Heaps

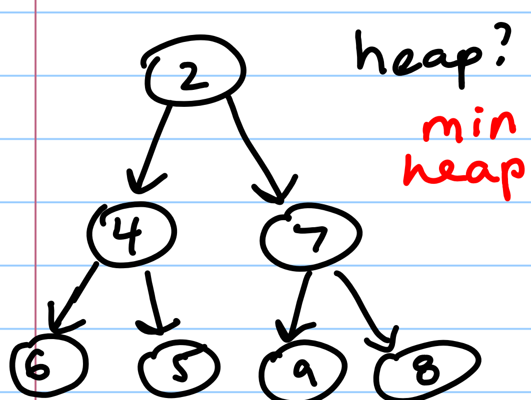
Note Title 11/1/2007

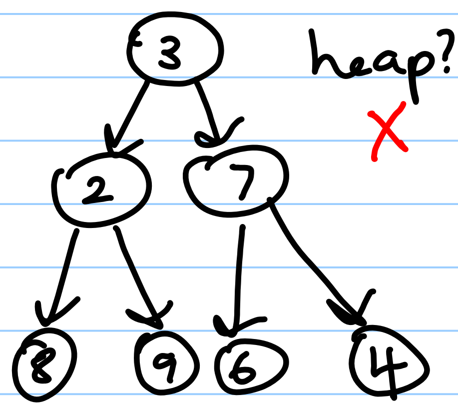
A heap is a completely filled binary tree (ok, maybe not full leaves). leaves filled left to right.
and either one of

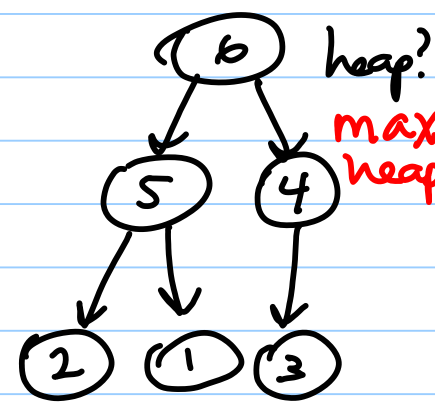
- Max-Heap
  For all nodes, value of parent $\geq$ value of self

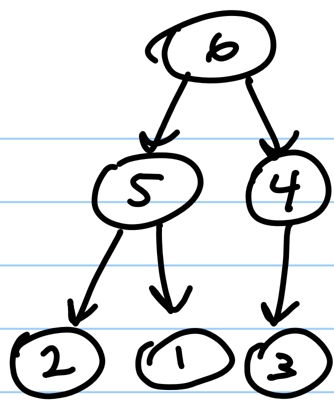
OR

- Min Heap
  For all nodes, value of parent $\leq$ value of self









How long to find 1 in the heap?

worst-case search: n

worst-case height: $\log n$

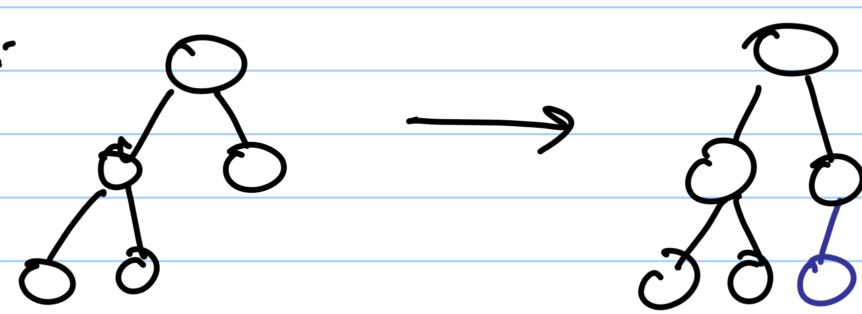
What is easy to find in a max-heap?

max is $O(1)$

Insert into a heap

- due to the structure of a heap, you always know the shape a heap will have after it goes from n to $n+1$ elements.

example:



Insert into a heap

- so, to insert a node into a heap,

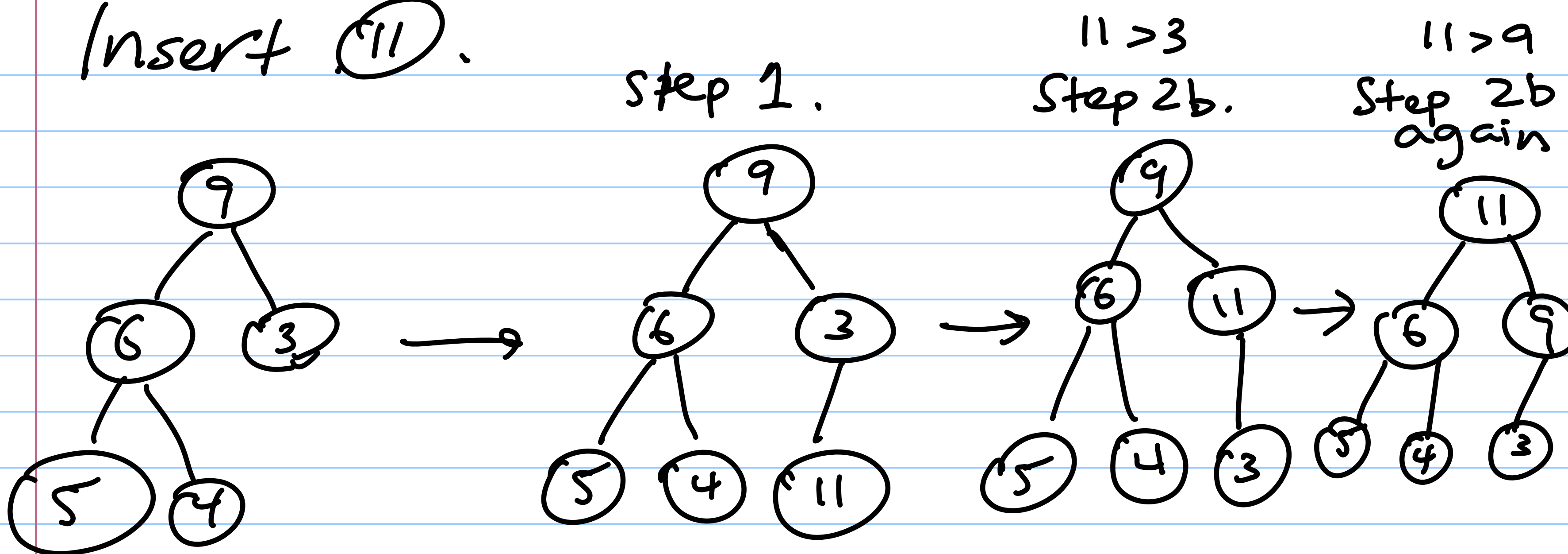
1. add new node to leftmost open position in leaves

2. adjust as needed to ensure heap property is met.

   a. compare new node to parent.
   b. swap if new node bigger.
   c. Repeat until swap not needed or new node is root.

example:

Insert 11.



Step 2 called "bubbling up" the new node.

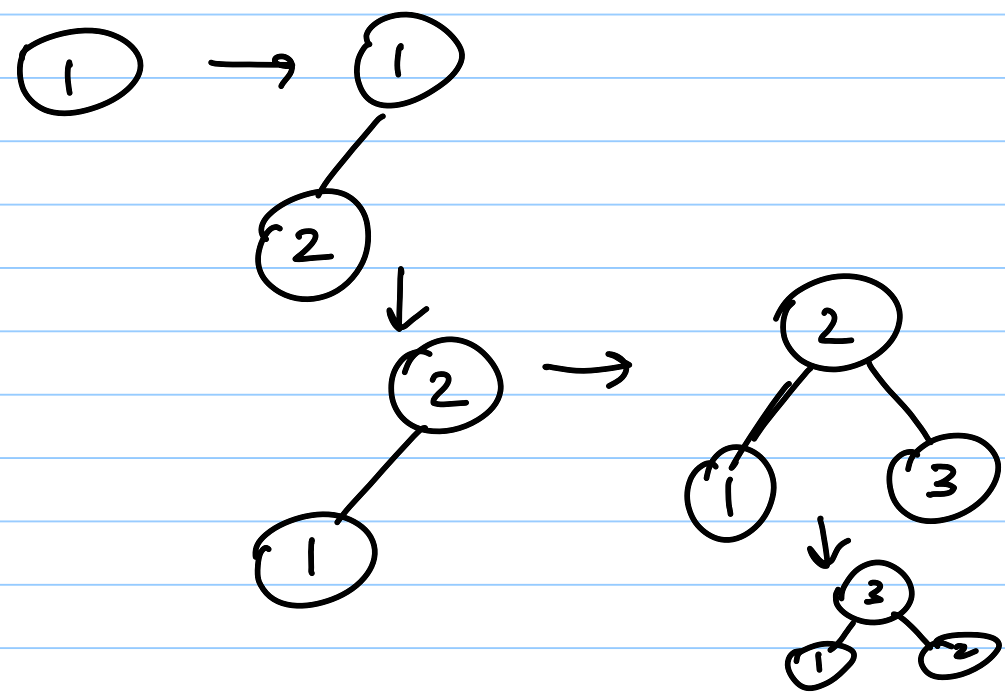
Insert runtime:

worst case → bubble new node up all the way to root

$$\underbrace{O(\log n)}_{\text{length of path from new node to root}}$$

Create a heap from an array of elements (Method I):

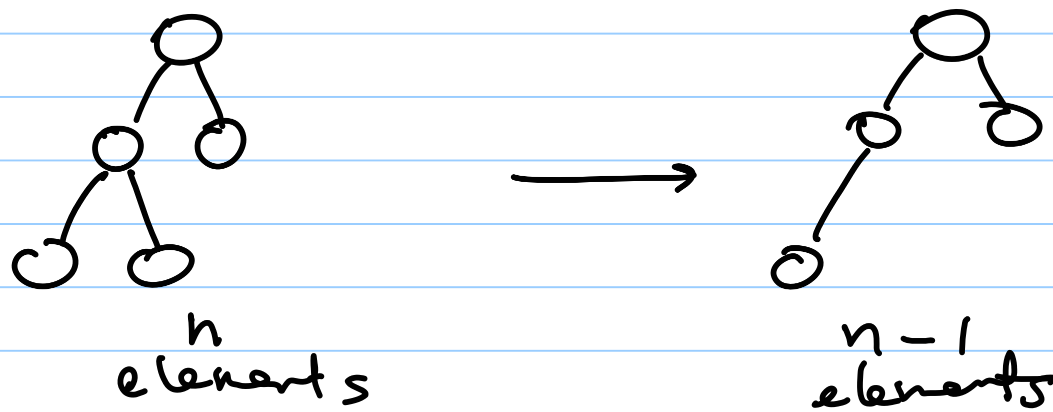
- one way is to insert n elements into the heap one after the other



n calls to insert.
upper bound $O(n \log n)$

Delete Root.

We know the shape of a heap with $n-1$ elements.



Delete Root

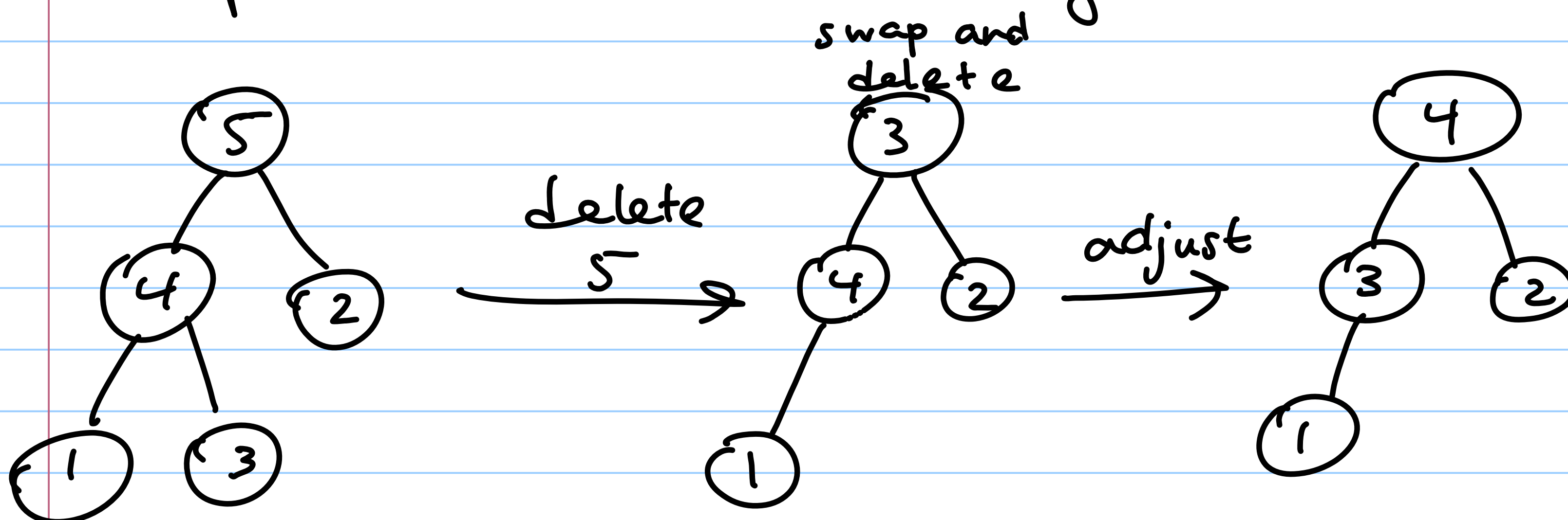
1. Swap root with rightmost leaf
2. Delete old root
3. Adjust to maintain heap property.

   a. Let n = new root, i = ∅
   b. Let largest = max(n, n.left, n.right);

   c.
```
If (largest > n) {
    swap (max, n)   // n now on level i+1
    i++;
}
```

   d. Repeat until no more swaps or n is a leaf

Step 3 called "Bubbling Down"



Step 3b is two comparisons.
worst case - bubble down all the way from root to leaf.
max path length $O(\log n)$

so runtime is $O(\log n)$

Another name for "Bubble Down" is Max-HEAPIFY.

1. Assume children of root are both heaps
2. Bubble down root if needed so whole data structure maintains heap property.

HeapSort (Array A[1...n]) {

(1) Turn A into a max-heap.

(2) Extract the Max using Delete Root.

(3) Repeat step 2 until only one element left in heap.
That element is min.

}.

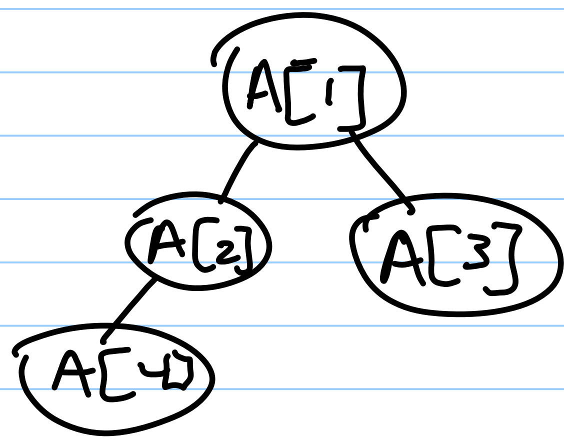
steps 2 and 3 called "harvesting the heap"

Runtime of Heapsort.

Step I: Build the heap.
We know upper bound is $n \log n$.

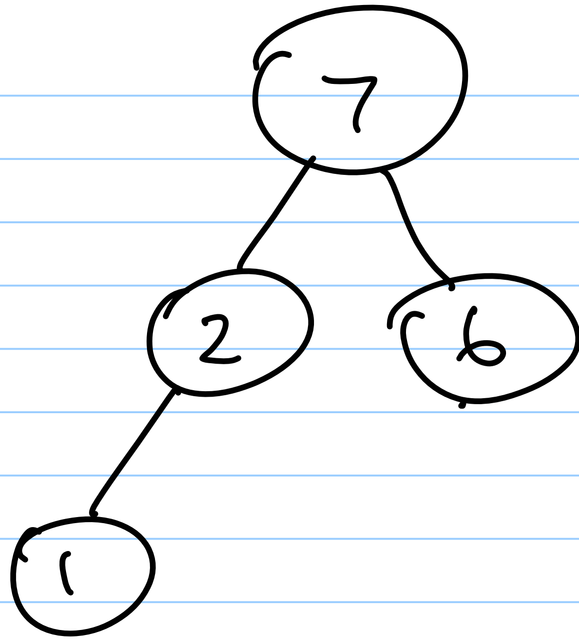
Is there a way to build in in place?

Treat array as a heap:



A[1]	A[2]	A[3]	A[4]

Storing a tree in an array....



1	2	3	4	5	6	7
7	2	6	1			

empty

Index of left child: $2 \cdot$ index of parent

index of right child: $(2 \cdot$ index of parent$) + 1$

(one-based indexing)

BUILD-MAX-HEAP {

1. Treat array like a candidate heap.

2. All leaves are heaps, so start at level before leaves.
   Go from left to right,
   calling MAX-HEAPIFY on each node.

3. Repeat on all higher levels until you finish calling MAX-HEAPIFY on root.

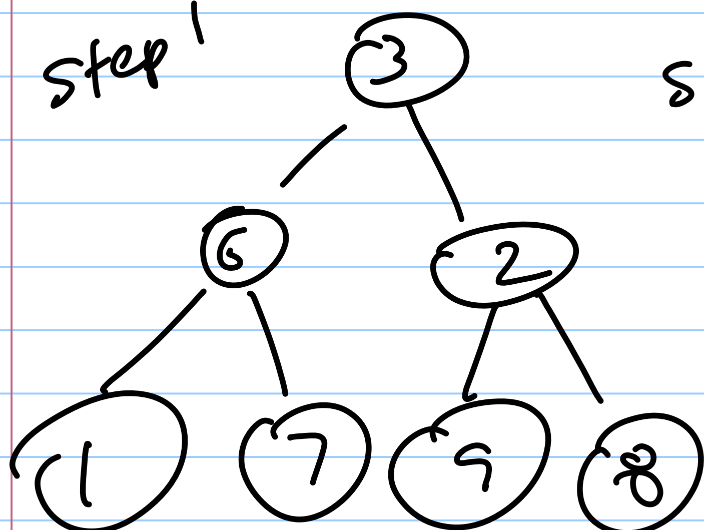
}

So you call MAX-HEAPIFY around $n/2$ times.

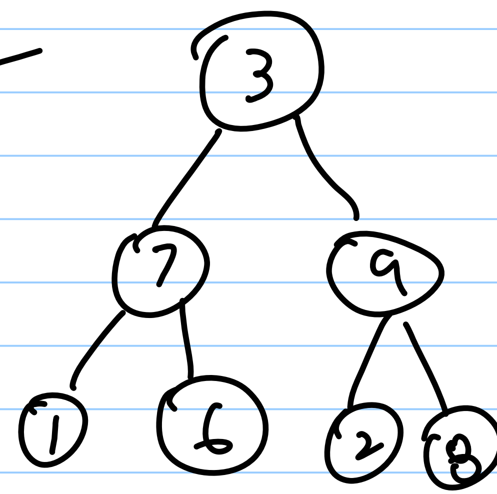
Example of Build-Max-Heap:

3	6	2	1	7	9	8

step 1

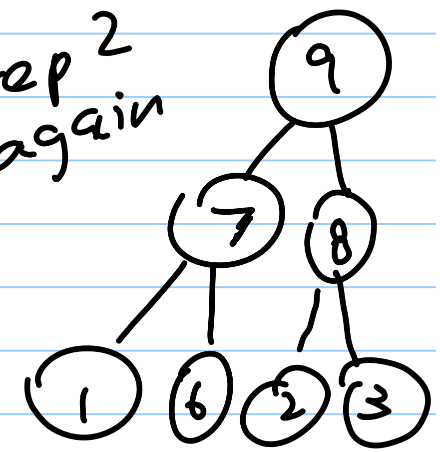


step 2



bubble down 6 and 2

step 2 again



bubbledown 3

Steps 2 and 3: Call DeleteRoot repeatedly
(harvest heap)

Is there a way to harvest heap in place?
Yes.

Let last = n
Put max in A[last].
Treat heap as A[1...(last-1)]
last--;
Repeat.

Runtime of Heapsort?

1. Build heap in place.
2. Harvest heap in place.

Worst case of Step 1: $\Theta(n \log n)$.

But note, when at low levels,

performance of MAX-HEAPIFY
is not $O(\log n)$,
because height of subtree rooted
at lower level nodes $< \log n$.

Runtime of BUILD-MAX-HEAP

is

$$\sum_{h=1}^{\log_2 n} \begin{pmatrix} \text{\# nodes} \\ \text{with height } h \end{pmatrix} O(h).$$

$$= \sum_{h=1}^{\log_2 n} \begin{pmatrix} \text{\# nodes} \\ \text{with height } h \end{pmatrix} (ah)$$

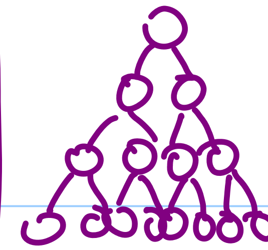
$$= \sum_{h=0}^{\log_2 n} \begin{pmatrix} \text{\# nodes} \\ \text{with height } h \end{pmatrix} (ah)$$

$$= a \sum_{h=0}^{\log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil (ah)$$

(ignore ceiling)

$$= \frac{1}{2} an \sum_{h=0}^{\log_2 n} h \left(\frac{1}{2}\right)^h$$

$$\leq \frac{1}{2} an \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1}{2} an \left[\frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \right]$$



$n=15$

$h=0$

nodes $=8$

$h=1$

nodes $=4$

$h=2$

nodes $=2$

runtime $\leqslant an$

runtime $\in O(n)$ wow!

---

derivation of $\sum_{k=0}^{\infty} k(x)^k = \frac{x}{(1-x)^2}, \quad x<1$

$$\sum_{k=0}^{\infty} (x)^k = \frac{1}{1-x}, \quad x-1$$

Take derivative of both sides

$$\sum_{k=0}^{\infty} kx^{k-1} = (-1)\left(\frac{1}{(1-x)^2}\right)(-1)$$

multiply both sides by x

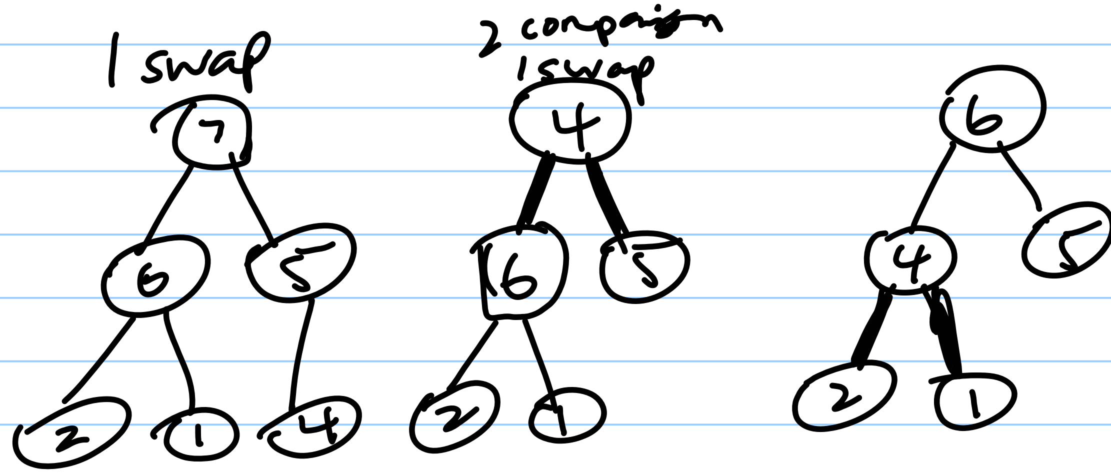
$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}.$$

Runtime of Heapsort?

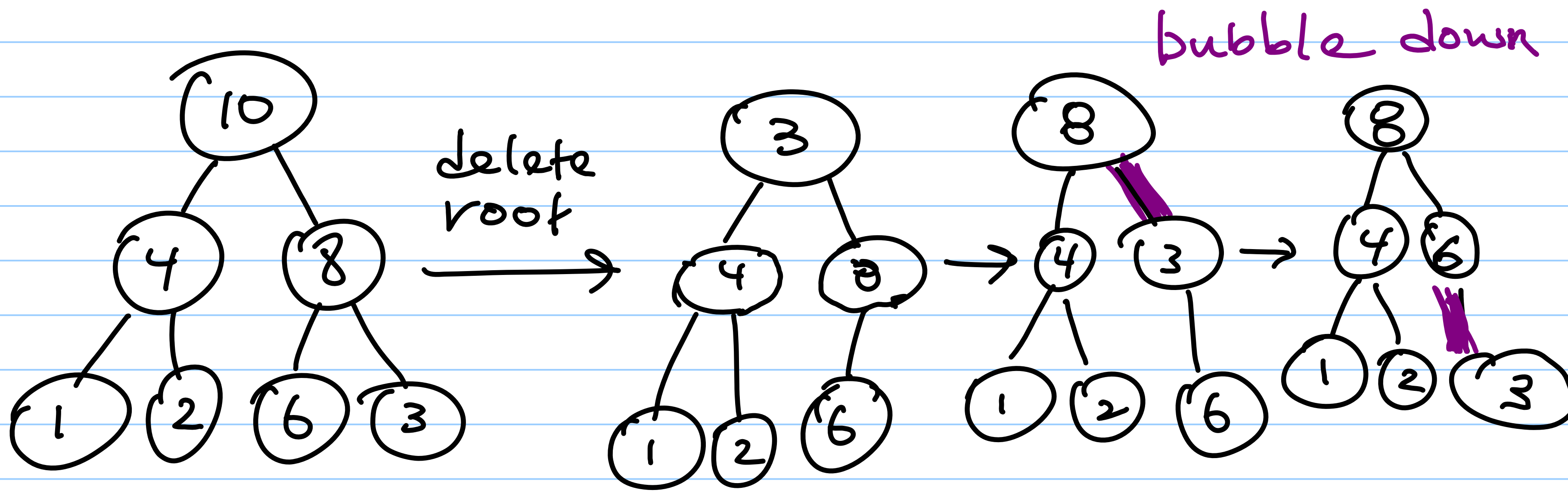
1. Build heap in place: $O(n)$

2. Harvest heap in place:

   Call Delete Root $O(n)$ times.



Each call to DeleteRoot starts with a new node at the ROOT which could bubble all the way down to a leaf.



So worst case path length for all DeleteRoot calls is $\log_2 n$.

So harvesting runtime is $O(n \log n)$.

Runtime of Heapsort?

1. Build heap in place: $O(n)$
2. Harvest heap in place: $O(n \log n)$

TOTAL RUNTIME: $O(n \log n)$

Another use of a heap
is a priority queue.

Convenient because if you have a multitheaded machine
you can return max to one thead that needs it
and heapify in another thread.