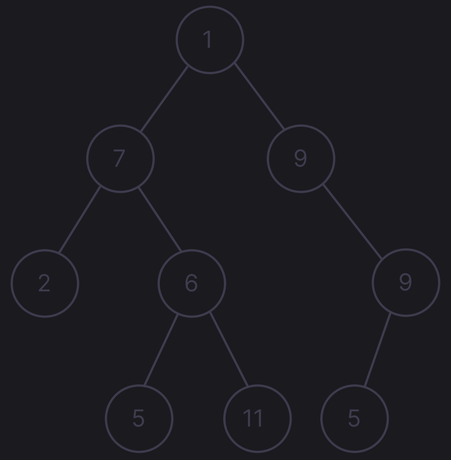
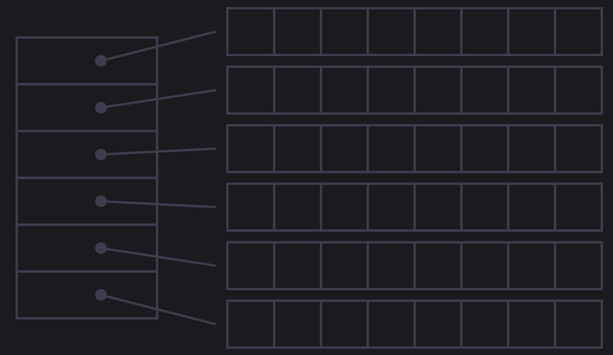
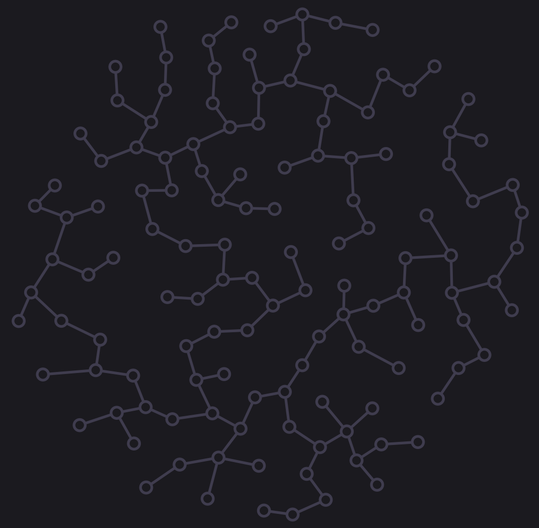


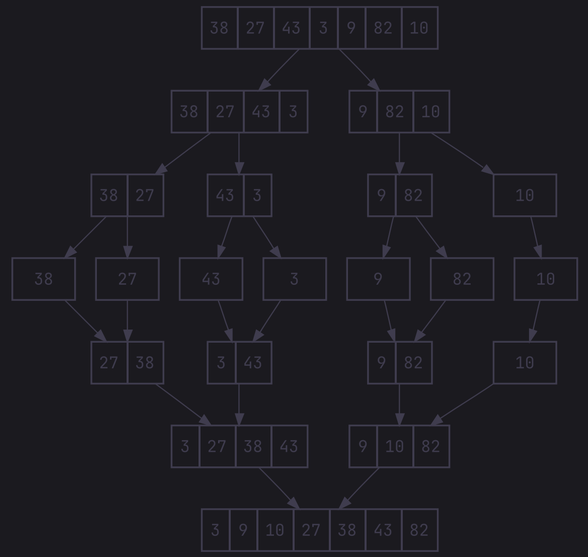


data structures & algorithms

Tutorial 7

(Week 8)





Burning questions from last week?

This week’s lab

Queues

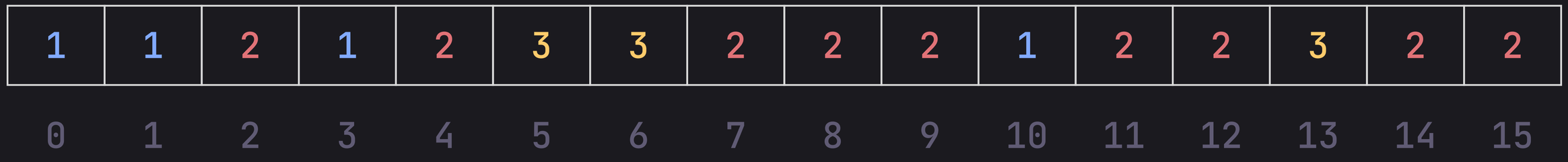
- Majority Element
- Priority Queue
- kth Largest Element
- Min Heap

Majority Element

In this challenge you are given a vector of ints of size n. You are ***guaranteed*** that some element of the vector appears **more than n/2 times**. The goal is to return this majority element.

Majority Element

In this challenge you are given a vector of ints of size n. You are ***guaranteed*** that some element of the vector appears **more than n/2 times**. The goal is to return this majority element.



Majority Element

In this challenge you are given a vector of ints of size n. You are ***guaranteed*** that some element of the vector appears **more than n/2 times**. The goal is to return this majority element.

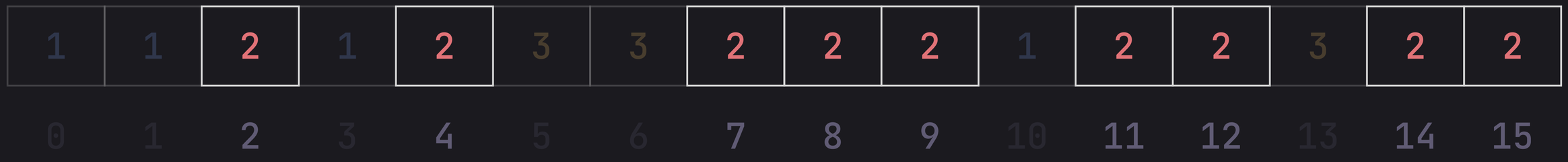
16 elements in total

1	1	2	1	2	3	3	2	2	2	1	2	2	3	2	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Majority Element

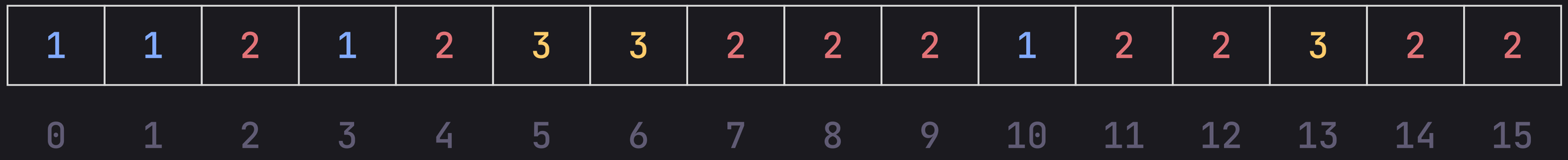
In this challenge you are given a vector of ints of size n. You are ***guaranteed*** that some element of the vector appears **more than n/2 times**. The goal is to return this majority element.

9 elements are equal to **2**



Majority Element

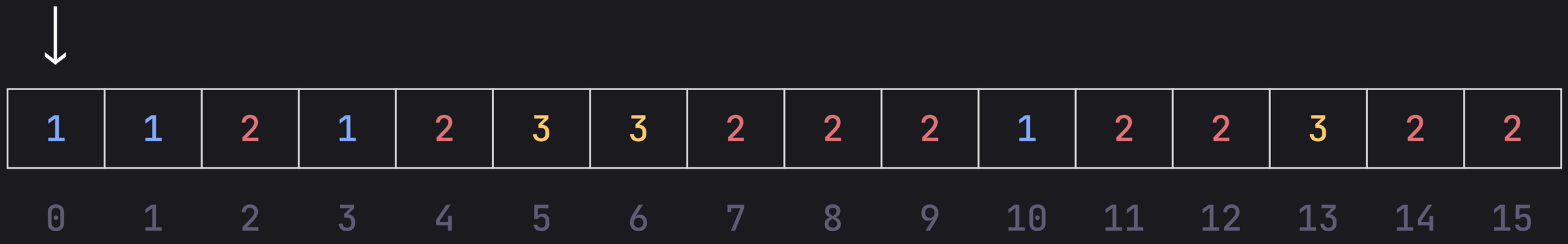
How can we use a
hash map?



Majority Element

Hashmap

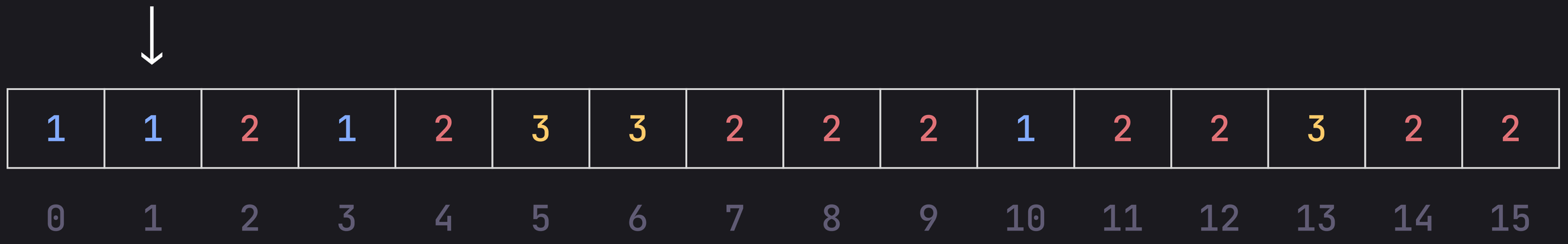
value	count
1	1



Majority Element

Hashmap

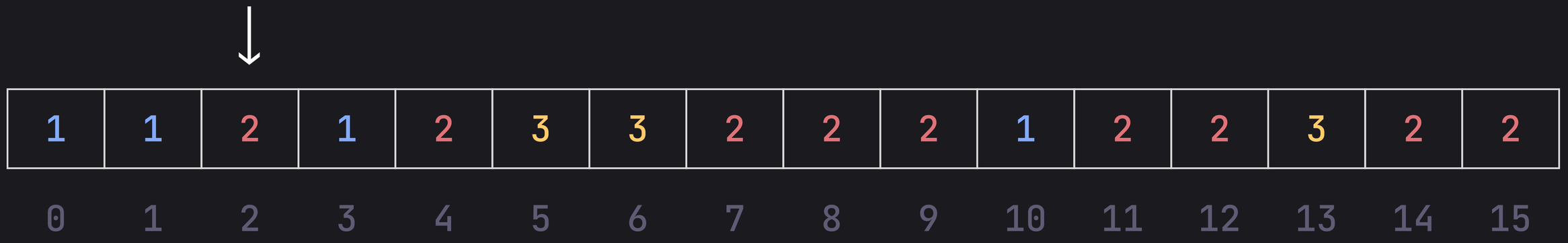
value	count
1	2



Majority Element

Hashmap

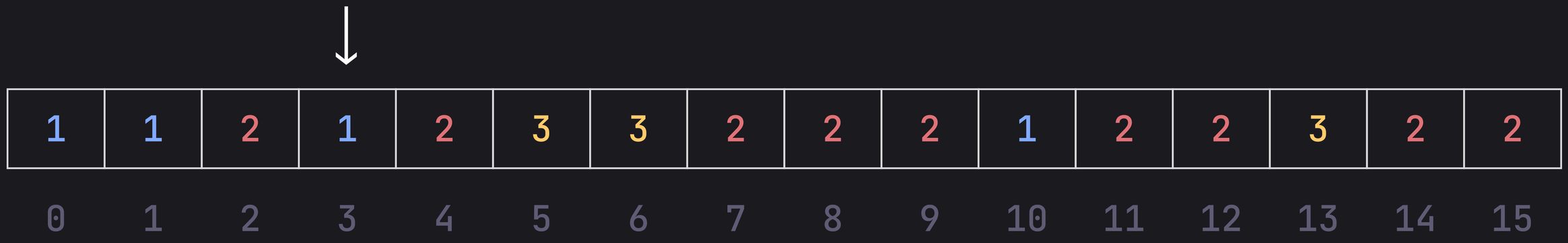
value	count
1	2
2	1



Majority Element

Hashmap

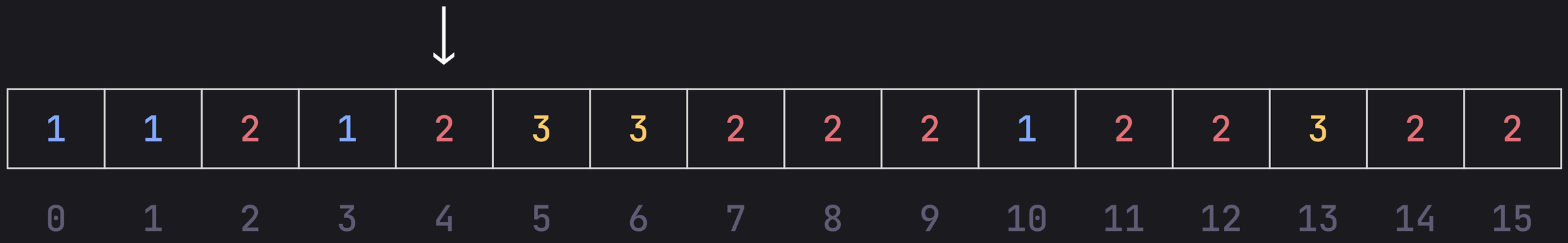
value	count
1	3
2	1



Majority Element

Hashmap

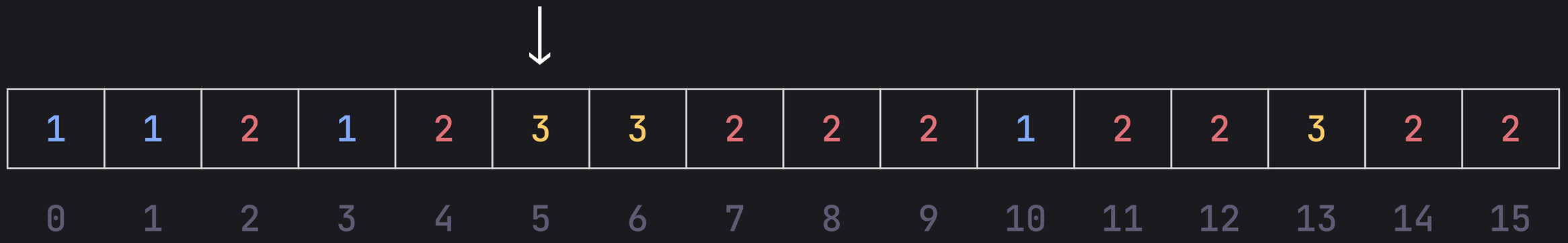
value	count
1	3
2	2



Majority Element

Hashmap

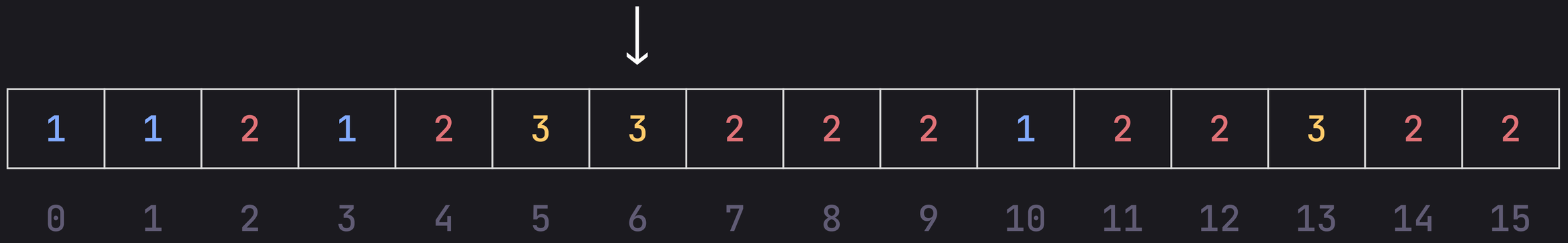
value	count
1	3
2	2
3	1



Majority Element

Hashmap

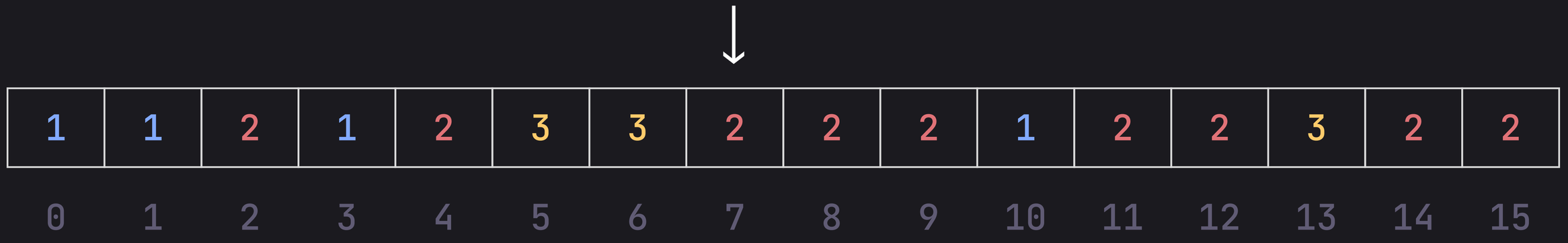
value	count
1	3
2	2
3	2



Majority Element

Hashmap

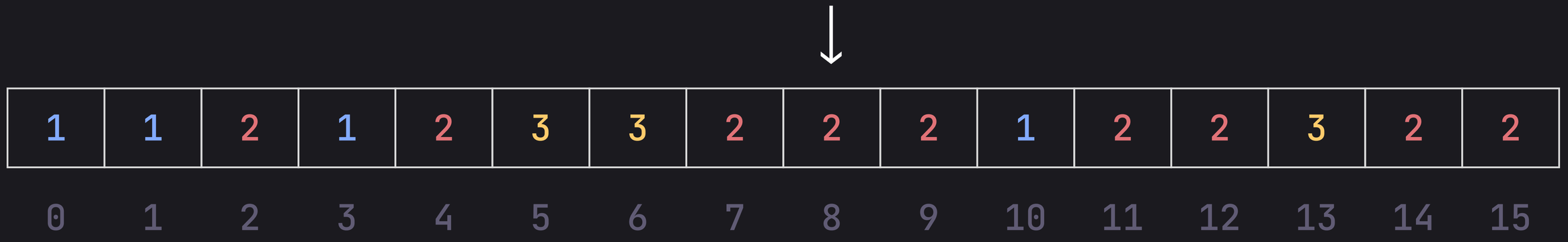
value	count
1	3
2	3
3	2



Majority Element

Hashmap

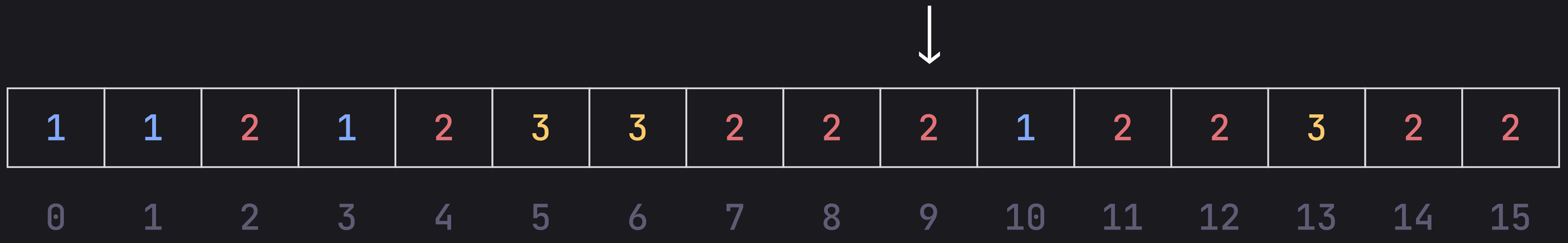
value	count
1	3
2	4
3	2



Majority Element

Hashmap

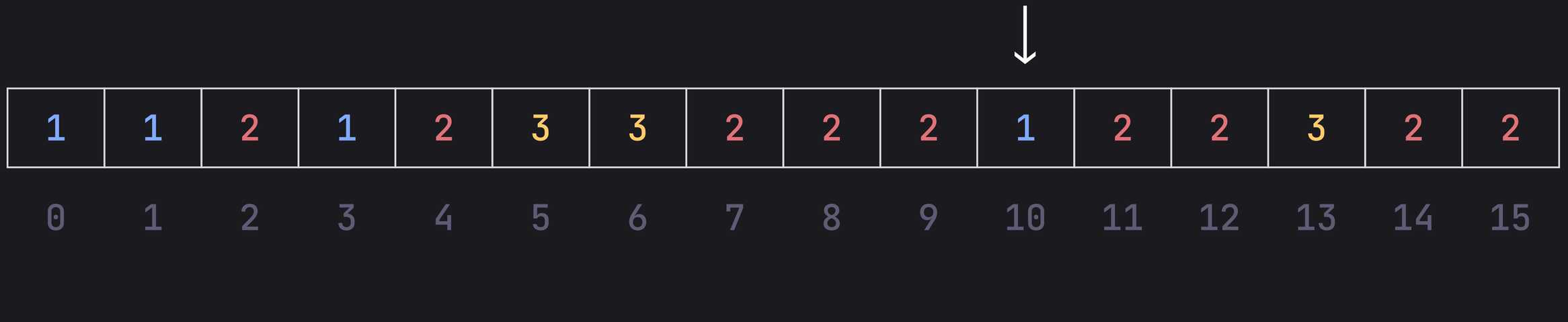
value	count
1	3
2	5
3	2



Majority Element

Hashmap

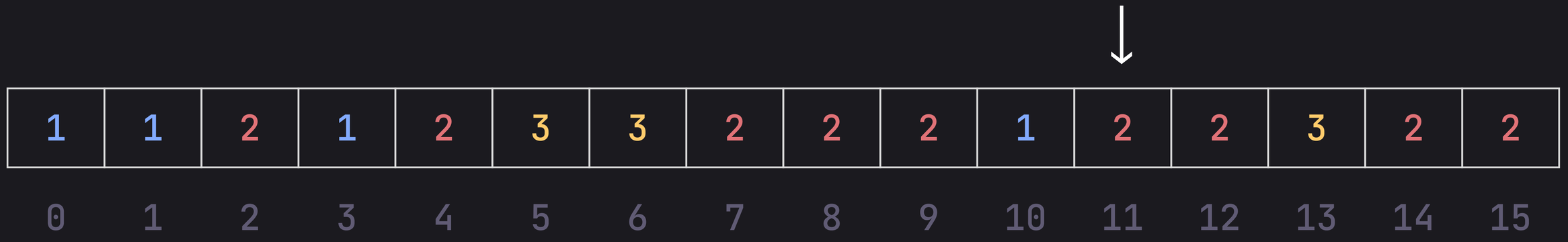
value	count
1	4
2	5
3	2



Majority Element

Hashmap

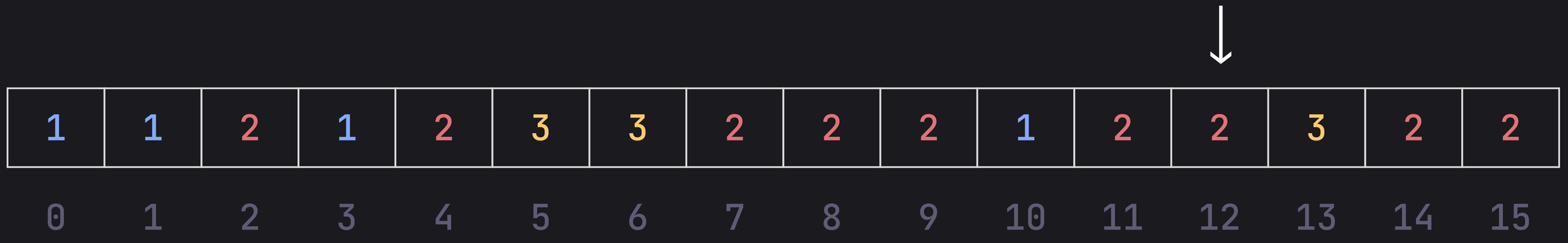
value	count
1	4
2	6
3	2



Majority Element

Hashmap

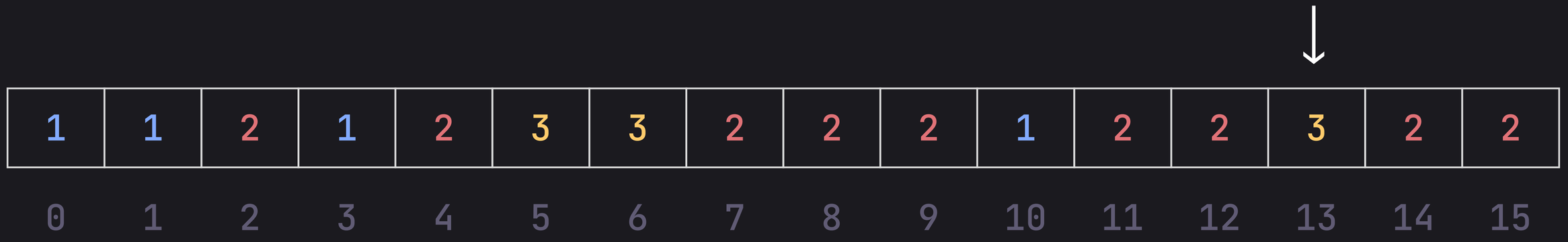
value	count
1	4
2	7
3	2



Majority Element

Hashmap

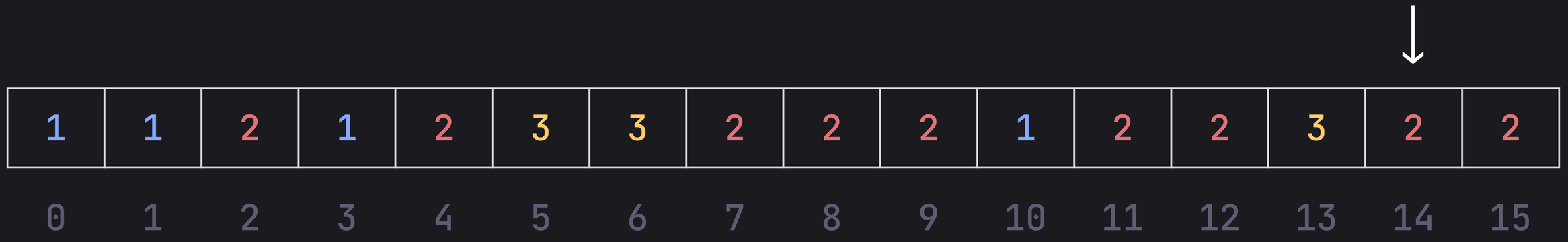
value	count
1	4
2	7
3	3



Majority Element

Hashmap

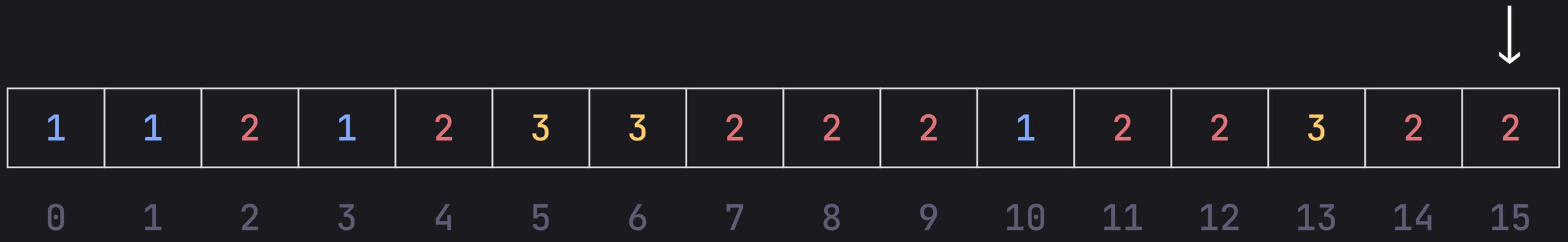
value	count
1	4
2	8
3	3



Majority Element

Hashmap

value	count
1	4
2	9
3	3



Majority Element

Hashmap

value	count
1	4
2	9
3	3

Majority Element

Hashmap

value	count
1	4
2	9
3	3

Now all we need to do is return the value
with the **highest count**

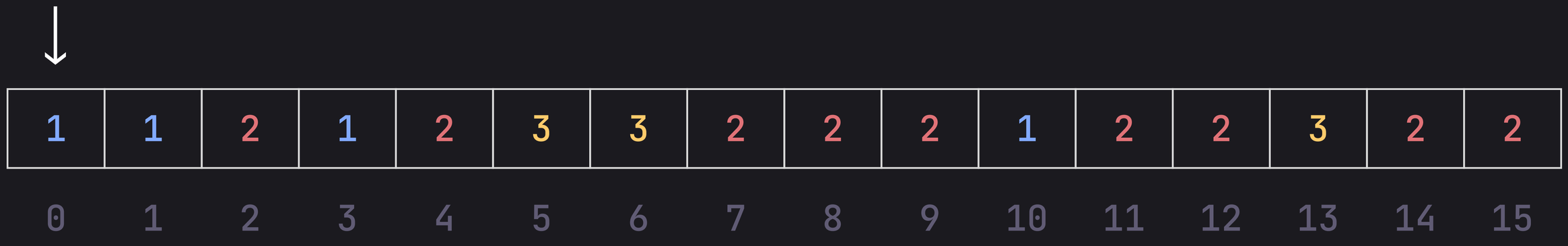
Majority Element

What is the **time complexity?**

Majority Element

What is the **time complexity?**

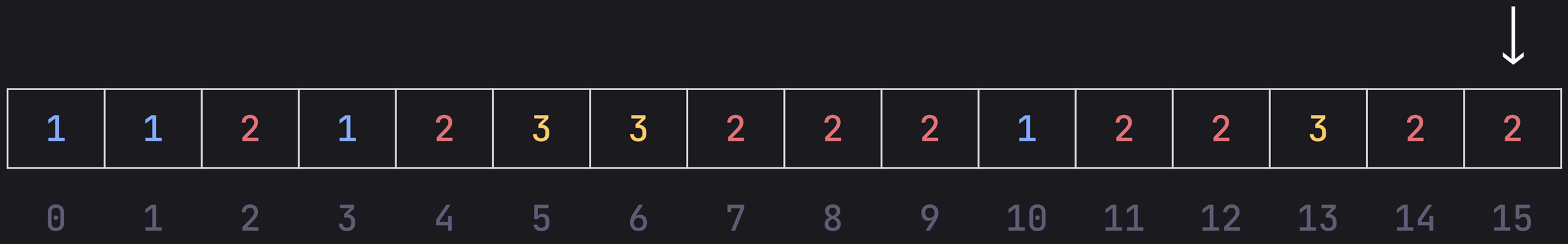
1. First we iterate through n elements



Majority Element

What is the **time complexity?**

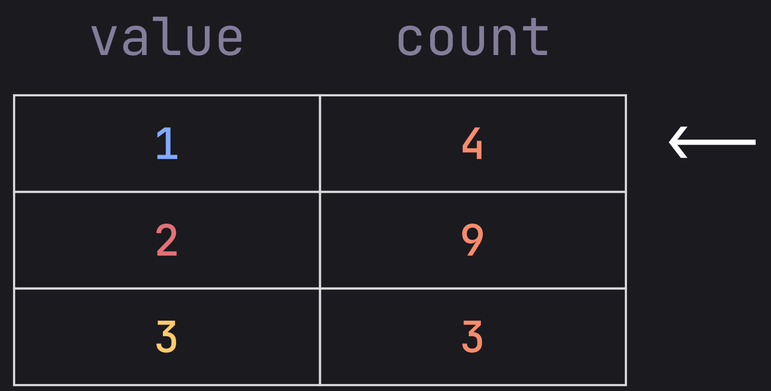
1. First we iterate through n elements



Majority Element

What is the **time complexity?**

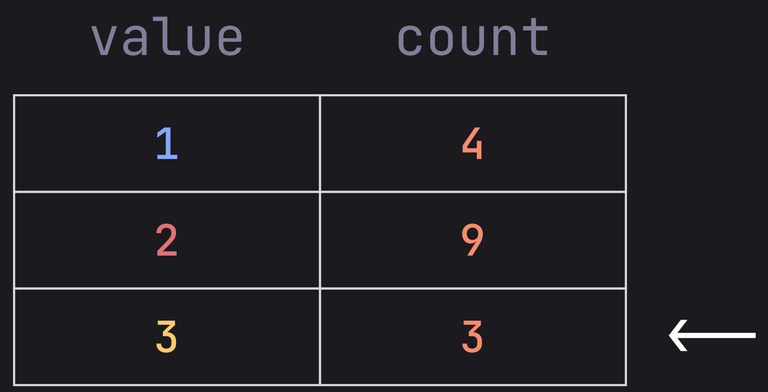
1. First we iterate through n elements
2. Then we scan though the hashmap (worst case ~n/2 elements)



Majority Element

What is the **time complexity?**

1. First we iterate through n elements
2. Then we scan though the hashmap (worst case n/2 elements)



Majority Element

What is the **time complexity?**

1. First we iterate through n elements
2. Then we scan though the hashmap (worst case n/2 elements)

$$O(n)$$

Majority Element

What is the **space complexity?**

Majority Element

What is the **space complexity?**

We store a count for all the elements, and in the worst case there are n/2 distinct elements

value	count
1	4
2	9
3	3

Majority Element

What is the **space complexity?**

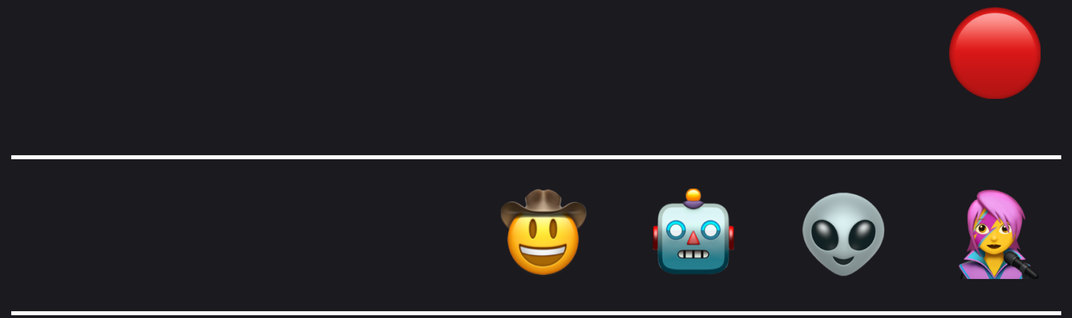
We store a count for all the elements, and in the worst case there are n/2 distinct elements

$$O(n)$$

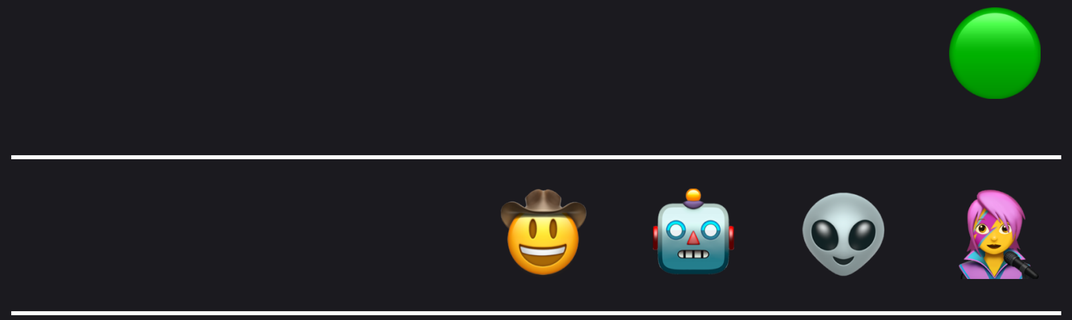
Majority Element

```
function majorityElement(vec):
  counts = new HashTable()
  for x in vec:
    if x not in counts:
      counts[x] = 0
    counts[x] += 1
  for [num, count] in counts:
    if count > vec.size() / 2:
      return num
  // some default value
  return 0
```

Queue



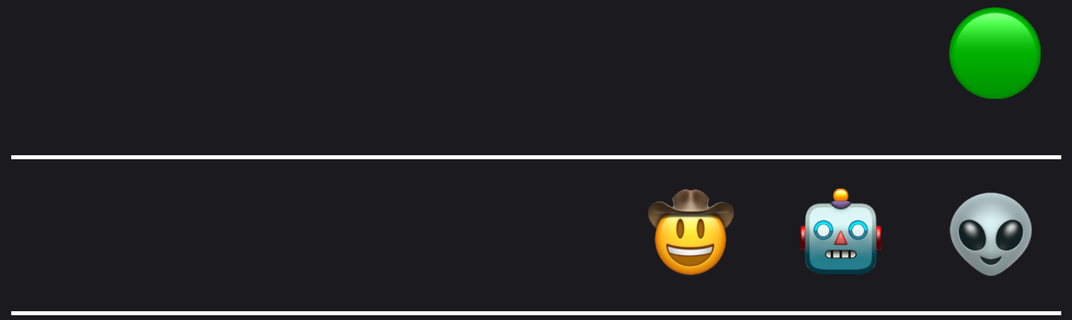
Queue



```
queue.pop()
```

Dequeue

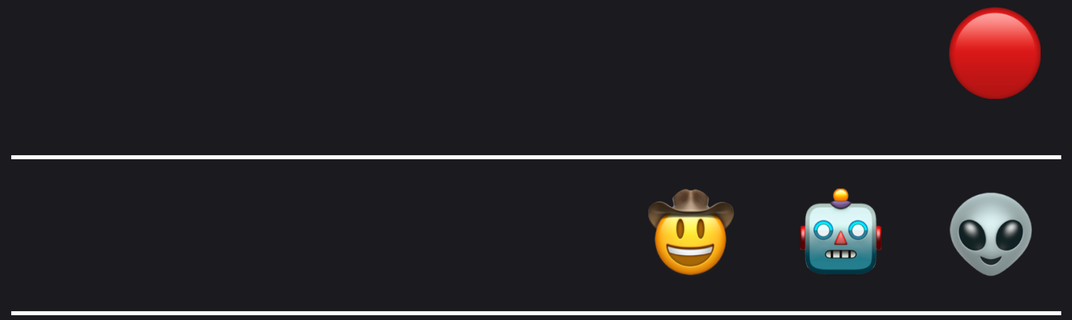
Queue



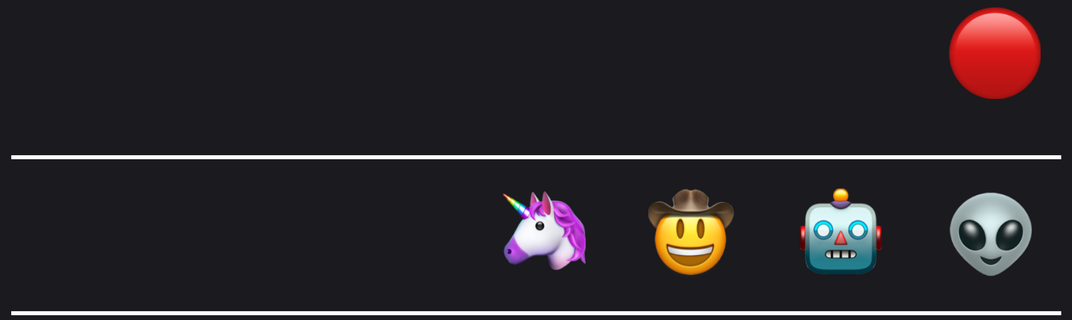
```
queue.pop()
```

Dequeue

Queue



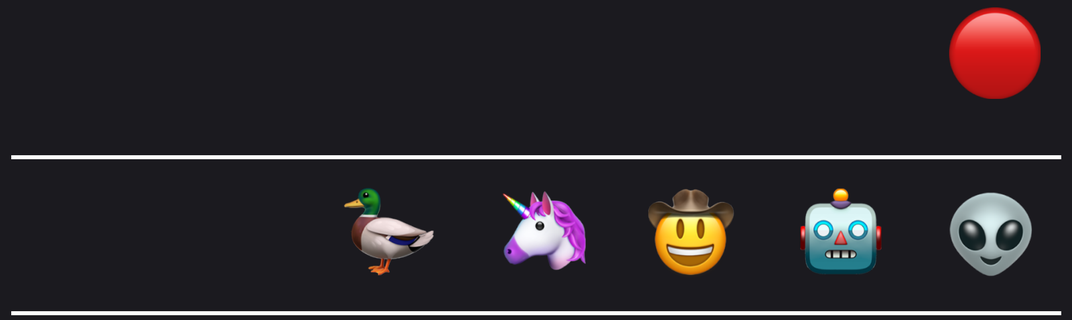
Queue



```
queue.push()
```

Enqueue

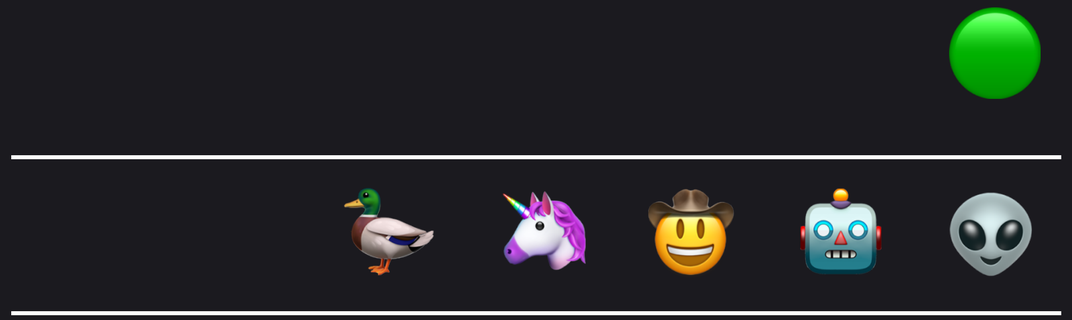
Queue



`queue.push()`

Enqueue

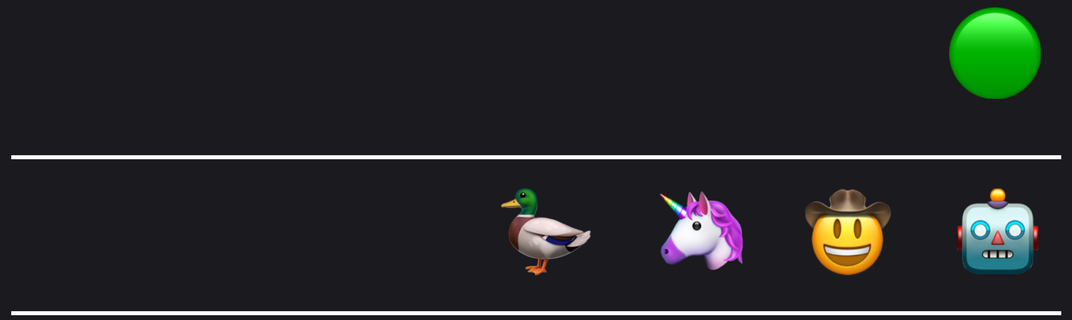
Queue



```
queue.pop()
```

Dequeue

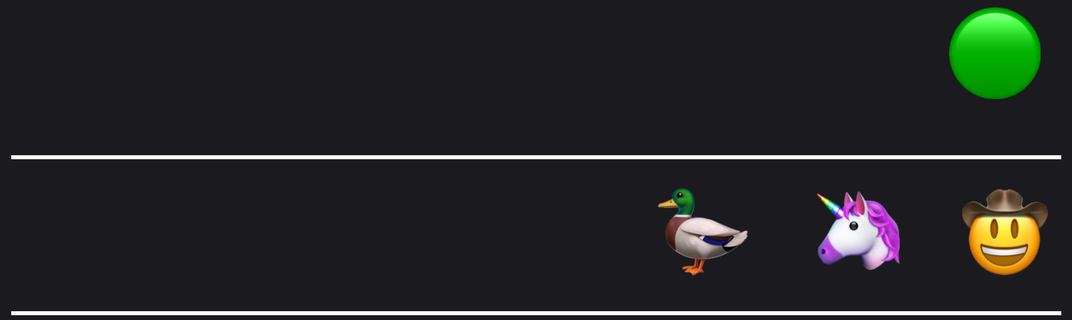
Queue



```
queue.pop()
```

Dequeue

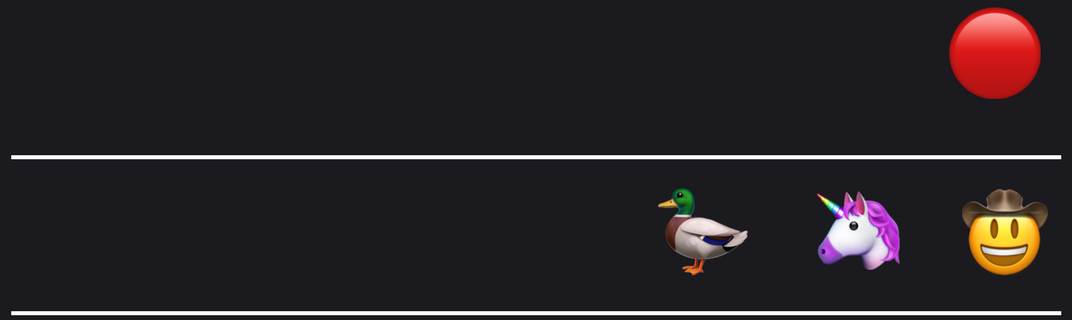
Queue



```
queue.pop()
```

Dequeue

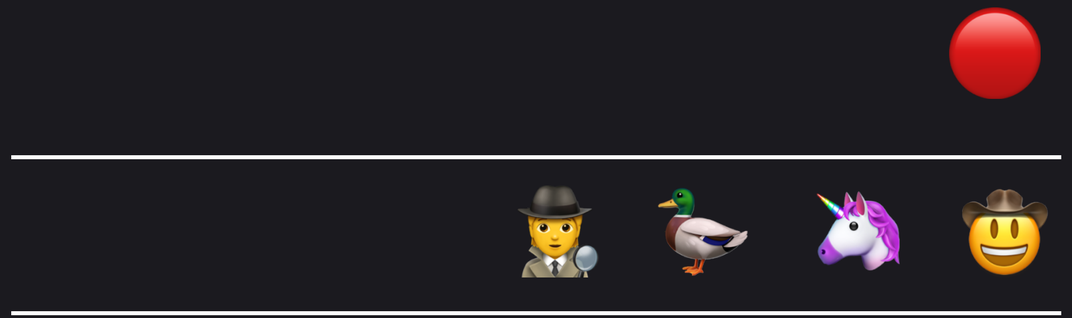
Queue



`queue.push()`

Enqueue

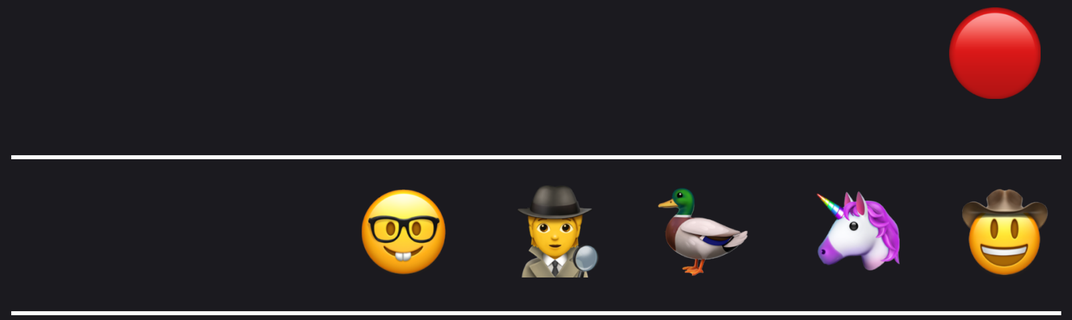
Queue



```
queue.push()
```

Enqueue

Queue



`queue.push()`

Enqueue

Queue

FIFO

First **I**n **F**irst **O**ut

Priority Queue

But sometimes we want a queue
where some people get **priority**.
Like at the emergency room

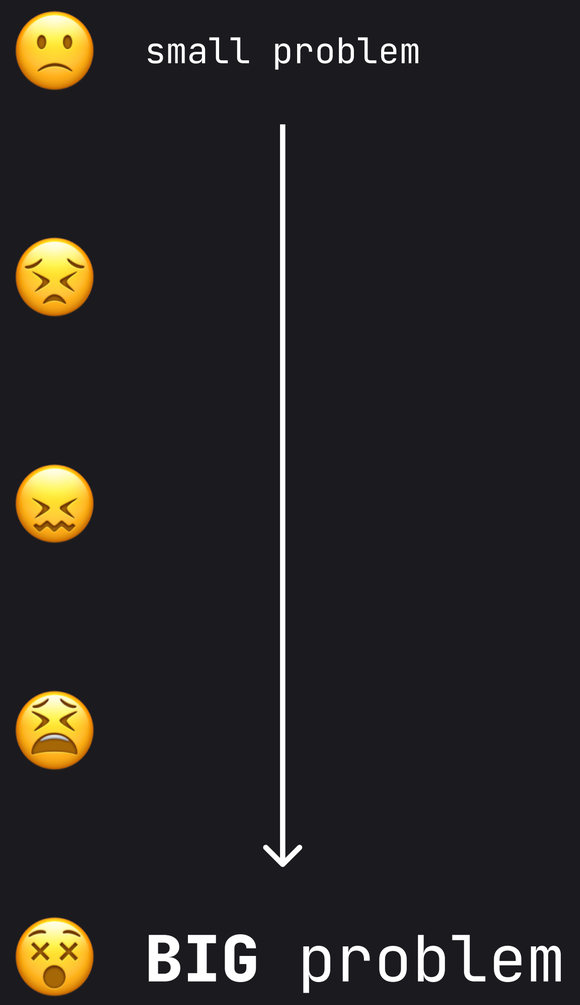
Priority Queue

small problem

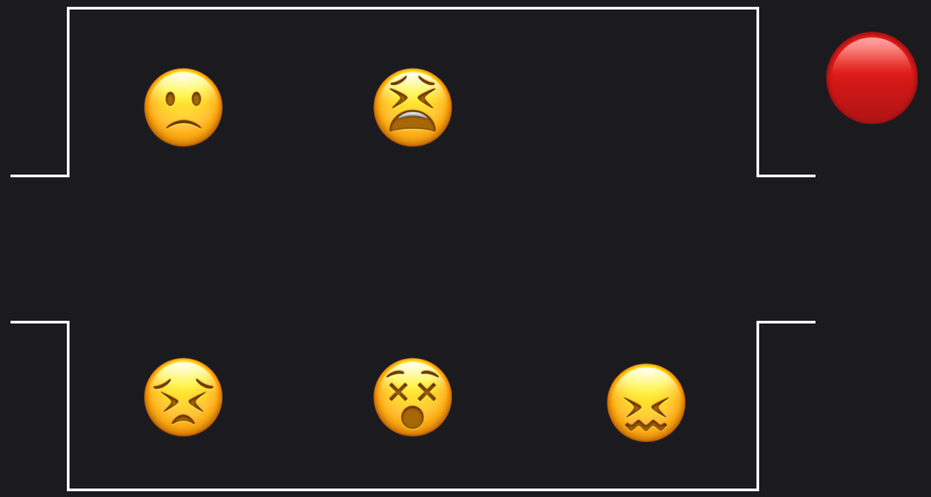
But sometimes we want a queue
where some people get **priority**.
Like at the emergency room

Priority Queue

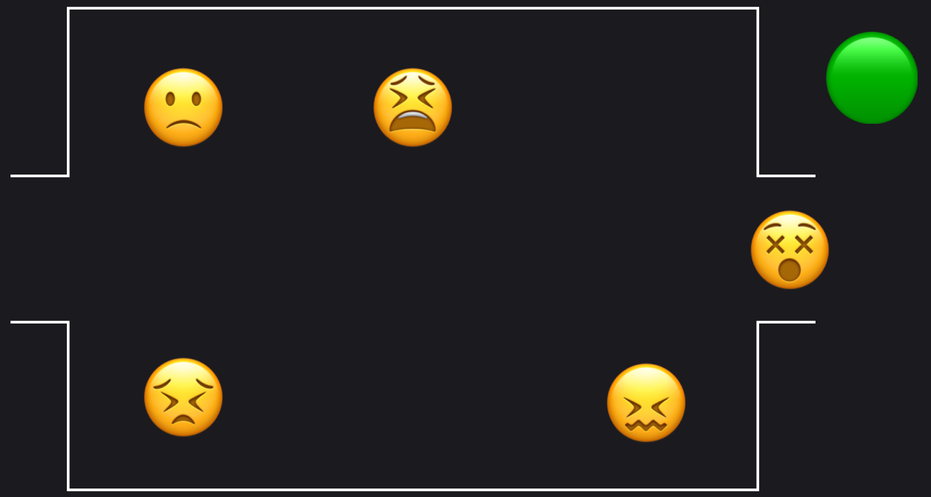
But sometimes we want a queue
where some people get **priority**.
Like at the emergency room



Priority Queue

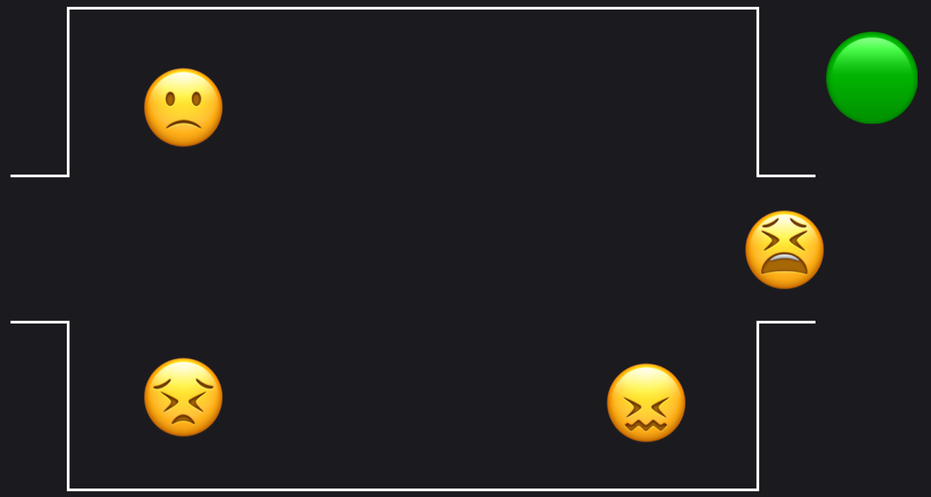


Priority Queue



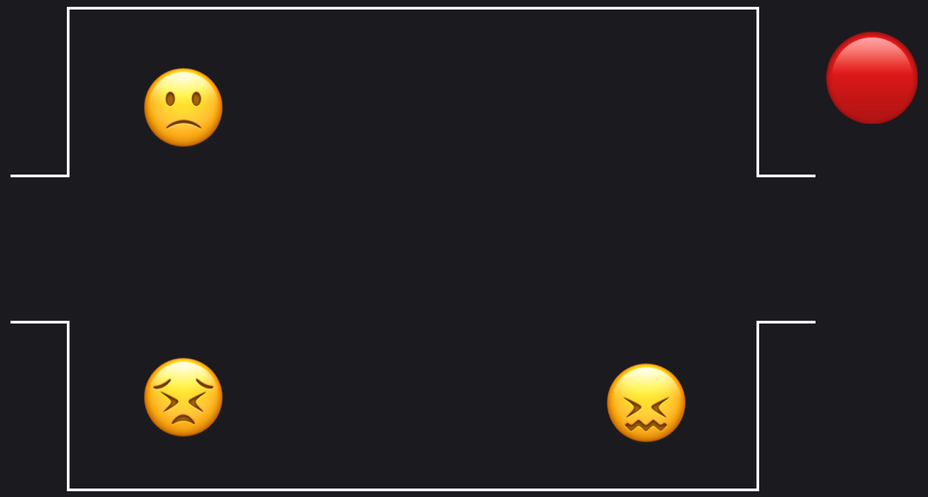
The person with the most urgent problem gets treated first

Priority Queue

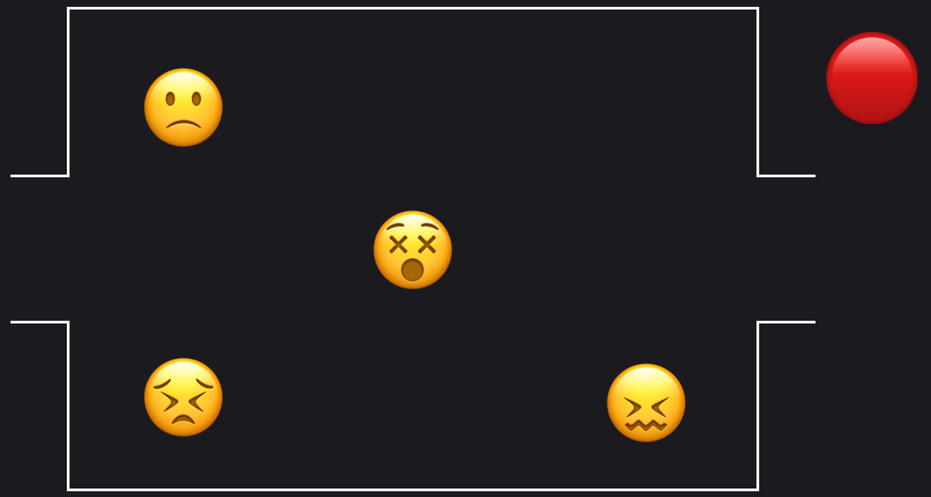


The person with the most urgent problem gets treated first

Priority Queue

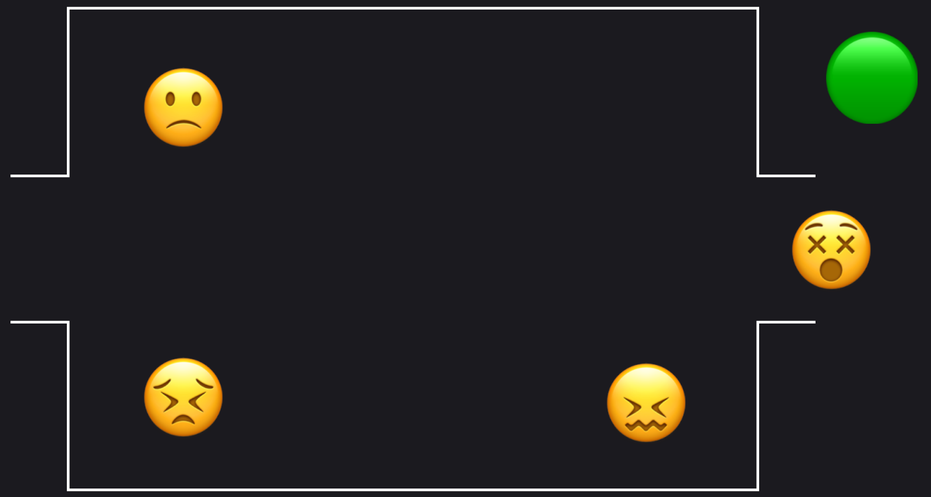


Priority Queue



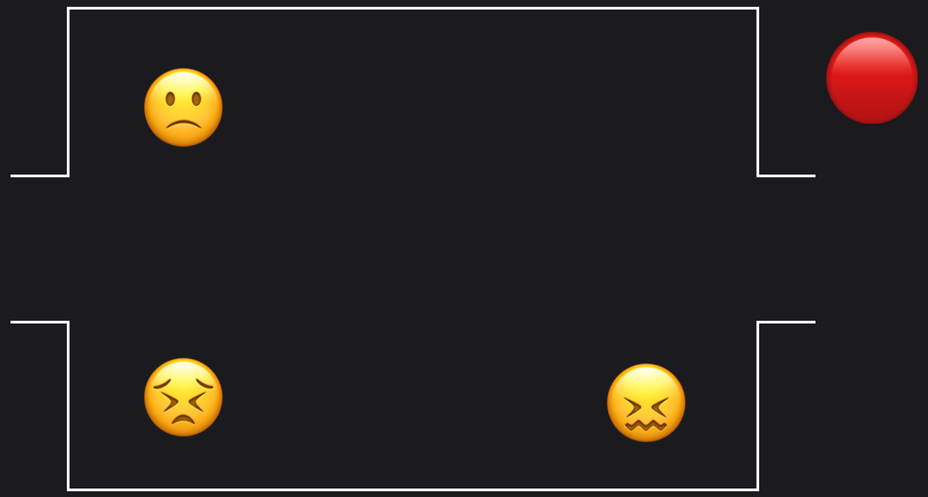
Suppose another patient just arrives
and they urgently need help

Priority Queue



They would move up to the top of the priority list

Priority Queue



Priority Queue

```
std::priority_queue<int> queue {};
```

Priority Queue

```cpp
std::priority_queue<int> queue {};

  queue.push(3) // add 3 to the queue
  queue.pop()   // remove the highest priority element
  queue.top()   // look at the highest priority element
```

Priority Queue

```
std::priority_queue<int> queue {};

  queue.push(3) // O(log(n)) - logarithmic time
  queue.pop()   // O(log(n)) - logarithmic time
  queue.top()   // O(1) - Constant time
```

Priority Queue

The default queue is a max priority queue
but we can make a min priority queue like so

```cpp
std::priority_queue<int, std::vector<int>, std::greater<int>> queue {};
```

Priority Queue

The default queue is a max priority queue
but we can make a min priority queue like so

```
std::priority_queue<int, std::vector<int>, std::greater<int>> queue {};
```

type of objects in the queue

Priority Queue

The default queue is a max priority queue
but we can make a min priority queue like so

```
std::priority_queue<int, std::vector<int>, std::greater<int>> queue {};
```

what kind of underlying container
to use for the priority queue

Priority Queue

The default queue is a max priority queue
but we can make a min priority queue like so

```
std::priority_queue<int, std::vector<int>, std::greater<int>> queue {};
```

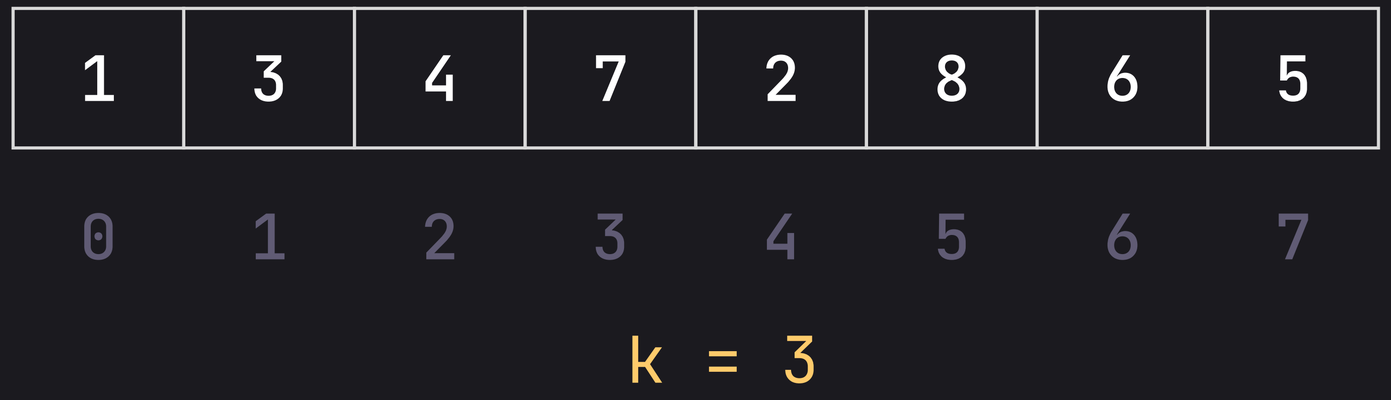
The method we use for ordering the queue
(first item has the highest priority)

Kth Largest Element

In this challenge we look at an application of a priority queue:
Finding the **Kth largest** element of a vector.

Kth Largest Element

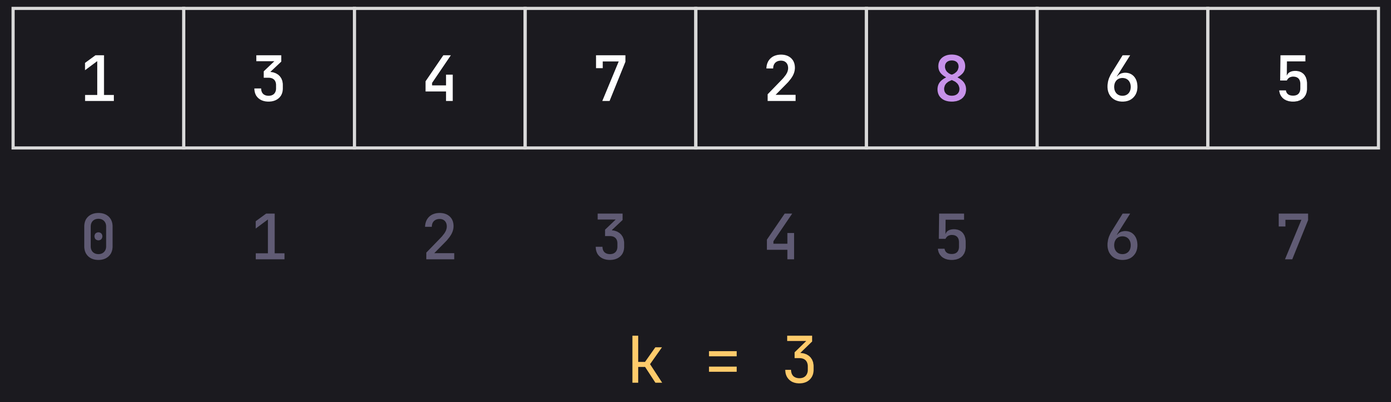
In this challenge we look at an application of a priority queue:
Finding the **Kth largest** element of a vector.



Kth Largest Element

In this challenge we look at an application of a priority queue:
Finding the **Kth largest** element of a vector.

1st largest = 8

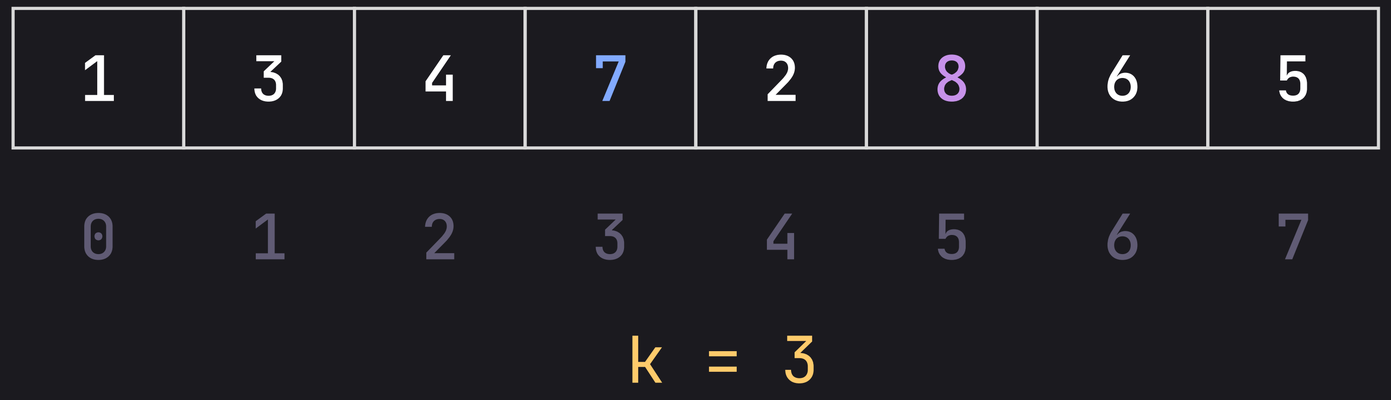


Kth Largest Element

In this challenge we look at an application of a priority queue:
Finding the **Kth largest** element of a vector.

1st largest = 8

2nd largest = 7



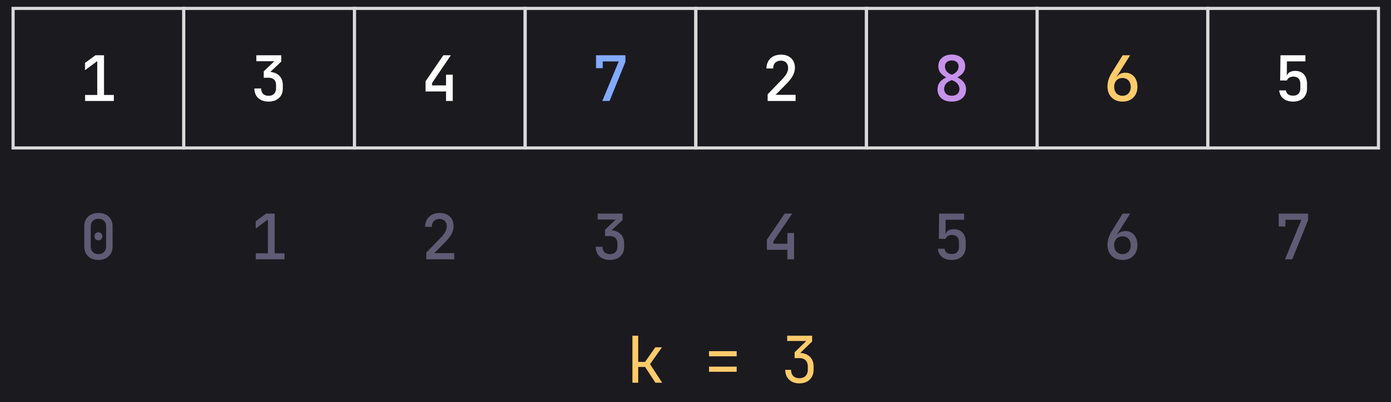
Kth Largest Element

In this challenge we look at an application of a priority queue:
Finding the **Kth largest** element of a vector.

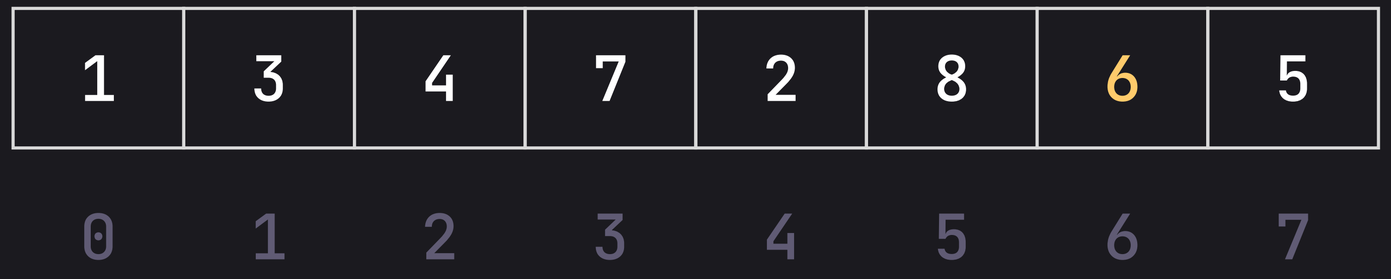
1st largest = 8

2nd largest = 7

3rd largest = 6

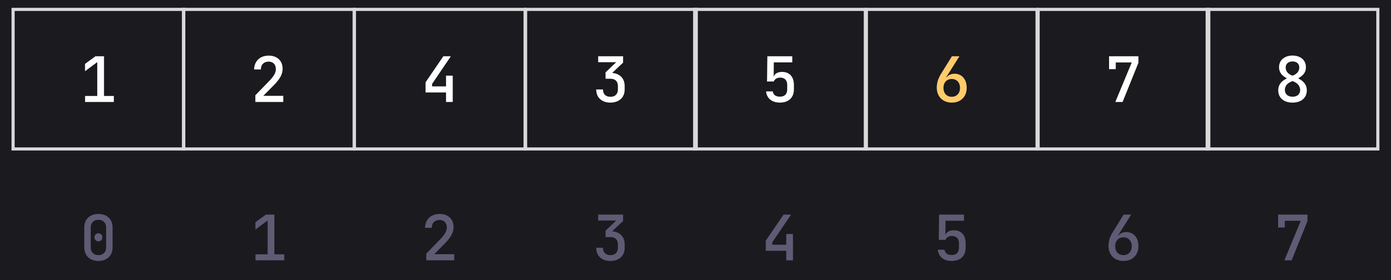


Kth Largest Element



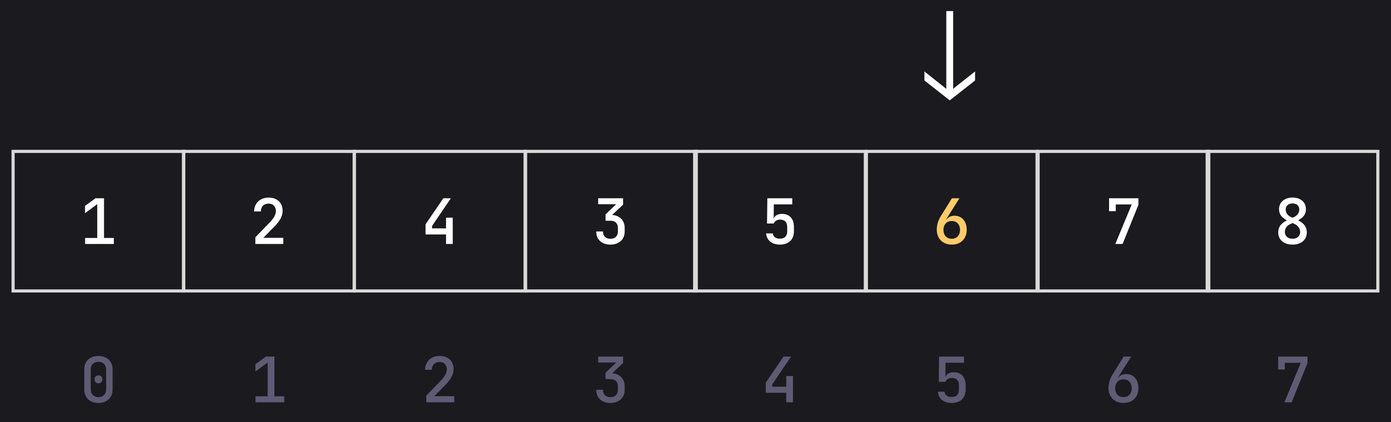
Kth Largest Element

1. Sort the vector O(n*log(n))



Kth Largest Element

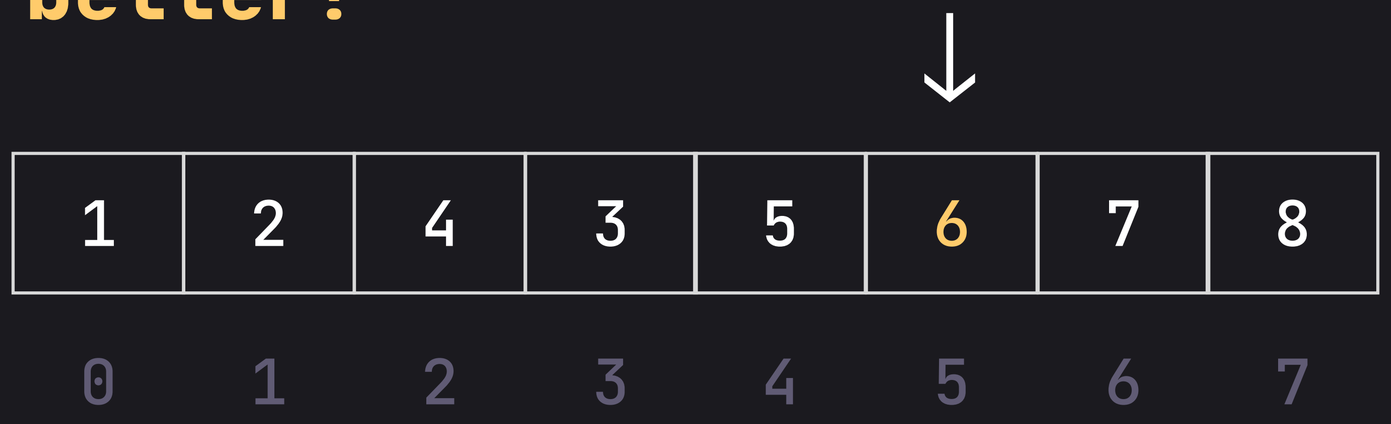
1. Sort the vector — O(n*log(n))
2. Select the 3rd last element — O(1)



Kth Largest Element

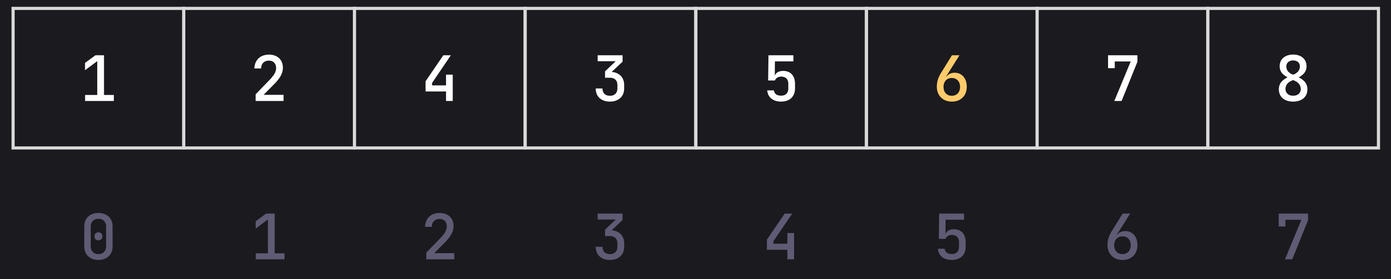
1. Sort the vector — O(n*log(n))
2. Select the 4th last element — O(1)

Can we do **better?**

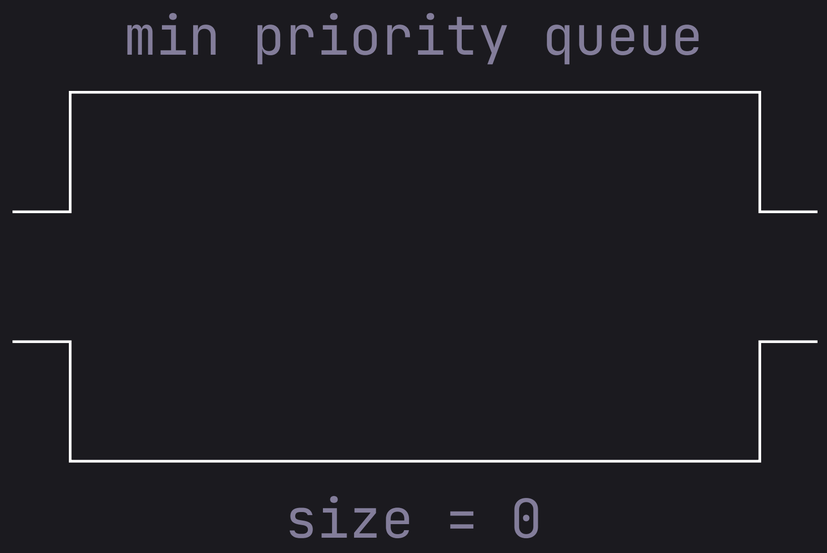


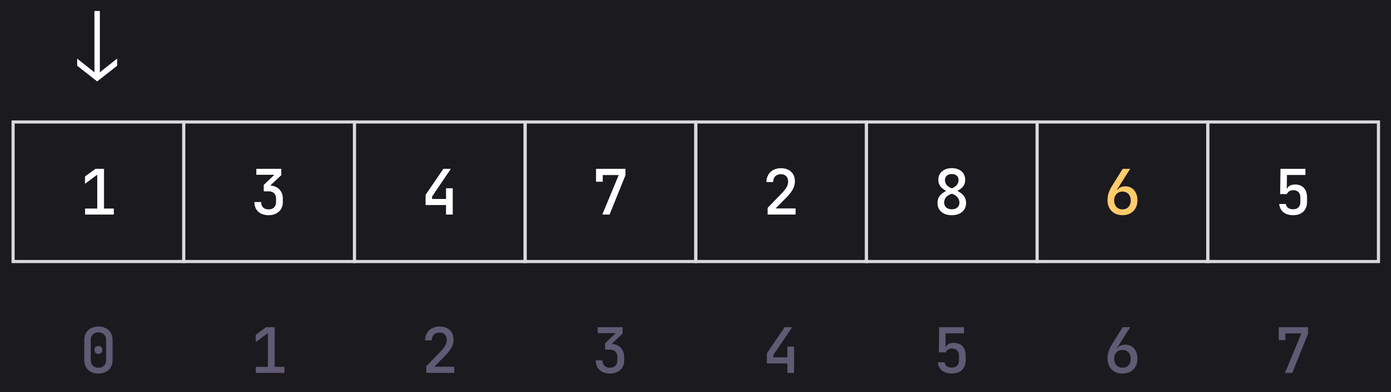
Kth Largest Element

1. Keep track of the k largest elements we have seen so far in a min priority queue
2. Every time we see a new element we add it the queue and then remove the smallest element from the queue

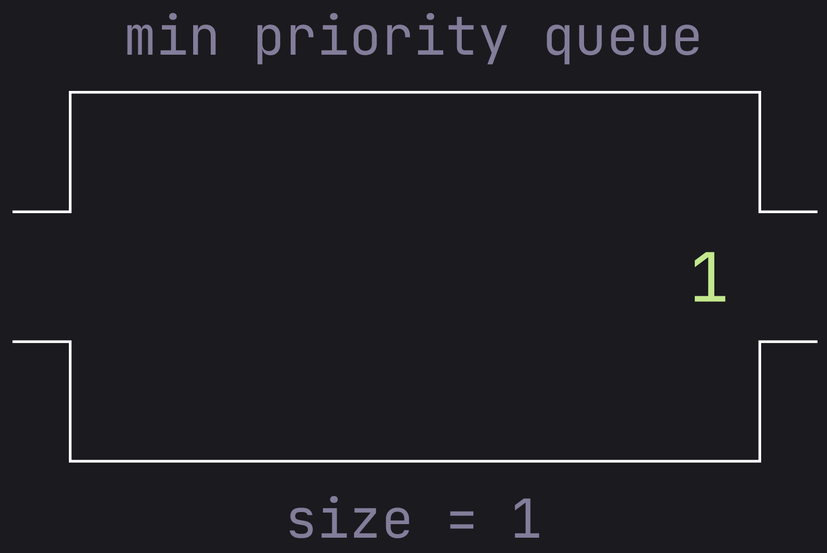


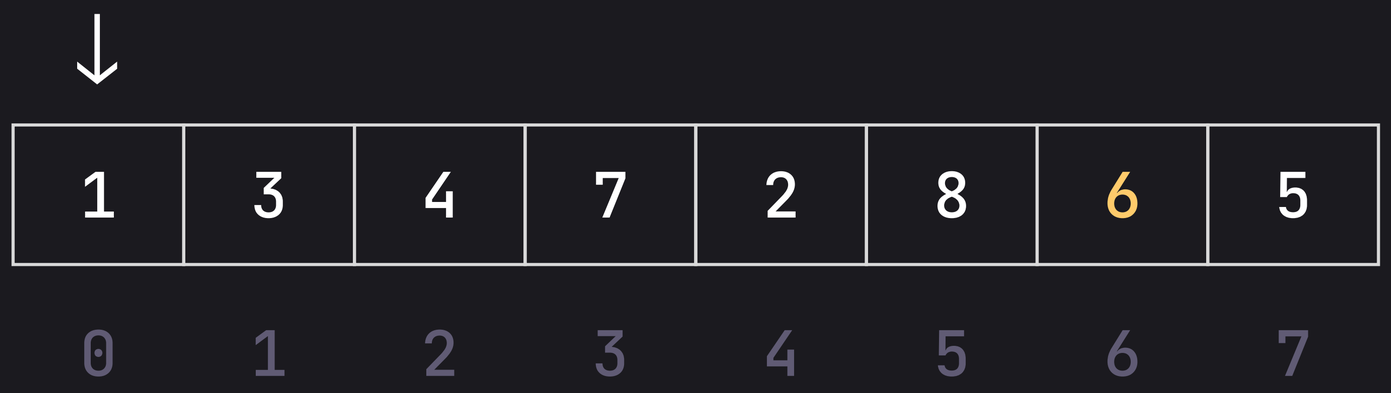
Kth Largest Element



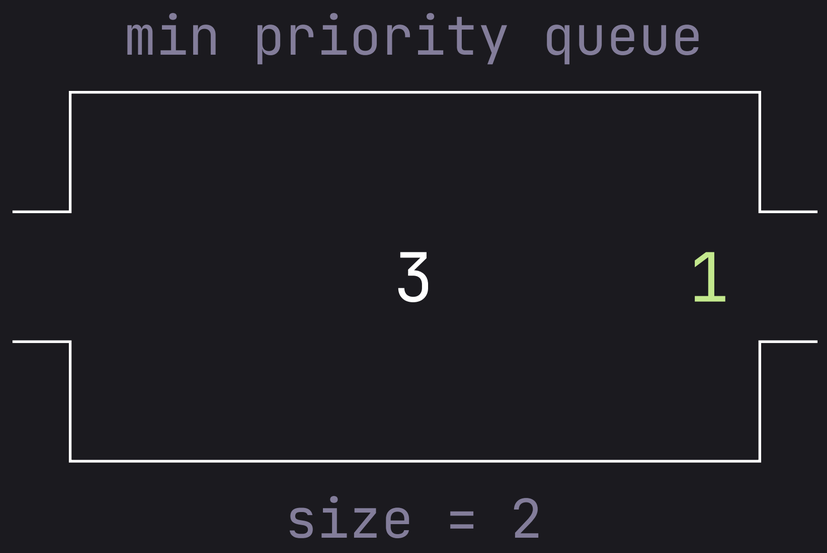


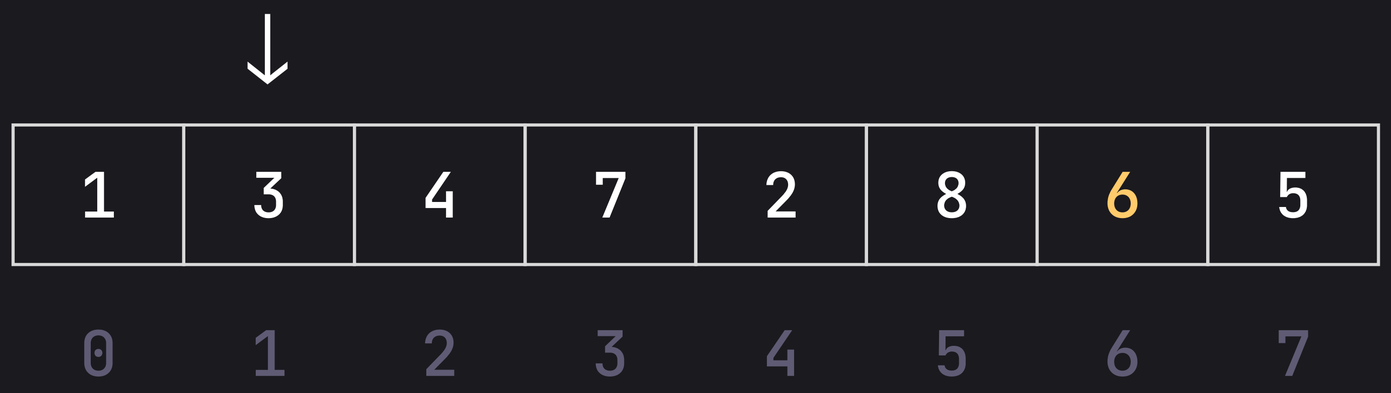
Kth Largest Element



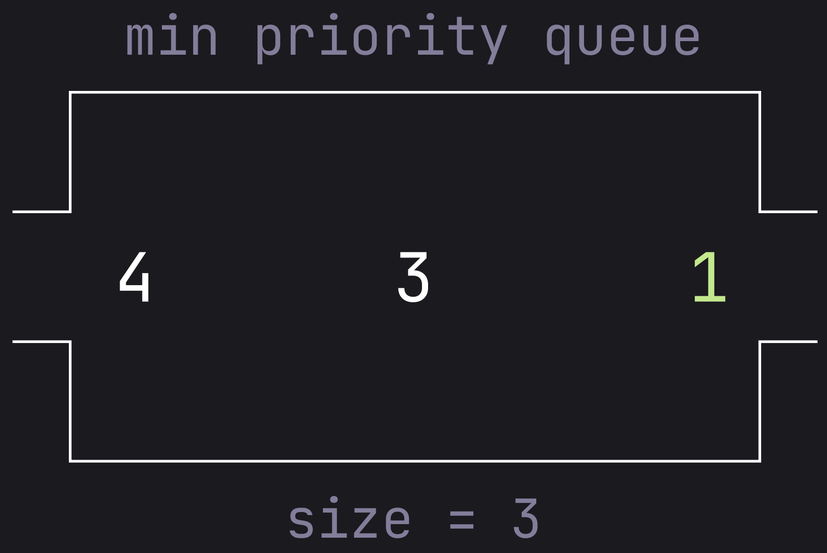


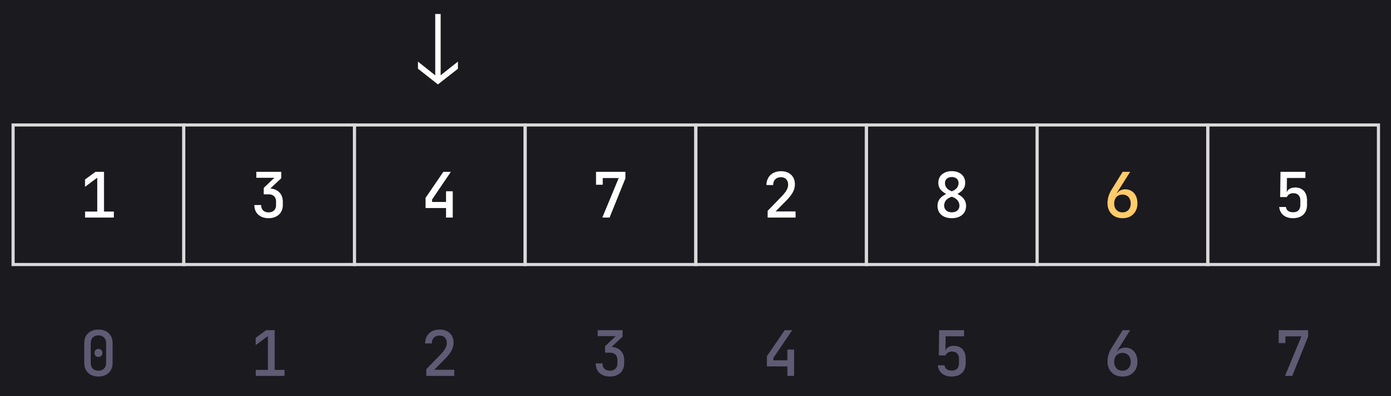
Kth Largest Element



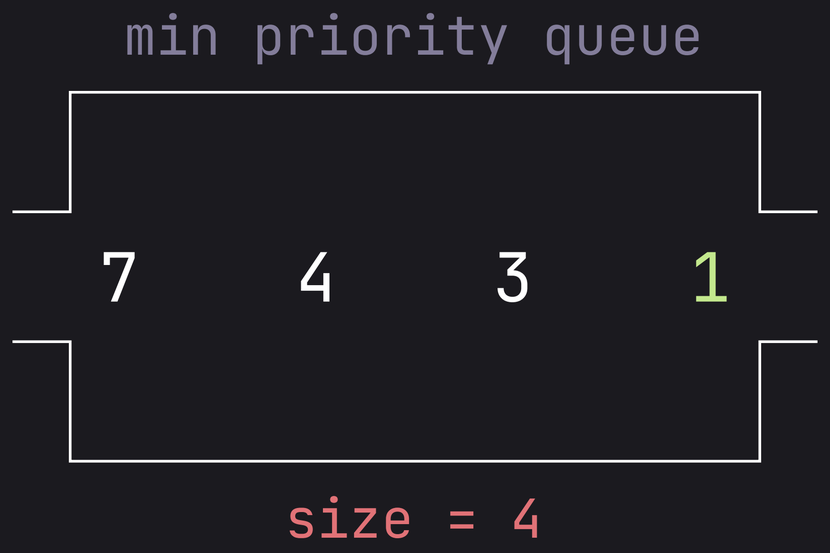


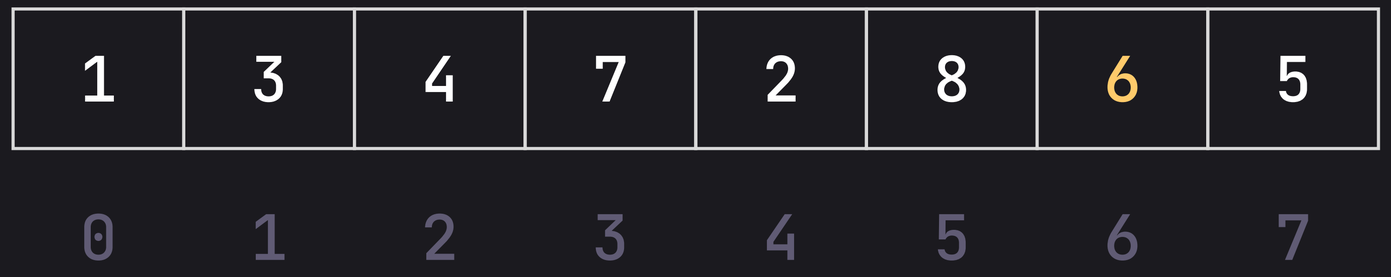
Kth Largest Element



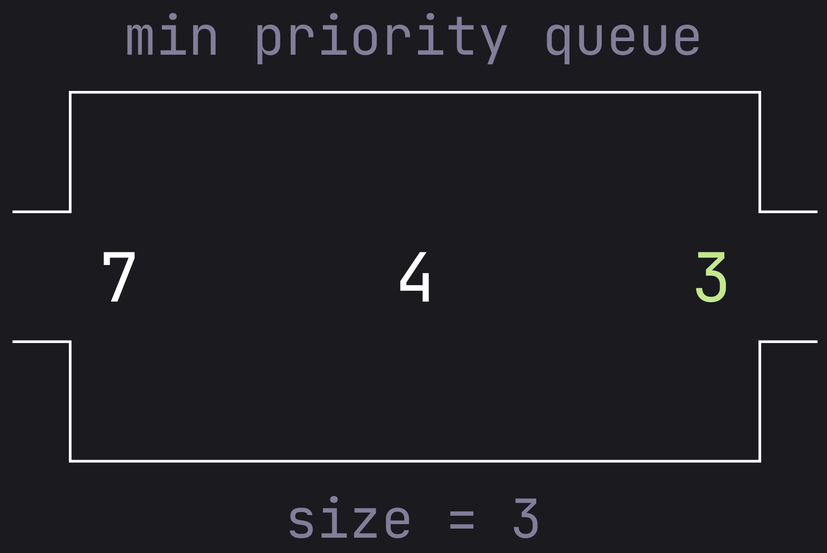


Kth Largest Element

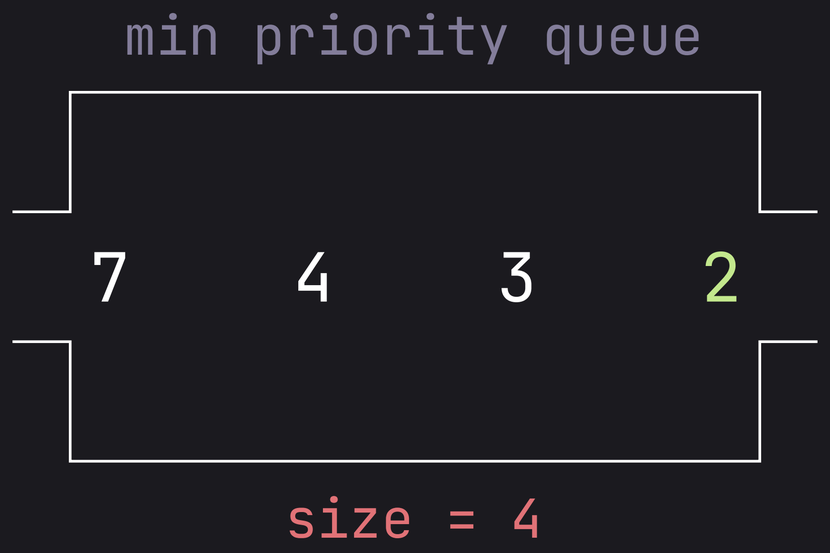


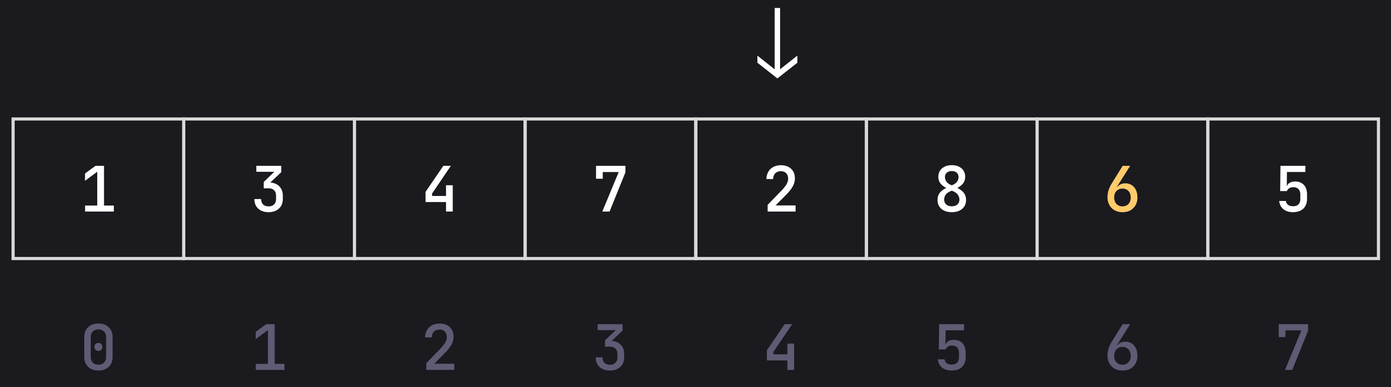


Kth Largest Element

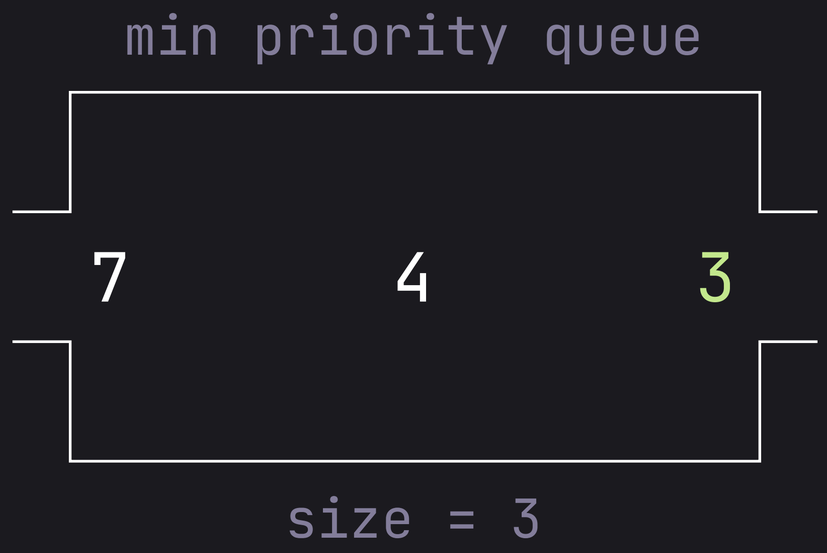


Kth Largest Element

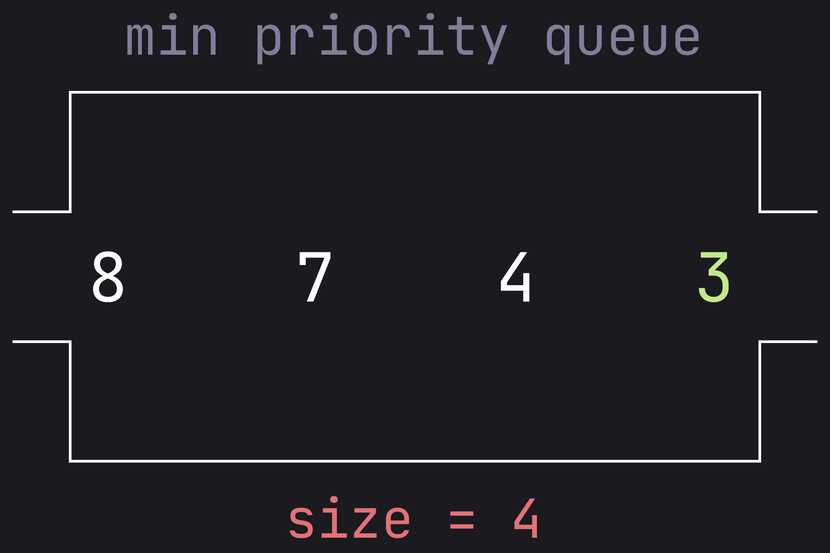


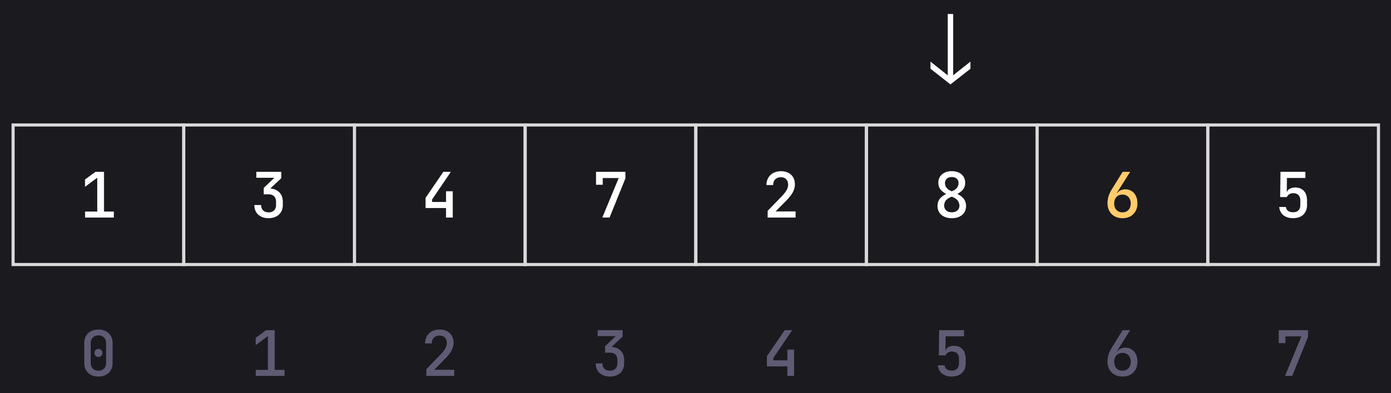


Kth Largest Element

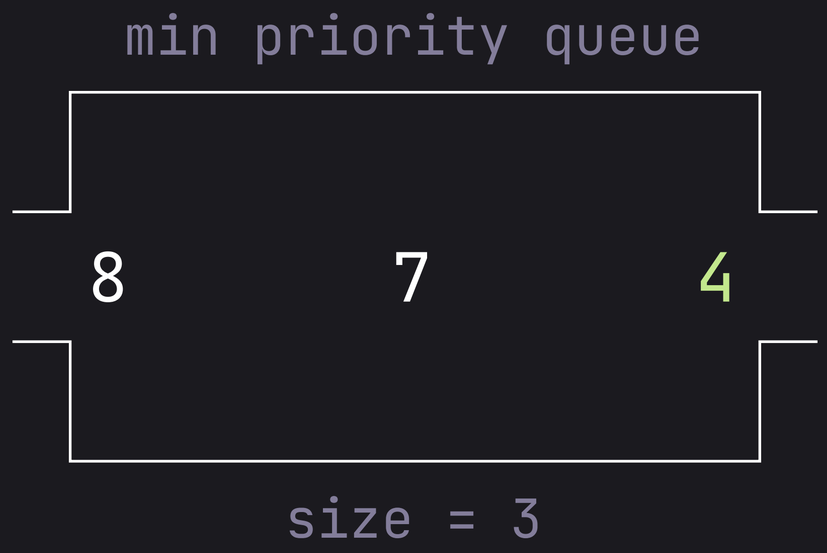


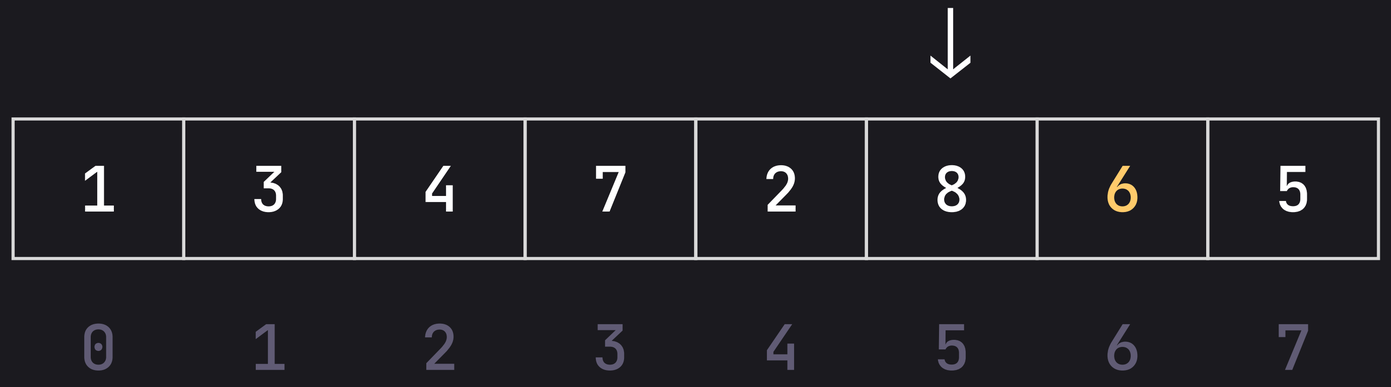
Kth Largest Element



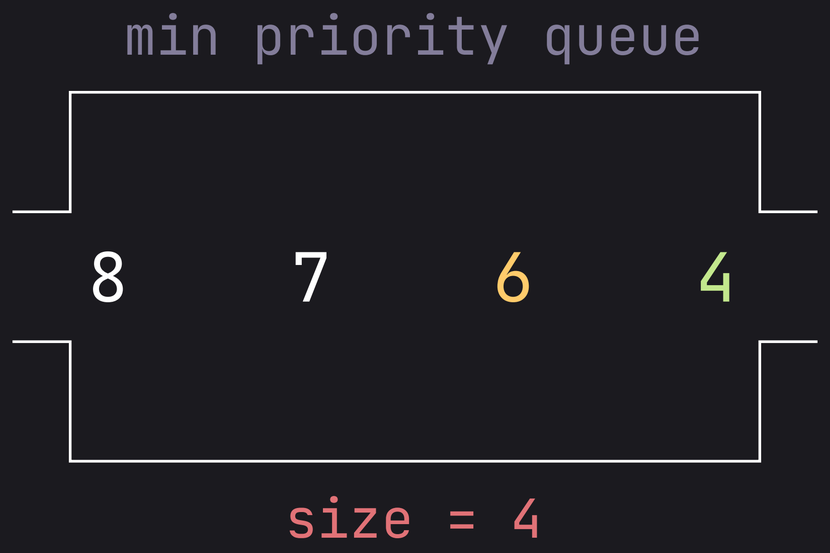


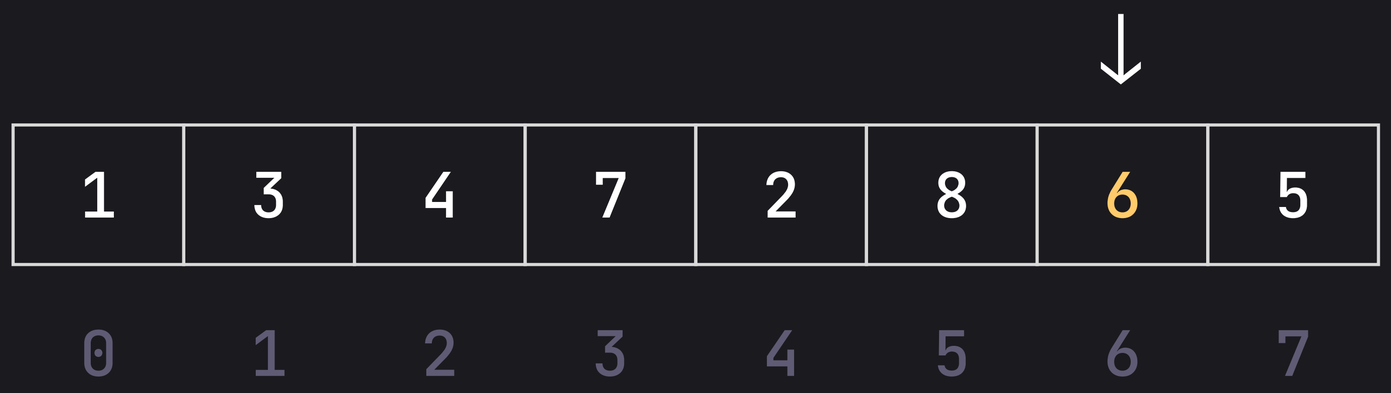
Kth Largest Element



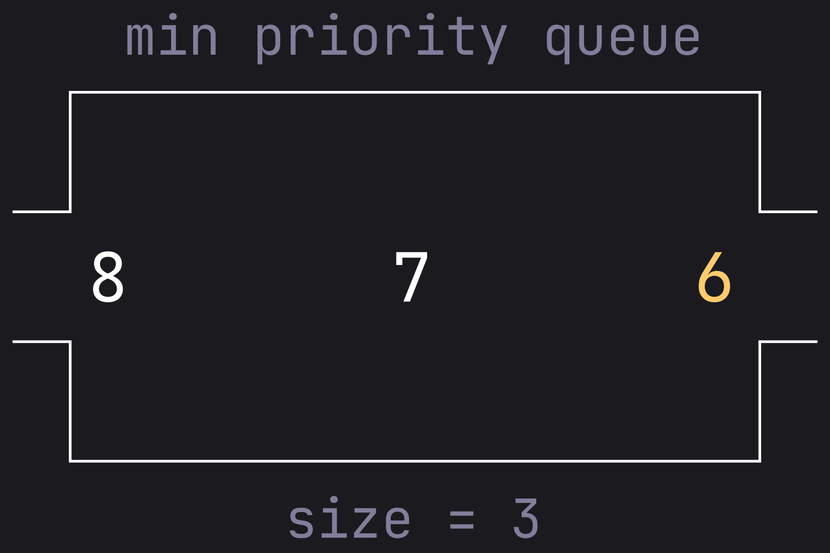


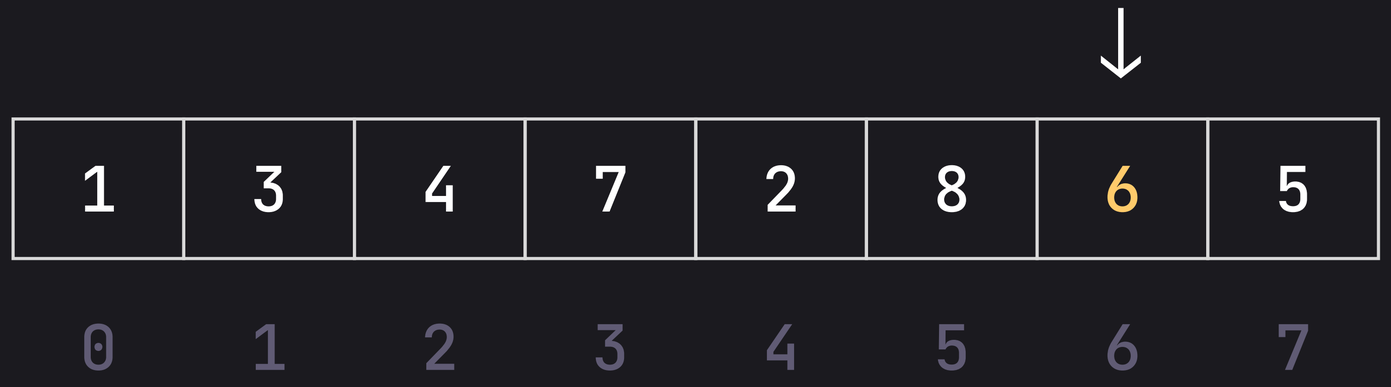
Kth Largest Element



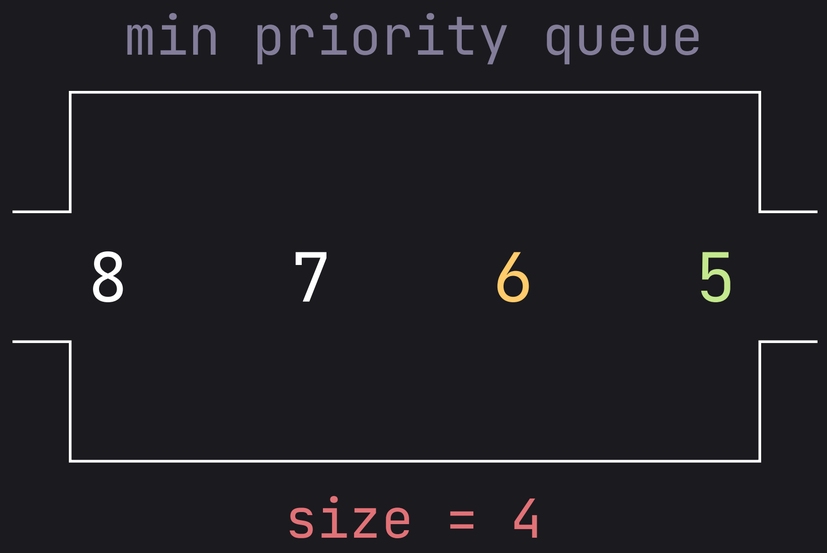


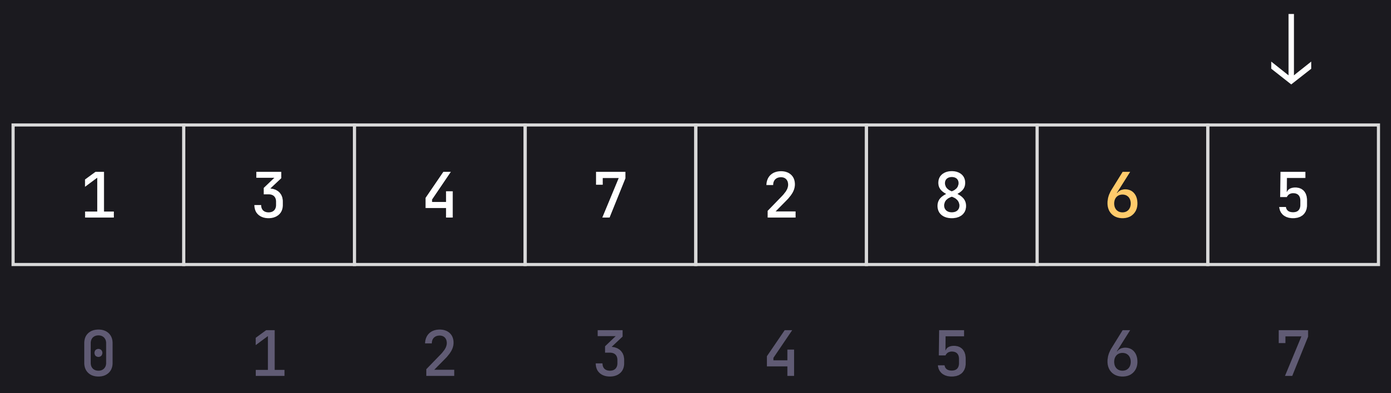
Kth Largest Element



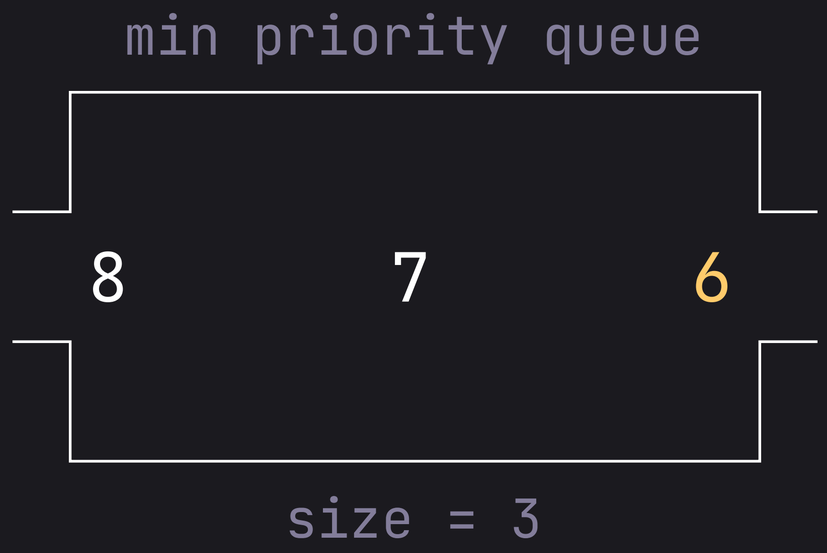


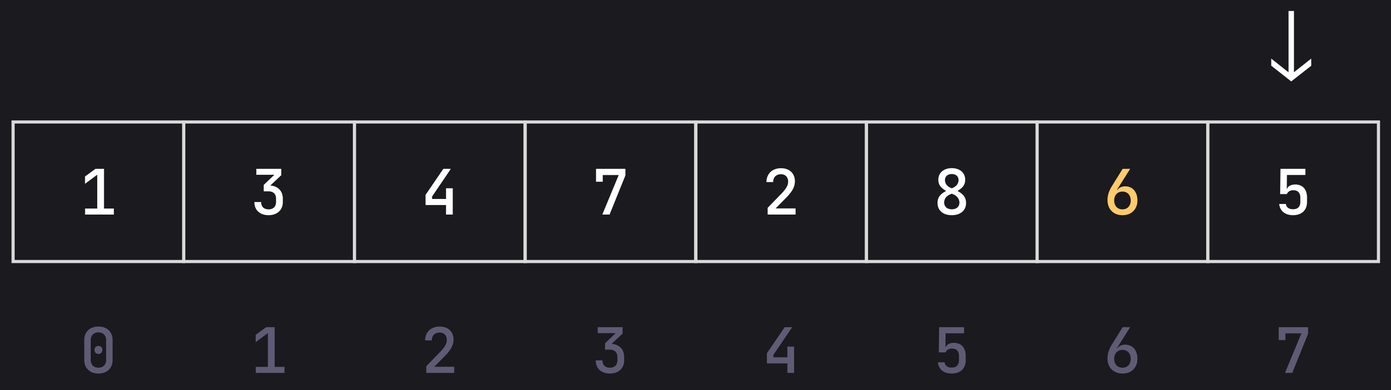
Kth Largest Element



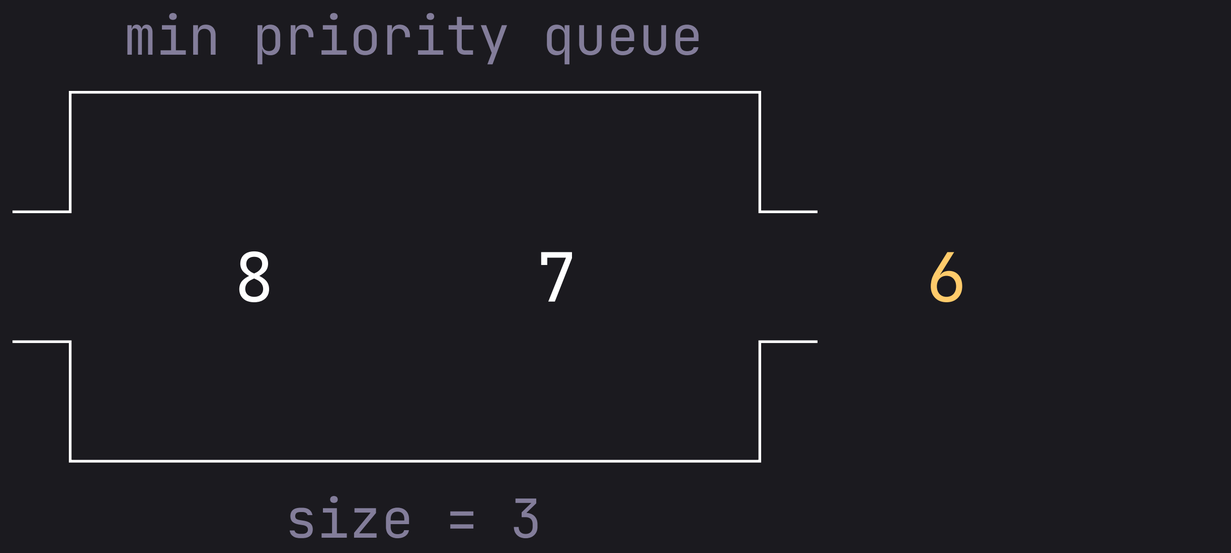


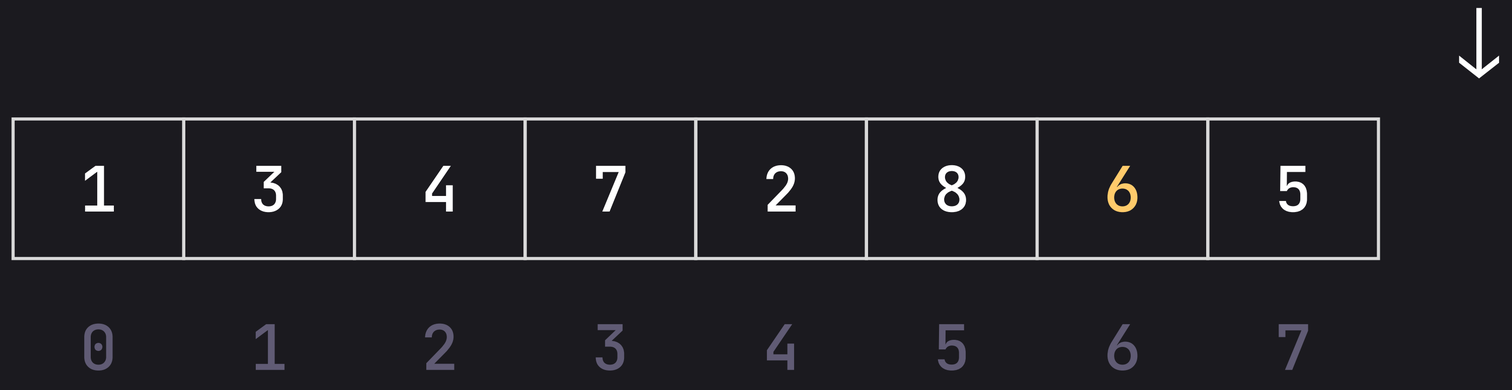
Kth Largest Element





Kth Largest Element

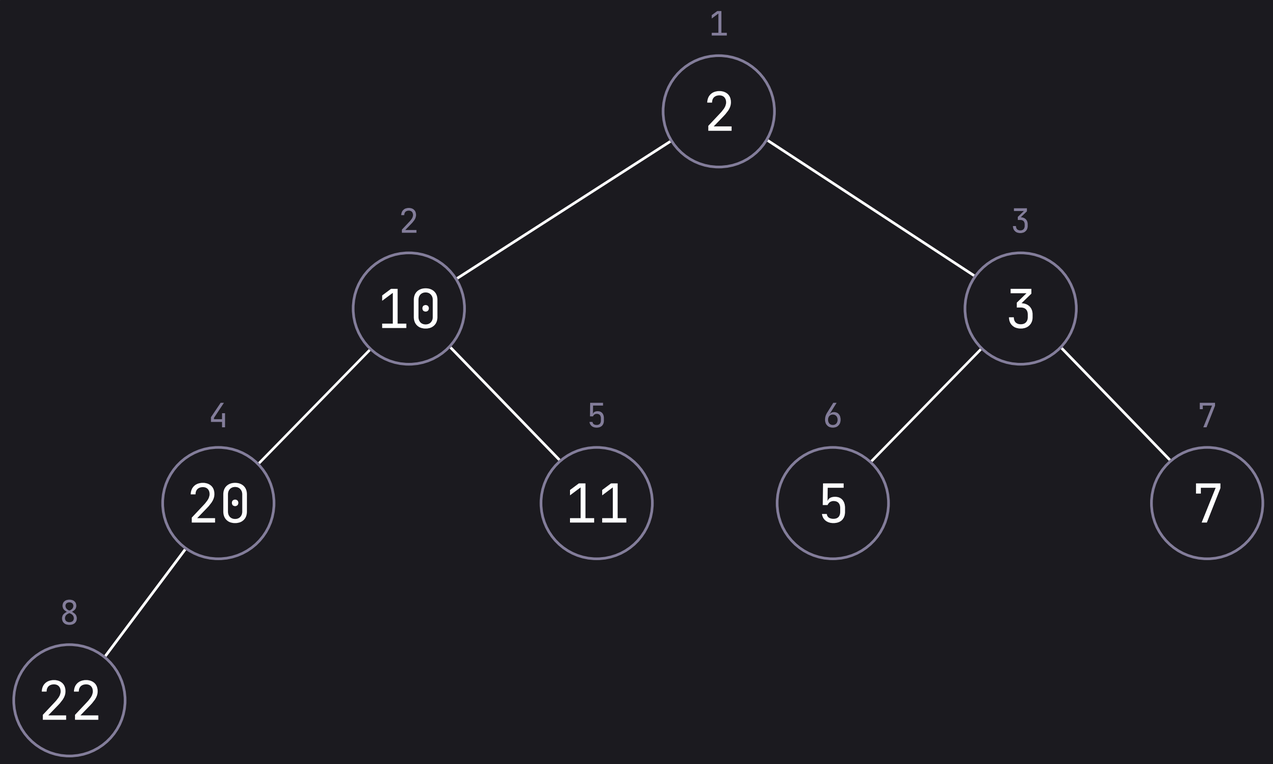




Kth Largest Element

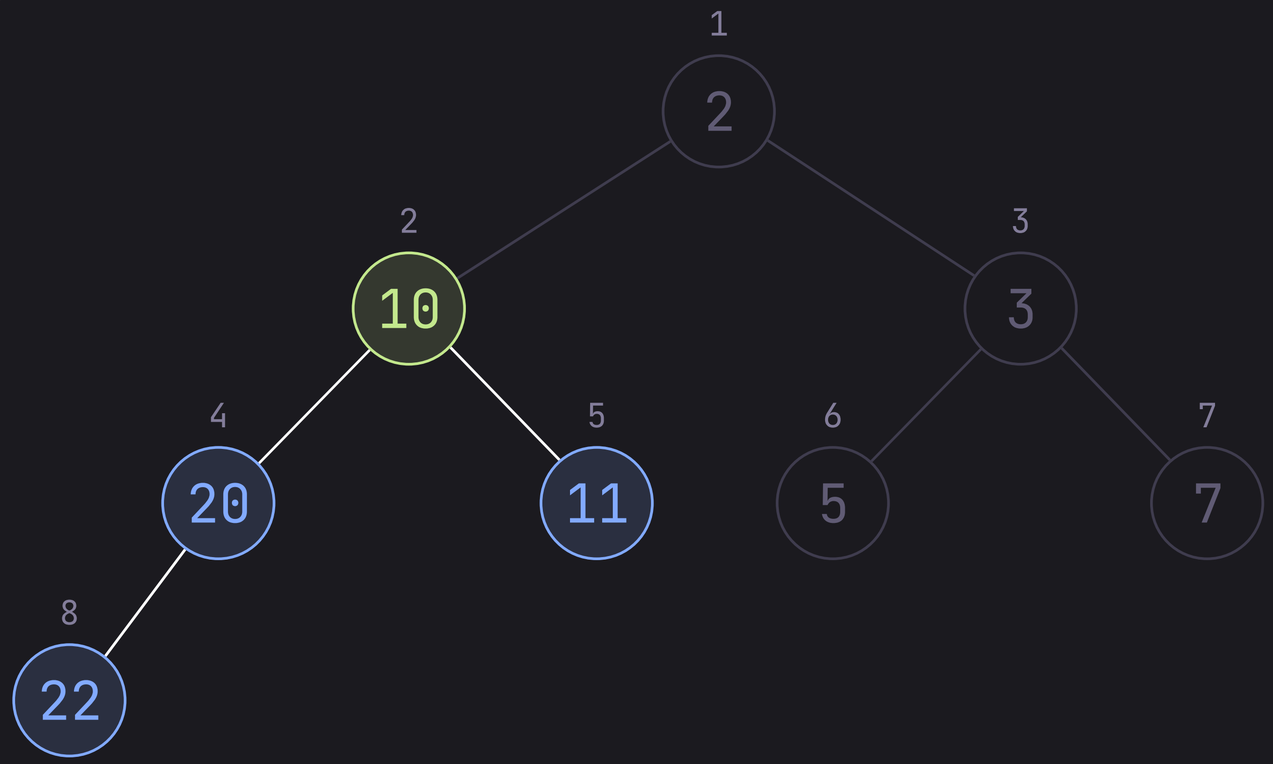
```
function kthLargest(vec, k):
  queue = new MinQueue()
  for x in vec:
    queue.push(x)
    if queue.size() > k:
      queue.pop()
  return queue.top()
```

Min Heap



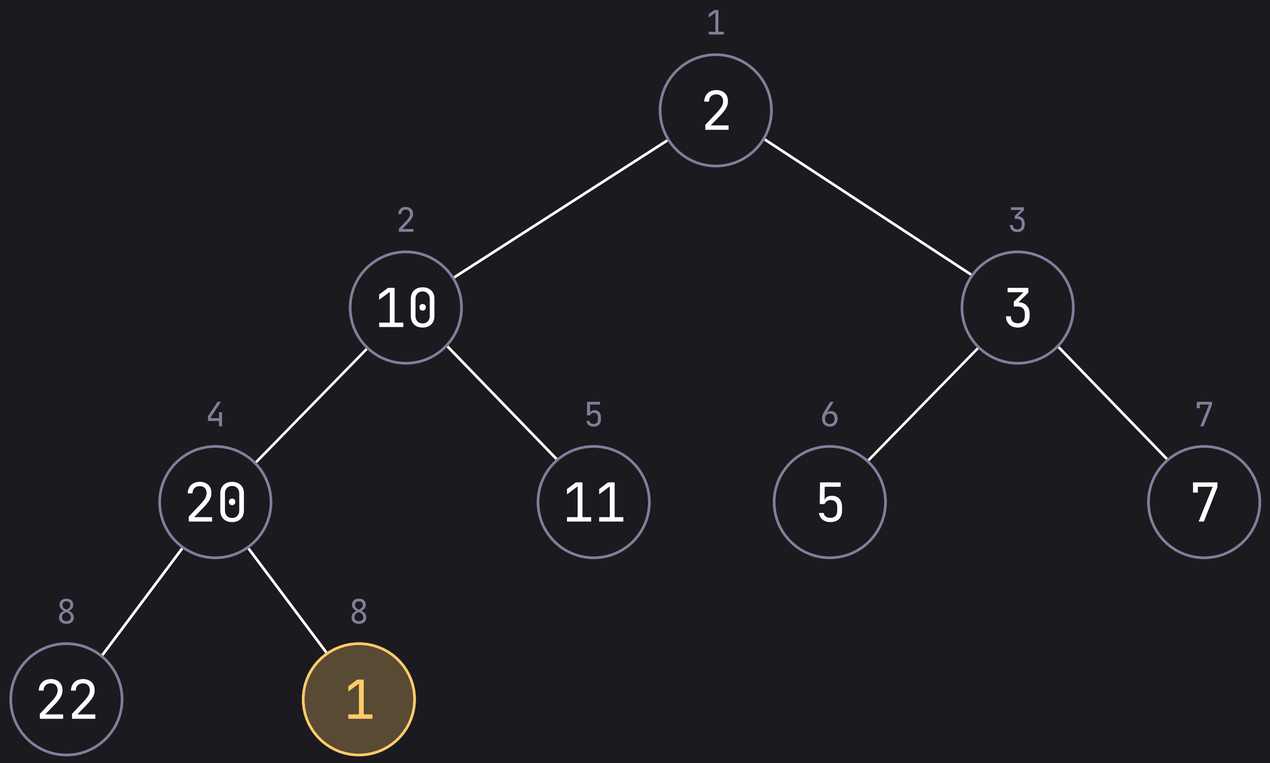
A min heap is a binary tree data structure where every node's value is less than or equal to its children's values

Min Heap



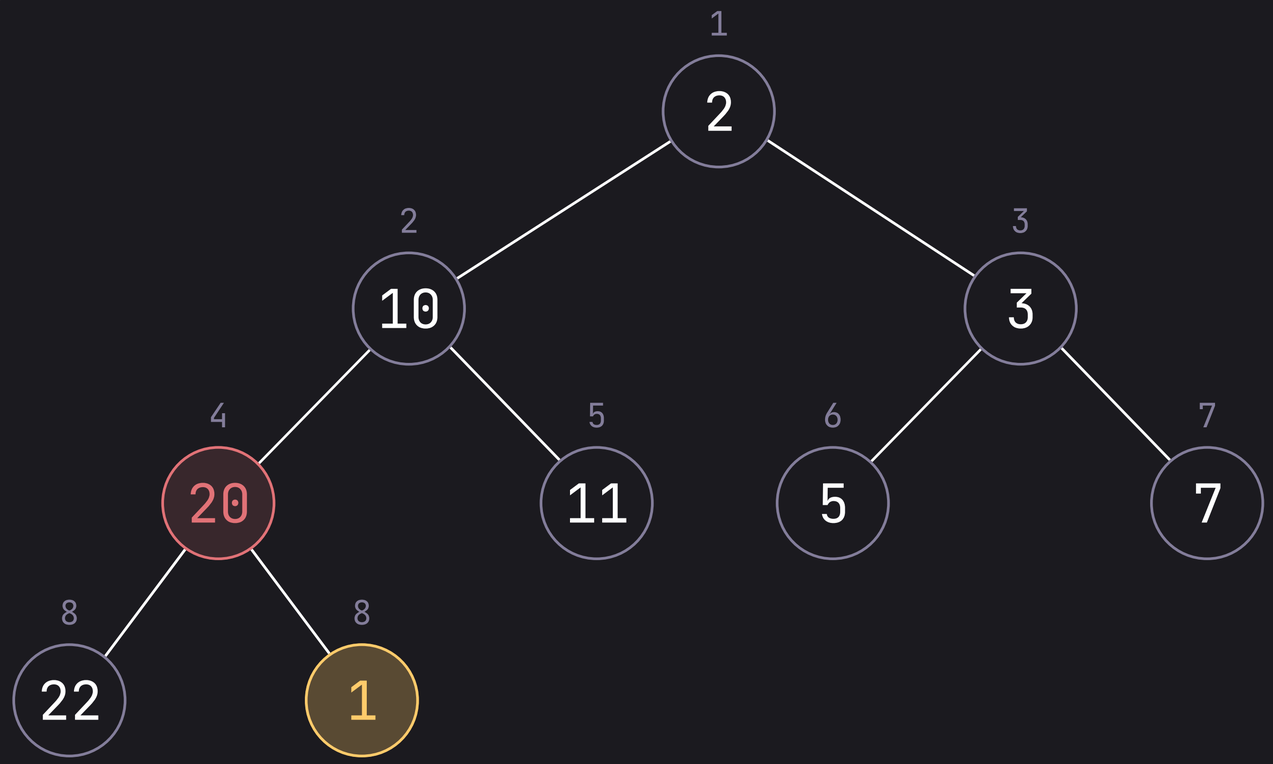
A min heap is a binary tree data structure where every node's value is less than or equal to its children's values

Min Heap



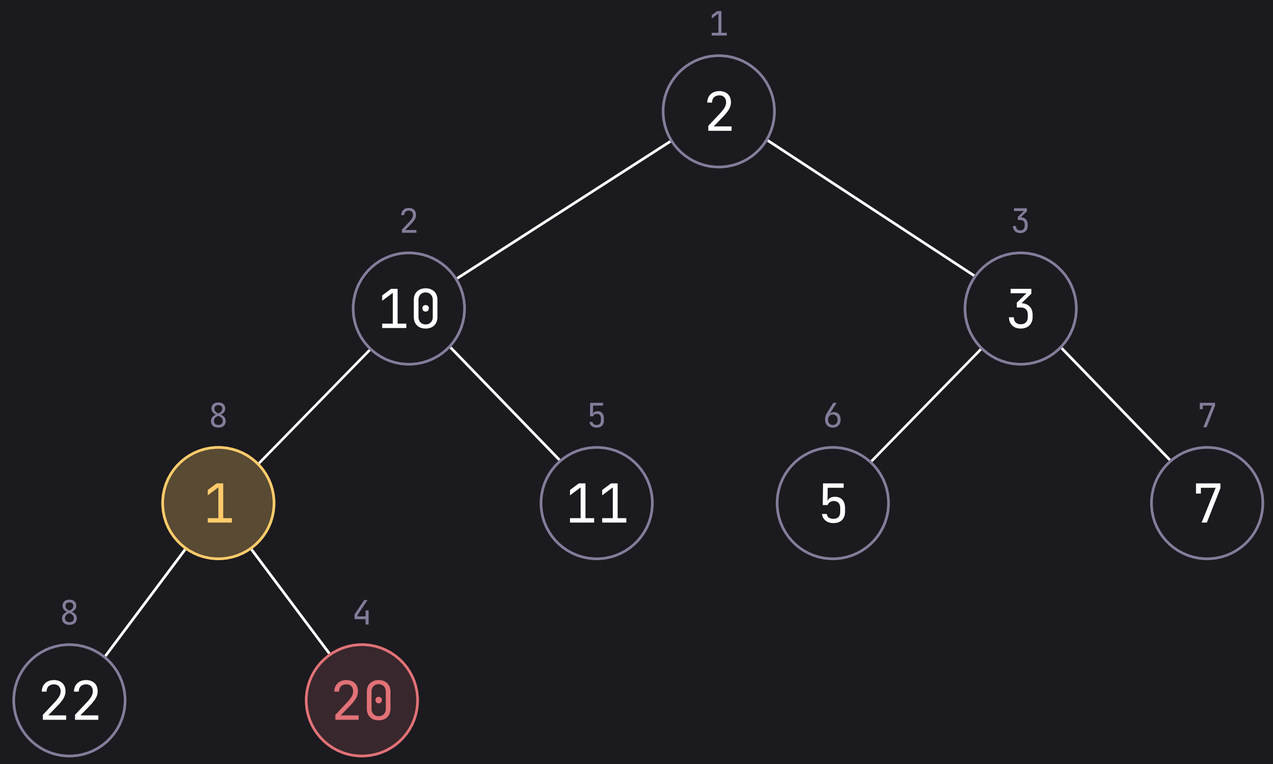
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



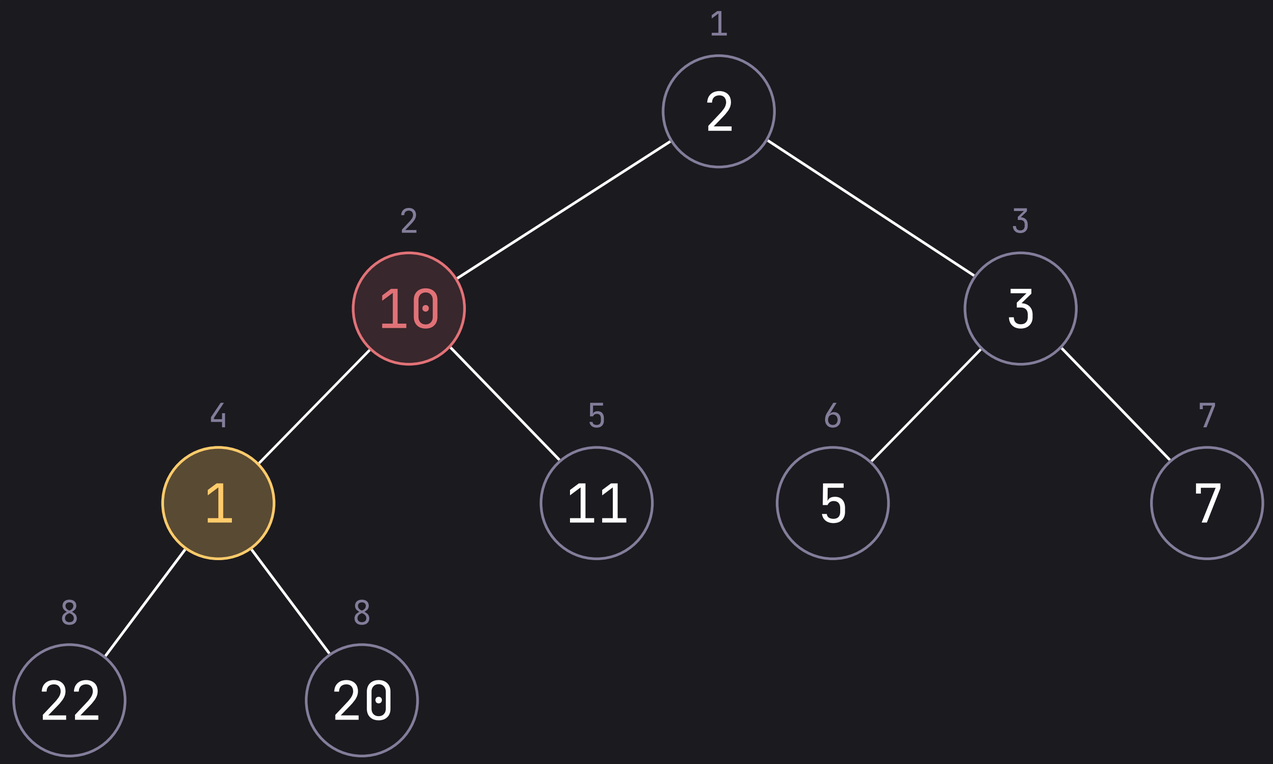
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



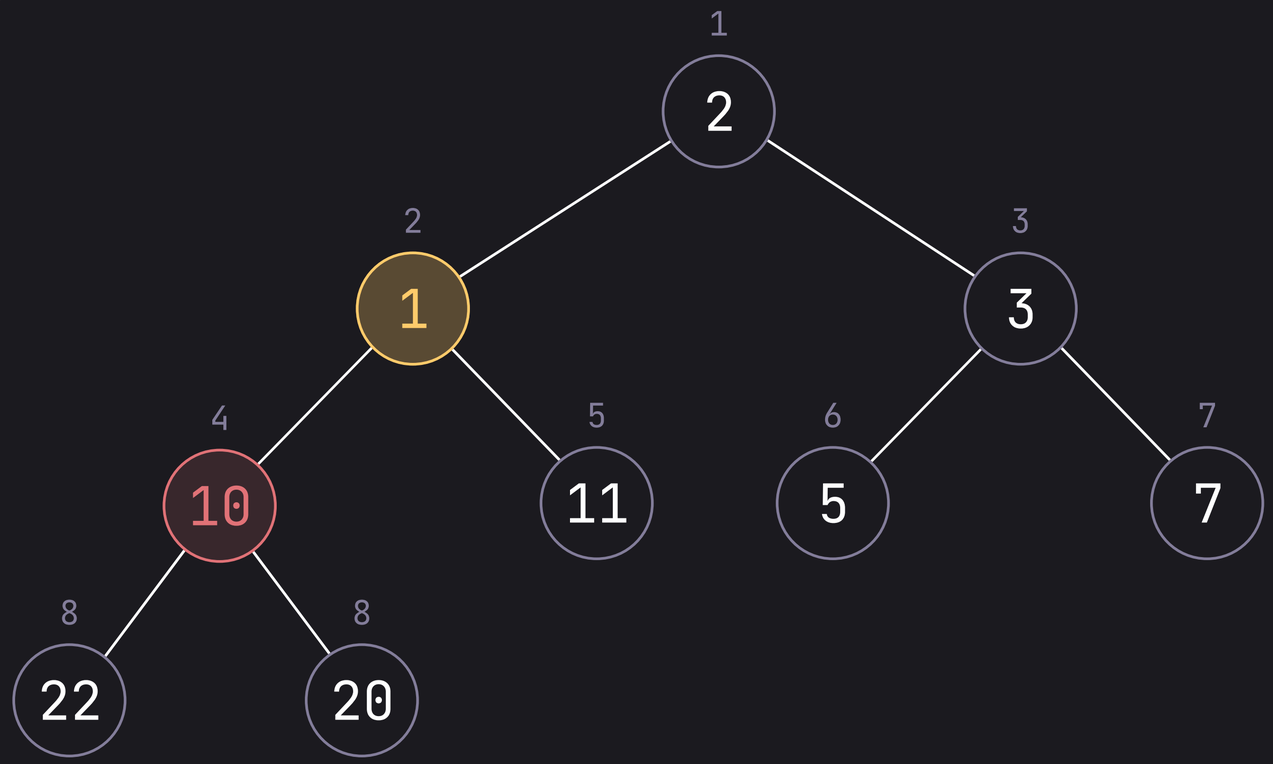
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



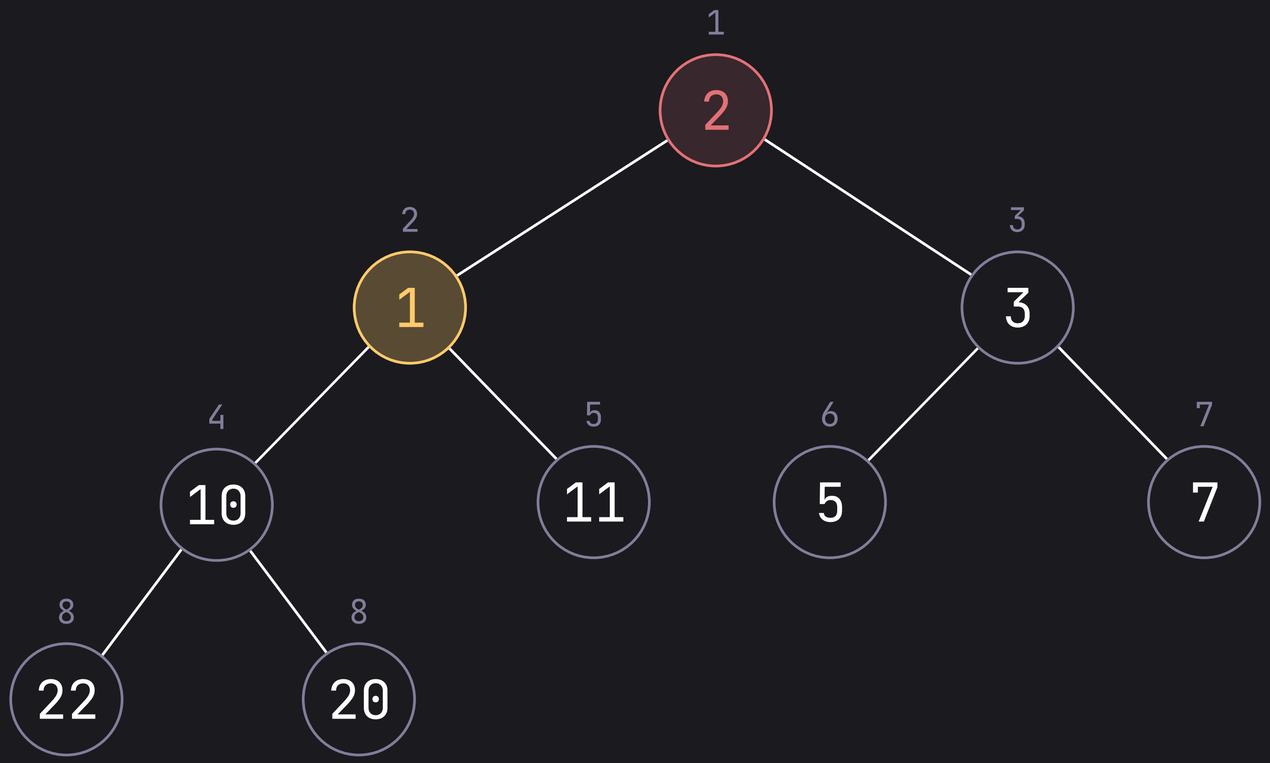
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



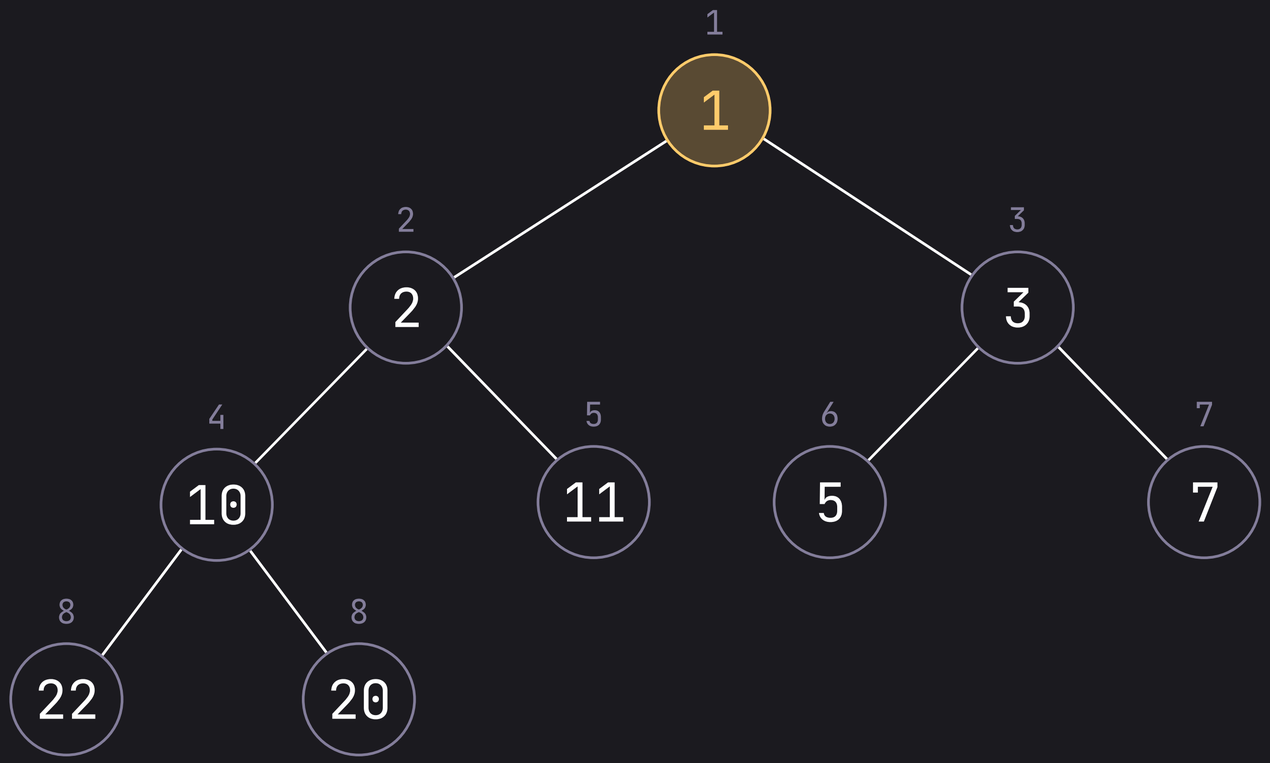
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



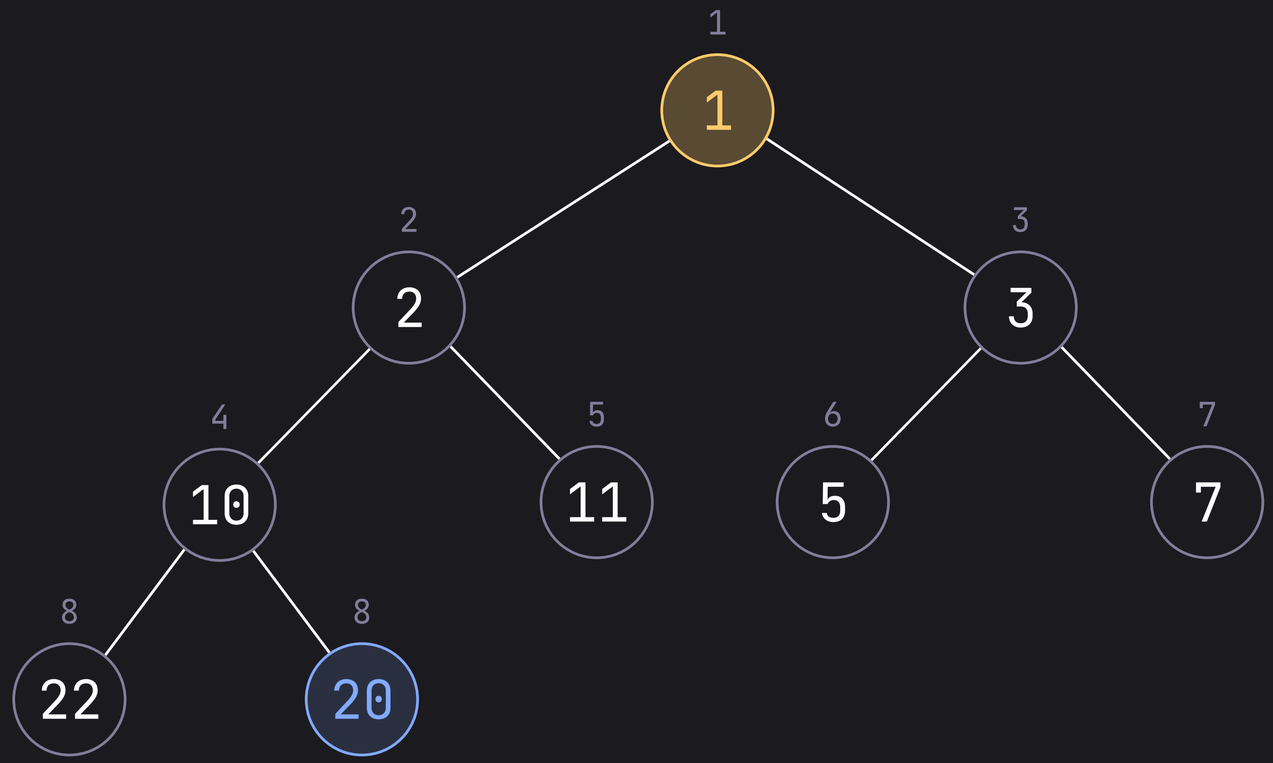
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



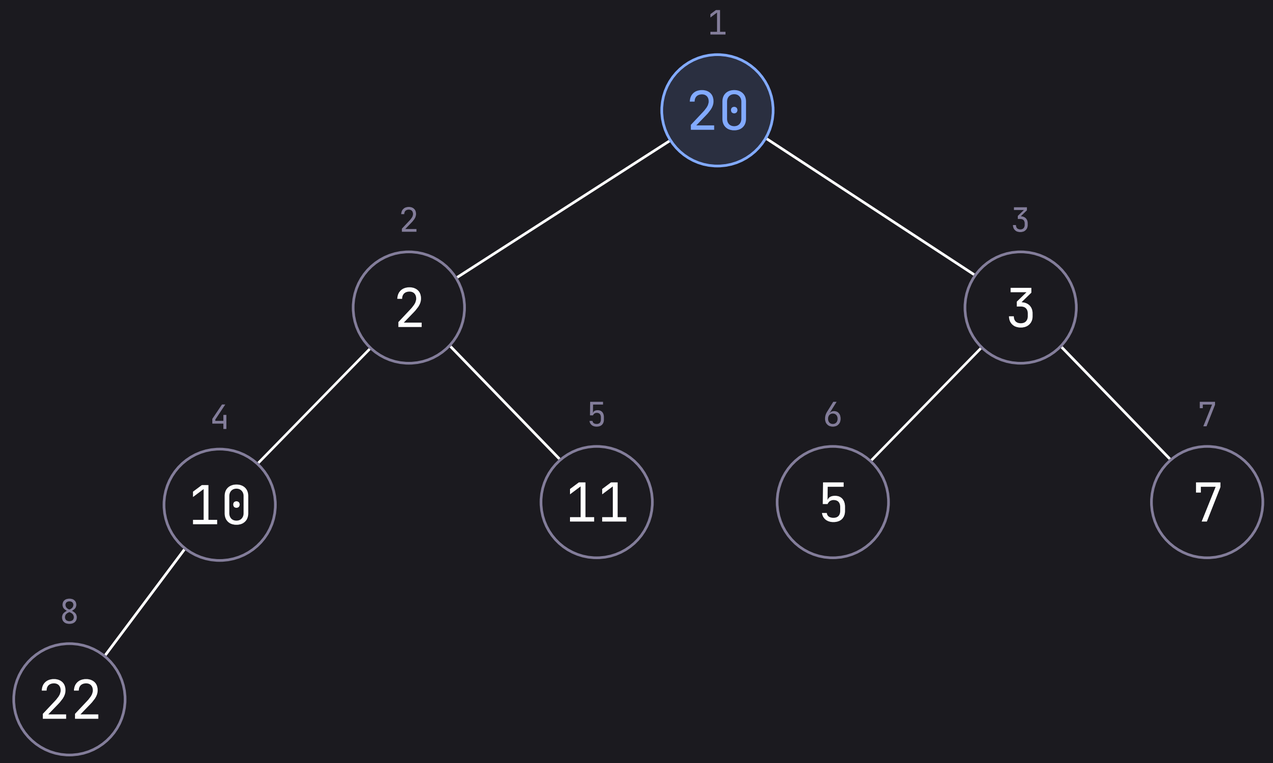
When we add a new element, we add it to the next free spot at the bottom.
And then we keep moving it up the tree until its in the right spot

Min Heap



When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

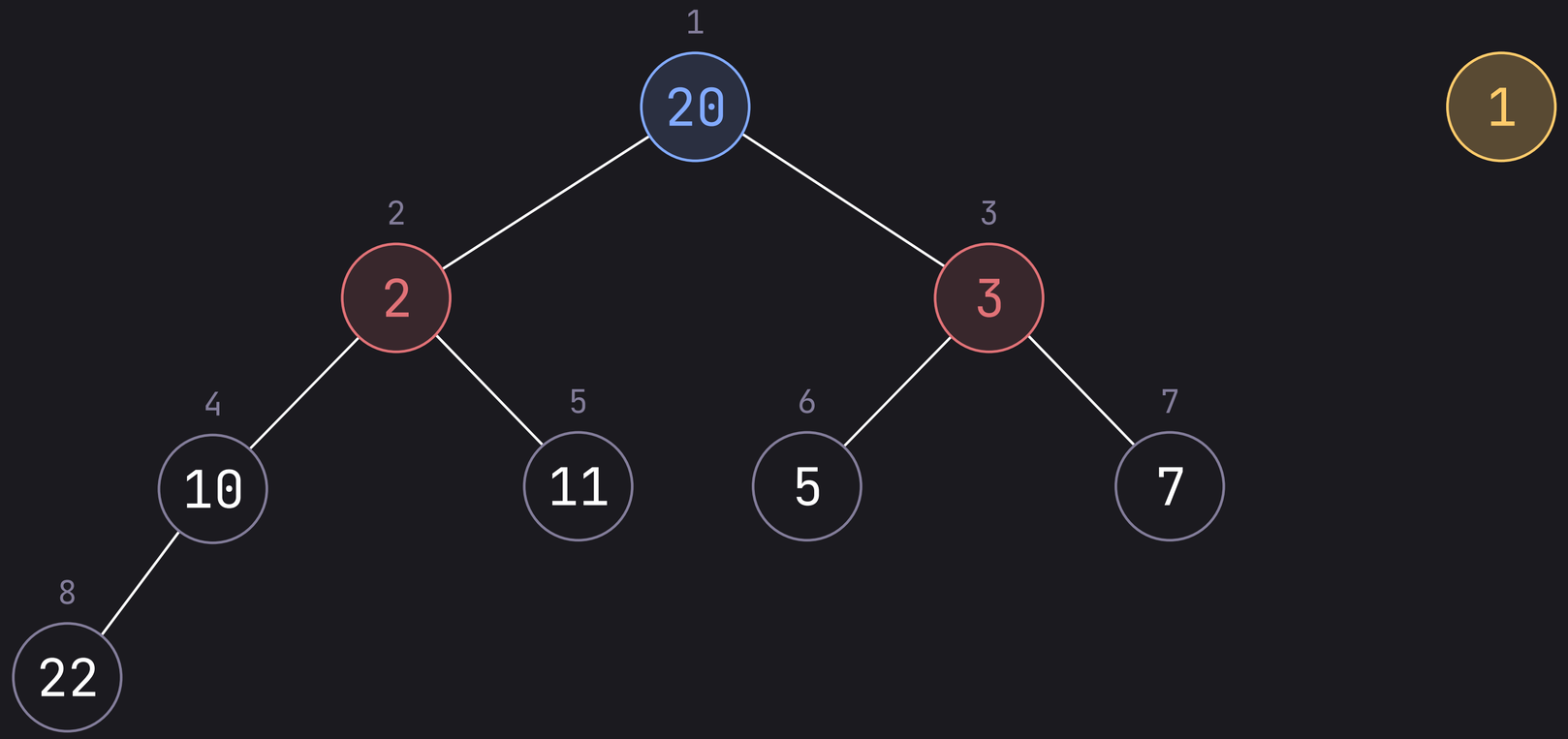
Min Heap



1

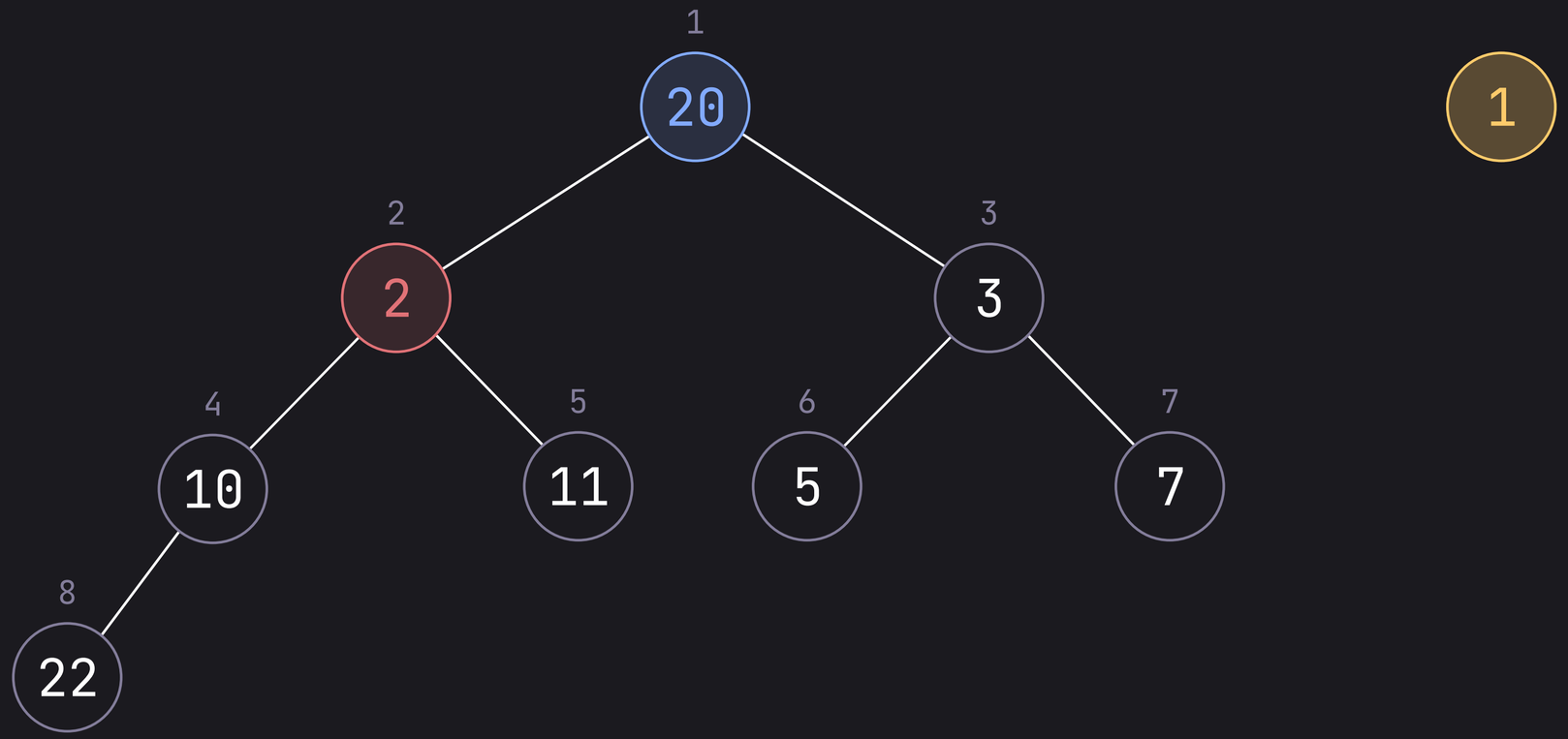
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



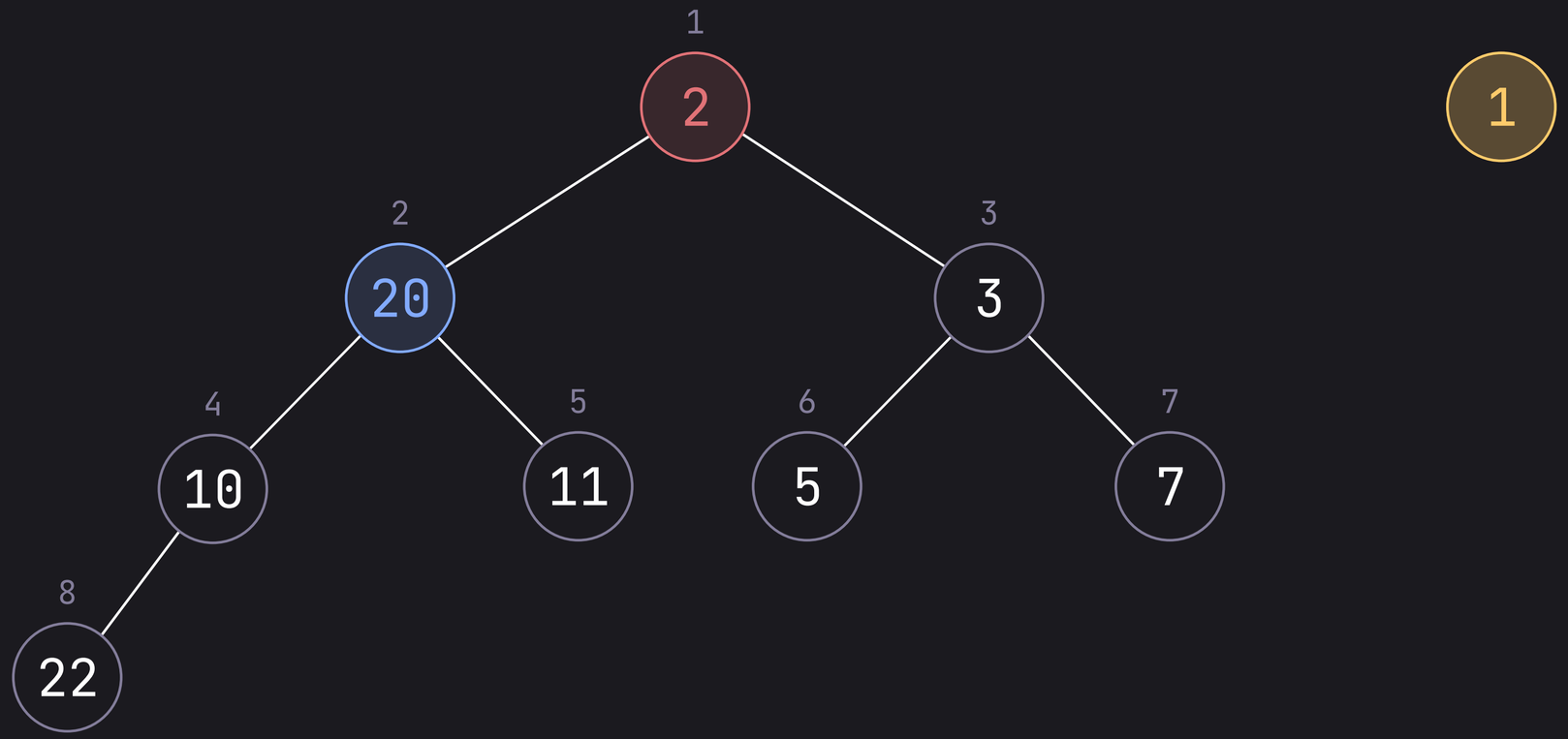
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



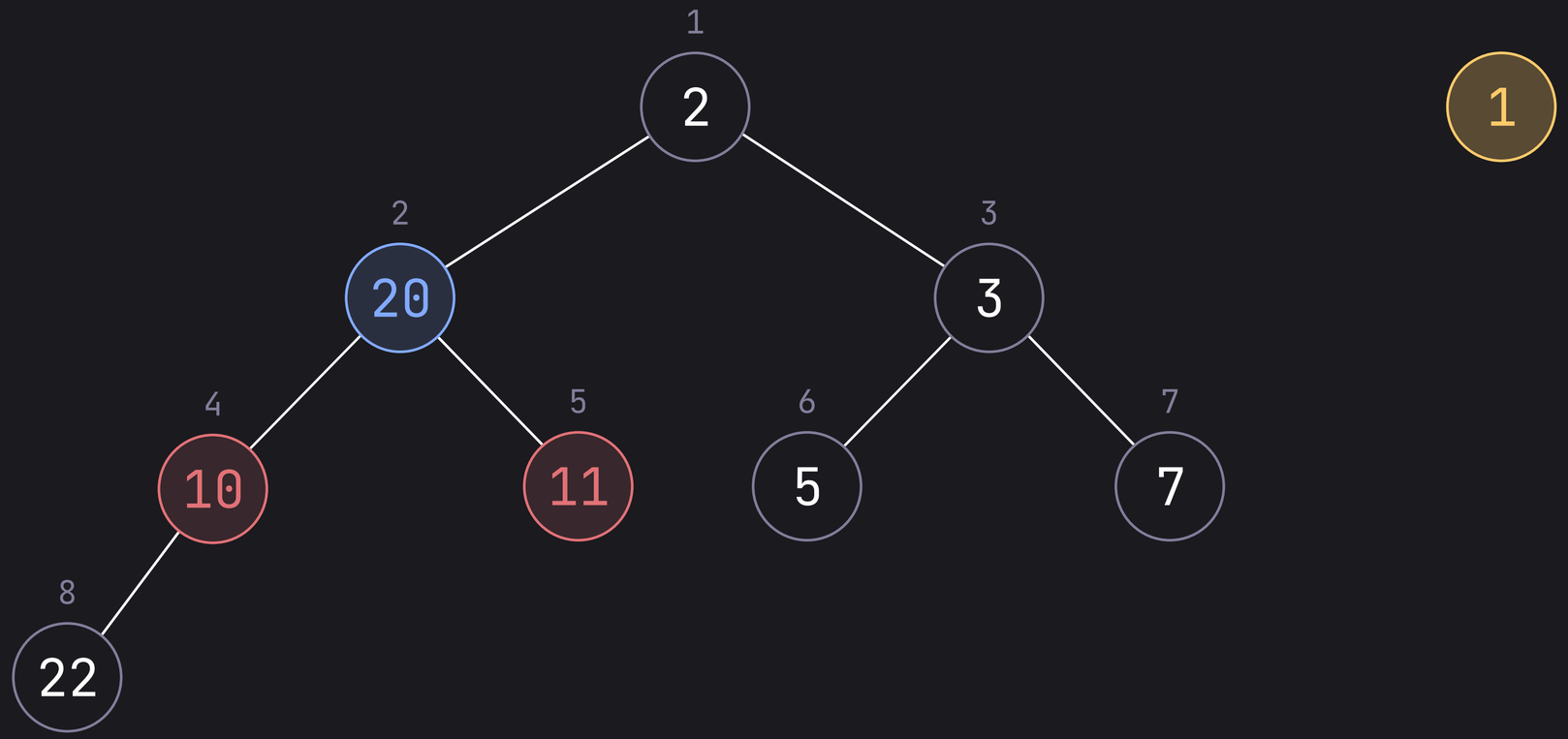
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



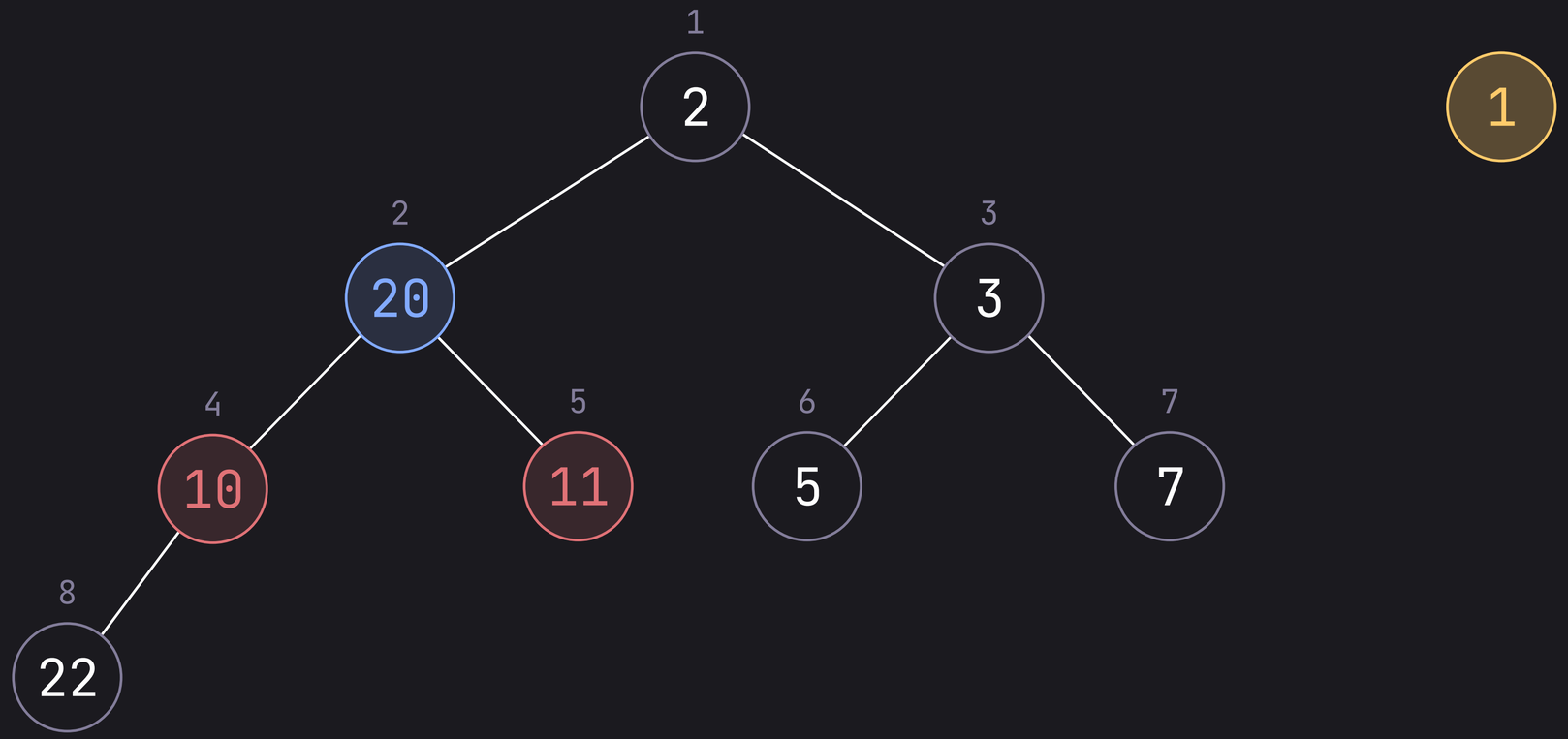
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



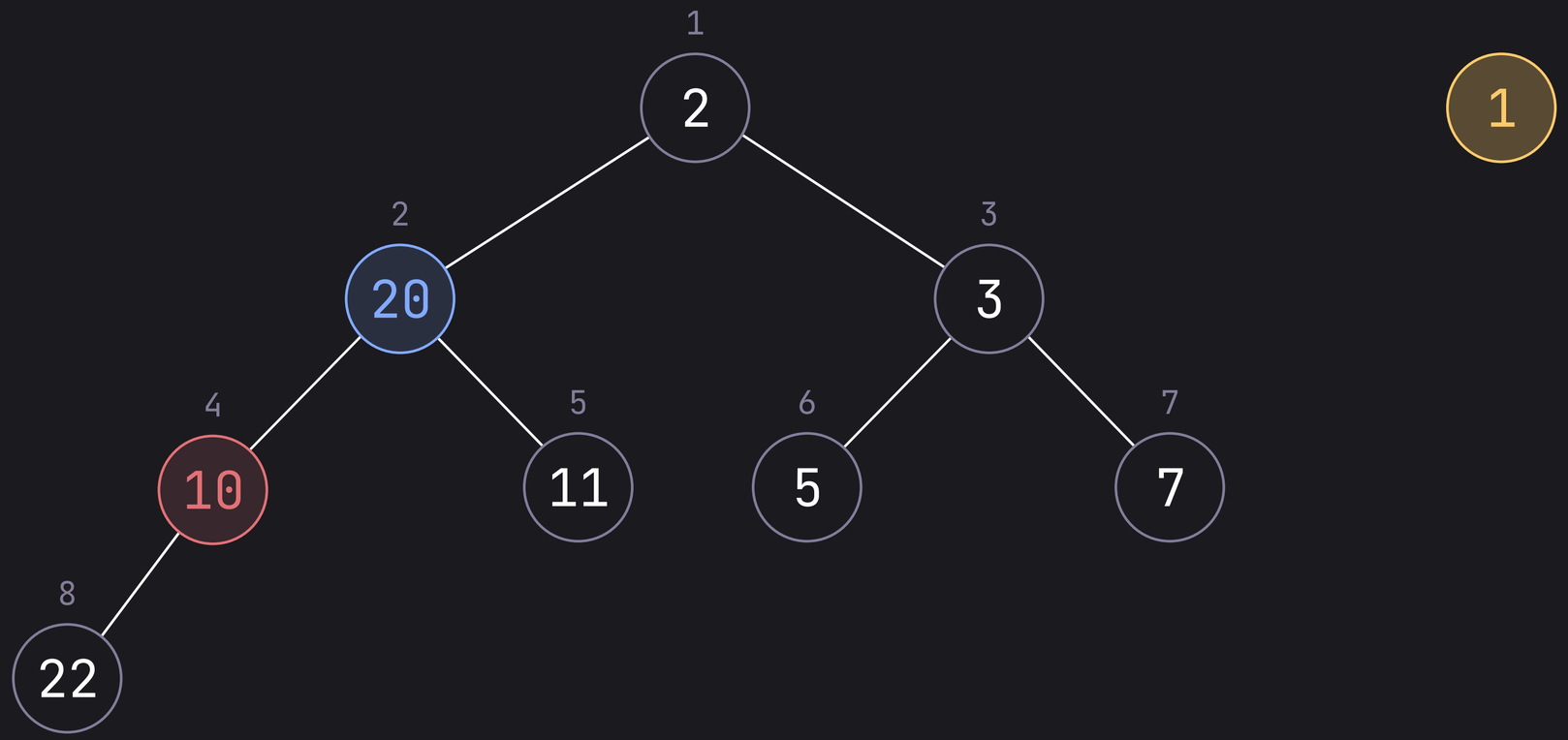
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



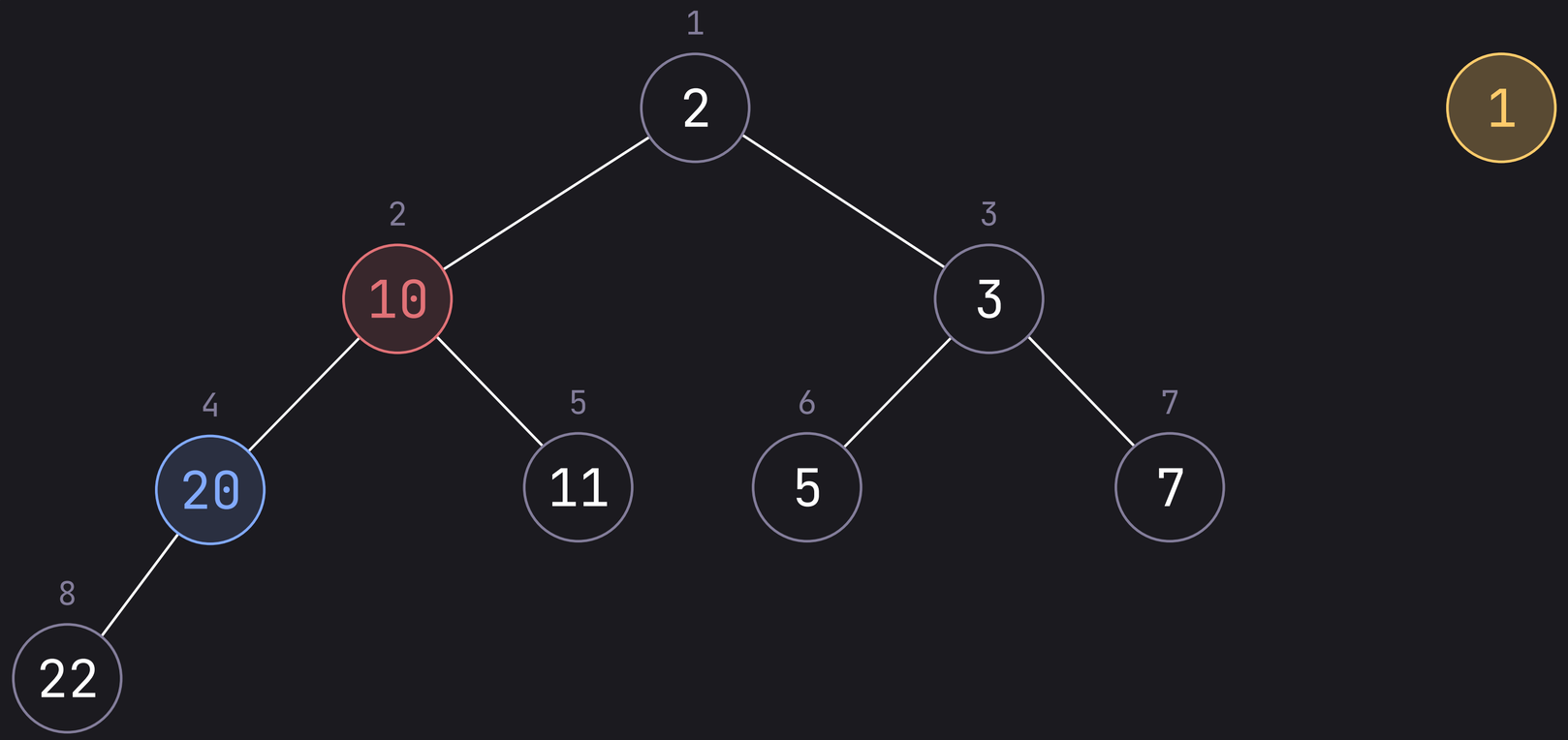
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



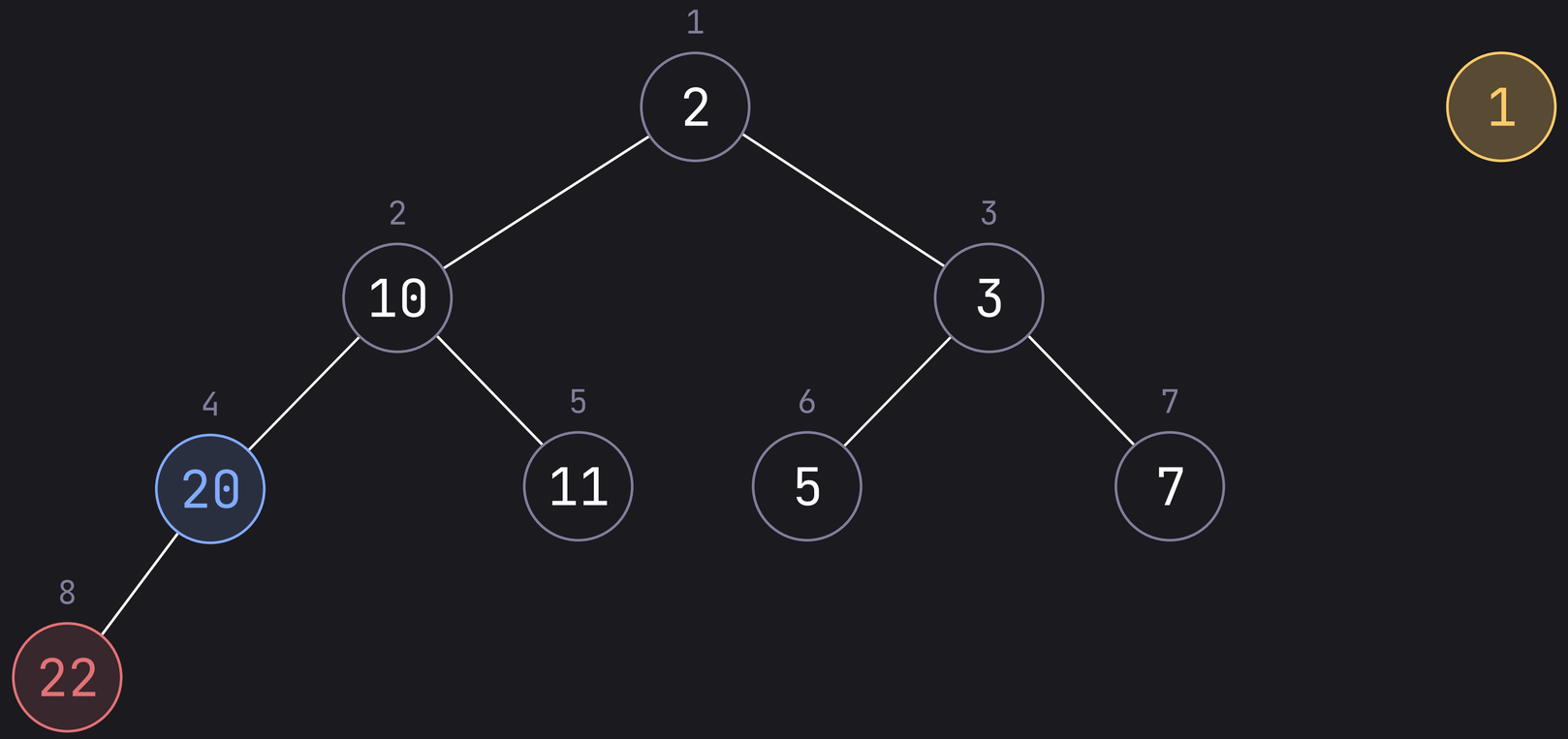
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



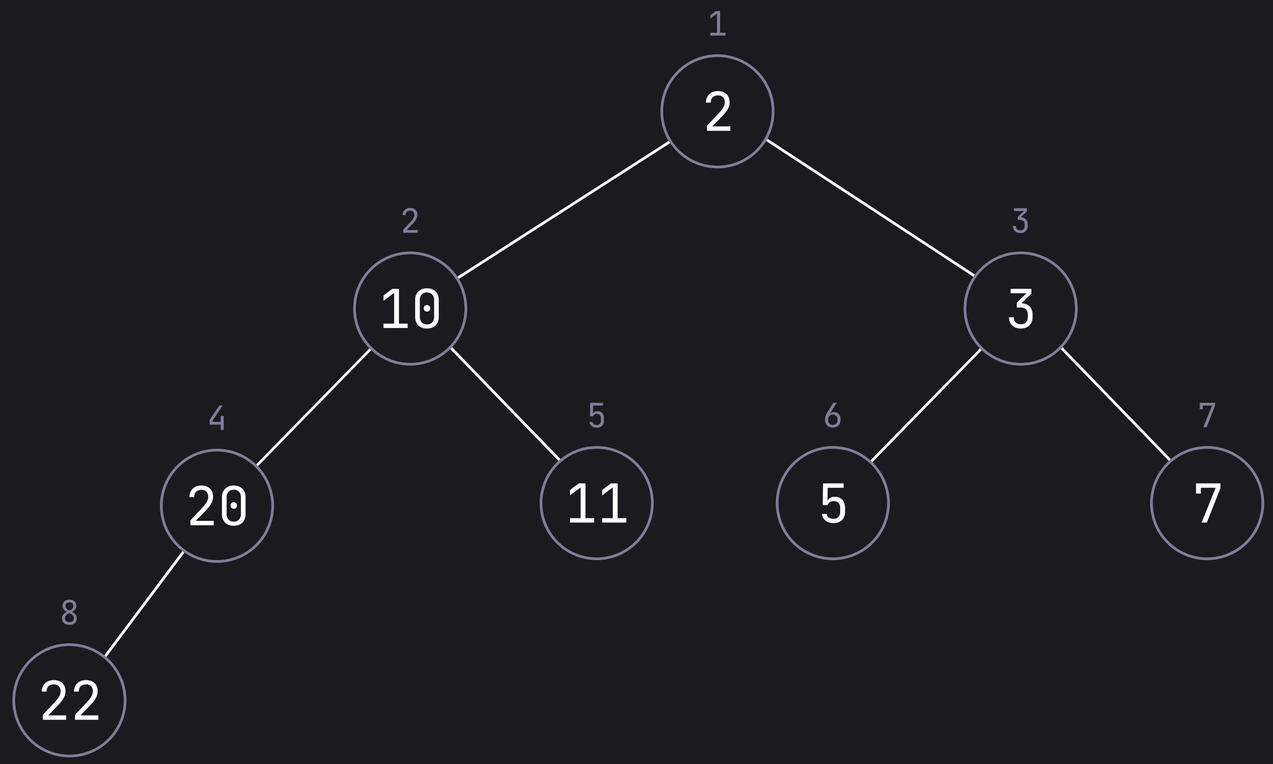
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap



When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

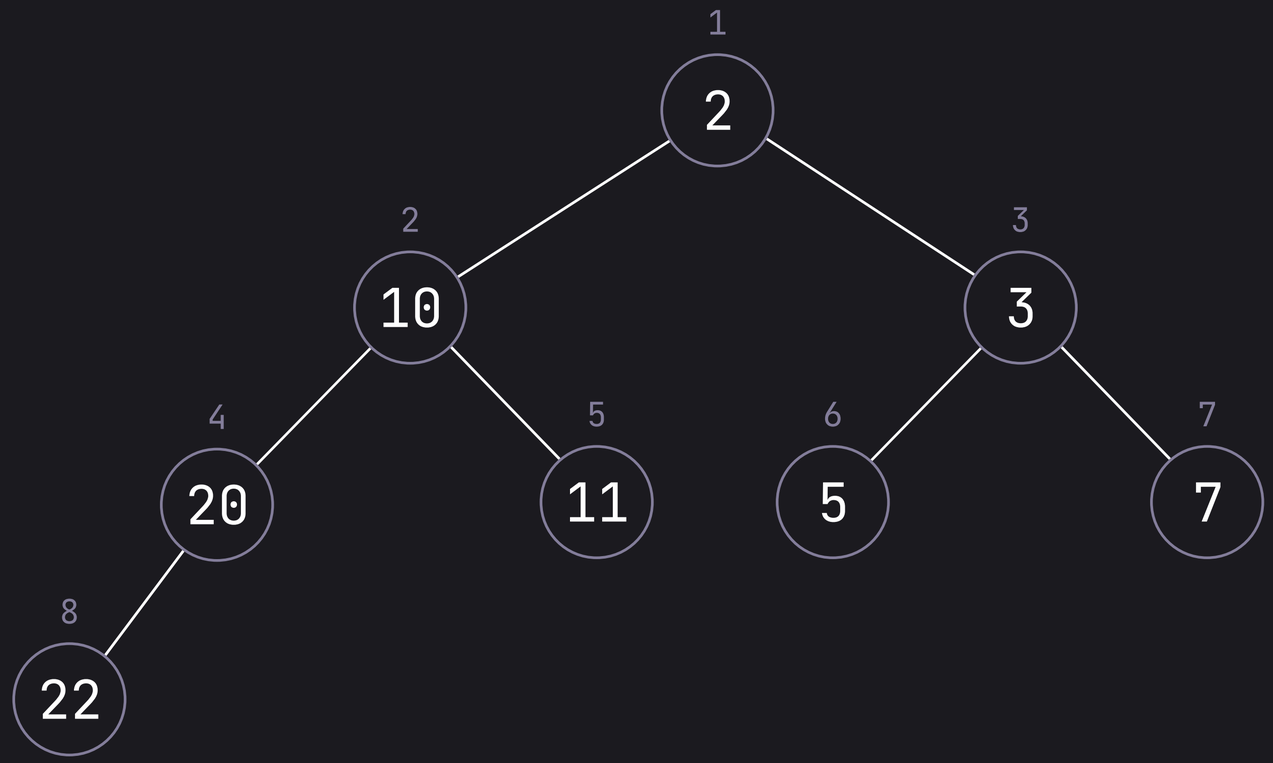
Min Heap



1

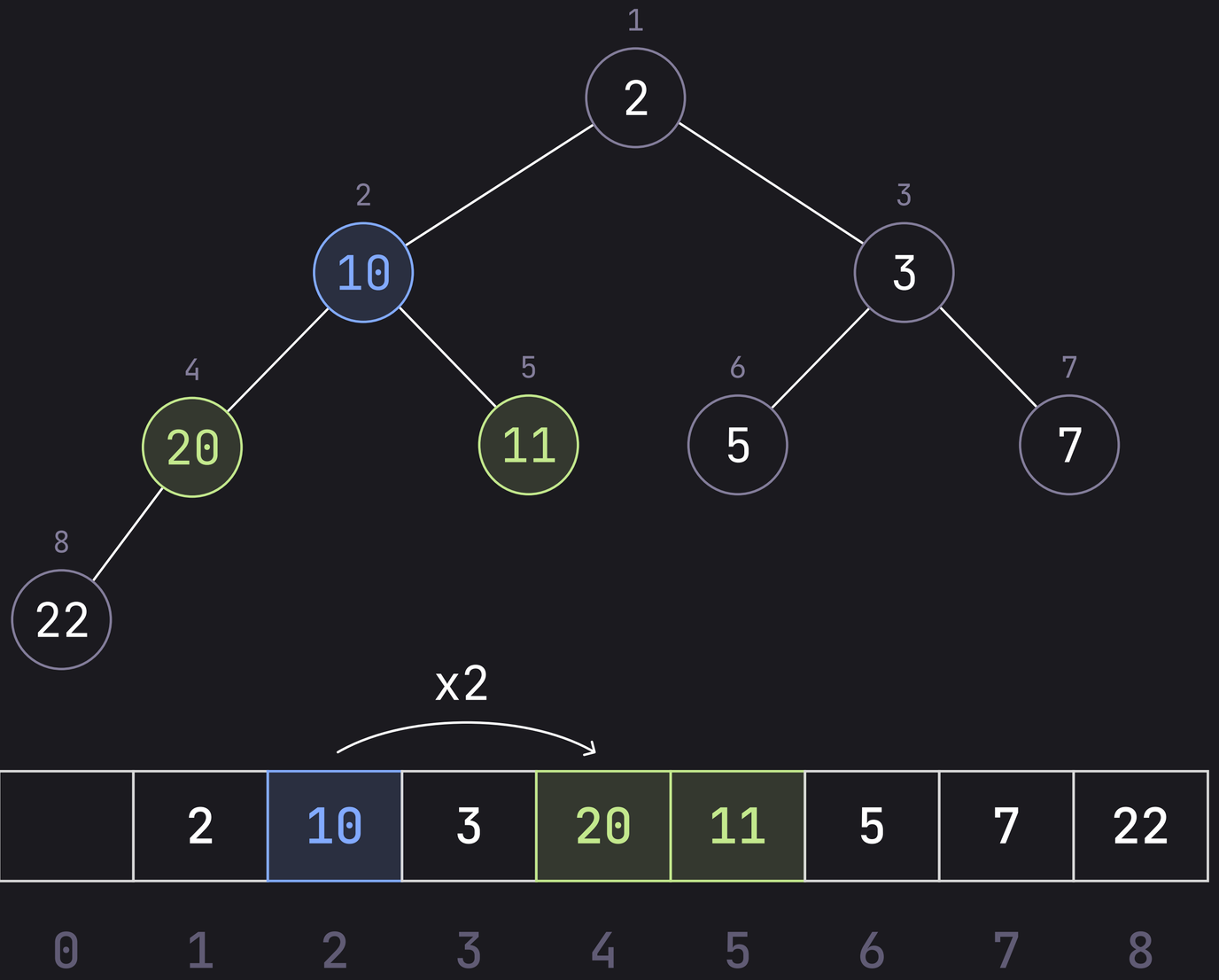
When we remove the min element, we replace it will the last element and then move it down the tree until its in the right spot

Min Heap

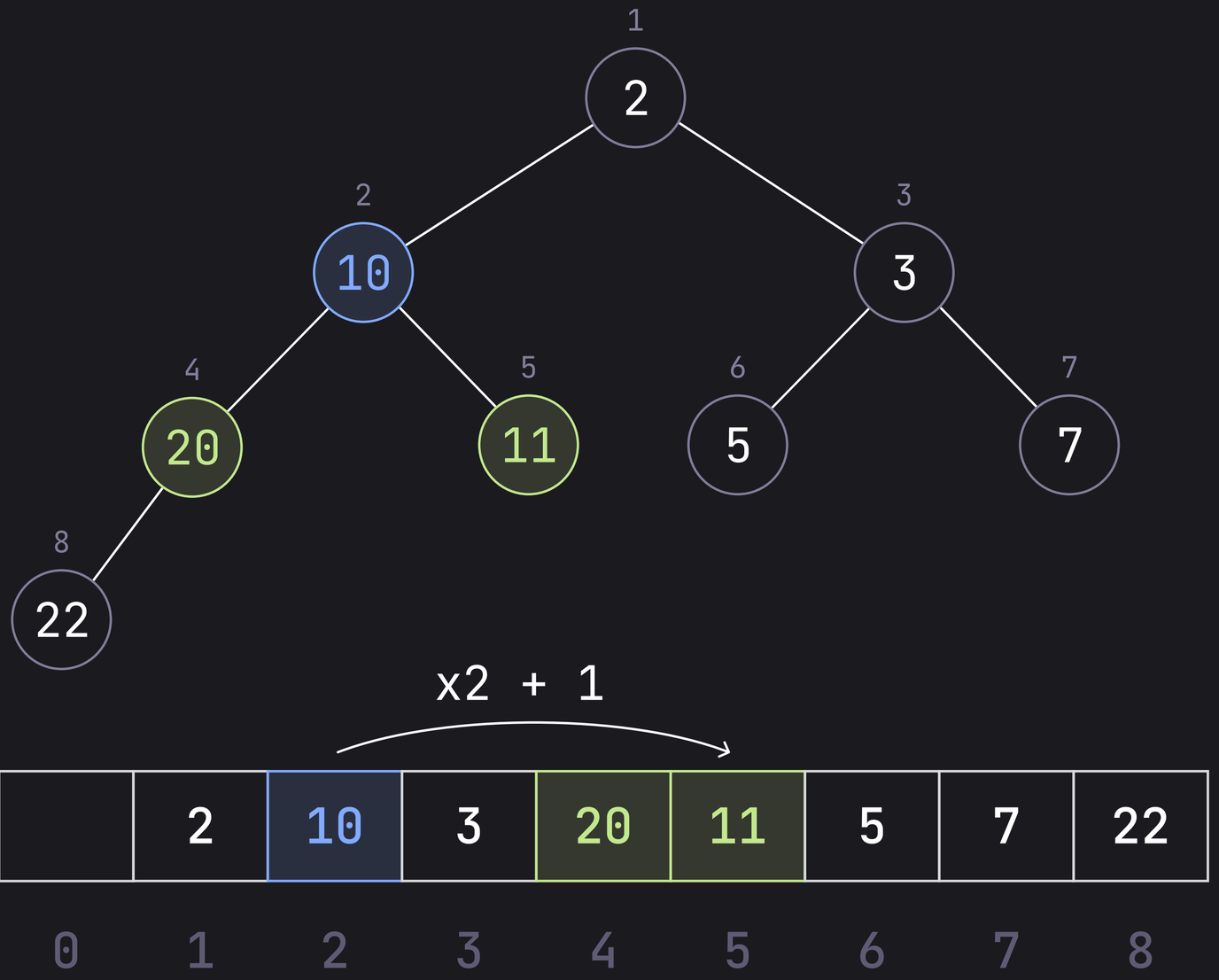


	2	10	3	20	11	5	7	22
0	1	2	3	4	5	6	7	8

Min Heap



Min Heap

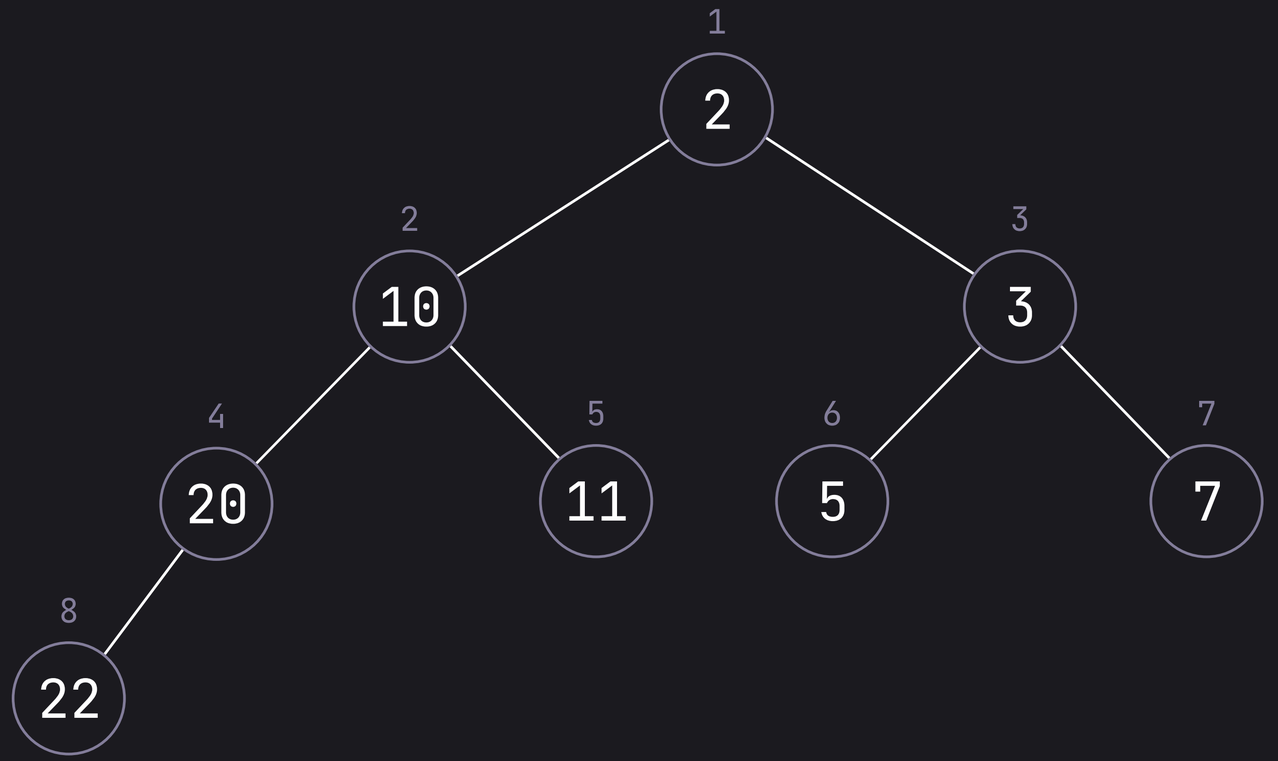


Min Heap

```
unsigned left(unsigned i) {
    return 2 * i ;
}

unsigned right (unsigned i) {
    return 2 * i+1 ;
}

unsigned parent(unsigned i) {
    return i / 2
}
```



	2	10	3	20	11	5	7	22
0	1	2	3	4	5	6	7	8