

Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- Continue discussion of **Trees**
 - › So far we've studied several types of Binary Trees:
 - Binary Heaps (Priority Queue)
 - Binary Search Trees/BSTs (Map)
 - We also heard about some variants and cousins of the BST: red-black trees, splay trees, B-Trees
- Today we're going to be talking about **Huffman trees**

Getting Started on Huffman

NEXT ASSIGNMENT AFTER
LINKED LISTS (WHICH IS DUE
WEDNESDAY)



Encoding with Huffman Trees:

- Today we're going to be talking about your next assignment: **Huffman coding**
 - › It's a compression algorithm
 - › It's provably optimal (take that, Pied Piper)
 - › It involves binary tree data structures, yay!

- **But before we talk about the tree structure and algorithm, let's set the scene a bit and talk about BINARY**
 - › (as in the 0/1 kind of binary)


Binary on computers

BINARY = BASE 2 NUMBERS
(WESTERN HUMANS USUALLY
USE BASE 10)



In a computer, everything is numbers!

Specifically, everything is binary

- | | |
|--|----------------|
| ▪ Integers (int): | binary numbers |
| ▪ Real numbers (double): | binary numbers |
| ▪ Letters and words (ASCII, Unicode): | binary numbers |
| ▪ Images (gif, jpg, png): | binary numbers |
| ▪ Music (mp3): | binary numbers |
| ▪ Movies/music (streaming): | binary numbers |
| ▪ Doge pictures (): | binary numbers |
| ▪ Email messages: | binary numbers |
| ▪ Program code: | binary numbers |

Encodings are what tell us how to translate

- › “if we interpret these binary digits as an image, it would look like this”
- › “if we interpret those same binary digits as music, it would sound like this”

In a computer, everything is numbers !

- Recall we represent variables as boxes
 - › What is contained in each box—whether it be an `int` or an `int*` or a string or anything else—is always some number of binary digits (bits)
 - › We can't know by looking at the bits whether they are being stored with the intention to be an `int` or an `int*` or a string or something else—just looks like bits
- Example of actual bits:

111001101110011011111010

Color
(RGB):



Number
(int):

15132410

ASCII encoding

1970S RETRO TIME



ASCII is an old-school encoding for characters

- The “char” type in C++ is based on ASCII
- Leftover from C in the 1970’s
- Recall from Katie Creel’s talk on representational harms:
 - › ASCII doesn’t play well with non-English languages, and today’s software can’t afford to be so America-centric, so Unicode is more common
- ASCII is simple so we use it for this assignment

ASCII Table

Notice each symbol
is encoded as 8
binary digits (8 bits)

There are 256
unique sequences of
8 bits, so numbers 0-
255 each correspond
to one character

(this only shows 32-74)

00111110 = ‘<’

DEC	OCT	HEX	BIN	Symbol	DEC	OCT	HEX	BIN	Symbol
32	040	20	00100000		53	065	35	00110101	5
33	041	21	00100001	!	54	066	36	00110110	6
34	042	22	00100010	"	55	067	37	00110111	7
35	043	23	00100011	#	56	070	38	00111000	8
36	044	24	00100100	\$	57	071	39	00111001	9
37	045	25	00100101	%	58	072	3A	00111010	
38	046	26	00100110	&	59	073	3B	00111011	
39	047	27	00100111	'	60	074	3C	00111100	
40	050	28	00101000	(61	075	3D	00111101	
41	051	29	00101001)	62	076	3E	00111110	>
42	052	2A	00101010	*	63	077	3F	00111111	?
43	053	2B	00101011	+	64	100	40	01000000	@
44	054	2C	00101100	,	65	101	41	01000001	A
45	055	2D	00101101	-	66	102	42	01000010	B
46	056	2E	00101110	.	67	103	43	01000011	C
47	057	2F	00101111	/	68	104	44	01000100	D
48	060	30	00110000	0	69	105	45	01000101	E
49	061	31	00110001	1	70	106	46	01000110	F
50	062	32	00110010	2	71	107	47	01000111	G
51	063	33	00110011	3	72	110	48	01001000	H
52	064	34	00110100	4	73	111	49	01001001	I
					74	112	4A	01001010	J

ASCII Example

char	ASCII	bit pattern (binary)
h	104	01101000
a	97	01100001
p	112	01110000
y	121	01111001
i	105	01101001
o	111	01101111
space	32	00100000

“happy hip hop” =

104 97 112 112 121 32 104 105 (decimal)

Or this in binary:

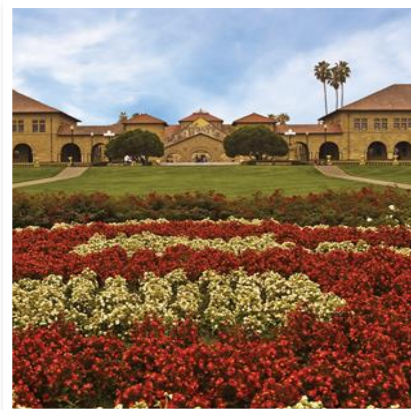
01101000	01100001	01110000	01110000	01111001	00100000	01101000
01101001	01110000	00100000	01101000	01101111	01110000	

FAQ: Why does 104 = ‘h’?

Answer: it’s arbitrary, like most encodings. Some people in the 1970s just decided to make it that way.

Craft Time!

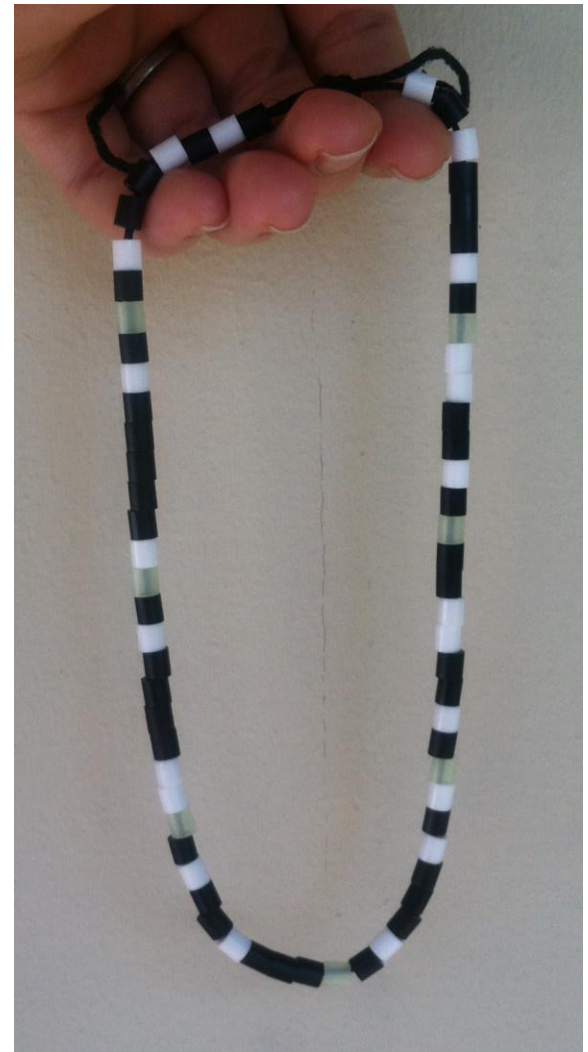
NERD FASHION



[Aside] Unplugged programming: The Binary Necklace

- Choose one color to represent 0's and another color to represent 1's
- Write your name in beads by looking up each letter's ASCII encoding
- For extra bling factor, this one uses glow-in-the dark beads as delimiters between letters

DEC	OCT	HEX	BIN	Symbol
65	101	41	01000001	A
66	102	42	01000010	B
67	103	43	01000011	C
68	104	44	01000100	D
69	105	45	01000101	E
70	106	46	01000110	F
71	107	47	01000111	G
72	110	48	01001000	H
73	111	49	01001001	I

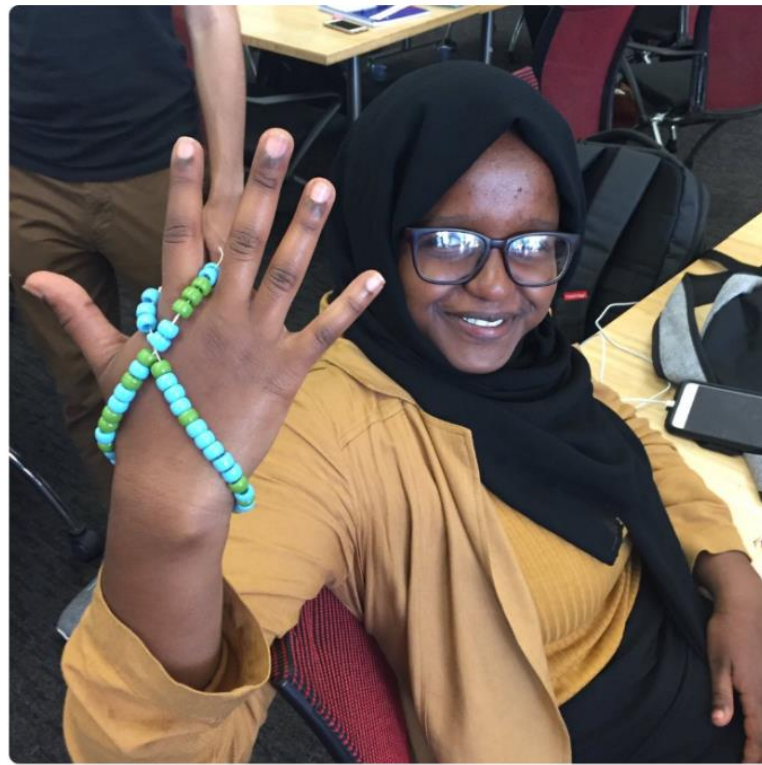


The Binary Necklace

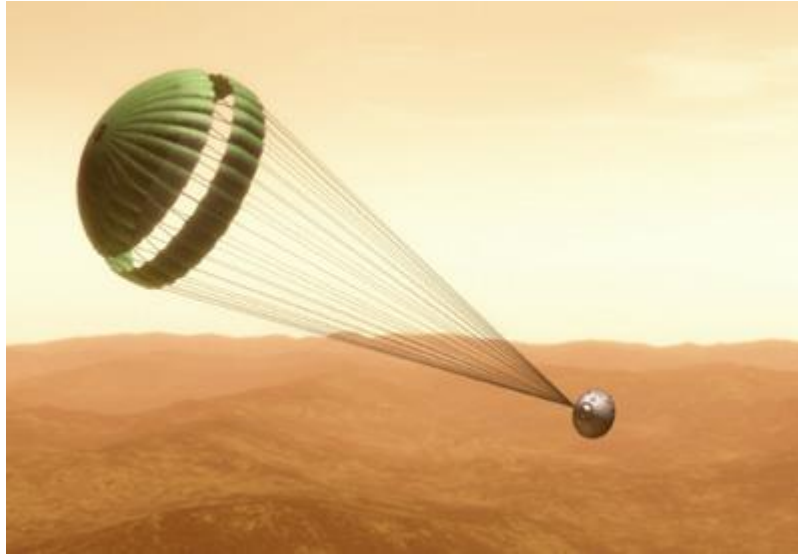
- Web tool to help you translate words to bead patterns:
 - › https://web.stanford.edu/~cbl/binary_bead_design.html

The Binary Necklace

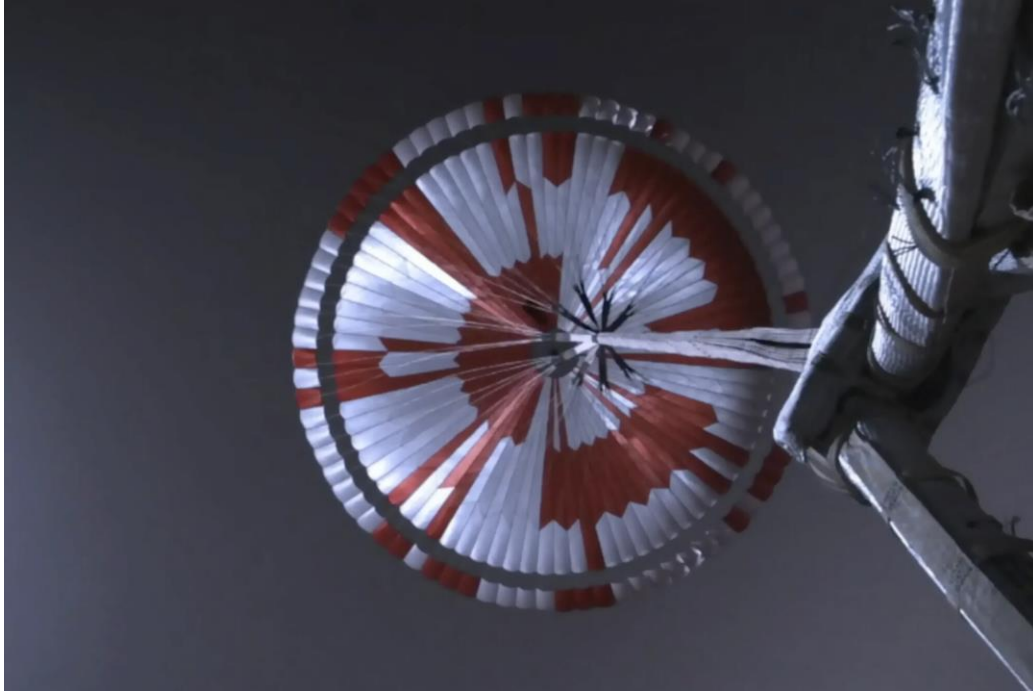
- Some ideas from previous students!



More binary fashion...the Mars Rover!



More binary fashion...the Mars Rover!



More binary fashion...the Mars Rover!

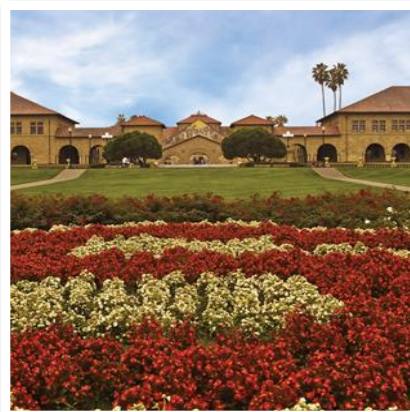


More binary fashion...the Mars Rover!



Non-ASCII encodings of characters

LOOKING TOWARDS
EFFICIENCY



Non-ASCII (variable-length) encoding example

char	bit pattern
h	01
a	000
p	10
y	1111
i	001
o	1110
space	110

“happy hip hop” =

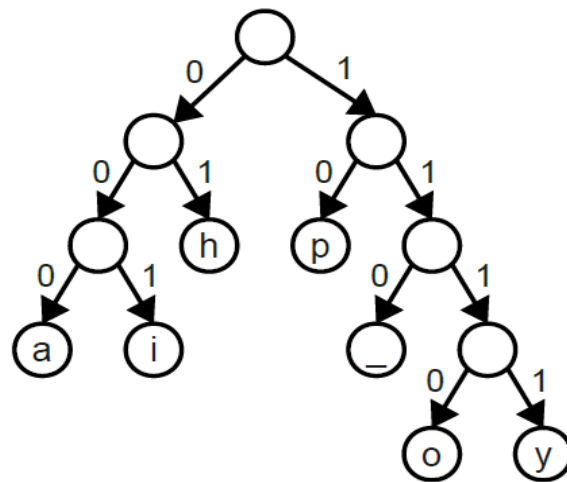
01	000	10	10	1111	110	01	001	10	110	01	1110	10
----	-----	----	----	------	-----	----	-----	----	-----	----	------	----

The variable-length encoding scheme makes a MUCH more space-efficient message than ASCII:

01101000	01100001	01110000	01110000	01111001	00100000	01101000
01101001	01110000	00100000	01101000	01101111	01110000	

Huffman encoding

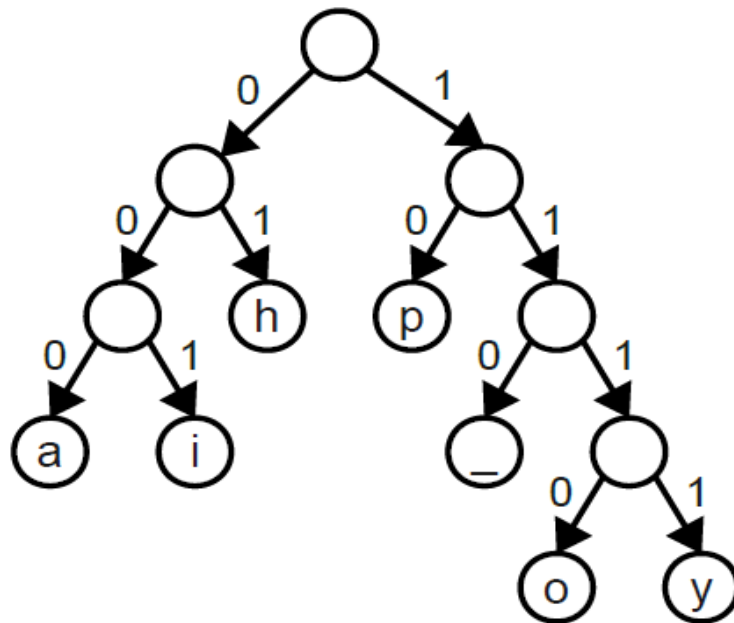
- Huffman encoding is a way of choosing which characters are encoded which ways, customized to the specific file you are using
- Example: character '#'
 - › Rarely used in Shakespeare (code could be longer, say ~10 bits)
 - › If you wanted to encode a Twitter feed, you'd see # often (maybe only ~4 bits) #contextmatters #thankshuffman
- We store the code translation as a tree:



Your turn

What would be the binary encoding of “hippo” using this Huffman encoding tree?

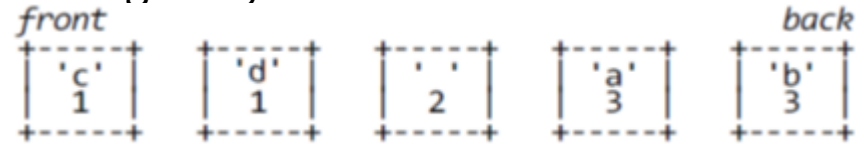
- A. 11000
- B. 0101101010
- C. 0100110101110
- D. 0100010101111
- E. Other/none/more than one



Okay, so how do we make the tree?

1. Read your file and count how many times each character occurs
2. Make a collection of tree nodes, each having a key = # of occurrences and a value = the character

› Example: “c aaa bbbd”

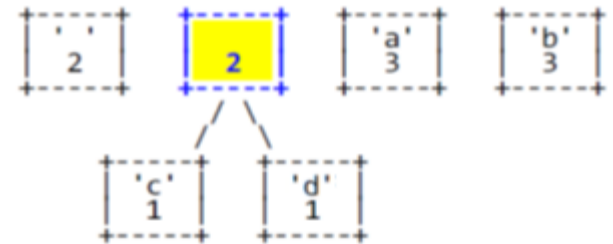


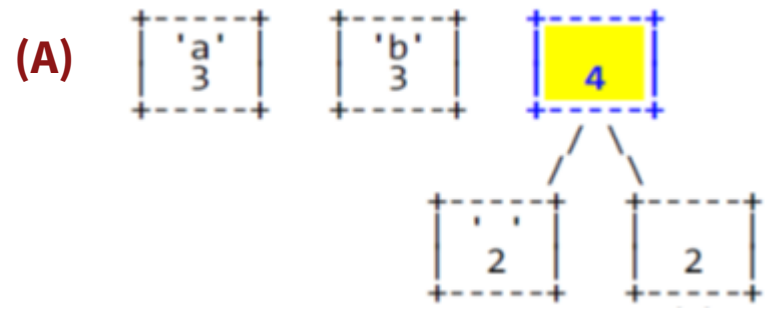
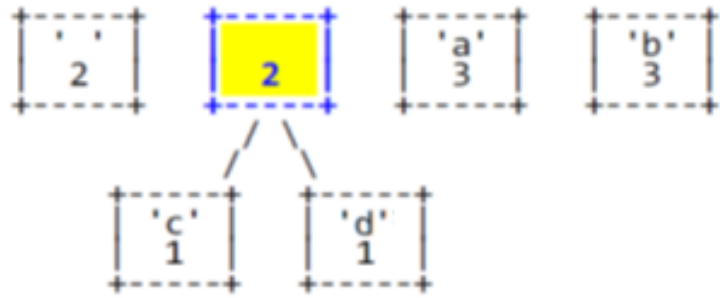
› For now, tree nodes are not in a tree shape

› We actually store them in a Priority Queue (yay!!) based on highest priority = LOWEST # of occurrences

› Next:

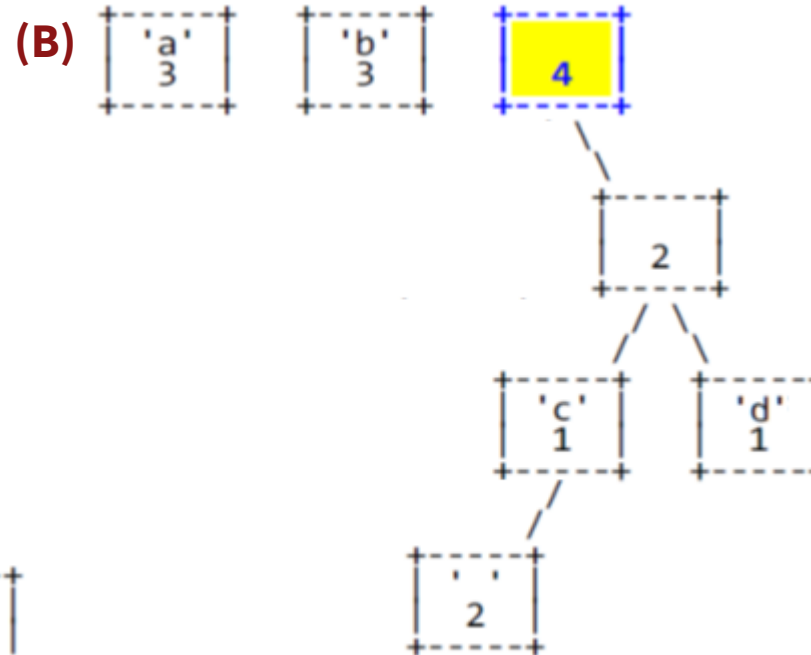
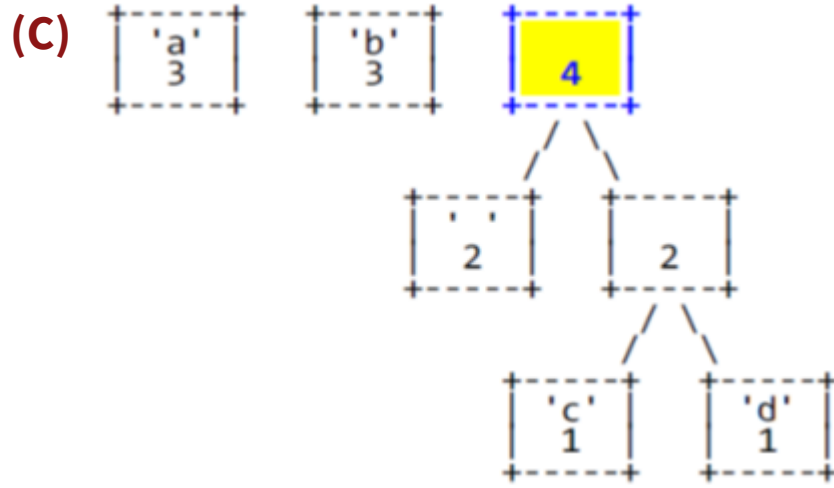
- Dequeue two nodes and make them the two children of a new node, with no character and # of occurrences is the sum,
- Enqueue this new node
- Repeat until `PQ.size() == 1`



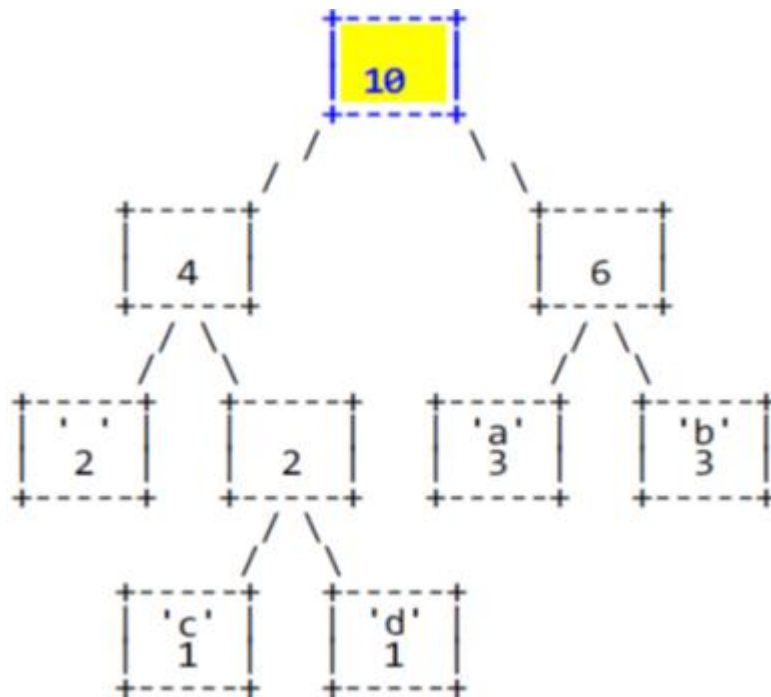
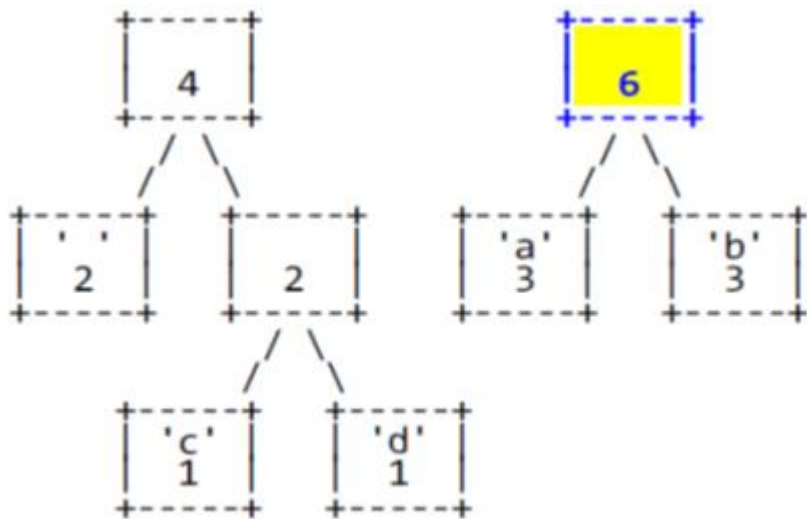


Your turn

If we start with the Priority Queue above, and execute one more step, what do we get?



Last two steps



Now assign codes

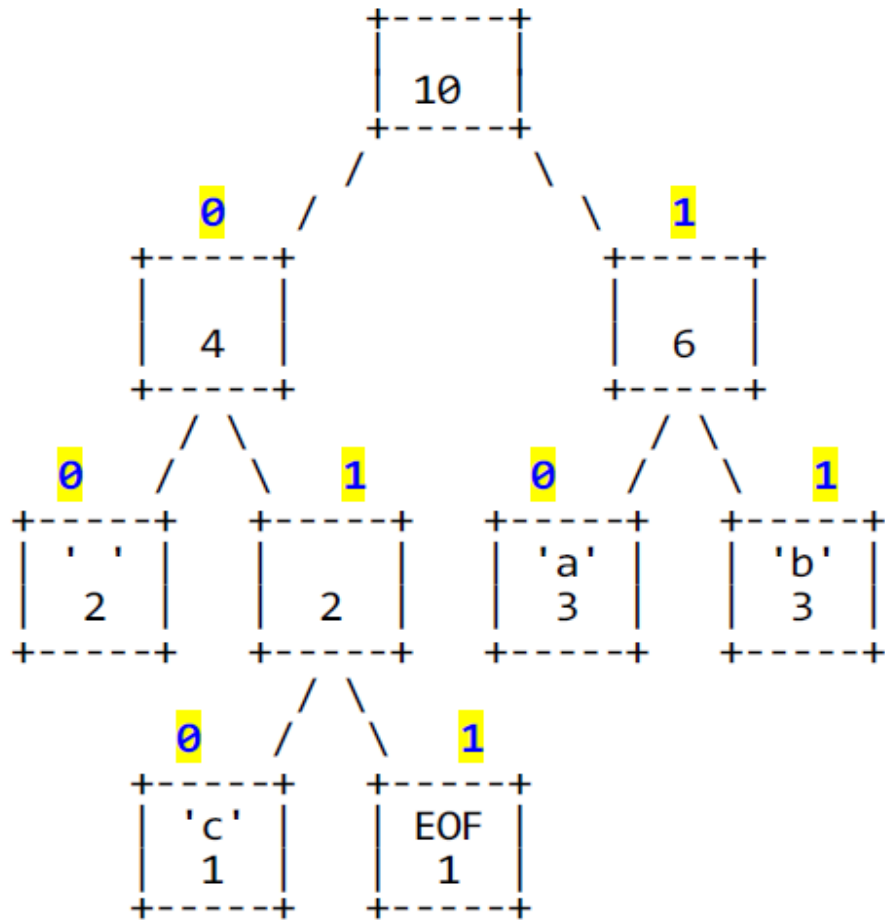
We interpret the tree as:

- Left child = 0
- Right child = 1

What is the code for “c”?

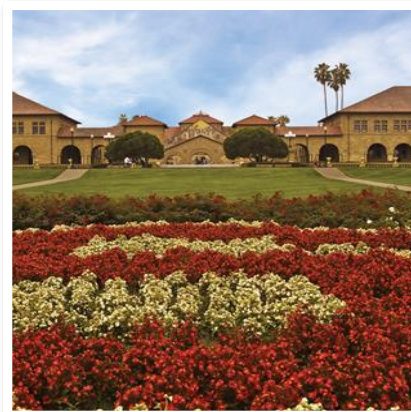
- A. 00
- B. 010
- C. 101
- D. Other/none

c	a	b
010	10	11



Pop Quiz Time

ABOUT THAT EFFICIENCY...



Key question: How do we know when one character's bits end and another's begin?

Your turn:

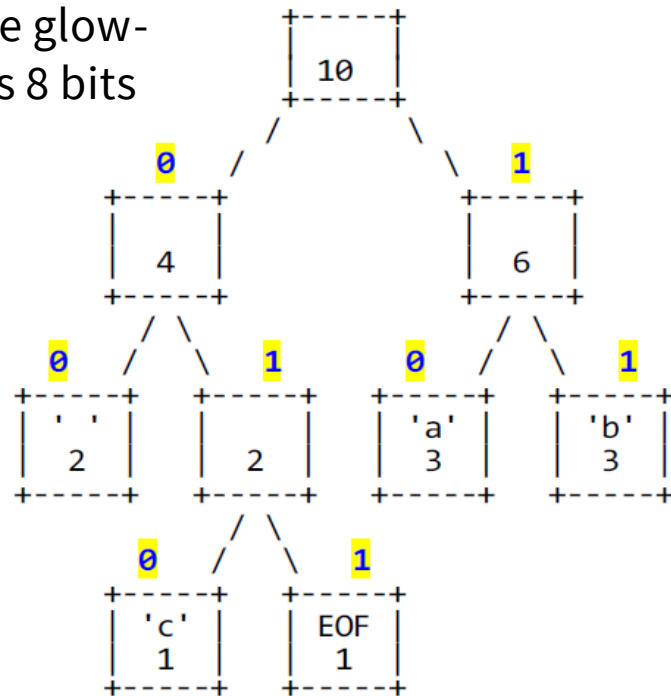
TRUE OR FALSE: Huffman needs delimiters (like the glow-in-the-dark beads), unlike ASCII, which is always 8 bits (and didn't really need the beads).

A. TRUE

B. FALSE

Discuss/prove it: why or why not?

c	a	b
010	10	11



ASCII

- ASCII's uniform encoding size makes it easy
 - › Don't really need those glow-in-the-dark beads as delimiters, because we know every 9th bead starts a new 8-bit letter encoding
- Key insight: also a bit wasteful (*ha! get it? a "bit"*)
 - › What if we took the most commonly used characters (according to *Wheel of Fortune*, some of these are RSTLNE) and encoded them with just 2 or 3 bits each?
 - › We let seldom-used characters, like &, have encodings that are longer, say 12 bits.
 - › Overall, we would save a lot of space!