HUFFMAN CODING 2

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

administrivia…

-assignment 12 is due next **TUESDAY**

-final project

-upcoming lectures

last time…

number encodings

binary

-each bit represents power of 2

1	0	0	1	0	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
on	off	off	on	off	off	on	on

-sum up all the bits that are on

-128 + 16 + 2 + 1 = 147

-how can we convert the other way?

ASCII

-each character corresponds to one byte

    -remember, a byte is just an 8-bit number! (0-255)

-for example:

00100000 = 32 = ‘ ‘ (blank space)
00111011 = 59 = ;
01000001 = 65 = A
01000010 = 66 = B

hexadecimal

-hexadecimal is the base-16 number system

-we only have 10 digits (0-9), so to use a number base greater than 10 we need more symbols

-in hex, we use the letters A through F

-A represents the value ten

-F represents the value fifteen

hex to binary

-each hex digit is a specific 4-bit sequence

0 = 0000

1 = 0001

...

E = 1110

F = 1111

-converting from hex to binary is as simple as representing each digit with its bit-sequence

12 EF = 0001 0010 1110 1111

-a single byte is two hex digits

-the bytes in the above are 12 and EF

WHAT IS THE HEX VALUE OF THESE 8 BITS?
1110 1000

A) **18**
B) **EF**
C) **E8**
D) **A4**

HOW MANY DIFFERENT VALUES CAN 3 BITS HOLD?

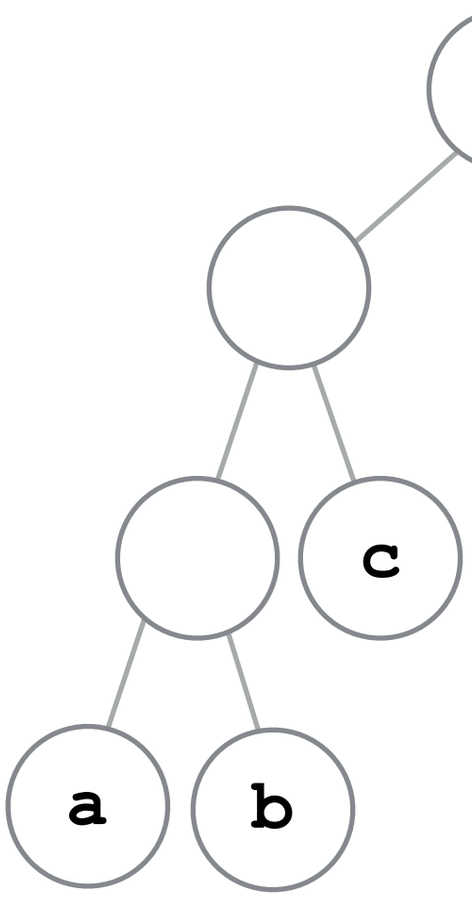
A) **3**
B) **4**
C) **7**
D) **8**
E) **15**
F) **16**

file compression

-suppose we the following string stored in a text file:
ddddddddddddabc

-how many bytes of disk space does it take to store these 15 characters using ASCII?

-is there any way to represent this file in fewer bytes?



a: 000
b: 001
c: 01
d: 1

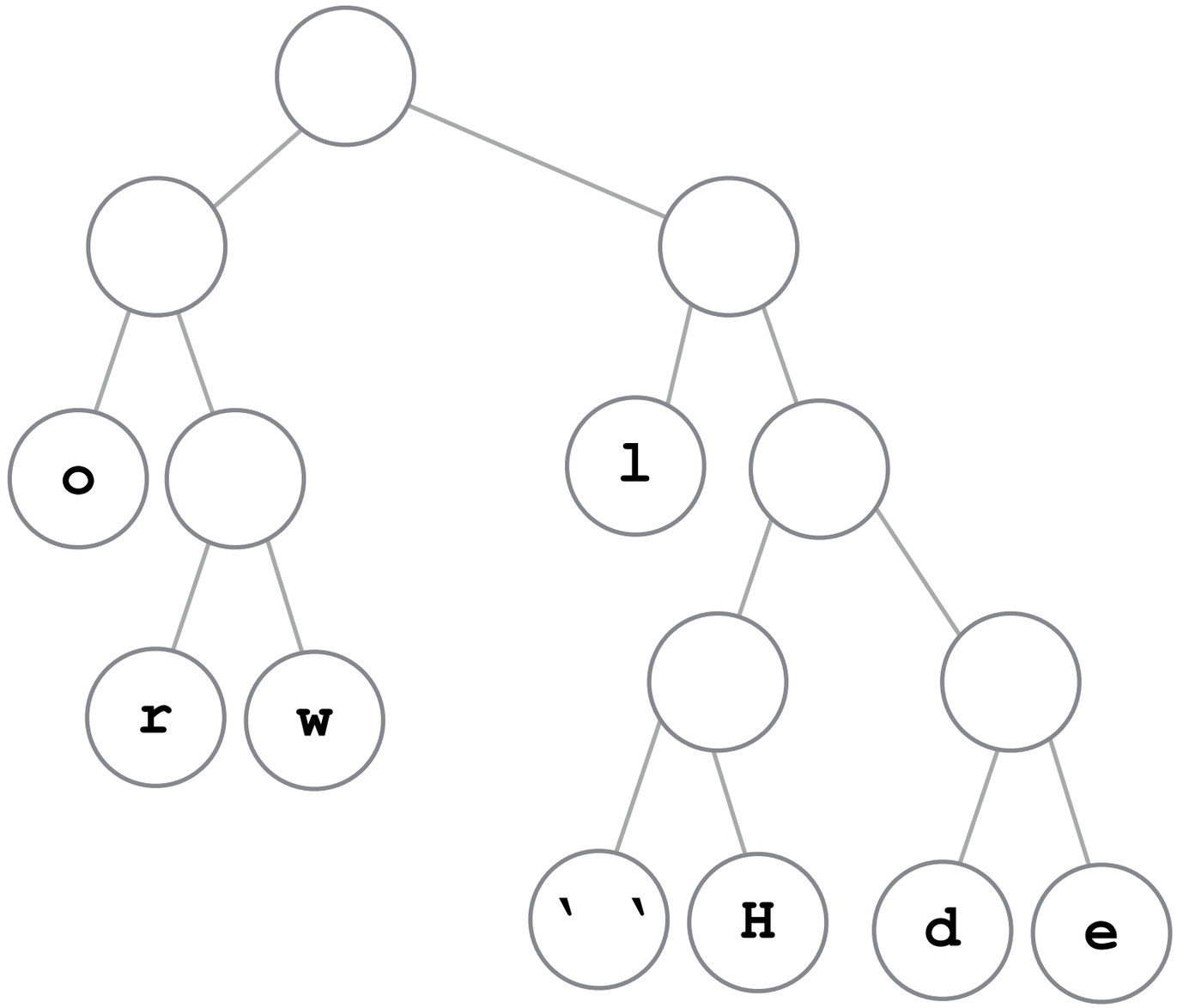
“ddddddddddddddabc” takes 15 bytes (120 bits) in ASCII

111111111111100000101 is less than 3 bytes (20 bits)

WHY IS THE D NEAR THE TOP OF THE TREE?

WHAT STRING DO THESE BITS ENCODE?
0 1 1 0 0 0 1 0 1 1 1 0

A) **word**
B) **wow**
C) **wool**
D) **were**



Huffman's algorithm

1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

today

-go through Huffman's algorithm for compression

  -using a priority queue

  -tie-breakers

  -writing the header

  -compressing the string

-decompression

1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

I heart data.

I heart data.

character	frequency
I	1
' '	2
h	1
e	1
a	3
r	1
t	2
d	1
.	1

1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

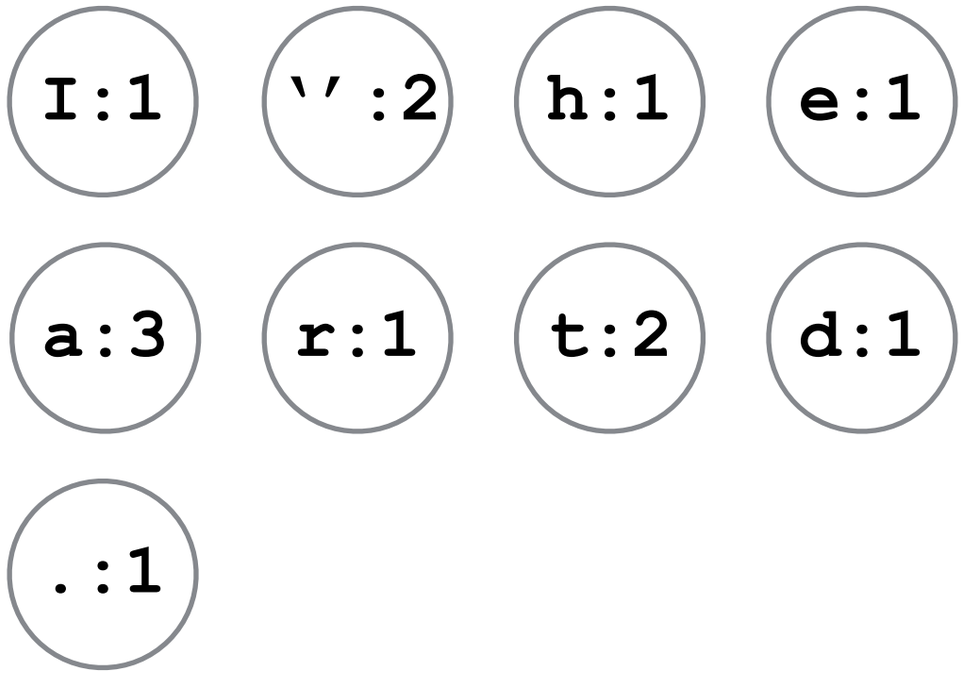
4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

I heart data.

character	frequency
I	1
' '	2
h	1
e	1
a	3
r	1
t	2
d	1
.	1



-but, let's think about what happens when we write our compressed codes

-files must be a *whole number* of bytes

-but, the length of our compressed string will not necessarily be a factor of 8

-what do we do?

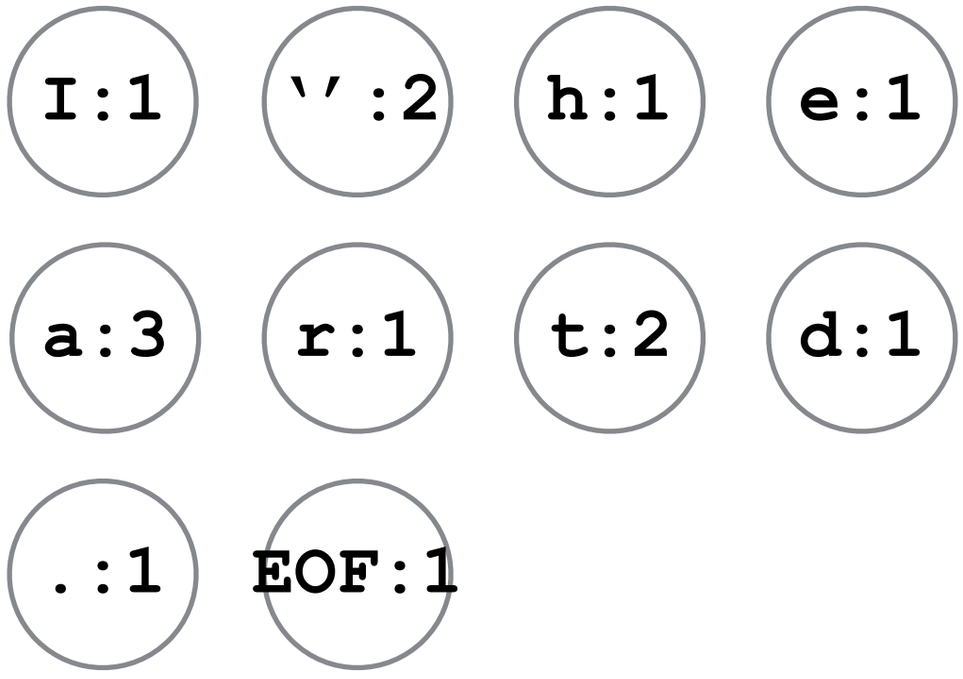
-to solve this, we add a special EOF leaf node 

-then, if we encounter this during decompression, we know we are done

-and we ignore the remaining bits

I heart data.

character	frequency
I	1
' '	2
h	1
e	1
a	3
r	1
t	2
d	1
.	1
EOF	1



1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

-how do we determine the *highest* priority?

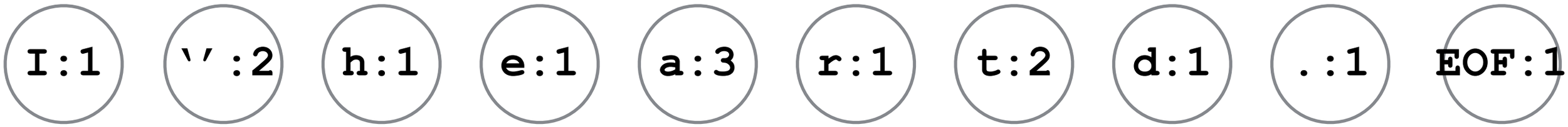
-what happens if two nodes have the same priority?

-need a tie-breaker!

-use ASCII value for character to break tie

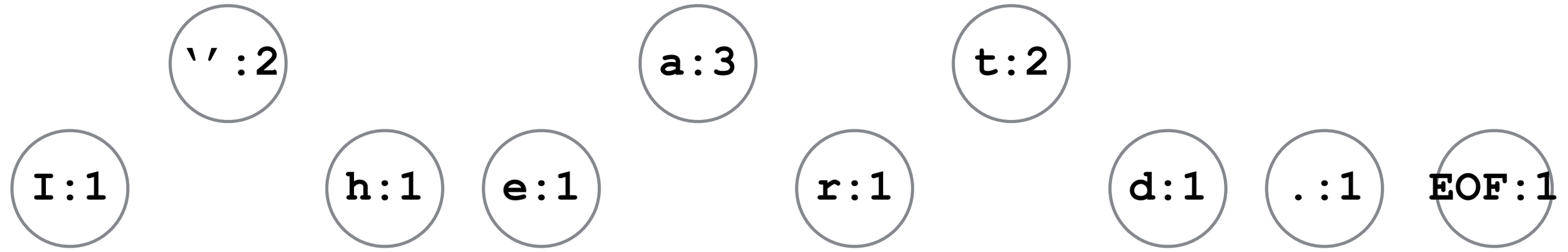
-lowest values have highest priority

I heart data.



PQ:

I heart data.

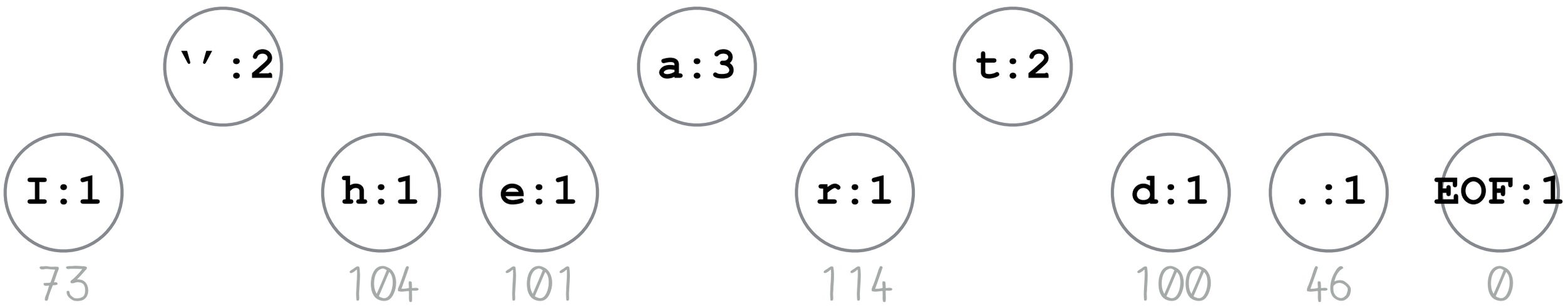


PQ:

I heart data.

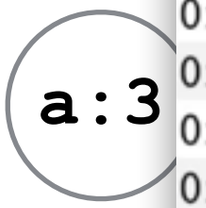
character	frequency	ASCII
I	1	73
' '	2	32
h	1	72
e	1	101
a	3	97
r	1	114
t	2	116
d	1	100
.	1	46
EOF	1	0

I heart data.



PQ:

I hea



73

104

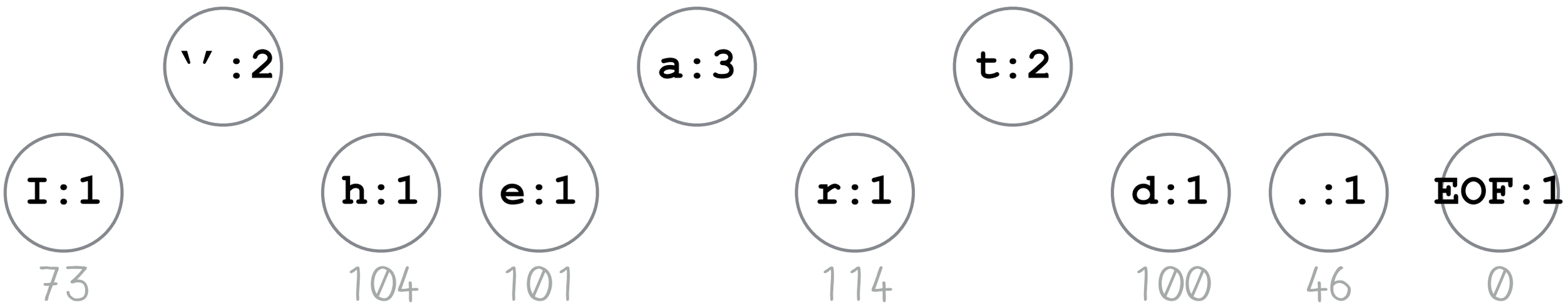
101

0

PQ:

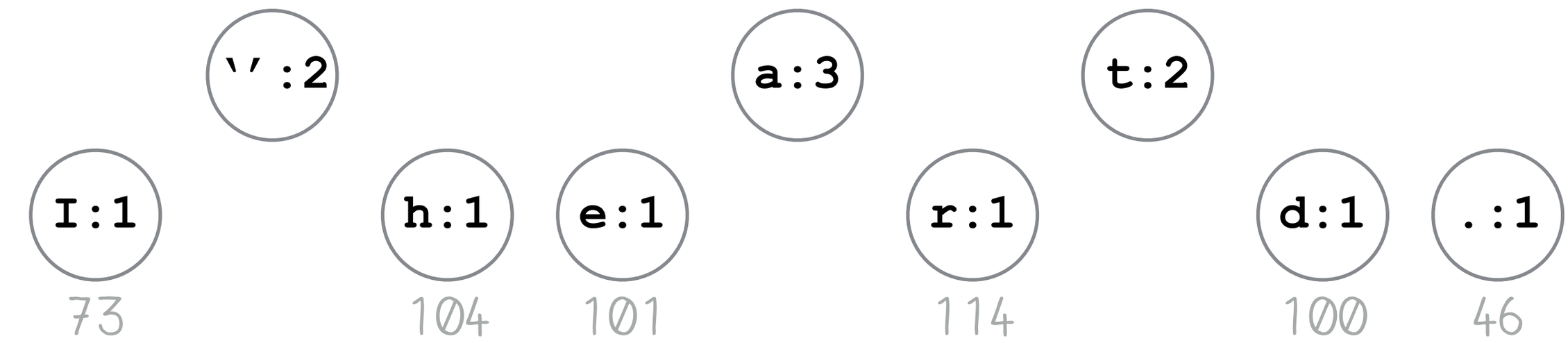
Hex	Dec	Char	
0x00	0	NULL	null
0x01	1	SOH	Start of heading
0x02	2	STX	Start of text
0x03	3	ETX	End of text
0x04	4	EOT	End of transmission
0x05	5	ENQ	Enquiry
0x06	6	ACK	Acknowledge
0x07	7	BELL	Bell
0x08	8	BS	Backspace
0x09	9	TAB	Horizontal tab
0x0A	10	LF	New line
0x0B	11	VT	Vertical tab
0x0C	12	FF	Form Feed
0x0D	13	CR	Carriage return
0x0E	14	SO	Shift out
0x0F	15	SI	Shift in
0x10	16	DLE	Data link escape
0x11	17	DC1	Device control 1
0x12	18	DC2	Device control 2
0x13	19	DC3	Device control 3
0x14	20	DC4	Device control 4
0x15	21	NAK	Negative ack
0x16	22	SYN	Synchronous idle
0x17	23	ETB	End transmission block
0x18	24	CAN	Cancel
0x19	25	EM	End of medium
0x1A	26	SUB	Substitute
0x1B	27	ESC	Escape

I heart data.

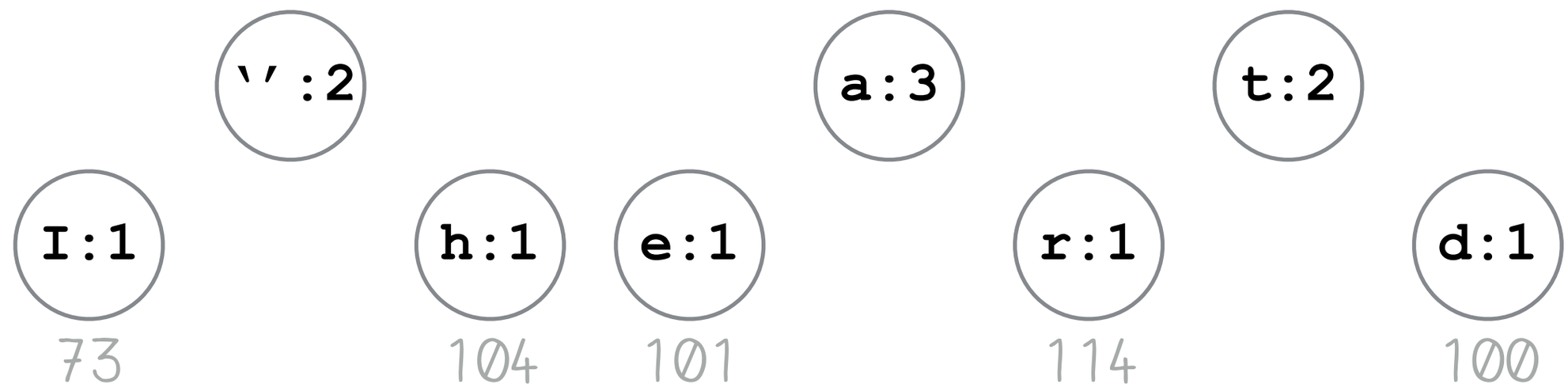


PQ:

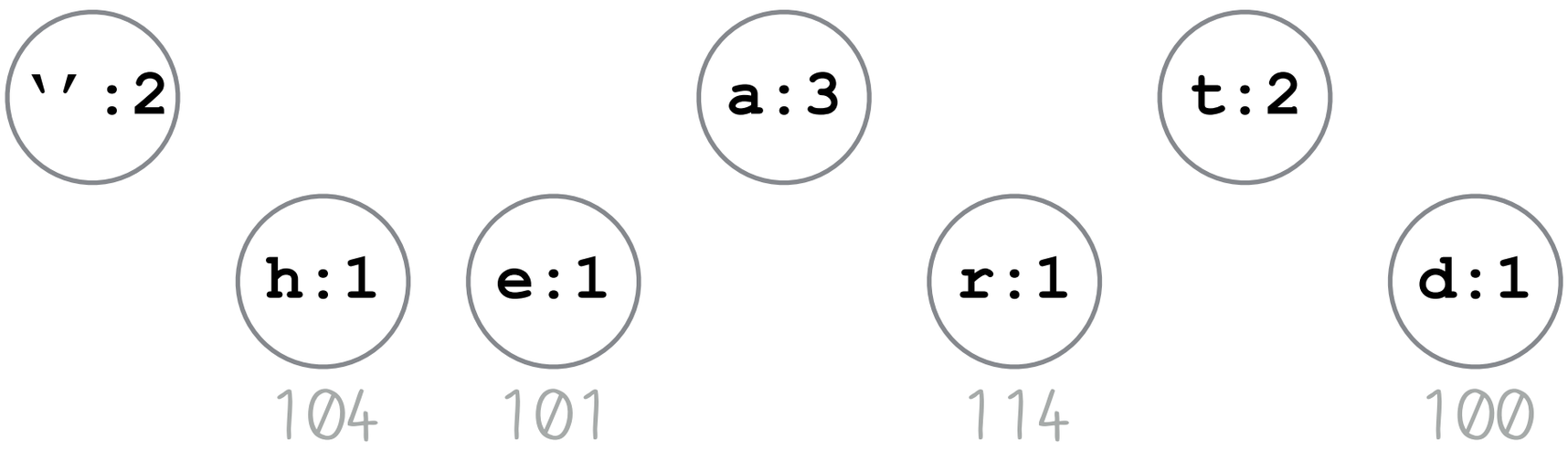
I heart data.

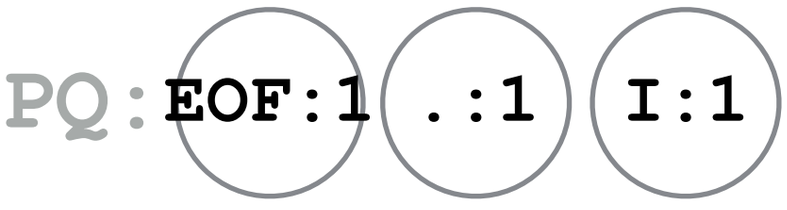


I heart data.

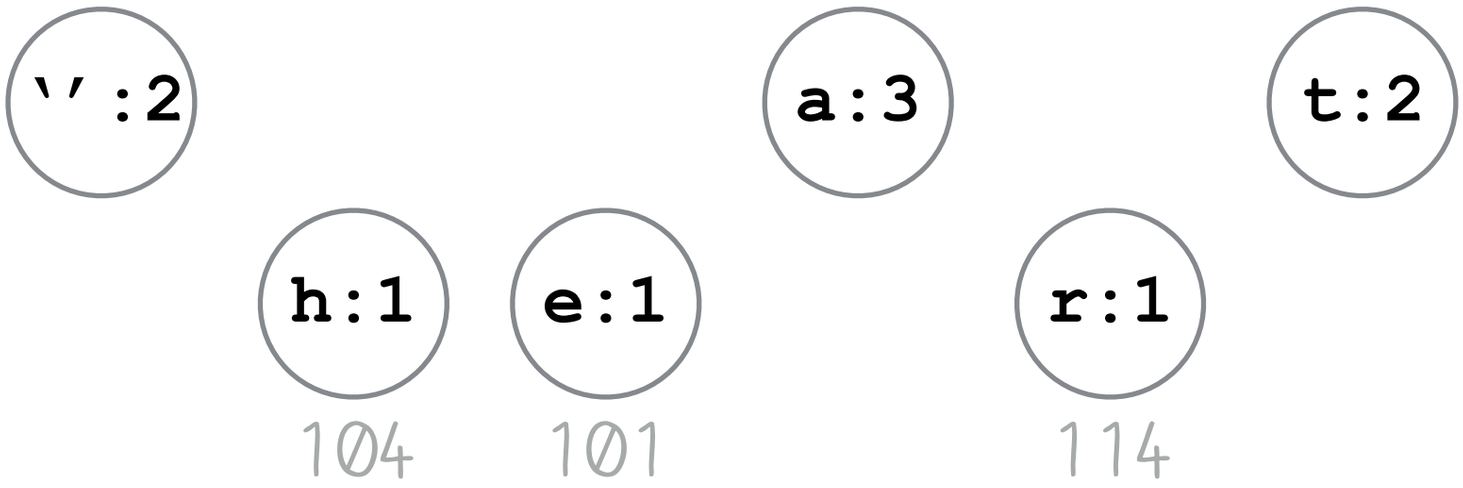


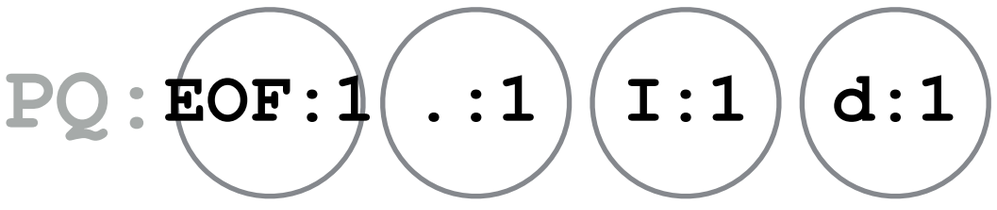
I heart data.



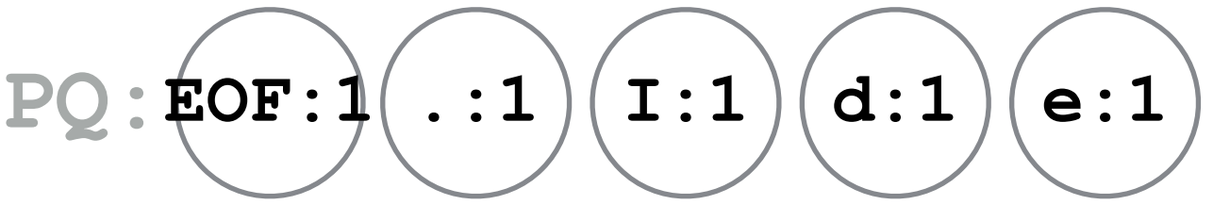


I heart data.

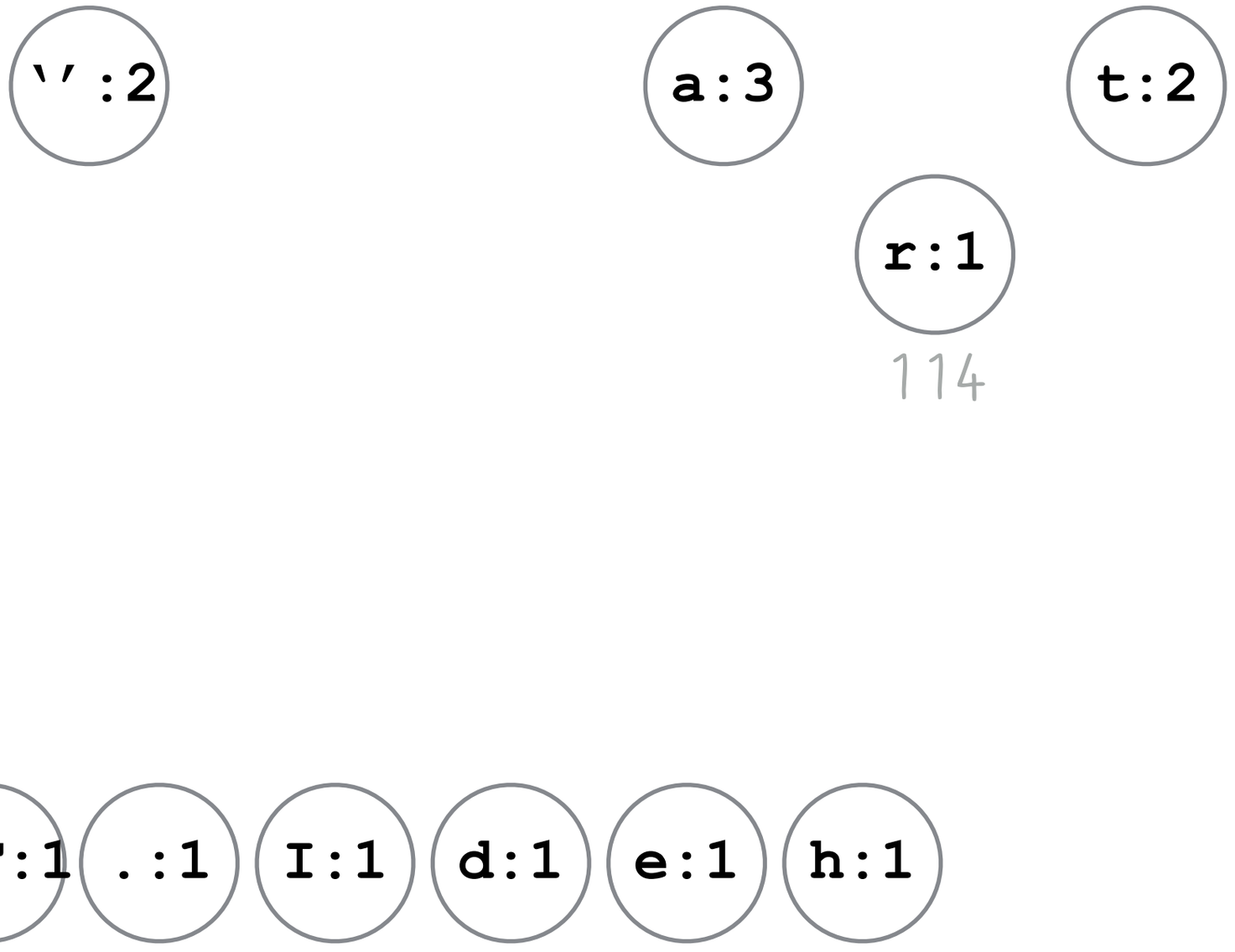




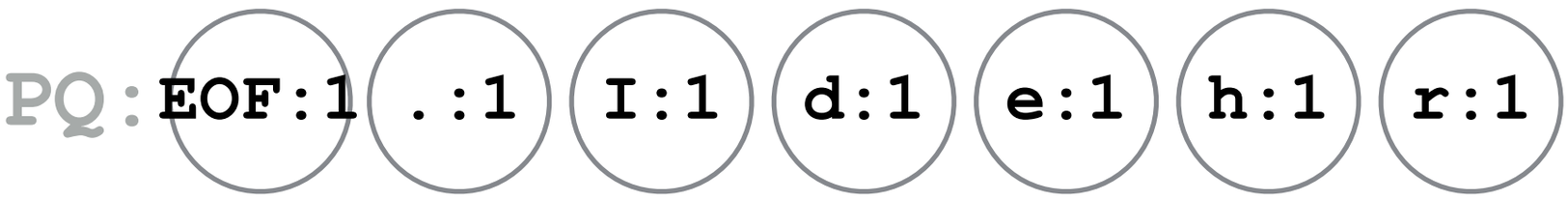
I heart data.



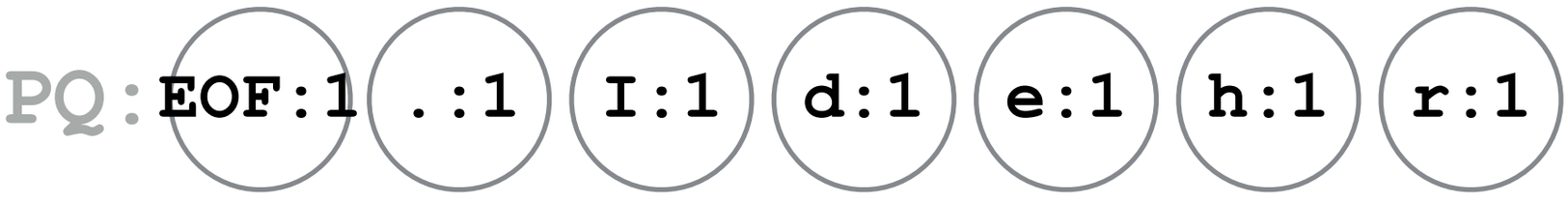
I heart data.



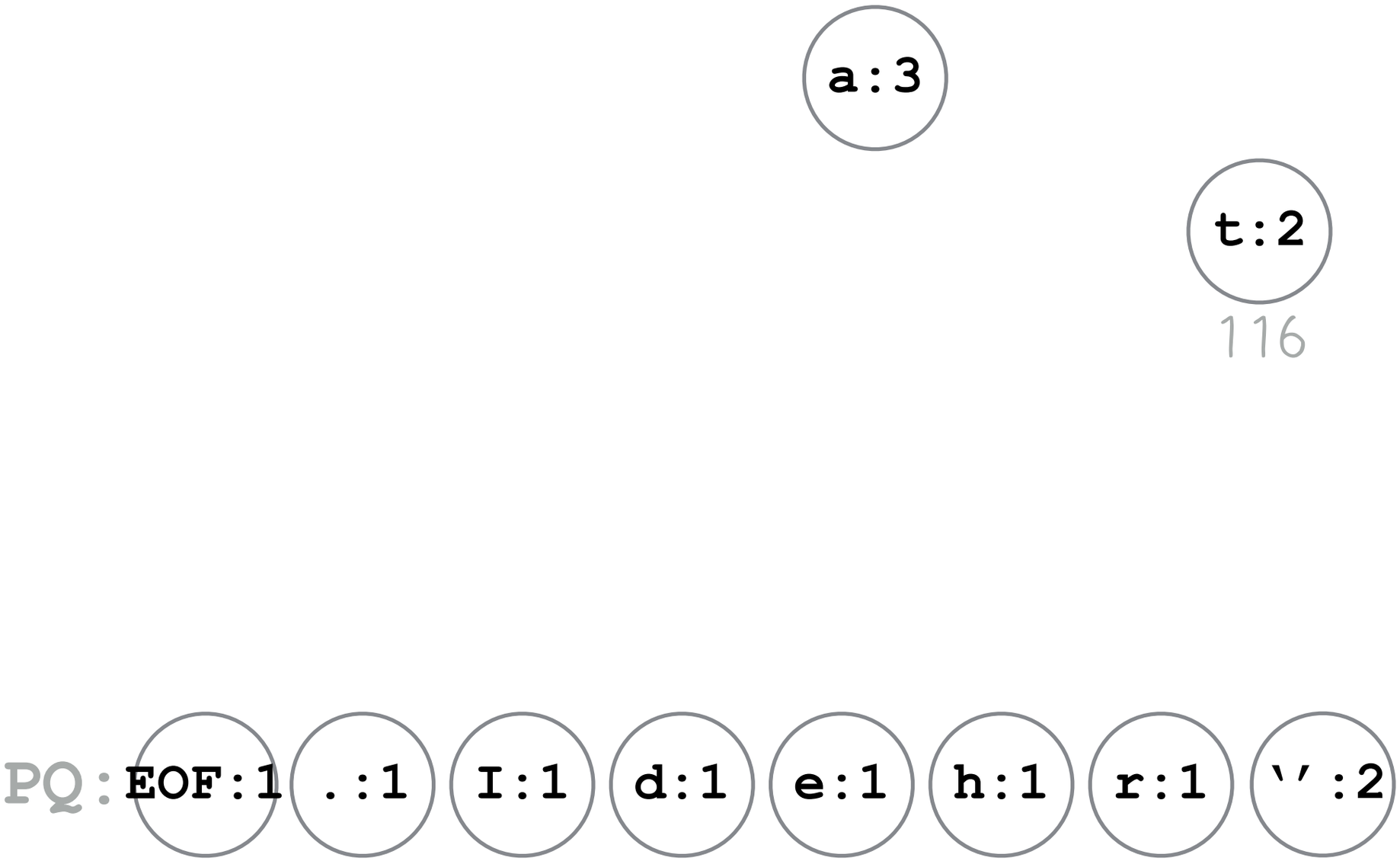
I heart data.



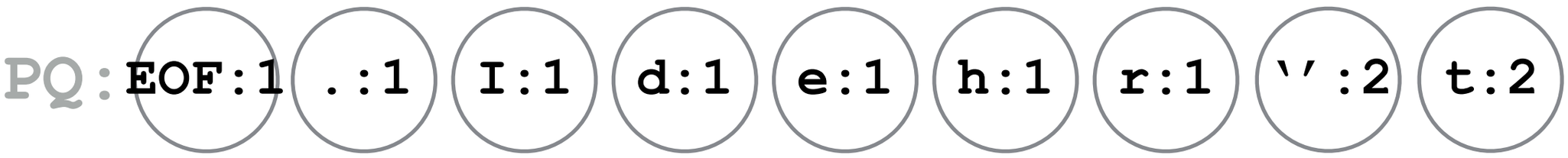
I heart data.



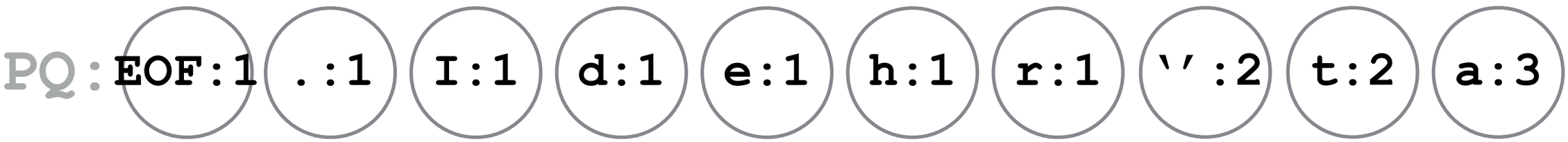
I heart data.



I heart data.



I heart data.



1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

-merge the two lowest weight trees together

  -make a new parent node with their combined weight

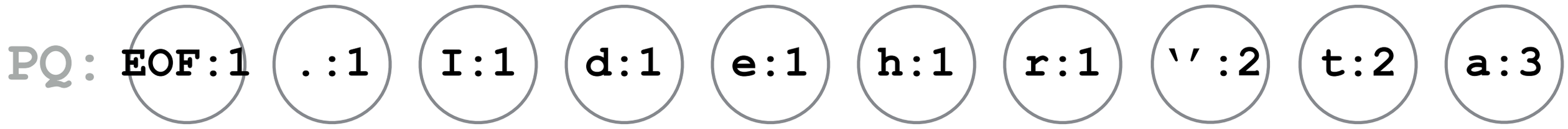
  -smaller node on the left, larger on the right

-reinsert new tree back into the queue

-but, what about ties?

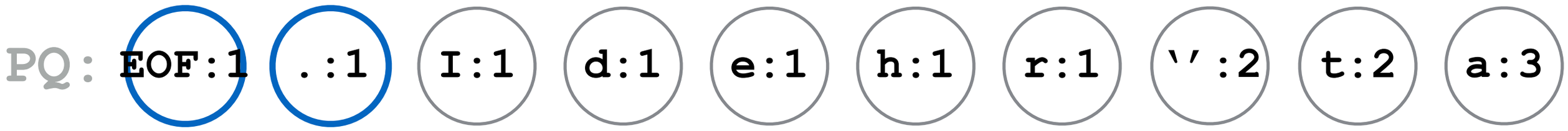
  -when trees have more than one node, break the tie with the ASCII value of the **leftmost** character

I heart data.



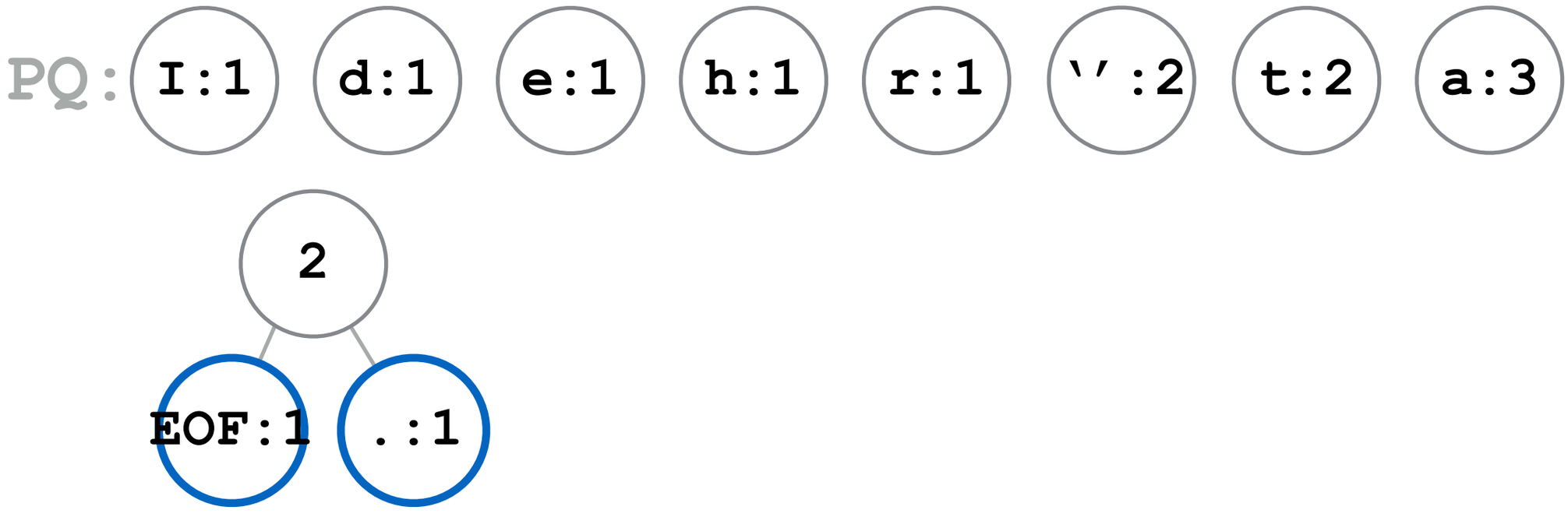
WHERE ARE THE TWO LOWEST WEIGHT TREES?

I heart data.

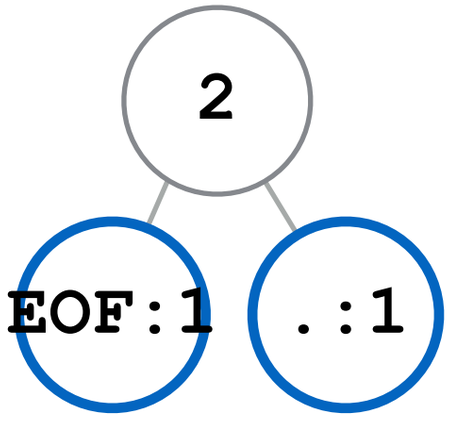


WHERE ARE THE TWO LOWEST WEIGHT TREES?

I heart data.

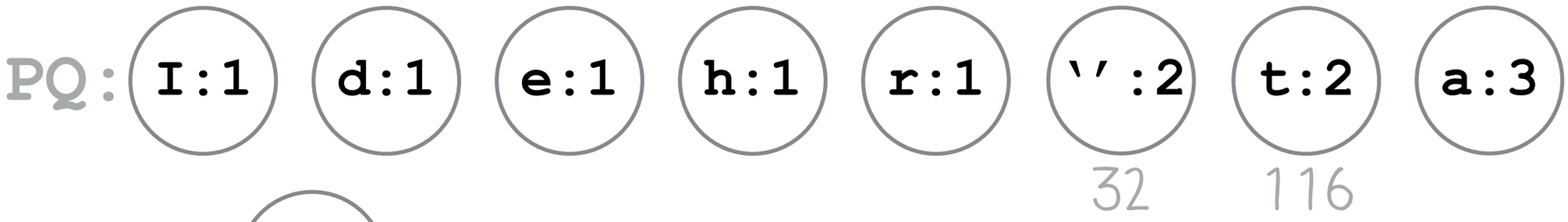


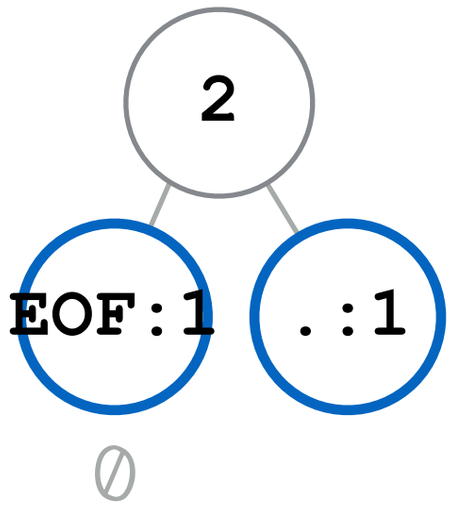
I heart data.



WHERE DO WE INSERT THIS NEW TREE?

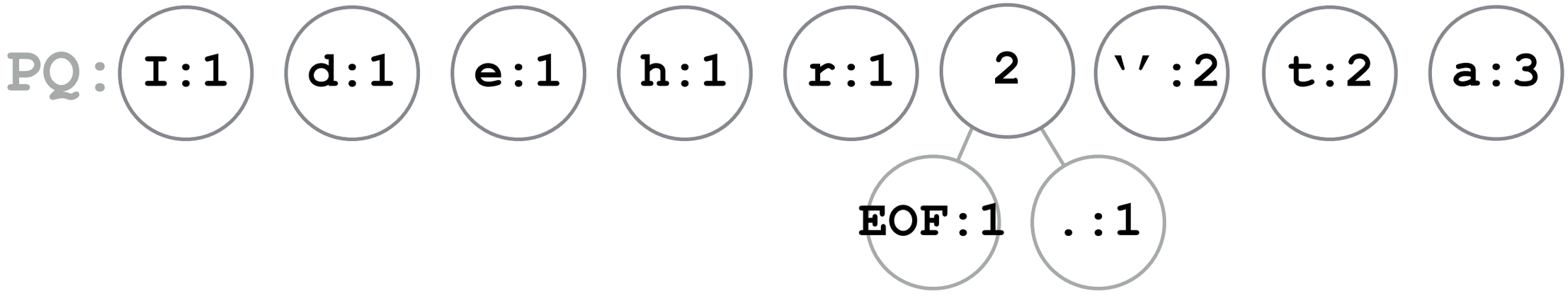
I heart data.



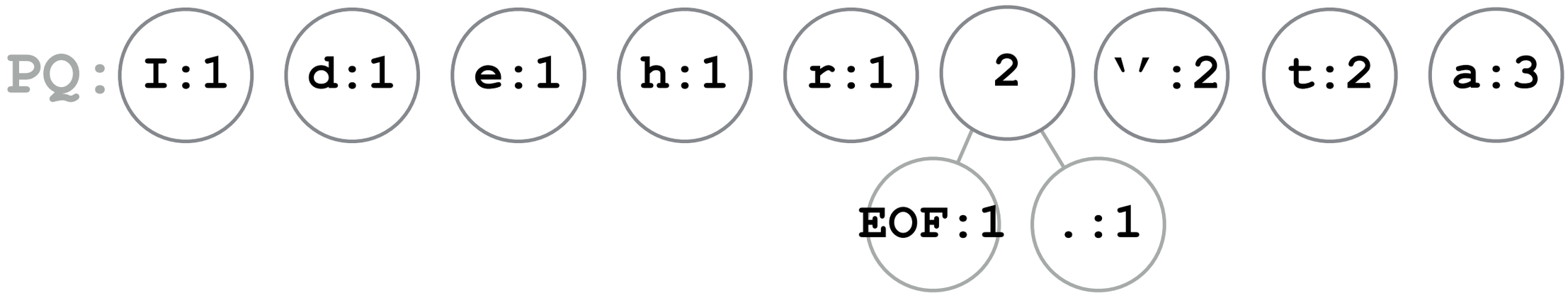


WHERE DO WE INSERT THIS NEW TREE?

I heart data.

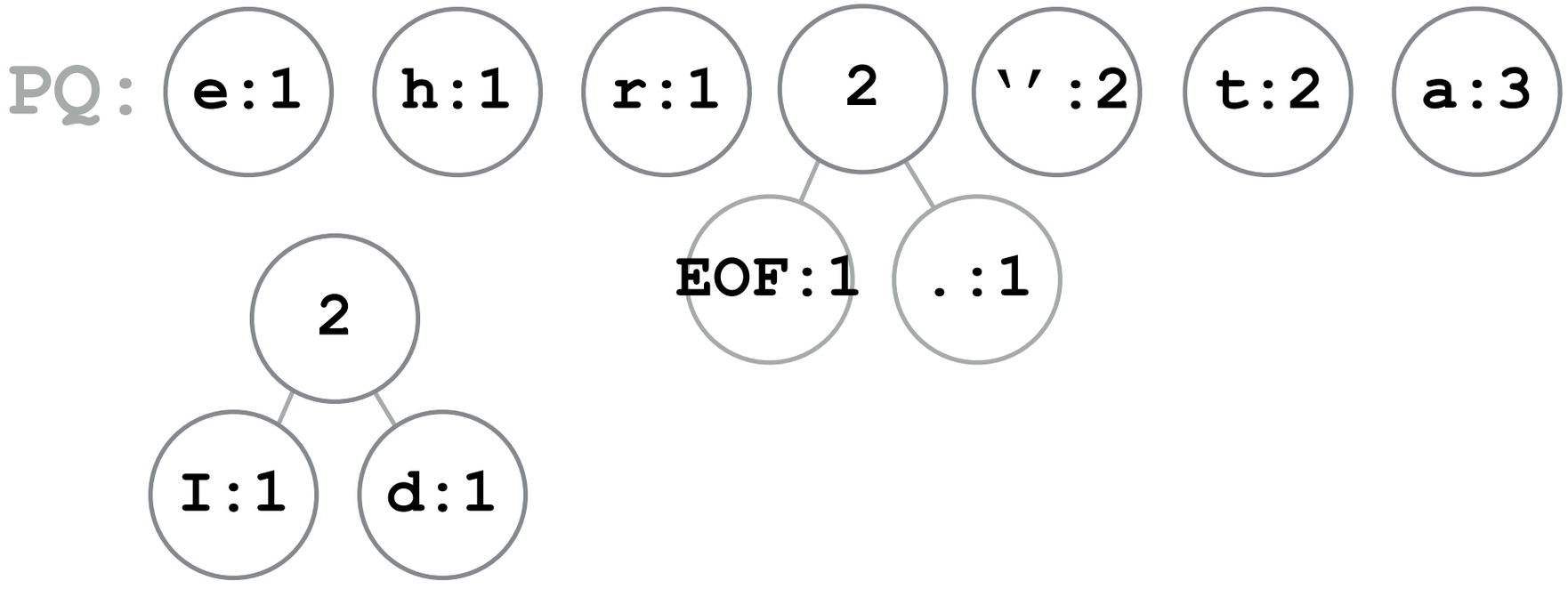


I heart data.

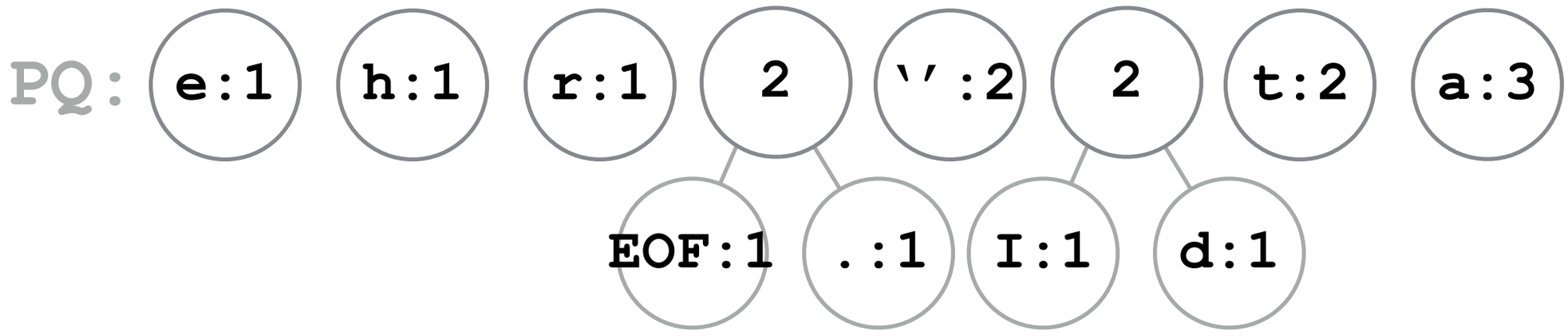


AND…repeat…

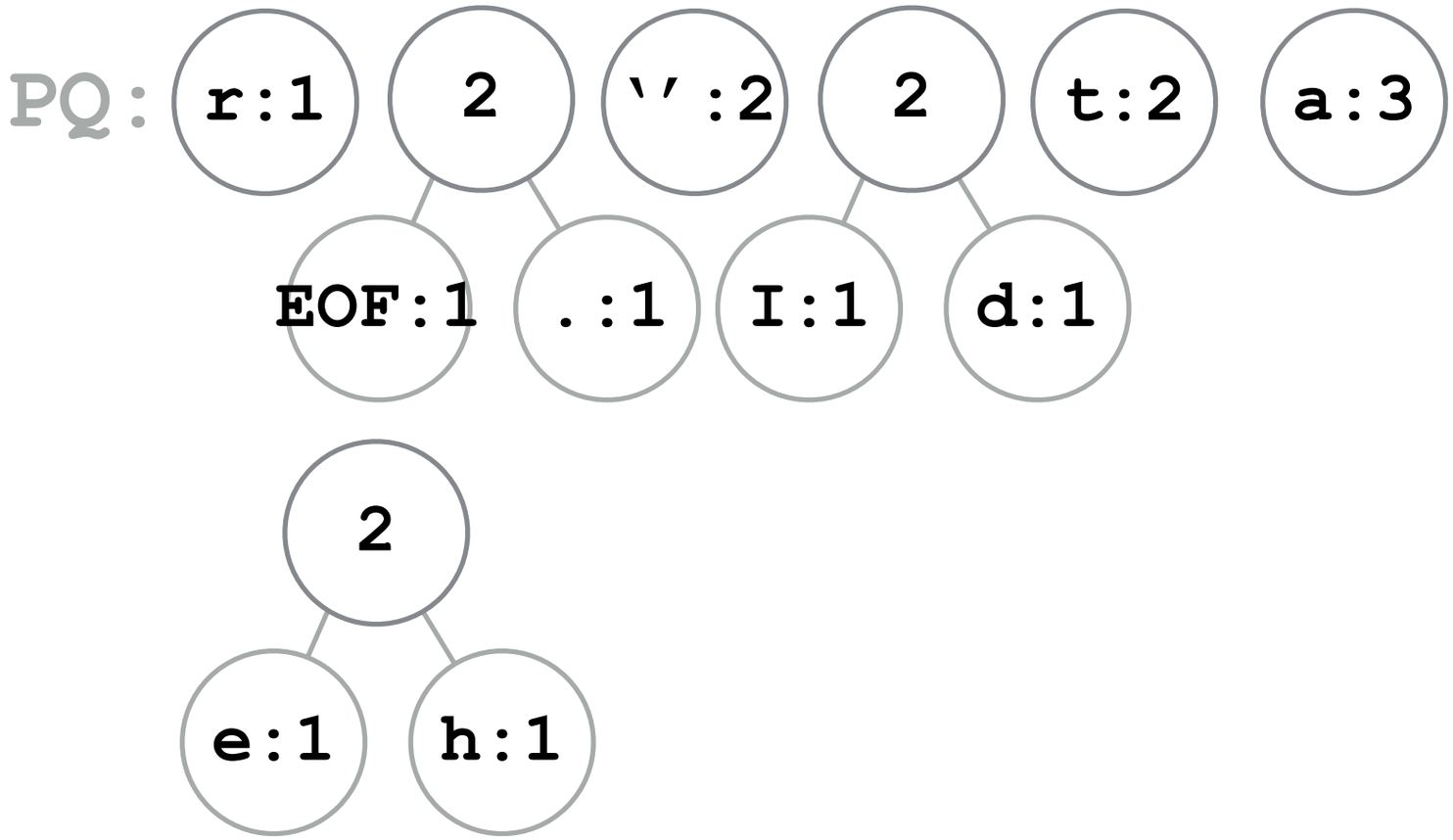
I heart data.



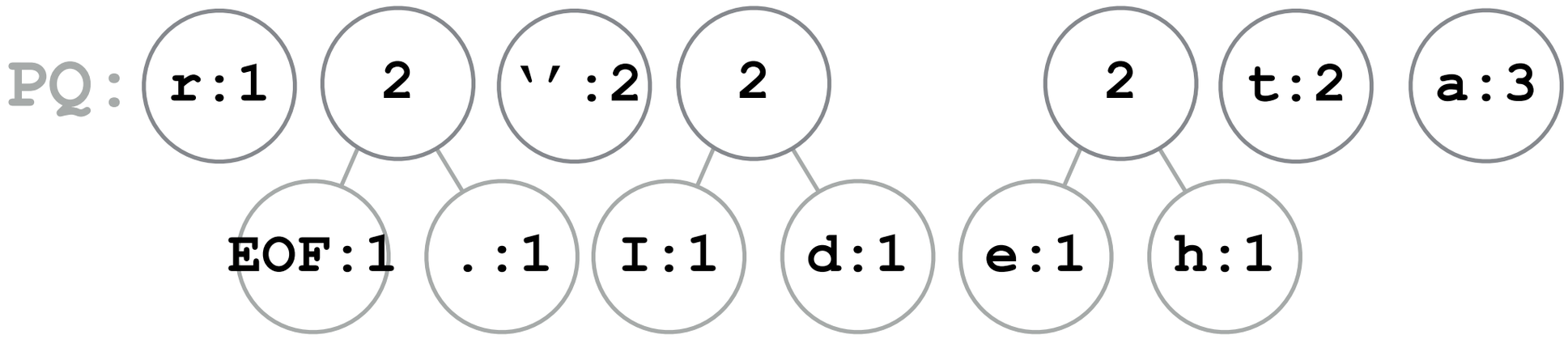
I heart data.



I heart data.

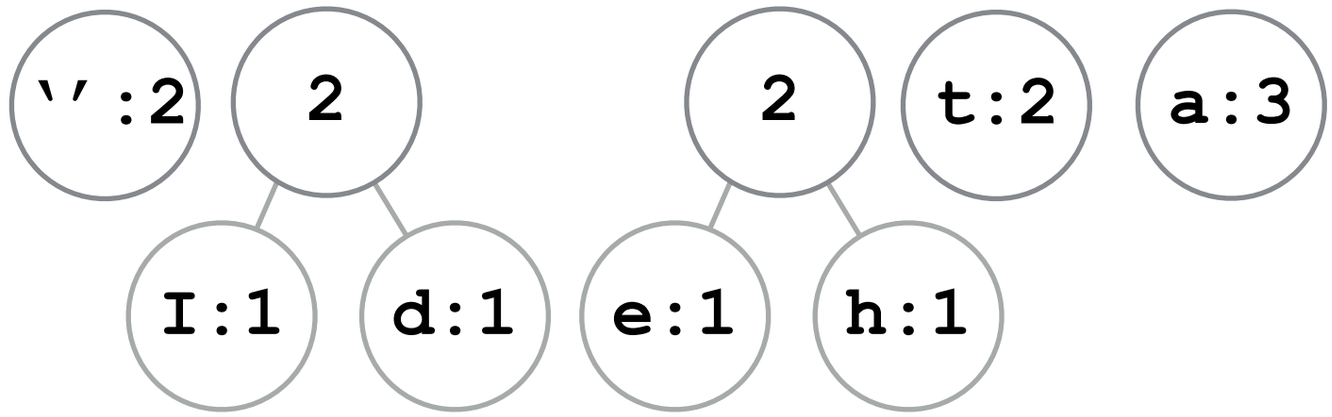


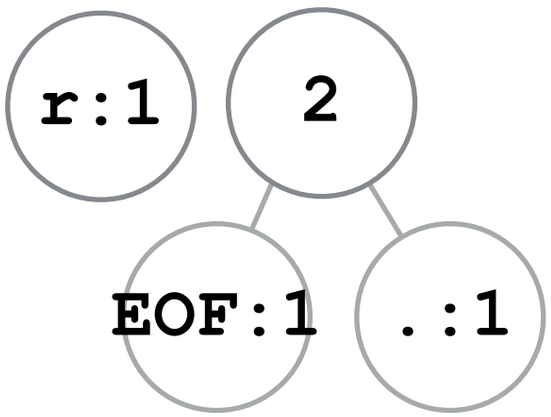
I heart data.



I heart data.

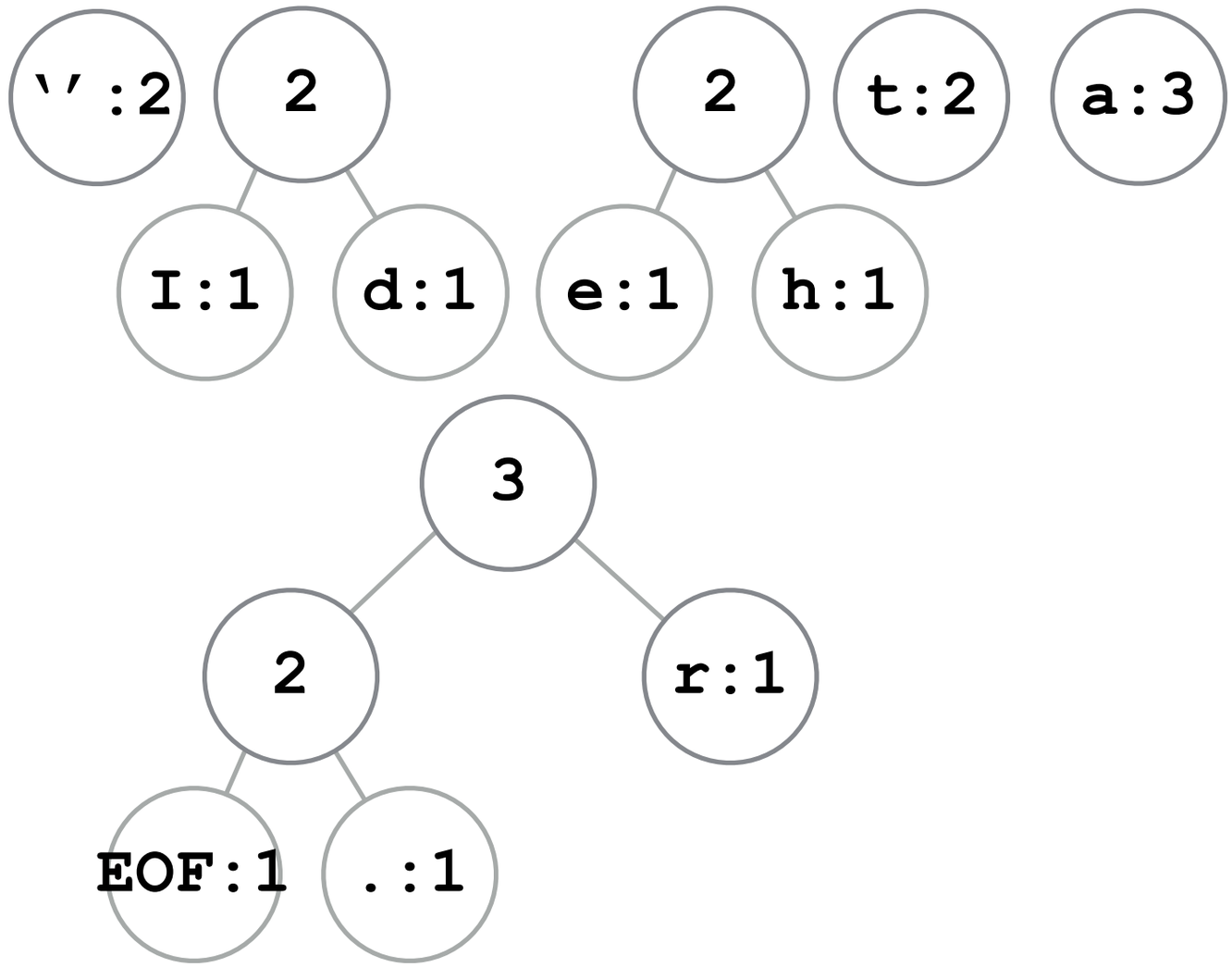
PQ:



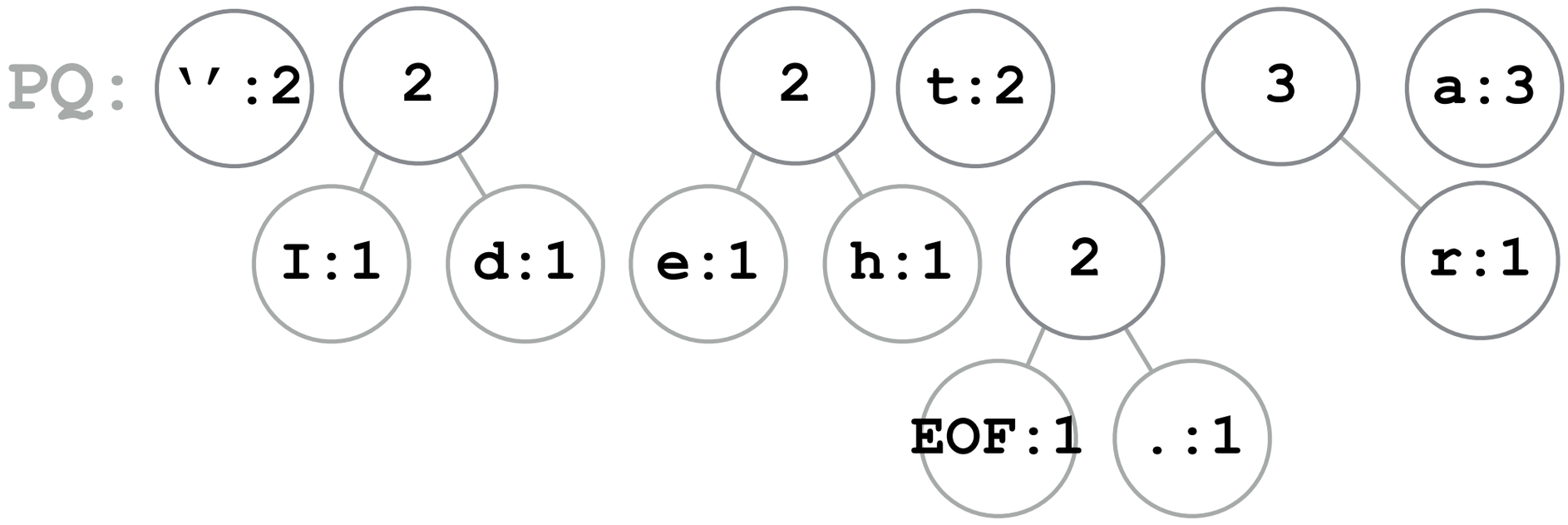


I heart data.

PQ:

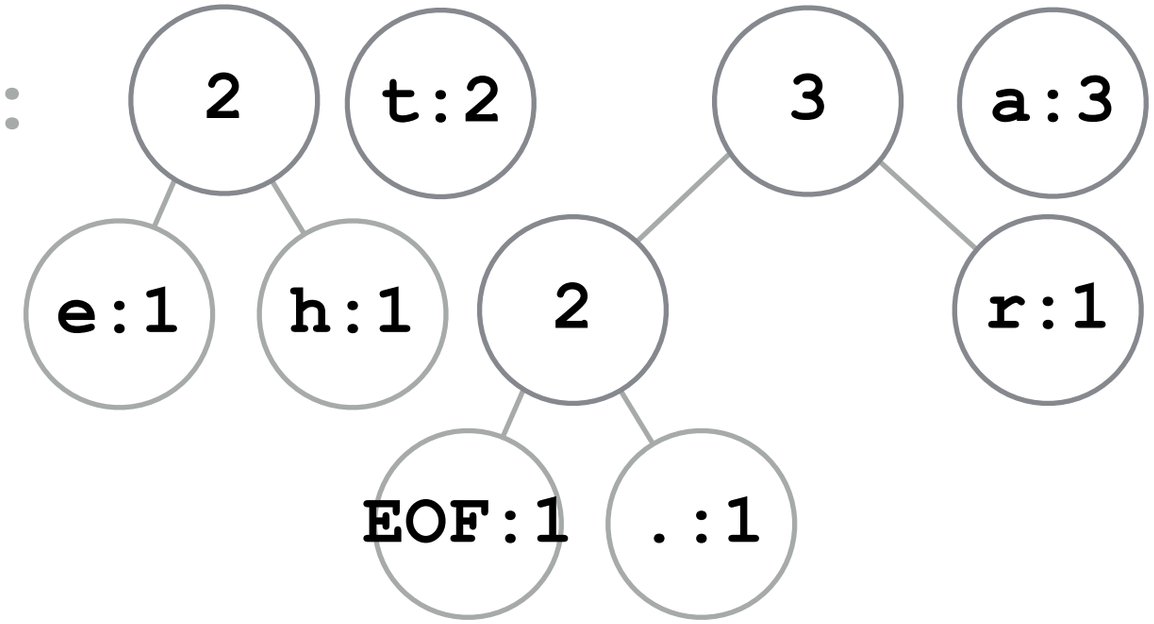


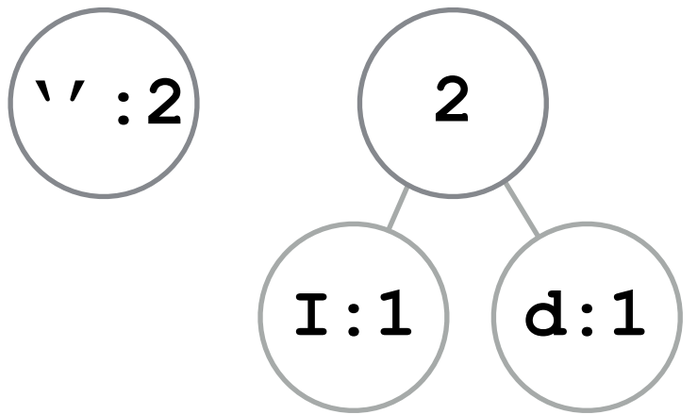
I heart data.



I heart data.

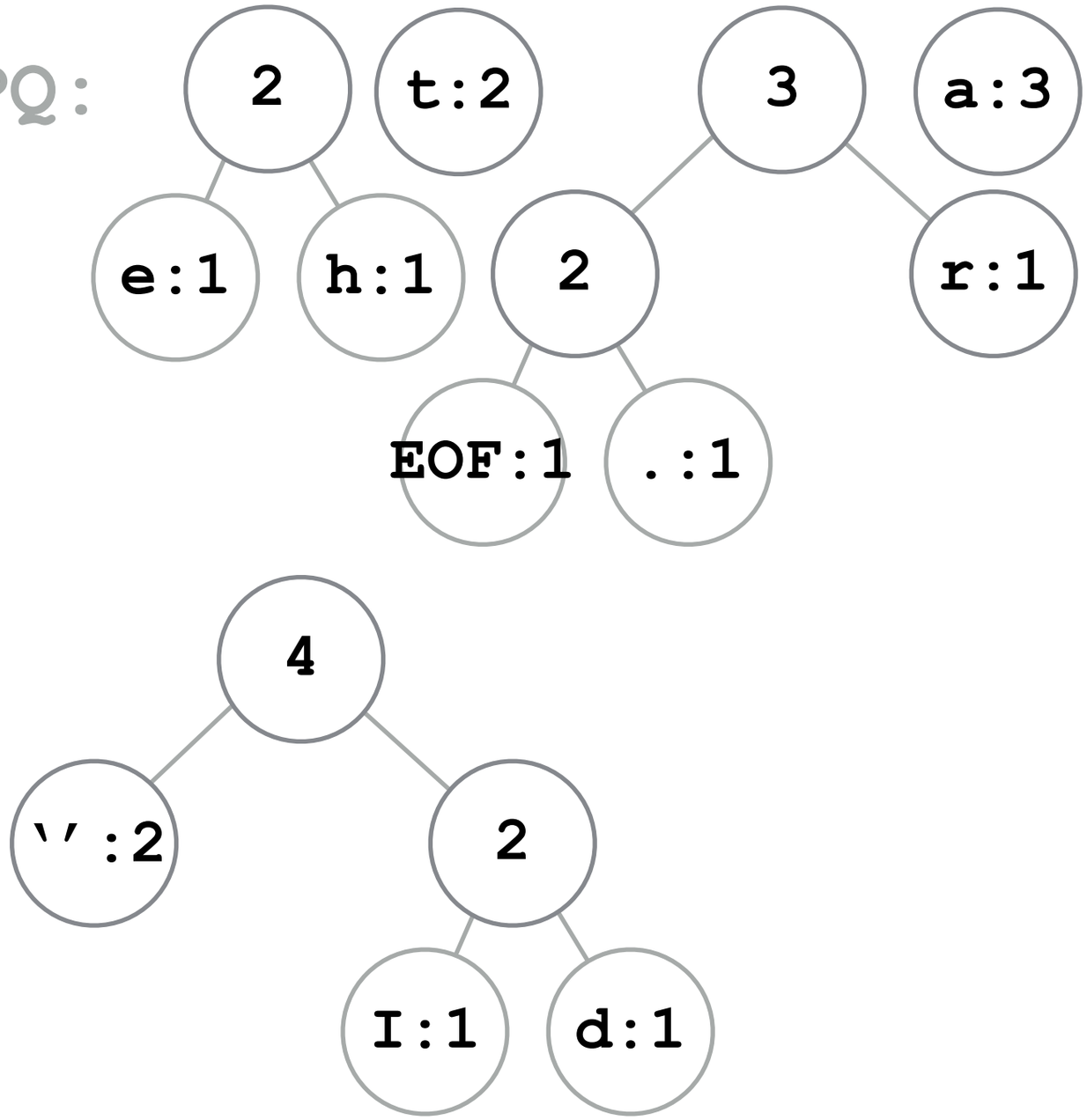
PQ:



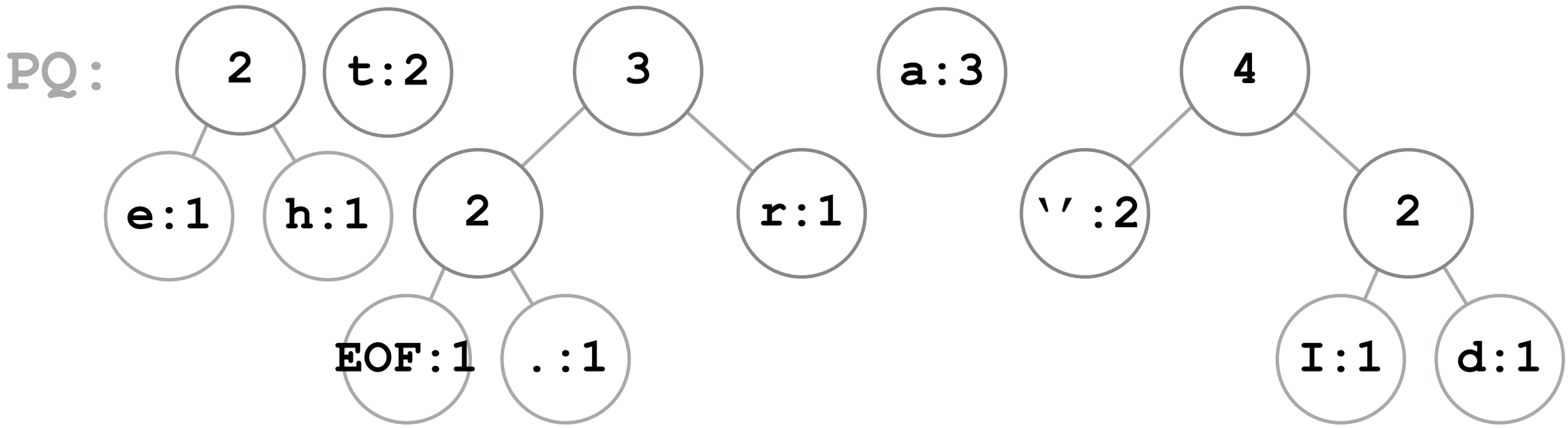


I heart data.

PQ:

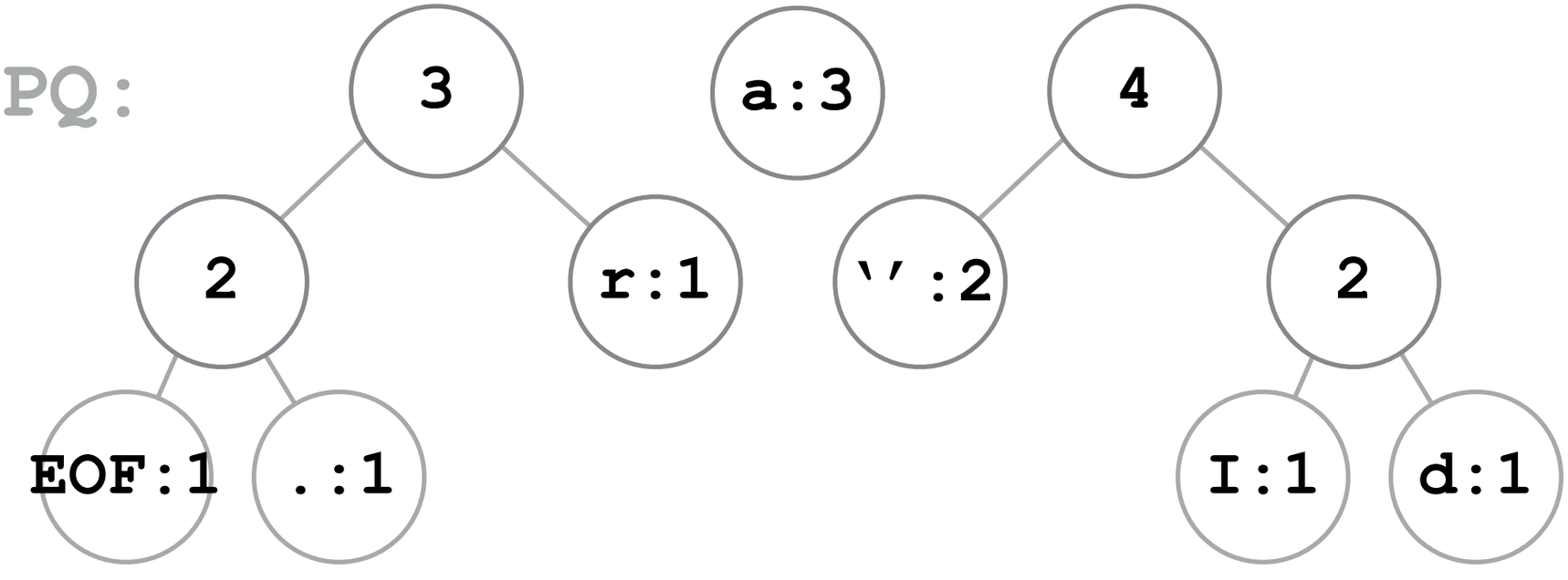


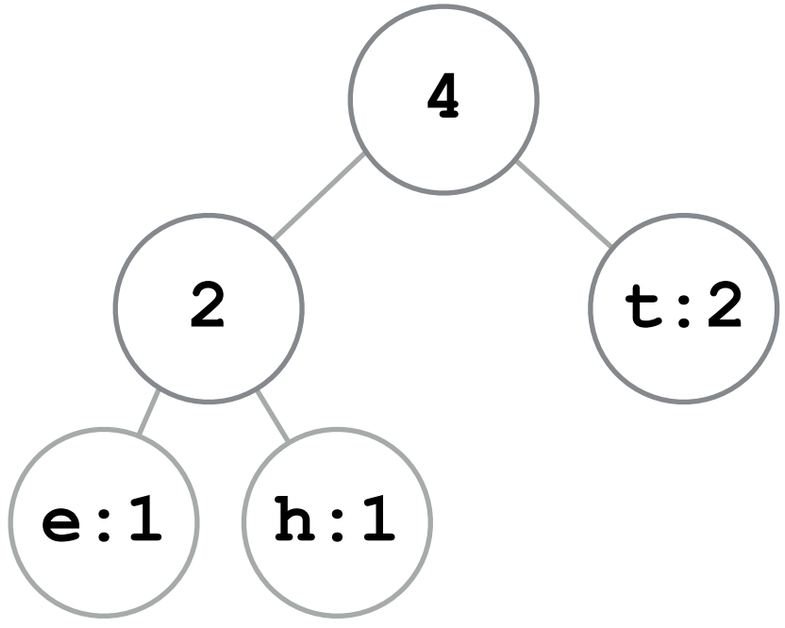
I heart data.



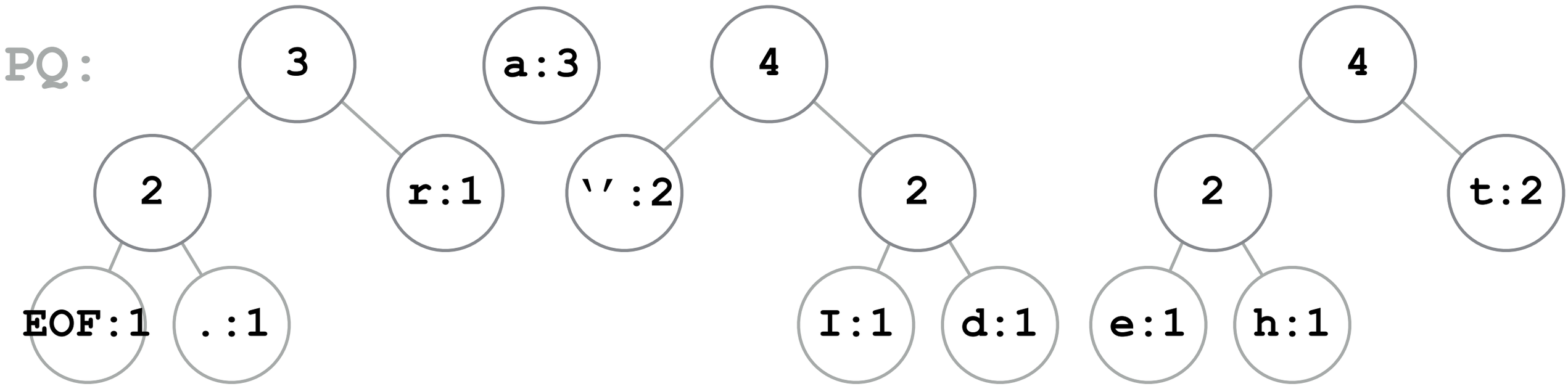
I heart data.

PQ:

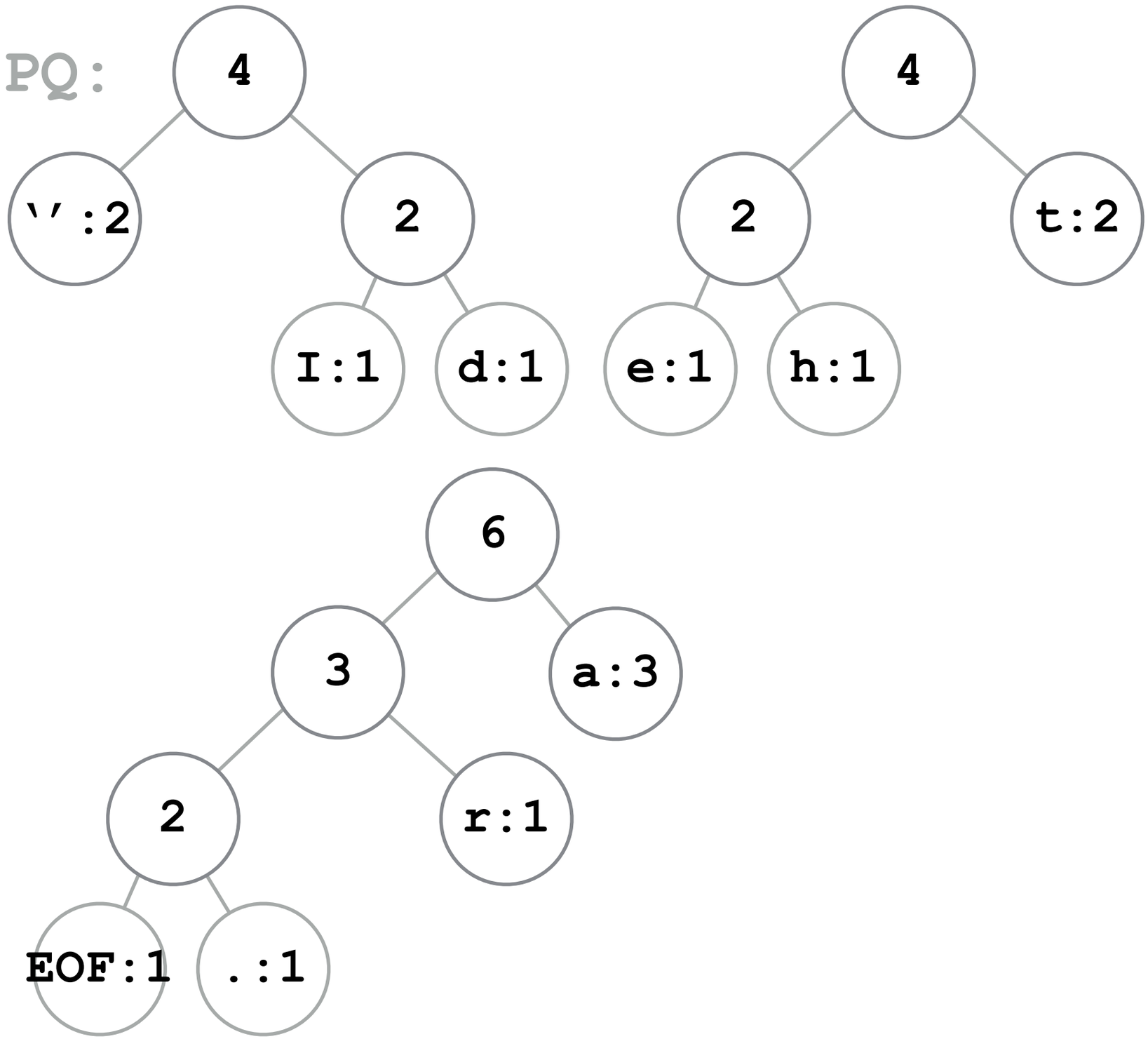




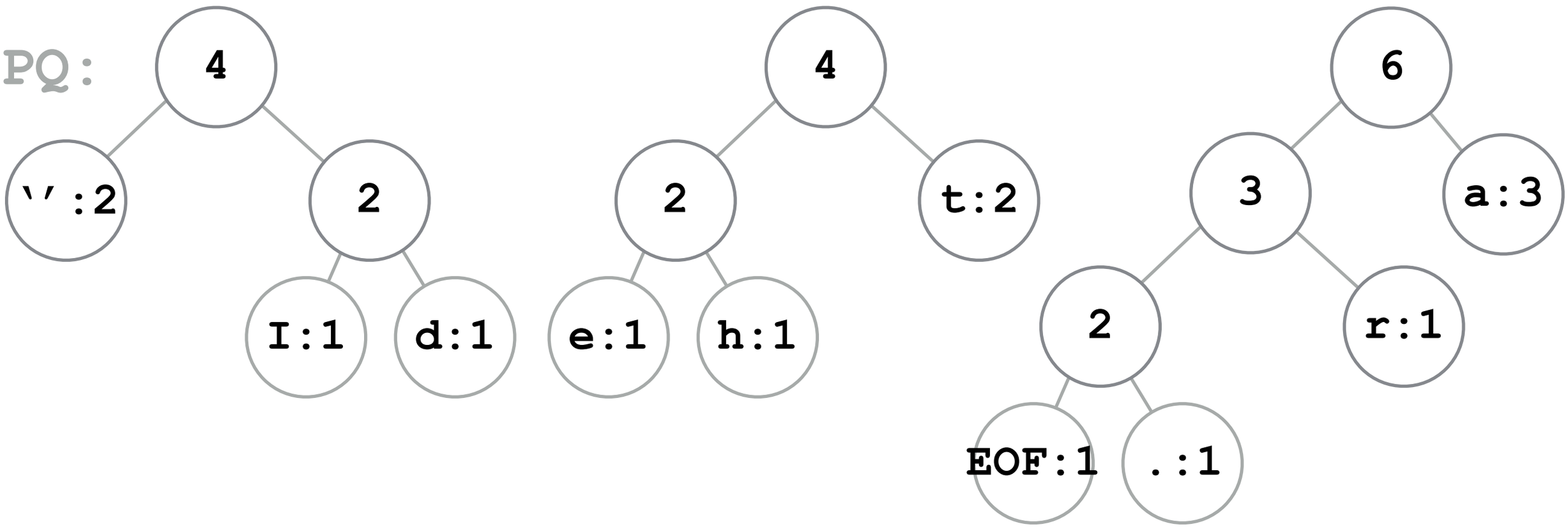
I heart data.



I heart data.

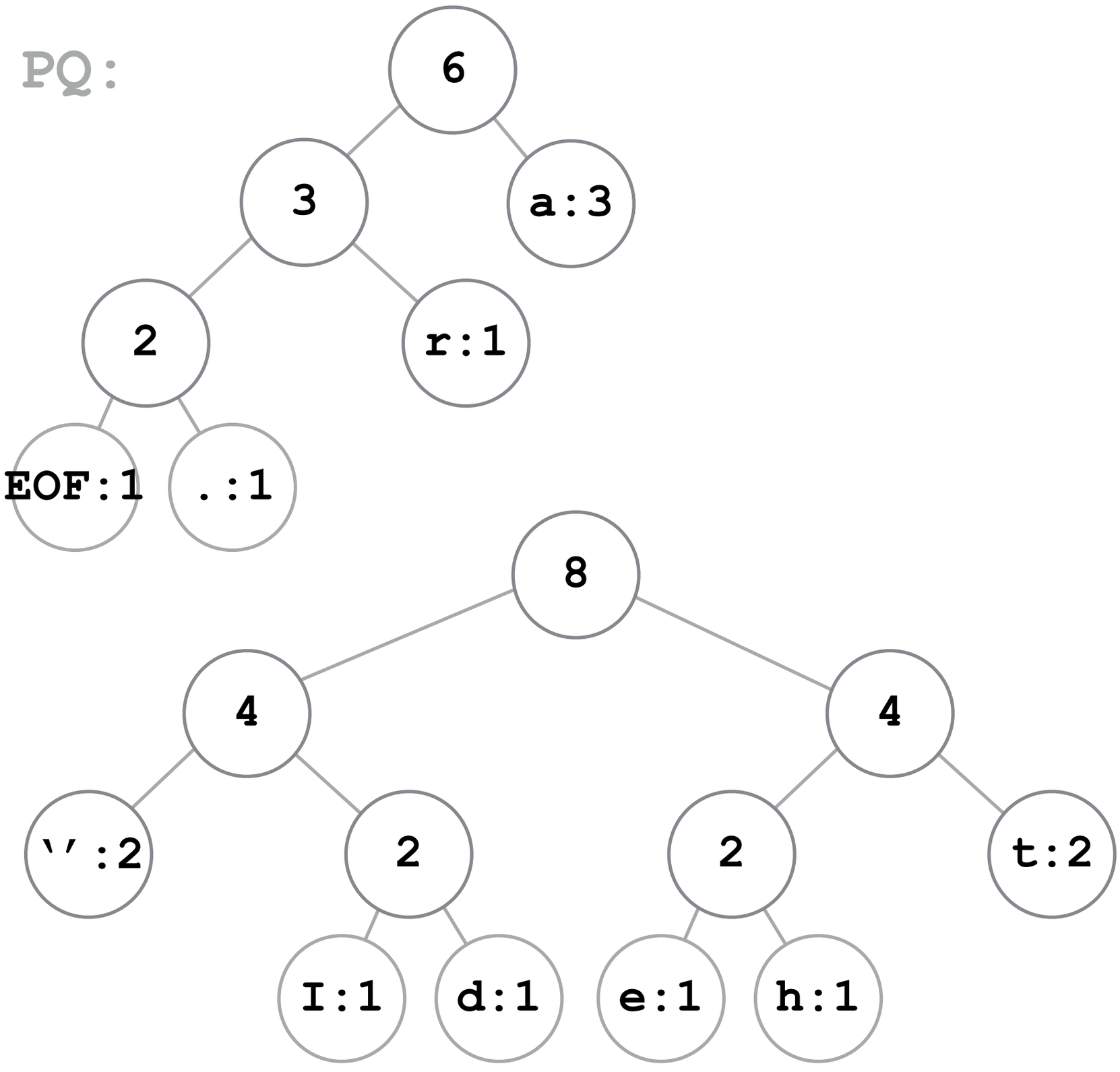


I heart data.



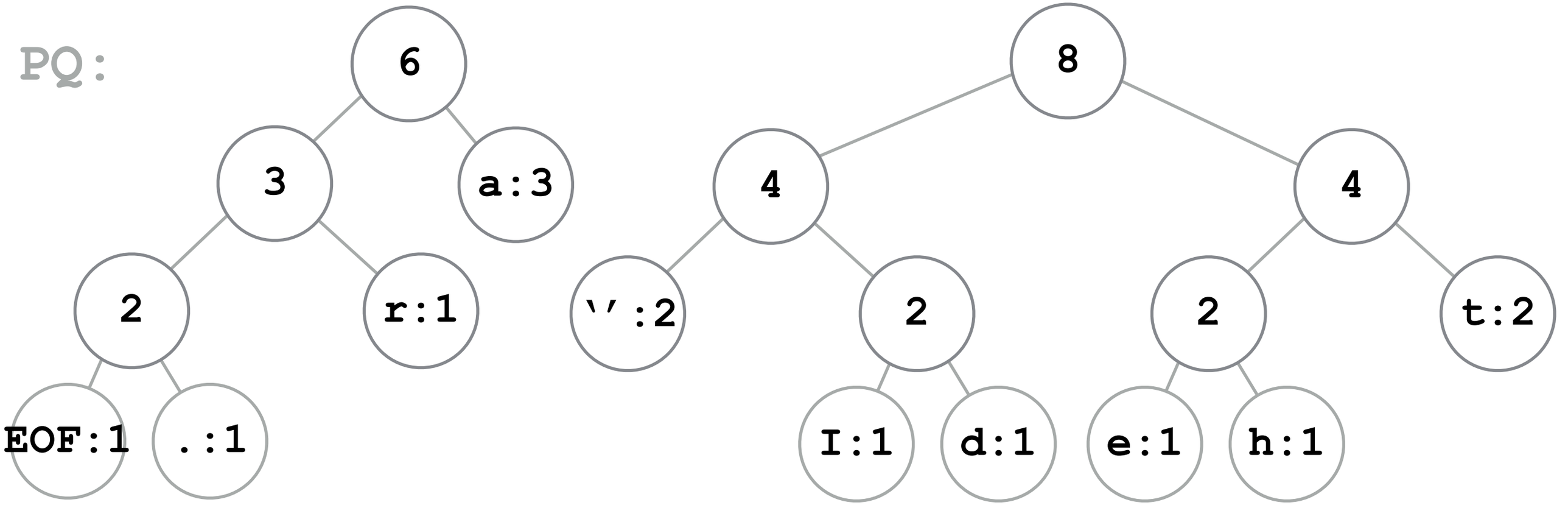
I heart data.

PQ:

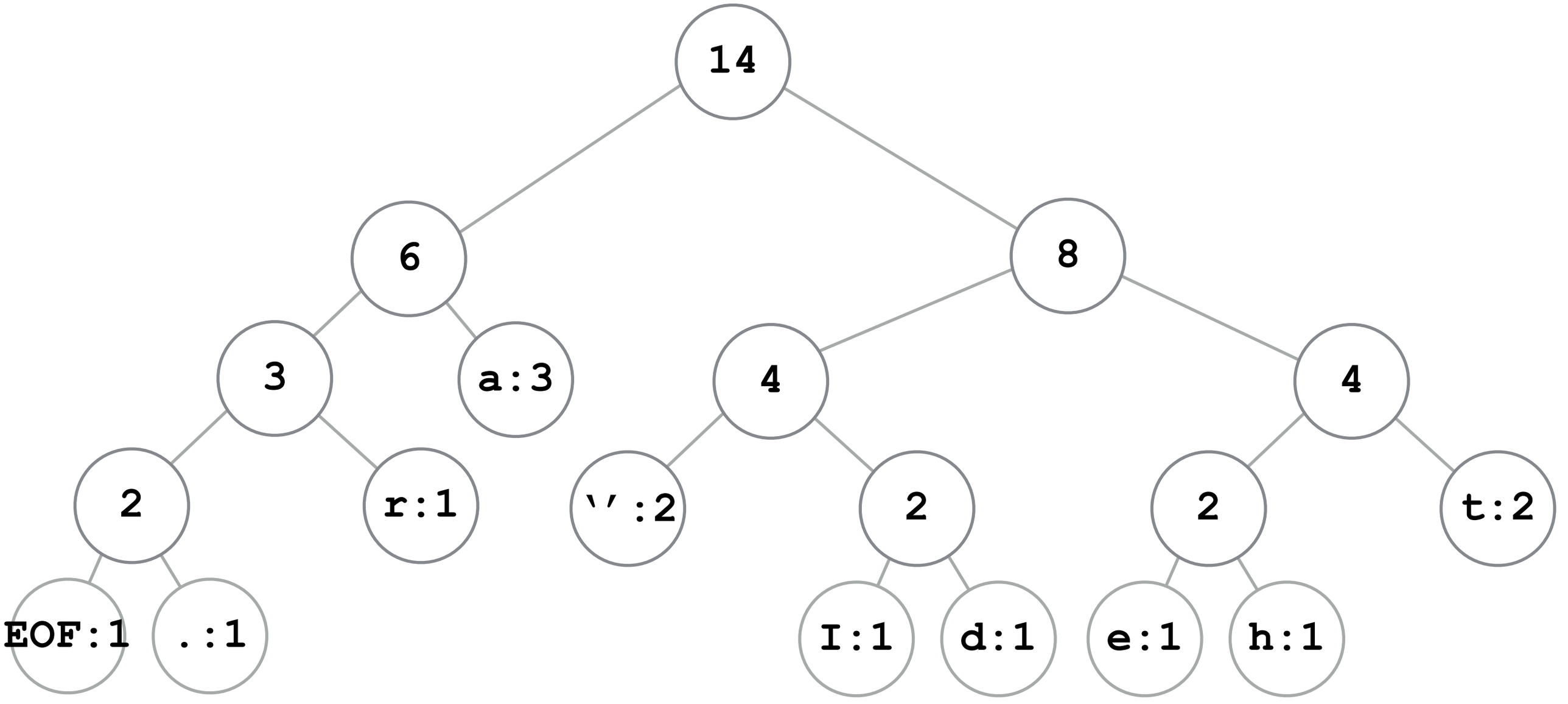


I heart data.

PQ:



I heart data.



```
while PQ contains at least two nodes
{
  dequeue two trees, compare leftmost
  nodes to determine left and right
  nodes, L and R

  node I = new internal node
  I.left = L
  I.right = R
  I.weight = L.weight + R.weight

  PQ.enqueue(I)
}

//PQ contains only one node
root = PQ.dequeue()
```

1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

-the compressed files needs info to be able to rebuild the same compression tree

-we call this the **header**

-what information do we need to store to reconstruct the compression tree?

1. store character frequencies
   -for every character
   -for only non-zero characters… must store pair

2. use a standardized letter frequency
   -use the same standard for compression and decompression

3. store the tree using a pre-order traversal
   -more complex, but smallest

-storing character frequencies

  -write each character, followed by the frequency

  -characters are 1 byte, integers are 4 bytes

  -so, each character will require 5 bytes to write

-how do we mark the end of the header (and beginning of the compressed string)?

  -write out `null` and `0`

-remember, we are going to write out our file as hex

I heart data.

character	hex value	frequency
I	49	1
' '	20	2
h	68	1
e	65	1
a	61	3
r	72	1
t	74	2
d	64	1
.	2E	1
EOF	00	1

```
49 00 00 00 01 20 00 00 00 02
68 00 00 00 01 65 00 00 00 01
61 00 00 00 03 72 00 00 00 01
74 00 00 00 02 64 00 00 00 01
2E 00 00 00 01 00 00 00 00 01
00 00 00 00 00
```

1. count occurrences of each character in a string

2. for each character, create a leaf node to store the character and count (ie. *weight*)

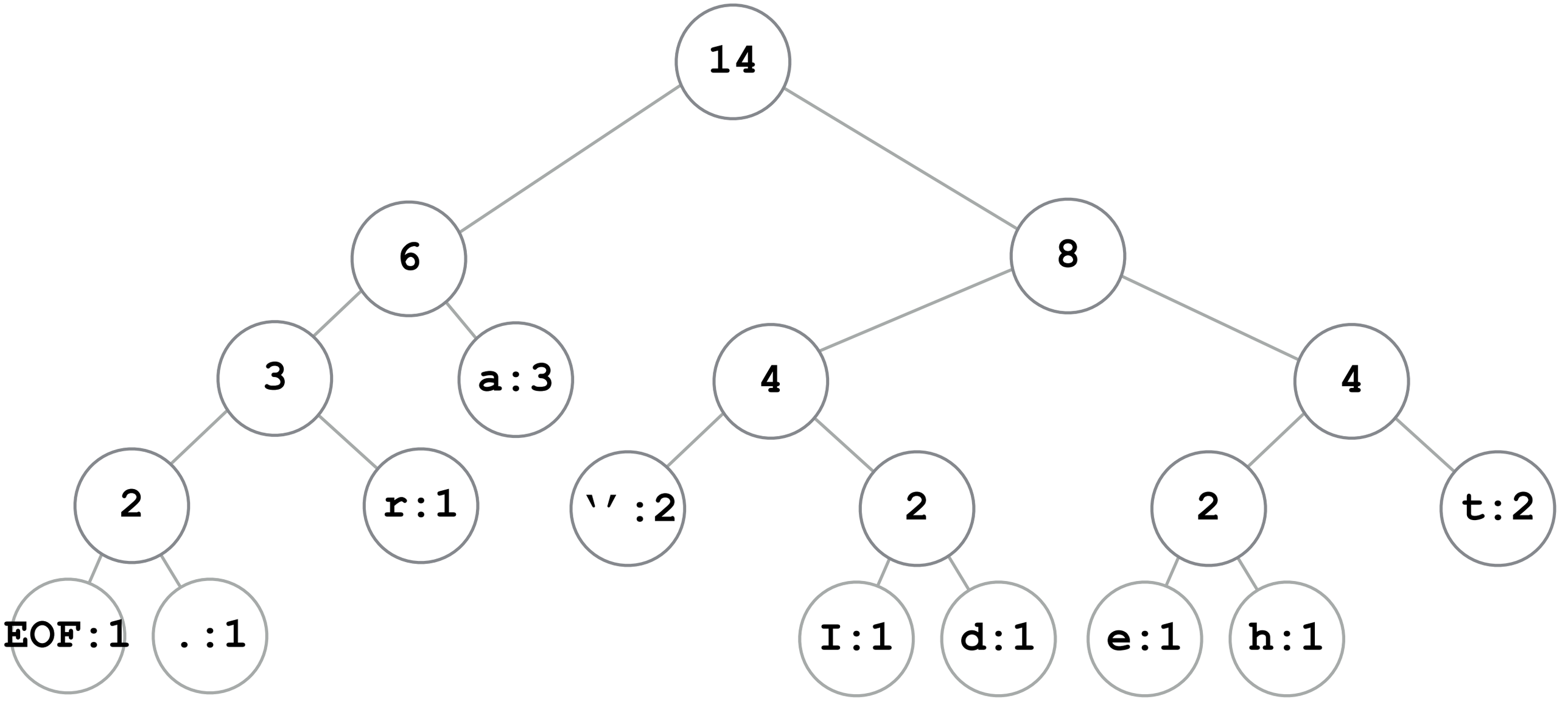
3. place each leaf node into a priority queue

4. construct the binary trie

5. write header with binary trie information

6. compress string using character codes

I heart data.



-want to create a look-up table to quickly convert from character to bit code

-what data structure can we use for this?

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010

I_heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 100

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 101

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 100

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1

NOW WHAT?

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1000 0

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1000 0

ARE WE DONE?

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1000 0000

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1000 0000

NOW, CONVERT TO HEX

I heart data.

character	hex value	bit-code
I	49	1010
' '	20	100
h	68	1101
e	65	1100
a	61	01
r	72	001
t	74	111
d	64	1011
.	2E	0001
EOF	00	0000

1010 1001 1011 1000 1001 1111
0010 1101 1110 1000 1000 0000

NOW, CONVERT TO HEX

A9 B8 9F 2D E8 80

I heart data.

character	hex value	bit-code
I	49	0110
' '	20	010
h	68	1101
e	65	1100
a	61	10
r	72	001
t	74	111
d	64	0111
.	2E	0001
EOF	00	0000

```
49 00 00 00 01 20 00 00 00 02
68 00 00 00 01 65 00 00 00 01
61 00 00 00 03 72 00 00 00 01
74 00 00 00 02 64 00 00 00 01
2E 00 00 00 01 00 00 00 00 01
00 00 00 00 00 A9 B8 9F 2D E8
80
```

I heart data.

character	hex value	bit-code
I	49	0110
' '	20	010
h	68	1101
e	65	1100
a	61	10
r	72	001
t	74	111
d	64	0111
.	2E	0001
EOF	00	0000

```
49 00 00 00 01 20 00 00 00 02
68 00 00 00 01 65 00 00 00 01
61 00 00 00 03 72 00 00 00 01
74 00 00 00 02 64 00 00 00 01
2E 00 00 00 01 00 00 00 00 01
00 00 00 00 00 A9 B8 9F 2D E8
80
```

WE JUST WENT FROM 13 BYTES TO 61 BYTES.

WHY SHOULD WE CARE ABOUT THIS????

decompression

WHAT STEPS DO I NEED TO TAKE TO DECOMPRESS THIS FILE?

```
49 00 00 00 01 20 00 00 00 02
68 00 00 00 01 65 00 00 00 01
61 00 00 00 03 72 00 00 00 01
74 00 00 00 02 64 00 00 00 01
2E 00 00 00 01 00 00 00 00 01
00 00 00 00 00 A9 B8 9F 2D E8
80
```

next time…

DOI:10.1145/1743546.1743567

Article development led by acmqueue
queue.acm.org

A survey of powerful visualization techniques, from the obvious to the obscure.

BY JEFFREY HEER, MICHAEL BOSTOCK, AND VADIM OGIEVETSKY

A Tour Through the Visualization Zoo

THANKS TO ADVANCES in sensing, networking, and data management, our society is producing digital information at an astonishing rate. According to one estimate, in 2010 alone we will generate 1,200

-homework

-assignment 12 due Tuesday night