Project: DATA COMPRESSION

Data Structure

Introduction

This project's purpose is to build a data compression method. That is, we want to convey the same information in a smaller amount of space given particular data. We'll be concentrating on compressing text files for this project, so we'll need to first learn how computers internally represent text.

Computers save information as a series of bytes. A byte is an eight-bit value that ranges from 0 to 255. We need a technique to assign each English letter, punctuation symbol, special character, and other characters to an eight-bit(a value from 0 to 255) sequence in order to represent English text. The ASCII encoding, as stated in the table below, is responsible for this mapping. Notice that ASCII only uses 128 out of the 256 possible values that a byte can store.

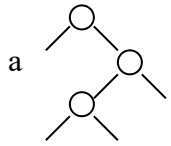
Decimal - Binary - Octal - Hex – ASCII Conversion Chart

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

For instance, consider the text “A b ????”. Using the table above, we can see that this is represented as the following sequence of bytes: “65 32 98 32 63 63 63 63”. Note that the space counts as a character, and its value in the ASCII encoding is 32.If we write out the binary string for each character according to the table above and concatenate them together, we get “01000001001000000110001000100000001111110011111100111111001111111”. Storing our original string with the ASCII encoding requires 8*8 = 64 bits. There are 8 characters in the text “A b ????” and each character is represented by a byte that is 8 bits long.

Now, imagine if we weren’t forced to use eight bits for every character, and we could instead use the binary encoding “? = 0, [space] = 10, A = 110, b = 111”. Then our string “A b ????” would become “11010111100000”. This is only 14 bits, significantly smaller than the 64 bits that ASCII requires. Also notice that none of these codes are a prefix of any others, so there is no ambiguity when decoding. Here, we compressed our string by finding a different encoding for the characters that minimized the number of bits we needed to use.

One of the most important lossless data compression algorithms is called Huffman coding. A Huffman code is defined by a tree, whose leaves are the symbols in the alphabet. For example, the tree



If we indicate going left by 0 and going right by 1, then the Huffman code for the above tree is:

a 0 b 100 c 101 d 11

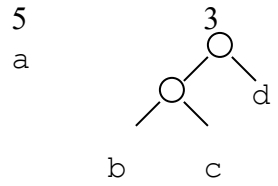
A string of symbols aadbaaca would be encoded as 0011100001010. This would yield compression considering the original requires 8 bits per symbol. So 64 bits have been reduced to 13 bits. The reason we get compression is that the symbol “a” occurs quite frequently in the original and the Huffman code uses just one bit to encode it. There is a simple process to decoding a Huffman code. Start at the root of the tree. If you are at a leaf output the symbol. Otherwise read a bit and go left if is 0 and go right if is 1 and continue in that manner until reaching a leaf. An *optimal Huffman* code is one that produces the shortest code given frequencies for the symbols.

It turns out there is an elegant algorithm for generating an optimal Huffman code. The algorithm uses a priority queue. First you need to calculate the frequency of each symbol in the input. Make a leaf node for each symbol and store its frequency in the node. Repeatedly do the following, find the two trees with the smallest frequencies. Make them the left and right children of a new node whose frequency the sum of the two frequencies. When one tree remains we are done. In the example above the frequencies are a:5, b:1, c:1, d:1.

```
      5    1    1 1  a    b    c d
5          2    1
a                        d
```

b

c



In one more step we are done. The average bit rate of the code can be computed as

$ABR = (F_1L_1 + F_2L_2 + \ldots + F_mL_m)/N$

where F_i is the frequency of the i'th symbol, L_i is the length of the code for the i'th symbol and N is the length of the file. Without actually compressing the file the compression ratio for a text file can be computed as 8/ABR. This is because the uncompressed tex file is stored with 8 bits per symbol.

Task No. 1:
Build simple Tree based Huffman coding scheme and show the results.

Task No. 2:
The second task in this project is to use predefined priority queues to build an optimal Huffman tree. Your priority queue will maintain the current set of trees ordered by their frequencies. One challenge is to efficiently traverse the optimal Huffman tree to generate the code to be printed out.

The User Interface

Again we are not too concerned with the user interface because we are working with the data structures. The program requests a text file name, then computes the optimal Huffman code and prints it. The compression ratio is then printed.

> Please enter a text file name:
foo.txt
> The optimal Huffman code is:
> a 0
> b 100
> c 101
> d 11
> The compression ratio is:
> 4.92

Your program should check for invalid inputs.