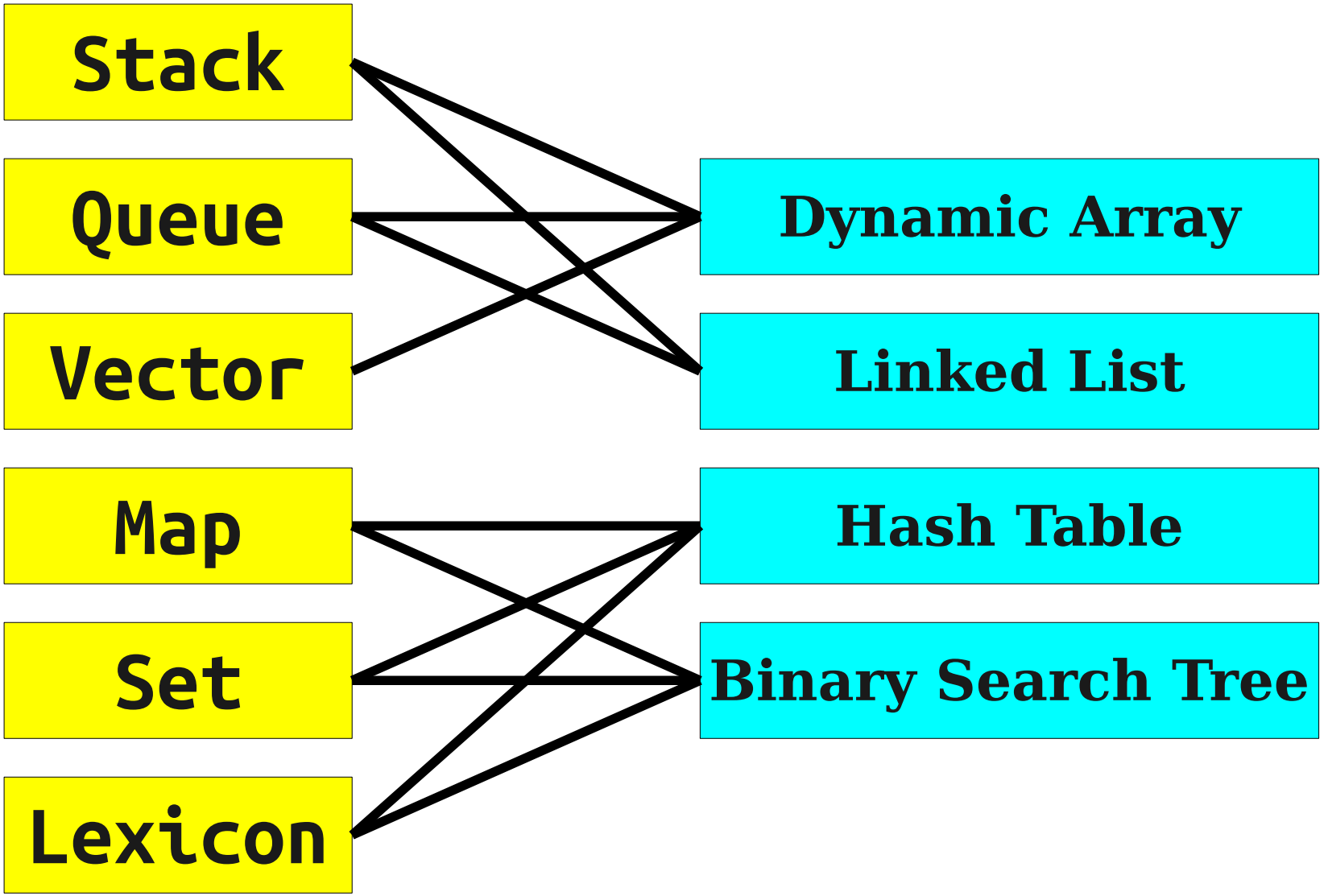
Beyond Data Structures

Interface and Implementation



Goals for Today

- Explore applications of techniques from data structures in other settings.
- Main example: **Huffman Encoding**.

Huffman Encoding

It's All Bits and Bytes

- Data stored on disk consists of 0s and 1s.
  - Usually encoded by magnetic orientation on small (10nm!) metal particles.
- A single 0 or 1 is called a **bit**.
- A group of eight bits is called a **byte**.
- There are $2^8 = 256$ different bytes.

Representing Text

- Suppose we want to represent English text (English letters, numbers, punctuation, spaces, etc.)
- There are fewer than 256 different symbols we would need to store.
- Consequently, we could assign one byte (eight bits) per character we want to store.
- One encoding for text (**ASCII**) does just that.
- (More on non-English characters later today).

Some Simple ASCII Values

K	01001011
I	01001001
R	01010010
K	01001011
'	00100111
S	01010011
	00100000
D	01000100
I	01001001
K	01001011
D	01000100
I	01001001
K	01001011

Some Simple ASCII Values

K	01001011
I	01001001
R	01010010
K	01001011
'	00100111
S	01010011
	00100000
D	01000100
I	01001001
K	01001011
D	01000100
I	01001001
K	01001011

```
01001011010010010101001001001011
00100111010100110010000001000100
01001001010010110100010001001001
01001011
```

Some Simple ASCII Values

```
01001011010010010101001001001011
00100111010100110010000001000100
01001001010010110100010001001001
01001011
```

Some Simple ASCII Values

01001011	01001001	01010010	01001011
00100111	01010011	00100000	01000100
01001001	01001011	01000100	01001001
01001011			

Some Simple ASCII Values

01001011
01001001
01010010
01001011
00100111
01010011
00100000
01000100
01001001
01001011
01000100
01001001
01001011

Some Simple ASCII Values

K	01001011
I	01001001
R	01010010
K	01001011
'	00100111
S	01010011
	00100000
D	01000100
I	01001001
K	01001011
D	01000100
I	01001001
K	01001011

Space Requirements

- ASCII is a **fixed-length encoding**; storing n letters encoded with ASCII always takes exactly exactly $8n$ bits.
  - Space for **KIRK'S DIKDIK**: 104 bits.
- This is useful if we represent each of the 2^8 possible characters with high frequency.
- It's not very useful if we only use a small subset of the characters.

A Different Encoding

- The phrase **KIRK'S DIKDIK** has exactly 7 different characters in it.
- We can use a different encoding to represent this string using many fewer bits:

A Different Encoding

- The phrase **KIRK'S DIKDIK** has exactly 7 different characters in it.
- We can use a different encoding to represent this string using many fewer bits:

K	000	S	100
I	001		101
R	010	D	110
'	011		

A Different Encoding

- The phrase **KIRK'S DIKDIK** has exactly 7 different characters in it.
- We can use a different encoding to represent this string using many fewer bits:

000	001	010	000	011	100	101	110	001	000	110	001	000
K	I	R	K	'	S		D	I	K	D	I	K

K	000	S	100
I	001		101
R	010	D	110
'	011		

A Different Encoding

- The phrase **KIRK'S DIKDIK** has exactly 7 different characters in it.
- We can use a different encoding to represent this string using many fewer bits:

000	001	010	000	011	100	101	110	001	000	110	001	000
K	I	R	K	'	S		D	I	K	D	I	K

- Down from 104 bits to 39 bits: using 37.5% as much space as before!

K	000	S	100
I	001		101
R	010	D	110
'	011		

Exploiting Redundancy

- Not all letters have the same frequency in “KIRK'S DIKDIK.”
- Frequency table:

K	4
I	3
D	2
R	1
'	1
S	1
	1

- What if we gave shorter encodings to more common characters?

A First Attempt

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

0101010111000000100010

0	1	01	0	10	11	100	00	001	0	00	1	0
K	I	R	K	'	S		D	I	K	D	I	K

A First Attempt

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

```
01010101110000000100010
```

A First Attempt

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

0101010111000000100010

0	1	01	0	10	11	100	00	001	0	00	1	0
K	I	R	K	'	S		D	I	K	D	I	K

A First Attempt

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

01010101111000000100010

01	01	01	11	00	00	00	100	01	0
R	R	R	S	D	D	D		R	K

A First Attempt

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

1000000100010

01	01	01	11	00	00	00	100	01	0
R	R	R	S	D	D	D		R	K

The Problem

- 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 character stops and the next starts.
- Is this possible? If so, how?

Prefix Codes

- A **prefix code** is an encoding system in which no two codes are prefixes of one another.
- For example:

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111110001000111011001101100110

10	01	1111	10	001	000	1110	110	01	10	110	01	10
K	I	R	K	'	S		D	I	K	D	I	K

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

```
10011111100010001110110011011001110
```

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001110

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001110

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001110

10
K

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001110

10
K

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001110

10
K

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001100110

10
K

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111111000100011101100110110011 0

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111111000100011101100110110011 0

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011100110

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001 10

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111111000100011101100110110011 0

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111110001000111011001101100110

10	01
K	I

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111110001000111011001101100110

10	01	1111
K	I	R

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111000100011101100110110011 0

10	01	1111
K	I	R

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001 10

10	01	1111
K	I	R

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001 10

10	01	1111	10
K	I	R	K

Prefix Codes

- Using this prefix code, we can represent KIRK'S DIKDIK as the sequence

  `10011111100010001110110011011100110`

- This uses just 34 bits, compared to our initial 39.

Some Quick Announcements

Friday Four Square!

Today at 4:15PM at
Gates Computer Science

Announcements

- Assignment 5 due right now.
  - Due on Wednesday with a late day, but we ***strongly recommend*** completing it before the midterm.
- Assignment 6 (**Huffman Encoding**) out, due Wednesday, June 5th at 2:15 PM.
  - Build a program to compress files!
  - See an awesome application of trees and priority queues.
  - We'll get there in just a second...

Midterm Logistics

- Midterm is this upcoming Tuesday, May 28 from 7PM – 10PM.
  - Covers material up through **and including** Wednesday's lecture on binary search trees.
  - Practice exam solutions released today.
  - Review session this Sunday from 3PM – 5PM in 420-040.
  - Exam locations posted on course website.

Ceci n'est pas une annonce.

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

10011111100010001110110011011001l0

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

```
10011111100010001110110011011001l0
```

K	0
I	10
D	110
R	1110
'	11110
S	111110
	1111110

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

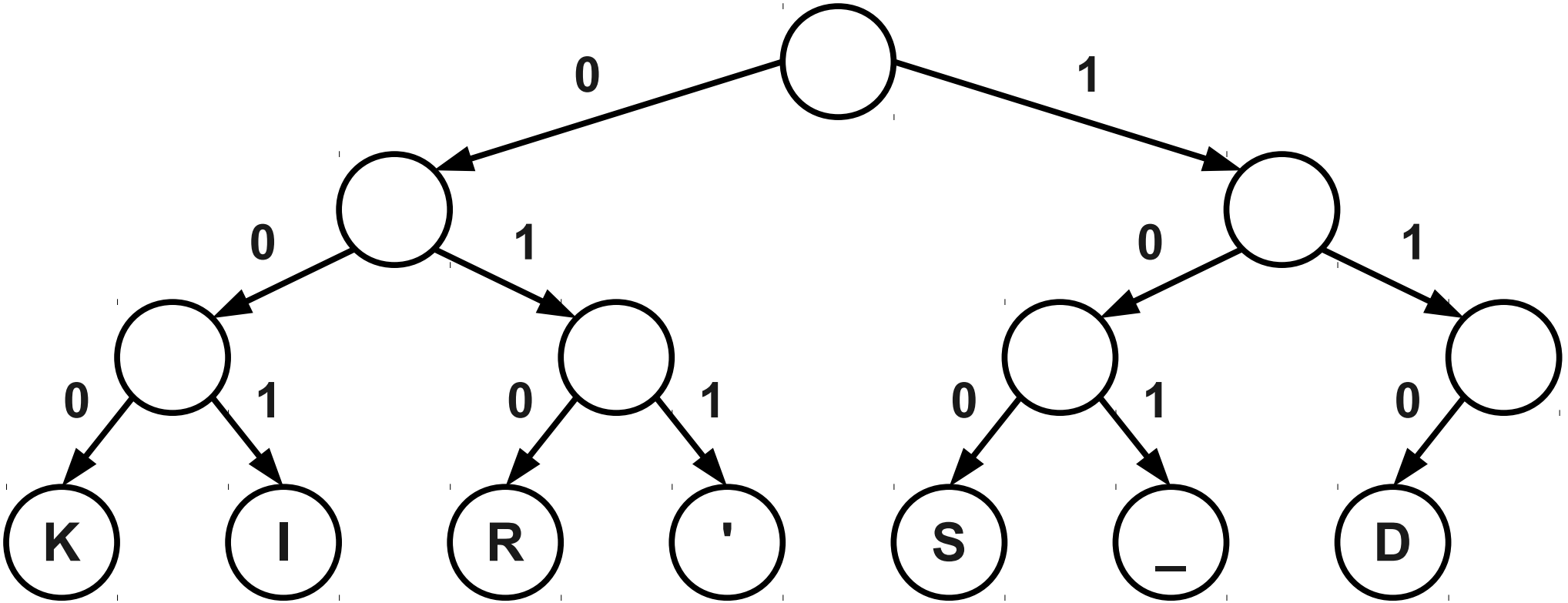
```
10011111100010001110110011011001l0
```

K	0
I	10
D	110
R	1110
'	11110
S	111110
	1111110

```
0101110011110111111011111101101001l0100
```

How do you find a “good” prefix code?

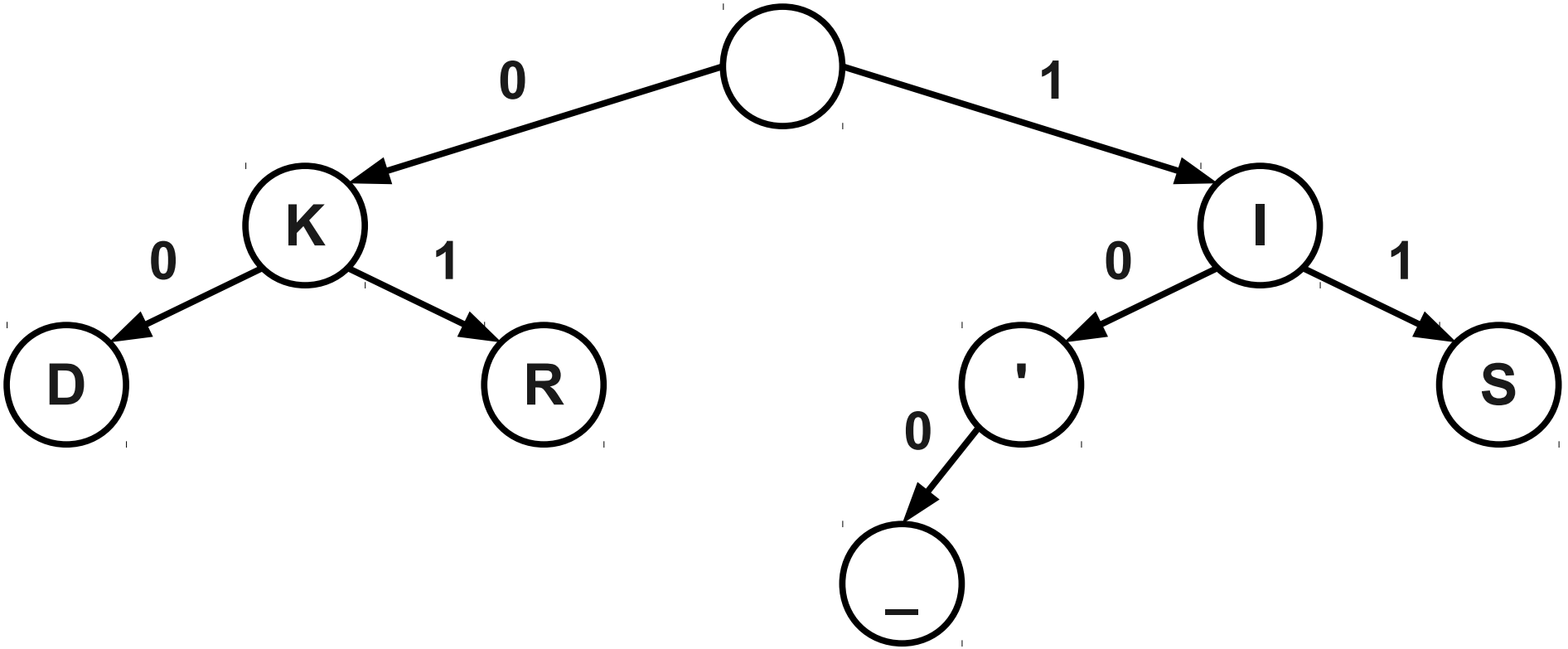
The Key Idea



K	000
I	001
R	010
'	011

S	100
	101
D	110

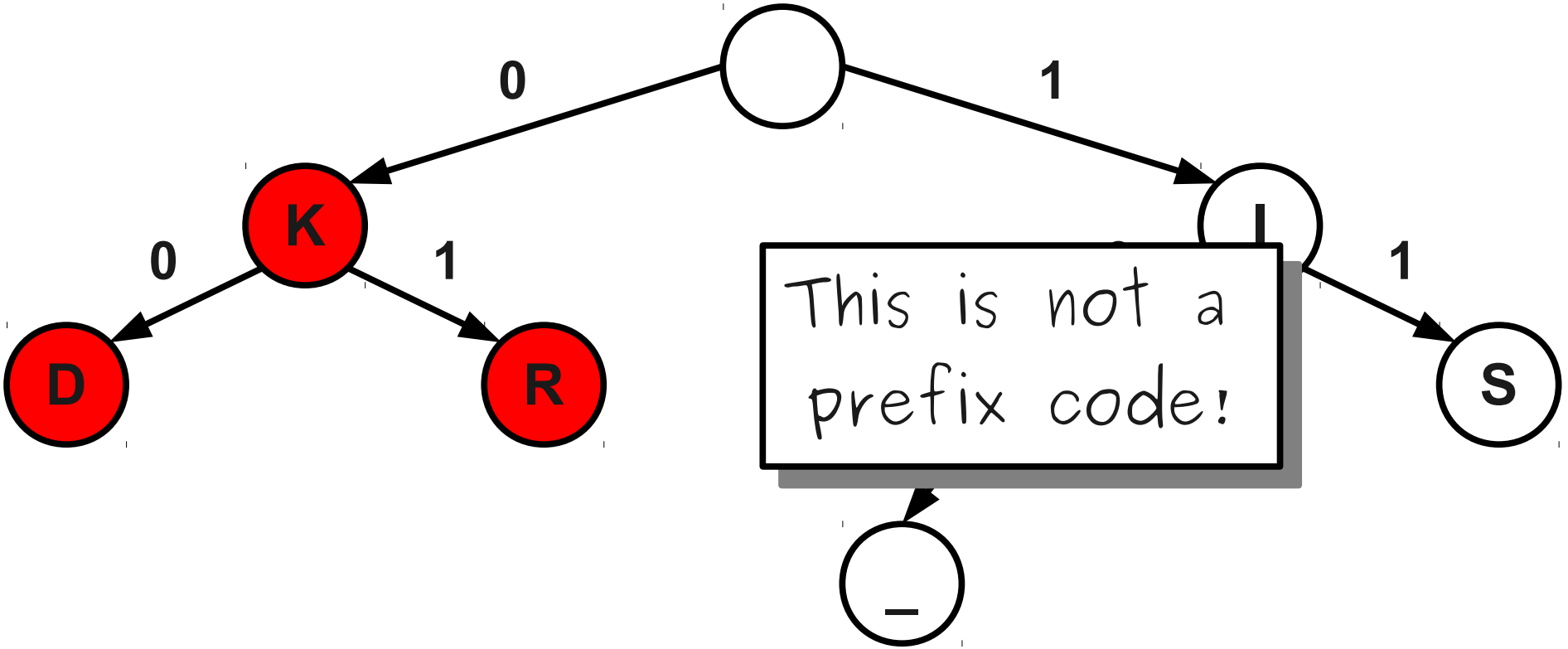
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

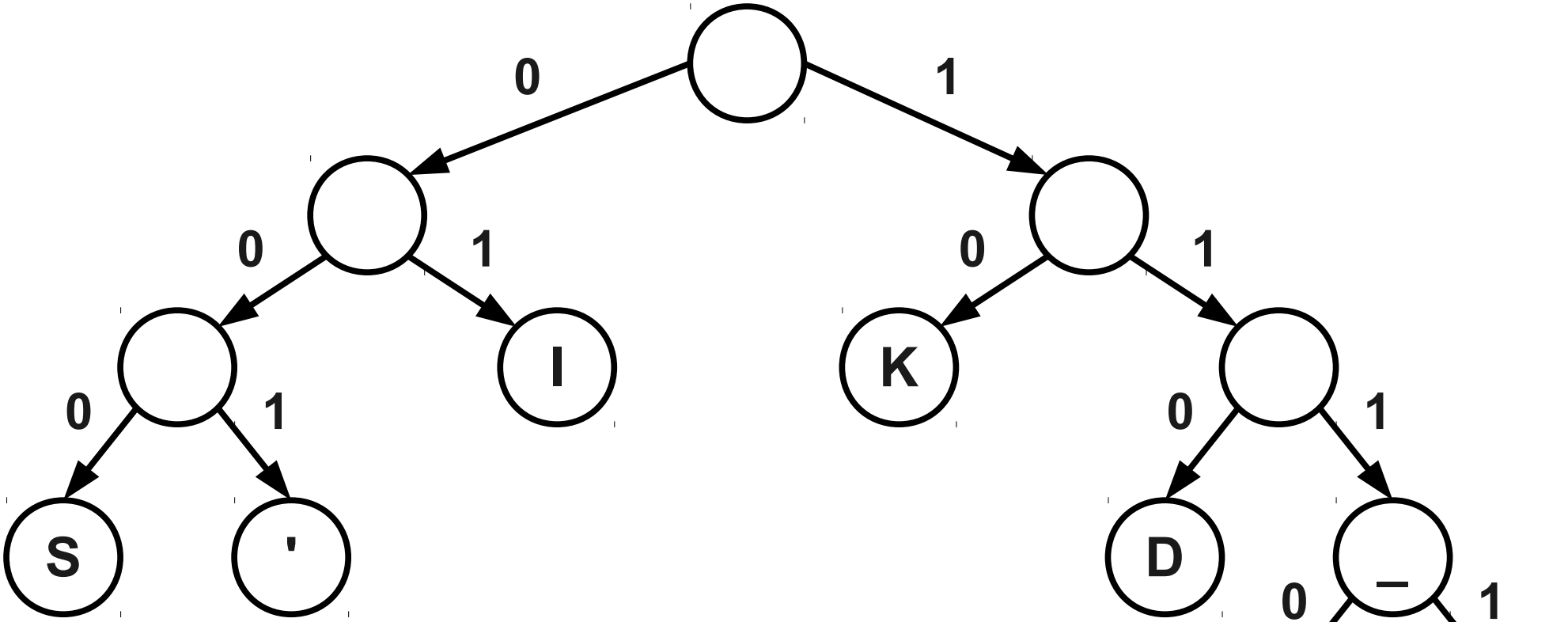
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

The Key Idea

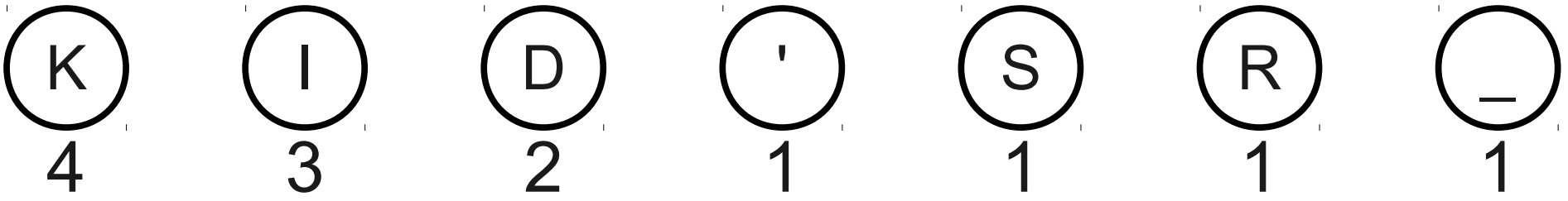


K	10
I	01
D	110
R	1111

'	001
S	000
	1110

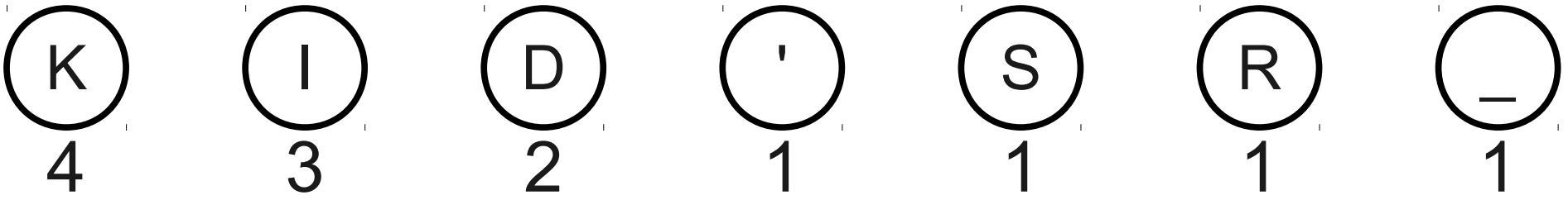
How do we find the best prefix-free binary tree?

Huffman Coding

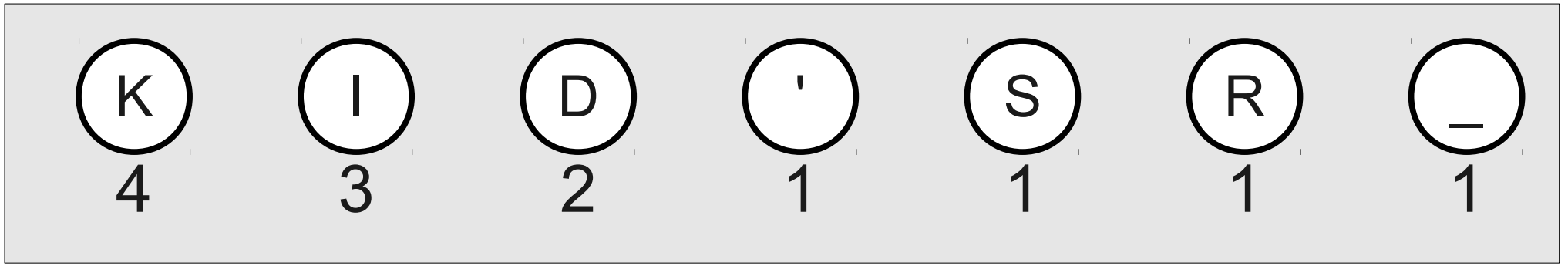


K	4
I	3
D	2
R	1
'	1
S	1
	1

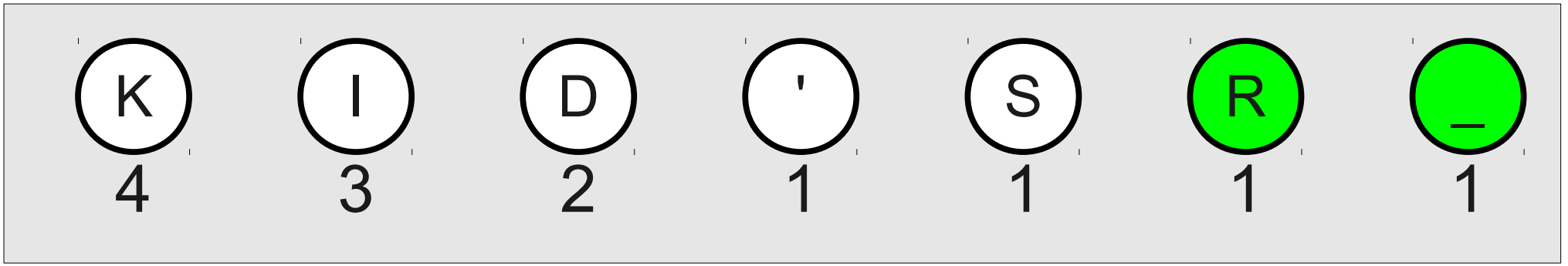
Huffman Coding



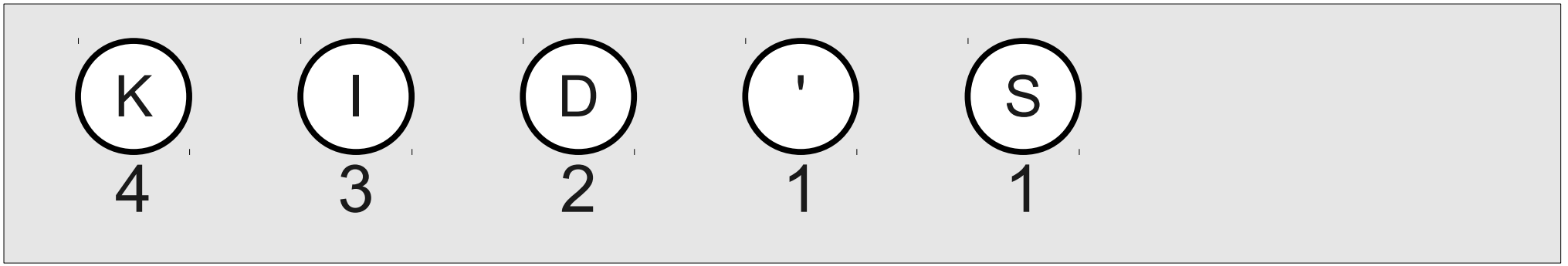
Huffman Coding

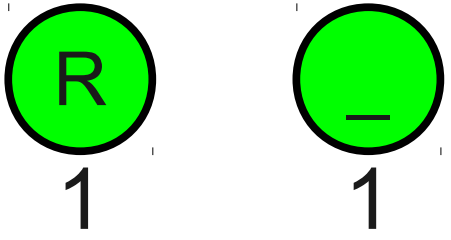


Huffman Coding

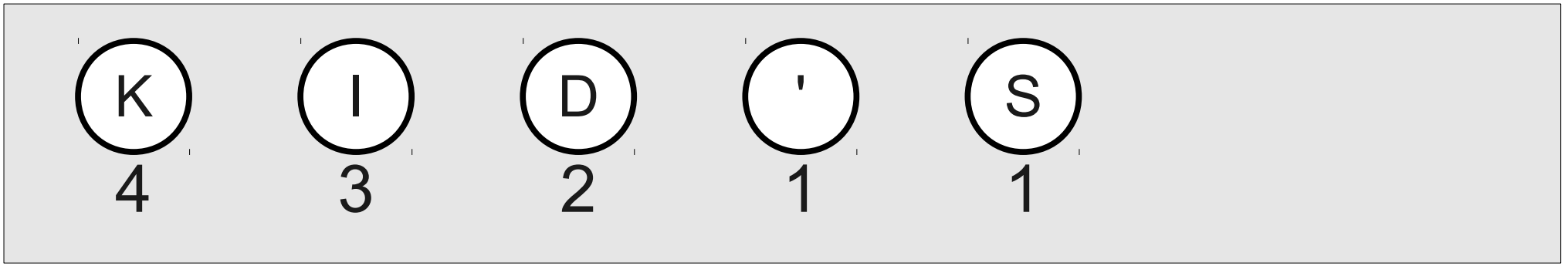


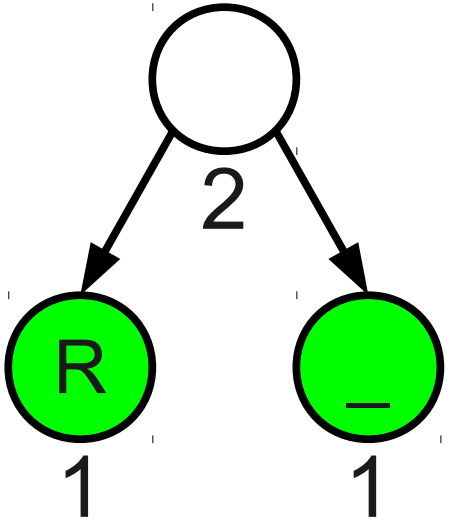
Huffman Coding



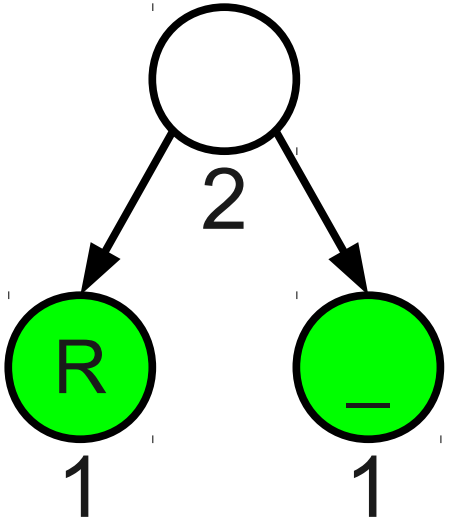


Huffman Coding

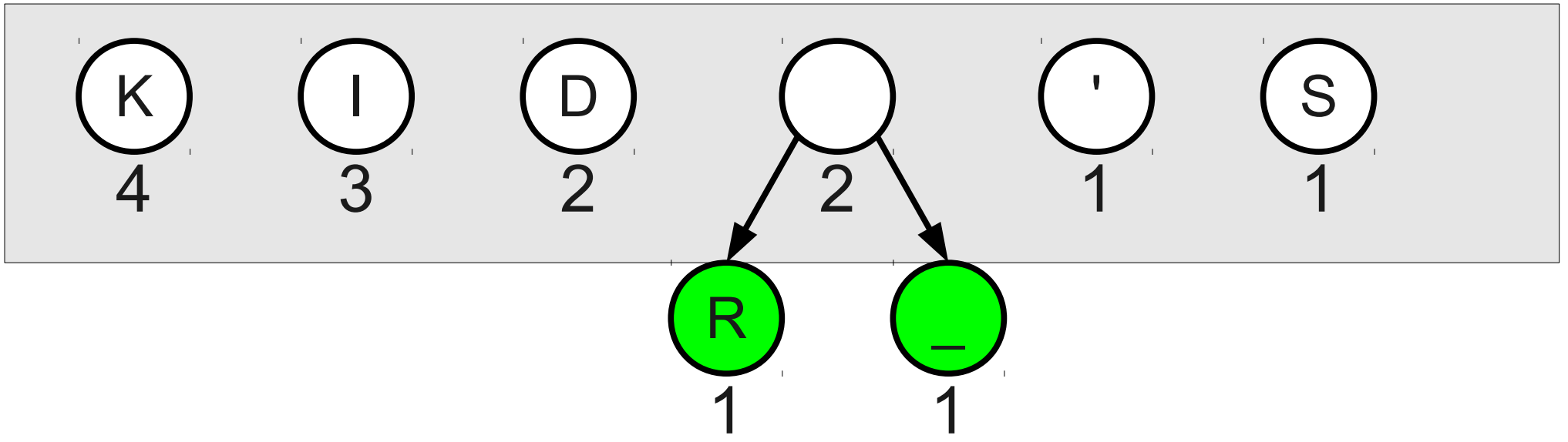




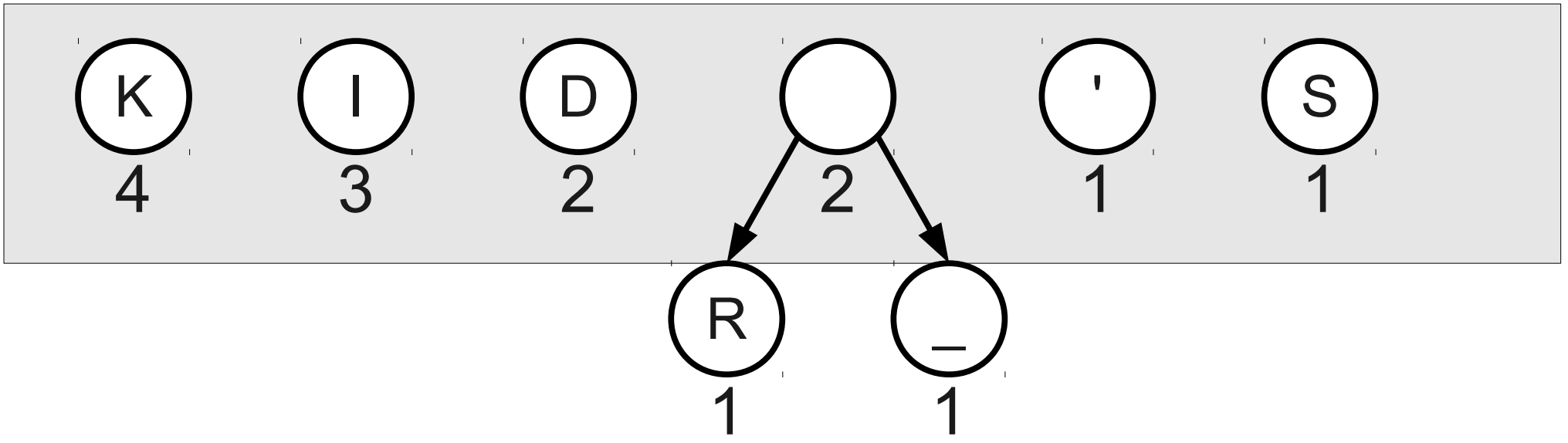
Huffman Coding



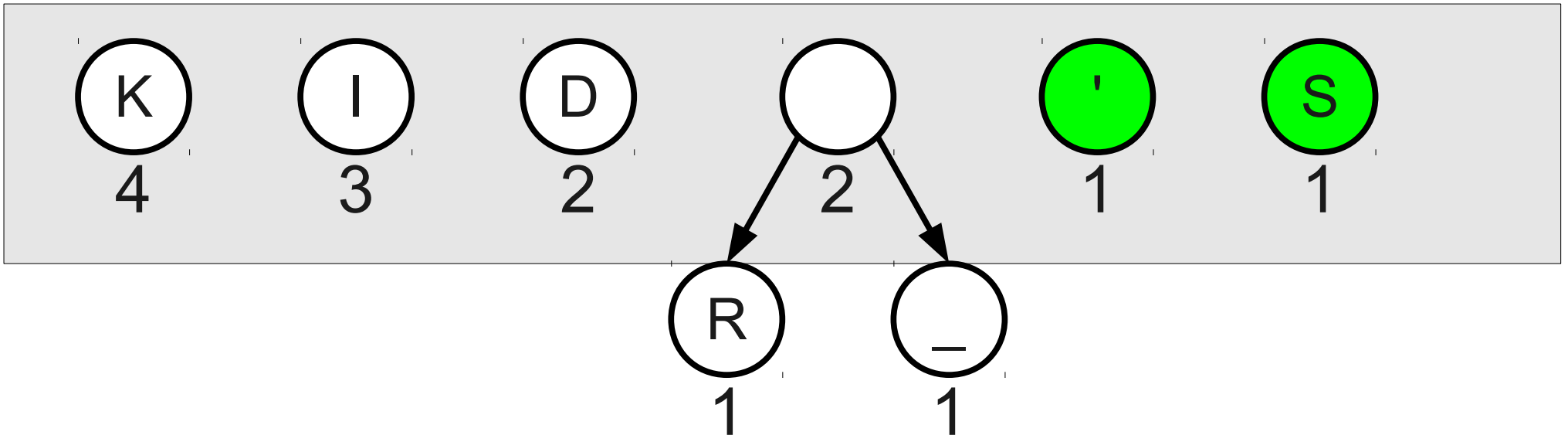
Huffman Coding



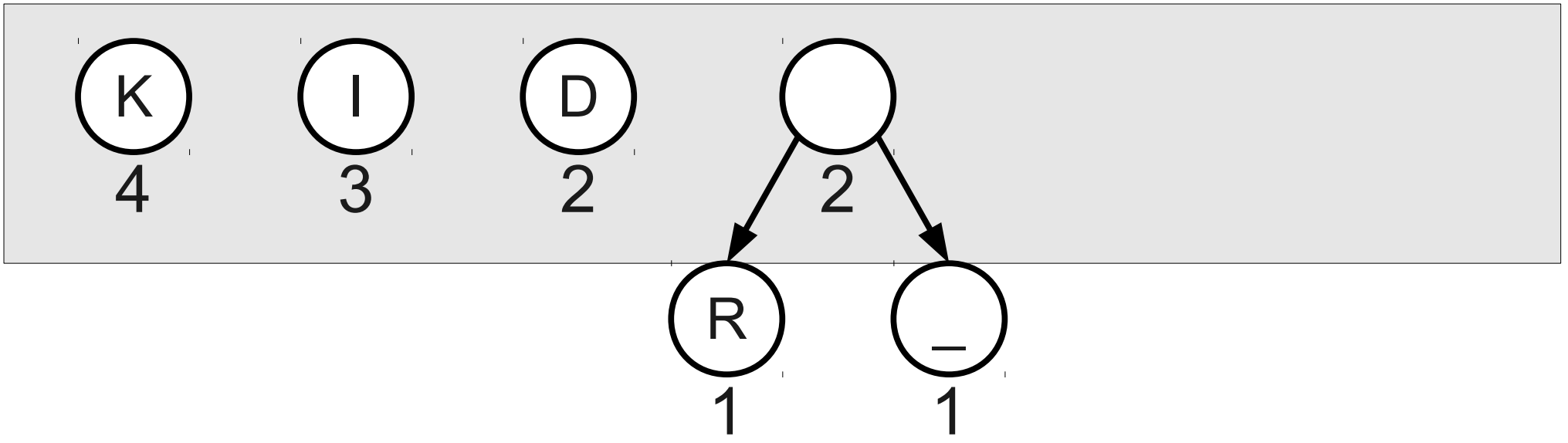
Huffman Coding

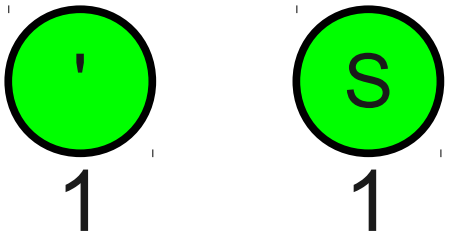


Huffman Coding

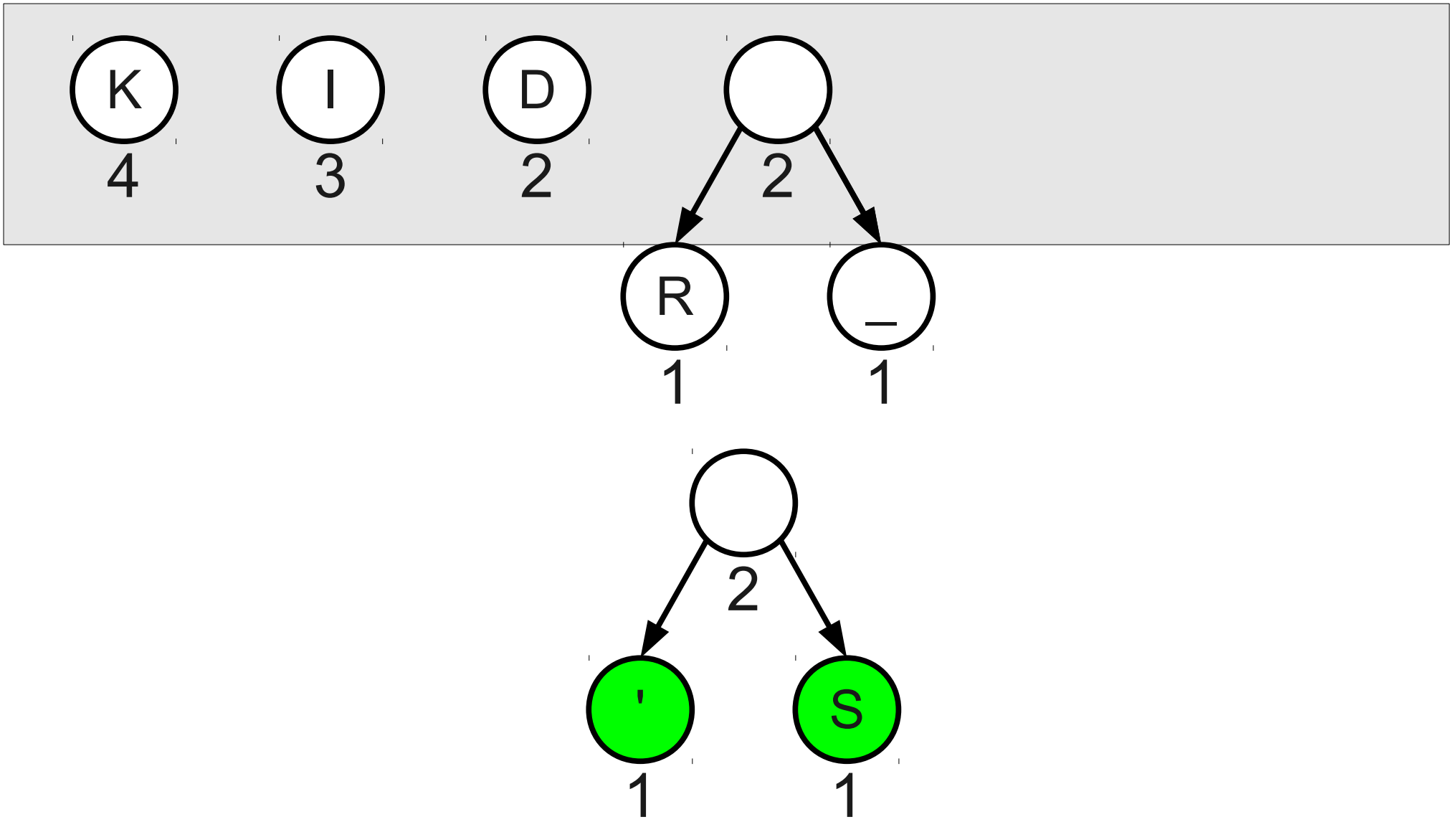


Huffman Coding

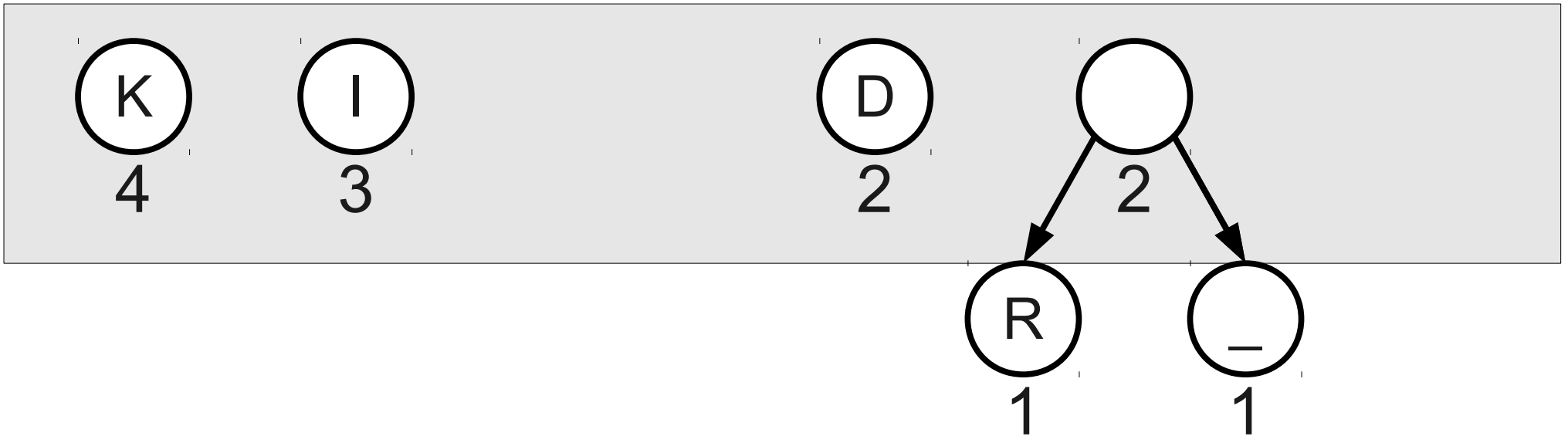


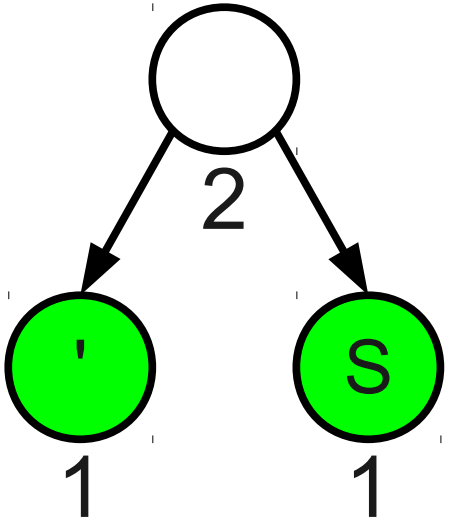


Huffman Coding

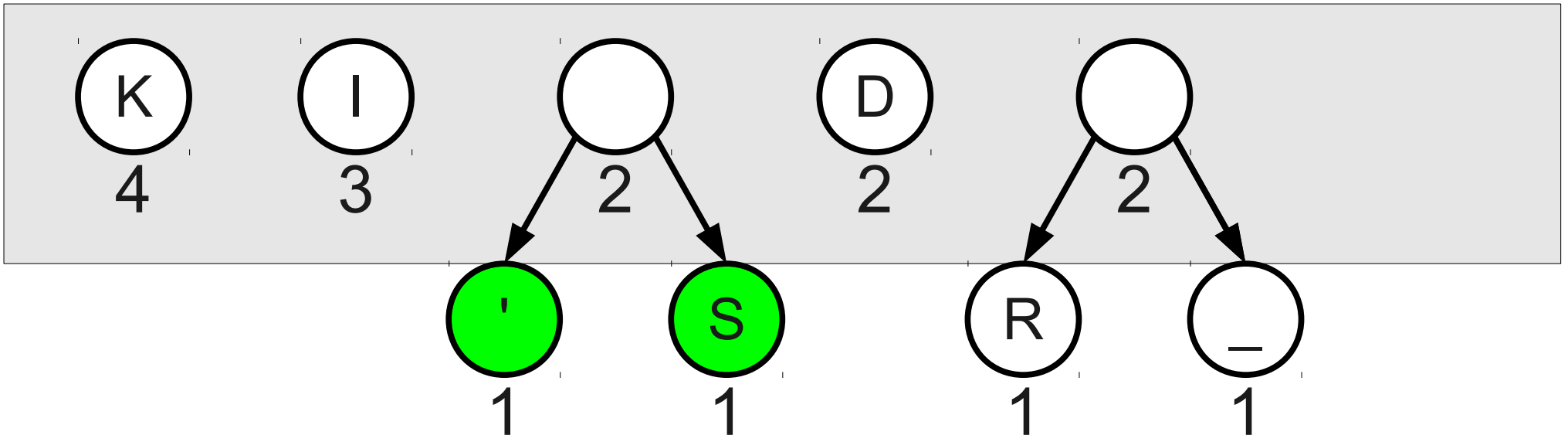


Huffman Coding

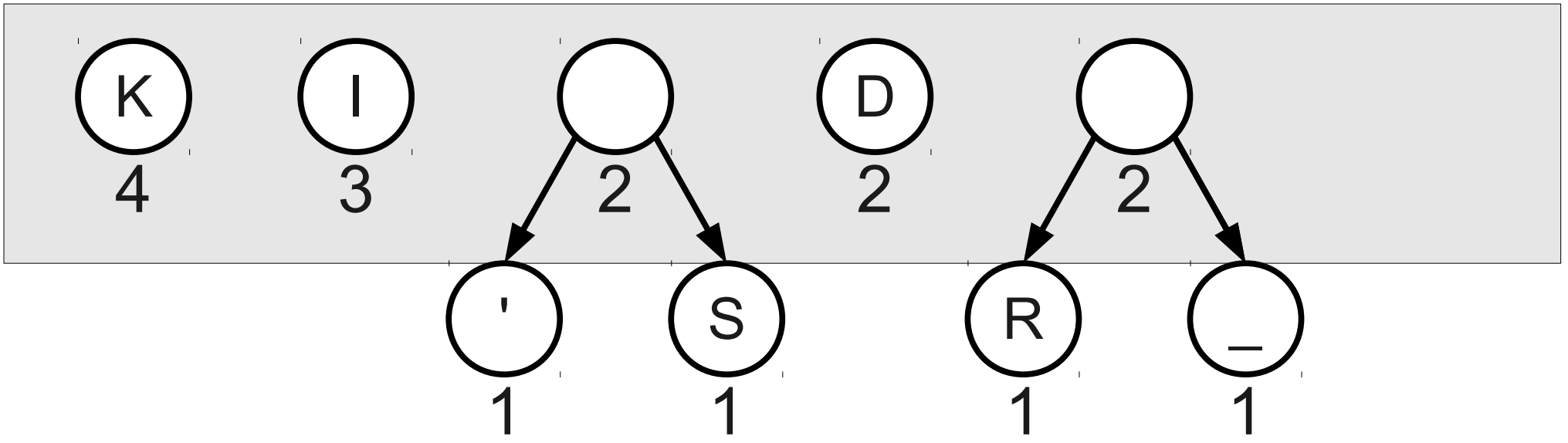




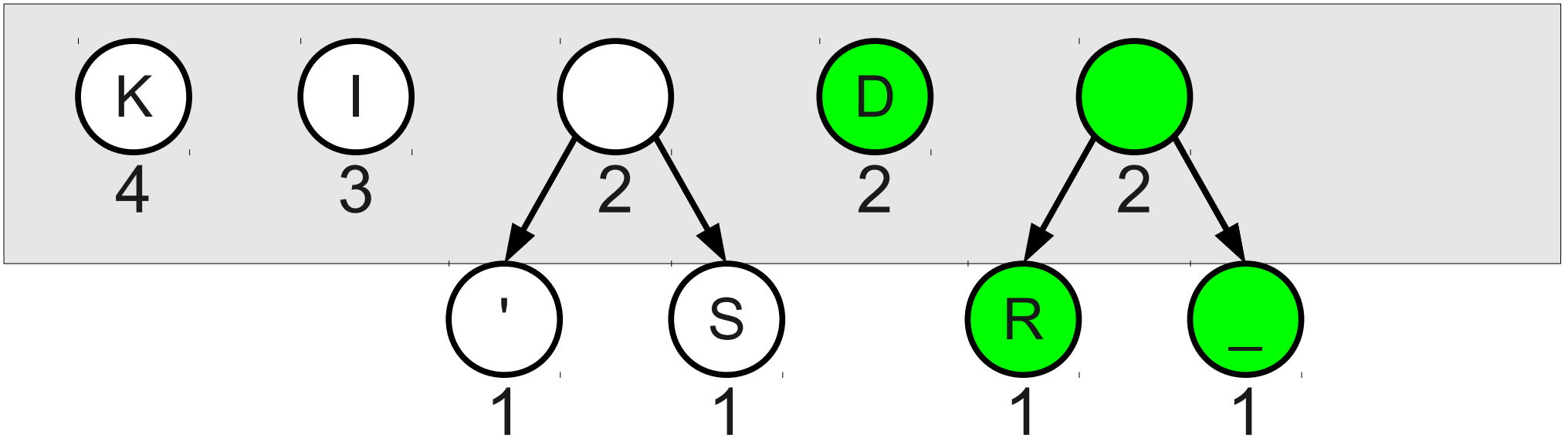
Huffman Coding



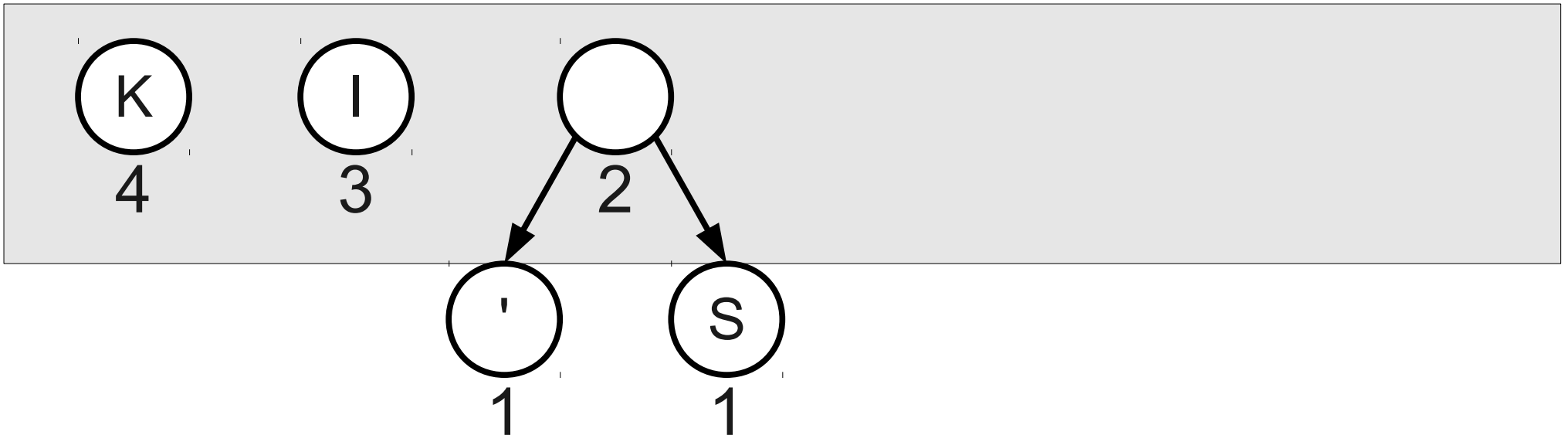
Huffman Coding

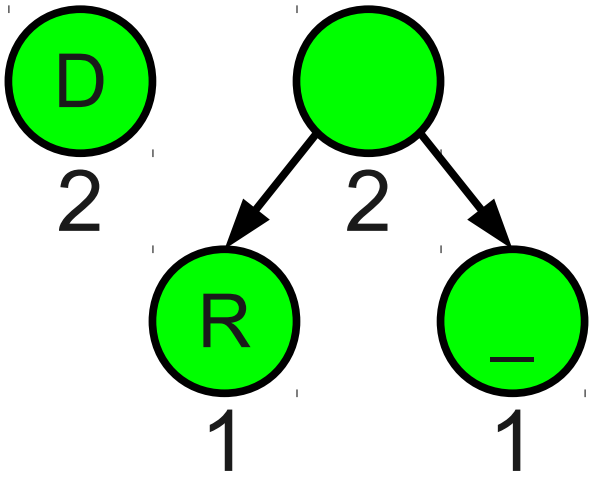


Huffman Coding

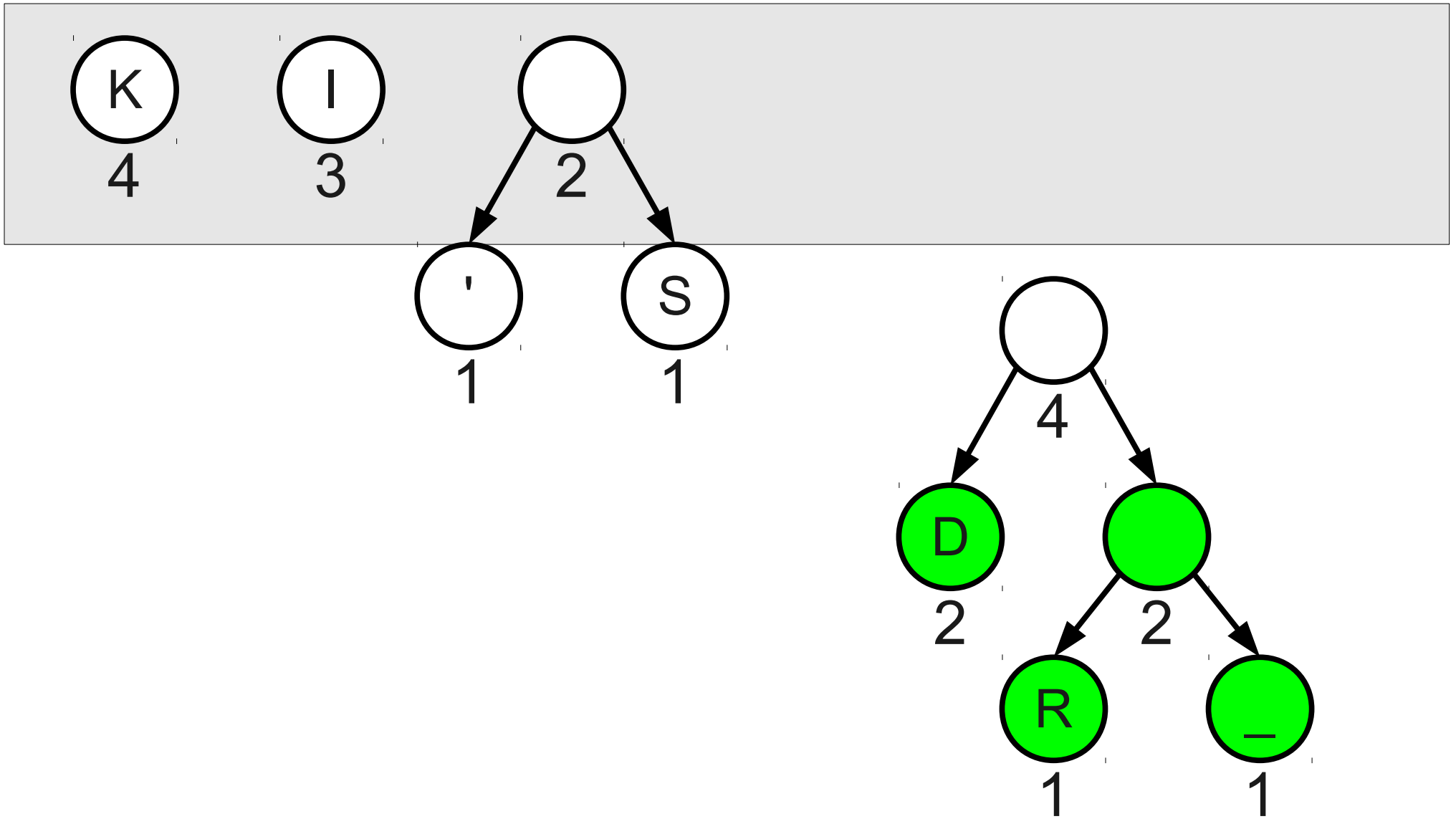


Huffman Coding

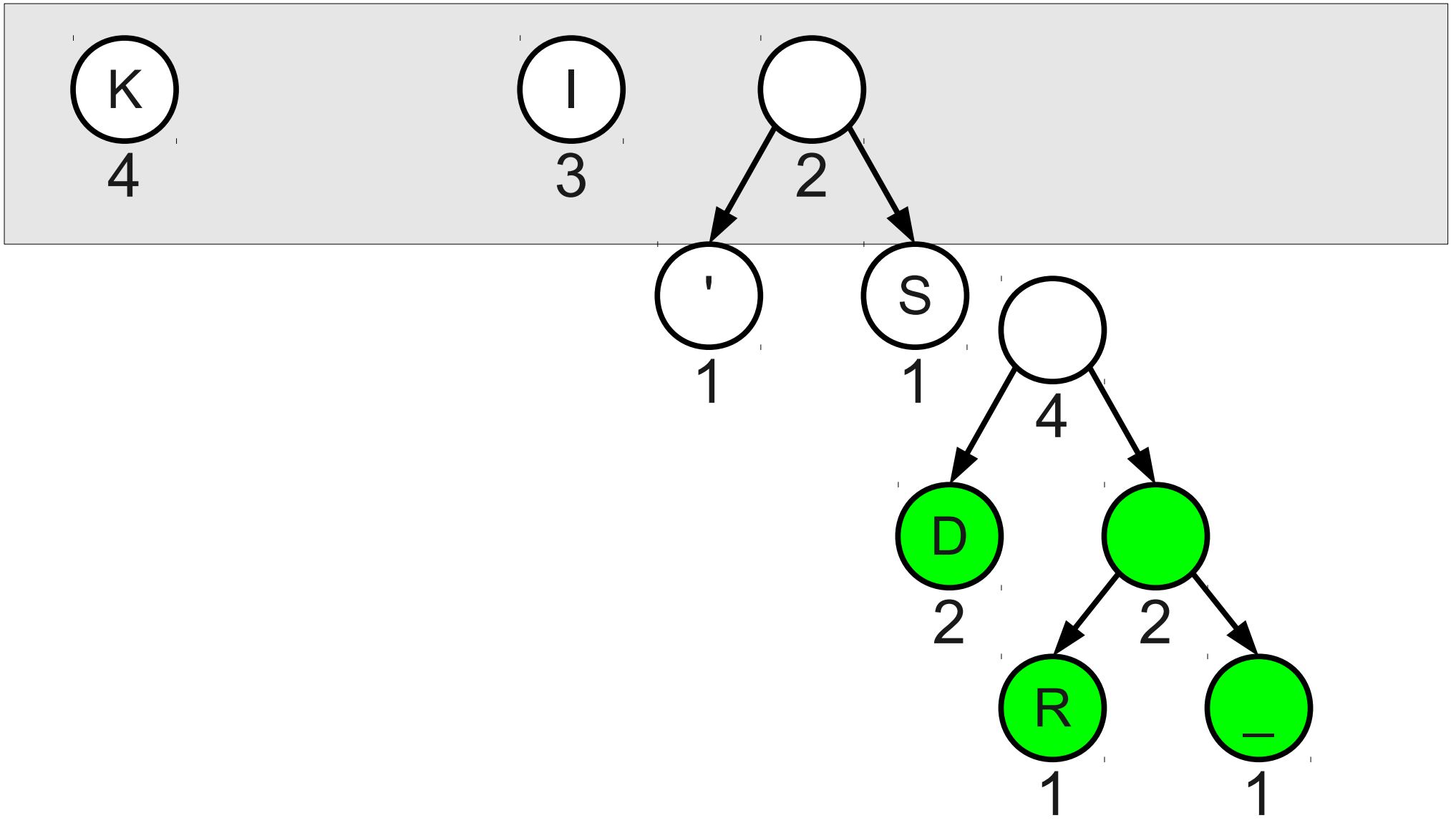




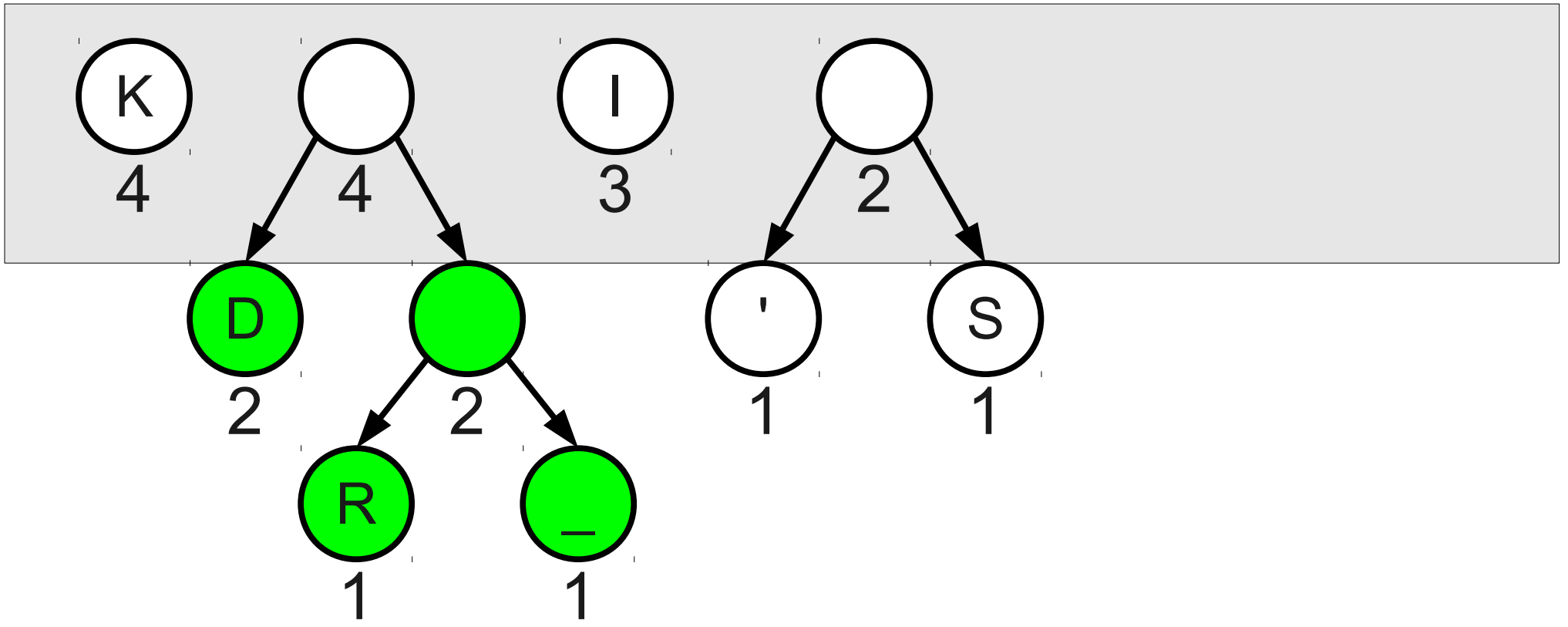
Huffman Coding



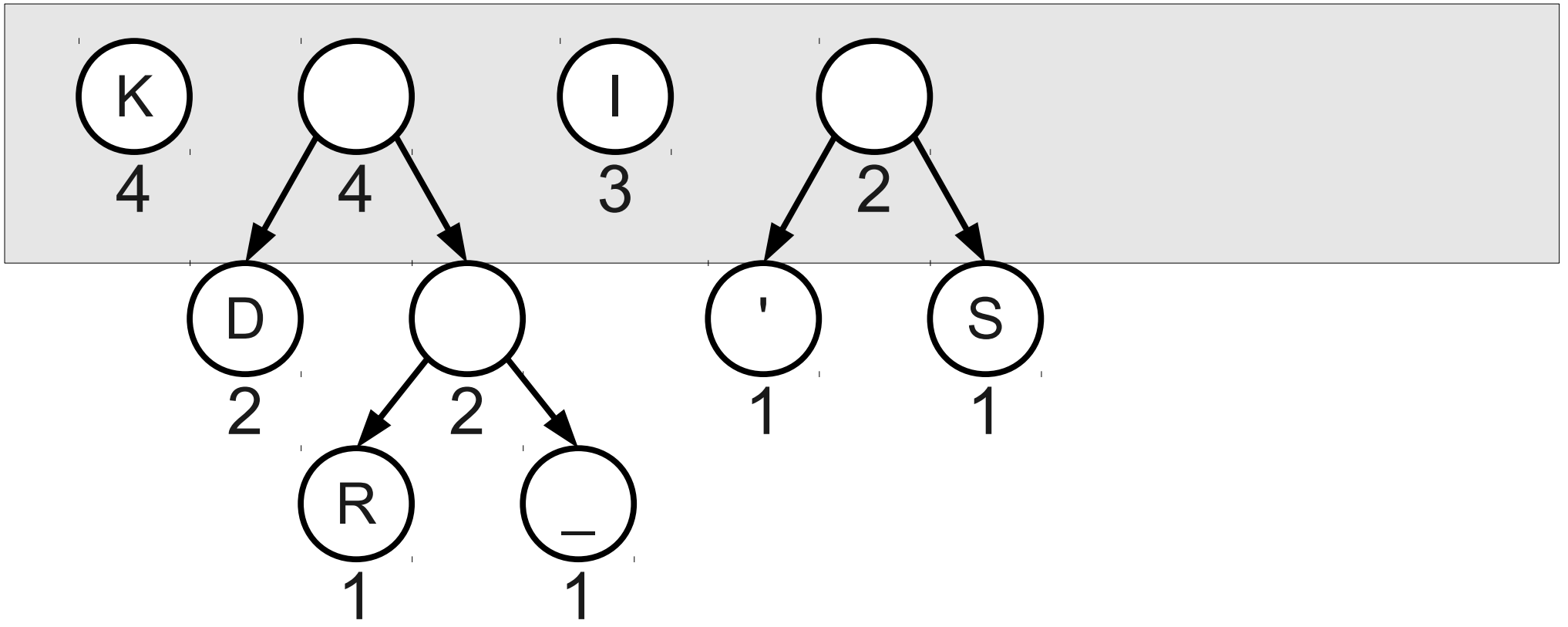
Huffman Coding



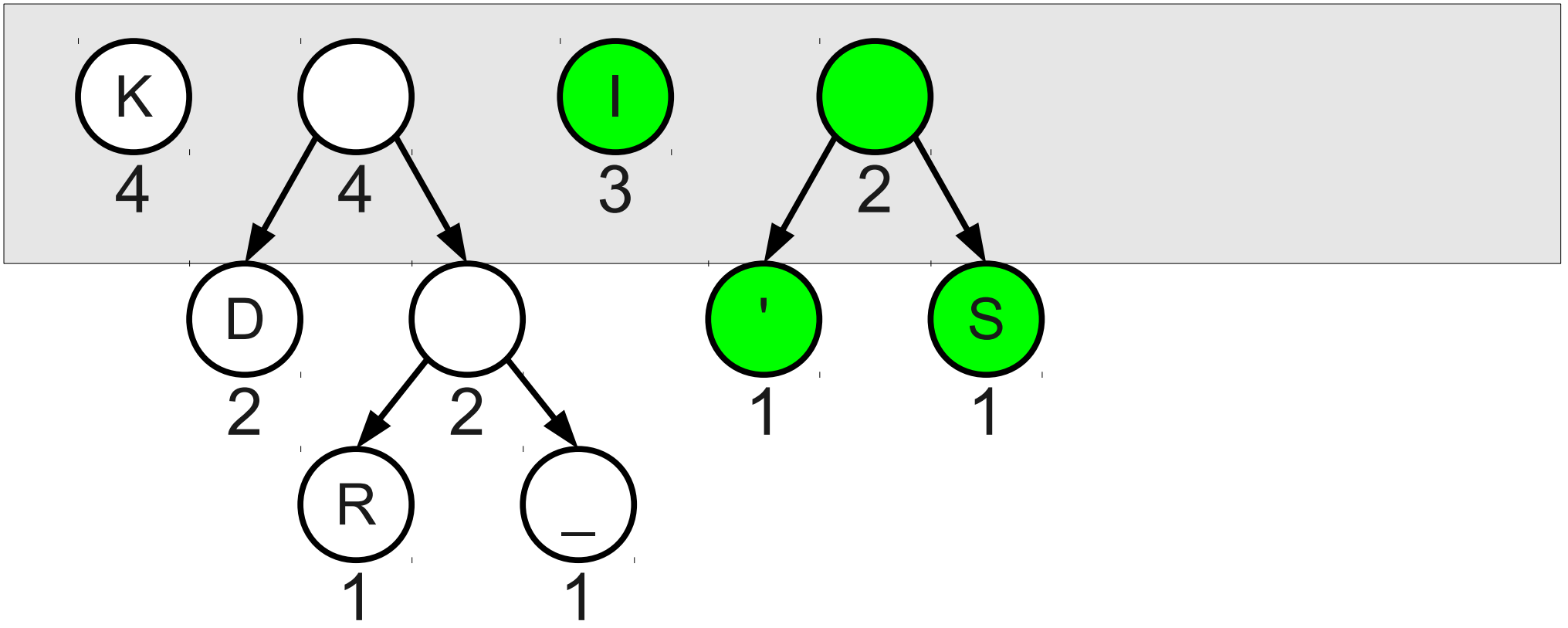
Huffman Coding



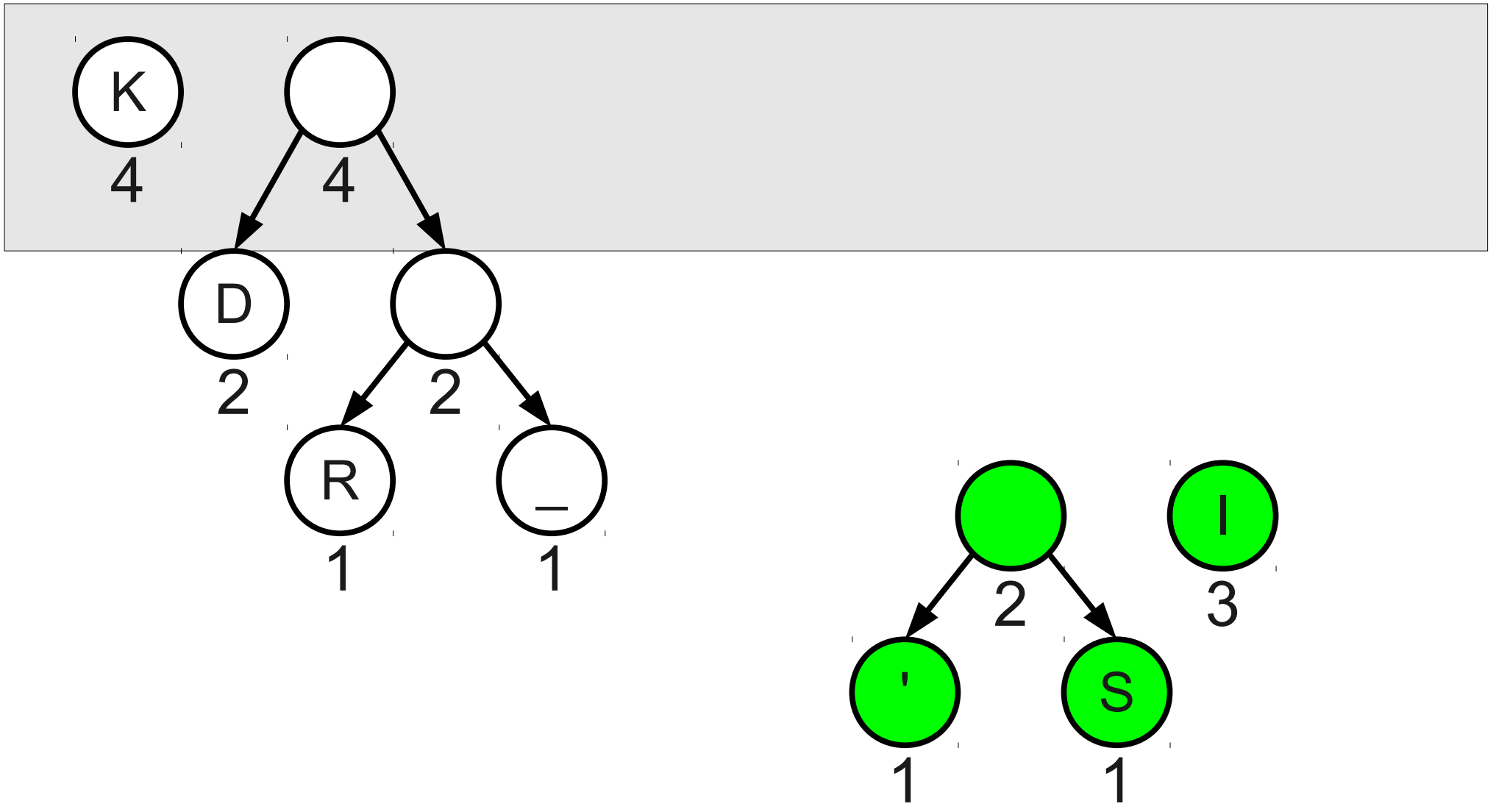
Huffman Coding



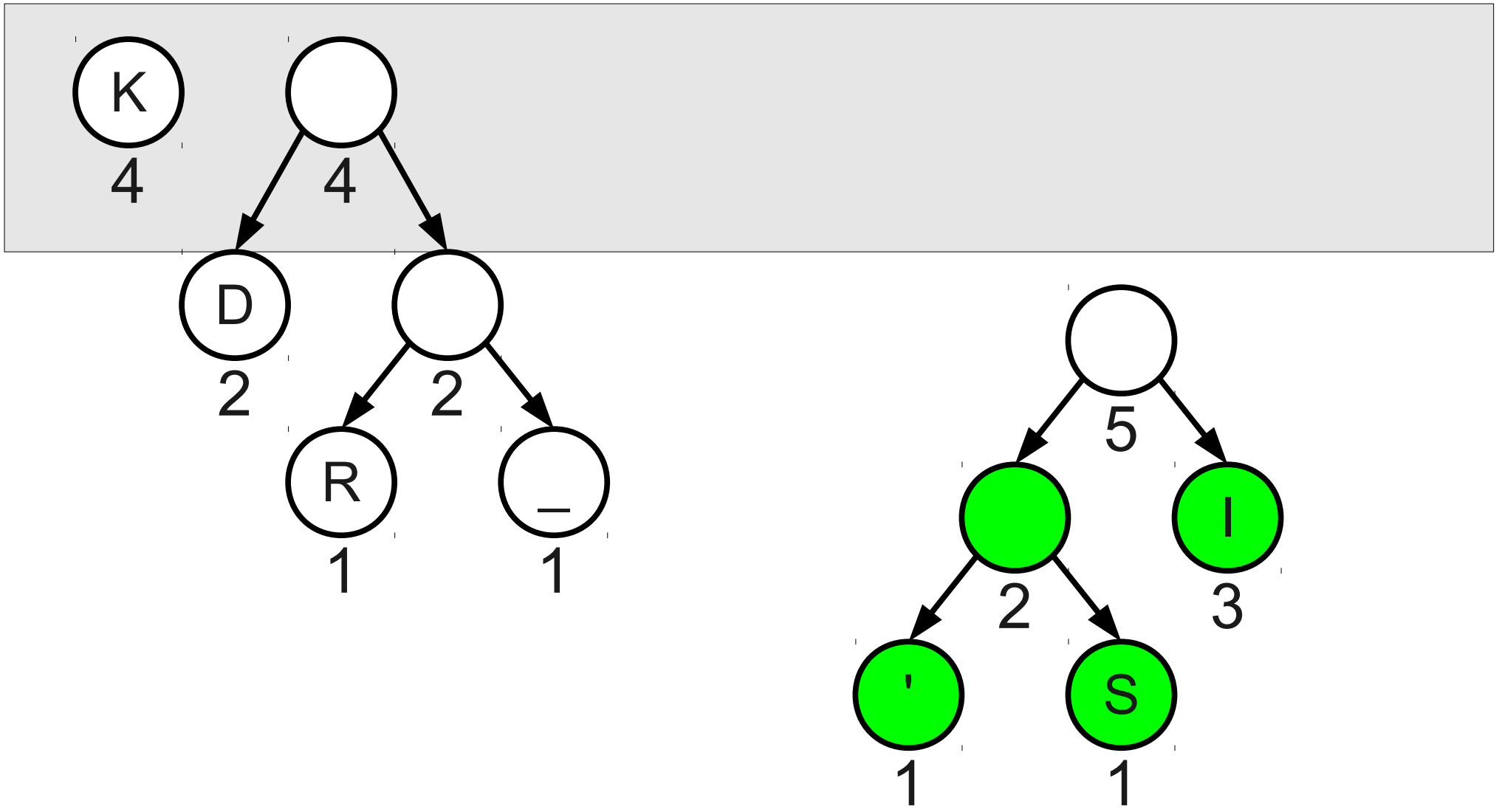
Huffman Coding



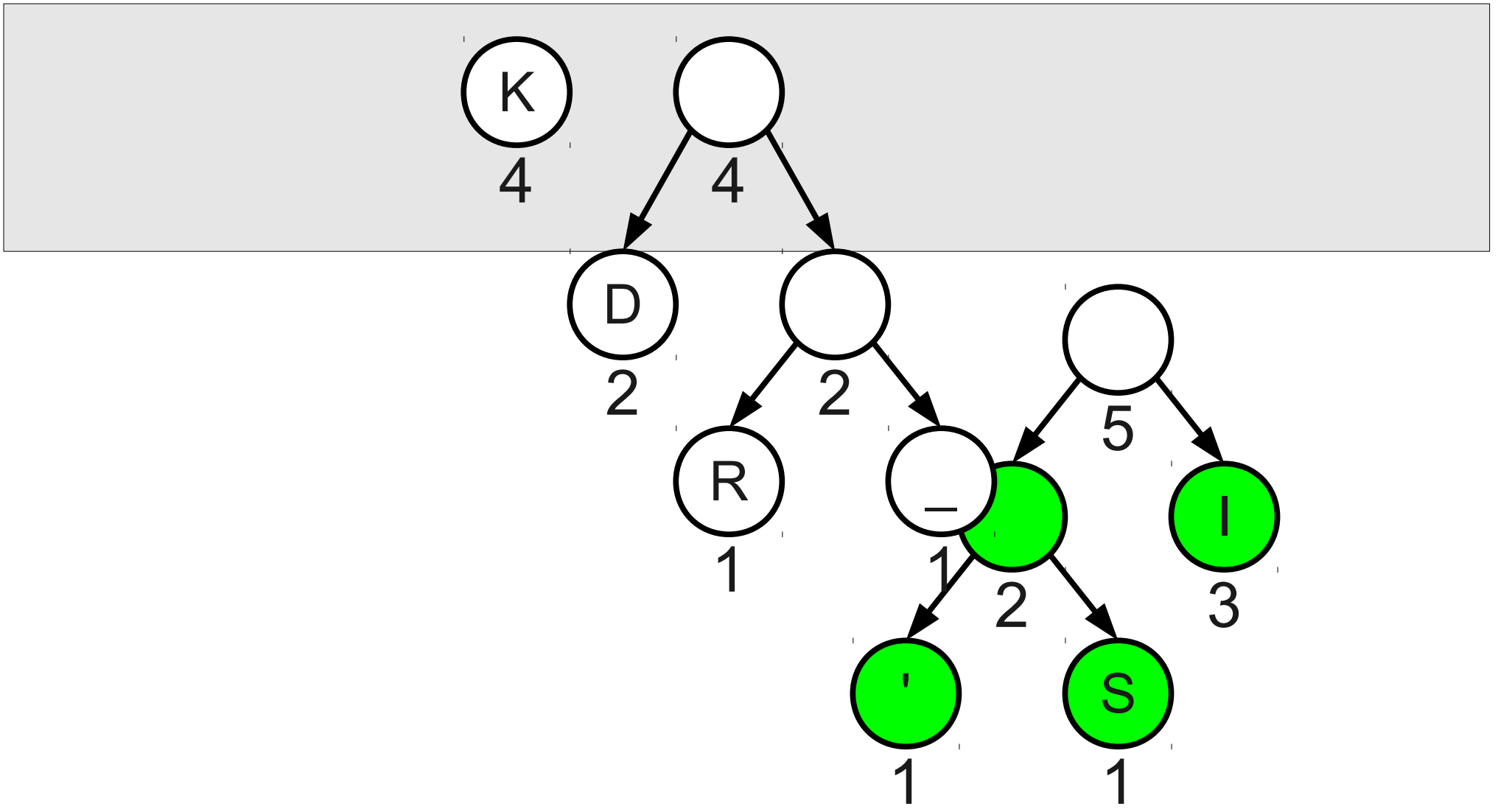
Huffman Coding



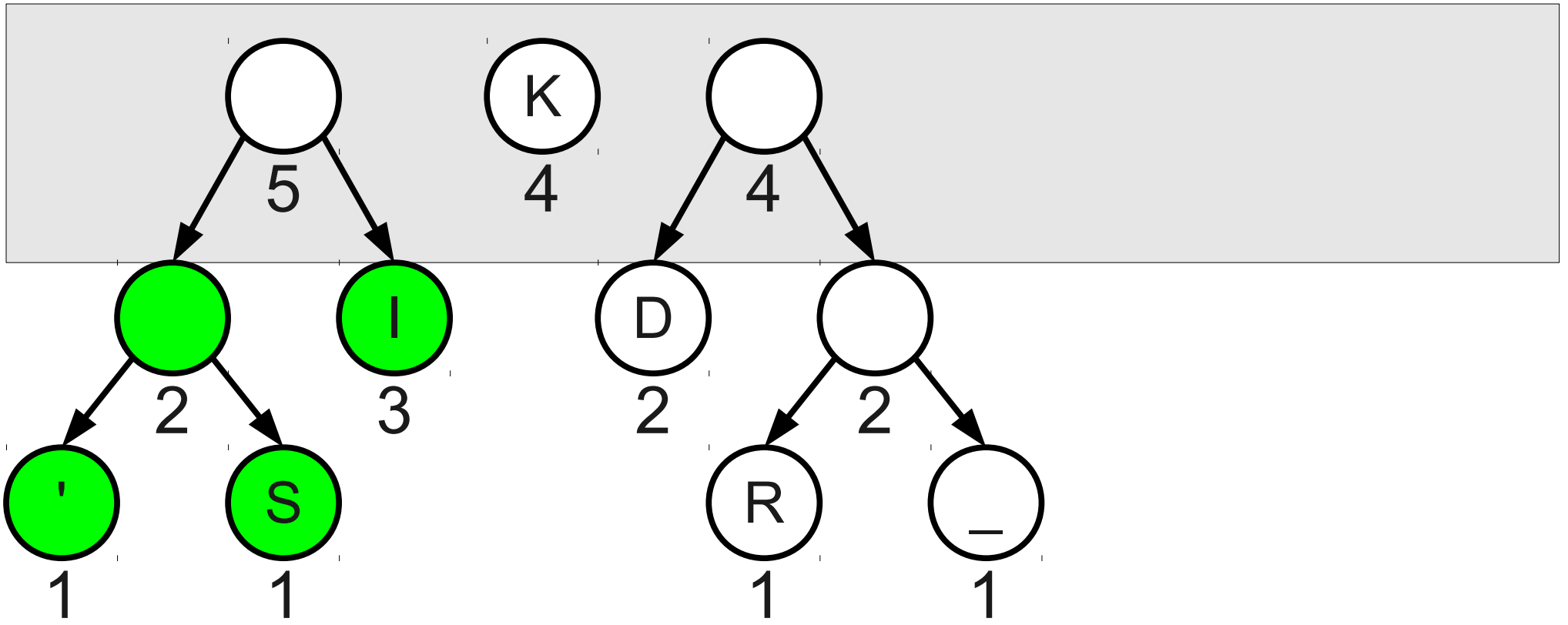
Huffman Coding



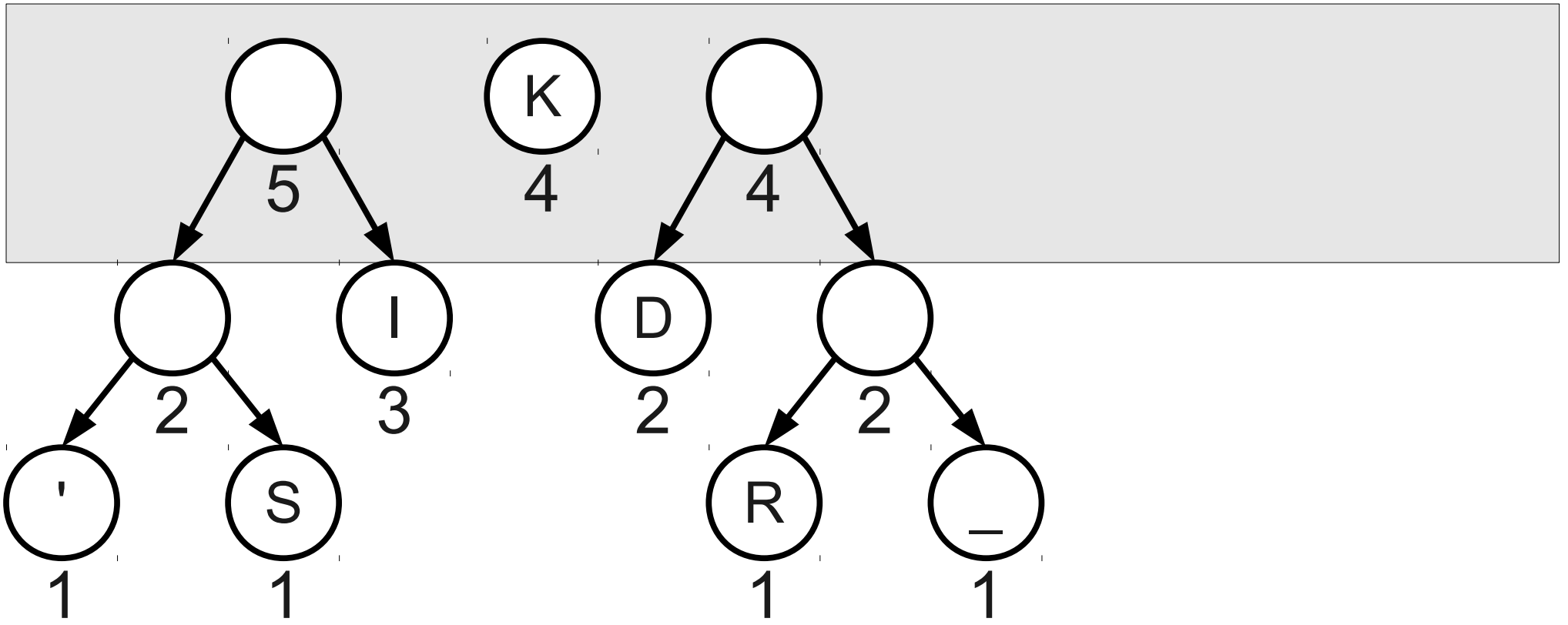
Huffman Coding



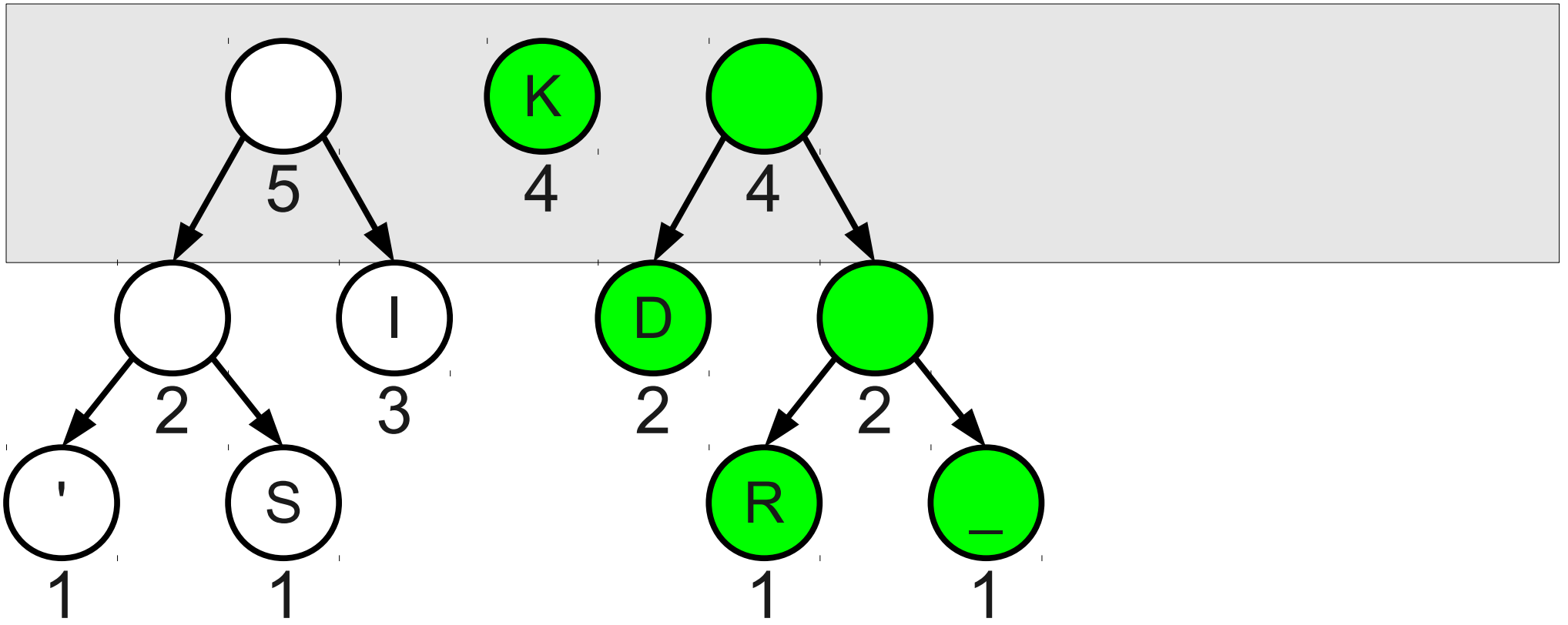
Huffman Coding



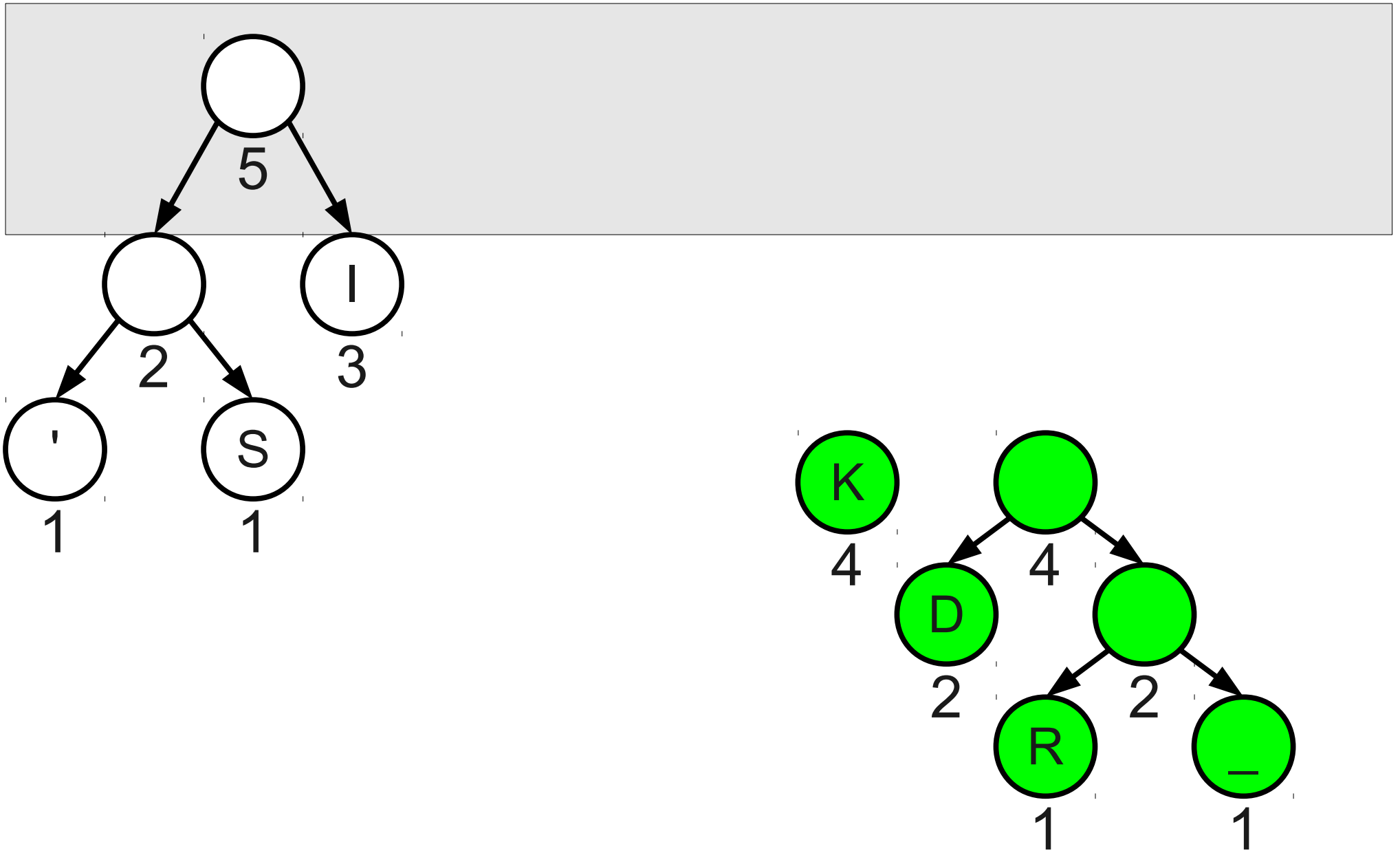
Huffman Coding



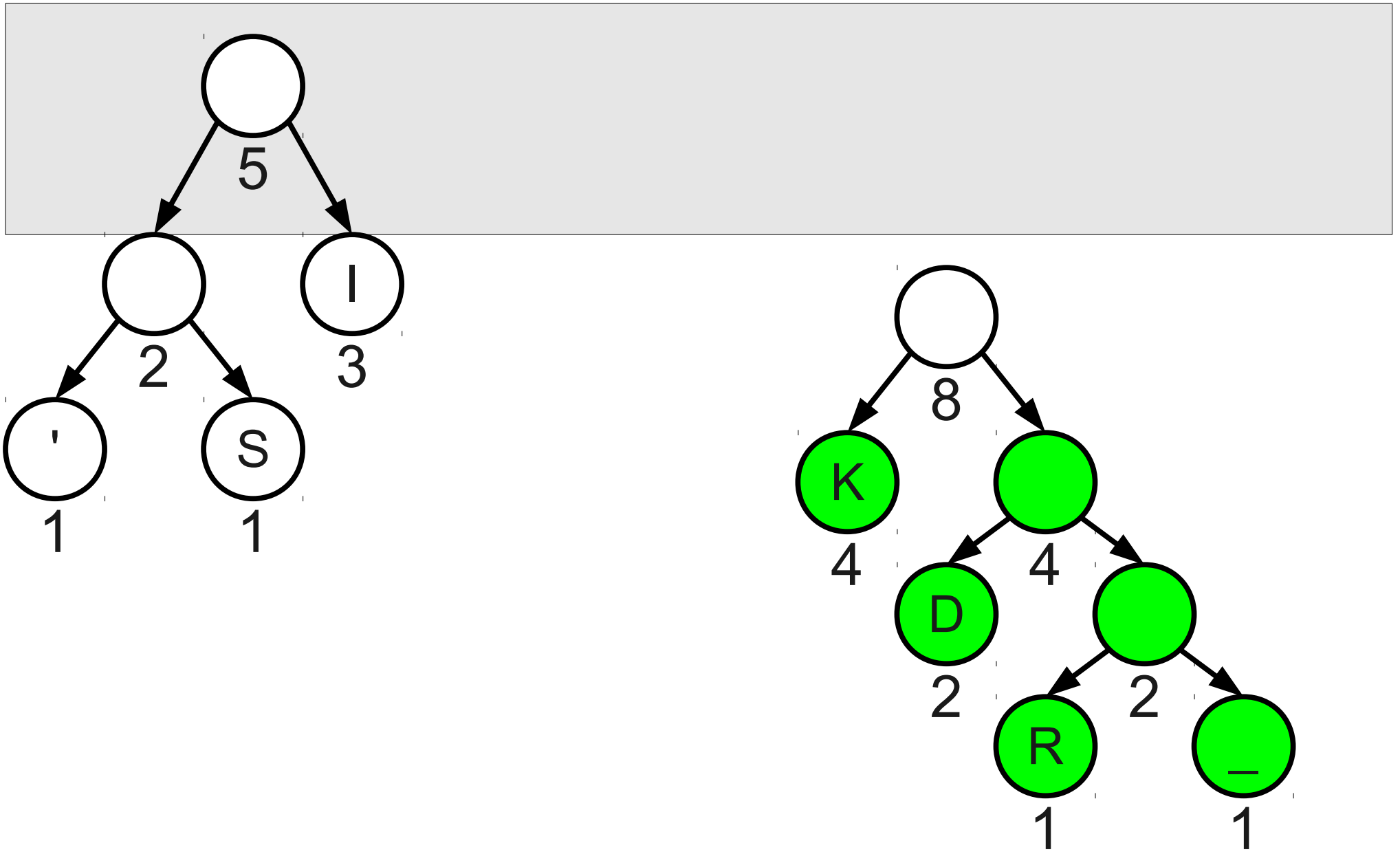
Huffman Coding



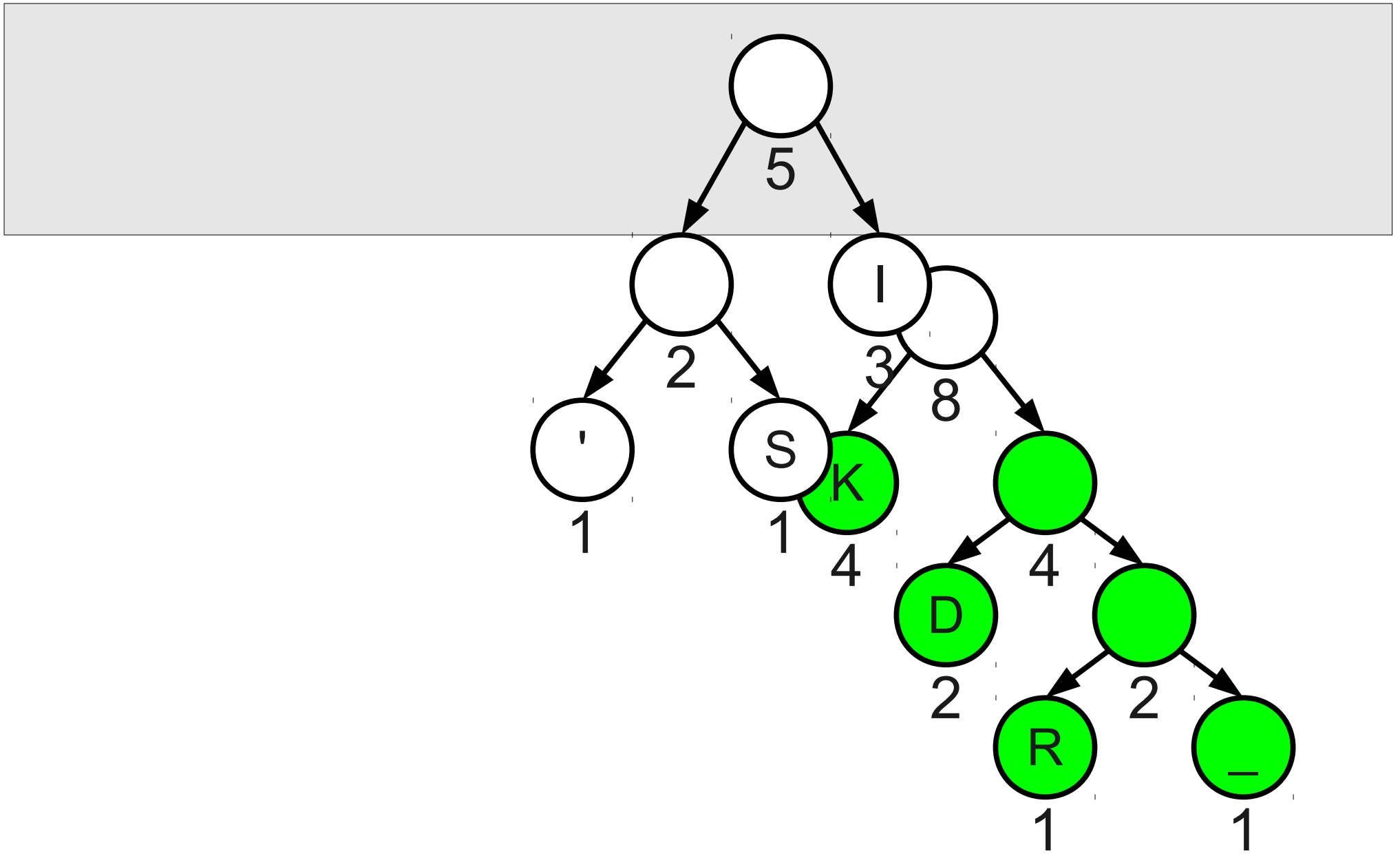
Huffman Coding



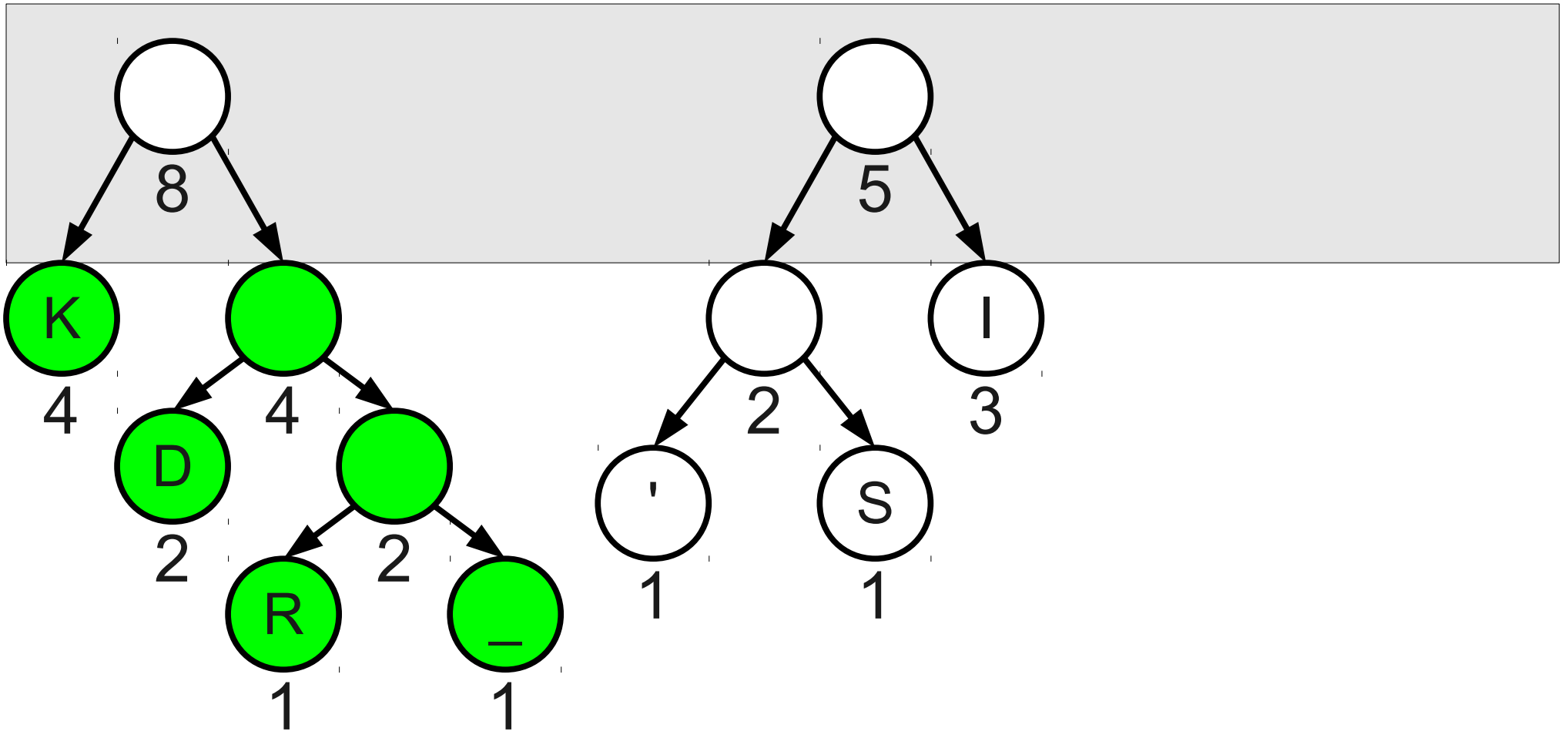
Huffman Coding



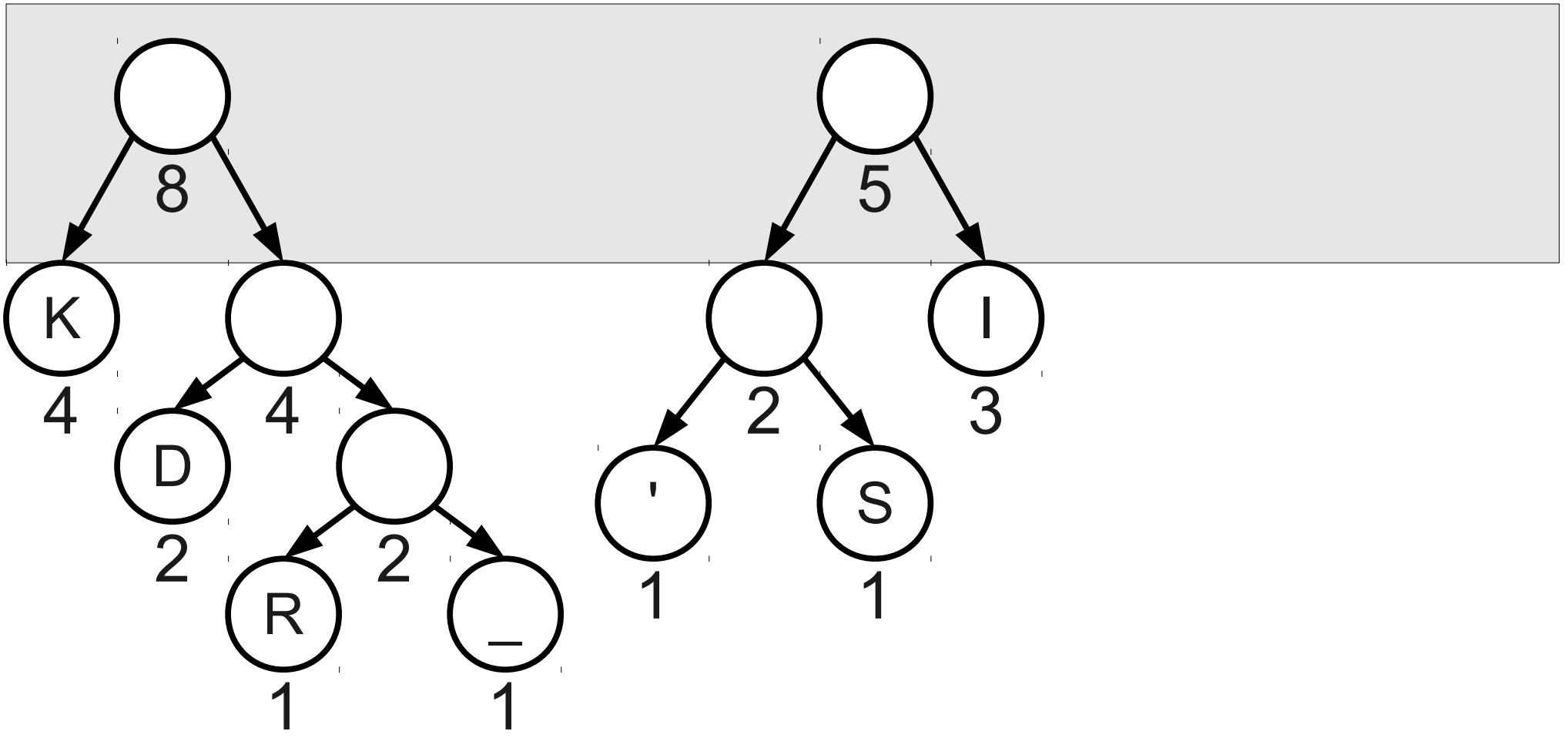
Huffman Coding



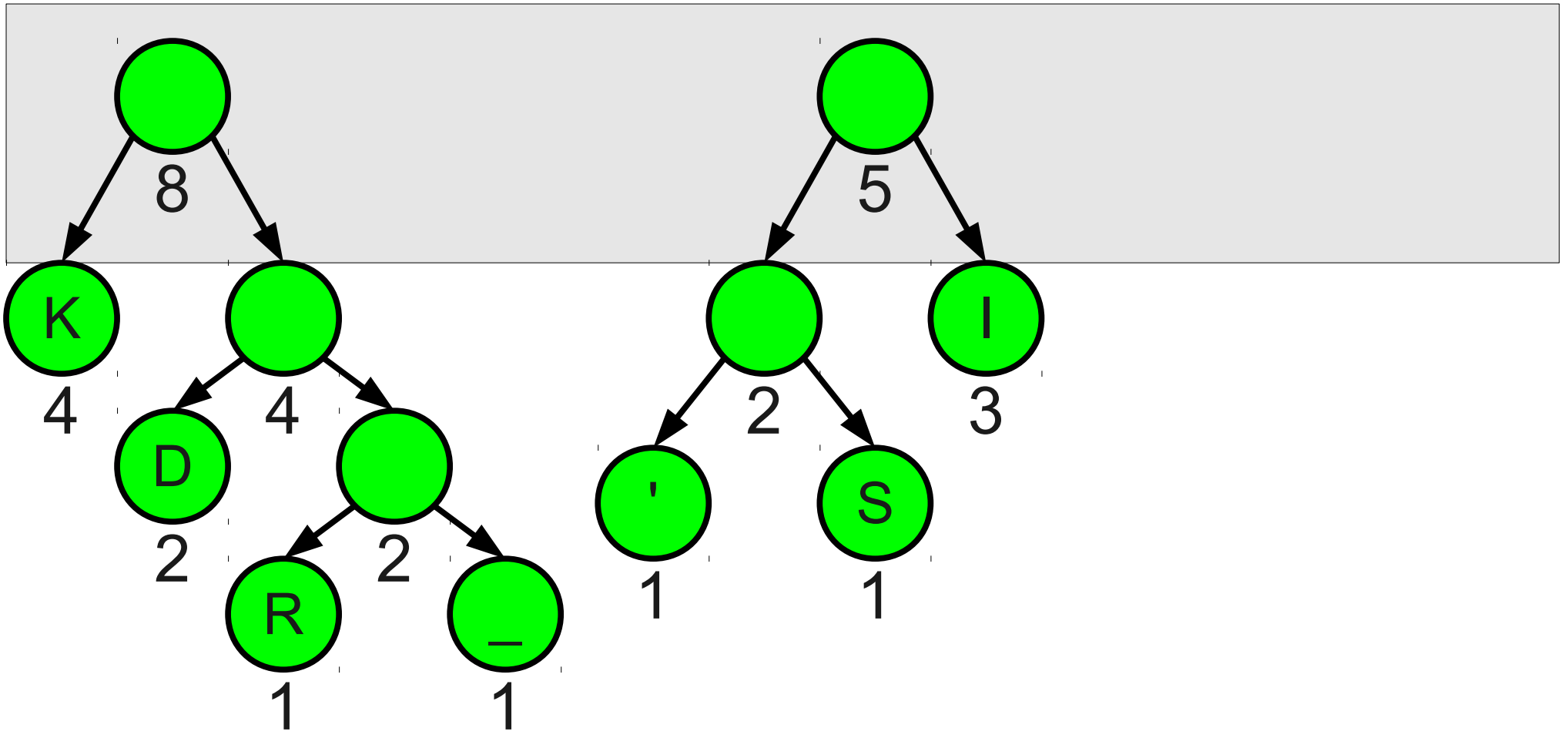
Huffman Coding



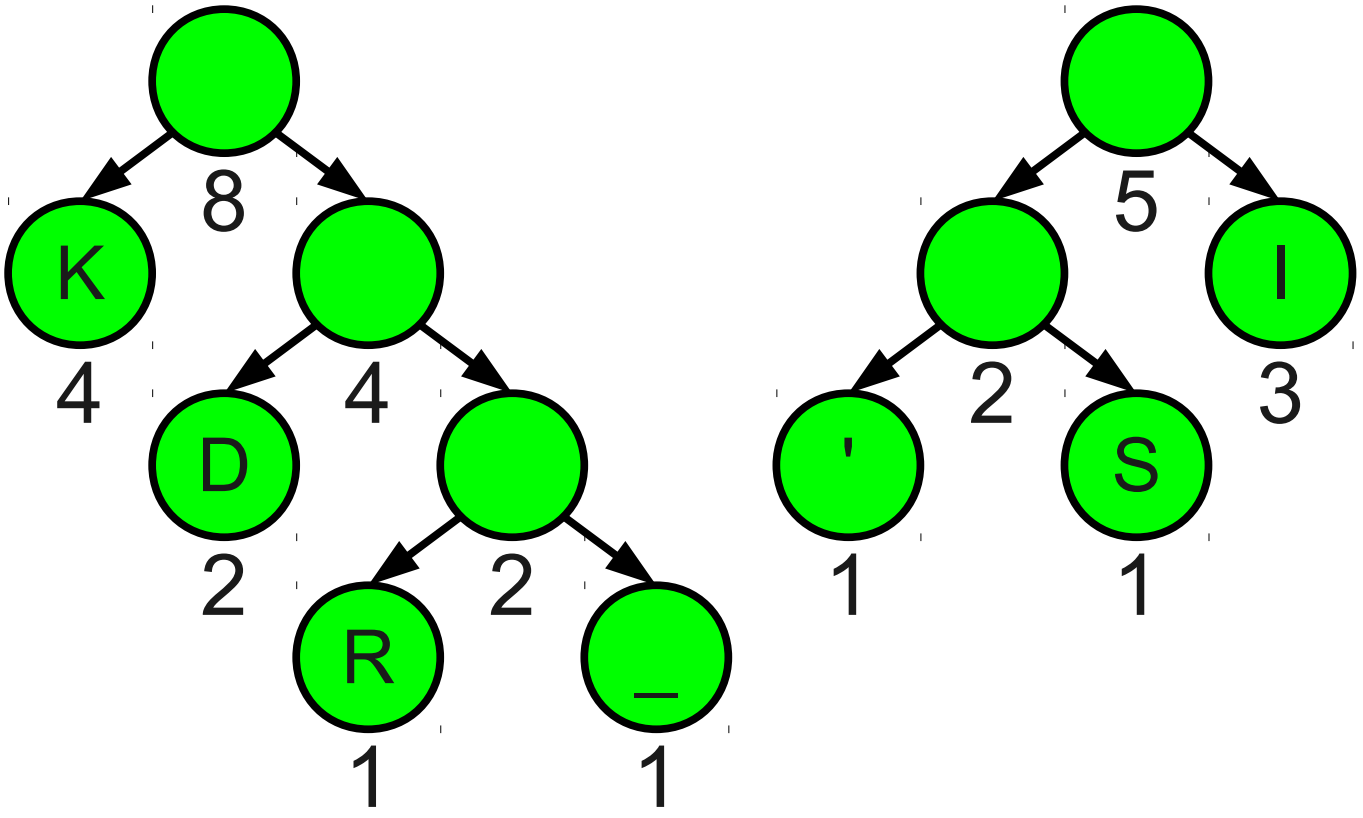
Huffman Coding



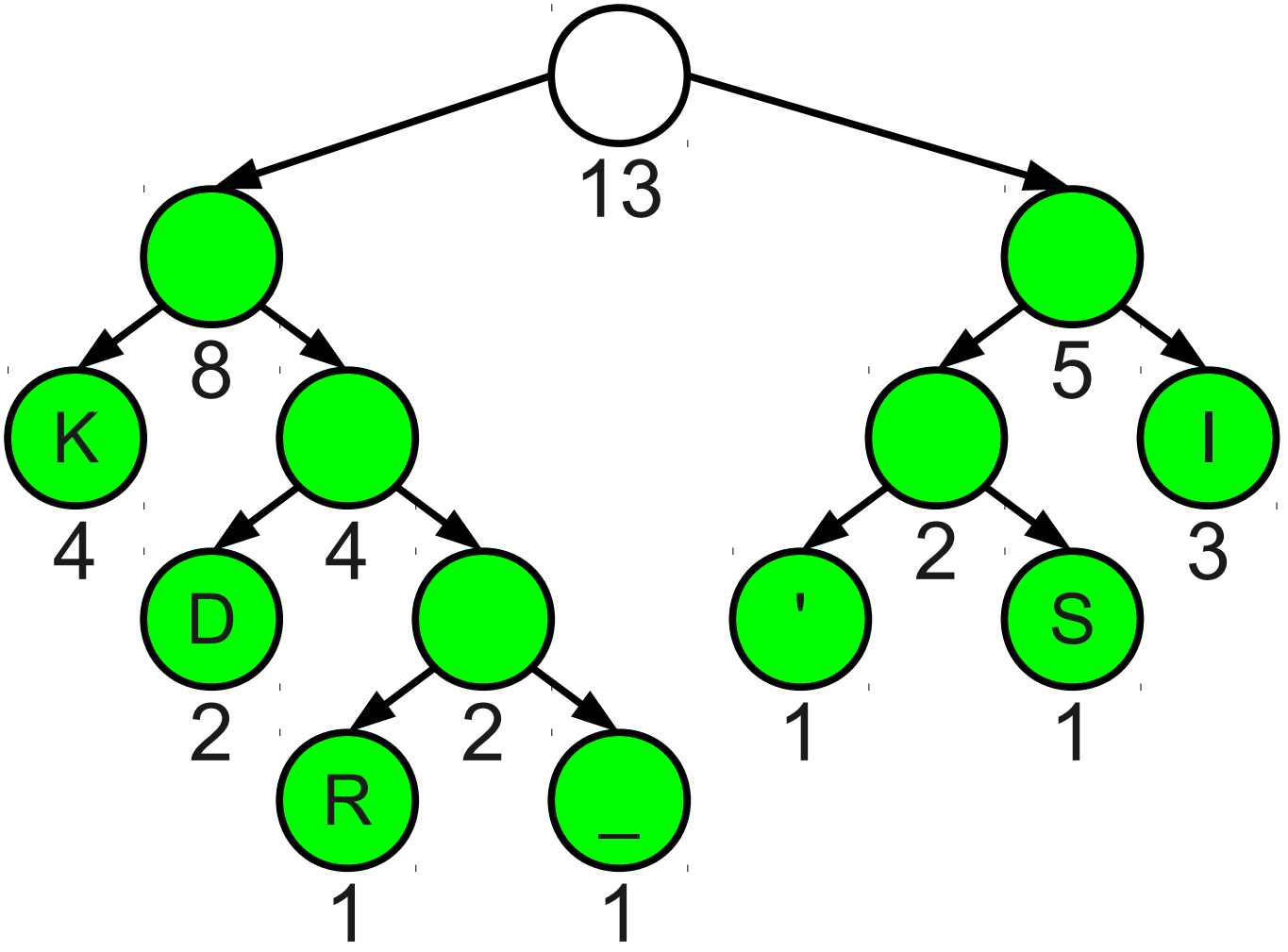
Huffman Coding



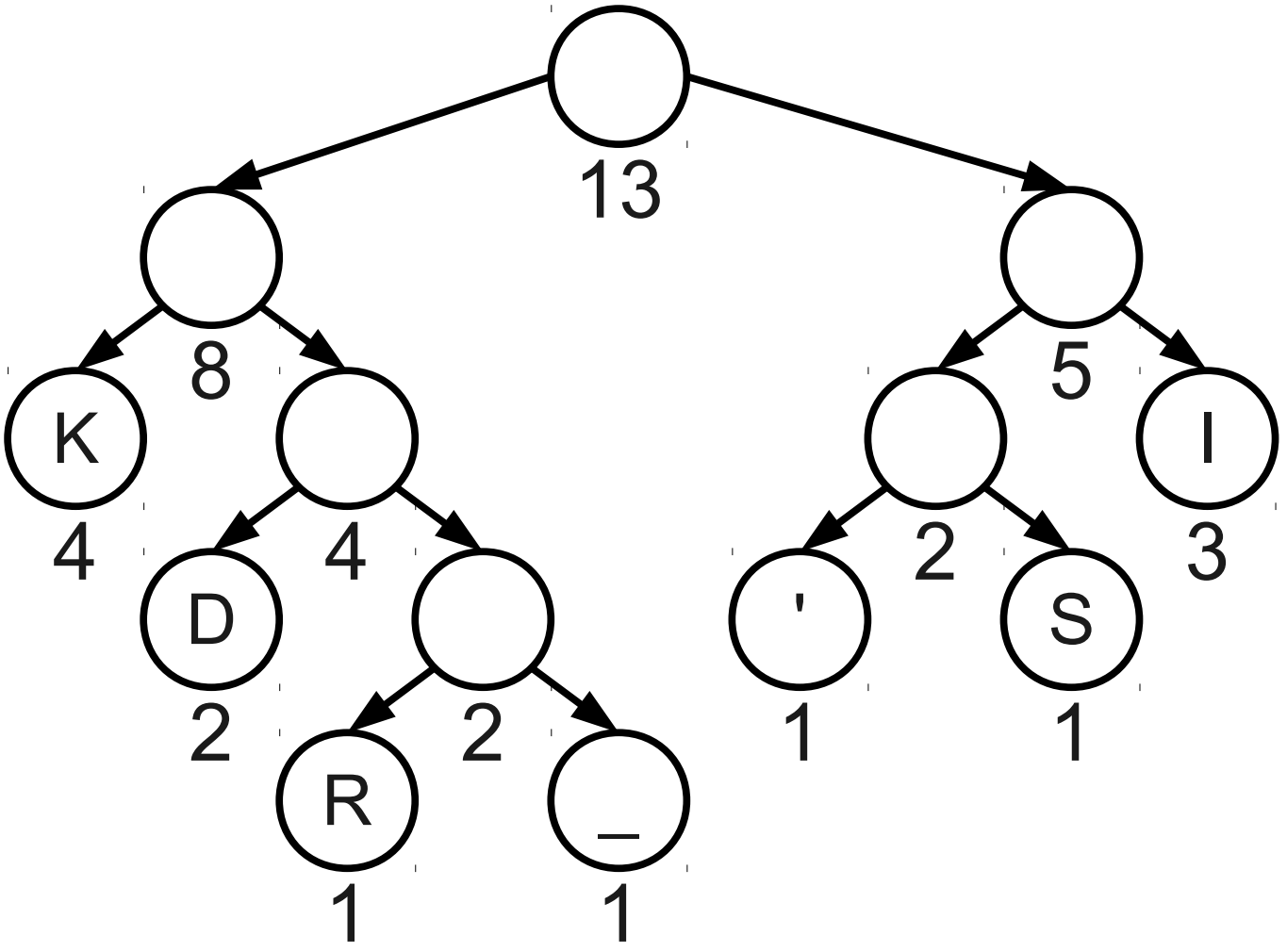
Huffman Coding



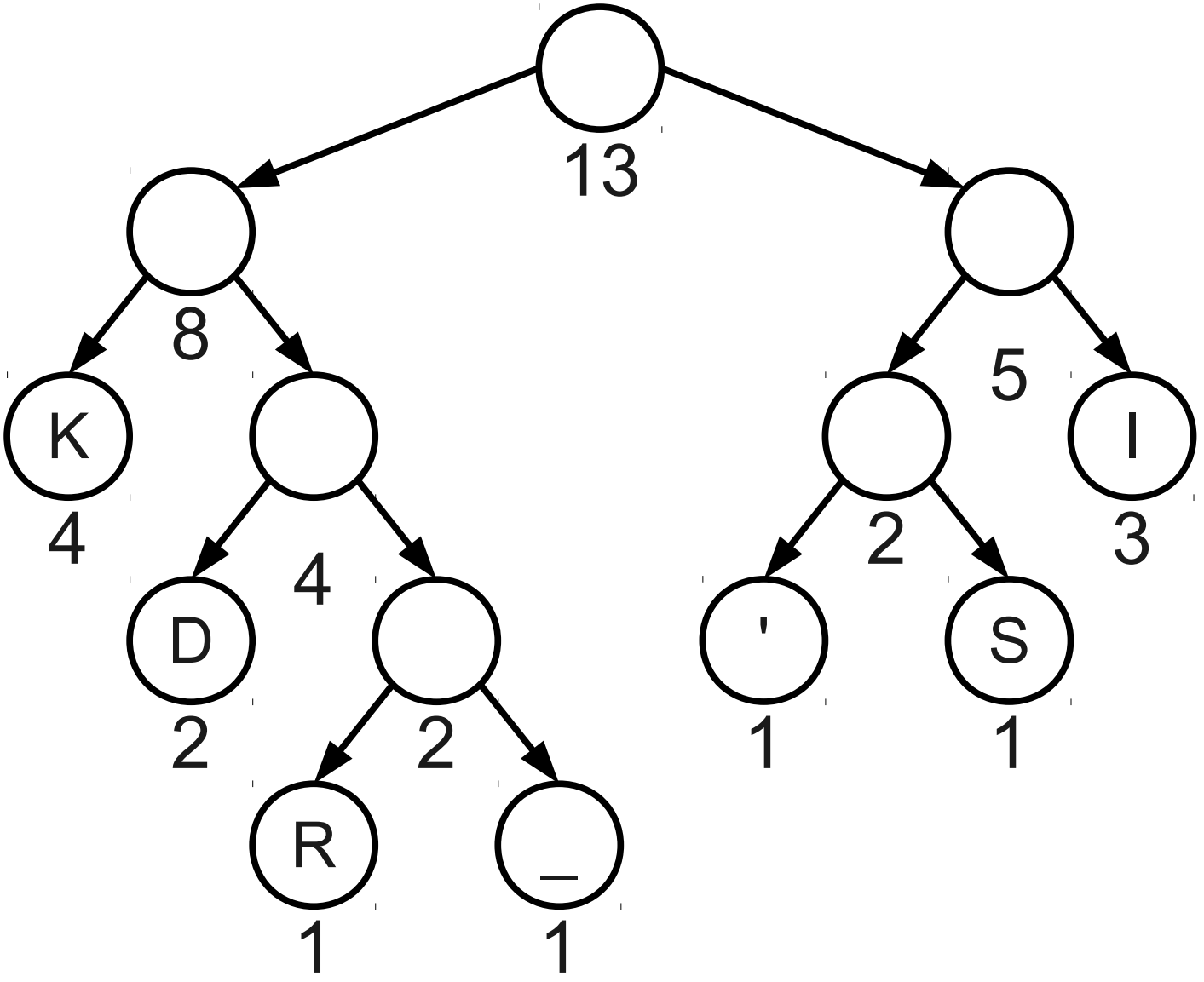
Huffman Coding



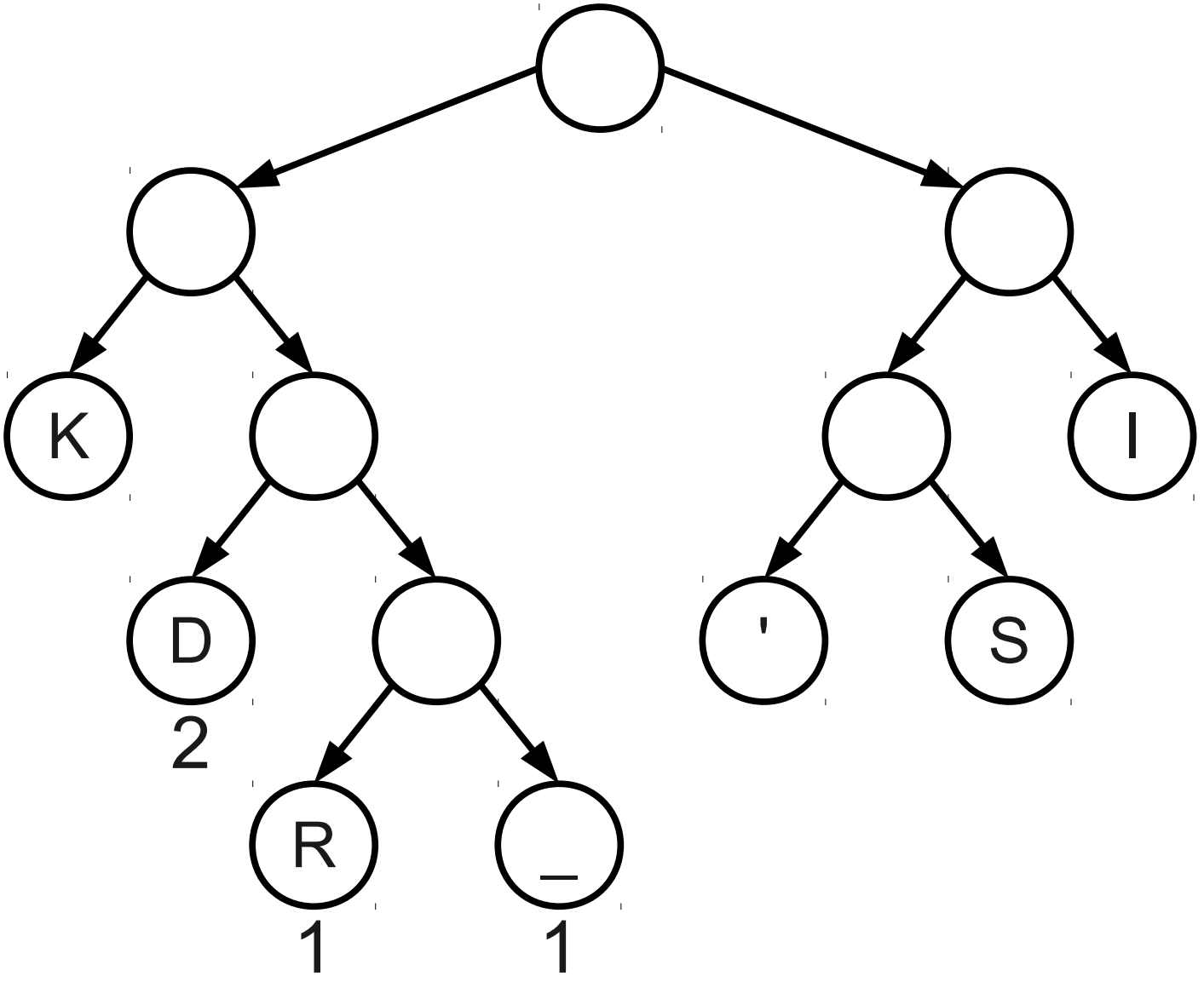
Huffman Coding



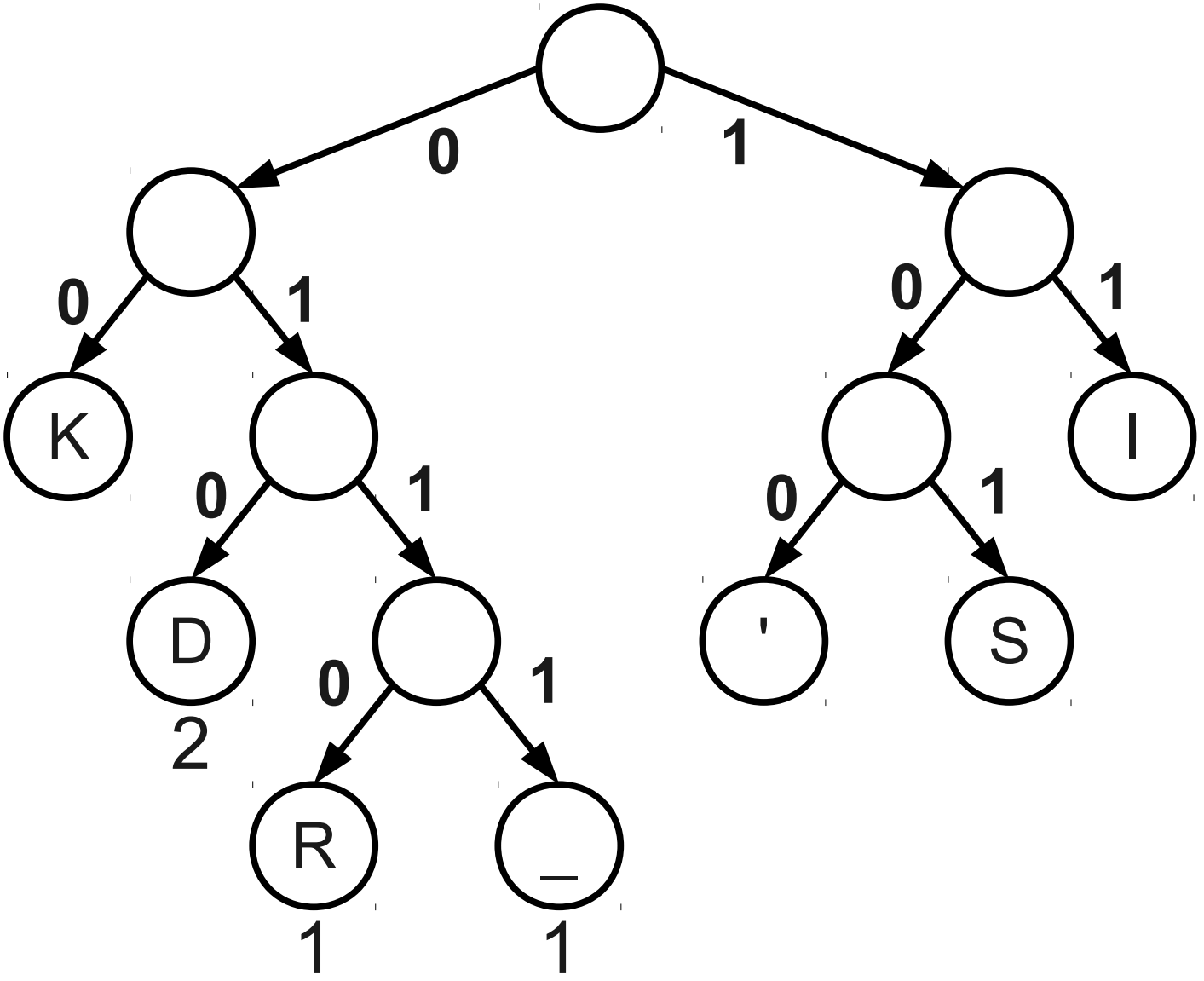
Huffman Coding



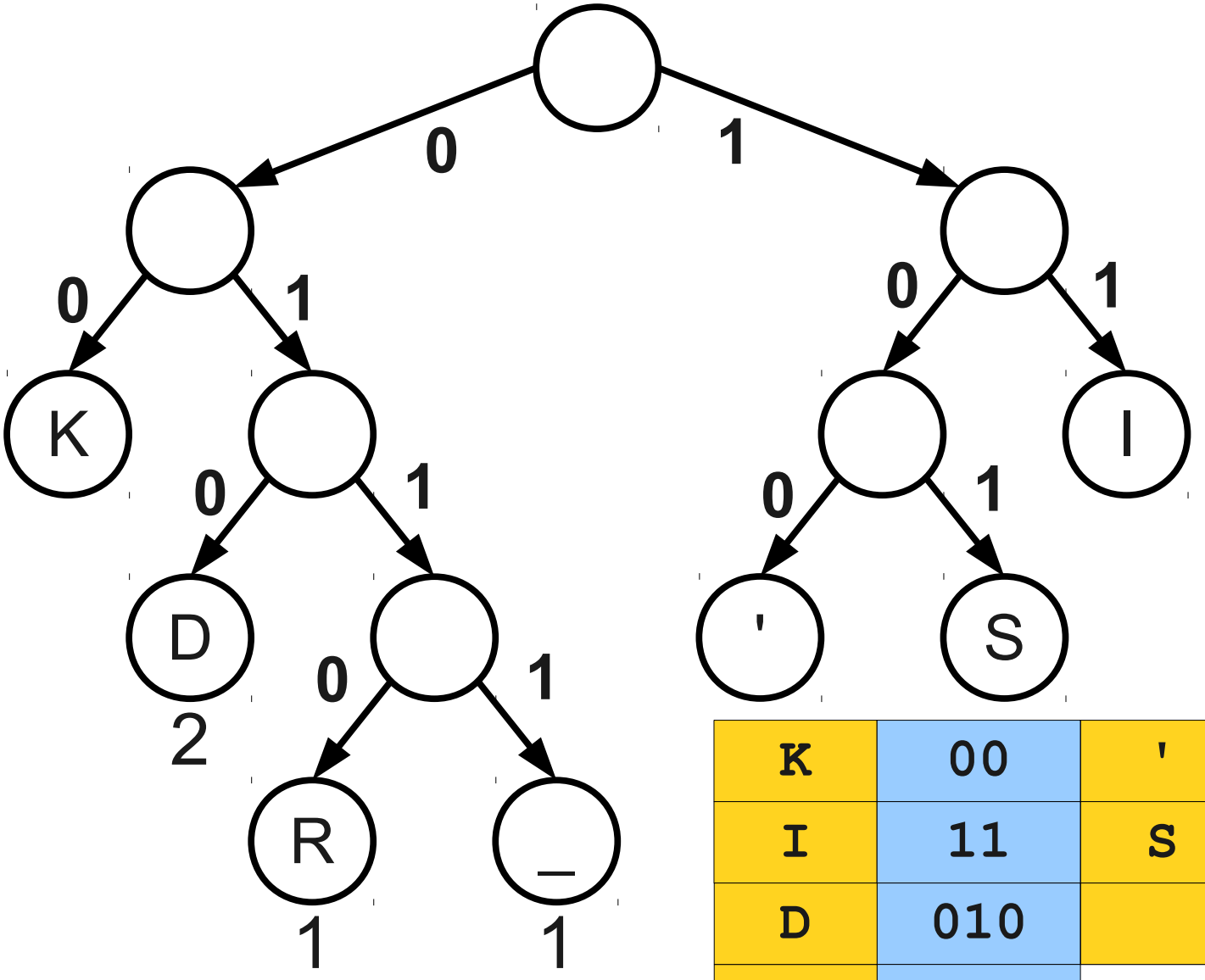
Huffman Coding



Huffman Coding



Huffman Coding



K	00	'	100
I	11	S	101
D	010		0111
R	0110		

Two Important Details

Prefix Codes

K	10
I	01
D	110
R	1111
'	001
S	000
	1110

1001111110001000111011001101100110

10	01	1111	10	001	000	1110	110	01	10	110	01	10
K	I	R	K	'	S		D	I	K	D	I	K

Prefix Codes

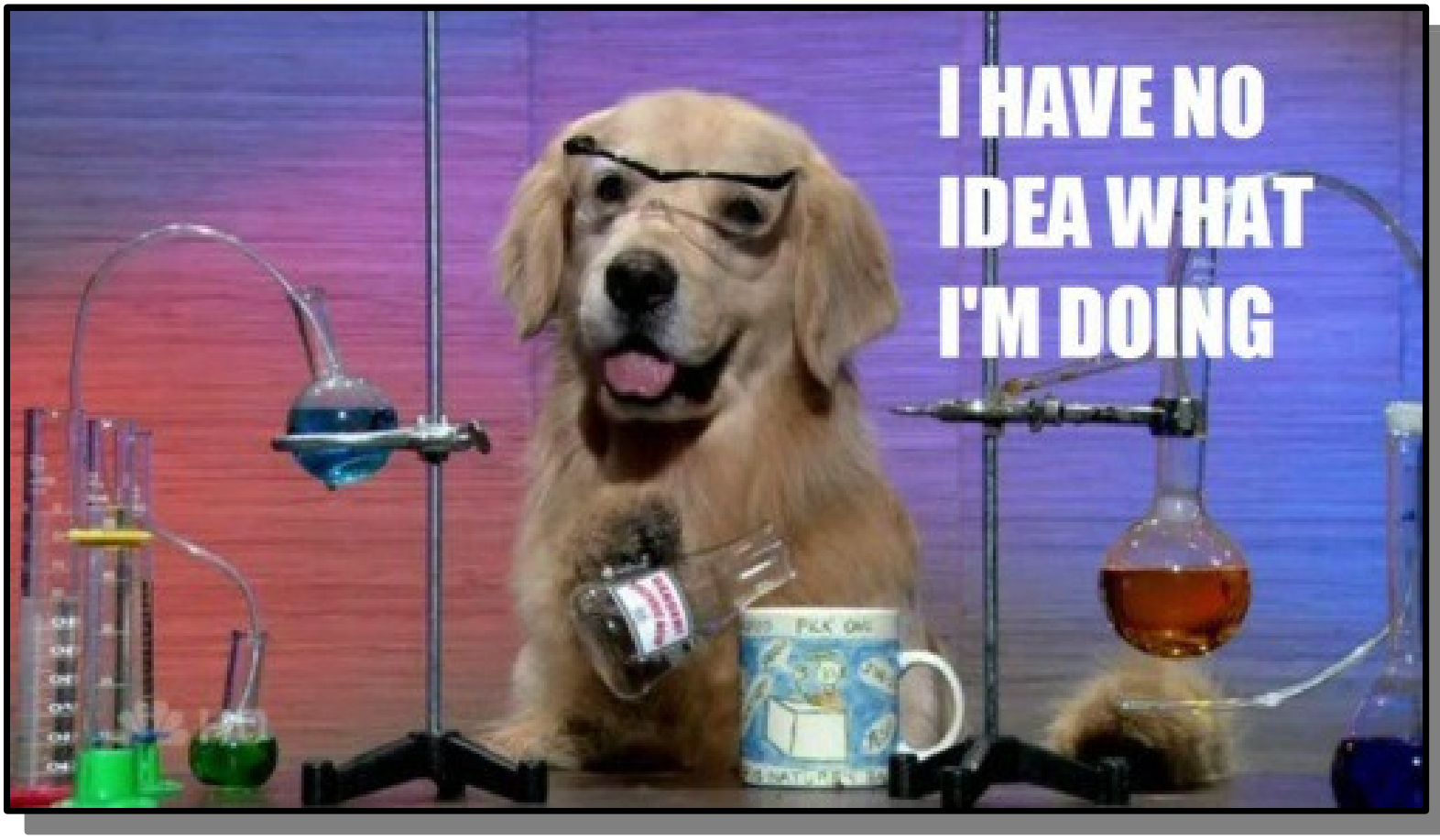
K	10
I	01
D	110
R	1111
'	001
S	000
	1110

```
10011111100010001110110011011001110
```

Prefix Codes

```
10011111100010001110110011011001110
```

Prefix Codes



100110

Transmitting the Tree

- In order to decompress the text, we have to remember what encoding we used!
- Idea: Prefix the compressed data with a **header** containing enough information to rebuild the table.

Encoding information	1101110010111011110001001101010111100

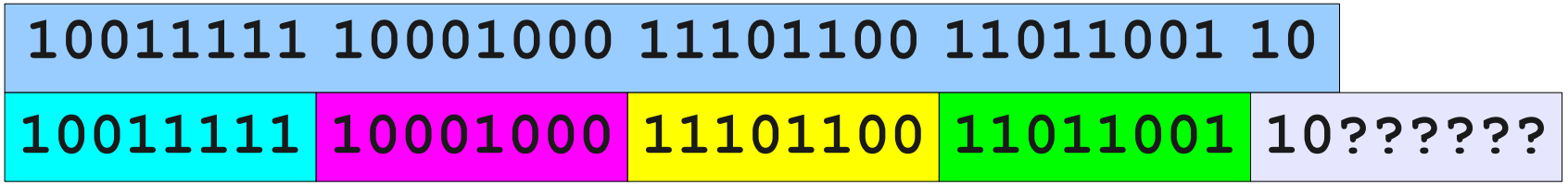
- This might increase the total file size!
- **Theorem**: There is no compression algorithm that can always compress all inputs.

One Last Thing...

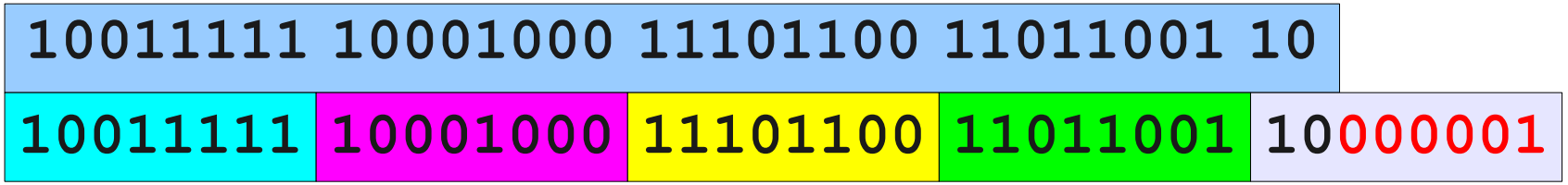
Bitten by Bytes

```
10011111100010001110110011011001l0
```

Bitten by Bytes



Bitten by Bytes



Bitten by Bytes

10011111	10001000	11101100	11011001	10000001

K	10
I	01
D	110
R	1111

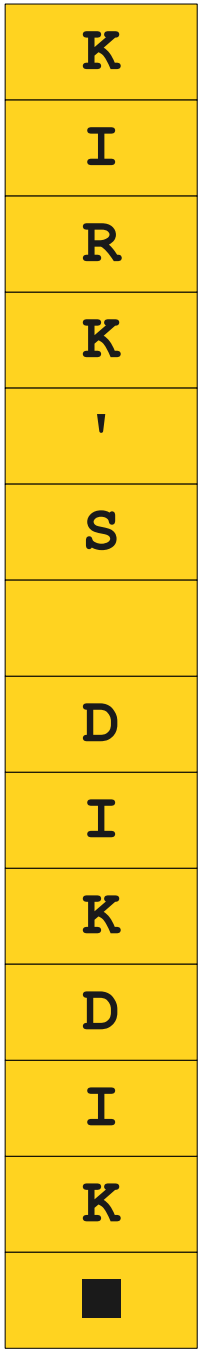
'	001
S	000
	1110

10	01	1111	10	001	000	1110	110	01	10	110	01	10	000	001
K	I	R	K	'	S		D	I	K	D	I	K	S	'

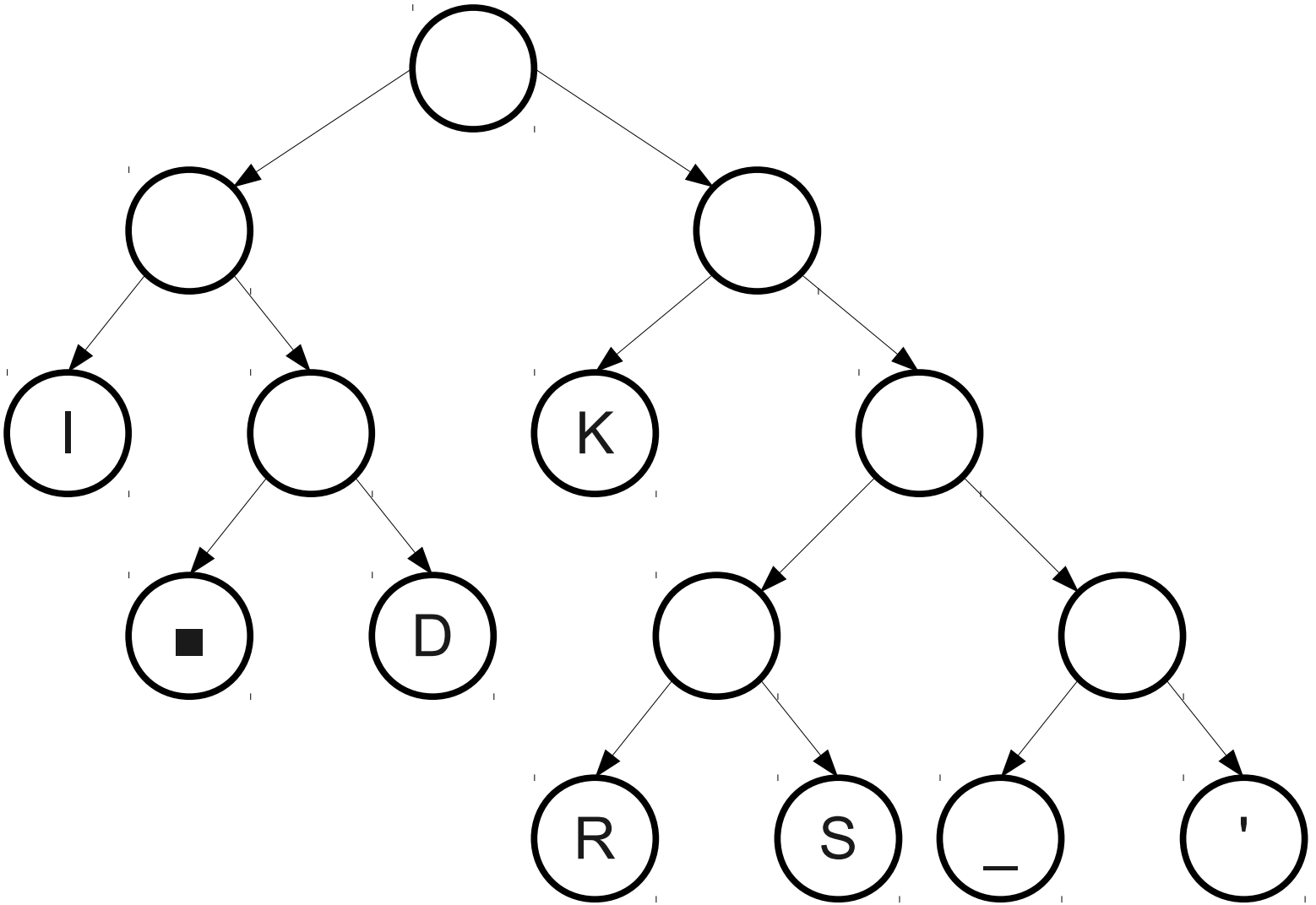
Spare Bits

- The encoded message might not actually use all 8 bits in its final byte.
- All files are stored as bytes, so those last bits will be filled in with garbage.
- If we don't know when to stop, we might decode extra garbage data when decompressing.


STOP

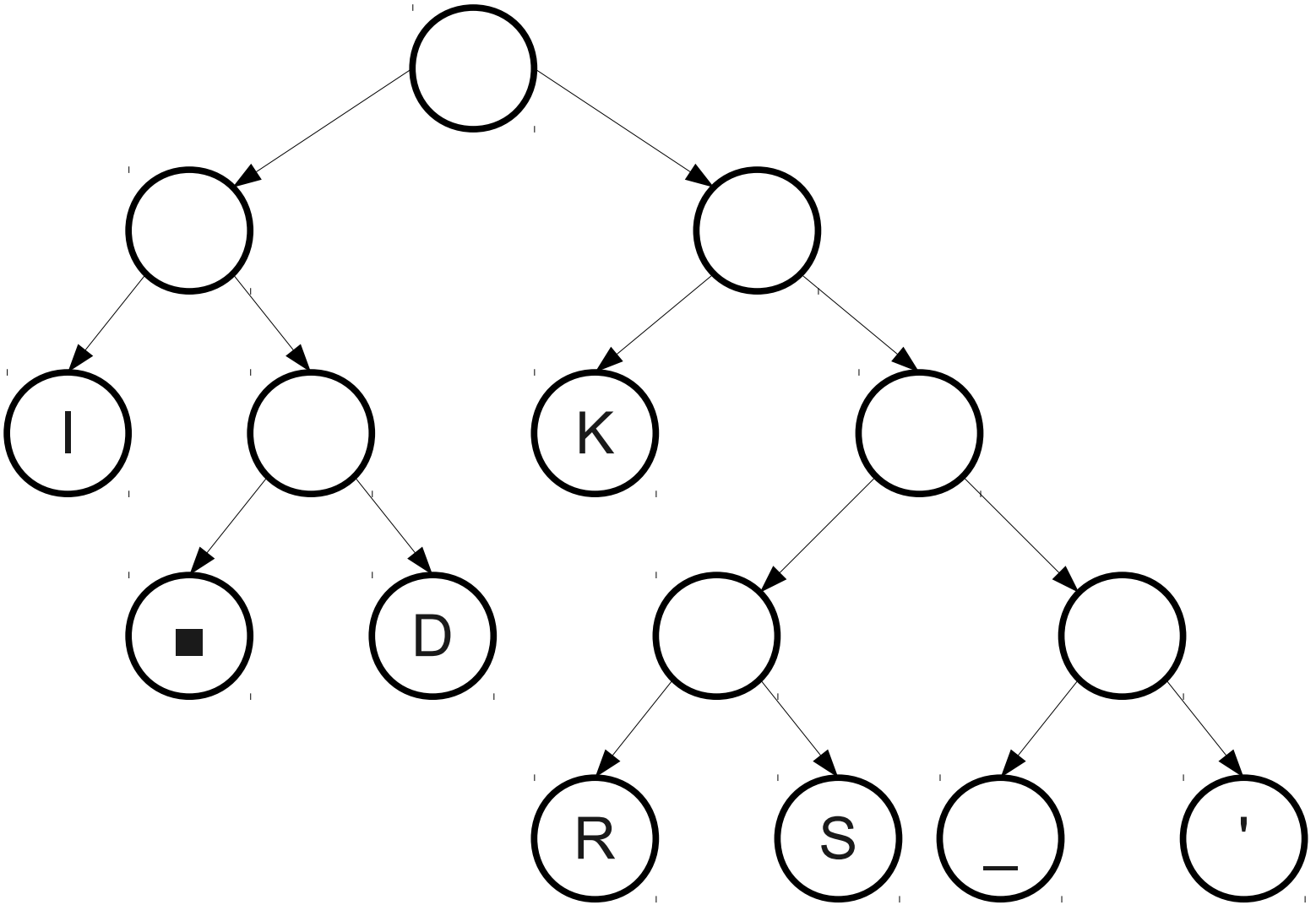


K	4
I	3
D	2
R	1
'	1
S	1
	1
■	1



K	4
I	3
D	2
R	1

'	1
S	1
	1
■	1



K	10
I	00
D	011
R	1100

'	1111
S	1101
	1110
■	010

Once More, With Stops

10	00	1100	10	1111	1101	1110	011	00	10	011	00	10	010
K	I	R	K	'	S		D	I	K	D	I	K	■

10001100 10111111 01111001 10010011 0010010?

K	10
I	00
D	011
R	1100

'	1111
S	1101
	1110
■	010

Pseudo-EOFs

- The marker ■ we inserted is called a **pseudo-end-of-file marker** (or **pseudo-EOF**).
- Indicates where the encoding stops.
- Similar to how RNA and DNA encode proteins – certain codons are reserved for “stop here.”

Summary of Huffman Encoding

- Prefix-free encodings can be modeled as binary tries.
- Huffman encoding uses a greedy algorithm to construct encodings.
- We need to send the encoding table with the compressed message.
- We use a pseudo-EOF as a marker that the end of the bits has been reached.

Beyond ASCII

- If you just want to store ASCII text (English characters, digits, etc.), then one byte per character suffices.
- What if you want to store non-Latin characters or more general symbols?

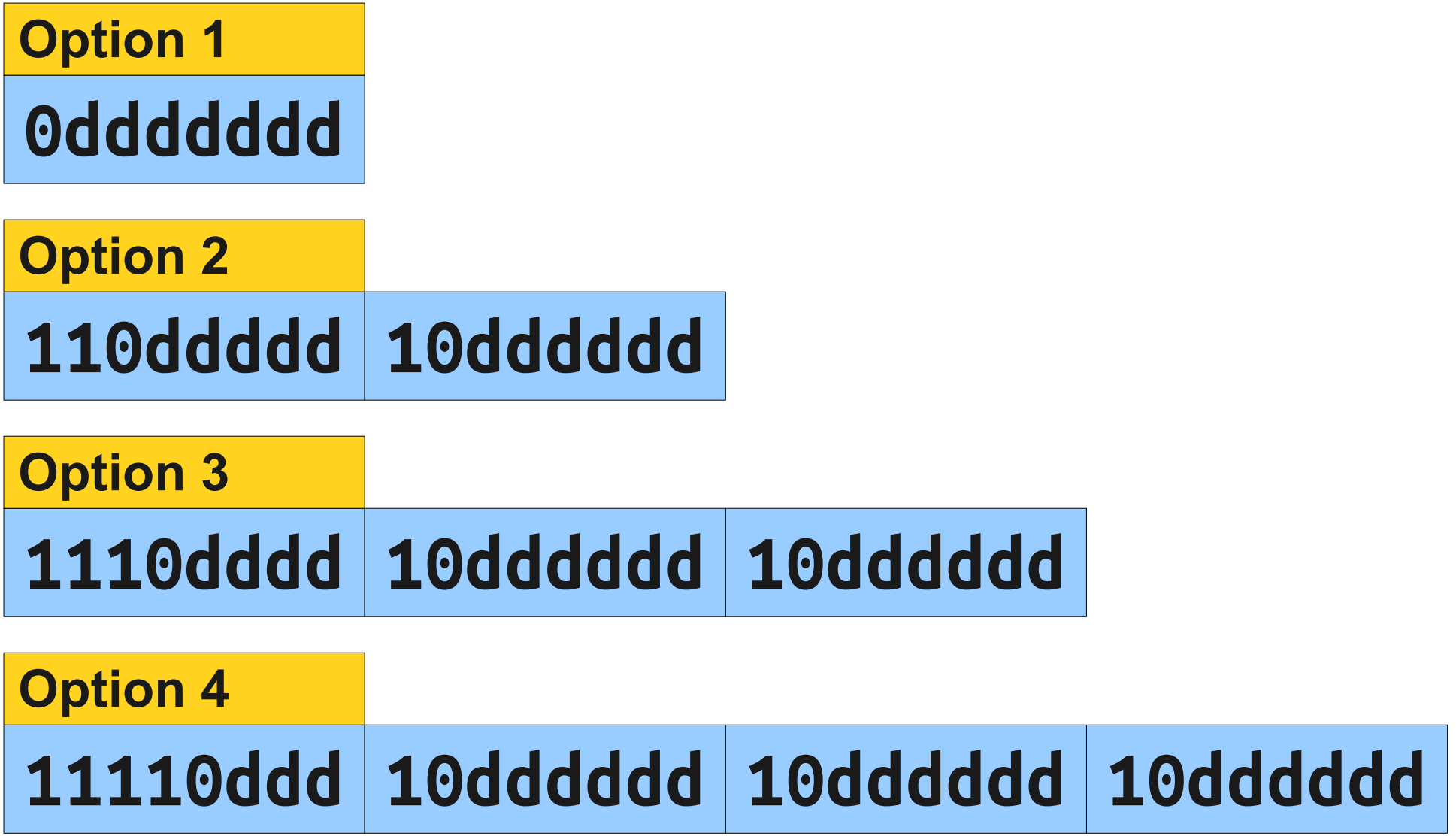
Beyond ASCII

- If you just want to store ASCII text (English characters, digits, etc.), then one byte per character suffices.
- What if you want to store non-Latin characters or more general symbols?
- For example:
  - ¿Cómo estás?
  - السلام عليكم
  - (ﾉ°□°)ﾉ ⁔ ┻┻)

Unicode

- **Unicode** is a system for representing glyphs and symbols from all languages and disciplines.
- Uses a two-level encoding system:
  - Each glyph has a **code point** (a number) associated with it.
  - The code points are then represented using one of several variable-length encodings.

UTF-8


Option 1
0ddddddd
Option 2
110ddddd
10dddddd
Option 3
1110dddd
10dddddd
10dddddd
Option 4
11110ddd
10dddddd
10dddddd
10dddddd

UTF-8

```
11100000101100101010000001011111111000001011001010100000
```

UTF-8

```
11100000 10110010 10100000 01011111 11100000 10110010 10100000
```

UTF-8

11100000 10110010 10100000 01011111 11100000 10110010 10100000

UTF-8

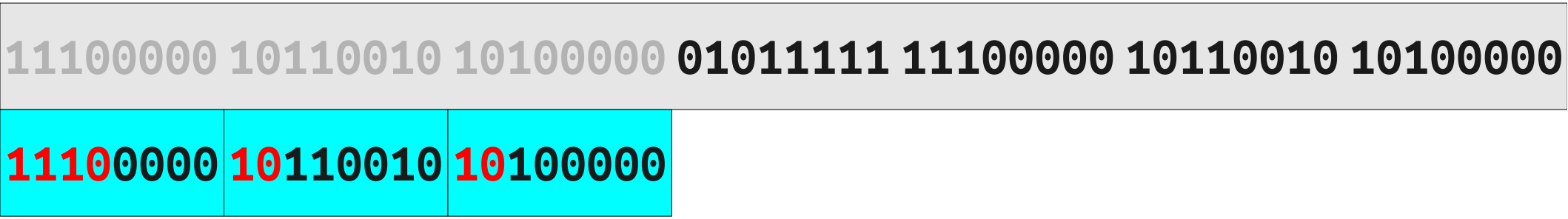
```
11100000 10110010 10100000 01011111 11100000 10110010 10100000
```

UTF-8

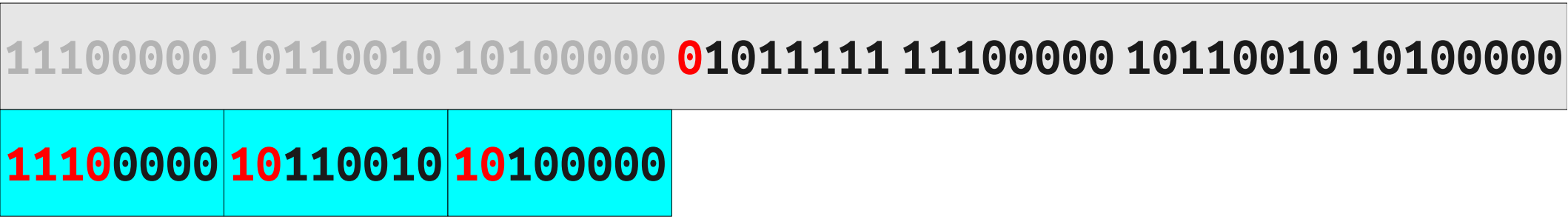
11100000 10110010 10100000 01011111 11100000 10110010 10100000

11100000	10110010	10100000

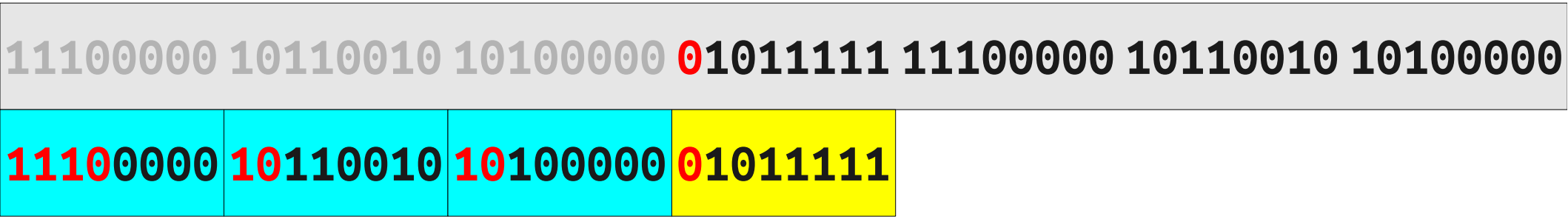
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000

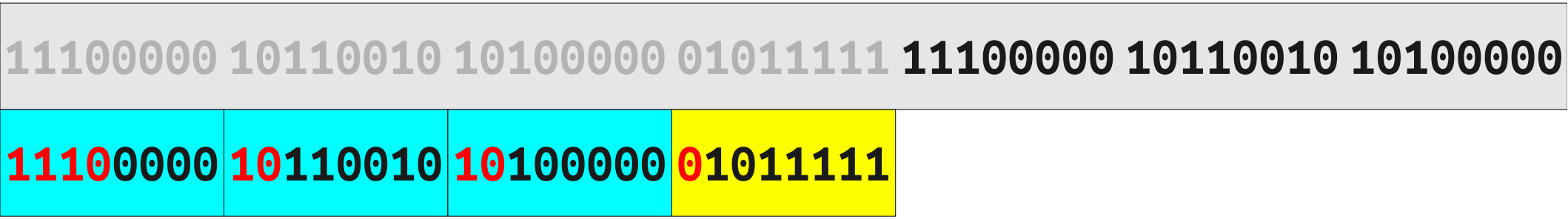
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000

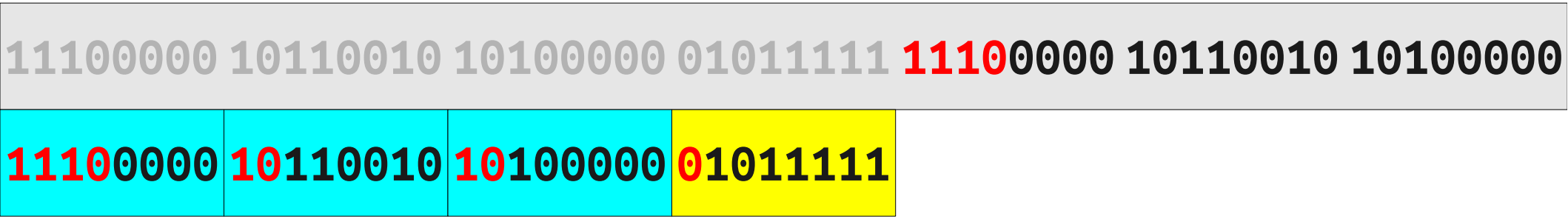
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111

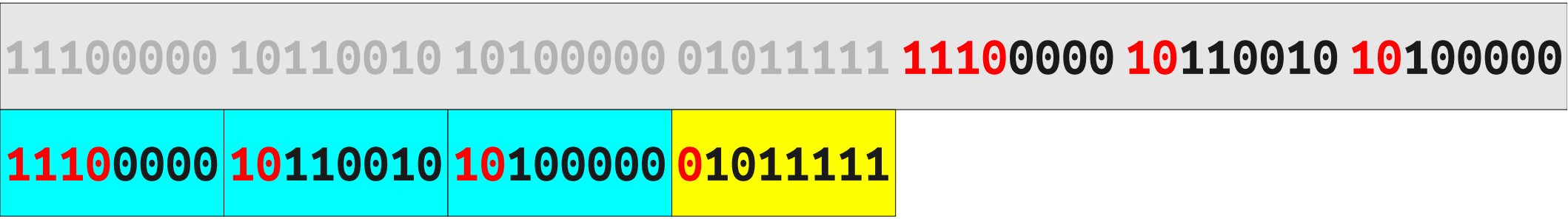
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111

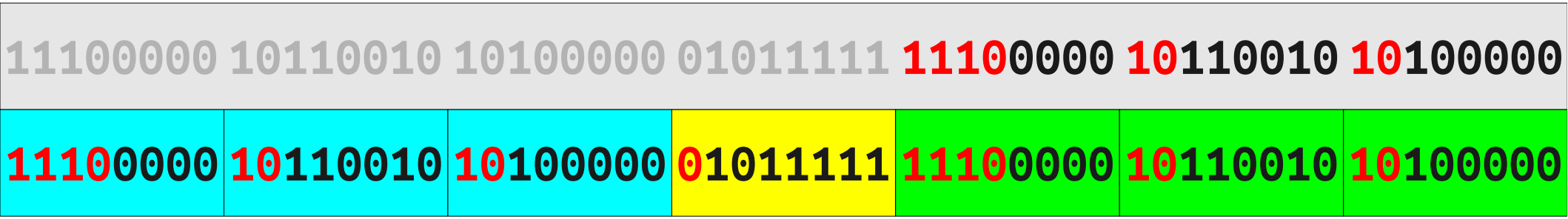
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111

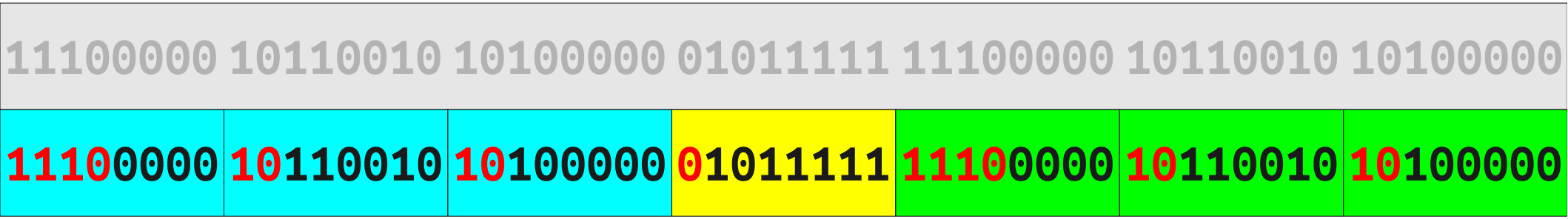
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111

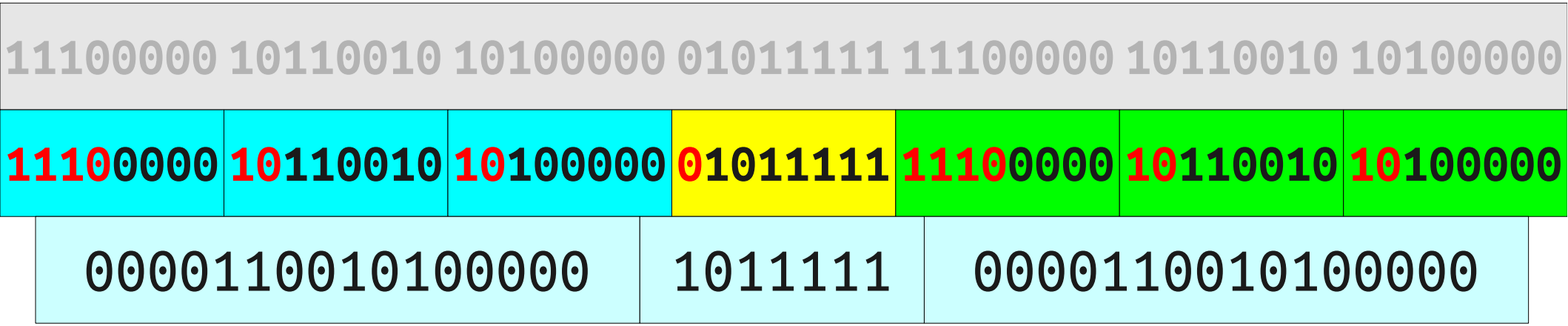
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111 11100000 10110010 10100000

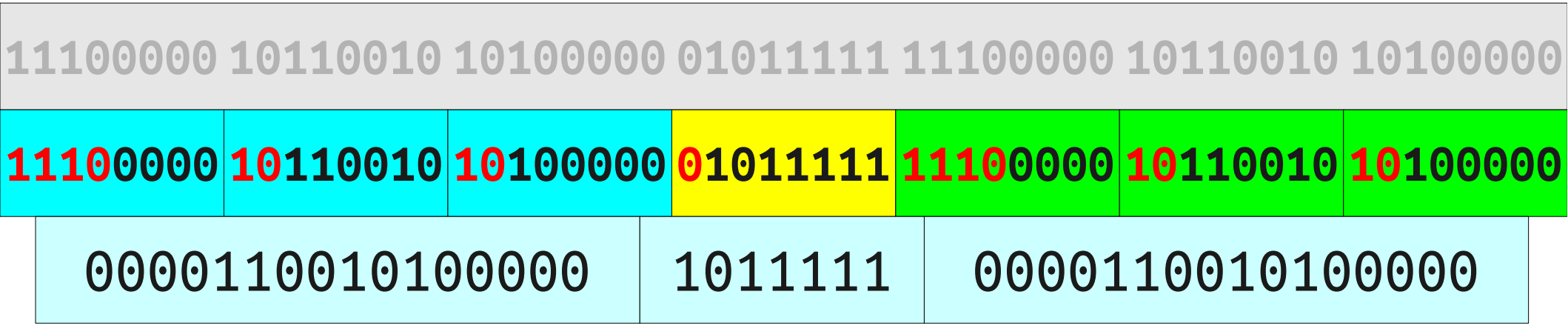
UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111 11100000 10110010 10100000

UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111 11100000 10110010 10100000
0000110010100000 1011111 0000110010100000

UTF-8


11100000 10110010 10100000 01011111 11100000 10110010 10100000
11100000 10110010 10100000 01011111 11100000 10110010 10100000
0000110010100000 1011111 0000110010100000

ಠ_ಠ

Next Time

- **Graphs and Graph Algorithms**
  - Representing relations and connections.
  - Representing graphs.
  - Graph search algorithms.

Good luck on the exam!