About CS314 Huffman Compressed File (.hf)

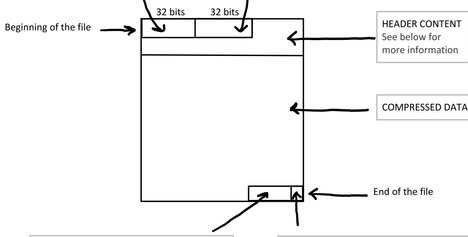
12/3/20 00:01

Structure

MAGIC NUMBER
Why do we include a magic number? We want to make sure this is actually a huffman compressed file before we start decompression. To do this, we chose an arbitrary integer and declared this to be the "Magic Number". We say that any file that starts with the value MUST be a huffman compressed file. Why can we do this? Since an int is 32 bits long and each bit can hold either a 1 or a 0, there is a 2^{32} chance that another file that is not a huffman compressed file will happen to start with the exact same 32 bits as our huffman "magic number". That's a 0.000000000002328% chance that a random file will be mistaken as a huffman compressed file. I like those odds :)

HEADER FORMAT
Why do we need to include a header format constant? We need to know exactly which type of header content is coming next so we can read the information properly and rebuild the huffman code tree correctly.

32 bits = the size of an integer
* Use the BITS_PER_INT constant provided *



PEOF
When we're eventually decompressing this file, it actually can become almost impossible to tell when we've read the last code to decode the last character/byte. To solve this problem, we invent a code for a fake value (256) and use this to signal to our program that all the meaningful bits of the file have been read. If we did not do this, we would likely end up trying to decode the bits used to pad the file (see padding box) and would result in exceptions being thrown in our code.

Extra Padding of 0's
(the BitOutputStream will handle this for you)
Why is there an extra padding of 0s? Computer systems are designed so that all files have to be in 8 bit (1 byte) increments. Depending on the lengths and frequencies of the codes generated in compression, we might be missing a few bits at the end. The BitOutputStream will add extra 0's after you close it and pad the file out to a size that is a multiple of 8 bits.

Header Formats/Content

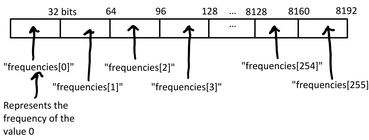
Store Counts Format

There are 256 (2^8) different values we can encode

bits per byte

to remake the tree, we need to know the frequency of each value

We can store them in our header like an array of length 256 (alphabet size)



Note:
We do not include the PEOF frequency in the store counts format because we know for sure that it will always have a frequency of 1. It would be a waste of space to include it. You will, however, have to manually add this frequency to the priority queue yourself when decompressing or you will not generate the proper codes

Store Tree Format

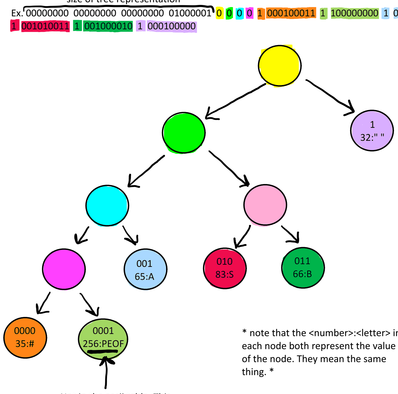
Instead of storing frequencies (because this can take quite a bit of memory when a lot of the values might have frequencies of 0) we can simply pass along the actual huffman code tree!

We can use 0 to represent a non-leaf node
We can use 1 to represent a leaf node which stores a value
- A leaf node represents a value that was compressed
- 9 bits are used to store the value (not frequency) of that tree node
  - Why 9? We have 256 possible values that we read in from the file + 1 extra code for the PEOF. That means there are a total of 257 values that a node could store and this fits into a minimum of 9 bits.

We start by using one int to write out the number of bits that the tree representation takes up ([num leaf nodes * 9] + size of tree). Then the nodes and leaf values are stored using a **pre-order traversal**.

size of tree representation

Ex: 00000000 00000000 00000000 01000001 0 1 0 0 1 000100011 1 100000000 1 001000001 0 1 001010011 1 001000010 1 000100000



* note that the <number>:<letter> in each node both represent the value of the node. They mean the same thing. *

Not in the ascii table. This value is an abstraction that we created!

smallest possible value a file could have

ASCII Table

Dec	Hex	Oct	Char
0	0	0	NUL
1	1	1	
2	2	2	
3	3	3	
4	4	4	
5	5	5	
6	6	6	
7	7	7	
8	8	10	
9	9	11	
10	A	12	
11	B	13	
12	C	14	
13	D	15	
14	E	16	
15	F	17	
16	10	20	
17	11	21	
18	12	22	
19	13	23	
20	14	24	
21	15	25	
22	16	26	
23	17	27	
24	18	30	
25	19	31	
26	1A	32	
27	1B	33	
28	1C	34	
29	1D	35	
30	1E	36	
31	1F	37	
32	20	40	
33	21	41	!
34	22	42	"
35	23	43	#
36	24	44	$
37	25	45	%
38	26	46	&
39	27	47	'
40	28	50	(
41	29	51	)
42	2A	52	*
43	2B	53	+
44	2C	54	,
45	2D	55	-
46	2E	56	.
47	2F	57	/
48	30	60	0
49	31	61	1
50	32	62	2
51	33	63	3
52	34	64	4
53	35	65	5
54	36	66	6
55	37	67	7
56	38	70	8
57	39	71	9
58	3A	72	:
59	3B	73	;
60	3C	74	<
61	3D	75	=
62	3E	76	>
63	3F	77	?
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4A	112	J
75	4B	113	K
76	4C	114	L
77	4D	115	M
78	4E	116	N
79	4F	117	O
80	50	120	P
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5A	132	Z
91	5B	133	[
92	5C	134	\
93	5D	135	]
94	5E	136	^
95	5F	137	_
96	60	140	`
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k
108	6C	154	l
109	6D	155	m
110	6E	156	n
111	6F	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t
117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x
121	79	171	y
122	7A	172	z
123	7B	173	{
124	7C	174	|
125	7D	175	}
126	7E	176	~
127	7F	177	

Extended ASCII Codes

128	Ç	144	É	160	á	176	▒	192	└	208	ð	224	ó	240	≡
129	ü	145	æ	161	í	177	▓	193	┴	209	Ð	225	ß	241	±
130	é	146	Æ	162	ó	178	│	194	┬	210	Ê	226	Ô	242	≥
131	â	147	ô	163	ú	179	┤	195	├	211	Ë	227	Ò	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	È	228	õ	244	⌠
133	à	149	ò	165	Ñ	181	Á	197	┼	213	ı	229	Õ	245	⌡
134	å	150	û	166	ª	182	Â	198	╞	214	Í	230	µ	246	÷
135	ç	151	ù	167	º	183	À	199	╟	215	Î	231	þ	247	≈
136	ê	152	ÿ	168	¿	184	©	200	╚	216	Ï	232	Þ	248	°
137	ë	153	Ö	169	®	185	╣	201	╔	217	┘	233	Ú	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Û	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	Ù	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	ý	252	ⁿ
141	ì	157	¥	173	¡	189	¢	205	═	221	¦	237	Ý	253	²
142	Ä	158	₧	174	«	190	¥	206	╬	222	Ì	238	¯	254	■
143	Å	159	ƒ	175	»	191	┐	207	¤	223	▀	239	´	255	

largest possible value a file could have

Pseudo-code for reading in the store tree header (use recursive helper):

```
Main method:
    Read in 32 bits (BITS_PER_INT) as the size of the tree
    Set the root equal to the result of the recursive method call
Recursive helper:
    Read in 1 bit
    If the bit is 0:
        This is an internal node
        Make a new empty node
        Set the left child to result of call to recursive helper
        Set the right child to result of call the recursive helper
        Return the new node
    If the bit is a 1:
        This is a leaf node
        Read in 9 more bits (this is the value of the node)
        Make a new node w/ this value and no child nodes
        Return the new node
    Else:
        We ran out of bits while trying to form our huffman code tree
        This means something is incorrect about format of the input
        file
        Throw an exception/show an error/report catastrophic failure
```