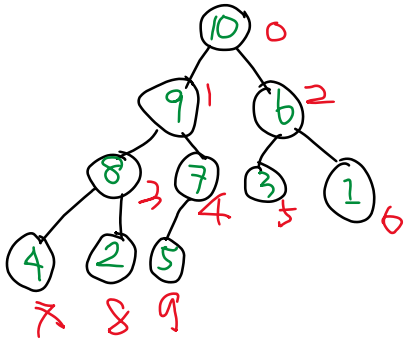
Priority Queue with Binary Heap

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

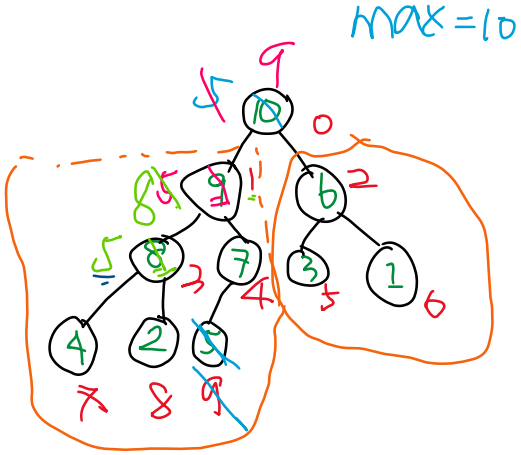


- Some methods
  - **__init__**$(self)$:
    $self.data = [\,]$
  - **max**$(self)$:
    return $self.data[0]$
  - **left**(i): // i is an index
    return $2i + 1$
  - **right**(i):
    return $2i + 2$
  - **parent**(i):
    return $(i - 1)$ // 2

1. True or False: the smallest item in a binary heap is always at the last index. $False$
2. What is the height of a binary heap with n nodes? $\lfloor \lg n \rfloor$
   - The height of a node is defined as the length of the longest path from this node to a leaf below it. For example, the node at index 9 has height 0, the node at index 1 has height 2, and the node at index 0 has height 3. The height of a tree is the height of its root.
3. How many leaves does a binary heap with n nodes have? $\lceil n/2 \rceil$
   - A leaf is a node who has no children.

- To pop_max from a binary heap, follow the pseudo-code below:

pop_max()
1 $max = self.data[0]$
2 **swap** $self.data[0]$ and $self.data[-1]$
3 $del\ self.data[-1]$
4 **max − heapify** (0)
5 **return** max



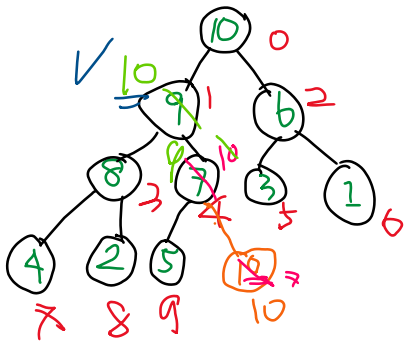
Here, $\textbf{max} - \textbf{heapify}\ (i)$ is a method that makes the sub-tree rooted at index i a heap again. This method assumes that item at index i is the only item that violates the max-heap property, and it needs to be "floated downward".

$\textbf{max} - \textbf{heapify}\ (i)$
1 $l = \textbf{left}\ (i)$
2 $r = \textbf{right}\ (i)$
3 **if** $l < len(self.data)$ and $self.data[l] > self.data[i]$
4 $\quad largest = l$
5 **else** $largest = i$
6 **if** $r < len(self.data)$ and $self.data[r] > self.data[largest]$
7 $\quad largest = r$
8 **if** $largest \neq i$
9 $\quad$ **swap** $self.data[i]$ and $self.data[largest]$
10 $\quad \textbf{max} - \textbf{heapify}\ (largest)$

It is easy to see that the running time of $\textbf{max} - \textbf{heapify}$ on a node at height h is $O(h)$. Since node i can be floated down from the root to a leaf, $\textbf{max} - \textbf{heapify}$ runs in $O(\lg n)$ time. This implies that pop_max also has time complexity $O(\lg n)$.

- To $add(item)$ to a binary heap, follow the pseudo-code below:

$\textbf{add}\ (item)$
1 $self.data.append(item)$
2 $\textbf{increase}(len(self.data) - 1, item)$



Here, $\textbf{increase}\ (i, item)$ is a method that increase the value of $self.data[i]$ to $item$. It won't do anything if $item$ is smaller than $self.data[i]$. When the value of $self.data[i]$ is increased, we might need to float this item upward.

$\textbf{increase}\ (i, item)$
1 **assert** $item \geq self.data[i]$
2 $self.data[i] = item$
3 **while** $i > 0$ and $self.data[\textbf{parent}(i)] < self.data[i]$
4 $\quad$ **swap** $self.data[i]$ and $self.data[\textbf{parent}(i)]$
5 $\quad i = \textbf{parent}(i)$

Similar to $\textbf{max} - \textbf{heapify}$, it is easy to see that the time complexity of **increase** is also bounded by the height of the binary heap, thus its time complexity is also $O(\lg n)$. This implies that $add(item)$ also has time complexity $O(\lg n)$.

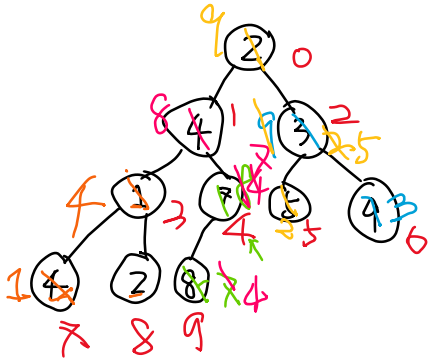
An alternative construction method

- Under many circumstances, we allow users to create a binary heap with a list of items. It will be really inefficient if we only allow users to build an empty binary heap, then add all items in the list to the binary heap one by

one. If there are n items in the list, adding all n items to an empty binary heap takes $O(n \lg n)$ time. Is possible to build heap faster? Look at the following pseudo-code.

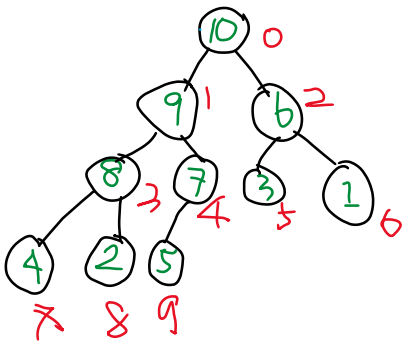
```
__init__(self, data = []):
1  self.data = data
2  if len(data) > 0:
3          for i in range(len(self.data) // 2 − 1, −1, −1) :
4                  max − heapify(i)
```

- A complete binary tree with n nodes has $\lceil n/2 \rceil$ leaves. Each leaf itself is a small heap.
- In line 3, i starts with the last non-leaf index, then goes down. When we look at the subtree that is rooted at $self.data[i]$, this item itself is the only one that might violate the max-heap property in the subtree. Thus, we can call max-heapify here. And after calling max-heapify, the subtree rooted at $self.data[i]$ will be a heap.
- We repeat this procedure to the root of the tree.



4. What is the time complexity of this construction method if we need to go into the for loop in line 3? Assume that there are n items in the binary heap.
   - Since there are $\left\lfloor \frac{n}{2} \right\rfloor$ calls of $\mathbf{max - heapify}$, so it looks like that this method has time complexity $O(n \lg n)$, but this upper bound is not tight.
   - As we mentioned earlier, $\mathbf{max - heapify}$ on a node with height h has running time $O(h)$. Thus, the total running time of the for loop is upper bounded by the total heights of all nodes in the binary heap.
   - To calculate the total heights, we patch this binary heap with n node to a full tree. Let k be the number of patched nodes, and $k < n$. (Why?

)
   - After patching, the total heights of all nodes can only go up.
   - There are $n + k$ nodes in the full binary tree now. Let $N = n + k + 1$, then there are $\frac{N}{2}$ nodes with height 0, $\frac{N}{2^2}$ nodes with height 1, $\frac{N}{2^3}$ nodes with height 2, $\frac{N}{2^4}$ nodes with height 3 …



   - Total height $= \frac{N}{2} \times 0 + \frac{N}{4} \times 1 + \frac{N}{8} \times 2 + \frac{N}{16} \times 3 + \cdots$
     $= N \times \left(\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + \cdots \right)$

- $\frac{1}{2^2}+\frac{2}{2^3}+\frac{3}{2^4}+\frac{4}{2^5}+\cdots$ $= \frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}+\frac{1}{2^5}+\cdots$
  $+ \quad \frac{1}{2^3}+\frac{1}{2^4}+\frac{1}{2^5}+\cdots$
  $+ \quad \frac{1}{2^4}+\frac{1}{2^5}+\cdots$
  $+ \quad \frac{1}{2^5}+\cdots$
  $+\cdots$
  $= \quad =$

- This shows that the total heights of nodes in the patched tree is N, which means that the total height in the binary heap is at most $N = n + k + 1 = O(n)$. Thus, if we go into the for loop, the construction method has time complexity $O(n)$.