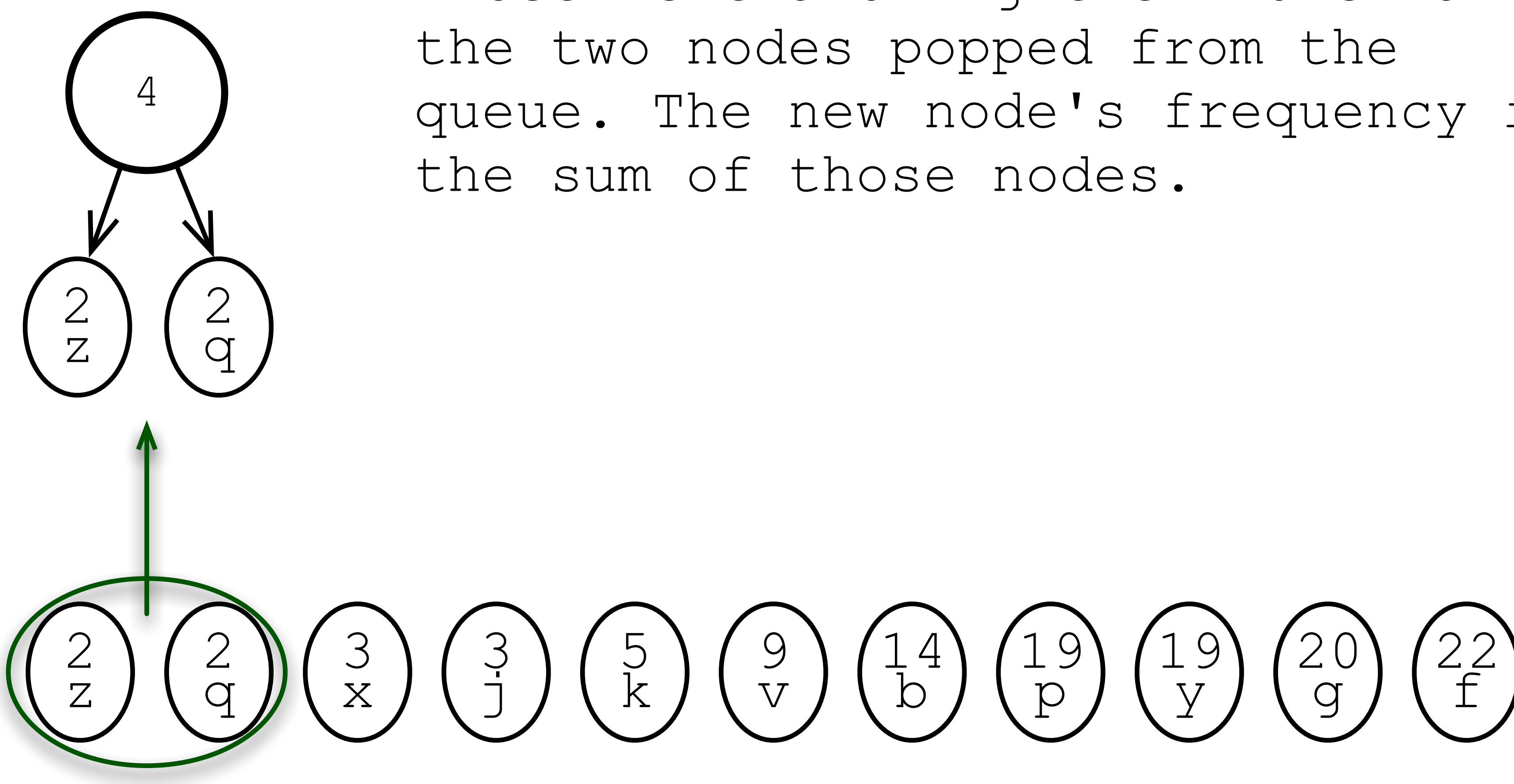
Use the priority queue to create a tree. The tree's leaves hold symbol data (e.g. 'z' or '#' or the space character). Internal nodes hold references to children, as well as the sum of the child node frequencies.

We'll pop two items from the priority queue, combine them to form an internal node, and put that node back in the priority queue.

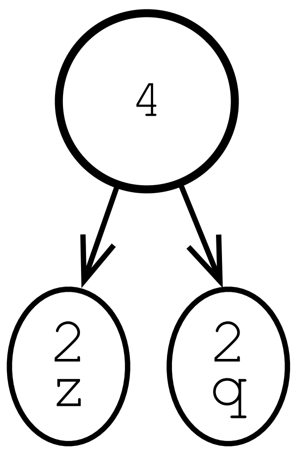


Step Three: Greedily Build A Huffman Tree

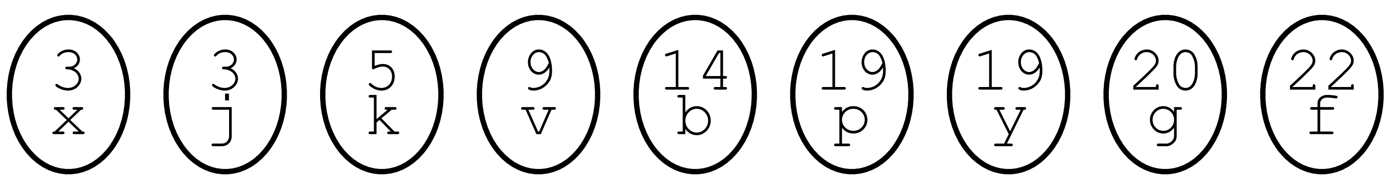
Make an internal binary tree node whose left and right children are the two nodes popped from the queue. The new node's frequency is the sum of those nodes.



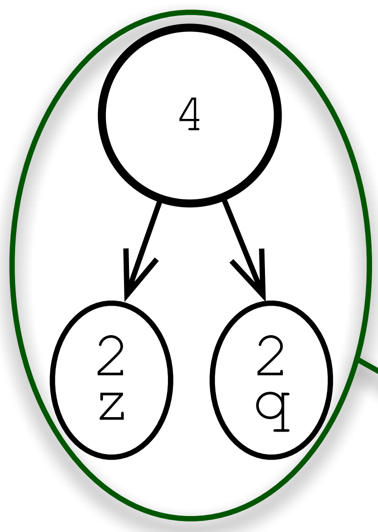
Step Three: Greedily Build A Huffman Tree



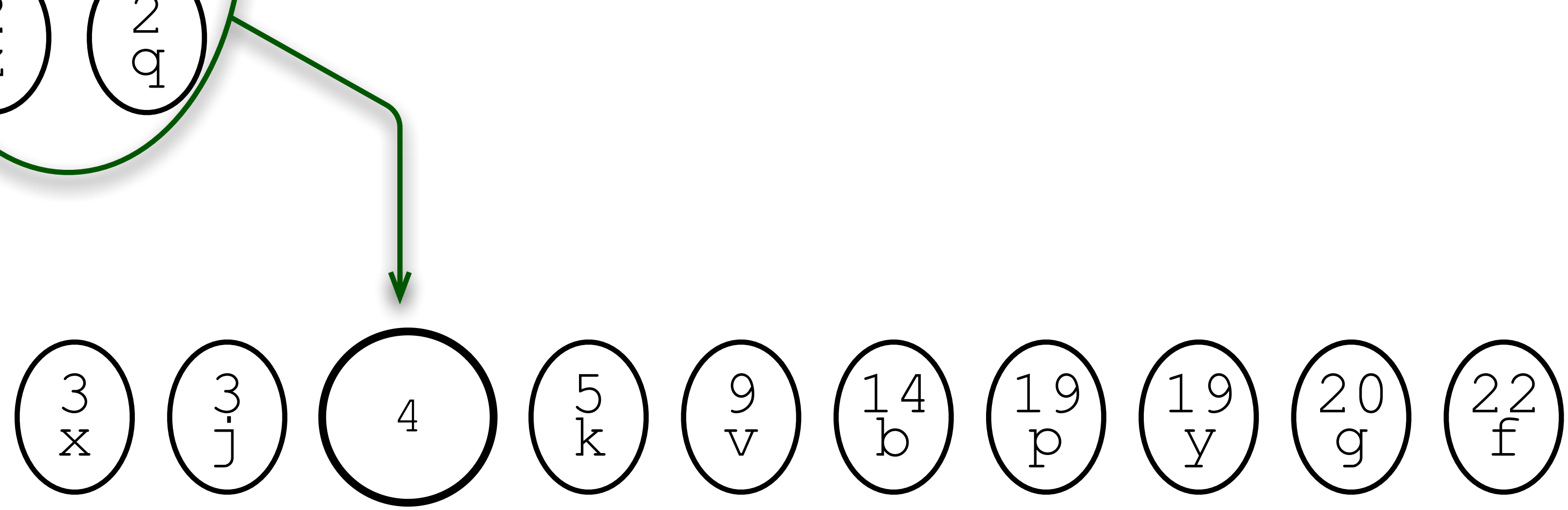
Note that the two nodes we popped are no longer in the priority queue.



Step Three: Greedily Build A Huffman Tree



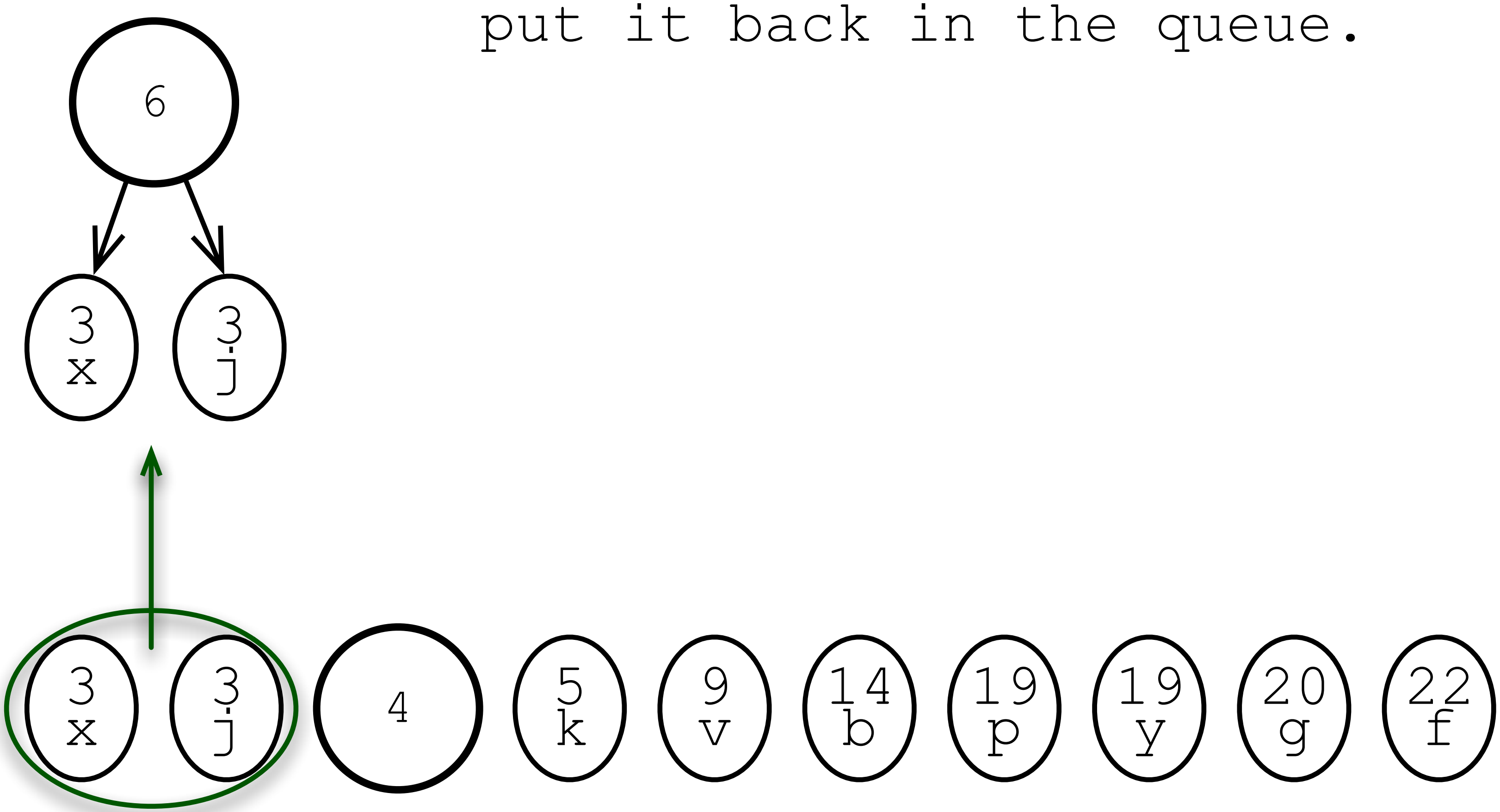
Now insert the newly created node *back* into the priority queue, using its frequency (4) to put it in the right spot.



Note: the children are still there, but they aren't drawn.

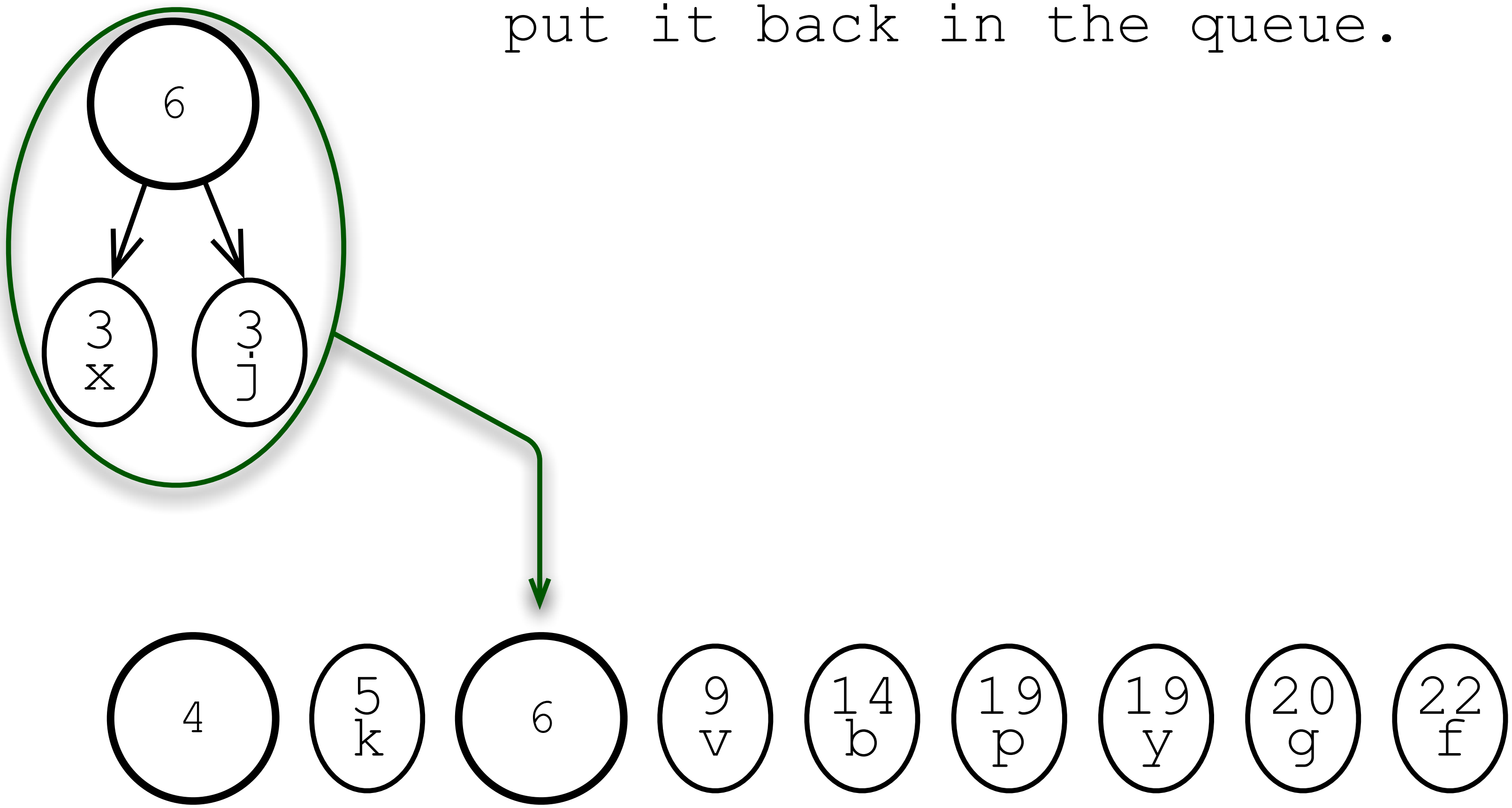
Step Three: Greedily Build A Huffman Tree

Pop two more nodes, make another internal node, and put it back in the queue.

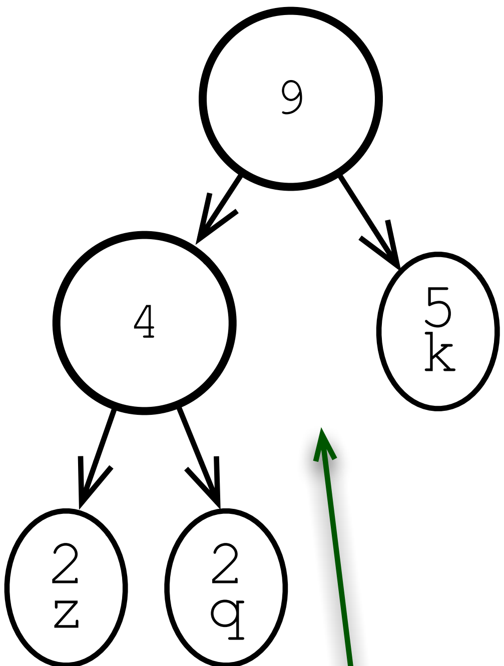


Step Three: Greedily Build A Huffman Tree

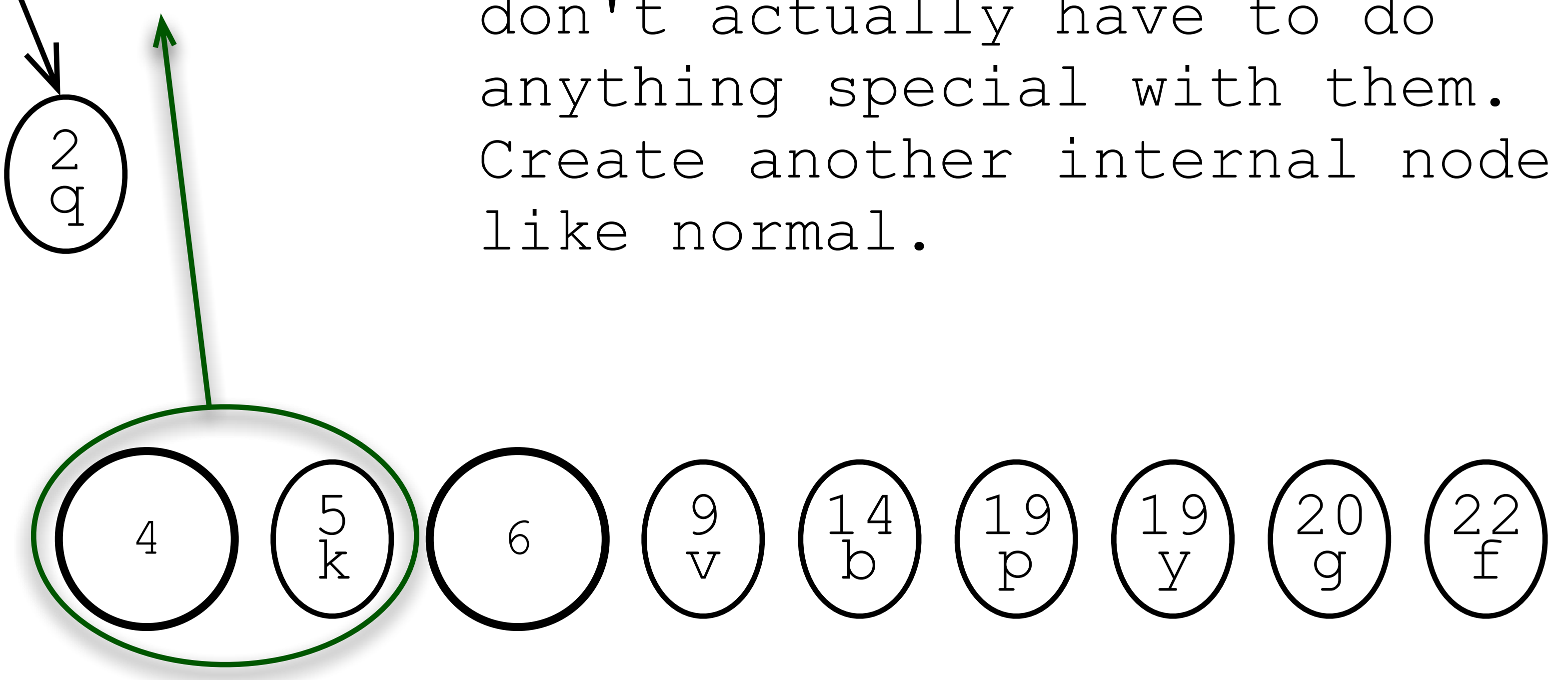
Pop two more nodes, make another internal node, and put it back in the queue.



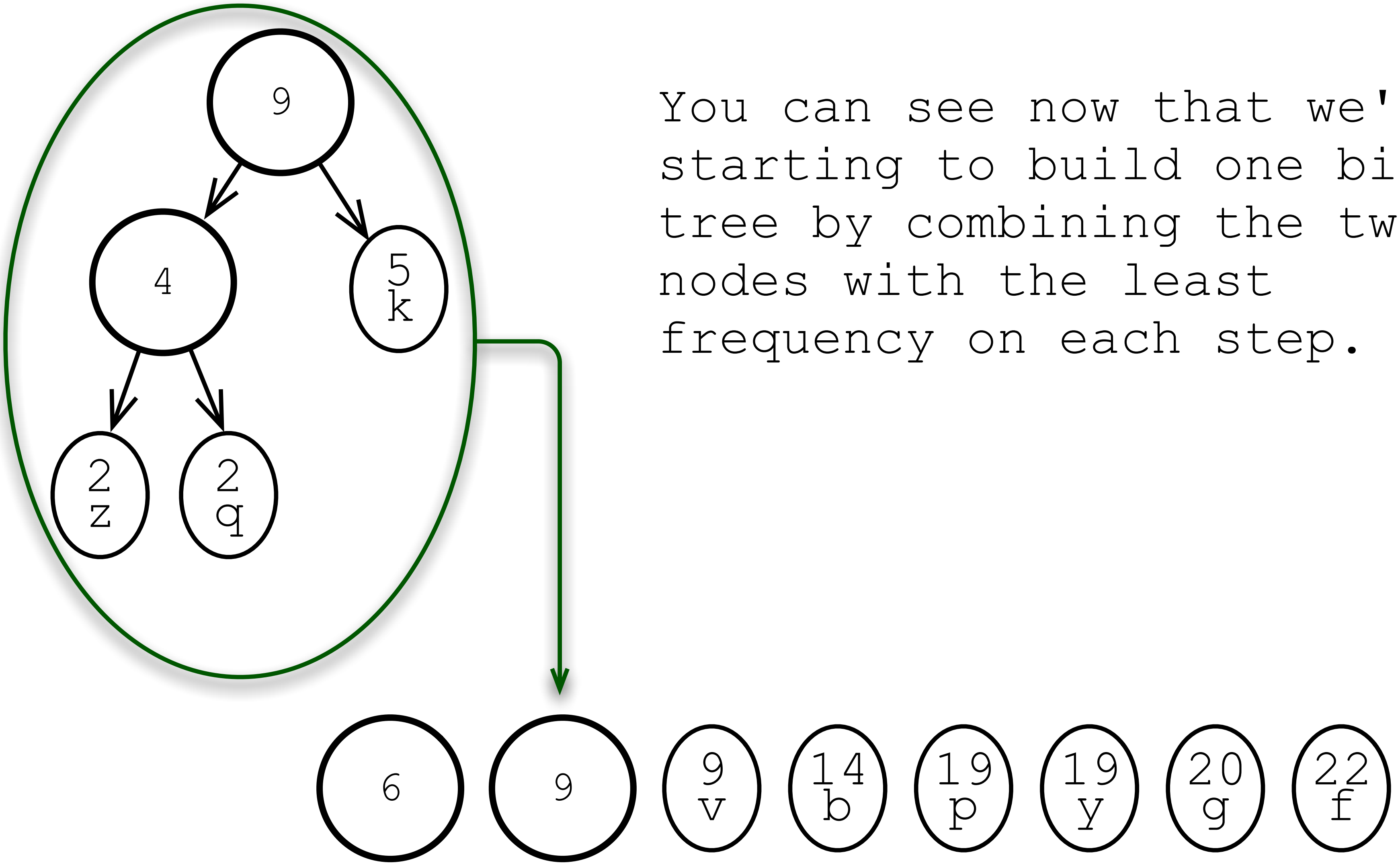
Step Three: Greedily Build A Huffman Tree



Now something interesting happens. One of the two nodes we pop is an internal node. Remember that node with frequency=4 has children. We don't actually have to do anything special with them. Create another internal node like normal.



Step Three: Greedily Build A Huffman Tree

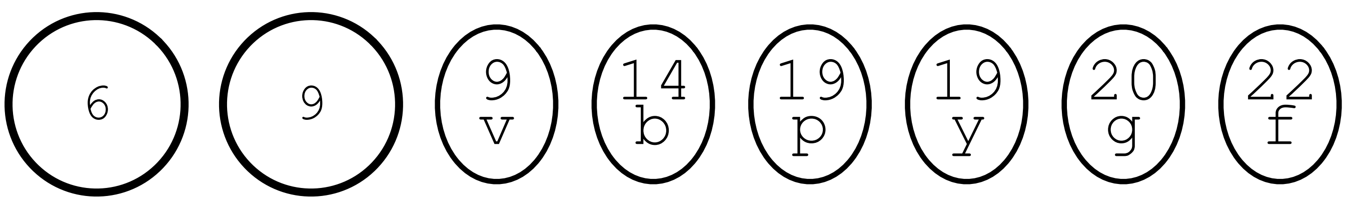


You can see now that we're starting to build one big tree by combining the two nodes with the least frequency on each step.

Step Three: Greedily Build A Huffman Tree

This is a 'greedy' algorithm because it chooses a locally optimal path--combining the two least frequent nodes at every step.

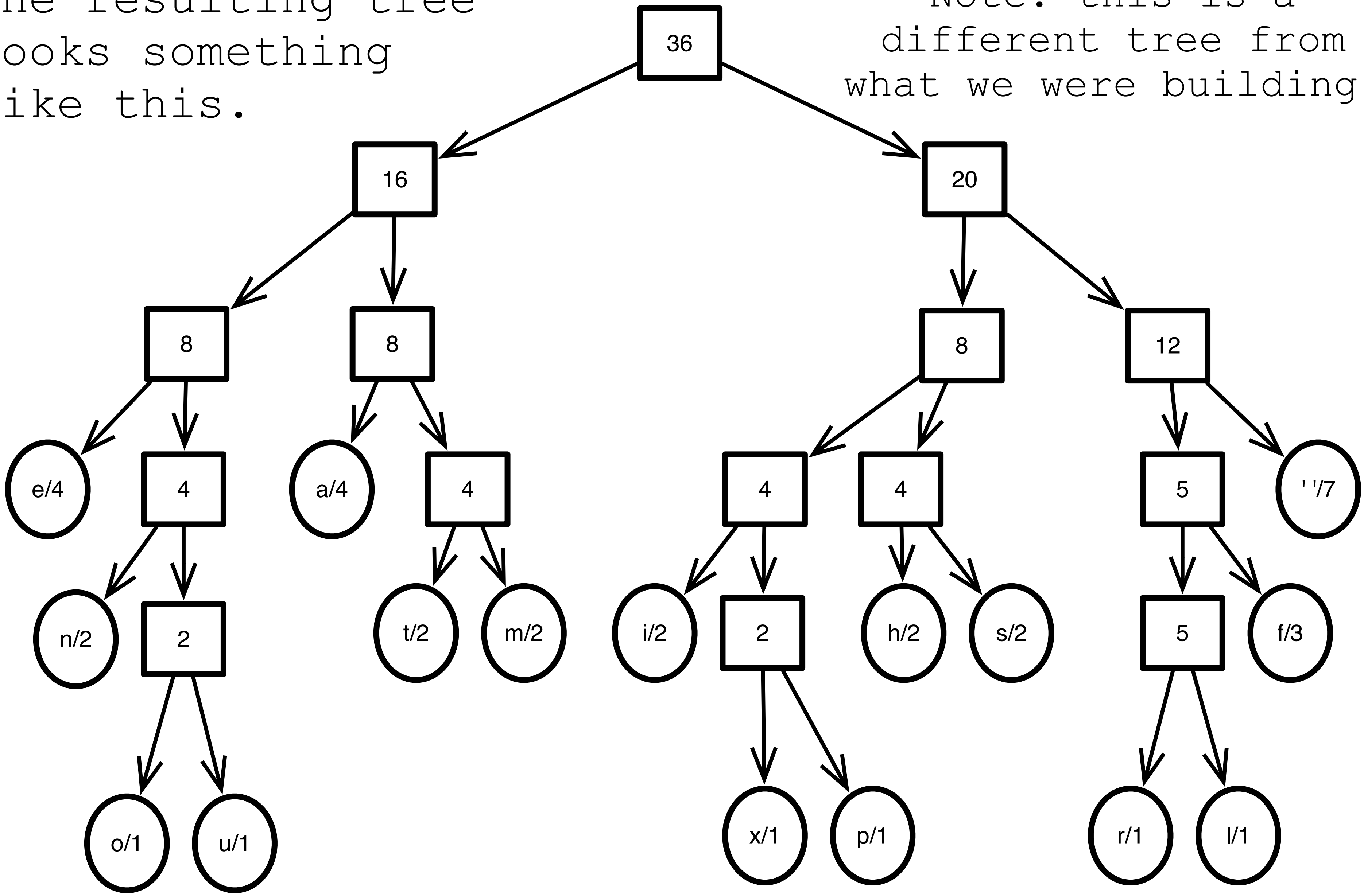
There are bazillions of algorithms that we call 'greedy'. Greedy algorithms only look one step into the future using local information, rather than looking for a global optimum using *all* (non-local) information.



Step Three: Greedily Build A Huffman Tree

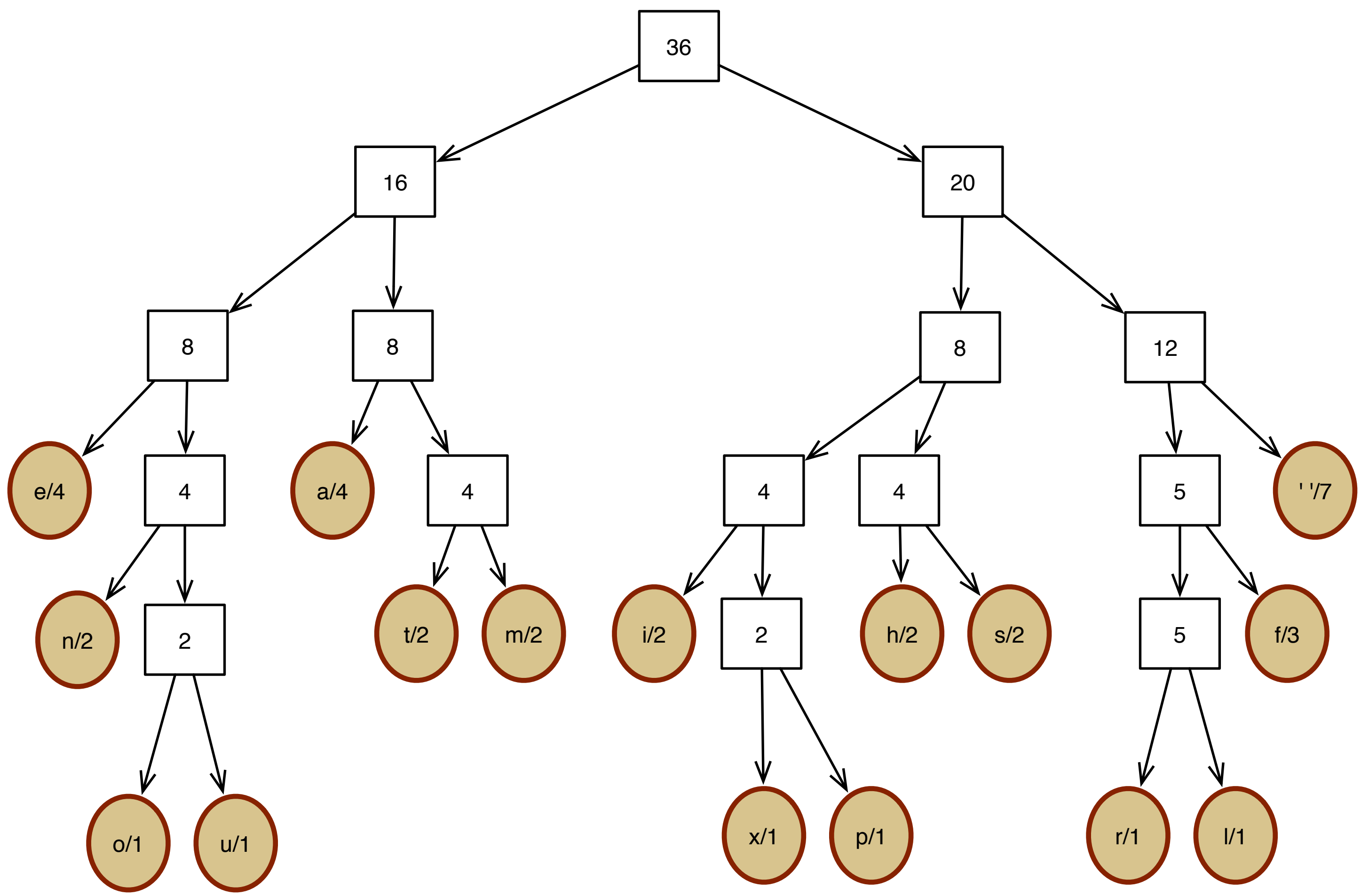
The resulting tree looks something like this.

Note: this is a different tree from what we were building.



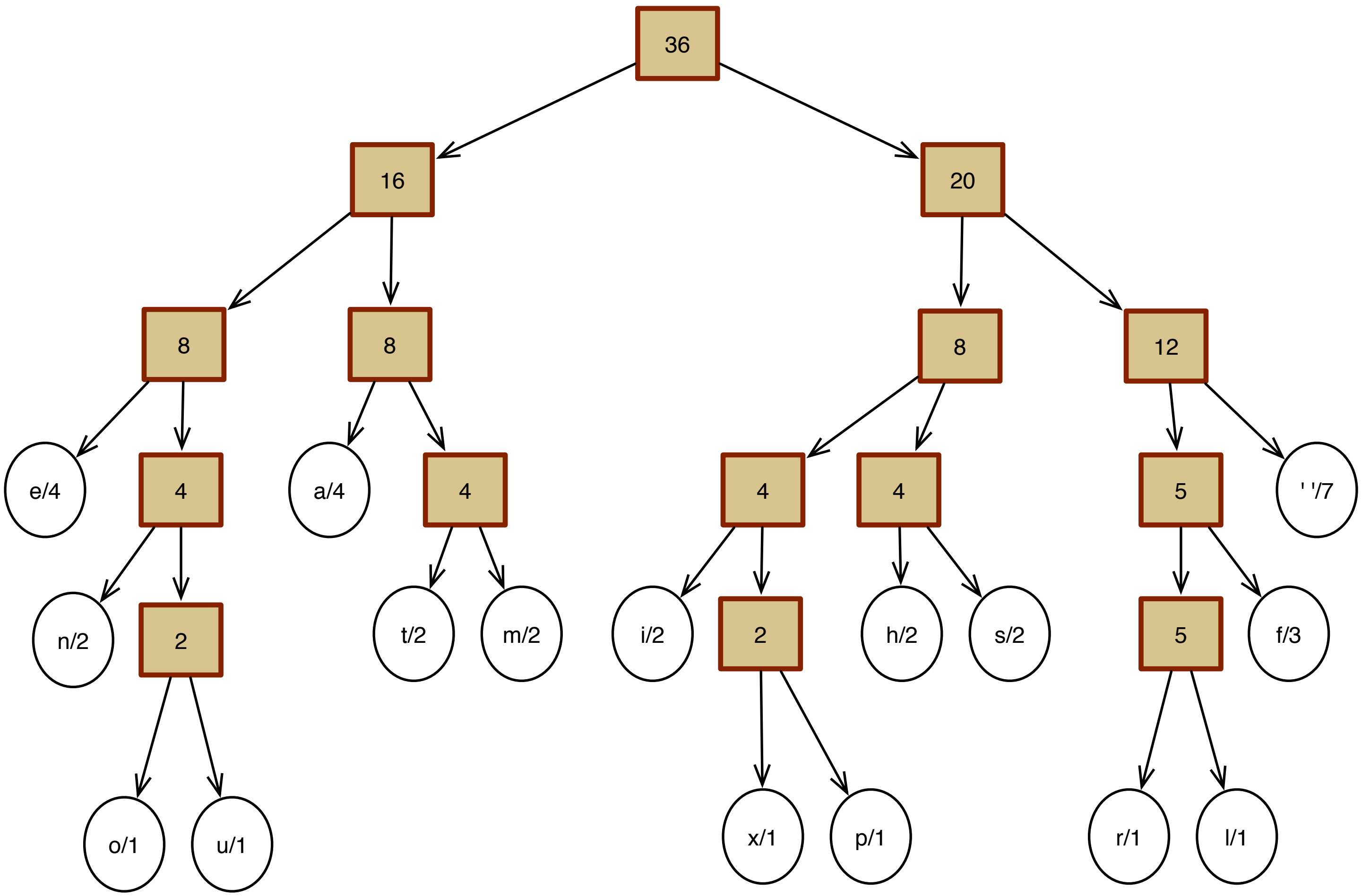
Properties of a Huffman Tree

1. Leaf nodes all have symbol data.



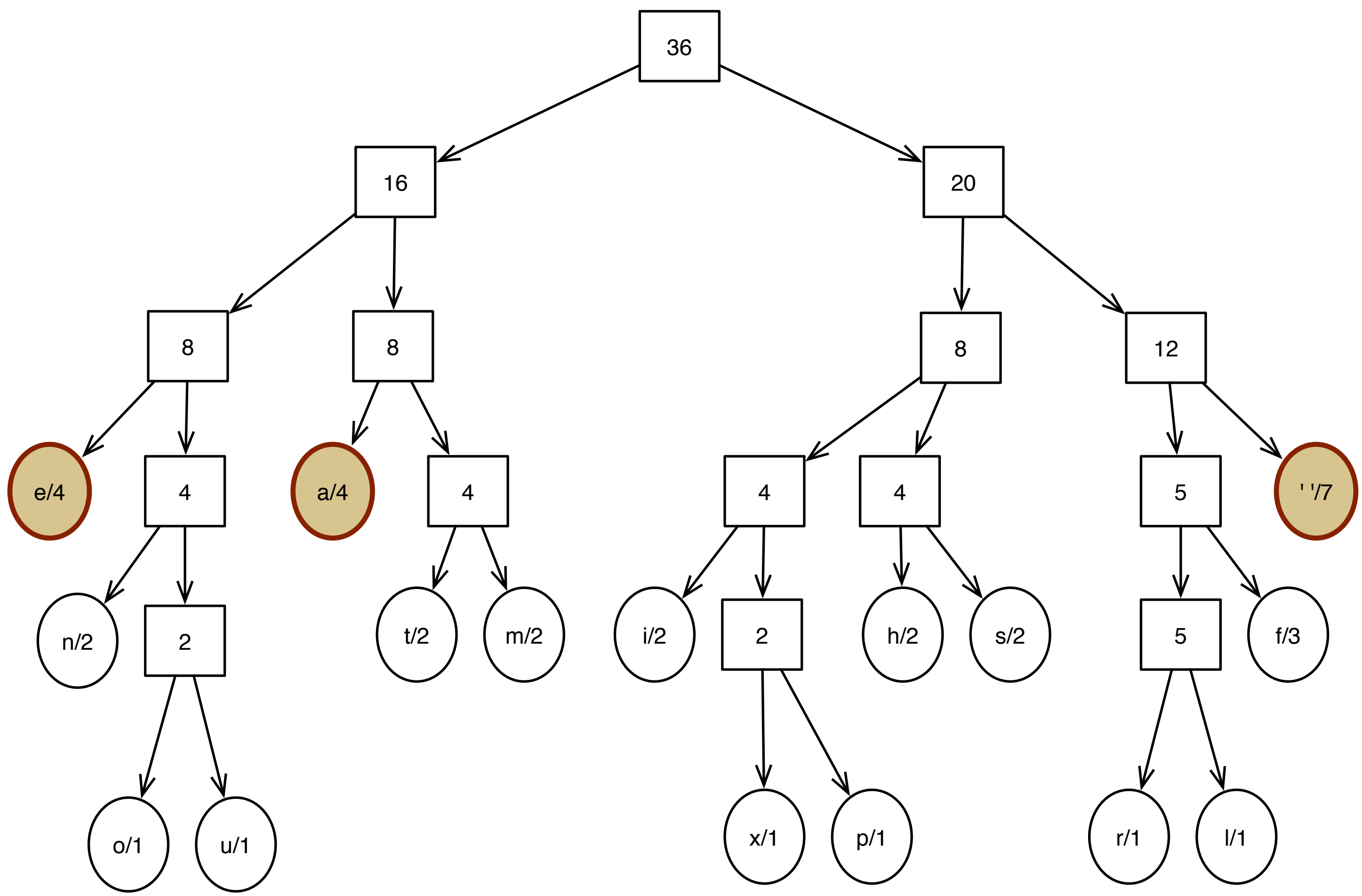
Properties of a Huffman Tree

2. Internal node frequencies are the sum of their child node frequencies.



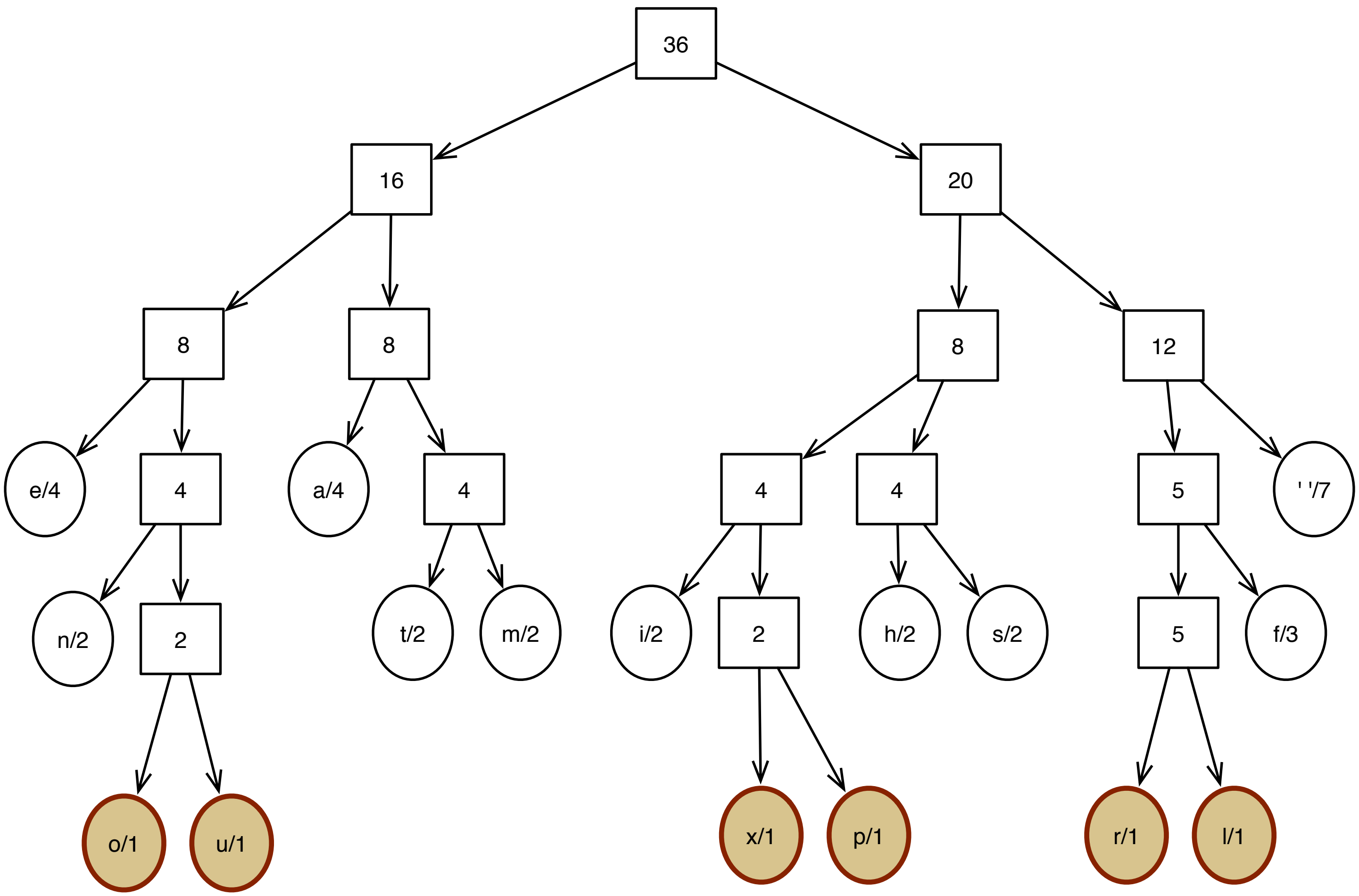
Properties of a Huffman Tree

3. The most frequent symbols have the shortest path to the root.



Properties of a Huffman Tree

4. The least frequent symbols have the longest path to the root.



Using A Huffman Tree to <u>Encode</u> Data

Say we want to encode the string

"this is an example of a huffman tree"

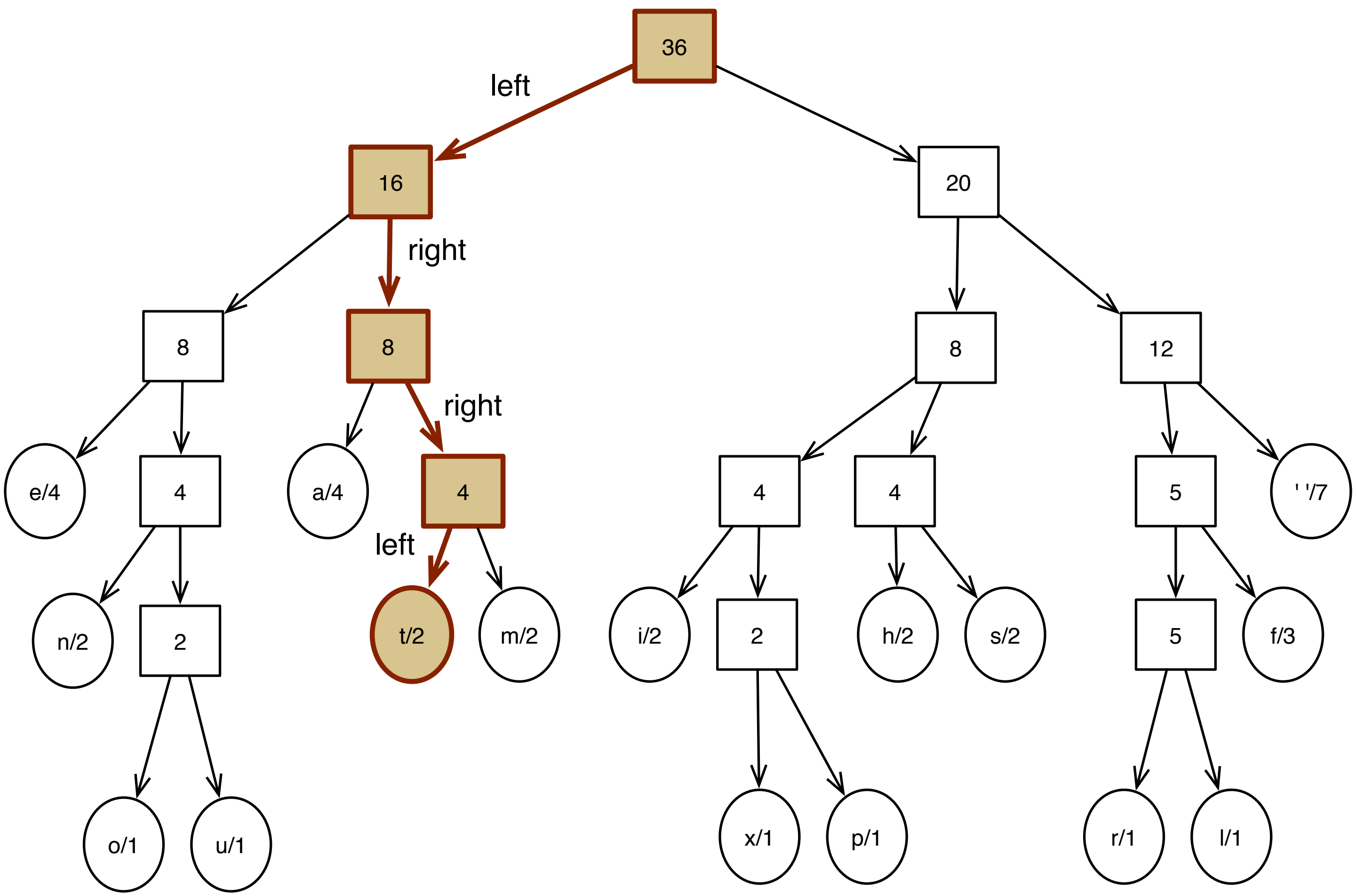
using our huffman tree. For each symbol we will output a bit string representing the path from root to that symbol's leaf node.

Without compression, each symbol requires the same number of bits. Using ASCII encoding, we need 8 bits per symbol.

With our Huffman encoding (using this particular tree) a symbol requires anywhere between three and five bits. This is because the leaf nodes appear at levels three, four, and five.

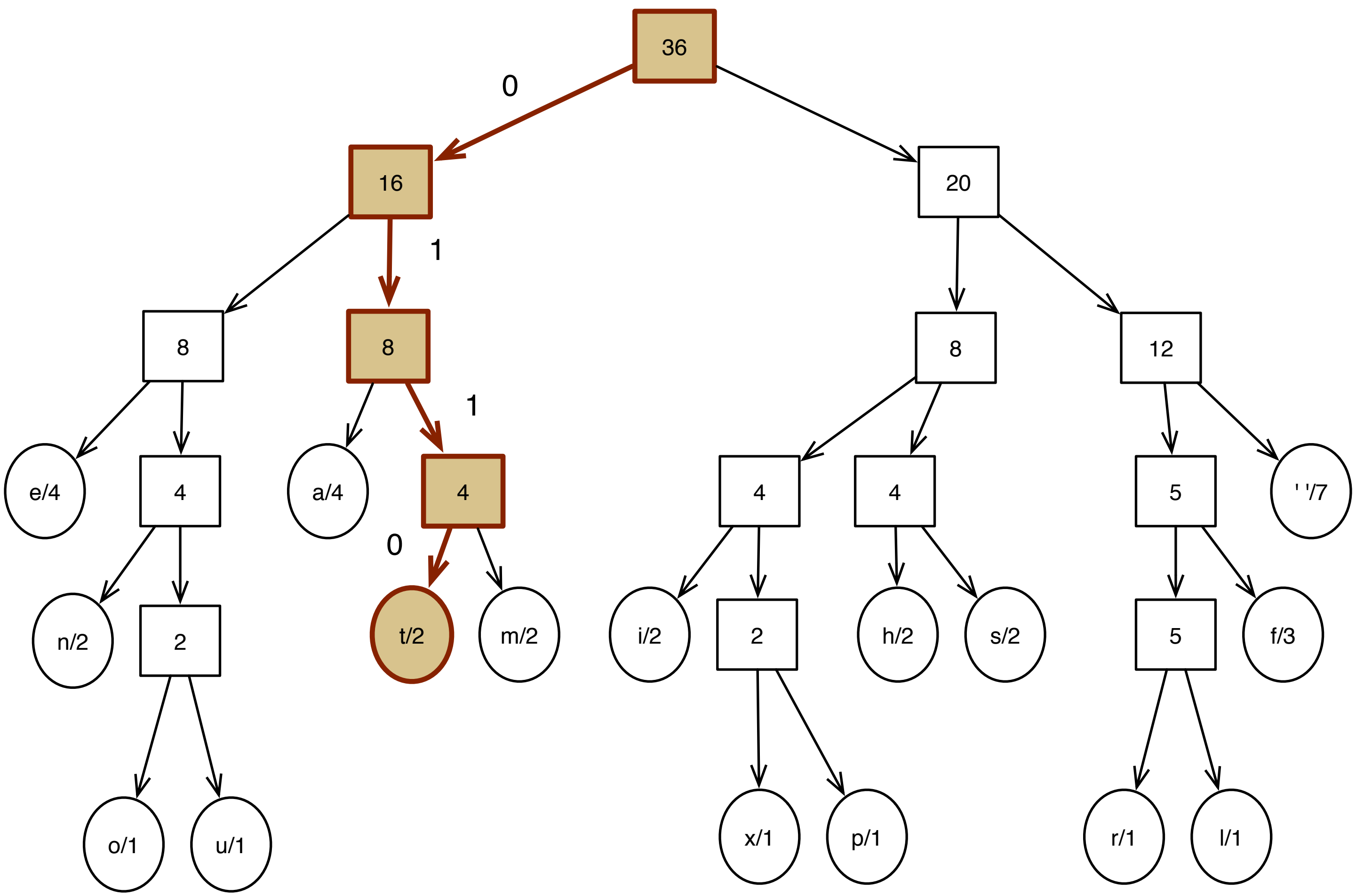
Using A Huffman Tree to Encode Data

Encoding for the letter 't'. Starting from the root, we must traverse child links in the order: left, right, right, left.



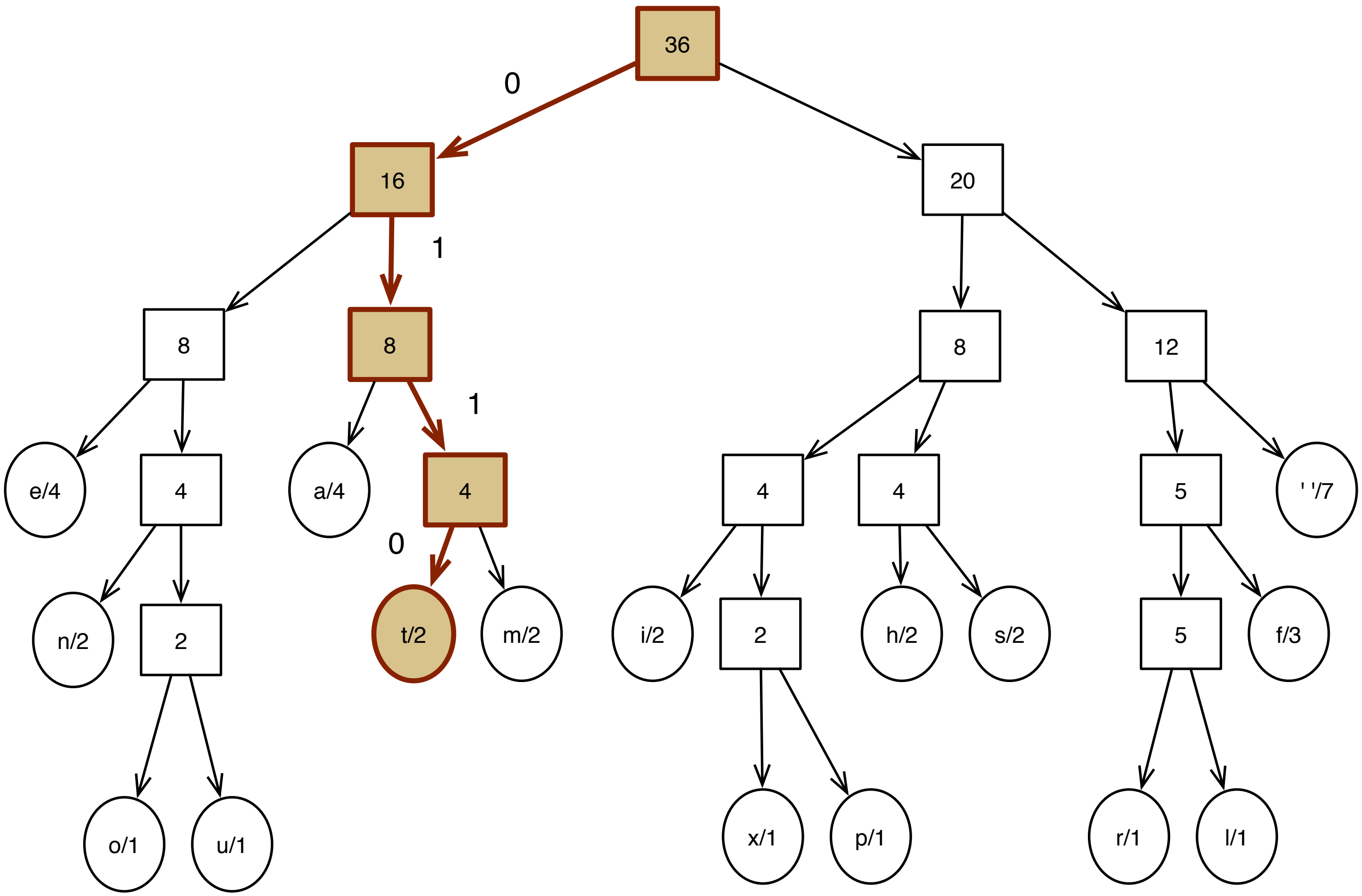
Using A Huffman Tree to Encode Data

If we treat a left turn to mean 'output 0' and right turn to mean 'output 1', we can use this traversal to produce the bit string: **0110**.



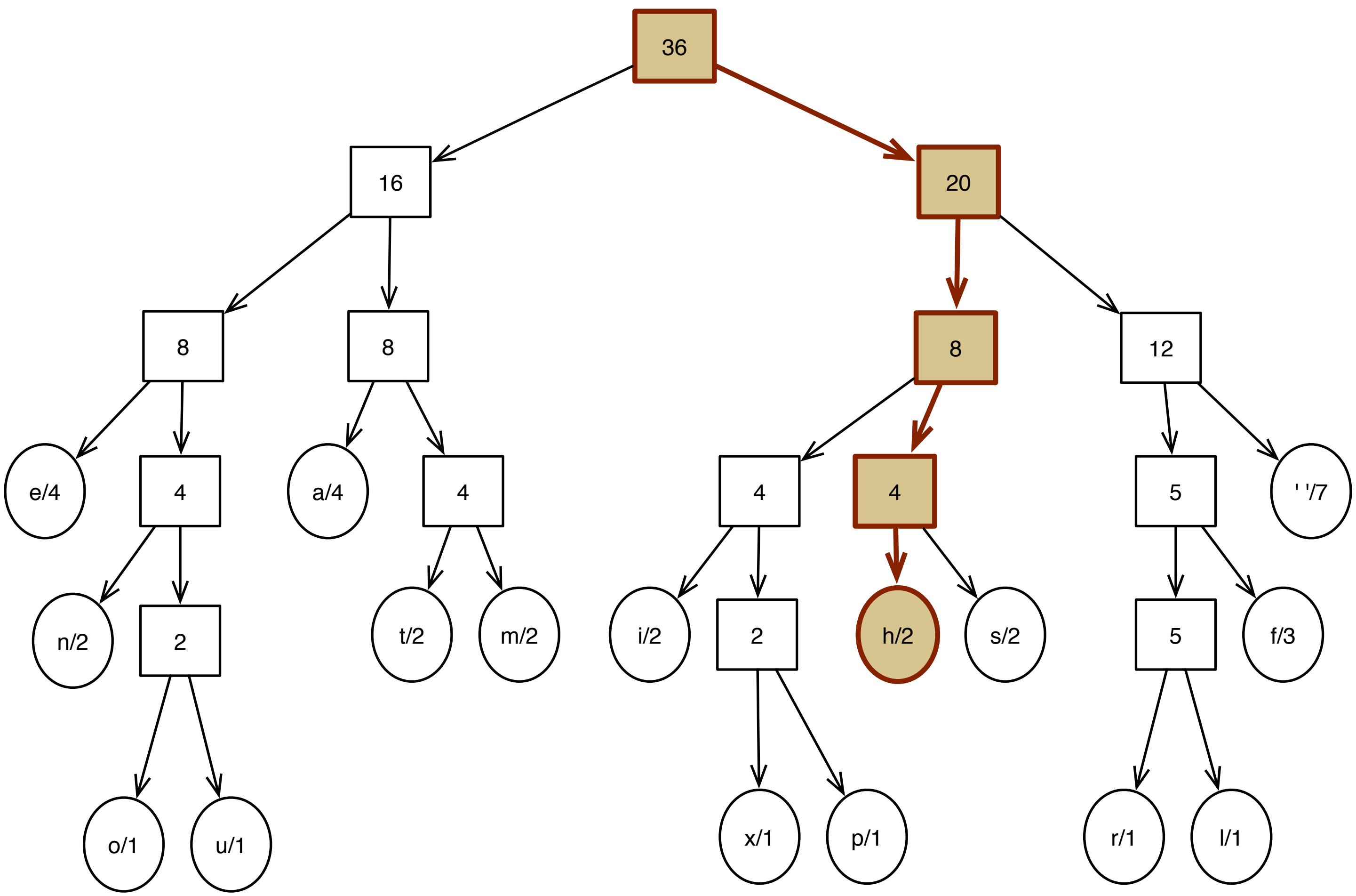
Using A Huffman Tree to <u>Encode</u> Data

_t__
0110



Using A Huffman Tree to **Encode** Data

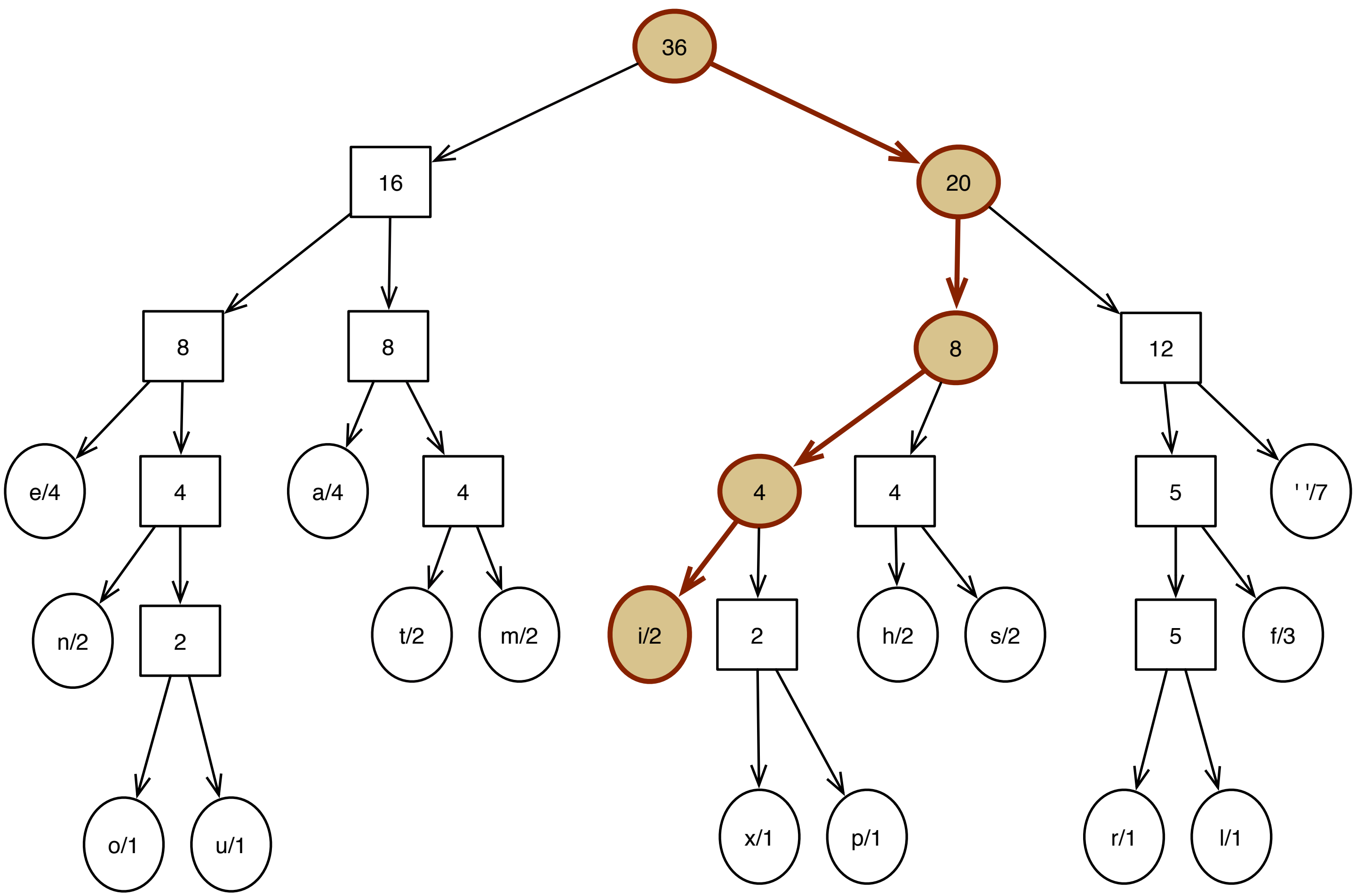
```
 t    h
0110 1010
```



Using A Huffman Tree to Encode Data

t h i

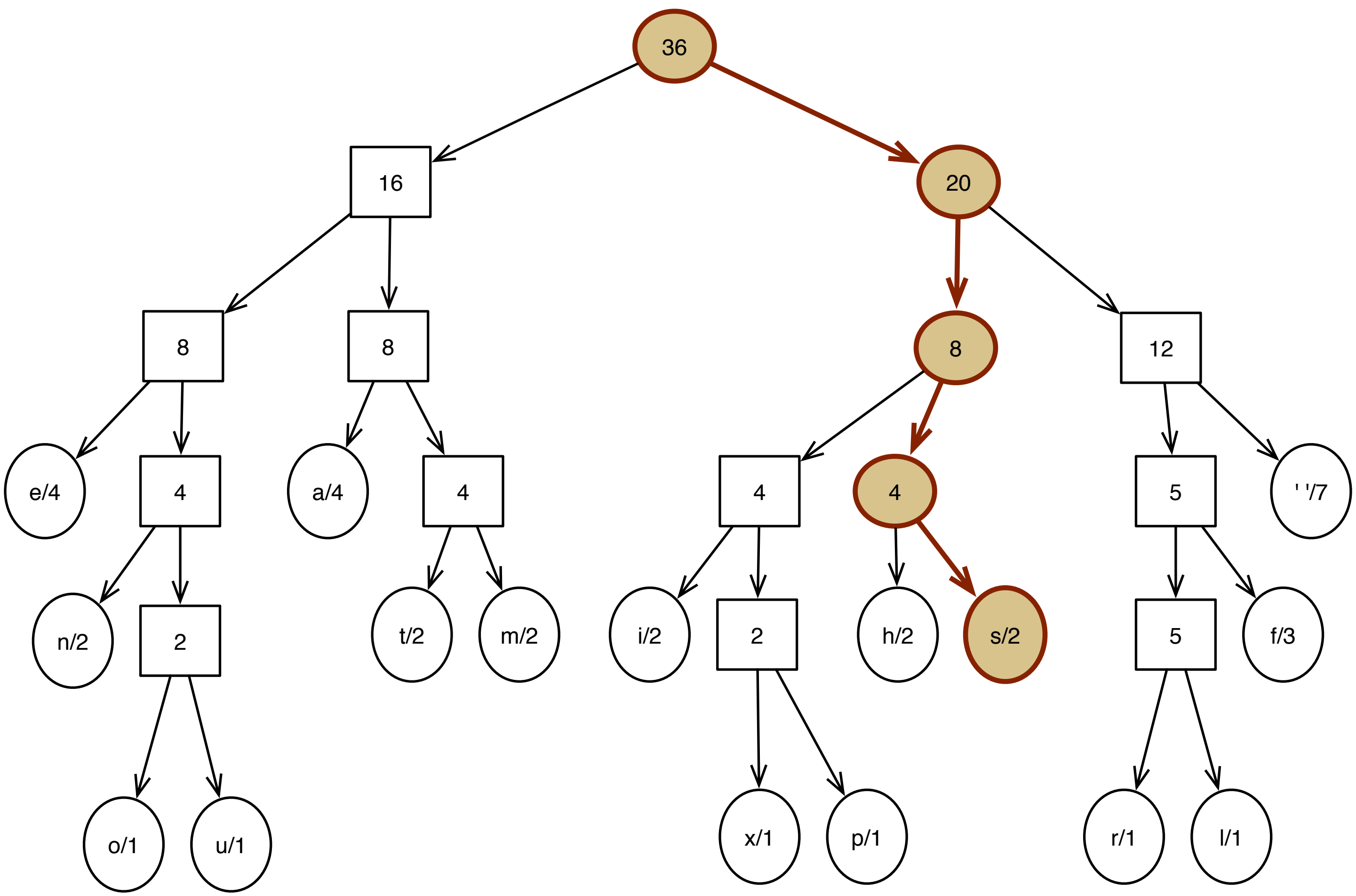
0110 1010 1000



Using A Huffman Tree to Encode Data

t h i s

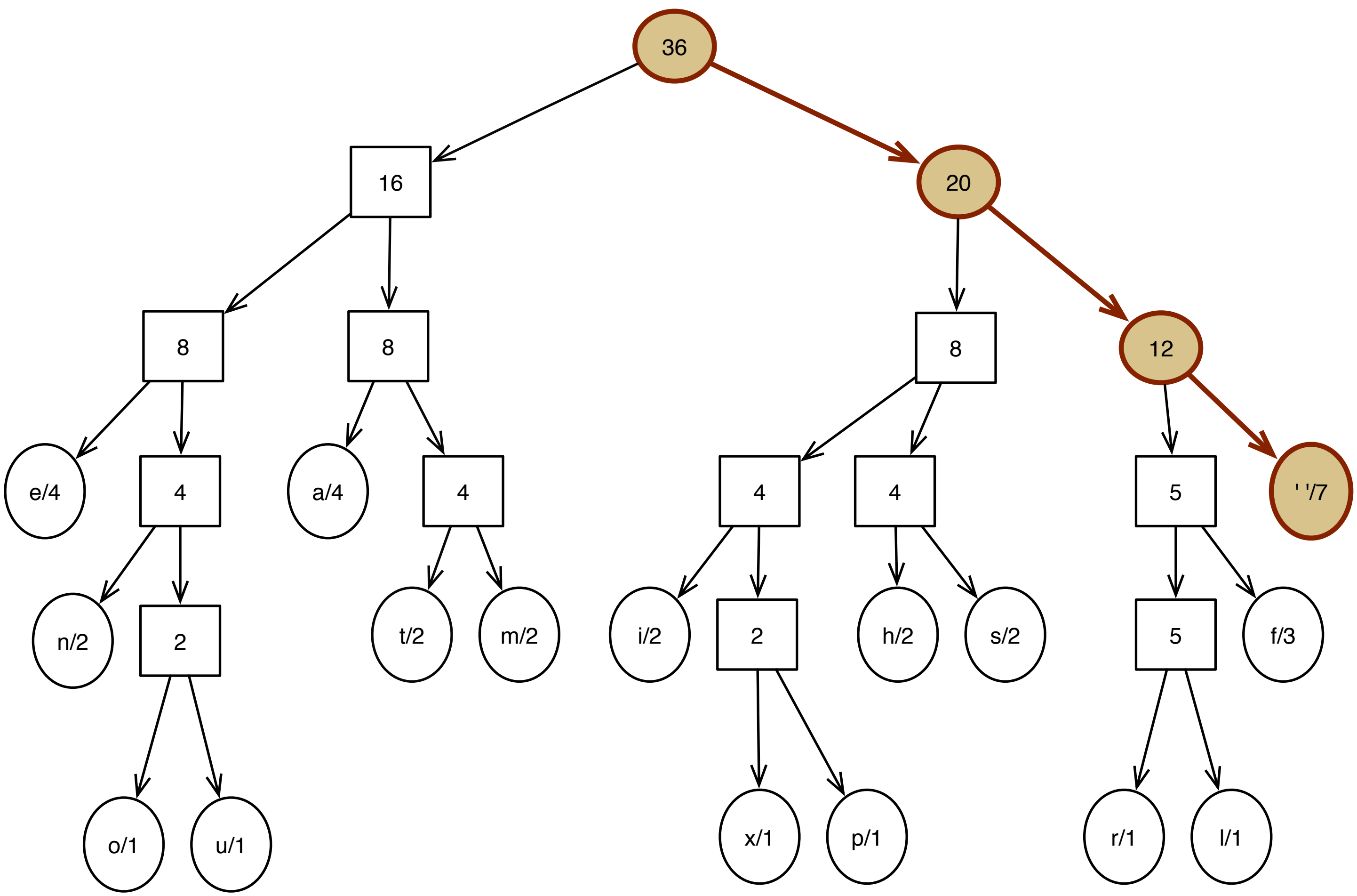
0110 1010 1000 1011



Using A Huffman Tree to <u>Encode</u> Data

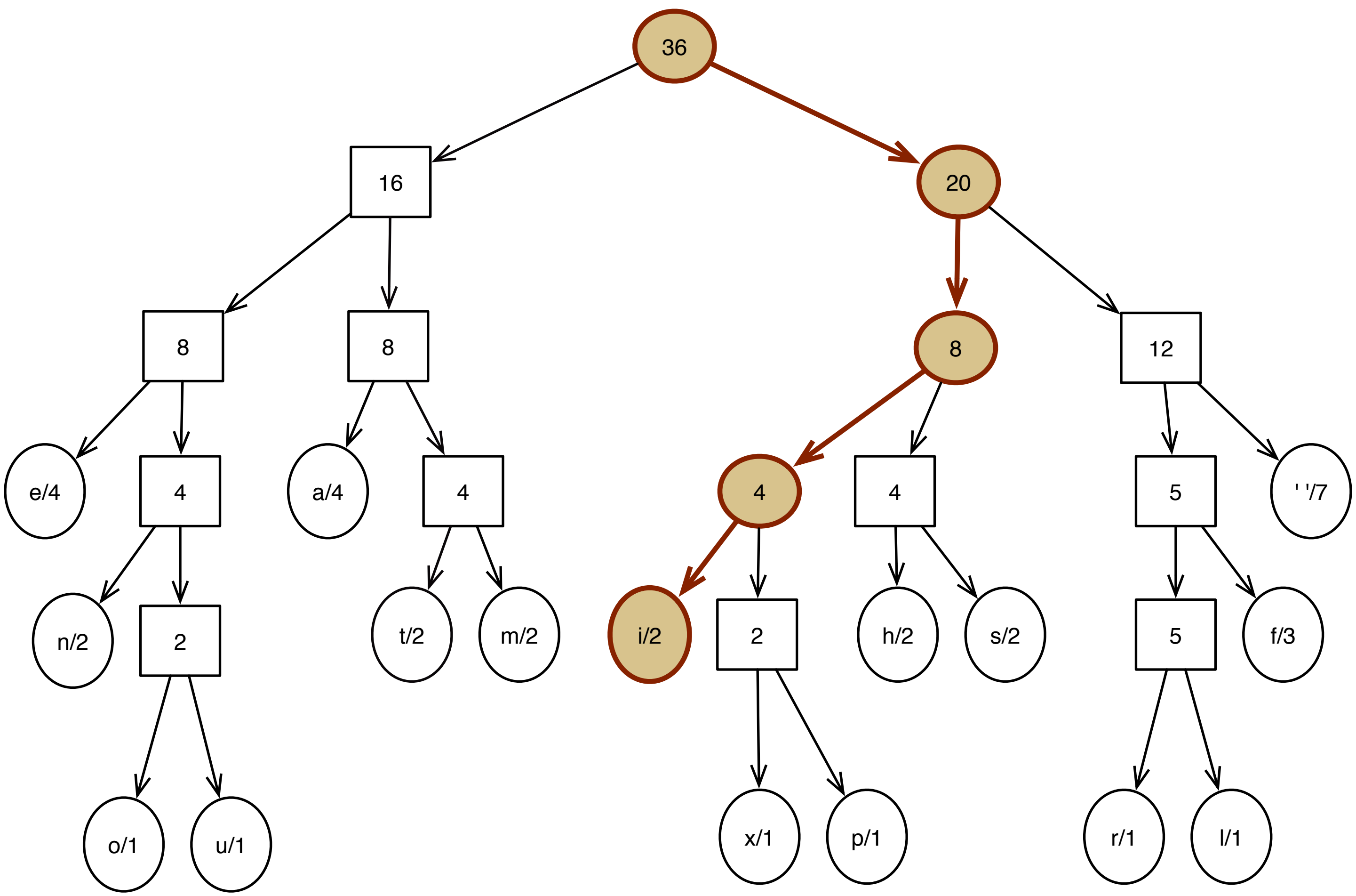
t	h	i	s	
0110	1010	1000	1011	111

(space character)



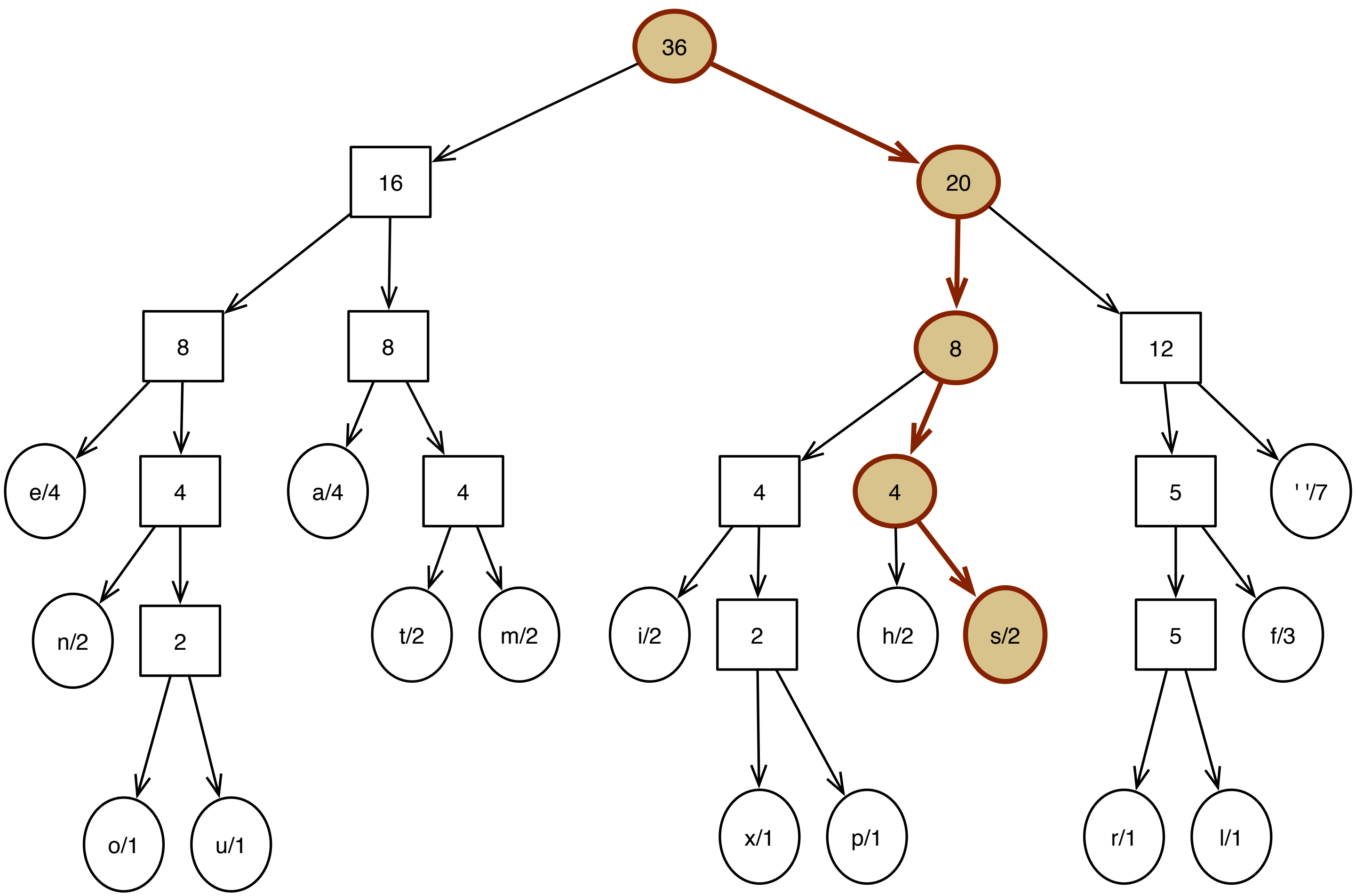
Using A Huffman Tree to Encode Data

t	h	i	s		i
0110	1010	1000	1011	111	1000



Using A Huffman Tree to <u>Encode</u> Data

_t__ _h__ _i__ _s__ _ _ _i__ _s__
0110 1010 1000 1011 111 1000 1011



Using A Huffman Tree to Encode Data

_t__ _h__ _i__ _s__ _ _ _i__ _s__
0110 1010 1000 1011 111 1000 1011

It continues like this for a while.

I've been showing you the individual bit string sequences above with spaces in between, just so you can see where we are. But the output bit string doesn't include them. So the bit string we actually have assembled so far looks like this:

011010101000101111110001011

Using A Huffman Tree to Decode Data

011010101000101111110001011

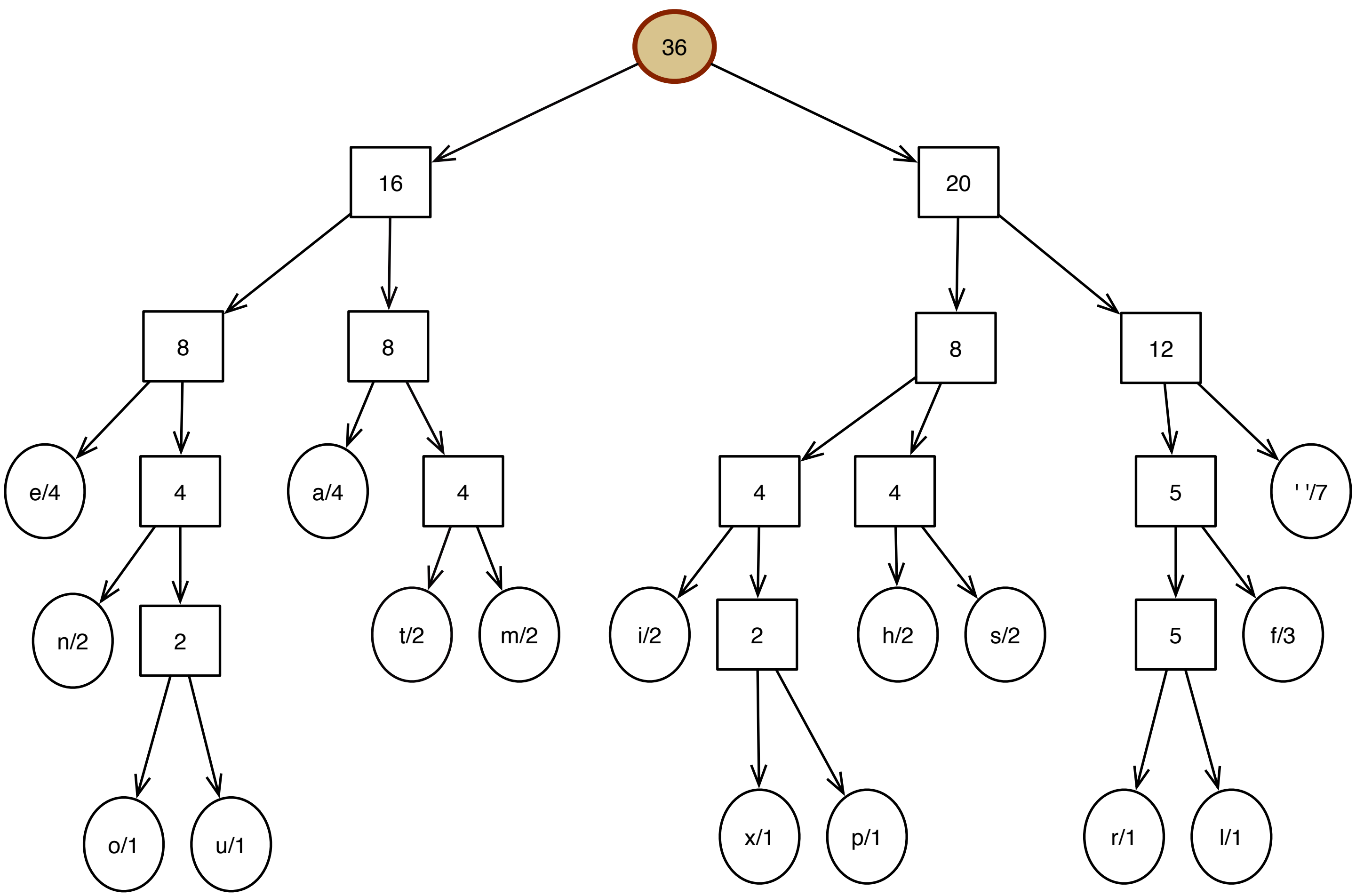
We have this bit string, without the visual benefit of spaces to separate the symbols. How's this work? We break out our Huffman Tree (the same one that was used to encode the data) and use it.

Start by placing a cursor at the root, and then reading each bit one at a time. When we see a zero we update the cursor to point to its left child; one means point it right.

When the cursor ends up on a leaf node, stop. We've reached the encoded character. Output that symbol, reset the cursor to the root and continue until we've run out of bits.

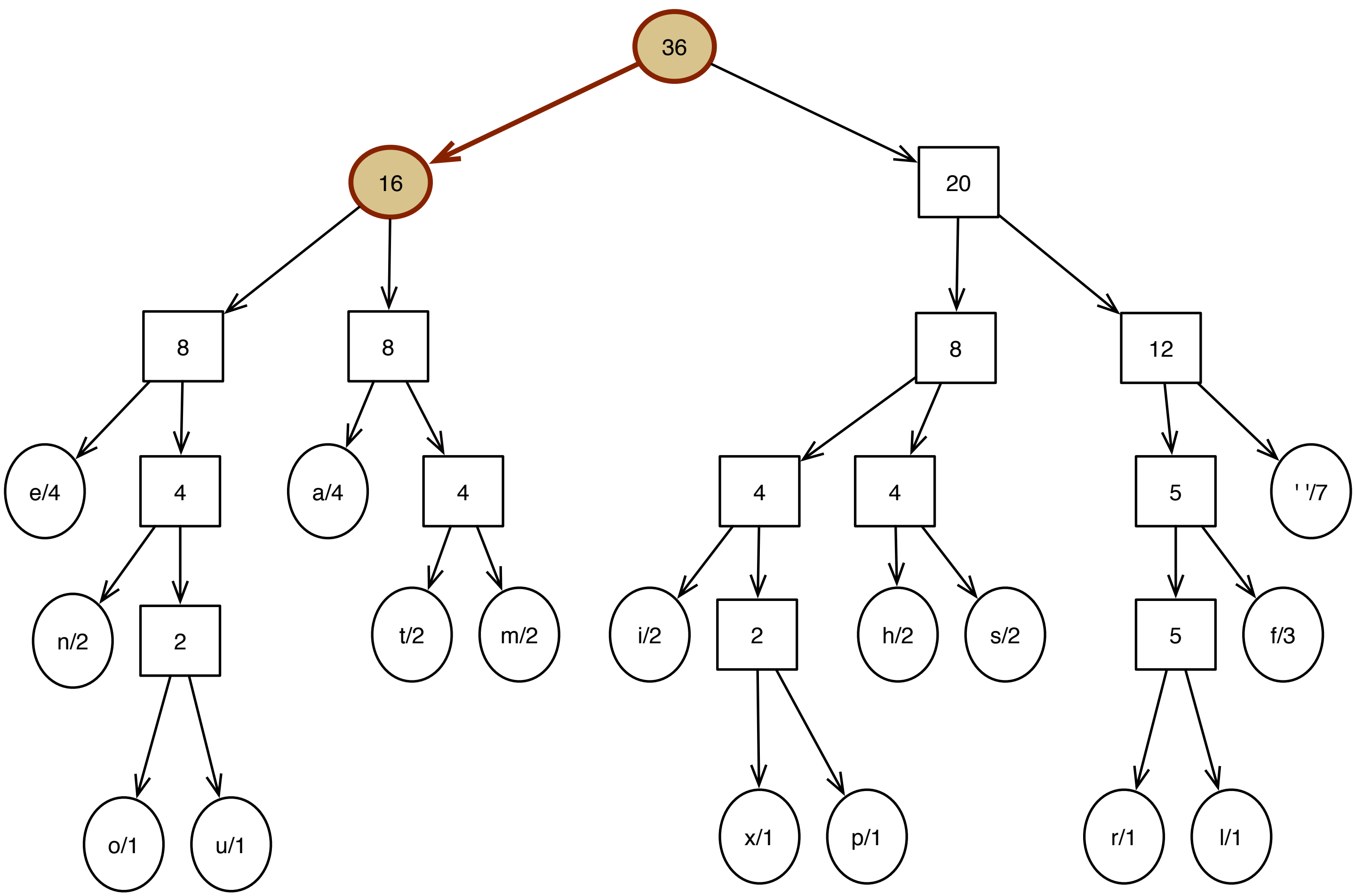
Using A Huffman Tree to <u>Decode</u> Data

011010101000101111110001011



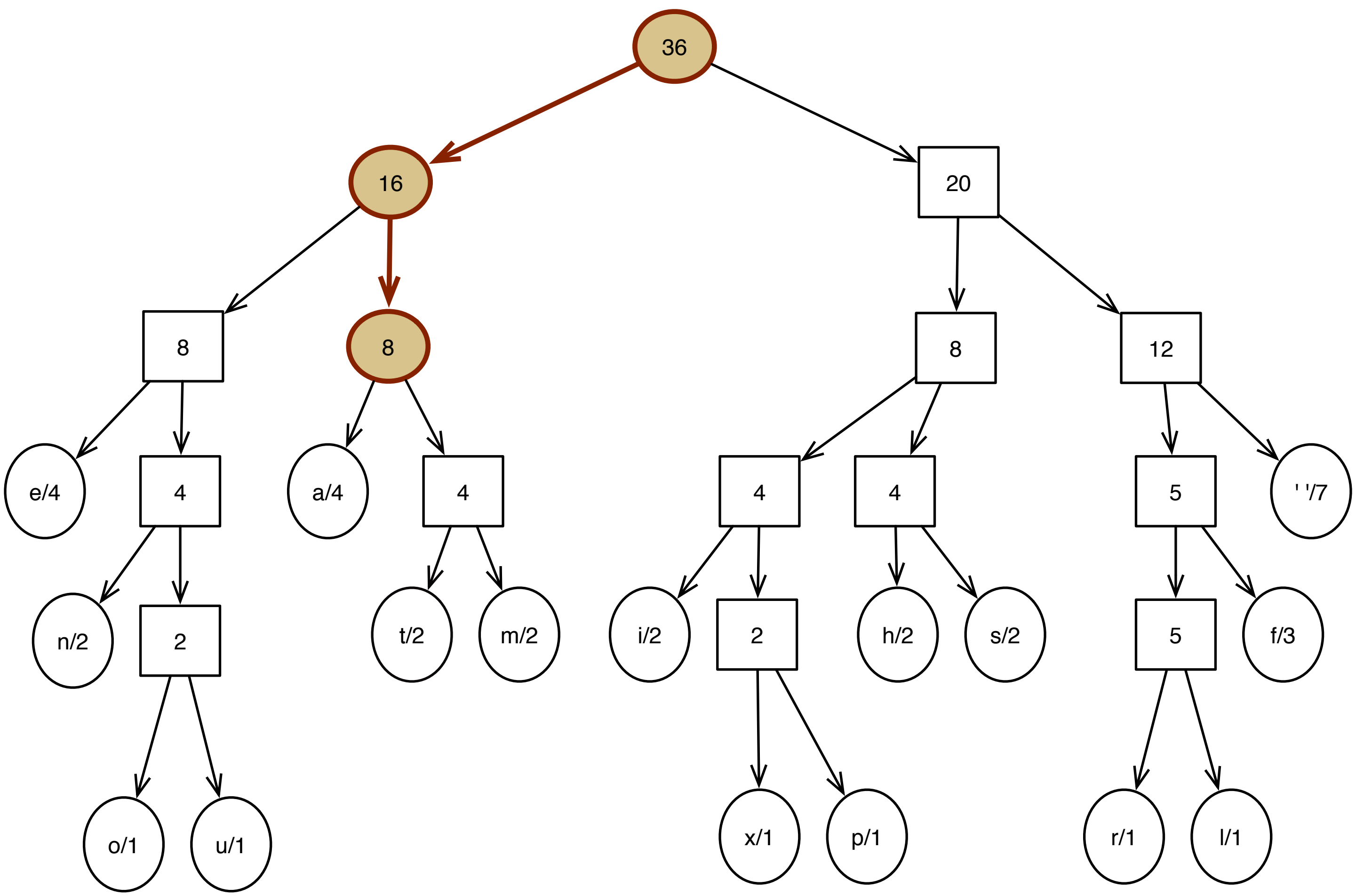
Using A Huffman Tree to <u>Decode</u> Data

011010101000101111110001011



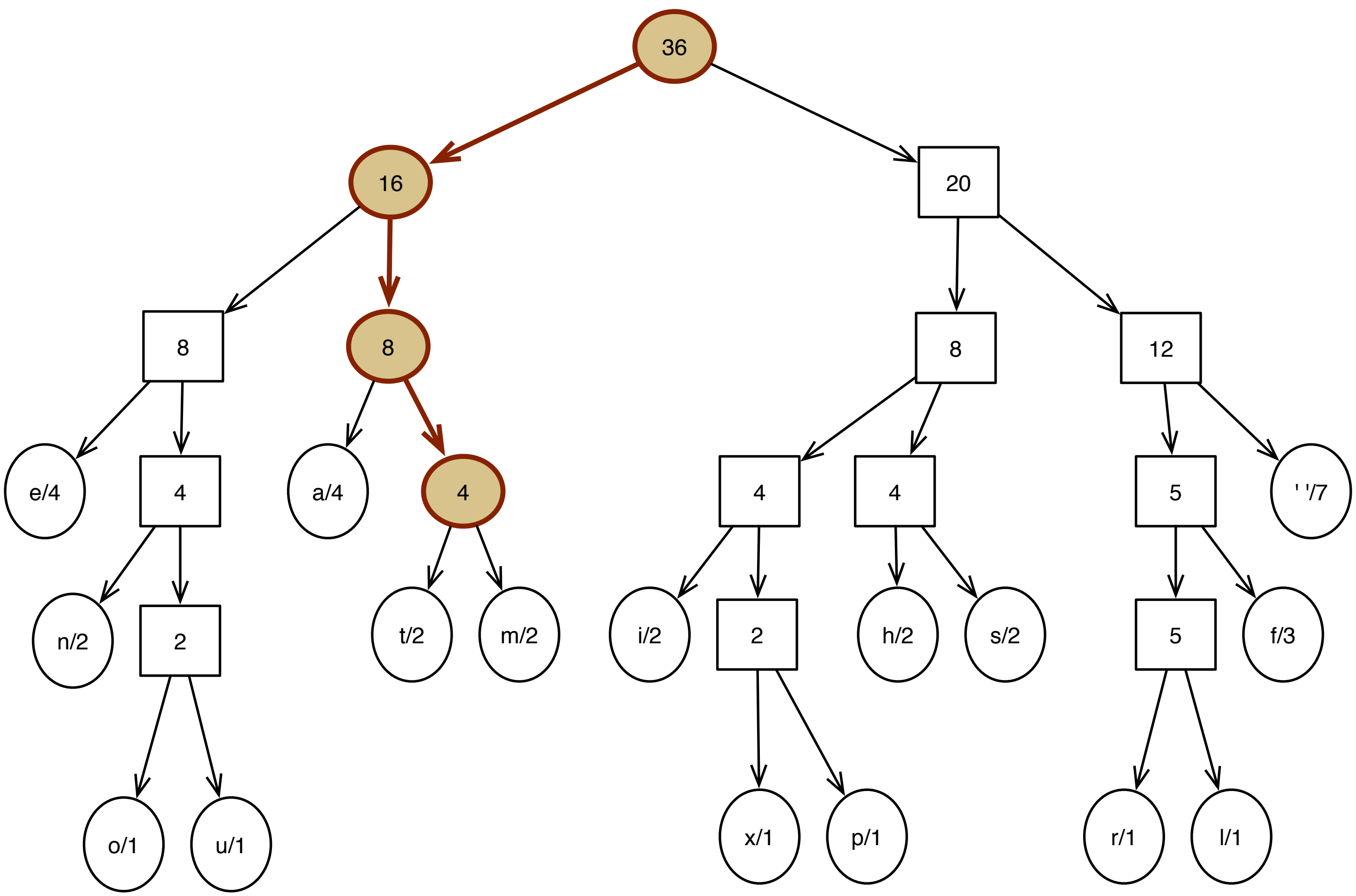
Using A Huffman Tree to <u>Decode</u> Data

0**1**1010101000101111110001011



Using A Huffman Tree to <u>Decode</u> Data

01**1**01010100010111111000101 1

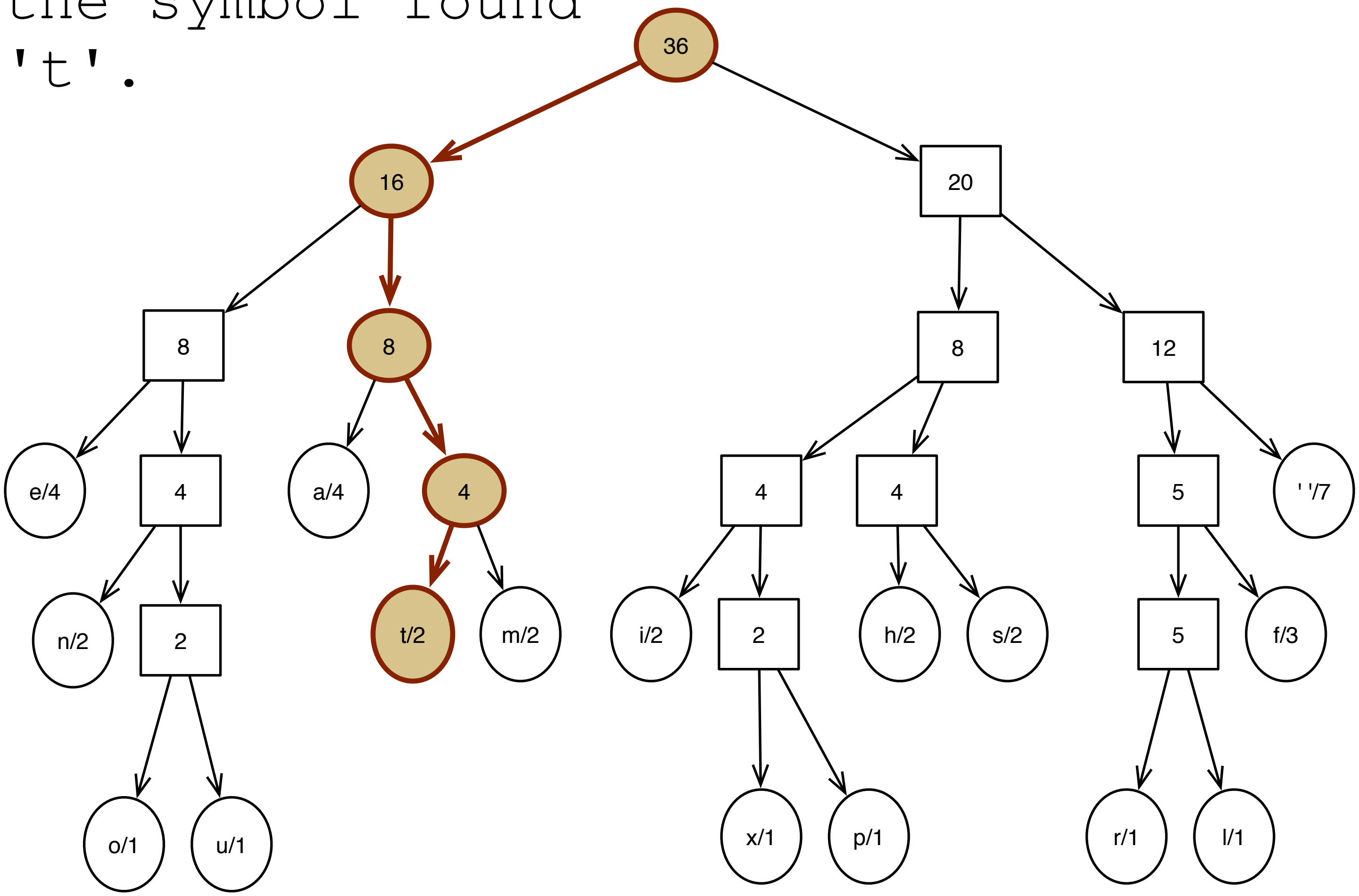


Using A Huffman Tree to <u>Decode</u> Data

011**0**10101000101111110001011

t

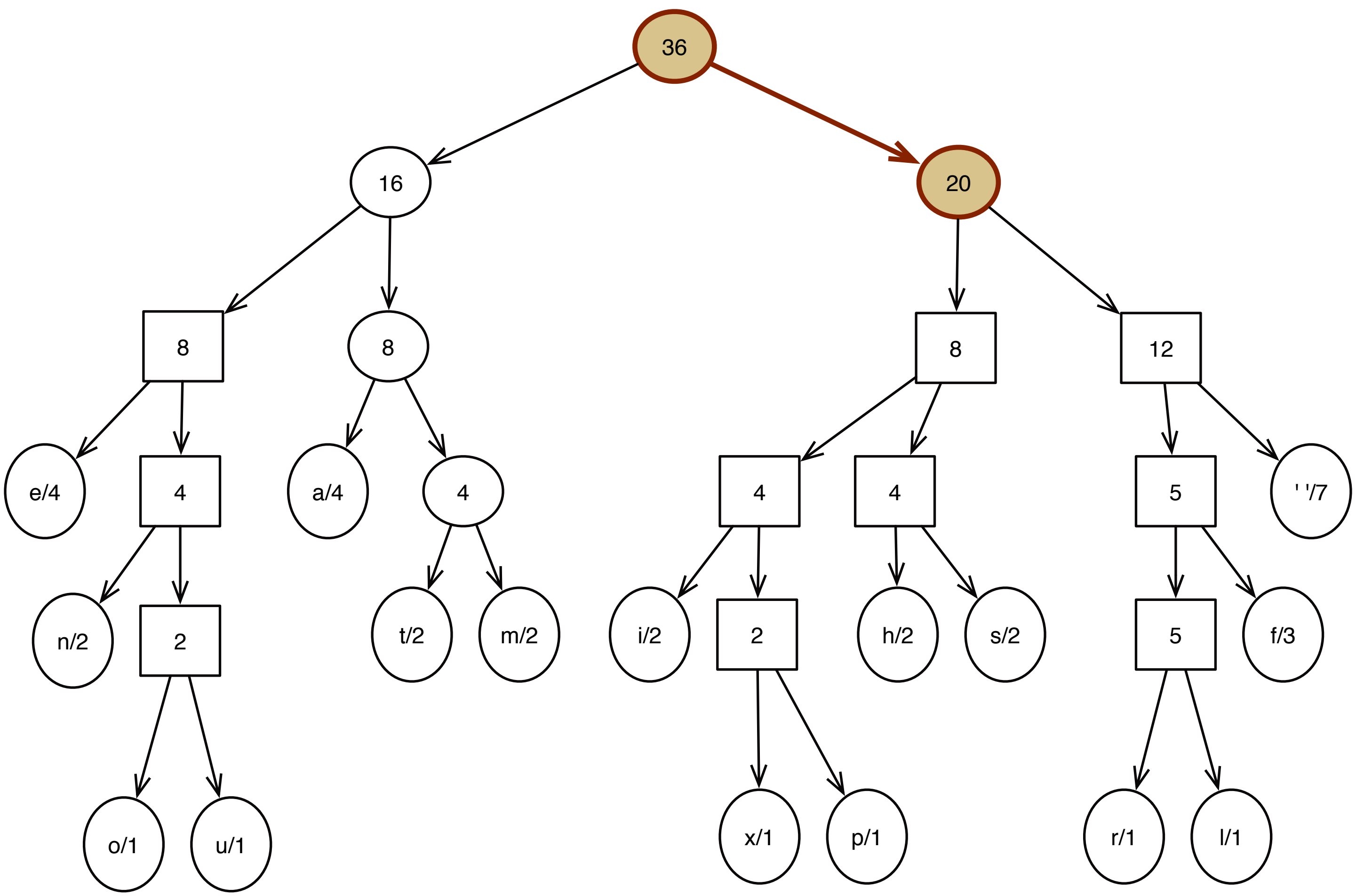
Found a leaf node!
Output the symbol found there: 't'.



Using A Huffman Tree to <u>Decode</u> Data

0110**1**0101000101111110001011

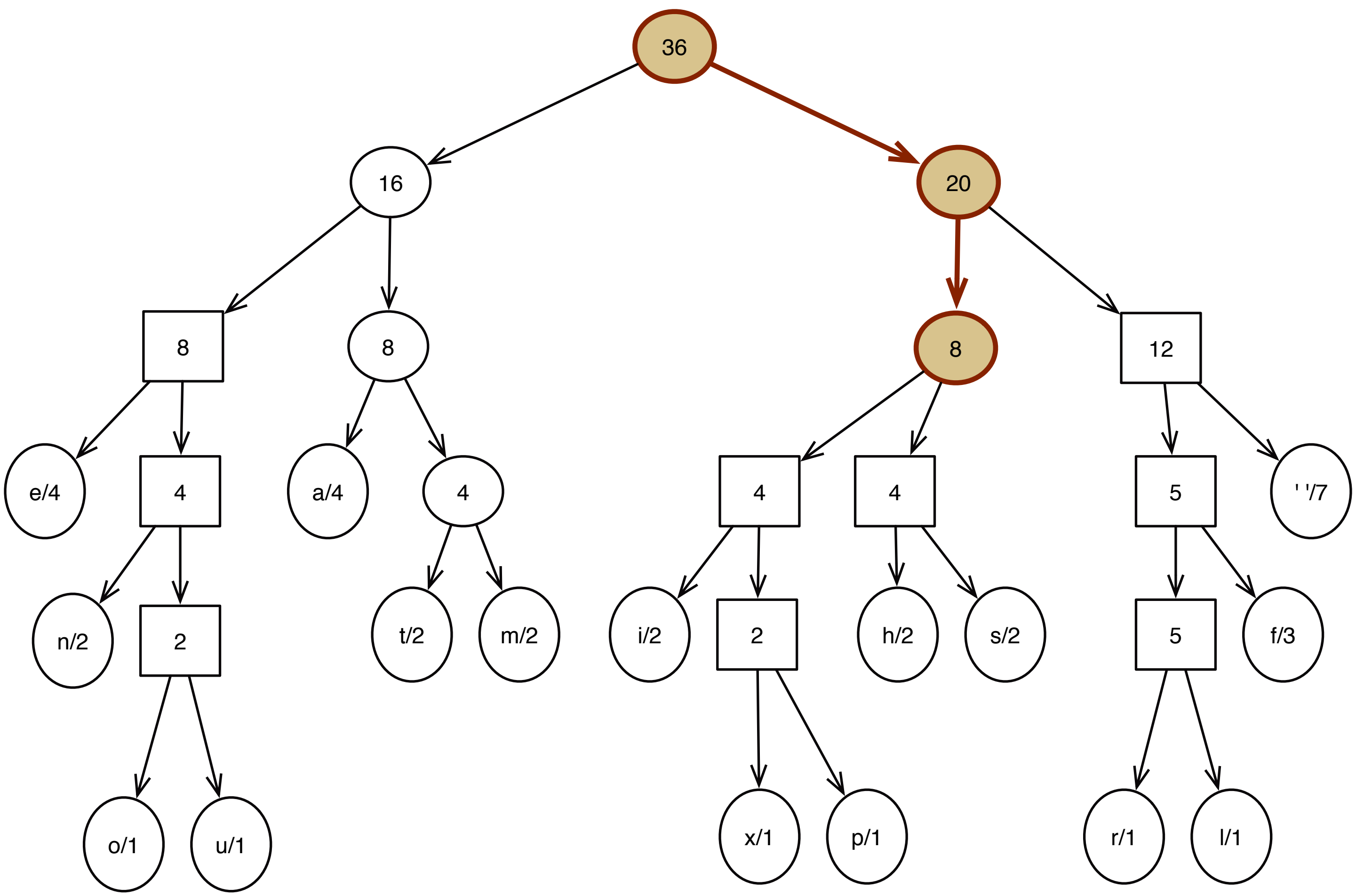
t



Using A Huffman Tree to <u>Decode</u> Data

011010101000101111110001011

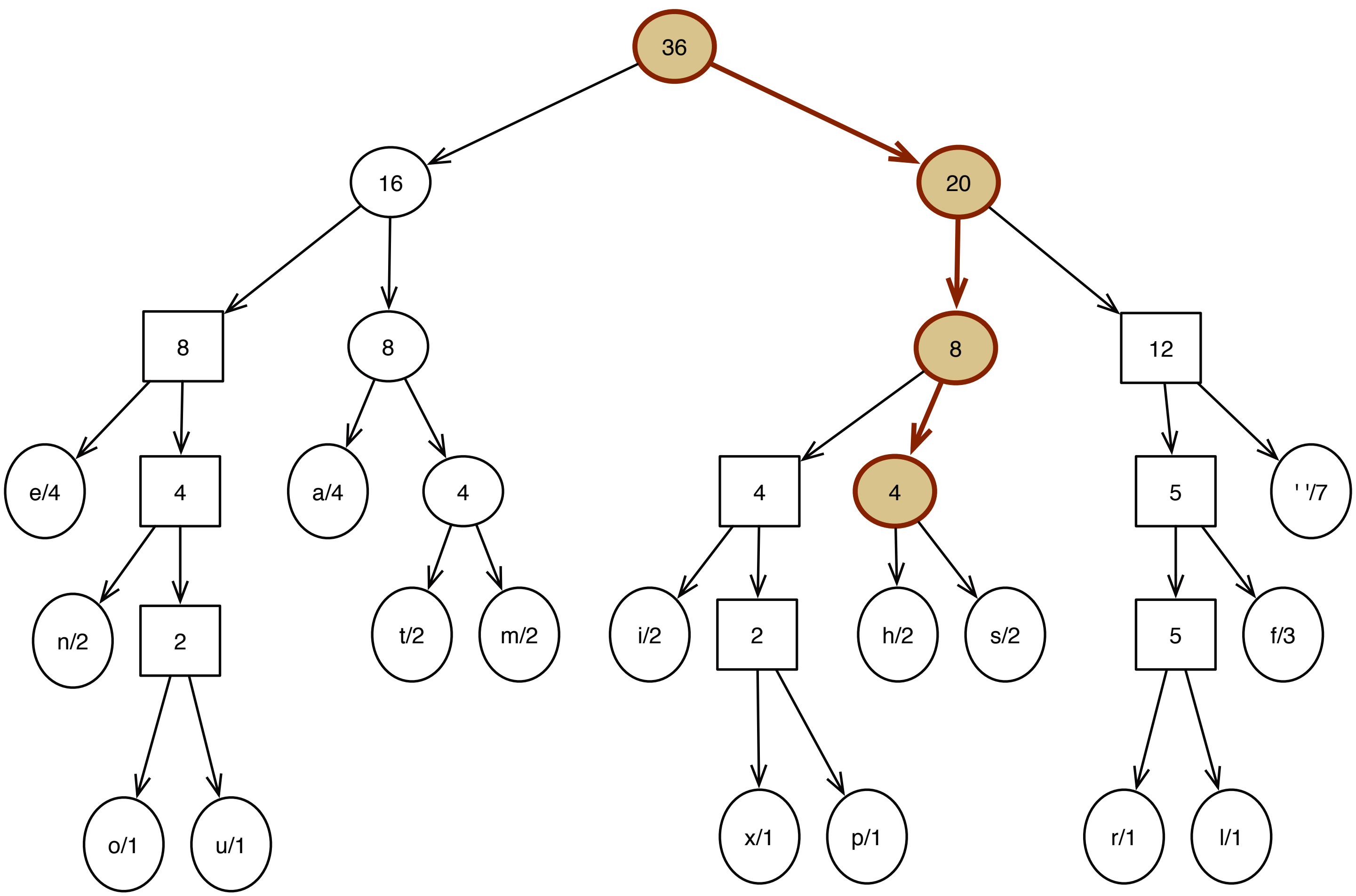
t



Using A Huffman Tree to <u>Decode</u> Data

0110101**1**010001011111110001011

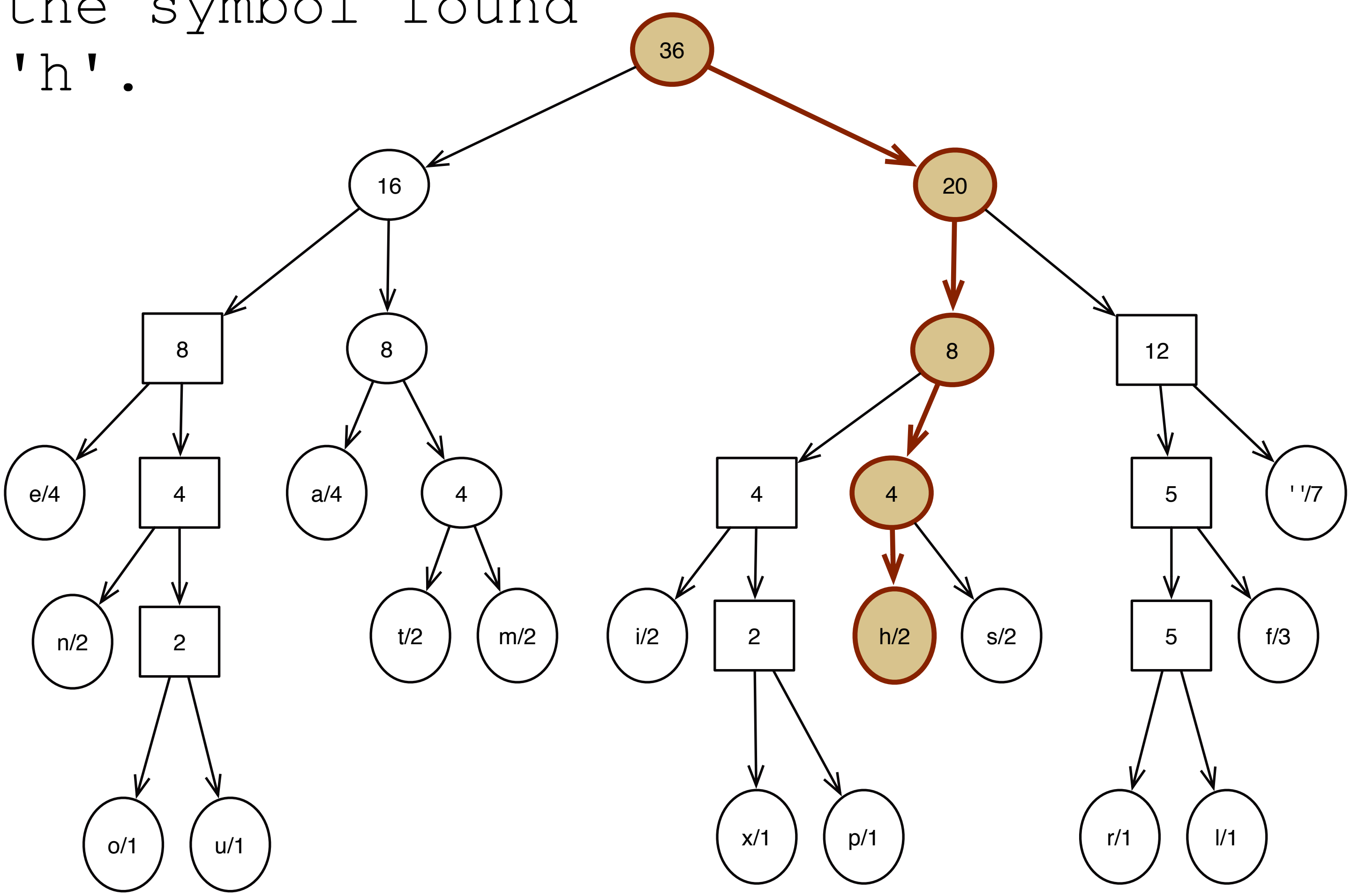
t



Using A Huffman Tree to <u>Decode</u> Data

0110101**0**10001011111110001011
th

Found a leaf node!
Output the symbol found there: 'h'.



Using A Huffman Tree to Decode Data

011010101000101111110001011
this is

It continues like this for a while, and you eventually run out of bits to decode.