CSCI 136
Data Structures &
Advanced Programming

Huffman Codes

Algorithm Design
Huffman Codes
(a CS 256 Preview)

Encoding Text

American Standard Code for Information Interchange.

USASCII code chart

				$b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4	b_3	b_2	b_1	Row ↓ \ Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

(courtesy of https://wikimedia.org)

Encoding Text

Extended (8-bit) ASCII

Dec	Symbol	Binary	Dec	Symbol	Binary
65	A	0100 0001	83	S	0101 0011
66	B	0100 0010	84	T	0101 0100
67	C	0100 0011	85	U	0101 0101
68	D	0100 0100	86	V	0101 0110
69	E	0100 0101	87	W	0101 0111
70	F	0100 0110	88	X	0101 1000
71	G	0100 0111	89	Y	0101 1001
72	H	0100 1000	90	Z	0101 1010
73	I	0100 1001	91	[	0101 1011
74	J	0100 1010	92	\	0101 1100
75	K	0100 1011	93	]	0101 1101
76	L	0100 1100	94	^	0101 1110
77	M	0100 1101	95	_	0101 1111
78	N	0100 1110	96	`	0110 0000
79	O	0100 1111	97	a	0110 0001
80	P	0101 0000	98	b	0110 0010
81	Q	0101 0001	99	c	0110 0011
82	R	0101 0010	100	d	0110 0100

(courtesy of https://knowthecode.io)

Binary Encodings

- Normally, use ASCII: 1 character = 8 bits (1 byte)
  - Allows for $2^8 = 256$ different characters
- Space to store "AN_ANTARCTIC_PENGUIN"
  - 20 characters -> 20*8 bits = 160 bits
- Is there a better way?
  - Only 11 symbols are used (ACEGINPRTU_)
  - Only need 4 bits per symbol (since $2^4>11$)!
    - 20*4 = 80 bits instead of 160!

A	C	E	G	I	N	P	R	T	U	_
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010

Can we do better?

Variable-Length Encodings

- Example
  - AN_ANTARCTIC_PENGUIN
  - Compute letter frequencies

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2

- Key Idea: Use fewer bits for most common letters

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011

- Uses 67 bits to encode entire string

Features of Good Encoding

- Letters with lower frequency have longer encodings
- Prefix property: No encoding is a prefix of another encoding

- All optimal length unambiguous encodings have these features

Variable-Length Encodings

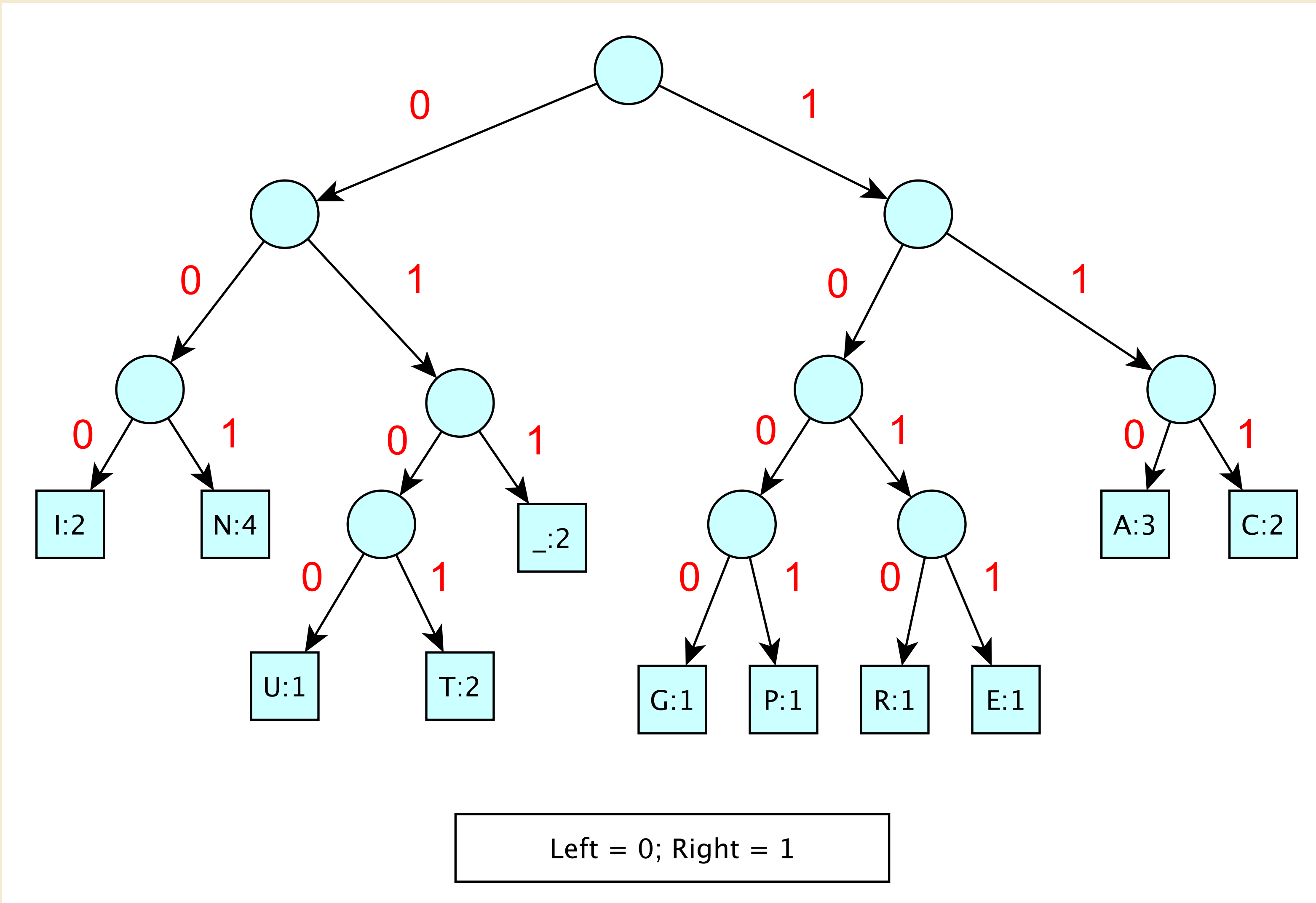
A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011

- Uses 67 bits to encode entire string
- Can we do better?

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
100	010	1100	1101	011	101	0001	0000	001	1110	1111

- Uses 67 bits to encode entire string

A	C	E	G	I	N	P	R	T	U	_
3	2	1	1	2	4	1	1	2	1	2
110	111	1011	1000	000	001	1001	1010	0101	0100	011



Features of Good Encoding

- Leaves with lower frequency have greater depth
- Prefix property: No encoding is a prefix of another encoding (letters only appear at leaves)
- No internal node has a single child

- All optimal length unambiguous encodings have these features
- They are called *Huffman encodings*

Huffman Encoding

- Input: symbols of alphabet with frequencies
- Huffman encode as follows
  - Create a single-node tree for each symbol: key is frequency; value is letter
  - while there is more than one tree
    - Find two trees T_1 and T_2 with lowest keys
    - Merge them into new tree T with key= T_1.key+ T_2.key
      - value of internal node can be anything
- Theorem: The tree computed by Huffman is an optimal encoding for given frequencies

How To Implement Huffman

- Keep a Vector of Binary Trees
- Sort them by decreasing frequency
  - Removing two smallest frequency trees is fast
- Insert merged tree into correct sorted location in Vector
- Running Time:
  - O(n log n) for initial sorting
  - O(n^2) for rest: O(n) re-insertions of merged trees
- Can we do better...?

What Huffman Encoder Needs

- A structure S to hold items with *priorities*
- S should support operations
  - add(E item);   // add an item
  - E removeMin(); // remove min priority item
- S should be designed to make these two operations fast
- If, say, they both ran in O(log n) time, the Huffman algorithm would take O(n log n) time instead of $O(n^2)$!
- Next time: Designing such a structure!