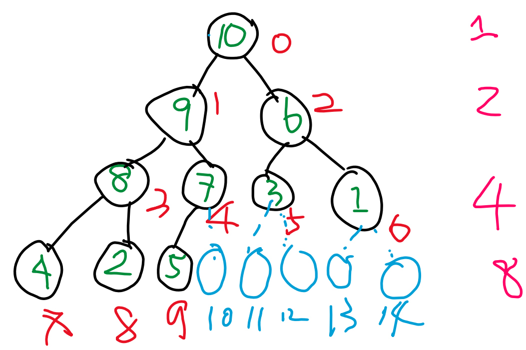
An alternative construction method for binary heap

```
    __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)
```

1. 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 in the last class, $\mathbf{max-heapify}$ 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? We can patch at most $\frac{N}{2} - 1$ nodes to the heap, but there are at least $\frac{N}{2}$ nodes in the binary heap before patching)
    - 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

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

$N=16$

$2^0 + 2^1 + 2^2 + 2^3 \cdots, 2^m$

$= 2^{m+1} - 1$

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

$$\begin{aligned} = & \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots \\ + & \quad\;\;\; \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots \\ + & \quad\quad\quad\;\;\; \frac{1}{2^4} + \frac{1}{2^5} + \cdots \\ + & \quad\quad\quad\quad\quad\;\;\; \frac{1}{2^5} + \cdots \\ + & \cdots \\ = & \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1 \end{aligned}$$

$\dfrac{\frac{1}{2^2}}{1 - \frac{1}{2}} = \frac{1}{2}$

- This shows that the total heights of nodes in the patched tree is at most 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 $O(n)$.

Heapsort

- Given an array (or ArrayList) with n elements, we can build a heap in $O(n)$ time and we can pop_max in $O(\lg n)$ time, it is straight forward to come up with the following sorting algorithm:

Heapsort $(A[0 \dots n-1])$: # A is an array or ArrayList with indices 0 to $n-1$

$bh = BinaryHeap(A)$ # $O(n)$
for $i = 1$ **to** n: # n iterations
    $item = bh.pop\_max()$ # $O(\lg n)$
    $A[n-i] = item$

$O(n + n\lg n) = \Theta(n \lg n)$

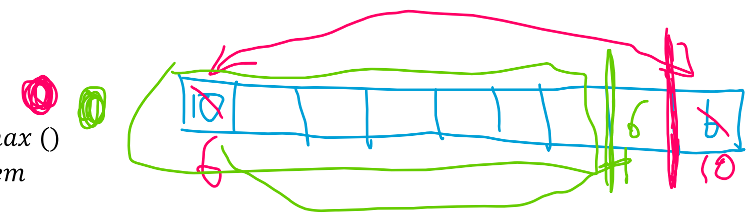
2. What is wrong in the following python implementation of $Heapsort$? How to fix it?

In the class $BinaryHeap$, we have the following construction method:

__init__$(self, data = [\,])$:
$self.data = data$
if $len(data) > 0$:
    for i **in range**$(len(self.data) // 2 - 1, -1, -1)$:
        $BinaryHeap.max-heapify(i)$

Outside of the class $BinaryHeap$, we implement heapsort as follows:

Heapsort $(A : [\,])$:
$bh = BinaryHeap(A)$
$n = len(A)$
for i **in range** (n):
    $item = bh.pop\_max()$
    $A[n-1-i] = item$



- In the construction method, we only let $self.data$ point to the list $data$. Thus, in $Heapsort$, $bh.data$ and A are the same list; whenever we pop_max, the length of list A is also decrease by 1.

- One way to fix this problem, is to hardcopy $data$ into $self.data$:

__init__$(self, data = [\,])$:
$self.data = [\,]$
for i **in** $data$:
    $self.data.append(i)$
if $len(data) > 0$:
for i **in range**$(len(self.data) // 2 - 1, -1, -1)$:
    $BinaryHeap.max-heapify(i)$

In this implementation, we need to use $\Theta(n)$ extra space to create a copy of $data$.

- Another simple way to fix this is to create a new list, whenever we pop_max, we add the popped-out item to the new list; this design also requires $\Theta(n)$ extra space.

- Under most of the circumstances, a $\Theta(n)$ extra space isn't a big problem. We still want to know how to $heapsort$ without using this much extra space. We need to augment the design of our $BinaryHeap$ class a little bit without affecting the time complexity of each method.

Our Final Design of a Binary Heap

- We keep an extra attribute $self.heapsize$ that represent the current number of items in the binary heap. Only items in slice $self.data[0 : self.heapsize]$ are considered as items in the binary heap.

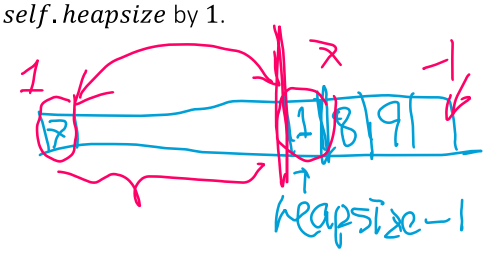
```
__init__(self, data = []):
self.data = data
self.heapsize = len(data)
if len(data) > 0:
        for i in range(len(self.data) // 2 − 1, −1, −1) :
                max − heapify(i)
```

- In **max − heapify**, when we compare in index of a left child or a right child with the size of the binary heap, use $self.heapsize$ instead of $len(self.data)$ since they can be different now.

```
max − heapify (i)
l = left (i)
r = right (i)
if l < self.heapsize and self.data[l] > self.data[i]
        largest = l
else   largest = i
if r < self.heapsize and self.data[r] > self.data[largest]
        largest = r
if largest ≠ i
        swap self.data[i] and self.data[largest]
        max − heapify (largest)
```

- In **pop_max**(), instead of deleting the maximum item, simply decrease $self.heapsize$ by 1.

```
pop_max()
max = self.data[0]
swap self.data[0] and self.data[self.heapsize − 1]
self.heapsize −= 1
max − heapify (0)
return max
```



- In **add(item)**, now we need to consider two cases: whether there are “empty” spots in the $self.data$.

```
add (item)
if self.heapsize < len(self.data):
        self.data[self.heapsize] = item
else:
        self.data.append(item)
self.heapsize += 1
increase(self.heapsize − 1, item)
```

- Now, let’s look at the pseudo-code of $Heapsort$ again, you will see it is an onsite sorting algorithm.