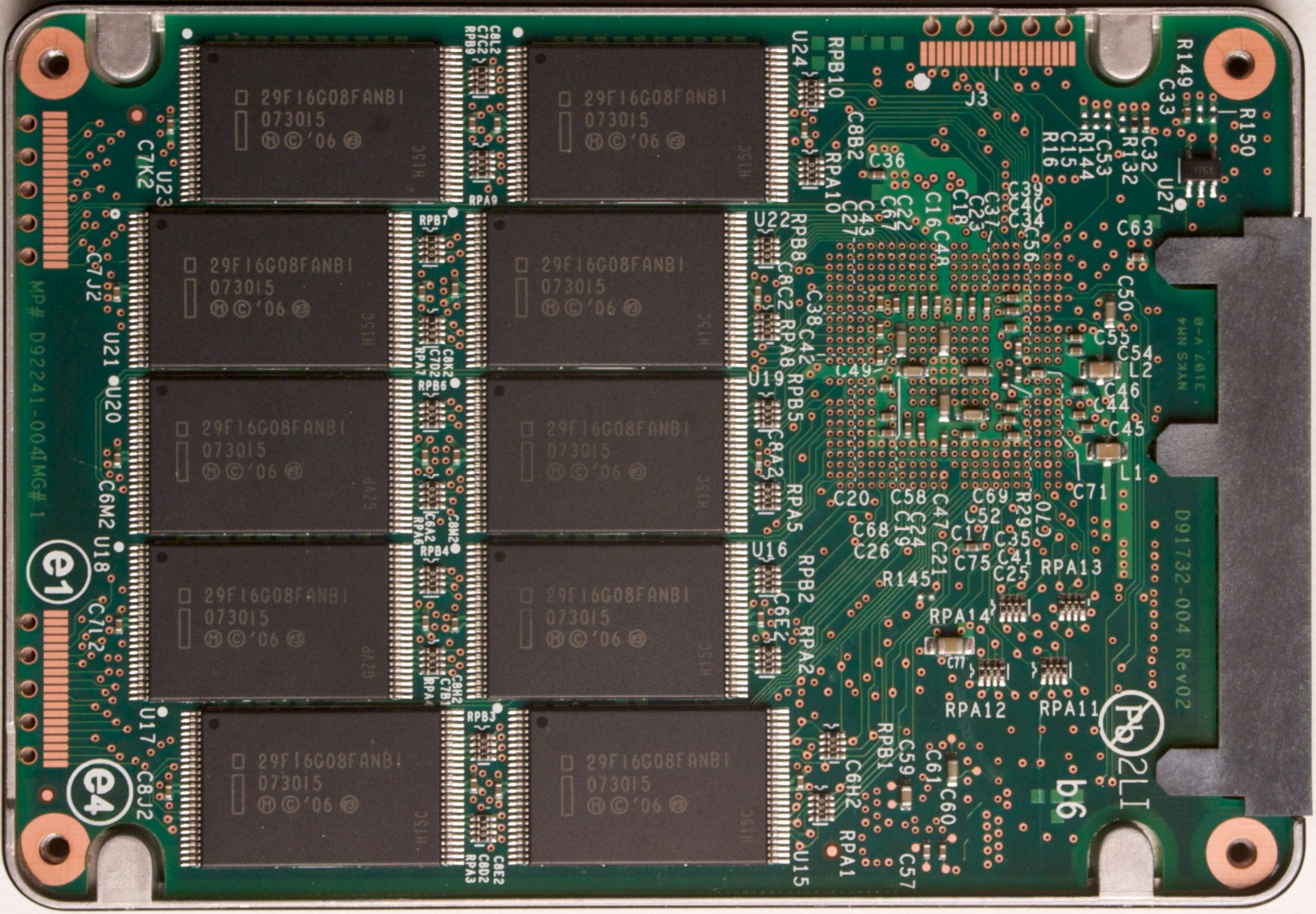
Beyond Data Structures

Huffman Encoding


29F16G08FANBI
073015
©'06
MP# D92241-004MG# 1
D91732-004 Rev02
J3
U27
L1
L2
e1
e4
Pb
b6

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, …

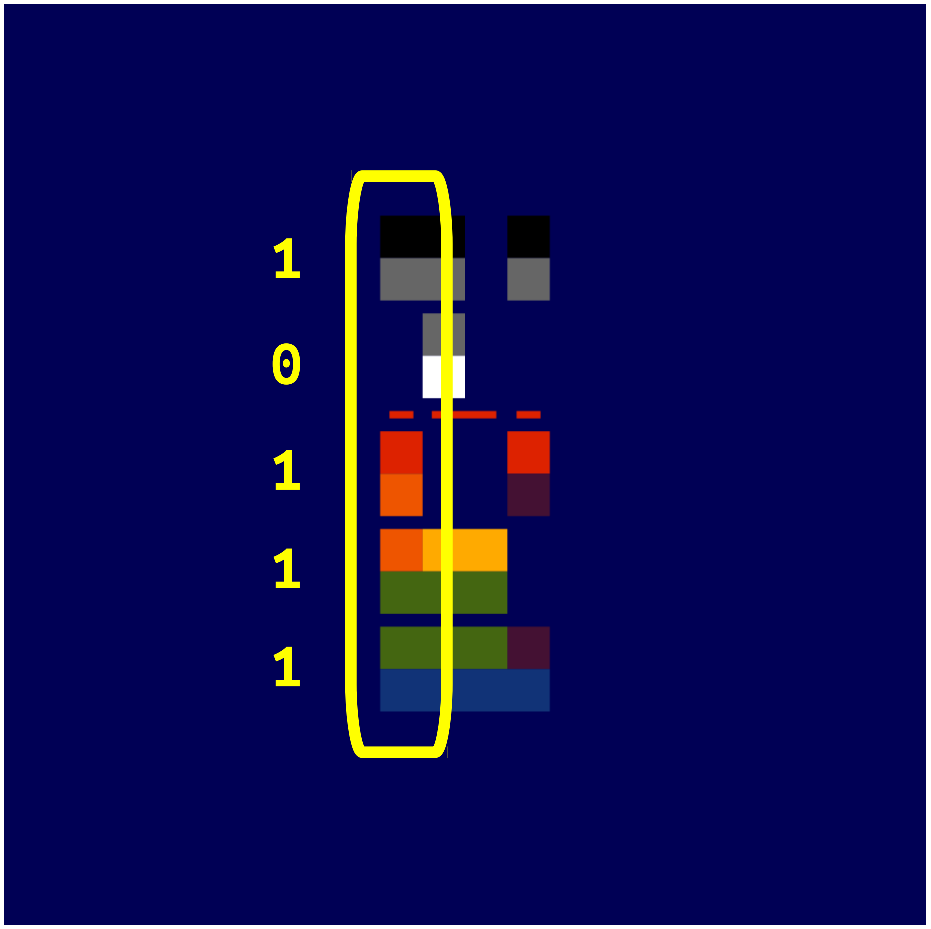
- There are $2^8 = 256$ different bytes.
  - ***Great recursion practice:*** Write a function to list all of them!

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
```

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:

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
```

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

```
1001111111000100011101100110110011 0
```

Prefix Codes

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

```
1001111110001000111011001101100110
```

10
K

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

```
10011111100010001110110011101100110
```

10	01	1111
K	I	R

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

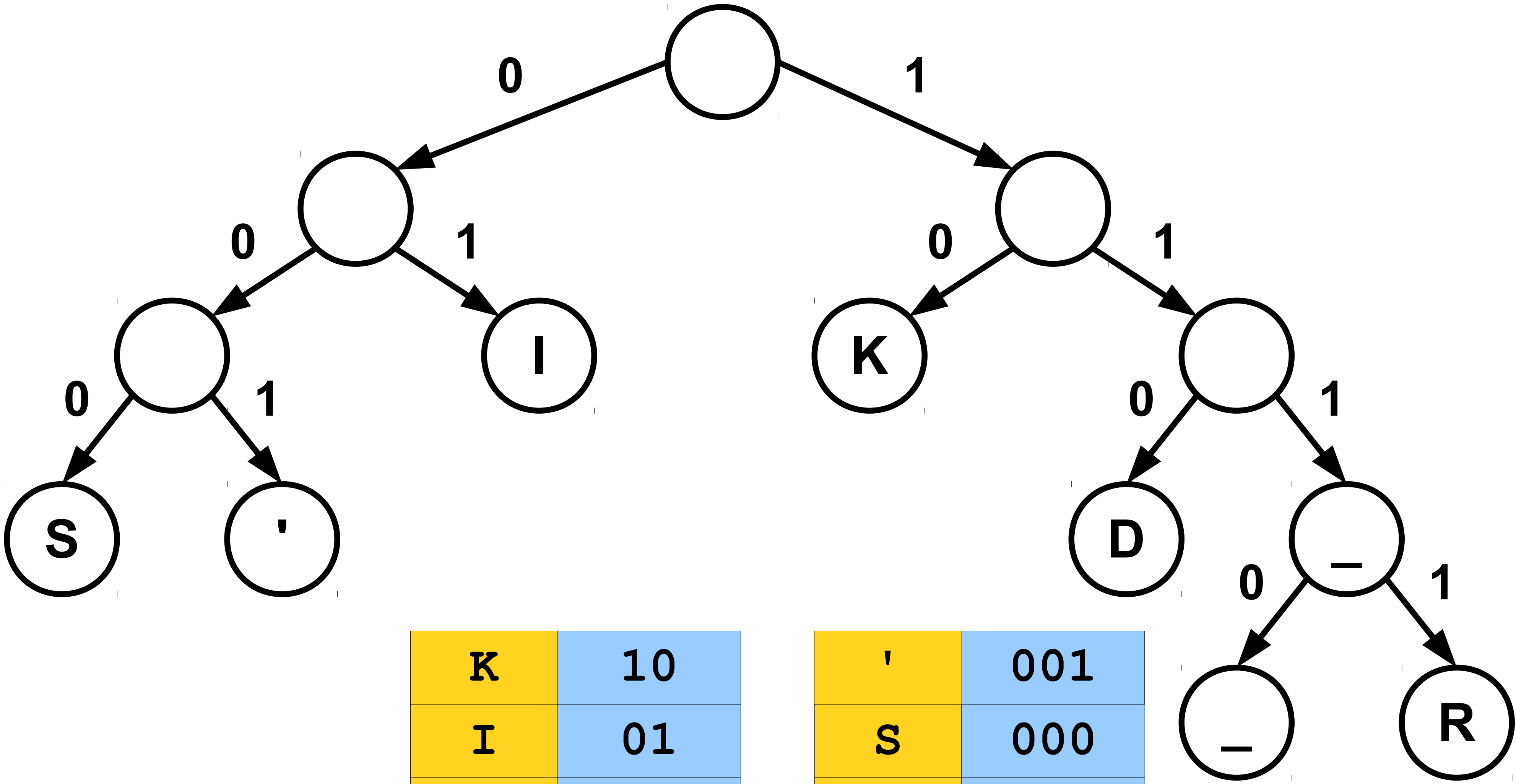
```
1001111110001000111011001101100110
```

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

```
11111101111101111011111011101101001110111111001111101111110
```

How do you find a “good” prefix code?

The Key Idea



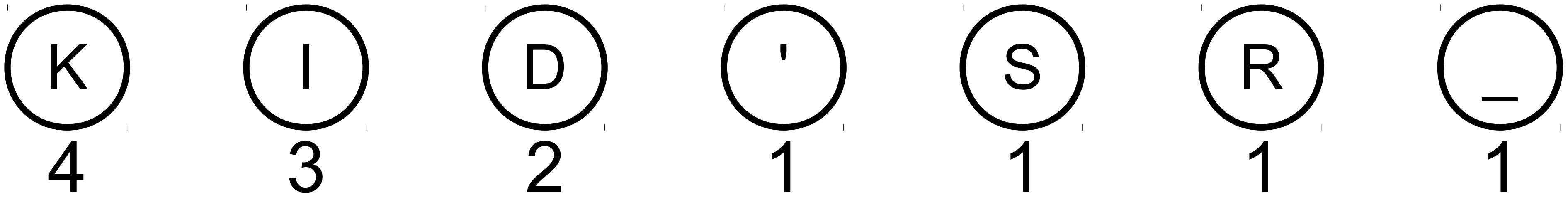
K	10
I	01
D	110
R	1111

'	001
S	000
	1110

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

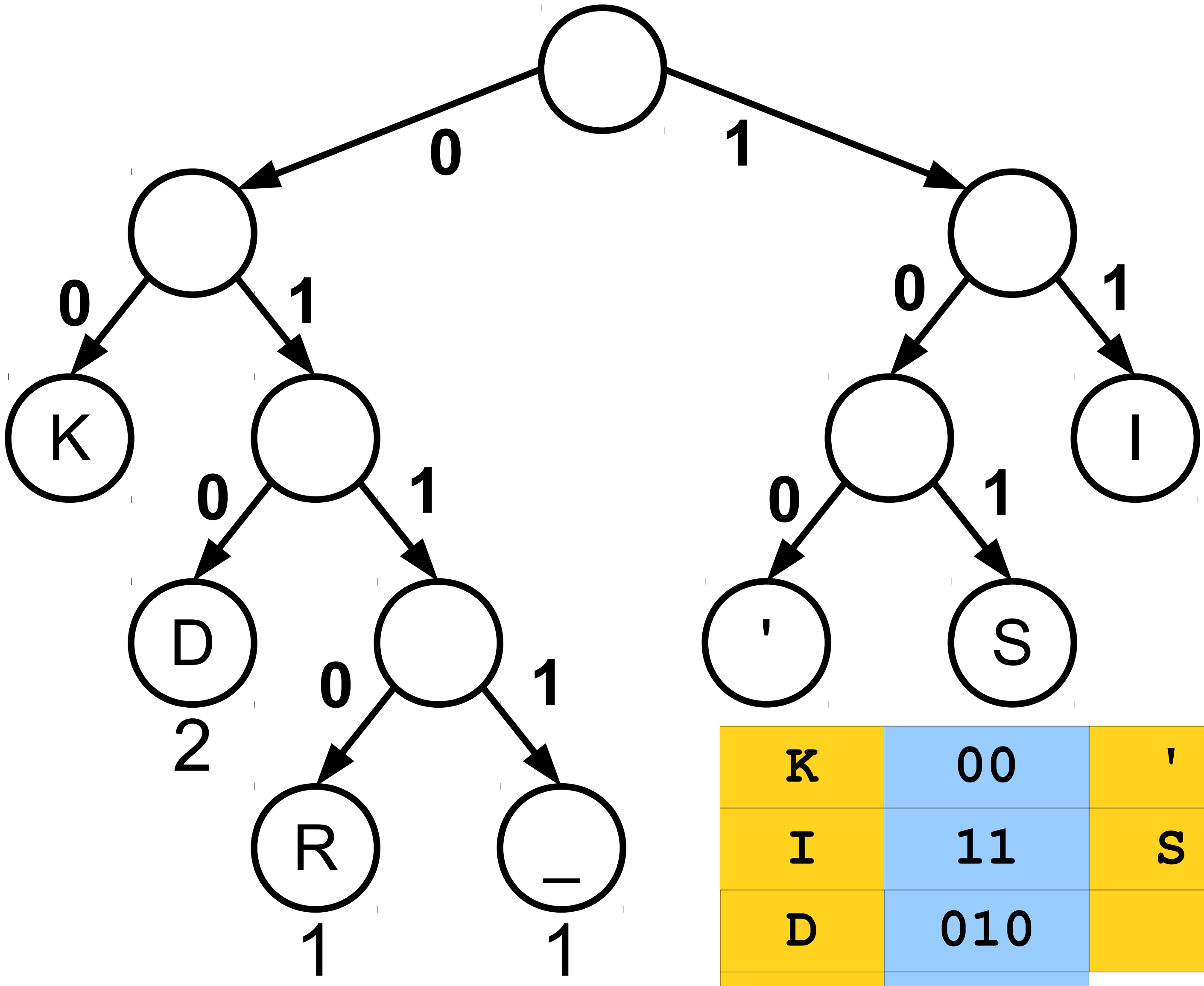
How do we find the best binary tree with this property?

Huffman Coding



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

Huffman Coding

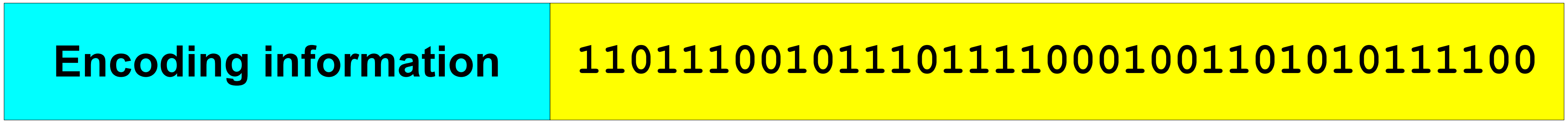


K	00	'	100
I	11	S	101
D	010		0111
R	0110		

Two Important Details

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.



- 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

```
10011111 10001000 11101100 11011001 10
```

10011111	10001000	11101100	11011001	10??????

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

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?
- For example:
  - ¿Cómo estás?
  - السلام عليكم
  - (╯°□°）╯︵ ┻━┻

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

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

```
0000011111010101001100
```

🕌

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!