Topic 20: Huffman Coding

April 4, 2024

Data Storage and Representation

Just a Little Bit of Magic

- Digital data is stored as sequences of 0s and 1s
  - These sequences are encoded in physical devices by magnetic orientation on small (10 nm) metal particles or by trapping electrons in small gates
- A single 0 or 1 is called a **bit**
- A group of 8 bits is called a **byte**

`00000000, 00000001, 00000010, 00000011, 00000100, …`

- There are 2^8, so 256, different bytes
  - Good recursive backtracking practice: Write a function that lists all possible byte sequences!

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc
- However, we just said that computers require everything to be written as zeros and ones
- To bridge the gap, we need to agree on some universal way of representing characters as sequences of bits
- Idea: ASCII!

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

`010010000100100101010000`

HIP

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Observations

- Every characters uses exactly the same number of bits, 8, which makes it very easy to differentiate between the characters
- Any message with n characters will use exactly 8n bits
  - Space for RAMBUNCTIOUS: 8 x 12 = 96 bits
  - Space for CS_314_ROCKS: 8 x 12 = 96 bits
- Let's make this more efficient by reducing the number of bits we need to encode text

Main Character Today

HAPPY HIP HOP

ASCII Encoding

- ASCII uses 8 bits to represent each character
- Let's represent **HAPPY HIP HOP** in ASCII code

character	*ASCII code*
_	00100000
A	01000001
H	01001000
I	01001001
O	01001111
P	01010000
Y	01011001

0100 1000	0100 0001	0101 0000	0101 0000	0101 1001	0010 0000	0100 1000	0100 1001	0101 0000	0010 0000	0100 1000	0100 1111	0101 0000
H	A	P	P	Y	_	H	I	P	_	H	O	P

A Different Encoding

- If we're specifically writing the string **HAPPY HIP HOP**, which only has 7 different characters, using full bytes (8 bits) is wasteful
- Let's use a 3-bit encoding instead

character	*code*
_	000
A	001
H	010
I	011
O	100
P	101
Y	110

010	001	101	101	101	000	010	011	101	000	010	100	101
H	A	P	P	Y	_	H	I	P	_	H	O	P

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

`010011101`

HIP

character	*code*
`_`	`000`
`A`	`001`
`H`	`010`
`I`	`011`
`O`	`100`
`P`	`101`
`Y`	`110`

A Different Encoding

character	code
_	000
A	001
H	010
I	011
O	100
P	101
Y	110

- When specifically writing the string **HAPPY HIP HOP**
  - ASCII used 13 x 8 bits = 104 bits
  - 3-bit encoding used 13 * 3 bits = 39 bits
- We used 37.5% of the space that ASCII uses!

010	001	101	101	101	000	010	011	101	000	010	100	101
H	A	P	P	Y	_	H	I	P	_	H	O	P

The Journey Ahead

- Storing data using the ASCII encoding is portable across systems, but is not ideal in terms of space usage
- Building custom codes for specific strings (and files!) might let us save space
- Idea: Use this approach to build a **compression algorithm** to reduce the amount of space needed to store text
- We want to find a way to

  give all characters a bit pattern,
  that both the sender and receiver know about, and
  that can be decoded uniquely, and
  that leads to less space usage.

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that have widespread prevalence and importance
- We're interested in algorithms that provide lossless compression on a stream of characters or other data
  - We make the amount of data smaller without losing any of the details, and we can decompress the data to exactly the same as it was before compression
- Virtually everything you do online involves data compression
  - When you visit a website, download a file, or transmit video/audio, the data is compressed when sending and decompressed when receiving
  - A video stream on Zoom has a compression of roughly 2000:1, meaning that a 2MB image is compressed down to just 1000 bytes
- Compression algorithms identify patterns in data and take advantage of those to come up with more efficient representations of that data

A Different Encoding

- Let's make this encoding even more efficient!

character	*code*
`_`	`000`
`A`	`001`
`H`	`010`
`I`	`011`
`O`	`100`
`P`	`101`
`Y`	`110`

`010`	`001`	`101`	`101`	`110`	`000`	`010`	`011`	`101`	`000`	`010`	`100`	`101`
`H`	`A`	`P`	`P`	`Y`	`_`	`H`	`I`	`P`	`_`	`H`	`O`	`P`

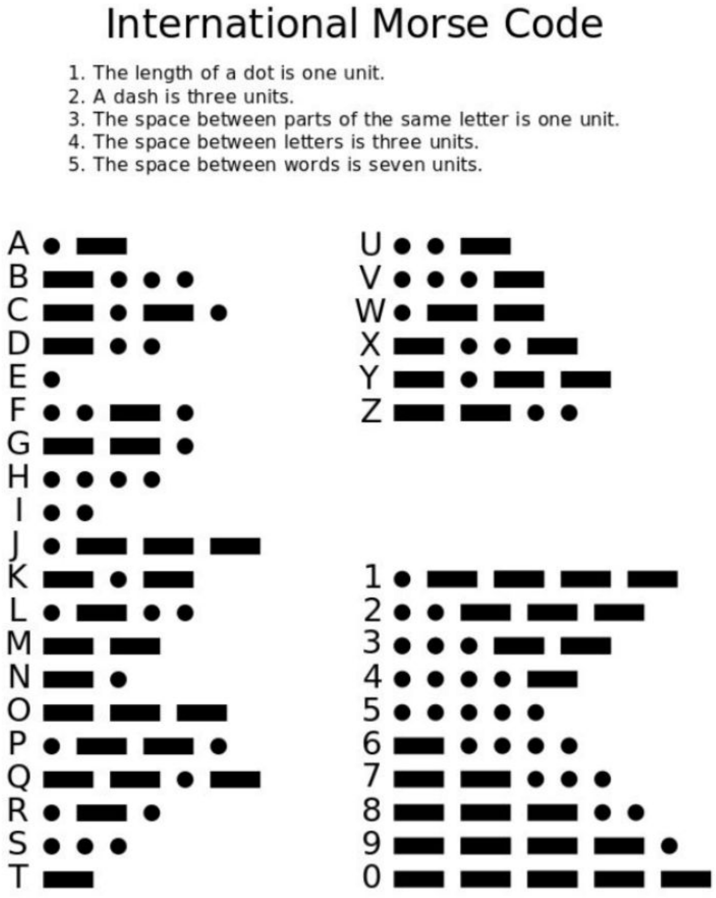
Take Advantage of Redundancy

- Not all letters have the same frequency in **`HAPPY HIP HOP`**
- We can calculate the frequencies of each letter
- So far, we've given each letter a code of the same length
- Maybe we can give shorter encodings to more frequent letters to save space?

character	*frequency*
_	2
A	1
H	3
I	1
O	1
P	4
Y	1

Morse Code

- Morse code is an example of an encoding system that makes use of this insight
- The codes for frequent letters (ex: e, t, a) are much shorter than the codes for infrequent letters (ex: q, y, j)



Our New Code

- When specifically writing the string **HAPPY HIP HOP**
  - ASCII used 13 x 8 bits = 104 bits
  - 3-bit encoding used 13 * 3 bits = 39 bits
  - Variable-length encoding used 20 bits
- We saved even more space!

character	*frequency*	*code*
P	4	0
H	3	1
_	2	00
A	1	01
I	1	10
O	1	11
Y	1	100

1	01	0	0	100	00	1	10	0	00	1	11	0
H	A	P	P	Y	_	H	I	P	_	H	O	P

Our New Code

What is the mystery word represented by this variable-length encoding?

`001100`

_O_

PPO_

PAY

character	*frequency*	*code*
P	4	0
H	3	1
_	2	00
A	1	01
I	1	10
O	1	11
Y	1	100

Our New Code

HAPPY HIP HOP

1010010000110000111O

IIPH_PAYPAOP

character	*frequency*	*code*
P	4	0
H	3	1
_	2	00
A	1	01
I	1	10
O	1	11
Y	1	100

What went wrong?

- If we use a different number of bits for each letter, we can’t necessarily uniquely determine the boundaries between letters
- We need an encoding that makes it possible to determine where one characters ends and the next begins
  - Codes for each character need to be unique and unambiguous
  - Otherwise, it isn't possible to decode the words accurately
- How can we do this?

Prefix Code

- A prefix code is an encoding system in which no code is a prefix of another code

character	*code*
`P`	`10`
`H`	`01`
`_`	`110`
`A`	`001`
`I`	`000`
`O`	`1111`
`Y`	`1110`

Prefix Code

What is the mystery word represented by this encoding?

100011110

PAY

character	*code*
P	**10**
H	**01**
_	**110**
A	**001**
I	**000**
O	**1111**
Y	**1110**

Prefix Code

character	*code*
P	10
H	01
_	110
A	001
I	000
O	1111
Y	1110

HAPPY HIP HOP

0100110101110110010001011001111110

01	001	10	10	1110	110	01	000	10	110	01	1111	10
H	A	P	P	Y	_	H	I	P	_	H	O	P

Prefix Code

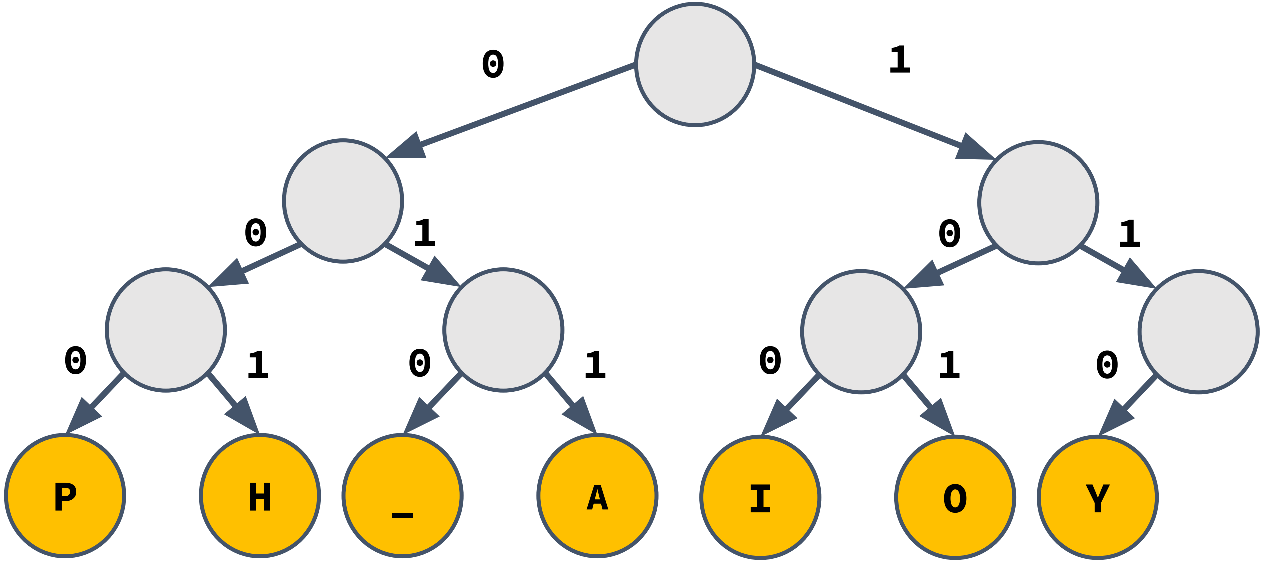
- When specifically writing the string **`HAPPY HIP HOP`**
  - ASCII used 13 x 8 bits = 104 bits
  - 3-bit encoding used 13 * 3 bits = 39 bits
  - Variable-length encoding used 34 bits
- We saved even more space and the encoding is ambiguous!

character	*code*
P	10
H	01
_	110
A	001
I	000
O	1111
Y	1110

01	001	10	10	1110	110	01	000	10	110	01	1111	10
H	A	P	P	Y	_	H	I	P	_	H	O	P

Coding Tree

- We can represent a prefix coding scheme using a binary tree, specifically a **coding tree**

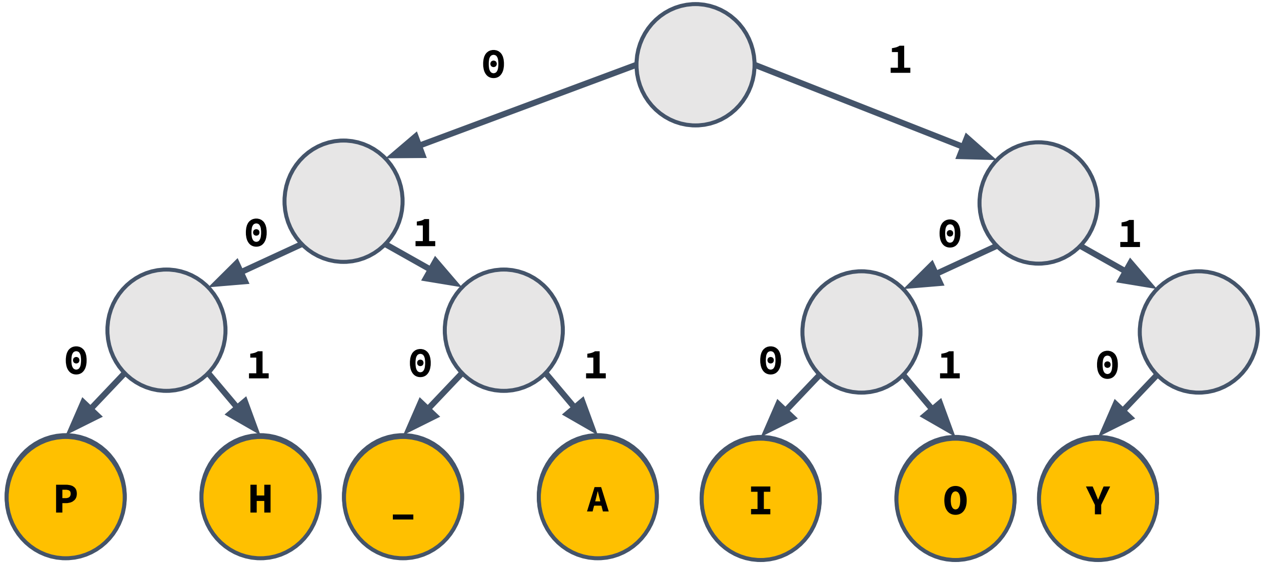


character	*code*
P	000
H	001
_	010
A	011
I	100
O	101
Y	110

Coding Tree

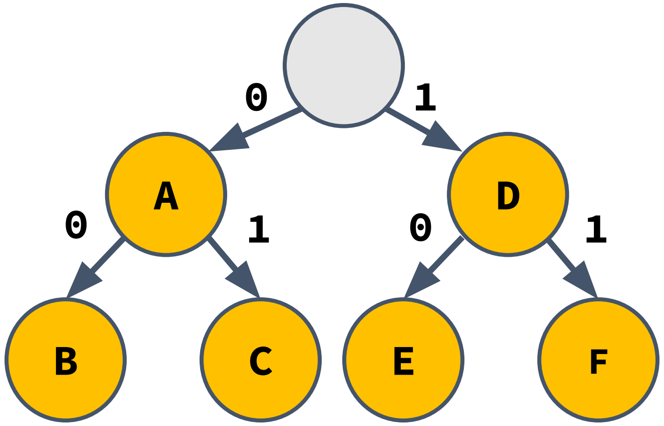
What is the mystery word represented by this encoding?

110011000



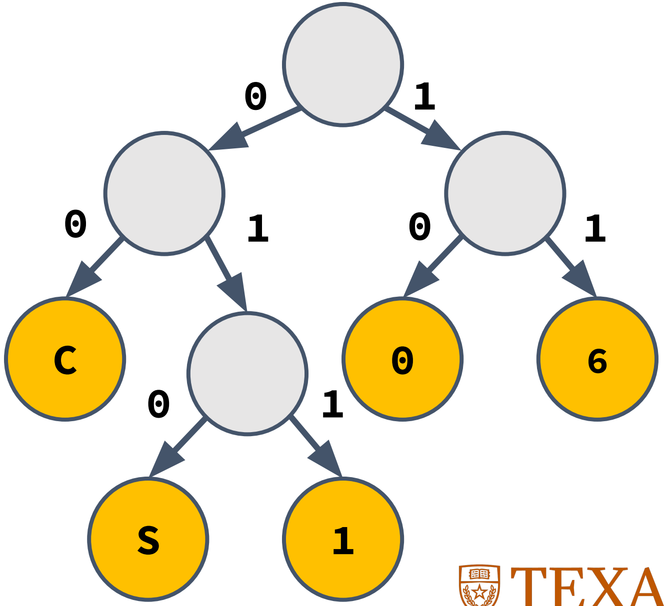
Coding Trees

- Not all binary trees work as coding trees
- Why is this binary tree not a coding tree?
  - Doesn’t give a prefix code!
  - The code for **A** is a prefix for the codes for **B** and **C**, and the code for **D** is a prefix for the codes for **E** and **F**

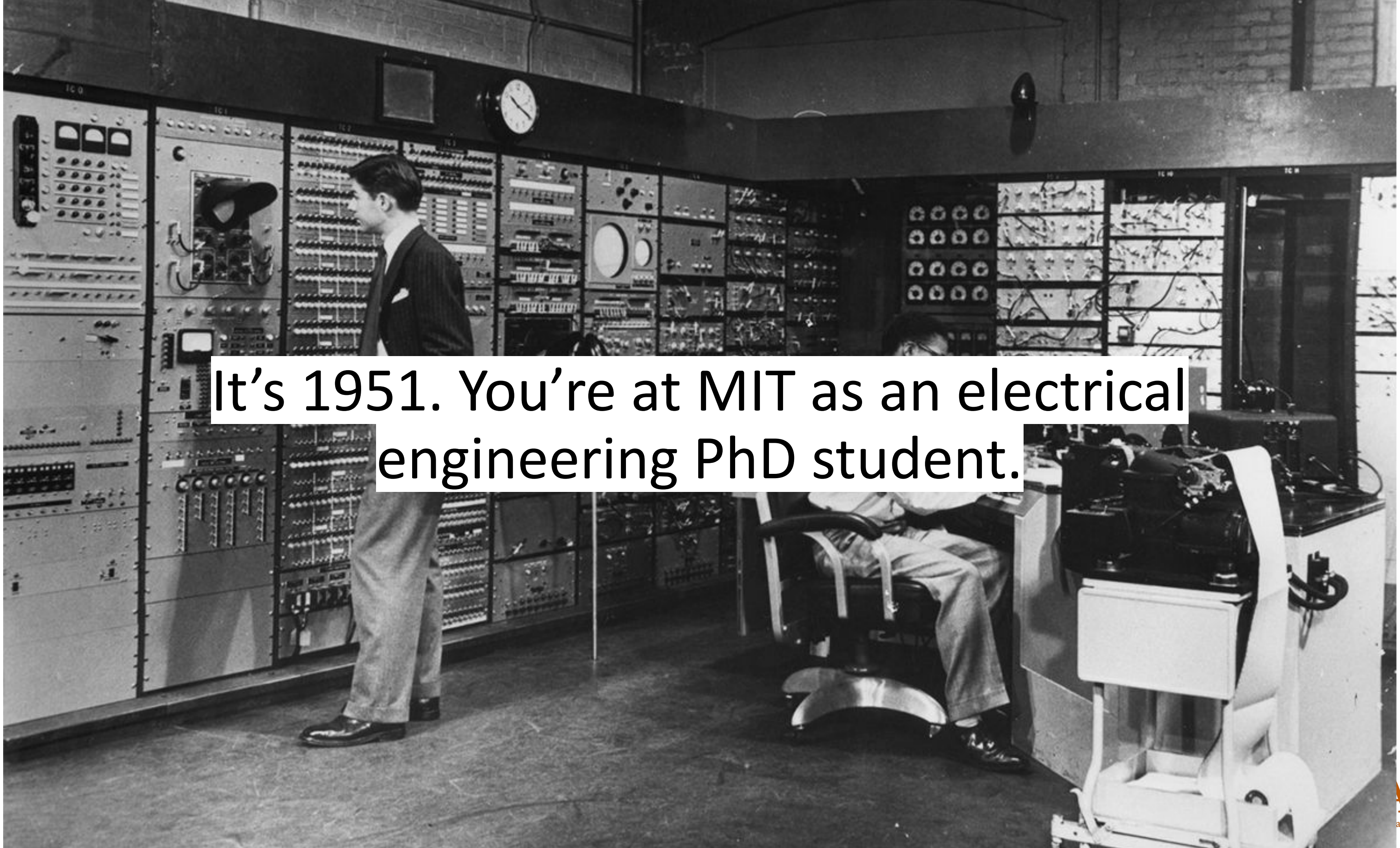


Coding Trees

- A coding tree is valid if all the letters are stored in the leaves, with internal nodes only used for routing



Huffman Coding



It’s 1951. You’re at MIT as an electrical engineering PhD student.

You have a choice for your class: take the final exam or write a term paper.

You choose to write the term paper.
The prompt is: find a provably most efficient method of representing numbers, letters, or symbols using binary code

David Huffman tries to solve this problem for **months**.

It’s 1951, so no Google or StackOverflow.

Important note:
Neither his professor, Robert M. Fano, nor the inventor of information theory, Claude Shannon, had any idea how to solve it

So David Huffman gives up, and starts studying for the final exam instead.

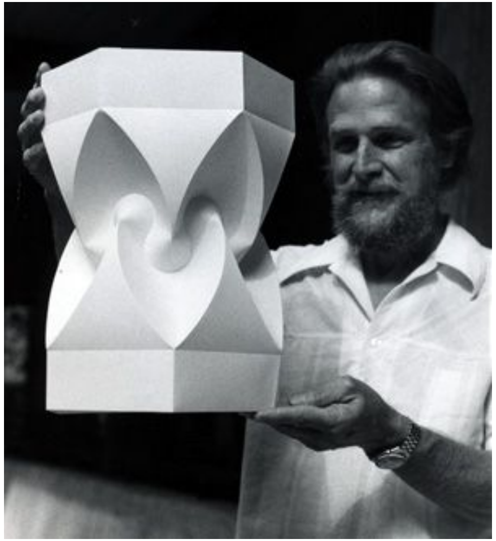
But then, epiphany!

"It was the most singular moment of my life. There was the absolute lightning of sudden realization."

- Huffman

"It was my luck to be there at the right time and also not have my professor discourage me by telling me that other good people had struggled with his problem."

- Huffman



[Link to full story]

The Algorithm

Huffman Coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies
- Applicable to many forms of data transmission
  - Our examples use characters and text files
  - JPEG and MP3 still use prefix codes
- Different data will each have their own personalized Huffman coding tree
- We want an encoding tree that
  - Allows for variable length codes (so most frequent characters can get shorter codes, aka their leaf nodes are closer to the root node)
  - Represents a prefix code system (no ambiguity in when characters stop and start)

Goal: Build the optimal encoding tree for

`HAPPY HIP HOP`

1. Build a frequency table

Input text: **`HAPPY HIP HOP`**

character	*frequency*
P	**4**
H	**3**
_	**2**
A	**1**
I	**1**
O	**1**
Y	**1**

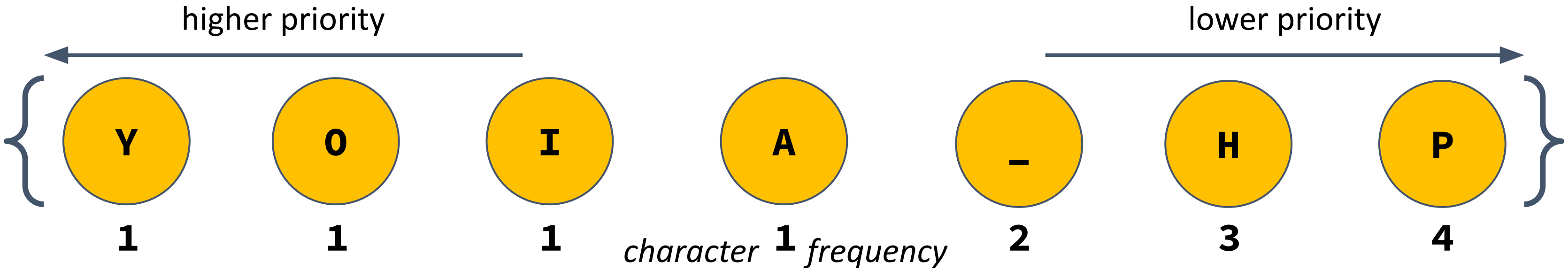
2. Initialize an empty priority queue

higher priority

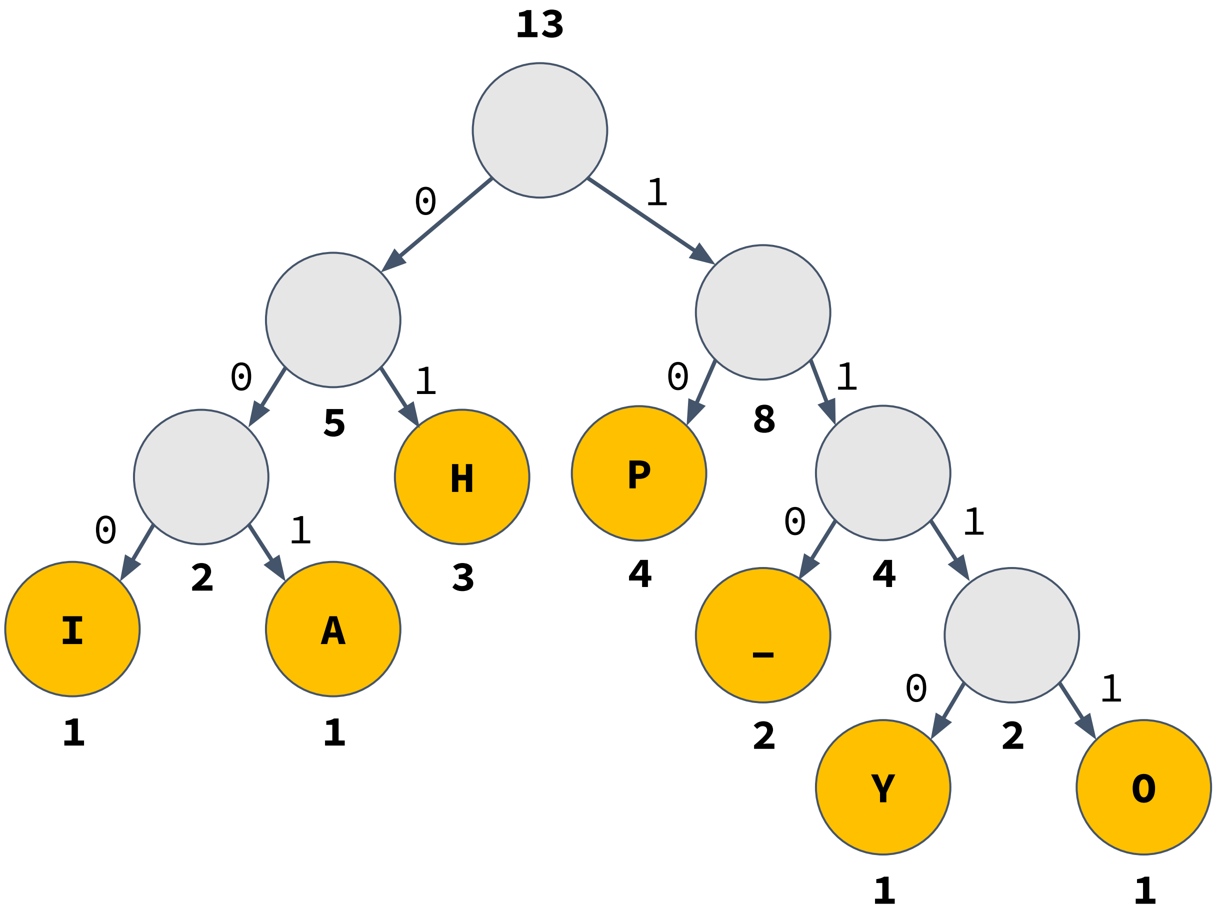
lower priority

$\{ \qquad \}$

3. Add all unique characters as leaf nodes to queue



character	frequency
P	4
H	3
_	2
A	1
I	1
O	1
Y	1

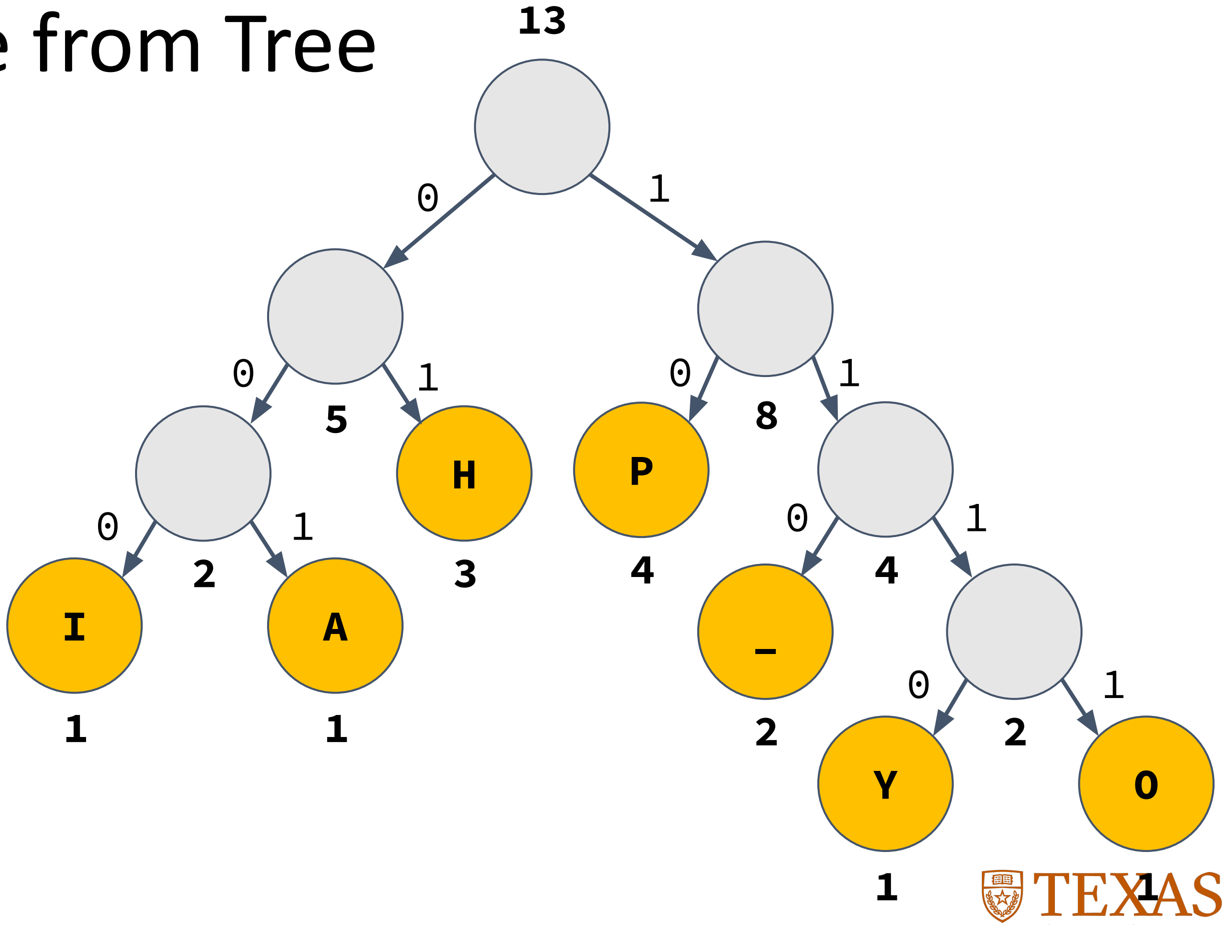

13
0
1
5
8
0
1
0
1
2
H
P
3
4
0
1
0
1
I
A
_
4
1
1
2
0
1
2
Y
O
1
1

Huffman Coding Algorithm

1. Scan the file to be compressed and build a frequency table that tallies the number of times each value appears.
2. Initialize an empty priority queue that will hold partial trees.
3. Create one leaf node per distinct value and add each leaf node to the queue where the priority is the frequency of the value.
4. While there are two or more trees in the priority queue:
    a. Dequeue the two lowest-priority trees.
    b. Combine them together to form a new tree whose priority is the sum of the priorities of the two trees.
    c. Add that tree back to the priority queue.

Generate Table from Tree

character	*code*
_	110
A	001
H	01
I	000
O	1111
P	10
Y	1110



Huffman Coding Algorithm

5. Traverse the tree to create the encoding table.
6. Scan the file again to create a new compressed file using the Huffman codes.

Prefix Code

character	code
P	10
H	01
_	110
A	001
I	000
O	1111
Y	1110

HAPPY HIP HOP

0100110101110110010001011001111110

01	001	10	10	1110	110	01	000	10	110	01	1111	10
H	A	P	P	Y	_	H	I	P	_	H	O	P

End of File

- Not possible to write a single bit at a time, all output is written in "chunks" (often 8 bits)
- If a program writes a number of bits that is not a multiple of 8, extra bits are added (usually 0s)
  - Can't just keep reading bits in from a file until you run our, since the bits might be dummy data at the end
- Instead, create a pseudo-EOF value that has its own Huffman encoding and write that to the compressed file
  - Add to frequency table with count of 1

Our Goal

We want to find a way to

- give all characters a bit pattern,
- that both the sender and receiver know about, and
- that can be decoded uniquely, and
- that leads to less space usage.

We've created an encoding, but need to make sure that the receiver can also decode the file.

Decode

111000010110111000010

Transmitting the Tree

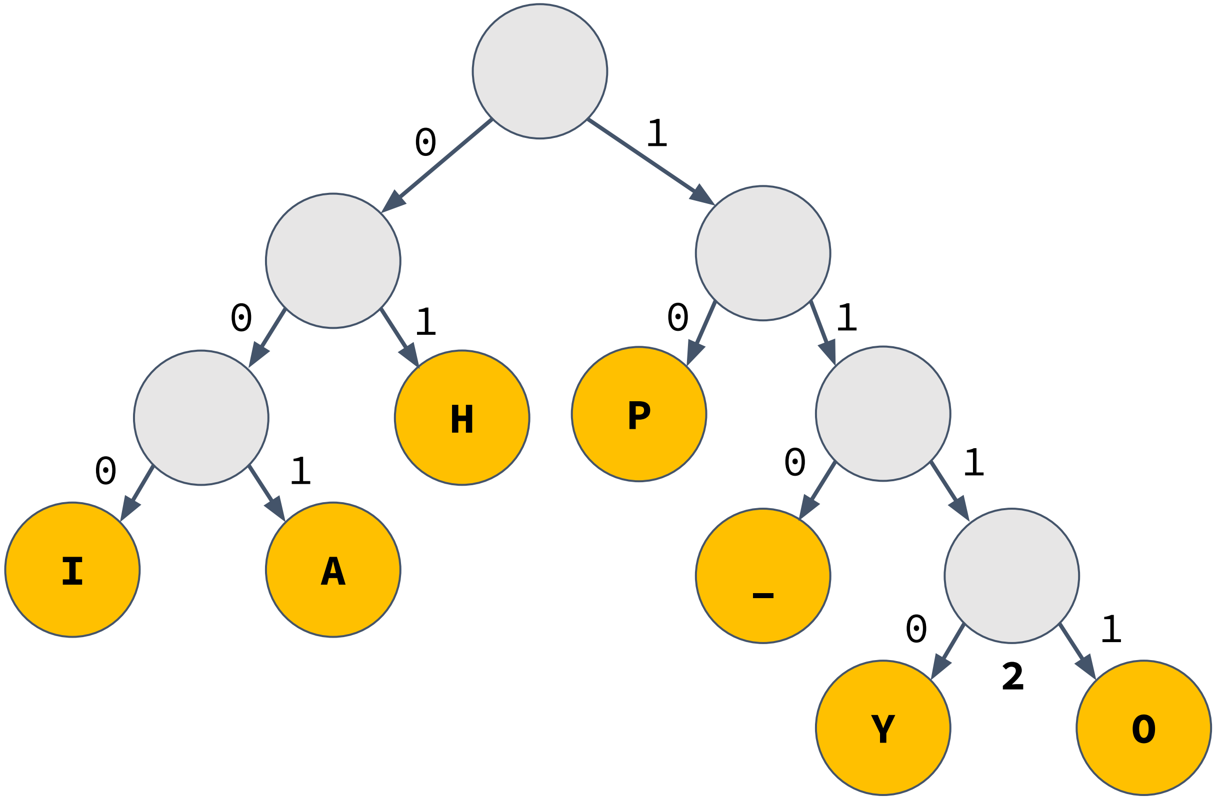
- In order to decode the file, we need to have access to the encoding scheme that was used.
- In the encoded file, prefix the compressed data with a header containing information to rebuild the tree

Info to Rebuild Tree	**111000010110111000010110111000010 1…**

- Option 1: Send the frequencies of the values.
- Option 2: Send a "flattened" tree.

Flattening a Tree

- If a node is a leaf, it is represented as the bit 1.
- If a node is an internal node, it is represented as the bit 0.
- Start at the root node, write its bit representation.
- Follow this with the flattened version of its left subtree and then the flattened version of its right subtree.
- Any time a leaf node is reached, include the ASCII (fixed-length) representation of the value of that node.



00010010010011001000001100100100001001010000…

Unflatten a Tree

001 011 101 001 100 010 011 000

char	*code*
I	000
N	001
E	010
C	011
PEOF	100

Decode a Message

`1101 1100 0110 0000`

Huffman Coding Recap

- In order to support variable-length encodings for data, we must use prefix coding schemes, which can be modeled as binary trees
- Huffman coding constructs encodings by building a tree from the bottom-up, putting the most frequent characters higher up in the coding tree.
- We must send a header with information to reconstruct the tree with the encoded message so that it can be decoded