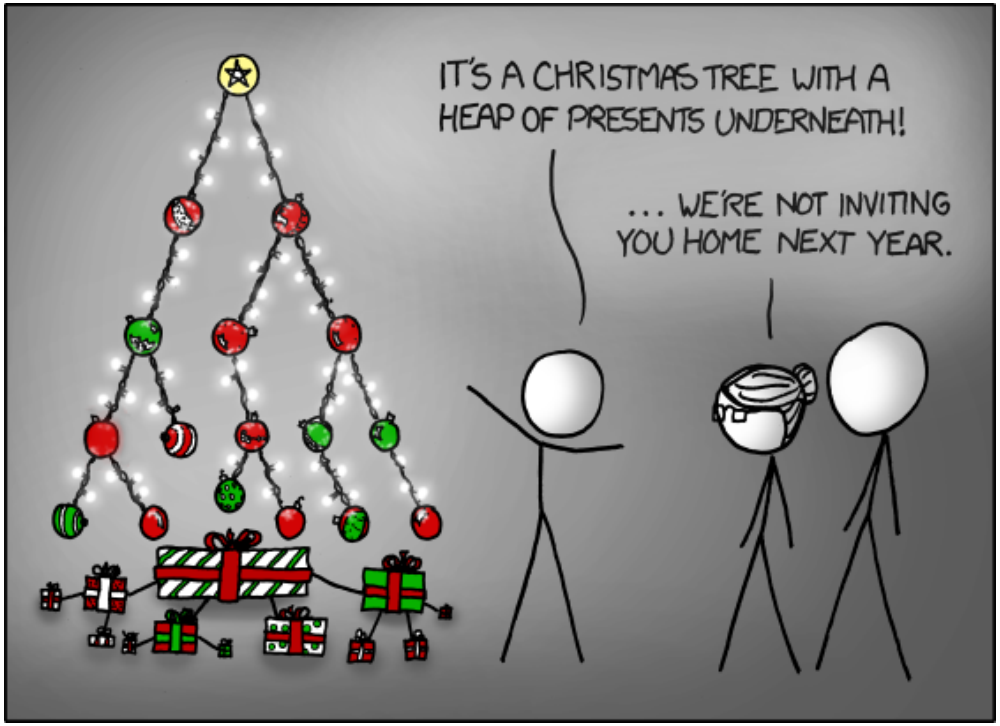
A6 - Huffman Encoding YEAH Hours



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

Problem

48 characters

ataata -> 01100001 01110100 01100001 01100001 01110100 01100001

a a a a

Thought: Let's represent 'a' with less characters!

Proposed Solution

Let's arbitrarily represent 'a' with `01`

24 characters!

```
ataata -> 01 01110100 01 01 01110100 01
          a           a  a           a
```

Why did we choose 'a'?

How do we scale this?

Huffman encoding

Uses variable lengths for different characters to take advantage of their relative frequencies.

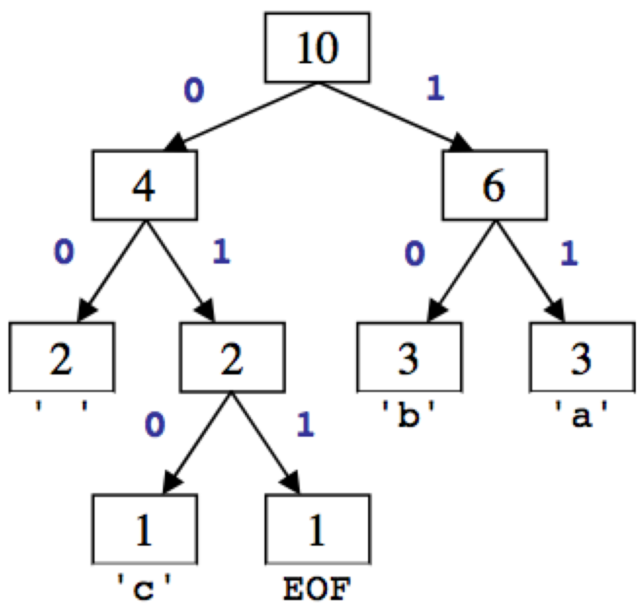
Char	ASCII value	ASCII (binary)	Hypothetical Huffman
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	01110100
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011110

Huffman Tree

file.txt

bac aab b

Frequencies: {' ':2, 'a':3, 'b':3, 'c':1, **EOF:1**}

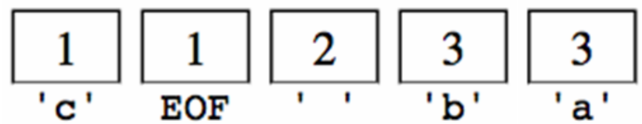


Huffman compression

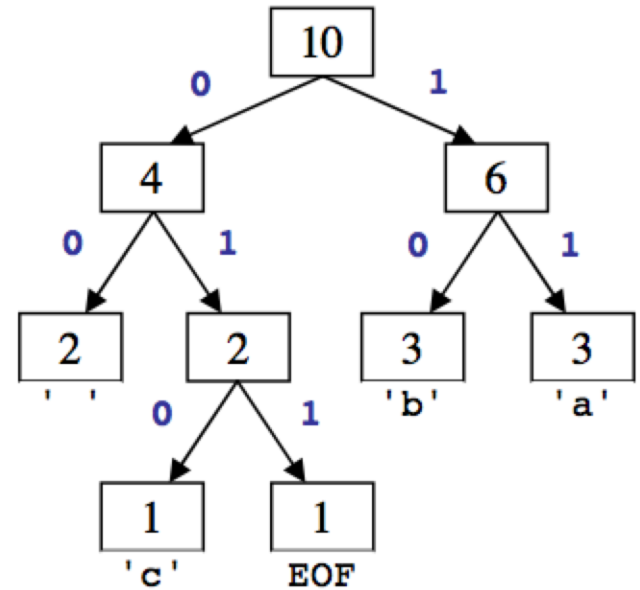
1. **Count** occurrences of each char in file

`{' ':2, 'a':3, 'b':3, 'c':1, EOF:1}`

2a. Place chars, counts into **priority queue**



2b. Use PQ to create **Huffman tree** →



3. Write logic to free the tree!

4. Traverse tree to find (char → binary) **encoding map**

`{' ':00, 'a':11, 'b':10, 'c':010, EOF=011}`

5. **Convert** to binary (For each char in file, look up binary rep in map)

`11 10 00 11 10 00 010 1 1 10 011 00`

(1) Count occurrences

```
Map<int, int> buildFrequencyTable(istream& input)
```

Take as input an **istream** containing the file to compress, then hands back a **Map** associating each character in the file with its frequency.

bac aab b

```
{' ':2, 'a':3, 'b':3, 'c':1, EOF:1}
```

(2) Build Huffman Tree

```
HuffmanNode* buildEncodingTree(Map<int, int> freqTable)
```

Take as input a **Map** associating each character in the file with its frequency containing the file to compress, then hands back a Huffman encoding tree

HuffmanNode

```
HuffmanNode* {
  int character;        // character being represented by this node
  int count;            // number of occurrences of that character
  HuffmanNode* zero;    // 0 (left) subtree (nullptr if empty)
  HuffmanNode* one;     // 1 (right) subtree (nullptr if empty)
}
```

The character field is declared as type `int`, but you should think of it as a `char`. The character field can take one of three types of values:

- `char` value
- `PSEUDO_EOF` which represents the pseudo-EOF value
- `NOT_A_CHAR` which represents something that isn't actually a character

(2a) Place occurrences into PQueue

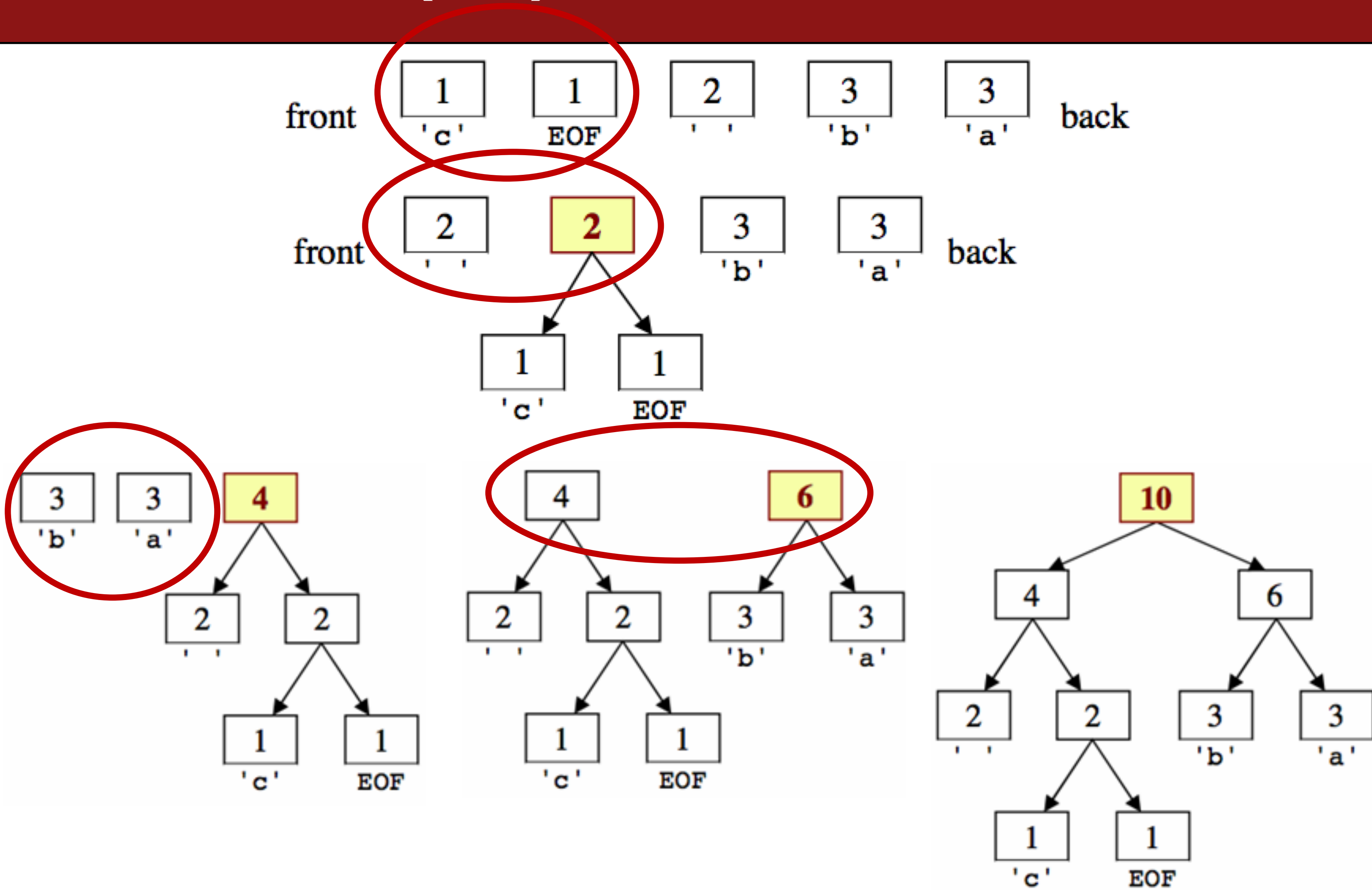
Stanford PQueue

```
pq.enqueue(value, priority)
```

```
Map: {' ':2, 'a':3, 'b':3, 'c':1, EOF:1}
```

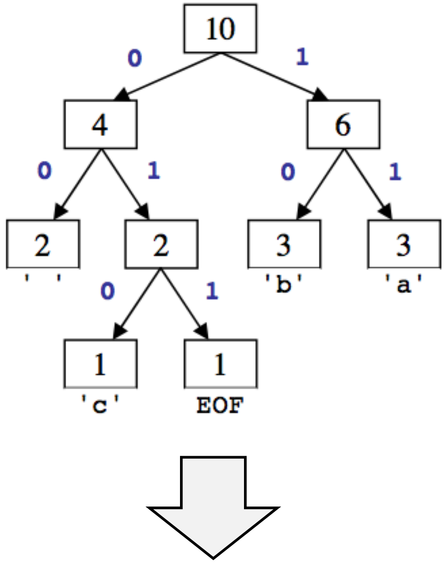
```
PQueue: [ EOF:1, 'c':1, ' ':2, 'a':3, 'b':3]
```

(2b) Build tree



(3) Tree to binary encodings

- The Huffman tree tells you the binary encodings to use.
  - example: `'b'` is `10`
  - example: `'c'` is `010`



`{' ':00, 'a':11, 'b':10, 'c':010, EOF:011}`

(4) Encode the file

```
void encodeData(istream input,
                Map<int, string> encodingMap,
                obistream output)
```

Take as input an **istream** of text to compress, a **Map** associating each character with the bit sequence to use to encode it, then writes everything to the **obitstream**

obitstream

`obitstream`: Writes one bit at a time to output.

`void` **`writeBit`**`(int bit)`	Writes a single bit (must be 0 or 1)

- `obitstream` also contains the members from `ostream`.
  - open, read, write, fail, close

(4) Encode the file

- Based on the preceding tree, we have the following encodings:

  `{' ':00, 'a':11, 'b':10, 'c':010, EOF:011}`

  – The text "**ab ab cab**" would be encoded as:

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	11	10	00	11	10	00	010	11	10	011

  – Overall: **1110001110000101110011**, (22 bits, ~3 bytes)

byte	1	2	3
char	a b a	b c a	b EOF
binary	11 10 00 11	10 00 010 1	1 10 011 00

Decompressing

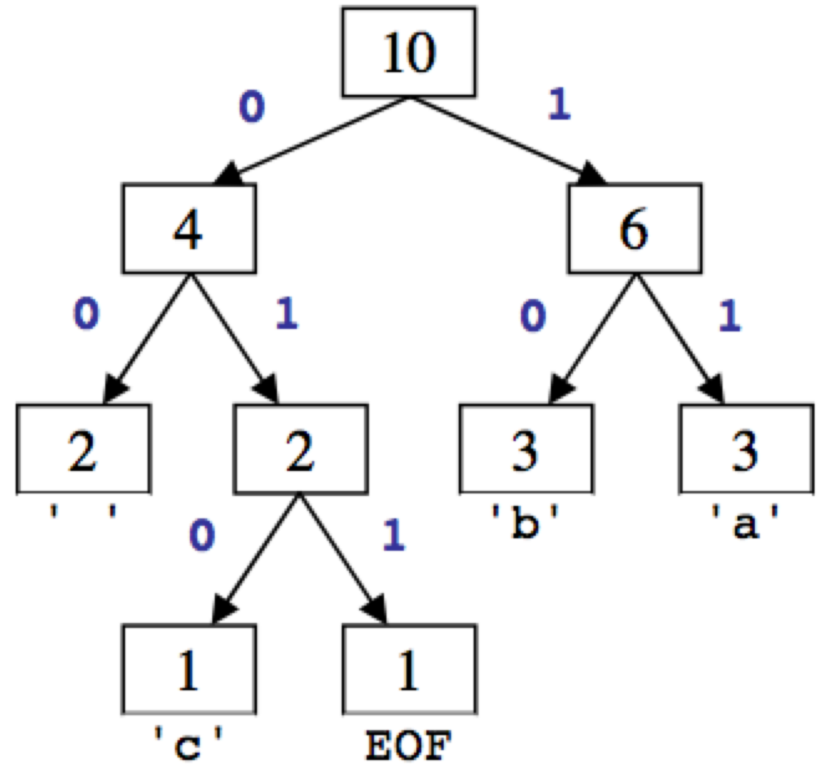
Wait… Don't you need delimiters?!?

```
1011010001101011011
 b a  c _ a  c a
```

– Read each bit one at a time.

– If it is 0, go left; if 1, go right.

– If you reach a leaf, output the character there and go back to the tree root.



- Output:

```
bac aca
```

Decompressing II

How do we know what the map is for decompressing?

Include the mapping in the file itself!

{32:2, 97:3, 98:3, 99:1, 256:1}

Hint: Maps can easily be read and written to/from streams using << and >> operators

(5) Decode the file

```
void decodeData(ibitstream input,
                HuffmanNode* encodingTree,
                ostream out)
```

Take as input an **ibitstream** of bits, a pointer **encodingTree** to the root of an encoding tree, then writes everything to **out**

ibitstream

`ibitstream`: Reads one bit at a time from input.

`int` **`readBit()`**	Reads a single 1 or 0; returns -1 at end of file

- `ibitstream` also contains the members from `istream`.
  - open, read, write, fail, close

Putting it all together

```
void compress(istream& input, obitstream& output)
```

This is the overall compression function; in this function you should compress the given input file into the given output file. You will take as parameters an input file that should be encoded and an output bit stream to which the compressed bits of that input file should be written. You should:

- Read the input file one character at a time,
- Build an encoding of its contents
- Write a compressed version of that input file, including a **header**, to the specified output file.

This function should be built on top of the other encoding functions and should call them as needed

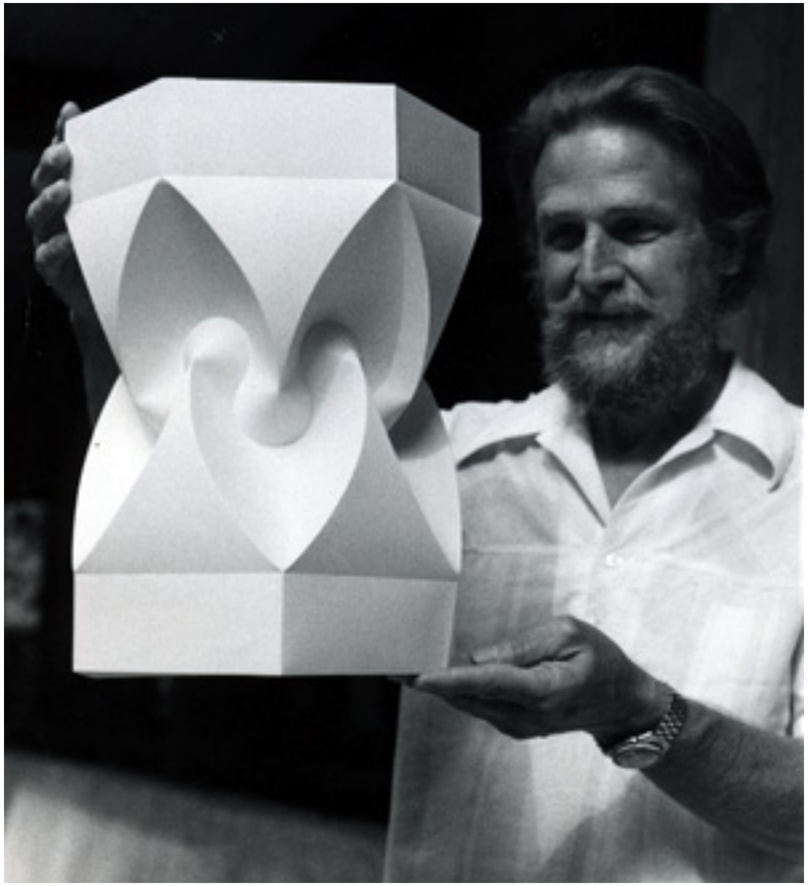
Putting it all together II

`void decompress(ibitstream& input, ostream& output)`

This function should do the opposite of compress;

- Read the bits from the given input file one at a time, including your header packed inside the start of the file
- Write the original contents of that file to the file specified by the output parameter.

Good luck Huffman encoding!



David A. Huffman