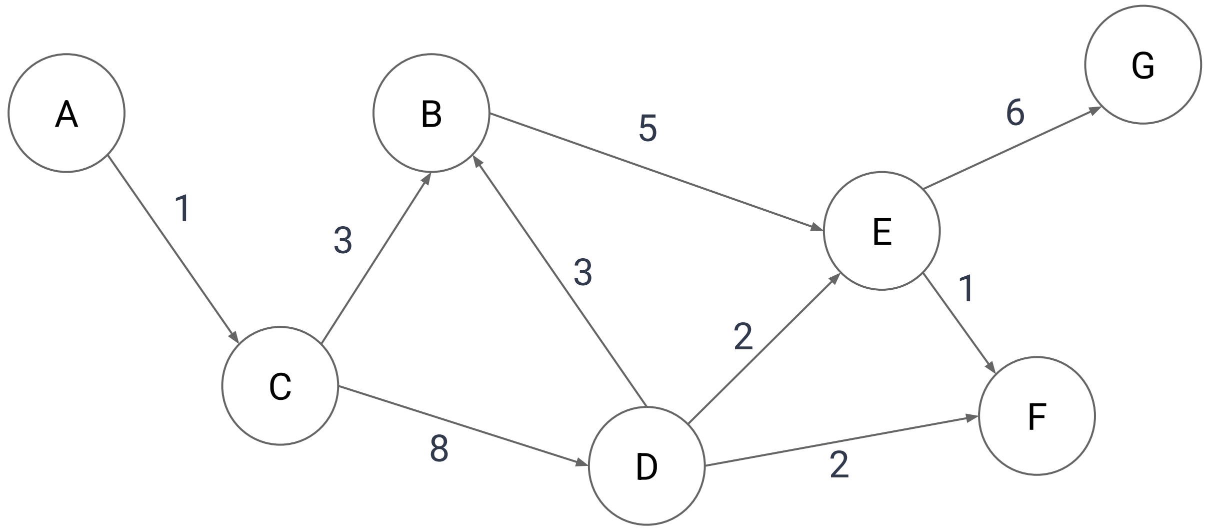
# CSE 250

## Data Structures

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

## Lec 24: Heaps

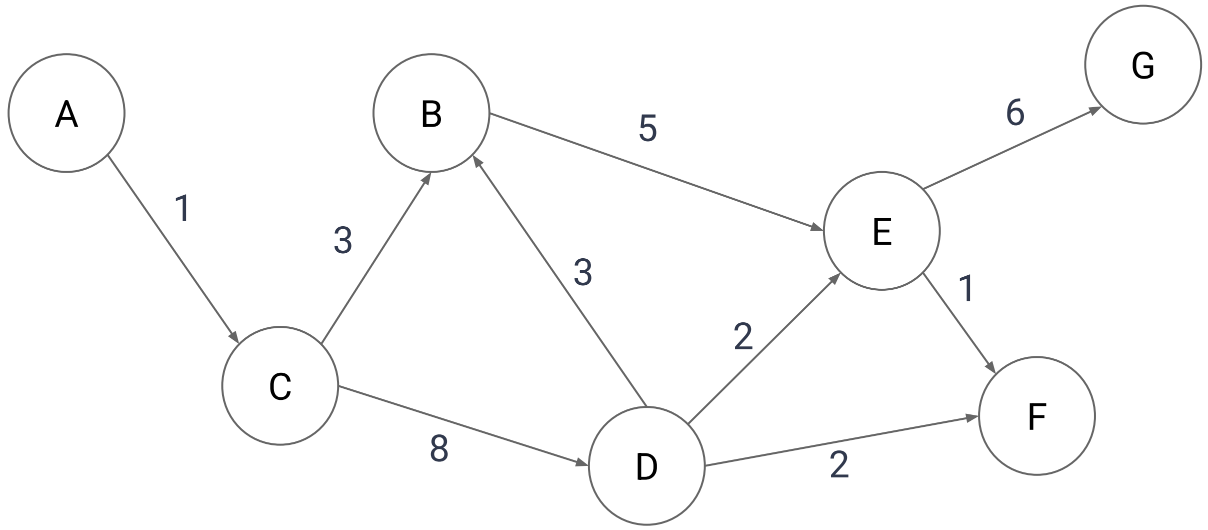
# Warm-Up Question



Which algorithm(s) **could** find the path A -> C -> B -> E -> F?

**A: BFS** **B: DFS** **C: Djikstra's**

# Warm-Up Question



Which algorithm(s) **could** find the path A -> C -> B -> E -> F?

~~**A: BFS**~~ [✓ **B: DFS**] [✓ **C: Djikstra's**]

# Announcements

- PA2 autolab is up – Check the FAQ @271 for more sweet tips

# `PriorityQueue` ADT

**`PriorityQueue<T>`**

**`void add(T value)`**

Insert **`value`** into the priority queue

**`T poll()`**

Remove the highest priority value in the priority queue

**`T peek()`**

Peek at the highest priority value 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 |
|---|---|---|
| add | $O(1)$ | $O(n)$ |
| poll | $O(n)$ | $O(1)$ |
| peek | $O(n)$ | $O(1)$ |

## Priority Queues

| Operation | Lazy | Proactive |
|---|---|---|
| add | $O(1)$ | $O(n)$ |
| poll | $O(n)$ | $O(1)$ |
| peek | $O(n)$ | $O(1)$ |

*Can we do better?*

## Priority Queues

**Lazy -** Fast add, Slow removal

**Proactive -** Slow add, Fast removal

# Priority Queues

**Lazy -** Fast add, Slow removal

**Proactive -** Slow add, Fast removal

**??? -** Fast(-ish) add, Fast(-ish) removal

## 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 higher priority 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 ***a* ≤ *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

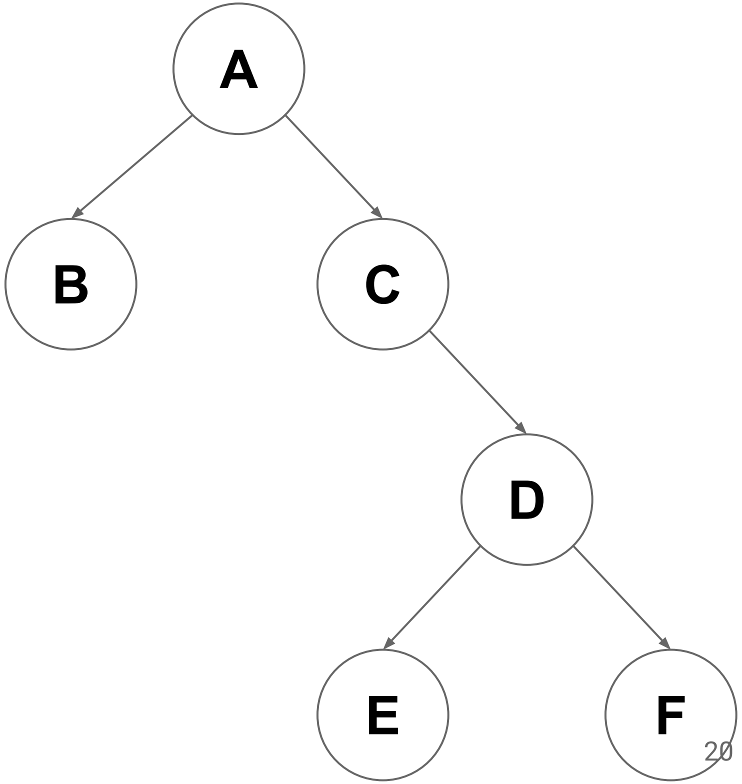
**A** is the root

**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 Min Heaps

Organize our priority queue as a directed tree

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

# Binary Min 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 Min 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 Min 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 Min 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)

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

# Binary Min Heaps

Organize our priority queue as a directed tree

**Directed:** A directed edge from ***a*** to ***b*** means that $\boxed{\boldsymbol{a \le b}}$

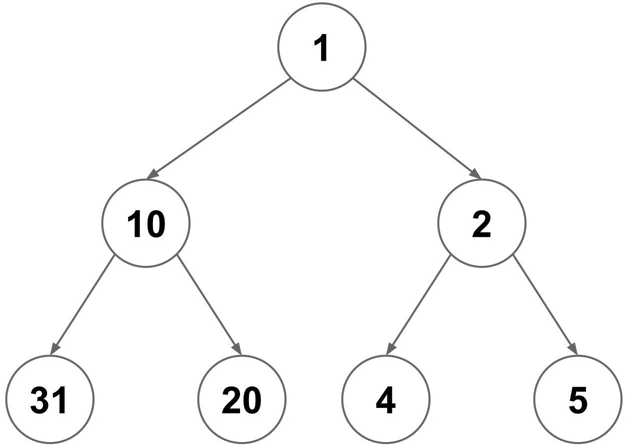
A max heap would reverse this ordering

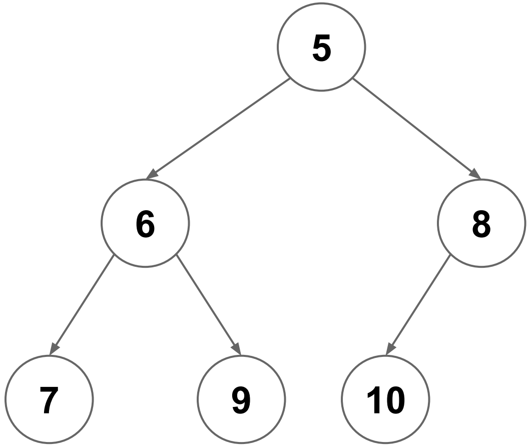
**Binary:** Max out-degree of 2 (easy to reason about)

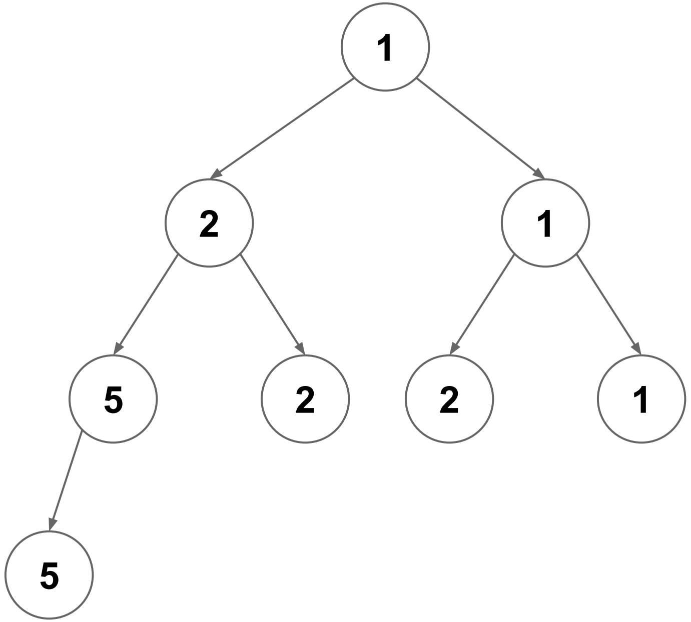
**Complete:** Every "level" except the last is full (from left to right)

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

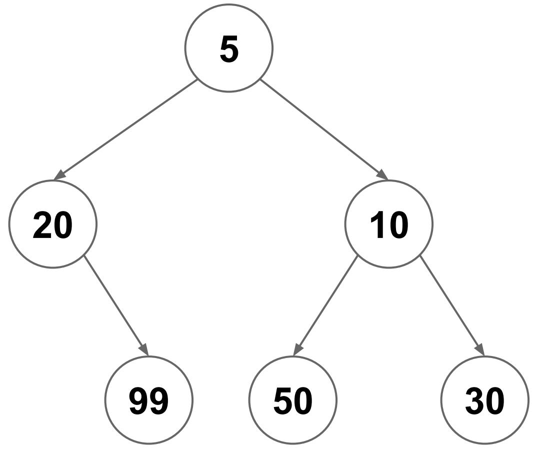
# Valid Min Heaps

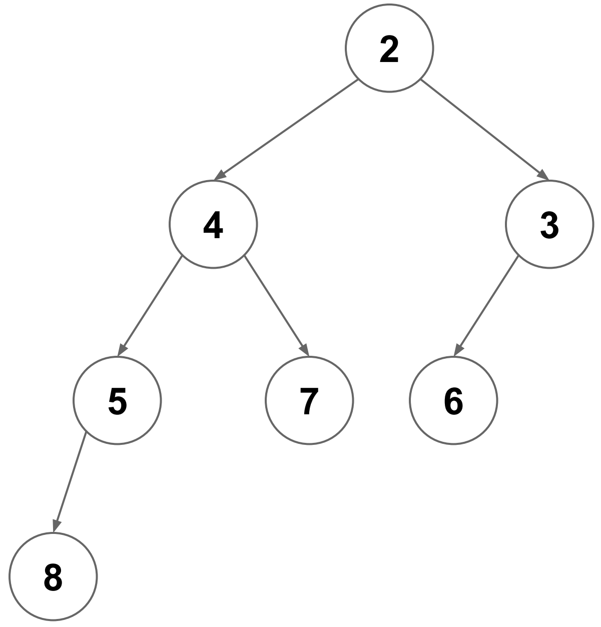


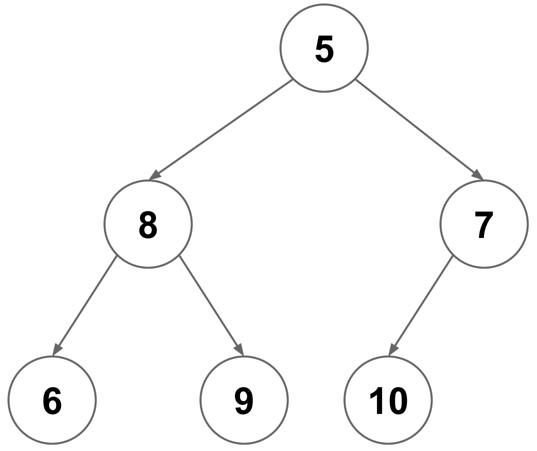




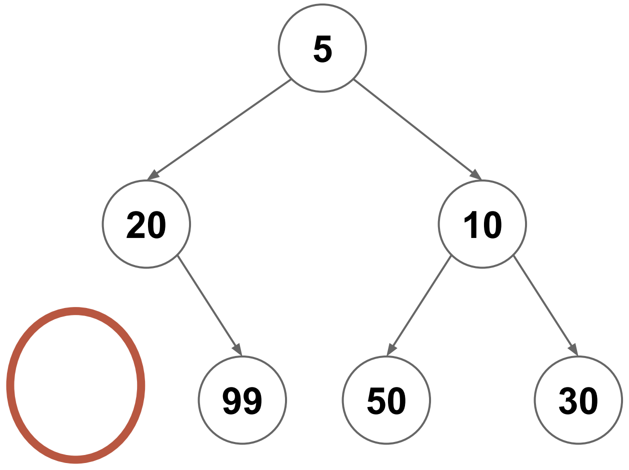
# Invalid Min Heaps



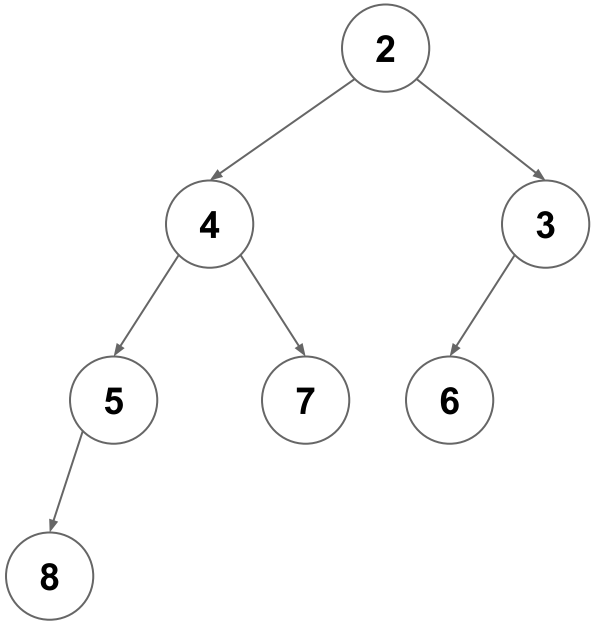


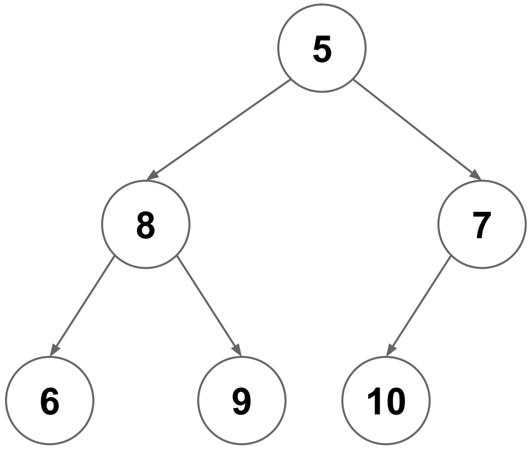


## Invalid Min Heaps

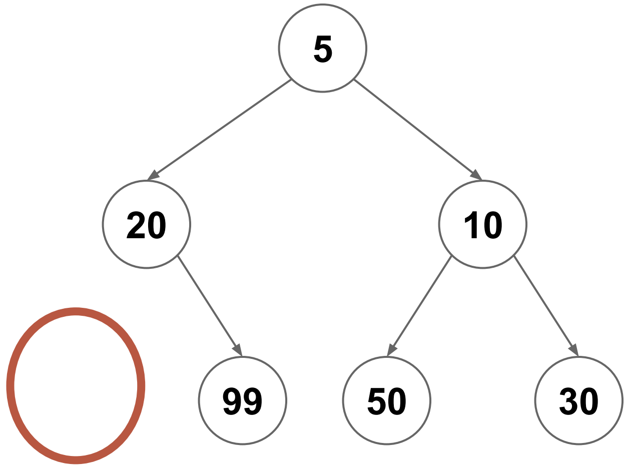


Need to fill from left to right

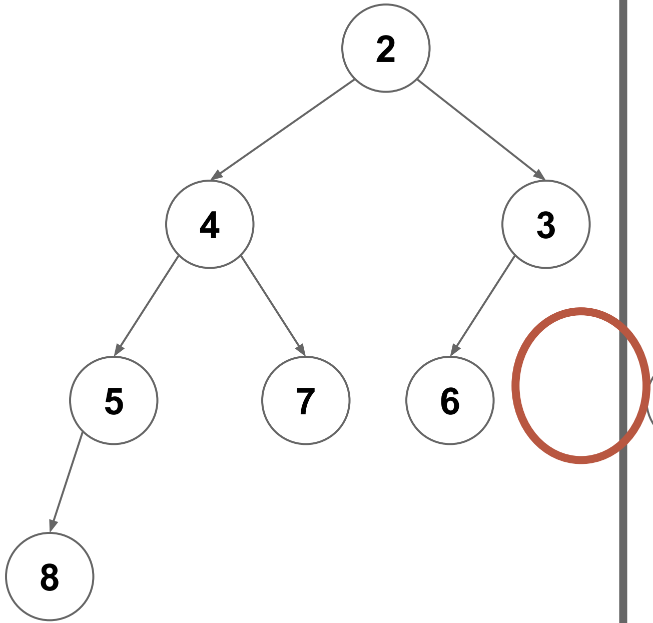




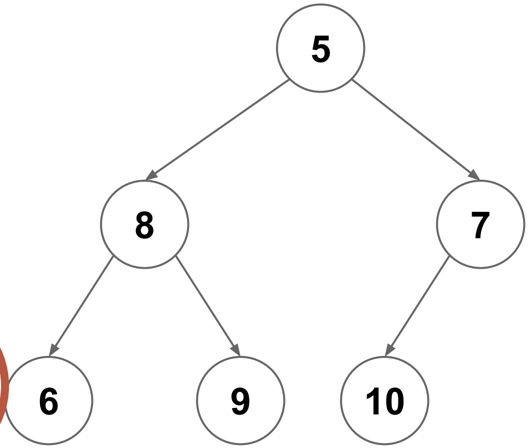
# Invalid Min Heaps



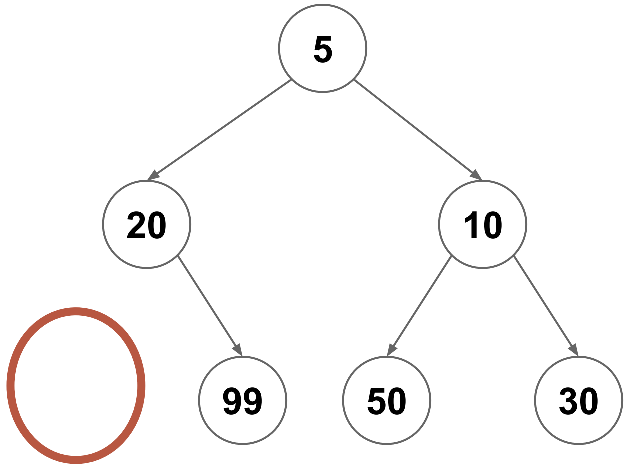
Need to fill from left to right



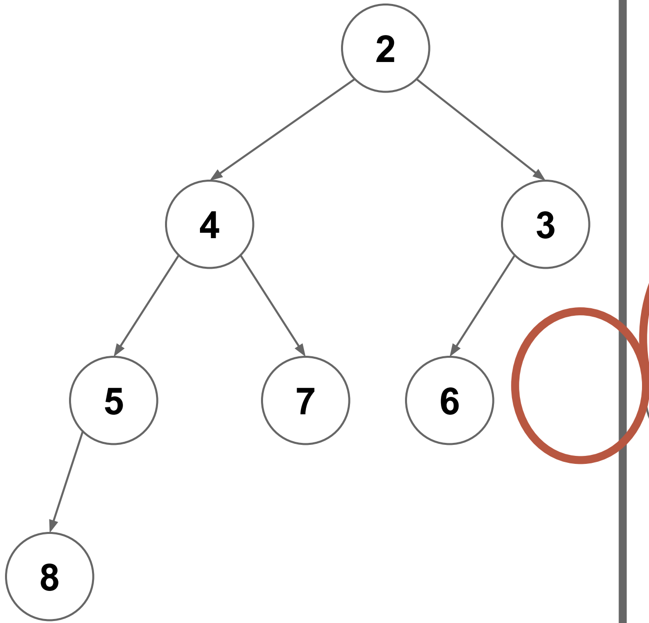
Need complete levels



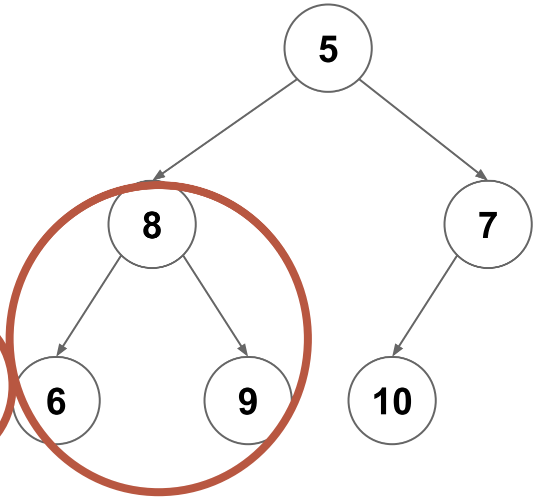
# Invalid Min Heaps



Need to fill from left to right



Need complete levels



Parents must be less than or equal to children

# 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 `MinHeap` ADT

**`void pushHeap(T value)`**

Place an item into the heap

**`T popHeap()`**

Remove and return the minimal element from the heap

**`T peek()`**

Peek at the minimal element in the heap

**`int size()`**

The number of elements in the heap

## pushHeap

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

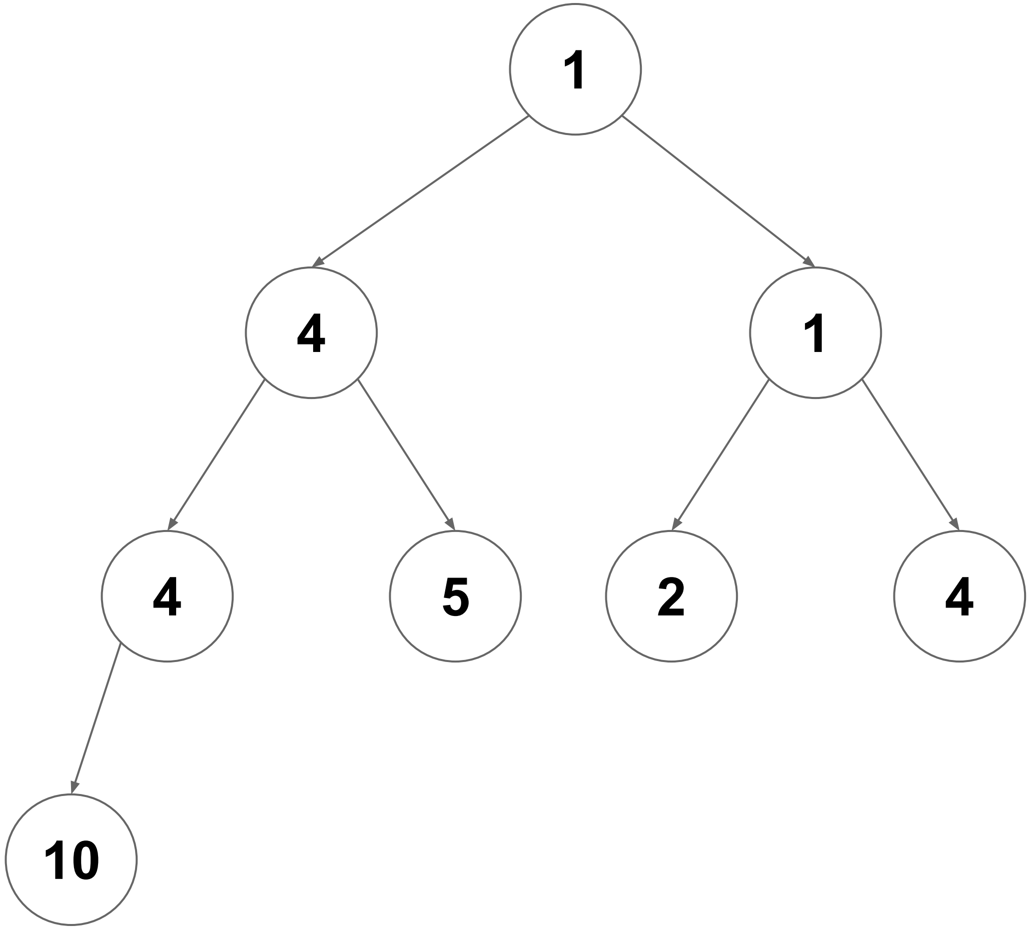
## `pushHeap`

**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. Set **`current = parent`**

## pushHeap

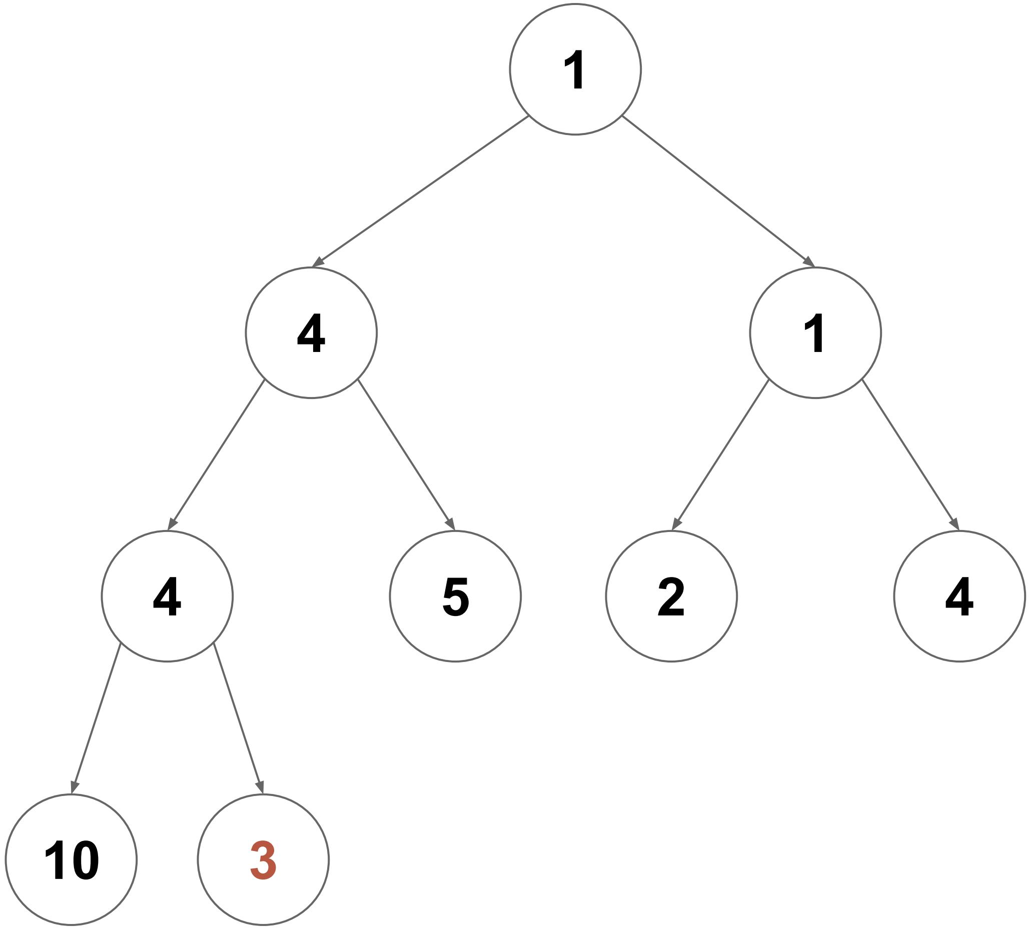
What if we add 3?



## pushHeap

What if we add 3?

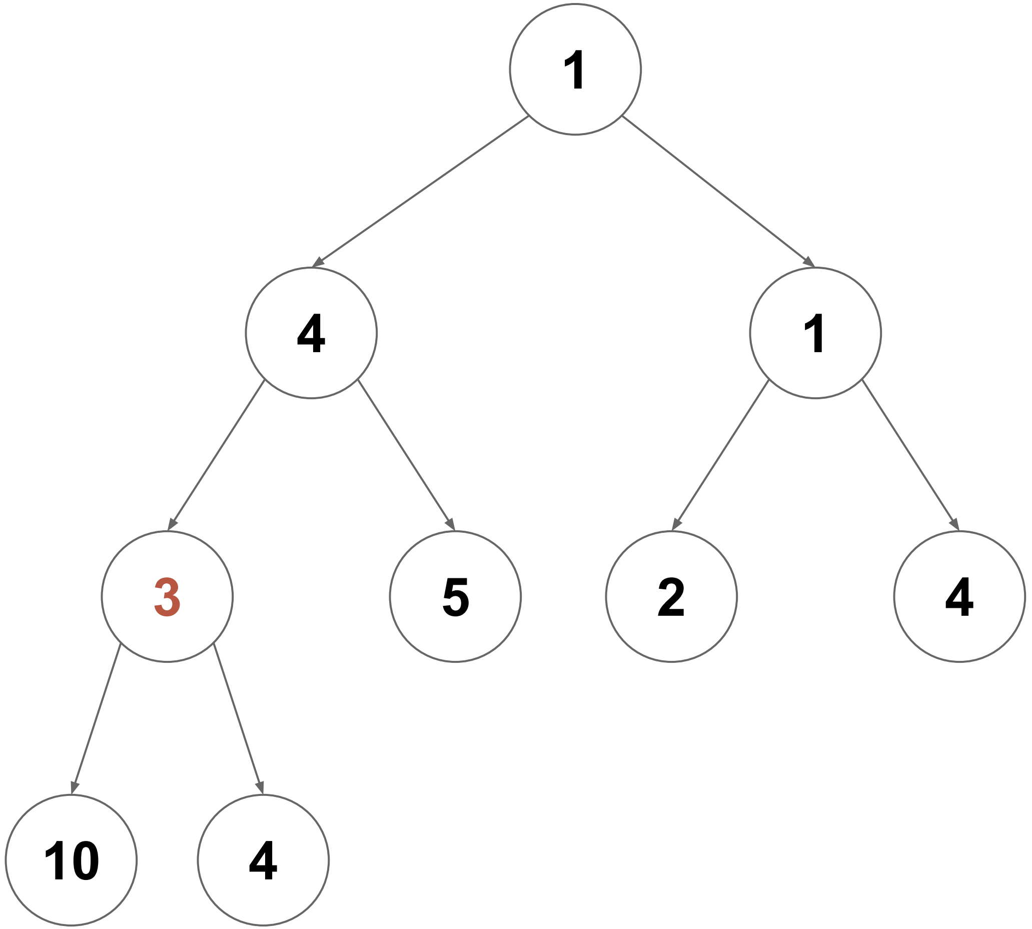
Place in the next available spot



## pushHeap

What if we enqueue 3?

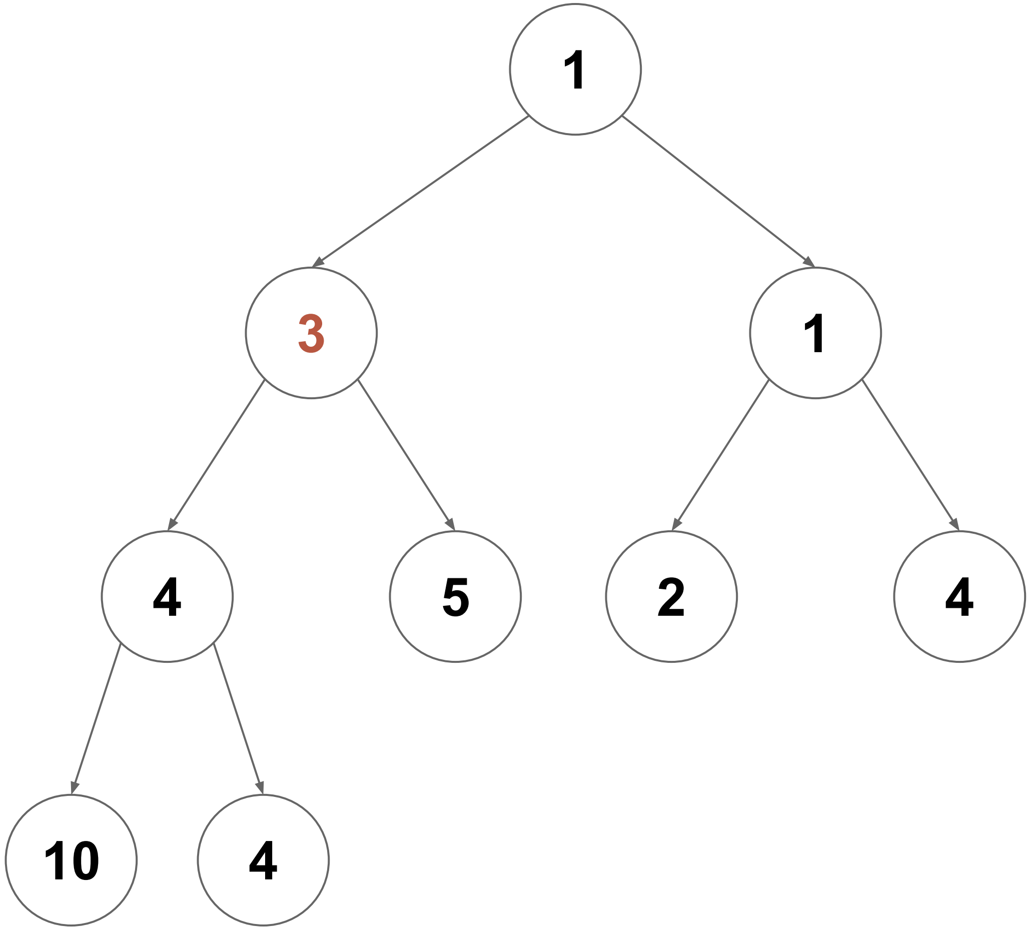
Swap with parent if it is smaller than the parent



## pushHeap

What if we enqueue 3?

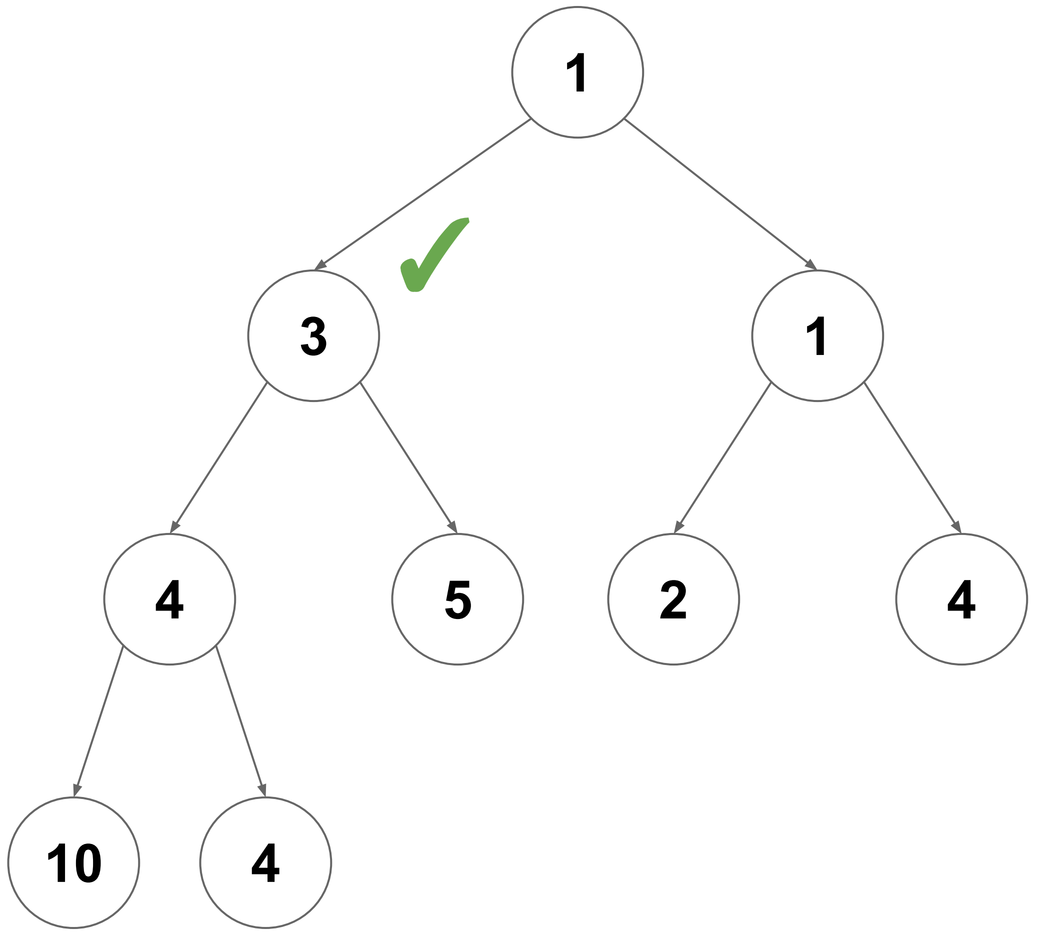
Continue swapping upwards…



## pushHeap

What if we enqueue 3?

Stop swapping when we are no longer smaller than our parent



## `pushHeap`

**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. Set **`current = parent`**

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

## `pushHeap`

**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. Set **`current = parent`**

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

## popHeap

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

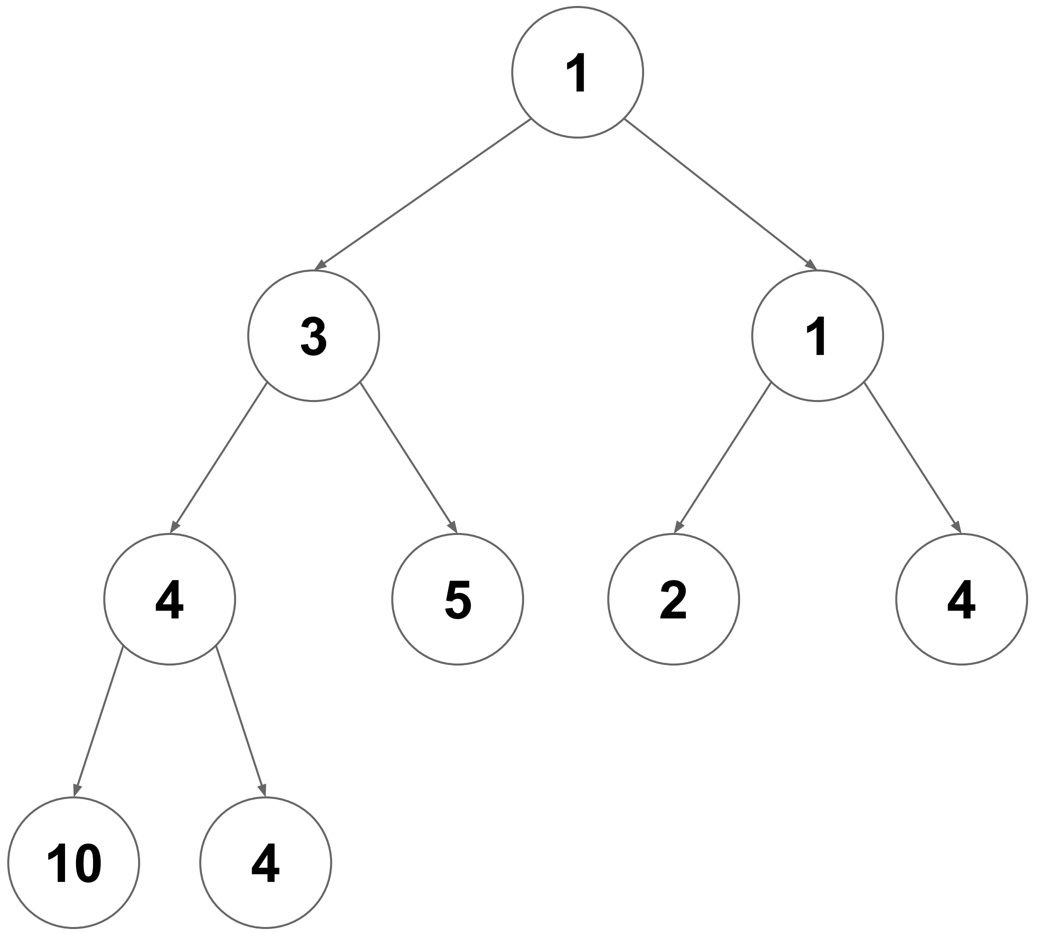
## popHeap

**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 smallest **`child`**
    b. Set **`current = child`**

## popHeap

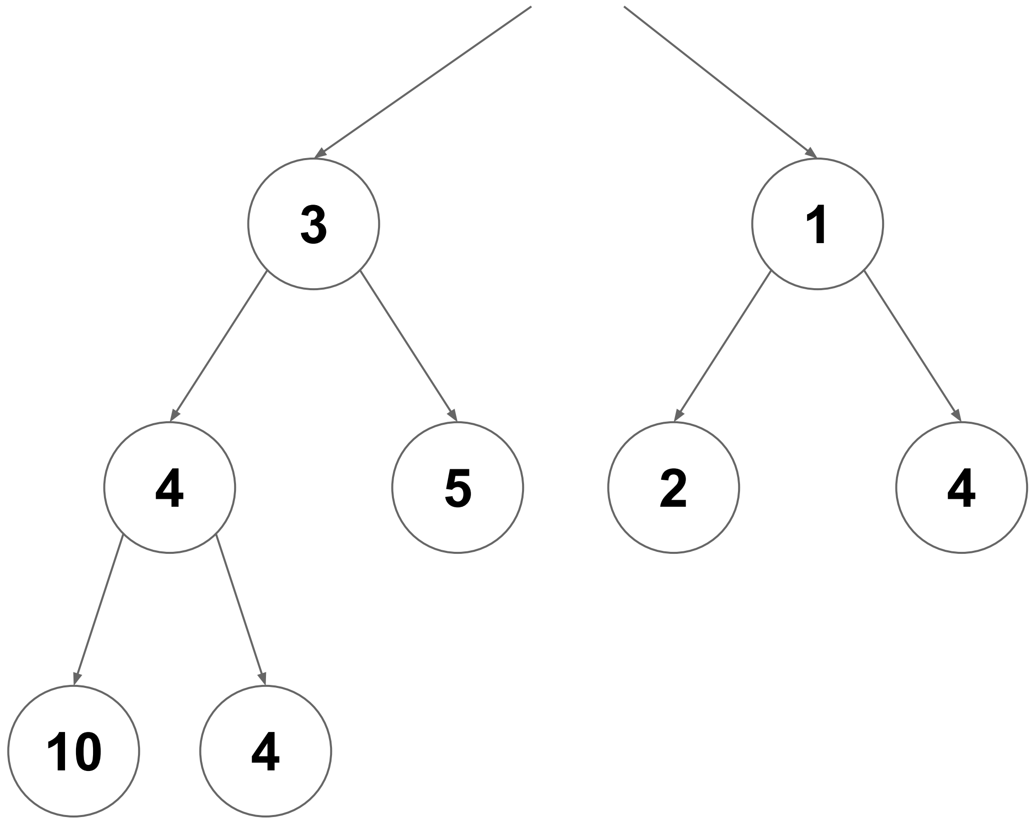
What if we call popHeap?



## popHeap

What if we call popHeap?

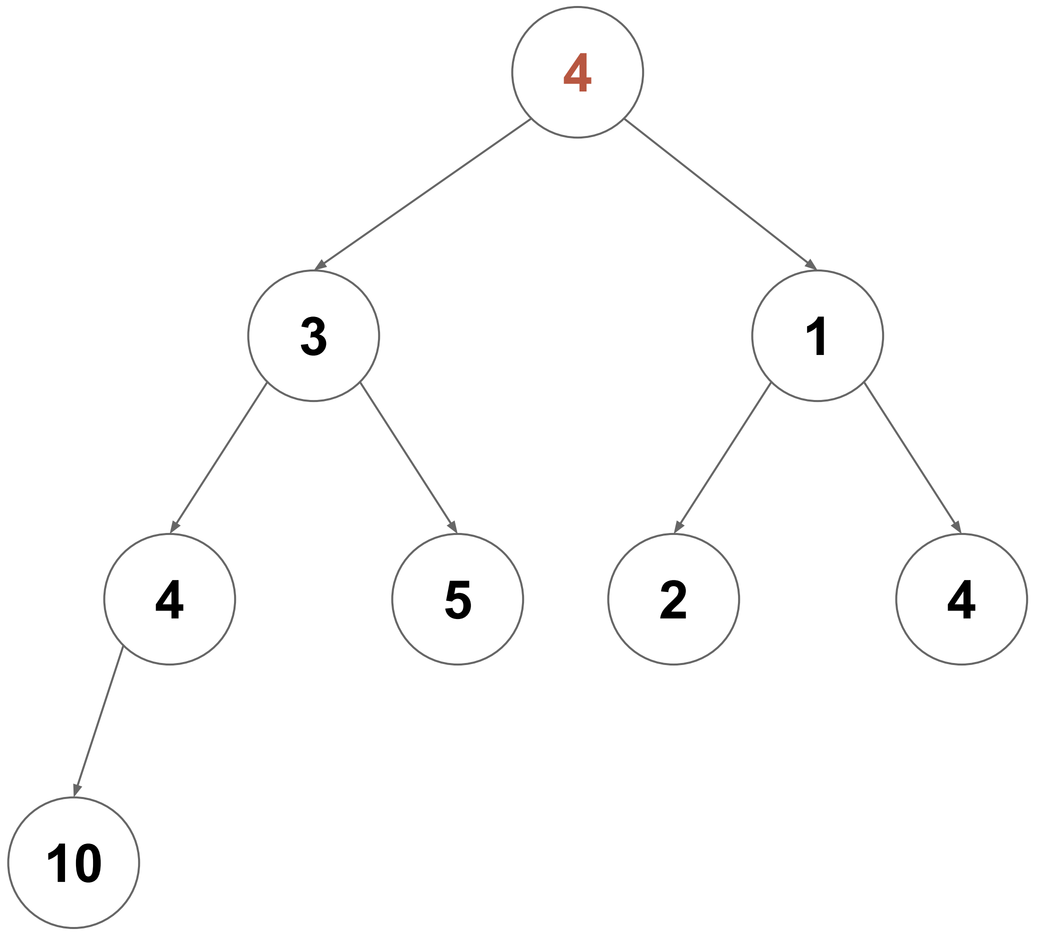
Remove and return the root



## popHeap

What if we call popHeap?

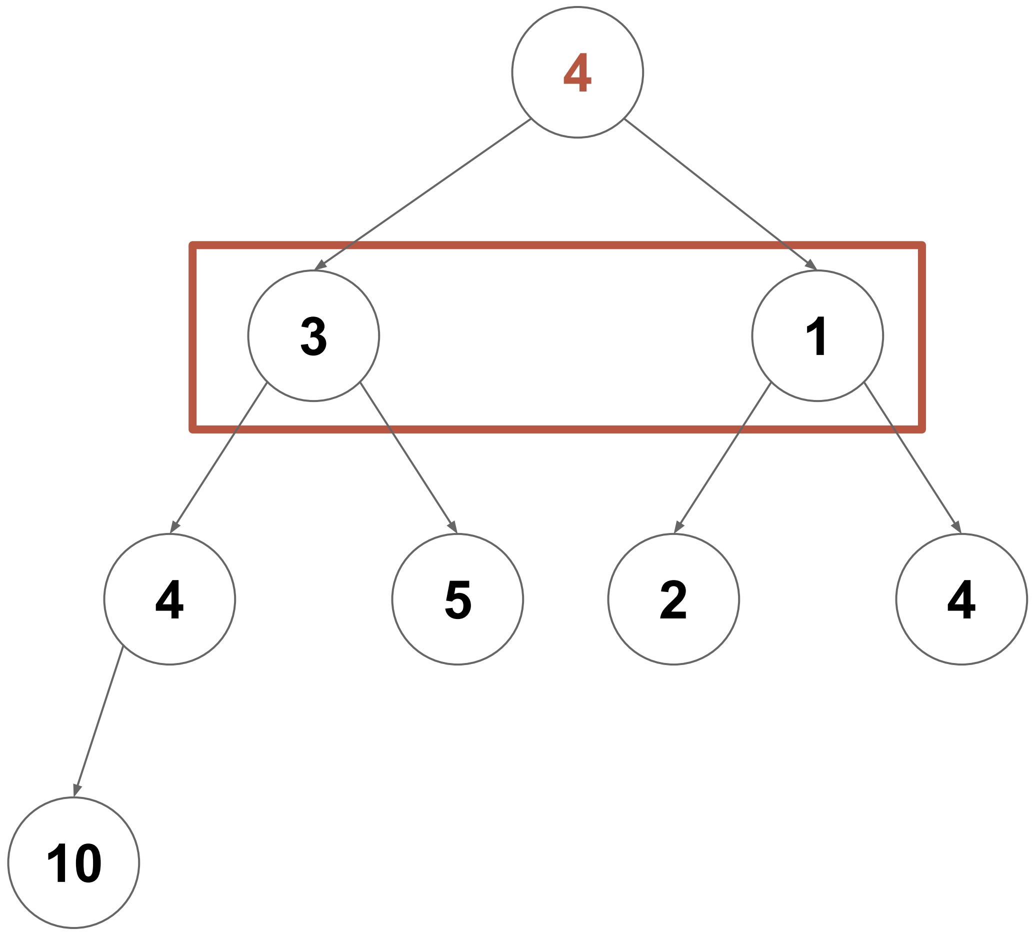
Make the last item the new root



## popHeap

What if we call popHeap?

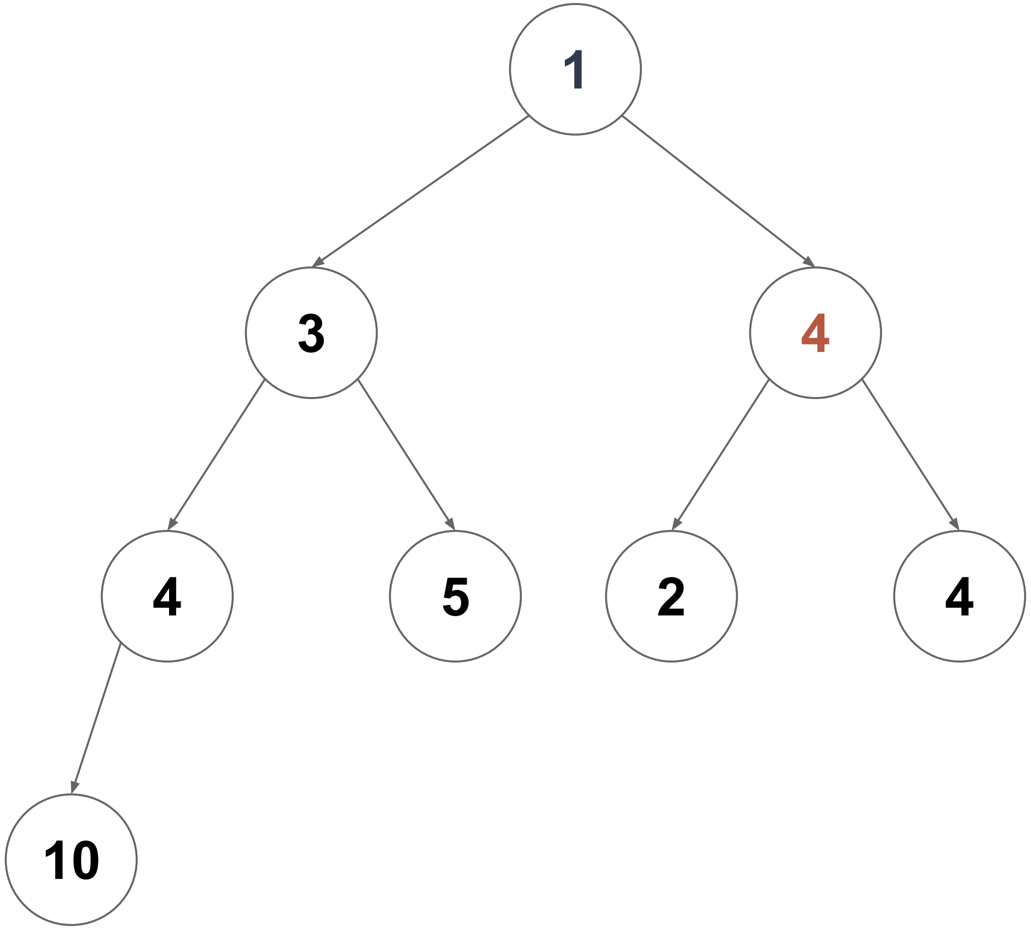
Check for our smallest child



# popHeap

What if we call popHeap?

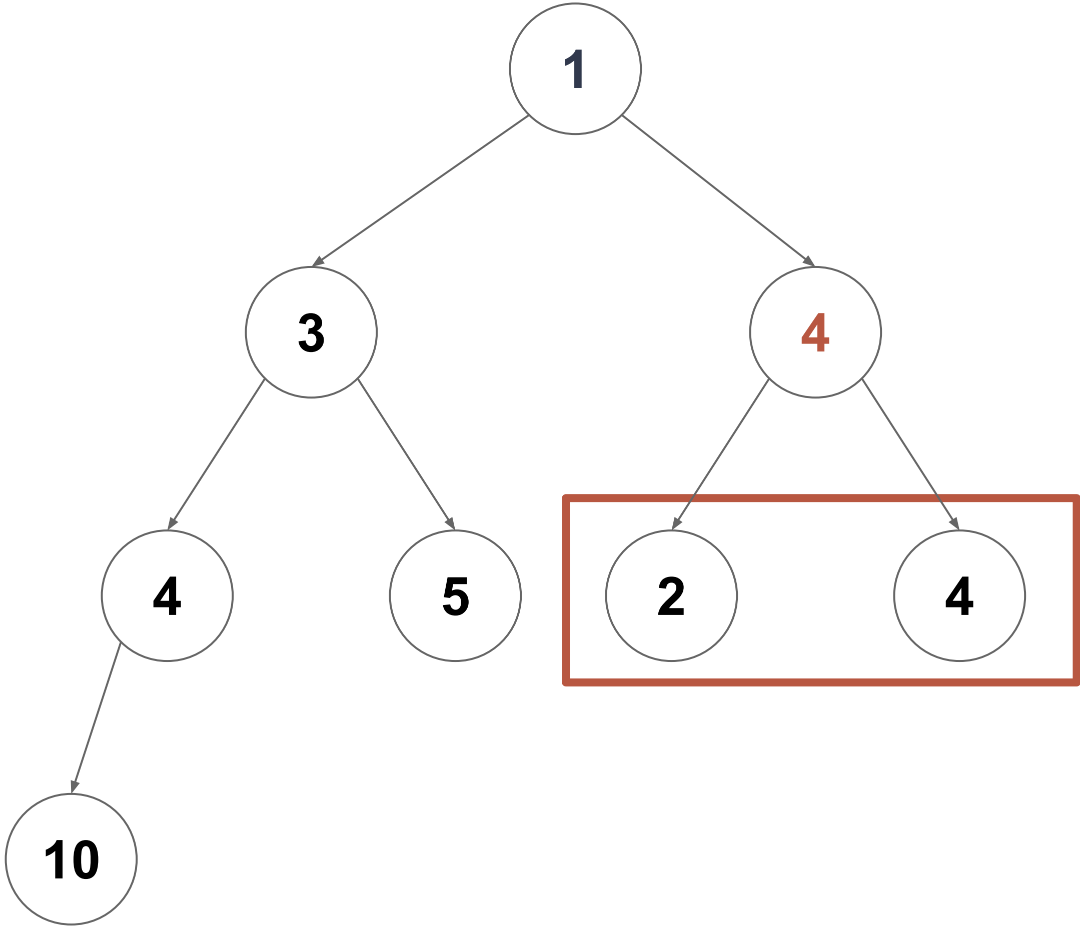
If the smallest child is smaller than us, swap



## popHeap

What if we call popHeap?

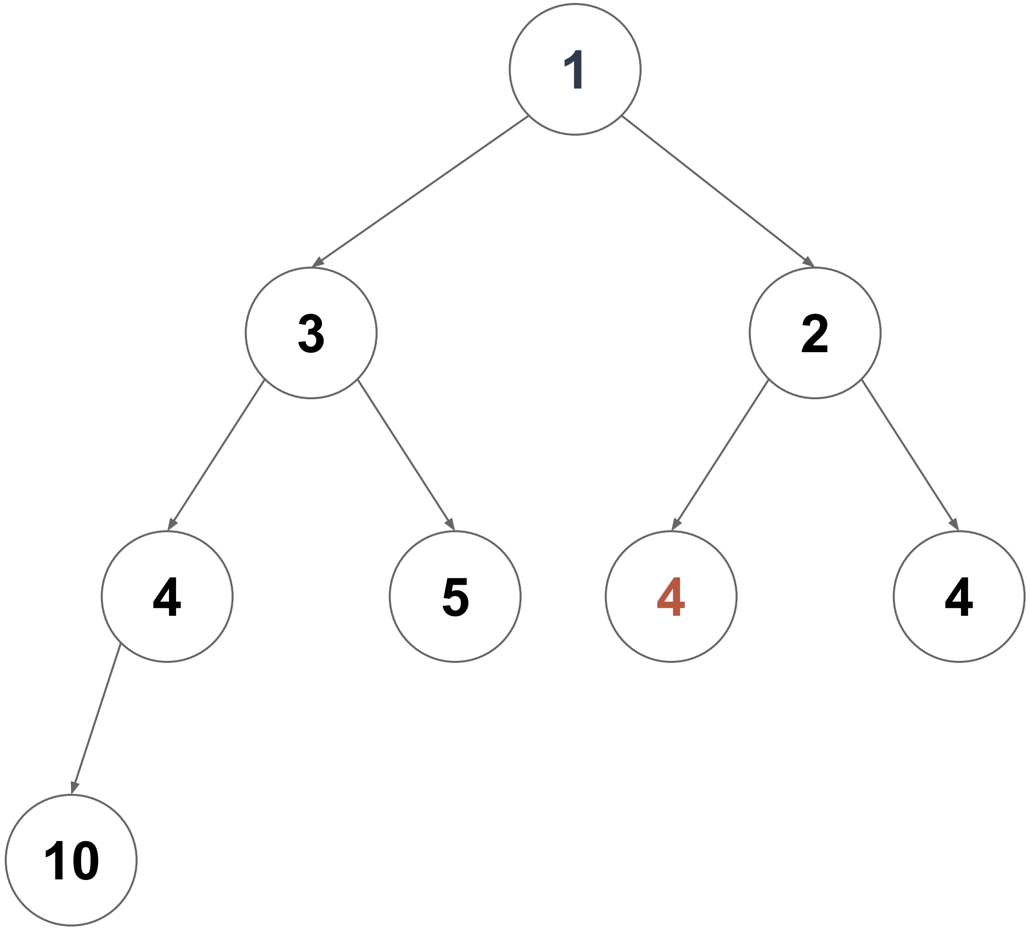
Continue swapping down the tree as necessary…



## popHeap

What if we call popHeap?

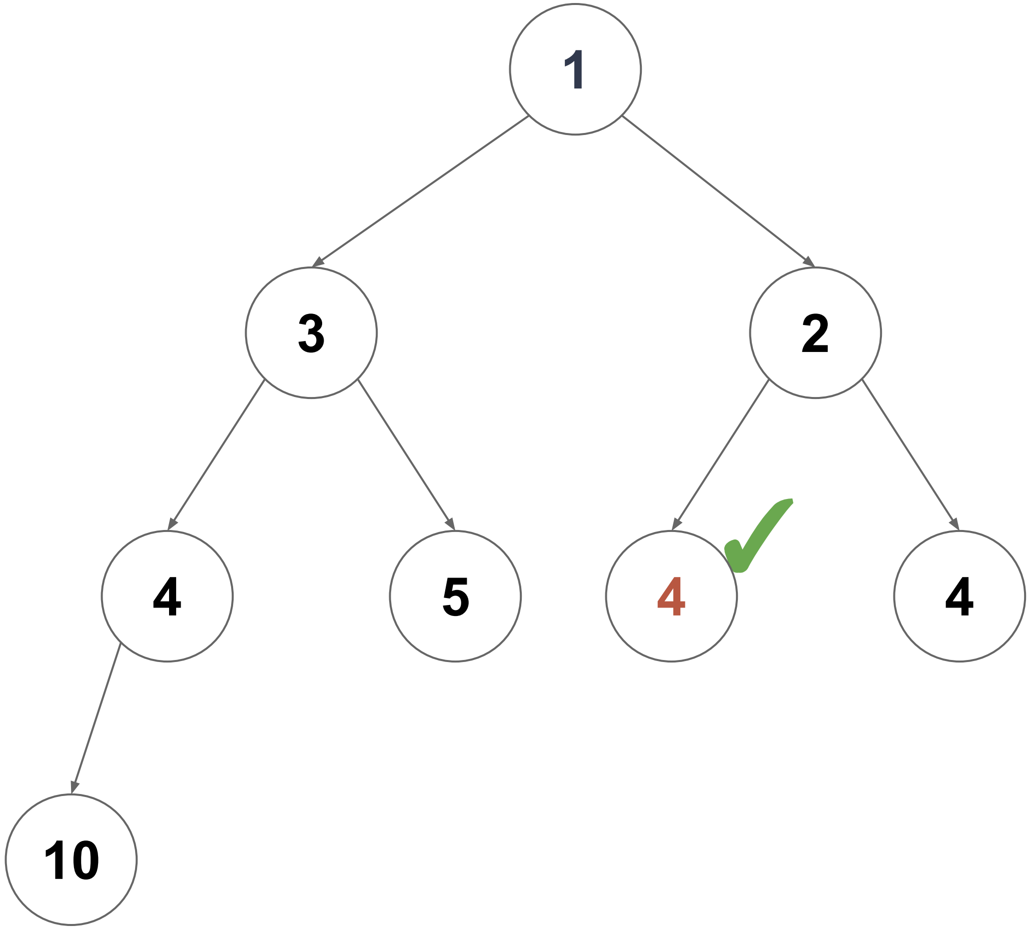
Continue swapping down the tree as necessary…



## popHeap

What if we call popHeap?

Stop swapping when our children are no longer bigger



## popHeap

**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 smallest **`child`**
    b. Set **`current = child`**

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

## popHeap

**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 smallest **`child`**
    b. Set **`current = child`**

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

## Priority Queues

| Operation | Lazy | Proactive | Heap |
|---|---|---|---|
| add | $O(1)$ | $O(n)$ | $O(\log(n))$ |
| poll | $O(n)$ | $O(1)$ | $O(\log(n))$ |
| peek | $O(n)$ | $O(1)$ | $O(1)$ |

# 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:**

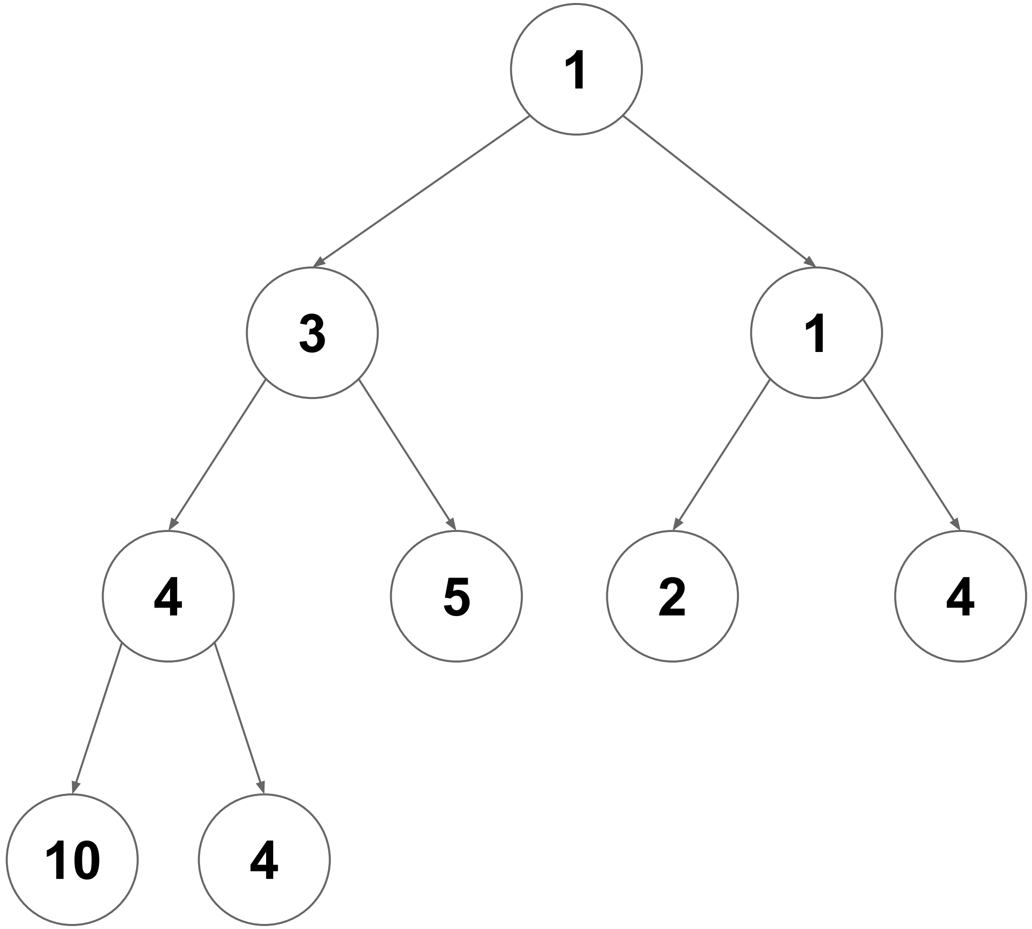
1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

*How can we compactly store a heap?*

**Idea:** Use an **`ArrayList`**

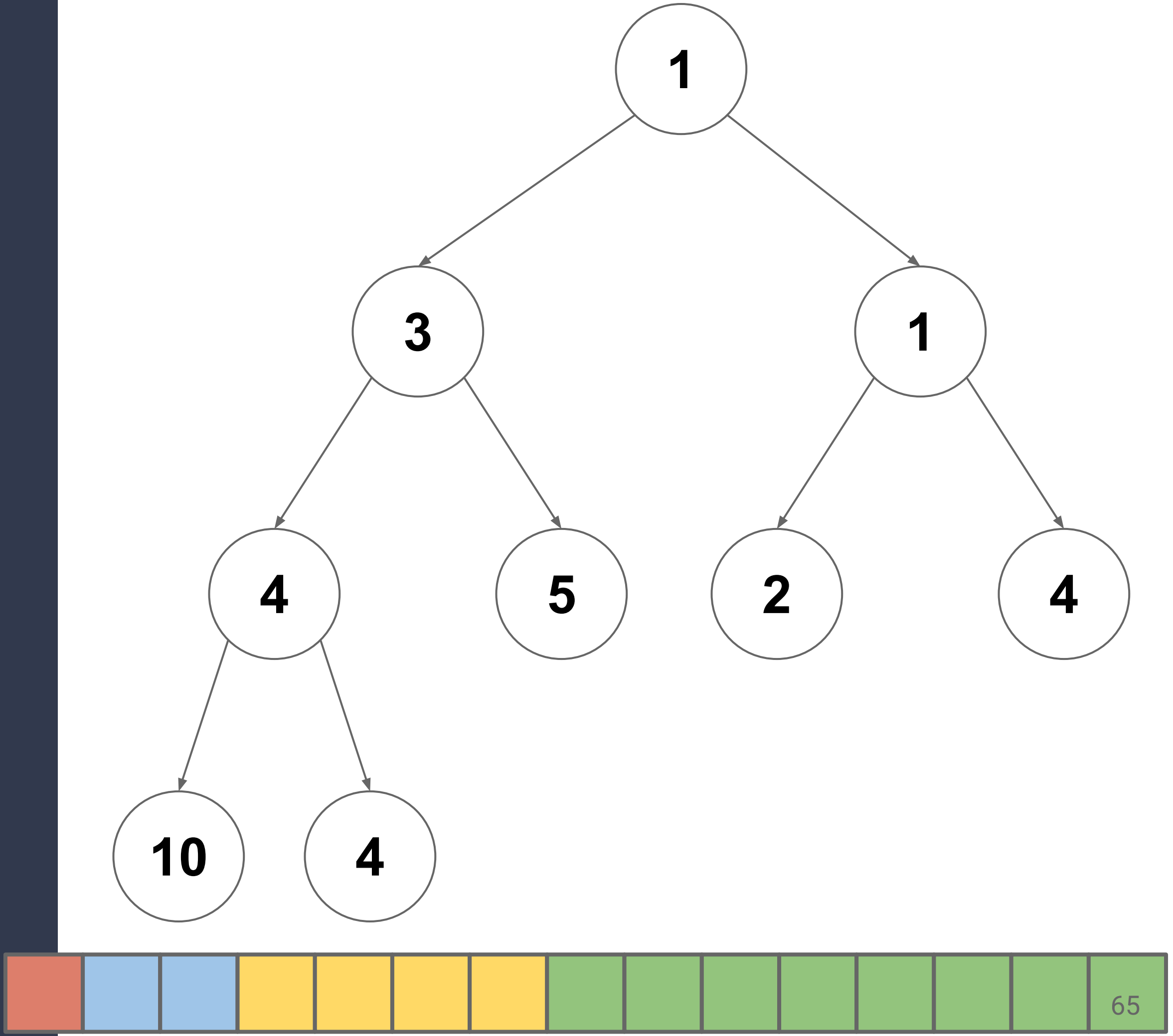
## Storing Heaps

How can we store this heap in an array buffer?



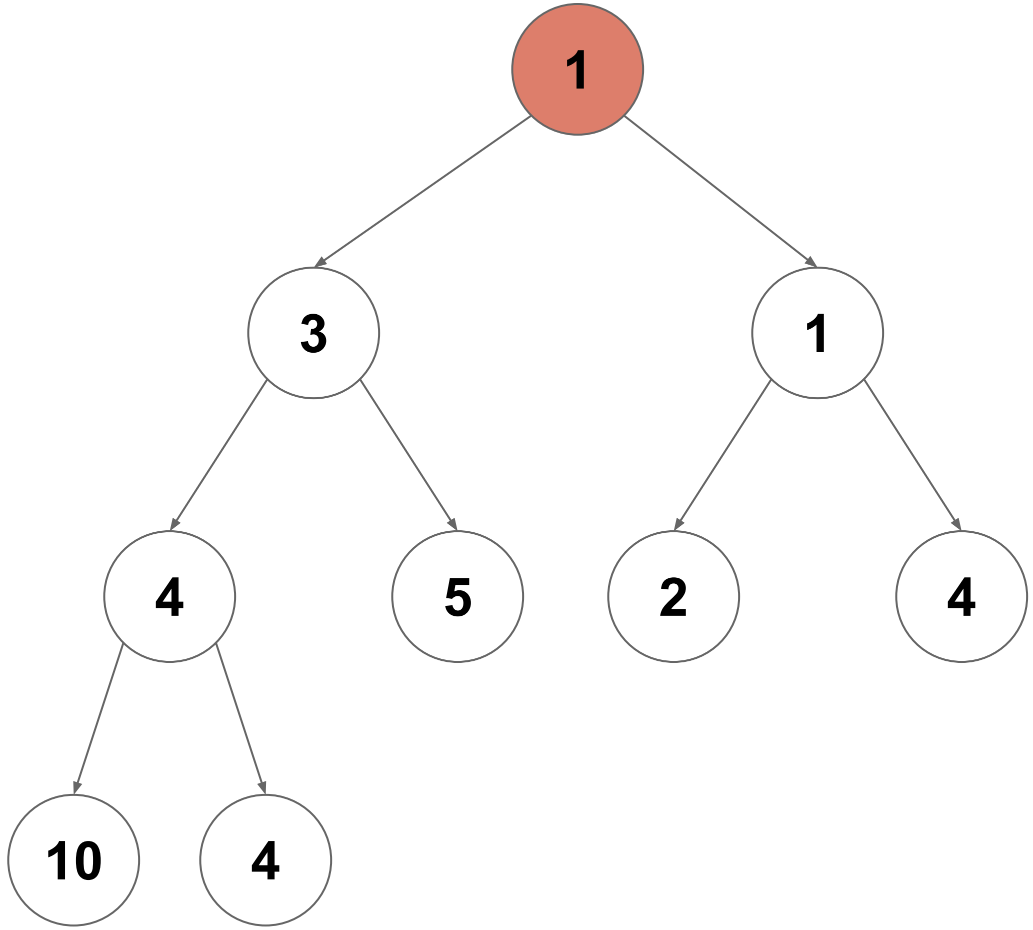
# Storing Heaps

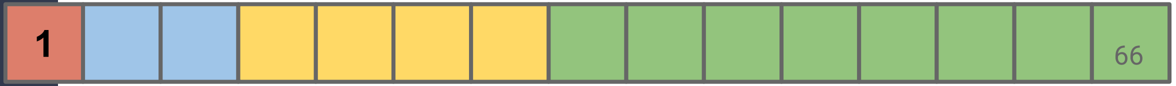
How can we store this heap in an array buffer?



# Storing Heaps

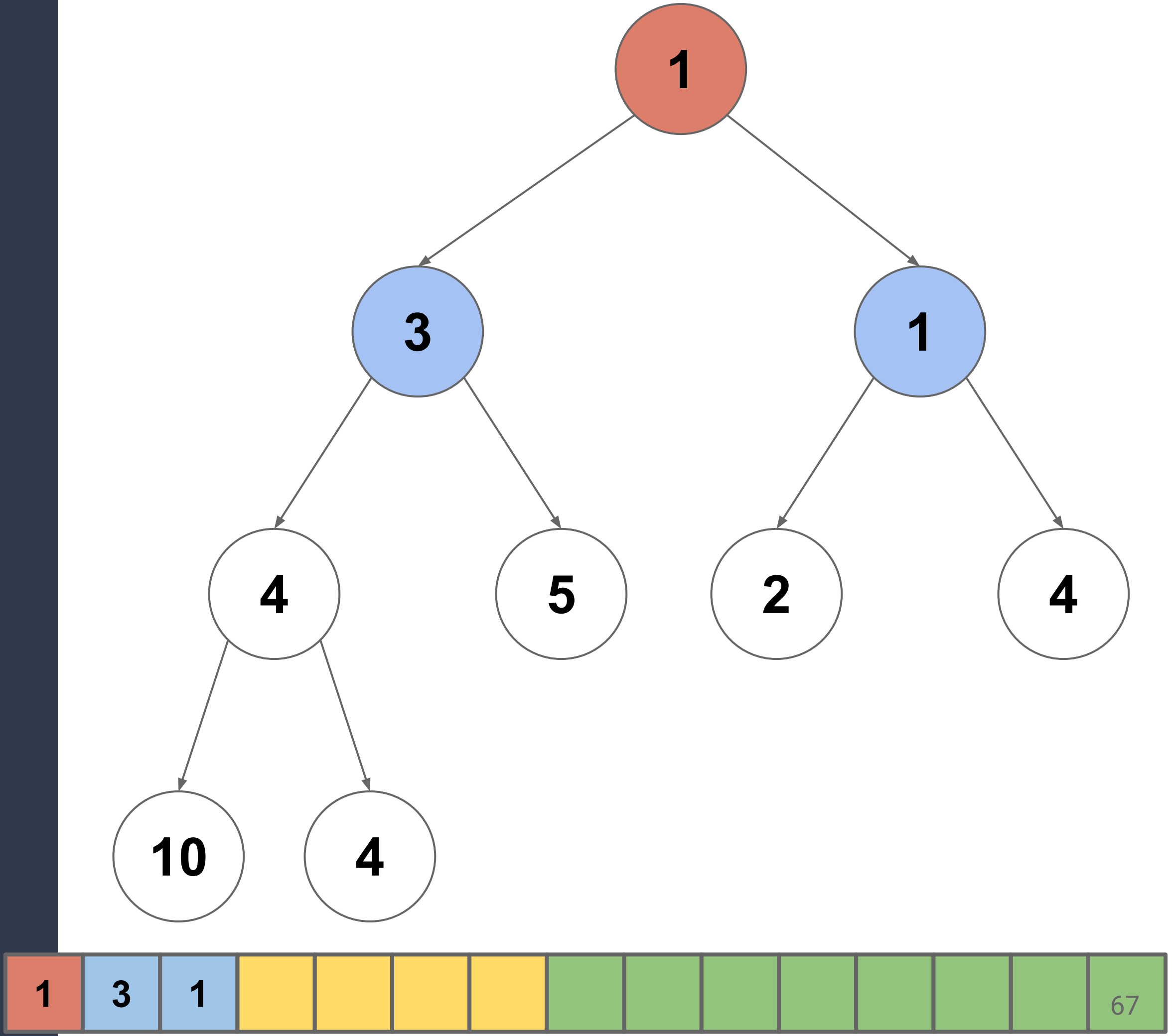
How can we store this heap in an array buffer?





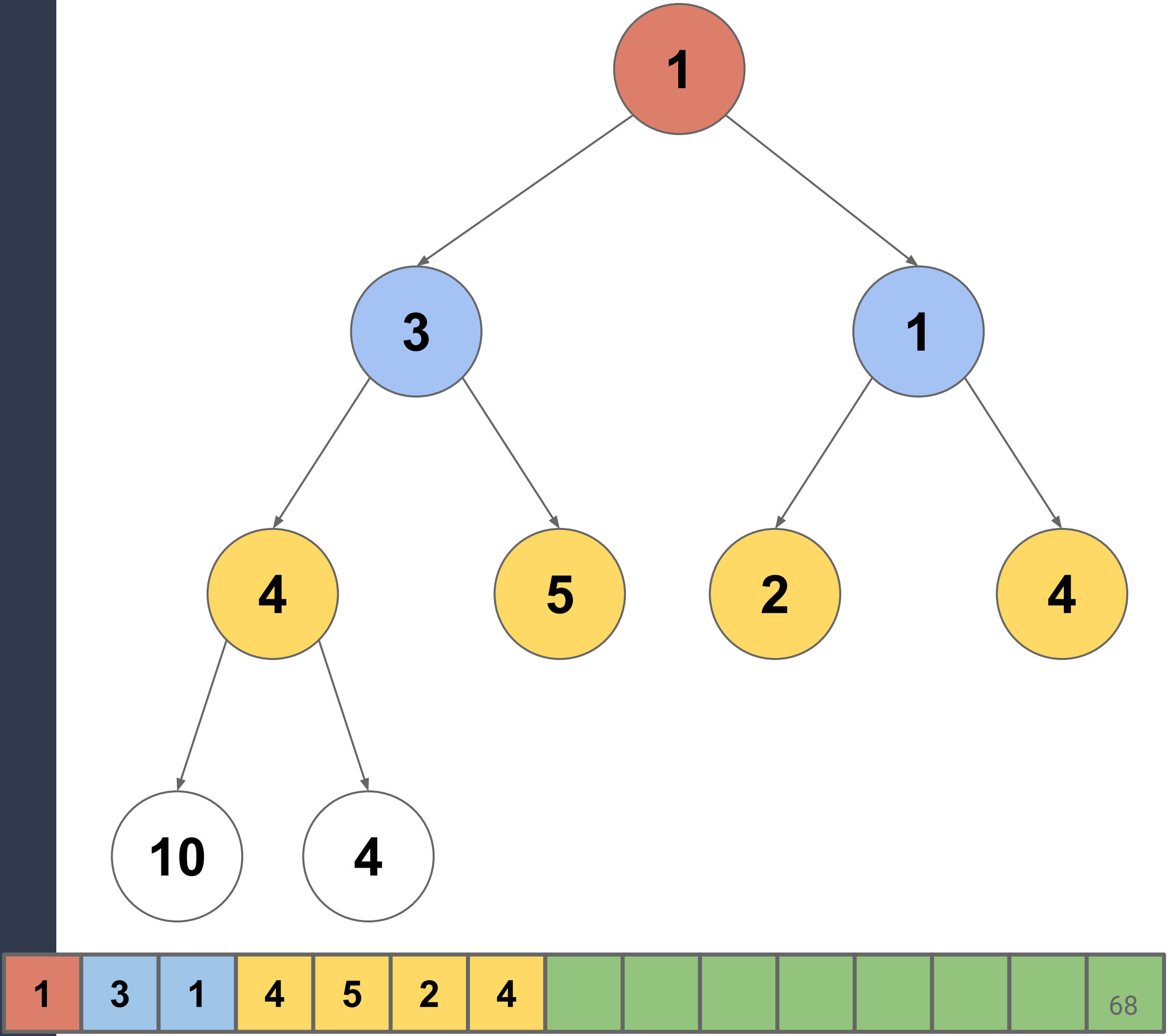
## Storing Heaps

How can we store this heap in an array buffer?



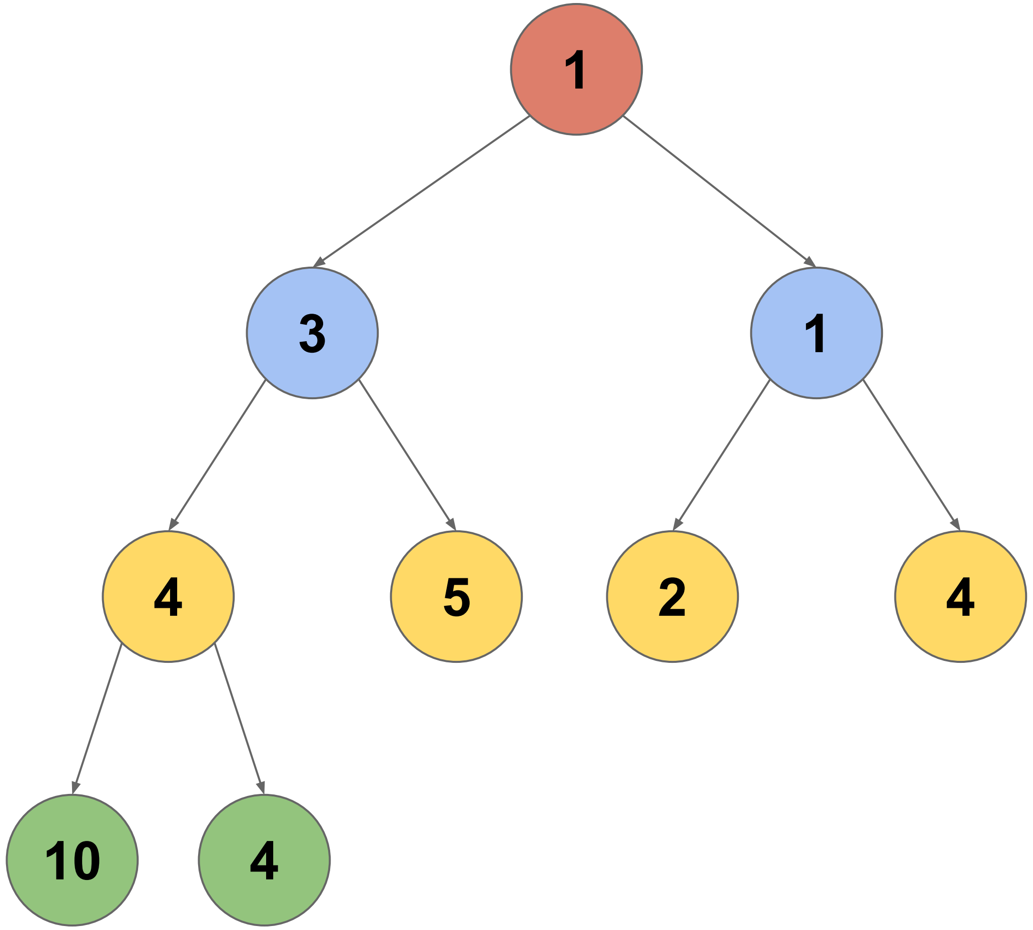
## Storing Heaps

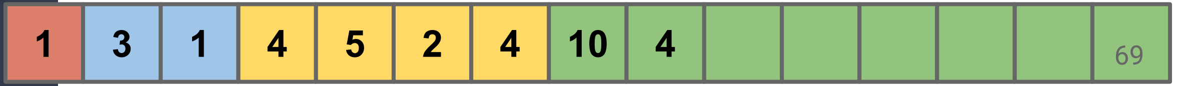
How can we store this heap in an array buffer?



# Storing Heaps

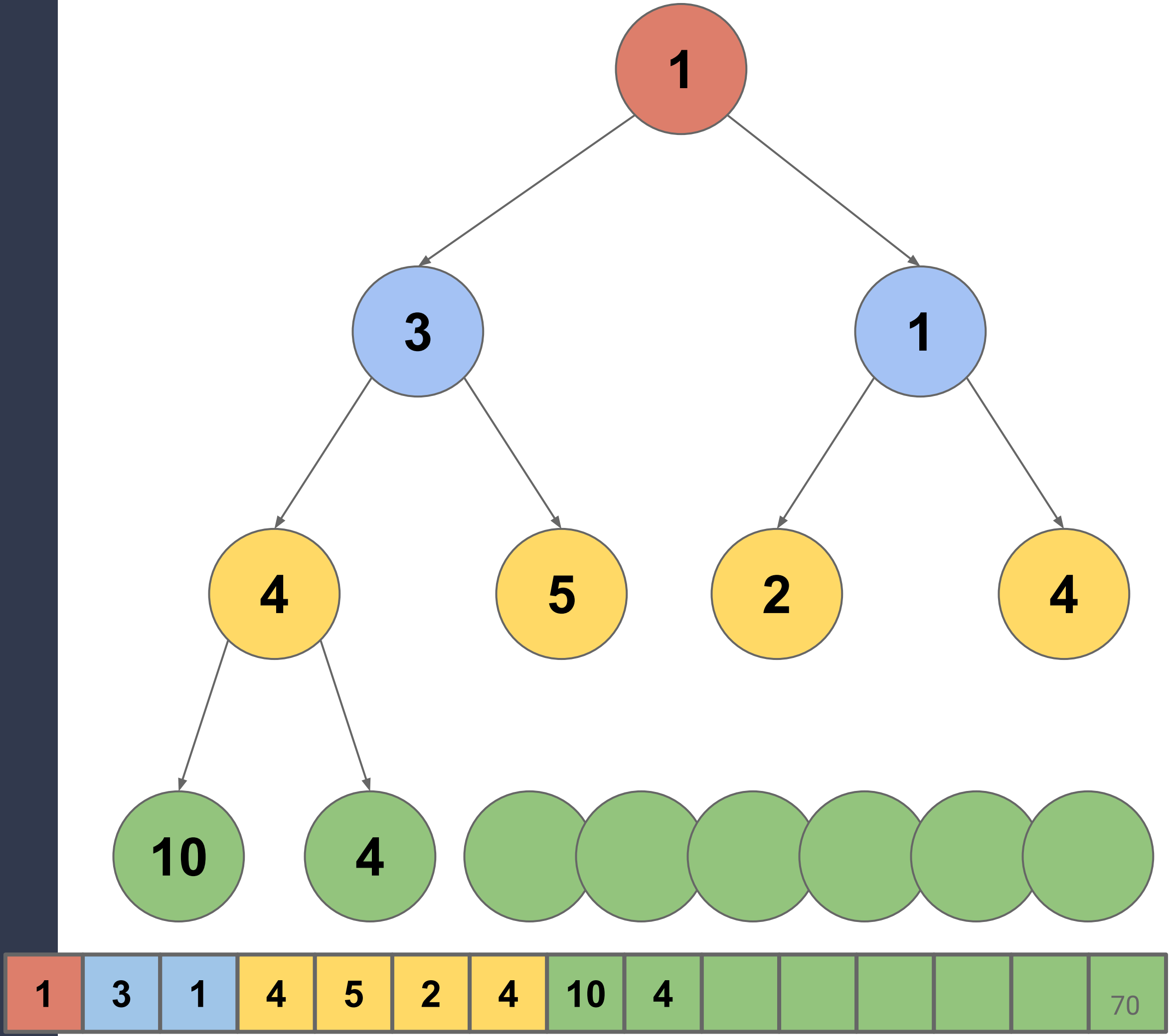
How can we store this heap in an array buffer?





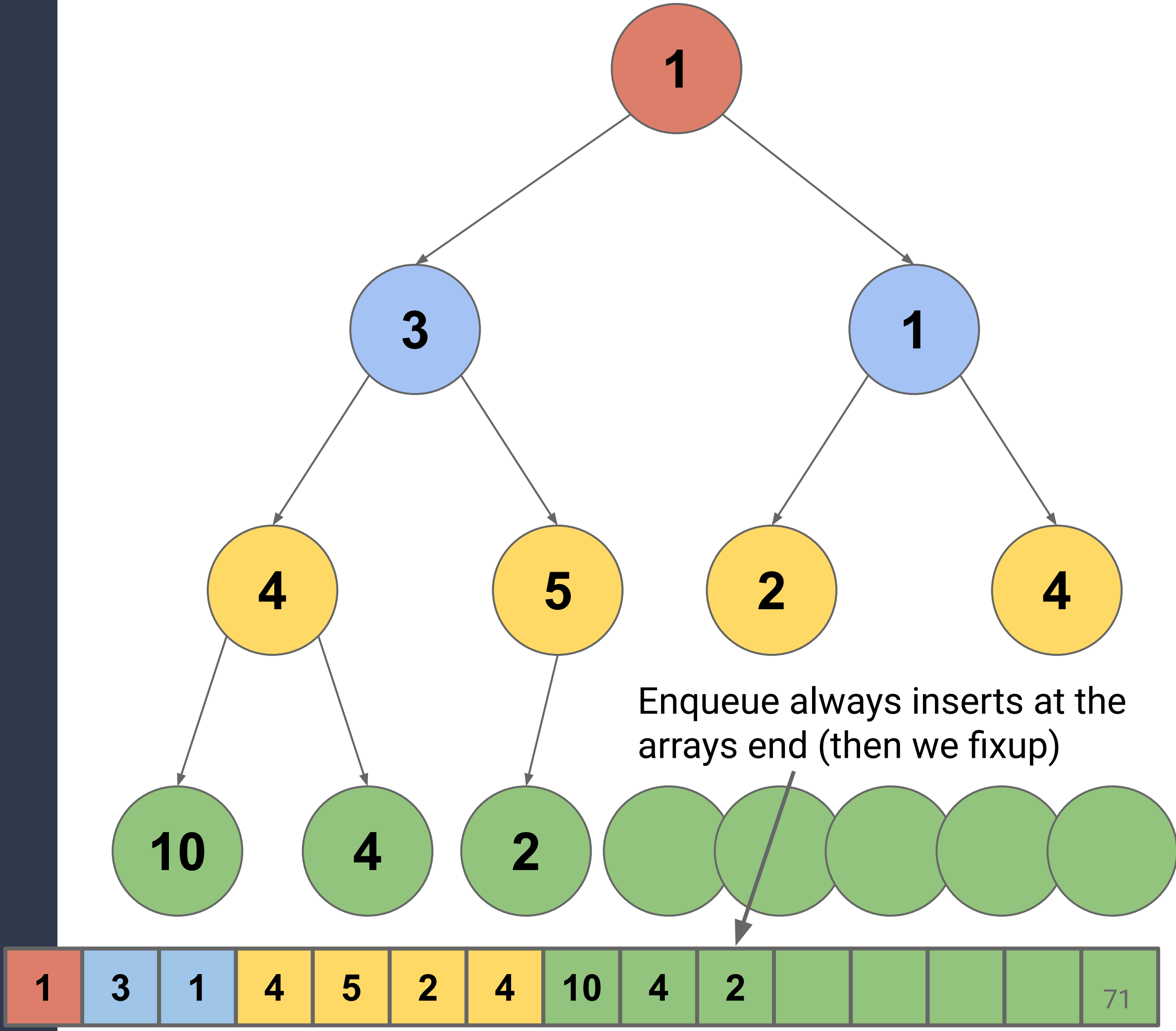
# Storing Heaps

How can we store this heap in an array buffer?



# Storing Heaps

How can we store this heap in an array buffer?



## Runtime Analysis

**`pushHeap`**

# Runtime Analysis

`pushHeap`

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

## Runtime Analysis

**`pushHeap`**

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

## Runtime Analysis

**`pushHeap`**

- **Append to `ArrayList`:** 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

`pushHeap`

- **Append to `ArrayList`:** 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 $O(n)$)*

`popHeap`

## Runtime Analysis

`pushHeap`

- **Append to `ArrayList`:** 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 $O(n)$)*

`popHeap`

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

## Runtime Analysis

`pushHeap`

- **Append to `ArrayList`:** 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))*

`popHeap`

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

## Runtime Analysis

**`pushHeap`**

- **Append to `ArrayList`**: 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 $O(n)$)*

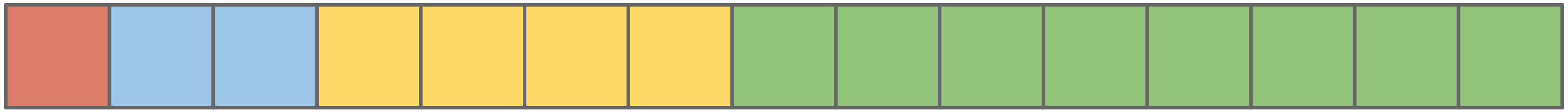
**`popHeap`**

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

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

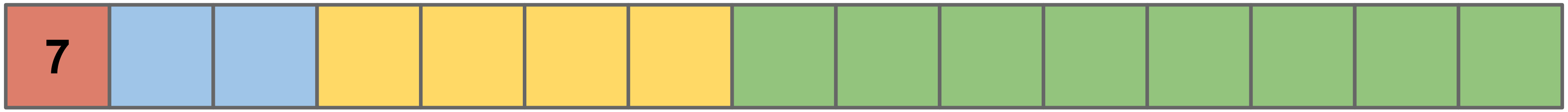
7, 4, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

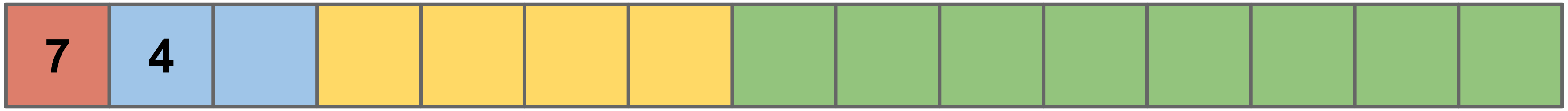
7, 4, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

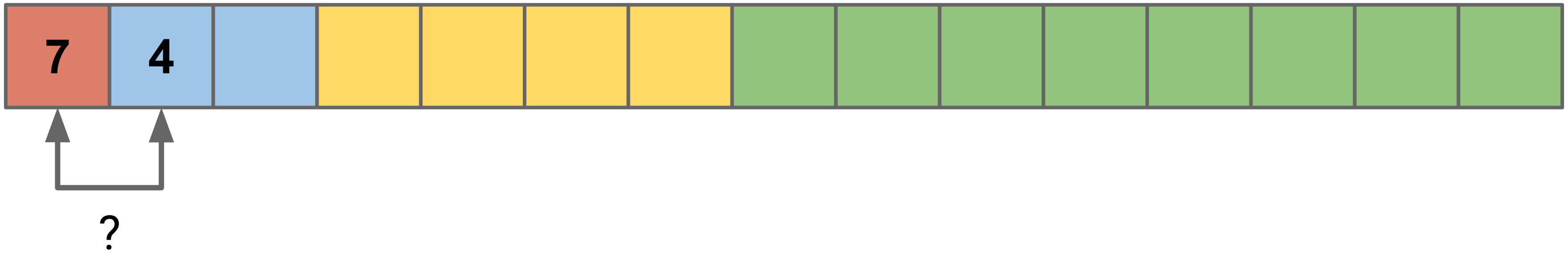
7, ***<u>4</u>***, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

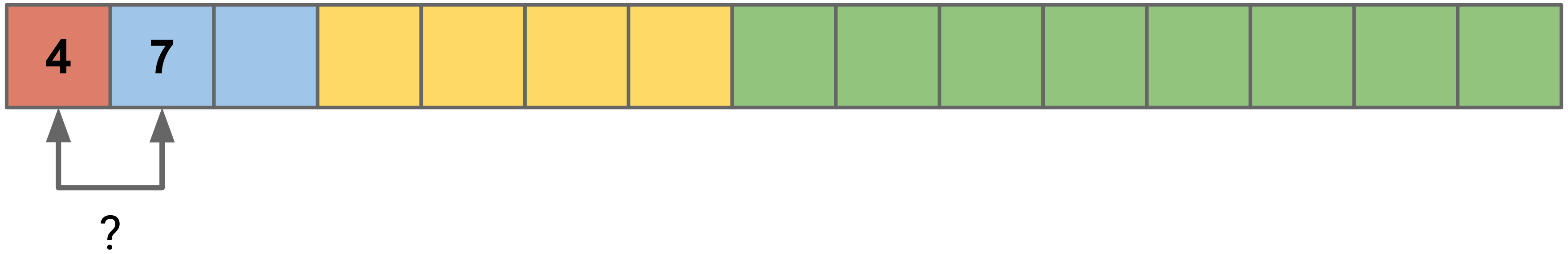
7, **<u>4</u>**, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

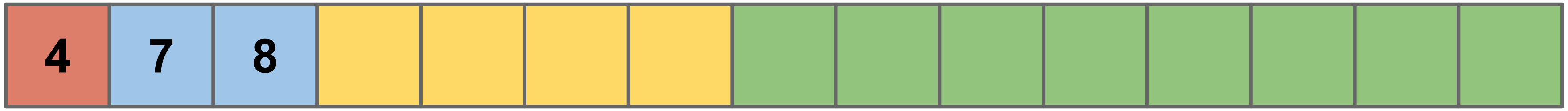
7, <u>**4**</u>, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

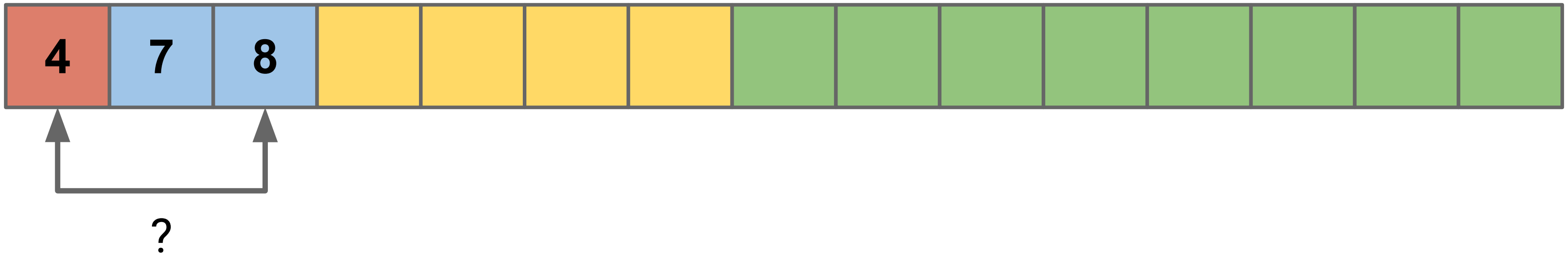
7, 4, **8**, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

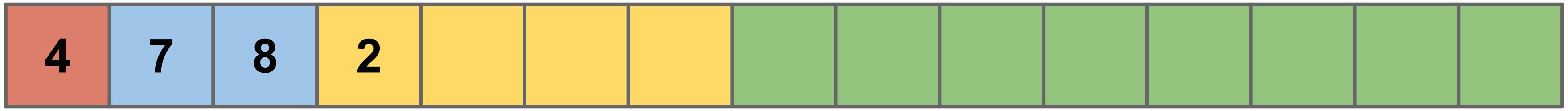
7, 4, **<u>8</u>**, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

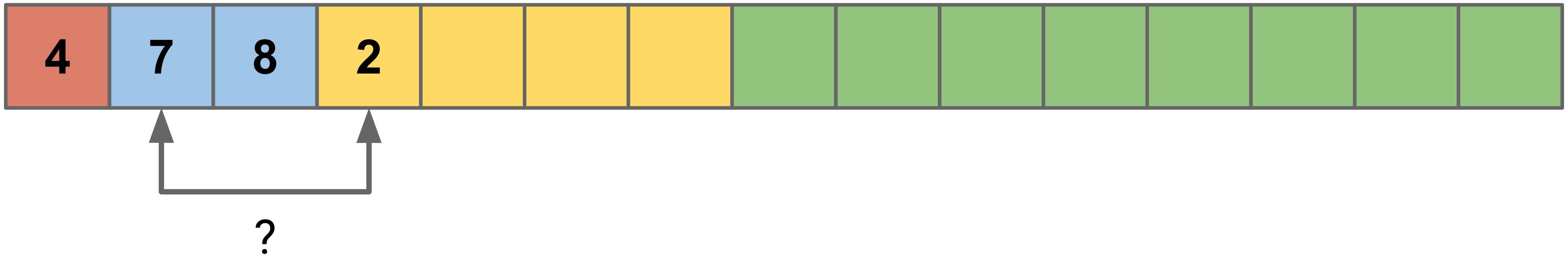
7, 4, 8, **<u>2</u>**, 5, 3, 9



# Heap Sort

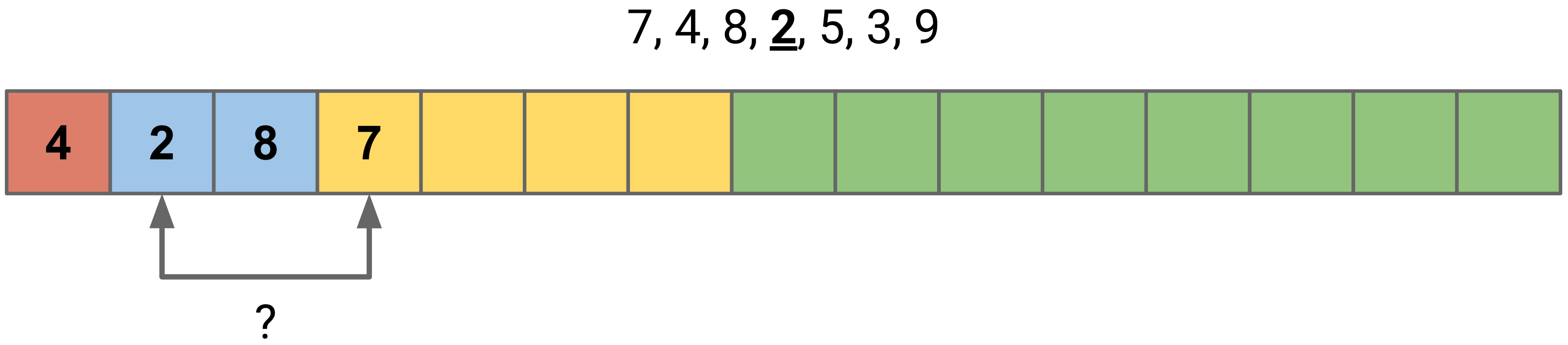
1. Insert items into heap
2. Reconstruct sequence with dequeue

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



# Heap Sort

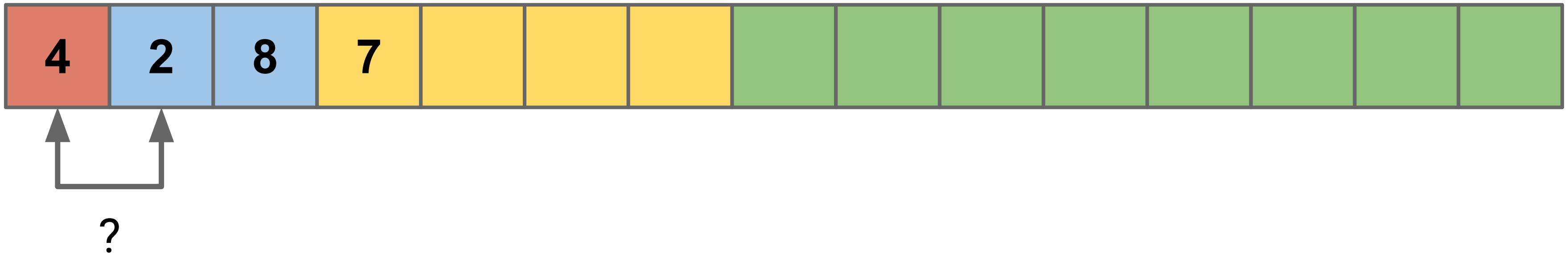
1. Insert items into heap
2. Reconstruct sequence with dequeue



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

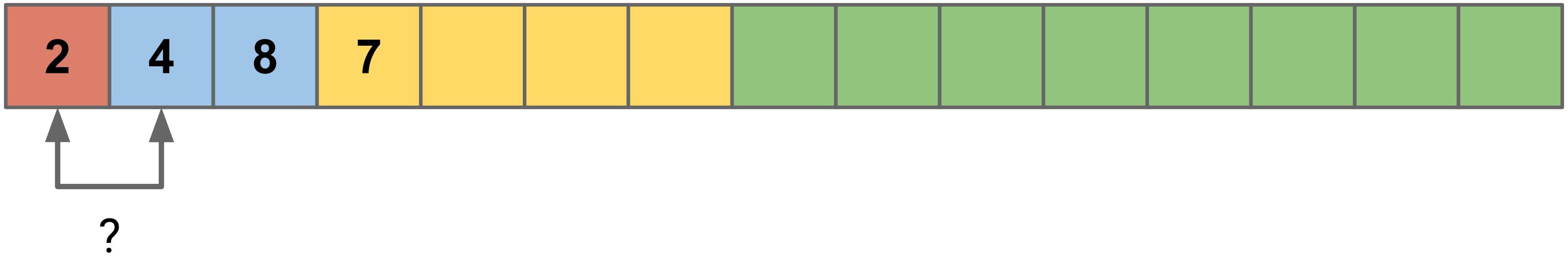
7, 4, 8, **2**, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

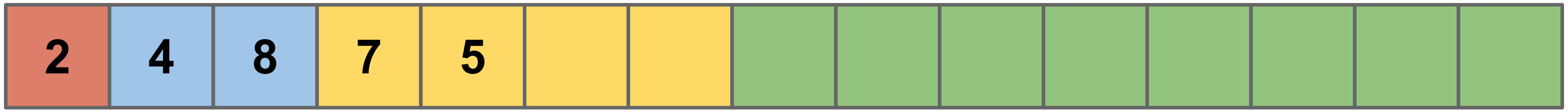
7, 4, 8, **2**, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

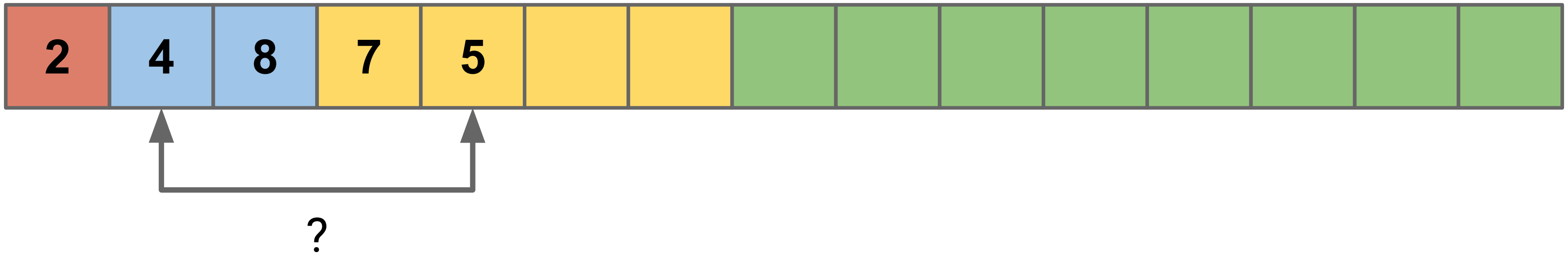
7, 4, 8, 2, **<u>5</u>**, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

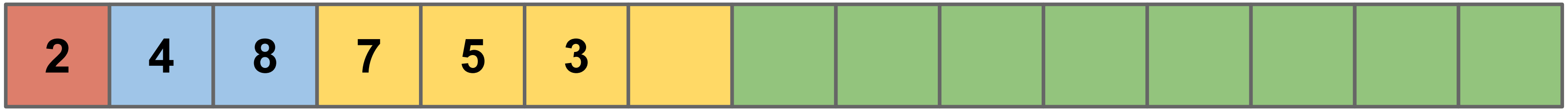
7, 4, 8, 2, **5**, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

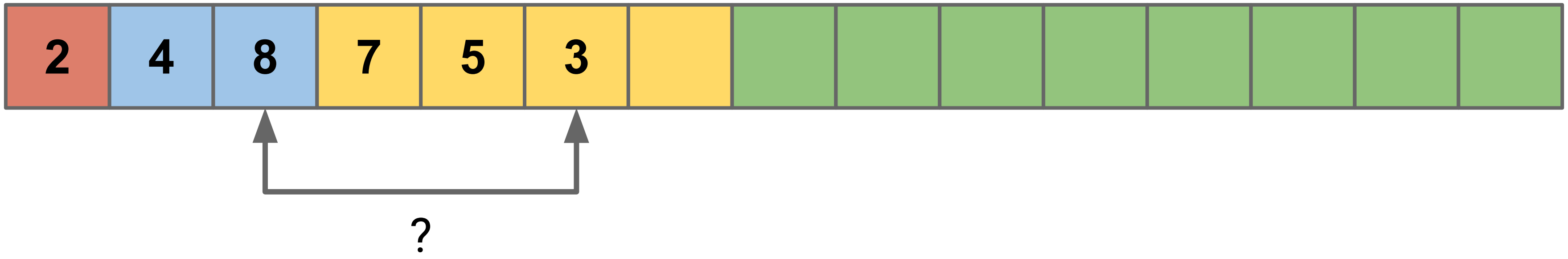
7, 4, 8, 2, 5, **3**, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

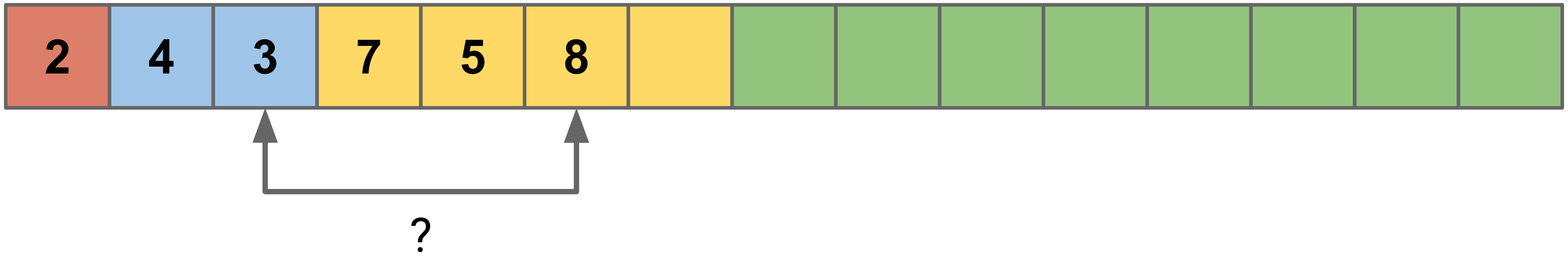
7, 4, 8, 2, 5, **3**, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

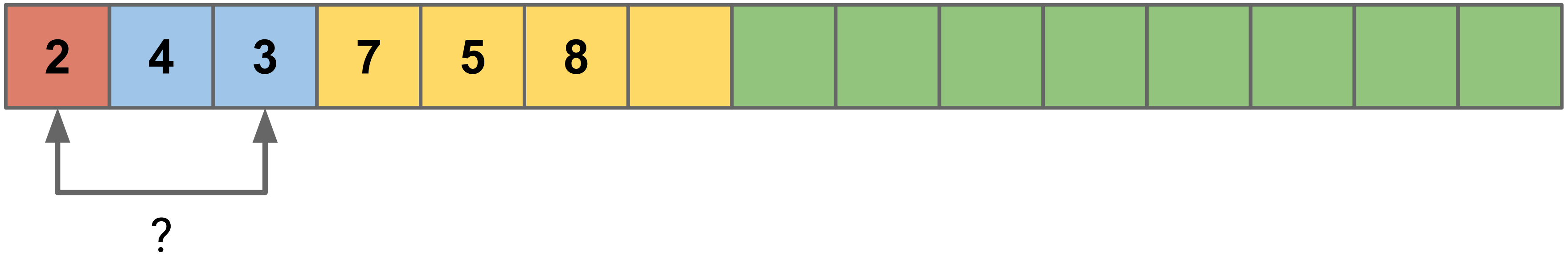
7, 4, 8, 2, 5, **3**, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

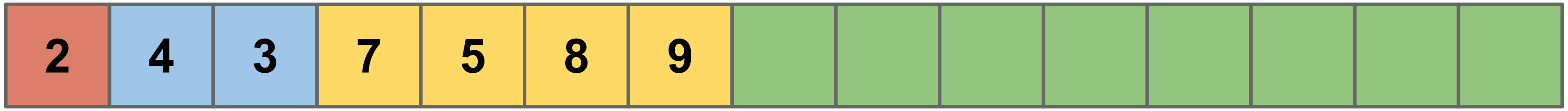
7, 4, 8, 2, 5, **3**, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

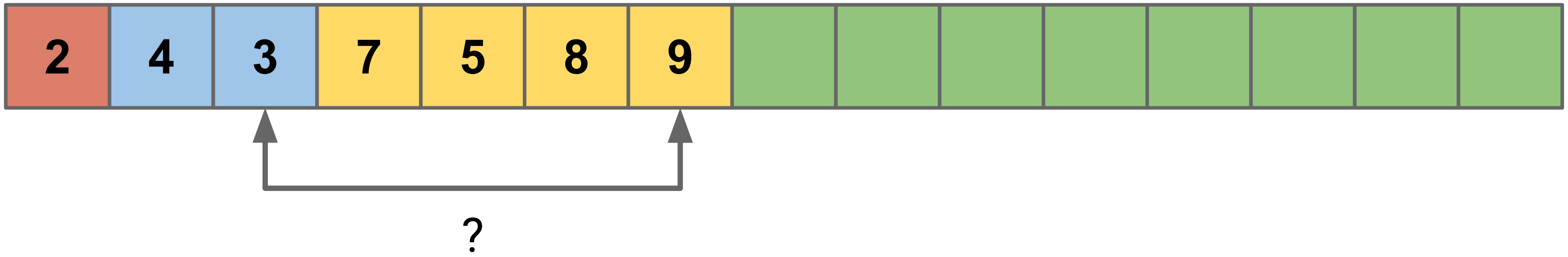
7, 4, 8, 2, 5, 3, **9**



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

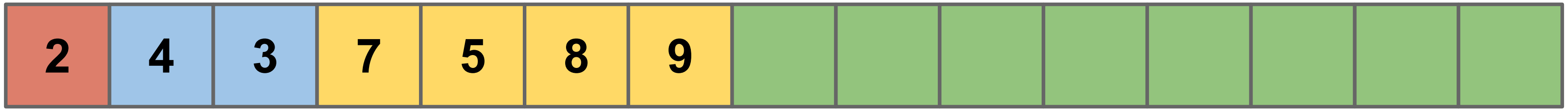
7, 4, 8, 2, 5, 3, **9**



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

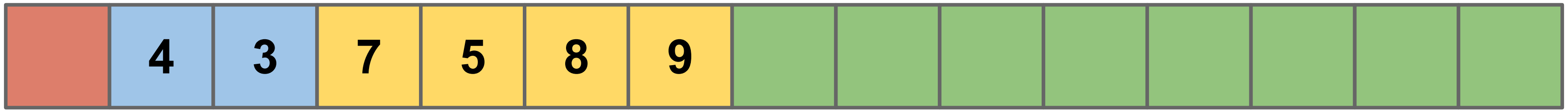
7, 4, 8, 2, 5, 3, 9



# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

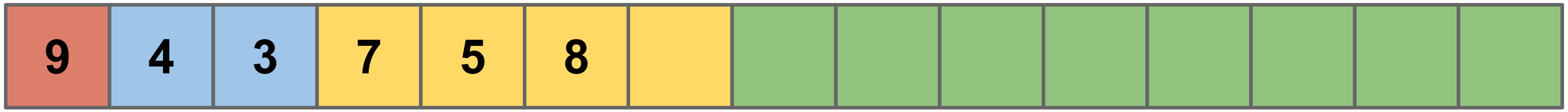


2

## Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

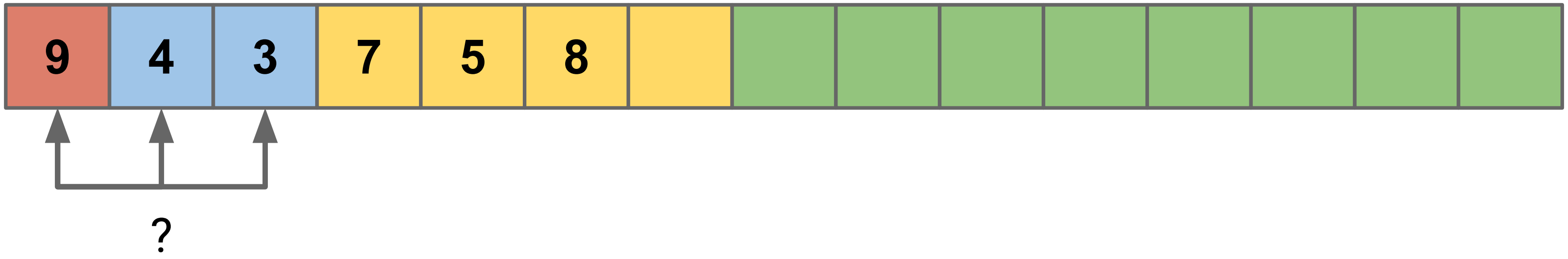


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

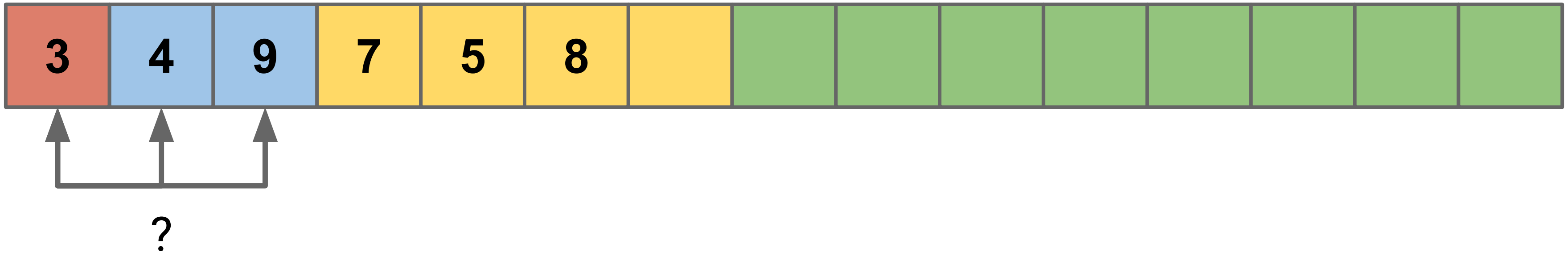


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

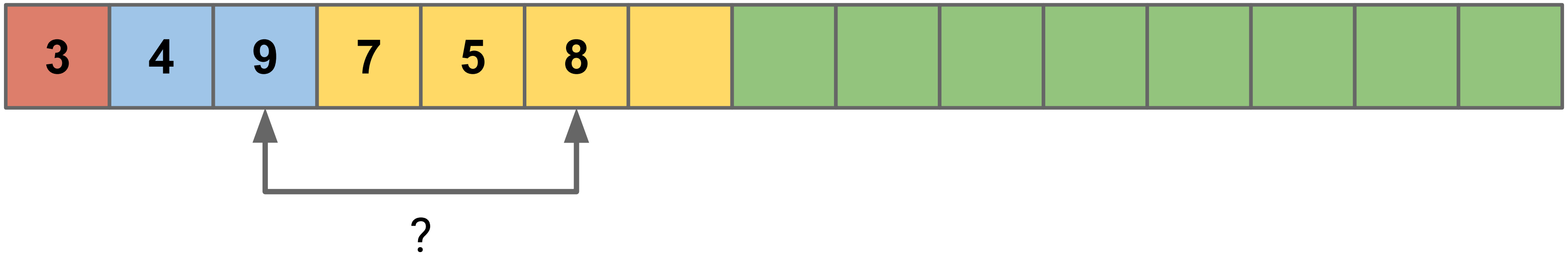


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

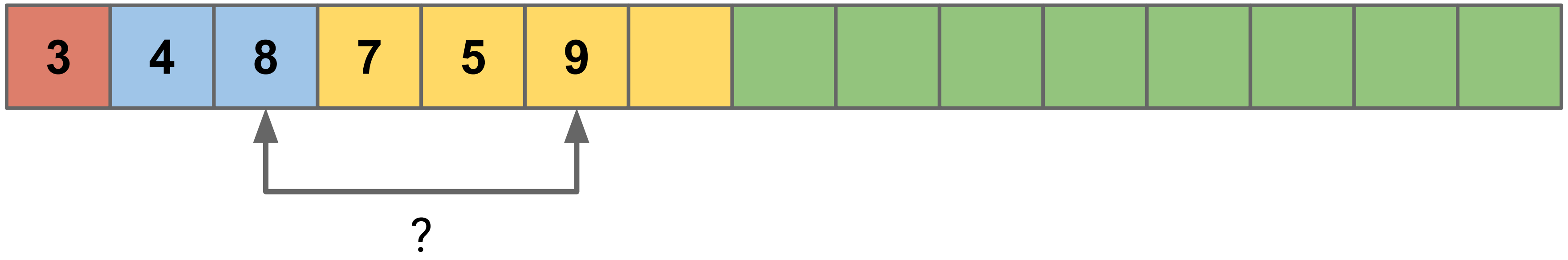


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

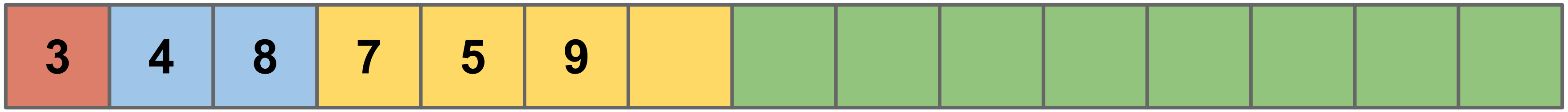


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

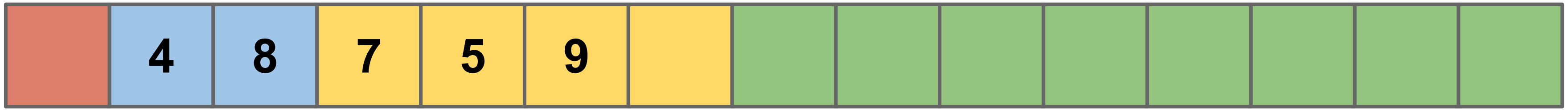


2

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

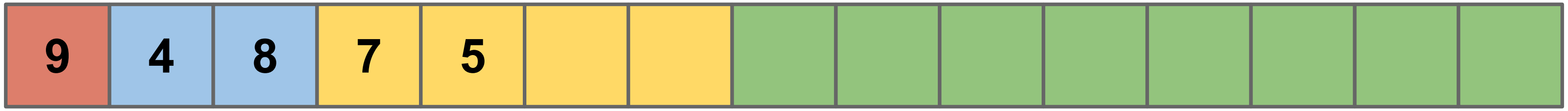


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

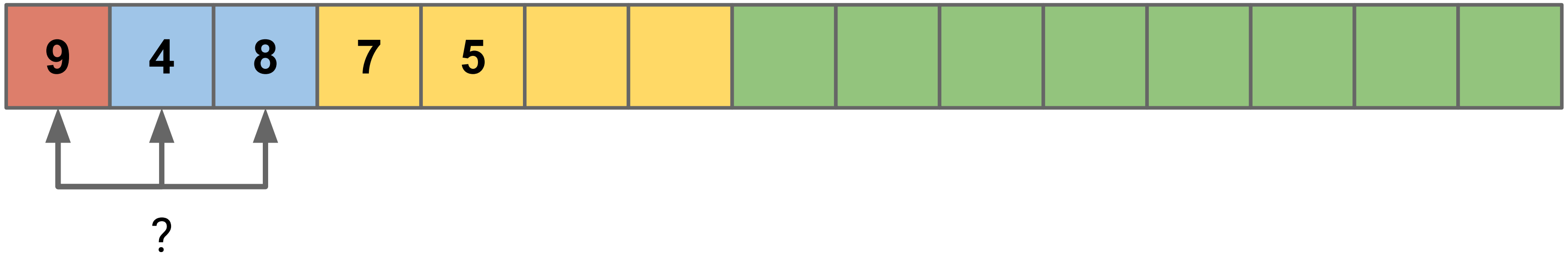


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

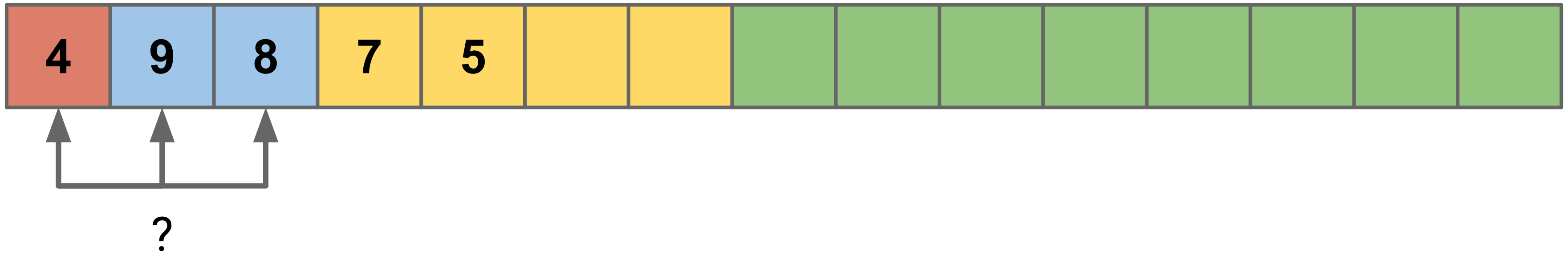


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

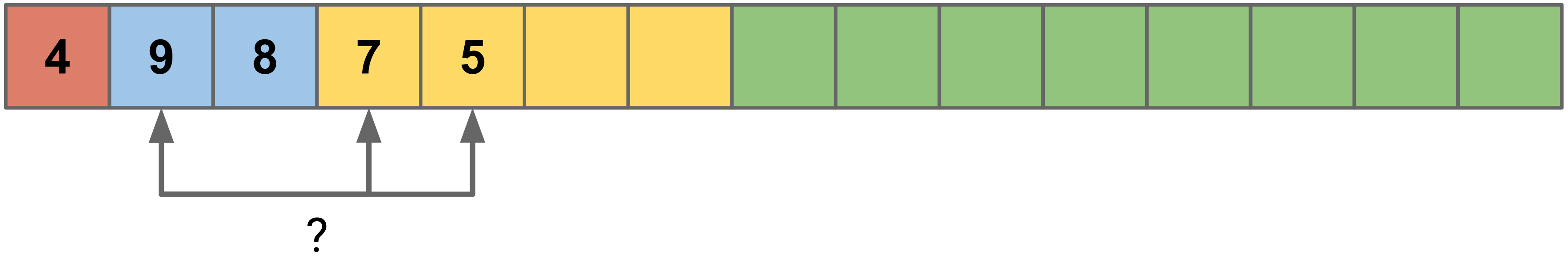


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

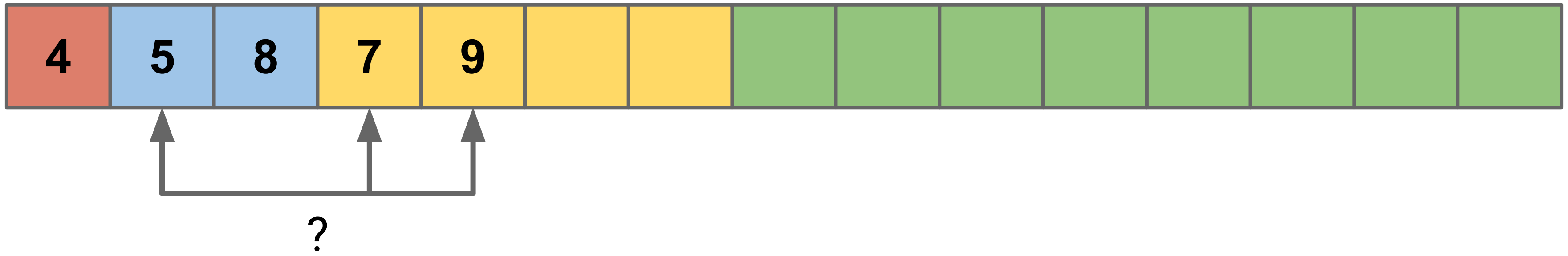


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

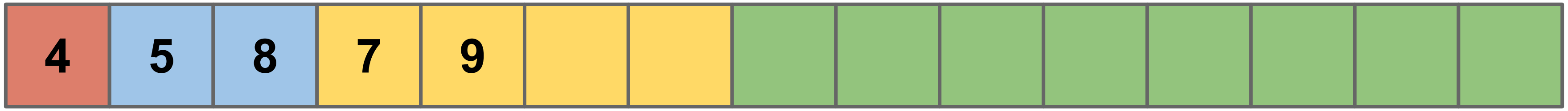


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

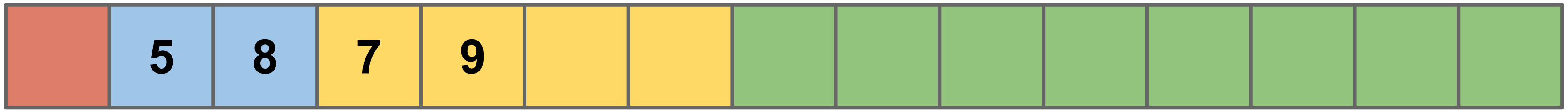


2, 3

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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

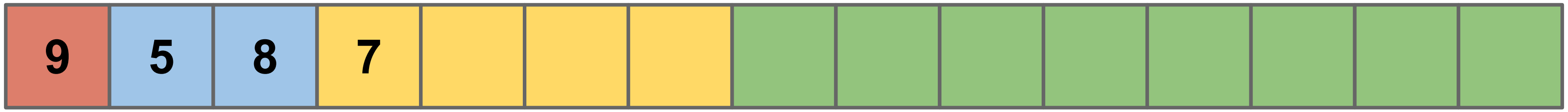


2, 3, 4

# Heap Sort

1. Insert items into heap
2. Reconstruct sequence with dequeue

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



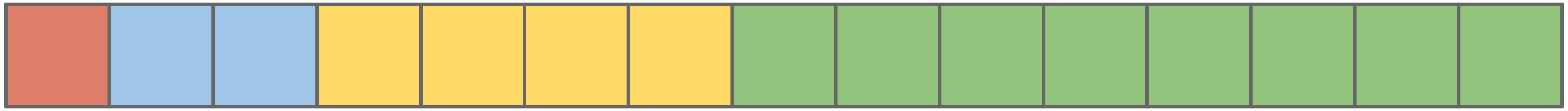
2, 3, 4

## A few moments later…

## Heap Sort

1. Insert items into heap
2. Reconstruct sequence 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?*

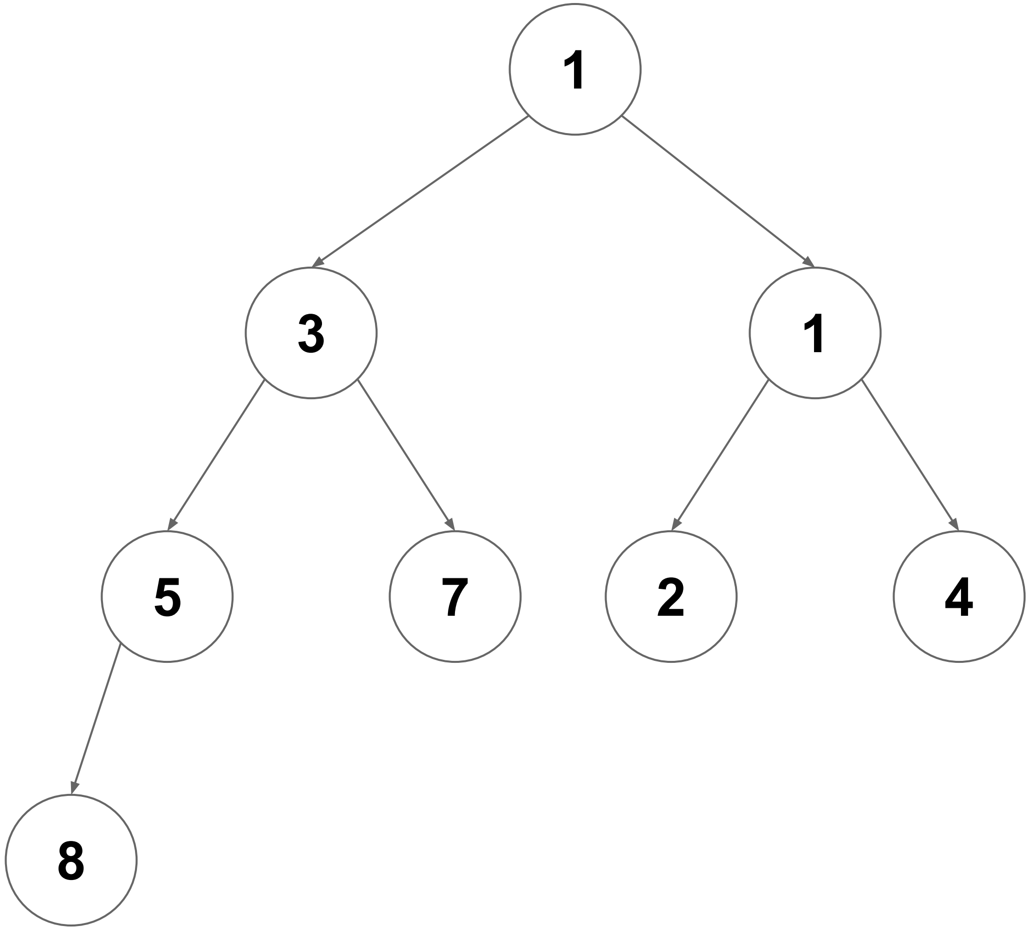
# Updating Heap Elements

*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

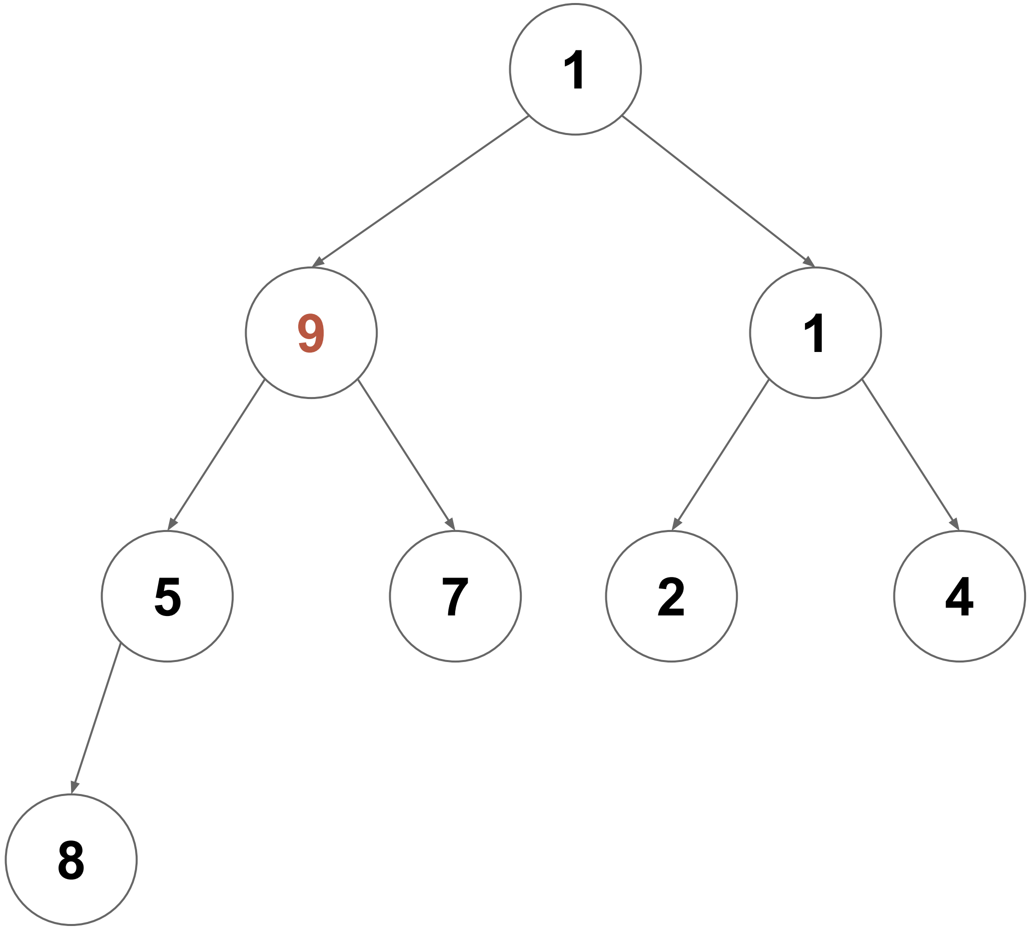
## `update`

What if we change the value of the 3 node to 9?



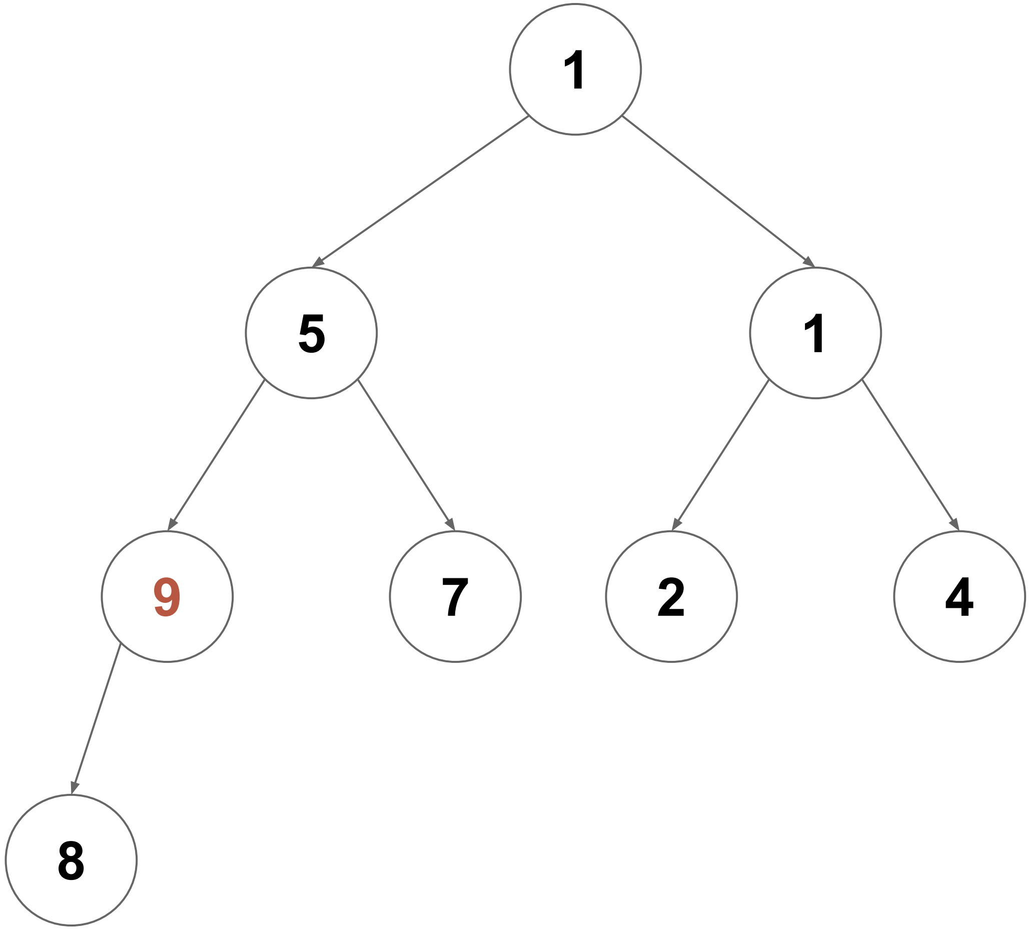
## `update`

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



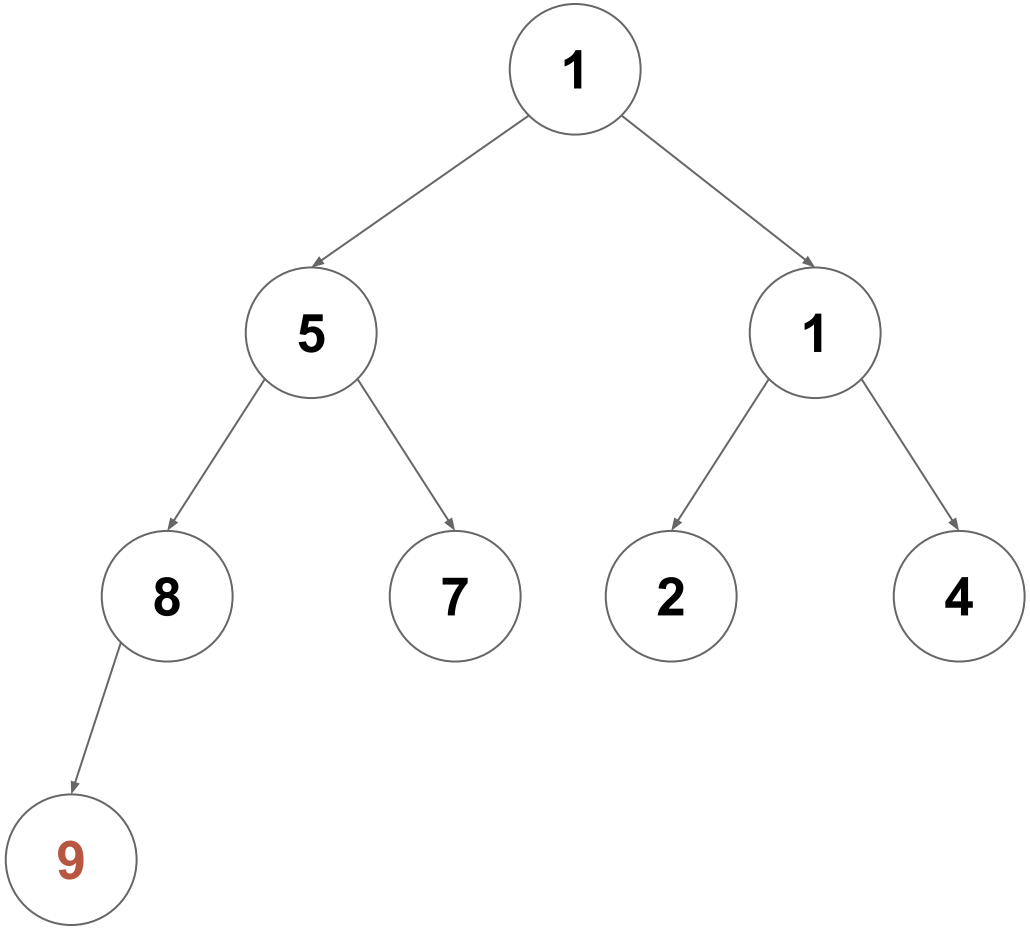
## `update`

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



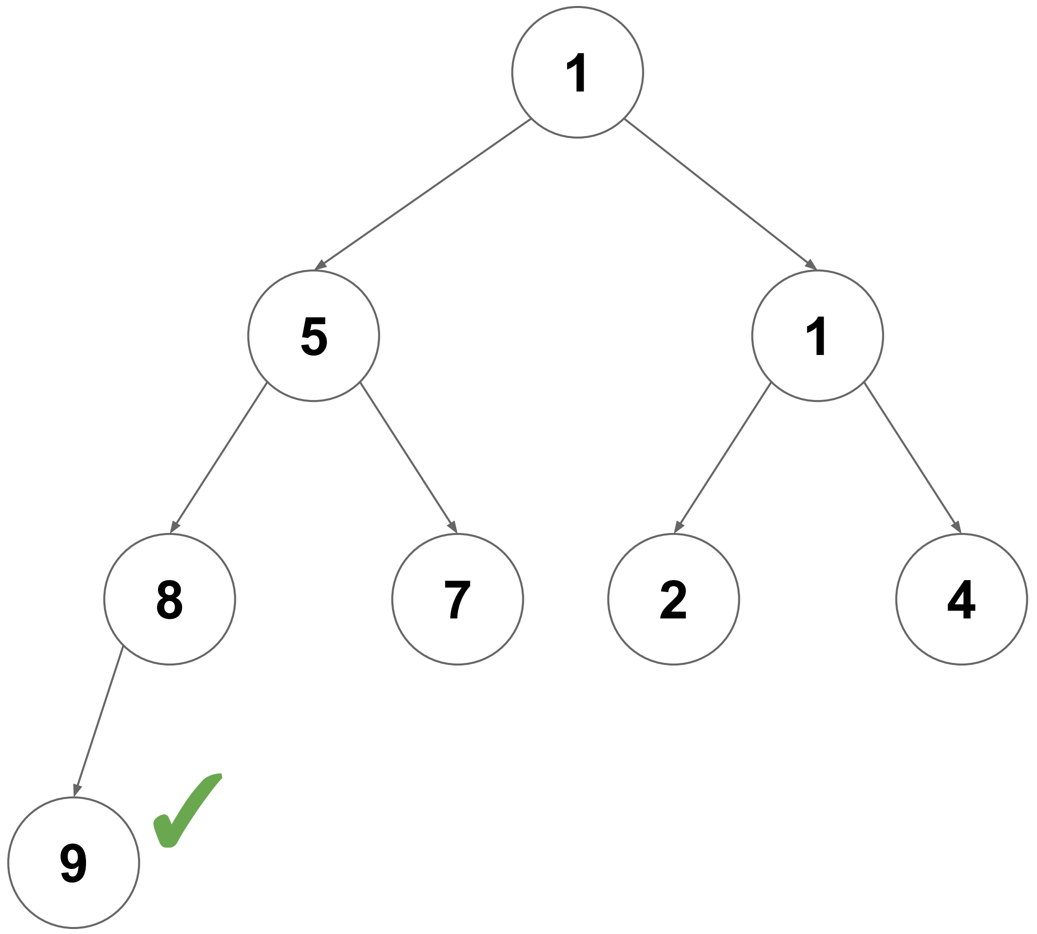
## `update`

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



## 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

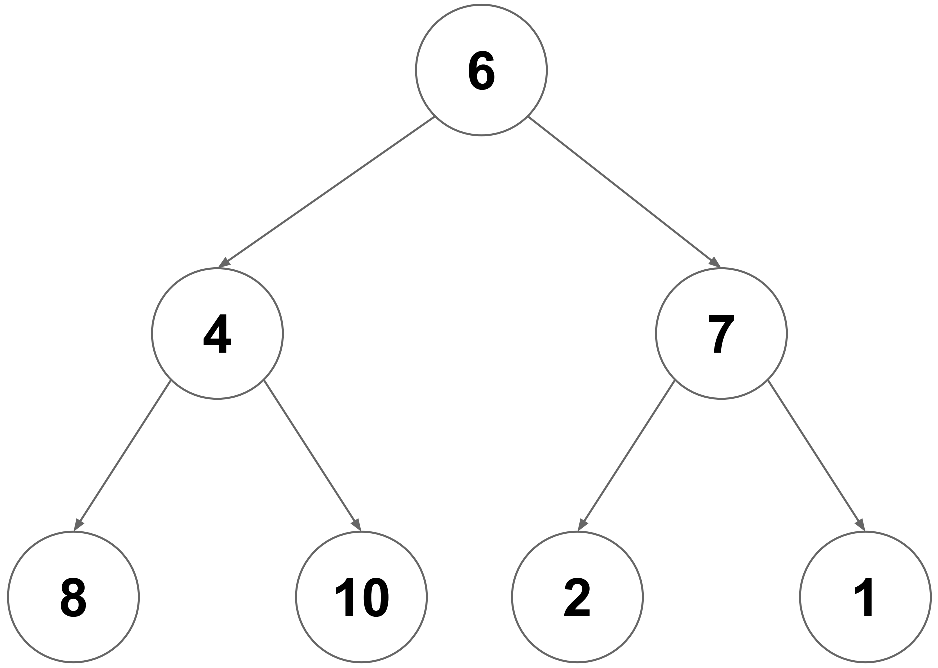
**Output:** Array re-ordered to be a heap

**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?*

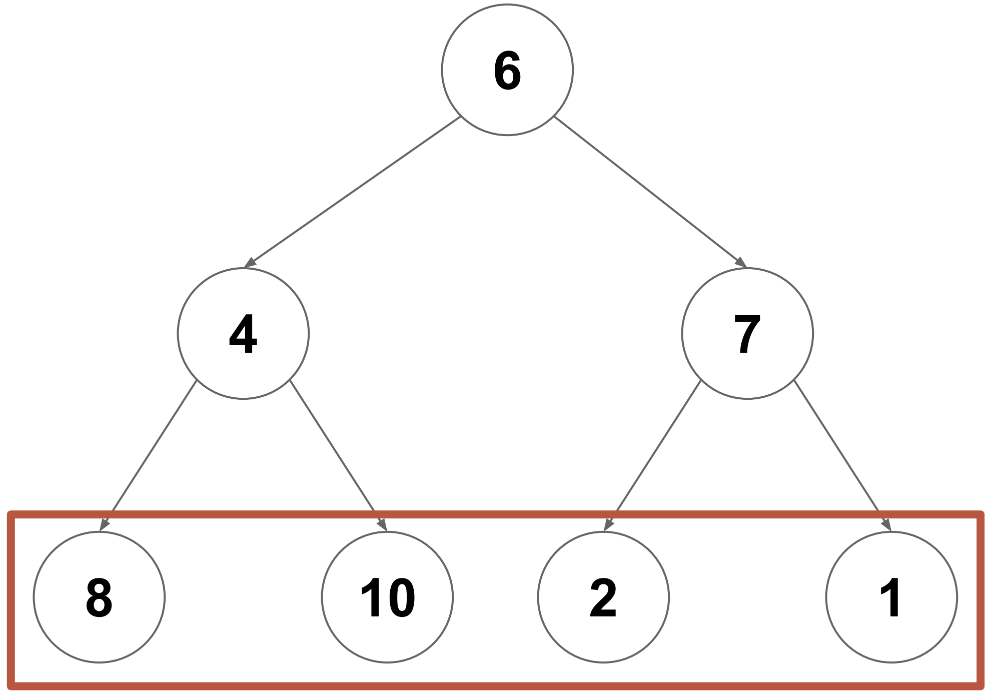
## Heapify

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



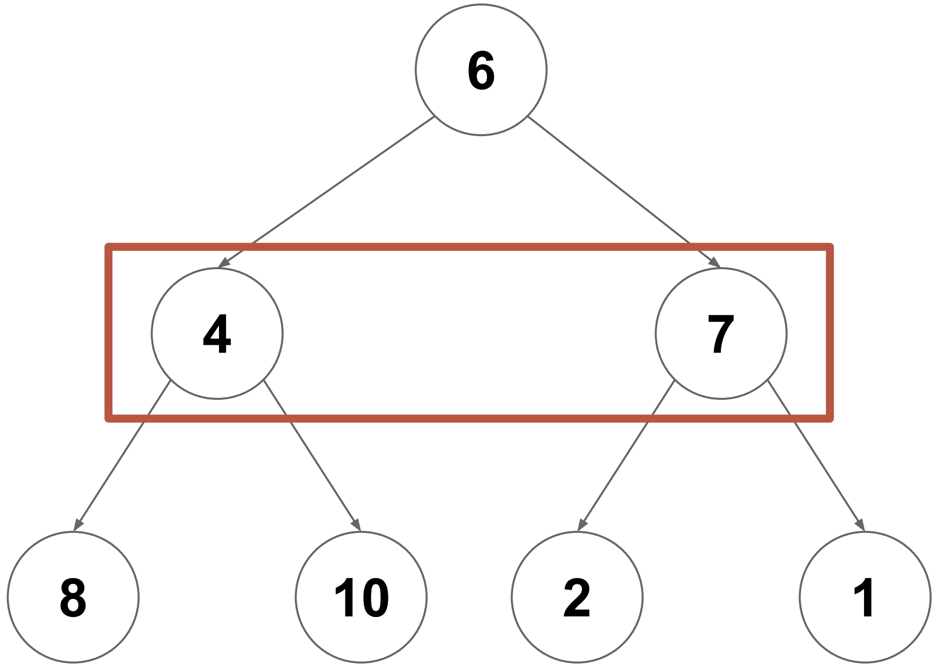
# Heapify

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



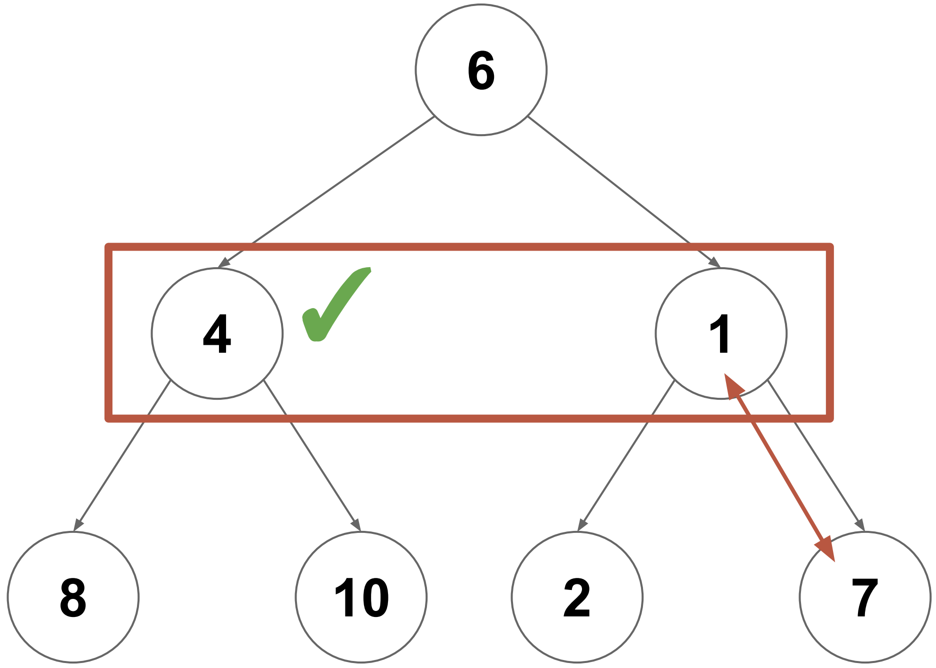
## Heapify

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



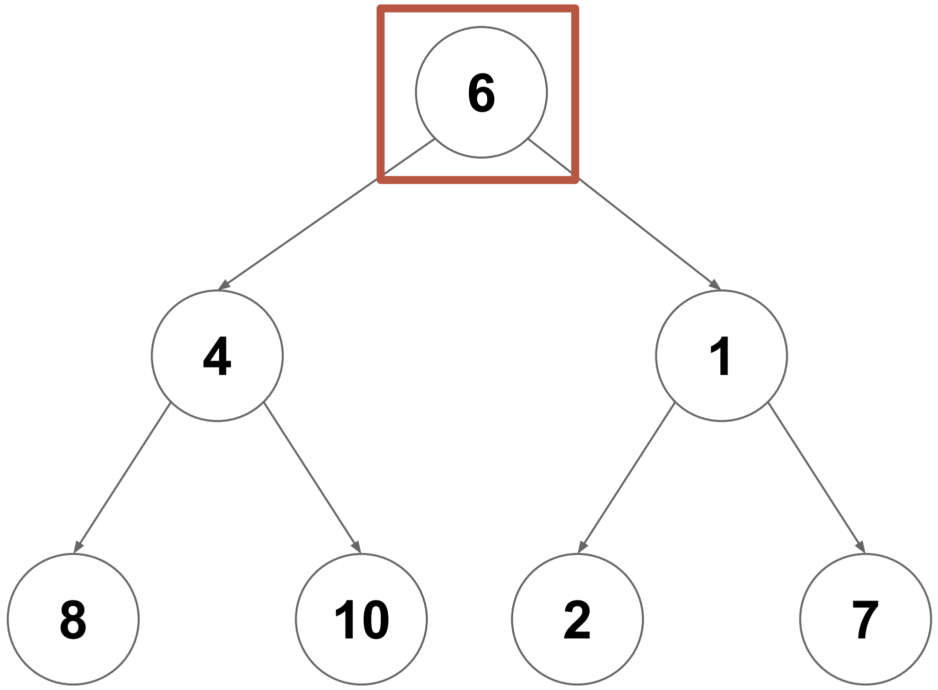
## Heapify

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



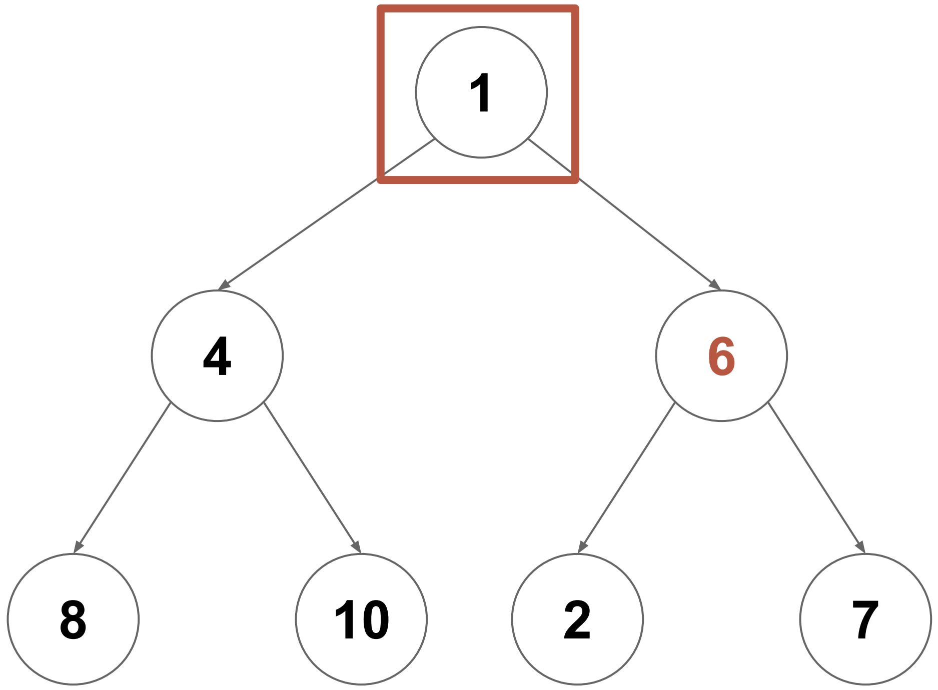
## Heapify

Continue upwards (now at most 2 swaps per node)



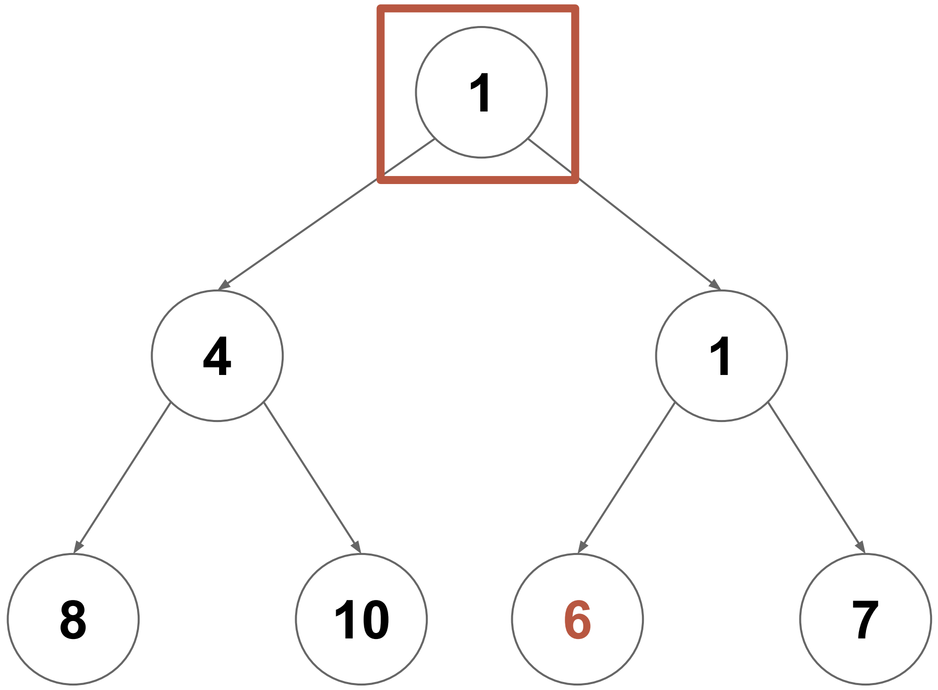
## Heapify

Continue upwards (now at most 2 swaps per node)



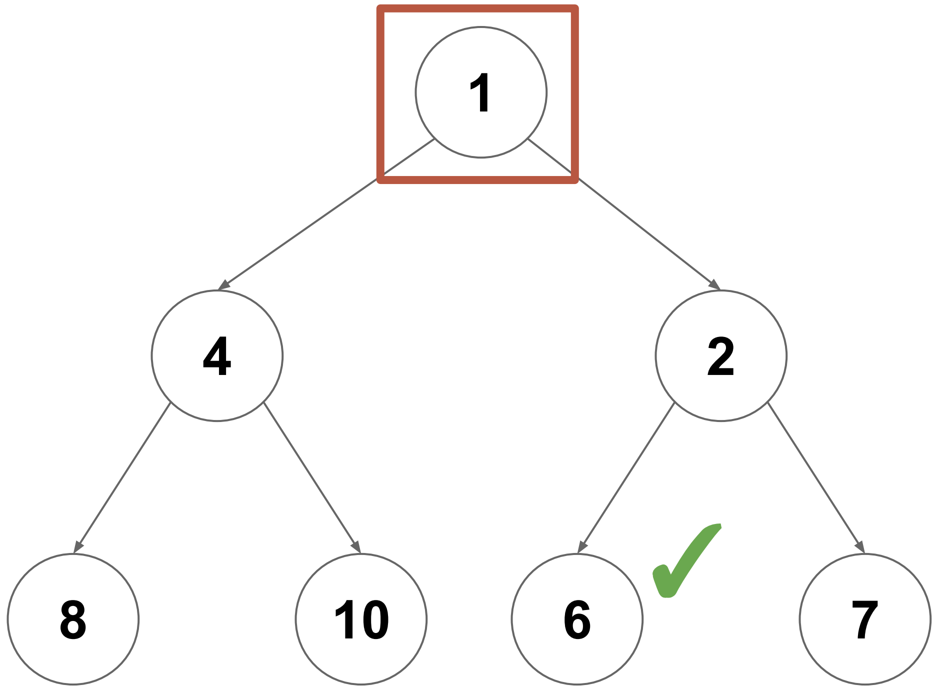
## Heapify

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)

## Heapify

Sum the number of swaps required by each level

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

## 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)

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

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

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

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