Huffman Coding

Amrita Kaur

August 8, 2023

Contributions made from previous CS106B Instructors

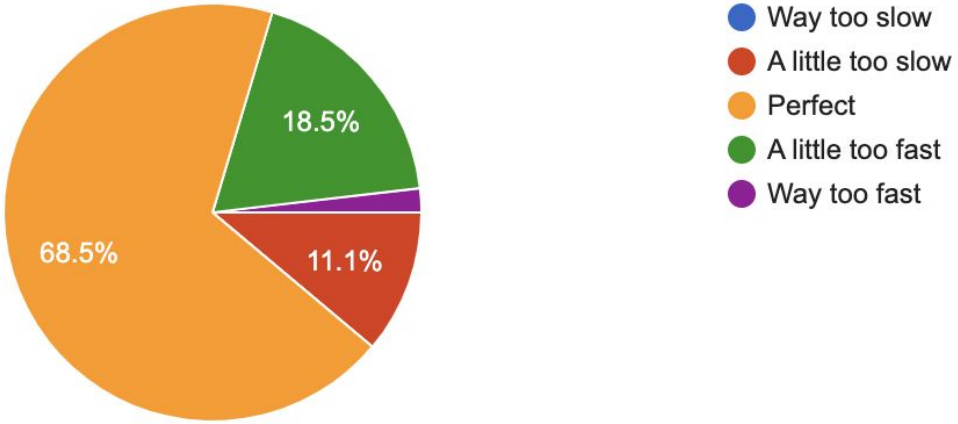
Announcements

- Assignment 5 due tomorrow at 11:59pm
- Assignment 6 will be released tomorrow
  - YEAH Hours from **4-5pm** with Bryant (different time!)
- Final exam materials are available here
  - August 18th from 3:30-6:30pm (3 hours)
  - Same format and grading criteria as midterm
  - Content from the entire quarter, with an emphasis of latter half

Feedback

Rate the pace of lecture

54 responses



Feedback

Things you liked:

"You use a **ton of examples** in class (really helps me :))"

"I LOVE THE HW STRUCTURE FOR THIS WEEK!!!! I really really like how there is a **requirement box** it helped me not real overwhelmed or like I was missing anything."

"**Friday review sessions** turn out to be super useful for topics that take a little more time & effort to sink in"

"I like the **code tracing and diagrams** on the slides!"

Feedback

Places we can improve:

“I think it would be great if maybe you could do this once or twice weekly (on Ed or maybe at the end of a lecture) where you show us correct code but **code that is stylistically bad**”

“Possibly **varying the feedback** questions a bit?”

“Can we do more **finals review** material earlier this time?”

Feedback

We hear you…

“Explain more niche cases, such as pointers and references, pointers vs references, dereferencing pointers, (void) pointer, ect.” **Take CS107!**

“Perhaps less required testing on the homework? It sometimes takes about as long as the actual coding :/”

“Maybe having some of the coding we did in lecture uploaded somewhere.” **On the course website!**

“[SL] is the best. [They] deserves a raise” **We agree!!**

Feedback

Anything else you would like us to know:

“The "cute" coding comics make me want to throw my computer out the window.”

“I feel like saying "my code is buggy" is too cute, it should be called waspy or something. Saying "I have a wasp in my code" feels way more fitting.”

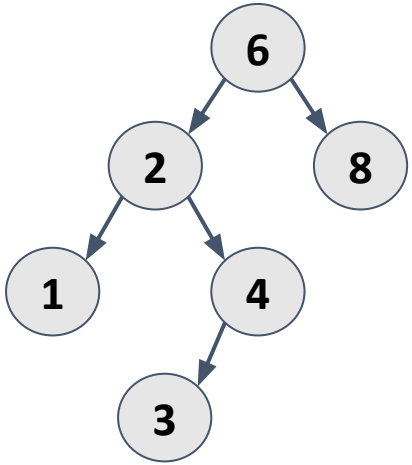
“I think I’m going to pass!”

“I asked a girl out to go watch the barbie movie with me and she said no :(“

Recap: Binary Search Trees

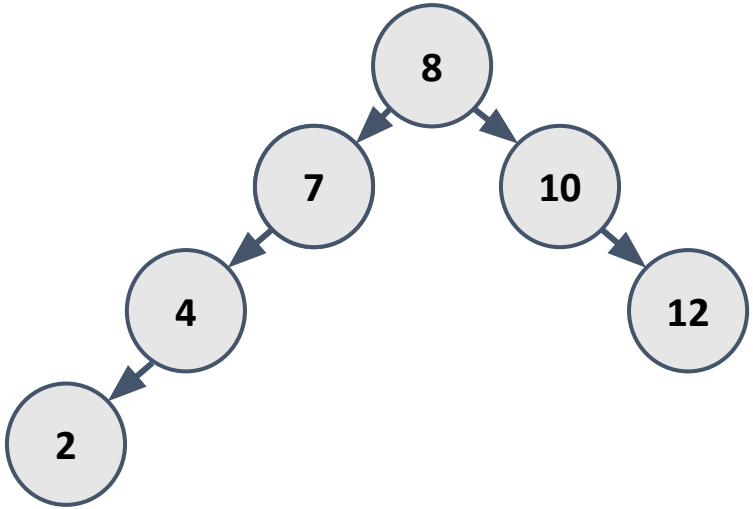
Binary Search Trees (BSTs)

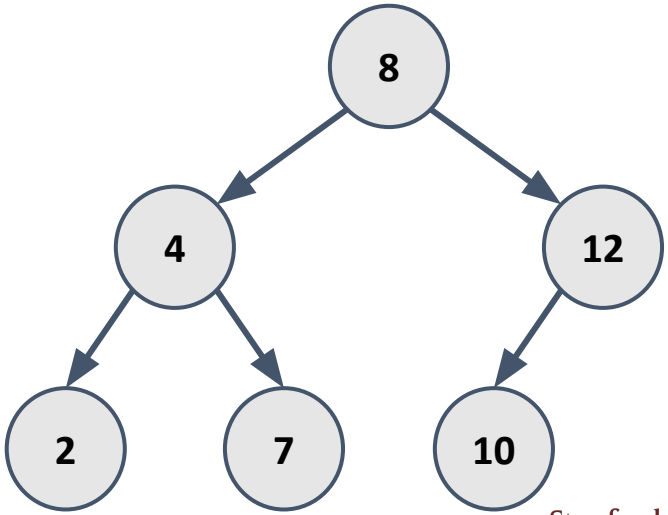
1. Binary tree (each node has 0, 1, or 2 children)
2. For a node with value X:
   a. All nodes in its left subtree must be less than X
   b. All nodes in its right subtree must be greater than X



Takeaways

- There can be multiple valid BSTs for the same set of data
- How you construct the tree matters!





Big-O of ADT Operations

Vectors

- `.size()` – O(1)
- `.add()` – O(1)
- `v[i]` – O(1)
- `.insert()` – O(n)
- `.remove()` – O(n)
- `.sublist()` – O(n)
- `traversal` – O(n)

Grids

- `.numRows()` – O(1)
- `.numCols()` – O(1)
- `grid[i][j]` – O(1)
- `.inBounds()` – O(1)
- `traversal` – $O(n^2)$

Queues

- `.size()` – O(1)
- `.peek()` – O(1)
- `.enqueue()` – O(1)
- `.dequeue()` – O(1)
- `.isEmpty()` – O(1)
- `traversal` – O(n)

Stacks

- `.size()` – O(1)
- `.peek()` – O(1)
- `.push()` – O(1)
- `.pop()` – O(1)
- `.isEmpty()` – O(1)
- `traversal` – O(n)

Sets

- `.size()` – O(1)
- `.isEmpty()` – O(1)
- **`.add()` – O(log n)**
- **`.remove()` – O(log n)**
- **`.contains()` – O(log n)**
- `traversal` – O(n)

Maps

- `.size()` – O(1)
- `.isEmpty()` – O(1)
- **`m[key]` – O(log n)**
- **`.contains()` – O(log n)**
- `traversal` – O(n)

The Power of Abstraction

- The client doesn't need to know we're using a BST behind the scenes, they just need to be able to store their data
  - After all, you've used a Set all quarter without needing to know this!

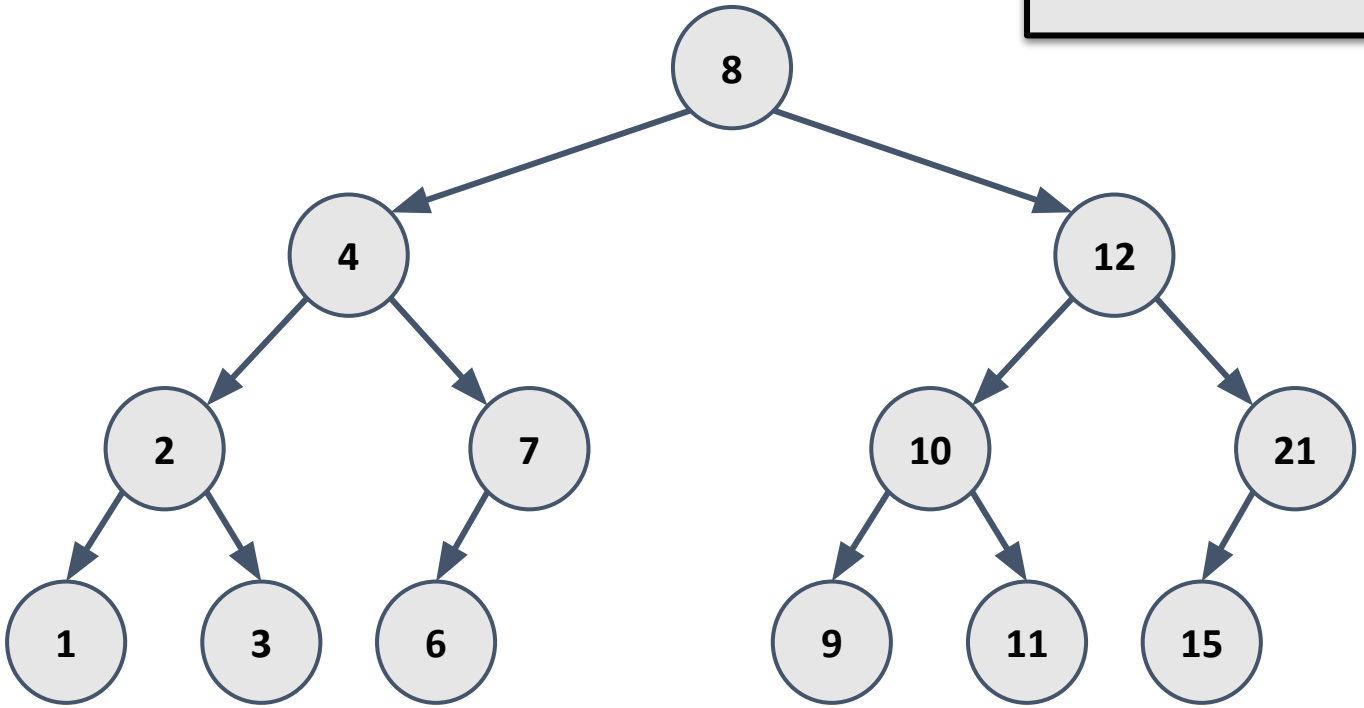
```
OurSet set;
set.add(8);
set.add(9);
set.add(4);
set.contains(5); // false
set.contains(4); // true
set.remove(8);
set.remove(9);
```

BST Lookups

These data structures are designed for fast lookups!

BST Lookups

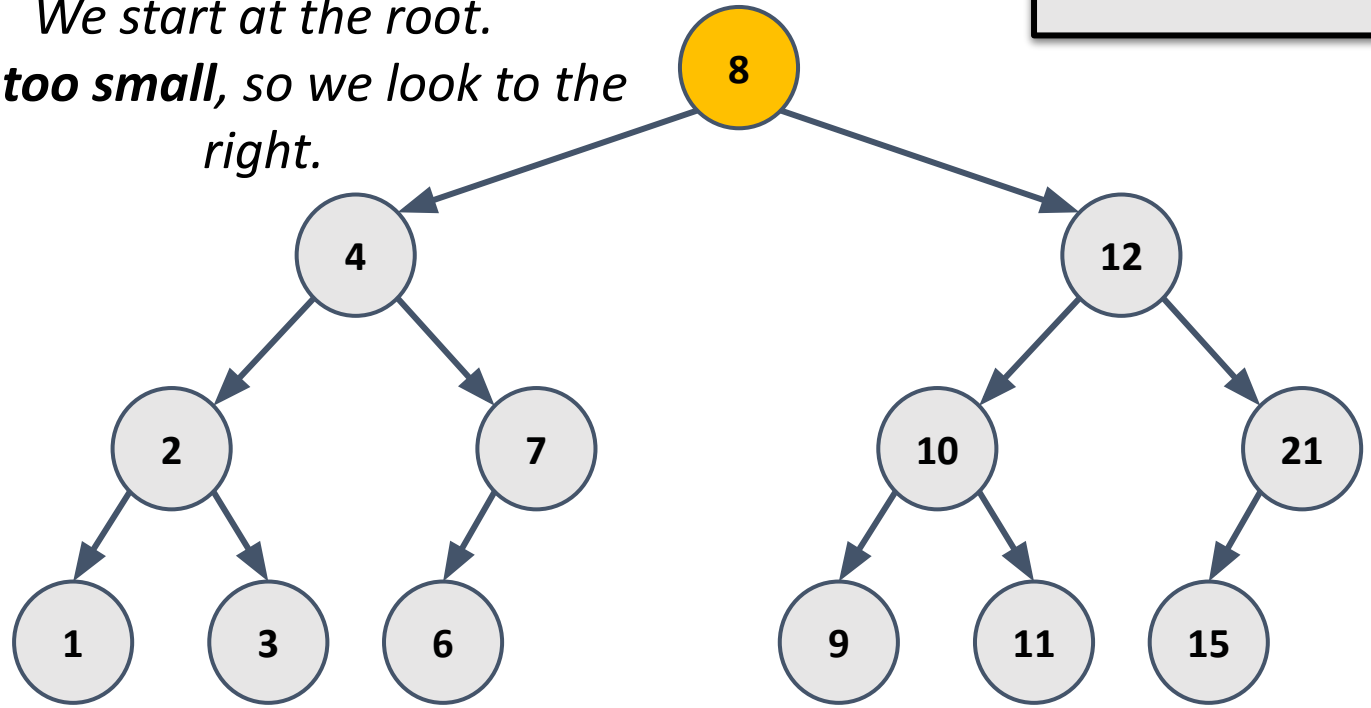
Is 11 in this BST?



BST Lookups

Is 11 in this BST?

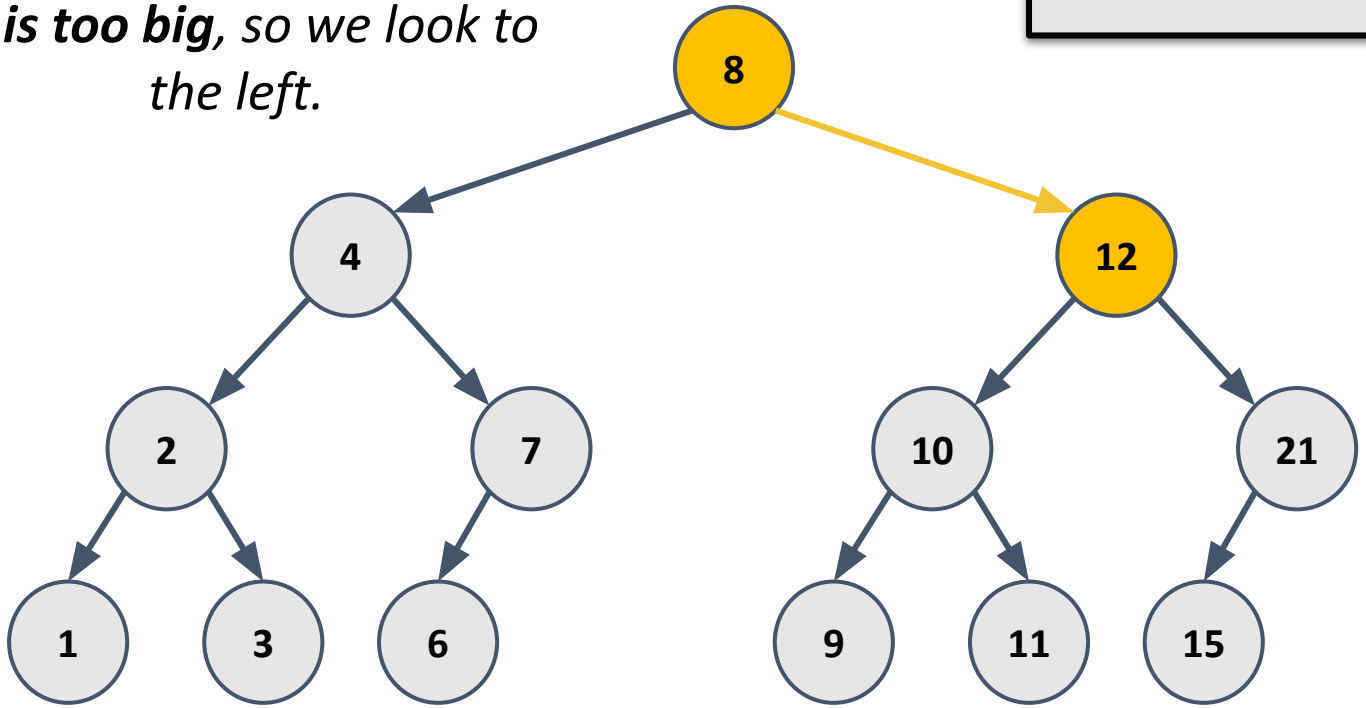
We start at the root.
***8 is too small**, so we look to the right.*



BST Lookups

Is 11 in this BST?

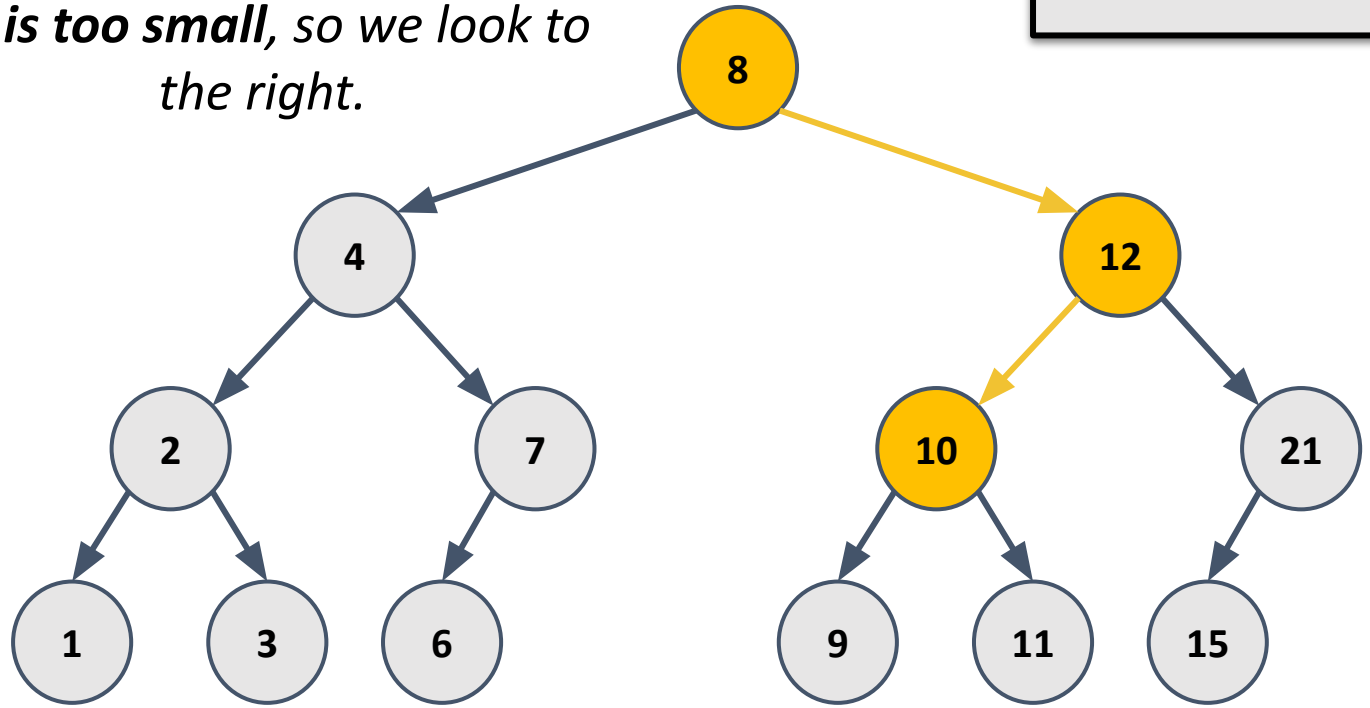
***12 is too big**, so we look to the left.*



BST Lookups

Is 11 in this BST?

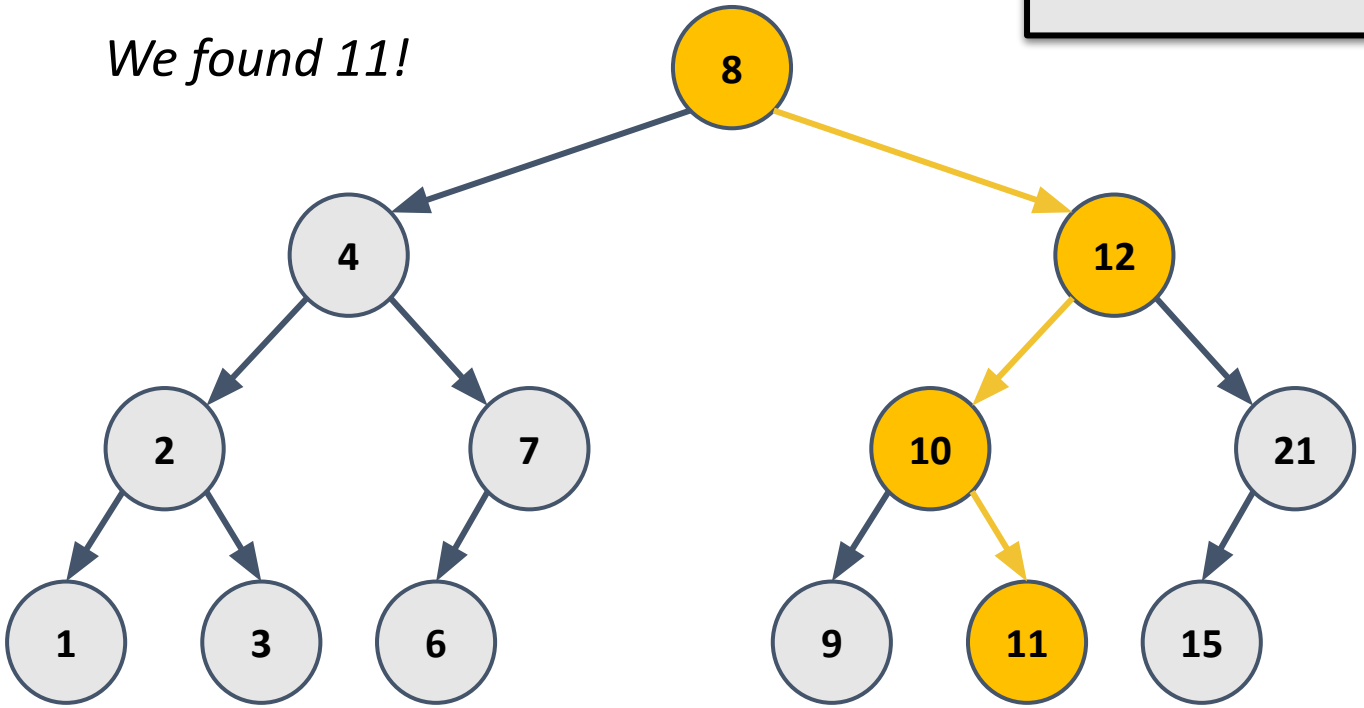
***10 is too small**, so we look to the right.*



BST Lookups

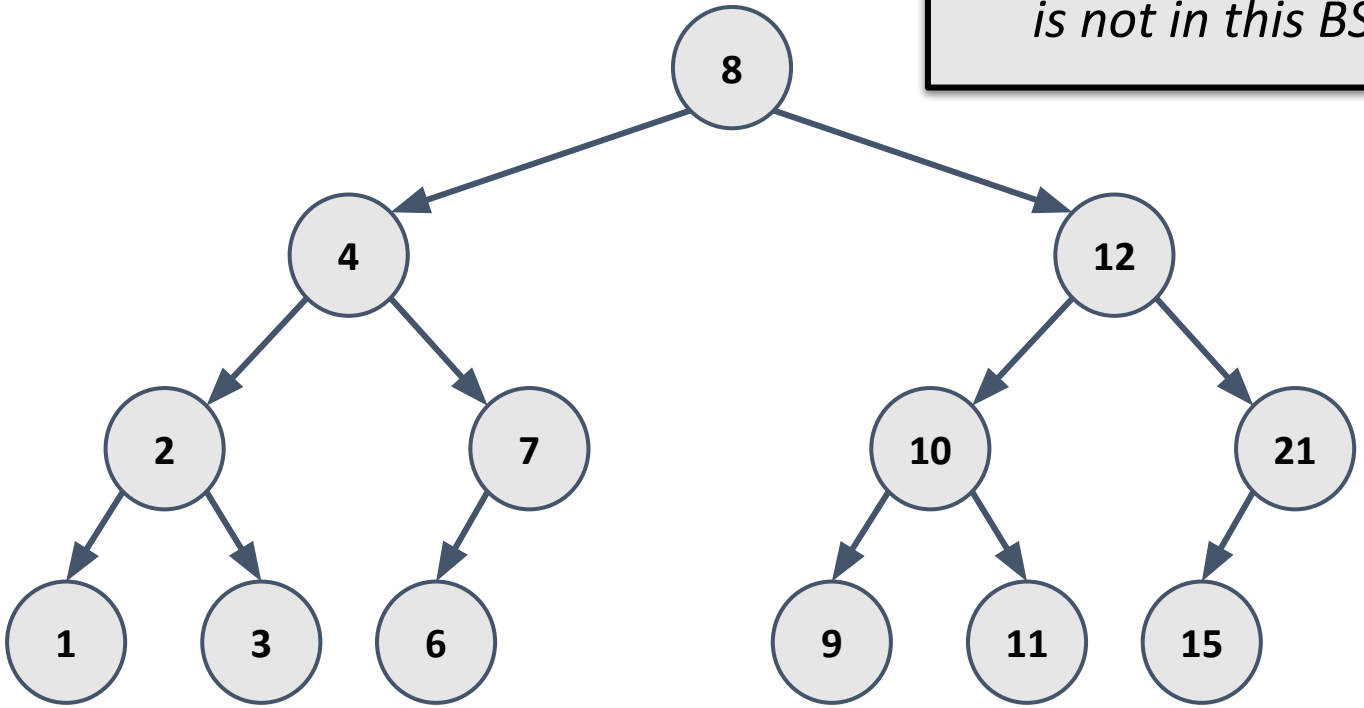
Is 11 in this BST?

We found 11!



BST Lookups

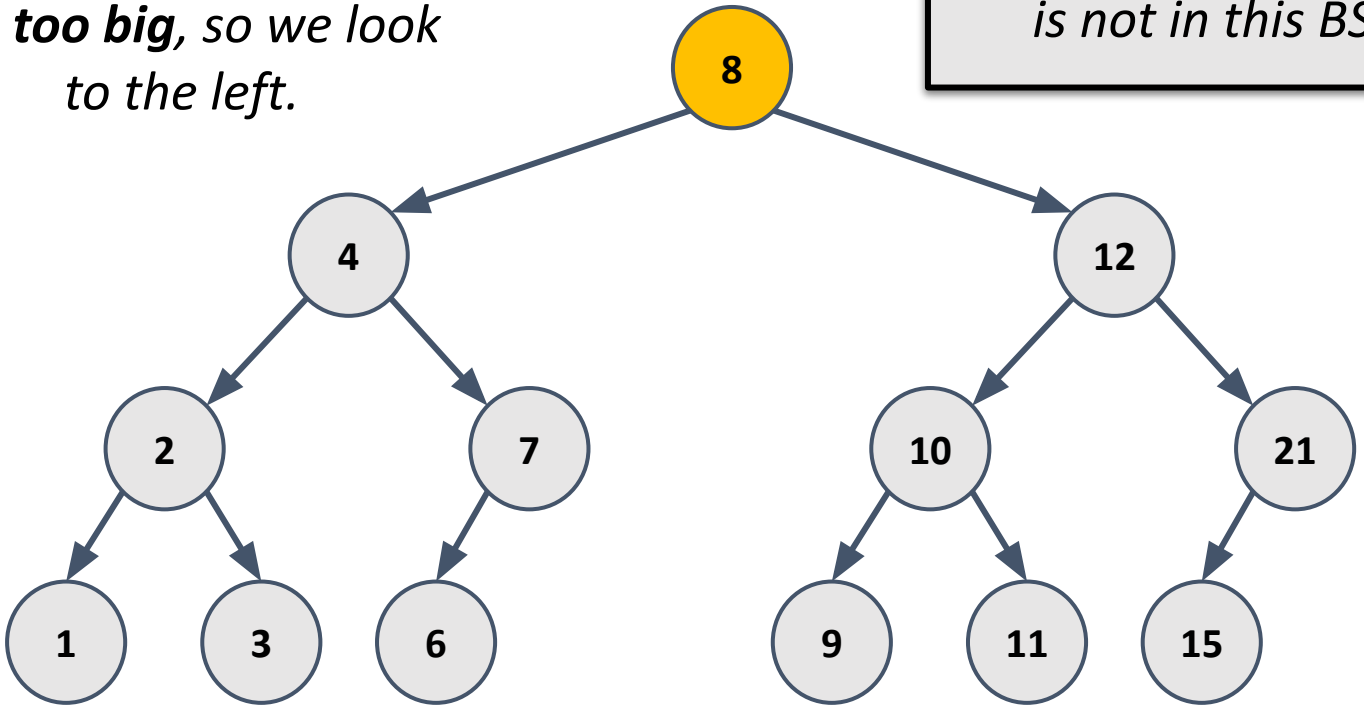
How do we know that 5 is not in this BST?



BST Lookups

***8 is too big**, so we look to the left.*

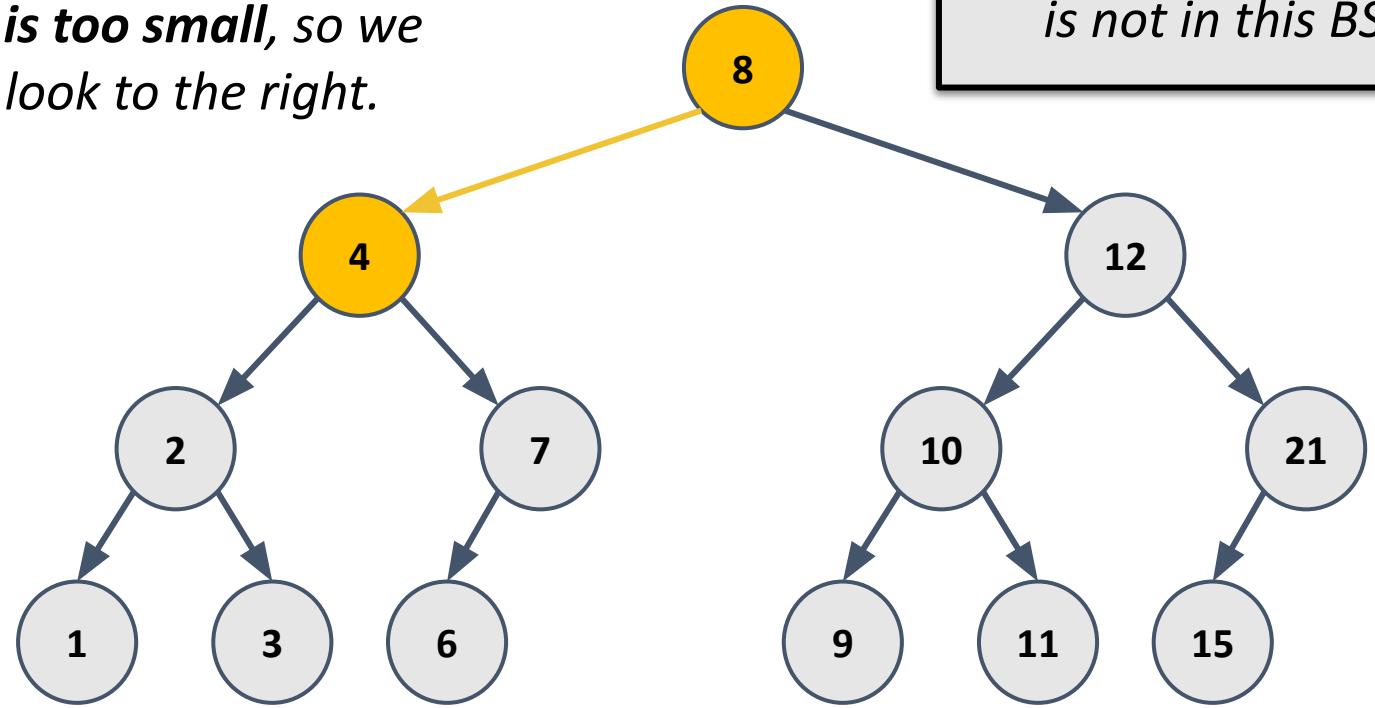
How do we know that 5 is not in this BST?



BST Lookups

***4 is too small**, so we look to the right.*

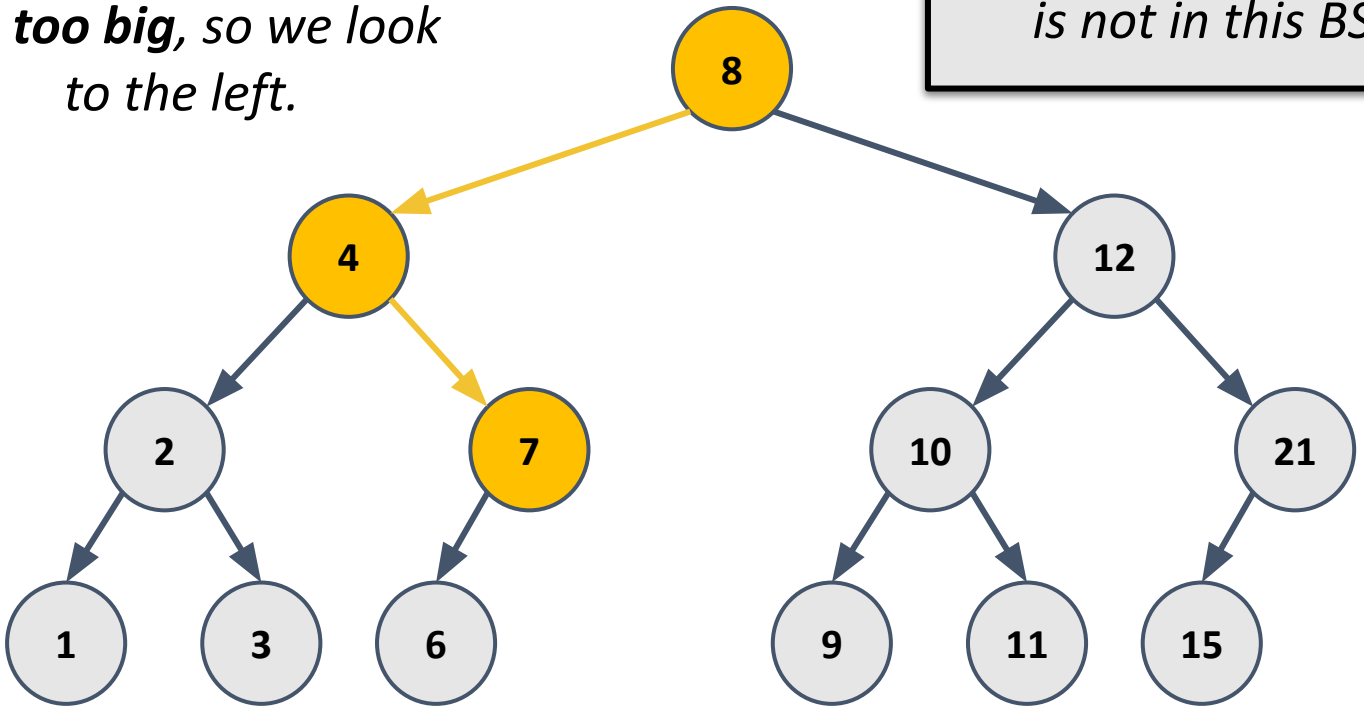
How do we know that 5 is not in this BST?



BST Lookups

***7 is too big**, so we look to the left.*

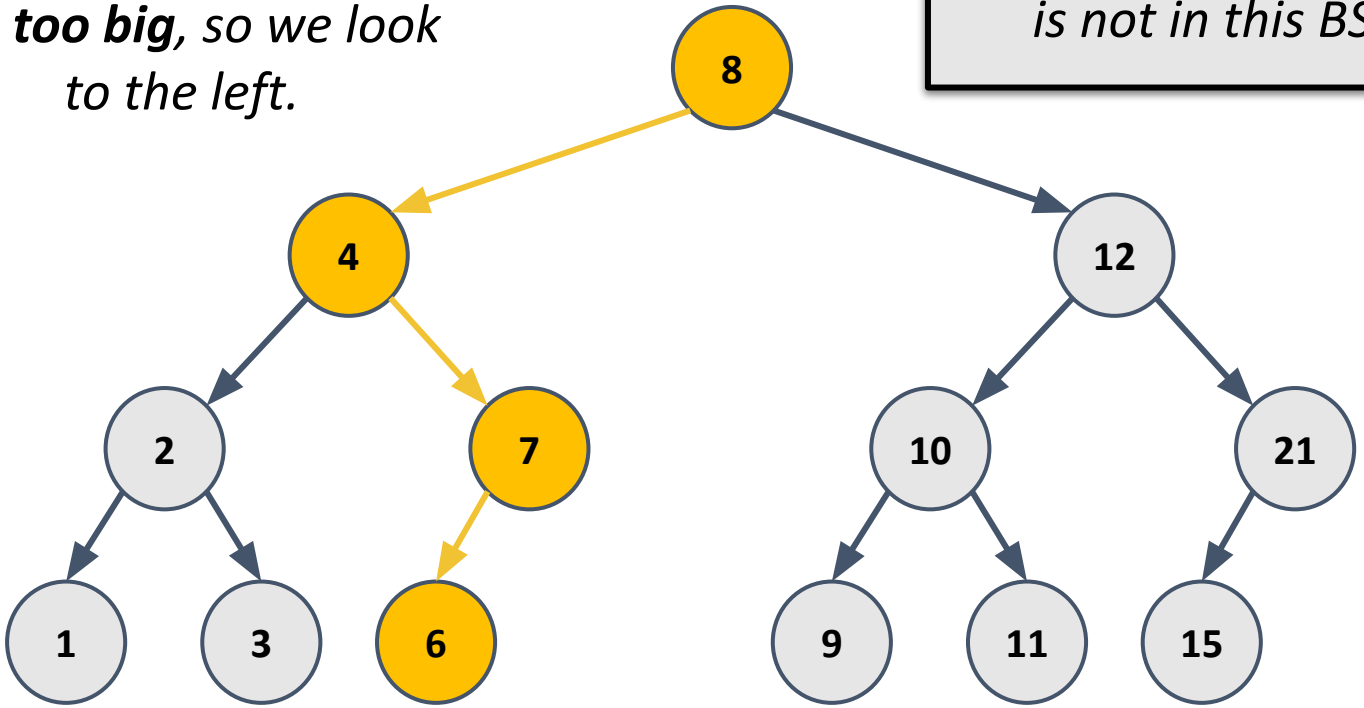
How do we know that 5 is not in this BST?



BST Lookups

***6 is too big**, so we look to the left.*

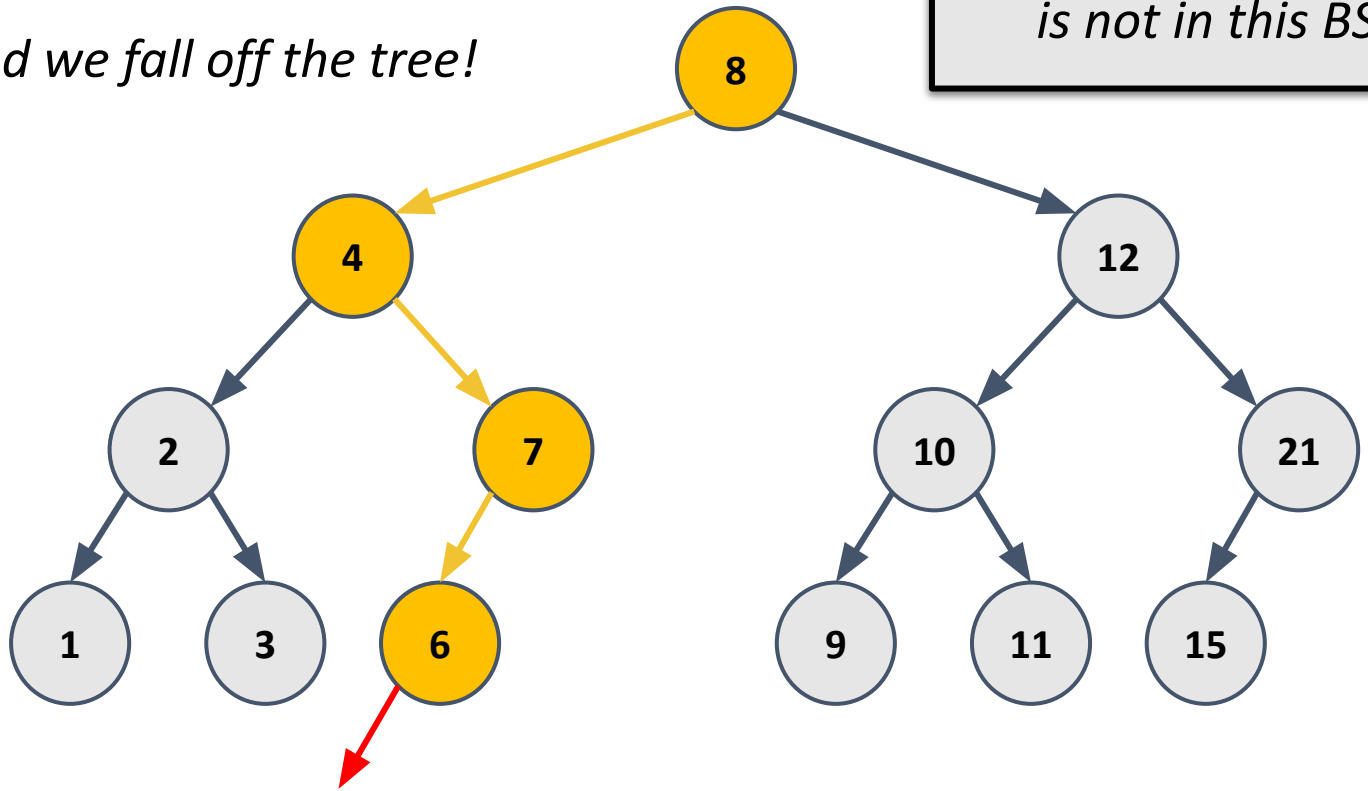
How do we know that 5 is not in this BST?



BST Lookups

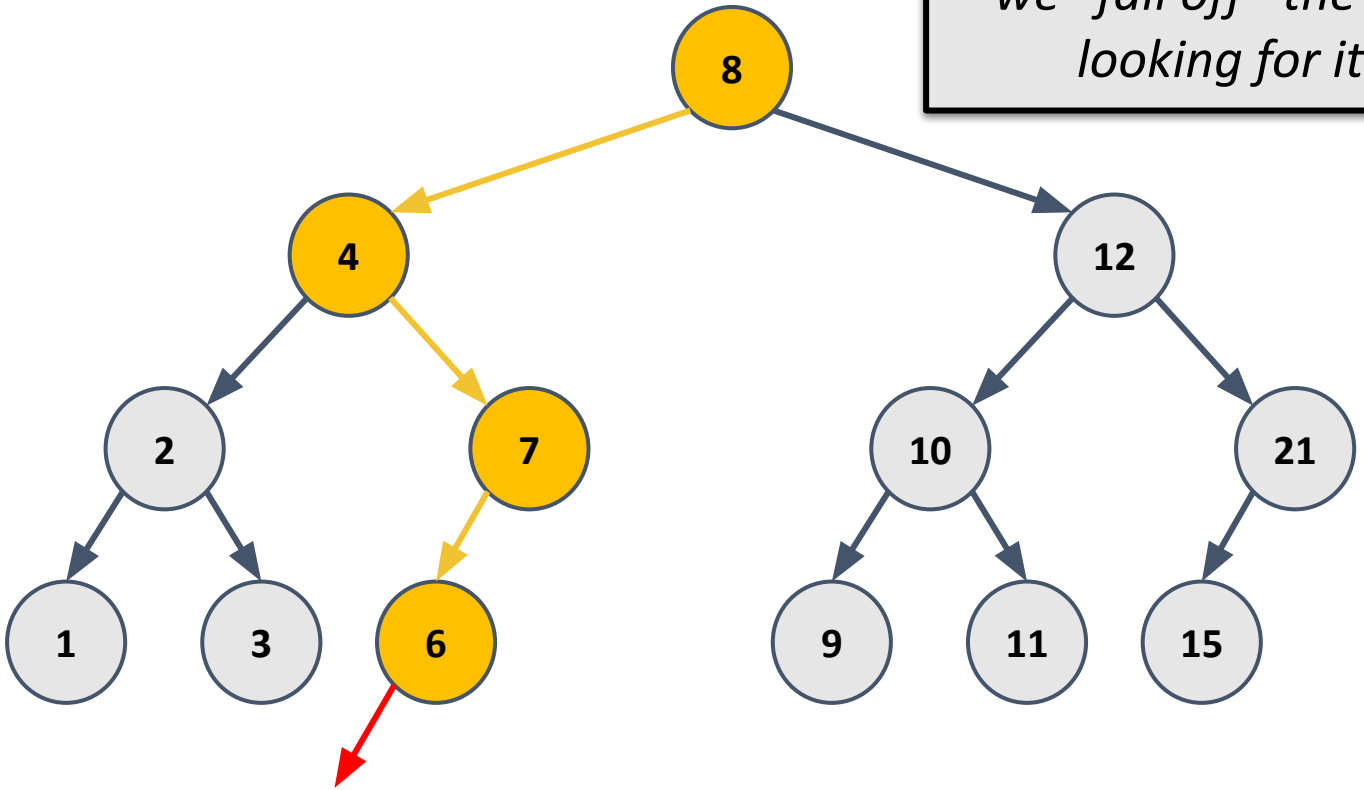
How do we know that 5 is not in this BST?

And we fall off the tree!



BST Lookups

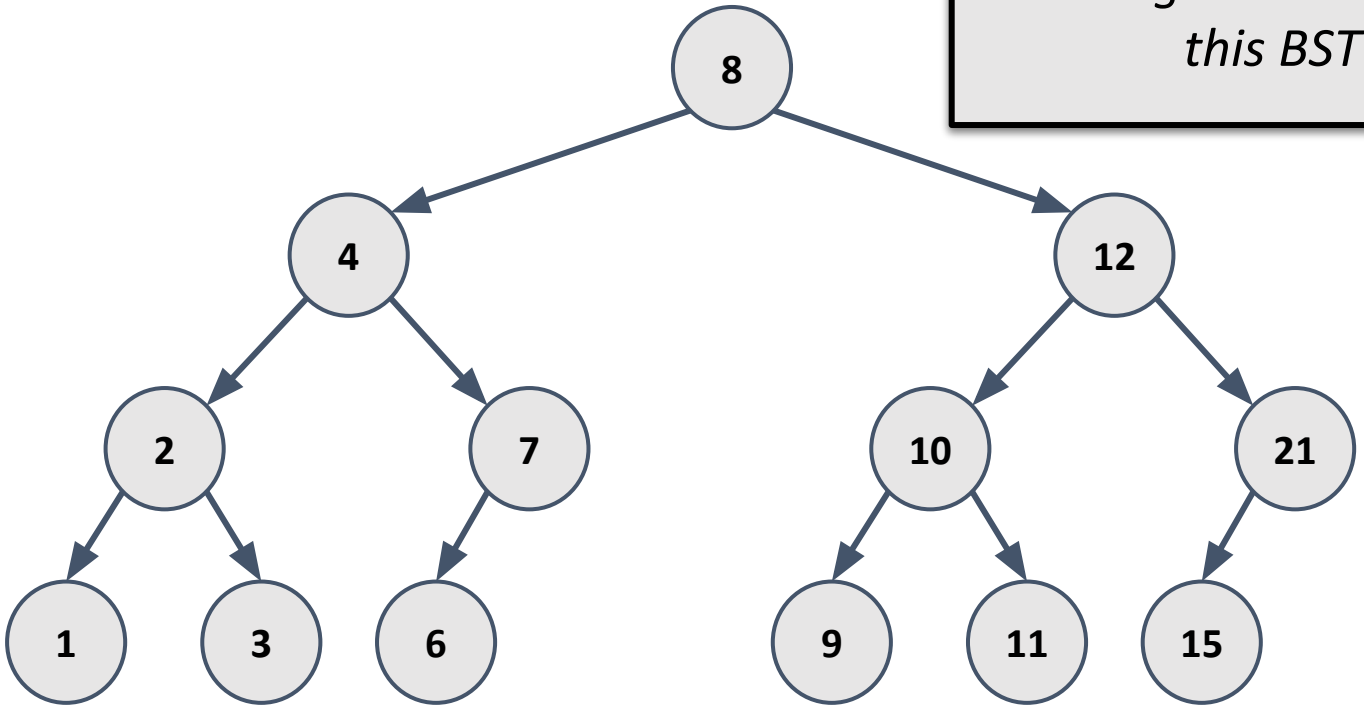
A value isn't in our BST if we "fall off" the tree looking for it.



BST Insertion

BST Insertion

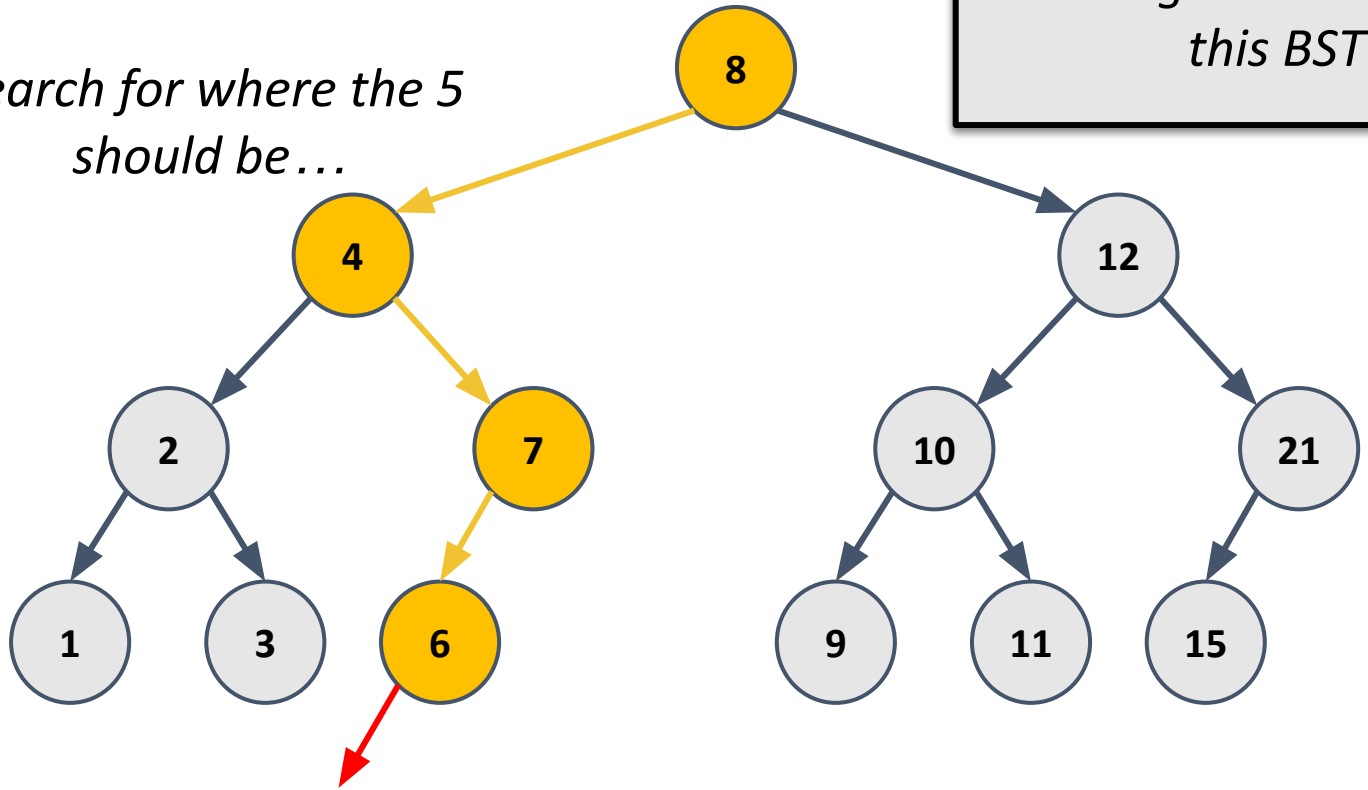
How might we insert 5 into this BST?



BST Insertion

How might we insert 5 into this BST?

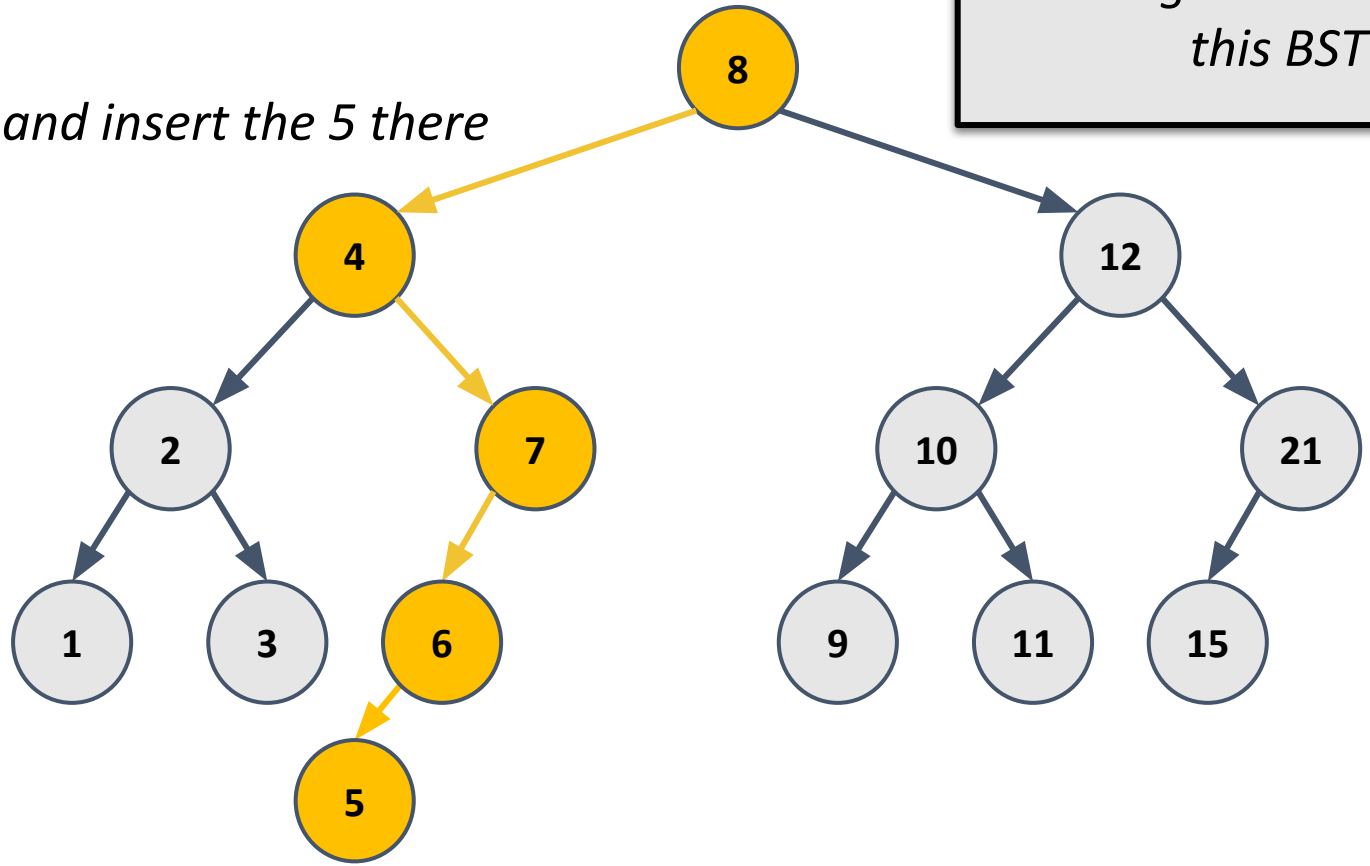
Search for where the 5 should be…



BST Insertion

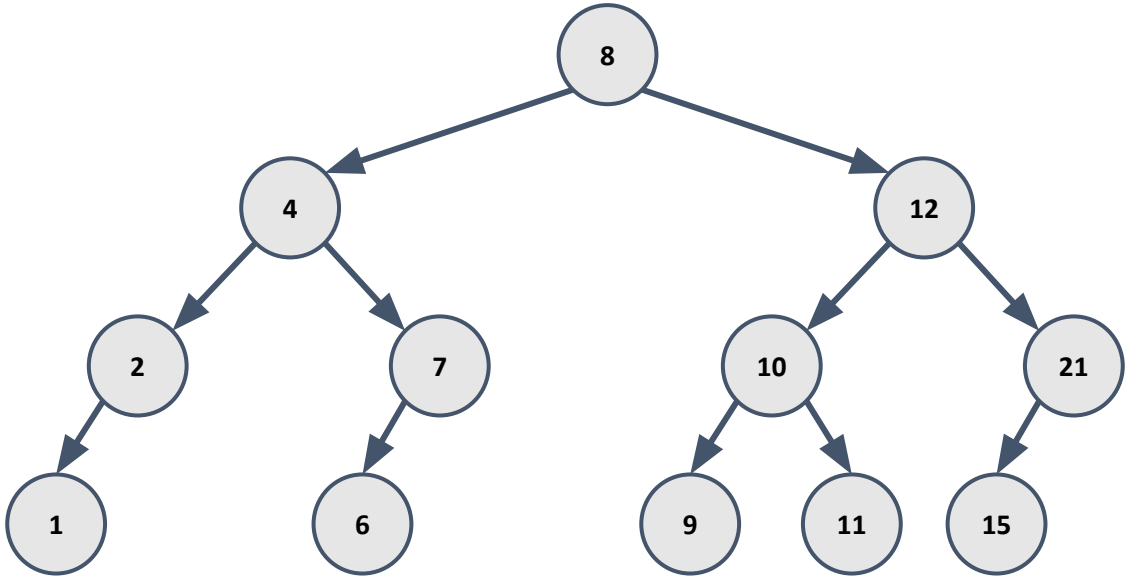
How might we insert 5 into this BST?

. . . and insert the 5 there



Takeaways

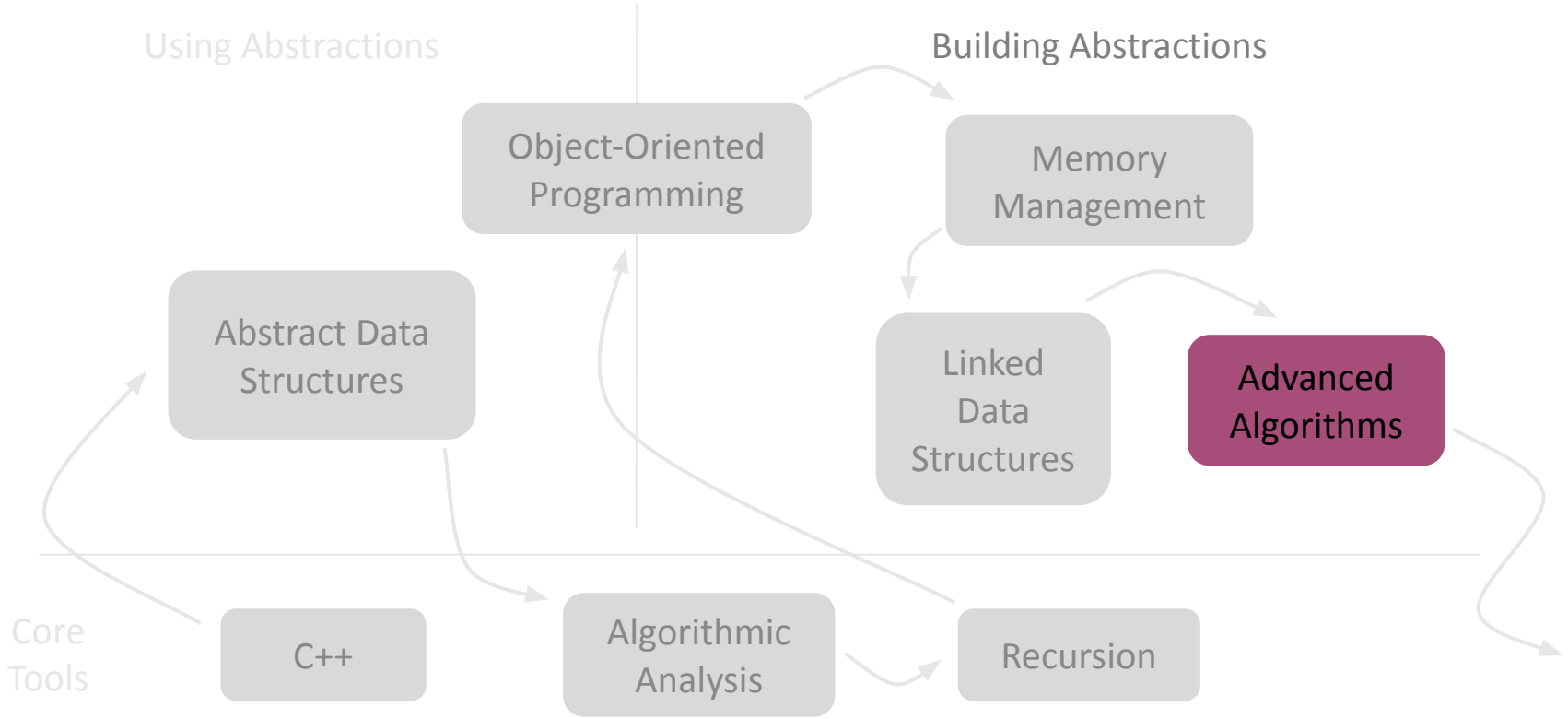
- To insert/delete nodes, we have to look them up in our BST
  - This is why insertions/deletions are $O(\log n)$, just like lookups



Let’s code it up!

Implement OurSet with a BST

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Data Storage and Representation

How do computers store and represent data?

How do computers store and represent data?

Just a Little Bit of Magic

- Digital data is stored as sequences of 0s and 1s
  - These sequences are encoded in physical devices by magnetic orientation on small (10 nm) metal particles or by trapping electrons in small gates
- A single 0 or 1 is called a **bit**
- A group of 8 bits is called a **byte**

```
00000000, 00000001, 00000010, 00000011, 00000100, …
```

- There are 2^8, so 256, different bytes
  - Good recursive backtracking practice: Write a function that lists all possible byte sequences!

Remember: The Hexadecimal Number System

- We typically represent numbers using the decimal (base-10) number system
  - Each place value represents a factor of ten (ones, tens, hundreds, etc.)
  - 10 possible digits for each place value
- In computer systems,it is often more convenient to express numbers using the hexadecimal (base-16) number system.
  - Each place value represents a factor of 16 (16^0, 16^1 , 16^2, etc.)
  - 16 possible "digits" for each place value.
    - 10 numerical digits (0-9) and the letters ‘a’ to ‘f’
    - `0 1 2 3 4 5 6 7 8 9 a(10) b(11) c(12) d(13) e(14) f(15)`
- The prefix `0x` is used to communicate that a number is being expressed in hexadecimal

Binary Number System

- The system of using sequences of 0s and 1s to represent data is called binary
  - Binary can be used to encode numbers, text, images, etc.
- Binary numbers are expressed using a base-2 number system
  - Each place value represents a power of 2 (2^0, 2^1 , 2^2, etc.)
- Representing my age in different numerical systems:
  - In base-10, I'm 24 ($2 * 10^1 + 4 * 10^0 = 20 + 4$)
  - In base-16, I'm 18 ($1 * 16^1 + 8 * 16^0 = 16 + 8$)
  - In base-2, I'm 11000 ($1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 16 + 8 + 0 + 0 + 0$)
- The prefix `0b` is (sometimes) used to communicate a number is being expressed in binary


THERE ARE ONLY
10 TYPES OF PEOPLE
IN THE WORLD:
THOSE WHO
UNDERSTAND BINARY
AND THOSE WHO DON'T.
FIRST & ELM
PRINTABLE ART • ON ETSY

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc
- However, we just said that computers require everything to be written as zeros and ones
- To bridge the gap, we need to agree on some universal way of representing characters as sequences of bits
- Idea: ASCII!

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

```
01001101010000010101000
```

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

`01001101|01000001|01010000`

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

01001101|01000001|01010000

M

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

`01001101|`**`01000001`**`|01010000`

M **A**

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

`01001101|01000001|`**`01010000`**

M A **P**

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Decoding

What is the mystery word represented by this ASCII encoding?

```
01001101010000010101000
```

MAP

Decimal	Binary	Octal	Hex	ASCII
64	01000000	100	40	@
65	01000001	101	41	A
66	01000010	102	42	B
67	01000011	103	43	C
68	01000100	104	44	D
69	01000101	105	45	E
70	01000110	106	46	F
71	01000111	107	47	G
72	01001000	110	48	H
73	01001001	111	49	I
74	01001010	112	4A	J
75	01001011	113	4B	K
76	01001100	114	4C	L
77	01001101	115	4D	M
78	01001110	116	4E	N
79	01001111	117	4F	O
80	01010000	120	50	P

ASCII Observations

- Every characters uses exactly the same number of bits, 8, which makes it very easy to differentiate between the characters
- Any message with n characters will use exactly 8n bits
  - Space for `RAMBUNCTIOUS`: 8*12 = 96 bits
  - Space for `CS106BisCool`: 8*12 = 96 bits
- Let's make this more efficient by reducing the number of bits we need to encode text

Main Character Today

KIRK'S DIKDIK

ASCII Encoding

- ASCII uses 8 bits to represent each character

character	*ASCII code*
K	01001011
I	01001001
R	01010010
'	00100111
S	01010011
_	00100000
D	01000100

ASCII Encoding

- ASCII uses 8 bits to represent each character
- Let's represent **KIRK'S DIKDIK** in ASCII code

character	*ASCII code*
K	01001011
I	01001001
R	01010010
'	00100111
S	01010011
_	00100000
D	01000100

K	I	R	K	'	S	_	D	I	K	D	I	K

ASCII Encoding

- ASCII uses 8 bits to represent each character
- Let's represent **`KIRK'S DIKDIK`** in ASCII code

character	*ASCII code*
K	01001011
I	01001001
R	01010010
'	00100111
S	01010011
_	00100000
D	01000100

0100 1011	0100 1001	0101 0010	0100 1011	0010 0111	0101 0011	0010 0000	0100 0100	0100 1001	0100 1011	0100 0100	0100 1001	0100 1011
K	I	R	K	'	S	_	D	I	K	D	I	K

A Different Encoding

- If we're specifically writing the string **KIRK'S DIKDIK**, which only has seven different characters, using full bytes is wasteful
- Let's use a 3-bit encoding instead

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

K	I	R	K	'	S	_	D	I	K	D	I	K

A Different Encoding

- If we're specifically writing the string **KIRK'S DIKDIK**, which only has seven different characters, using full bytes is wasteful
- Let's use a 3-bit encoding instead

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

000	001	001	000	011	100	101	101	001	000	101	001	000
K	I	R	K	'	S	_	D	I	K	D	I	K

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

```
010001110
```

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

```
010|001|110
```

character	*code*
K	000
I	001
R	010
’	011
S	100
_	101
D	110

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

010|001|110

R

character	*code*
K	000
I	001
R	010
’	011
S	100
_	101
D	110

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

```
010|001|110
 R   I
```

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

```
010|001|110
 R   I   D
```

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

A Different Encoding

What is the mystery word represented by this 3-bit encoding?

```
010001110
```

RID

character	*code*
K	000
I	001
R	010
’	011
S	100
_	101
D	110

A Different Encoding

- If we're specifically writing the string **KIRK'S DIKDIK**, which only has seven different characters, using full bytes is wasteful
- Let's use a 3-bit encoding instead
- This uses 37.5% of the space as what ASCII uses!

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

000	001	001	000	011	100	101	101	001	000	101	001	000
K	I	R	K	'	S	_	D	I	K	D	I	K

The Journey Ahead

- Storing data using the ASCII encoding is portable across systems, but is not ideal in terms of space usage
- Building custom codes for specific strings might let us save space
- Idea: Use this approach to build a **compression algorithm** to reduce the amount of space needed to store text
- We want to find a way to
  give all characters a bit pattern,
  that both the sender and receiver know about, and
  that can be decoded uniquely

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that have widespread prevalence and importance
- We're interested in algorithms that provide lossless compression on a stream of characters or other data
  - We make the amount of data smaller without losing any of the details, and we can decompress the data o exactly the same as it was before compression
- Virtually everything you do online involves data compression
  - When you visit a website, download a file, or transmit video/audio, the data is compressed when sending and decompressed when receiving
  - A video stream on Zoom has a compression of roughly 2000:1, meaning that a 2MB image is compressed down to just 1000 bytes
- Compression algorithms identify patterns in data and take advantage of those to come up with more efficient representations of that data

A Different Encoding

- Let's make this encoding even more efficient!

character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

000	001	001	000	011	100	101	101	001	000	101	001	000
K	I	R	K	'	S	_	D	I	K	D	I	K

Take Advantage of Redundancy

- Not all letters have the same frequency in **`KIRK'S DIKDIK`**
- We can calculate the frequencies of each letter

character	*frequency*
K	4
I	3
R	1
'	1
S	1
_	1
D	2

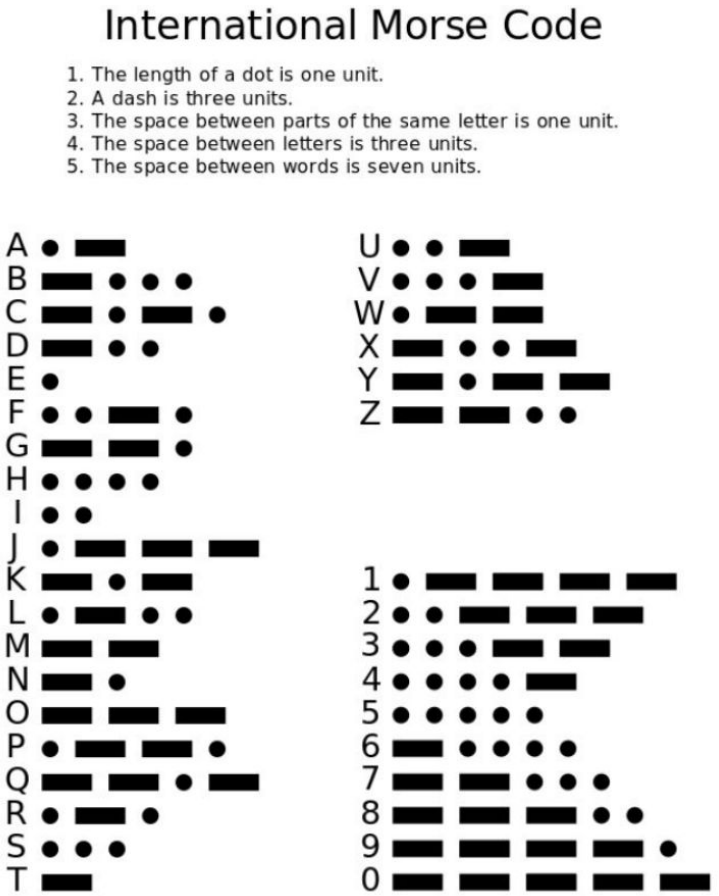
Take Advantage of Redundancy

- Not all letters have the same frequency in **`KIRK'S DIKDIK`**
- We can calculate the frequencies of each letter
- So far, we've given each letter a code of the same length
- Maybe we can give shorter encodings to more frequent letters to save space?

character	*frequency*
K	4
I	3
R	1
'	1
S	1
_	1
D	2

Morse Code

- Morse code is an example of a coding system that makes use of this insight
- The codes for frequent letters (ex: e, t, a) are much shorter than the codes for infrequent letters (ex: q, y, j)



Our New Code

KIRK’S DIKDIK

01010101110000100010

character	*frequency*	*code*
K	4	0
I	3	1
D	2	00
R	1	01
’	1	10
S	1	11
_	1	100

0	1	01	0	10	11	100	00	1	0	00	1	0
K	I	R	K	’	S	_	D	I	K	D	I	K

Our New Code

What is the mystery word represented by this encoding?

```
0110011
```

character	*frequency*	*code*
K	4	0
I	3	1
D	2	00
R	1	01
’	1	10
S	1	11
_	1	100

Our New Code

KIRK’S DIKDIK

01010101110000100010

character	*frequency*	*code*
K	4	0
I	3	1
D	2	00
R	1	01
’	1	10
S	1	11
_	1	100

Our New Code

KIRK'S_DIKDIK

01010101110000100010

RRRRI_KK'D'

character	*frequency*	*code*
K	4	0
I	3	1
D	2	00
R	1	01
'	1	10
S	1	11
_	1	100

What went wrong?

- If we use a different number of bits for each letter, we can't necessarily uniquely determine the boundaries between letters
- We need an encoding that makes it possible to determine where one characters ends and the next begins
  - Codes for each character need to be unique and unambiguous
- How can we do this?

Prefix Code

- A prefix code is an encoding system in which no code is a prefix of another code
- Here's a sample prefix code for the letters in **KIRK'S_DIKDIK**

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Prefix Code

KIRK'S_DIKDIK

10010011000011011100111011011110110

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Prefix Code

What is the mystery word represented by this encoding?

```
001011111101
```

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

001011111101

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

001011111101

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

001011111101

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

001|011111101

R

character	*code*
K	**10**
I	**01**
D	**111**
R	**001**
'	**000**
S	**1101**
_	**1100**

Prefix Code

What is the mystery word represented by this encoding?

```
001|011111101
 R
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01111101
 R
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|1111101
 R   I
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|1111101
 R   I
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|11111101
 R   I
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|1111101
 R   I
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I    D
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I    D
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I    D
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I    D
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I    D
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001|01|111|1101
 R   I   D   S
```

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

What is the mystery word represented by this encoding?

```
001011111101
```

RIDS

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Prefix Code

- A prefix code is an encoding system in which no code is a prefix of any other code
- Here's a sample encoding of **KIRK'S_DIDIK**
- This uses just 34 bits, compared to 104 with ASCII (32.7% of the space)

Where did this code come from?

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Prefix Code

- A prefix code is an encoding system in which no code is a prefix of any other code.
- Here's a sample encoding for the string **KIRK'S_DIDIK**
- This uses just 34 bits, compared to 104 with ASCII (32.7% of the space)

How can we come up with codes like this for other strings?

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Prefix Code

- A prefix code is an encoding system in which no code is a pref
- Here's a samp **KIRK'S_DI**
- This uses just 34 bits, compared to 104 with ASCII (32.7% of the space)

What makes a "good" prefix coding scheme?

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Prefix Code

- A prefix code is an encoding system in which no code is a prefix
- Here's a sample: **KIRK'S_DIRK**
- This uses just 34 bits, compared to 104 with ASCII (32.7% of the space)

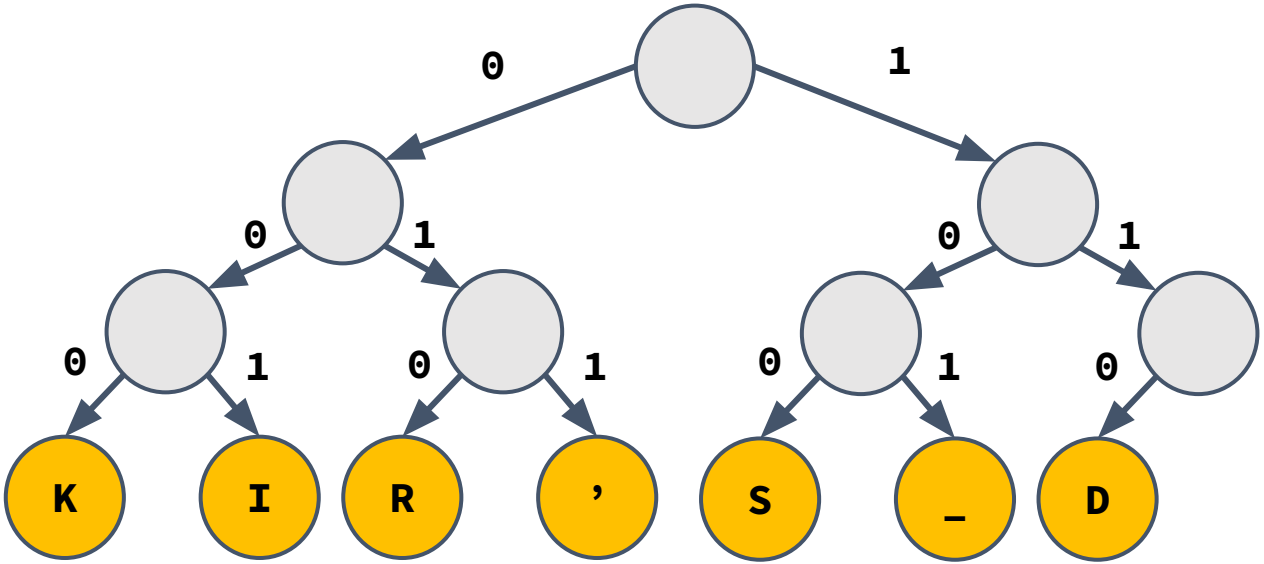
What does this have to do with trees?

character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
_	1100

10	01	001	10	000	1101	1100	111	01	10	111	01	10
K	I	R	K	'	S	_	D	I	K	D	I	K

Coding Tree

- We can represent a prefix coding scheme using a binary tree, which is called a coding tree

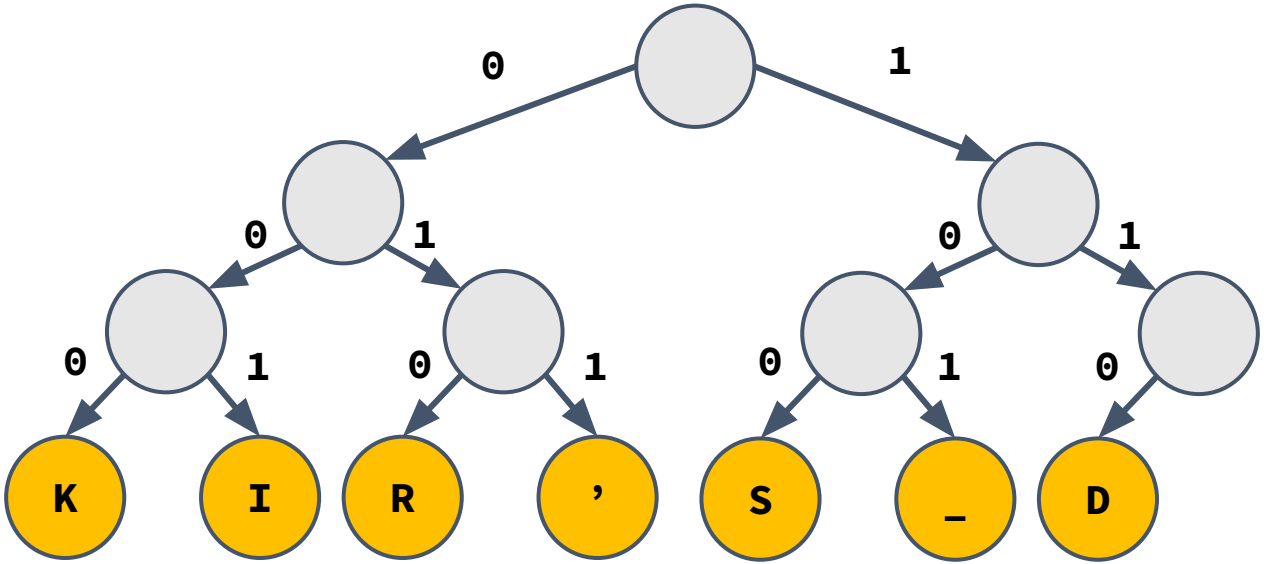


character	*code*
K	000
I	001
R	010
'	011
S	100
_	101
D	110

Coding Tree

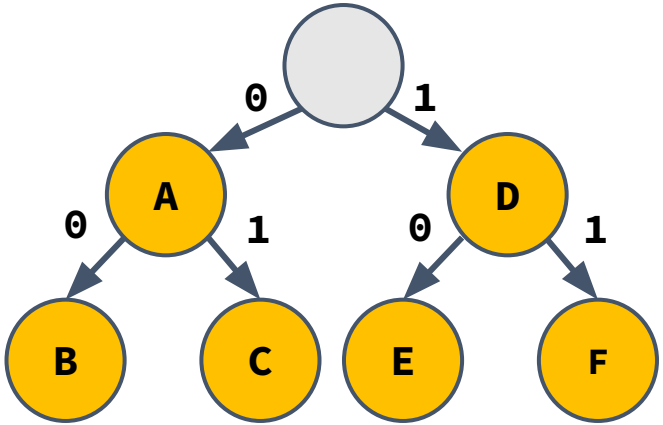
What is the mystery word represented by this encoding?

```
110001010
```



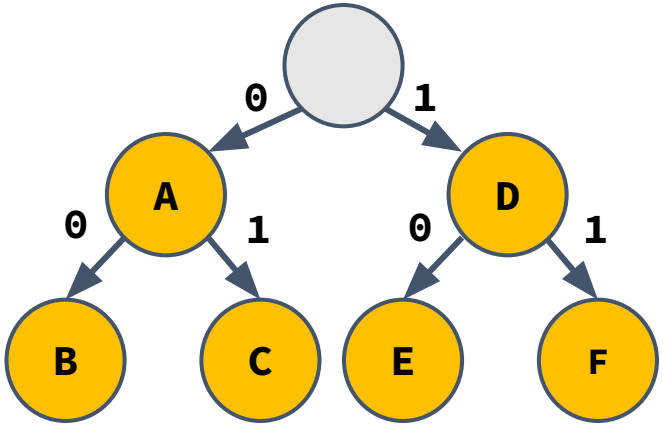
Coding Trees

- Not all binary trees work as coding trees
- Why is this binary tree not a coding tree?



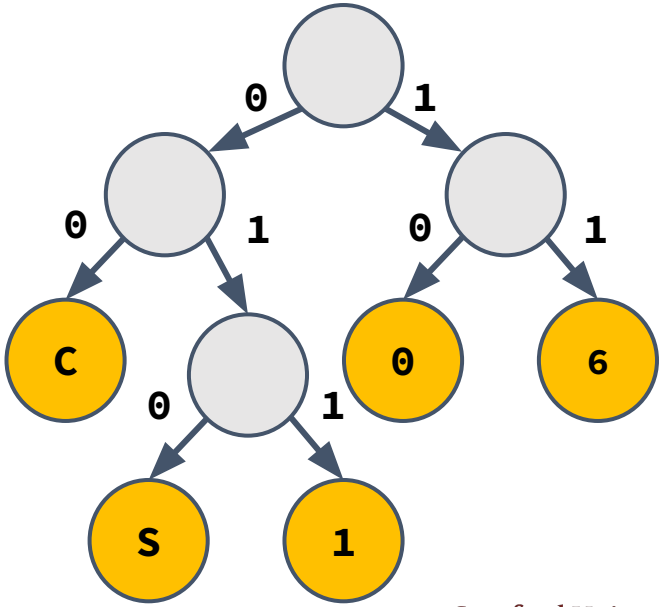
Coding Trees

- Not all binary trees work as coding trees
- Why is this binary tree not a coding tree?
  - Doesn't give a prefix code!
  - The code for **A** is a prefix for the codes for **B** and **C**, and the code for **D** is a prefix for the codes for **E** and **F**



Coding Trees

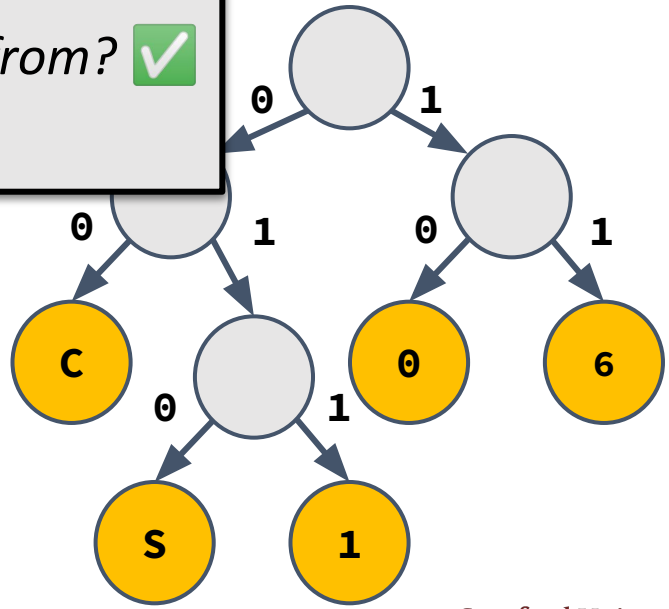
- A coding tree is valid if all the letters are stored in the **leaves**, with internal nodes only used for routing



Coding Trees

- A coding tree is valid if all the letters are stored in the [illegible] only used for

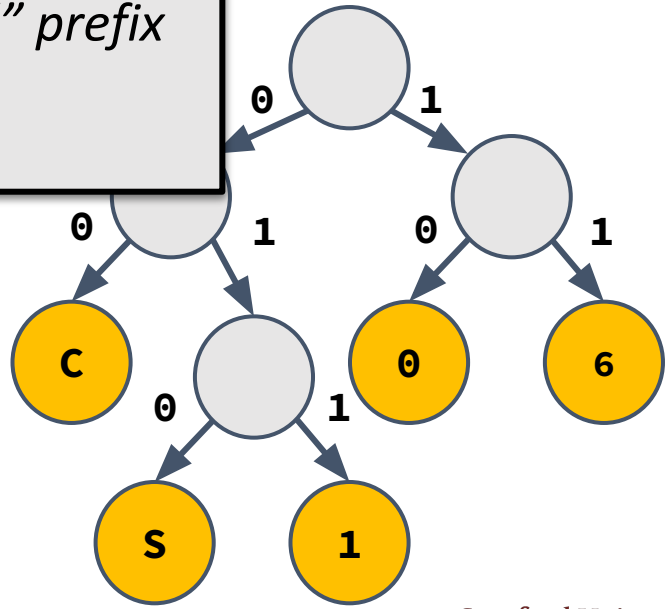
Where did this code come from? ✅



Coding Trees

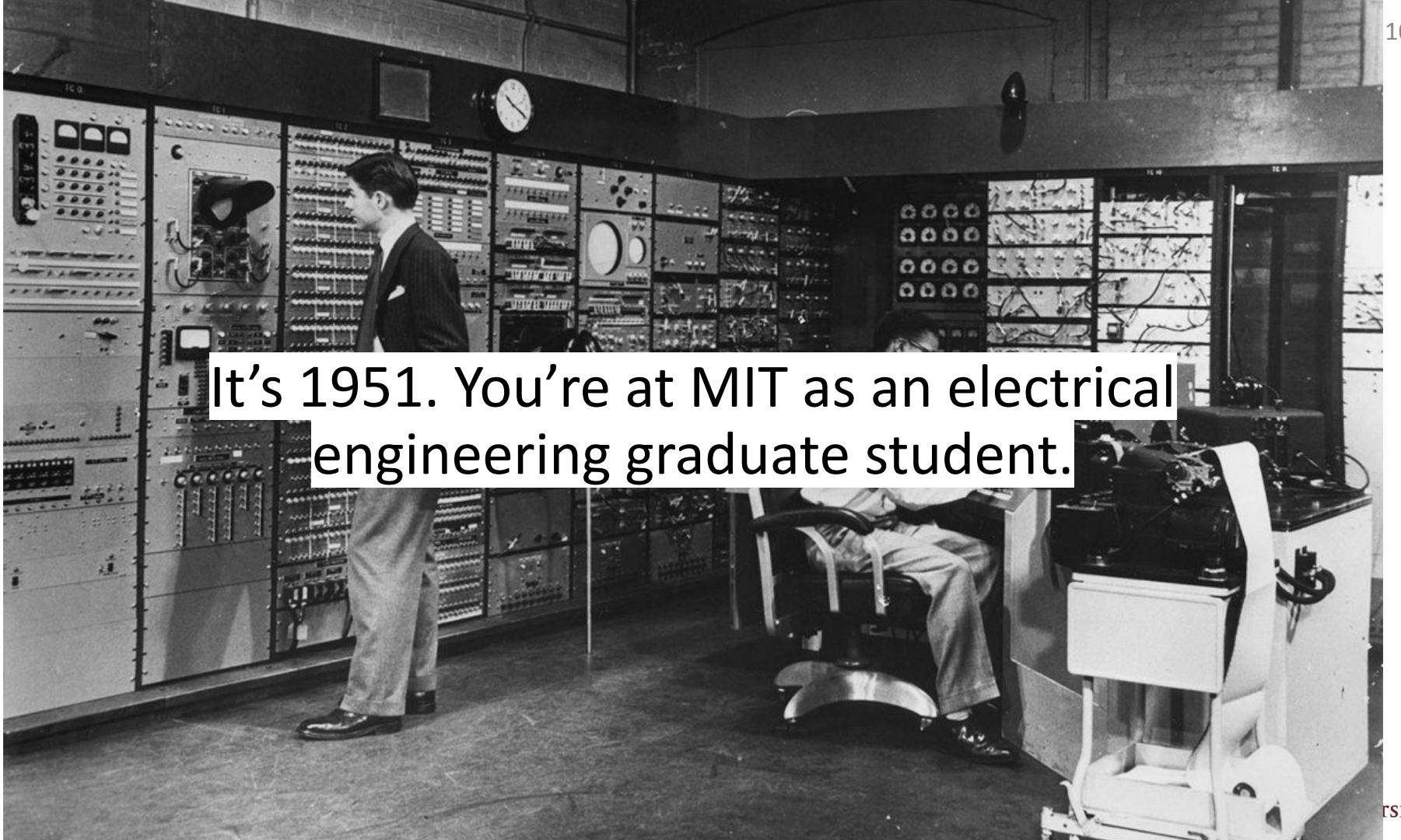
- A coding tree is valid if all the letters are stored in the [illegible] only used for

How do we make a "good" prefix coding scheme?



Huffman Coding

It’s 1951. You’re at MIT as an electrical engineering graduate student.



It’s 1951. You’re at MIT as an electrical engineering graduate student.

You have a choice for your class: take the final exam or write a term paper

You choose to write the term paper.
The prompt is: “Find the most efficient method of representing numbers, letters, or symbols using binary code”

David Huffman tries to solve this problem for **months**.

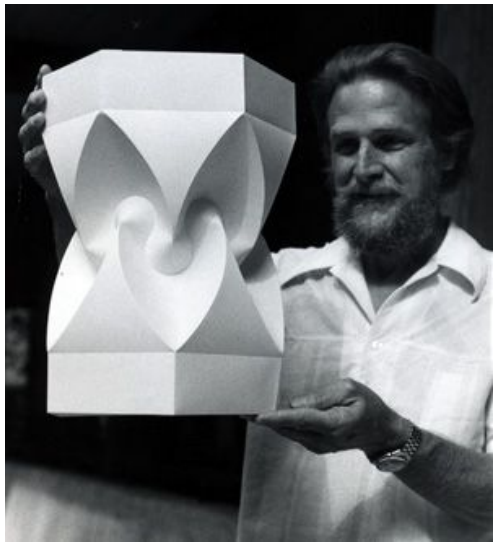
It’s 1951, so no Google or StackOverflow.

Important note:
Neither his professor, Robert M. Fano, nor the inventor of information theory, Claude Shannon, had any idea how to solve it

So David Huffman gives up, and starts studying for the final exam instead.

But then, epiphany!

"It was my luck to be there at the right time and also not have my professor discourage me by telling me that other good people had struggled with his problem."



Link to full story

The Algorithm

Huffman Coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies
- Different data (different text, different images, etc) will each have their own personalized Huffman coding tree
- We want an encoding tree that
  - Allows for variable length codes (so most frequent characters can get shorter codes, aka their leaf nodes are closer to the root node)
  - Represents a prefix code system (no ambiguity in when characters stop and start)

Goal: Build the optimal encoding tree for

KIRK'S DIKDIK

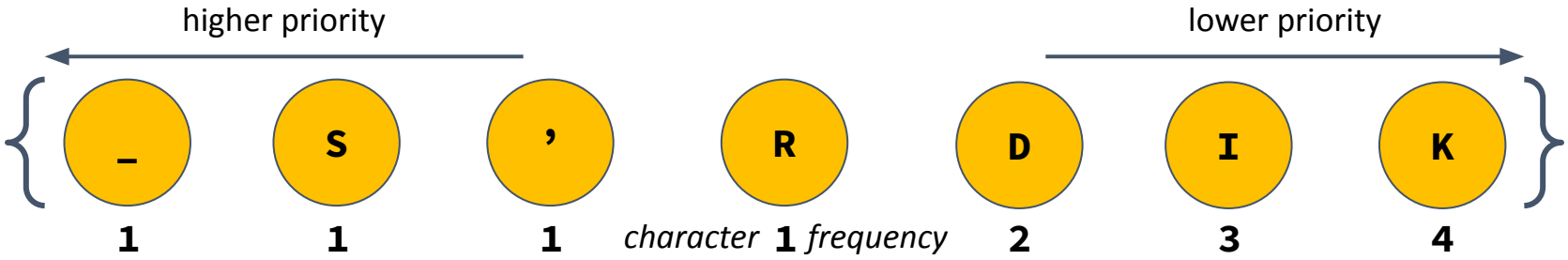
1. Build the frequency table

Input text: **KIRK'S DIKDIK**

character	*frequency*
K	4
I	3
R	1
'	1
S	1
_	1
D	2

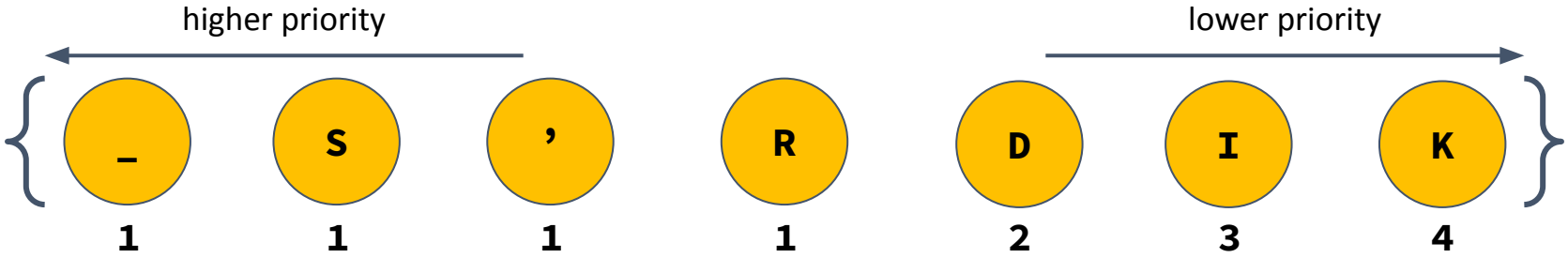
2. Initialize an empty priority queue

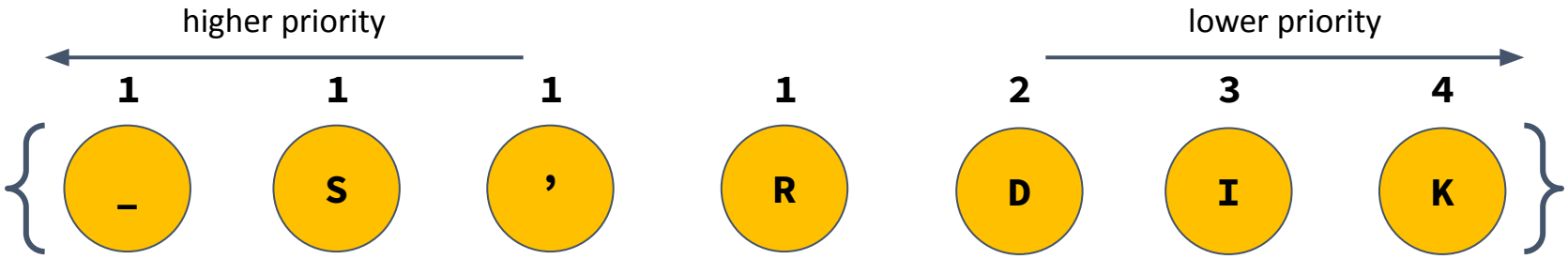
3. Add all unique characters as leaf nodes to queue

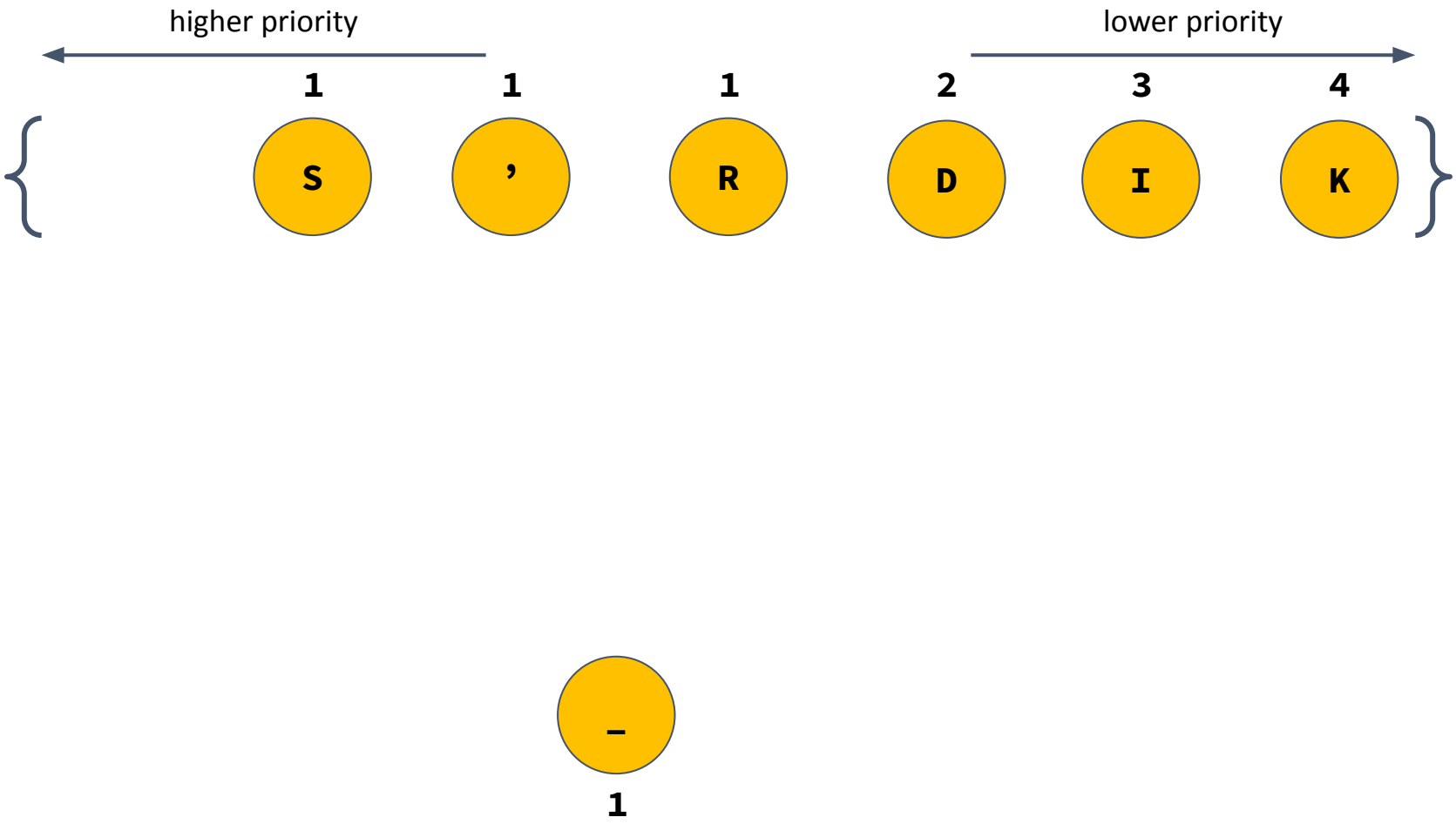


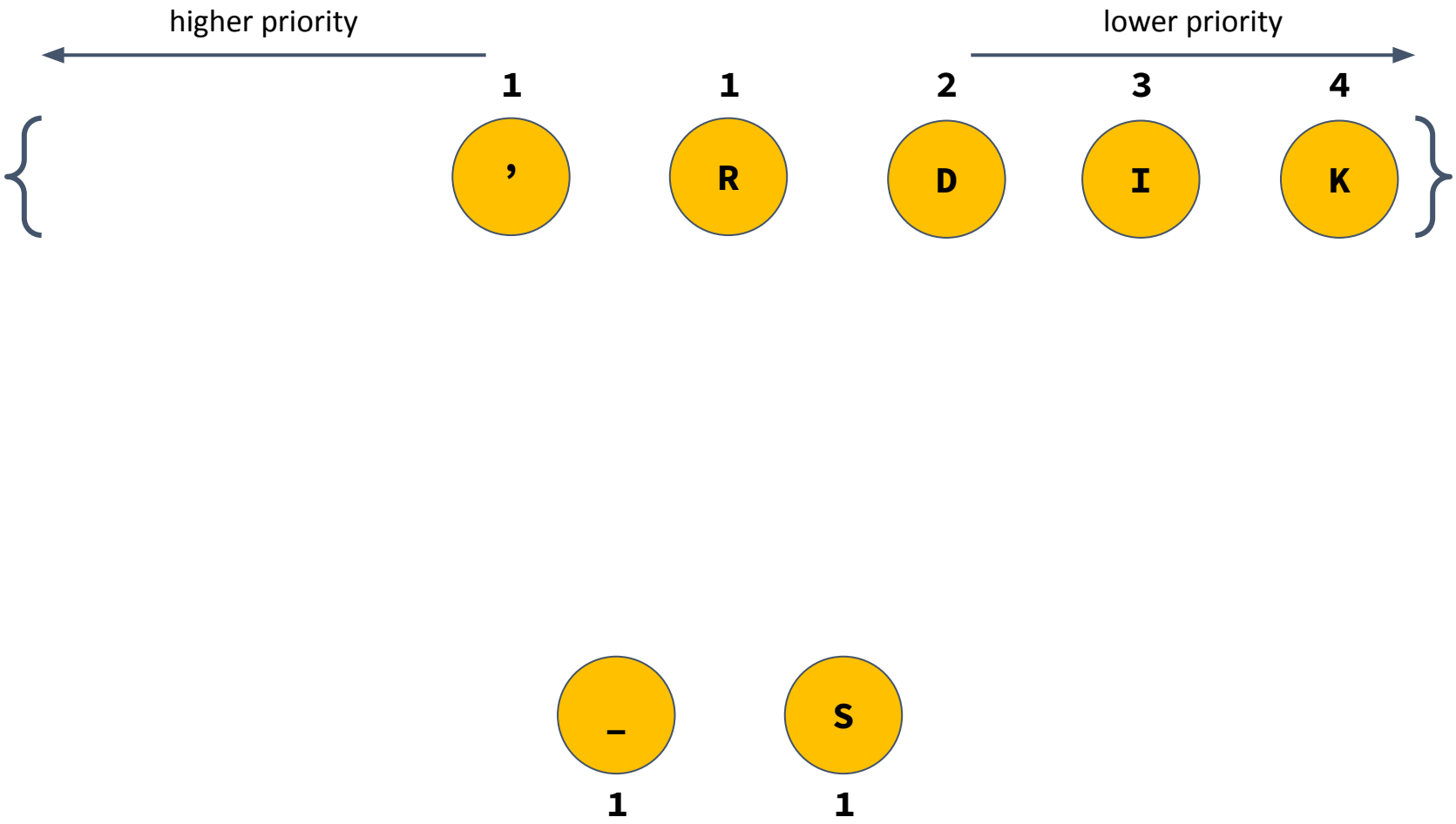
character	frequency
K	4
I	3
R	1
'	1
S	1
_	1
D	2

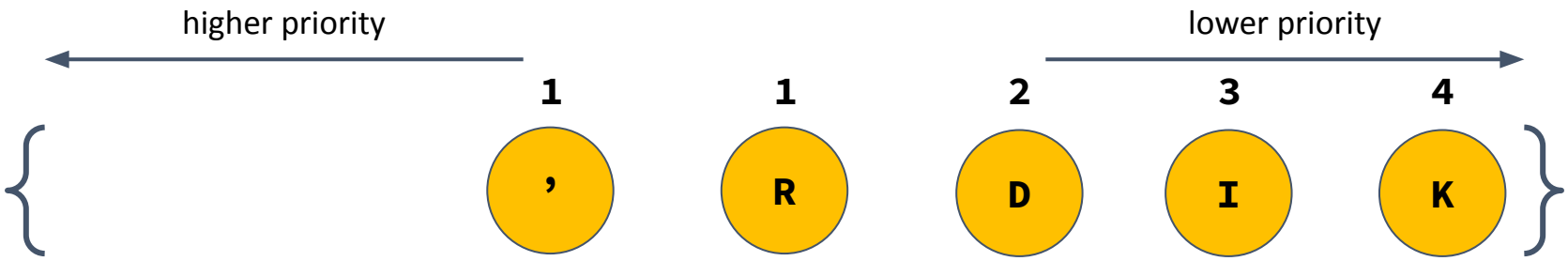
4. Build the Huffman tree by merging nodes

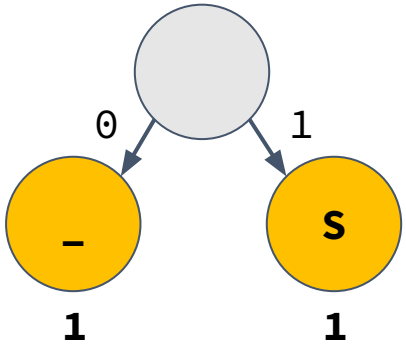


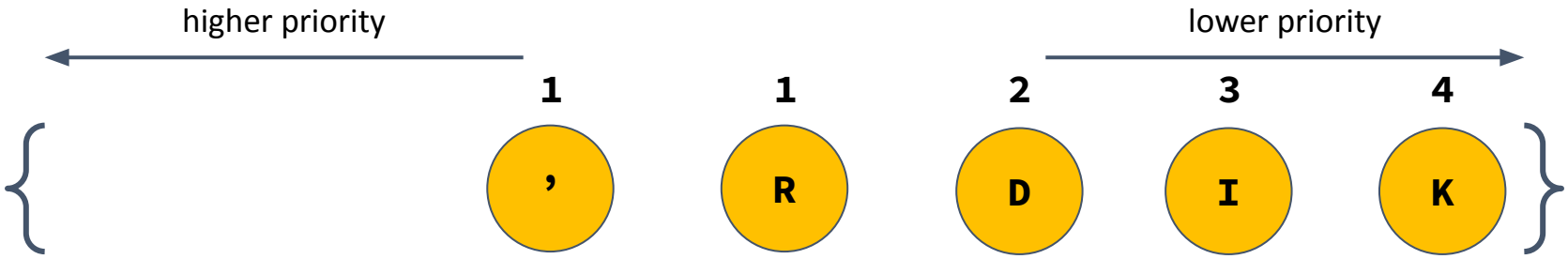

higher priority
lower priority
1
1
1
1
2
3
4
_
S
'
R
D
I
K

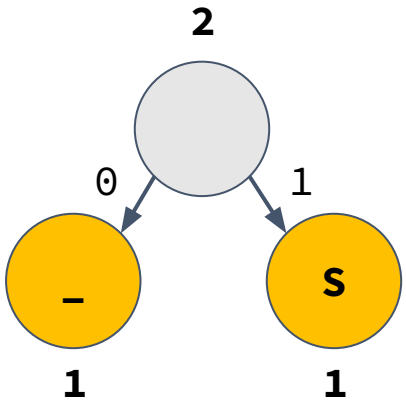

higher priority
lower priority
1
1
1
2
3
4
S
'
R
D
I
K
_
1


higher priority
lower priority
1
1
2
3
4
,
R
D
I
K
_
S
1
1

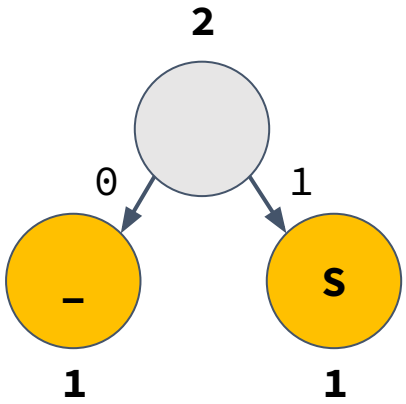

higher priority
lower priority
1
1
2
3
4
,
R
D
I
K

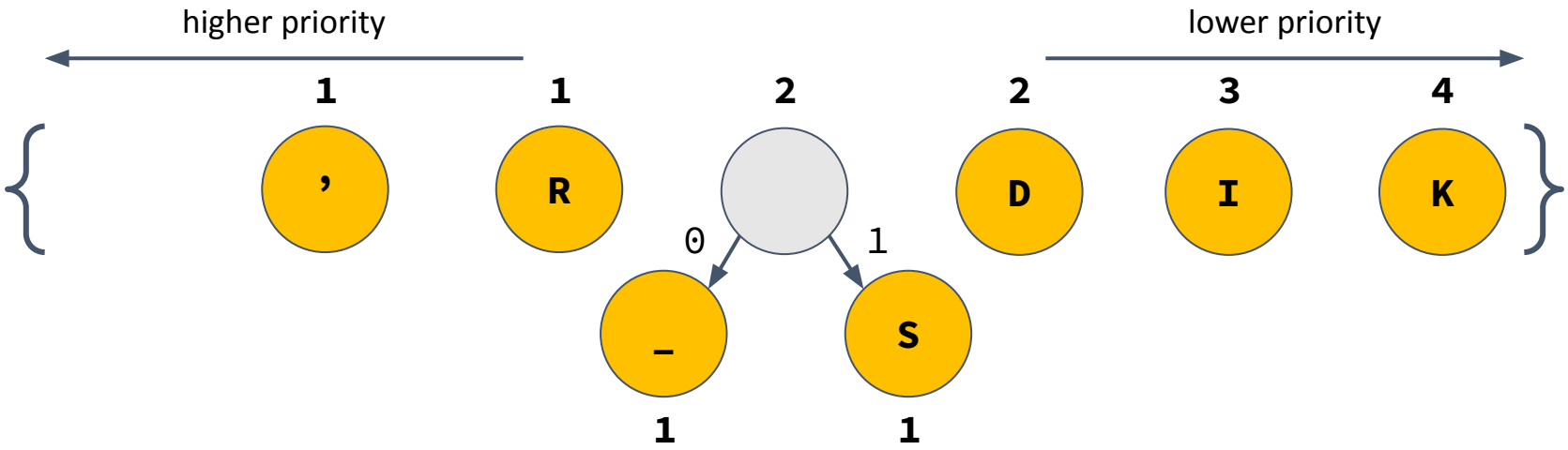

0
1
_
S
1
1

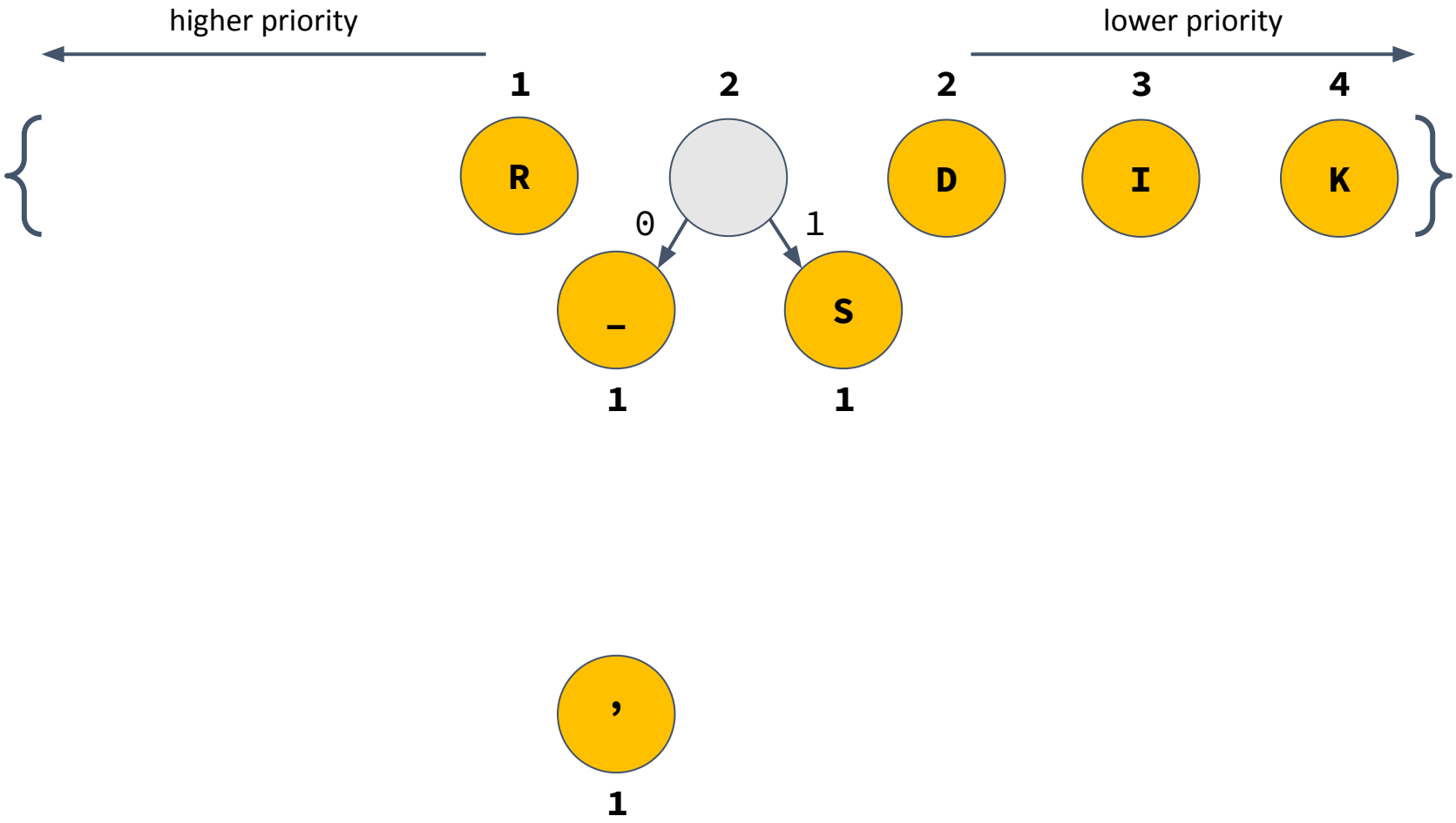

higher priority
lower priority
1
1
2
3
4
,
R
D
I
K

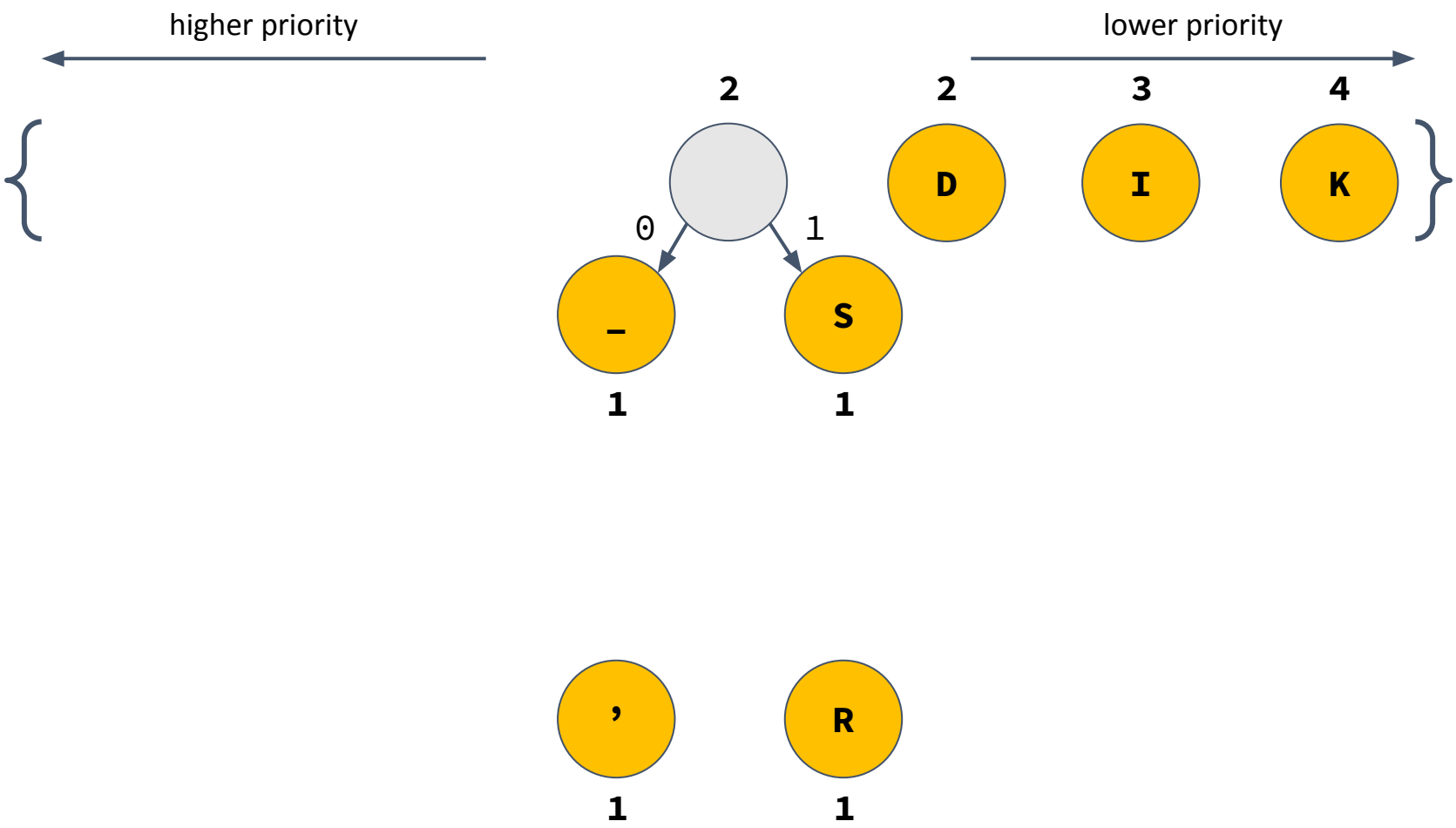

2
0
1
_
S
1
1

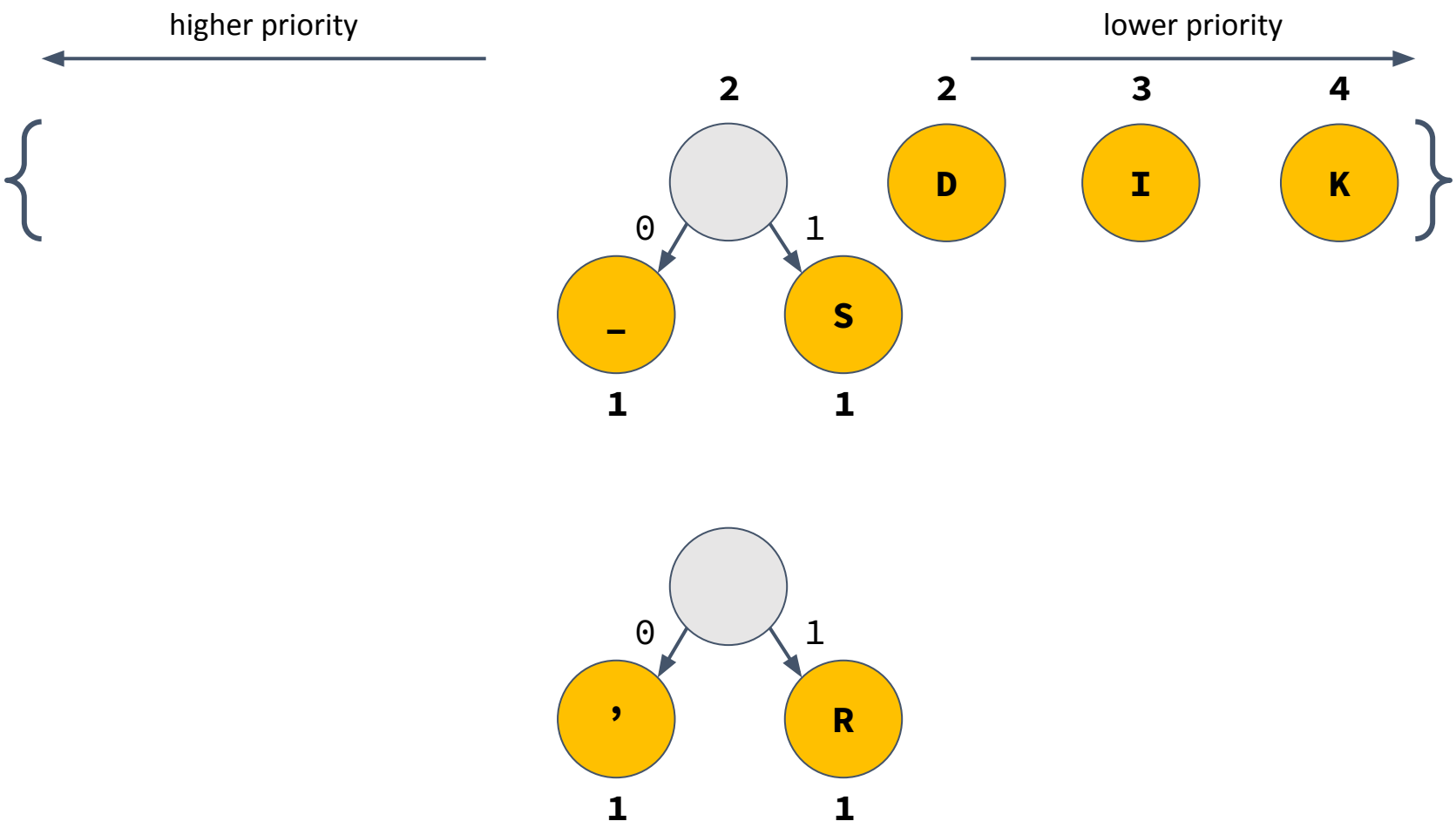

higher priority
lower priority
1
1
2
3
4
,
R
D
I
K

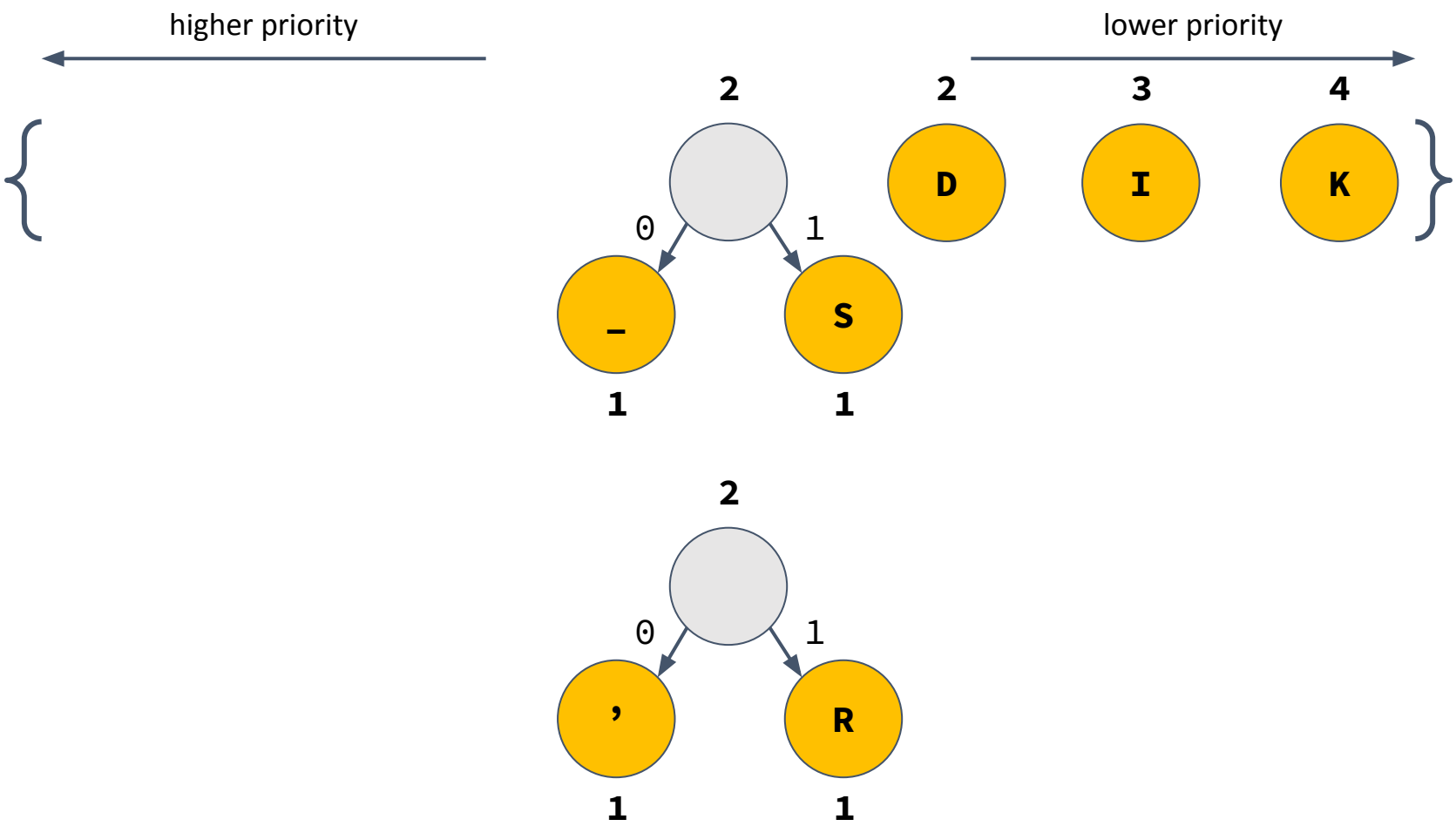

2
0
1
_
S
1
1

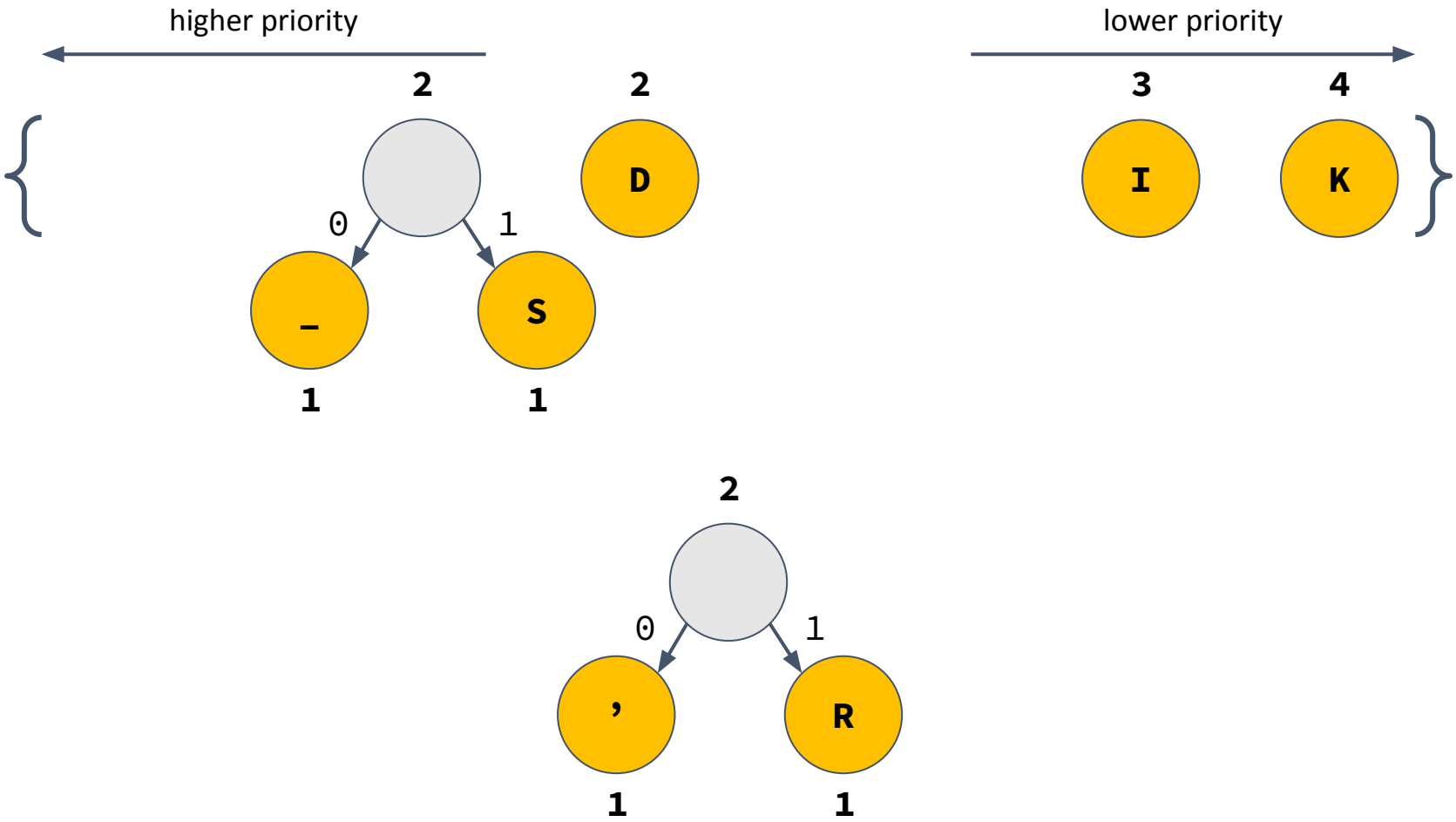

higher priority
lower priority
1
1
2
2
3
4
'
R
D
I
K
0
1
_
S
1
1

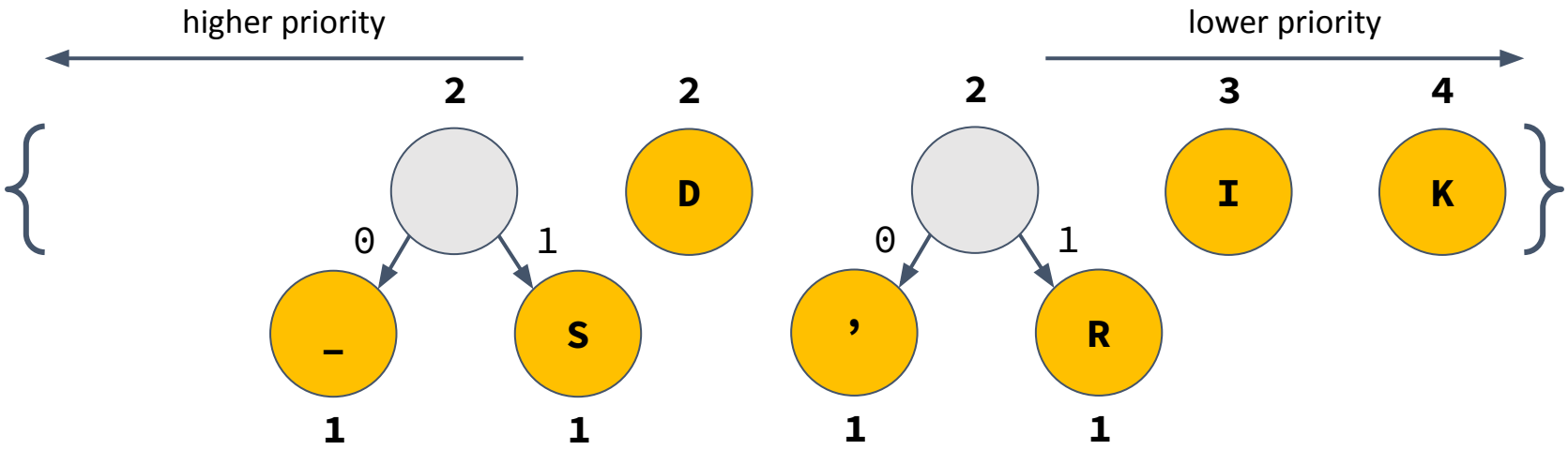

higher priority
lower priority
1
2
2
3
4
R
D
I
K
0
1
_
S
1
1
,
1

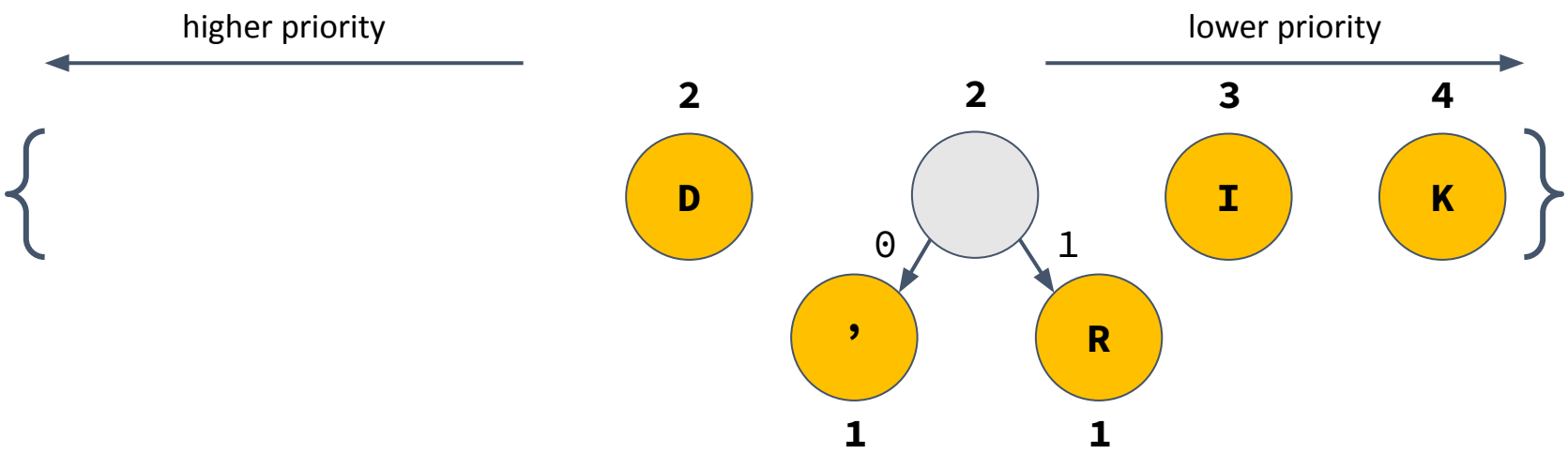

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
,
R
1
1

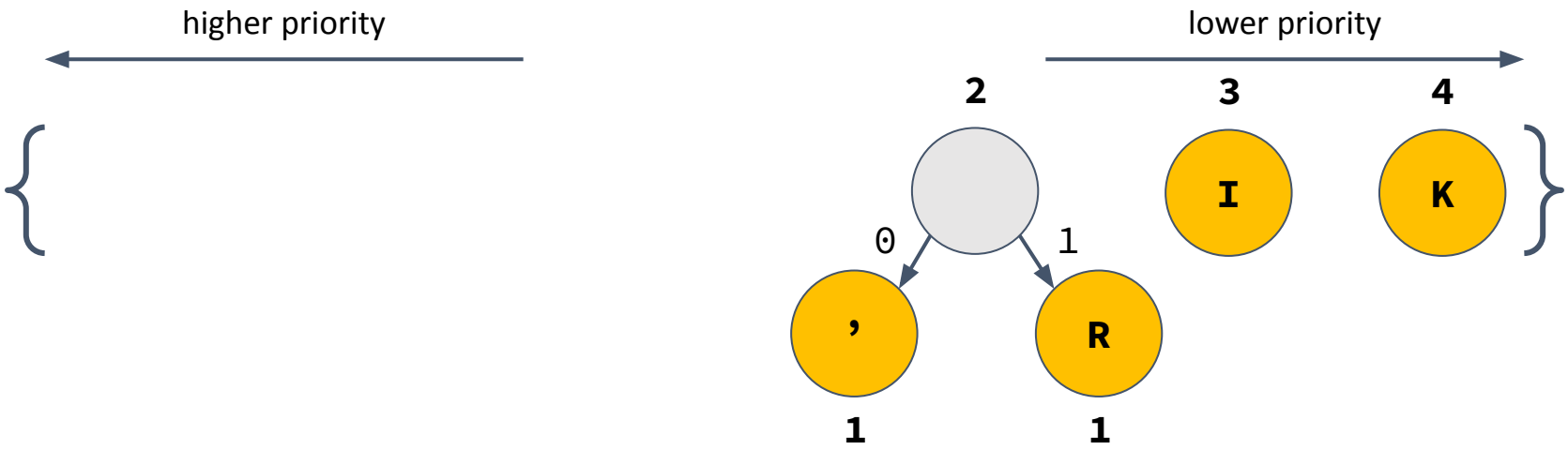

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
0
1
,
R
1
1

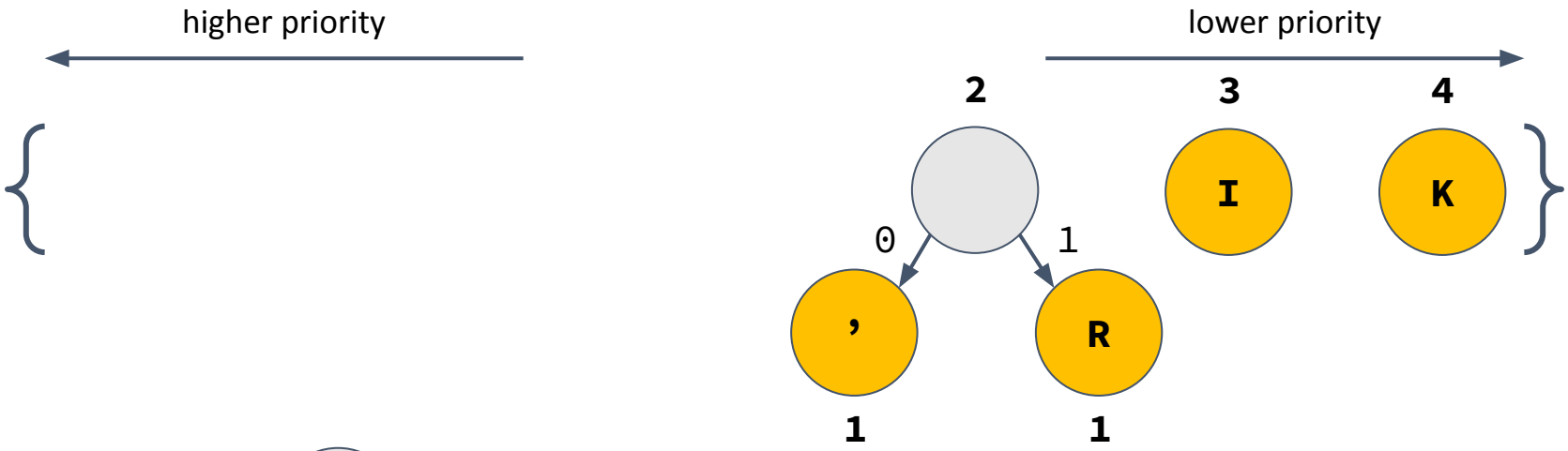

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
2
0
1
,
R
1
1

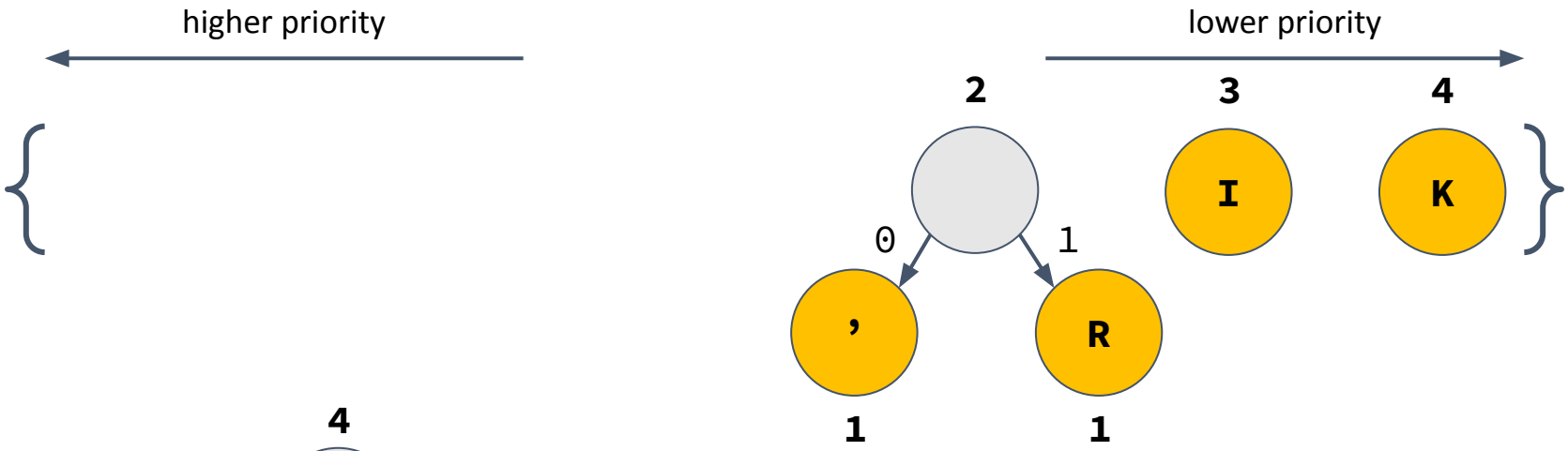

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
2
0
1
,
R
1
1

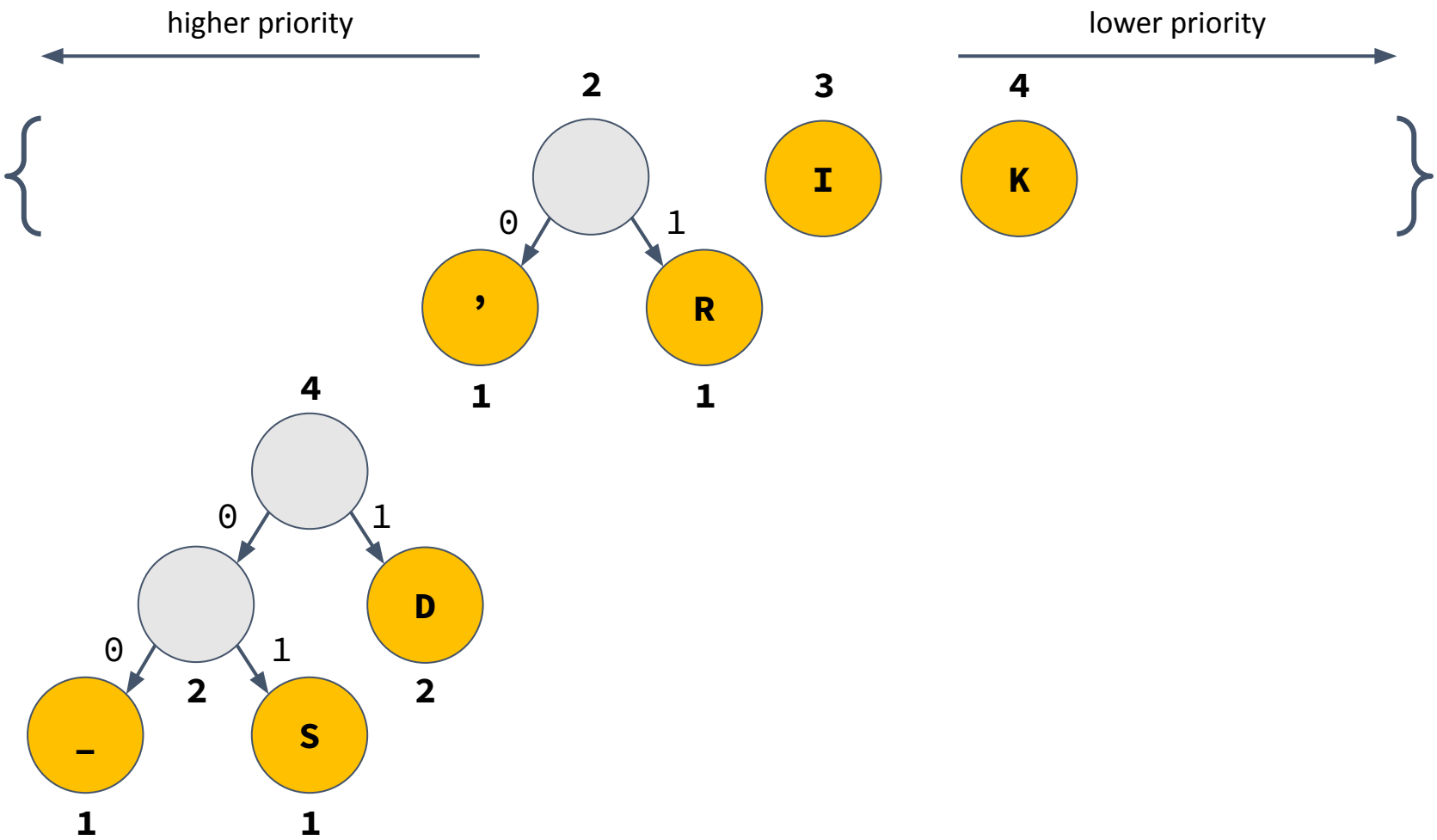

higher priority
lower priority
2
2
2
3
4
0
1
0
1
D
I
K
_
S
'
R
1
1
1
1

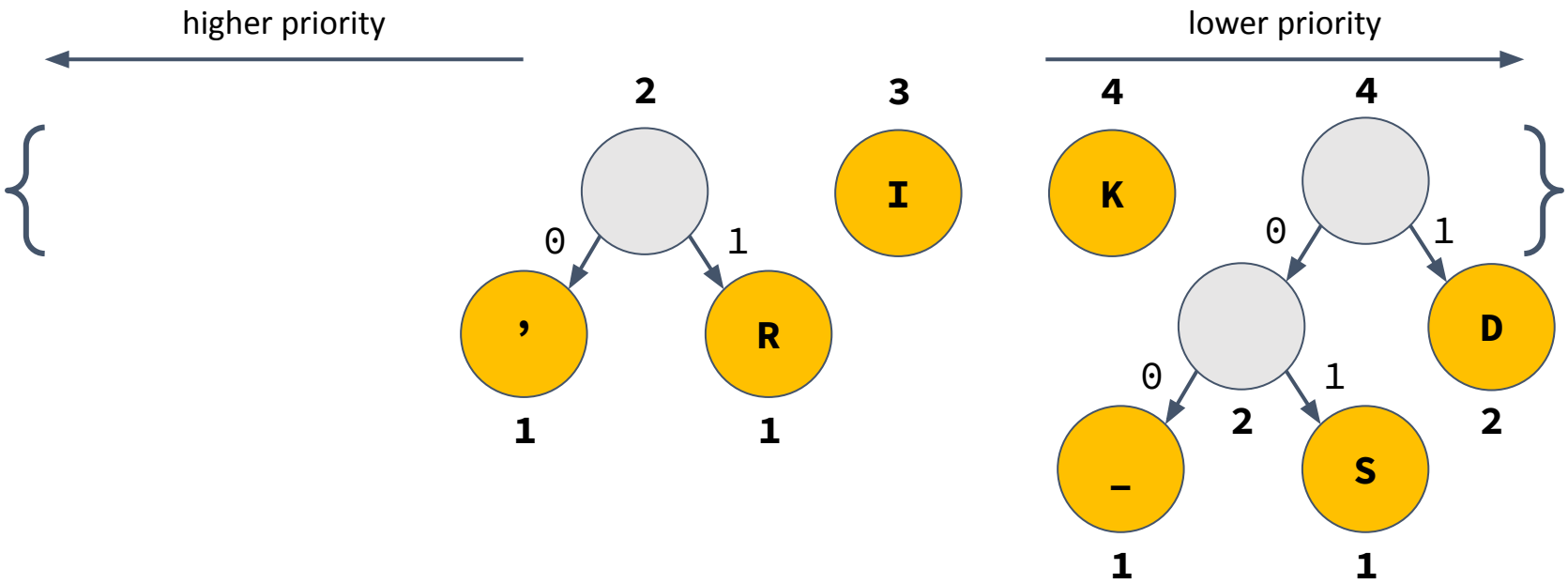

higher priority
lower priority
2
2
3
4
D
I
K
0
1
,
R
1
1
0
1
2
_
S
1
1

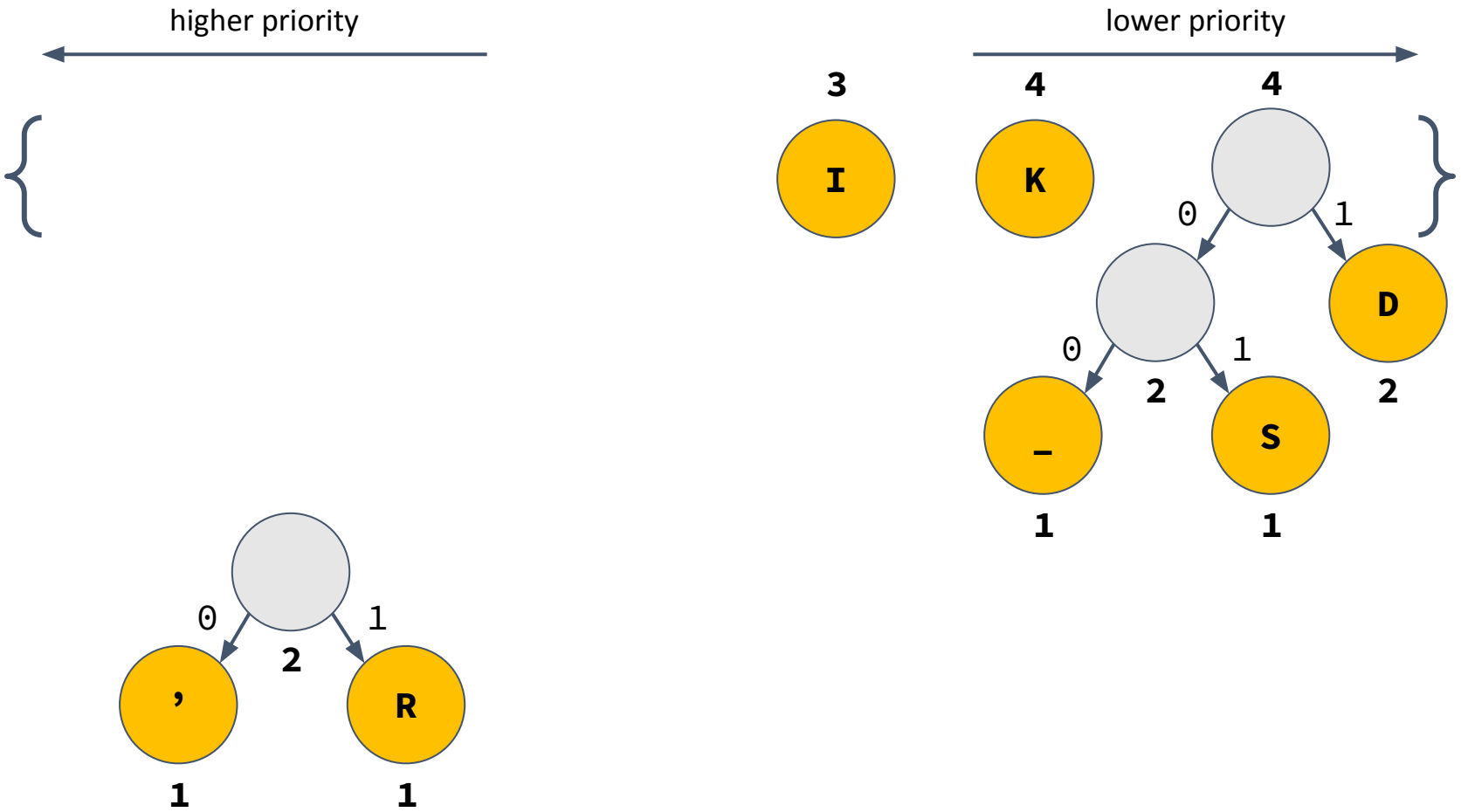

higher priority
lower priority
2
3
4
I
K
0
1
,
R
1
1
0
1
2
2
D
2
_
S
1
1


higher priority
lower priority
2
3
4
I
K
0
1
,
R
1
1
0
1
D
0
1
2
2
_
S
1
1


higher priority
lower priority
2
3
4
I
K
0
1
,
R
1
1
4
0
1
D
0
1
2
2
_
S
1
1

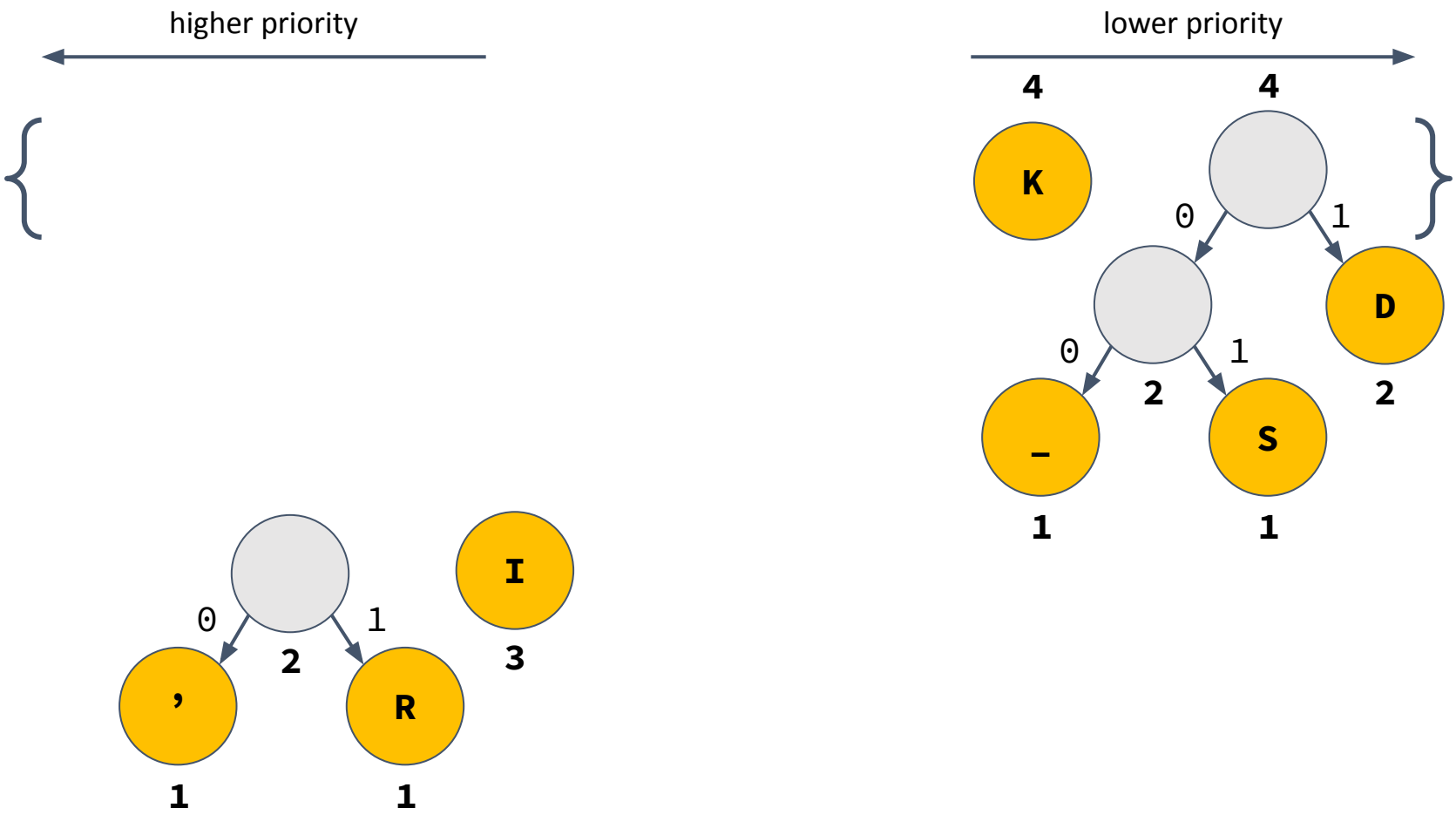

higher priority
lower priority
2
3
4
I
K
0
1
,
R
1
1
4
0
1
D
0
1
2
2
_
S
1
1


higher priority
lower priority
2
3
4
4
I
K
0
1
0
1
,
R
D
0
1
2
2
_
S
1
1
1
1

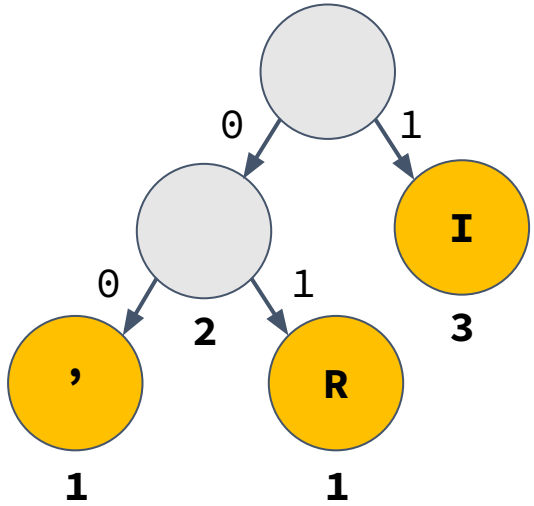

higher priority
lower priority
3
I
4
K
4
0
1
D
2
0
1
2
_
1
S
1
0
1
2
,
1
R
1

higher priority

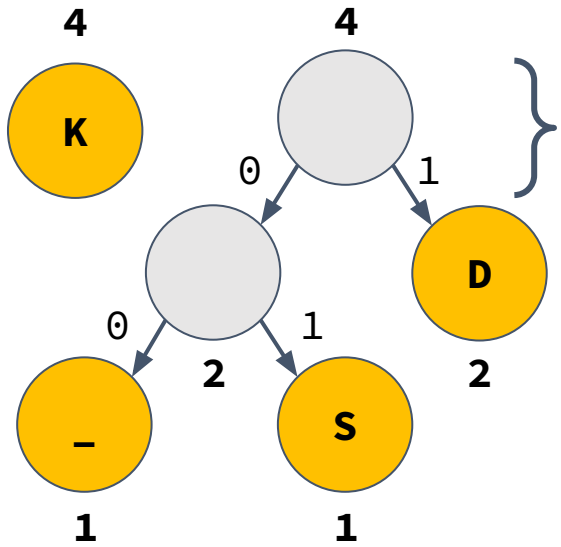
lower priority



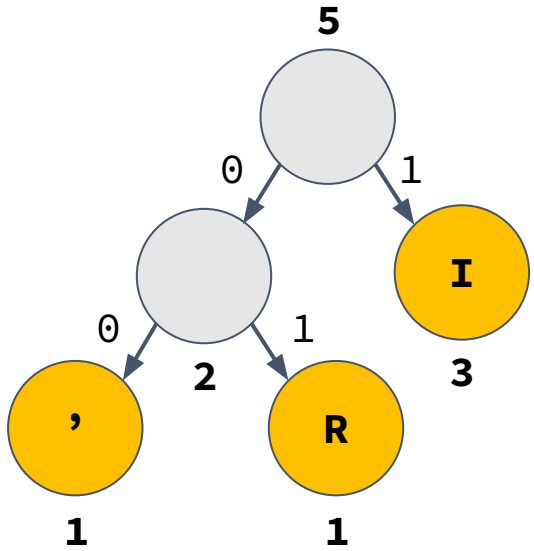
higher priority



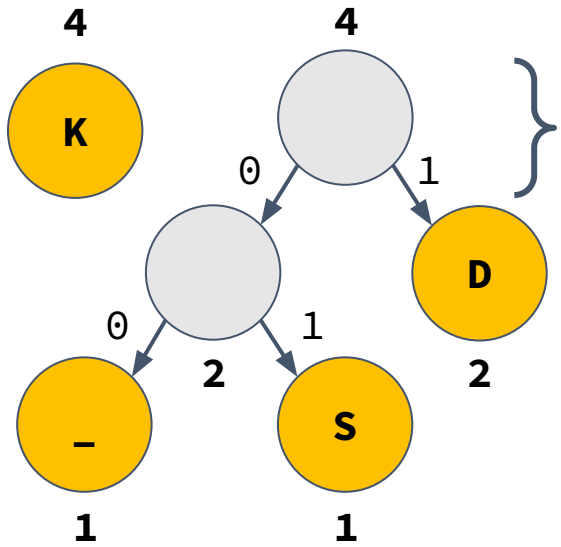
lower priority

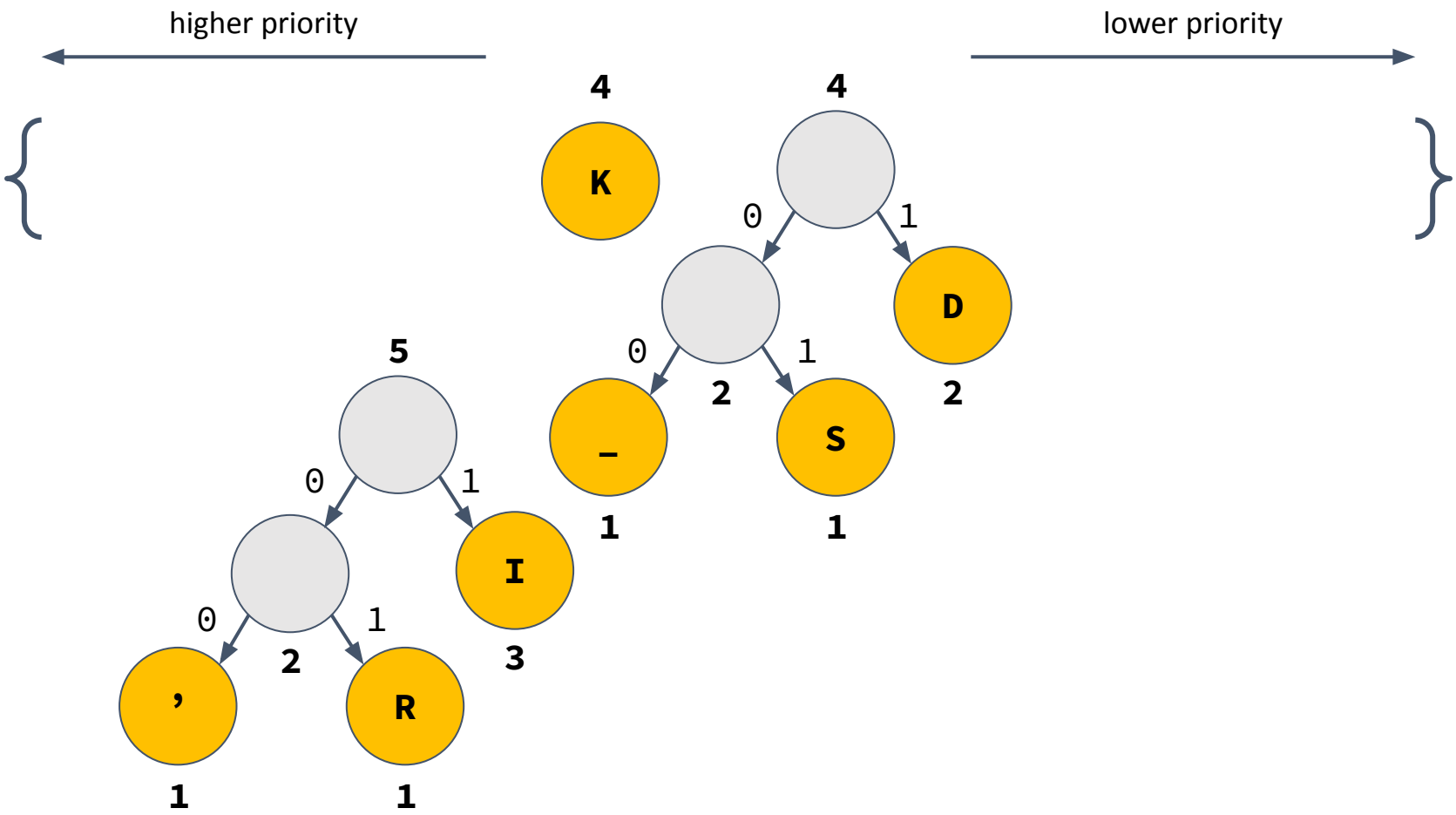


higher priority



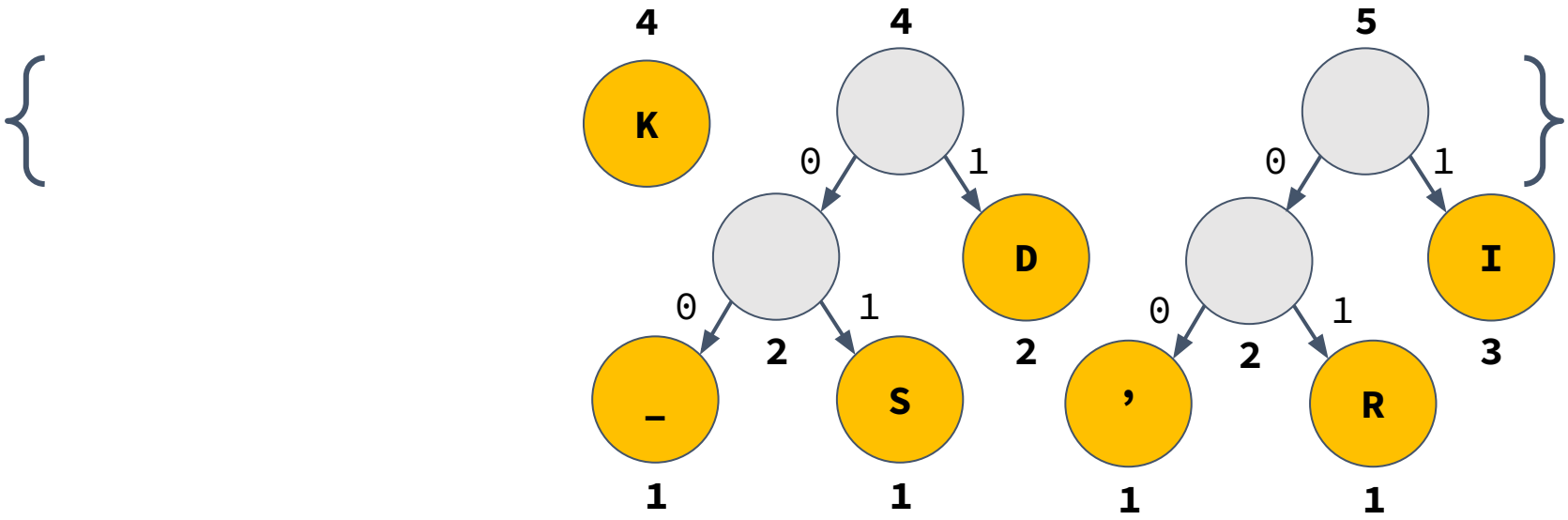
lower priority

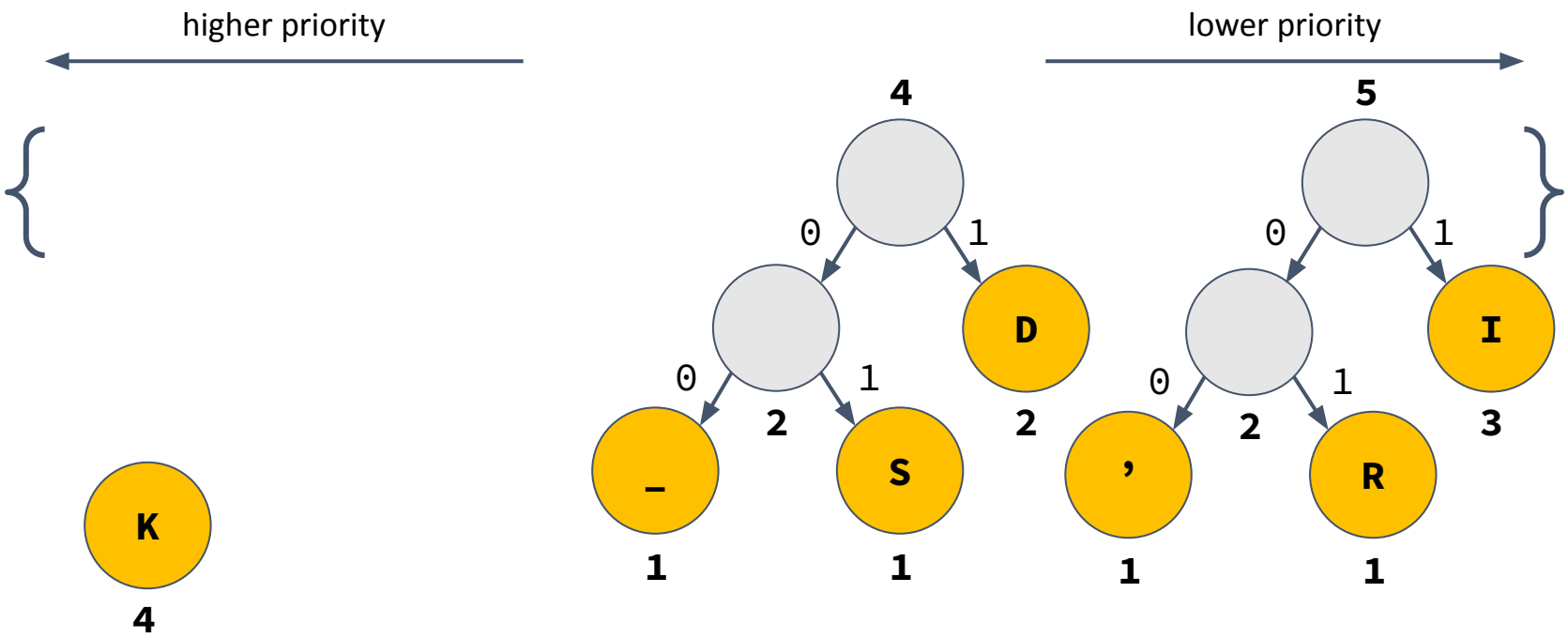



higher priority
lower priority
4
K
4
0
1
D
2
0
1
2
_
1
S
1
5
0
1
I
3
0
1
2
'
1
R
1

higher priority

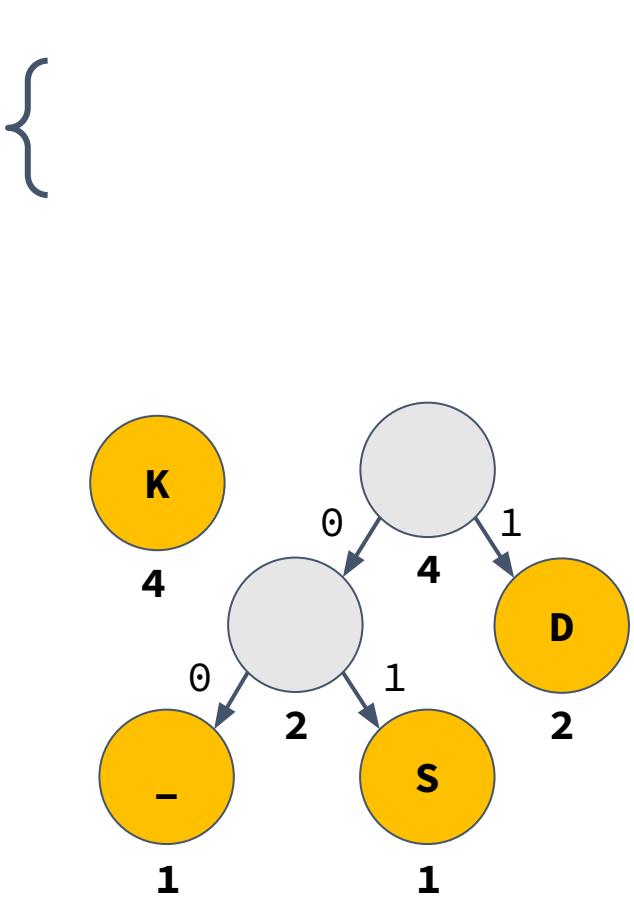
lower priority

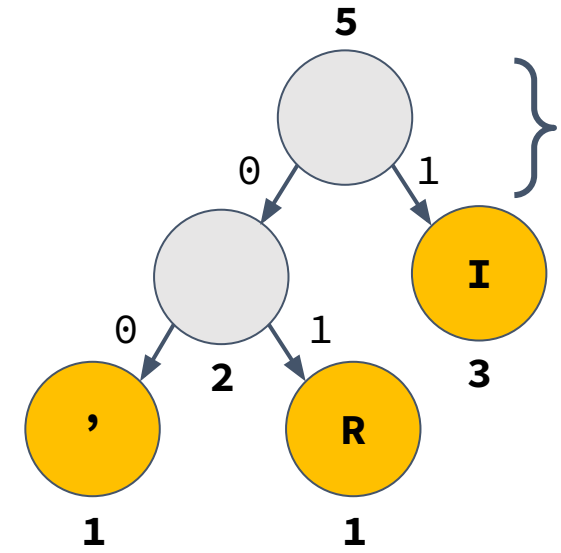



higher priority
lower priority
K
4
4
0
1
_
1
2
0
1
S
1
D
2
5
0
1
'
1
2
0
1
R
1
I
3

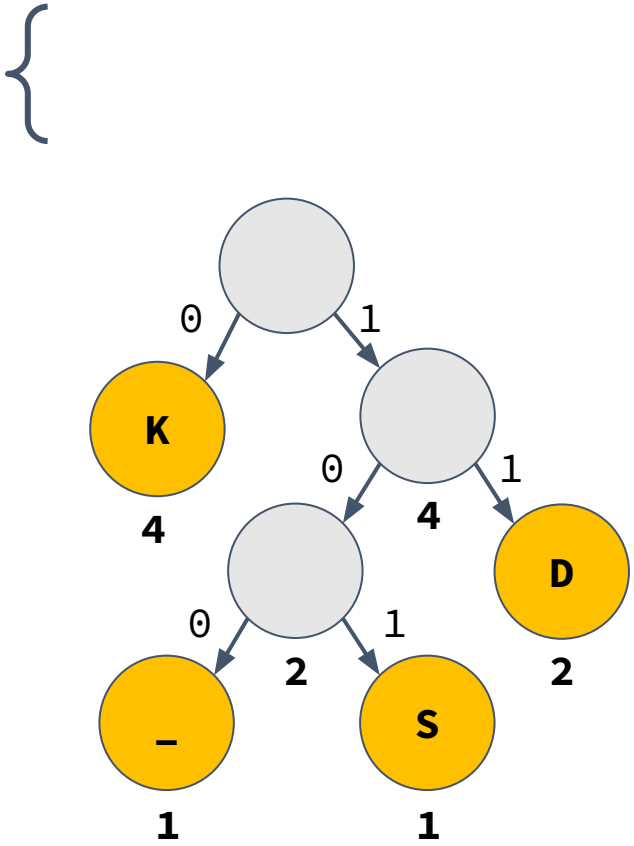
higher priority

lower priority

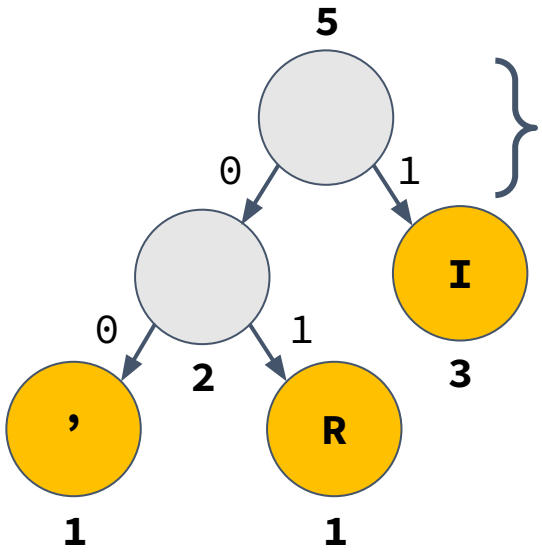




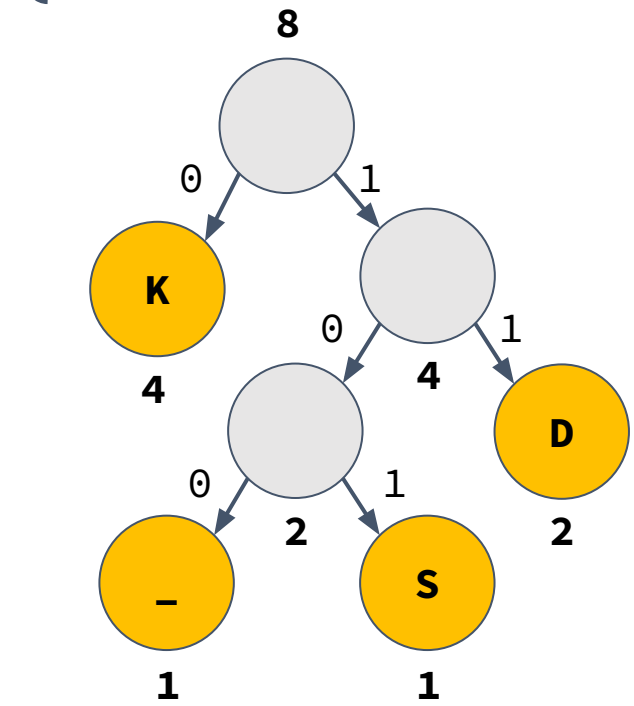
higher priority



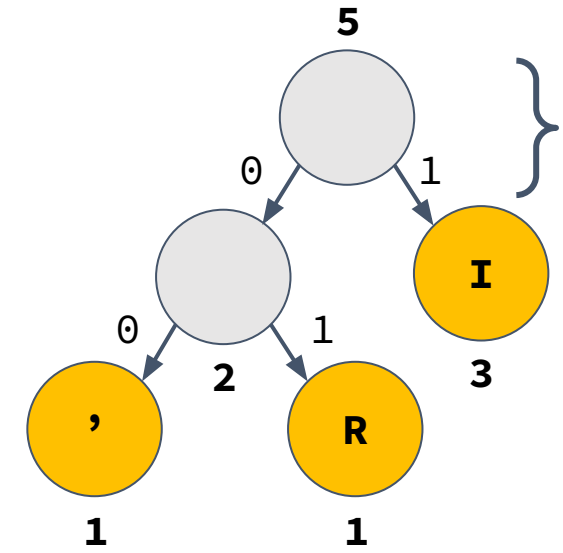
lower priority

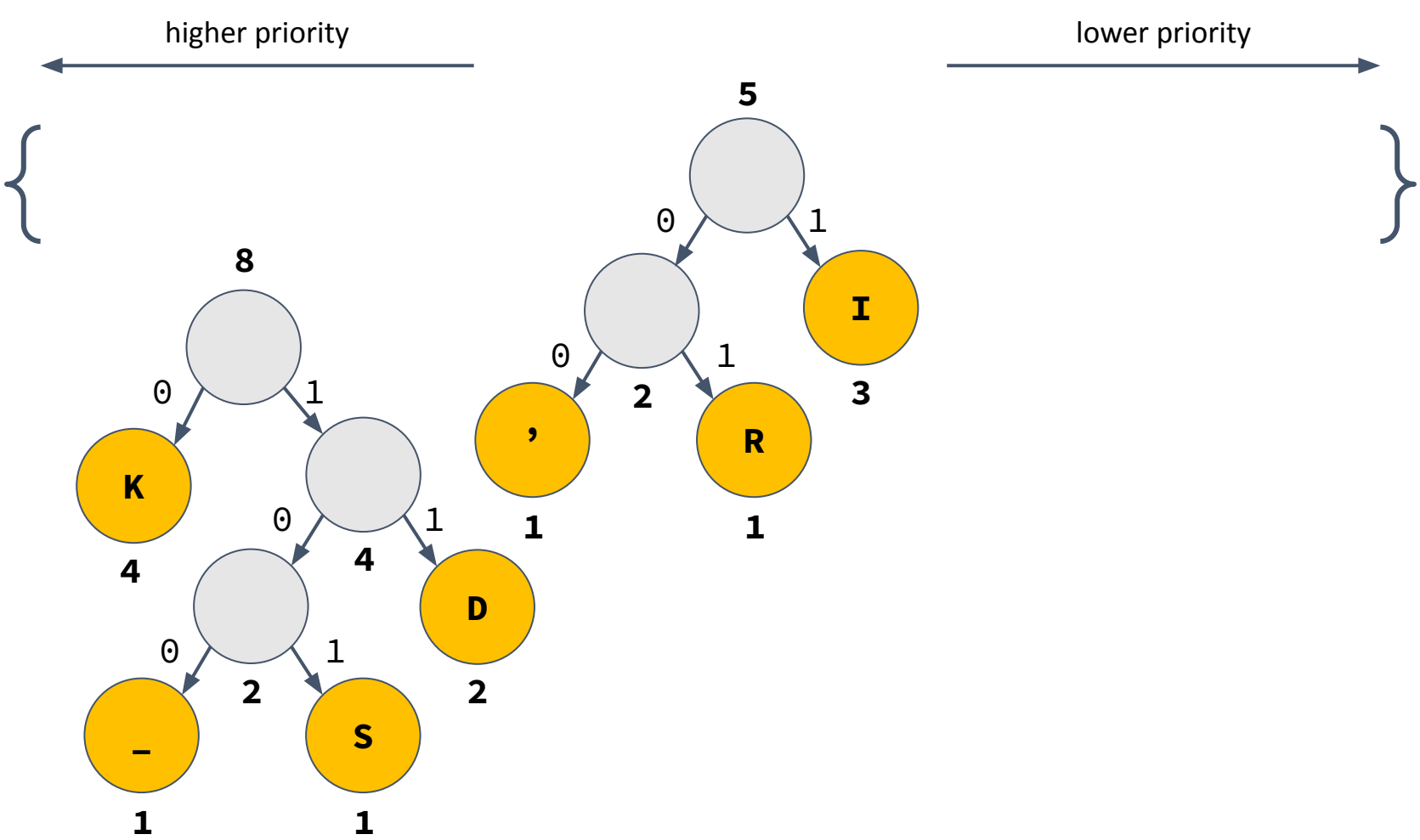


higher priority



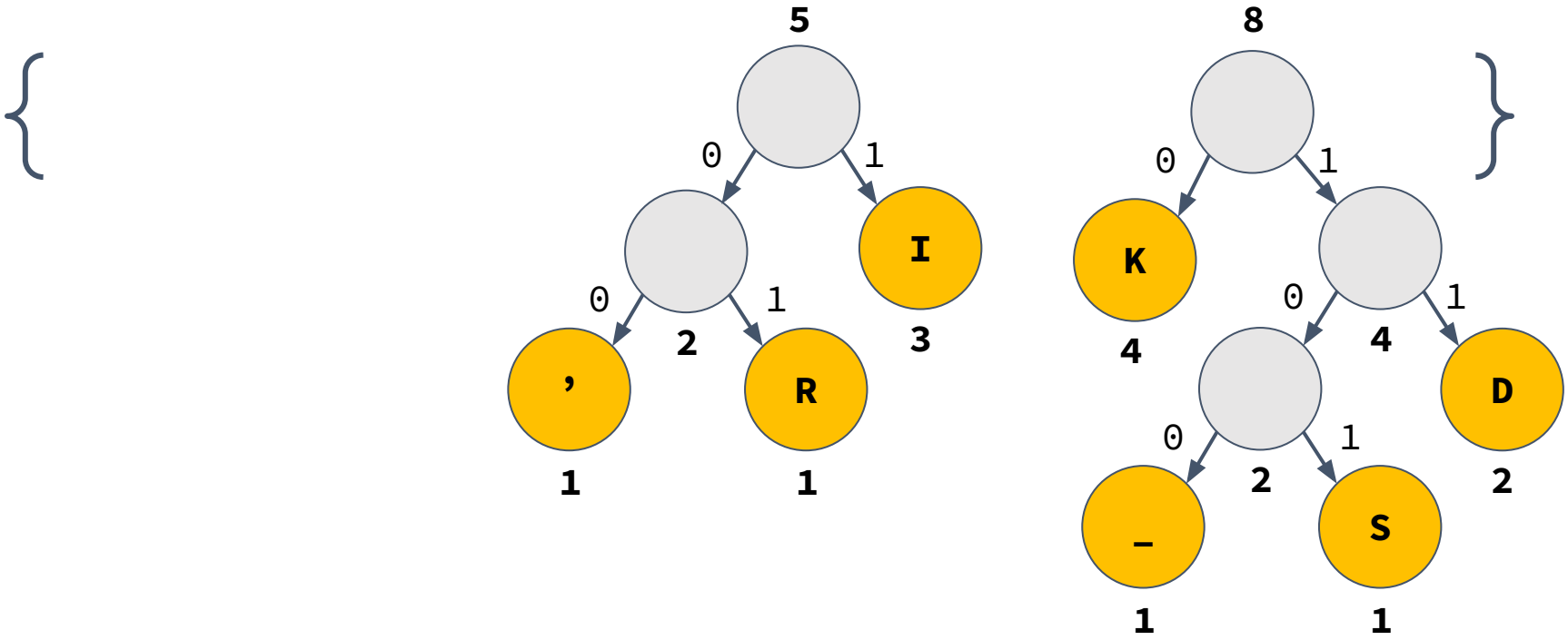
lower priority




higher priority
lower priority
8
0
1
K
4
0
1
4
0
1
D
2
0
2
1
_
1
S
1
5
0
1
0
2
1
I
3
,
1
R
1

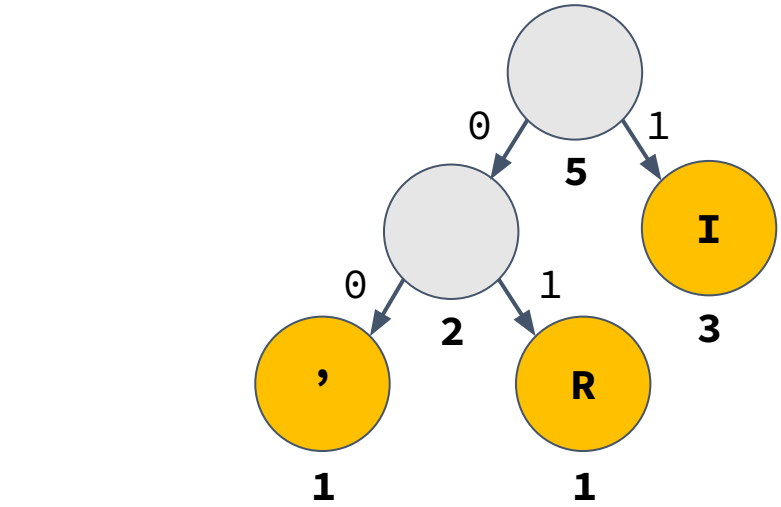
higher priority

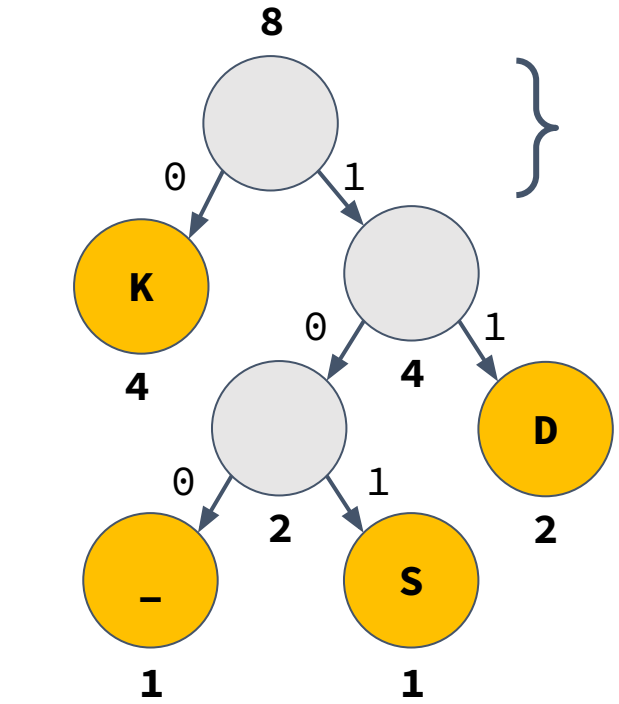
lower priority



higher priority

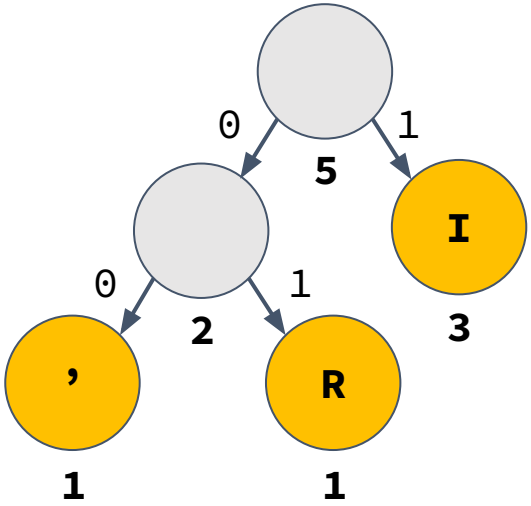
lower priority

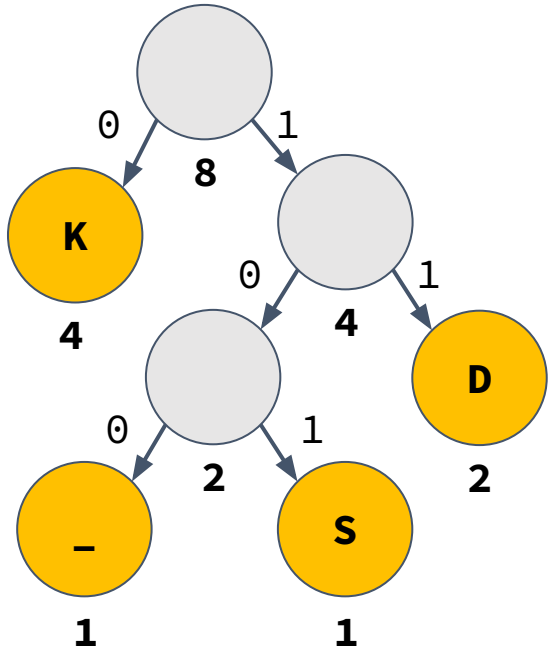




higher priority

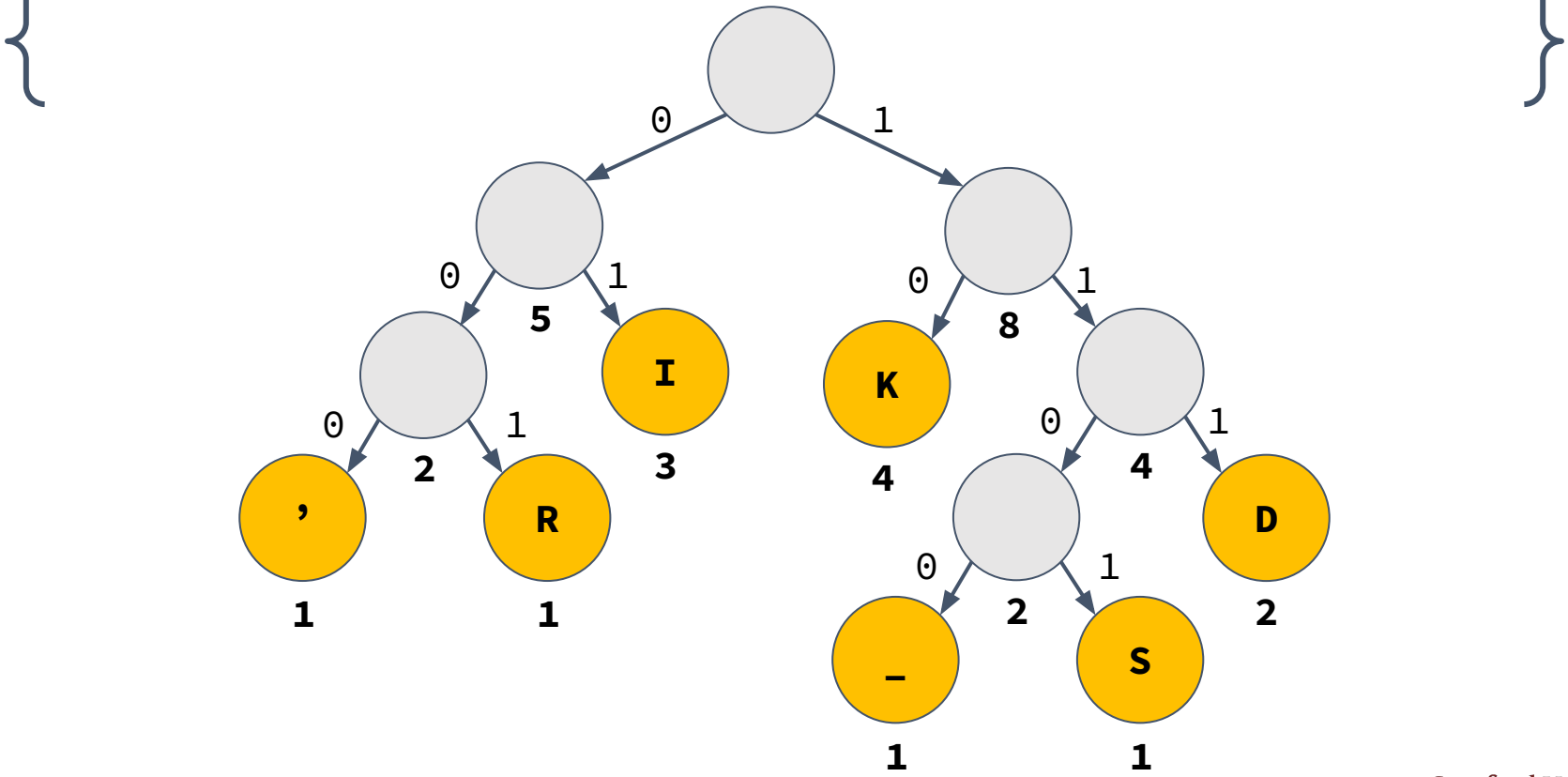
lower priority





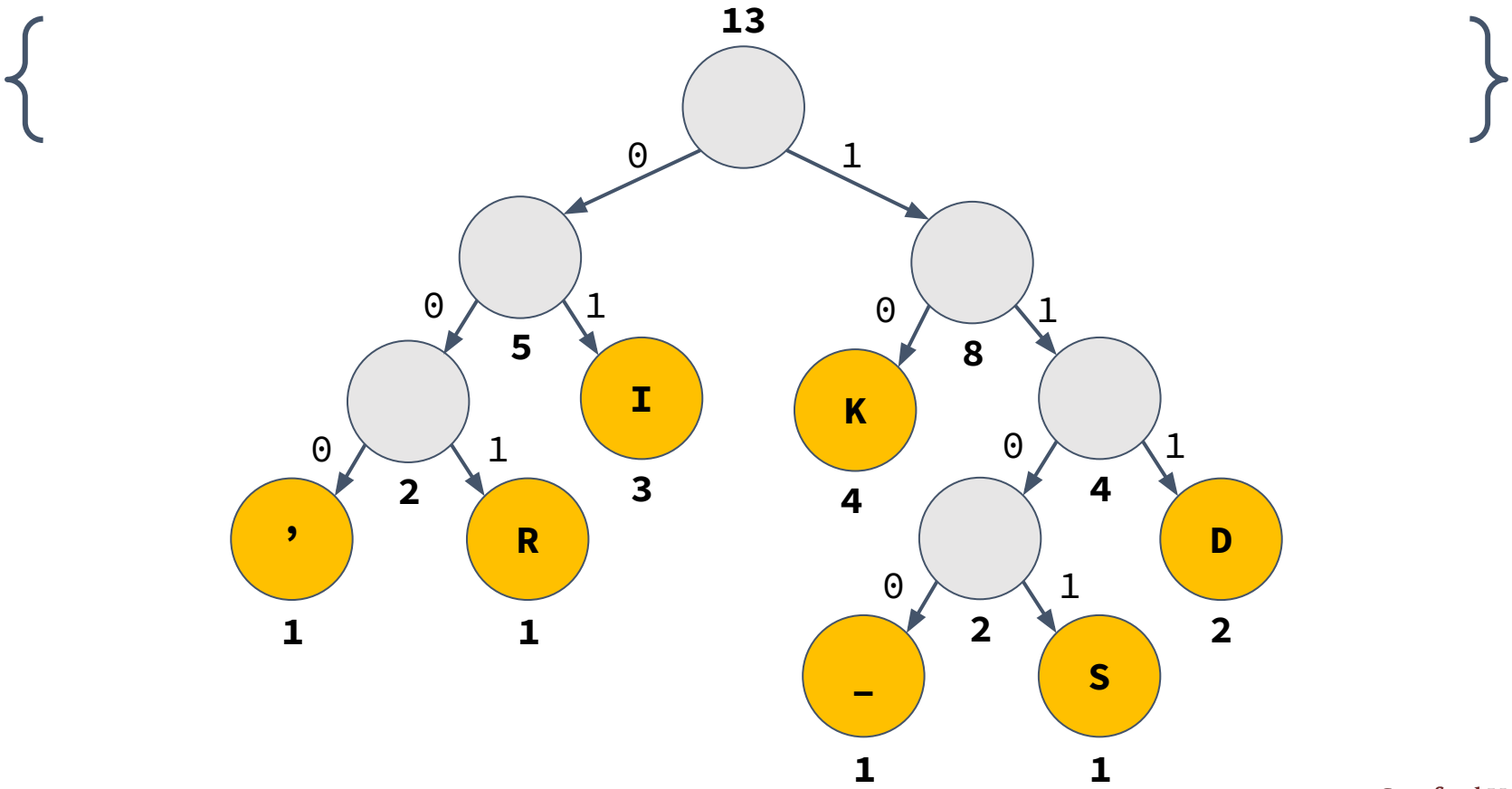
higher priority

lower priority



higher priority

lower priority



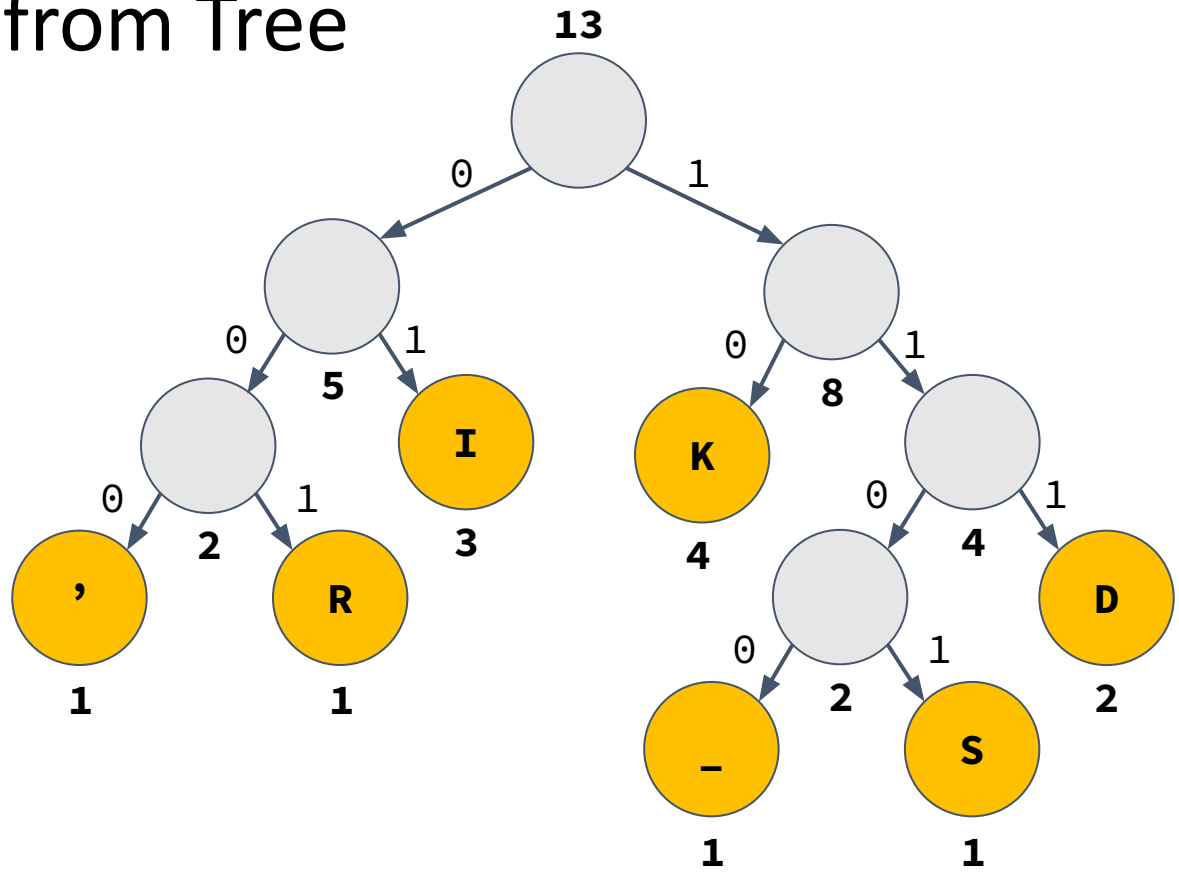
Huffman Coding Pseudocode

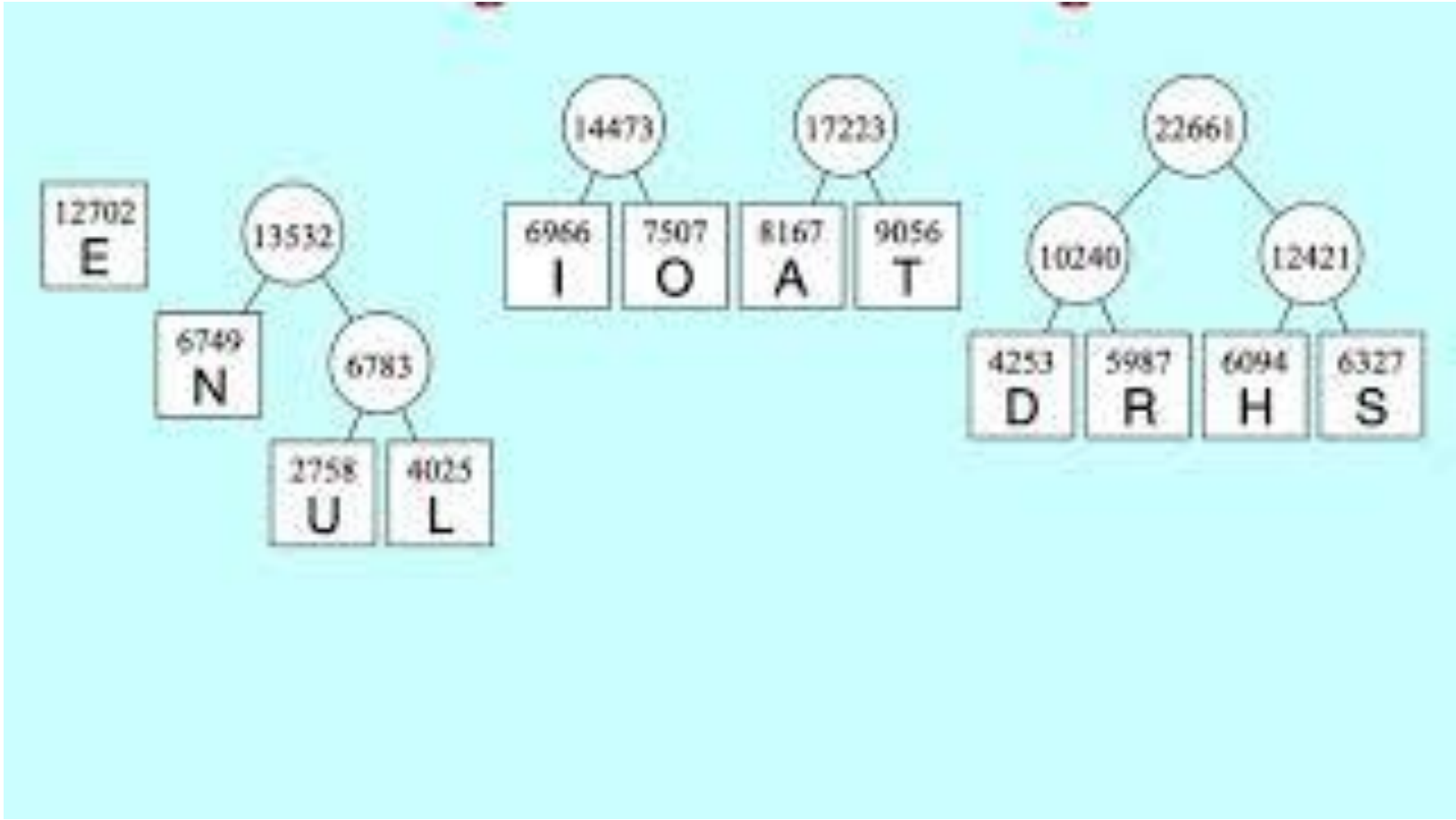
To generate the optimal encoding tree for a given piece of text

1. Build a frequency table that tallies the number of times each character appears in the text
2. Initialize an empty priority queue that will hold partial trees
3. Create one leaf node per distinct character in the text, and add each leaf node to the queue where the priority is the frequency of the character
4. While there are two or more trees in the priority queue:
    a. Dequeue the two lowest-priority trees
    b. Combine them together to form a new tree whose priority is the sum of the priorities of the two trees
    c. Add that tree back to the priority queue

Generate Table from Tree

character	*code*
K	10
I	01
D	111
R	001
,	000
S	1101
_	1100




12702
E
13532
6749
N
6783
2758
U
4025
L
14473
6966
I
7507
O
17223
8167
A
9056
T
22661
10240
4253
D
5987
R
12421
6094
H
6327
S

Decompress

100100110000110111001110110110110

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Decompress

100100110000110111001110110110110

character	*code*
K	10
I	01
D	111
R	001
’	000
S	1101
_	1100

Decompress

10010011000011011100111011011110110

Transmitting the Tree

- In order to decompress the text, we have to remember what encoding scheme we used
- Prefix the compressed data with a header containing information to rebuild the tree

- Theorem: There is no compression algorithm that can always compress all inputs
  - Proof: Take CS103!

Huffman Coding Recap

- Data compression is a very important real world problem that relies of patterns in data to find efficient, compact data representation schemes
- In order to support variable-length encodings for data, we must use prefix coding schemes, which can be modeled as binary trees
- Huffman coding uses a greedy algorithm to construct encodings by building a tree from the bottom-up, putting the most frequency characters higher up in the coding tree
- We must send the encoding table/tree with the compressed message

Assignment 6 - Huffman Coding

- Decode/decompress some data
  - Given a flattened tree, turn it back into an encoding tree
  - Given a sequence of bits and an encoding tree, decode a message
  - Decode a mystery file

Assignment 6 - Huffman Coding

- Decode/decompress some data
  - Given a flattened tree, turn it back into an encoding tree
  - Given a sequence of bits and an encoding tree, decode a message
  - Decode a mystery file
- Encode/compress some data
  - Build a Huffman Encoding Tree for a particular string of text
  - Given an encoding tree, flatten it
  - Encode some text to your SL

More to Explore

- UTF-8 and Unicode
  - A variable length encoding that has replaced ASCII
- Kolmogorov Complexity
  - What's the theoretical limit to compression techniques?
- Adaptive Coding Techniques
  - Can you change your encoding system as you go?
- Shannon Entropy
  - A mathematical bound on Huffman coding

See you tomorrow!