Beyond Data Structures

It's All Bits and Bytes

- Digital data is stored as sequences 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`, ...

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

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc.
- Computers require everything to be written as zeros and ones.
- To bridge the gap, we need to agree on some way of representing characters as sequences of bits.
- ***Idea:*** Assign each character a sequence of bits called a ***code***.

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
```

01001000010001010100000101000100

- Here’s a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

0100100001000101010000001000100

- Here’s a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H 01000101010000000101000100

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H 01000101010000000101000100

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H E 0100000101000100

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H E 0100000101000100

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	code
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H **E** **A** 01000100

- Here’s a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H E A 01000100

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

H E A D

- Here's a small segment from the ASCII encodings for characters.
- What is the title of this slide?

character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

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 `CS106BLECTURE`: 104 bits.
  - Space for `COPYRIGHTABLE`: 104 bits.
- ***Question:*** Can we reduce the number of bits needed to encode text?

KIRK'S DIKDIK

A Different Encoding

- ASCII uses one byte per character. There are 256 possible bytes.
- If we're specifically writing the string **`KIRK'S DIKDIK`**, which has only seven different characters, using full bytes is wasteful.
- Here's a three-bit encoding we can use to represent the letters in **`KIRK'S DIKDIK`**.
- This uses 37.5% as much space as what ASCII uses.

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

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

Where We're Going

- Storing data using the ASCII encoding is portable across systems, but is not ideal in terms of space usage.
- Building custom codes for specific strings might let us save space.
- ***Idea:*** Use this approach to build a ***compression algorithm*** to reduce the amount of space needed to store text.

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**.
- Here's the frequencies of each letter.
- So far, we've given each letter codes of the same length.
- ***Key Question:*** Can we give shorter encodings to more common characters?

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

A First Attempt

character	*code*
K	0
I	1
D	00
R	01
'	10
S	11
␣	100

01010101110000100010

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

A First Attempt

character	*code*
K	0
I	1
D	00
R	01
'	10
S	11
␣	100

0101010111000010010

A First Attempt

character	*code*
K	0
I	1
D	00
R	01
'	10
S	11
␣	100

0101010111000001000010

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

A First Attempt

character	*code*
K	0
I	1
D	00
R	01
'	10
S	11
␣	100

01010101110000100010

01	01	01	01	1	10	0	00	10	0	0	10
R	R	R	R	I	'	K	D	'	K	K	'

A First Attempt

character	*code*
K	0
I	1
D	00
R	01
'	10
S	11
␣	100

01010101110

01	01	01	01	1	10	0	00	10	0	0	10
R	R	R	R	I	'	K	D	'	K	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.
- Here's a sample prefix code for the letters in **`KIRK'S DIKDIK`**.

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

Prefix Codes

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

Prefix Codes

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

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

Prefix Codes

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

1001001100001101110011101101110110

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

Prefix Codes

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

1001001100001101110011101101110110

Prefix Codes

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

`1001001100001101110011101101110110`

Prefix Codes

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

1001001100001101110011101101110110

Prefix Codes

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

<u>10</u>0100110000110111001110110111 0110

Prefix Codes

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

<u>10</u>01001100001101110011101101110110

10
K

Prefix Codes

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

`1001001100001101110011101101110110`

10
K

Prefix Codes

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

10**0**100110000110111001110110110110

10	
K	

Prefix Codes

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

10**01**001100001101110011101101110110

10
K

Prefix Codes

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

10**01**001100001101110011101101110110

10	01
K	I

Prefix Codes

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

`1001001100001101110011101101110110`

10	01
K	I

Prefix Codes

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

1001**0**0110000110111001110110110110

10	01
K	I

Prefix Codes

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

1001**00**110000110111001110110111011 0

10	01
K	I

Prefix Codes

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

1001**001**100001101110011101101110110

10	01
K	I

Prefix Codes

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

1001**001**100001101110011101101110110

10	01	001
K	I	R

Prefix Codes

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

`1001001100001101110011101101110110`

10	01	001
K	I	R

Prefix Codes

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

  `1001001100001101110011101101110110`

- This uses just 34 bits, compared to our initial 104. Wow!
- But where did this code come from? How could you come up with codes like this for other strings?

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

1001001100001101110011101101110110

---

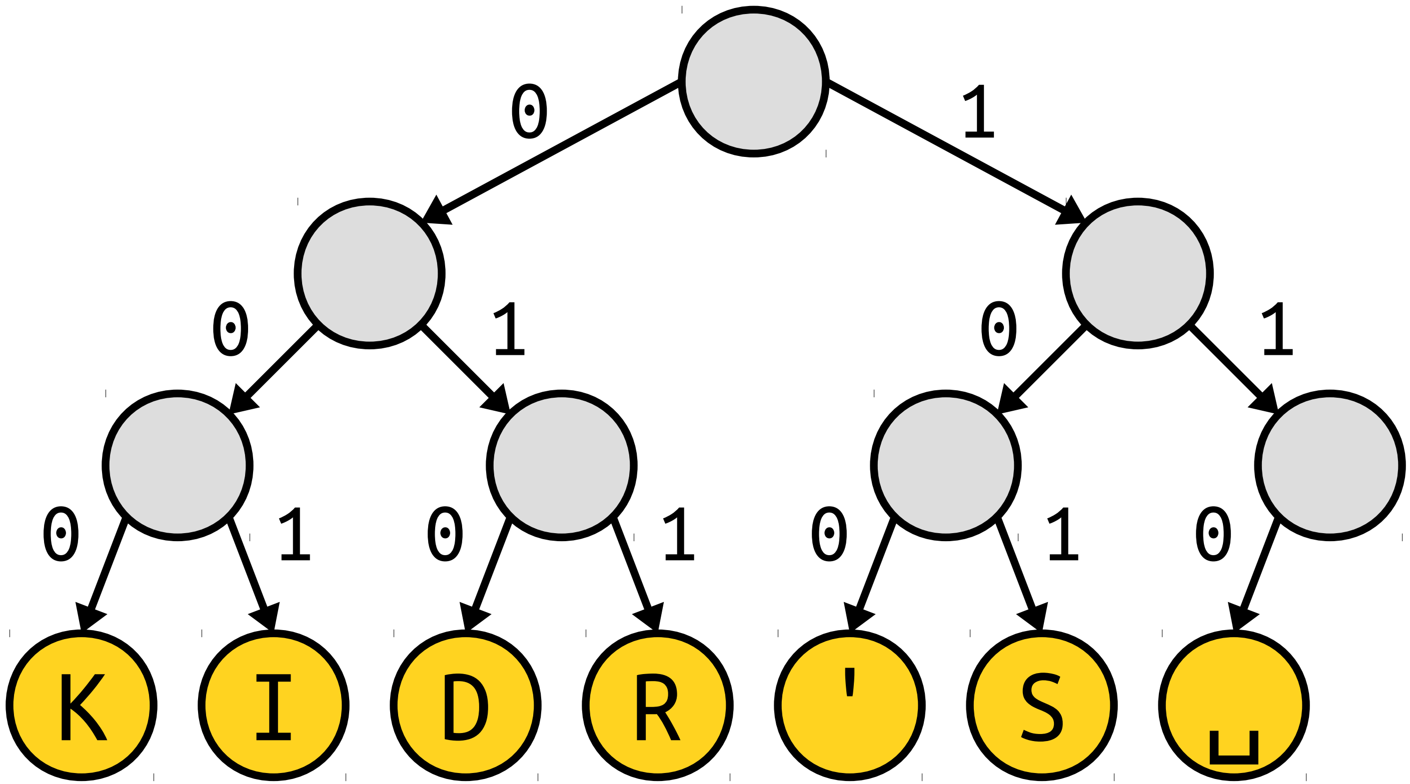
character	*code*
K	1111110
I	111110
D	11110
R	1110
'	110
S	10
␣	0

1111110111110111011111101101001111011111011111101111011111101111110

How do you find a “good” prefix code?

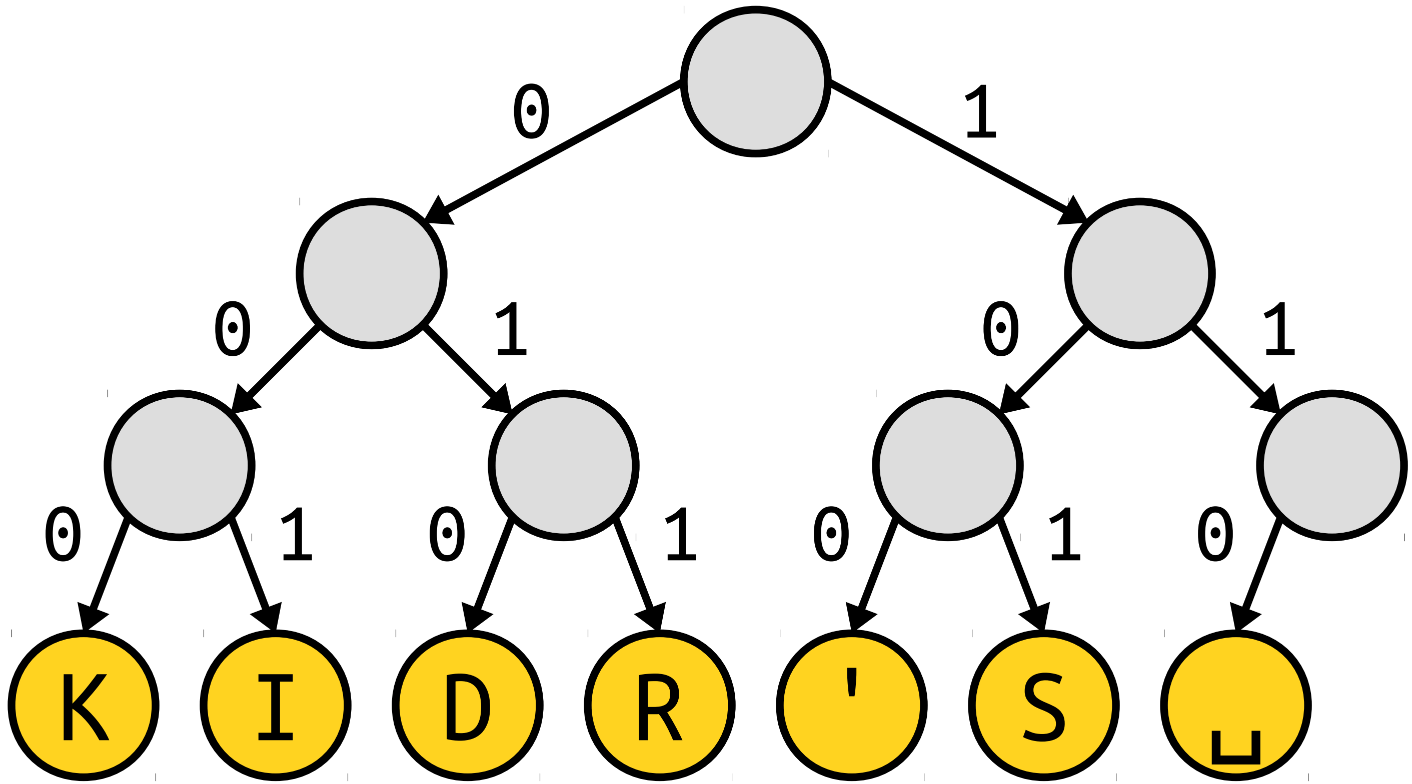
The Main Insight

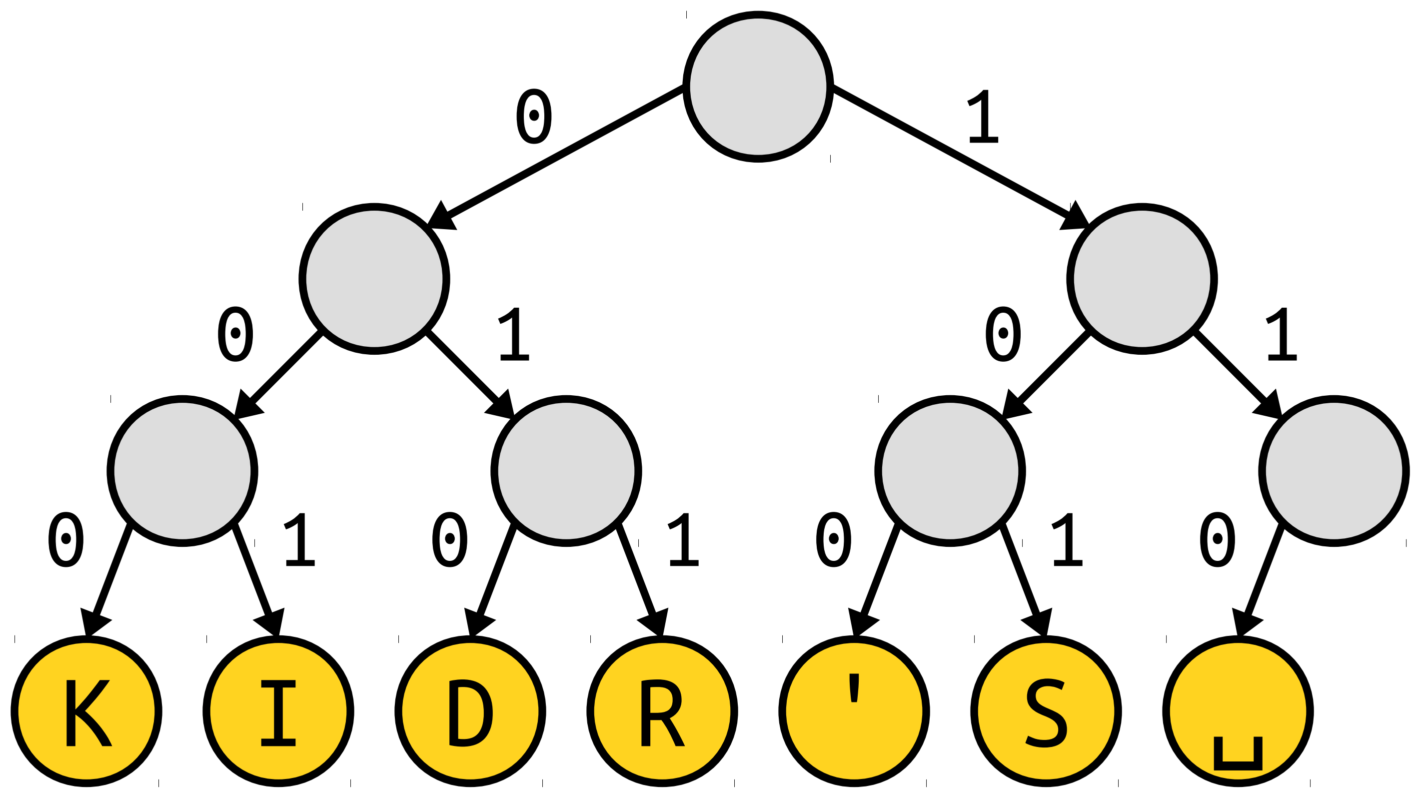
character	*code*
K	000
I	001
D	010
R	011
'	100
S	101
	110



This special type of binary tree is called a ***coding tree***.

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

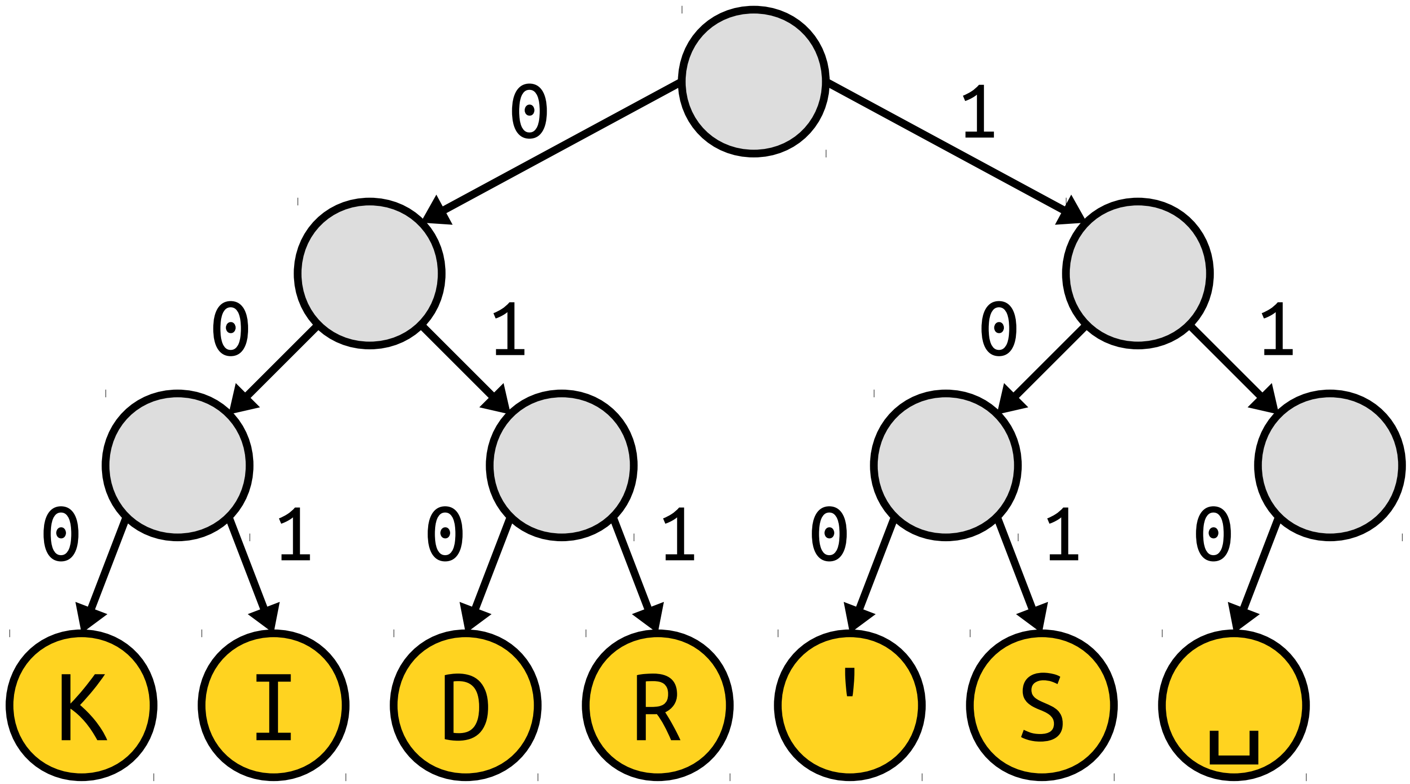




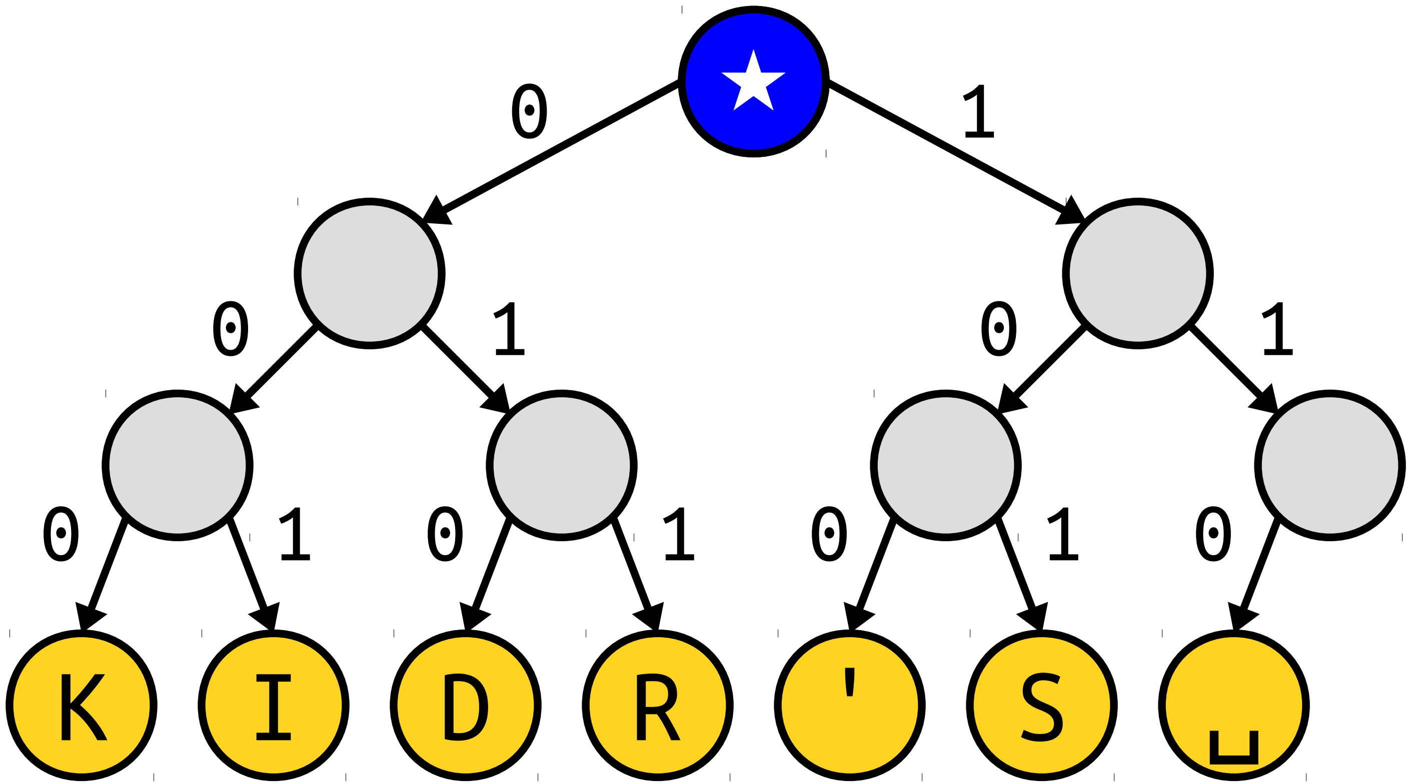
0
1
0
1
0
1
0
1
0
1
0
1
0
K
I
D
R
'
S
␣

101000001

What is the title of this slide?



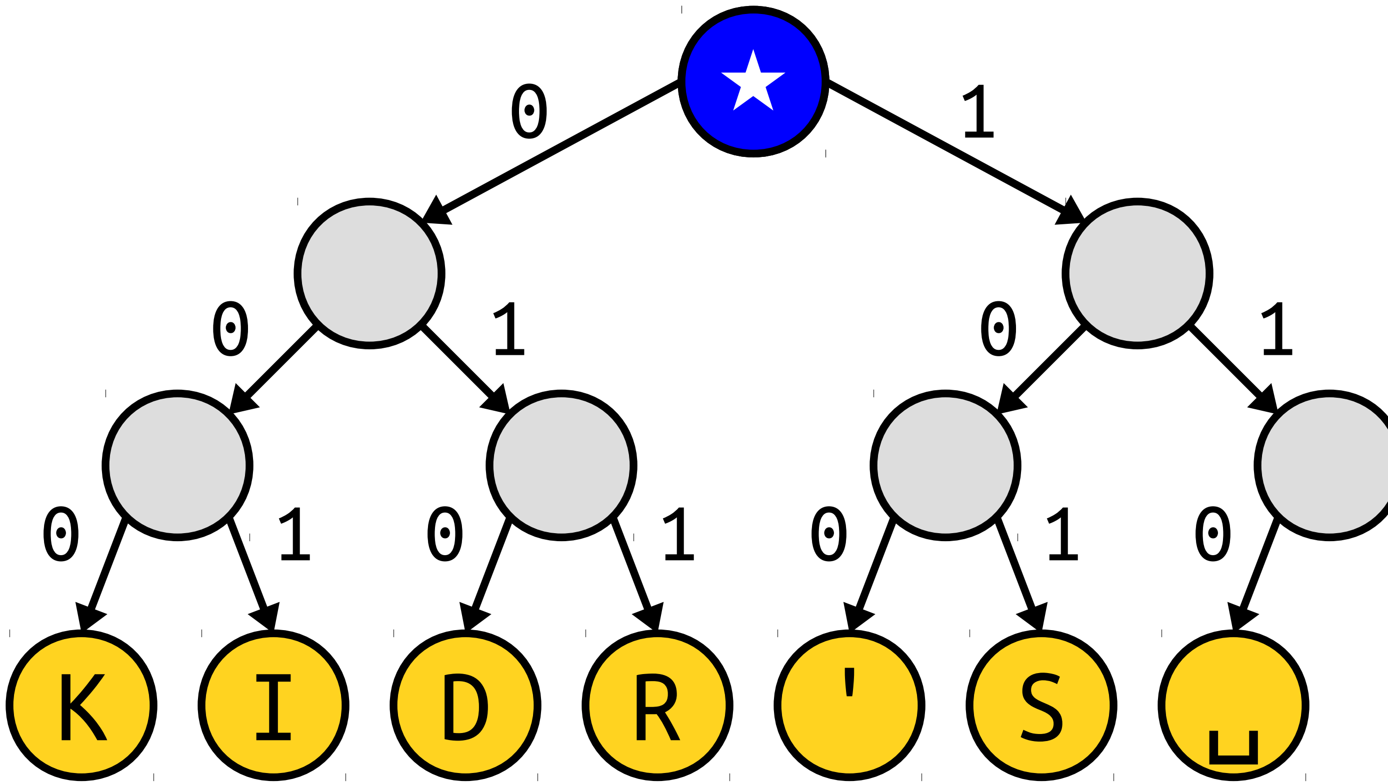
101000001



What is the title of this slide?

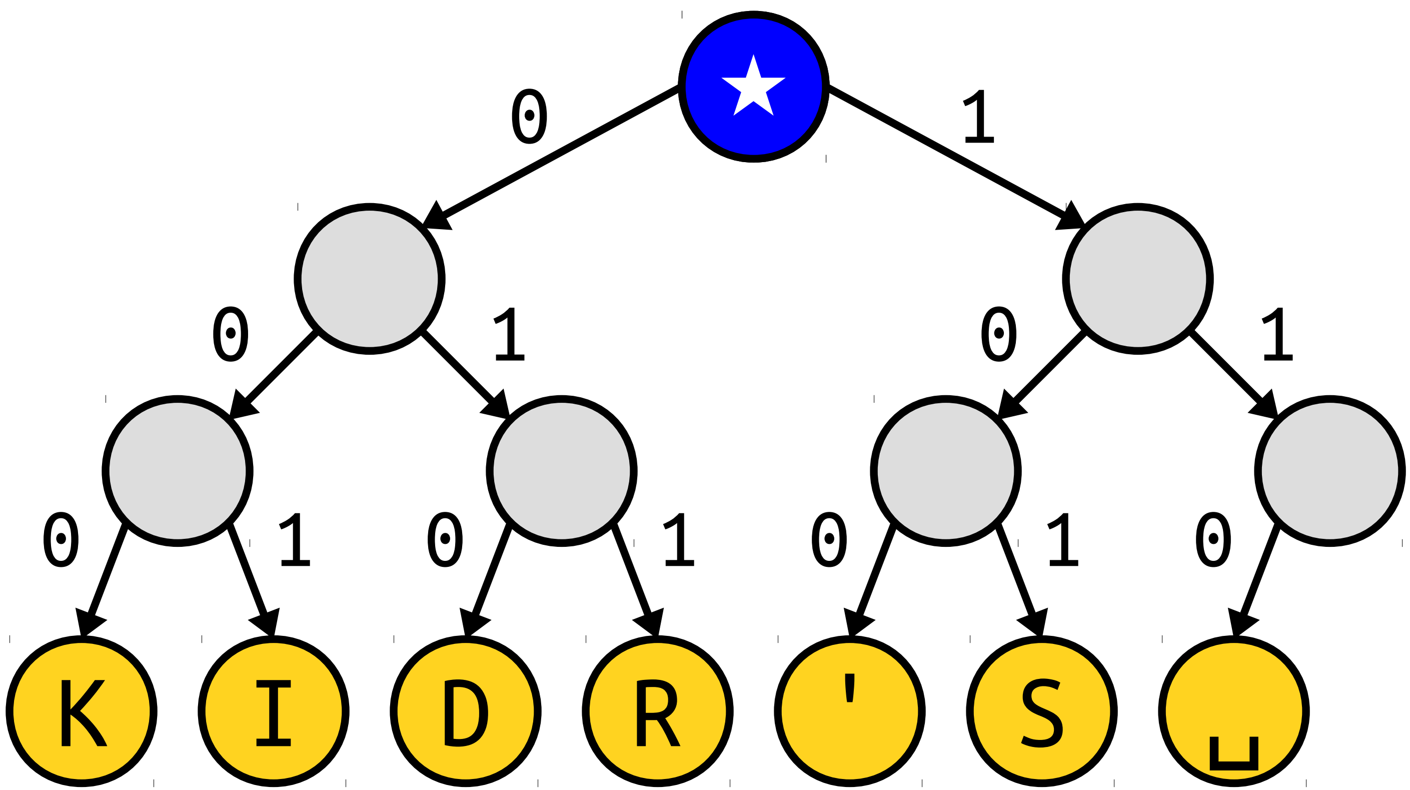
101000001

What is the title of this slide?



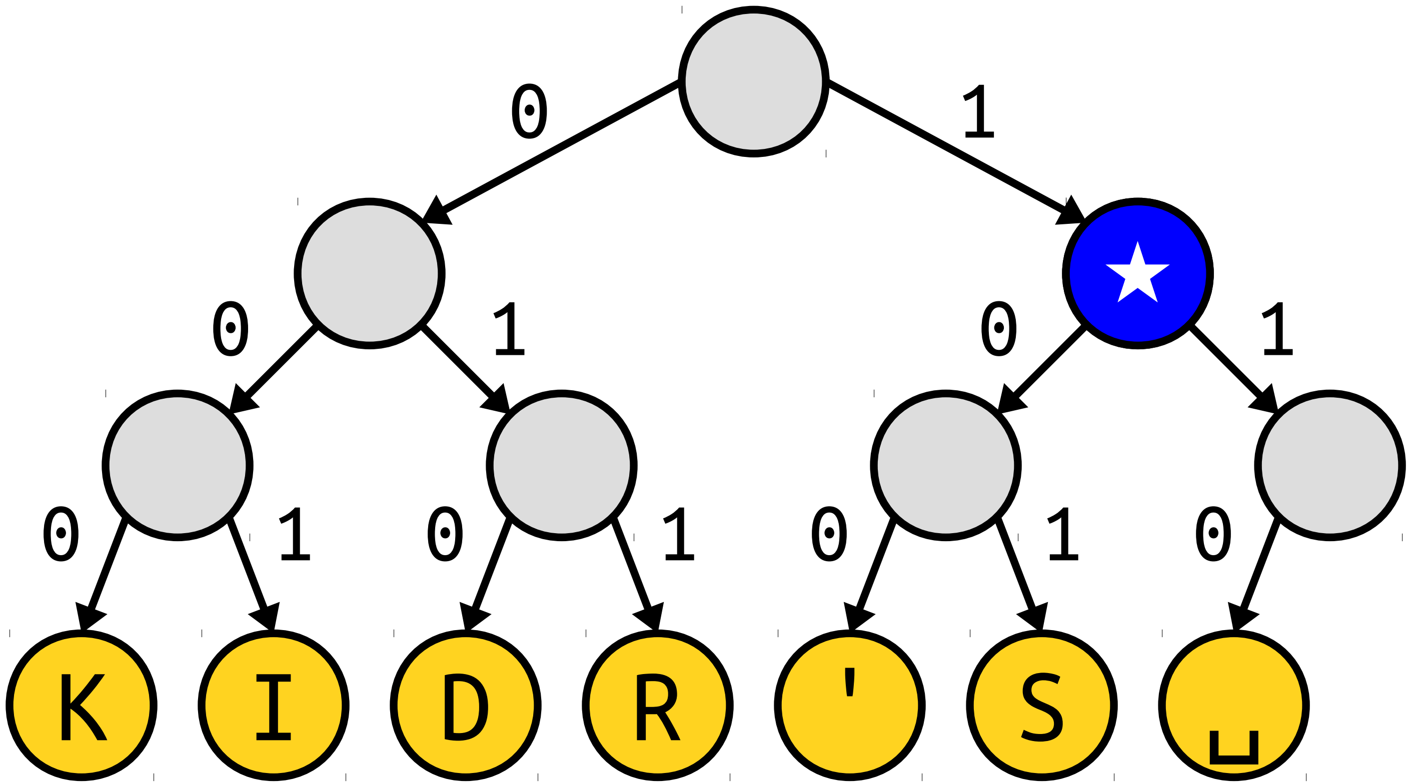
101000001

What is the title of this slide?

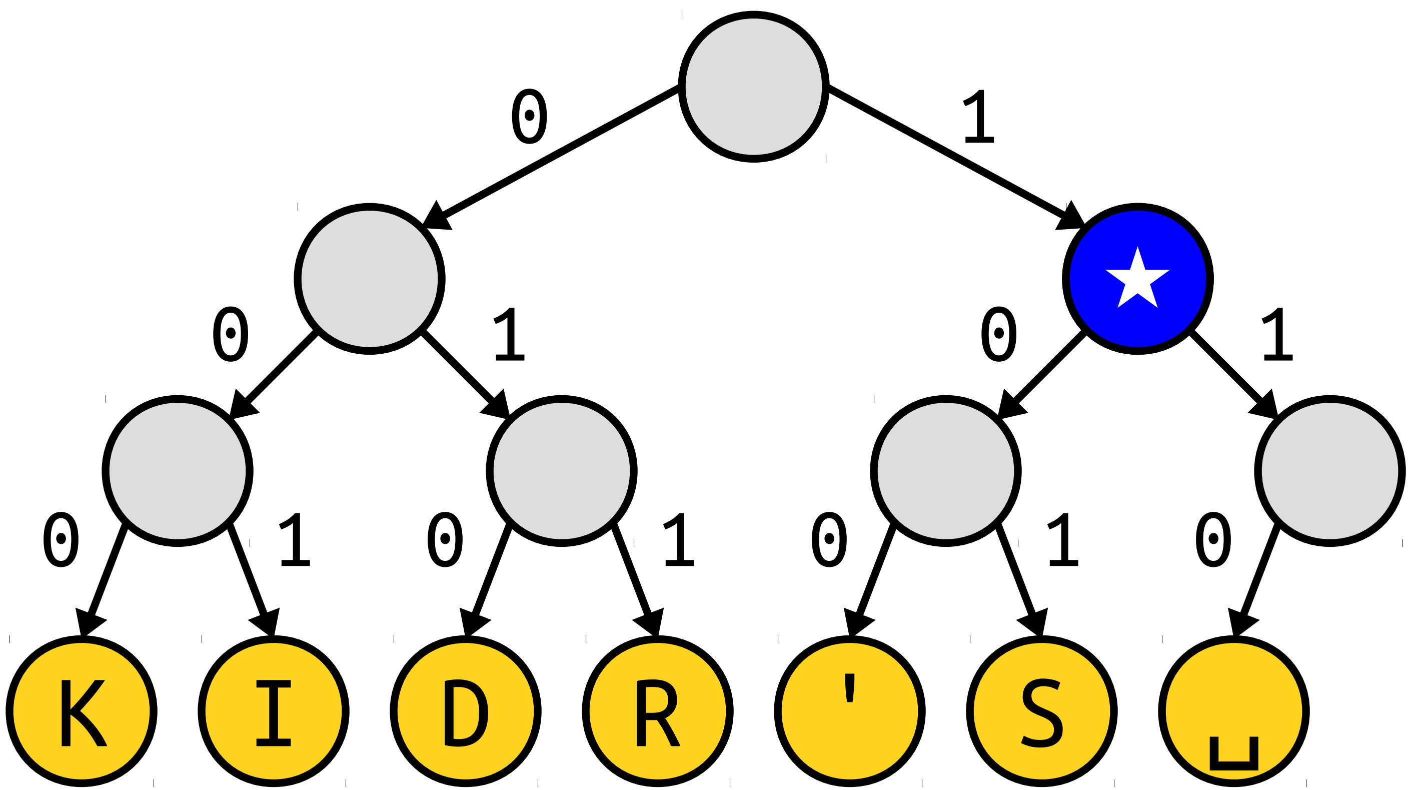


101000001

What is the title of this slide?



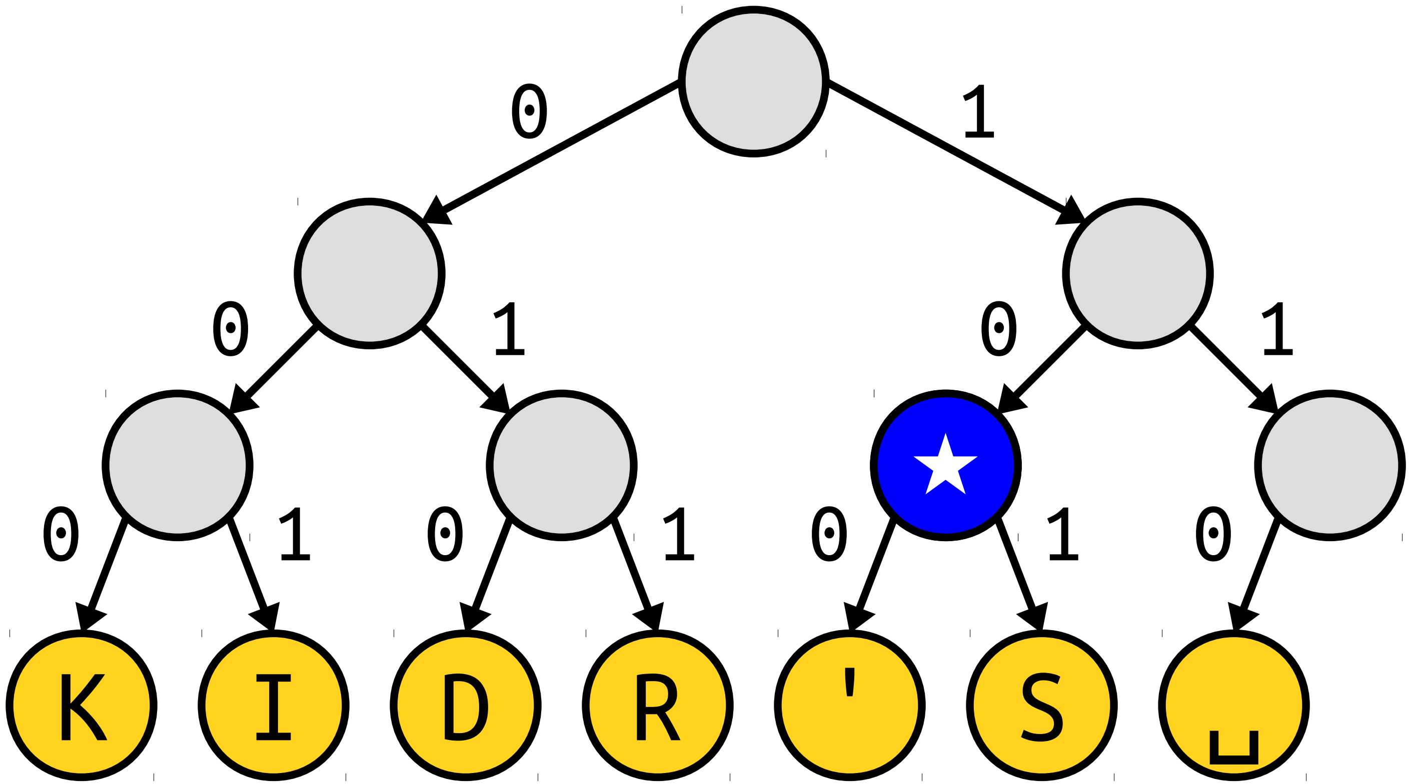
101000001



What is the title of this slide?

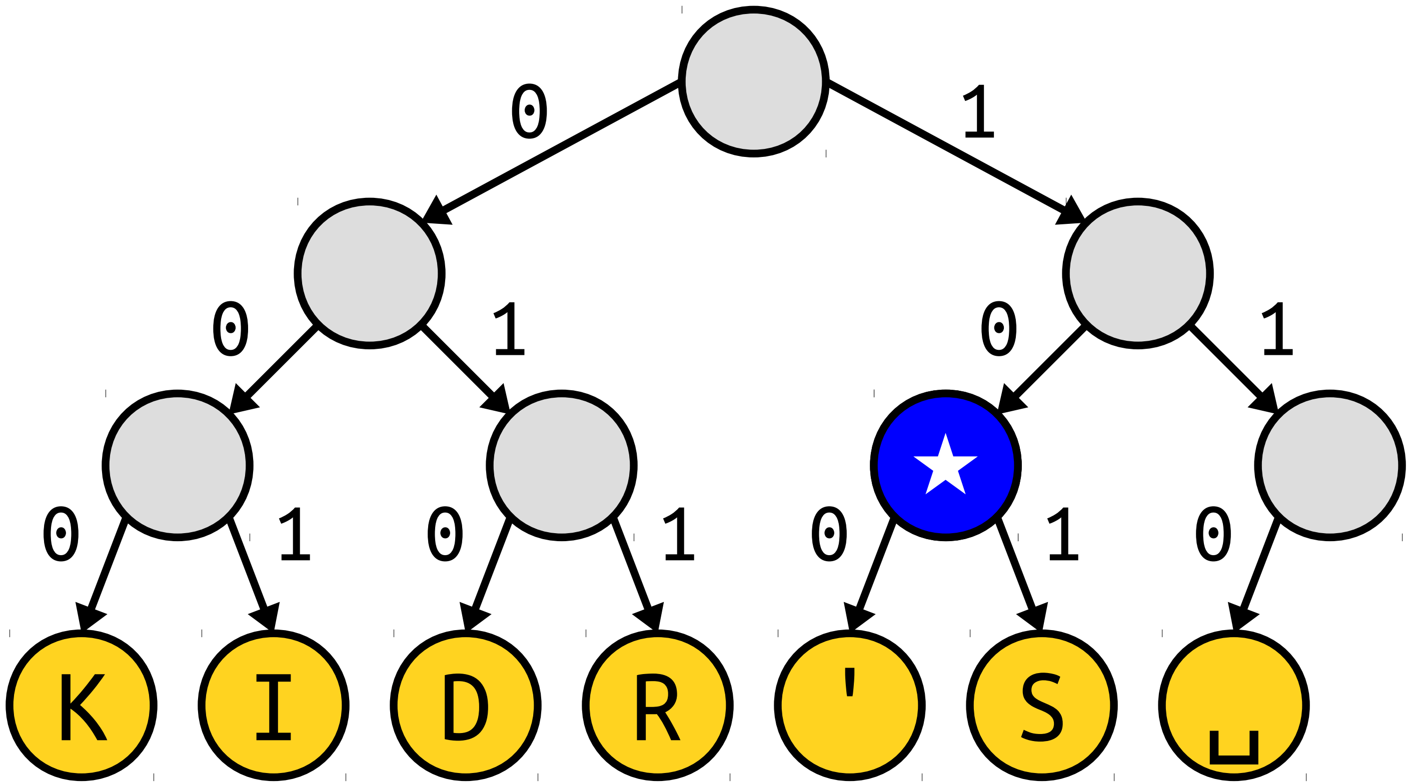
10100000001

What is the title of this slide?

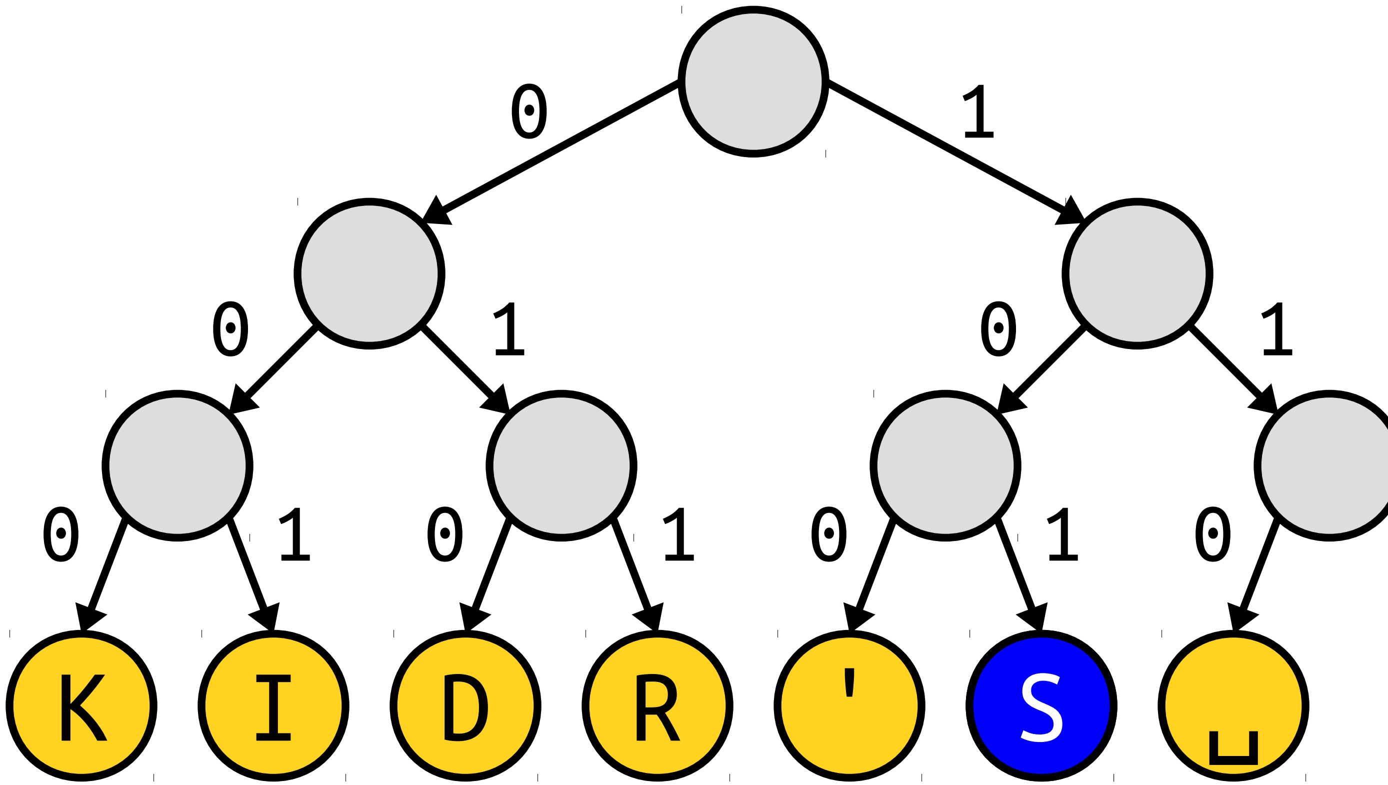


101000001

What is the title of this slide?



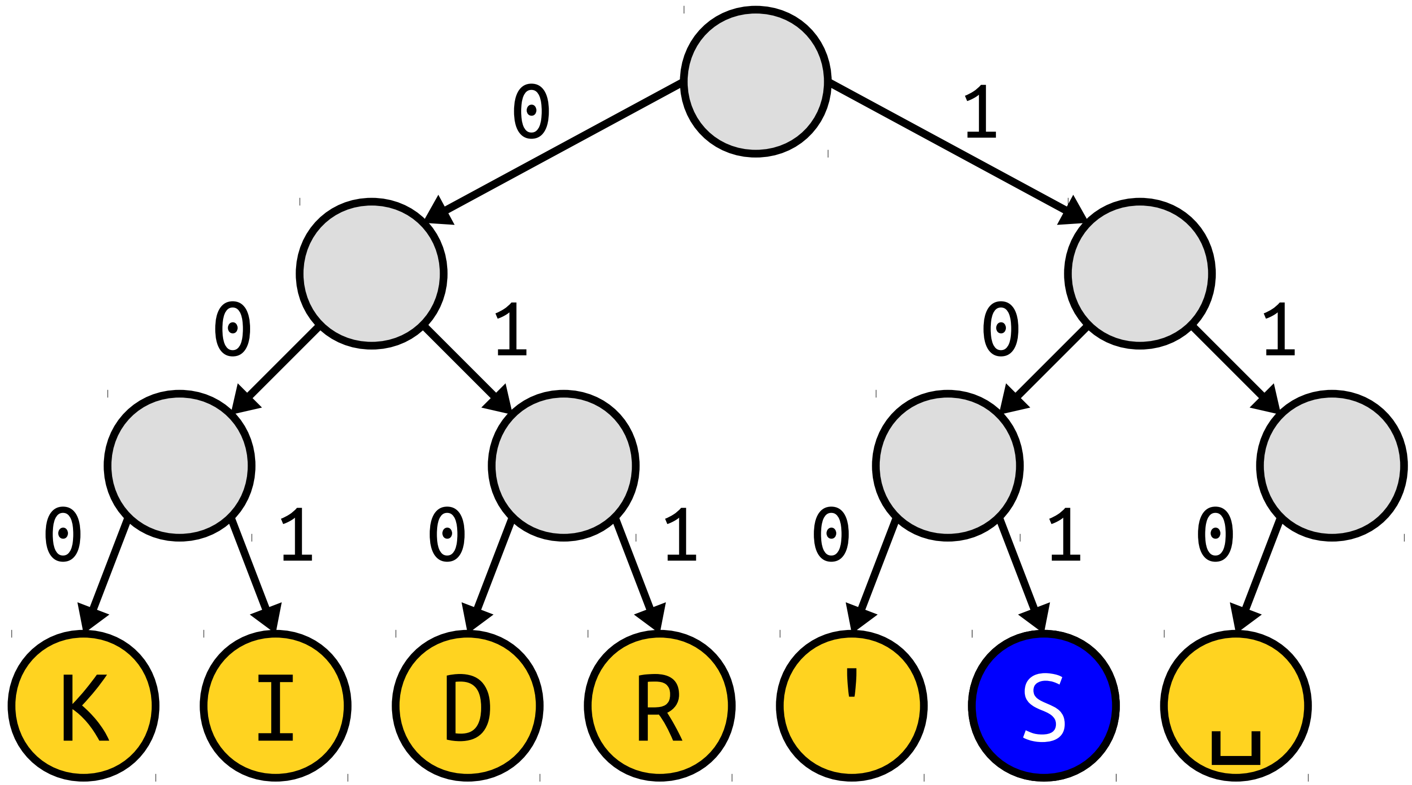
101000001



What is the title of this slide?

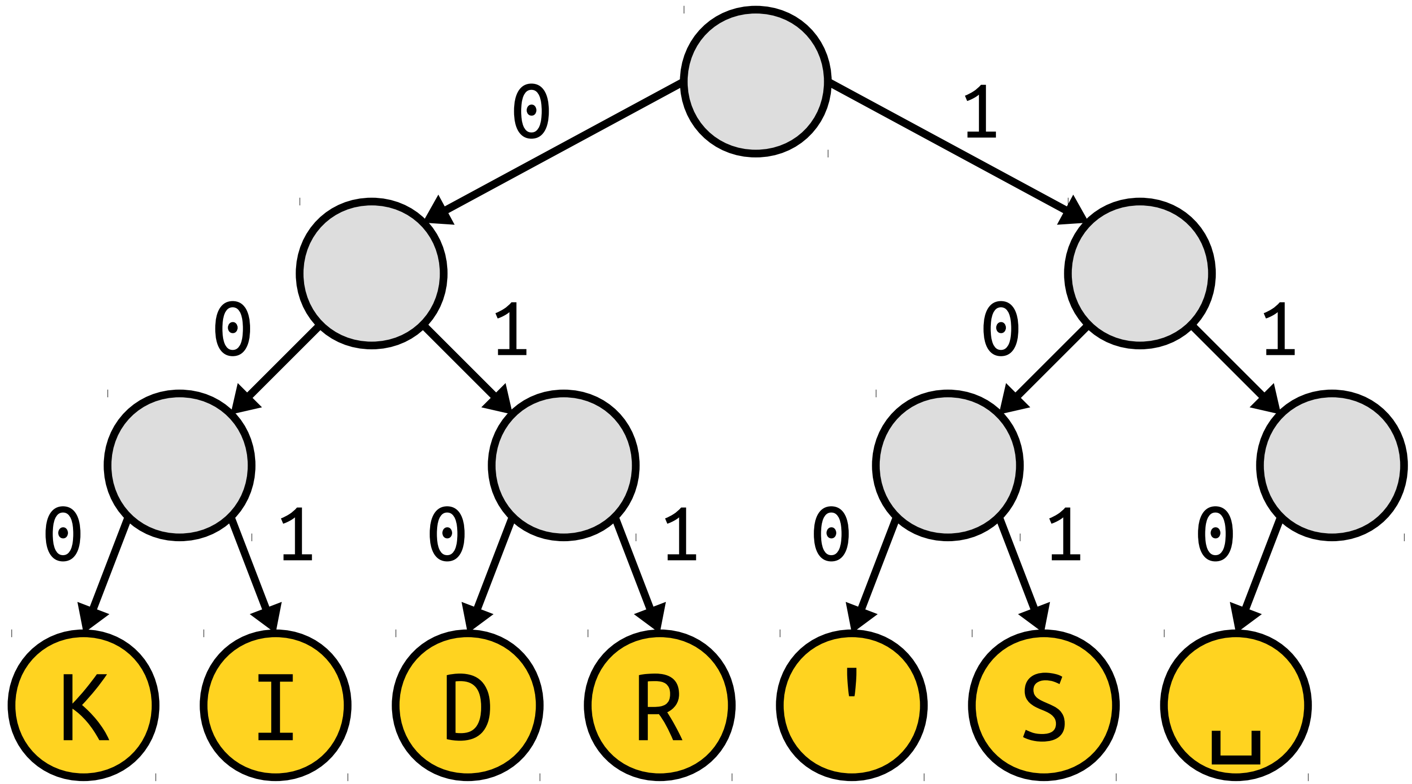
S 000001

What is the title of this slide?



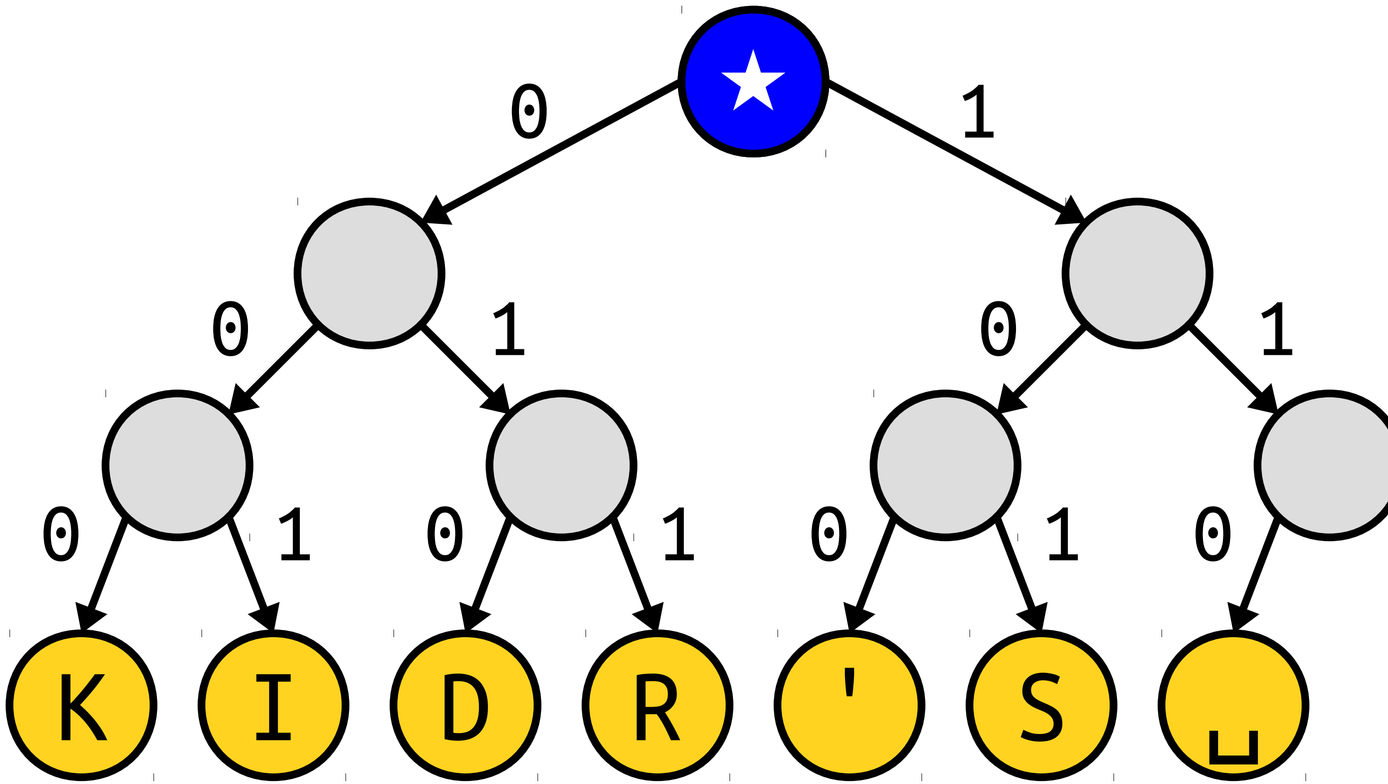
S 000001

What is the title of this slide?



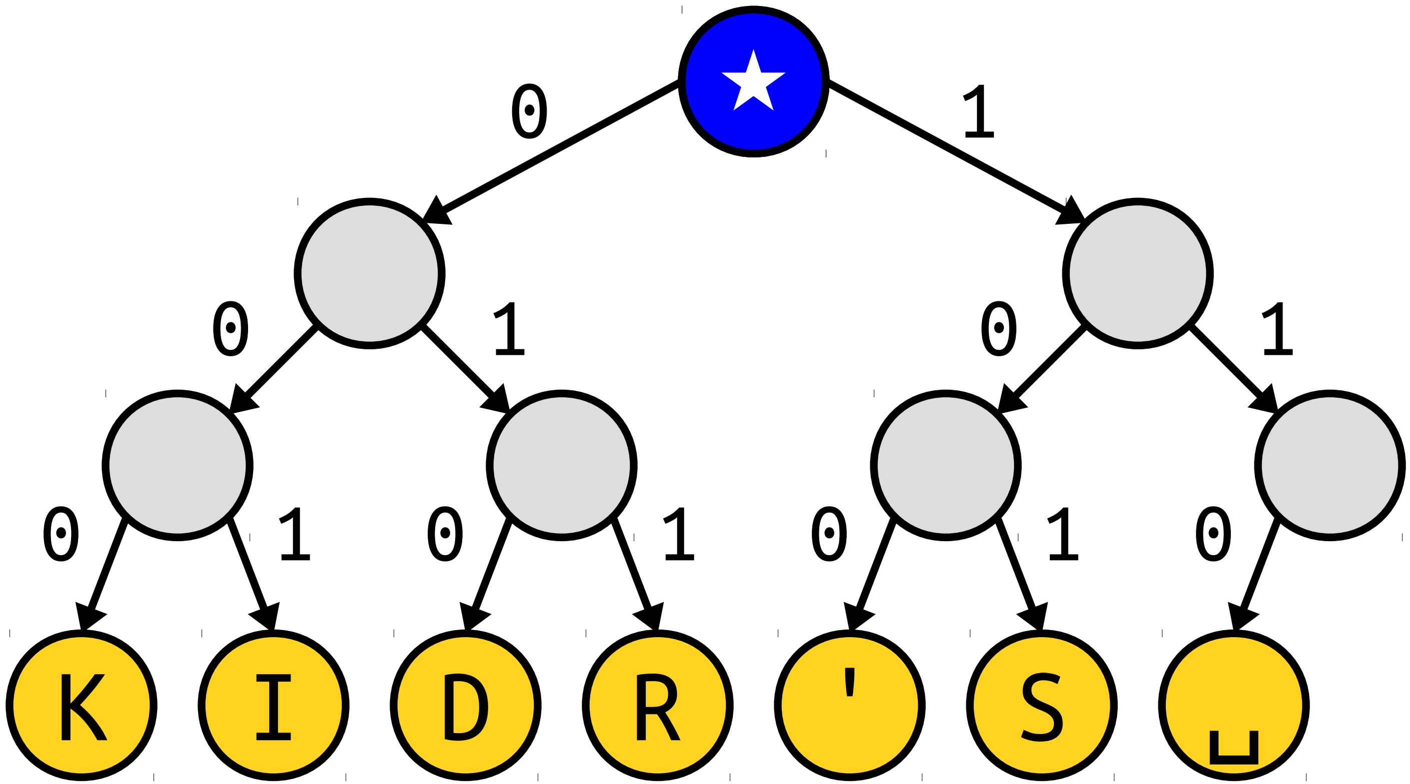
S 000001

What is the title of this slide?



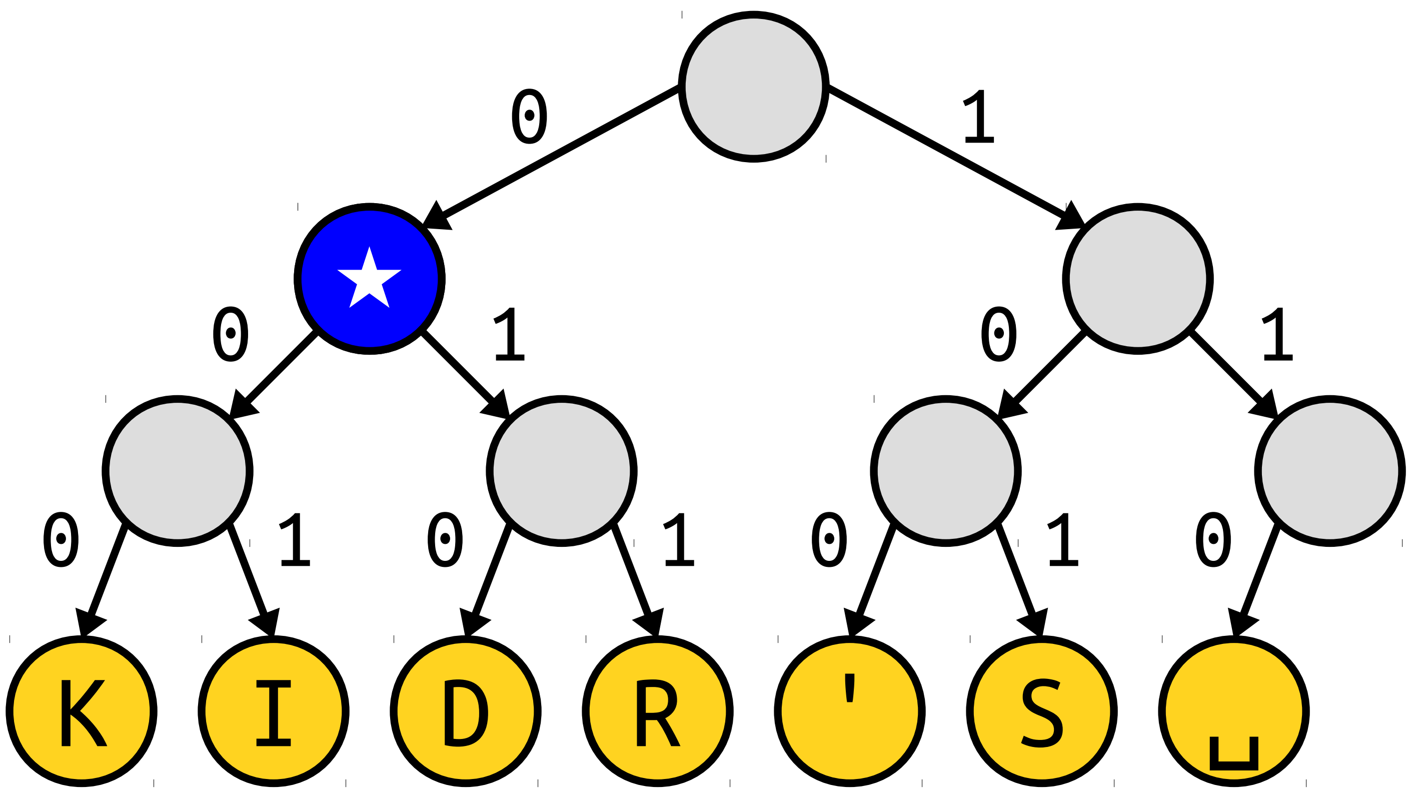
S 000001

What is the title of this slide?

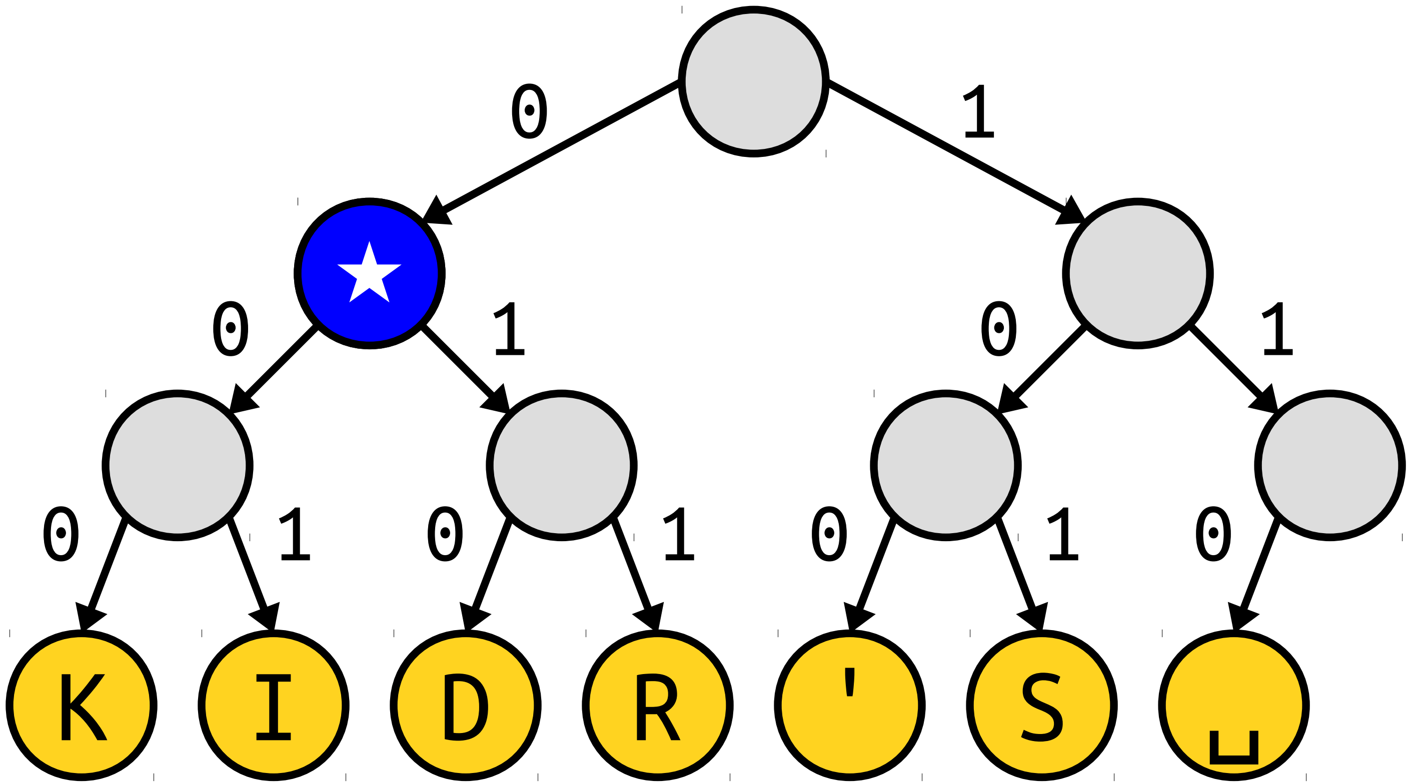


S 000001

What is the title of this slide?



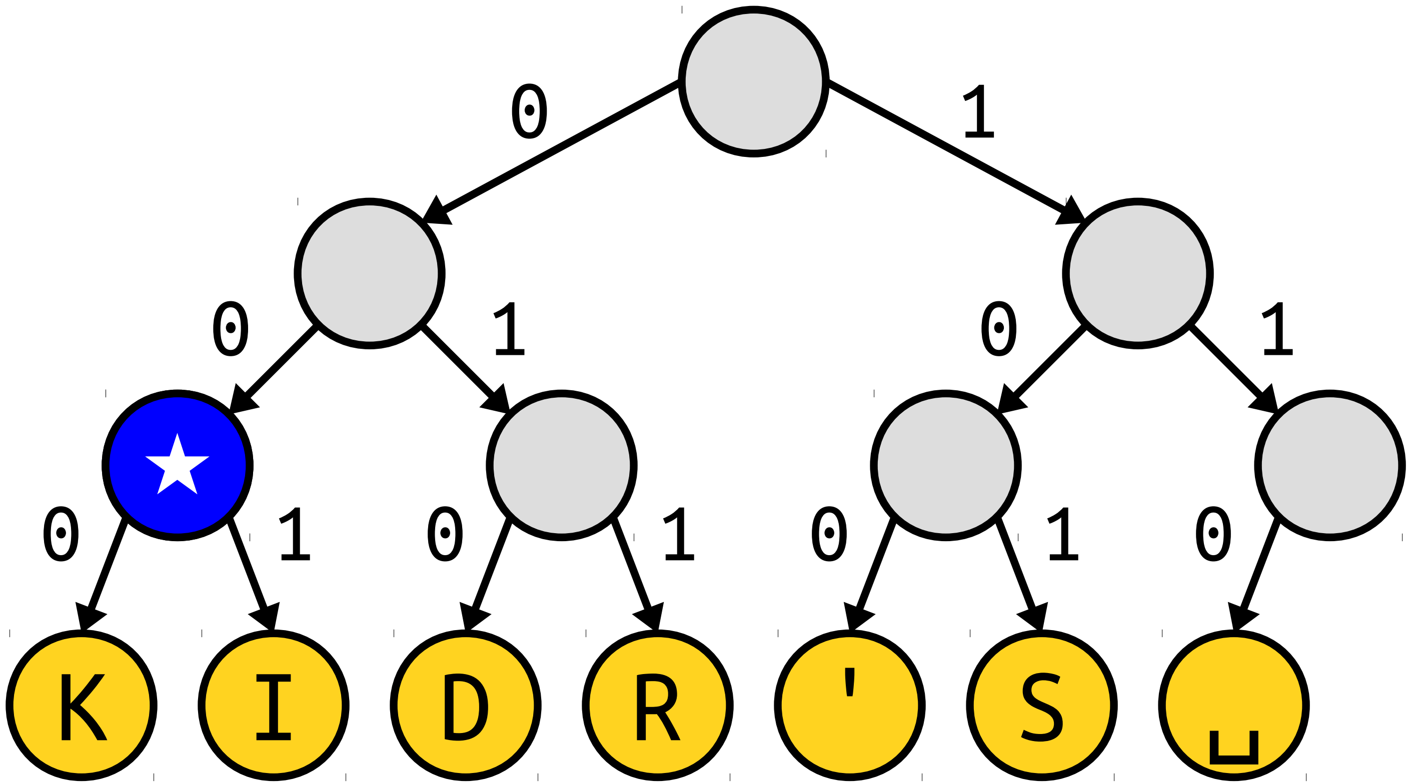
S 000001



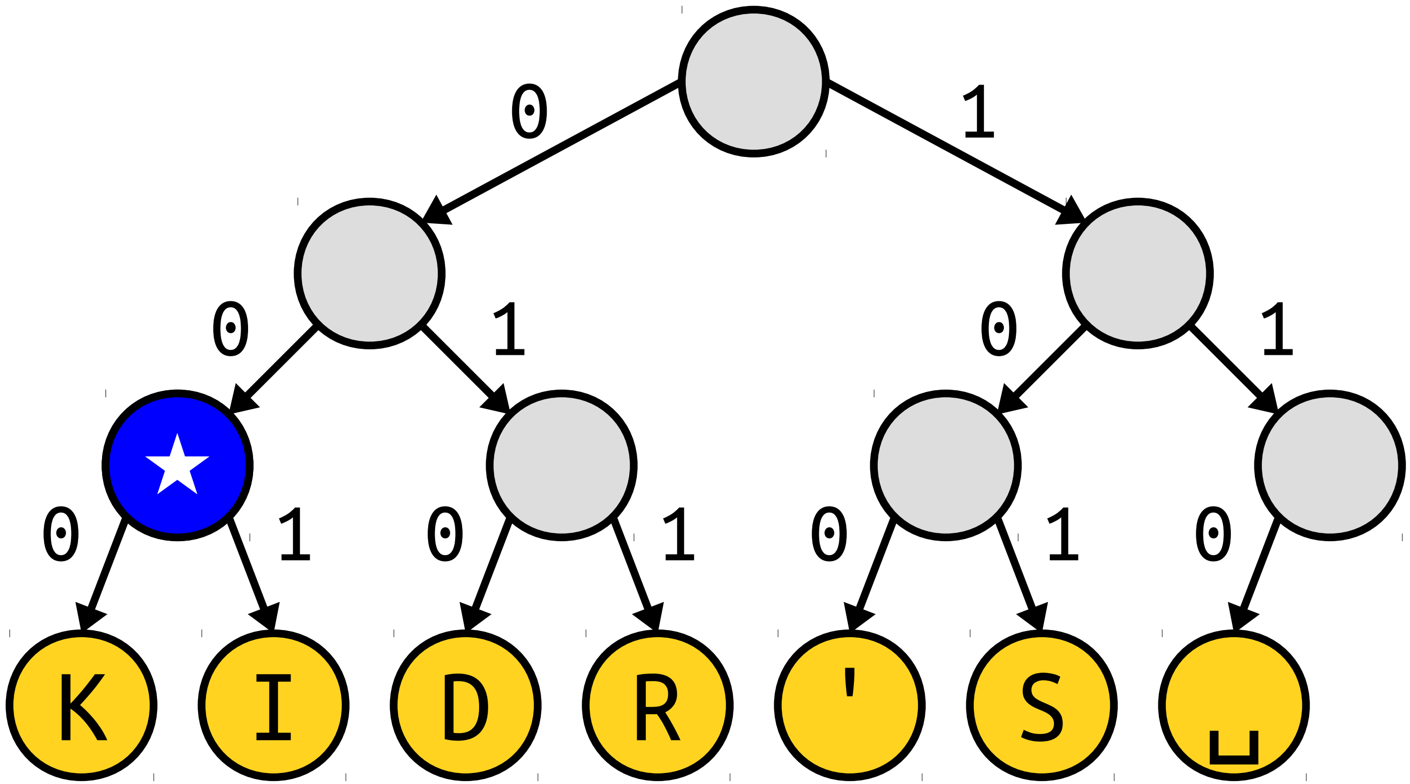
What is the title of this slide?

S 000001

What is the title of this slide?



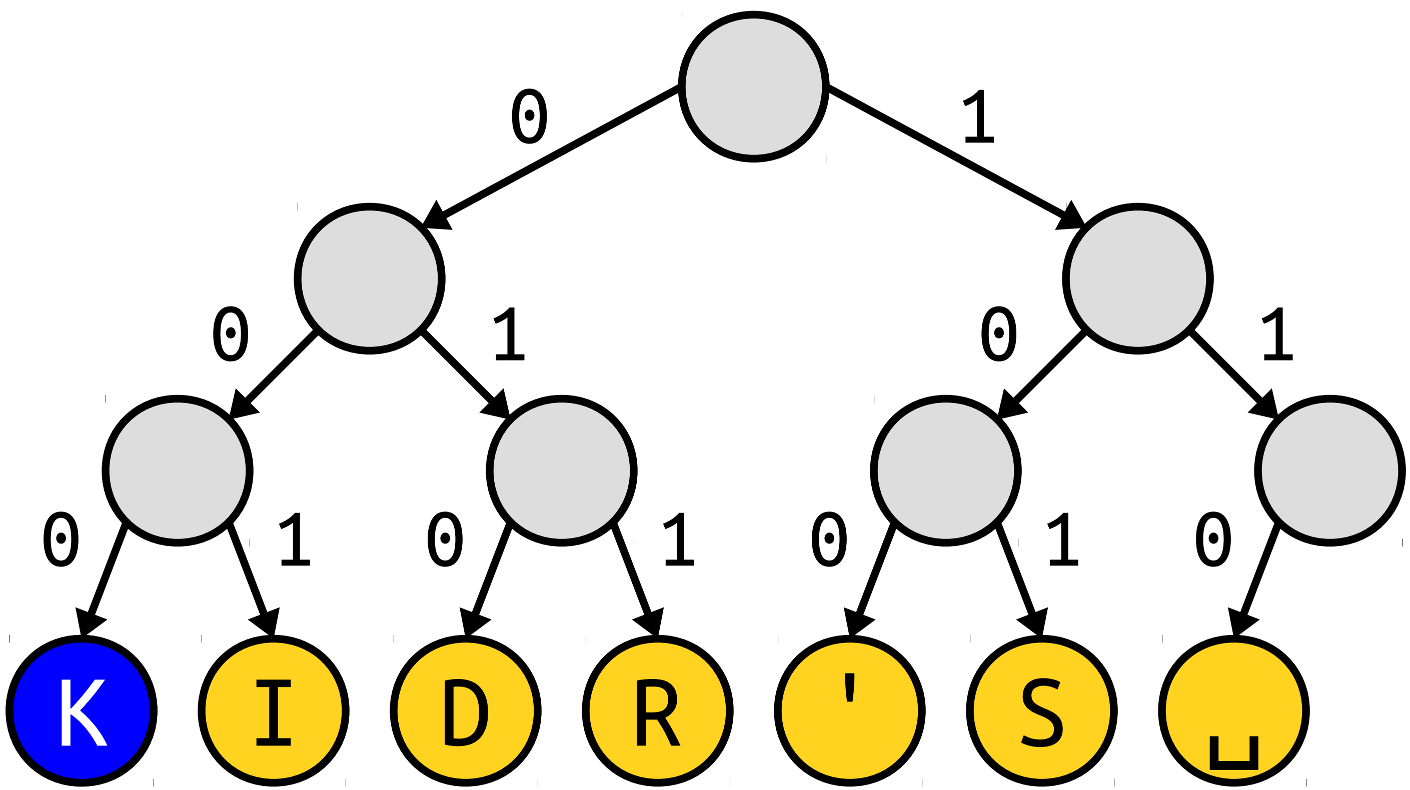
S 000001



What is the title of this slide?

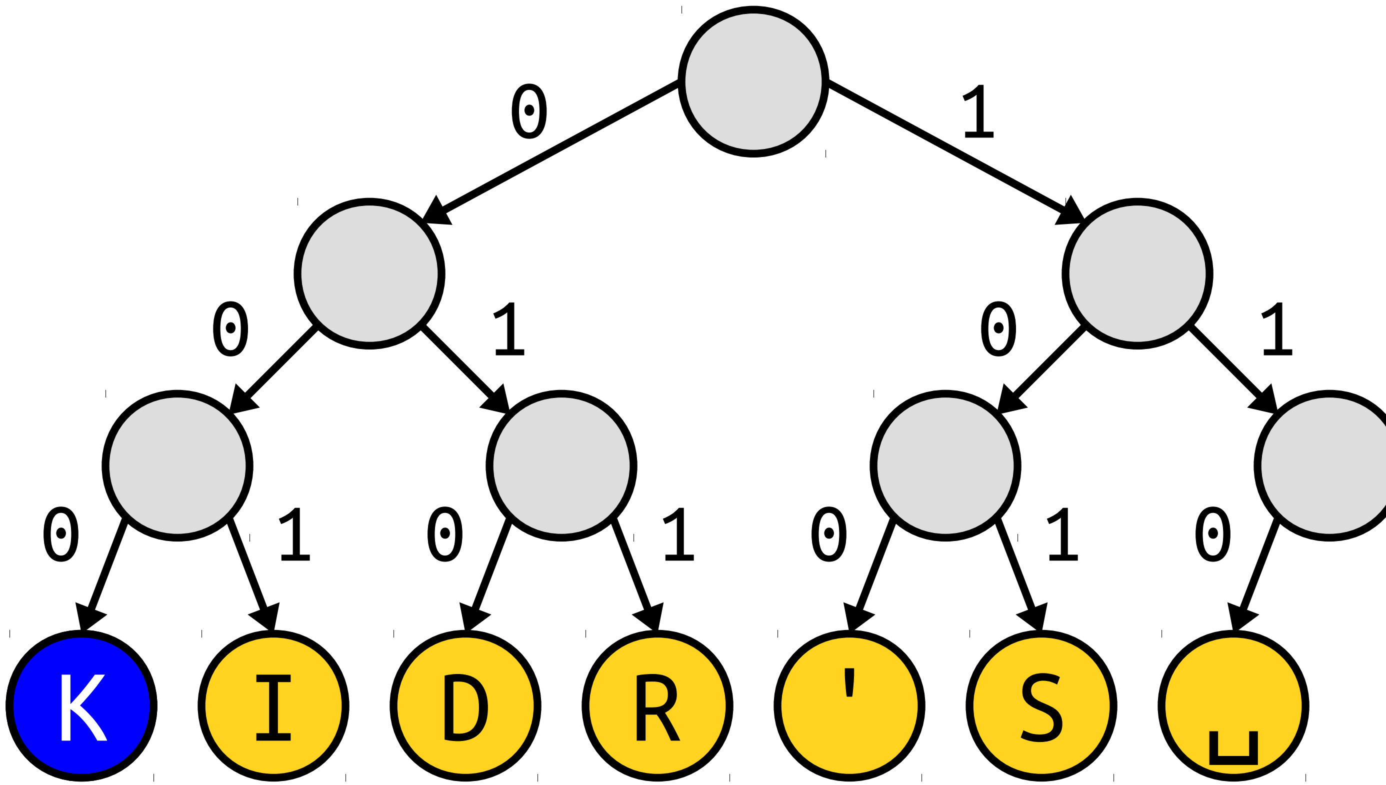
S 000001

What is the title of this slide?



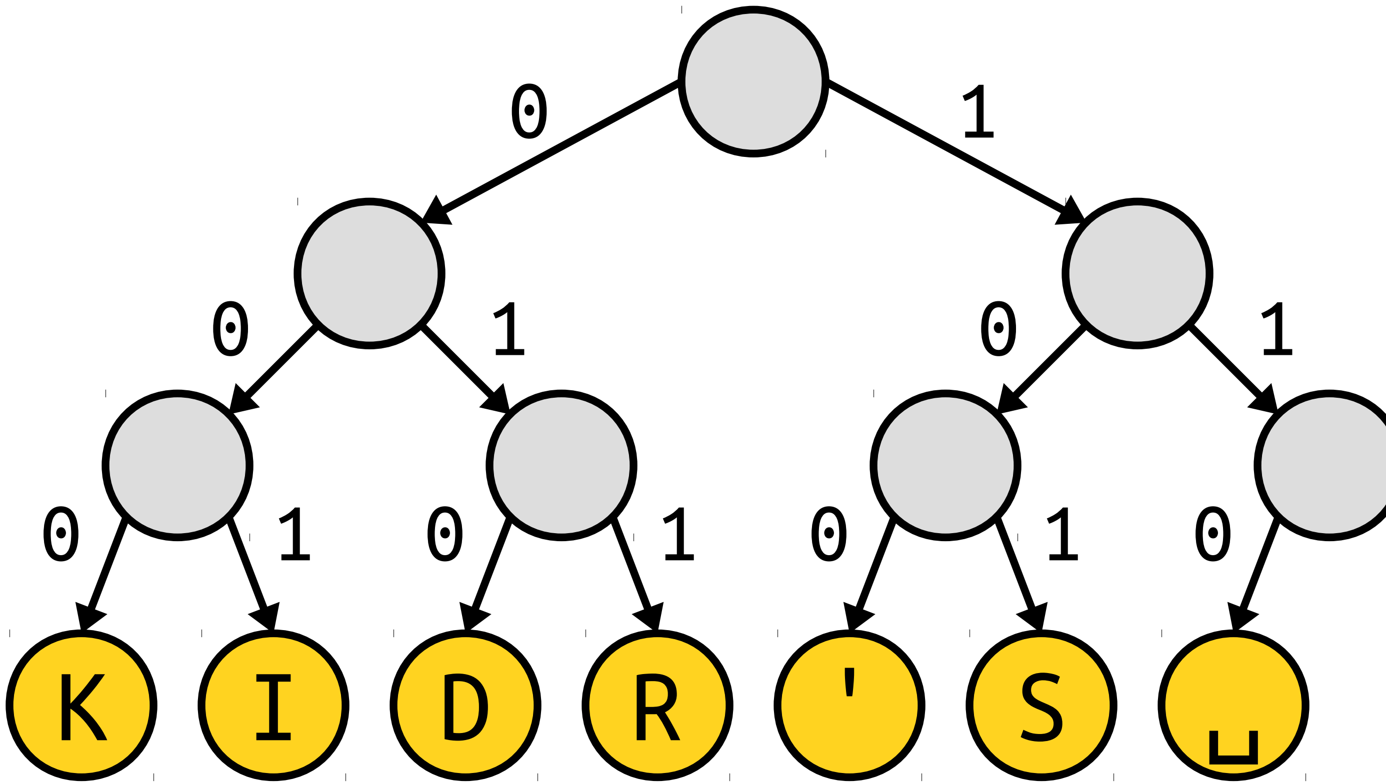
S K 001

What is the title of this slide?



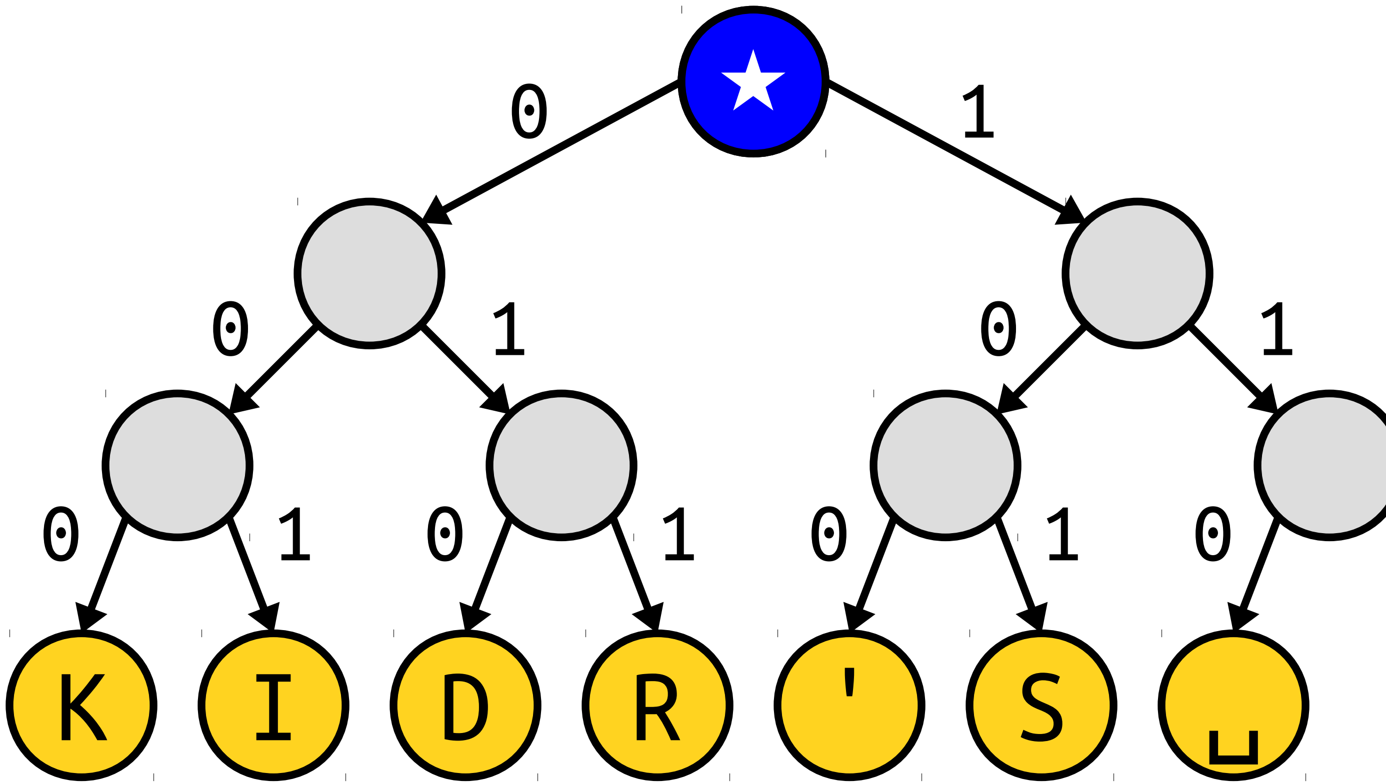
S K 001

What is the title of this slide?



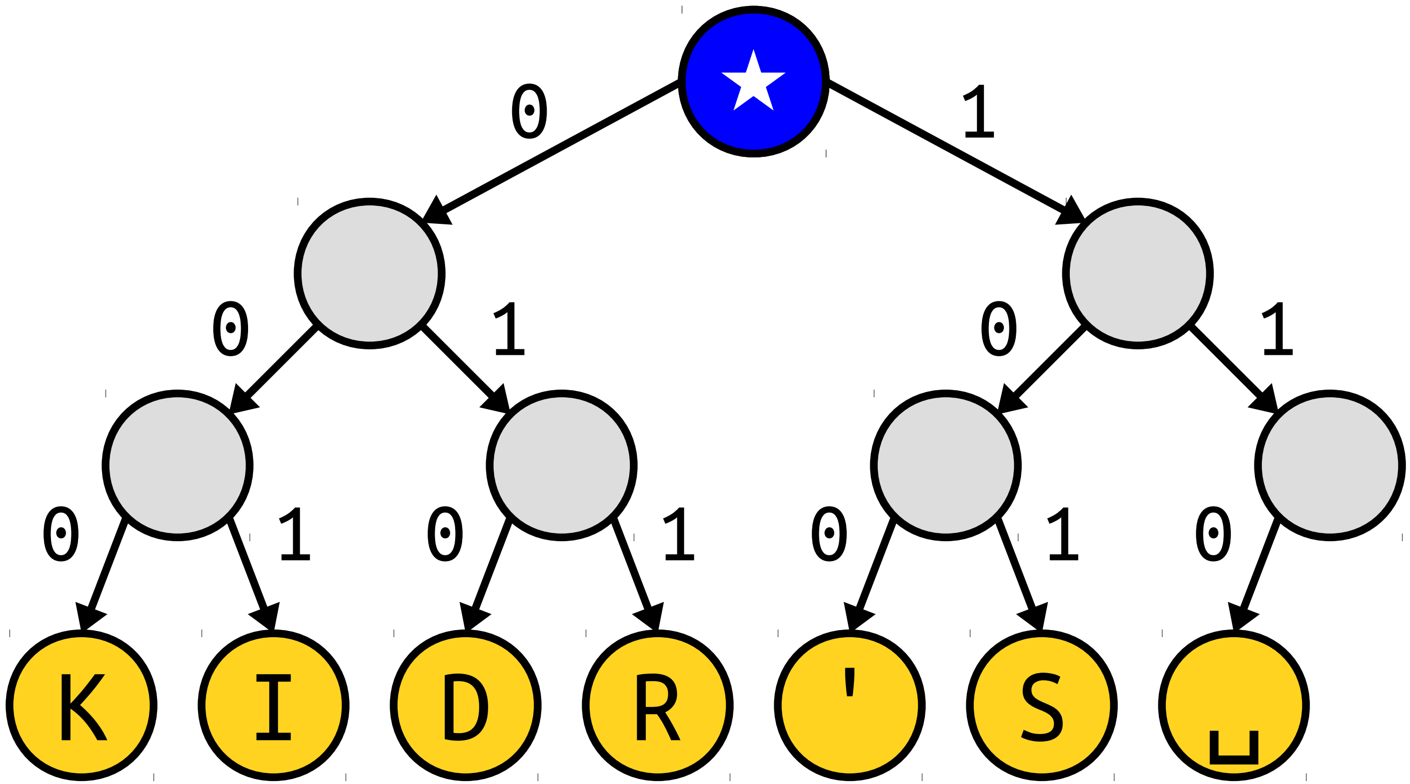
S K 001

What is the title of this slide?

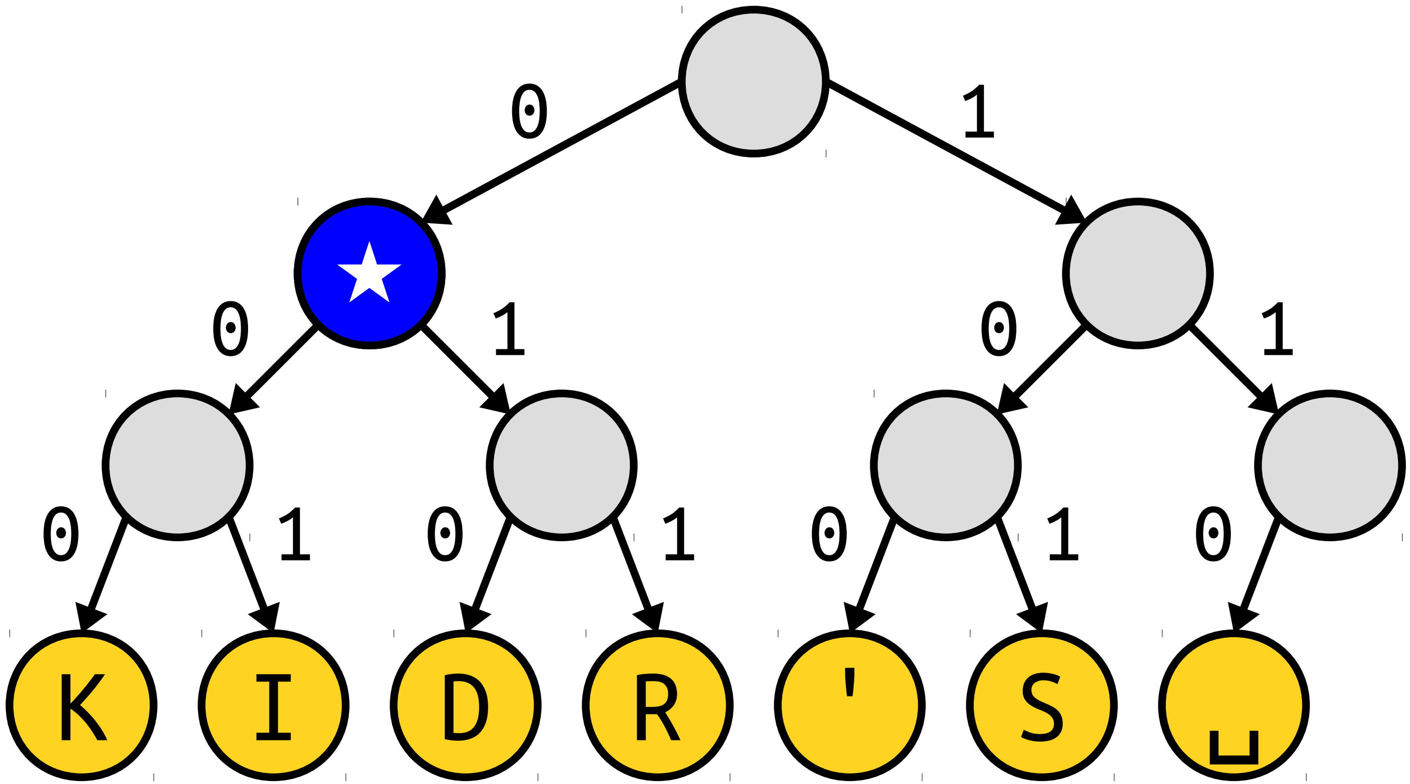


S K 001

What is the title of this slide?



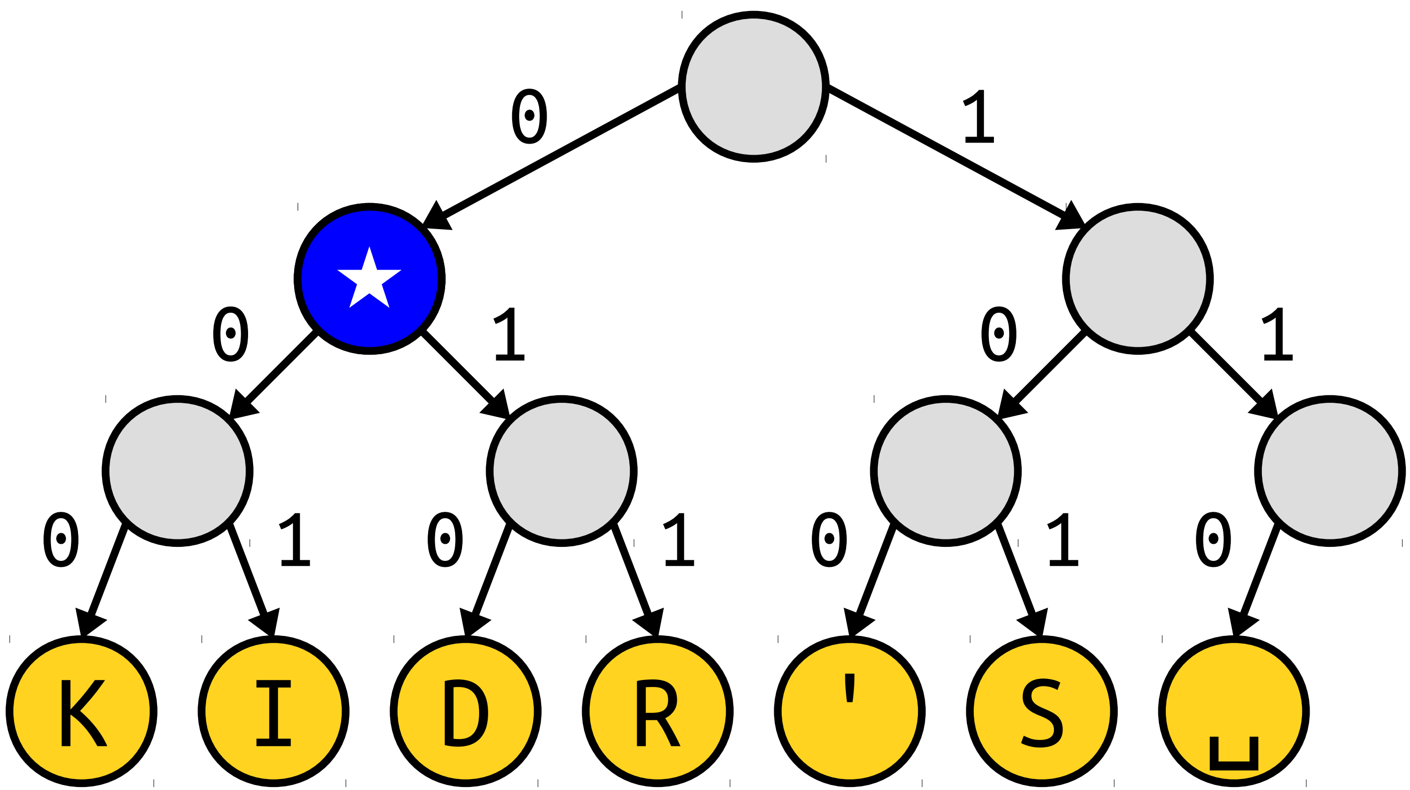
S K 001



What is the title of this slide?

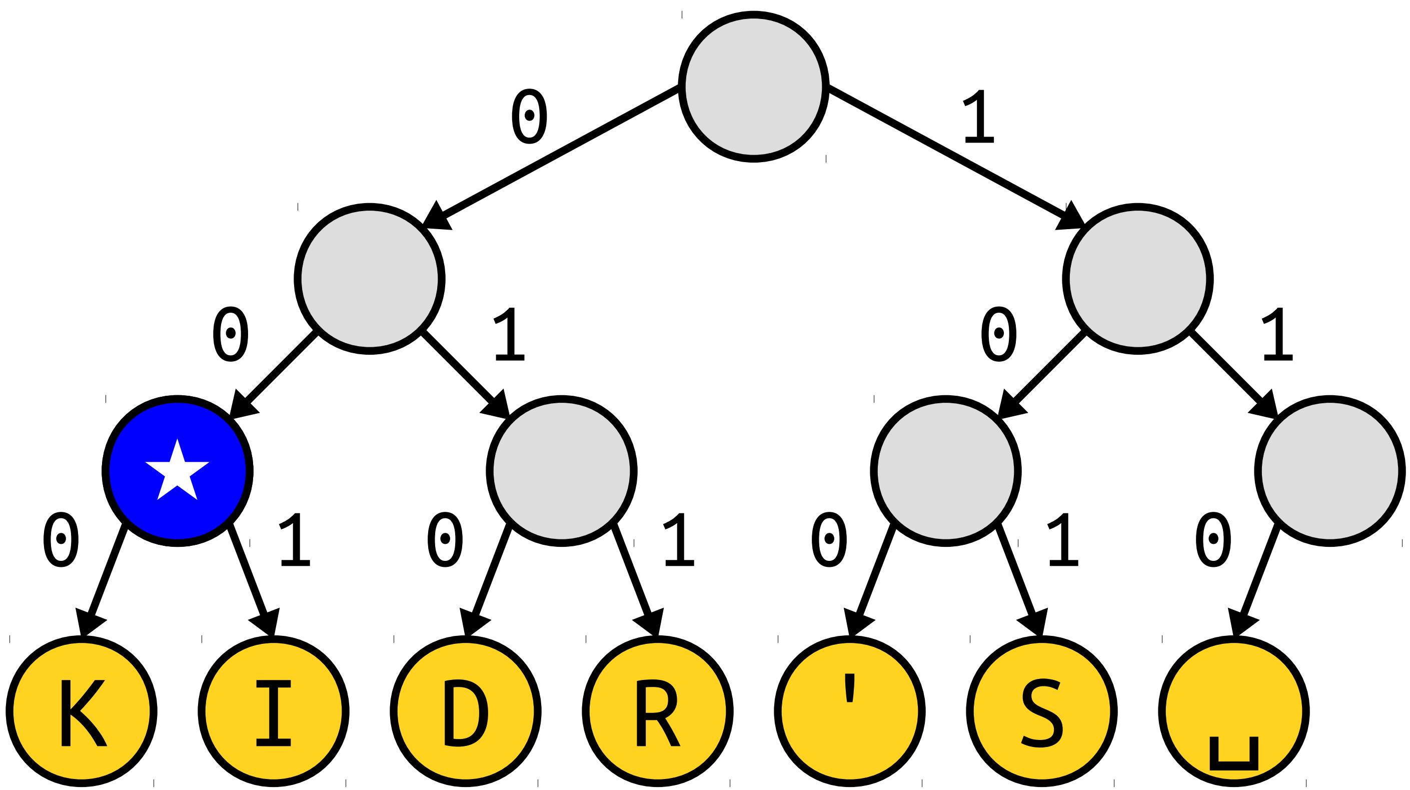
S K 001

What is the title of this slide?

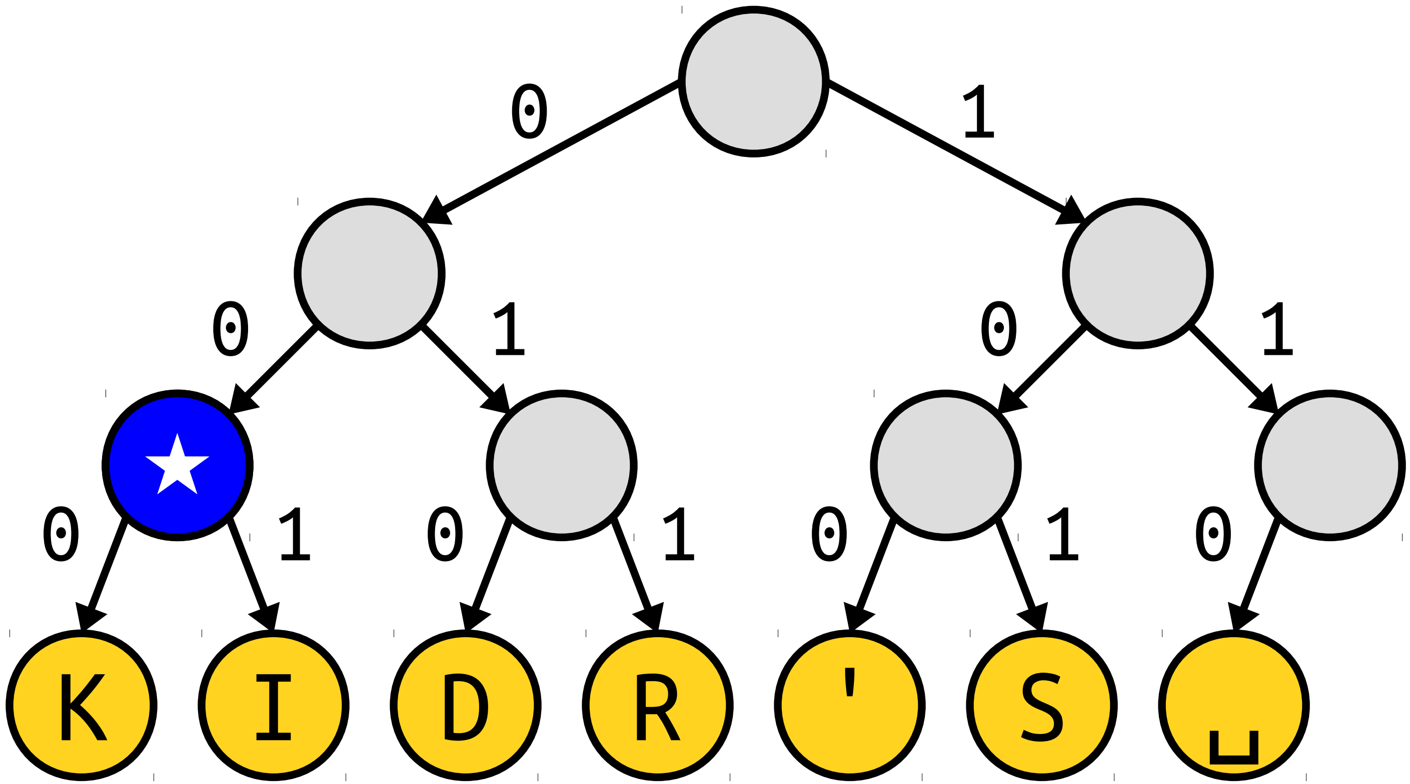


S K 001

What is the title of this slide?

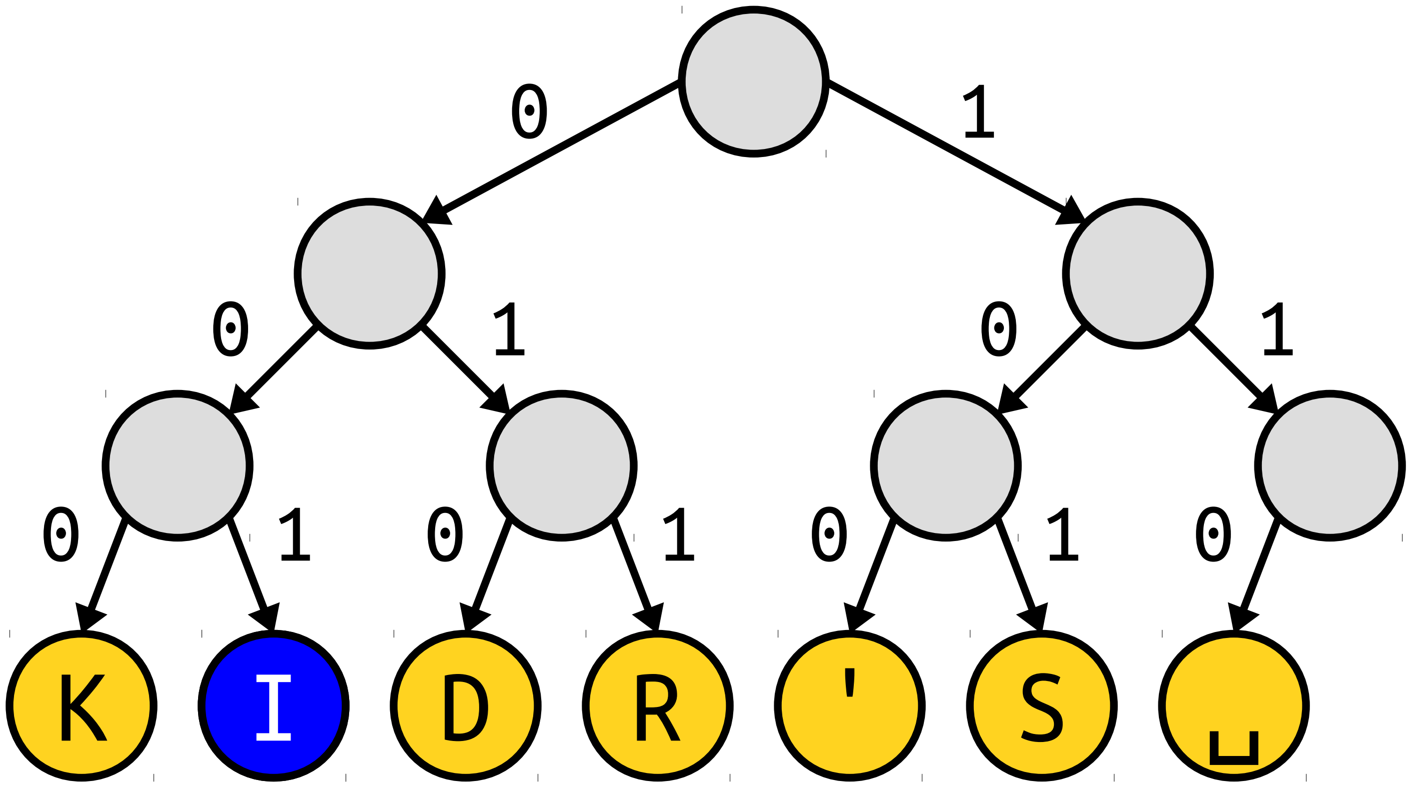


S K 001



What is the title of this slide?

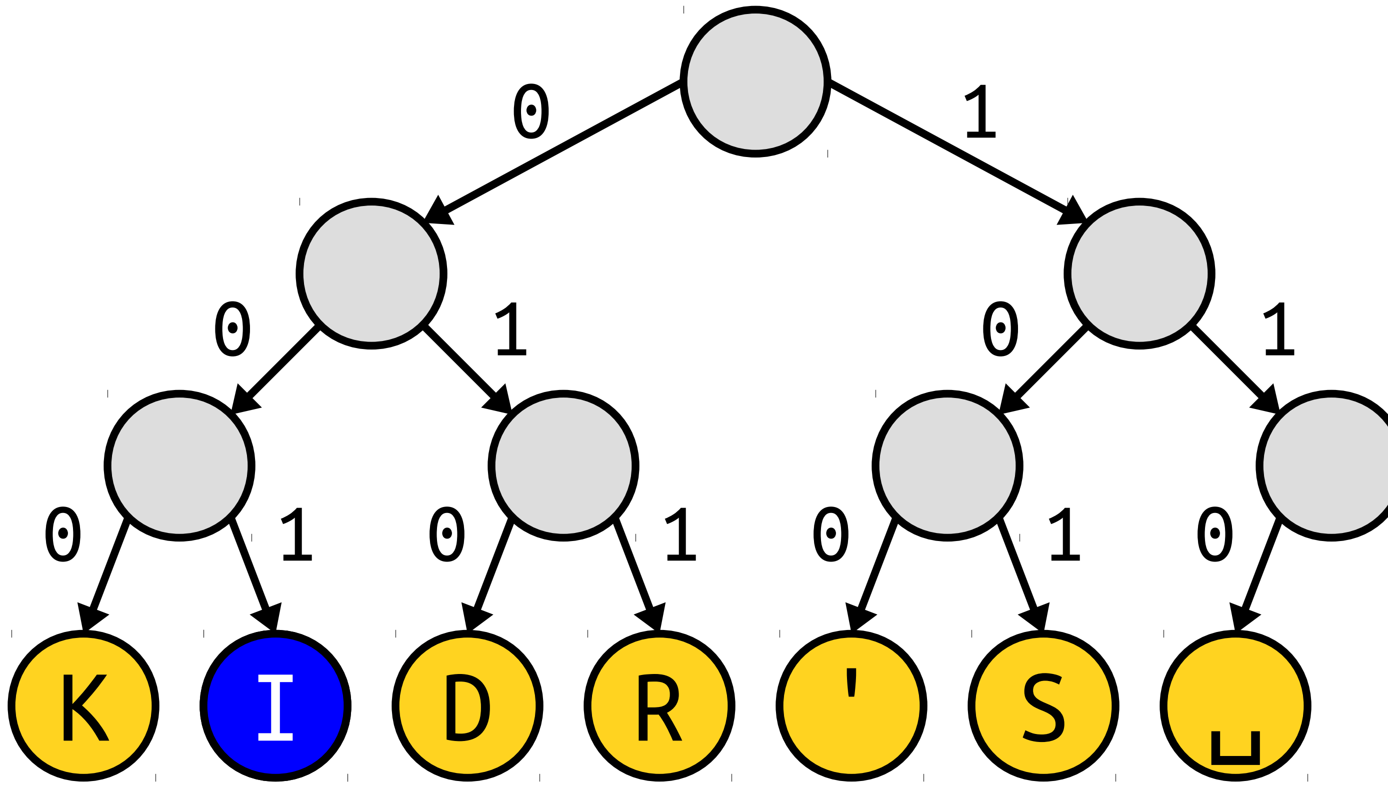
S K 001



What is the title of this slide?

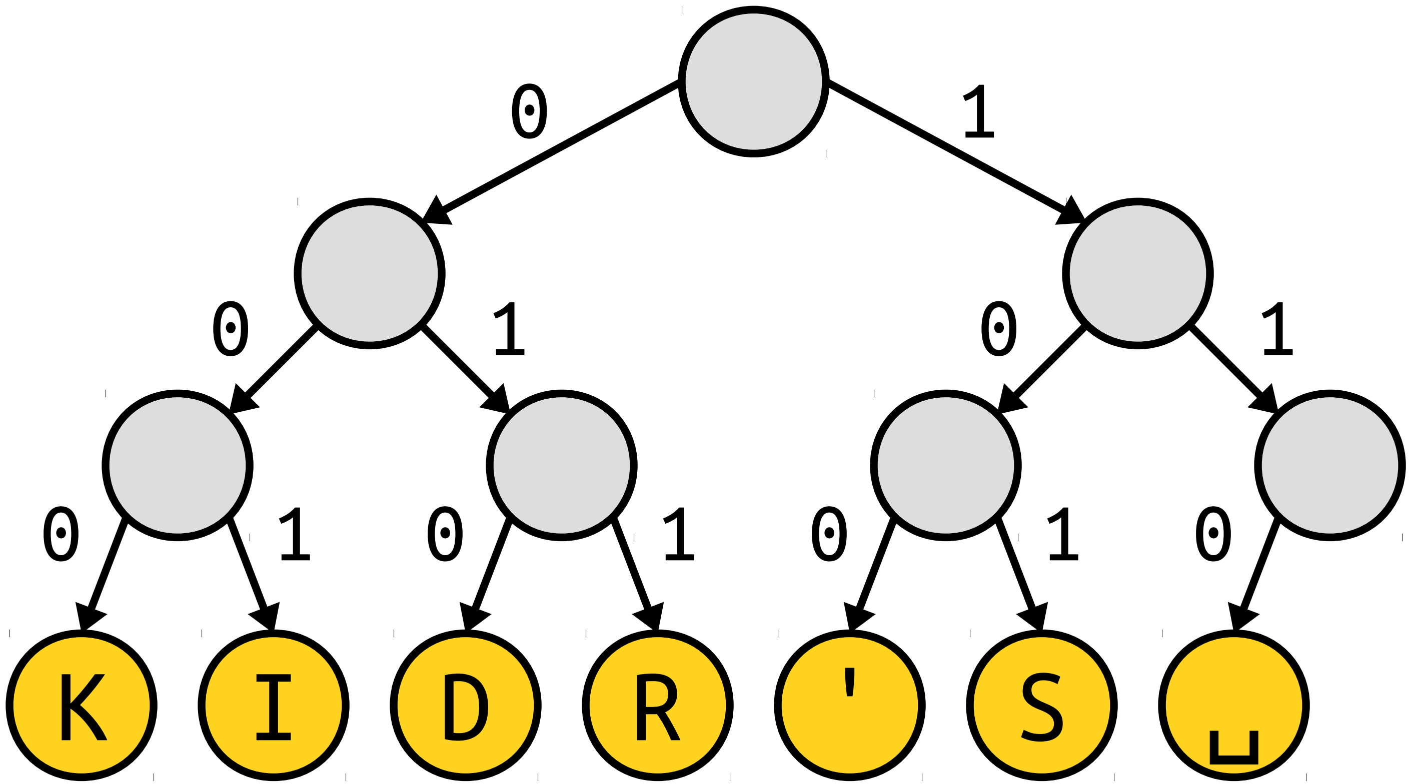
S K I

What is the title of this slide?



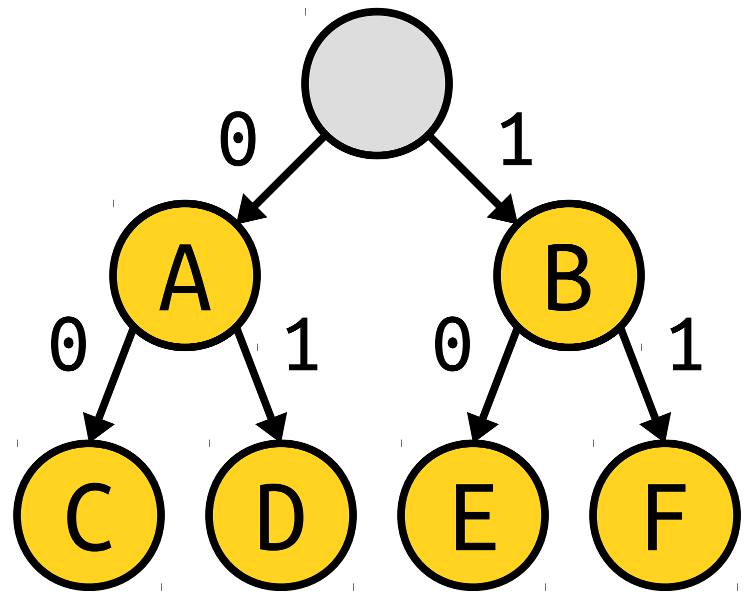
S K I

What is the title of this slide?



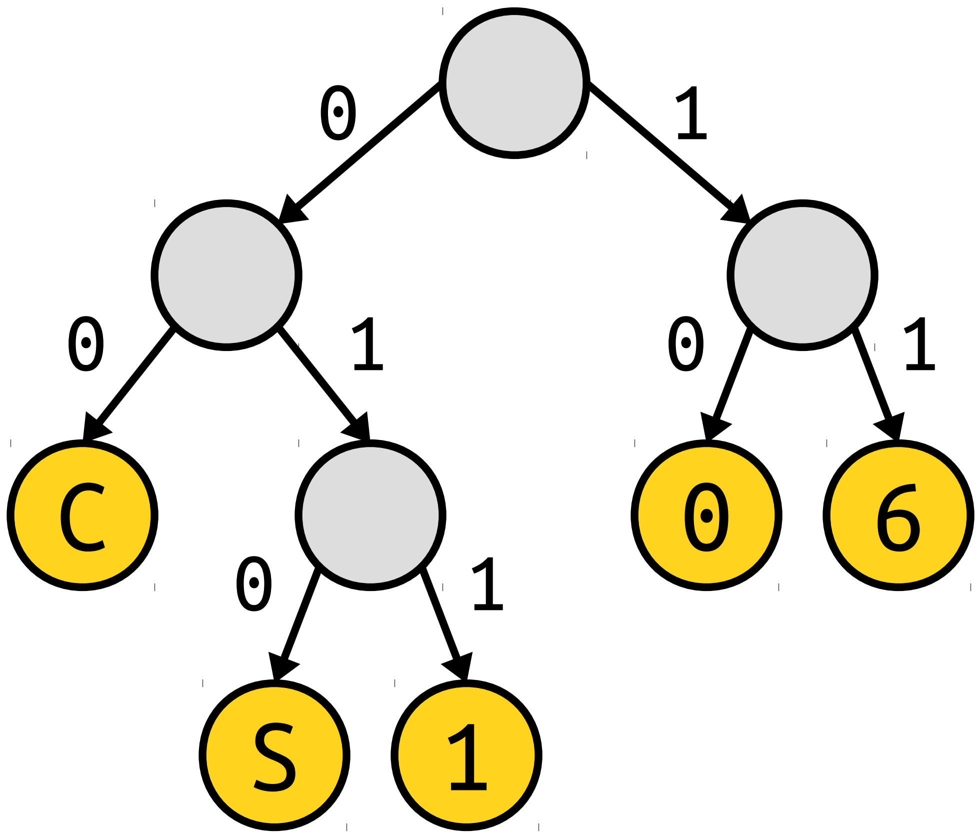
Coding Trees

- Not all binary trees will work as coding trees.
- Why is the one to the right not a valid coding tree?
- ***Answer:*** It doesn't give a prefix code. The code for `A` is a prefix for the codes for `C` and `D`.



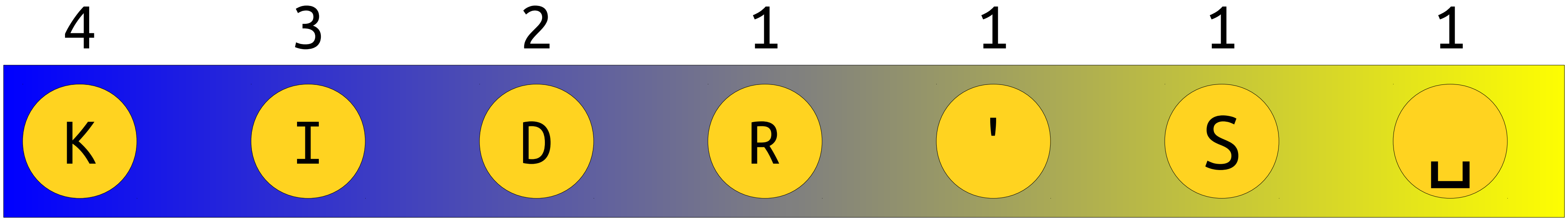
Coding Trees

- A coding tree is valid if all the letters are stored at the ***leaves***, with internal nodes just doing the routing.
- ***Goal:*** Find the best coding tree for a string.

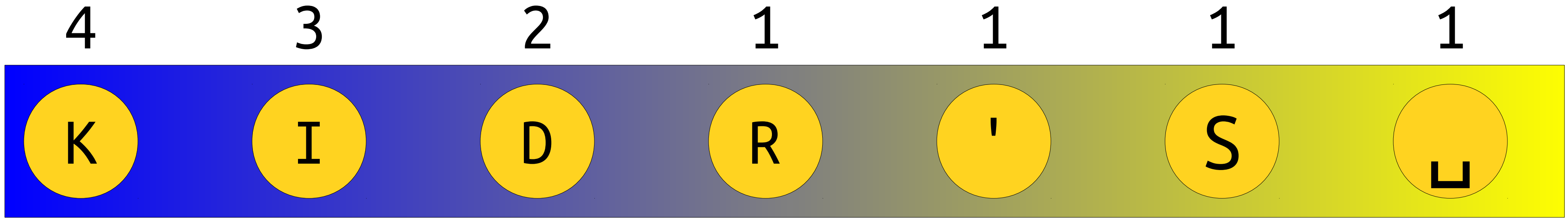


How do we find the best binary tree with this property?

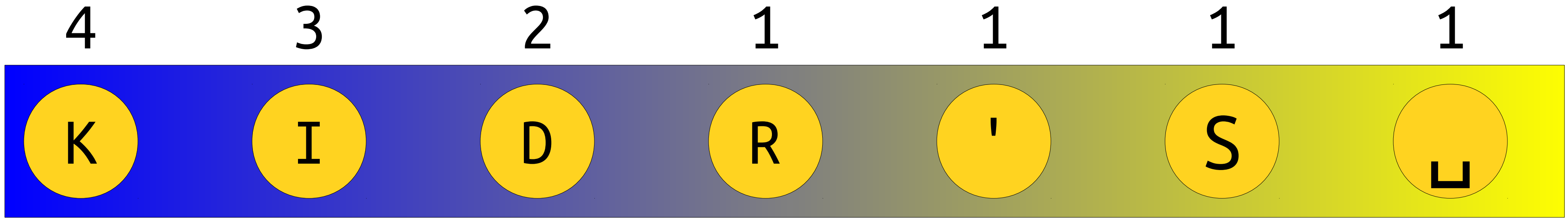
Huffman Coding

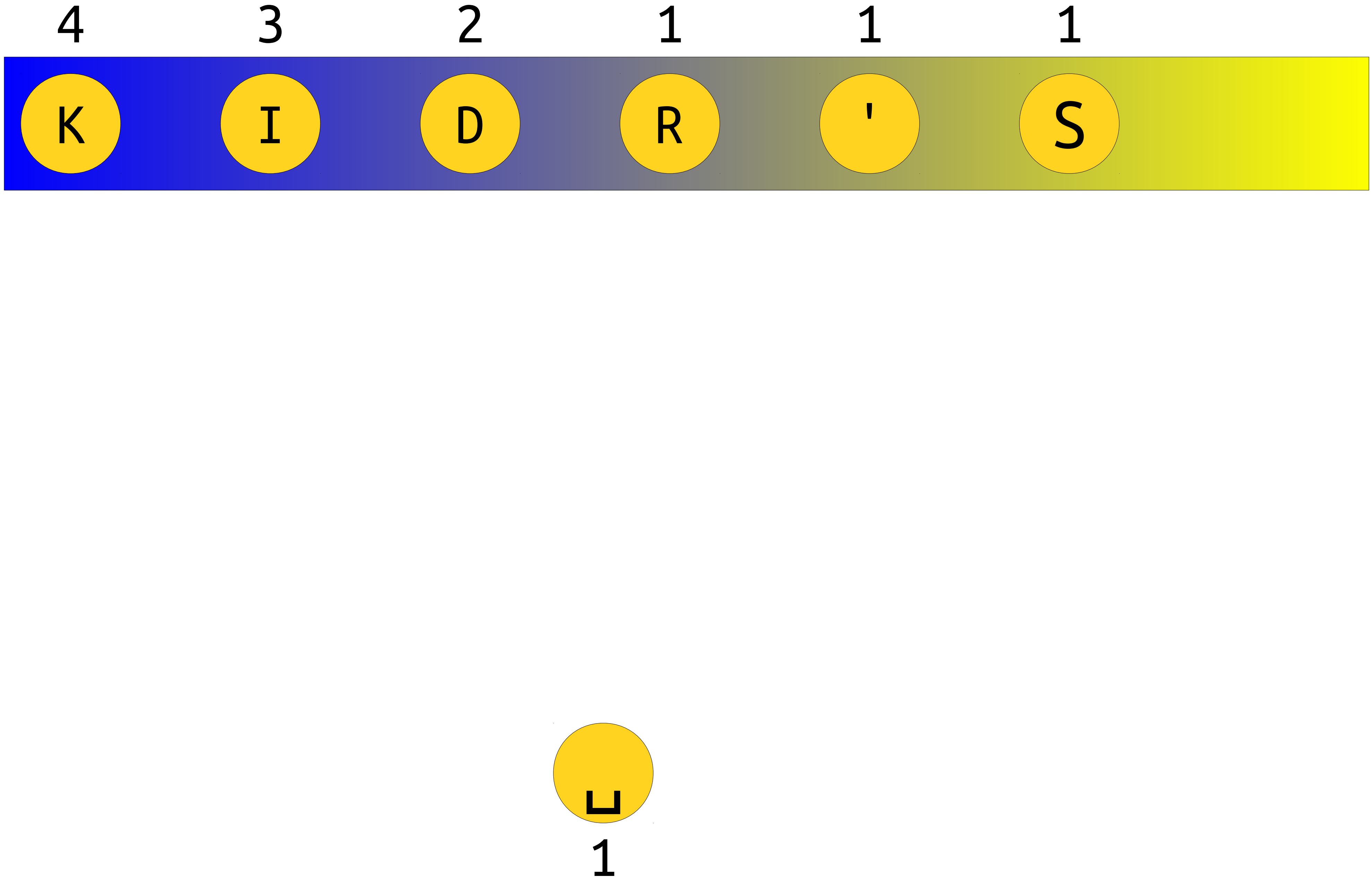


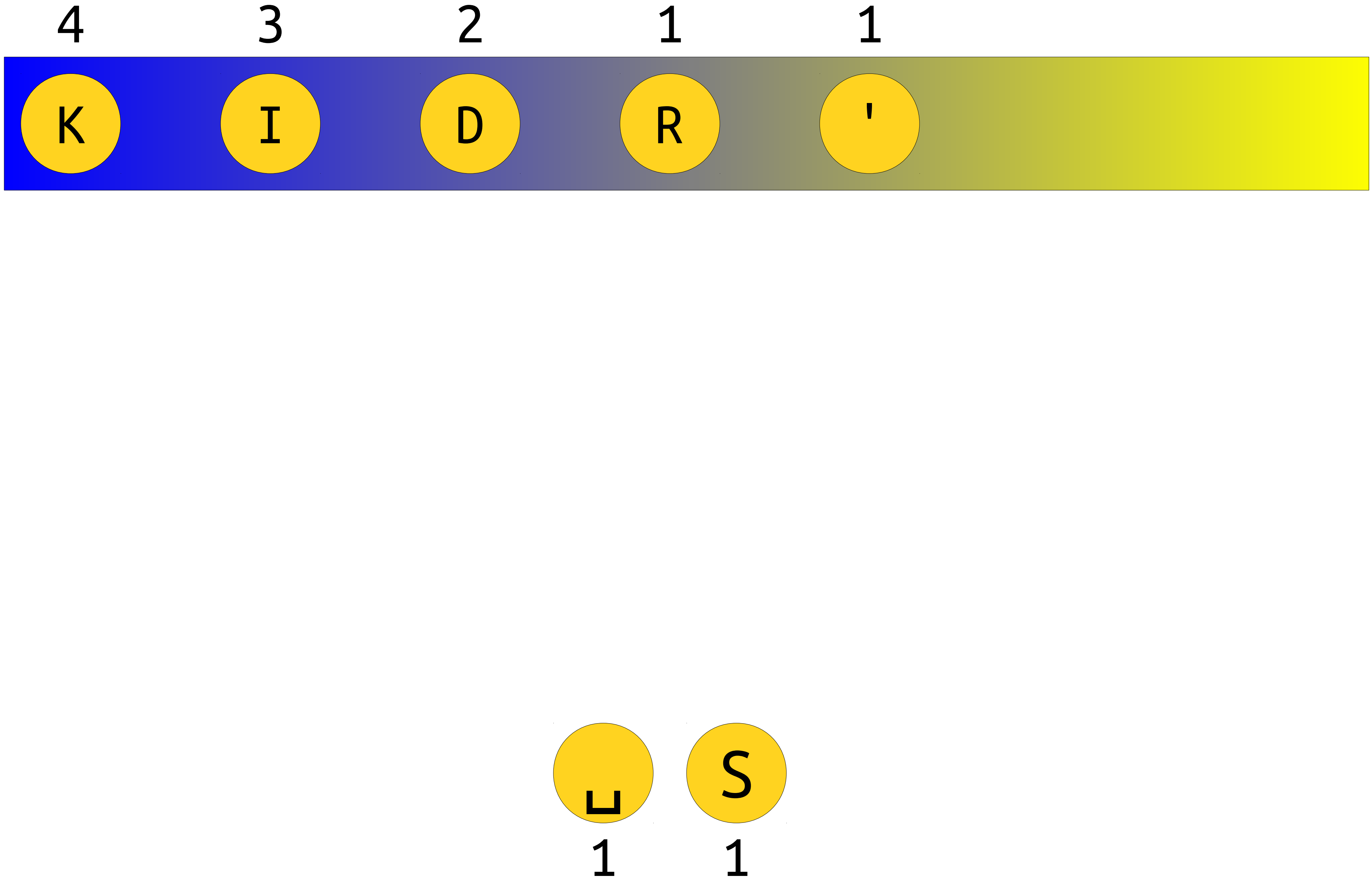
character	*frequency*
K	4
I	3
D	2
R	1
'	1
S	1
	1



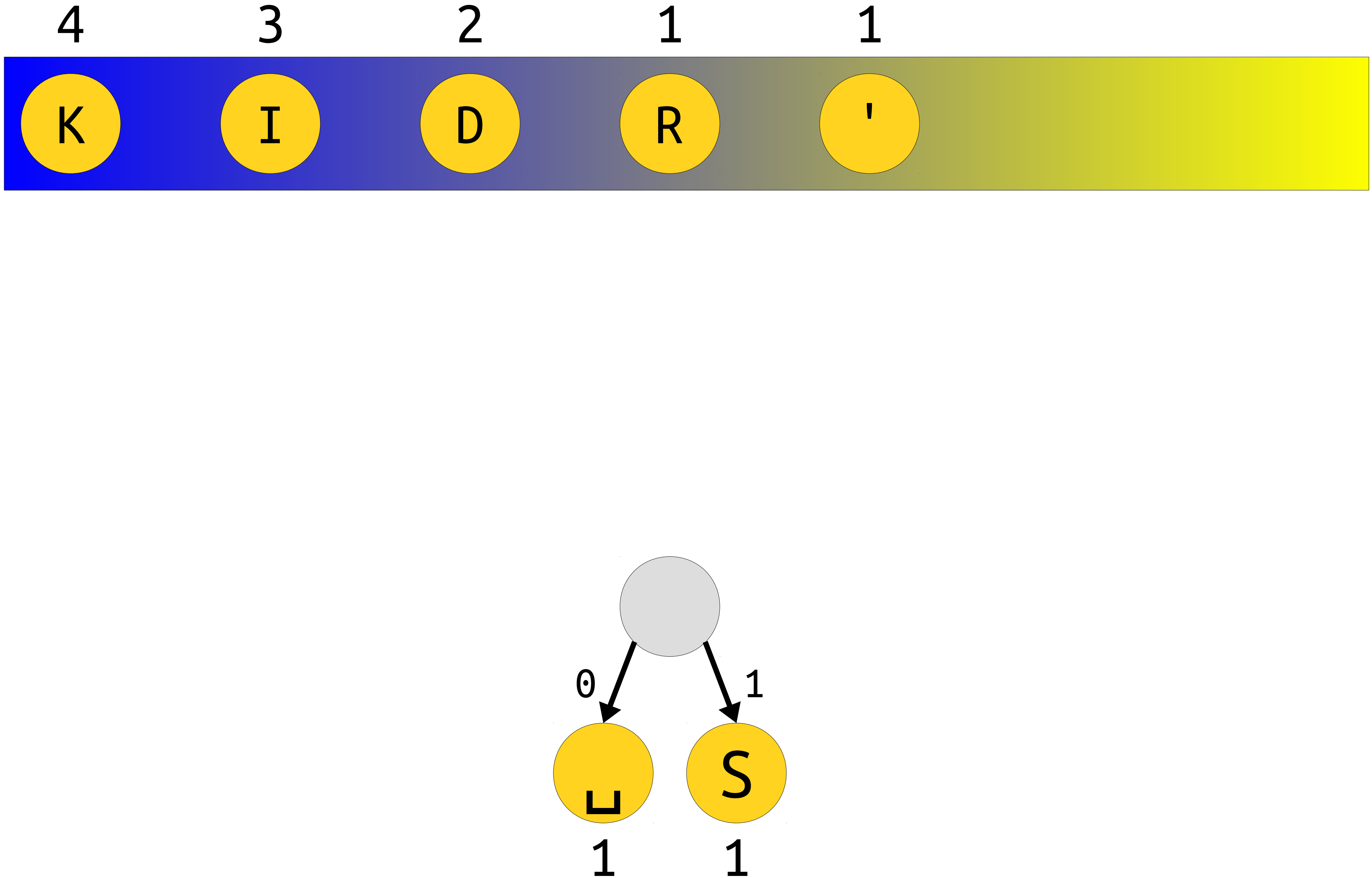
Right now, we have all the leaves of the tree. We now need to build the tree around them.

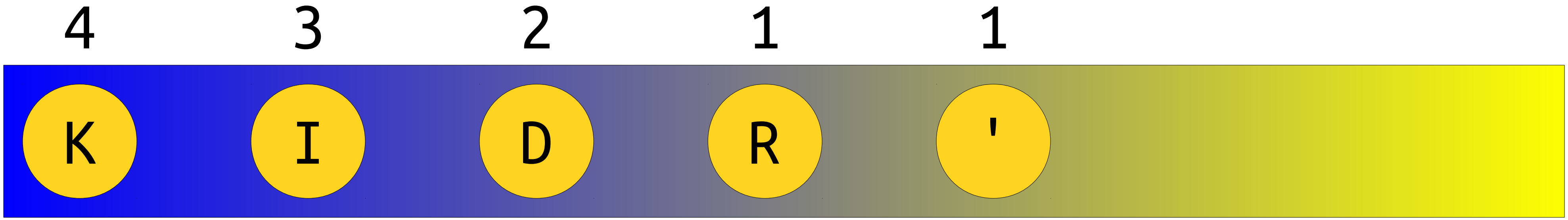

4 3 2 1 1 1 1
K I D R ' S ␣

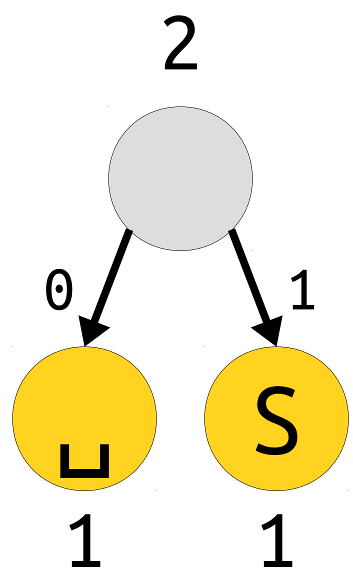

4
K
3
I
2
D
1
R
1
'
1
S
␣
1

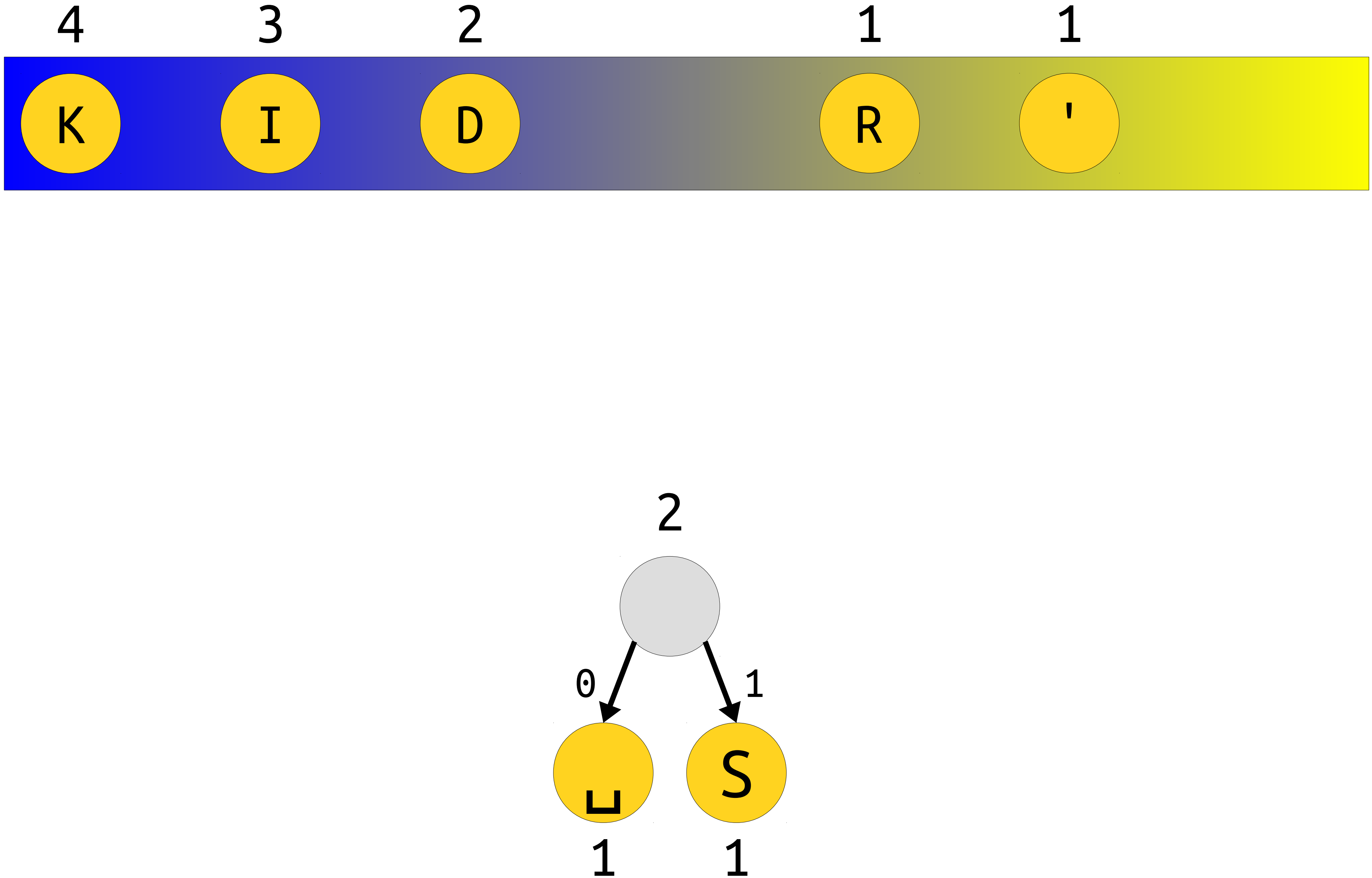


4
3
2
1
1
K
I
D
R
'
␣
S
1
1

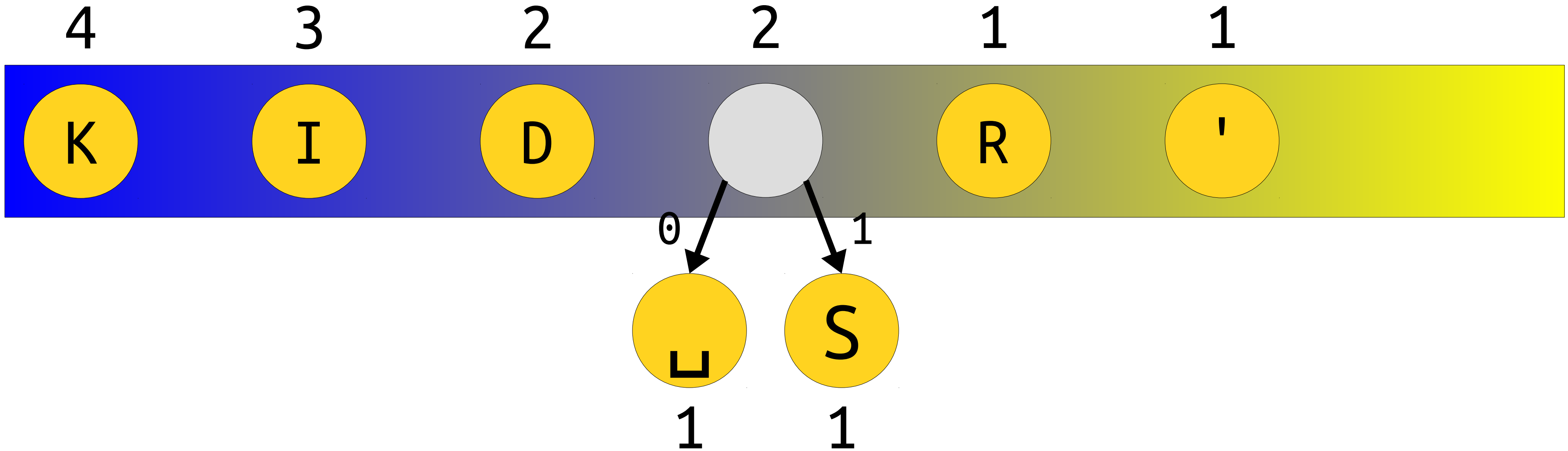

4
3
2
1
1
K
I
D
R
'
0
1
␣
S
1
1


4
3
2
1
1
K
I
D
R
'

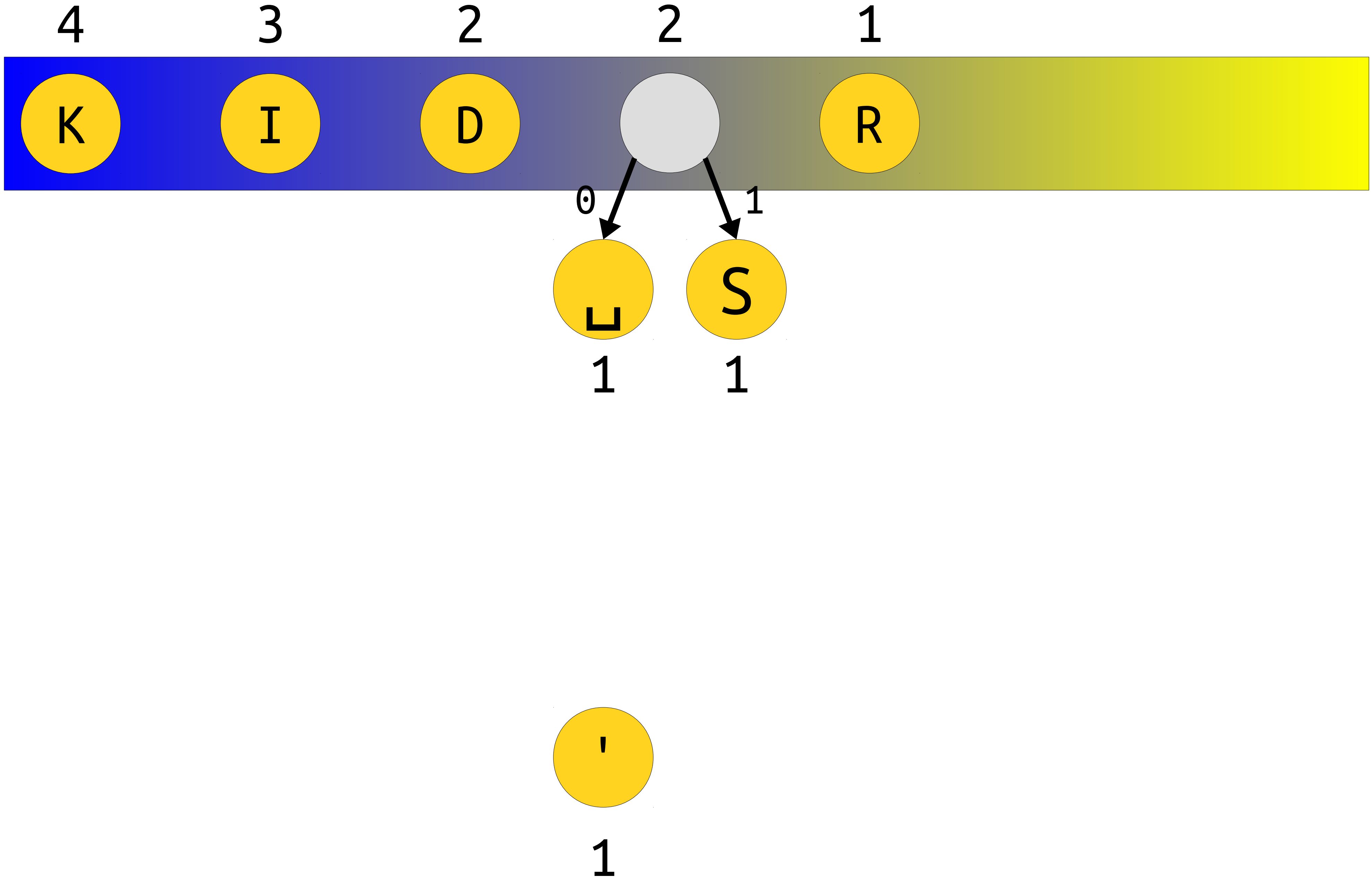

2
0
1
␣
S
1
1

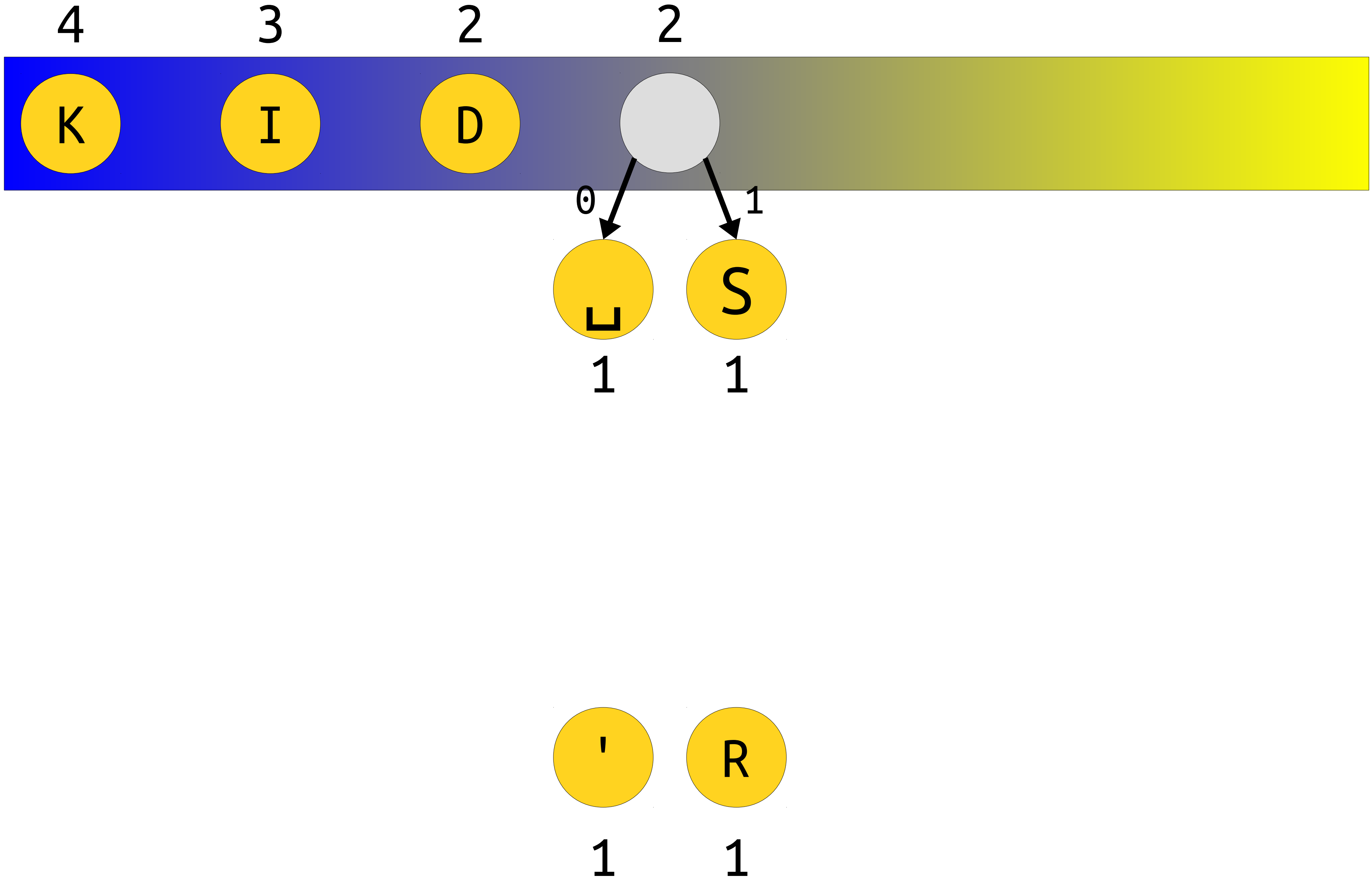


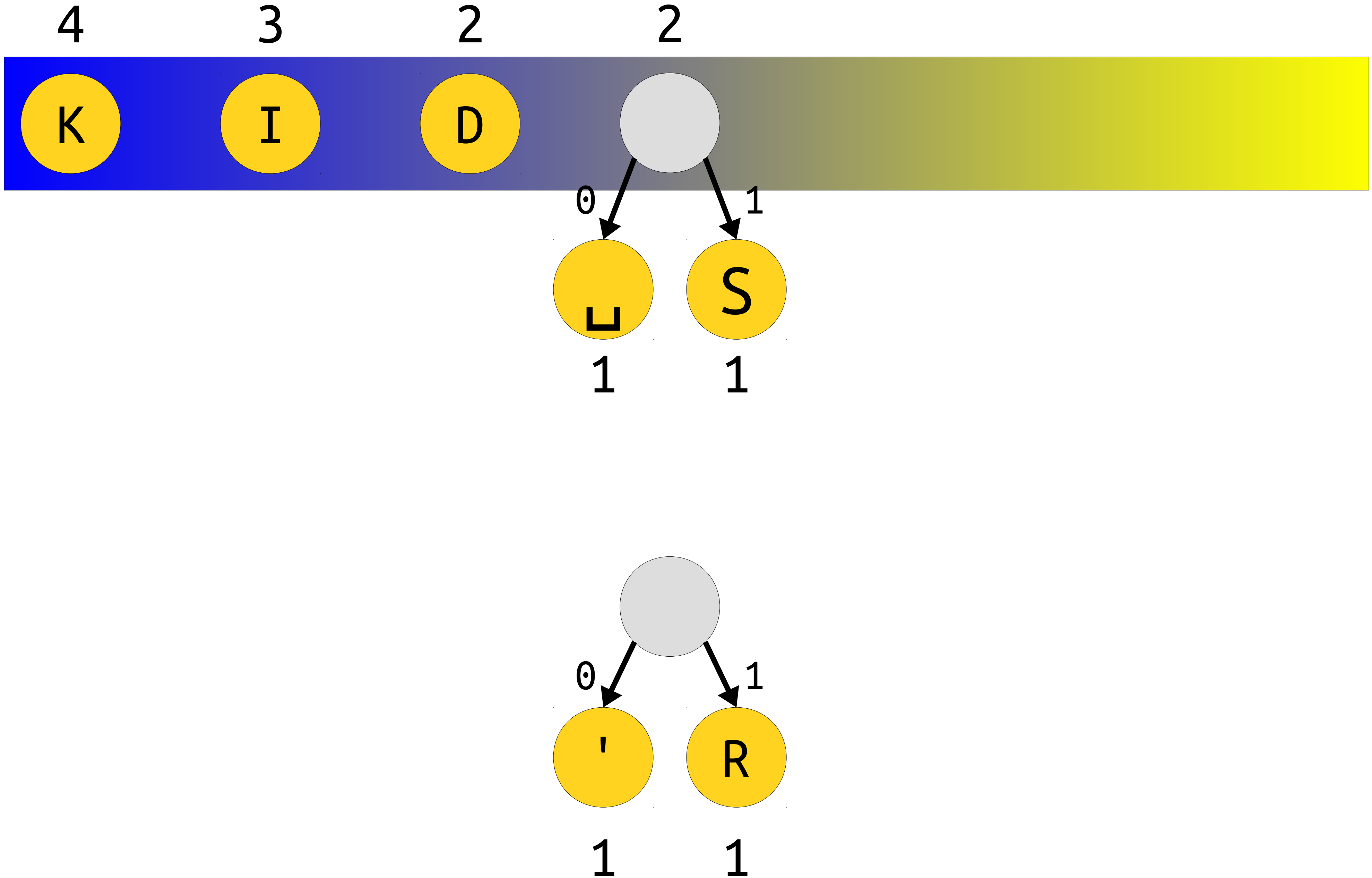
4
3
2
1
1
K
I
D
R
'
2
0
1
␣
S
1
1



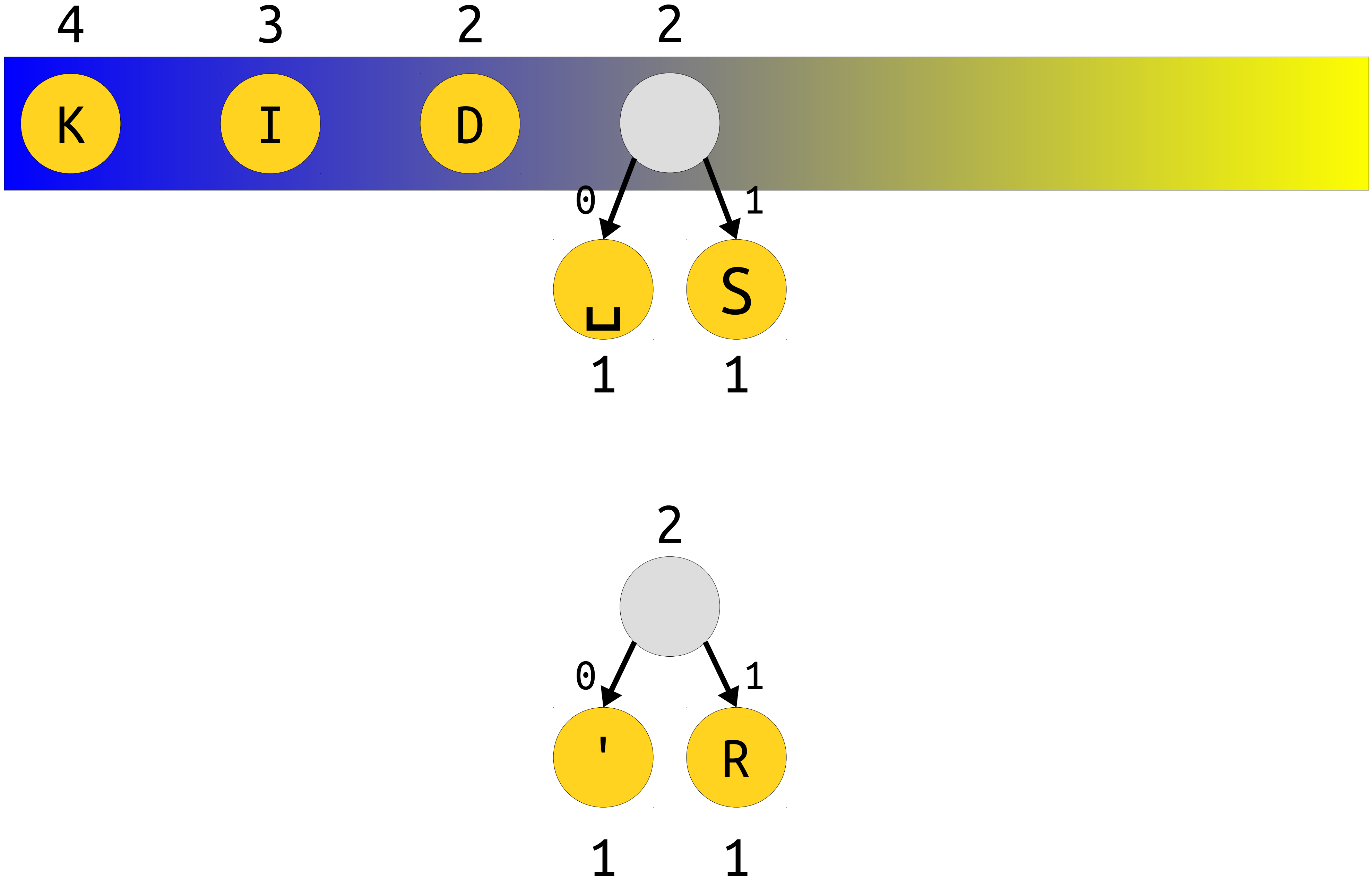
4
3
2
2
1
1
K
I
D
R
'
0
1
␣
S
1
1

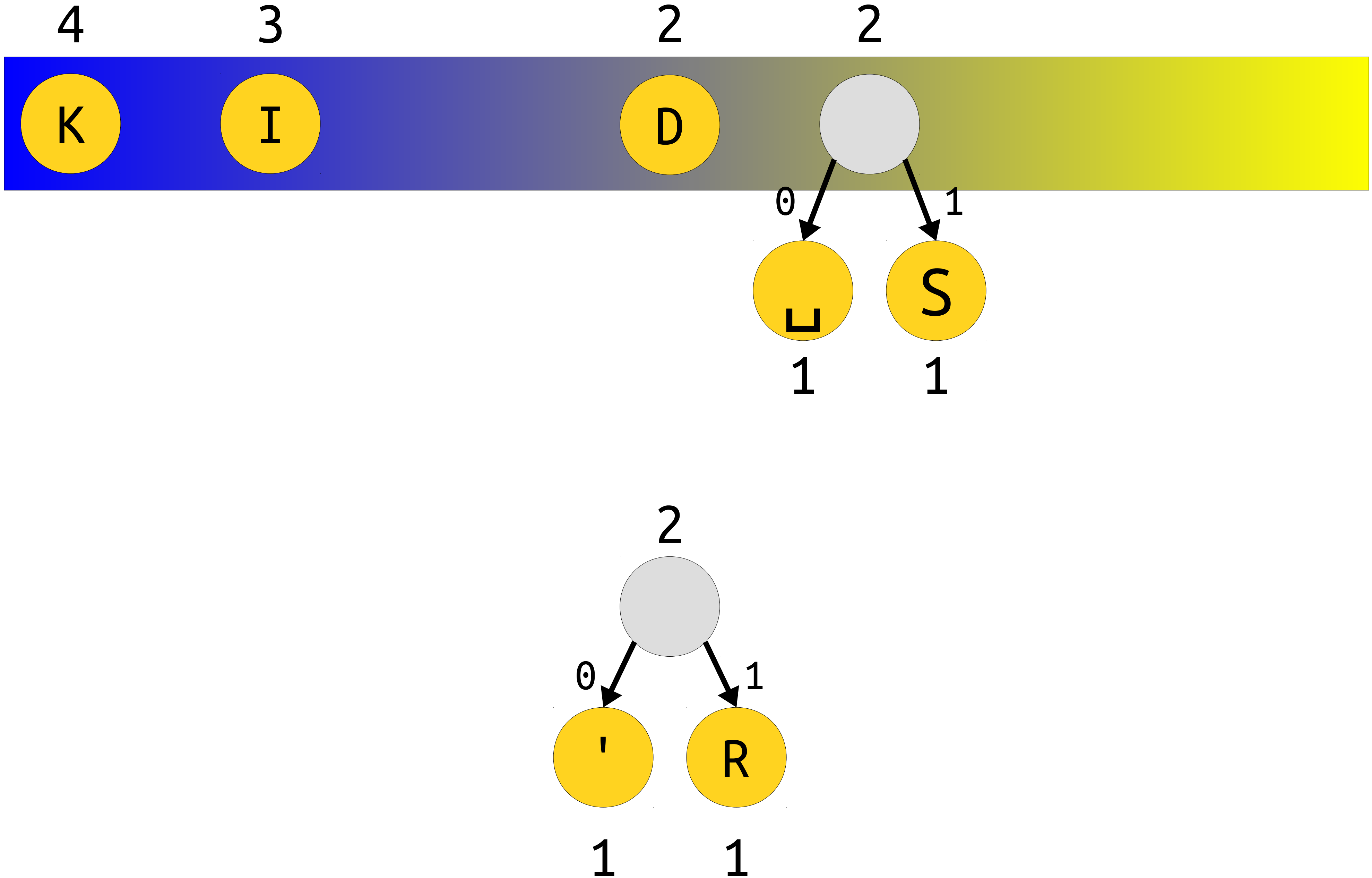

4
3
2
2
1
K
I
D
R
0
1
␣
S
1
1
'
1

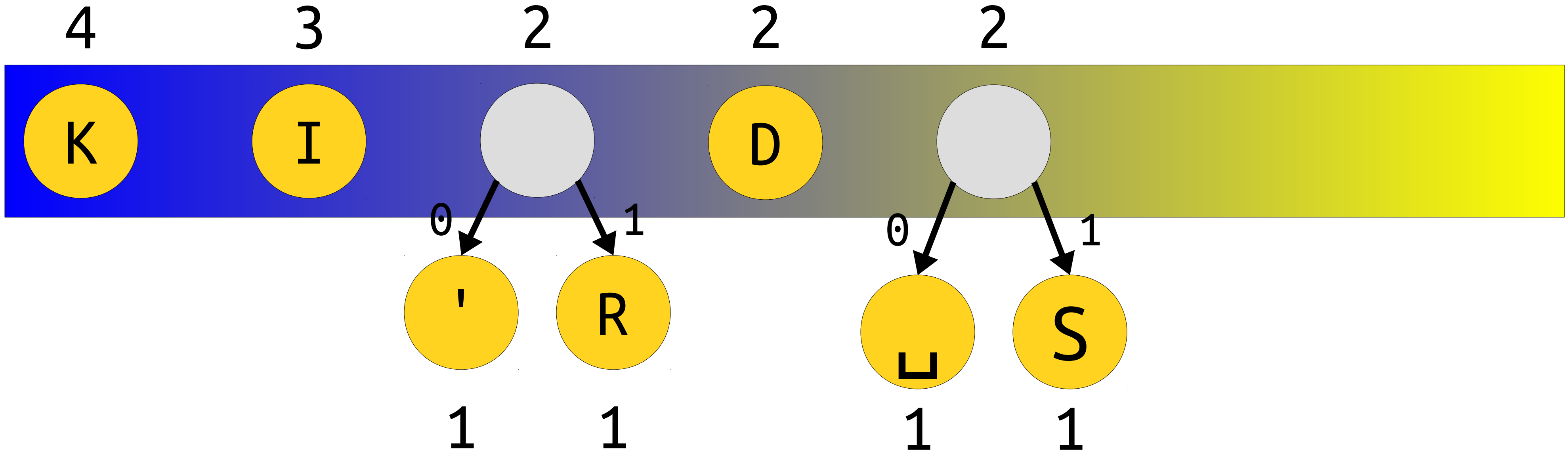

4
3
2
2
K
I
D
0
1
␣
S
1
1
'
R
1
1

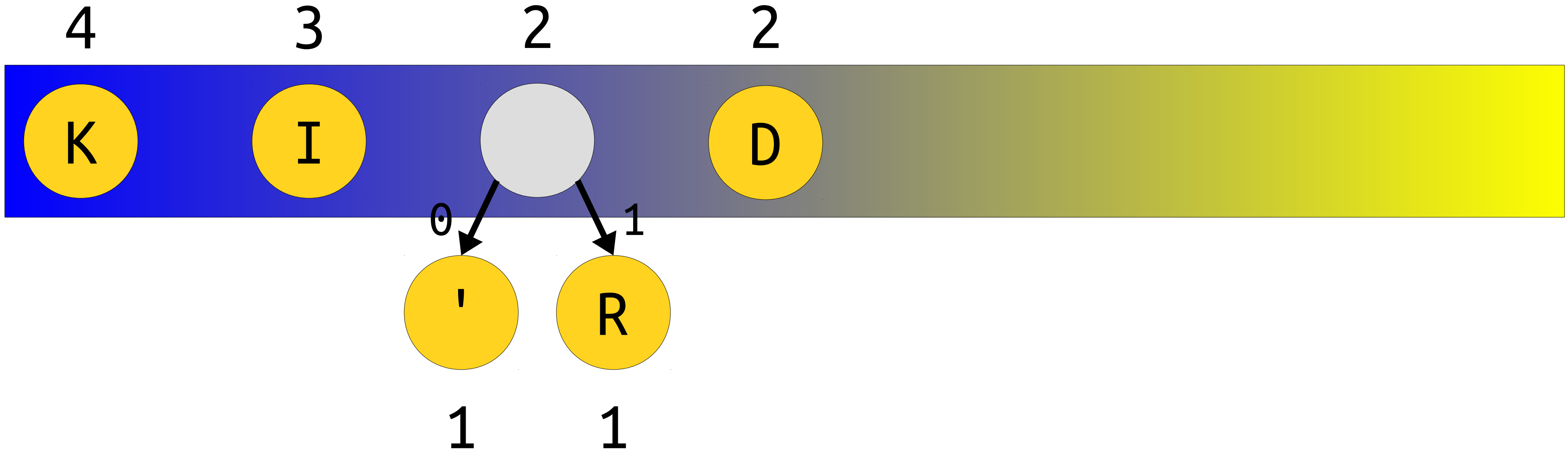


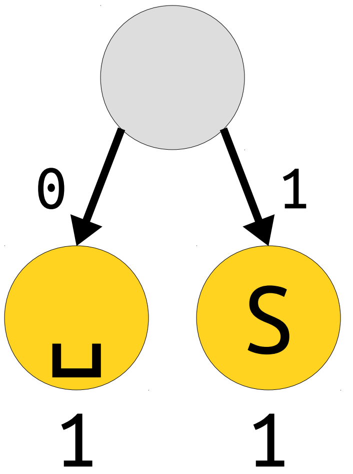
4
3
2
2
K
I
D
0
1
␣
S
1
1
0
1
'
R
1
1

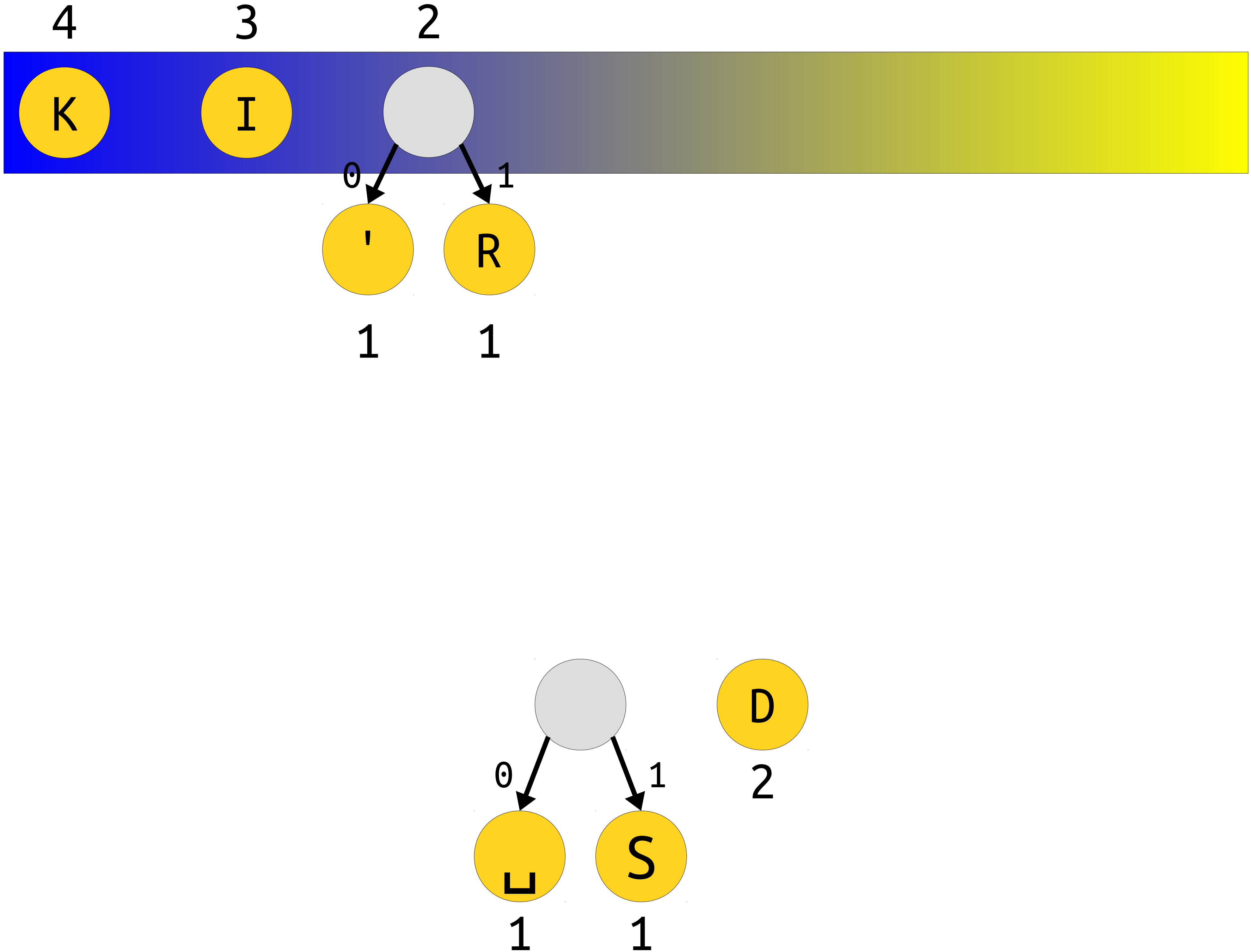

4
K
3
I
2
D
2
0
1
␣
S
1
1
2
0
1
'
R
1
1


4
3
2
2
K
I
D
0
1
␣
S
1
1
2
0
1
'
R
1
1

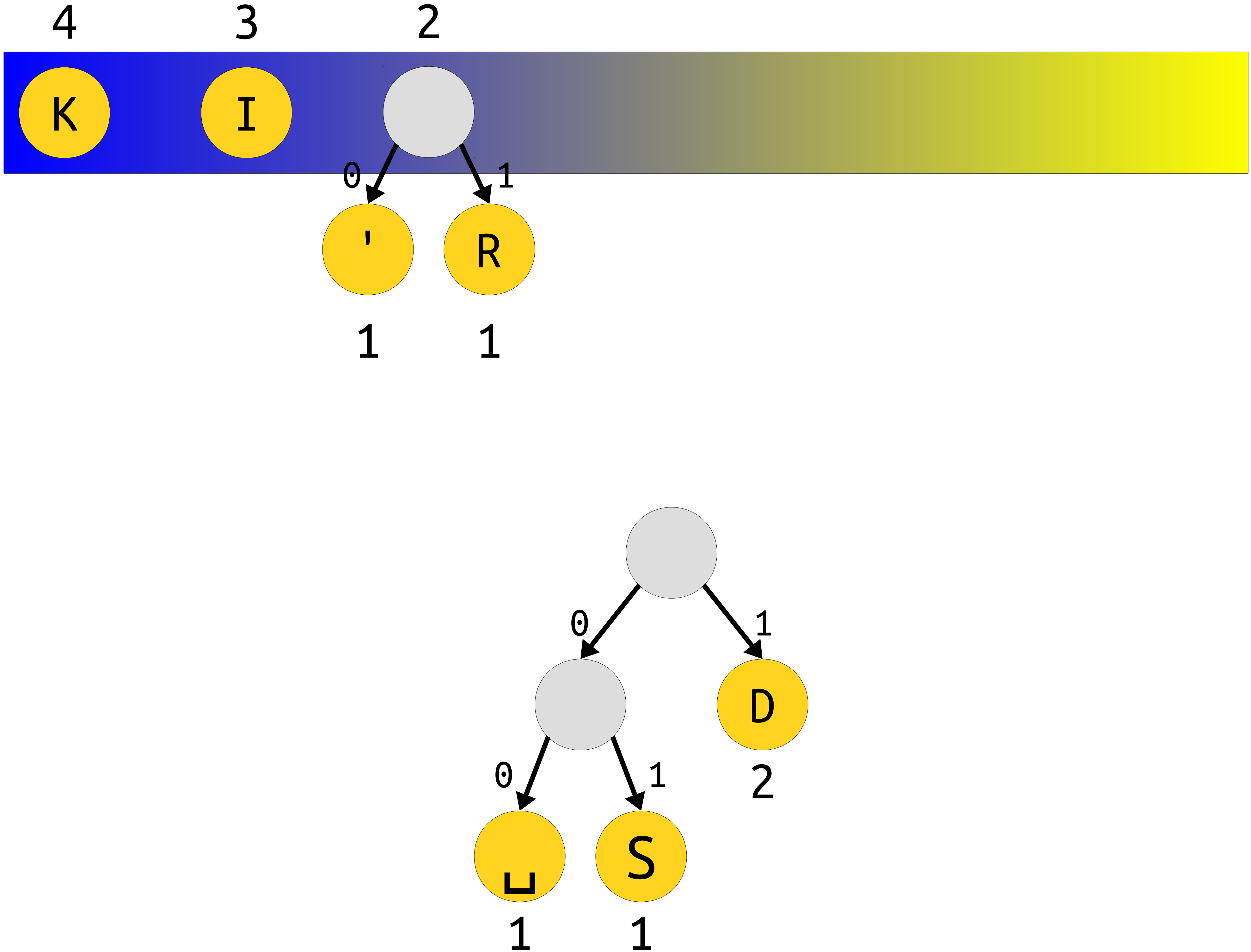

4
3
2
2
2
K
I
D
0
1
0
1
'
R
␣
S
1
1
1
1

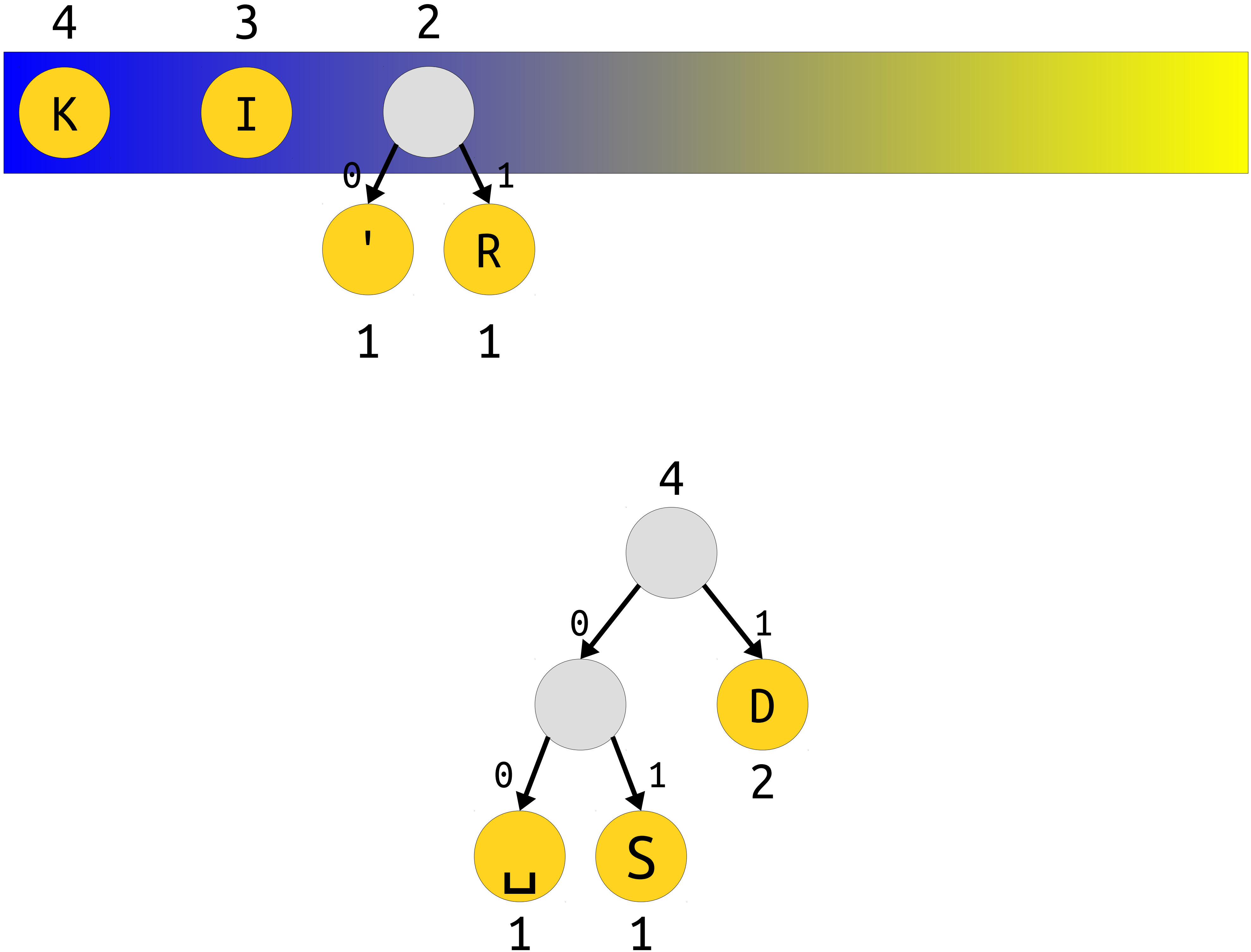

4
3
2
2
K
I
D
0
1
'
R
1
1


0
1
␣
S
1
1

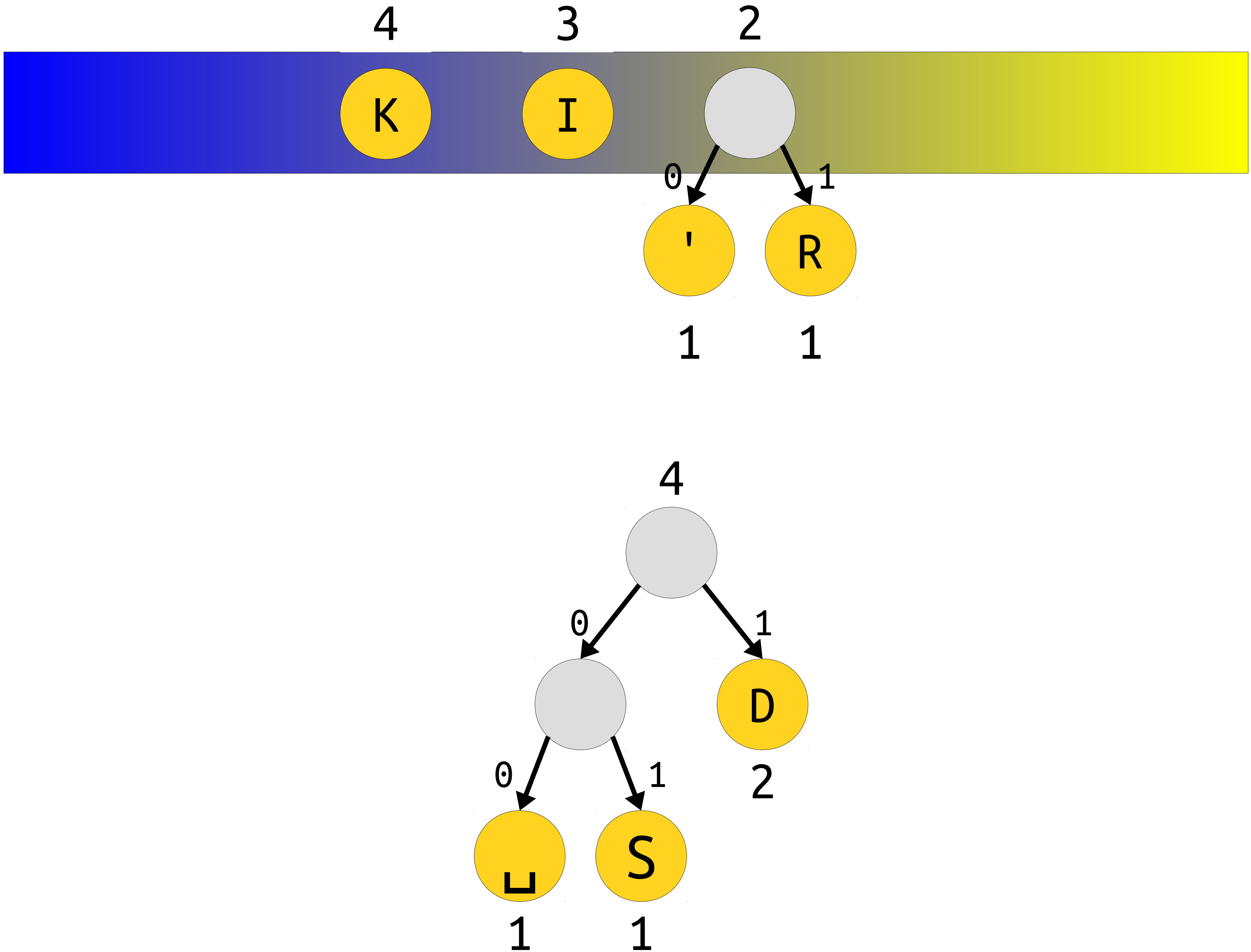


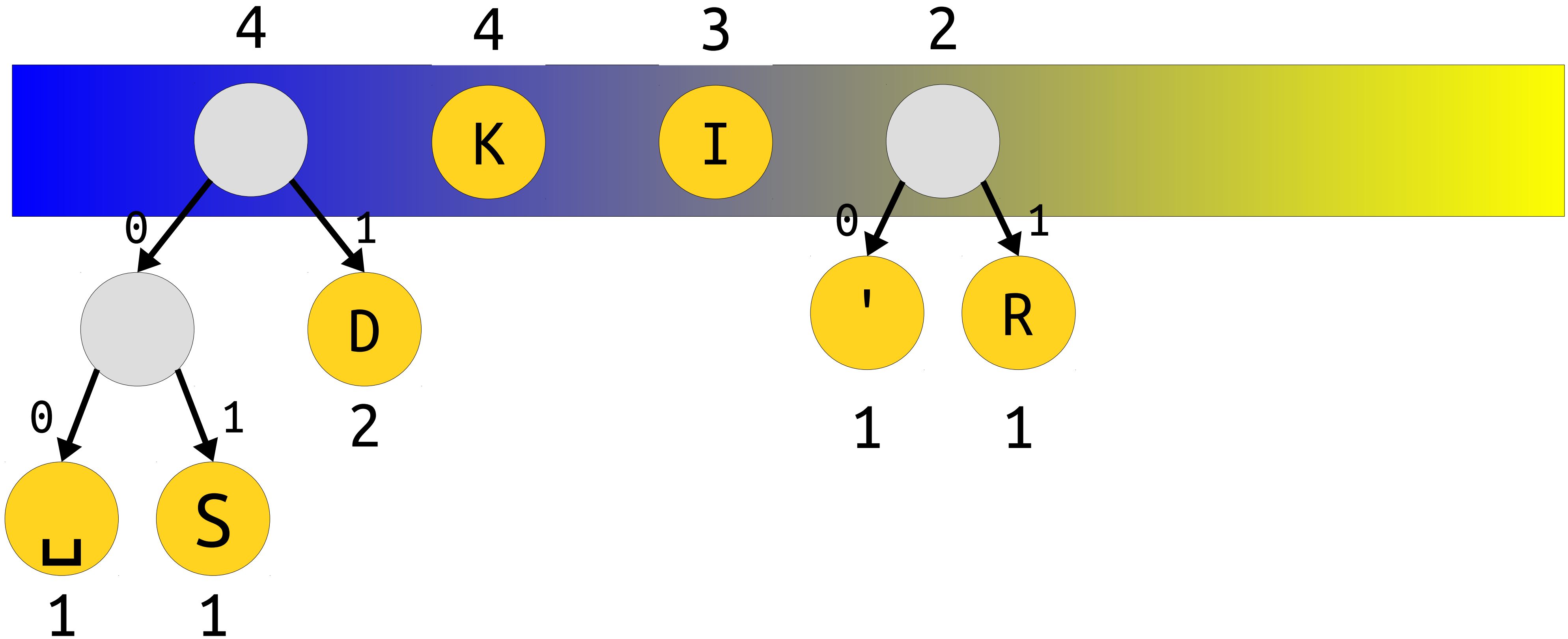
4
3
2
K
I
0
1
'
R
1
1
0
1
␣
S
D
2
1
1

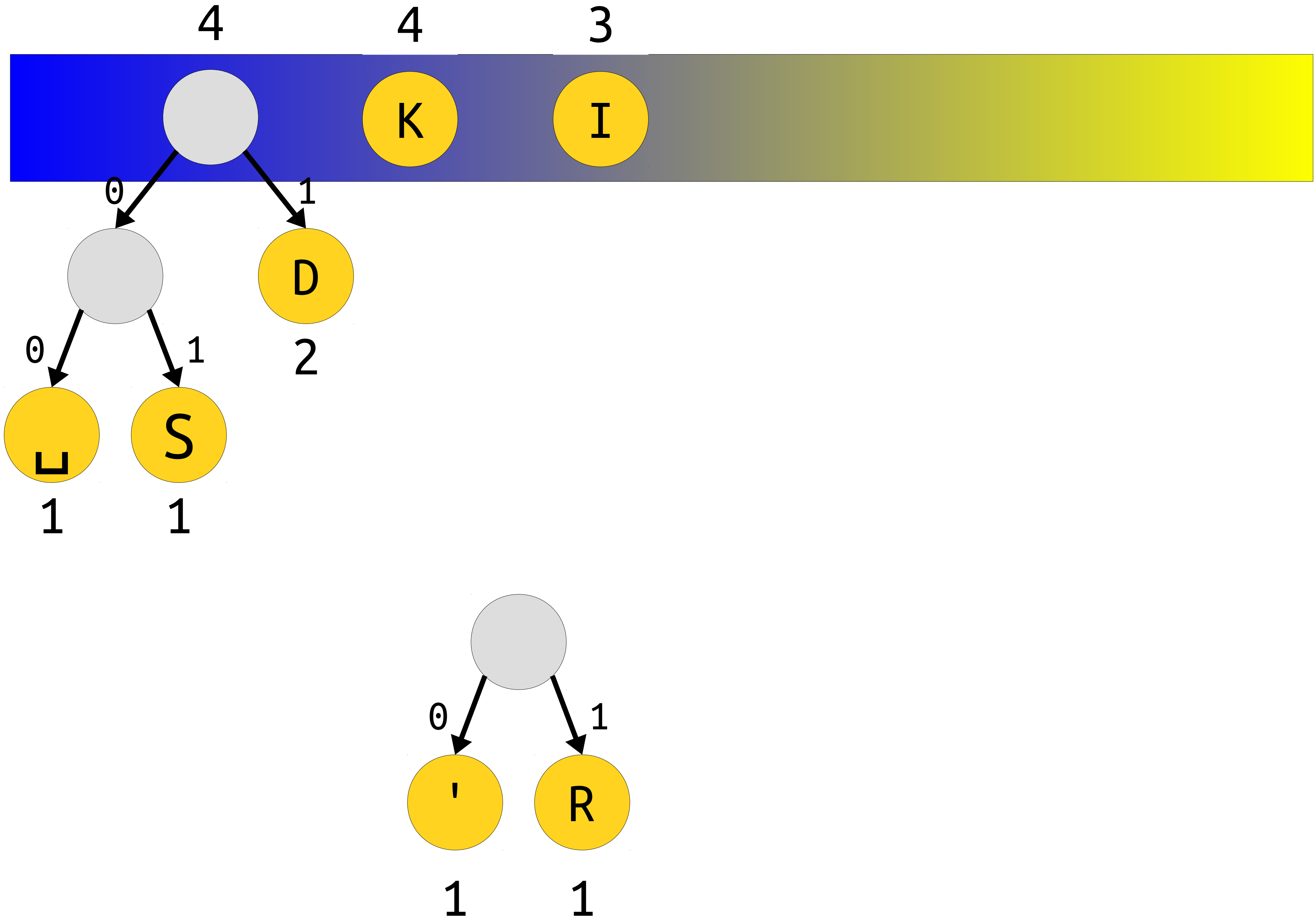

4
3
2
K
I
0
1
'
R
1
1
0
1
D
2
0
1
␣
S
1
1



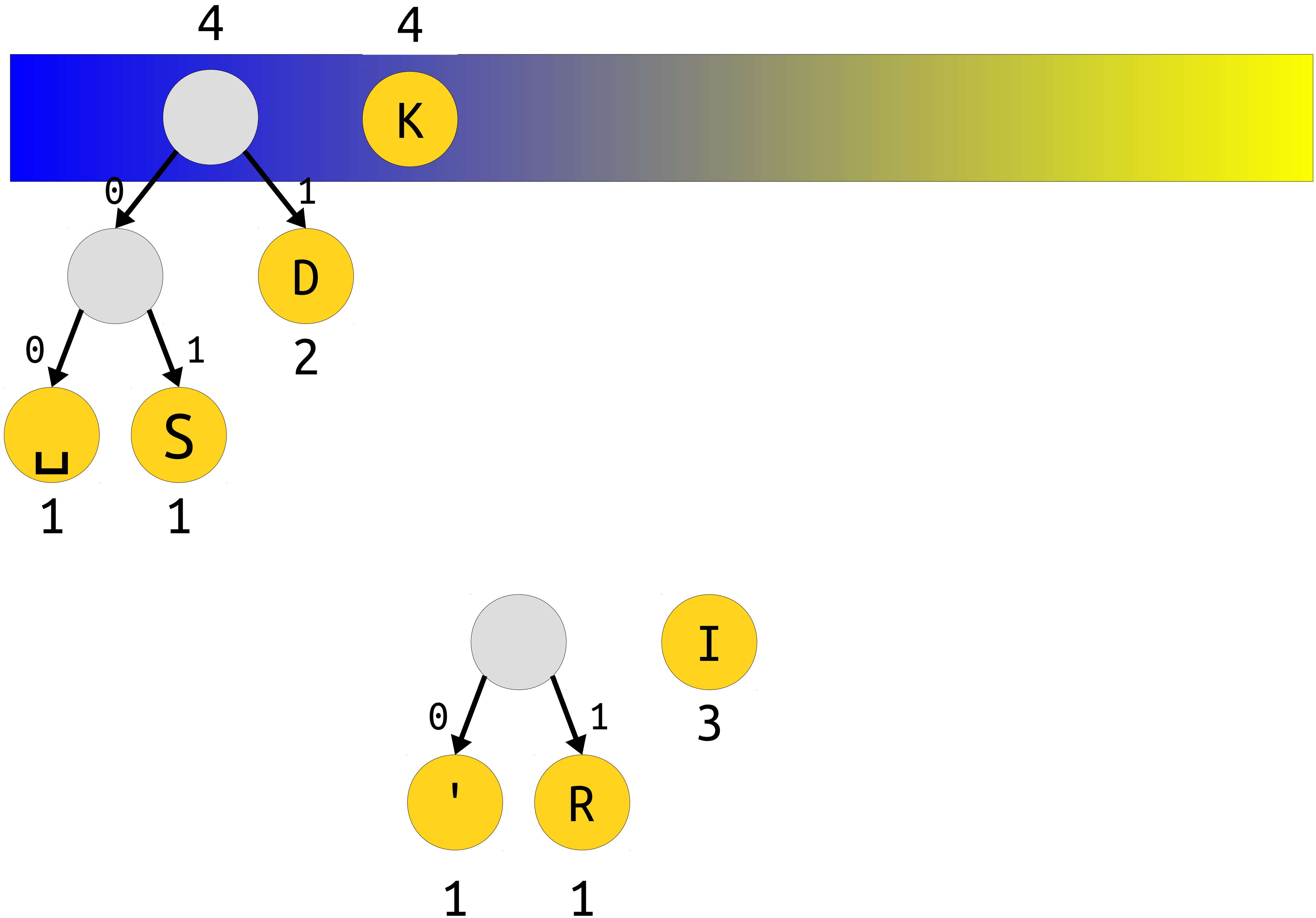
4
3
2
K
I
0
1
'
R
1
1
4
0
1
D
2
0
1
␣
S
1
1

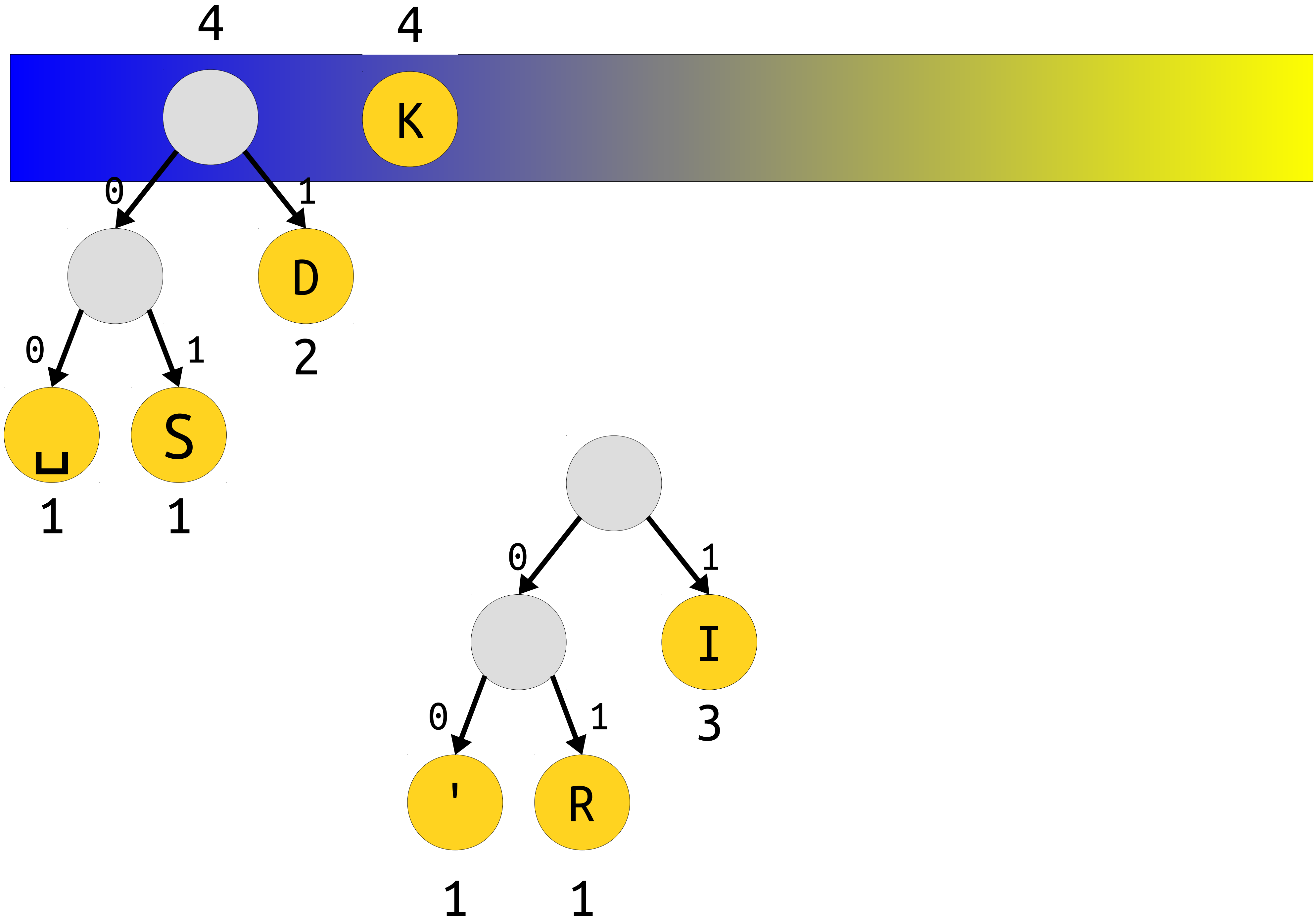

4
3
2
K
I
0
1
'
R
1
1
4
0
1
D
0
1
2
␣
S
1
1


4
4
3
2
K
I
0
1
0
1
D
'
R
0
1
2
1
1
␣
S
1
1

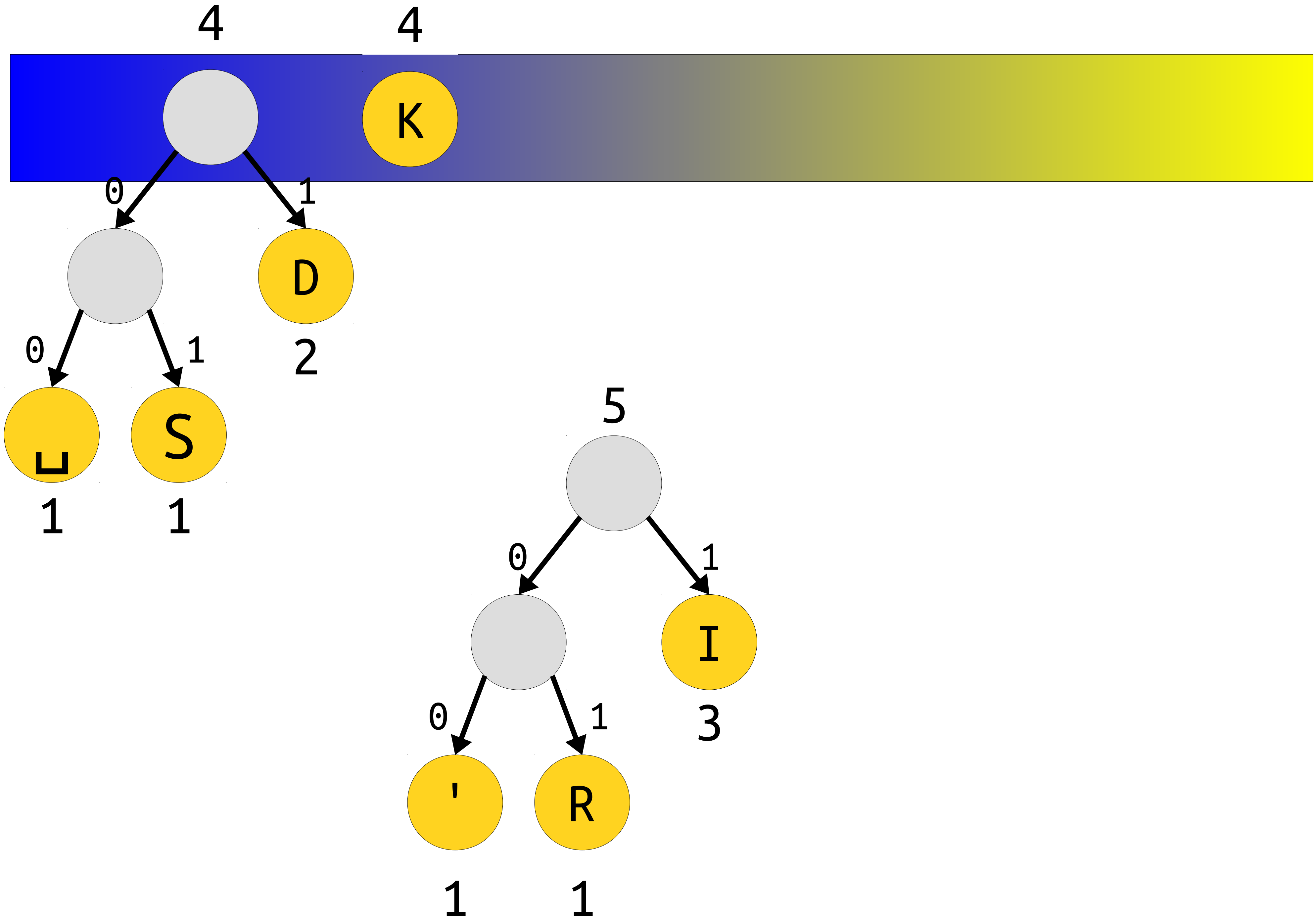


4
4
3
K
I
0
1
D
2
0
1
␣
S
1
1
0
1
'
R
1
1

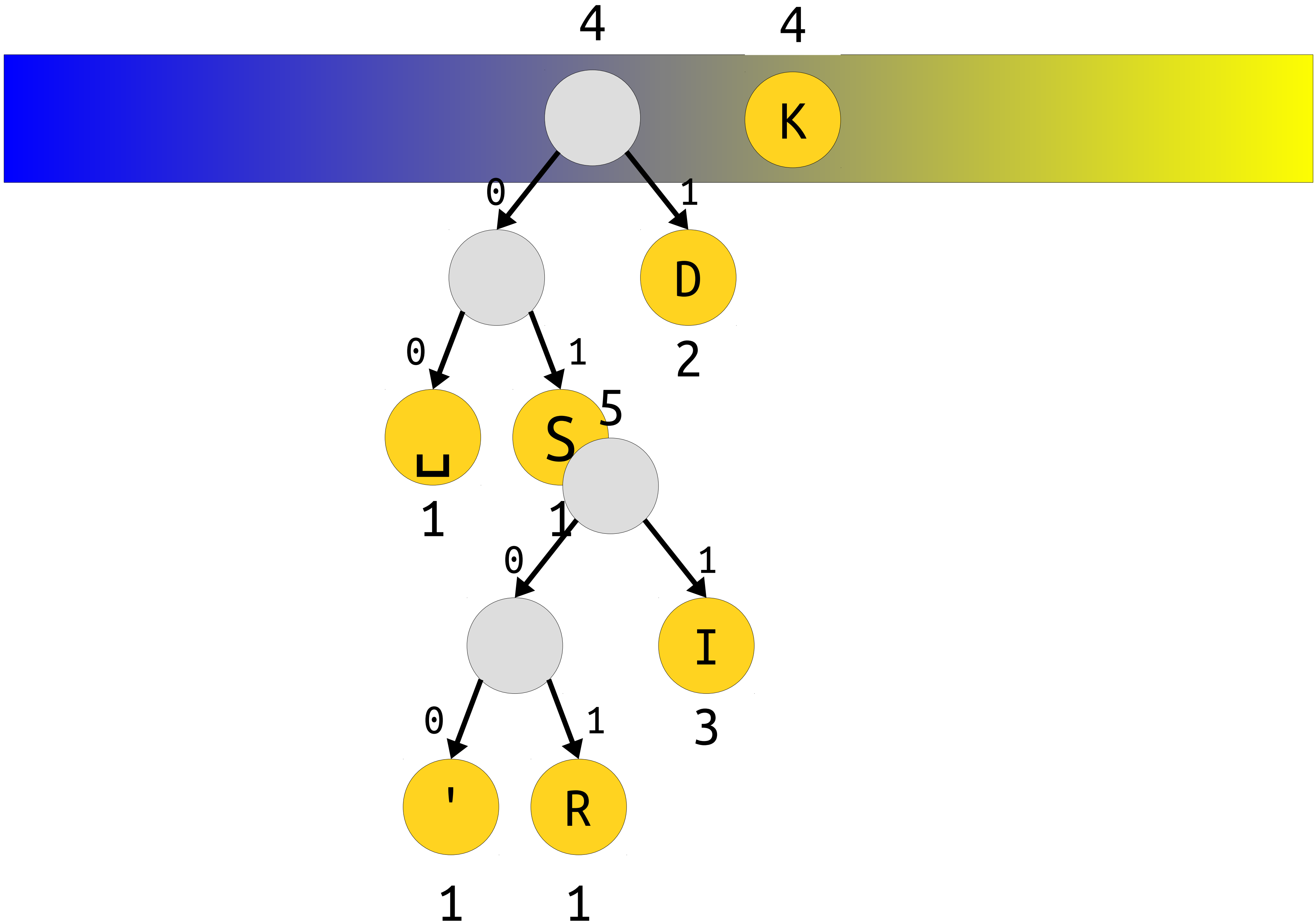

4
4
K
0
1
D
2
0
1
␣
S
1
1
0
1
I
3
'
R
1
1

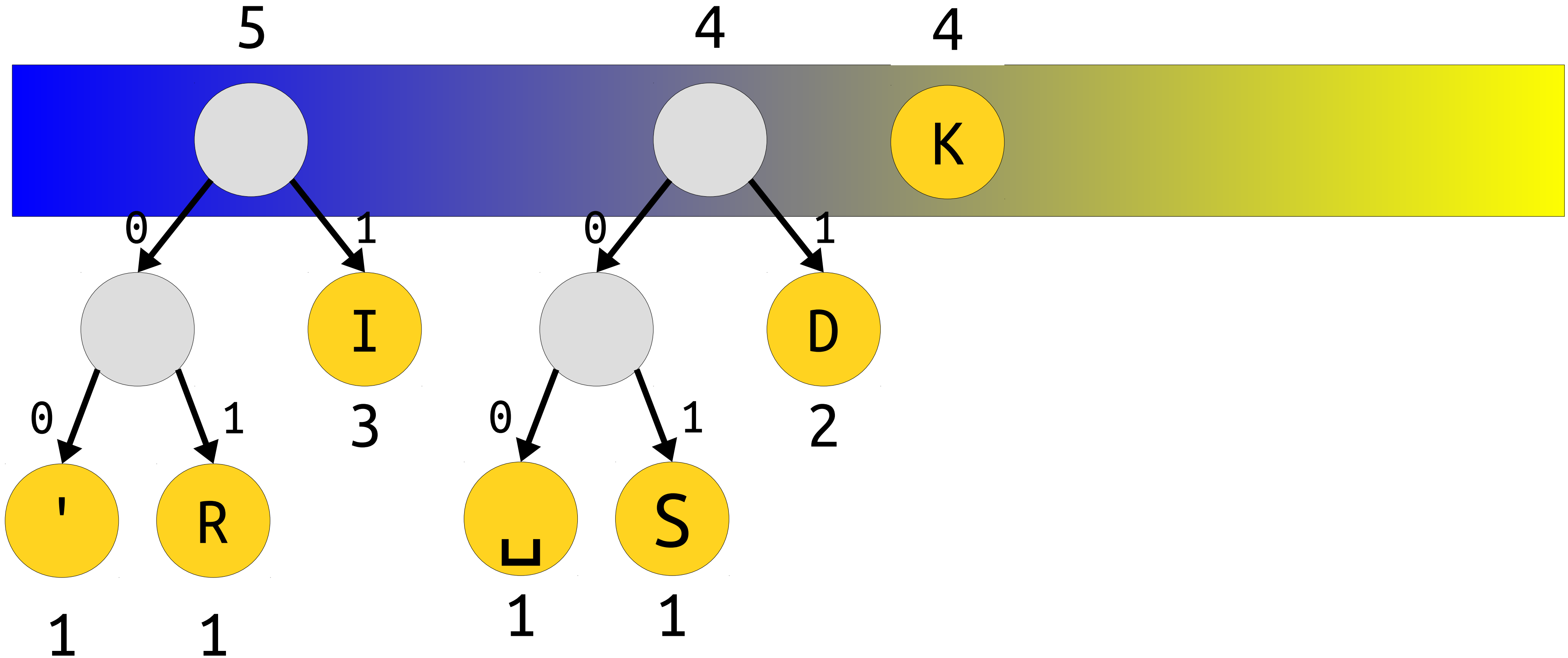


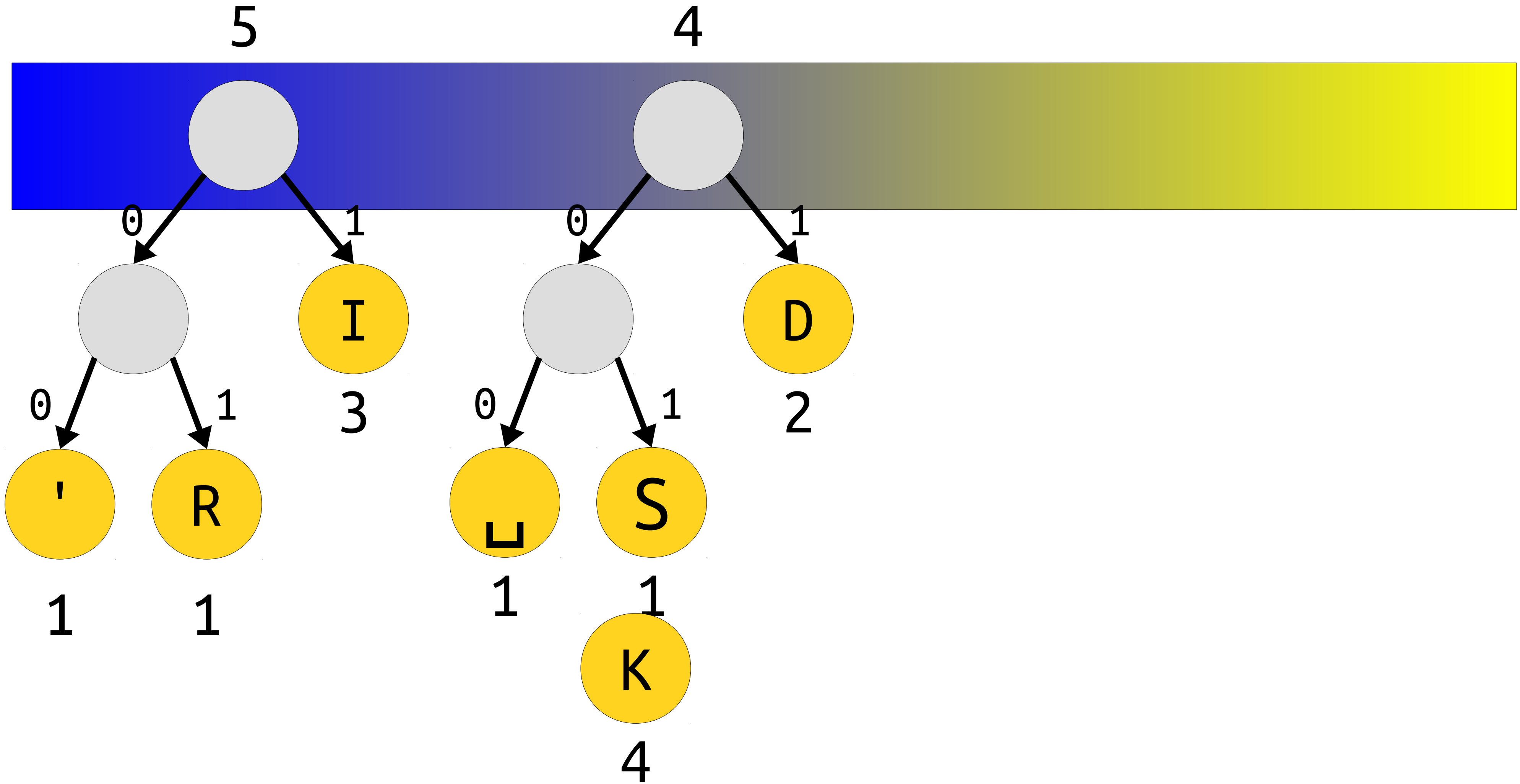
4
4
K
0
1
D
2
0
1
␣
S
1
1
0
1
I
3
0
1
'
R
1
1

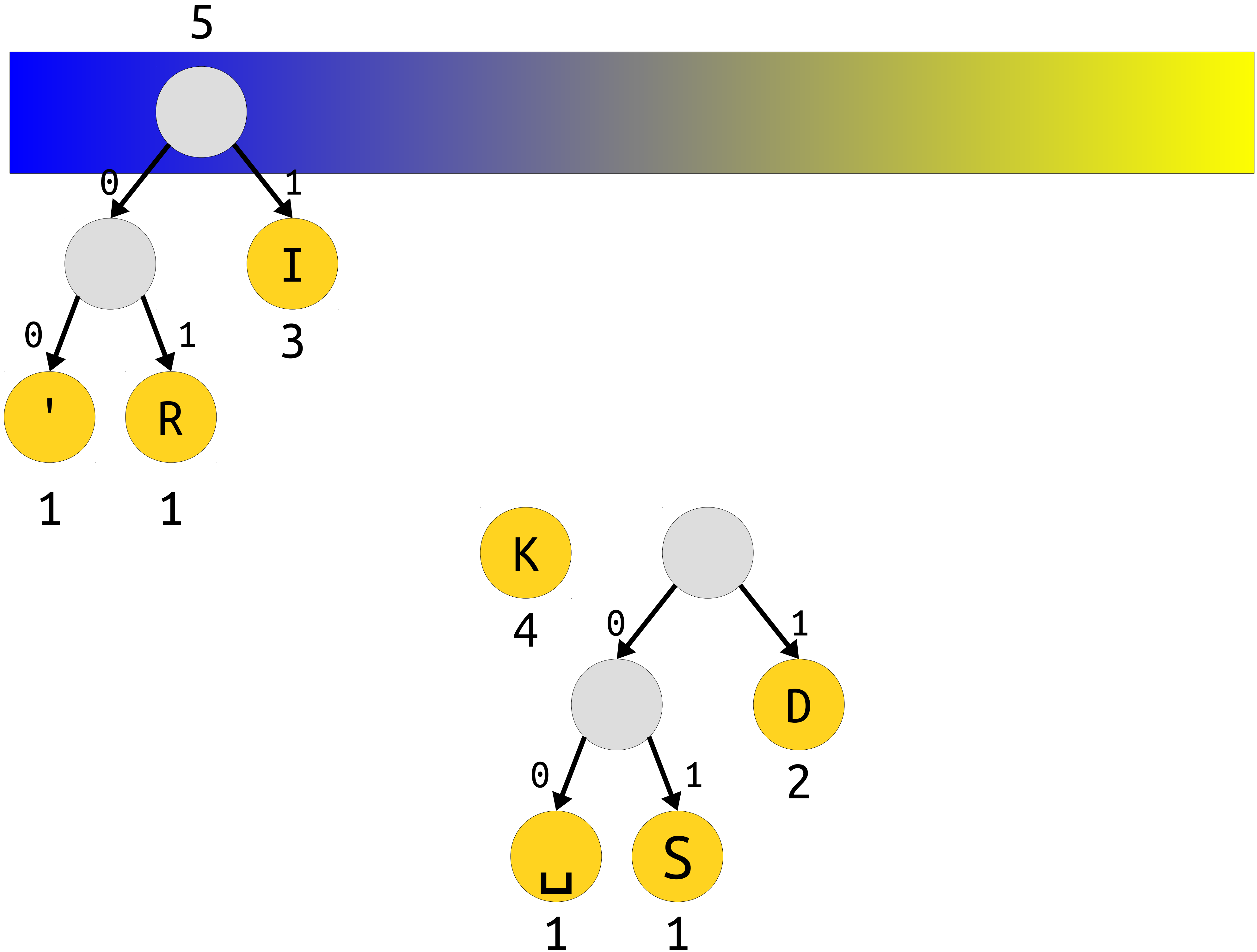


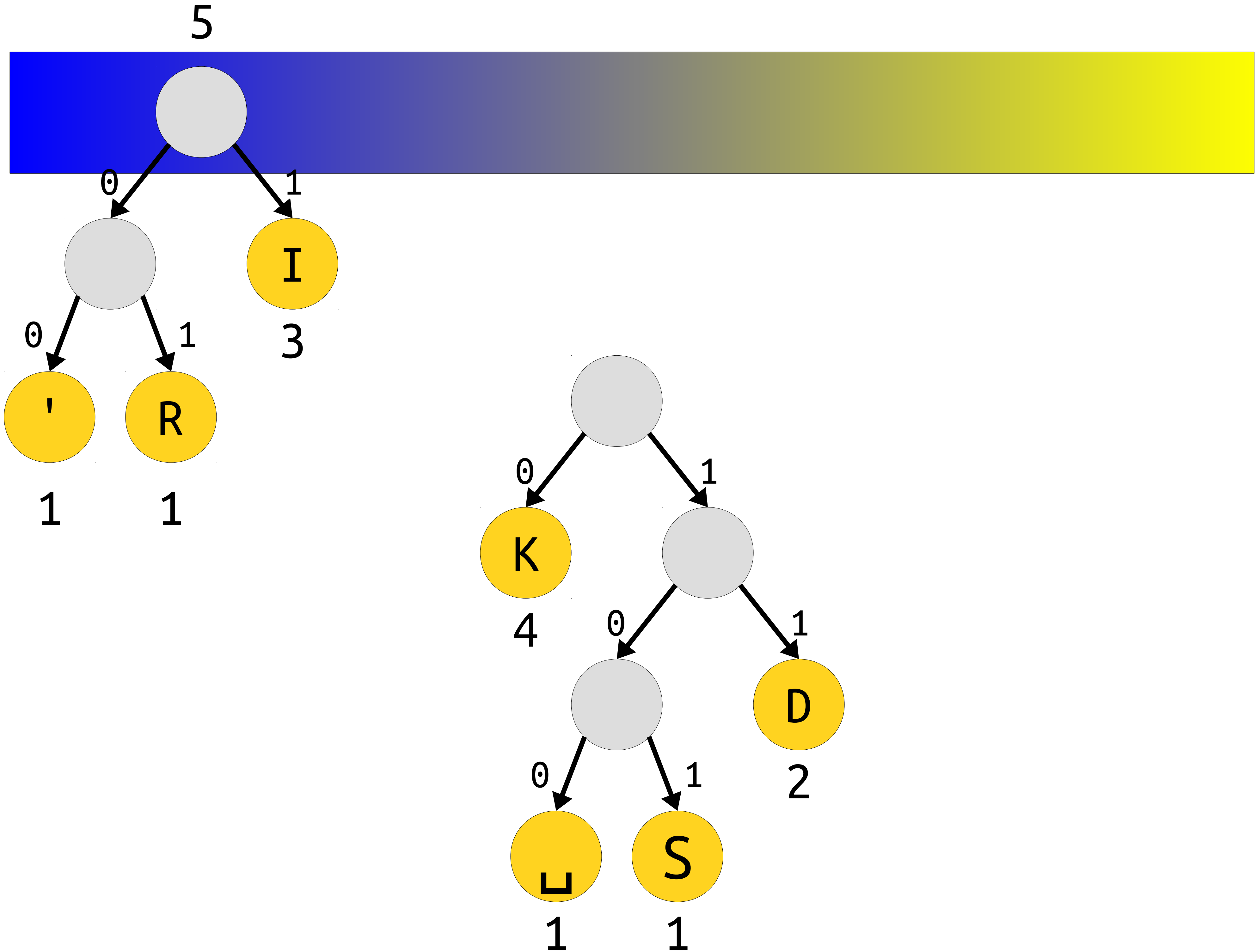
4
4
K
0
1
D
2
0
1
␣
S
1
1
5
0
1
I
3
0
1
'
R
1
1

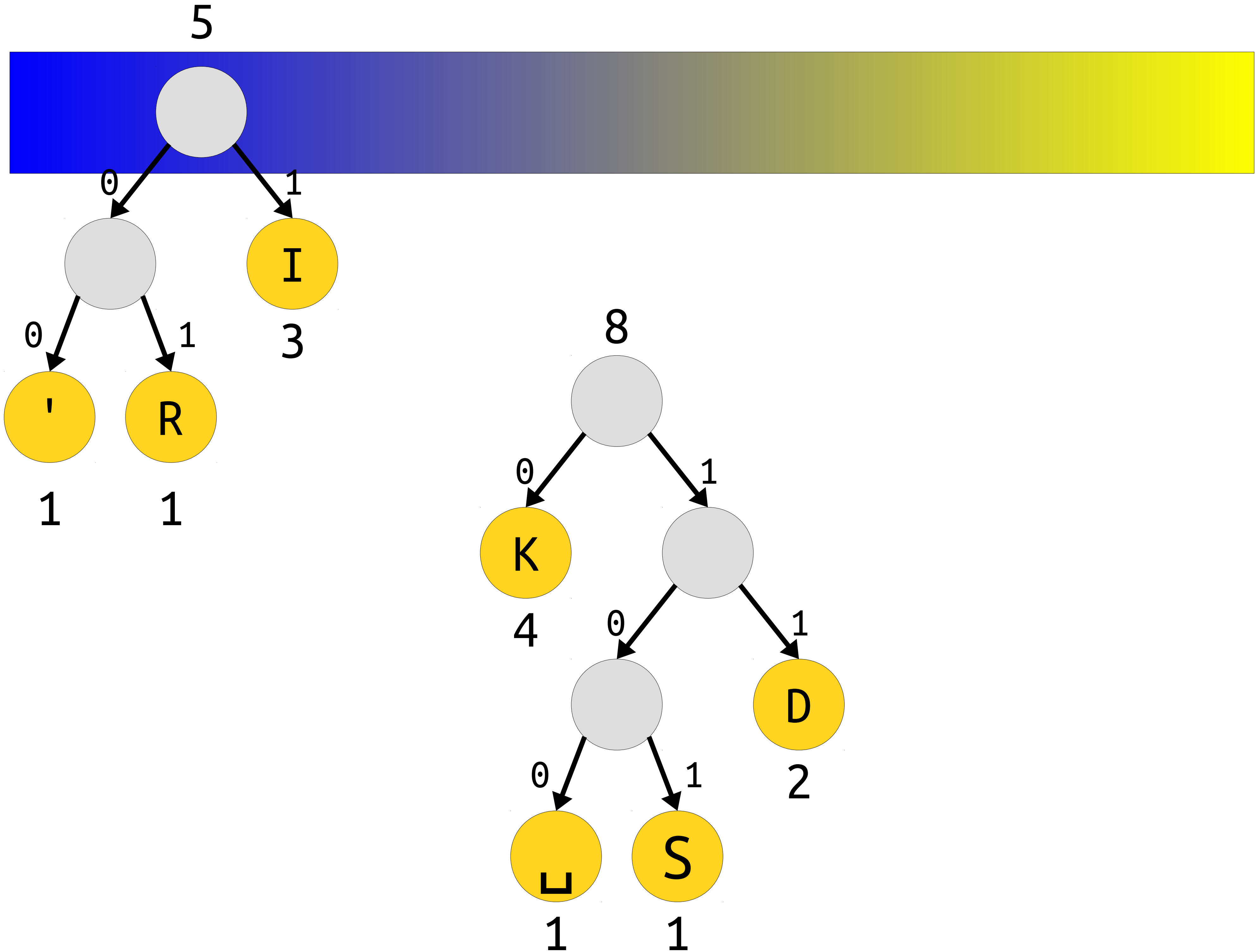

4
4
K
0
1
D
2
0
1
5
S
␣
1
1
0
1
I
3
0
1
'
R
1
1


5
4
4
K
0
1
0
1
I
D
0
1
0
1
3
2
'
R
␣
S
1
1
1
1

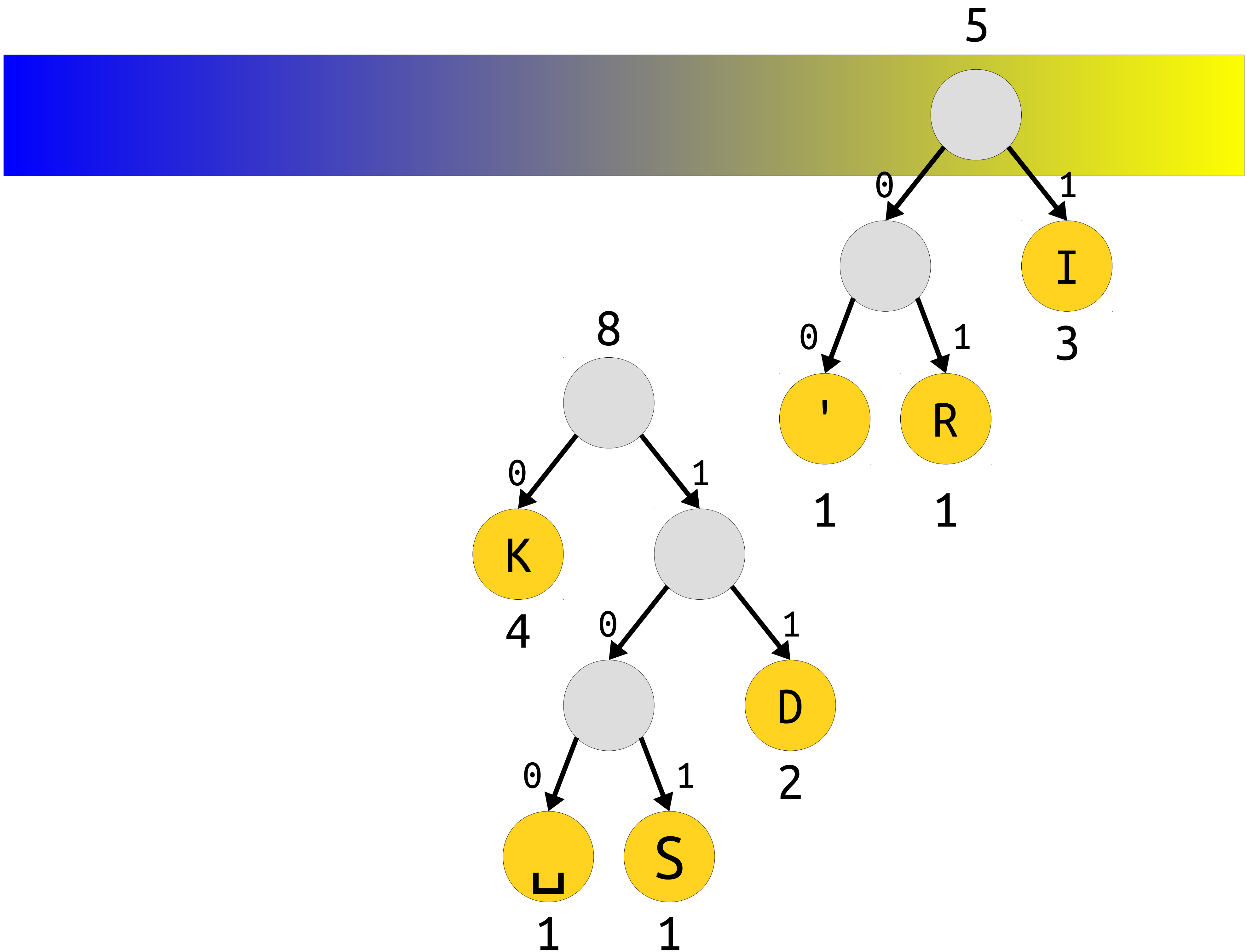

5
4
0
1
0
1
I
D
0
1
0
1
3
2
'
R
␣
S
1
1
1
1
K
4

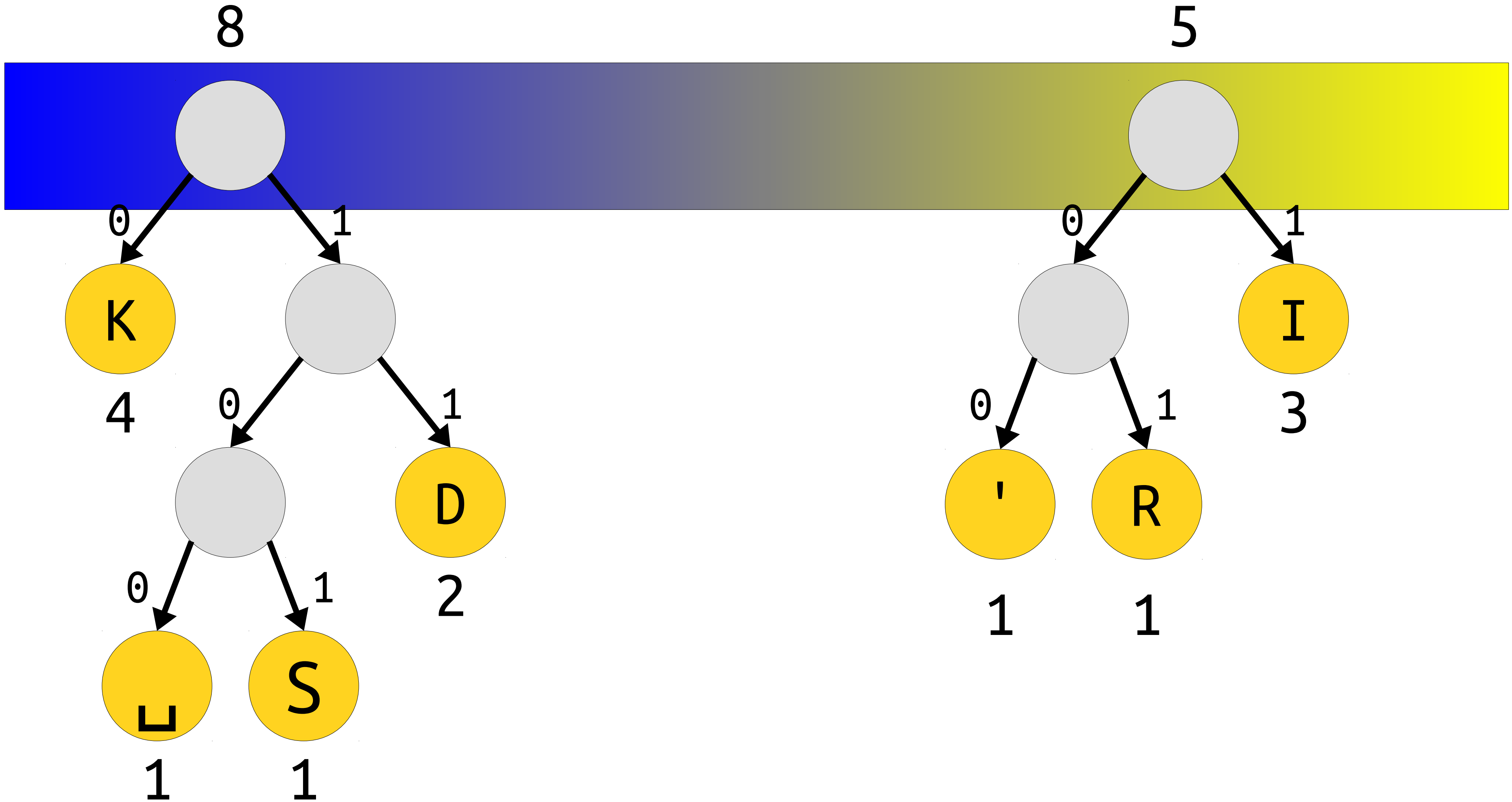

5
0
1
I
3
0
1
'
R
1
1
K
4
0
1
D
2
0
1
␣
S
1
1

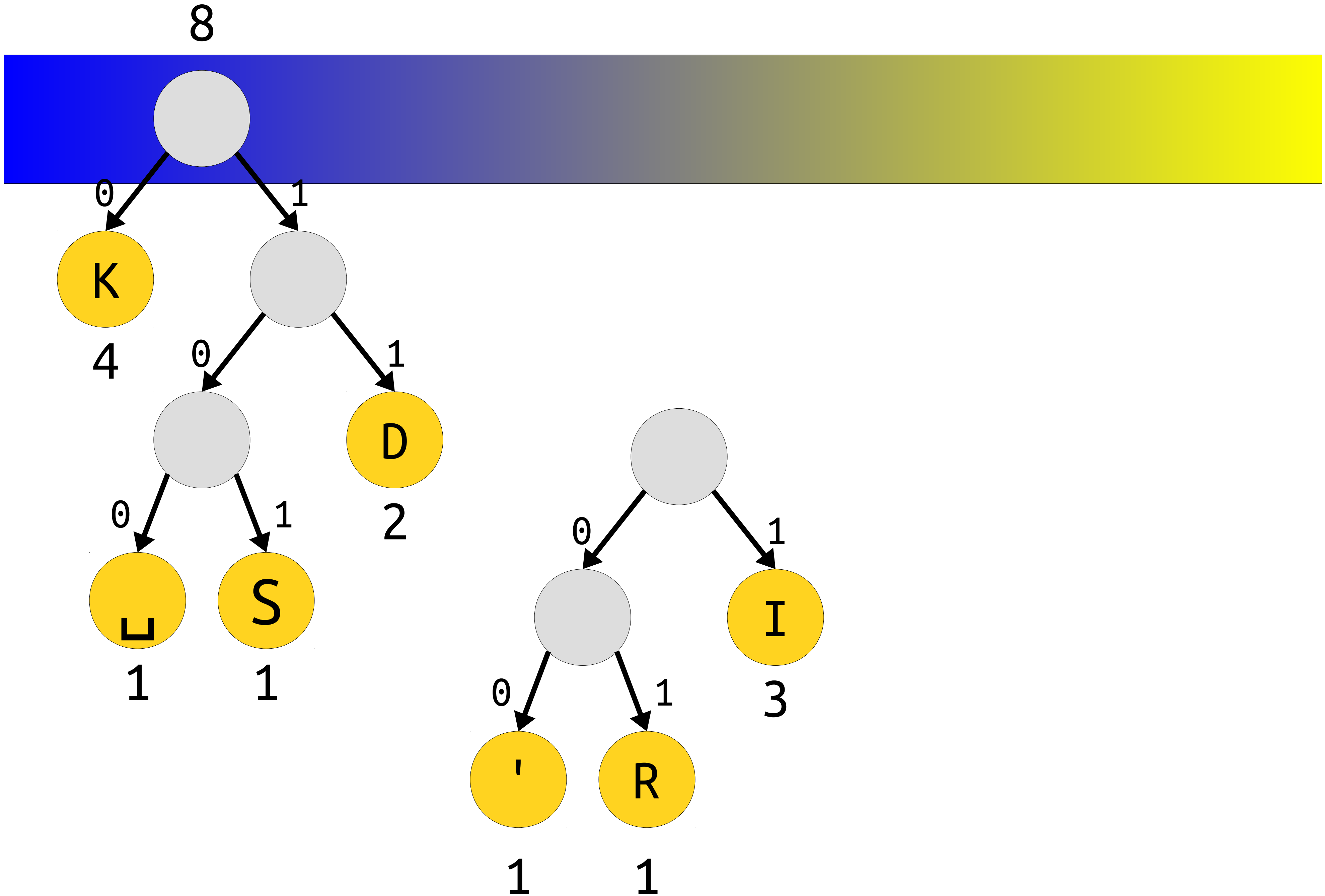

5
0
1
I
3
0
1
'
R
1
1
0
1
K
4
0
1
D
2
0
1
␣
S
1
1



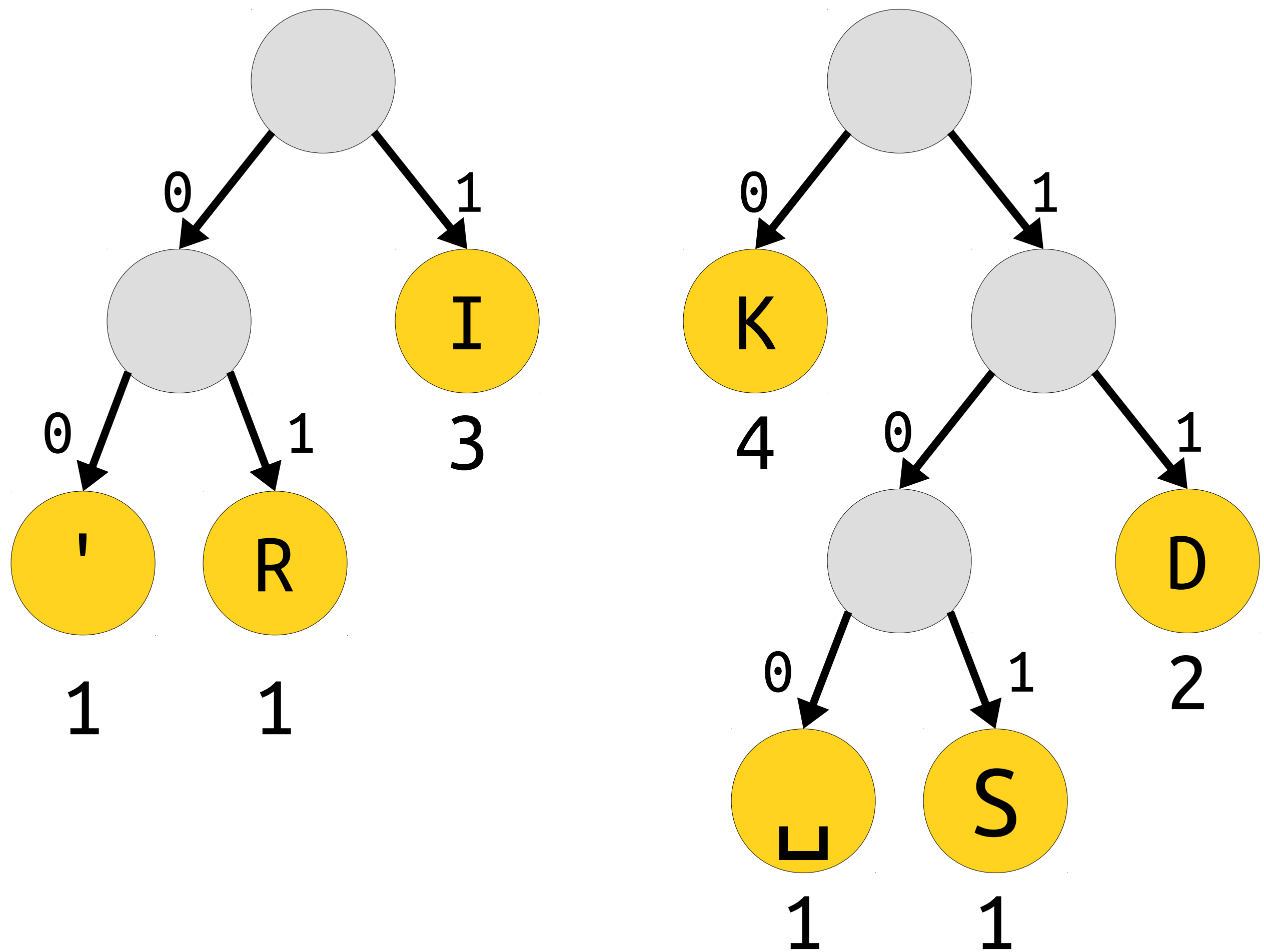
5
0
1
I
3
0
1
'
R
1
1
8
0
1
K
4
0
1
D
2
0
1
␣
S
1
1


5
0
1
I
3
0
1
'
R
1
1
8
0
1
K
4
0
1
D
2
0
1
␣
S
1
1

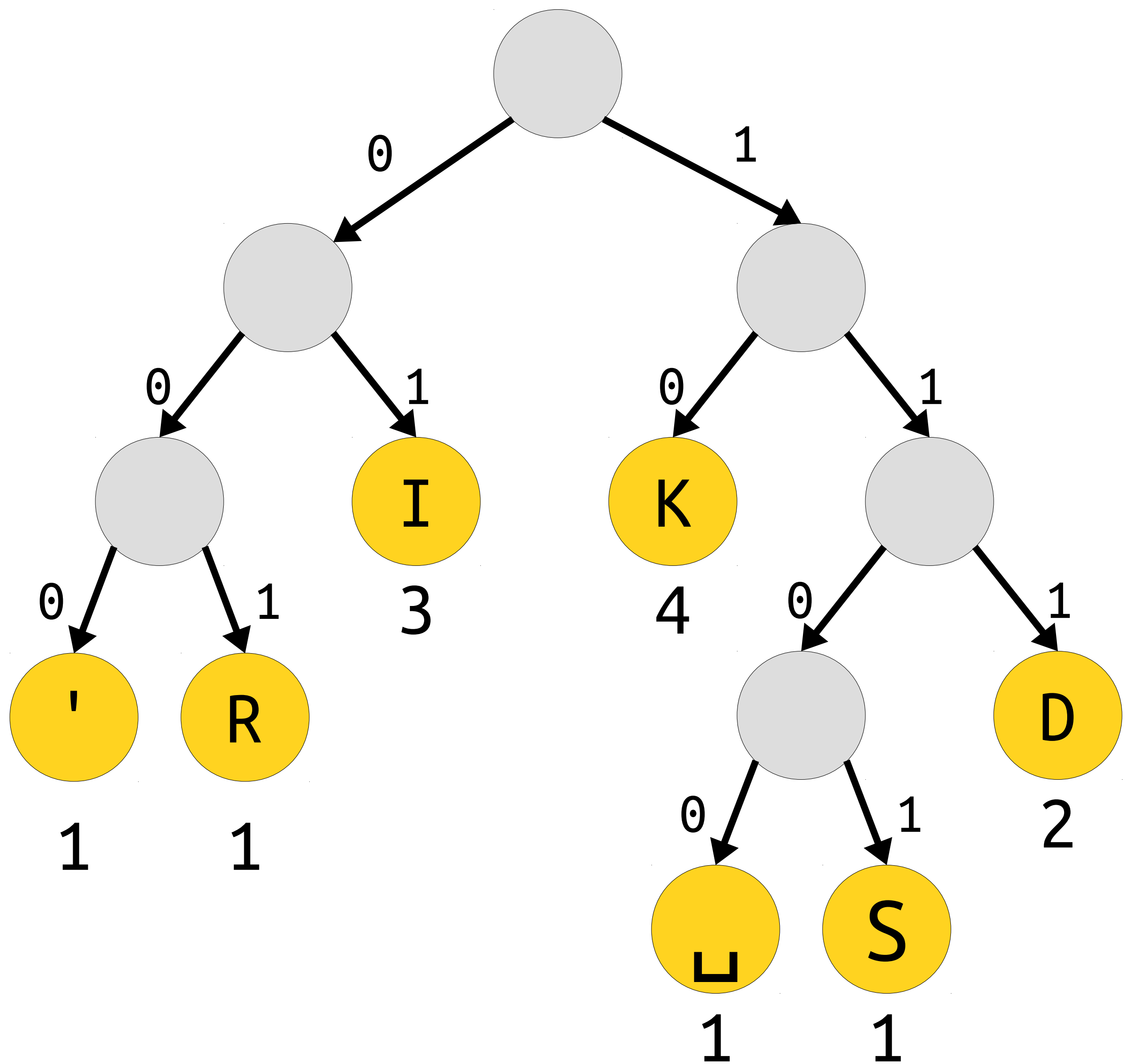

8
5
0
1
0
1
K
I
4
3
0
1
0
1
D
'
R
2
1
1
0
1
␣
S
1
1

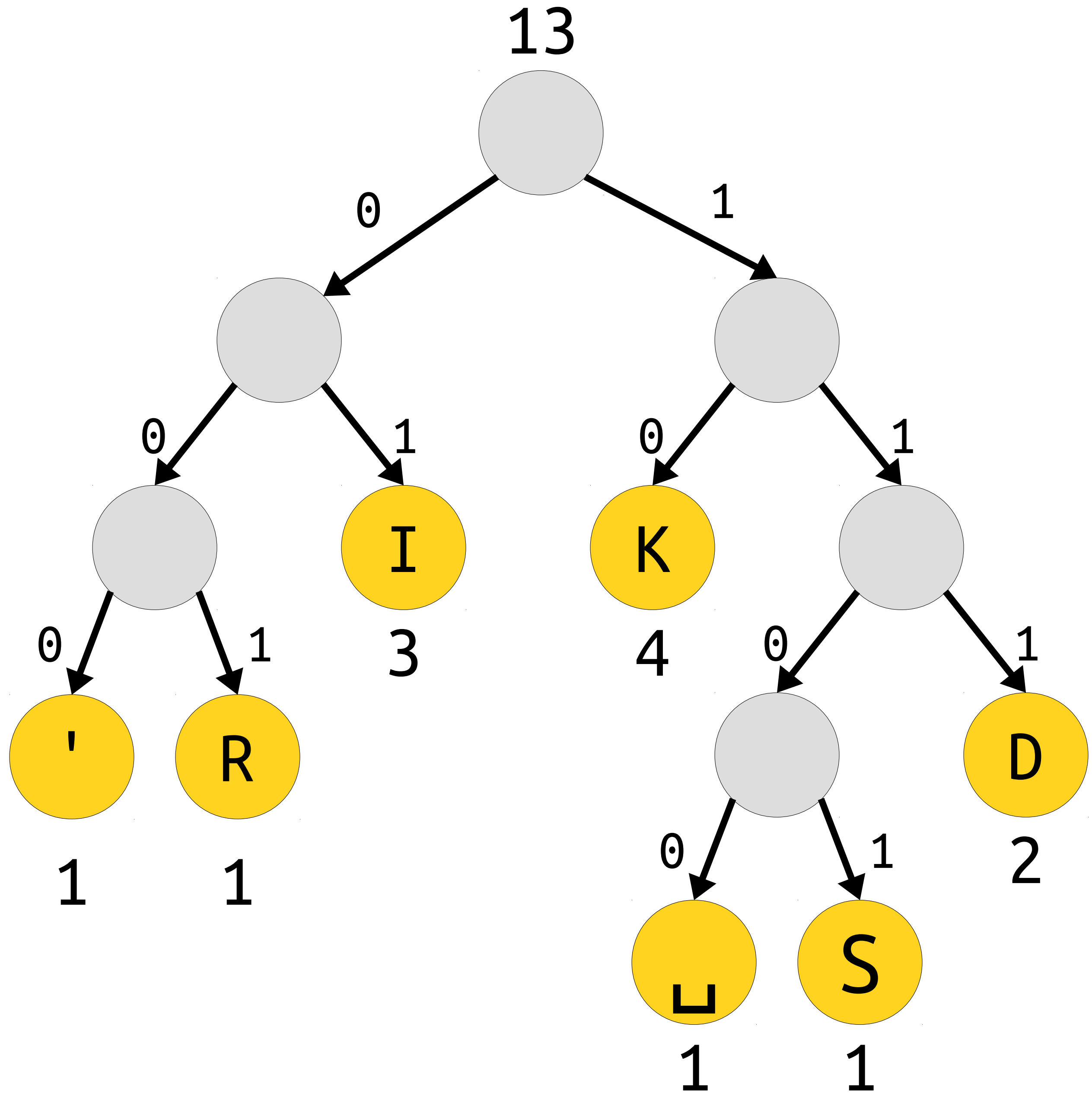


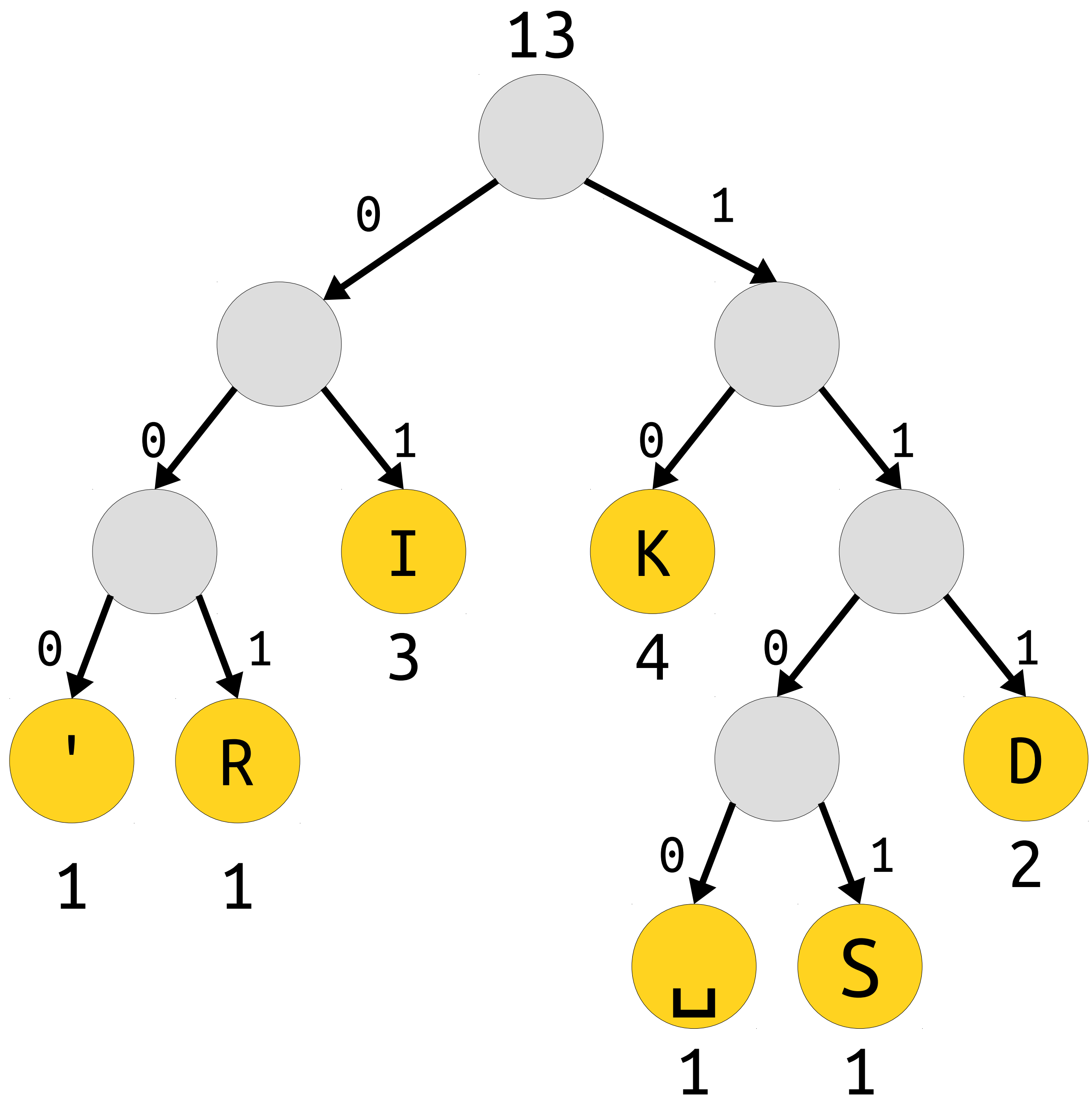
8
0
1
K
4
0
1
D
2
0
1
␣
1
S
1
0
1
I
3
0
1
'
1
R
1

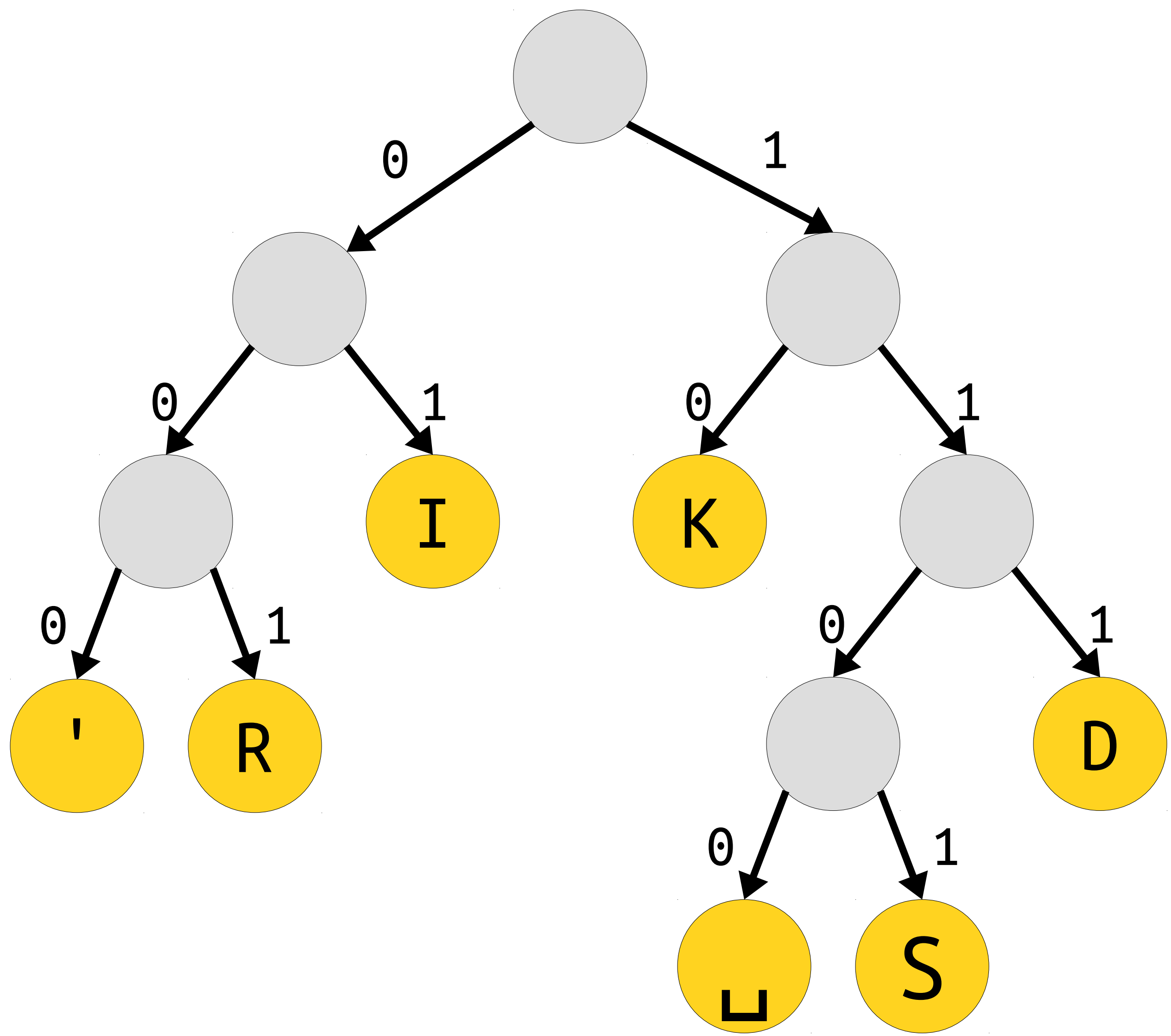


0
1
I
3
0
1
'
R
1
1
0
1
K
4
0
1
D
2
0
1
␣
S
1
1


0
1
0
1
0
1
I
K
0
1
0
1
3
4
'
R
D
1
1
2
0
1
␣
S
1
1

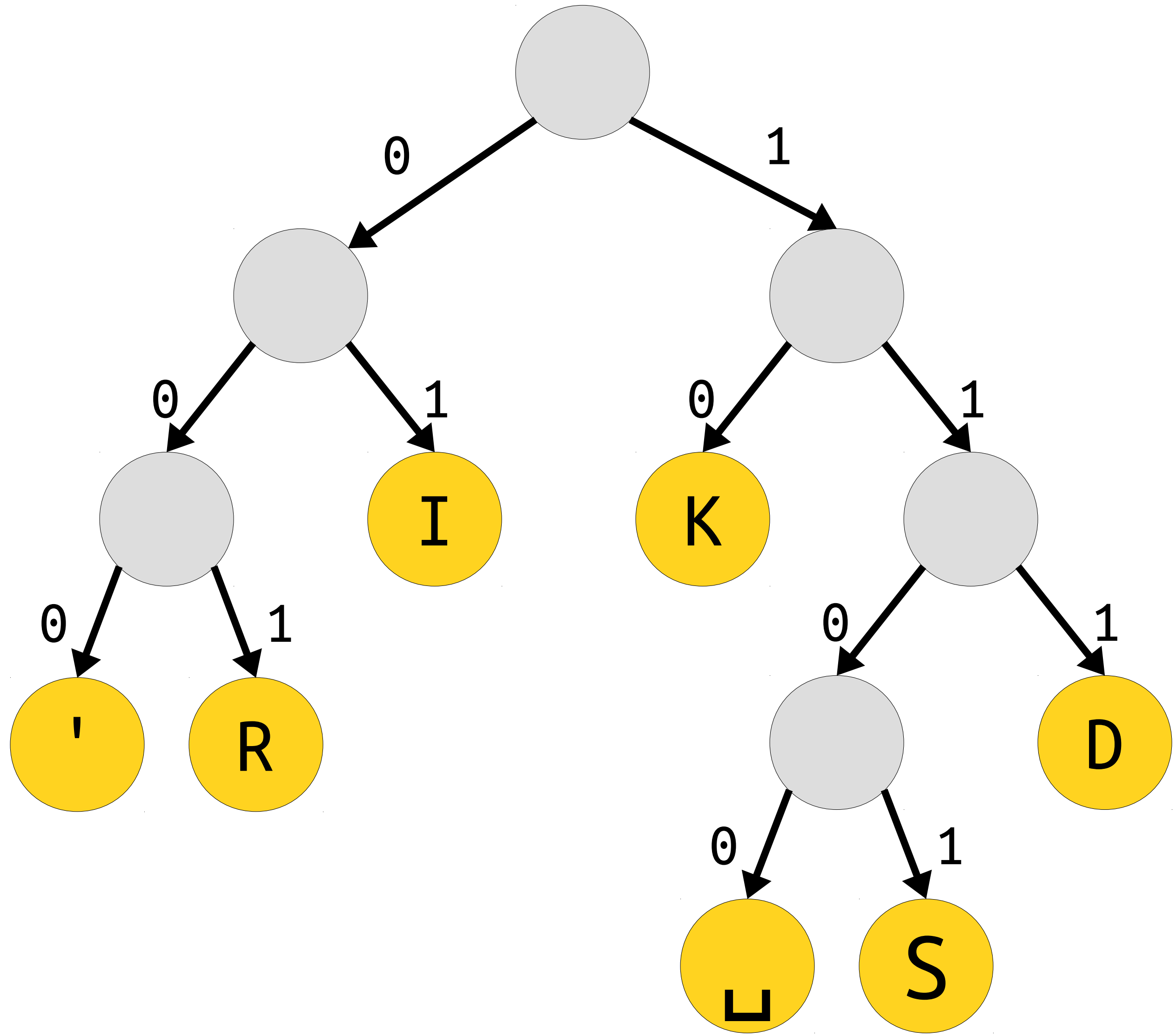

13
0
1
0
1
0
1
I
K
3
4
0
1
0
1
'
R
D
1
1
2
0
1
␣
S
1
1


13
0
1
0
1
0
1
I
K
0
1
0
1
3
4
0
1
'
R
D
1
1
0
1
2
␣
S
1
1



0
1
0
1
0
1
I
K
0
1
0
1
'
R
D
0
1
␣
S

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



★ Huffman Coding ★

- Create a priority queue that holds partial trees.
- Create one leaf node per distinct character in the input string. The weight of that leaf is the frequency of the character. Add each to the priority queue.
- While there are two or more trees in the priority queue:
  - Dequeue the two lowest-priority trees.
  - Combine them together to form a new tree whose weight is the sum of the weights of the two trees.
  - Add that tree back to the priority queue.

An Important Detail

Prefix Codes

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

Prefix Codes

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

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

Prefix Codes

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

1001001100001101110011101101110110

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

Prefix Codes

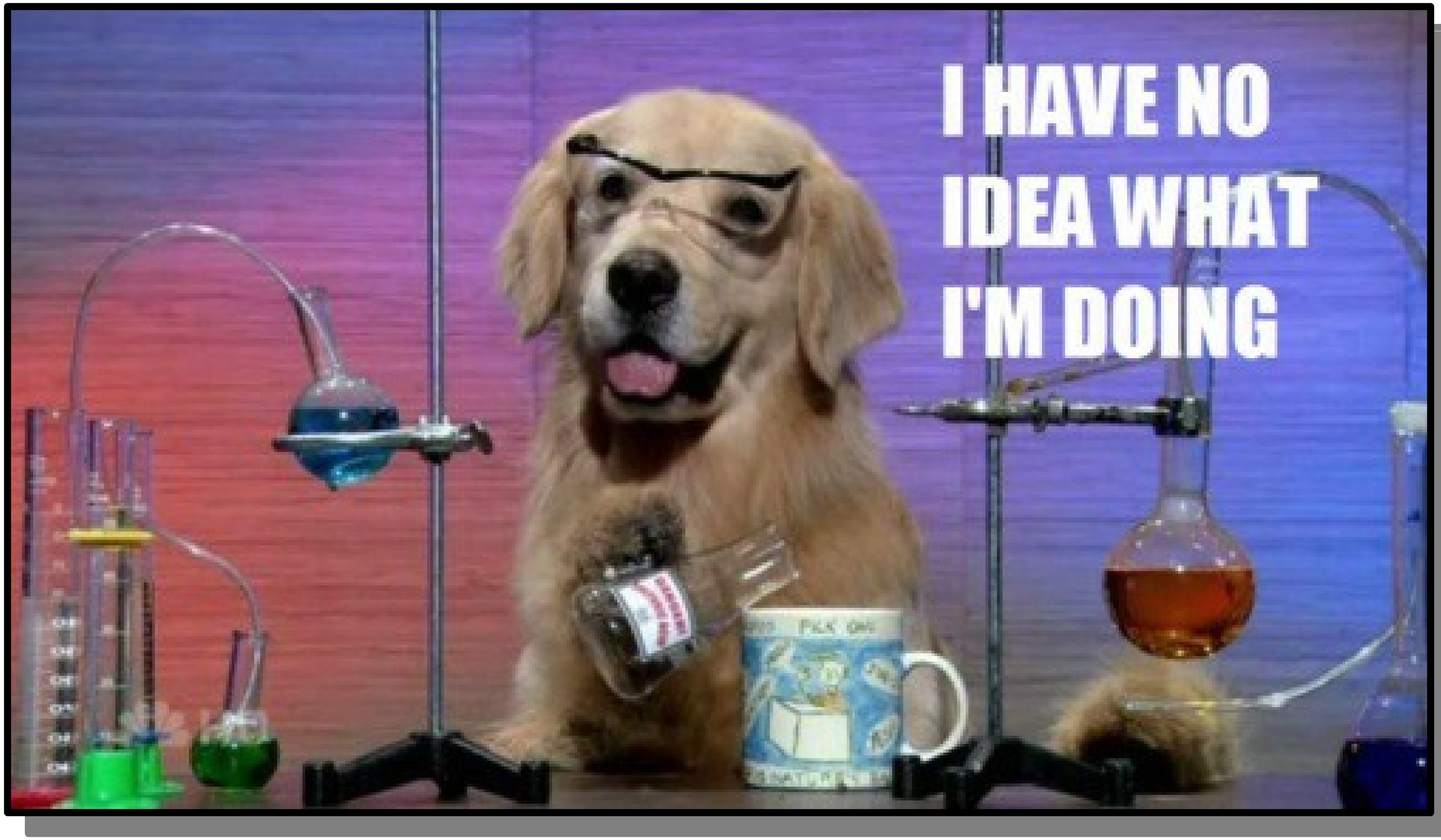
character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
␣	1100

1001001100001101110011101101110110

Prefix Codes

1001001100001101110011101101110110

Prefix Codes



110110

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 information to rebuild the tree.

Encoded Tree	`11011100101110111100010011010101110…`

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

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.

More to Explore

- ***UTF-8 and Unicode***
  - A variable-length encoding that has since replaced ASCII.
- ***Kolmogorov Complexity***
  - What's the theoretical limit to compression techniques?
- ***Adaptive Coding Techniques***
  - Can you change your encoding system as you go?
- ***Shannon Entropy***
  - A mathematical bound on Huffman coding.
- ***Binary Tries***
  - Other applications of trees like these!

Your Action Items

- ***Start Assignment 8.***
  - You have plenty of time to finish this one if you begin early.
  - Please don’t wait until the last minute – no late submissions will be accepted.

Next Time

- ***Graphs***
  - Representing networks of all sorts.
- ***Graph Searches***
  - A new perspective on some earlier ideas.