CS 241
Data Organization
Huffman Coding

Brooke Chenoweth

University of New Mexico

Fall 2014

Encoding Data

```
a small sample string
```

How many bits would we need to encode the string?

Encoding Data – as chars

`a small sample string`

- 21 characters
- 8 bits per character
- 168 bits

Smaller fixed length code

`a small sample string`

- Only 12 unique characters in this string
- Why not represent them with 4 bits each?
- This only needs 84 bits.
- Can we do better?

Variable length code

- Some symbols appear more often than others.
- Why not use shorter codes for more common symbols?
- How would we know when a code word is over if the length varies?

Prefix Codes

- A *prefix code* is a code system that has the *prefix property*.
- Prefix property – There is no valid code word in the system that is a prefix of any other valid code word in the set.
- Uniquely decodable – No need for delimiters.

Count Symbol Frequencies

`a small sample string`

Symbol	**Freq**
space	3
a	3
e	1
g	1
i	1
l	3
m	2
n	1
p	1
r	1
s	3
t	1

Frequencies and Codes

`a small sample string`

Symbol	**Freq**	**Code**
space	3	111
a	3	110
e	1	1010
g	1	0101
i	1	0100
l	3	100
m	2	001
n	1	0001
p	1	0000
r	1	10111
s	3	011
t	1	10110

Code length

`a small sample string`

Symbol	Freq	Code	Length	Total
space	3	111	3	9
a	3	110	3	9
e	1	1010	4	4
g	1	0101	4	4
i	1	0100	4	4
l	3	100	3	9
m	2	001	3	6
n	1	0001	4	4
p	1	0000	4	4
r	1	10111	5	5
s	3	011	3	9
t	1	10110	5	5
				72

Length Comparison

- Fixed code needed 84 bits.
- Huffman code only needed 72 bits.
- Huffman coding produces prefix codes that achieve lowest possible expected code word length.

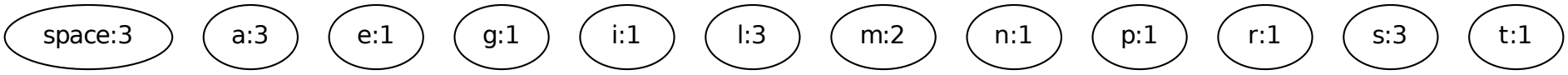
How does it work?

- Start with set of symbols and their weights (frequencies or probabilities).
- Generate a binary tree with minimum weighted path length from root. (This tree represents a *prefix code* with minimum expected codeword length.)
- Use the tree to encode/decode data.

Binary Code as Tree

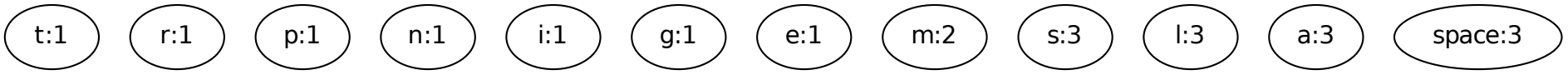
- Symbols are stored as leaves.
- Number left edges 0 and right edges 1
- Path from root to leaf is the code for the symbol at the leaf.

Building Huffman Tree



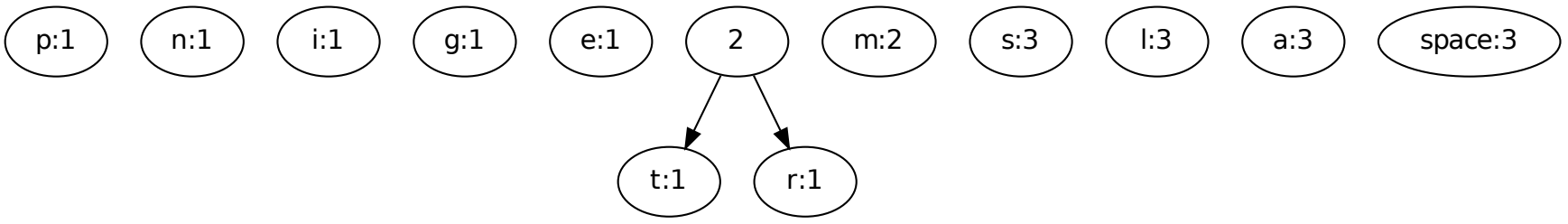
Construct leaf nodes from symbols and weights

Building Huffman Tree



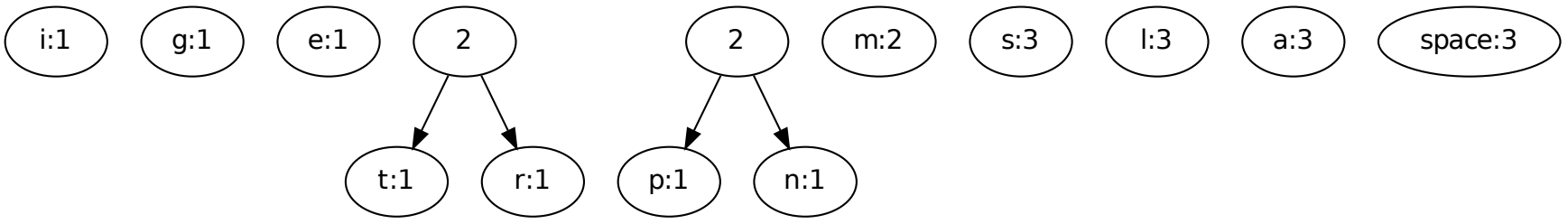
Place nodes in a priority queue

Building Huffman Tree



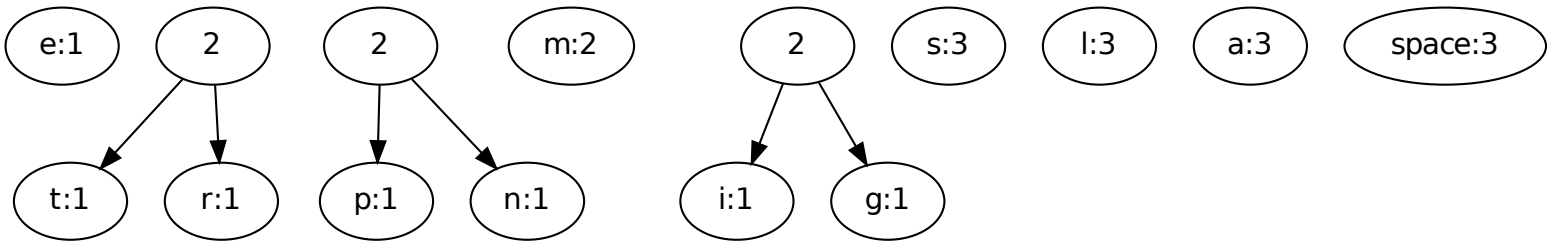
Remove first two nodes,
create new node with sum of weights,
add to queue

Building Huffman Tree



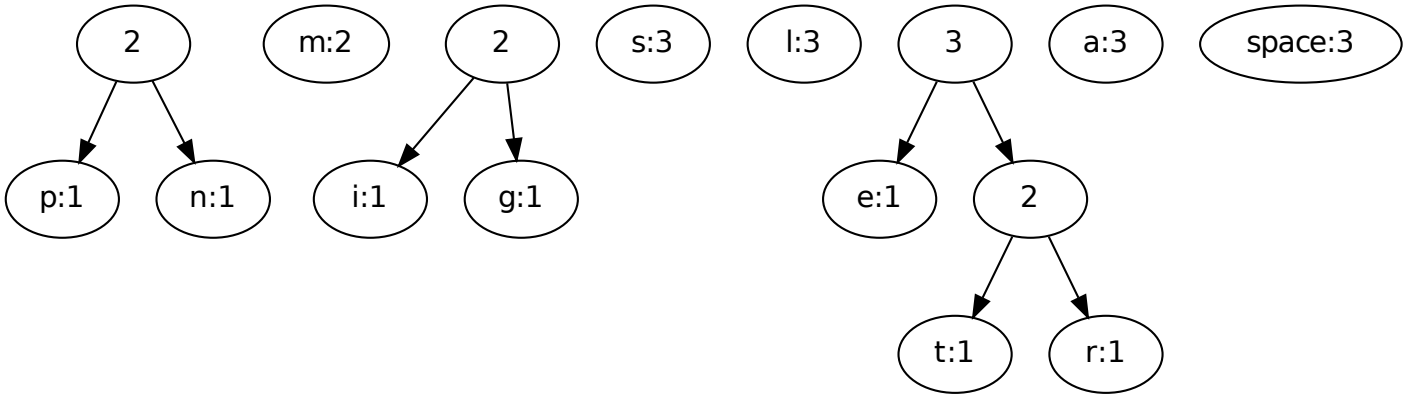
Keep going. . .

Building Huffman Tree



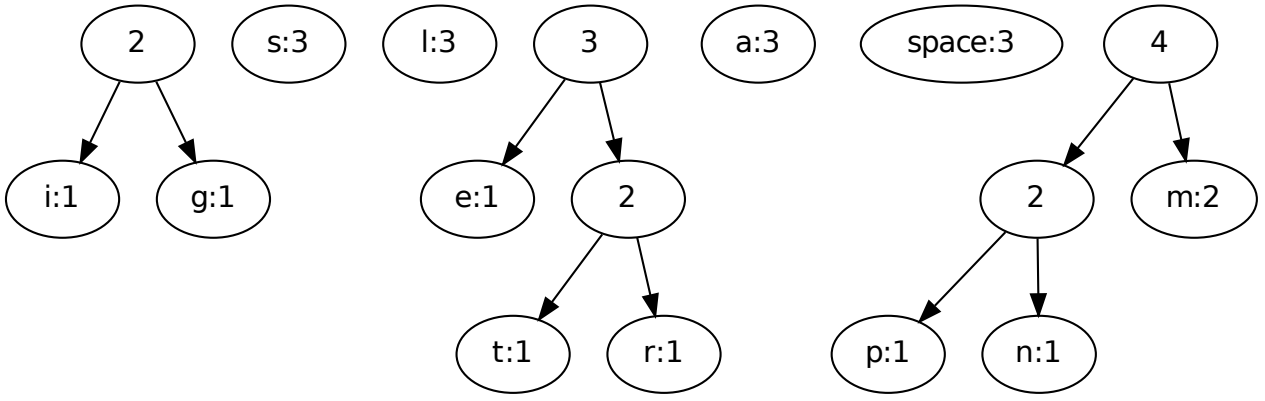
Keep going. . .

Building Huffman Tree



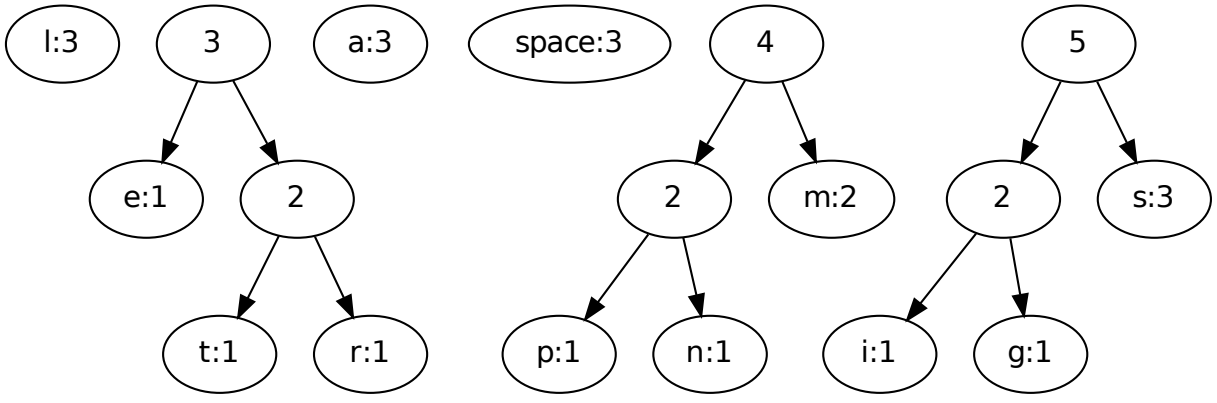
Keep going. . .

Building Huffman Tree



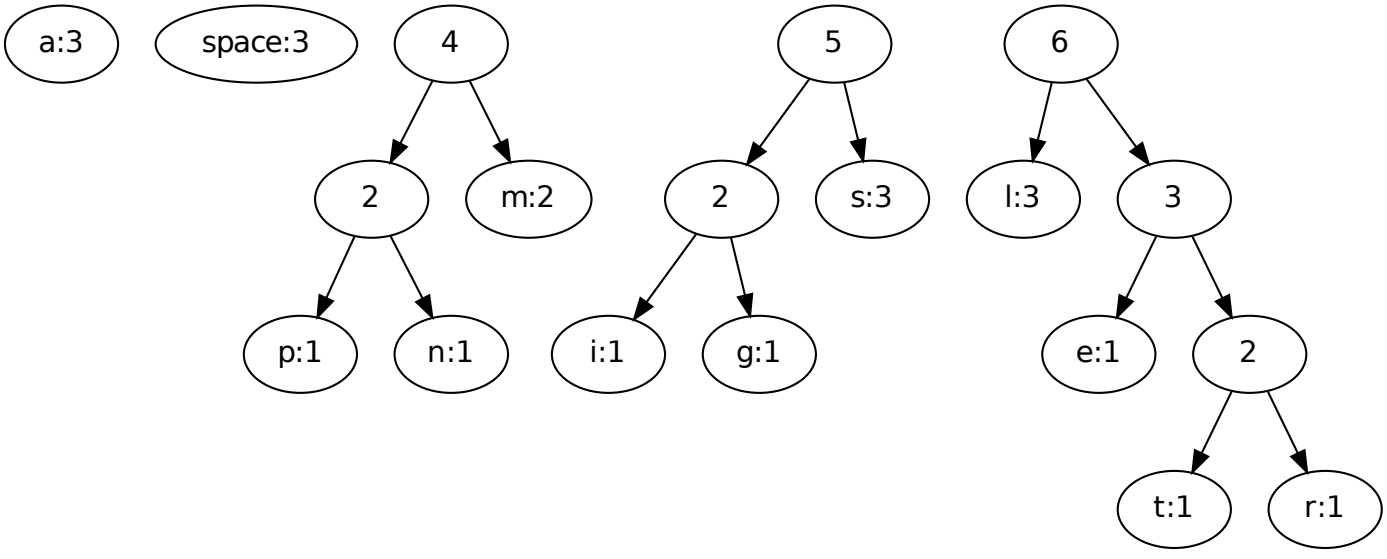
Keep going. . .

Building Huffman Tree



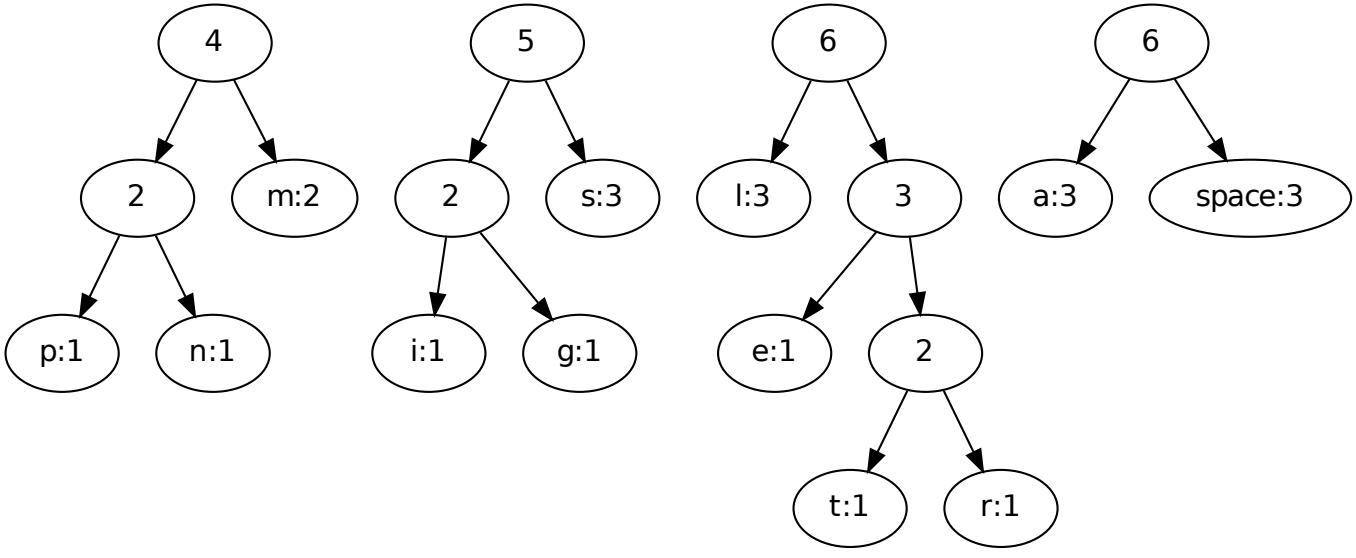
Keep going. . .

Building Huffman Tree



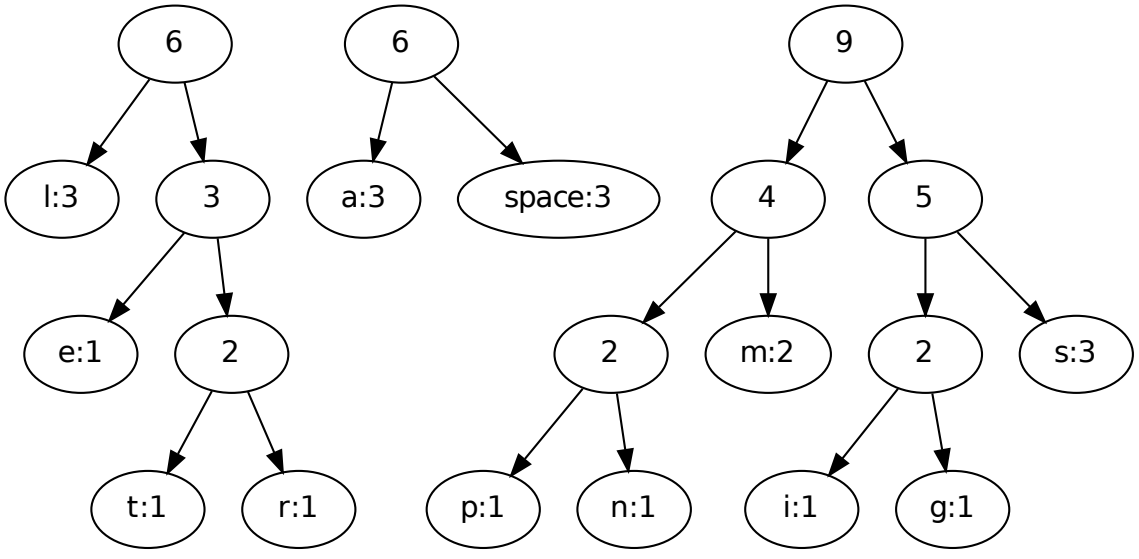
Keep going. . .

Building Huffman Tree



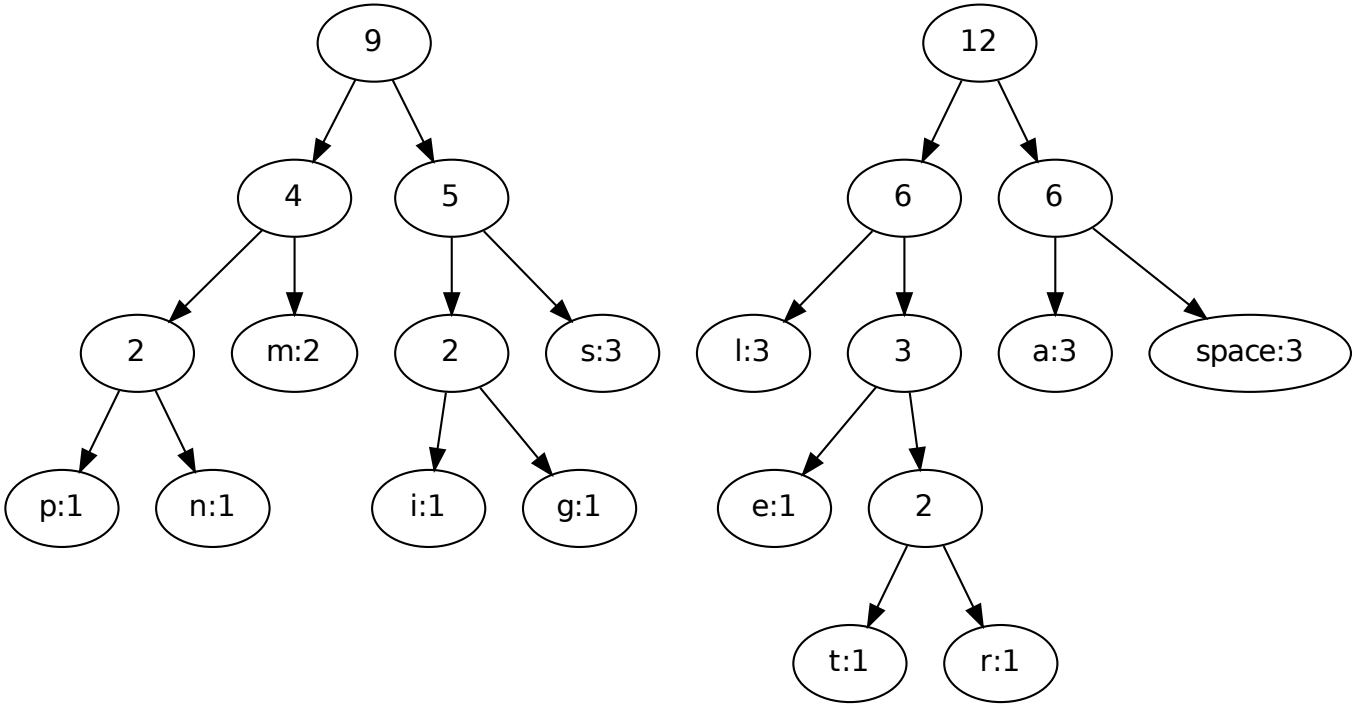
Keep going. . .

Building Huffman Tree



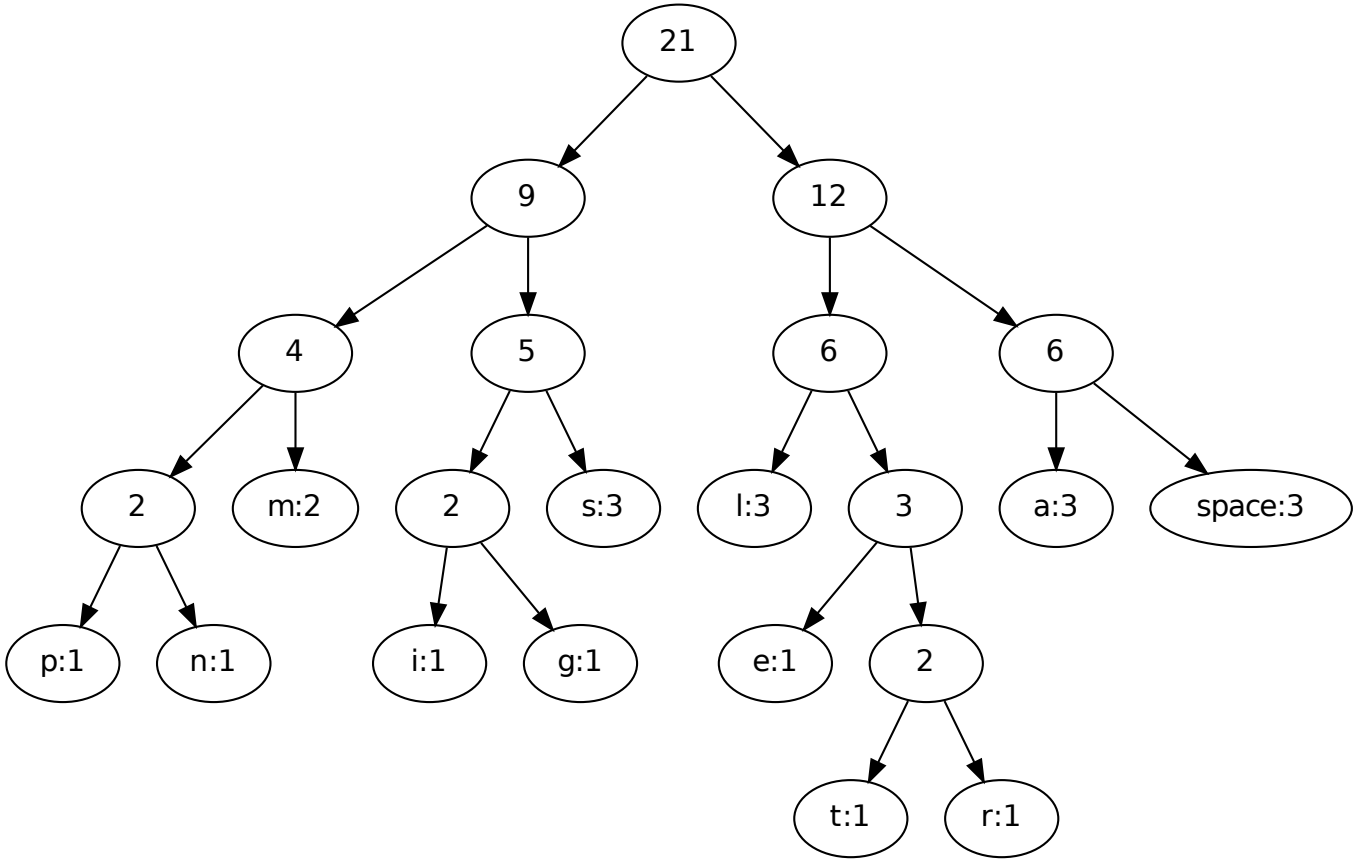
Keep going. . .

Building Huffman Tree



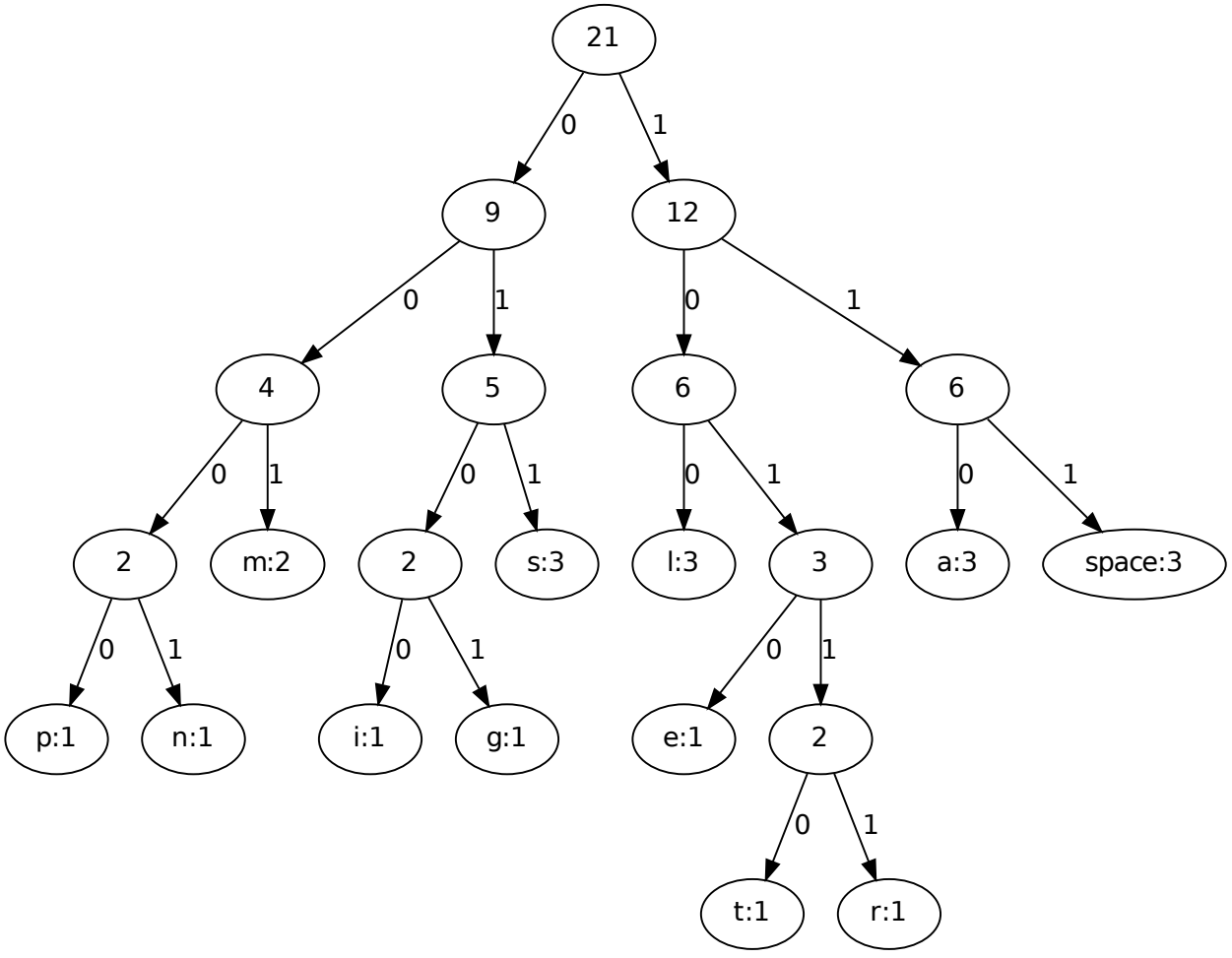
Keep going. . .

Building Huffman Tree



. . . until you have a single tree

Building Huffman Tree



Path from root to leaf is code for symbol.

Huffman Coding Project – Encoding a File

- Count occurances of symbols (bytes)
- Build Huffman tree from frequency data.
- Generate Huffman code from tree.
- Write frequency table, symbol count
- Encode data with Huffman code, write to output

Huffman Coding Project – Decoding a File

- Read frequency table, symbol count
- Build Huffman tree from frequency data.
- Use tree to decode encoded data.

Building the Code – Queue

- Count frequency of symbols in a stream.
  - stdin is fine for testing.
  - Final program will read data from a file.
- Create leaf nodes from symbol frequency data.
  - Don't include symbols with zero occurances.
- Add all nodes to priority queue.
  - Simple implementation – ordered linked list
  - More efficient implementation – heap

Building the Code – Tree

- While queue has more than one item:
  - Remove two trees from queue.
  - Create new tree with those two as children.
    - First removed should be left child.
    - Tree frequency count is sum of children's counts.
  - Add new tree to priority queue.
- Use final Huffman tree to generate codes.
- Huffman code for symbol at leaf is the path from root to leaf.
  - `0` for each left branch
  - `1` for each right branch

Printing the Frequency and Code Table

```
Symbol   Freq     Code
=32      3        111
a        3        110
e        1        1010
g        1        0101
i        1        0100
l        3        100
m        2        001
n        1        0001
p        1        0000
r        1        10111
s        3        011
t        1        10110
Total chars = 21
```

- For debugging and grading purposes, print out all the symbols, frequencies, and codes. (Ascending symbol order)
- Columns are separated with tab character.
- Visible ASCII characters (33-126) are printed as characters. Values outside that range are displayed as equals sign followed by integer value of the byte.

Huffman Tree Node

- All nodes have a frequency count – `unsigned long`
- Leaf nodes have a symbol – `unsigned char`
- Internal nodes have references to two subtrees.
- May want some additional bookkeeping fields. (isLeaf, parentNode, etc.)

Comparing Huffman Trees

The exact code generated depends on the order of the trees in the priority queue. For this project, we'll compare trees in the following manner.

- First, compare frequency values for the trees. The tree with the lower frequency has higher priority.
- If the two trees have the same frequency, compare the symbols in the leftmost leaf nodes of the two trees. Larger value has higher priority.

Encoded File Format

Binary files written on one system may not match file written by same program on another system.

- Number of symbols in table – `unsigned short`
- Symbols and Frequency pairs
  - Symbol – `unsigned char`
  - Frequency for symbol – `unsigned long`
- Total number of encoded symbols – `unsigned long`
- Bits of encoded data. Last byte may be padded with zeros if code string did not end on byte boundary.

Encoded File Example

data	a	*space*	s	m	a	l	l	*space*
codes	110	111	011	001	110	100	100	111
code stream	110111011001110100100111...							
bytes	10111011 10111001 11100100 ...							
hex	BB B9 E4 ...							

You can examine the encoded file in a hex editor. In emacs, use `M-x hexl-find-file` to do this.