CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 24
Heaps, Sets, Bags, and Ordered Trees
Textbook Ch. 16, 18

Announcements

Priority Queues

Lazy - Fast Enqueue, Slow Dequeue

Proactive - Slow Enqueue, Fast Dequeue

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

Binary Heaps

Organize our priority queue as a directed tree

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

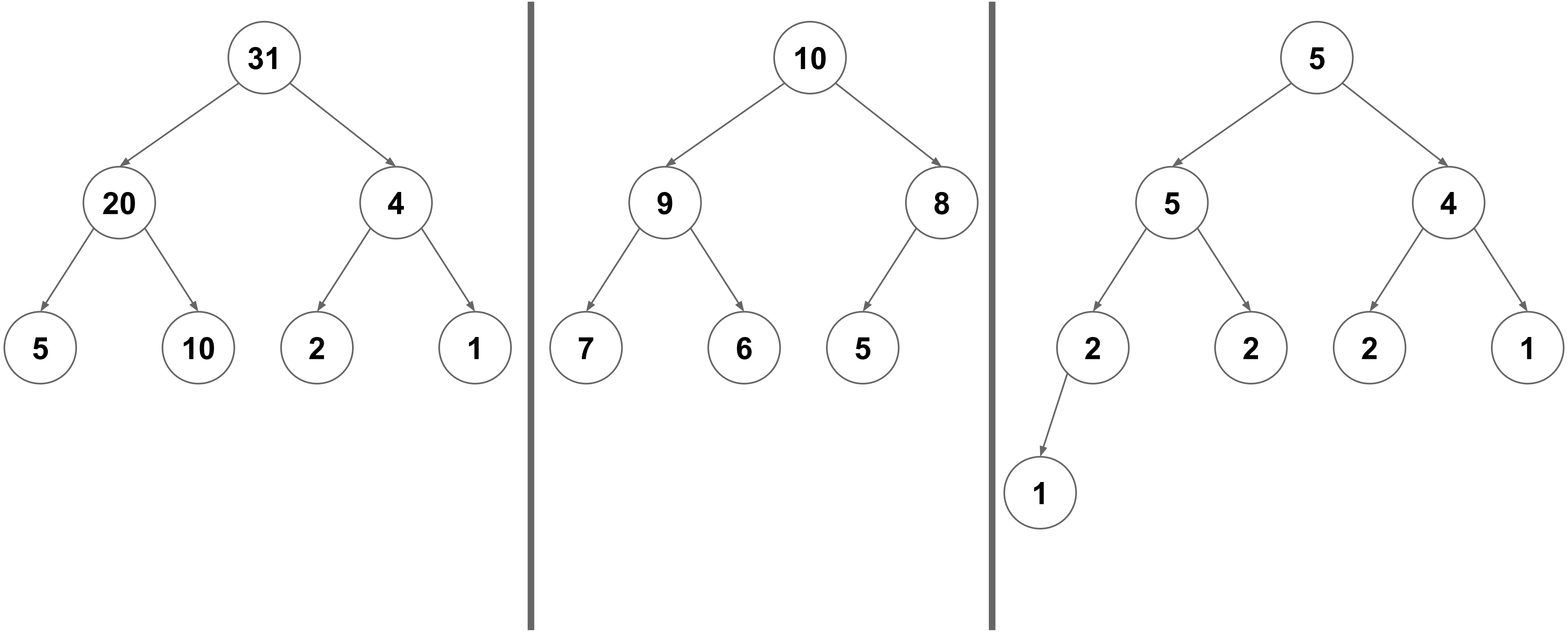
Binary: Max out-degree of 2 (easy to reason about)

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

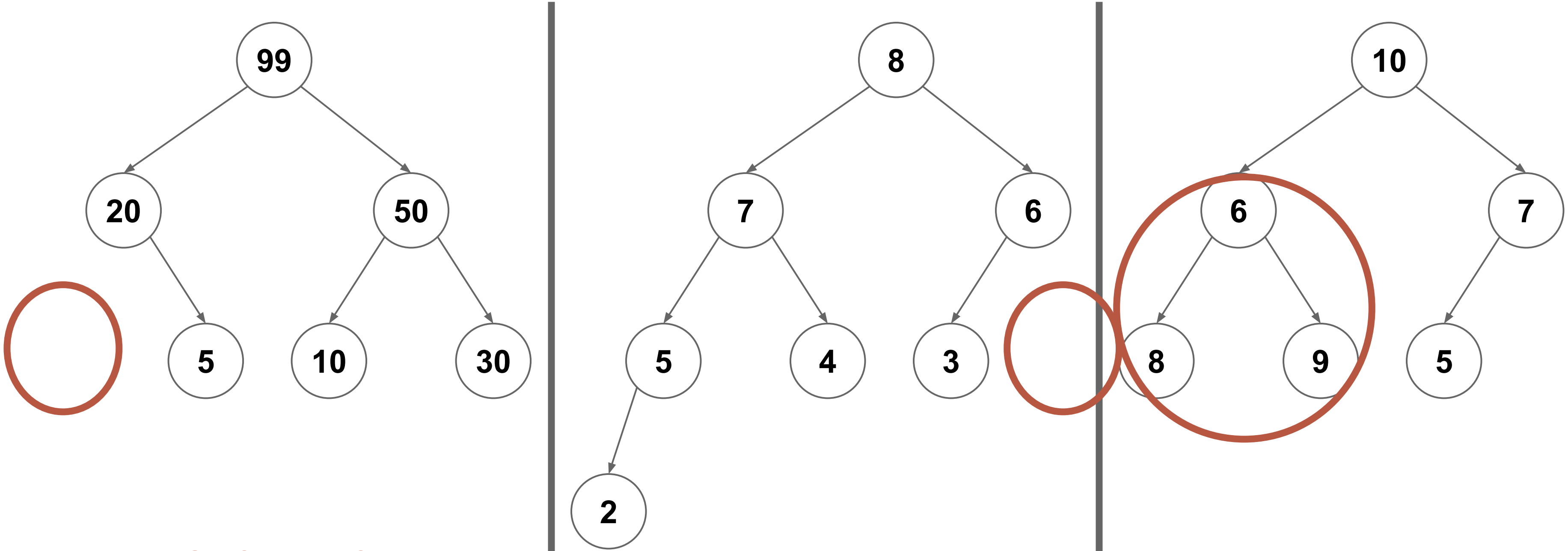
Balanced: TBD (basically, all leaves are roughly at the same level)

This makes it easy to encode into an array

Valid Max Heaps



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?*

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

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

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

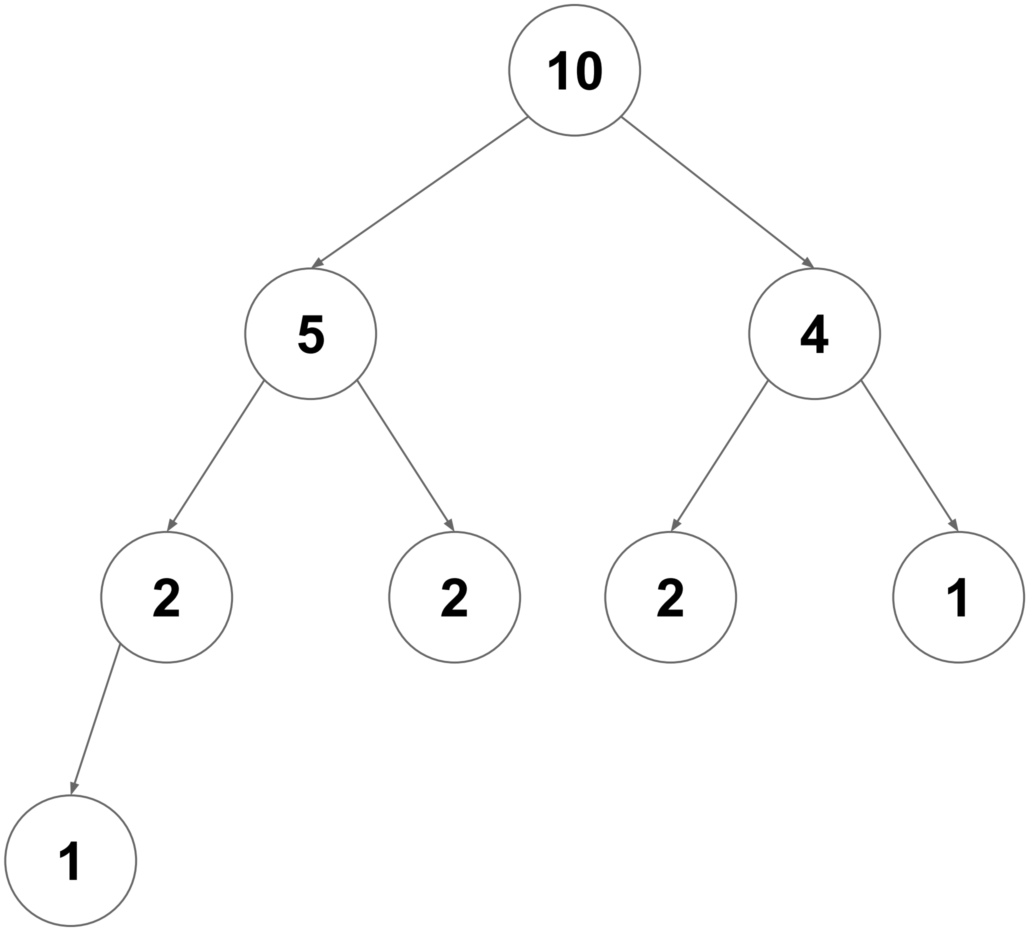
`Heap.enqueue`

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

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

Heap.enqueue

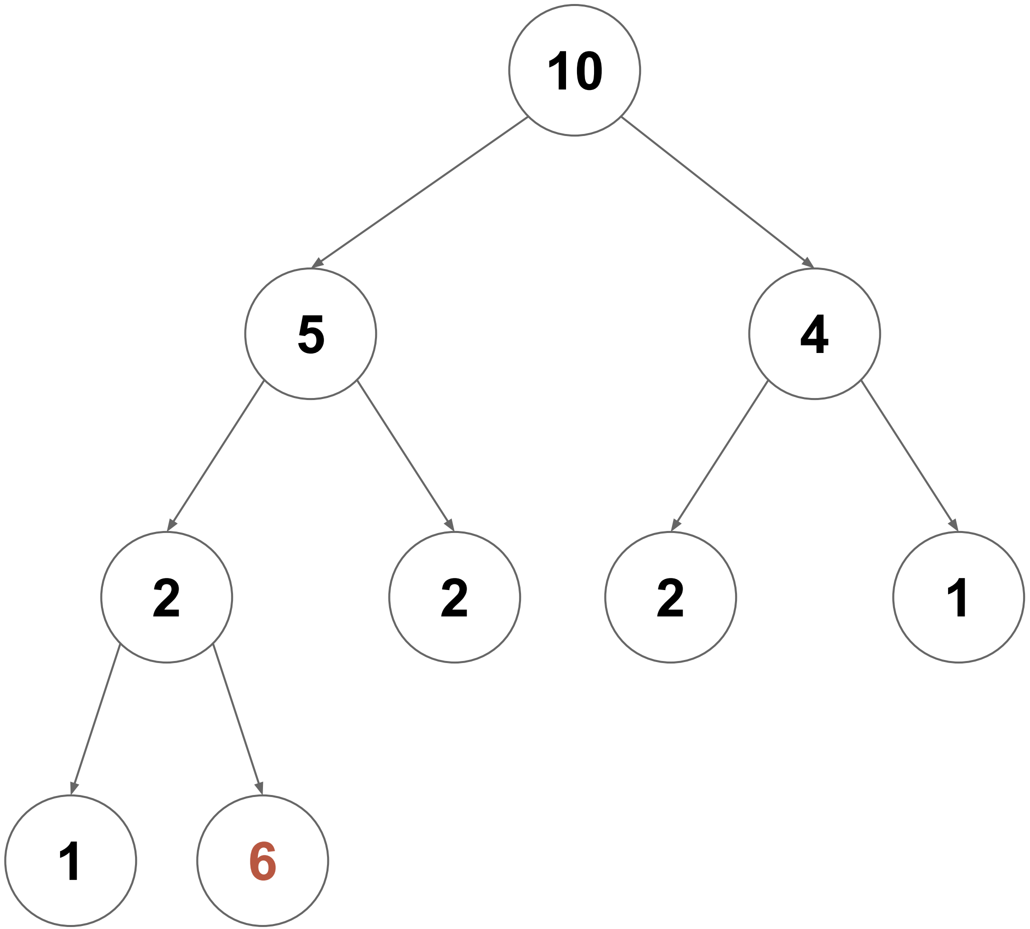
What if we enqueue 6?



Heap.enqueue

What if we enqueue 6?

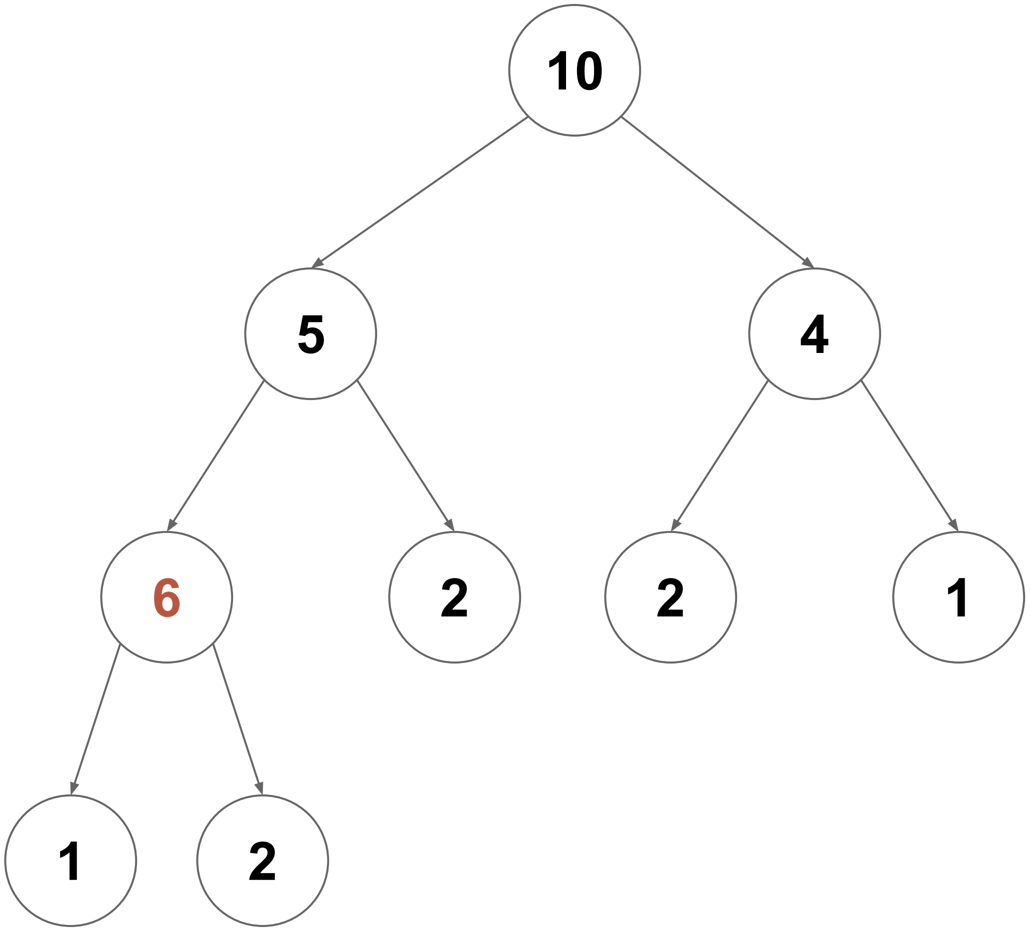
Place in the next available spot



`Heap.enqueue`

What if we enqueue 6?

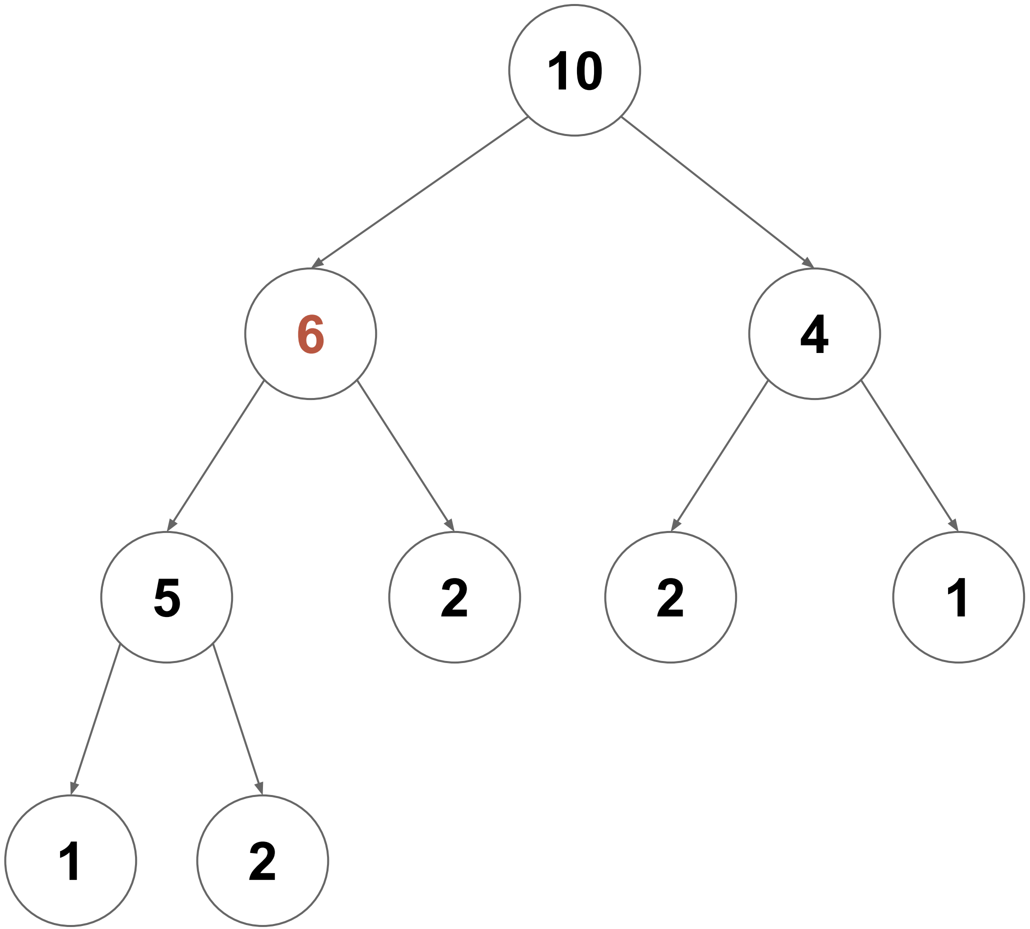
Swap with parent if it is bigger than the parent



Heap.enqueue

What if we enqueue 6?

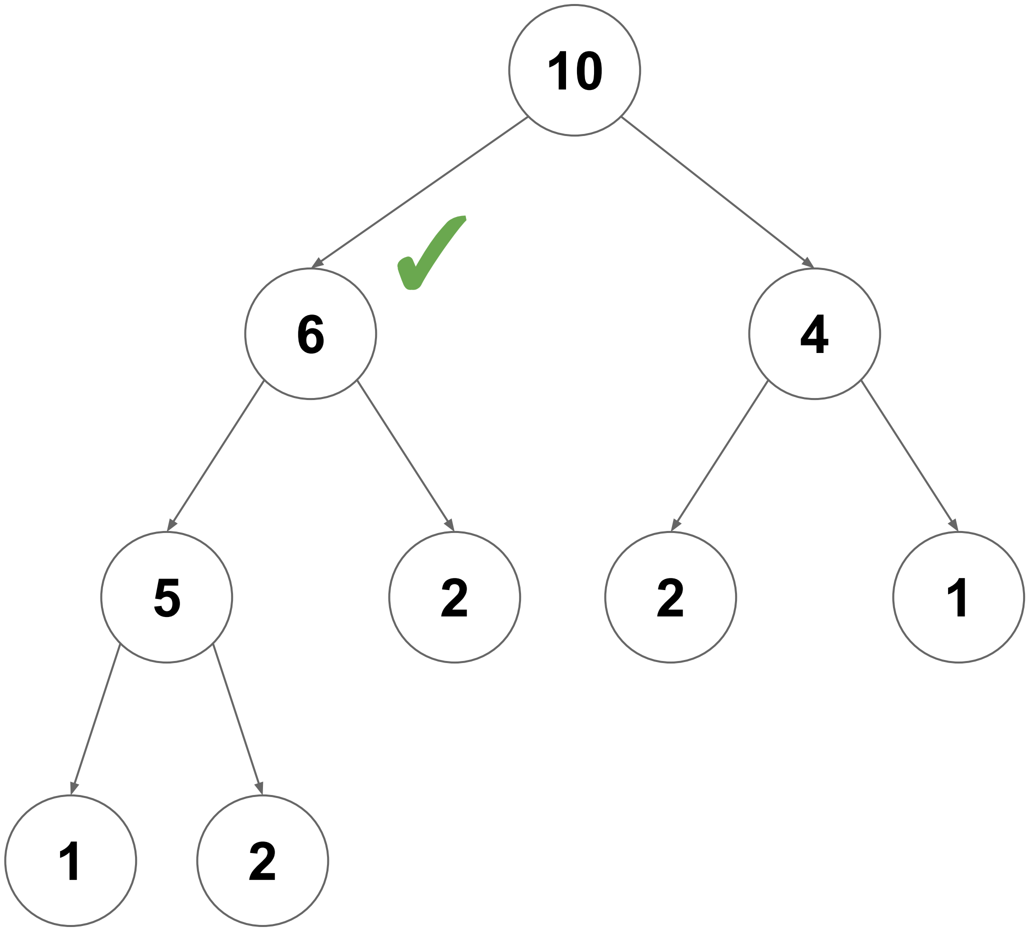
Continue swapping upwards…



Heap.enqueue

What if we enqueue 6?

Stop swapping when we are no longer bigger than our parent



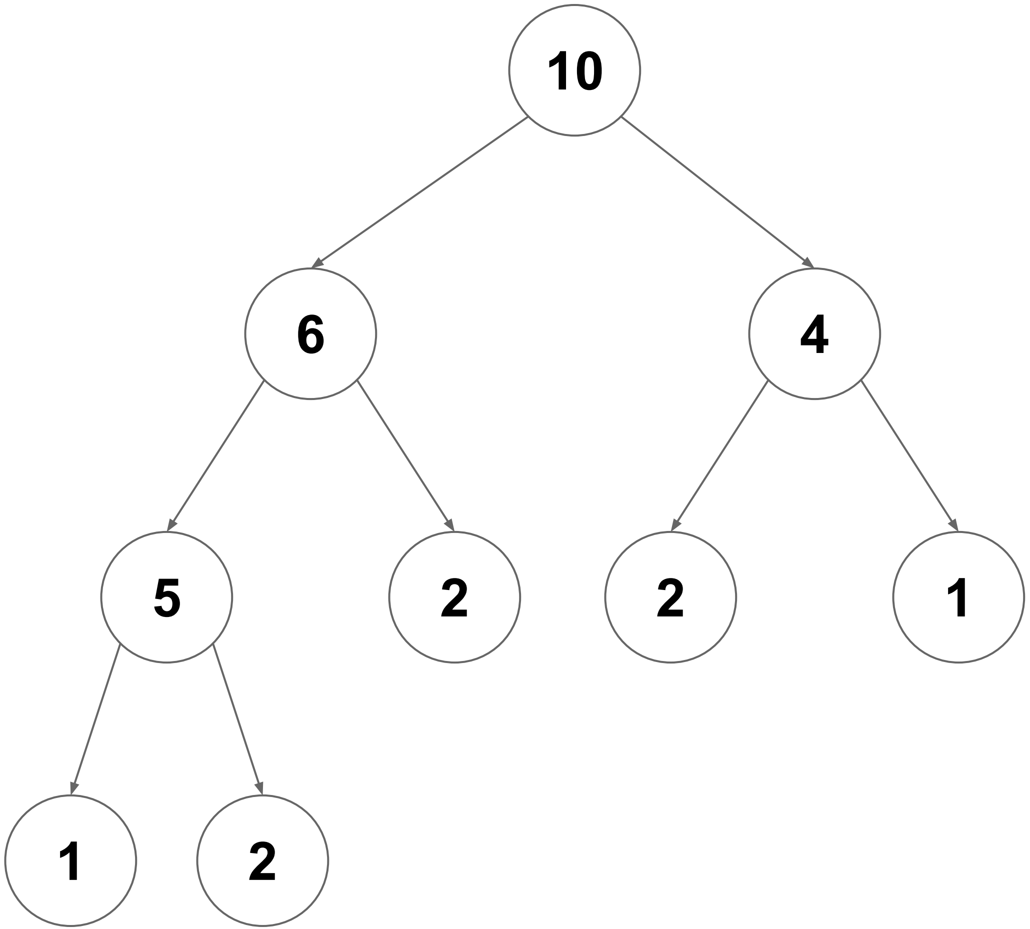
`Heap.dequeue`

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

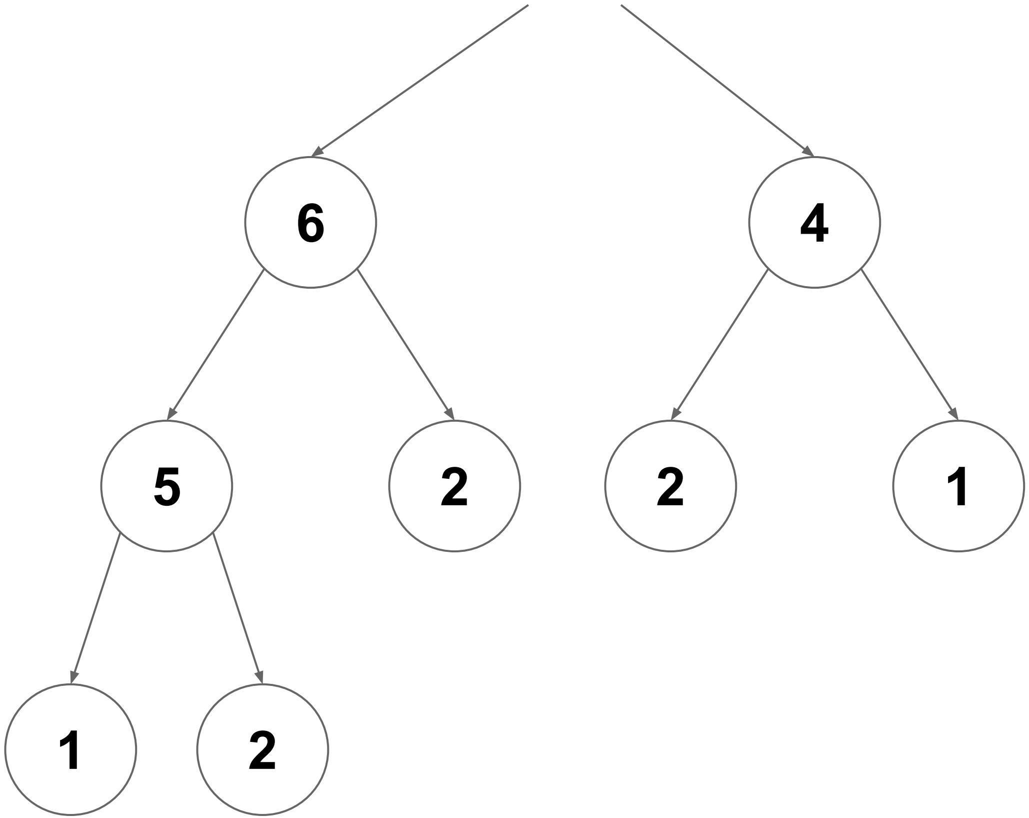
What if we call dequeue?



Heap.dequeue

What if we call dequeue?

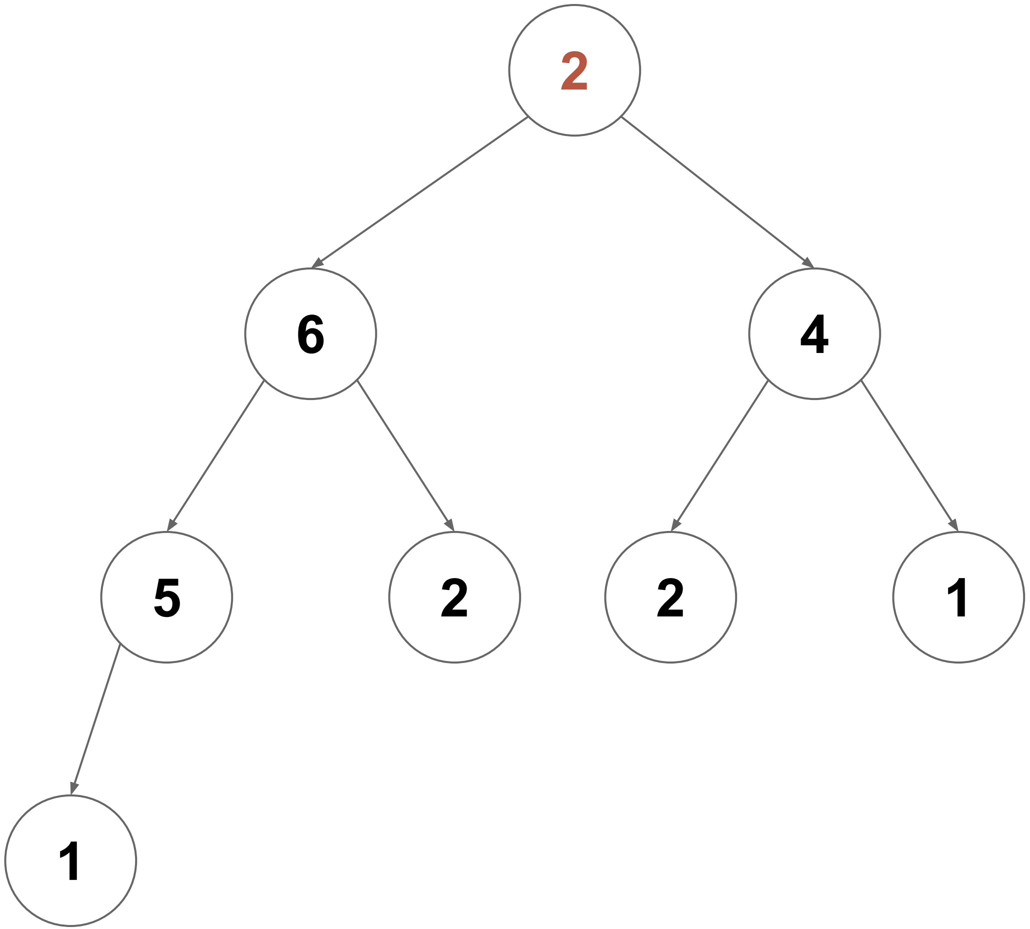
Remove and return the root



Heap.dequeue

What if we call dequeue?

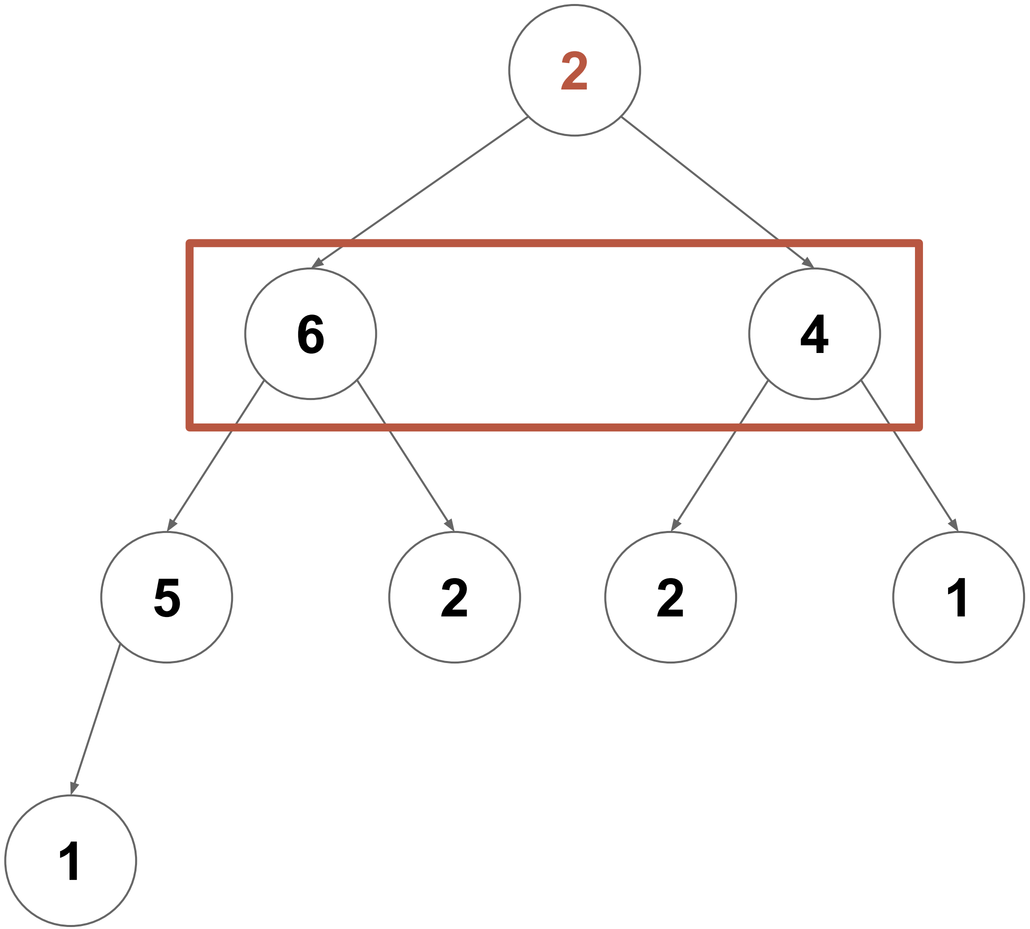
Make the last item the new root



Heap.dequeue

What if we call dequeue?

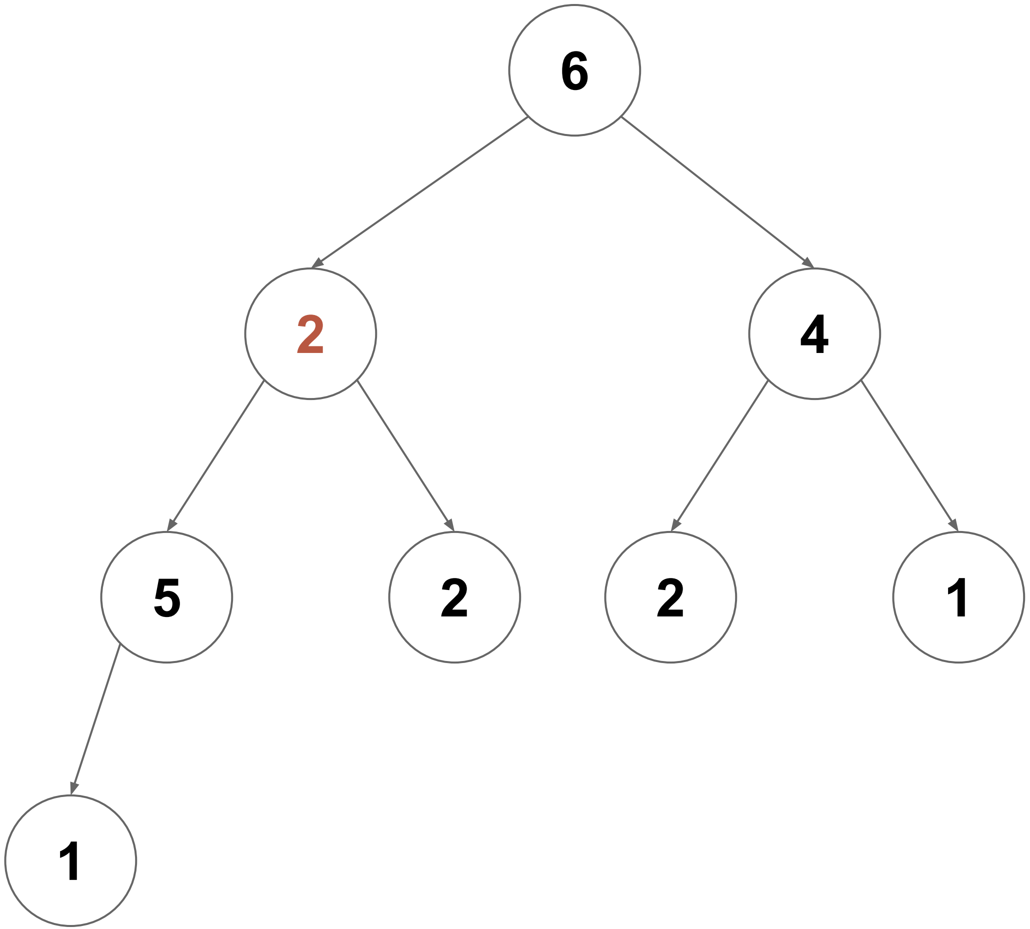
Check for our largest child



Heap.dequeue

What if we call dequeue?

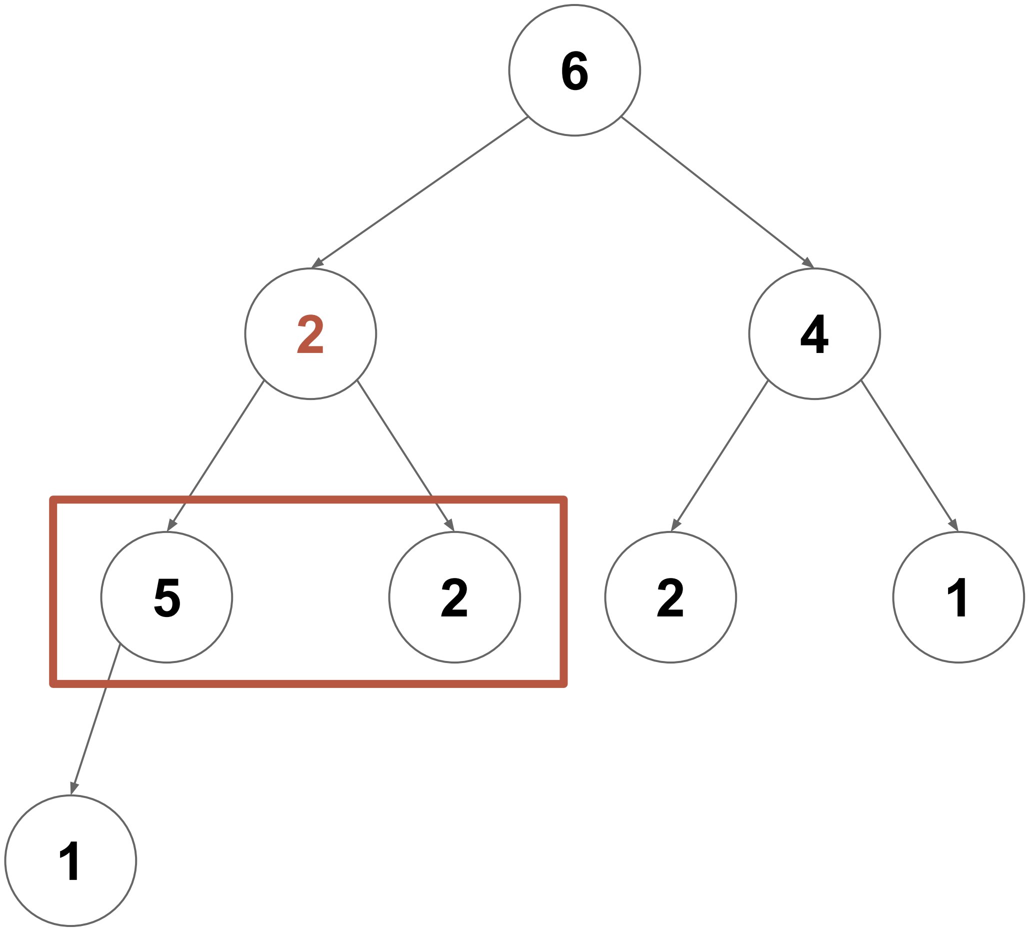
If the largest child is bigger than us, swap



Heap.dequeue

What if we call dequeue?

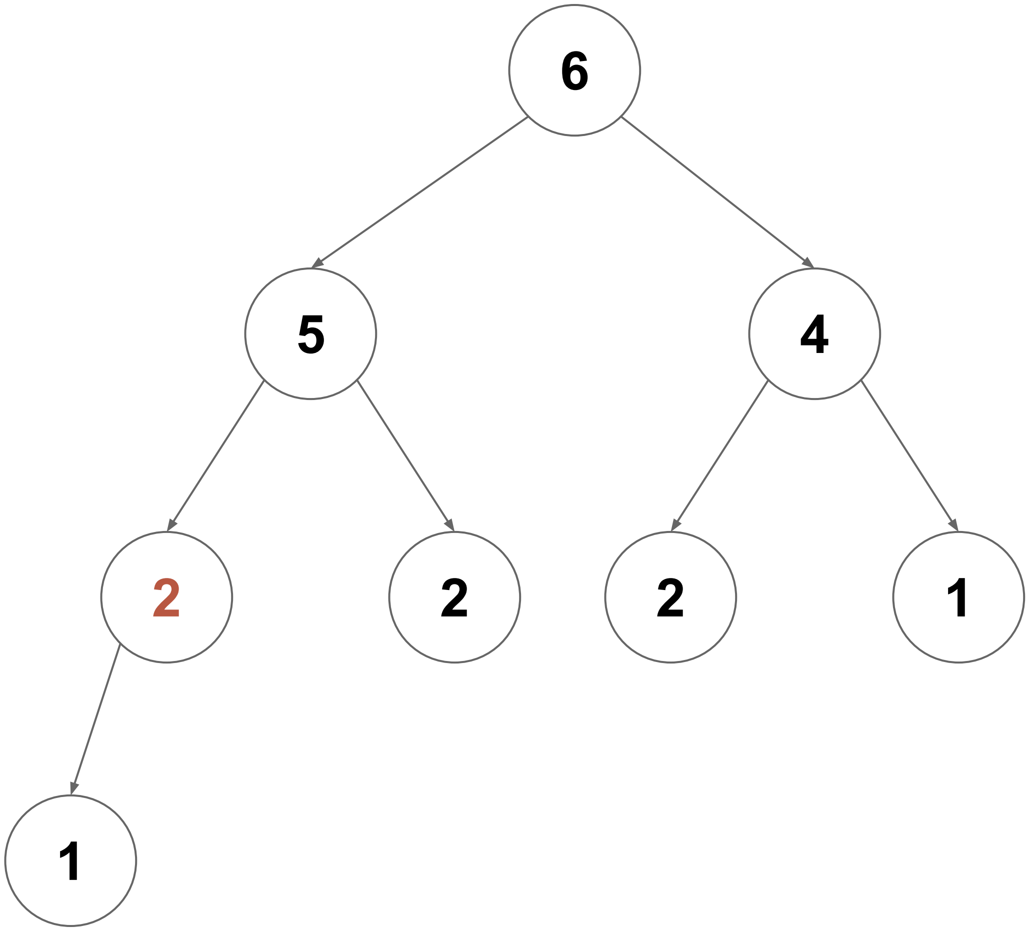
Continue swapping down the tree as necessary…



Heap.dequeue

What if we call dequeue?

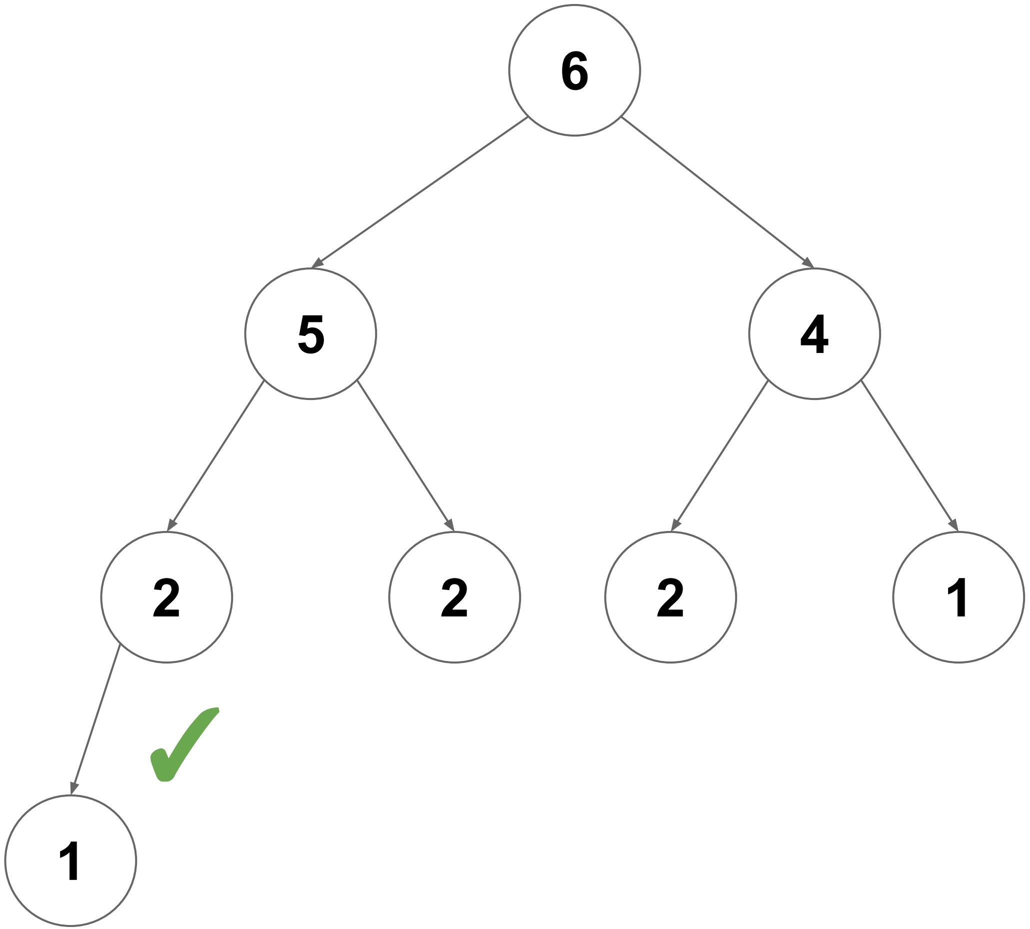
Continue swapping down the tree as necessary…



Heap.dequeue

What if we call dequeue?

Stop swapping when our children are no longer bigger



Storing heaps

Notice that:

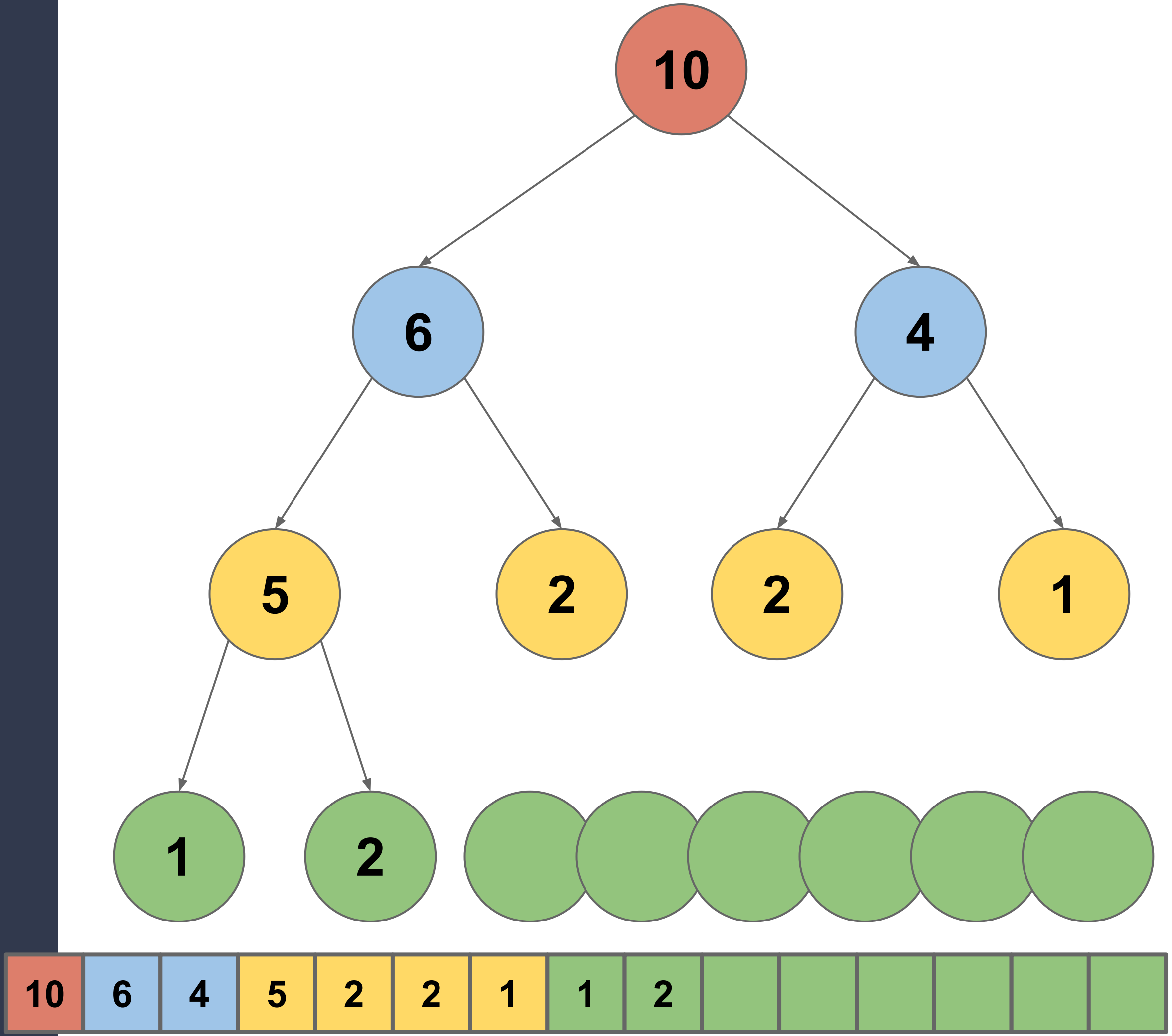
1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

How can we compactly store a heap?

Idea: Use an `ArrayBuffer`

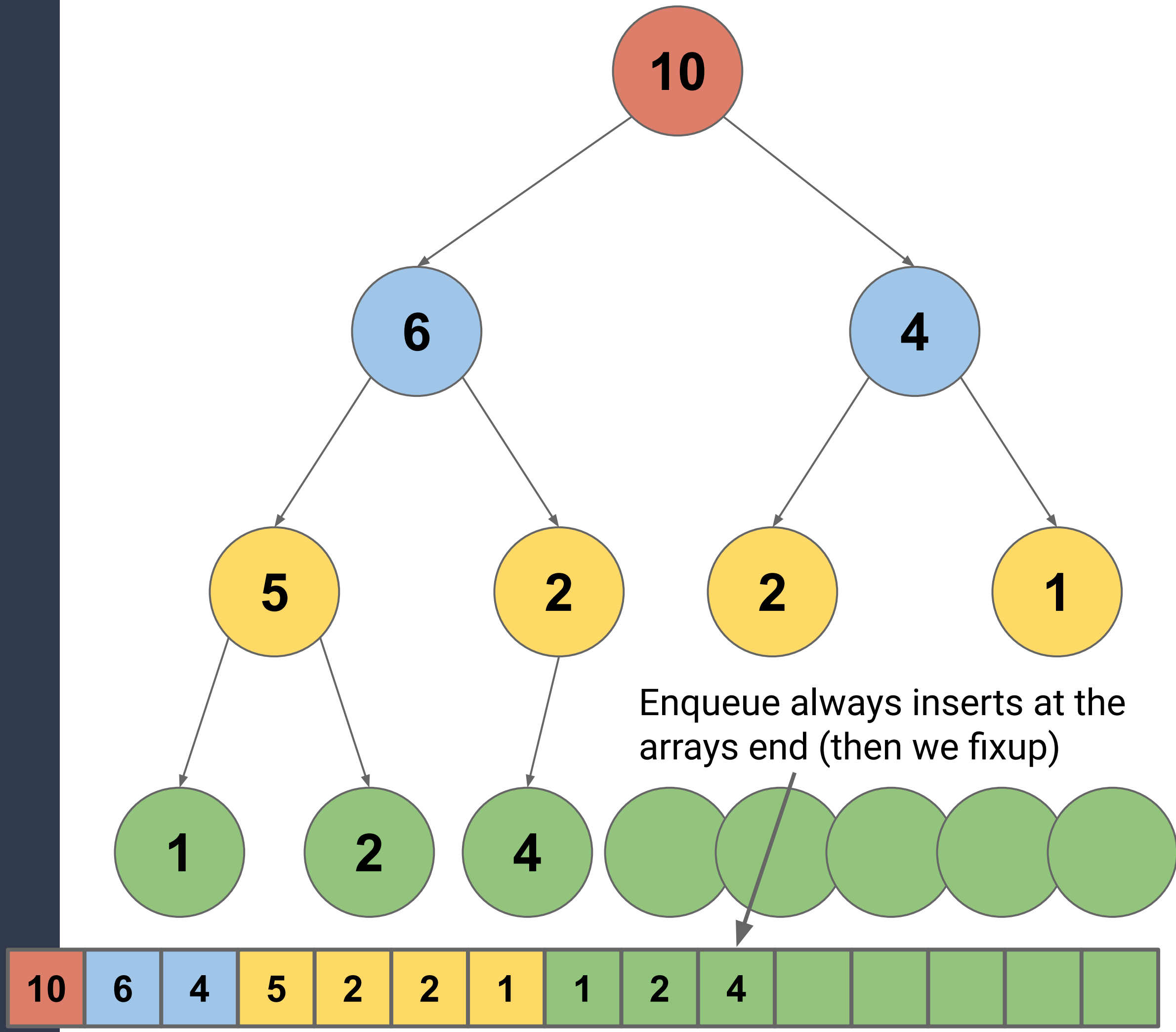
Storing Heaps

How can we store this heap in an array buffer?



Storing Heaps

How can we store this heap in an array buffer?



Runtime Analysis

`enqueue`

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

`dequeue`

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

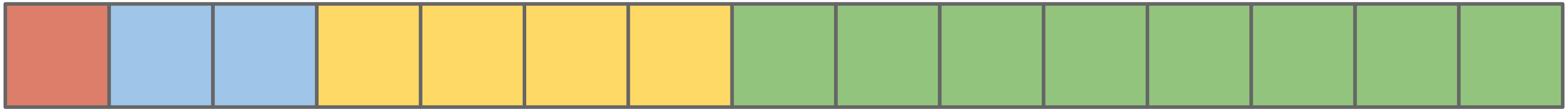
Priority Queues

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

Heap Sort

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

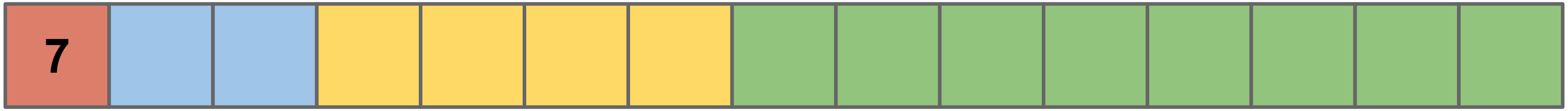
7, 4, 8, 2, 5, 3, 9



Heap Sort

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

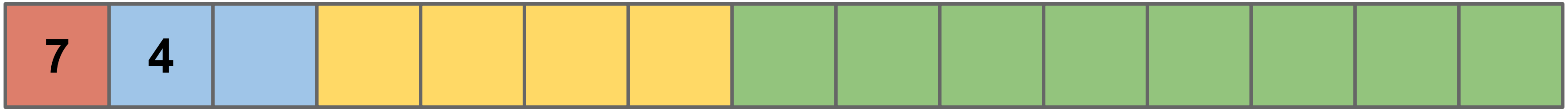
<u>7</u>, 4, 8, 2, 5, 3, 9



Heap Sort

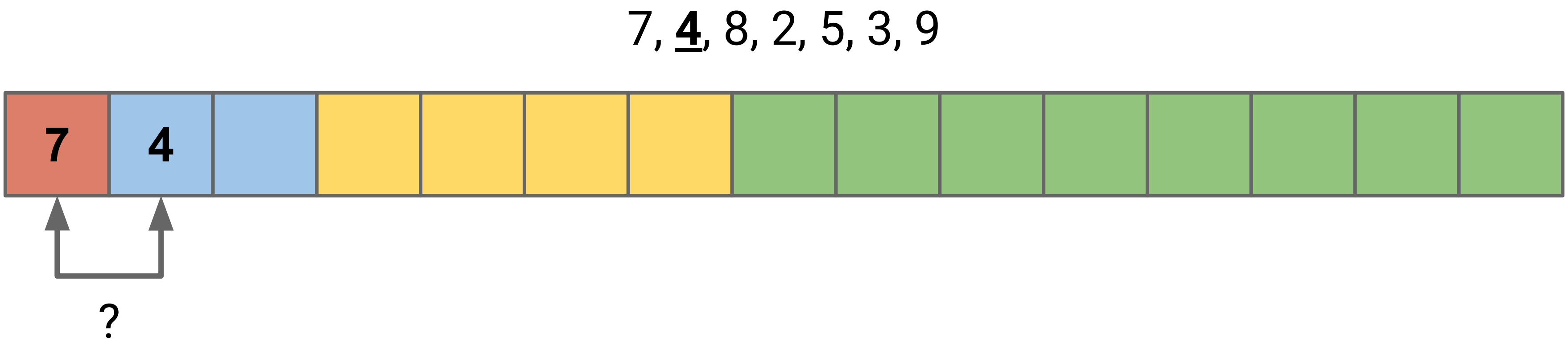
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

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



Heap Sort

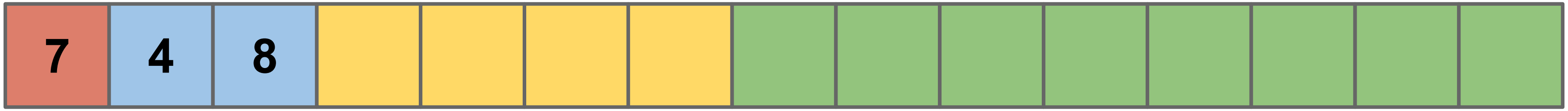
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue



Heap Sort

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

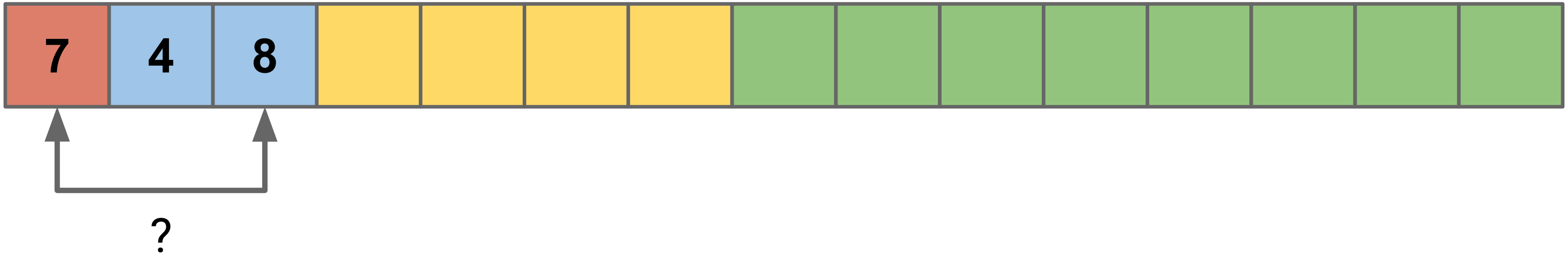
7, 4, **8**, 2, 5, 3, 9



Heap Sort

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

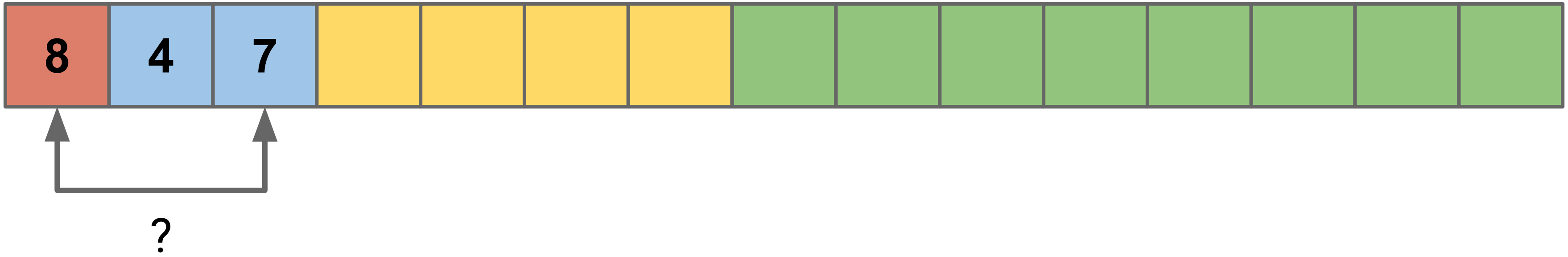
7, 4, **8**, 2, 5, 3, 9



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

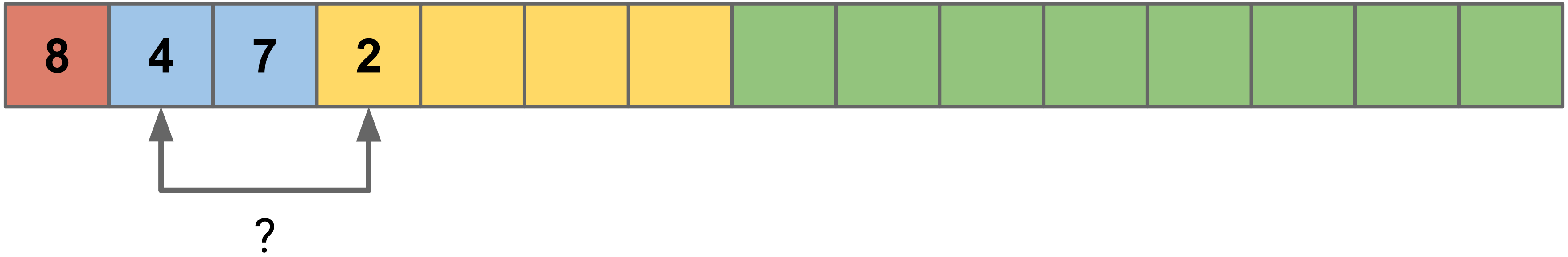
7, 4, 8, **2**, 5, 3, 9

8	4	7	2											

Heap Sort

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

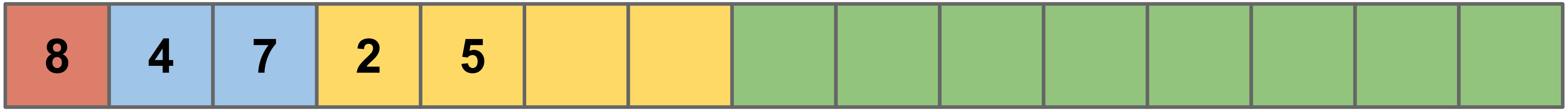
7, 4, 8, **2**, 5, 3, 9



Heap Sort

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

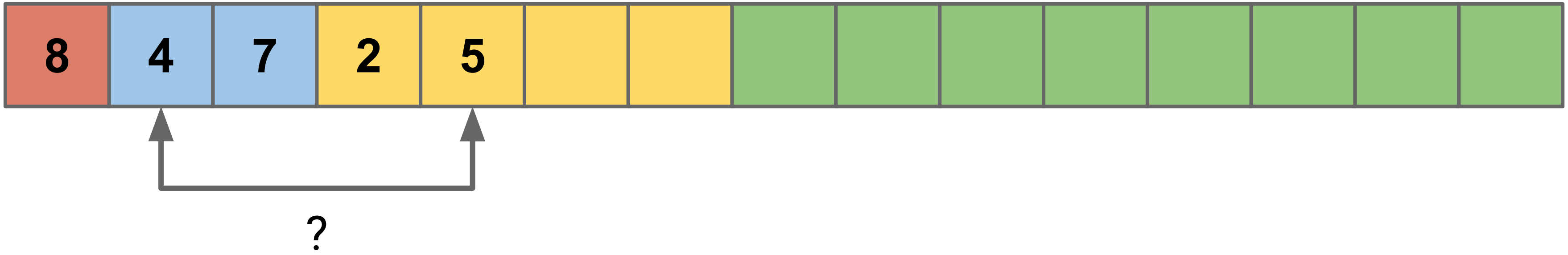
7, 4, 8, 2, <u>**5**</u>, 3, 9



Heap Sort

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

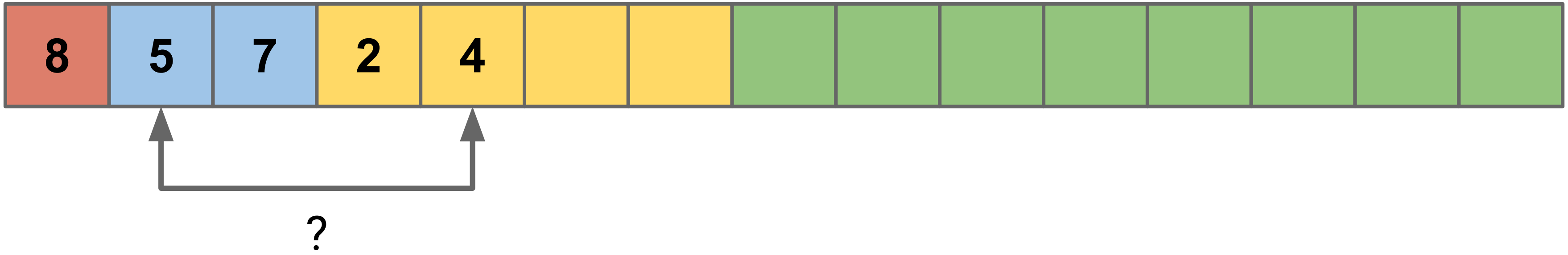
7, 4, 8, 2, **5**, 3, 9



Heap Sort

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

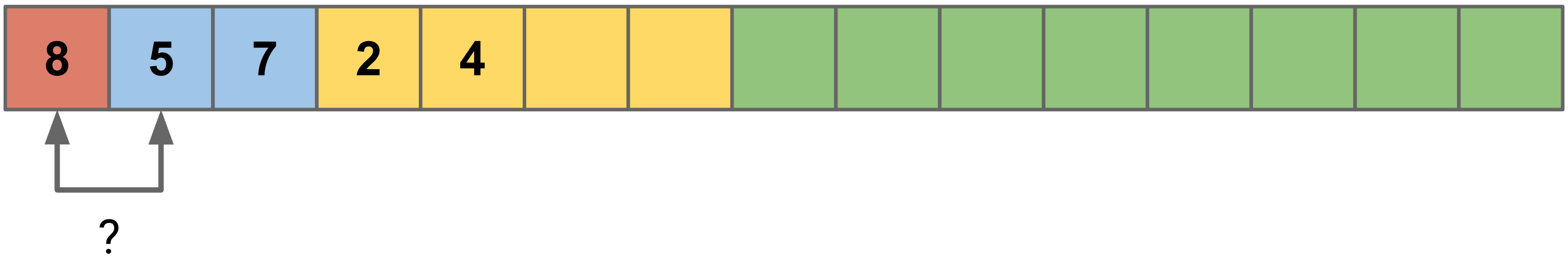
7, 4, 8, 2, **5**, 3, 9



Heap Sort

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

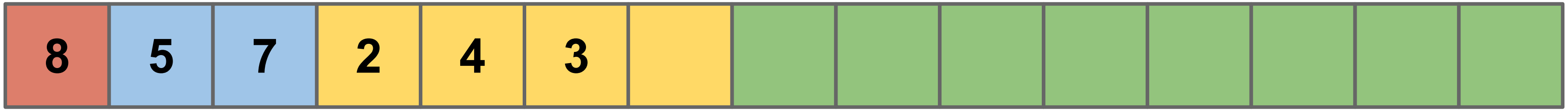
7, 4, 8, 2, **5**, 3, 9



Heap Sort

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

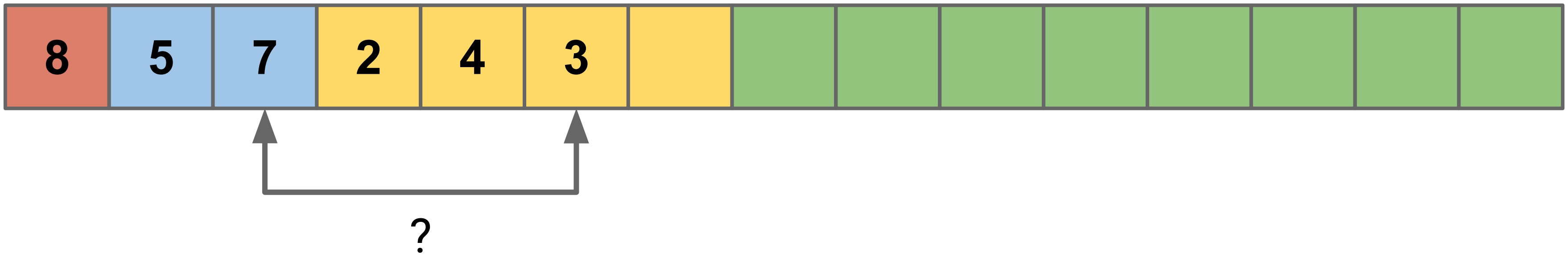
7, 4, 8, 2, 5, **3**, 9



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

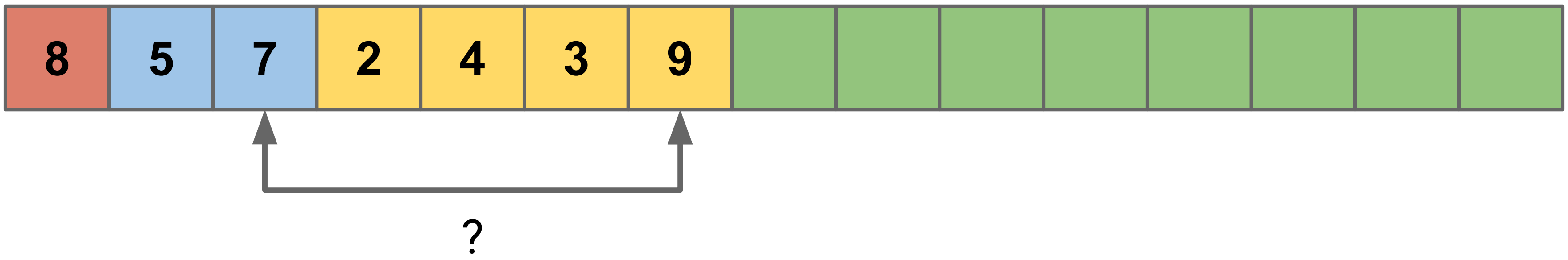
7, 4, 8, 2, 5, 3, **9**

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

Heap Sort

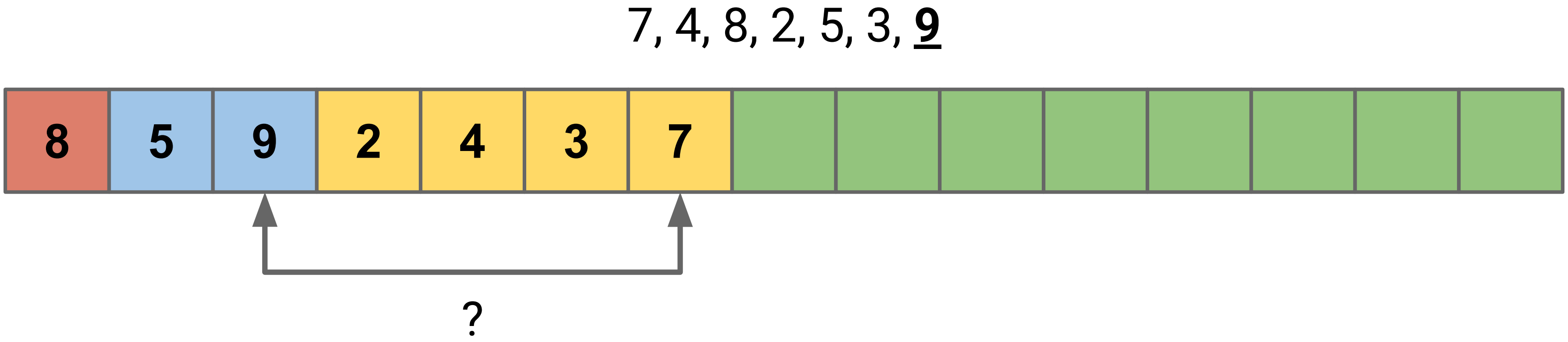
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue

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



Heap Sort

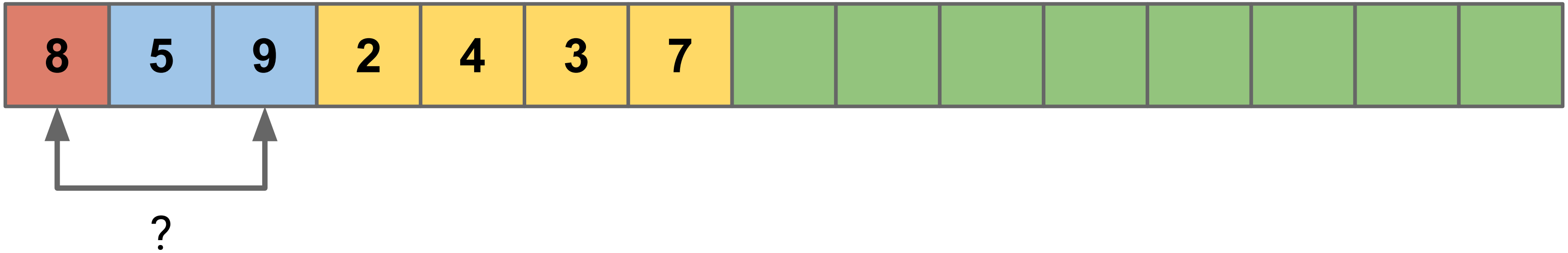
1. Insert items into heap
2. Reconstruct sequence (in reverse order) with dequeue



Heap Sort

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

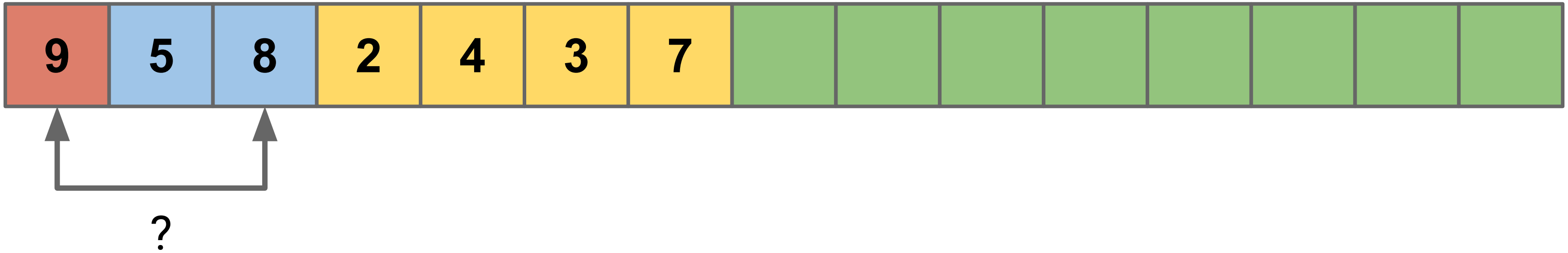
7, 4, 8, 2, 5, 3, **9**



Heap Sort

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

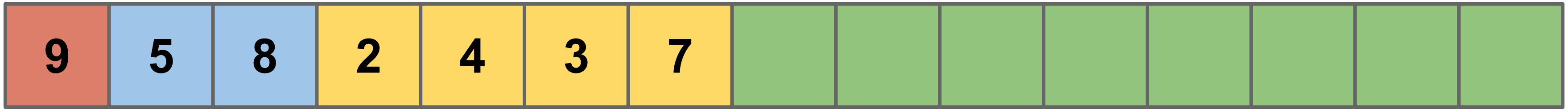
7, 4, 8, 2, 5, 3, **9**



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

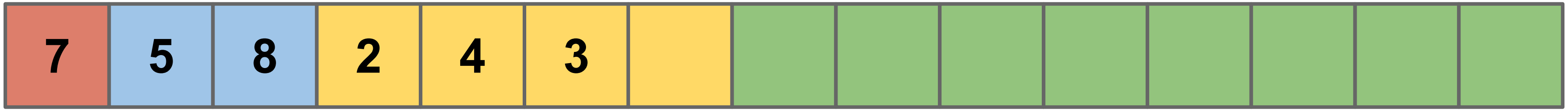
	5	**8**	**2**	**4**	**3**	**7**								

9

Heap Sort

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

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

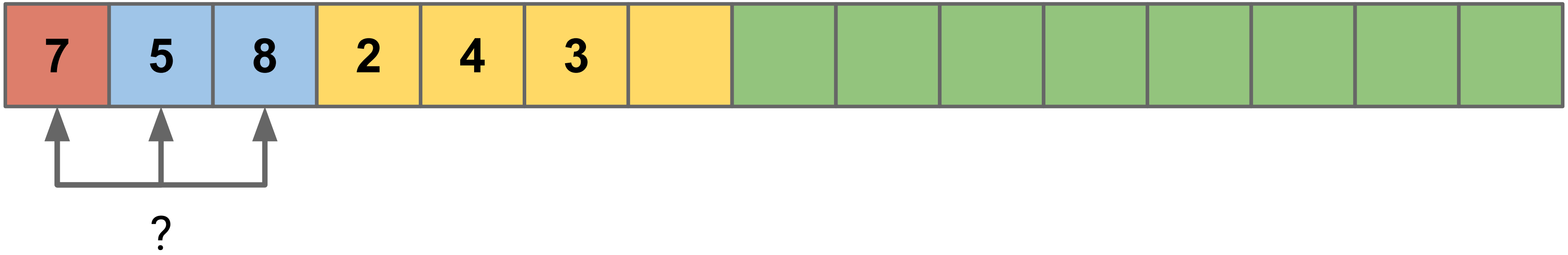


9

Heap Sort

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

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

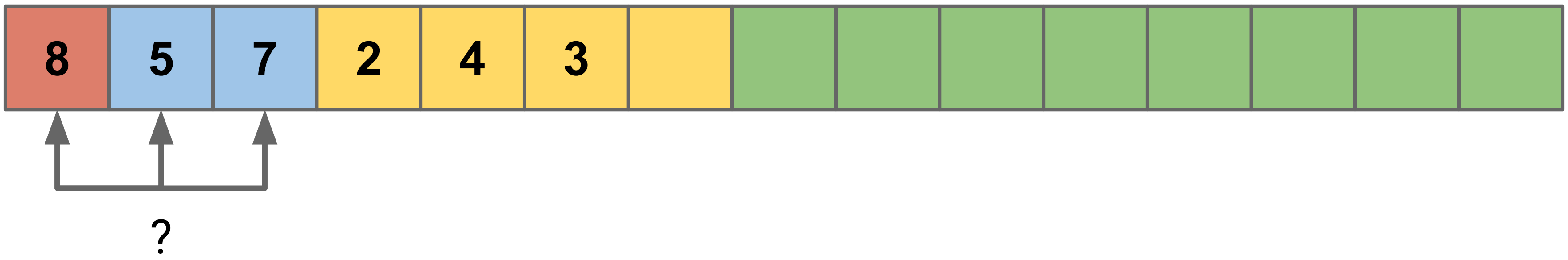


9

Heap Sort

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

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

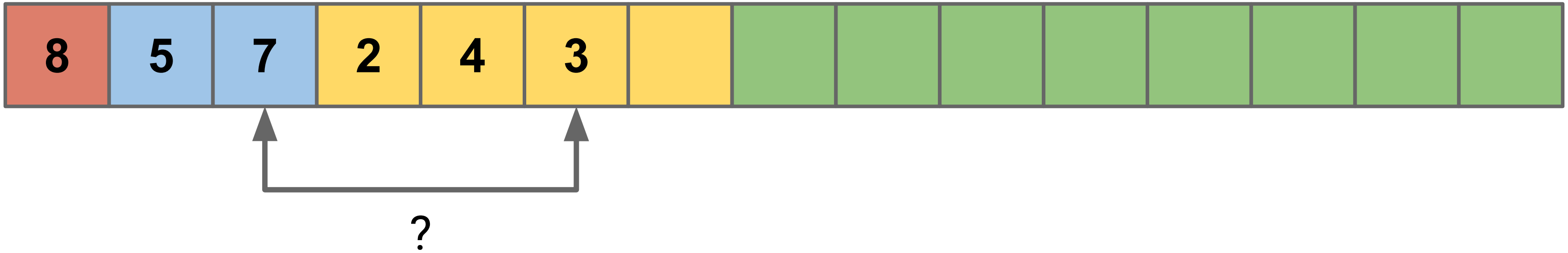


9

Heap Sort

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

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

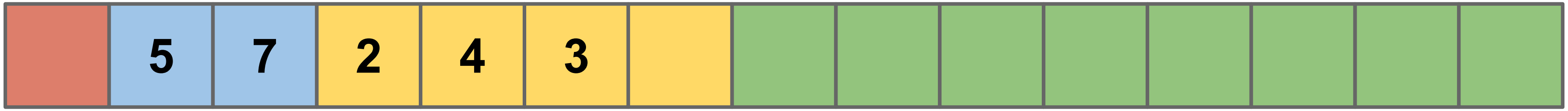


9

Heap Sort

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

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

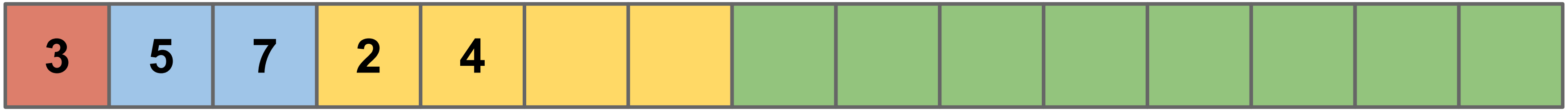


8, 9

Heap Sort

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

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

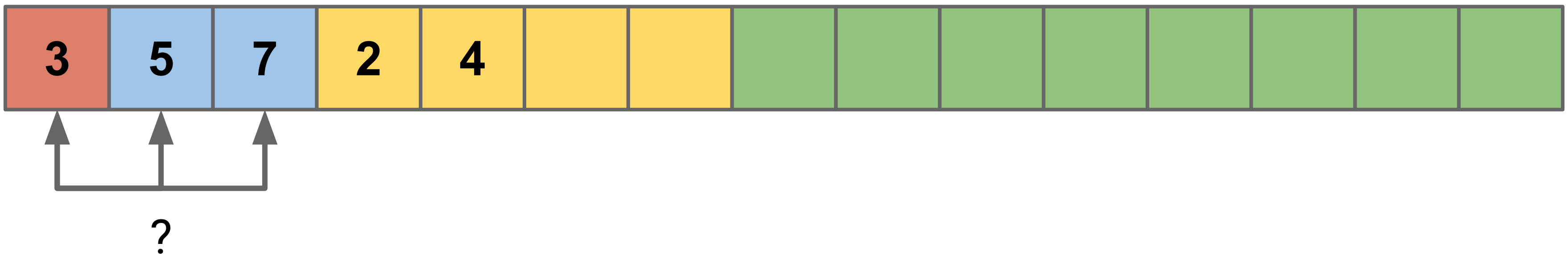


8, 9

Heap Sort

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

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

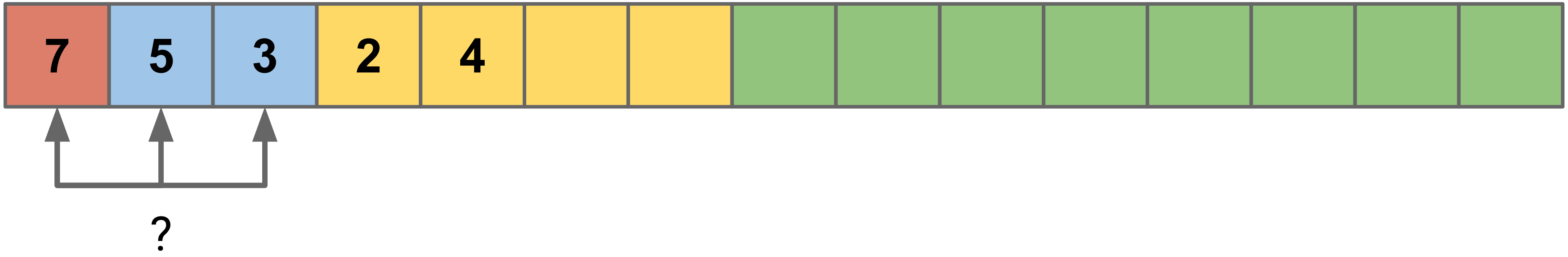


8, 9

Heap Sort

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

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

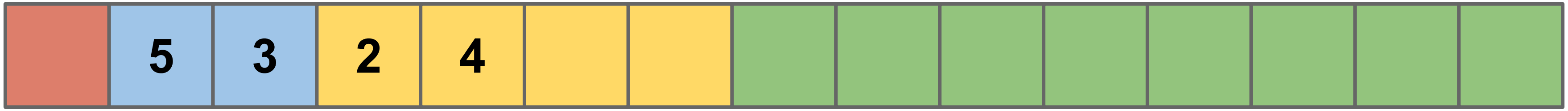


8, 9

Heap Sort

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

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

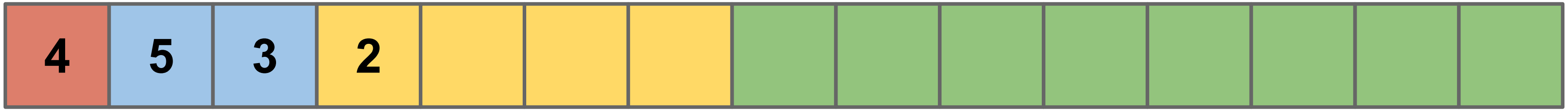


7, 8, 9

Heap Sort

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

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

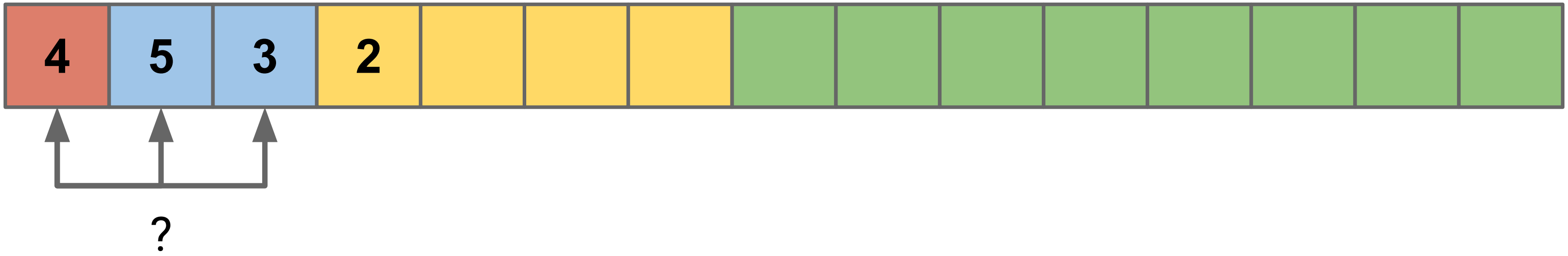


7, 8, 9

Heap Sort

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

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

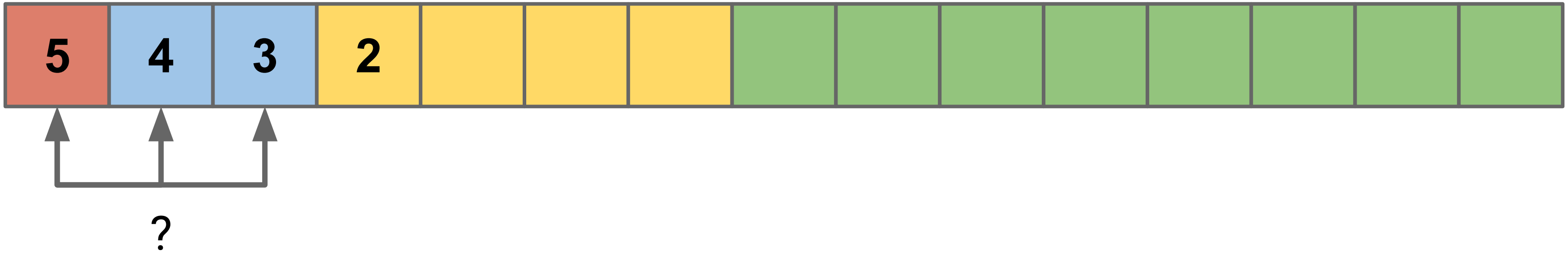


7, 8, 9

Heap Sort

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

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

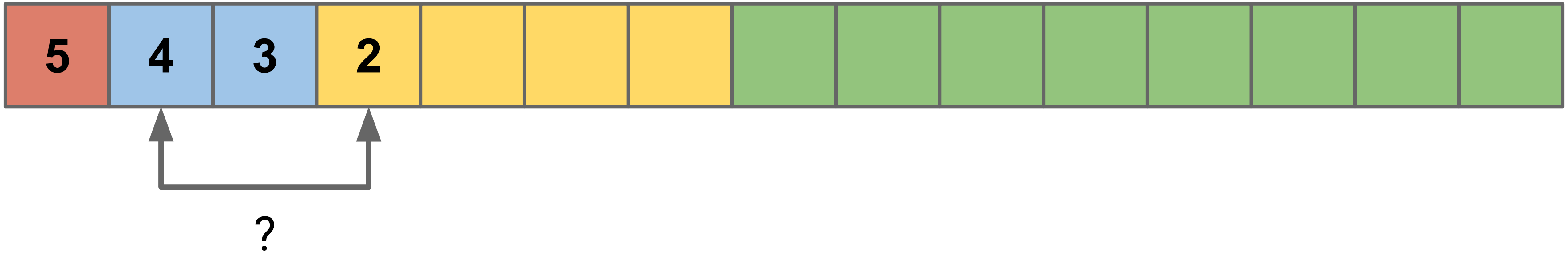


7, 8, 9

Heap Sort

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

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

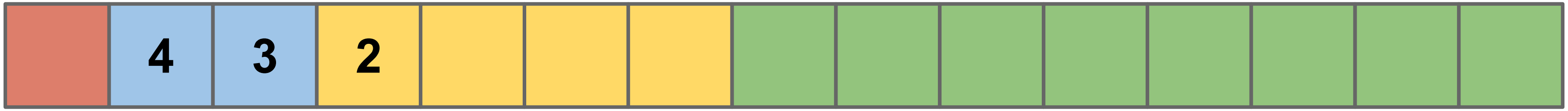


7, 8, 9

Heap Sort

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

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

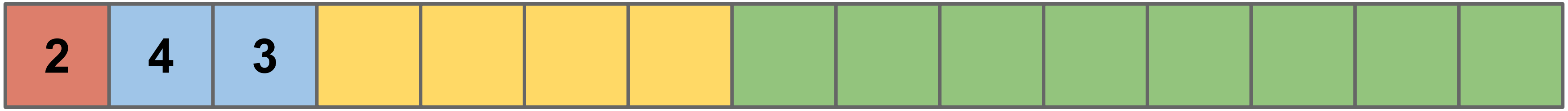


5, 7, 8, 9

Heap Sort

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

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

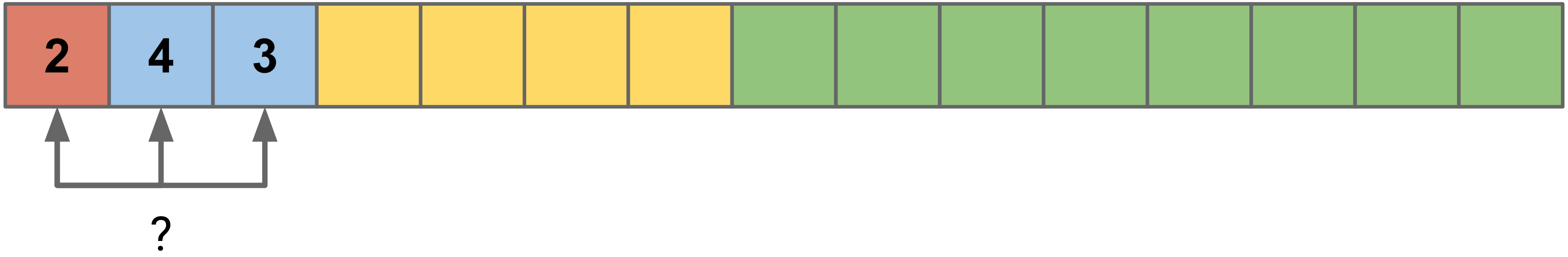


5, 7, 8, 9

Heap Sort

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

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

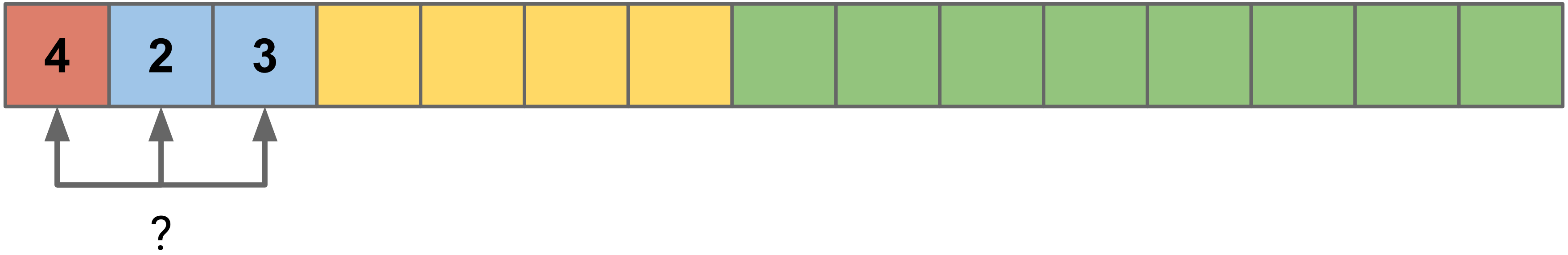


5, 7, 8, 9

Heap Sort

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

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

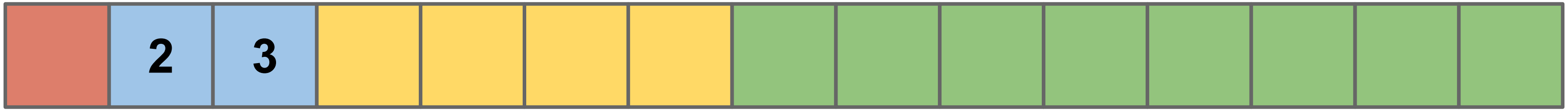


5, 7, 8, 9

Heap Sort

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

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

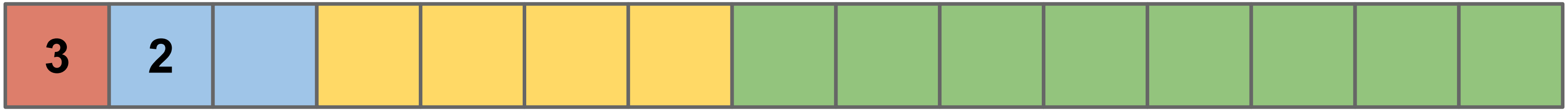


4, 5, 7, 8, 9

Heap Sort

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

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

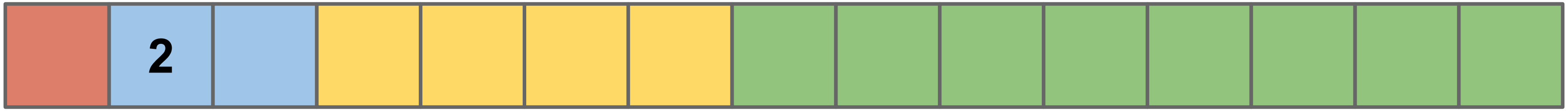


4, 5, 7, 8, 9

Heap Sort

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

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

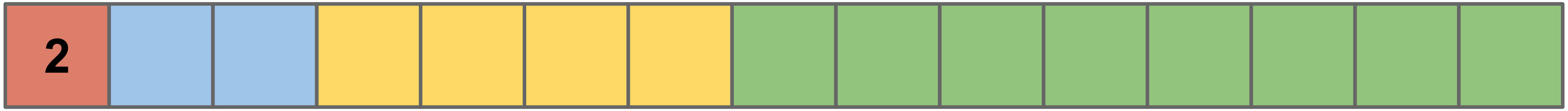


3, 4, 5, 7, 8, 9

Heap Sort

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

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

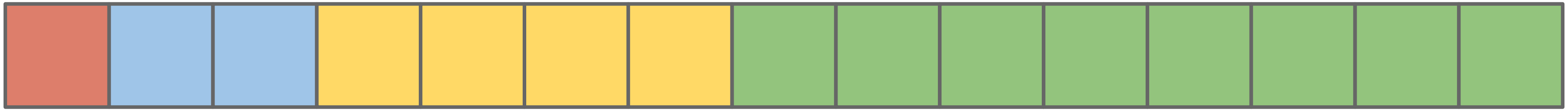


3, 4, 5, 7, 8, 9

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?

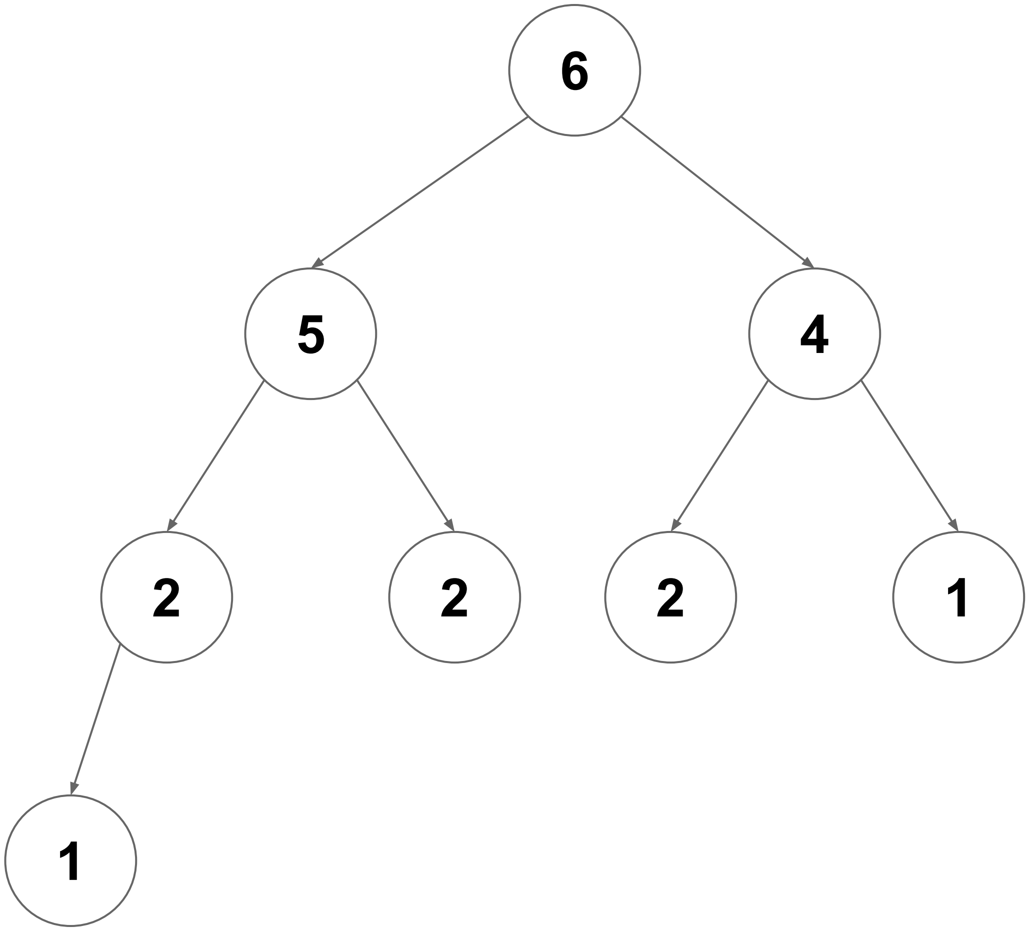
Updating Heap Elements

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

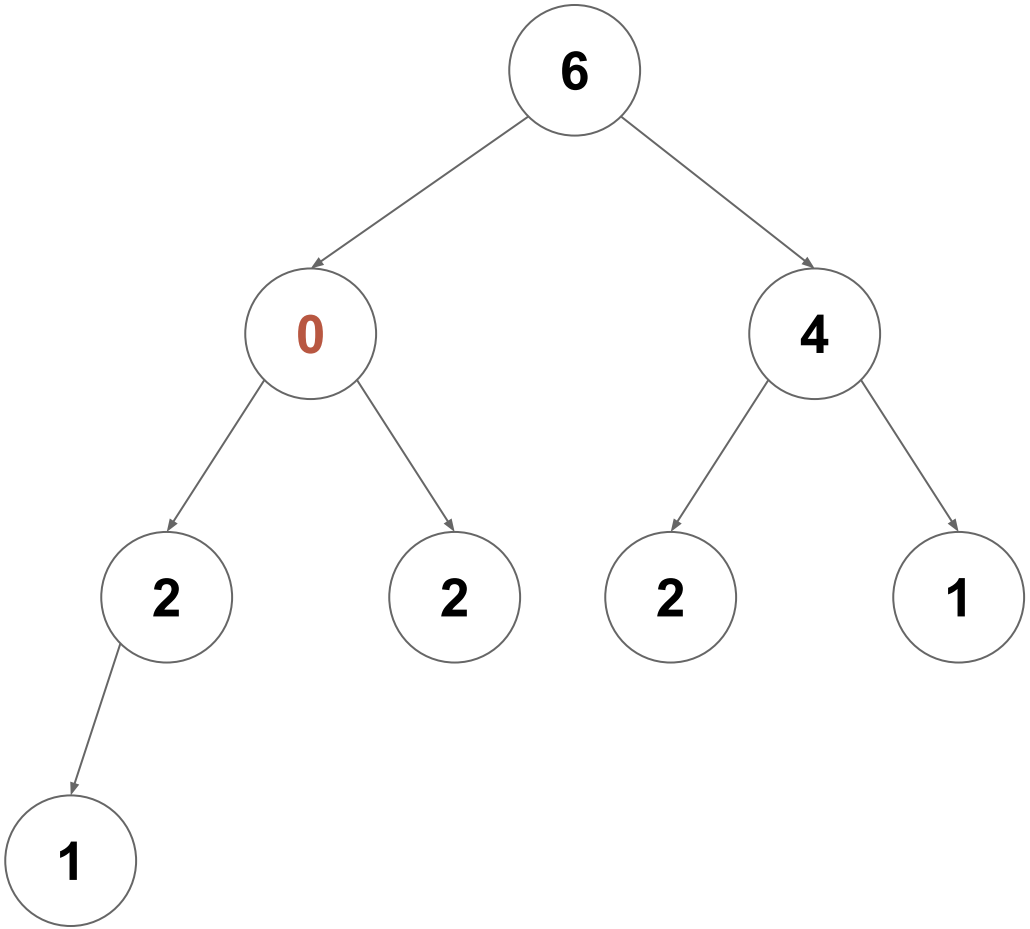
`Heap.update`

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



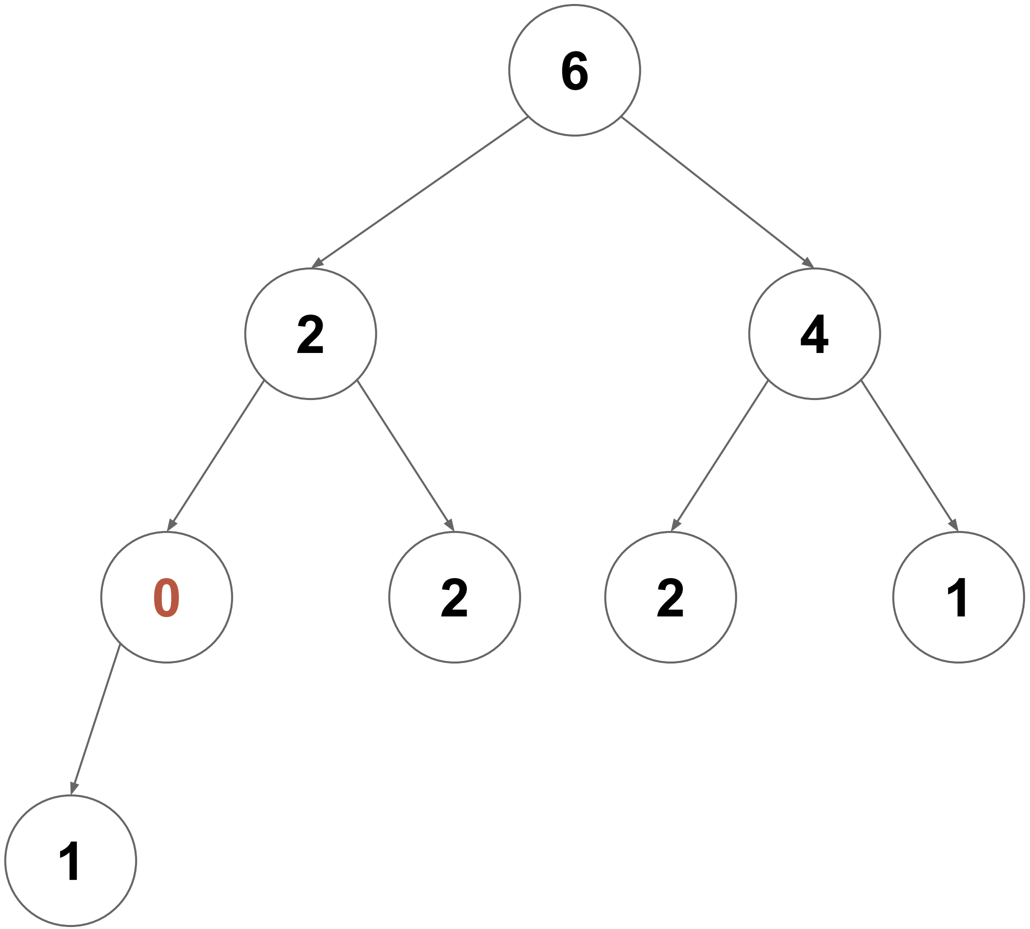
`Heap.update`

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



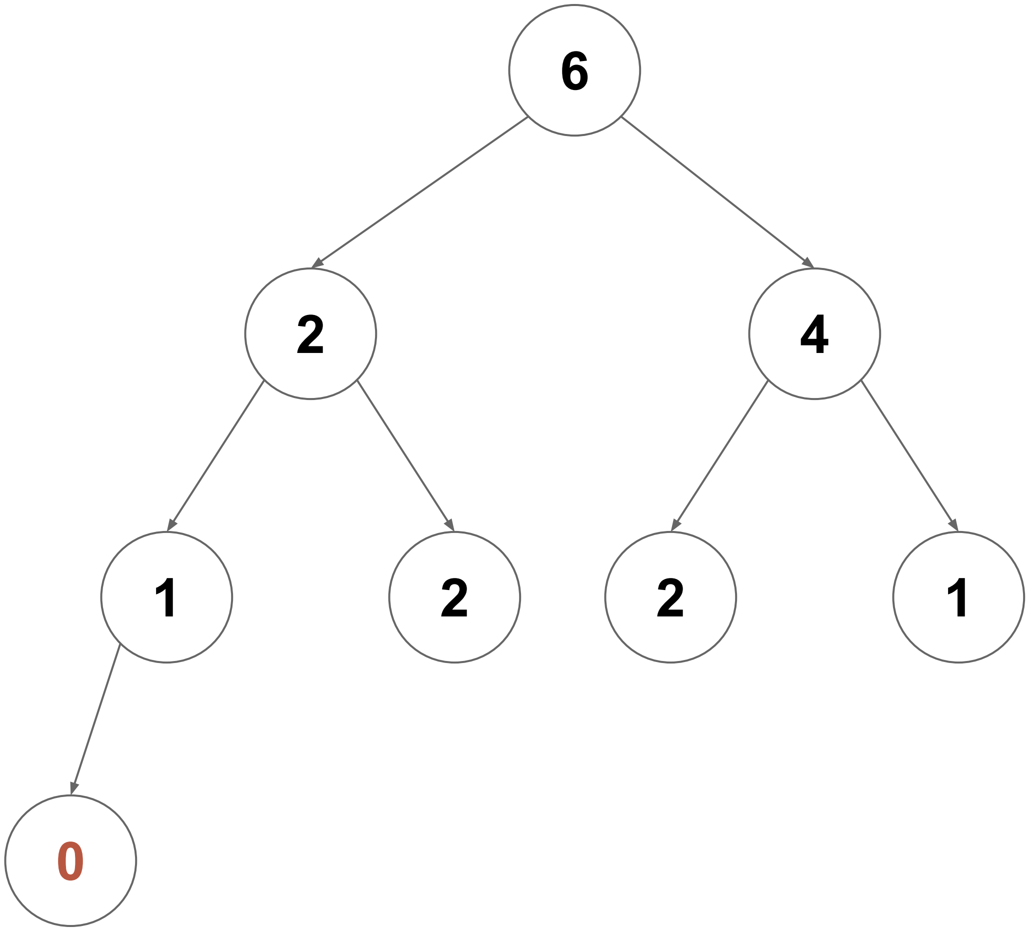
`Heap.update`

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



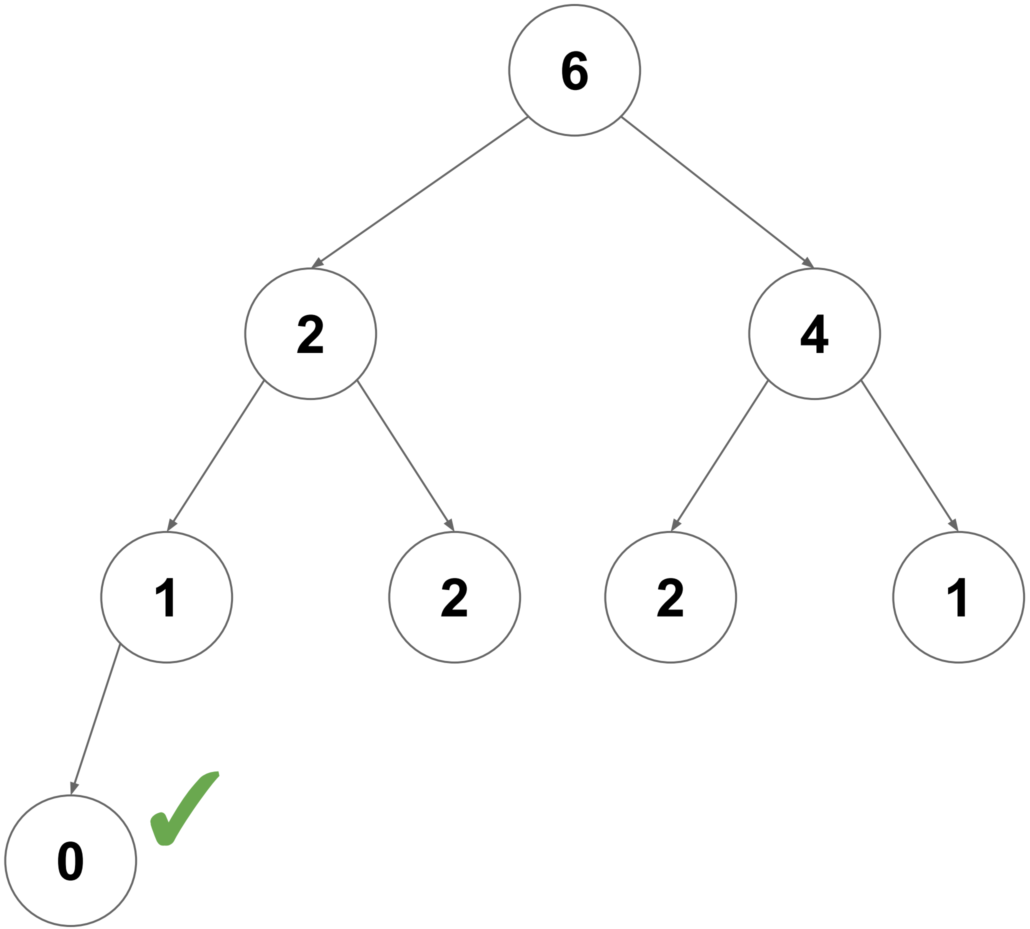
`Heap.update`

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

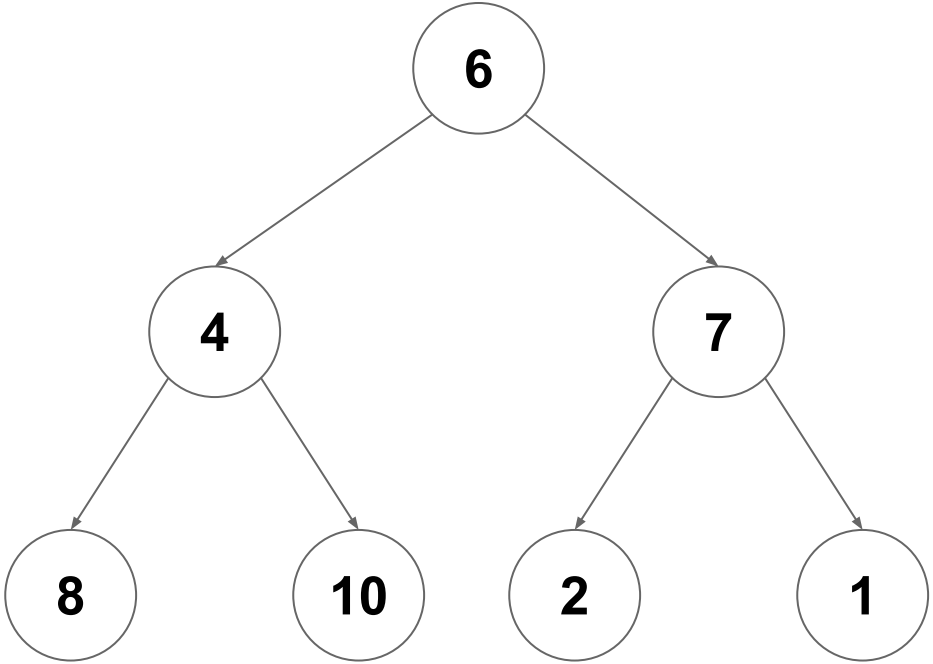
Output: Array re-ordered to be a heap

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?*

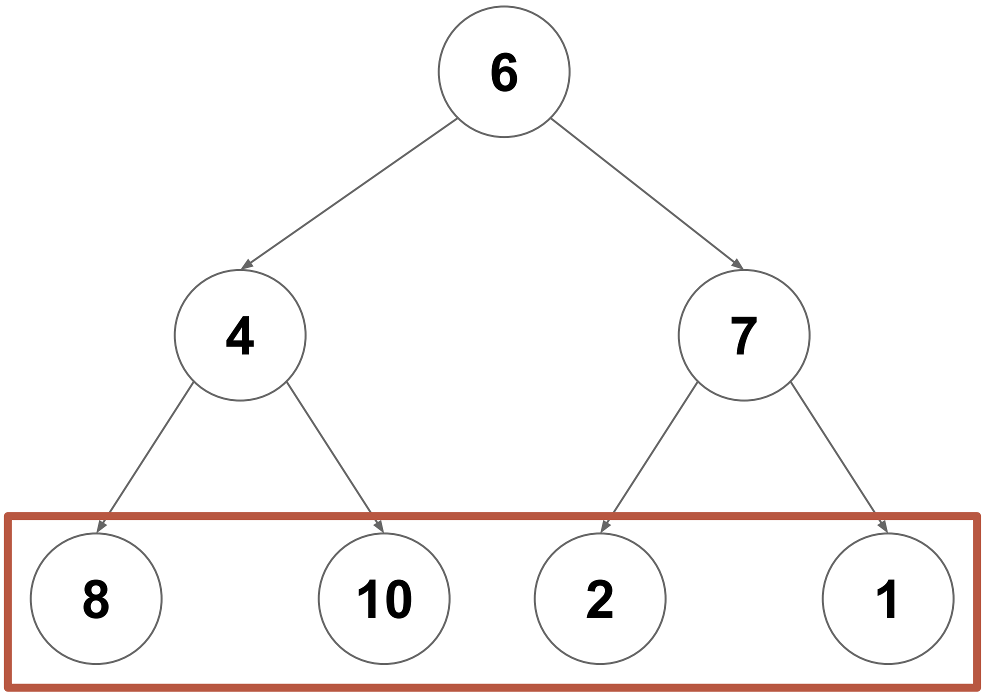
Heapify

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



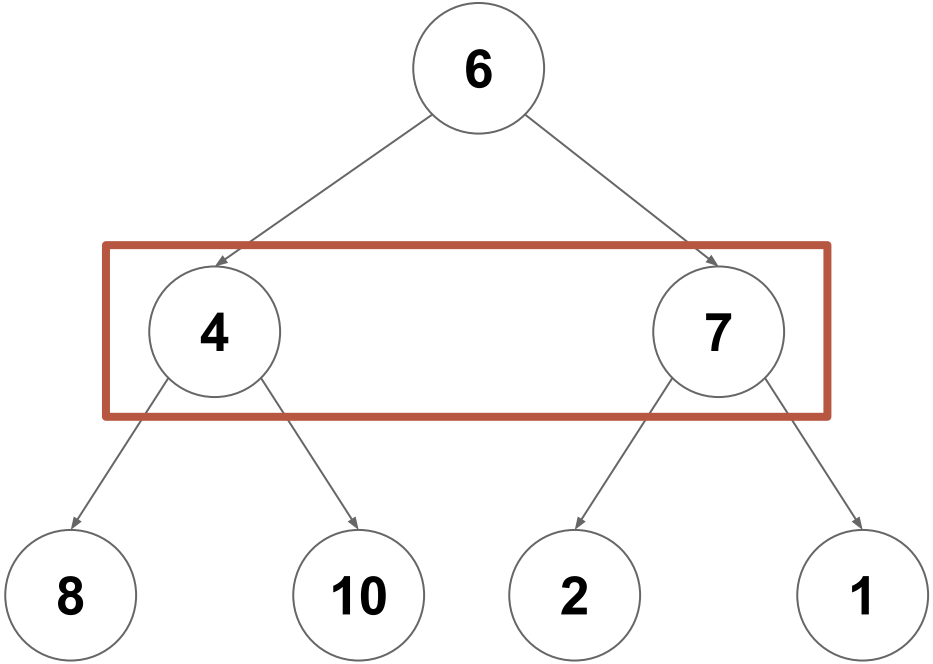
Heapify

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



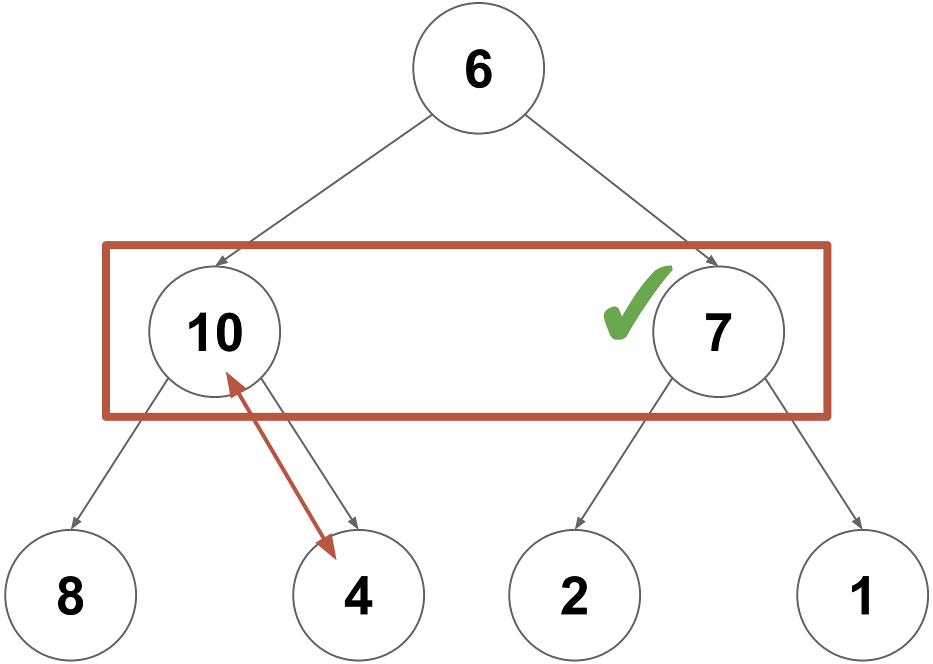
Heapify

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



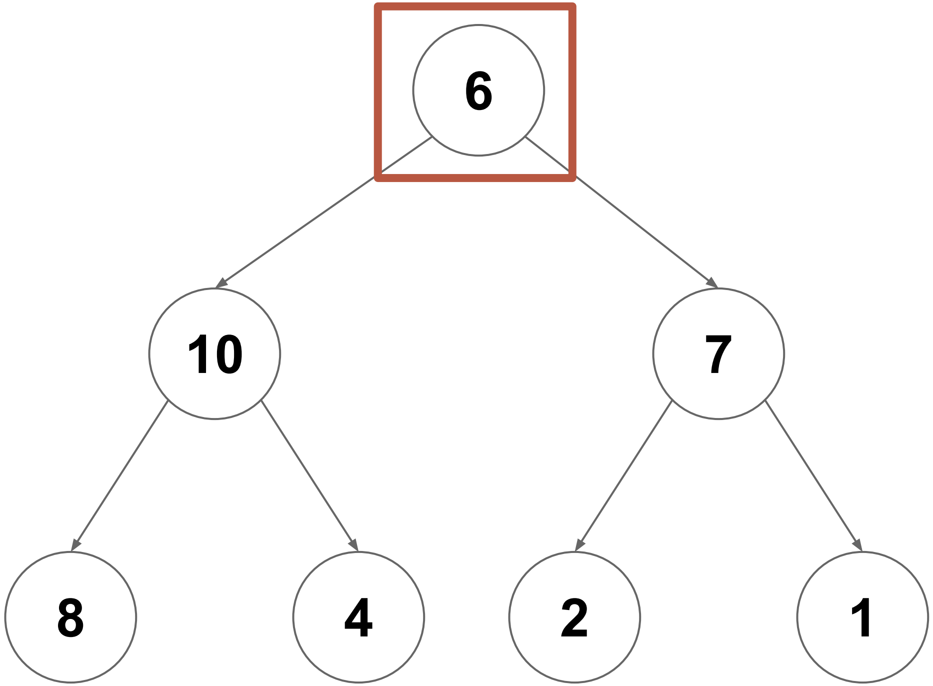
Heapify

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



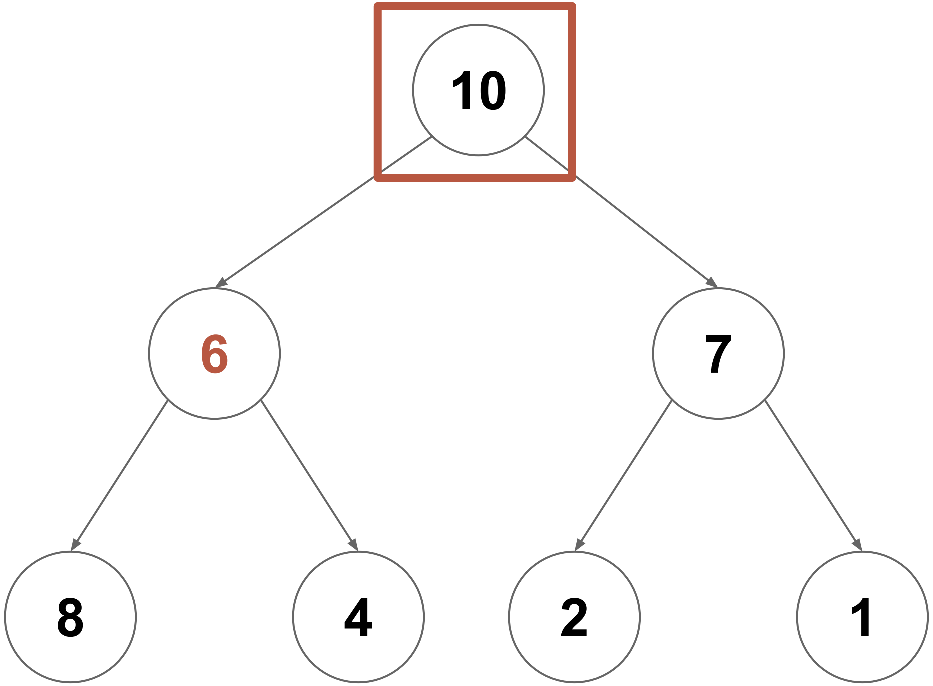
Heapify

Continue upwards (now at most 2 swaps per node)



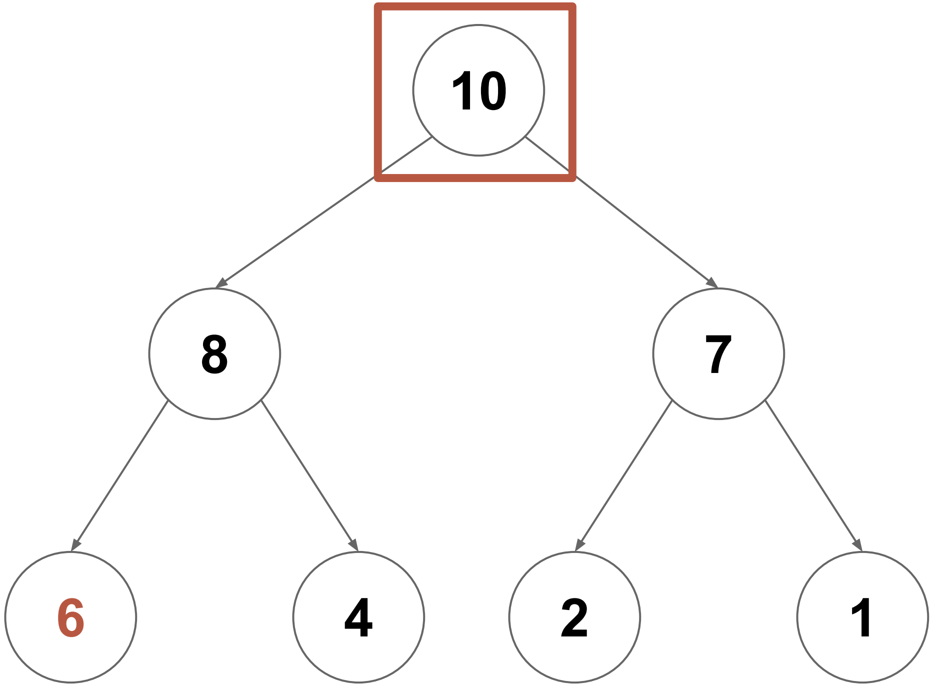
Heapify

Continue upwards (now at most 2 swaps per node)



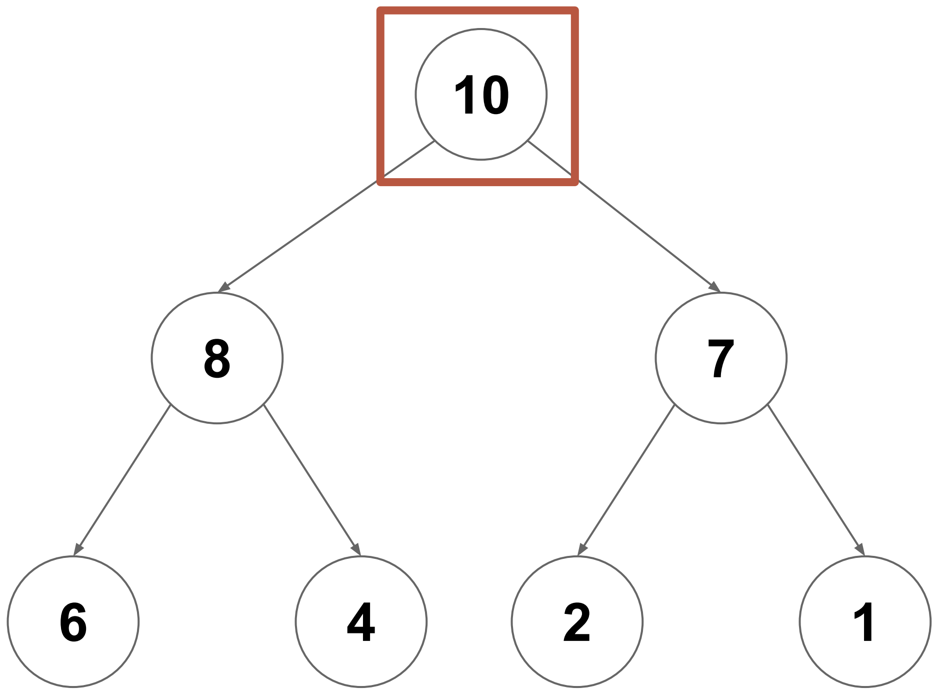
Heapify

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)

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

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

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

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

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

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)

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

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

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

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

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each item)

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The `mutable.Set[T]` ADT

`add(element: T): Unit`

Store one copy of **`element`** if not already present

`apply(element: T): Boolean`

Return true if **`element`** is present in the set

`remove(element: T): Boolean`

Remove **`element`** if present, or return false if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order doesn't matter, and multiple copies with the same key is OK)

The `mutable.Bag[T]` ADT

`add(element: T): Unit`

Register the presence of a new (copy of) **`element`**

`apply(element: T): Boolean`

Return the number of copies of **`element`** in the bag

`remove(element: T): Boolean`

Remove one copy of **`element`** if present, or return false if not

Collection ADTs

Propery	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
Iterable	✓	✓	✓

(Rooted) Trees

(Even More) Tree Terminology

Rooted, Directed Tree - Has a single root node (node with no parents)

Parent of node X - A node with an out-edge to X (max 1 parent per node)

Child of node X - A node with an in-edge from X

Leaf - A node with no children

Depth of node X - The number of edges in the path from the root to X

Height of node X - The number of edges in the path from X to the deepest leaf

(Even More) Tree Terminology

Level of a node - Depth of the node + 1

Size of a tree (*n*) - The number of nodes in the tree

Height/Depth of a tree (*d*) - Height of the root/depth of the deepest leaf

(Even More) Tree Terminology

Binary Tree - Every vertex has at most 2 children

Complete Binary Tree - All leaves are in the deepest two levels

Full Binary Tree - All leaves are at the deepest level, therefore every vertex has exactly 0 or 2 children, and $\boldsymbol{d = \log(n)}$

Quick Scala Tips

```
class TreeNode[T](
  var _value: T,
  var _left: Option[TreeNode[T]]
  var _right: Option[TreeNode[T]]
)

class Tree[T] {
  var root: Option[TreeNode[T]] = None // empty tree
}
```

We've seen how we can use options for objects that may not exist…

Quick Scala Tips

```scala
trait Tree[+T]

case class TreeNode[T](
  value: T,
  left: Tree[T],
  right: Tree[T]
) extends Tree[T]

case object EmptyTree extends Tree[Nothing]
```

But we can also use Traits and case classes…

Quick Scala Tips

```scala
trait Tree[+T]

case class TreeNode[T](
  value: T,
  left: Tree[T],
  right: Tree[T]
) extends Tree[T]

case object EmptyTree extends Tree[Nothing]
```

`TreeNode` and `EmptyTree` are two cases of `Tree`

But we can also use Traits and case classes…

Case Classes/Objects

Case Classes/Objects have two important features:

1. **Inline Constructors (no `new`):**

```
TreeNode(10,EmptyTree,EmptyTree)
```

2. **Match deconstructors:**

```
foo match { case TreeNode(v, l, r) => … }
```

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

If `root` is a `TreeNode` with value `v`, and subtrees `left` and `right`, print `v`, then call `printTree` on `left` and `right`

Case Classes/Objects

```
def printTree[T](root: ImmutableTree[T], indent: Int) = {
  root match {
    case TreeNode(v, left, right) =>
      print((" " * indent) + v)
      printTree(left, indent + 2)
      printTree(right, indent + 2)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

If `root` is an `EmptyTree` then don't do anything

Computing Tree Height

The height of a tree is the height of the root

Computing Tree Height

The height of a tree is the height of the root

The children of the root are each roots of the left and right subtrees

Computing Tree Height

The height of a tree is the height of the root

The children of the root are each roots of the left and right subtrees

So we can compute height recursively:

$$h(root) = \begin{cases} 0 & \textbf{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \textbf{otherwise} \end{cases}$$

Computing Tree Height

```
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

$$h(root) = \begin{cases} 0 & \text{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \text{otherwise} \end{cases}$$

Computing Tree Height

```
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

Case classes have a nice mapping onto functions with multiple cases

$$h(root) = \begin{cases} 0 & \text{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \text{otherwise} \end{cases}$$

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L`.key` < X`.key`

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node $\boldsymbol{X_L}$ in the left subtree of node $\boldsymbol{X}$: $\boldsymbol{X_L}$`.key` < $\boldsymbol{X}$`.key`
- For every node $\boldsymbol{X_R}$ in the right subtree of node $\boldsymbol{X}$: $\boldsymbol{X_R}$`.key` > $\boldsymbol{X}$`.key`

Binary Search Tree

A **<u>Binary Search Tree</u>** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L`.key` < X`.key`
- For every node X_R in the right subtree of node X: X_R`.key` > X`.key`

X **<u>partitions</u>** its children

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at `root`

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (If yes, search the right subtree)

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity?

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity? (how many times do we call **`find`***)?*

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity? (how many times do we call `find`)? ***O(d)***

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at `root`

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at `root`

1. Is `root` empty? (insert here)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)?

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)? ***O*(*d*)**

Remove

Goal: Remove the item with key ***k*** from a BST rooted at **root**

1. **`find`** the iterm
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

We'll look at this in more detail later, but for now…

What's the complexity? ***O*(*d*)**

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of **<**)

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of **<**)

Idea 2: Only store one copy of each element, but also store a count

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?*

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of ***n****?* $\mathbf{O(n)}$

BST Operations

Operation	Runtime
`find`	***O*(*d*)**
`insert`	***O*(*d*)**
`remove`	***O*(*d*)**

*What is the runtime in terms of **n**? **O(n)***

Does it need to be that bad?

Next time...

Balancing Trees…