# Hashing

Elyse Cornwall

August 9, 2023

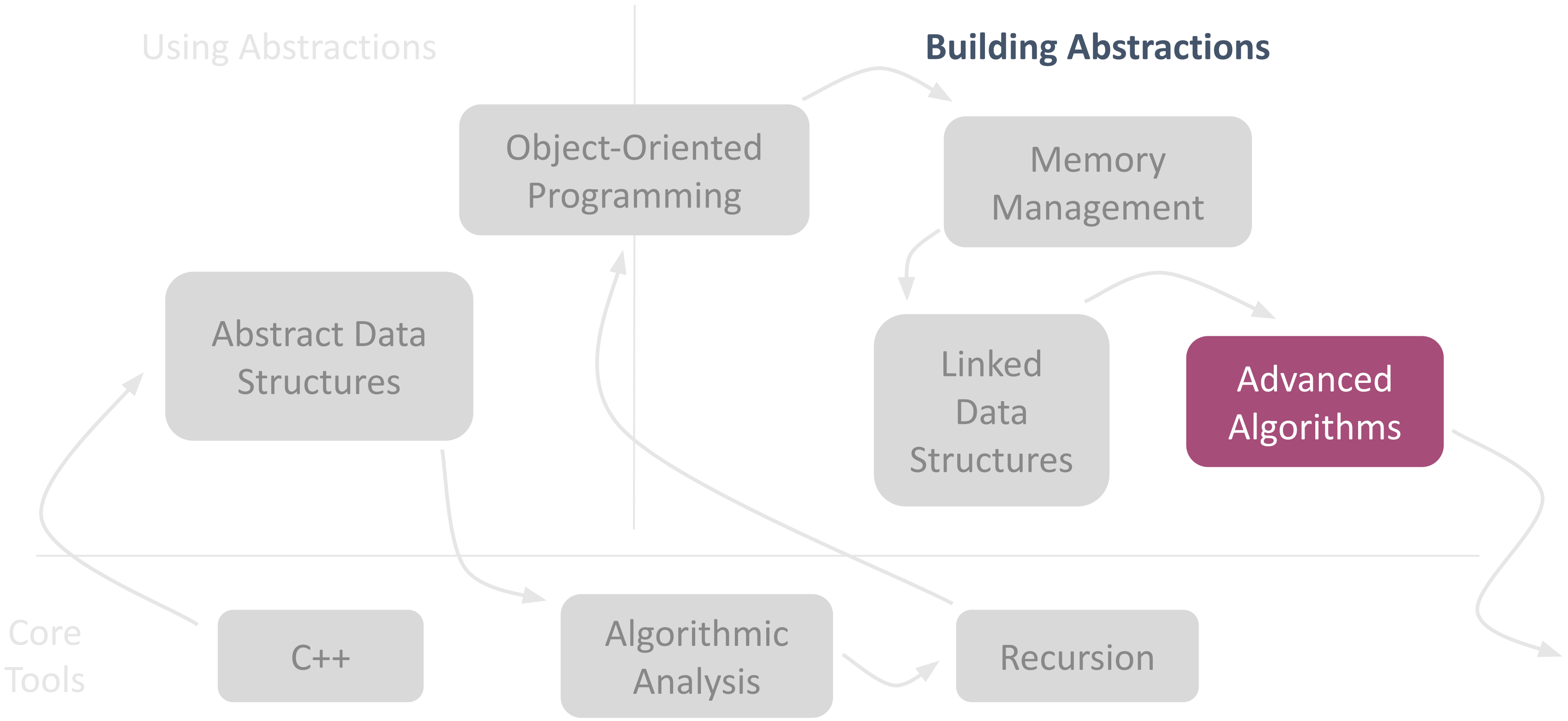
Contributions made from previous CS106B Instructors

Stanford University

# Announcements

- Assignment 5 is due tonight
- Assignment 6 (last assignment!) comes out this afternoon
  - **No late days** beyond the grace period (next Thursday 11:59pm)
  - YEAH hours 4-5pm
- Exam next Friday (8/18) from 3:30-6:30pm
  - Final exam info is published on the website under "Assessments"
  - Final review session next Tuesday in class

# Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

# Recap: Huffman Coding

# ASCII Encoding

- ASCII uses 8 bits to represent each character
- Let's represent **`KIRK'S DIKDIK`** in ASCII code

| *character* | *ASCII code* |
|---|---|
| K | 01001011 |
| I | 01001001 |
| R | 01010010 |
| ' | 00100111 |
| S | 01010011 |
| _ | 00100000 |
| D | 01000100 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0100 1011 | 0100 1001 | 0101 0010 | 0100 1011 | 0010 0111 | 0101 0011 | 0010 0000 | 0100 0100 | 0100 1001 | 0100 1011 | 0100 0100 | 0100 1001 | 0100 1011 |
| K | I | R | K | ' | S | _ | D | I | K | D | I | K |

# A Different Encoding

What is the mystery word represented by this 3-bit encoding?

010 | 001 | 110

R I D

| *character* | *code* |
|---|---|
| K | 000 |
| I | 001 |
| R | 010 |
| ’ | 011 |
| S | 100 |
| _ | 101 |
| D | 110 |

# Our New Code

**KIRK'S_DIKDIK**

**010101011110000100010**

**RRRRI_KK'D'**

| *character* | *frequency* | *code* |
|---|---|---|
| K | 4 | 0 |
| I | 3 | 1 |
| D | 2 | 00 |
| R | 1 | 01 |
| ' | 1 | 10 |
| S | 1 | 11 |
| _ | 1 | 100 |

# Prefix Code

- 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 |

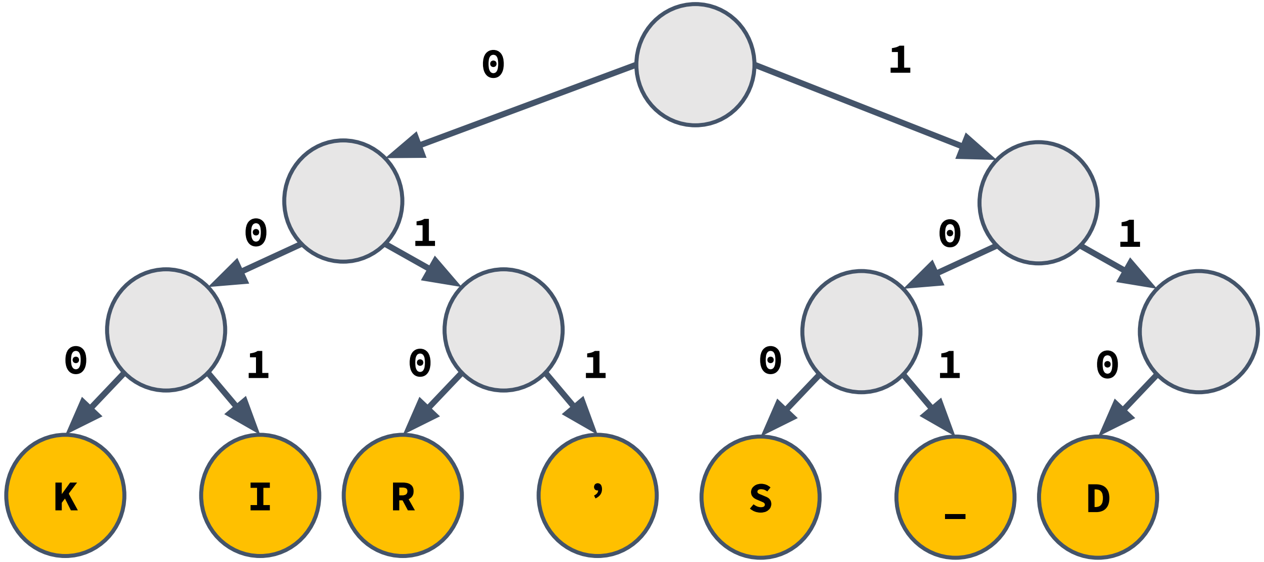
| | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 01 | 001 | 10 | 000 | 1101 | 1100 | 111 | 01 | 10 | 111 | 01 | 10 |
| K | I | R | K | ' | S | _ | D | I | K | D | I | K |

What is the mystery word represented by this encoding?

110001010

# Coding Tree

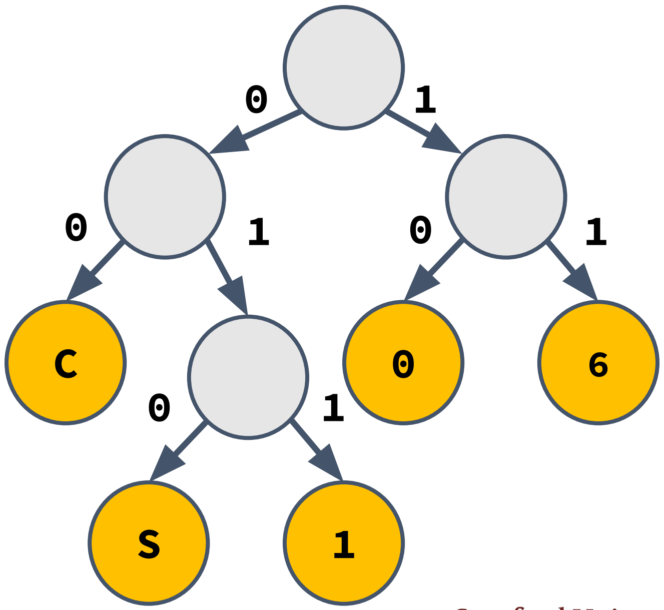
- We can represent a prefix coding scheme using a binary tree, which is called a coding tree




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

# Coding Trees

- A coding tree is valid if all the letters are stored in the leaves, with internal nodes only used for routing



# 1. Build the frequency table

Input text: **KIRK'S DIKDIK**

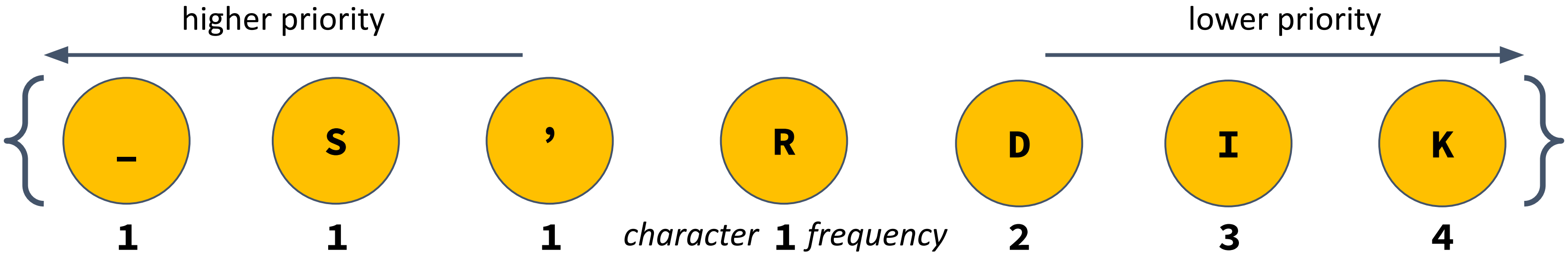
| *character* | *frequency* |
|---|---|
| K | 4 |
| I | 3 |
| R | 1 |
| ' | 1 |
| S | 1 |
| _ | 1 |
| D | 2 |

# 2. Initialize an empty priority queue

higher priority ← → lower priority

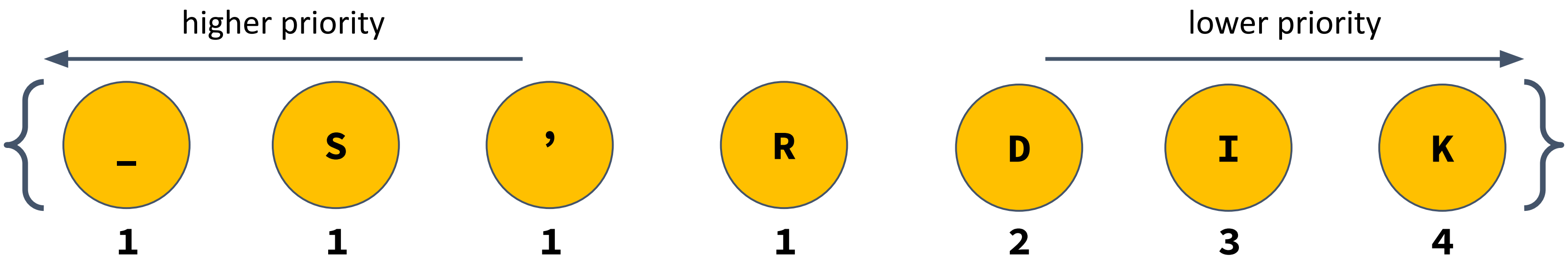
{ }

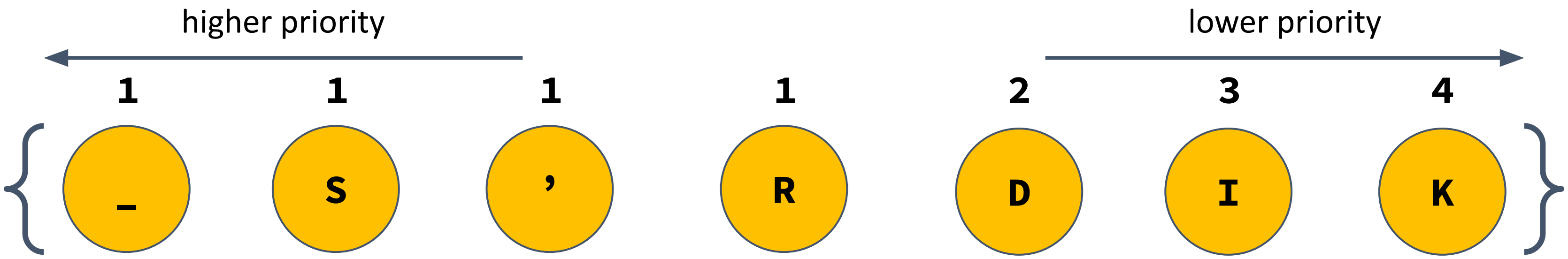
# 3. Add all unique characters as leaf nodes to queue

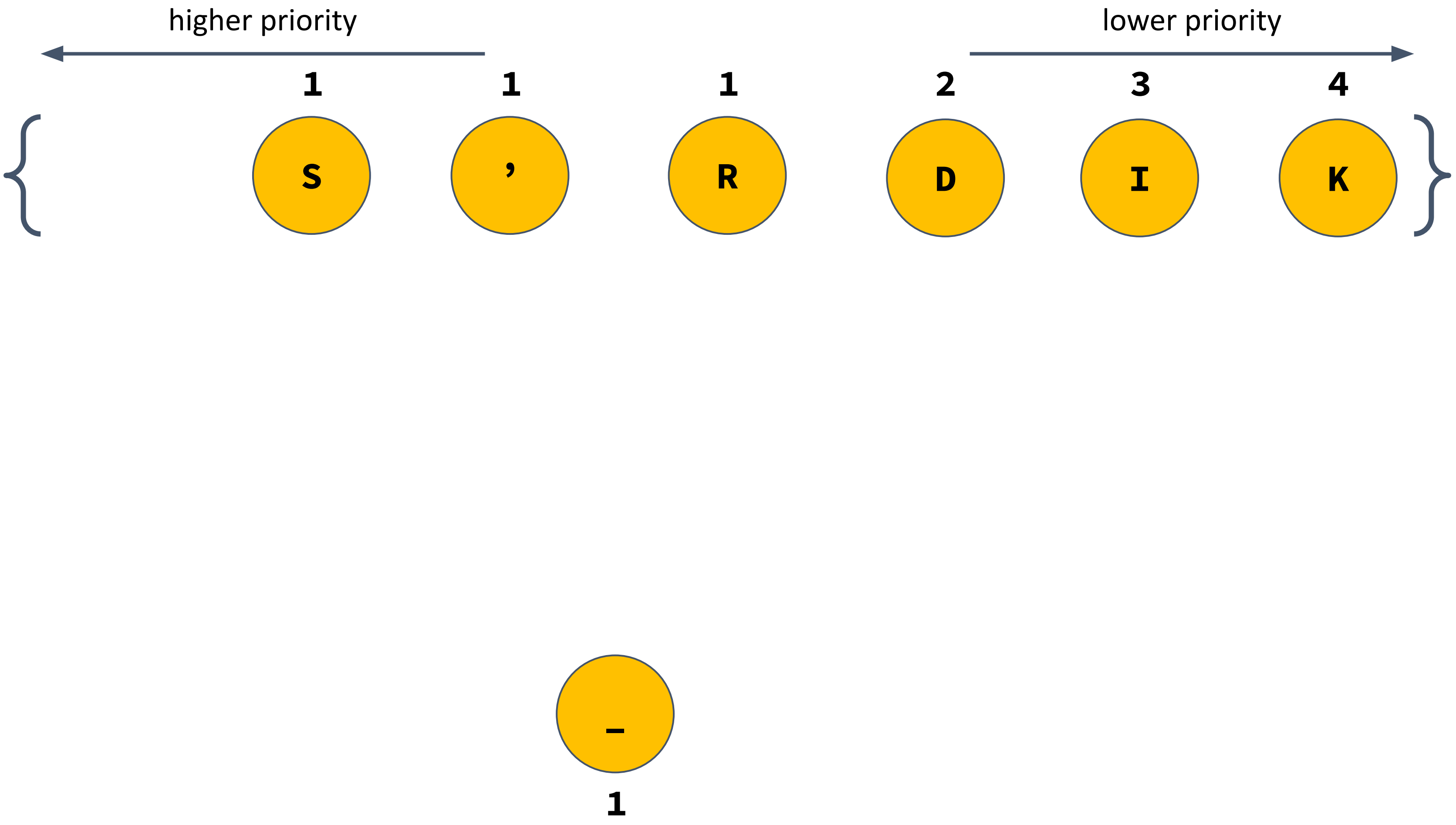


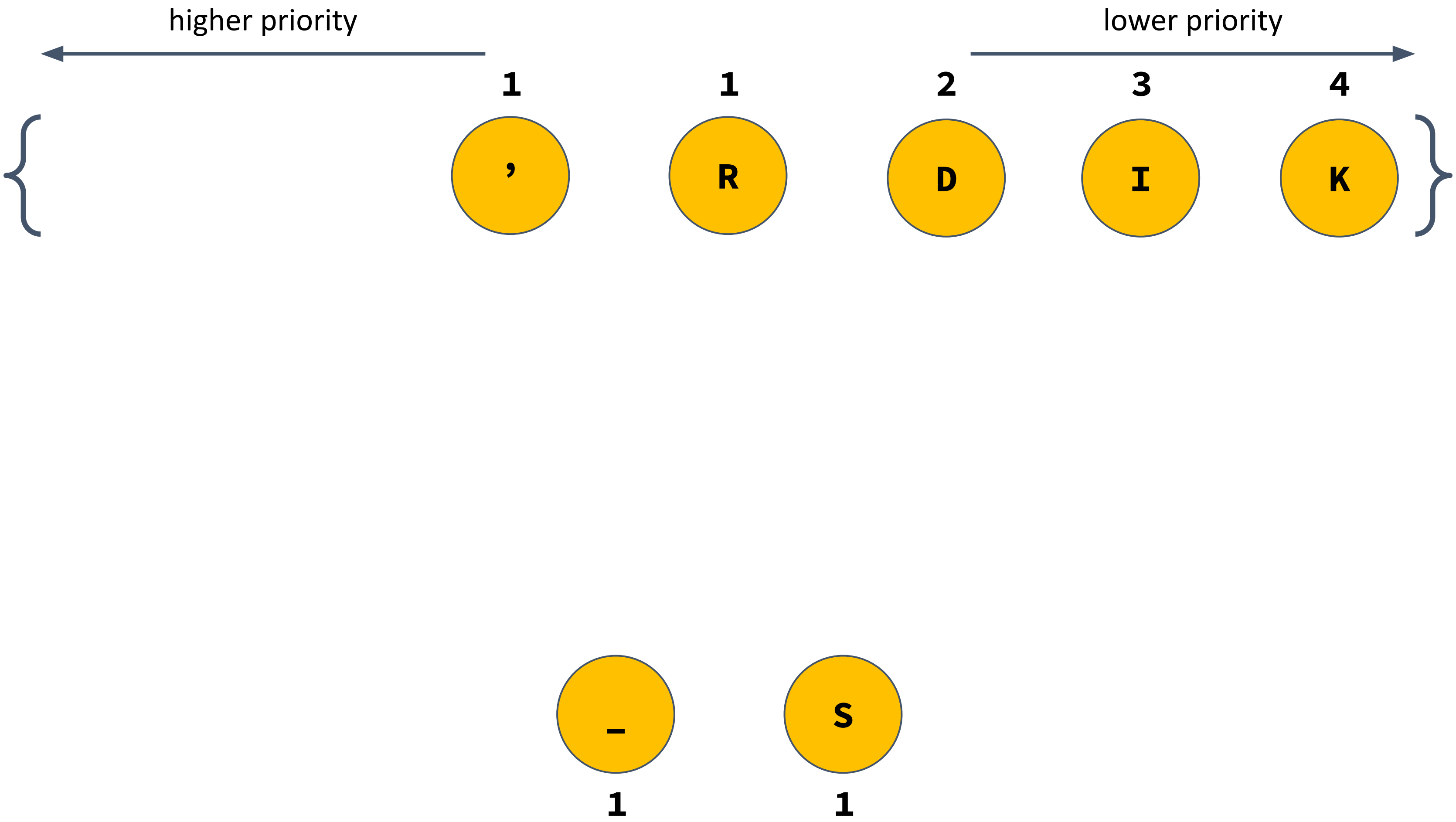

| character | frequency |
| --- | --- |
| K | 4 |
| I | 3 |
| R | 1 |
| ’ | 1 |
| S | 1 |
| _ | 1 |
| D | 2 |

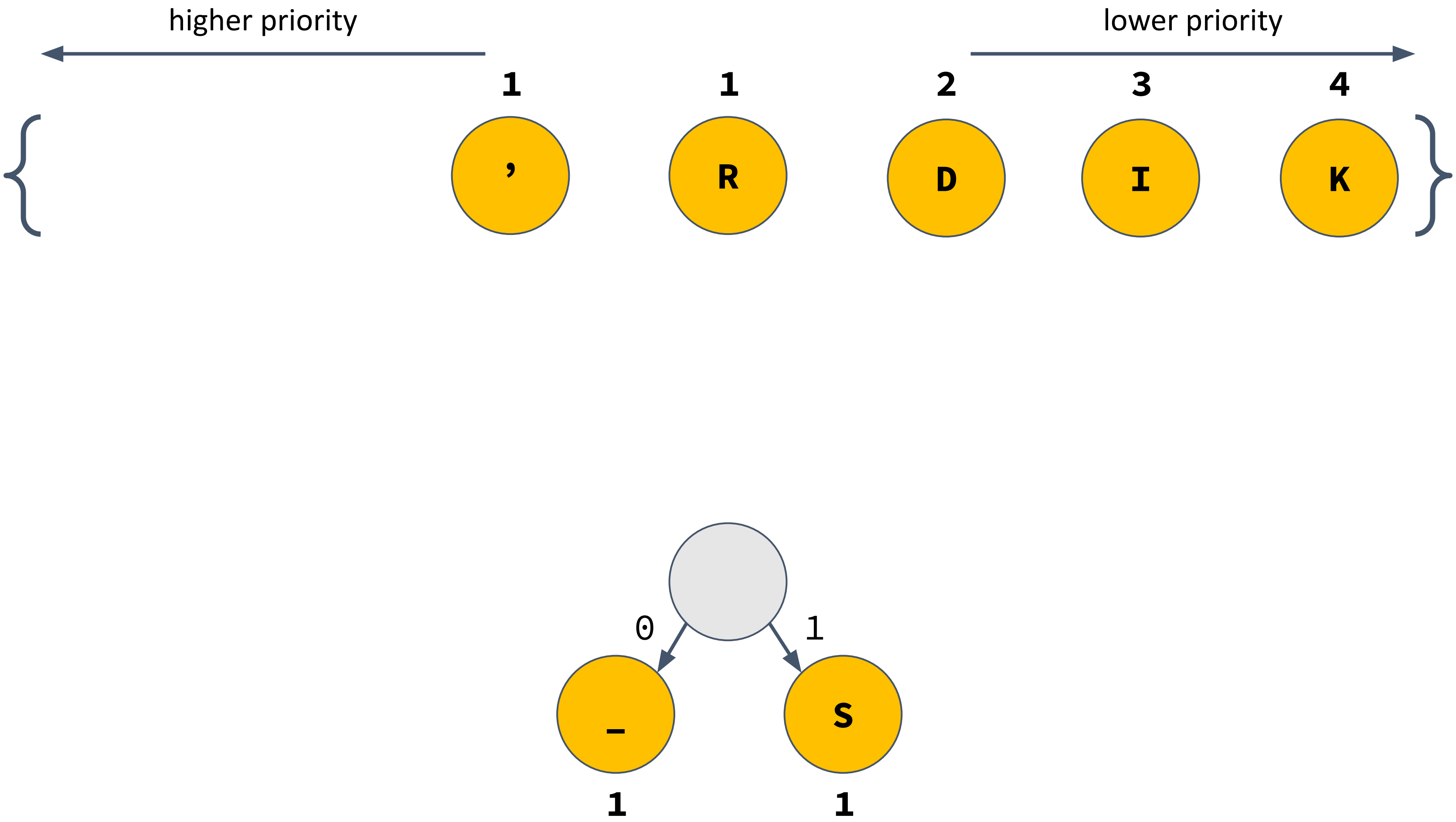
# 4. Build the Huffman tree by merging nodes

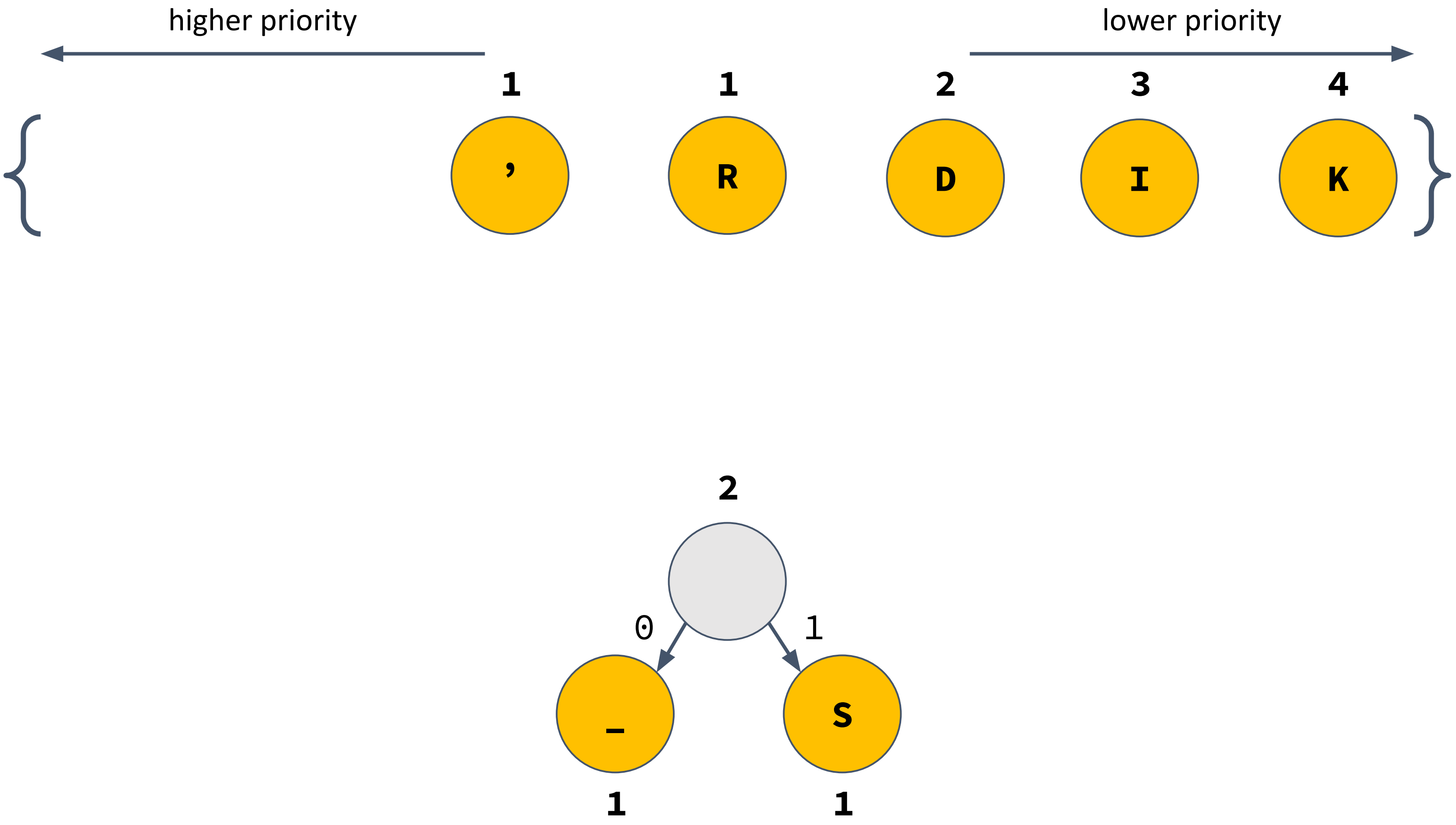


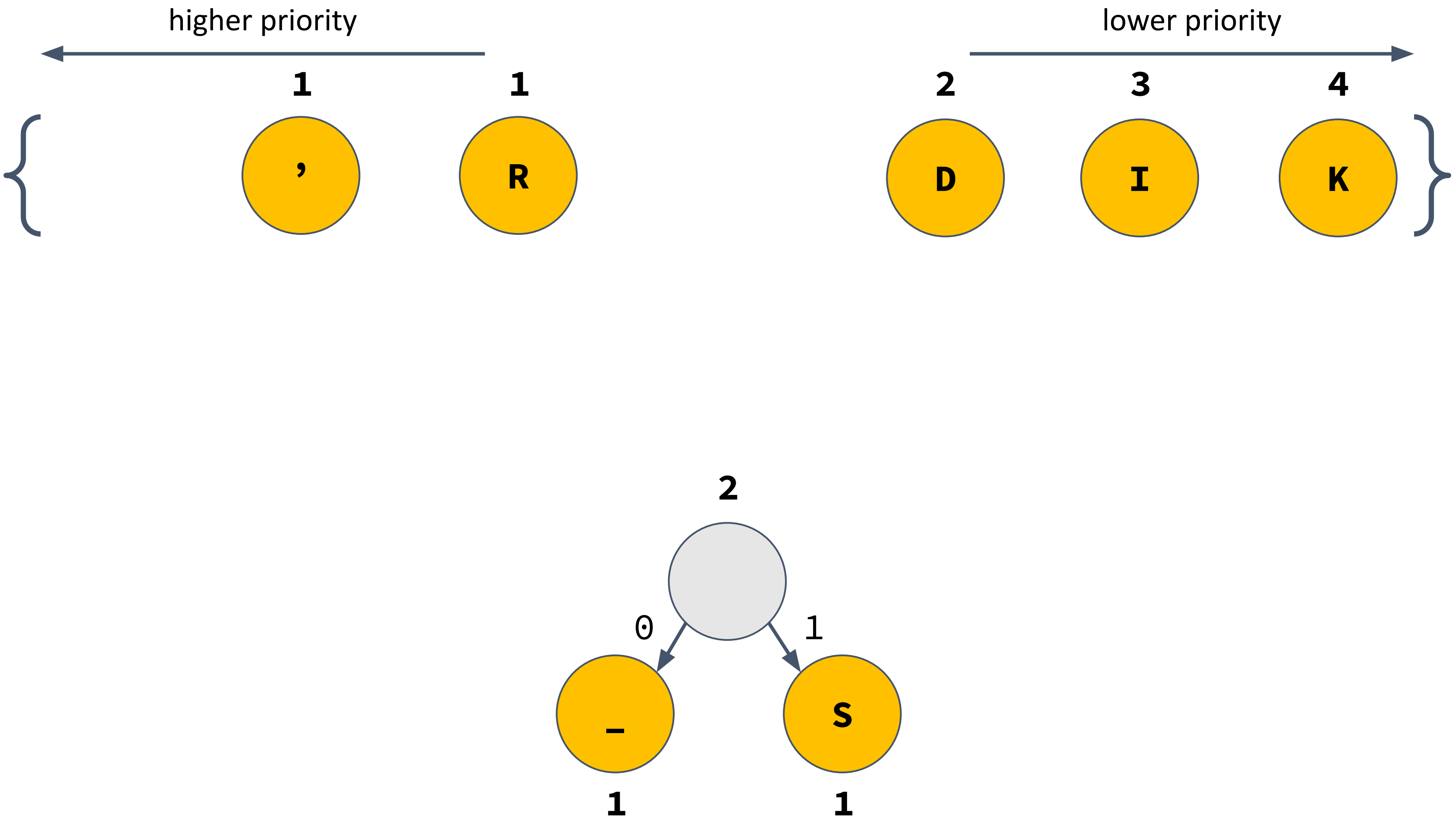

higher priority
lower priority
1
1
1
1
2
3
4
_
S
'
R
D
I
K

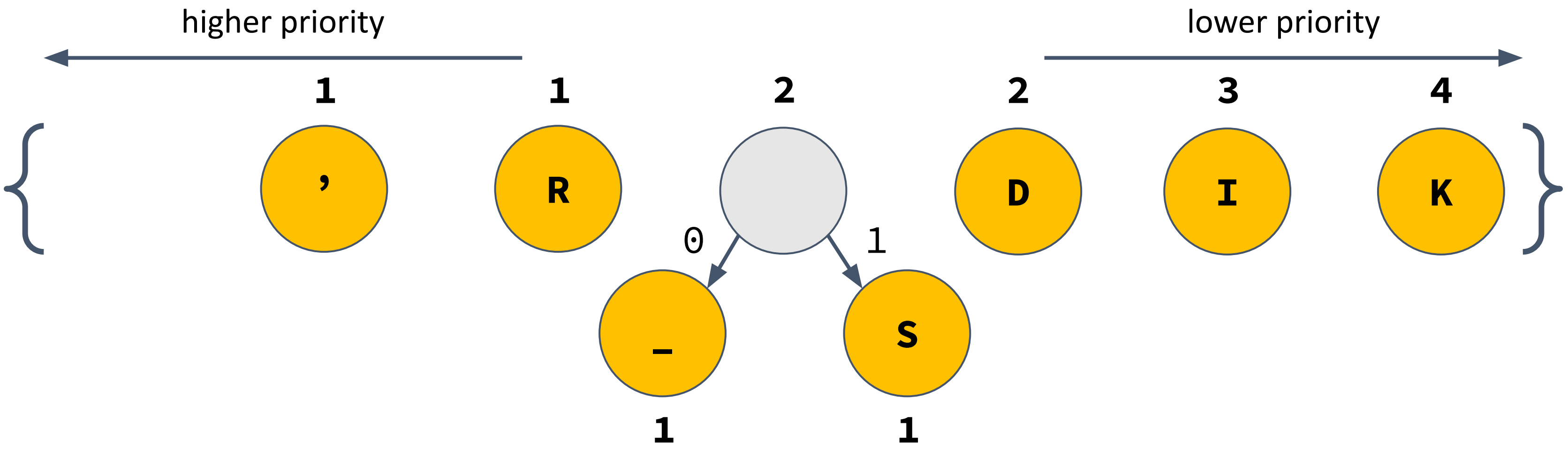

higher priority
lower priority
1
1
1
2
3
4
S
’
R
D
I
K
_
1

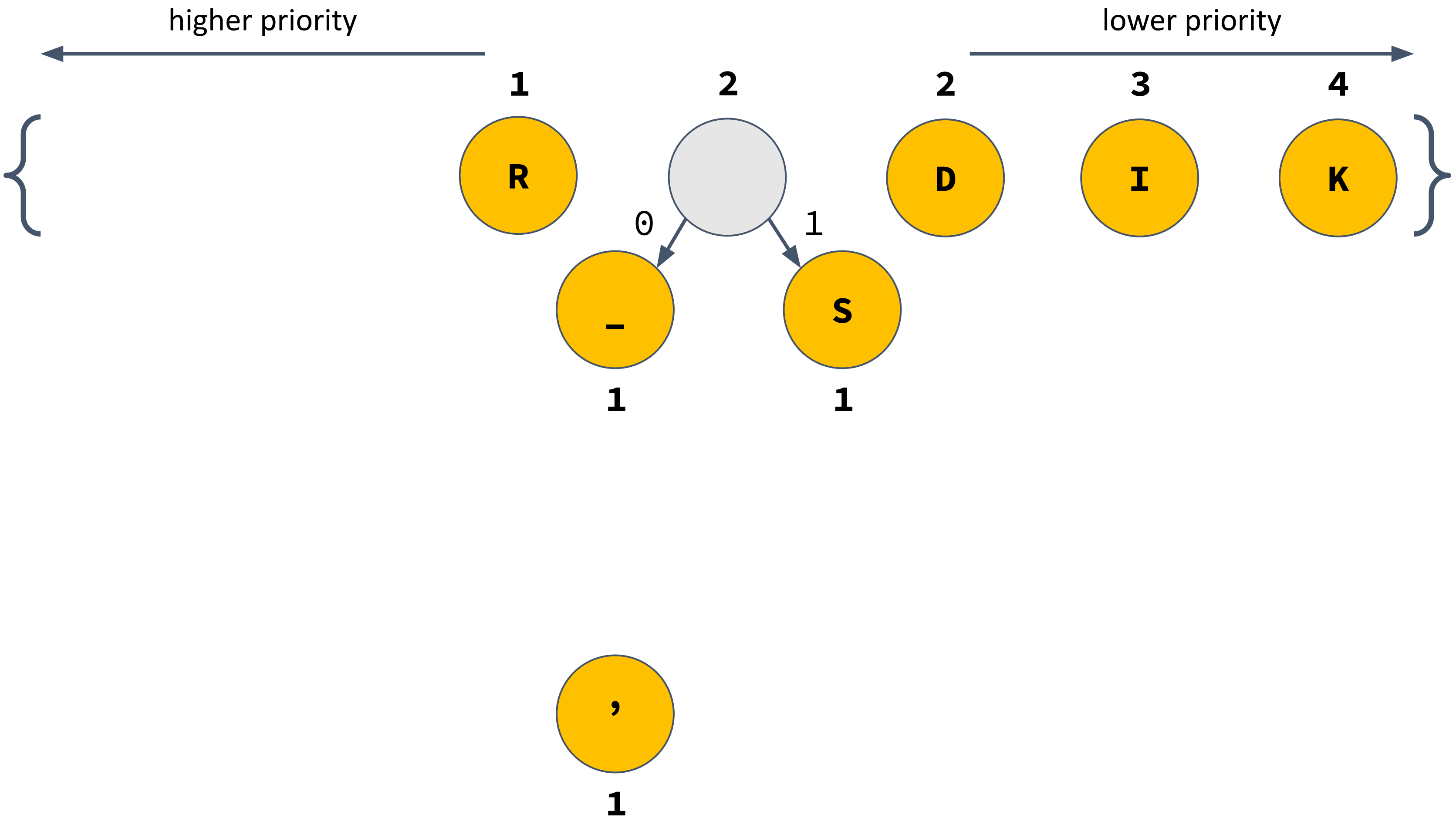

higher priority
lower priority
1
1
2
3
4
{
,
R
D
I
K
}
_
S
1
1

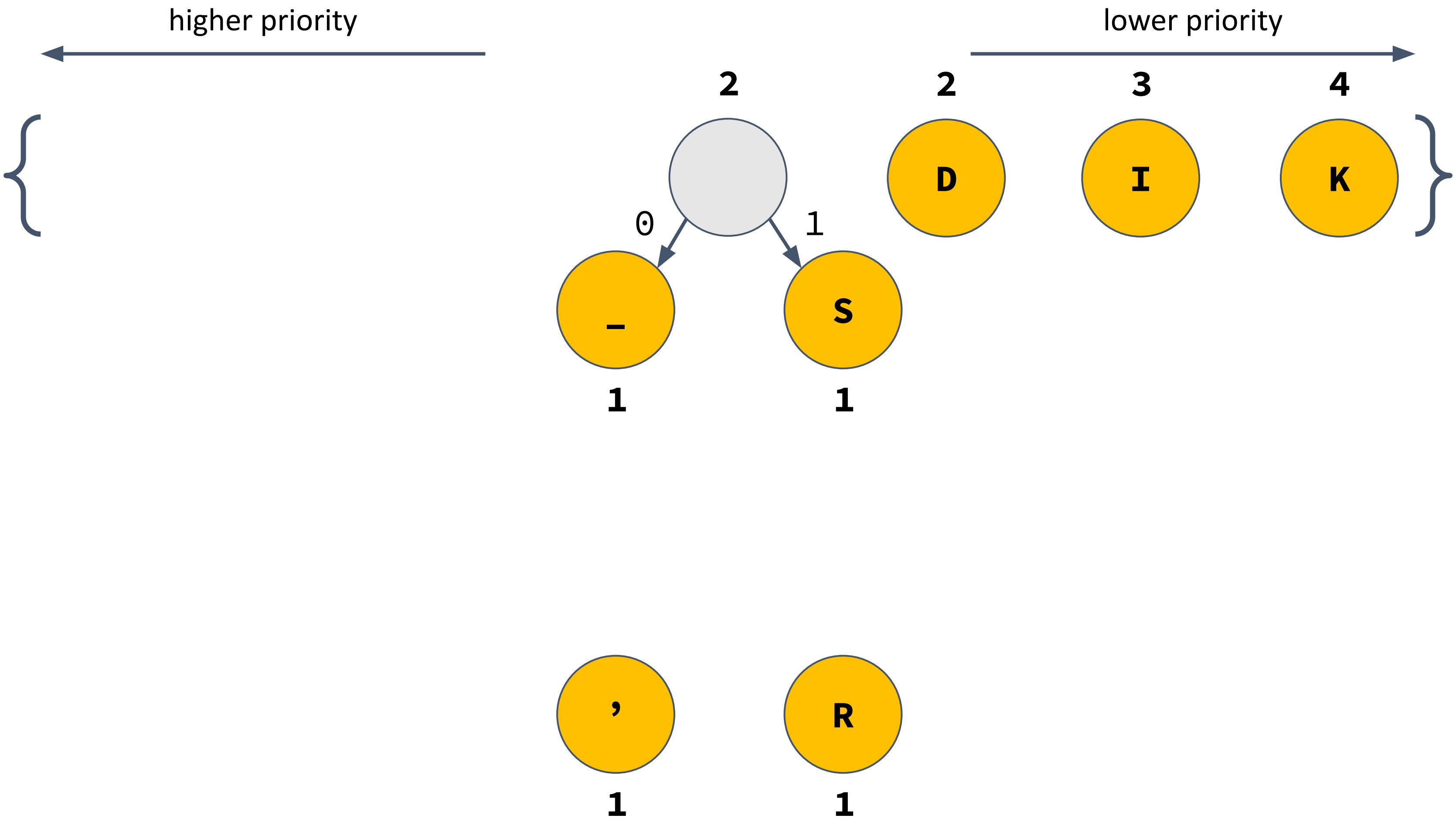

higher priority
lower priority
1
1
2
3
4
'
R
D
I
K
0
1
_
S
1
1

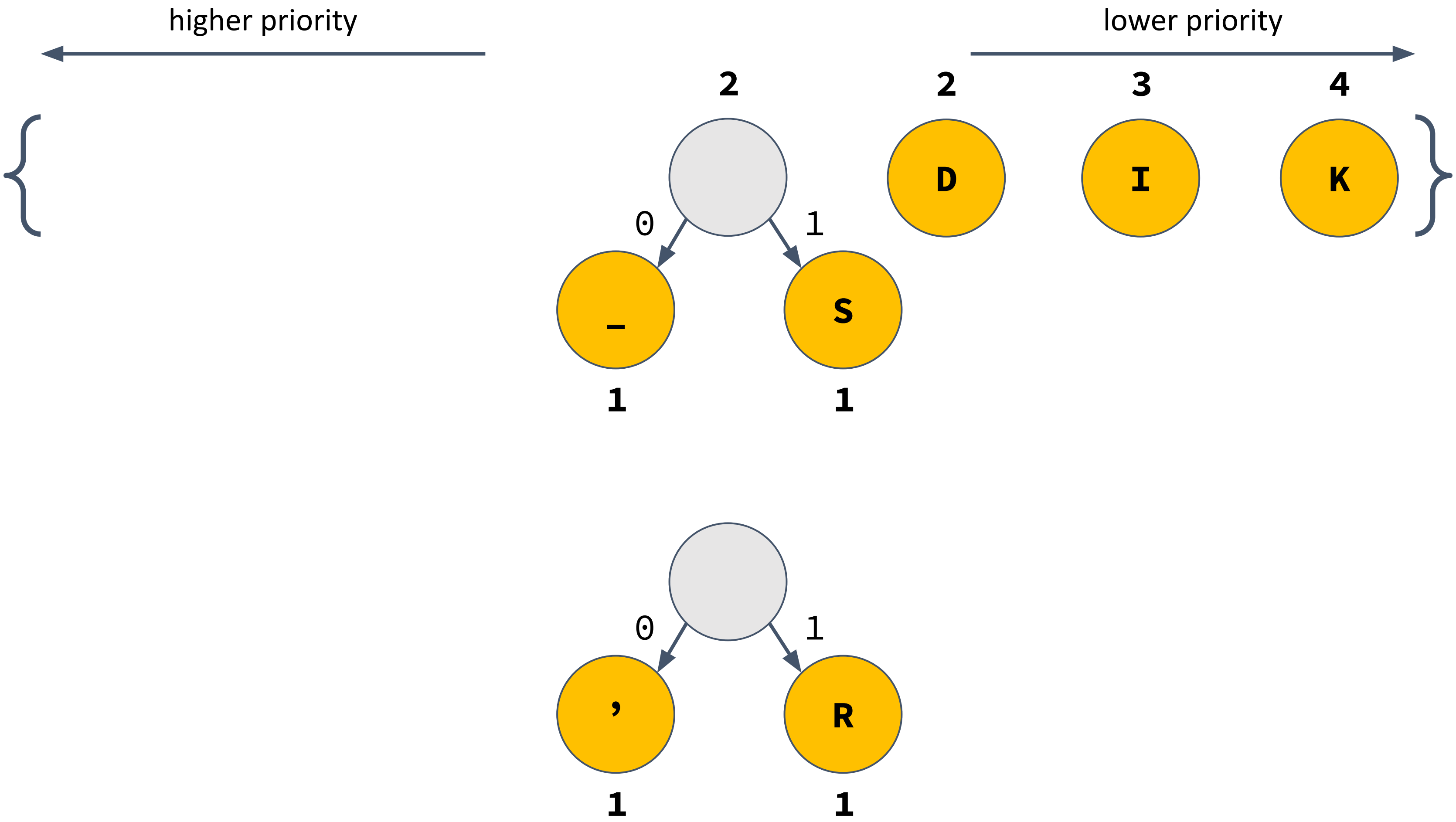

higher priority
lower priority
1
1
2
3
4
,
R
D
I
K
2
0
1
_
S
1
1

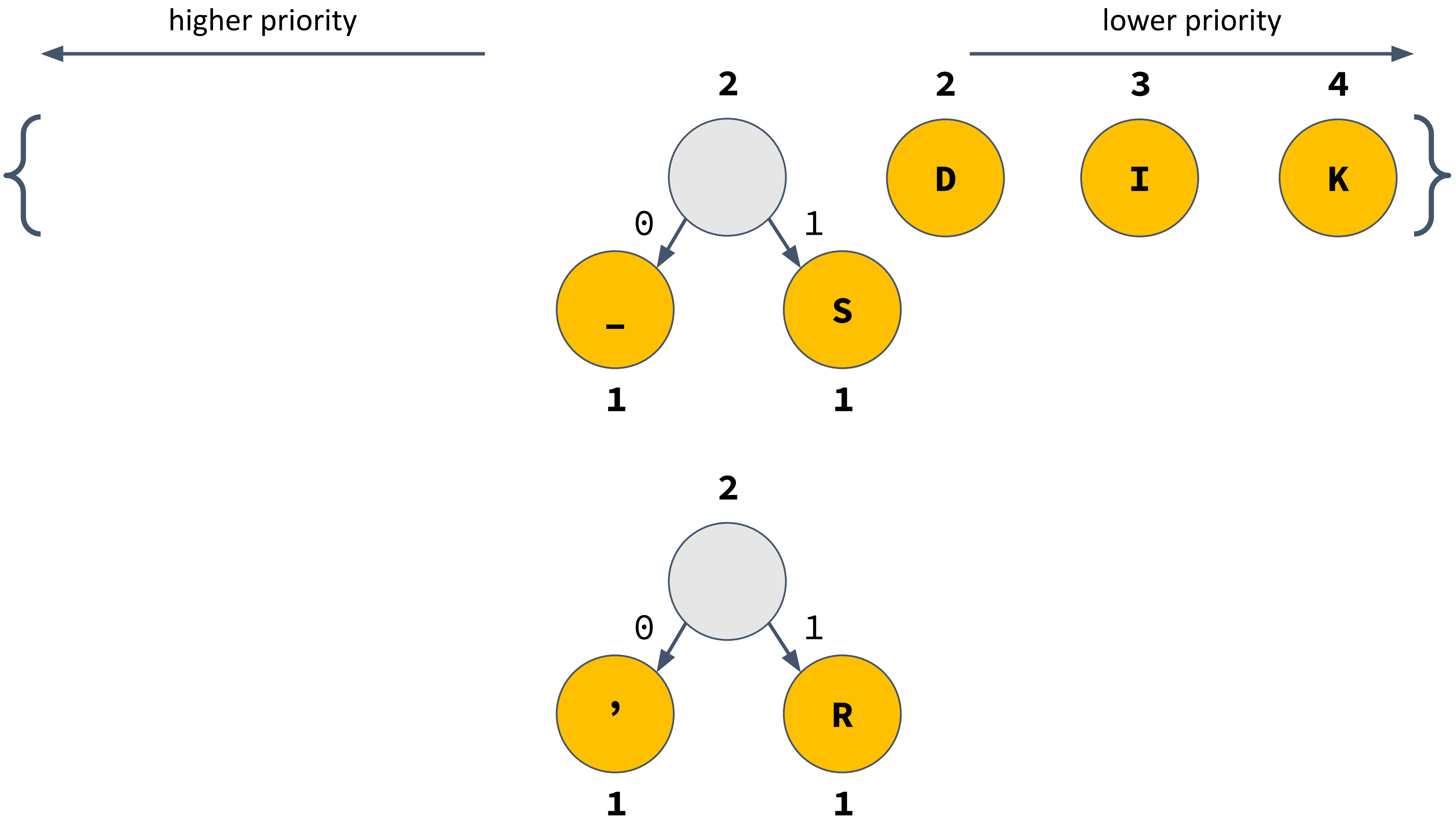

higher priority
lower priority
1
1
2
3
4
'
R
D
I
K
2
0
1
_
S
1
1

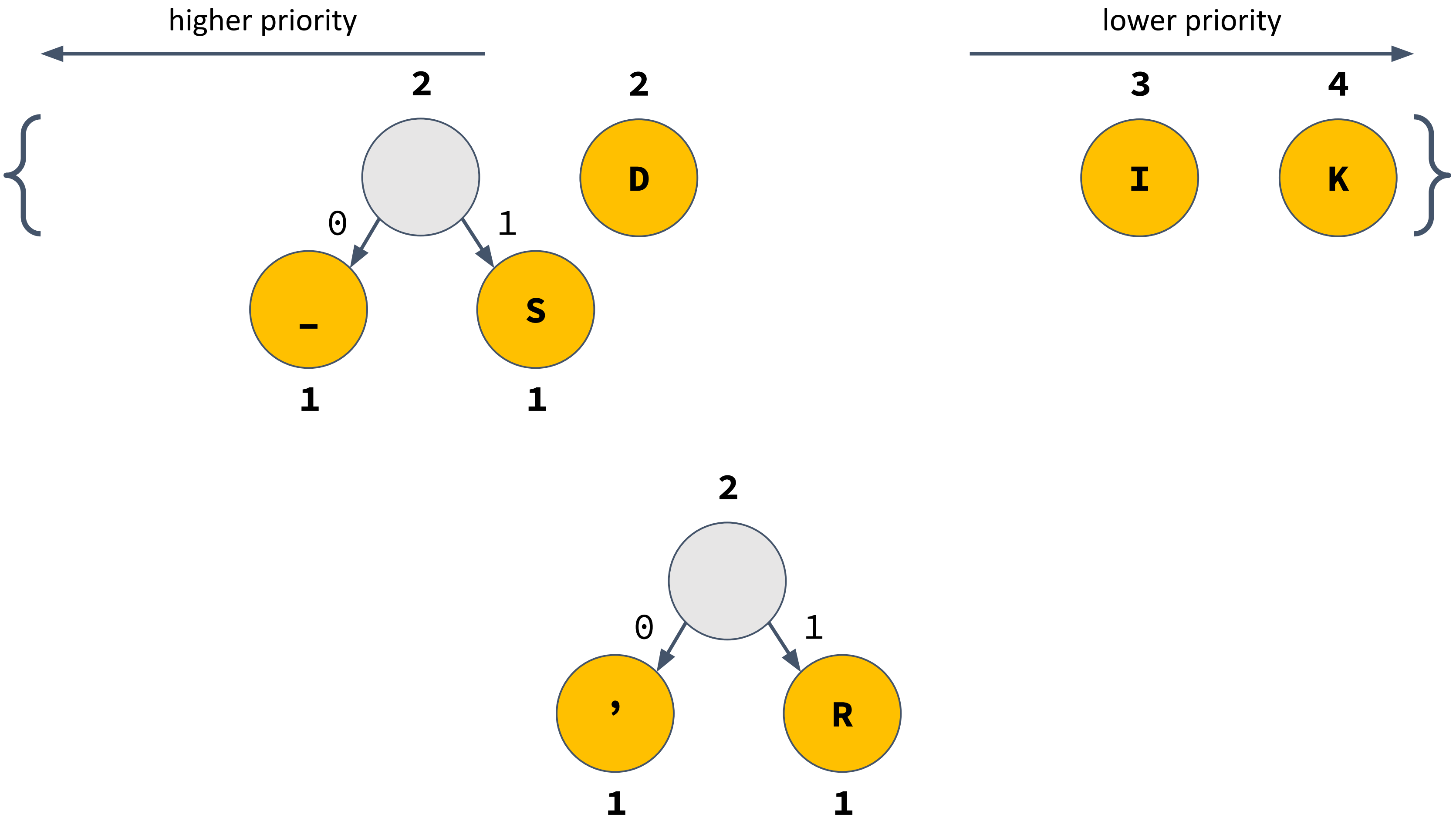

higher priority
lower priority
1
1
2
2
3
4
'
R
D
I
K
0
1
_
S
1
1

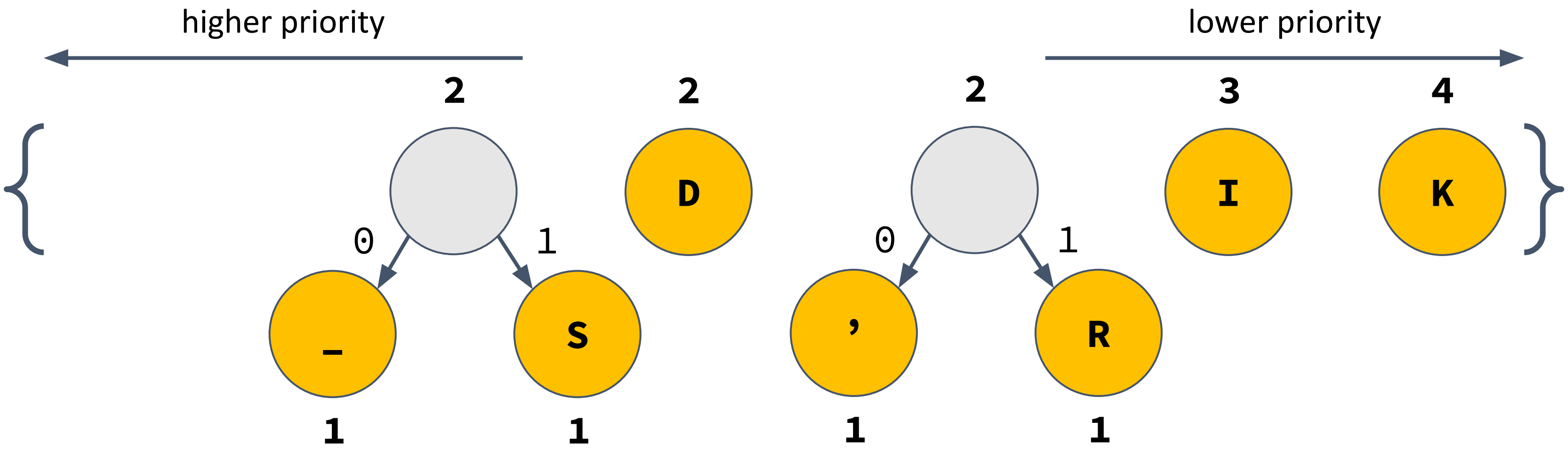

higher priority
lower priority
1
R
2
0
1
_
1
S
1
2
D
3
I
4
K
,
1

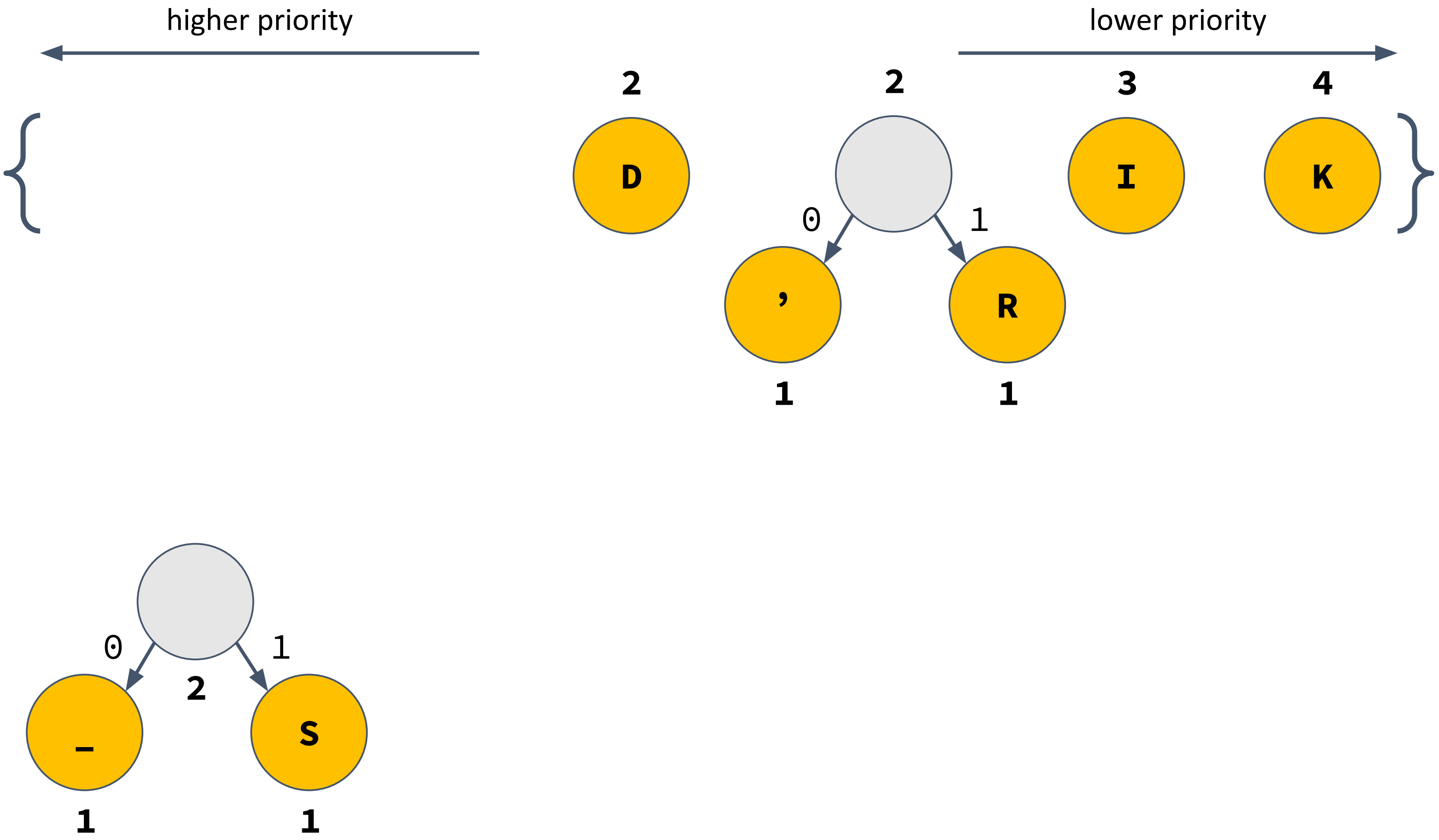

higher priority
lower priority
2
2
3
4
0
1
D
I
K
_
S
1
1
’
R
1
1

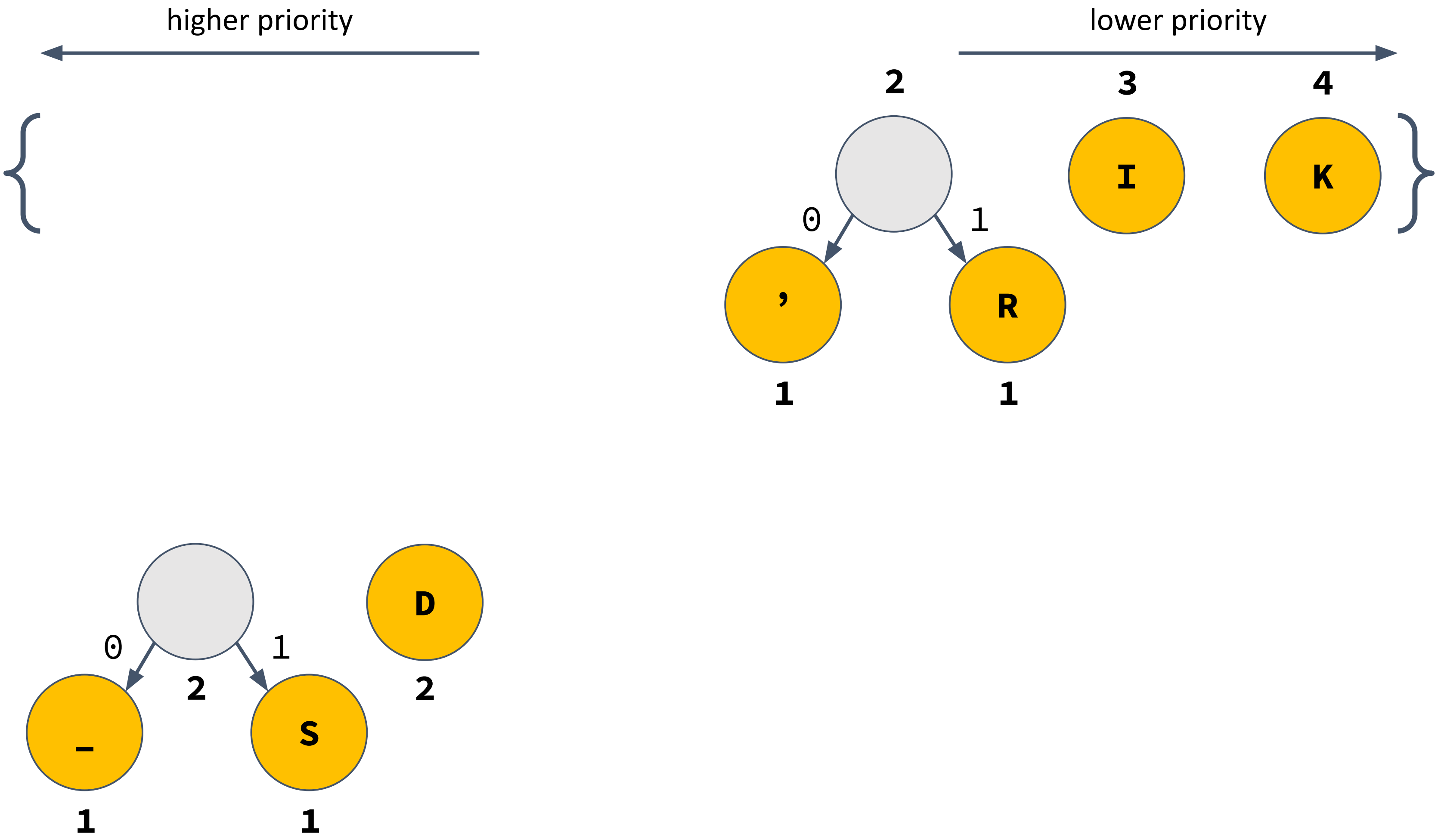

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
0
1
’
R
1
1

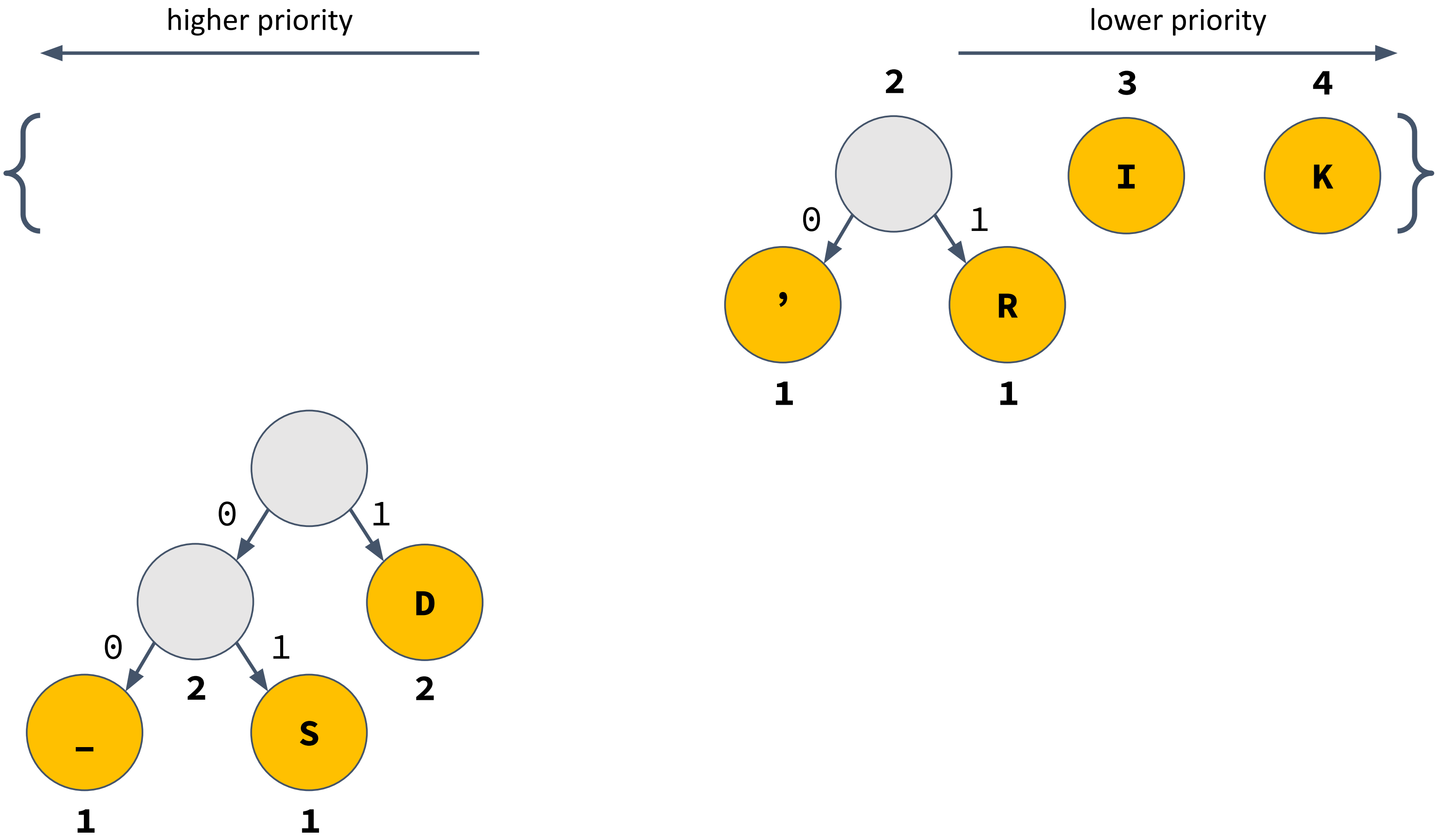

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
2
0
1
’
R
1
1

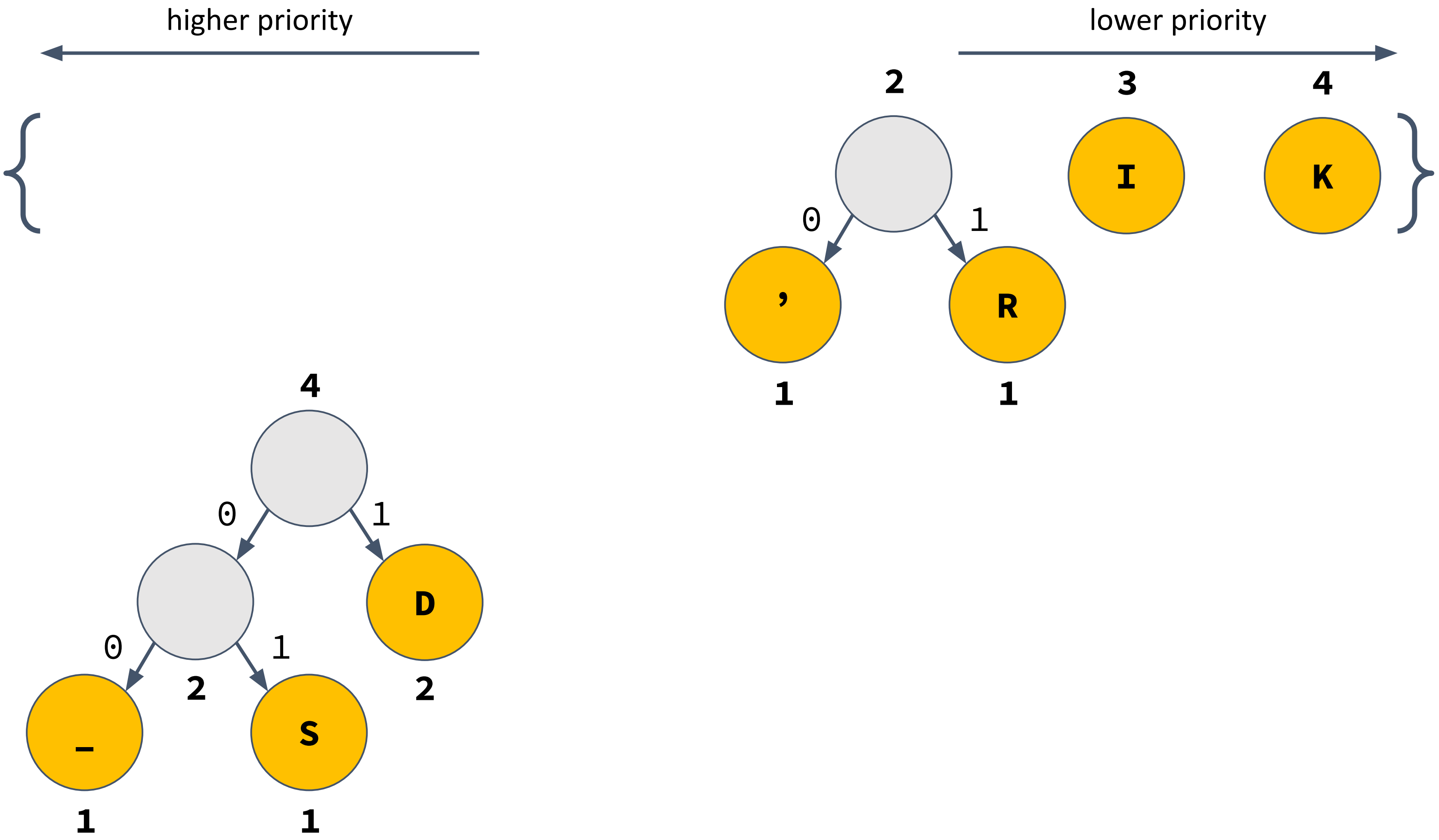

higher priority
lower priority
2
2
3
4
D
I
K
0
1
_
S
1
1
2
0
1
,
R
1
1

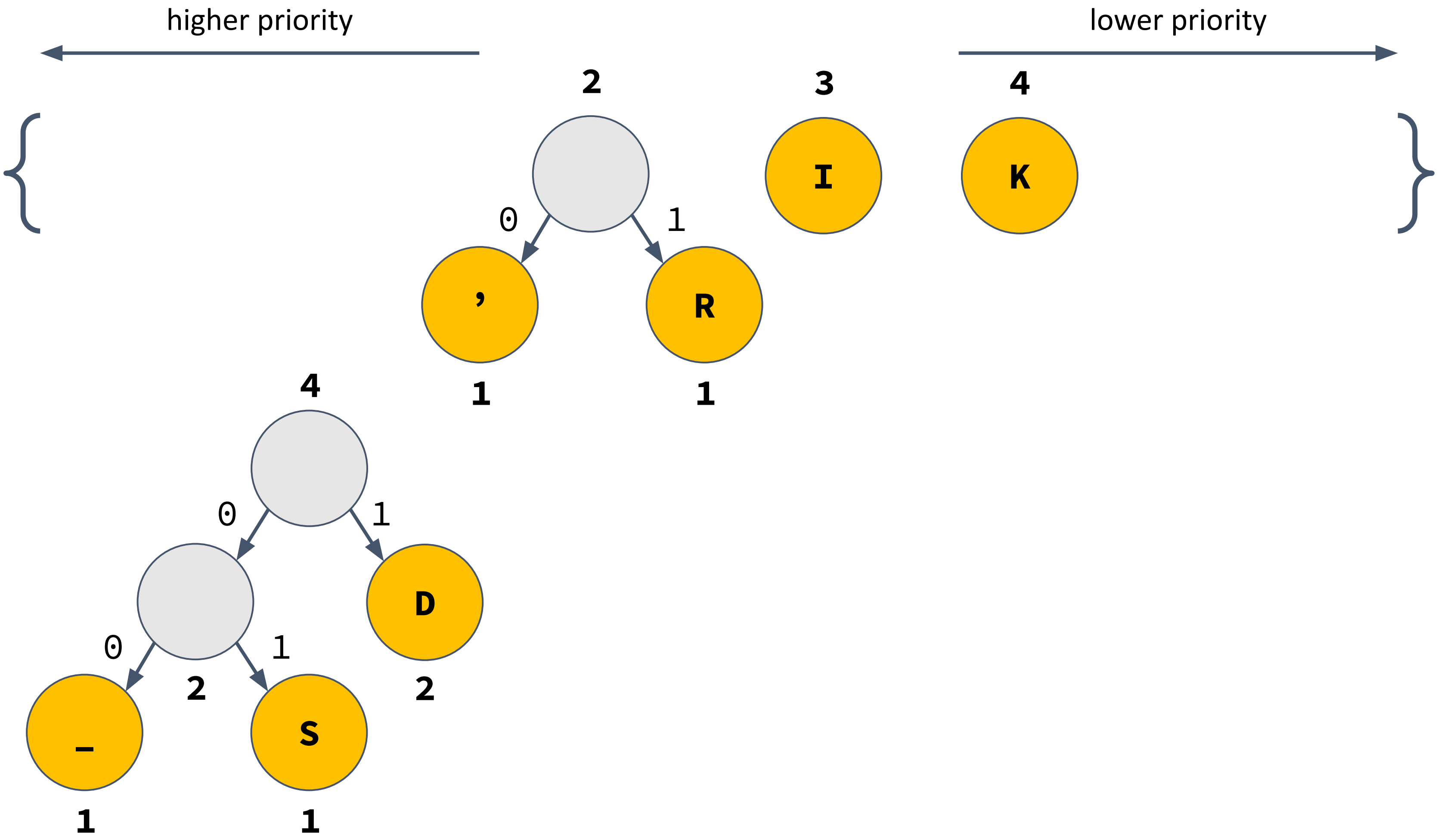

higher priority
lower priority
2
2
2
3
4
0
1
0
1
D
I
K
_
S
'
R
1
1
1
1

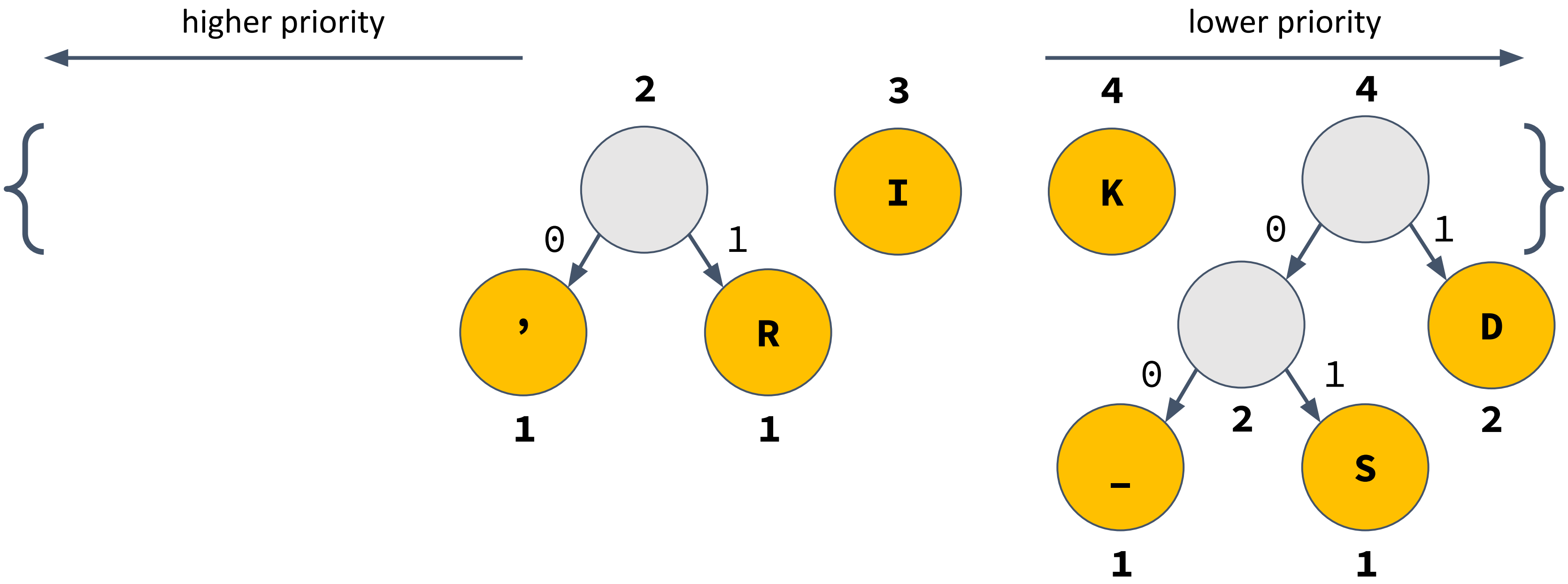

higher priority
lower priority
2
D
2
0
1
,
R
1
1
3
I
4
K
0
1
2
_
S
1
1

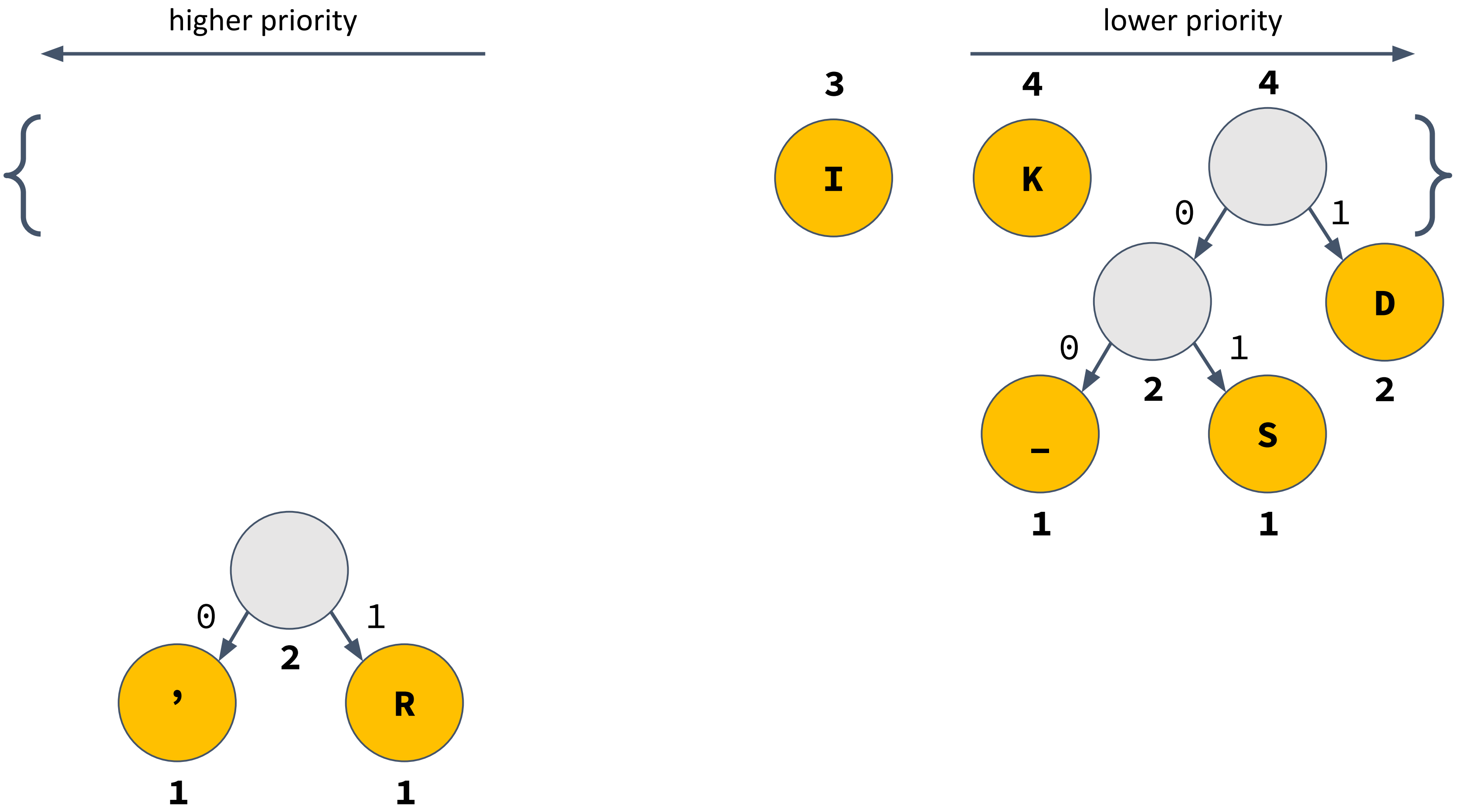

higher priority
lower priority
2
3
4
0
1
I
K
,
R
1
1
D
0
1
2
2
_
S
1
1

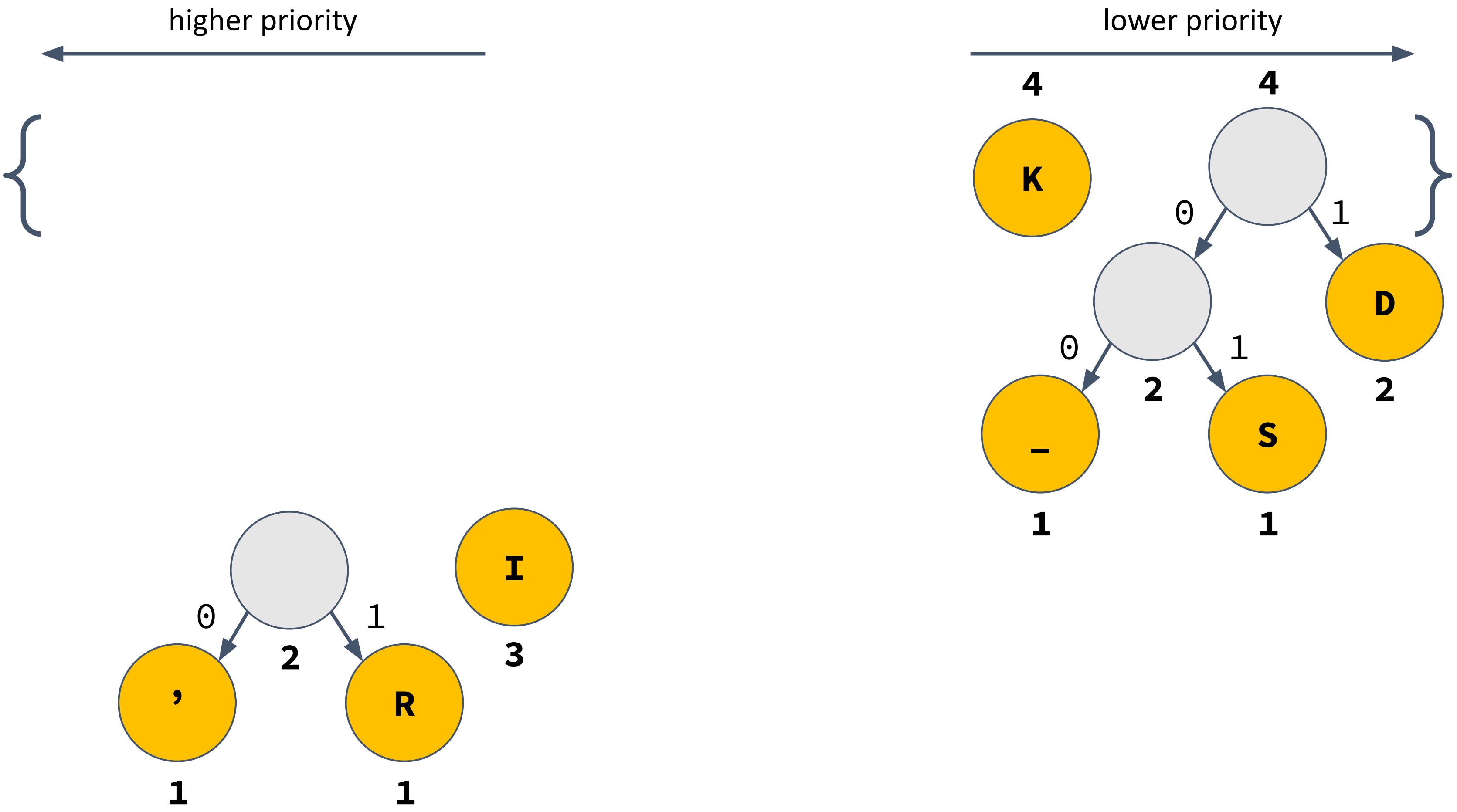

higher priority
lower priority
2
3
4
I
K
0
1
,
R
1
1
0
1
0
1
D
2
2
_
S
1
1

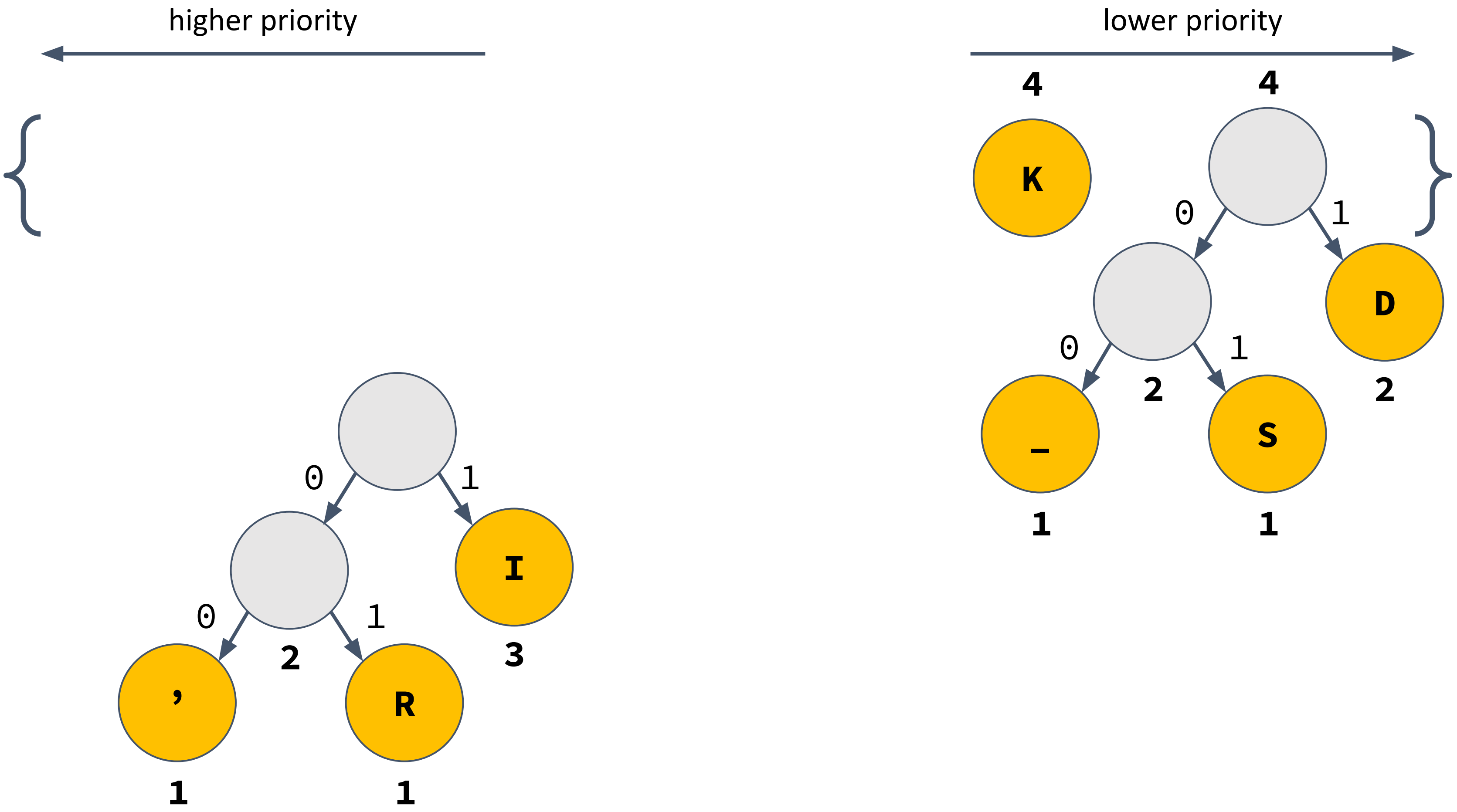

higher priority
lower priority
{
}
2
0
1
,
R
1
1
3
I
4
K
4
0
1
D
2
0
1
_
S
2
1
1

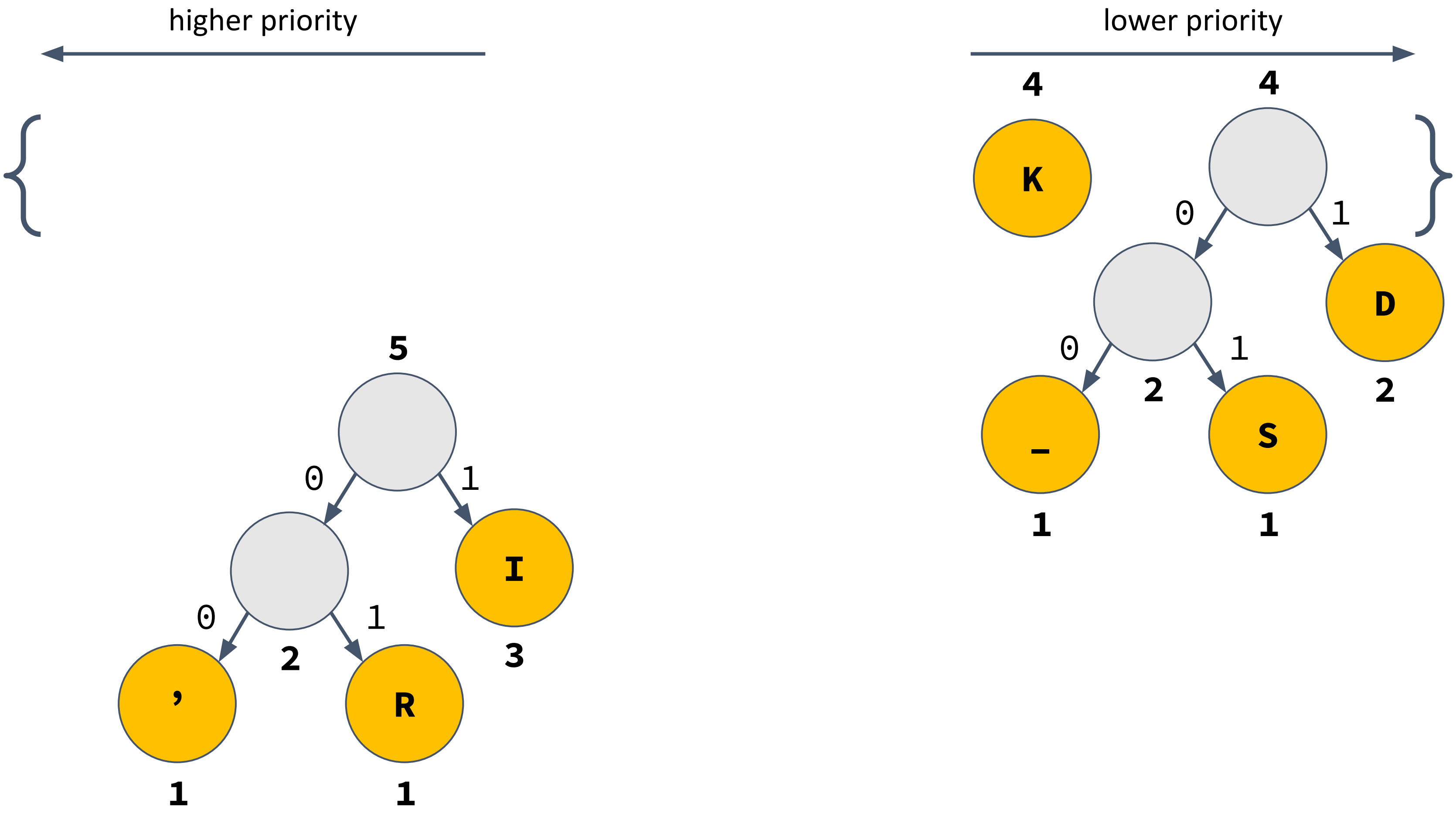

higher priority
lower priority
2
3
4
0
1
I
K
’
R
1
1
4
0
1
D
0
1
2
2
_
S
1
1

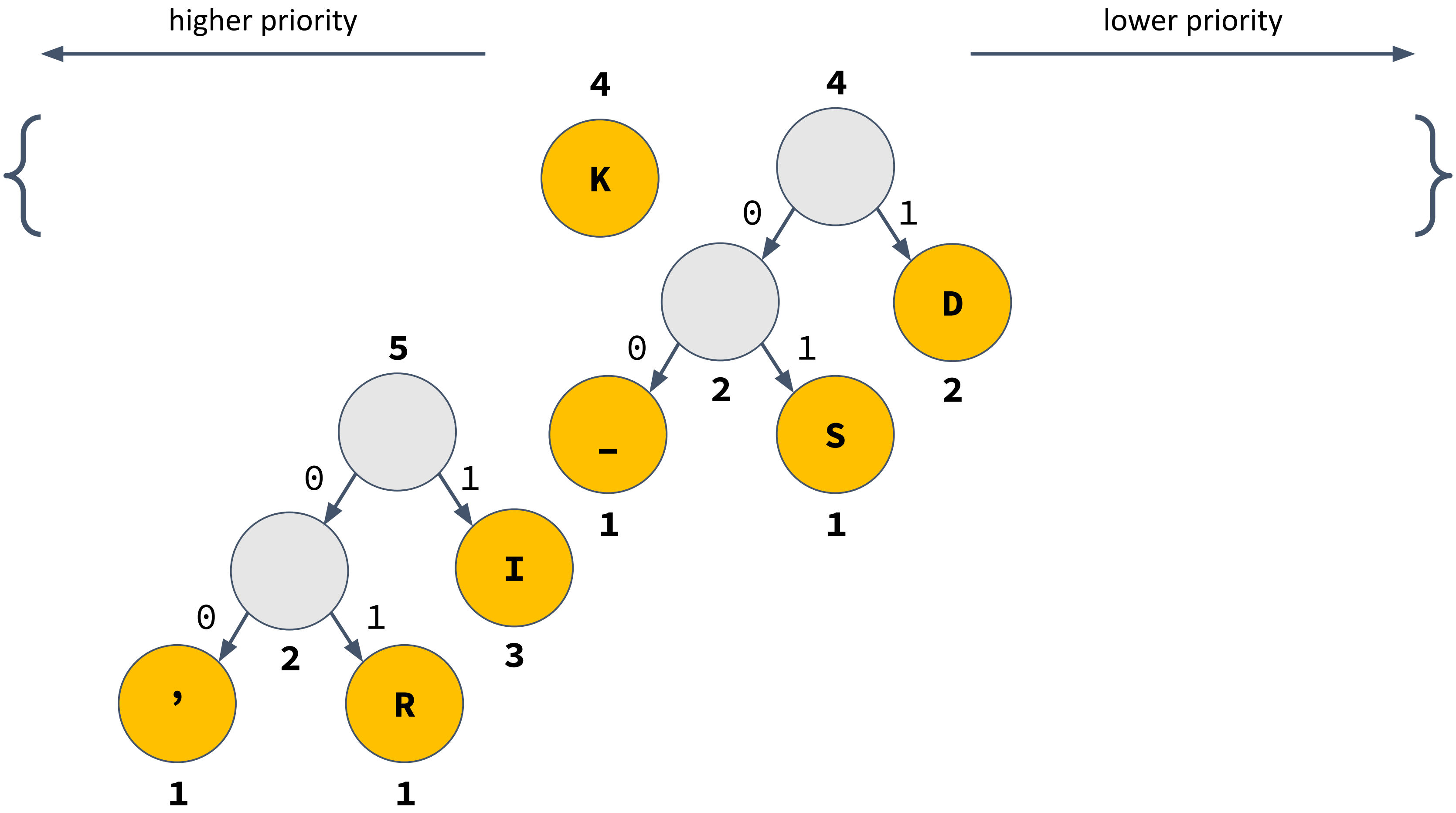

higher priority
lower priority
2
0
1
’
R
1
1
3
I
4
K
4
0
1
2
0
1
_
S
1
1
D
2


higher priority
lower priority
3
I
4
K
4
0
1
0
1
2
D
2
_
S
1
1
0
1
2
,
R
1
1

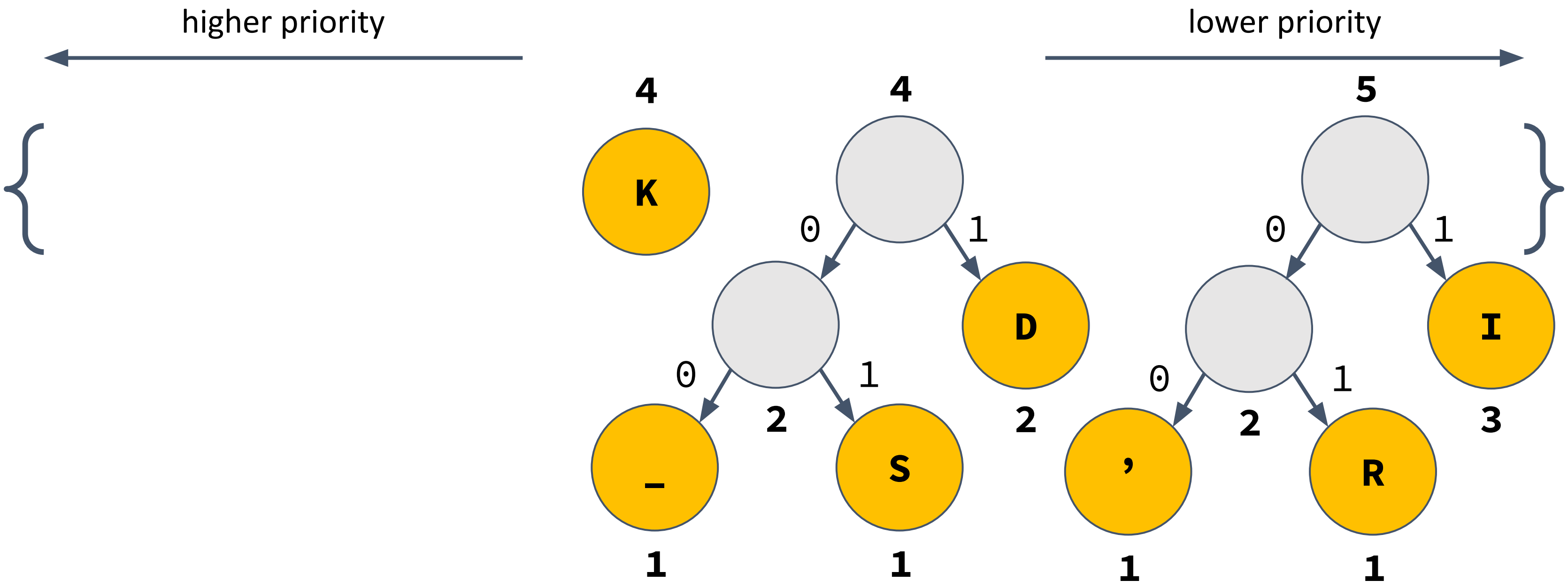

higher priority
lower priority
4
K
4
0
1
D
0
2
1
2
_
S
1
1
0
1
2
I
3
,
R
1
1

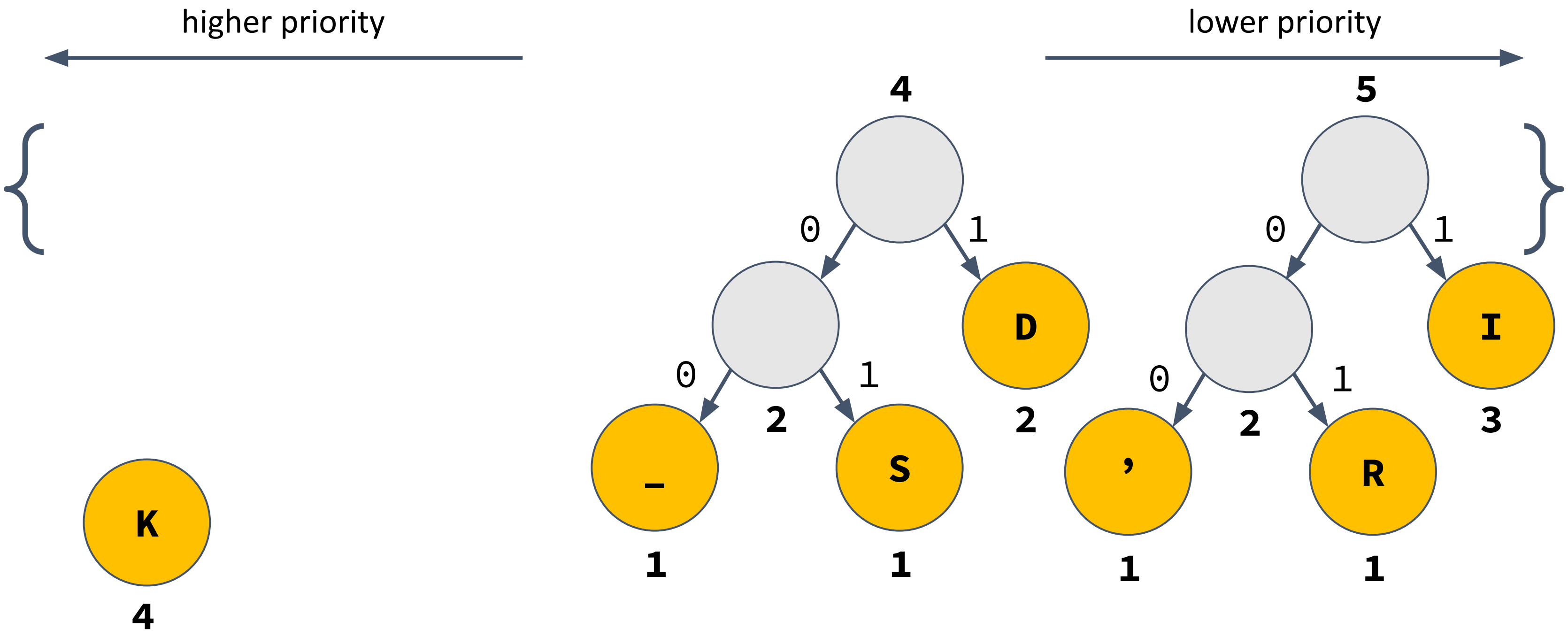

higher priority
lower priority
4
K
4
0
1
D
2
0
1
2
_
S
1
1
0
1
I
3
0
1
2
,
R
1
1

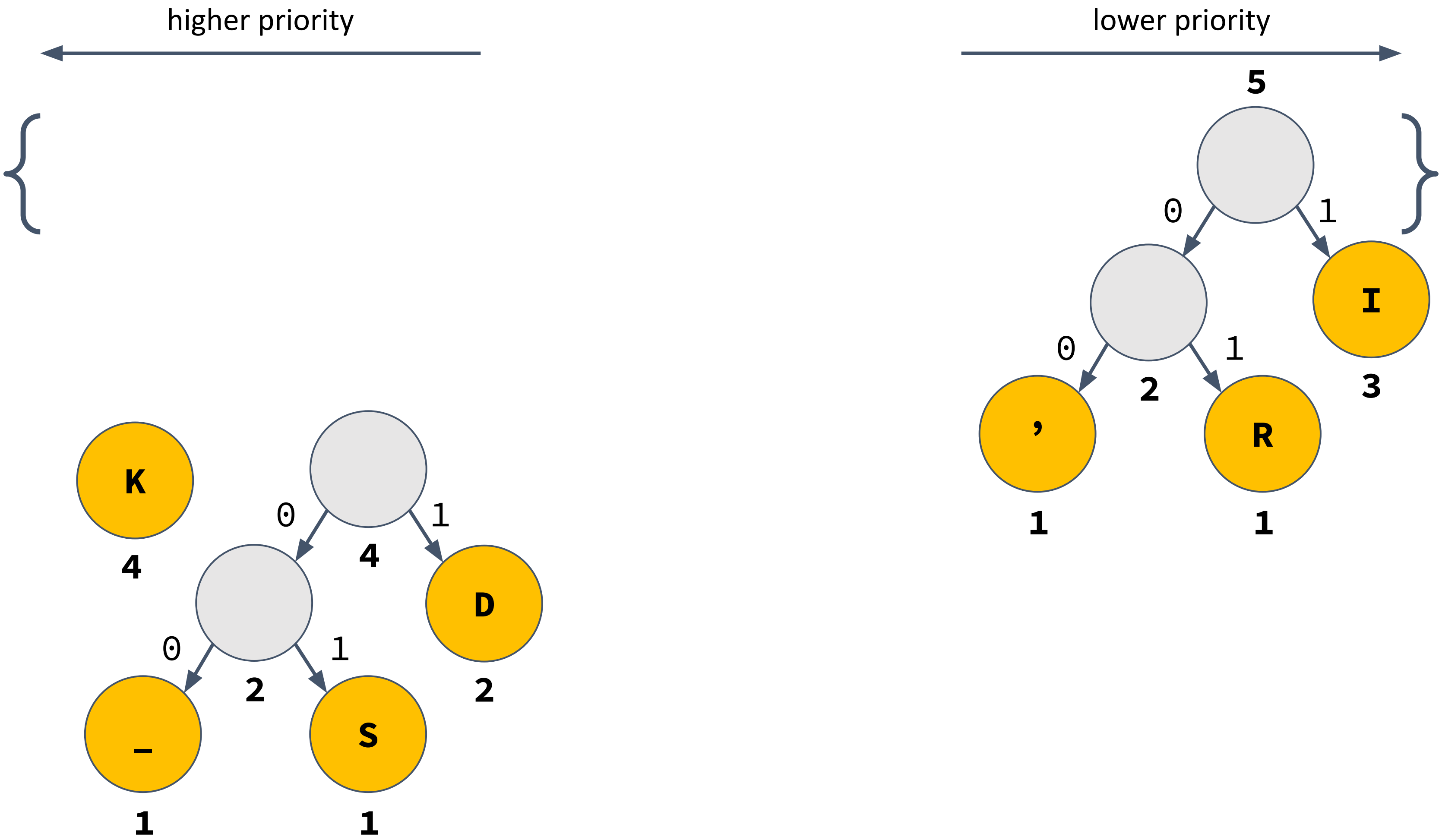

higher priority
lower priority
4
K
4
0
1
D
0
1
2
2
_
S
1
1
5
0
1
I
0
1
2
3
’
R
1
1

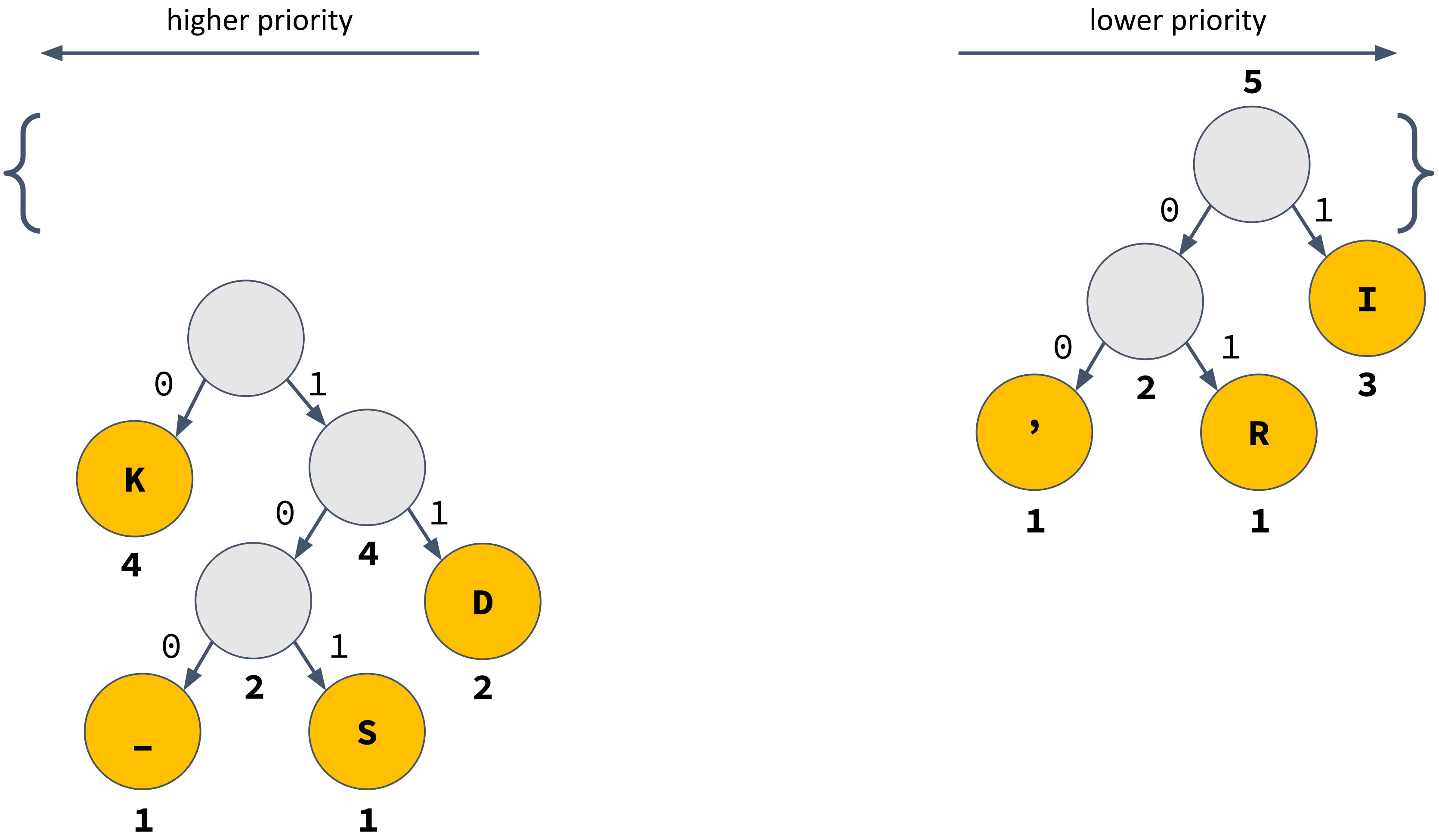


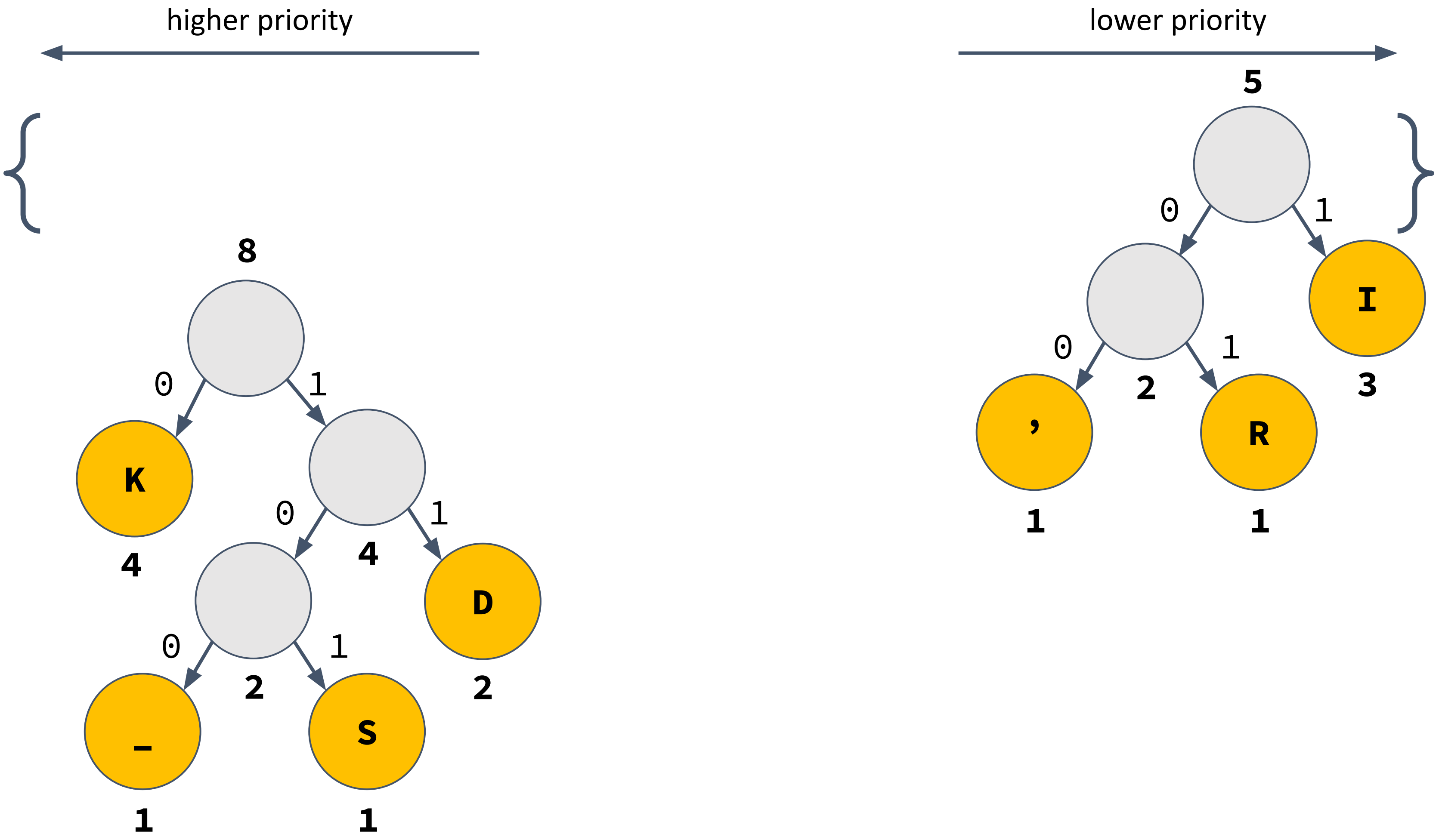
higher priority
lower priority
4
K
4
0
1
D
2
0
1
_
2
S
1
1
5
0
1
I
3
0
1
,
2
R
1
1

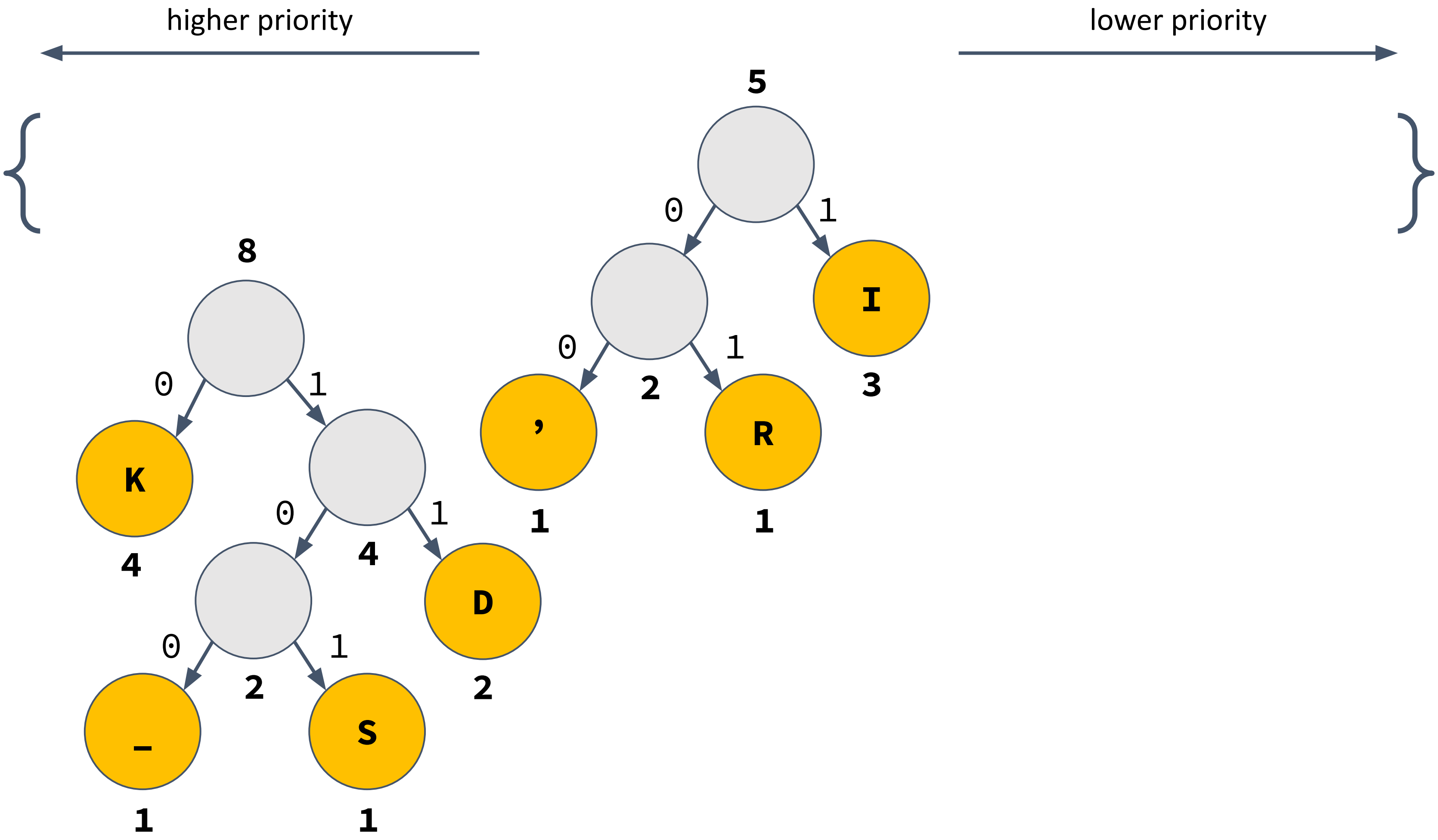

higher priority
lower priority
4
K
4
0
1
2
D
2
0
1
_
1
S
1
5
0
1
2
I
3
0
1
’
1
R
1


higher priority
lower priority
K
4
4
0
1
0
1
_
S
D
2
1
1
2
5
0
1
0
1
,
R
I
2
1
1
3


higher priority
lower priority
K
4
0
1
4
D
2
0
1
2
_
1
S
1
5
0
1
I
3
0
1
2
’
1
R
1

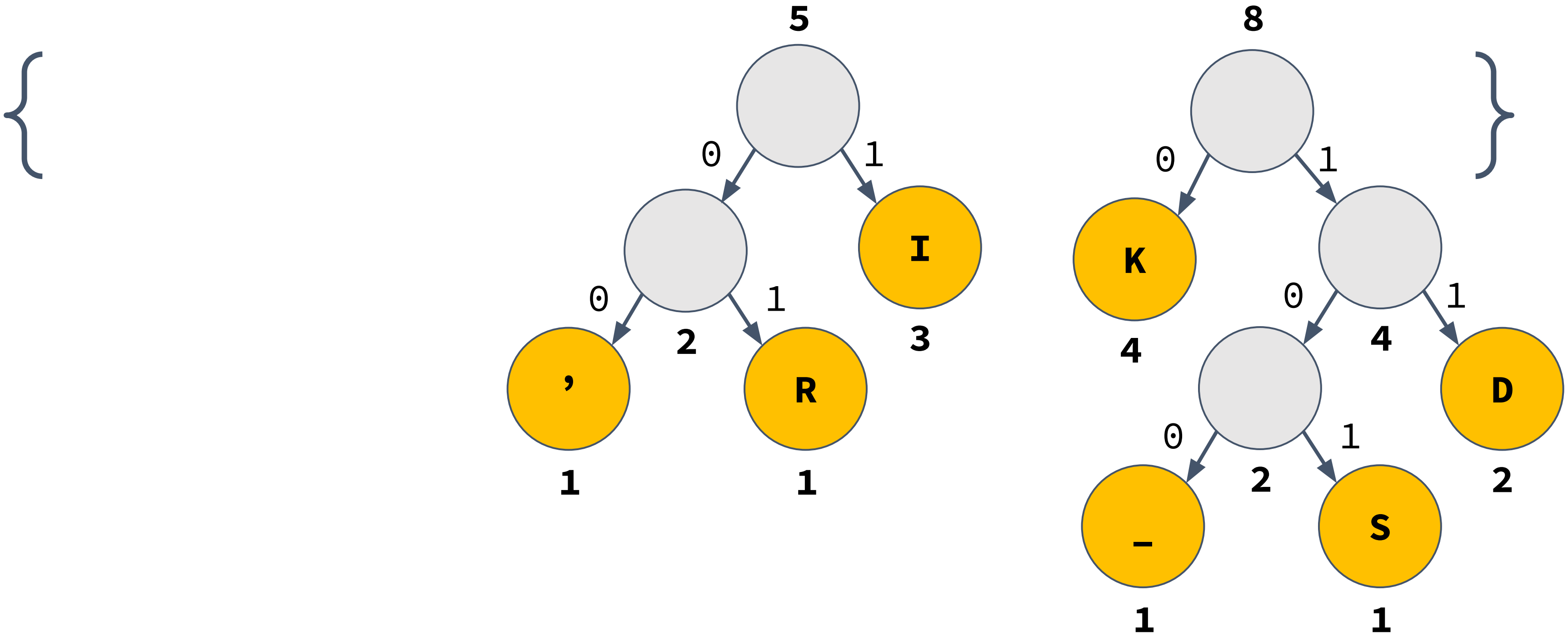

higher priority
lower priority
0
1
K
4
0
1
4
D
0
1
2
2
_
S
1
1
5
0
1
0
1
I
2
3
,
R
1
1

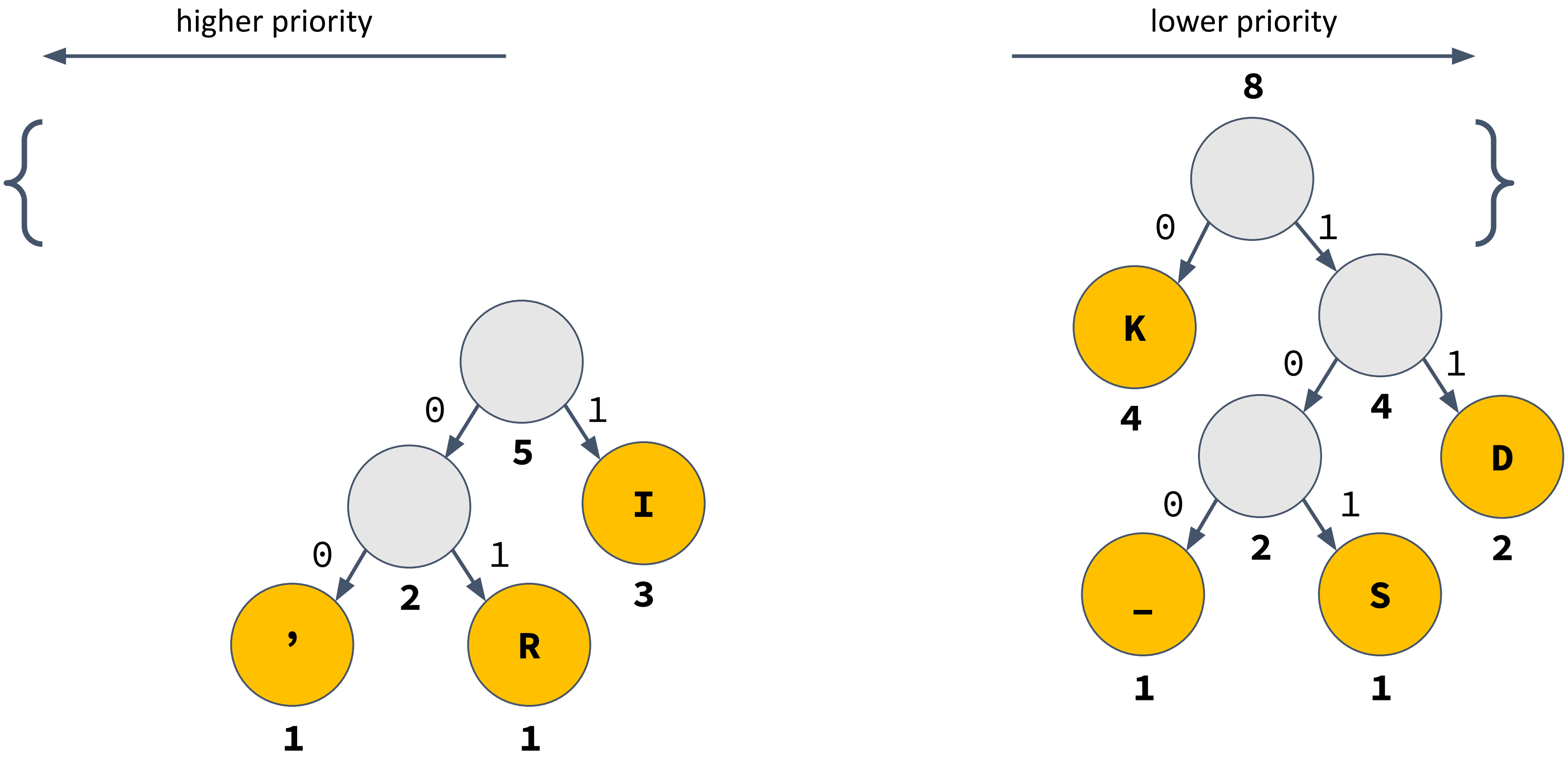

higher priority
lower priority
8
0
1
K
4
0
1
4
D
2
0
1
2
_
S
1
1
5
0
1
I
3
0
1
2
,
R
1
1


higher priority
lower priority
8
0
1
K
4
0
1
4
0
1
_
2
S
D
2
1
1
5
0
1
0
1
,
2
R
I
3
1
1

higher priority

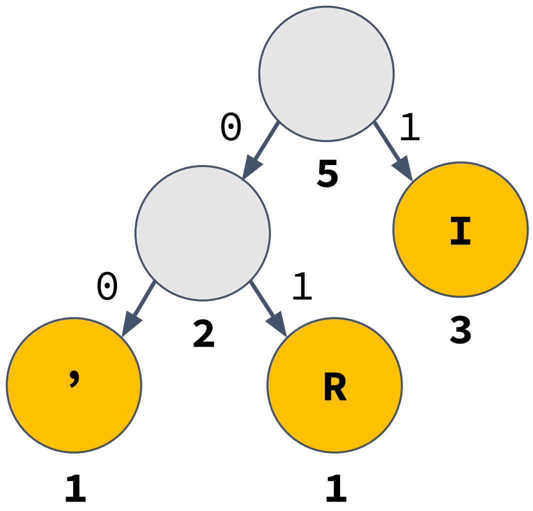
lower priority

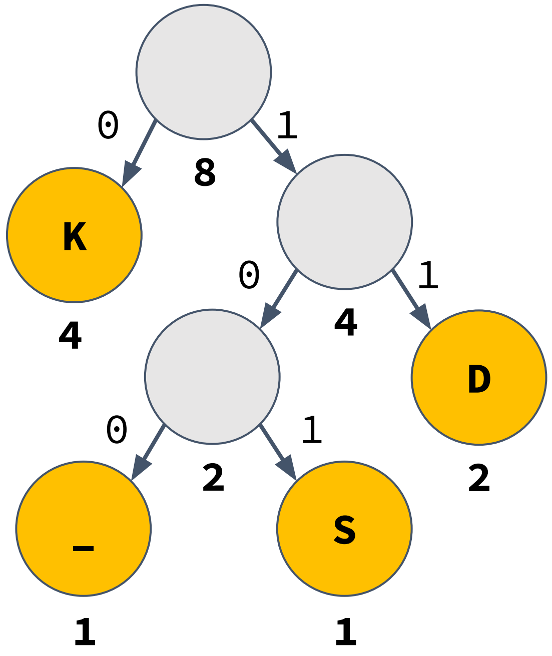



higher priority
lower priority
0
1
5
I
3
0
1
2
'
1
R
1
8
0
1
K
4
0
1
4
D
2
0
1
2
_
1
S
1

higher priority ← → lower priority

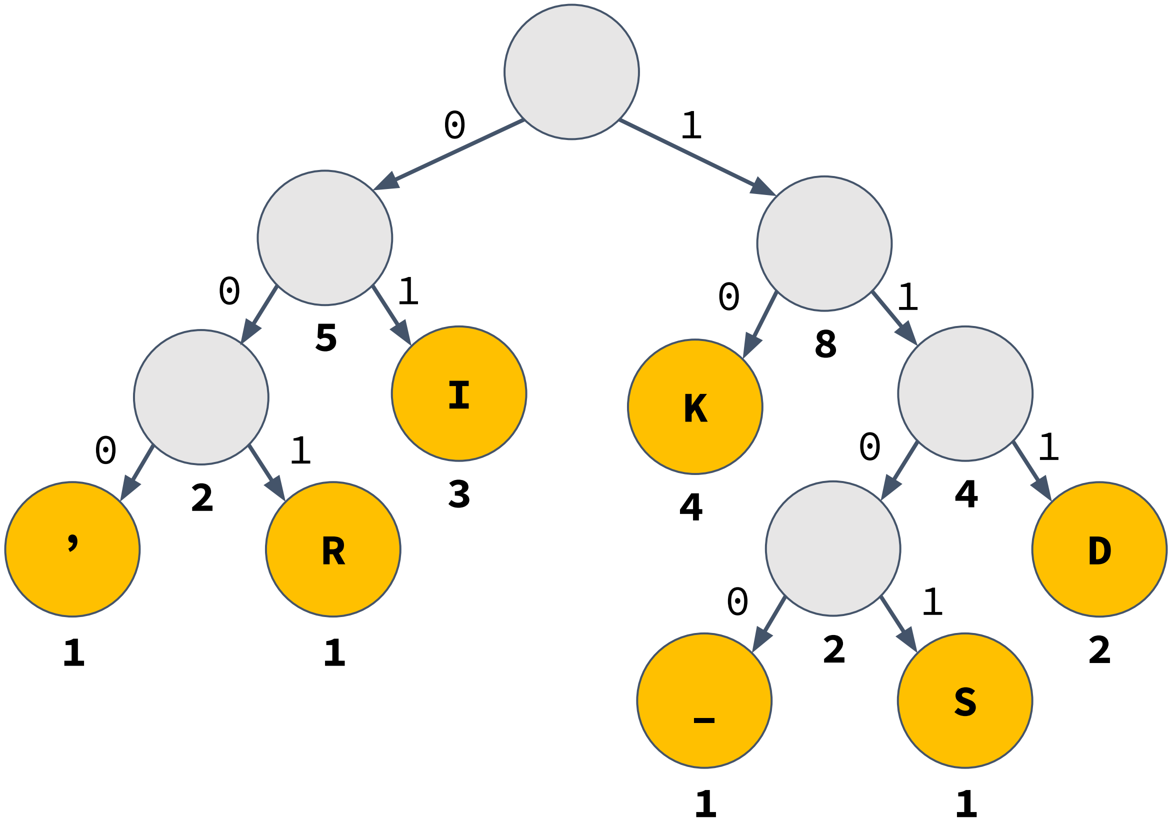
{ }



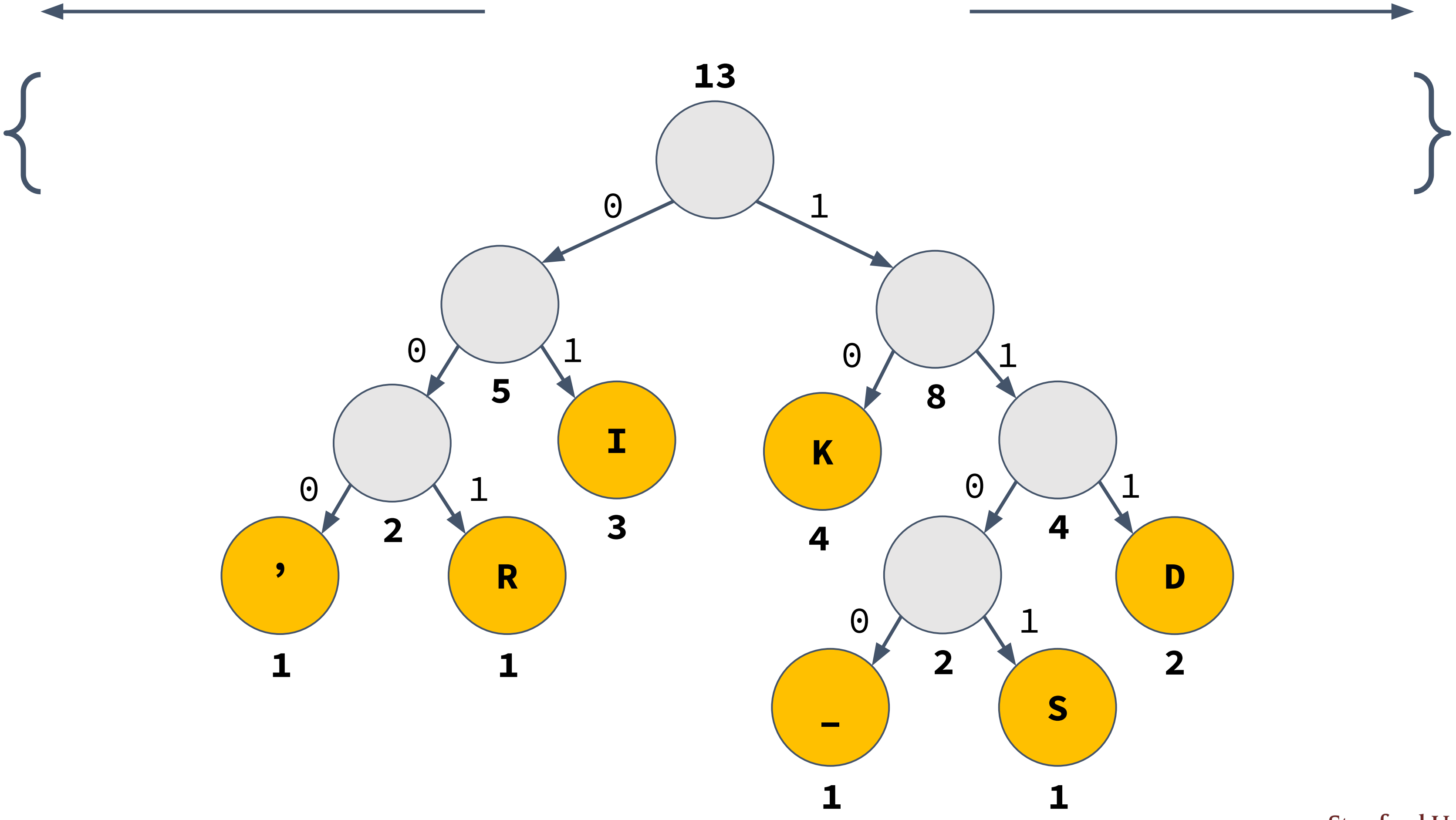
higher priority ← → lower priority

{ }



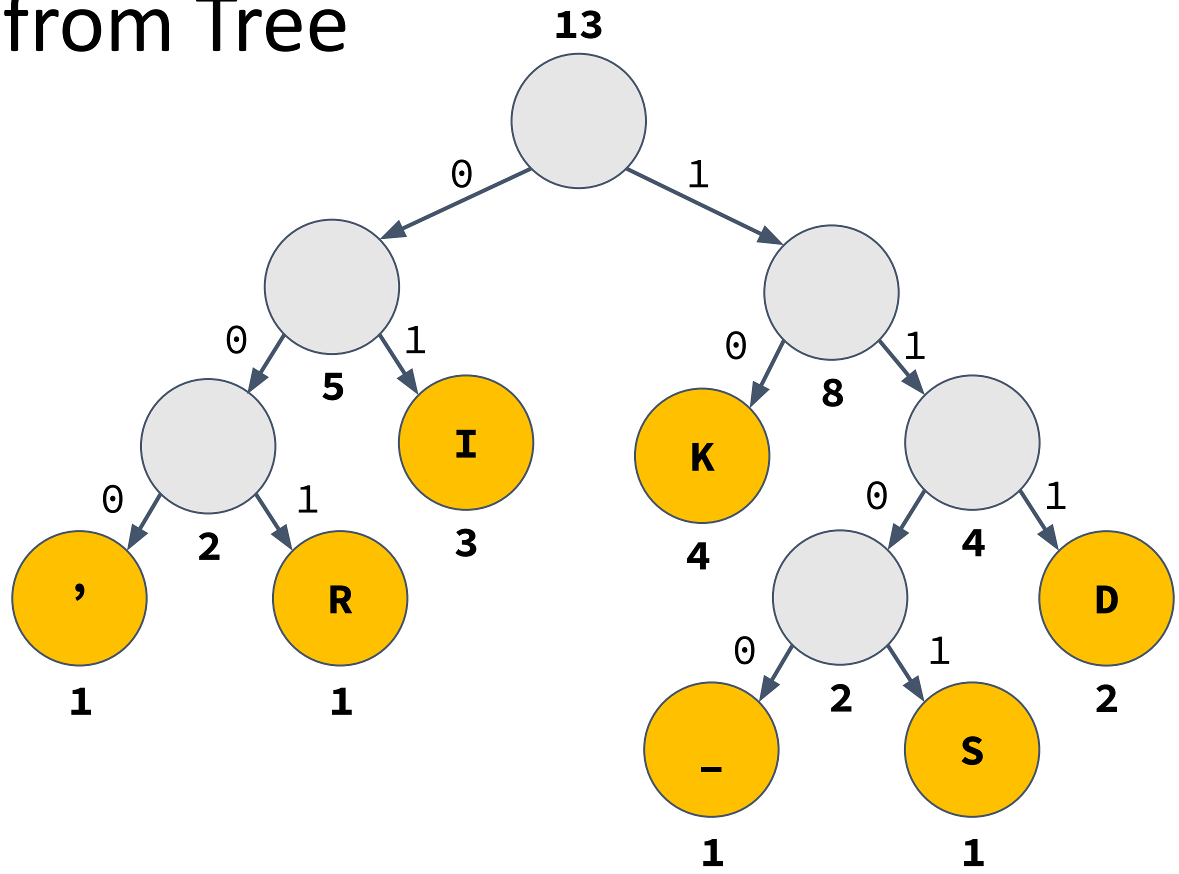
higher priority

lower priority



# Generate Table from Tree

| *character* | *code* |
|---|---|
| K | 10 |
| I | 01 |
| D | 111 |
| R | 001 |
| , | 000 |
| S | 1101 |
| _ | 1100 |



# Transmitting the Tree

- In order to decompress the text, we have to remember what encoding scheme we used
- Prefix the compressed data with a header containing information to rebuild the tree

| | |
|---|---|
| **Encoded Tree** | **1001001100001101110011101101110110…** |

# ADT Showdown

Let's compare the performance of different abstract data types

# ADT Performance

When we analyze an ADT, we care about how quickly we can:

- Look up elements (`contains`)
- Add elements (`insert`/`add`)
- Remove elements (`remove`)

# Unsorted Array

| Operation | Runtime |
| --- | --- |
| Contains | |
| Insert | |
| Remove | |

| | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 14 | 3 | 16 | 7 | 9 | 2 | 10 | 5 |

# Unsorted Array

| **Operation** | **Runtime** |
| --- | --- |
| Contains | O(n) |
| Insert | O(n) |
| Remove | O(n) |

| | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 14 | 3 | 16 | 7 | 9 | 2 | 10 | 5 |

# Sorted Array

| **Operation** | **Runtime** |
|---|---|
| Contains | |
| Insert | |
| Remove | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 9 | 10 | 14 | 16 |

# Sorted Array

*Binary search to the rescue!*

| Operation | Runtime |
|---|---|
| Contains | O(log n) |
| Insert | |
| Remove | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 9 | 10 | 14 | 16 |

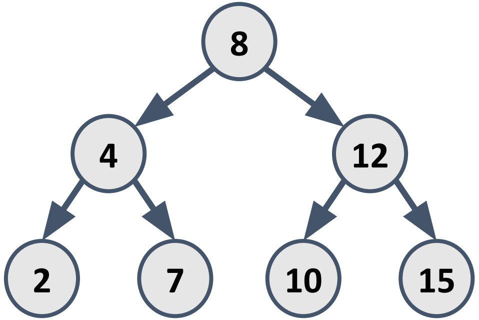
# Sorted Array

| Operation | Runtime |
| --- | --- |
| Contains | O(log n) |
| Insert | O(n) |
| Remove | O(n) |

| | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 6 | 7 | 9 | 10 | 14 | 16 |

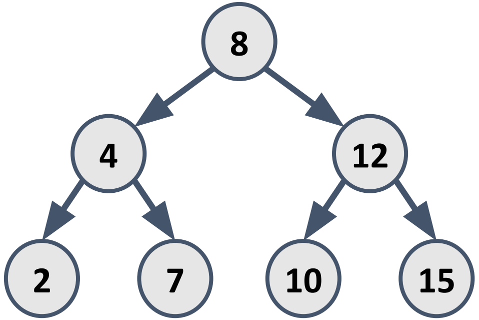
# Binary Search Tree (and Set)

| Operation | Runtime |
| --- | --- |
| Contains | |
| Insert | |
| Remove | |



# Binary Search Tree (and Set)

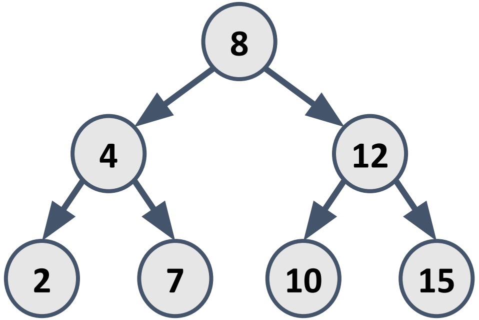
| Operation | Runtime |
| --- | --- |
| Contains | O(log n) |
| Insert | O(log n) |
| Remove | O(log n) |



# Binary Search Tree (and Set)

| Operation | Runtime |
| --- | --- |
| Contains | O(log n) |
| Insert | O(log n) |
| Remove | O(log n) |

*As always, we ask:*
*Can we do better?*



# Idea 1: Count Array

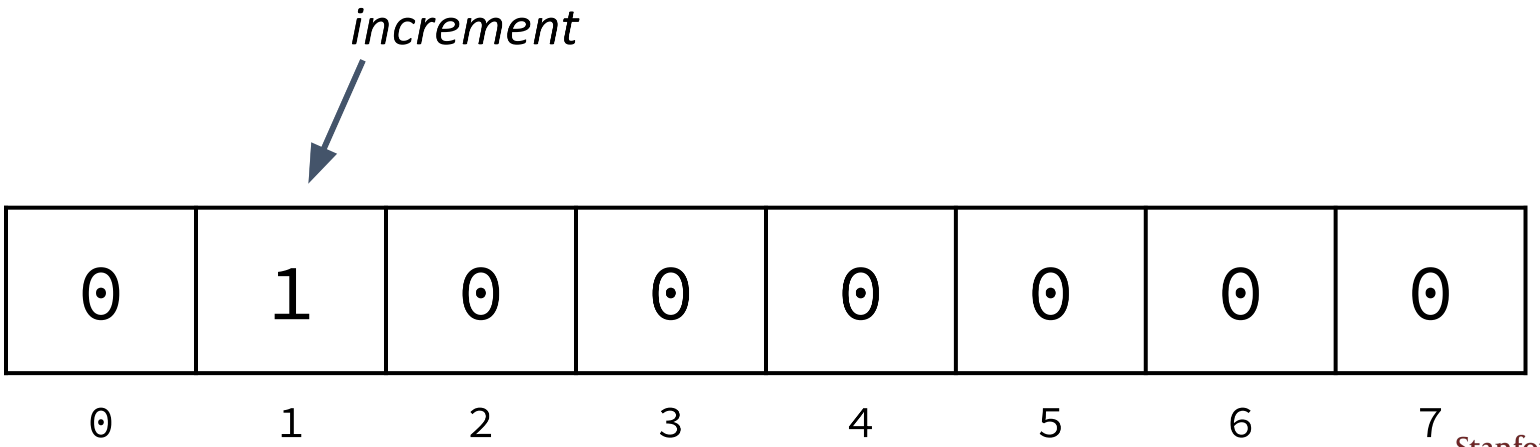
Each number gets its own index in the array, which stores a count

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Add 1*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Add 1*
*Add 6*

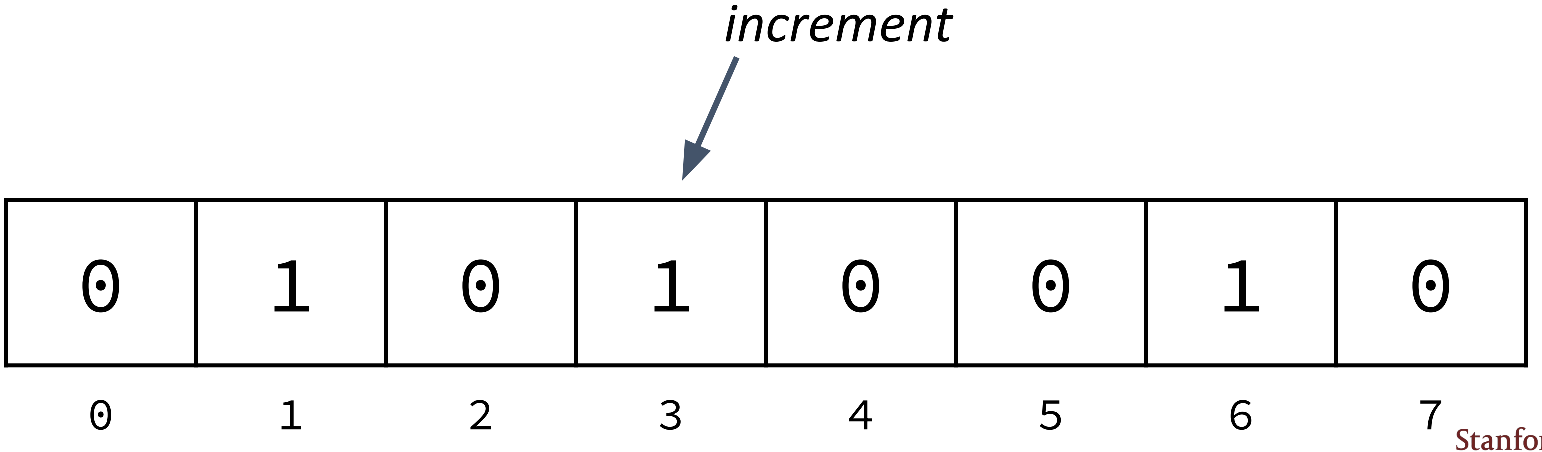
*increment*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

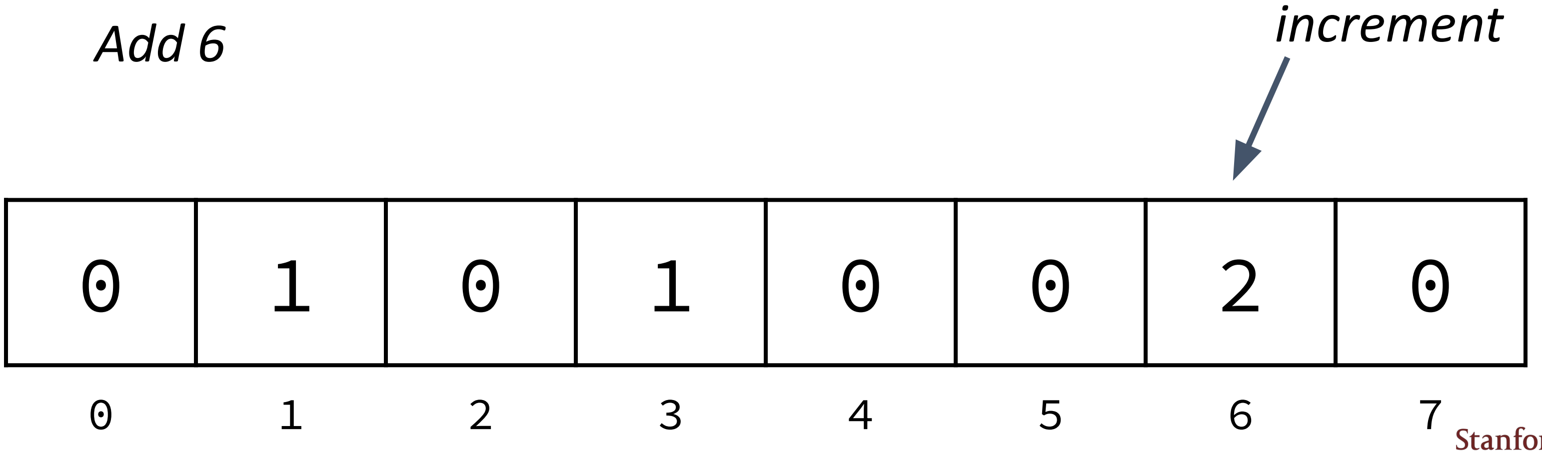
*Add 1*
*Add 6*
*Add 3*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Add 1*
*Add 6*
*Add 3*
*Add 6*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Add 1*
*Add 6*
*Add 3*
*Add 6*
*Add 5*

*increment*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

| 0 | 1 | 0 | 1 | 0 | 1 | 2 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Contains 3?*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Contains 3?* ***Yes.***

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | **1** | 0 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Contains 3? **Yes.***
*Contains 7?*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Contains 3?* ***Yes.***
*Contains 7?* ***No.***

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 2 | **0** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

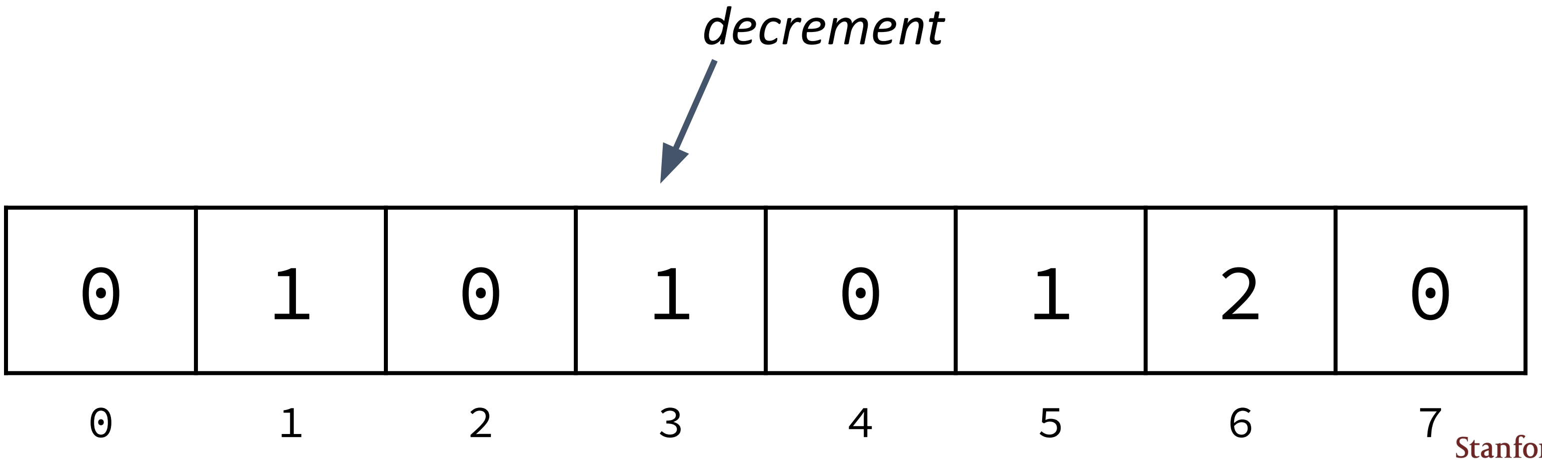
Each number gets its own index in the array, which stores a count

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 2 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Remove 3*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Remove 3*

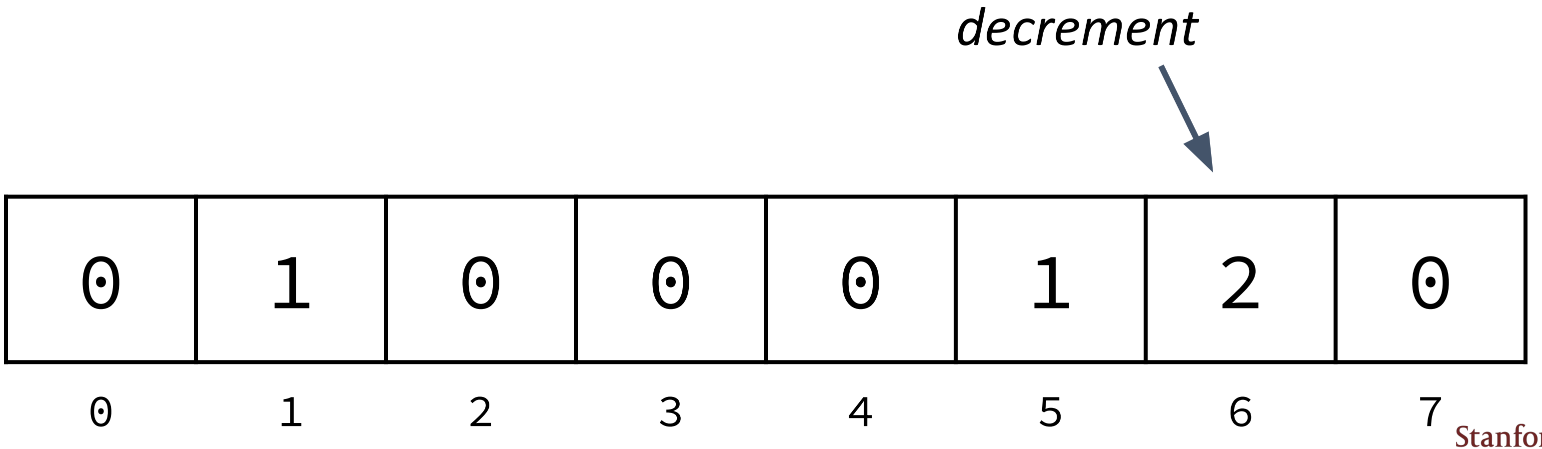
*decrement*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 2 | 0 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Remove 3*
*Remove 6*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

*Remove 3*
*Remove 6*

*decrement*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

> *What do we like about this approach?*
> *What don't we like?*

| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- What about bigger numbers? *How do we add 1732?*

| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- What about bigger numbers? *How do we add 1732?*

*We need to increment at index 1732*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- What about bigger numbers? *How do we add 1732?*

*We need to increment at index 1732*

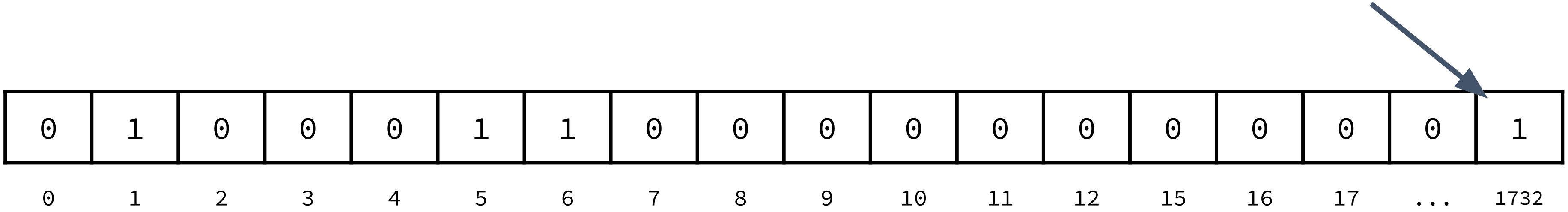
| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 16 | 17 | ... | 1732 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains/add/remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- What about bigger numbers? *How do we add 1732?*

*We need to increment at index 1732*



# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- What about bigger numbers? *How do we add 1732?*

> *Now we have a **sparse array**…*
> *This is a waste of space!*

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 16 | 17 | ... | 1732 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- Lots of wasted space if we're storing a large range of numbers ❌

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 16 | 17 | ... | 1732 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

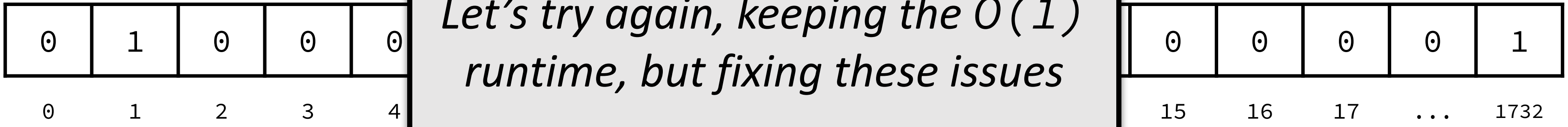
- `contains`/`add`/`remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- Lots of wasted space if we're storing a large range of numbers ❌
- We can't store negative numbers (no negative indices) ❌

| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 16 | 17 | ... | 1732 |

# Idea 1: Count Array

Each number gets its own index in the array, which stores a count

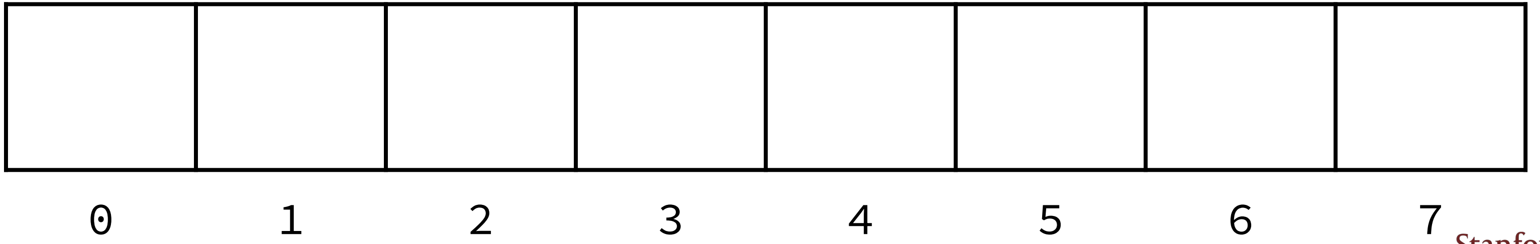
- `contains/add/remove` are all `O(1)` ✅
  - This is because we can index into an array in constant time!
- Lots of wasted space if we're storing a large range of numbers ❌
- We can't store negative numbers (no negative indices) ❌




*Let's try again, keeping the `O(1)` runtime, but fixing these issues*

# Idea 2: Modulo Array

- We have an array with b "buckets" - these are just the indices!
- We store each value `num` in bucket `num % b`



# Aside: Modulo Operator

Modulo is the remainder of a division operation

- `16 % 8 = 0`
  - 8 fits into 16 twice, with none left over
- `-10 % 8 = 6`
  - -10 is 6 away from -16
- `39 % 8 = 7`
  - 8 fits into 39 four times, with 1 left over

# Aside: Modulo Operator

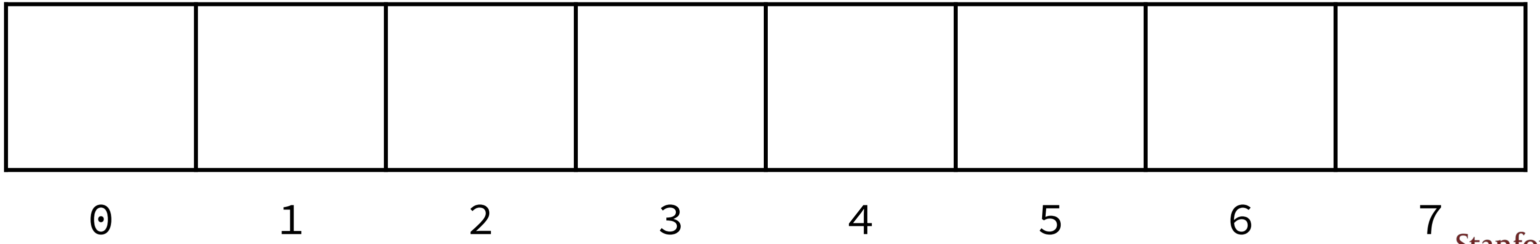
Modulo is the remainder of a division operation

- `16 % 8 =` **`0`**
  - 8 fits into 16 twice, with none left over
- `-10 % 8 =` **`6`**
  - -10 is 6 away from -16
- `39 % 8 =` **`7`**
  - 8 fits into 39 four times, with 1 left over

*When we mod by a number X, the result will be less than X*

# Idea 2: Modulo Array

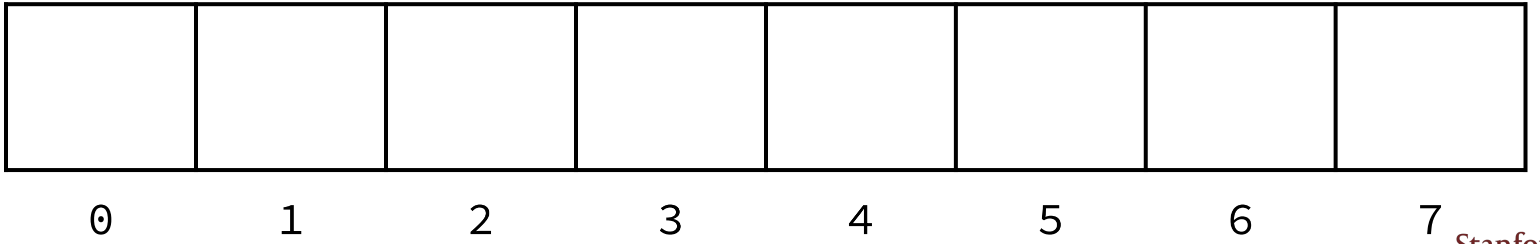
- We have an array with b "buckets" - these are just the indices!
- We store each value num in bucket `num % b`



# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*

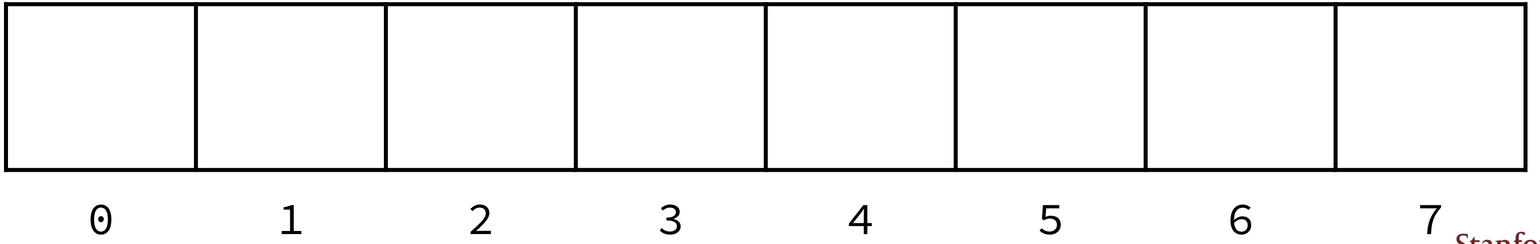


# Idea 2: Modulo Array

- We have an array with b "buckets" - these are just the indices!
- We store each value `num` in bucket `num % b`

*Add 5*

`5 % 8 = 5`



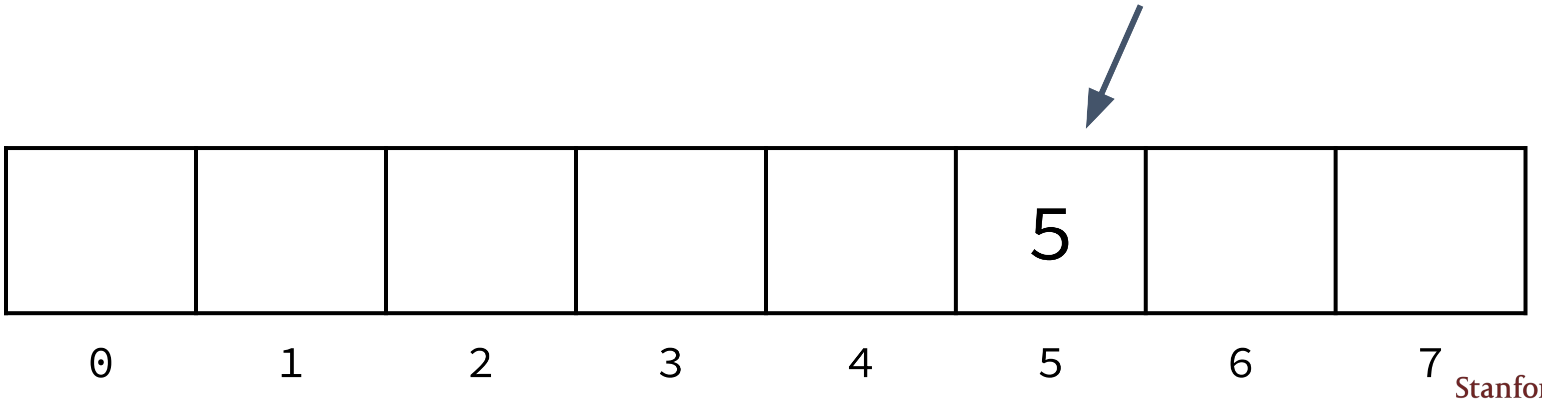
# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value `num` in bucket `num % b`

*Add 5*

`5 % 8 = 5`

*store here*



# Idea 2: Modulo Array

- We have an array with b "buckets" - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*
*Add 1732*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | 5 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*

*Add 1732* `1732 % 8 = 4`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | 5 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

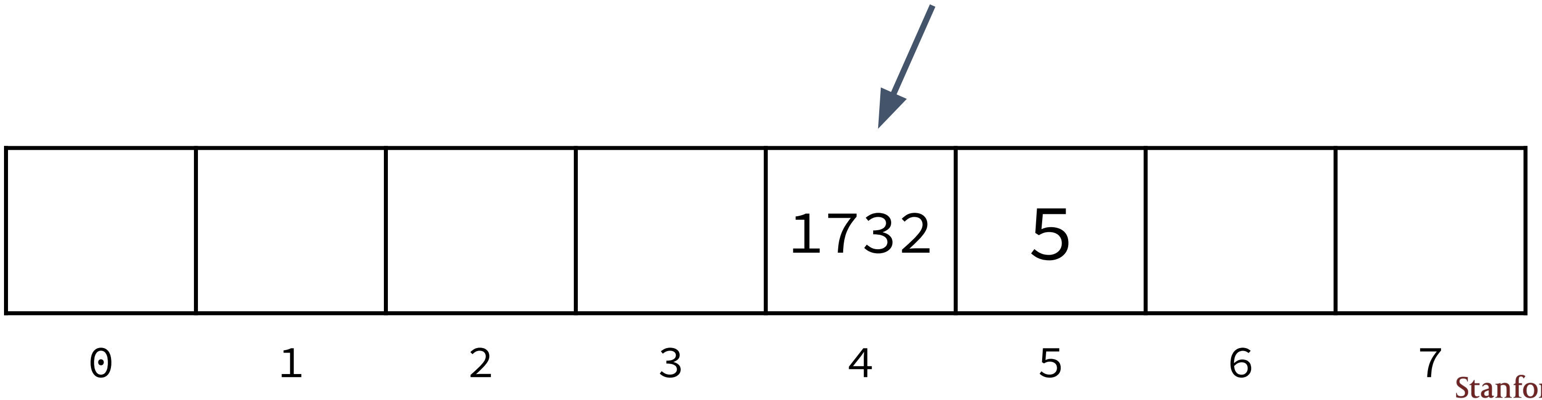
- We have an array with b "buckets" - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*

*Add 1732*

`1732 % 8 = 4`

*store here*



# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*
*Add 1732*
***Add 8***
***Add -2***

*Try adding these values to our array!*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | 1732 | 5 | | |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*
*Add 1732*
***Add 8***
*Add -2*

`8 % 8 = 0`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **8** | | | | 1732 | 5 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 5*

*Add 1732*

*Add 8*

***Add -2***

`−2 % 8 = 6`

| 8 | | | | 1732 | 5 | **−2** | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*`num % b` gives us a valid index within our array*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | | | | 1732 | 5 | -2 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value `num` in bucket `num % b`

*Add 4*

| 8 | | | | 1732 | 5 | -2 | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b "buckets" - these are just the indices!
- We store each value num in bucket `num % b`

*Add 4*

`4 % 8 = 4`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | | | | 1732 | 5 | -2 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Idea 2: Modulo Array

- We have an array with b “buckets” - these are just the indices!
- We store each value num in bucket `num % b`

*Add 4*

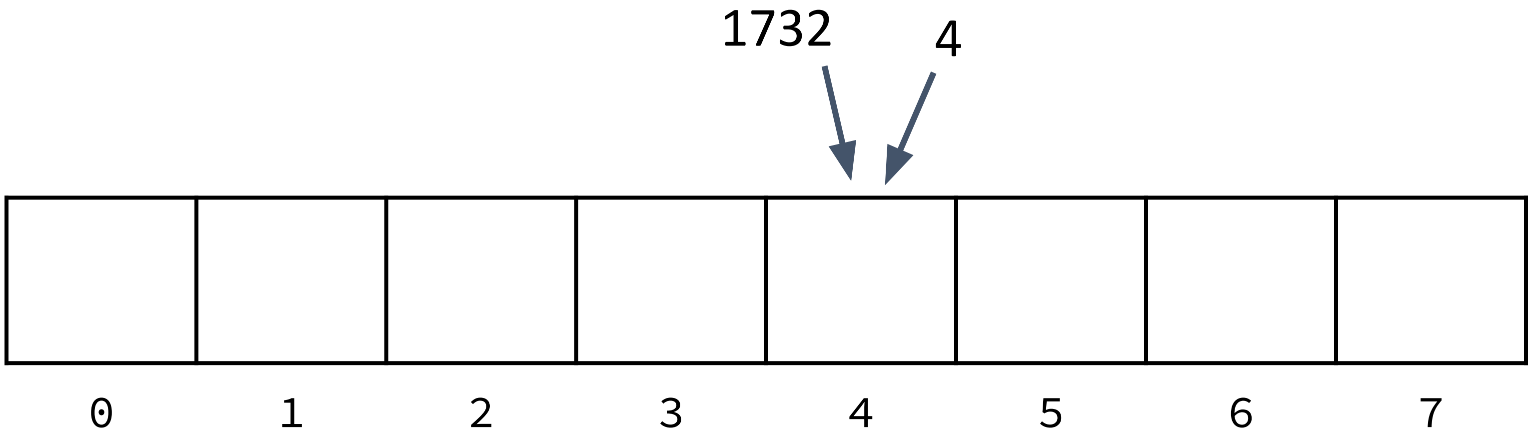
`4 % 8 = 4`

COLLISION!

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 8 | | | | **1732** | 5 | -2 | |

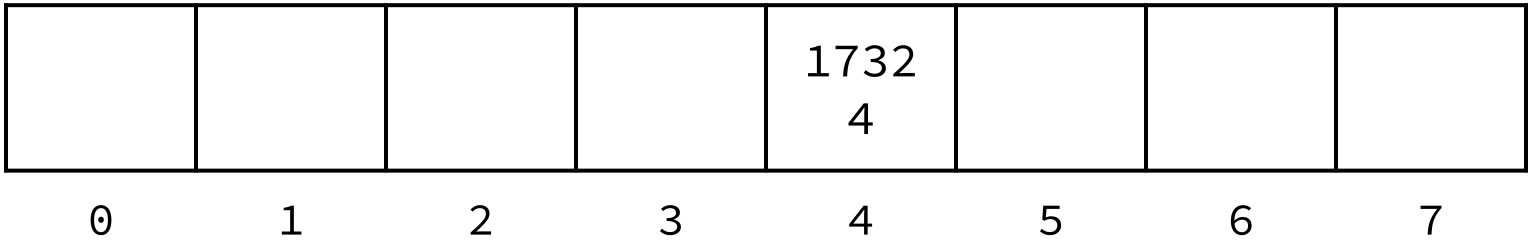
# Dealing with Collisions

- Sometimes, two elements will be assigned to the same bucket
  - This is called a collision!



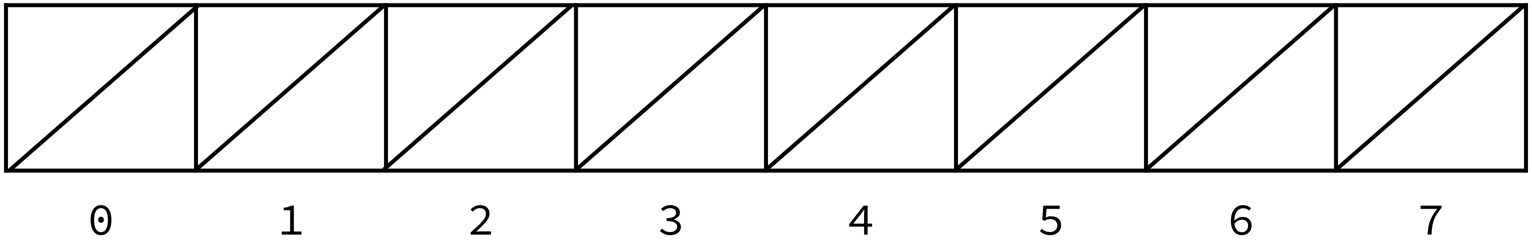
# Dealing with Collisions

- Sometimes, two elements will be assigned to the same bucket
  - This is called a collision!
- We'd like to be able to store multiple elements in the same bucket



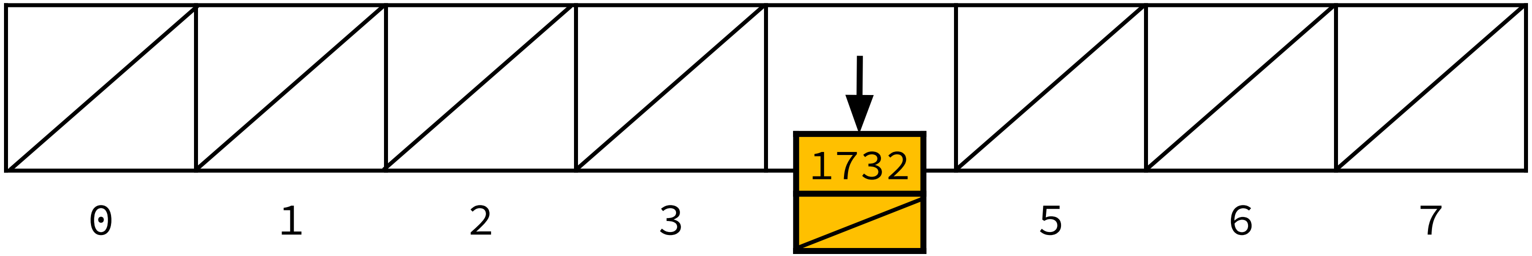
# Dealing with Collisions

- Sometimes, two elements will be assigned to the same bucket
  - This is called a collision!
- We'd like to be able to store multiple elements in the same bucket
- One idea: each bucket stores a linked list of elements
  - If we prepend new nodes to the beginning of our list, this is still `O(1)`



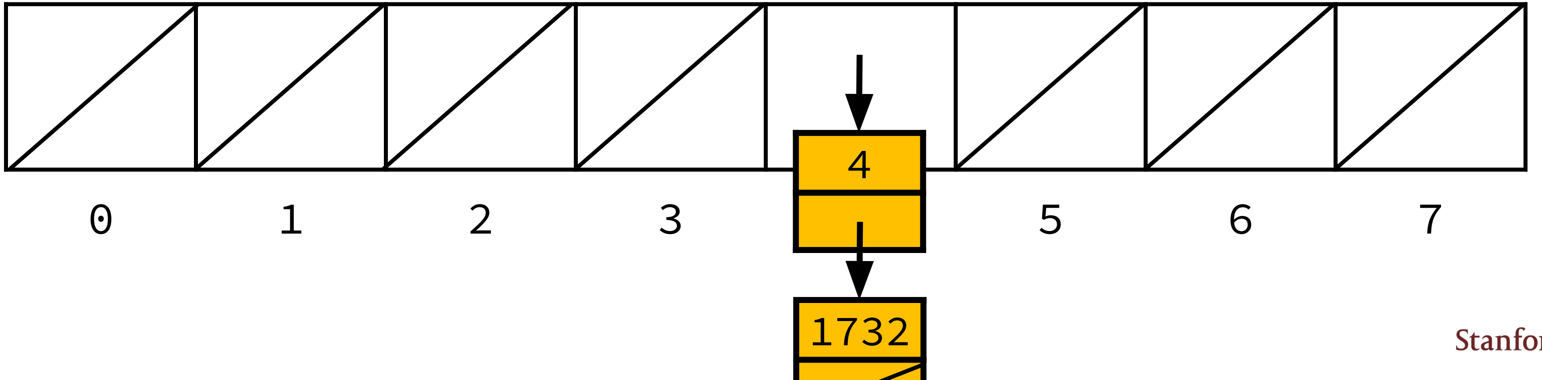
# Dealing with Collisions

- Sometimes, two elements will be assigned to the same bucket
  - This is called a collision!
- We'd like to be able to store multiple elements in the same bucket
- One idea: each bucket stores a linked list of elements
  - If we prepend new nodes to the beginning of our list, this is still O(1)



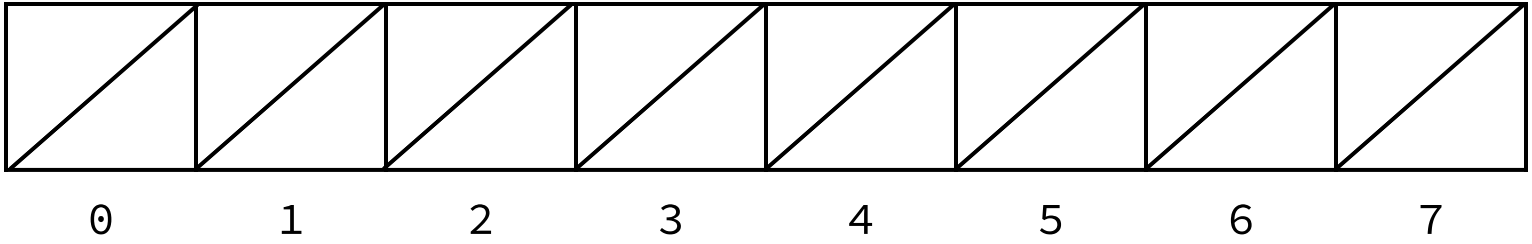
# Dealing with Collisions

- Sometimes, two elements will be assigned to the same bucket
  - This is called a collision!
- We'd like to be able to store multiple elements in the same bucket
- One idea: each bucket stores a linked list of elements
  - If we prepend new nodes to the beginning of our list, this is still `O(1)`



# Idea 3: Array of Linked Lists

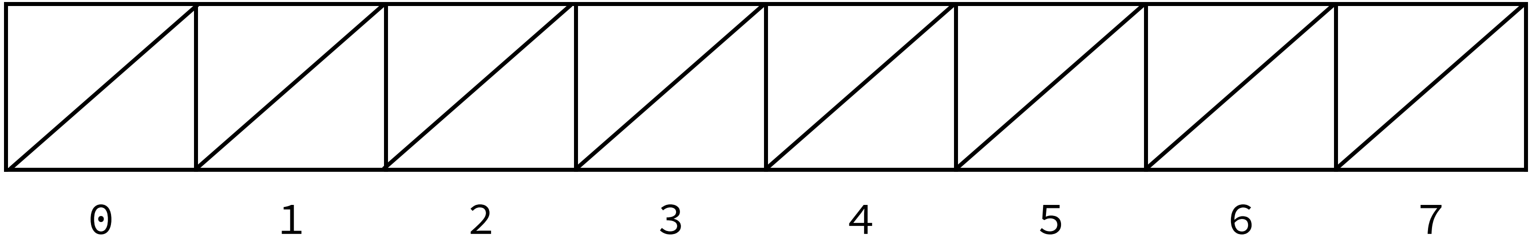
- We have an array of linked lists with b "buckets"
- We store each value `num` in the linked list of bucket `num % b`



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b "buckets"
- We store each value num in the linked list of bucket num % b

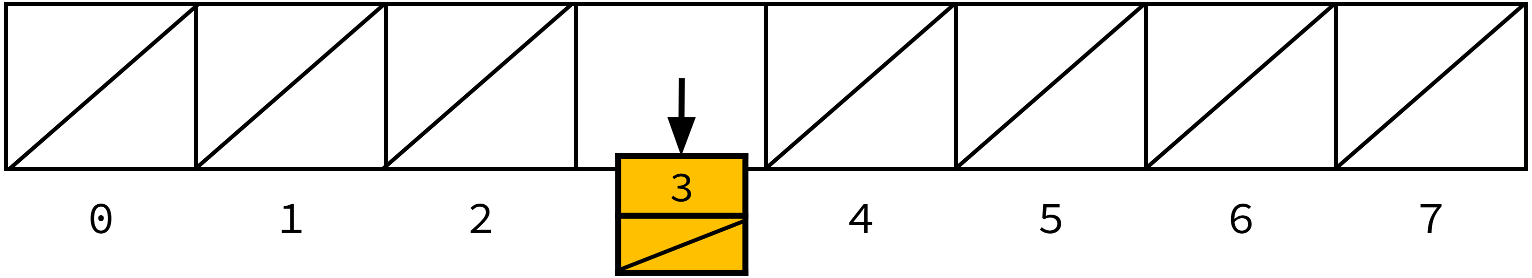
*Add 3*



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket num % b

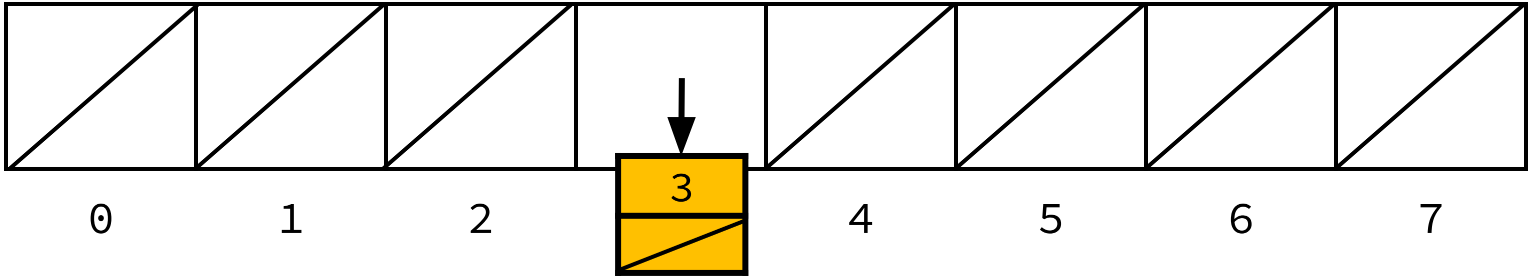
*Add 3* 3 % 8 = 3



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b "buckets"
- We store each value num in the linked list of bucket `num % b`

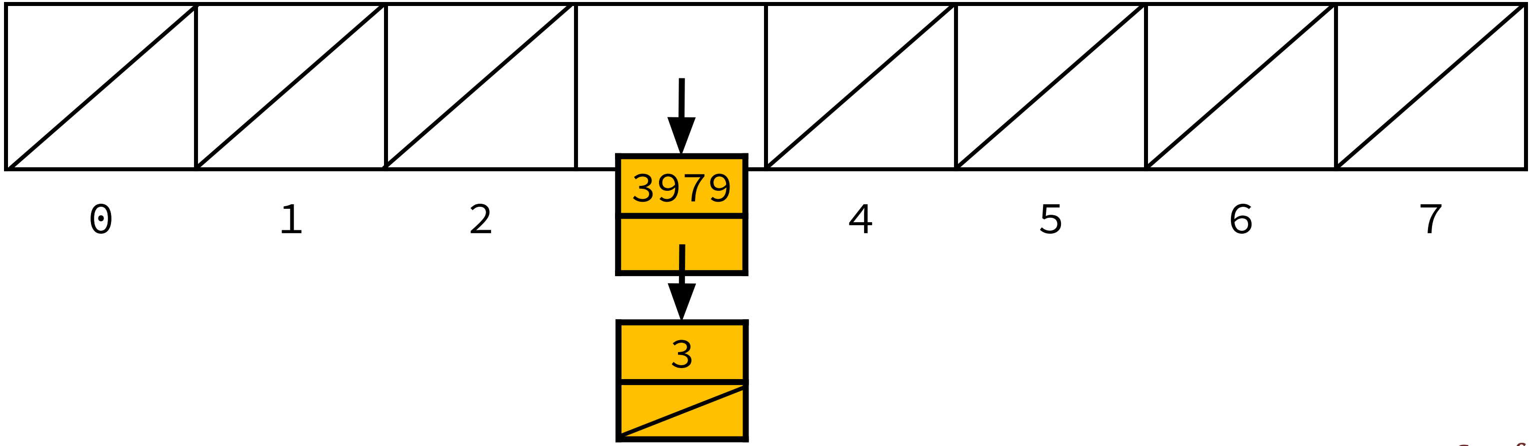
*Add 3979*



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b "buckets"
- We store each value num in the linked list of bucket num % b

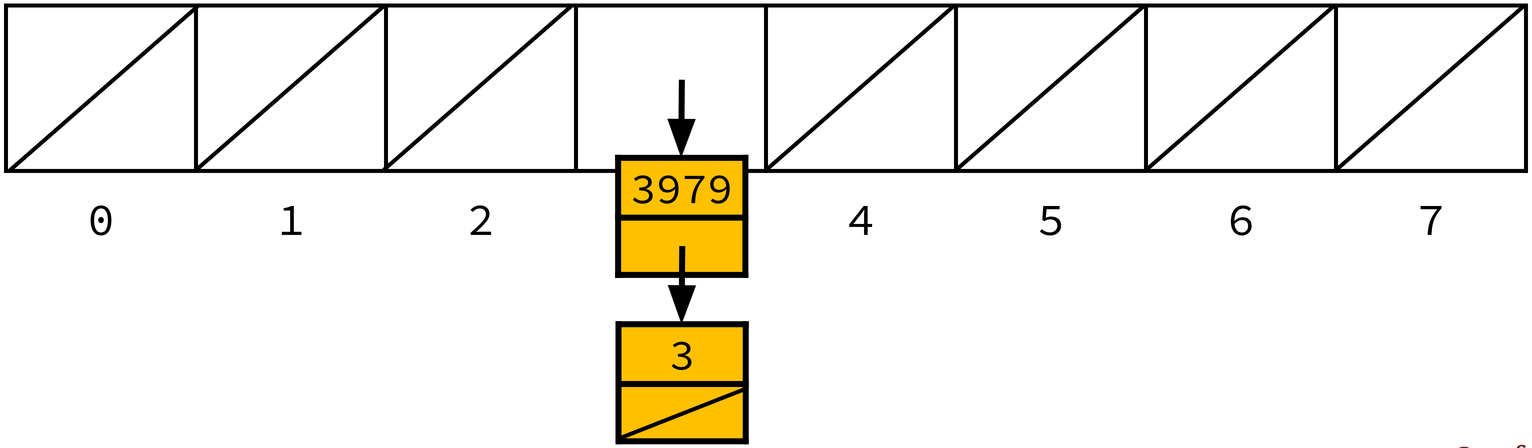
*Add 3979* 3979 % 8 = 3



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b "buckets"
- We store each value num in the linked list of bucket num % b

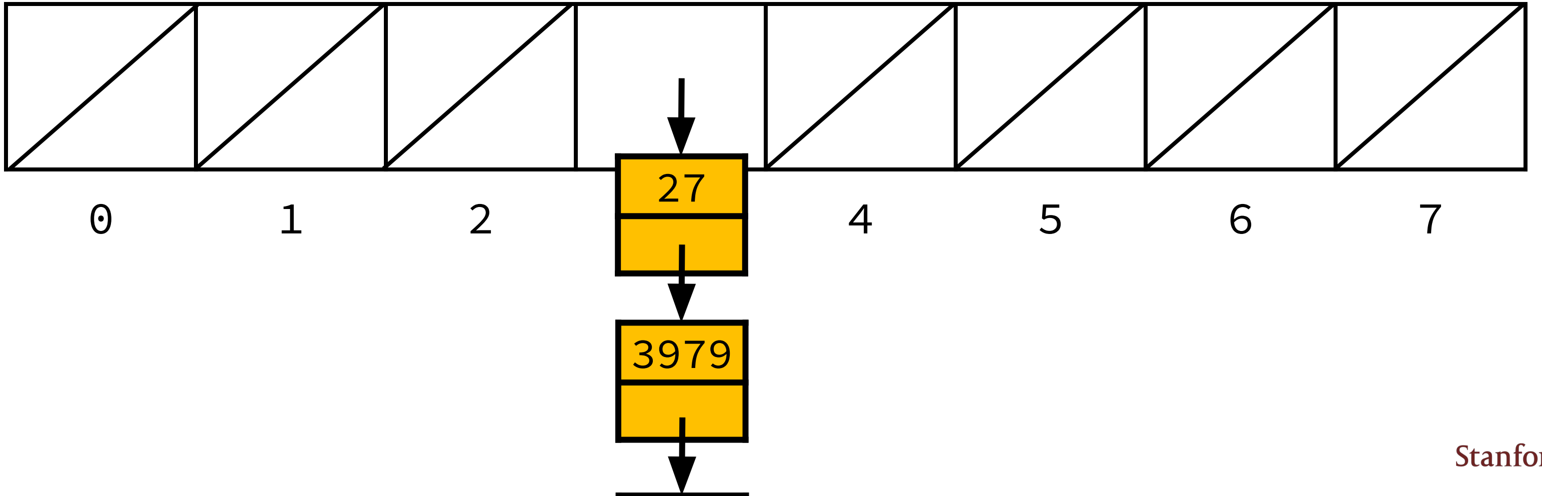
*Add 27*



# Idea 3: Array of Linked Lists

- We have an array of linked lists with b “buckets”
- We store each value num in the linked list of bucket `num % b`

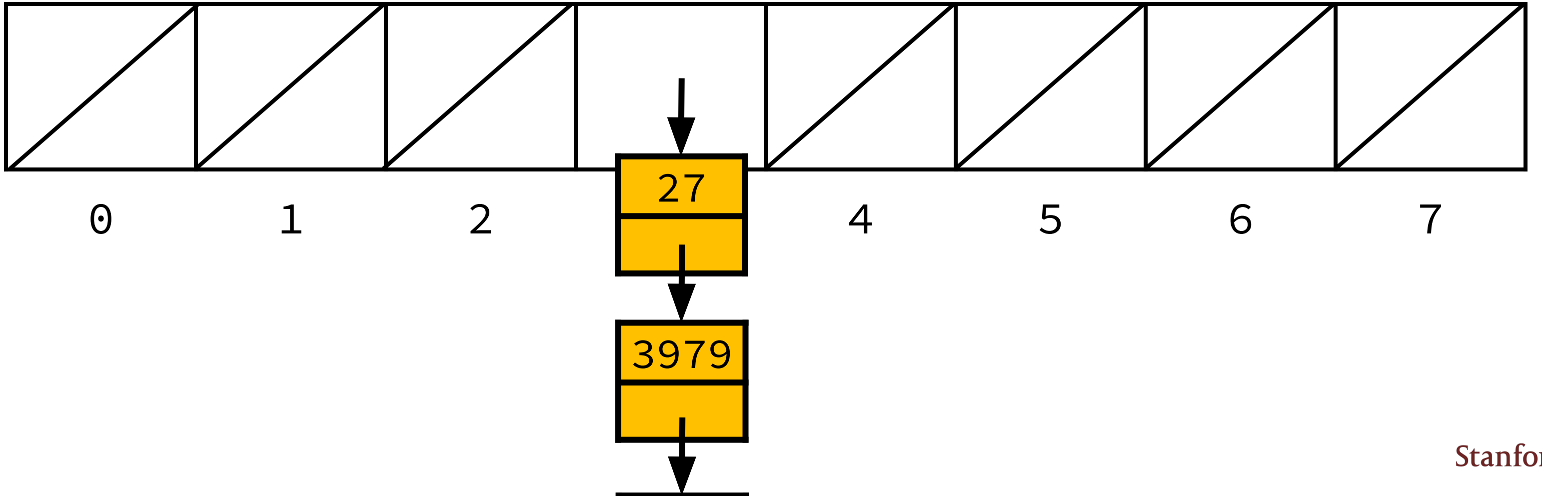
*Add 27* `27 % 8 = 3`



# Idea 3: Array of Linked Lists

- We have a
- We store ... um % b

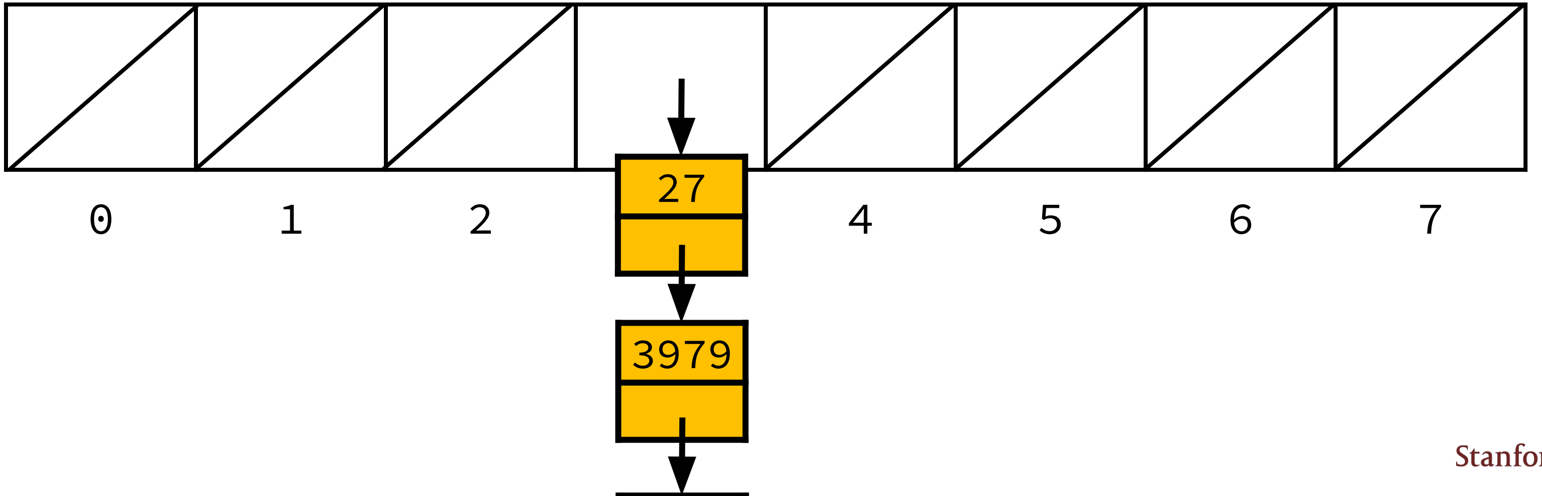
🤔 *Why don't we like this?*
*Hint: think of* `contains` *and* `remove`



# Idea 3: Array of Linked Lists

- We hav
- We sto % b

If all n of our elements end up in the same bucket, *contains and remove* will be `O(n)`



# Hashing

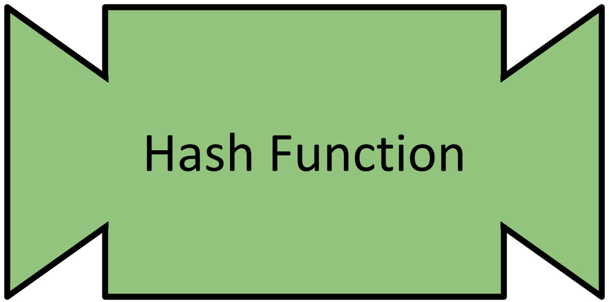
How can we evenly distribute our elements across our buckets?

# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!

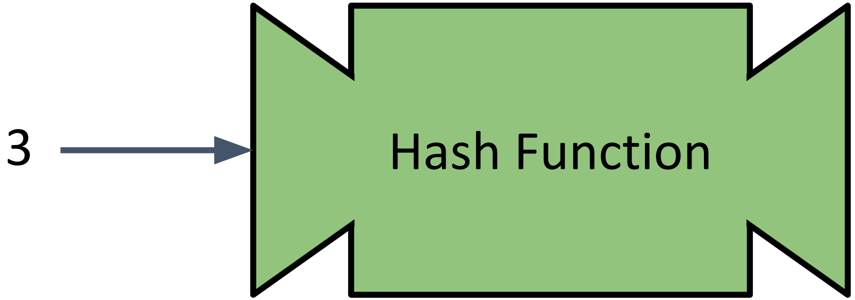
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!



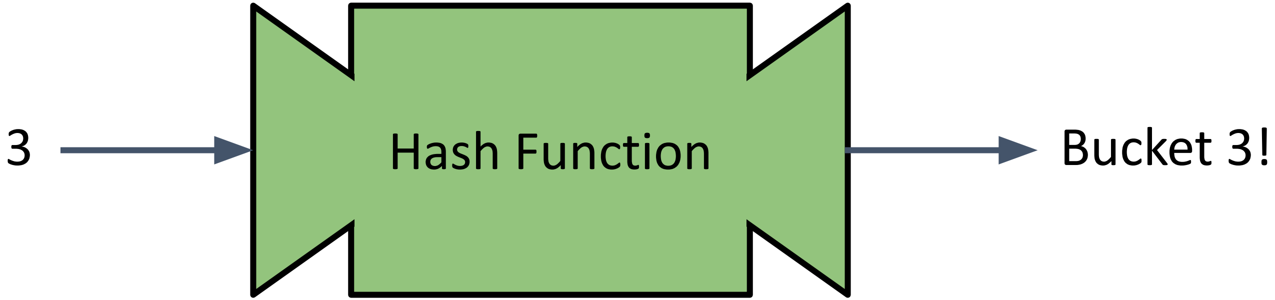
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!



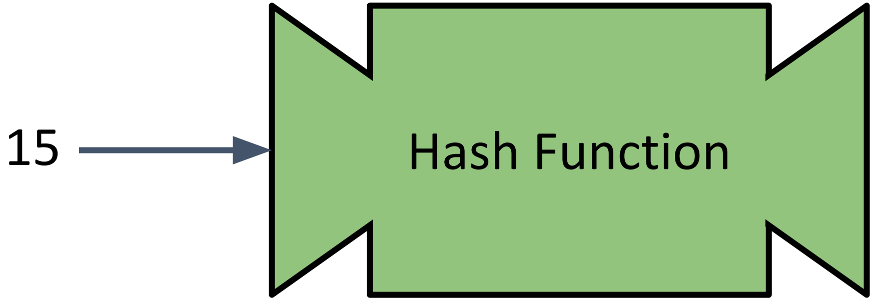
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!



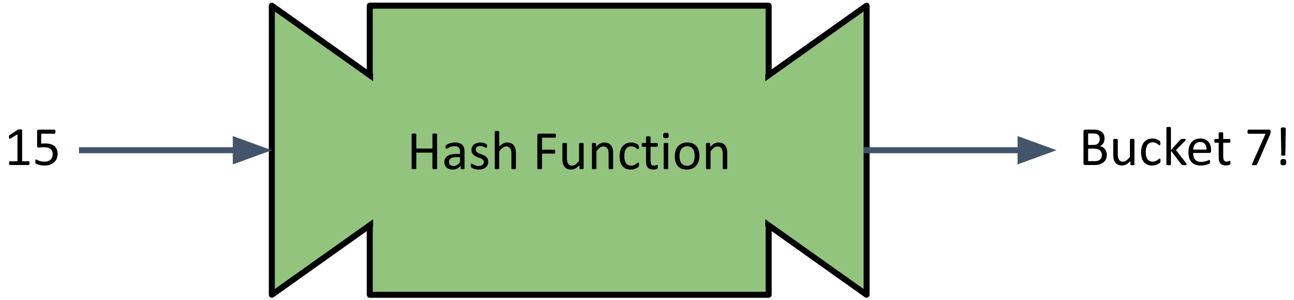
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!



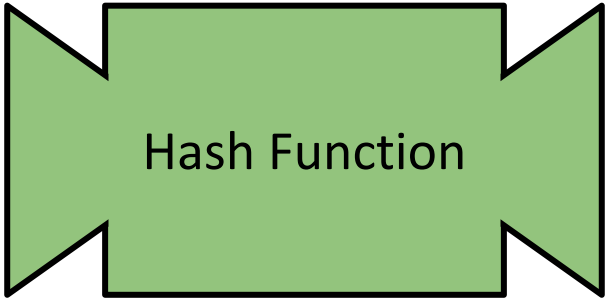
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!



# Hash Functions

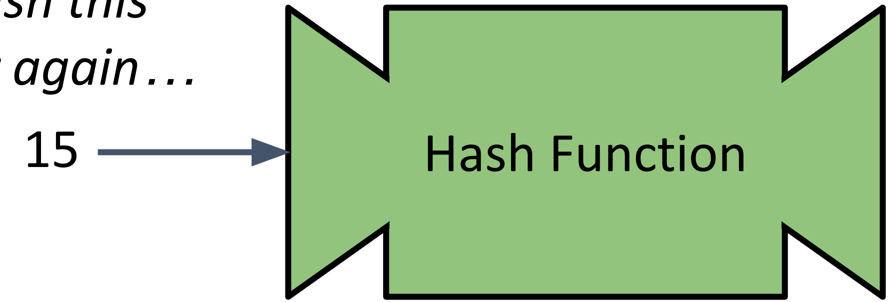
- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!
- A hash function must be **deterministic**: same input produces same output



# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!
- A hash function must be **deterministic**: same input produces same output

*If we hash this number again…*

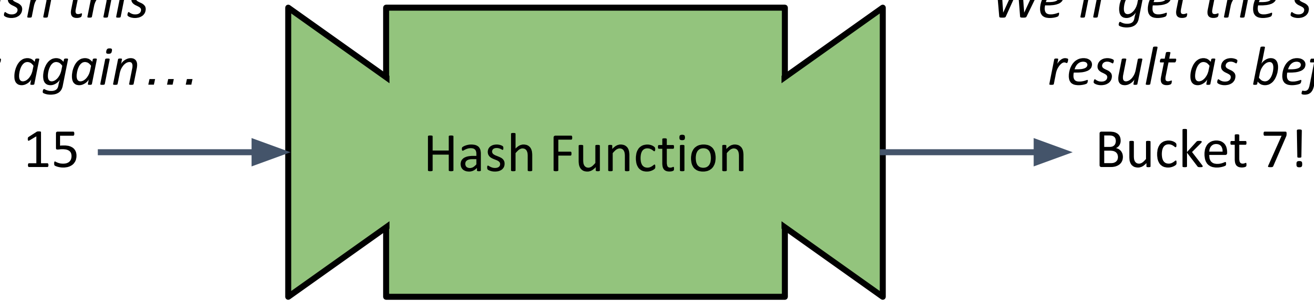


# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!
- A hash function must be **deterministic**: same input produces same output

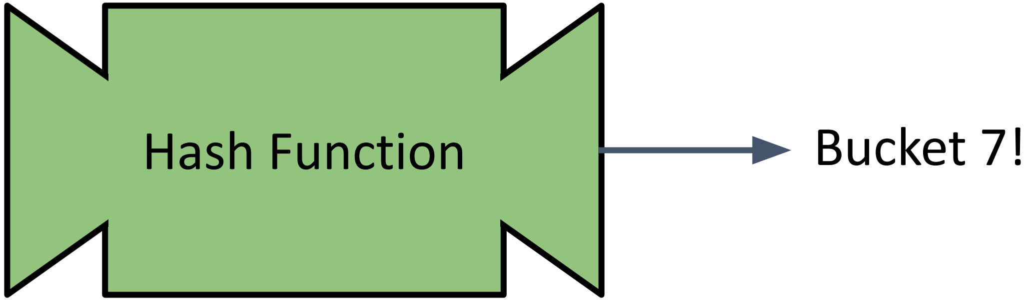
*If we hash this number again…*

*We'll get the same result as before.*



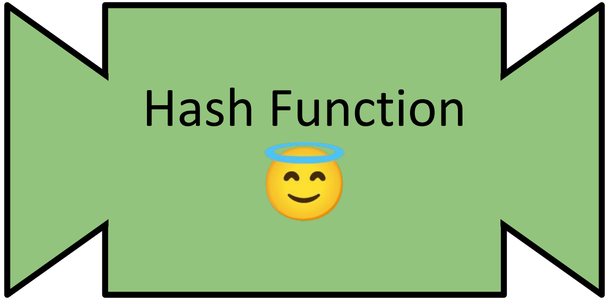
# Hash Functions

- A **hash function** is a function that assigns elements to buckets
  - We've been using the % operator as our hash function thus far!
- A hash function must be **deterministic**: same input produces same output
- We call the output of a hash function a **hash code** or **hash value**



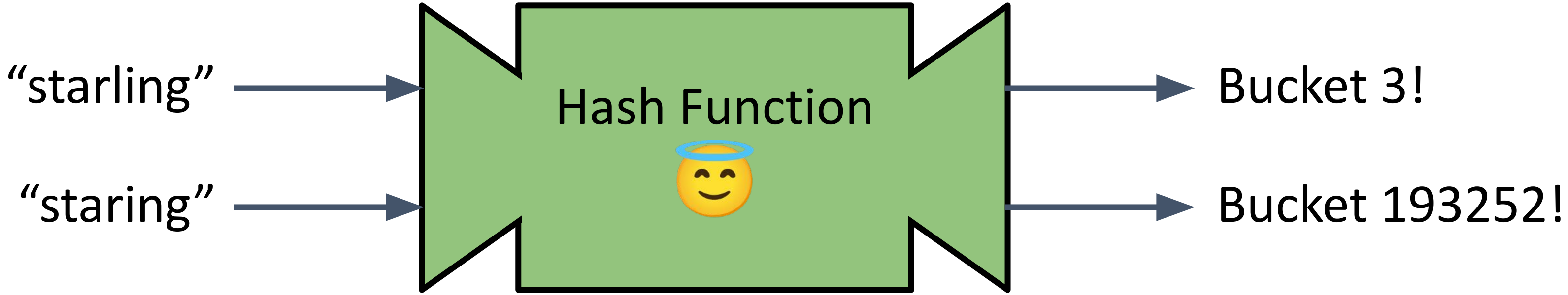
# Good Hash Functions

- A good hash function distributes elements evenly across buckets
  - This way, no bucket contains too many elements



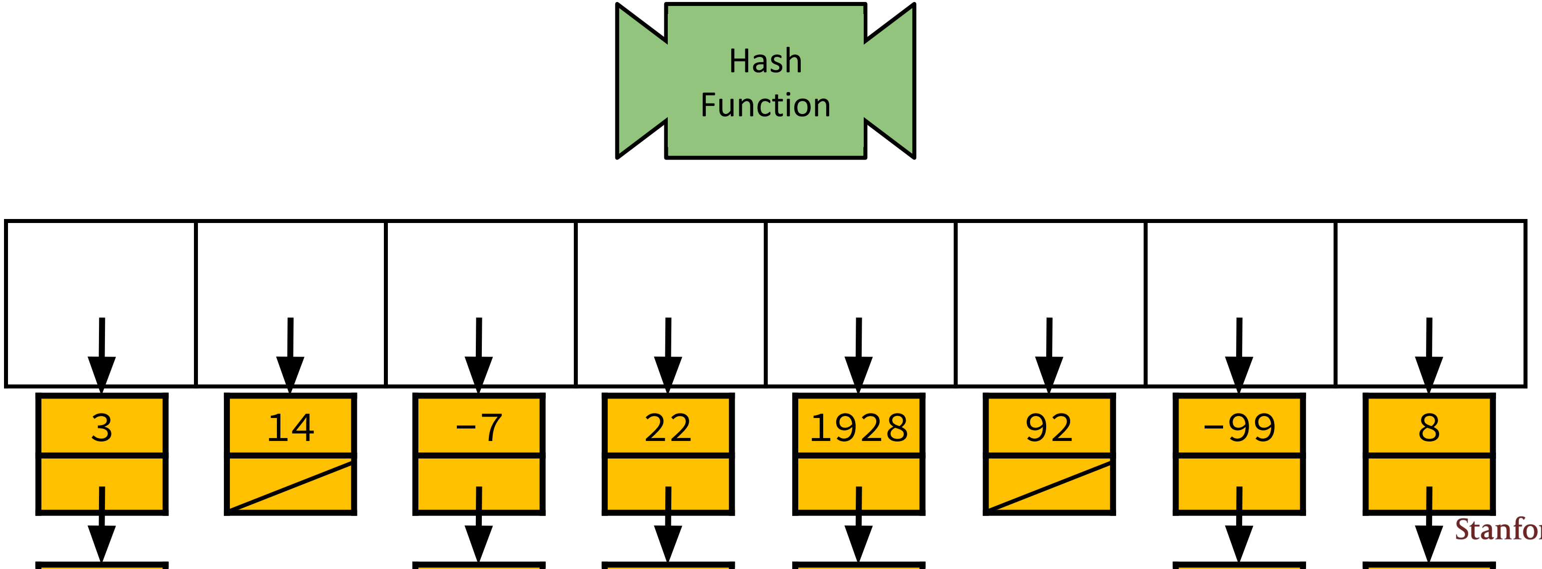
# Good Hash Functions

- A good hash function distributes elements evenly across buckets
  - This way, no bucket contains too many elements
- Similar inputs will not necessarily have similar hash codes



# A Great Idea: Chaining Hash Table

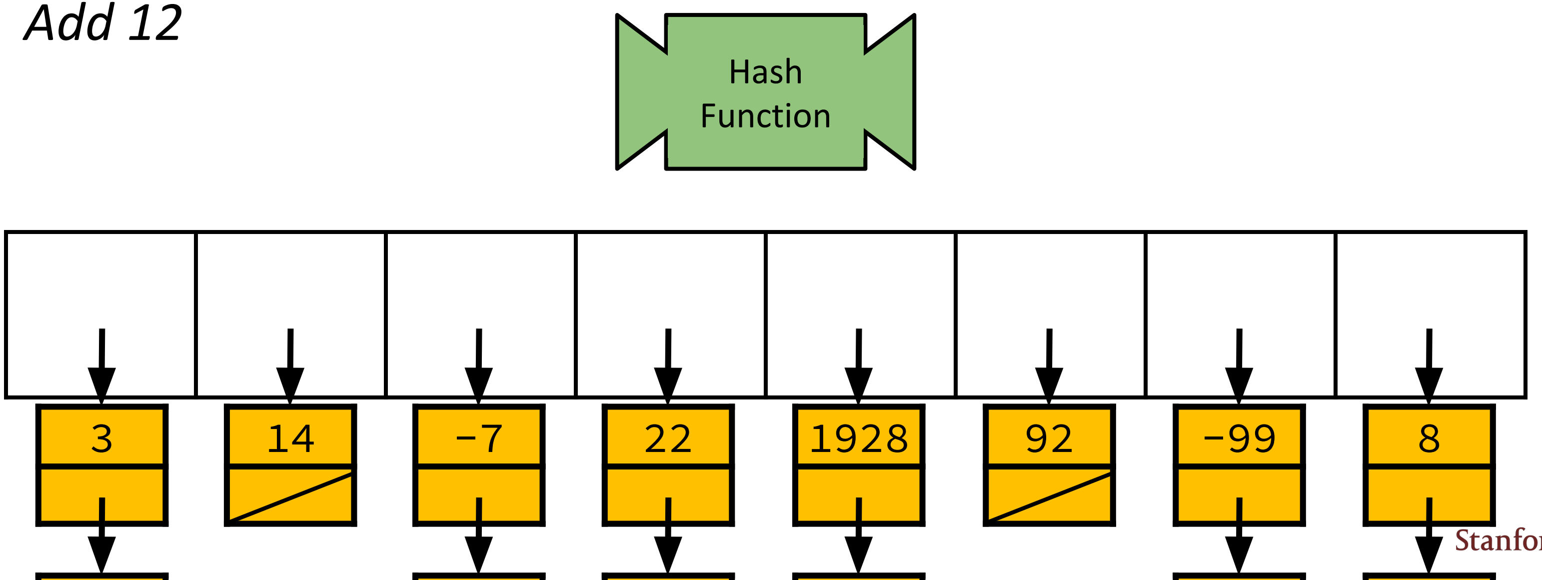
- We have an array of linked lists with b "buckets"
- We store each value `num` in the linked list of bucket `hash(num)`



# A Great Idea: Chaining Hash Table

- We have an array of linked lists with b "buckets"
- We store each value `num` in the linked list of bucket `hash(num)`

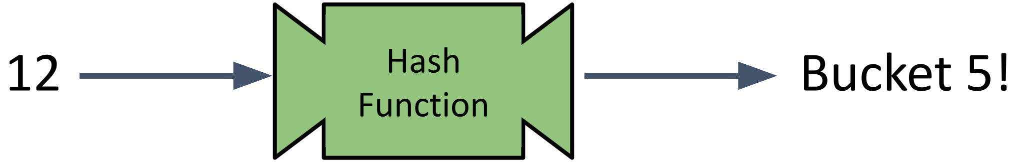
*Add 12*

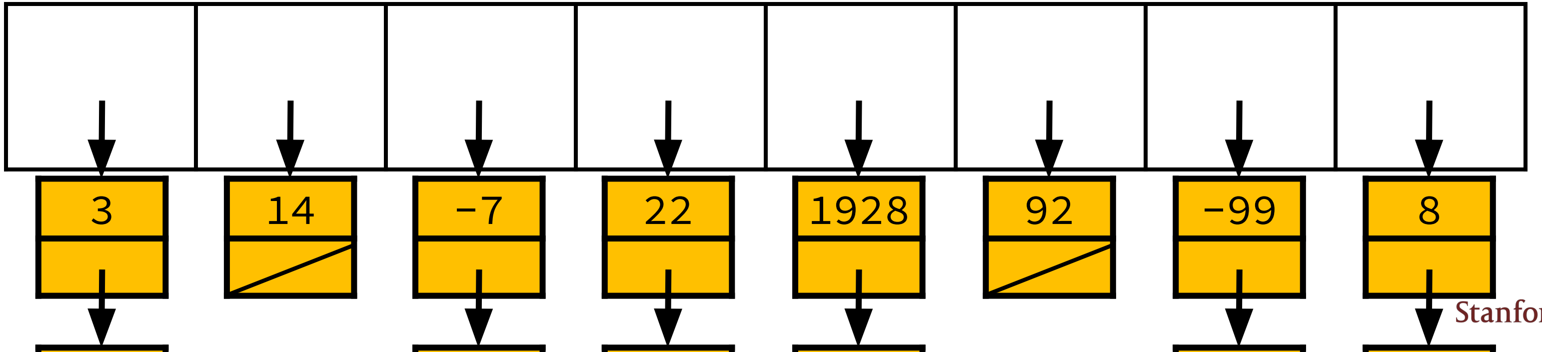


# A Great Idea: Chaining Hash Table

- We have an array of linked lists with b "buckets"
- We store each value `num` in the linked list of bucket `hash(num)`

*Add 12*

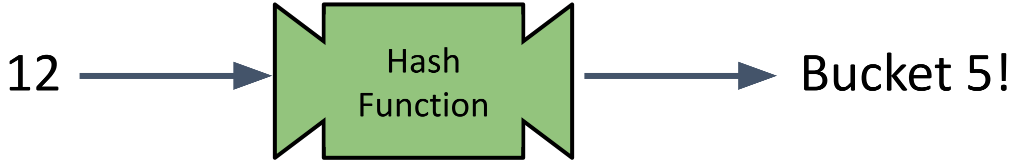


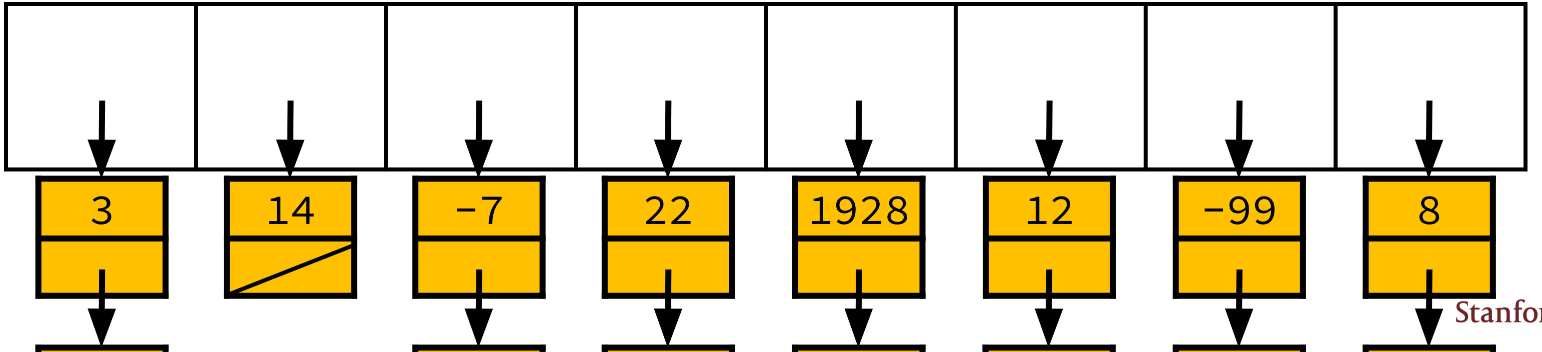



# A Great Idea: Chaining Hash Table

- We have an array of linked lists with b "buckets"
- We store each value num in the linked list of bucket hash(num)

*Add 12*



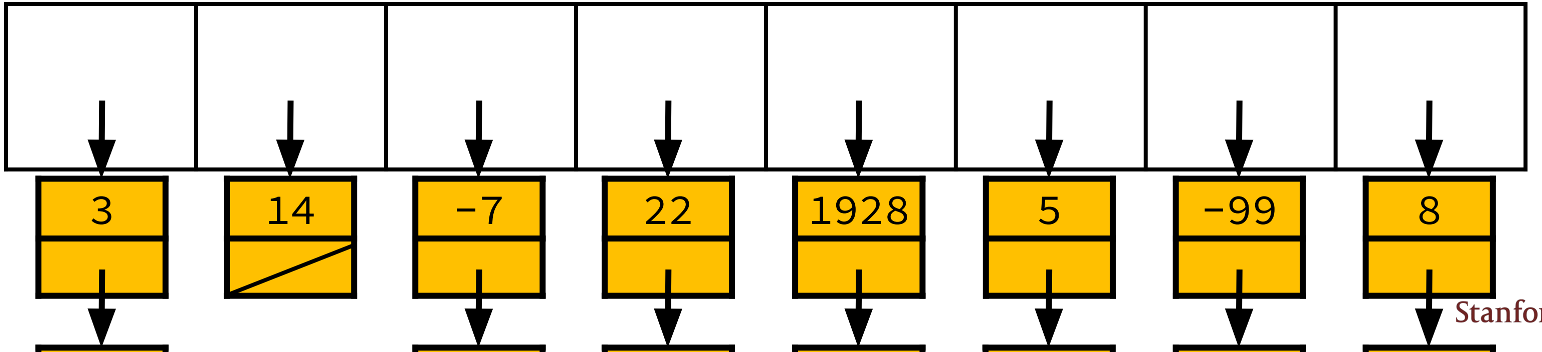
# A Great Idea: Chaining Hash Table

- We have an array of linked lists with b "buckets"
- We store each value `num` in the linked list of bucket `hash(num)`

*Add 12*

*If we've got a good hash function, and we've hashed n elements into b buckets, what's our average bucket size?*

5!



# Load Factor: n / b

- The average number of elements in each bucket
  - If the load factor is low: lots of empty buckets, wasted space
  - If the load factor is high: very full buckets, slow operations

# Load Factor: `n/b`

- The average number of elements in each bucket
  - If the load factor is low: lots of empty buckets, wasted space
  - If the load factor is high: very full buckets, slow operations
- This means we'll have to look through `O(n/b)` elements for `contains` and `remove`… is this better than `O(n)`?

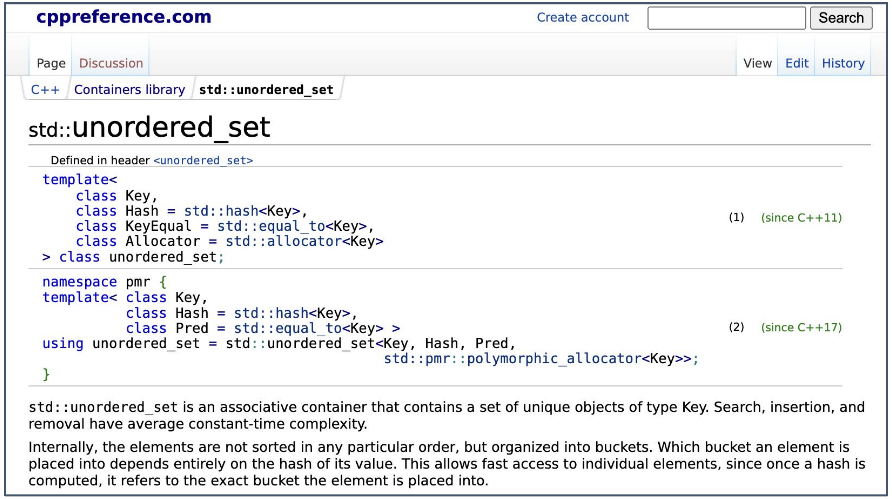
# Load Factor: `n/b`

- The average number of elements in each bucket
  - If the load factor is low: lots of empty buckets, wasted space
  - If the load factor is high: very full buckets, slow operations
- This means we'll have to look through `O(n/b)` elements for `contains` and `remove`… is this better than `O(n)`?

**Big idea**: if we choose b (# of buckets) to be a number close to n, then `n/b` will be constant.

# Introducing… HashSet!

A Stanford ADT that leverages Hash Tables to store a set of unique elements



cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::unordered_set

## std::unordered_set

Defined in header <unordered_set>

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```
(1) (since C++11)

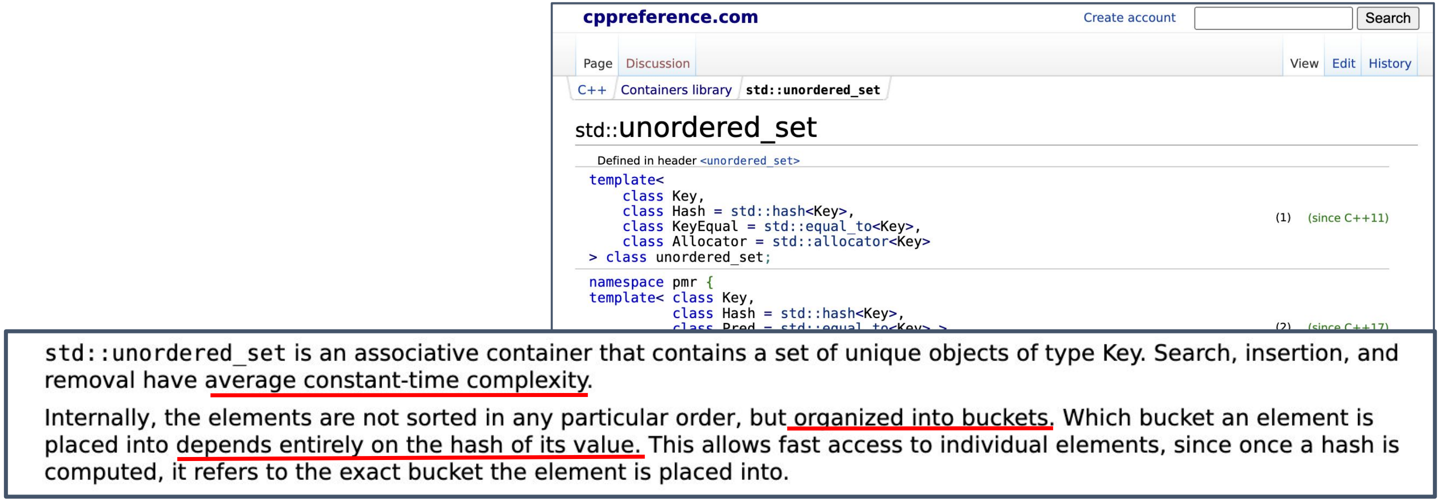
```
namespace pmr {
template< class Key,
          class Hash = std::hash<Key>,
          class Pred = std::equal_to<Key> >
using unordered_set = std::unordered_set<Key, Hash, Pred,
                                         std::pmr::polymorphic_allocator<Key>>;
}
```
(2) (since C++17)

std::unordered_set is an associative container that contains a set of unique objects of type Key. Search, insertion, and removal have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its value. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.

# Introducing… HashSet!

A Stanford ADT that leverages Hash Tables to store a set of unique elements



cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::unordered_set

## std::unordered_set

Defined in header <unordered_set>

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

(1) (since C++11)

```
namespace pmr {
template< class Key,
          class Hash = std::hash<Key>,
          class Pred = std::equal_to<Key> >
```

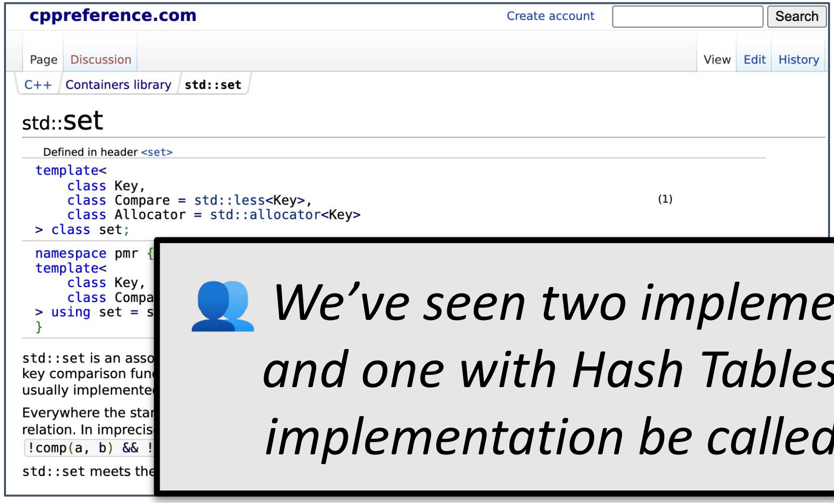
(2) (since C++17)

std::unordered_set is an associative container that contains a set of unique objects of type Key. Search, insertion, and removal have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its value. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.

# Introducing… HashSet!

A Stanford ADT that leverages Hash Tables to store a set of unique elements



cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::set

std::set

Defined in header <set>

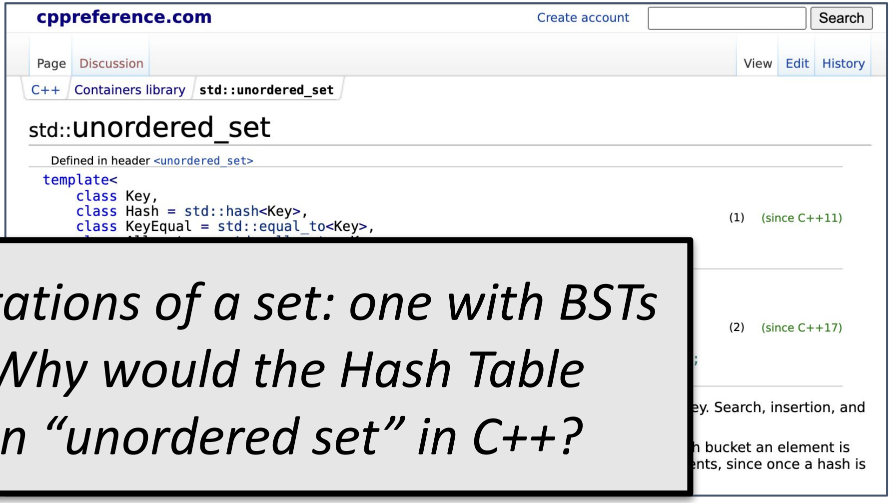
```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```
(1)

```
namespace pmr {
template<
    class Key,
    class Compa
> using set = s
}
```

std::set is an asso
key comparison fun
usually implemente

Everywhere the sta
relation. In imprecis
!comp(a, b) && !

std::set meets the



cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::unordered_set

std::unordered_set

Defined in header <unordered_set>

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
```
(1) (since C++11)

(2) (since C++17)

ey. Search, insertion, and

h bucket an element is
nts, since once a hash is

👥 *We've seen two implementations of a set: one with BSTs and one with Hash Tables. Why would the Hash Table implementation be called an "unordered set" in C++?*

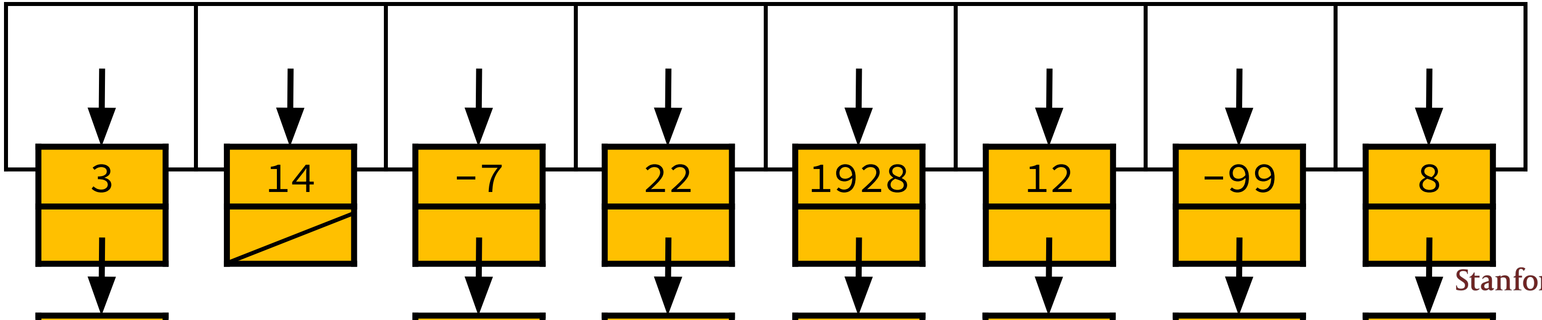
# Introducing… HashSet!

A Stanford ADT that leverages Hash Tables to store a set of unique elements

# Let’s Draw it Out!

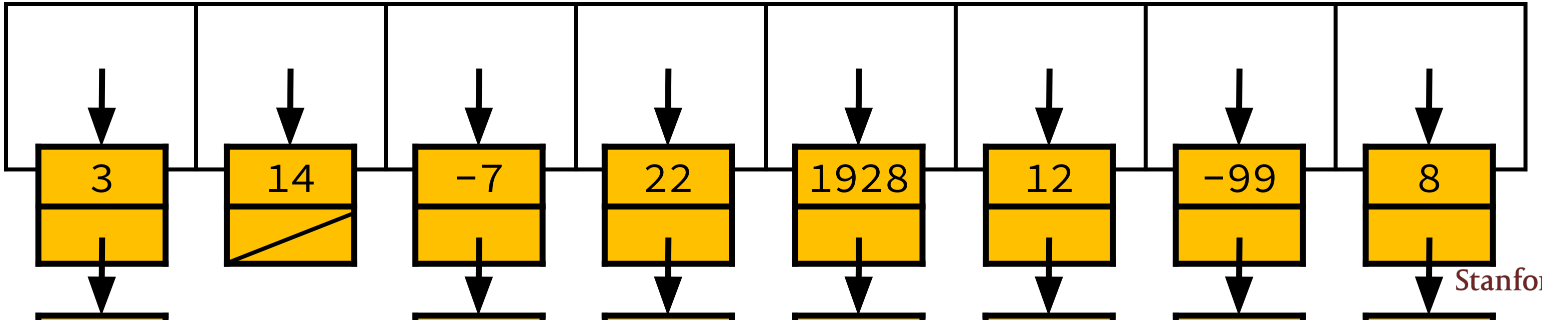
# HashSet

| Operation | Runtime |
| --- | --- |
| Contains | O(n/b) |
| Insert | O(n/b) |
| Remove | O(n/b) |



# HashSet

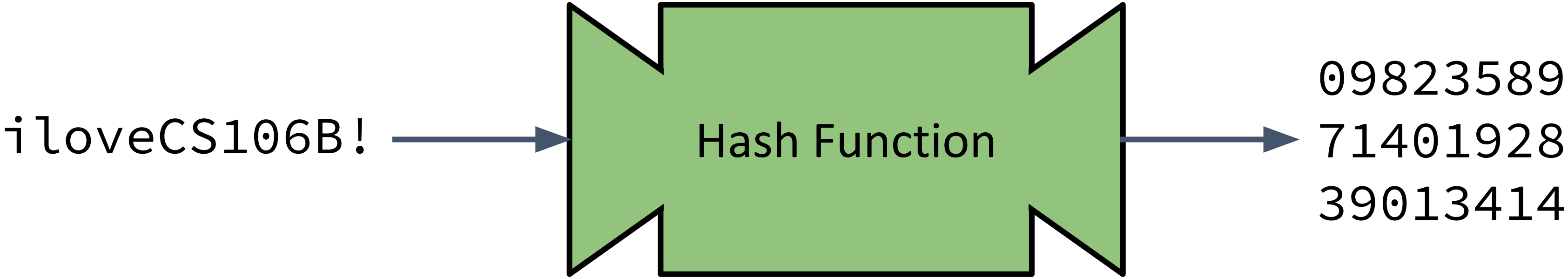
| Operation | Runtime |
| --- | --- |
| Contains | O(1) |
| Insert | O(1) |
| Remove | O(1) |



# Applications of Hashing

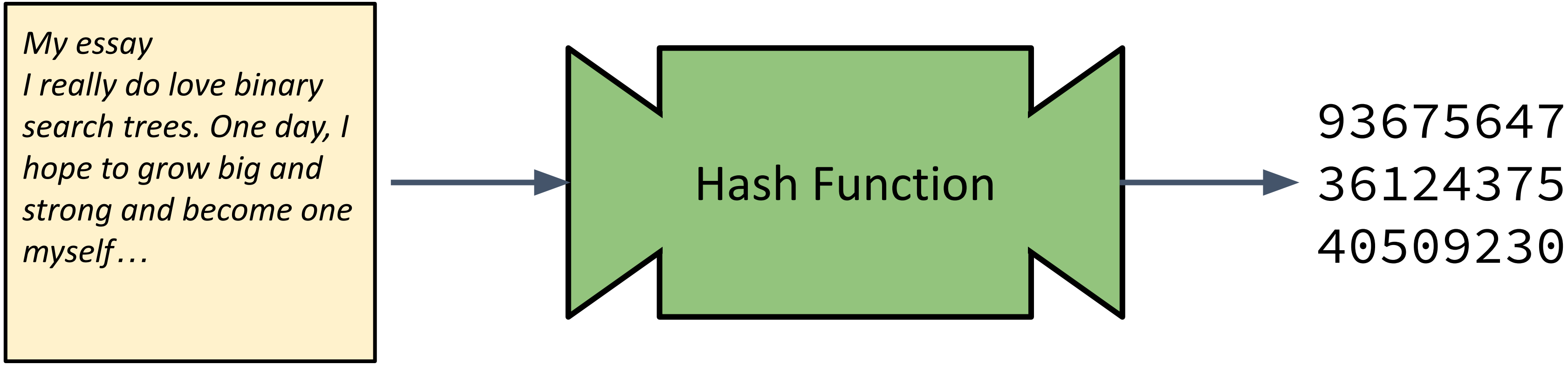
# Cryptography

- Rather than storing your password, websites will store a hash of your password



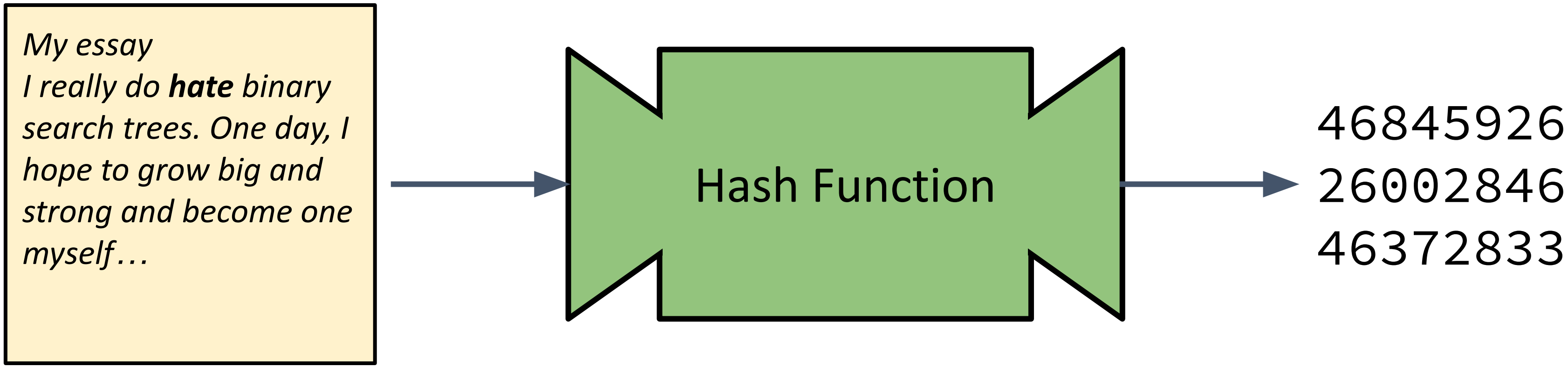
# Cryptography

- We can use hash functions to verify data



# Cryptography

- We can use hash functions to verify data



# Assignment 0

- We used a hash function to assign every student a unique hash code that couldn't be replicated without running the hash function

```
int nameHash(string first, string last) {
    static const int kLargePrime = 16908799;
    static const int kSmallPrime = 127;
    int hashVal = 0;

    for (char ch: first + last) {
        ch = tolower(ch);
        hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
    }

    return hashVal;
}
```

# Recap

- ADT showdown
- Achieving `O(1)` `contains/add/remove`
- Hash functions and hash tables
- `HashSet/unordered_set`
- Applications of hash functions

# Thank you!