CS 10C - INTRO TO DATA STRUCTURES AND ALGORITHMS, RUSICH

HUFFMAN CODING

HUFFMAN CODING

- Created by Computer Scientist *David A. Huffman* in 1951 at MIT in a graduate EE course on Information Theory.
- Professor Robert M. Fano assigned as a term paper alternate to the final exam. Fano and Claude Shannon had worked on shortest prefix problem.
- Problem: find the most efficient method of representing numbers, letters or other symbols using a binary code.
- Huffman, 25 years old at the time, nearly gave up to begin study for the final.
- Epiphany - when he crumpled his notes to toss into to the waste bin, the solution was was revealed “It was the most singular moment of my life,” Huffman says. “There was the absolute lightning of sudden realization.”

https://www.huffmancoding.com/my-uncle/scientific-american

MORSE CODE

- Method to encode text, used to transmit message across communication line (telegraph).
- Case insensitive - letter 'A' treated same as 'a'
  - 1 dot = 1 unit
  - 1 dash = 3 units
  - space between dots and dashes in a letter = 1 unit
  - space between letters = 3 units
  - space between words = 7 units
- Letters that occur more frequently have shorter encodings.
  - Example: letter 'e' most common letter used in English, has a morse code length of 1.

A	● ▬
B	▬ ● ● ●
C	▬ ● ▬ ●
D	▬ ● ●
E	●
F	● ● ▬ ●
G	▬ ▬ ●
H	● ● ● ●
I	● ●
J	● ▬ ▬ ▬
K	▬ ● ▬
L	● ▬ ● ●
M	▬ ▬
N	▬ ●
O	▬ ▬ ▬
P	● ▬ ▬ ●
Q	▬ ▬ ● ▬
R	● ▬ ●
S	● ● ●
T	▬
U	● ● ▬
V	● ● ● ▬
W	● ▬ ▬
X	▬ ● ● ▬
Y	▬ ● ▬ ▬
Z	▬ ▬ ● ●
1	● ▬ ▬ ▬ ▬
2	● ● ▬ ▬ ▬
3	● ● ● ▬ ▬
4	● ● ● ● ▬
5	● ● ● ● ●
6	▬ ● ● ● ●
7	▬ ▬ ● ● ●
8	▬ ▬ ▬ ● ●
9	▬ ▬ ▬ ▬ ●
0	▬ ▬ ▬ ▬ ▬

NAIVE ENCODING OF TEXT

- ASCII - American Standard Code for Information Interchange
    - character encoding for storage and transmission of text
    - 7 bits per character: [0-9], [A-Z], [a-z] , space
    - '_' denotes a space

`do_or_do_not_there_is_no_try`

1100100 1101111 0100000 1101111 1110010
0100000 1100100 1101111 0100000 1101110
1101111 1110100 0100000 1110100 1110100
1100101 1110010 1100101 0100000 1101001
1110011 0100000 1101110 1101111 0100000
1110100 1110010 1111001

ASCII TABLE

Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char
0	00	000	0000000	NUL (null character)	32	20	040	0100000	space	64	40	100	1000000	@	96	60	140	1100000	`
1	01	001	0000001	SOH (start of header)	33	21	041	0100001	!	65	41	101	1000001	A	97	61	141	1100001	a
2	02	002	0000010	STX (start of text)	34	22	042	0100010	"	66	42	102	1000010	B	98	62	142	1100010	b
3	03	003	0000011	ETX (end of text)	35	23	043	0100011	#	67	43	103	1000011	C	99	63	143	1100011	c
4	04	004	0000100	EOT (end of transmission)	36	24	044	0100100	$	68	44	104	1000100	D	100	64	144	1100100	d
5	05	005	0000101	ENQ (enquiry)	37	25	045	0100101	%	69	45	105	1000101	E	101	65	145	1100101	e
6	06	006	0000110	ACK (acknowledge)	38	26	046	0100110	&	70	46	106	1000110	F	102	66	146	1100110	f
7	07	007	0000111	BEL (bell (ring))	39	27	047	0100111	'	71	47	107	1000111	G	103	67	147	1100111	g
8	08	010	0001000	BS (backspace)	40	28	050	0101000	(	72	48	110	1001000	H	104	68	150	1101000	h
9	09	011	0001001	HT (horizontal tab)	41	29	051	0101001	)	73	49	111	1001001	I	105	69	151	1101001	i
10	0A	012	0001010	LF (line feed)	42	2A	052	0101010	*	74	4A	112	1001010	J	106	6A	152	1101010	j
11	0B	013	0001011	VT (vertical tab)	43	2B	053	0101011	+	75	4B	113	1001011	K	107	6B	153	1101011	k
12	0C	014	0001100	FF (form feed)	44	2C	054	0101100	,	76	4C	114	1001100	L	108	6C	154	1101100	l
13	0D	015	0001101	CR (carriage return)	45	2D	055	0101101	-	77	4D	115	1001101	M	109	6D	155	1101101	m
14	0E	016	0001110	SO (shift out)	46	2E	056	0101110	.	78	4E	116	1001110	N	110	6E	156	1101110	n
15	0F	017	0001111	SI (shift in)	47	2F	057	0101111	/	79	4F	117	1001111	O	111	6F	157	1101111	o
16	10	020	0010000	DLE (data link escape)	48	30	060	0110000	0	80	50	120	1010000	P	112	70	160	1110000	p
17	11	021	0010001	DC1 (device control 1)	49	31	061	0110001	1	81	51	121	1010001	Q	113	71	161	1110001	q
18	12	022	0010010	DC2 (device control 2)	50	32	062	0110010	2	82	52	122	1010010	R	114	72	162	1110010	r
19	13	023	0010011	DC3 (device control 3)	51	33	063	0110011	3	83	53	123	1010011	S	115	73	163	1110011	s
20	14	024	0010100	DC4 (device control 4)	52	34	064	0110100	4	84	54	124	1010100	T	116	74	164	1110100	t
21	15	025	0010101	NAK (negative acknowledge)	53	35	065	0110101	5	85	55	125	1010101	U	117	75	165	1110101	u
22	16	026	0010110	SYN (synchronize)	54	36	066	0110110	6	86	56	126	1010110	V	118	76	166	1110110	v
23	17	027	0010111	ETB (end transmission block)	55	37	067	0110111	7	87	57	127	1010111	W	119	77	167	1110111	w
24	18	030	0011000	CAN (cancel)	56	38	070	0111000	8	88	58	130	1011000	X	120	78	170	1111000	x
25	19	031	0011001	EM (end of medium)	57	39	071	0111001	9	89	59	131	1011001	Y	121	79	171	1111001	y
26	1A	032	0011010	SUB (substitute)	58	3A	072	0111010	:	90	5A	132	1011010	Z	122	7A	172	1111010	z
27	1B	033	0011011	ESC (escape)	59	3B	073	0111011	;	91	5B	133	1011011	[	123	7B	173	1111011	{
28	1C	034	0011100	FS (file separator)	60	3C	074	0111100	<	92	5C	134	1011100	\	124	7C	174	1111100	|
29	1D	035	0011101	GS (group separator)	61	3D	075	0111101	=	93	5D	135	1011101	]	125	7D	175	1111101	}
30	1E	036	0011110	RS (record separator)	62	3E	076	0111110	>	94	5E	136	1011110	^	126	7E	176	1111110	~
31	1F	037	0011111	US (unit separator)	63	3F	077	0111111	?	95	5F	137	1011111	_	127	7F	177	1111111	DEL

CAN WE DO BETTER?

do_or_do_not_there_is_no_try

d	o	SPACE	o	r	SPACE	d	o	SPACE	n	o	t
1100100	1101111	0100000	1101111	1110010	0100000	1100100	1101111	0100000	1101110	1101111	1110100

SPACE	t	h	e	r	e	SPACE	i	s	SPACE	n	o
0100000	1110100	1110100	1100101	1110010	1100101	0100000	1101001	1110011	0100000	1101110	1101111

SPACE	t	r	y
0100000	1110100	1110010	1111001

- Fixed length bit strings - 28 * 7 bits = 196 bits (actually need 8 bits for ASCII, so 224 bits)
- What if we used bit strings of variable lengths?
- Encode high frequency characters with shortest bit strings.
- Huffman coding does this.

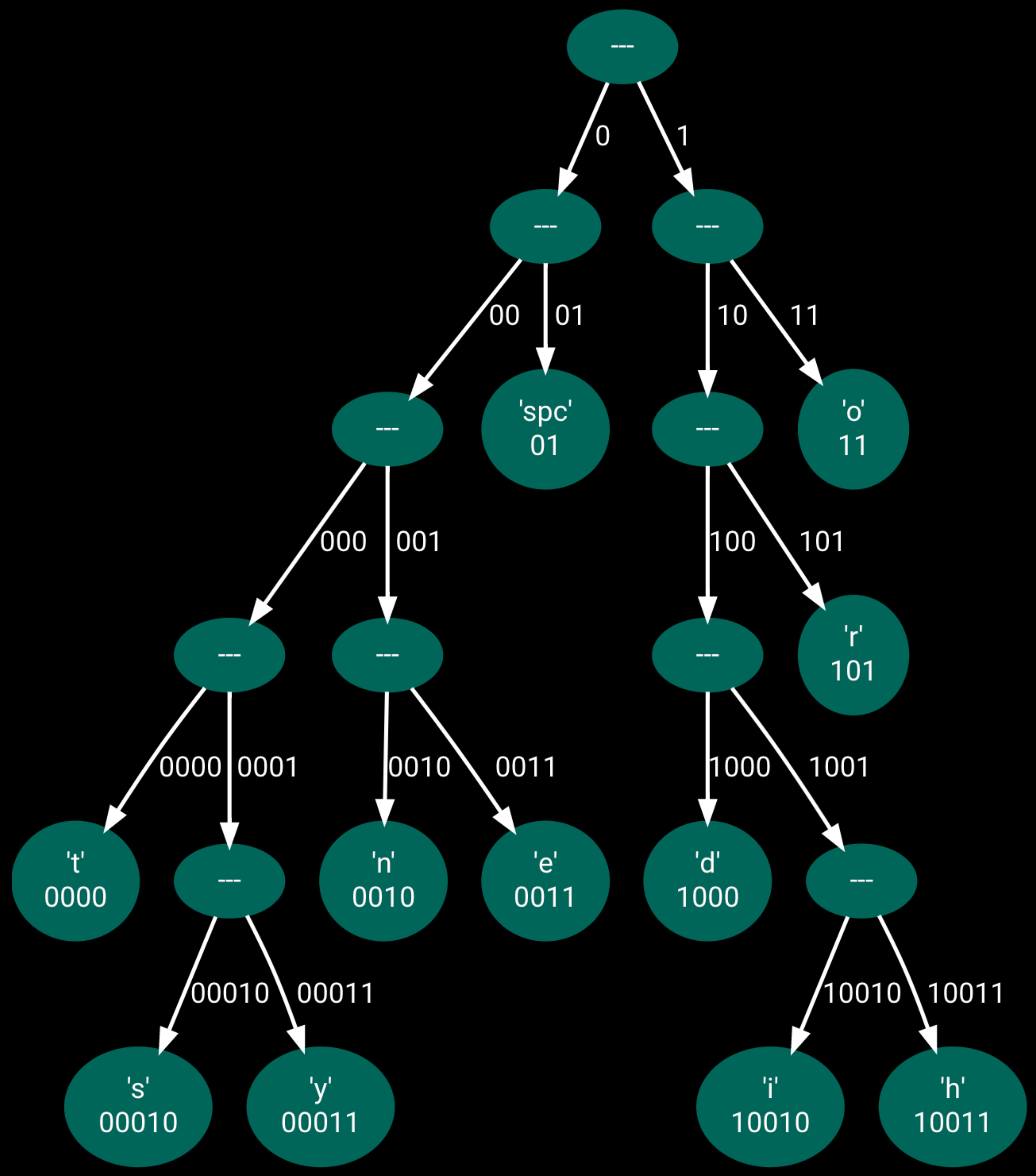
CALCULATE THE CHARACTER FREQUENCIES

- Count the number of instances of each character in the file to be compressed.
- Huffman coding will use longer bit streams to represent characters that occur less frequently.

letter	frequency	percentage
space	7	0.25000000
o	5	0.17857143
t	3	0.10714286
r	3	0.10714286
d	2	0.07142857
n	2	0.07142857
e	2	0.07142857
h	1	0.03571429
i	1	0.03571429
s	1	0.03571429
y	1	0.03571429
TOTAL	28	1.00000000

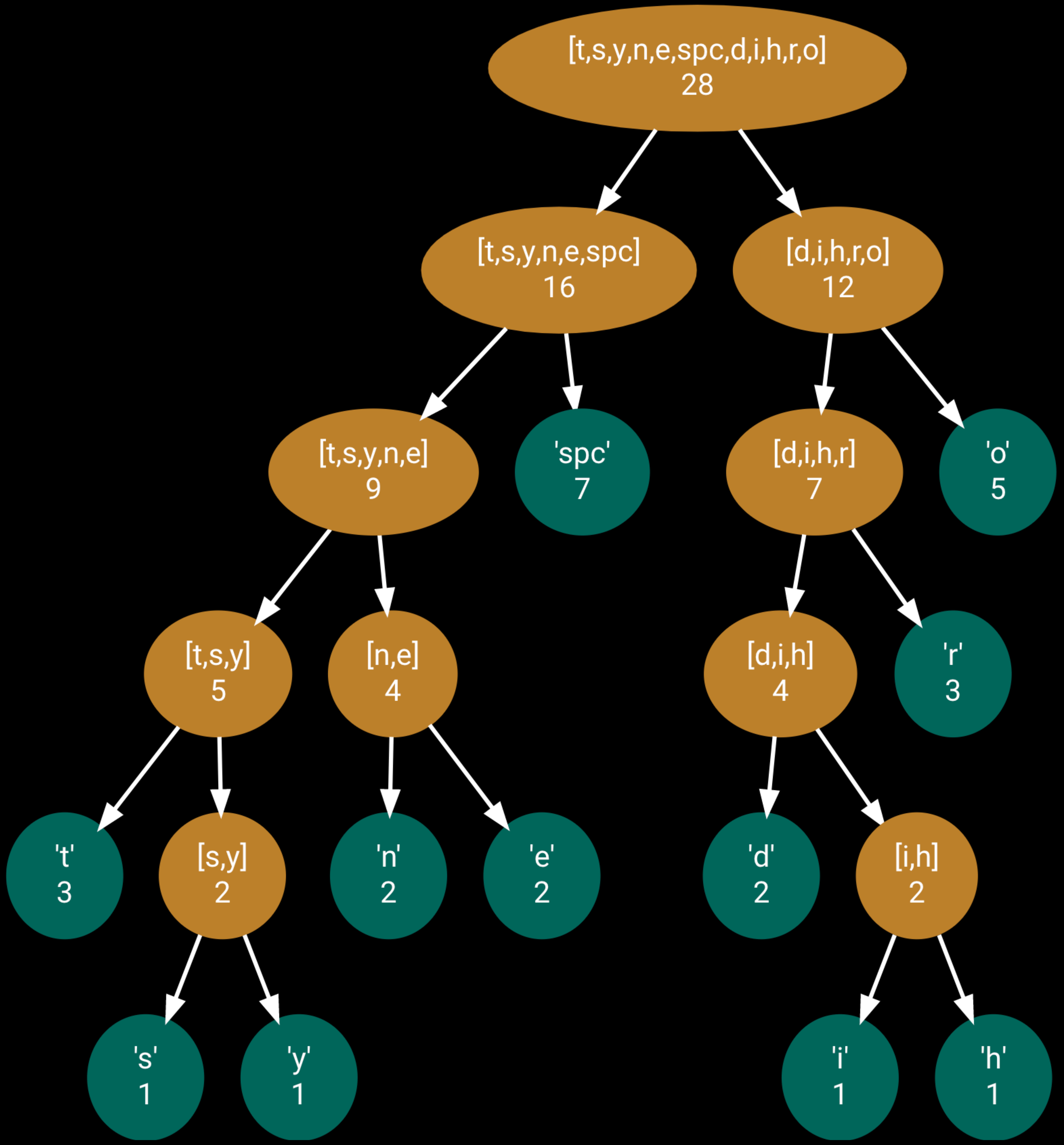
BUILD PREFIX TREE WITH PRIORITY QUEUE

- Bottom-up construction
- Store trees in Priority Queue by lowest frequency count for single character or tree formed from multiple letters with combined frequency count.
- Pop top two items in PQ (smallest frequency count), merge into tree.
- Continue until one last node left in PQ.
- Last node will be the root of prefix tree that has letters in leaf nodes. The root node’s frequency count will be the combined number of total characters.
- Read the root node from PQ to process in Huffman.



BUILD PREFIX TREE WITH PRIORITY QUEUE

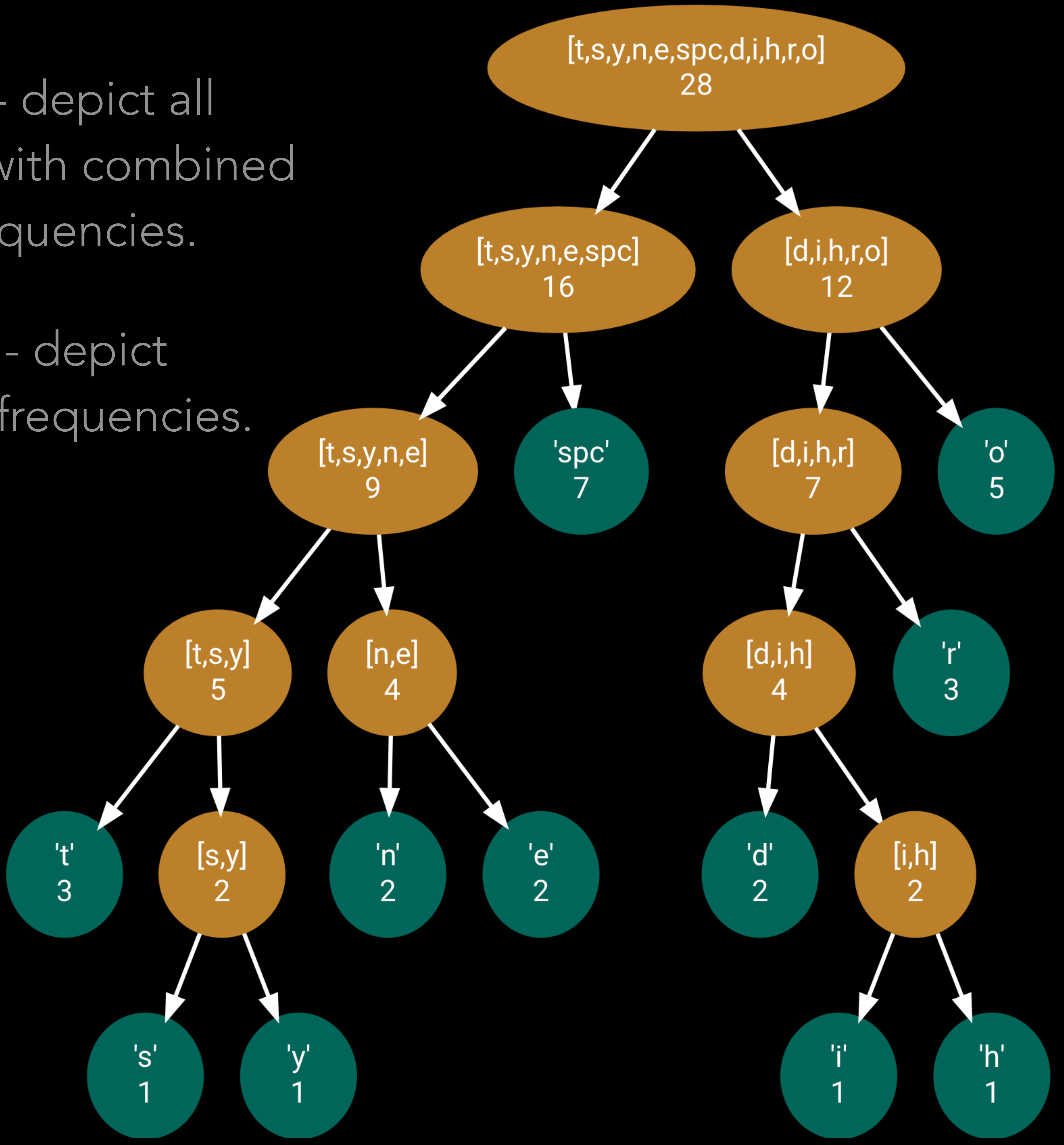
- Bottom-up construction
- Store trees in Priority Queue by lowest frequency count for single character or tree formed from multiple letters with combined frequency count.
- Pop top two items in PQ (smallest frequency count), merge into tree.
- Continue until one last node left in PQ.
- Last node will be the root of prefix tree that has letters in leaf nodes. The root node's frequency count will be the combined number of total characters.
- Read the root node from PQ to process in Huffman.
- The frequency count of the roots (and sub-roots) are the combined frequencies of their children.



PREFIX TREE WITH CHAR FREQUENCIES

letter	frequency	percentage
space	7	0.25000000
o	5	0.17857143
t	3	0.10714286
r	3	0.10714286
d	2	0.07142857
n	2	0.07142857
e	2	0.07142857
h	1	0.03571429
i	1	0.03571429
s	1	0.03571429
y	1	0.03571429
TOTAL	28	1.00000000

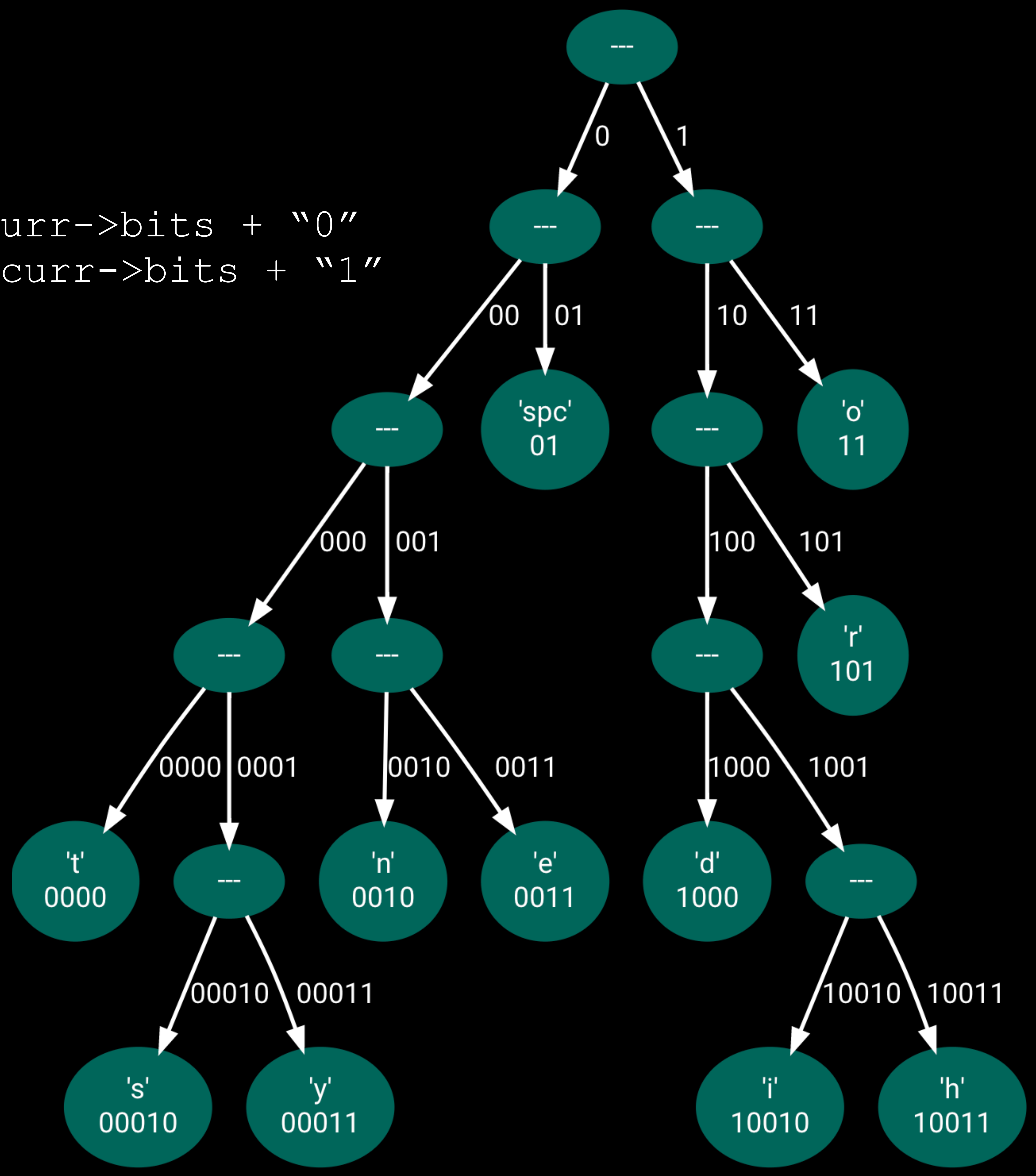
- Internal nodes - depict all children chars with combined sum of their frequencies.
- External nodes - depict individual char frequencies.



LESS FREQUENT CHARACTERS AT BOTTOM OF TREE

```
Node* curr
curr->left->bits = curr->bits + "0"
curr->right->bits = curr->bits + "1"
```

letter	frequency	percentage
space	7	0.25000000
o	5	0.17857143
t	3	0.10714286
r	3	0.10714286
d	2	0.07142857
n	2	0.07142857
e	2	0.07142857
h	1	0.03571429
i	1	0.03571429
s	1	0.03571429
y	1	0.03571429
TOTAL	28	1.00000000



Node.h

```cpp
/*
 * Node.h
 * cs10c_sum21
 * huffman
 */
#include <string>

using namespace std;

#ifndef __NODE_H_
#define __NODE_H_

class Node {
public:
  Node();
  Node(char c, int count);
  Node(Node* left, Node* right, char c, int count);
  ~Node();
  Node* left;
  Node* right;
  char c;
  int count;
  string bits;
};

#endif /* NODE_H_ */
```

- Each Node object is a binary node with left and right pointer.
- Nodes store character, count, and bits (string of 0s and 1s),.. bits are computed in **Huffman::SetBitCodes()**

Huffman.h

```cpp
/* Huffman.h
 * cs10c_sum21
 * huffman
 */

#ifndef HUFFMAN_H_
#define HUFFMAN_H_

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <list>
#include <stack>
#include <queue>
#include <string>
#include <map>
#include "heap/pq_zero.H"
#include "Node.h"

using namespace std;

class Huffman {
public:
  Huffman();
  Huffman(const string& inputFile);
  ~Huffman();
  void CountChars();
  void PrintMap();
  void BuildPQ(); // adds Nodes to priority queue with frequency counts
  void BuildHuffmanTree(); // combines lowest count nodes into Huffman Tree
  void SetBitsPerChar(); // calls private function
  void DisplayPrefixTree(); // optional, calls private function
  void SetBitCodes();  // postorder traversal, calls private function
  void Stats(); // prints the num bits used: non-compressed/compressed format
  void BitMap(); // prints char, bitstream
  void PrintMessage(); //prints original message in huffman codes

private:
  list<string> message_list; // stores multiple input files if needed
  string message;       // stores a single input file
  map<char,int> mymap; // maps frequency "count" indexed by char 'c'
  map<char,string> mymap_compress; // maps frequency "count" to bitstream
  pq_zero<Node*> pq;   // binary heap priority queue stores nodes with priority
  Node* root; // root of prefix tree
  void DisplayPrefixTree(Node* t); // optional, use for debugging, display tree
  void SetBitCodes(Node* n); // sets string bits for leaf nodes
  void SetBitsPerChar(Node* n); // maps bitstream to char
};

#endif /* HUFFMAN_H_ */
```

main.cpp

```cpp
/* main.cpp
 * cs10c_sum21
 * huffman
 */
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include "Huffman.h"

using namespace std;

int main(int argc, char* argv[]) {

  if (argc != 2) {
    cerr << "Usage error, expected: ./a.out *.txt\n";
    exit(1);
  }

  string input_file = argv[1];
  Huffman h(input_file);
  h.BuildHuffmanTree();
  h.PrintCharFrequencies();
  h.SetBitCodes();
  h.SetBitsPerChar();
  h.Stats();
  h.BitMap();

  return 0;
}
```

Huffman.h

```
Huffman(const string& inputFile); // constructor

// reads in a single file (message), one char at a time

// HINT: watch for newline and EOF.

// calls BuildPQ();, after file closed.
```

```
void BuildPQ(); // Adds Nodes to priority queue. Nodes have char and count.

pq_zero<Node*> pq;   // binary heap priority queue stores nodes with priority
```

```
void SetBitsPerChar(Node* n);

// Function maps characters and their string bits
```

Huffman.h

```
void PrintCharFrequencies();

// prints "char => char_frequency"

// see OUTPUT on right
```

- OUTPUT to right was produced via a traversal of
  - `std::map<char, int> mymap`
- A map iterator (it) and map member functions begin() and end() are used.
- So the order that the items in map are displayed are defined by the map increment operator (++), begin() and end() functions.

OUTPUT

```
  => 7
d => 2
e => 2
h => 1
i => 1
n => 2
o => 5
r => 3
s => 1
t => 3
y => 1
```

Huffman.h

```
void BuildHuffmanTree(); // combines lowest count nodes into Huffman Tree

// Pops top two Nodes off of PQ and combines them into a new node with sum
of frequency counts of the two children. Push new combined node into PQ.

// HINT: Leave last node in the PQ (this is the root).
```

```
void SetBitCodes(Node* n);

// Function traverses tree to compute string bits for each leaf node
in Huffman tree.

// HINT: Node* curr
         curr->left->bits = curr->bits + "0"
         curr->right->bits = curr->bits + "1"
```

Huffman.h

```
void Stats();

// see OUTPUT on right

// MUST match OUTPUT exactly!
```

OUTPUT

```
Without compression, 8-bit characters:
  occurs 7 times. Cost of: 56 bits. Total so far: 56 bits.
d occurs 2 times. Cost of: 16 bits. Total so far: 72 bits.
e occurs 2 times. Cost of: 16 bits. Total so far: 88 bits.
h occurs 1 times. Cost of: 8 bits. Total so far: 96 bits.
i occurs 1 times. Cost of: 8 bits. Total so far: 104 bits.
n occurs 2 times. Cost of: 16 bits. Total so far: 120 bits.
o occurs 5 times. Cost of: 40 bits. Total so far: 160 bits.
r occurs 3 times. Cost of: 24 bits. Total so far: 184 bits.
s occurs 1 times. Cost of: 8 bits. Total so far: 192 bits.
t occurs 3 times. Cost of: 24 bits. Total so far: 216 bits.
y occurs 1 times. Cost of: 8 bits. Total so far: 224 bits.
Total bits = 224

Huffman codes used for lossless compression:
  occurs 7 times. Bit sequence: 01. Cost of: 14 bits. Total so far: 14 bits.
d occurs 2 times. Bit sequence: 1000. Cost of: 8 bits. Total so far: 22 bits.
e occurs 2 times. Bit sequence: 0011. Cost of: 8 bits. Total so far: 30 bits.
h occurs 1 times. Bit sequence: 10011. Cost of: 5 bits. Total so far: 35 bits.
i occurs 1 times. Bit sequence: 10010. Cost of: 5 bits. Total so far: 40 bits.
n occurs 2 times. Bit sequence: 0010. Cost of: 8 bits. Total so far: 48 bits.
o occurs 5 times. Bit sequence: 11. Cost of: 10 bits. Total so far: 58 bits.
r occurs 3 times. Bit sequence: 101. Cost of: 9 bits. Total so far: 67 bits.
s occurs 1 times. Bit sequence: 00010. Cost of: 5 bits. Total so far: 72 bits.
t occurs 3 times. Bit sequence: 0000. Cost of: 12 bits. Total so far: 84 bits.
y occurs 1 times. Bit sequence: 00011. Cost of: 5 bits. Total so far: 89 bits.
Total bits = 89
```

Huffman.h

```
void BitMap();

// prints “char, bit stream”

// see OUTPUT on right
```

OUTPUT

```
 ,01
d,1000
e,0011
h,10011
i,10010
n,0010
o,11
r,101
s,00010
t,0000
y,00011
```

CODE COMPILATION & TEST CASE CRITERIA

To compile and run program use the following commands.

- $ g++ -W -Wall -Werror -g -std=c++11 Node.cpp Huffman.cpp main.cpp
- $ ./a.out yoda.txt

Program is required to handle input message files with 2 or more paragraphs (ending in a newline followed by an EOF).

Ensure your program works with these test files: `yoda.txt, 1_pp.txt, 2_pp.txt`

```
//optional, used for debugging

void DisplayPrefixTree();
```

space = blank (at root =>28)

do_or_do_not_there_is_no_try

OUTPUT

```
  o=>5=>11
 =>12=>1
   r=>3=>101
  =>7=>10
     h=>1=>10011
    =>2=>1001
     i=>1=>10010
   =>4=>100
    d=>2=>1000
=>28=>
   =>7=>01
 =>16=>0
    e=>2=>0011
   =>4=>001
    n=>2=>0010
  =>9=>00
     y=>1=>00011
    =>2=>0001
     s=>1=>00010
   =>5=>000
    t=>3=>0000
```

REFERENCES

- https://www.huffmancoding.com/my-uncle/scientific-american
- Morse code table - James Kanjo, CC BY-SA 3.0 <http://creativecommons.org/licenses/by-sa/3.0/>, via Wikimedia Commons
- https://www.cis.upenn.edu/~cis110/current/homework/hw03_base/ascii_table.png