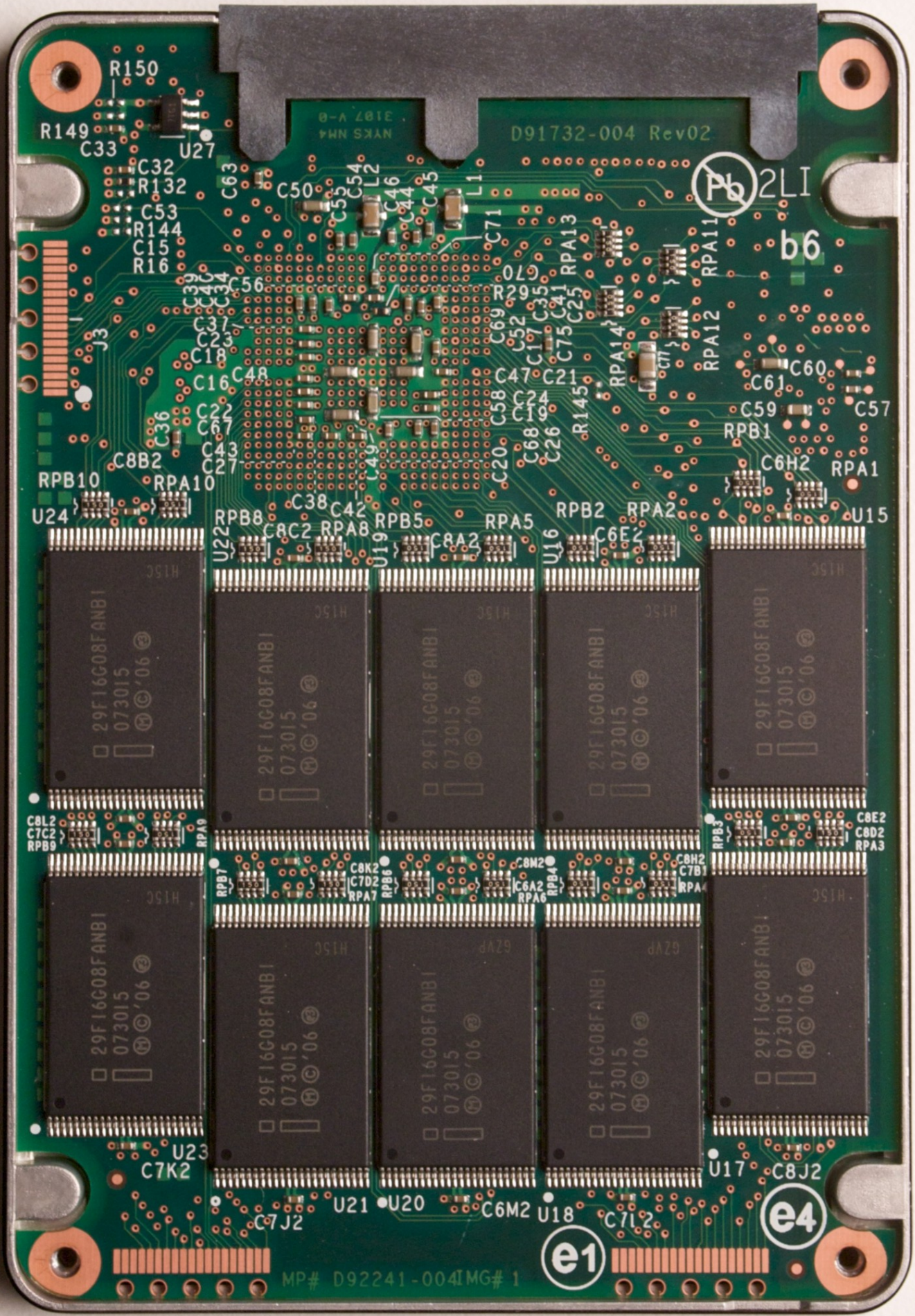
Beyond Data Structures

Huffman Encoding


D91732-004 Rev02
MP# D92241-004 IMG# 1
R150
R149
C33
U27
C32
R132
C53
R144
C15
R16
C63
C50
C54
L2
C46
C44
C45
L1
C71
RPA13
RPA11
C70
R29
C56
C39
C34
C37
C23
C18
C16
C48
J3
C69
C52
C17
C35
C41
C25
C75
RPA14
RPA12
C47
C21
C58
C24
C19
R145
C68
C26
C20
C49
C22
C67
C36
C43
C27
C8B2
RPB10
RPA10
U24
C38
C42
RPB8
C8C2
RPA8
U22
RPB5
U19
C8A2
RPA5
U16
RPB2
RPA2
C6E2
C60
C61
C59
C57
RPB1
C6H2
RPA1
U15
29F16G08FANBI
073015
U23
C7K2
U21
U20
C6M2
U18
C7L2
U17
C8J2
e1
e4
C8L2
C7C2
RPB9
RPA9
RPB7
C8K2
C7D2
RPA7
RPB6
C8M2
C6A2
RPA6
RPB4
C8H2
C7B1
RPA4
RPB3
C8E2
C8D2
RPA3

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 or by trapping electrons in small gates.
- A single 0 or 1 is called a ***bit***.
- A group of eight bits is called a ***byte***.

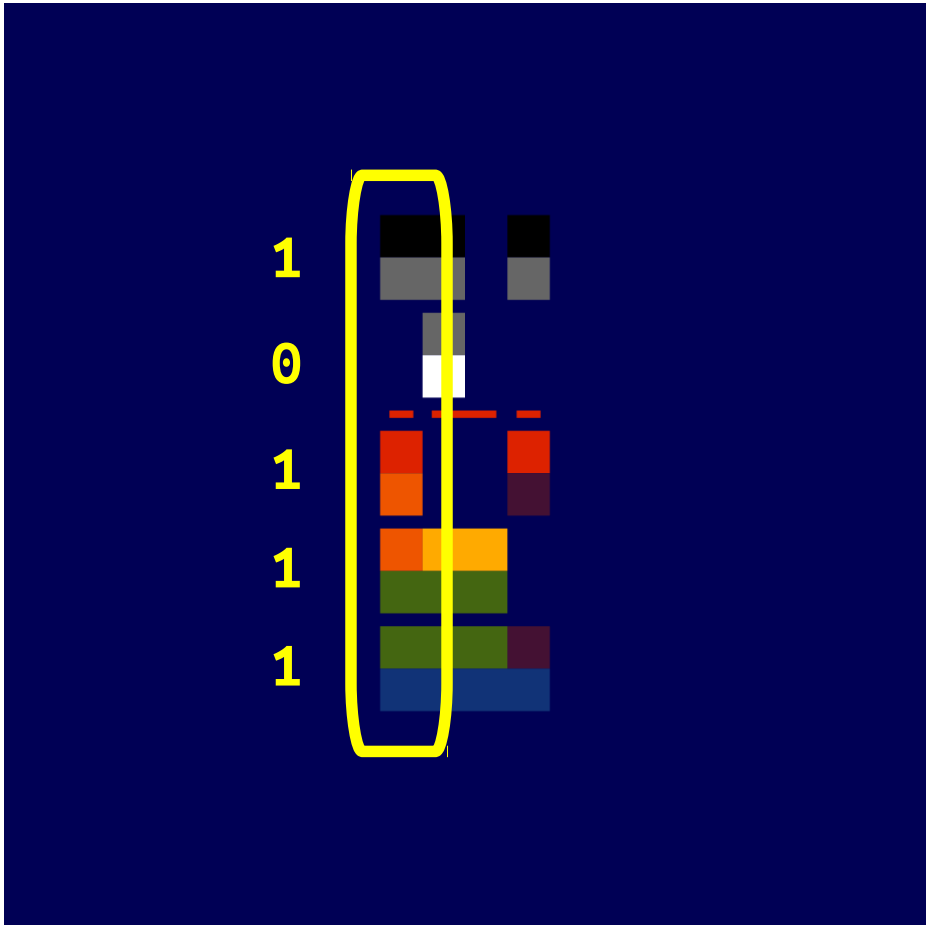
  00000000, 00000001, 00000010, 00000011,
  00000100, 00000101, 00000110, …
- There are $2^8 = 256$ different bytes.
  - ***Great recursion practice:*** Write a function to list all of them!

Representing Text

- Text uses all sorts of different characters.
- Computers require everything to be written as zeros and ones.
- To represent text inside the computer, we can choose some way of mapping characters to series of zeros and ones and vice-versa.
- There are many ways to do this.

Baudot Codes

- The ***Baudot code*** was one of the first codes for representing text as 0s and 1s.
- It was used by early teleprinters and, while mostly obsoleted, is still around.
- Coldplay's album ***X&Y*** used it for their Hip and Stylish album cover.



ASCII

- Early (American) computers needed some standard way to send output to their (physical!) printers.
- Since there were fewer than 256 different characters to print (1960's America!), each character was assigned a one-byte value.
- This initial code was called ***ASCII***. Surprisingly, it's still around, though in a modified form (more on that later).
- For example, the letter `A` is represented by the byte `01000001` (65). You can still see this in C++:

```
cout << int('A') << endl; // Prints 65
```

ASCII

- In ASCII:
  - Each character is ***exactly one byte*** (8 bits).
  - All computers agree ***in advance*** which characters have which values.
- This makes it possible to transmit text data from one computer to another by just writing out a series of bits.
- Here's what this might look like:

Transmitting the Dikdik

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

Transmitting the Dikdik

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
```

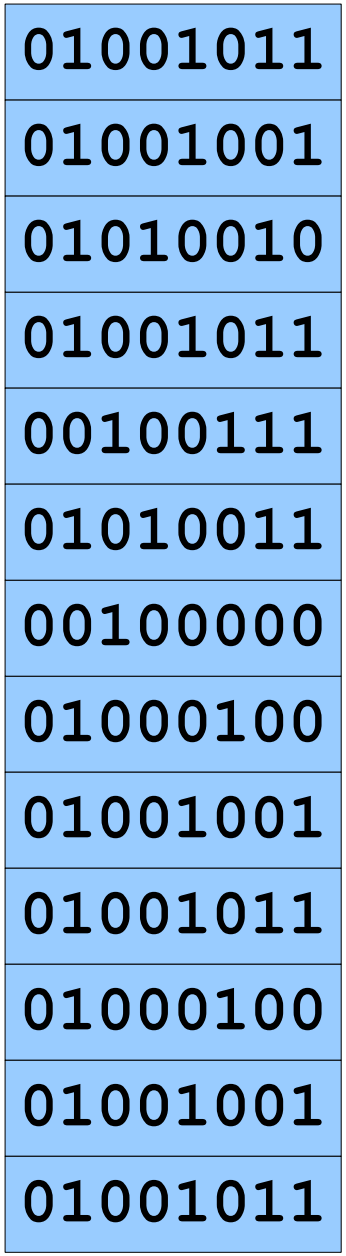
Transmitting the Dikdik

```
01001011010010010101001001001011
00100111010100110010000001000100
01001001010010110100010001001001
01001011
```

Transmitting the Dikdik

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

Transmitting the Dikdik



Transmitting the Dikdik

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

An Observation

- In ASCII, every character has exactly the same number of bits in it.
- Any message with n characters will use up exactly $8n$ bits.
  - Space for `KIRK'S DIKDIK`: 104 bits.
  - Space for `COPYRIGHTABLE`: 104 bits.
- We say that ASCII is a ***fixed-length encoding***.

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		

The Key Idea

- If we can find a way to

  give all characters a bit pattern,

  that both the sender and receiver know about, and

  that can be decoded uniquely,

  then we can represent the same piece of text in multiple different ways.

- ***Goal:*** Find a way to do this that uses *less space* than the standard ASCII representation.

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

0101010111000000100010

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

01010101110000010010

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 code is a prefix of another code.
- 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

10011111110001000111011001101100110

10
K

Prefix Codes

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

1001111111000100011101100110011011001100110

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

10011111110001000111011001101100110

10	01
K	I

Prefix Codes

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

```
1001111111000100011101100110110011O
```

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

10011111100010001110110011011001110

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

  `1001111110001000111011001101100110`

- This uses just 34 bits, compared to our initial 39.
- Remaining questions:
  - How do you generate a prefix code?
  - And what does any of this have to do with binary trees?

Time-Out for Announcements!

Assignment 5

- Assignment 5 is due on Friday.
  - Want to use a late day? Turn it in on Monday of next week.
- Have questions?
  - Stop by the LaIR!
  - Stop by our office hours!
  - Ask your section leader!
  - Ask a partner!
  - Ask on Piazza!

Ceçi n'est pas une annonce.

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

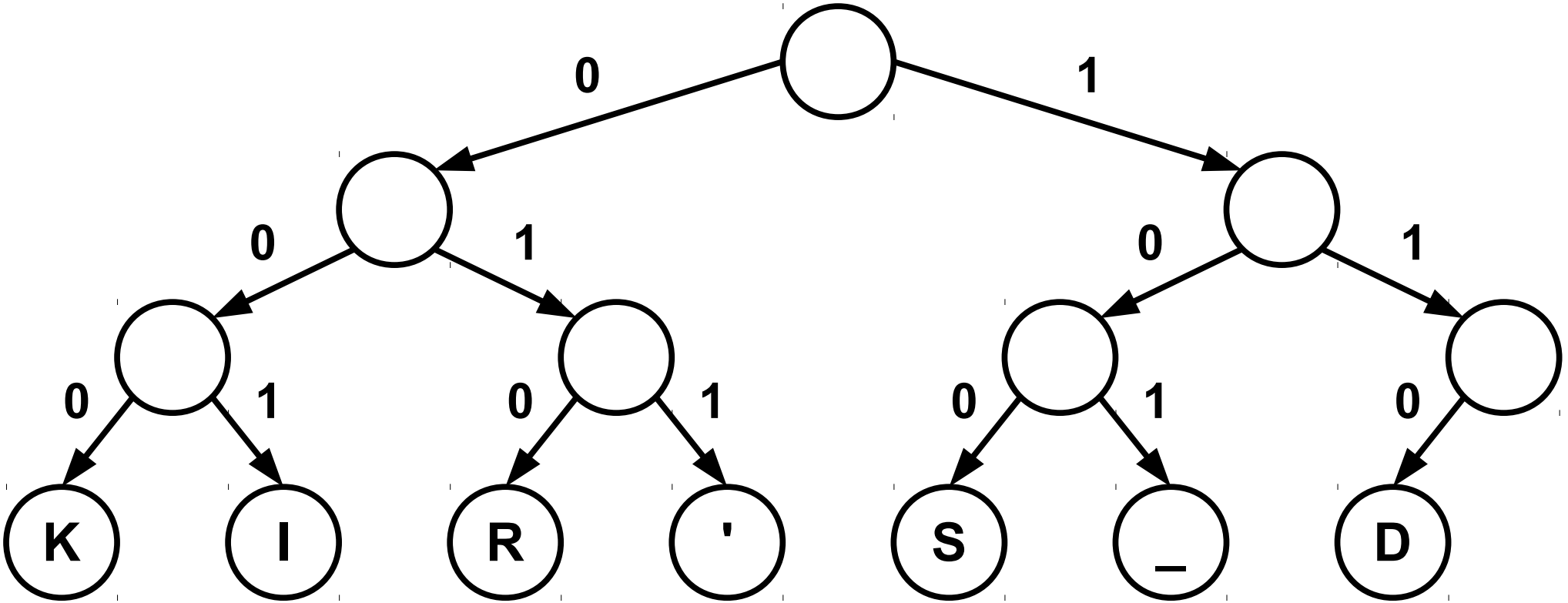
```
10011111100010001110110011011001l0
```

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

```
11111101111101111011111011101101001111011111100111110111111O
```

How do you find a “good” prefix code?

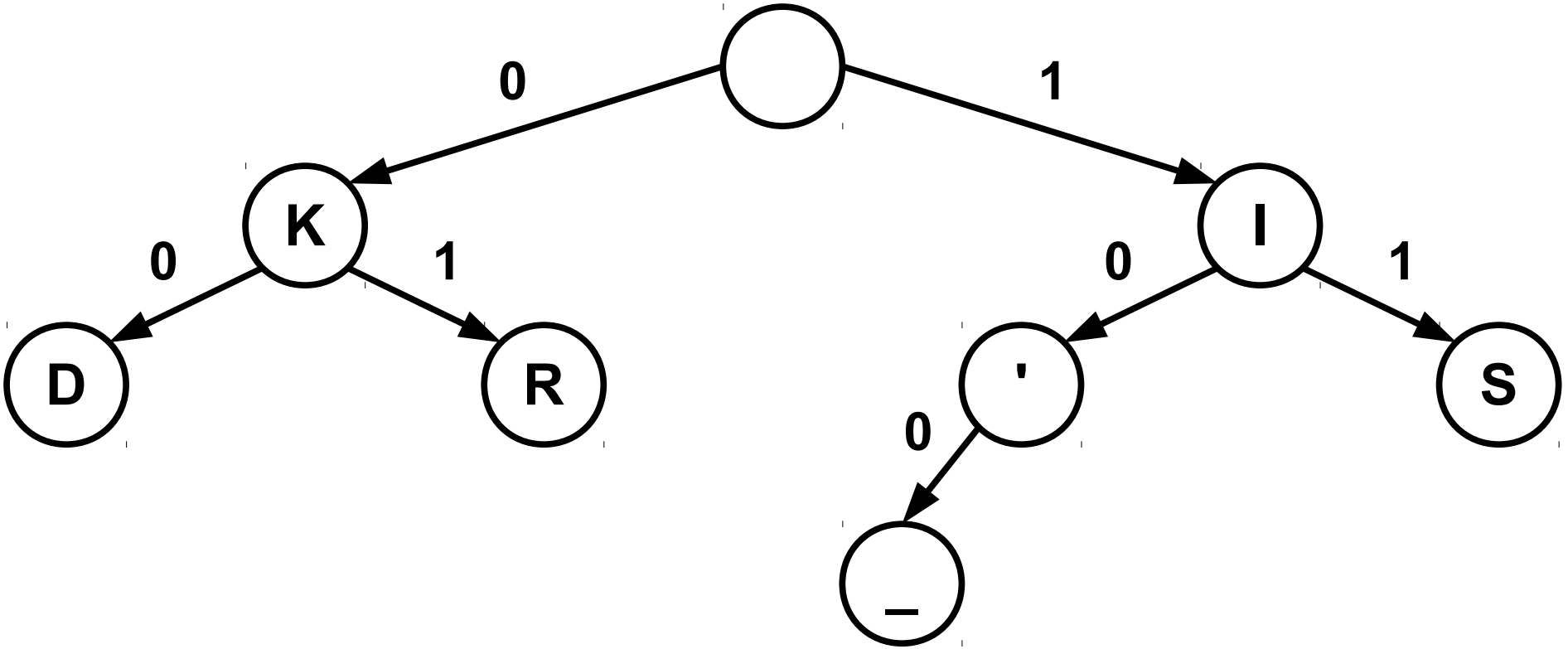
The Key Idea



K	000
I	001
R	010
'	011

S	100
	101
D	110

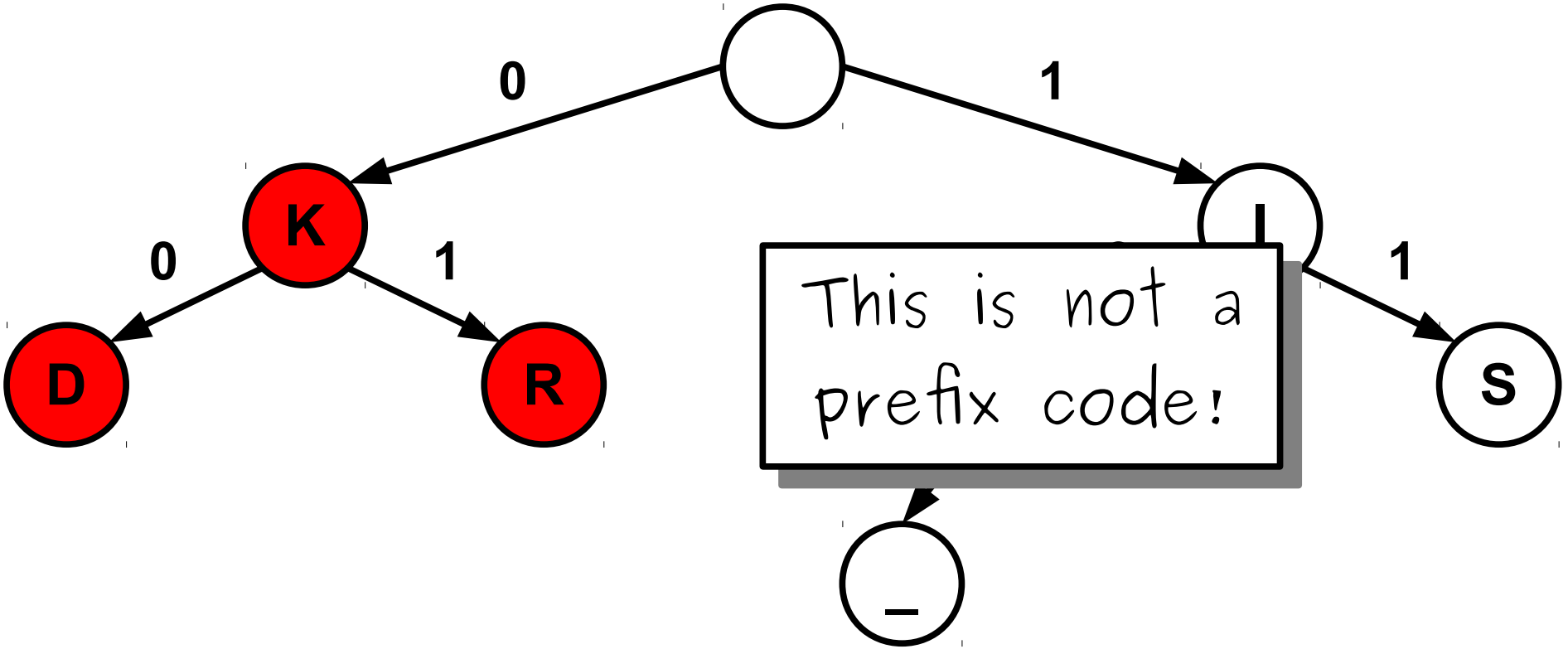
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

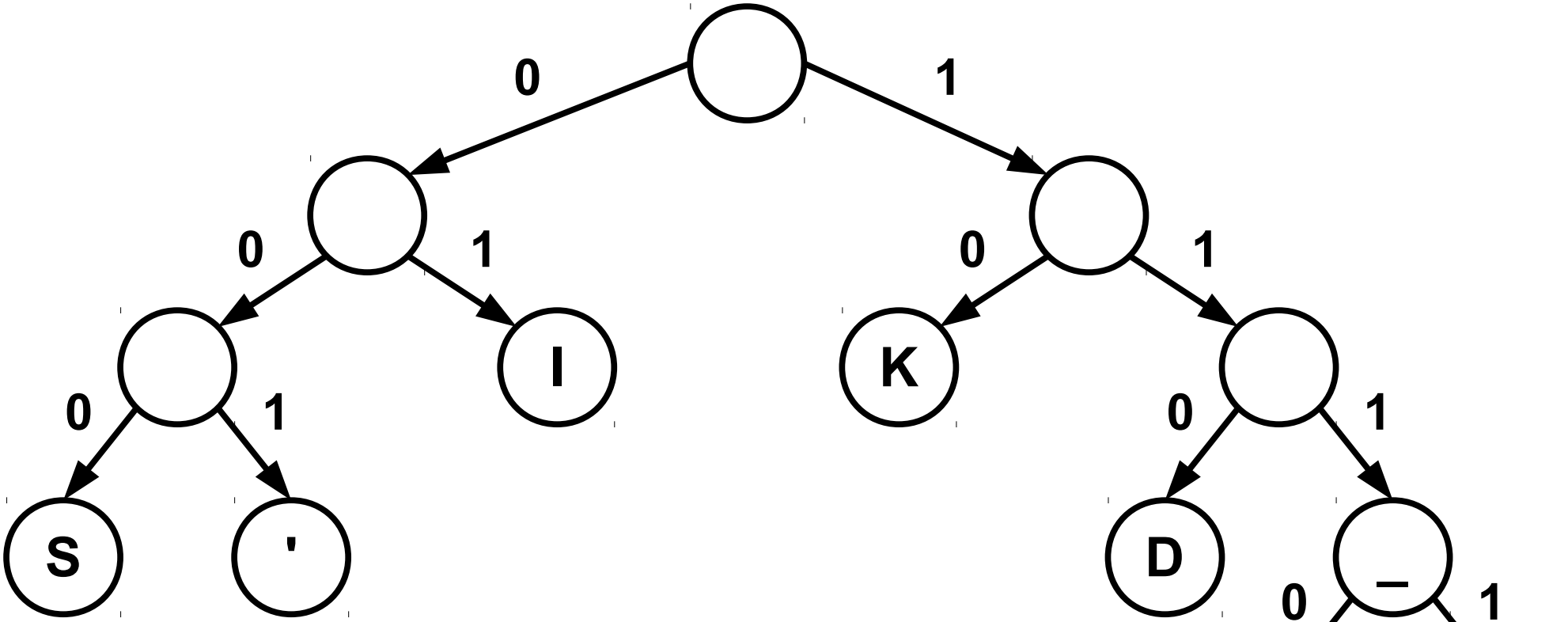
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

The Key Idea



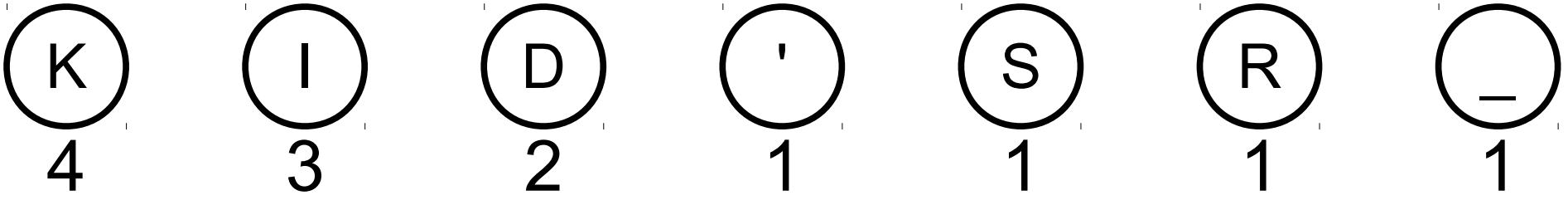
K	10
I	01
D	110
R	1111

'	001
S	000
	1110

Finding a good prefix code is equivalent to finding a good binary tree with all values stored at the leaves.

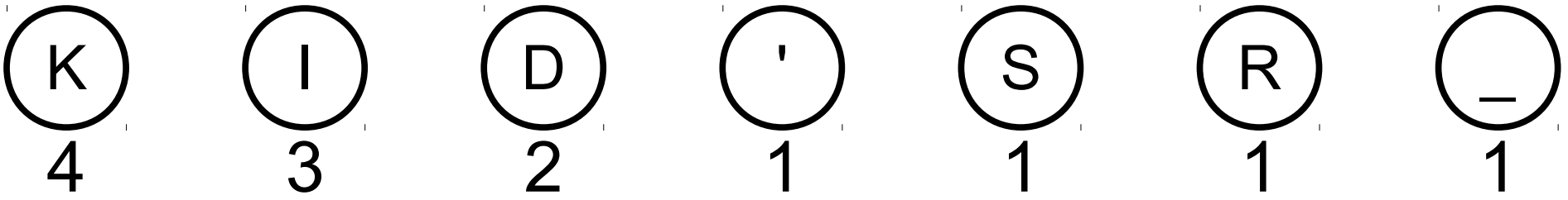
How do we find the best binary tree with this property?

Huffman Coding

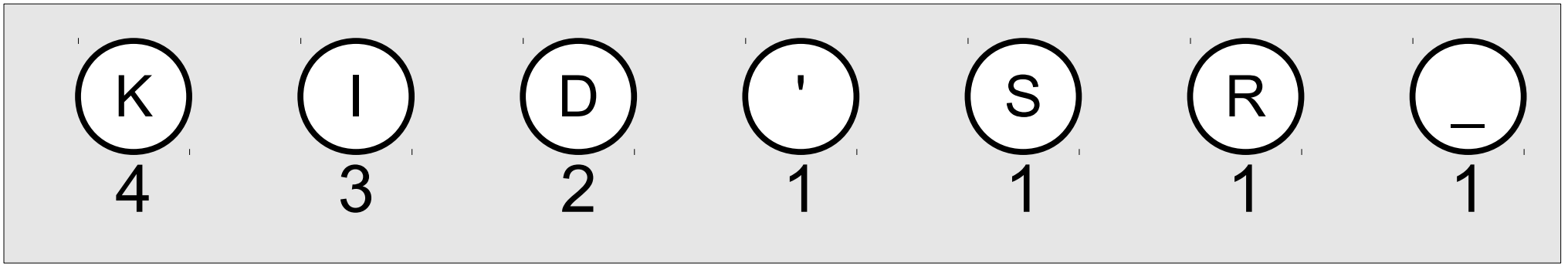


K	4
I	3
D	2
R	1
'	1
S	1
	1

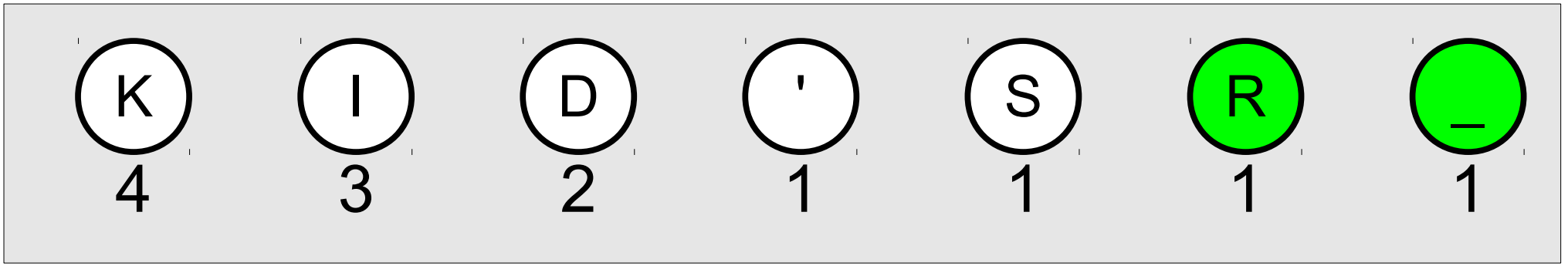
Huffman Coding



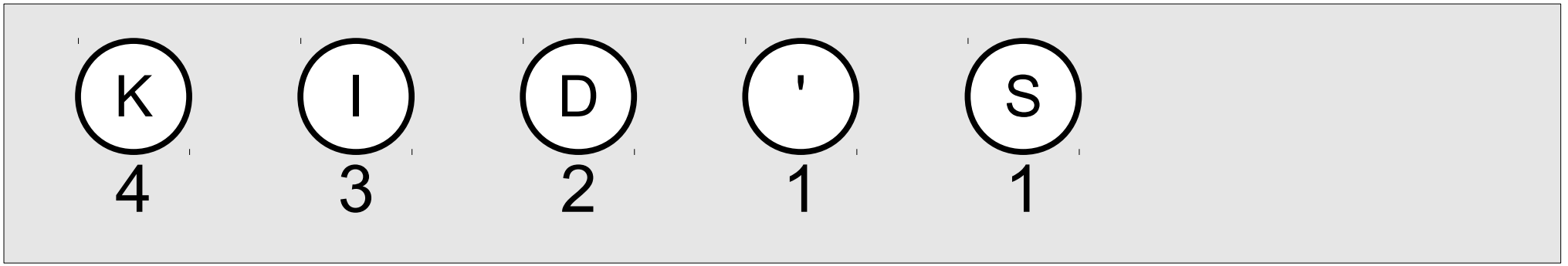
Huffman Coding

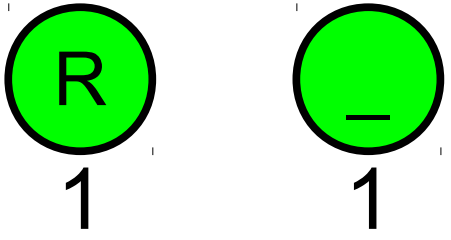


Huffman Coding

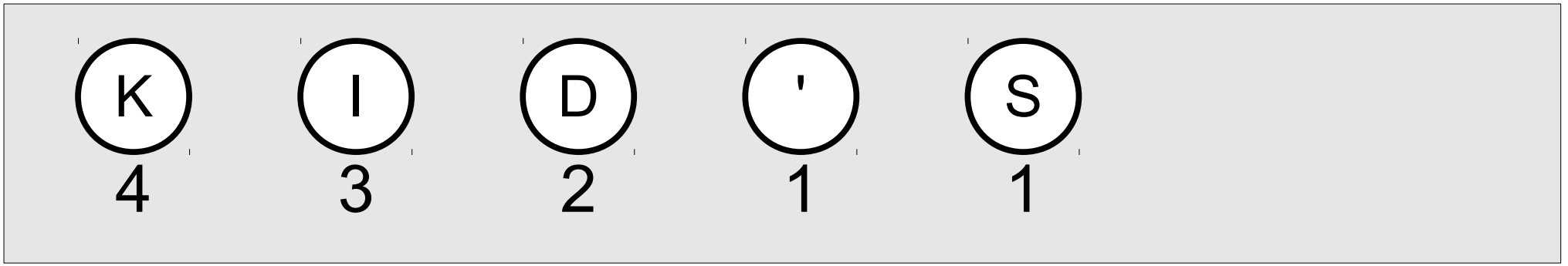


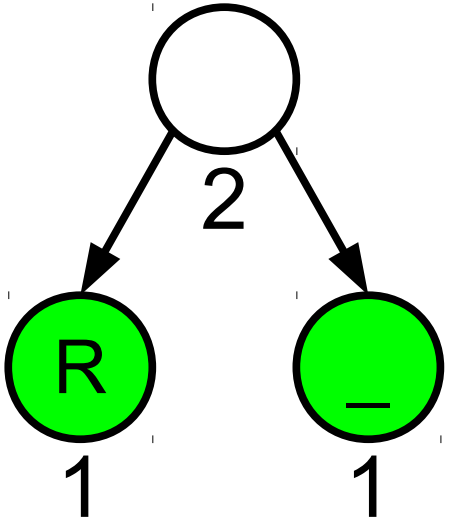
Huffman Coding



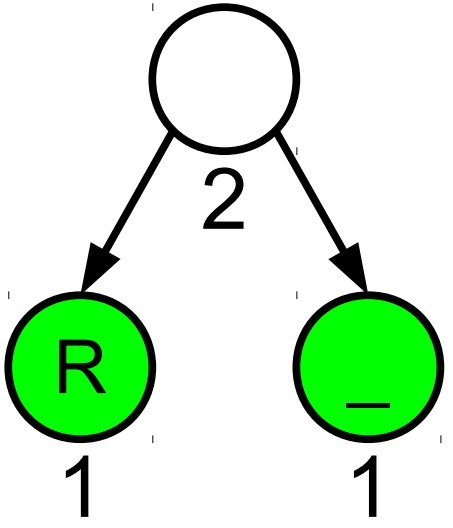


Huffman Coding

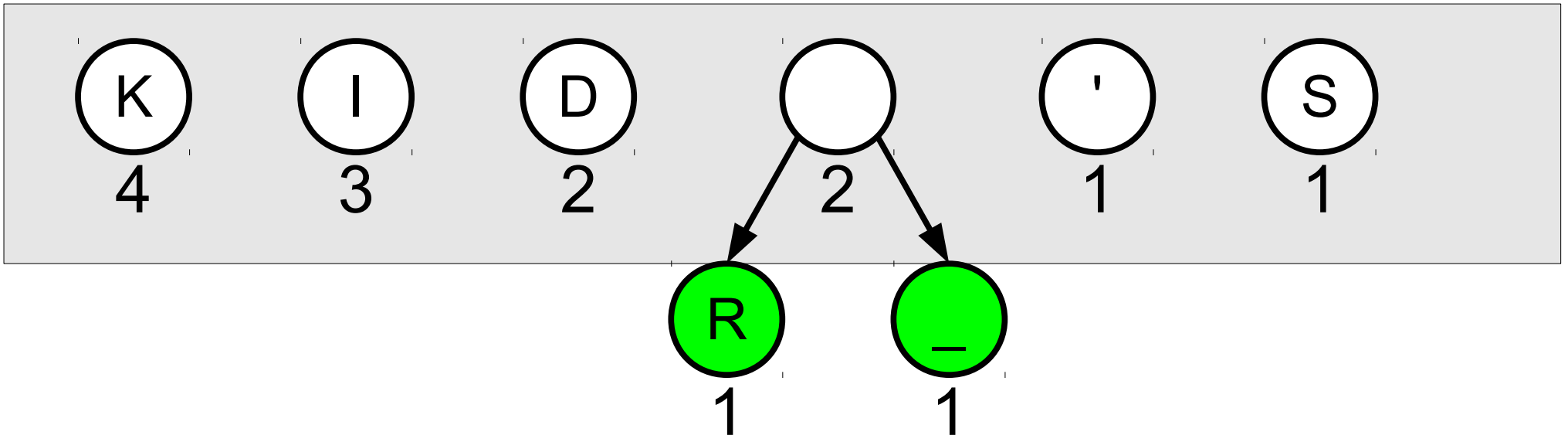




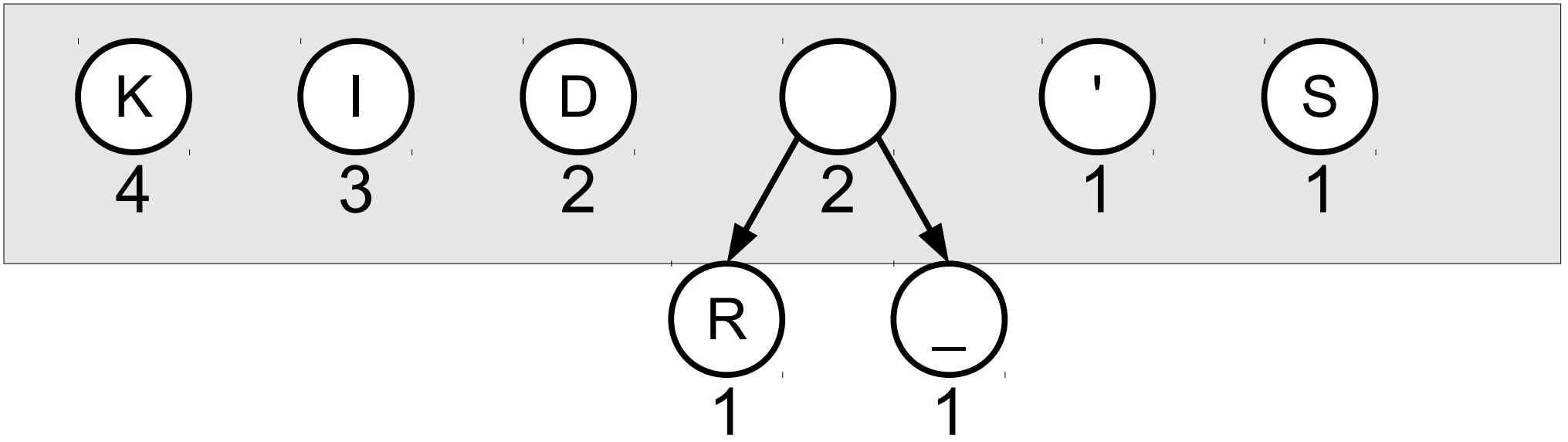
Huffman Coding



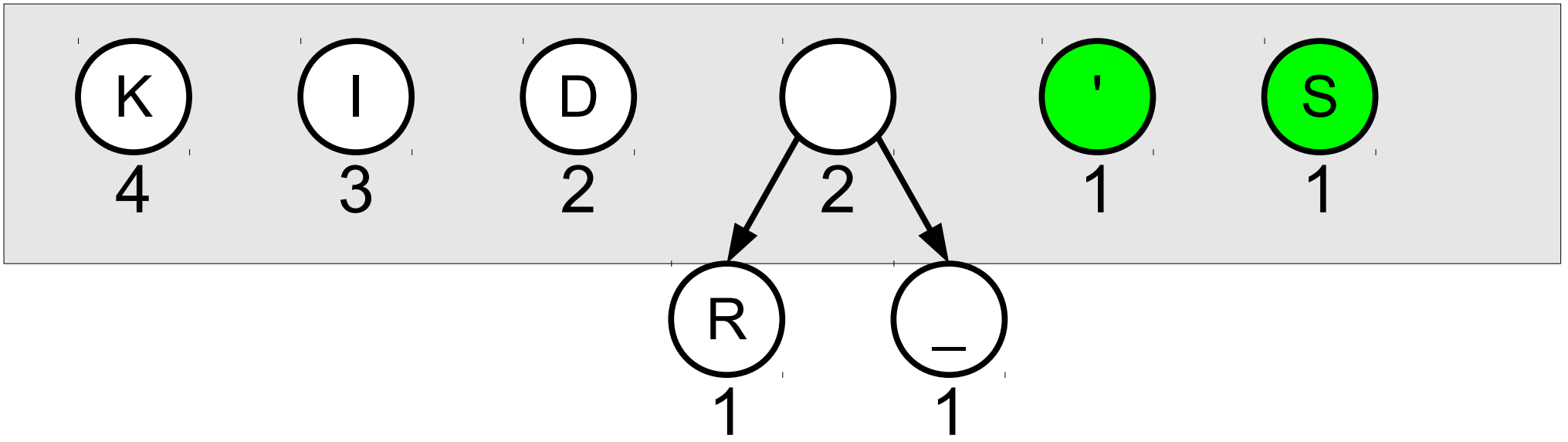
Huffman Coding



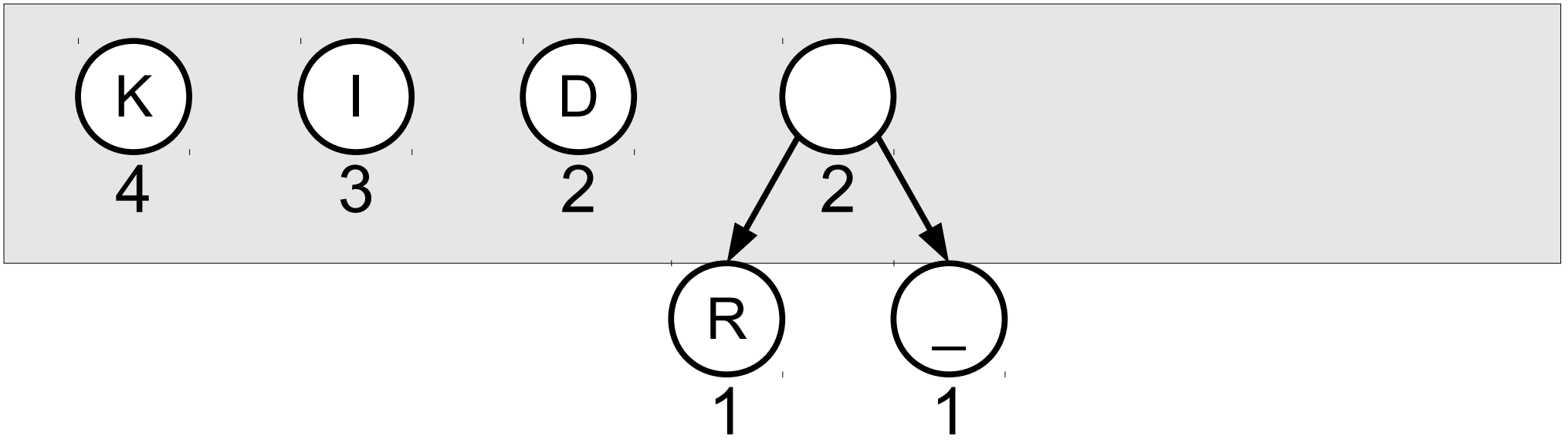
Huffman Coding

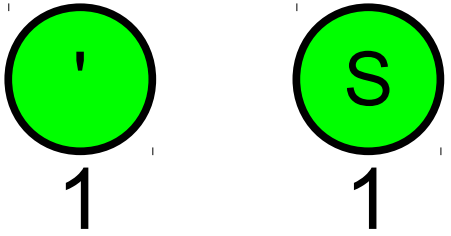


Huffman Coding

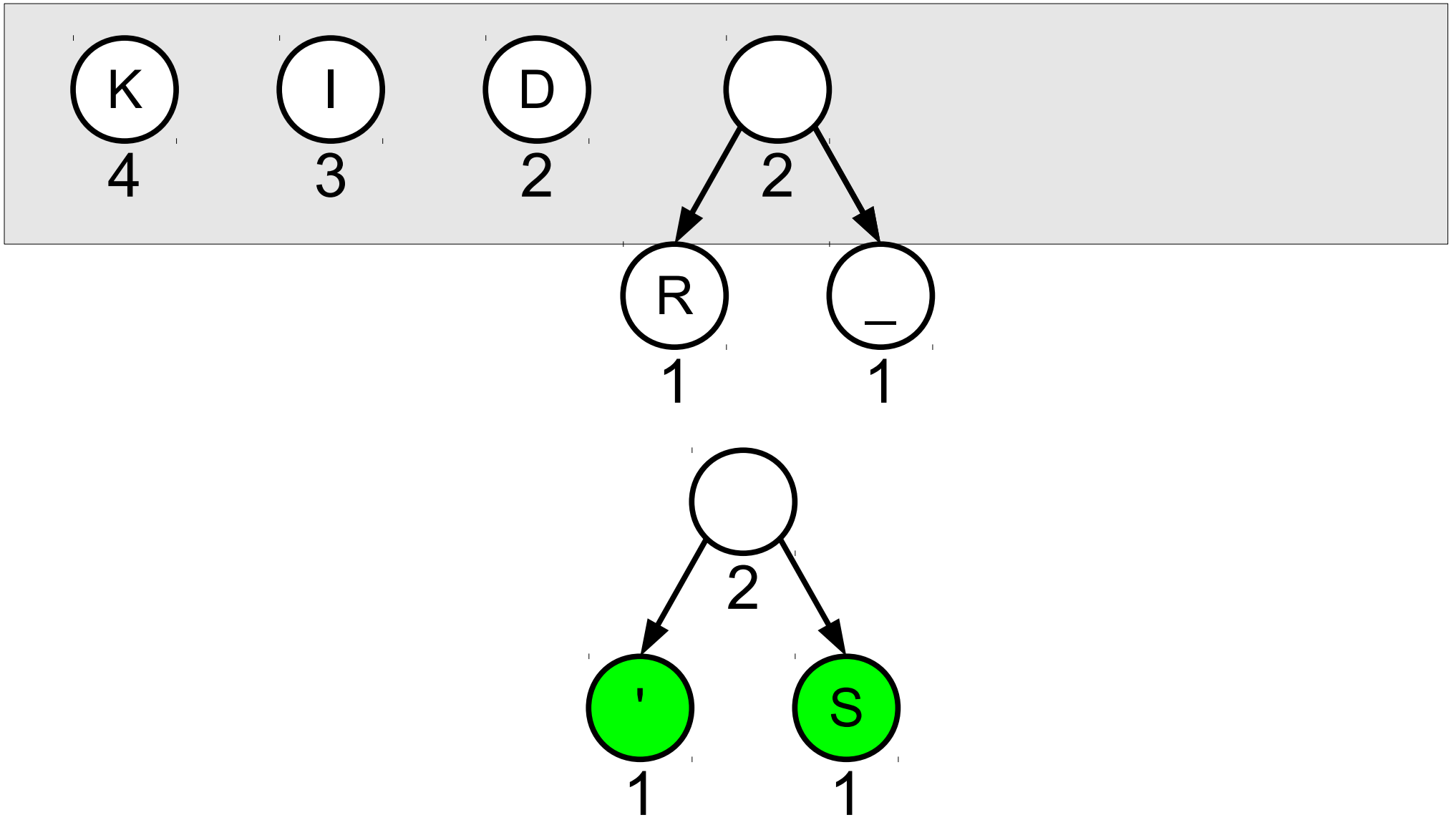


Huffman Coding

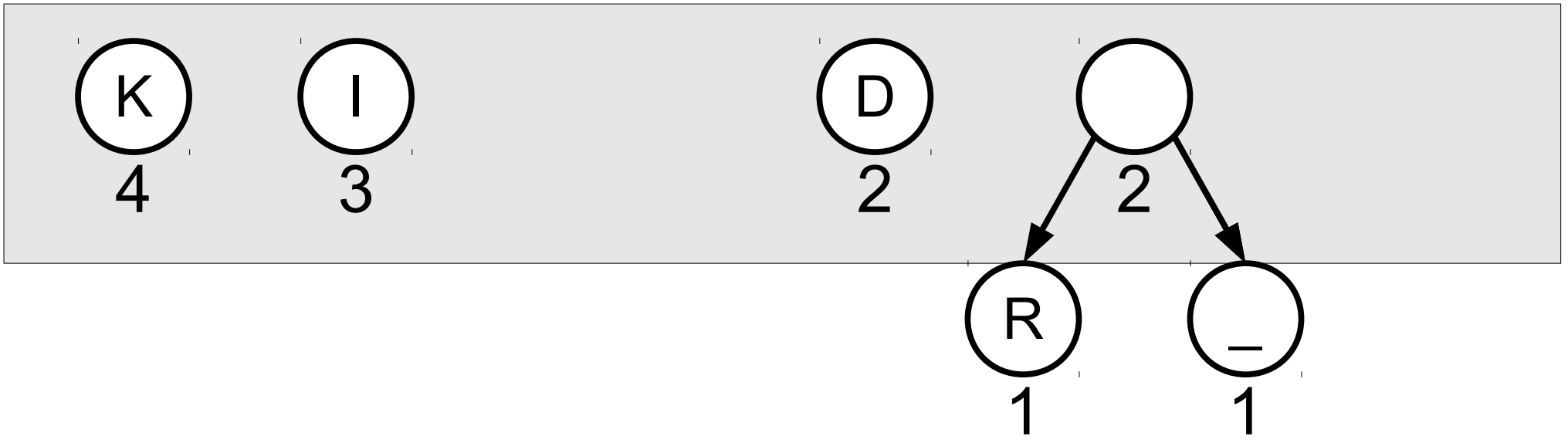


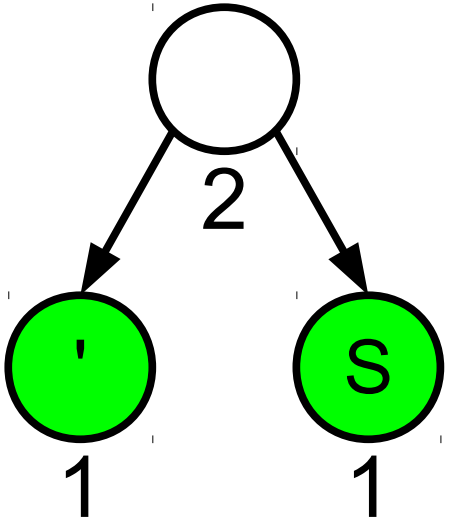


Huffman Coding

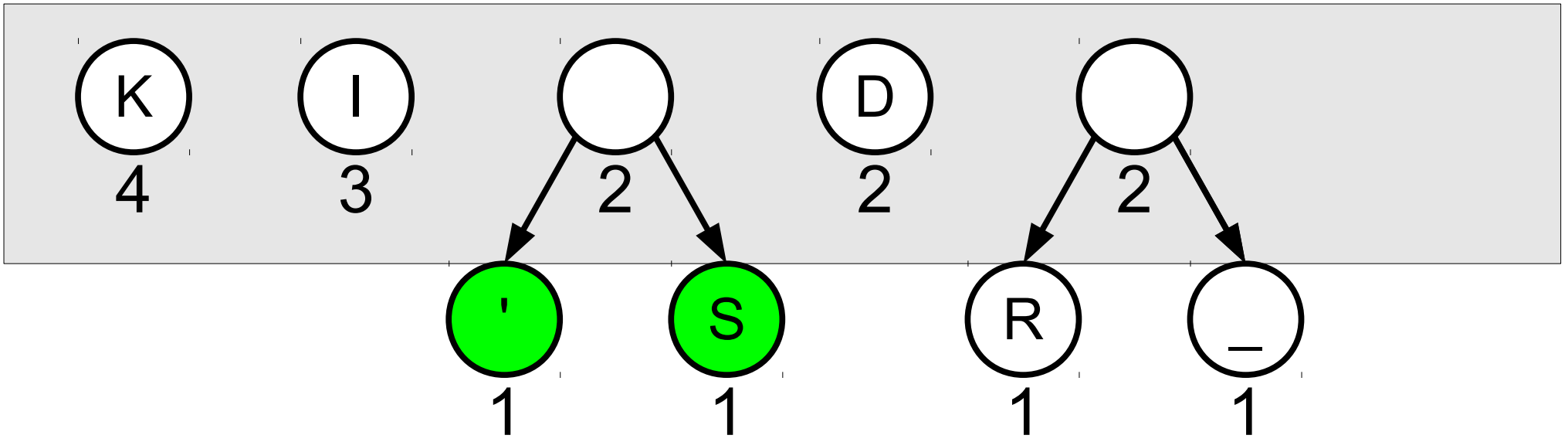


Huffman Coding

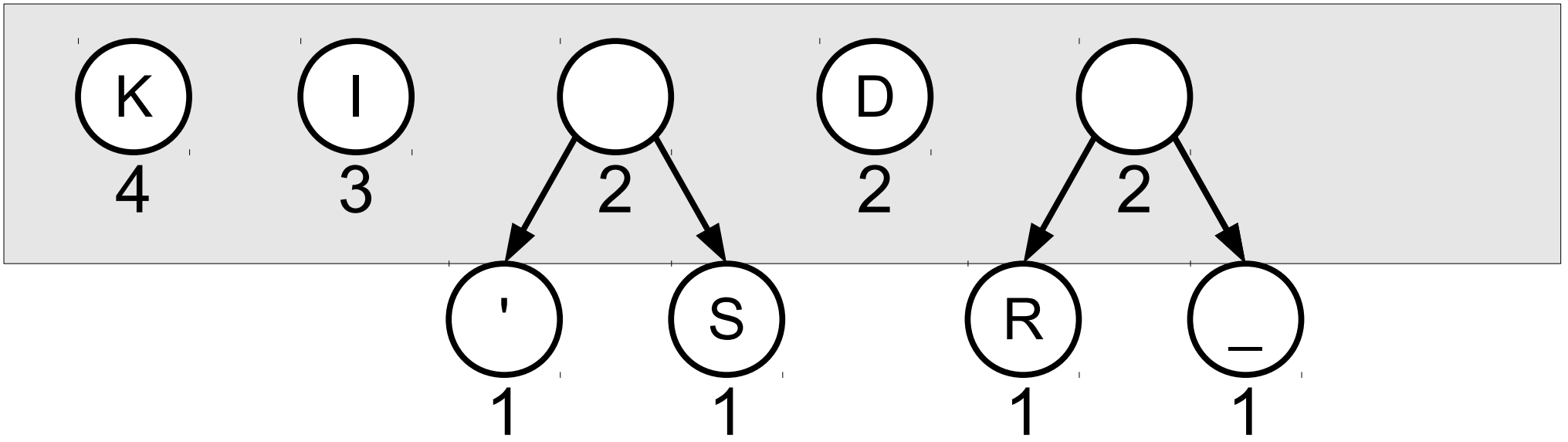




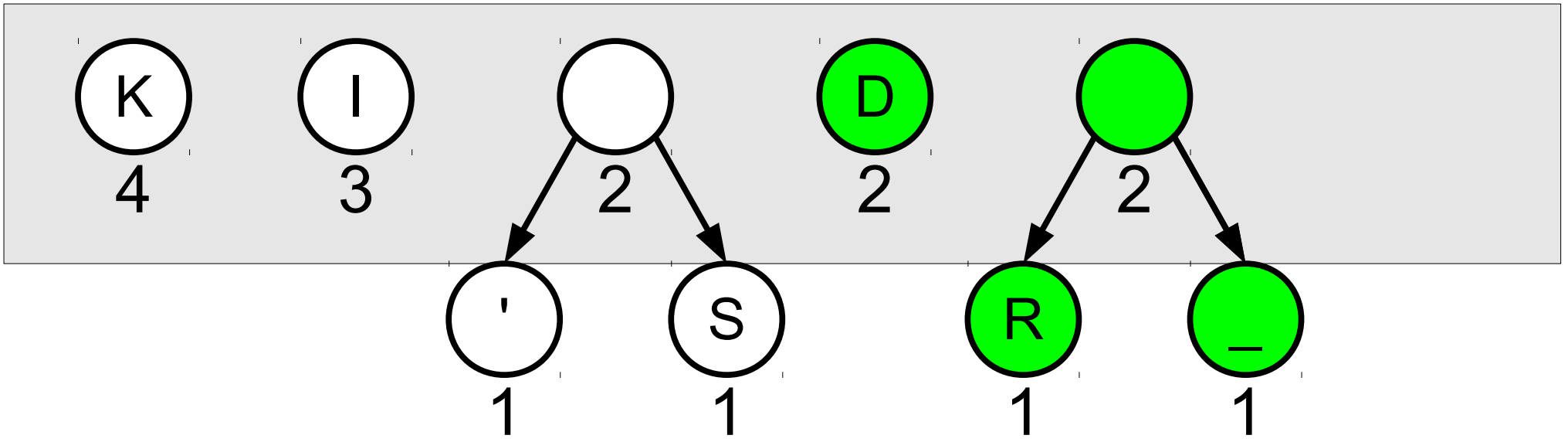
Huffman Coding



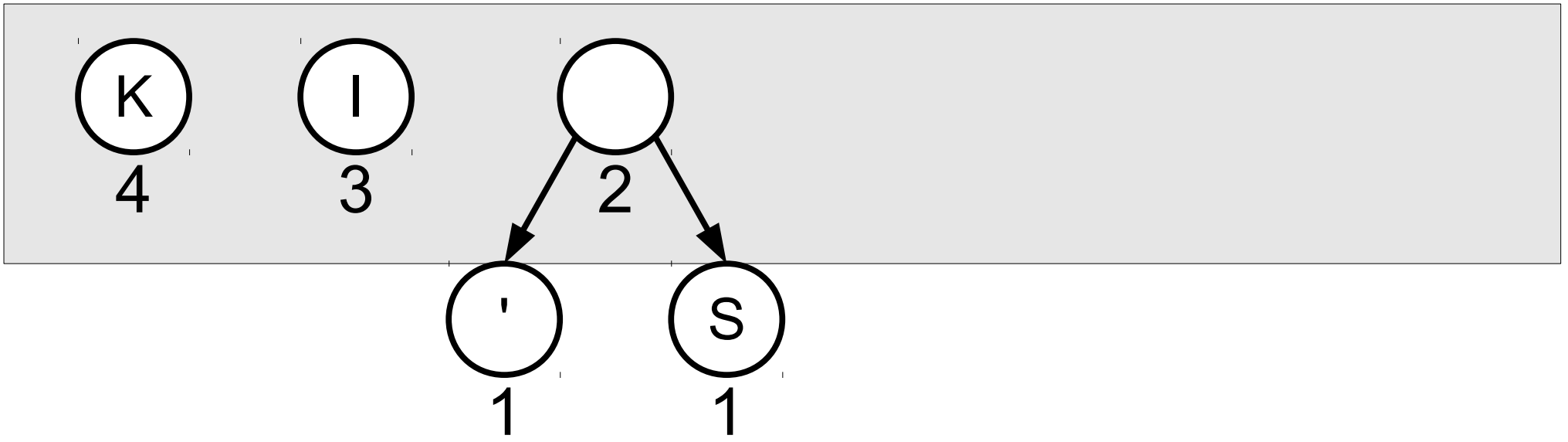
Huffman Coding

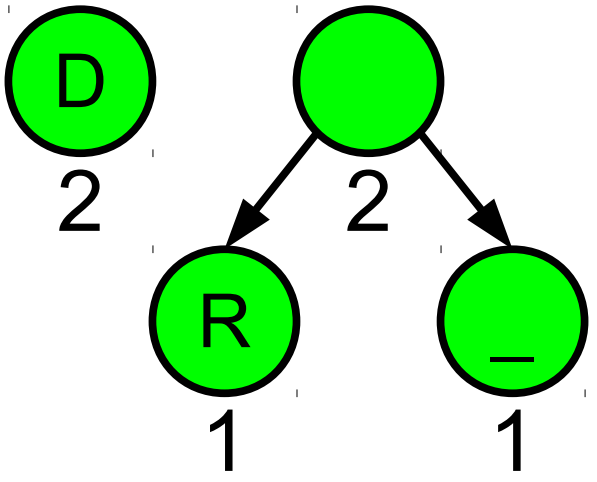


Huffman Coding

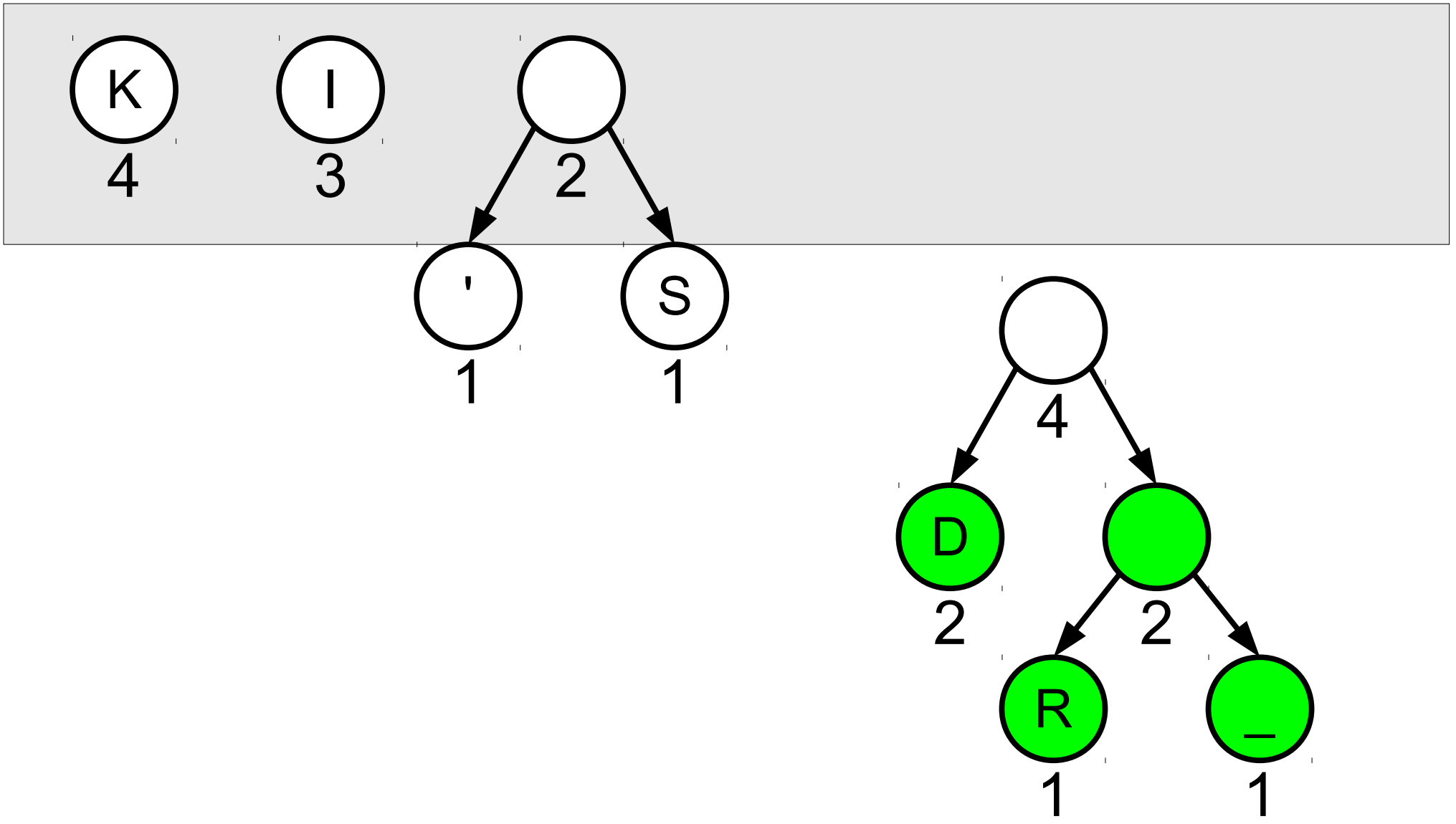


Huffman Coding

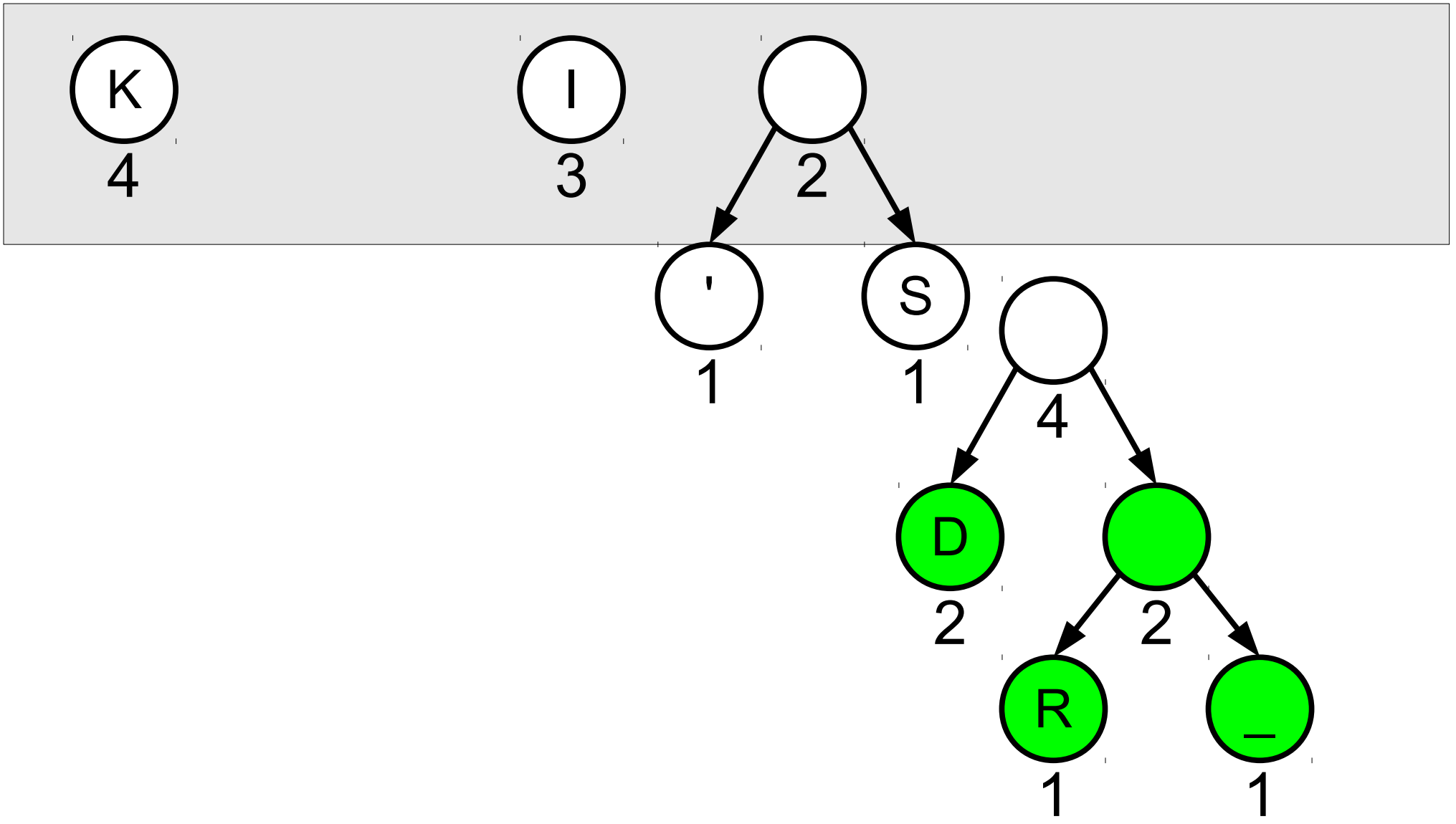




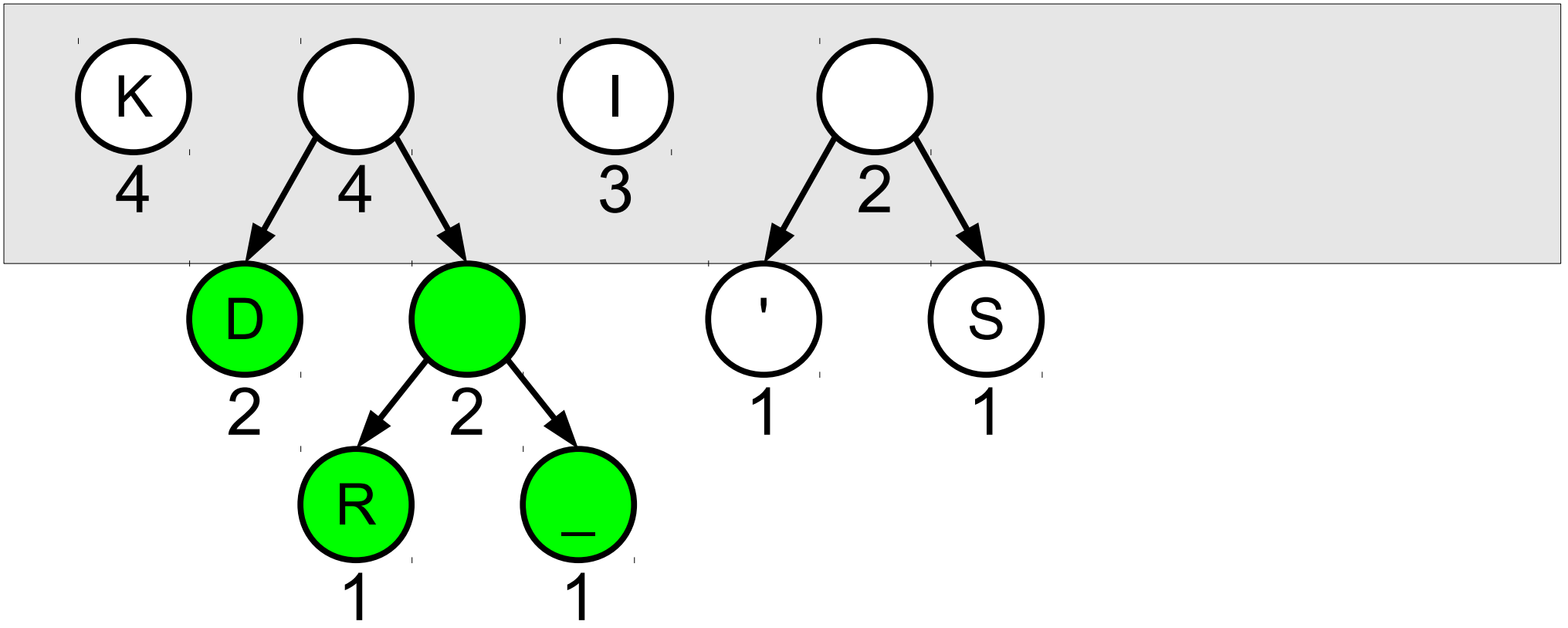
Huffman Coding



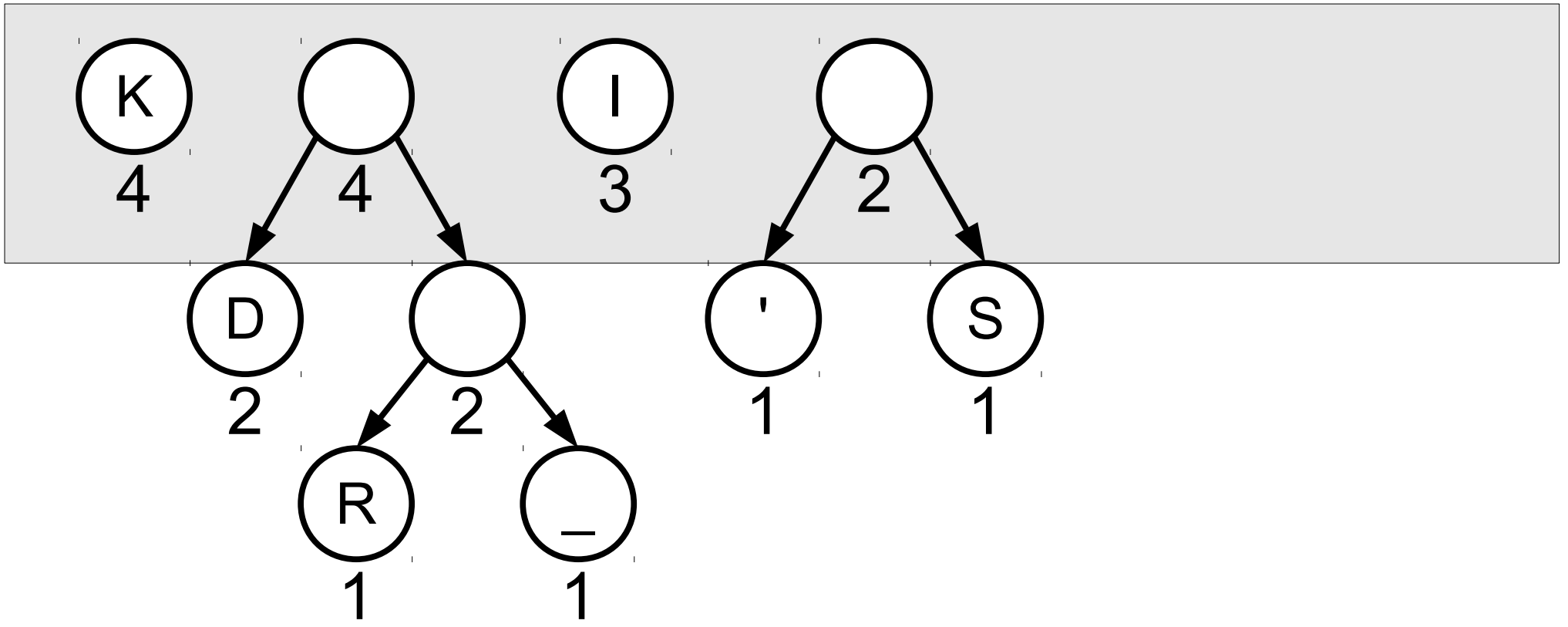
Huffman Coding



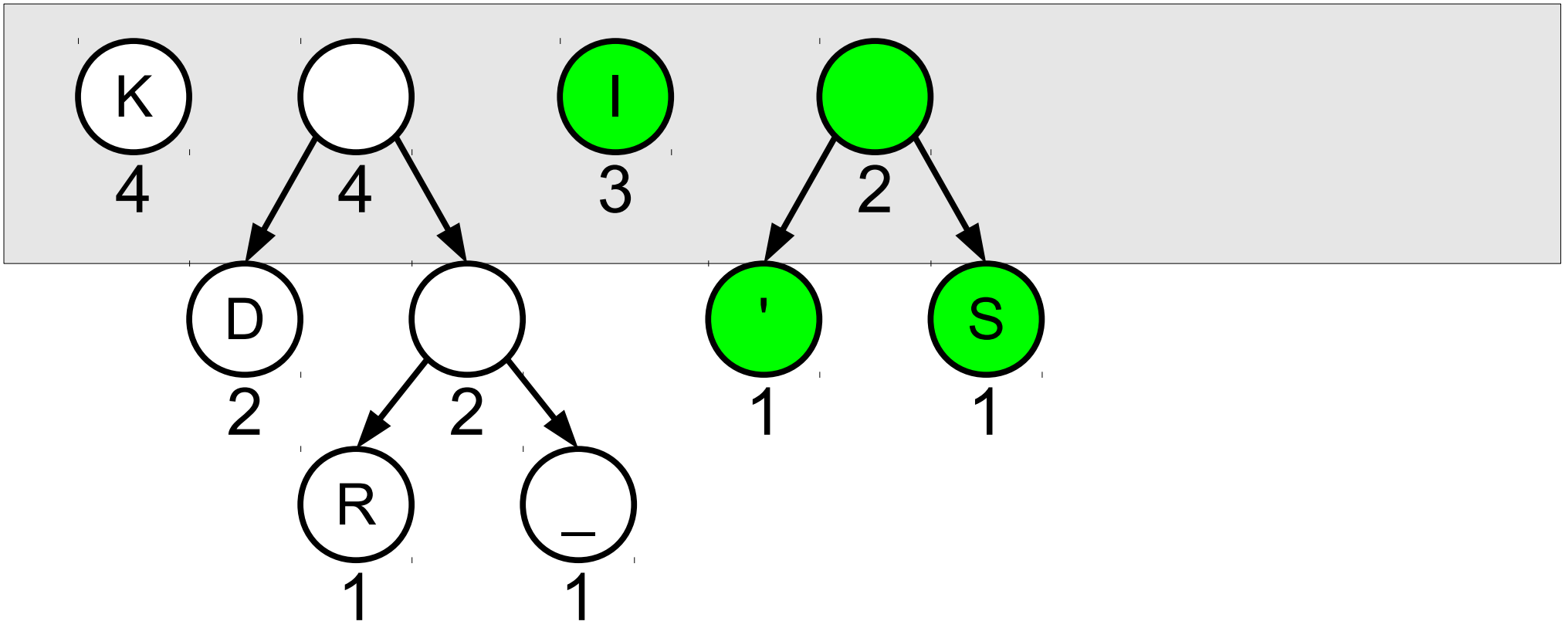
Huffman Coding



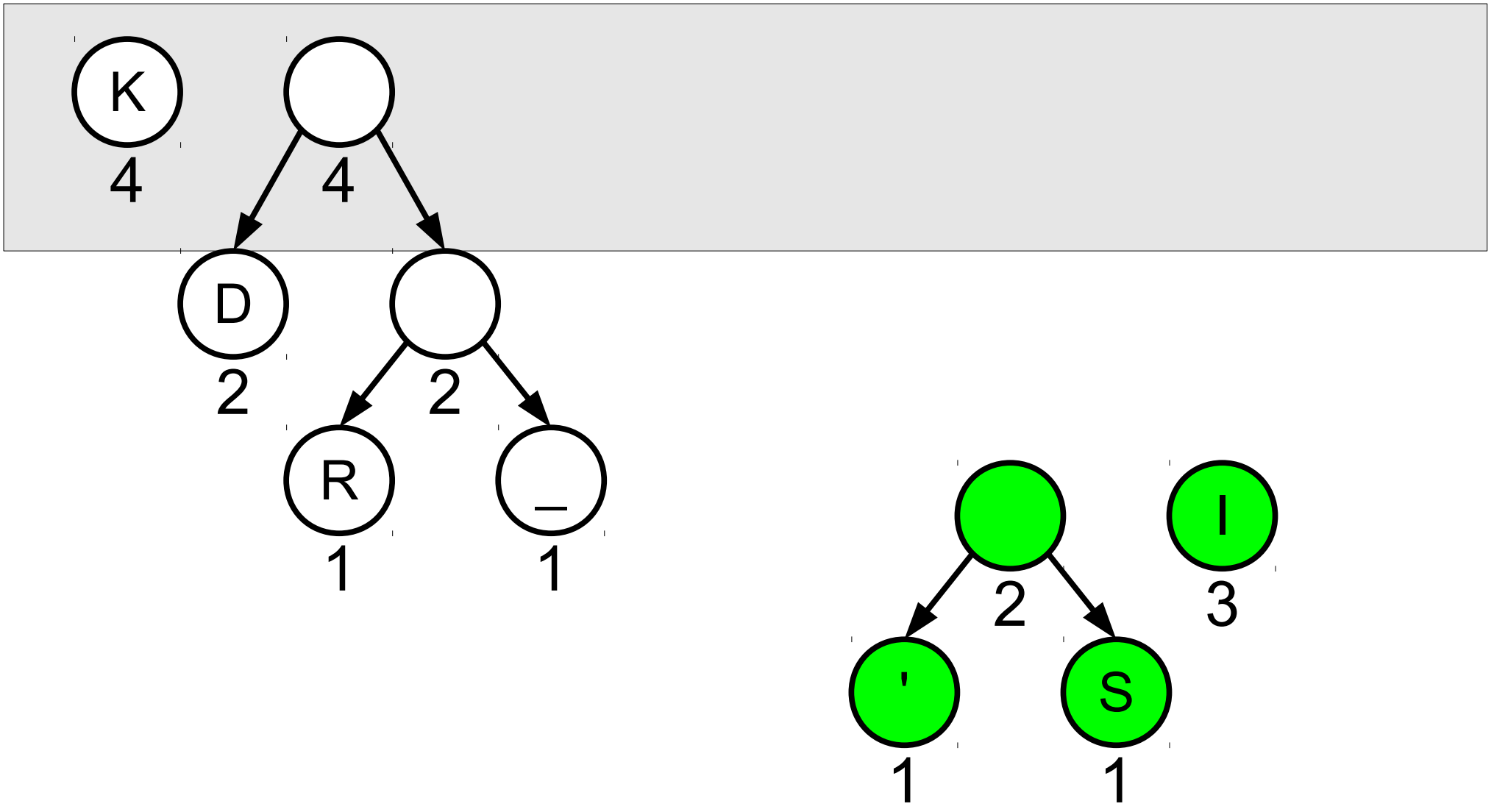
Huffman Coding



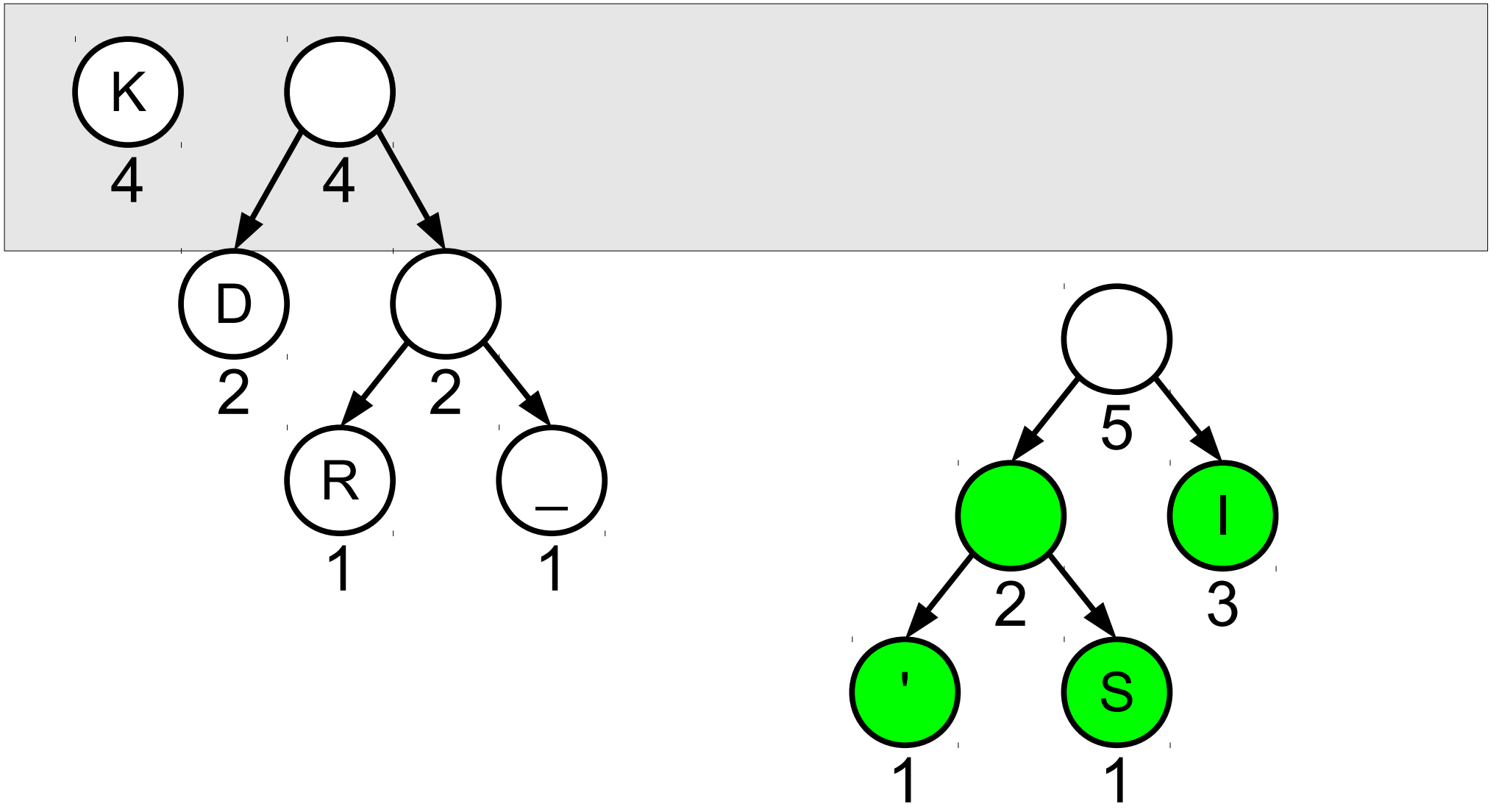
Huffman Coding



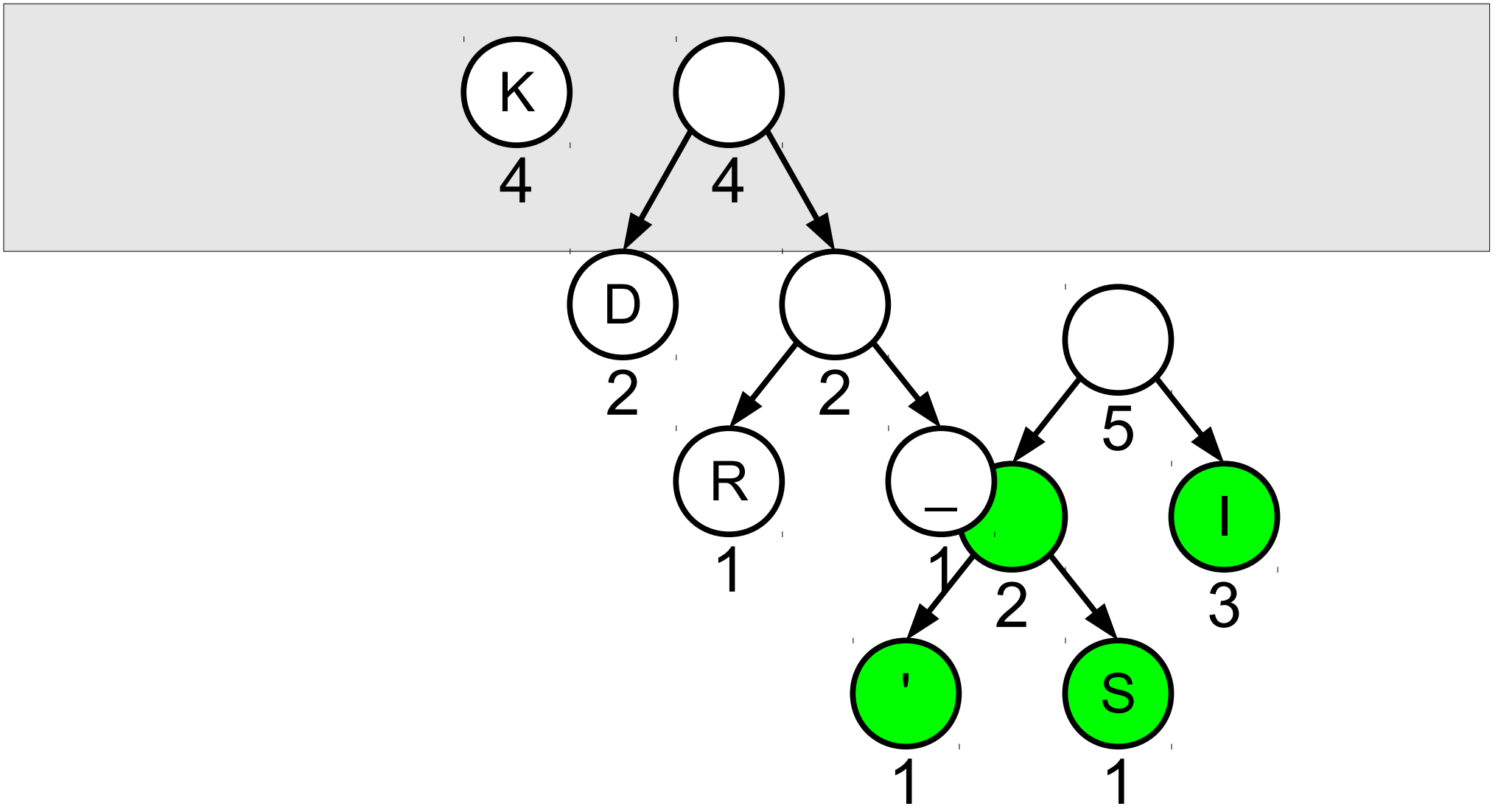
Huffman Coding



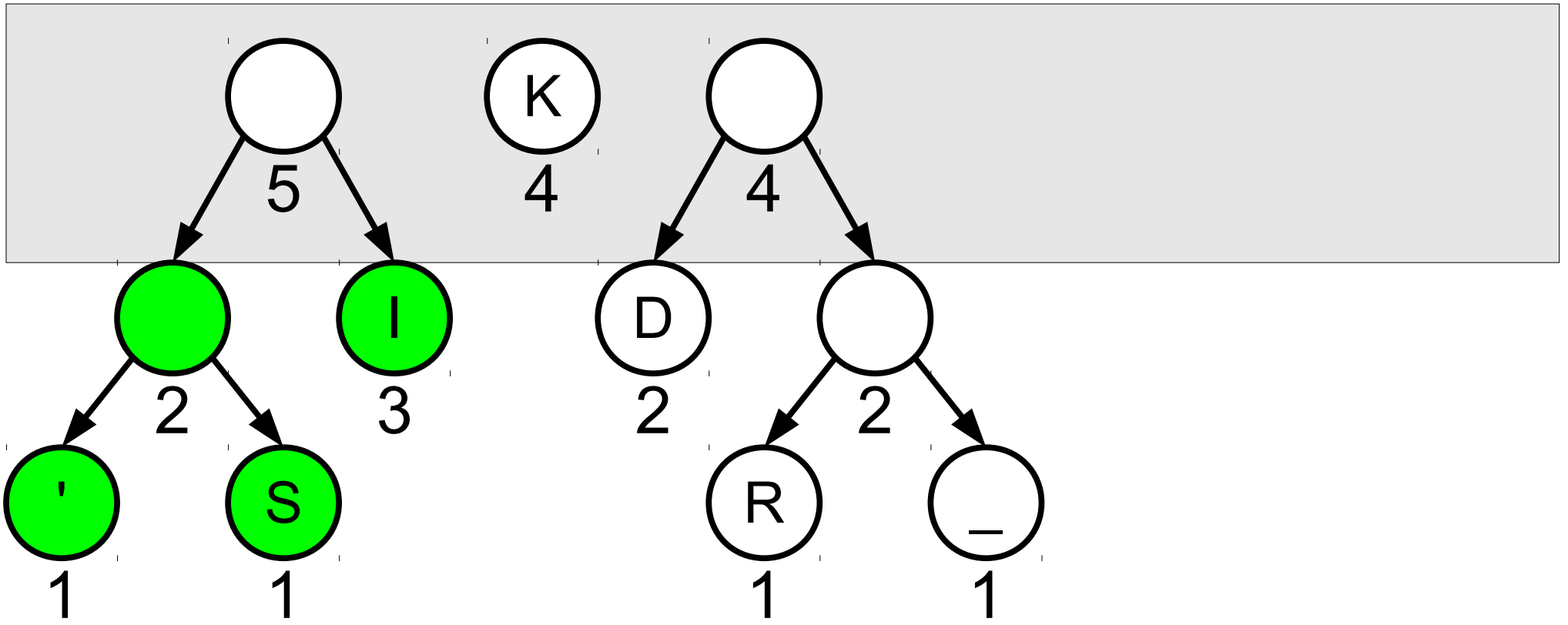
Huffman Coding



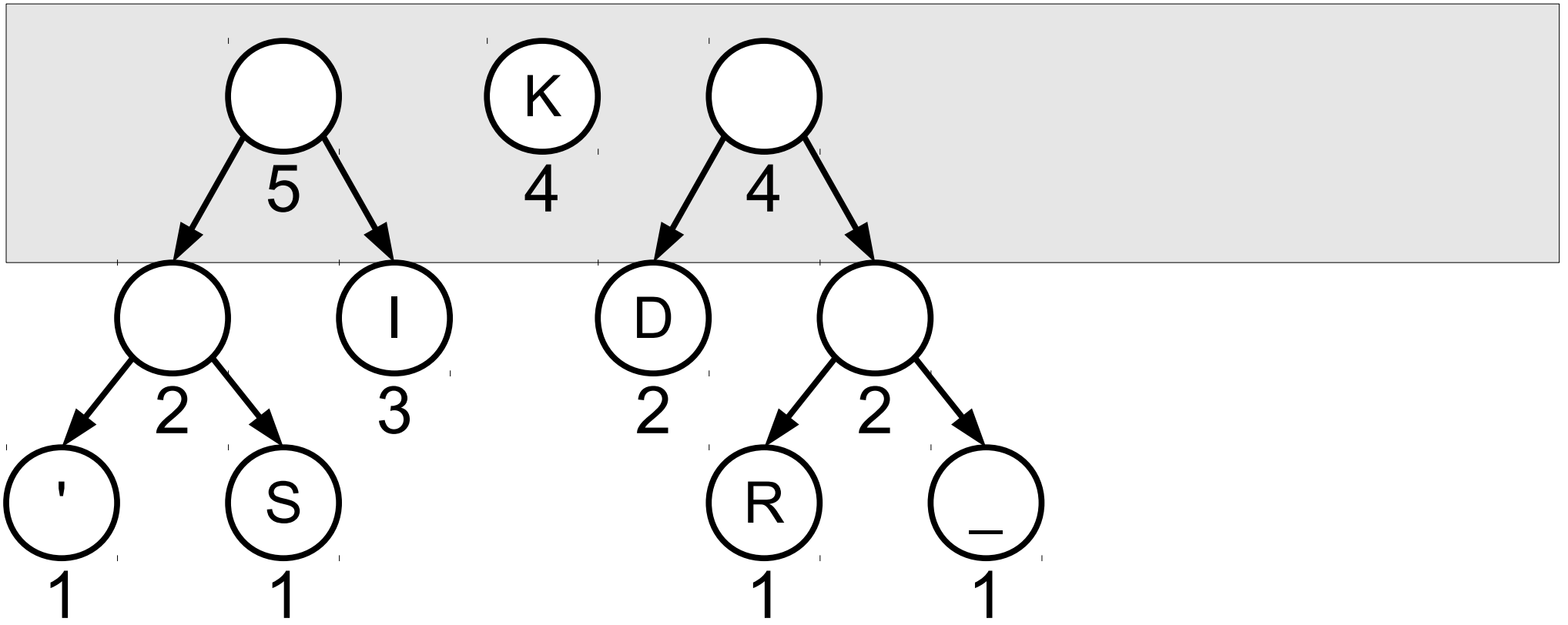
Huffman Coding



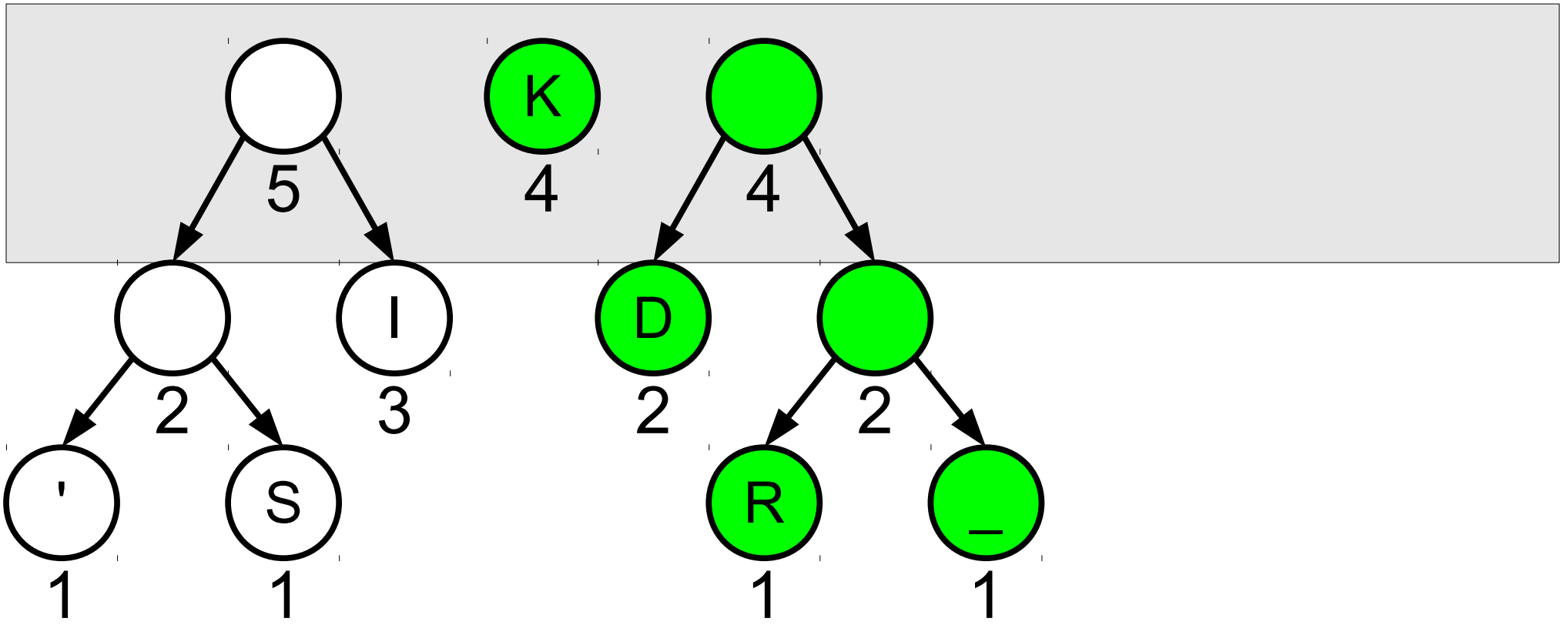
Huffman Coding



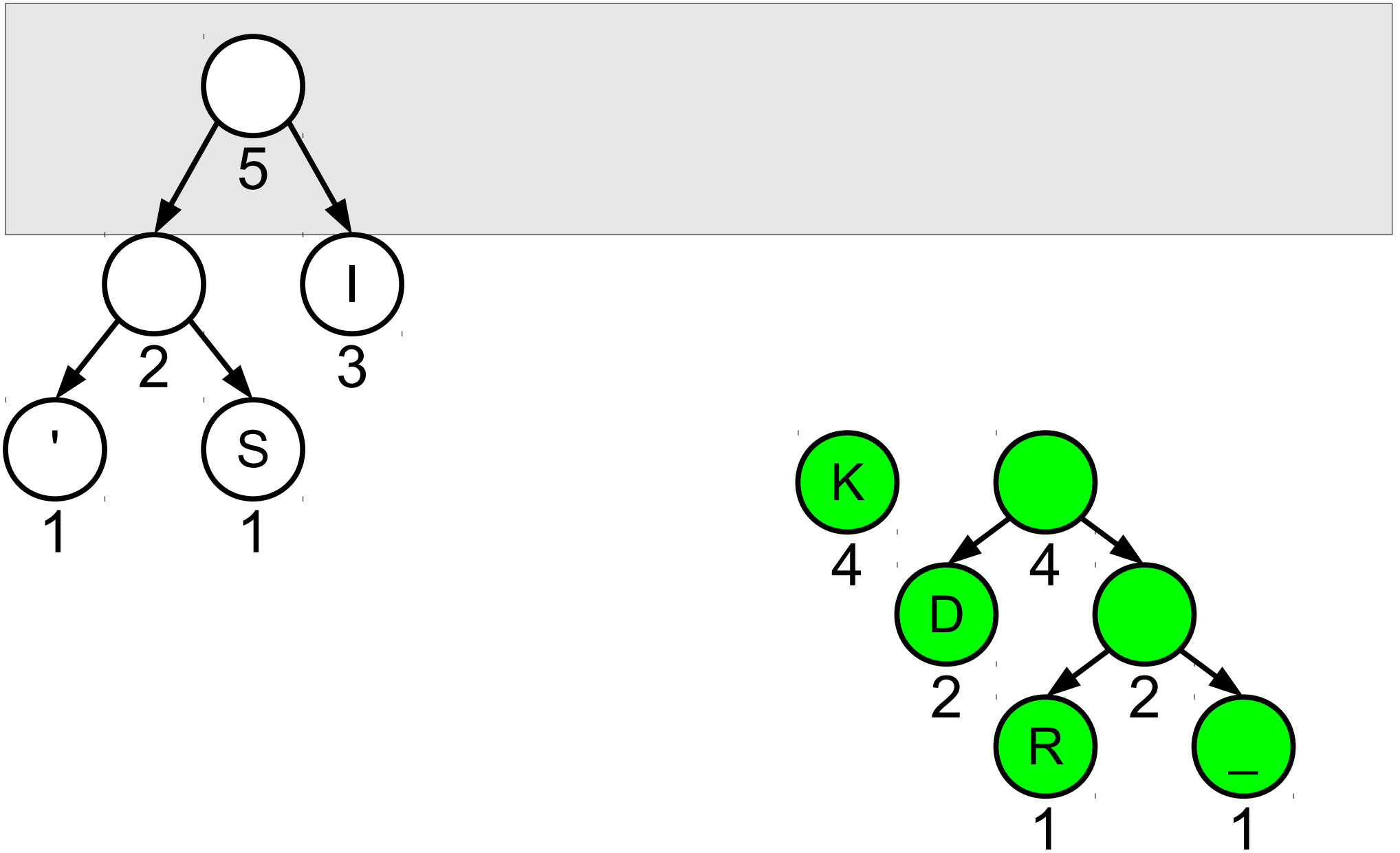
Huffman Coding



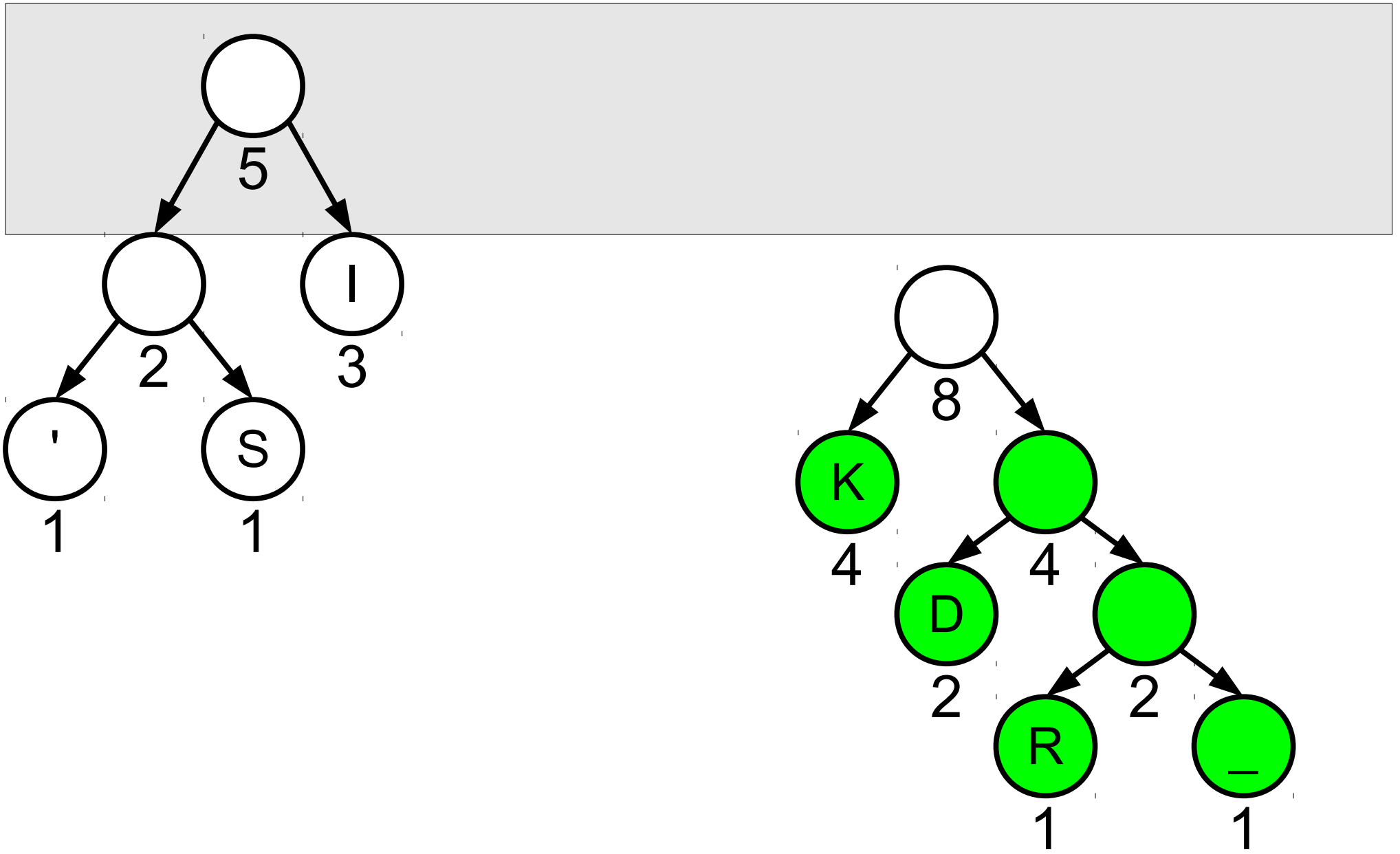
Huffman Coding



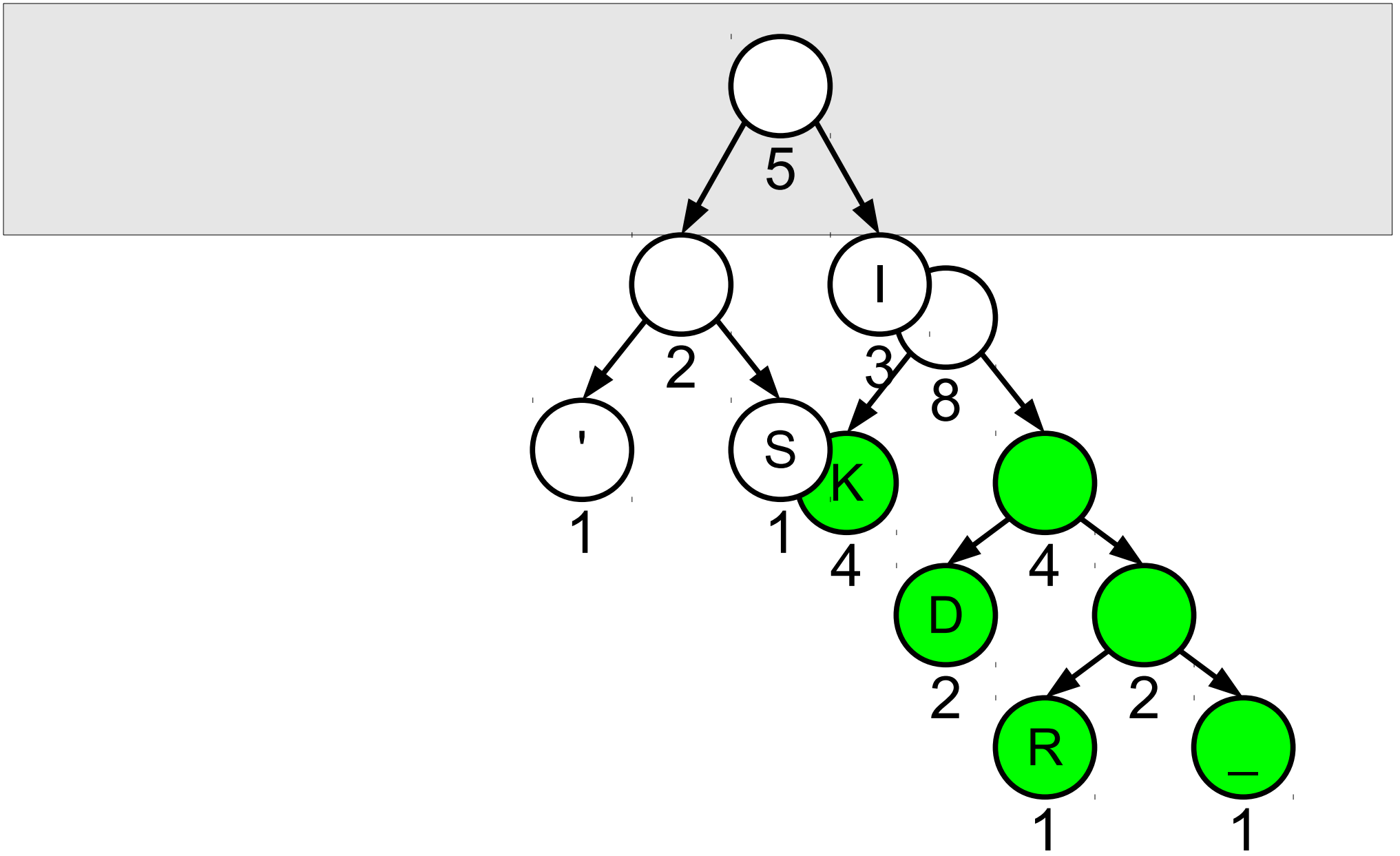
Huffman Coding



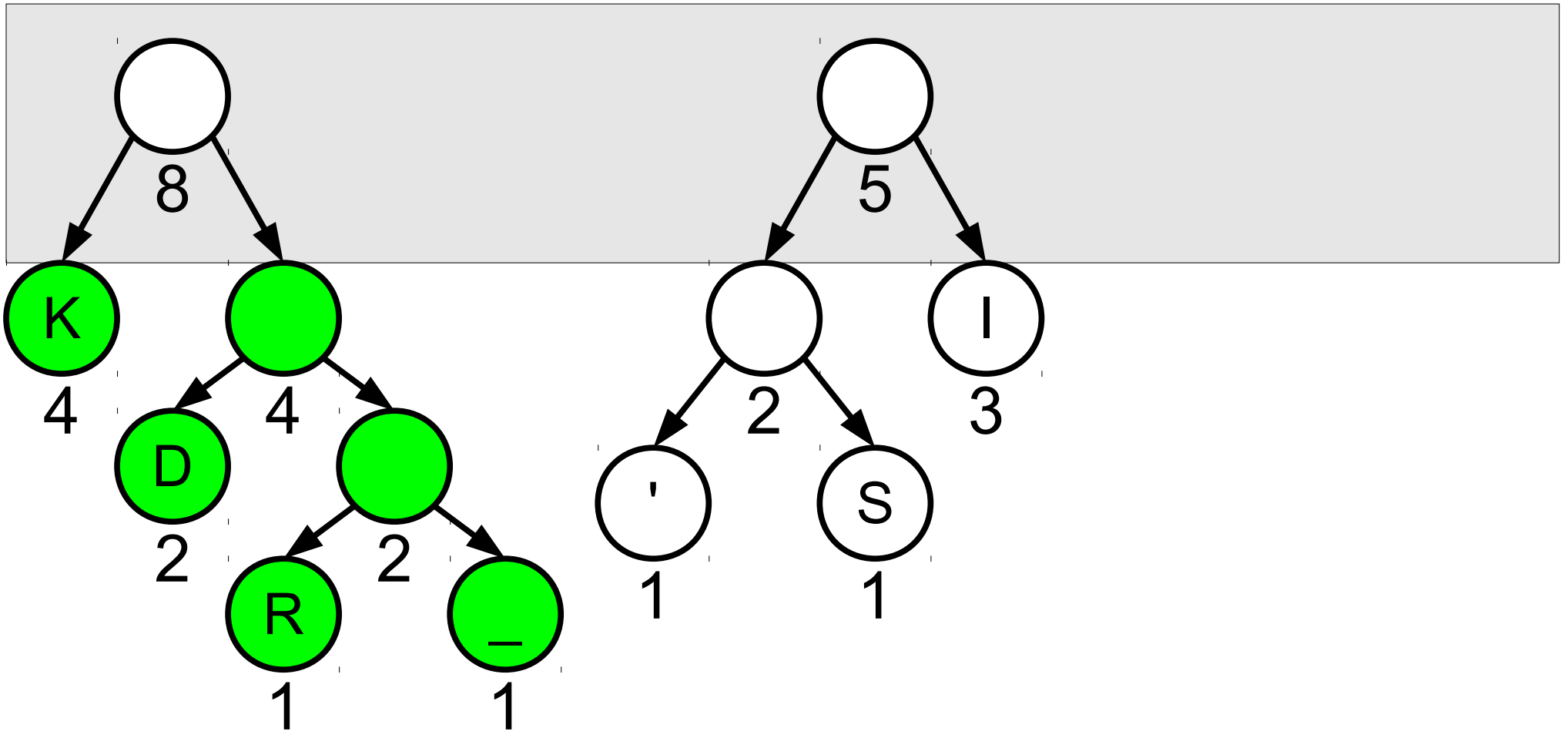
Huffman Coding



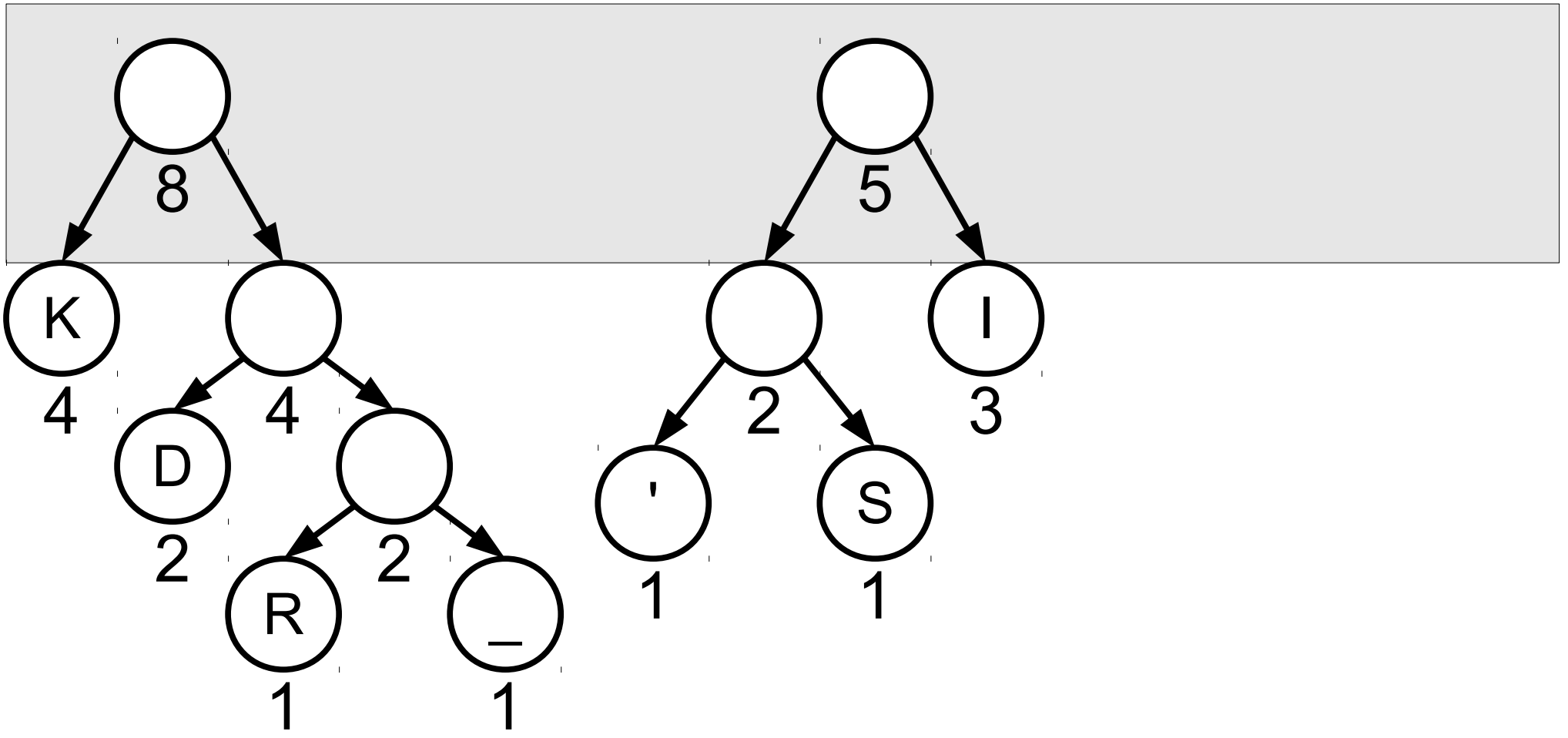
Huffman Coding



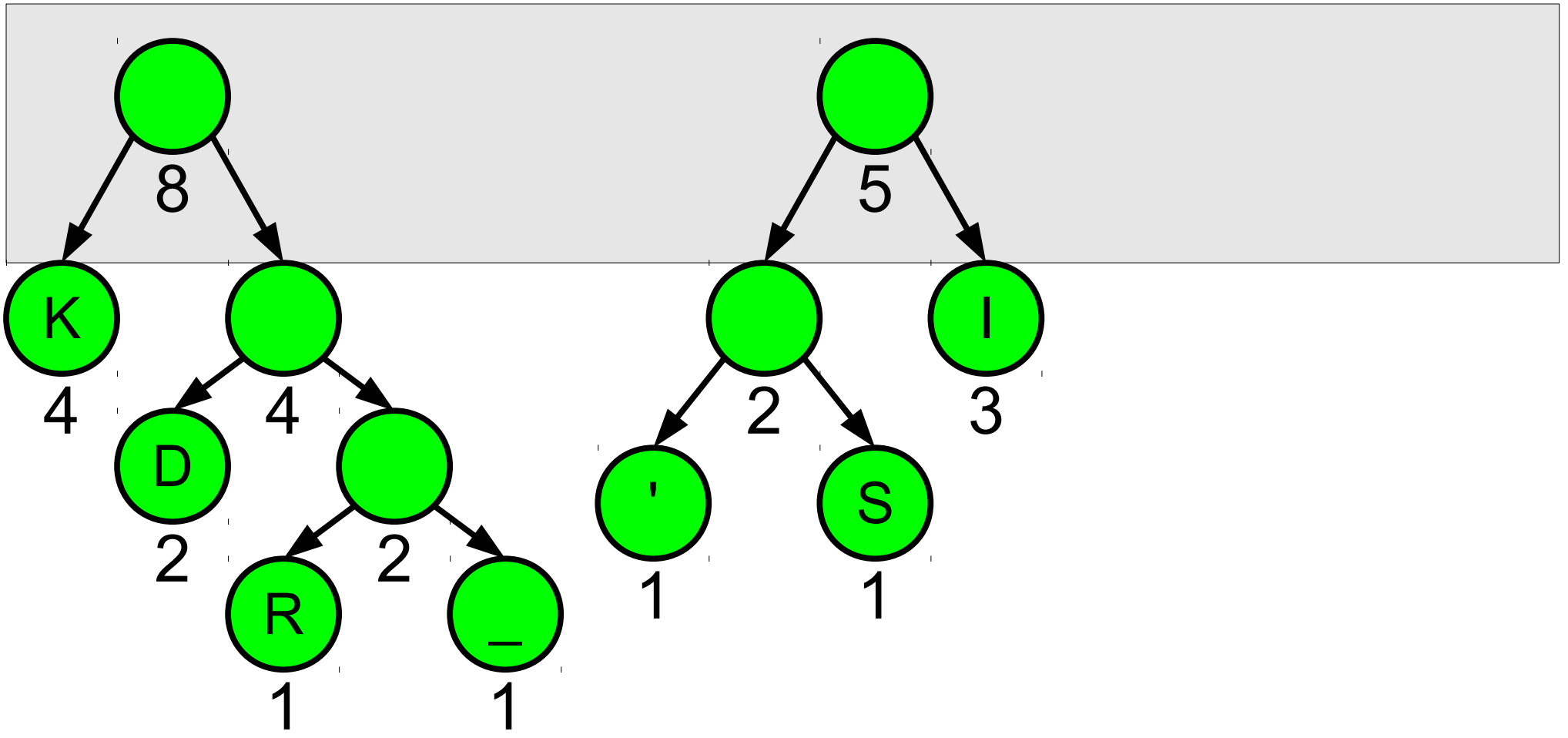
Huffman Coding



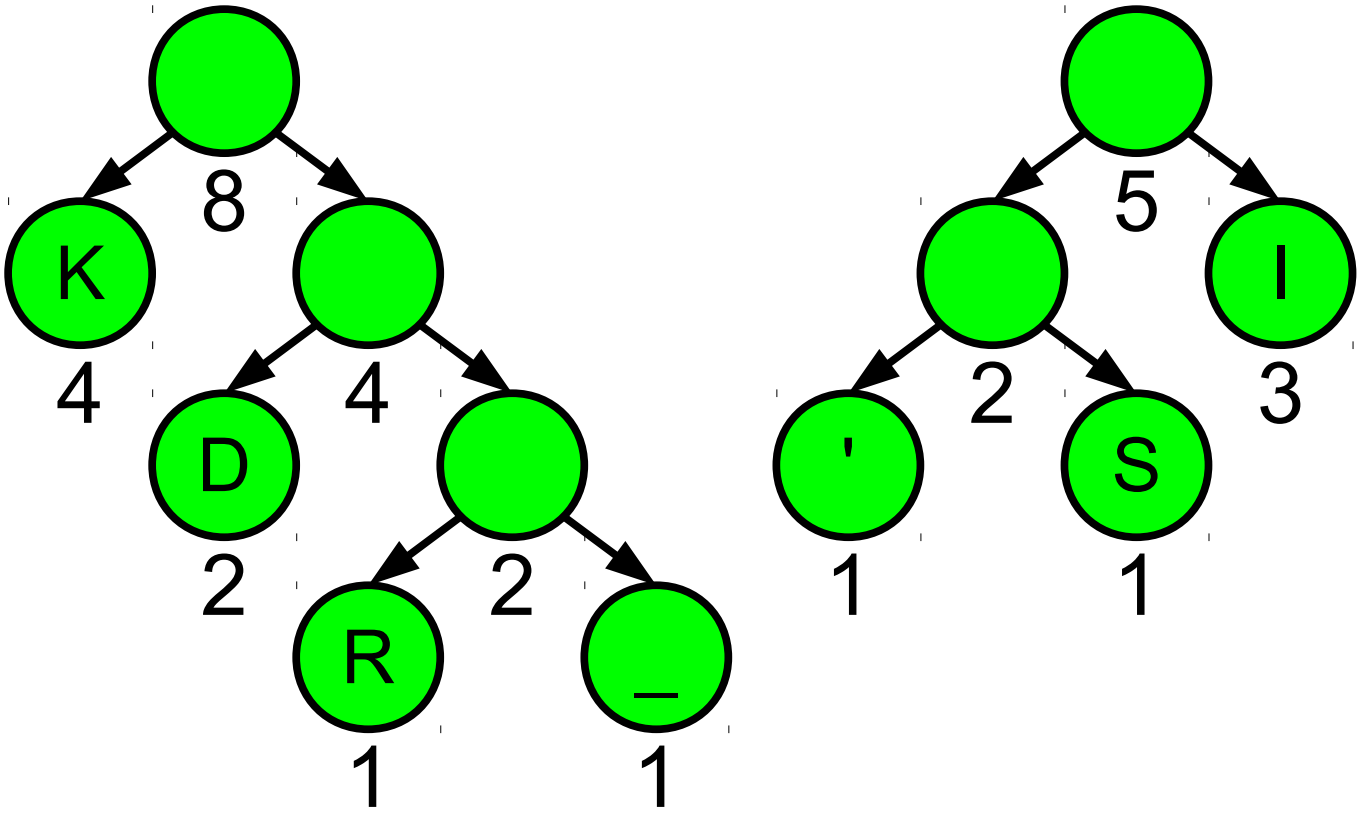
Huffman Coding



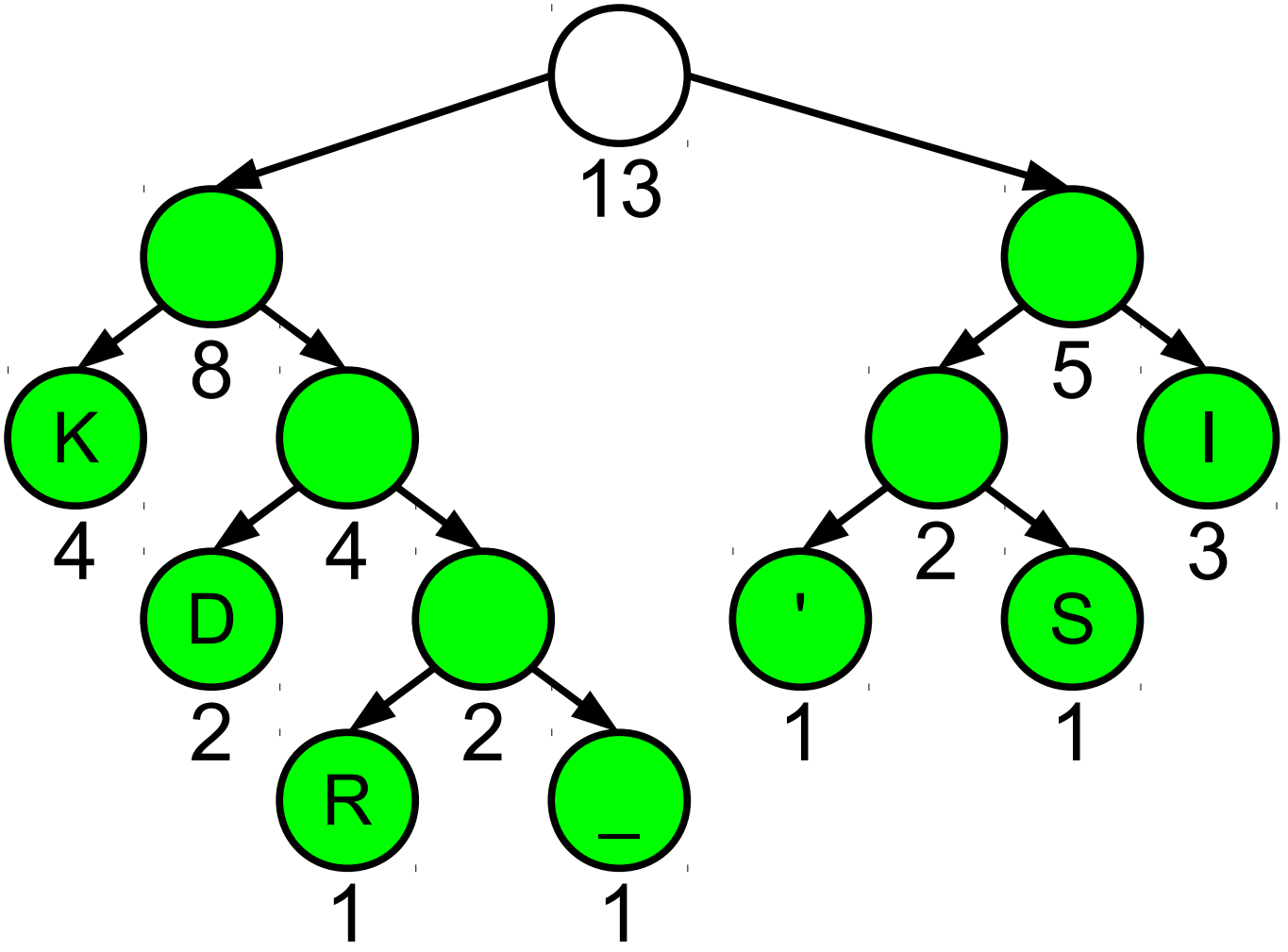
Huffman Coding



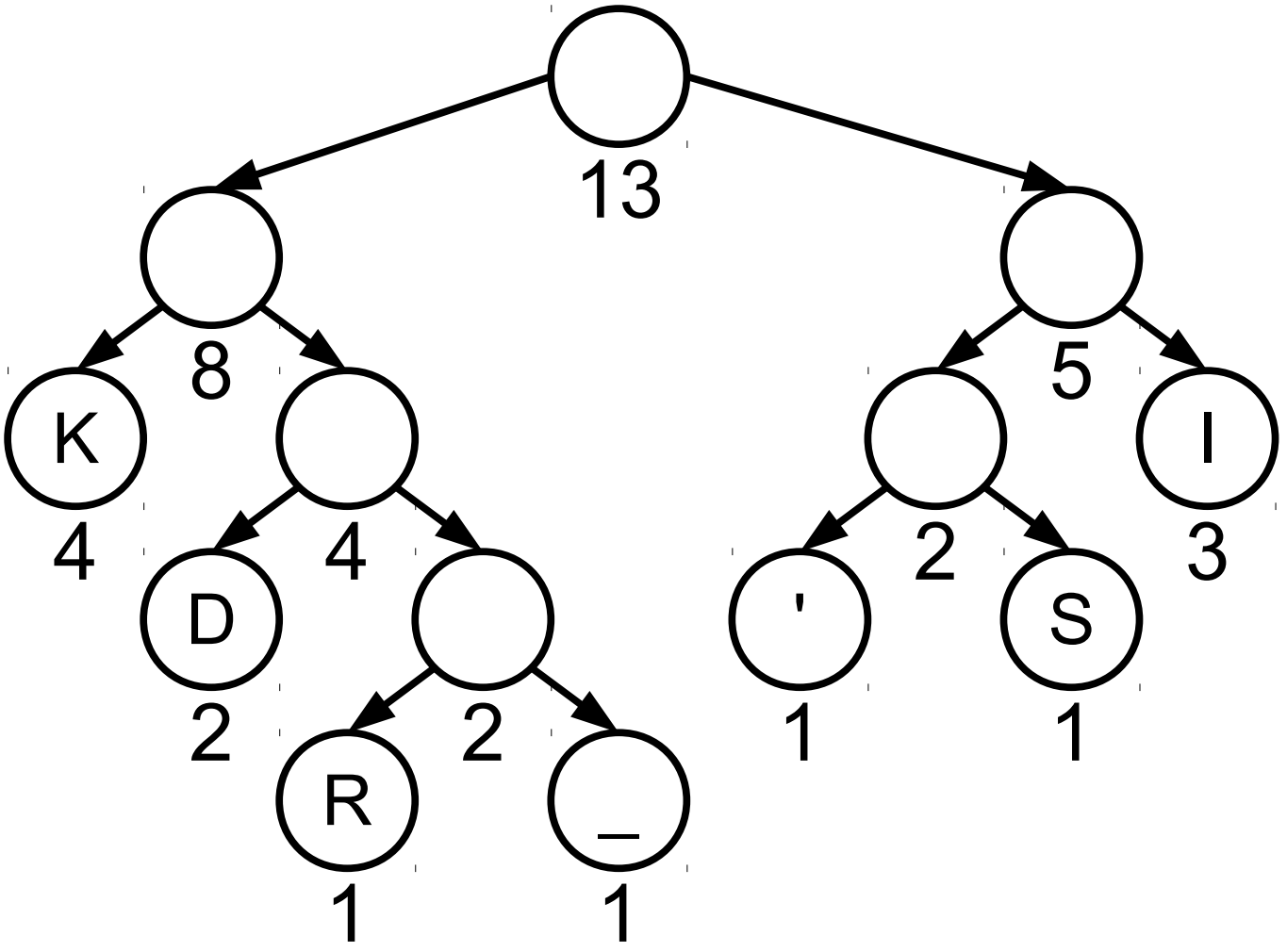
Huffman Coding



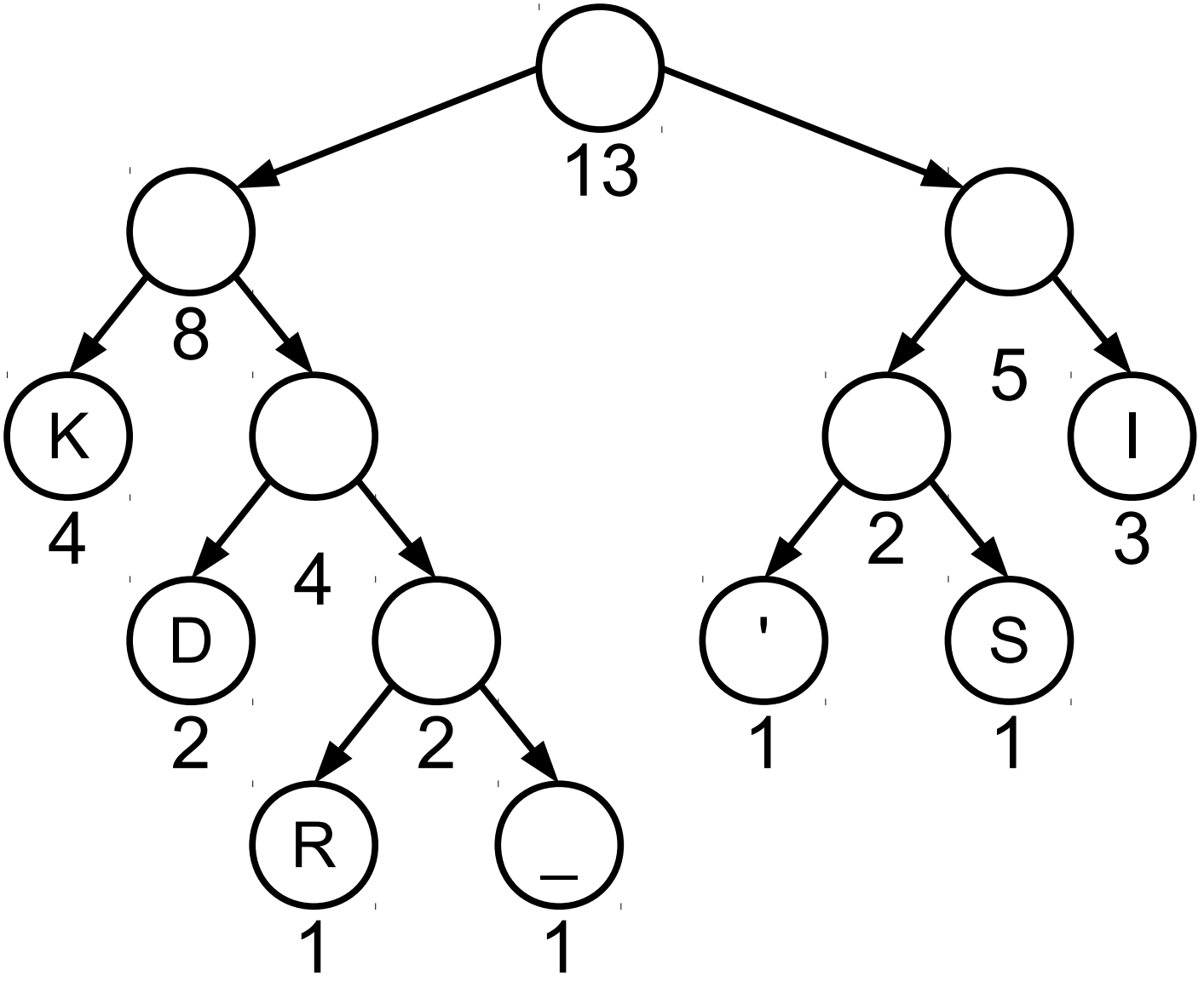
Huffman Coding



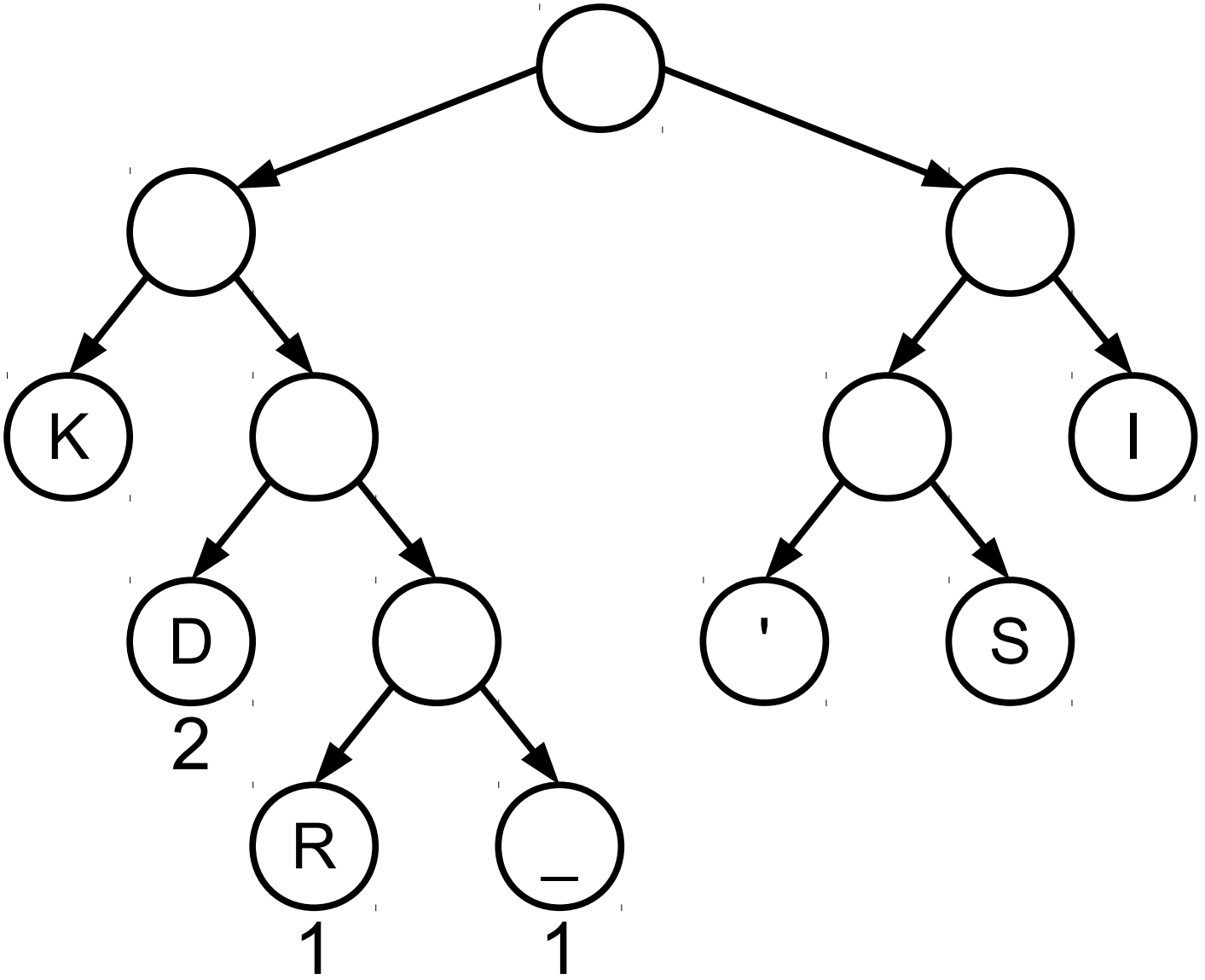
Huffman Coding



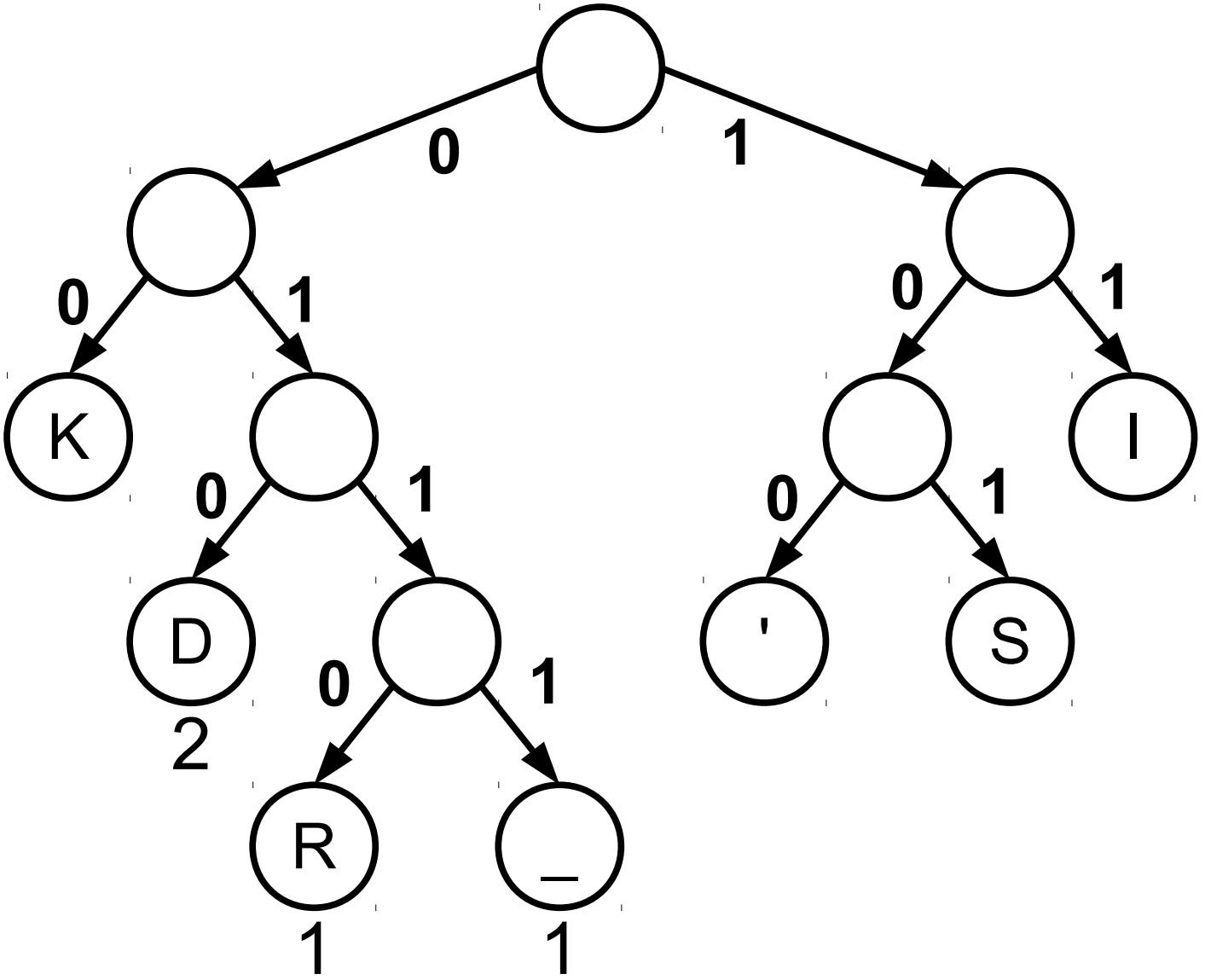
Huffman Coding



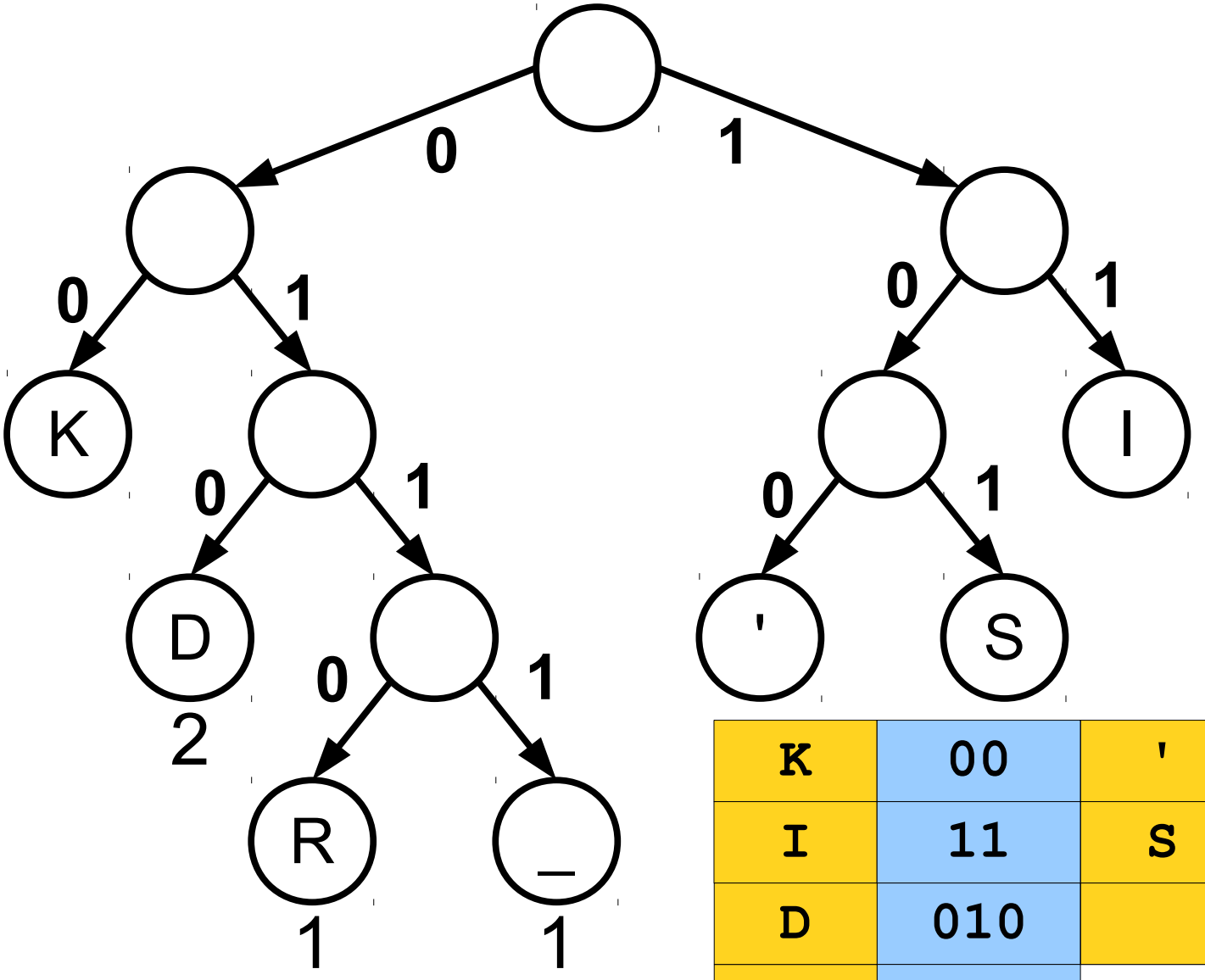
Huffman Coding



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

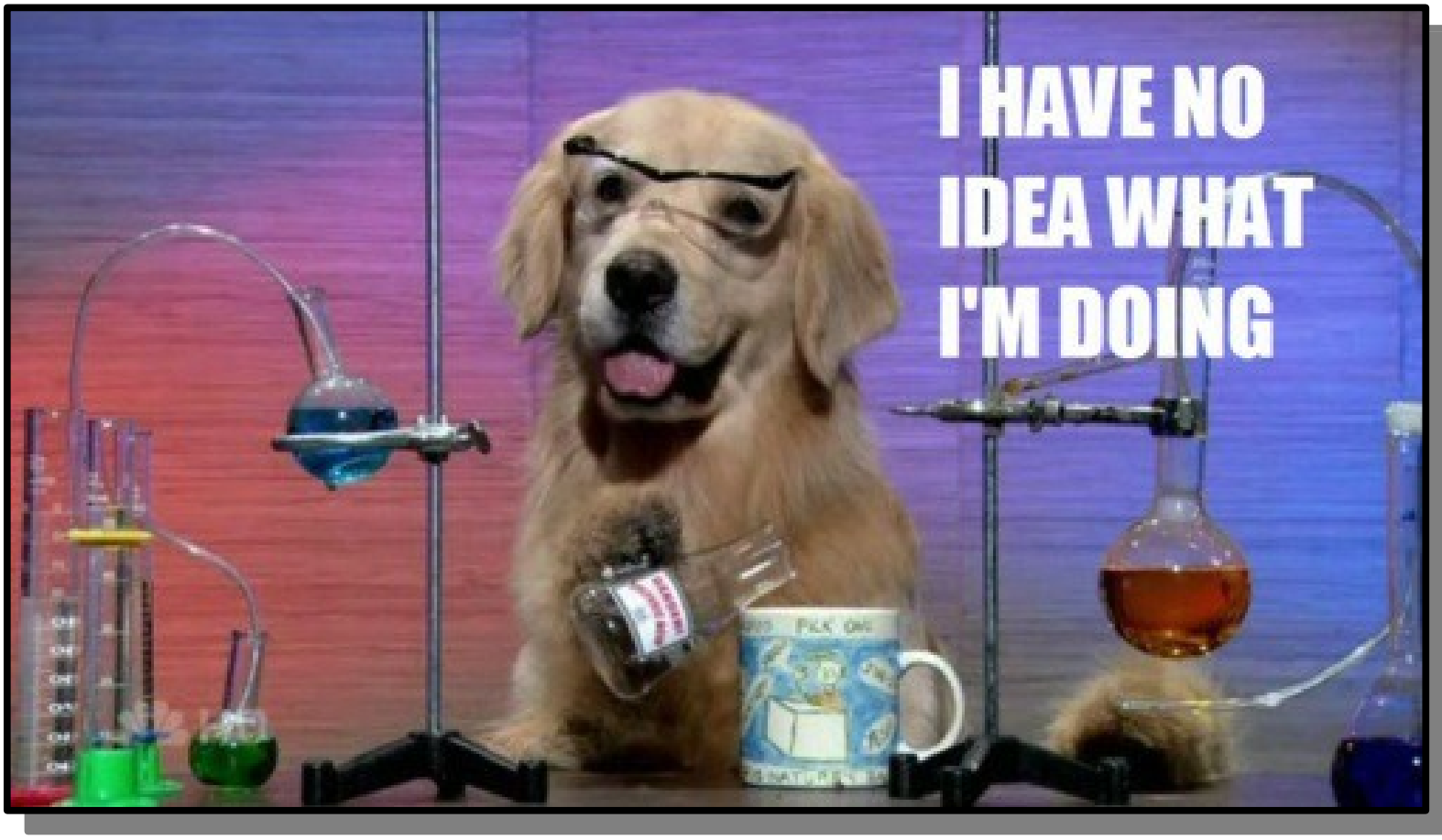
K	10
I	01
D	110
R	1111
'	001
S	000
	1110

```
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

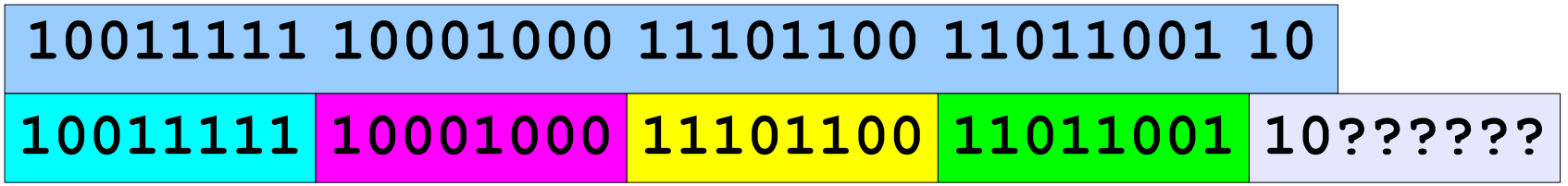
- This might increase the total file size!
- ***Theorem***: There is no compression algorithm that can always compress all inputs.
  - ***Proof:*** Take CS103!

One Last Thing...

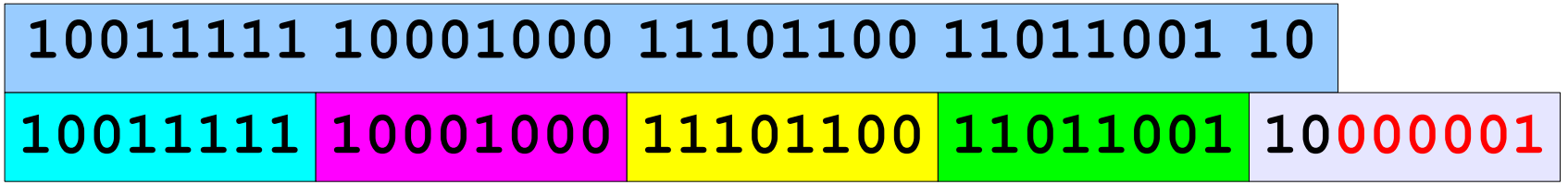
Bitten by Bytes

```
10011111100010001110110011011001l0
```

Bitten by Bytes



Bitten by Bytes



Bitten by Bytes

10011111	10001000	11101100	11011001	10000001

K	10
I	01
D	110
R	1111

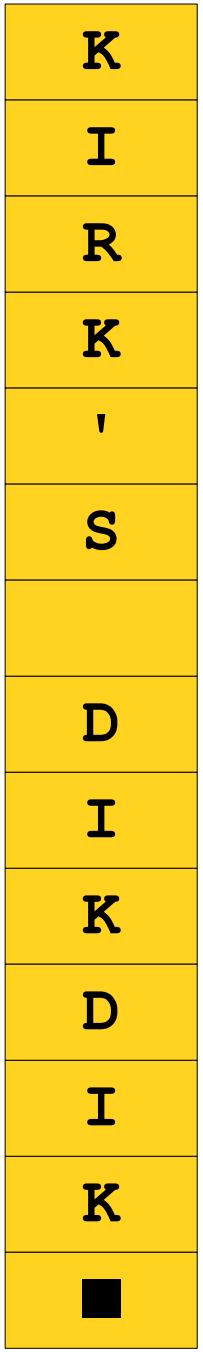
'	001
S	000
	1110

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	'

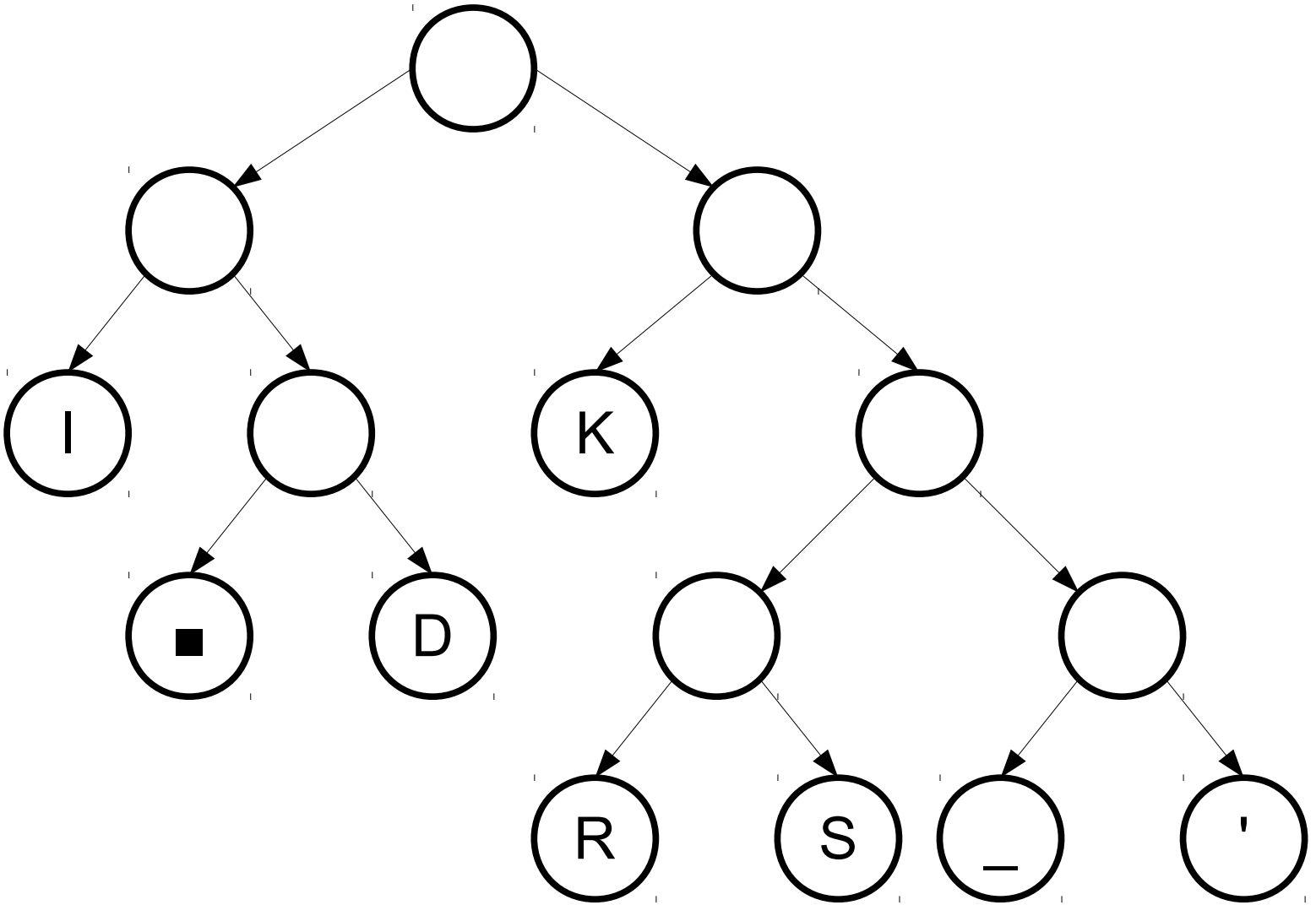
Spare Bits

- 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

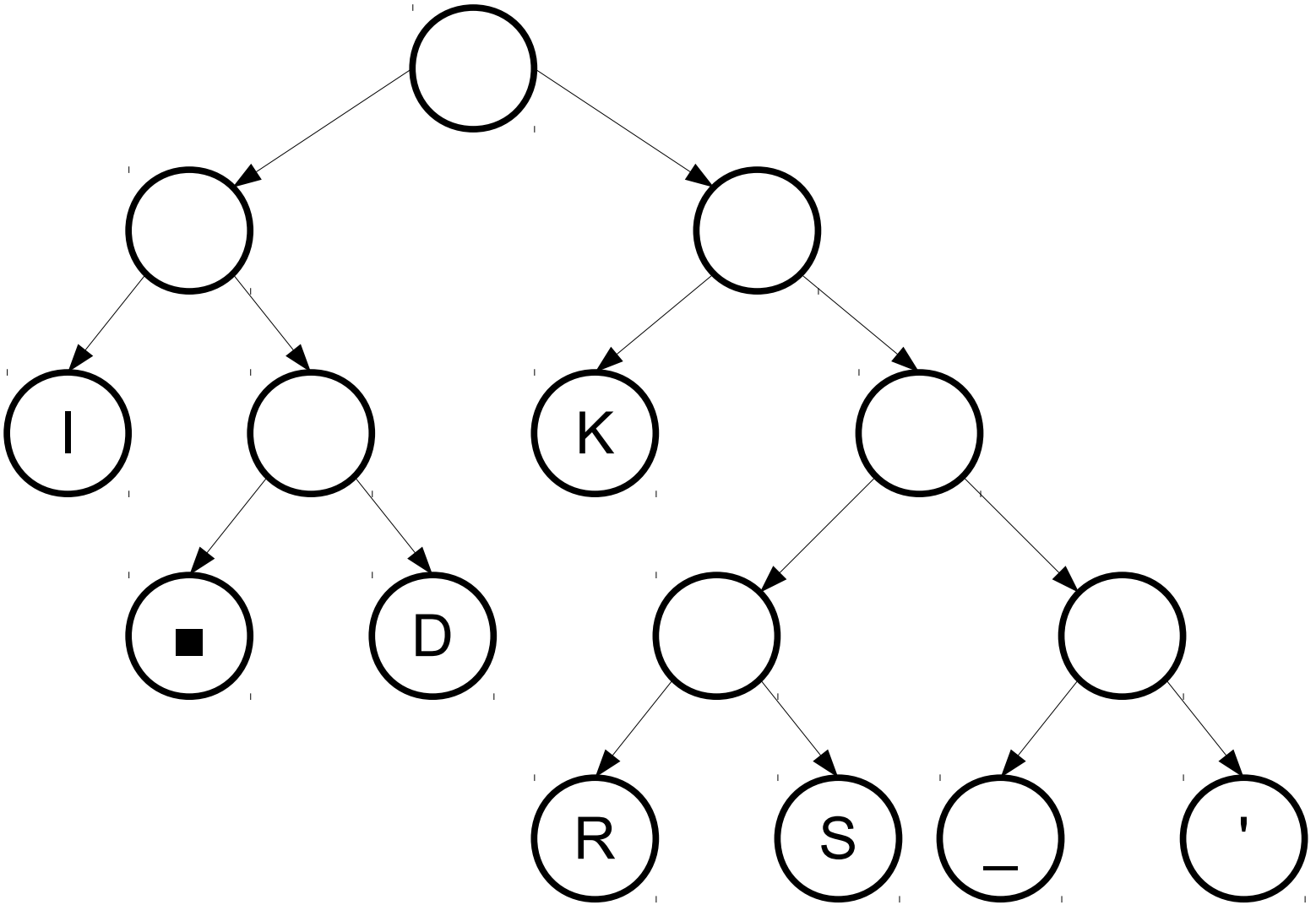


K	4
I	3
D	2
R	1
'	1
S	1
	1
■	1



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 trees.
- 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-English characters or more general symbols?

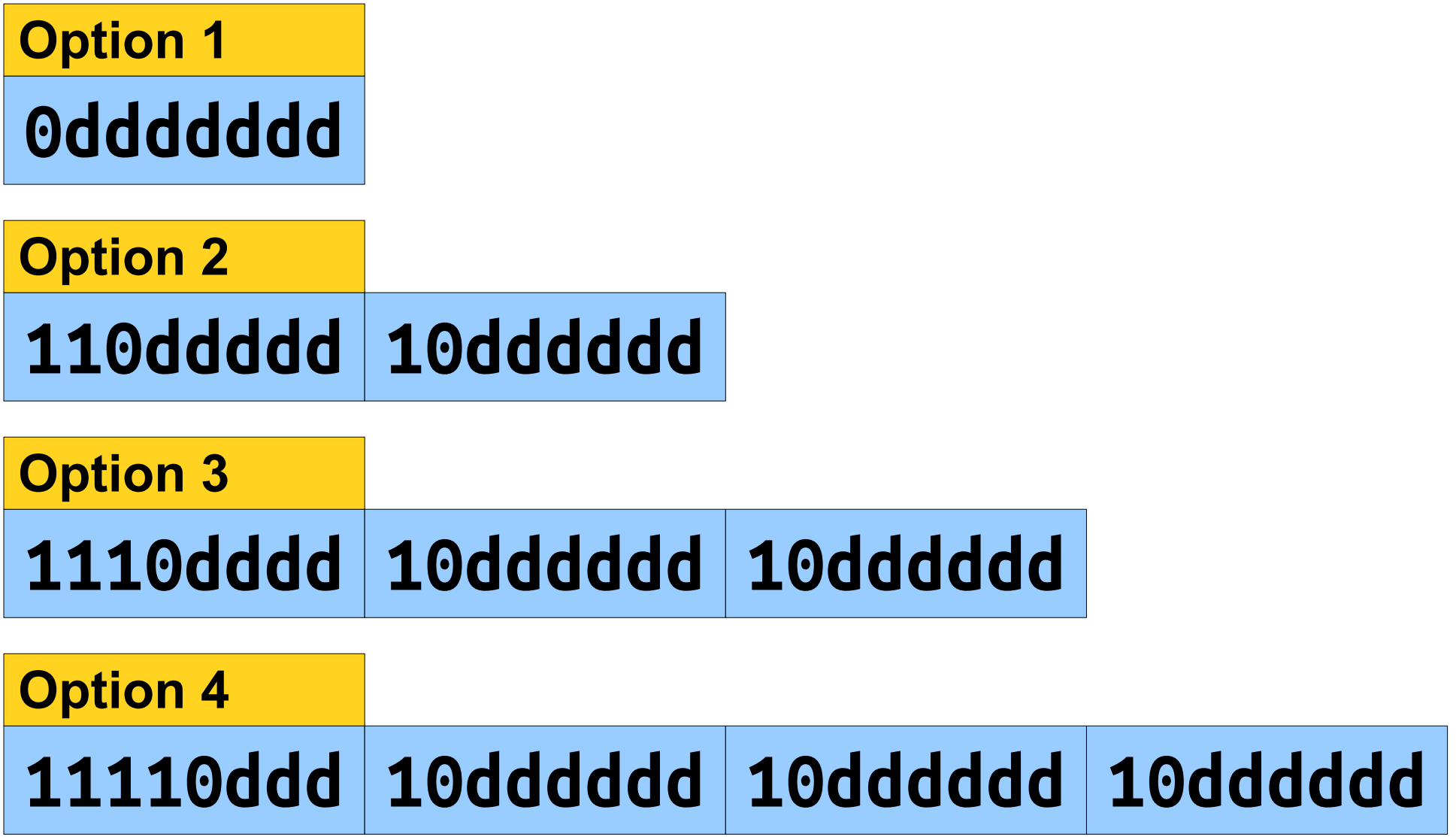
Beyond ASCII

- 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-English characters or more general symbols?
- For example:
  - ¿Cómo estás?
  - السلام عليكم
  - (ಠ°□°) 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

```
11100000100111111001010110001100
```

UTF-8

```
11100000 10011111 10010101 10001100
```

UTF-8

11100000	10011111	10010101	10001100
11100000	10011111	10010101	10001100

UTF-8

11100000	10011111	10010101	10001100
1110**0000**	10**011111**	10**010101**	10**001100**

UTF-8

11100000	10011111	10010101	10001100
1110**0000**	10**011111**	10**010101**	10**001100**

00000111111010101001100

UTF-8

11100000	10011111	10010101	10001100
1110**0000**	10**011111**	10**010101**	10**001100**

00000111111010101001100

Further Topics

More to Explore

- ***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.
- ***Binary Tries***
  - Other applications of trees like these!