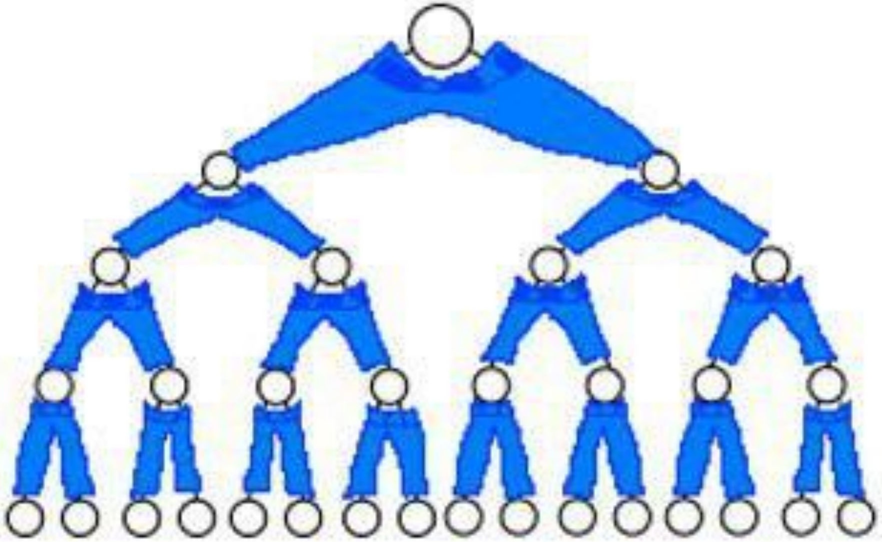
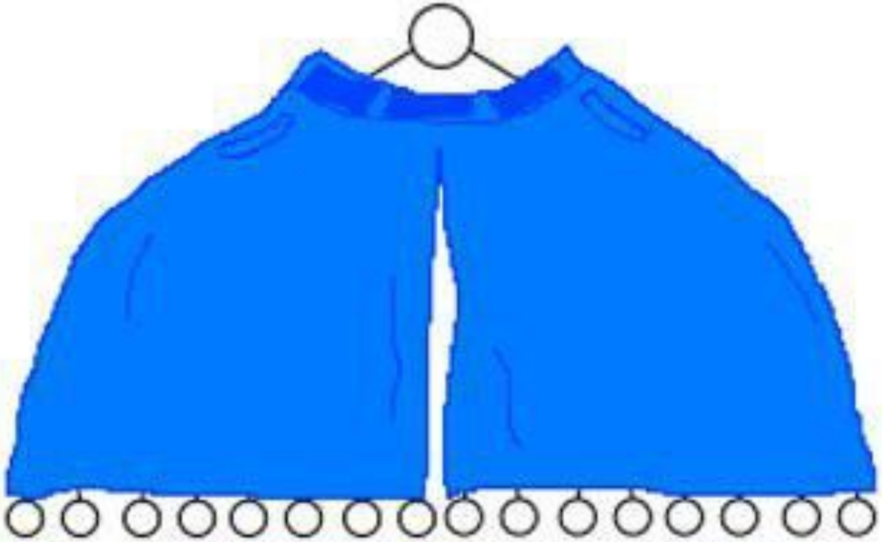
Huffman Coding

Today's question has a visual component, posted on the next slide.

A

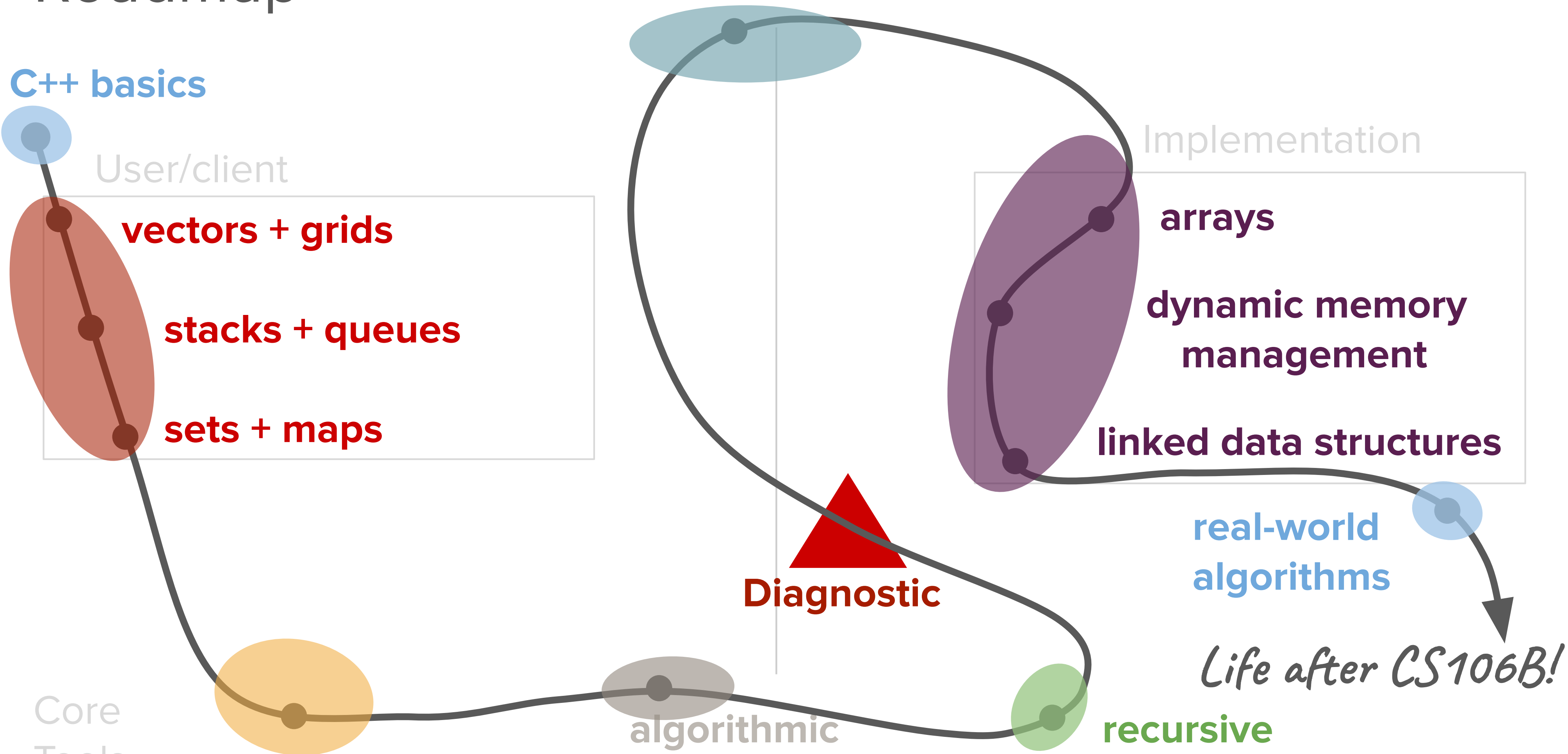


B



If a binary tree wore pants, would it wear them like in picture A or in picture B?

Roadmap



Object-Oriented Programming

C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Implementation

arrays

dynamic memory management

linked data structures

real-world algorithms

Diagnostic

Life after CS106B!

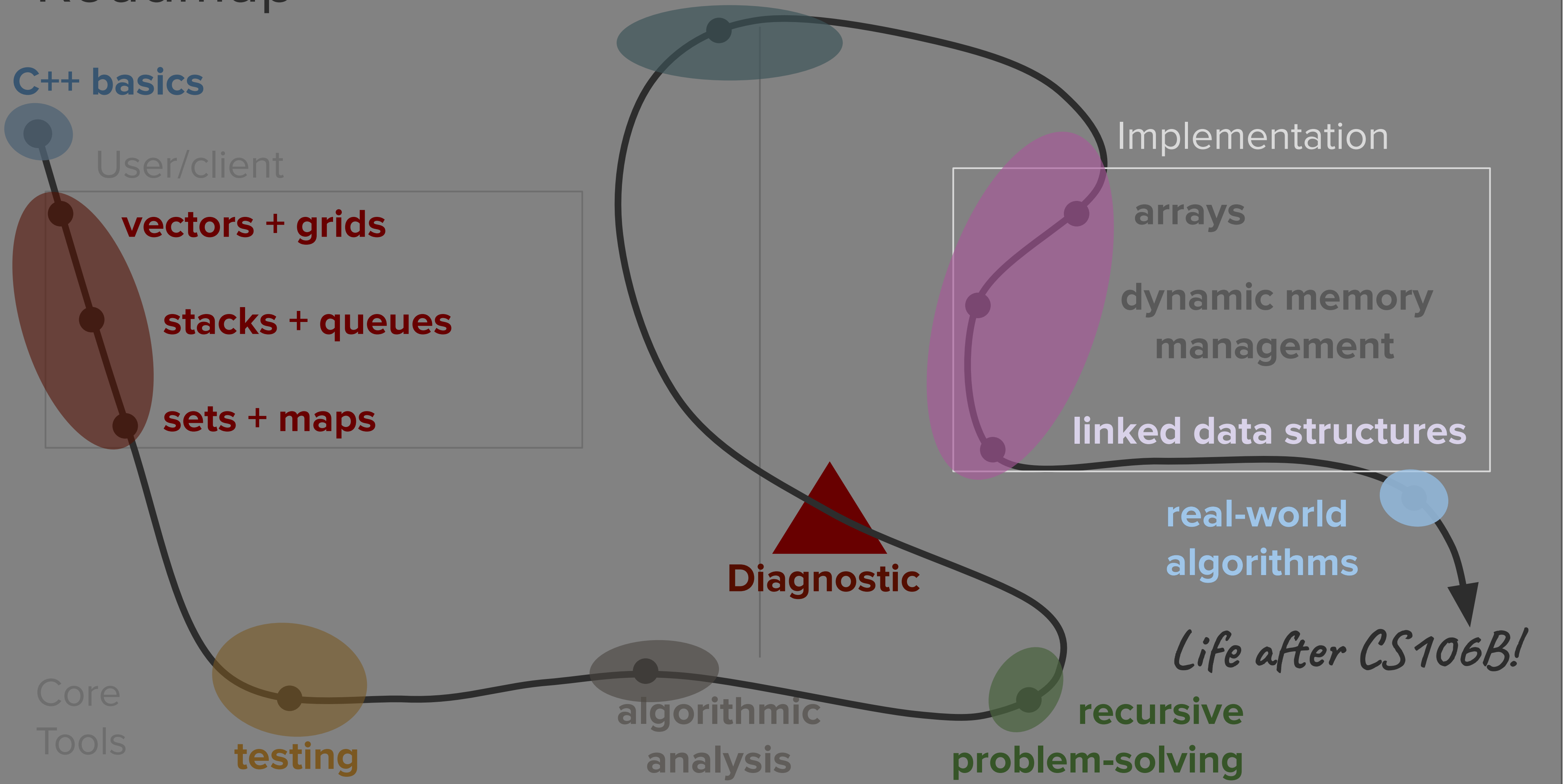
Core Tools

testing

algorithmic analysis

recursive problem-solving

Roadmap



C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Core Tools

testing

algorithmic analysis

recursive problem-solving

Diagnostic

Object-Oriented Programming

Implementation

arrays

dynamic memory management

linked data structures

real-world algorithms

Life after CS106B!

Today's questions

How can we use trees to develop more compact and efficient data representation techniques?

Today's topics

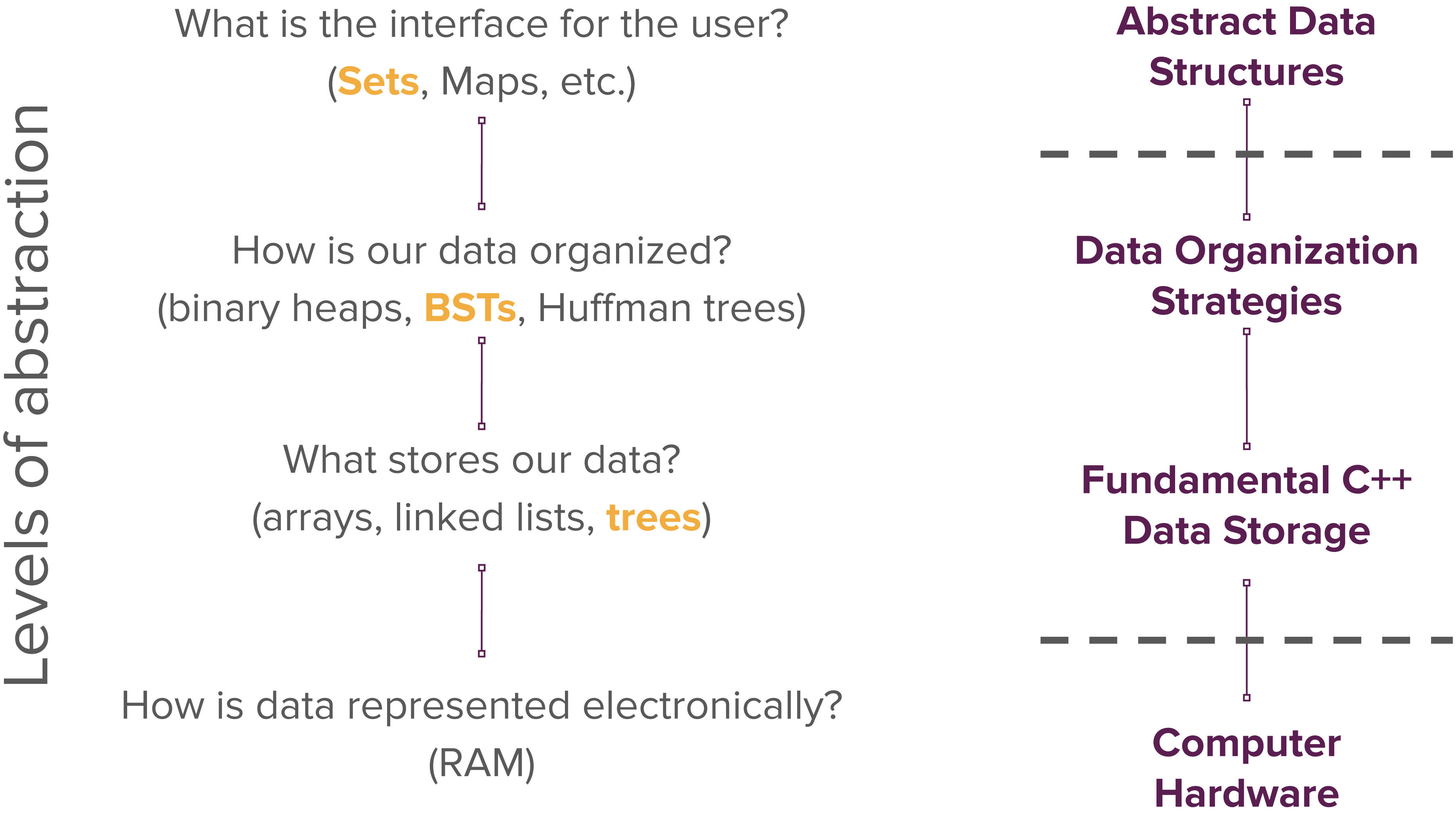
1. Binary Search Tree Review
2. Data Compression and Encoding
3. Huffman Coding

Review

[binary search trees]

Key Idea: The distance from each element (node) in a tree to the top of the tree (the root) is small, even if there are many elements.

How can we take advantage of trees to structure and efficiently manipulate data?



Levels of abstraction

What is the interface for the user?
(**Sets**, Maps, etc.)

Abstract Data Structures

How is our data organized?
(binary heaps, **BSTs**, Huffman trees)

Data Organization Strategies

What stores our data?
(arrays, linked lists, **trees**)

Fundamental C++ Data Storage

How is data represented electronically?
(RAM)

Computer Hardware

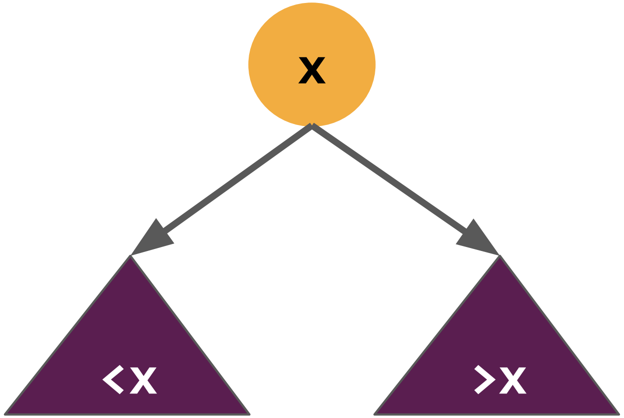
ADT Big-O Matrix

- Vectors
  - `.size() – O(1)`
  - `.add() – O(1)`
  - `v[i] – O(1)`
  - `.insert() – O(n)`
  - `.remove() – O(n)`
  - `.clear() – O(n)`
  - `traversal – O(n)`
- Grids
  - `.numRows()/.numCols() – O(1)`
  - `g[i][j] – O(1)`
  - `.inBounds() – O(1)`
  - `traversal –` $O(n^2)$
- Queues
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.enqueue() – O(1)`
  - `.dequeue() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Stacks
  - `.size() – O(1)`
  - `.peek() – O(1)`
  - `.push() – O(1)`
  - `.pop() – O(1)`
  - `.isEmpty() – O(1)`
  - `traversal – O(n)`
- Sets
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `.add() – O(log(n))`
  - `.remove() – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`
- Maps
  - `.size() – O(1)`
  - `.isEmpty() – O(1)`
  - `m[key] – O(log(n))`
  - `.contains() – O(log(n))`
  - `traversal – O(n)`

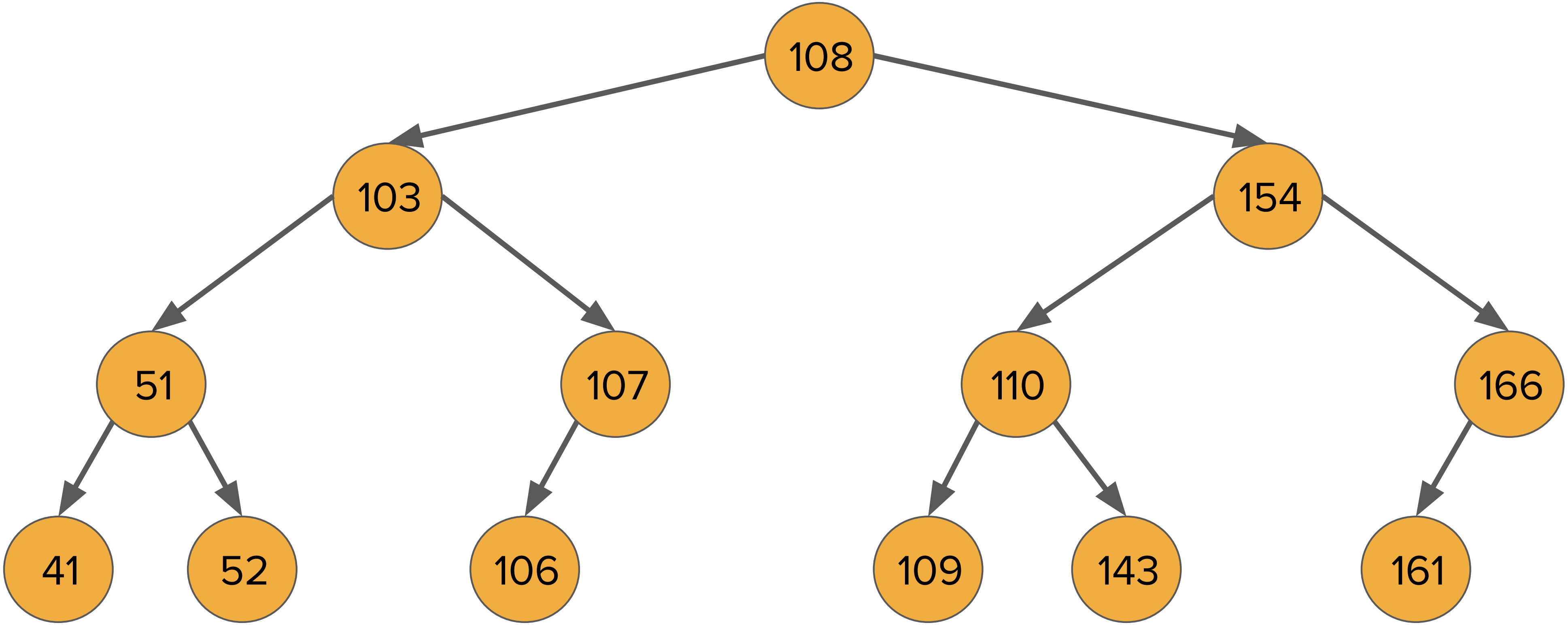
A binary search tree is either...

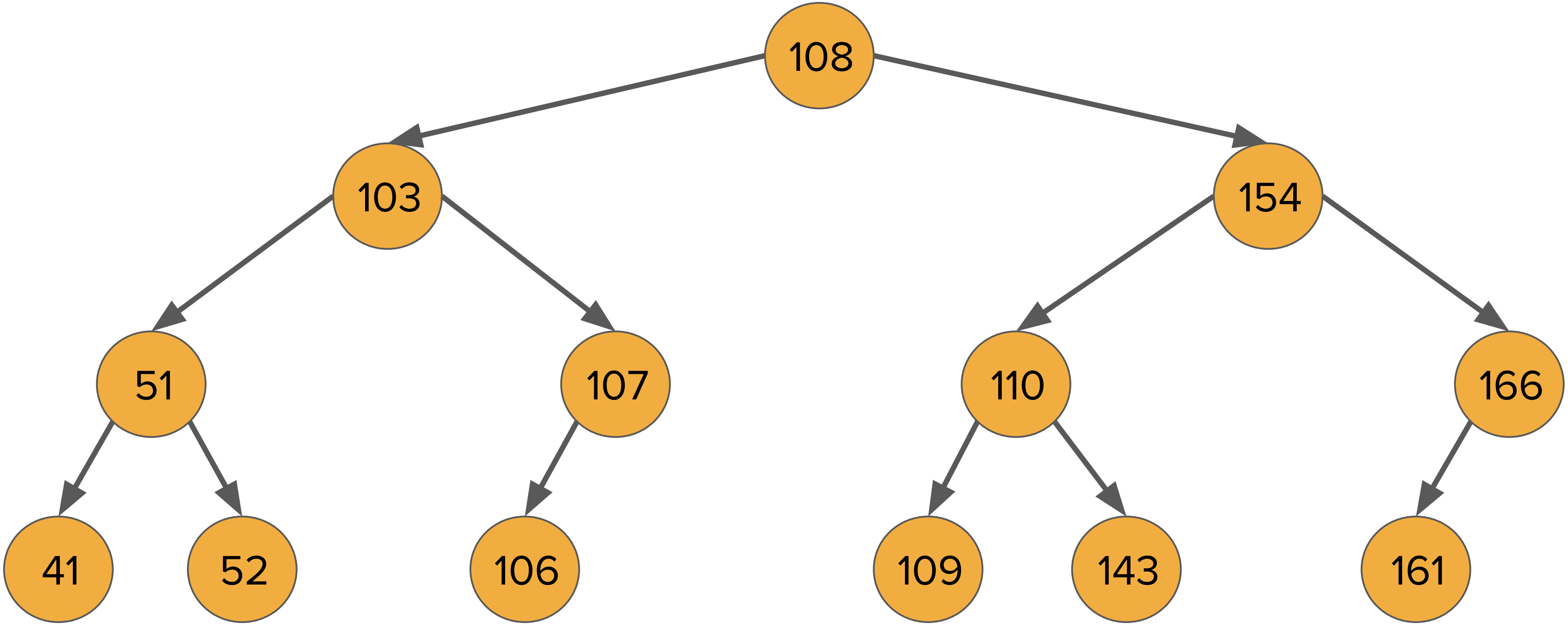
an empty data structure represented by nullptr or...

a single node, whose left subtree is a BST of smaller values than **x**...



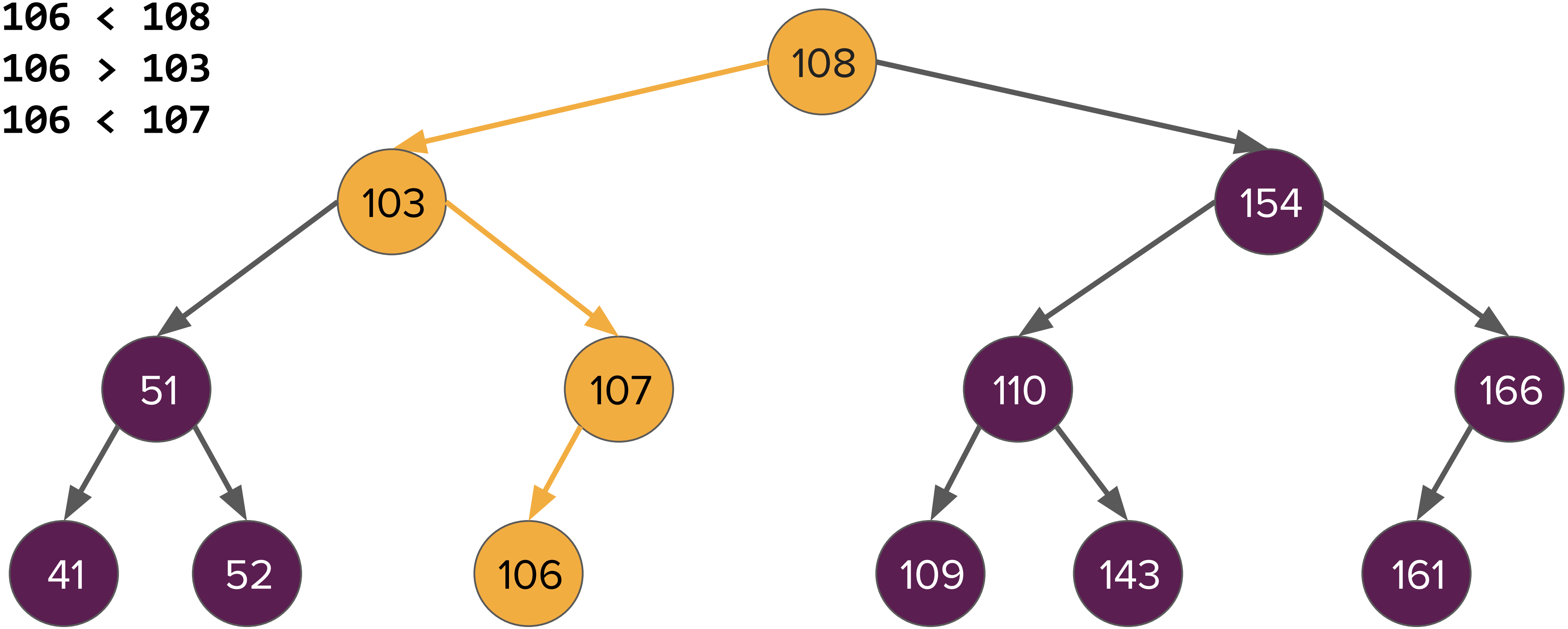
and whose right subtree is a BST of larger values than **x**.


108
103
154
51
107
110
166
41
52
106
109
143
161



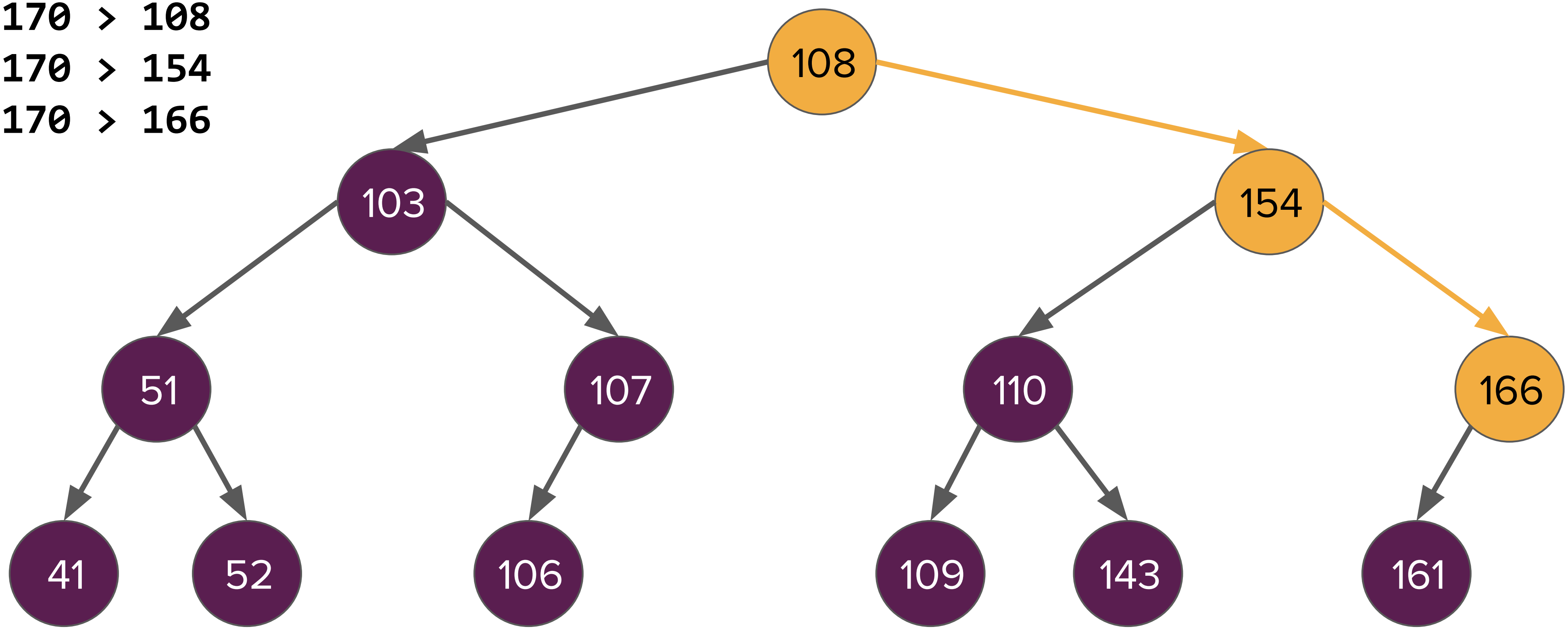
There are **n** nodes in the tree, but the path to each node is short (**`~O(log n)`**)!

```
106 < 108
106 > 103
106 < 107
```



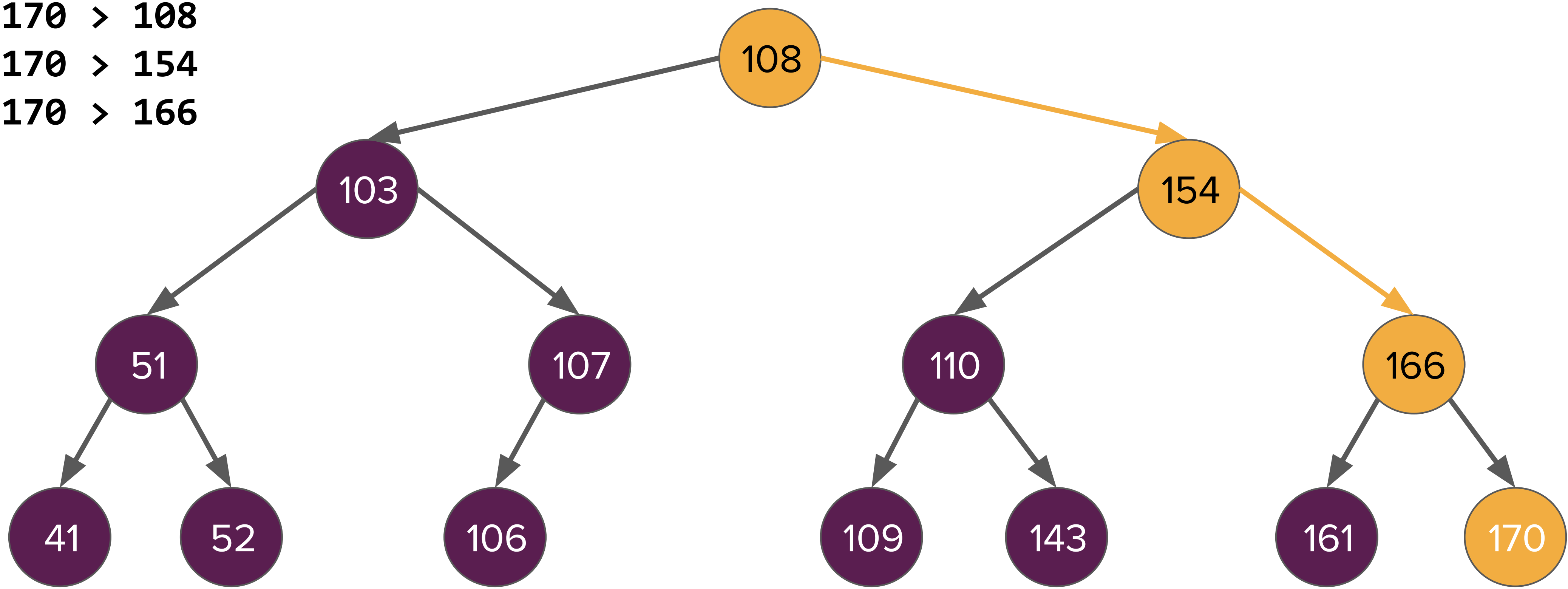
How could we check if **106** is in this tree?

```
170 > 108
170 > 154
170 > 166
```



How could we add **`170`** to this tree?

`170 > 108`
`170 > 154`
`170 > 166`



How could we add **170** to this tree?

Binary Search Tree Properties

- There are multiple valid BSTs for the same set of data. How you construct the tree/the order in which you add the elements to the tree matters!

- A binary search tree is **balanced** if its height is `O(log n)`, where `n` is the number of nodes in the tree (i.e. left/right subtrees don't differ in height by more than 1).
    - An **optimal (balanced) BST** is built by repeatedly choosing the median element as the root node of a given subtree and then separating elements into groups less than and greater than that median.
    - Lookup, insertion, and deletion with balanced BSTs all operate in `O(log n)` runtime.
    - A **self-balancing** BST reshapes itself on insertions and deletions to stay balanced (how to do this is beyond the scope of this class).

Implementing a Set with a BST

- Binary search trees are a great backing store for a data structure in which lookup/additional/removal all needs to be fast and the order of elements doesn't matter.

- This makes them a great choice for the internal data storage of a Set or Map ADT!

- Thus, we are able to build our own version of the Set ADT by using a BST to organize the internal structure of the data.

`OurSet` summary

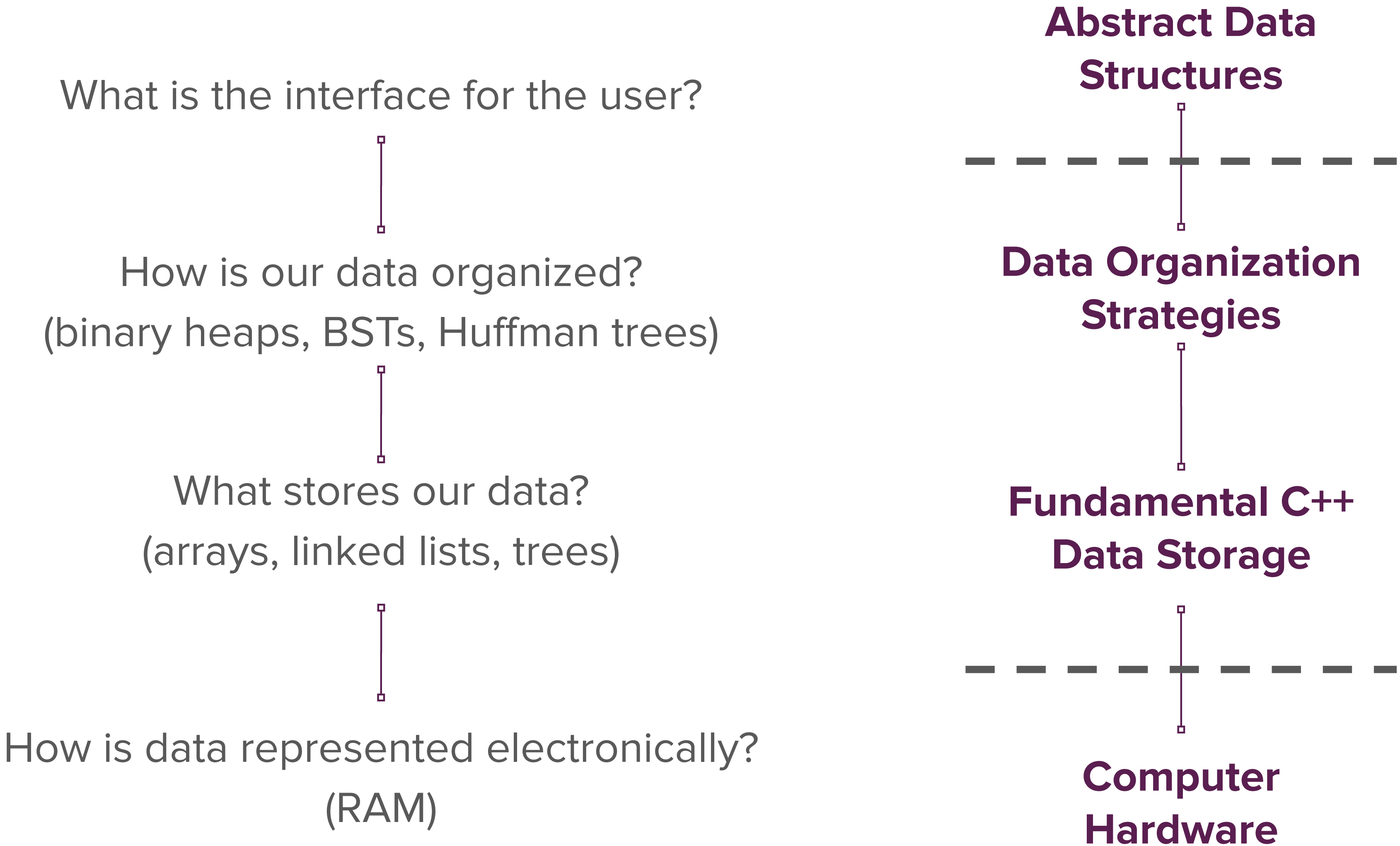
- Our tree utility functions (**`inorderPrint`**, **`freeTree`**) showed up as private member functions/helpers!
    - In-order traversal prints our elements in the correctly sorted order!

- Using a BST allowed us to take advantage of recursion to traverse our data and get an **`O(log n)`** runtime for our methods.

- Rewiring trees can be complicated!
    - Make sure to consider when nodes need to be passed by reference.
    - Check out the remove method after class if you're interested in seeing an example of tree rewiring (you won't be required to do anything this complex with tree rewiring).

How can we use trees to develop more compact and efficient data representation techniques?

Levels of abstraction



What is the interface for the user?

How is our data organized?
(binary heaps, BSTs, Huffman trees)

What stores our data?
(arrays, linked lists, trees)

How is data represented electronically?
(RAM)

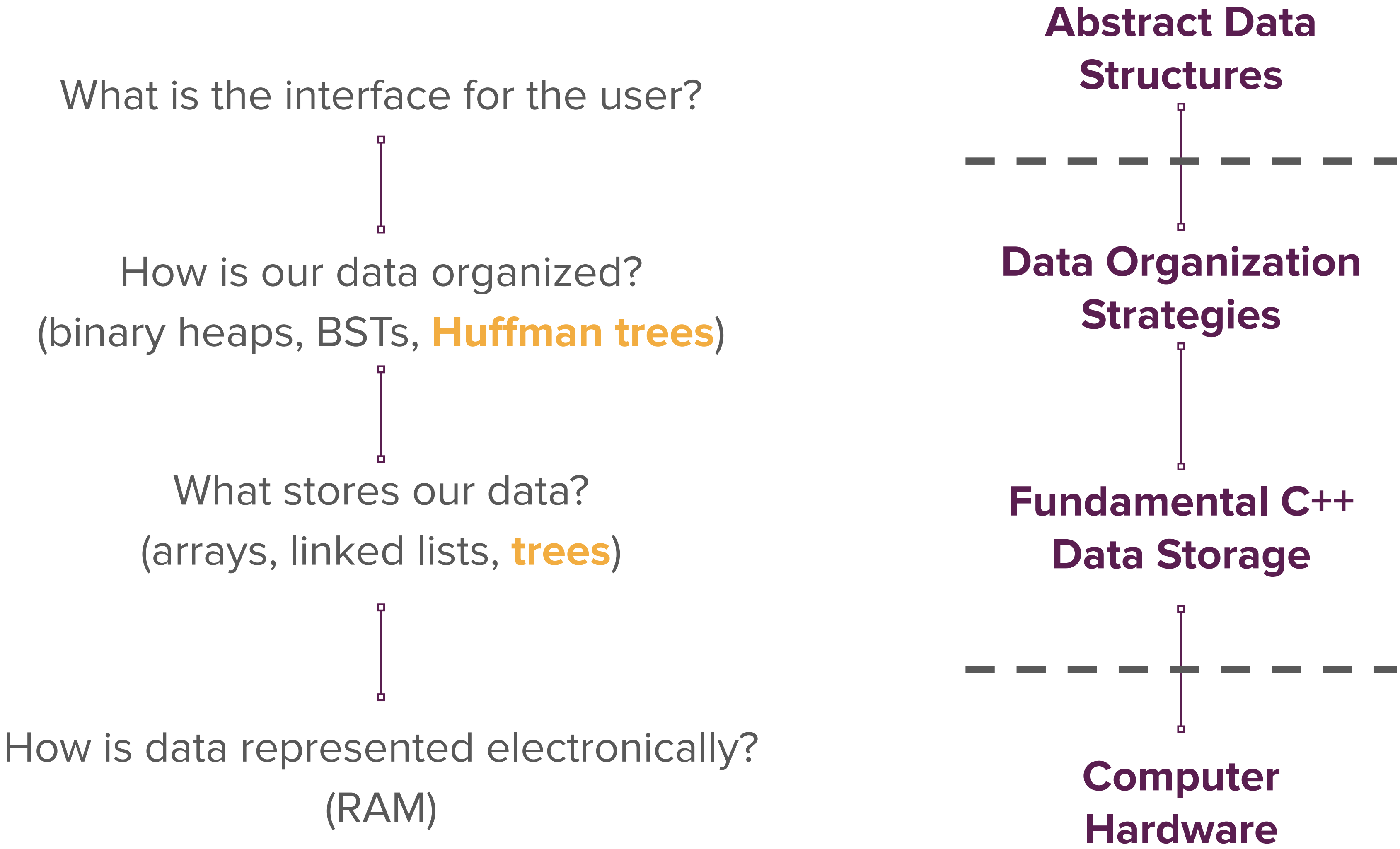
Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Levels of abstraction



What is the interface for the user?

How is our data organized?
(binary heaps, BSTs, **Huffman trees**)

What stores our data?
(arrays, linked lists, **trees**)

How is data represented electronically?
(RAM)

Abstract Data Structures

Data Organization Strategies

Fundamental C++ Data Storage

Computer Hardware

Acknowledgement: Many of the following slides were adapted from Keith Schwarz's Winter 2020 "Beyond Data Structures" lecture. Thank you Keith for having such great lecture examples!

Data Storage and Representation

How do computers store and represent data?

How do computers store and represent data?

How do computers store and represent data?

Just a Little Bit of Magic

- Digital data is stored as **sequences of 0s and 1s**.

Just a Little Bit of Magic

- Digital data is stored as **sequences of 0s and 1s**.
  - These sequences are encoded in physical devices by magnetic orientation on small (10nm!) metal particles or by trapping electrons in small gates. This is where the magic happens!

Just a Little Bit of Magic

- Digital data is stored as **sequences of 0s and 1s**.
    - These sequences are encoded in physical devices by magnetic orientation on small (10nm!) metal particles or by trapping electrons in small gates. This is where the magic happens!
- A single 0 or 1 is called a **bit**.

Just a Little Bit of Magic

- Digital data is stored as **sequences of 0s and 1s**.
  - These sequences are encoded in physical devices by magnetic orientation on small (10nm!) metal particles or by trapping electrons in small gates. This is where the magic happens!
- A single 0 or 1 is called a **bit**.
- A group of eight bits is called a **byte**.

```
00000000, 00000001, 00000010, ...
00000011, 00000100, 00000101, ...
```

Just a Little Bit of Magic

- Digital data is stored as **sequences of 0s and 1s**.
  - These sequences are encoded in physical devices by magnetic orientation on small (10nm!) metal particles or by trapping electrons in small gates. This is where the magic happens!
- 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.
  - **Good recursive backtracking practice:** Write a function to list all possible byte sequences!

Binary Representation

- The system of using sequences of 0s and 1s to represent data is called **binary**.
    - Binary can be used to encode numbers, text, images, etc.

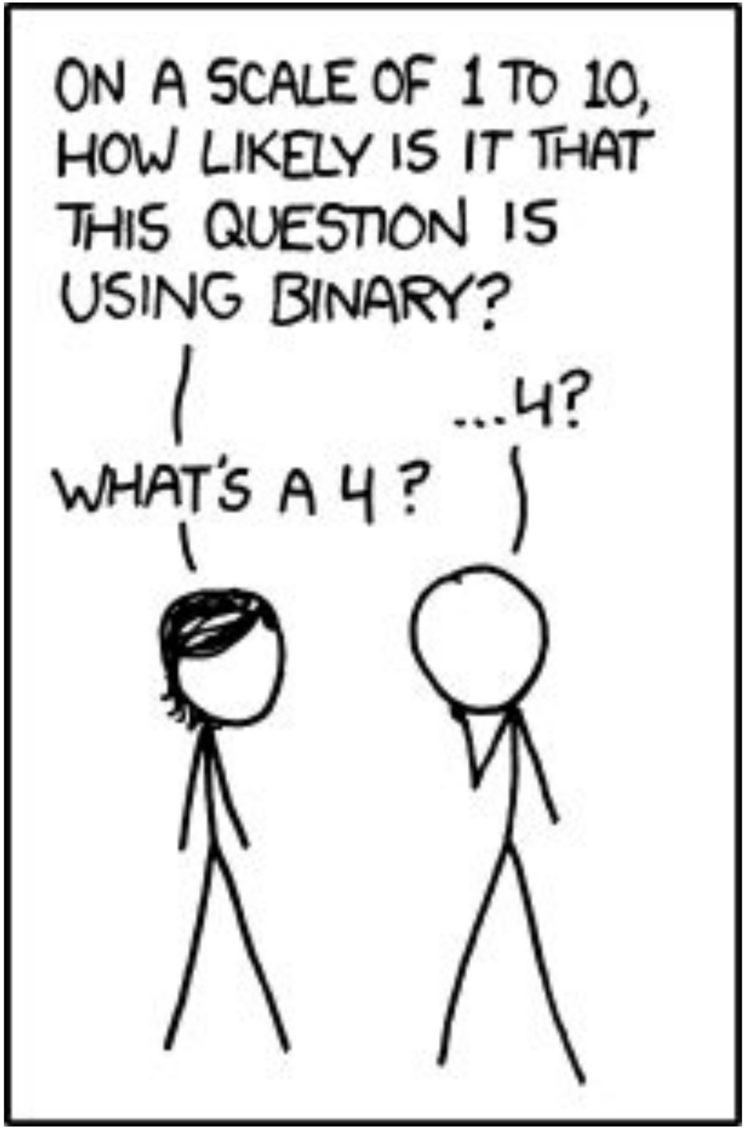
Binary Representation

- The system of using sequences of 0s and 1s to represent data is called **binary**.

- Similar to how we previously encountered hexadecimal (base-16) numbers, binary numbers can be thought of as expressed in a base-2 system.
  - To produce a number in base 2, **each digit represents a power of 2** (exactly analogous to how in base 10 each digit represents a power of 10).

Binary Representation

- The system of using sequences of 0s and 1s to represent data is called **binary**.
- Similar to how we previously encountered hexadecimal (base-16) numbers, binary numbers can be thought of as expressed in a base-2 system.
- Representing my age in different numerical systems
  - Base 10: **22** = **2** $* 10^1 +$ **2** $* 10^0 = 20 + 2 =$ **22**
  - Base 2: **10110** = **1** $* 2^4 +$ **0** $* 2^3 +$ **1** $* 2^2 +$ **1** $* 2^1 +$ **0** $* 2^0 = 16 + 4 + 2 =$ **22**


ON A SCALE OF 1 TO 10, HOW LIKELY IS IT THAT THIS QUESTION IS USING BINARY?
WHAT'S A 4?
...4?

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc.

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc.

- However, we just said that computers require everything to be written as **zeros and ones**.

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc.
- However, we just said that 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**.

Representing Text

- We think of strings as being made of characters representing letters, numbers, emojis, etc.
- However, we just said that 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.

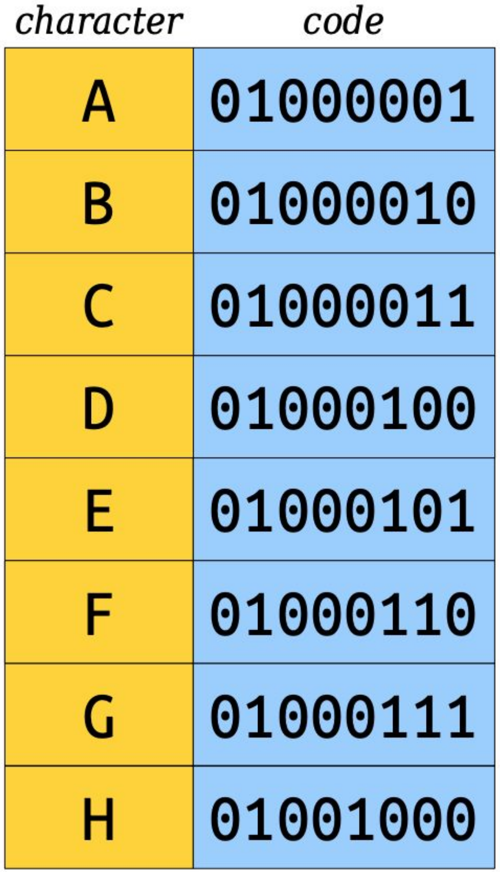
- For example, the letter A is represented by the byte **`01000001`** (whose numerical representation is 65). You can still see this in C++:

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

ASCII Mystery: `010000100100000101000111`

ASCII Mystery: `010000100100000101000111`

- Here’s a small segment from the ASCII encodings for characters.



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

ASCII Mystery: `0100000100100000101000111`

- Here’s a small segment from the ASCII encodings for characters.

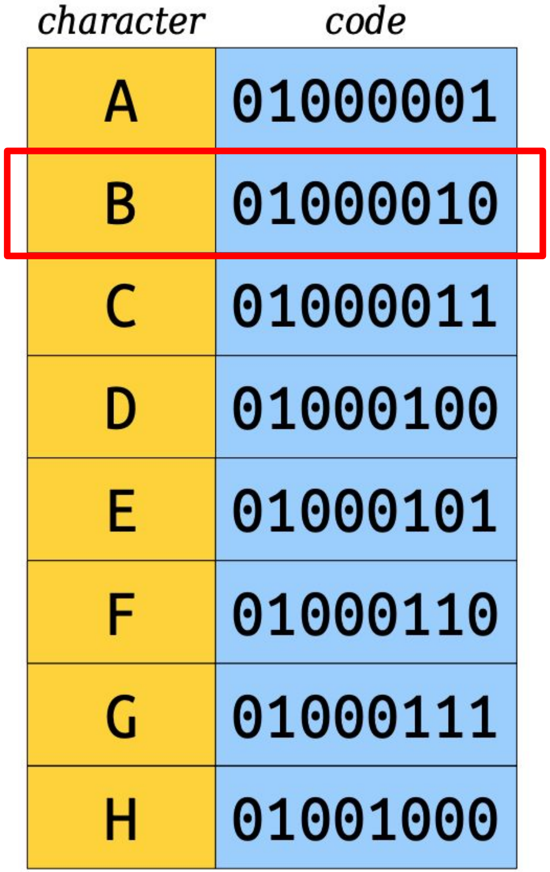
- What is the mystery word in the title of this slide?

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

ASCII Mystery: 010000100100000101000111

- Here’s a small segment from the ASCII encodings for characters.

- What is the mystery word in the title of this slide?

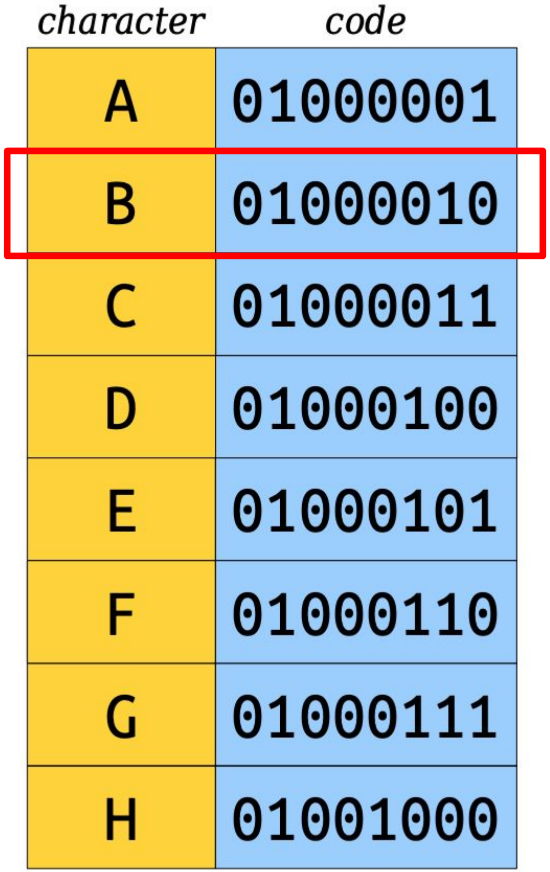


character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

ASCII Mystery: **B** `0100000101000111`

- Here’s a small segment from the ASCII encodings for characters.

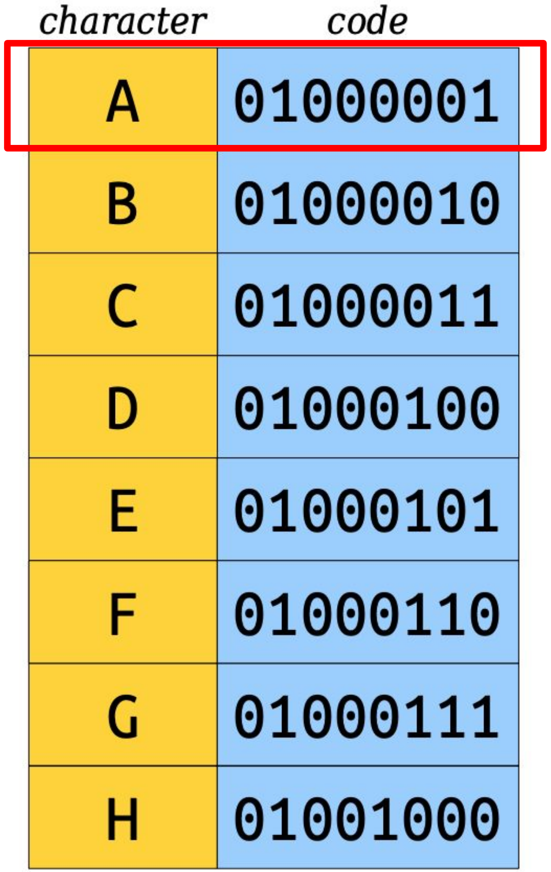
- What is the mystery word in the title of this slide?



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

ASCII Mystery: **B 0100000101000111**

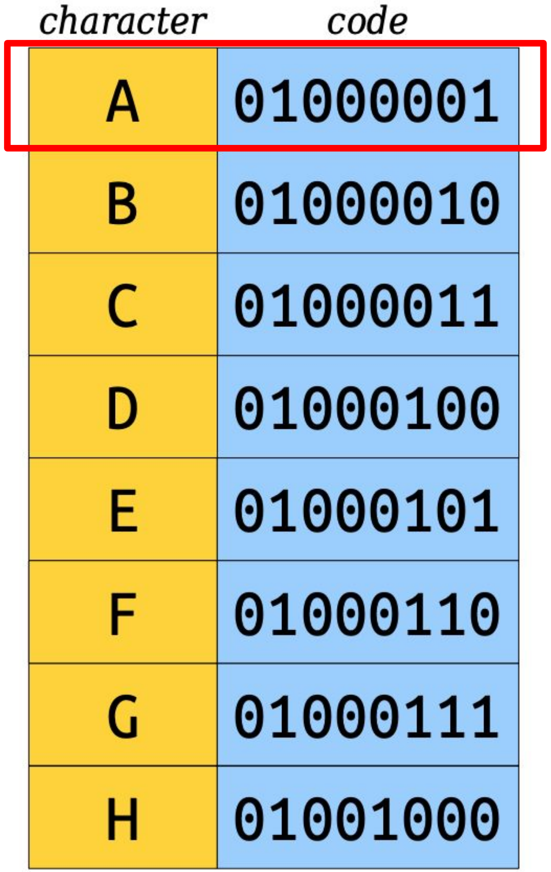
- Here’s a small segment from the ASCII encodings for characters.
- What is the mystery word in the title of this slide?



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

ASCII Mystery: **B A 01000111**

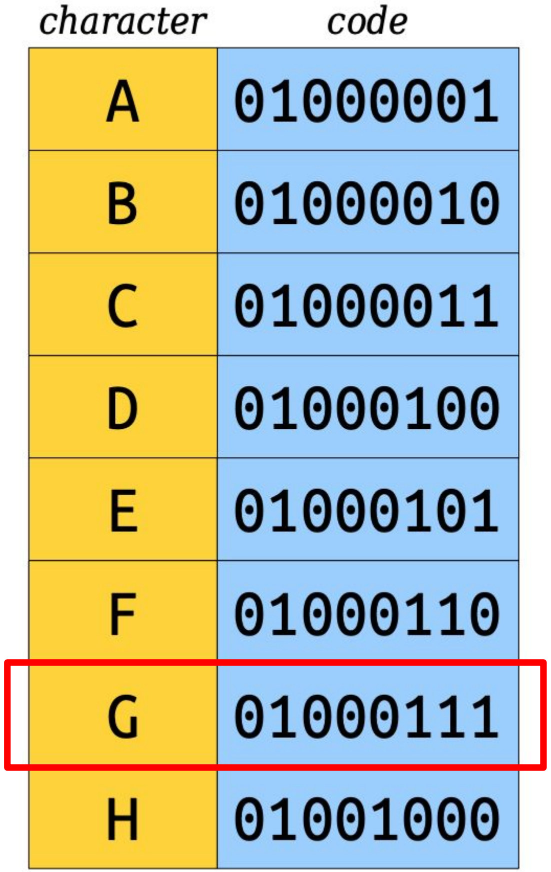
- Here's a small segment from the ASCII encodings for characters.
- What is the mystery word in the title of this slide?



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

ASCII Mystery: **B A 01000111**

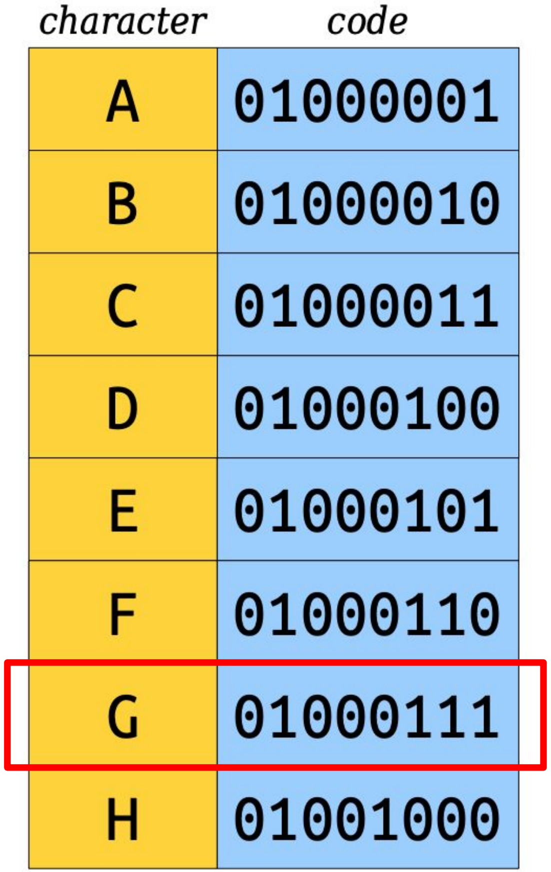
- Here’s a small segment from the ASCII encodings for characters.
- What is the mystery word in the title of this slide?



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

ASCII Mystery: **`B A G`**

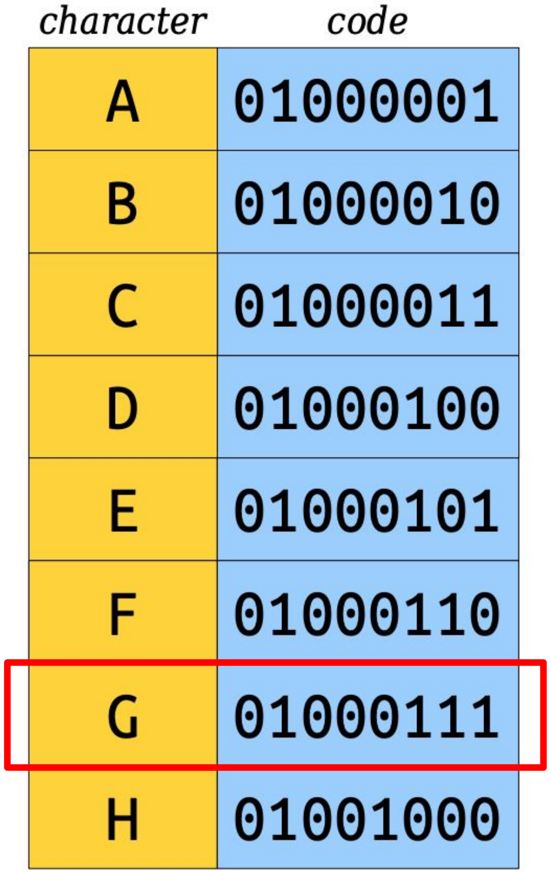
- Here's a small segment from the ASCII encodings for characters.
- What is the mystery word in the title of this slide?



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

ASCII Mystery: **B A G**

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



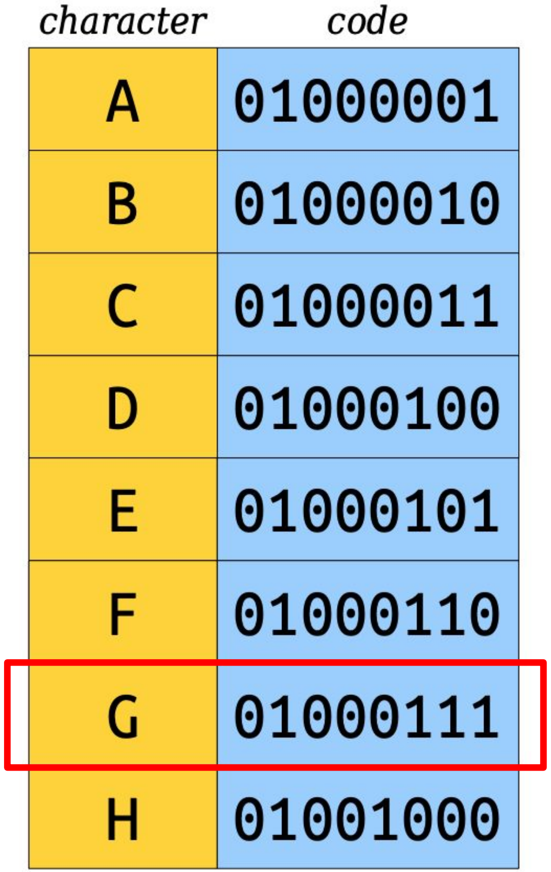
character	*code*
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110
G	01000111
H	01001000

ASCII Mystery: **B A G**

- Here’s a small segment from the ASCII encodings for characters.

- What is the mystery word in the title of this slide?

- Thus, in the computer's eyes, "BAG" is equivalent to the bit sequence **`010000100100000101000111`**



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?

The Star of Today's Show

The Star of Today's Show

The Star of Today's Show

KIRK'S DIKDIK

A Different Encoding

- ASCII uses one byte per character. There are 256 possible bytes.

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.

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`**.

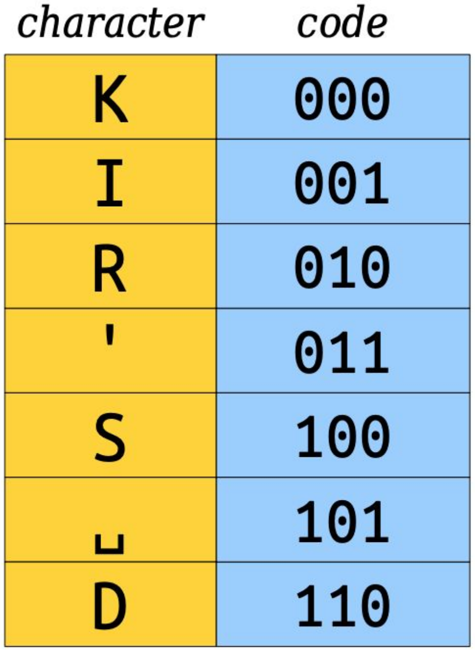
A Different Encoding

- 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`**.

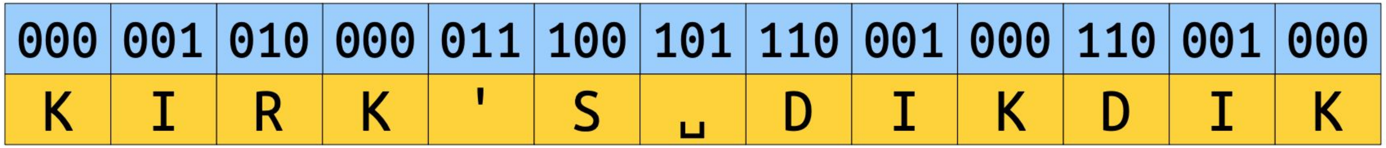
character	*code*
K	000
I	001
R	010
'	011
S	100
␣	101
D	110

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`**.



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

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**. That's a big improvement!

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`

The Journey Ahead

- 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.

Compression Algorithms

Today's Main 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.

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that are have widespread prevalence and importance.

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that are have widespread prevalence and importance.

- In particular, we are interested in algorithms that provide **lossless compression** on a stream of characters or other data.
    - Lossless compression means that we make the amount of data smaller without losing any of the details, and we can decompress the data to exactly the same as it was before compression.

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that are have widespread prevalence and importance.

- In particular, we are interested in algorithms that provide **lossless compression** on a stream of characters or other data.

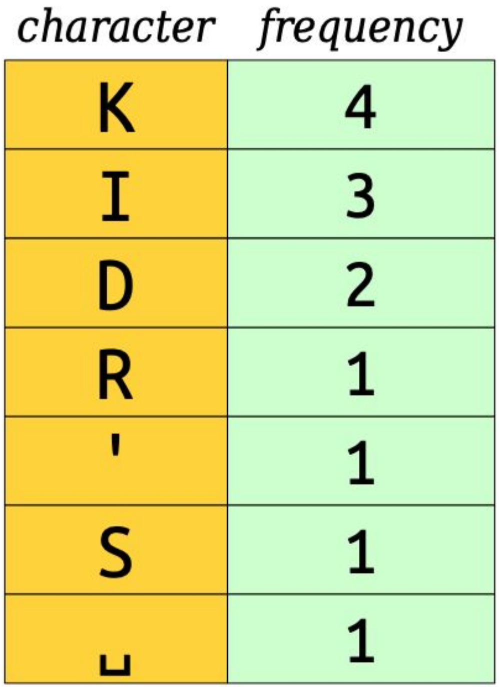
- Virtually everything that you do online involves data compression.
  - When you visit a website, download a file, or transmit video/audio, the data is **compressed** when sending and **decompressed** when receiving.
  - The video stream you're watching on Zoom right now has a compression of roughly 2000:1, meaning that a 2MB image is compressed down to 1000 bytes!

Compression Algorithms

- Compression algorithms are a whole class of real-world algorithms that are have widespread prevalence and importance.
- In particular, we are interested in algorithms that provide **lossless compression** on a stream of characters or other data.
- Virtually everything that you do online involves data compression.
- Compression algorithms **identify patterns in data** and take advantage of those patterns to come up with more efficient representations of that data!

Taking Advantage of Redundancy

- Not all letters have the same frequency in **`KIRK'S DIKDIK`**.
- The frequencies of each letter are shown to the right.
- So far, we’ve given each letter a code of the same length.
- **Key Question:** Can we give shorter encodings to more common characters?

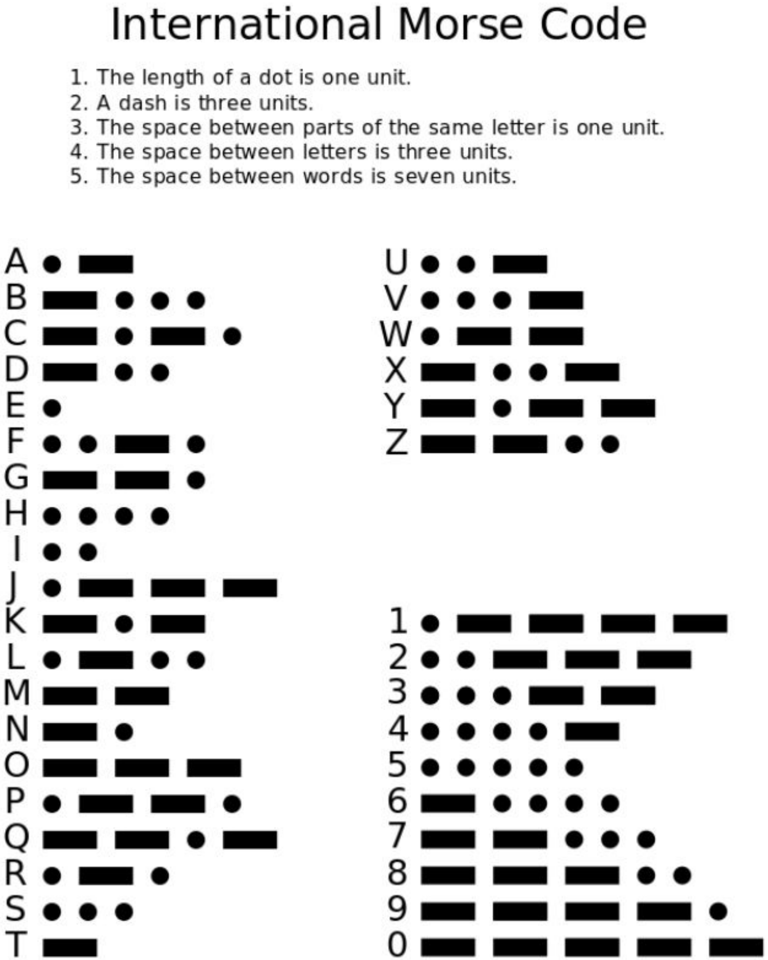


character	*frequency*
K	4
I	3
D	2
R	1
'	1
S	1
␣	1

Morse Code

- Morse Code is one coding system that makes use of this insight!

- The code for very frequent letters (e, t, a) are much shorter than the codes for very infrequent letters (q, k, j).



International Morse Code

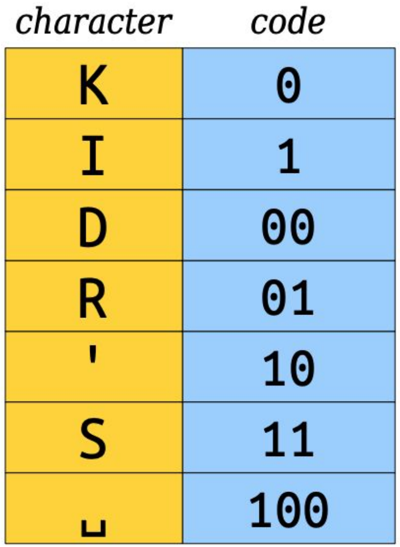
1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A ● ▬
B ▬ ● ● ●
C ▬ ● ▬ ●
D ▬ ● ●
E ●
F ● ● ▬ ●
G ▬ ▬ ●
H ● ● ● ●
I ● ●
J ● ▬ ▬ ▬
K ▬ ● ▬
L ● ▬ ● ●
M ▬ ▬
N ▬ ●
O ▬ ▬ ▬
P ● ▬ ▬ ●
Q ▬ ▬ ● ▬
R ● ▬ ●
S ● ● ●
T ▬

U ● ● ▬
V ● ● ● ▬
W ● ▬ ▬
X ▬ ● ● ▬
Y ▬ ● ▬ ▬
Z ▬ ▬ ● ●

1 ● ▬ ▬ ▬ ▬
2 ● ● ▬ ▬ ▬
3 ● ● ● ▬ ▬
4 ● ● ● ● ▬
5 ● ● ● ● ●
6 ▬ ● ● ● ●
7 ▬ ▬ ● ● ●
8 ▬ ▬ ▬ ● ●
9 ▬ ▬ ▬ ▬ ●
0 ▬ ▬ ▬ ▬ ▬

A First Attempt



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

Shorter codes for more frequent characters

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

How do we decode this if we don't know the original message?

01010101110000100010

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

01010101110000100010

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

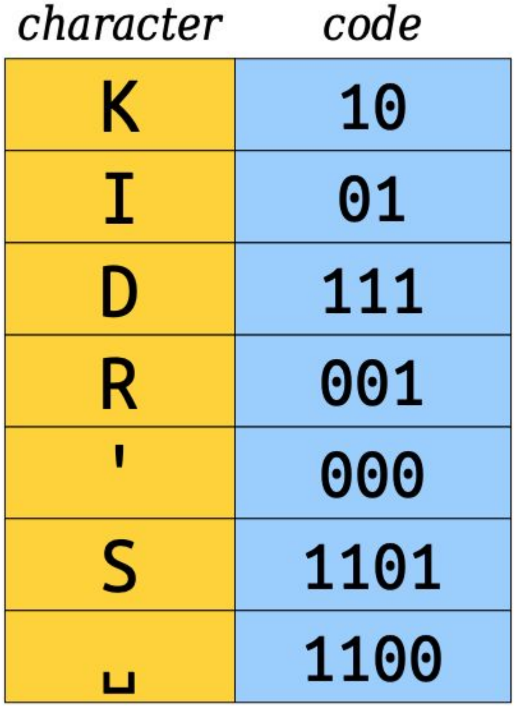
What Went Wrong?

- 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 Example

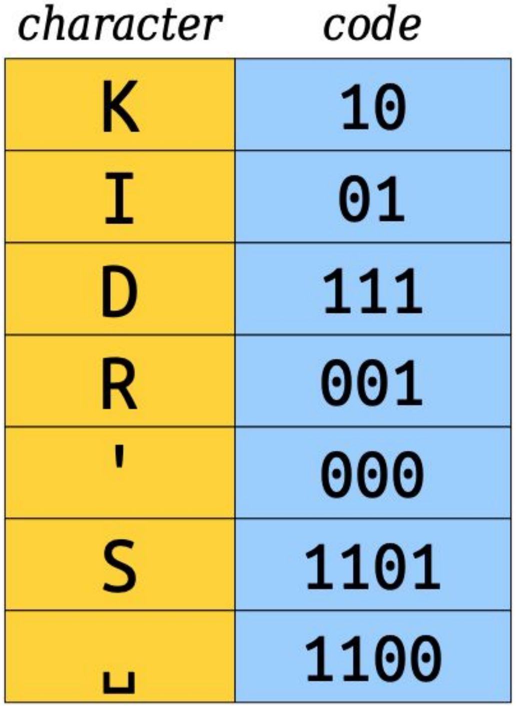
```
10010011000011011100
11101101110110
```

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 Example

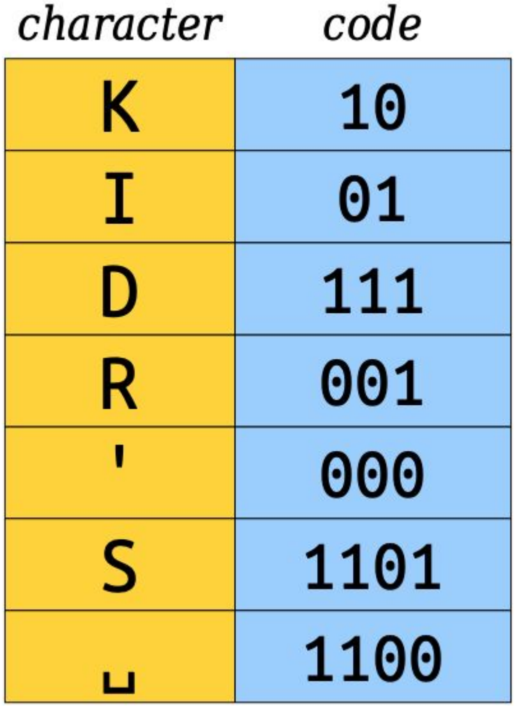
10010011000011011100
11101101110110



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

Prefix Codes Example

1001001100000110111001
1101101110110



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

Prefix Codes Example

100100110000011011100
11101101110110

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

Prefix Codes Example

100100110000011011100
11101101110110

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

10
K

Prefix Codes Example

100100110000011011100
11101101110110

10	
K	

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

Prefix Codes Example

100100110000011011100
11101101110110

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

10
K

Prefix Codes Example

10**01**0011000011011100
11101101110110

10	01
K	I

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

Prefix Codes Example

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

10010011000011011100
11101101110110

10	01
K	I

Prefix Codes Example

100100110000011011100
11101101110110

10	01
K	I

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

Prefix Codes Example

100100110000011011100
11101101110110

10	01
K	I

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

Prefix Codes Example

100100110000011011100
11101101110110

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

10	01	001
K	I	R

Prefix Codes Summary

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

  `1001001100001101110011101101110110`

- This uses just 34 bits, compared to our initial 104 (using ASCII). Wow!

- Many questions remain: Where did this code come from? How could you come up with codes like this for other strings? What makes a "good" prefix coding scheme? What does this all have to do with trees?

Prefix Codes Summary

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

  `1001001100001101110011101101110110`

- This uses just 34 bits, compared to our initial 104 (using ASCII). Wow!

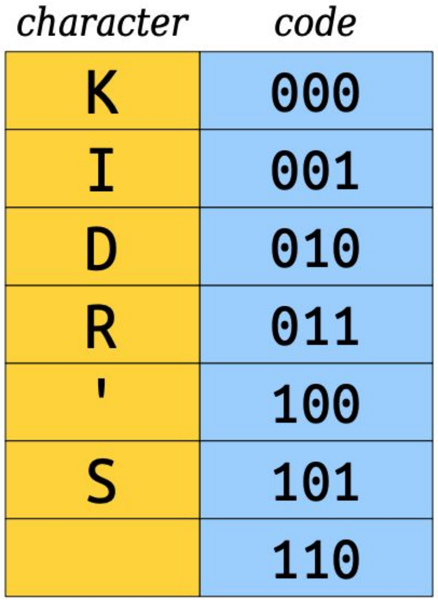
- Many questions remain: Where did this code come from? How could you come up with codes like this for other strings? What makes a "good" prefix coding scheme? **What does this all have to do with trees?**

The Trees are Back in Town

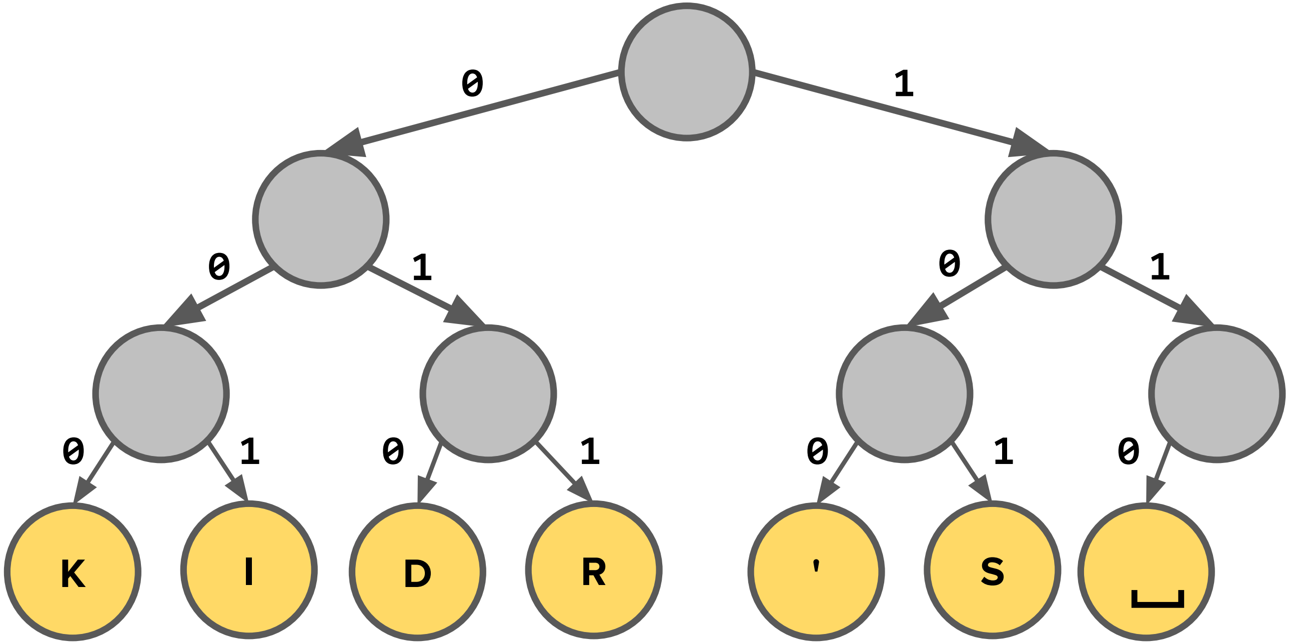
- **Main Insight:** We can represent a prefix coding scheme with a binary tree! This special type of binary tree is called a **coding tree**.

The Trees are Back in Town

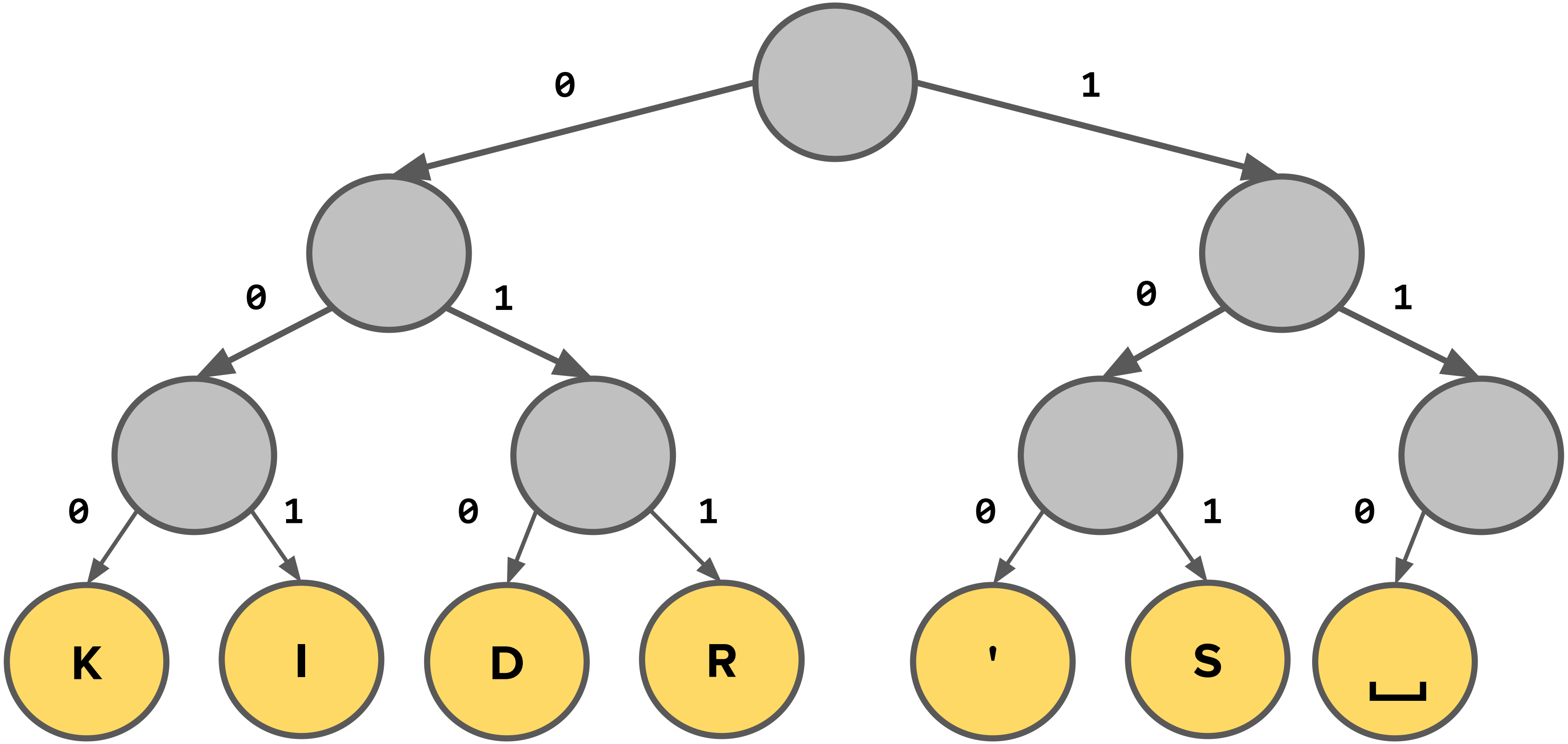
- **Main Insight:** We can represent a prefix coding scheme with a binary tree! This special type of binary tree is called a **coding tree**.



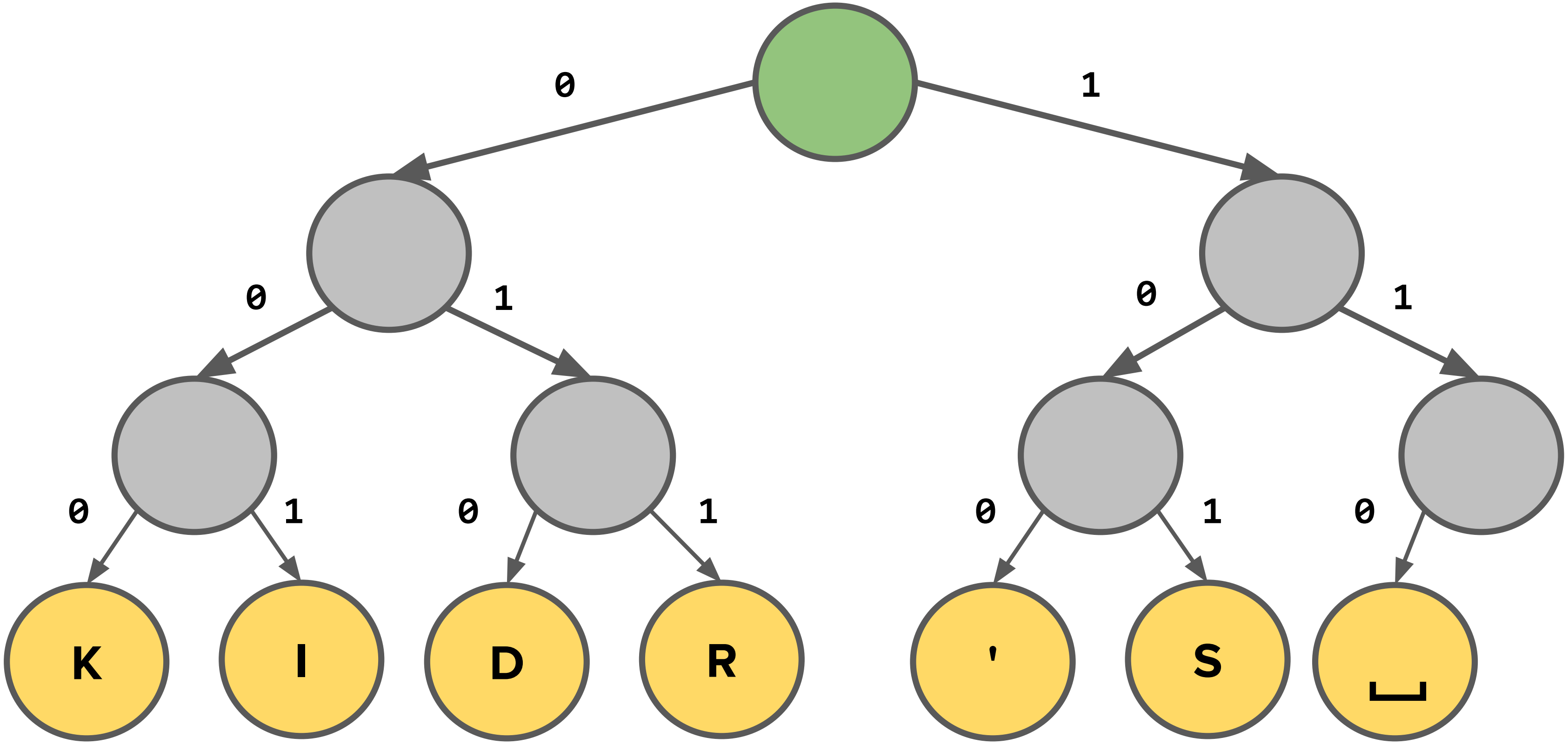
character	*code*
K	000
I	001
D	010
R	011
'	100
S	101
	110



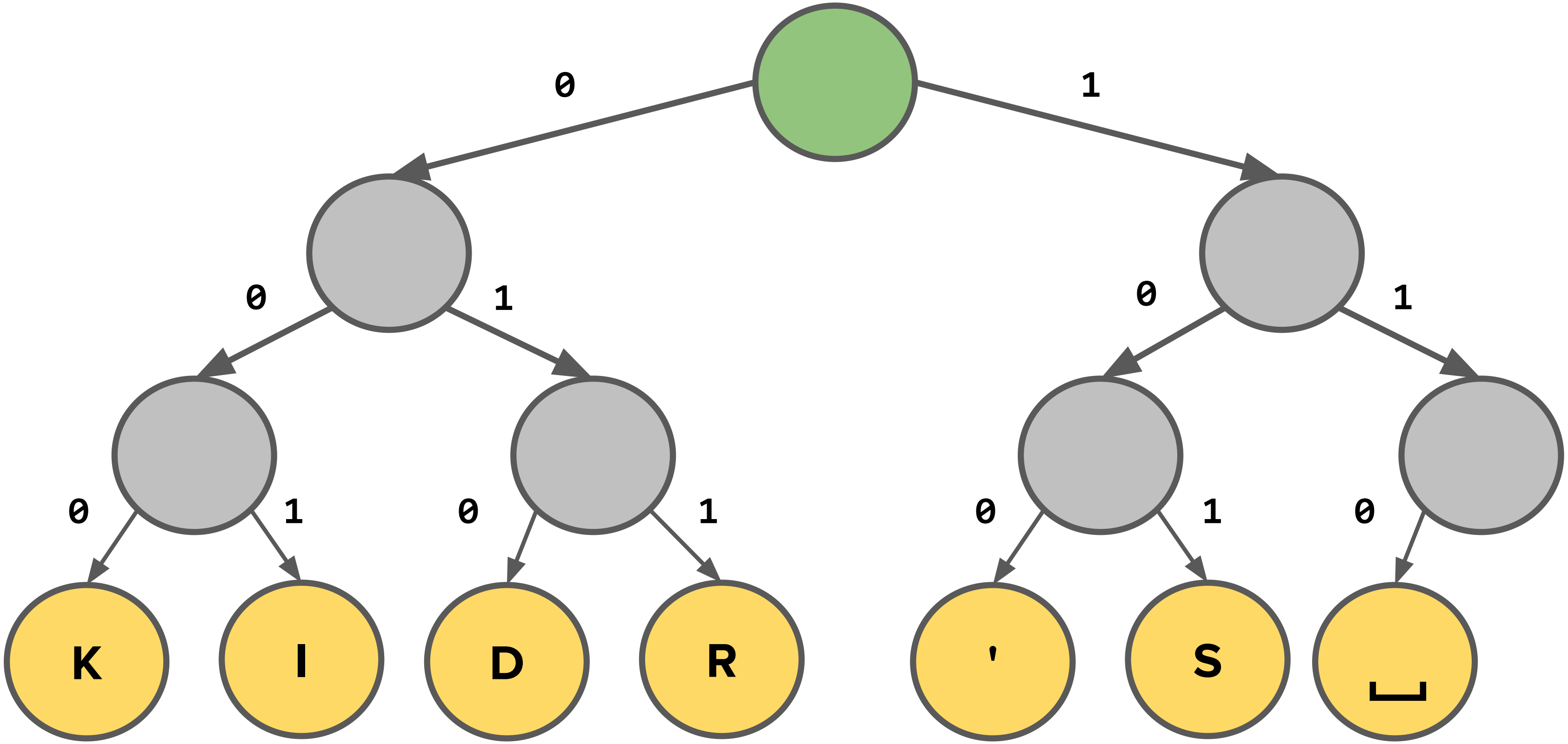
Prefix Coding Mystery: **101000001**



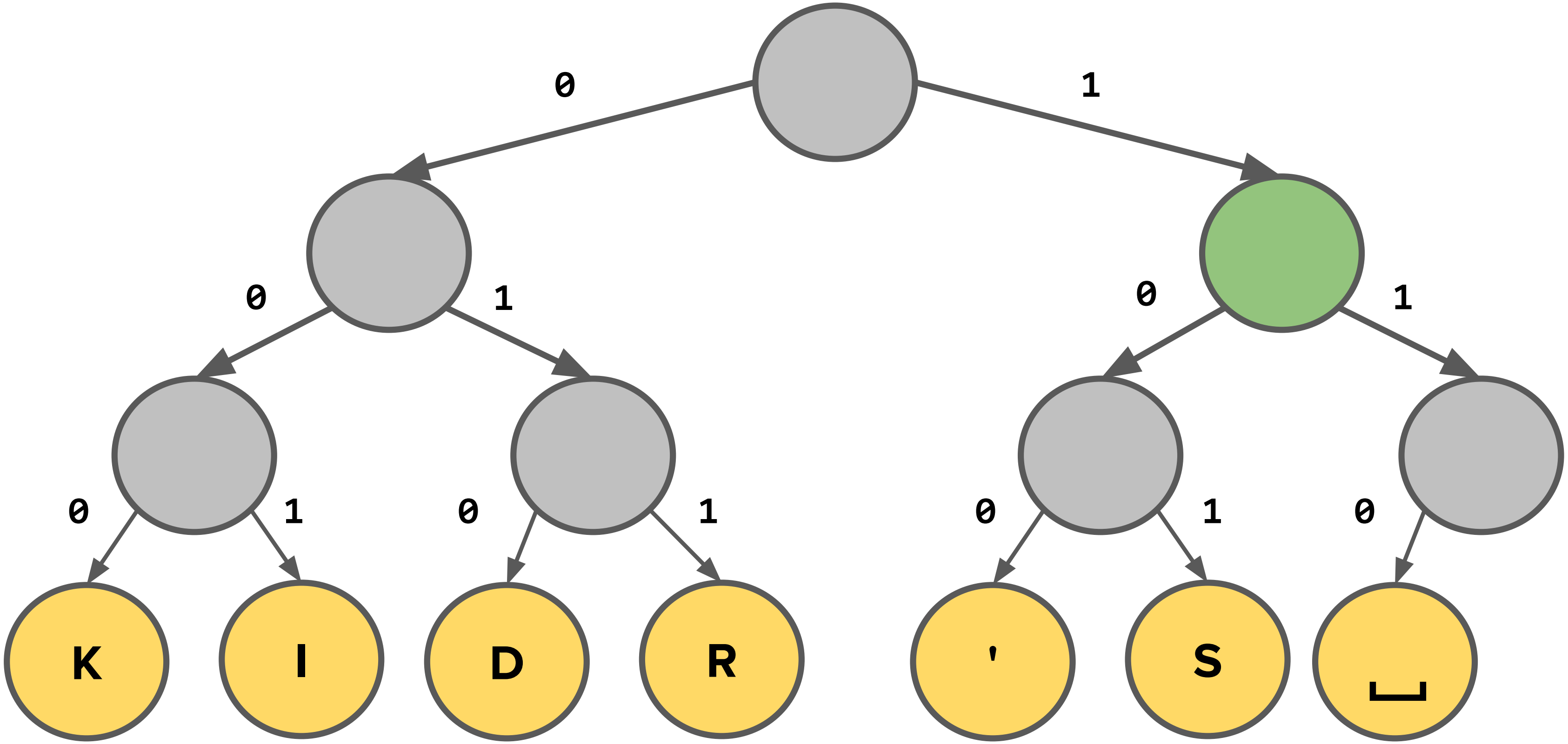
Prefix Coding Mystery: **101000001**



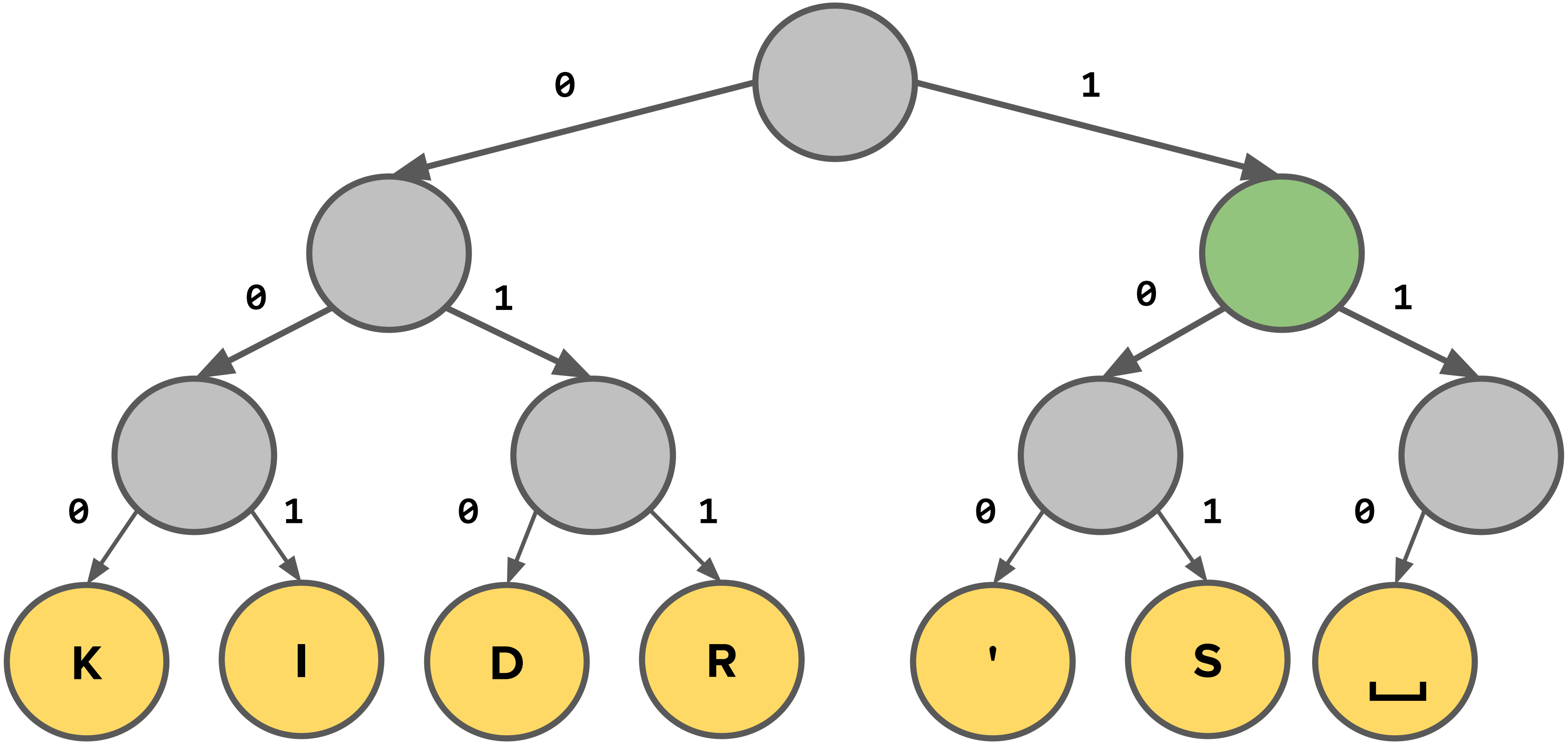
Prefix Coding Mystery: 101000001



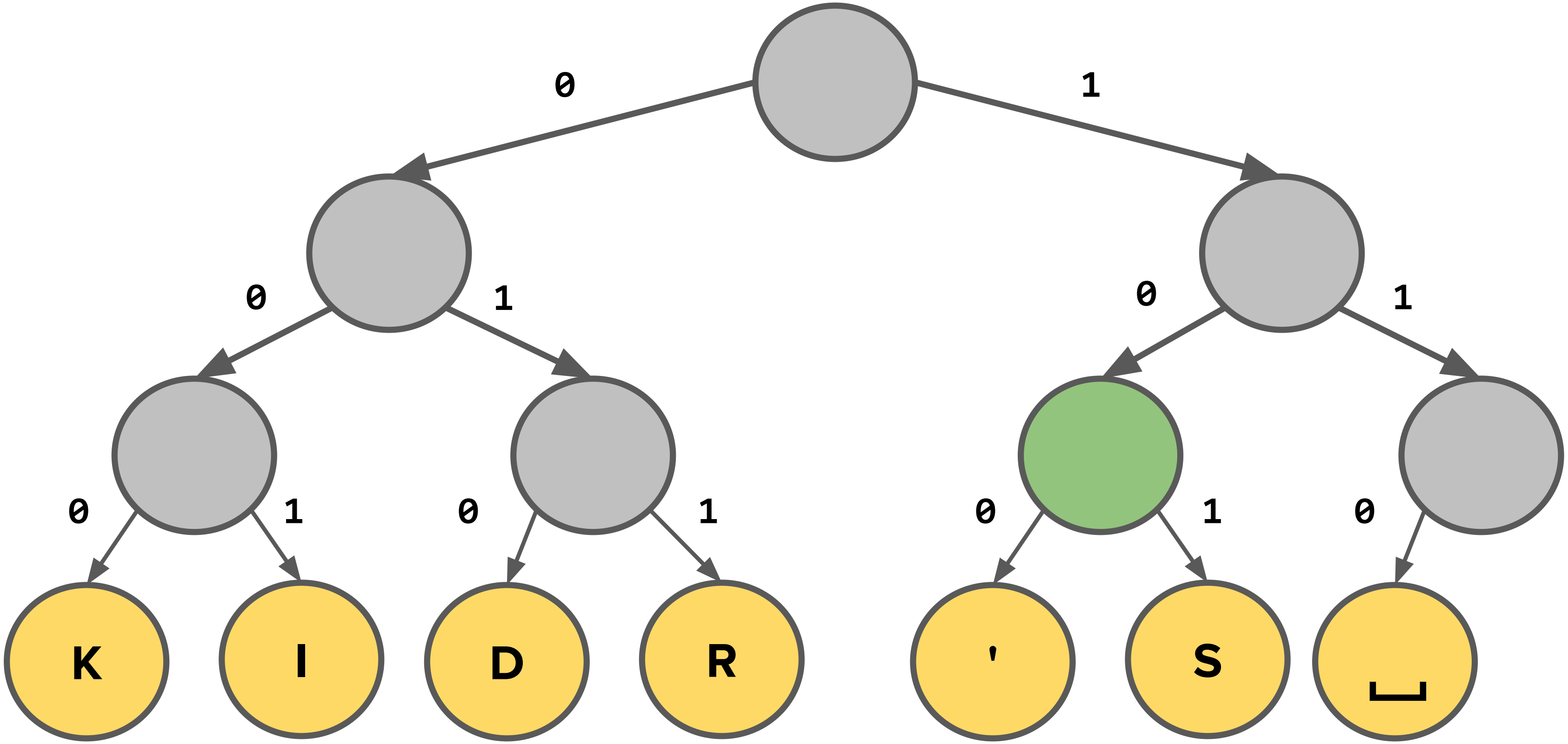
Prefix Coding Mystery: 101000001



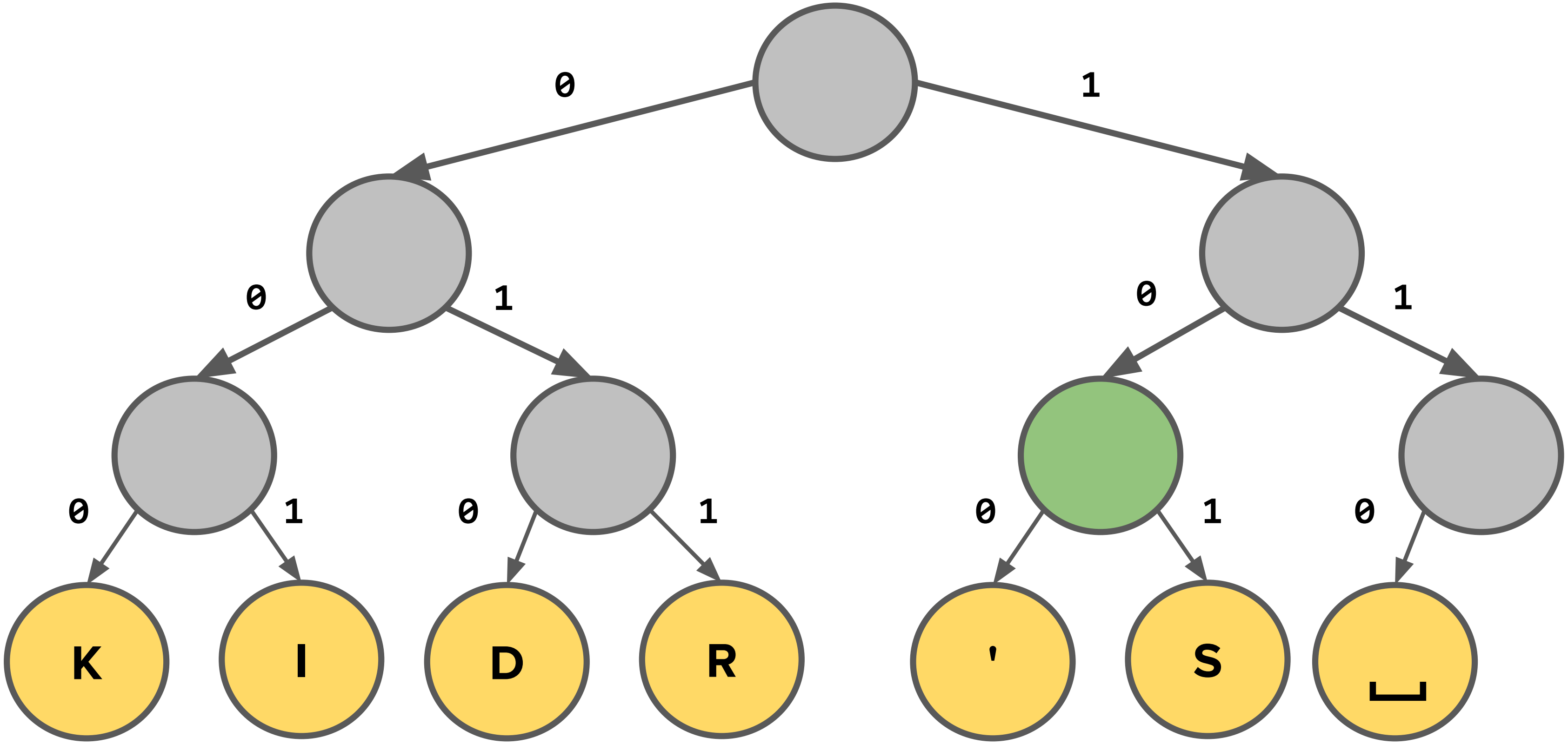
Prefix Coding Mystery: 101000001



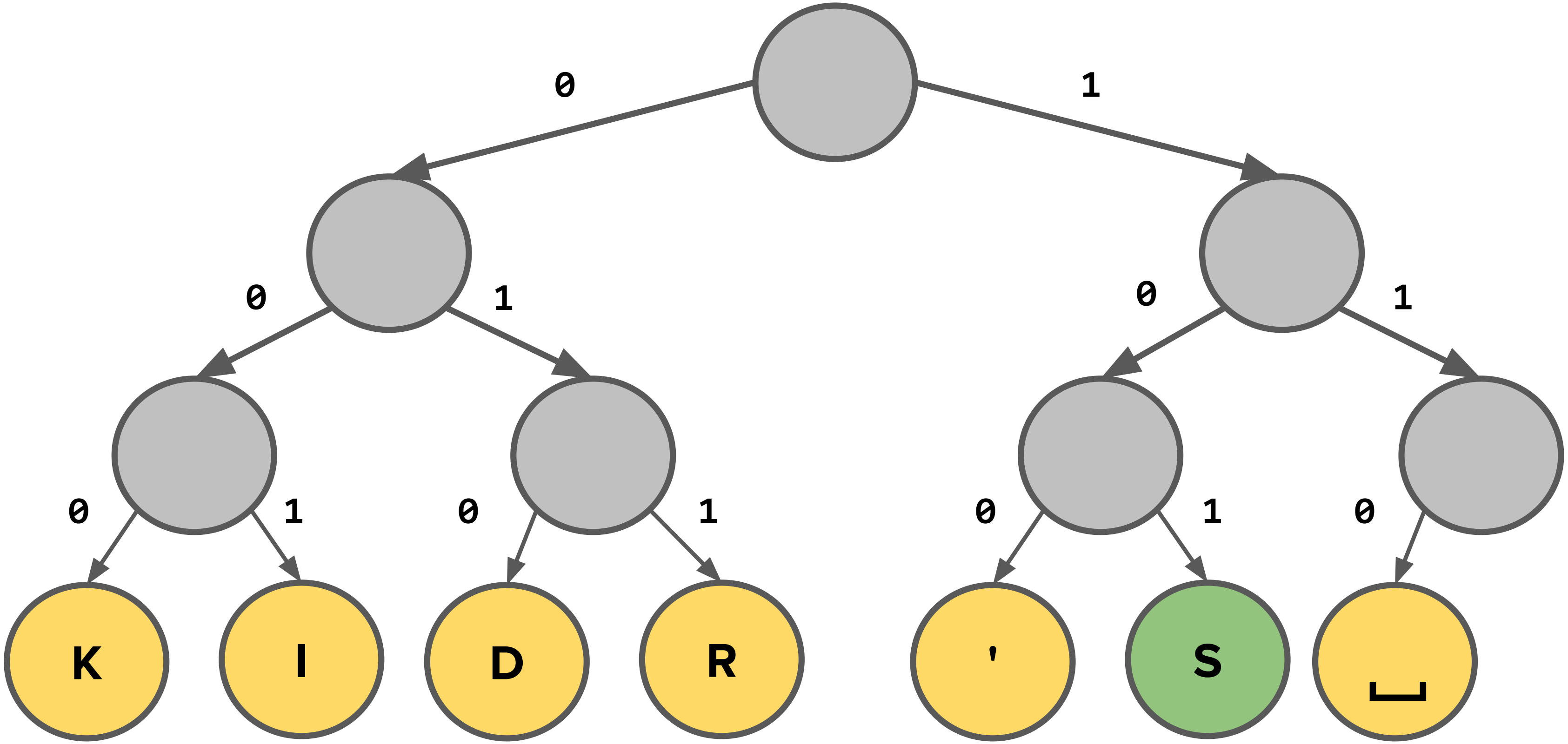
Prefix Coding Mystery: **10**1000001



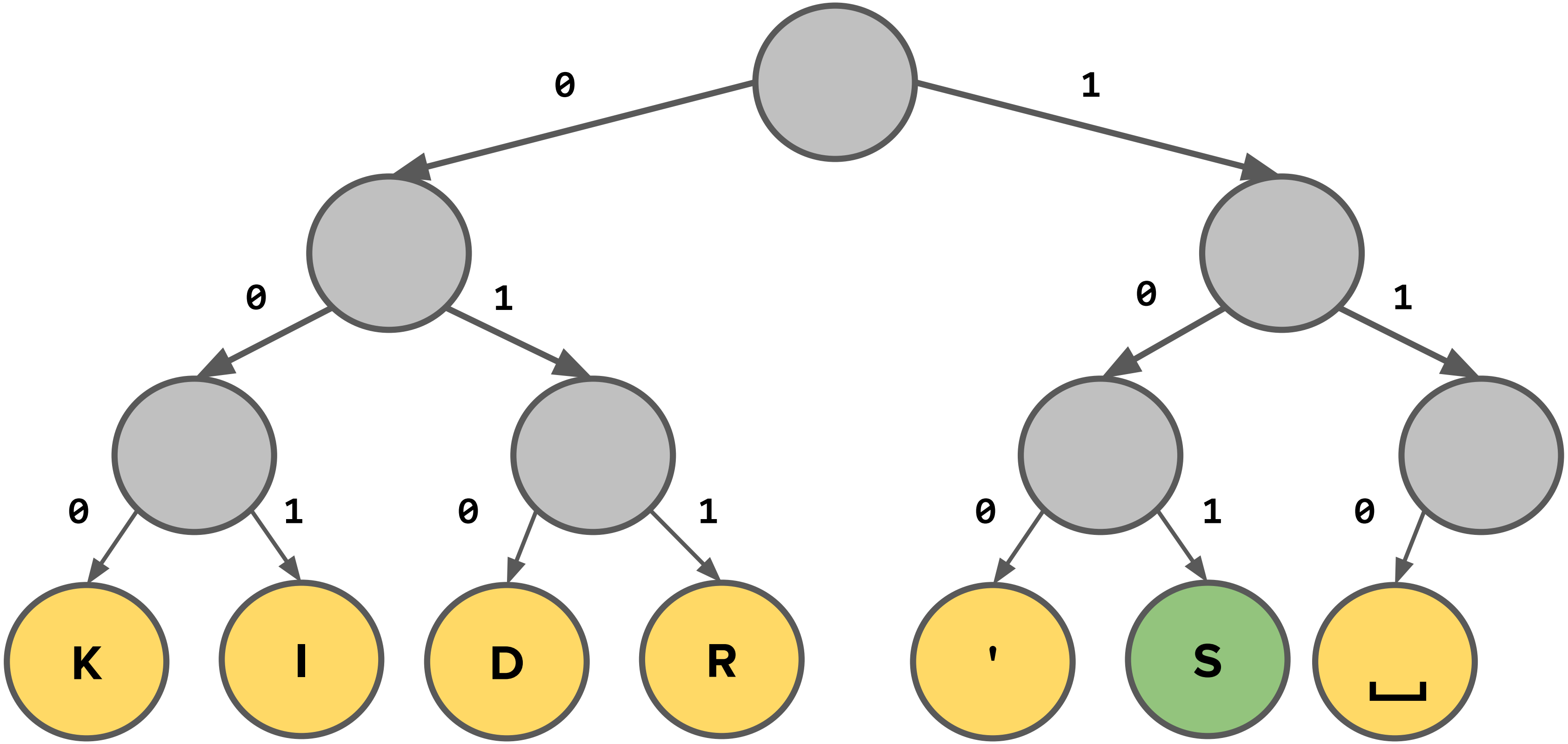
Prefix Coding Mystery: 101000001



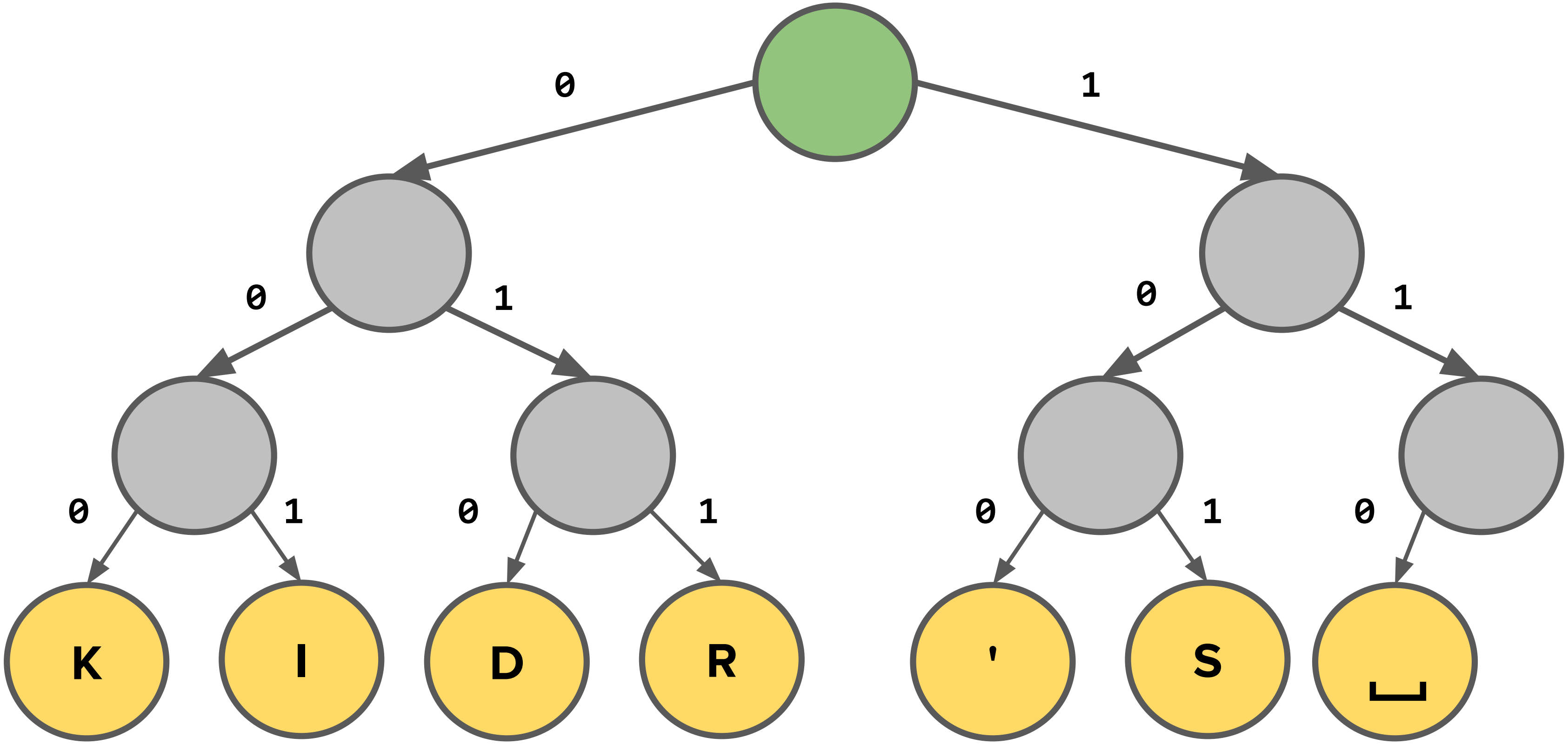
Prefix Coding Mystery: **101**000001



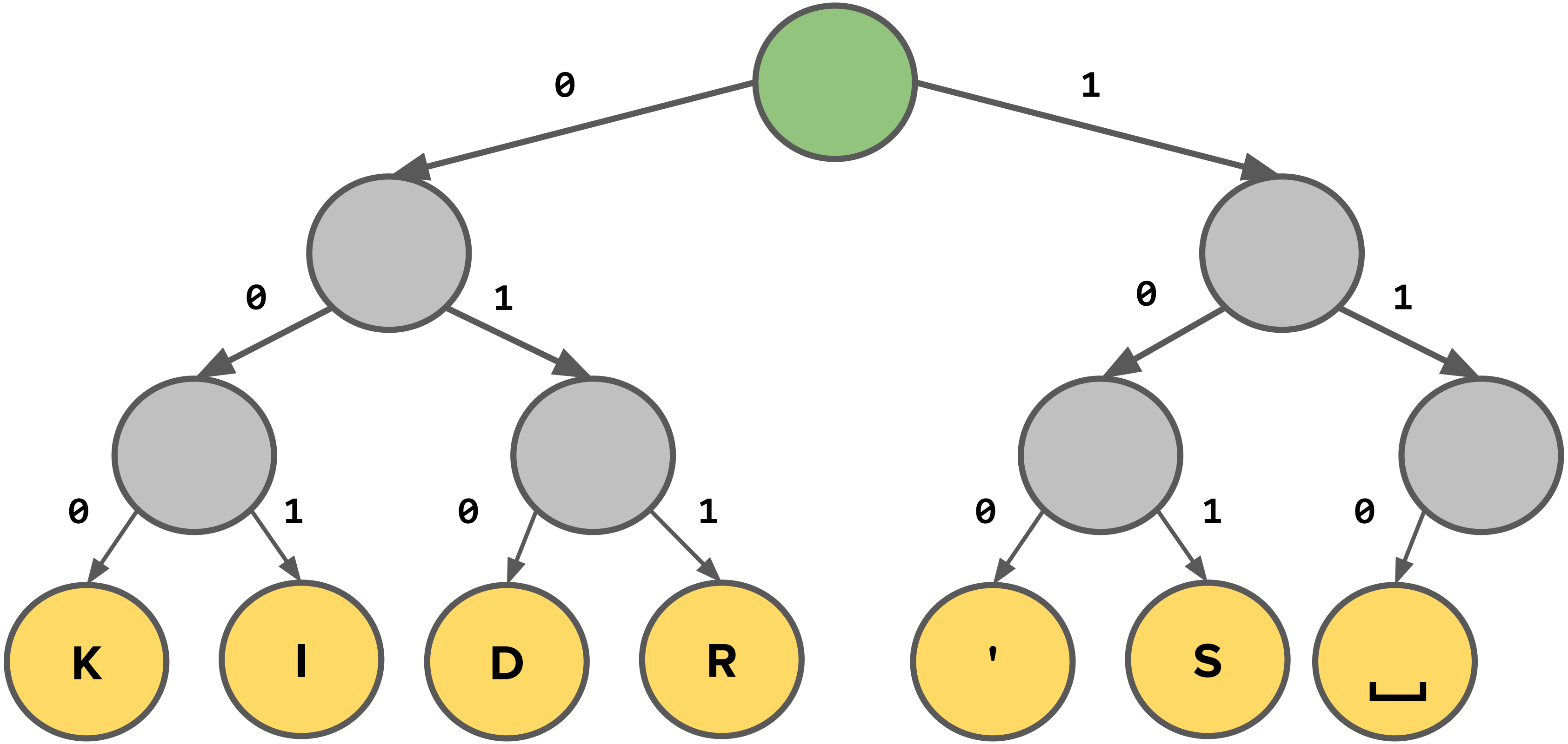
Prefix Coding Mystery: S `000001`



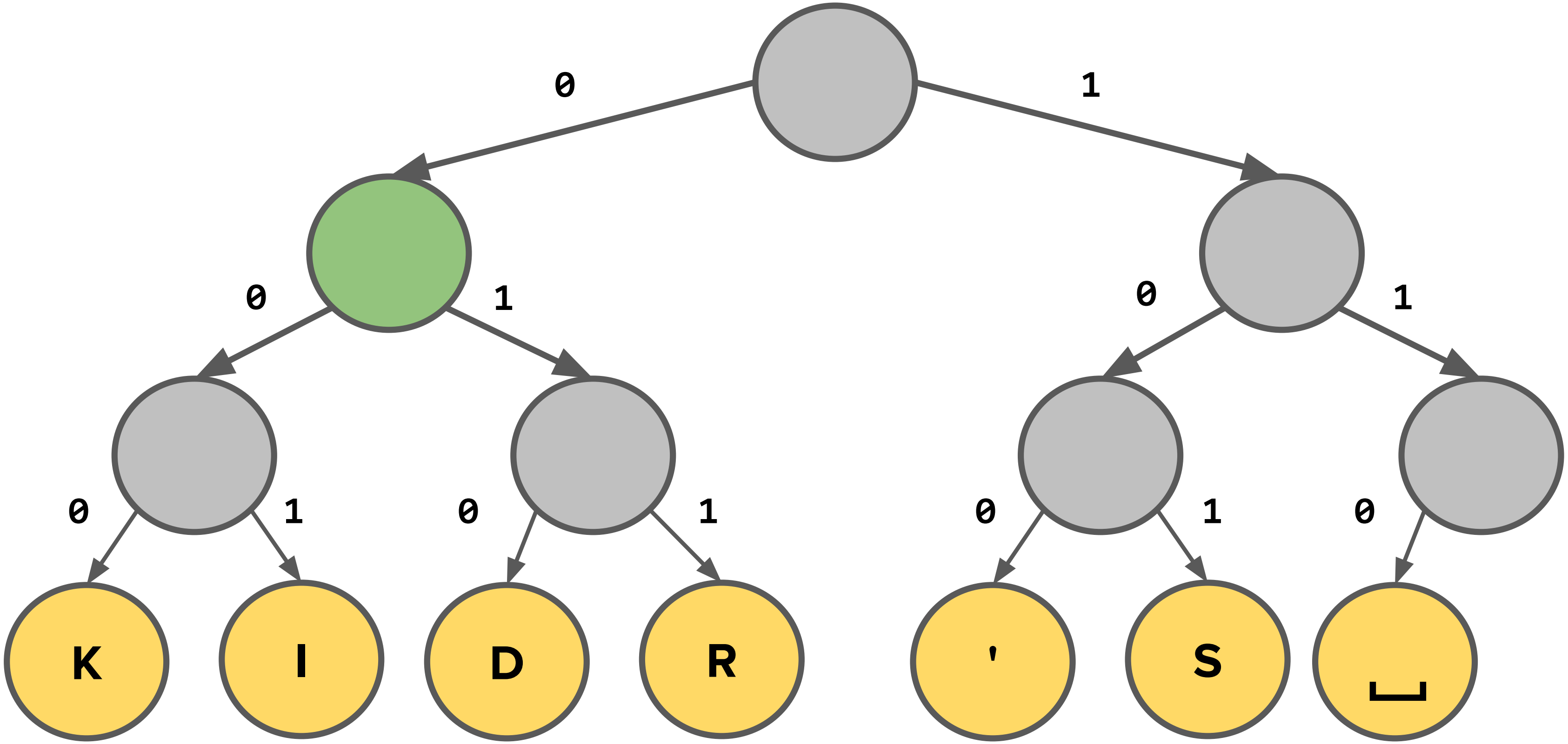
Prefix Coding Mystery: **`S 000001`**



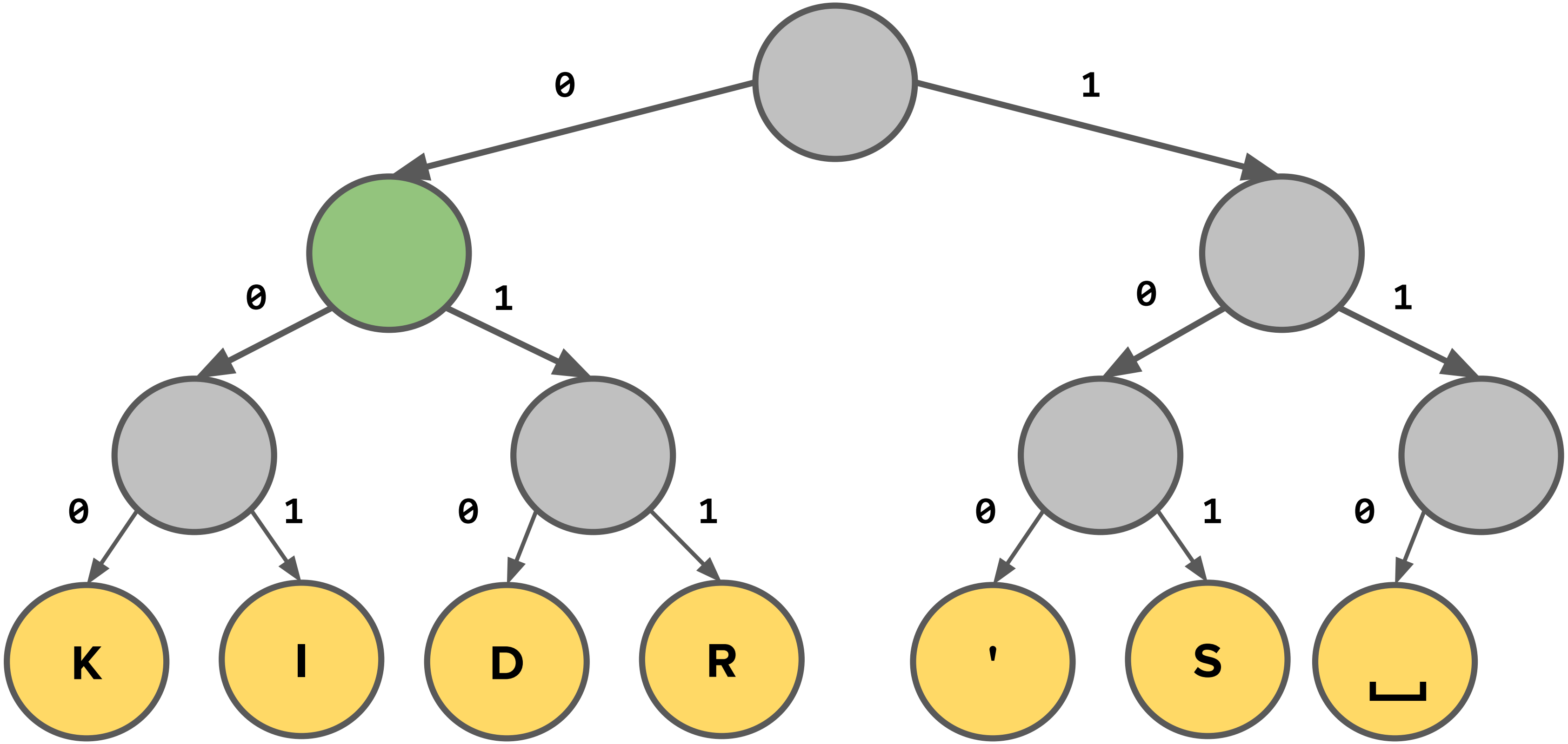
Prefix Coding Mystery: S 000001



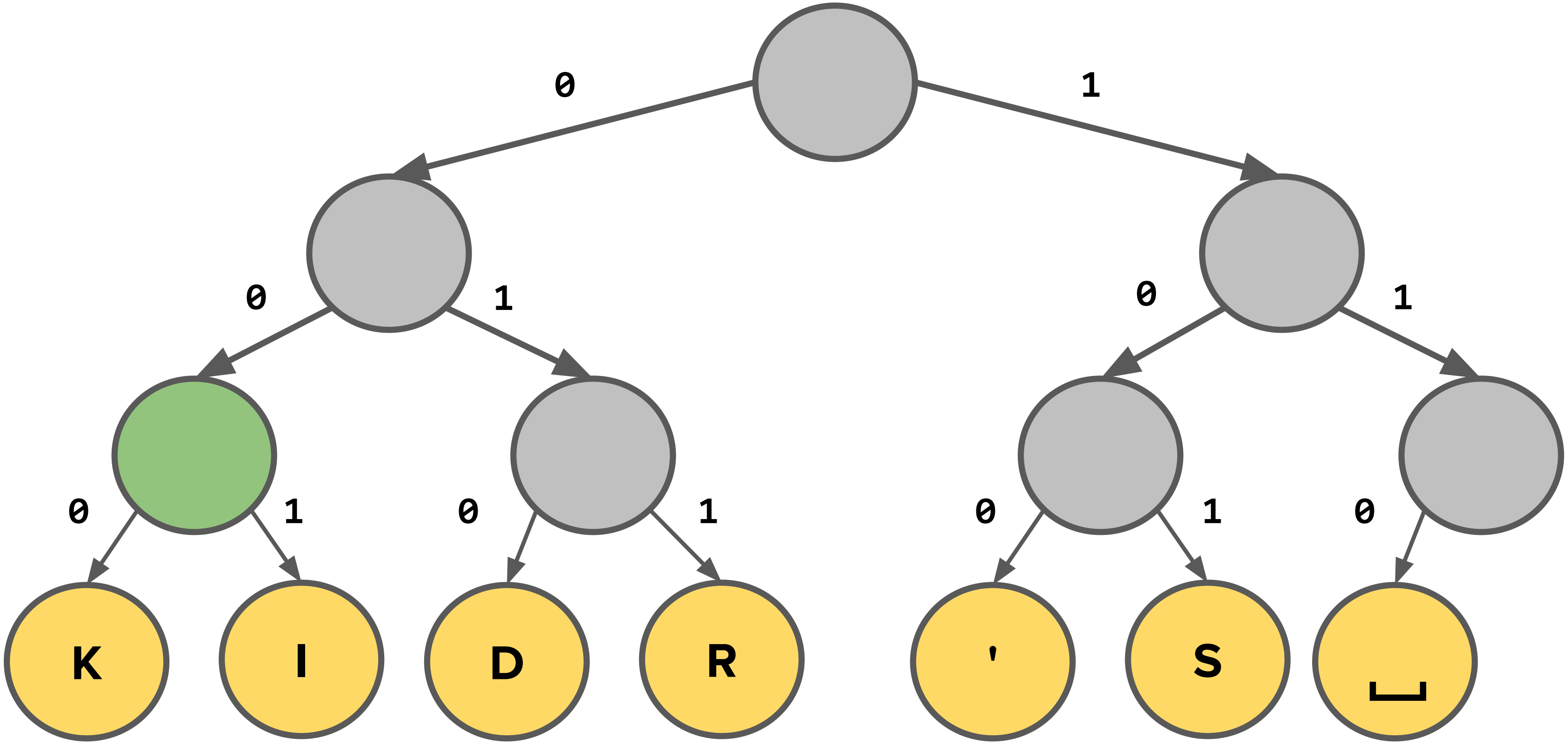
Prefix Coding Mystery: **S** `000001`



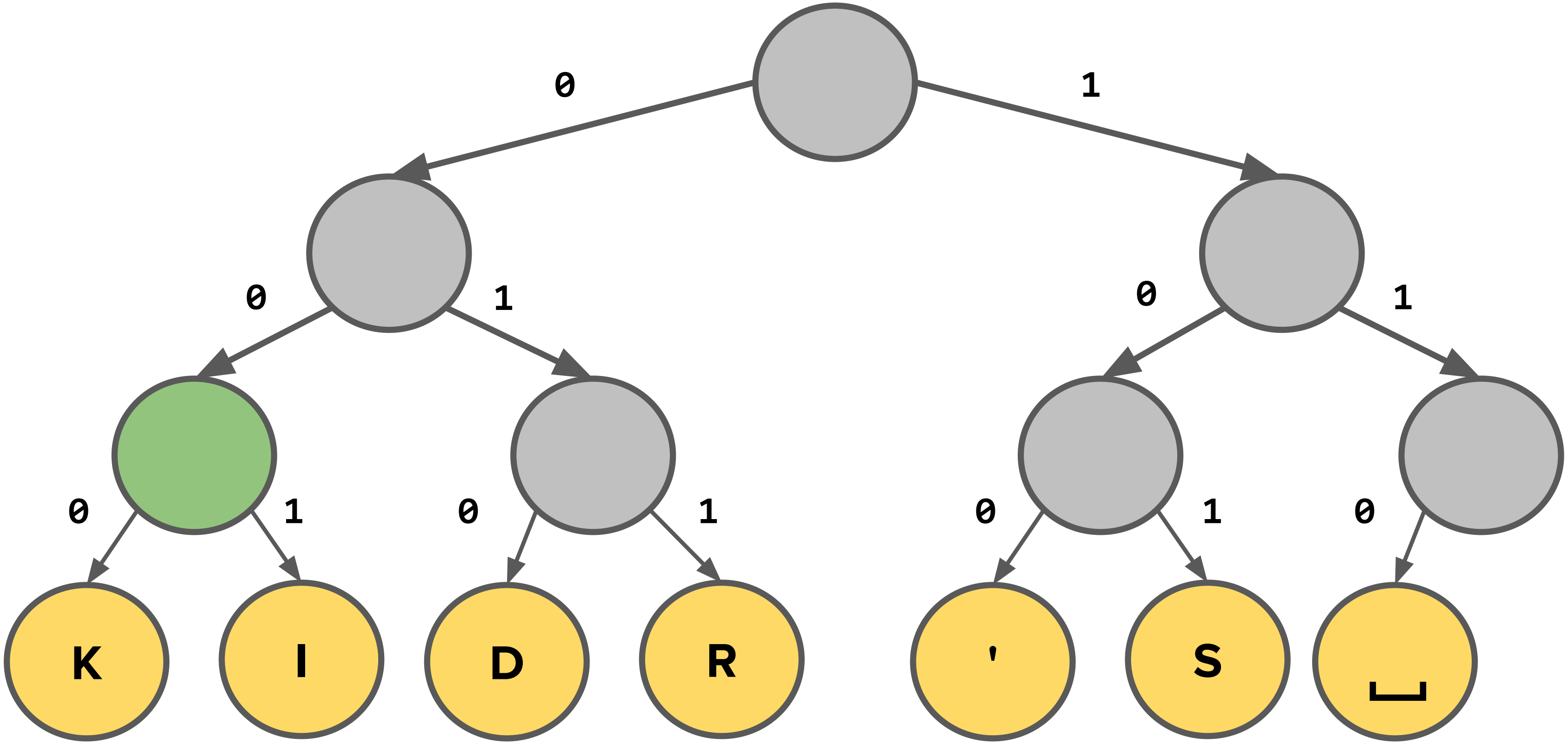
Prefix Coding Mystery: **S** `000001`



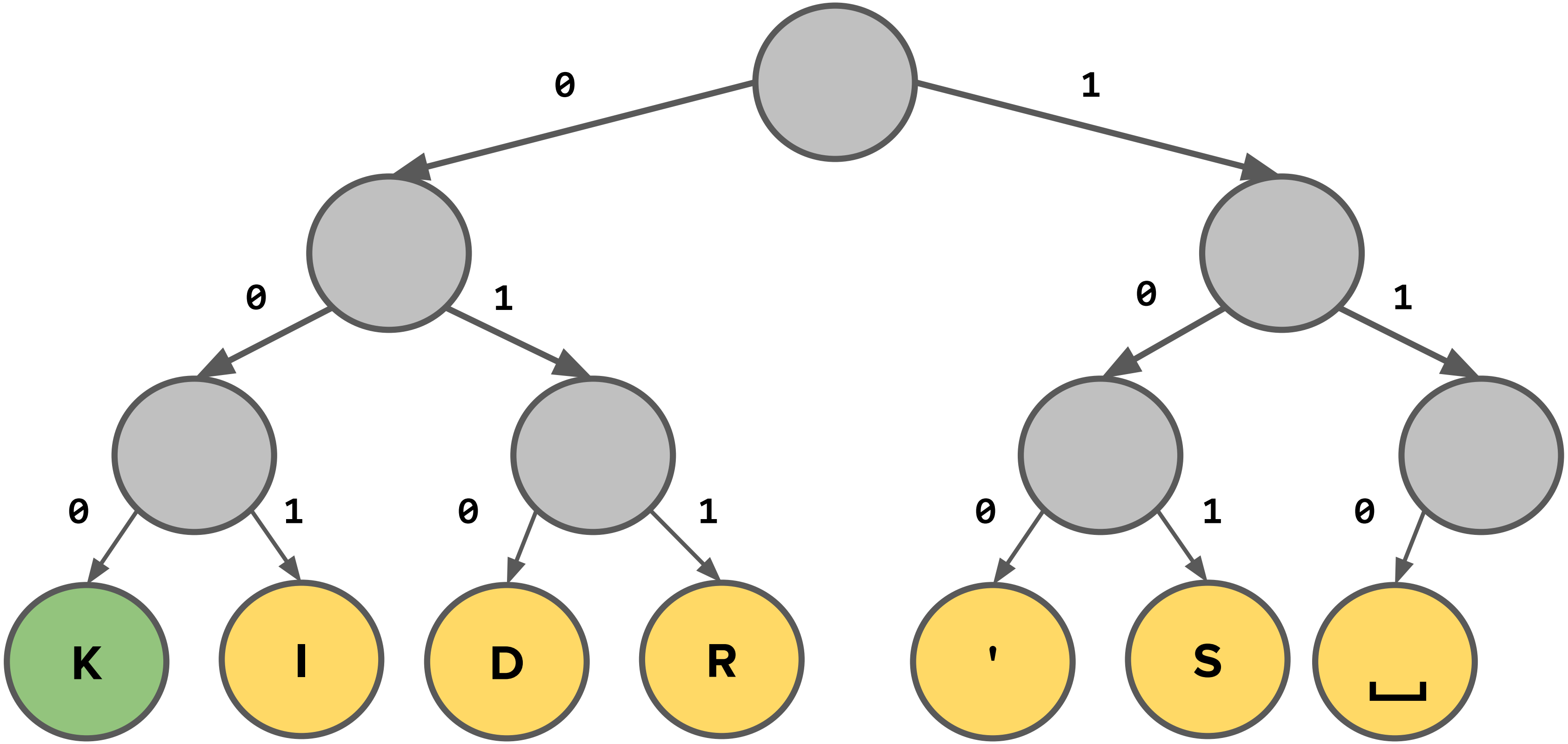
Prefix Coding Mystery: S 000001



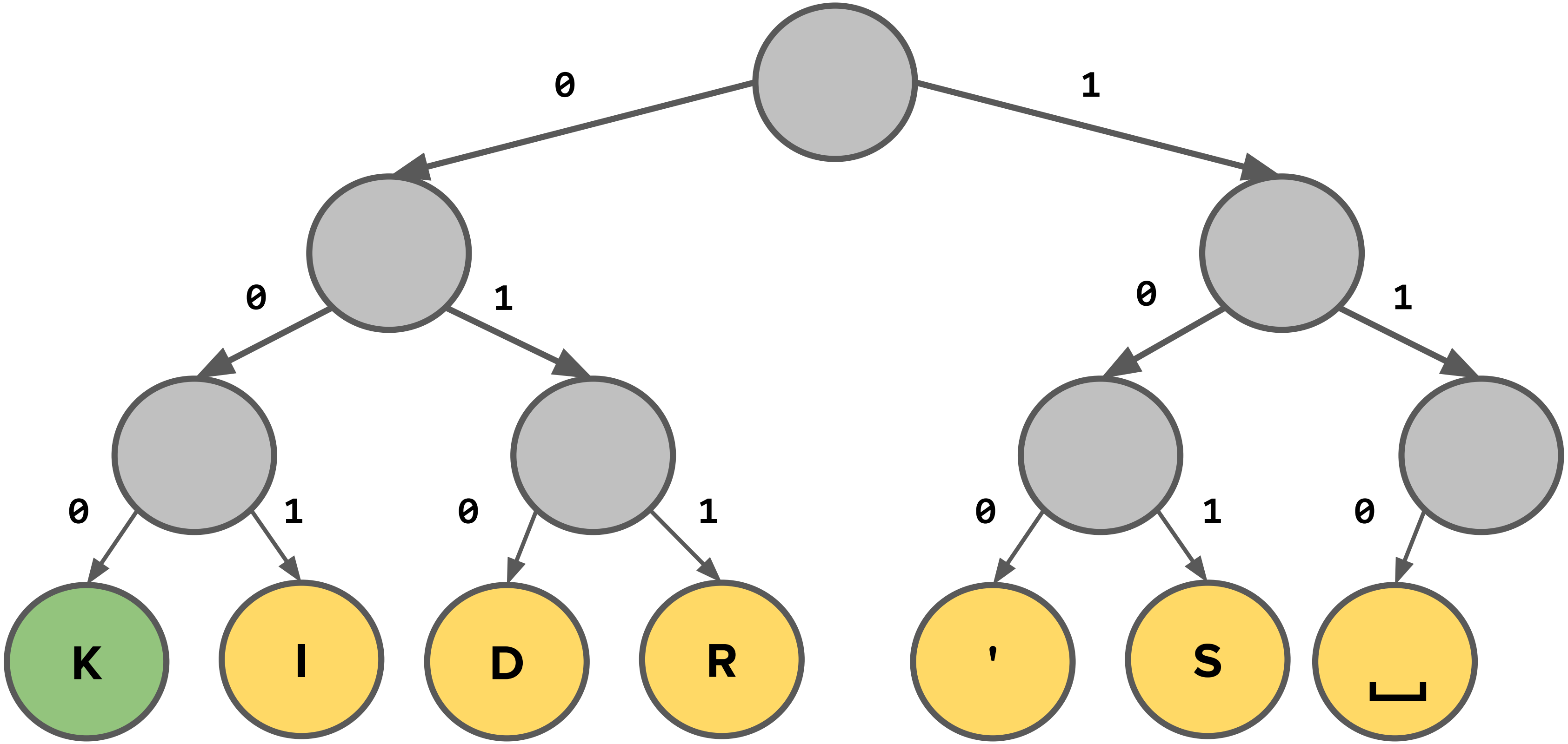
Prefix Coding Mystery: **S** `000001`



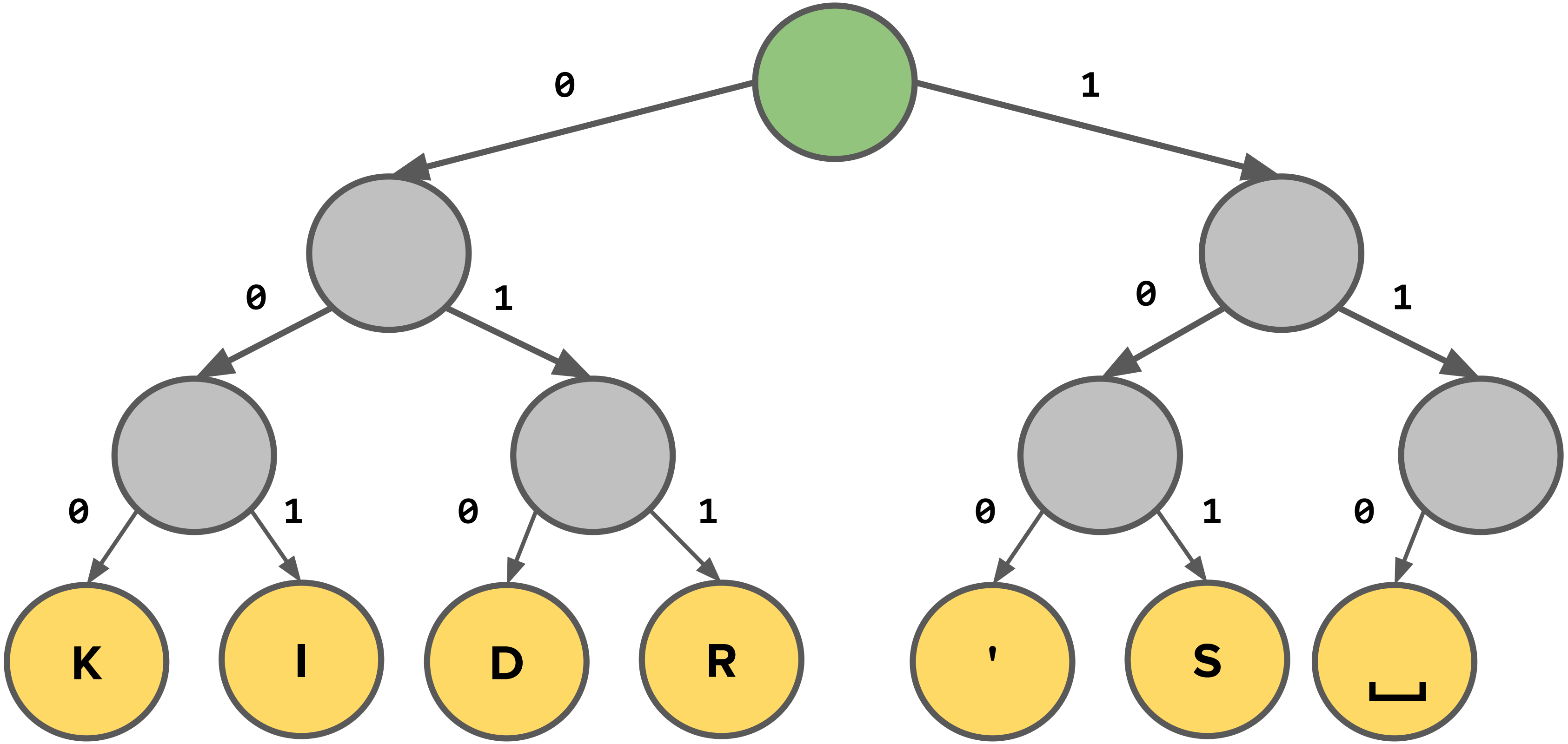
Prefix Coding Mystery: **S** `000001`



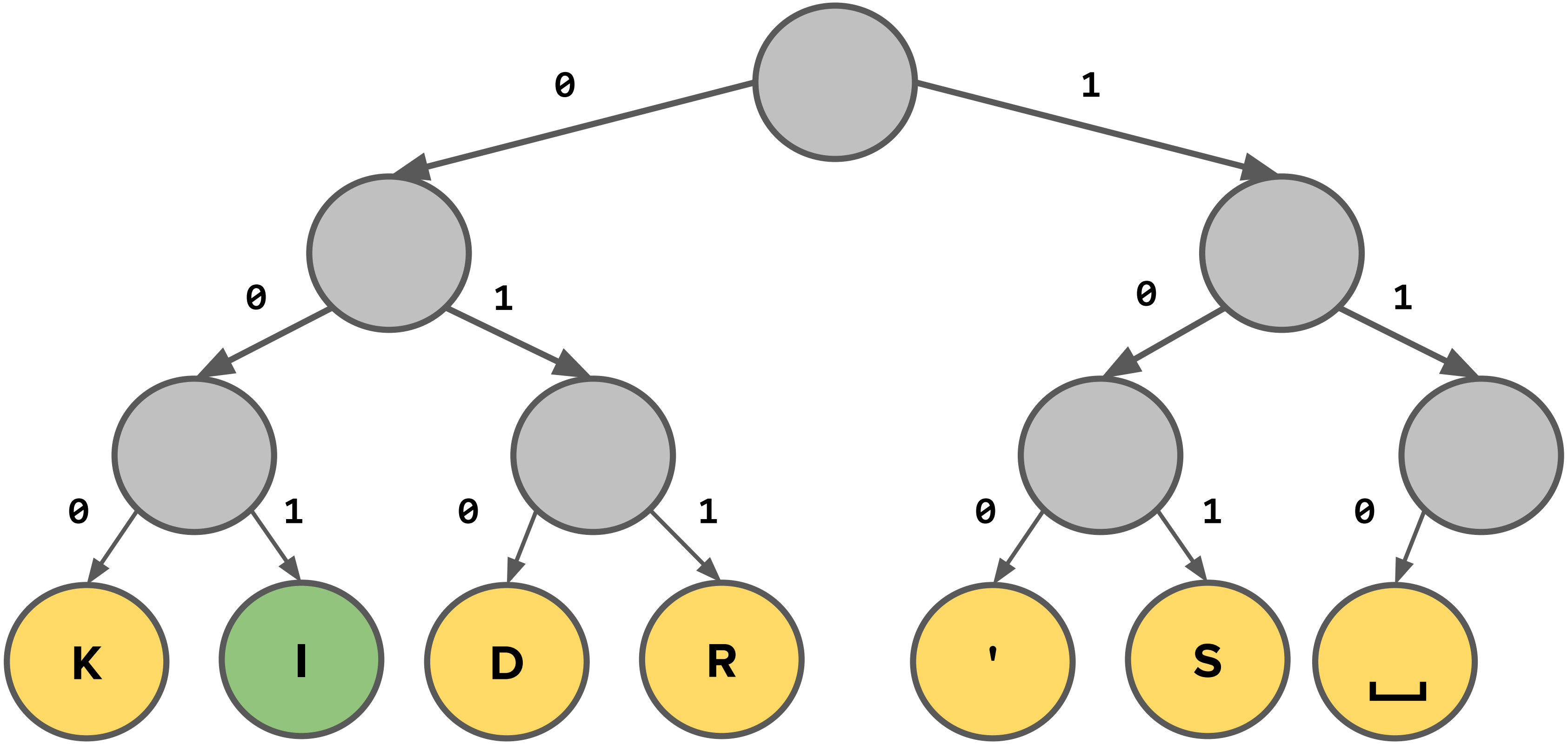
Prefix Coding Mystery: **S K 001**



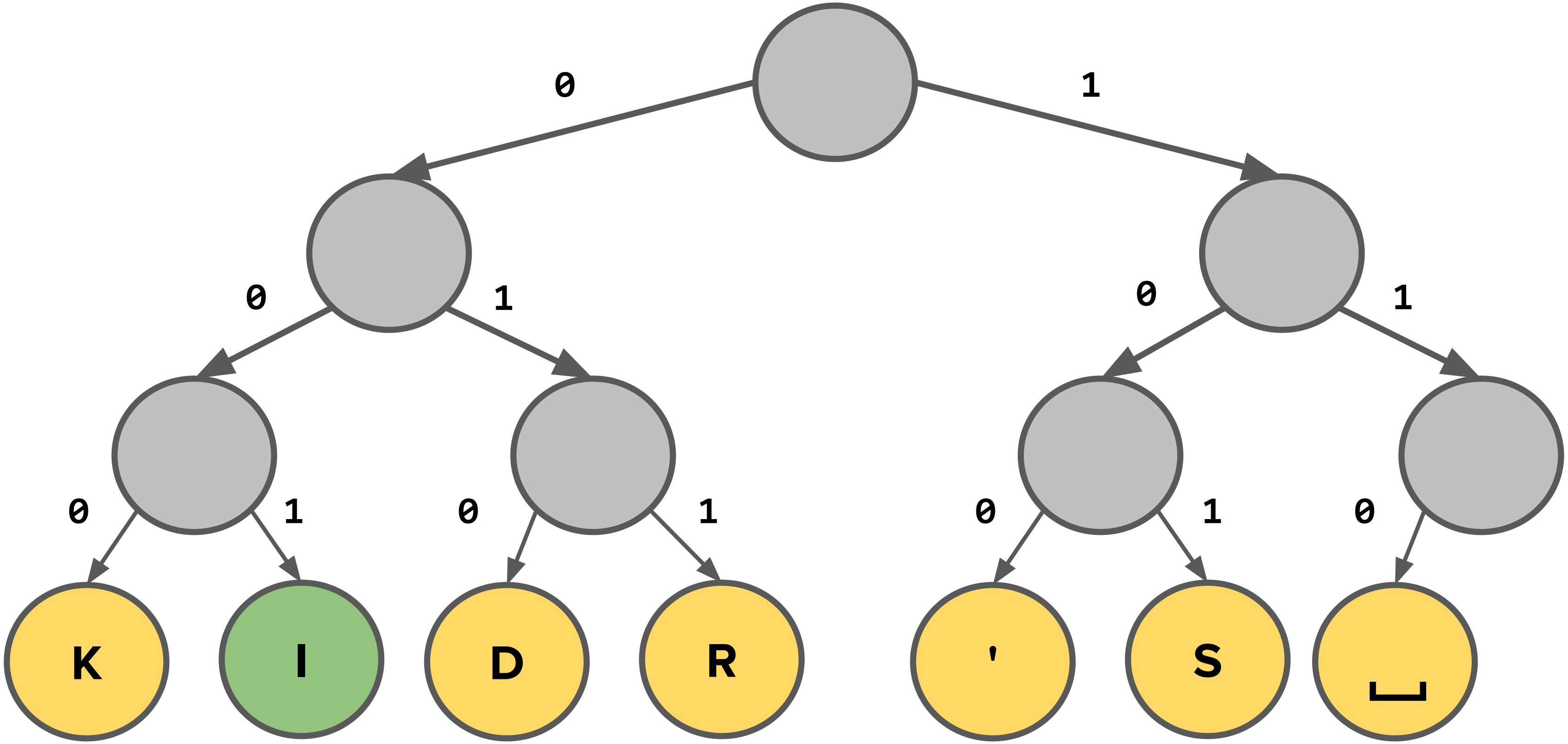
Prefix Coding Mystery: `S K 001`



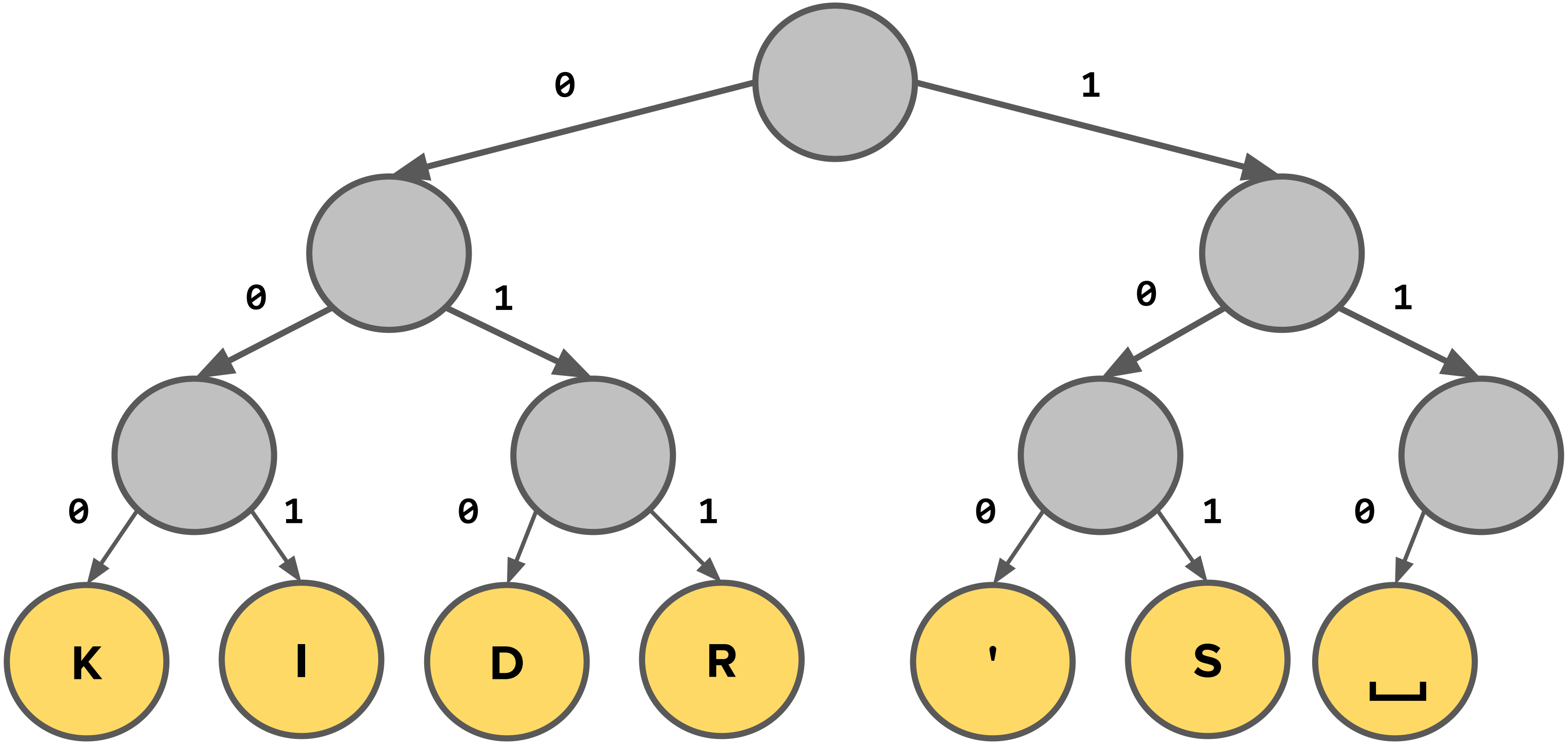
Prefix Coding Mystery: **`S K`** `001`



Prefix Coding Mystery: **S K I**



Prefix Coding Mystery: **SKI**



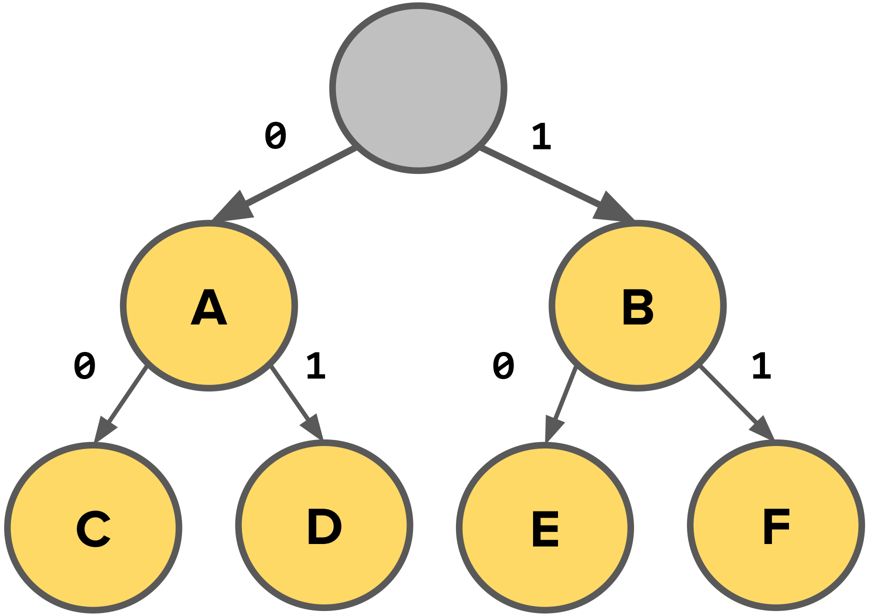
Coding Trees

- Not all binary trees will work as coding trees.

Coding Trees

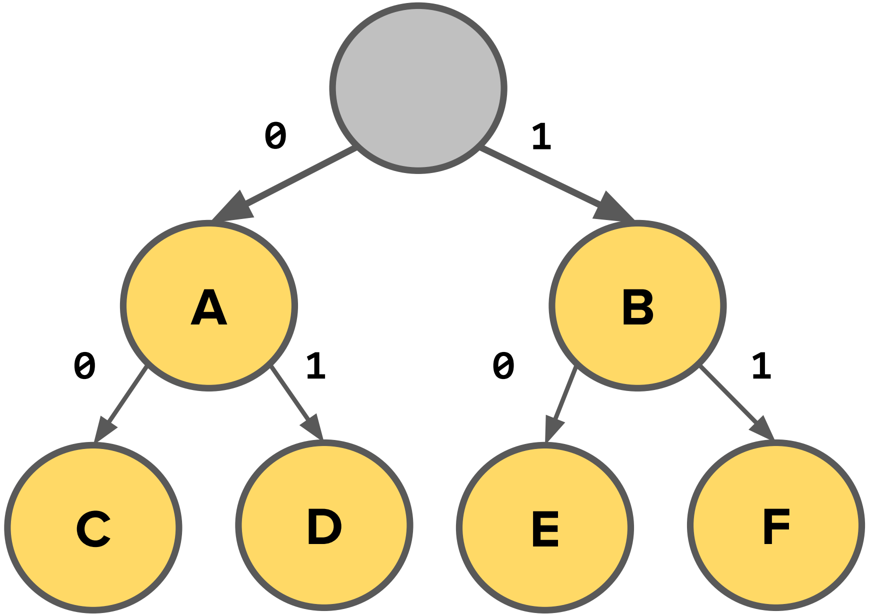
- Not all binary trees will work as coding trees.

- Why is the one to the right not a valid coding tree?



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.

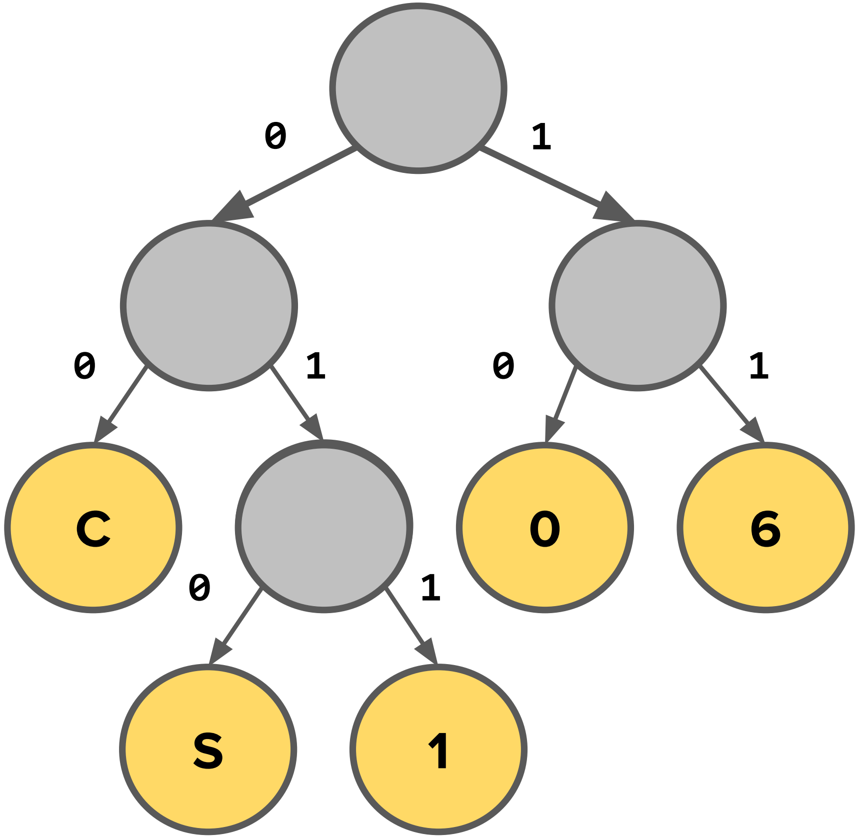


Coding Trees

- 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.

- **Question:** How do we find the best binary tree with this property?



Announcements

Announcements

- Assignment 6 will be released by the end of the day today and will be due on **Wednesday, August 12 at 11:59pm PDT**. This is a hard deadline – there is **no grace period and no submissions will be accepted after this time**.

- Final project reports are due on **Sunday, August 9 at 11:59pm PDT**. You will have the opportunity to schedule your final presentation time after submitting. Reports should be submitted to Paperless and time slot sign-ups will also happen through Paperless.

Huffman Coding

Story Time

Link to full story here:

https://www.maa.org/sites/default/files/images/upload_library/46/Pengelley_projects/Project-14/Huffman.pdf

The Algorithm

Huffman Coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies.

Huffman Coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies.

- Different data (different text, different images, etc.) will each have their own personalized Huffman coding tree.

Huffman Coding

- Huffman coding is an algorithm for generating a coding tree for a given piece of data that produces a **provably minimal encoding** for a given pattern of letter frequencies.

- Different data (different text, different images, etc.) will each have their own personalized Huffman coding tree.

- The Huffman coding algorithm is a flexible, powerful, adaptive algorithm for data compression. And you will implement it on the final assignment as your capstone accomplishment of the quarter!

Huffman Coding Pseudocode

- To generate the optimal encoding tree for a given piece of text:

Huffman Coding Pseudocode

- To generate the optimal encoding tree for a given piece of text:
  - Build a **frequency table** that tallies the number of times each character appears in the text.

Huffman Coding Pseudocode

- To generate the optimal encoding tree for a given piece of text:
  - Build a **frequency table** that tallies the number of times each character appears in the text.
  - Initialize an empty **priority queue** that will hold partial trees (represented as **`TreeNode*`**)

Huffman Coding Pseudocode

- To generate the optimal encoding tree for a given piece of text:
    - Build a **frequency table** that tallies the number of times each character appears in the text.
    - Initialize an empty **priority queue** that will hold partial trees (represented as `TreeNode*`)
    - Create **one leaf node per distinct character in the input string**. Add each new leaf node to the priority queue. The weight of that leaf is the frequency of the character.

Huffman Coding Pseudocode

- To generate the optimal encoding tree for a given piece of text:
    - Build a **frequency table** that tallies the number of times each character appears in the text.
    - Initialize an empty **priority queue** that will hold partial trees (represented as `TreeNode*`)
    - Create **one leaf node per distinct character in the input string**. Add each new leaf node to the priority queue. The weight of that leaf is the frequency of the character.
    - 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.

Huffman in Action

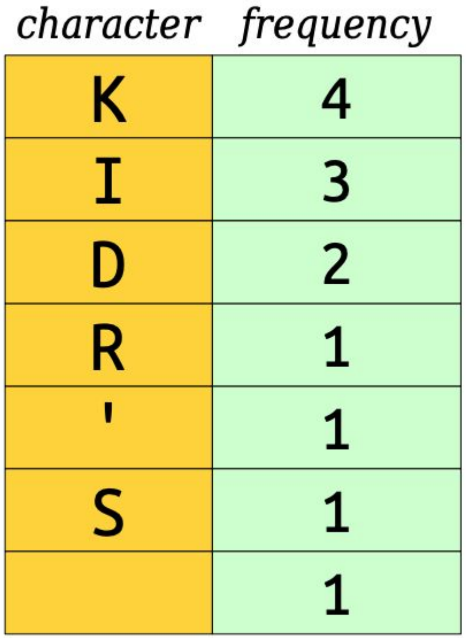
Our goal: Build the optimal encoding tree for **`KIRK'S DIKDIK`**

1) Build the frequency table

Input Text: **`KIRK'S DIKDIK`**

1) Build the frequency table

Input Text: **`KIRK'S DIKDIK`**



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

2) Initialize the priority queue

← higher priority

lower priority →

{ }

3) Add all unique characters as leaf nodes to queue

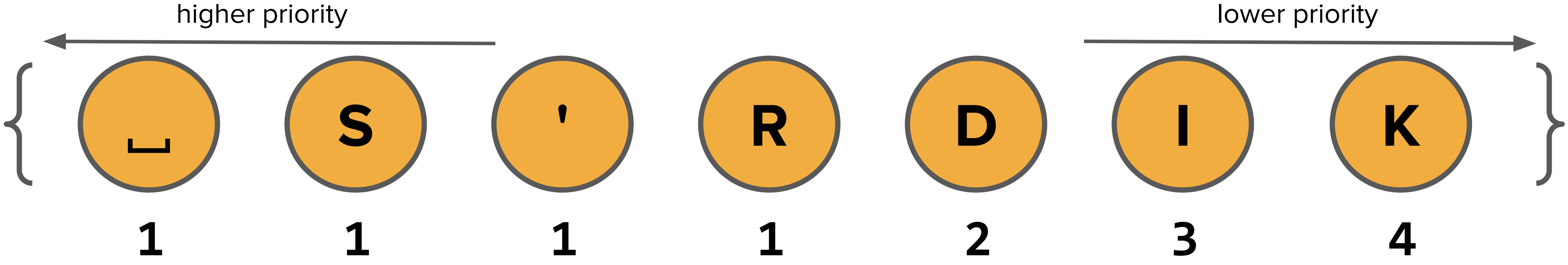
← higher priority

lower priority →

{ }

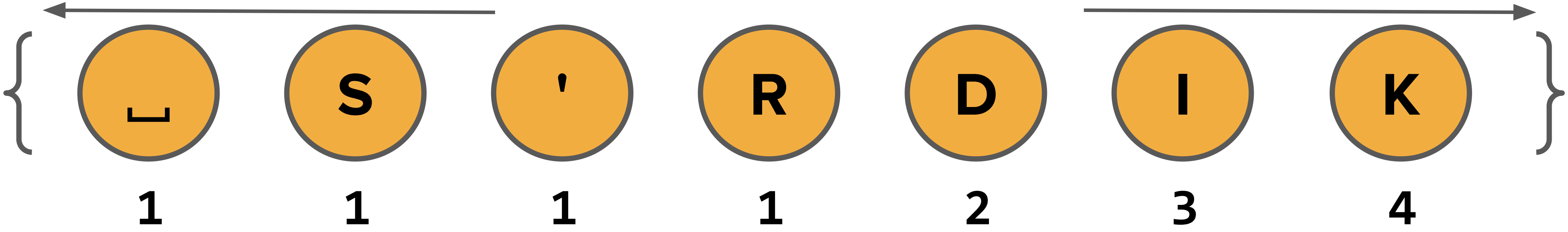
character	frequency
K	4
I	3
D	2
R	1
'	1
S	1
	1

3) Add all unique characters as leaf nodes to queue



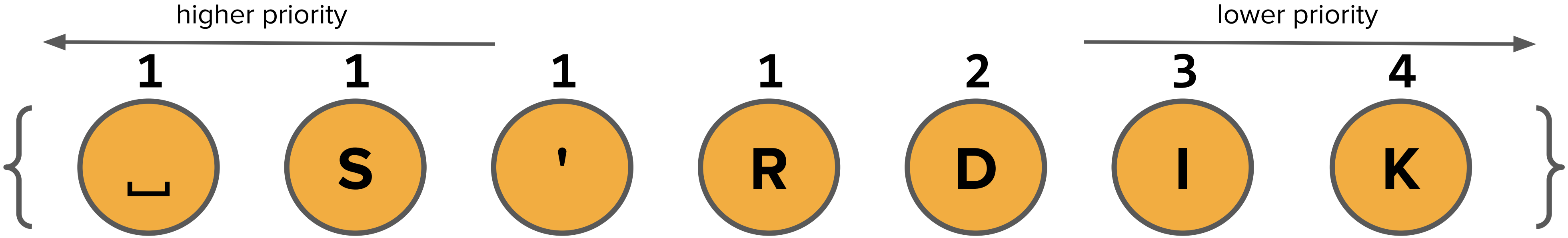
character	*frequency*
K	4
I	3
D	2
R	1
'	1
S	1
	1

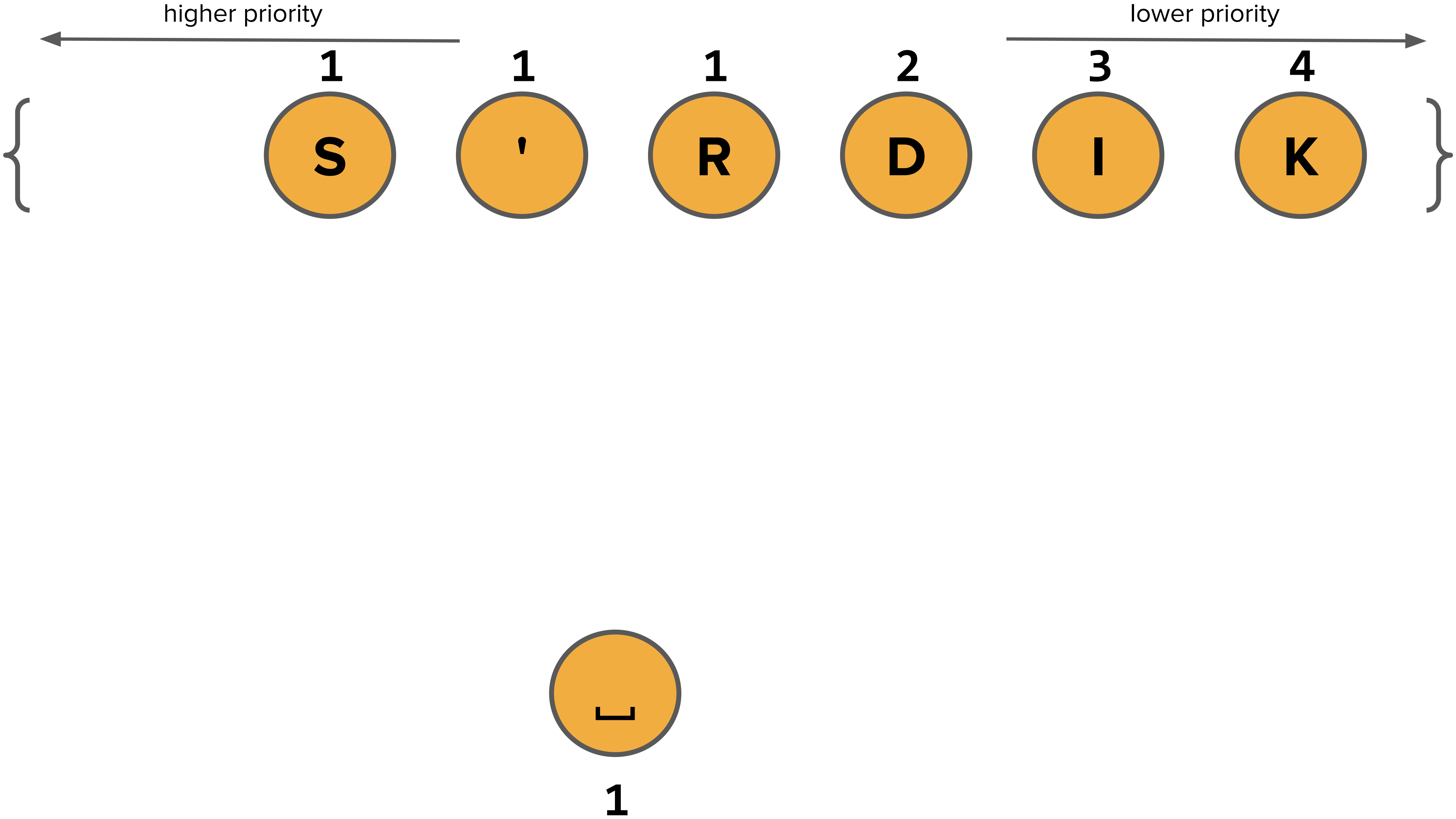
4) Build the Huffman tree by joining adjacent nodes

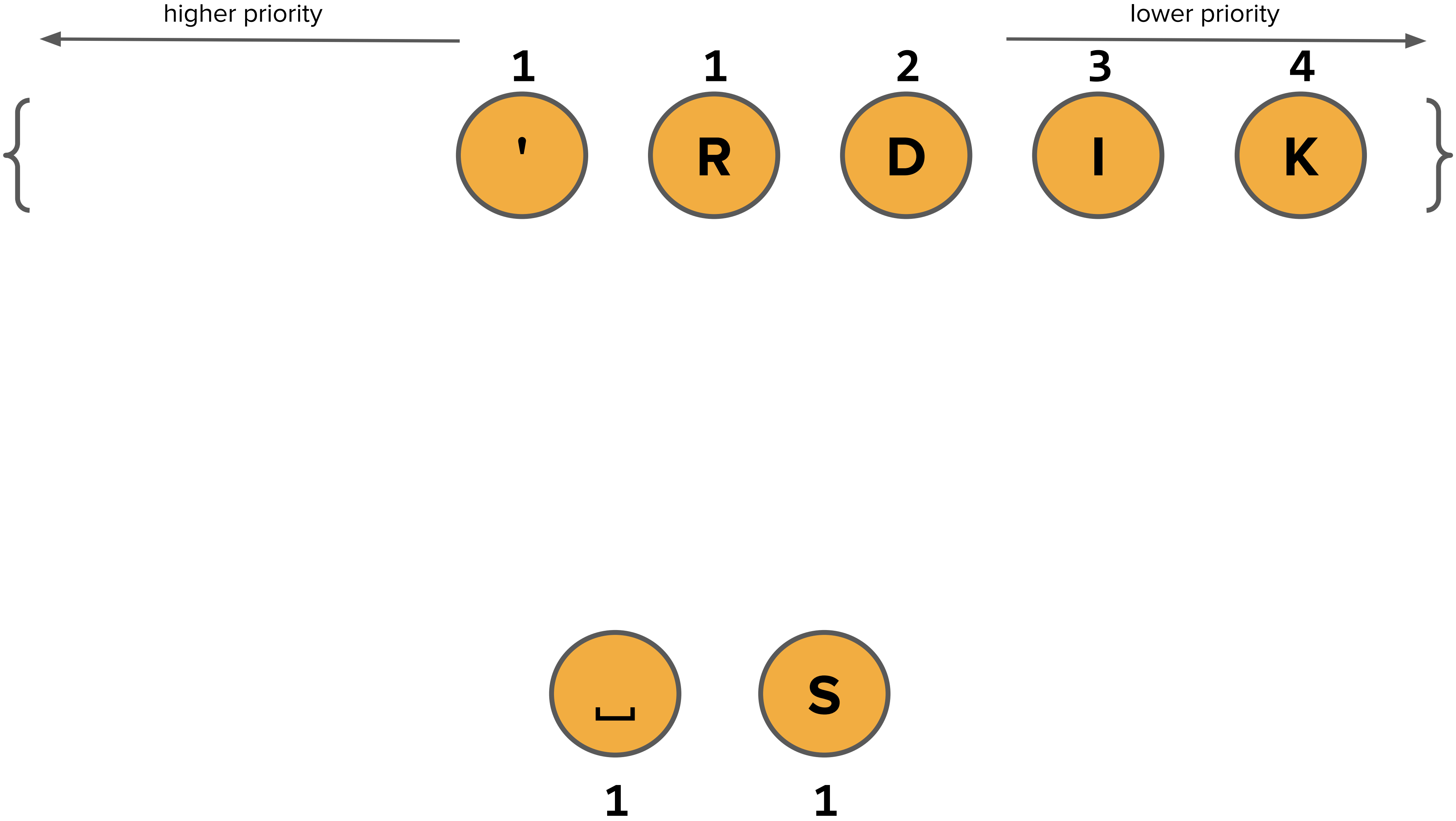


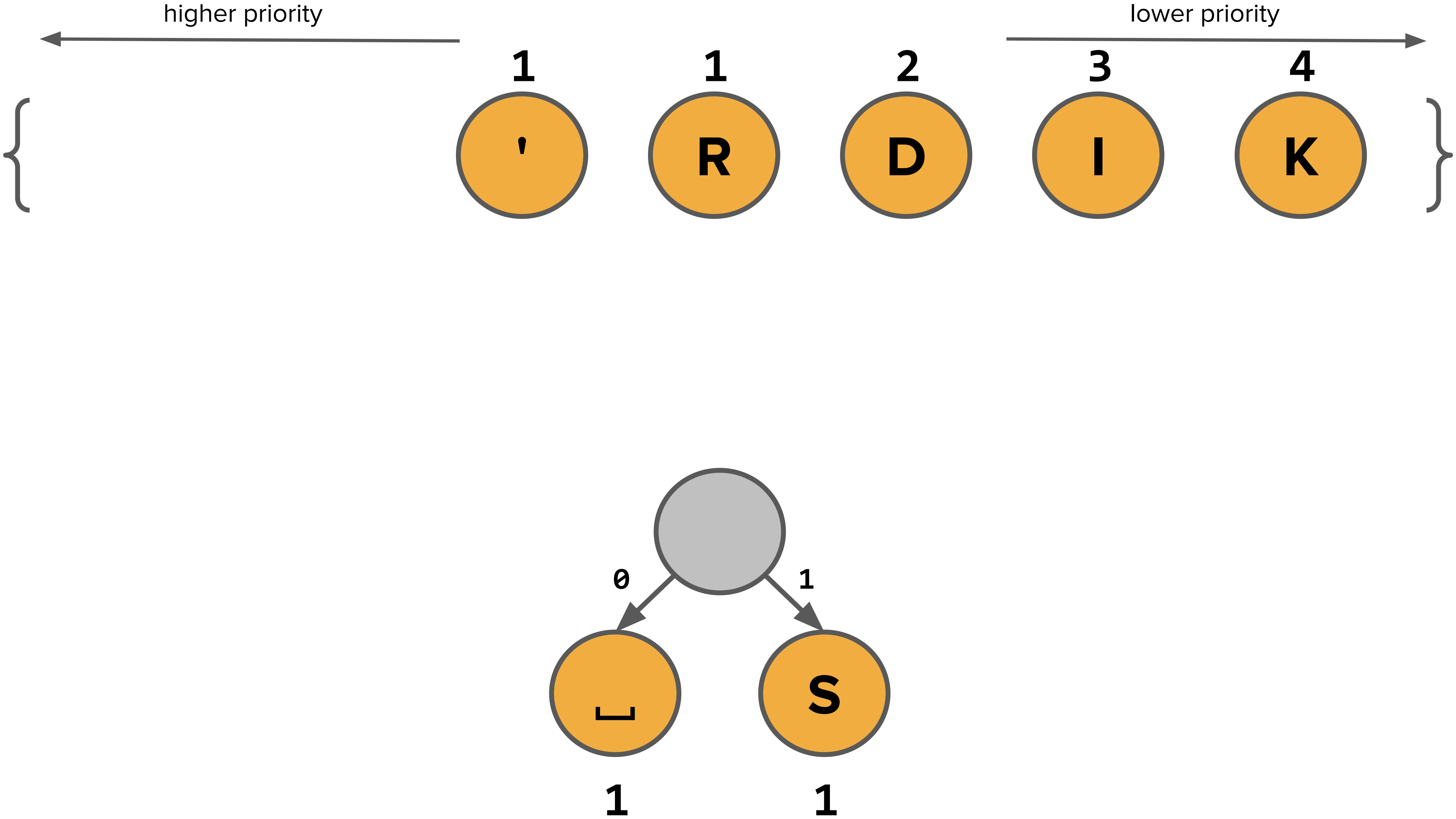
higher priority

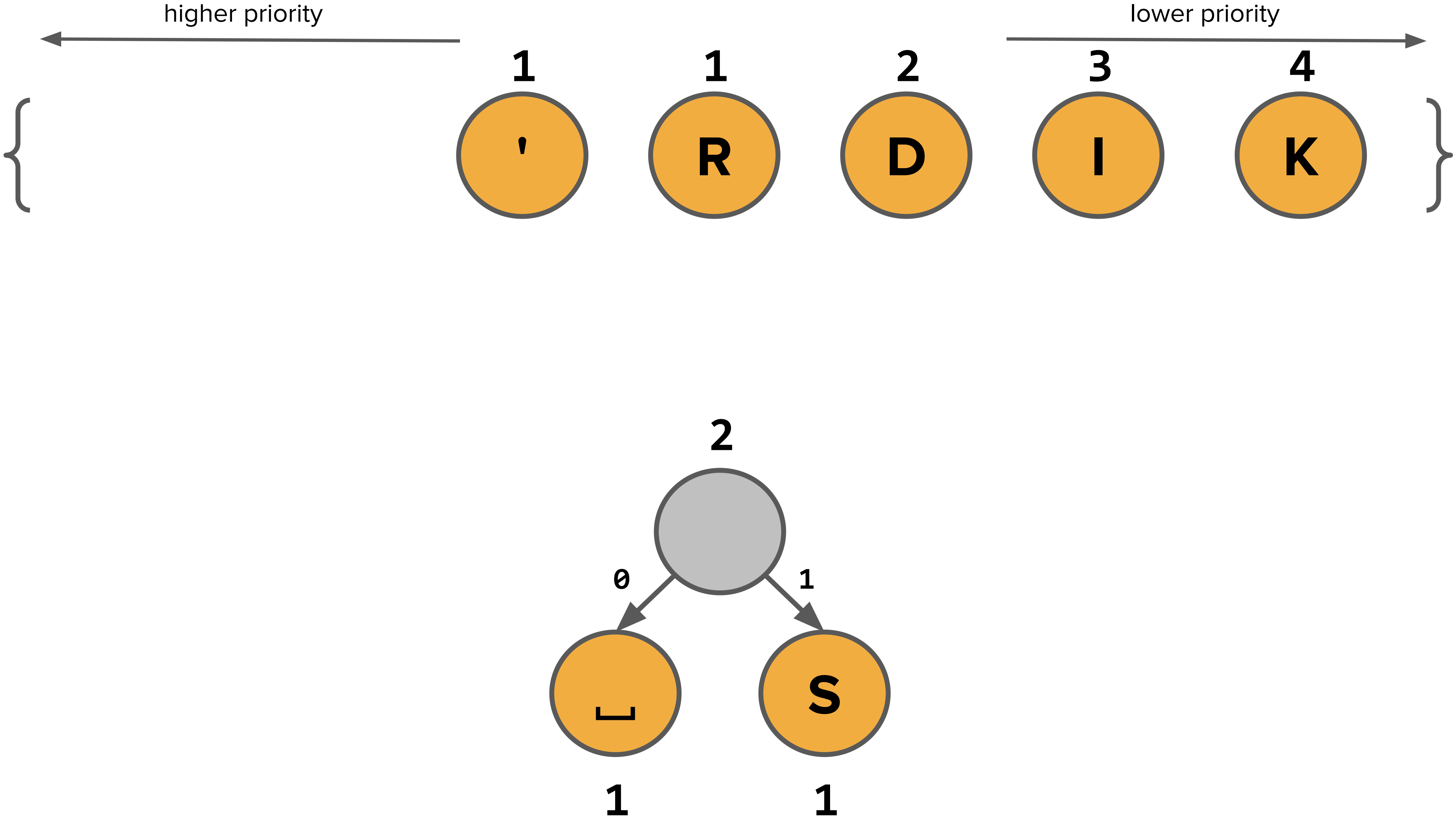
lower priority

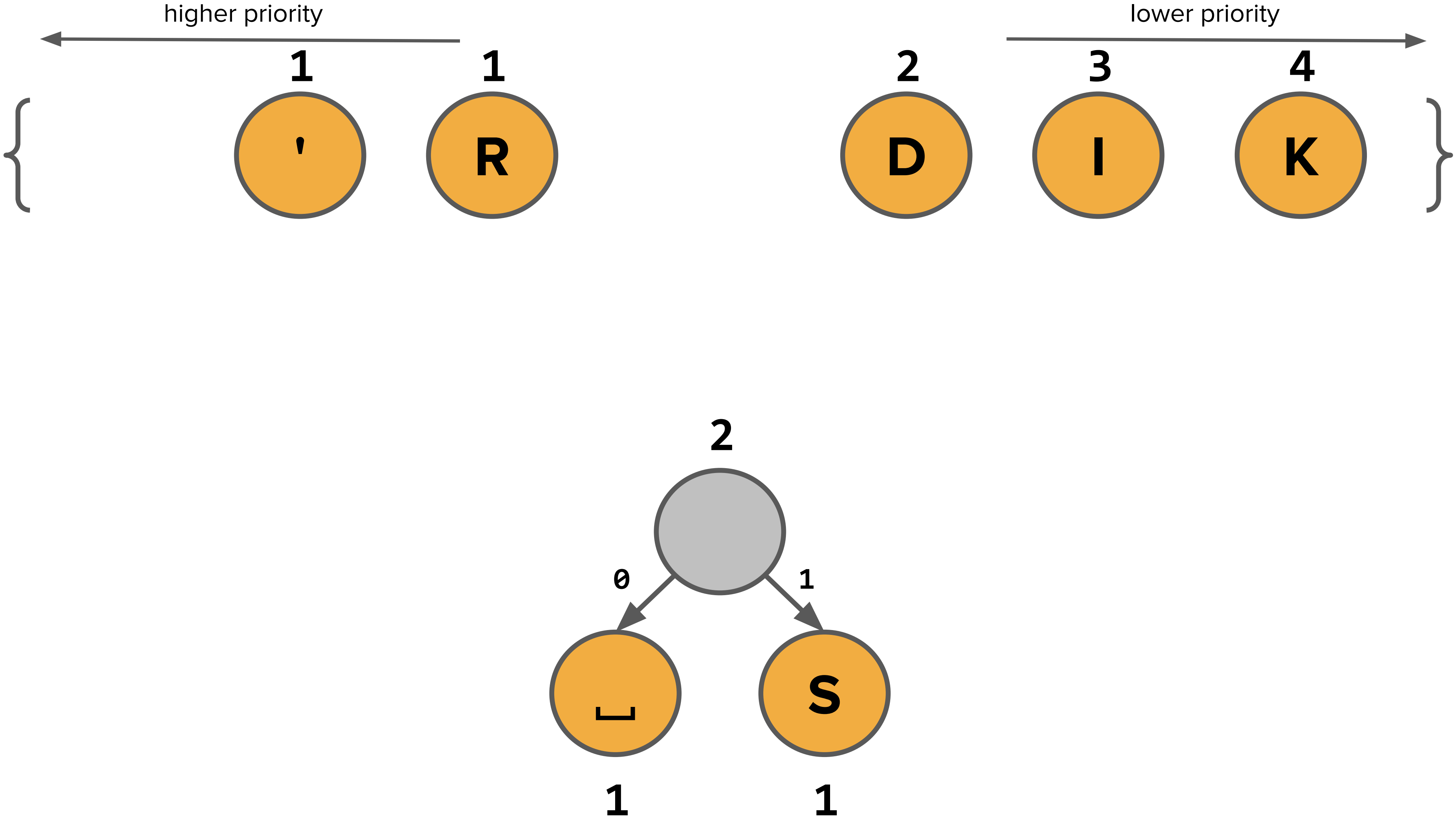

higher priority
lower priority
1
␣
1
S
1
'
1
R
2
D
3
I
4
K

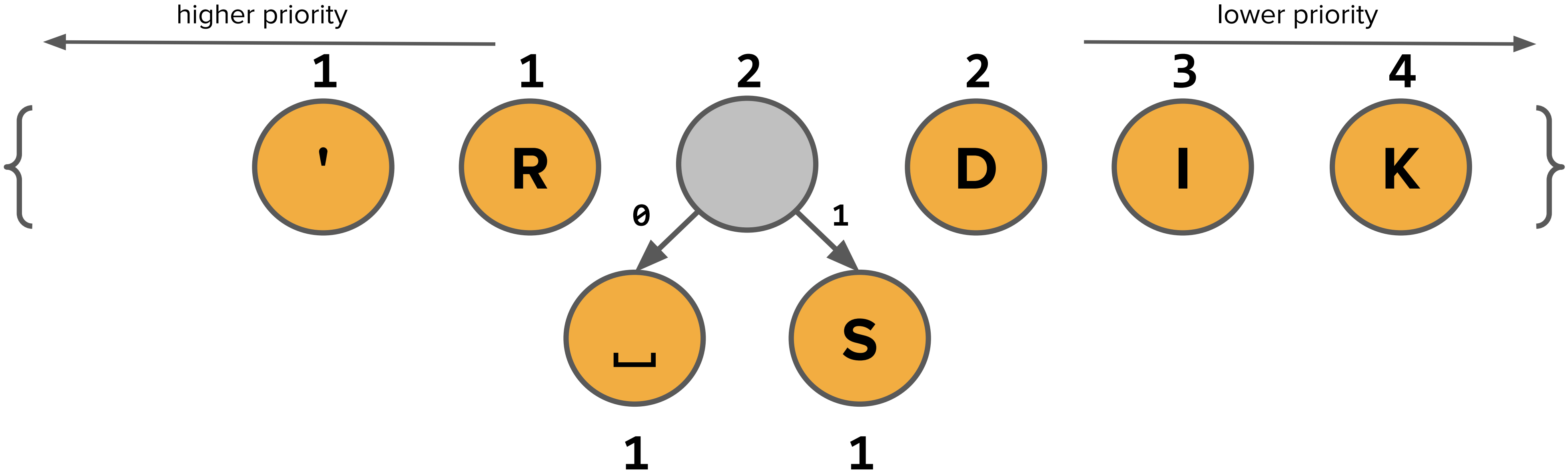

higher priority
lower priority
1
1
1
2
3
4
S
'
R
D
I
K
␣
1

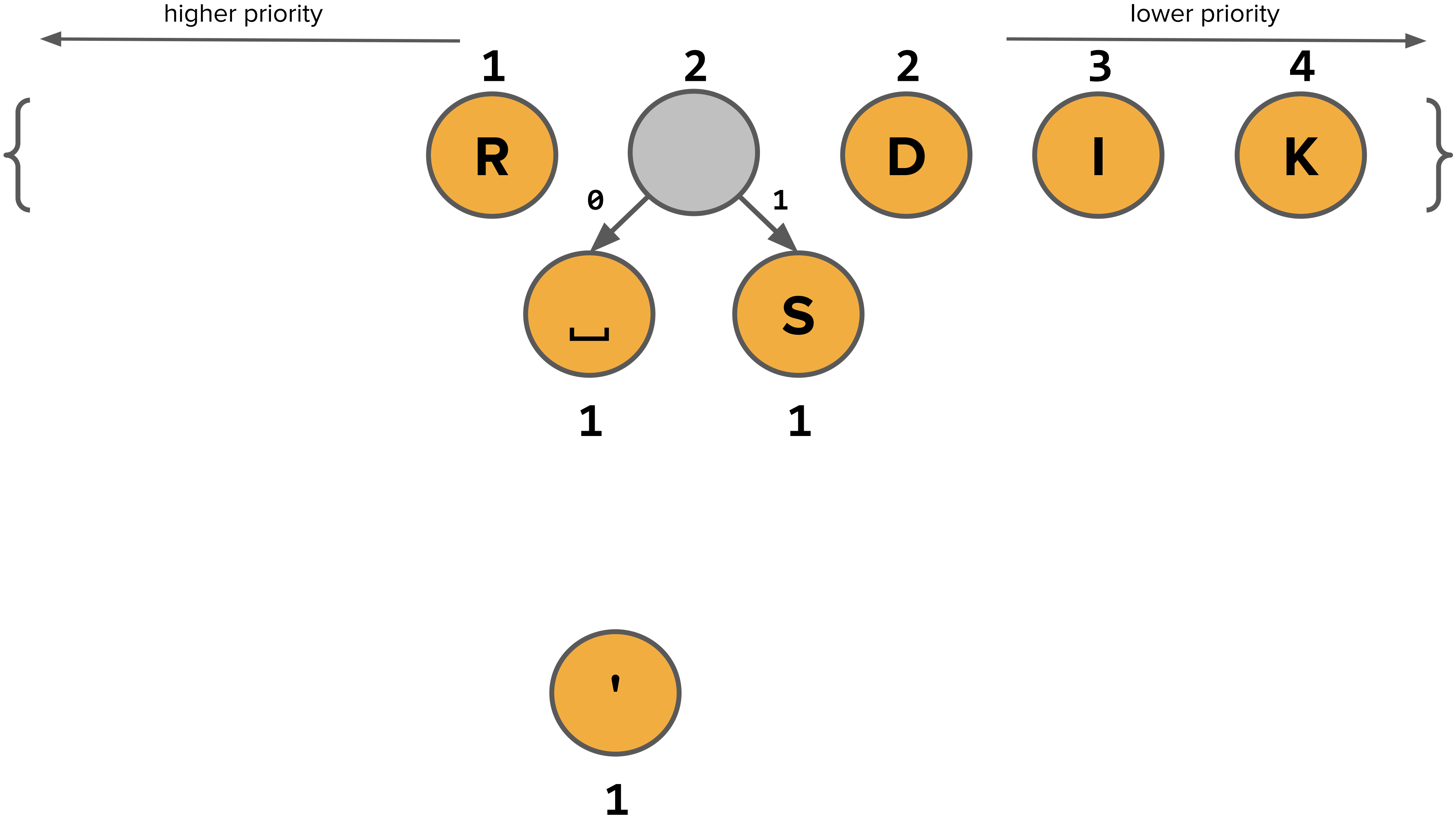

higher priority
lower priority
1
'
1
R
2
D
3
I
4
K
␣
1
S
1

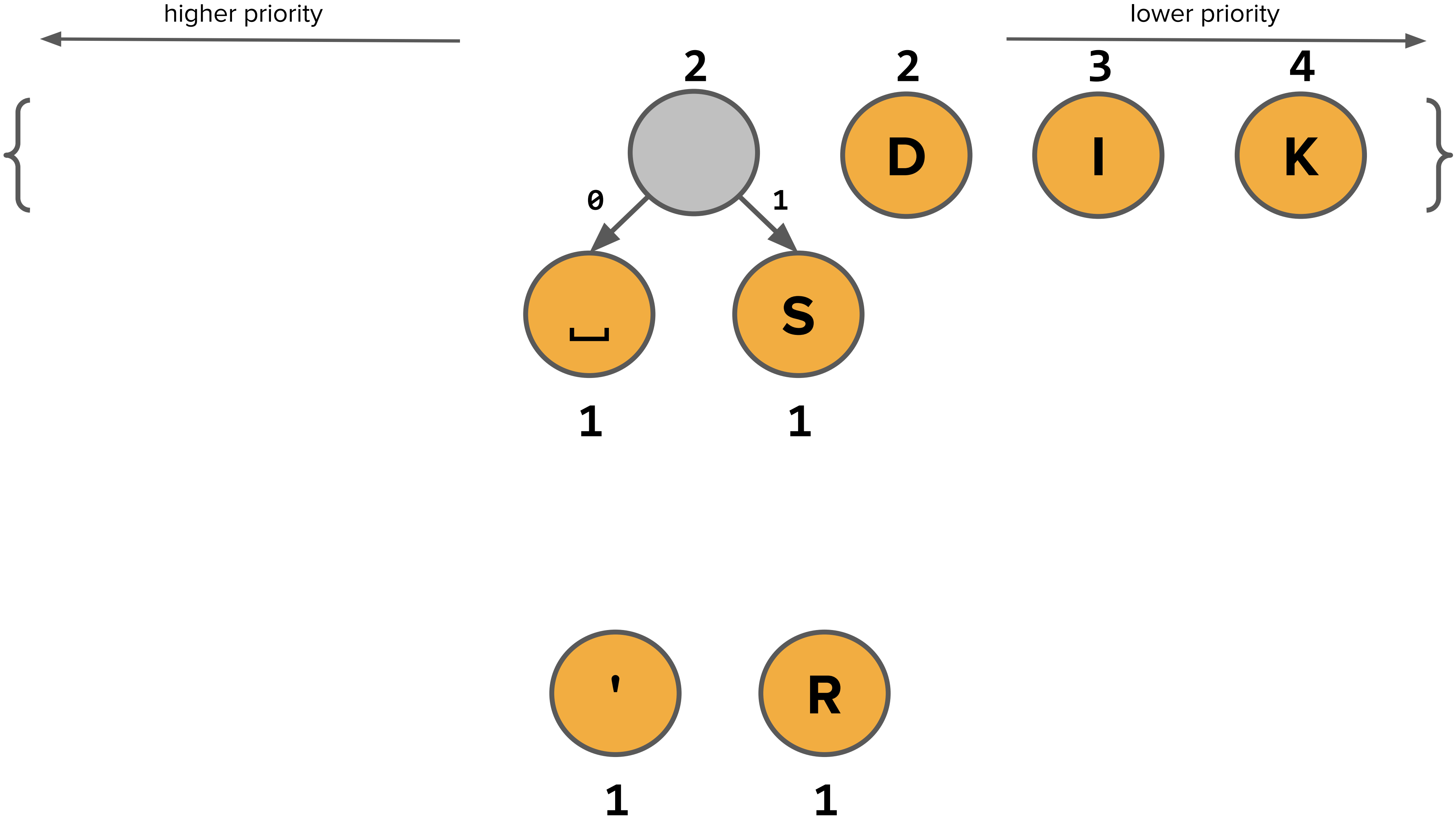

higher priority
lower priority
1
'
1
R
2
D
3
I
4
K
0
1
␣
S
1
1

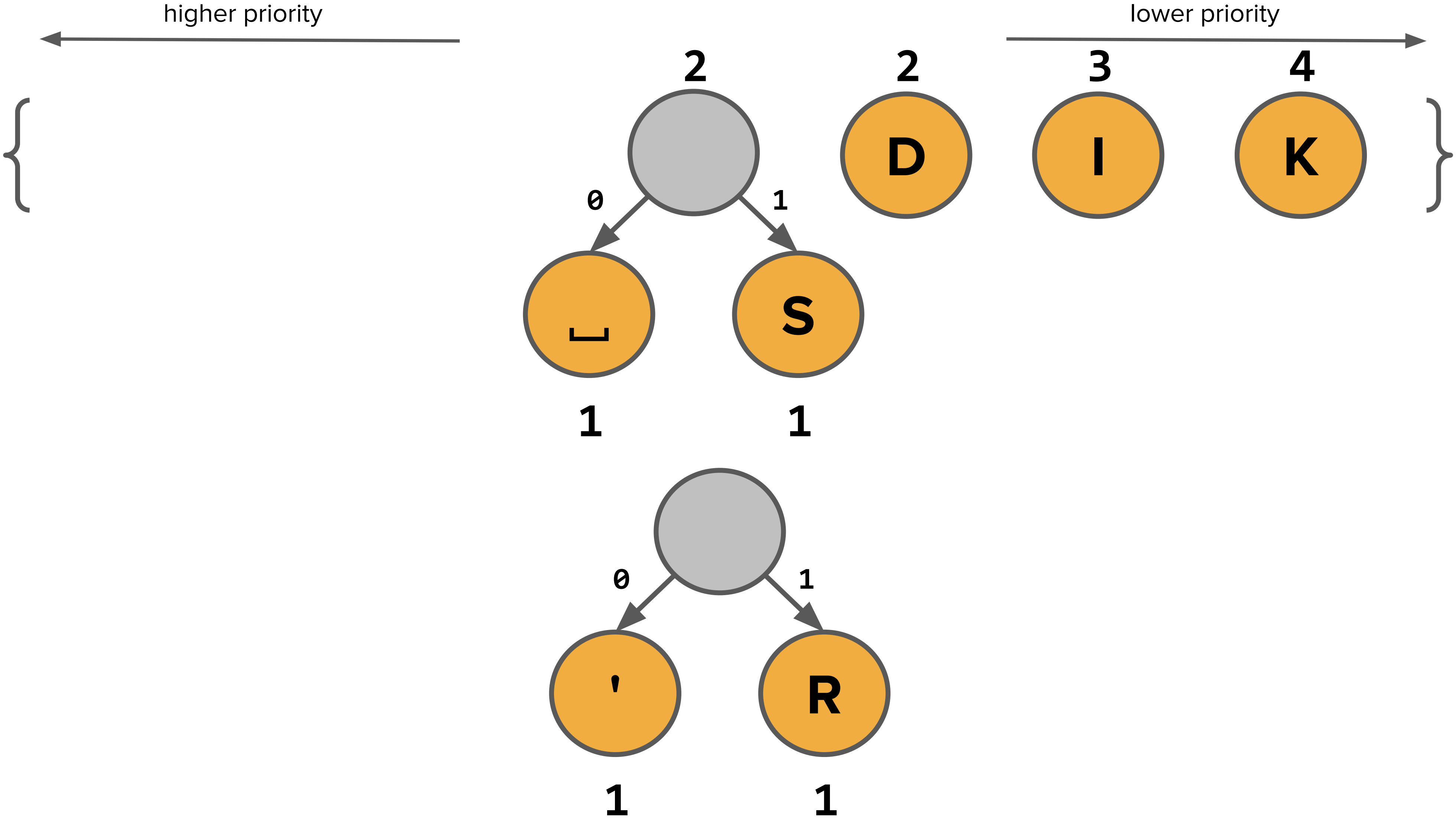

higher priority
lower priority
1
'
1
R
2
D
3
I
4
K
2
0
1
␣
S
1
1

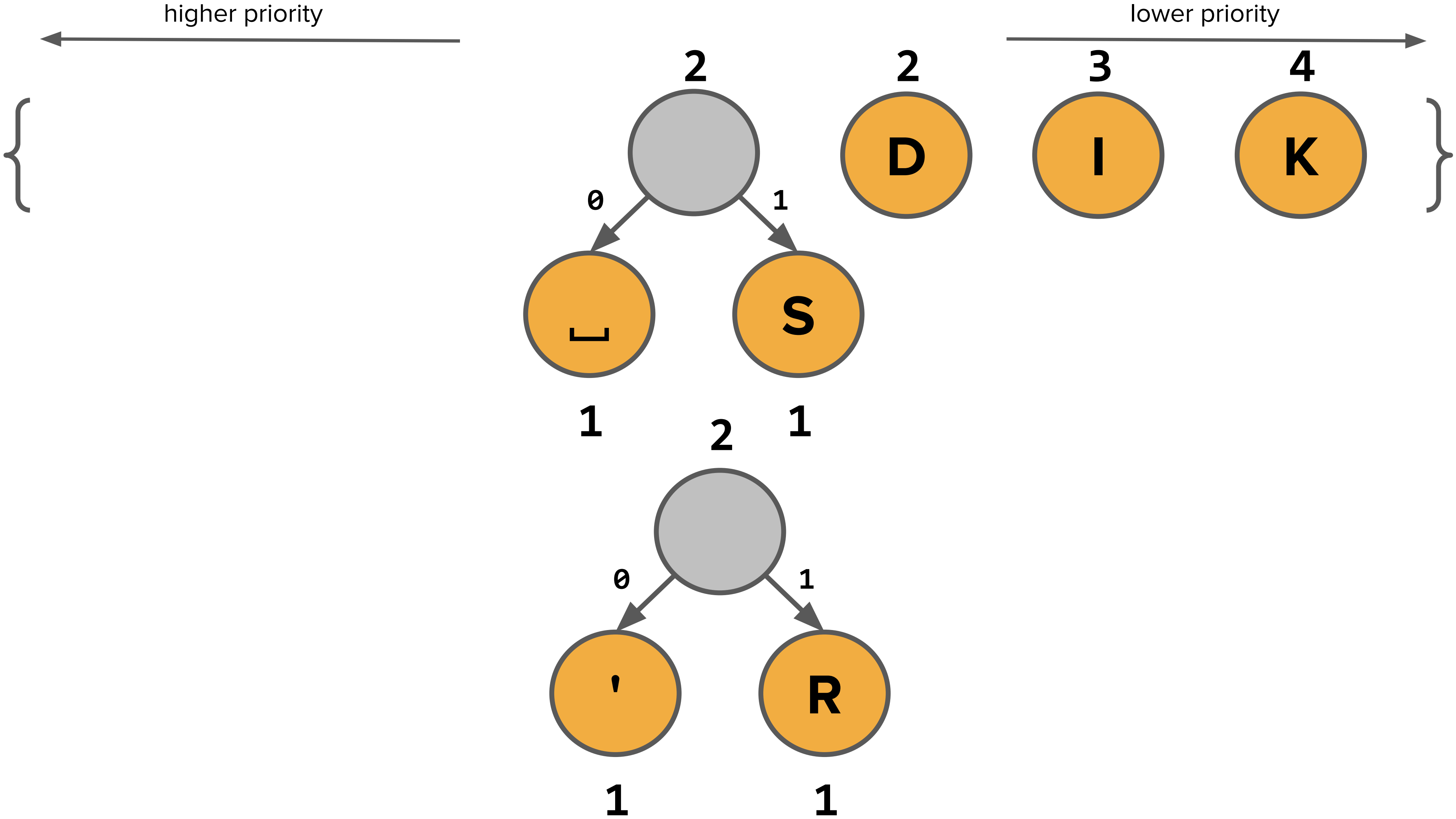

higher priority
lower priority
1
'
1
R
2
D
3
I
4
K
2
0
1
␣
1
S
1

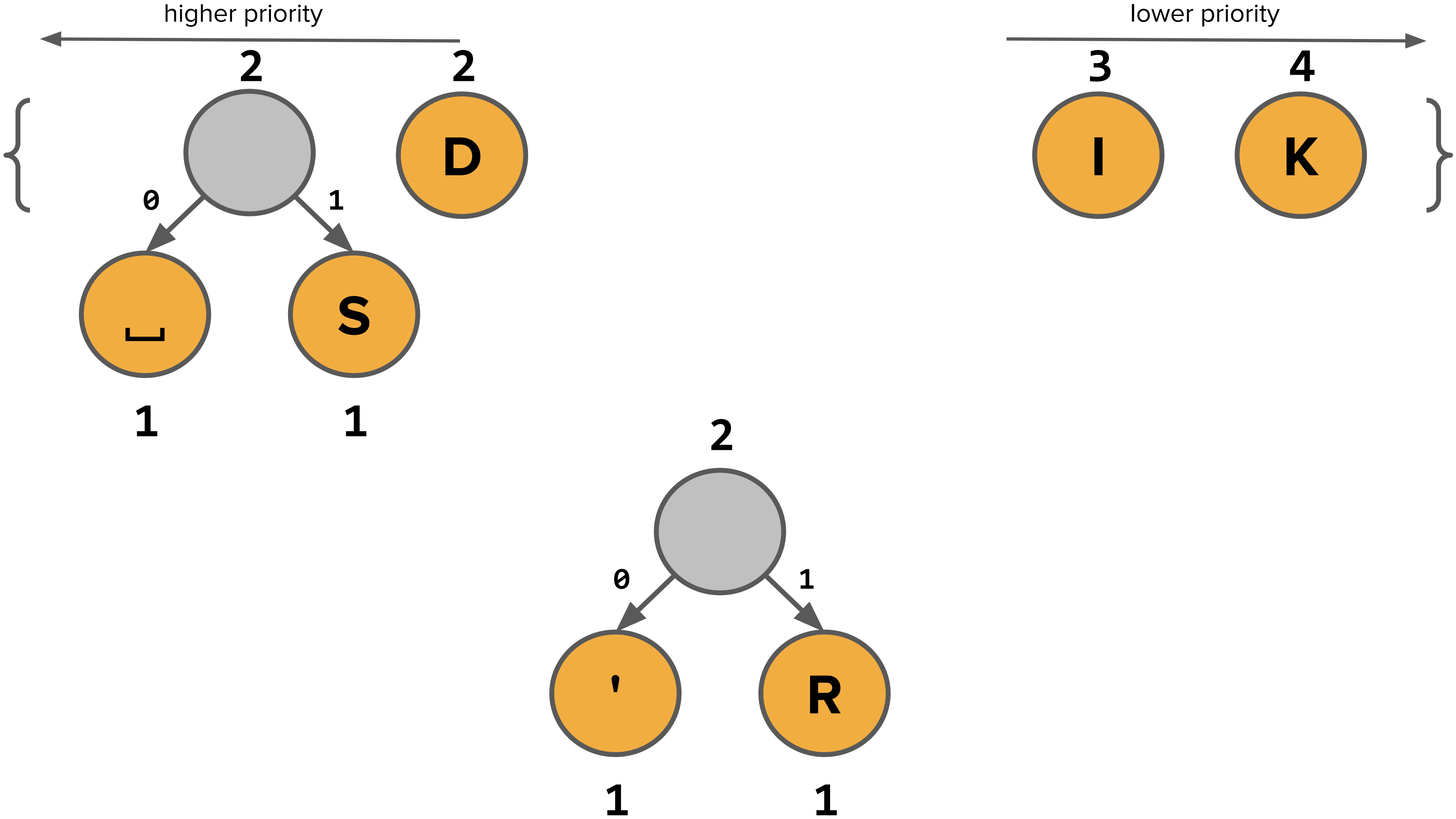

higher priority
lower priority
1
'
1
R
2
0
1
␣
1
S
1
2
D
3
I
4
K

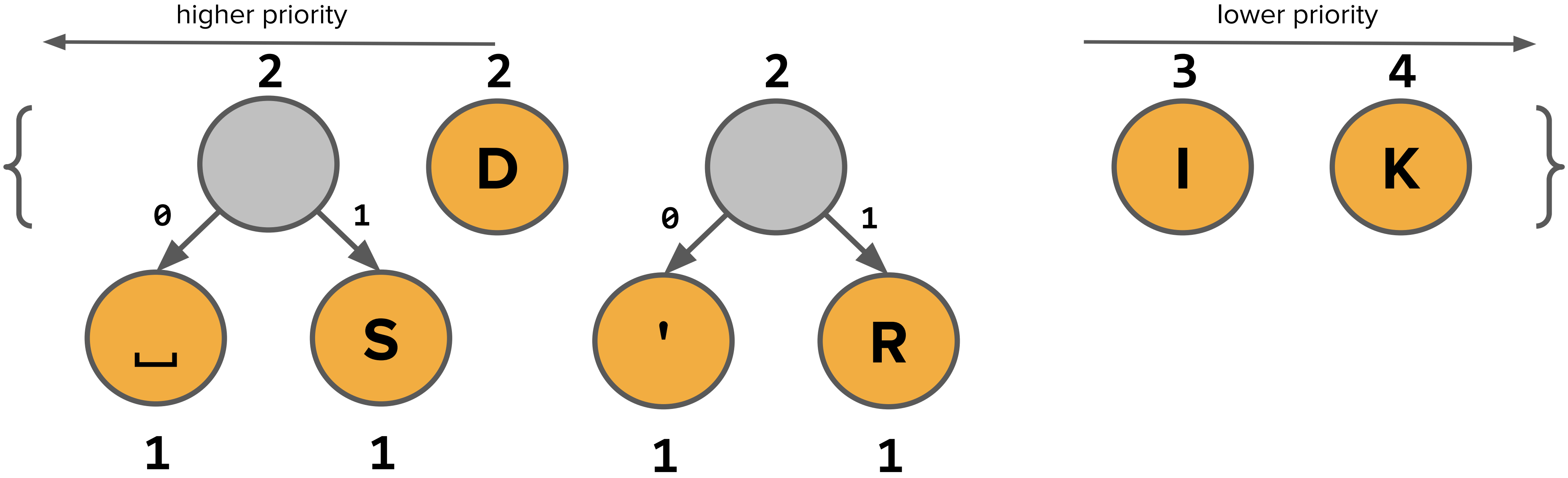

higher priority
lower priority
1
2
2
3
4
R
D
I
K
0
1
␣
S
1
1
'
1

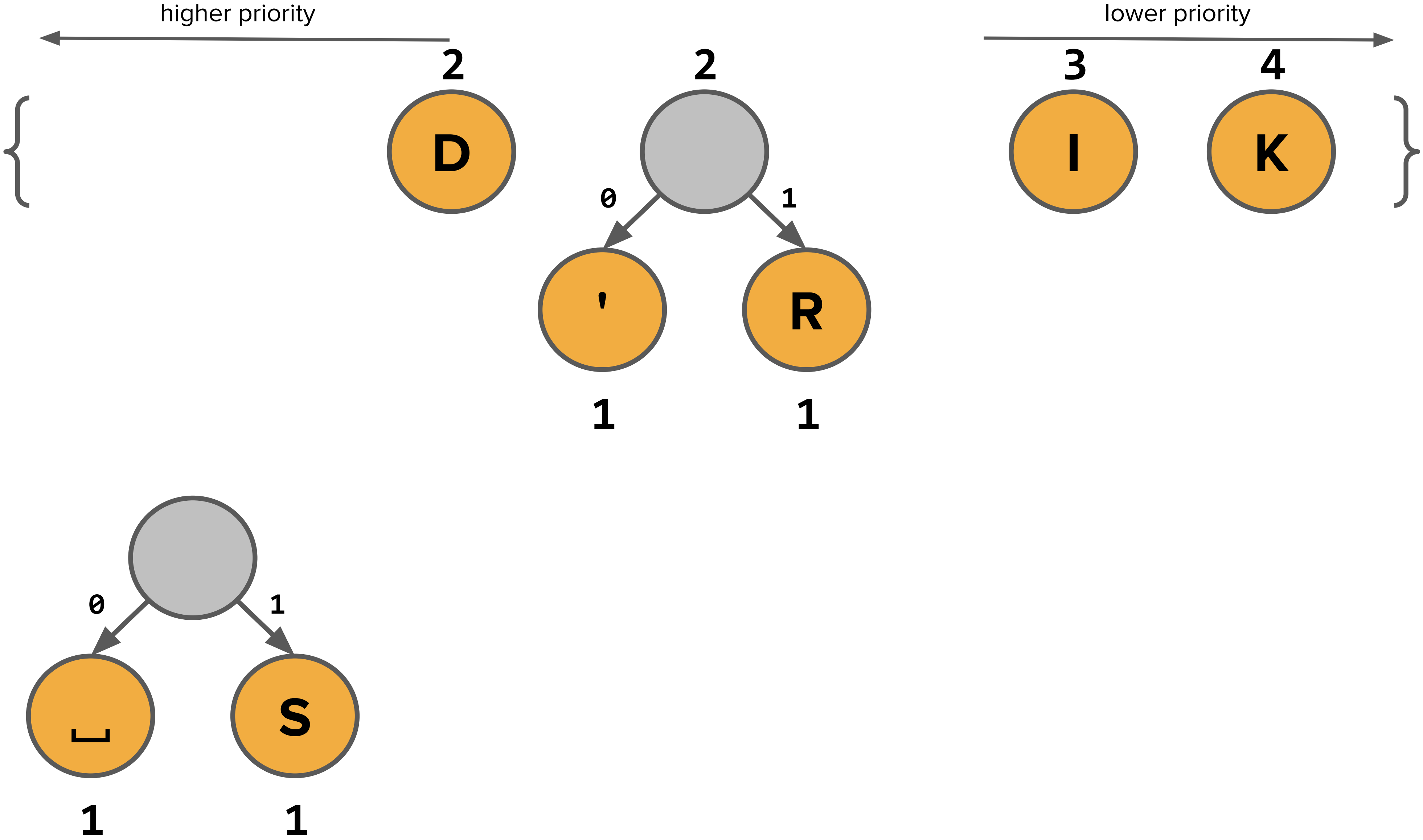

higher priority
lower priority
2
2
D
3
I
4
K
0
1
␣
1
S
1
'
1
R
1

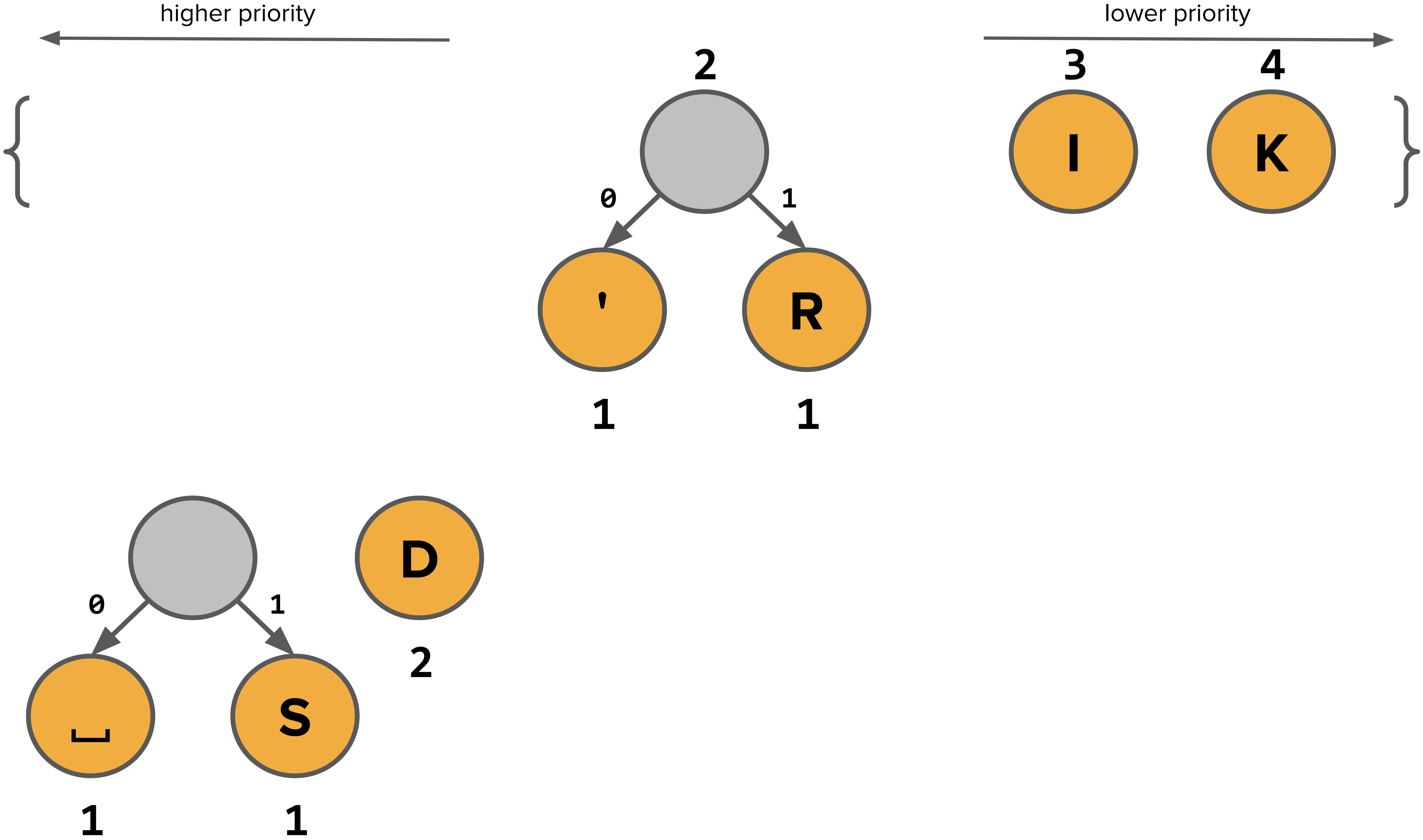

higher priority
lower priority
2
2
3
4
D
I
K
0
1
␣
S
1
1
0
1
'
R
1
1

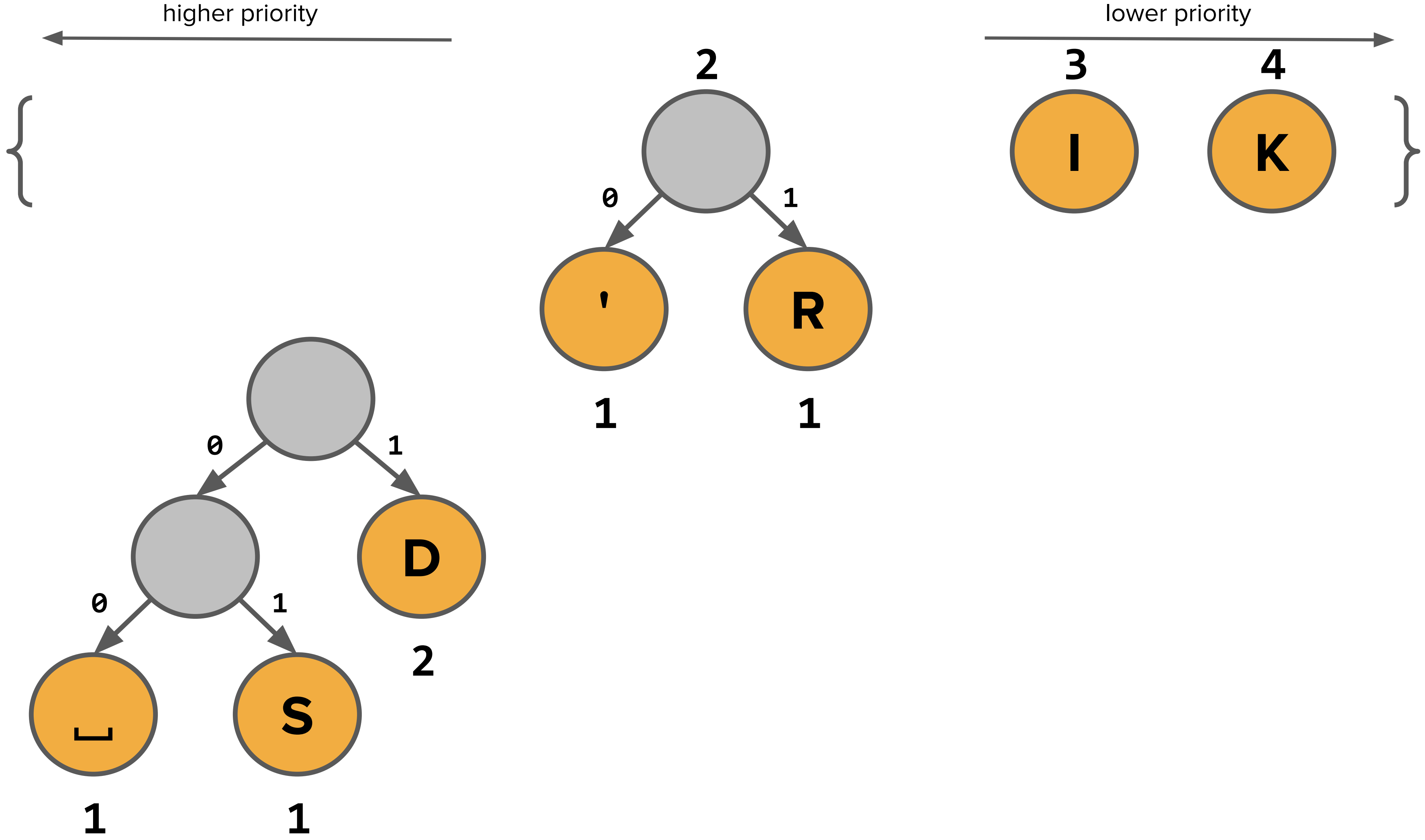

higher priority
lower priority
2
2
3
4
D
I
K
0
1
␣
S
1
1
2
0
1
'
R
1
1

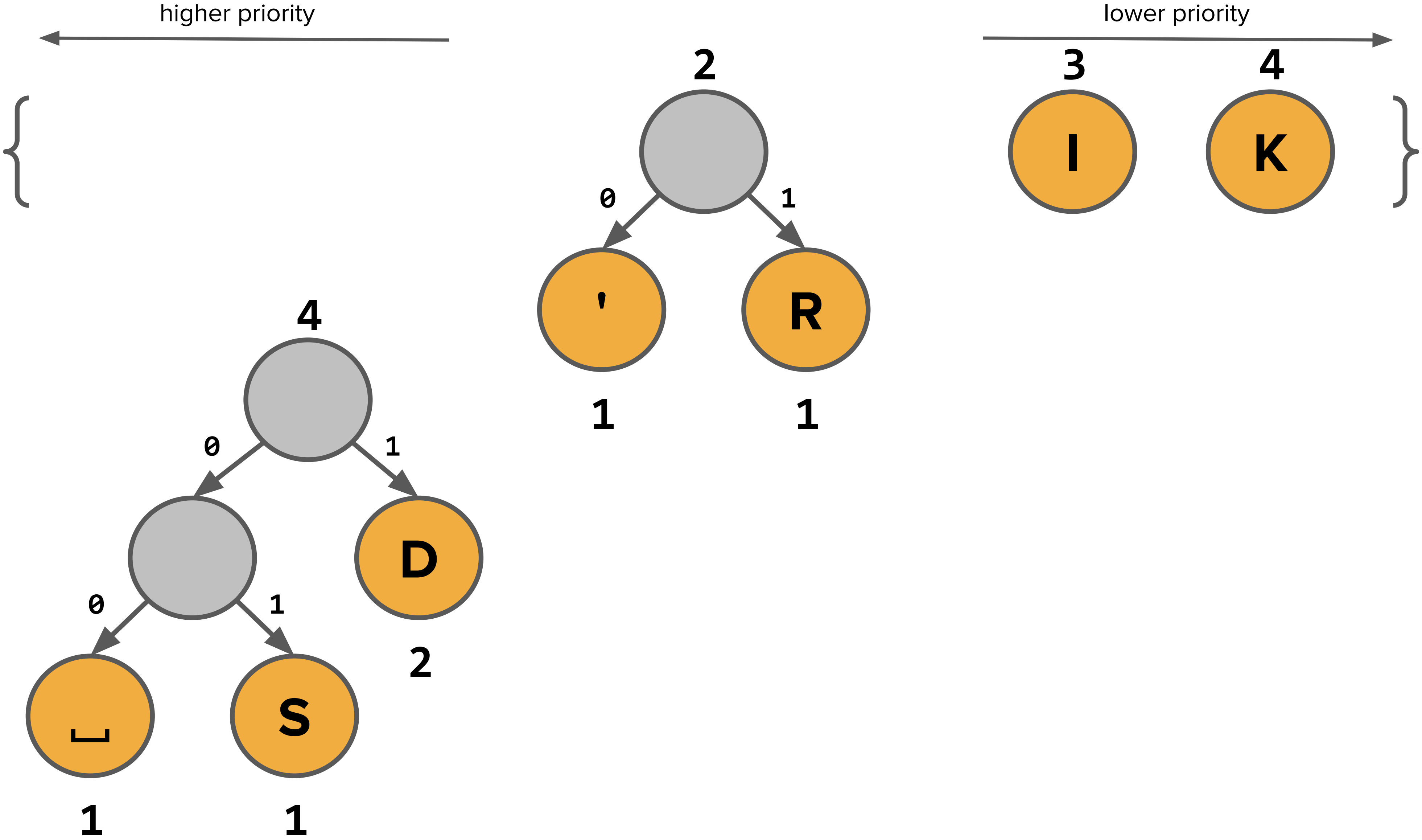

higher priority
lower priority
2
2
3
4
D
I
K
0
1
␣
S
1
1
2
0
1
'
R
1
1

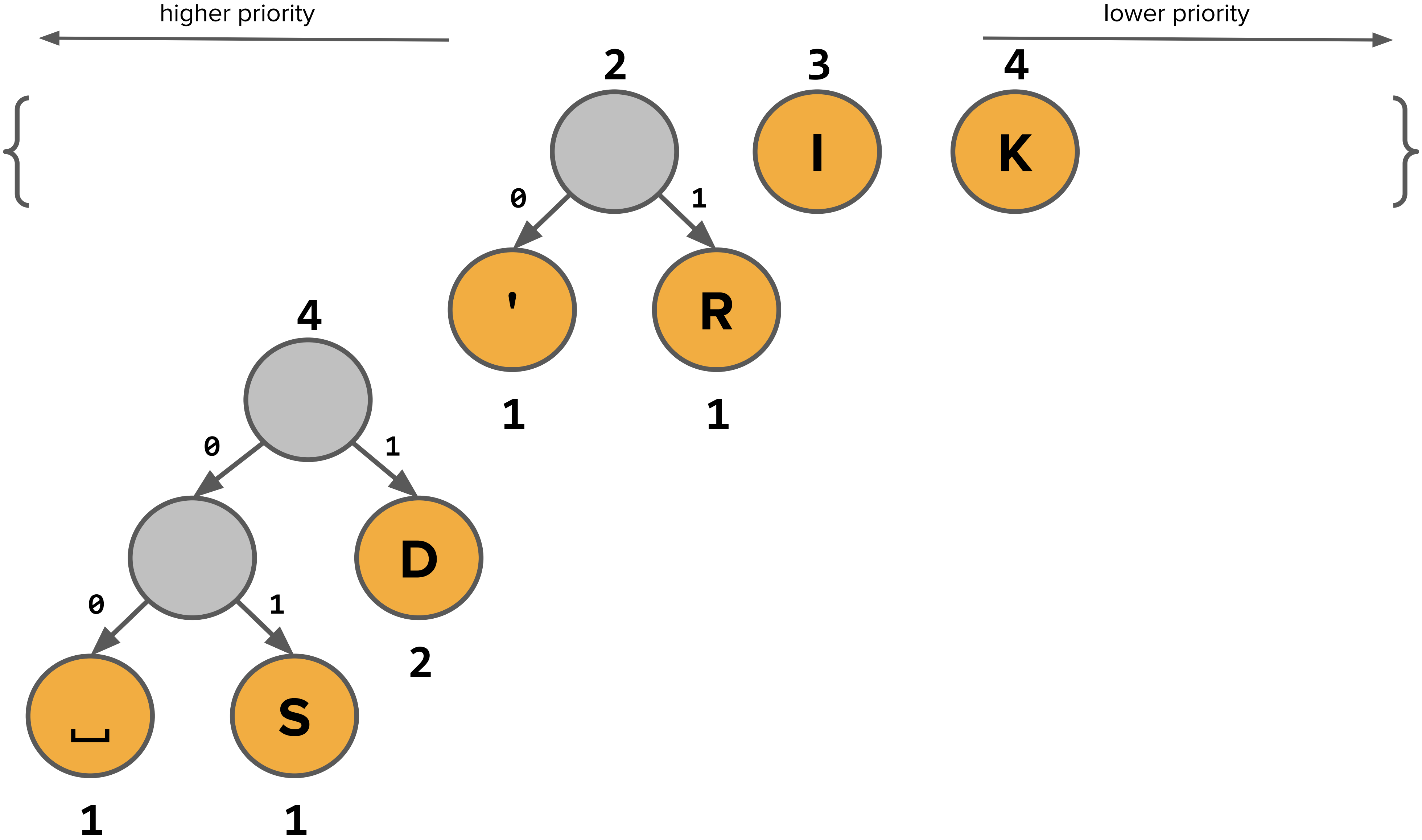

higher priority
lower priority
2
2
2
3
4
D
I
K
0
1
0
1
␣
S
'
R
1
1
1
1

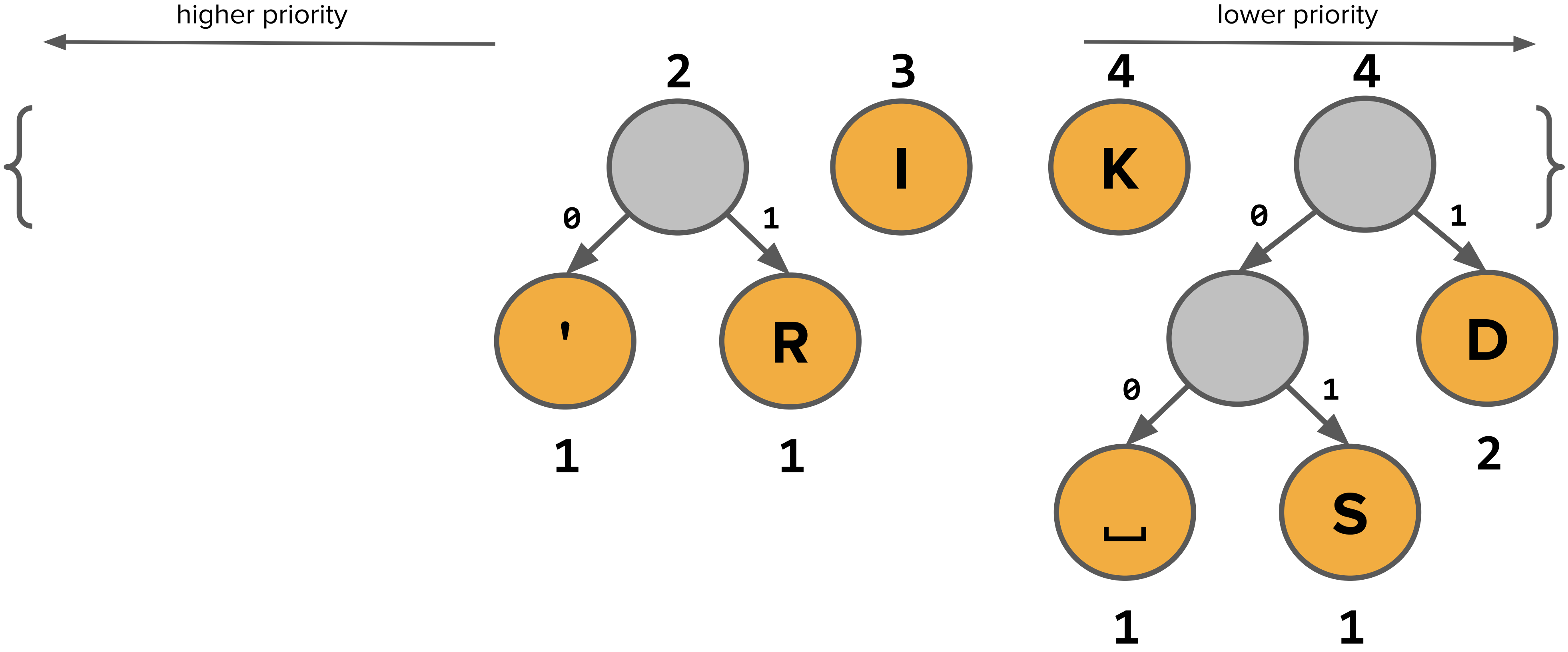

higher priority
lower priority
2
D
2
0
1
'
R
1
1
3
I
4
K
0
1
␣
S
1
1

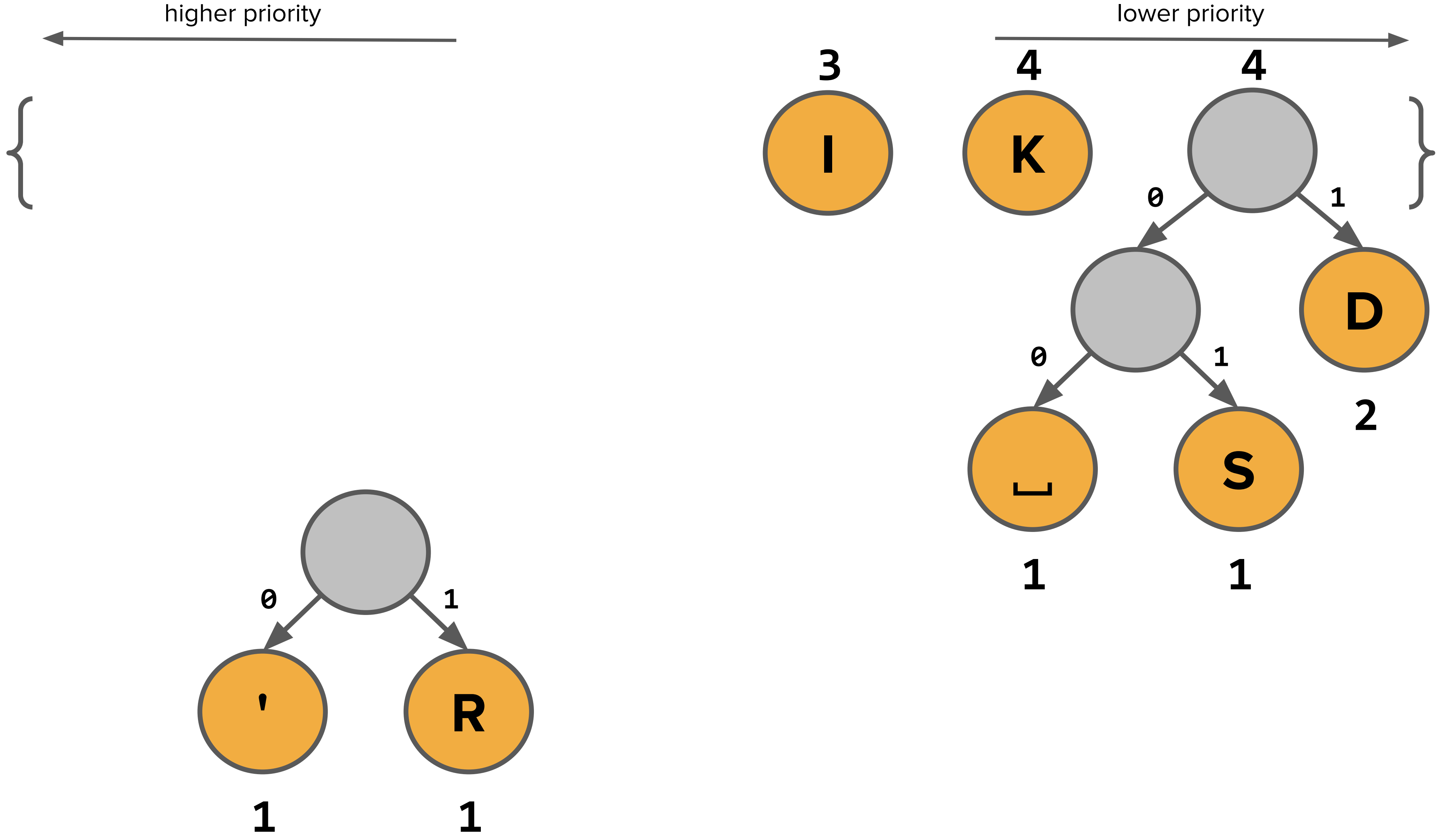

higher priority
lower priority
2
3
4
0
1
'
R
I
K
1
1
0
1
D
␣
S
2
1
1

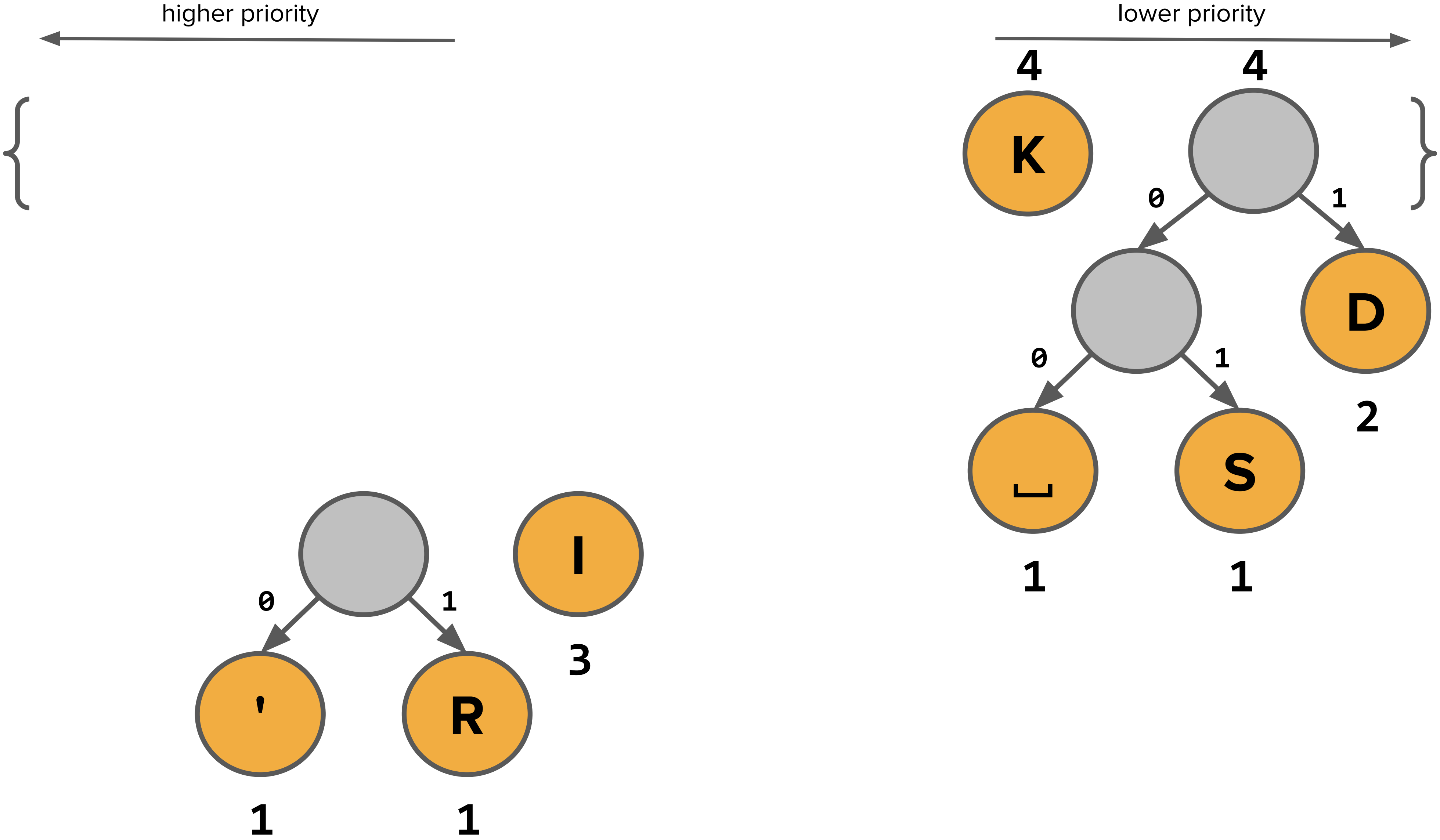

higher priority
lower priority
{
2
0
1
'
R
1
1
3
4
I
K
}
0
1
D
0
1
␣
S
2
1
1

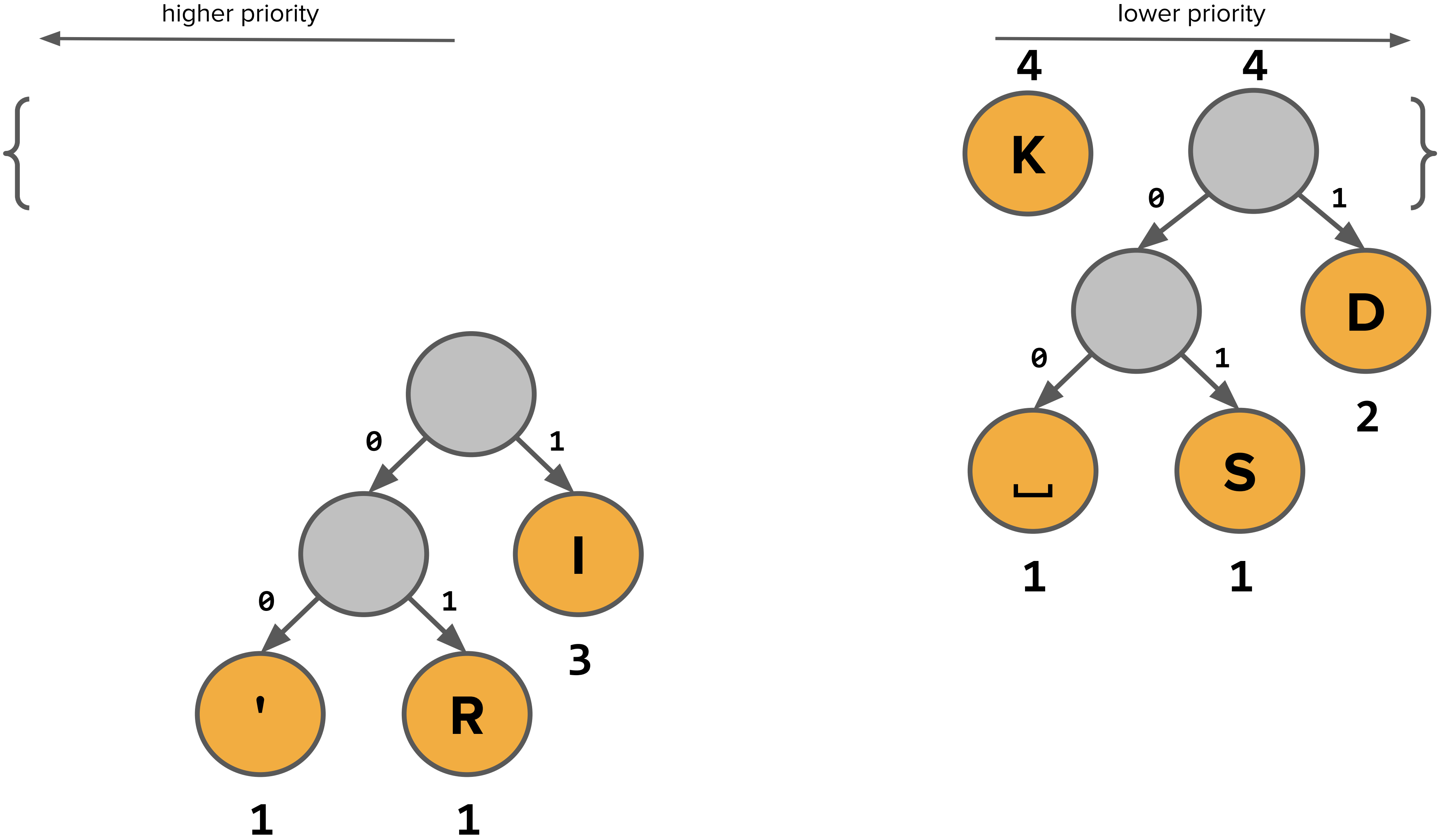

higher priority
lower priority
4
0
1
0
1
␣
S
D
1
1
2
2
0
1
'
R
1
1
3
4
I
K

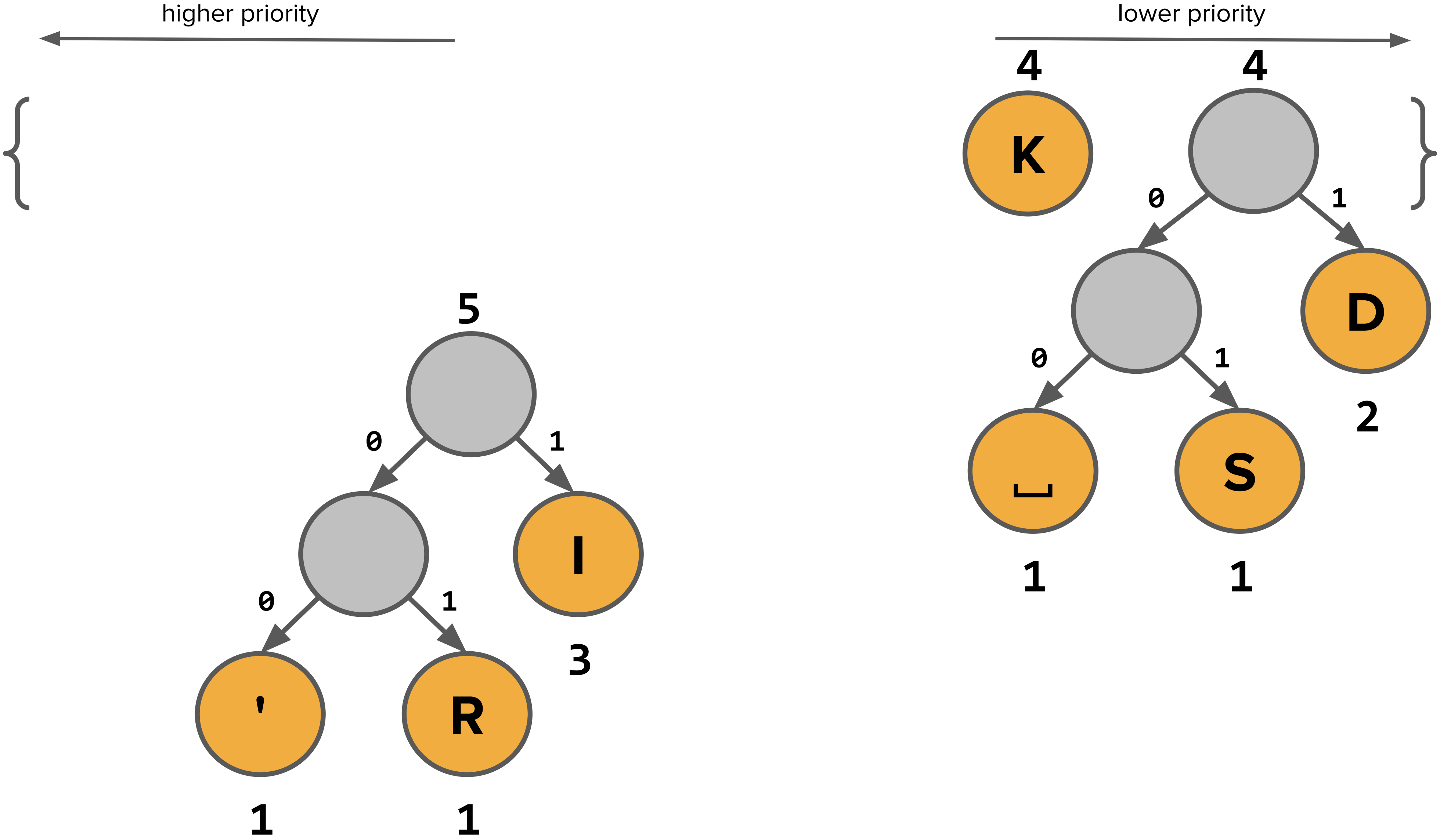

higher priority
lower priority
2
3
4
I
K
0
1
'
R
1
1
4
0
1
D
2
0
1
␣
S
1
1

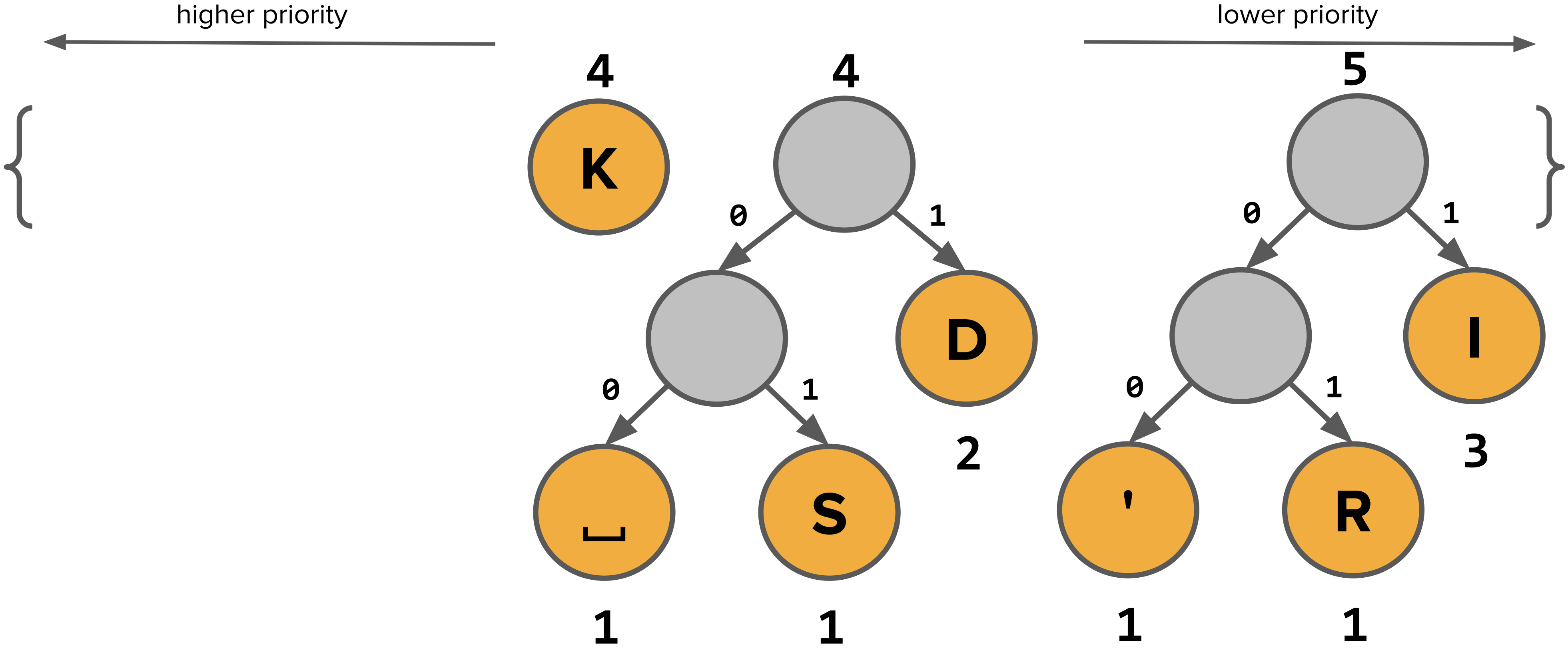

higher priority
lower priority
2
0
1
'
R
1
1
3
I
4
K
4
0
1
0
1
D
2
␣
S
1
1


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


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

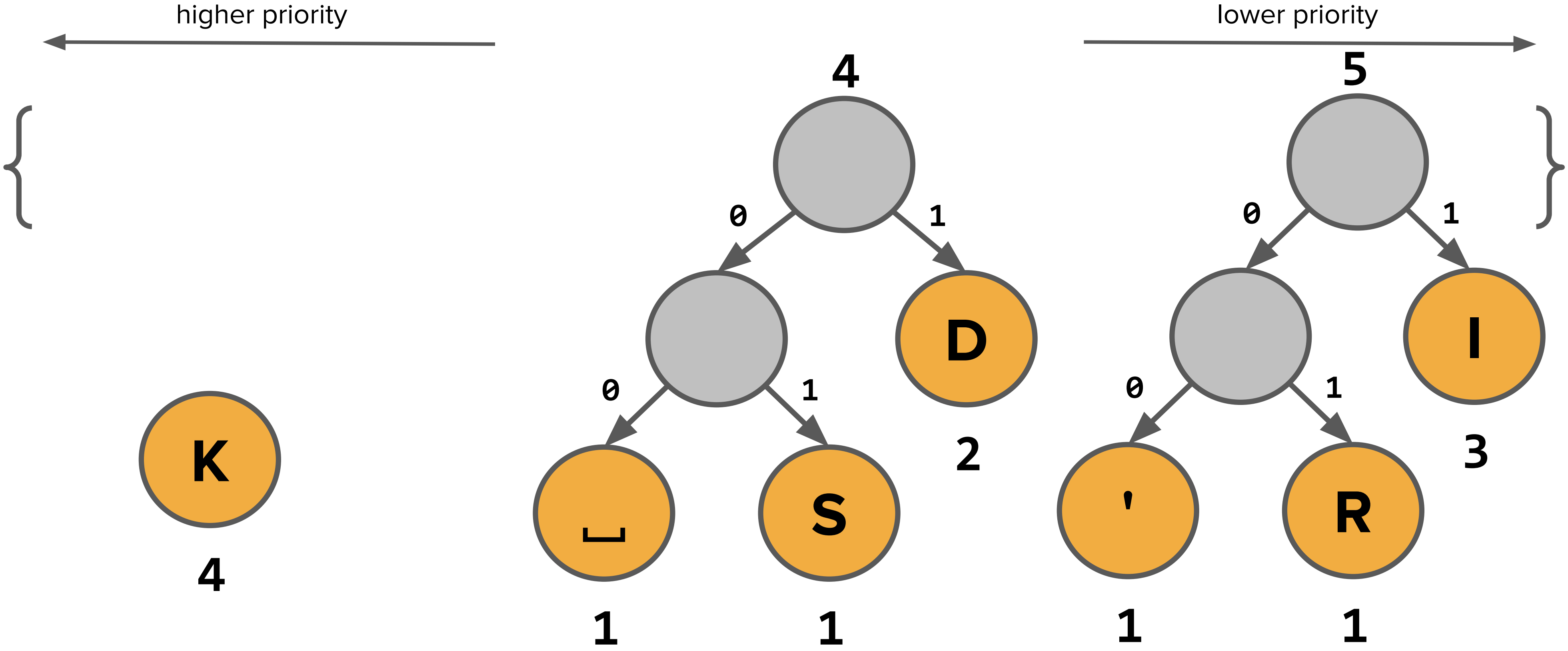

higher priority
lower priority
4
K
4
0
1
D
2
0
1
␣
S
1
1
0
1
I
3
0
1
'
R
1
1

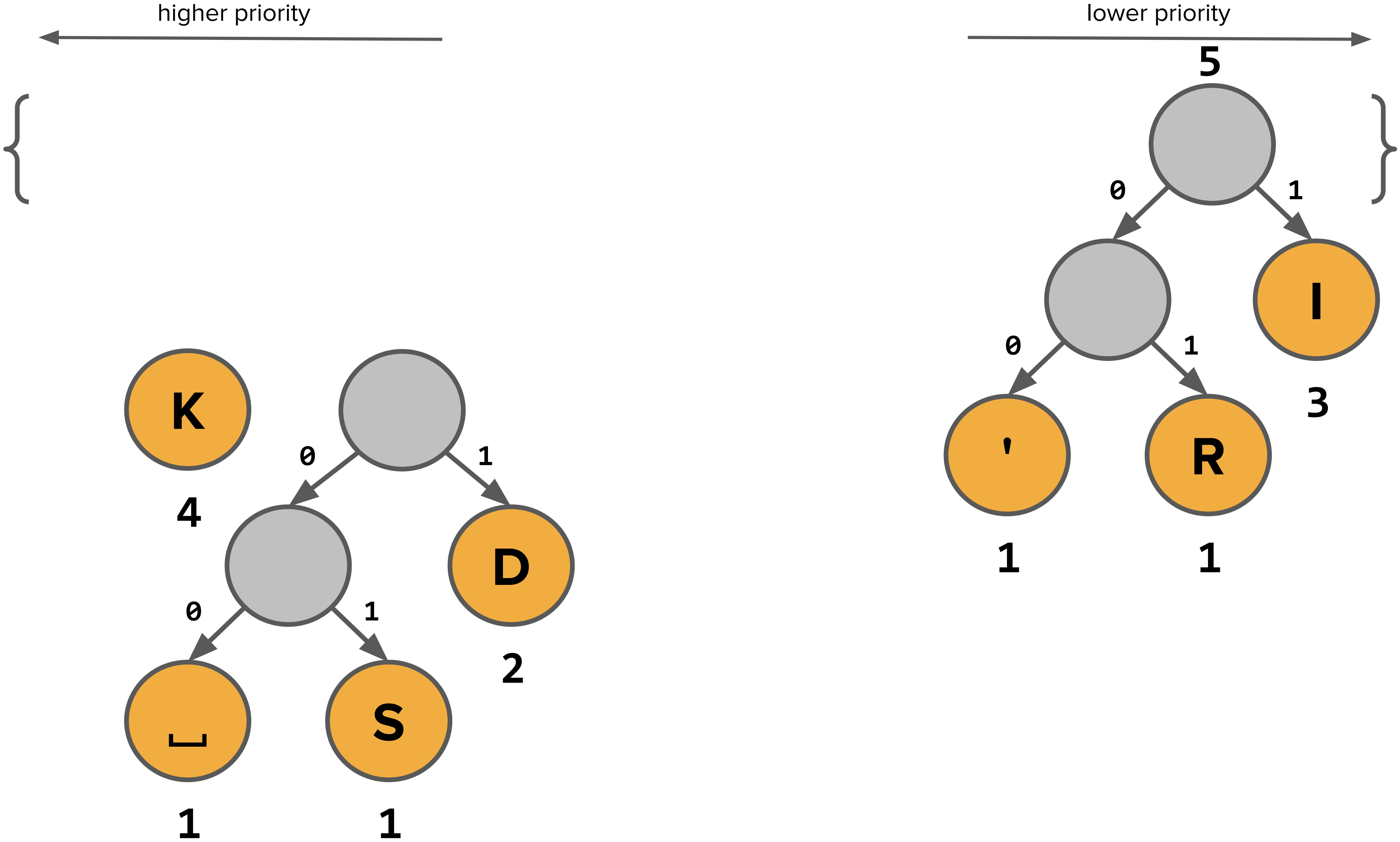

higher priority
lower priority
5
0
1
0
1
'
R
I
1
1
3
4
K
4
0
1
0
1
␣
S
D
1
1
2

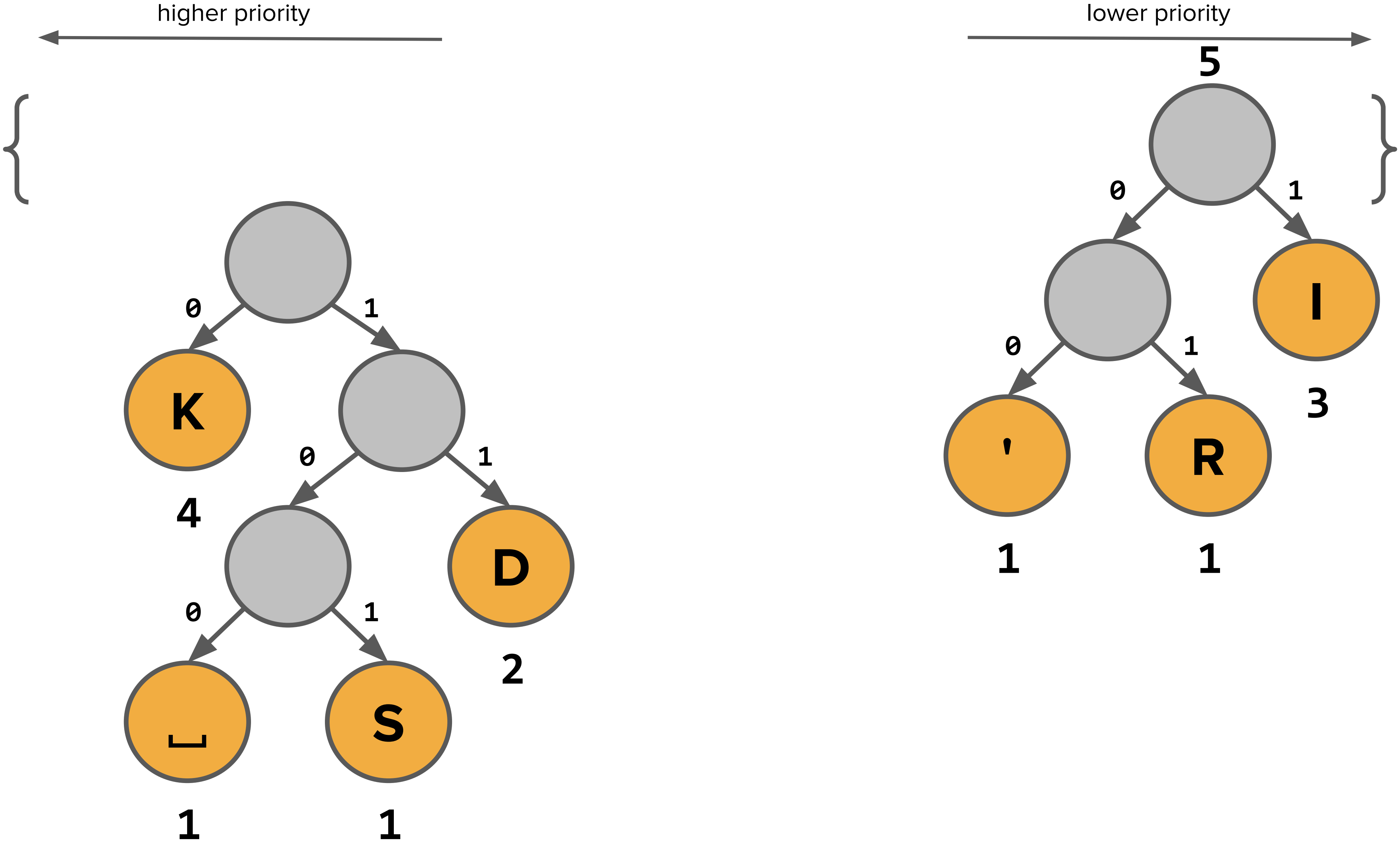


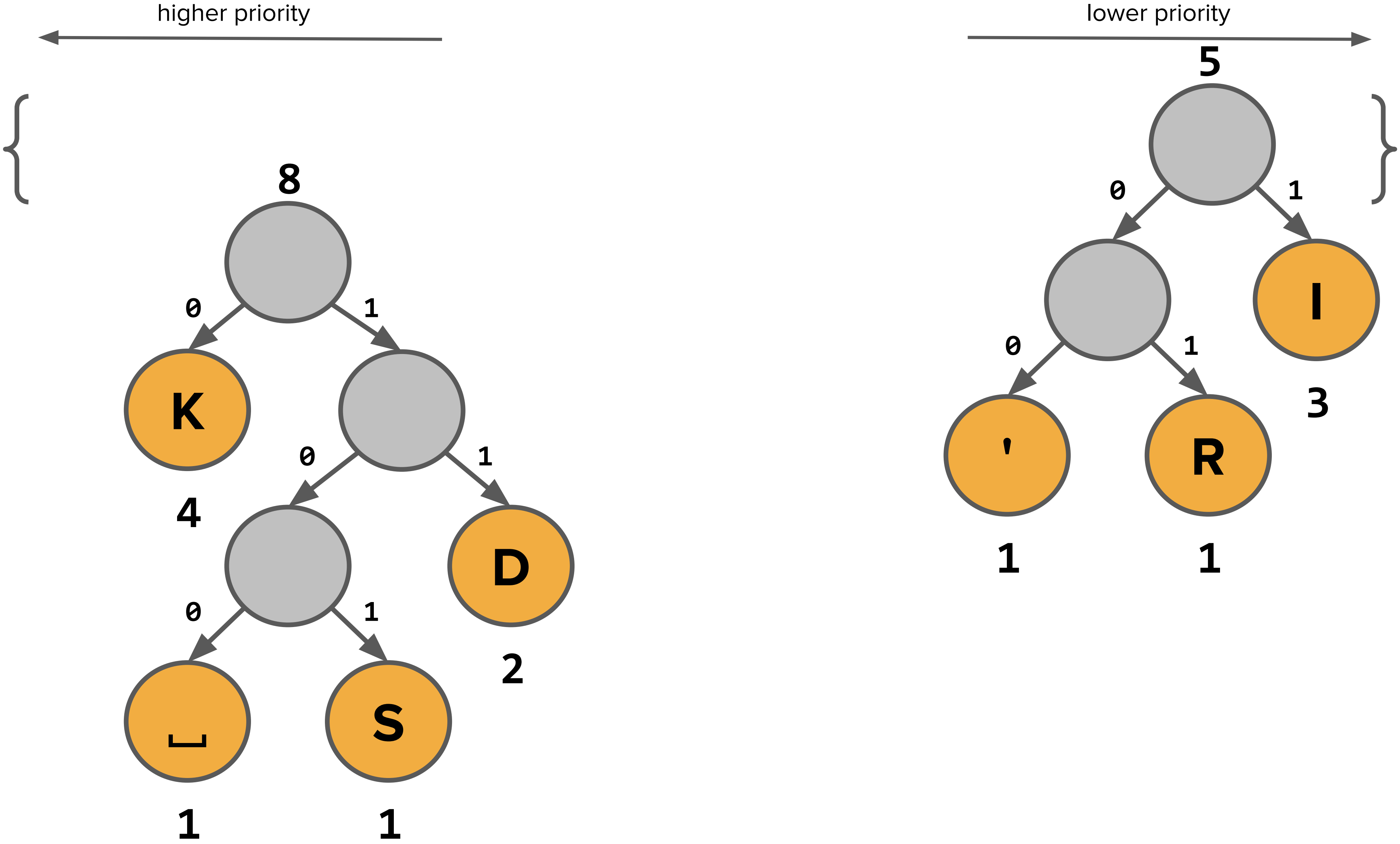
higher priority

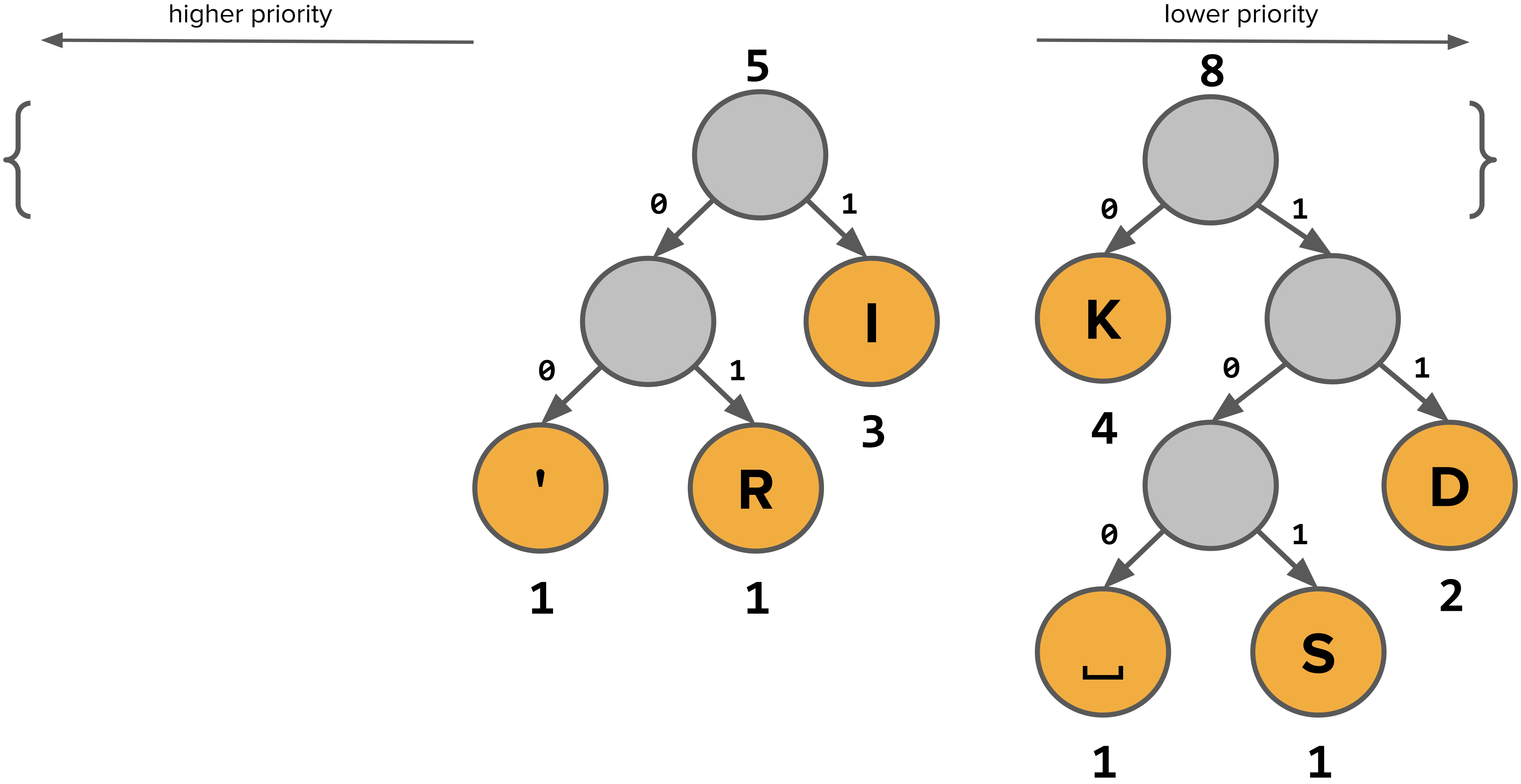
lower priority

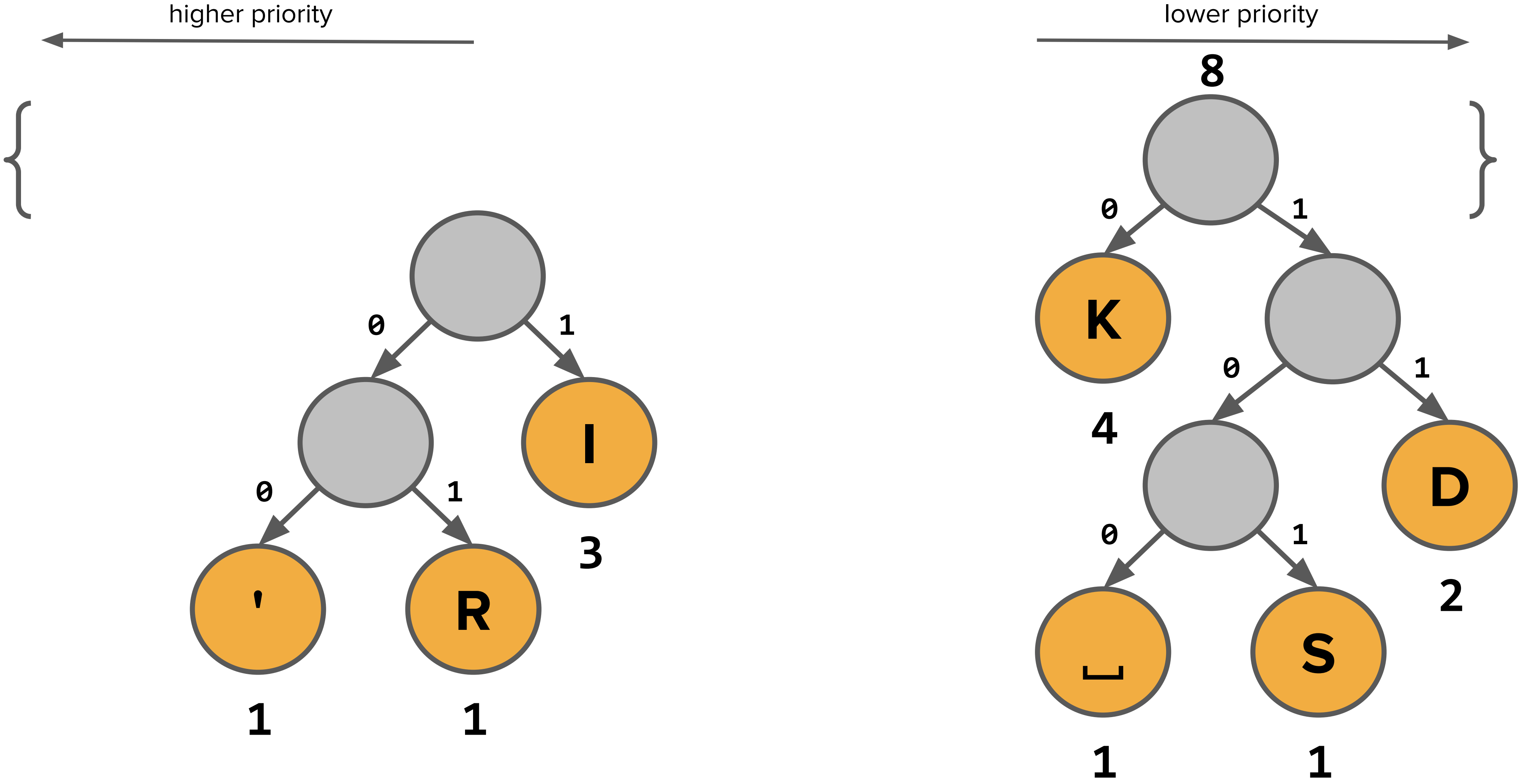

higher priority
lower priority
K
4
4
0
1
0
1
␣
S
D
1
1
2
5
0
1
0
1
'
R
I
1
1
3

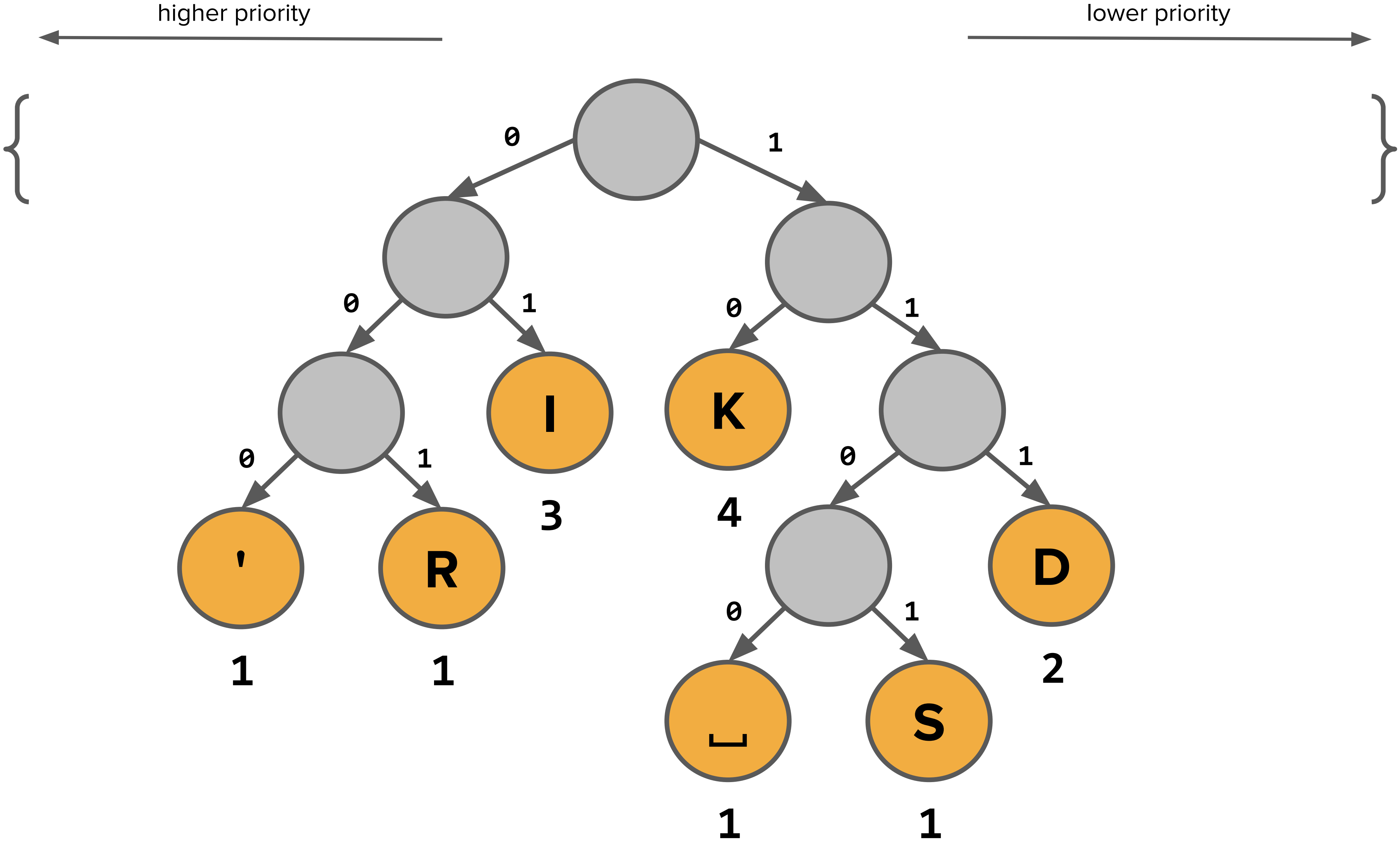

higher priority
lower priority
K
4
0
1
0
1
␣
S
D
1
1
2
5
0
1
0
1
'
R
I
1
1
3

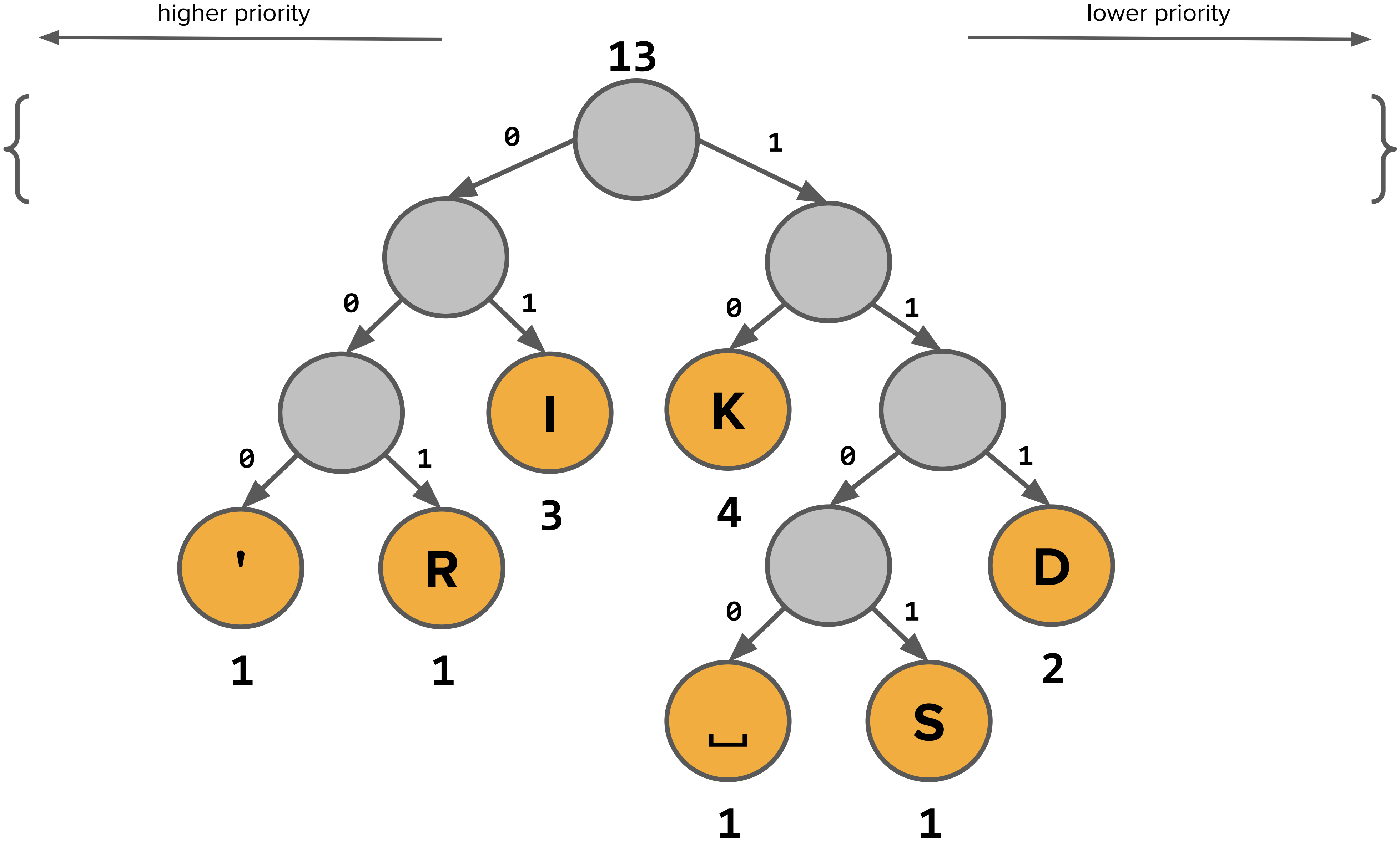

higher priority
lower priority
0
1
K
4
0
1
D
2
0
1
␣
1
S
1
5
0
1
I
3
0
1
'
1
R
1

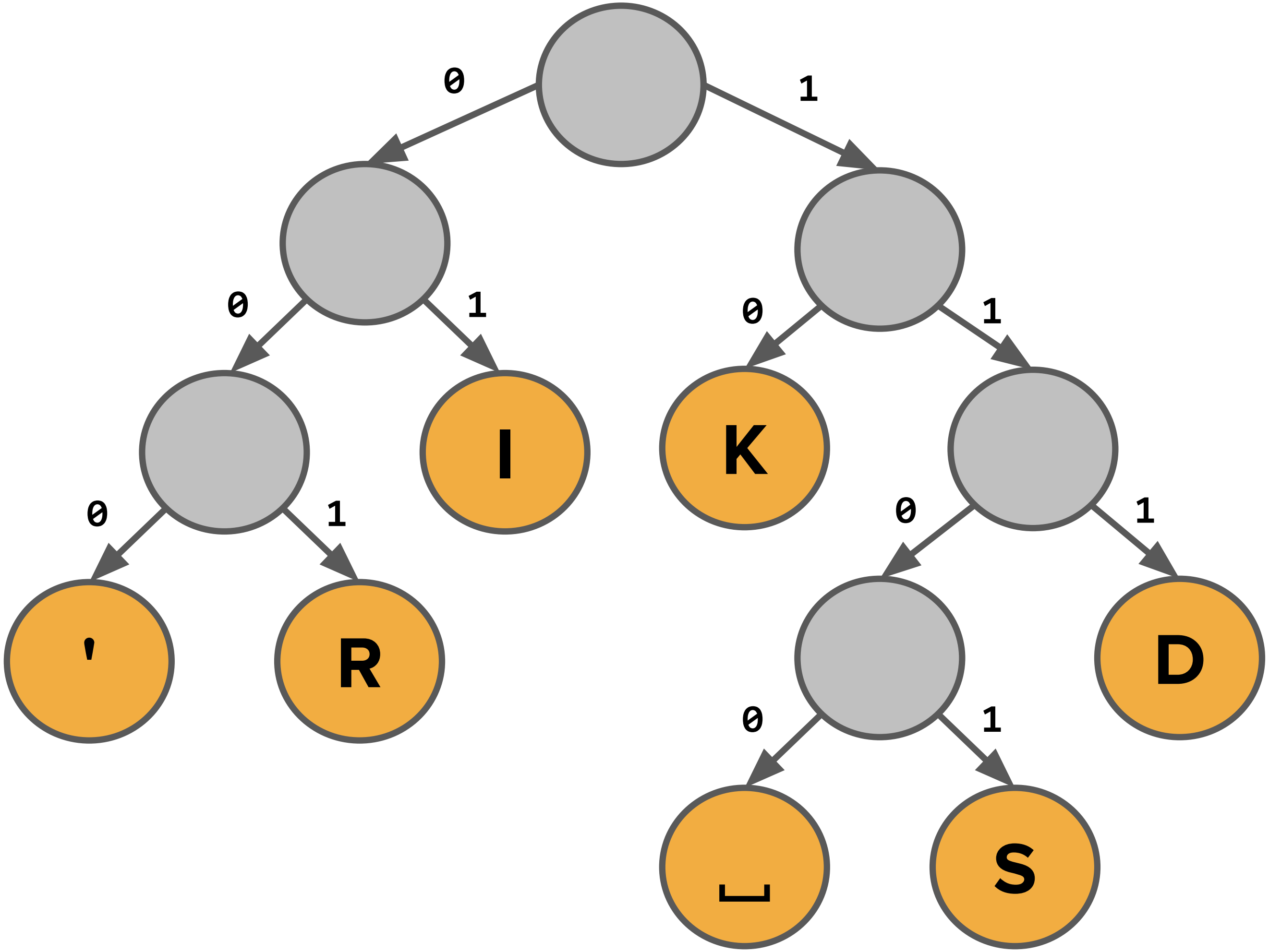

higher priority
lower priority
{
8
0
1
K
4
0
1
0
1
␣
1
S
1
D
2
5
0
1
0
1
'
1
R
1
I
3
}


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

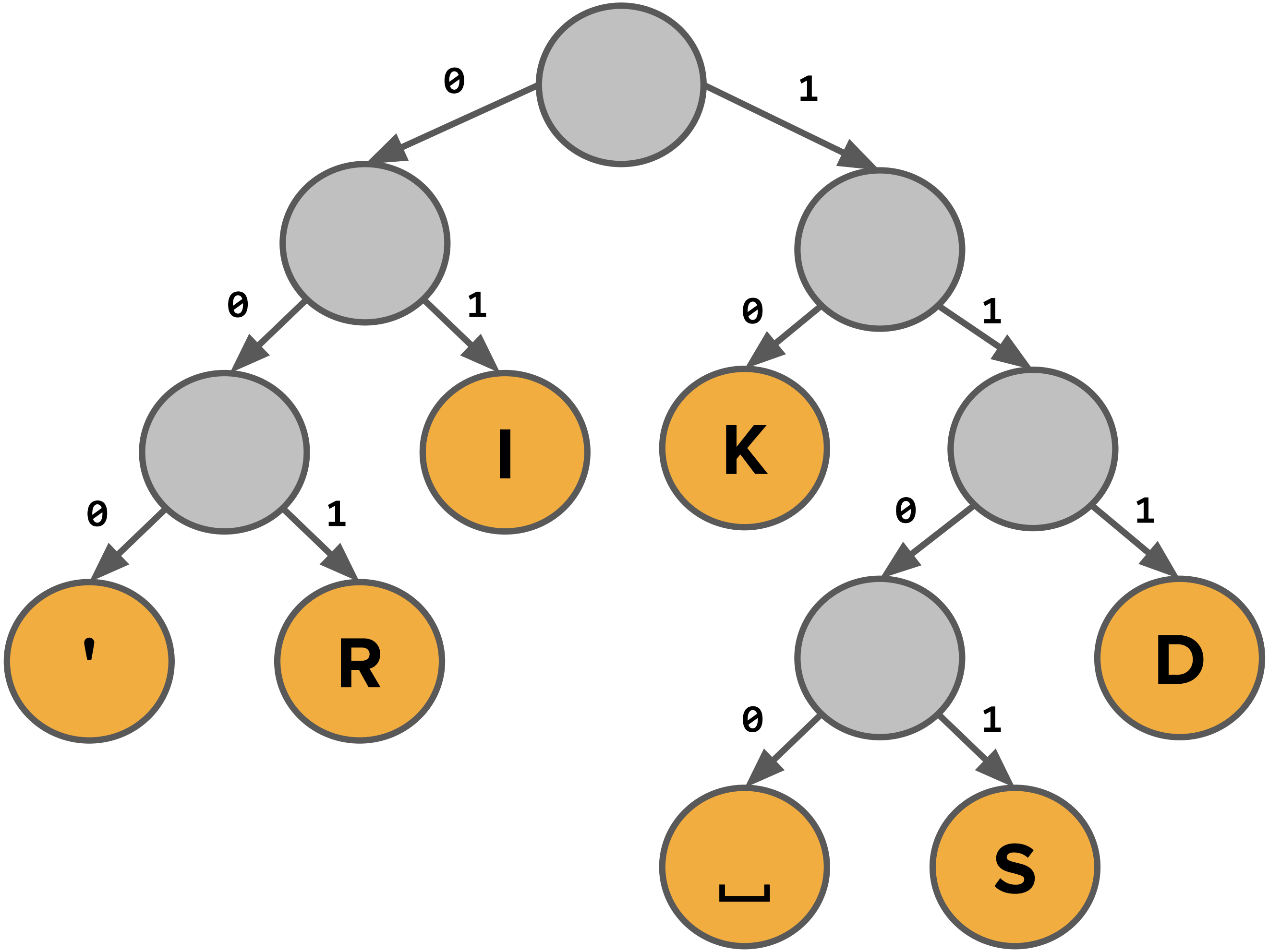

higher priority
lower priority
0
1
0
1
'
R
I
1
1
3
8
0
1
K
4
0
1
D
2
0
1
␣
S
1
1


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

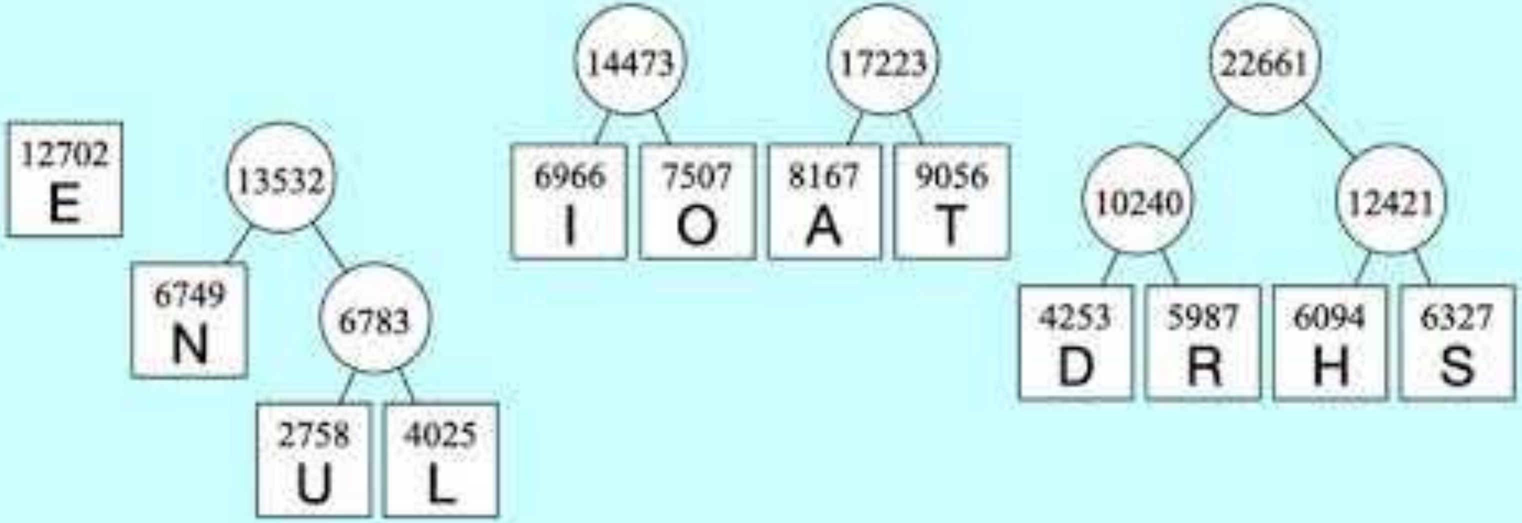

higher priority
lower priority
13
0
1
0
1
0
1
0
1
0
1
0
1
'
R
I
K
D
␣
S
1
1
3
4
2
1
1


0
1
0
1
0
1
0
1
0
1
0
1
'
R
I
K
D
␣
S

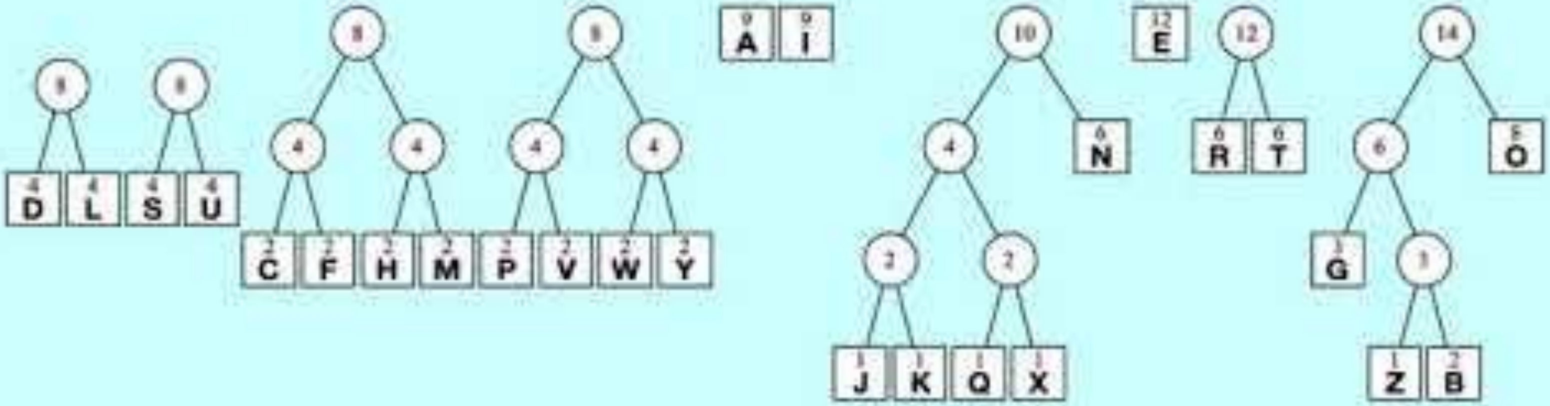
character	*code*
K	10
I	01
D	111
R	001
'	000
S	1101
␣	1100



Illustrating the Huffman Algorithm



The Huffman Tree for Scrabble Tiles



One important final detail...

Prefix Codes Example

```
100100110000011011100
11101101110110
```

So far we've only thought about transmitting the compressed message.

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

Prefix Codes Example

```
1001001100000110111100
11101101110110
```

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

But we need this information in order to be able to decompress.

Prefix Codes Example

```
10010011000011011100
11101101110110
```

Prefix Codes Example

10010011000011011100
11101101110110

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. This might increase the total file size in some cases!

- **Theorem:** There is no compression algorithm that can always compress all inputs.
  - **Proof:** Take CS103!

Summary

Huffman Encoding Summary

- Data compression is a very important real-world problem that relies on patterns in data to find efficient, compact data representations schemes.

- In order to support variable-length encodings for data, we must use prefix coding schemes. Prefix coding schemes can be modeled as binary trees.

- Huffman encoding uses a greedy algorithm to construct encodings by building a tree from the bottom up, putting the most frequent characters higher up in the coding tree.

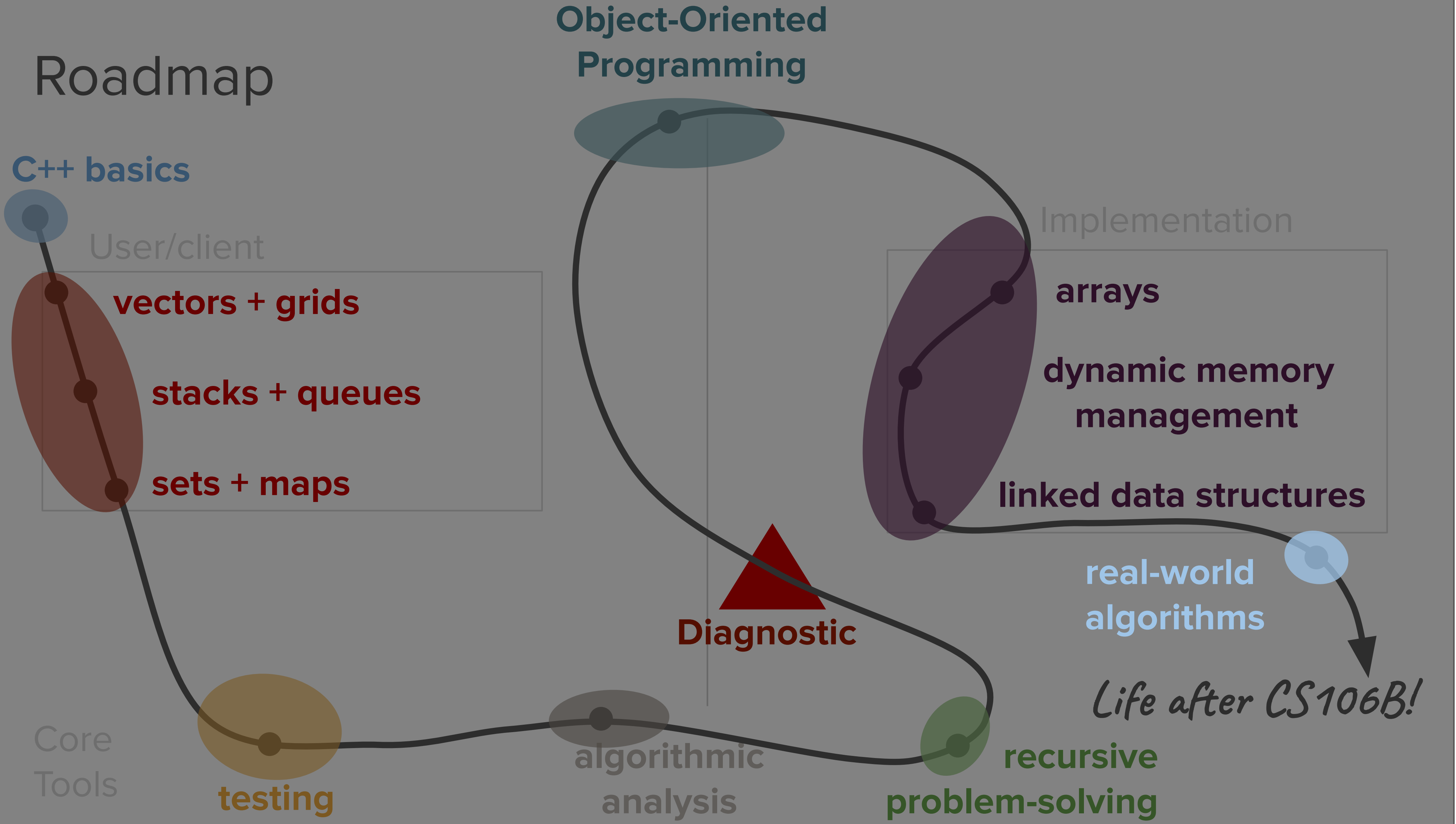
- We need to send the encoding table with the compressed message.

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!

What’s next?

Roadmap



C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Core Tools

testing

algorithmic analysis

recursive problem-solving

Diagnostic

Object-Oriented Programming

Implementation

arrays

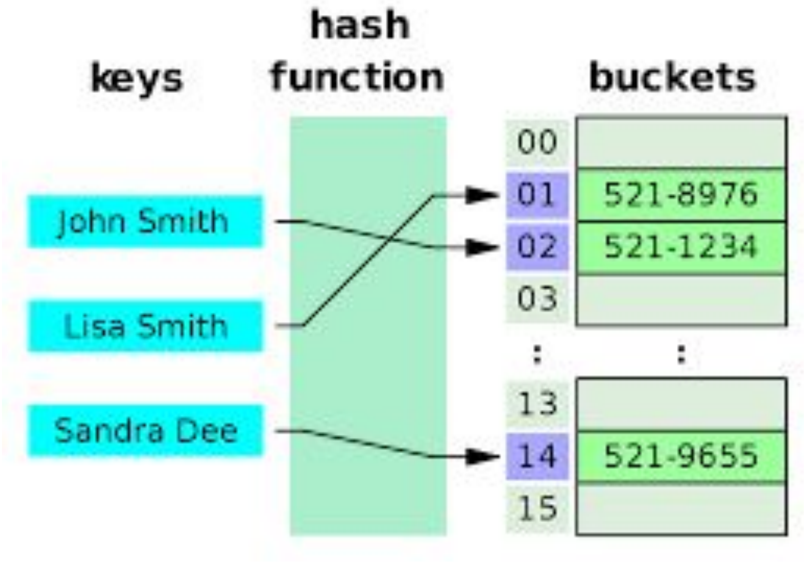
dynamic memory management

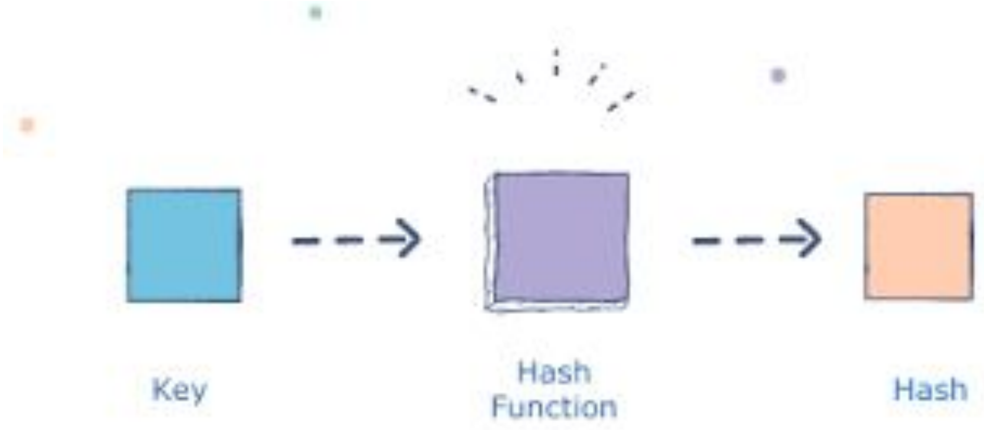
linked data structures

real-world algorithms

Life after CS106B!

Hashing


hash function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
:
13
14
15
521-8976
521-1234
:
521-9655


Key
Hash Function
Hash