Huffman character encoding

CMSC 420

You bigoted UTF-8 you!

- “Archimedes” will take 10 bytes to store…
- “Αρχιμήδης” will take >> 10 bytes to store!

You bigoted UTF-8 you!

- "Archimedes" will take 10 bytes to store…
- "Αρχιμήδης" will take >> 10 bytes to store!
- But with the vast majority of text being in English… people have adapted!
  - And the Earth keeps on turning.

You bigoted UTF-8 you!

- “Archimedes” will take 10 bytes to store…
- “Αρχιμήδης” will take >> 10 bytes to store!

- But with the vast majority of text being in English… people have adapted!
  - And the Earth keeps on turning.
- UTF8 is a **variable-length encoding** that has taken advantage of the frequency of English communications on the Web to present an overall very efficient way to store character data.

Quiz

- Is ASCII a variable – length encoding?

Yes

No

Quiz

- Is ASCII a variable – length encoding?

Yes

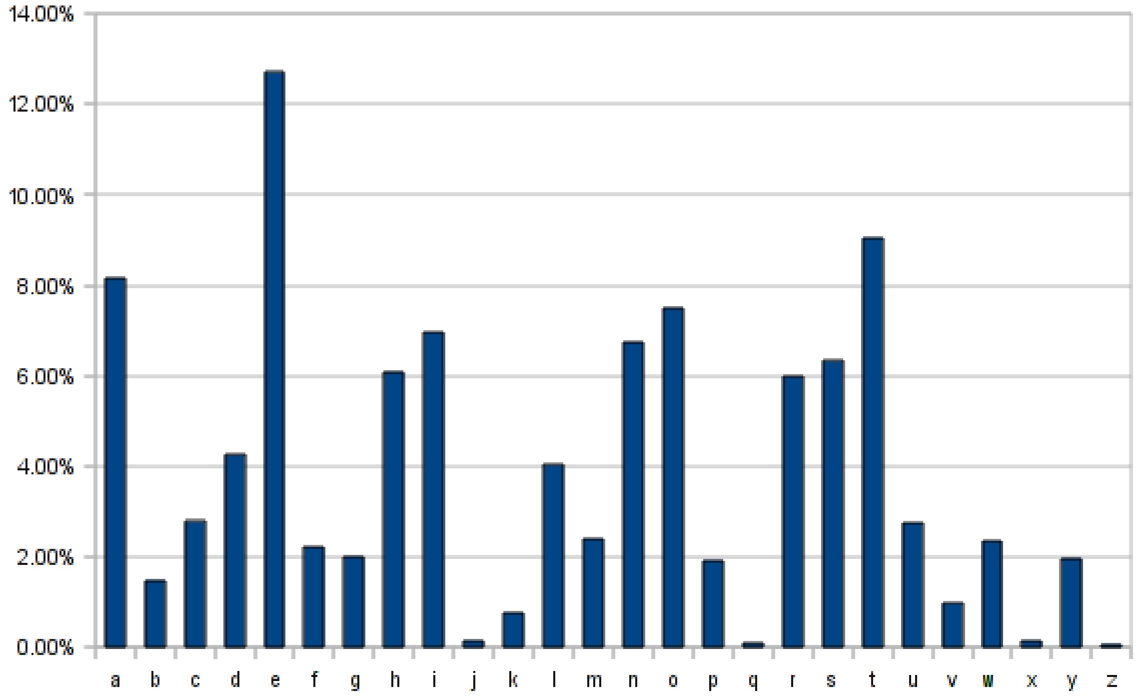
No

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

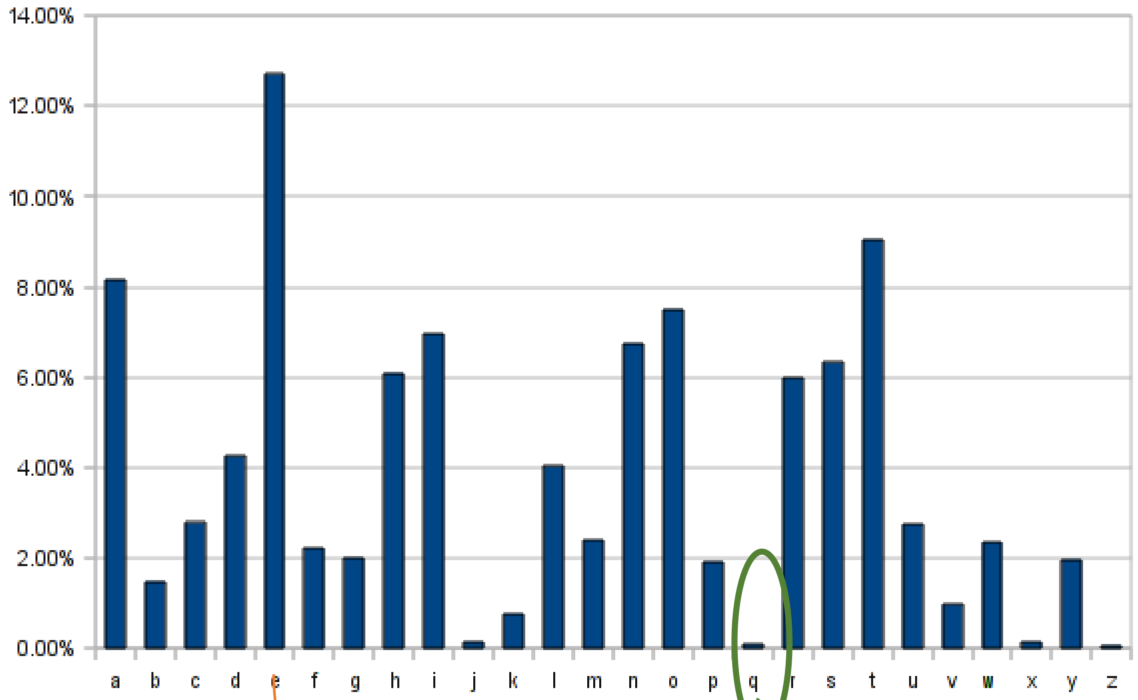
Source: www.LookupTables.com

- Remember: 7 bits per character.
- Can represent 128 characters, with about 30 of them non-printable.
- But still, a remarkably useful subset, with 51.6% of the Web written in English, and lots of those characters including:
  - Punctuation (, , . ,!, …) almost universally used
  - MANY characters that other Romance languages, like Spanish and French, use (a, s, t, r, …)
  - **ALL** different kinds of **whitespace (‘\0’, tabs, space, CRLF,…)**!

A different kind of redundancy…



A different kind of redundancy…



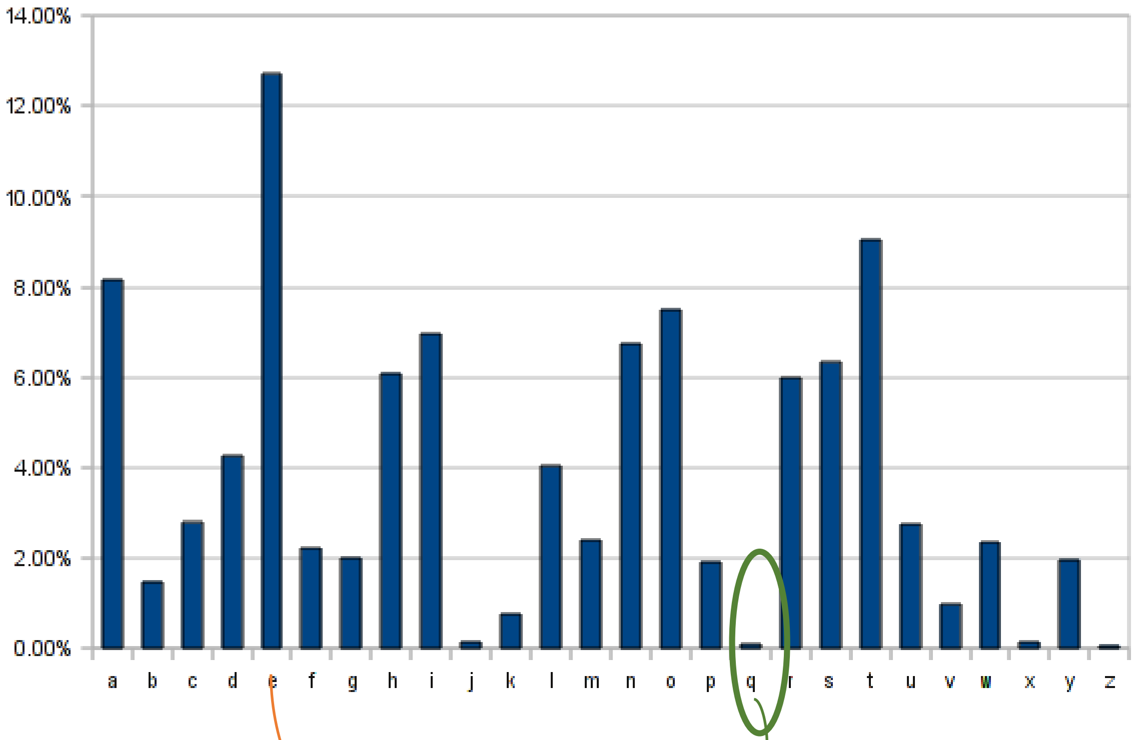
Encoding cost: 7 bits

Encoding cost: **also 7 bits!**

This doesn't seem fair!

A different kind of redundancy…

We should do better.



Encoding cost: 7 bits

Encoding cost: **also 7 bits!**

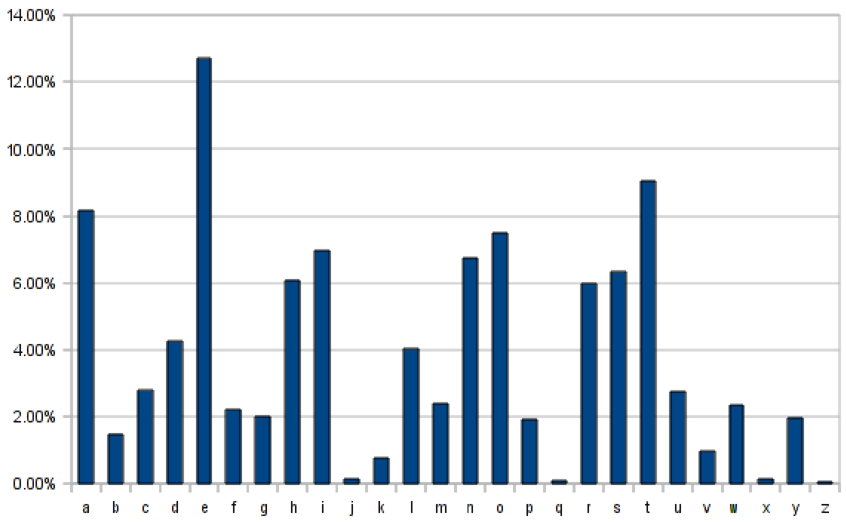
This doesn't seem fair!

Oh, by the way, bits matter now

- When we move from memory storage to transmitting information through a network, every little bit counts! 
- The bit is the most basic unit information in Claude Shannon's *Information Theory.*
- We essentially forget any kind of packet padding that might exist at the packet – level representation of data, and count the transmitted bits from source to sink.

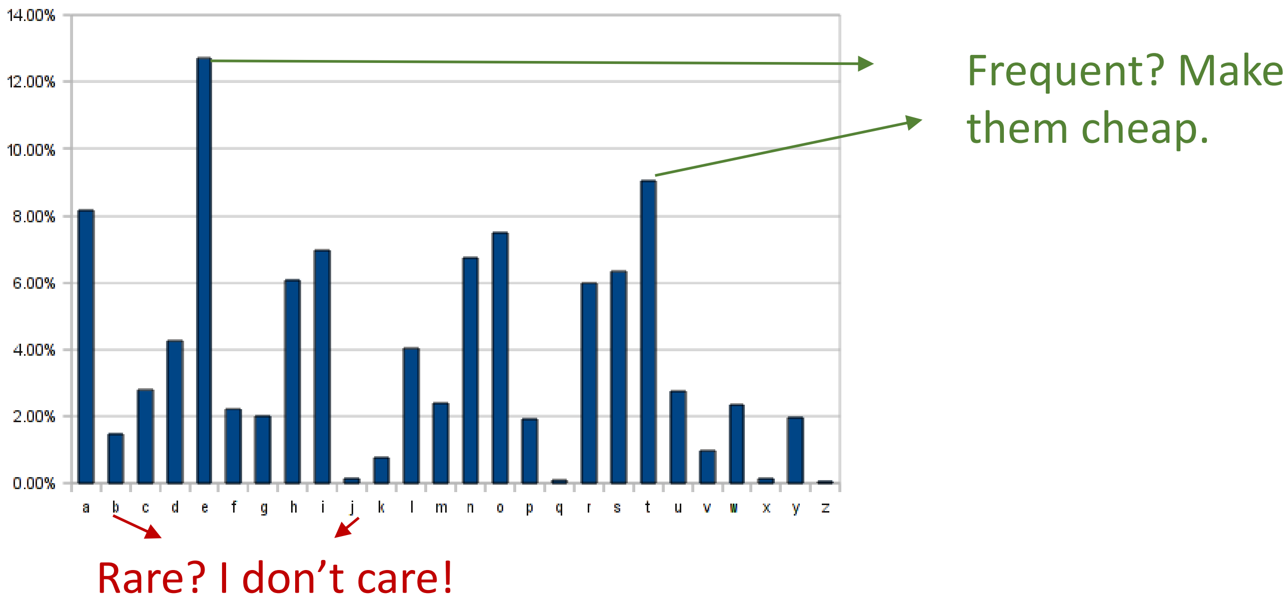
What we want

- For unigrams (characters) with the largest frequencies in this histogram to receive economical storage, since we expect to use them a lot, given their frequency.
- For those that we expect to rarely use, accept a long bit representation.



What we want

- For unigrams (characters) with the largest frequencies in this histogram to receive economical storage, since we expect to use them a lot, given their frequency.
- For those that we expect to rarely use, accept a long bit representation.



Careful!

- A crucial detail: We want our encodings to not be confusible! For example, if our encoding for e is 01, that of a is 00 and that of h is 0100, what should the sink interpret if it sees 0100?

- So any encoding algorithm we come up with has this constraint as well!

Careful!

- A crucial detail: We want our encodings to not be confusible! For example, if our encoding for e is 01, that of a is 00 and that of h is 0100, what should the sink interpret if it sees 0100?

- So any encoding algorithm we come up with has this constraint as well!
- We call this property the prefix property.

Huffman Coding

- We will describe a simple greedy algorithm that is guaranteed to maintain that property.
  - This algorithm is called **Huffman Coding**, pioneered by **David A. Huffman**
- The algorithm will build a binary trie, bottom-up.

Huffman Coding

- We will describe a simple greedy algorithm that is guaranteed to maintain that property.
  - This algorithm is called **Huffman Coding**, pioneered by **David A. Huffman**
- The algorithm will build a binary trie, bottom-up.
- Assumption: A loop has been run over the entire text, and a histogram of character frequencies has been constructed.
  - This can be parallelized across $k \geq 2$ threads so that the time is reduced from $|T|$ to $\frac{|T|}{k} + c$, where c is the small cost of adding up and normalizing the individual frequencies of the characters, as they were returned by the threads.

Huffman Coding

- We insert all the <character, frequency> pairs in a Priority Queue
- Our goal will be, at every point in time, to have fast access to the two nodes with the smallest frequencies

Building the binary tree….

- Running example string:

“Jason loves chocolate.”

Building the binary tree….

- Running example string:

“Jason loves chocolate.”

Don’t forget spaces and punctuation!

Building the binary tree….

- Running example string:

"Jason loves chocolate."

Don't forget spaces and punctuation!

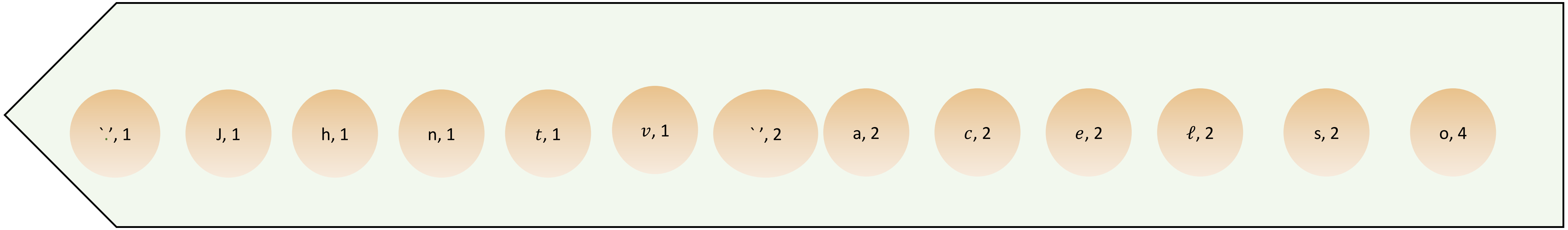
1. Scan the text and build nodes containing <character, #occurrences> pairs

SPACE CHARACTER!

J, 1 | a, 2 | s, 2 | o, 4 | n, 1 | ℓ, 2 | v, 1 | e, 2 | c, 2 | h, 1 | t, 1 | ` ', 2 | `.', 1

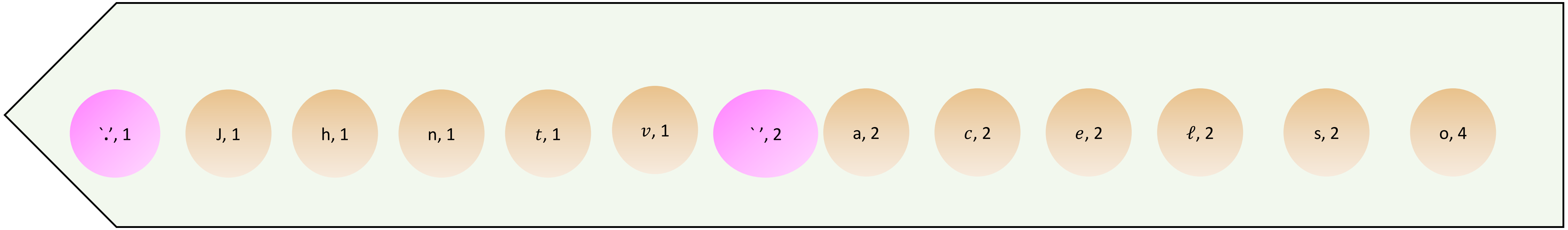
Building the binary tree....

2. Put all of the nodes in a Priority Queue sorted on frequencies, in ascending order. Smaller is higher priority.
   - Ties will be broken lexicographically.
     - Again, smaller up front ☺



Building the binary tree….

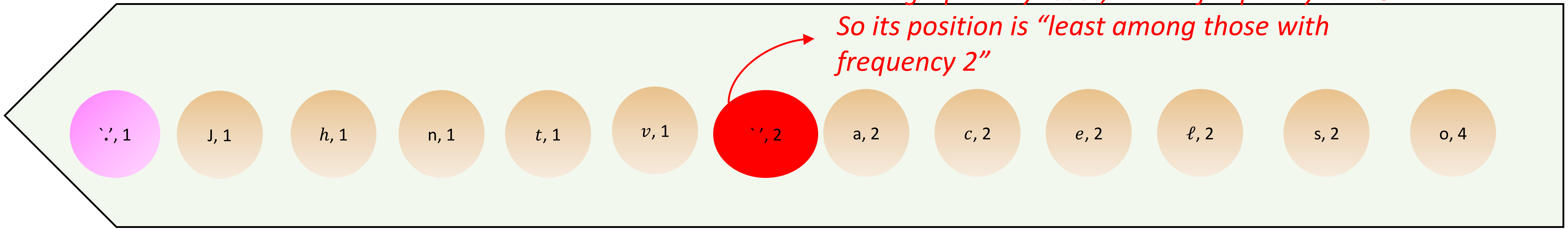
2. Put all of the nodes in a Priority Queue sorted on frequencies, in ascending order. Smaller is higher priority.
    - Ties will be broken lexicographically.
        - Again, smaller up front ☺
    - *Lexicographically, periods < characters and spaces < periods < characters*
    - *Don't worry: In exams,* ***you will be given an ASCII table*** *to deduce lexicographic order on the fly.*



Building the binary tree….

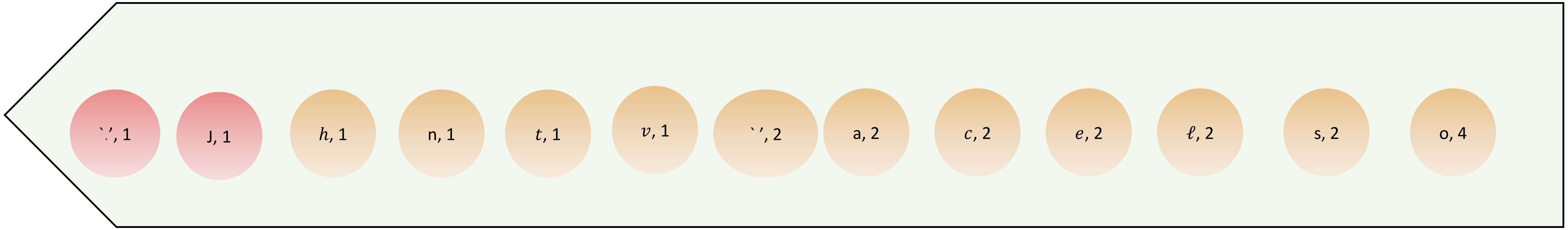
2. Put all of the nodes in a Priority Queue sorted on frequencies, in ascending order. Smaller is higher priority.
   - Ties will be broken lexicographically.
     - Again, smaller up front ☺
   - *Lexicographically, periods < characters and spaces < periods < characters*
   - *Don't worry: In exams, **you will be given an ASCII table** to deduce lexicographic order on the fly.*

*This is not an error; SPACE might be lexicographically **least**, but its frequency is 2 ☺*
So its position is "least among those with frequency 2"



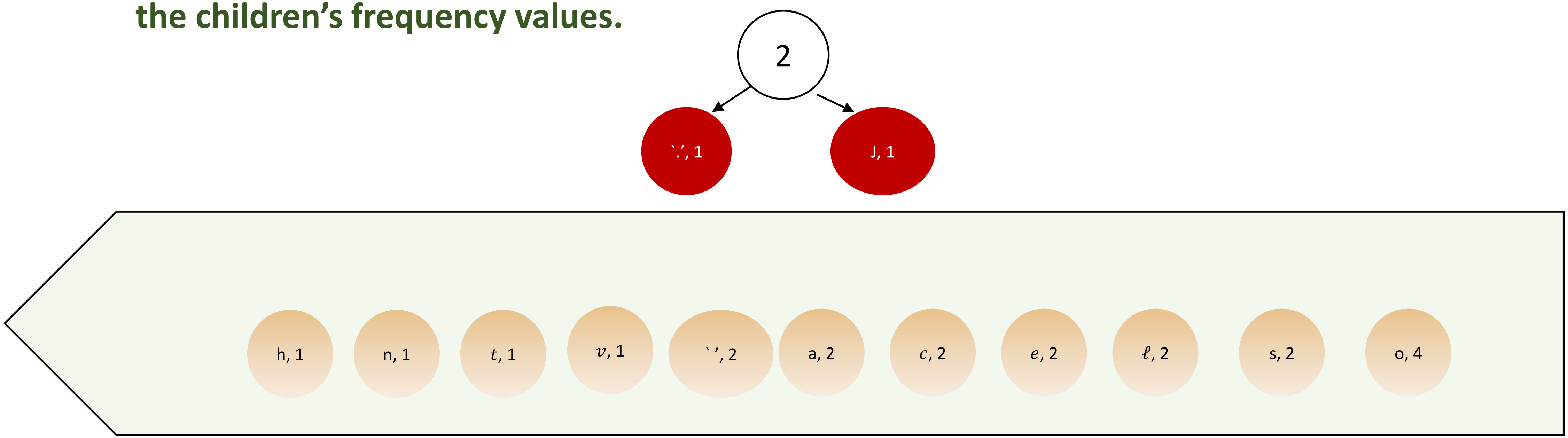
Building the binary tree….

3. As long as the queue has more than 1 elements in it
   - Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
     - For a **MinHeap**-based PQ, this is **extremely fast:** $2 \cdot \log_2 |A|$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535$ we have $2 \cdot \log_2 |A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)



Building the binary tree….

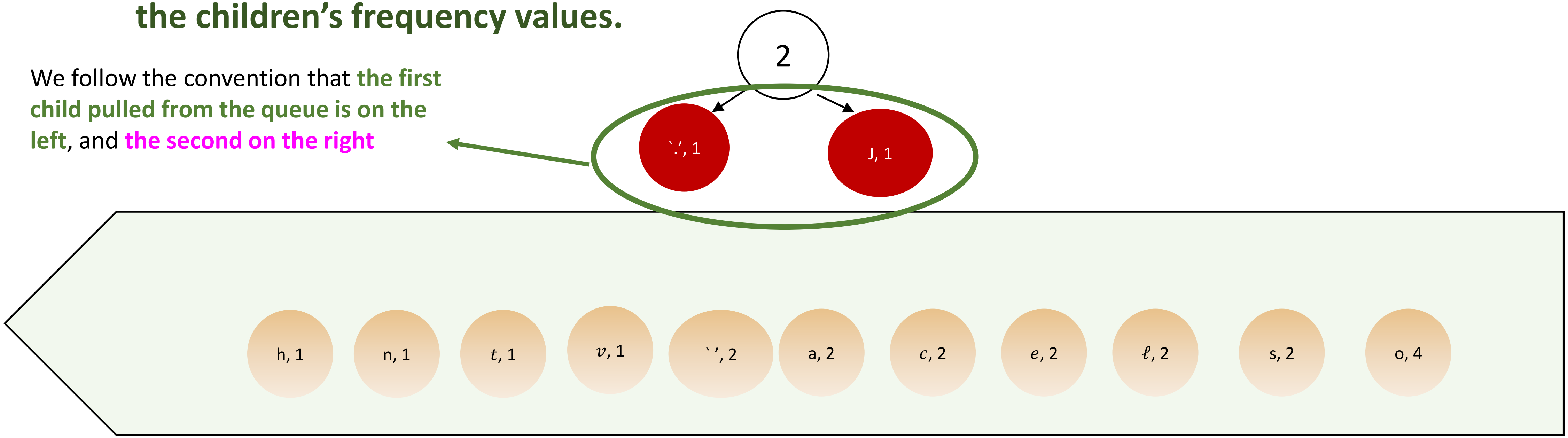
3. As long as the queue has more than 1 elements in it
   - Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
     - For a **MinHeap**-based PQ, this is **extremely fast:** $\mathcal{O}(2 \cdot \log_2 |A|)$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535 \Rightarrow 2 \cdot \log_2 |A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)
   - Create a **new node** with those two nodes as children and a frequency value equal to the **sum of the children's frequency values.**



Building the binary tree….

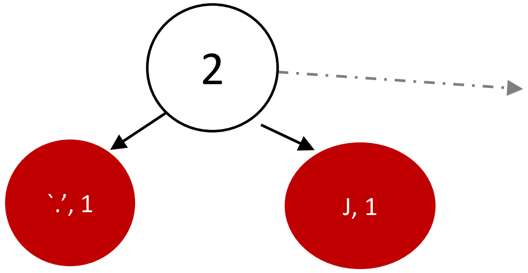
3. As long as the queue has more than 1 elements in it
   - ➢ Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
     - For a **MinHeap**-based PQ, this is **extremely fast:** $\mathcal{O}(2 \cdot \log_2|A|)$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535 \Rightarrow 2 \cdot \log_2|A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)
   - ➢ Create a **new node** with those two nodes as children and a frequency value equal to the **sum of the children's frequency values.**

We follow the convention that **the first child pulled from the queue is on the left**, and **the second on the right**

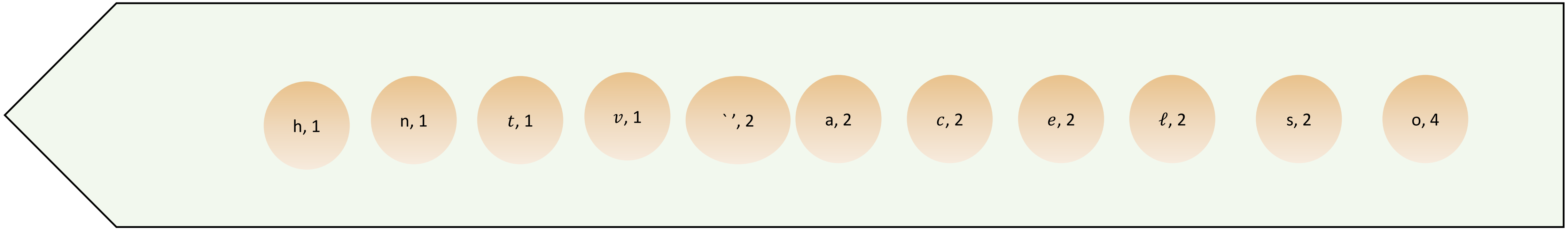


Building the binary tree….

3. As long as the queue has more than 1 elements in it
   - ➢ Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
     - For a **MinHeap**-based PQ, this is **extremely fast:** $\mathcal{O}(2 \cdot \log_2|A|)$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535 \Rightarrow 2 \cdot \log_2|A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)
   - ➢ Create a **new node** with those two nodes as children and a frequency value equal to the **sum of the children's frequency values.**



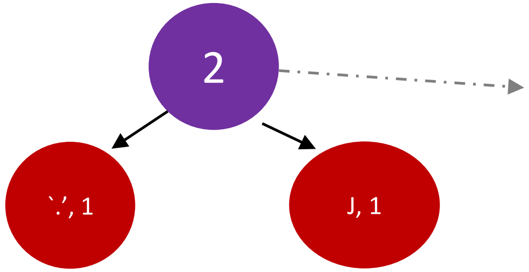
Important implementation detail; These nodes are assumed to hold the NULL character (first in ASCII table!)



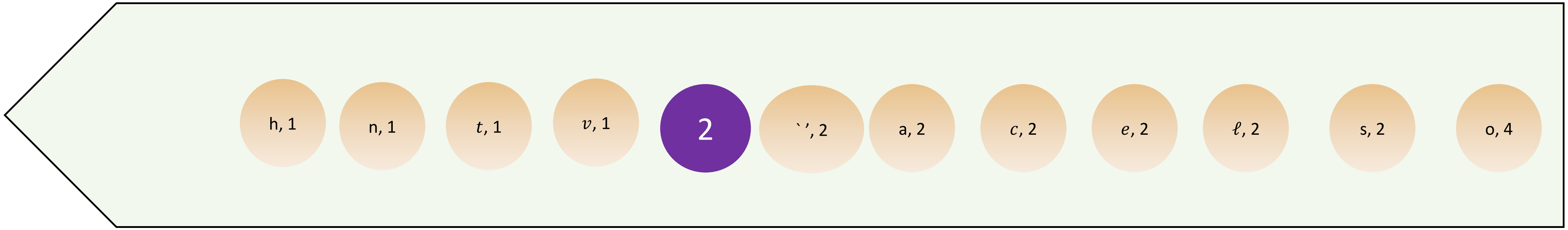
Building the binary tree….

3. As long as the queue has more than 1 elements in it

- Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
    - For a **MinHeap**-based PQ, this is **extremely fast:** $\mathcal{O}(2 \cdot \log_2|A|)$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535 \Rightarrow 2 \cdot \log_2|A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)
- Create a **new node** with those two nodes as children and a frequency value equal to the **sum of the children's frequency values.**
- **Insert that node into the trie!**



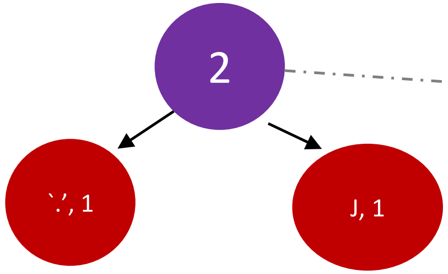
__Important implementation detail__; These nodes are assumed to hold the NULL character (first in ASCII table!)



Building the binary tree….

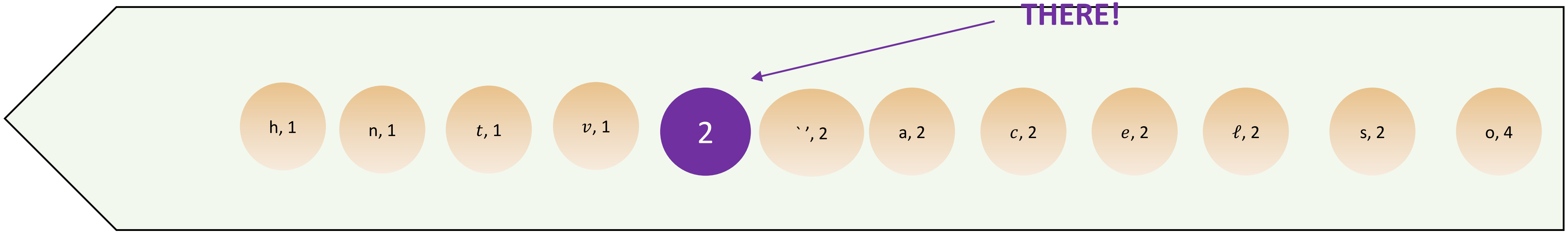
3. As long as the queue has more than 1 elements in it

- Apply `getMin()` **twice** to dequeue the 2 **least** elements in the queue.
  - For a **MinHeap**-based PQ, this is **extremely fast:** $\mathcal{O}(2 \cdot \log_2|A|)$ where A is your alphabet. Even for the subset of printable Unicode U with $|U| = 65535 \Rightarrow 2 \cdot \log_2|A| = 32$ compares in the **worst case** for a queue that will be pinned to cache because of temporal locality! Recall: heaps stored as arrays (consecutive cache storage)
- Create a **new node** with those two nodes as children and a frequency value equal to the **sum of the children's frequency values.**
- **Insert that node into the trie!**

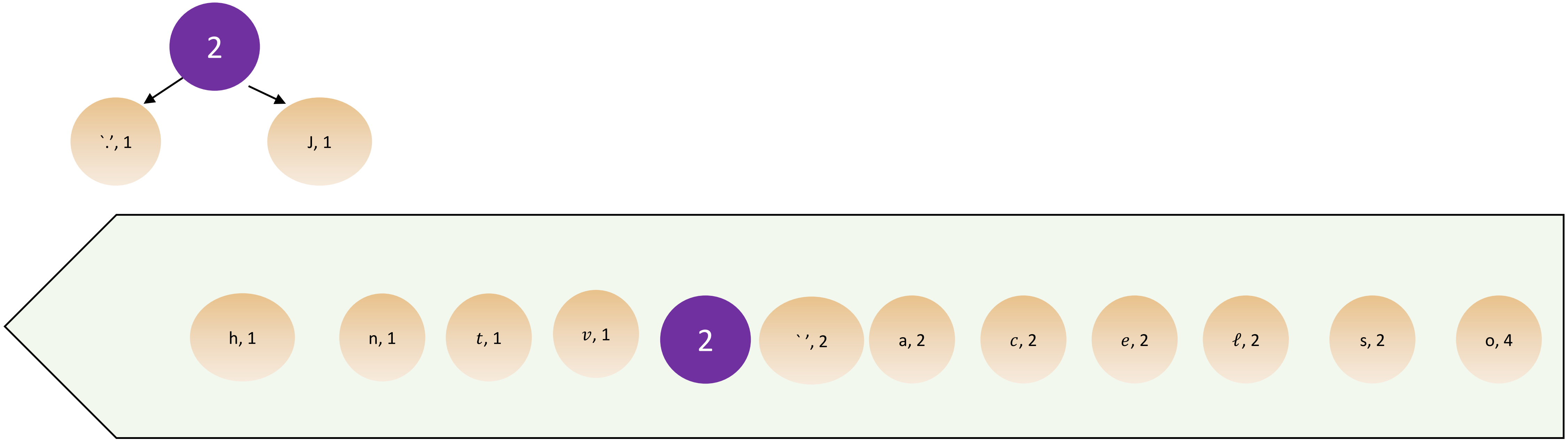


Important implementation detail; *These nodes are assumed to hold the NULL character (first in ASCII table!)*

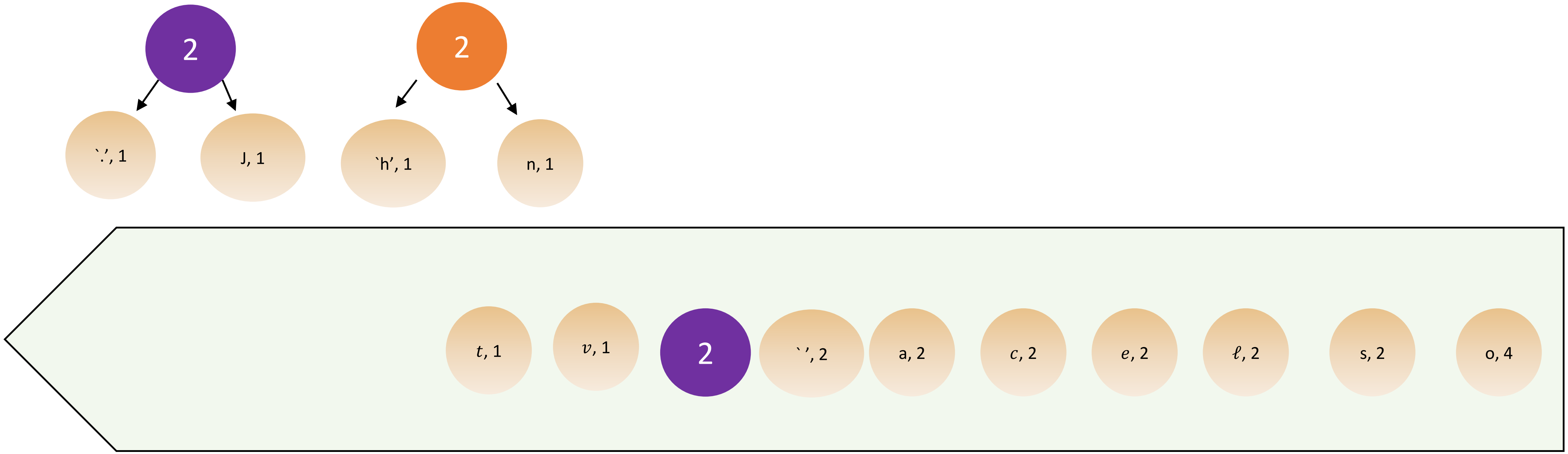
And that is why the node fits **EXACTLY THERE!**



Building the binary tree….

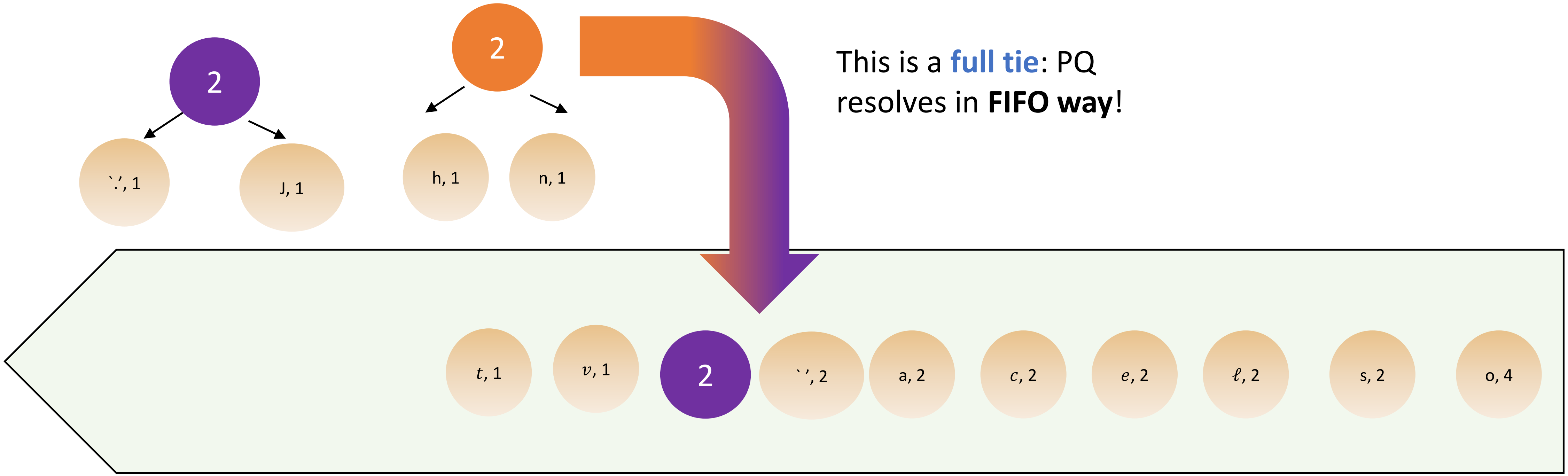


Building the binary tree….

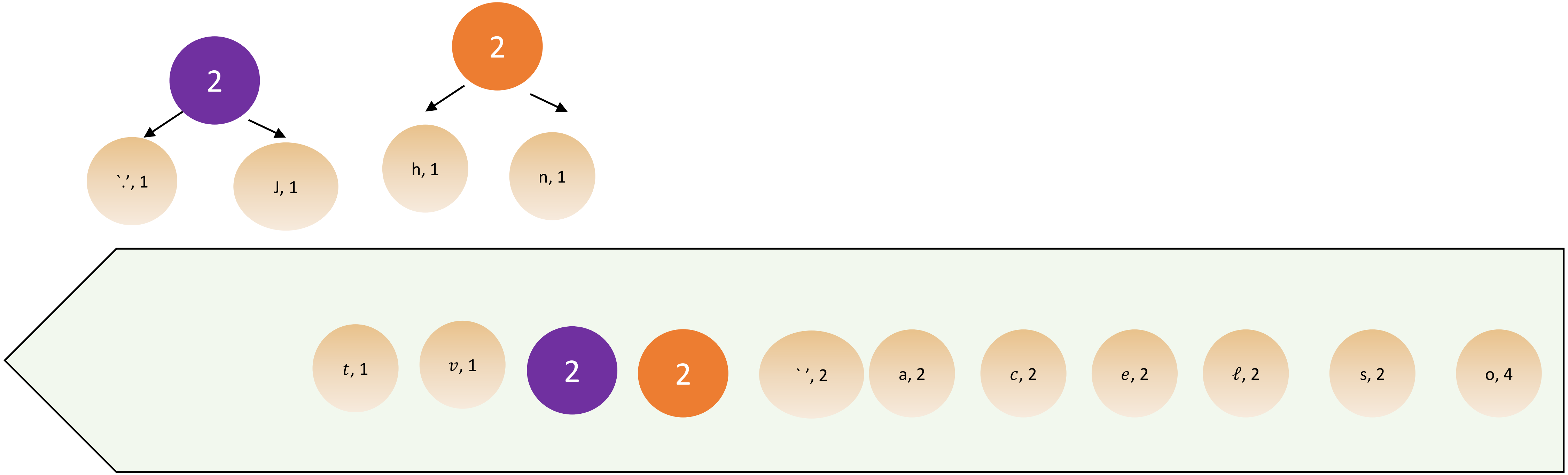


Building the binary tree….

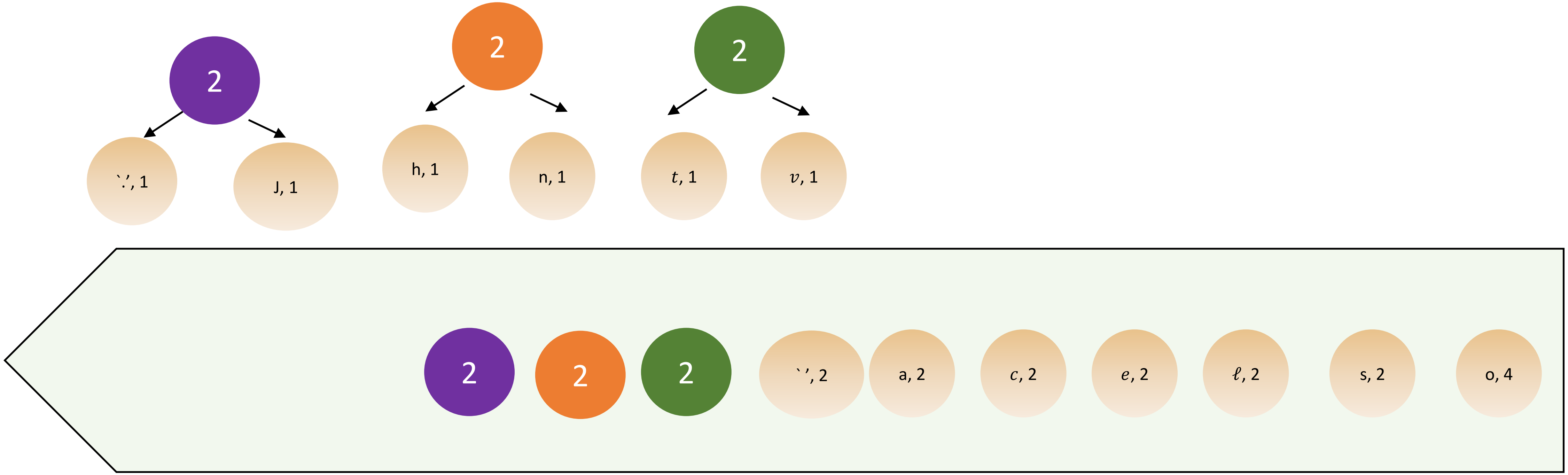
This is a **full tie**: PQ resolves in **FIFO way**!



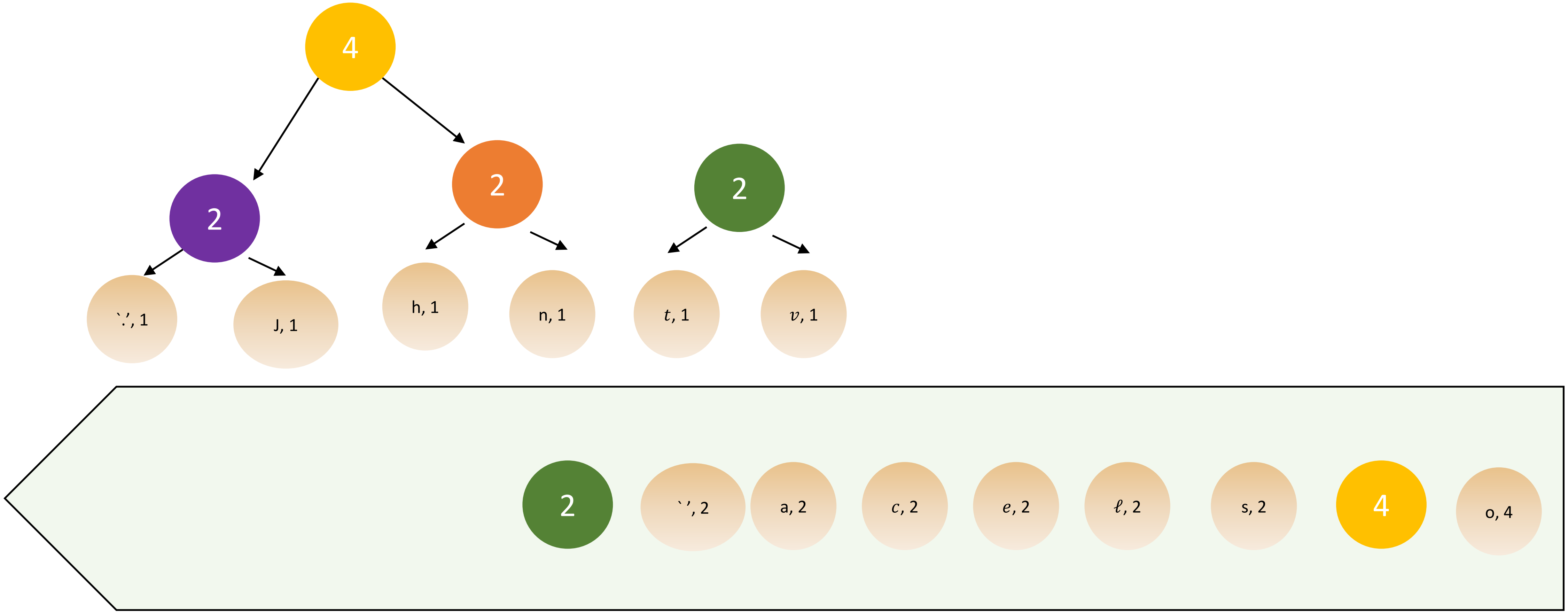
Building the binary tree….



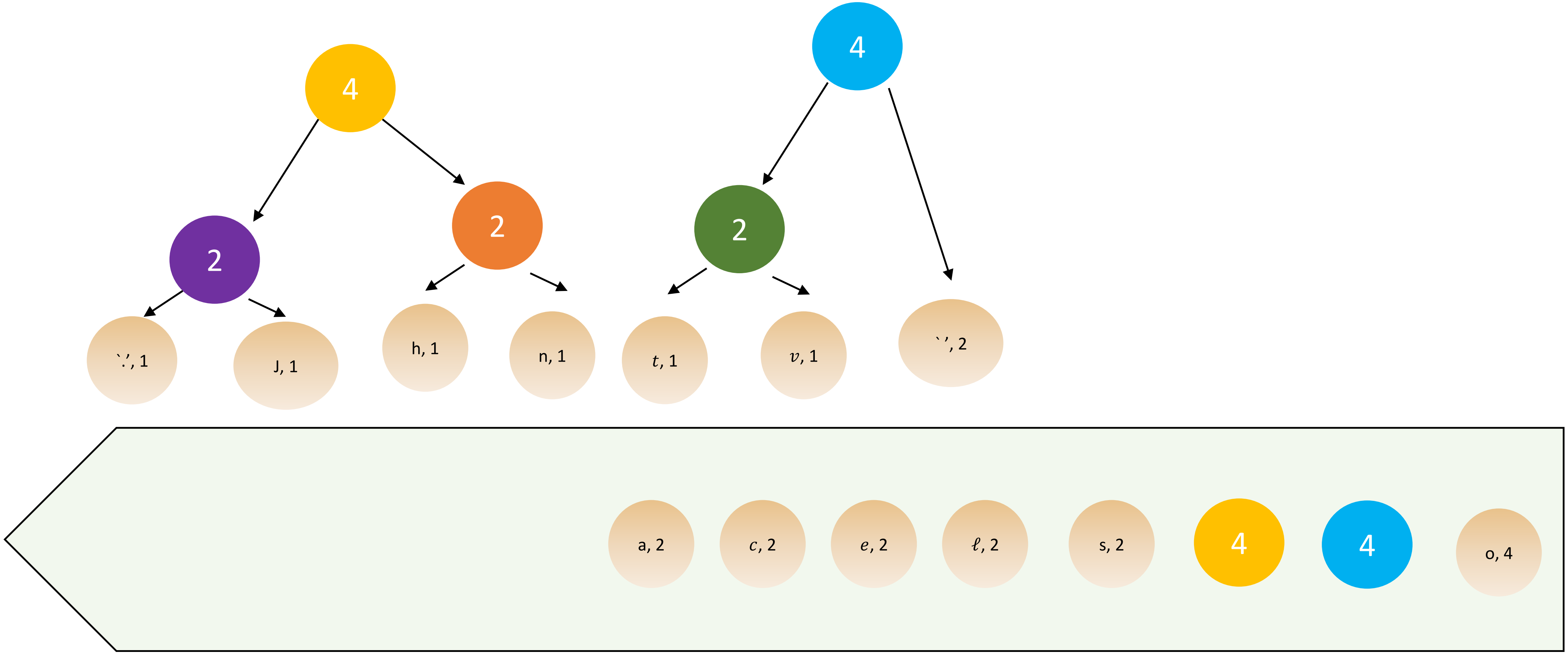
Building the binary tree….



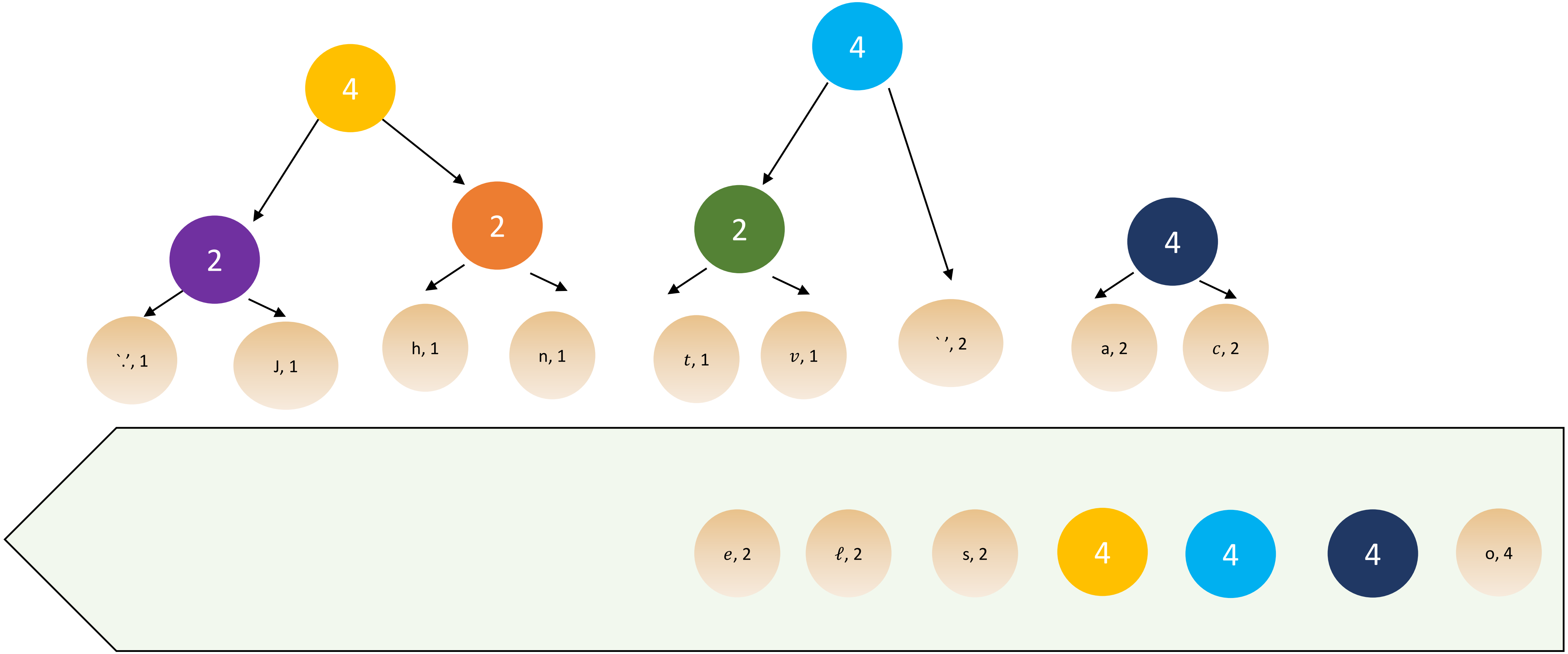
Building the binary tree….



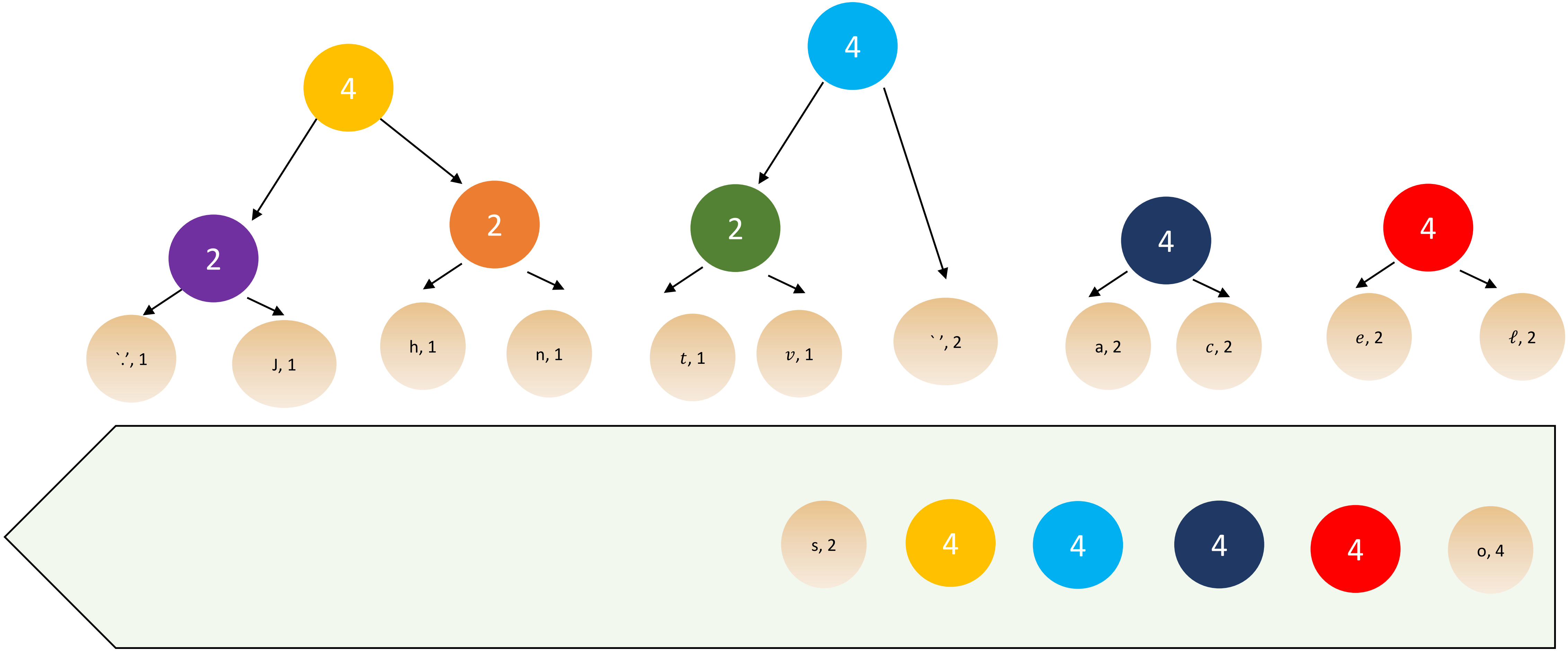
Building the binary tree....



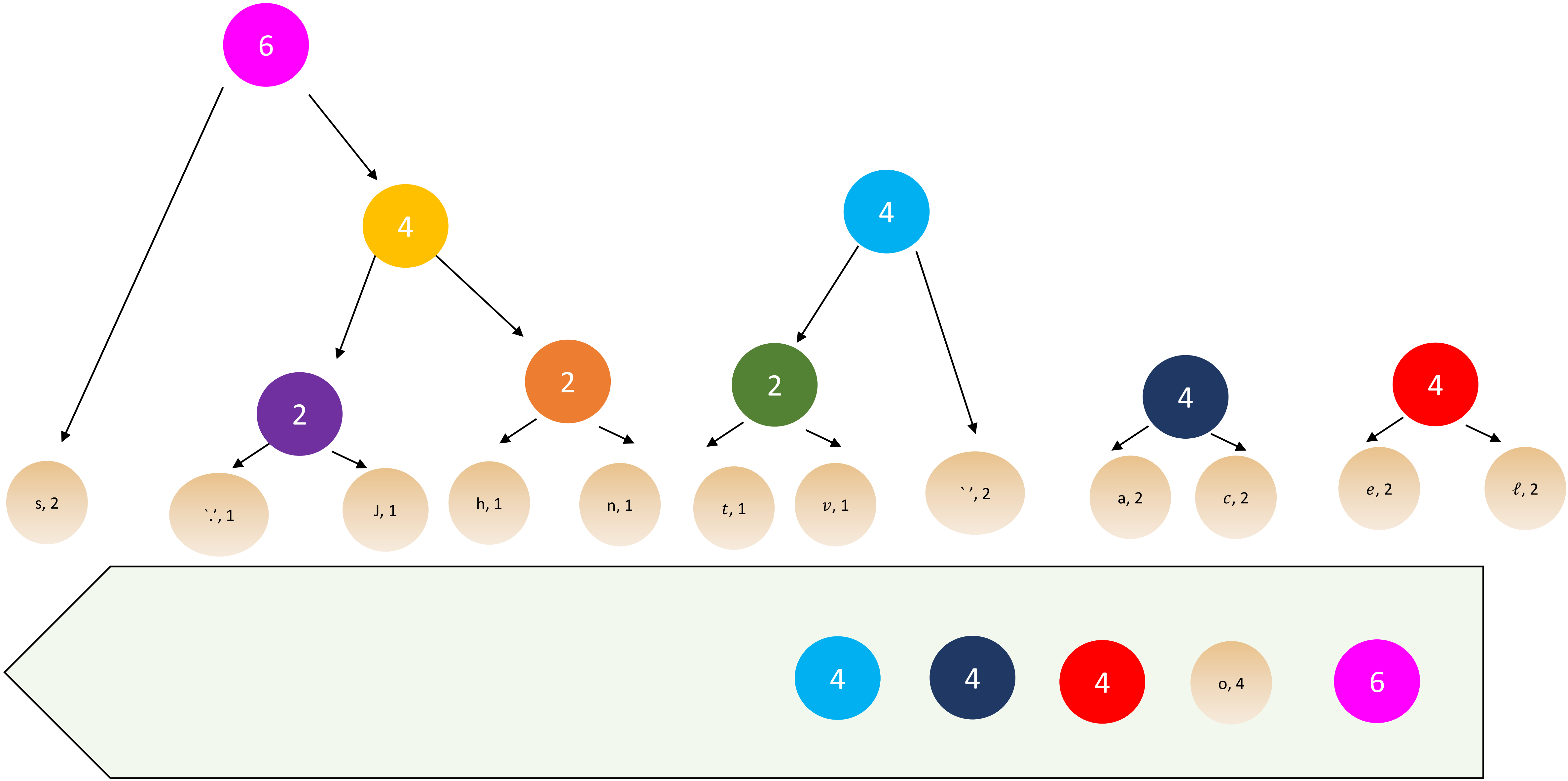
Building the binary tree….



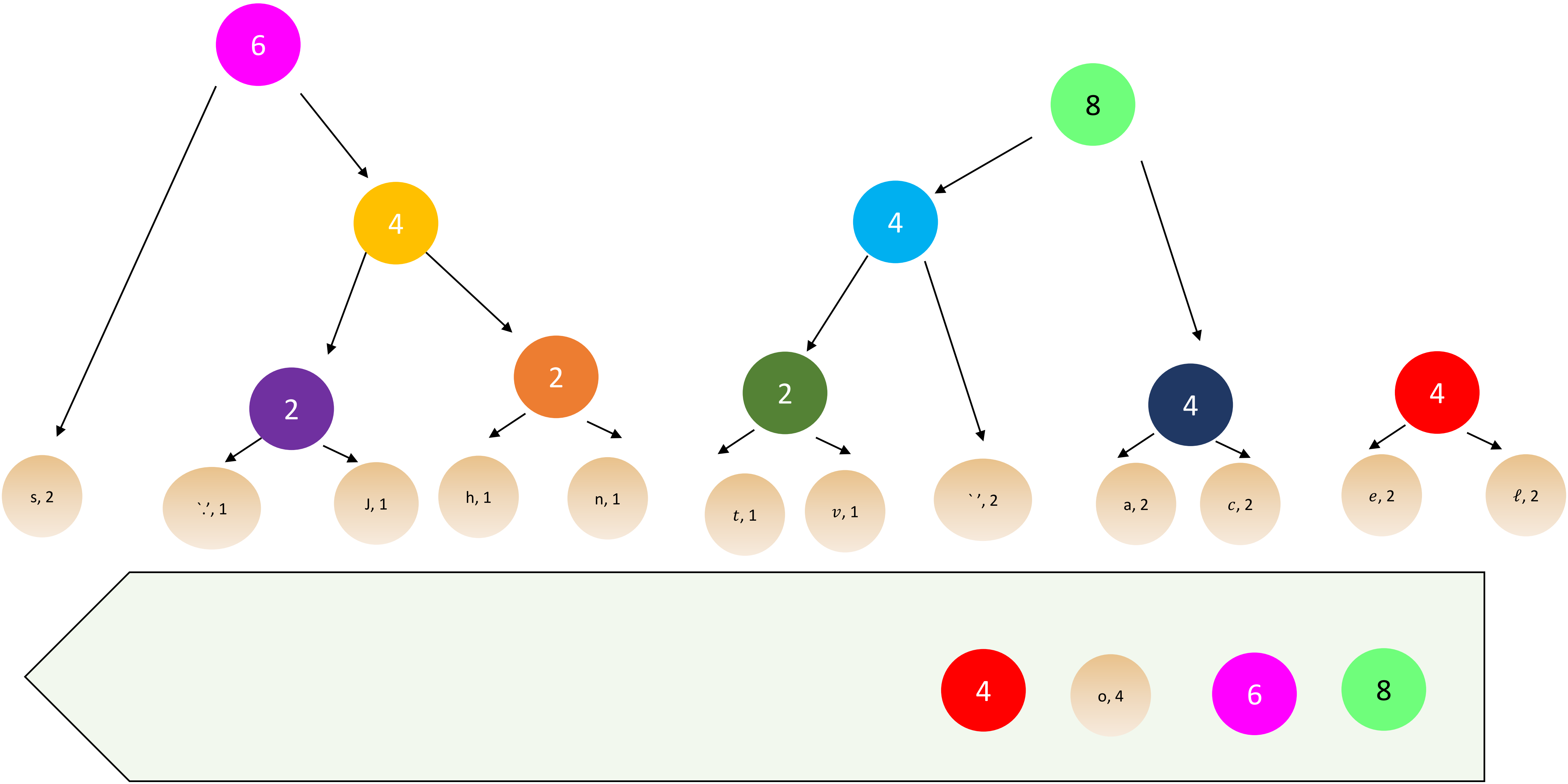
Building the binary tree….



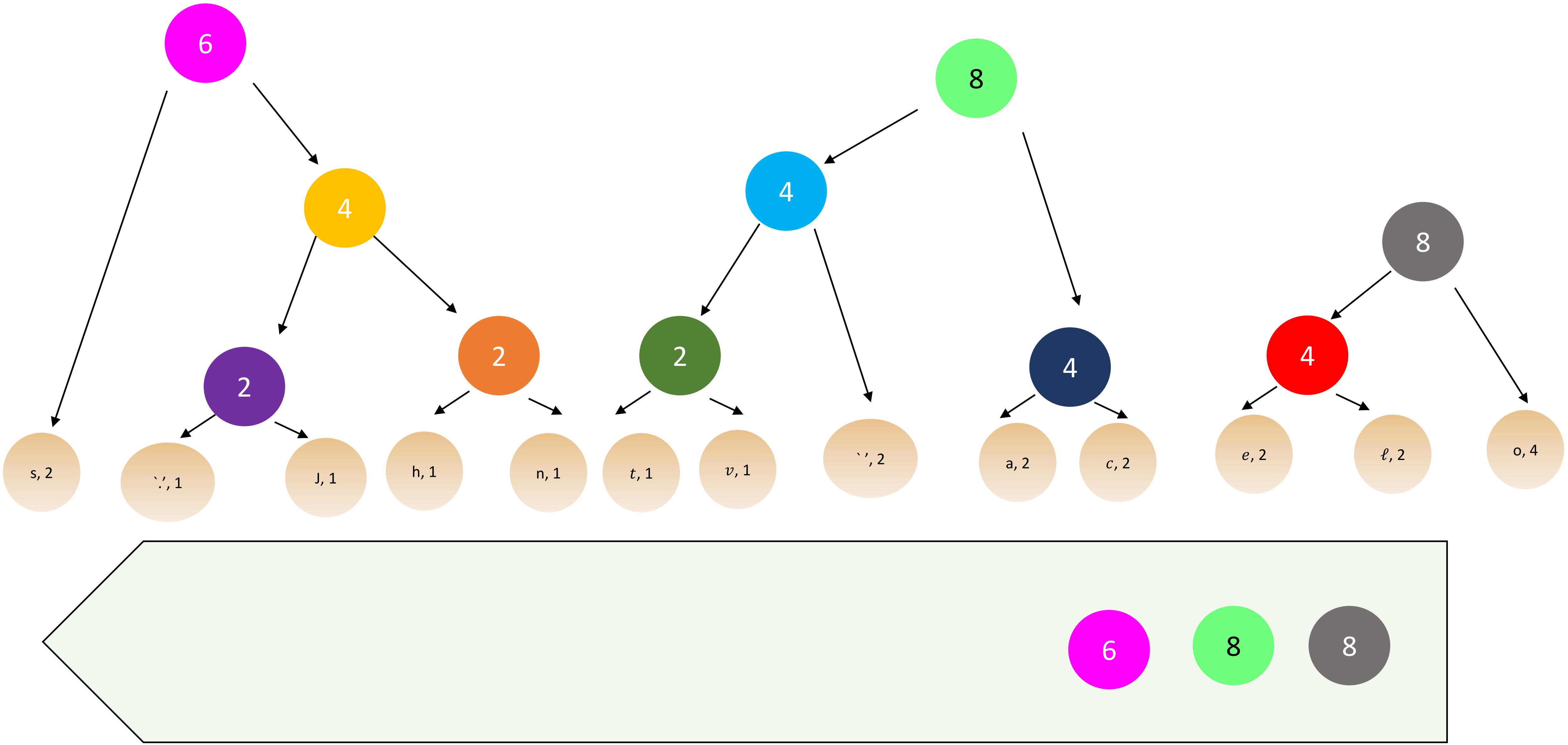
Building the binary tree….



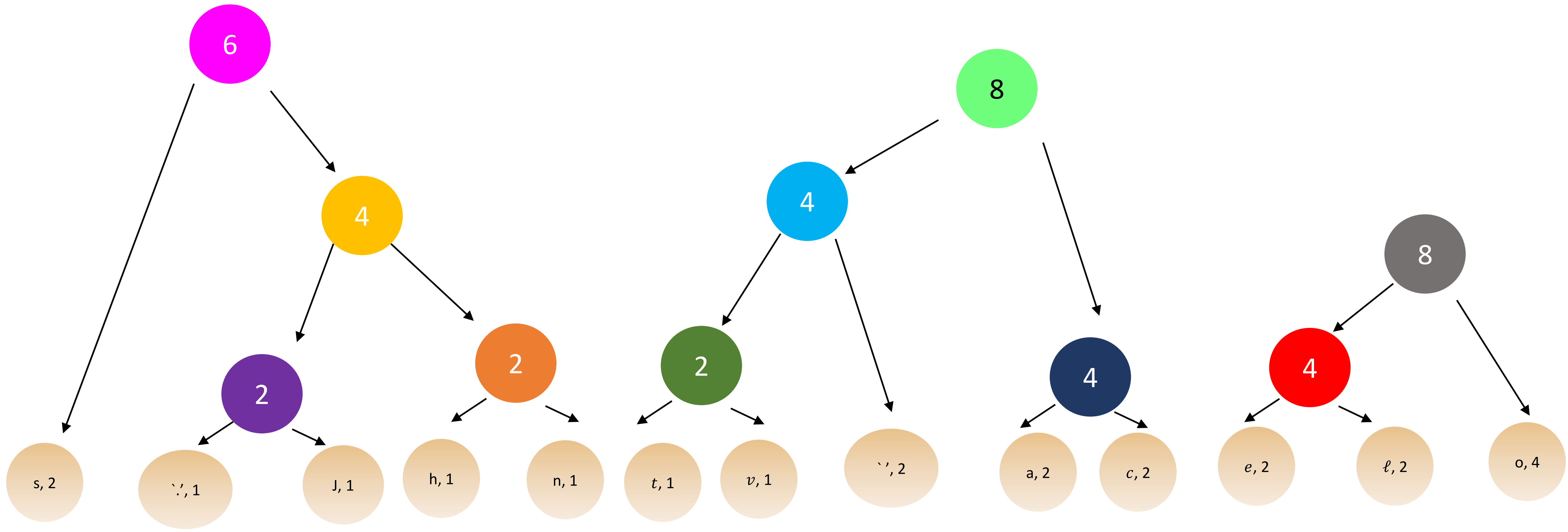
Building the binary tree….

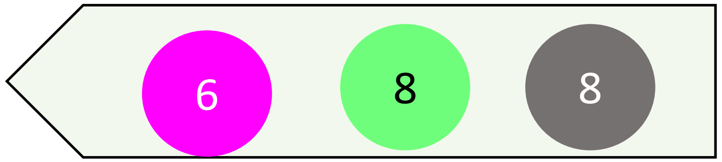


Building the binary tree….



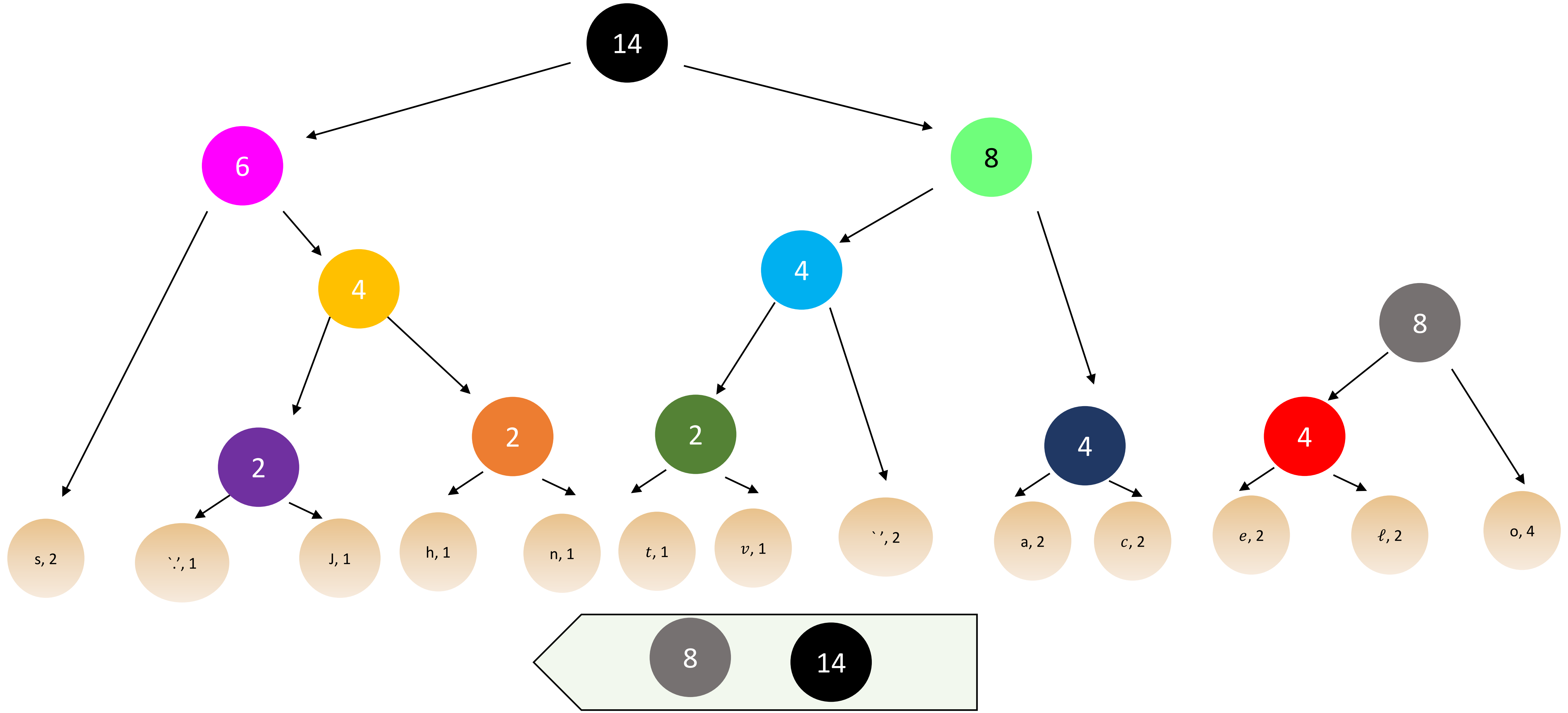
Building the binary tree….



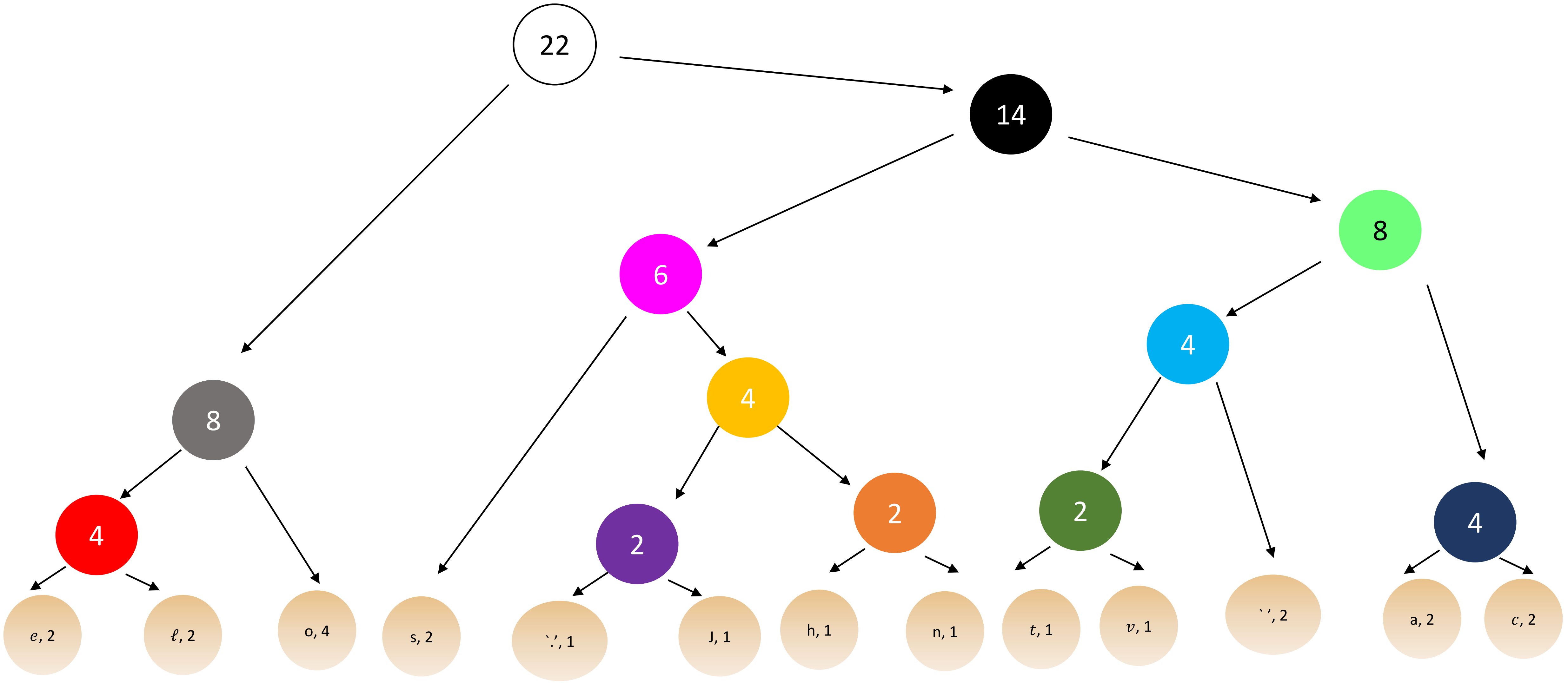


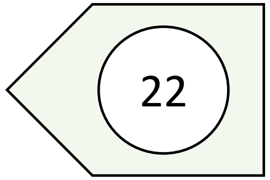
Just made PQ slimmer and moved everything down a bit…

Building the binary tree….



Building the binary tree....

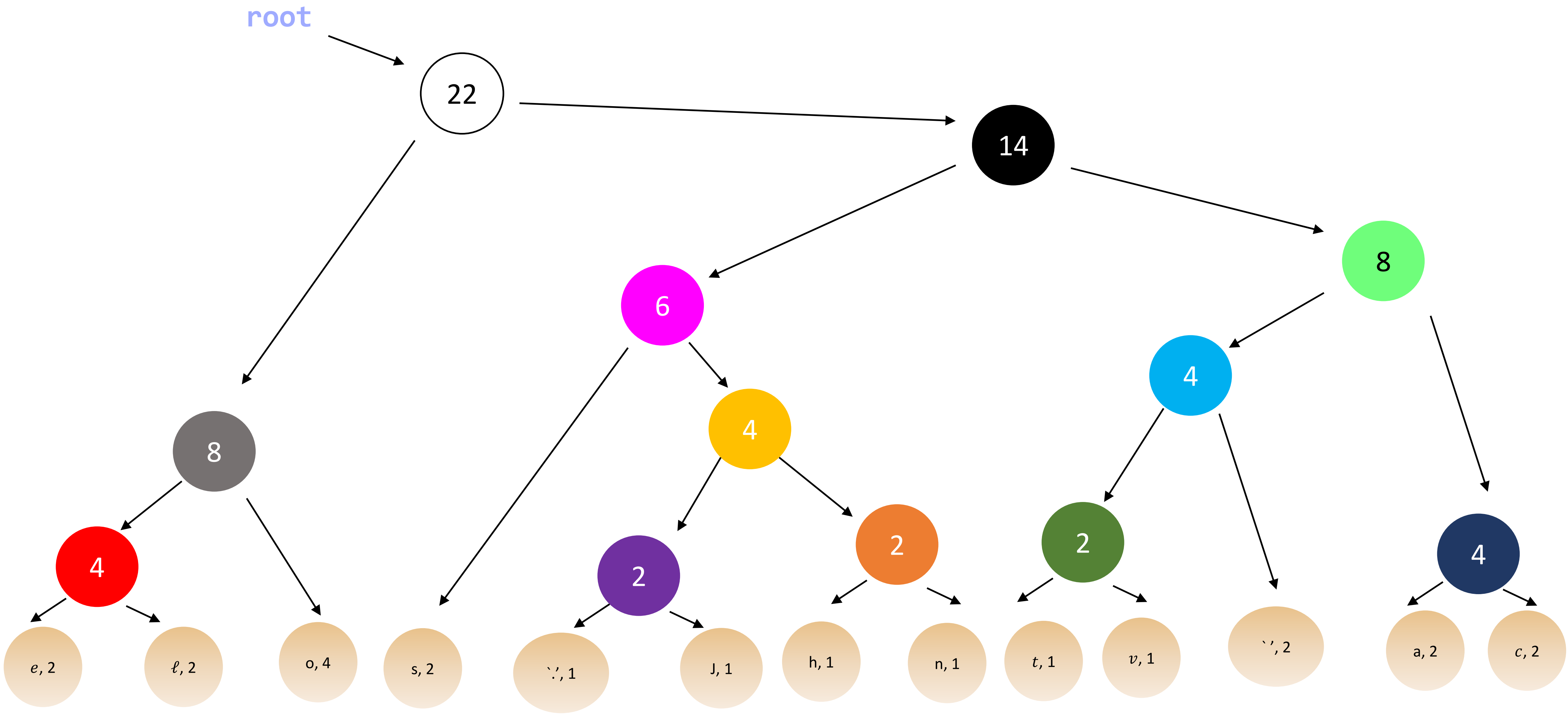




Queue now has single element; loop ends!

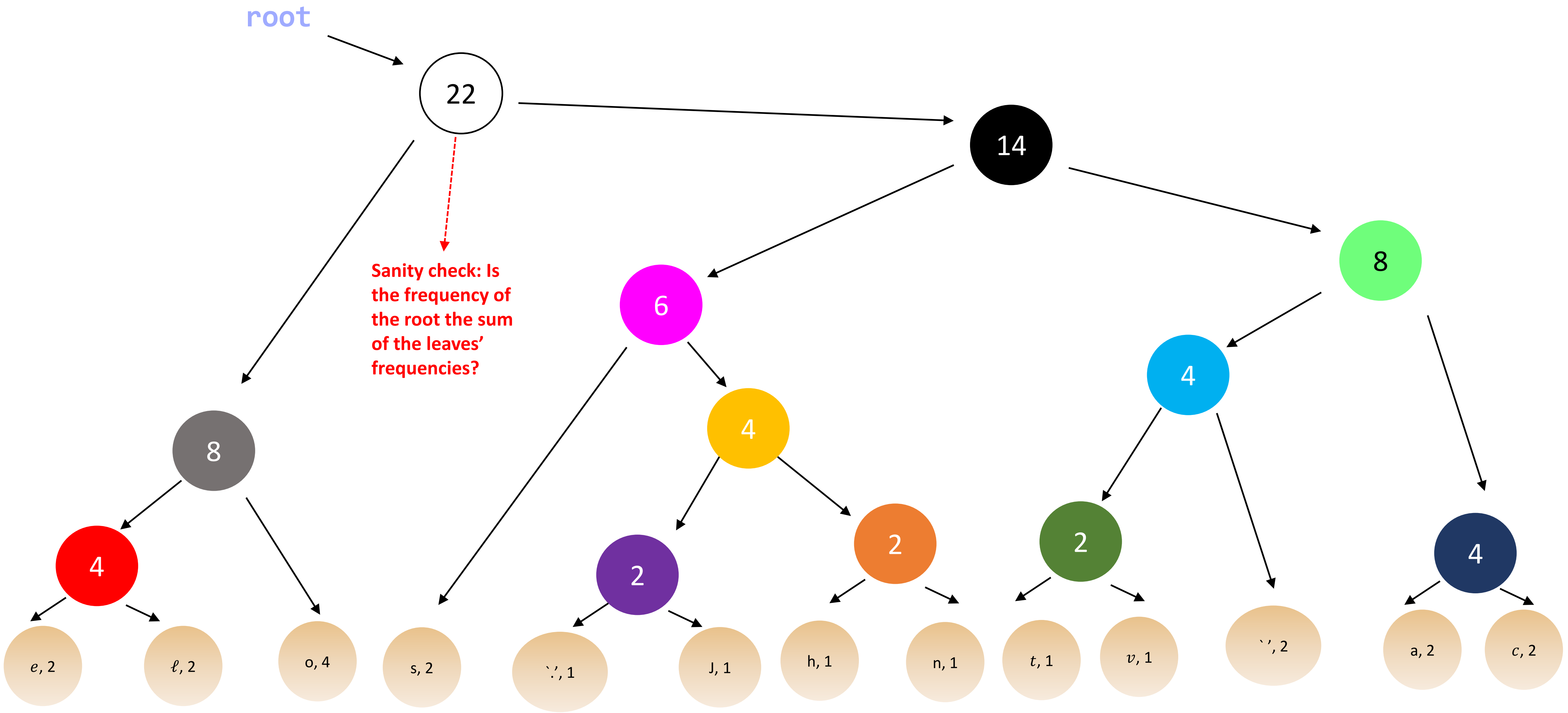
Building the binary tree….

4. Last node in PQ becomes **root of tree**

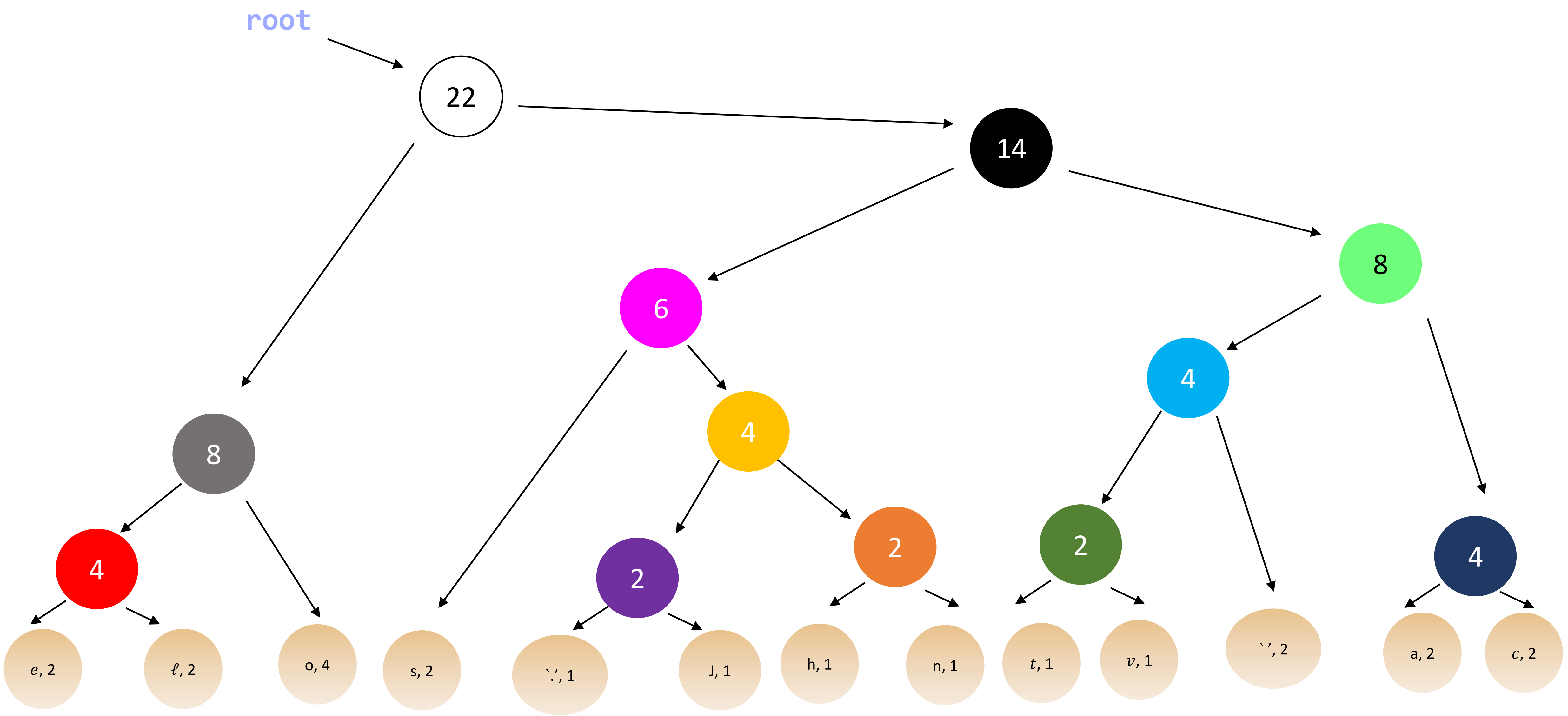


Building the binary tree….

4. Last node in PQ becomes **root of tree**

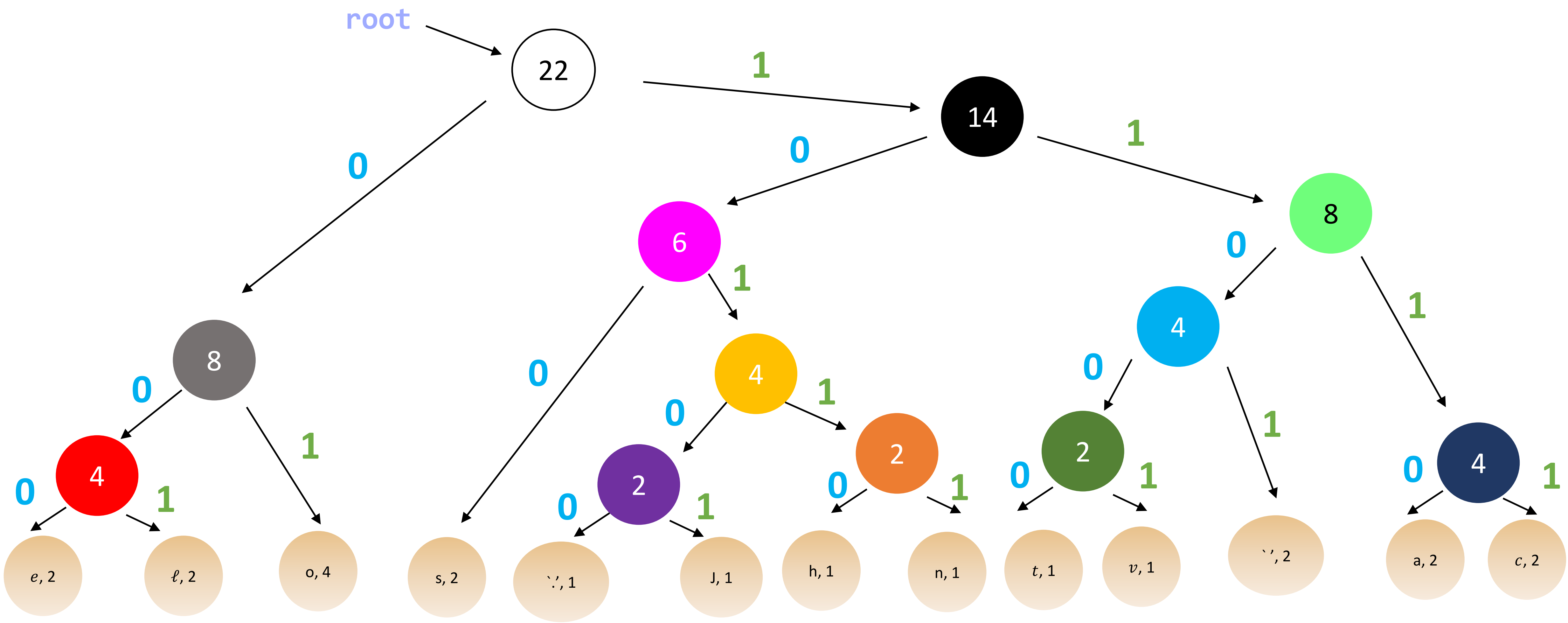


AND NOW FOR THE KILLER....



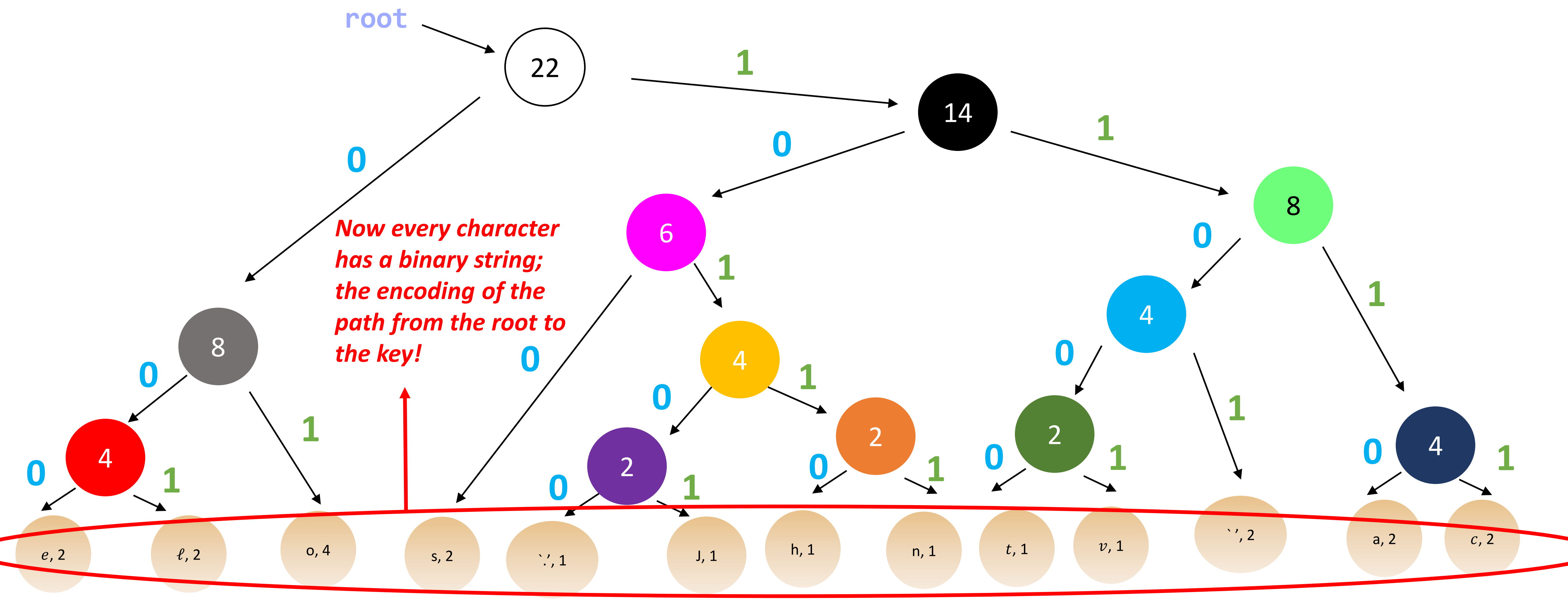
Building the binary trie!

5. Associate **left links with 0s** and **right links with 1s**, turning the binary tree into a **binary trie**!



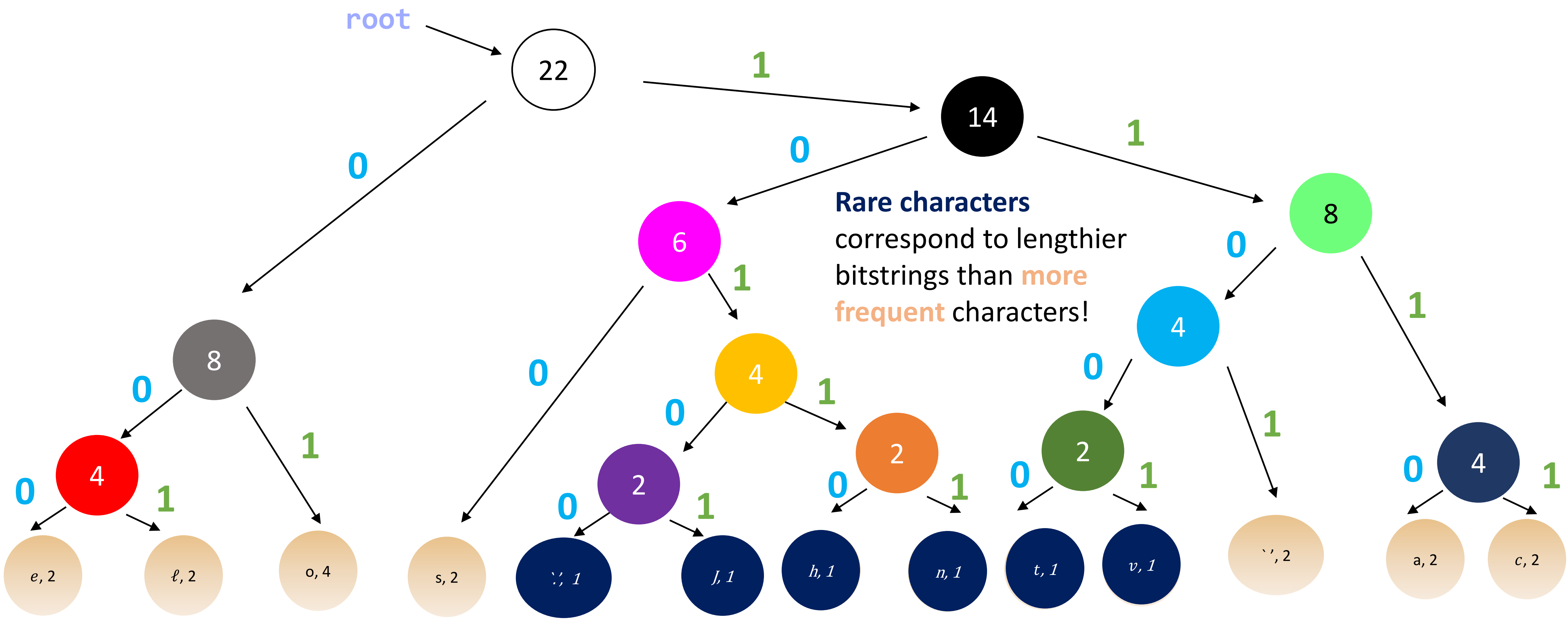
Building the binary trie!

5. Associate **left links with 0s** and **right links with 1s**, turning the binary tree into a **binary trie**!



Building the binary trie!

5. Associate **left links with 0s** and **right links with 1s**, turning the binary tree into a **binary trie**!



Rare characters correspond to lengthier bitstrings than more frequent characters!

Final structure: lookup table

6. Perform a preorder traversal of the trie to build a two-way lookup table that associates characters with Huffman binary encodings and vice versa
    - Likeliest inner implementation of this lookup table: two hash tables.
    - **In exams, you won't have to do this, since you can immediately "see" the encoding without the use of a lookup table.**

Final structure: lookup table

Character	Binary encoding
e	000
ℓ	001
o	01
s	100
.	10100
J	10101
h	10110
n	10111
t	11000
v	11001
SPACE	1101
a	1110
c	1111

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Note that the most expensive Huffman encodings in this table are still under 7 bits!

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Let's compare how the message

"Jason loves chocolate"

decodes through both ASCII and Huffman:

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Let's compare how the message

"Jason loves chocolate"

decodes through both ASCII and Huffman:

- **ASCII**: 10010101100001111001111011111101110001000011011001101111111011011001011110011001000011000111101000 1101111 1100011 1101111 11011001101111 1100001 1110100 1100101, 147 bits total.

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Let's compare how the message

  "Jason loves chocolate"

decodes through both ASCII and Huffman:

- **ASCII**: 10010101100001111001111011111101110001000011011001101111111011011001011110011100100001100011 1101000 1101111 1100011 1101111 11011001101111 1100001 1110100 1100101, 147 bits total.

- **Huffman:** 101011110100011011111101001011100100010011011111101100111110100111011000000, 75 bits total

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Let's compare how the message *"Jason loves chocolate"* decodes through both ASCII and Huffman:

- **ASCII**: 1001010110000111100111101111 1101110001000011011001101111111011011001011 1100110010000 1100011 1101000 1101111 1100011 1101111 11011001101111 1100001 1110100 1100101, 147 bits total.

- **Huffman:** 101011110100011011111101001011100100010011101 111110110011111010011110110000000, 75 bits total $\approx$ **50% of ASCII !**

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Can I encode other strings using this table (remember, we created it using the string “Jason loves chocolate”)?

Yes

No

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

- Can I encode other strings using this table (remember, we created it using the string "Jason loves chocolate")?

Yes

No

- As long as the table covers the subset of the alphabet they need!
- As an example, let's encode the string "Jonas loves nachos" in both ASCII and Huffman, using this table ☺

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

Don’t worry, Jason has made sure that the alphabet is the same! :)

“Jonas loves nachos”

- ASCII:1001010110111111011101100001111001100100001101100110111111011011001011110011001000011011101100001110001111010001101111110011 = 126 bits

Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

Alphabet is the same!
Σ ={a, c,e,h, j, l, n, o, s, v}

"Jonas loves nachos"

- ASCII:1001010110111111011101100001111001100100001101100110111111101101100101111001100100001101110110000111000111101000110111111110011 = 126 bits
- Huffman:1010111101000110111110100101110010001001101101111111011111011001100 = 66 bits≈ 52.39% of ASCII

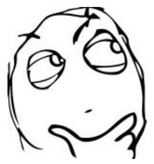
Encoding our message in two ways

Character	Huffman Binary encoding	ASCII (7-bit) encoding
e	000	1100101
ℓ	001	1101100
o	01	1101111
s	100	1110011
.	10100	101110
J	10101	1001010
h	10110	1101000
n	10111	1101110
t	11000	1110100
v	11001	1110110
SPACE	1101	0010000
a	1110	1100001
c	1111	1100101

Alphabet is the same!
Σ ={a, c,e,h, j, l, n, o, s, v}

"Jonas loves nachos"

- ASCII:1001010110111111011101100001111001100100000110110011011111110110010111100110010000110111011000011100011110100011011111110011 = 126 bits
- Huffman:101010111101000110111110100101110010001001101101111111010111011001100 = 66 bits ≈ 52.39% of ASCII
  - Slightly worse than before!
    - Why?.



Your turn!

Jason: Give students an ASCII table on a different projector….

- Build the Huffman trie for the string

“Data structures”

Solution

“Data structures”

- SPC: 1
- ‘a’: 2
- ‘c’: 1
- ‘D’: 1
- ‘e’: 1
- ‘r’: 2
- ‘s’: 2
- ‘t’: 3
- ‘u’: 2

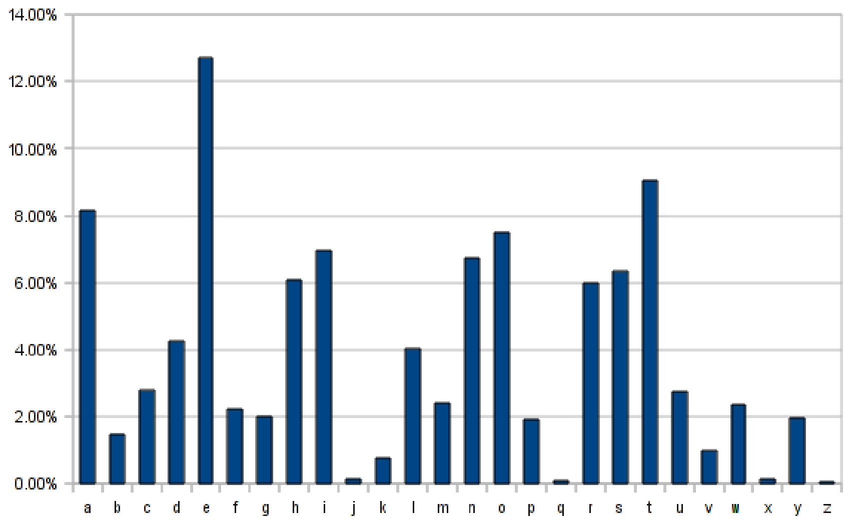
Solution

"Data structures"

- SPC: 1
- 'a': 2
- 'c': 1
- 'd': 1
- 'e': 1
- 'r': 2
- 's': 2
- 't': 3
- 'u': 2

In this text, there are more 't's and 'u's than 'e's. This is unrealistic in the full spectrum of English.

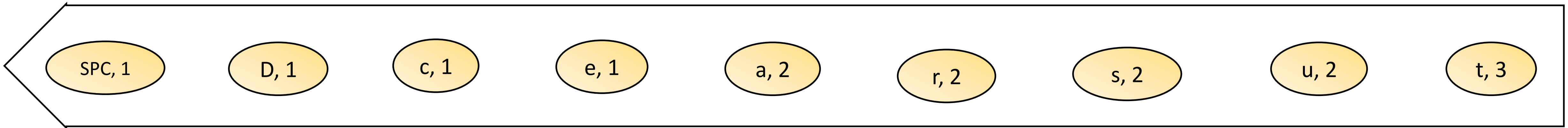
As our text grows, your histogram will look more and more like the one we've seen:



Solution

“Data structures”

- SPC: 1
- ‘a’: 2
- ‘c’: 1
- ‘d’: 1
- ‘e’: 1
- ‘r’: 2
- ‘s’: 2
- ‘t’: 3
- ‘u’: 2

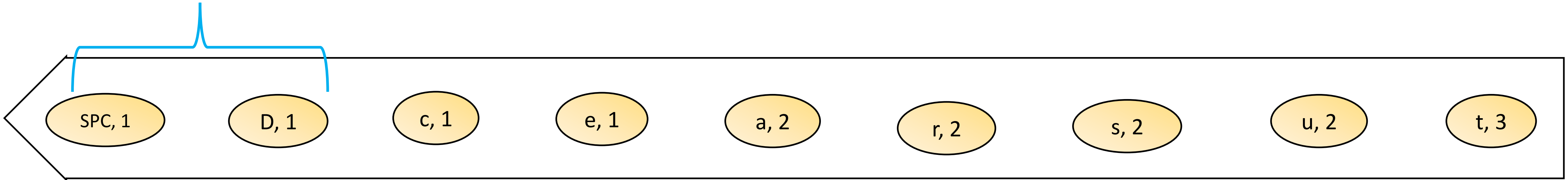


Solution

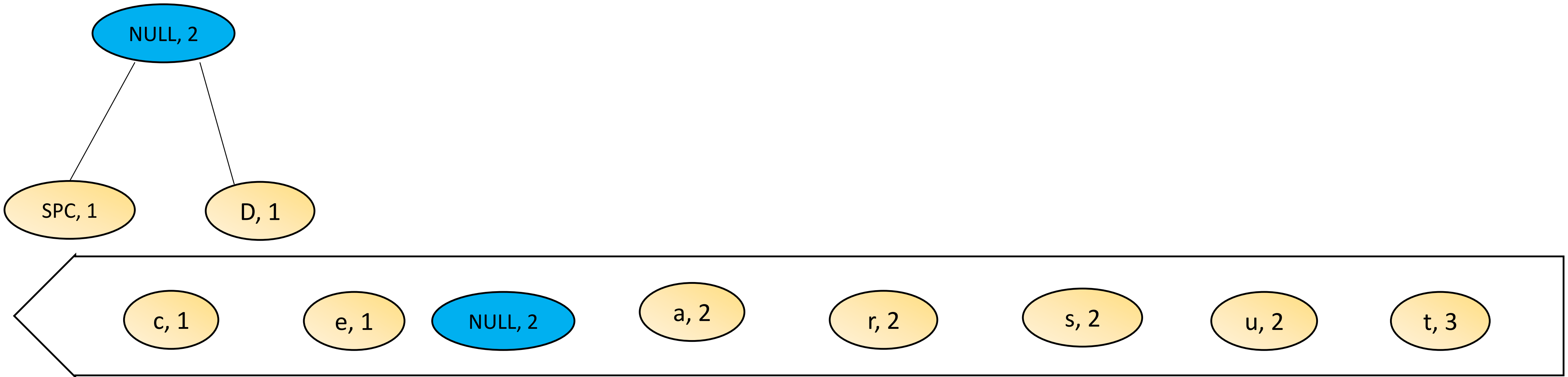
“Data structures”

- SPC: 1
- ‘a’: 2
- ‘c’: 1
- ‘D’: 1
- ‘e’: 1
- ‘r’: 2
- ‘s’: 2
- ‘t’: 3
- ‘u’: 2

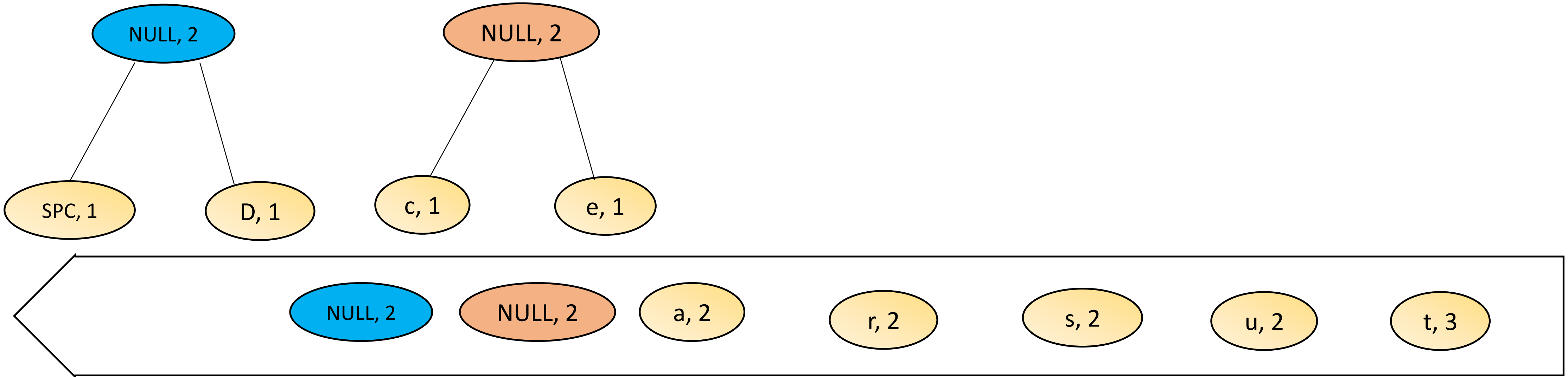
Note: In ASCII, Space precedes caps, which precede lowercase!



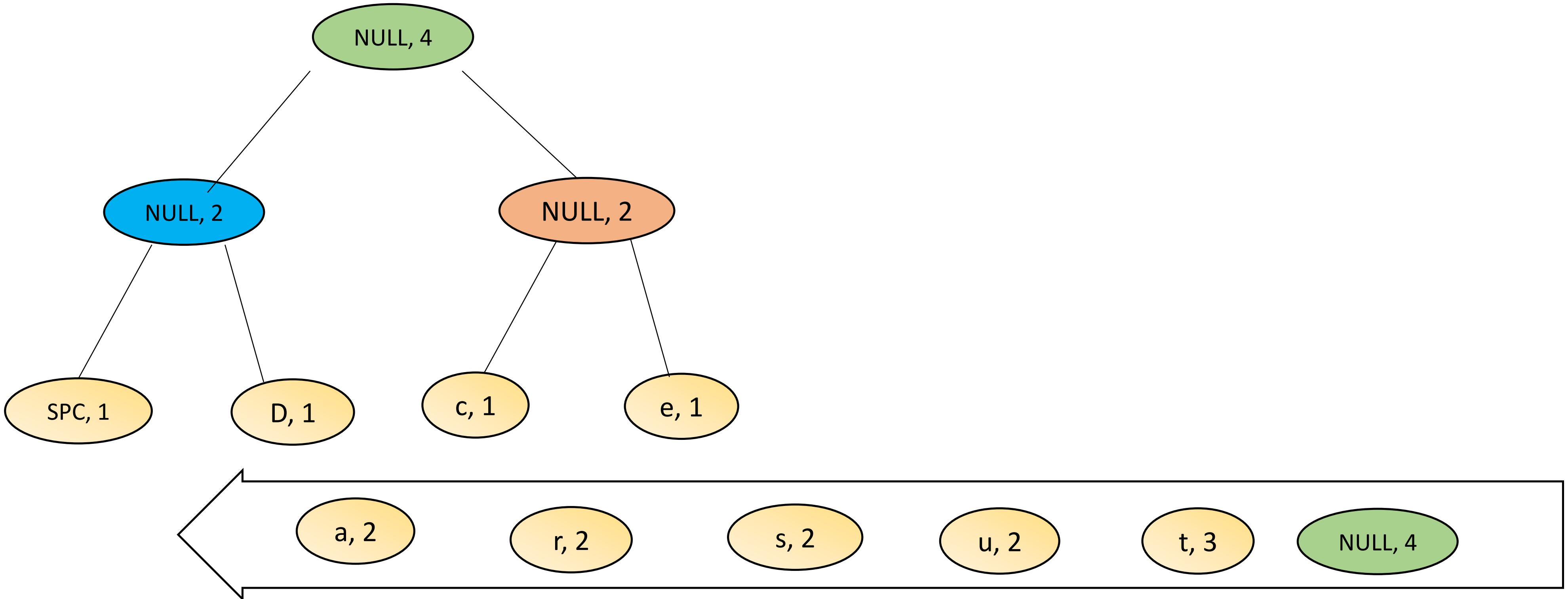
Solution



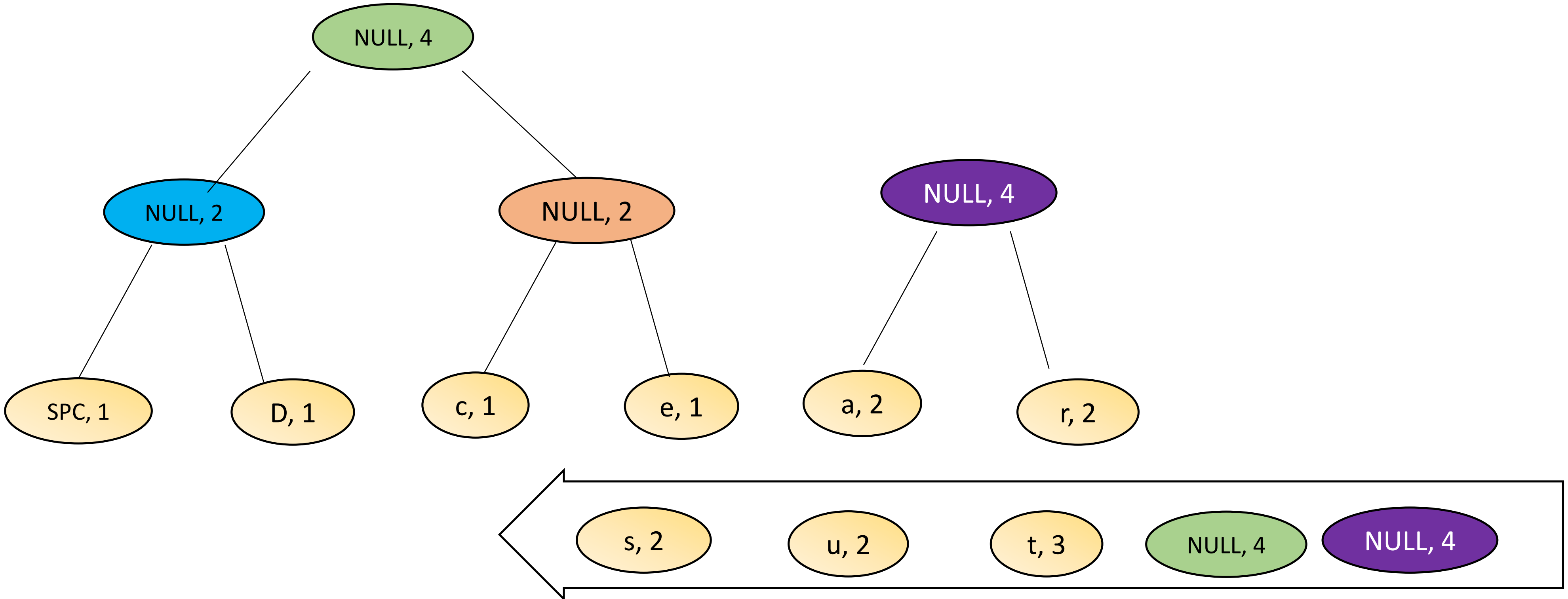
Solution



Solution

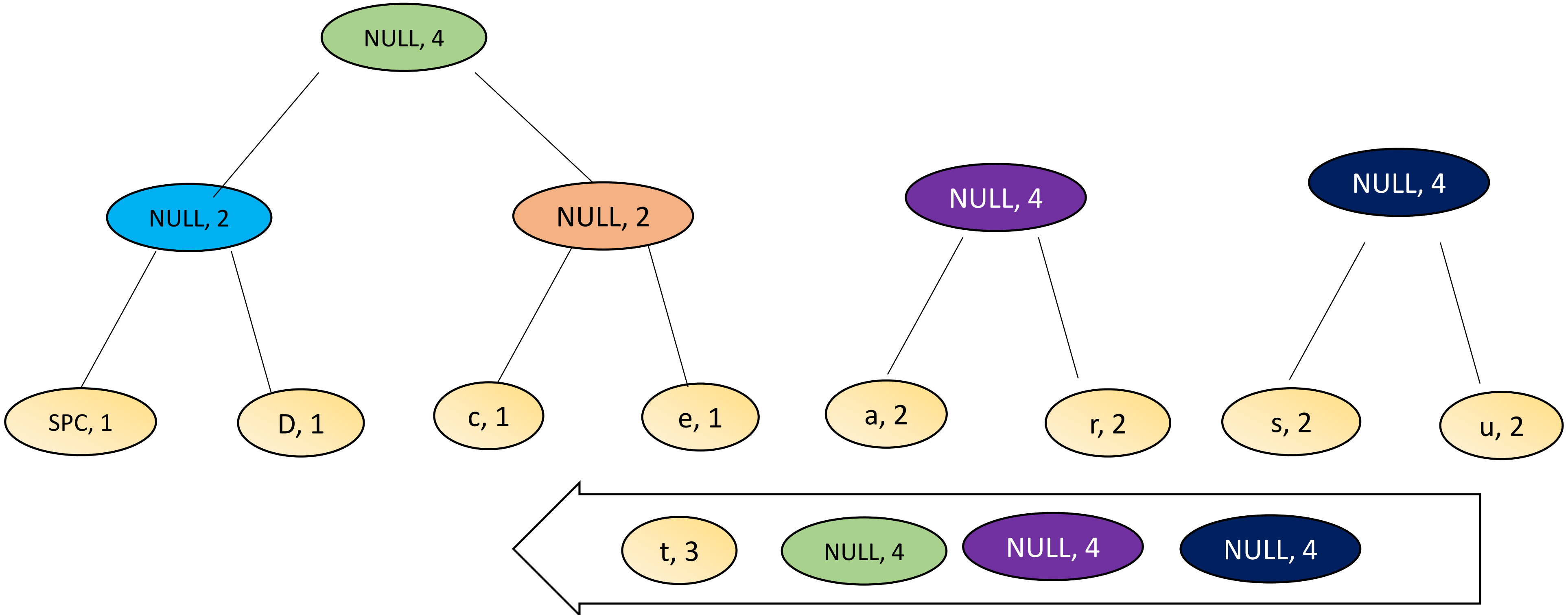


Solution

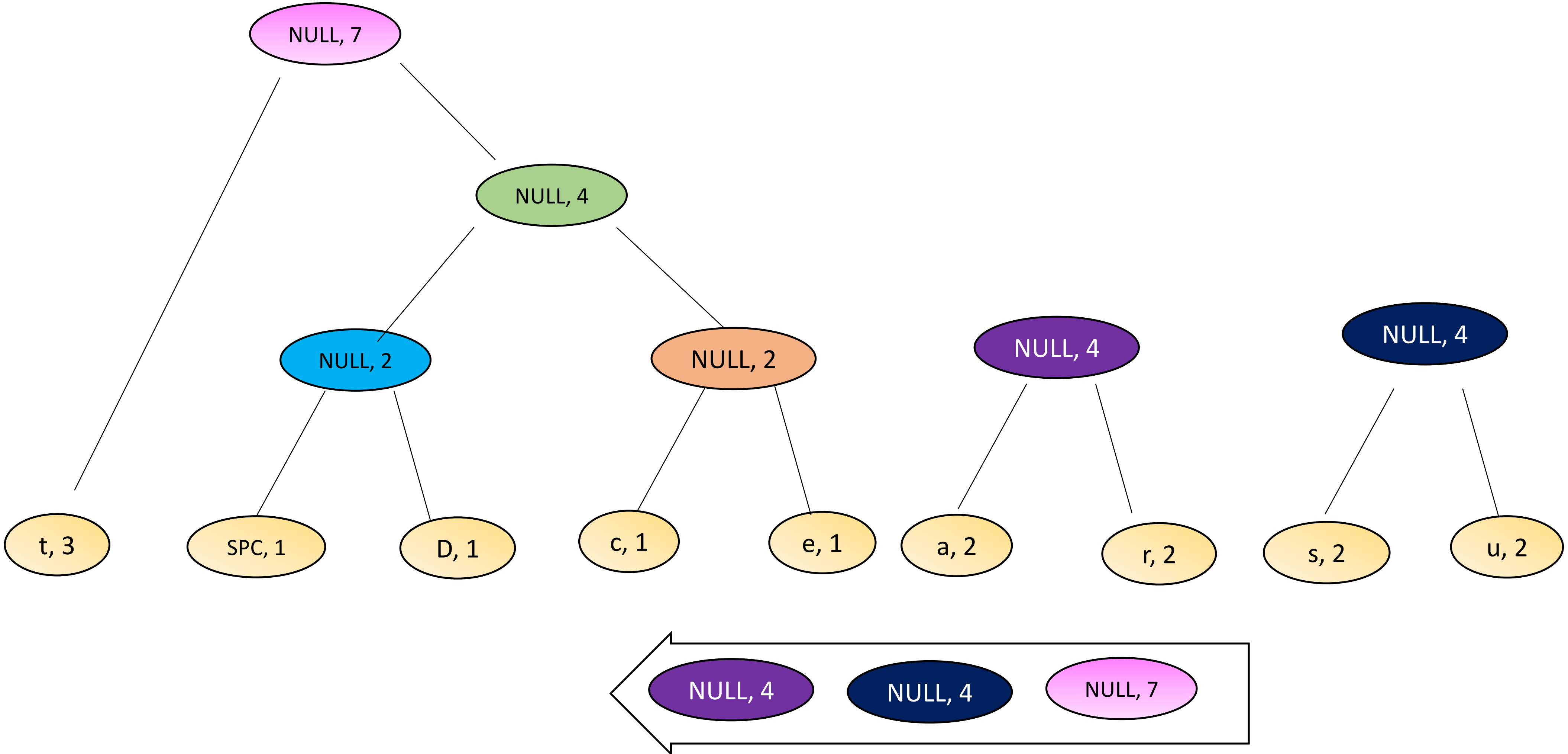


NULL, 4
NULL, 2
NULL, 2
NULL, 4
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2
t, 3
NULL, 4
NULL, 4

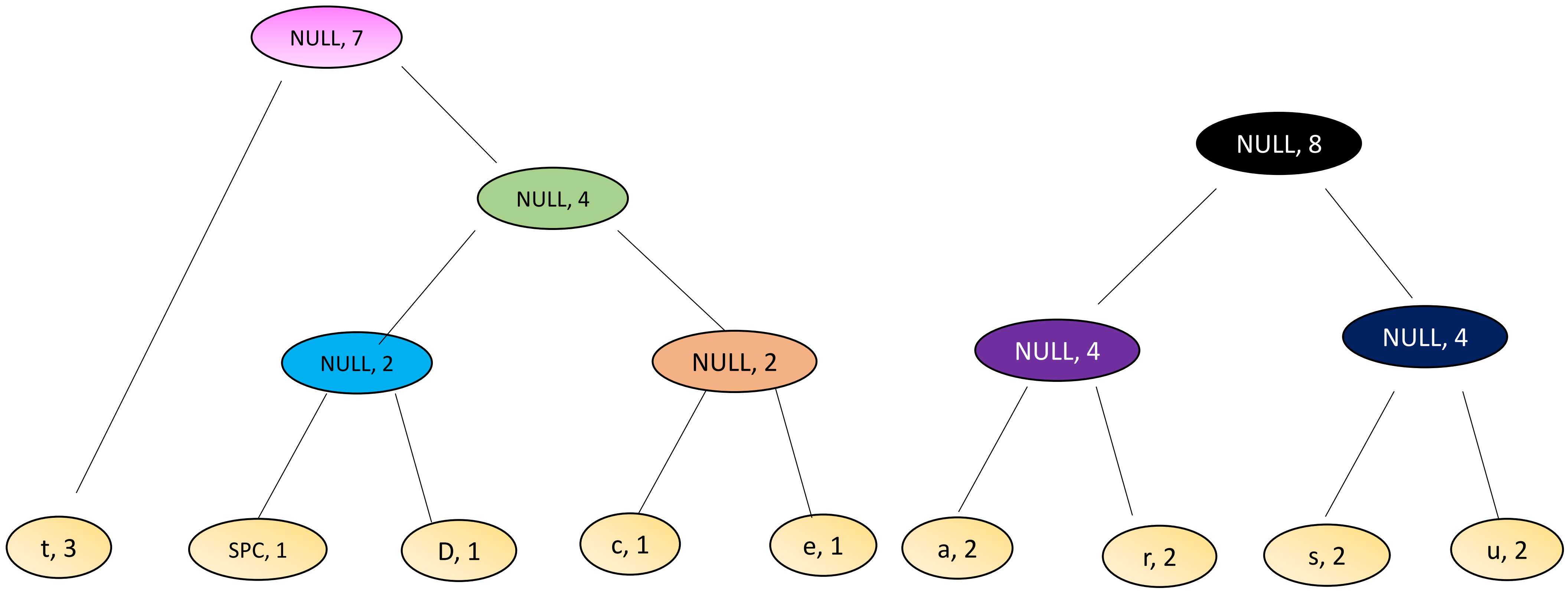
Solution

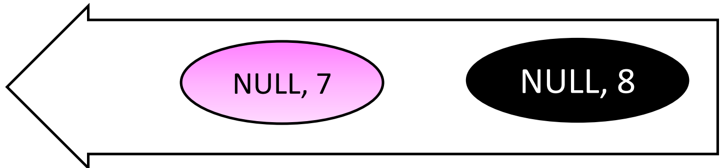

NULL, 4
NULL, 2
NULL, 2
NULL, 4
NULL, 4
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2
t, 3
NULL, 4
NULL, 4
NULL, 4

Solution

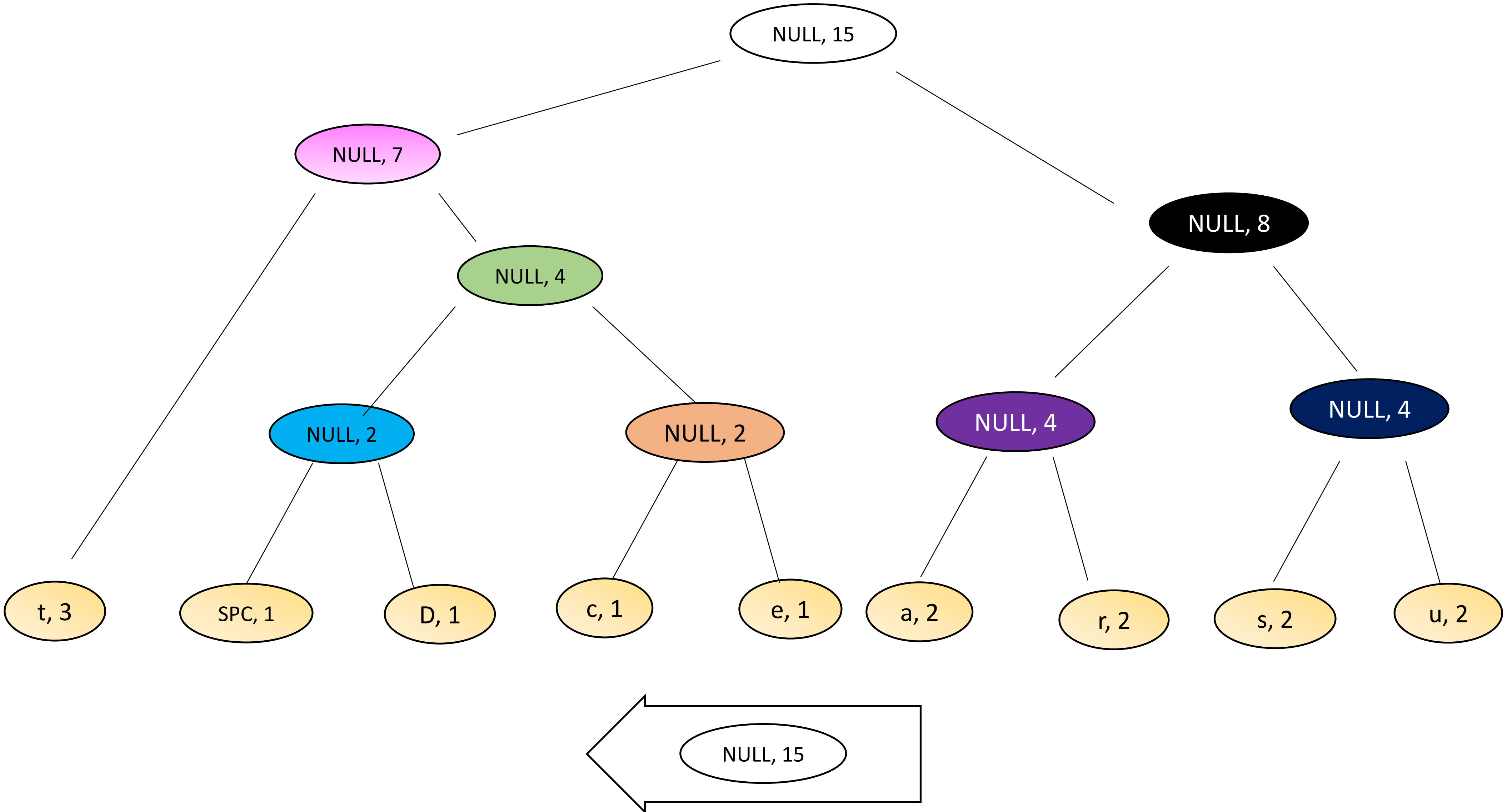

NULL, 7
NULL, 4
NULL, 2
NULL, 2
NULL, 4
NULL, 4
t, 3
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2
NULL, 4
NULL, 4
NULL, 7

Solution


NULL, 7
NULL, 4
NULL, 2
NULL, 2
t, 3
SPC, 1
D, 1
c, 1
e, 1
NULL, 8
NULL, 4
NULL, 4
a, 2
r, 2
s, 2
u, 2

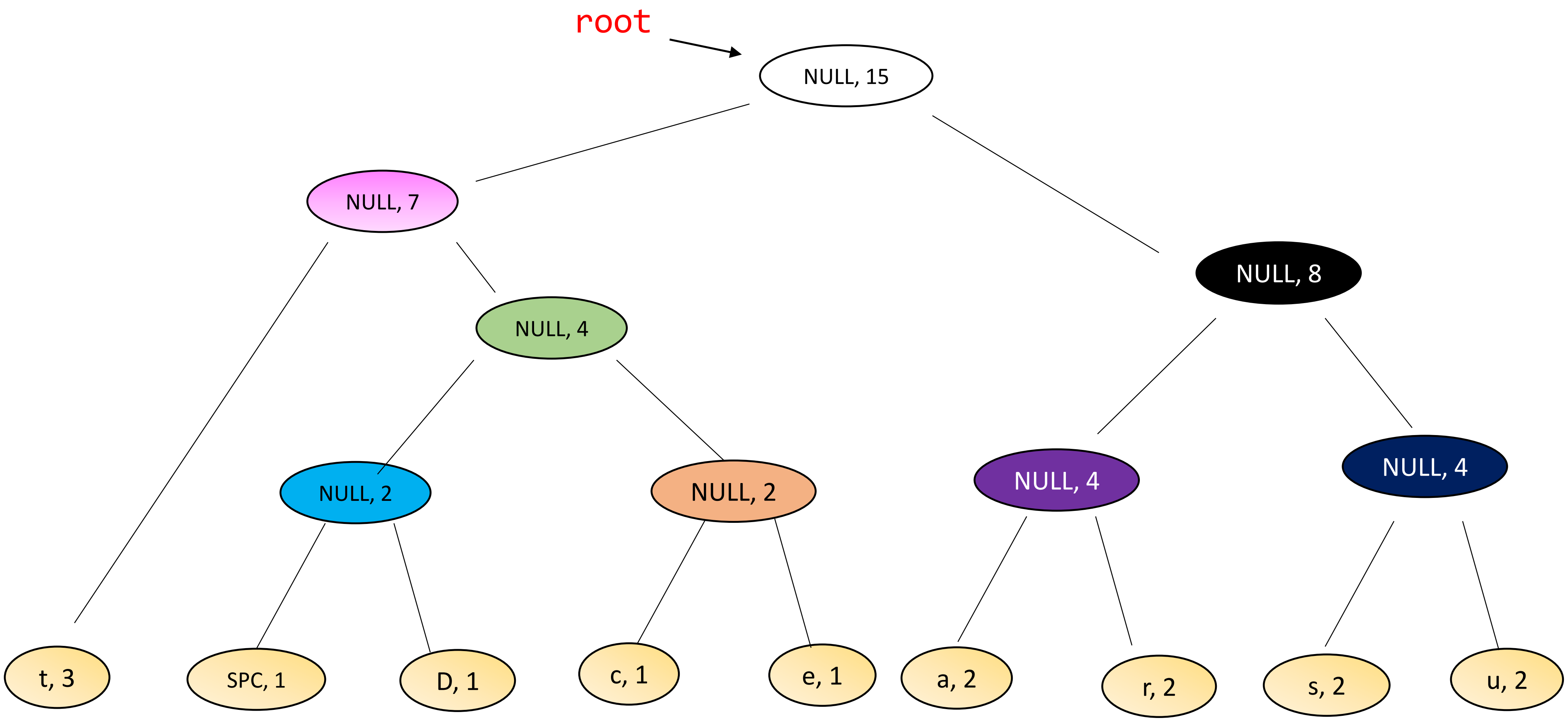

NULL, 7
NULL, 8

Solution

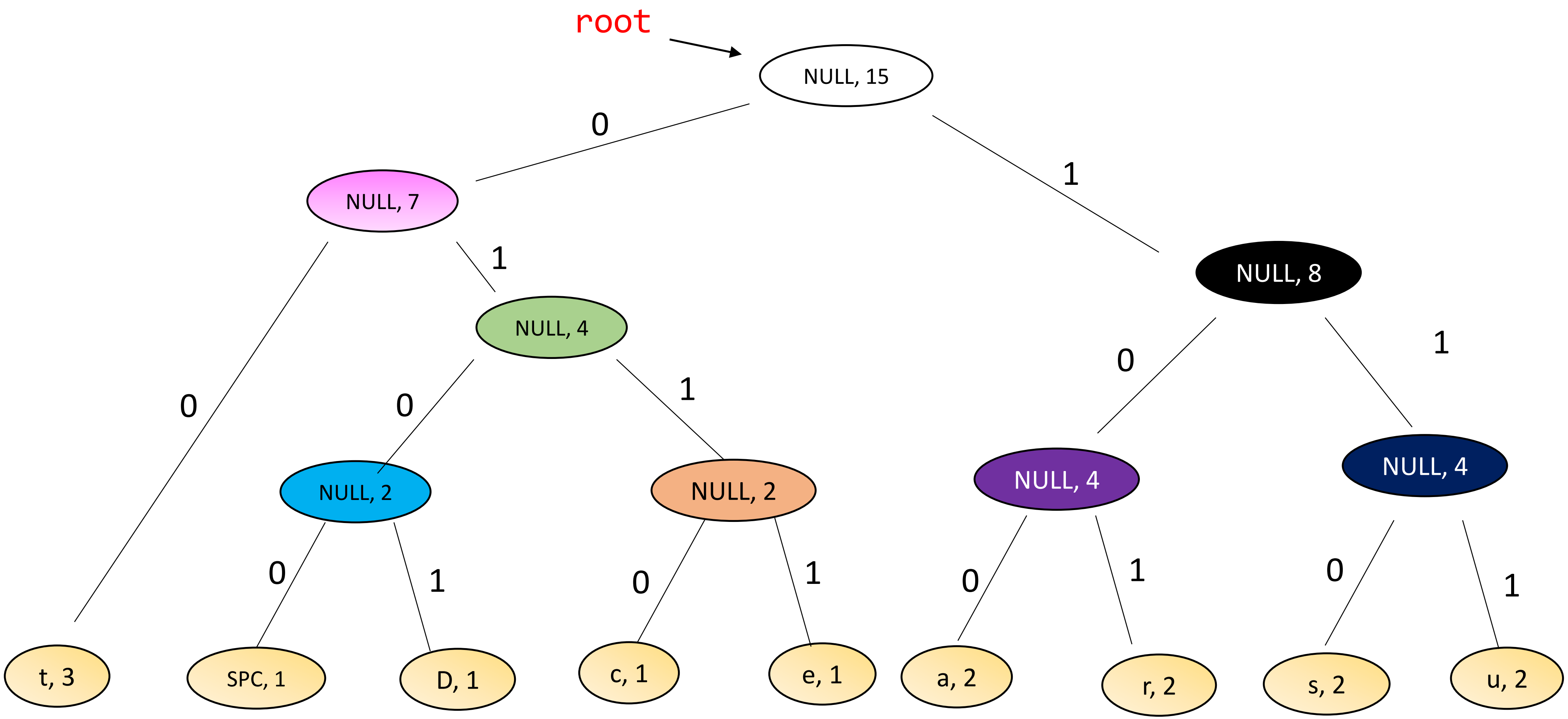


NULL, 15
NULL, 7
NULL, 8
NULL, 4
NULL, 2
NULL, 2
NULL, 4
NULL, 4
t, 3
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2
NULL, 15

Solution


root
NULL, 15
NULL, 7
NULL, 8
NULL, 4
NULL, 2
NULL, 2
NULL, 4
NULL, 4
t, 3
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2

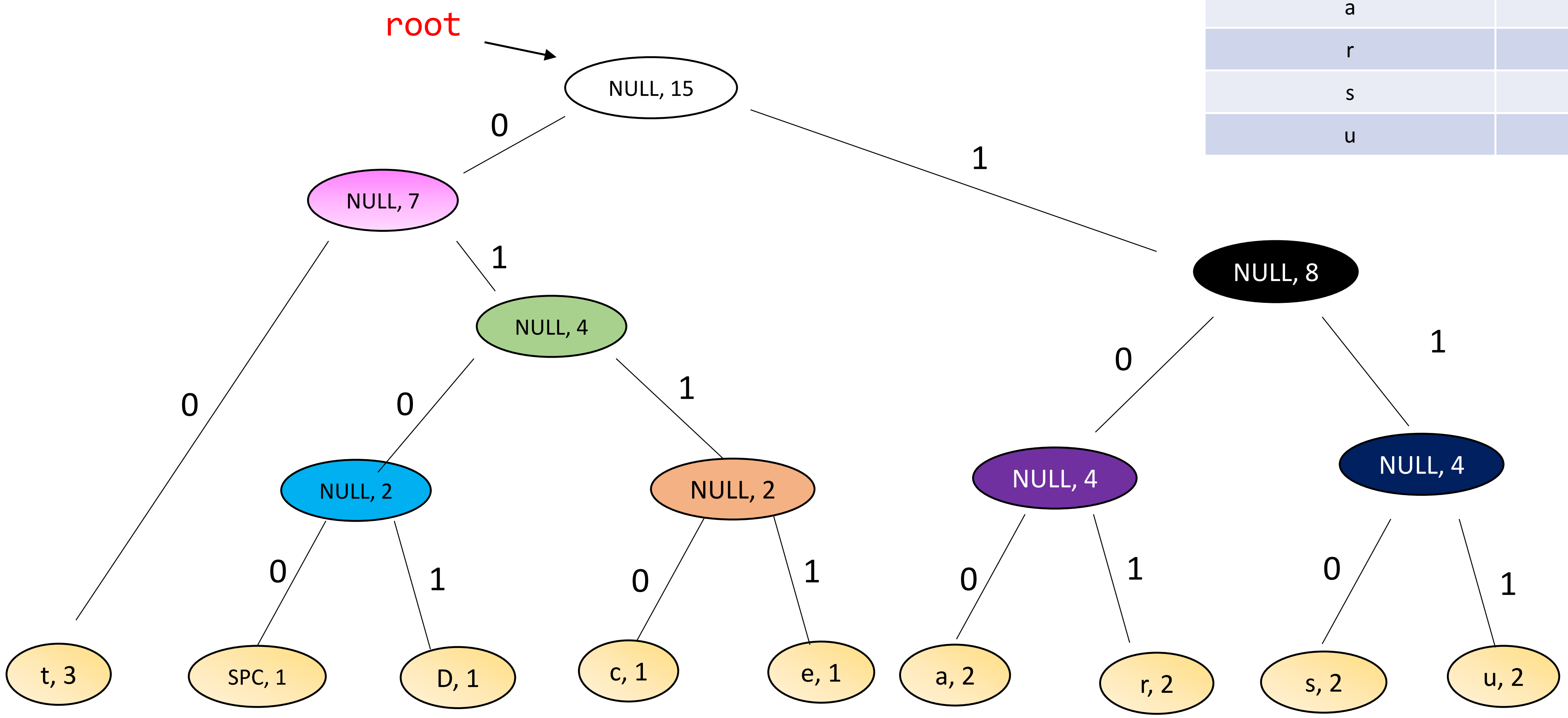
Solution



root
NULL, 15
0
1
NULL, 7
NULL, 8
0
1
NULL, 4
0
1
0
1
NULL, 2
NULL, 2
NULL, 4
NULL, 4
0
1
0
1
0
1
0
1
t, 3
SPC, 1
D, 1
c, 1
e, 1
a, 2
r, 2
s, 2
u, 2

Solution

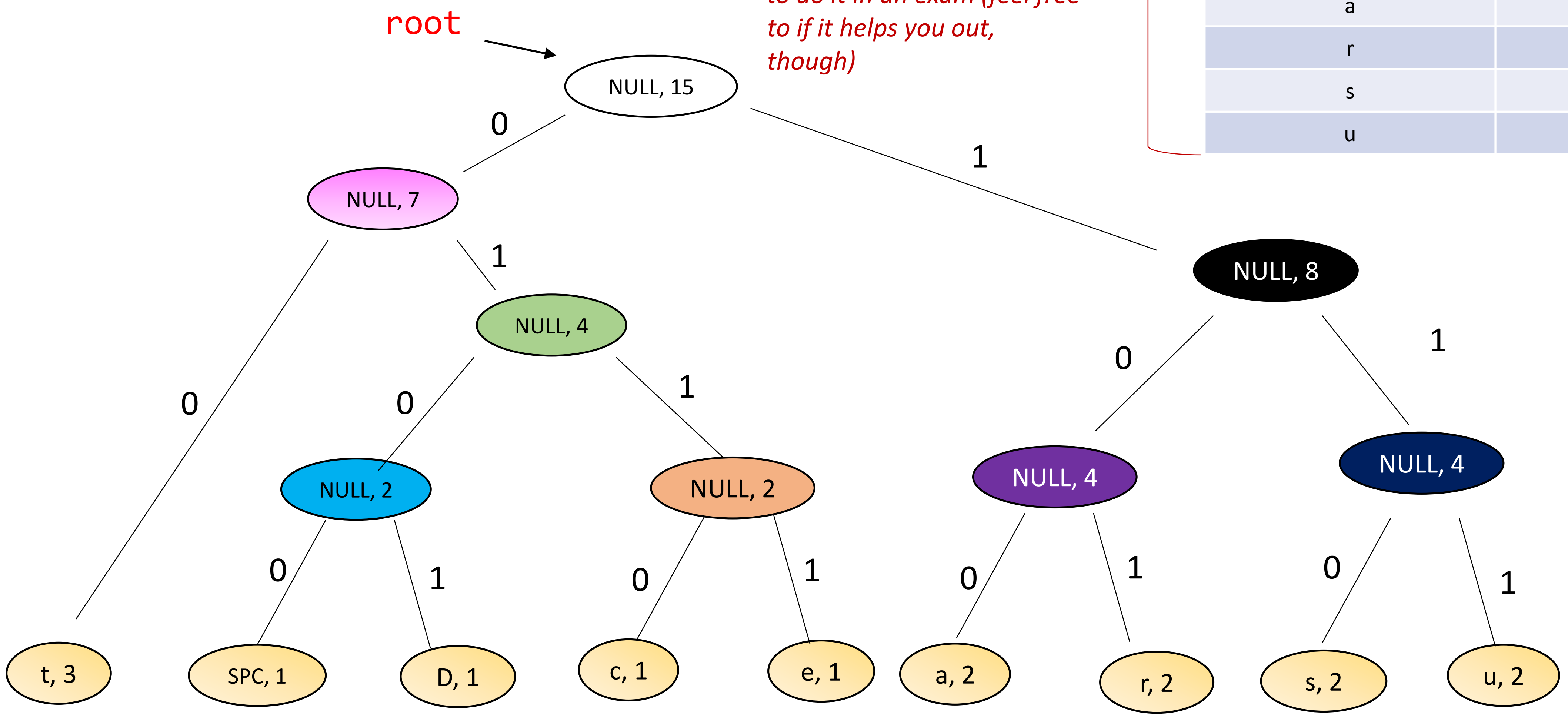
Character	Huffman Encoding
t	00
SPC	0100
D	0101
c	0110
e	0111
a	100
r	101
s	110
u	111



Solution

Remember: This step is necessary to encode the string, but you won't have to do it in an exam (feel free to if it helps you out, though)

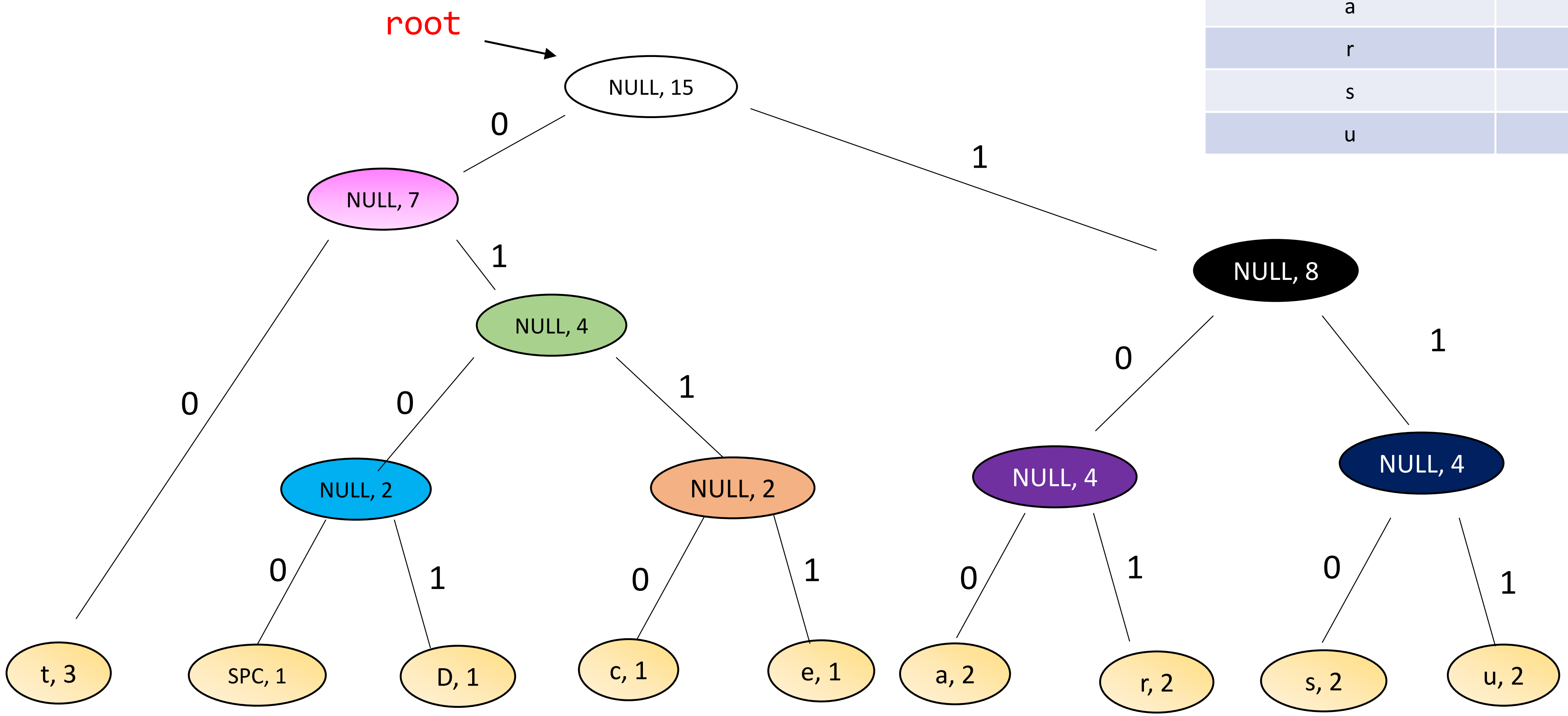
Character	Huffman Encoding
t	00
SPC	0100
D	0101
c	0110
e	0111
a	100
r	101
s	110
u	111



Solution

Transmission: “Data structures” =
0101100001000100110001011
1101100011110101111110

Character	Huffman Encoding
t	00
SPC	0100
D	0101
c	0110
e	0111
a	100
r	101
s	110
u	111

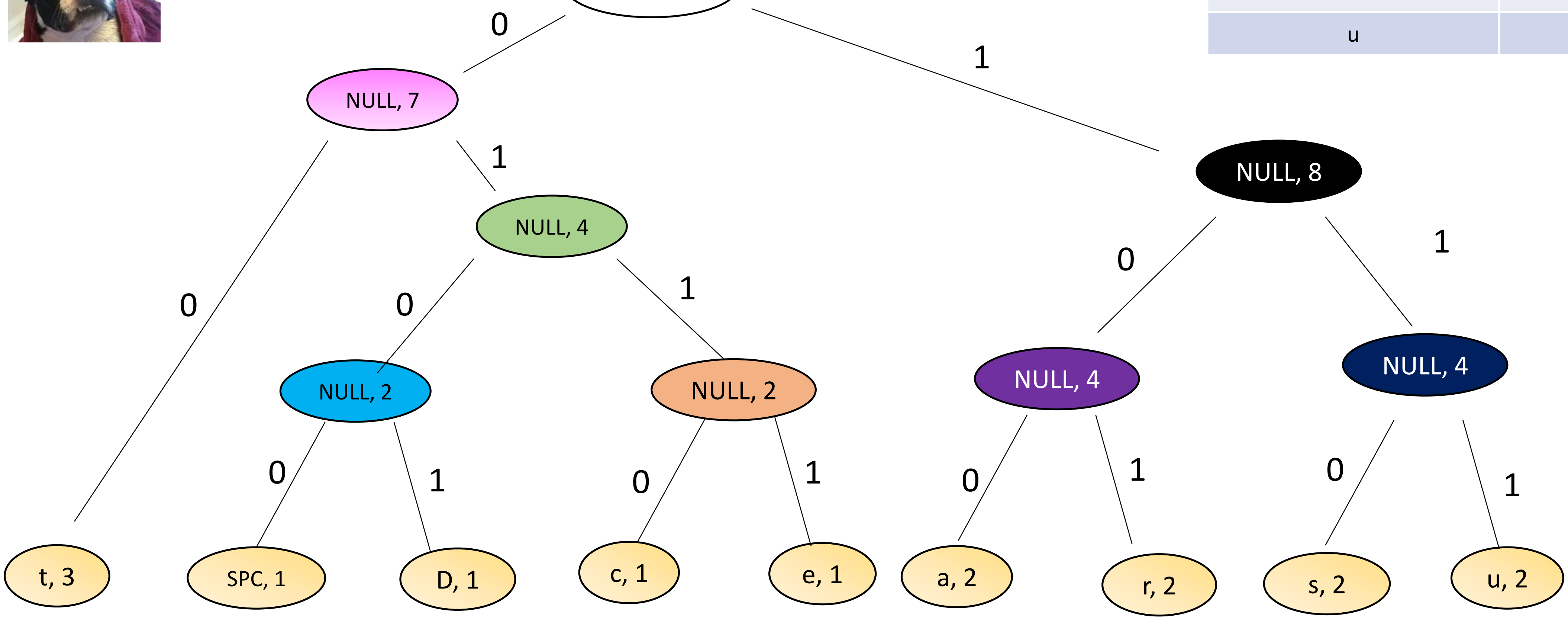


Solution

Transmission: "Data structures" =
0101100001000100110001011
1101100011110101111110

46 bits. Compare with 15 * 7 = 105 for ASCII… (43.8%)

Character	Huffman Encoding
t	00
SPC	0100
D	0101
c	0110
e	0111
a	100
r	101
s	110
u	111



Encoding phase

- Formally, what we did is known as the encoding phase of Huffman.
- We encoded characters from some alphabet Σ based on their frequencies in a (hopefully representative!) text.

Encoding phase

- Formally, what we did is known as the encoding phase of Huffman.
- We encoded characters from some alphabet Σ based on their frequencies in a (hopefully representative!) text.
- **But what if some characters from Σ never appear in the text?**

Encoding phase

- Formally, what we did is known as the encoding phase of Huffman.
- We encoded characters from some alphabet Σ based on their frequencies in a (hopefully representative!) text.
- **But what if some characters from Σ never appear in the text?**
  - Well, then maybe there's something to be said about the "representativeness" of the text…

Encoding phase

- Formally, what we did is known as the encoding phase of Huffman.
- We encoded characters from some alphabet Σ based on their frequencies in a (hopefully representative!) text.
- **But what if some characters from Σ never appear in the text?**
  - Well, then maybe there's something to be said about the "representativeness" of the text…
  - **Easy solution:** Add 1 to every other node's values and set the values for all the non-appearing characters be the previous minimum value. This we do while we compute the frequencies, by another linear scan over a $|\Sigma|$ −large histogram, not the text T! So we still only have to scan the text **once** to build the trie.

Encoding phase

- **Invariant satisfaction**: Recall; one of our goals was for no code to be the prefix of another.

Encoding phase

- **Invariant satisfaction**: Recall; one of our goals was that no code to be the prefix of another.
- This is trivially satisfied by Huffman Encoding, since all codes are generated by following paths from the root to the leaves!
  - And there's nothing "below" the leaves!

Criticism of Huffman's encoding phase

- Huffman has to traverse the string twice, once for frequency computation and once for encoding. ☹
- This is one of the main criticisms of Huffman encoding.

Criticism of Huffman's encoding phase

- Huffman has to traverse the string twice, once for frequency computation and once for encoding. ☹
- This is one of the main criticisms of Huffman encoding.
- Huffman encoding can be seen as a method for string compression, since we save memory space per character.

Criticism of Huffman's encoding phase

- Huffman has to traverse the string twice, once for frequency computation and once for encoding. ☹
- This is one of the main criticisms of Huffman encoding.
- Huffman encoding can be seen as a method for string compression, since we save memory space per character.
  - However, the LZW compression algorithm does better ☺