Huffman, YEAH!

Sasha Harrison
Spring 2018

Overview

- Brief History Lesson
- Step-wise Assignment Explanation
- Starter Files, Debunked

What is Huffman Encoding?

- File compression scheme
- In text files, can we decrease the number of bits needed to store each character?

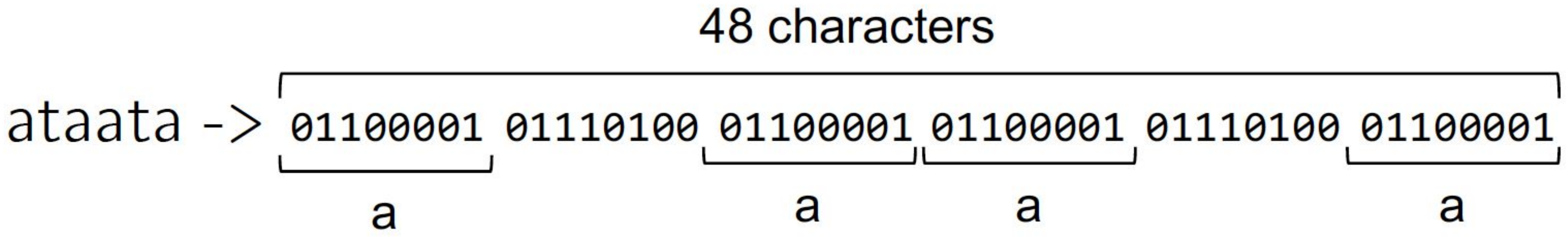
Intuition:

File 1	File 2
"ataata"	"CS106B is the best class ever, we love computer science!"
SMALLER	LARGER

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					

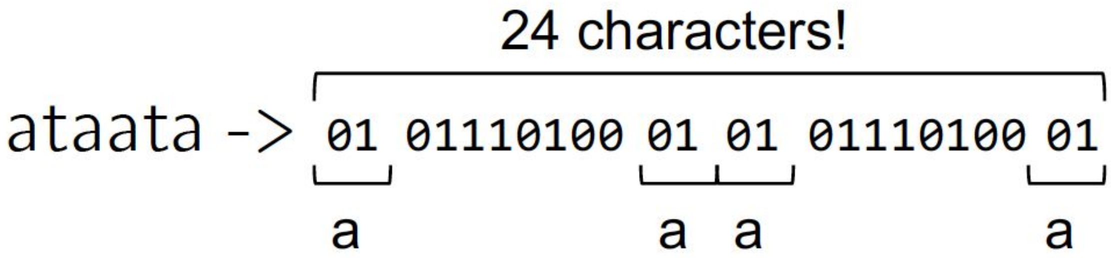
What is Huffman Encoding?



What if we could represent **'a'** in fewer than 8 bits?

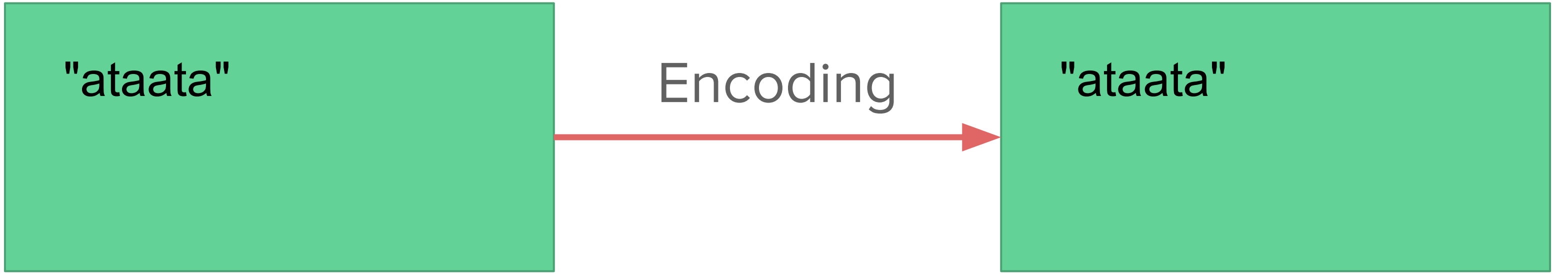
What is Huffman Encoding?

Let's arbitrarily use **01** to represent **'a'**



This is much shorter!

What is Huffman Encoding?



Original File size: 48 bits

New File size: 24 bits

How do we scale this to all characters, not just **'a'** ?

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

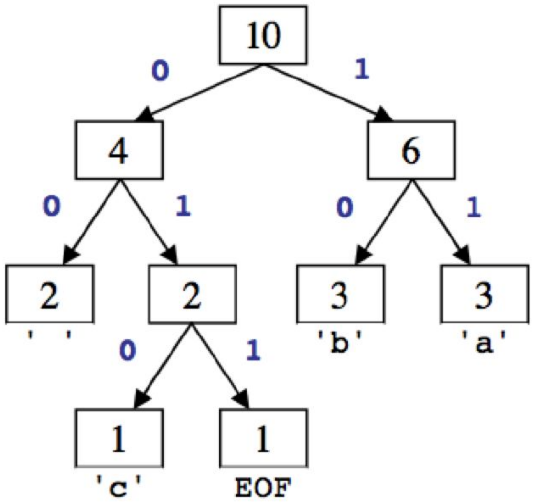
Huffman Encoding is a 5 Step Process

Huffman Tree

file.txt

bac aab b

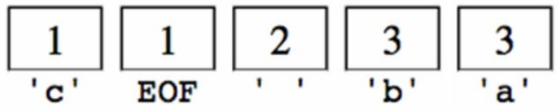
Frequencies: {' ':2, 'a':3, 'b':3, 'c':1, **EOF:1**}



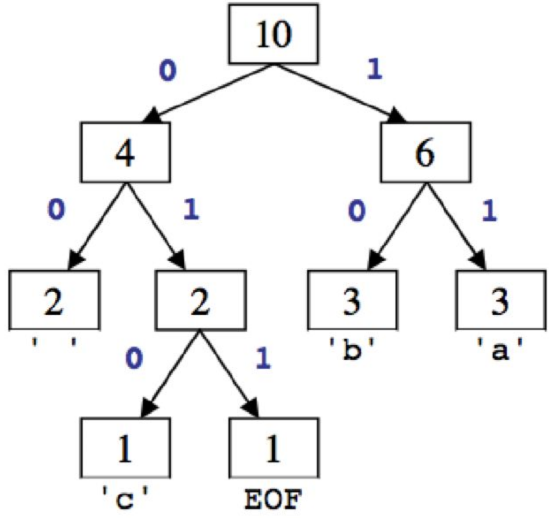
1. **Count** occurrences of each char in file

`{' ':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

Step 1: Count Occurrences

"bac aab a"

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

Step 1: Count Occurrences

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

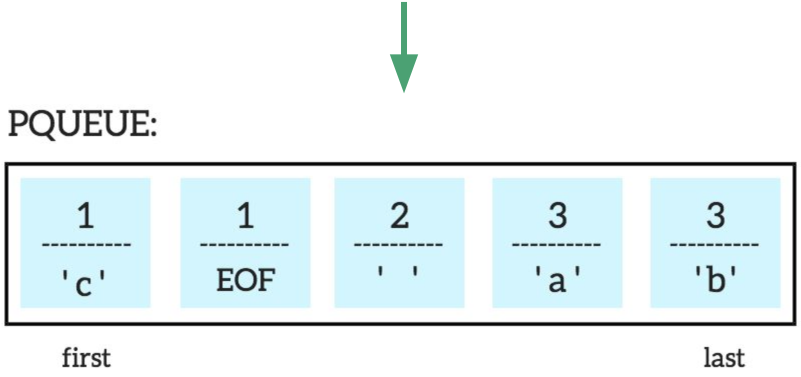
Takes as input an istream containing the file to compress, returns a Map<int, int> associating each character in the file with its frequency.

"bab aab c" → **{ ' ' : 2, 'b' : 3, 'a' : 3, 'c': 1, EOF : 1 }**

Step 2a: Sort Characters By Frequency

Key Idea: Use a PQueue of Huffman Nodes to sort characters based on their frequency.

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



Step 2a: Sort Characters By Frequency

What is a Huffman Node? Struct provided in the starter code

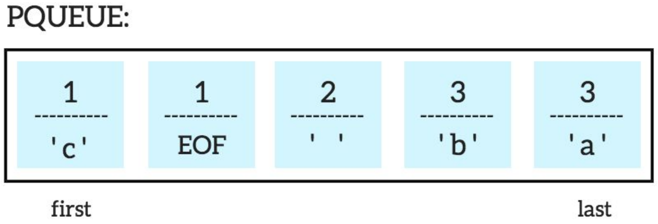
```
HuffmanNode* {
  int character;        // character being represented by this node
  int count;            // number of occurrences of that character
  HuffmanNode* zero;    // 0 (left) subtree (nullptr if empty)
  HuffmanNode* one;     // 1 (right) subtree (nullptr if empty)
}
```

Step 2a: Sort Characters By Frequency

```
HuffmanNode* {
  int character;
  int count;
  HuffmanNode* zero;
  HuffmanNode* one;
}
```

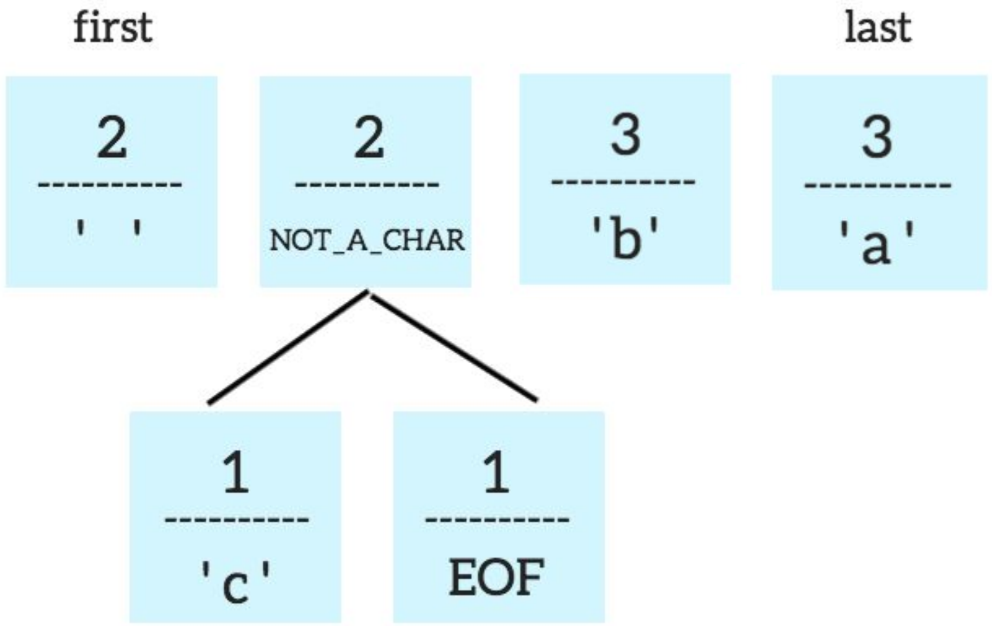
➔ The character field has type "int", but you should just think of it as a char.
It has three possible values:
  - **char** value - regular old character.
  - PSEUDO_EOF - represent the pseudo-eof value
  - NOT_A_CHAR - represents something that's not a character

Step 2b: Build a binary Tree using the PQueue

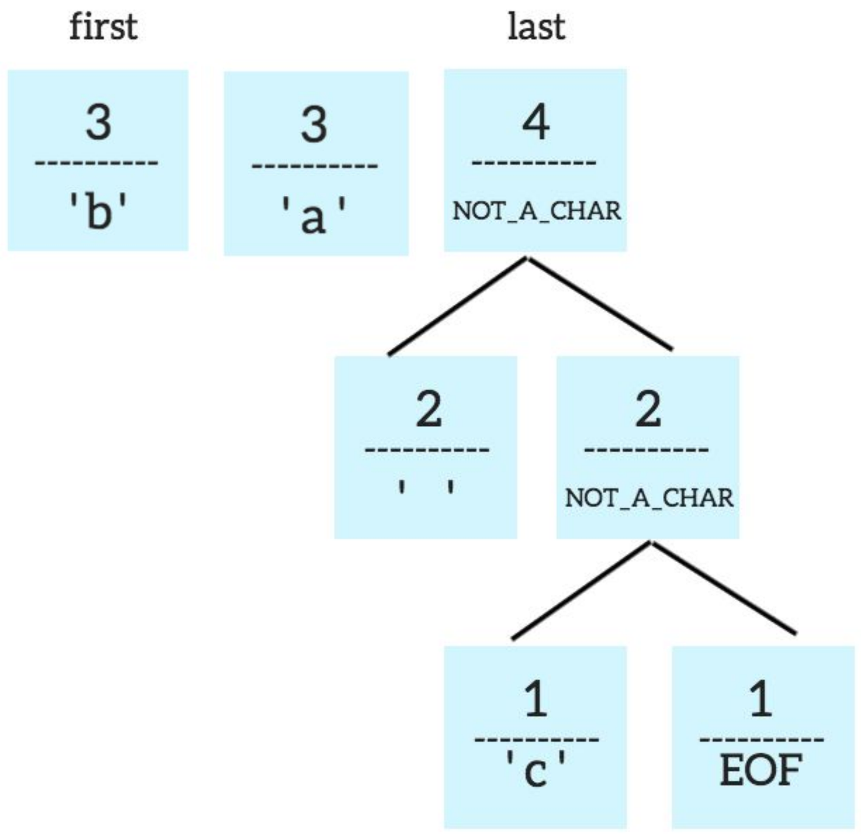


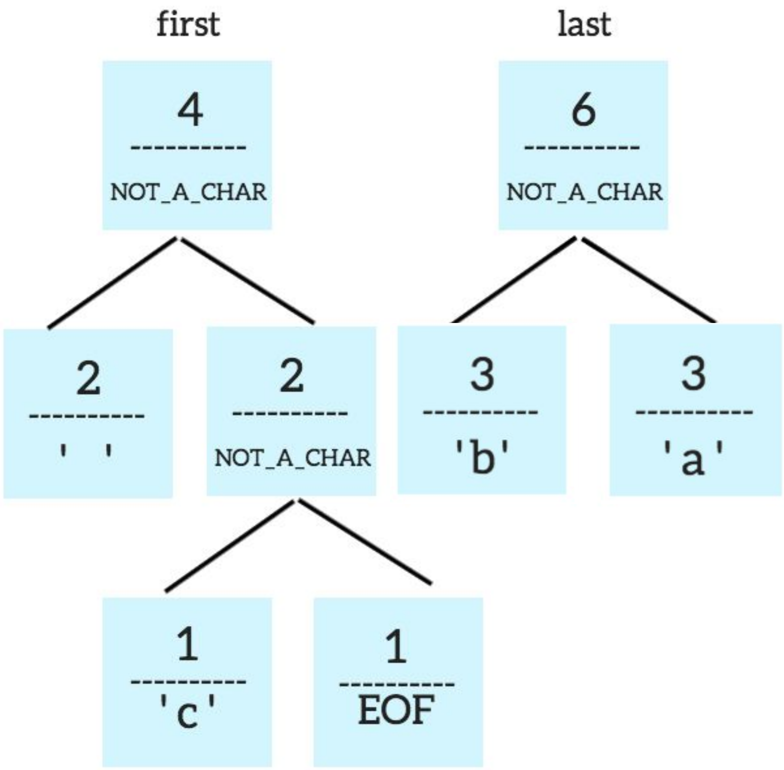
Procedure:

1. Remove two nodes from the front of the queue
2. Create a new node, whose frequency is their sum, and whose character field is NOT_A_CHAR
3. Add the two dequeued nodes as children of this new node.
   a. First dequeued is left child
   b. Second dequeued is right child
4. Reinsert the parent node into the PQueue
5. Repeat until the queue contains only tree root.

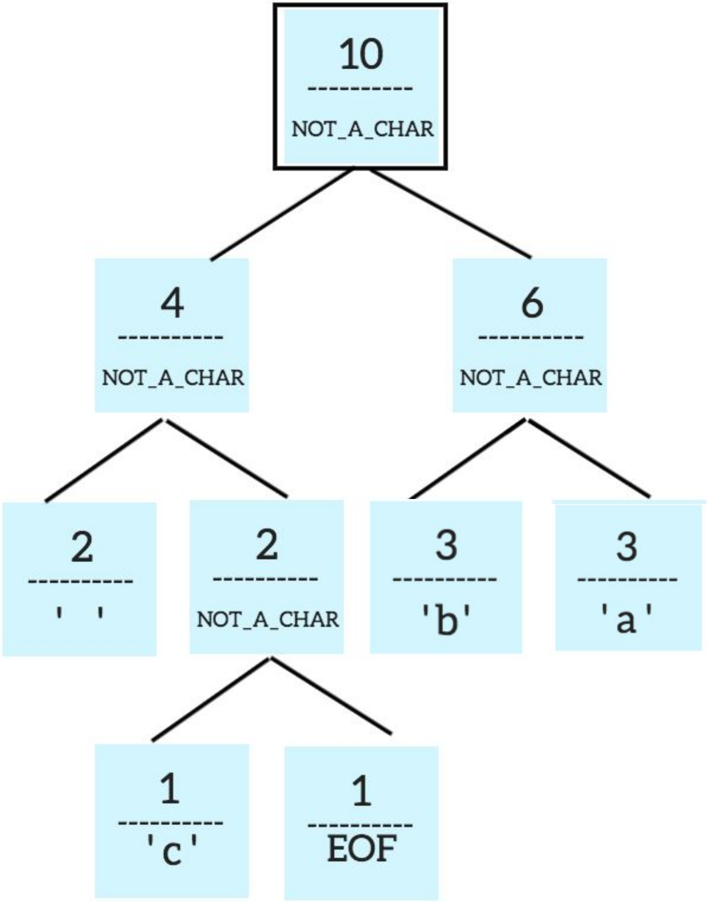


first
last
2
' '
2
NOT_A_CHAR
3
'b'
3
'a'
1
'c'
1
EOF


first
last
3
'b'
3
'a'
4
NOT_A_CHAR
2
' '
2
NOT_A_CHAR
1
'c'
1
EOF


first
last
4
NOT_A_CHAR
6
NOT_A_CHAR
2
' '
2
NOT_A_CHAR
3
'b'
3
'a'
1
'c'
1
EOF

PQUEUE:

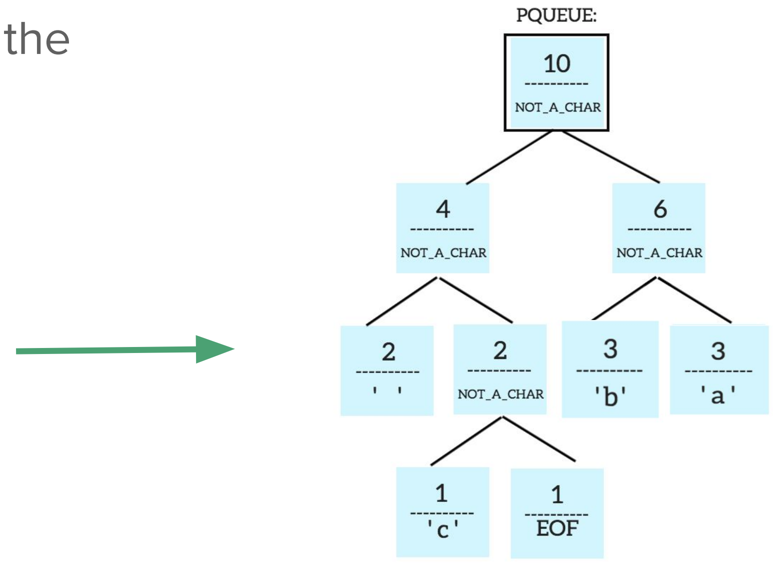


Step 2b: Build a binary Tree using the PQueue

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

Takes map of frequencies as input, returns the HuffmanNode* pointing to the root of the encoding tree.

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



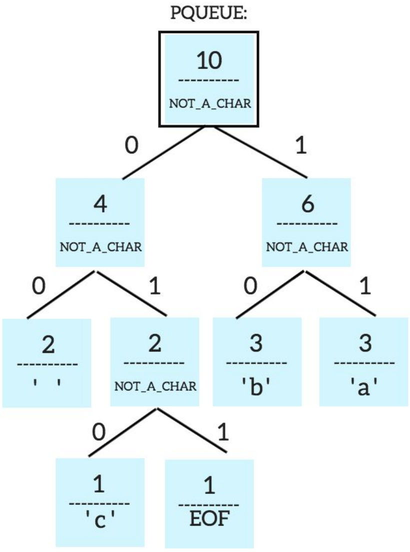
Step 3: Use Tree to Determine Encodings

The Huffman Tree tells you the encodings to use for each character.

Example: 'b' is **1 0**

Example: 'c' is **0 1 0**

Hint: Create an "encoding map", Map<int,string> mapping characters to their new encodings



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

Step 4: Encode the File

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

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

Step 4: Encode the File

`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

Step 4: Convert to binary

- 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

That's all for compression!

Step 5: How about Decompressing?

Wait, don't you need delimiters??

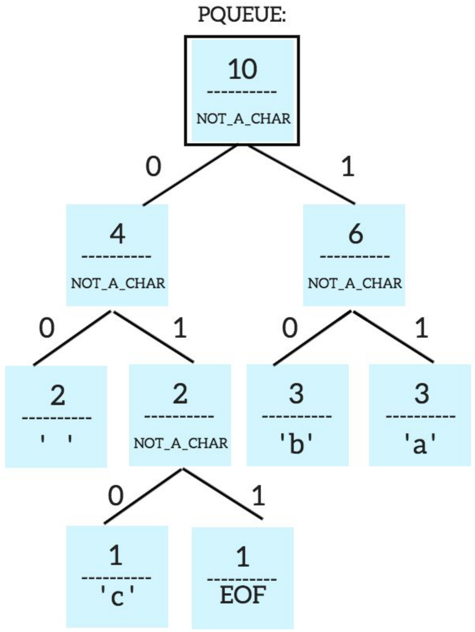
10110100011010111011 → 10110100011010111011

b a c _ a c a

Procedure:

- ➔ Read one bit at a time
- ➔ If 0, go left, if 1, go right.
- ➔ If you reach a leaf, print out the character that maps to the bits you read. Then, go back to the root of the tree.

Output: bac aca



Step 5: How about decompressing?

For a given file, how do we know what the mapping is?

➔ **We include the mapping in the file.**

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

You can easily read/write a map to streams using the << and >> operators.

Header: When you write your compressed file, write the contents of the map into the **obitstream** before you write the file contents.

Putting it all together

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

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

`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)
```

TL;DR: Chain together all the functions you wrote to make one function that does the whole 5 step compression process.

It should compress the given input file, and write the resulting bits into the given output file.

Putting it all together

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

This should do the exact 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.

Optional Extension: MyMap Class

- ➔ If you're interested in going above and beyond, one cool extension would be to define your own map class that mimics a HashMap
  - More info for difference between Maps and HashMaps: Here and Here
- ➔ What are the advantages of a HashMap?
  - O(1) lookup and O(1) deletion, on average (that's V fast)
- ➔ You can then use this map that you defined to store the character frequencies and Huffman encodings!

Files to define:

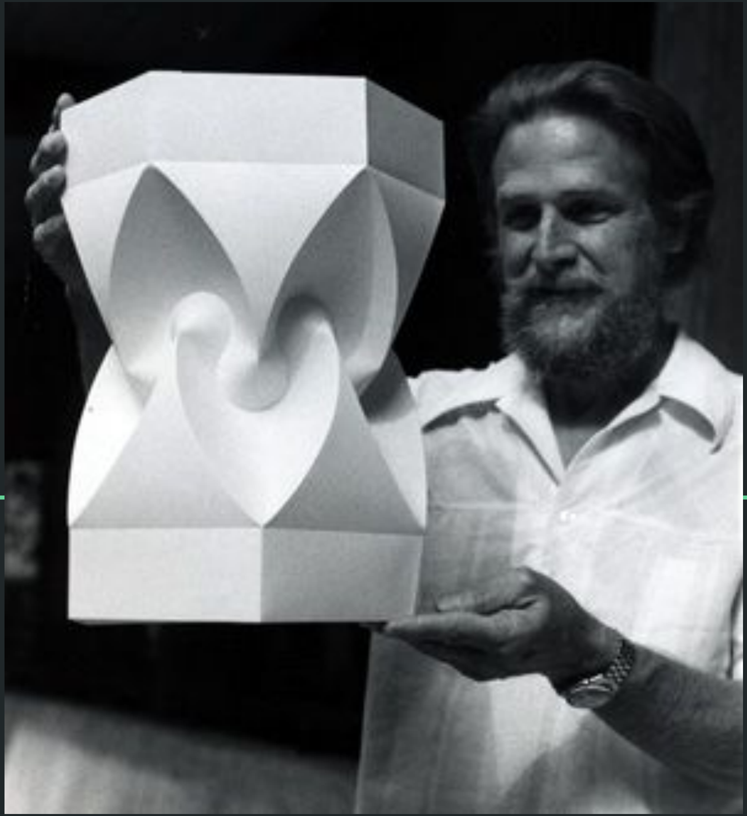
- mymap.cpp
- mymap.h

Optional Extension: MyMap Class

General idea:

- ➔ Create a struct to store key value pairs (both of type 'int')
- ➔ As a private member variable, store an array of buckets, where each bucket is the head of a linked list of key value pairs
- ➔ Define a hash function that deterministically gives you a bucket into which the key value pair should be places
    - ◆ More info on hash functions here

Go Encode!



David A. Huffman