CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Heaps

Textbook Ch. 18

Announcements

- WA2 due Sunday

`PriorityQueue` ADT

`PriorityQueue[A <:Ordering]`

`enqueue(v: A): Unit`

Insert value ***v*** into the priority queue

`dequeue: A`

Remove the greatest element in the priority queue

`head: A`

Peek at the greatest element in the priority queue

Priority Queues

Two mentalities…

Lazy: Keep everything a mess ("Selection Sort")

Proactive: Keep everything organized ("Insertion Sort")

Priority Queues

Operation	Lazy	Proactive
enqueue	$O(1)$	$O(n)$
dequeue	$O(n)$	$O(1)$
head	$O(n)$	$O(1)$

Priority Queues

Operation	Lazy	Proactive
enqueue	$O(1)$	$O(n)$
dequeue	$O(n)$	$O(1)$
head	$O(n)$	$O(1)$

Can we do better?

Priority Queues

Lazy - Fast Enqueue, Slow Dequeue

Proactive - Slow Enqueue, Fast Dequeue

Priority Queues

Lazy - Fast Enqueue, Slow Dequeue

Proactive - Slow Enqueue, Fast Dequeue

??? - Fast(-ish) Enqueue, Fast(-ish) Dequeue

Priority Queues

Idea: Keep the priority queue "kinda" sorted.

Hopefully "kinda" sorted is cheaper to maintain than a full sort,

but still gives us some of the benefits.

Priority Queues

Idea: Keep the priority queue "kinda" sorted.

Keep larger items towards the front of the list,

and keep the front of the list more sorted than the back…

Binary Heaps

Challenge: If we are only "kinda" sorting, how do we know which elements are actually sorted?

Binary Heaps

Idea: Organize the priority queue as a *directed* tree!

A directed edge from ***a*** to ***b*** means that $\boldsymbol{a \geq b}$

More Tree Terminology

Child - An adjacent node connected by an out-edge

More Tree Terminology

Child - An adjacent node connected by an out-edge

Leaf - A node with no children

More Tree Terminology

Child - An adjacent node connected by an out-edge

Leaf - A node with no children

Depth (of a node) - The number of edges from the root to the node

More Tree Terminology

Child - An adjacent node connected by an out-edge

Leaf - A node with no children

Depth (of a node) - The number of edges from the root to the node

Depth (of a tree) - The maximum depth of any node in the tree

More Tree Terminology

Child - An adjacent node connected by an out-edge

Leaf - A node with no children

Depth (of a node) - The number of edges from the root to the node

Depth (of a tree) - The maximum depth of any node in the tree

Level (of a node) - depth + 1

More Tree Terminology

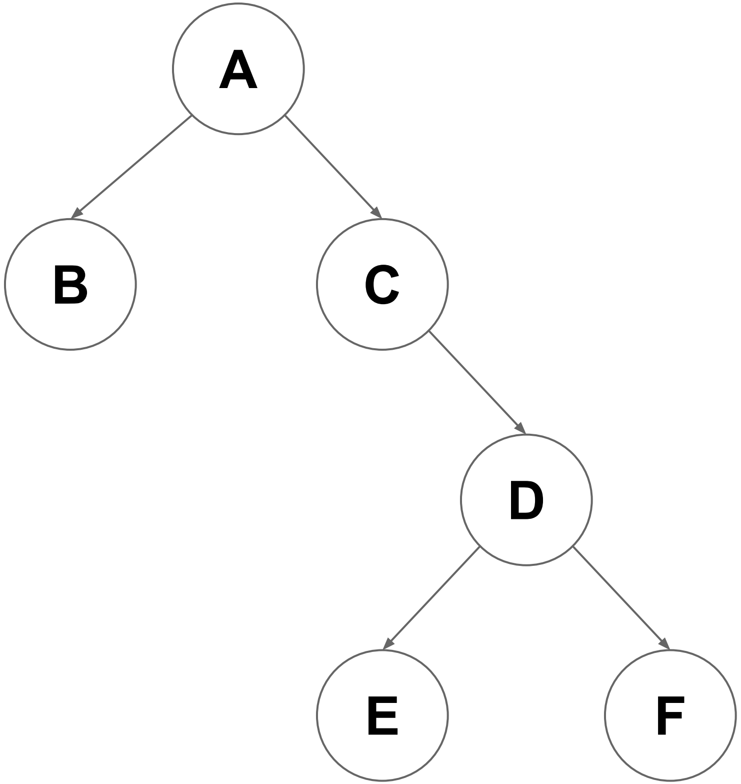
A is the root

B and **C** are children of **A**
D is a child of **C**
E and **F** are children of **D**

B, **E** and **F** are leaves

The depth of **A** is 0, **B** and **C:** 1, **D**: 2, **E** and **F**: 3

The depth of the tree is 3



Binary Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that $a \geq b$

Binary Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that ***a* ≥ *b***

Binary: Max out-degree of 2 (easy to reason about)

Binary Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that ***a* ≥ *b***

Binary: Max out-degree of 2 (easy to reason about)

Complete: Every "level" except the last is full (from left to right)

Binary Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that ***a* ≥ *b***

Binary: Max out-degree of 2 (easy to reason about)

Complete: Every "level" except the last is full (from left to right)

Balanced: TBD (basically, all leaves are roughly at the same level)

Binary Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that ***a* ≥ *b***

Binary: Max out-degree of 2 (easy to reason about)

Complete: Every "level" except the last is full (from left to right)

Balanced: TBD (basically, all leaves are roughly at the same level)

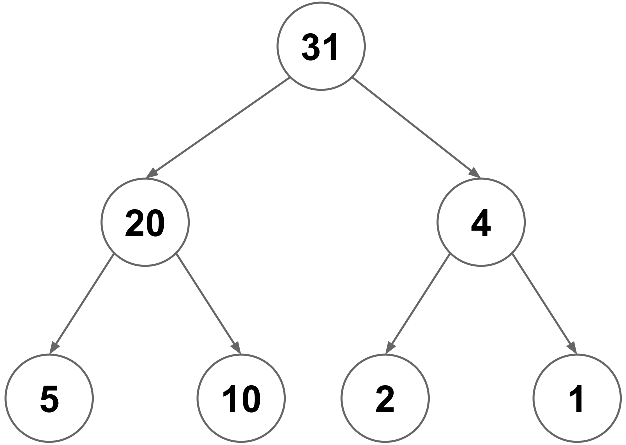
This makes it easy to encode into an array (later today)

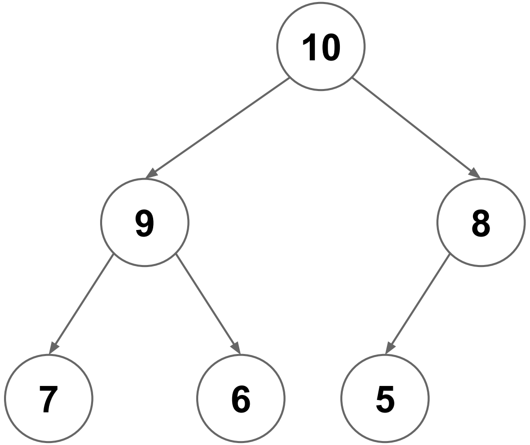
Max Heaps

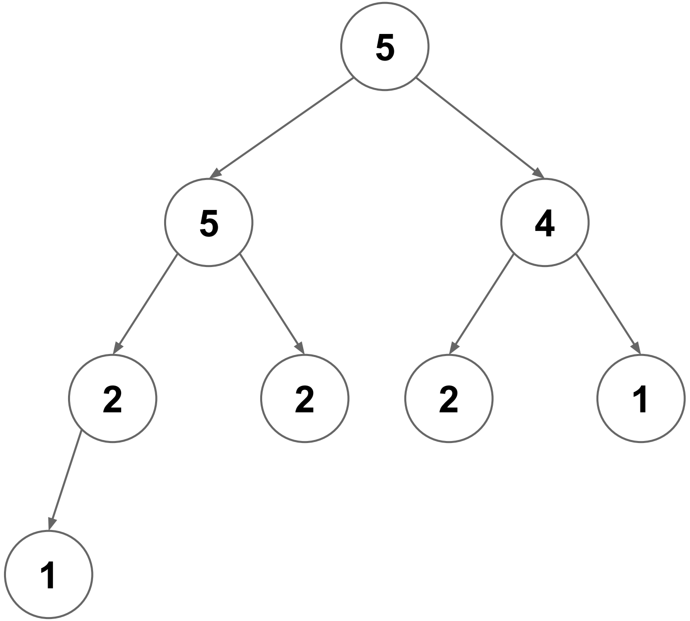
If we use ≥ as our ordering operation, we have a Max Heap

(as compared to a Min Heap)

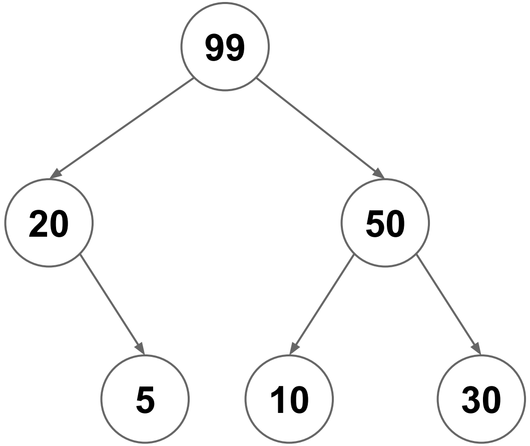
Valid Max Heaps

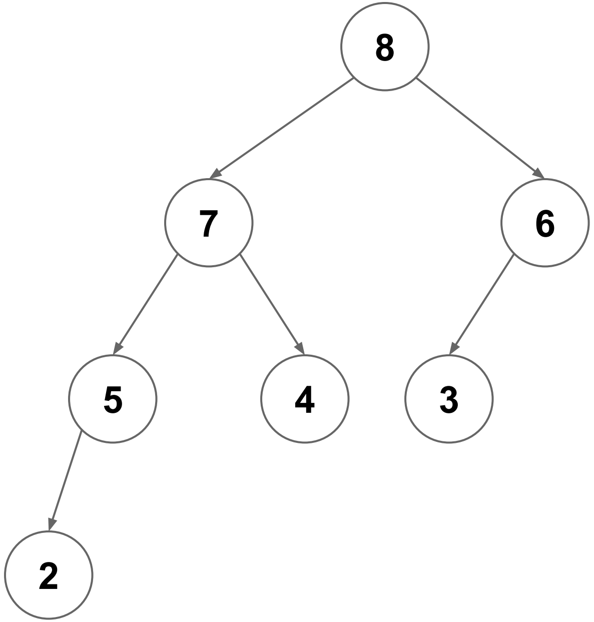


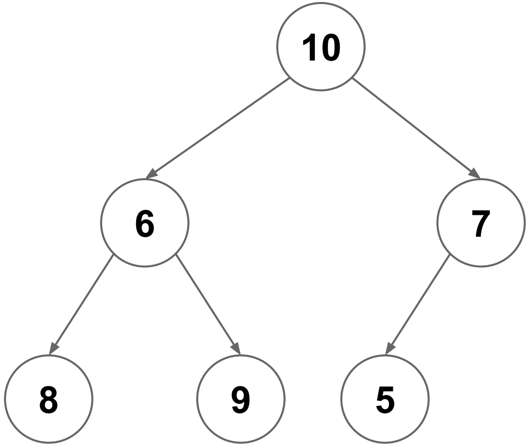




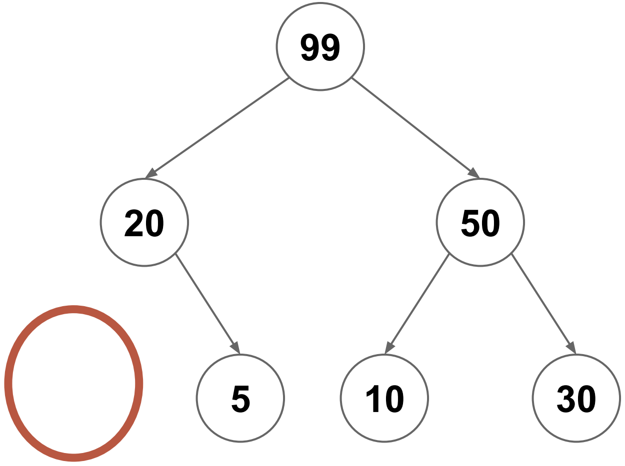
Invalid Max Heaps



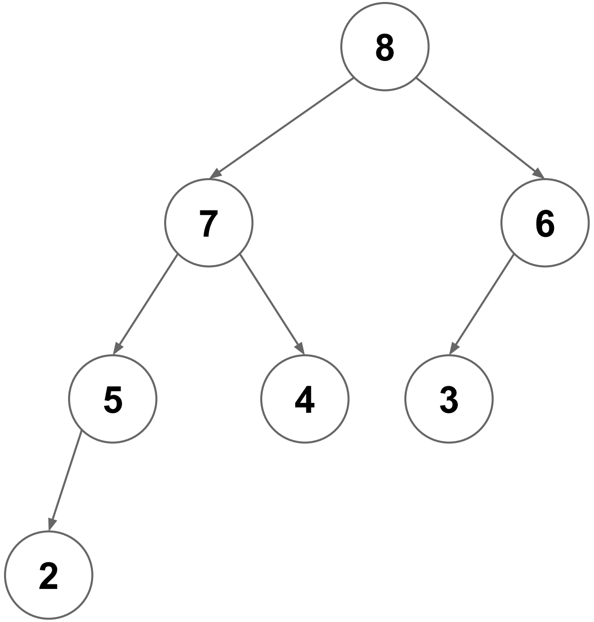


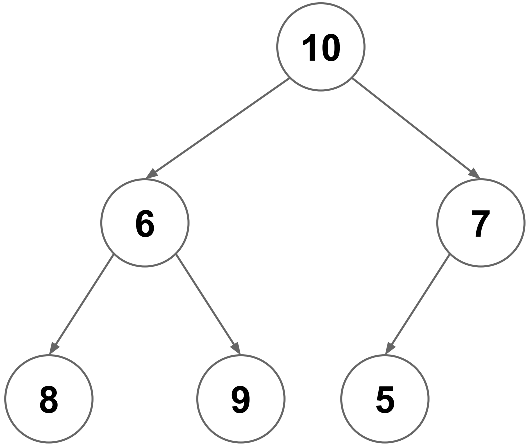


Invalid Max Heaps

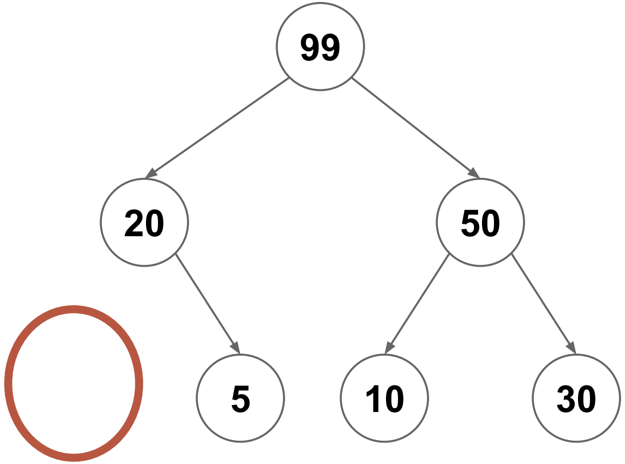


Need to fill from left to right

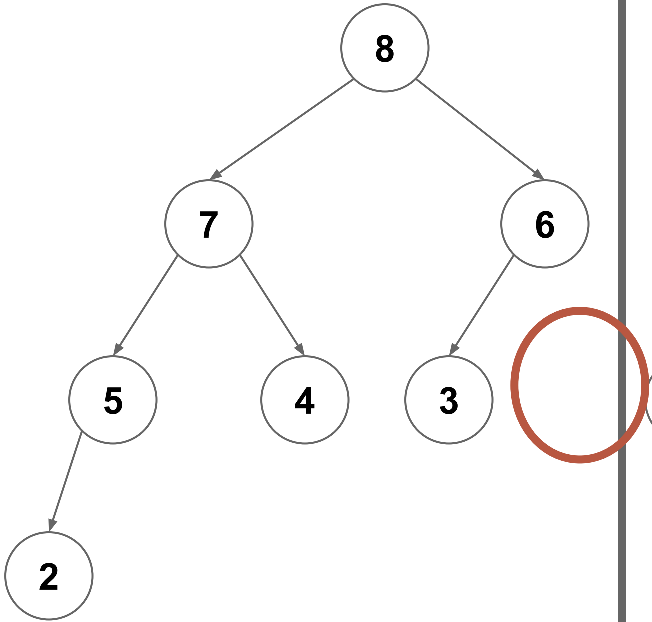




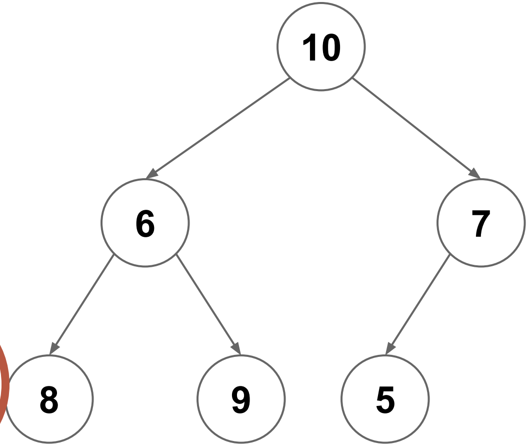
Invalid Max Heaps



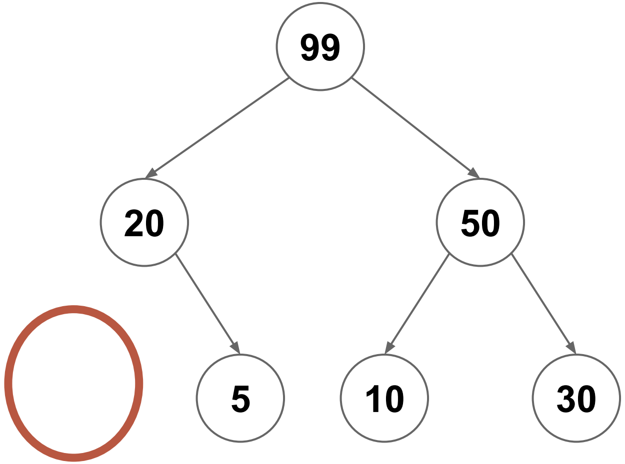
Need to fill from left to right



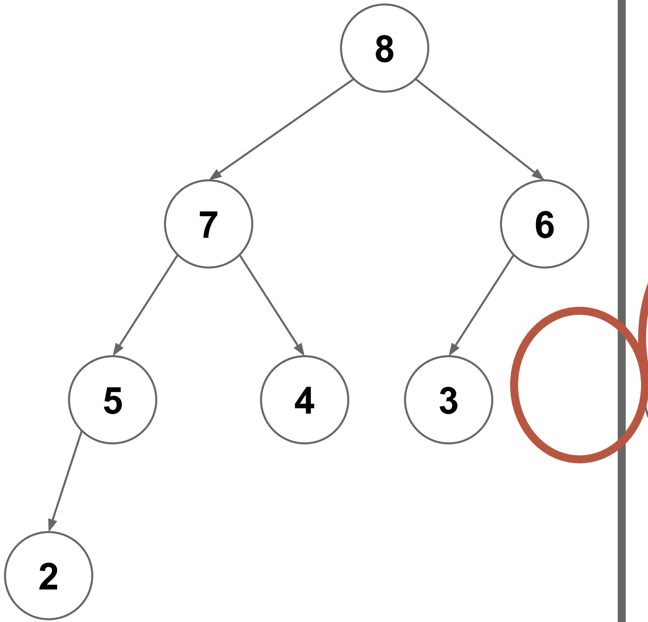
Need complete levels



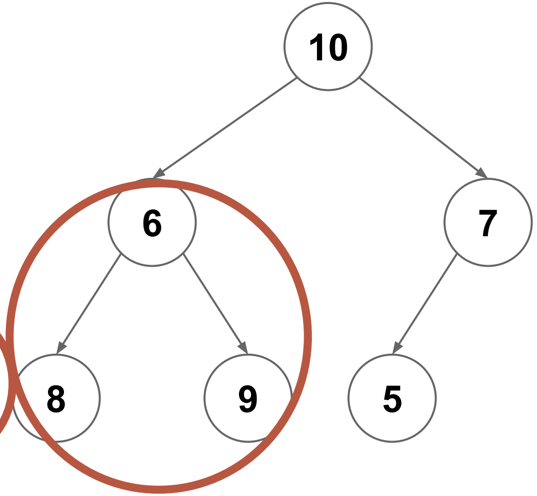
Invalid Max Heaps



Need to fill from left to right



Need complete levels



Children must be less than or equal to parents

Heaps

*What is the depth of a binary heap containing **n** items?*

Level 1: holds up to 1 item

Heaps

*What is the depth of a binary heap containing **n** items?*

Level 1: holds up to 1 item

Level 2: holds up to 2 items

Heaps

*What is the depth of a binary heap containing **n** items?*

Level 1: holds up to 1 item

Level 2: holds up to 2 items

Level 3: holds up to 4 items

Heaps

What is the depth of a binary heap containing ***n*** *items?*

Level 1: holds up to 1 item

Level 2: holds up to 2 items

Level 3: holds up to 4 items

Level 4: holds up to 8 items

Heaps

*What is the depth of a binary heap containing **n** items?*

Level 1: holds up to 1 item

Level 2: holds up to 2 items

Level 3: holds up to 4 items

Level 4: holds up to 8 items

…

Level *i*: holds up to 2^{i-1} items

Heaps

*What is the depth of a binary heap containing **n** items?*

$$n = O\left(\sum_{i=1}^{\ell_{max}} 2^i\right) = O\left(2^{\ell_{max}}\right)$$

Heaps

*What is the depth of a binary heap containing **n** items?*

$$n = O\left(\sum_{i=1}^{\ell_{max}} 2^i\right) = O\left(2^{\ell_{max}}\right)$$

$$\ell_{max} = O\left(\log(n)\right)$$

The `Heap` ADT

`enqueue(elem: A): Unit` *`[AKA pushHeap]`*

Place an item into the heap

`dequeue: A` *`[AKA popHeap]`*

Remove and return the maximal element from the heap

`head: A`

Peek at the maximal element in the heap

`length: Int`

The number of elements in the heap

`Heap.enqueue`

Idea: Insert the element at the next available spot, then fix the heap.

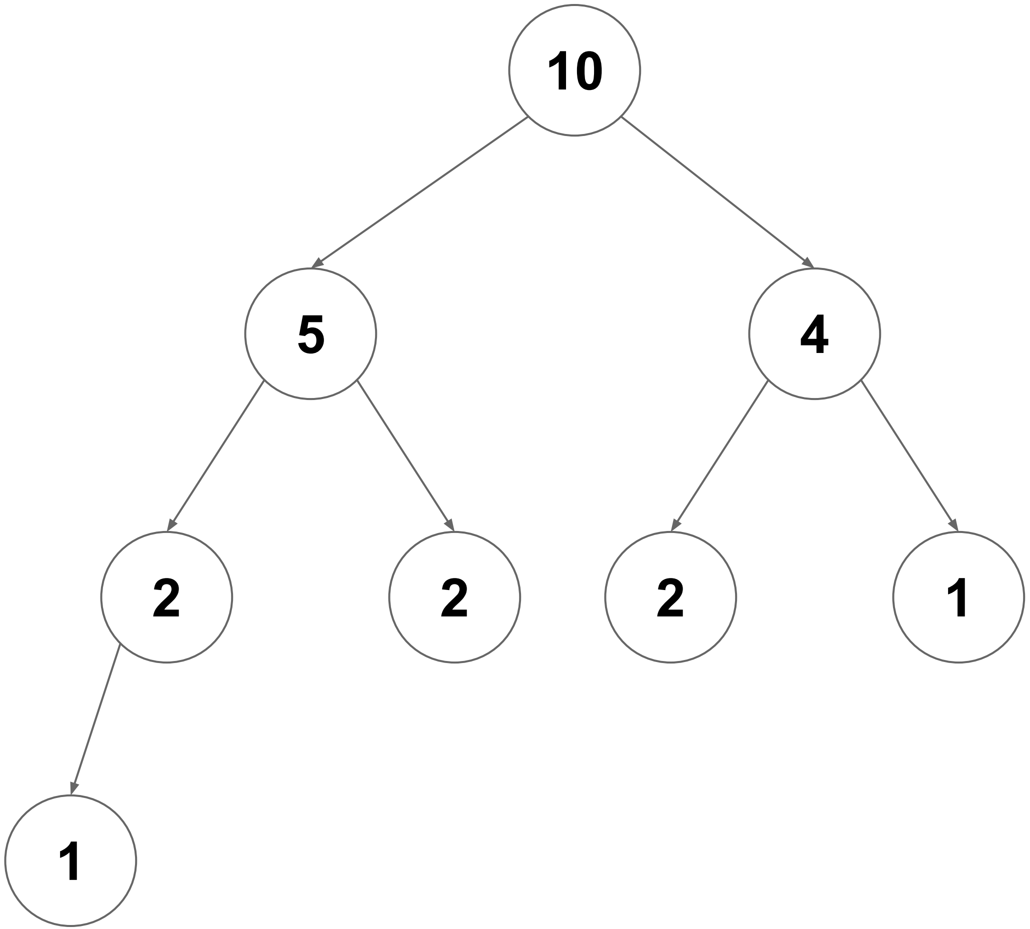
`Heap.enqueue`

Idea: Insert the element at the next available spot, then fix the heap.

1. Call the insertion point **`current`**
2. While **`current != root`** and **`current > parent`**
   a. Swap **`current`** with **`parent`**
   b. Repeat with **`current`** $\leftarrow$ **`parent`**

`Heap.enqueue`

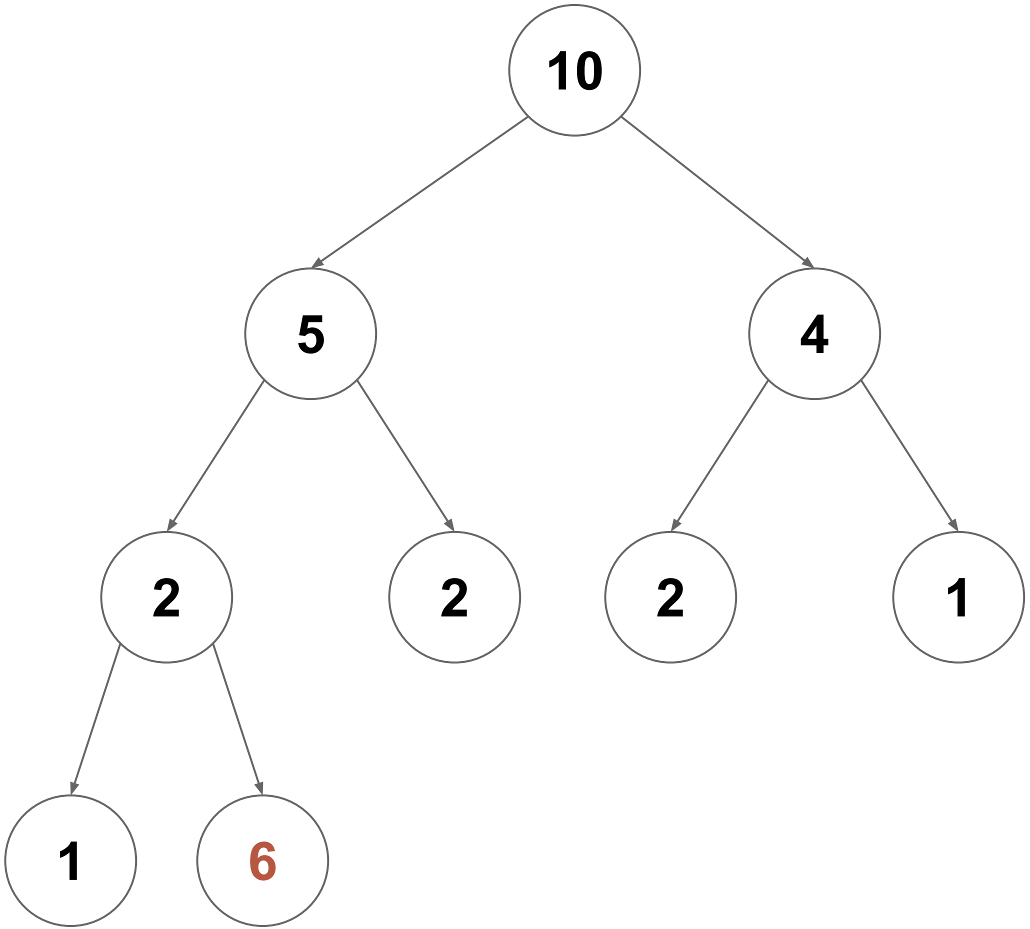
What if we enqueue 6?



`Heap.enqueue`

What if we enqueue 6?

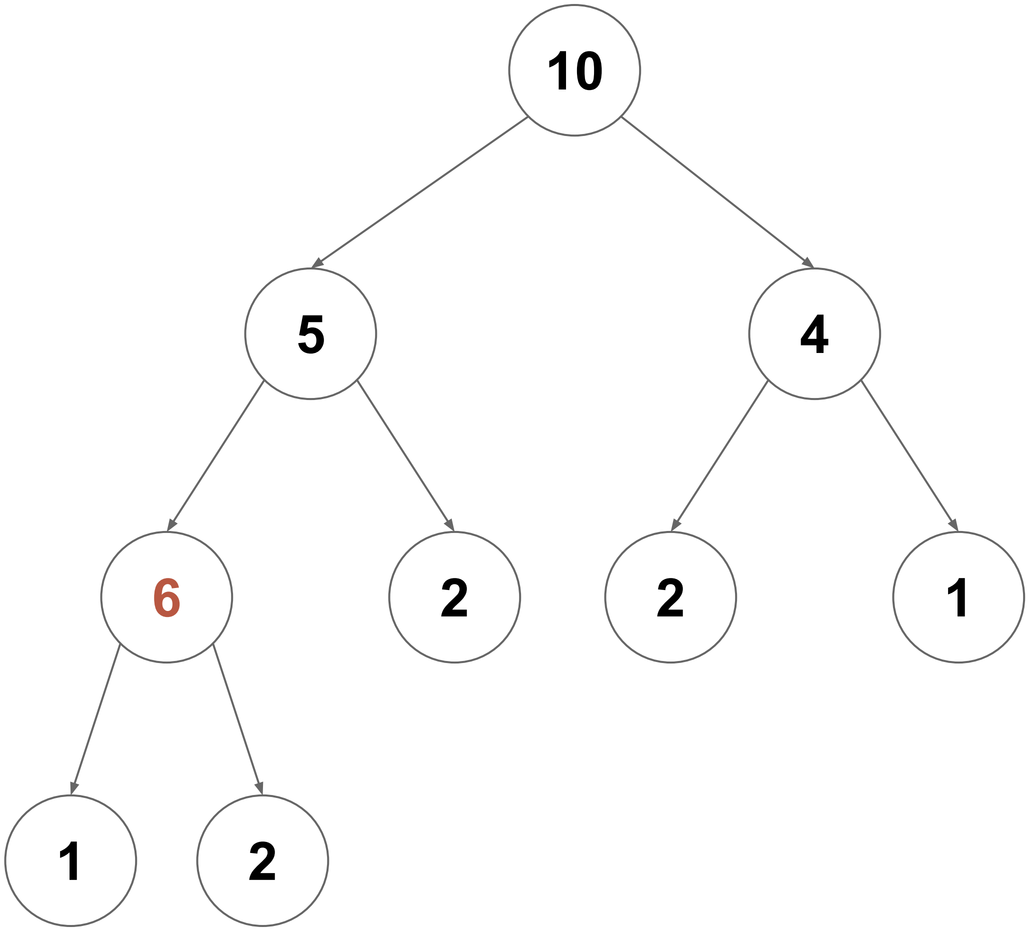
Place in the next available spot



Heap.enqueue

What if we enqueue 6?

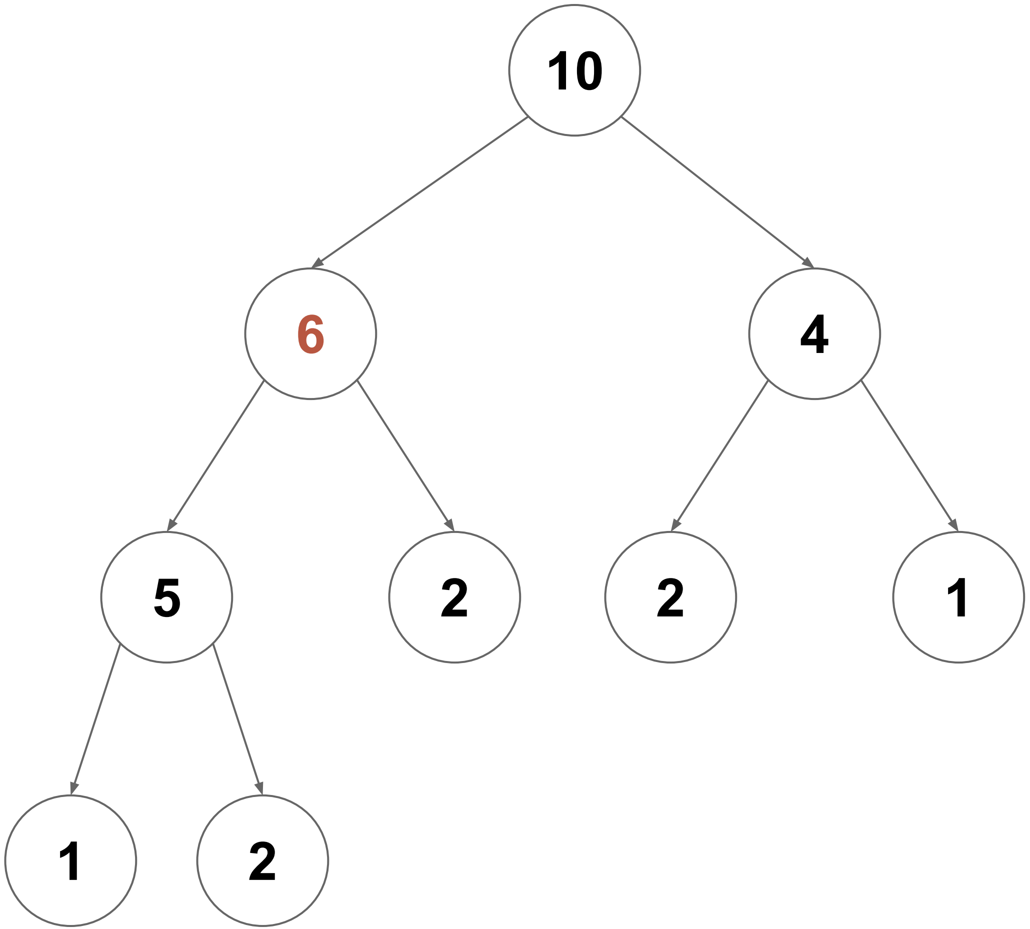
Swap with parent if it is bigger than the parent



Heap.enqueue

What if we enqueue 6?

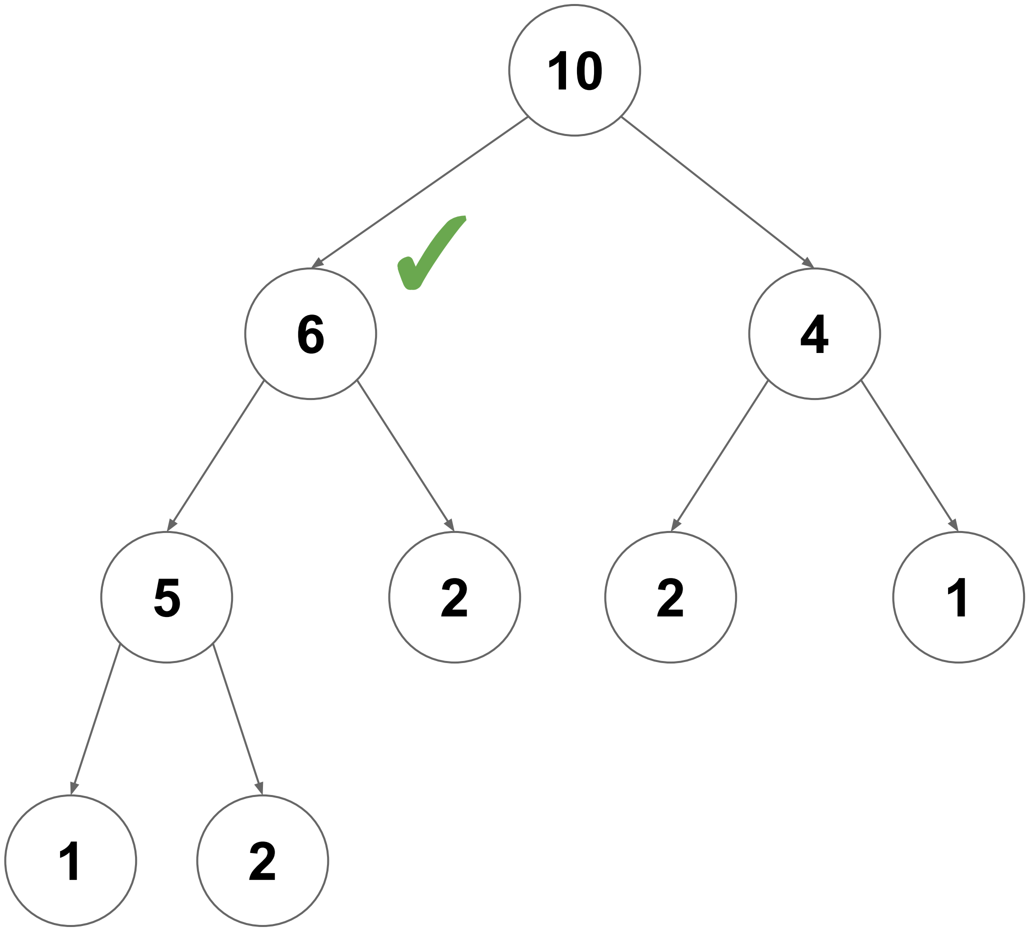
Continue swapping upwards…



Heap.enqueue

What if we enqueue 6?

Stop swapping when we are no longer bigger than our parent



Heap.enqueue - fixUp

```
def fixUp[A: Ordering](current: Vertex[A]): Unit = {
  if(current.parent.isDefined){
    val parent = current.parent.get
    if( Ordering[A].lt( parent.value, current.value ) ){
      swap(current.value, parent.value)
      fixUp(parent)
    }
  }
}
```

Heap.enqueue - fixUp

```
def fixUp[A: Ordering](current: Vertex[A]): Unit = {
  if(current.parent.isDefined){
    val parent = current.parent.get
    if( Ordering[A].lt( parent.value, current.value ) ){
      swap(current.value, parent.value)
      fixUp(parent)
    }
  }
}
```

What is the complexity (or how many swaps occur)?

Heap.enqueue - fixUp

```
def fixUp[A: Ordering](current: Vertex[A]): Unit = {
  if(current.parent.isDefined){
    val parent = current.parent.get
    if( Ordering[A].lt( parent.value, current.value ) ){
      swap(current.value, parent.value)
      fixUp(parent)
    }
  }
}
```

What is the complexity (or how many swaps occur)? **O(log(*n*))**

Heap.dequeue

`Heap.dequeue`

Idea: Replace root with the last element then fix the heap

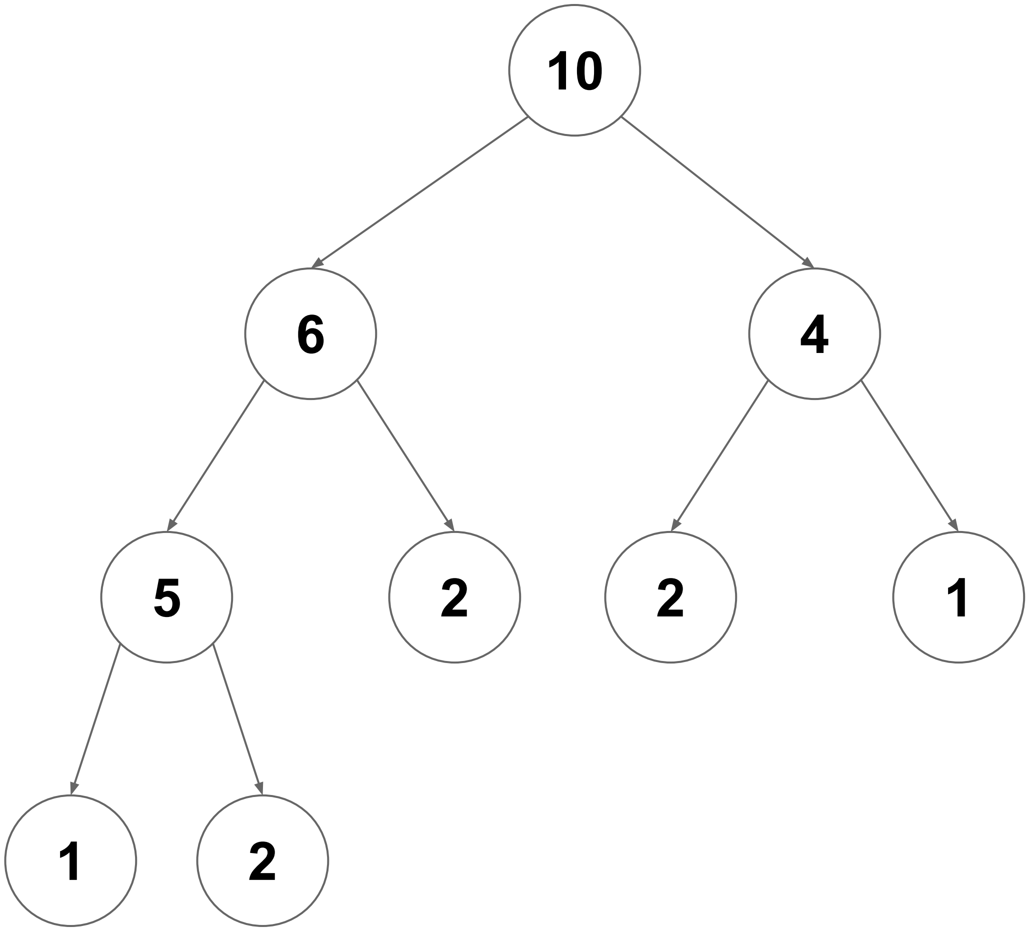
`Heap.dequeue`

Idea: Replace root with the last element then fix the heap

1. Start with `current ← root`
2. While `current` has a `child > current`
    a. Swap `current` with its largest `child`
    b. Repeat with `current ← child`

Heap.dequeue

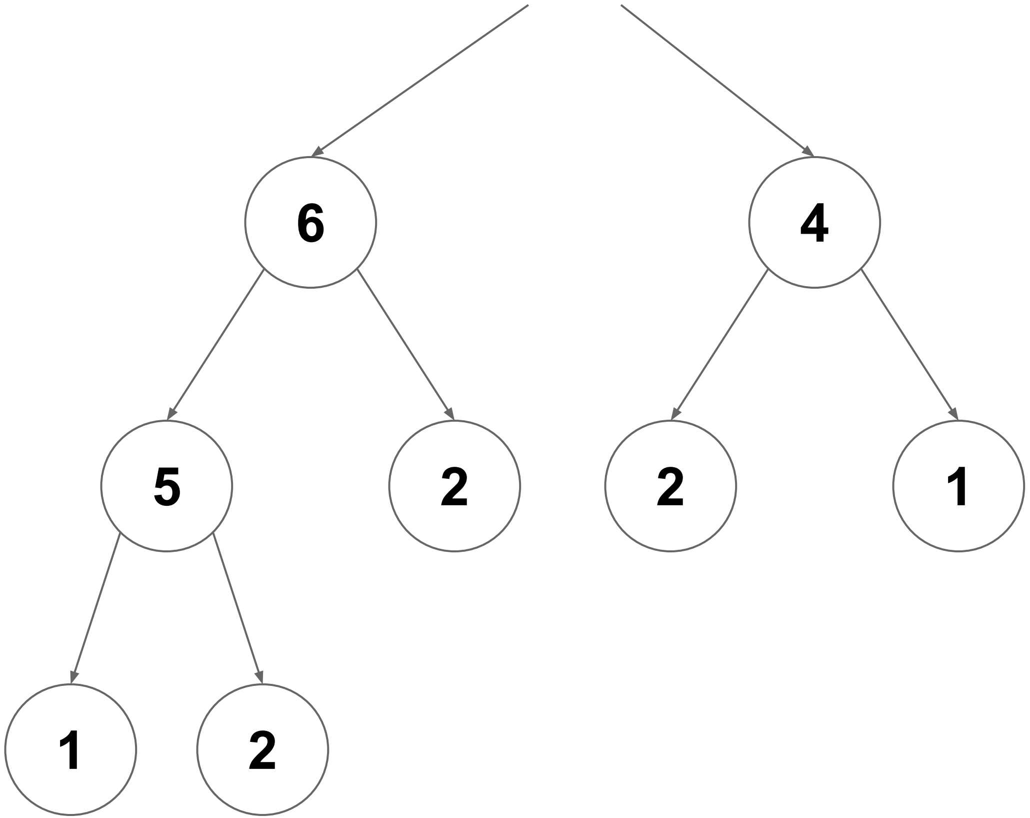
What if we call dequeue?



`Heap.dequeue`

What if we call dequeue?

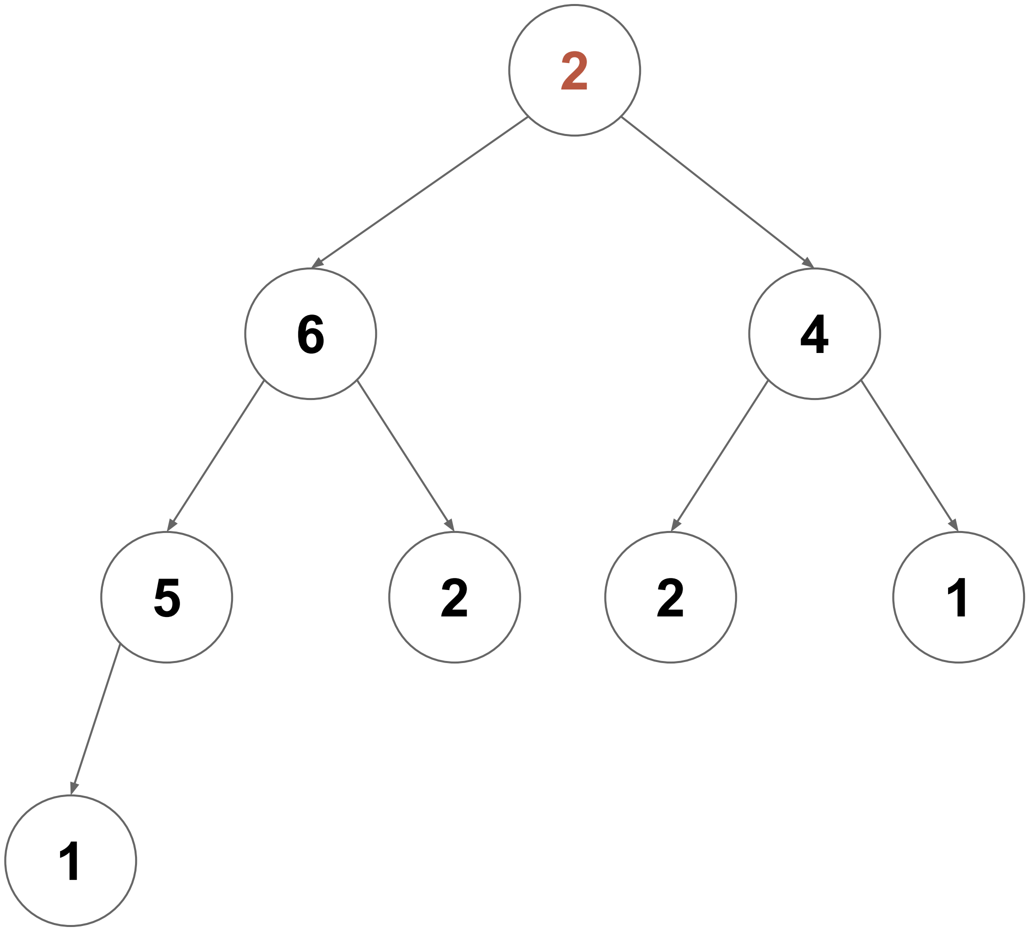
Remove and return the root



Heap.dequeue

What if we call dequeue?

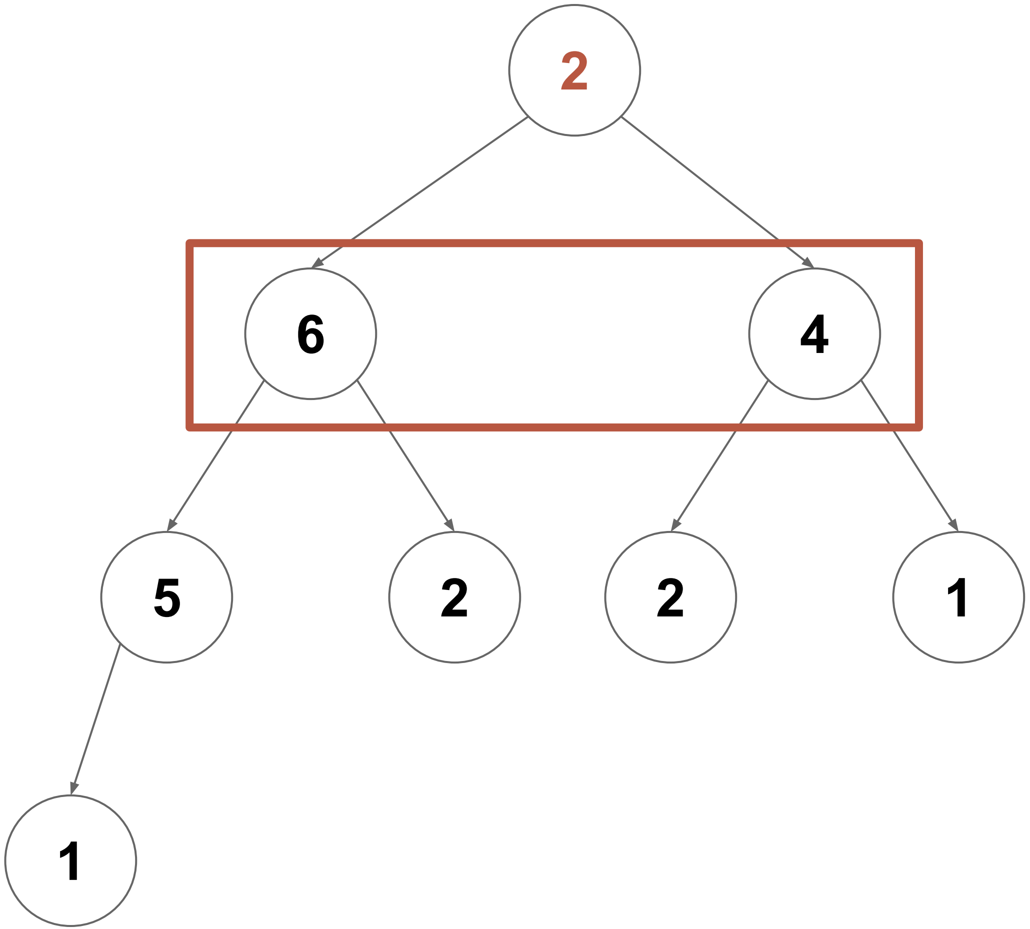
Make the last item the new root



Heap.dequeue

What if we call dequeue?

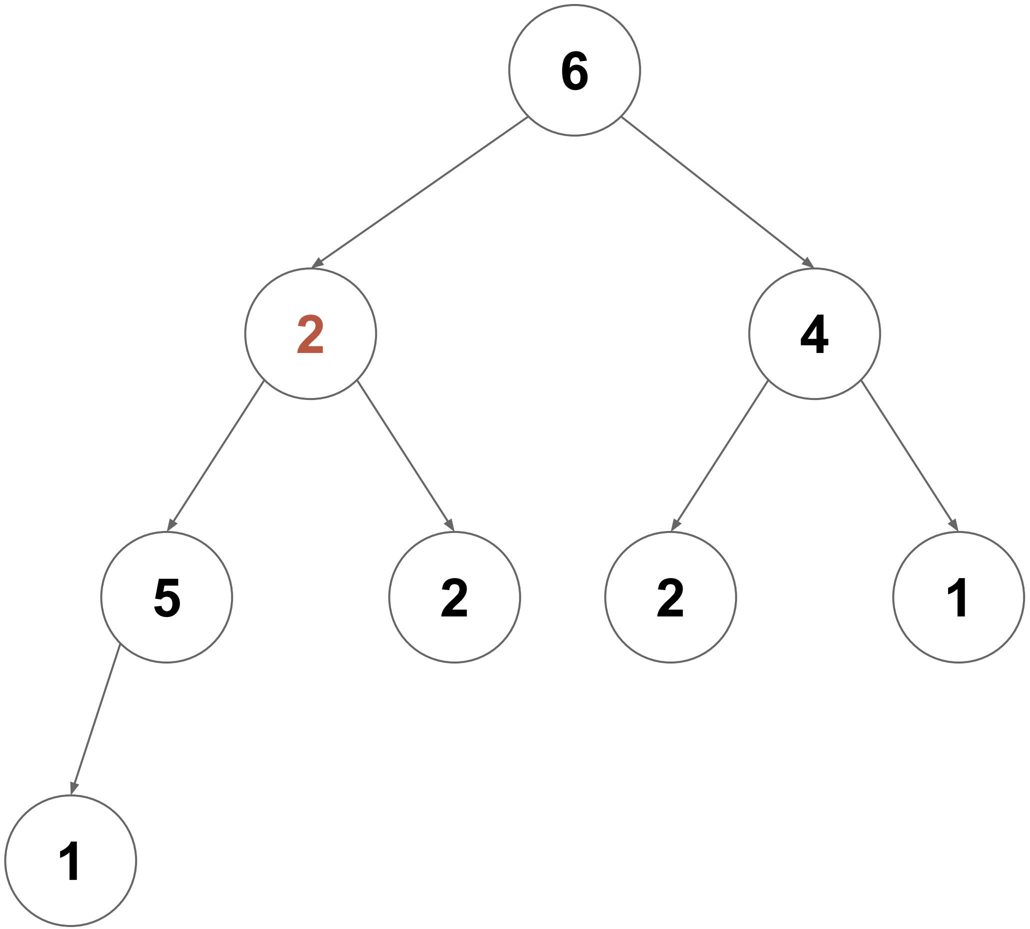
Check for our largest child



Heap.dequeue

What if we call dequeue?

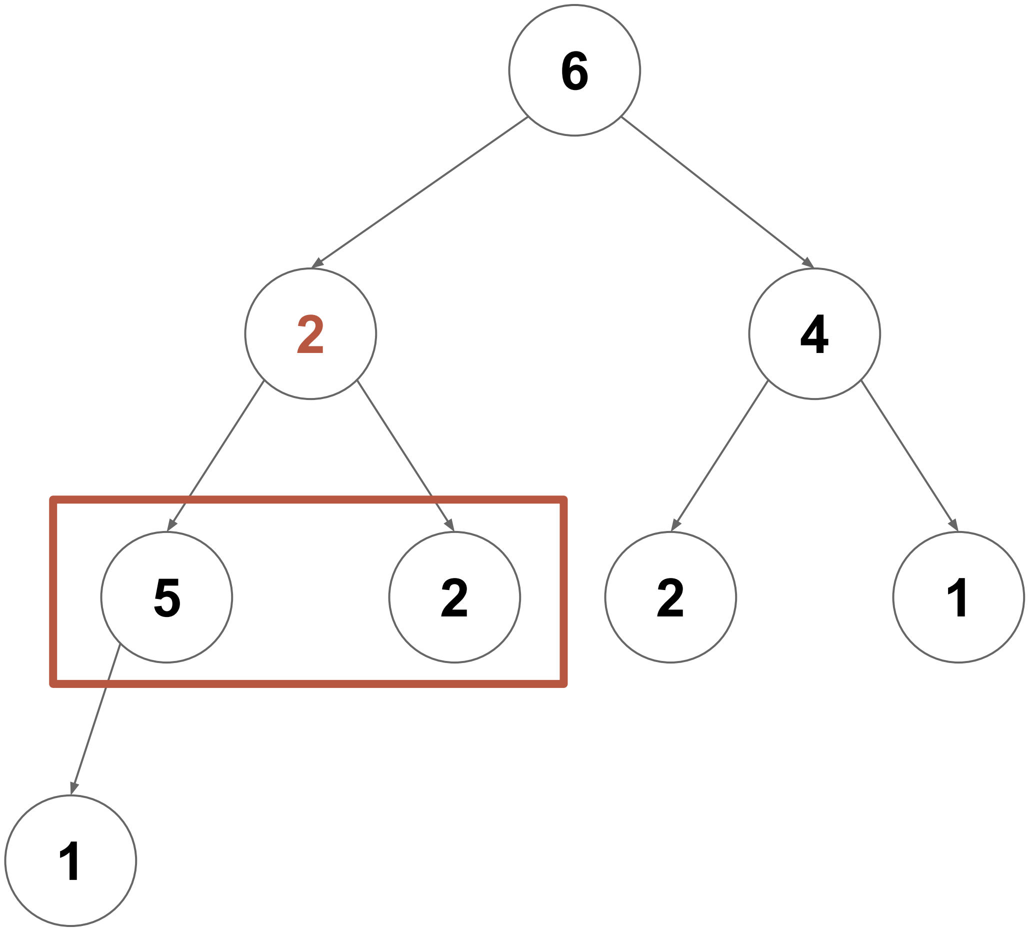
If the largest child is bigger than us, swap



Heap.dequeue

What if we call dequeue?

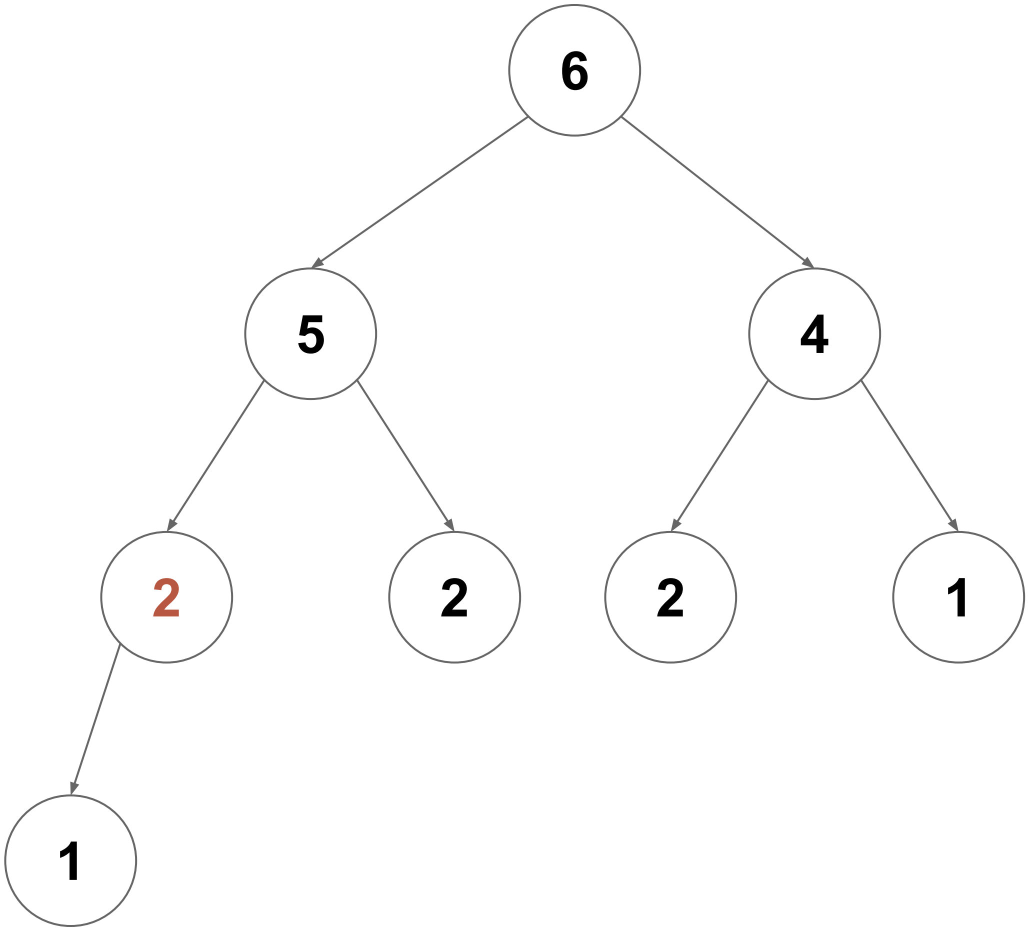
Continue swapping down the tree as necessary…



Heap.dequeue

What if we call dequeue?

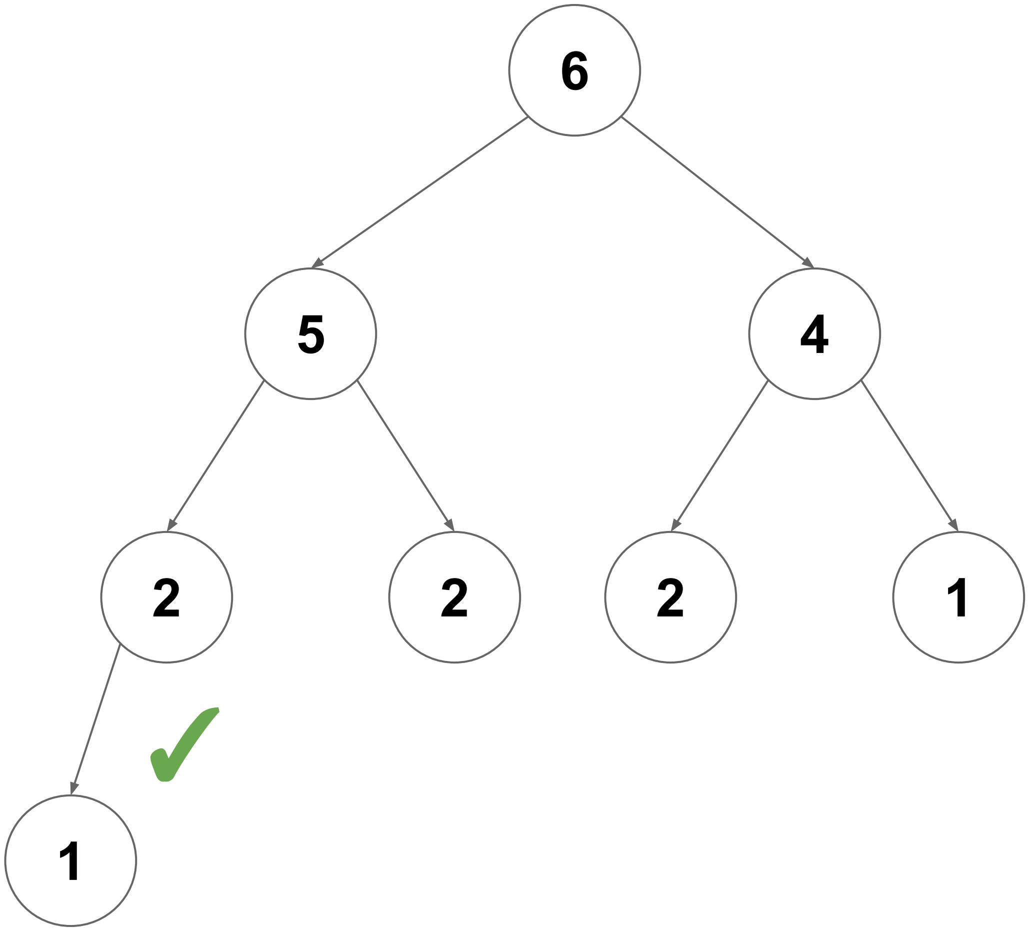
Continue swapping down the tree as necessary…



Heap.dequeue

What if we call dequeue?

Stop swapping when our children are no longer bigger



Heap.dequeue - fixDown

```
def fixDown[A: Ordering](current: Vertex[A]): Unit = {
  val maxChild = getMaxChildOf(current)
  if( maxChild.isDefined ) {
    val max = maxChild.get
    if( Ordering[A].lt( current.value, max.value ) ){
      swap(current.value, max.value);
      fixDown(max)
    }
  }
}
```

Heap.dequeue - fixDown

```
def fixDown[A: Ordering](current: Vertex[A]): Unit = {
  val maxChild = getMaxChildOf(current)
  if( maxChild.isDefined ) {
    val max = maxChild.get
    if( Ordering[A].lt( current.value, max.value ) ){
      swap(current.value, max.value);
      fixDown(max)
    }
  }
}
```

What is the complexity (or how many swaps occur)?

Heap.dequeue - fixDown

```
def fixDown[A: Ordering](current: Vertex[A]): Unit = {
  val maxChild = getMaxChildOf(current)
  if( maxChild.isDefined ) {
    val max = maxChild.get
    if( Ordering[A].lt( current.value, max.value ) ){
      swap(current.value, max.value);
      fixDown(max)
    }
  }
}
```

What is the complexity (or how many swaps occur)? ***O*(log(*n*))**

Priority Queues

Operation	Lazy	Proactive	Heap
enqueue	$O(1)$	$O(n)$	$O(\log(n))$
dequeue	$O(n)$	$O(1)$	$O(\log(n))$
head	$O(n)$	$O(1)$	$O(1)$

Storing heaps

Notice that:

1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

How can we compactly store a heap?

Storing heaps

Notice that:

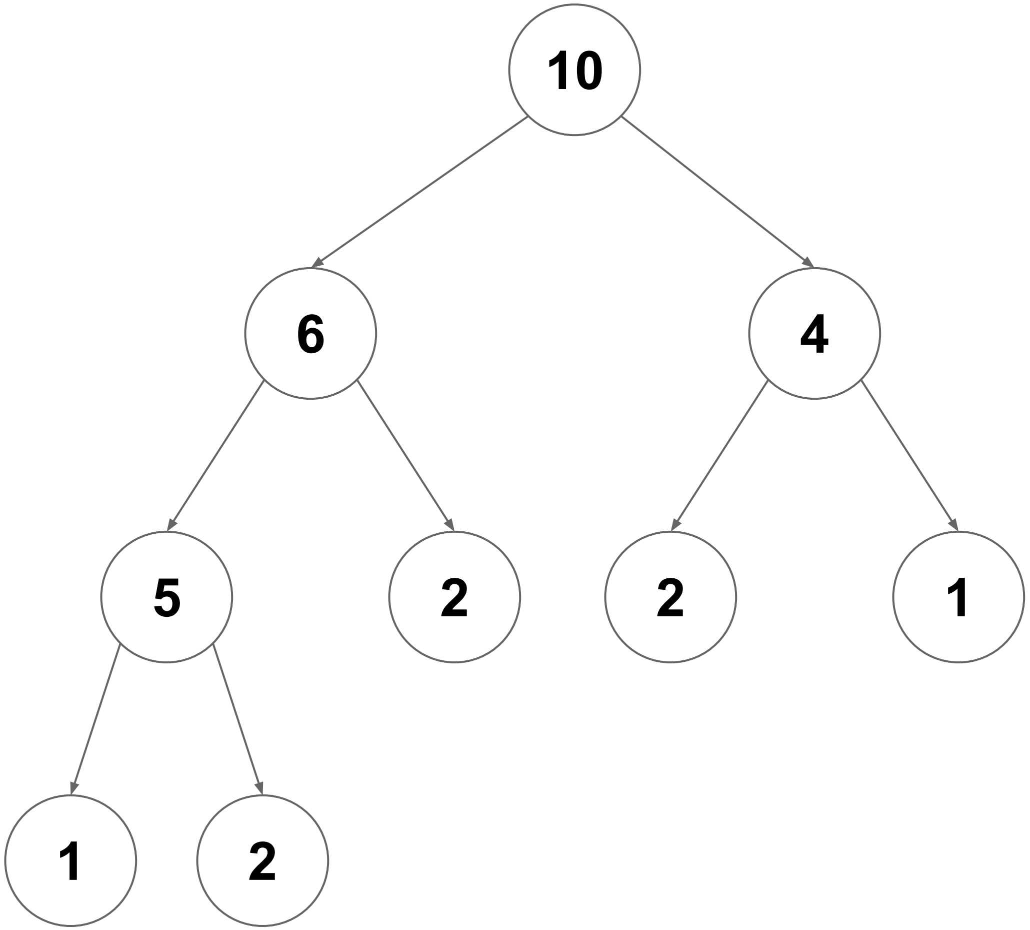
1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

How can we compactly store a heap?

Idea: Use an `ArrayBuffer`

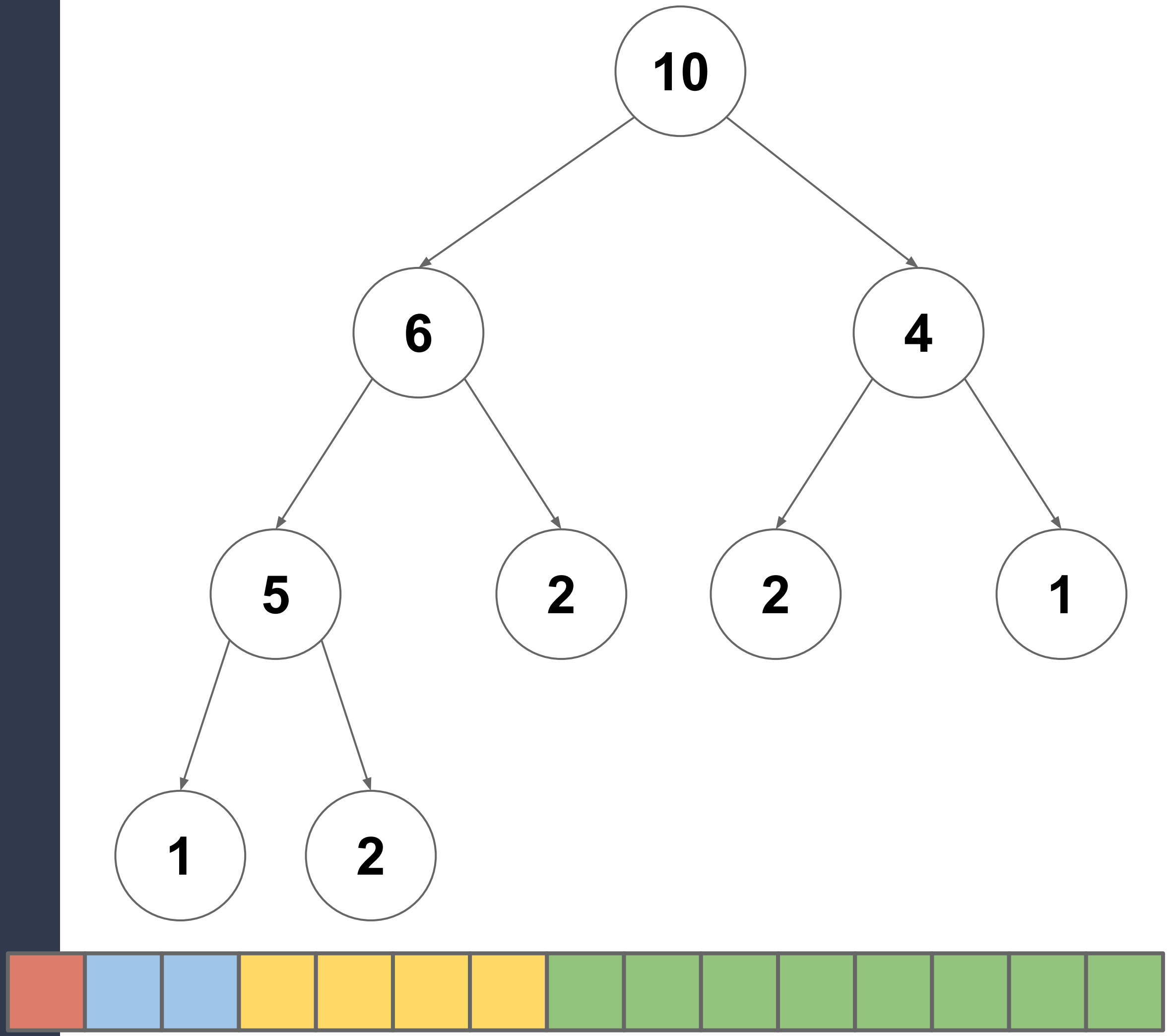
Storing Heaps

How can we store this heap in an array buffer?



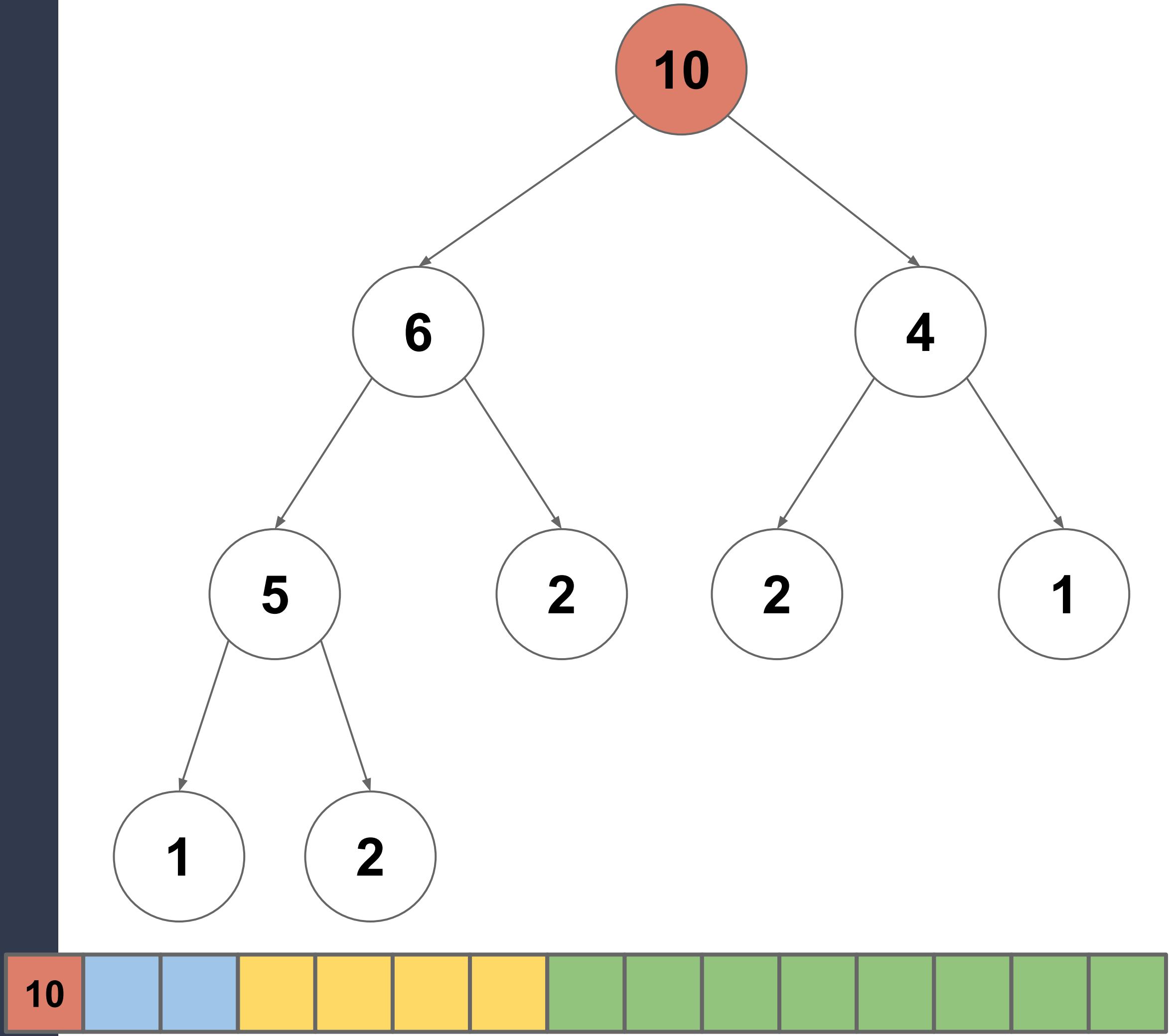
Storing Heaps

How can we store this heap in an array buffer?



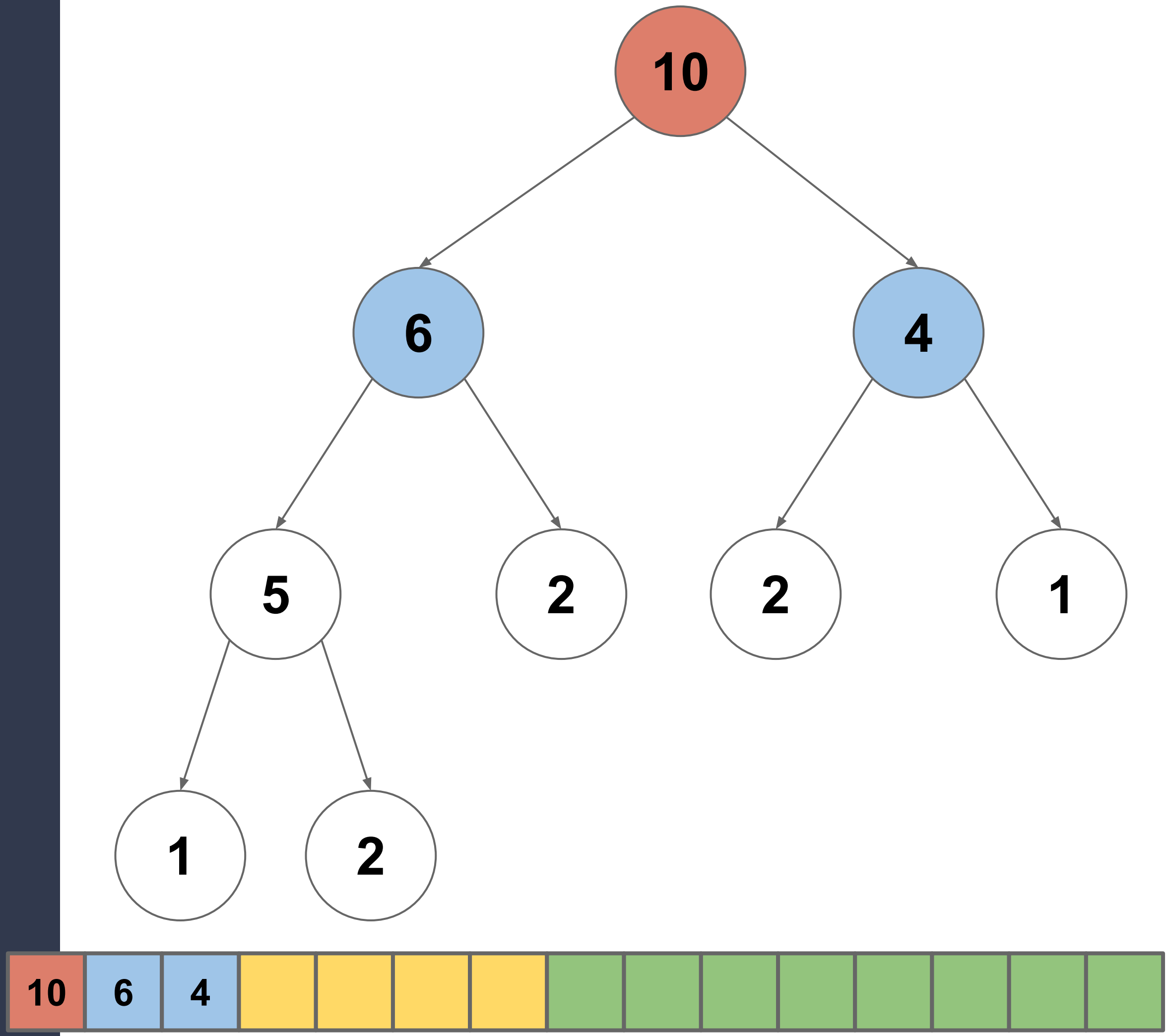
Storing Heaps

How can we store this heap in an array buffer?



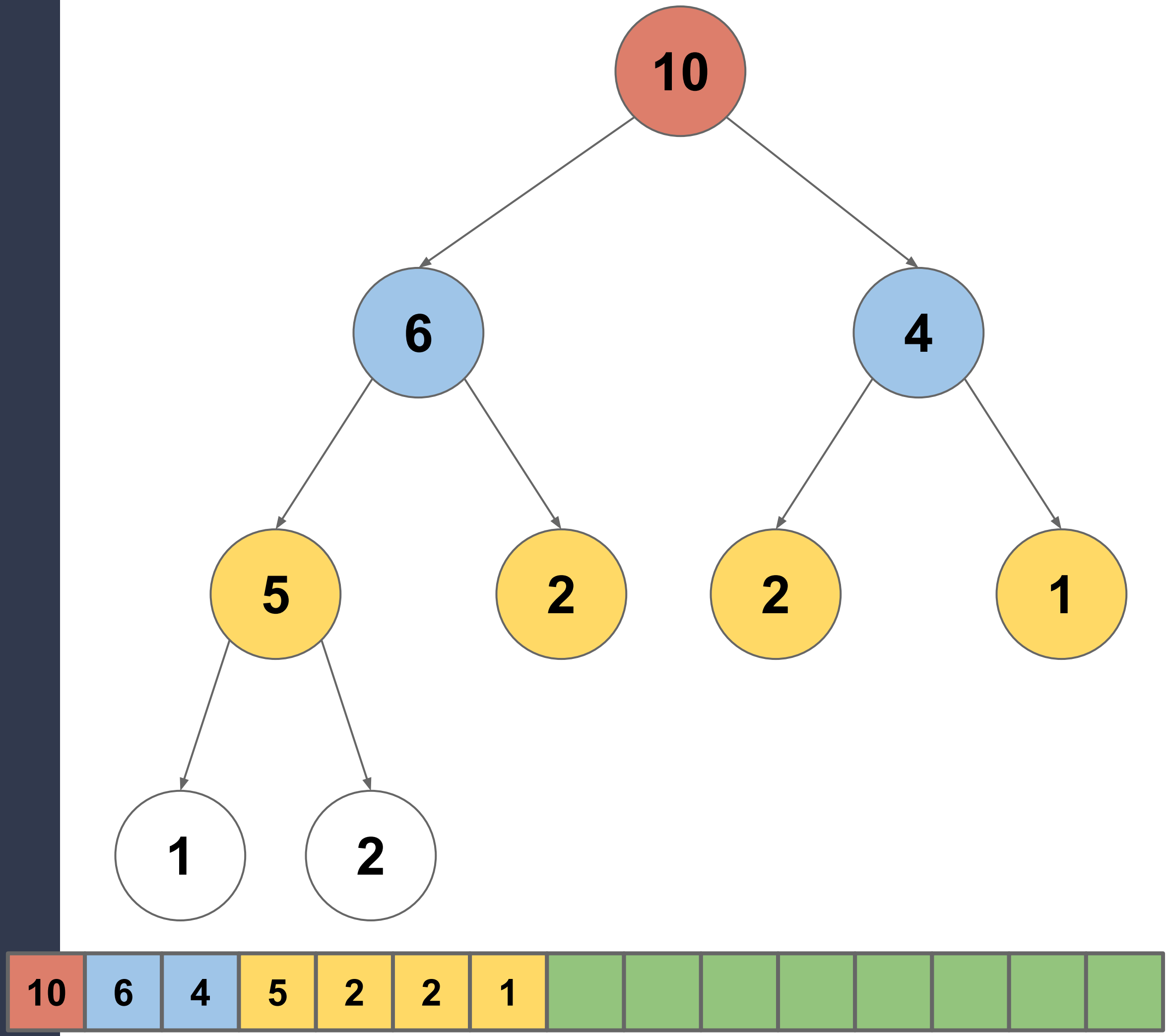
Storing Heaps

How can we store this heap in an array buffer?



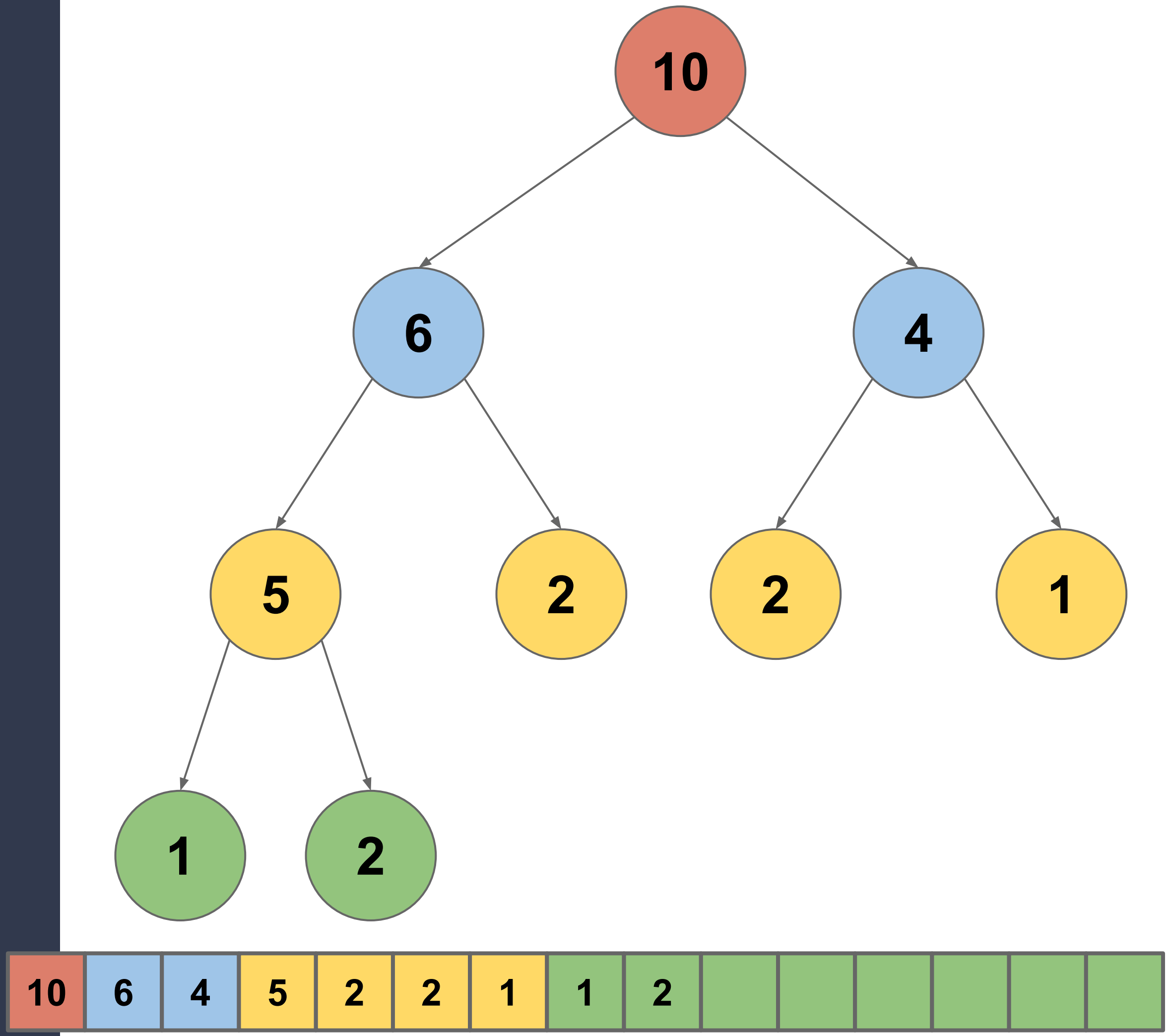
Storing Heaps

How can we store this heap in an array buffer?



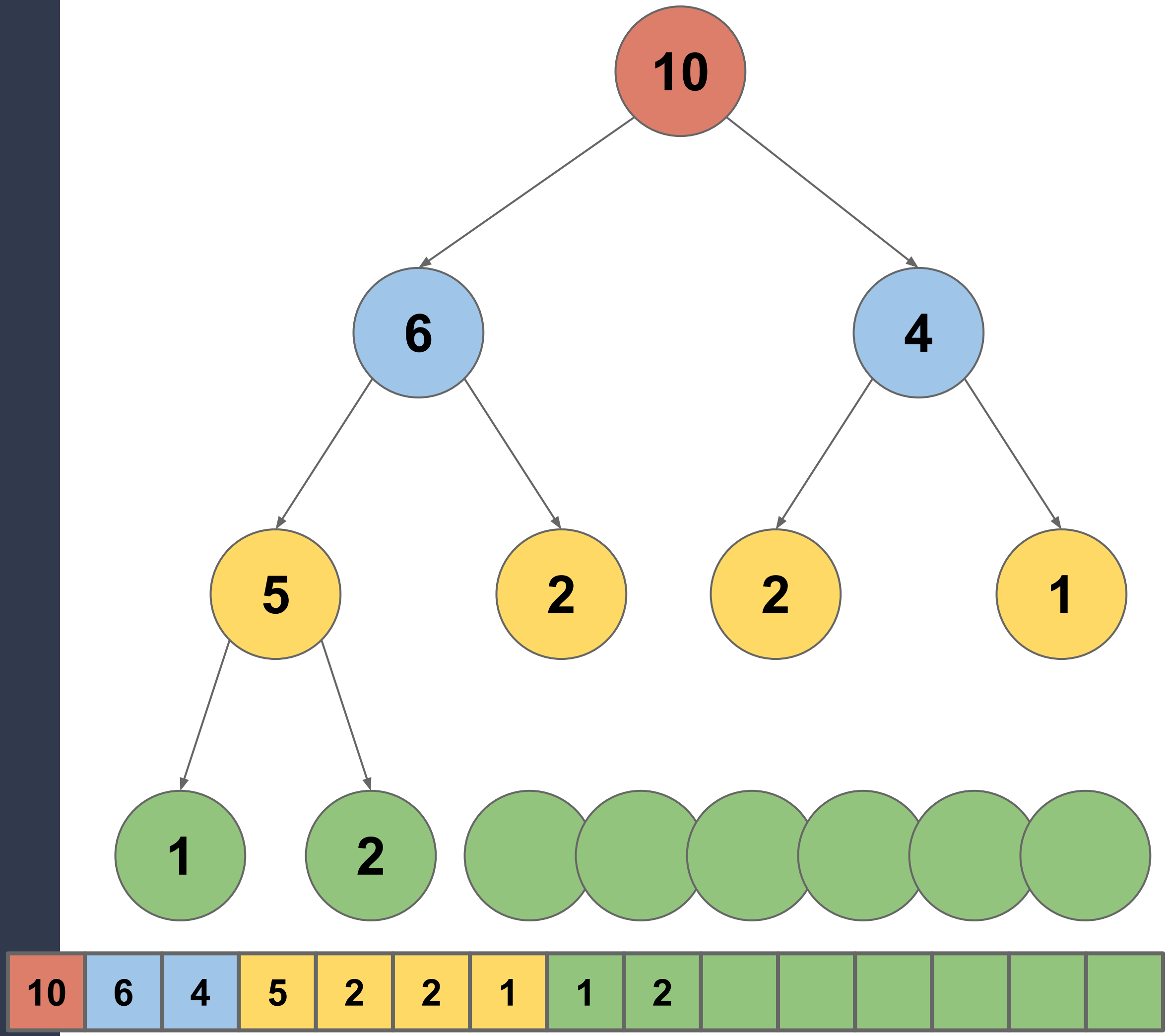
Storing Heaps

How can we store this heap in an array buffer?



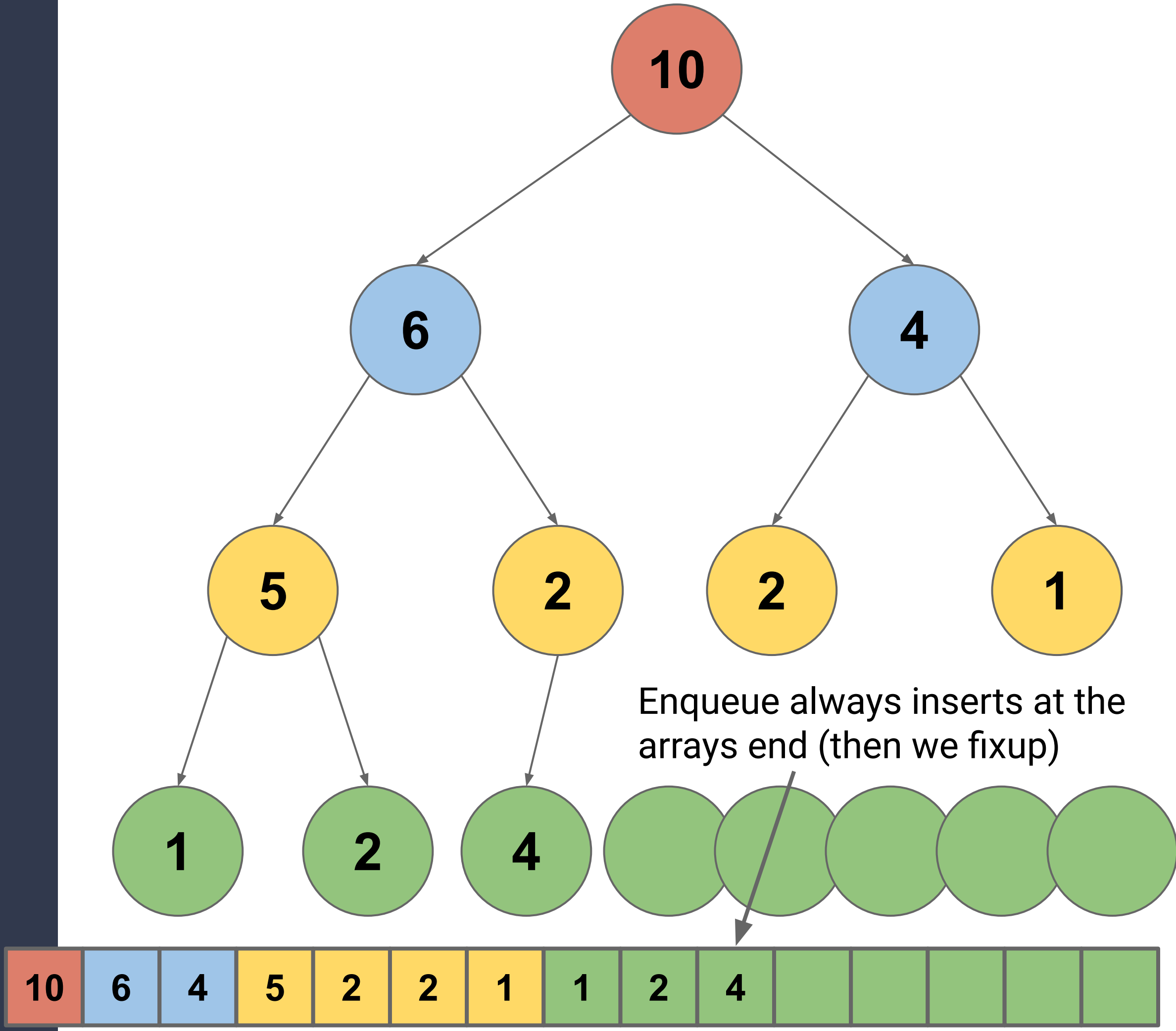
Storing Heaps

How can we store this heap in an array buffer?



Storing Heaps

How can we store this heap in an array buffer?



Runtime Analysis

`enqueue`

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`**: amortized $O(1)$ *(unqualified $O(n)$)*

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`**: amortized $O(1)$ *(unqualified $O(n)$)*
- **`fixUp`**: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`:** amortized $O(1)$ *(unqualified $O(n)$)*
- **`fixUp`:** $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** amortized $O(\log(n))$ *(unqualified $O(n)$)*

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`:** amortized $O(1)$ *(unqualified $O(n)$)*
- **`fixUp`:** $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** amortized $O(\log(n))$ *(unqualified $O(n)$)*

`dequeue`

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`:** amortized $O(1)$ *(unqualified $O(n)$)*
- **`fixUp`:** $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** amortized $O(\log(n))$ *(unqualified $O(n)$)*

`dequeue`

- **Remove end of `ArrayBuffer`:** $O(1)$

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`**: amortized $O(1)$ *(unqualified $O(n)$)*
- **`fixUp`**: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** amortized $O(\log(n))$ *(unqualified $O(n)$)*

`dequeue`

- **Remove end of `ArrayBuffer`**: $O(1)$
- **`fixDown`**: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$

Runtime Analysis

`enqueue`

- **Append to `ArrayBuffer`**: amortized $O(1)$ *(unqualified $O(n)$)*
- `fixUp`: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** amortized $O(\log(n))$ *(unqualified $O(n)$)*

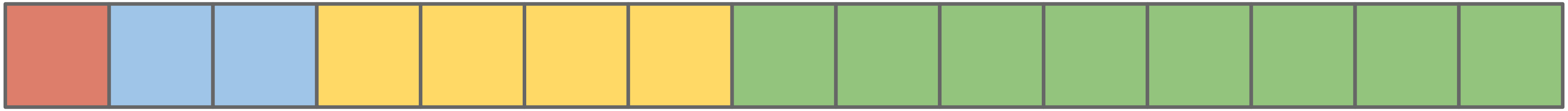
`dequeue`

- **Remove end of `ArrayBuffer`**: $O(1)$
- `fixDown`: $O(\log(n))$ fixes, each one costs $O(1) = O(\log(n))$
- **Total:** worst-case $O(\log(n))$

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

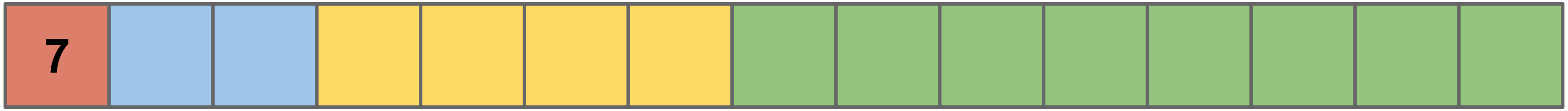
7, 4, 8, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

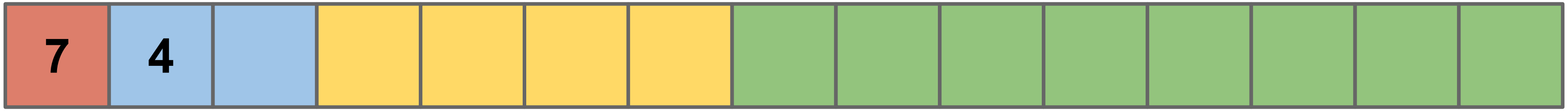
7, 4, 8, 2, 5, 3, 9



Heap Sort

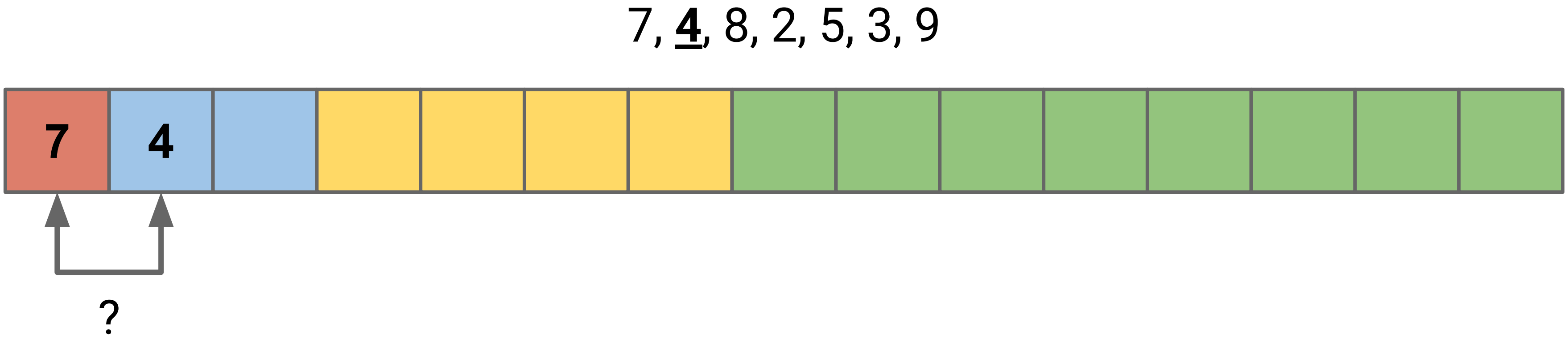
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, **<u>4</u>**, 8, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

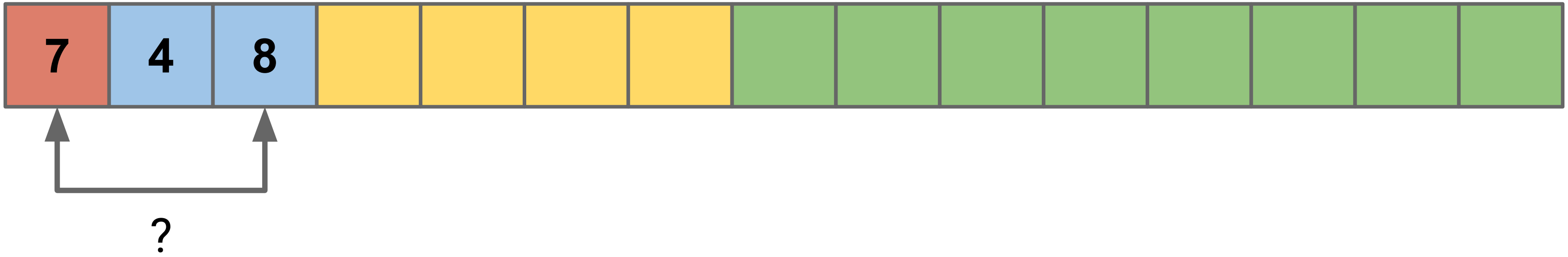
7, 4, **8**, 2, 5, 3, 9

7	**4**	**8**												

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

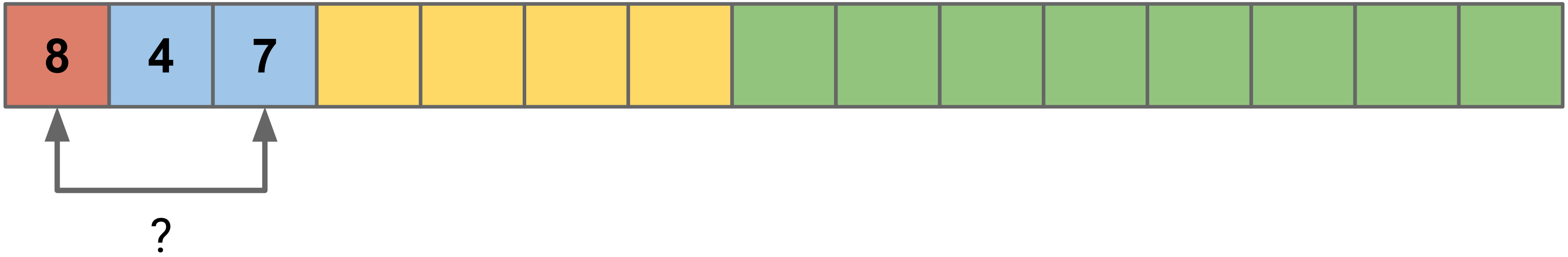
7, 4, **8**, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

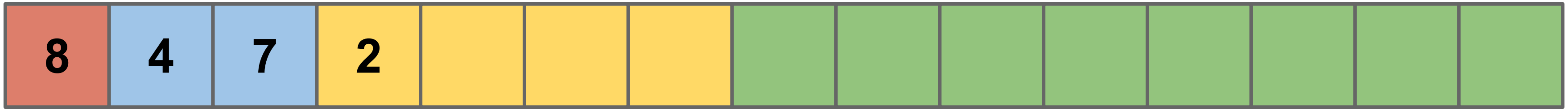
7, 4, **8**, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

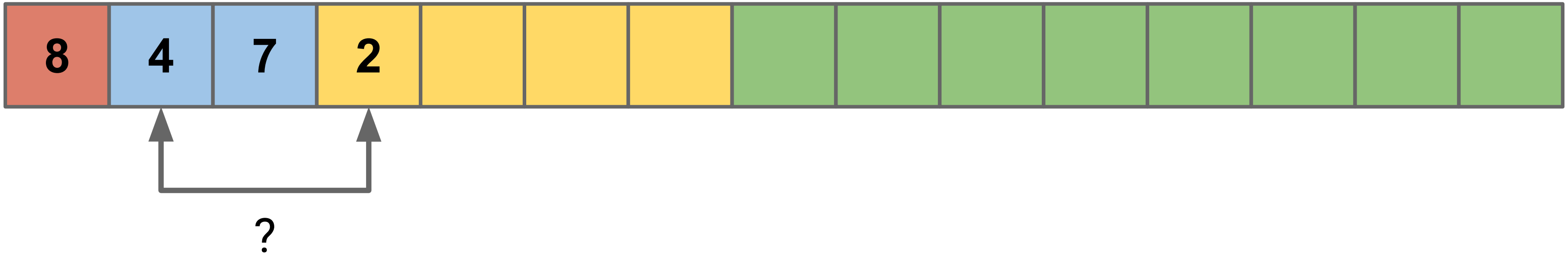
7, 4, 8, **2**, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

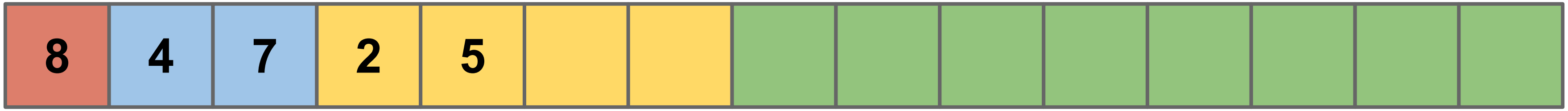
7, 4, 8, **2**, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

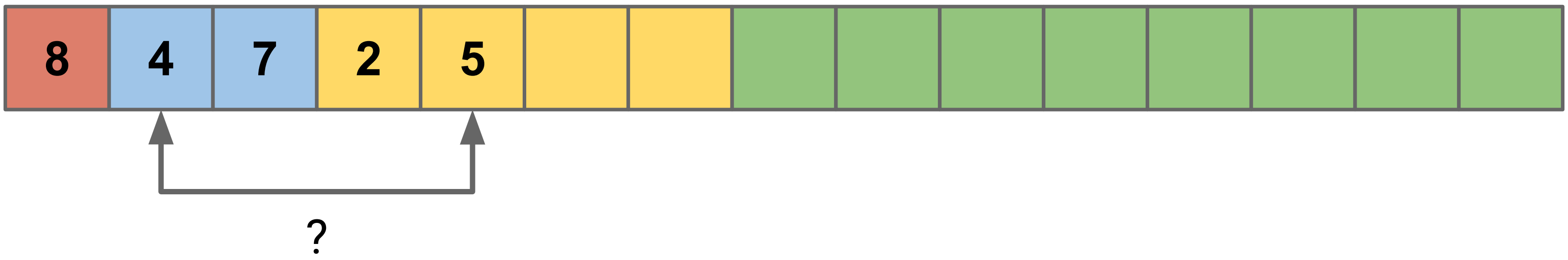
7, 4, 8, 2, <u>**5**</u>, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

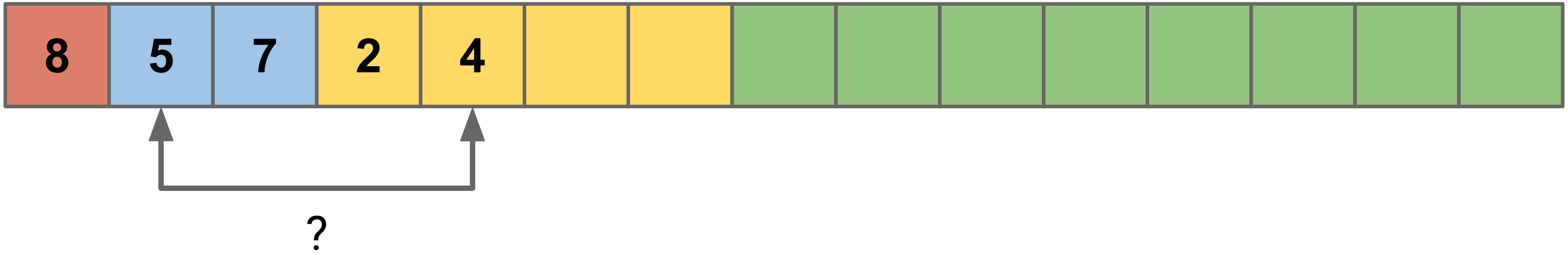
7, 4, 8, 2, **5**, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

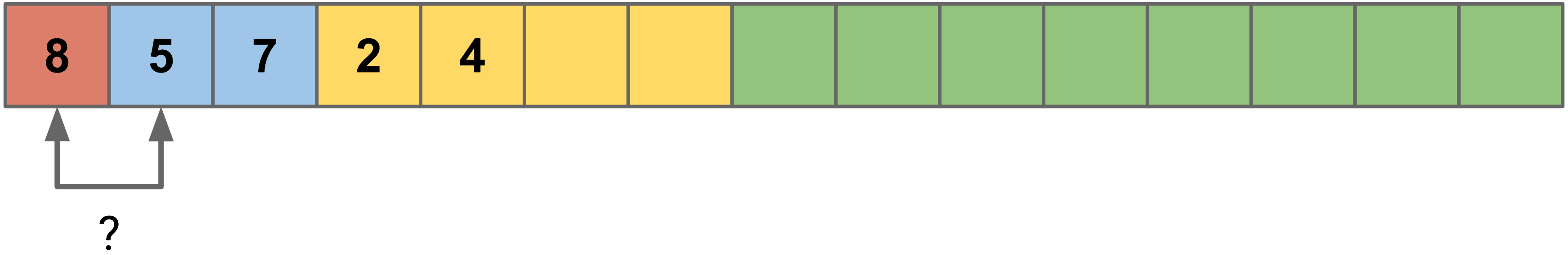
7, 4, 8, 2, **5**, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, **5**, 3, 9



Heap Sort

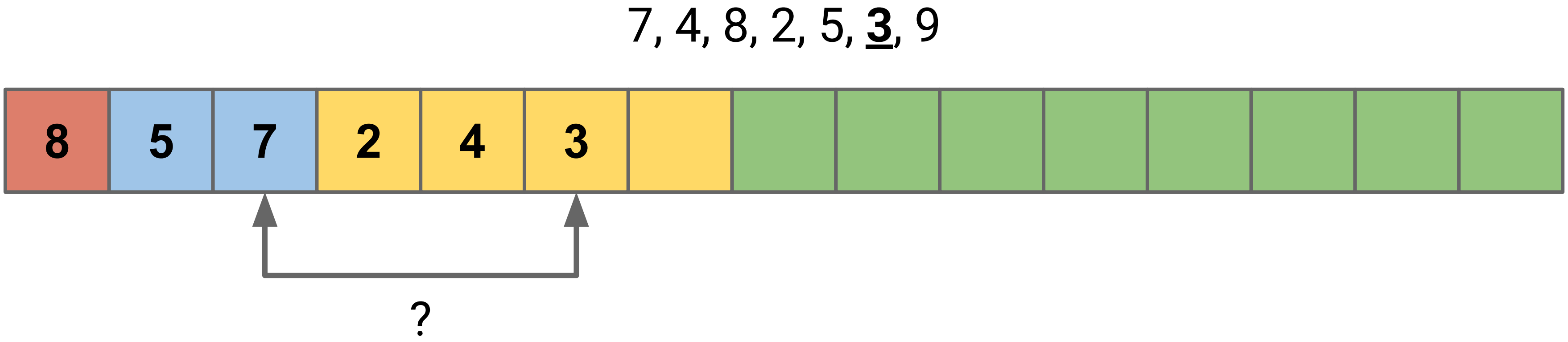
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, <u>**3**</u>, 9

8	**5**	**7**	**2**	**4**	**3**									

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

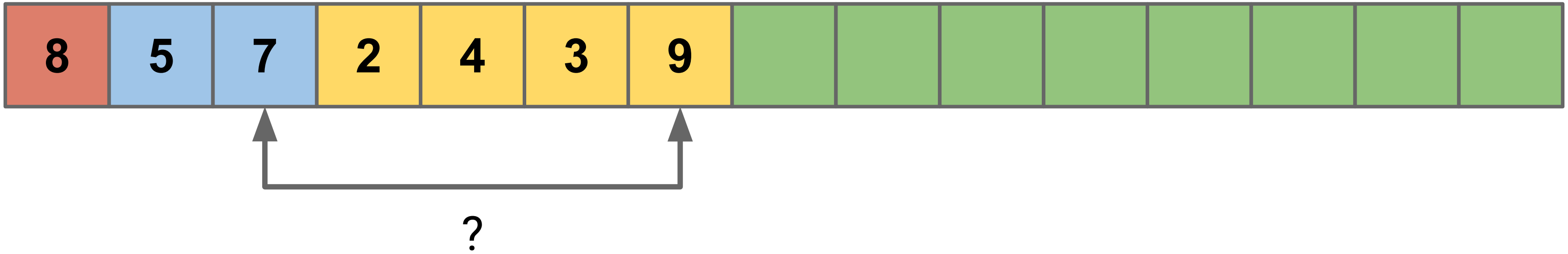
7, 4, 8, 2, 5, 3, **9**

8	5	7	2	4	3	9								

Heap Sort

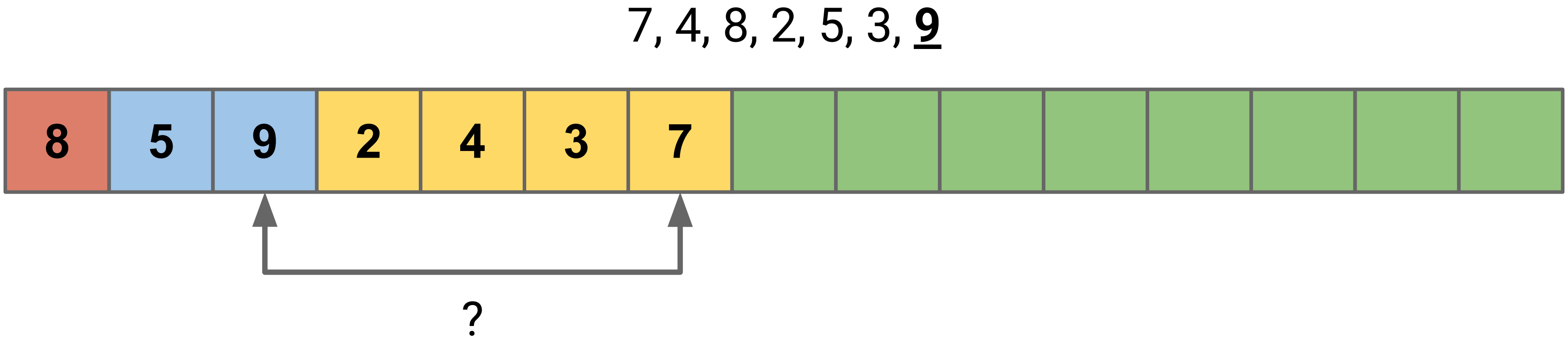
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, **9**



Heap Sort

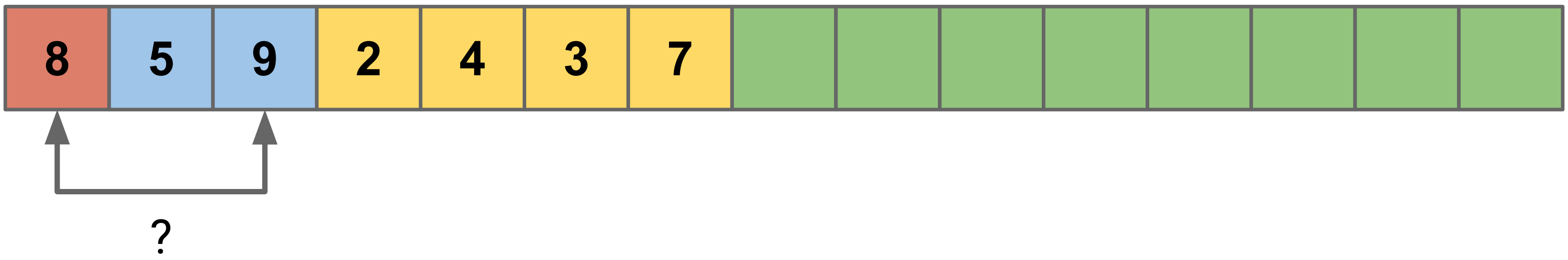
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

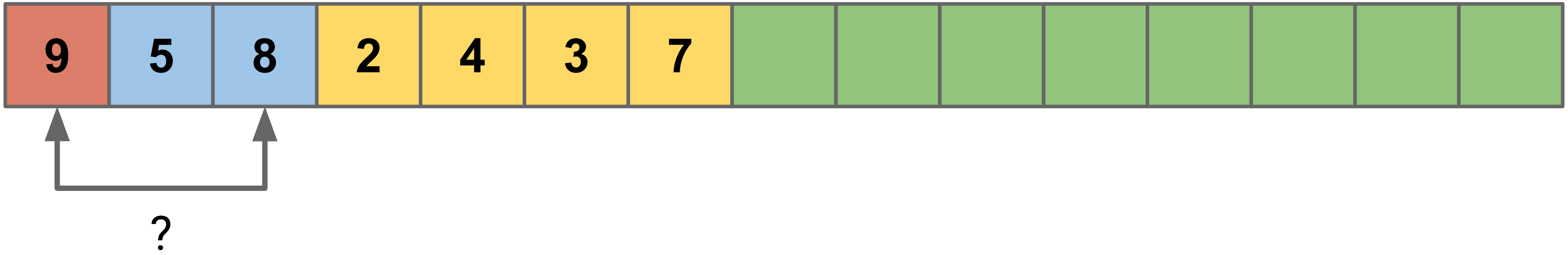
7, 4, 8, 2, 5, 3, **9**



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

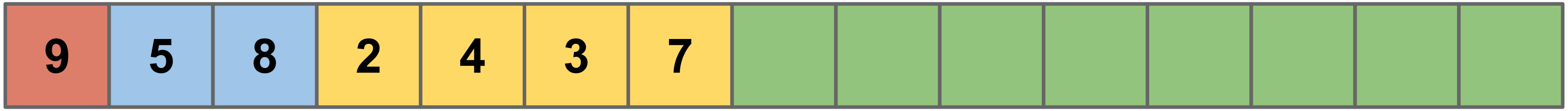
7, 4, 8, 2, 5, 3, **<u>9</u>**



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

	5	**8**	**2**	**4**	**3**	**7**								

9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

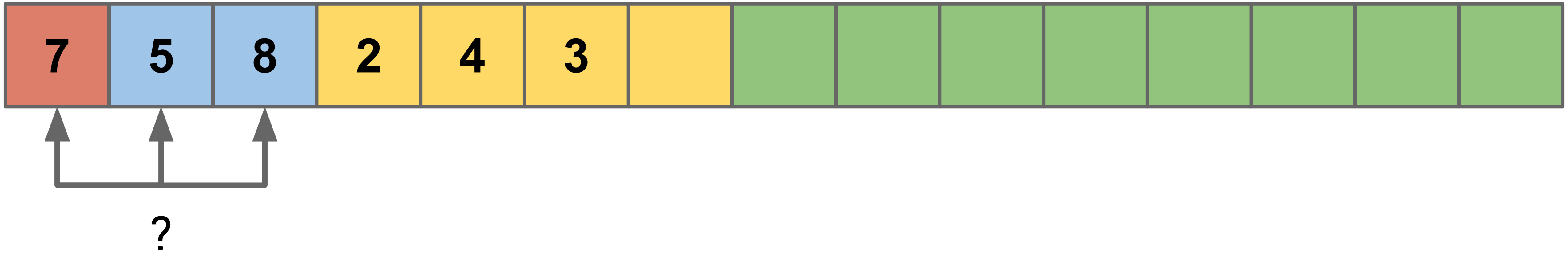
7	**5**	**8**	**2**	**4**	**3**									

9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

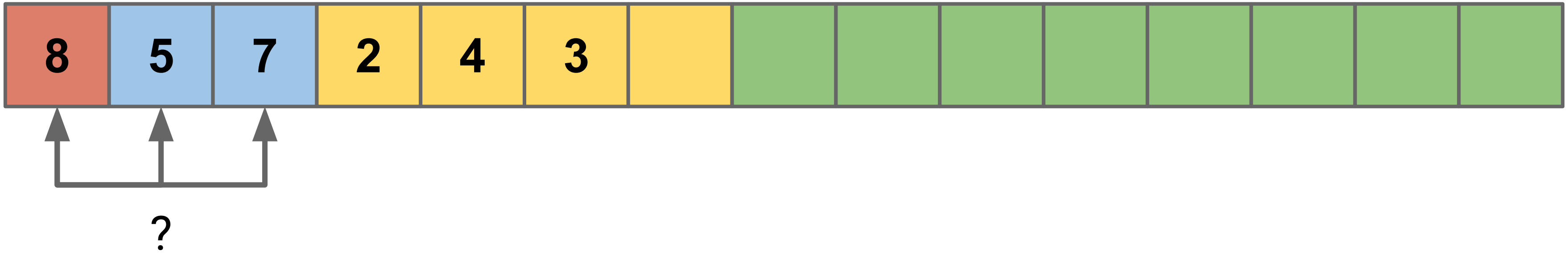


9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

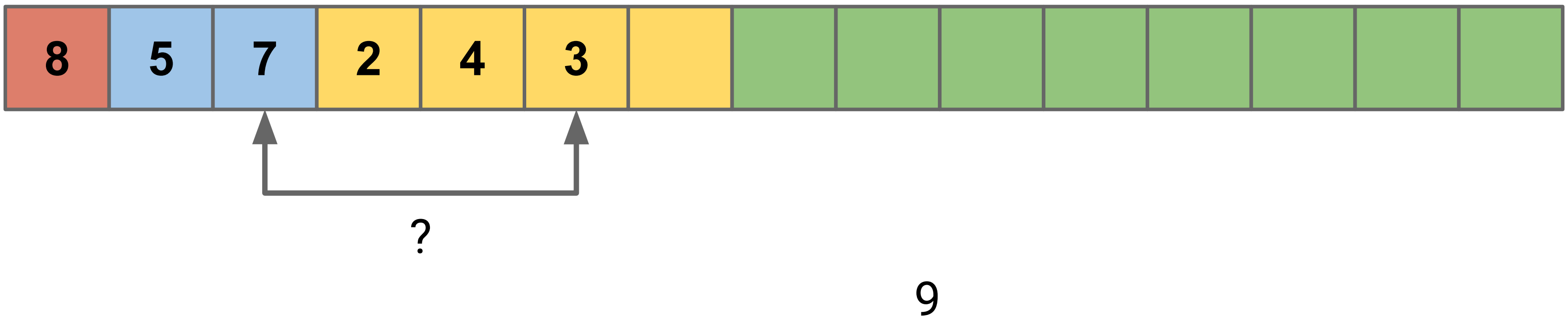


9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9



Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

	5	7	2	4	3									

8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

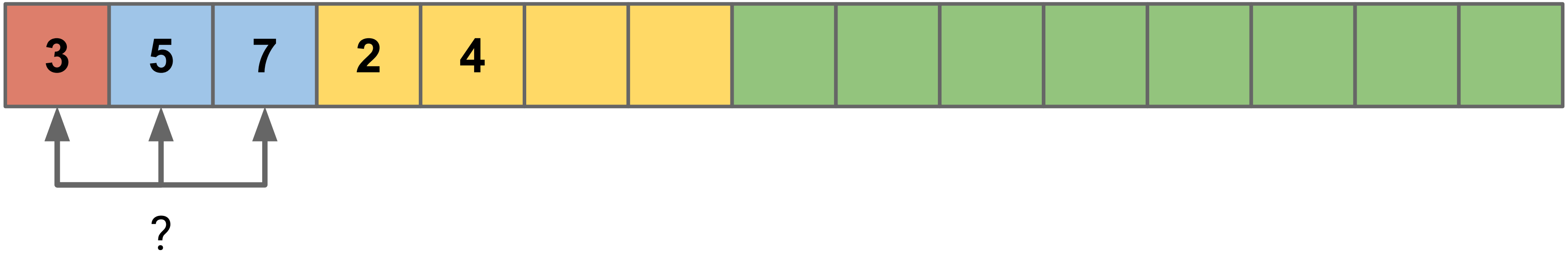
3	**5**	**7**	**2**	**4**										

8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

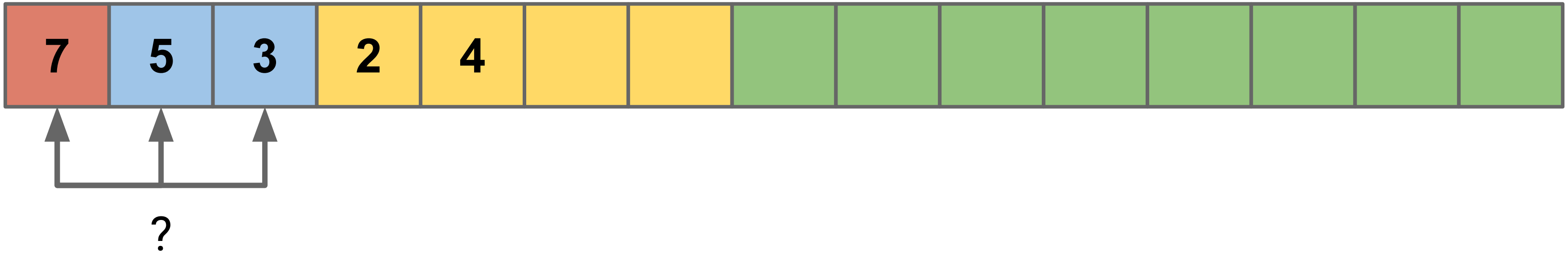


8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

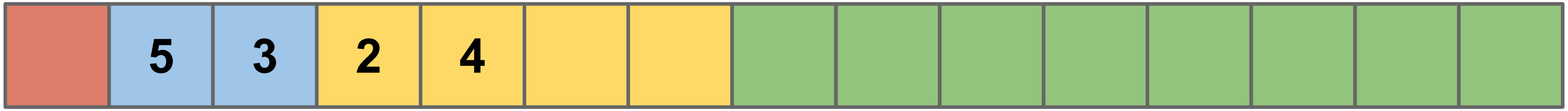


8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

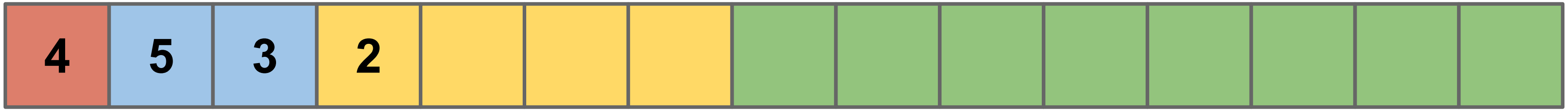


7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

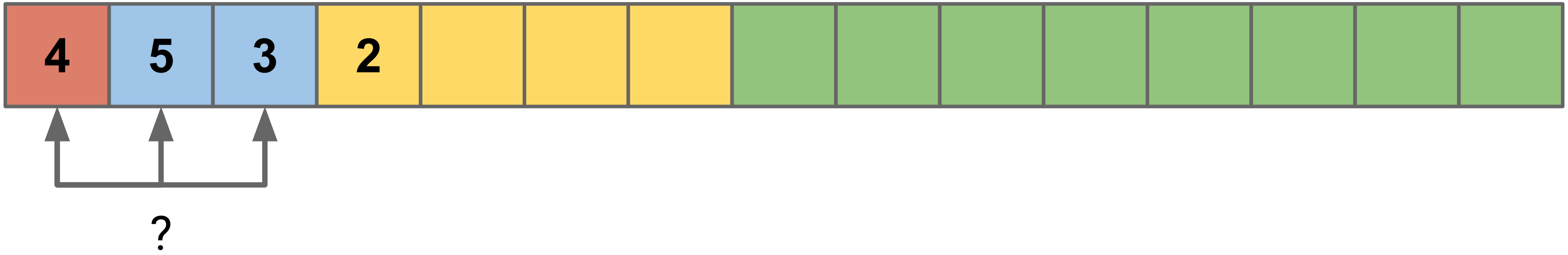


7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

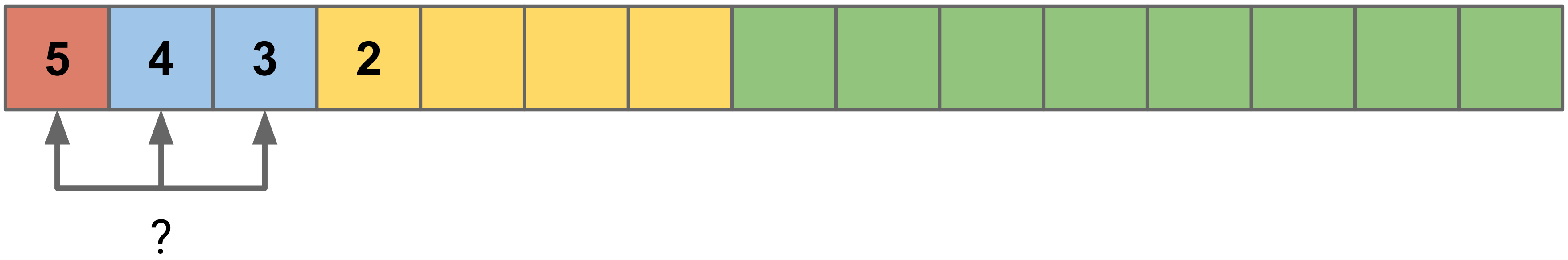


7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

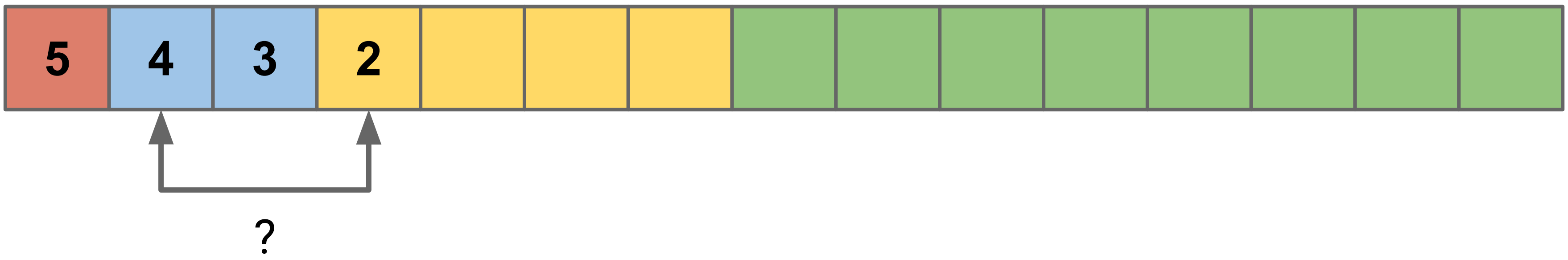


7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

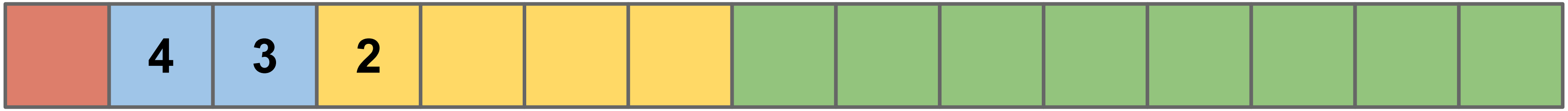


7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

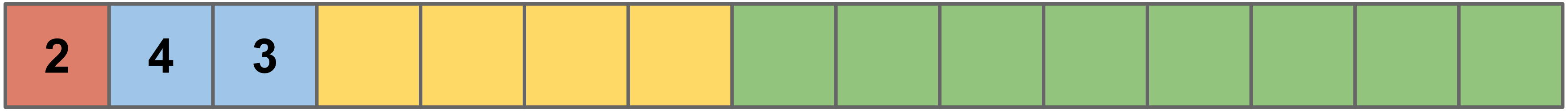


5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

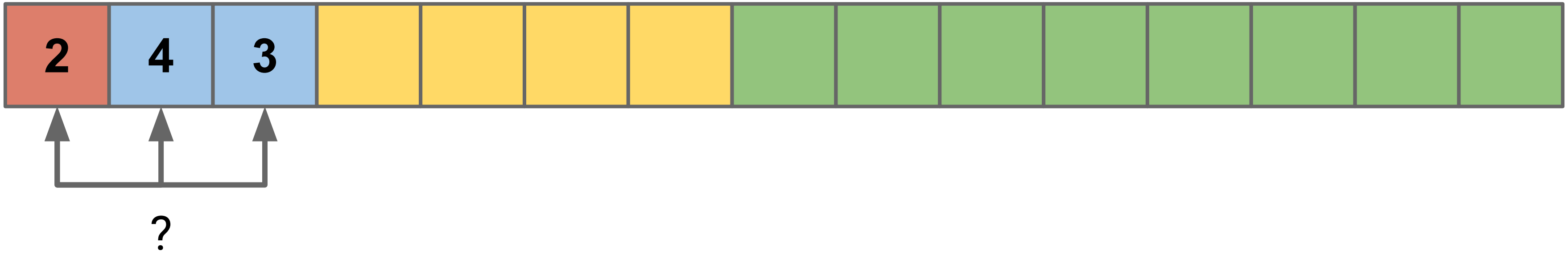


5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

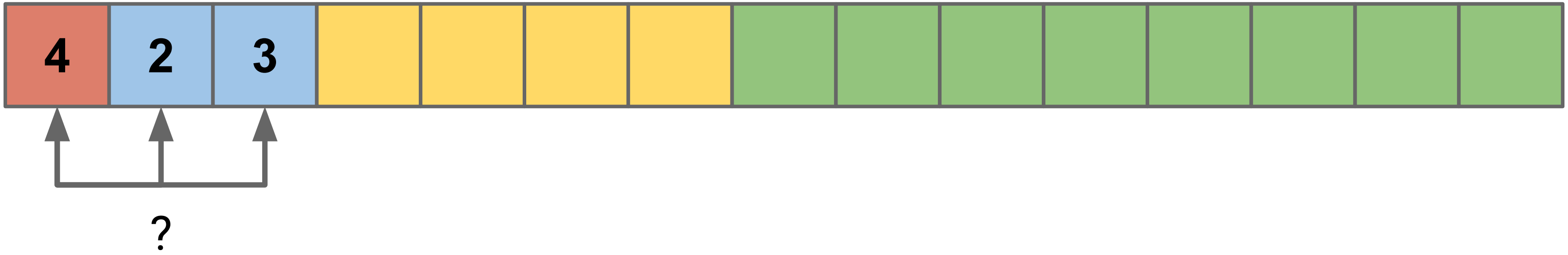


5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

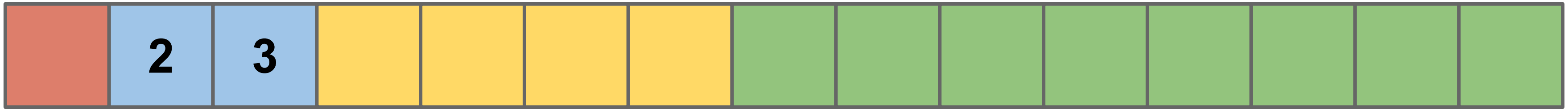


5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

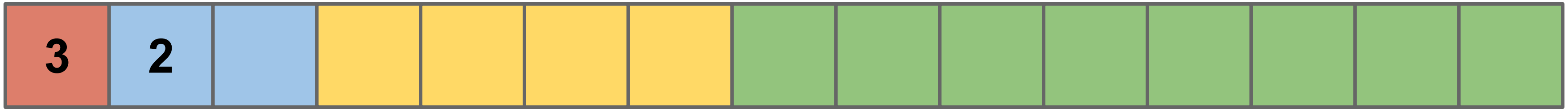


4, 5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

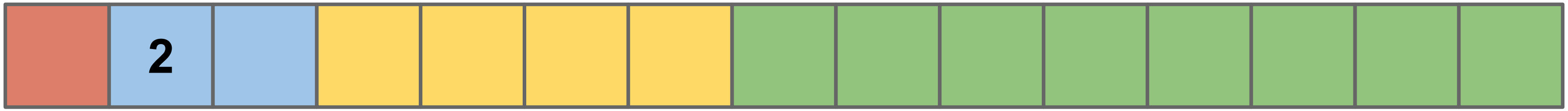


4, 5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

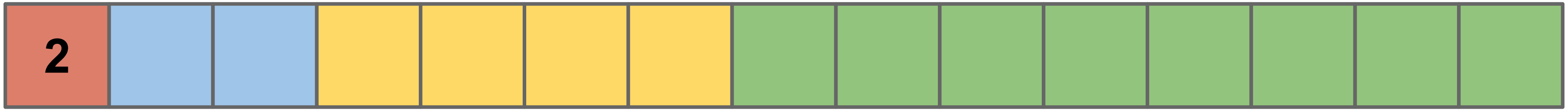


3, 4, 5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9

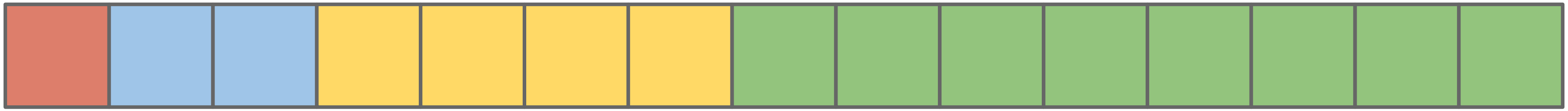


3, 4, 5, 7, 8, 9

Heap Sort

1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

7, 4, 8, 2, 5, 3, 9



2, 3, 4, 5, 7, 8, 9

Heap Sort

Heap Sort

Enqueue element *i*: $O(\log(i))$

Heap Sort

Enqueue element *i*: $O(\log(i))$

Dequeue element *i*: $O(\log(n - i))$

Heap Sort

Enqueue element *i*: $O(\log(i))$

Dequeue element *i*: $O(\log(n-i))$

$$\left(\sum_{i=1}^{n} O(\log(i))\right) + \left(\sum_{i=1}^{n} O(\log(n-i))\right)$$

Heap Sort

Enqueue element *i*: $O(\log(i))$

Dequeue element *i*: $O(\log(n-i))$

$$\left(\sum_{i=1}^{n} O(\log(i))\right) + \left(\sum_{i=1}^{n} O(\log(n-i))\right) \quad < O(n\log(n))$$

Updating Heap Elements

What if we want to update a value in our Heap?

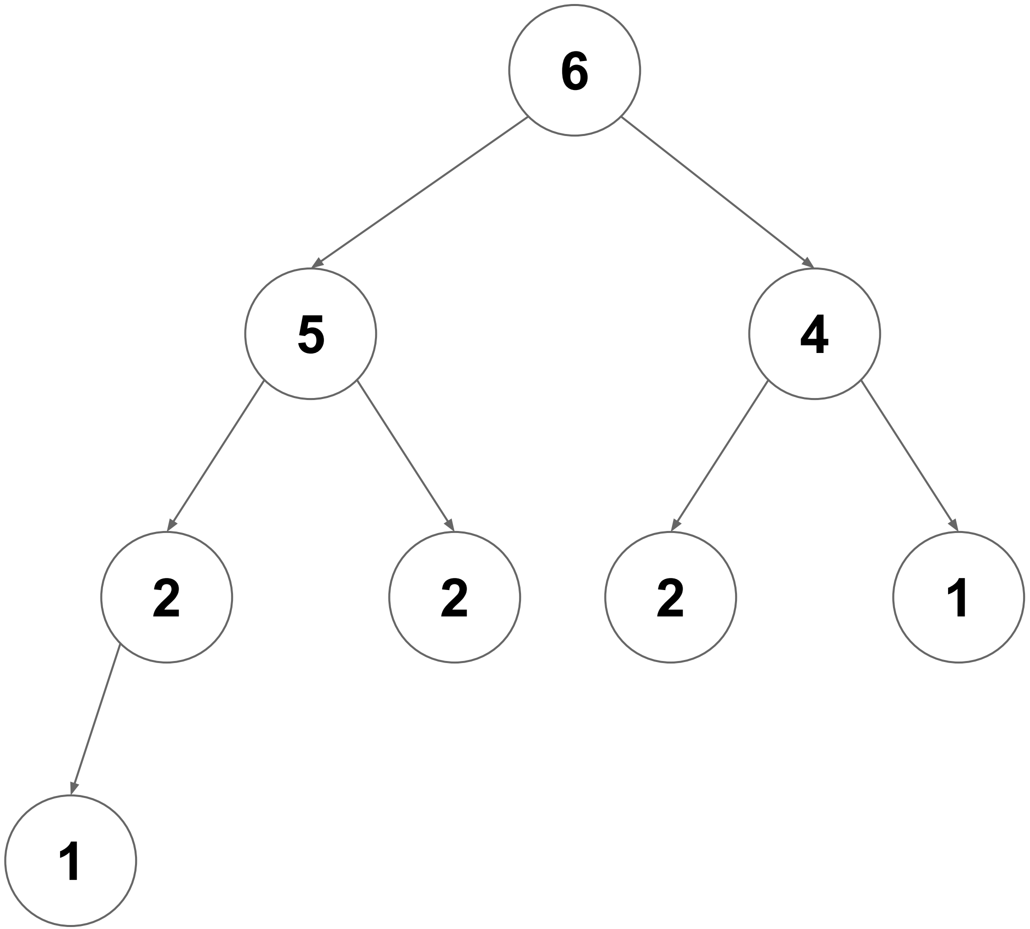
Updating Heap Elements

What if we want to update a value in our Heap?

After update we can just call **`fixUp`** or **`fixDown`** based on the new value

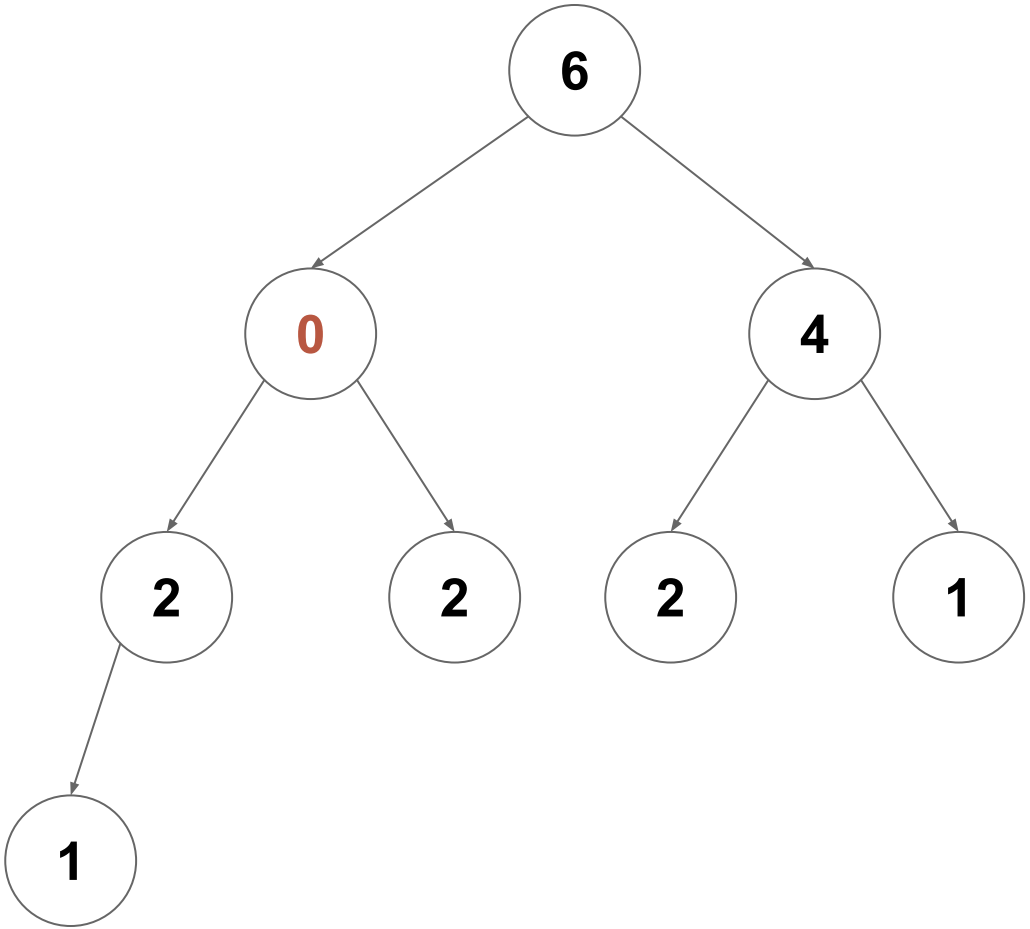
`Heap.update`

What if we change the value of the 5 node to 0?



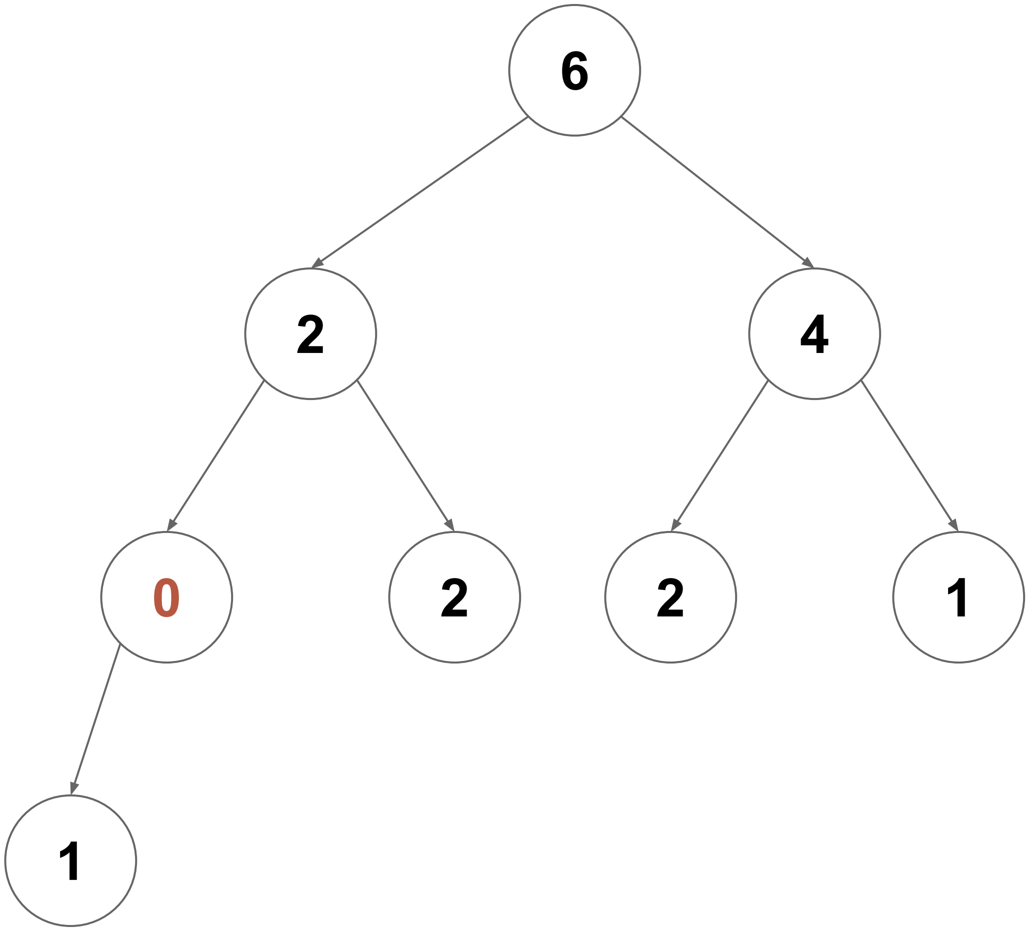
`Heap.update`

We now have to **`fixUp`** or **`fixDown`** based on the new value



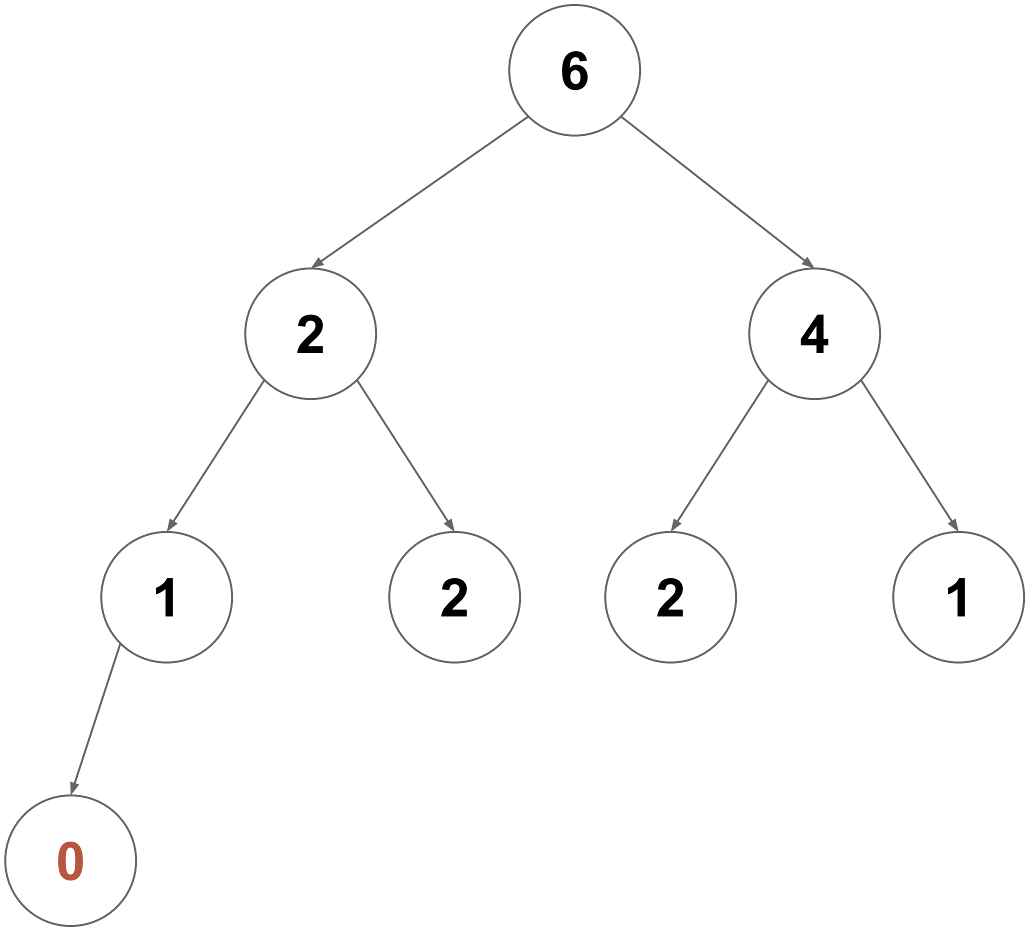
`Heap.update`

We now have to **`fixUp`** or **`fixDown`** based on the new value



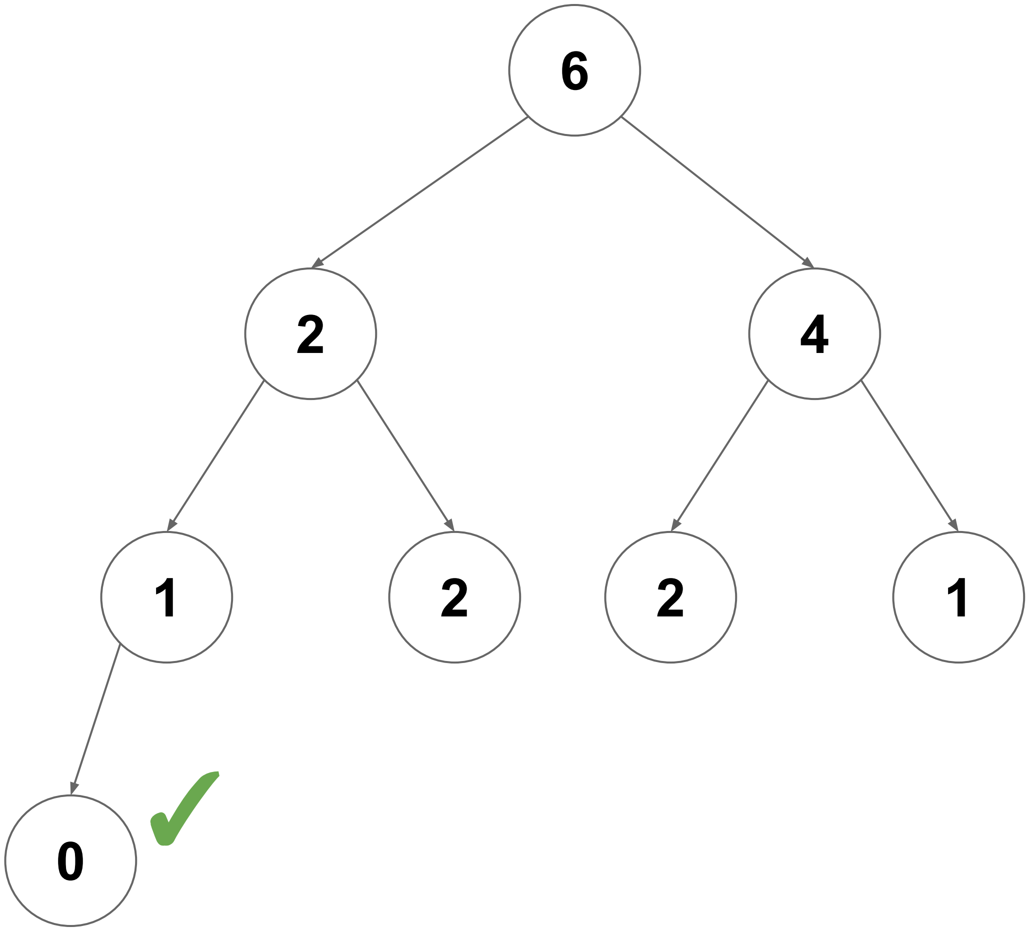
`Heap.update`

We now have to **`fixUp`** or **`fixDown`** based on the new value



`Heap.update`

We now have to **`fixUp`** or **`fixDown`** based on the new value



Updating Heap Elements

What if we want to update a value in our Heap?

After update we can just call **`fixUp`** or **`fixDown`** based on the new value

Updating Heap Elements

What if we want to update a value in our Heap?

After update we can just call **`fixUp`** or **`fixDown`** based on the new value

Can we apply this idea to an entire array?

Heapify

Input: Array

Output: Array re-ordered to be a heap

Heapify

Input: Array

Output: Array re-ordered to be a heap

Idea: **`fixUp`** or **`fixDown`** all ***n*** elements in the array

Heapify

Input: Array

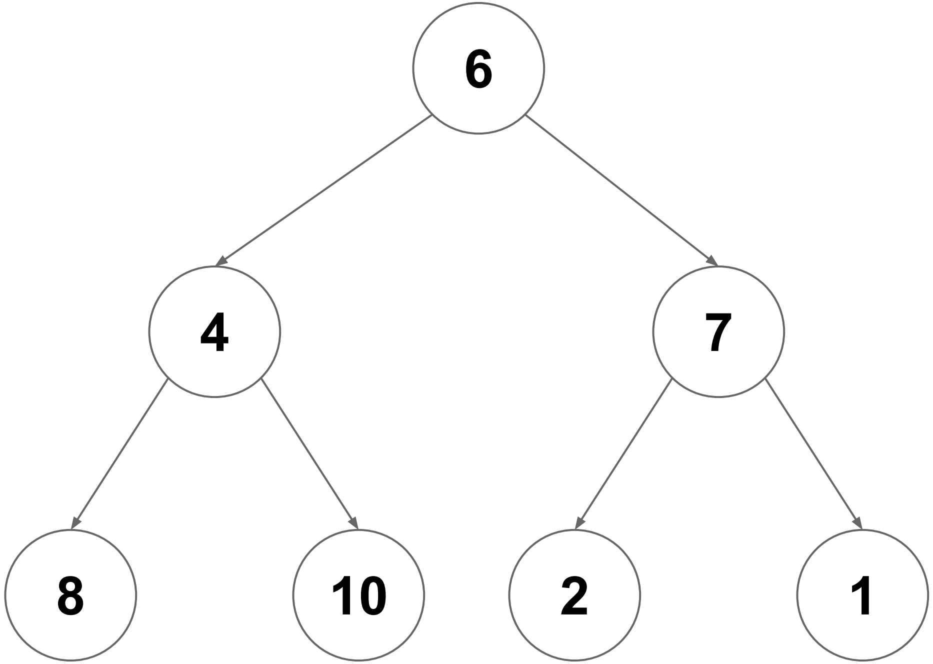
Output: Array re-ordered to be a heap

Idea: **`fixUp`** or **`fixDown`** all ***n*** elements in the array

Given the cost of **`fixUp`** *and* **`fixDown`** *what do we expect the complexity* **`Heapify`** *will be?*

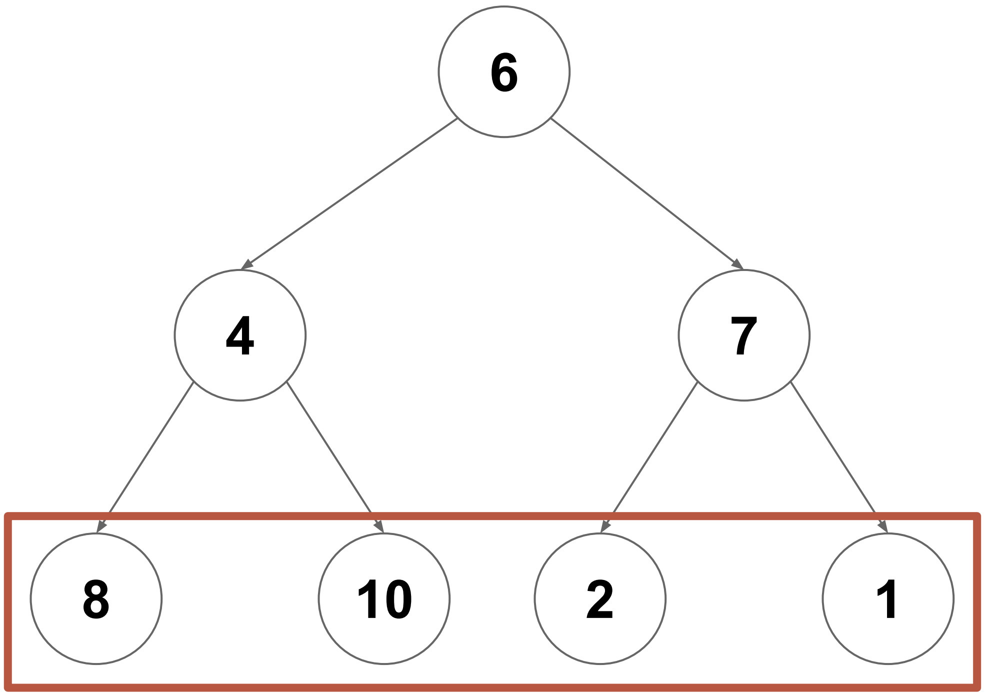
Heapify

Given an arbitrary array (show as a tree here) turn it into a heap



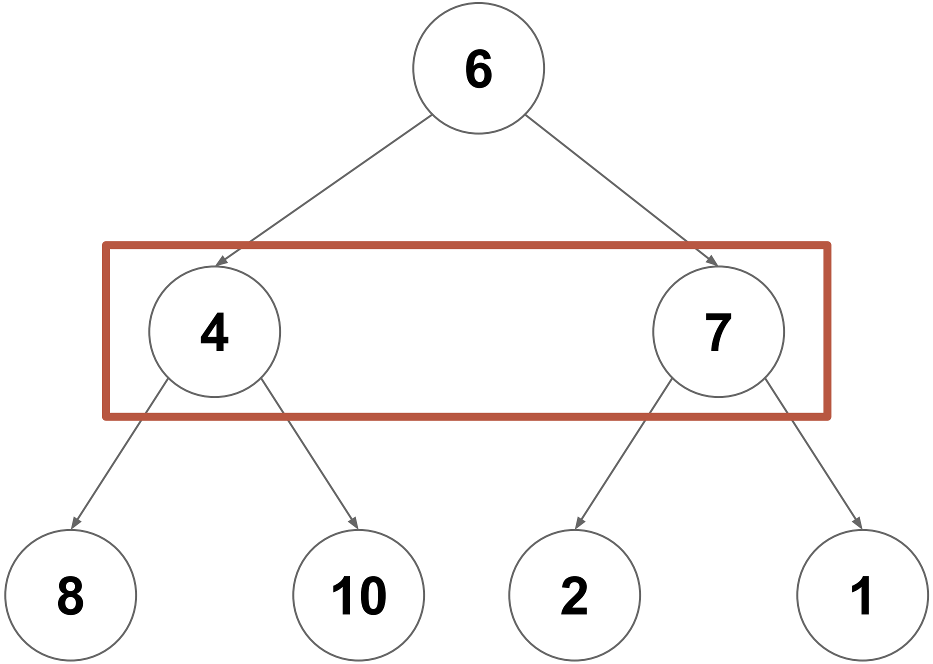
Heapify

Start at the lowest level, and call **`fixDown`** on each node (0 swaps per node)



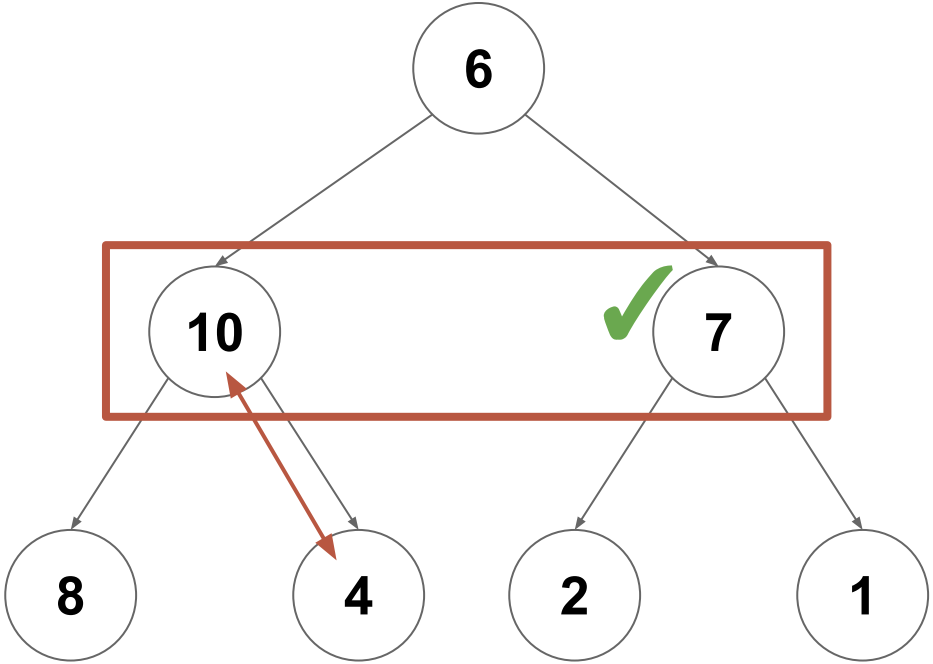
Heapify

Do the same at the next lowest level (at most one swap per node)



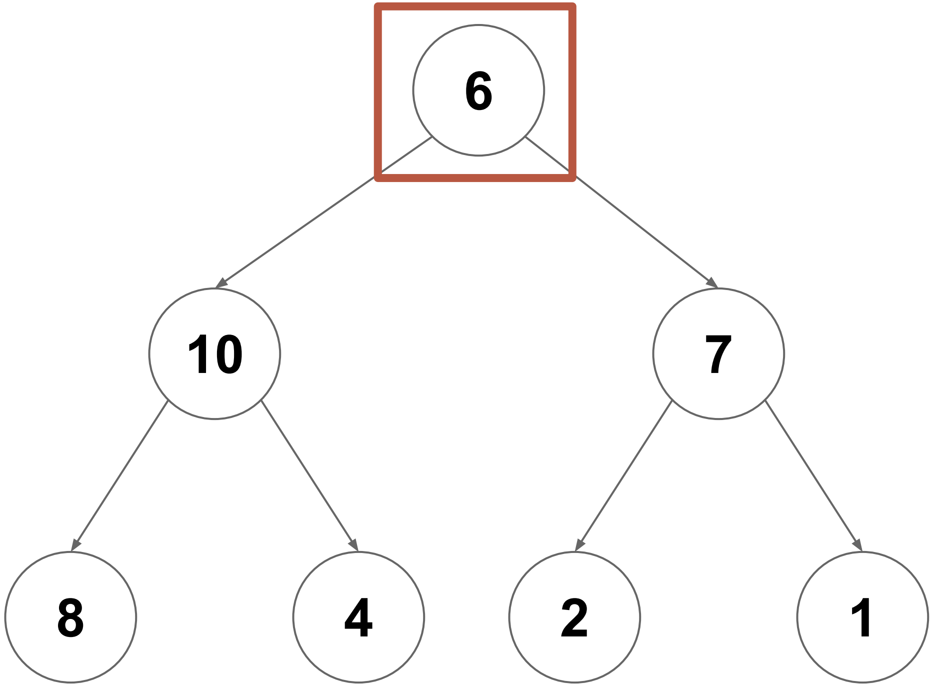
Heapify

Do the same at the next lowest level (at most one swap per node)



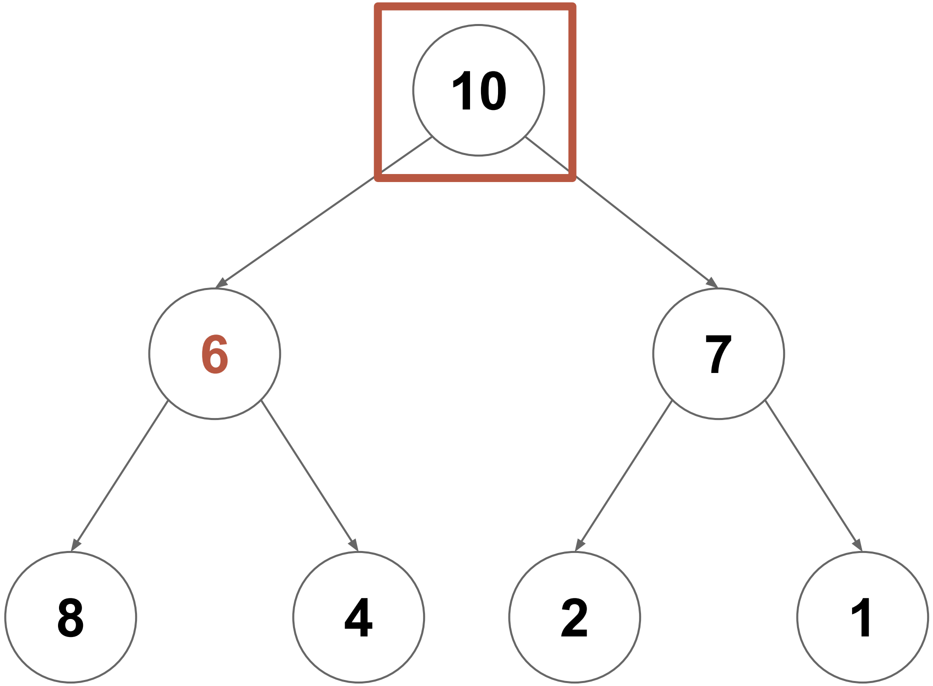
Heapify

Continue upwards (now at most 2 swaps per node)



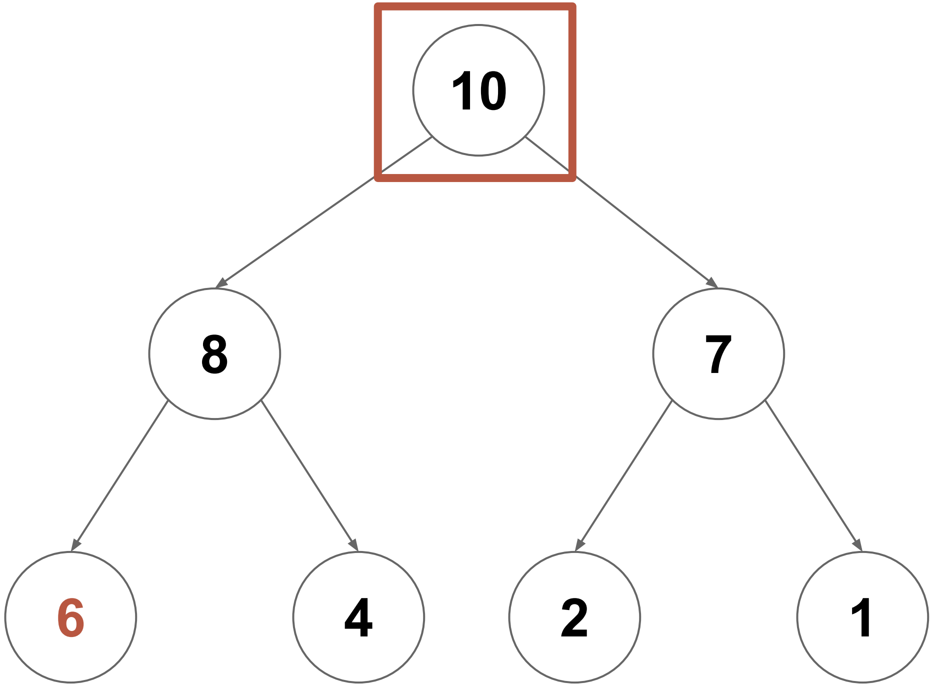
Heapify

Continue upwards (now at most 2 swaps per node)



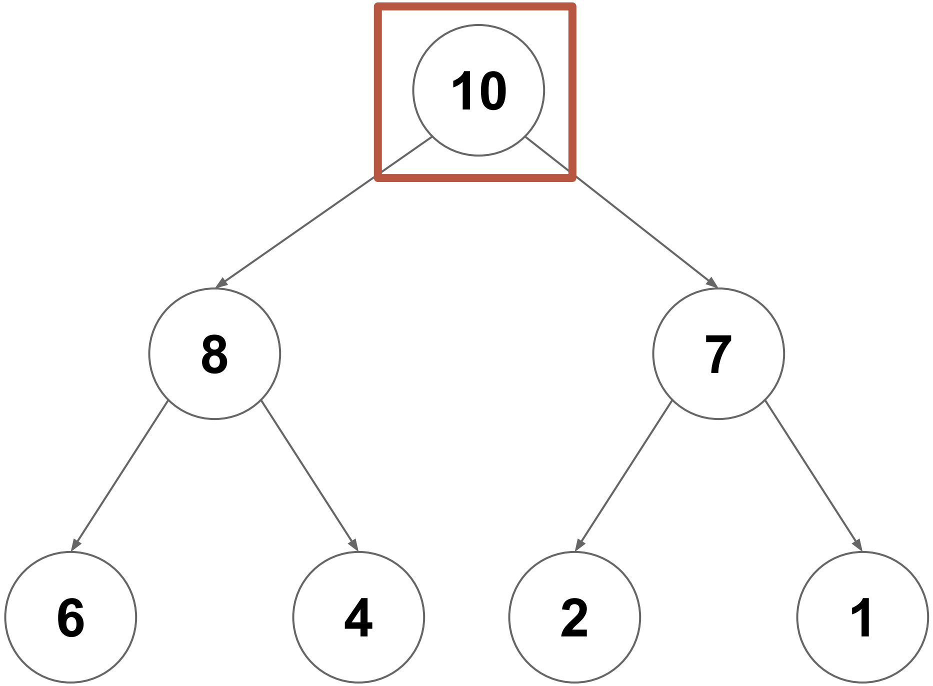
Heapify

Continue upwards (now at most 2 swaps per node)



Heapify

Continue upwards (now at most 2 swaps per node)



Heapify

Heapify

At level log(*n*): Call `fixDown` on all *n*/2 nodes at this level (max 0 swaps each)

Heapify

At level log(*n*): Call `fixDown` on all *n*/2 nodes at this level (max 0 swaps each)

At level log(*n*)-1: Call `fixDown` on all *n*/4 nodes at this level (max 1 swaps each)

Heapify

At level log(*n*): Call `fixDown` on all *n*/2 nodes at this level (max 0 swaps each)

At level log(*n*)-1: Call `fixDown` on all *n*/4 nodes at this level (max 1 swaps each)

At level log(*n*)-2: Call `fixDown` on all *n*/8 nodes at this level (max 2 swaps each)

Heapify

At level log(*n*): Call `fixDown` on all *n*/2 nodes at this level (max 0 swaps each)

At level log(*n*)-1: Call `fixDown` on all *n*/4 nodes at this level (max 1 swaps each)

At level log(*n*)-2: Call `fixDown` on all *n*/8 nodes at this level (max 2 swaps each)

…

At level 1: Call `fixDown` on all 1 nodes at this level (max log(*n*) swaps each)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

Heapify

Sum the number of swaps required by each level

Heapify

Pull out the *n* as a constant and distribute multiplication

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

Heapify

Focus on the dominant term only

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

Heapify

Change log(n) to infinity (can only increase complexity class if anything)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i}\right)$$

Heapify

We can now treat the sum as a constant

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \boxed{\sum_{i=1}^{\infty} \frac{i}{2^i}}\right)$$

This is known to converge to a constant

Heapify

Therefore we can heapify an array of size *n* in *O*(*n*)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i}\right) = O\left(n\right)$$

Heapify

Therefore we can heapify an array of size *n* in *O*(*n*)

(but heap sort still requires *n* log(*n*) due to dequeue costs)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i}\right) = O(n)$$