Priority Queues

Priority Queues

- Priority queues prioritize collections of key-value pairs, based on a total order on the keys.

  - Example: Values are airline flights, and the keys are (arrival or departure) times, e.g.,

    (11:16 am, DL3347).

- An entry with any key may be inserted at any time.

- You may only examine or remove entry with lowest key.

Priority Queues in Simulation

- Priority is used as an “event queue”.
- Values are events,
- Keys are times events take place.
- Simulation removes successive events from queue and simulates them.

Methods

- **insert:** Add a new element to the queue.

  - *Duplicates allowed*

- **min / max:** Return the element with the smallest / largest key in the queue.

- **removeMin / removeMax:** Return and remove the element with the smallest / largest key in the queue.

`Queue` Interface

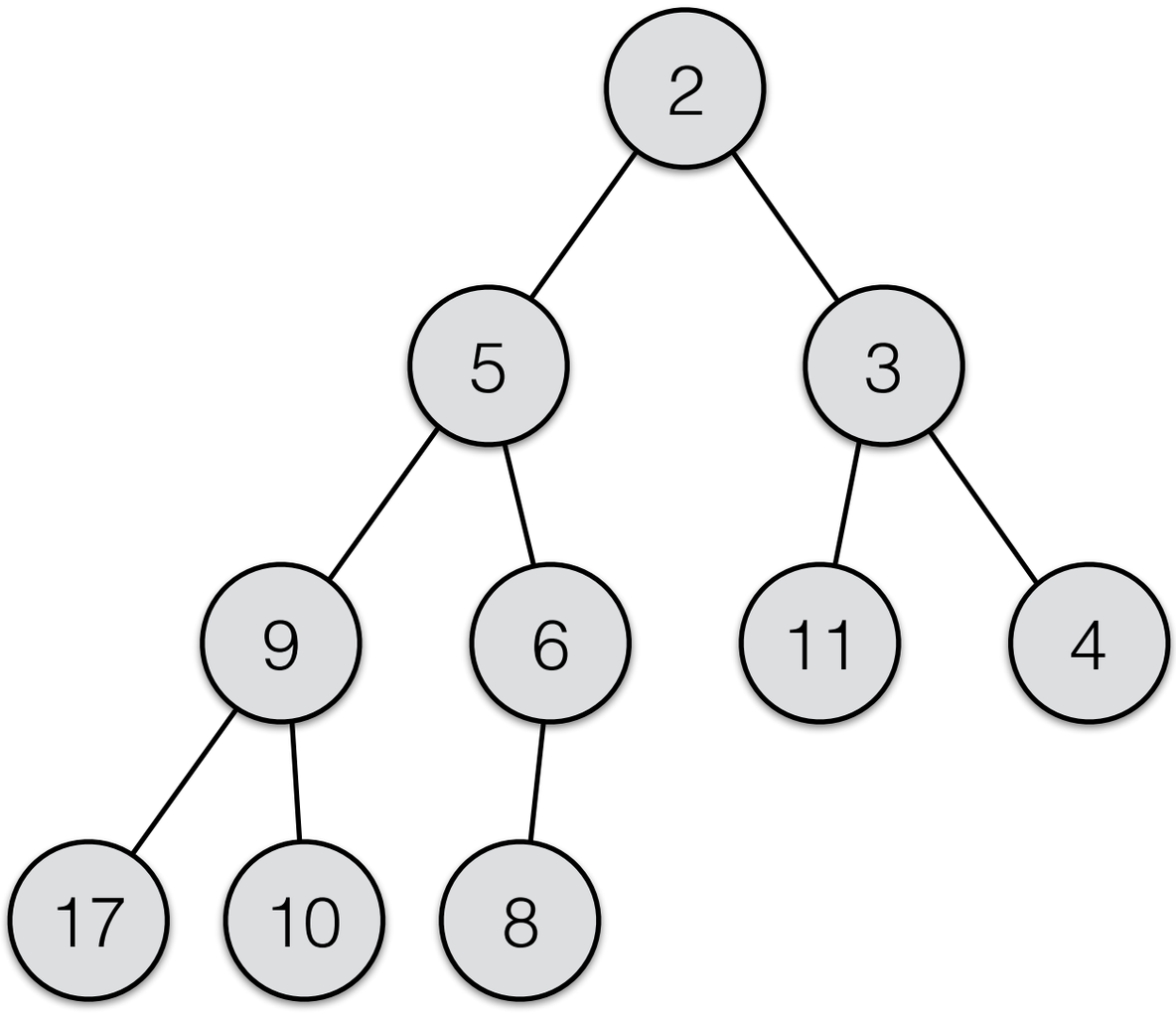
- **`add()`** = insert
- **`peek()`** = min / max
- **`remove()`** = removeMin / removeMax

Simple Implementations

	List/Array Sorted	List/Array Unsorted
`peek()`	O(1)	O(n)
`add()`	O(n)	O(1)
`remove()`	O(1)	O(n)

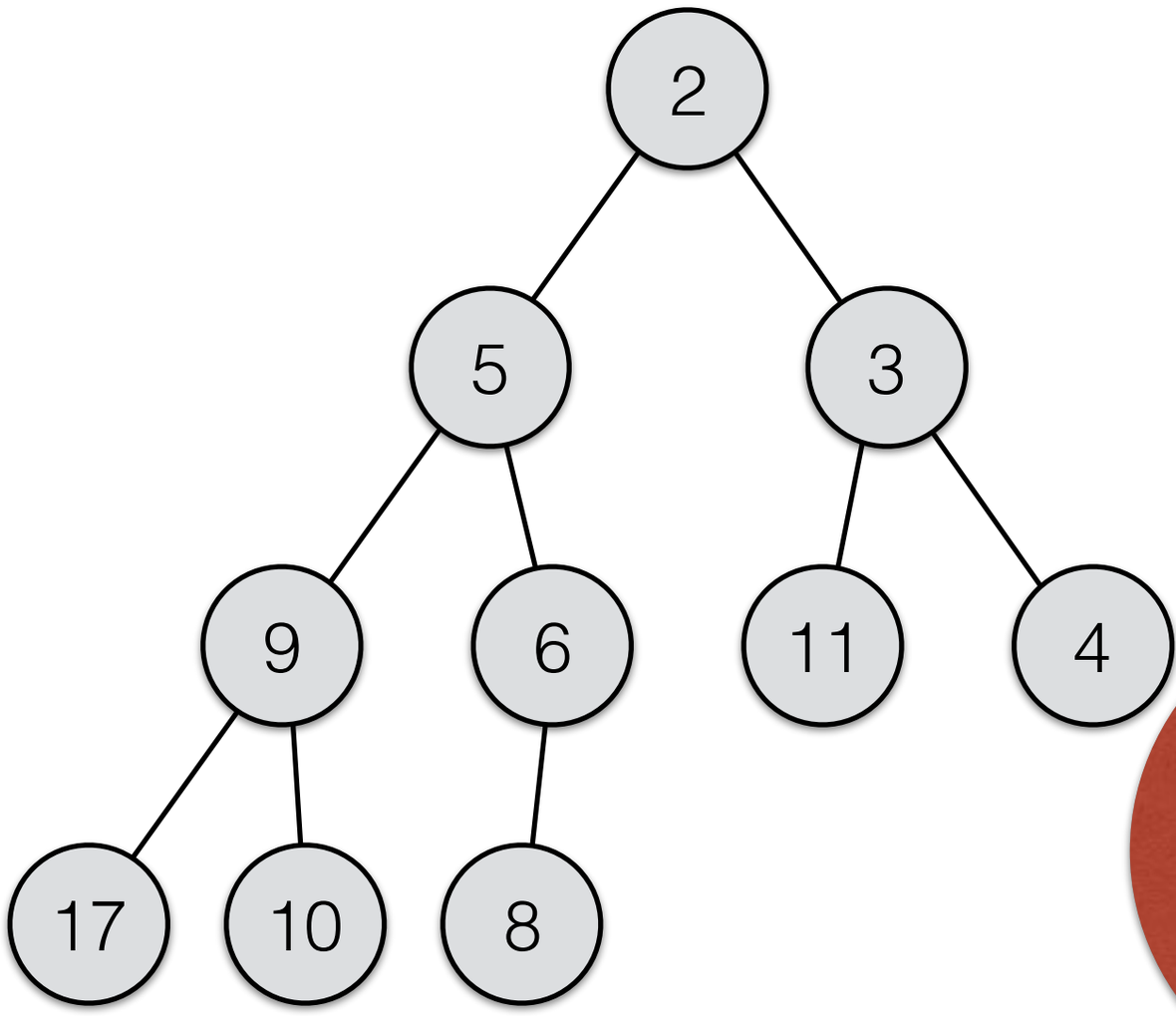
n = # elements

Heap Order



key(child) ≥ key(parent)

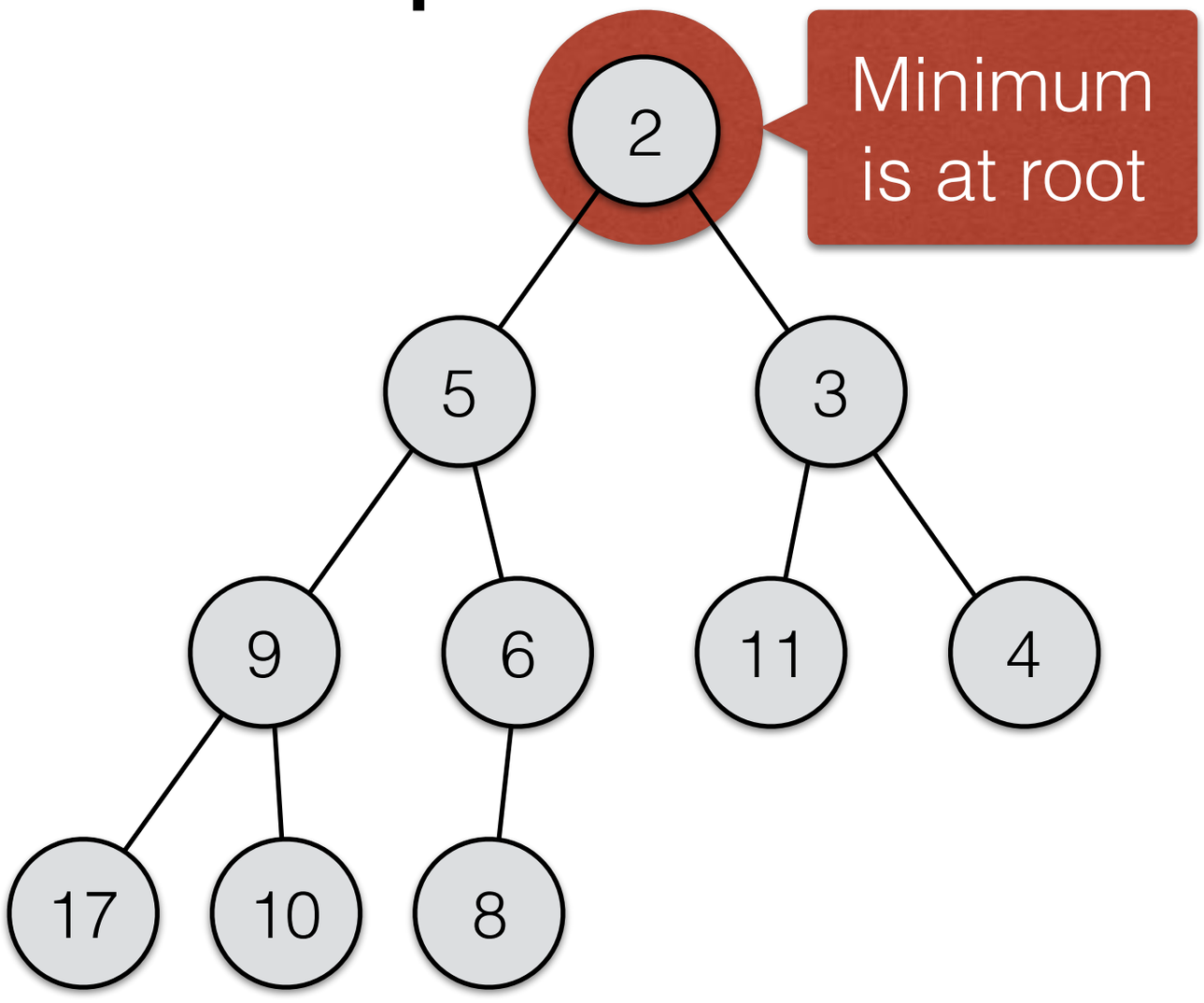
Heap Order



key(child) ≥ key(parent)

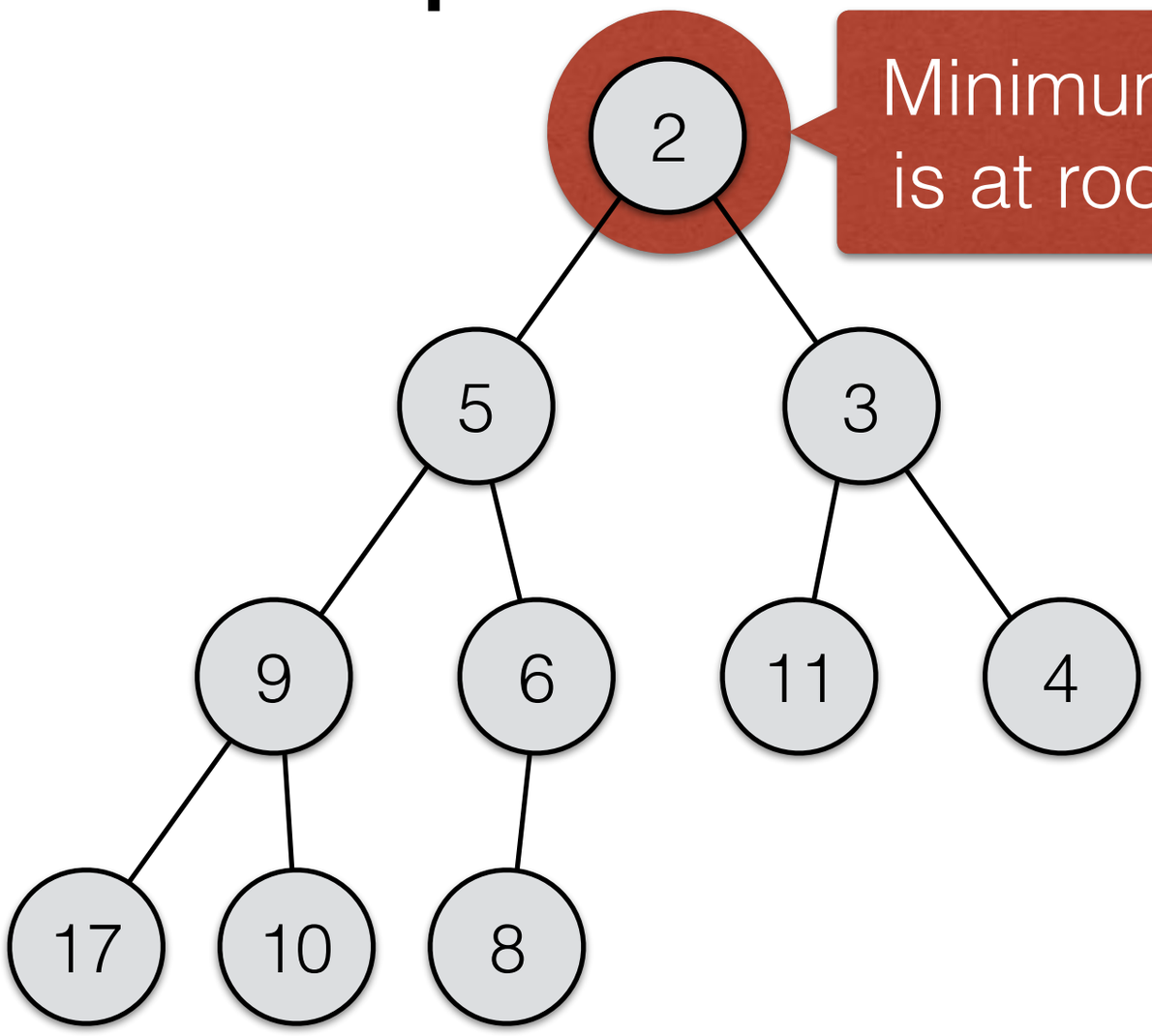
Not the same as BST Property!

Heap Order



key(child) ≥ key(parent)

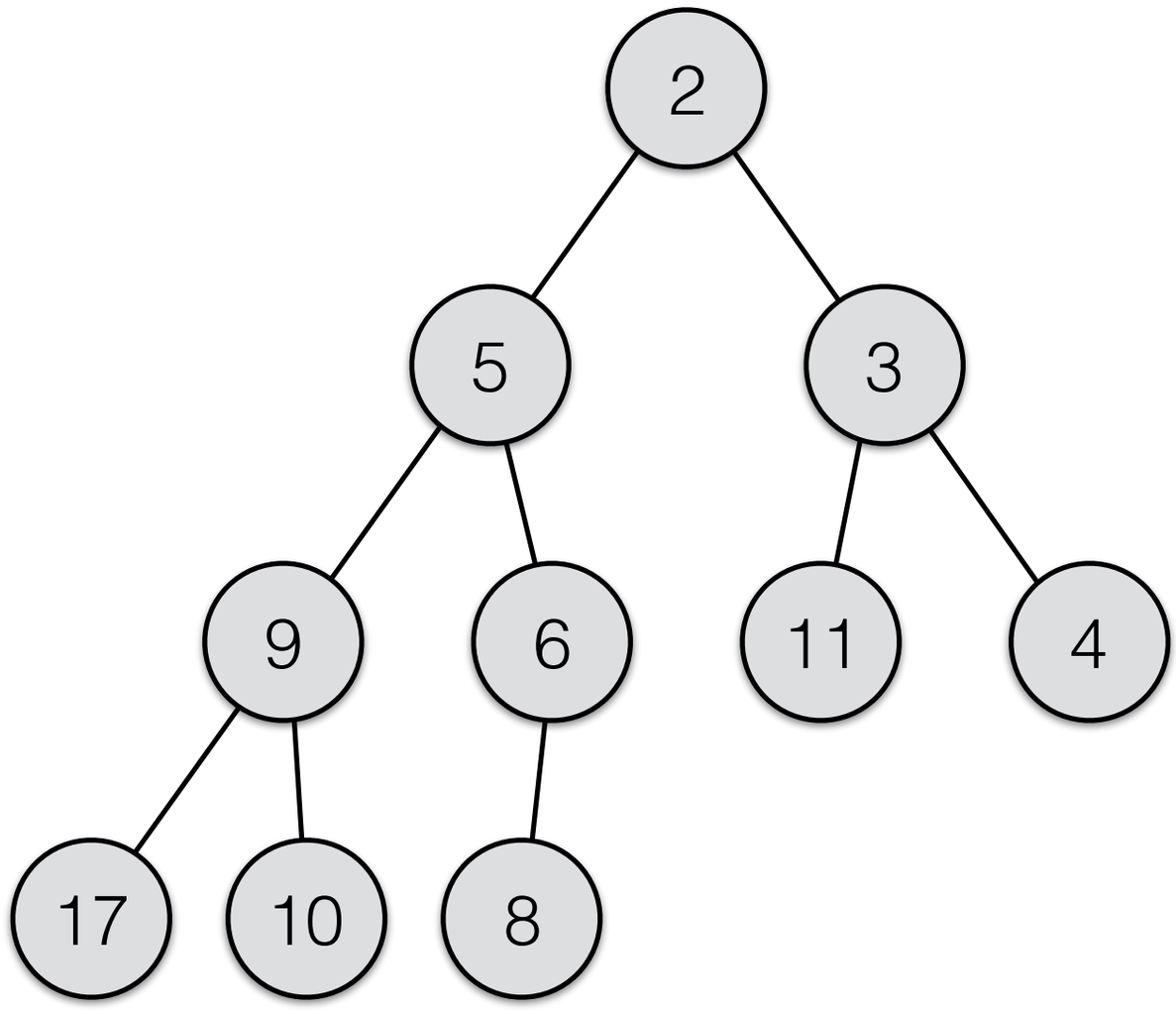
Heap Order



peek()
is O(1)

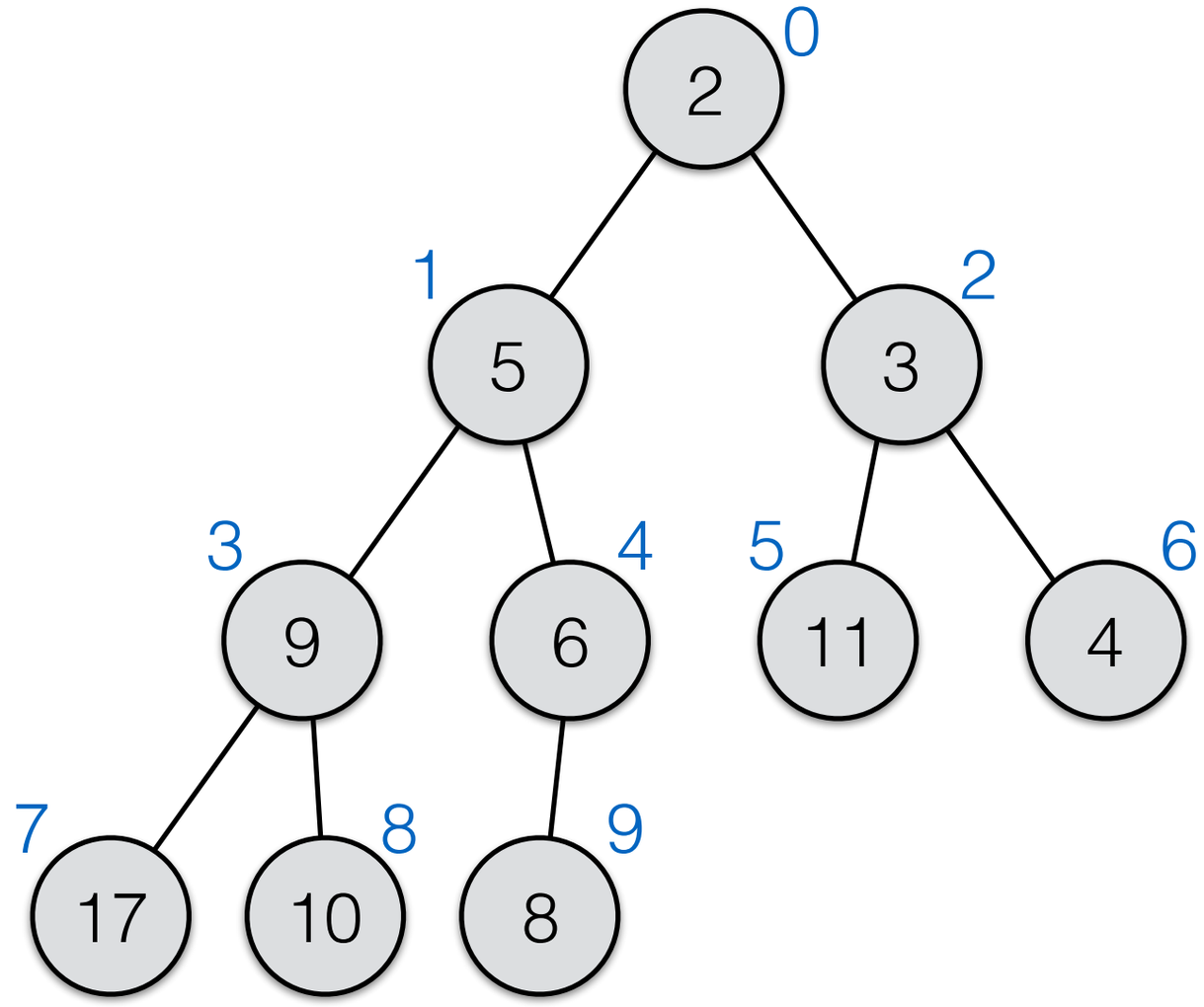
key(child) ≥ key(parent)

Complete Binary Tree

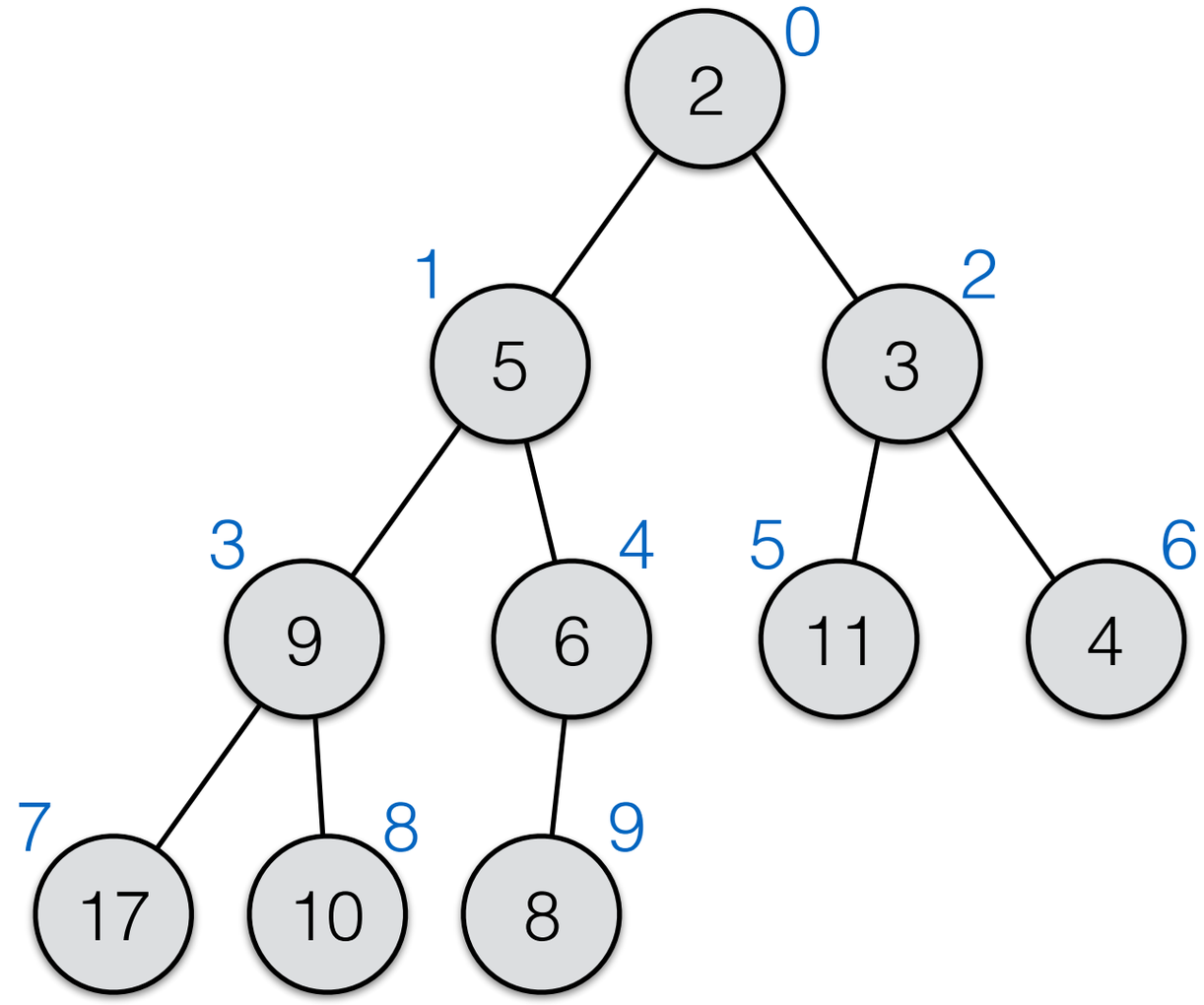


Every row is full, except, possibly, the bottom row, which is filled from left to right

Array Representation



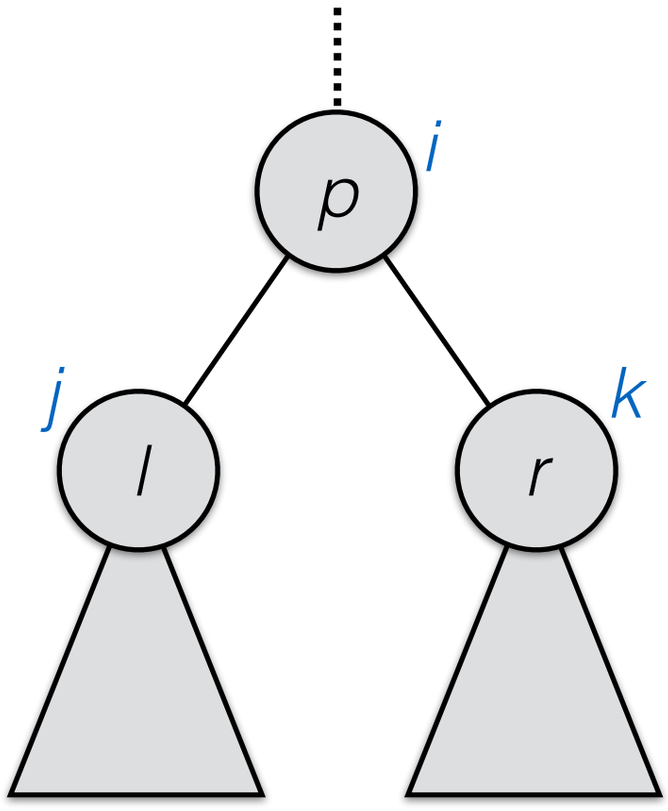
Array Representation



data:

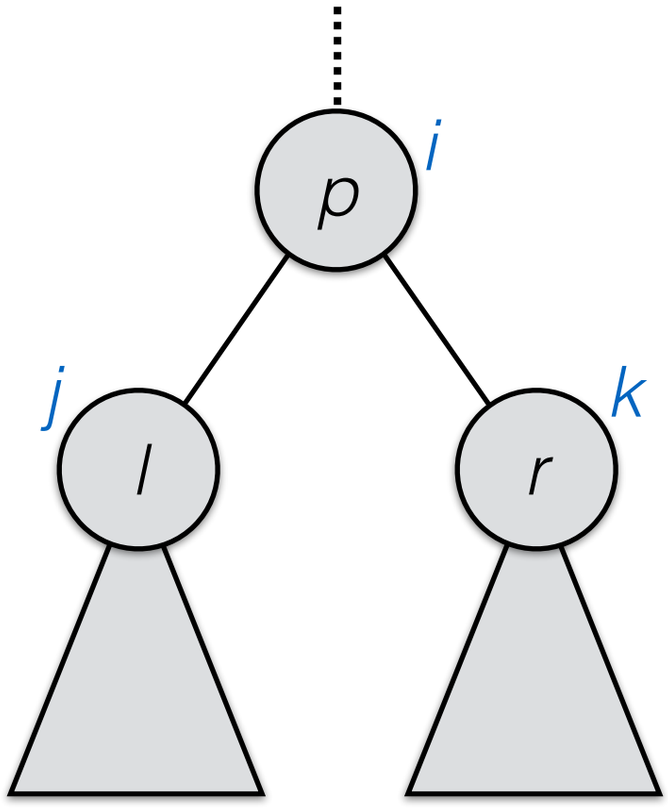
2	5	3	9	6	11	4	17	10	8
0	1	2	3	4	5	6	7	8	9

Array Representation



$$j = 2i + 1$$
$$k = 2i + 2$$
$$i = \text{floor}((j - 1)/ 2) = \text{floor}((k - 1)/ 2)$$

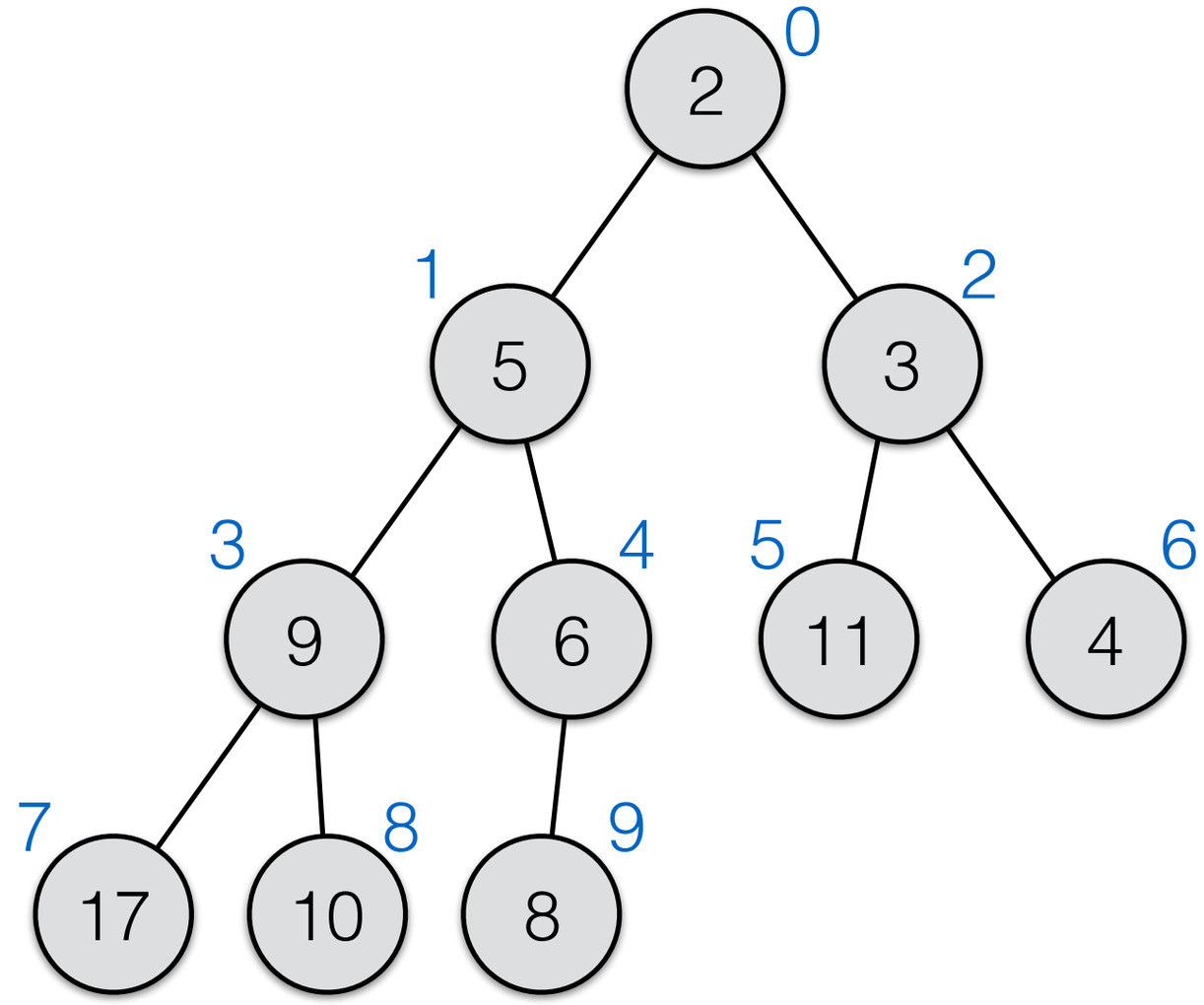
Array Representation



$$j = 2i + 1$$
$$k = 2i + 2$$
$$i = \text{floor}((j - 1)/2) = \text{floor}((k - 1)/2)$$

No need for links

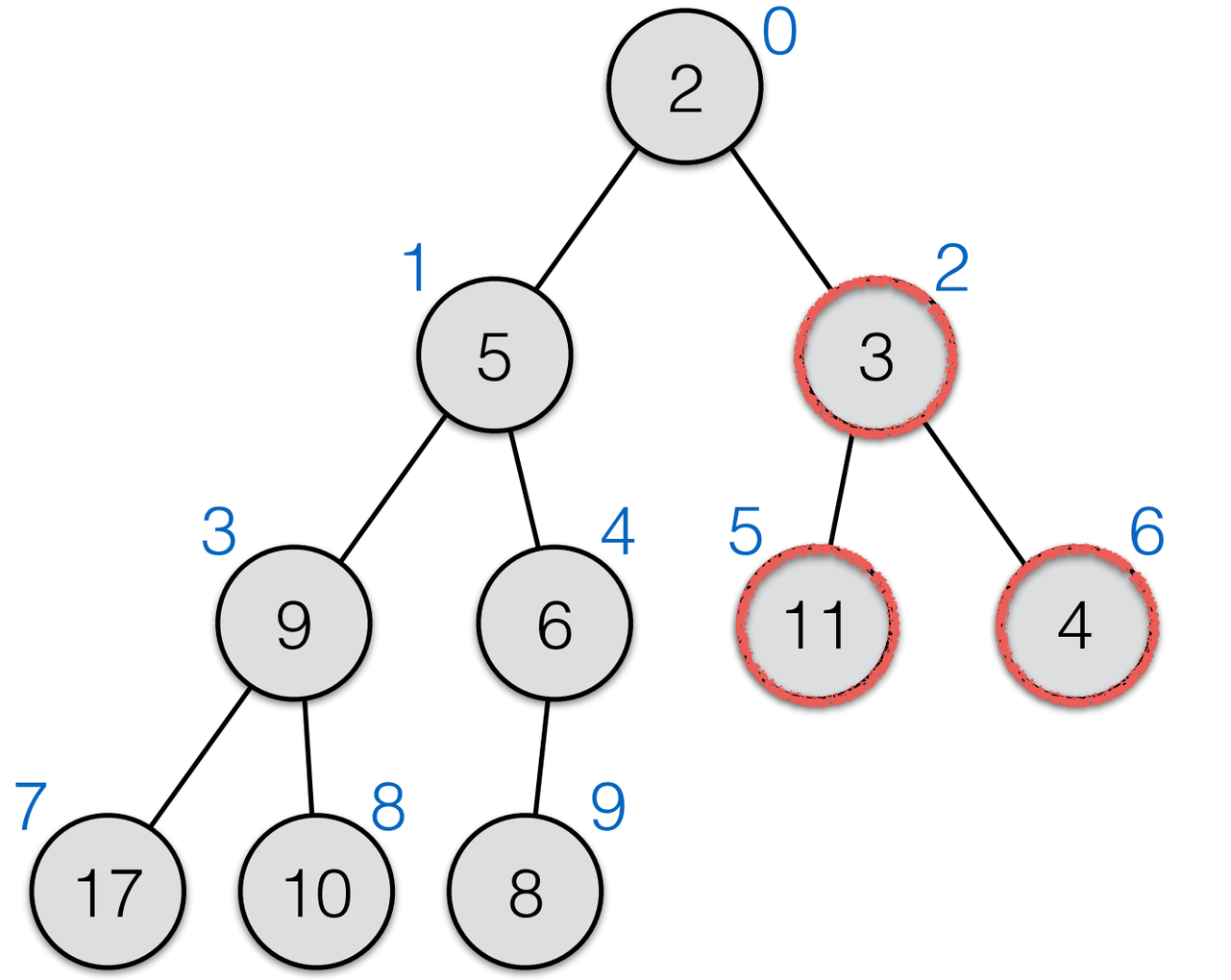
Array Representation

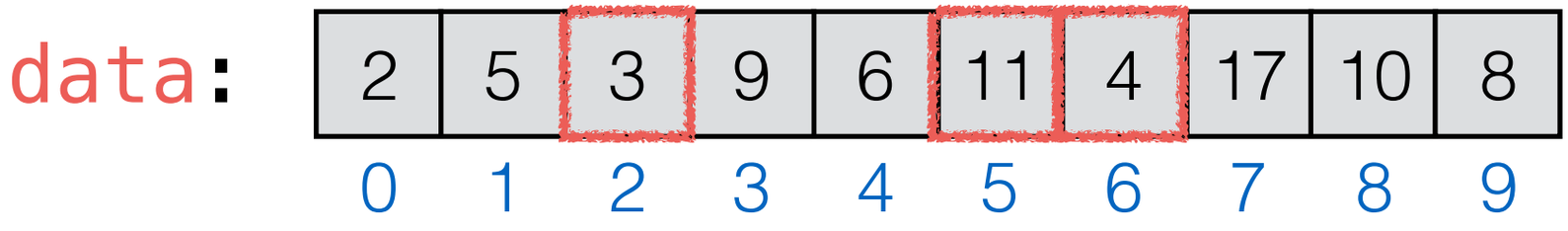


data:

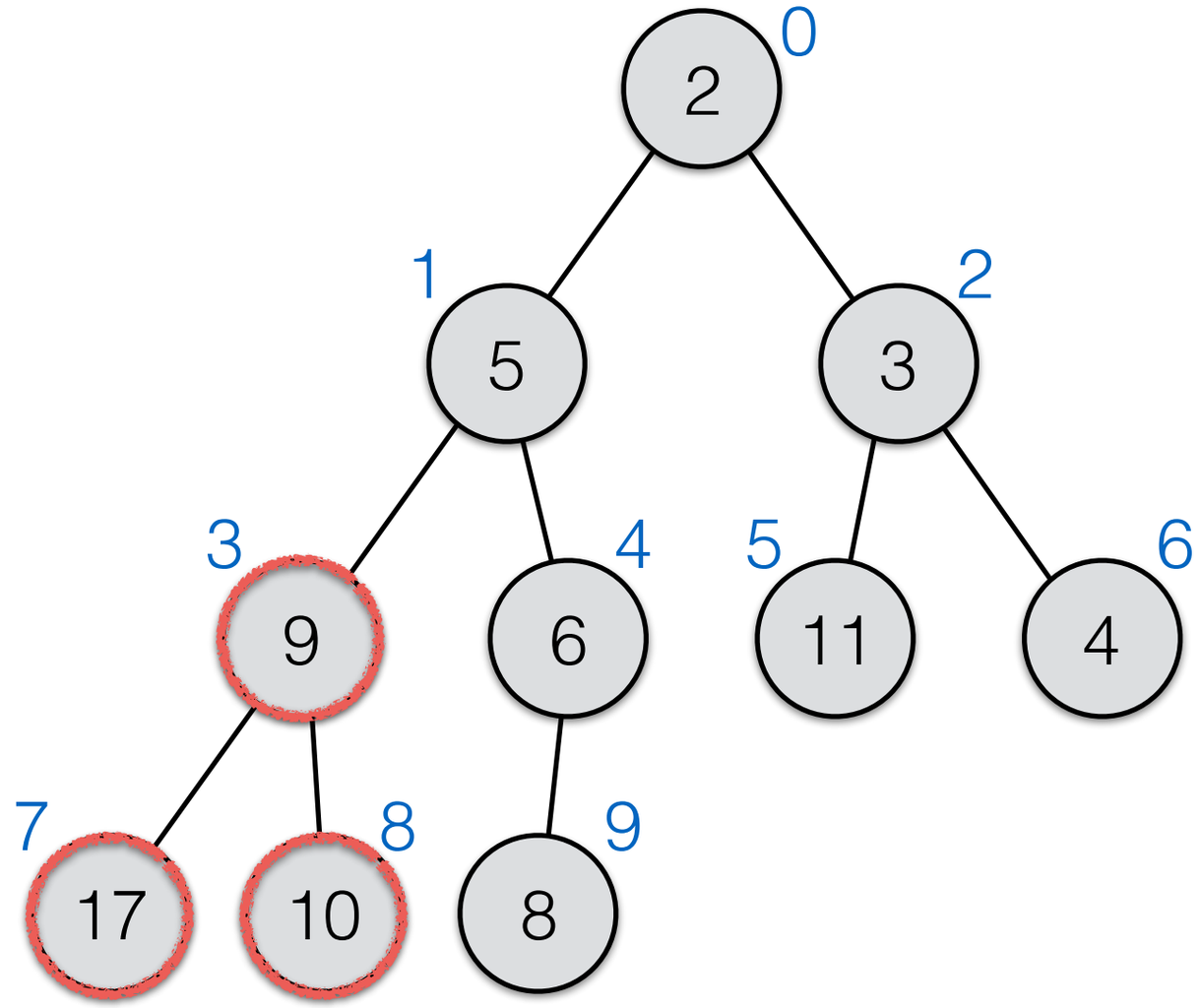
2	5	3	9	6	11	4	17	10	8
0	1	2	3	4	5	6	7	8	9

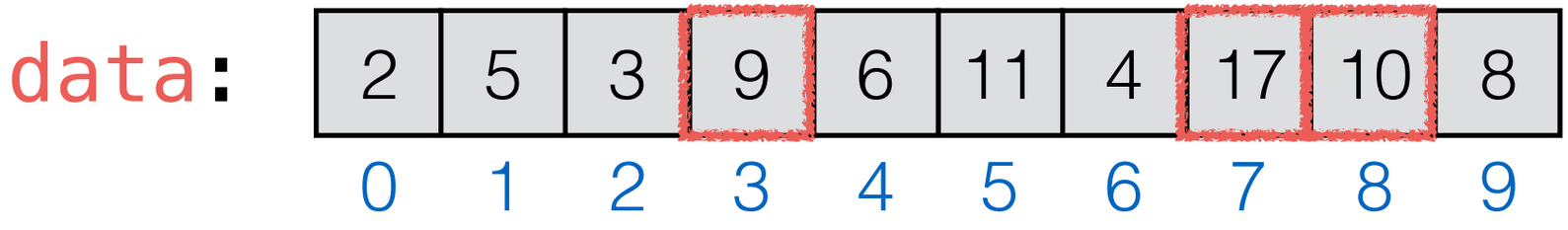
Array Representation





Array Representation





BinaryHeap

- On Blackboard

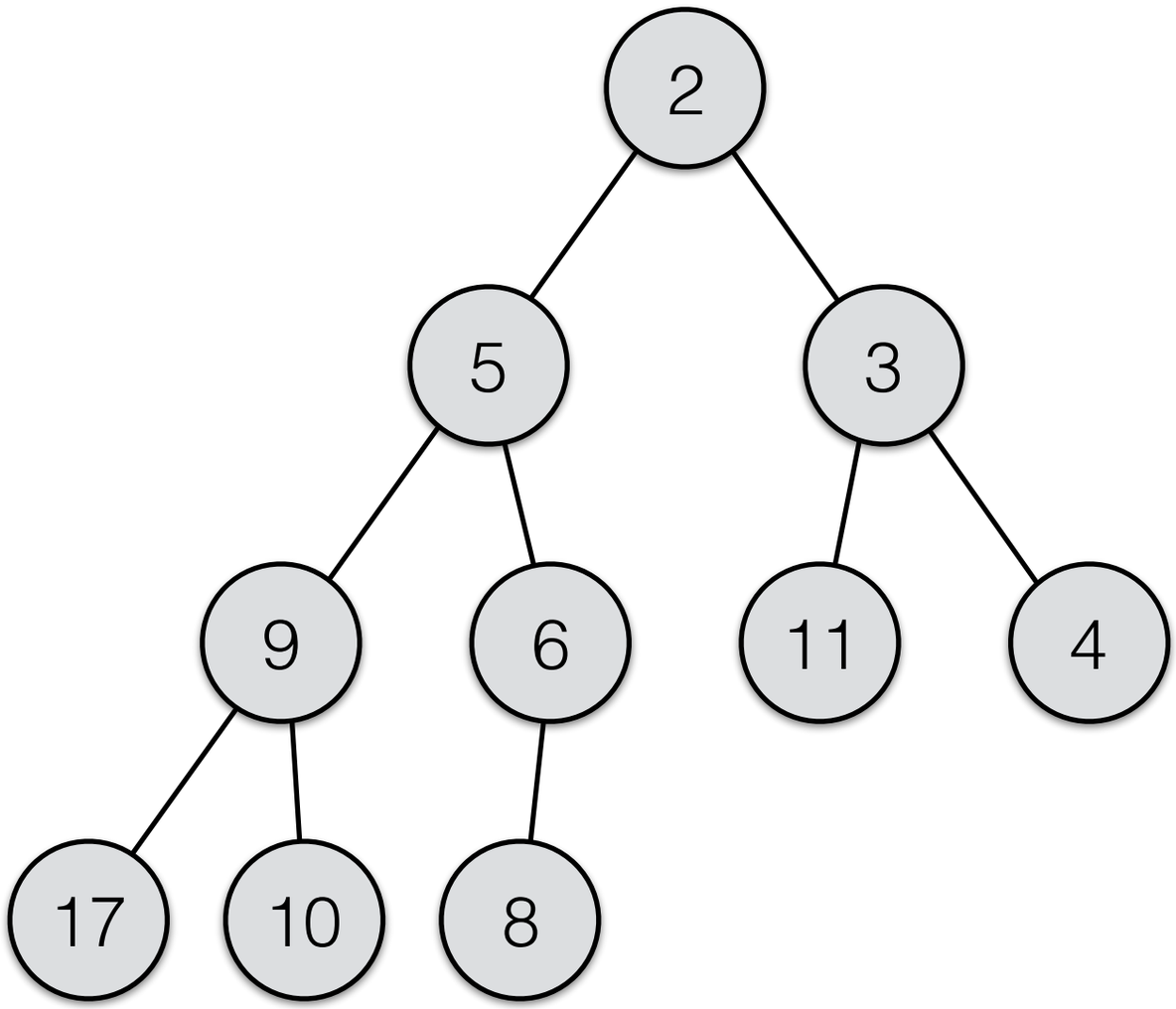
```
public class BinaryHeap<E extends
Comparable<? super E>>
```

- Methods
  - `E peek()`
  - `boolean add(E x)`
  - `E remove()`

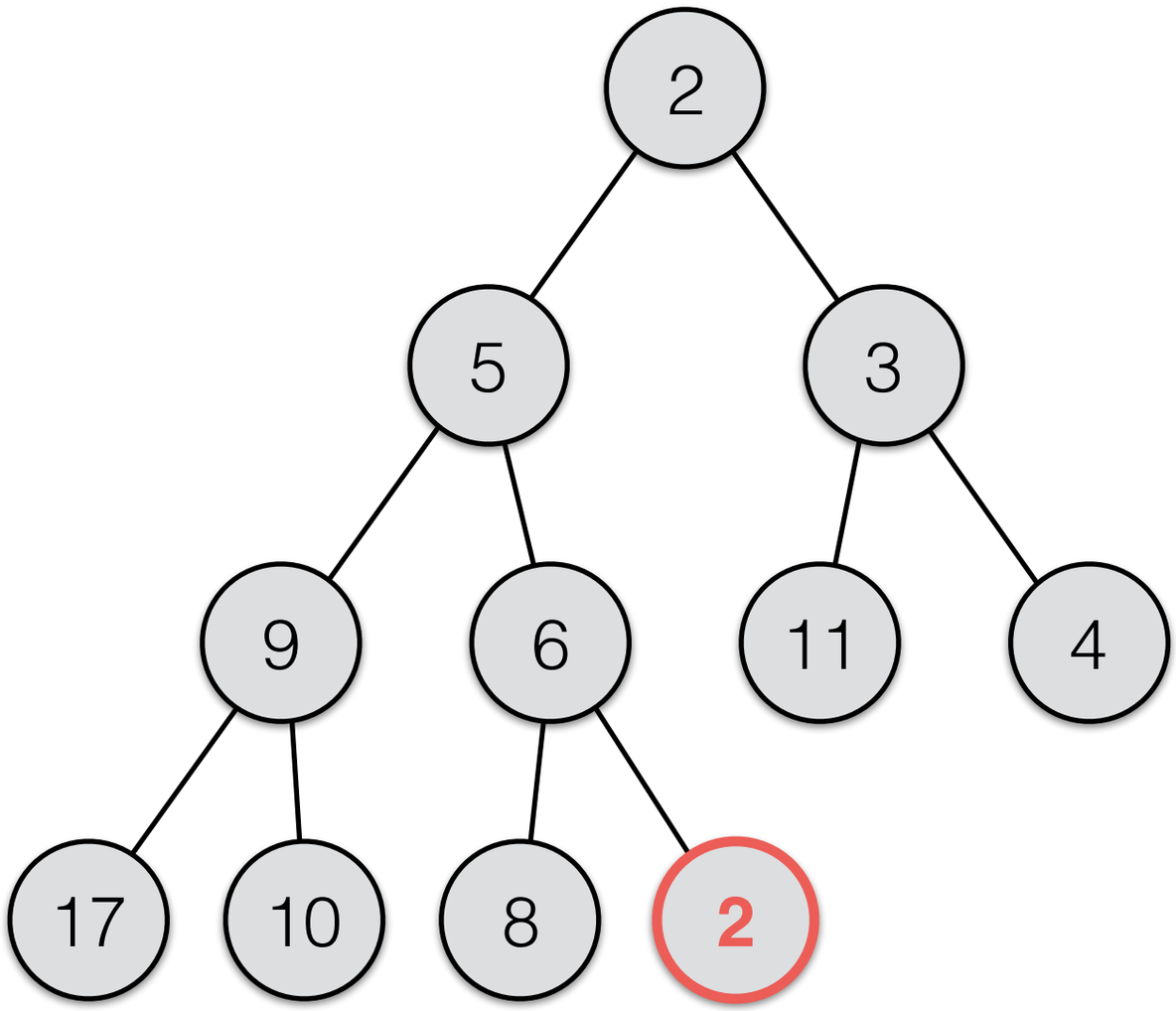
add(x)

1. Place x in bottom level of tree, at the first free spot from left.
   - In array-based implementation, place x in first available entry.
2. Percolate x up to restore heap-order property.

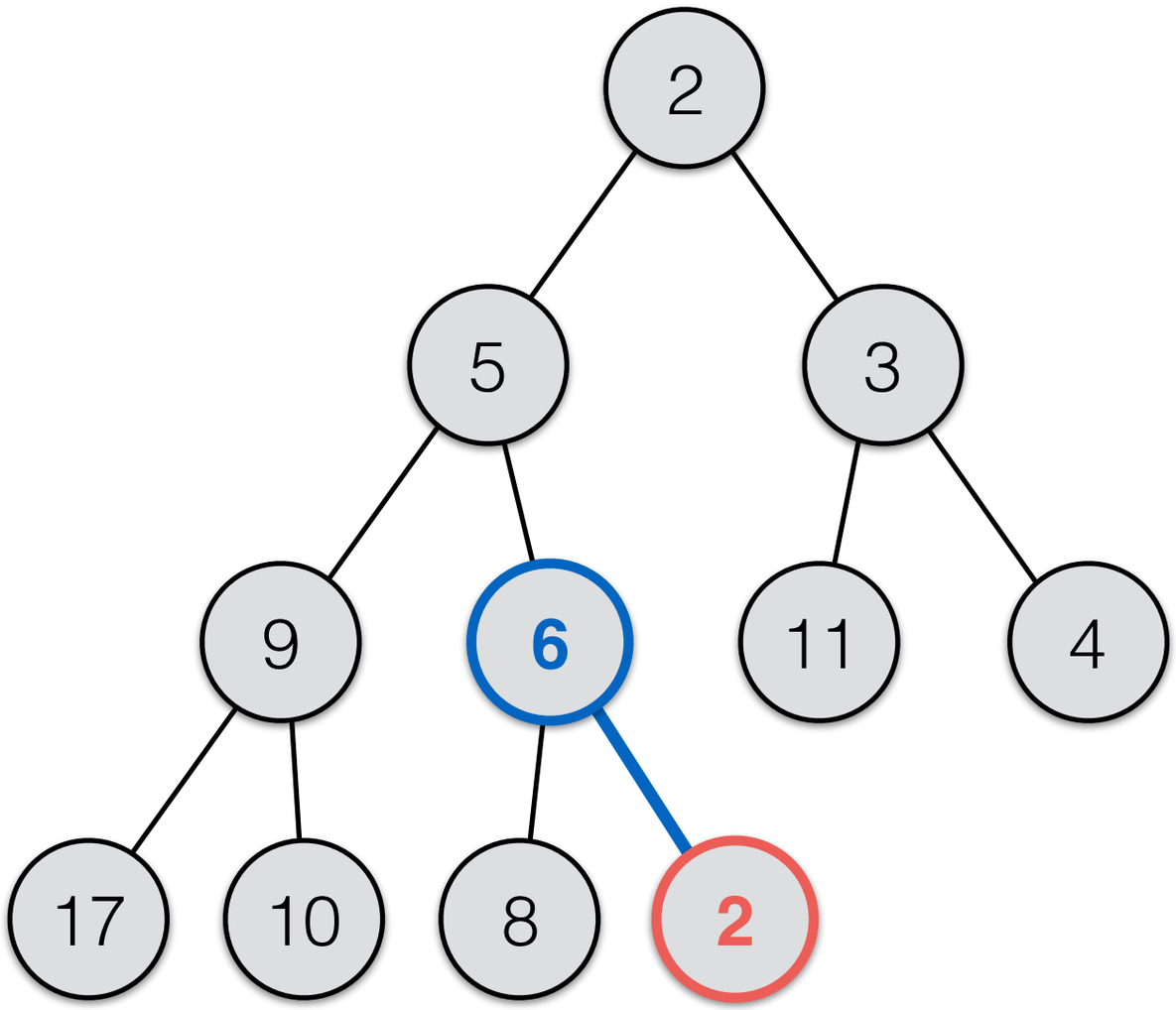
add(2)



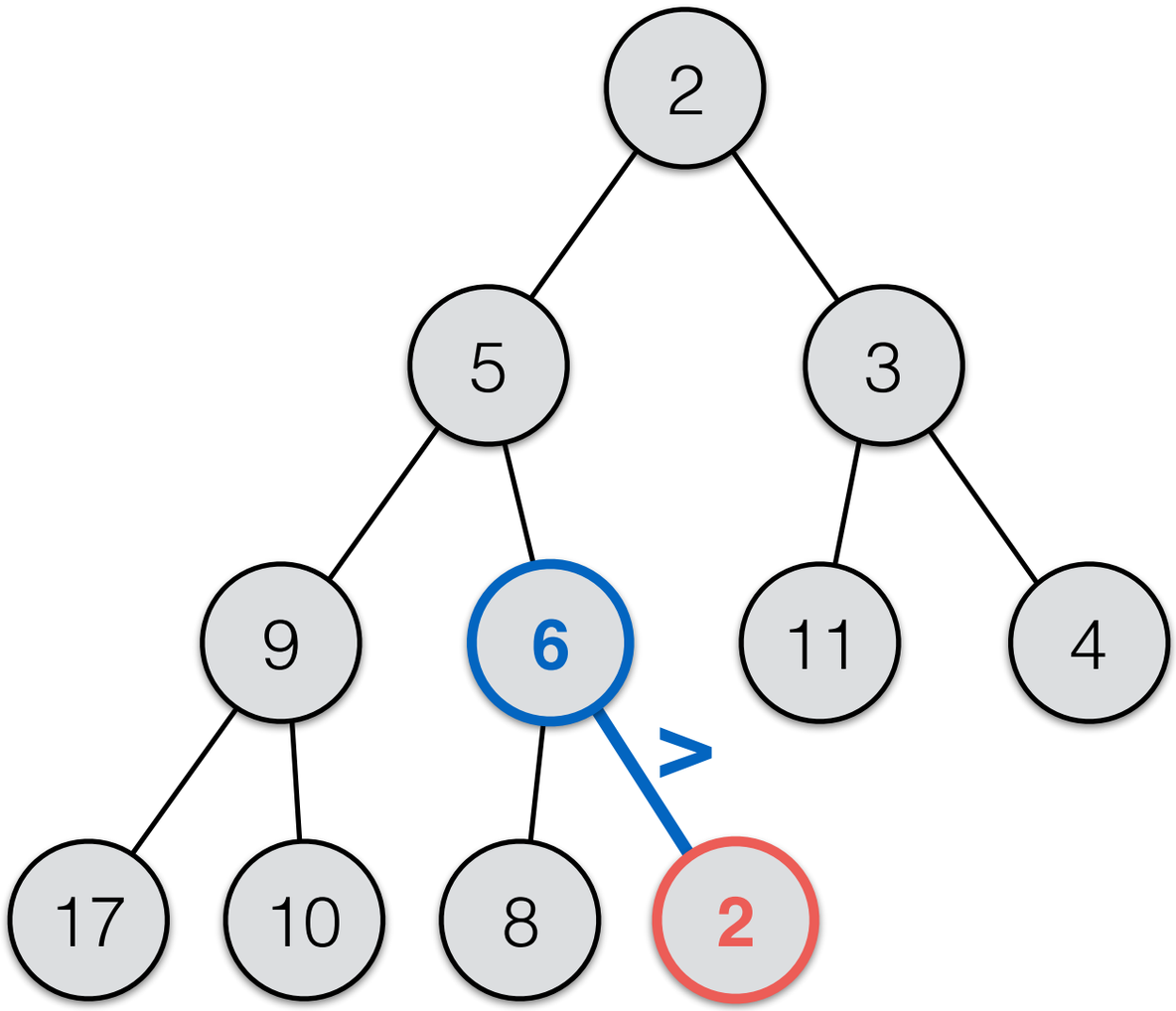
add(2)



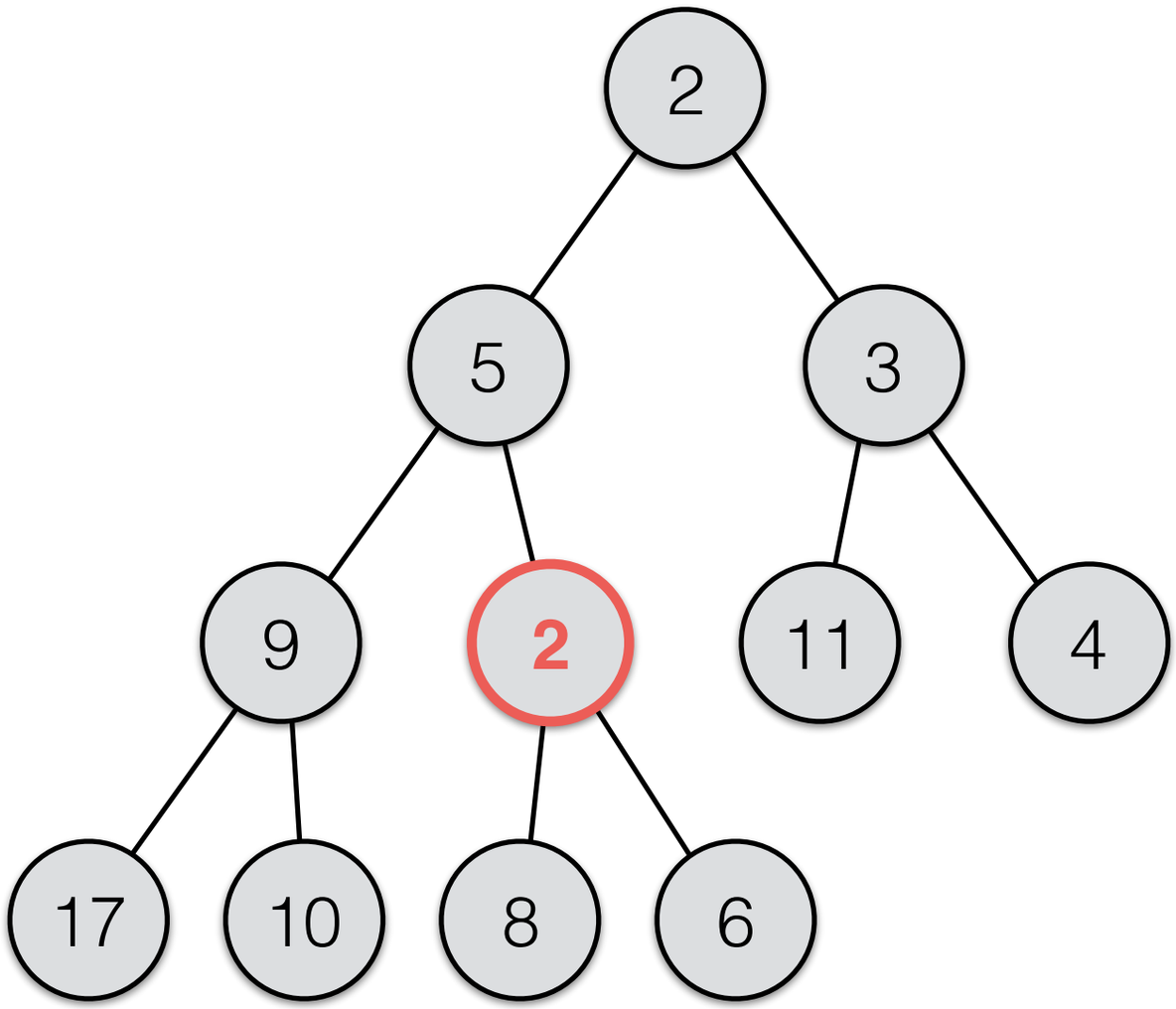
add(2)



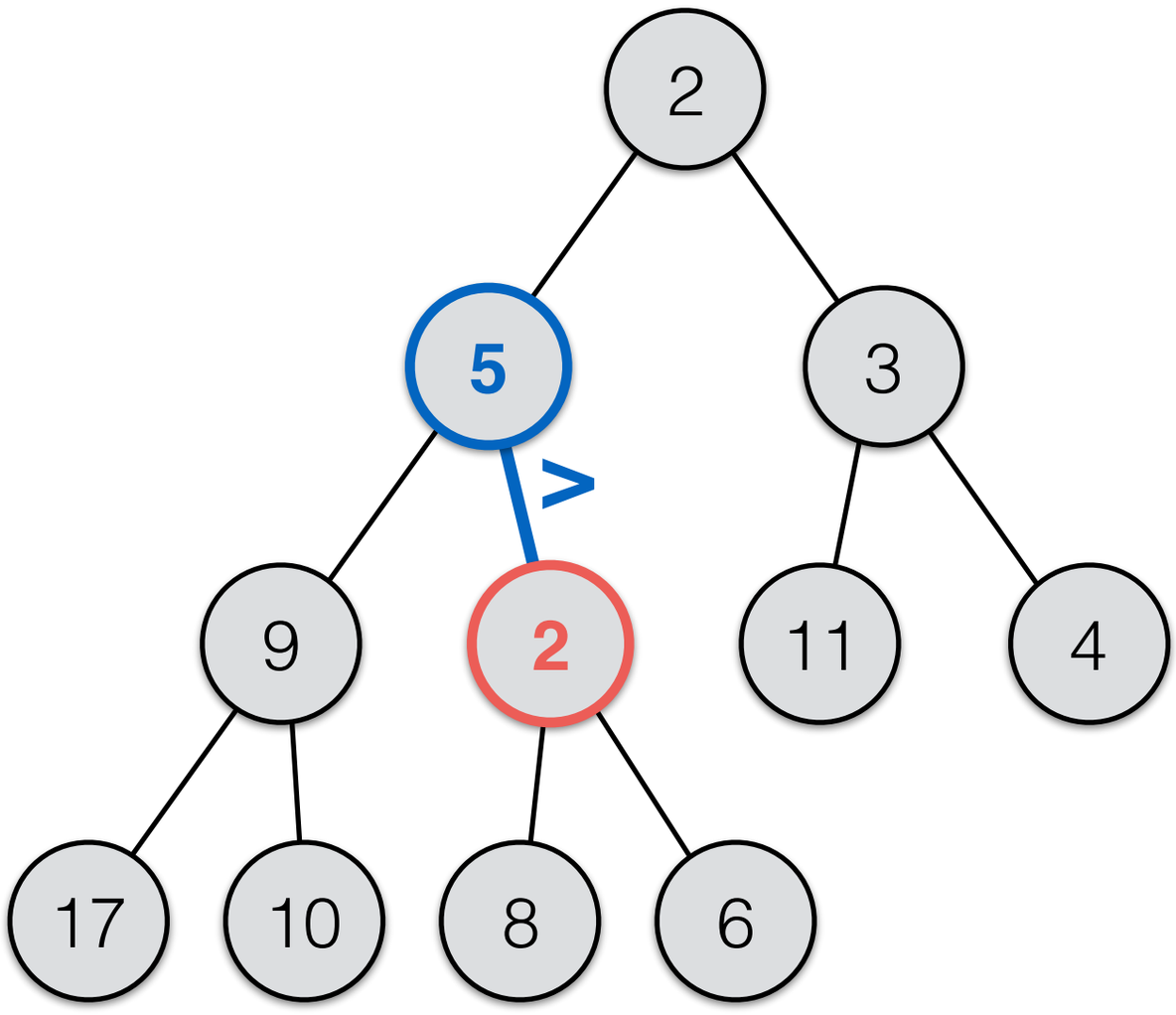
add(2)



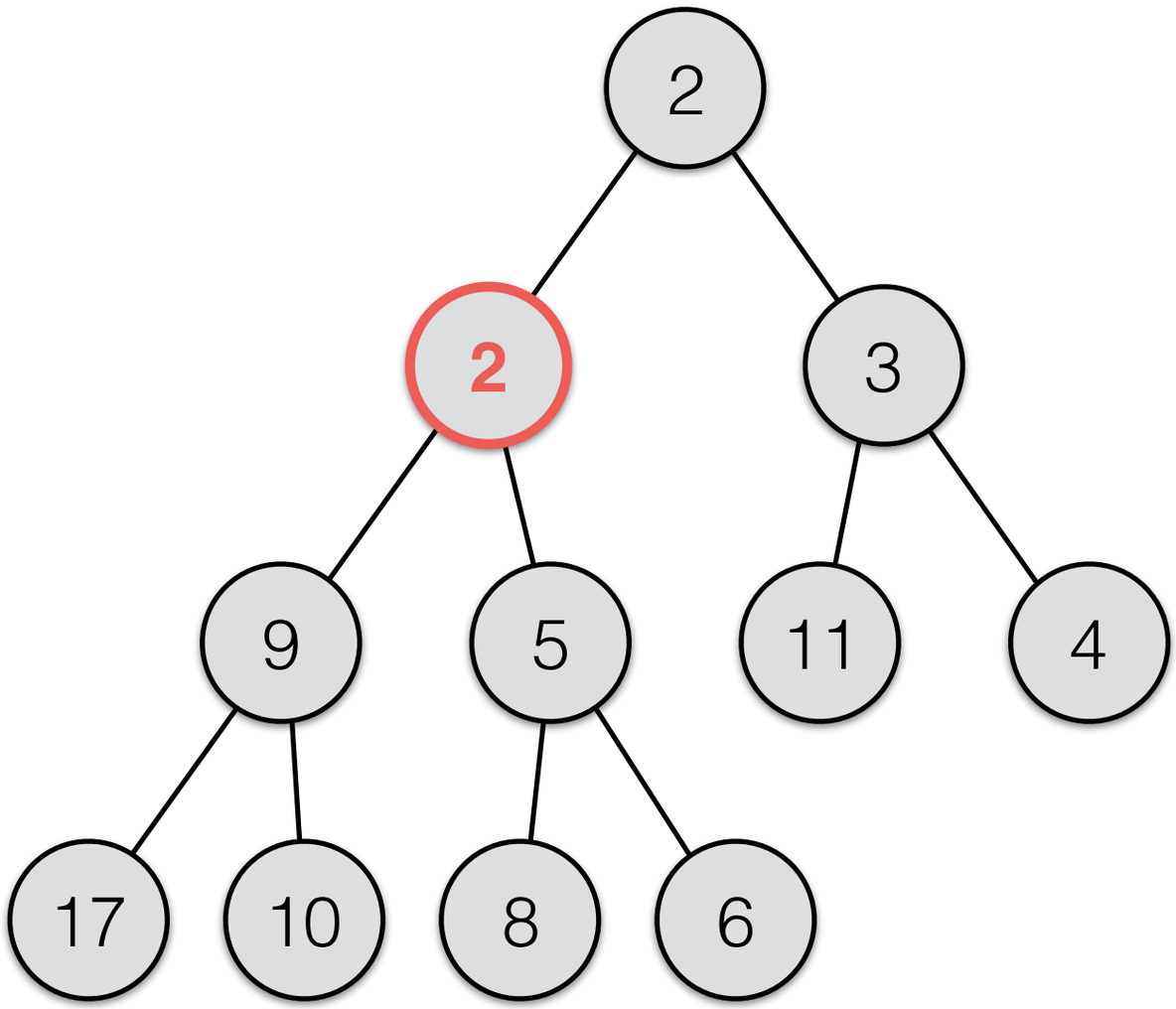
add(2)



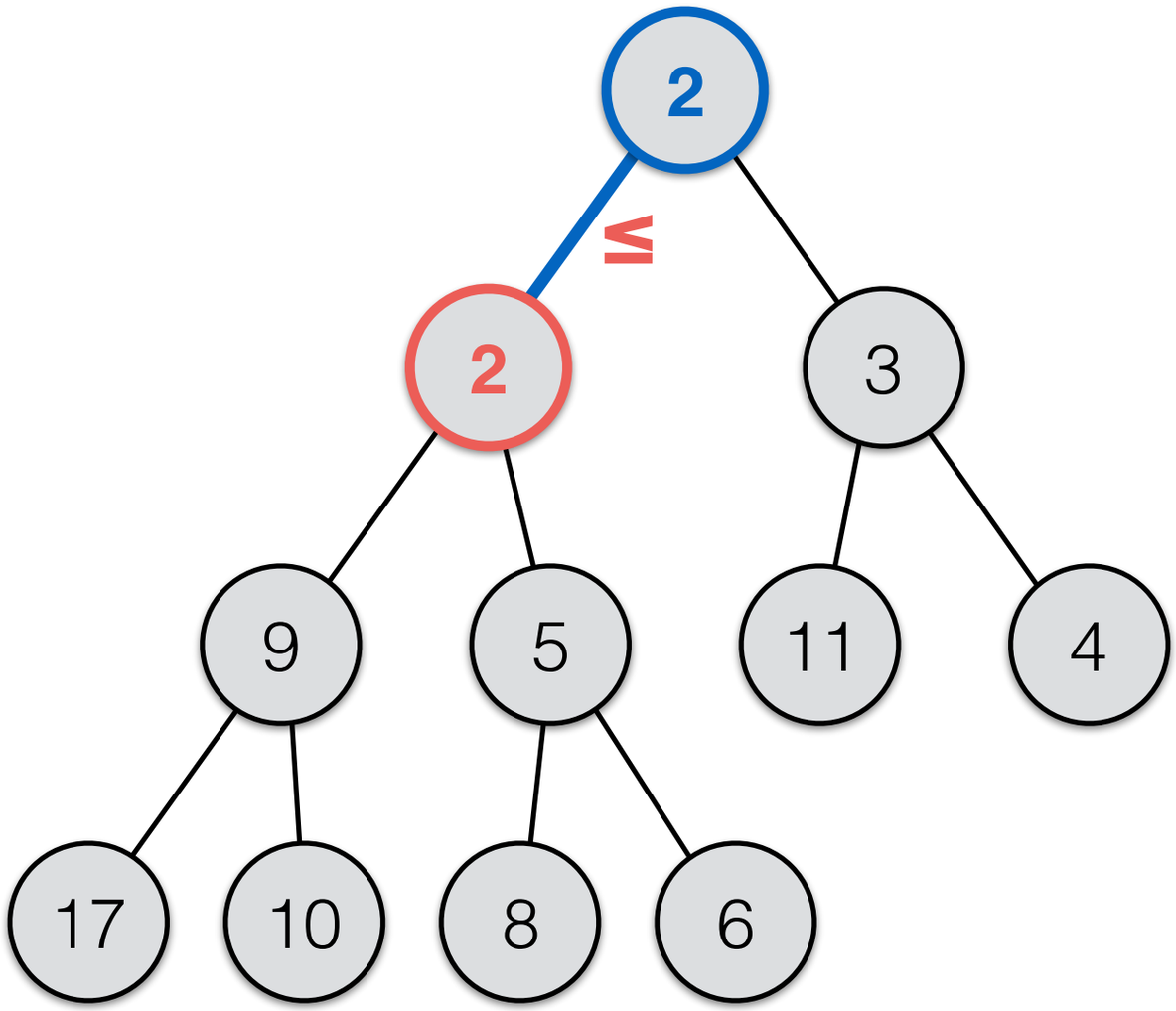
add(2)



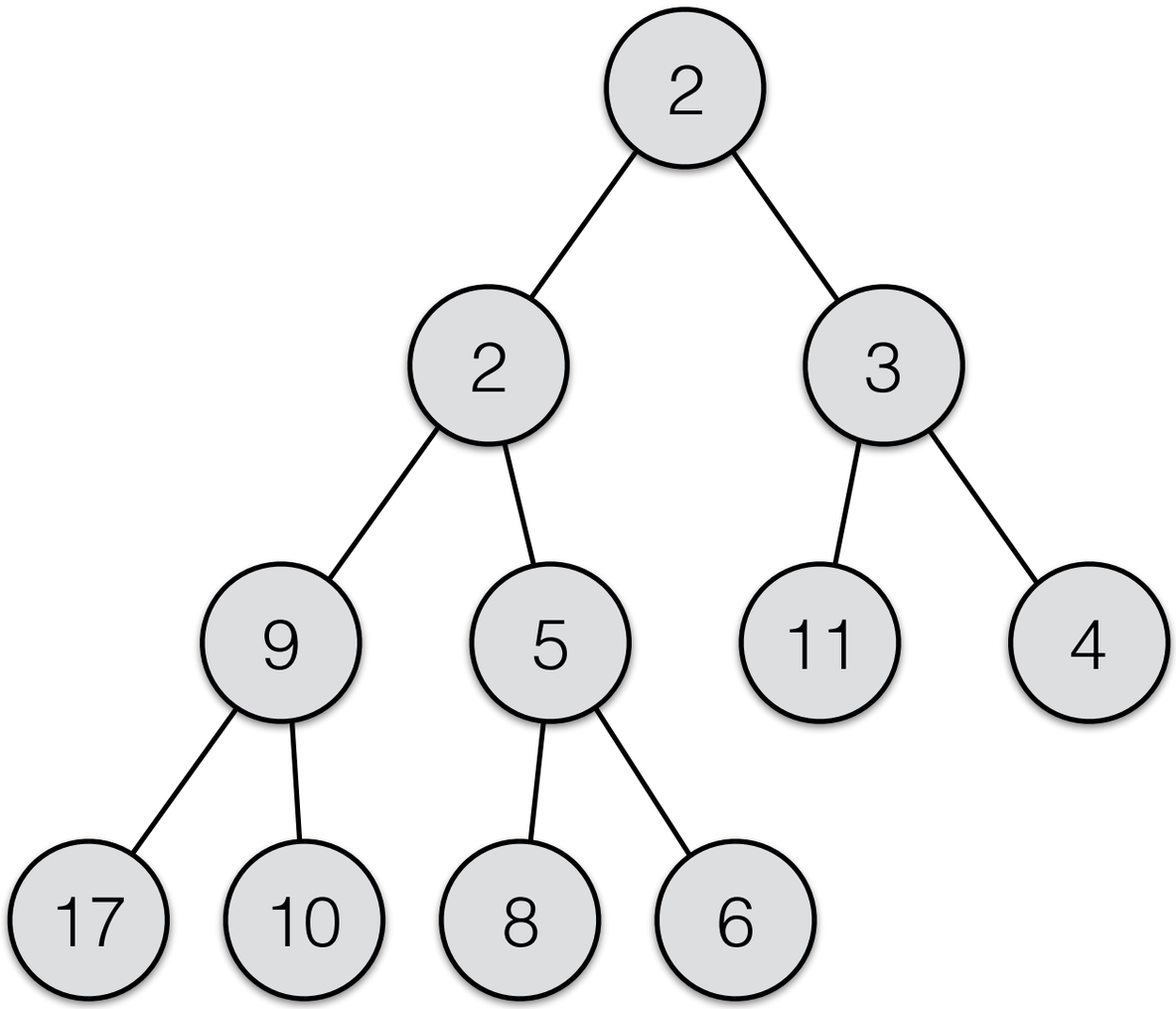
add(2)



add(2)

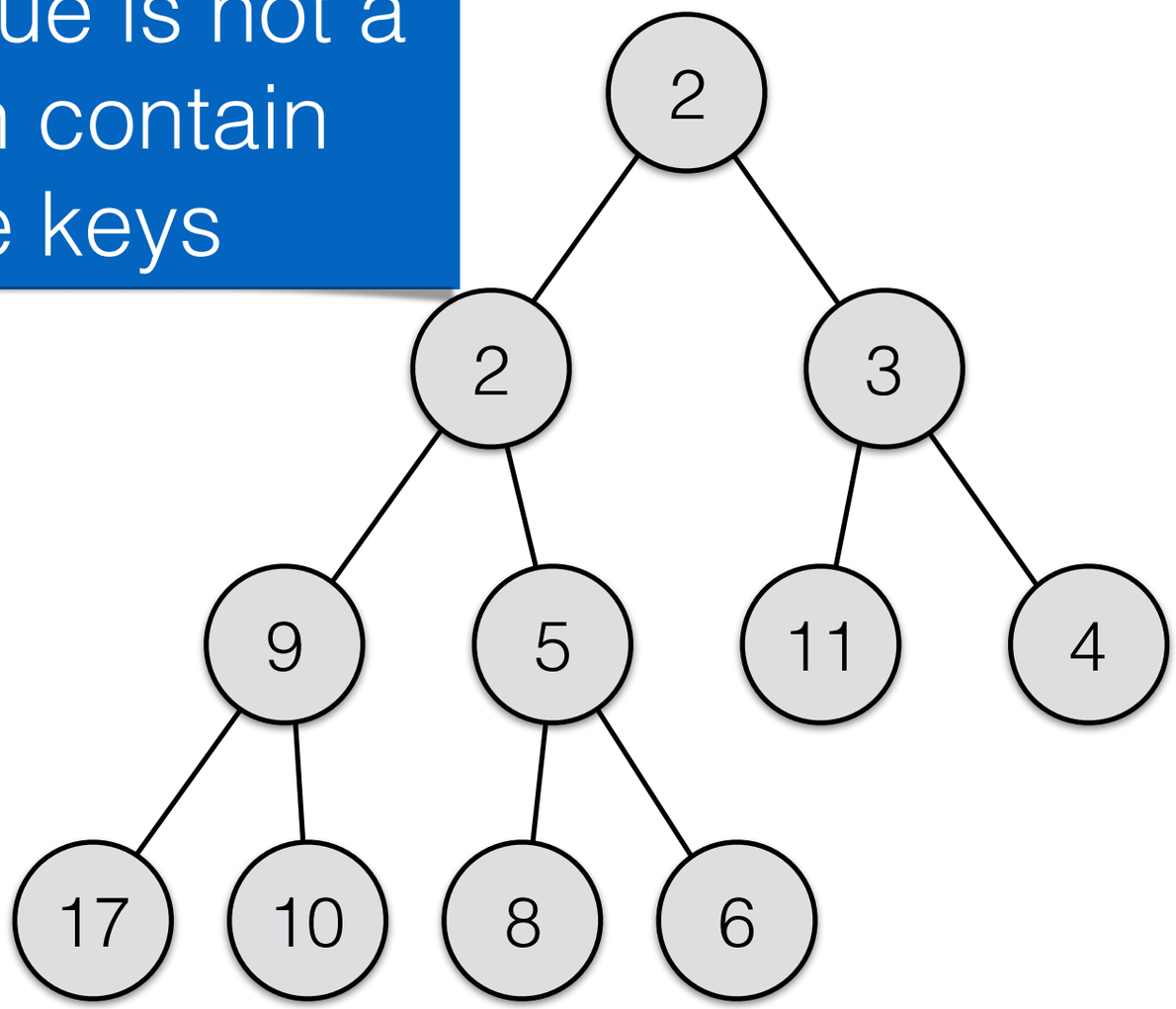


add(2)



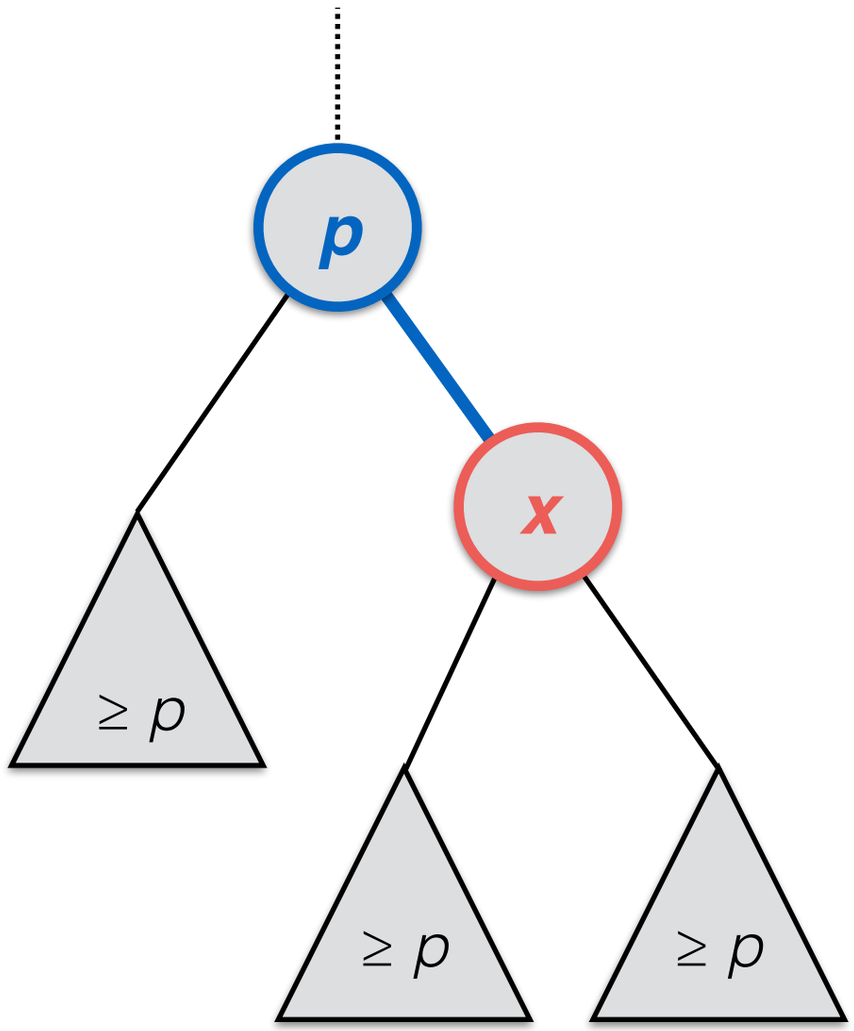
add(2)

A priority queue is not a set — it can contain duplicate keys

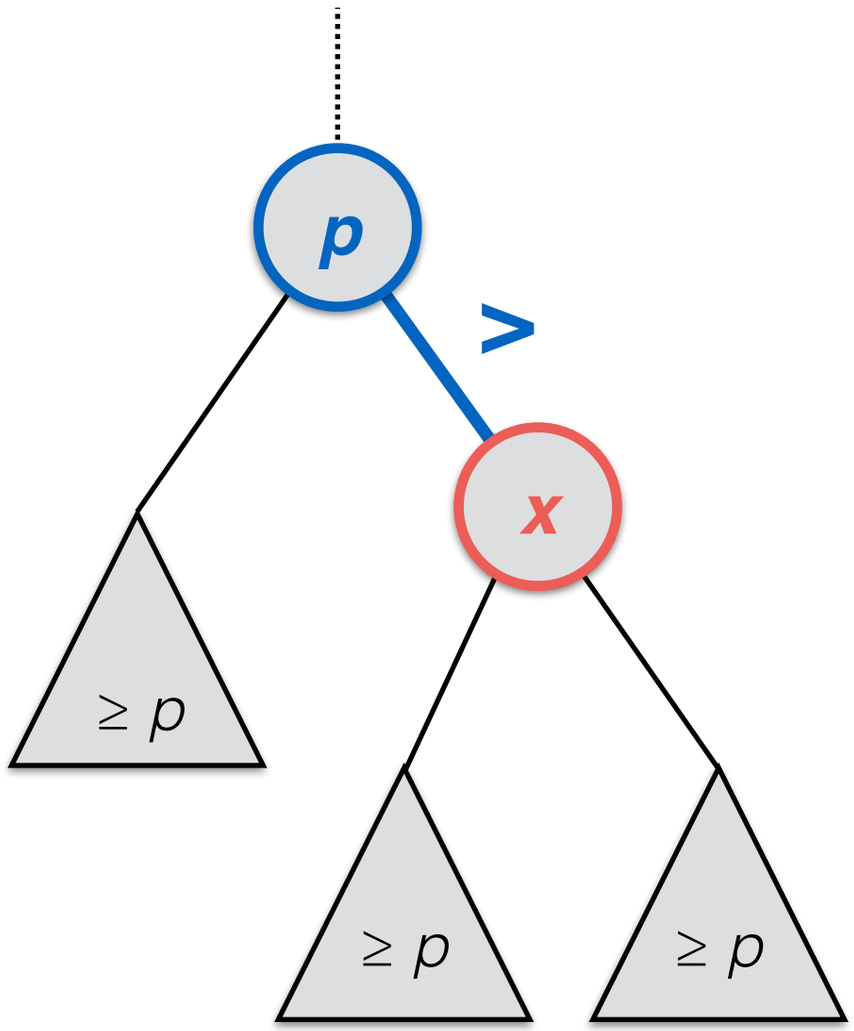


```
percolateUp(data,current):
    parent = (current - 1) / 2
    while (current > 0 && data[current] < data[parent])
        swap data[current] and data[parent]
        current = parent
        parent = (current - 1) / 2
```

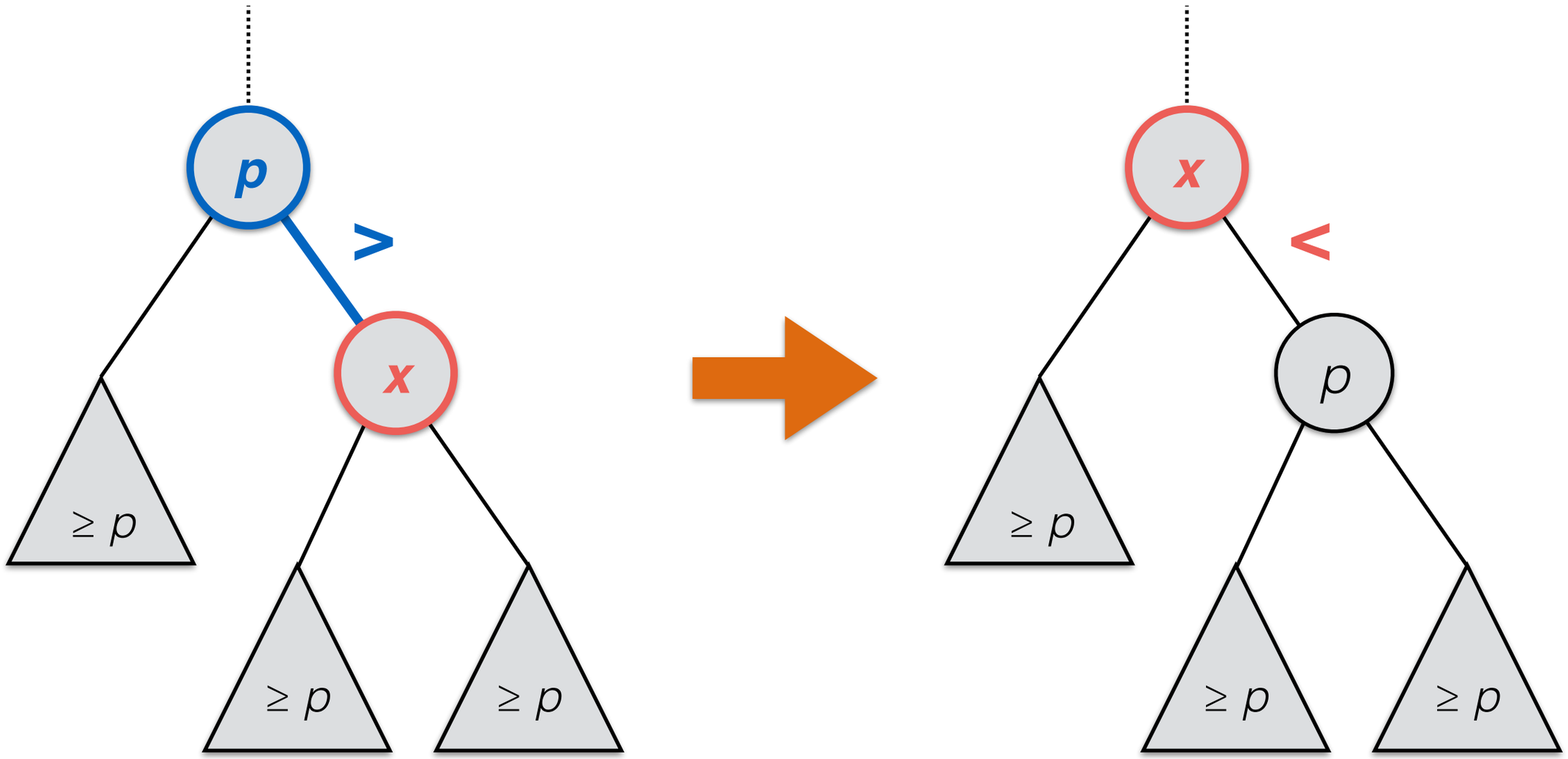
Correctness of `percolateUp()`



Correctness of `percolateUp()`



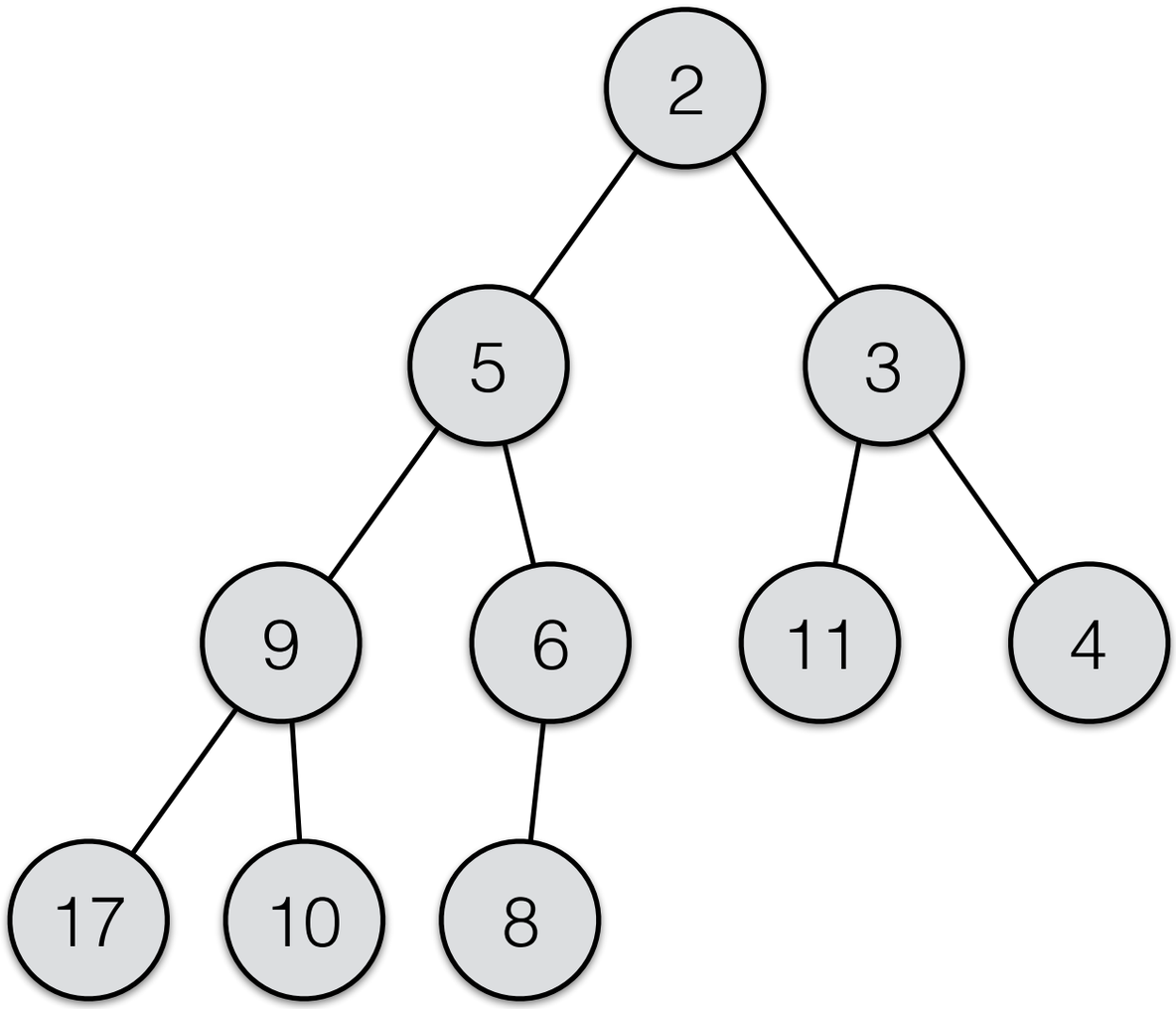
Correctness of `percolateUp()`



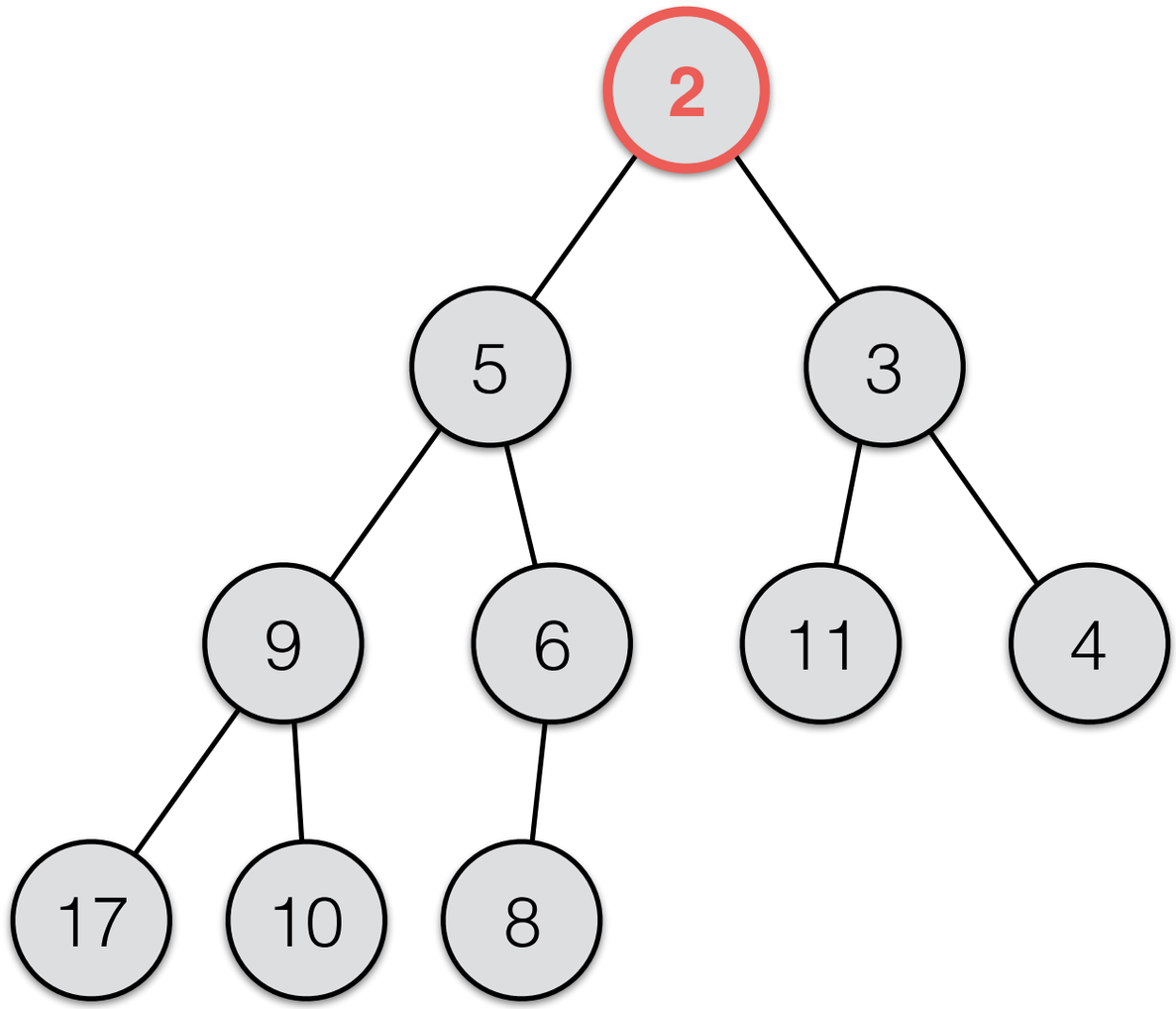
remove()

1. Save entry at root for return value.
2. Replace root entry by last entry in heap — call this entry x.
3. Percolate x down to restore heap order.
4. Return value saved in Step 1.

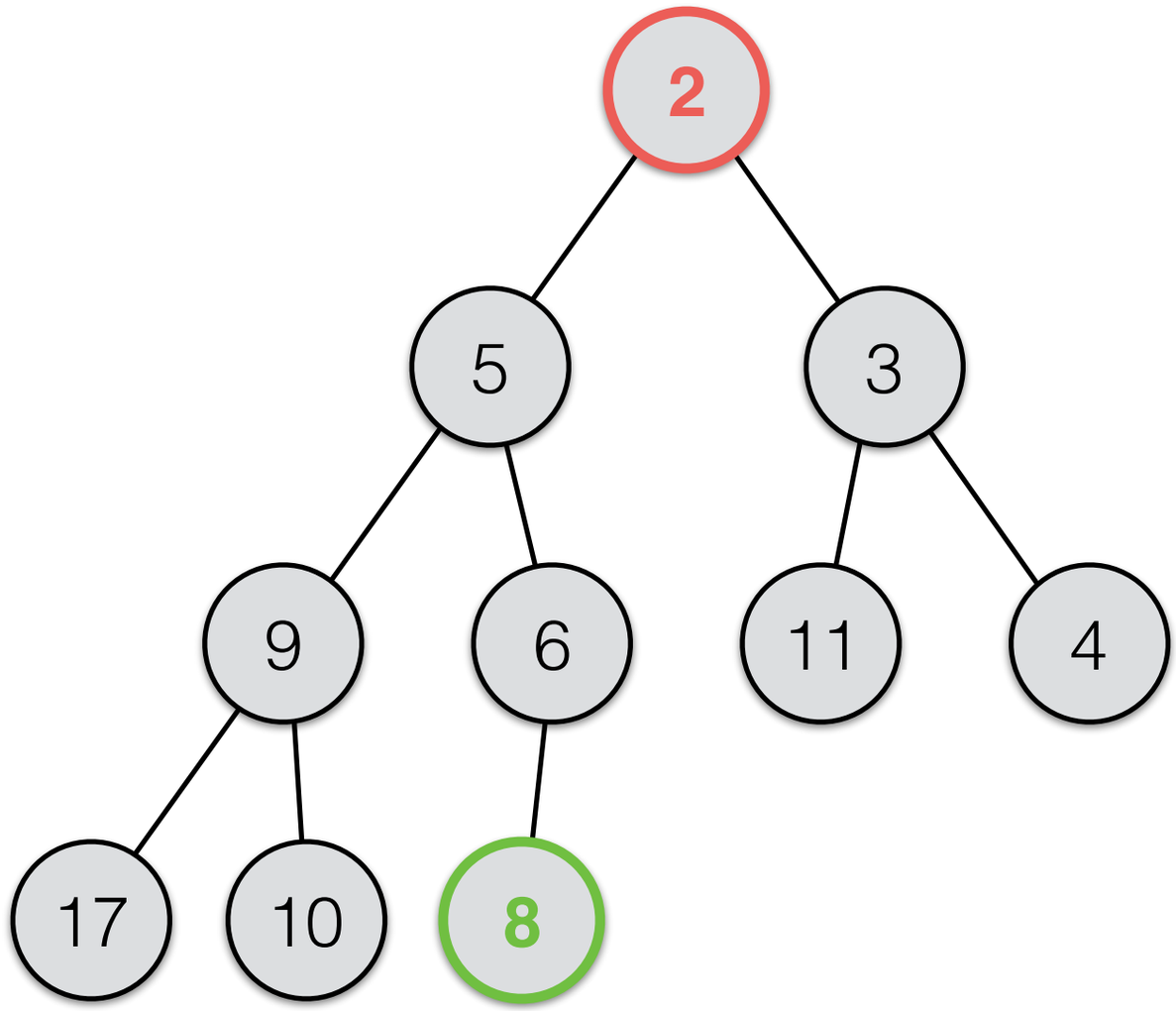
remove()



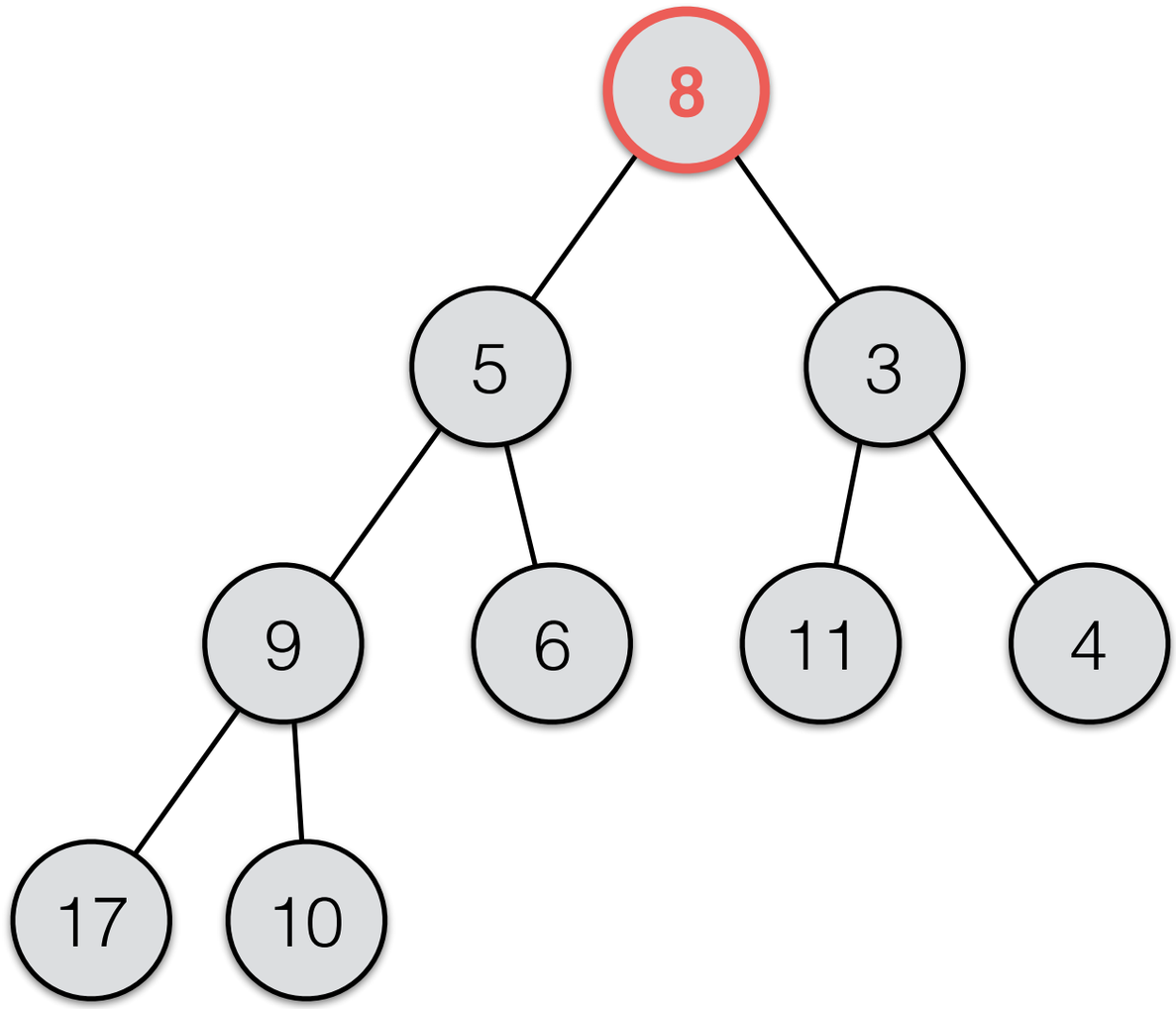
remove()



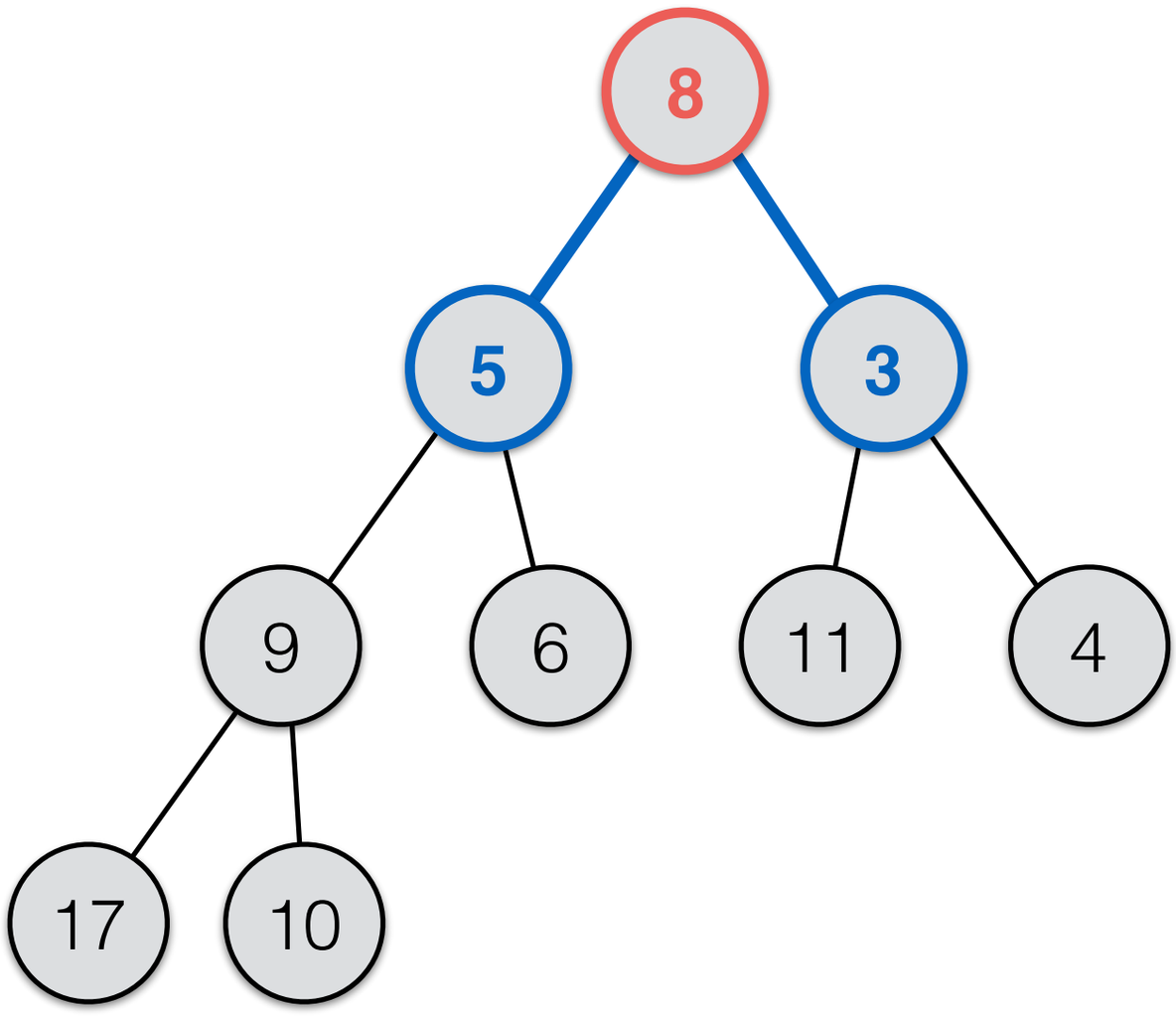
remove()



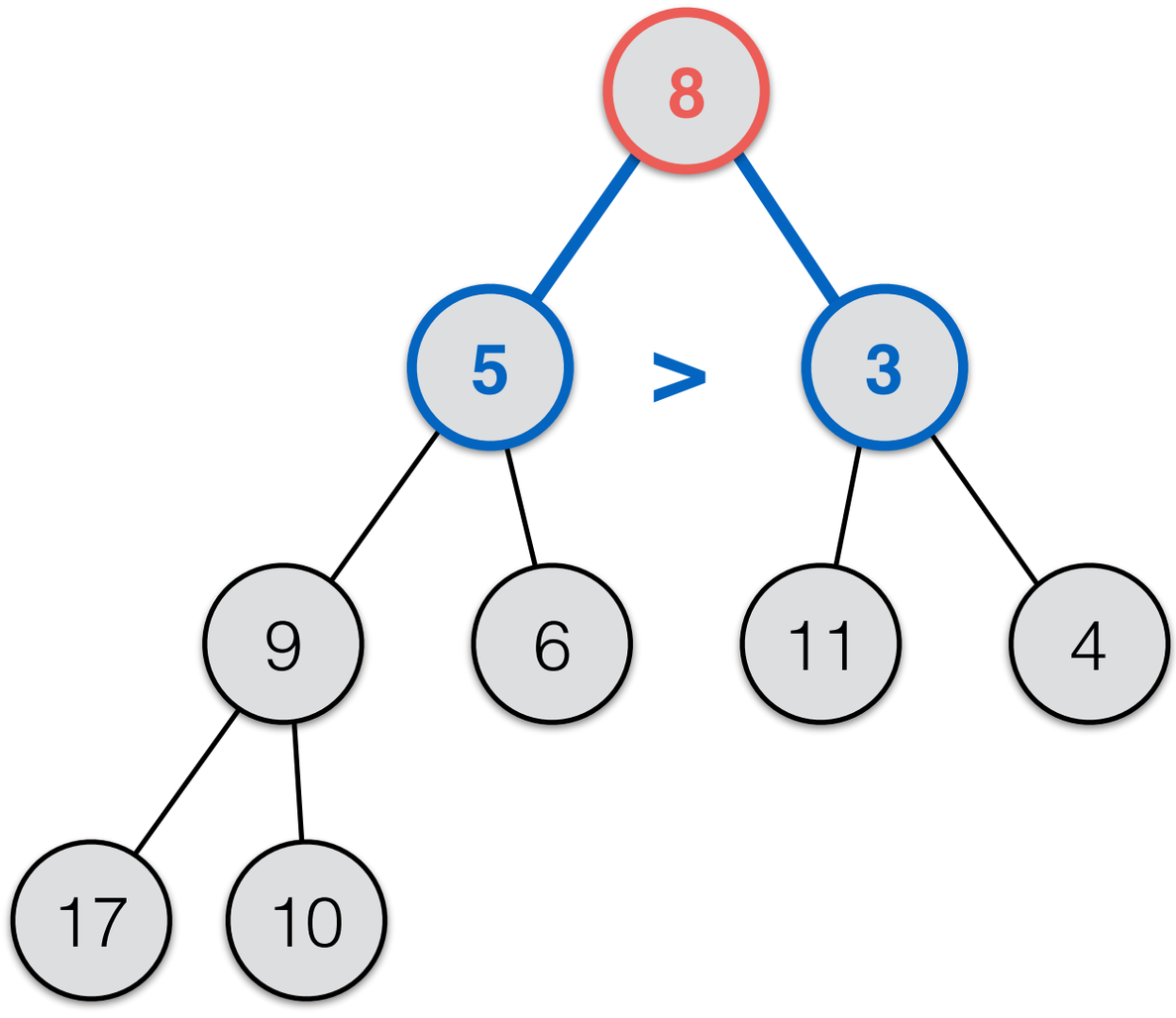
remove()



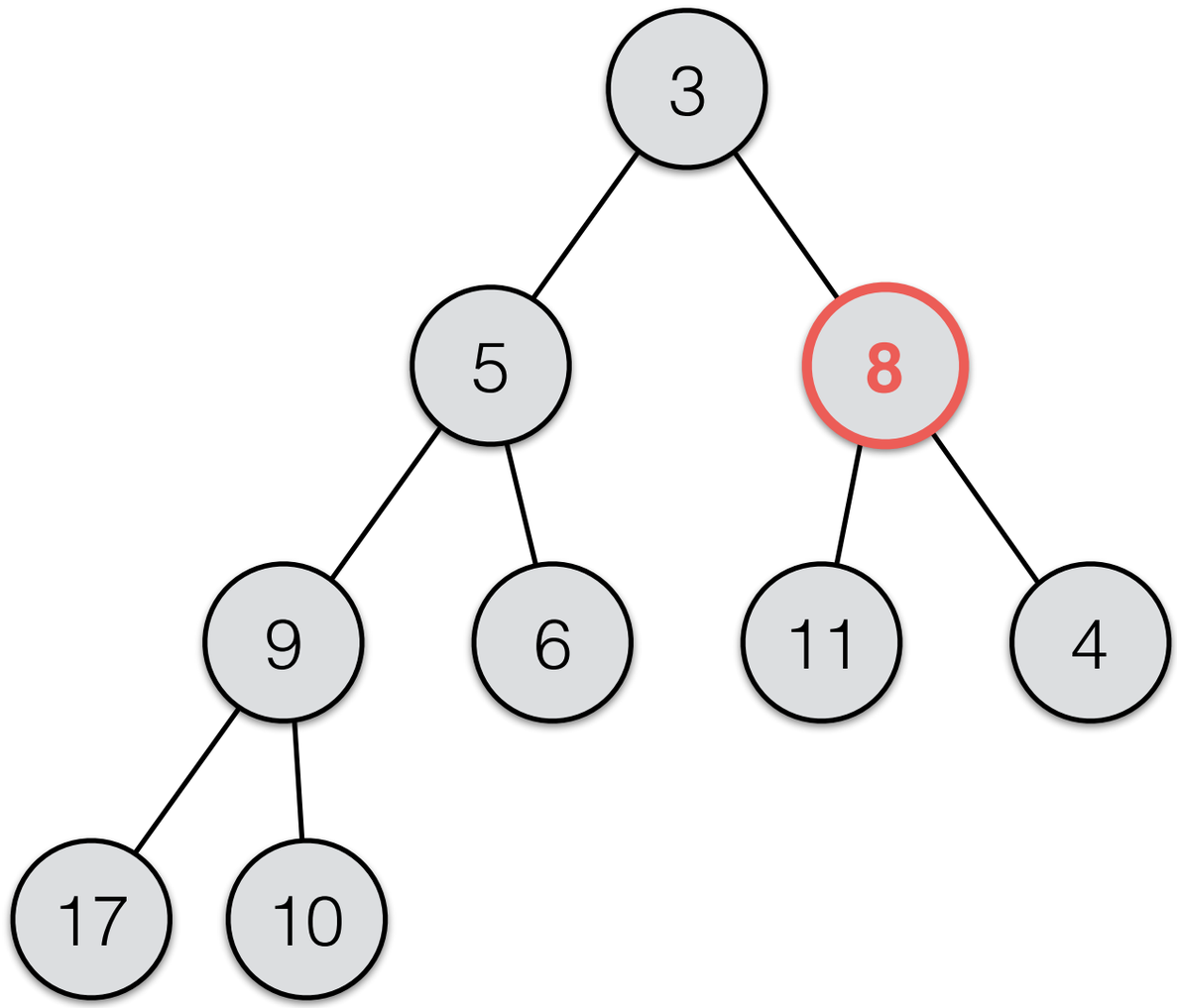
remove()



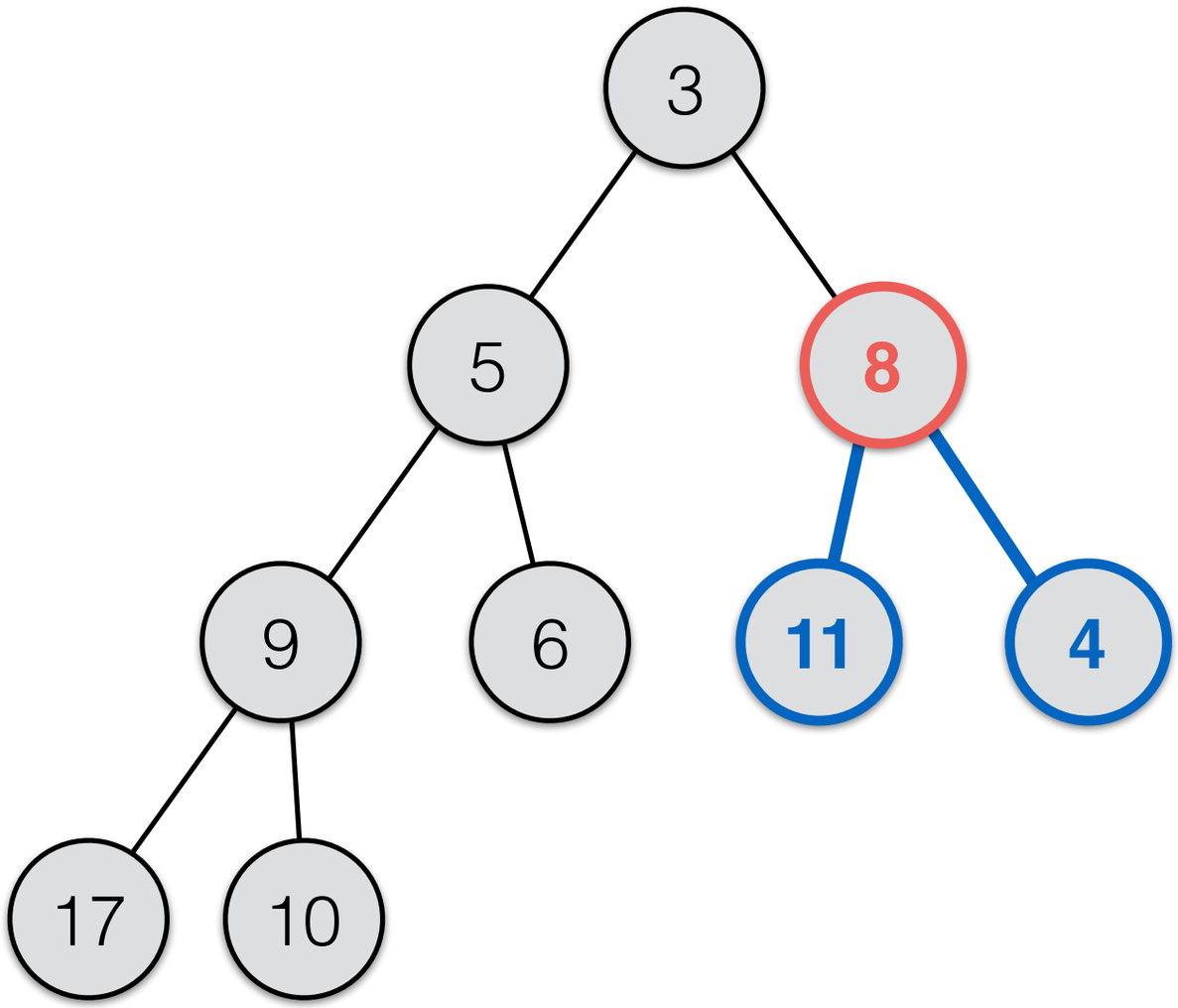
remove()



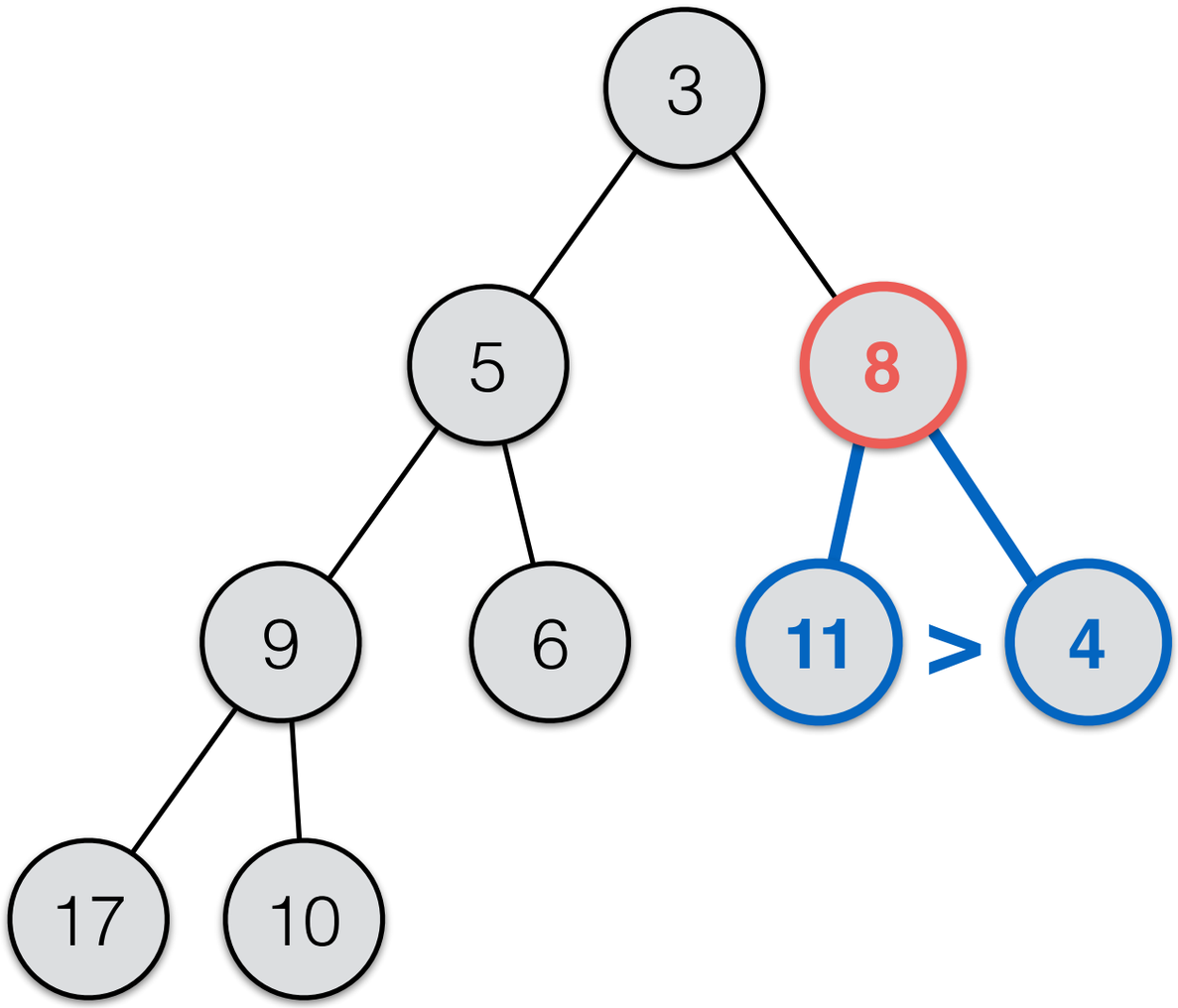
remove()



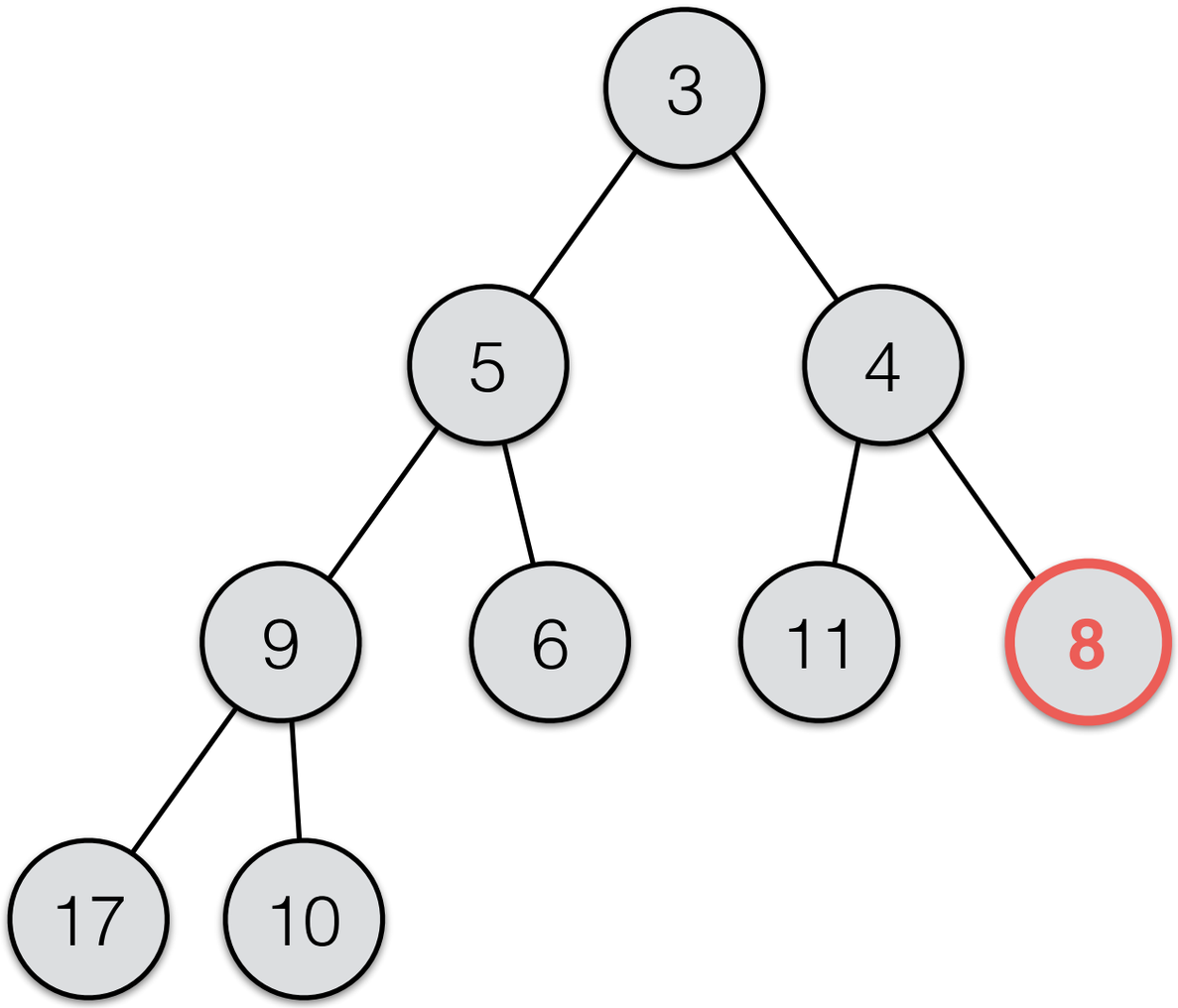
remove()



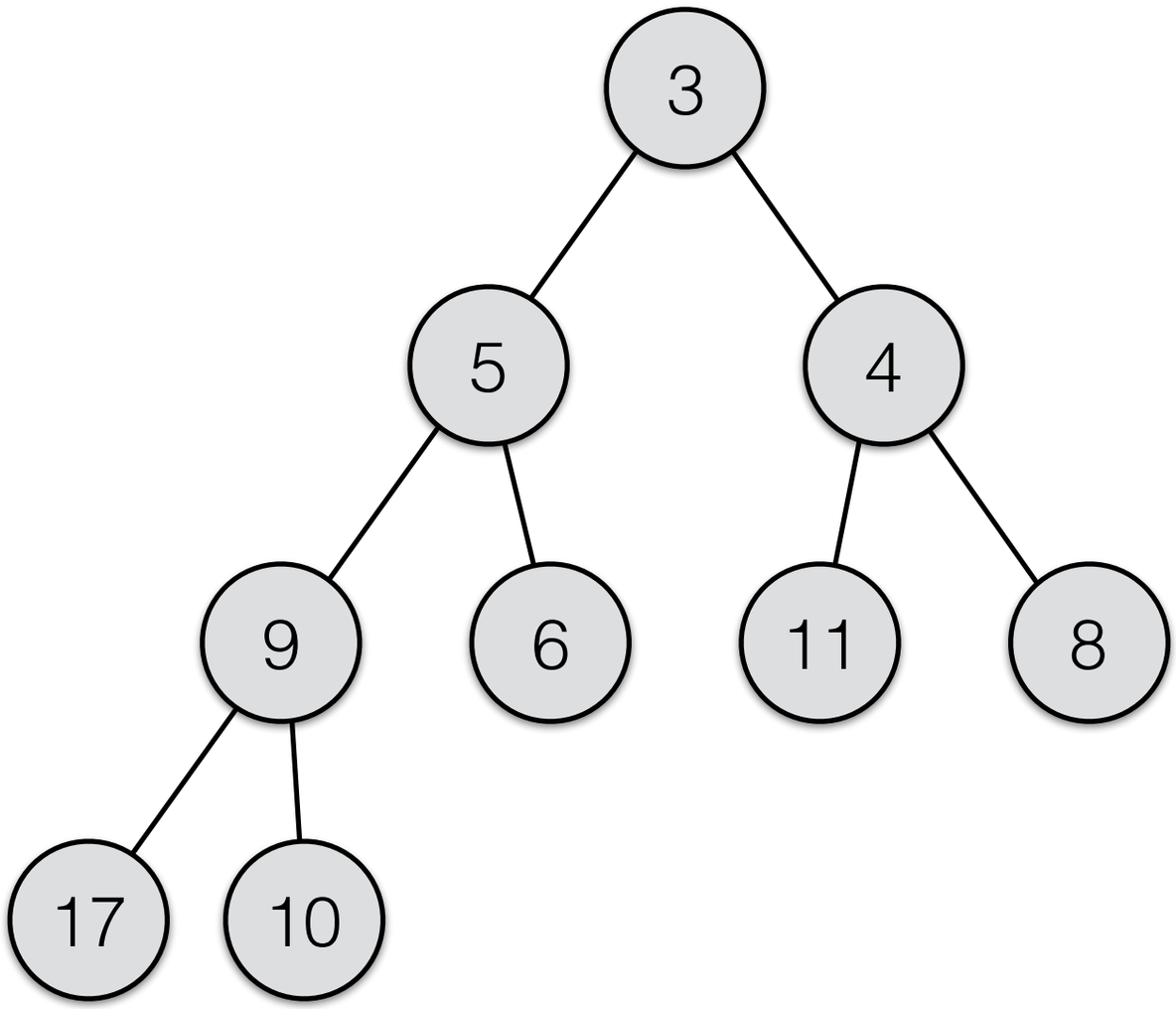
remove()



remove()



remove()



```
percolateDown(data,current):
    // Find left child of current
    child = 2 * current + 1

    while (child < size)
        if child + 1 < size
            // Find smaller of two children
            if data[child] > data[child + 1]
                child = child + 1
        if data[current] > data[child]
            swap data[current] and data[child]
        current = child
        child = 2 * current + 1
```

Time Complexity

Time Complexity

Fact 1.
`add()` and `remove()` take O(height) time

Time Complexity

Fact 1.
`add()` and `remove()` take O(height) time

Fact 2 (Height Bound).
The height of an n-node heap is $\leq \log_2 n$.

Time Complexity

Fact 1.
`add()` and `remove()` take O(height) time

Fact 2 (Height Bound).
The height of an n-node heap is $\leq \log_2 n$.

$\Rightarrow$ `add()` and `remove()` take $O(\log_2 n)$ time

Building a Heap

Building a Heap

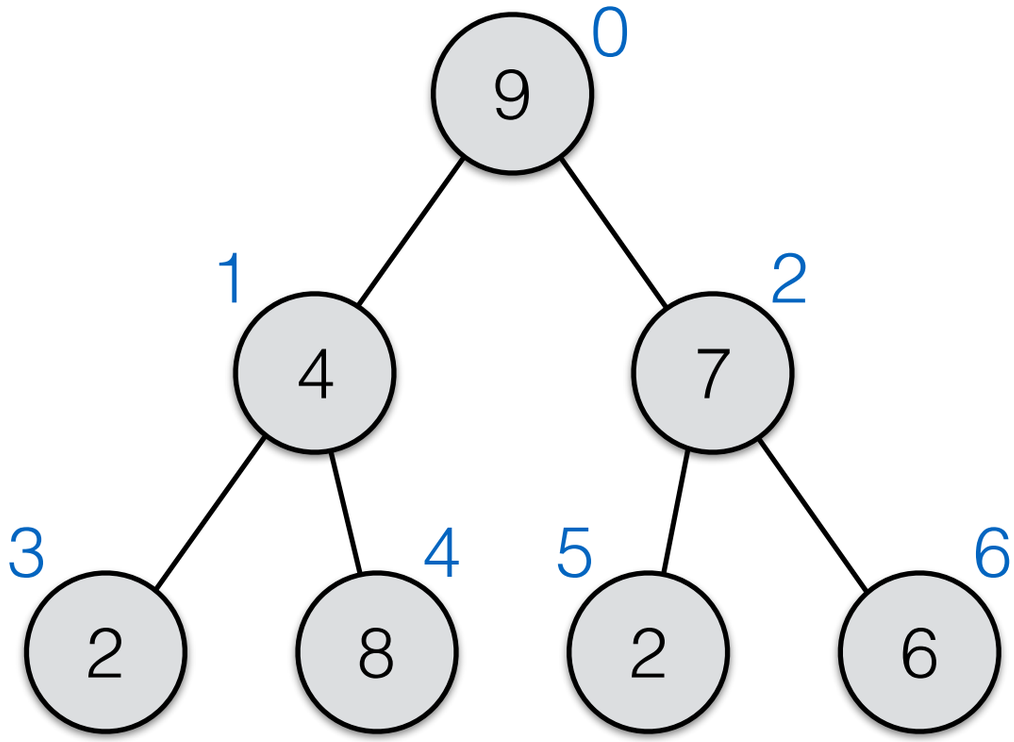
- Naïve approach:
  - `add()` keys one at a time — `percolateUp()`.
  - O(n log n) time
- Faster approach: `heapify()`
  - `percolateDown()`
  - O(n) time

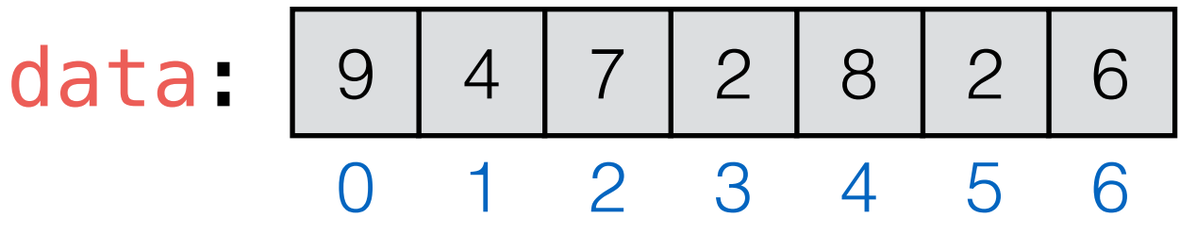
`heapify()`

1. Make a complete tree out of the entries by putting them, in any order, into `data` array.
2. Work backward from the last non-leaf node to the root, in reverse order in the `data` array.

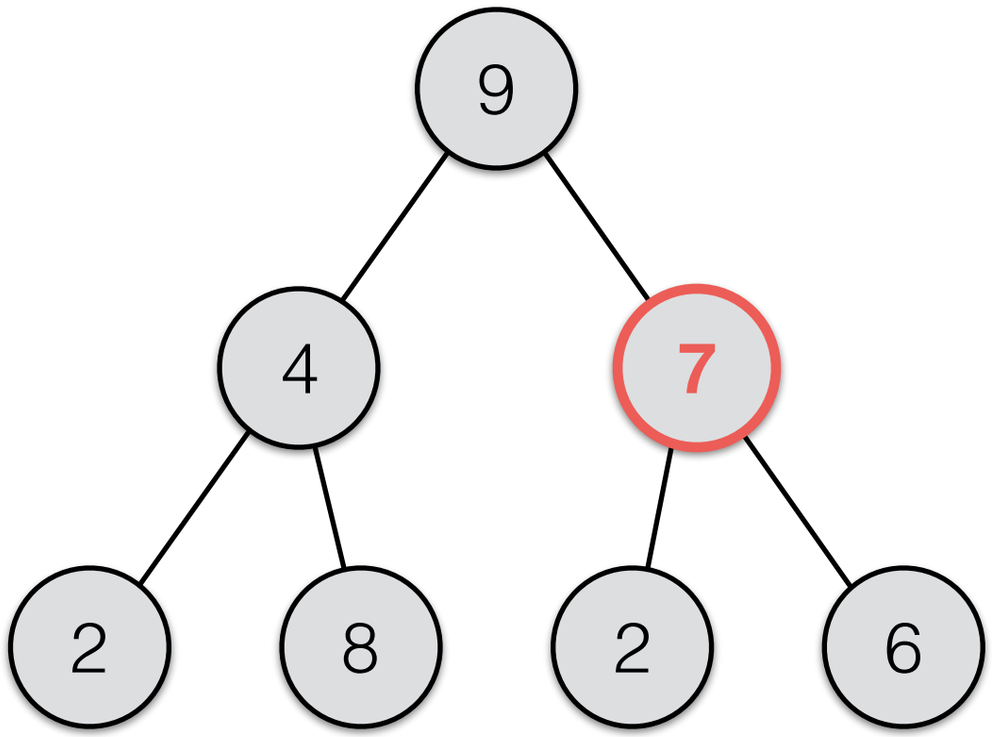
    - When visiting a node, percolate its entry down as in `remove()`.

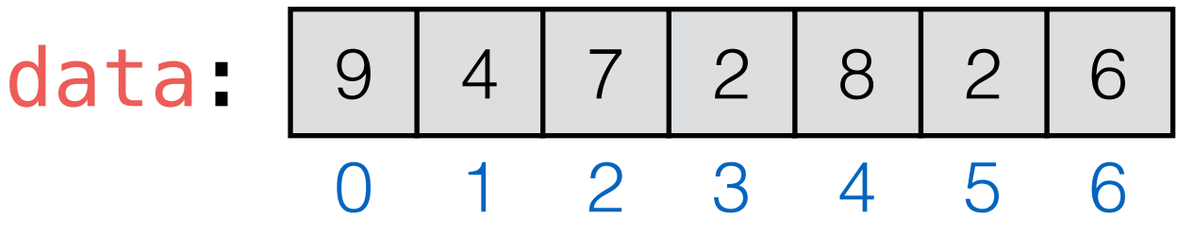
heapify()



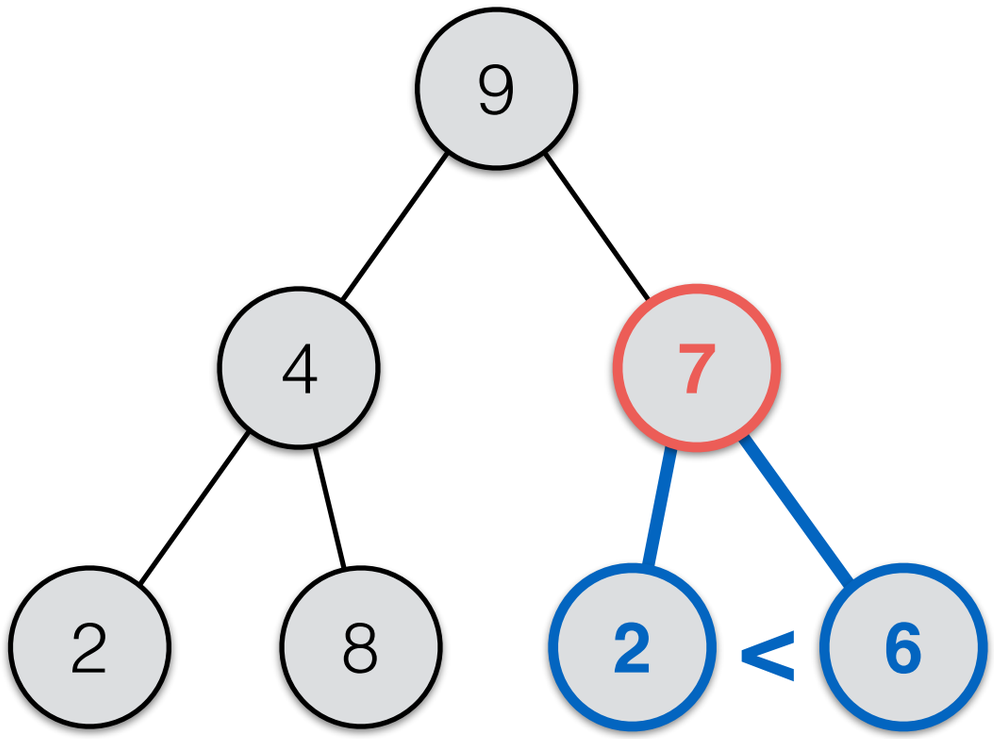


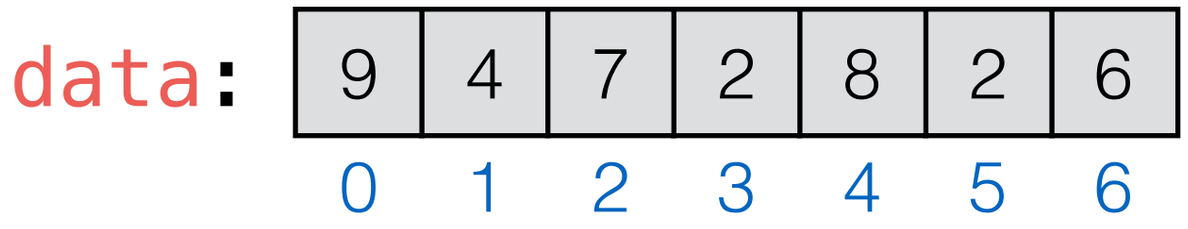
heapify()



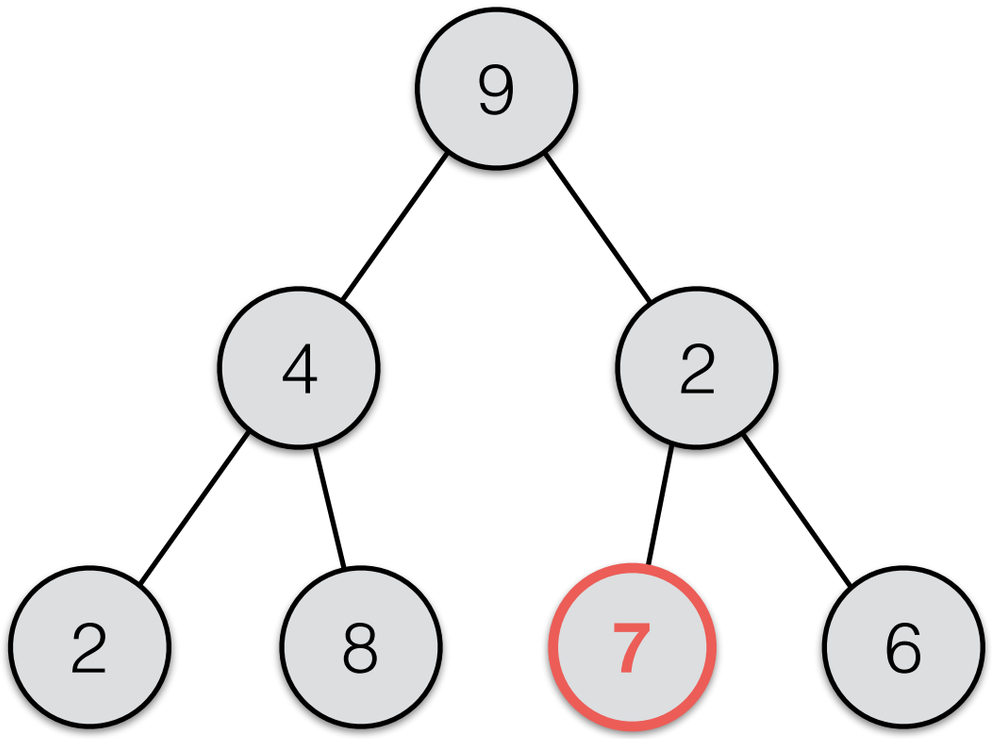


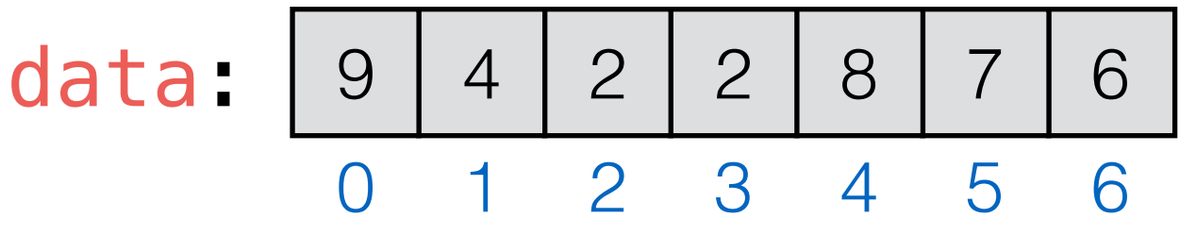
heapify()



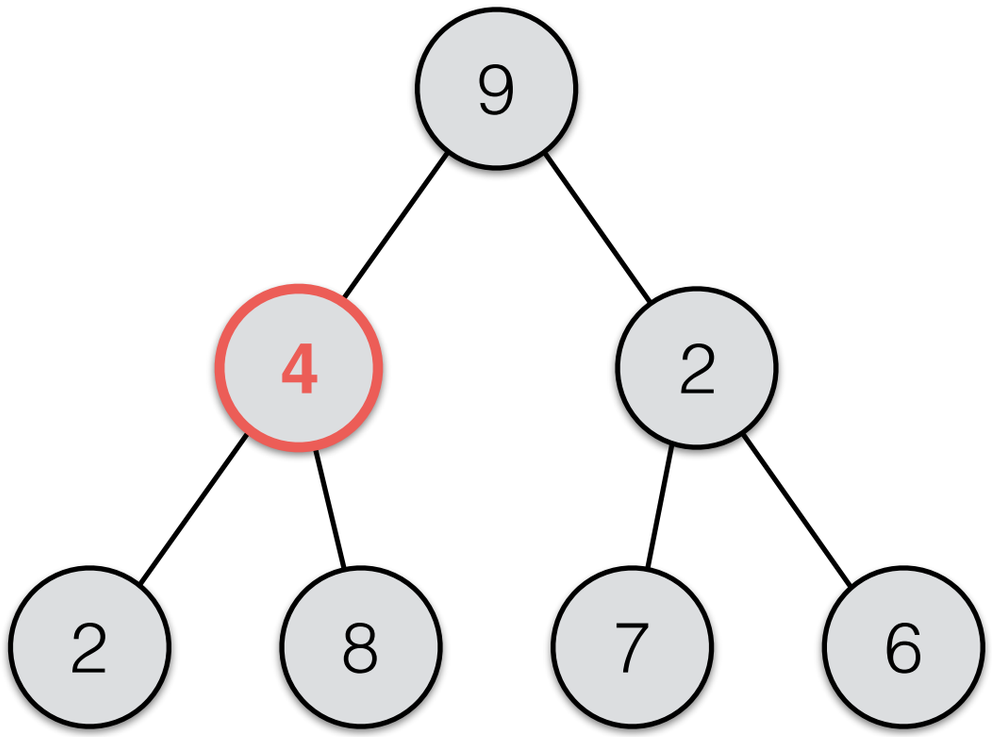


heapify()





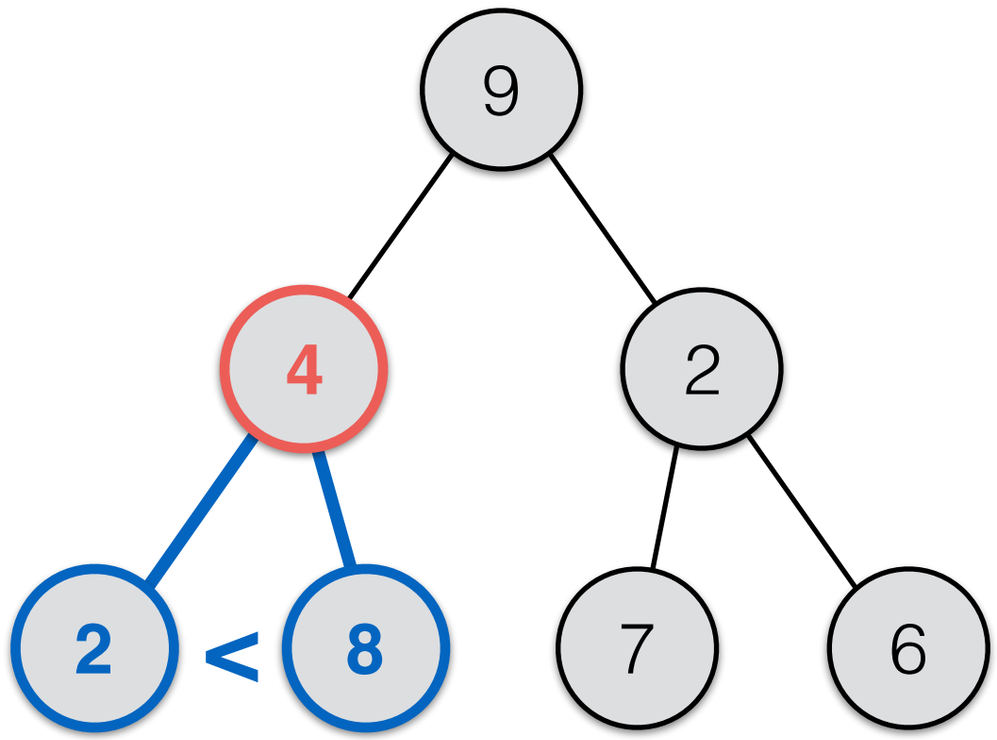
heapify()



data:

9	4	2	2	8	7	6
0	1	2	3	4	5	6

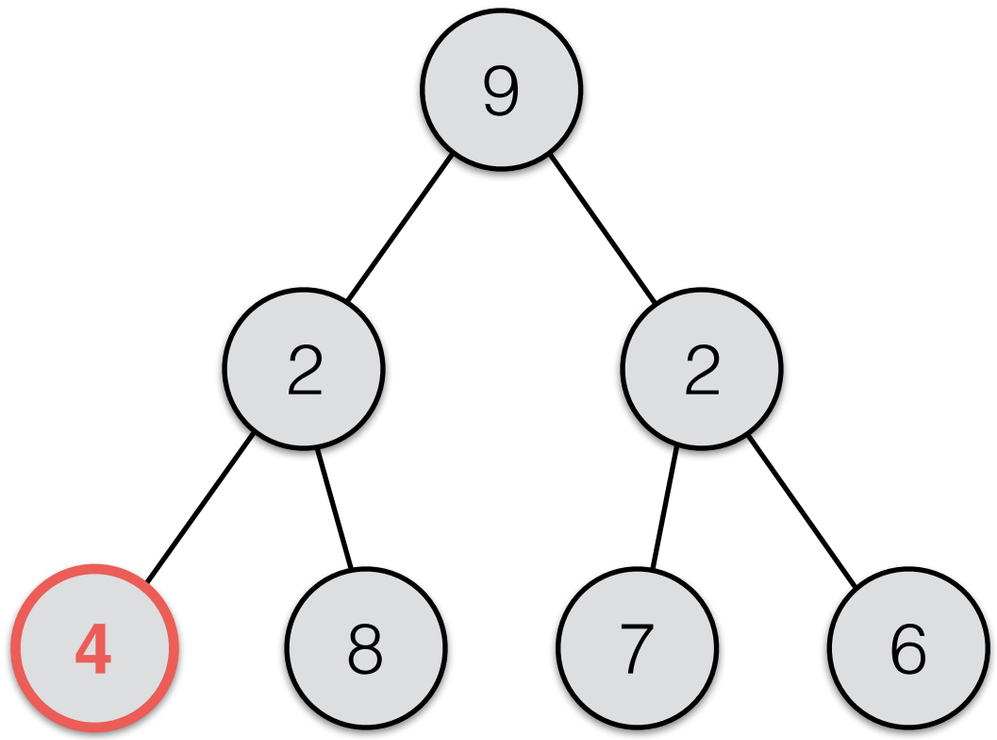
heapify()

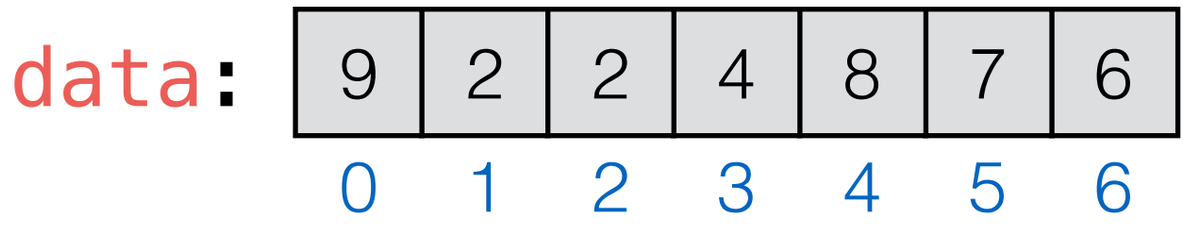


data:

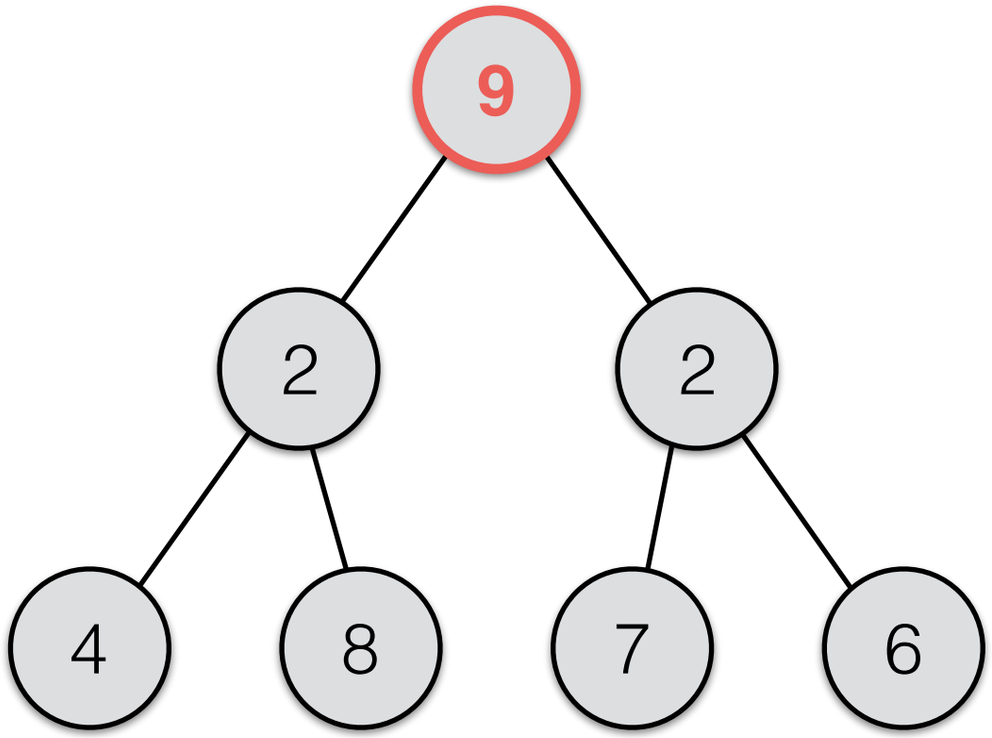
9	4	2	2	8	7	6
0	1	2	3	4	5	6

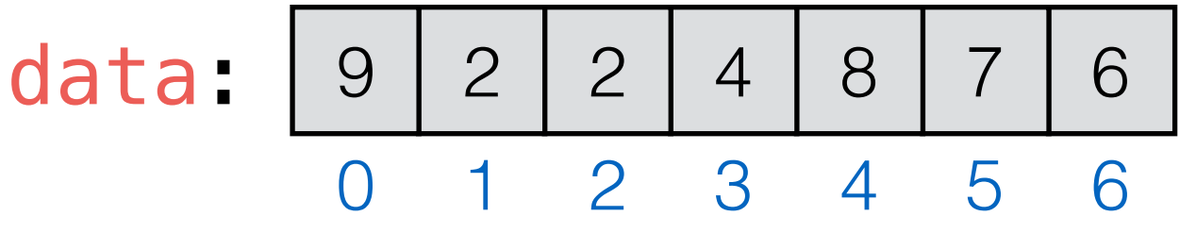
heapify()



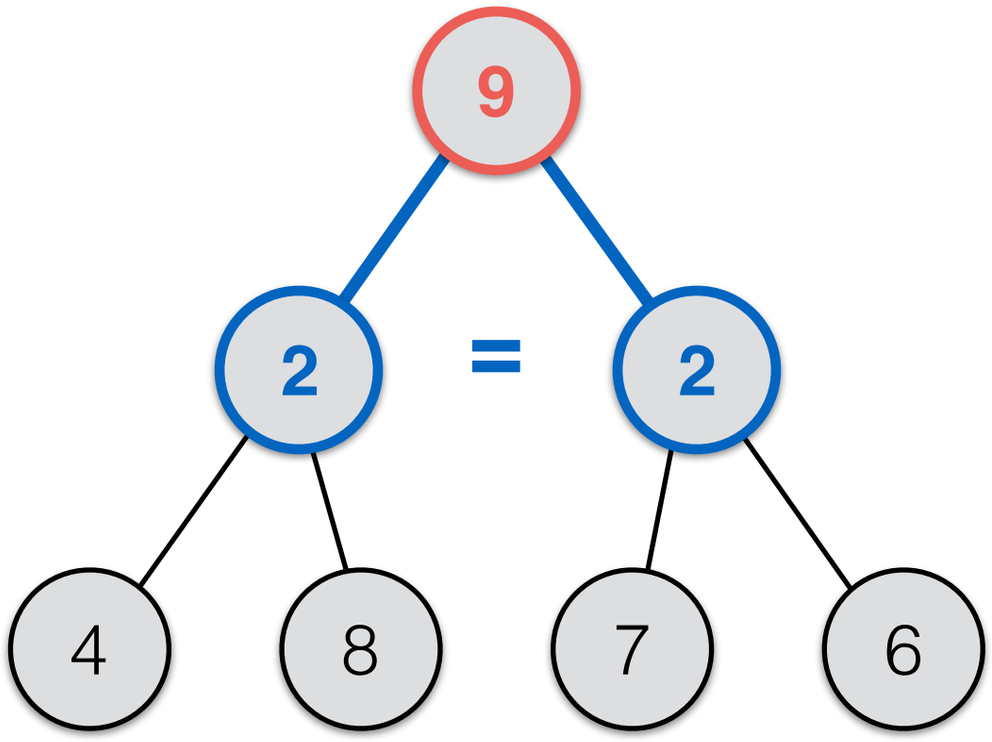


heapify()





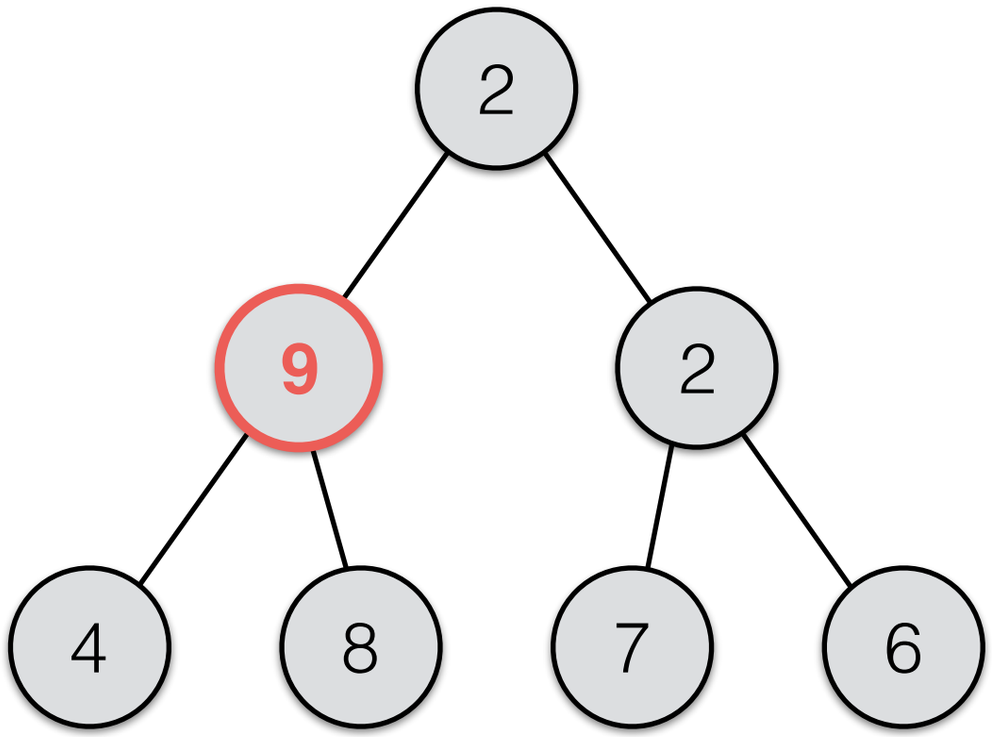
heapify()

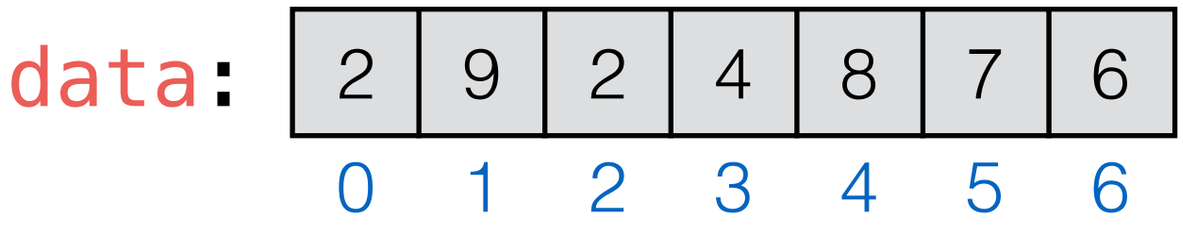


data:

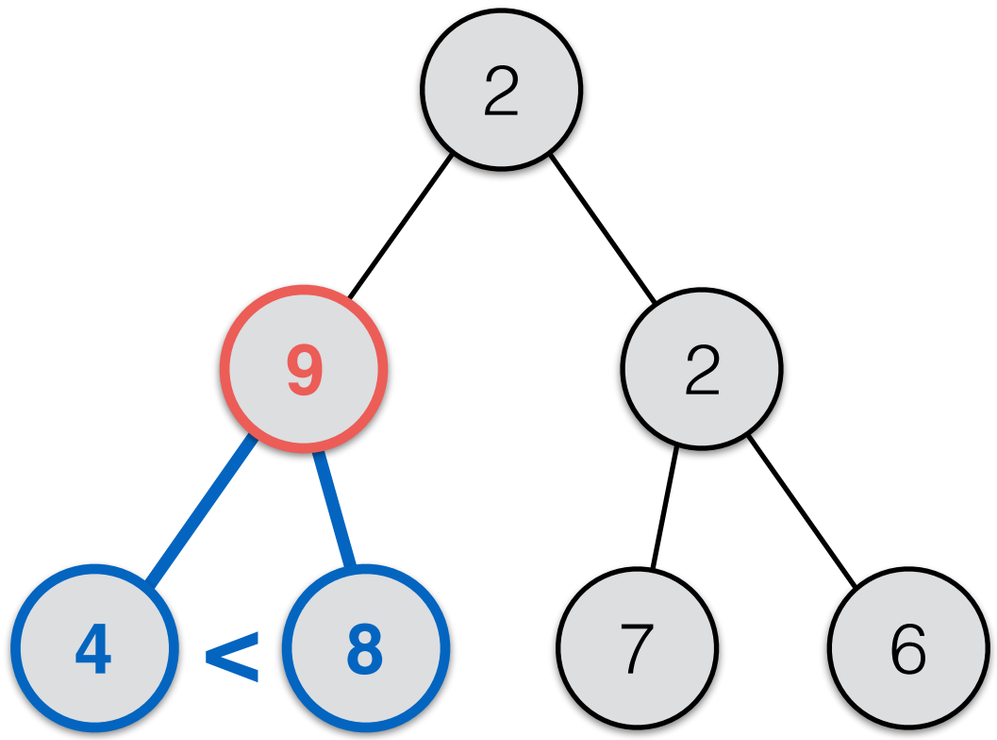
9	2	2	4	8	7	6
0	1	2	3	4	5	6

heapify()





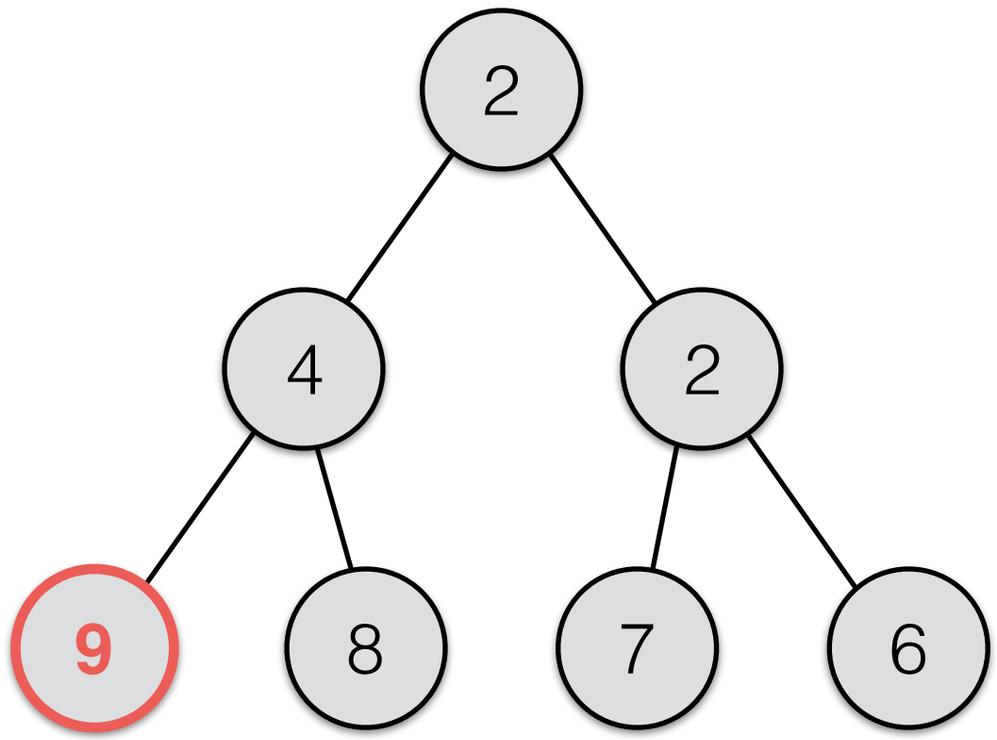
heapify()



data:

2	9	2	4	8	7	6
0	1	2	3	4	5	6

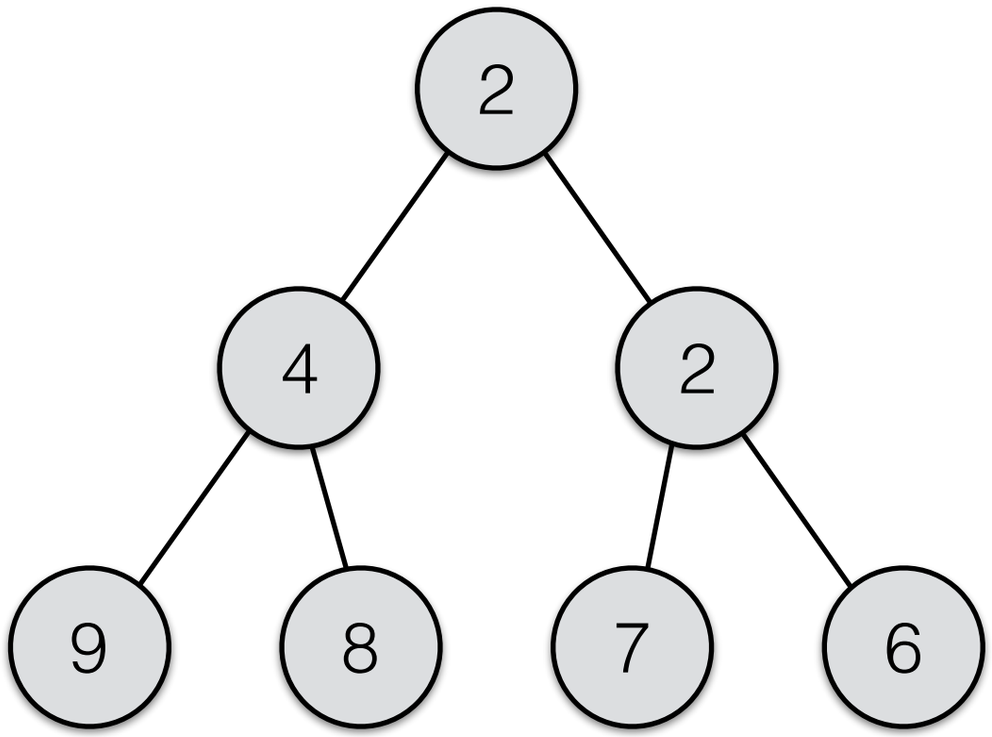
heapify()



data:

2	4	2	9	8	7	6
0	1	2	3	4	5	6

heapify()



data:

2	4	2	9	8	7	6
0	1	2	3	4	5	6

```
heapify(data, current):
   // Loop invariant: for each index i > current,
   // the subtree rooted at data[i] is heap-
   // ordered.
  current =  size/2 – 1;
  while (current >= 0)
      percolateDown(data, current);
      --current;
```