Revised: 6/6/2020

Heaps

Michael C. Hackett

Assistant Professor, Computer Science

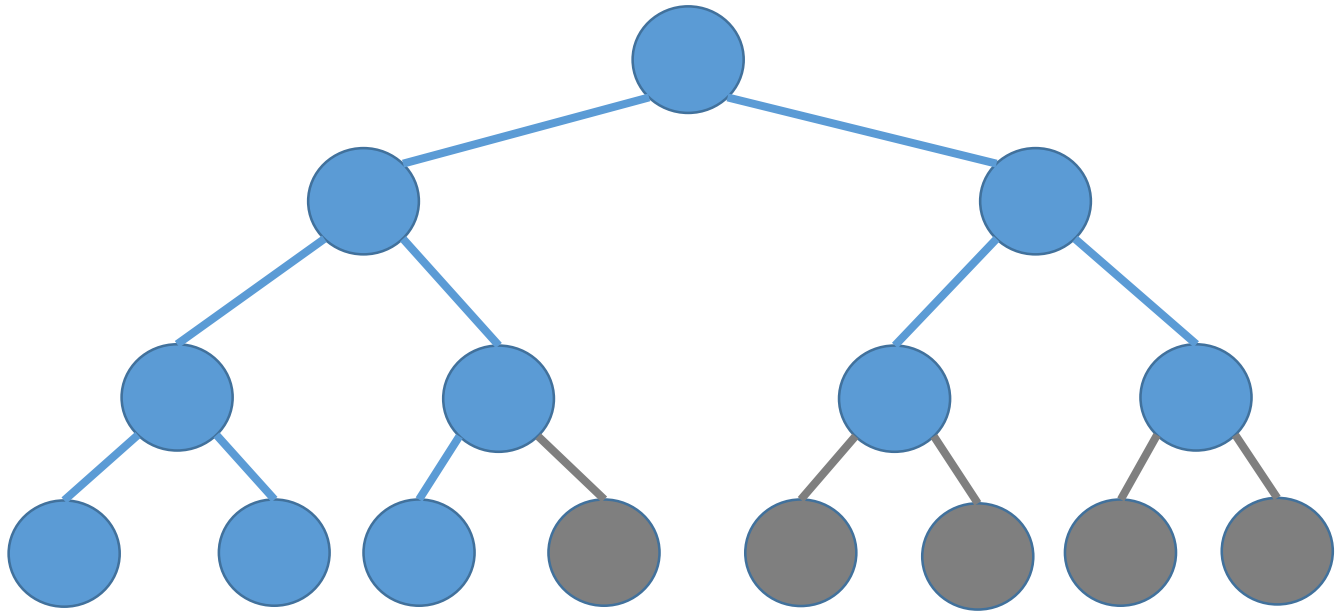
Community College of Philadelphia

Lecture Topics

- Heaps
  - Insertion/Bubble Up
  - Removal/Percolate Down
- Implementation
- Complexity
- Max-Heaps
- Priority Queues
- Heapsorting

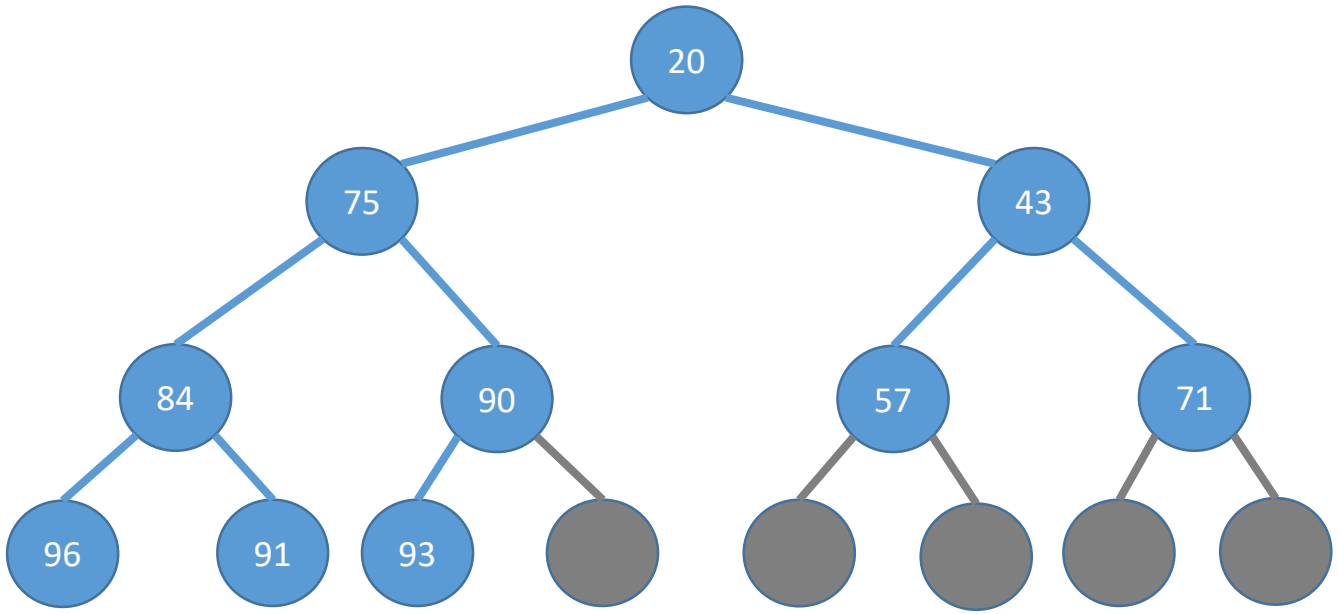
Heaps

- A **heap** is a non-linear data structure, specifically a binary tree, with two properties:
  1. It is complete: All nodes are filled in, except the last level which has all any leaves on the left side.



Heaps

- A **heap** is a non-linear data structure, specifically a binary tree, with two properties:
  2. Each node's value is no larger than the values of all its descendants



Heaps

- Heaps first appear to be similar to BSTs but there are two big differences:
  - The shape is always balanced
  - A node's left and right subtrees both contain values *greater than* the node's value.
    - Recall that nodes in a BST store smaller values in its left subtree and values larger than its own in its right subtree

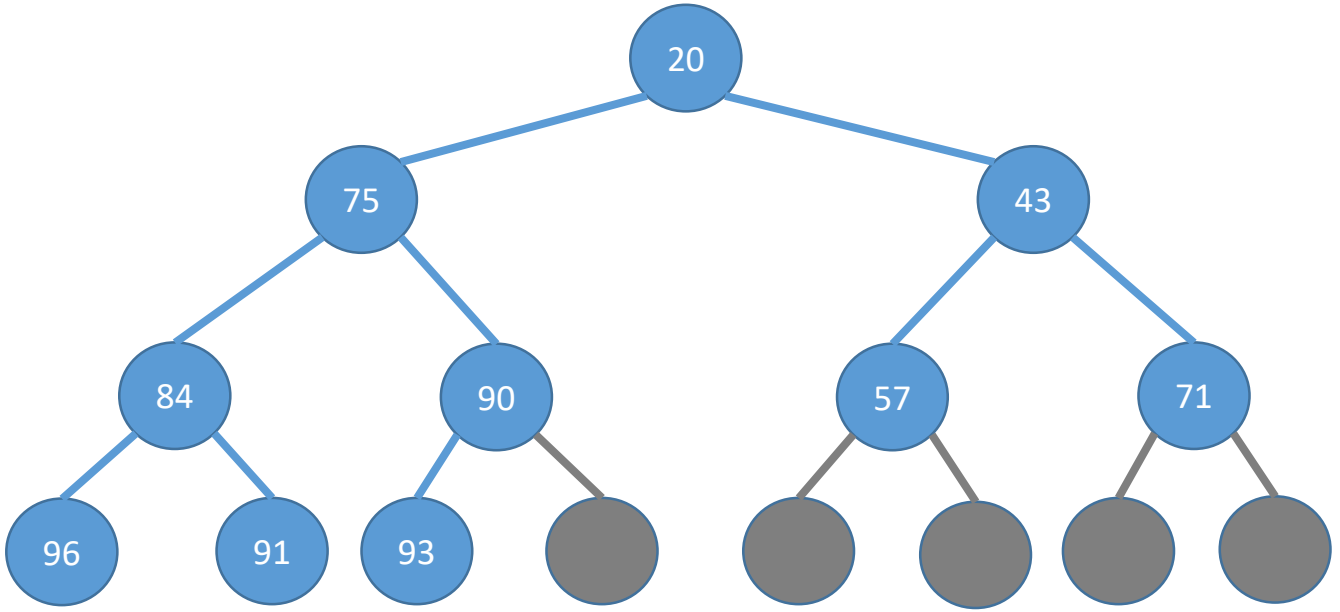
- The root node is the smallest value in the structure.
  - The heap being described here is a ***min-heap***
  - A **max-heap** is the opposite: A node's left and right subtrees contain values *less than* the node's value.

Insertion

- To insert a new value into a heap (3-Step Process):
  1. Add a new node in the next available location
  2. If the value to insert is less than the new node's parent, move the parent's value down into the new node. Repeat until a parent node's value less than the insertion value (or the root node) is reached.
     - Another way to look at this is the value to insert ***bubbles up*** to the node where it belongs.
  3. Once a parent node with a value less than the insertion value (or the root node) is reached, store the new value at this location.

Insertion

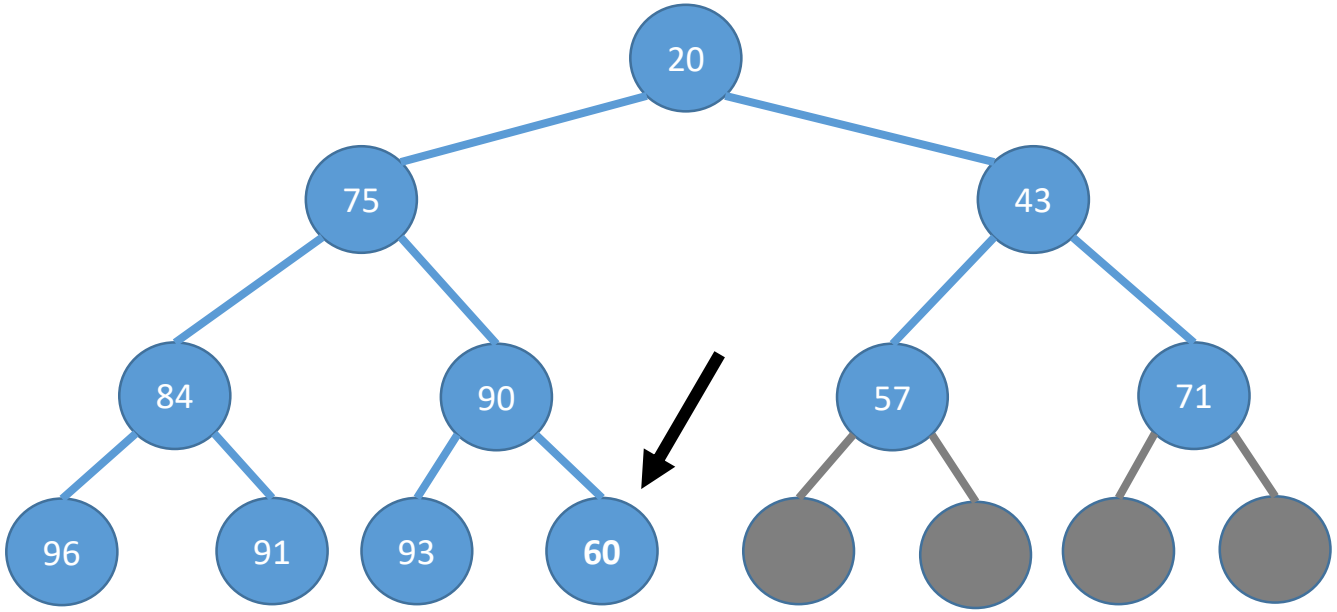
- Value to insert: 60



Insertion

- Value to insert: 60

1. Add a new node in the next available location



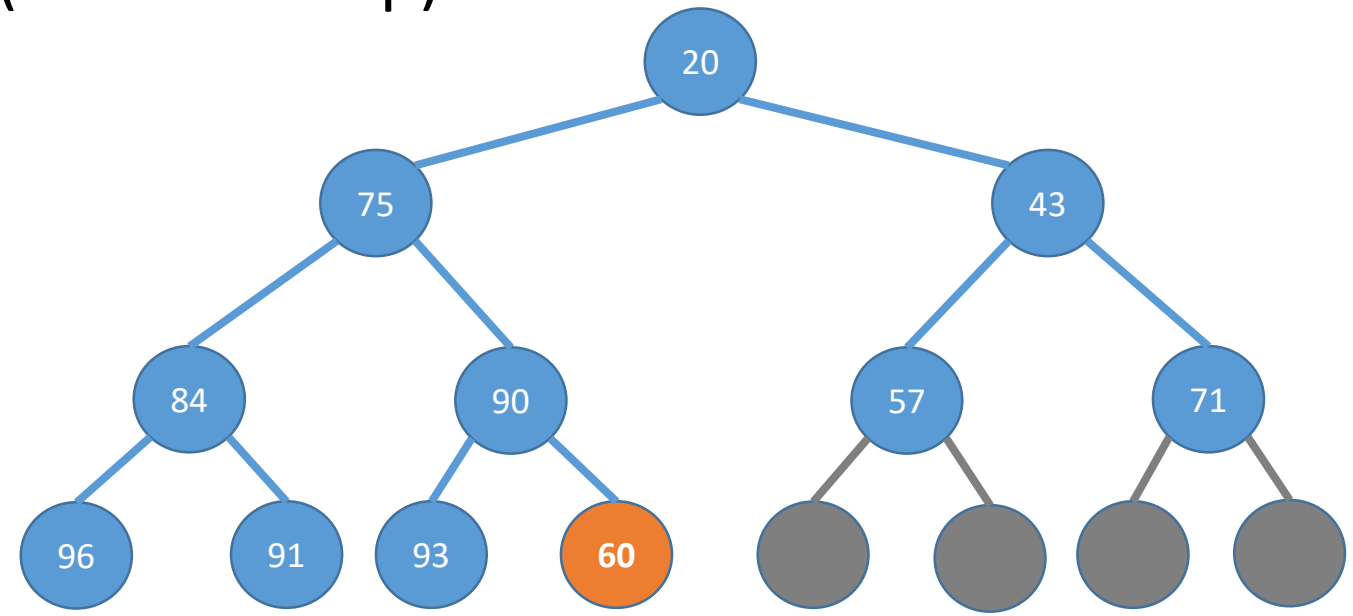
Insertion

- Value to insert: 60

2. If the value to insert is less than the new node's parent, move the parent's value down into the new node. Repeat until a parent node's value less than the insertion value (or the root node) is reached.

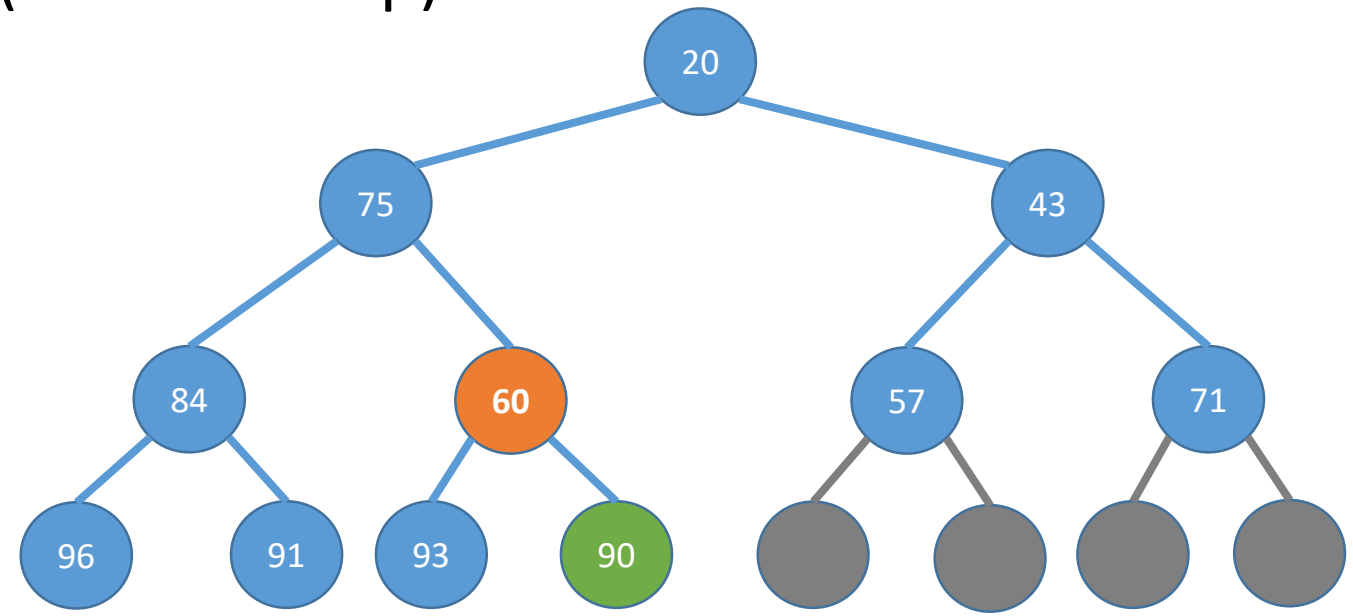
Insertion

- Value to insert: 60
- 90 > 60 (Bubble 60 up)



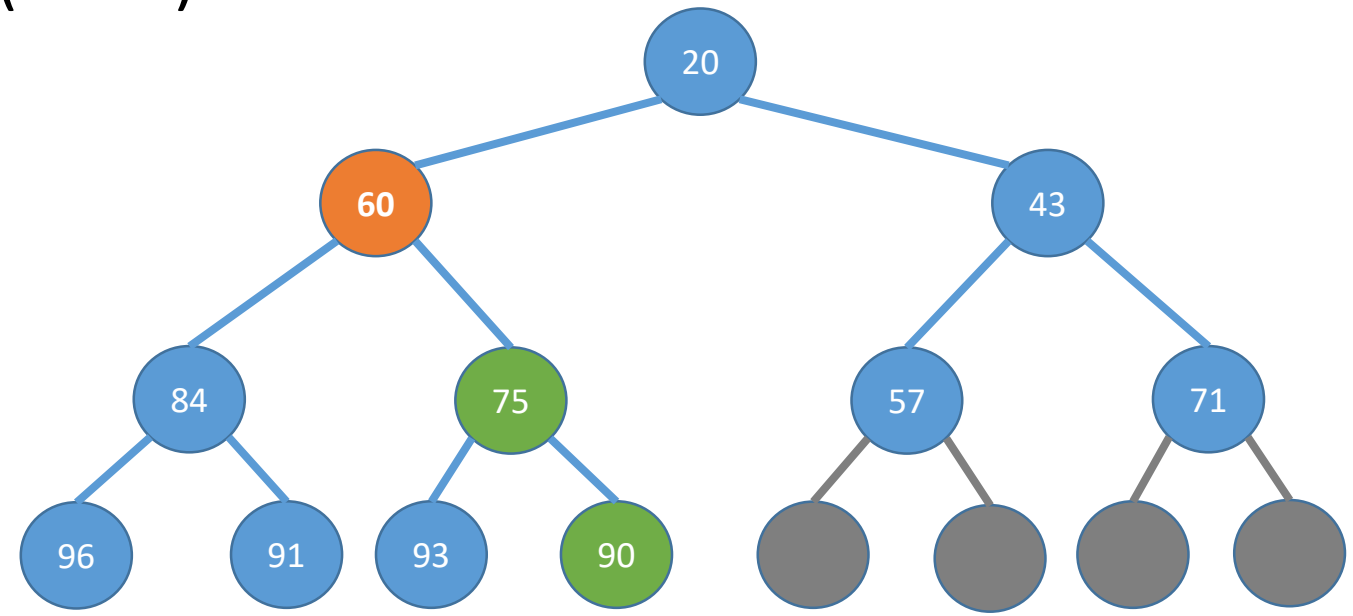
Insertion

- Value to insert: 60
- 75 > 60 (Bubble 60 up)



Insertion

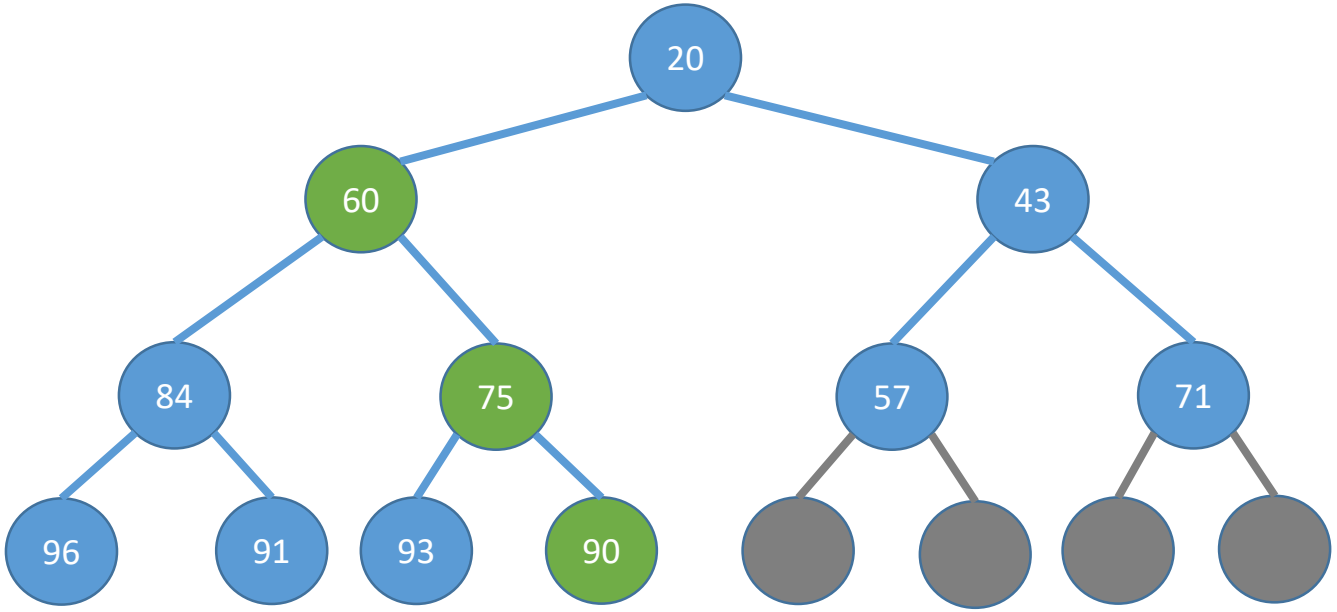
- Value to insert: 60
- 20 > 60 (STOP)



Insertion

- Value to insert: 60

3. Once a parent node with a value less than the insertion value (or the root node) is reached, store the new value at this location.



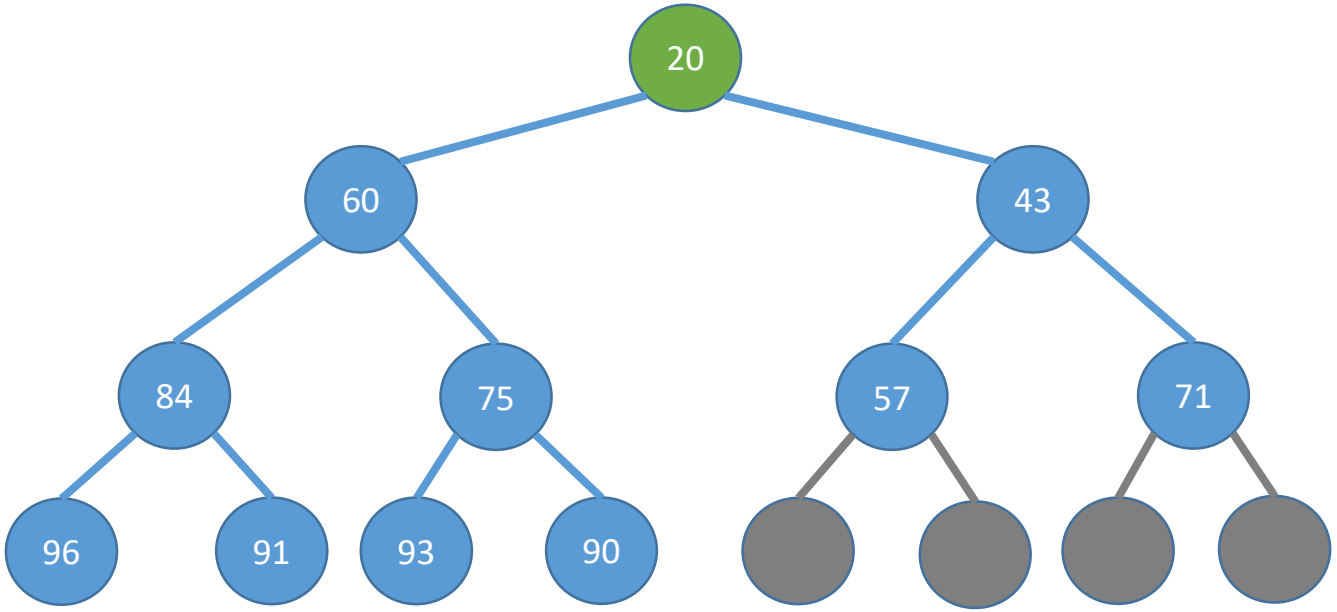
Removal

- We'll only ever remove the root node's value from a heap.
- To remove the root node from a heap (3-Step Process):
  1. Retrieve the root node's value
  2. Move the last node of the heap into the root node and remove the last node.
  3. Promote the smaller of the root node's children and move the existing value down. Repeat until the values of both children are greater than this value (or the last node of a subtree is reached).
     - Another way to look at this is the value ***percolates down*** to the node where it belongs.

Removal

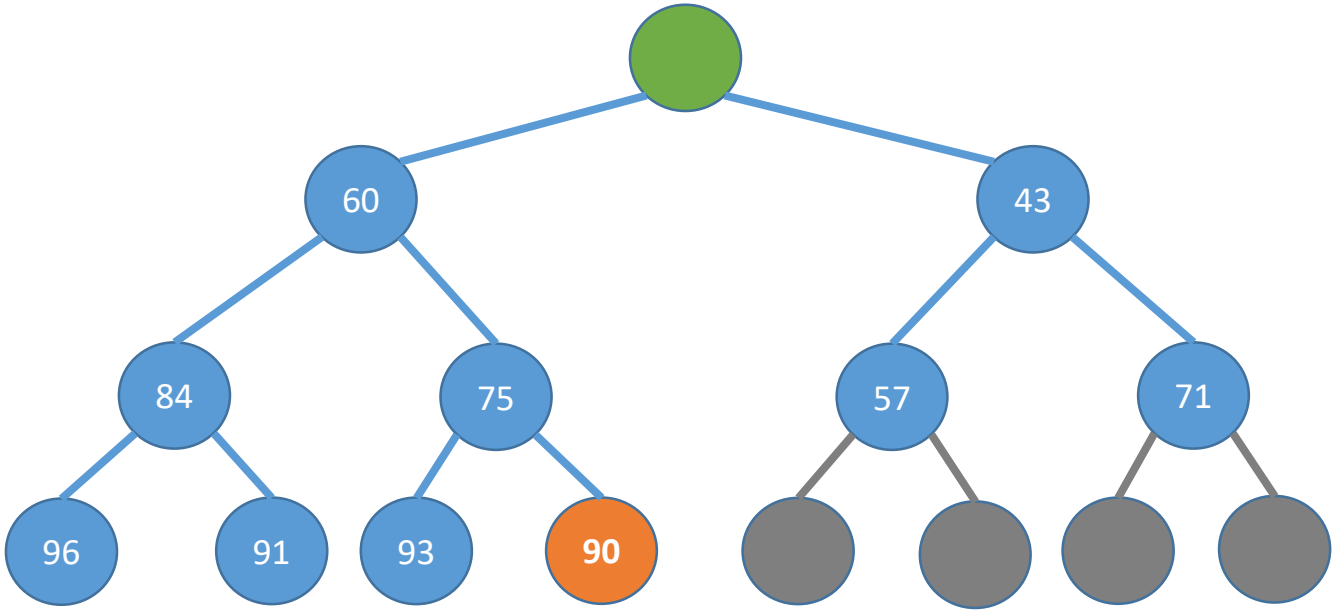
- Value to be removed: 20 (the root node)

1. Retrieve the root node's value



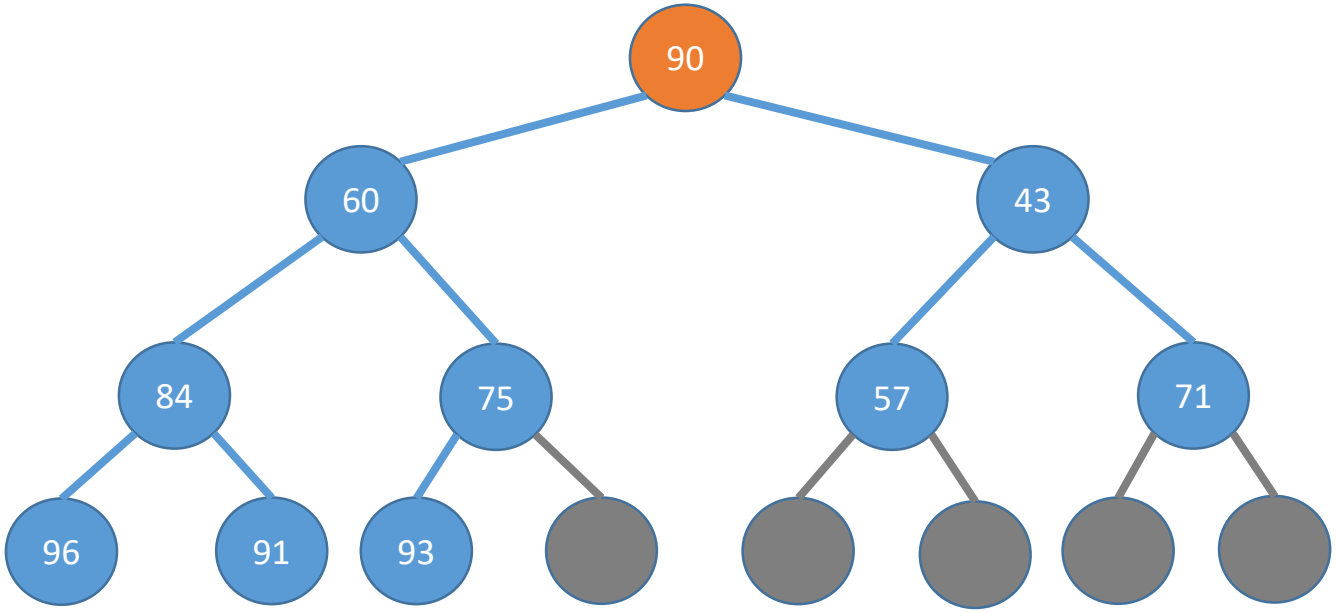
Removal

2. Move the last node of the heap into the root node and remove the last node



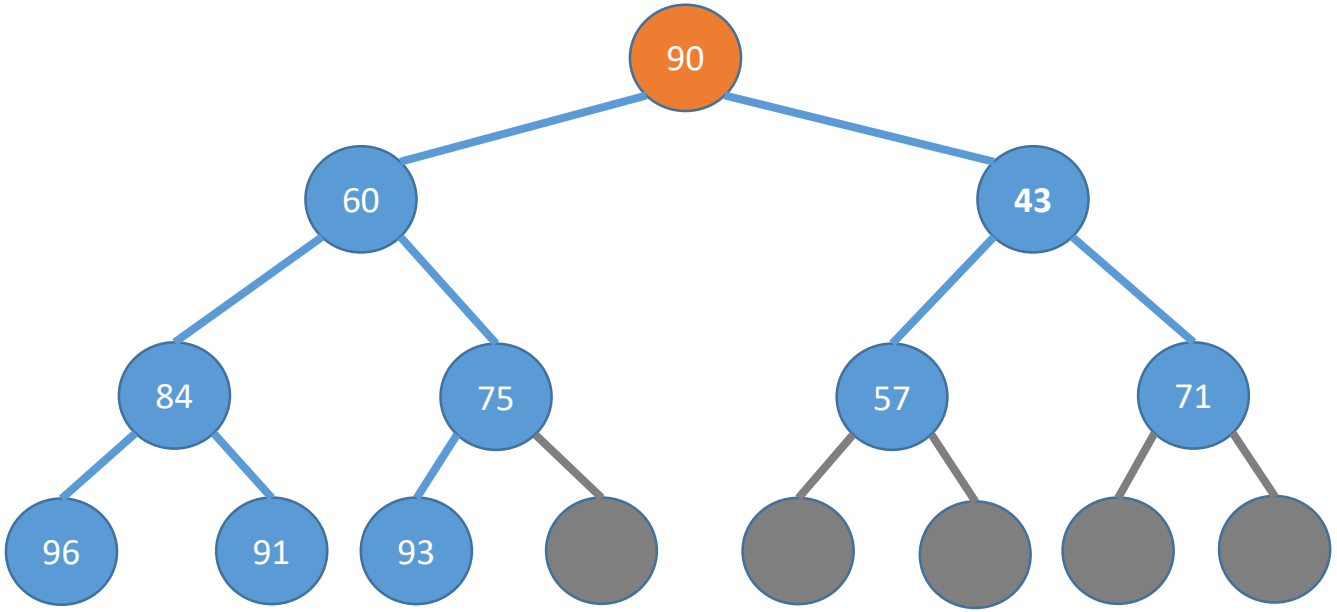
Removal

2. Move the last node of the heap into the root node and remove the last node



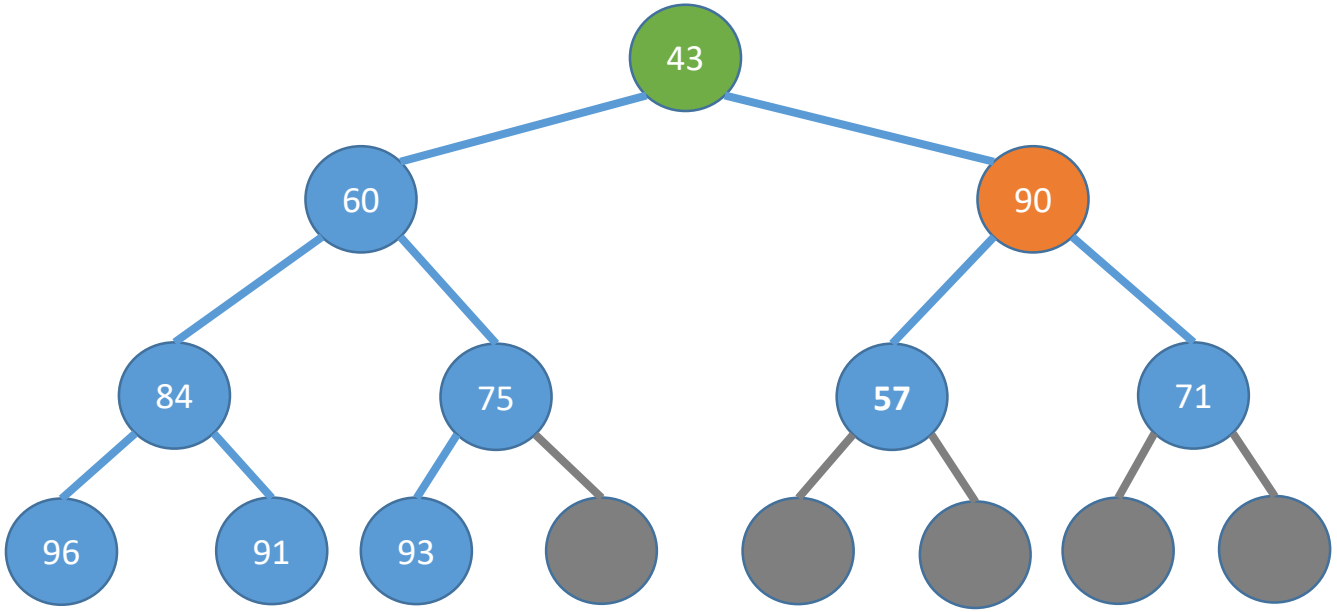
Removal

3. Promote the smaller of the root node's children and move the existing value down. Repeat until the values of both children are greater than this value (or the last node of a subtree is reached).



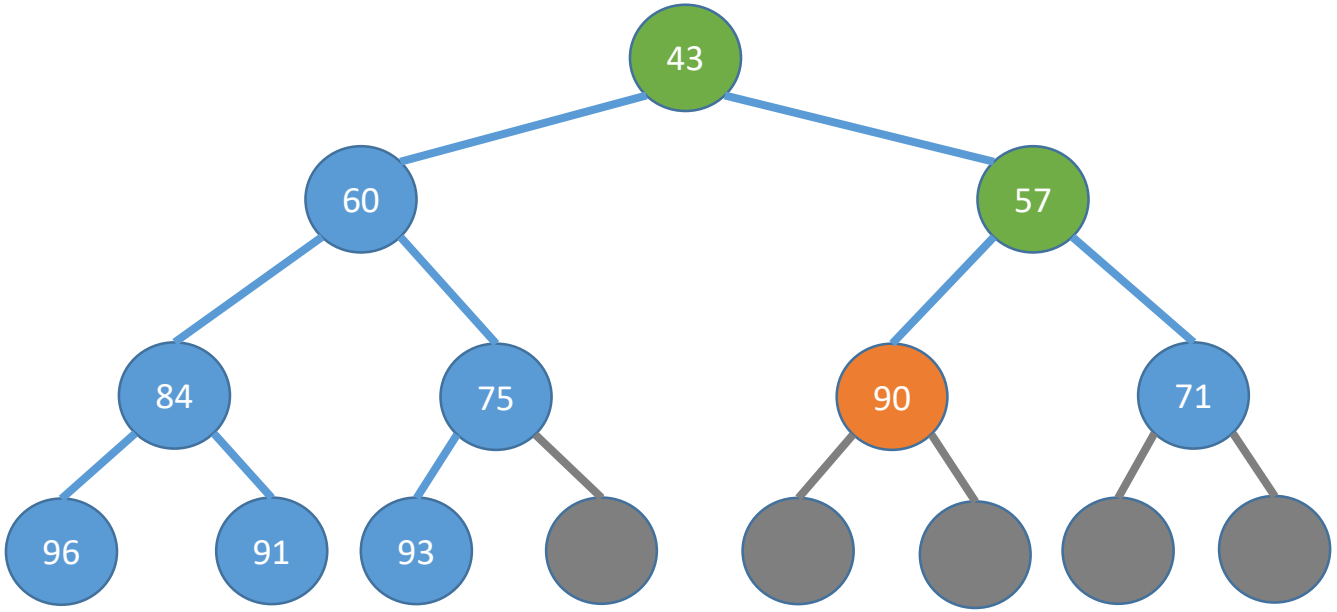
Removal

3. Promote the smaller of the root node's children and move the existing value down. Repeat until the values of both children are greater than this value (or the last node of a subtree is reached).



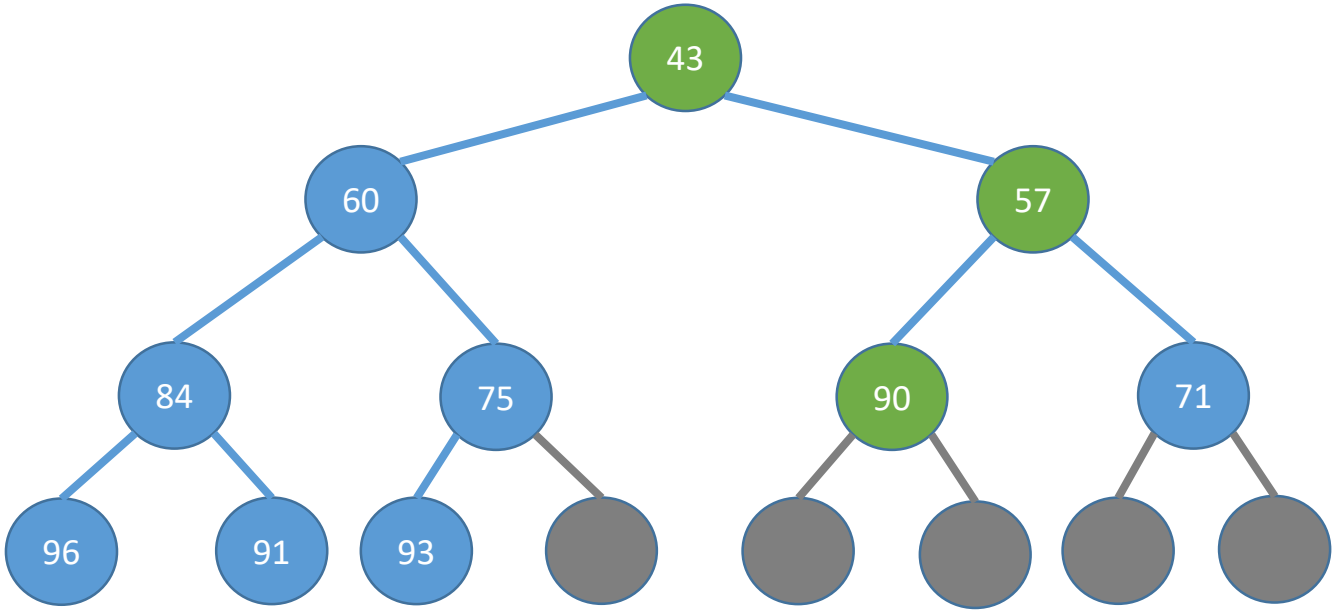
Removal

3. Promote the smaller of the root node's children and move the existing value down. Repeat until the values of both children are greater than this value (or the last node of a subtree is reached).



Removal

3. Promote the smaller of the root node's children and move the existing value down. Repeat until the values of both children are greater than this value (or the last node of a subtree is reached).



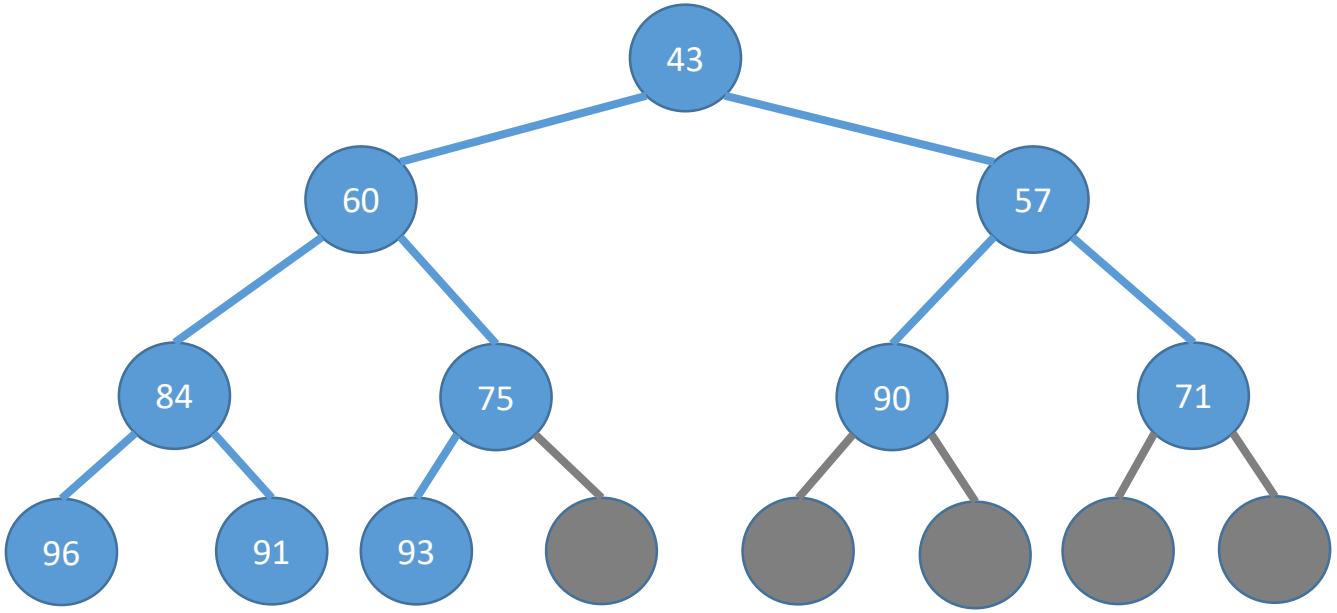
Implementation

- At first glance, it looks like we should implement this structure just as we did with previous binary tree or BST examples.
  - Node objects with left, right, and parent pointers/references.

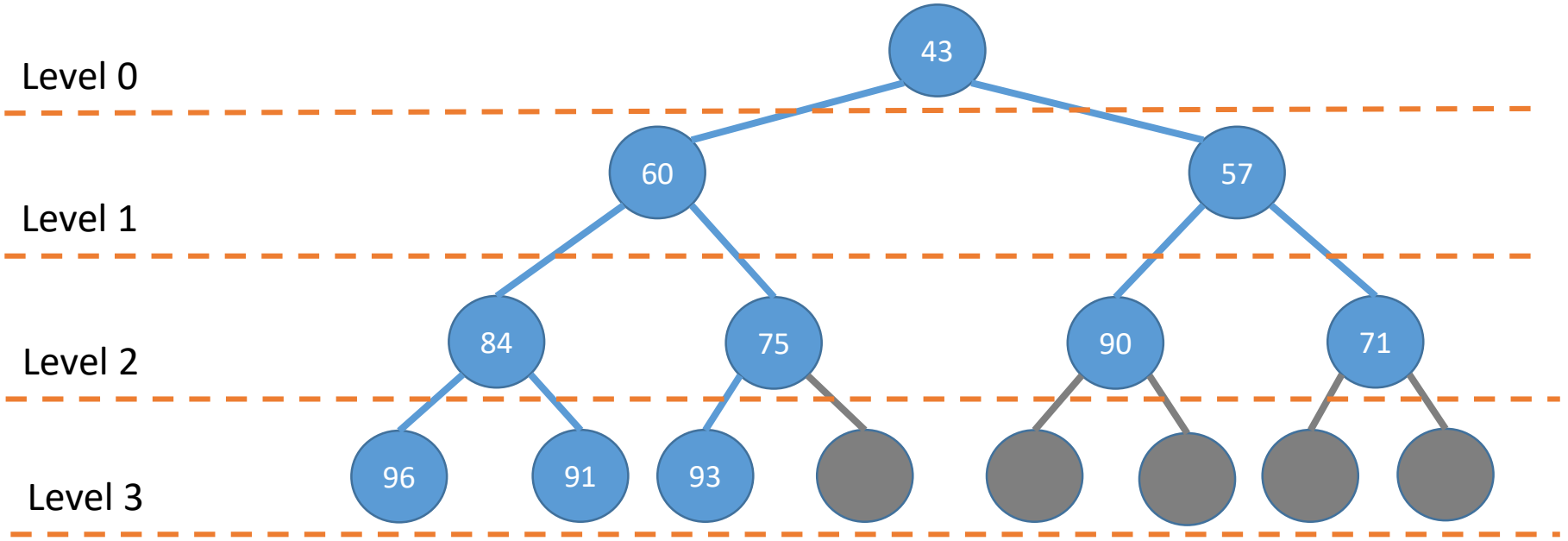
- We could implement it like that, but we will instead see how this can be done using an array and some simple, O(1), calculations.

Implementation

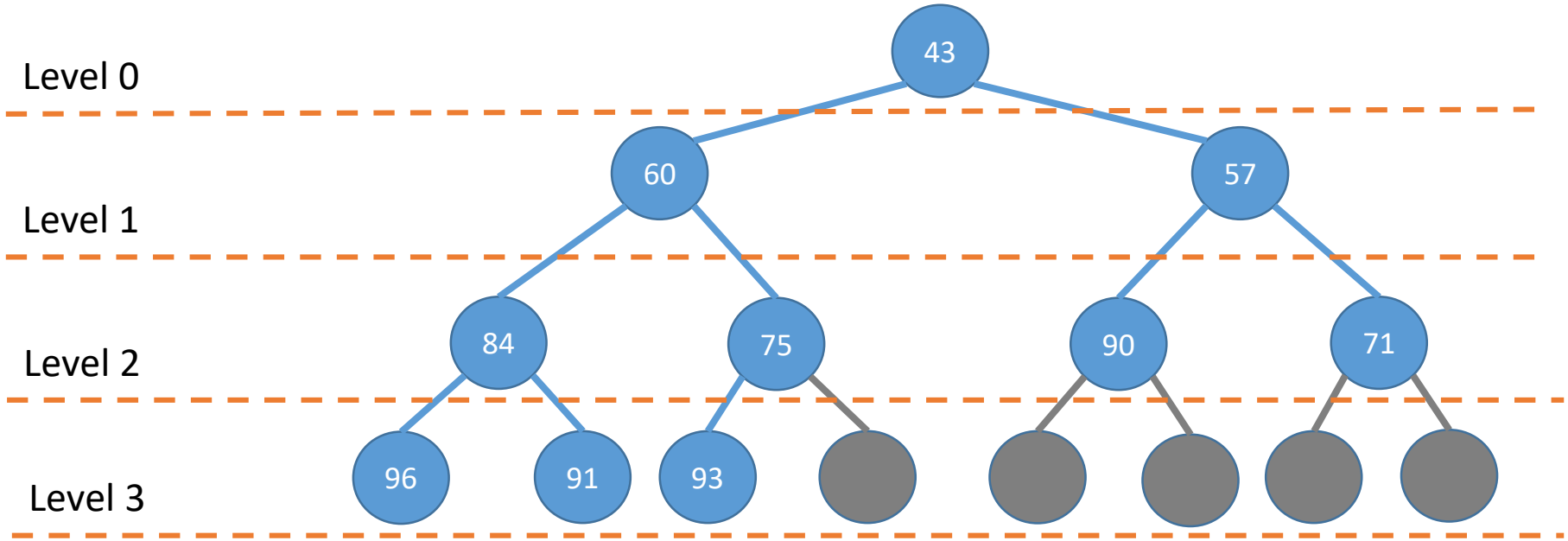
- Consider the current state of our example heap:

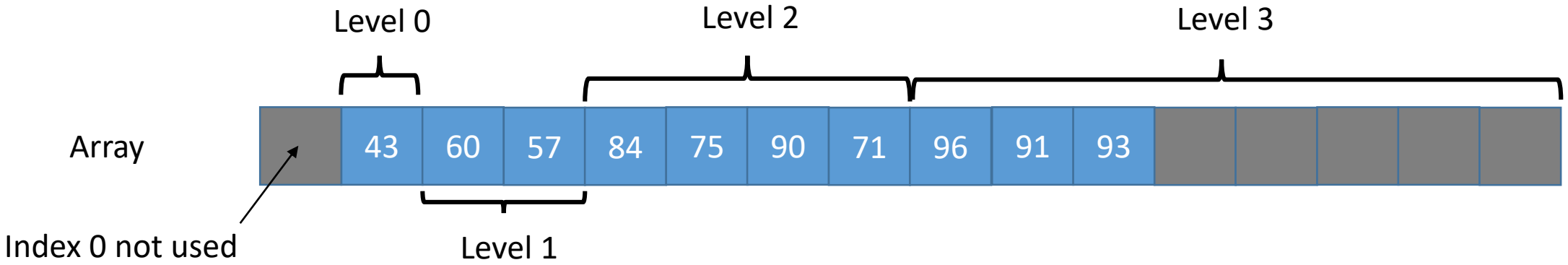


Implementation

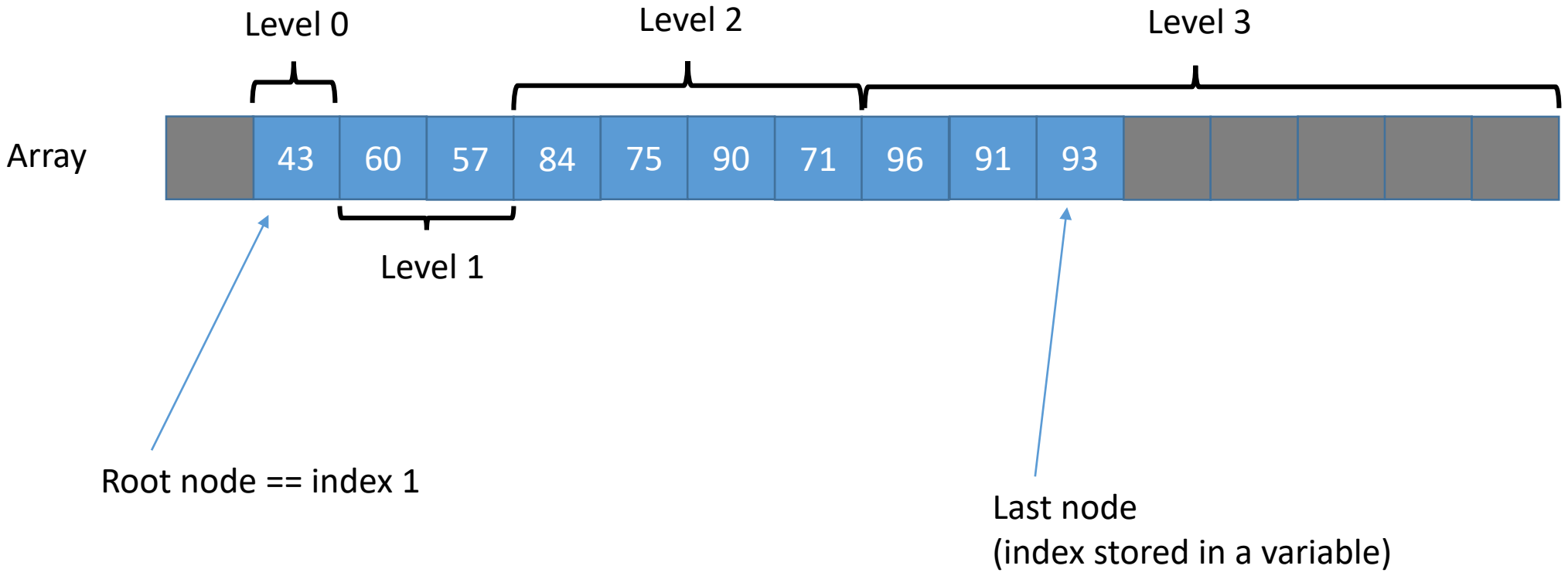


Implementation





Implementation



Implementation

- Where I is a node's index in the array
  - To find the node's parent:

    $$\frac{I}{2}$$

  - To find the node's left child:

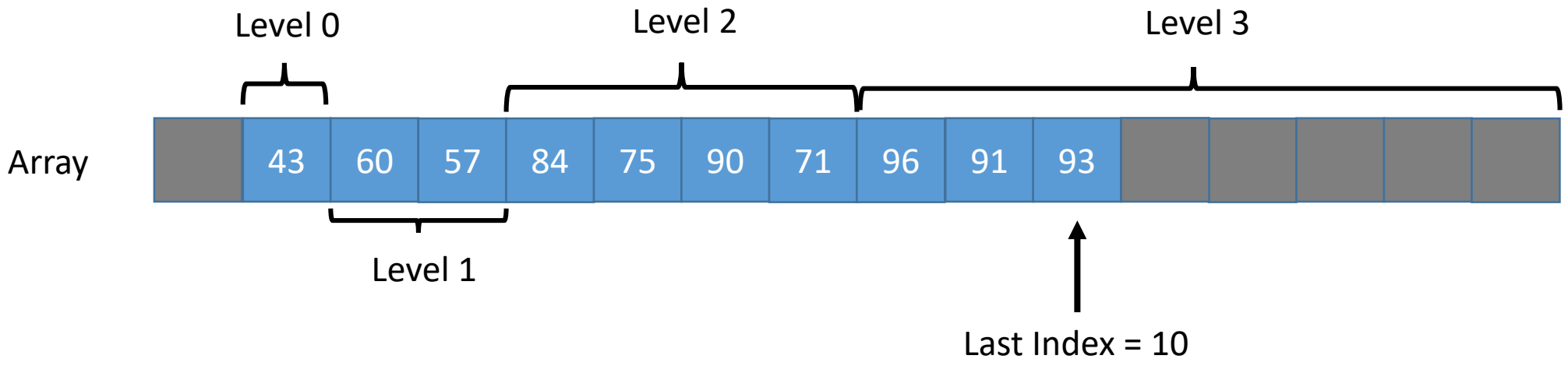
    $$I \times 2$$

  - To find the node's right child:

    $$I \times 2 + 1$$

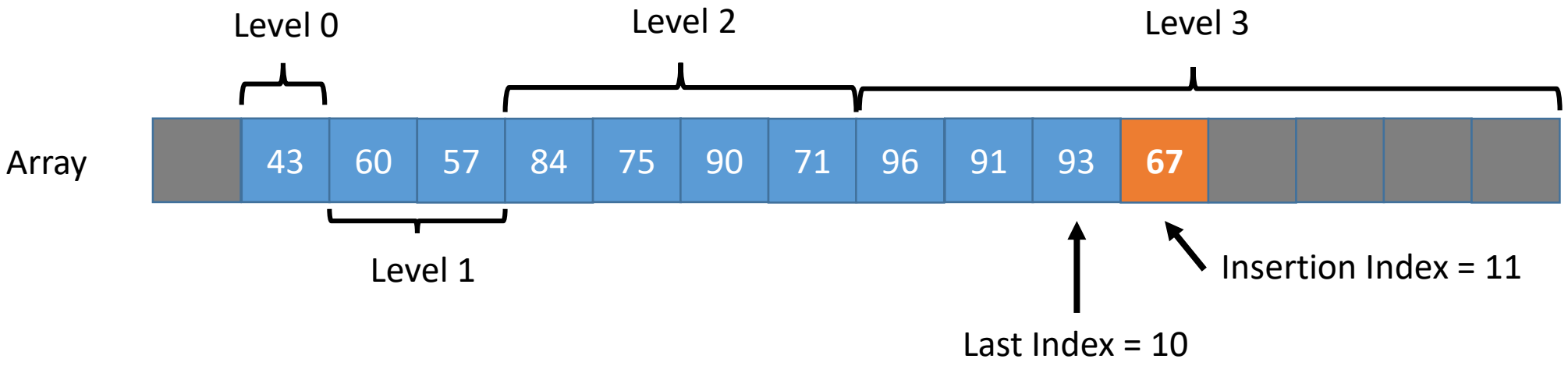
Implementation - Insertion

- A min-heap prior to inserting a new value



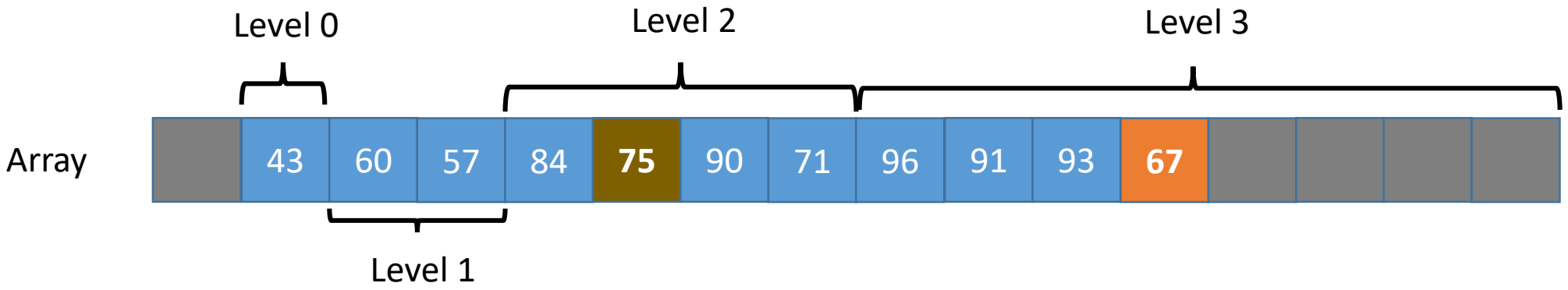
Implementation - Insertion

- Value to be inserted: **67**
  - Inserted at next available spot
  - Insertion Index = Last Index + 1
  - 10 + 1 = 11



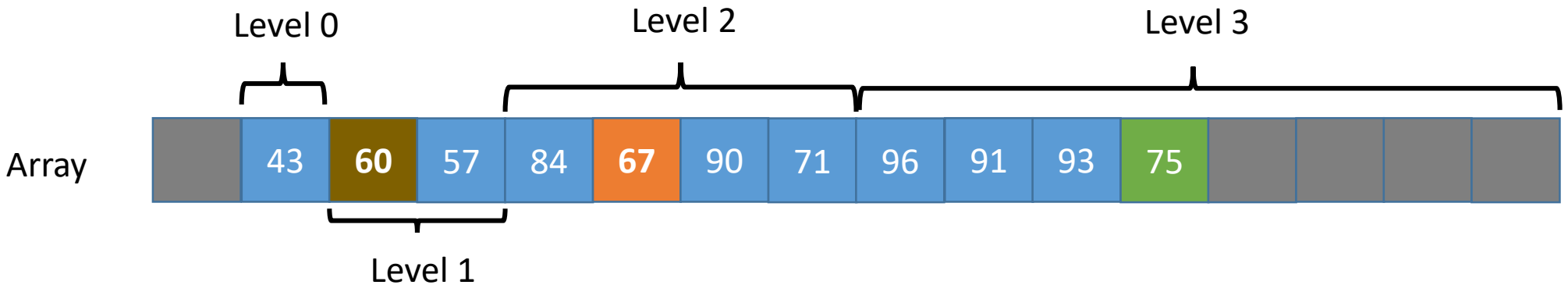
Implementation - Insertion

- Value to be inserted: **67**
  - Parent index = index / 2
  - 11 / 2 = 5.5 = 5
  - 75 > 67 == TRUE/swap

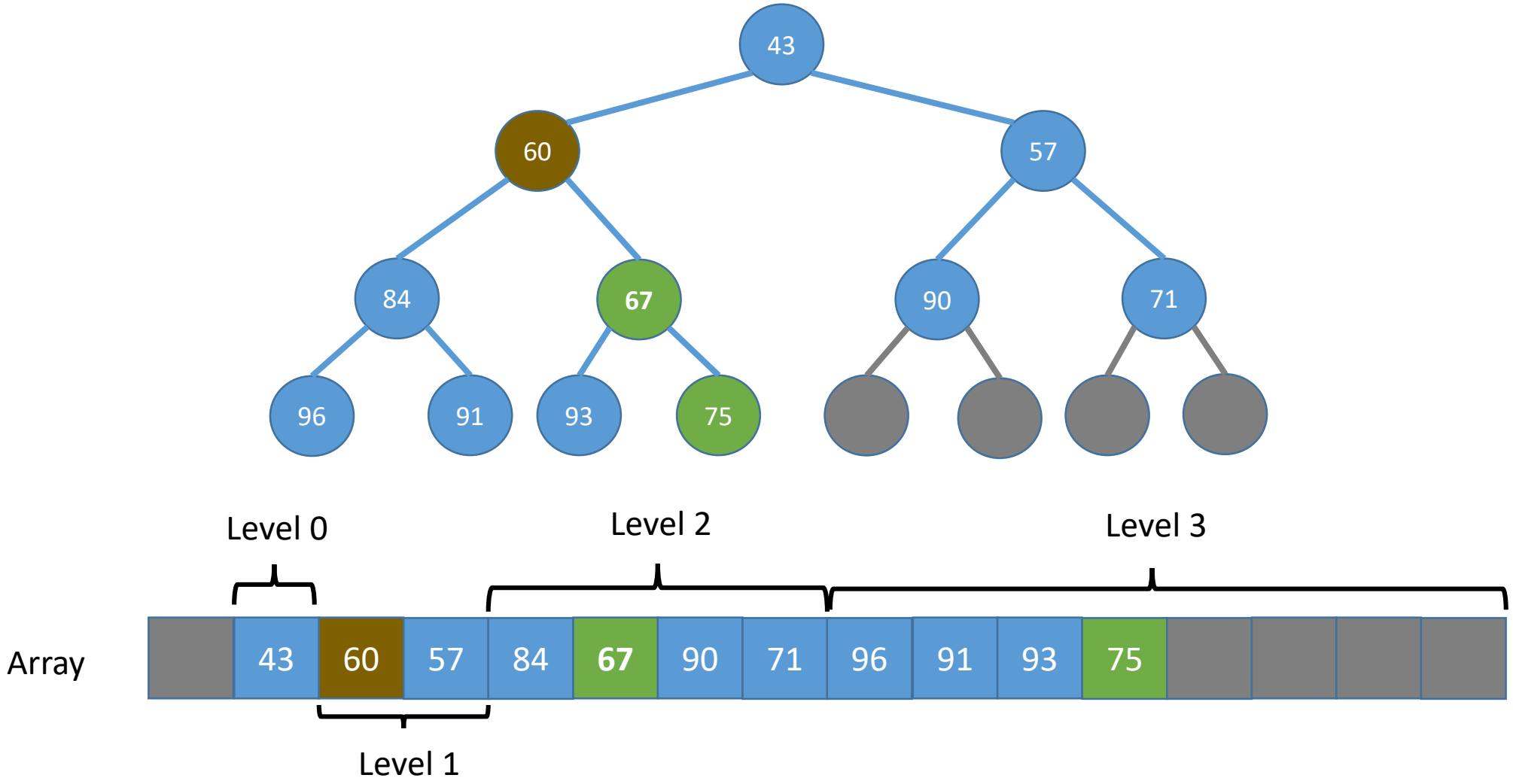


Implementation - Insertion

- Value to be inserted: 67
  - Parent index = index / 2
  - 5 / 2 = 2.5 = 2
  - 60 > 67 == FALSE/stop

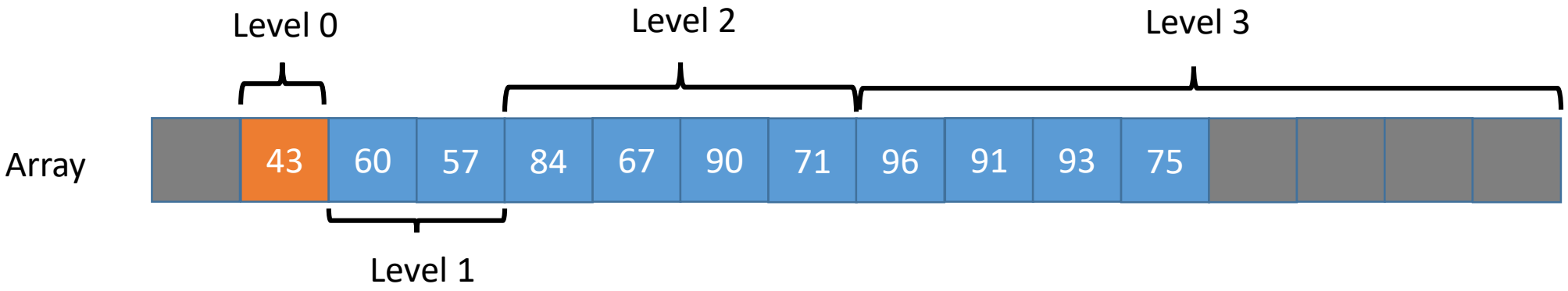


Implementation - Insertion



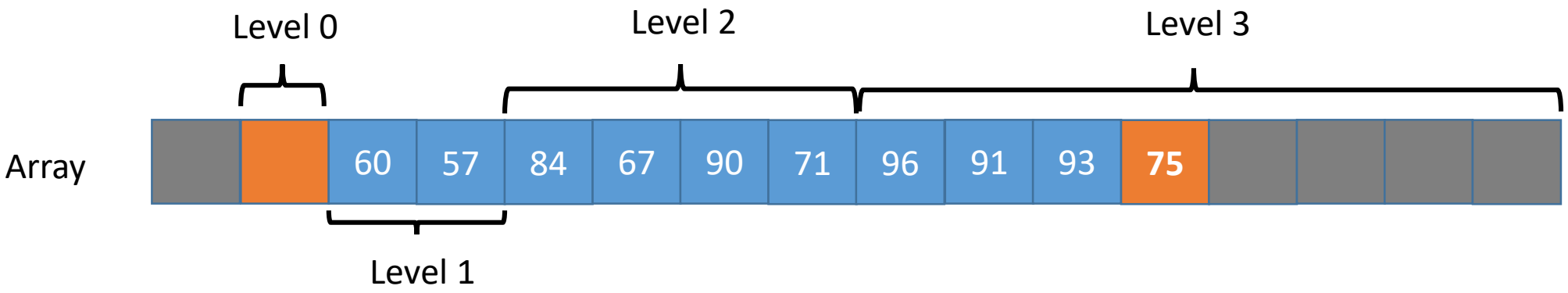
Implementation - Removal

- Value to be removed: **43**
  - Root node is only ever removed.



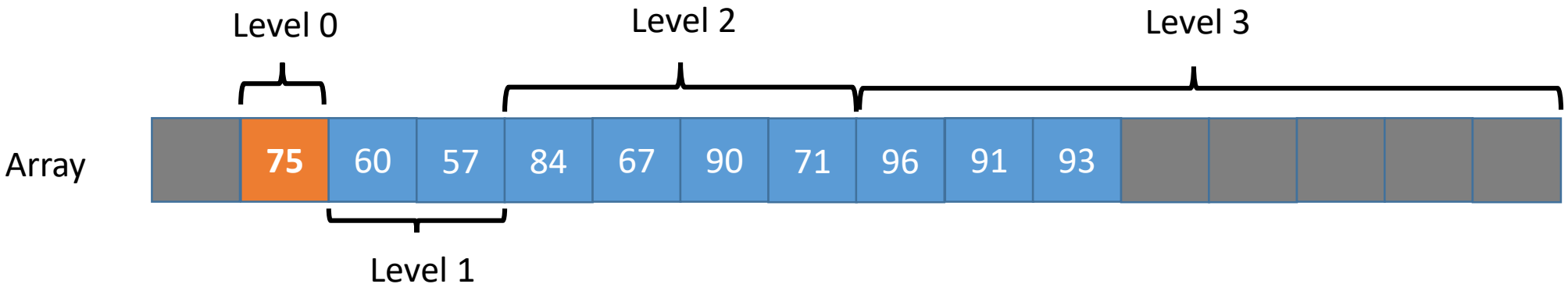
Implementation - Removal

- Move last node to the root
  - Clear the last node's spot



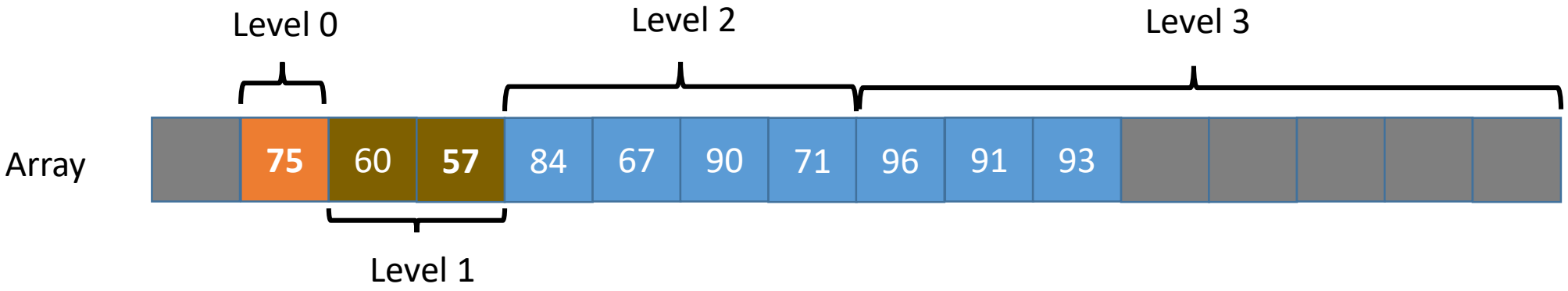
Implementation - Removal

- Move last node to the root
  - Clear the last node's spot



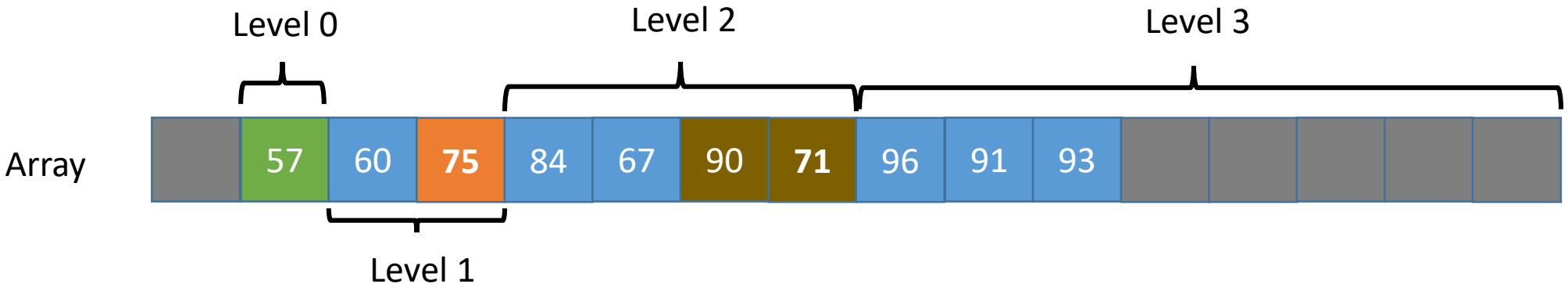
Implementation - Removal

- Swap with the smaller of its two children (if they both exist and both are smaller than its value)
  - Left node = index * 2 = 1 * 2 = **Index 2**
  - Right node = index * 2 + 1 = 1 * 2 + 1 = **Index 3**



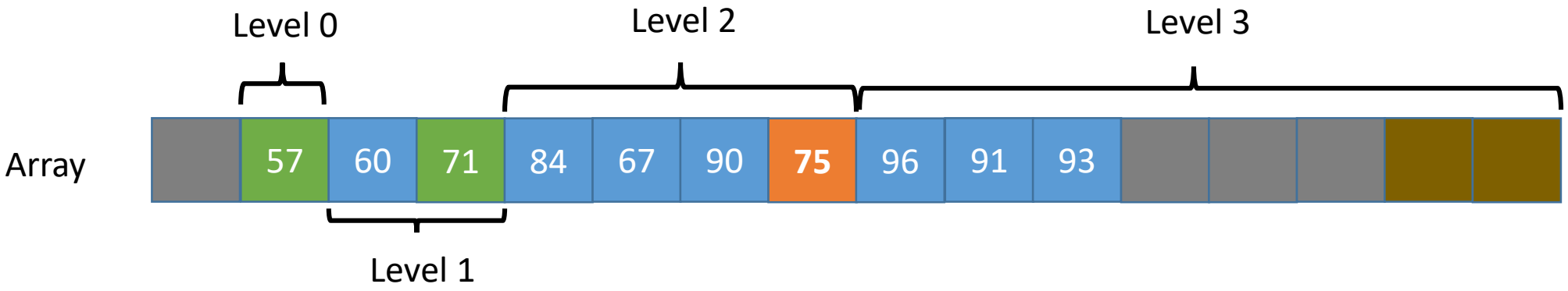
Implementation - Removal

- Swap with the smaller of its two children (if they both exist and both are smaller than its value)
  - Left node = index * 2 = 3 * 2 = **Index 6**
  - Right node = index * 2 + 1 = 3 * 2 + 1 = **Index 7**

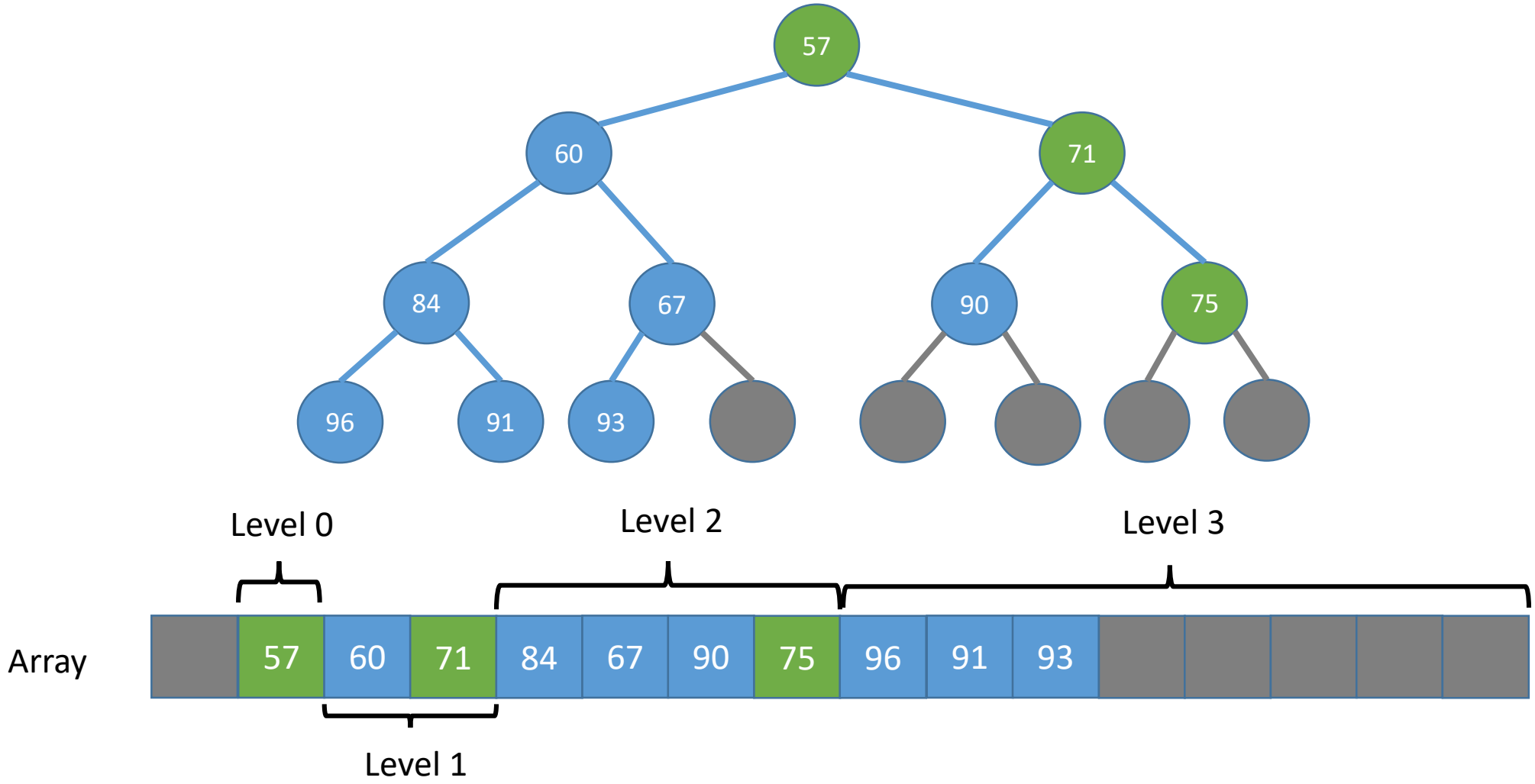


Implementation - Removal

- Swap with the smaller of its two children (**if they both exist** and both are smaller than its value)
  - Left node = index * 2 = 7 * 2 = **Index 14**
  - Right node = index * 2 + 1 = 7 * 2 + 1 = **Index 15**



Implementation - Removal



Max-Heaps

- A max-heap functions almost identically to min-heaps (the heap previously described in the examples) except:
  - Each node's value is *greater than* the values of its left and right children/subtrees.
  - In a min-heap, each node's value was *less than* the values in left and right children/subtrees

- This ensures the **largest** value in is always at the root node.
  - Unlike the min-heap, where the **smallest** value is always at the root node.

Heap Complexity

- Where *h* is the height of the heap, insertion and removal will only visit, at most, *h* nodes

- At the last level of the heap:
  - At least 2^{h-1} nodes
  - No more than 2^{h} nodes

- Where *n* is the number of elements:

$$2^{h-1} \leq n < 2^{h} \equiv h - 1 \leq \log_2(n) < h$$

- Insertion and Removal operations are logarithmic: O(log n)

Tree Structure Complexities

Structure	Insertion	Removal	Find Min	Find Max
Binary Search Tree*	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)
Red-Black Tree	O(log n)	O(log n)	O(log n)	O(log n)
Min-Heap	O(log n)	O(log n)	O(1)	O(n)
Max-Heap	O(log n)	O(log n)	O(n)	O(1)

*- Worst case/Pathological Tree

Priority Queues

- A **priority queue** is like a queue or stack structure where each node/element is also assigned a priority.
  - Nodes/elements with *highest priority* are removed first.

- Usually implemented using a max-heap.

- The node with the highest priority (largest value) will always be at the root.
  - Priority queues are often synonymous with heaps

Heapsorting

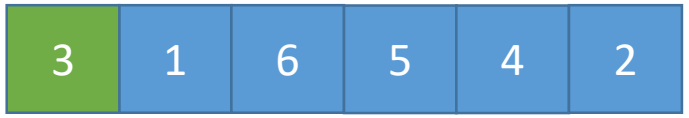
- Heapsorting can sort an array or list through the use of a min- or max-heap.

- Each element of an unsorted array is moved into a heap: O(n)
  - For each element, an insert operation is performed: O(log n)
  - O(n) * O(log n) = O(n * log n)
- Elements are removed from the heap: O(log n)
  - and placed in the array: O(n)
  - O(log n) * O(n) = O(n * log n)

- O(n * log n) + O(n * log n) = **O(n * log n)**

Heapsorting

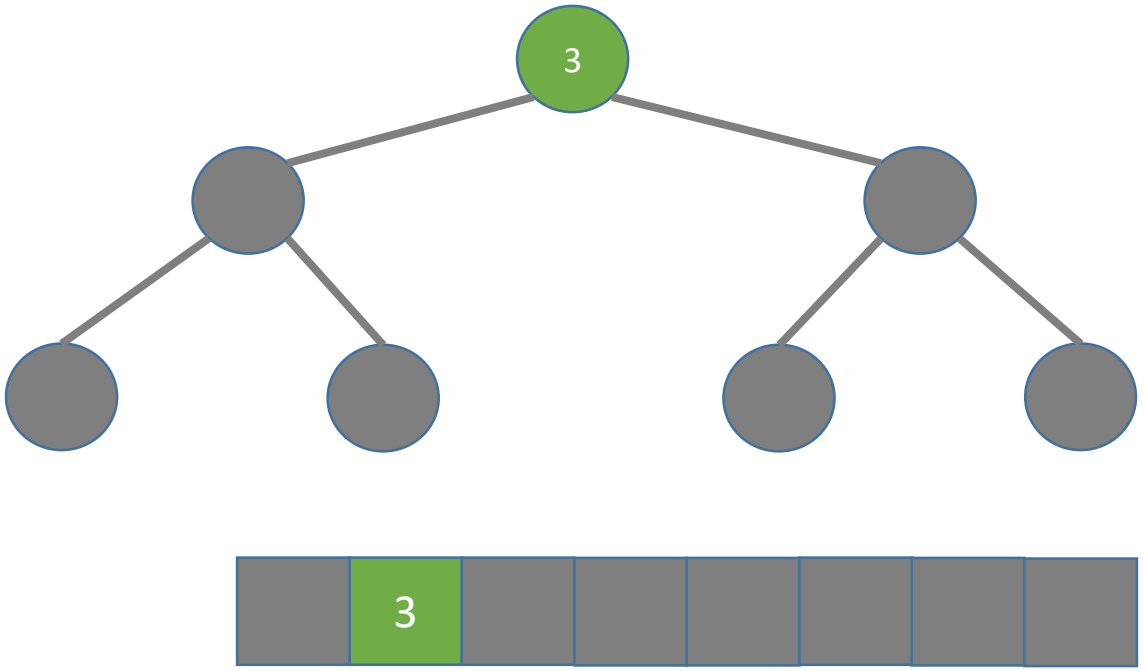
- Copy each element from the unsorted array to the (max)heap.



Number of levels needed:
length / 2

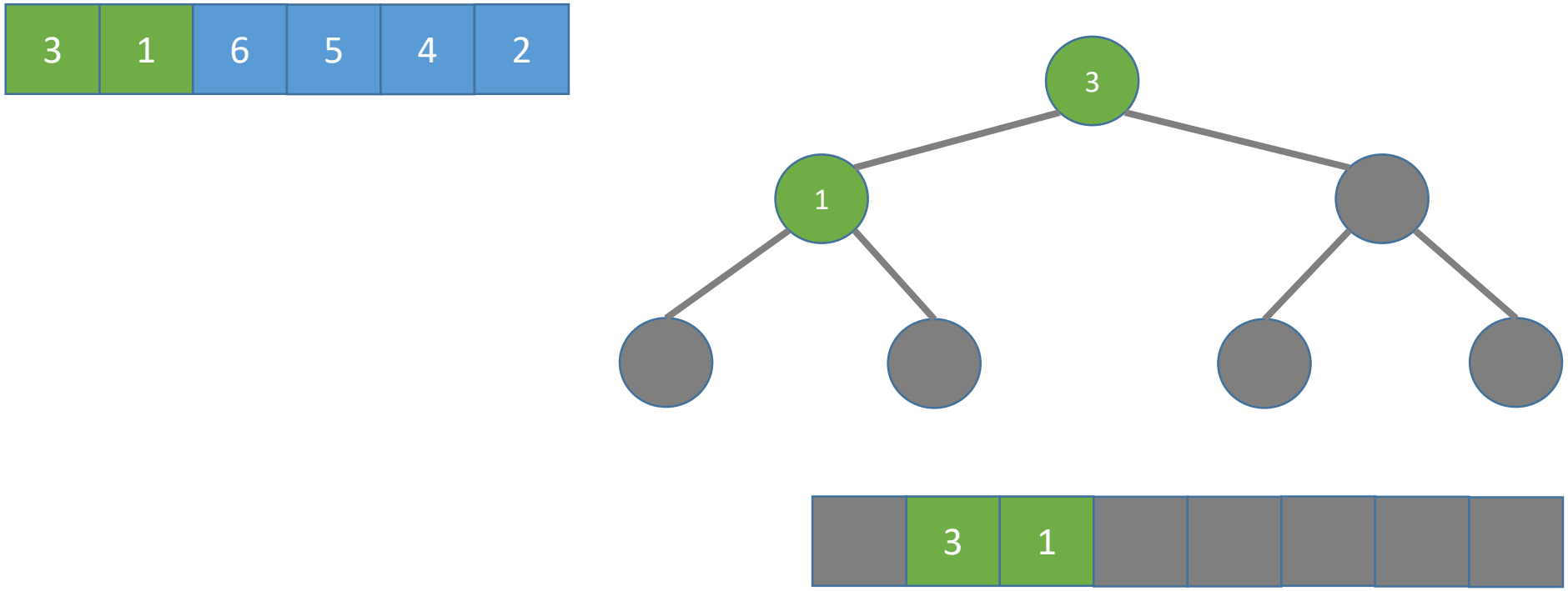
Heap array length needed:
$2^{length/2}$

(Index 0 not used)



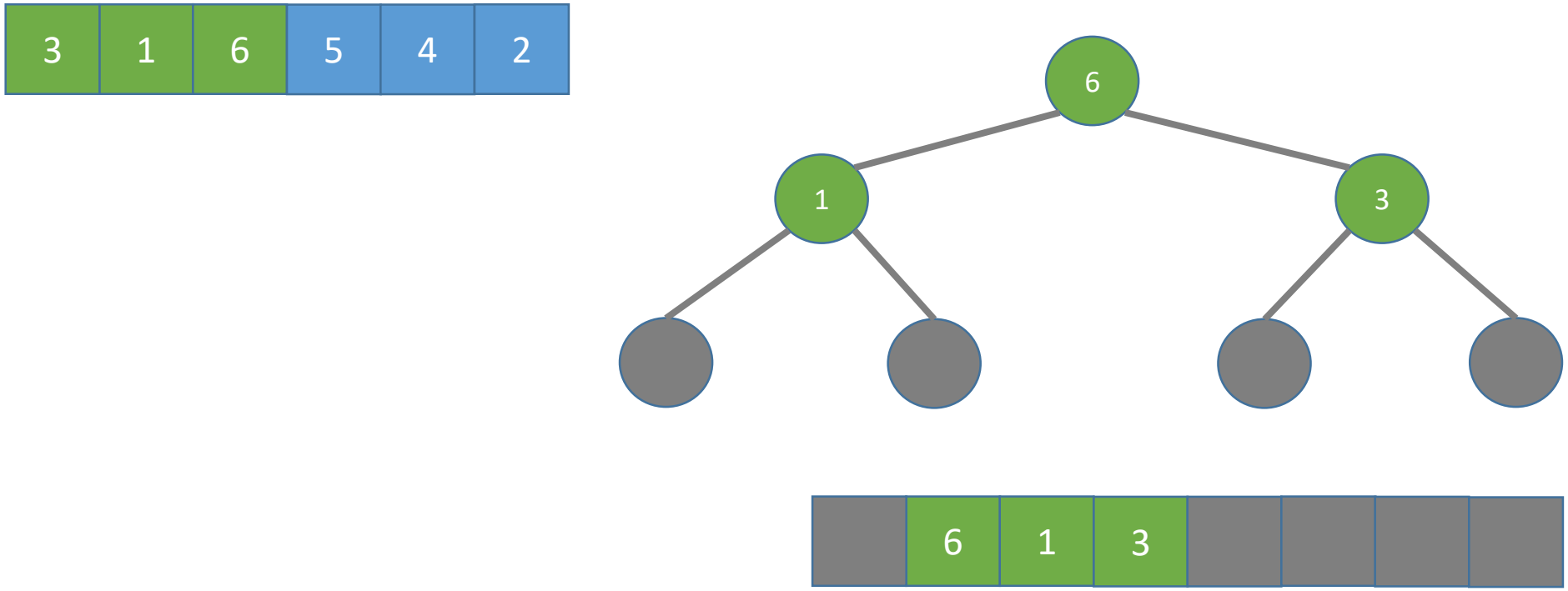
Heapsorting

- Use a max-heap for descending order.



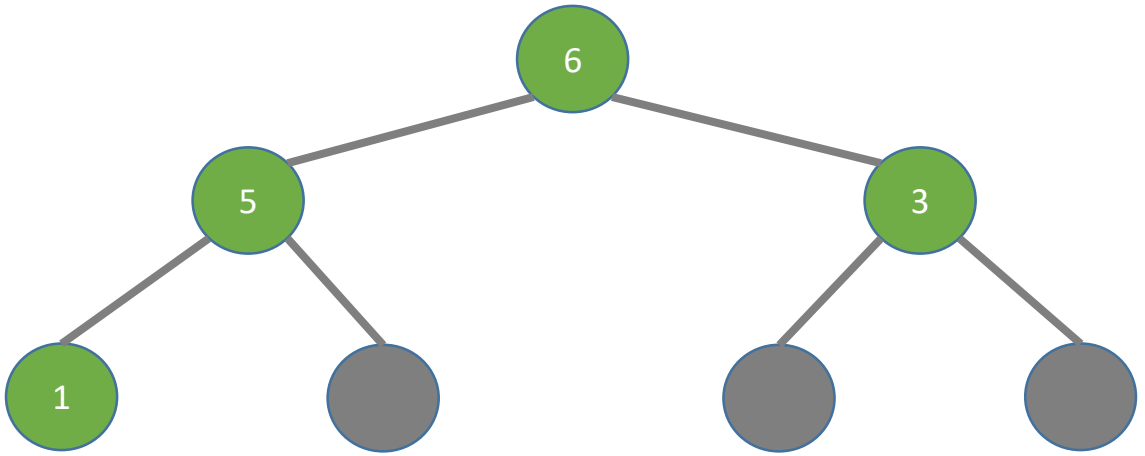
Heapsorting

- Use a max-heap for descending order.



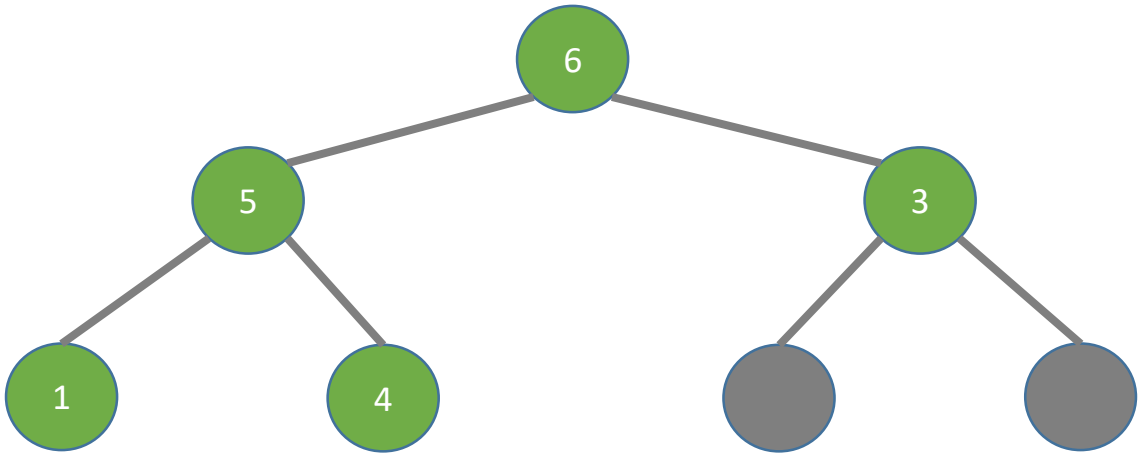
Heapsorting

- Use a max-heap for descending order.



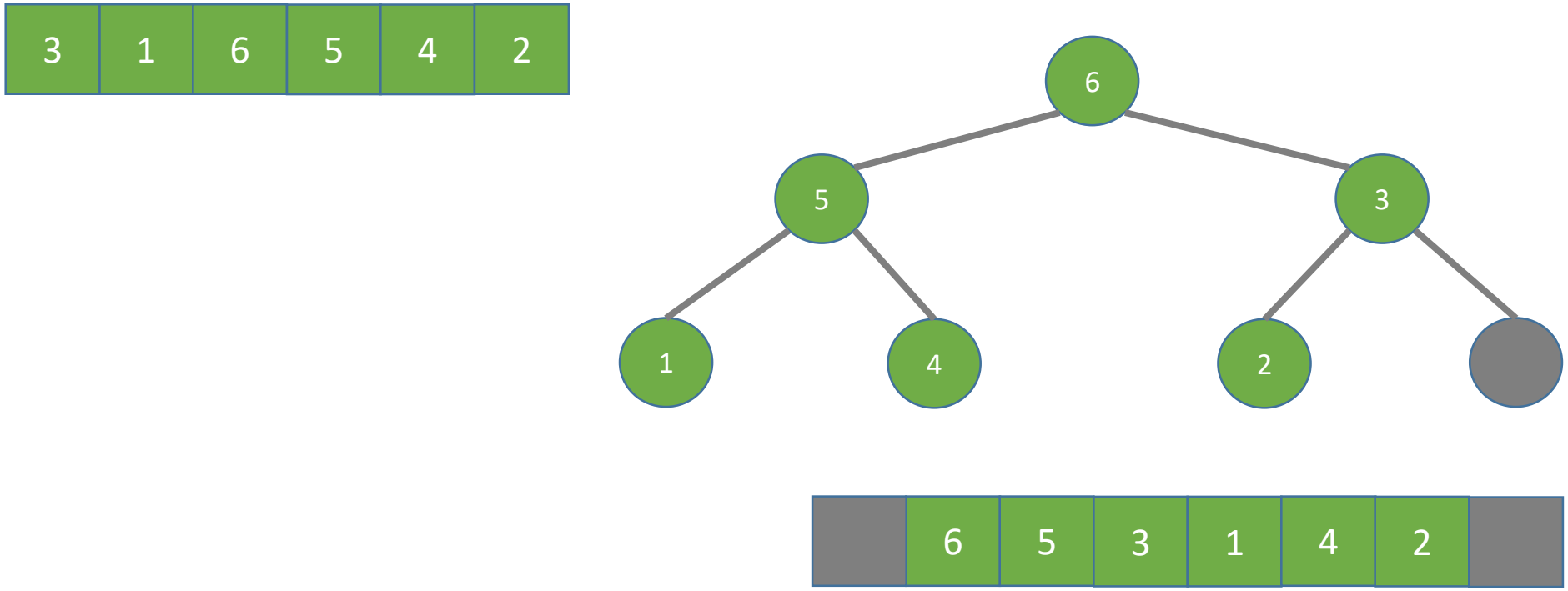
Heapsorting

- Use a max-heap for descending order.



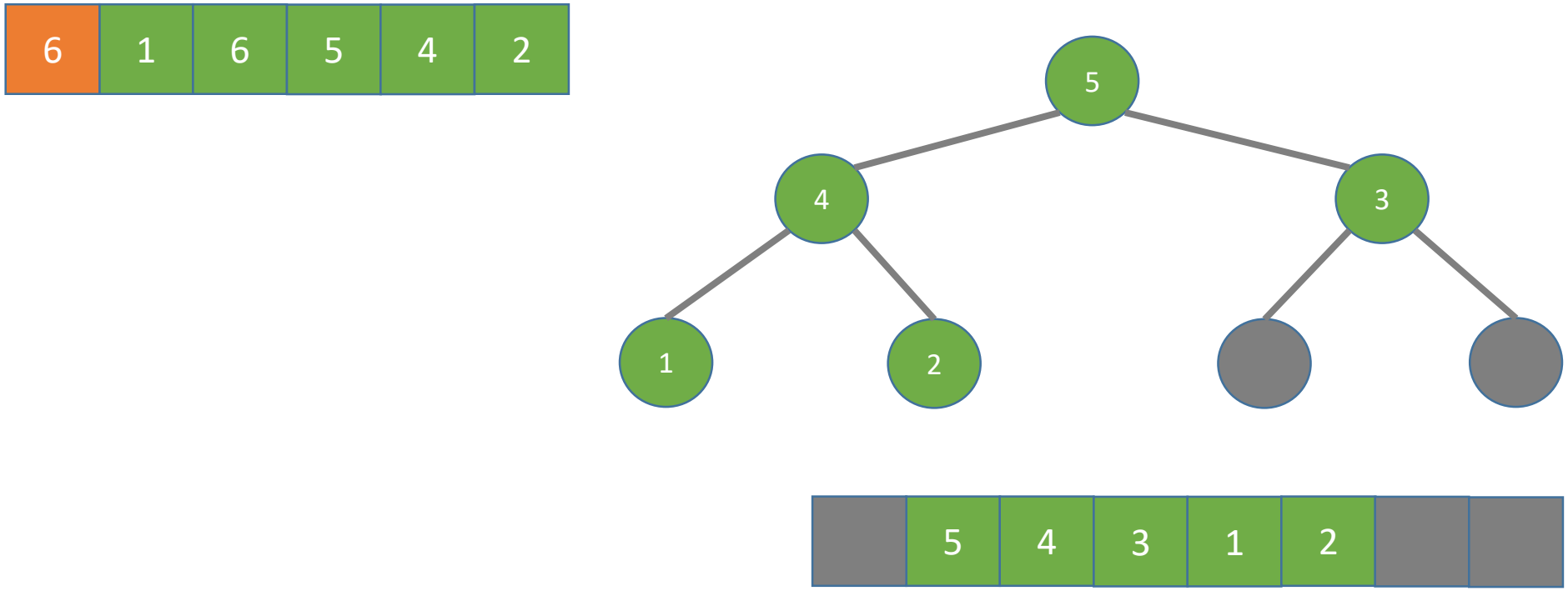
Heapsorting

- Use a max-heap for descending order.



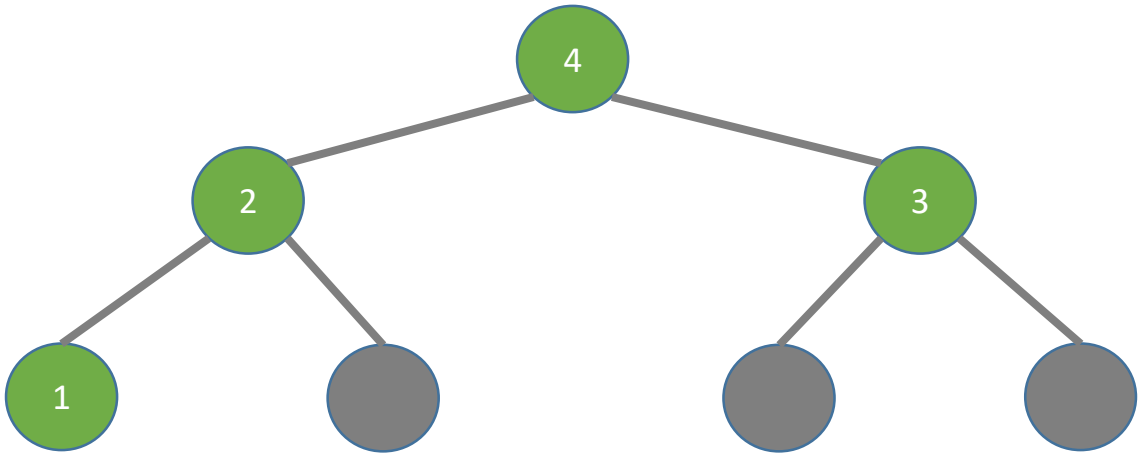
Heapsorting

- Retrieve the root node until the heap is empty.



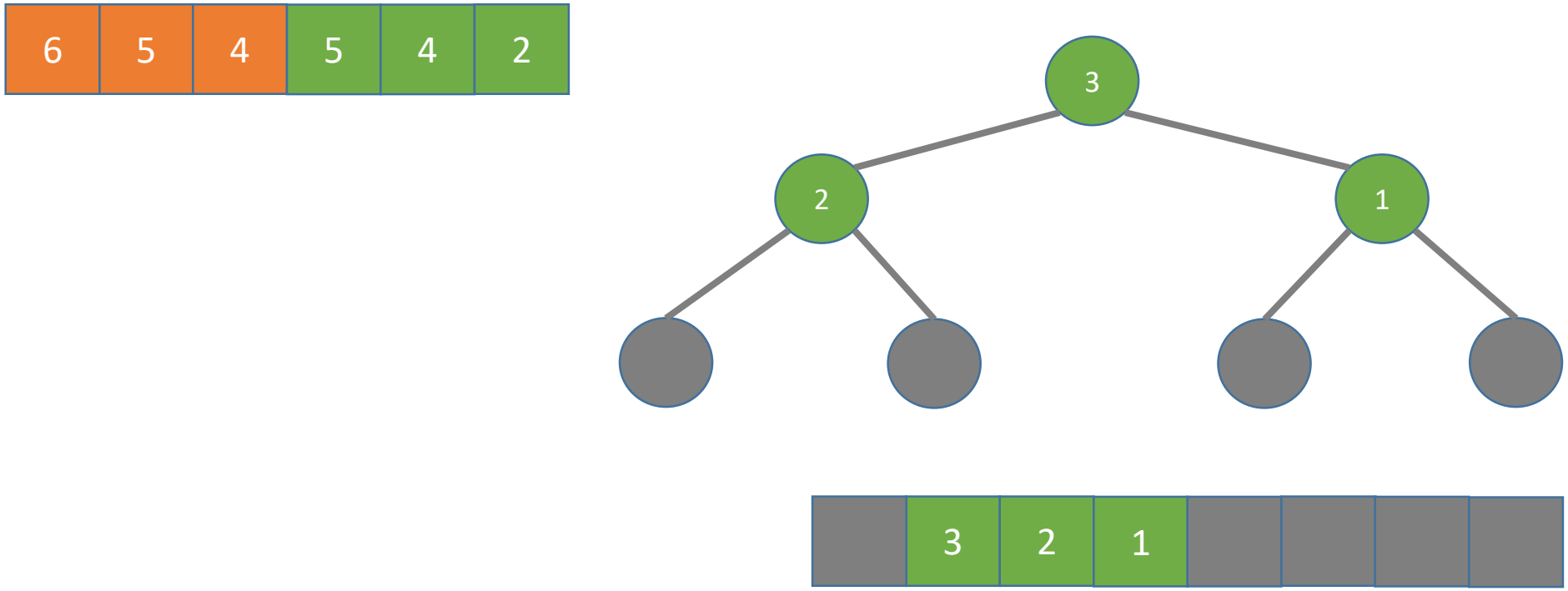
Heapsorting

- Retrieve the root node until the heap is empty.



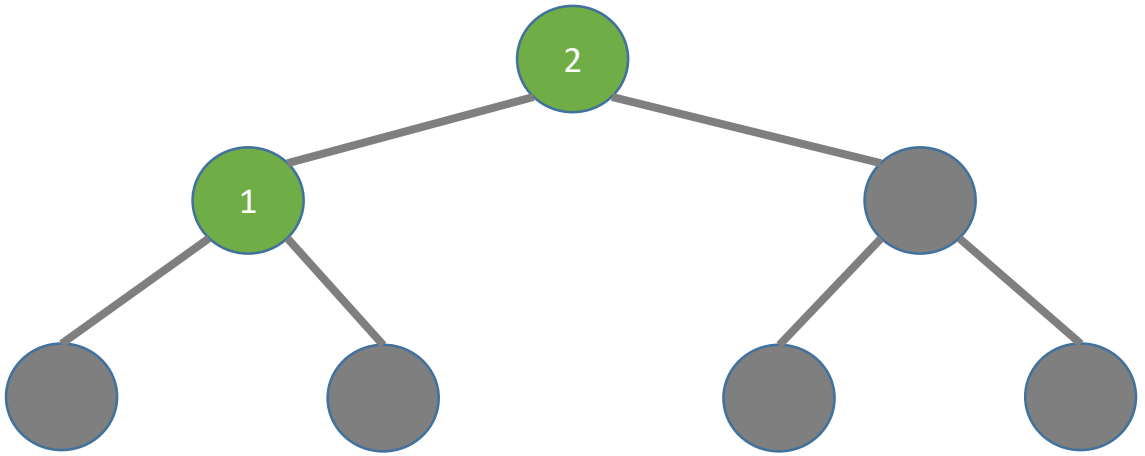
Heapsorting

- Retrieve the root node until the heap is empty.



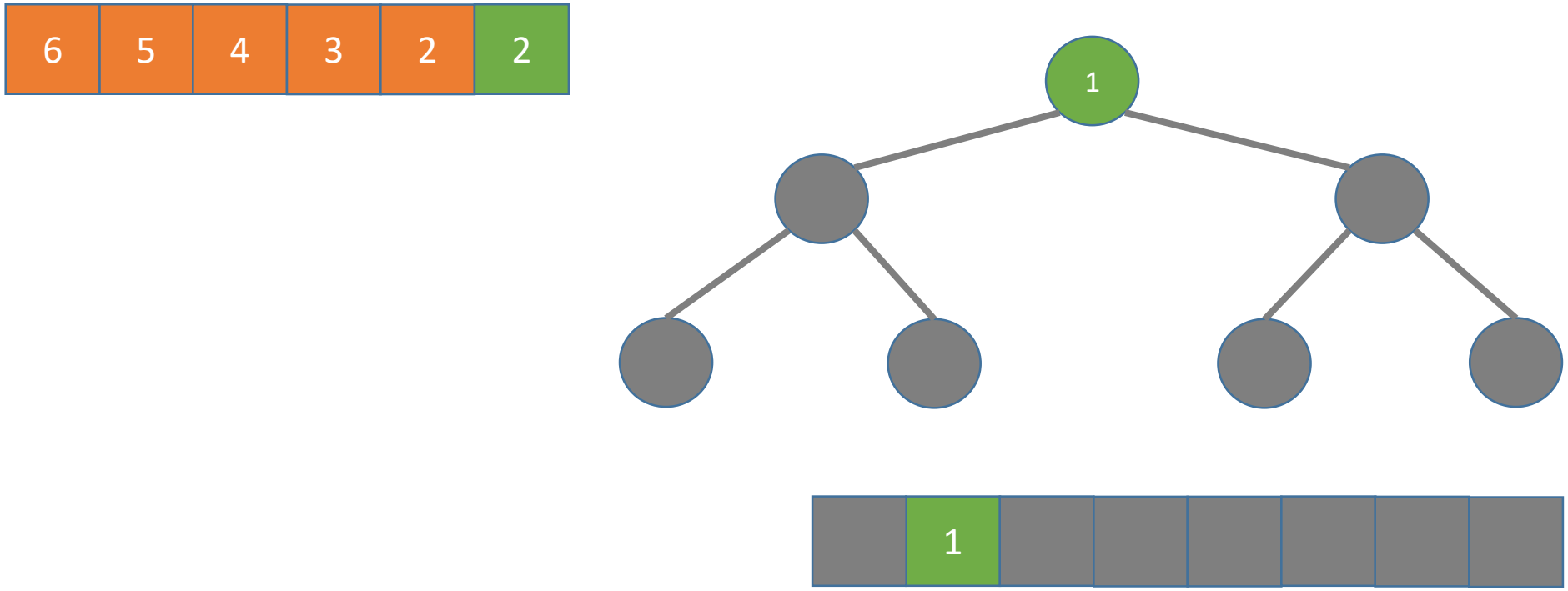
Heapsorting

- Retrieve the root node until the heap is empty.



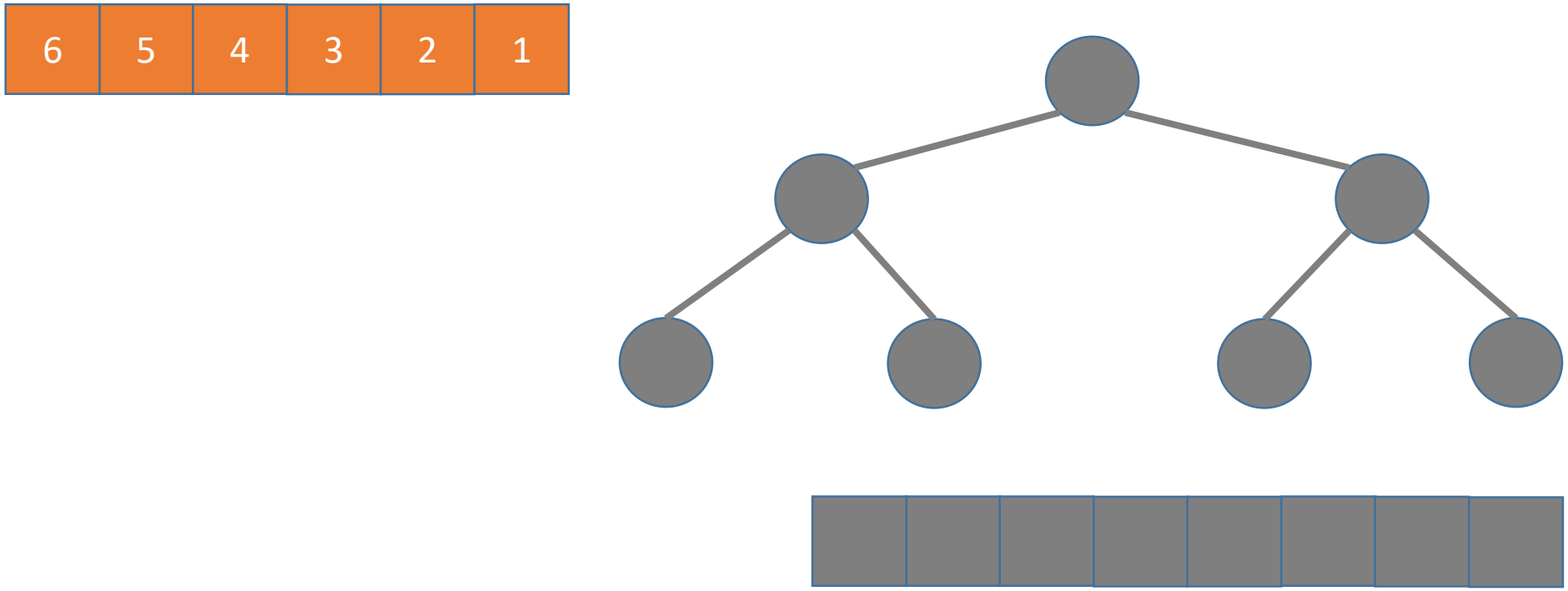
Heapsorting

- Retrieve the root node until the heap is empty.



Heapsorting

- Retrieve the root node until the heap is empty.



Heapsorting

- The previous example shows using a max-heap can achieve descending order.
  - Or ascending order if we re-filled the array starting at the end and working backwards

- A min-heap can be used for ascending order.
  - Or descending order if we re-filled the array starting at the end and working backwards