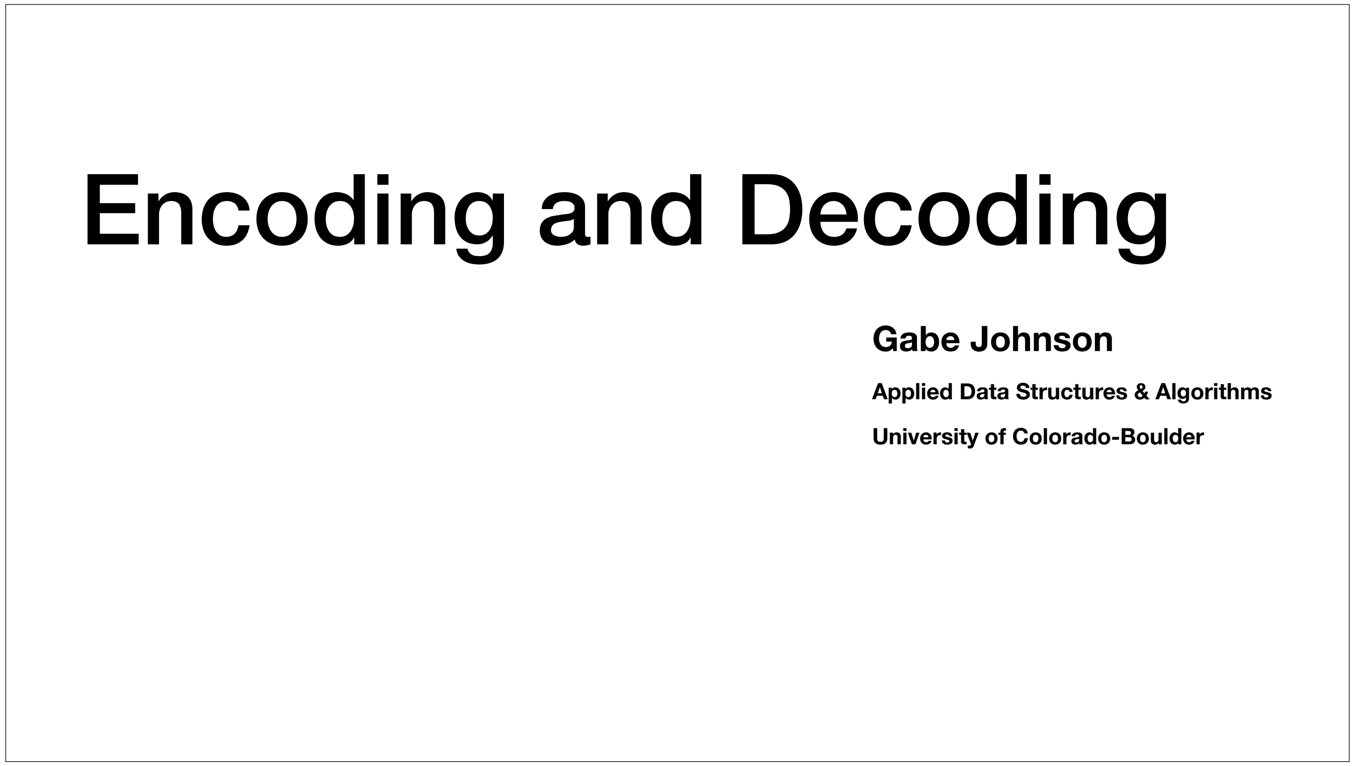


Encoding and Decoding

Gabe Johnson

Applied Data Structures & Algorithms

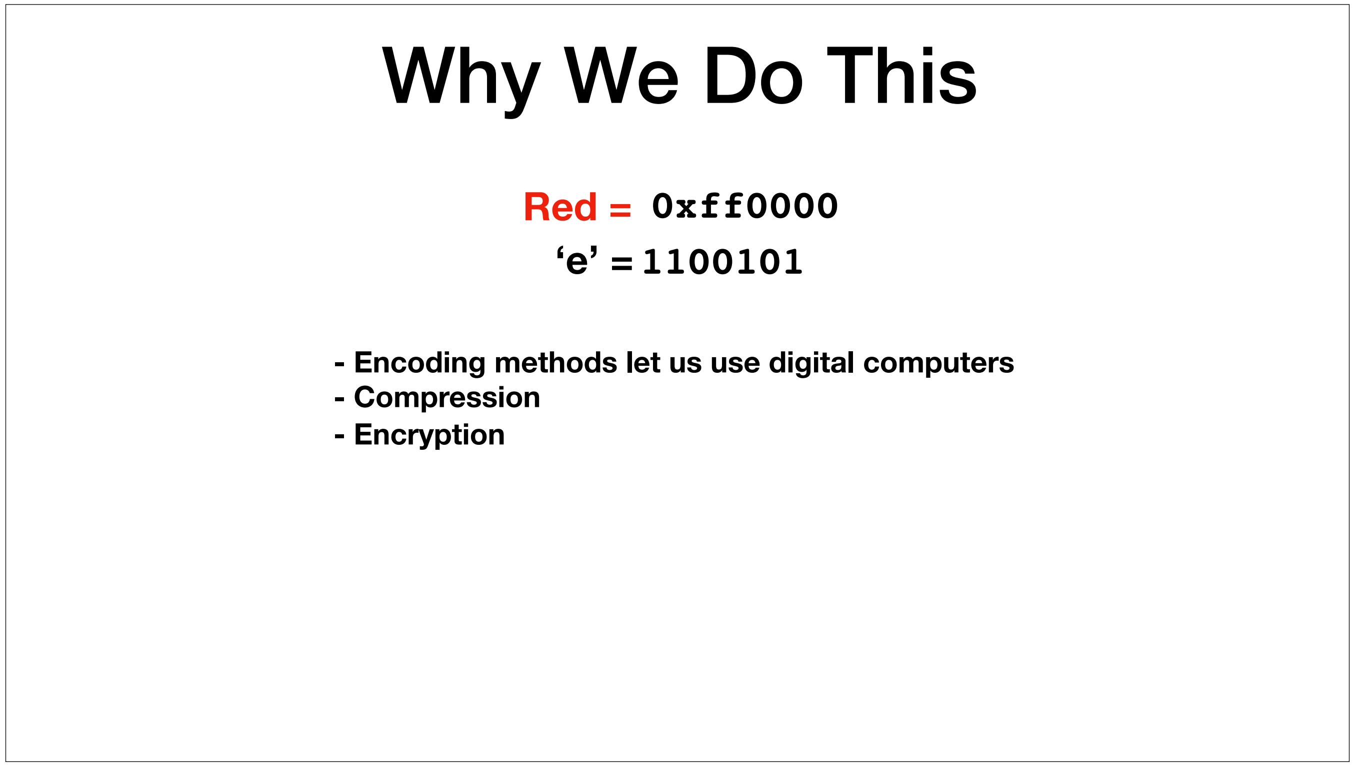
University of Colorado-Boulder

This next sequence introduces data encoding and decoding, which is sometimes called `codec`. That's how we take data from one format like a high quality audio format, and encode it into another format like MP3 or FLAC. Then once you have the encoded version, you'll probably need to decode it to do something useful, like play the song encoded in the MP3.

Episode 1

Data Coding

I’m going to start by talking about data encoding and decoding, why we care, forms it can take, and various use cases.



Why We Do This

Red = 0xff0000

‘e’ = 1100101

- Encoding methods let us use digital computers
- Compression
- Encryption

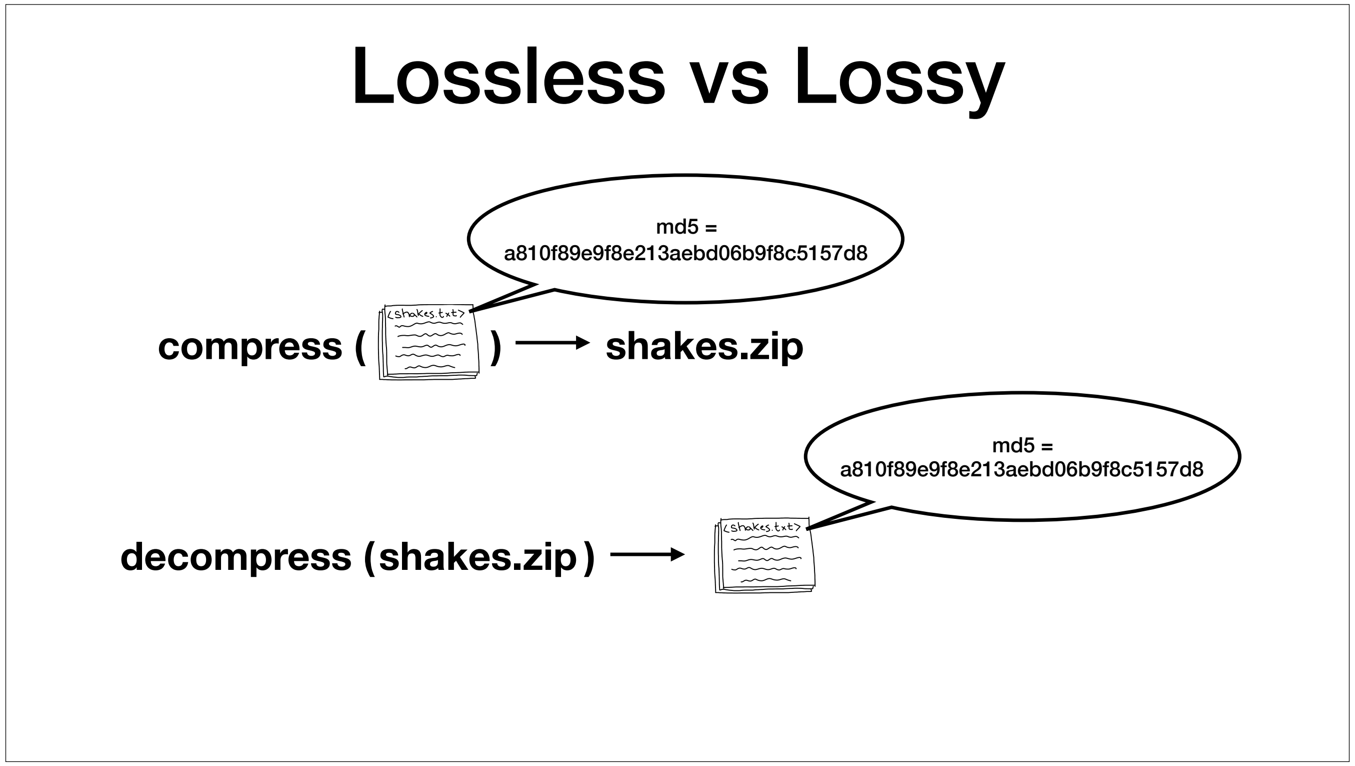
All the data you use in a computer is encoded in some way. It could be a standard encoding, like the standard that says `#ff0000` is red, or that the letter e will be stored by the seven bits `1100101`. After all, digital computers can only store numbers --- how we interpret those numbers is up to us. We need to devise an encoding scheme.

<next>

So, for starters, encoding is a basic requirement to do anything useful with a digital computer.

Some encoding is meant to squash data down into a smaller size. This is called compression, and it is one manifestation of encoding. We do this to make better use of disk space and transmit information faster.

Some encoding is meant to hide information, even in plain sight. You can encode some text using encryption to get cyphertext, which can (or should) only be decoded using some decryption method.



Lossless vs Lossy

md5 =
a810f89e9f8e213aebd06b9f8c5157d8

compress ( ) → shakes.zip

md5 =
a810f89e9f8e213aebd06b9f8c5157d8

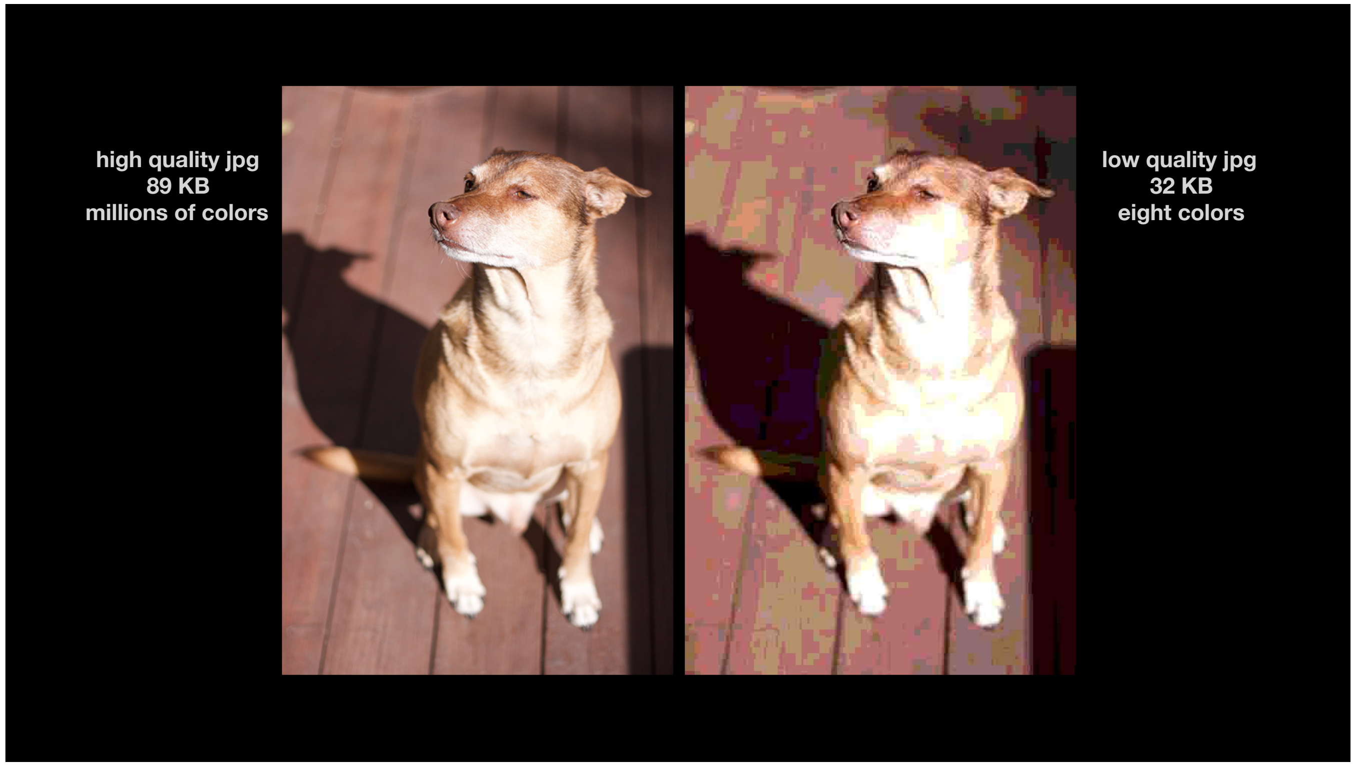
decompress (shakes.zip) →

An encoding can either be lossless or lossy, and which you choose depends on the situation.

A lossless codec will let you take your data (works of shakespeare), encode it into some other format (shakes.zip), and then decode that version into a copy that is identical to the original.

<next>

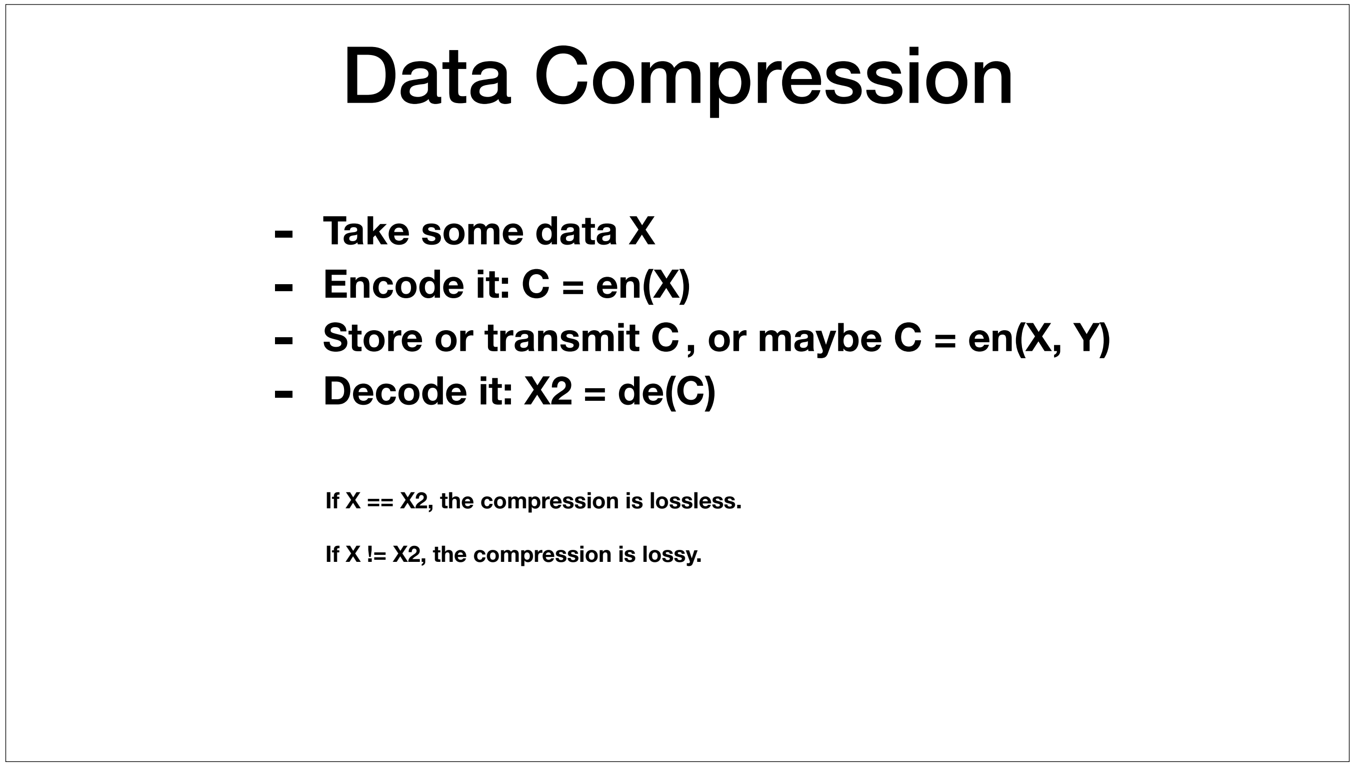
If you need bit-for-bit replication, you'll need a lossless format.



high quality jpg
89 KB
millions of colors

low quality jpg
32 KB
eight colors

A lossy codec sacrifices some information during the encoding process, so that when it is decoded we have an approximation of the original, but not a bit-for-bit replica. Now, exactly which information gets left out depends on the application. A lossy image format might reduce the number of colors. Or a lossy audio format might sacrifice frequencies that humans can't hear anyway.



Data Compression

- **Take some data X**
- **Encode it: C = en(X)**
- **Store or transmit C , or maybe C = en(X, Y)**
- **Decode it: X2 = de(C)**

If X == X2, the compression is lossless.

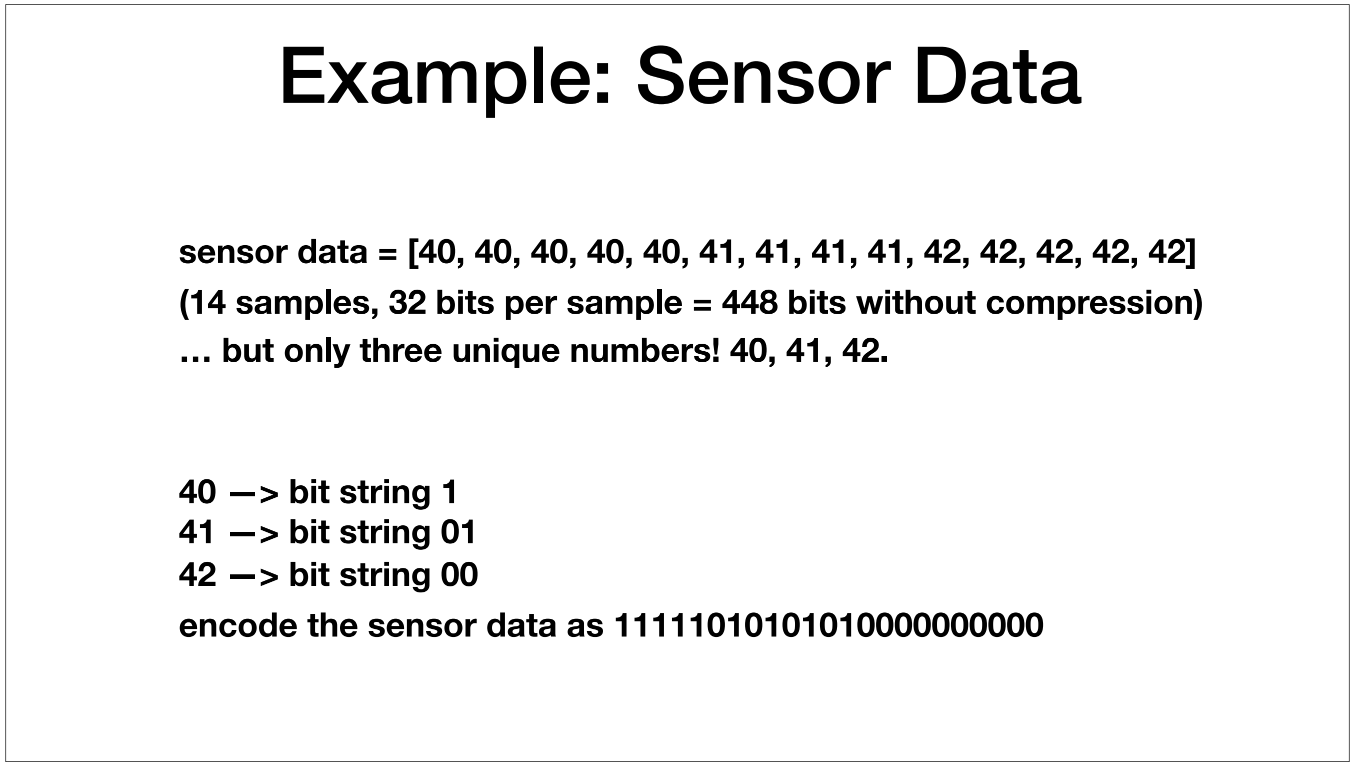
If X != X2, the compression is lossy.

The general idea with compression is to take any data, and do some math on it to get a version that requires less information, while retaining the ability to uncompress it into something that matches the first version to some degree. This might be entirely a function of the input, or **<next>** you might have some external information, like statistics about how common certain words appear.

The strategy you take might depend on the nature of the input. So you'd use one method for compressing sound files (mp3), another for compressing images (jpg), and another for compressing text.

And within a category, like compressing text, you can tweak the strategy based on the specific information. For example, compressing English text vs. German vs. Chinese.

If you go full circuit with your data, and in the end you decoded version is not identical to the input, you’re using a lossy compression scheme. MP3 is like that.



Example: Sensor Data

sensor data = [40, 40, 40, 40, 40, 41, 41, 41, 41, 42, 42, 42, 42, 42]
(14 samples, 32 bits per sample = 448 bits without compression)
… but only three unique numbers! 40, 41, 42.

40 —> bit string 1
41 —> bit string 01
42 —> bit string 00
encode the sensor data as 11111010101010000000000

Let's go through an example of lossless data compression. We have a remote, solar-powered temperature sensor that is going to transmit data every once in a while in a batch in order to save power.

Let's say the sensor has recorded these values over the course of the morning:

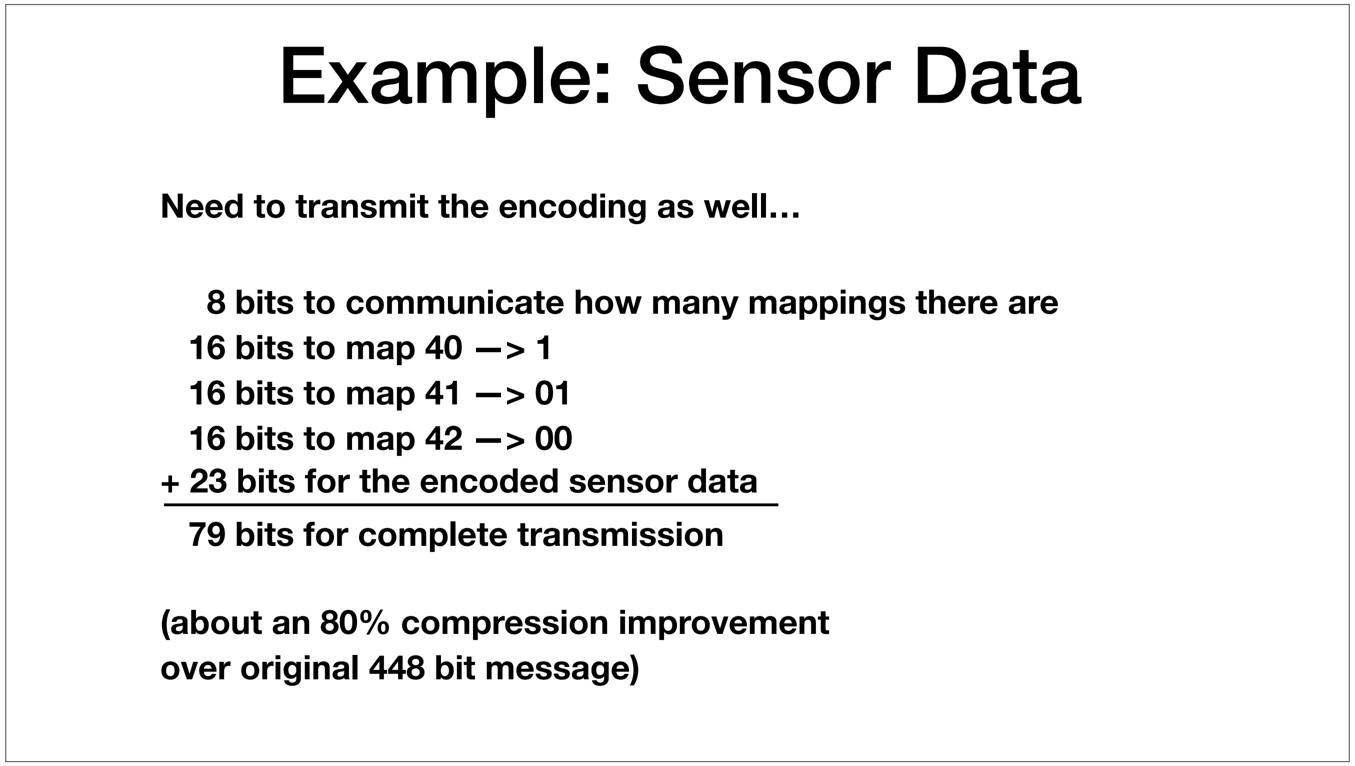
<40, 40, 40, 40, 40, 41, 41, 41, 41, 42, 42, 42, 42, 42>

<next> There are 14 total samples, and each one is stored as a 32-bit integer in the sensor's memory. If we were to transmit this information without compressing it at all, it would require 448 bits of data.

But let's look at the data. There's only three numbers! **<next>** So we could compress this by coming up with an encoding strategy based on the data we have. Let's encode the samples like this: **<next>**

40 == bit string 1
41 == bit string 01
42 == bit string 00

In that case the data encoding would be **<next>** 11111010101010000000000. That’s 23 bits of information.



Example: Sensor Data

Need to transmit the encoding as well…

8 bits to communicate how many mappings there are
16 bits to map 40 —> 1
16 bits to map 41 —> 01
16 bits to map 42 —> 00
+ 23 bits for the encoded sensor data
79 bits for complete transmission

(about an 80% compression improvement
over original 448 bit message)

So in addition to those 23 bits, we also have to send up information about how it was encoded so that the other side knows how to deal with it. I'm just making up a scheme here.

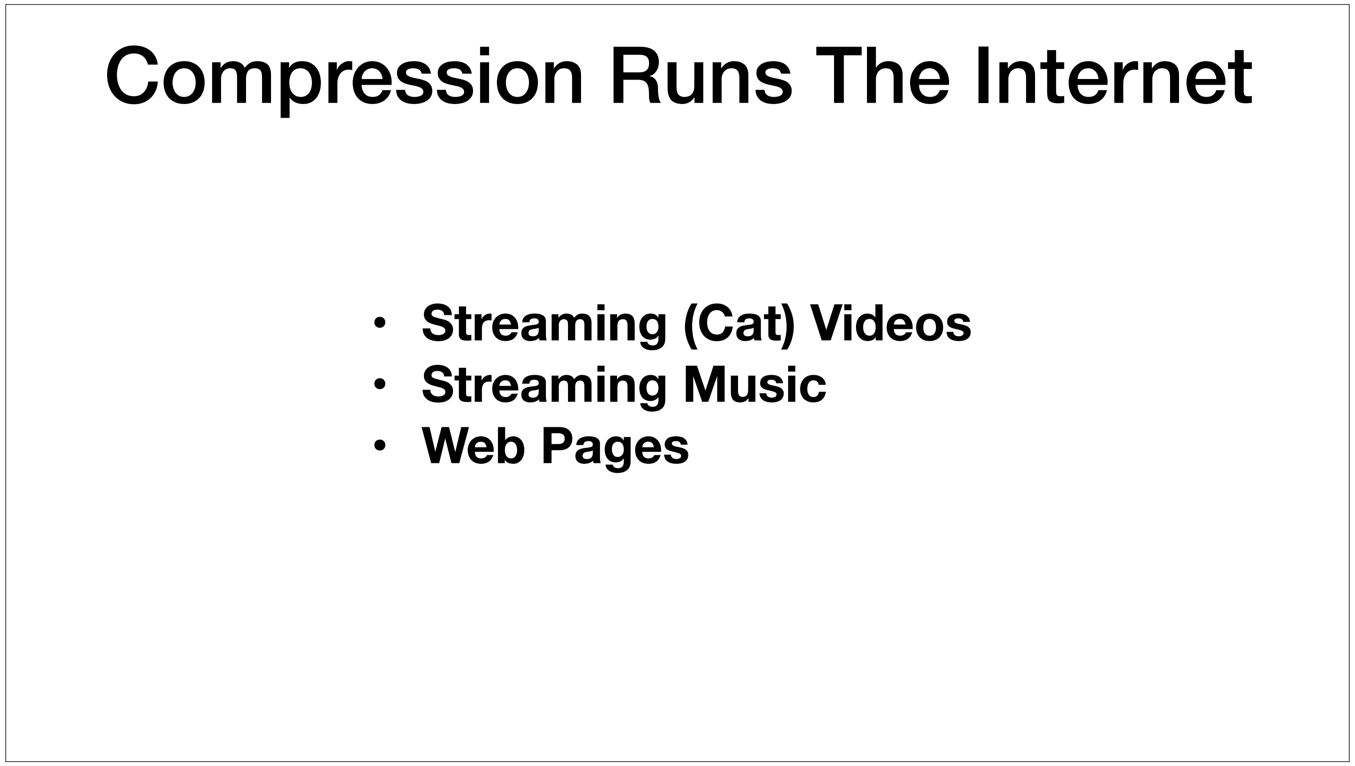
Let's say in this made up scheme, there's one byte (8 bits) to communicate how many mappings there are. One byte lets us send up to 256 mappings.

Then each mapping gets 16 bits, to say for example the original number 41 maps to the bitstring 01.

Last but not least, we also transmit the 23 bits of encoded data.

<next>

This adds up to 79 bits for encoded data set. That's about an 80% reduction. And that reduction would be much much better if we had more than 14 samples!

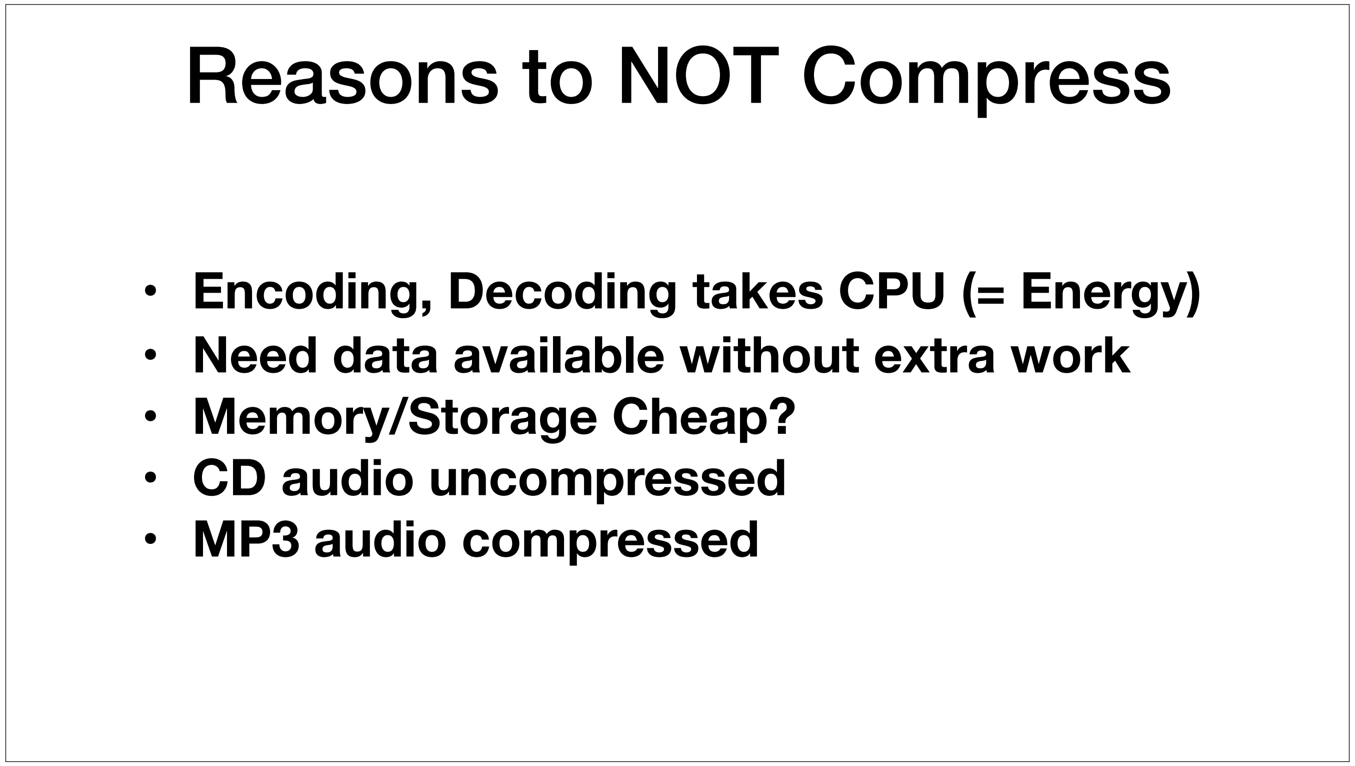


Compression Runs The Internet

- **Streaming (Cat) Videos**
- **Streaming Music**
- **Web Pages**

Data compression is one of the things that lets us have the Internet. Without it we'd not be able to have streaming video or audio. Even plain old boring text transfer is improved by compressing data when transmitting, because web pages themselves are just marked up text.

Any time you have statistical reduancy in the information, it can be compressed.

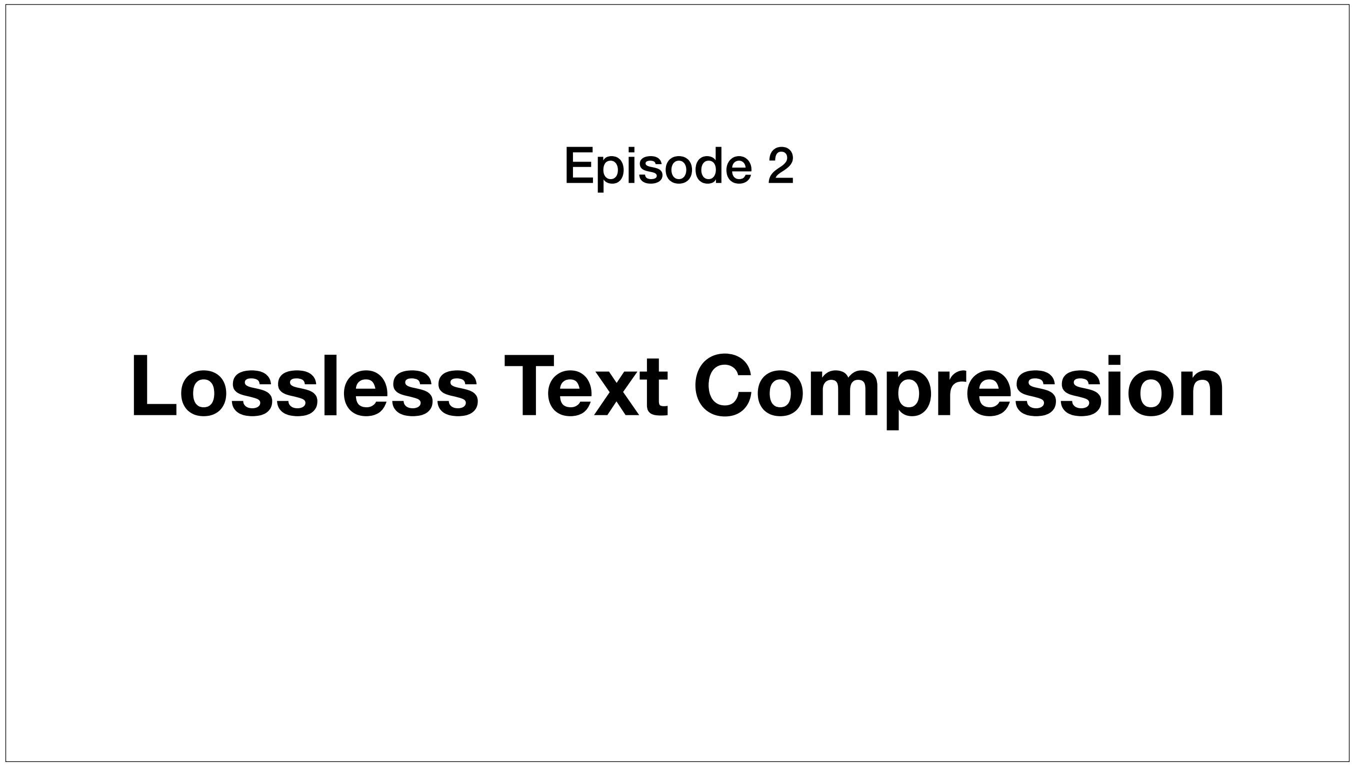


Reasons to NOT Compress

- **Encoding, Decoding takes CPU (= Energy)**
- **Need data available without extra work**
- **Memory/Storage Cheap?**
- **CD audio uncompressed**
- **MP3 audio compressed**

But there are reasons to not compress your data. Compressing and decompressing takes computational overhead. And that might be an annoying step to decompress the data every time you need it. If memory or storage is cheap but CPU time is not, it might make sense to leave the data uncompressed.

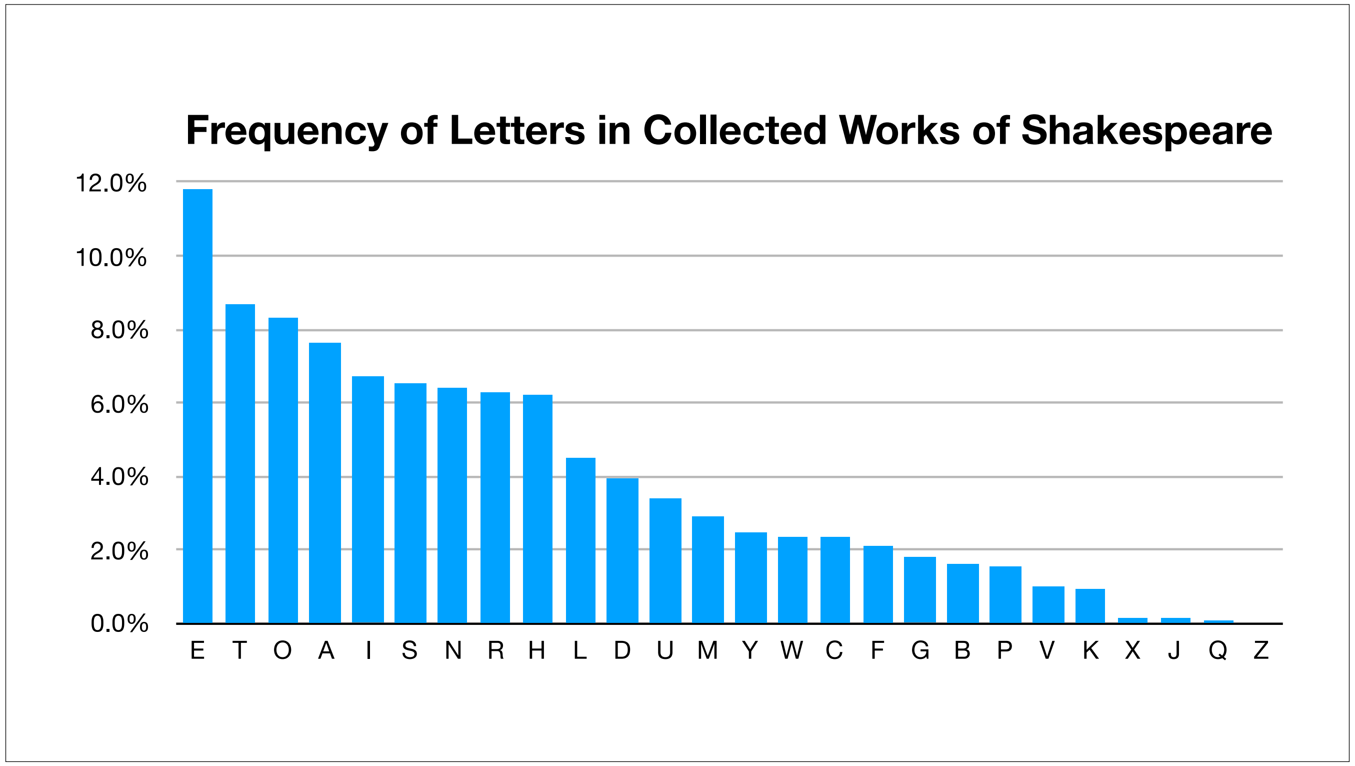
For example, CD audio is not compressed, so it takes little processing power to play the music. But MP3 audio is compressed and takes computation to decompress it. We now have hardware-based mp3 decoders that makes this practical now.



Episode 2

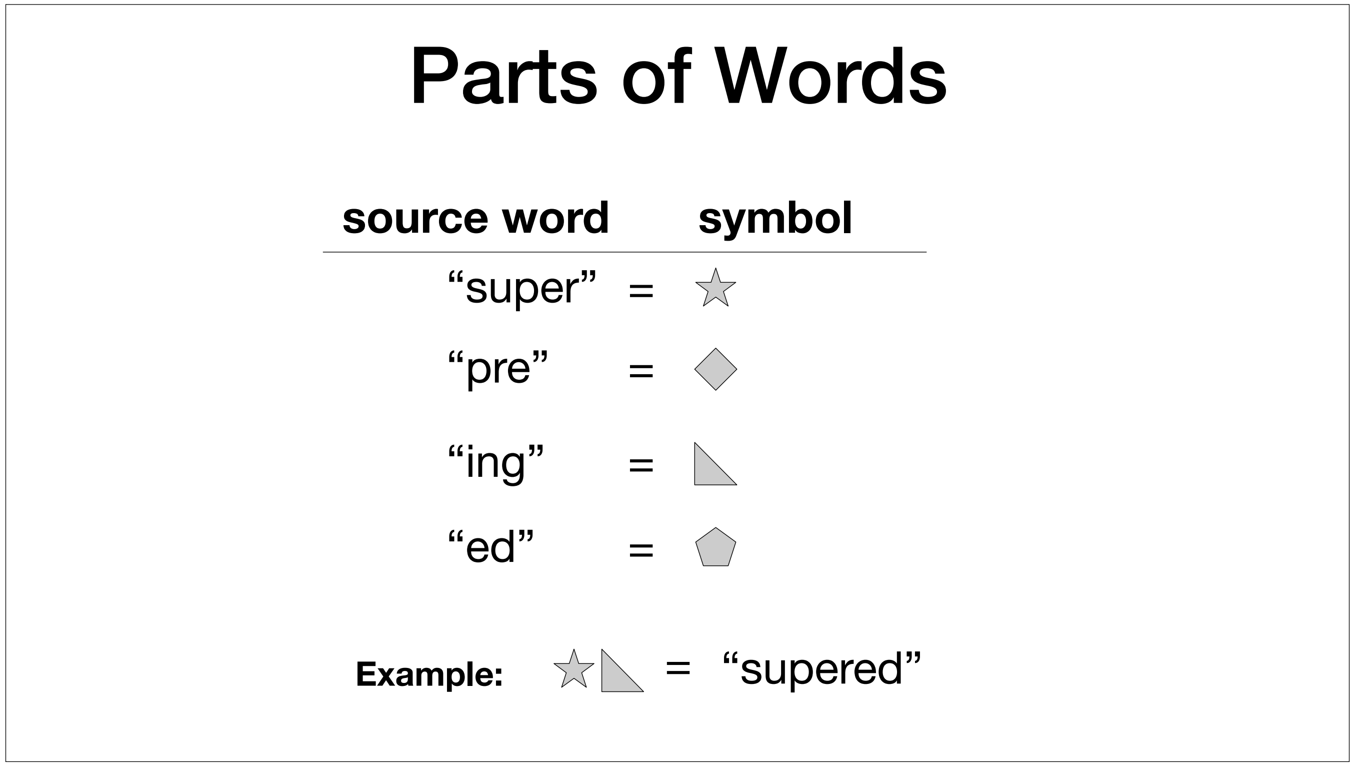
Lossless Text Compression

Text is everywhere, in every written language. And if you can twist your perspective a bit, even other media like images and genetics can have text-like properties. So people have found ways to encode large blocks of text into smaller blocks of data, so they can be stored on disks or transmitted over the internet more efficiently.



Here's a fun chart that shows the frequency of letters in the collected works of Shakespeare. I got the source text off of Project Gutenberg, and wrote a little Go program to do the analysis.

You probably already had an intuitive awareness that some letters like E appear way more often than letters like Q or Z. But just looking at this chart you can see that the first six or so letters comprise like half of the overall text. Statistically speaking, if you were to open this book and close your eyes and pick a letter at random, there's like a 12% chance it would be an E.

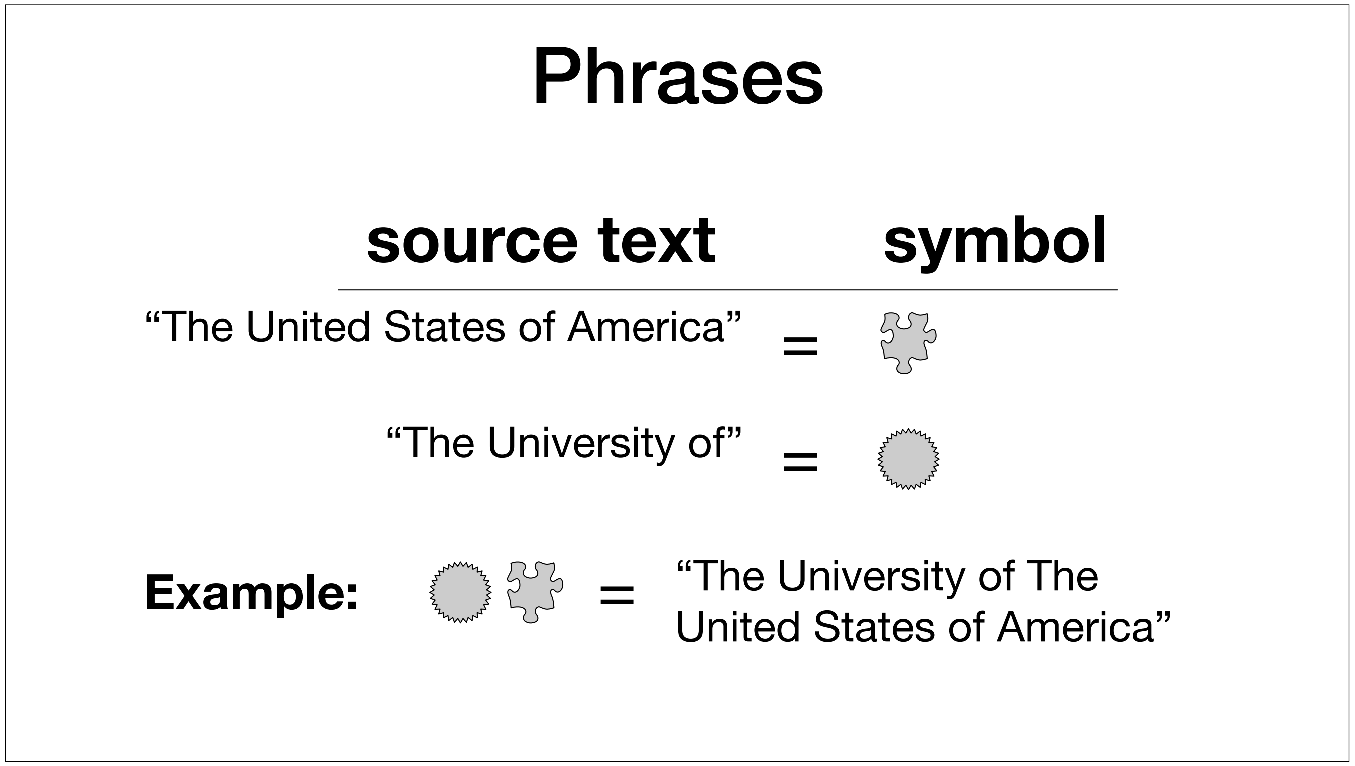


Parts of Words

source word		symbol
“super”	=	☆
“pre”	=	◇
“ing”	=	◺
“ed”	=	⬠

Example: ☆◺ = “supered”

Fancier approaches look at common sequences of letters, like suffixes '-ing' or '-ered', and prefixes like 'super-' and 'pre-'. The idea here is to find some common sequence of letters or words and be able to substitute them with some shorter symbol. Once we agree on a substitution, and by that I mean an encoding, we can write the shorter symbols instead, and anyone who knows the encoding scheme will be able to decode it.



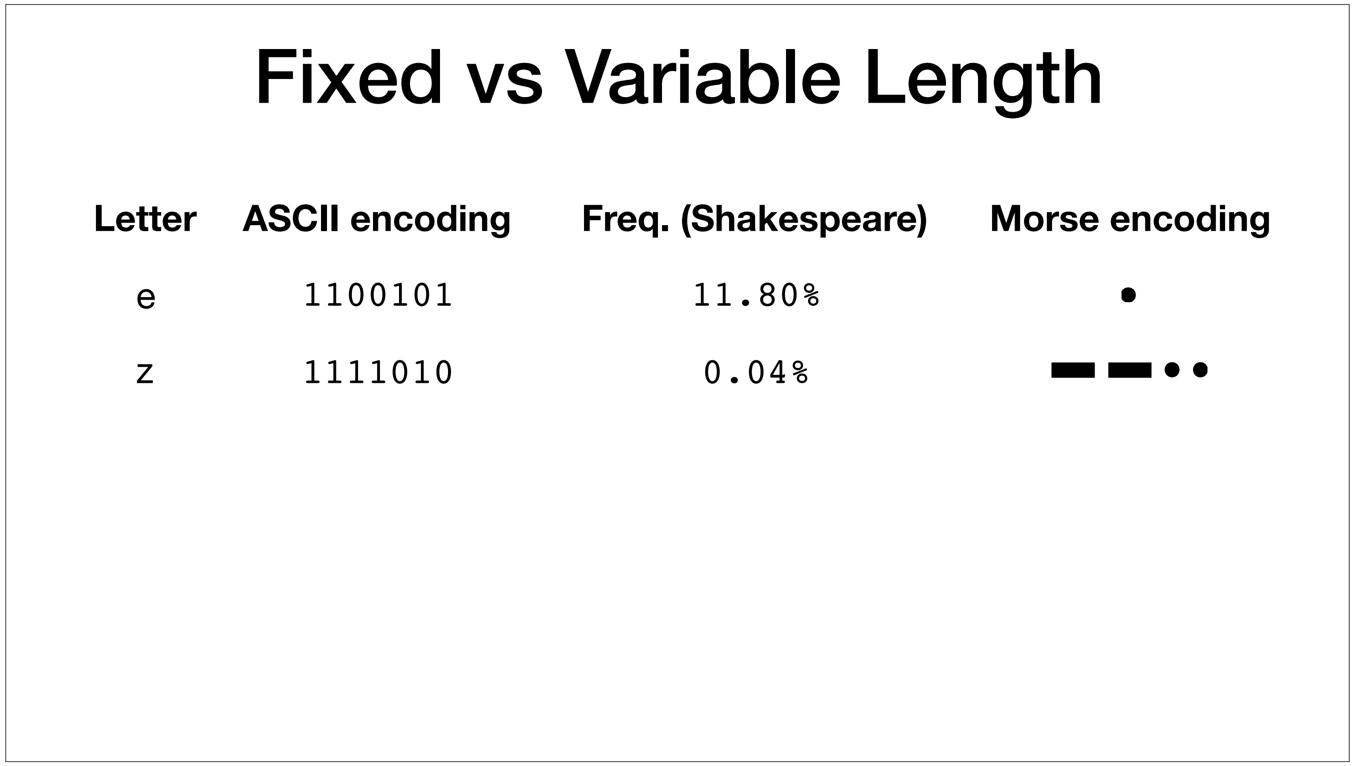
Phrases

source text		symbol
“The United States of America”	=	🧩
“The University of”	=	✹

Example: ✹🧩 = “The University of The United States of America”

And you don't have to stop at word boundaries. Some words just appear in the same sequence, like "The United States of America", or "The University of". So I'm not sure how to pronounce that in English, sawblade-puzzle piece, but in this encoding scheme, that's The University of The United States of America.

At any rate, you can chop up your input however you'd like; some approaches can work better than others. It depends on the length of the input, what language it is written in, and so forth. Once you've chopped up the language into chunks, then you can assign symbols to each chunk, and put those symbols in the right order to encode the original text but in a much compact format.



Fixed vs Variable Length

Letter	ASCII encoding	Freq. (Shakespeare)	Morse encoding
e	1100101	11.80%	•
z	1111010	0.04%	▬ ▬ • •

A common way to store characters in a computer is with an encoding scheme where each character has the same fixed length. ASCII is a common one, and it uses seven bits to represent each character.

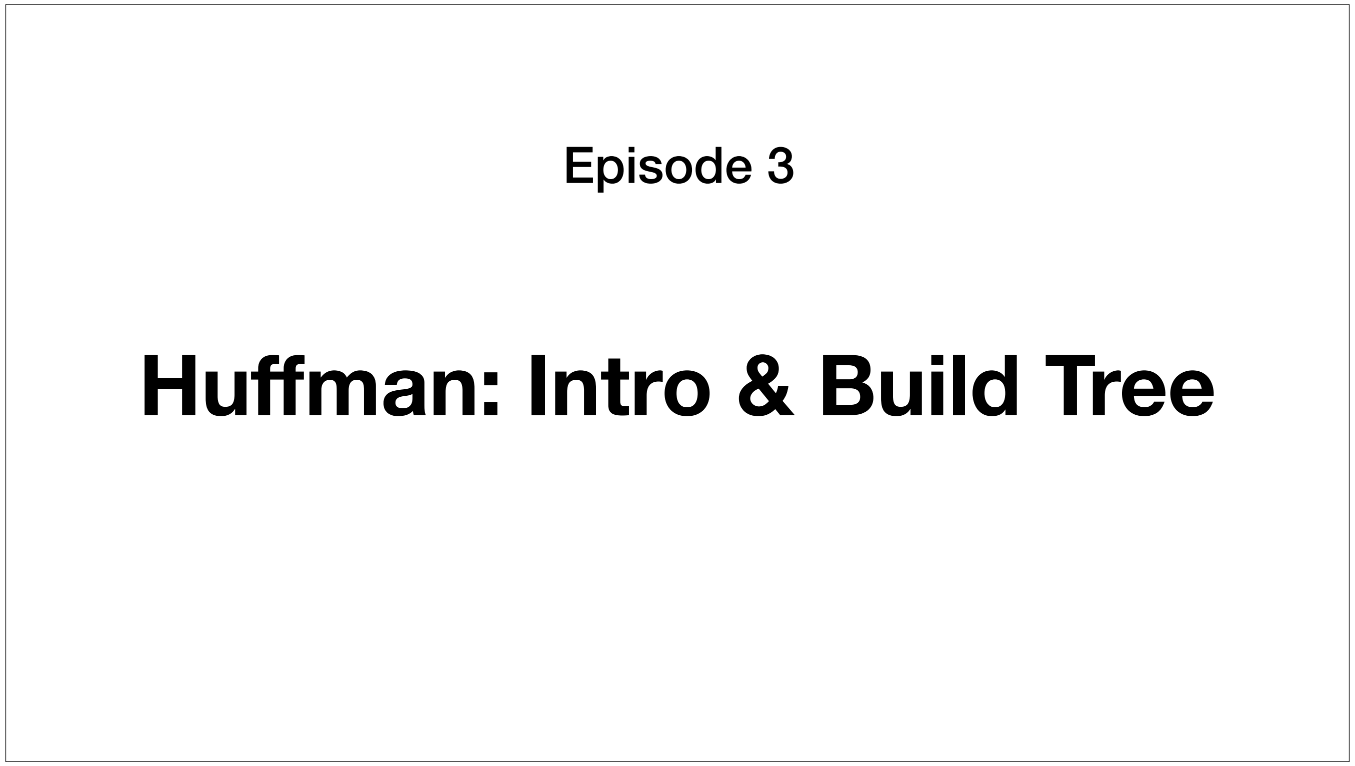
This is useful because it is very easy to write and read. But it is not very informationally efficient. E takes just as many bits to encode as Z, even though E is way way way more common than Z.

<next>

Some encoding methods don't have a fixed size for each chunk to convey. Morse code is like that. E is super common, so it is short (single dot!), and Z is not common so it is pretty long (dash dash dot dot).

Using a fixed length code makes decoding it easier, since the reader can grab chunks at a predictable rate, like a byte at a time. But it means the encoded message is likely longer than it would be using a variable length code.

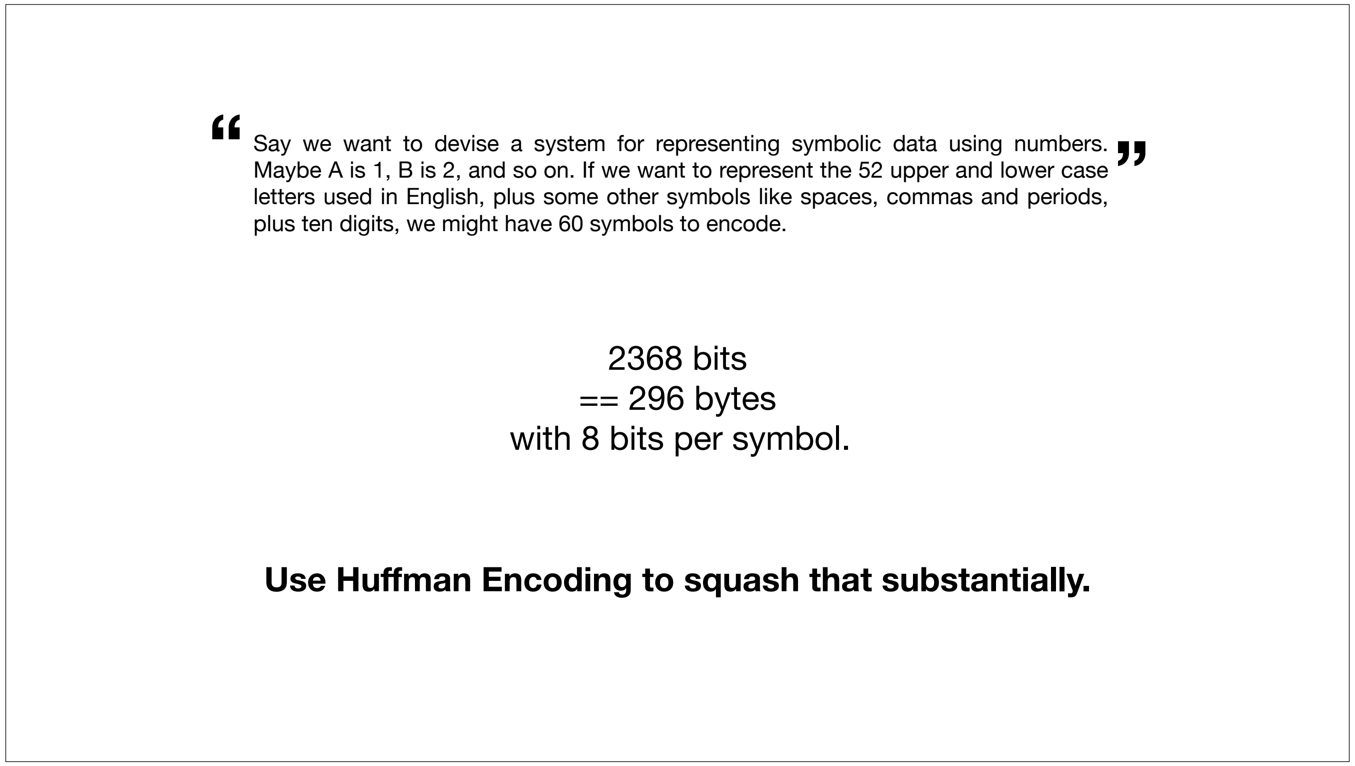
Using a variable length code means the reader has to study the message carefully as it is parsed, sometimes using external information. For example, both the ASCII and Morse Code strategies have established mappings between characters and encoding. But some other methods might include that mapping in the message itself, or the mapping has to be transmitted separately.



Episode 3

Huffman: Intro & Build Tree

In this episode we’re going to look at an encoding scheme called Huffman encoding, and learn how to build the data structure used to encode and decode data.



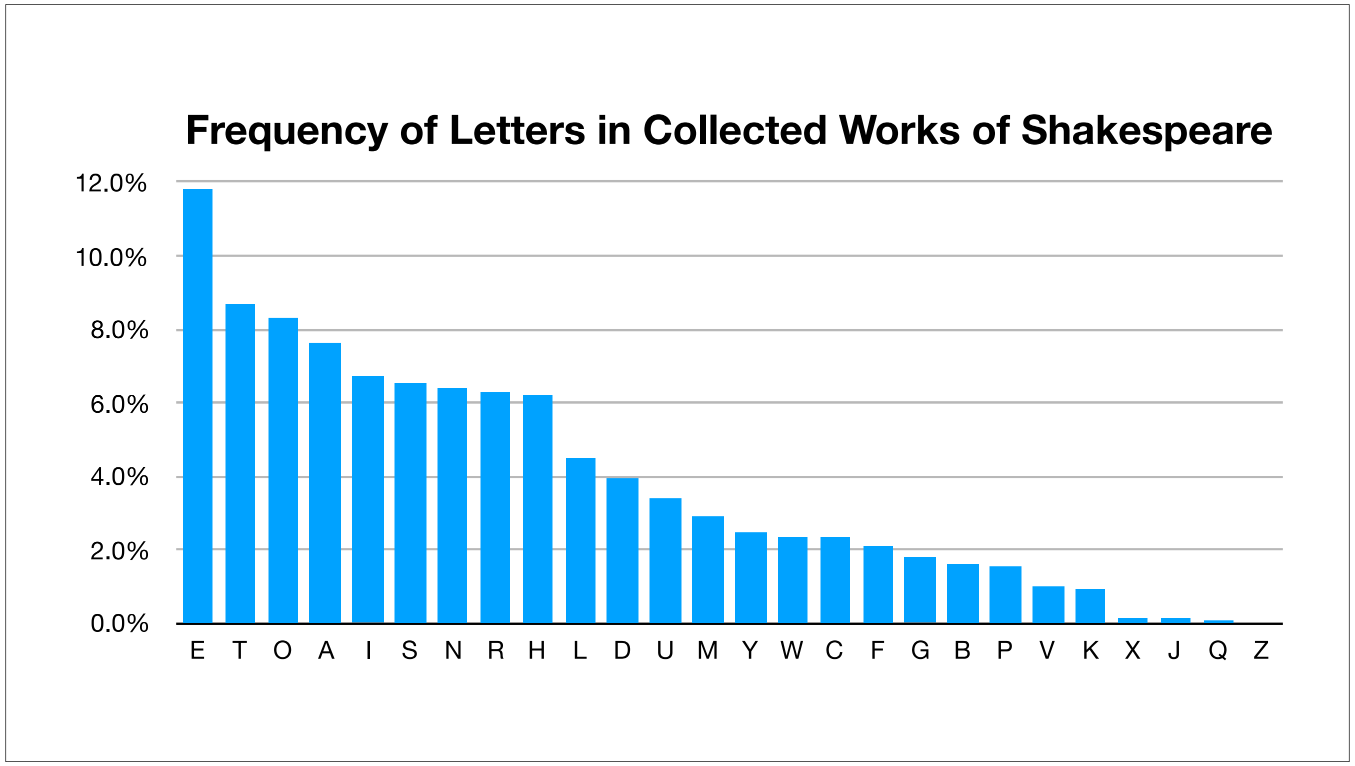
> “Say we want to devise a system for representing symbolic data using numbers. Maybe A is 1, B is 2, and so on. If we want to represent the 52 upper and lower case letters used in English, plus some other symbols like spaces, commas and periods, plus ten digits, we might have 60 symbols to encode.”

2368 bits
== 296 bytes
with 8 bits per symbol.

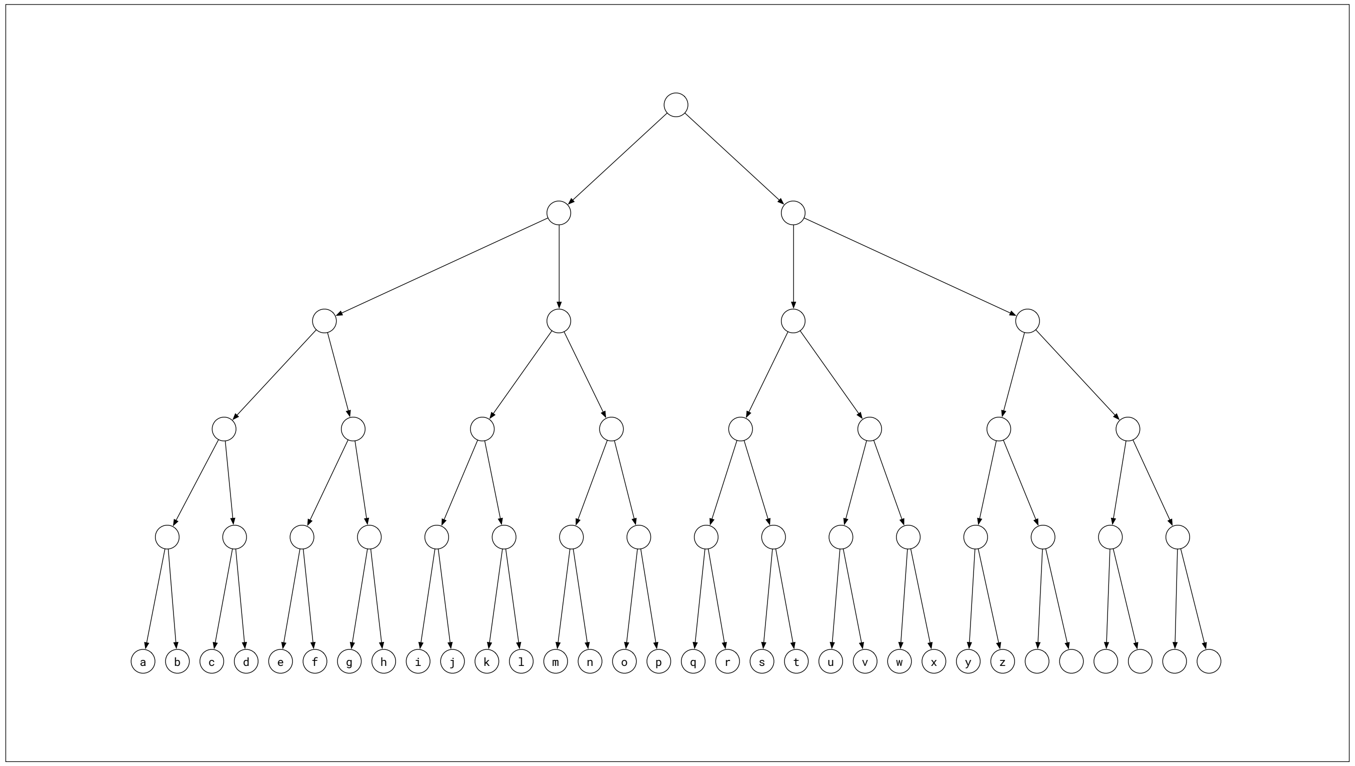
Use Huffman Encoding to squash that substantially.

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, plus ten digits, we might have 60 symbols to encode.

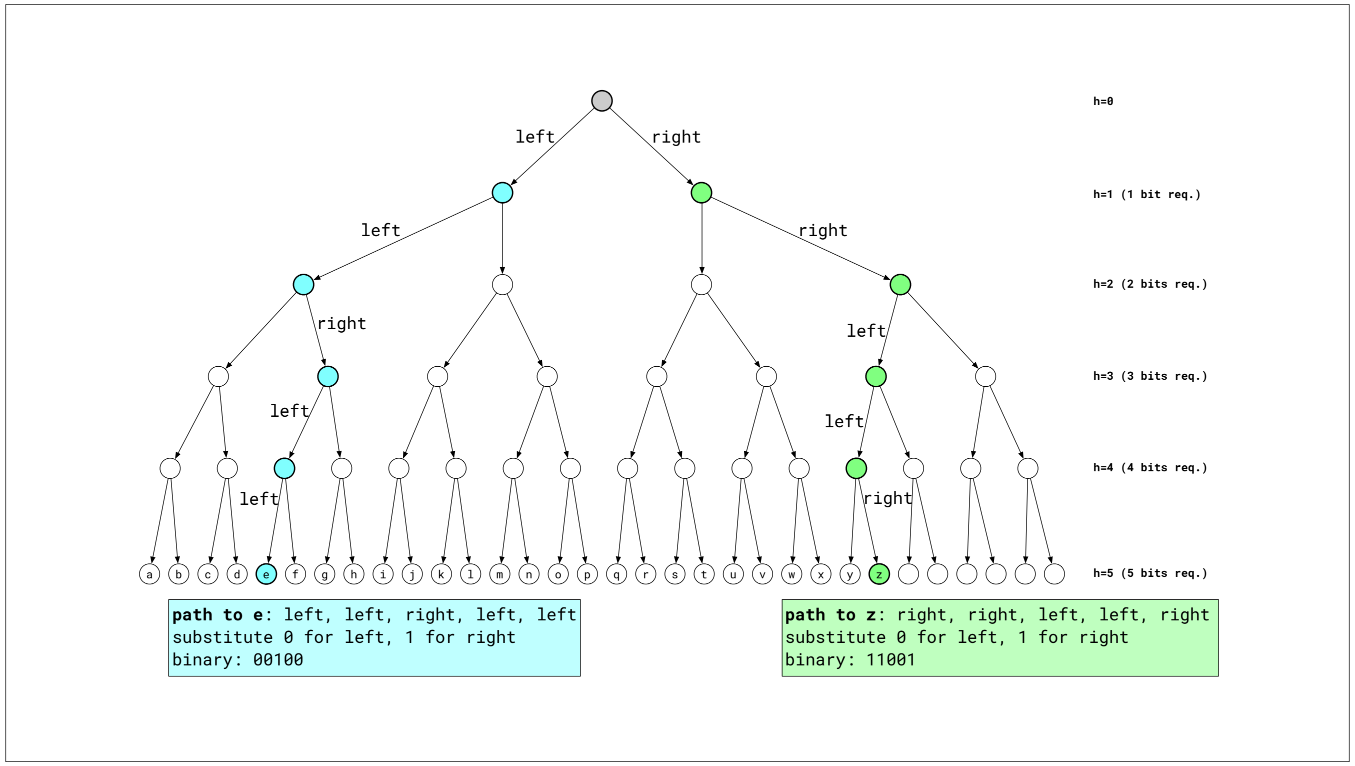
Using this scheme, we can encode the entire paragraph above using **<next>** 2368 bits (296 bytes), using eight bits per symbol. But we could compress this data so it it uses much less memory. Huffman encoding is one way to do this. It is also a really fun algorithm.



This is the distribution of letters from the collected works of shakespeare, sorted from most to least frequent. It only includes letters, so no punctuation or spaces. The letter E is almost 300 times more common than 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.

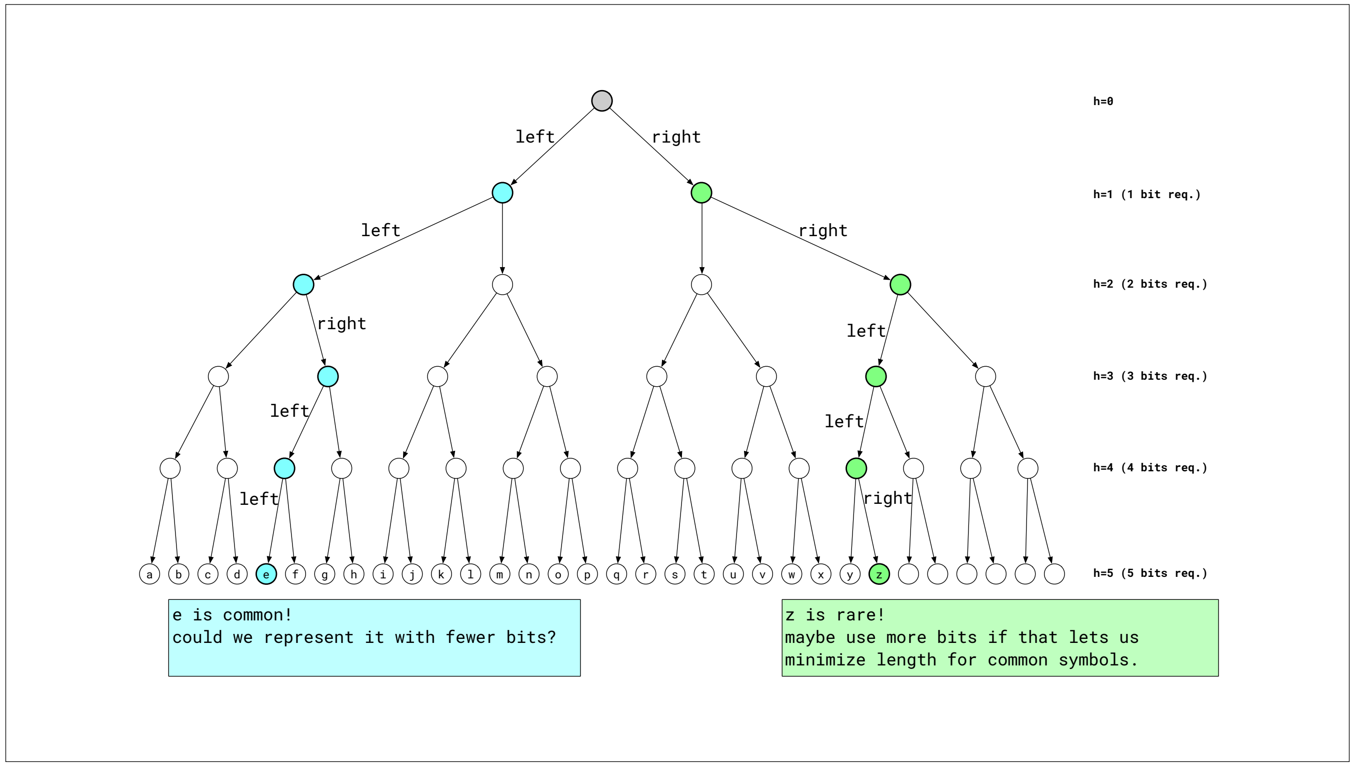


path to e: left, left, right, left, left
substitute 0 for left, 1 for right
binary: 00100

path to z: right, right, left, left, right
substitute 0 for left, 1 for right
binary: 11001

Using this tree, following the path from the root down to E, we go left, left, right, left, and then left. If you substitute 0 for left, and 1 for right, that gives us a bitstring 0 0 1 0 0. So that's five bits to encode an E in this scheme.

Similarly, you can follow the path for the letter Z, or Zed if you're into that sort of thing, and that also gives you a five-bit encoding.

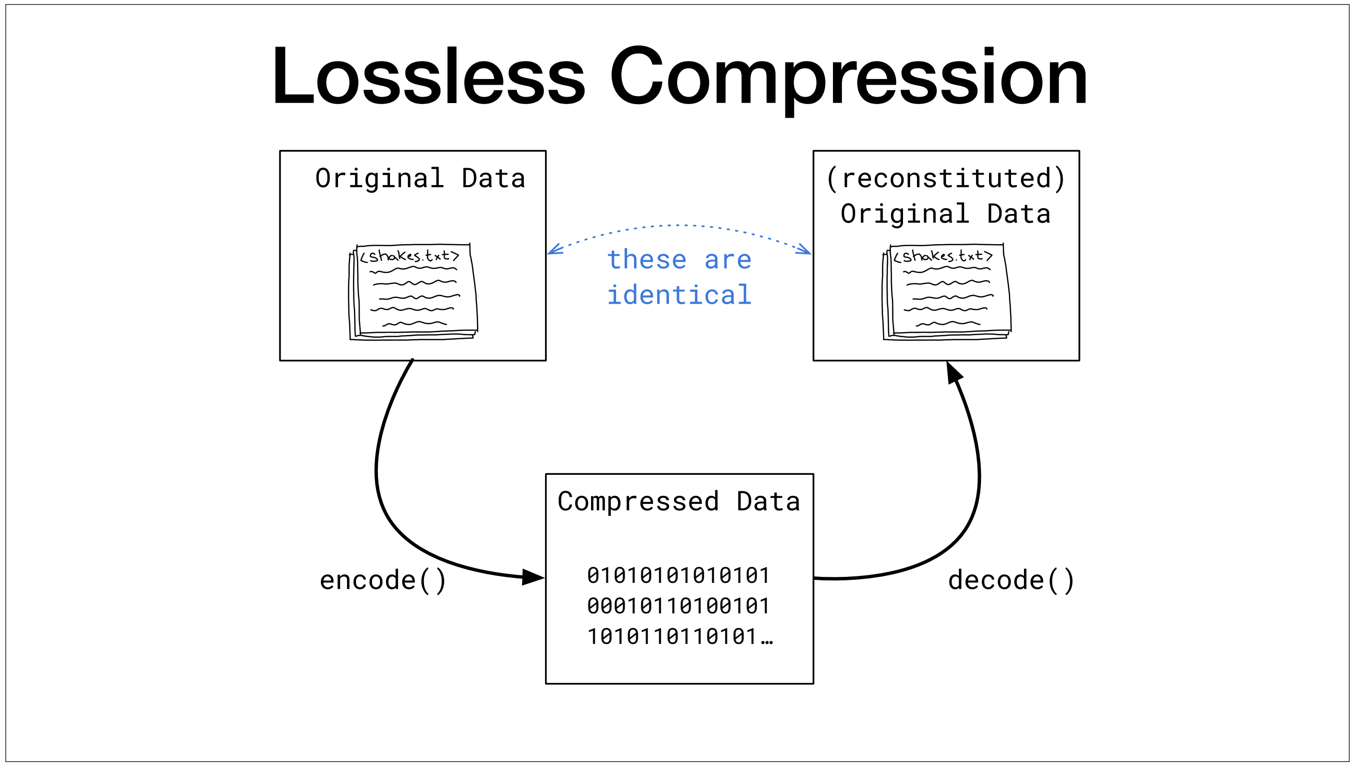


e is common!
could we represent it with fewer bits?

z is rare!
maybe use more bits if that lets us
minimize length for common symbols.

But remember that E is 300 times more common than Z. It is a shame that it takes the same number of bits.

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.



Lossless Compression

Original Data

these are identical

(reconstituted) Original Data

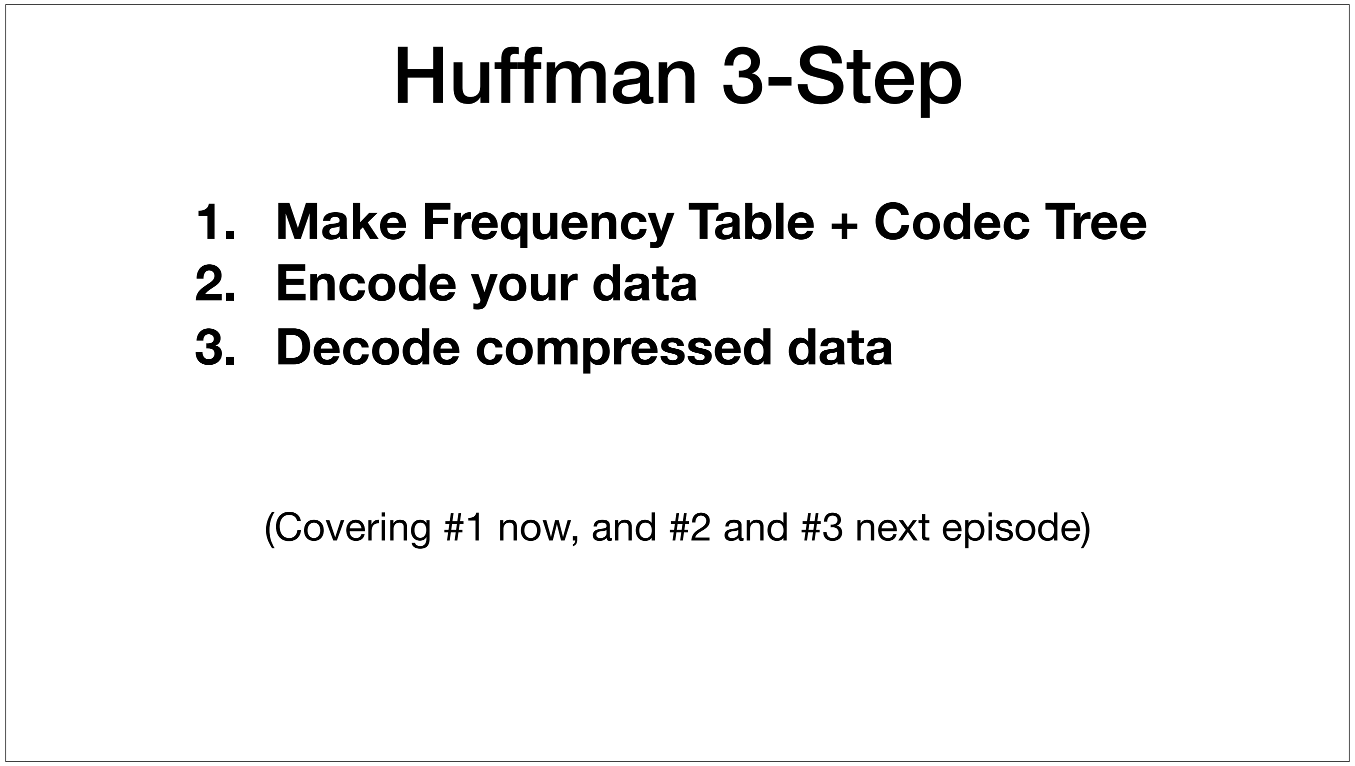
encode()

Compressed Data

01010101010101
00010110100101
1010110110101…

decode()

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



Huffman 3-Step

1. **Make Frequency Table + Codec Tree**
2. **Encode your data**
3. **Decode compressed data**

(Covering #1 now, and #2 and #3 next episode)

There are three main steps to take for the Huffman encoding strategy.

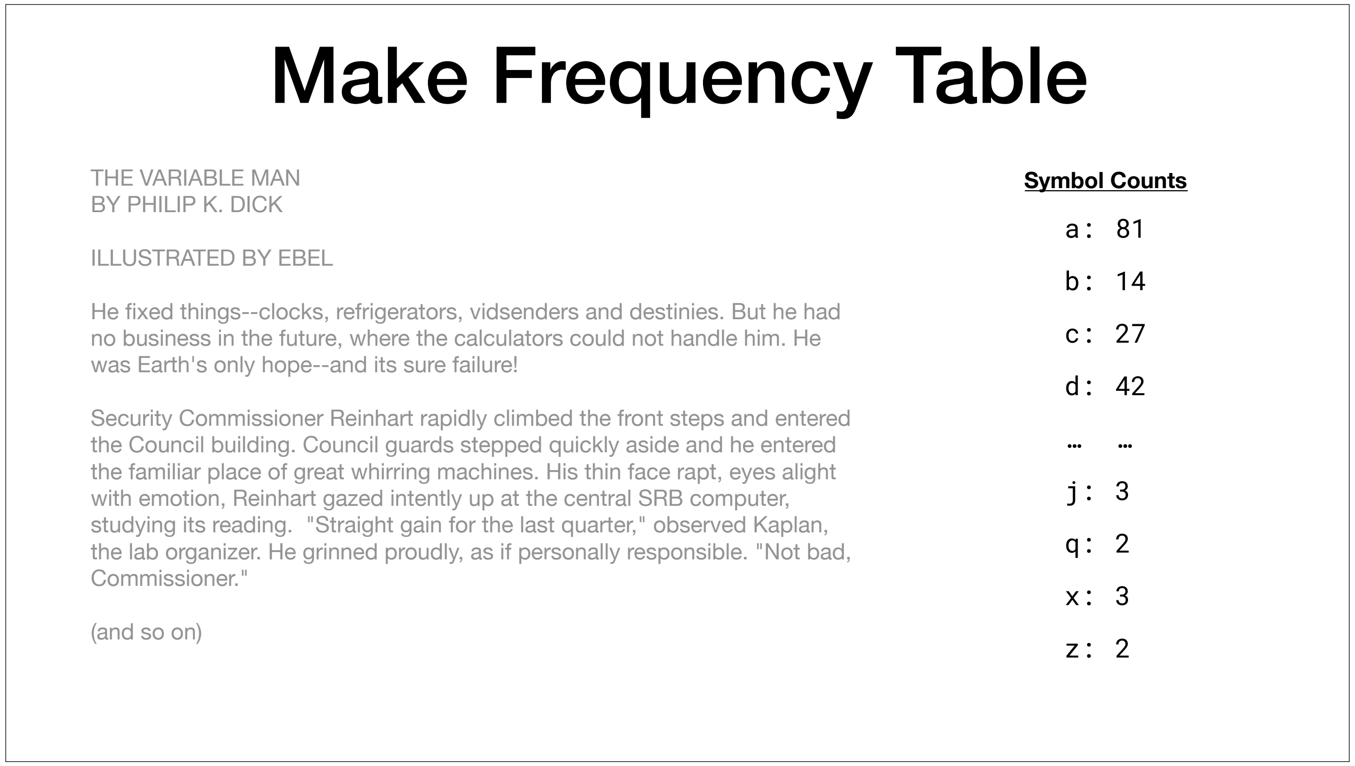
First, you'll use some corpus of information to create a frequency table and a codec lookup tree.

Second, use the frequency table to encode whatever data you have. This gives you a compressed version of the data.

Third, use the same frequency table to decode the compressed version, which should give you a replica of the original data.

<next>

We'll spend the rest of this episode going over the frequency tree / codec lookup tree step, and the next episode going over the encoding and decoding steps.



Make Frequency Table

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!

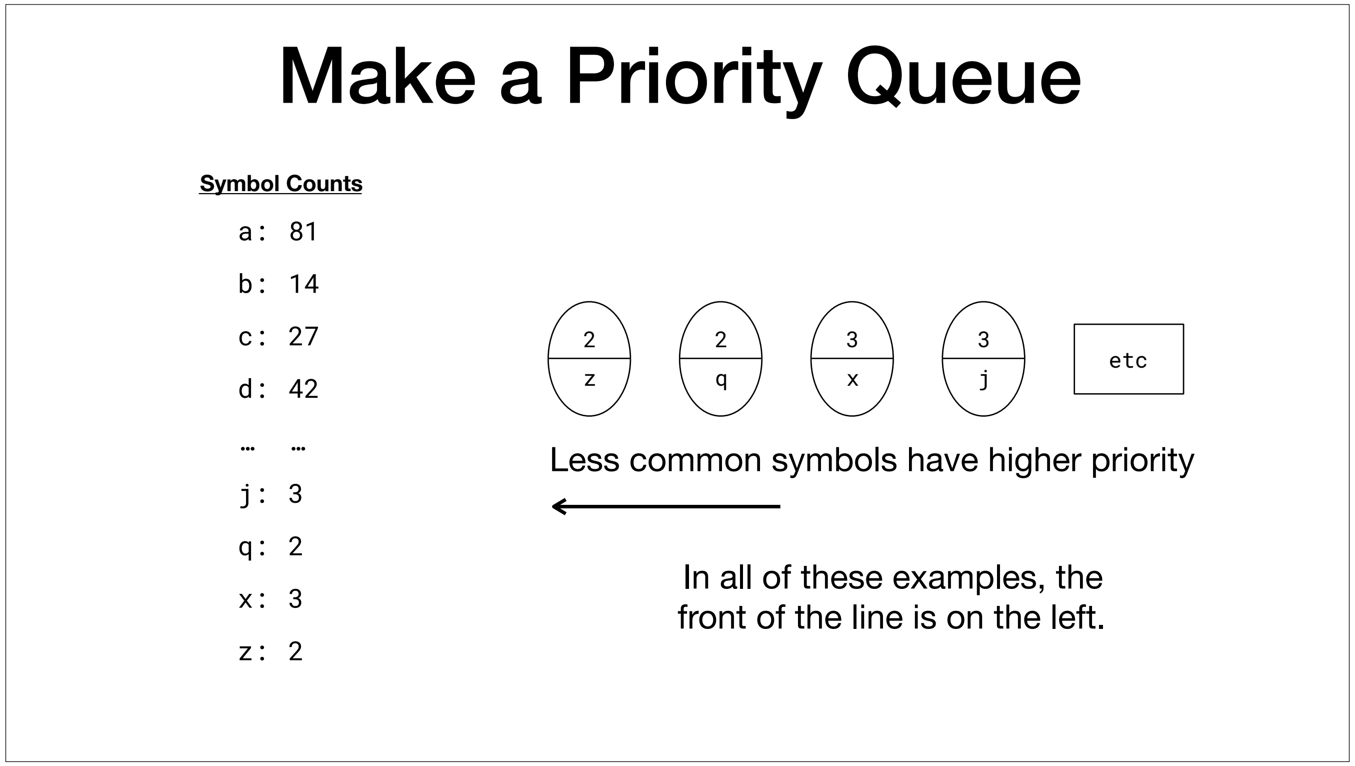
Security Commissioner Reinhart rapidly climbed the front steps and entered the Council building. Council guards stepped quickly aside and he entered the familiar place of great whirring machines. His thin face rapt, eyes alight with emotion, Reinhart gazed intently up at the central SRB computer, studying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

a:	81
b:	14
c:	27
d:	42
…	…
j:	3
q:	2
x:	3
z:	2

To do this, scan a corpus of text (or other symbolic data) and create a map of characters with how often they appear. You can do this as a frequency (like 10%) or a count. I've done it using a symbol count. Either way works, you just want the numbers to be larger for more common symbols.



Make a Priority Queue

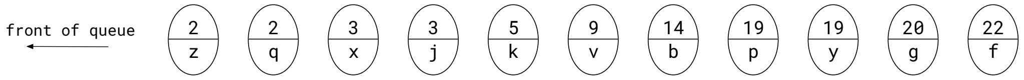
Symbol Counts

a:	81
b:	14
c:	27
d:	42
…	…
j:	3
q:	2
x:	3
z:	2

Less common symbols have higher priority

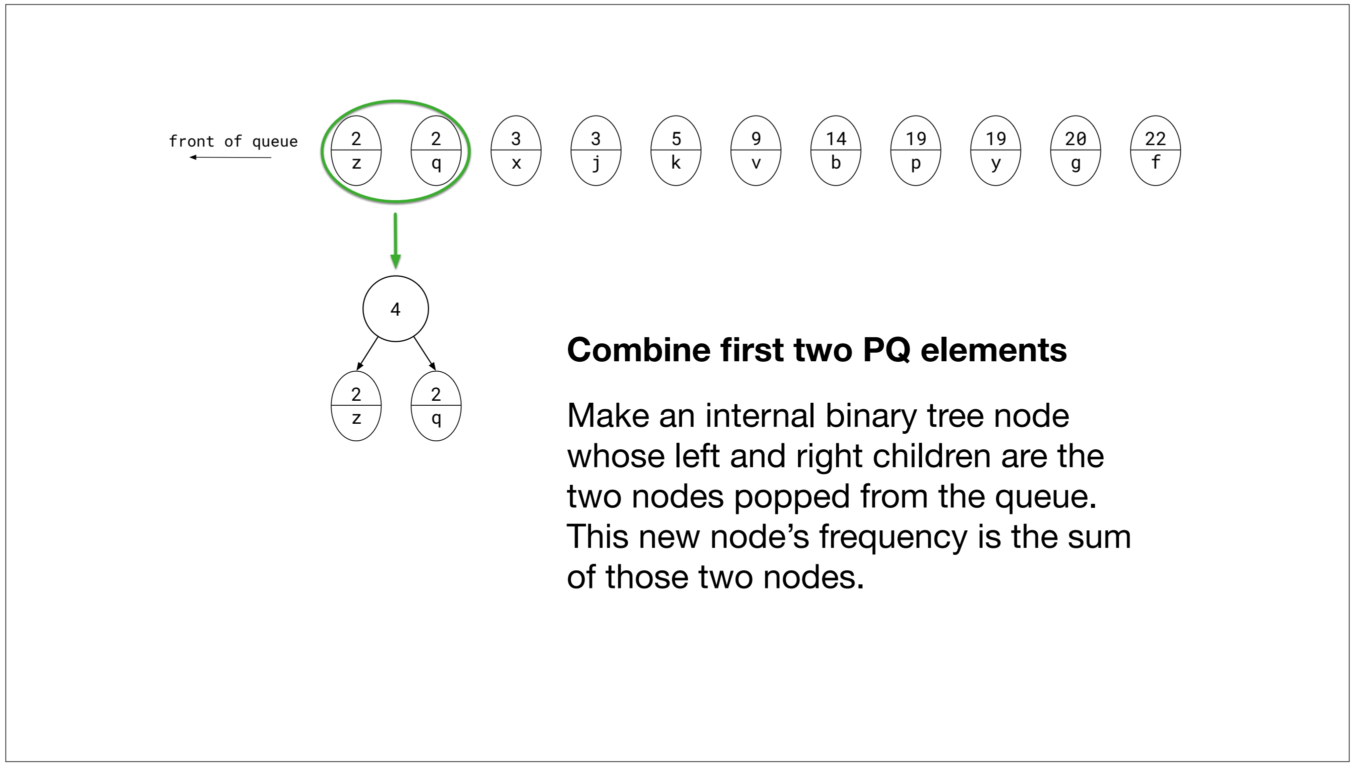
In all of these examples, the front of the line is on the left.

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. In all of these examples the front of the queue is on the left.



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.

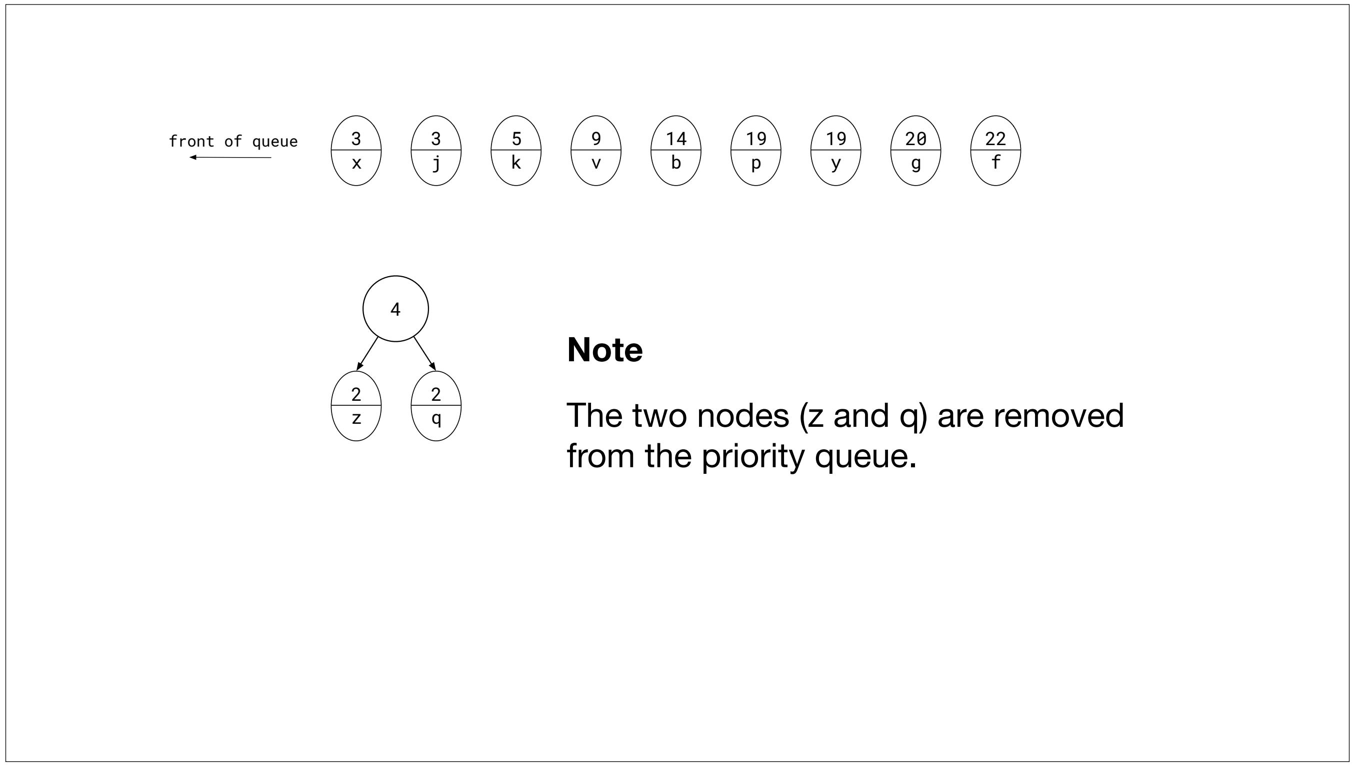
We'll pop two items from the priority queue, combine them to form an internal node, and put that node back in the priority queue.



Combine first two PQ elements

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

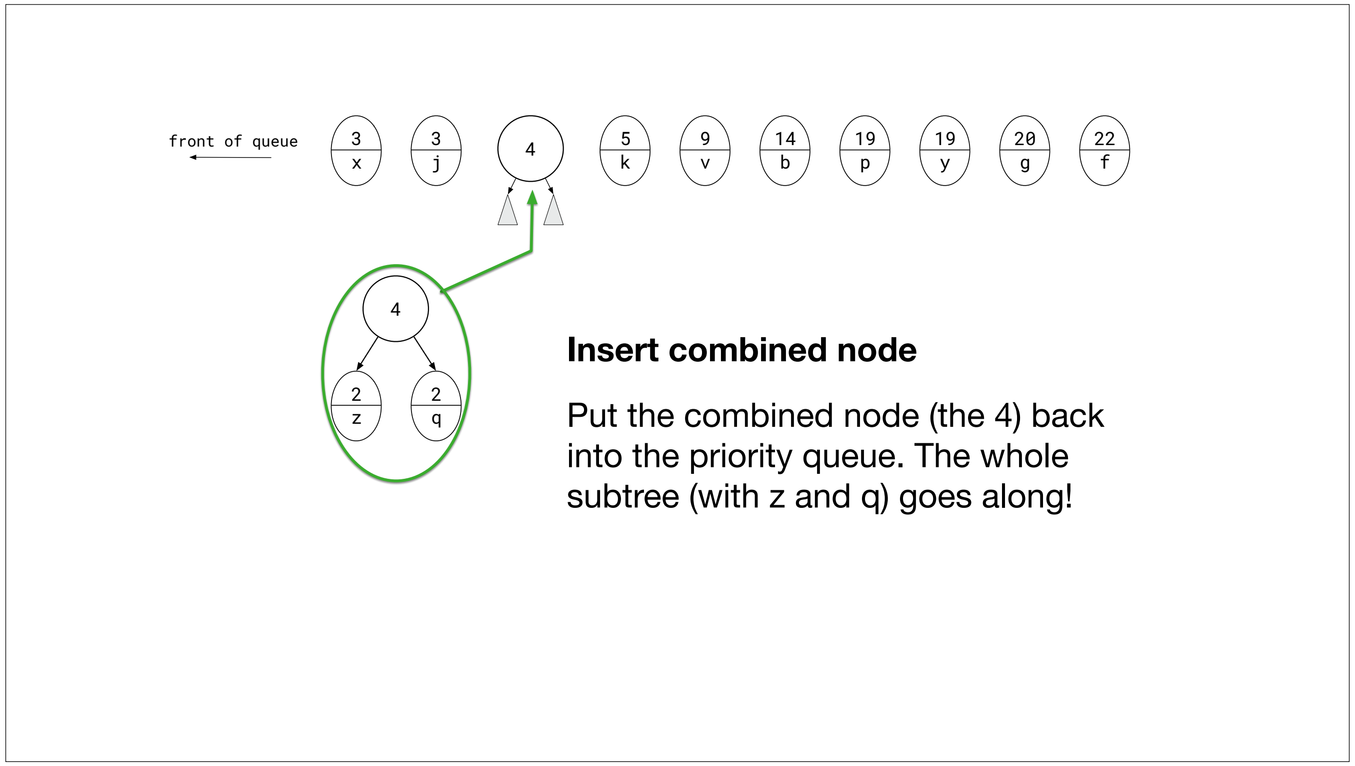
Make an internal binary tree node whose left and right children are the two nodes popped from the queue. The new node's frequency is the sum of those nodes.



Note

The two nodes (z and q) are removed from the priority queue.

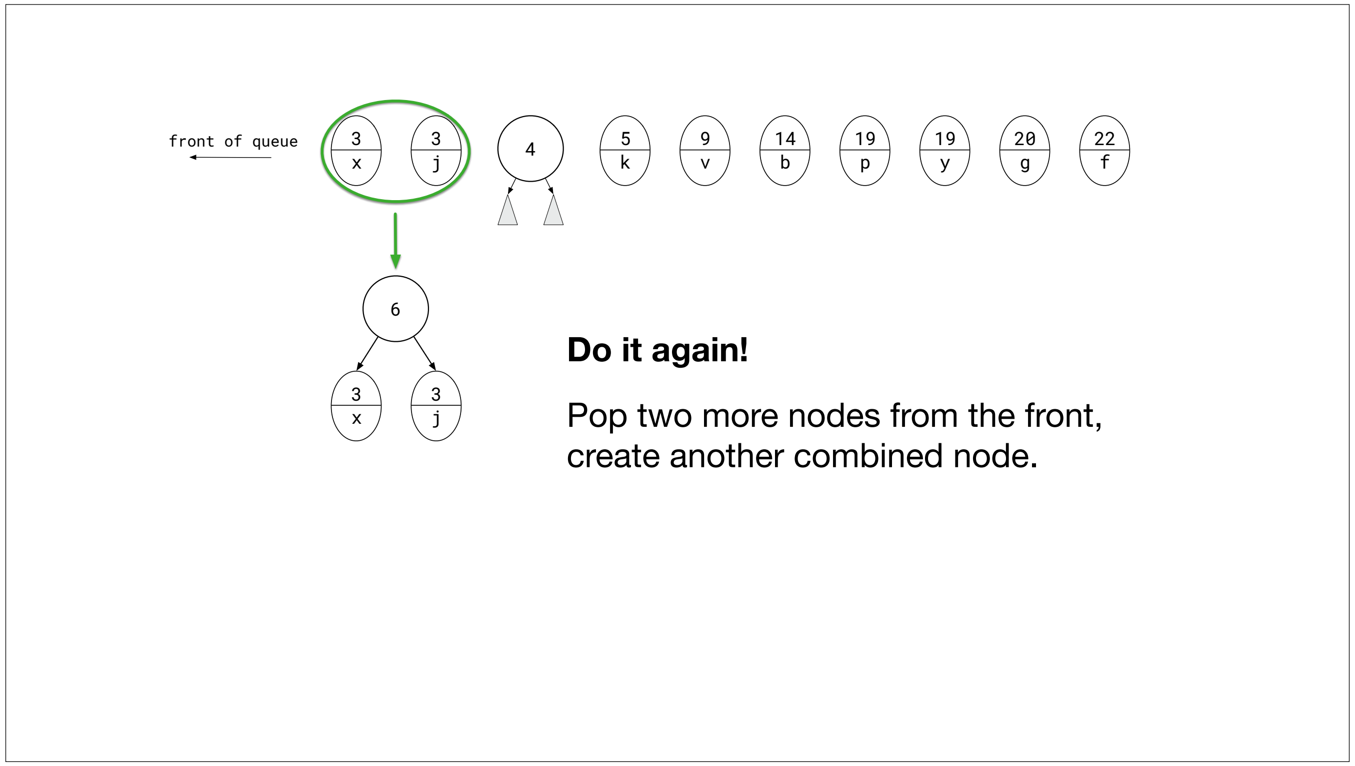
Note that the two nodes we popped are no longer in the priority queue.



Insert combined node

Put the combined node (the 4) back into the priority queue. The whole subtree (with z and q) goes along!

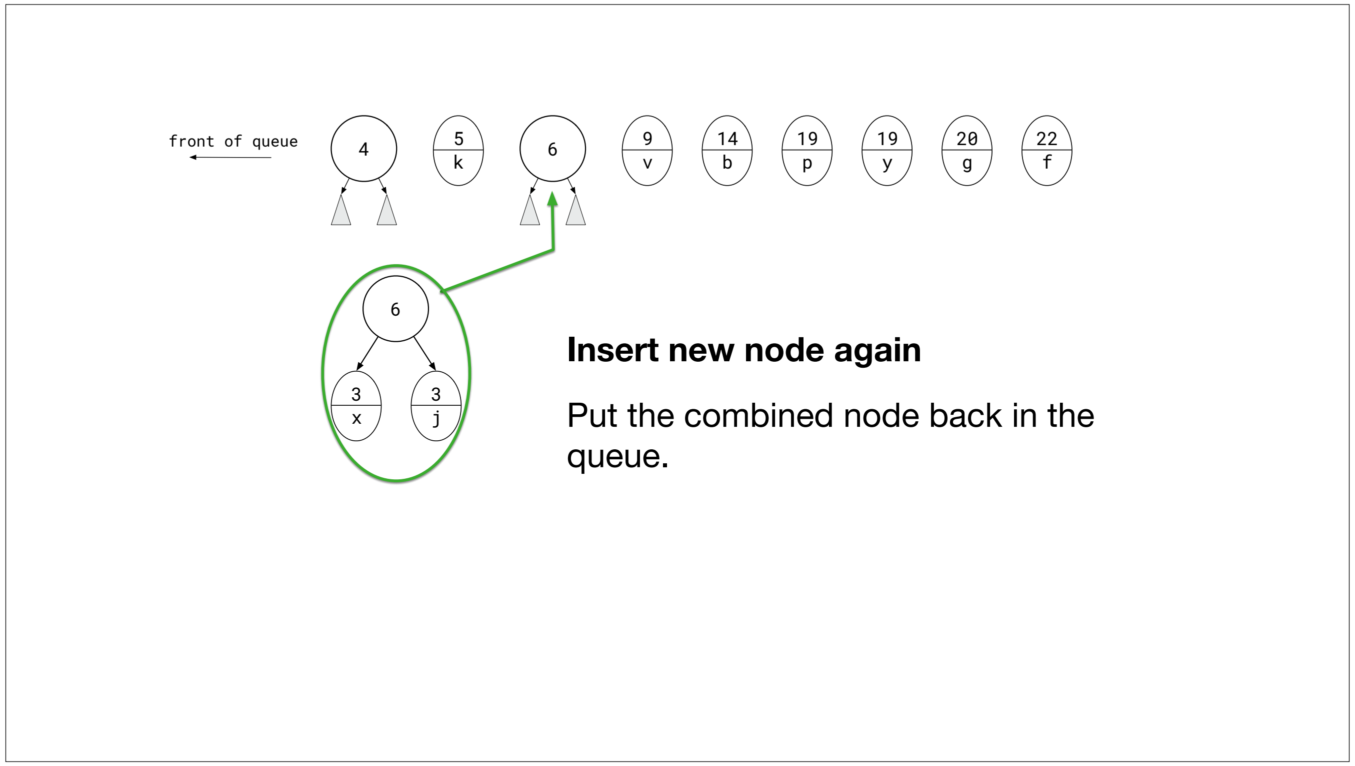
Now insert the newly created node back into the priority queue, using its frequency (4) to put it in the right spot.



Do it again!

Pop two more nodes from the front,
create another combined node.

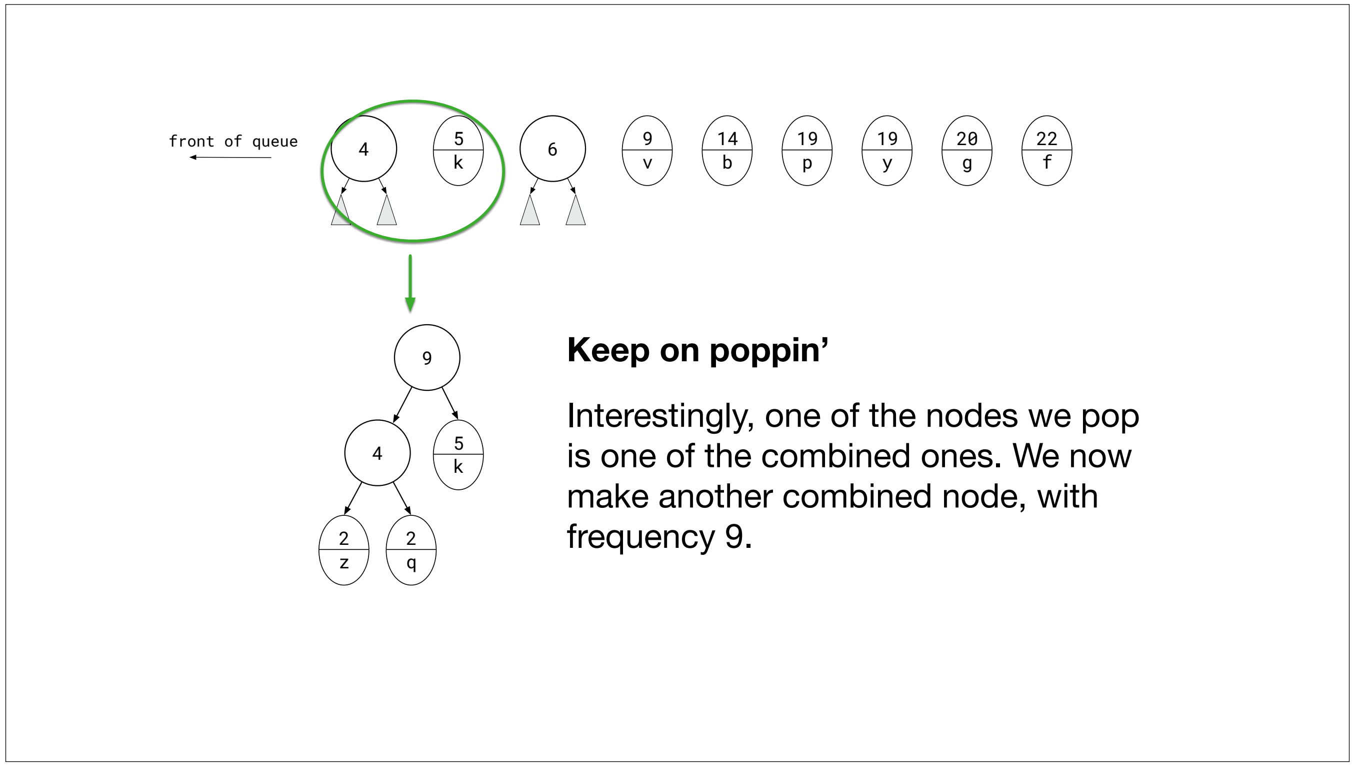
Now we pop two more nodes, make another internal node…



Insert new node again

Put the combined node back in the queue.

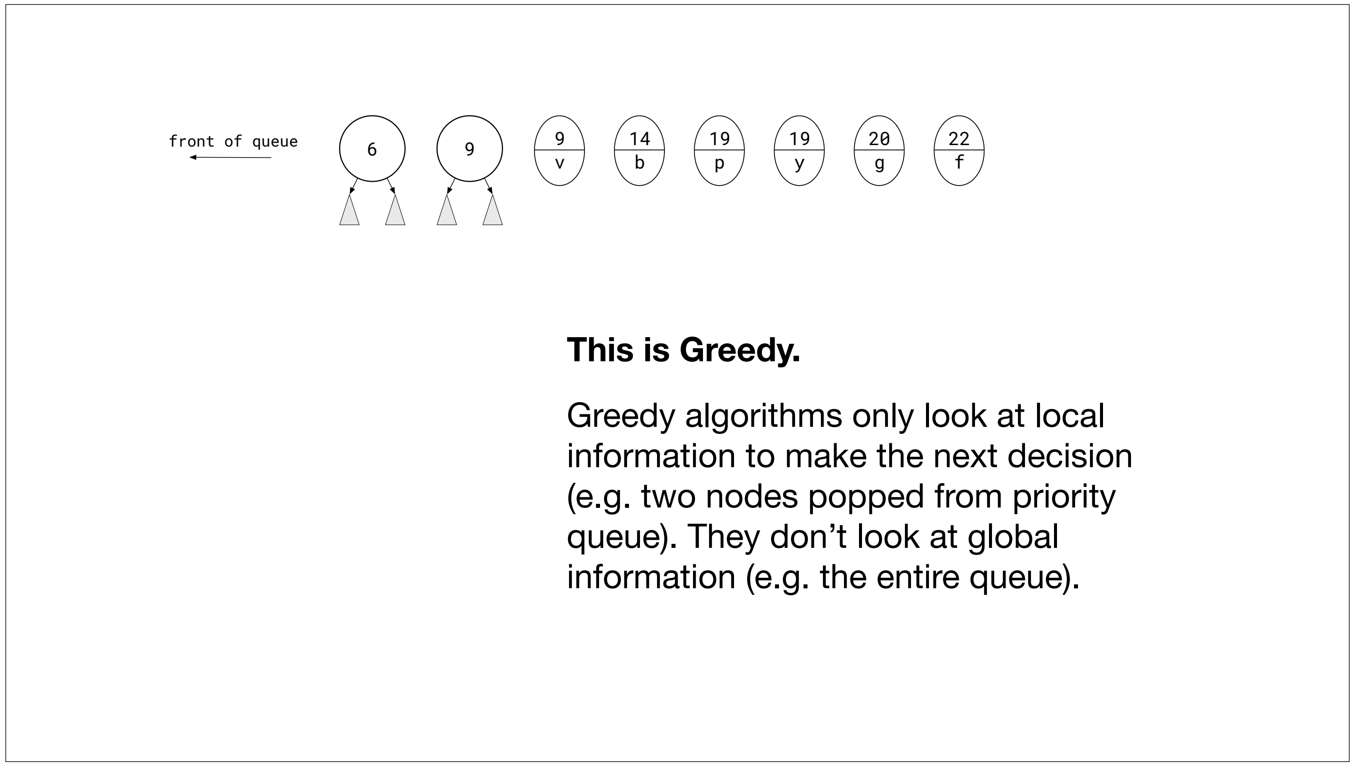
… and put it back in the queue.



Keep on poppin'

Interestingly, one of the nodes we pop is one of the combined ones. We now make another combined node, with frequency 9.

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

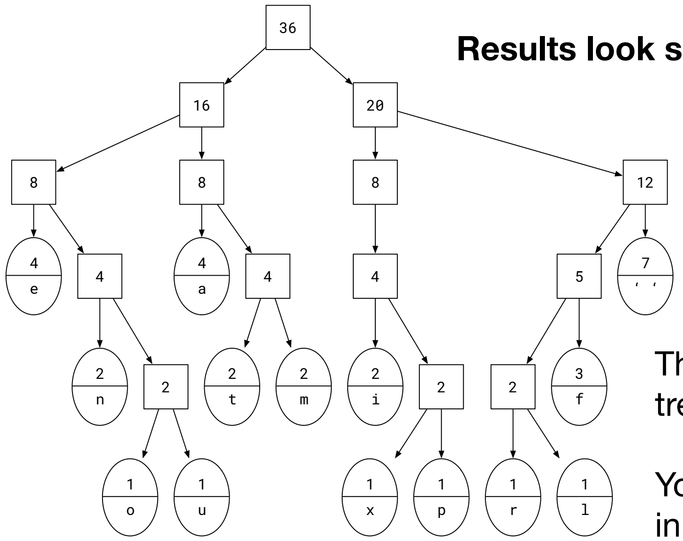


This is Greedy.

Greedy algorithms only look at local information to make the next decision (e.g. two nodes popped from priority queue). They don’t look at global information (e.g. the entire queue).

You can see now that we're starting to build one big tree by combining the two nodes with the lowest frequency on each step and reinserting the combined node back into the priority queue.

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.

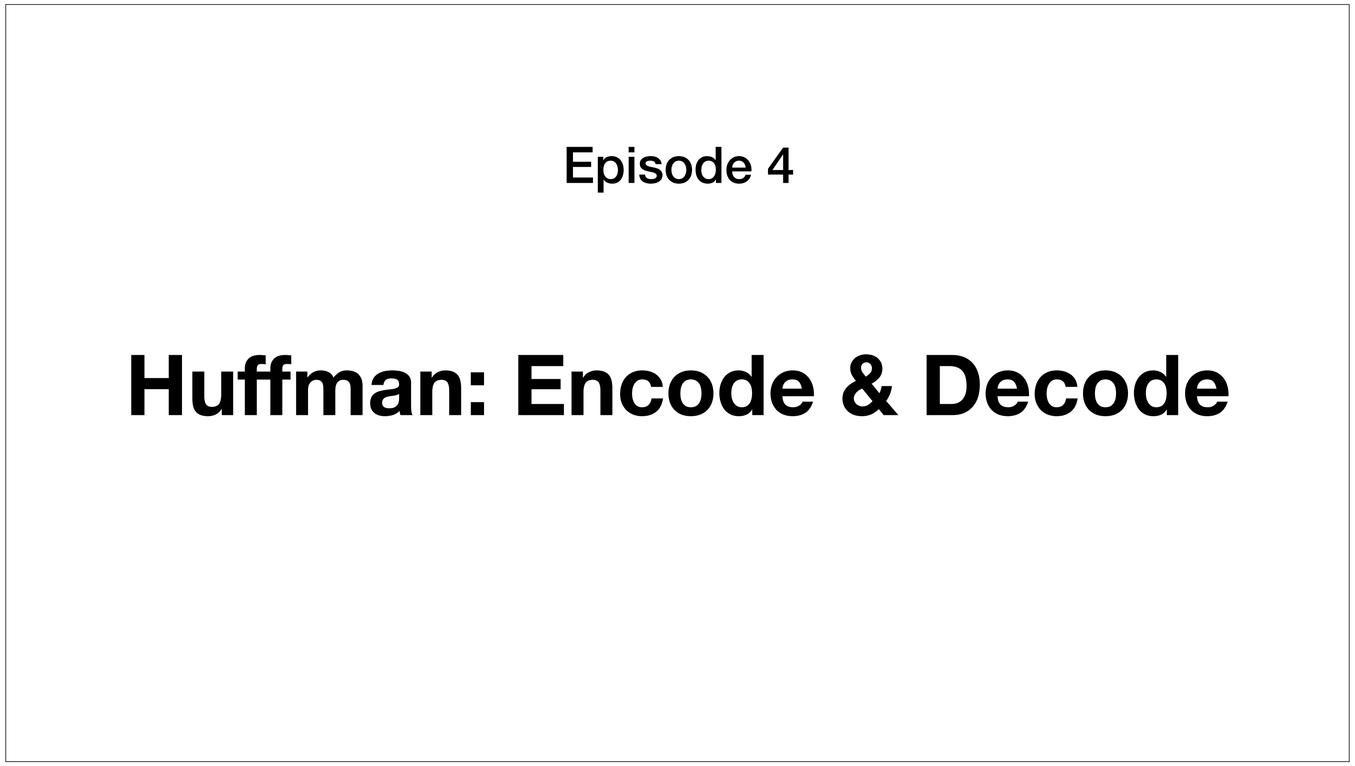


Results look something like this.

This is different from the tree we were building.

You'll see this tree more in the next episode!

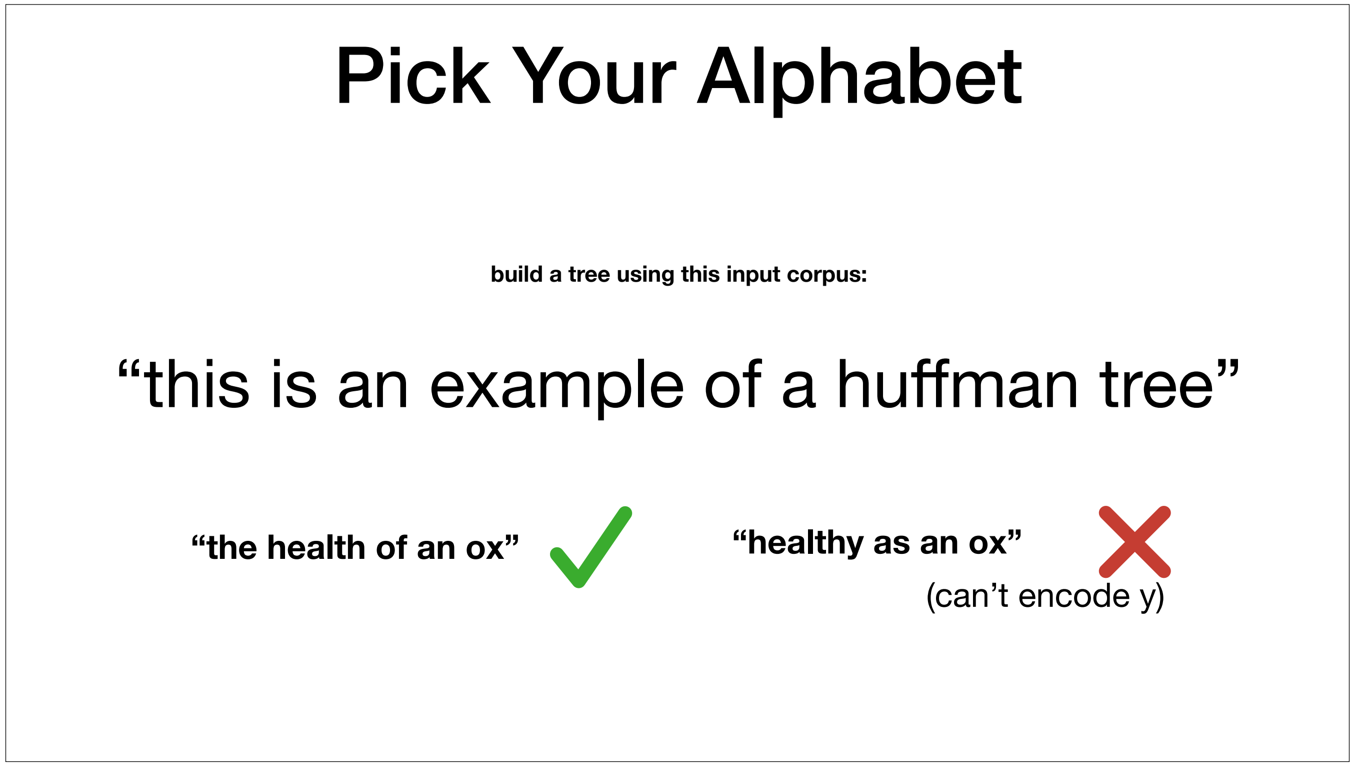
The resulting tree has this general appearance. This is actually a different tree from what we were building. You'll see this tree a lot in the next episode.



Episode 4

Huffman: Encode & Decode

Now we’re going to use our Huffman tree to encode and decode messages. We’ll take a little detour first, to set things up, and to talk about the properties that a Huffman tree has.



Pick Your Alphabet

build a tree using this input corpus:

“this is an example of a huffman tree”

“the health of an ox” ✓

“healthy as an ox” ✗

(can’t encode y)

When you want to encode or decode messages using Huffman codes, it is critically important that you use an encoding scheme that actually contains all the symbols you want to encode and decode. So in this example we're going to build a Huffman tree using this sentence as the corpus:

"This is an example of a huffman tree"

This is somewhat shorter than the collected works of Shakespeare, and as you can see, many letters are left out of the encoding entirely. So if we are going to use this corpus, we can only encode and decode message that use the letters you see here.

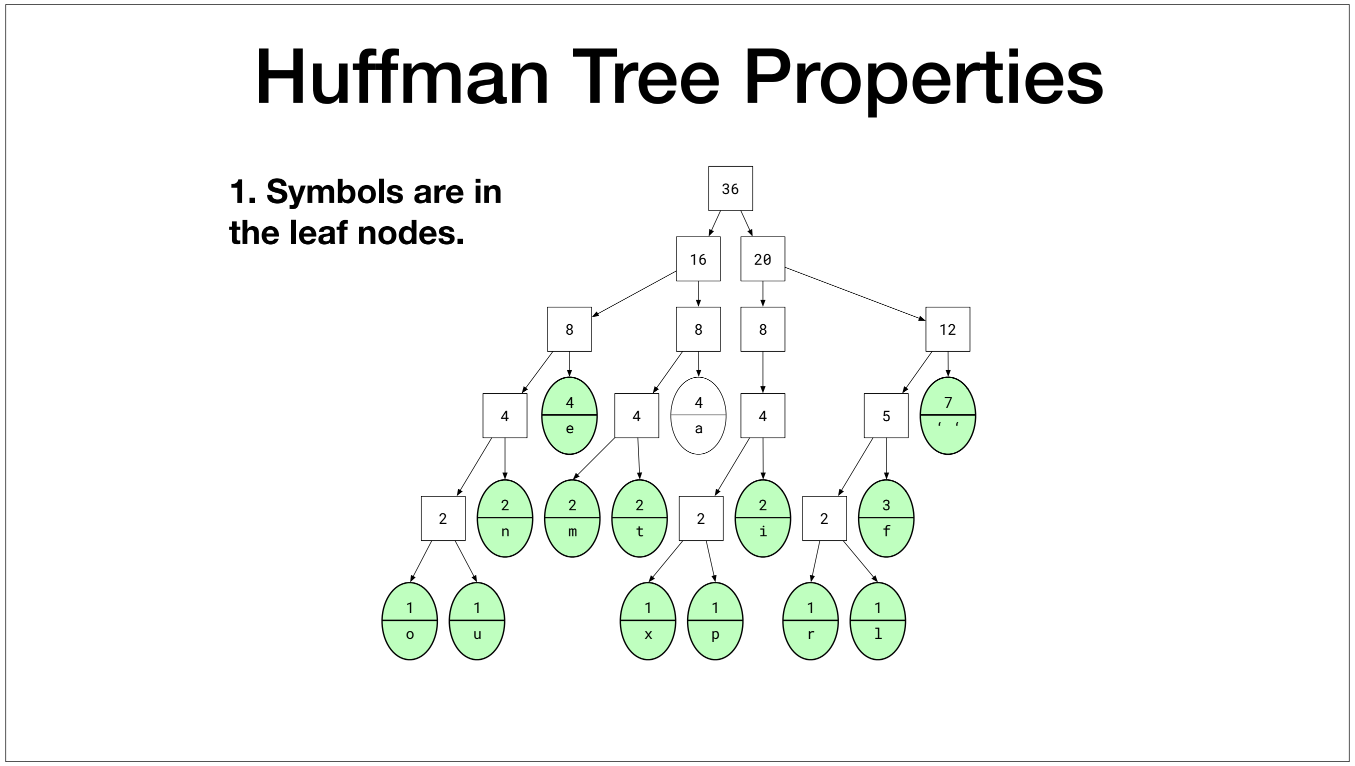
We can encode the sentence <next>

"the health of an ox”

but not <next>

"healthy as an ox"

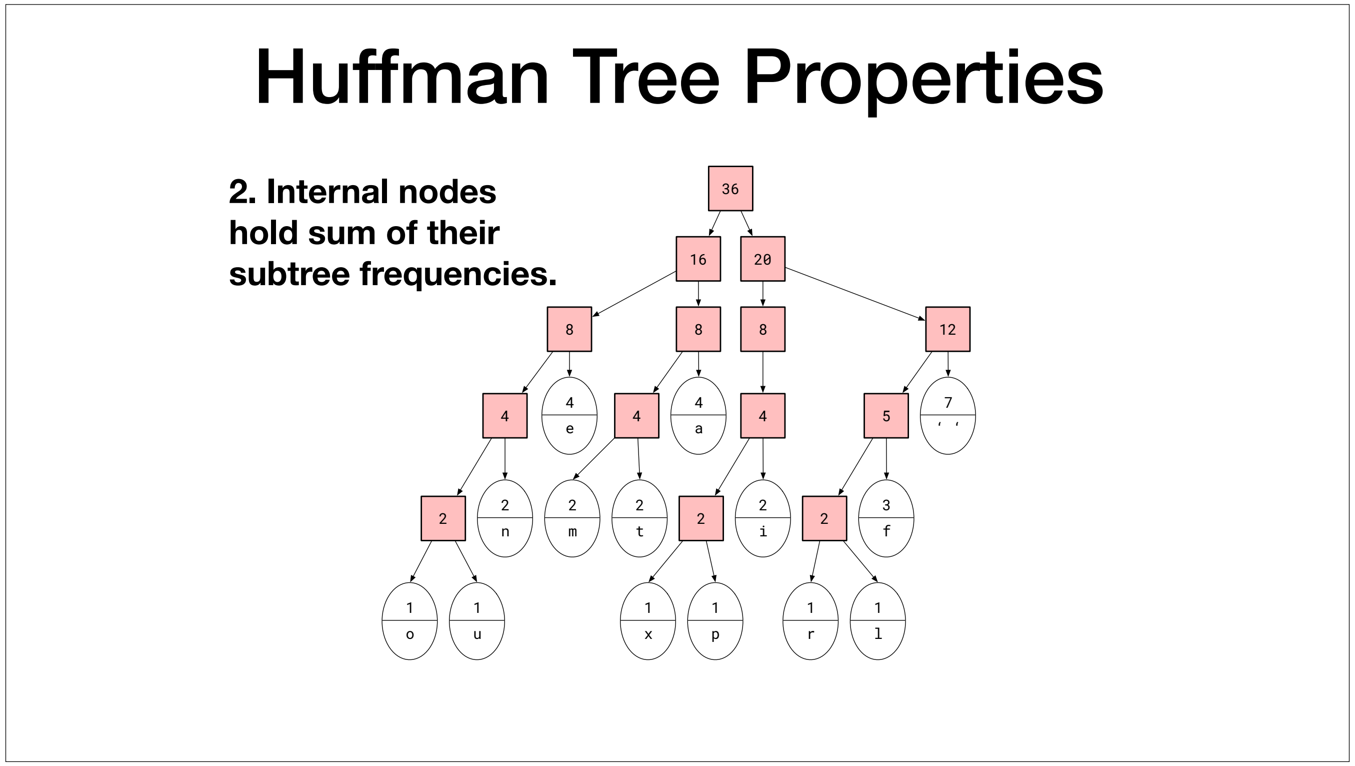
because there is not a Y in the original corpus.



Huffman Tree Properties

1. Symbols are in the leaf nodes.

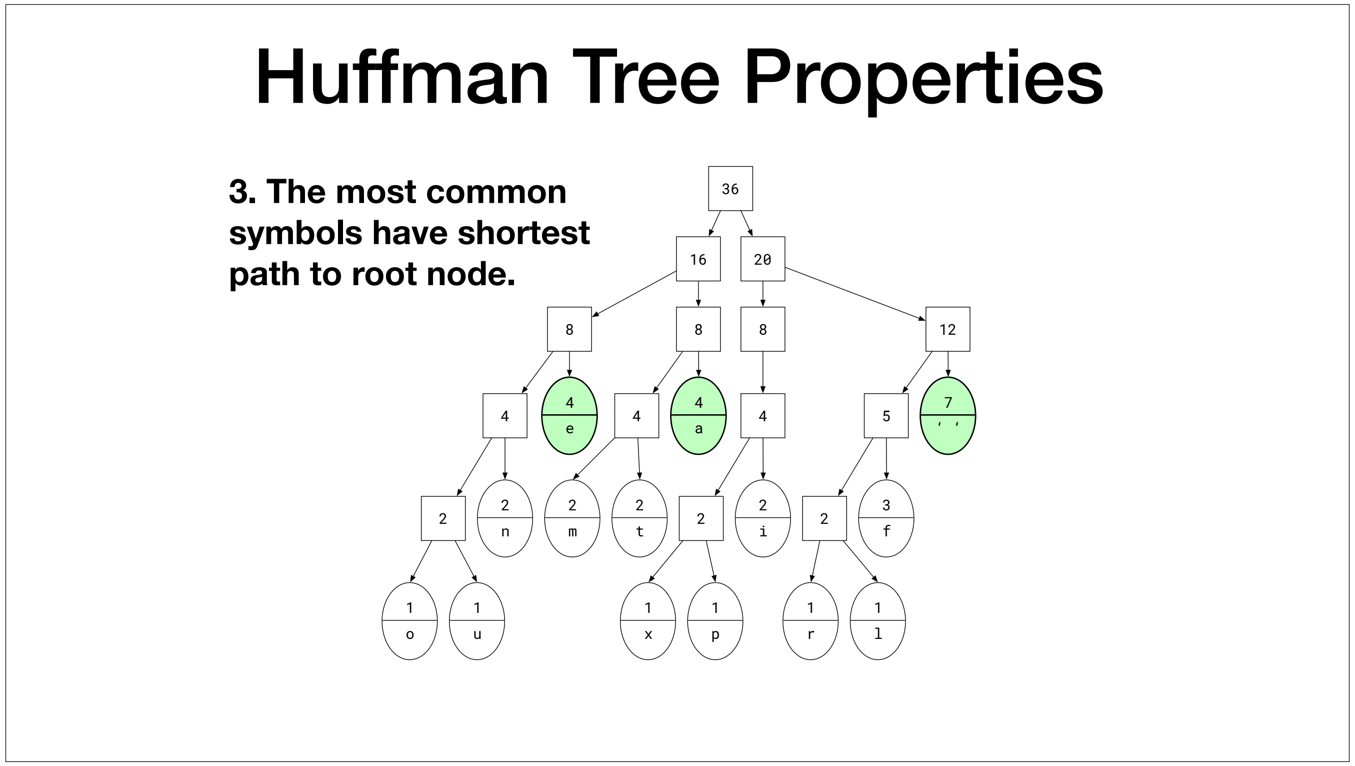
Huffman trees have some interesting properties that we'll make use of. First, all Leaf nodes have symbol data.



Huffman Tree Properties

2. Internal nodes hold sum of their subtree frequencies.

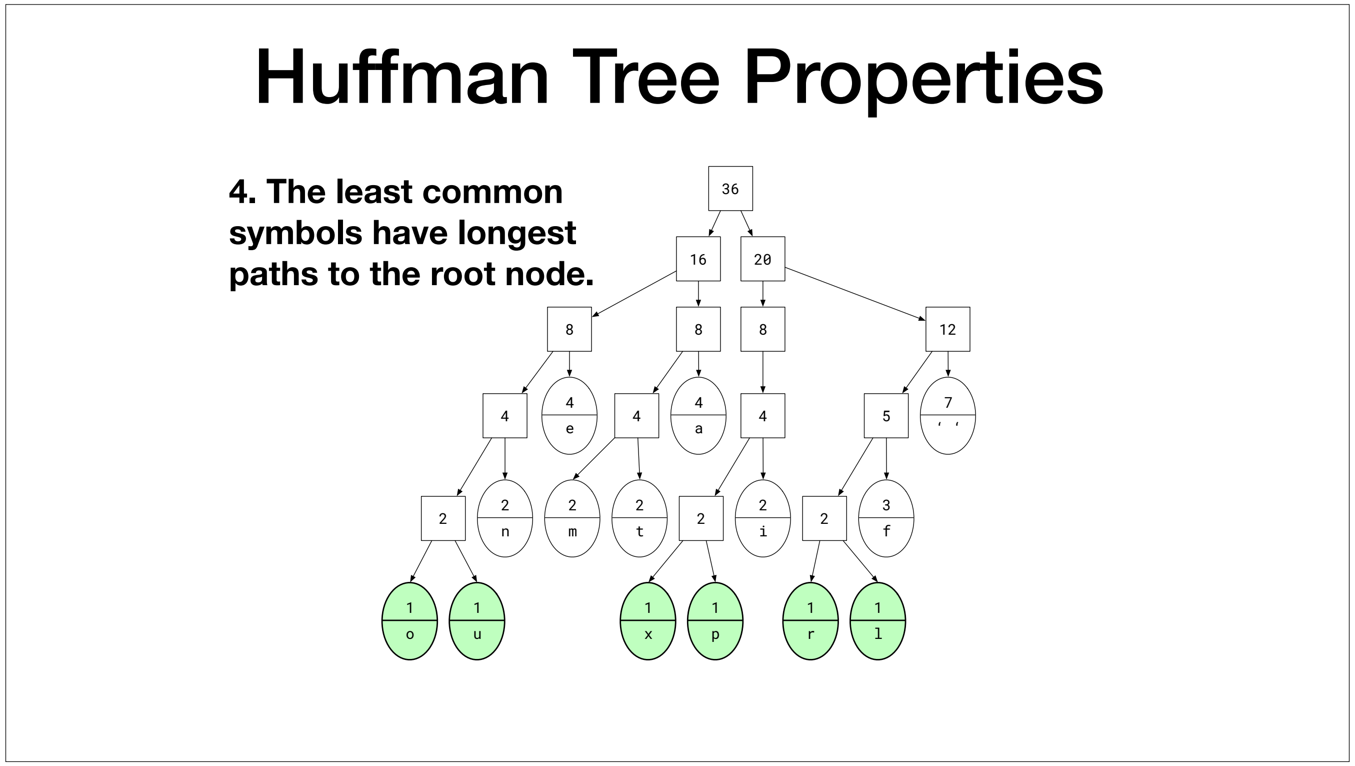
Second, all internal nodes store the sum of their child node frequencies.



Huffman Tree Properties

3. The most common symbols have shortest path to root node.

The most frequent symbols have the shortest path to the root. This is the main property that lets us compress data effectively.

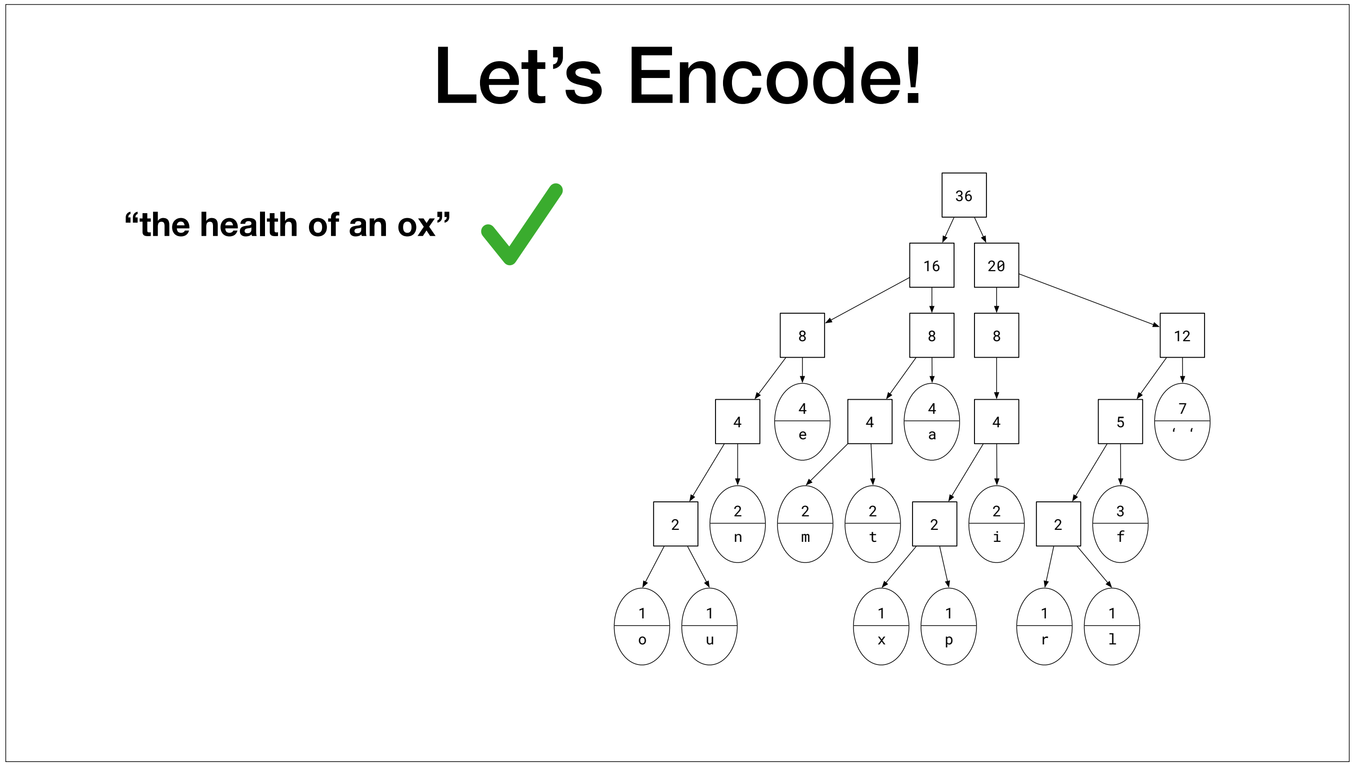


Huffman Tree Properties

4. The least common symbols have longest paths to the root node.

And the converse of that is that the least frequent symbols have the longest path to the root.

If we encoded sentences that involved lots of O’s and X’s and R’s, this particular huffman tree would not be super efficient.



Let’s Encode!

“the health of an ox” ✓

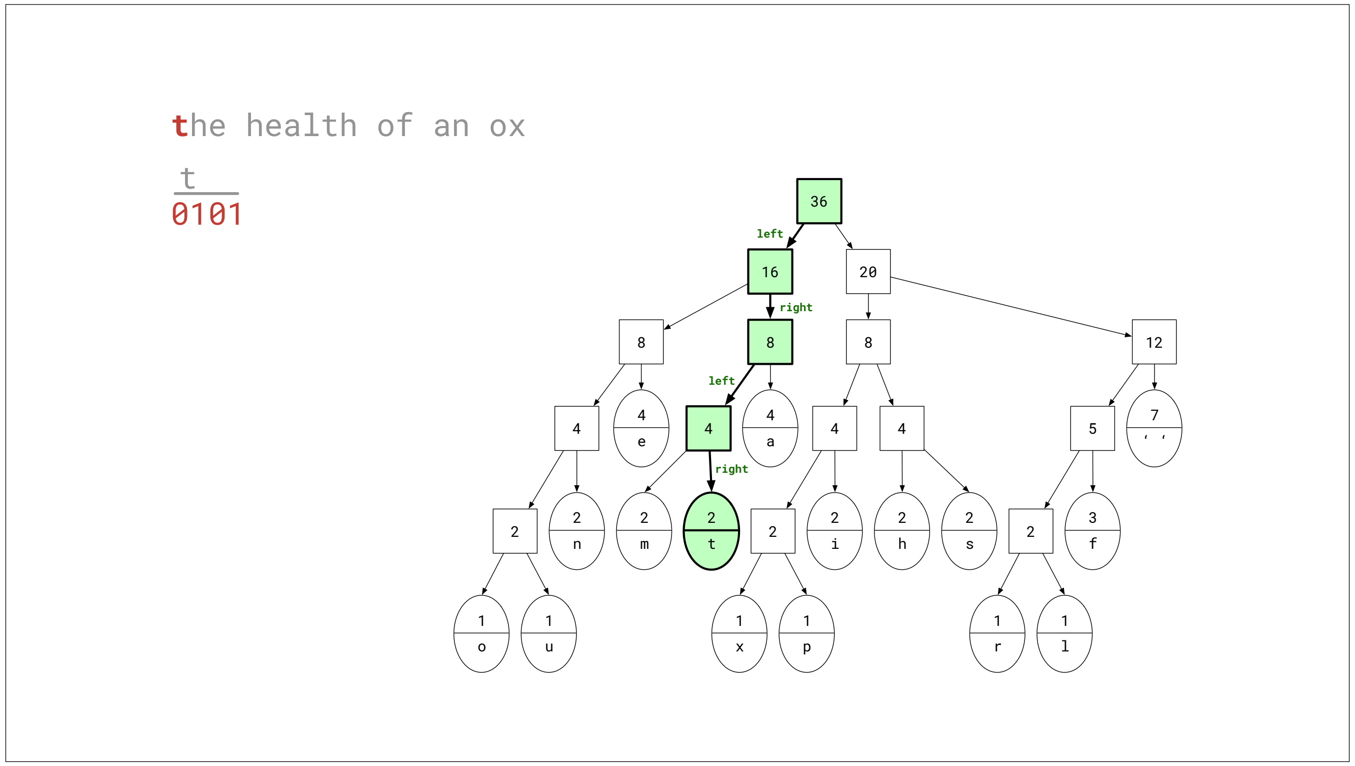
Say we want to encode the string

"the health of an ox"

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.

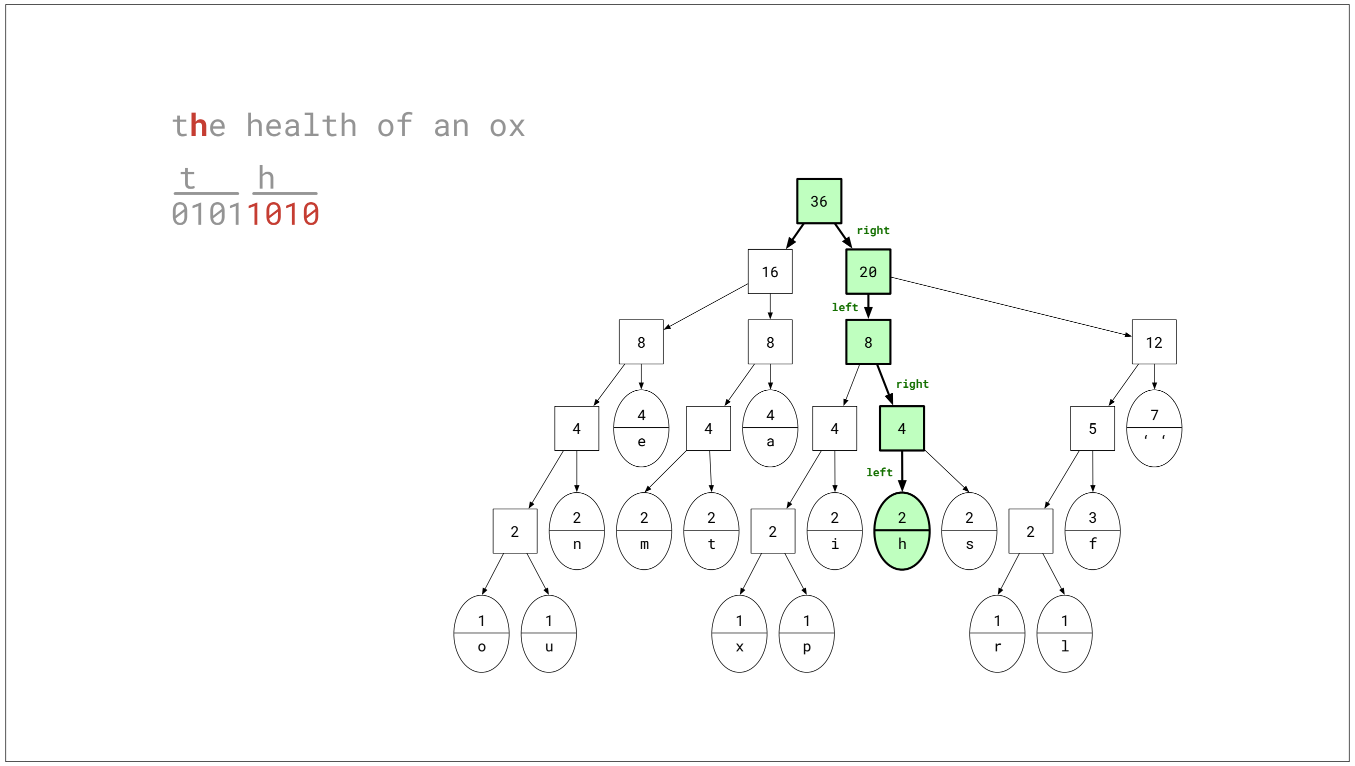


the health of an ox

t

0101

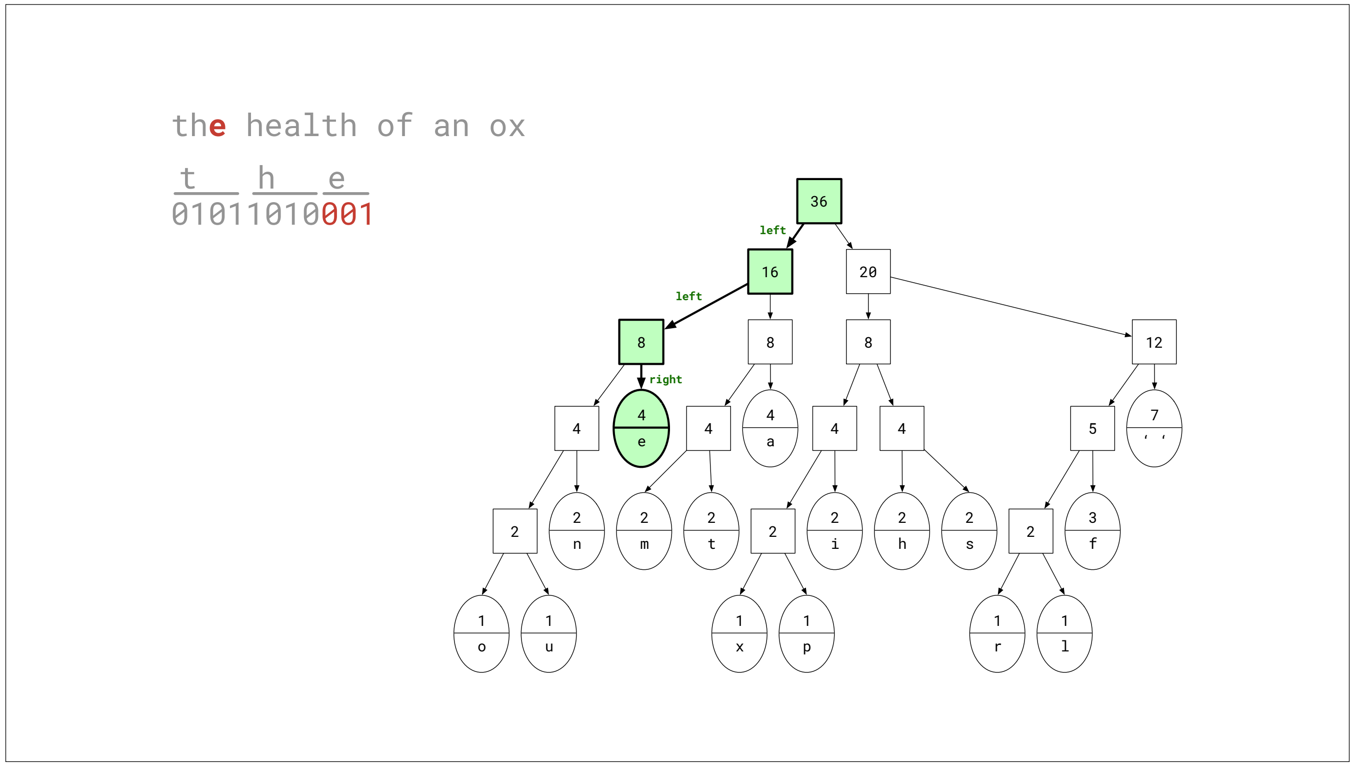
When encoding, we proceed through our input text one symbol at a time. The first symbol is T. So we identify the path from the root to our T node. Every time we go left, we emit a zero. Every time we go right we emit a one. So for T, we emit 0 1 0 1, since we go left, right, left, right.



the health of an ox

t h
01011010

Then we do it again. H is next, and we go right, left, right left, so we output 1 0 1 0.

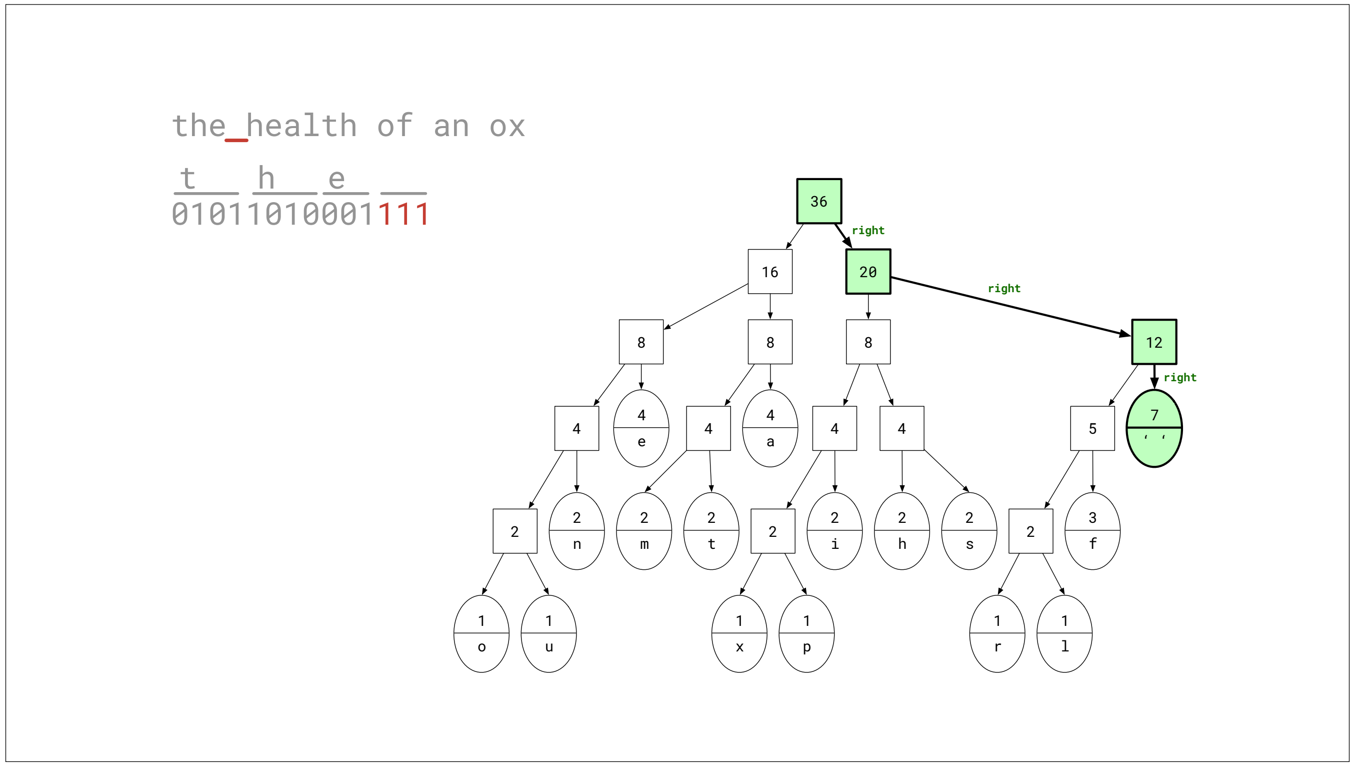


the**e** health of an ox

t h e

01011010**001**

Now we encode the E. Notice that the E is higher than the other symbols we've encoded. So it takes fewer bits to encode - 0 0 1.



the_health of an ox

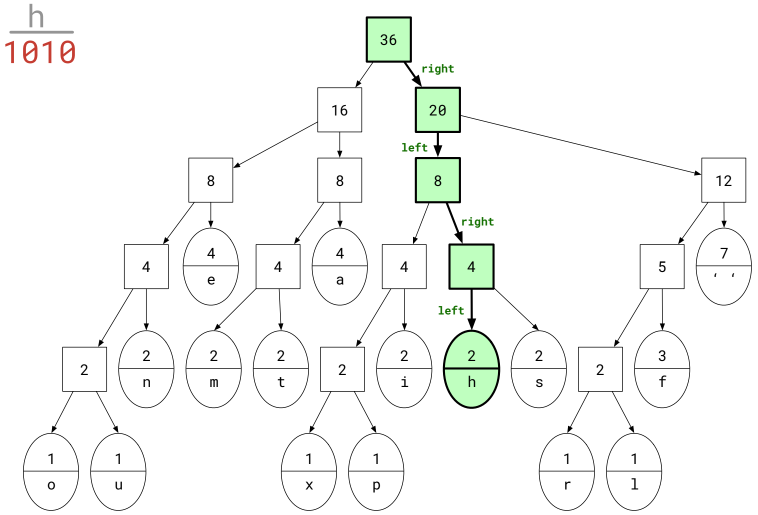
t h e

01011010001111

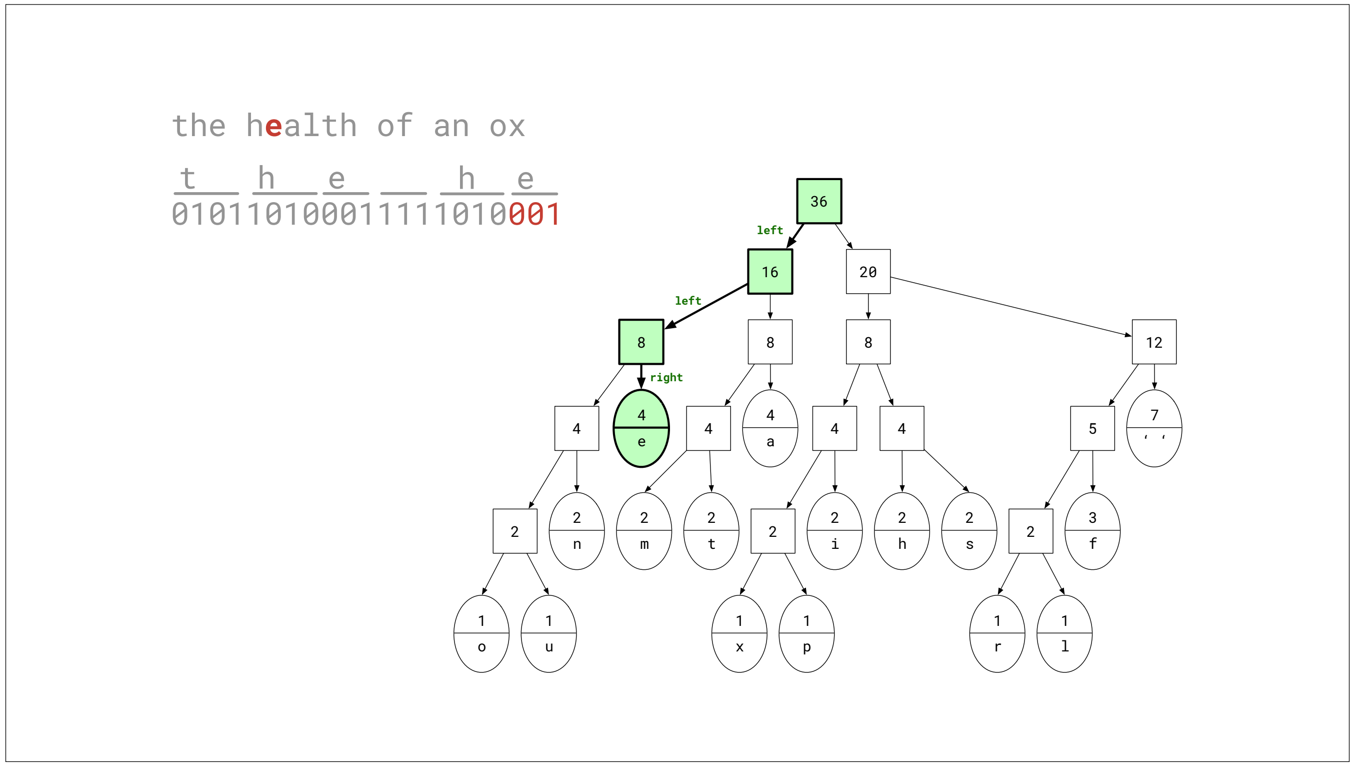
Then we encode the space character, which is actually the most frequent character, and it also only takes three bits, 1 1 1.

the **h**ealth of an ox

```
 t   h   e       h
01011010001111 1010
```



Another H, which we’ve seen before, so you can check that it is the same. 1 0 1 0.

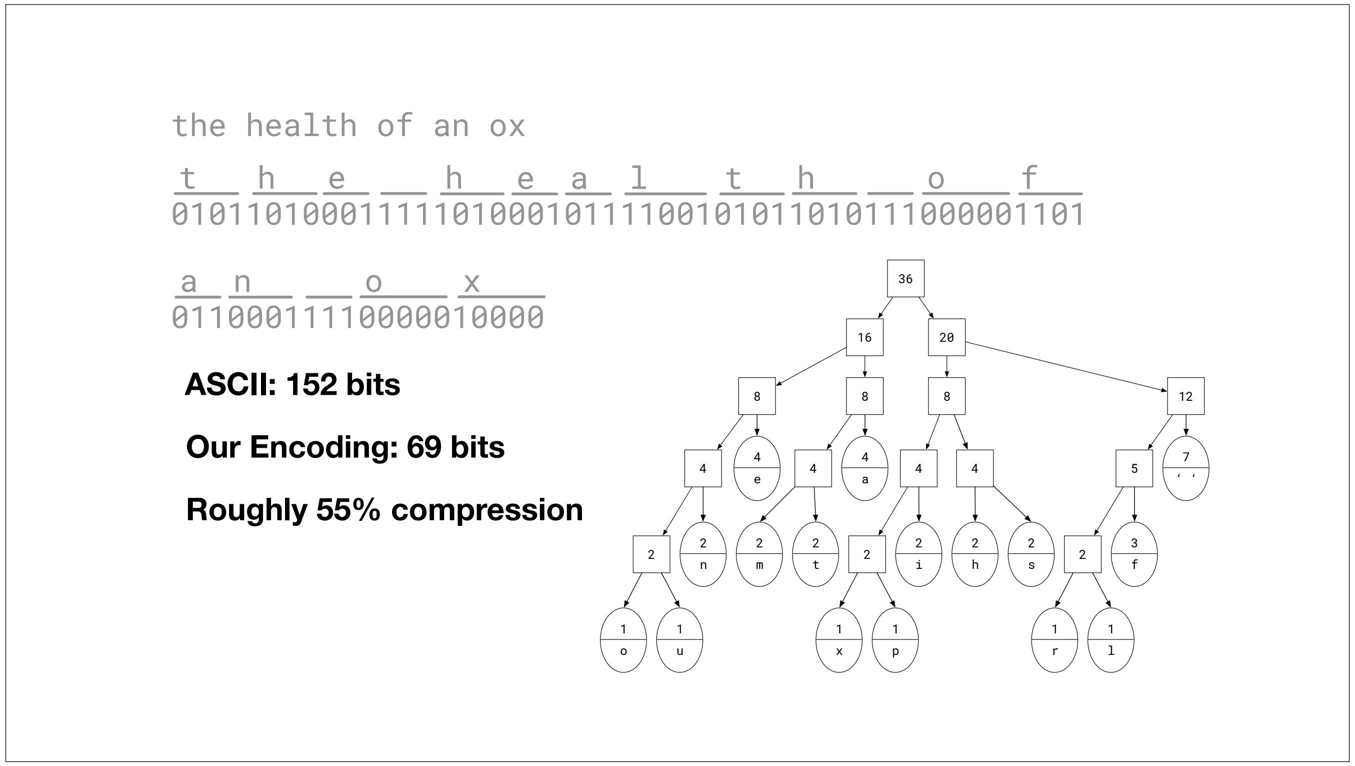


the health of an ox

t h e _ h e

010110100011111010001

And an E, 0 0 1. It keeps on going until we’ve encoded the whole input sentence.



```
the health of an ox

 t   h   e       h   e  a  l    t   h       o     f
0101101000111110100010111100101011010111000001101

 a  n      o     x
011000111100000010000
```

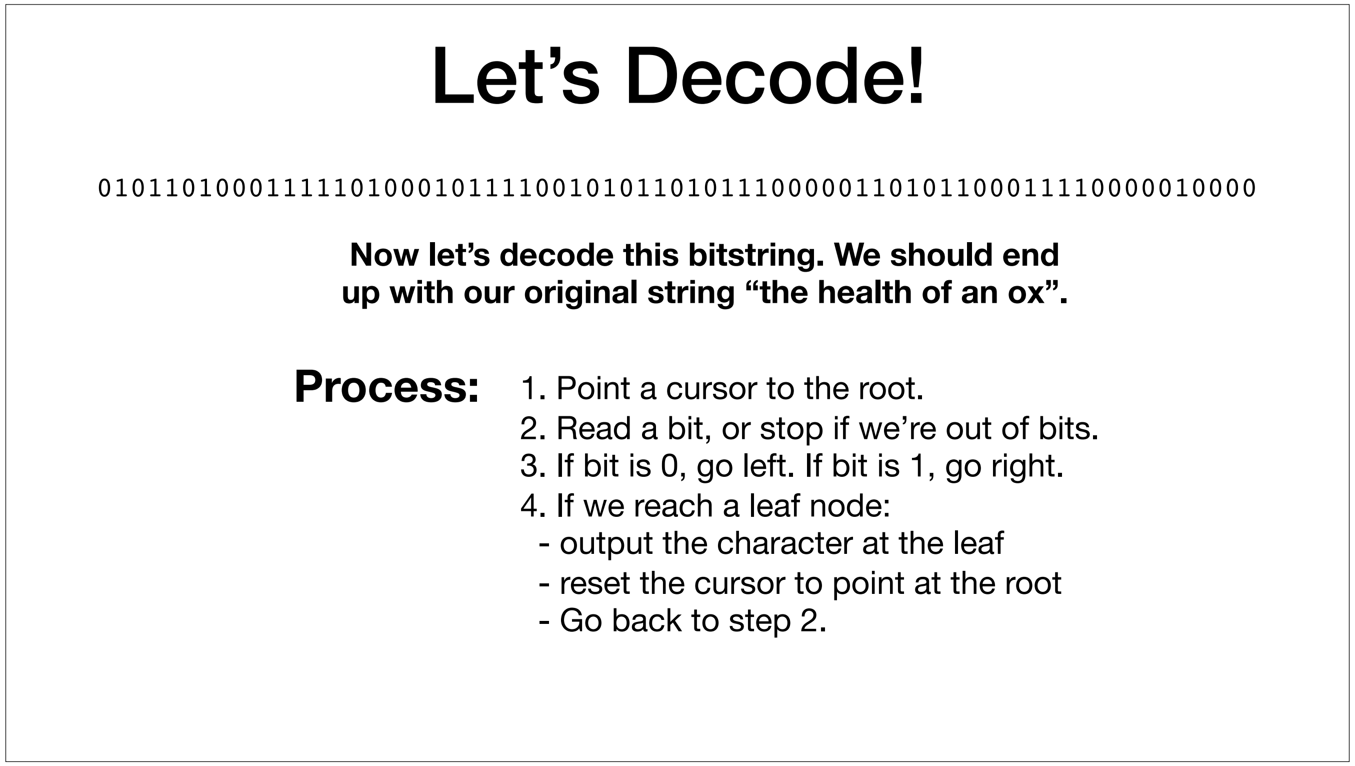
ASCII: 152 bits

Our Encoding: 69 bits

Roughly 55% compression

By the end, we have this bitstring. To encode this sentence using ASCII, it takes 152 bits, since each character is padded out to fill an 8-bit byte.

Our encoding takes 69 bits. This is about a 55% compression rate. Not too bad!



Let’s Decode!

```
010110100011111010001011110010101101011100000110101100011110000010000
```

Now let’s decode this bitstring. We should end up with our original string “the health of an ox”.

Process:

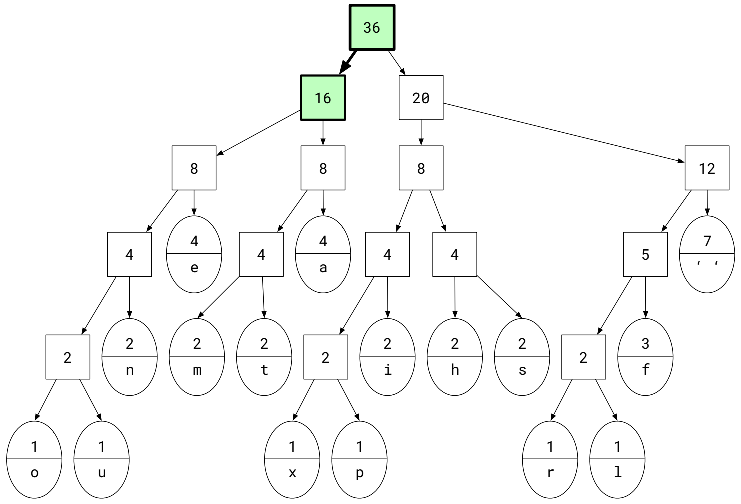
```
1. Point a cursor to the root.
2. Read a bit, or stop if we’re out of bits.
3. If bit is 0, go left. If bit is 1, go right.
4. If we reach a leaf node:
  - output the character at the leaf
  - reset the cursor to point at the root
  - Go back to step 2.
```

Now 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.

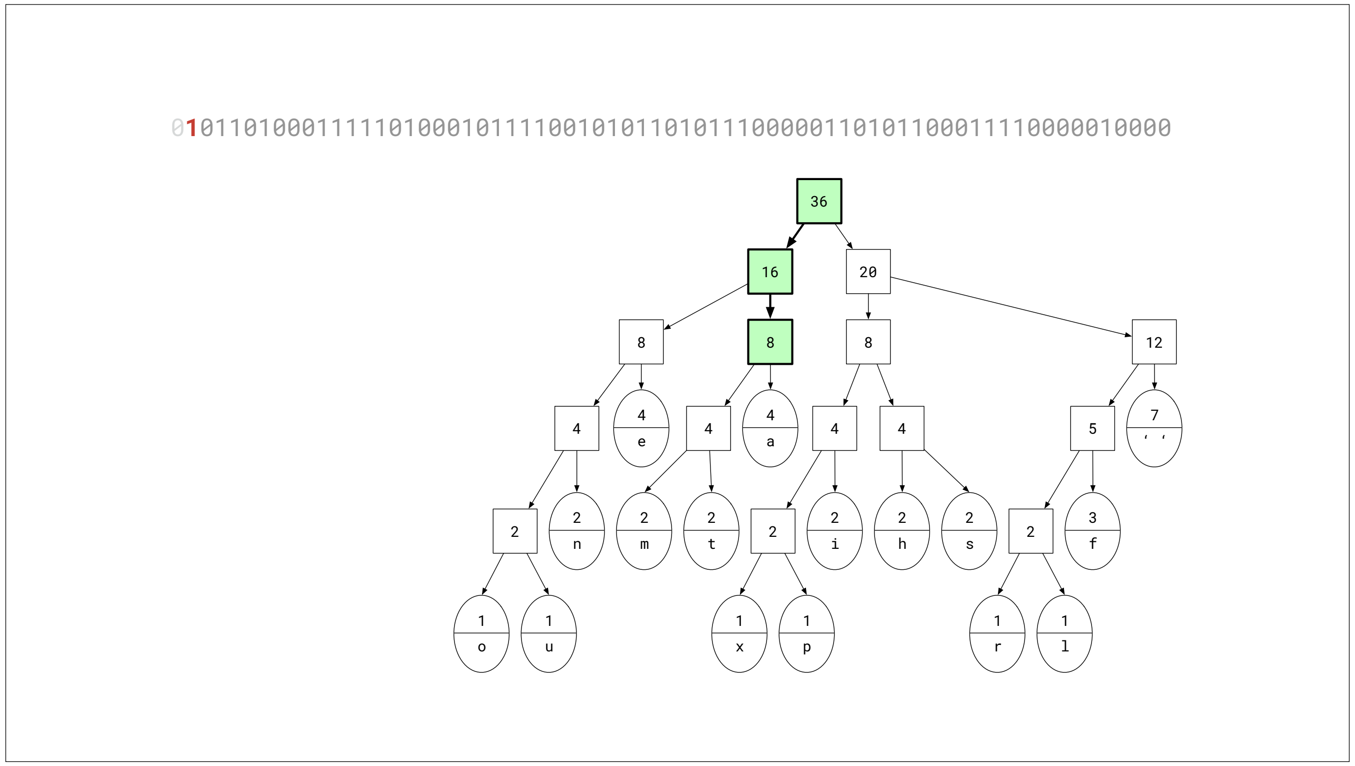
0101110100011111010001011110010101101011100000110101100011110000010000



0 = left

1 = right

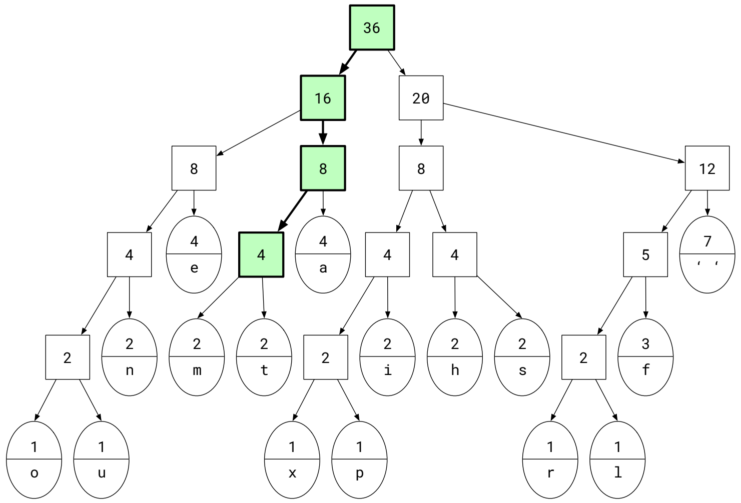
Starting from the root, we first see a zero, so we follow the left child. It isn’t a leaf node, so we just read the next bit.



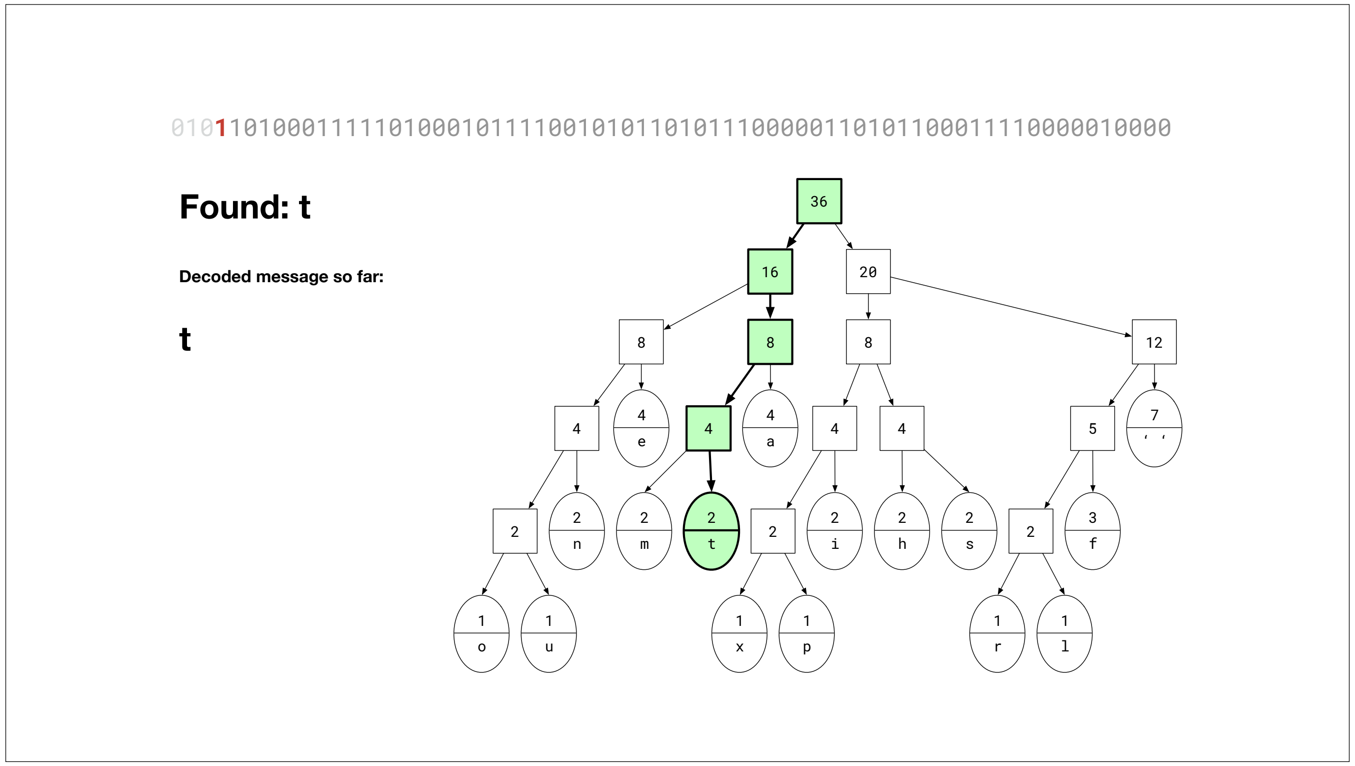
01011010001111101000101111001010110101110000011010110001111000001 0000

Next we get a one, so go right. Also not a leaf.

010110100011111010001011110010101101011100000110101100011110000010000



Read a zero, go left. Still an internal node, not a leaf.



0101101000111110100010111100101011010111000001101011000111110000010000

Found: t

Decoded message so far:

t

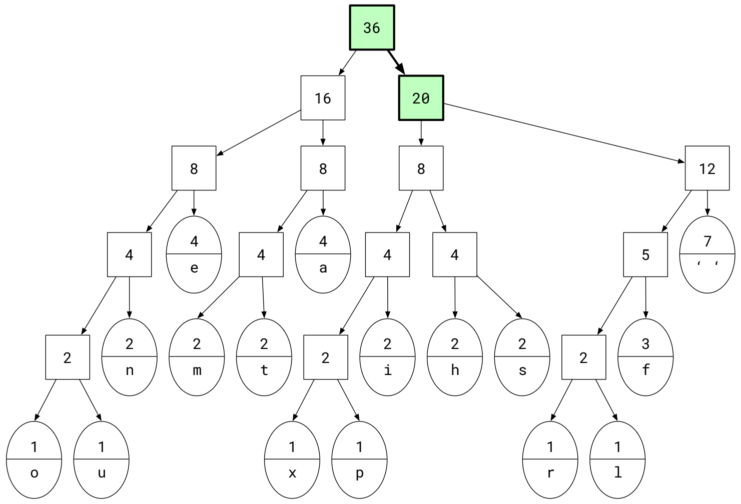
Read one, so go right. Now we encounter a leaf node, so we emit the symbol at that node, a T.

Reset the pointer to the root node and keep going.

01011010001111101000101111001010110101110000011010110001111000001000

Decoded message so far:

t

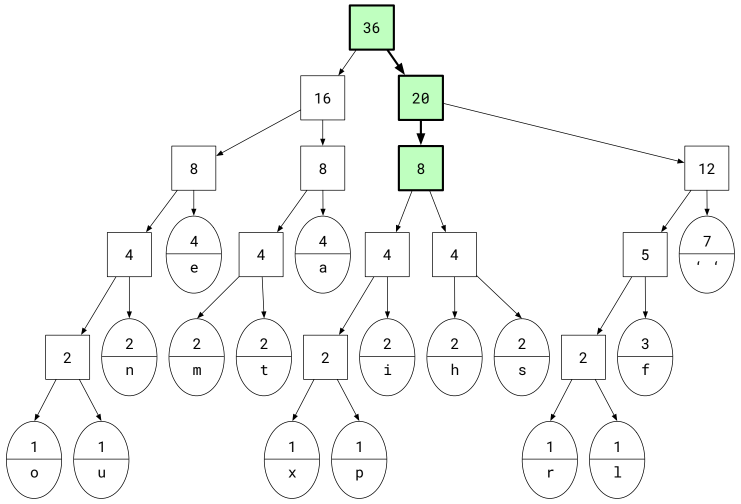


One, go right.

010110100011111010001011110010101101011100000110101100011110000010000

Decoded message so far:

t

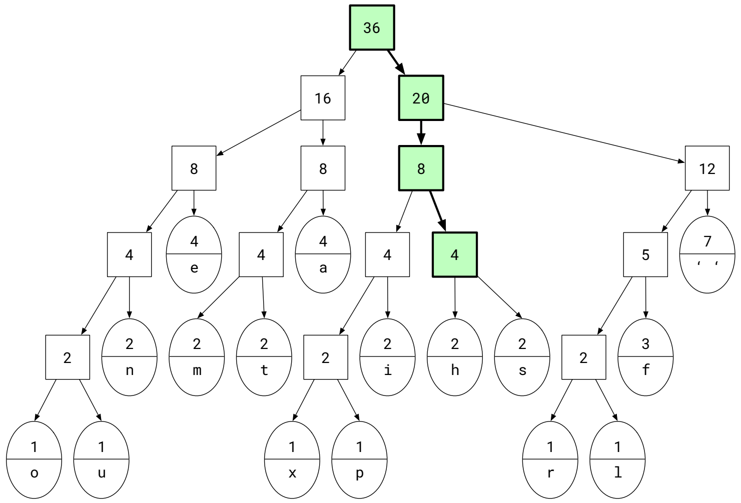


Zero, go left.

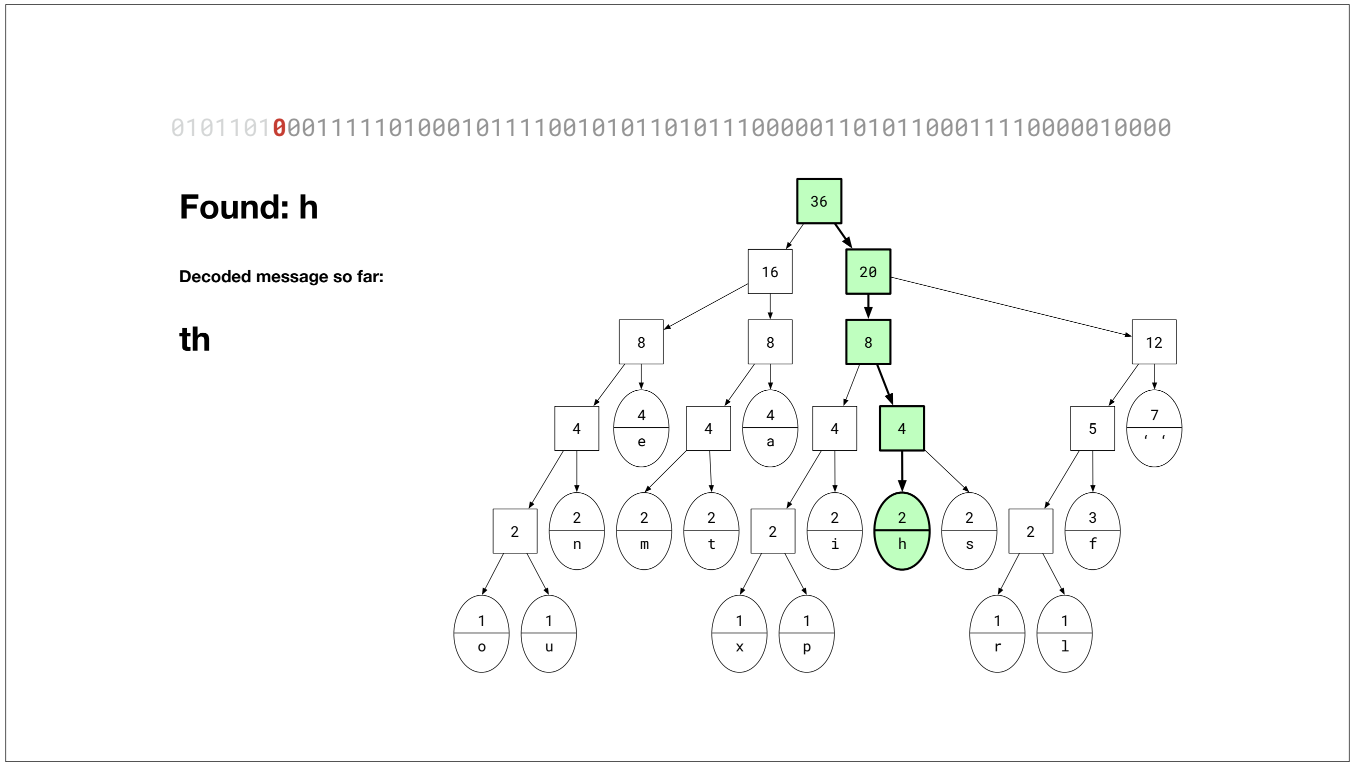
01011010001111101000101111001010110101110000011010110001111000001000 0

Decoded message so far:

t



One, go right.



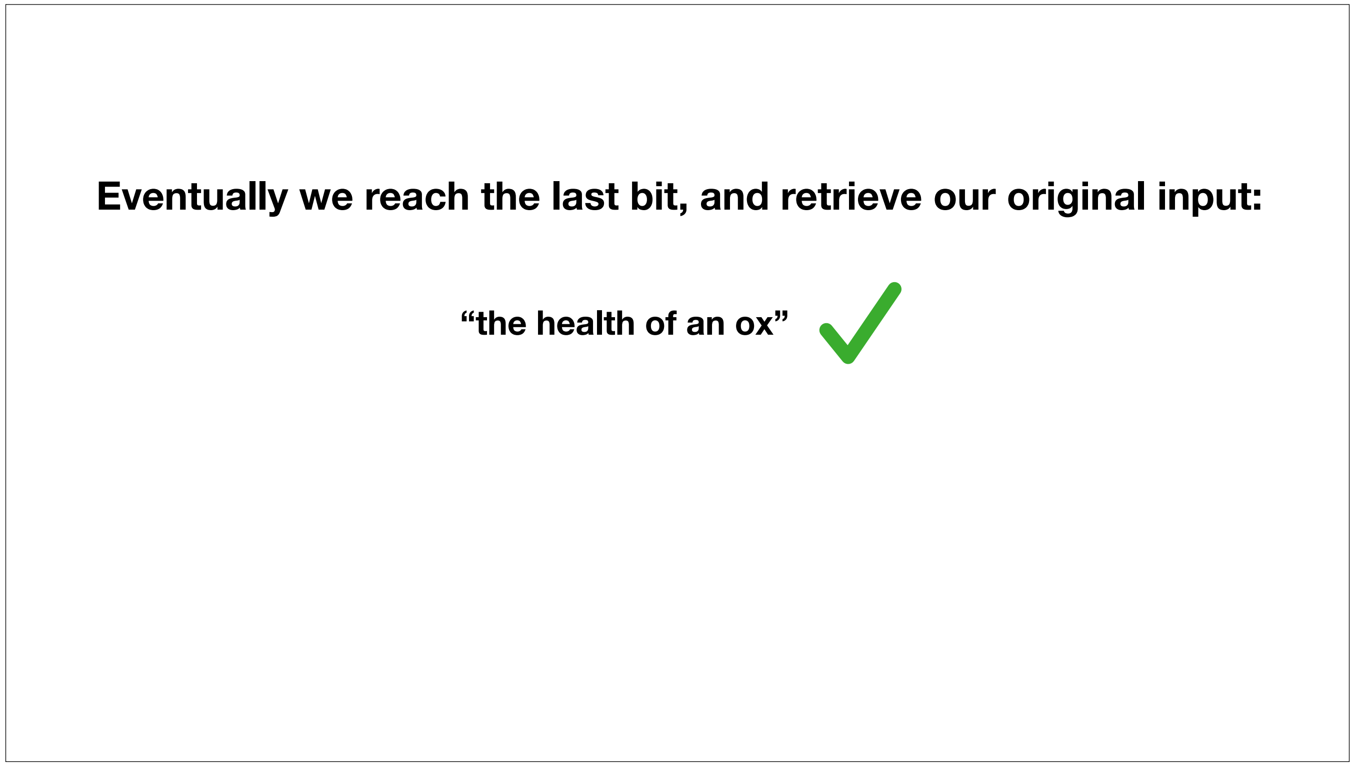
010110100011111010001011110010101101011100000110101100011110000010000

Found: h

Decoded message so far:

th

Zero, go left. Found a leaf node for H, so we emit that.



Eventually we reach the last bit, and retrieve our original input:

“the health of an ox” ✓

This process continues for a while, and eventually we decode the message, which was **<next>** ‘the health of an ox’. Honestly I don’t know why I picked that string, it was the first thing that came to mind that could be encoded with the corpus string that we started with.

So now you know how to take a corpus text and use it to make a frequency table, a Huffman tree, and then use that tree to encode and decode messages that is composed of symbols that appear in the original corpus text.