

# Heaps and Heapsort



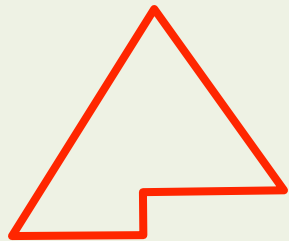
# Heaps

- A **heap** is a **binary tree of  $T$**  that satisfies two properties:
  - Global shape property: it is a **complete binary tree**
  - Local ordering property: the label in each node is “smaller than or equal to” the label in each of its child nodes

# Heaps

- A **heap** is a **binary tree of  $T$**  that satisfies two properties:
  - Global shape property: it is a **complete**

A **complete** binary tree is one in which all levels are “full” except possibly the bottom level, with any nodes on the bottom level as far left as possible.



in each  
” the label in

# Heaps

Also in the picture is (as with BSTs, sorting, etc.) a total preorder that makes this notion precise.

of  $T$  that

a **complete**

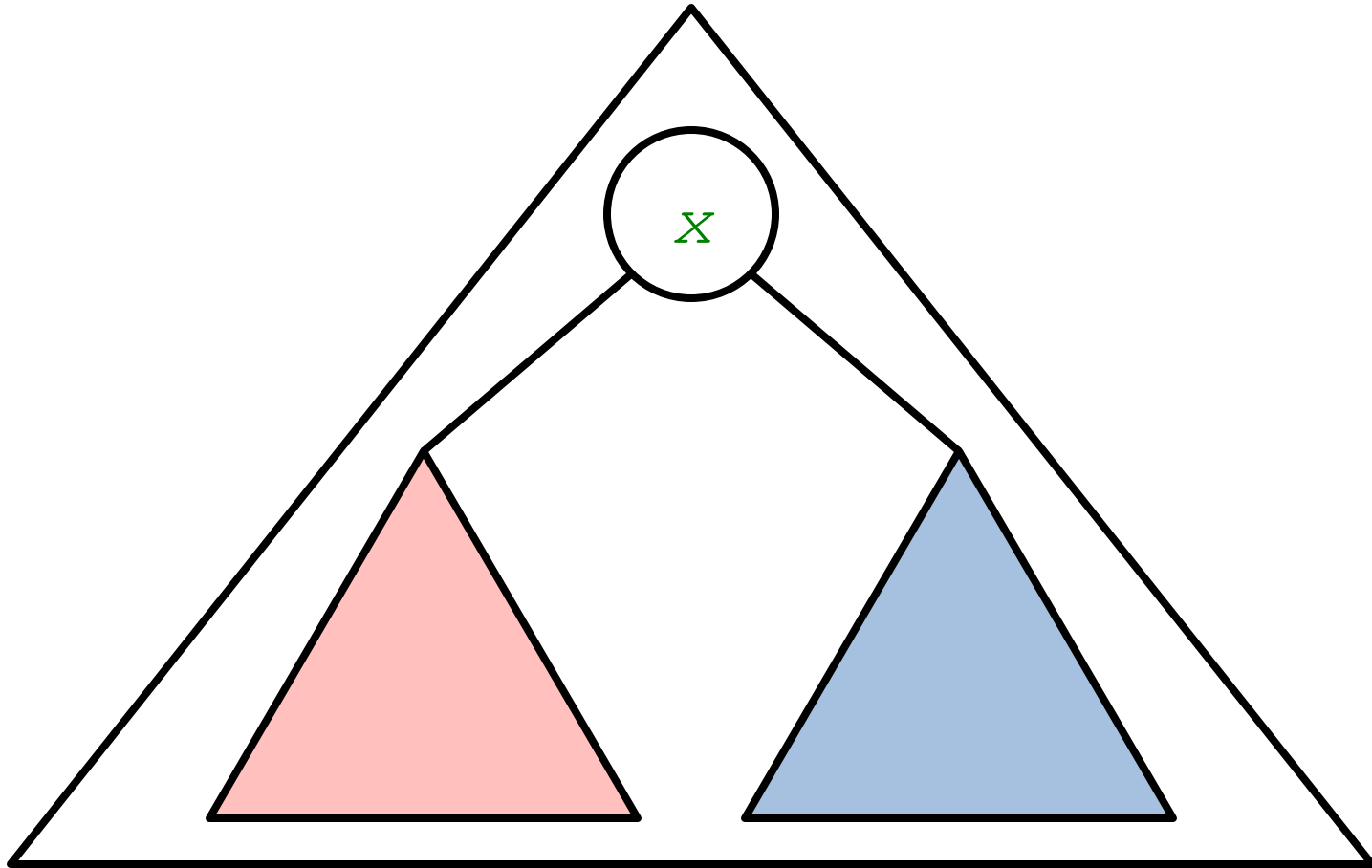
**binary tree**

- Local ordering property: the label in each node is “smaller than or equal to” the label in each of its child nodes

# Simplification

- For simplicity in the following illustrations, we use only one kind of example:
  - $T = \textit{integer}$
  - The ordering is  $\leq$
- Because heaps are used in sorting, where duplicate values may be involved, we allow that multiple nodes in a heap may have the same labels (i.e., we will *not* assume that the labels are unique)

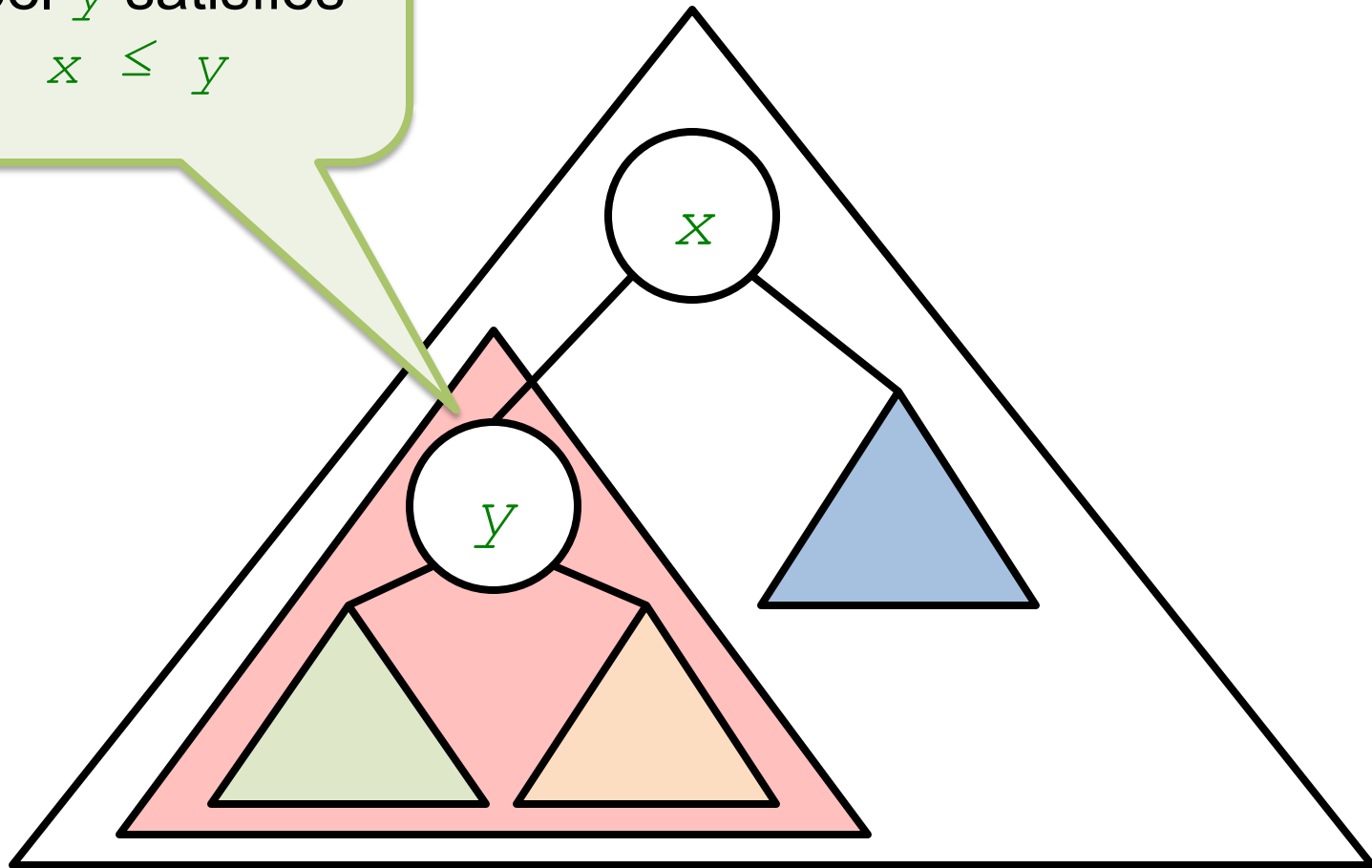
# The Big Picture



# The Big Picture

This tree's root  
label  $y$  satisfies

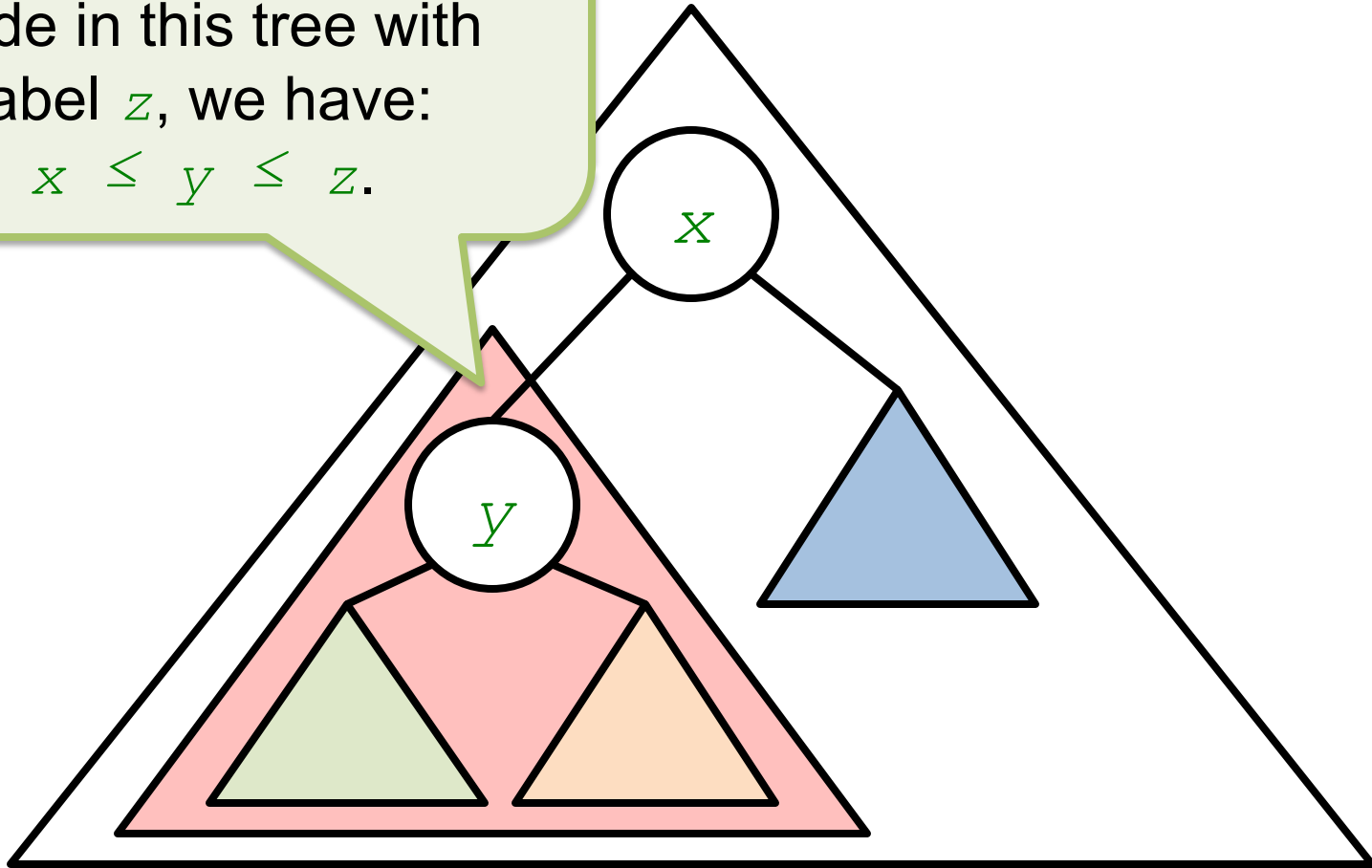
$$x \leq y$$



Observe: This tree is also a heap; and for each node in this tree with label  $z$ , we have:

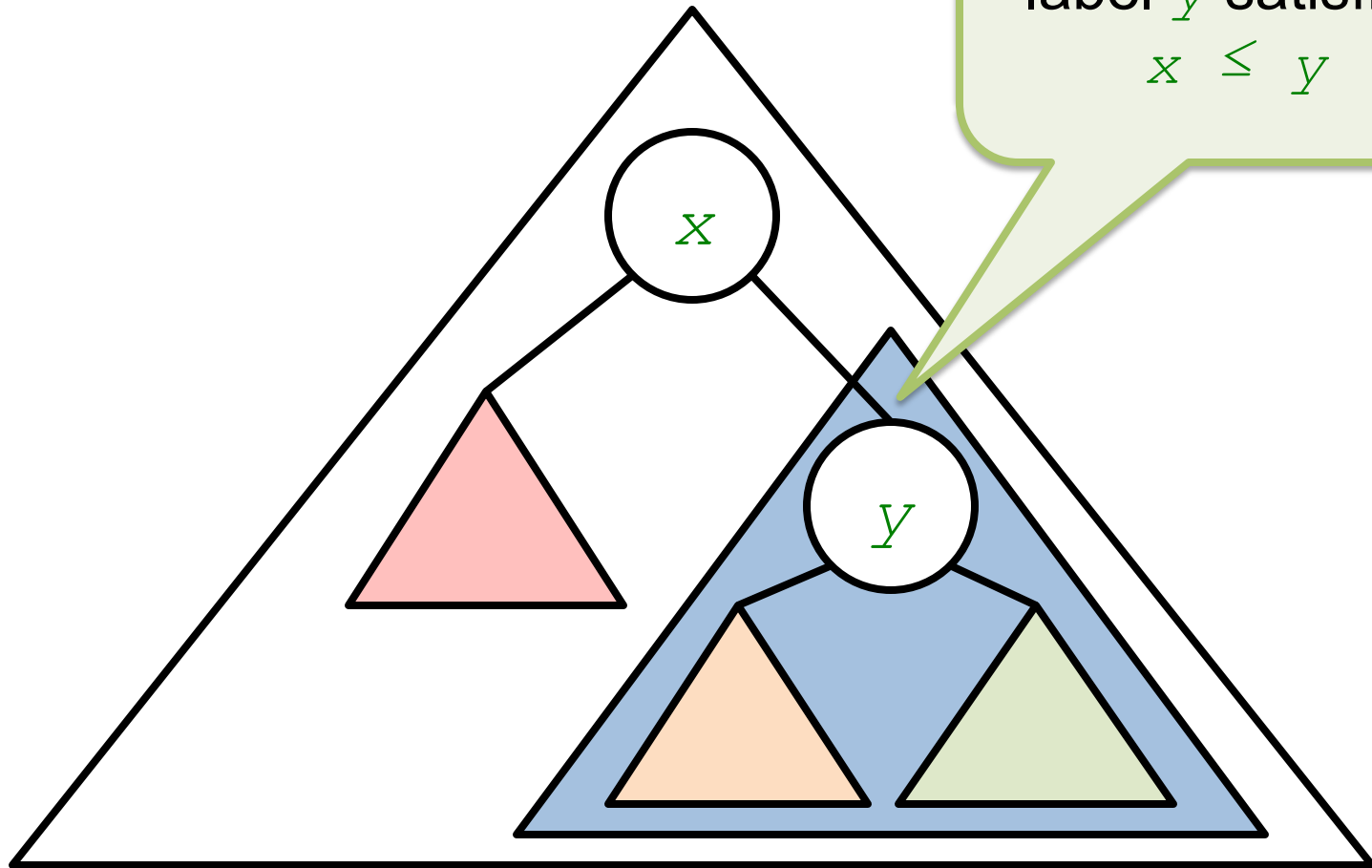
$$x \leq y \leq z.$$

# ig Picture





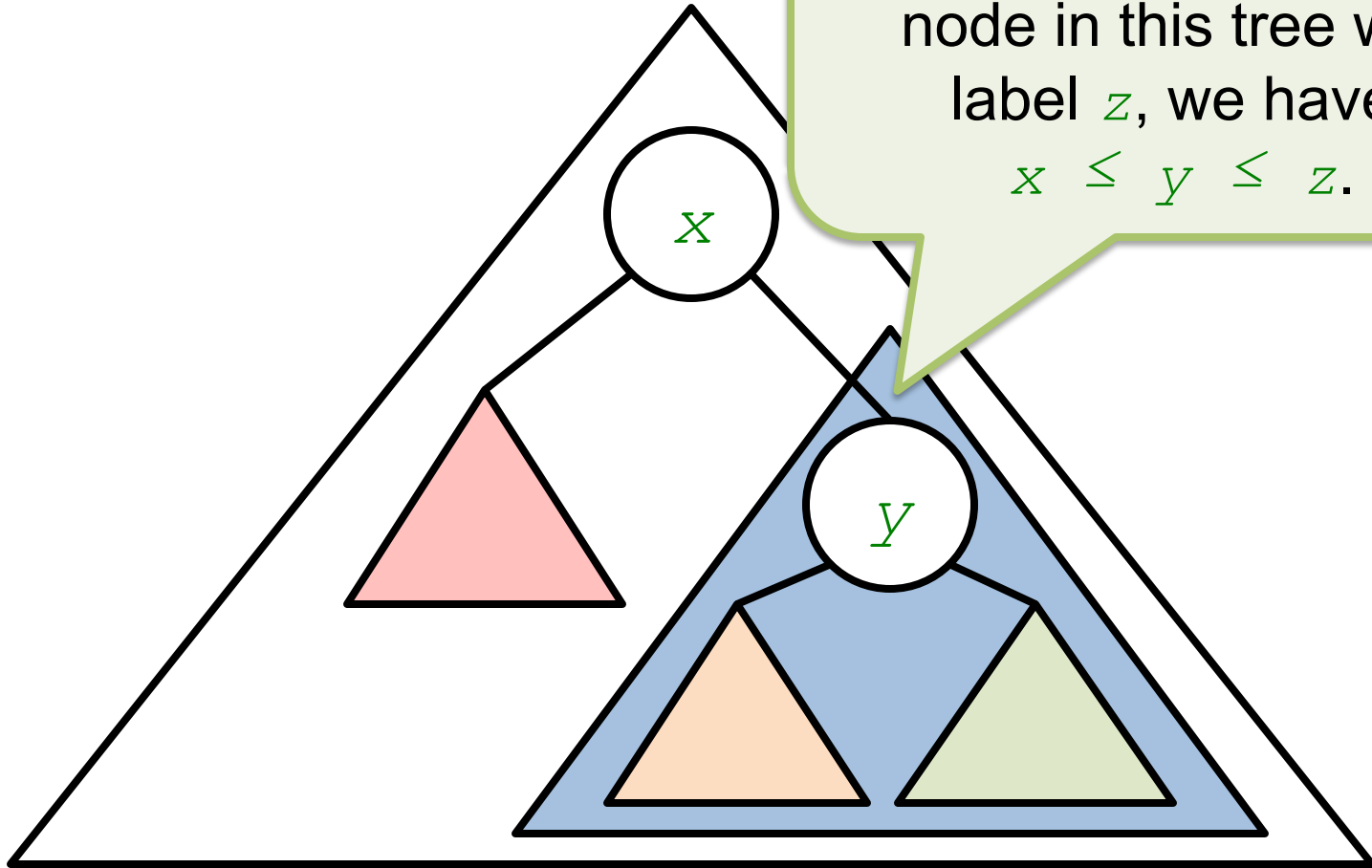
# The Big Picture



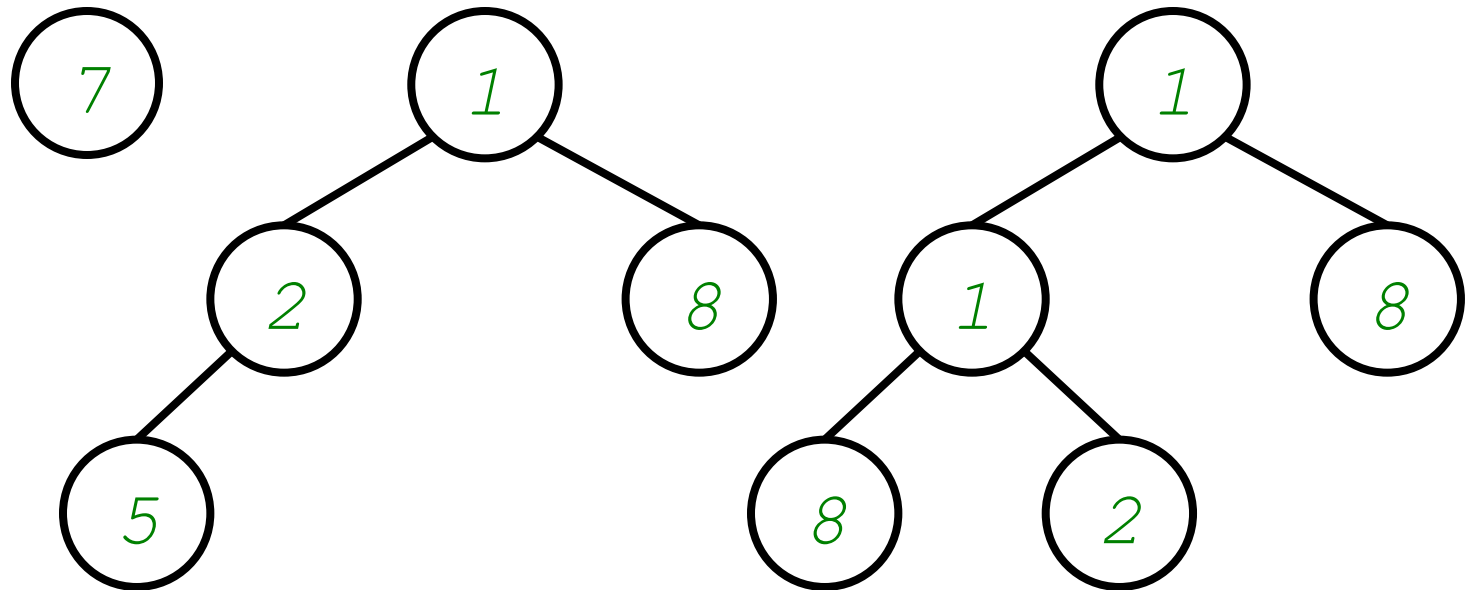
# The Big P

Observe: This tree is also a heap; and for each node in this tree with label  $z$ , we have:

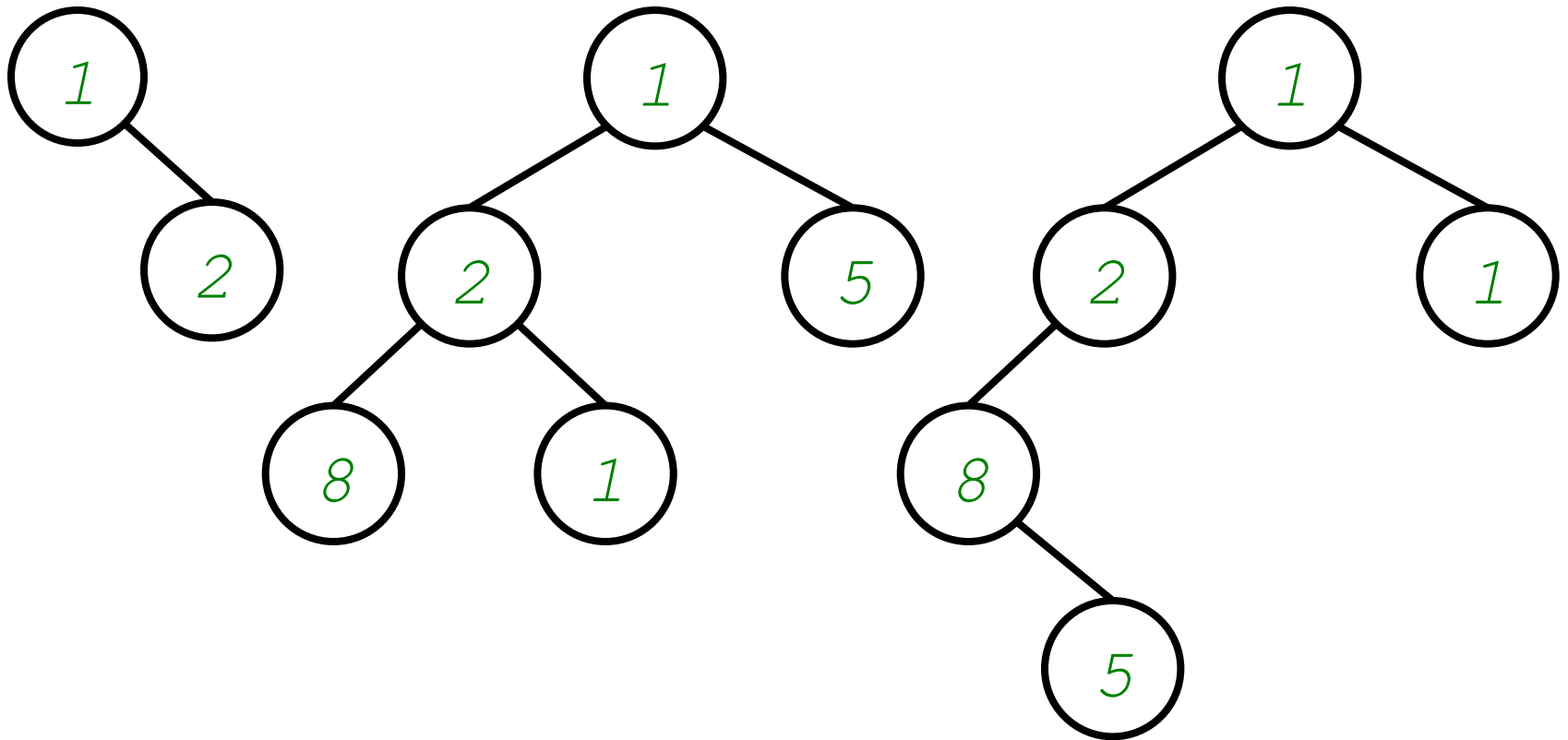
$$x \leq y \leq z.$$



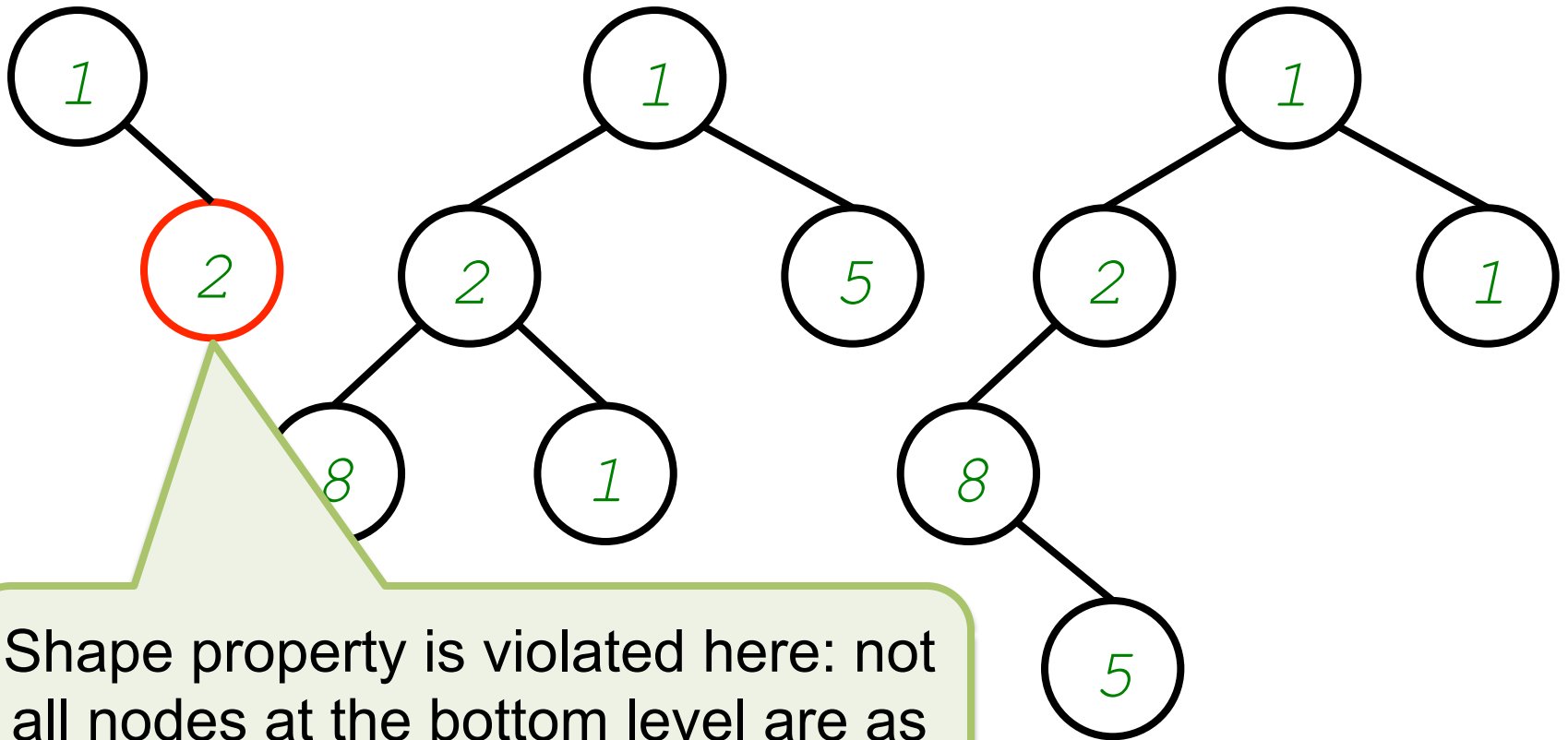
# Examples of Heaps



# Non-Examples of Heaps

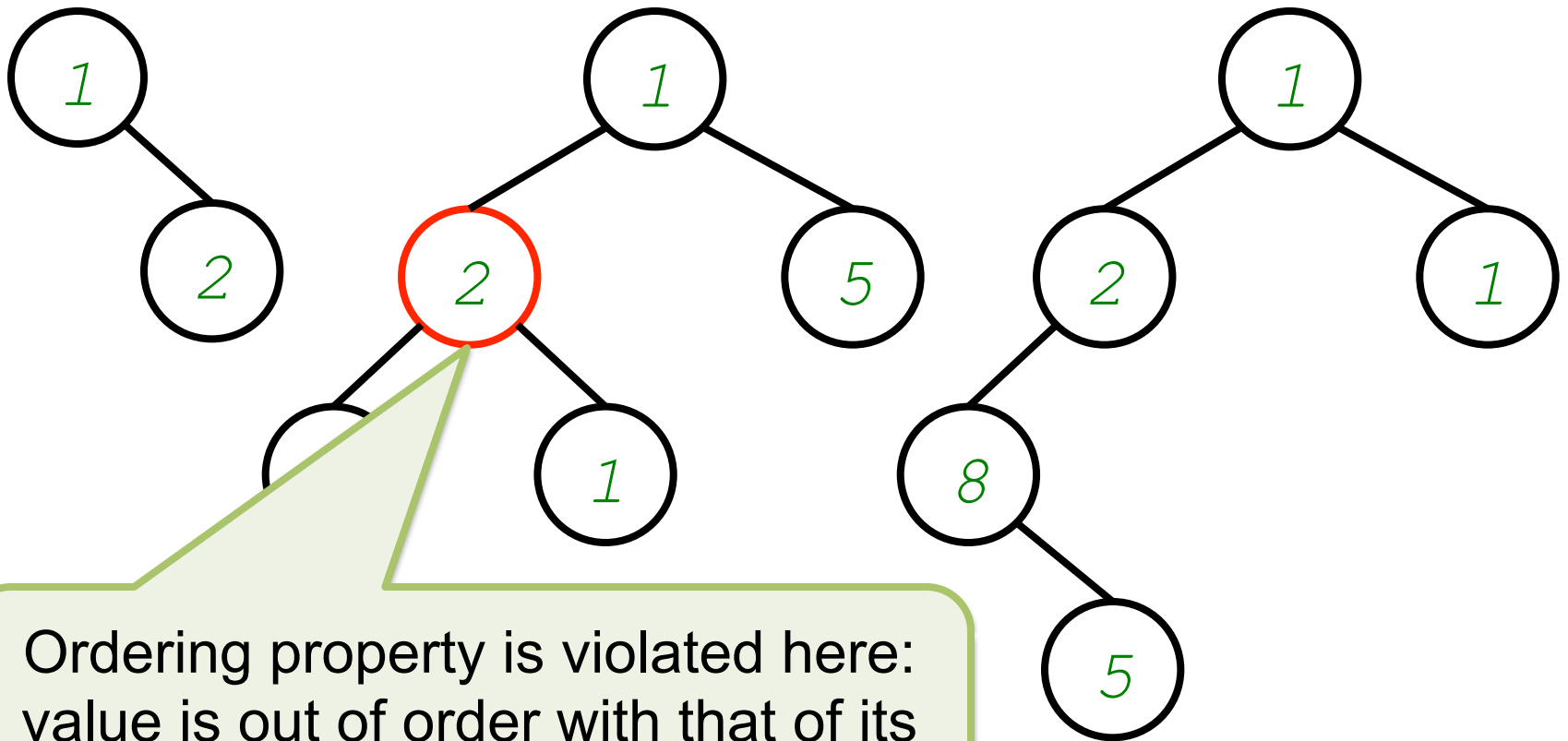


# Non-Examples of Heaps



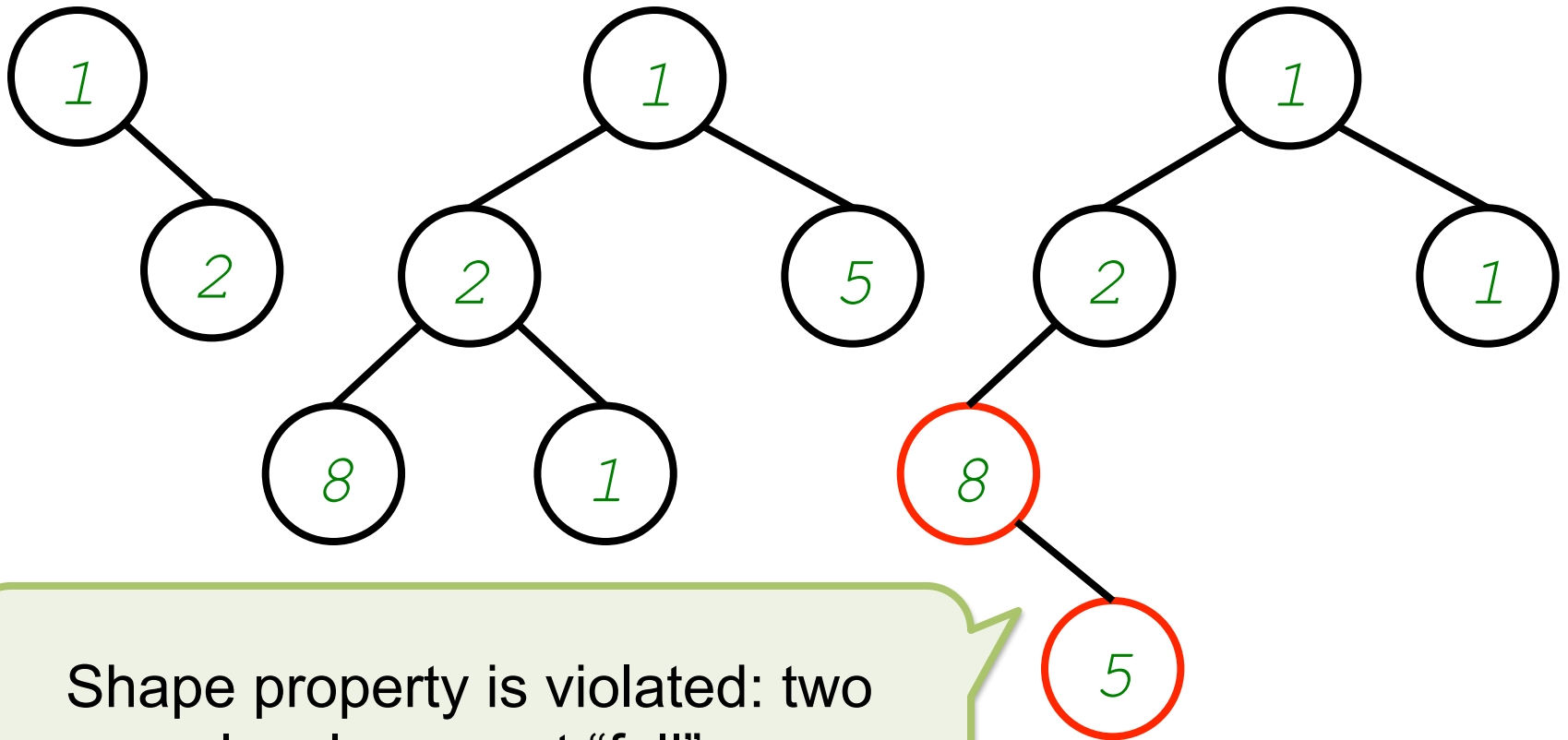
Shape property is violated here: not all nodes at the bottom level are as far left as possible.

# Non-Examples of Heaps



Ordering property is violated here:  
value is out of order with that of its  
right child.

# Non-Examples of Heaps



# Heapsort

- A heap can be used to represent the values in a `SortingMachine`, as follows:
  - In `changeToExtractionMode`, arrange all the values into a heap
  - In `removeFirst`, remove the root, and adjust the slightly mutilated heap to make it a heap again



# Heapsort

Why should this work?

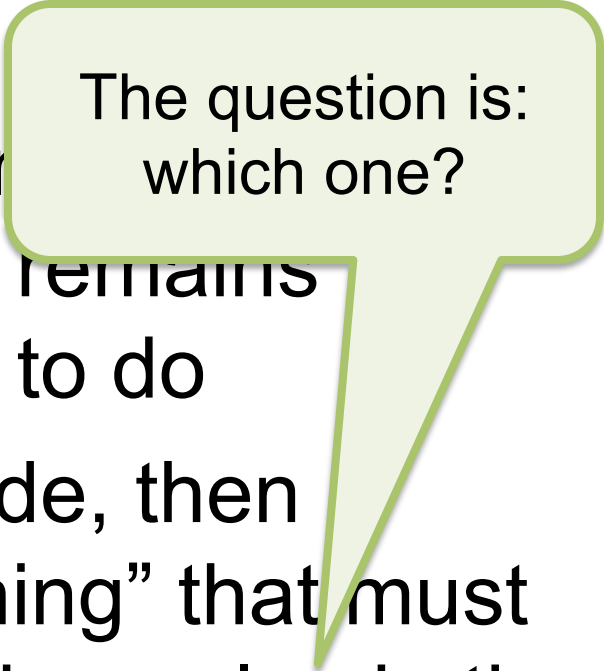
- A heap can be used to sort values in a `SortingMachine`, as follows:
  - In `changeToExtractionMode`, arrange all the values into a heap
  - In `removeFirst`, remove the root, and adjust the slightly mutilated heap to make it a heap again

# How `removeFirst` Can Work

- If the root is the only node in the heap, then after removing it, what remains is already a heap; nothing left to do
- If the root is not the only node, then removing it leaves an “opening” that must be filled by moving some other value in the heap into the opening

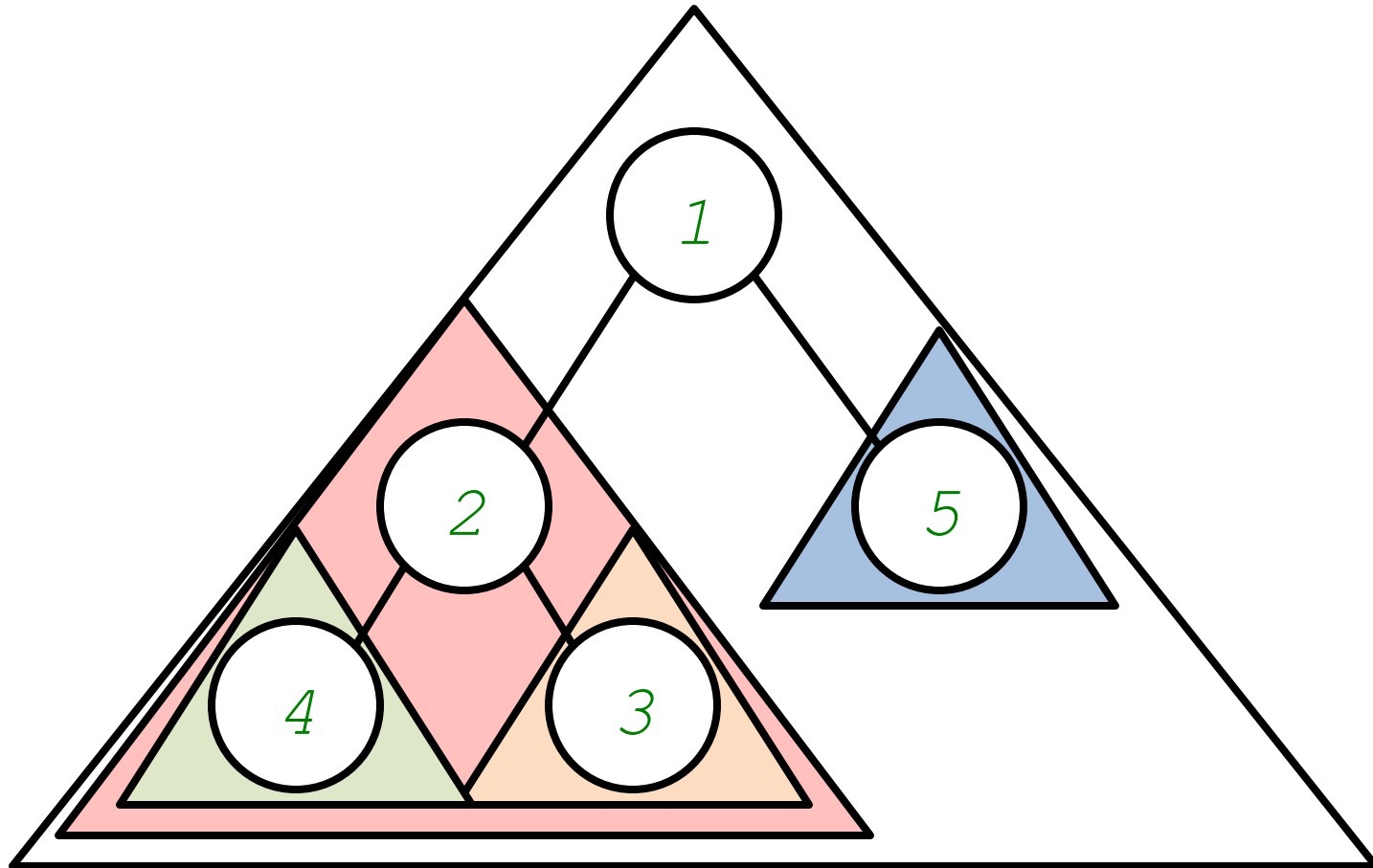
# How `removeFirst` Can Work

- If the root is the only node in the heap, then after removing it, what remains is already a heap; nothing left to do
- If the root is not the only node, then removing it leaves an “opening” that must be filled by moving some other value in the heap into the opening

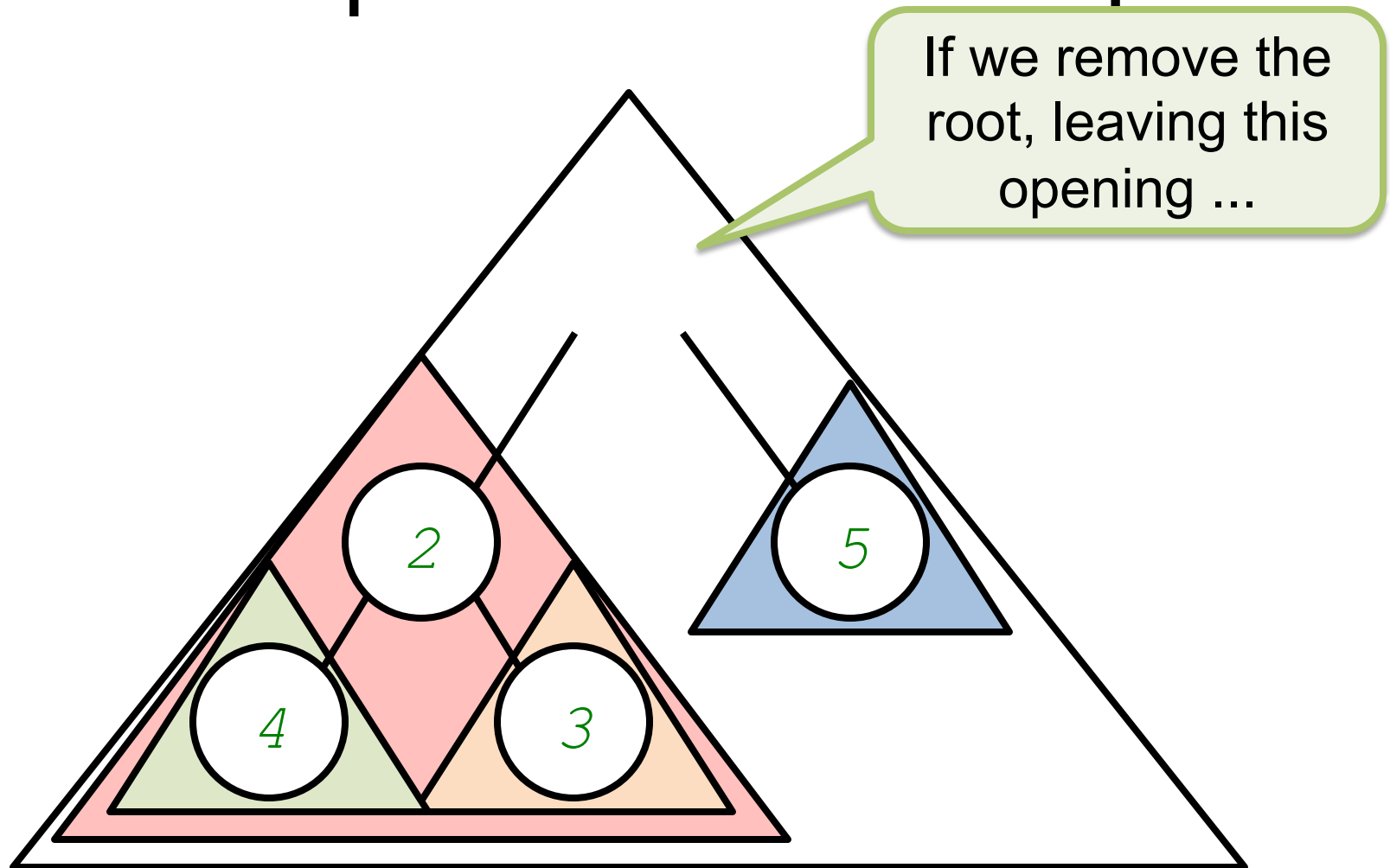


The question is:  
which one?

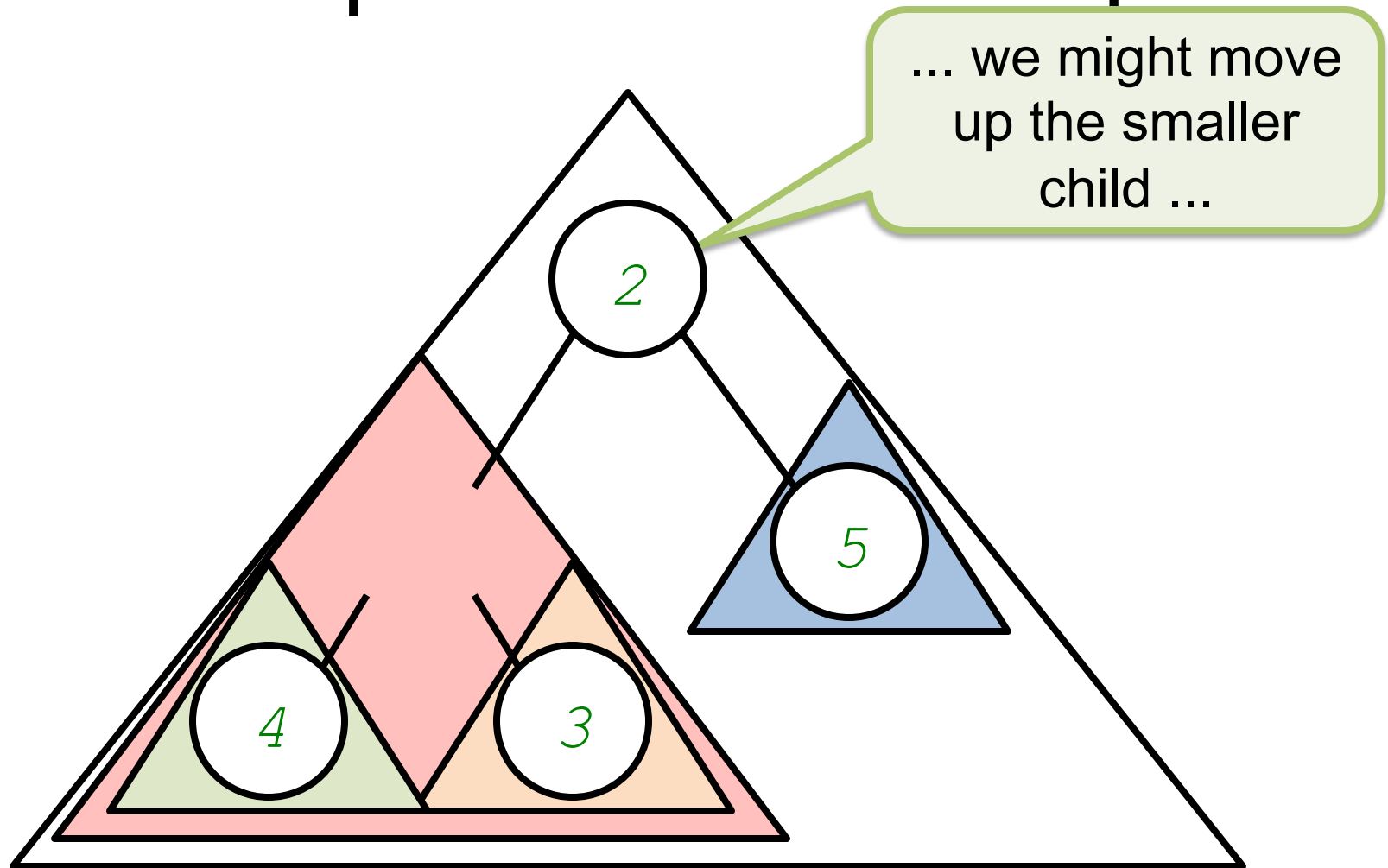
# Example: A First Attempt



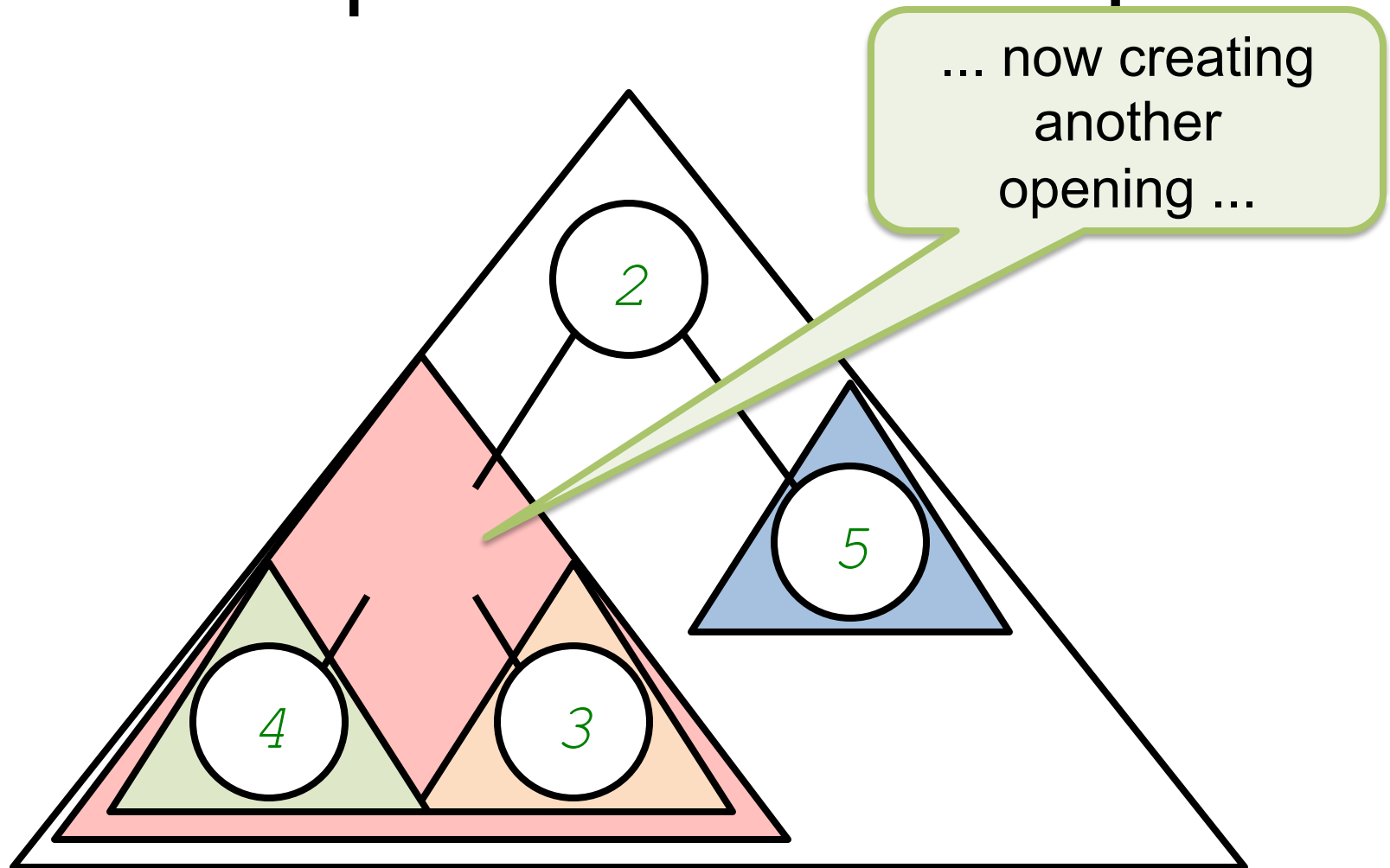
# Example: A First Attempt



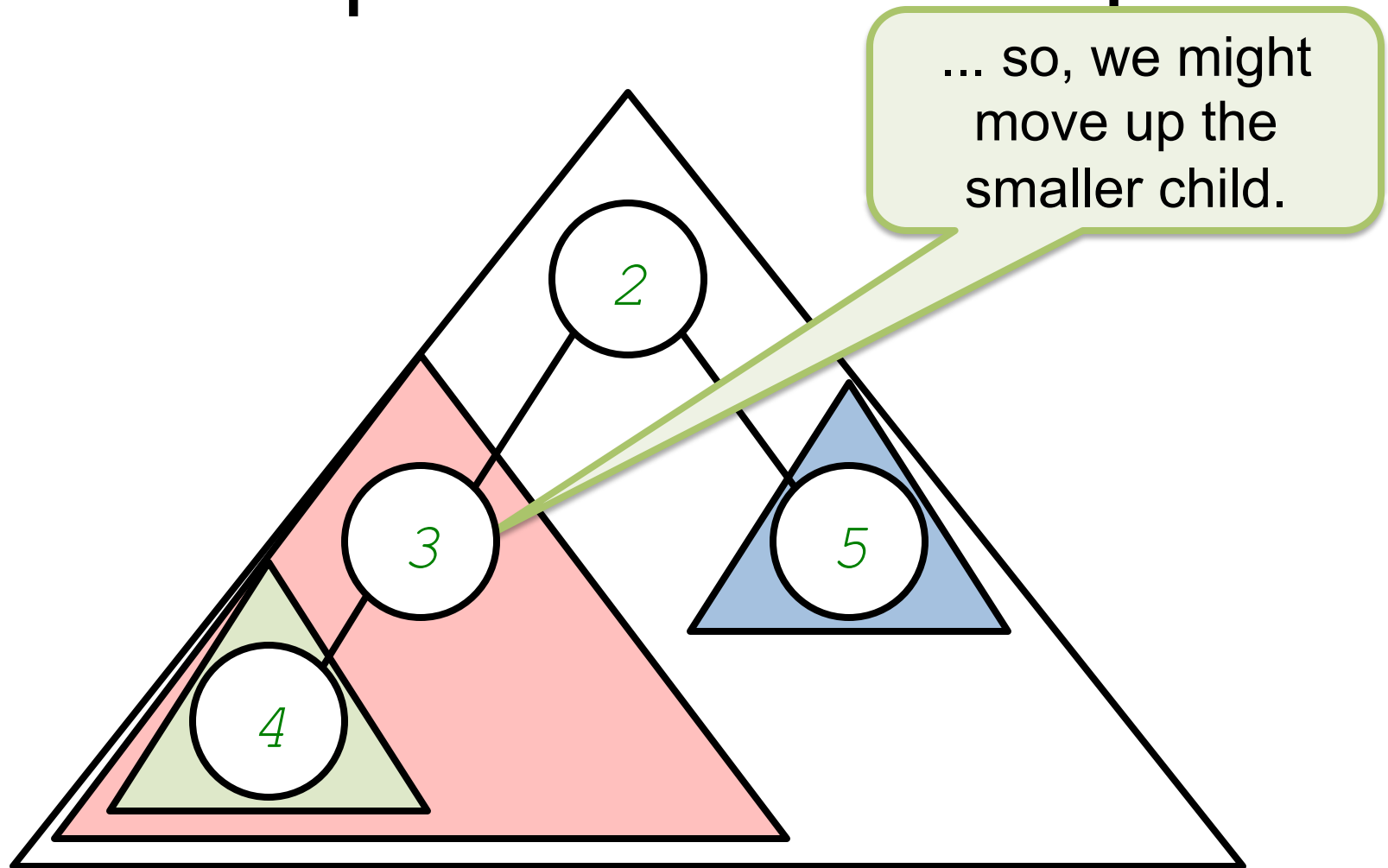
# Example: A First Attempt



# Example: A First Attempt

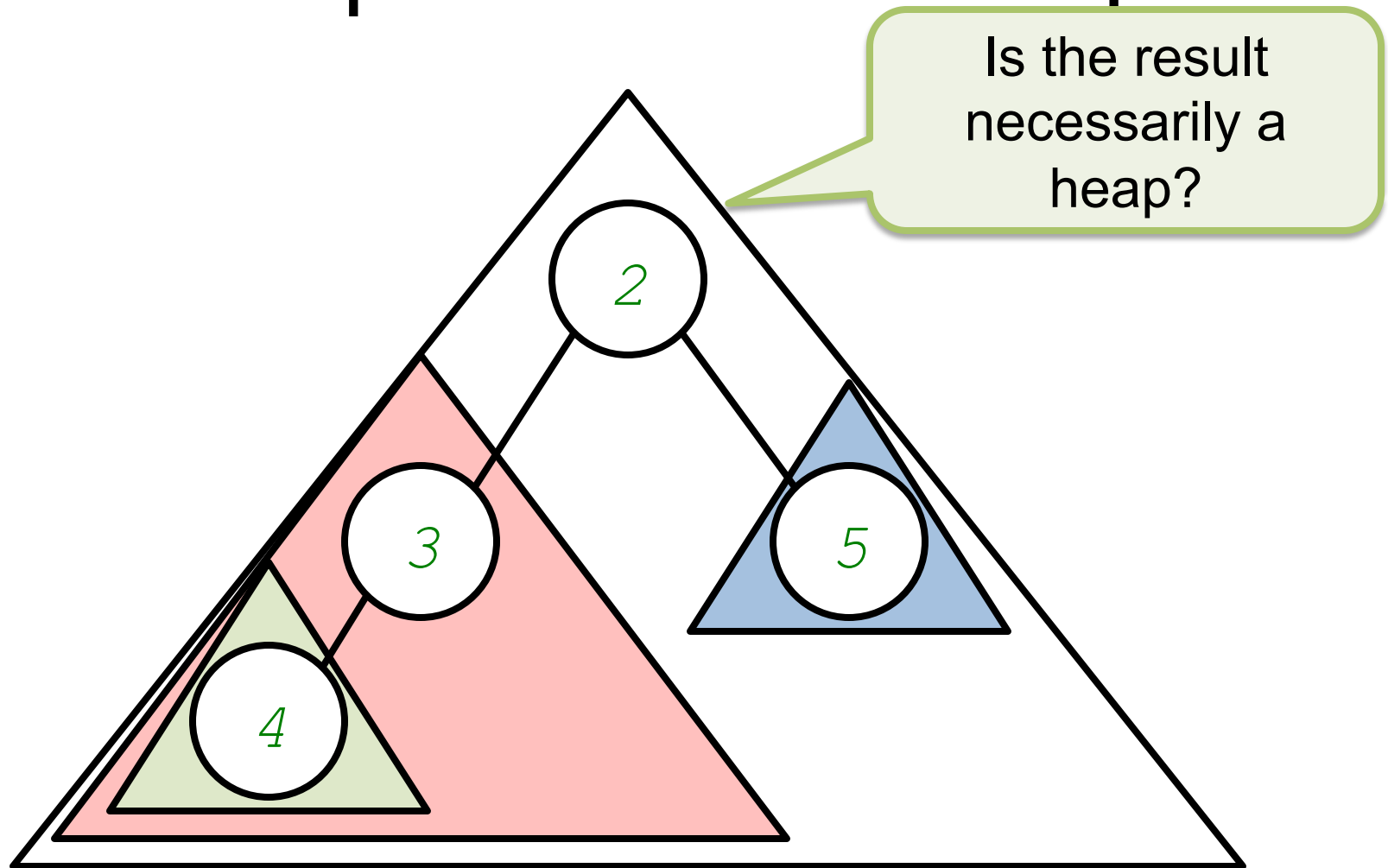


# Example: A First Attempt

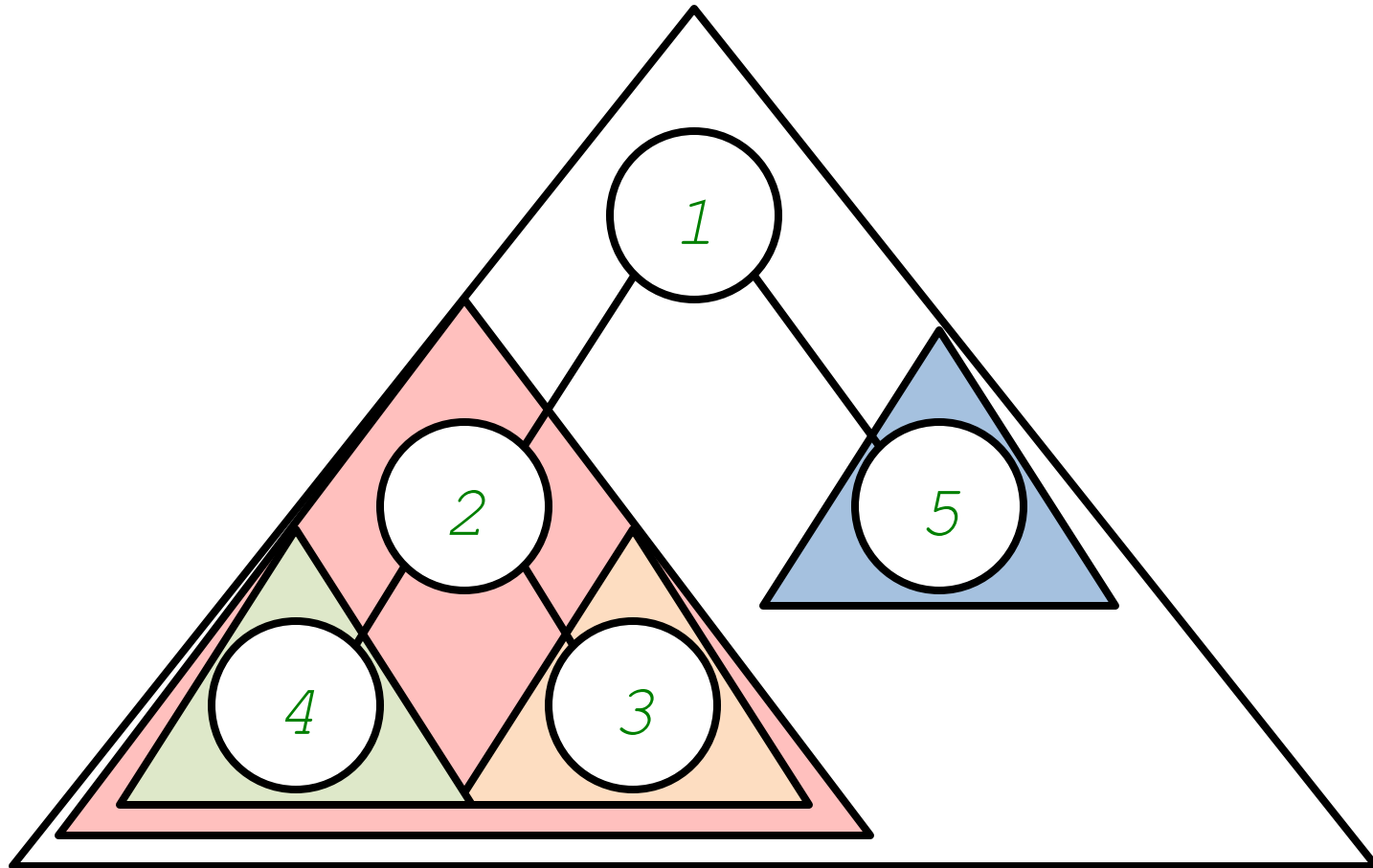




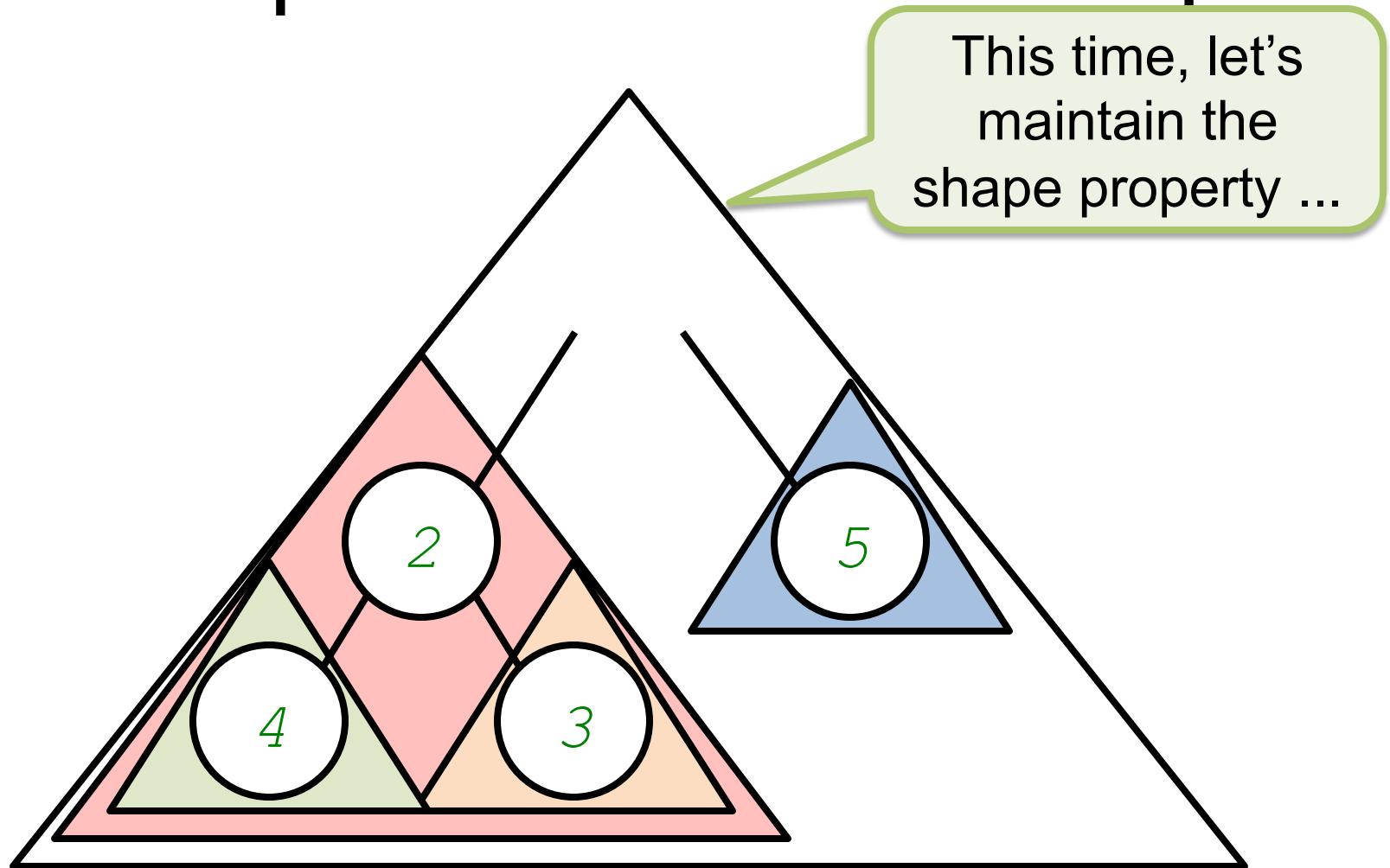
# Example: A First Attempt



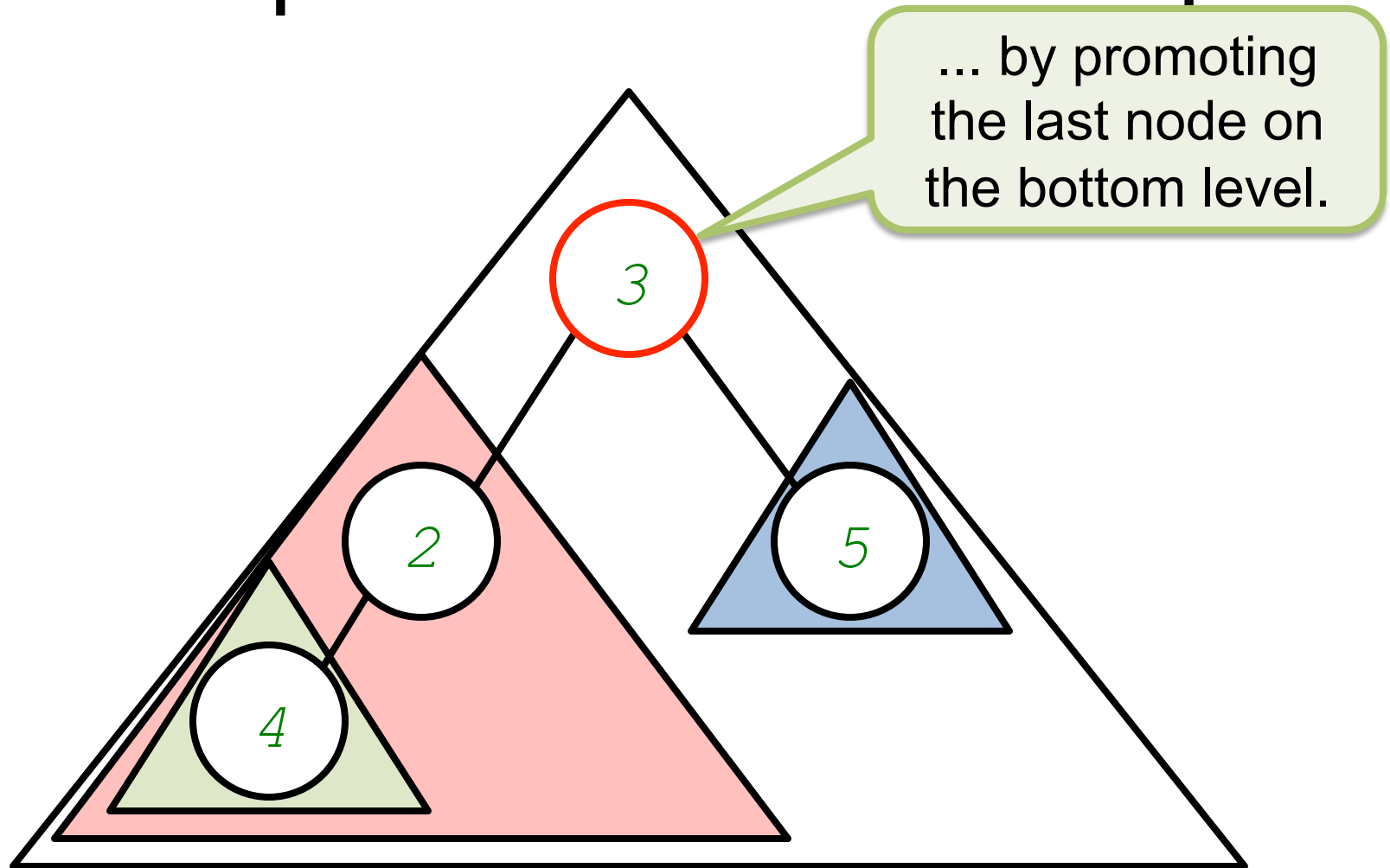
# Example: A Second Attempt



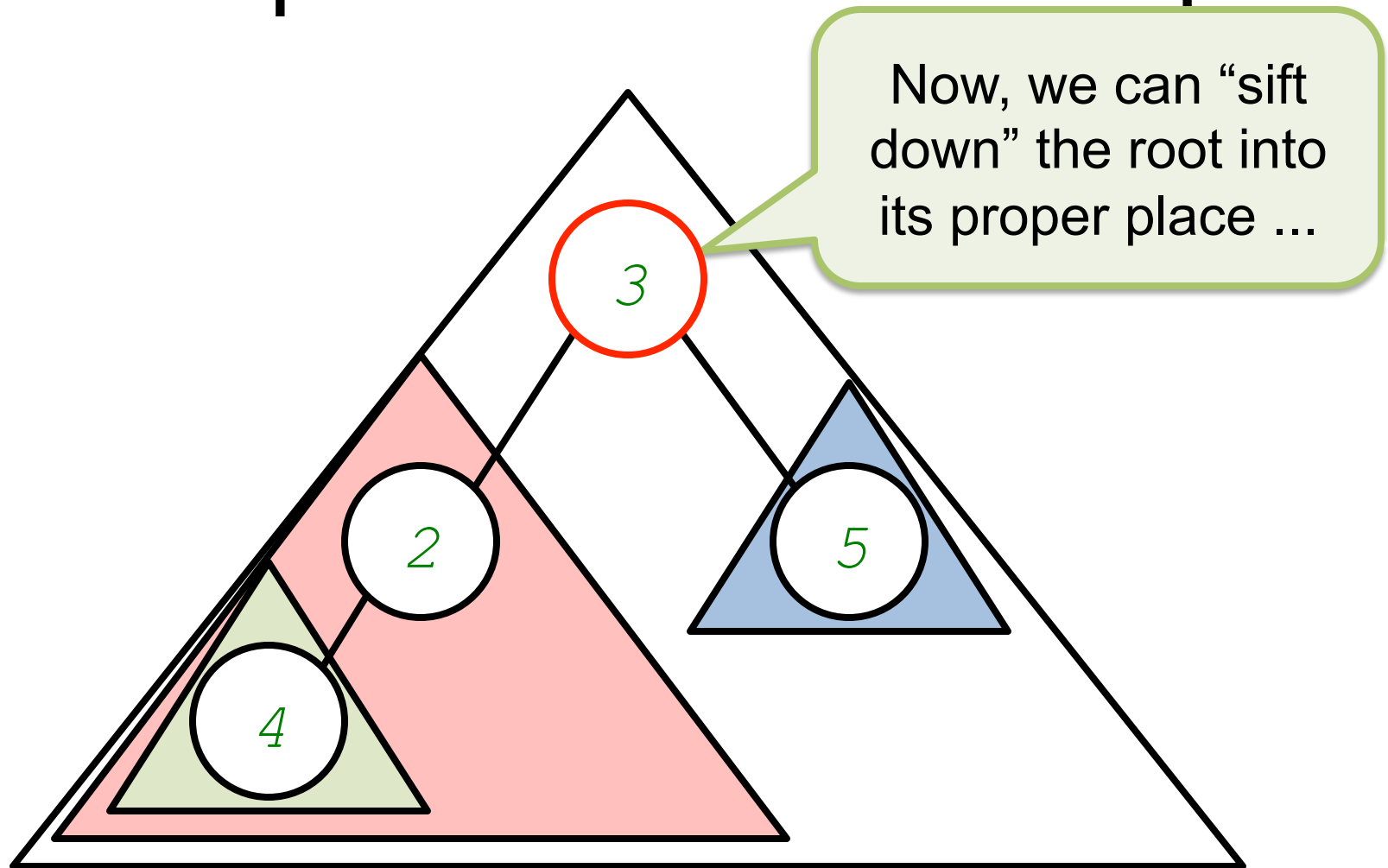
# Example: A Second Attempt



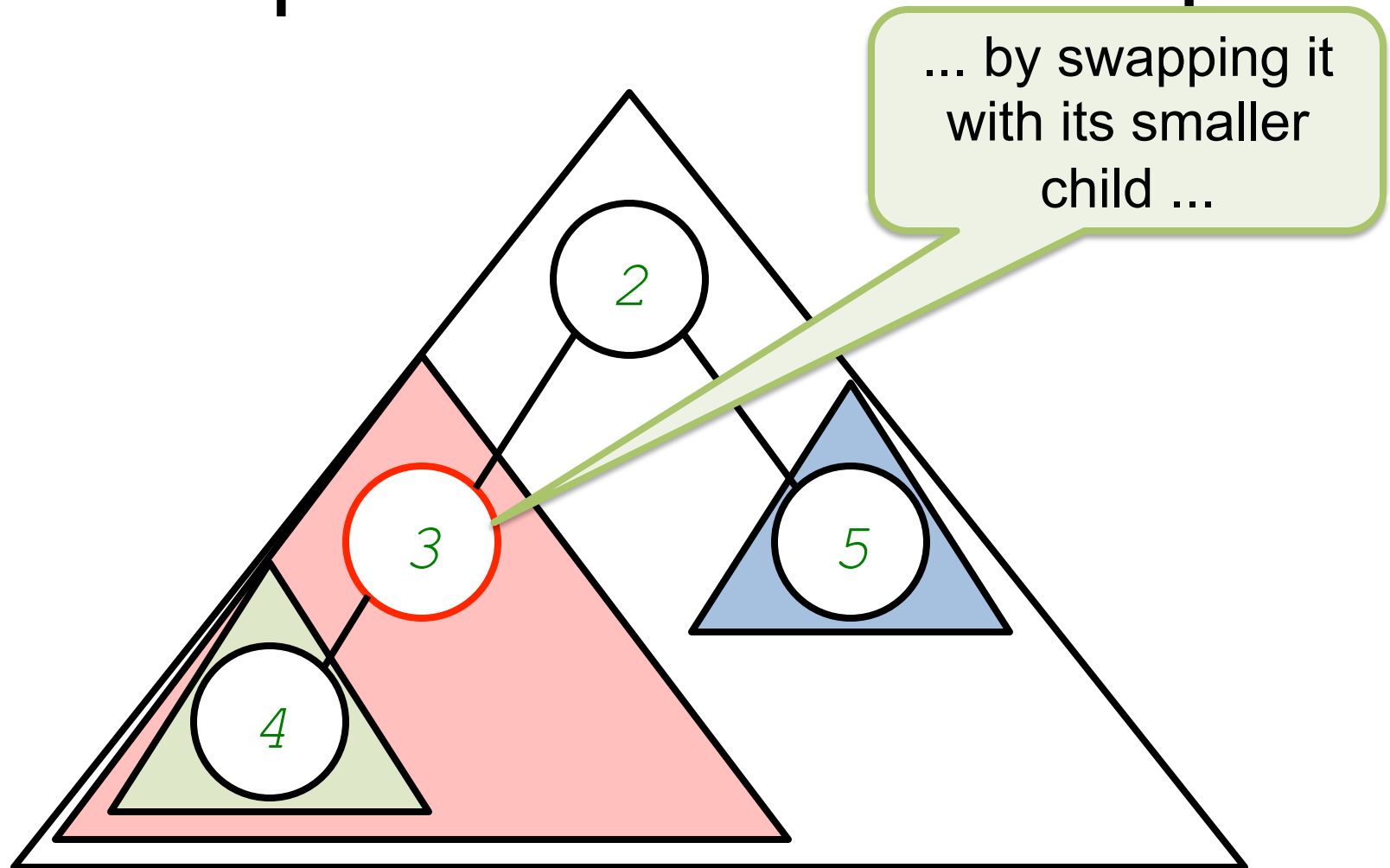
# Example: A Second Attempt



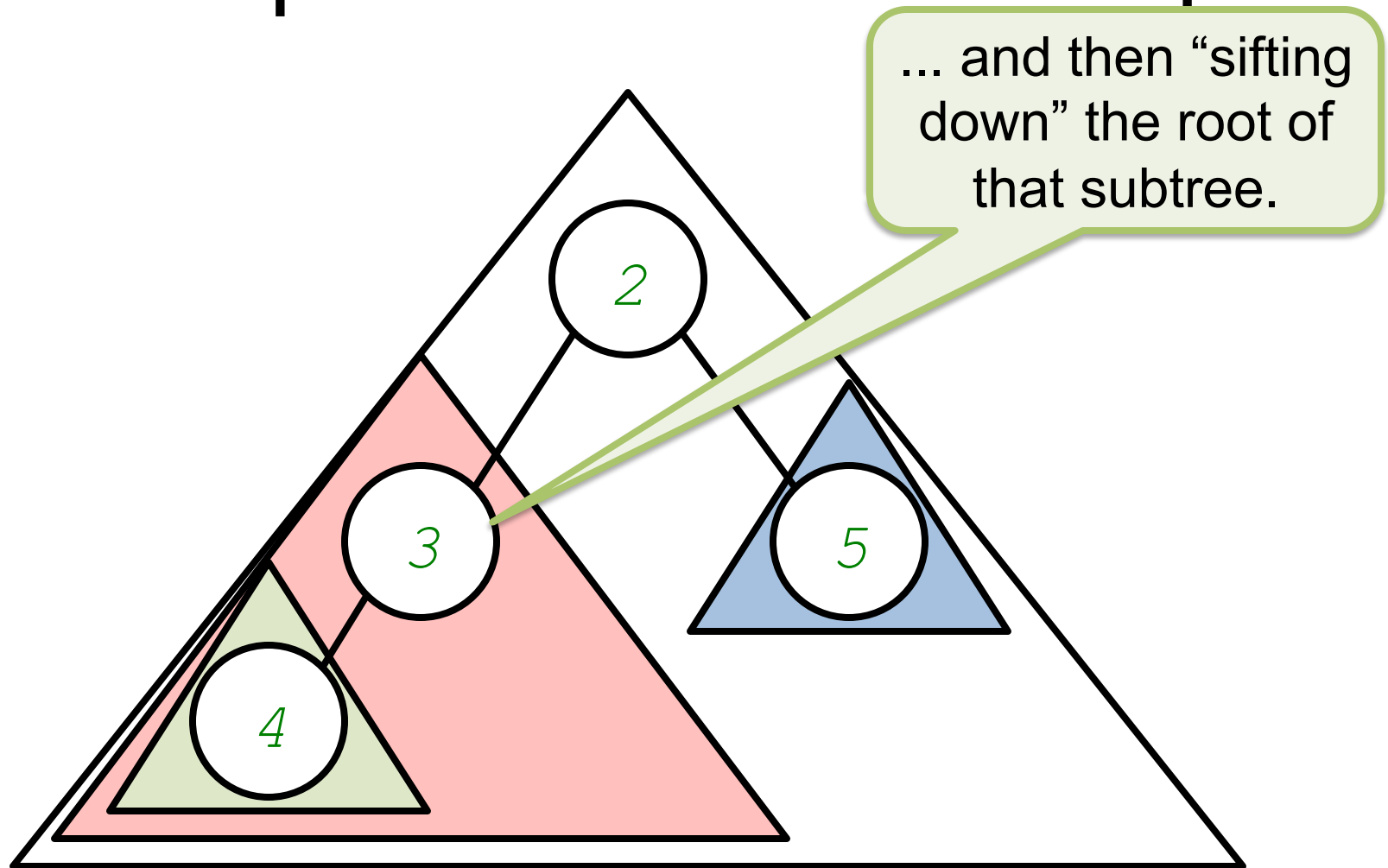
# Example: A Second Attempt



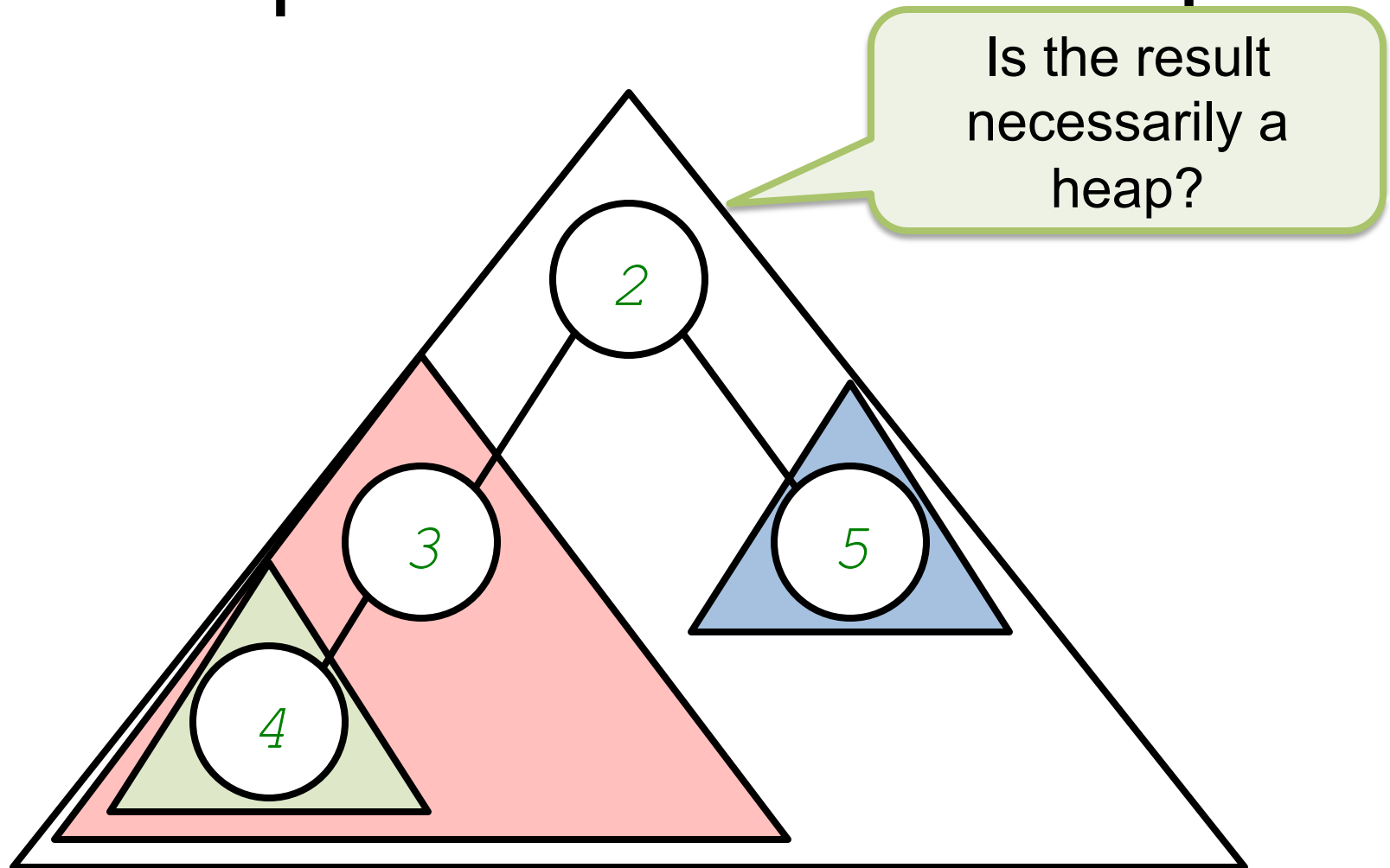
# Example: A Second Attempt



# Example: A Second Attempt



# Example: A Second Attempt





# Pseudo-Contract

```
/**
 * Restores a complete binary tree to be a heap
 * if only the root might be out of place.
 * @updates t
 * @requires
 *   [t is a complete binary tree] and
 *   [both subtrees of the root of t are heaps]
 * @ensures
 *   [t is a heap with the same values as #t]
 */
public static void siftDown (BinaryTree<T> t)
{ ... }
```

# Building a Heap In the First Place

- Suppose we have  $n$  values in a complete binary tree, but they are arranged without regard to the heap ordering
- How can we “heapify” it?

# Pseudo-Contract

```
/**  
 * Makes a complete binary tree into a heap.  
 * @updates t  
 * @requires  
 * [t is a complete binary tree]  
 * @ensures  
 * [t is a heap with the same values as #t]  
 */  
public static void heapify (BinaryTree<T> t)  
{ ... }
```

# Hint

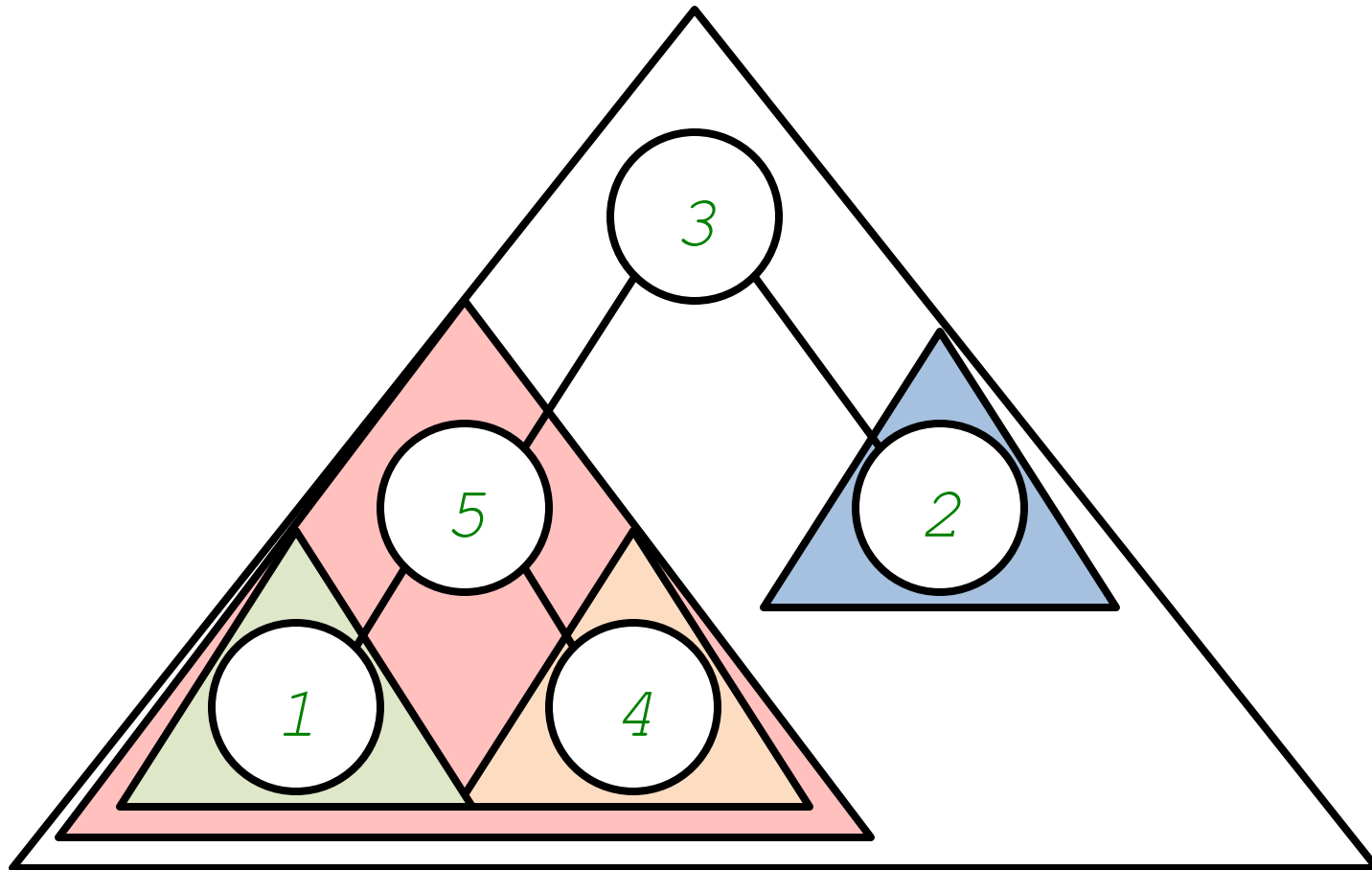
- To see how you might implement `heapify`, compare the contracts of `siftDown` and `heapify`
- The only difference: before we can call `siftDown` to make a heap, both subtrees of the root must already be heaps
  - Once they are heaps, just a call to `siftDown` will finish the job

# Hint

How do we make the subtrees into heaps?

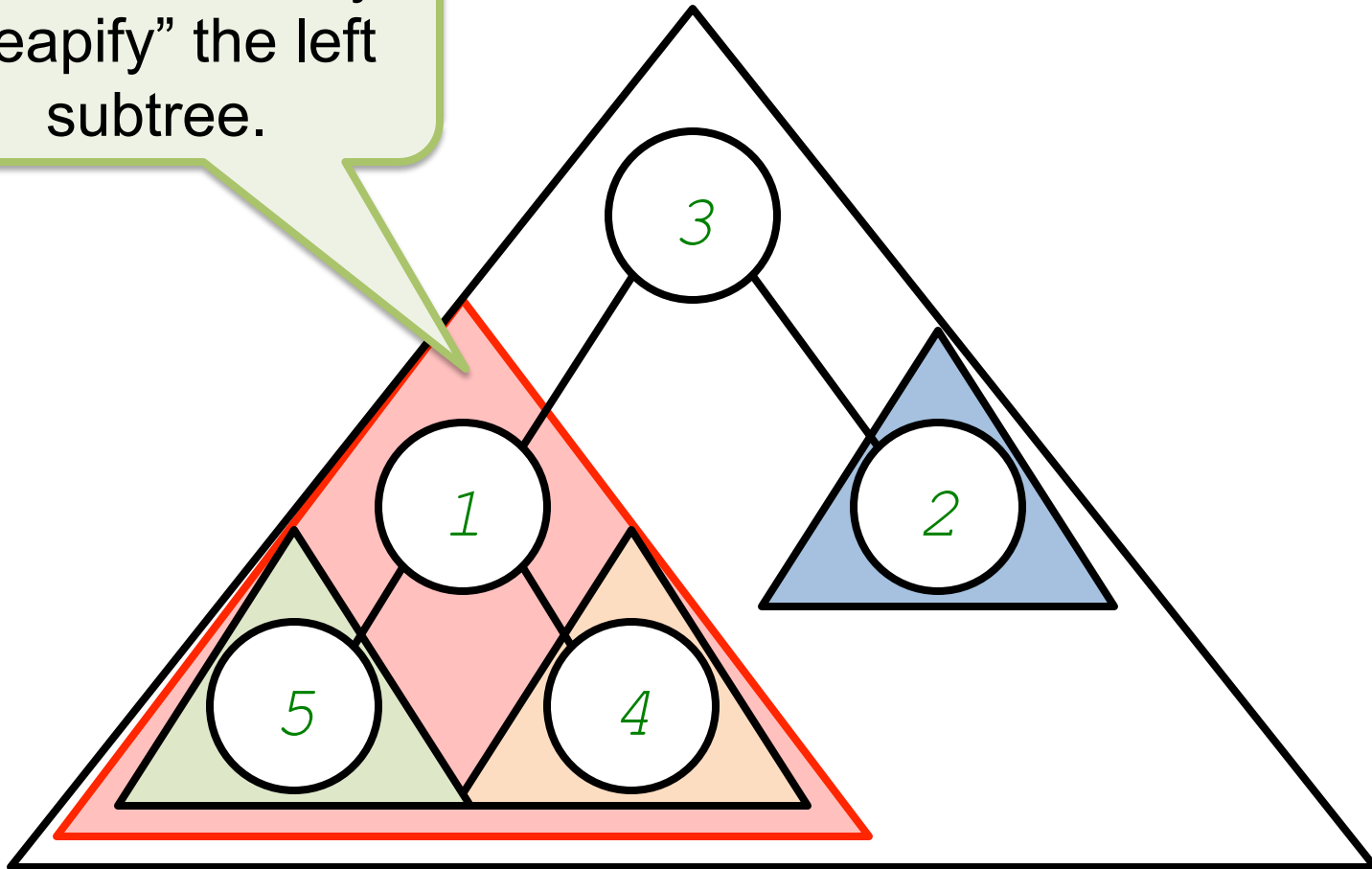
- To see how you might implement `heapify`, compare the contracts of `siftDown` and `heapify`
- The only difference: before we can call `siftDown` to make a heap, both subtrees of the root must already be heaps
  - Once they are heaps, just a call to `siftDown` will finish the job

# Example

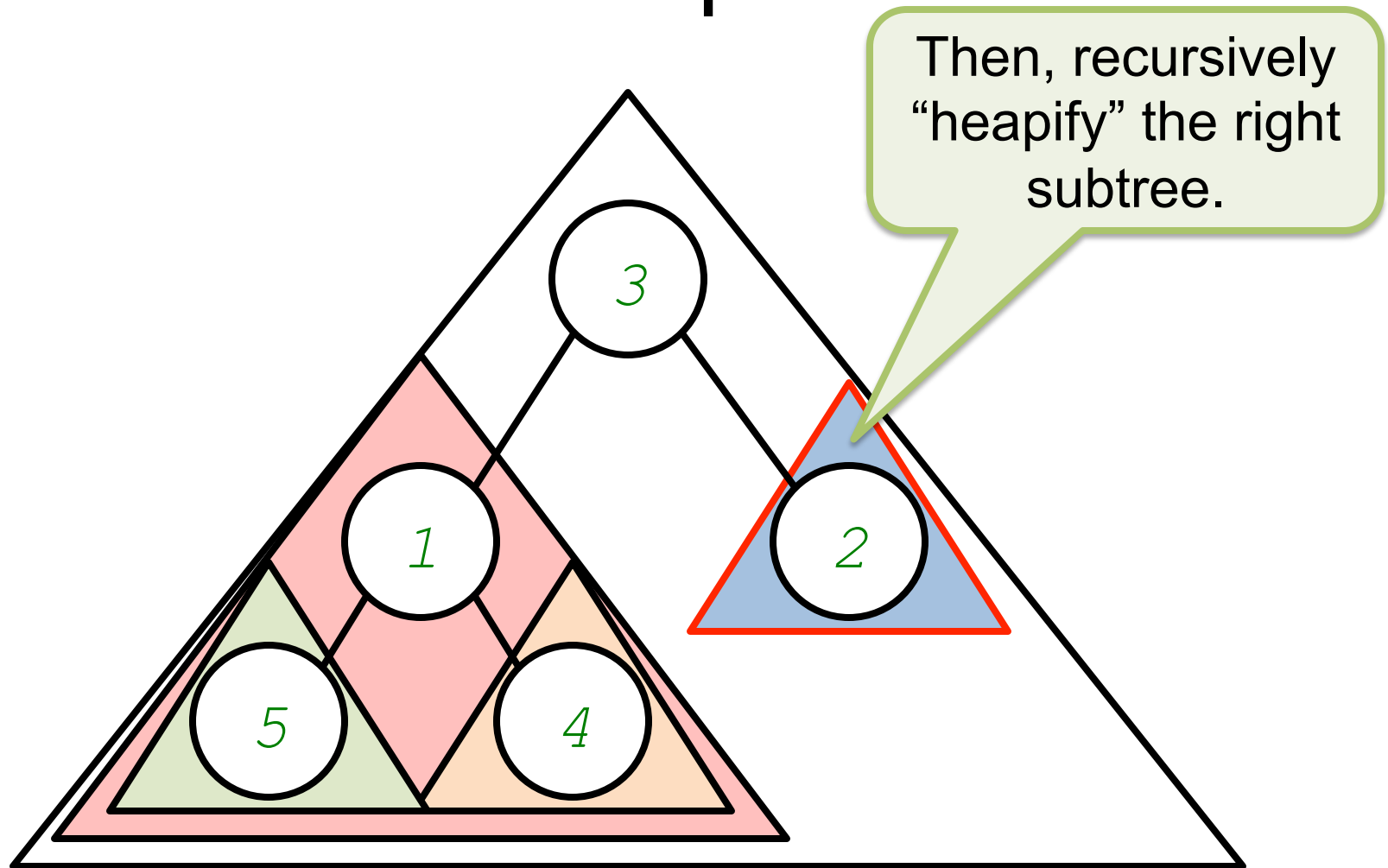


# Example

First, recursively  
“heapify” the left  
subtree.

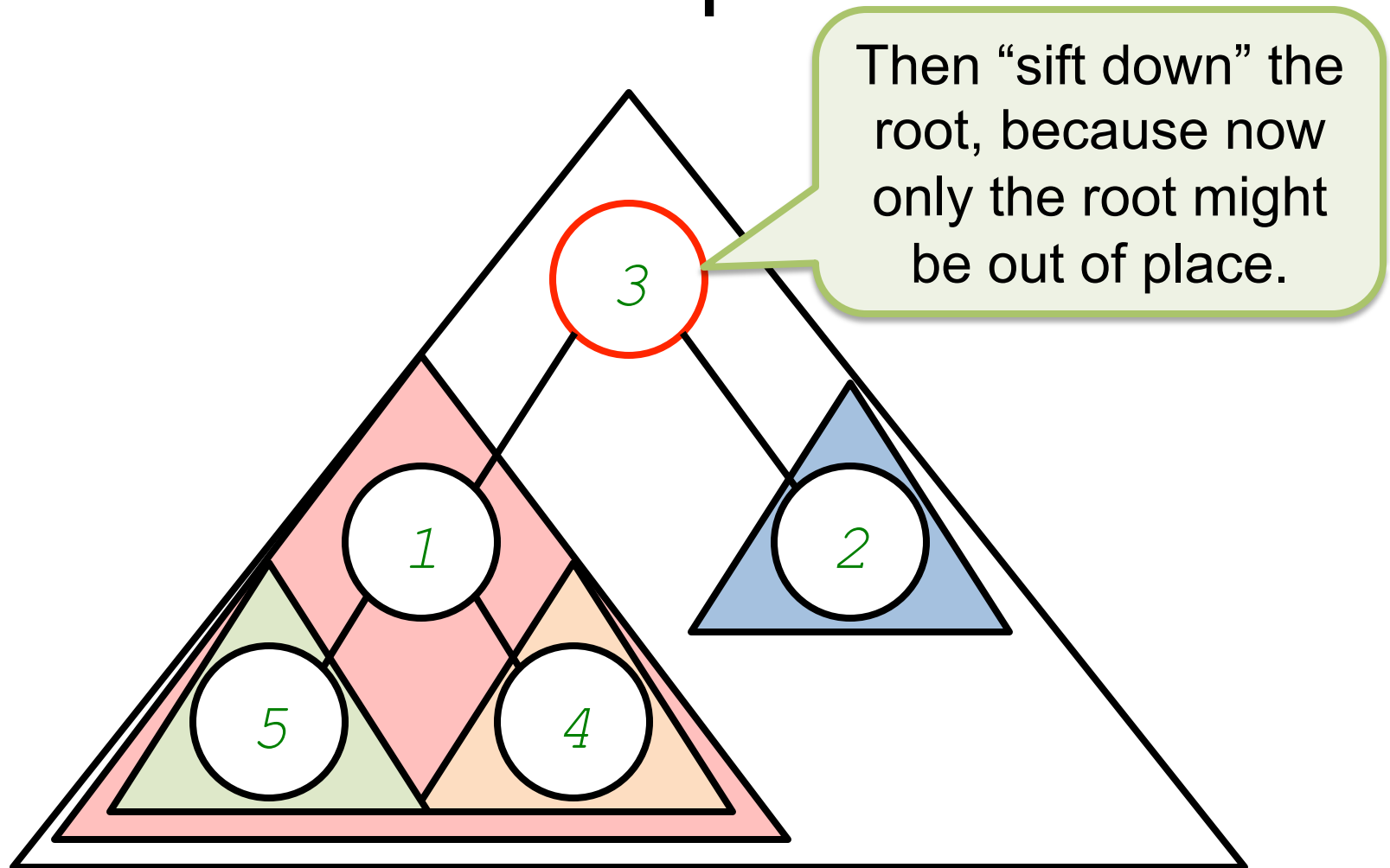


# Example





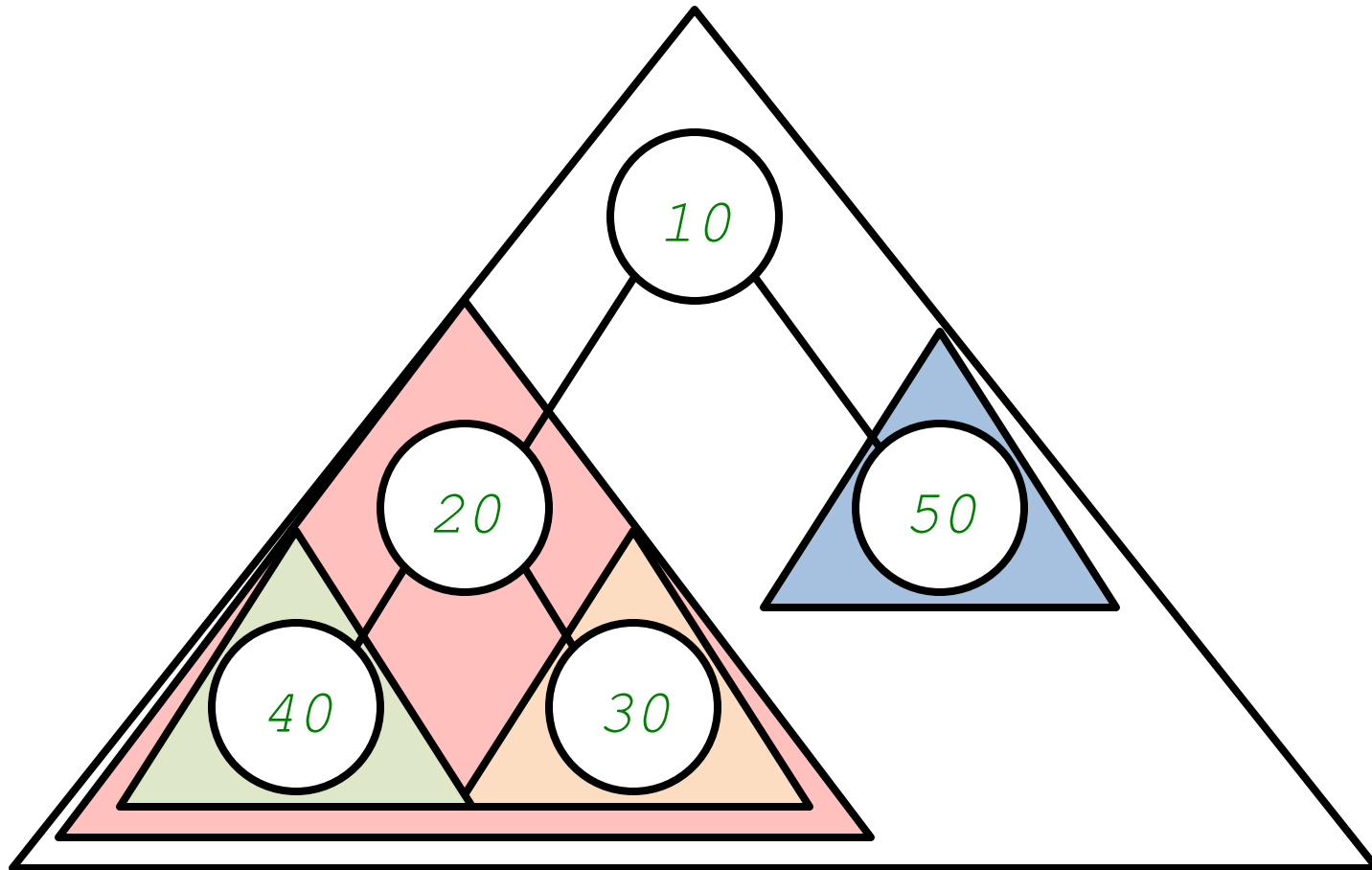
# Example



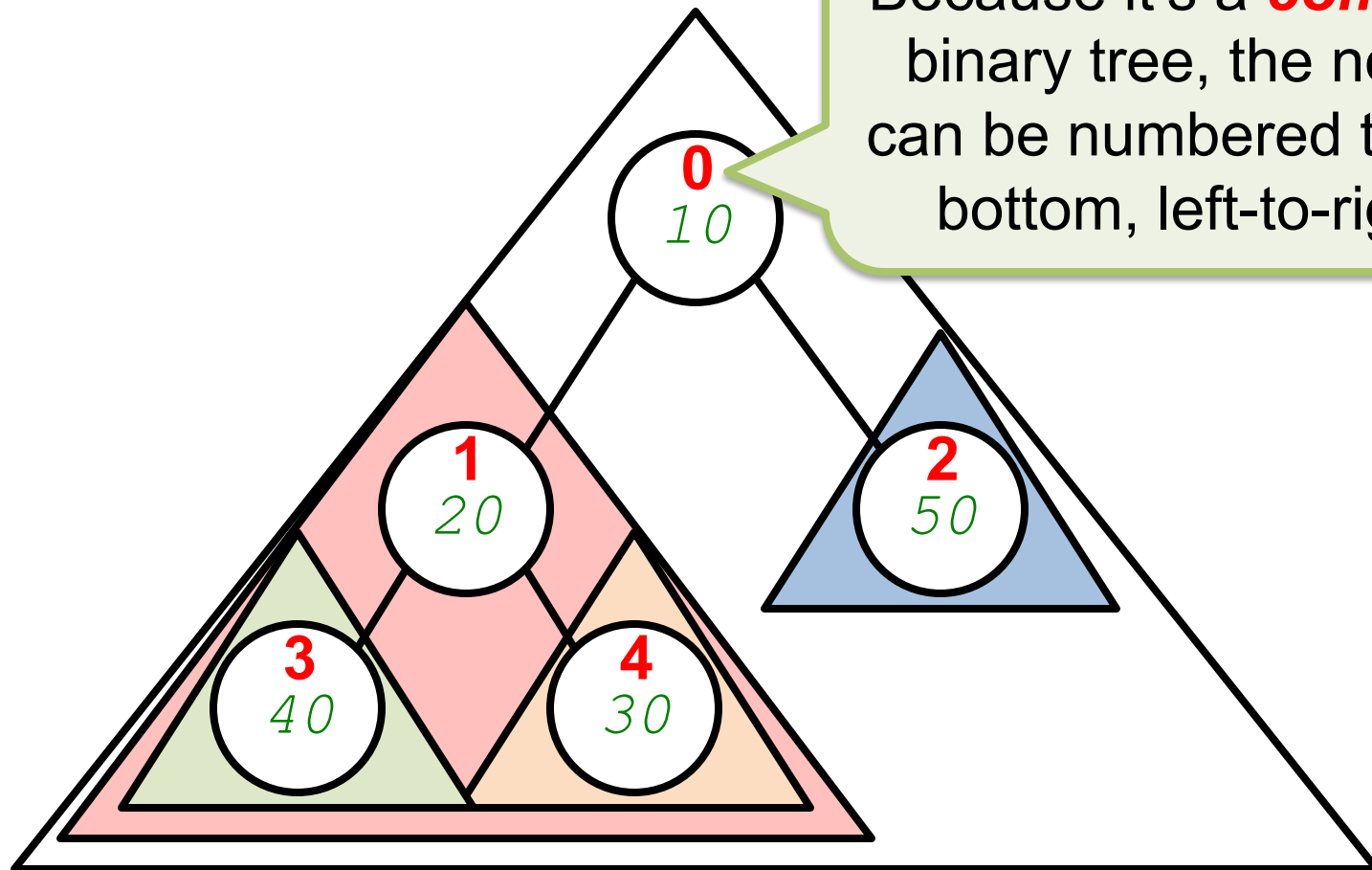
# Embedding a Heap in an `Array`

- While one could represent a heap using a `BinaryTree<T>` (as suggested in the pseudo-contracts above), it is generally not done this way
- Instead, a heap is usually represented “compactly” using an `Array<T>`

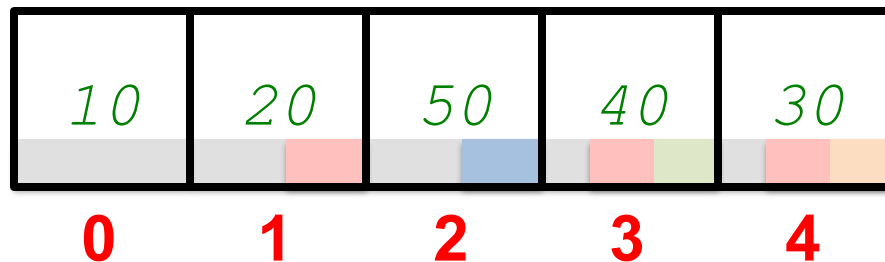
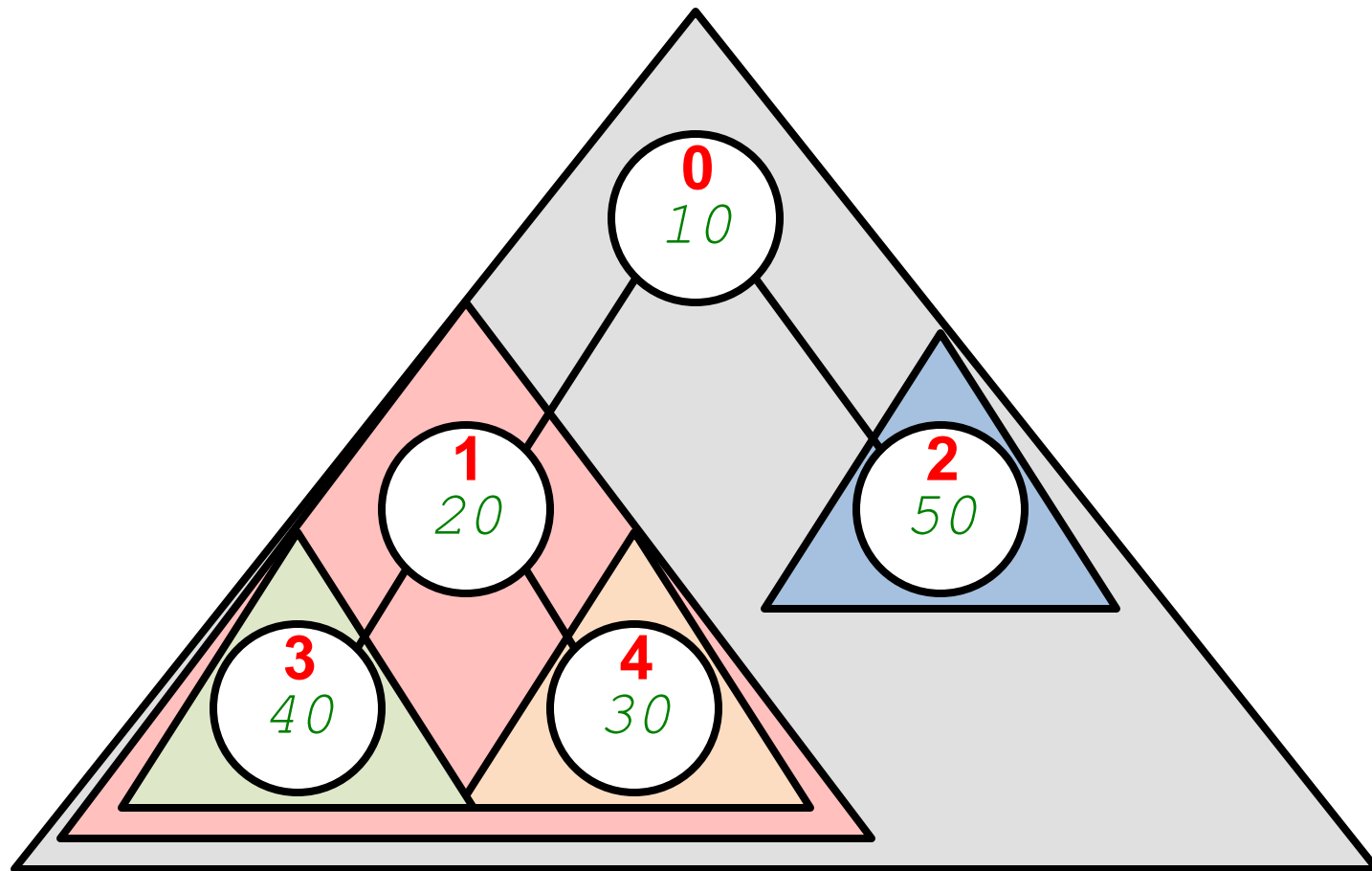
# Interpreting an `Array` as a Heap



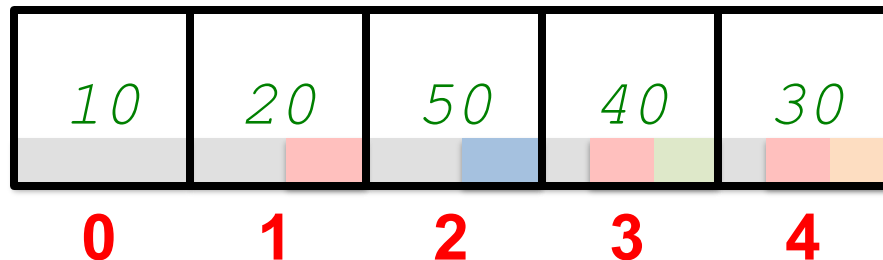
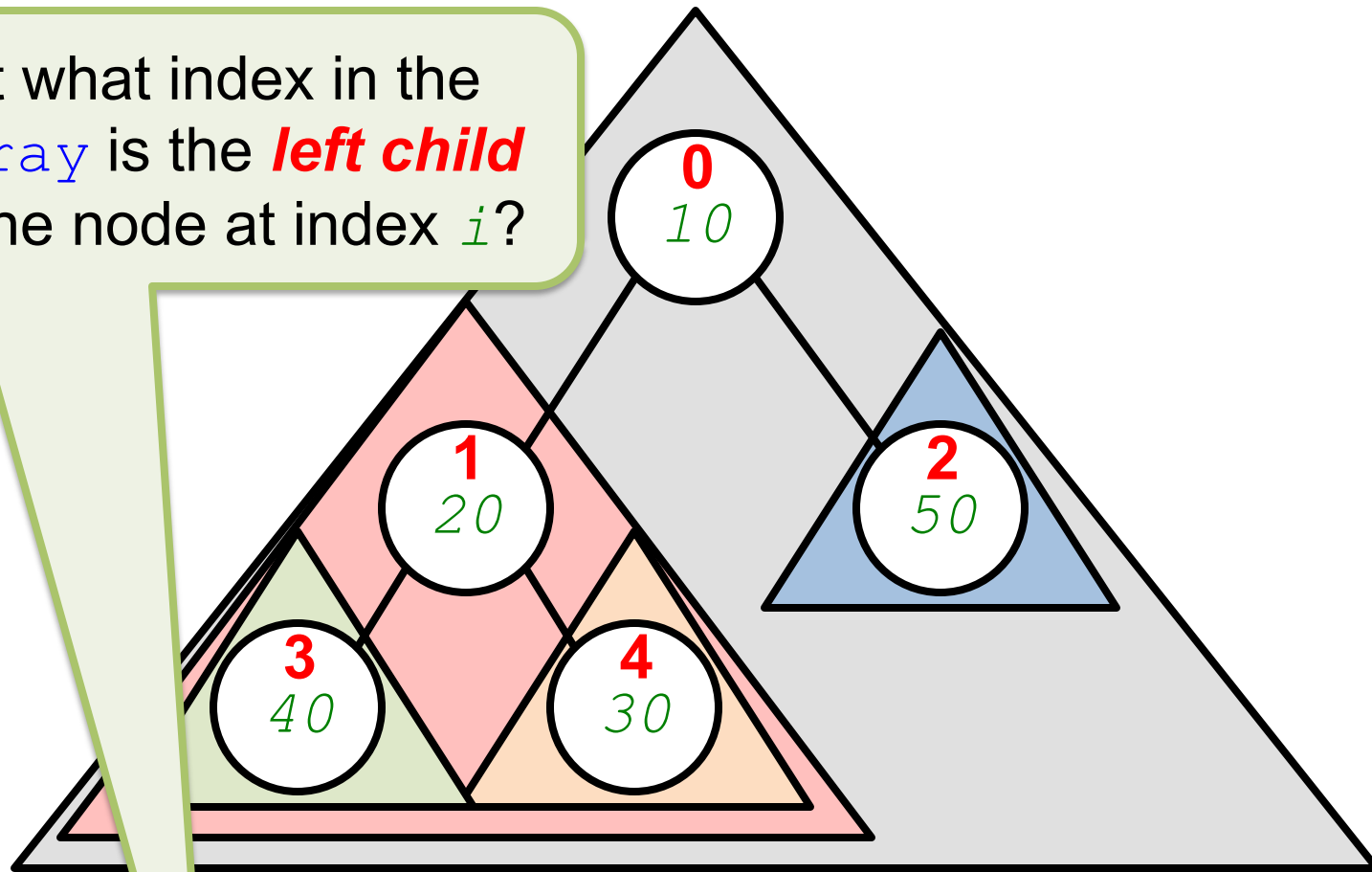
# Interpreting an *Array* as a Heap



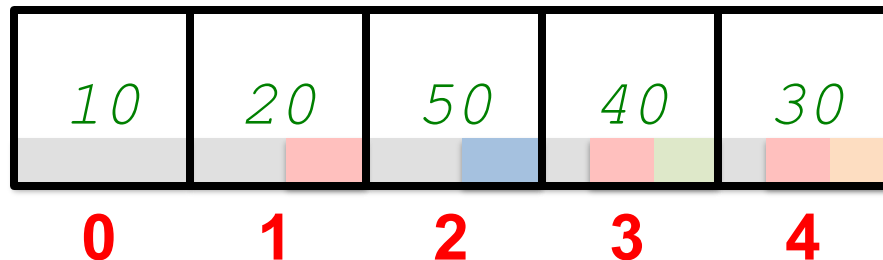
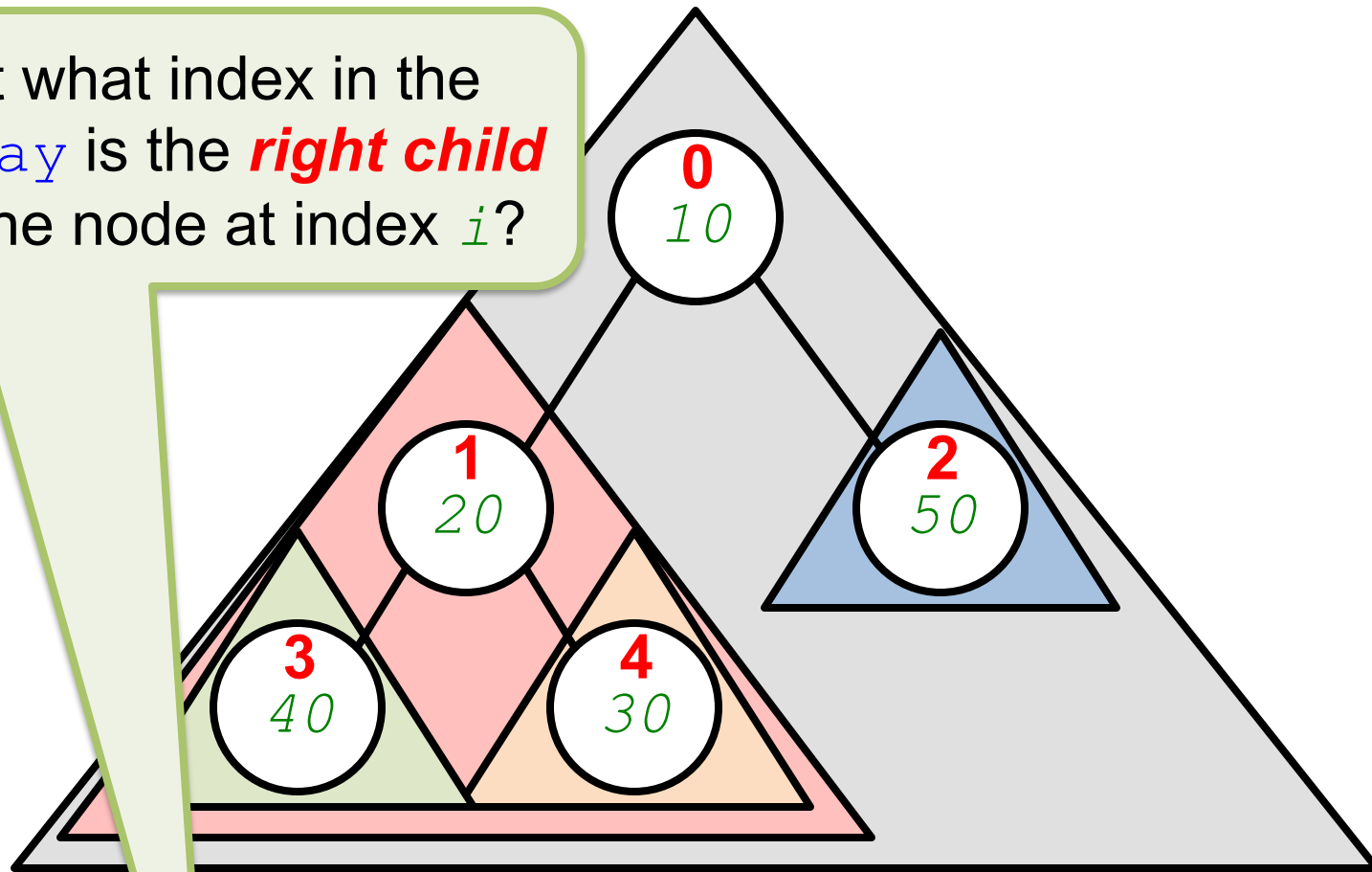
Because it's a **complete** binary tree, the nodes can be numbered top-to-bottom, left-to-right.



At what index in the  
*Array* is the **left child**  
of the node at index  $i$ ?



At what index in the  
Array is the **right child**  
of the node at index  $i$ ?



# Resources

- Wikipedia: Heapsort
  - <http://en.wikipedia.org/wiki/Heapsort>
- Wikipedia: Heap (data structure)
  - [http://en.wikipedia.org/wiki/Heap\\_\(data\\_structure\)](http://en.wikipedia.org/wiki/Heap_(data_structure))
- *Big Java Late Objects*, Section 17.6
  - <http://proquest.safaribooksonline.com.proxy.lib.ohio-state.edu/book/programming/java/9781118087886/chapter-17-tree-structures/navpoint-139>
- *Big Java Late Objects*, Section 17.7
  - <http://proquest.safaribooksonline.com.proxy.lib.ohio-state.edu/book/programming/java/9781118087886/chapter-17-tree-structures/navpoint-140>