Beyond Data Structures

Announcements

- Assignment 5 due right now.
- Assignment 6 (**Huffman Encoding**) out, due Monday, June 4th at 10:00AM.
  - Build a program to compress files!
  - See an awesome application of trees and priority queues.
  - More on that later...

Beyond Data Structures

Goals for Today

- Explore applications of techniques from data structures in other settings.
- Two major examples:
  - Binary trees and tries: **Huffman Encoding**
  - Hash functions: **Secure Passwords**

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 (Latin 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-Latin 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.

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

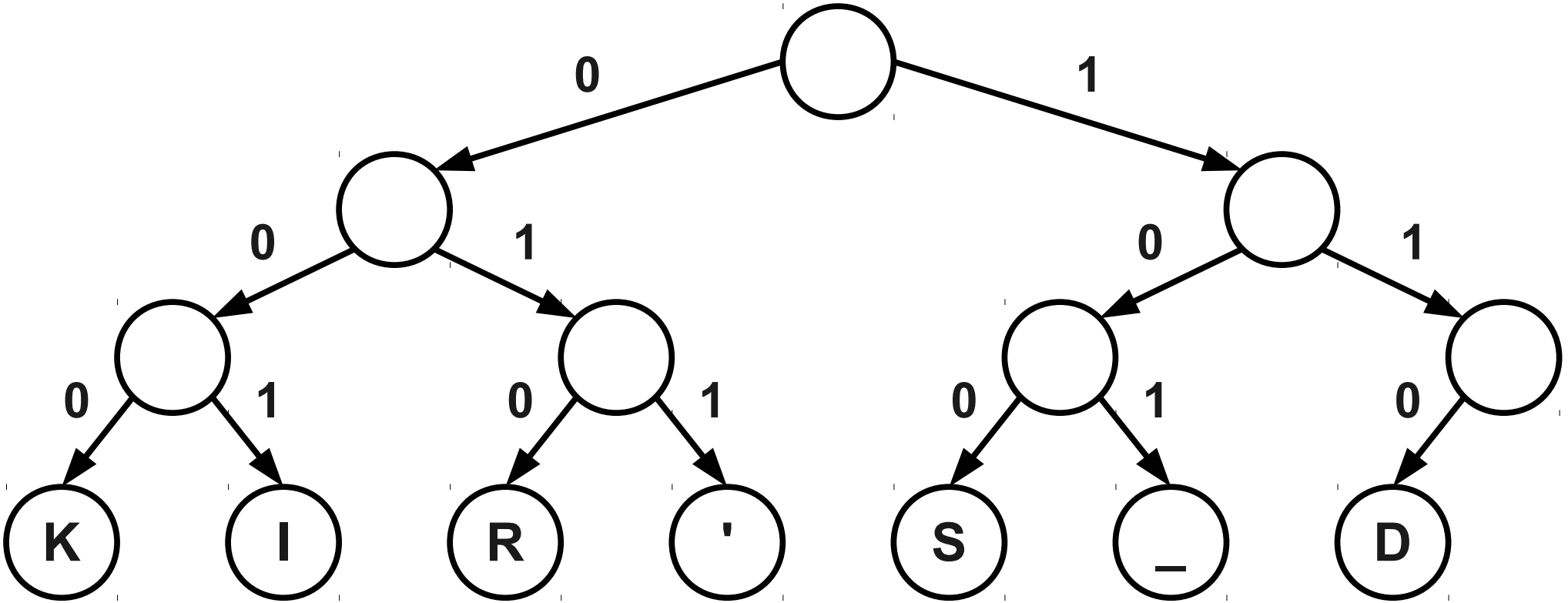
- Using this prefix code, we can represent KIRK'S DIKDIK as the sequence

  `10011111100010001110110011011100110`

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

Finding Optimal Prefix Codes

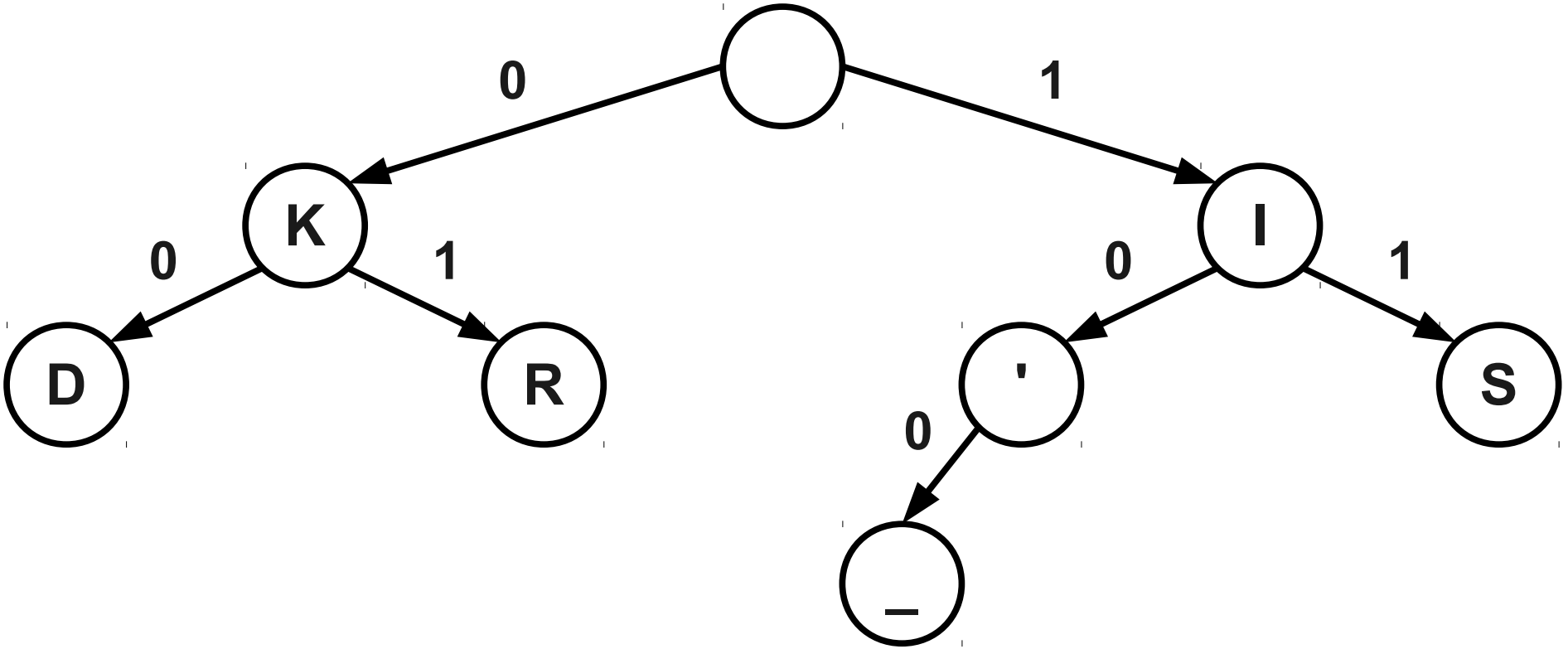
The Key Idea



K	000
I	001
R	010
'	011

S	100
	101
D	110

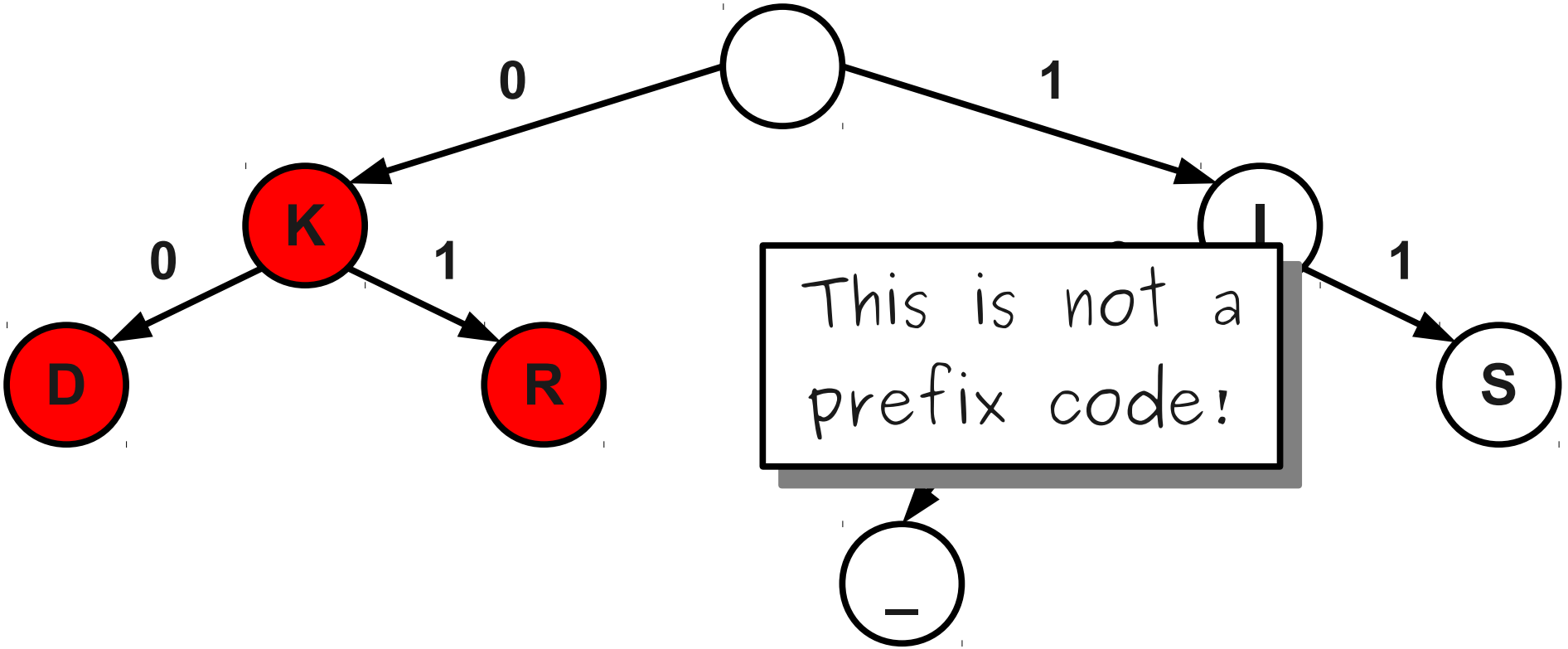
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

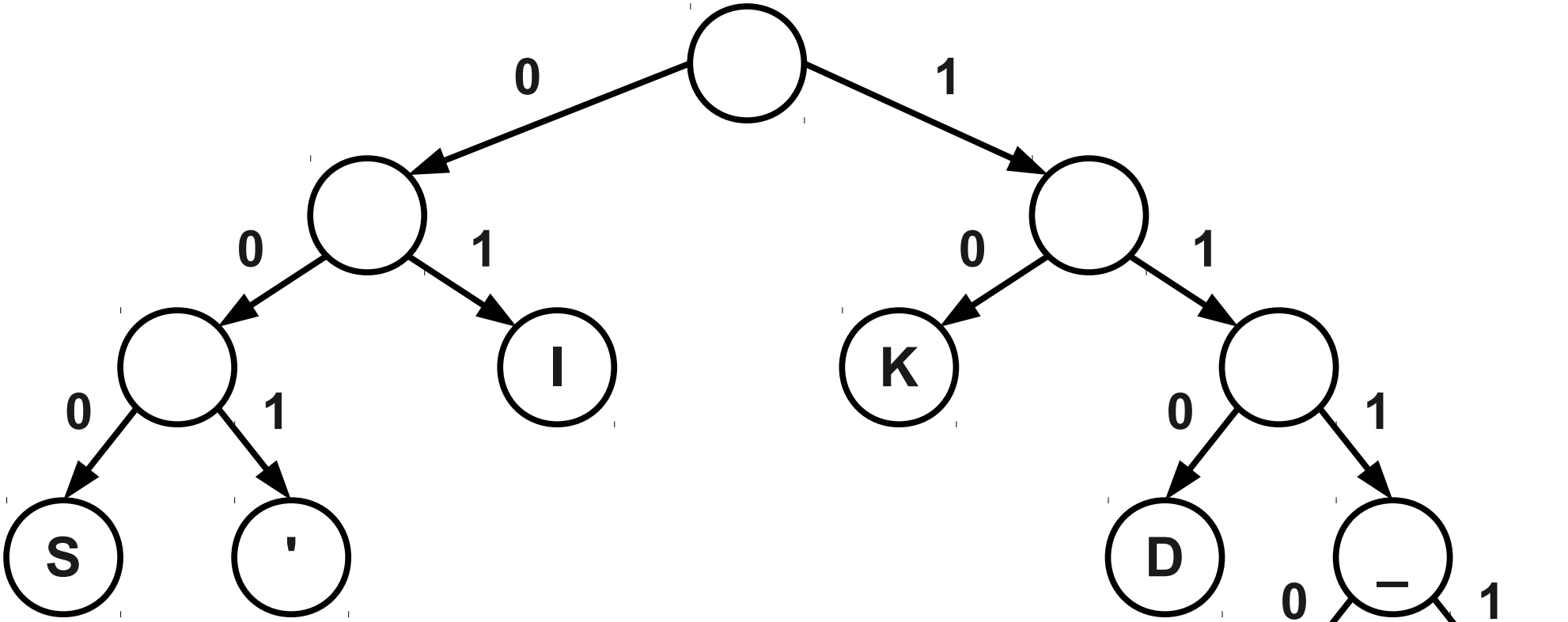
The Key Idea



K	0
I	1
D	00
R	01

'	10
S	11
	100

The Key Idea

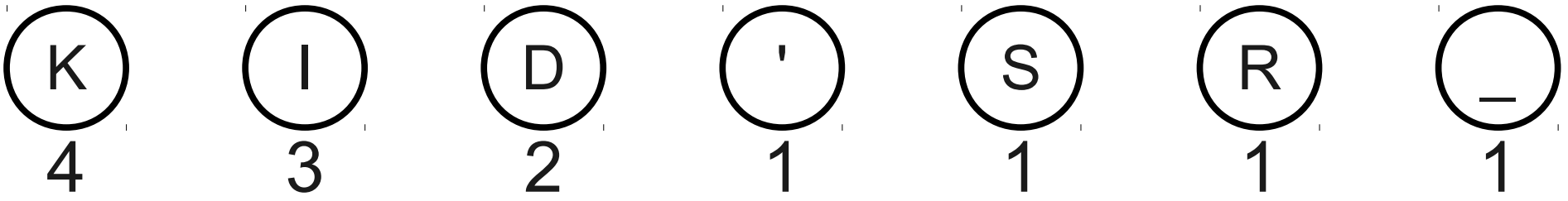


K	10
I	01
D	110
R	1111

'	001
S	000
	1110

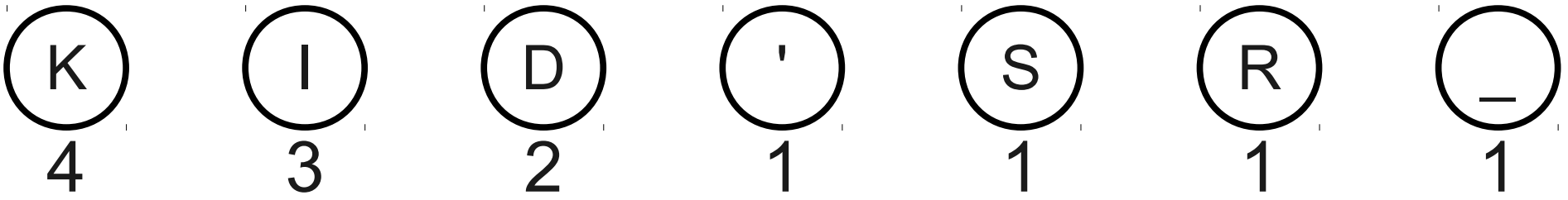
How do we find the best prefix-free binary tree?

Huffman Coding

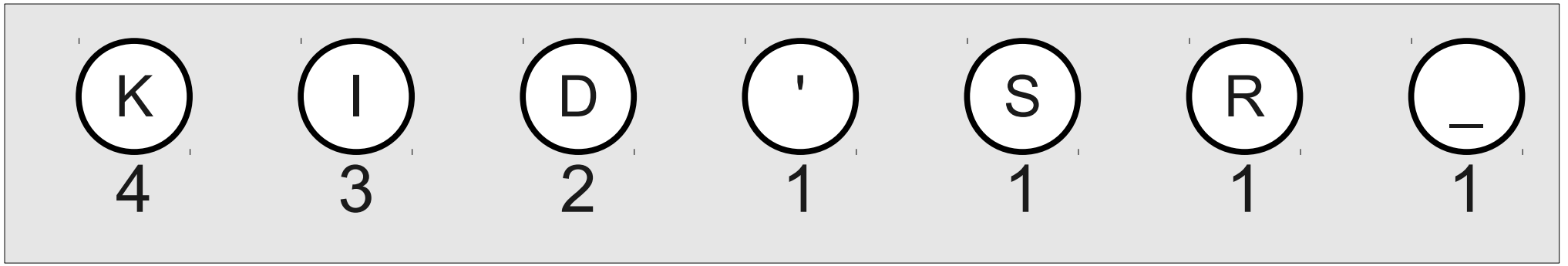


K	4
I	3
D	2
R	1
'	1
S	1
	1

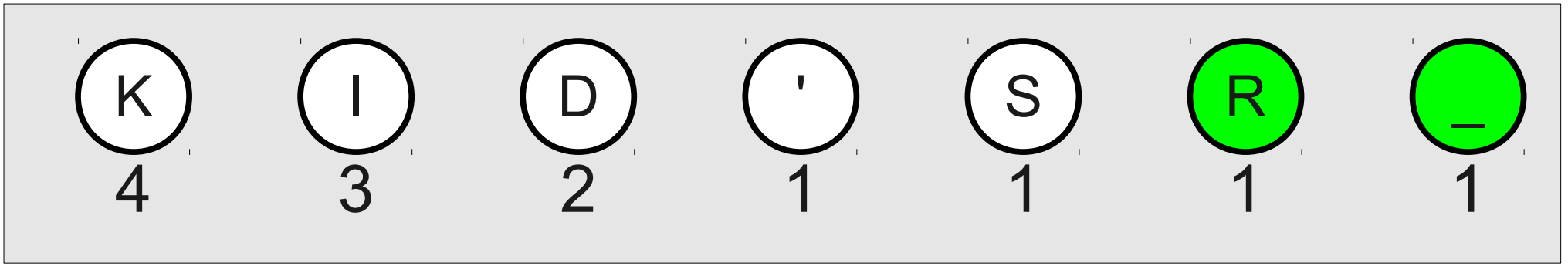
Huffman Coding



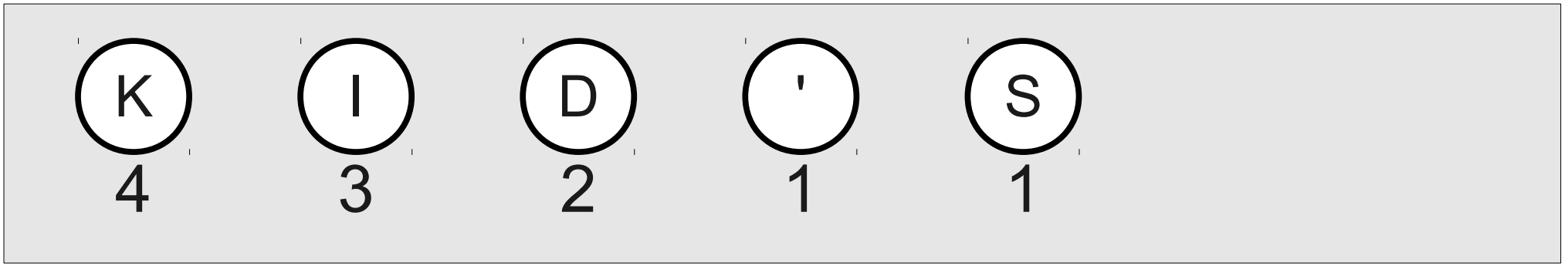
Huffman Coding

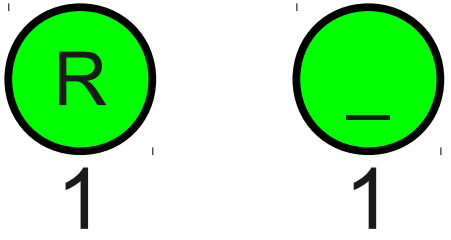


Huffman Coding

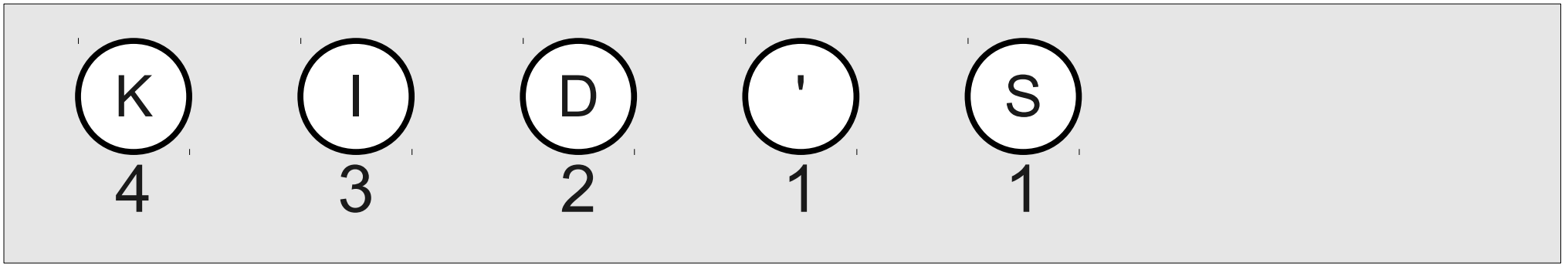


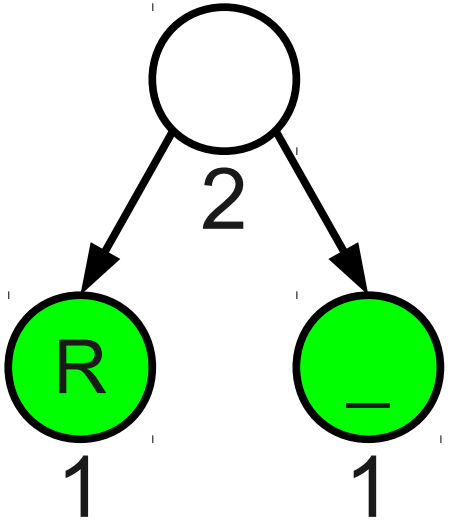
Huffman Coding



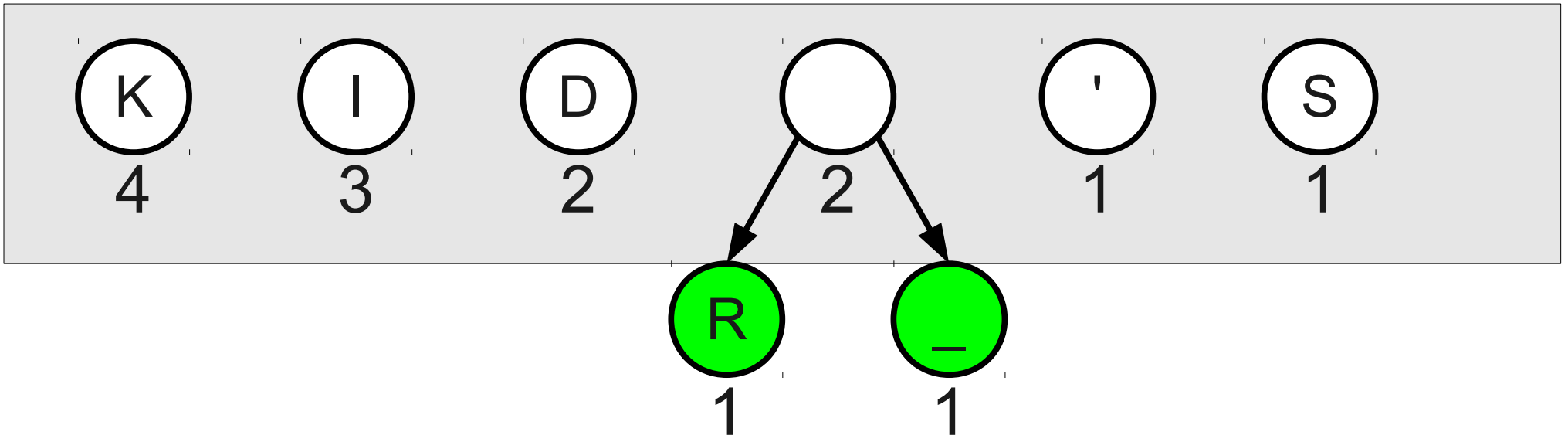


Huffman Coding

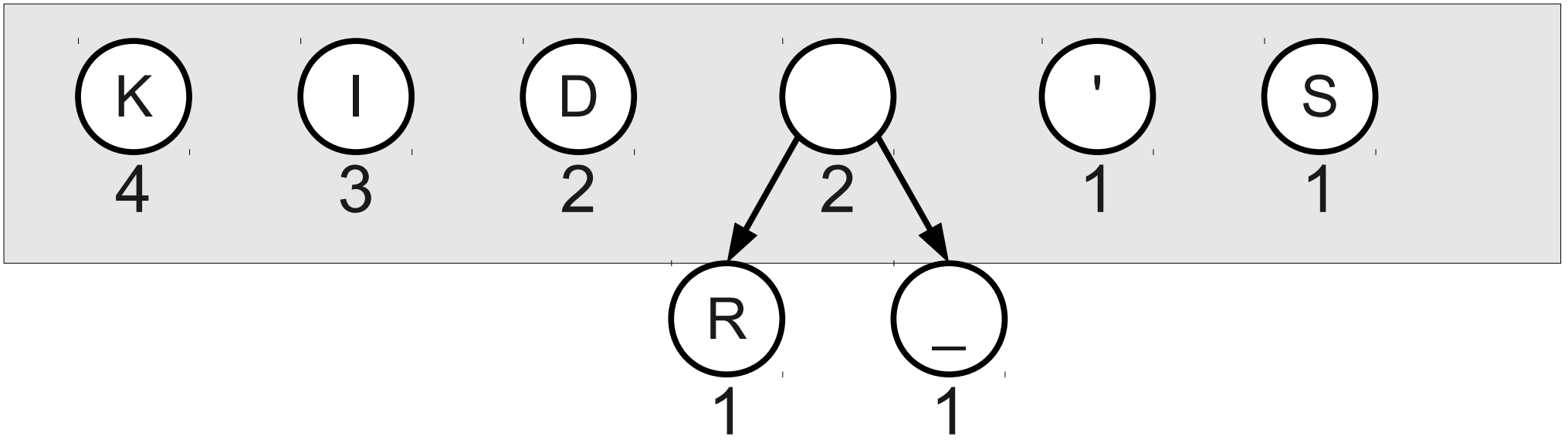




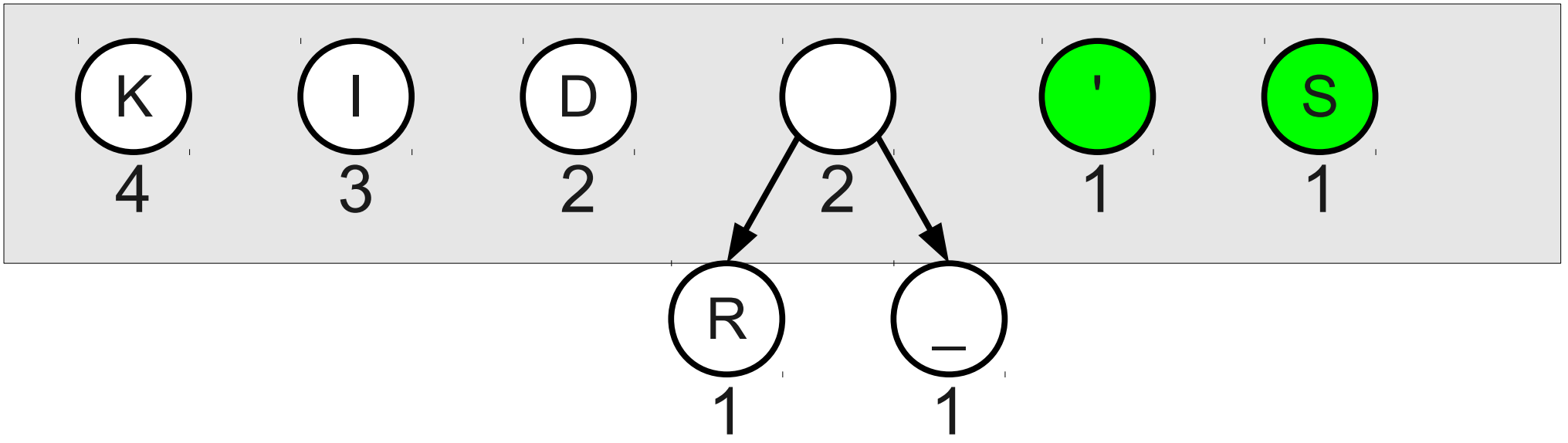
Huffman Coding



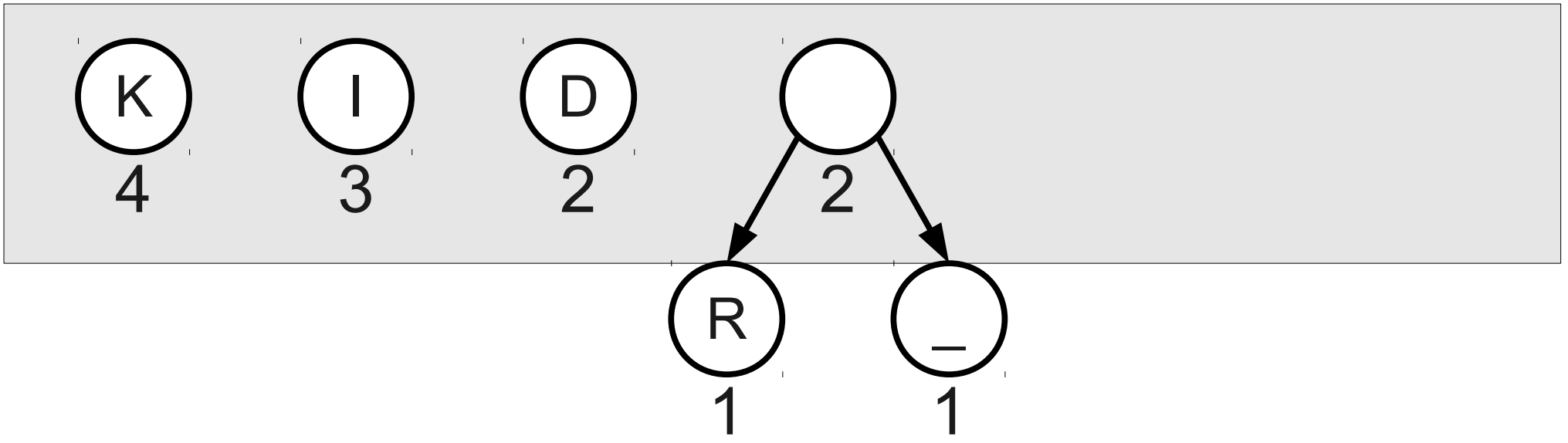
Huffman Coding

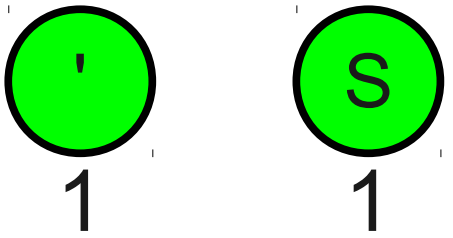


Huffman Coding

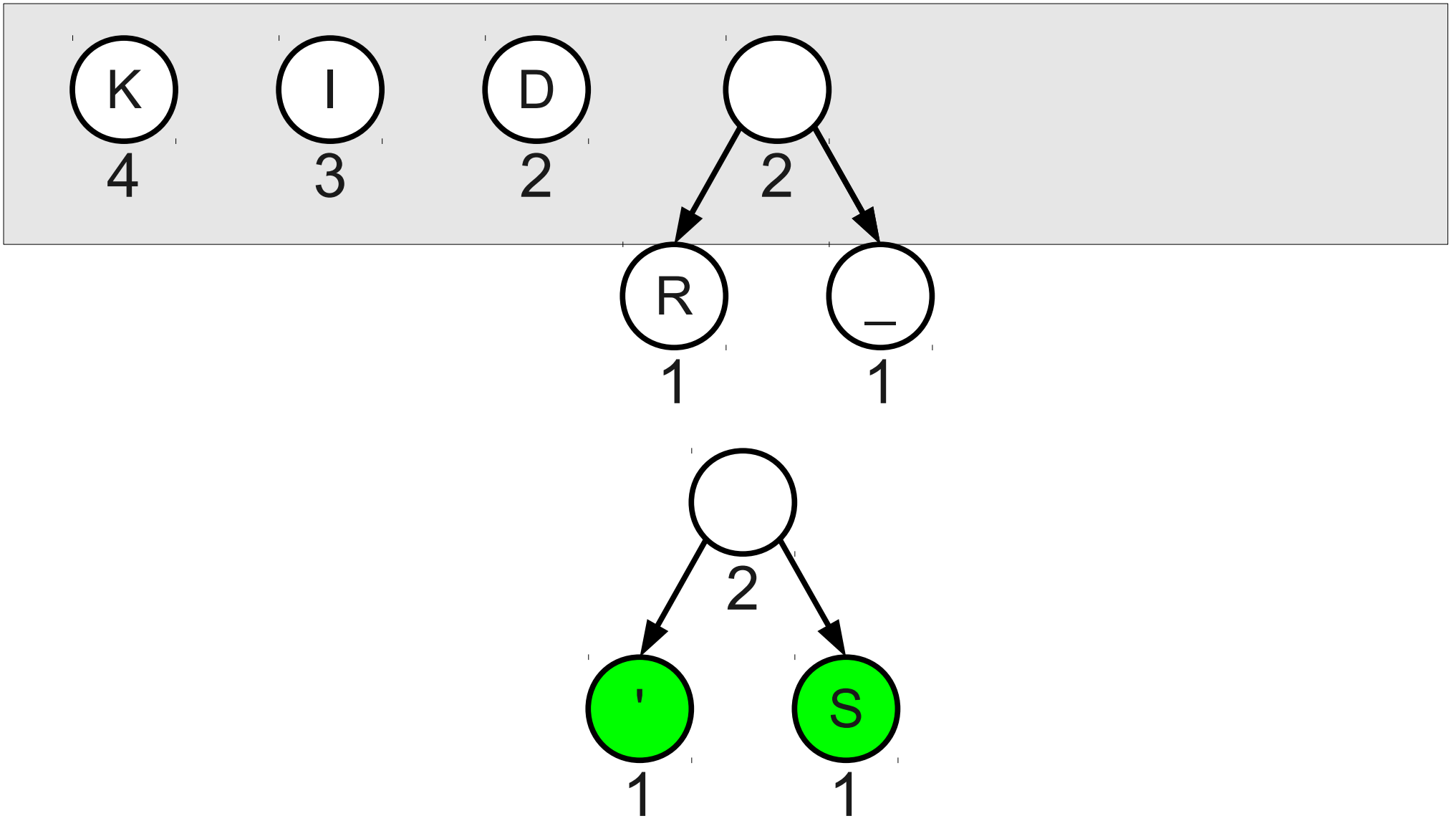


Huffman Coding

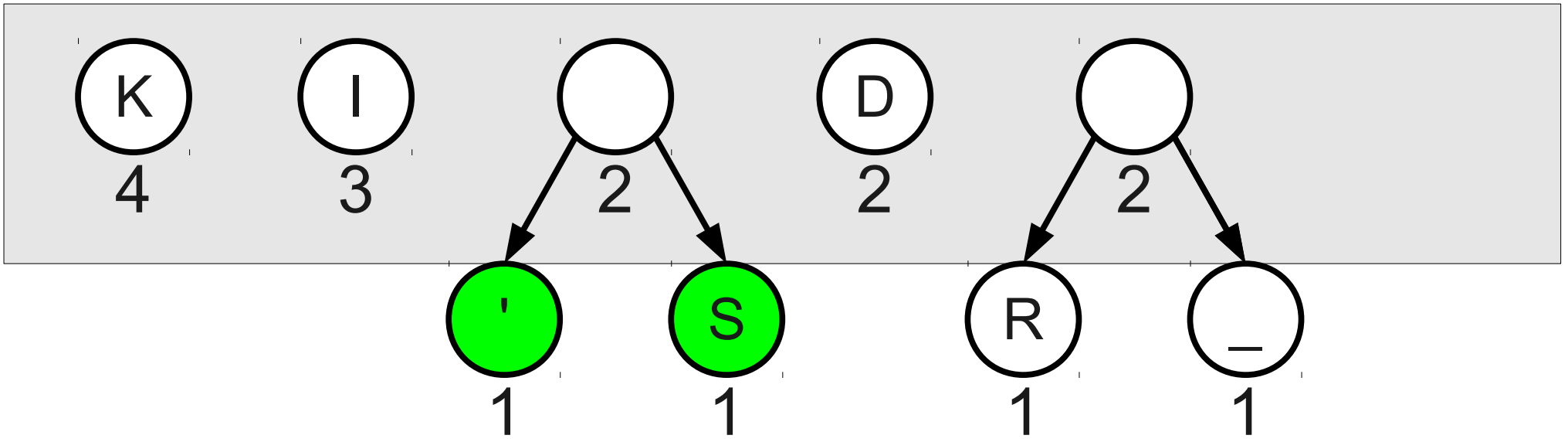




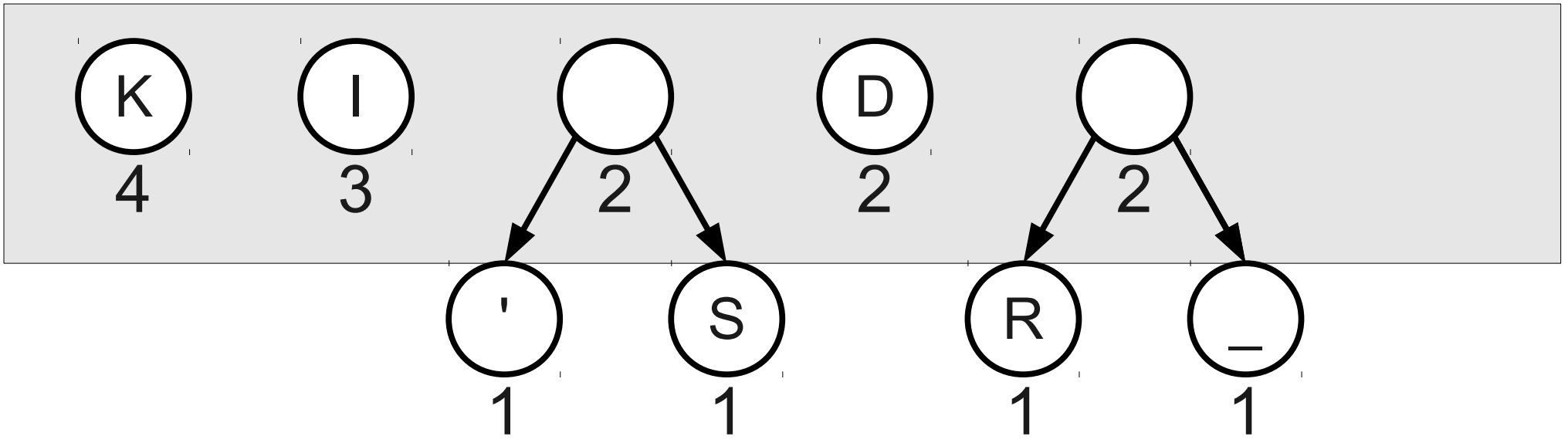
Huffman Coding



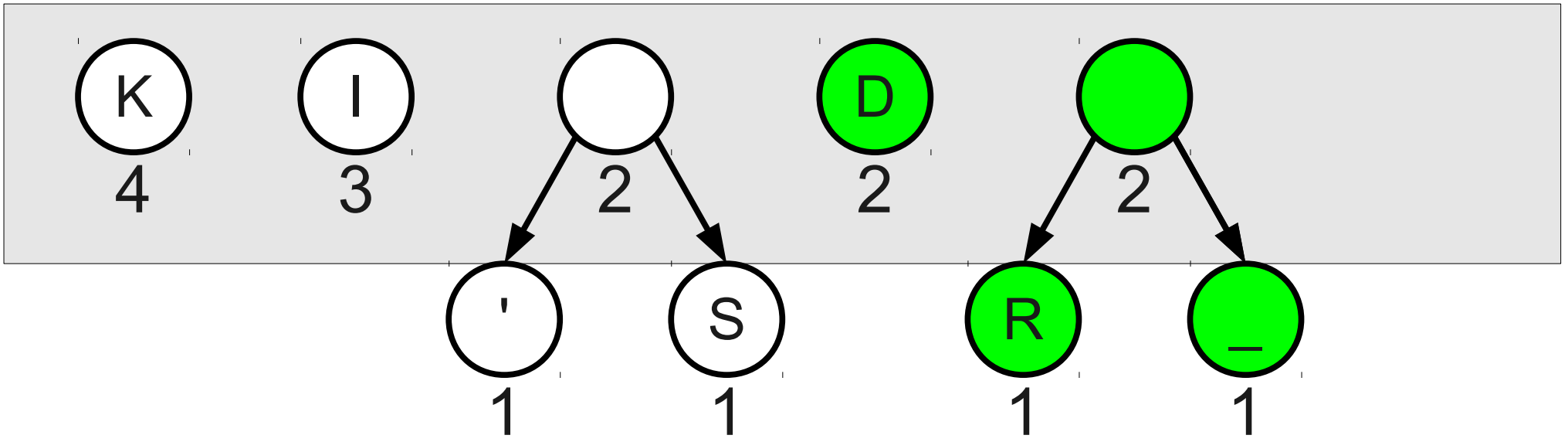
Huffman Coding



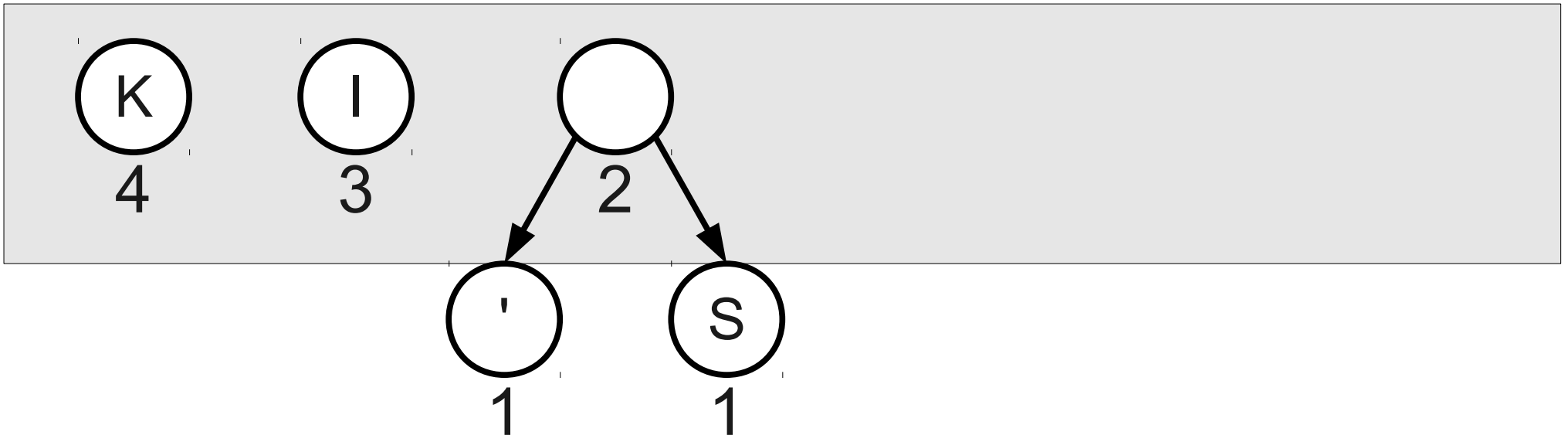
Huffman Coding

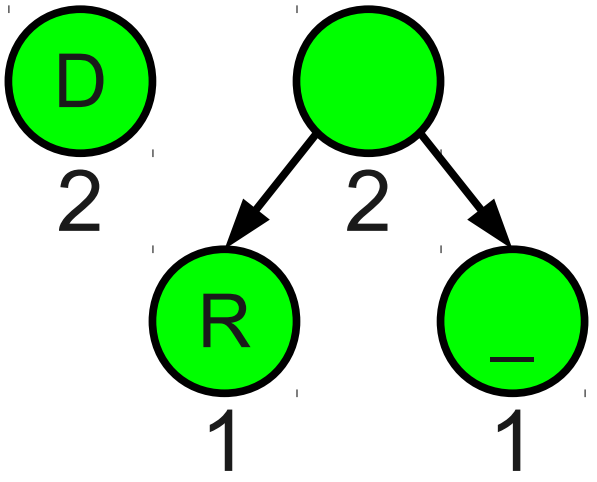


Huffman Coding

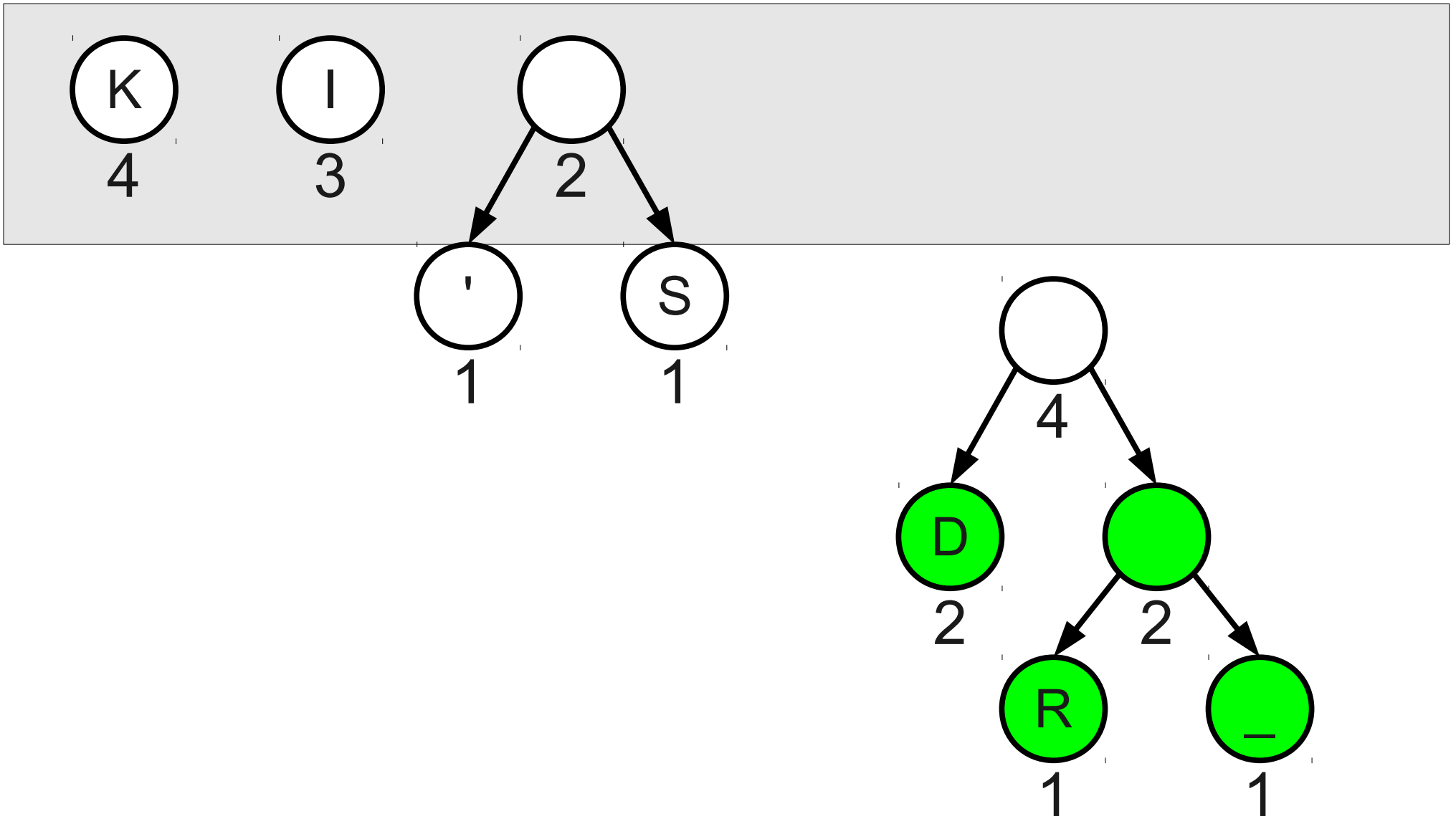


Huffman Coding

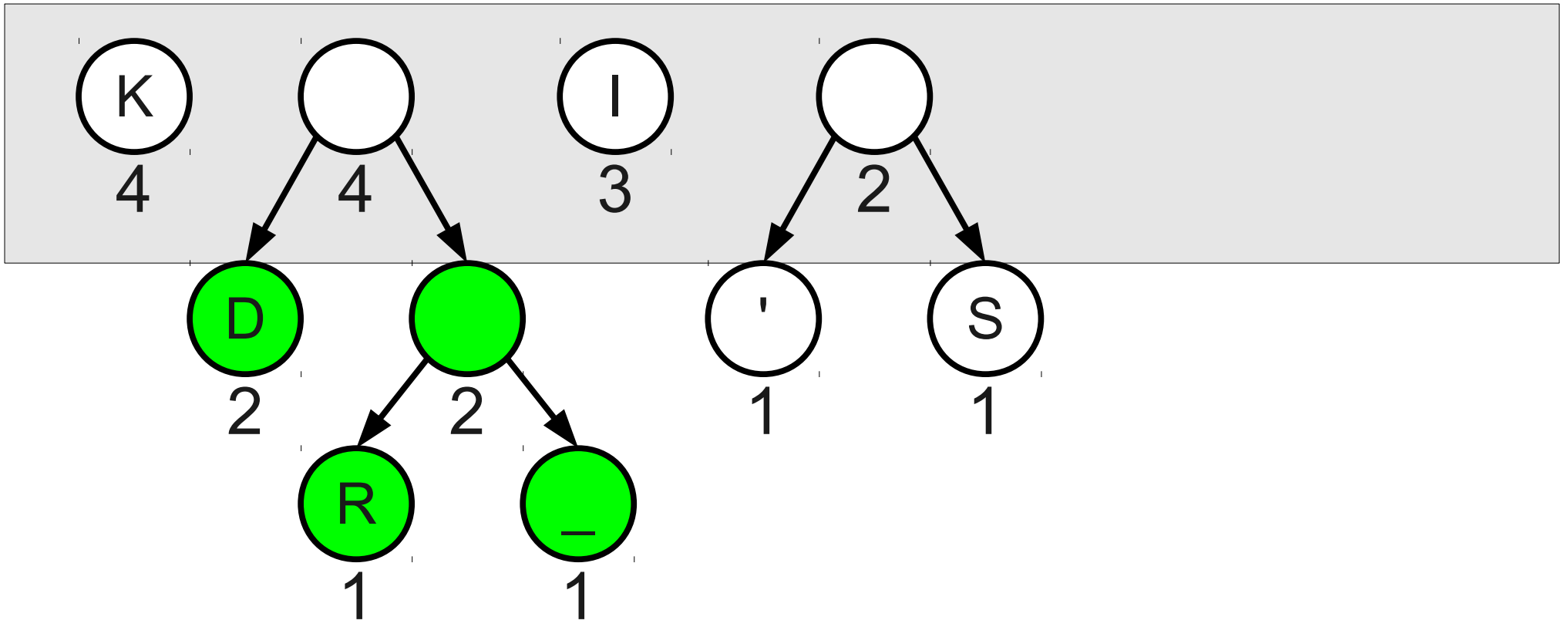




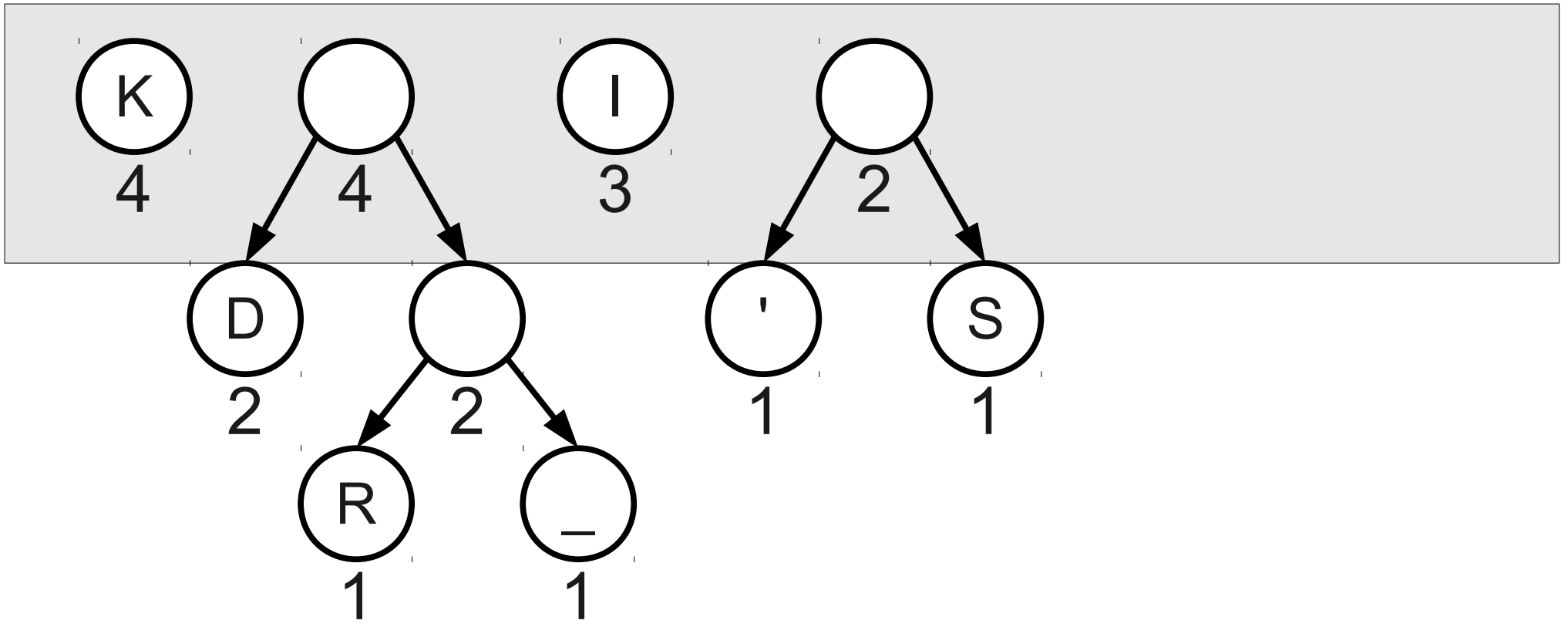
Huffman Coding



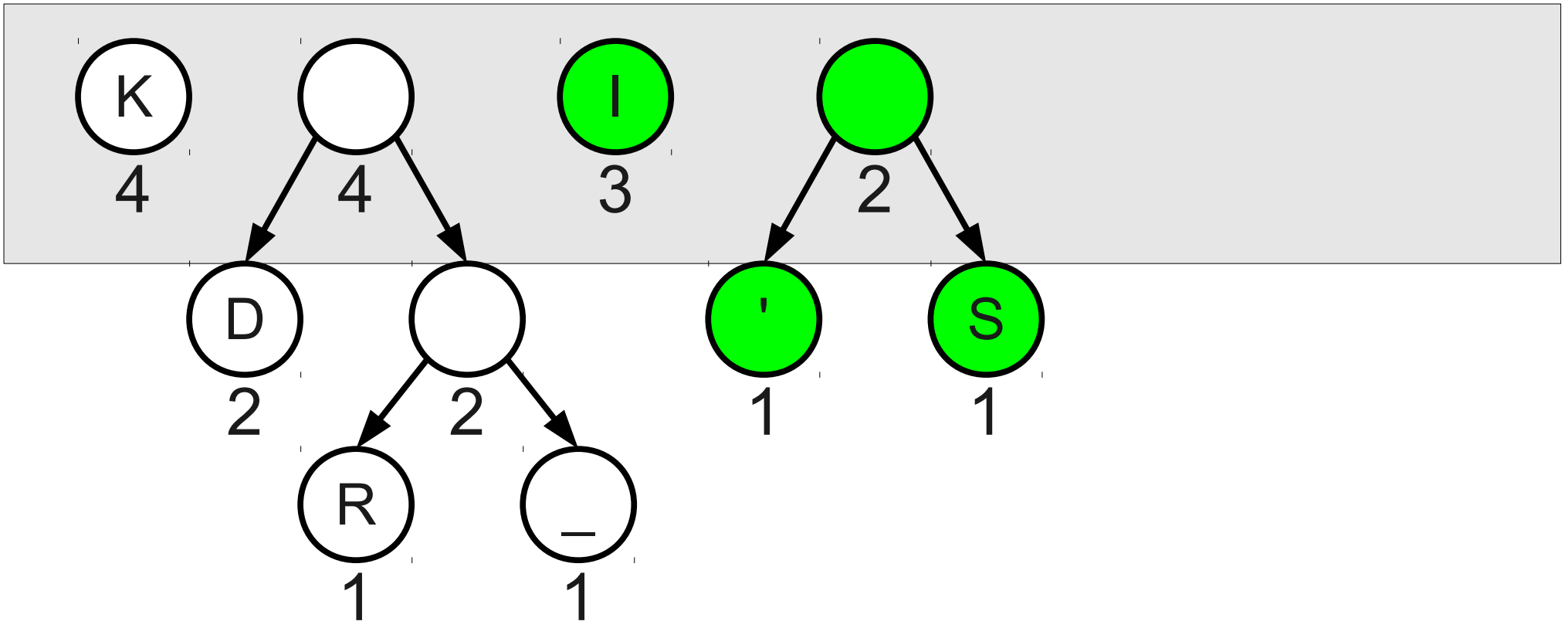
Huffman Coding



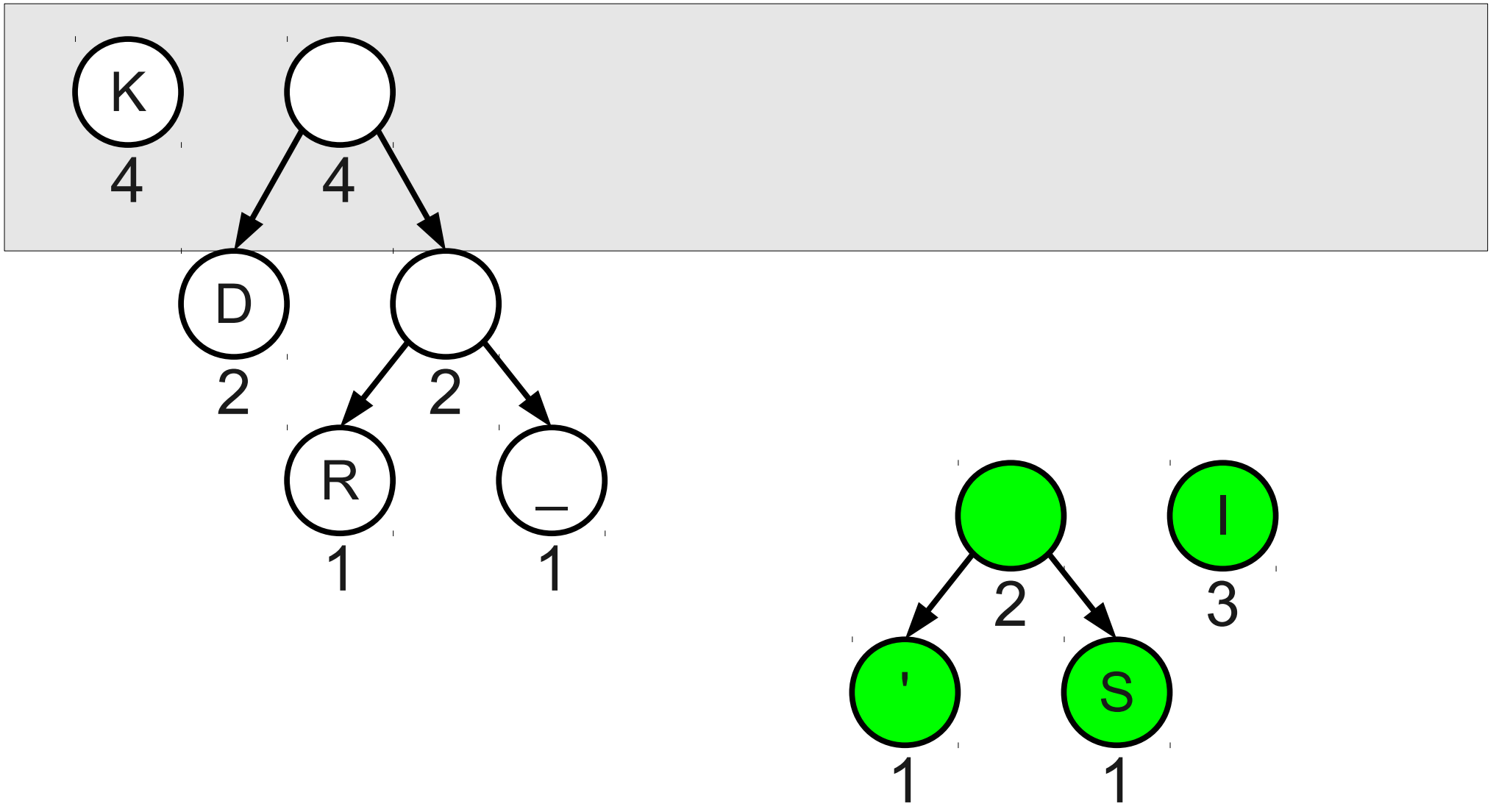
Huffman Coding



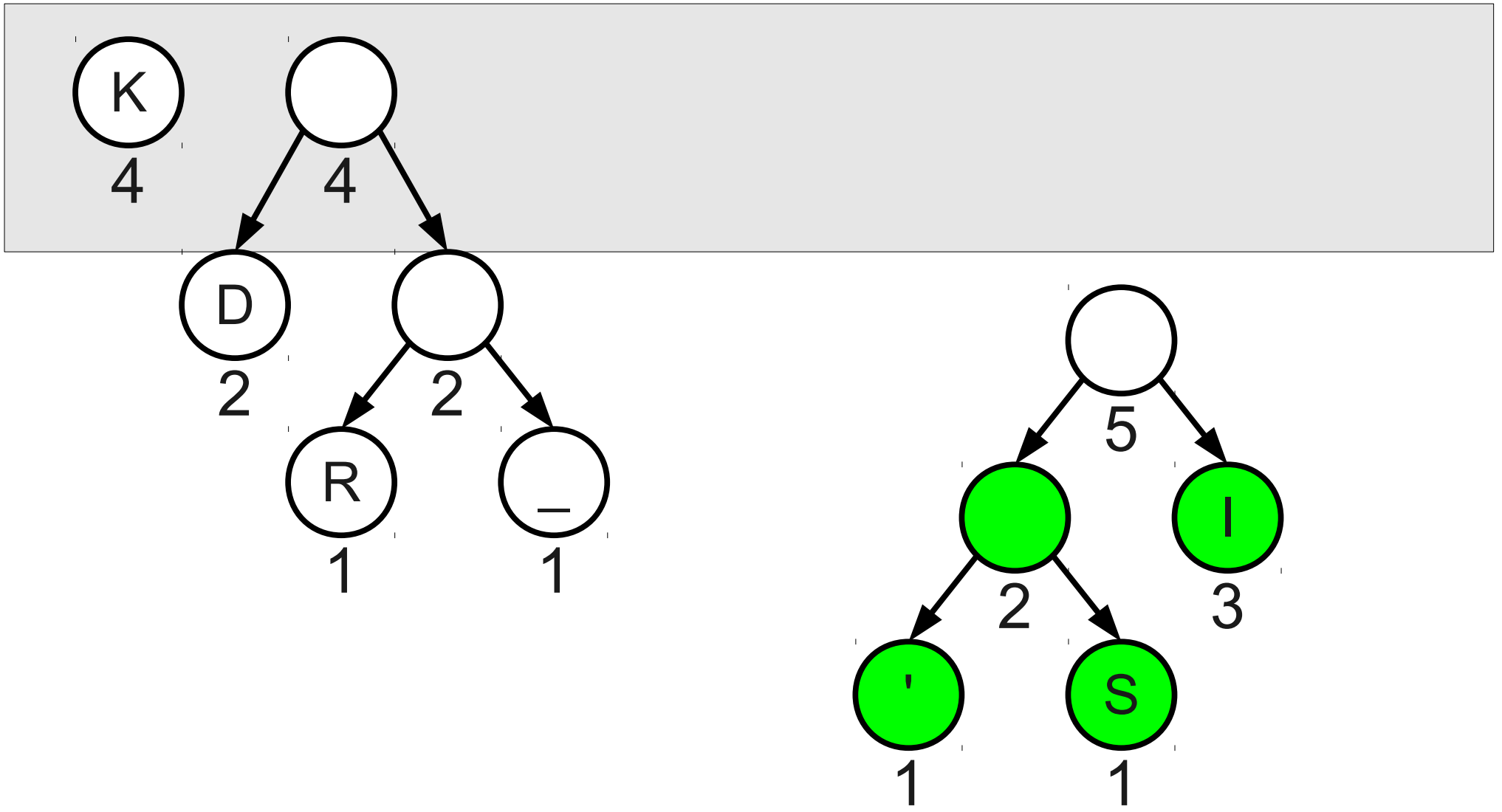
Huffman Coding



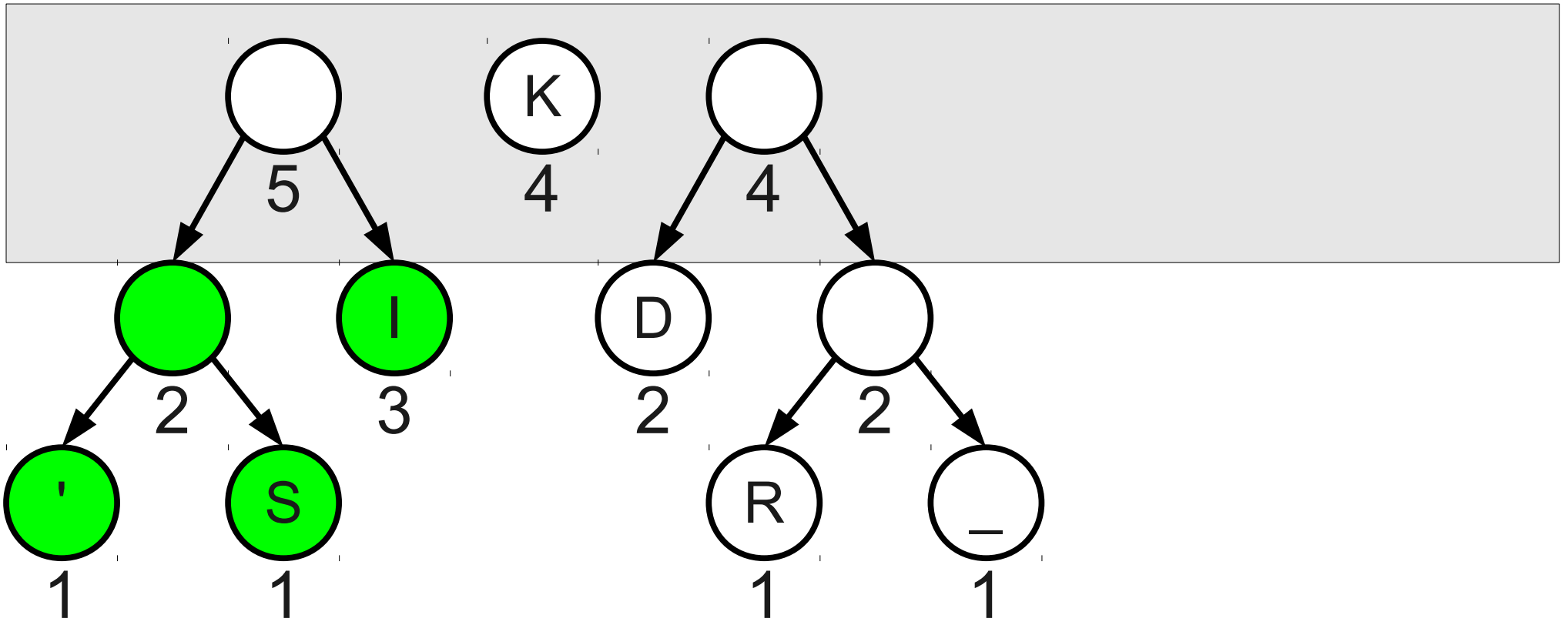
Huffman Coding



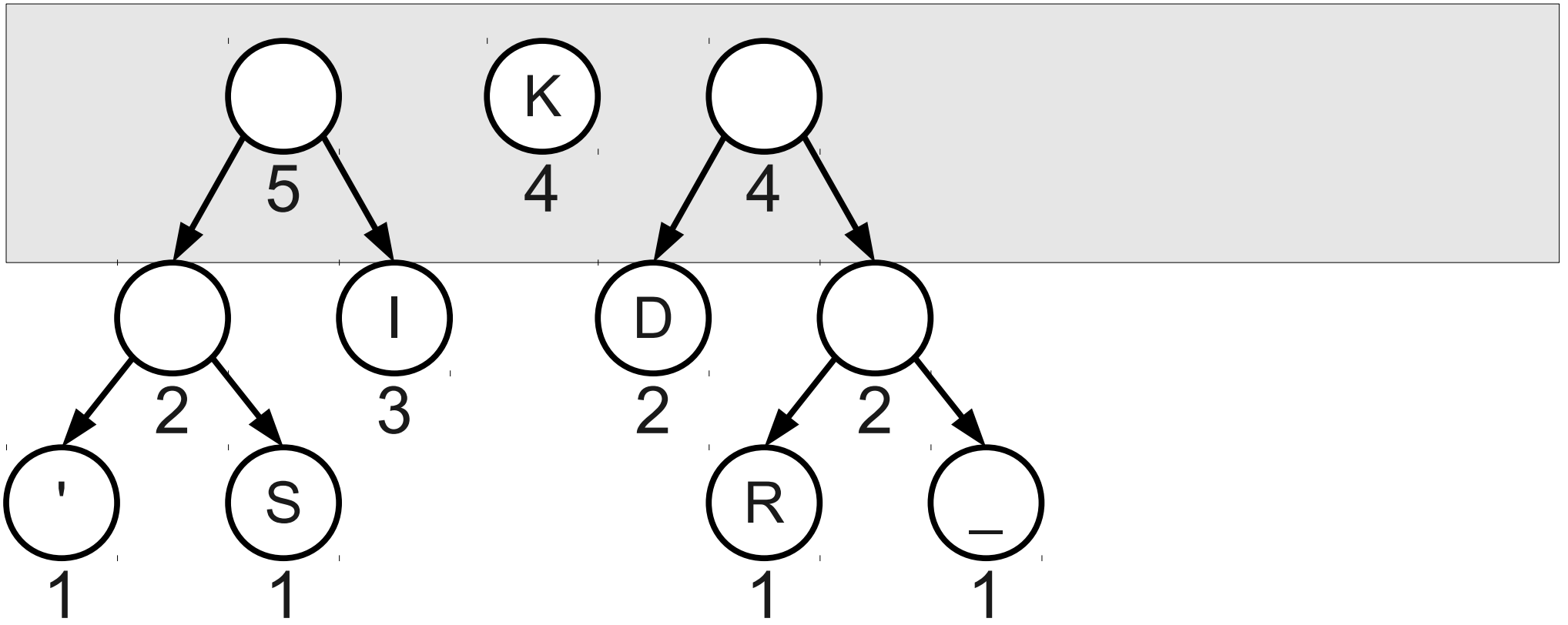
Huffman Coding



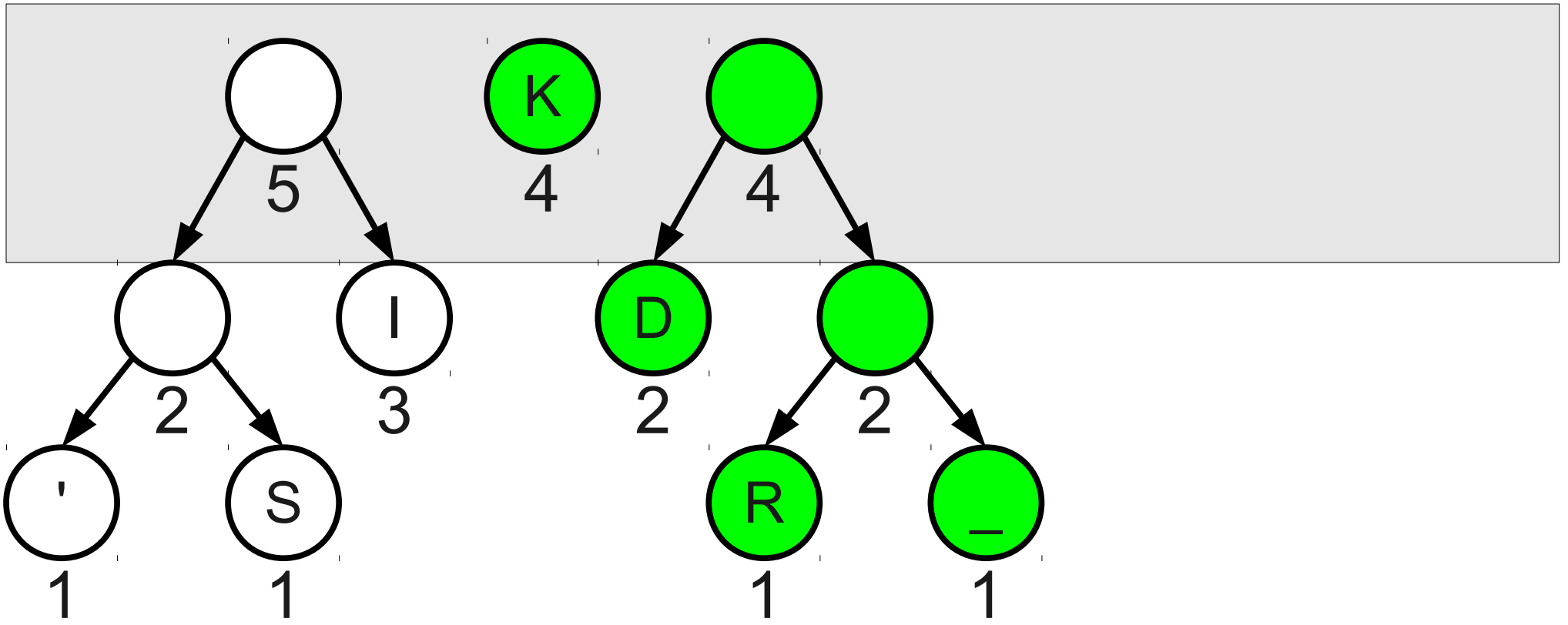
Huffman Coding



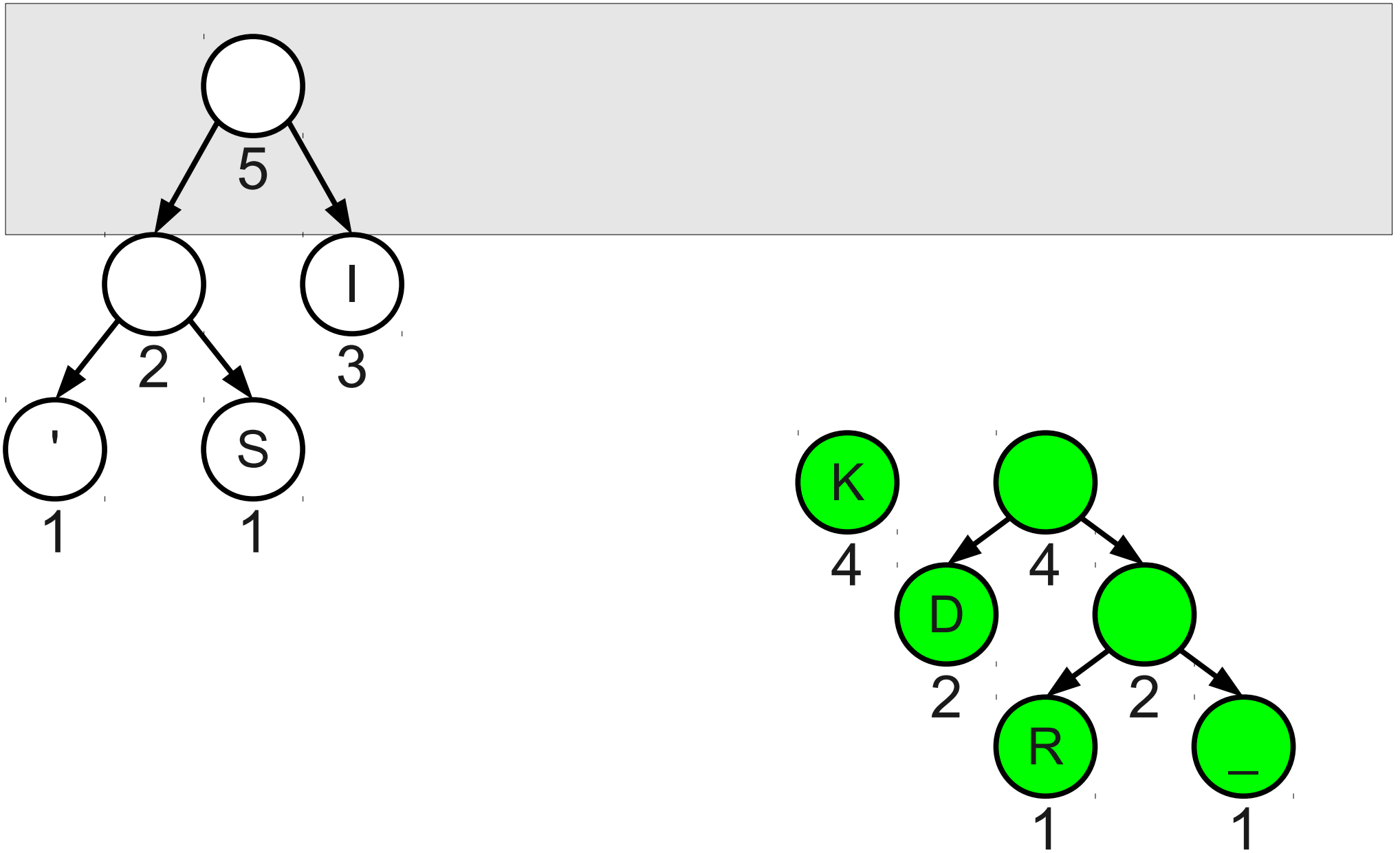
Huffman Coding



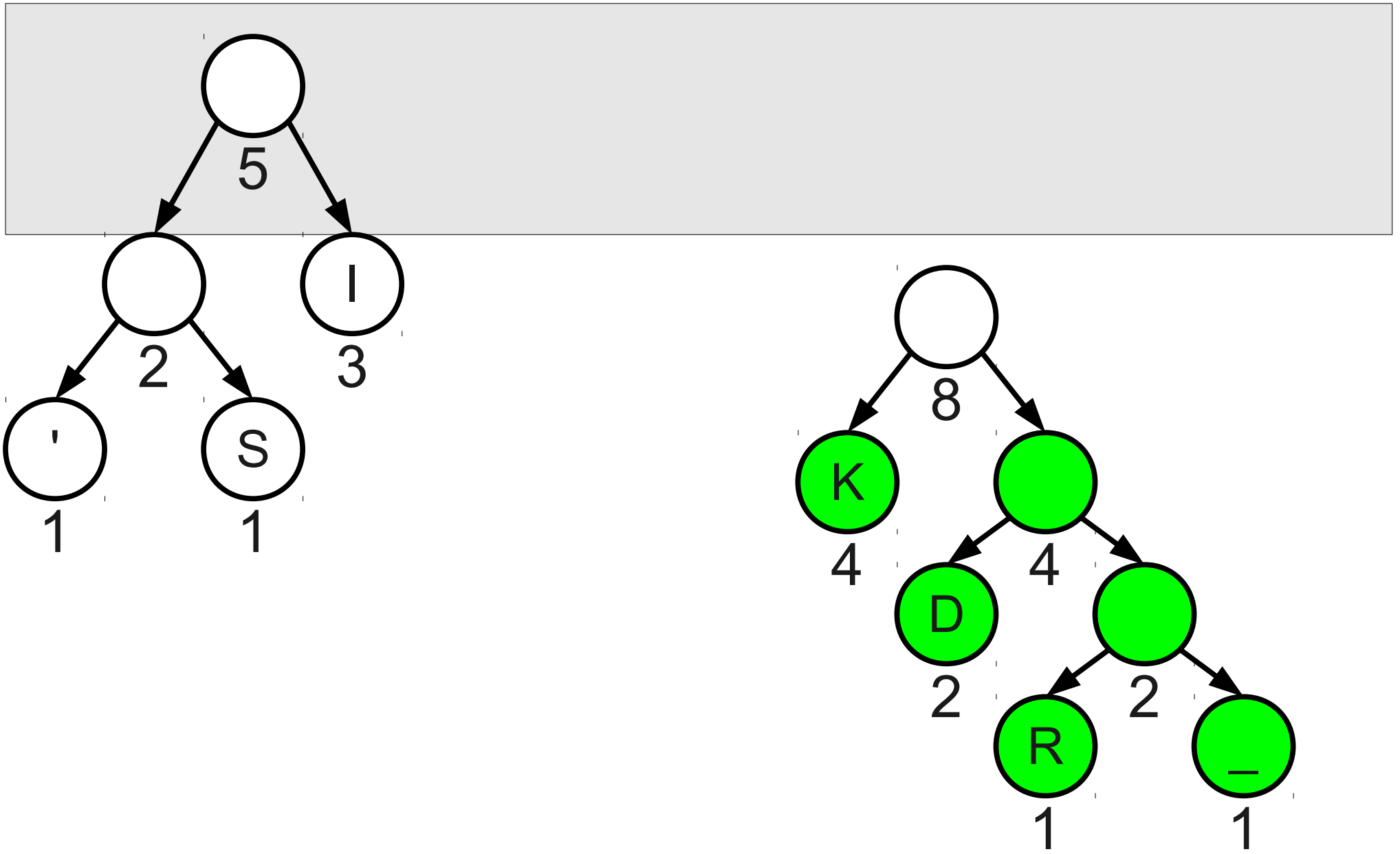
Huffman Coding



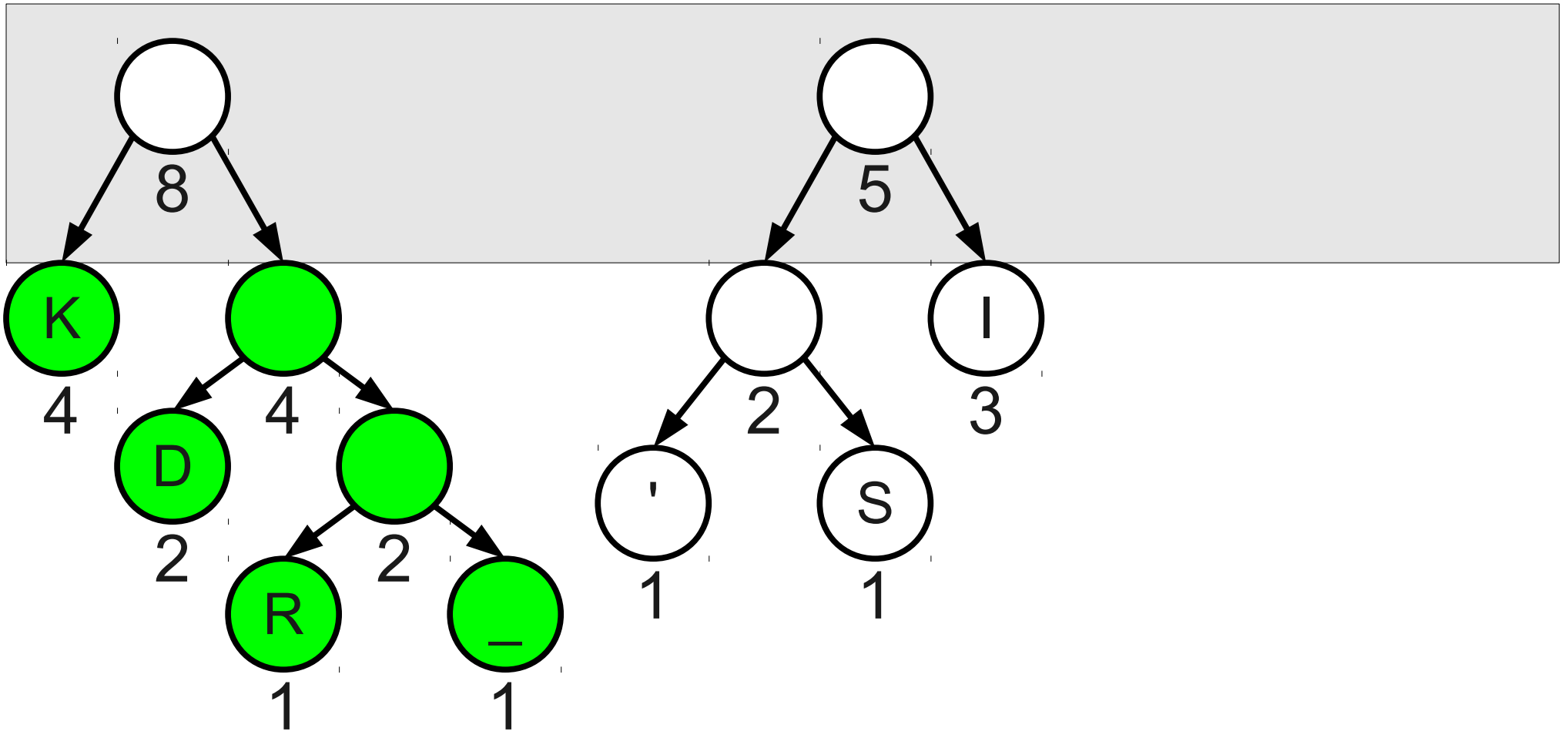
Huffman Coding



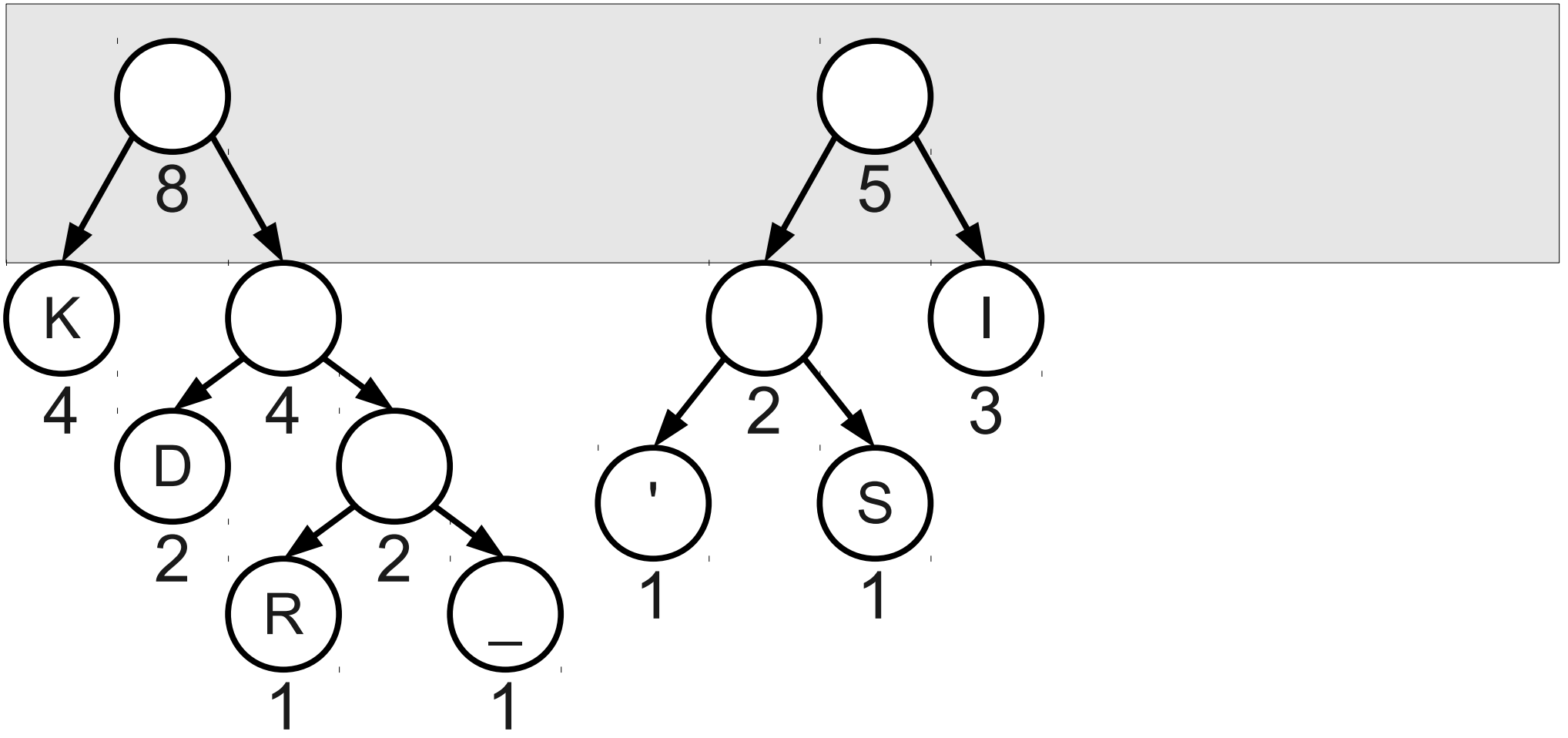
Huffman Coding



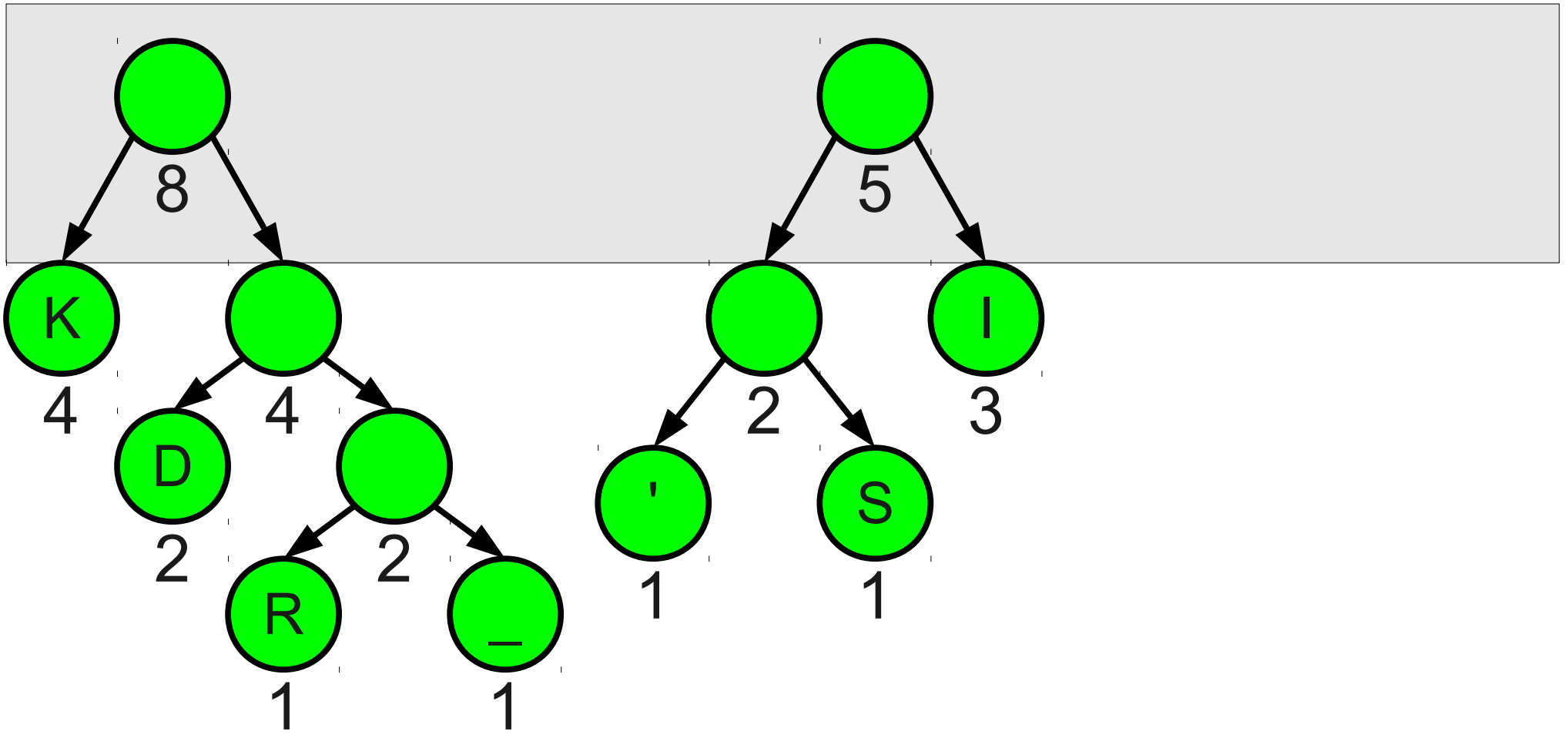
Huffman Coding



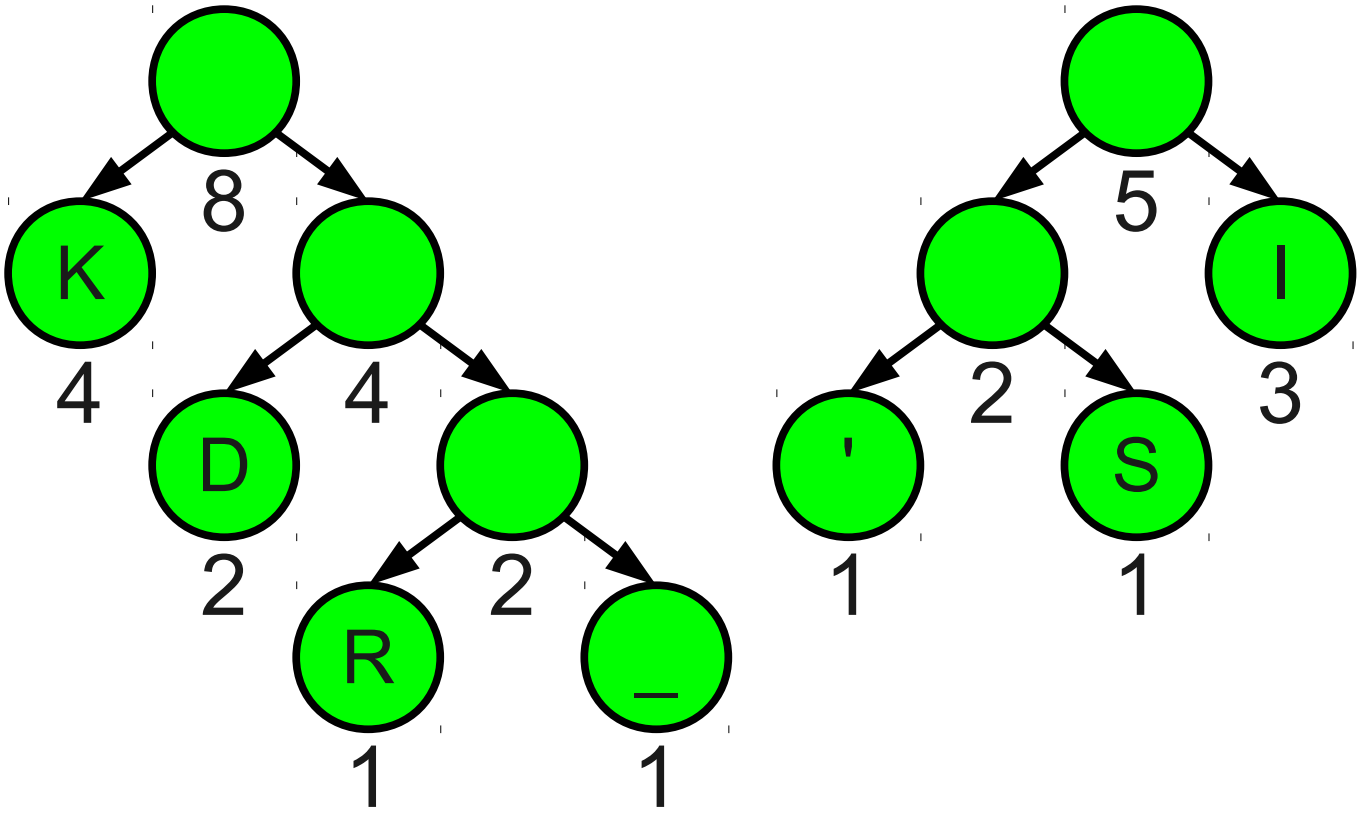
Huffman Coding



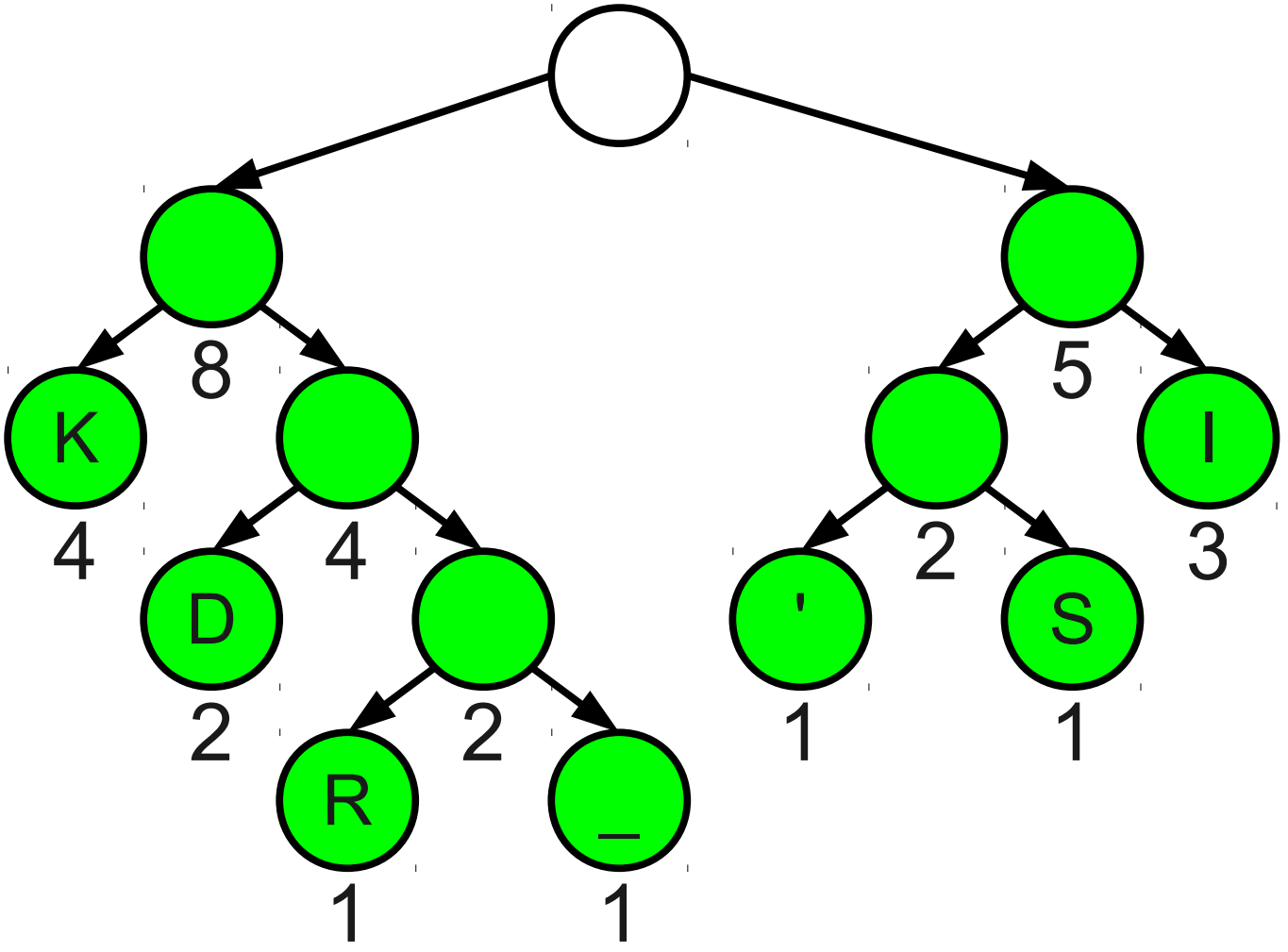
Huffman Coding



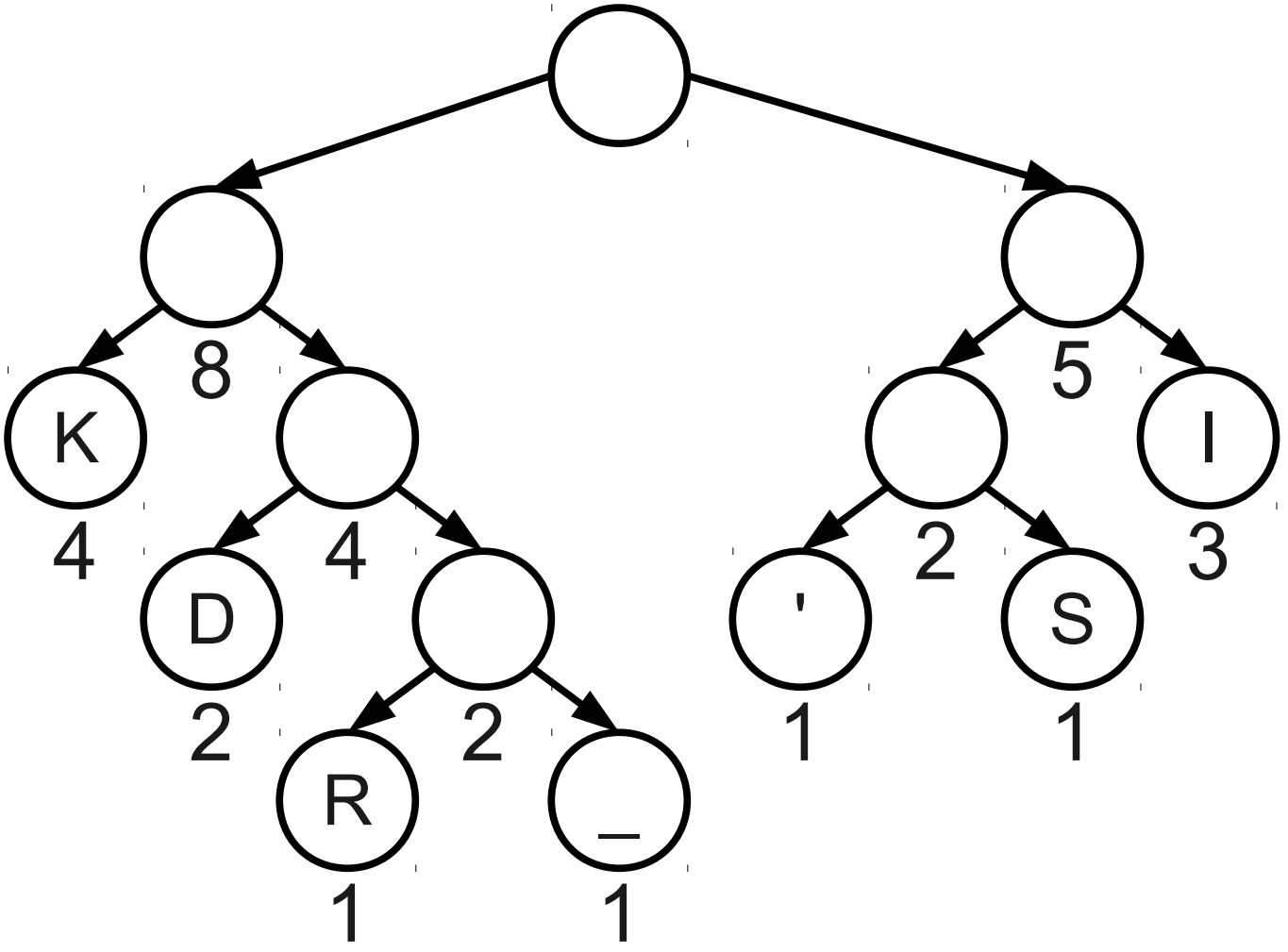
Huffman Coding



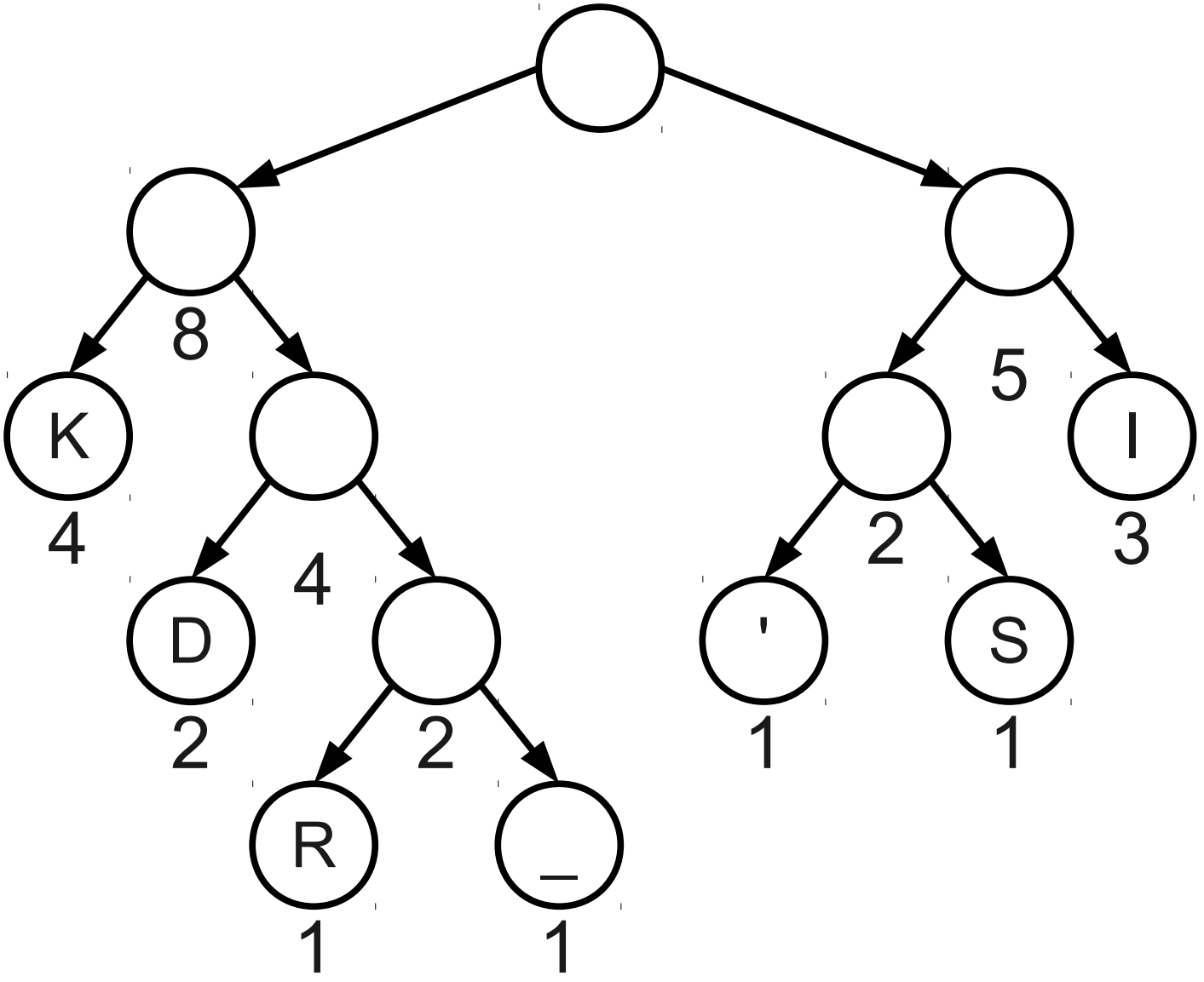
Huffman Coding



Huffman Coding



Huffman Coding



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

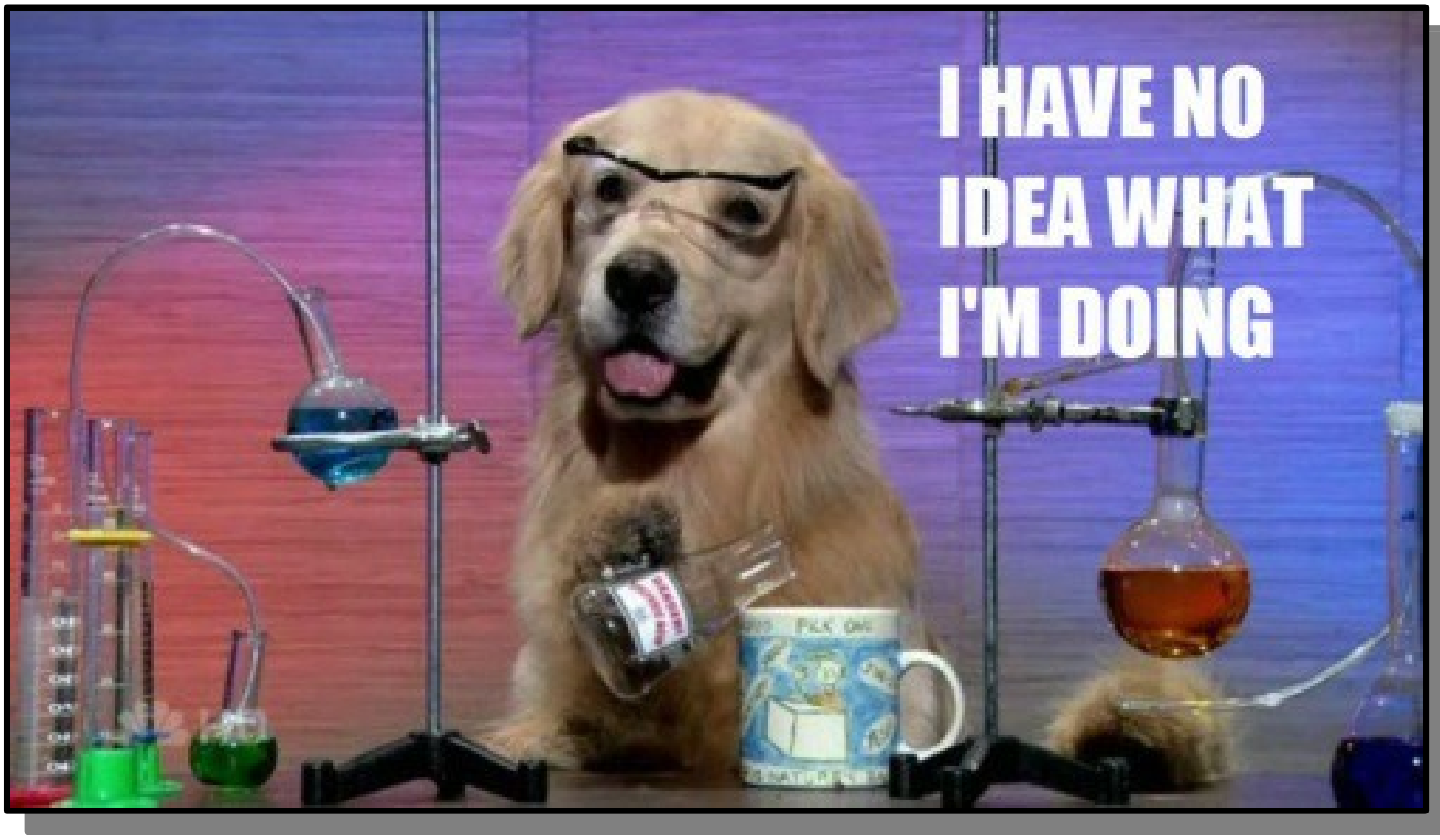
K	10
I	01
D	110
R	1111
'	001
S	000
	1110

```
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

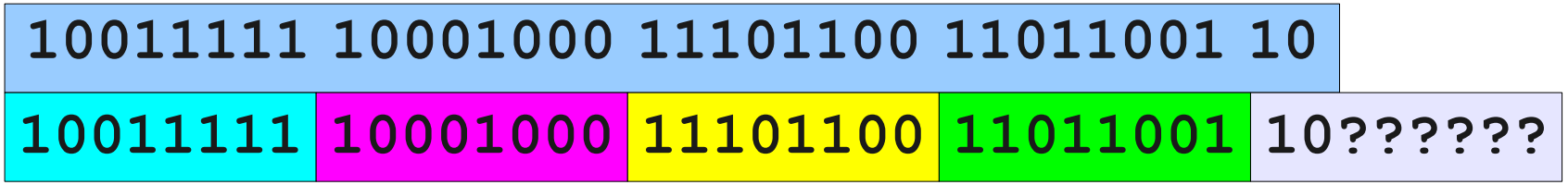
- This might increase the total file size!
- **Theorem**: There is no compression algorithm that can always compress all inputs.

One Last Thing...

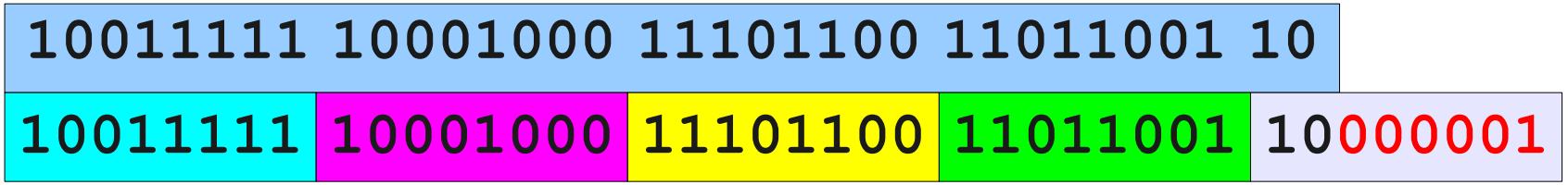
Bitten by Bytes

```
10011111100010001110110011011001l0
```

Bitten by Bytes



Bitten by Bytes



Bitten by Bytes

10011111	10001000	11101100	11011001	10000001

K	10
I	01
D	110
R	1111

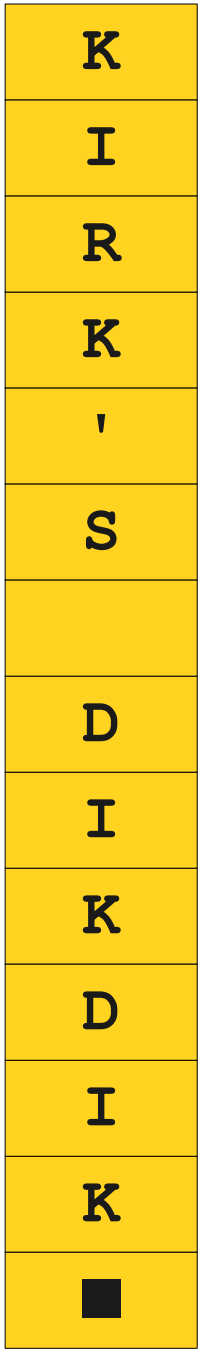
'	001
S	000
	1110

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	'

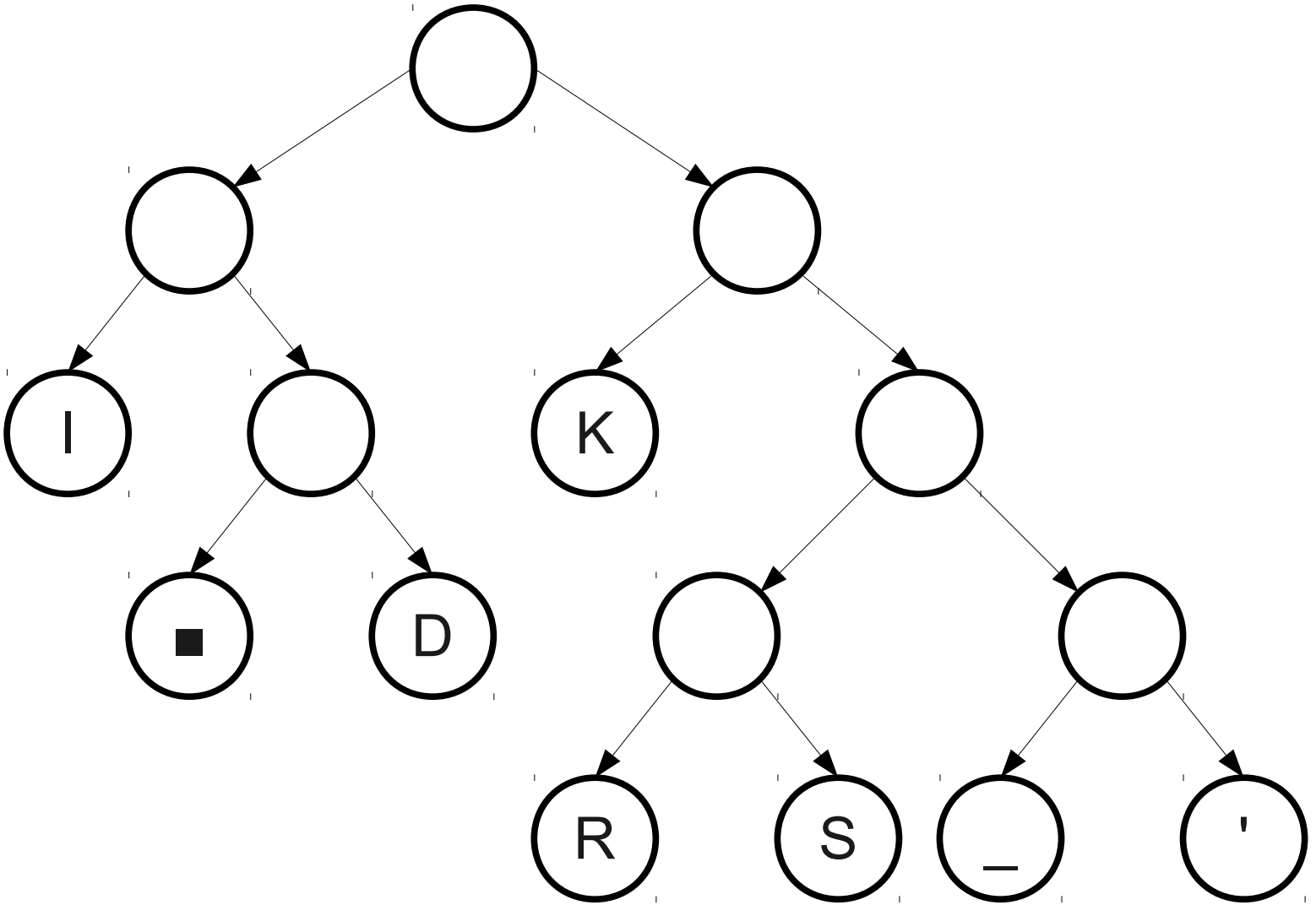
Spare Bits

- 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

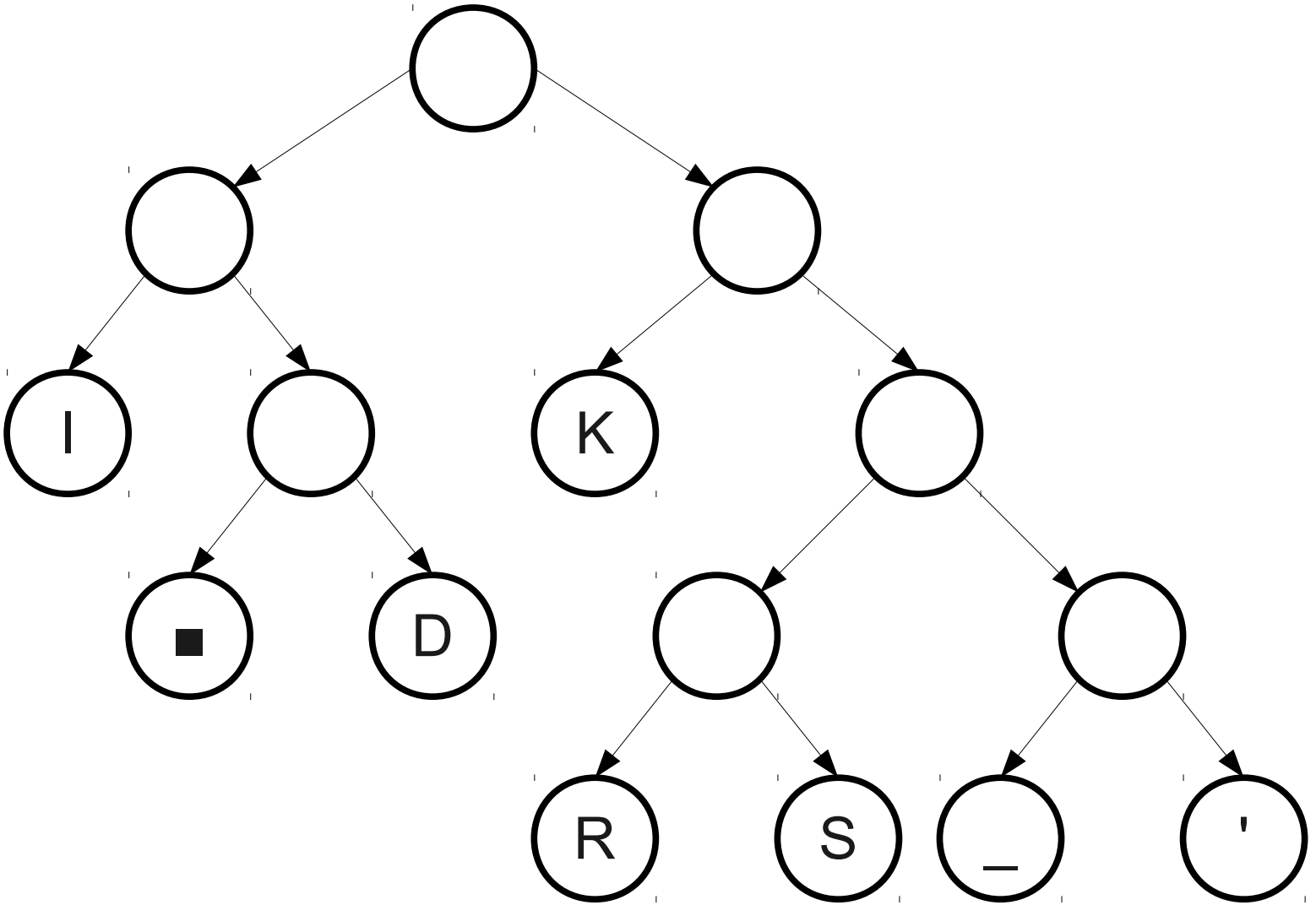


K	4
I	3
D	2
R	1
'	1
S	1
	1
■	1



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

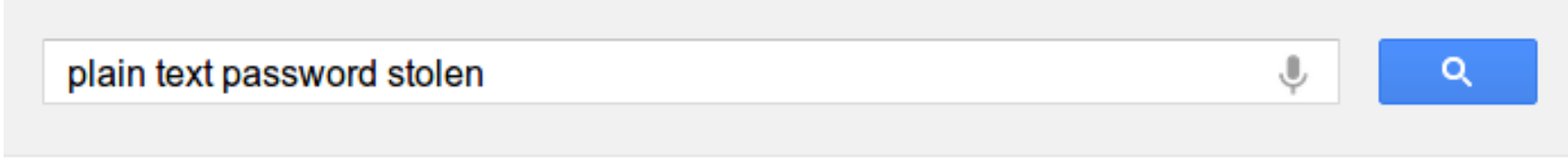
Beyond ASCII

- If you just want to store ASCII text (Latin 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.

Securing Passwords



About 516,000 results (0.32 seconds)

Microsoft Store India Hacked, Passwords Stored in Plain Text
www.tomshardware.com › News › Solutions › Software
Feb 13, 2012 – Last summer\'s PSN breach has meant companies are being watched more closely than ever when it comes to protecting users and securing ...

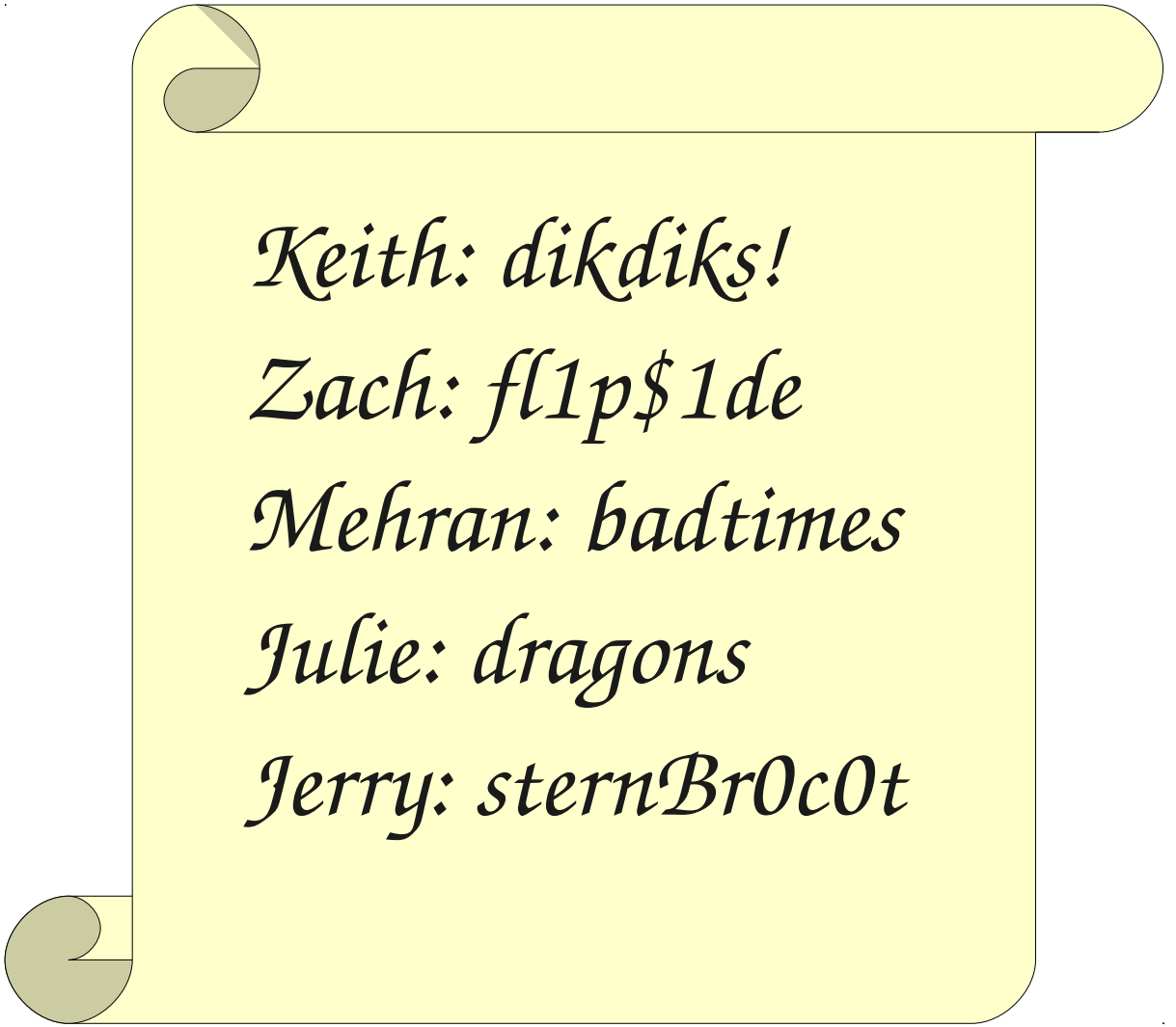
FYI: Newegg stores your password as plaintext (or not?) - Jerry ...
ashercodes.com/fyi-newegg-stores-your-password-as-plaintext
May 24, 2011 – FYI: Newegg stores your **password** as **plaintext** (or not?) .... to you as a result of having your **password stolen** (which would make you paranoid ...

How is "anonymous" getting ahold of all these plain-text passwords
security.stackexchange.com/.../how-is-anonymous-getting-ahold-of-a...
6 answers - Aug 17, 2011
The worst part of having one site hacked and account info (with **plaintext passwords**) **stolen** means there is a security risk for other sites, ...

PLEASE Stop Doing Passwords Wrong! - TechRant
techrant.co.uk/2012/02/please-stop-doing-passwords-wrong-2/
Feb [illegible] ... RoE stores **password** in **plain text** ... it is an unprofessional outfit ... or unhappy employee or your database **stolen**, and your **passwords** ...

Bitscalper passwords have been leaked
https://bitcointalk.org/index.php?topic=63659.60
2 posts - 2 authors - Feb 16
If **plaintext passwords** were **stolen**, either the attacker modified the code of the website to prevent pre-transmisssion hashing, or passwords ...

How Not to Store Passwords



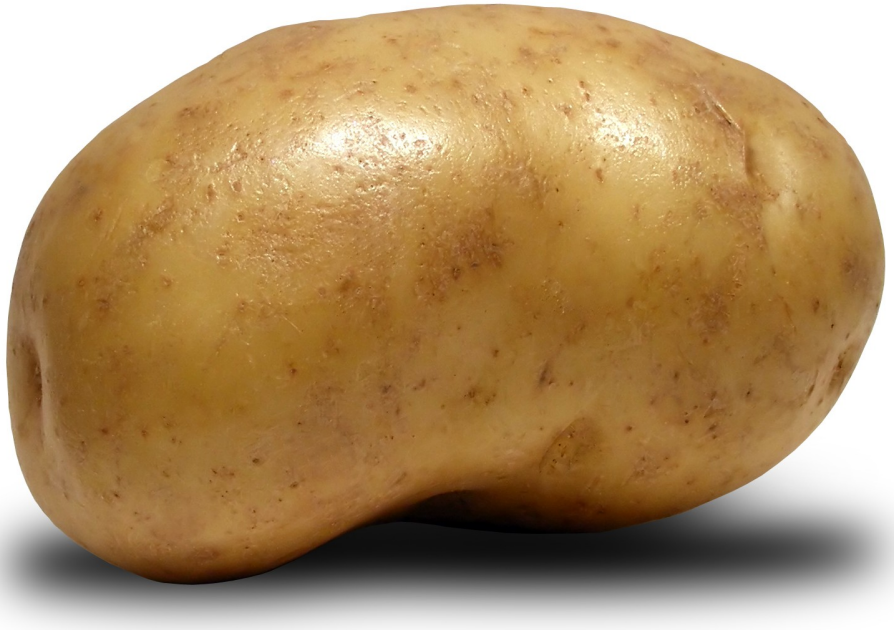
Never store passwords in plain text!

What We Need

- We need a system where
  - The computer can confirm that you are who you claim to be, but
  - Someone stealing files off the computer can't log in as you.
- How can we do this?
- Let's return to hashing...

Good Hash Functions

- A good hash function typically will scramble all of the bits of the input together in a way that appears totally random.
- Hence the name “hash function.”



Good Hash Functions

- Good hash functions are similar to completely random functions: there should be no (apparent) patterns.
  - Making a tiny change to the input should make a huge change in the output.
  - Looking at the output of the hash function, it is hard to figure out any possible input.
  - Hard to find two inputs that produce a hash collision.
- There are many uses of these functions beyond hash tables!

Good Hash Functions

- Most hash functions take in an object (usually a string) that can be arbitrarily large, then produce a sequence of bits.
  - Can think of those bits as a string or as a number.
- Many hash functions exist that produce 256 or 512 bits as output.
- This means that there are 2^{256} or 2^{512} possible outputs from one of these hash functions.
- Assuming the hash function is “mostly random,” the odds of finding two inputs that produce the same output is essentially 0.

A Better Way to Store Passwords

Keith: 1679a56fdc0d471aa8aa2dd614dfe7be

Zach: 12461f0d665f443a9ec346ba6c146d0a

Mehran: 25c219e080ed455cbbf0ec8a17a773a8

Julie: 85ad6c9fb97a4425891b1e19dcfa9eb1

Jerry: e9f976b558fe424baf544d60c17ce6a9

Securing a Site

- Never directly store passwords; instead store hashes of the passwords.
- To check if the password is correct:
  - Compute the hash of the password.
  - See if it matches the stored hash.
  - If so, allow access.
  - If not, deny access.
- Odds of wrong logins succeeding extremely small.

More on Hashing

- Computer security is an extremely important topic from both an implementation and theory perspective:
  - How do you design systems that are resistant to attack?
  - How do you build secure encryption and hash systems?
- Take CS155 or CS255 for more information.

Next Time

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