Data Structures

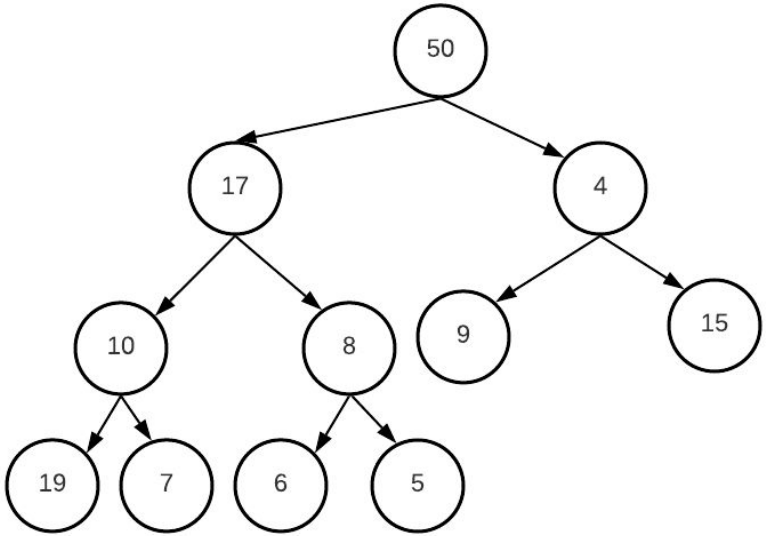
Heap Creation 2

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

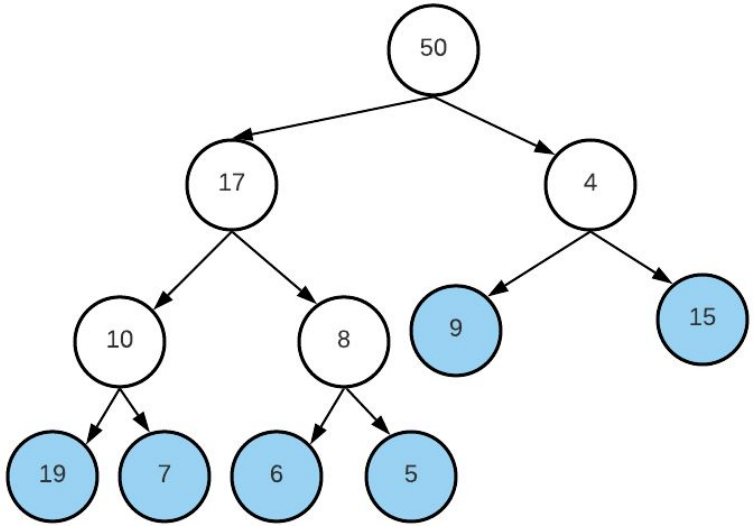
Let’s simulate

- Let's assume we have an array as follows:
  - 50, 17, 4, 10, 8, 9, 15, 19, 7, 6, 5
- We can represent this array as a binary tree
- It's clearly NOT a binary heap!



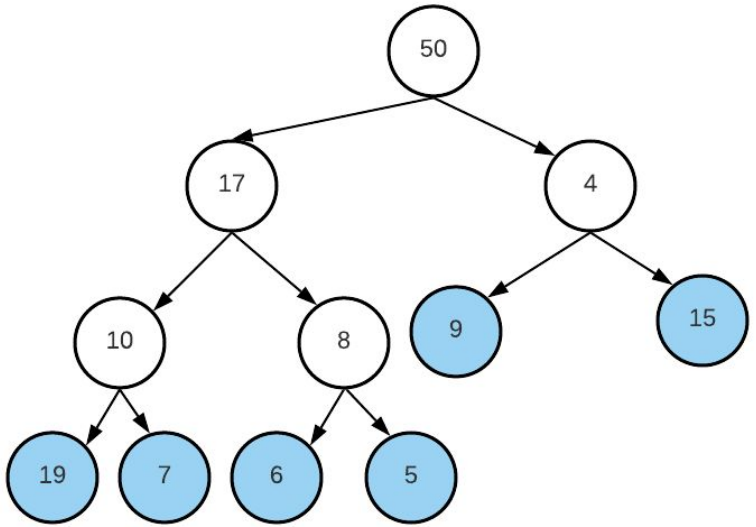
We meet the leaf nodes first

- Our algorithm works backwards
  - **5, 6, 7, 19, 15, 9**, 8, 10, 4, 17, 50
- The first 6 calls of heapify_down(idx) do nothing, as they are leaf nodes
  - Indices: 10, 9, 8, 7, 6, 5
  - Now we have 6 scattered nodes



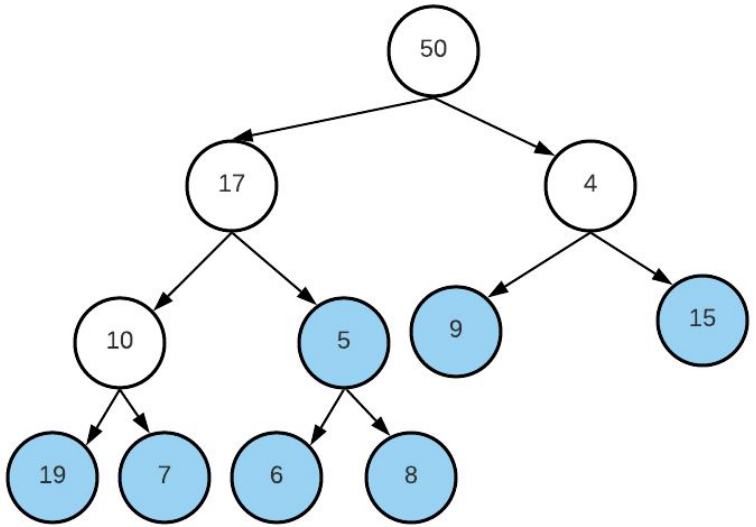
Next: value 8 - index 4

- The next position to heapify down is 4
- 8 is a parent node to both 6 and 5
- Clearly 5 needs to be swapped with 8
- After that, 8 is now a leaf; and we stop



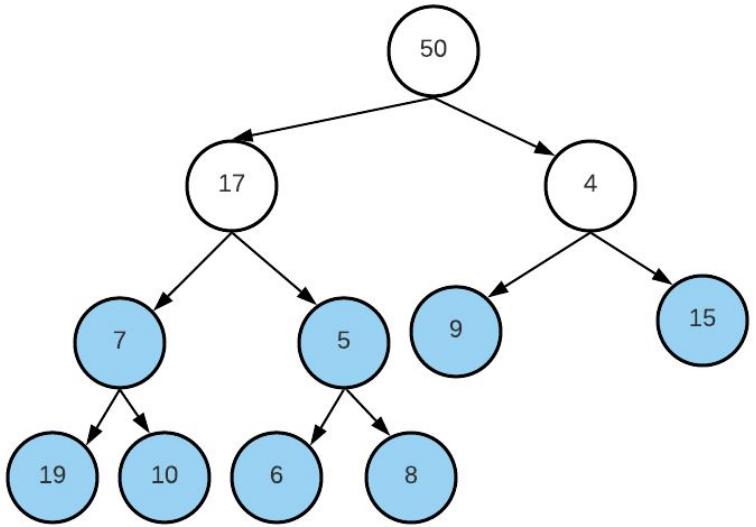
Next: value 10 - index 3

- *Observe: root (5) is a min heap*
- The next position to heapify down is 3
- 10 is a parent to both 19 and 7
- Clearly 10 needs to be swapped with 7
- 10 is now a leaf node; we can stop



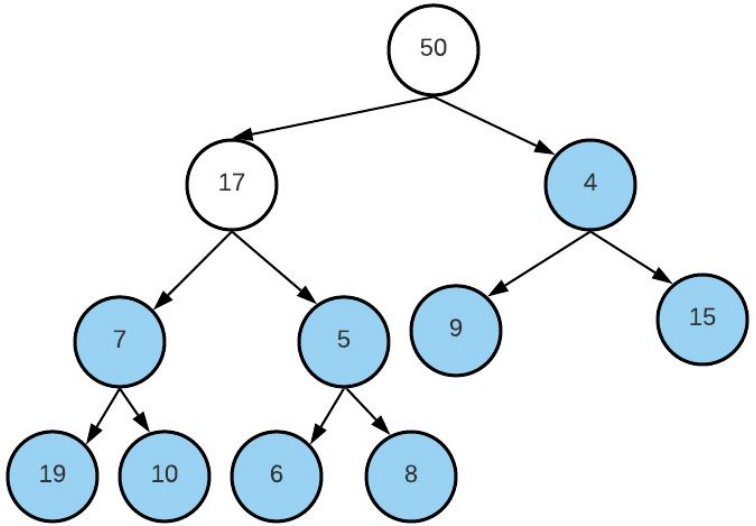
Next: value 4 - index 2

- *Observe: root (7) is a min heap*
- The next position to heapify down is 2
- 4 is a parent to both 9 and 15
- This is already perfect (as a parent!). Stop
  - In a large tree, this will happen a lot.
  - Remember, left and right subtrees are already min-heaps, as we are building from bottom to up



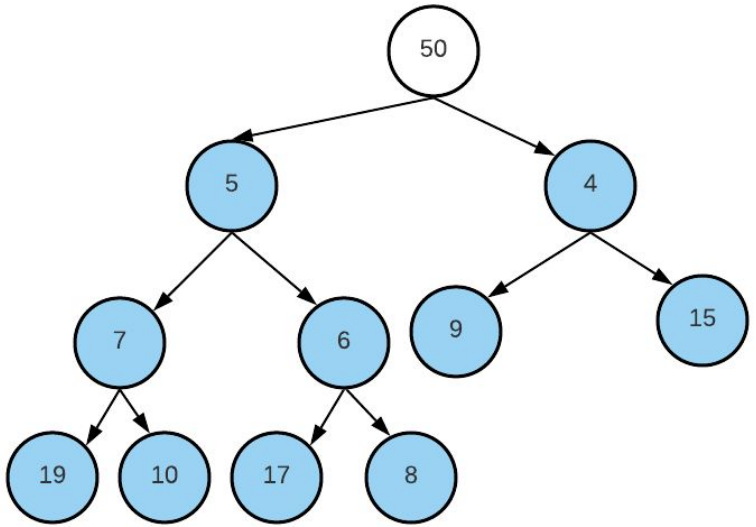
Next: value 17 - index 1

- The next position to heapify down is 1
- 17 is a parent to both 7 and 5
- 17 must swap places with 5
- 17 will now be a parent for both 6 and 8
- Swap 17 with 6



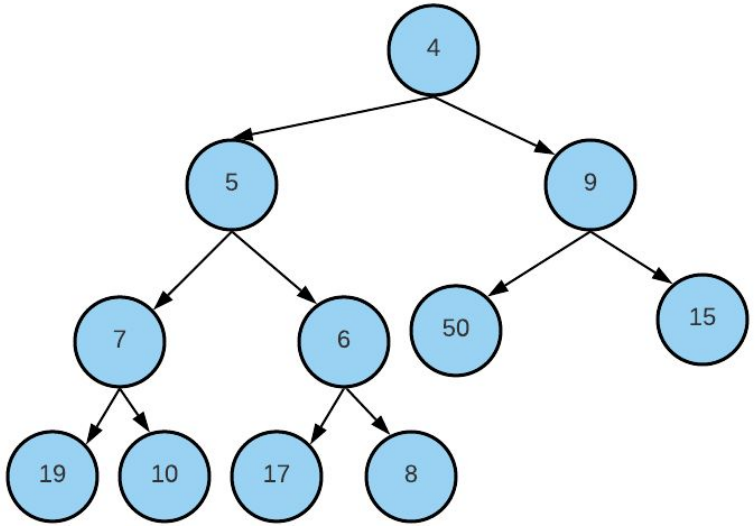
Next: value 50 - index 0

- *Observe: root (5) is a min heap*
- The next position to heapify down is 0
- 50 is a parent to both 5 and 4
  - Swap 50 and 4
- 50 is now a parent to both 9 and 15
  - Swap 50 and 9
- 50 is now a leaf node; we can stop here



Done

- As you can see, we were building bottom-up **separate binary** trees
- As we go up each level, we sometimes create new parents for our nodes
- We fix any new parents, and the child subtrees
- Interestingly: we were creating a tree but using a deletion procedure



Optimization: Skipping the leaves

- We iterate on many leaf nodes.
  - By definition, a leaf node has no children
- To optimize, we can simply process **non-leaf nodes** alone!
- How many non-leaf nodes are in a complete binary tree of n nodes?
- Think for 5 minutes. Just try some trees and guess the formula

Optimization: Skipping the leaves

- How many non-leaf nodes are in a complete binary tree of n nodes?
  - Let's enumerate
  - 16 ⇒ 8, 15 ⇒ 7, 14 ⇒ 7, 13 ⇒ 6, 12 ⇒ 6, 11 ⇒ 5, 10 ⇒ 5
  - Clearly floor (n/2) non-leaf nodes
- We need only process the first n/2 nodes (working backwards)
  - In a 0-based array (as in C++), the position n/2-1 is the first non-leaf node in a complete binary tree

```
def _heapify(self):
    # Iterate from the first NON-leaf node
    for i in range(self.size//2 -1, -1, -1):
        self._heapify_down(i)
```

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."