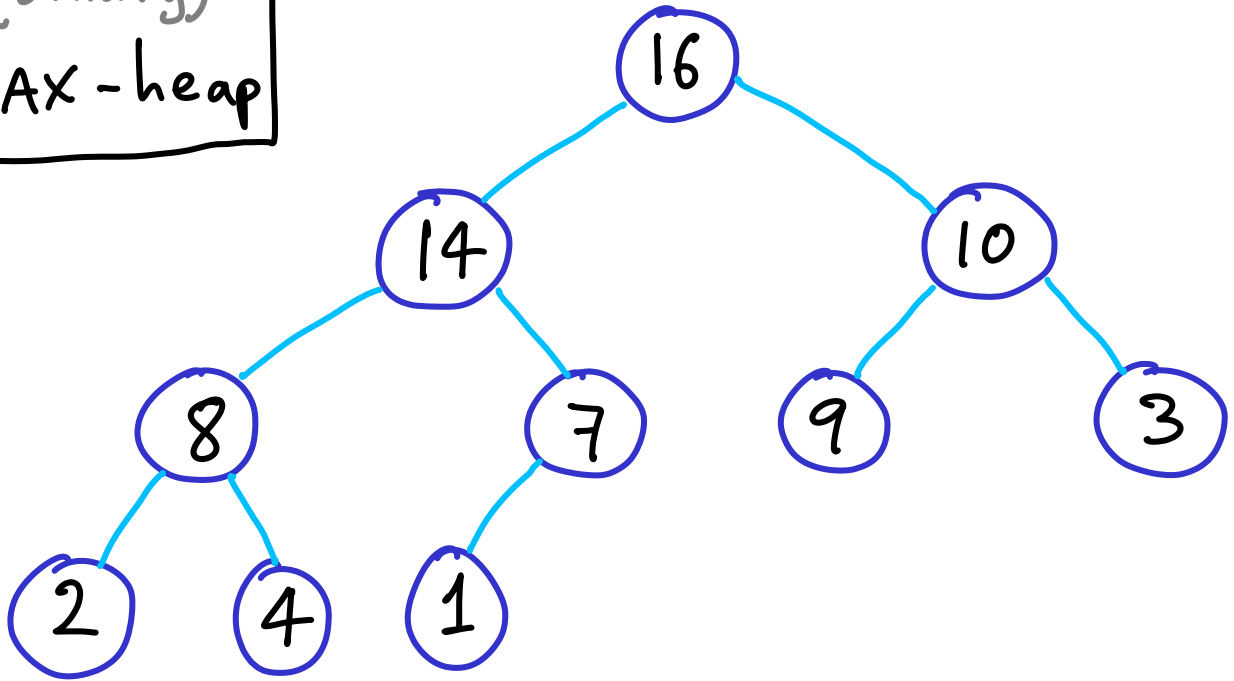
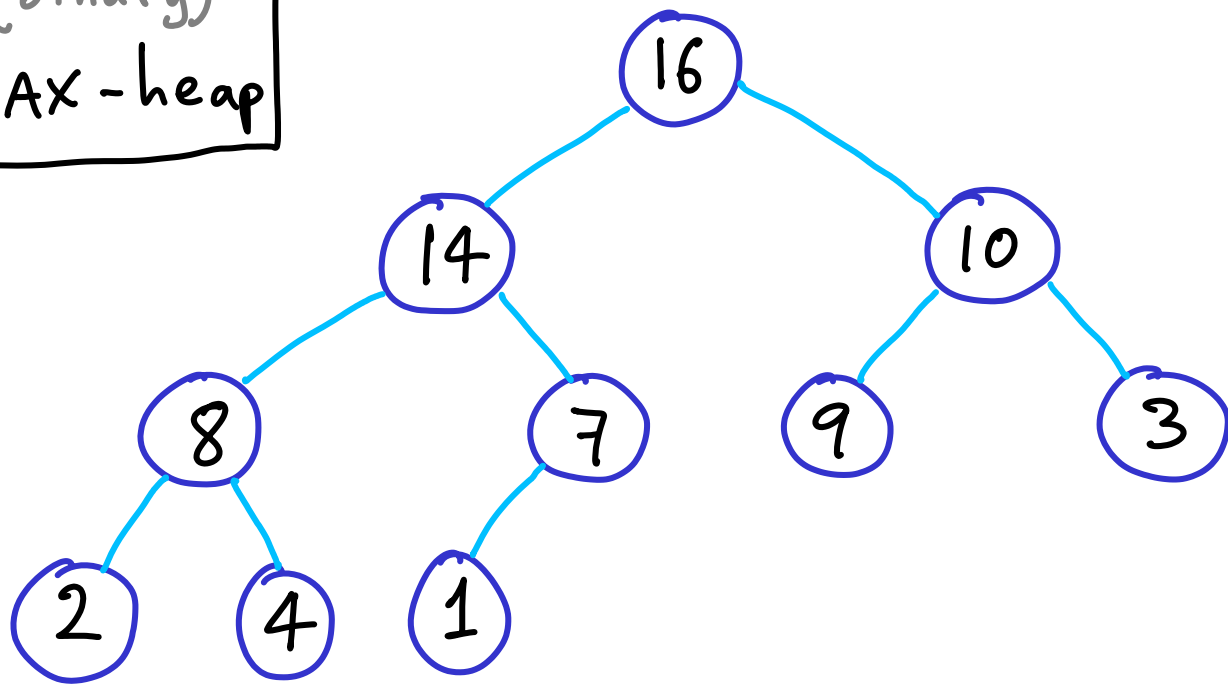


(binary)  
MAX-heap



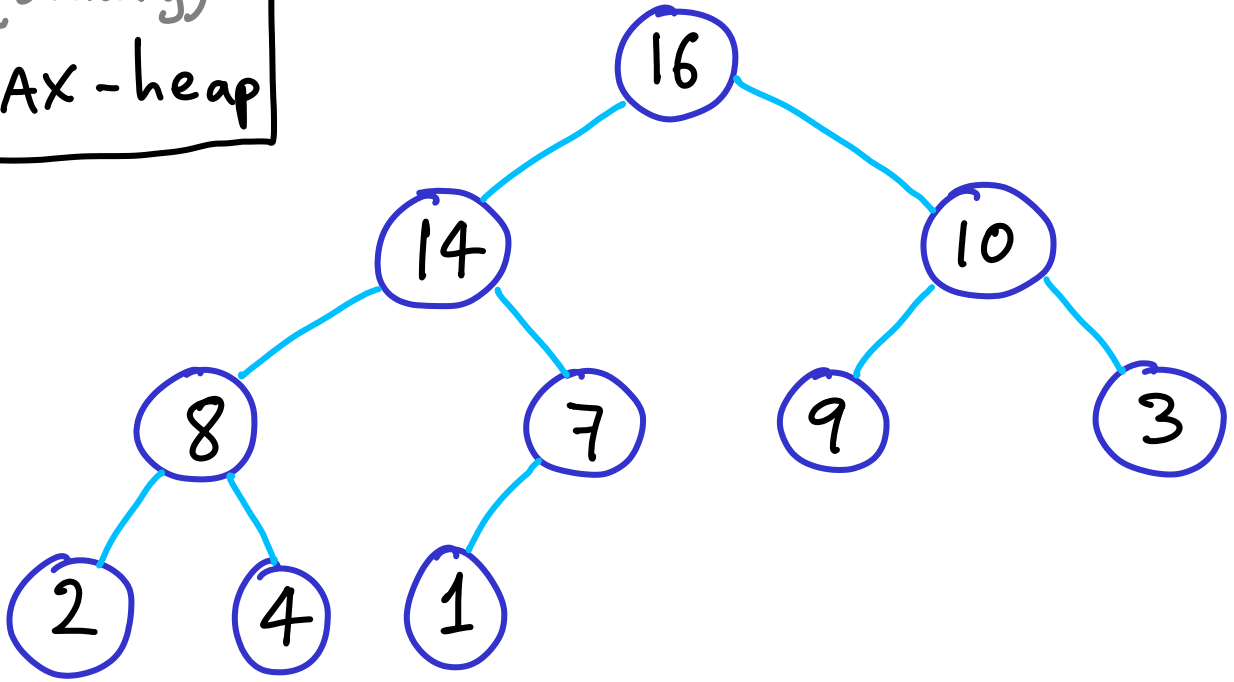
(binary)  
MAX-heap



Rules:

- last row filled from left
- other rows full
- parent > children

(binary)  
MAX-heap



Rules:

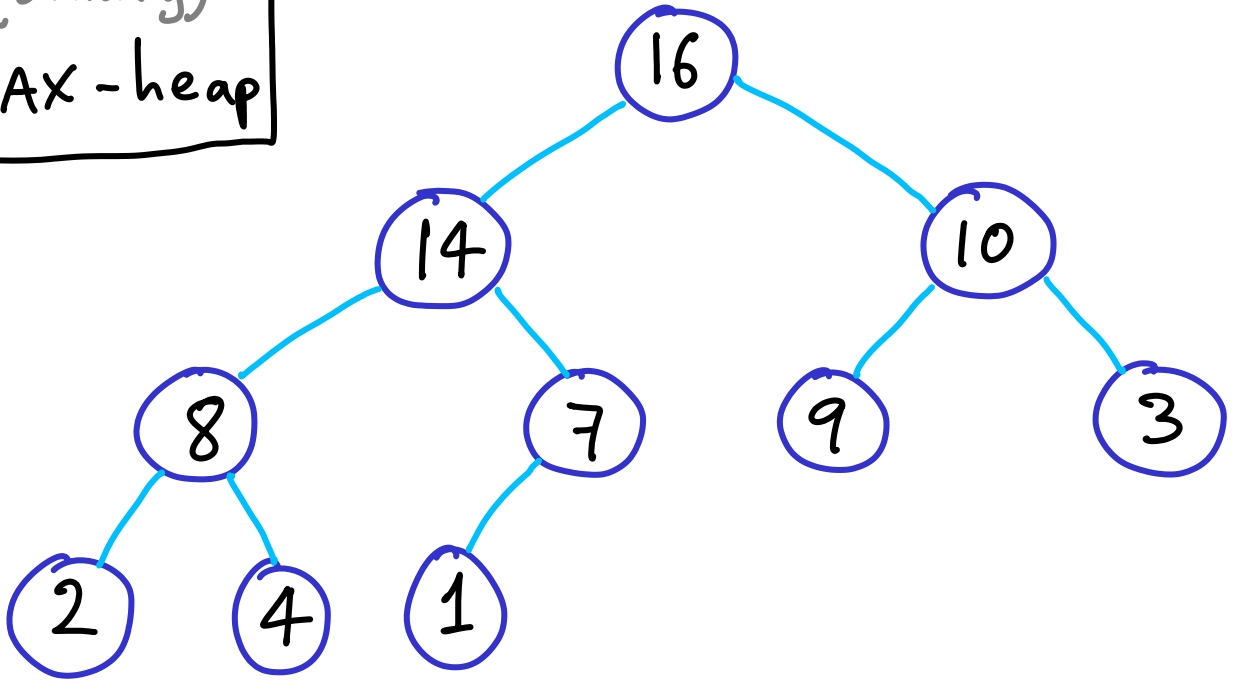
- 1 • last row filled from left
- other rows full
- 2 • parent > children

Rule 1 (structure) is not always required in all applications

E.g. "delete only"

We will assume it's required.

(binary)  
MAX-heap

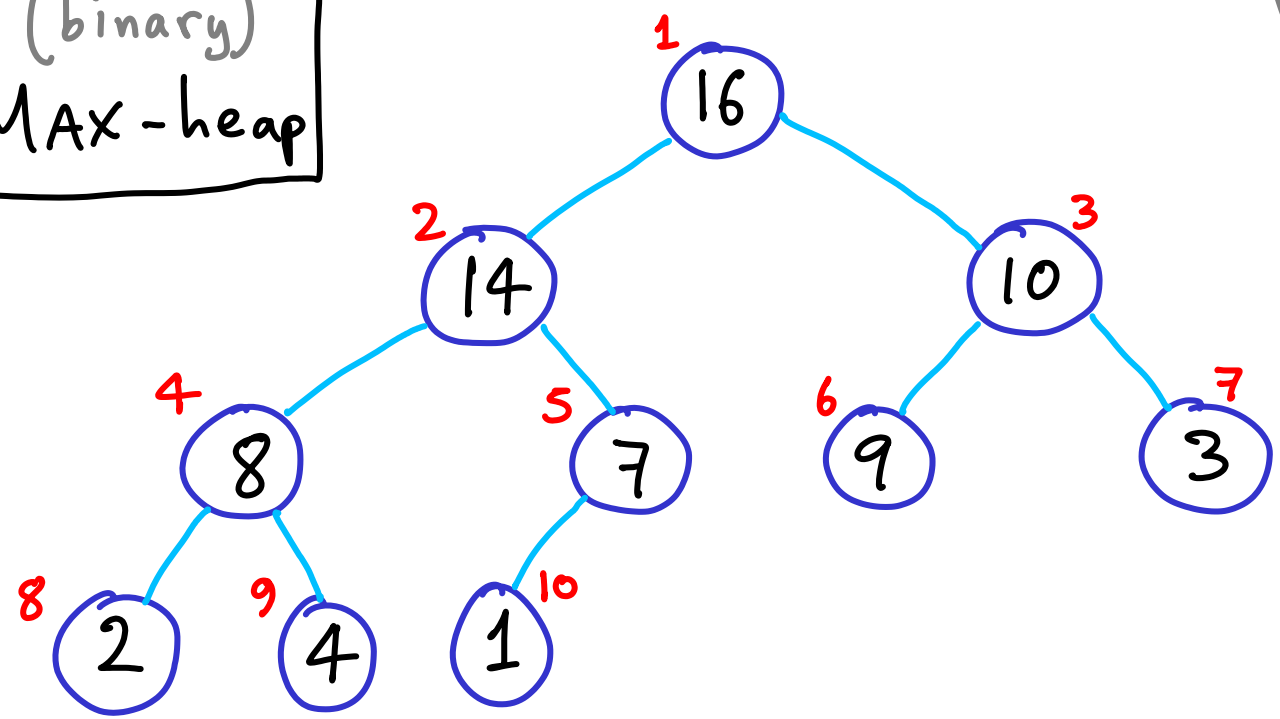


Rules:

- last row filled from left
- other rows full
- parent > children

[Notice every subtree is also a heap]

(binary)  
MAX-heap



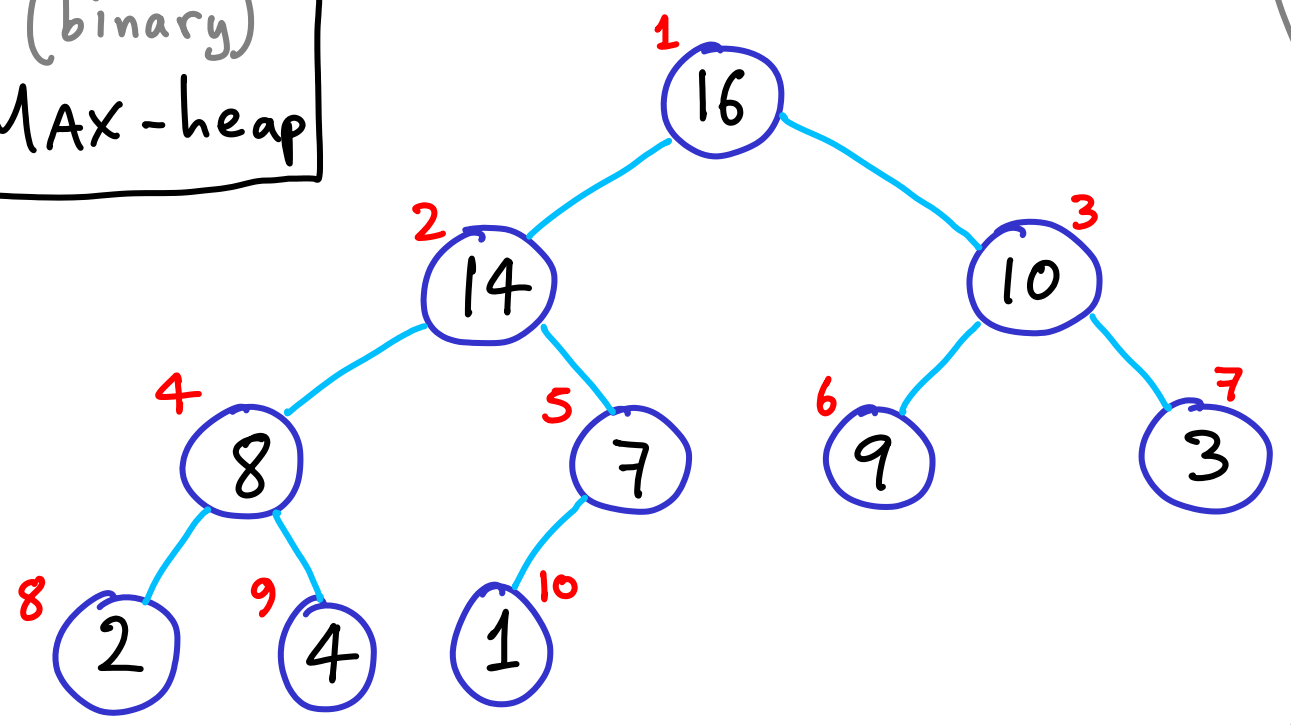
[Notice every subtree is also a heap]

Rules:

- last row filled from left
- other rows full
- parent > children

How can we identify the indices of the children of a node?

(binary)  
MAX-heap



[Notice every subtree is also a heap]

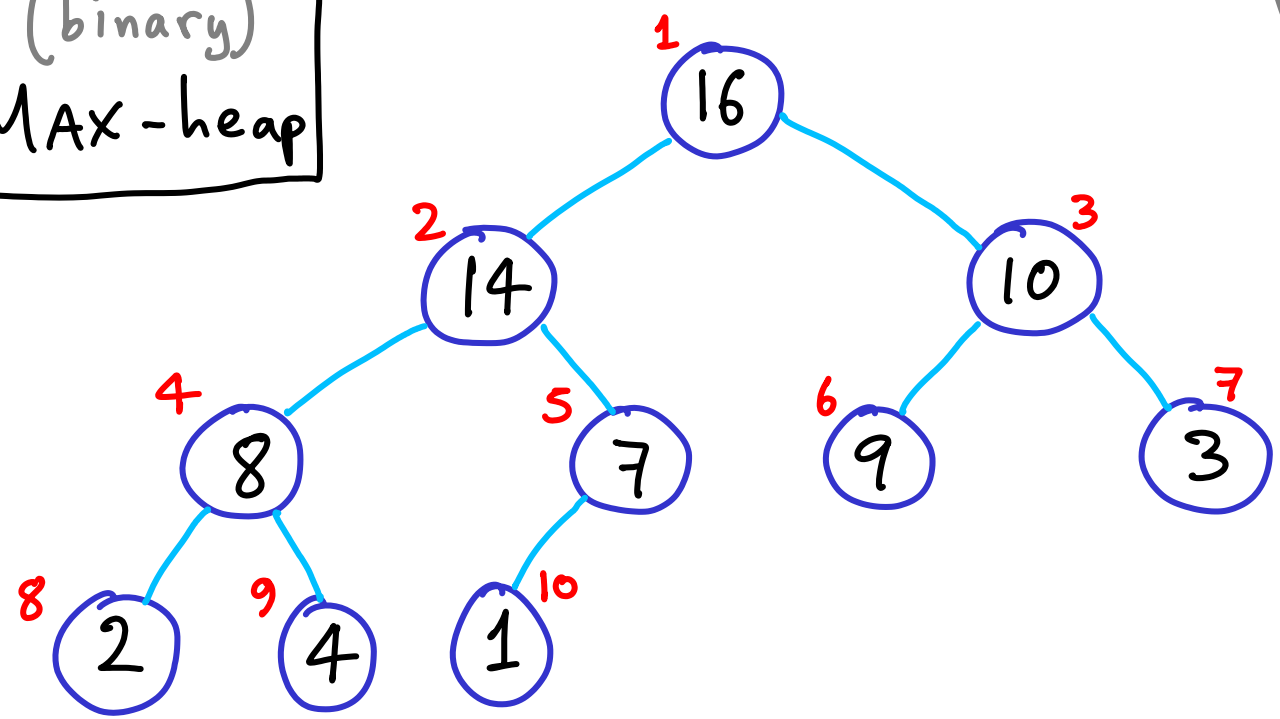
Rules:

- last row filled from left
- other rows full
- parent > children

$$\text{left-child}(i) = 2i$$

$$\text{right-child}(i) = 2i+1$$

(binary)  
MAX-heap



[Notice every subtree is also a heap]

Rules:

- last row filled from left
- other rows full
- parent > children

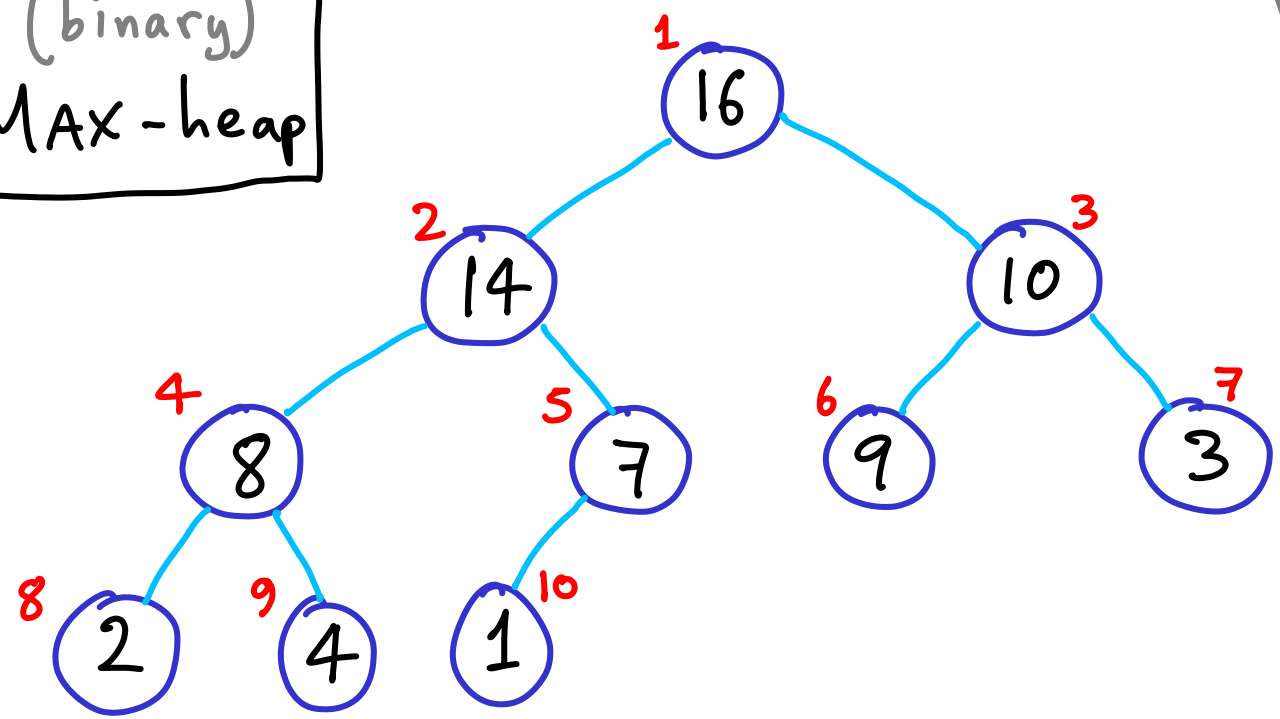
$$\text{parent}(i) = \lfloor i/2 \rfloor$$

$$\text{left-child}(i) = 2i$$

$$\text{right-child}(i) = 2i+1$$



(binary)  
MAX-heap



[Notice every subtree is also a heap]

We can use an array to store a heap. No pointers.

1	2	3	4	5	6	7	8	9	10				
16	14	10	8	7	9	3	2	4	1	...	...	...	...

Rules:

- last row filled from left
- other rows full
- parent > children

$$\text{parent}(i) = \lfloor i/2 \rfloor$$

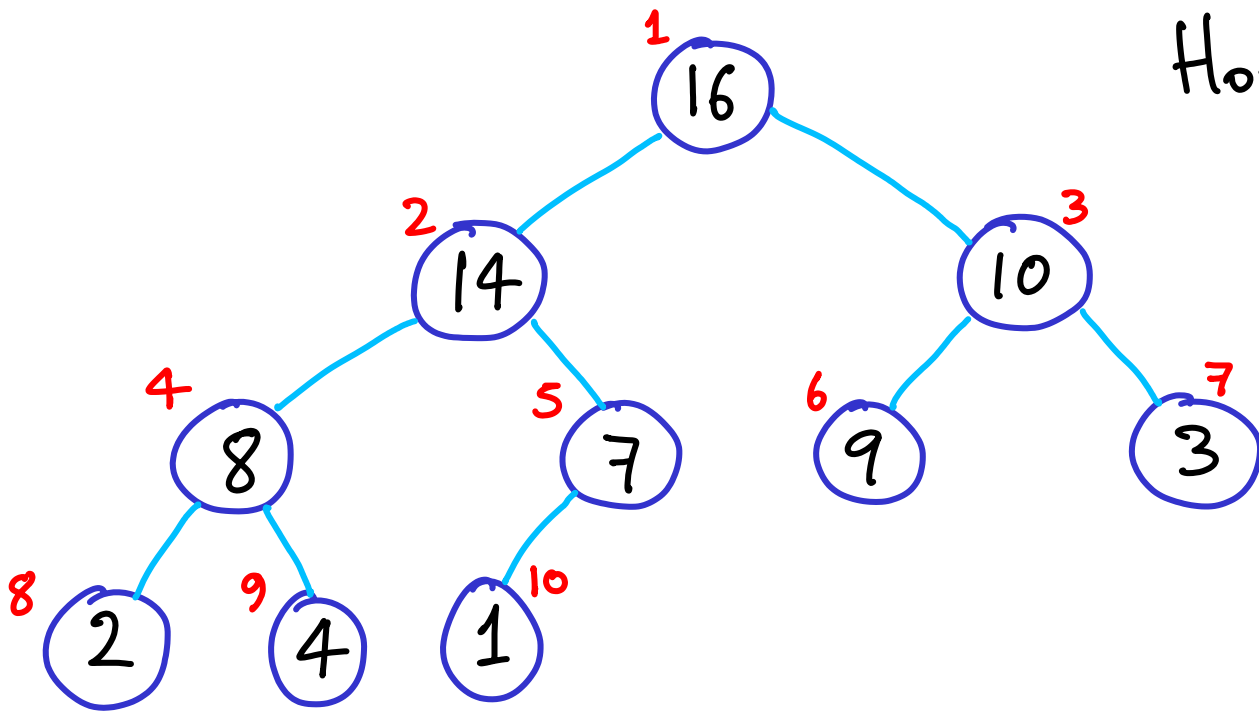
$$\text{left-child}(i) = 2i$$

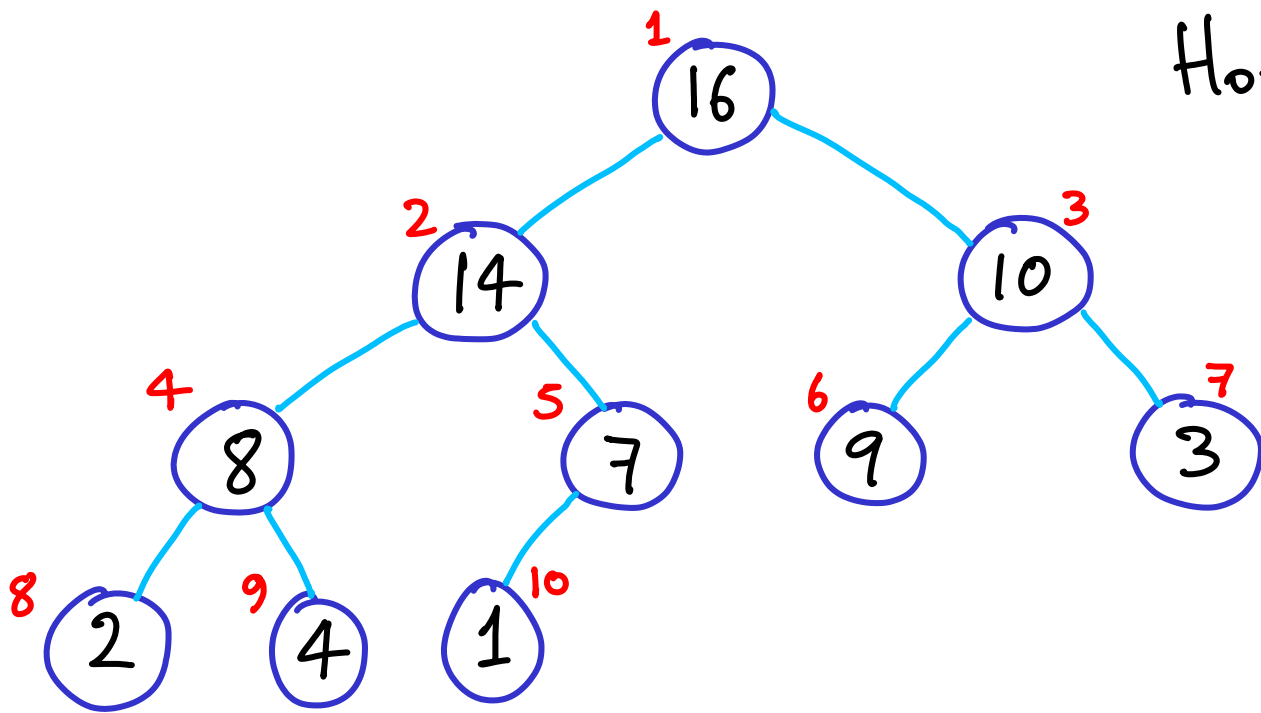
$$\text{right-child}(i) = 2i+1$$





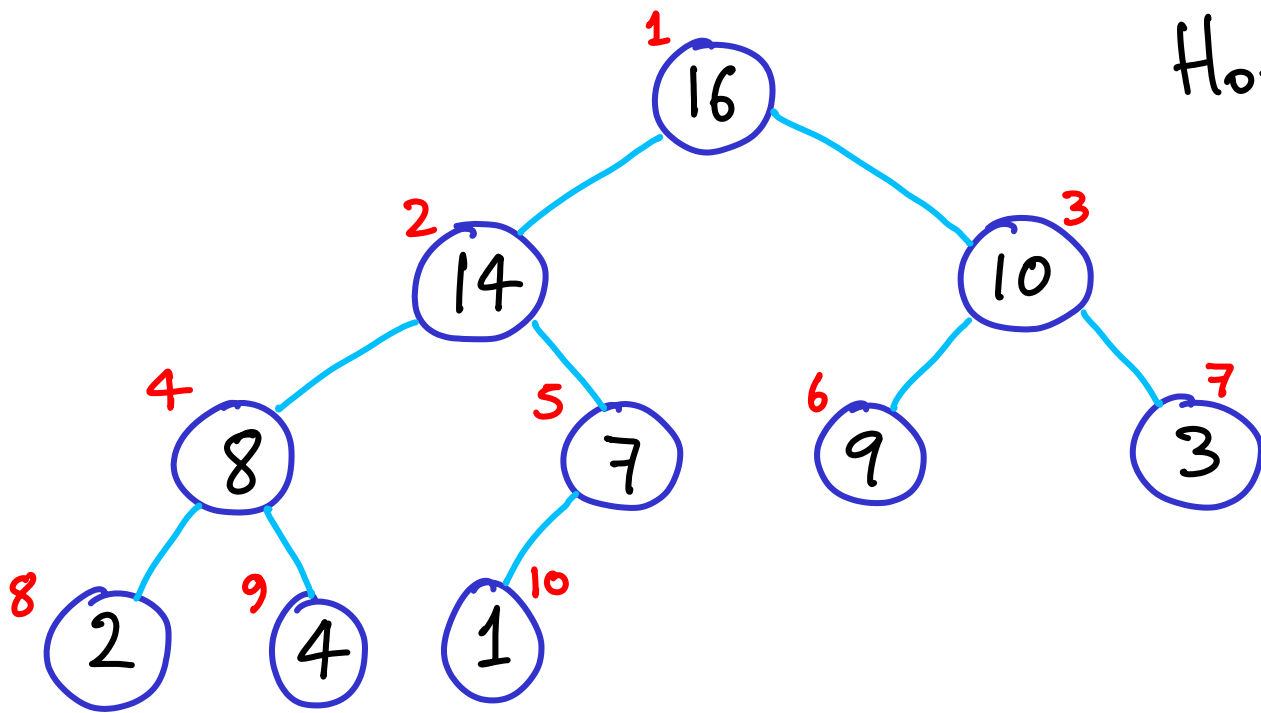
How does this relate to sorting?





How does this relate to sorting?

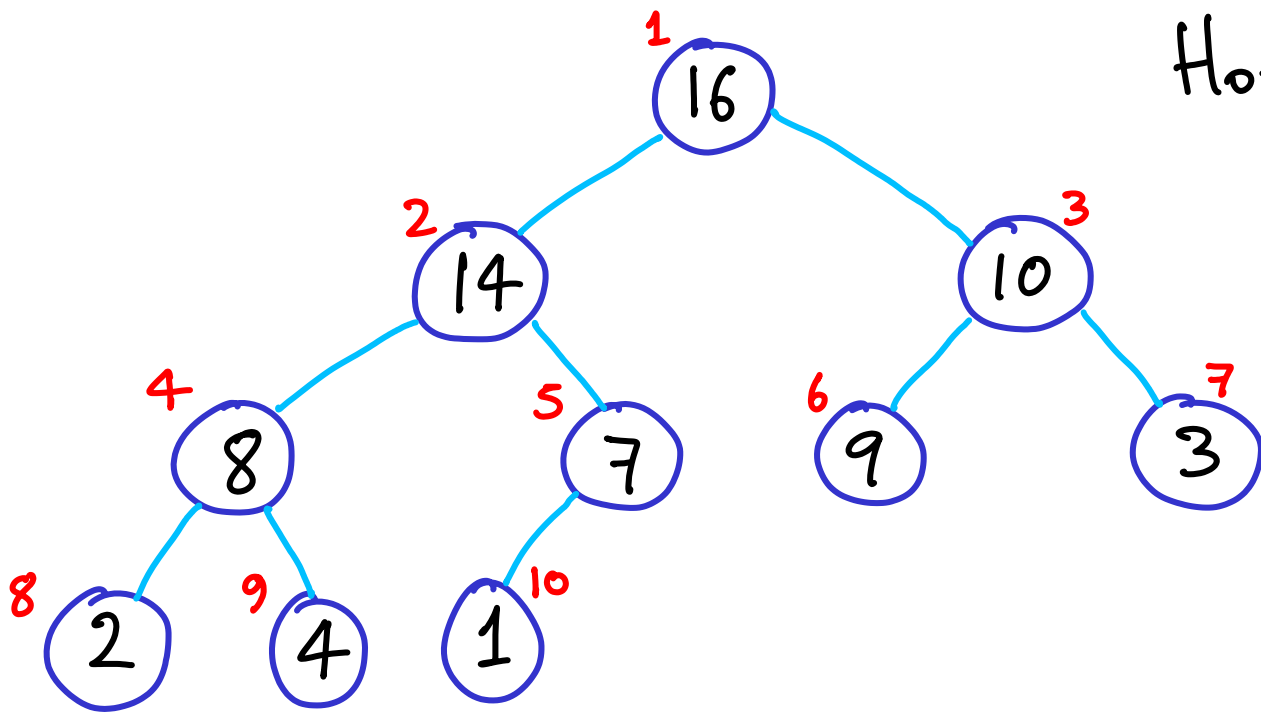
Largest element is on top.



How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.



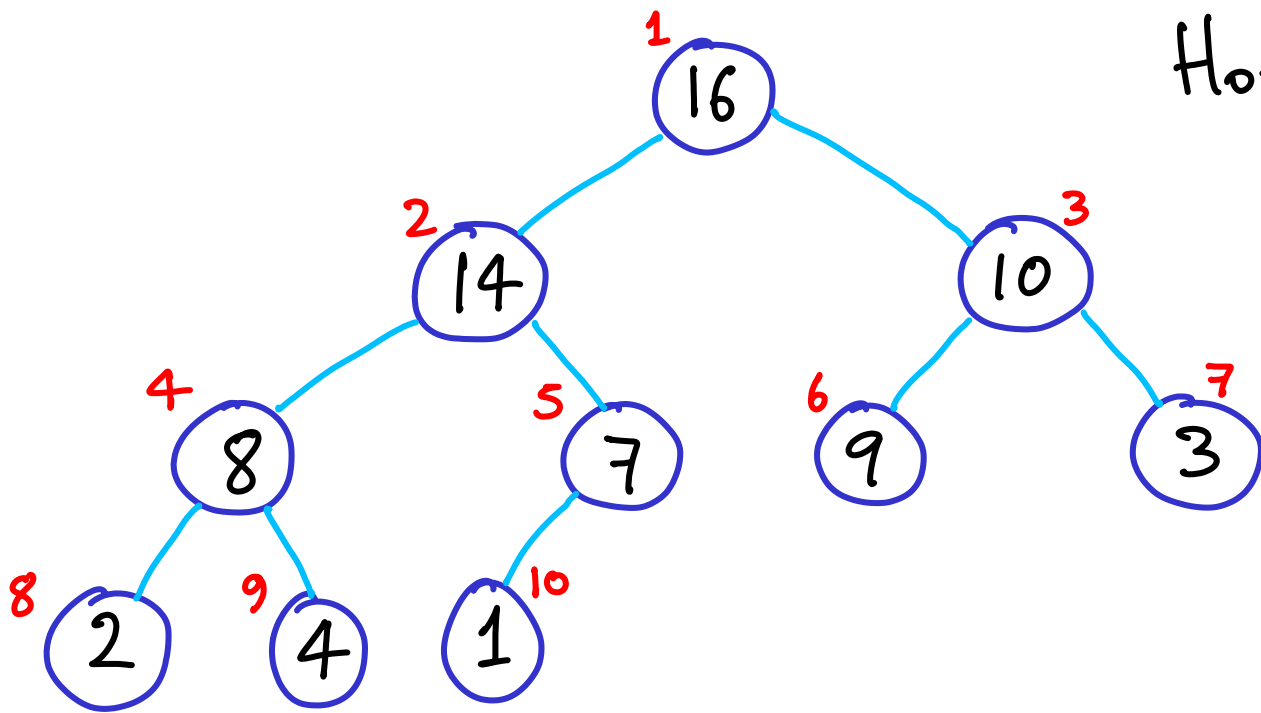
How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

3rd largest is

↳ in level 2



How does this relate to sorting?

Largest element is on top.

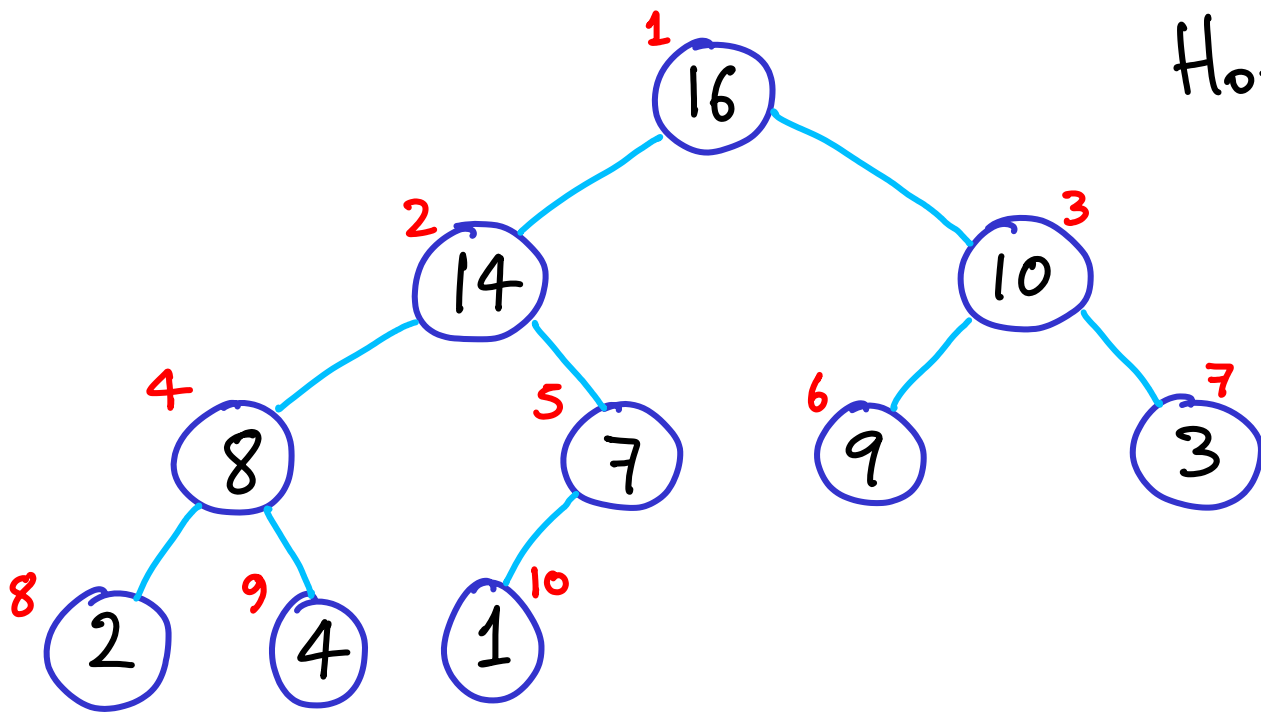
2nd largest is in level 2.

3rd largest is

↳ in level 2

OR

↳ in level 3



How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

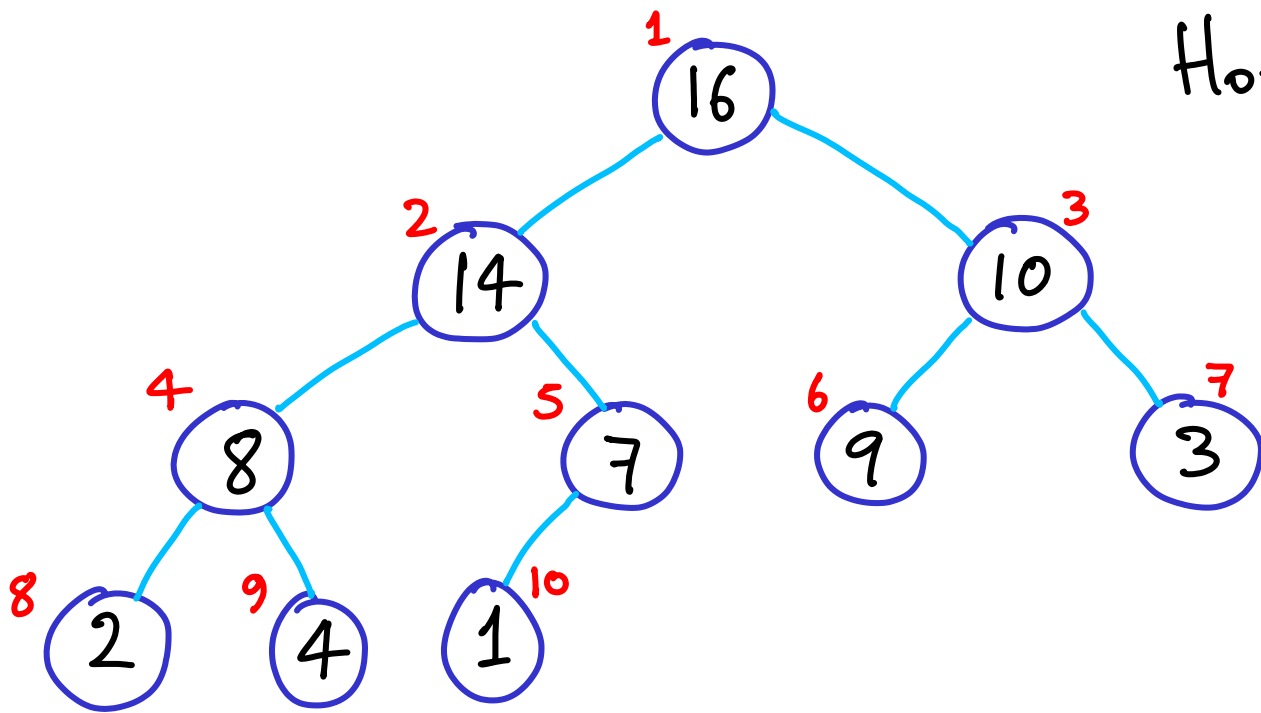
3rd largest is

↳ in level 2

OR

↳ in level 3

& child of 2nd



How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

3rd largest is

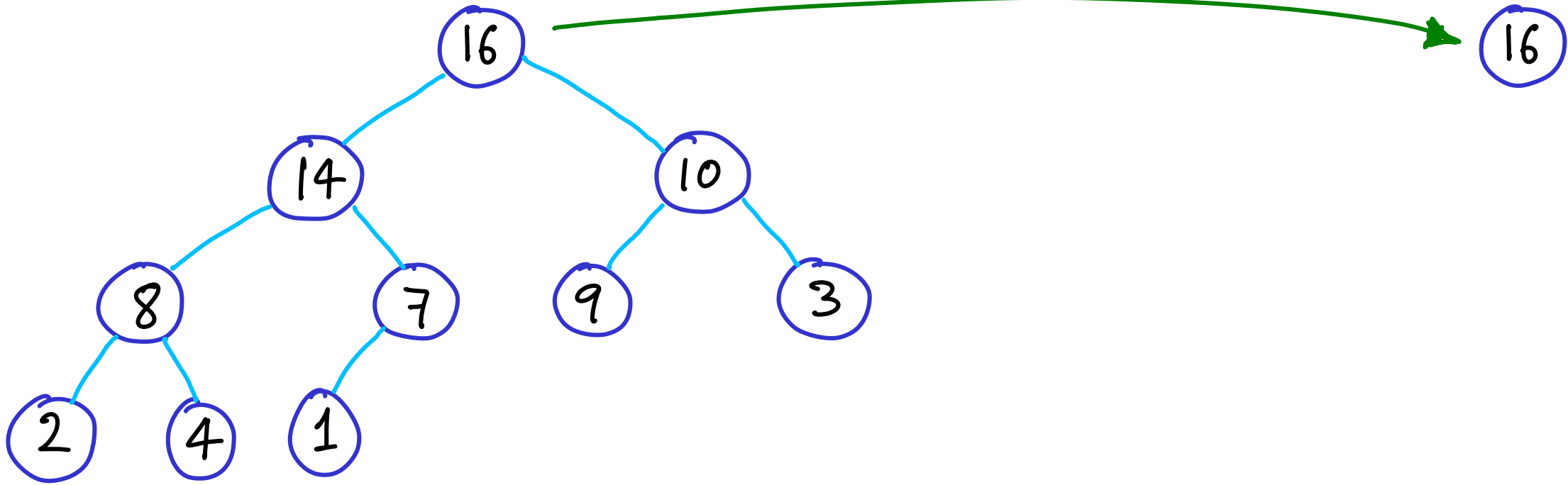
↳ in level 2

OR

↳ in level 3

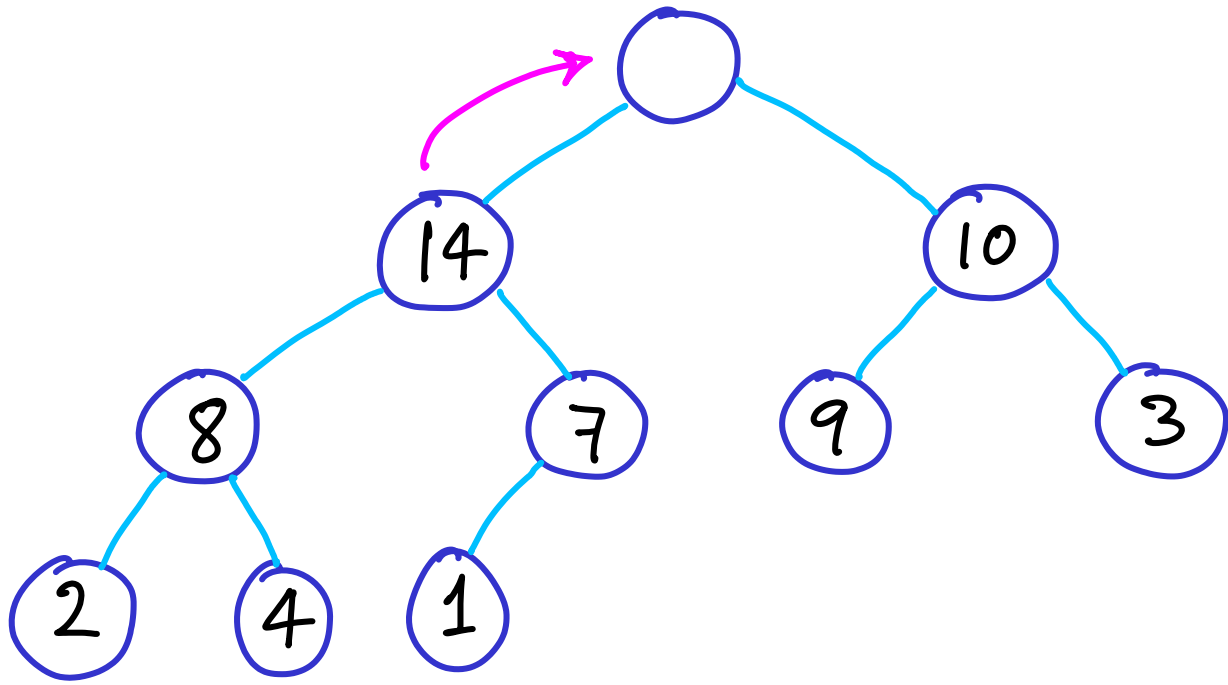
& child of 2nd

⋮  
getting messy



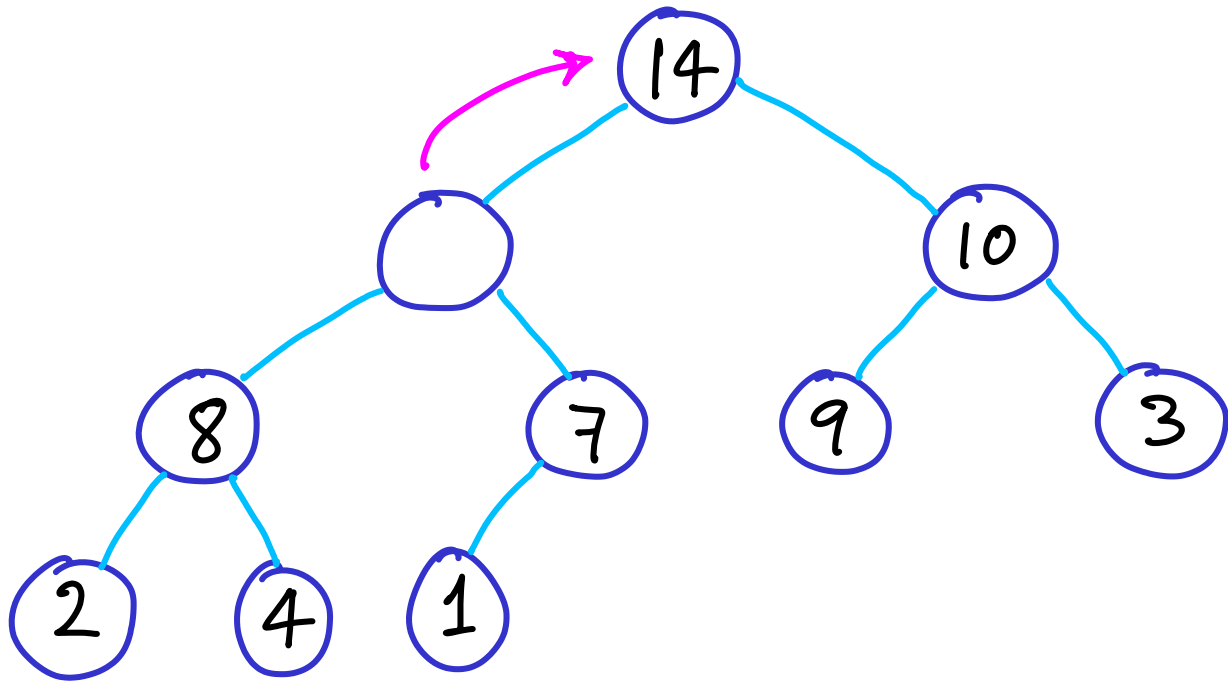
extract MAX





16

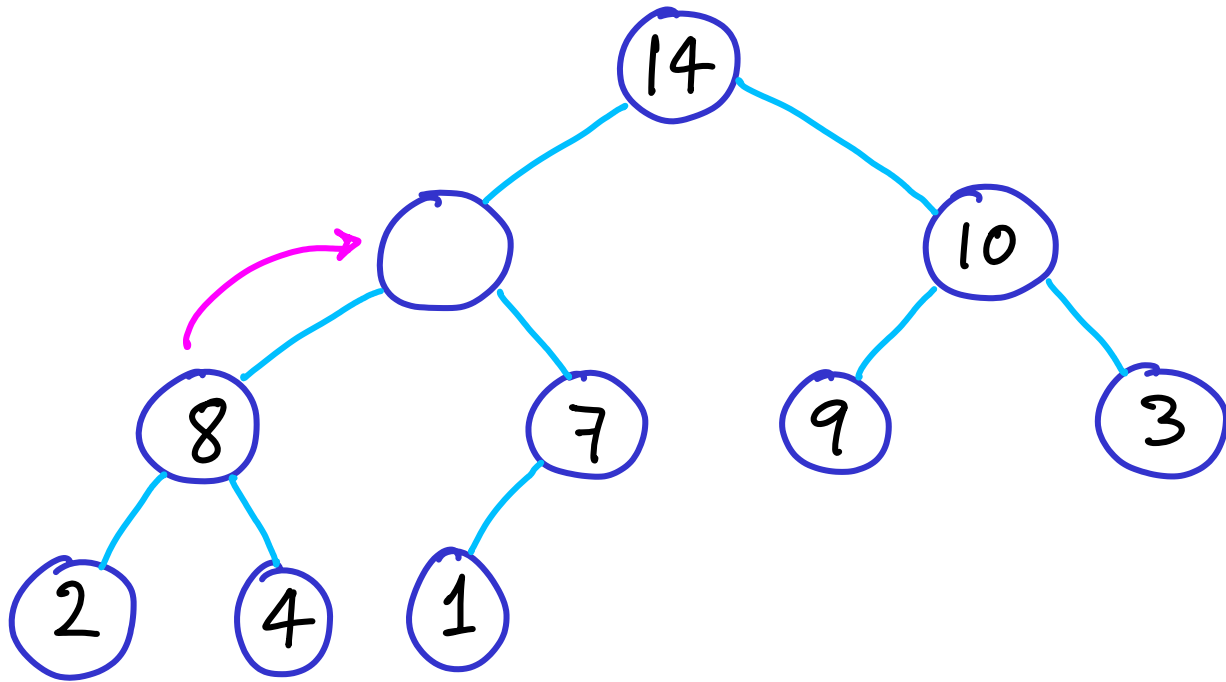
attempt to extract MAX & restore heap



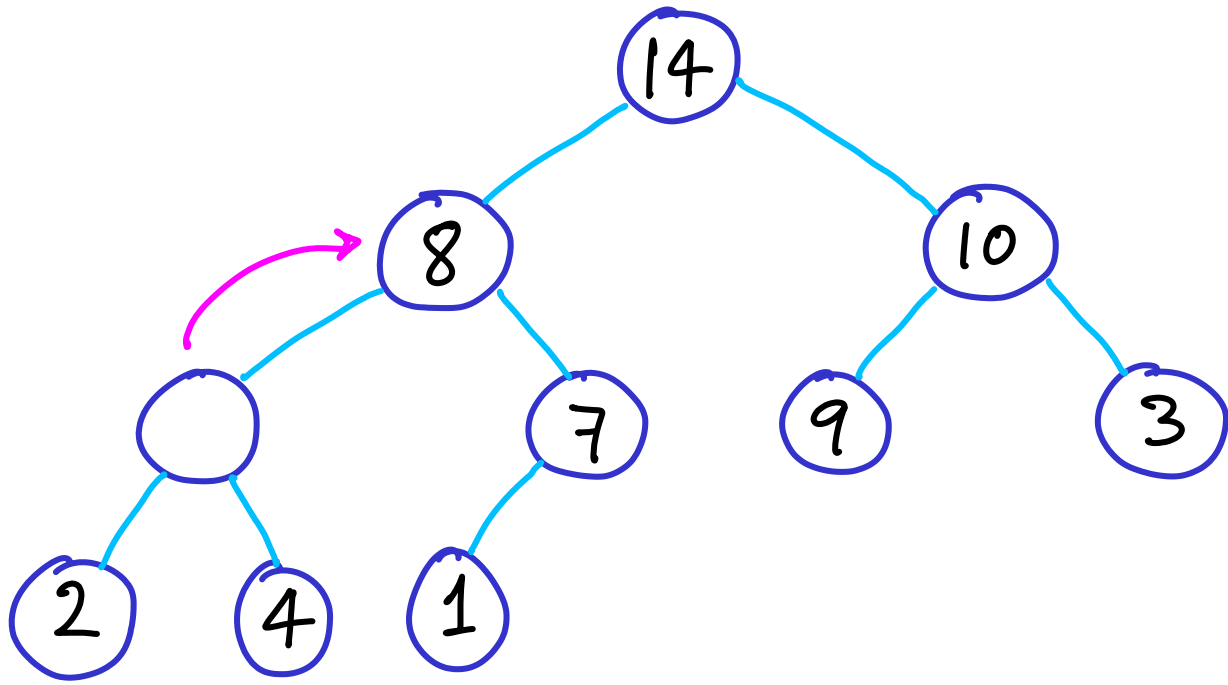
16

attempt to extract MAX & restore heap

16

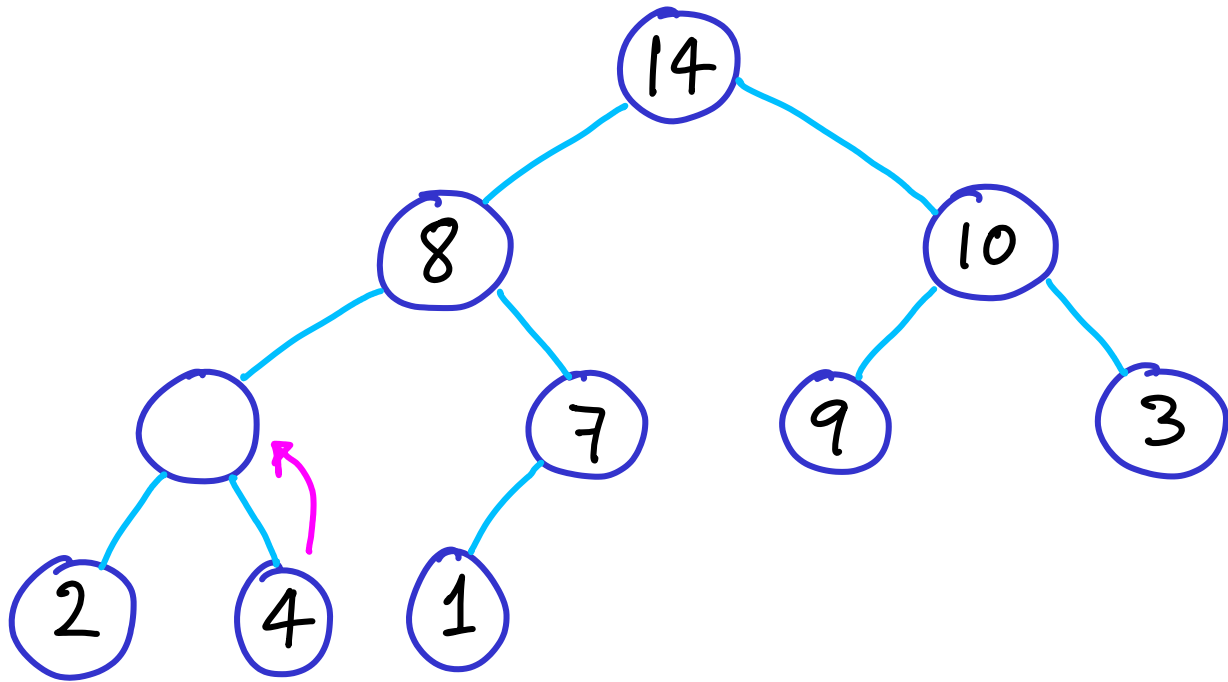


attempt to extract MAX & restore heap



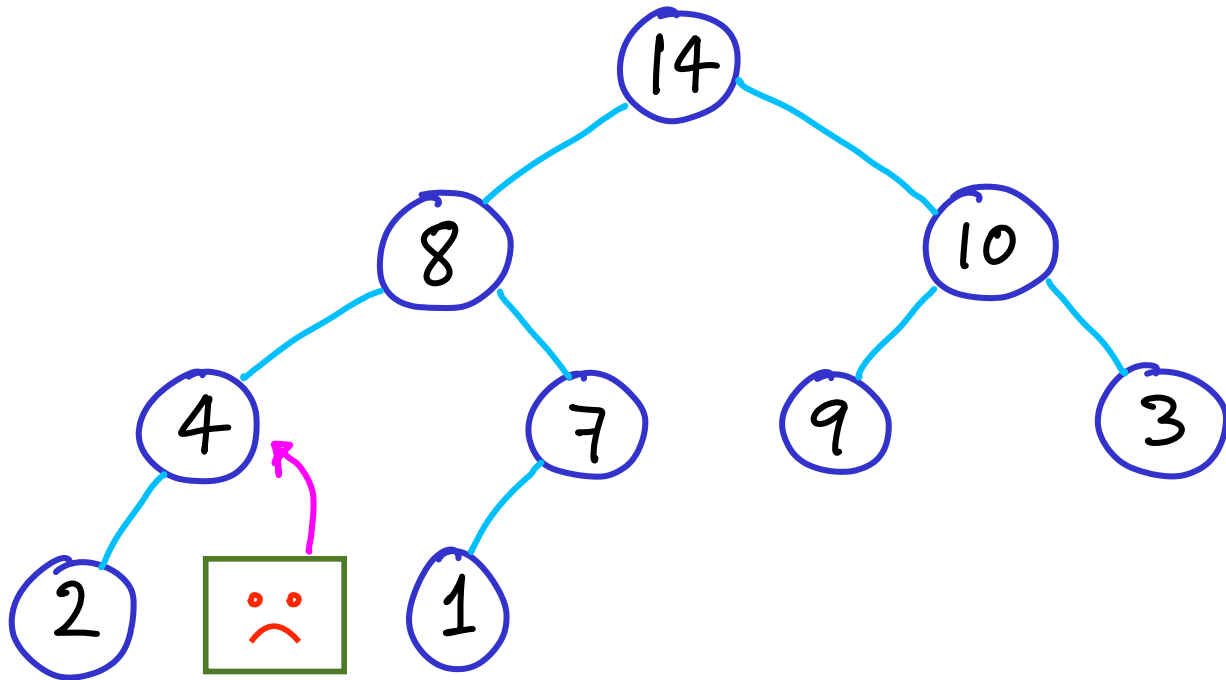
16

attempt to extract MAX & restore heap



16

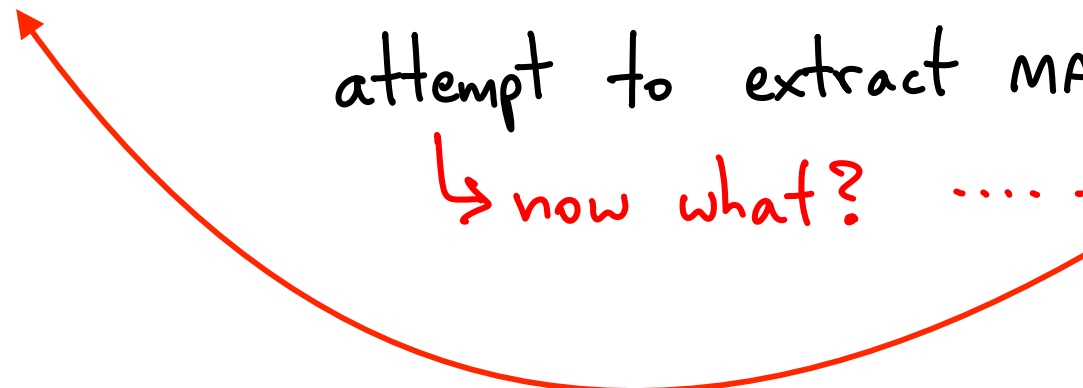
attempt to extract MAX & restore heap

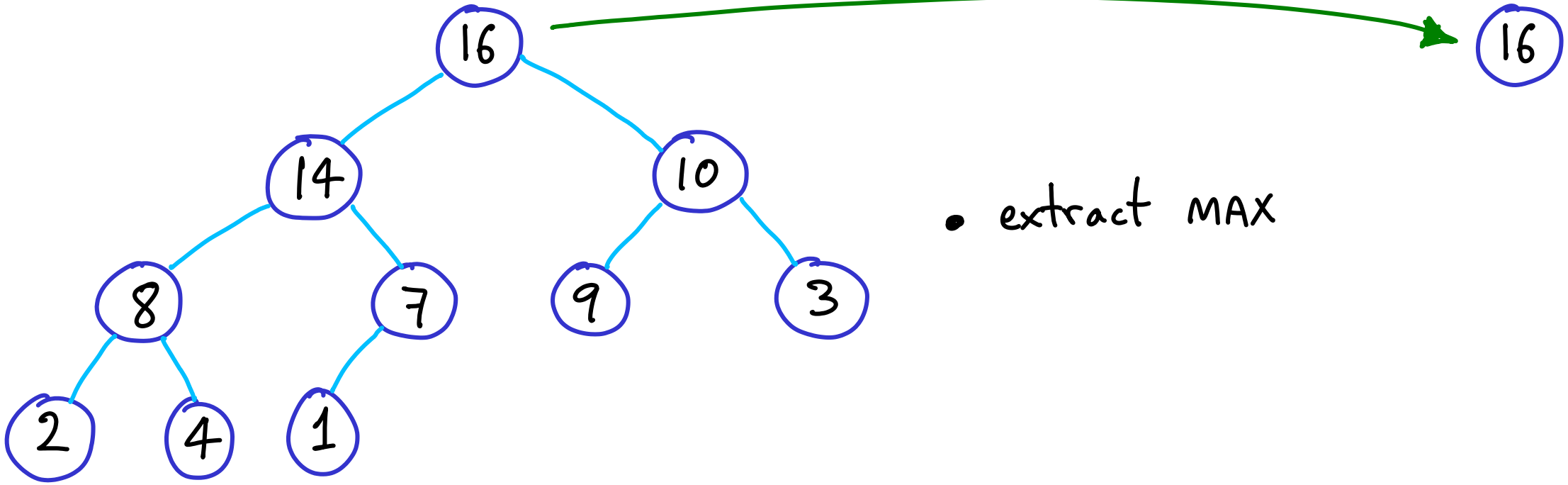


attempt to extract MAX & restore heap

↳ now what? .... failed

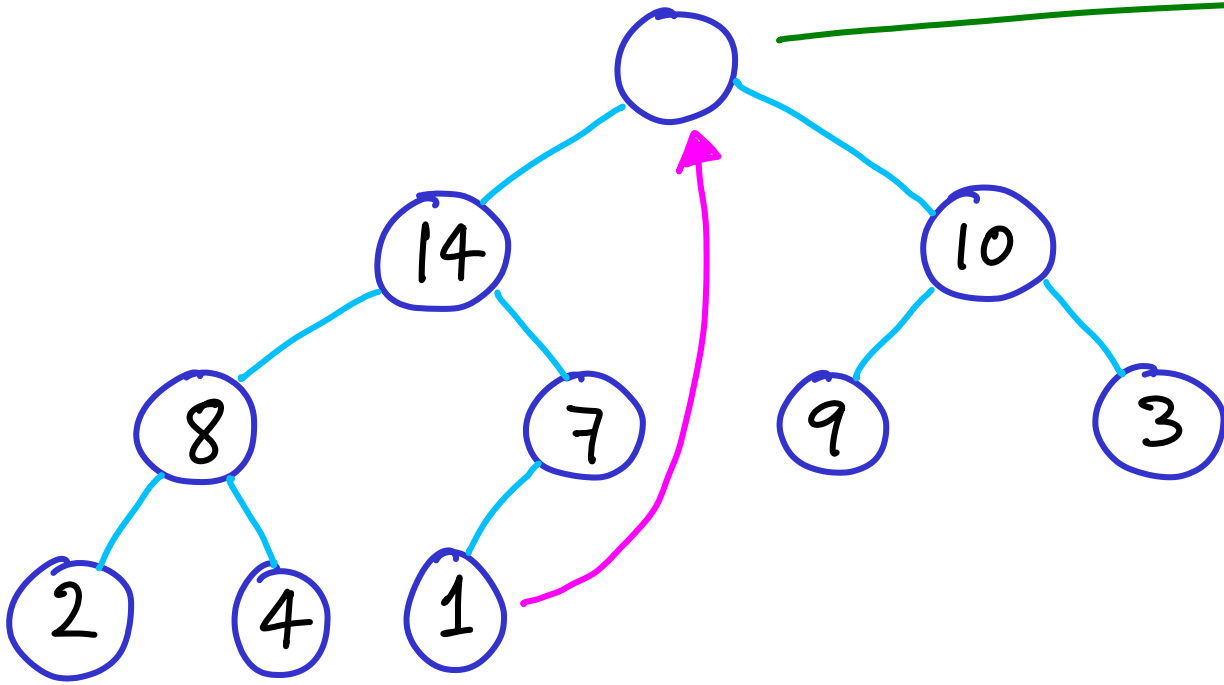
(we want to preserve the structure)





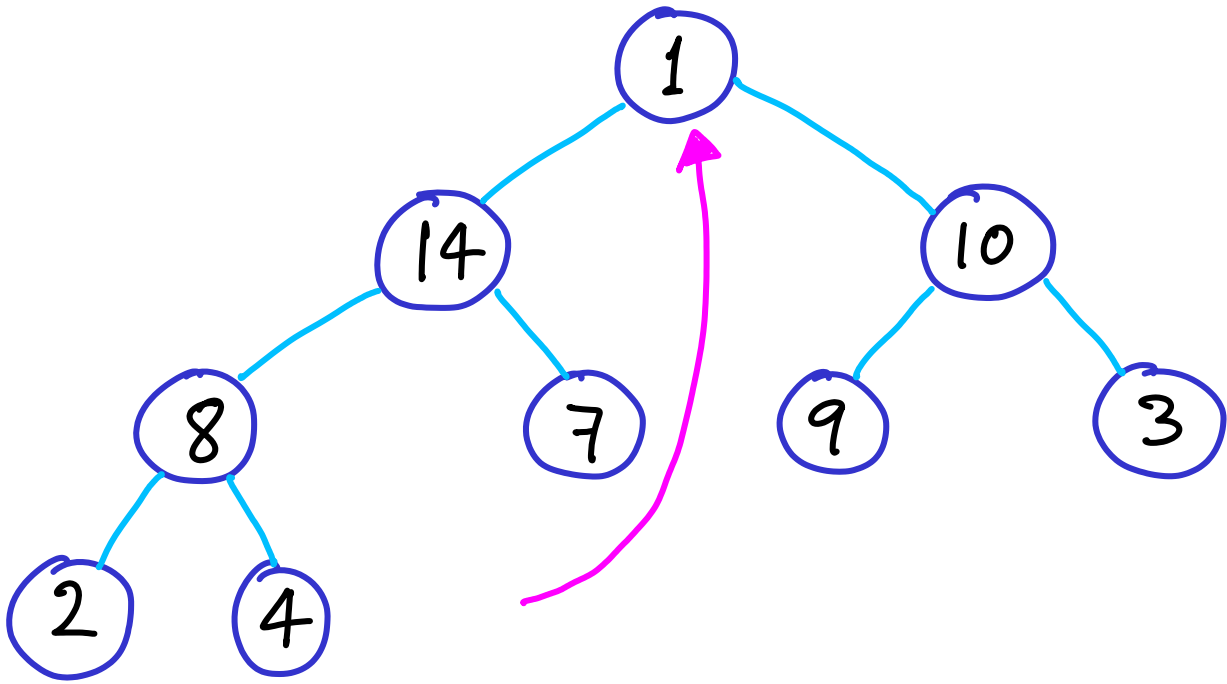
• extract MAX

Try again:  
Modify extraction



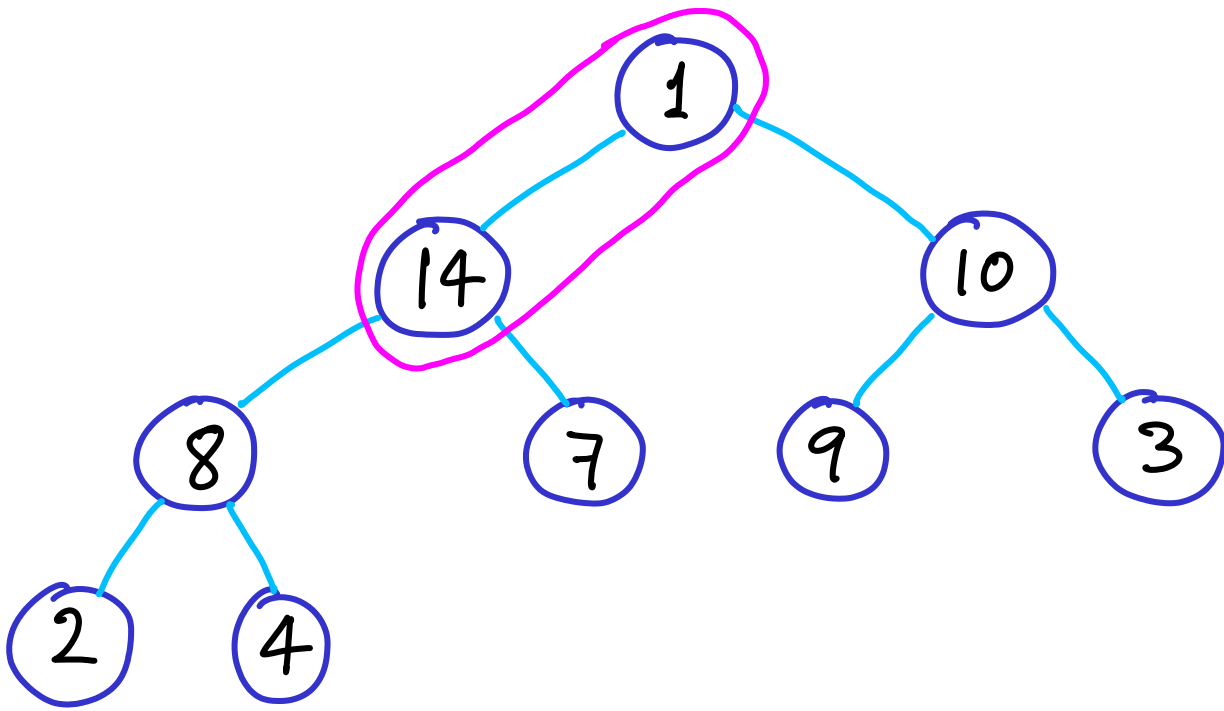
- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)





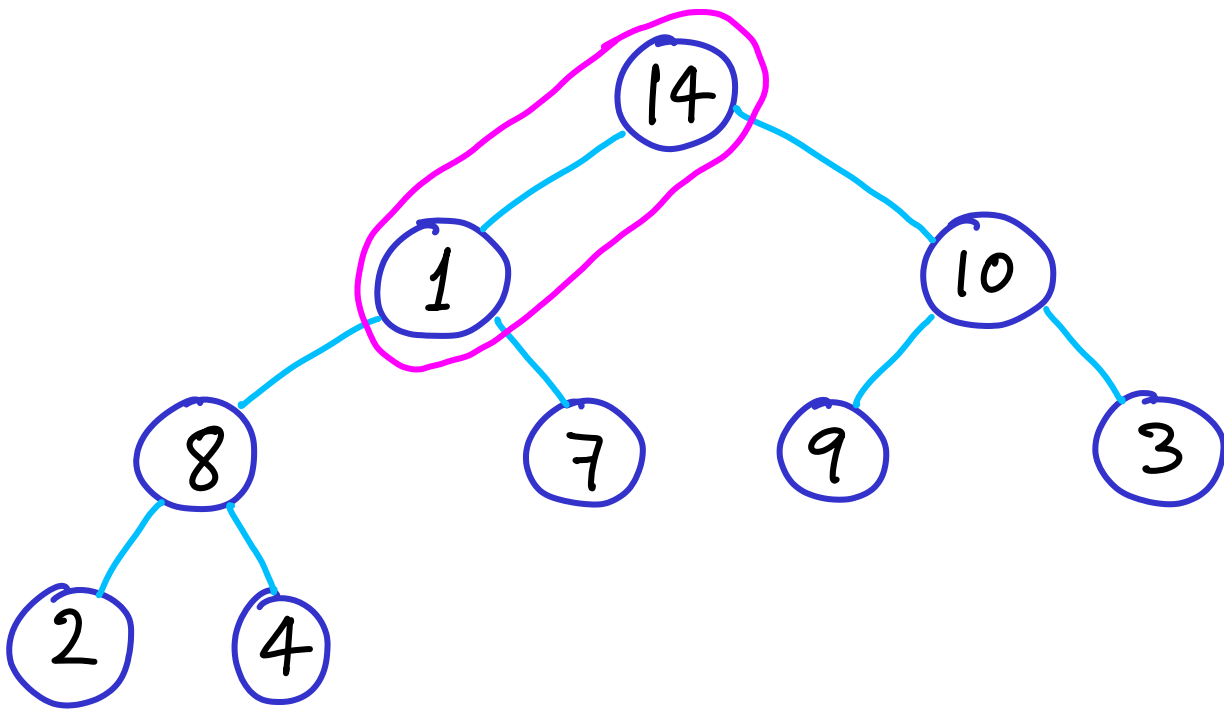
16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)



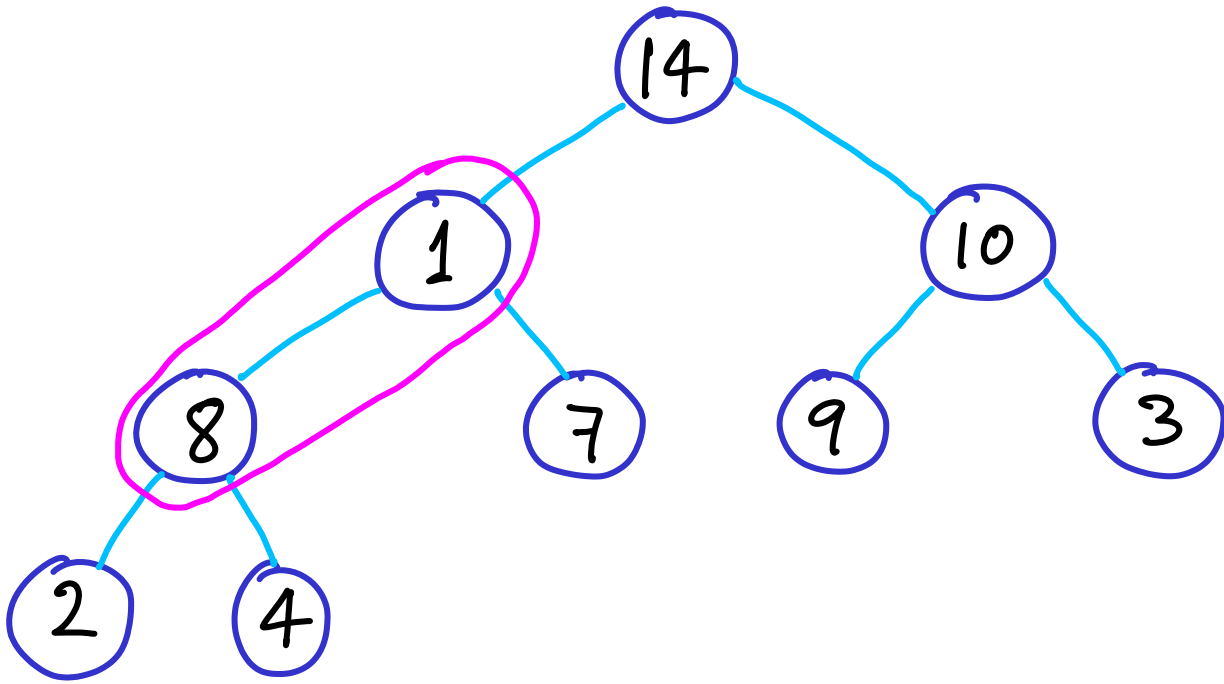
16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)



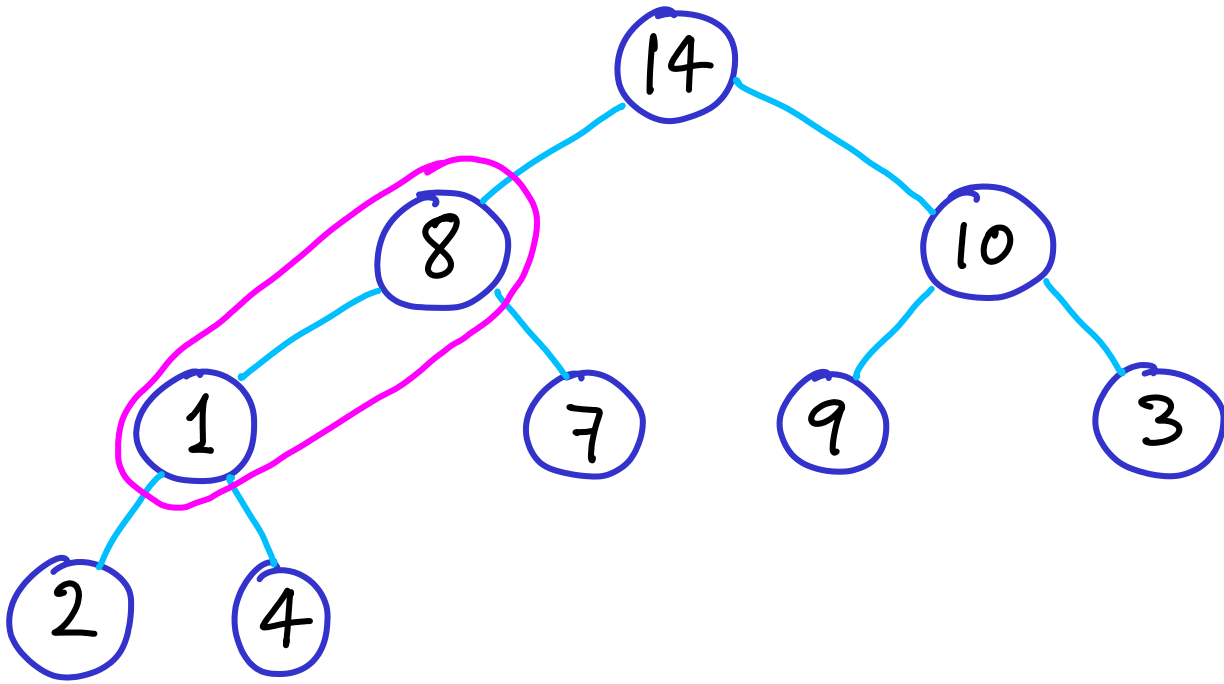
16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)



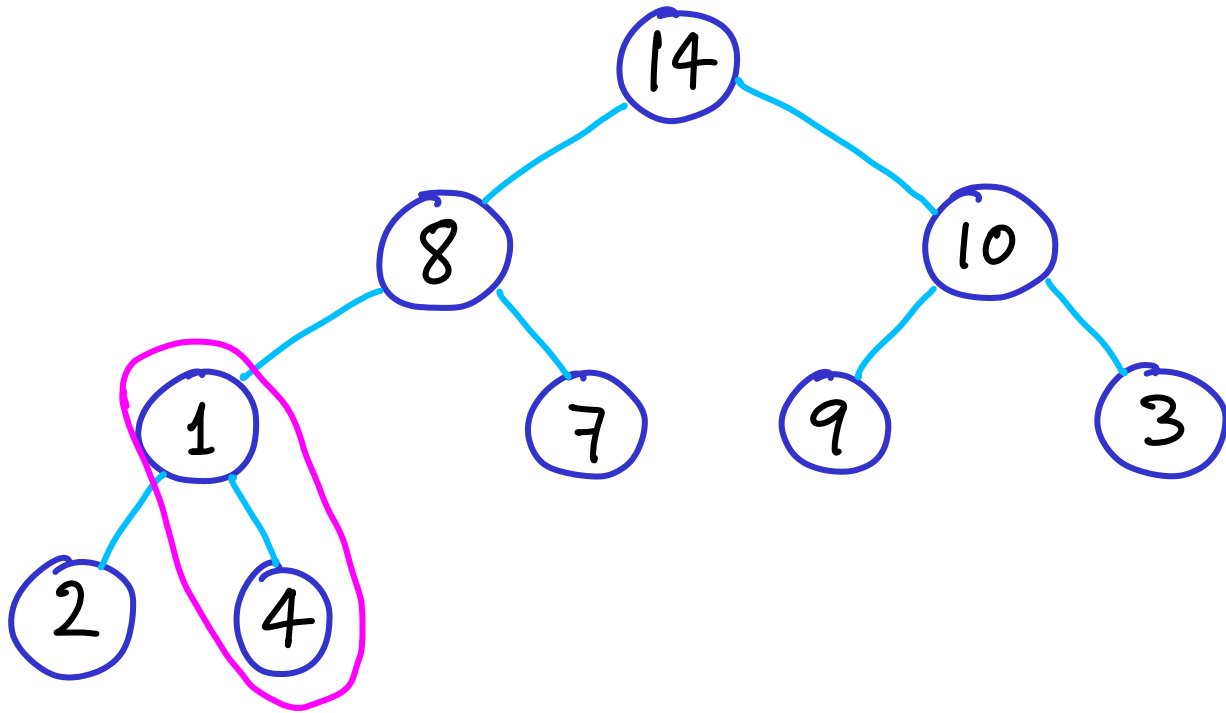
16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)
- repeat downward while needed



16

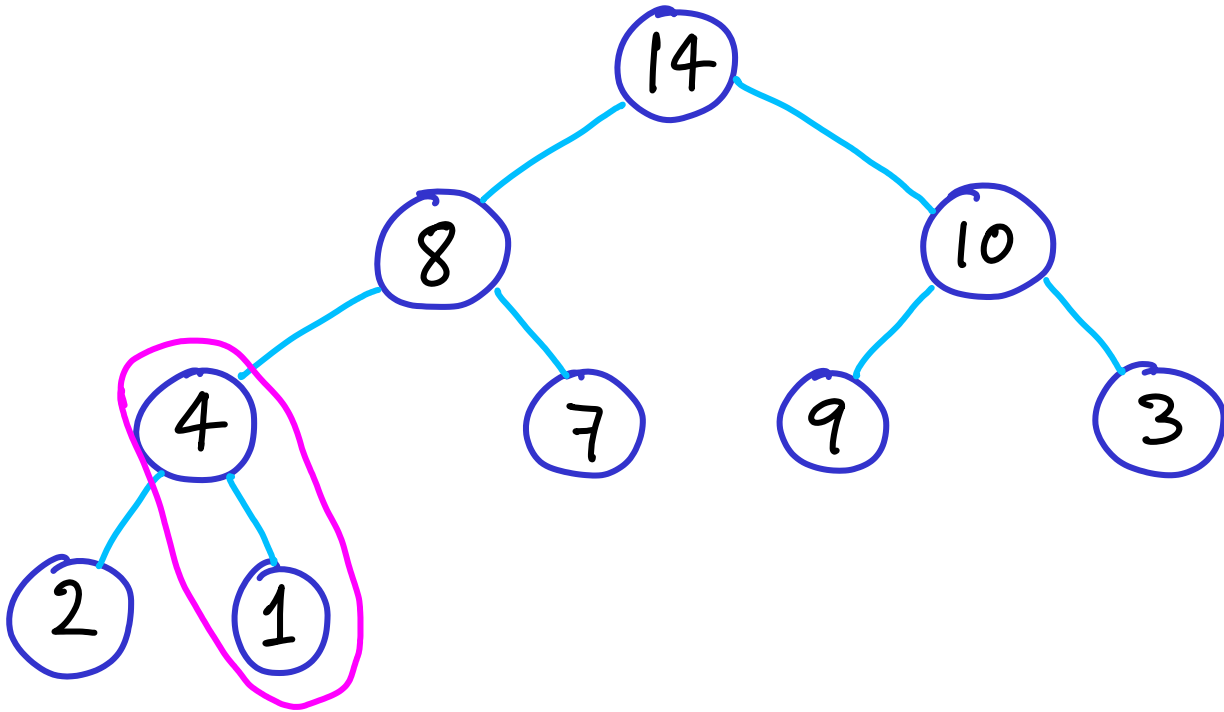
- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)
- repeat downward while needed



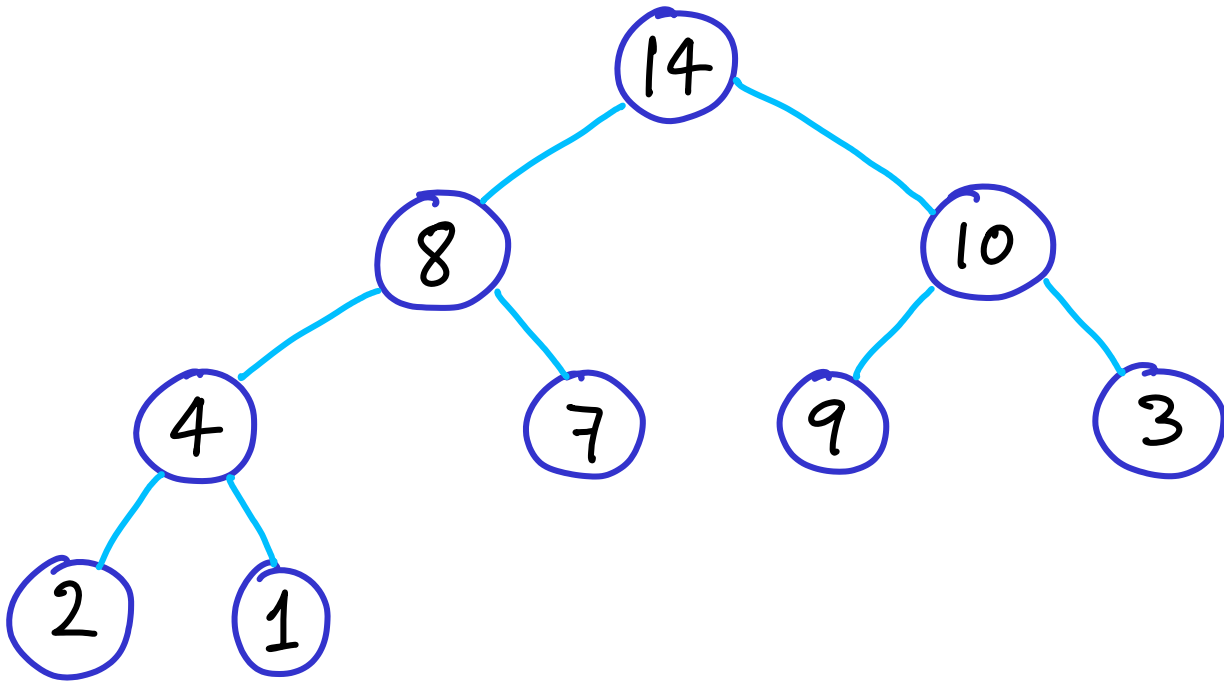
16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)
- repeat downward while needed

16



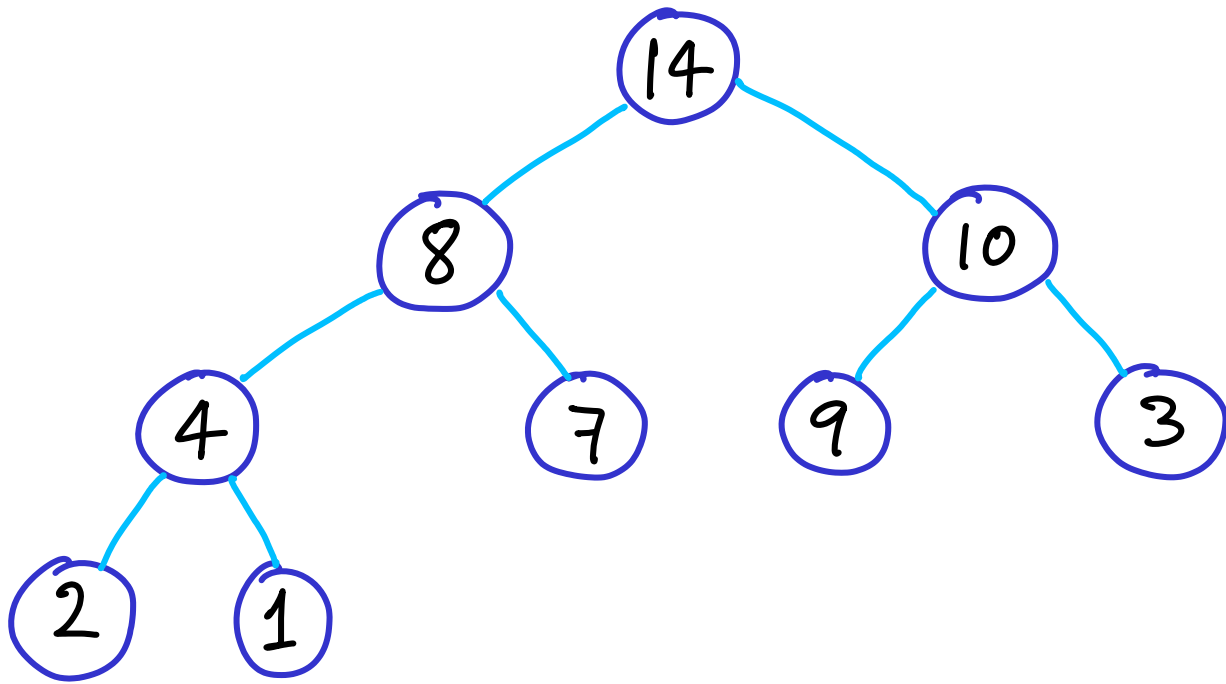
- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)
- repeat downward while needed



16

- extract MAX
  - place rightmost leaf at top  
(strange move... it's small!)
  - swap top w/ largest child  
(locally restore parent > child)
  - repeat downward while needed
- time?

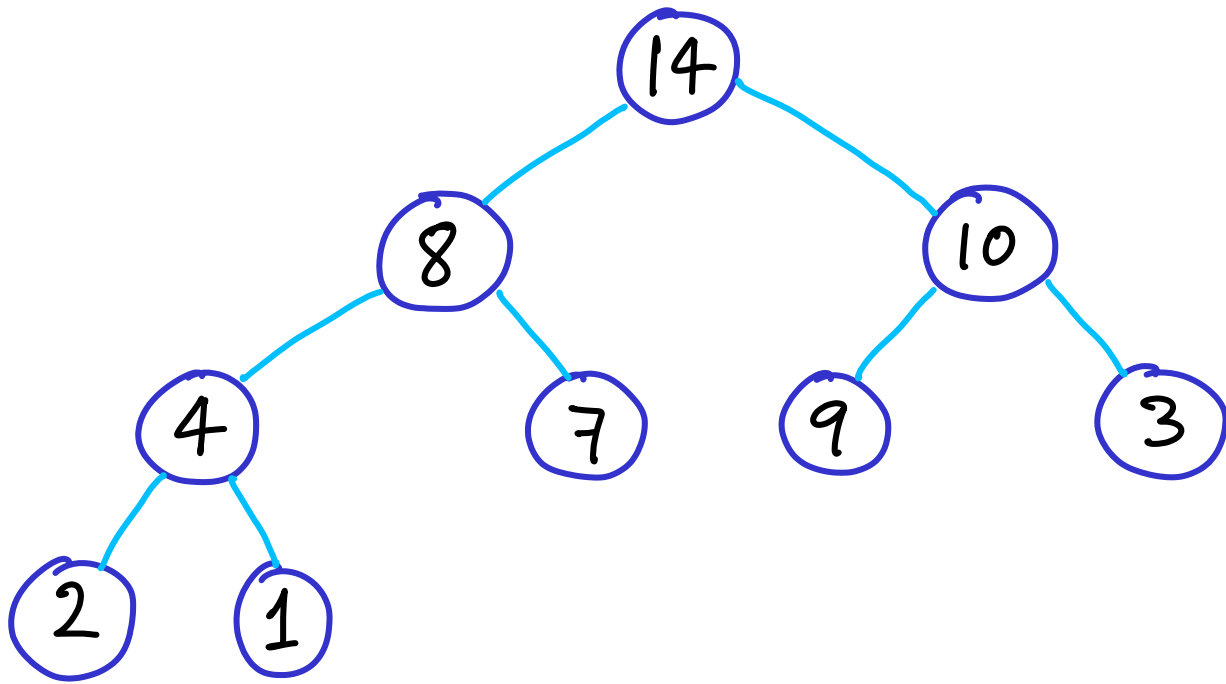




height of heap?

16

- extract MAX
  - place rightmost leaf at top  
(strange move... it's small!)
  - swap top w/ largest child  
(locally restore parent > child)
  - repeat downward while needed
- time?

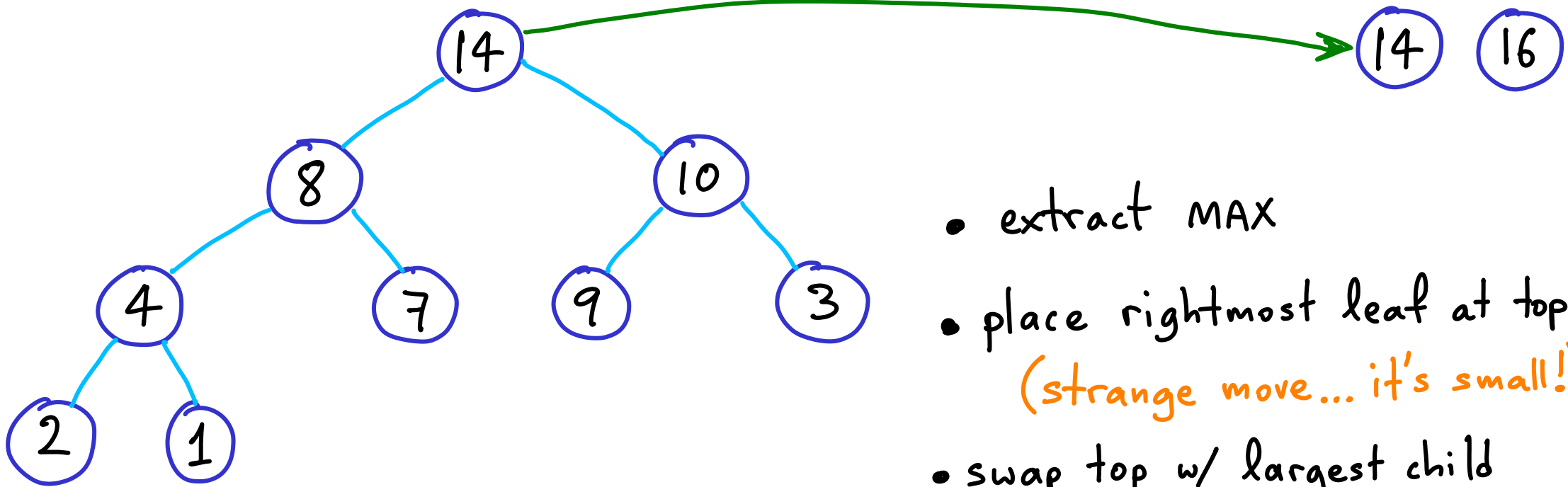


16

- extract MAX
- place rightmost leaf at top  
(strange move... it's small!)
- swap top w/ largest child  
(locally restore parent > child)
- repeat downward while needed

height of heap?  $\rightarrow \Theta(\log n)$

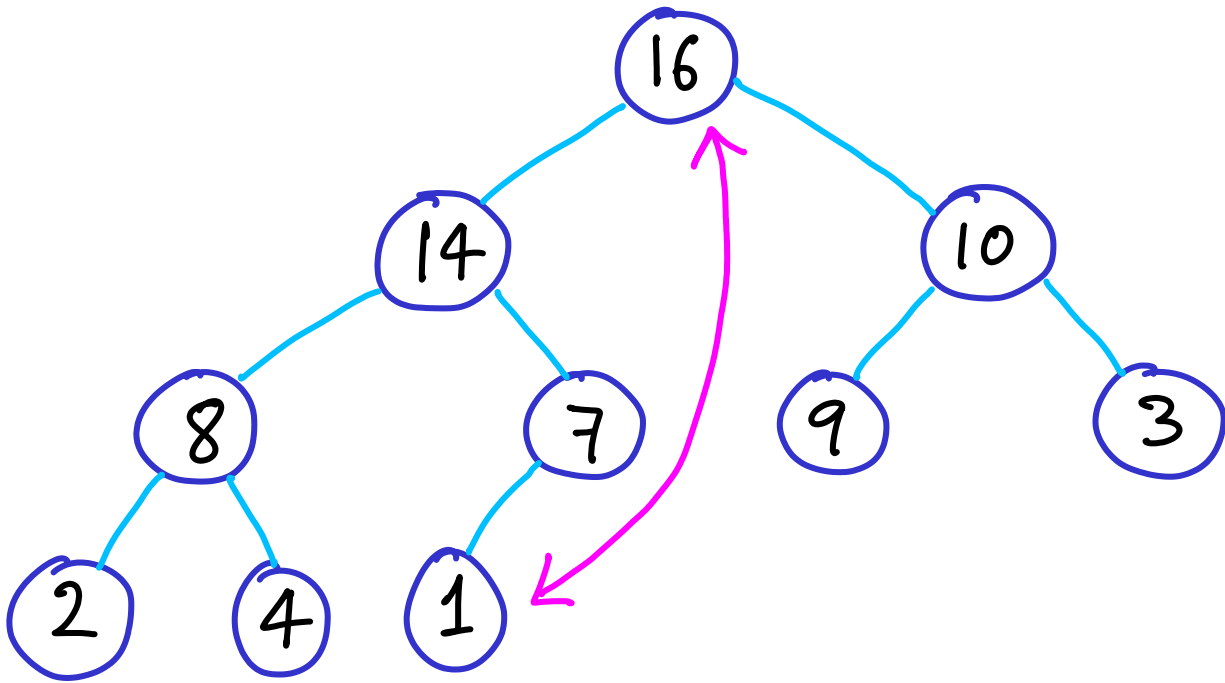
time?



height of heap?  $\rightarrow \Theta(\log n)$

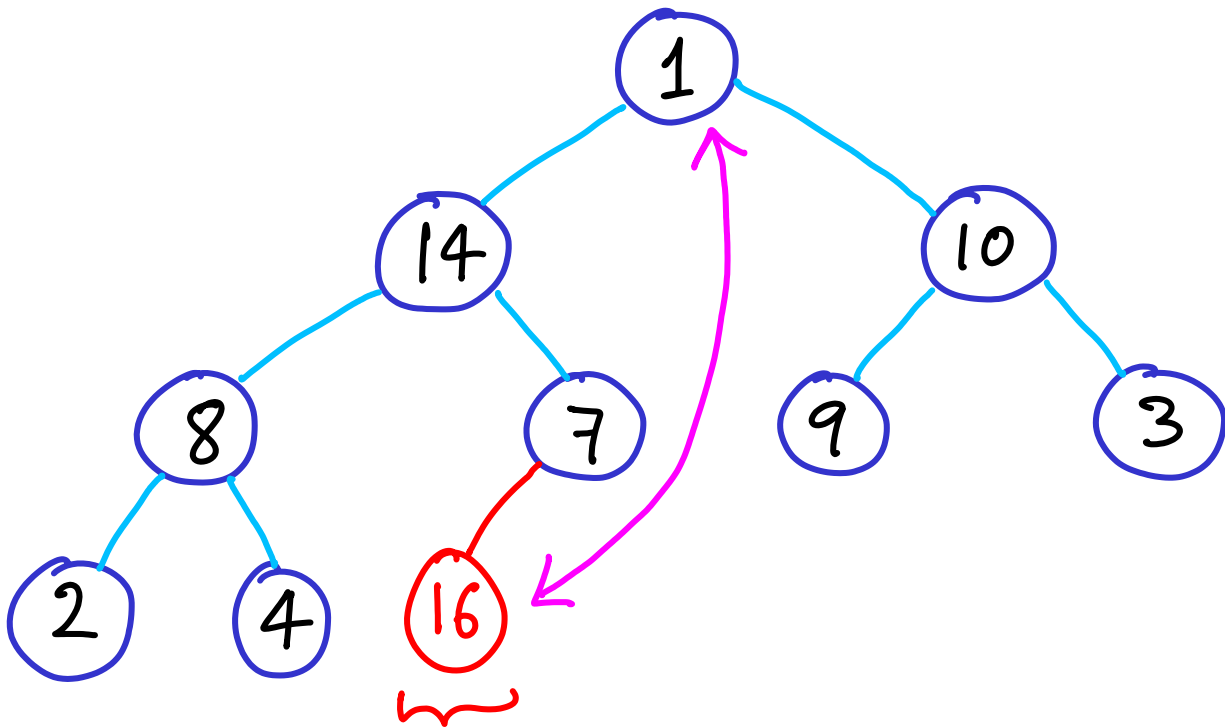
Ready to extract new MAX

time?



To work in-place  
when extracting MAX  
we can swap it w/ leaf.

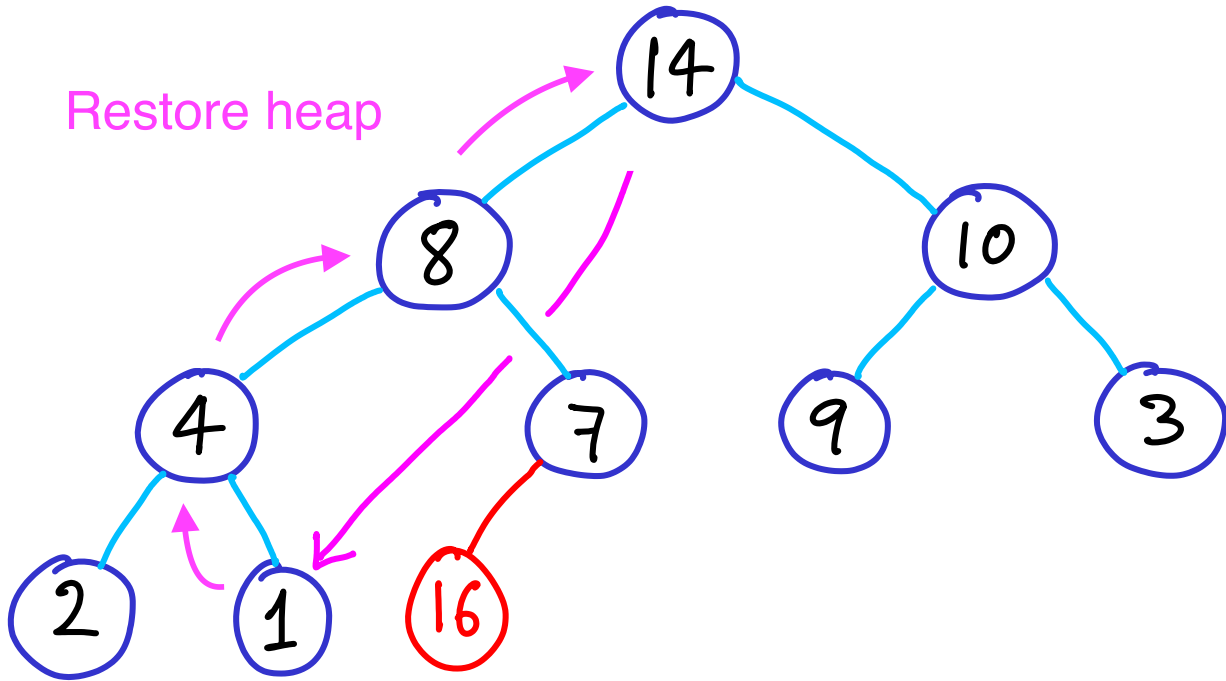
One more time:  
More space-efficient



To work in-place  
when extracting MAX  
we can swap it w/ leaf.

instead of deleting this node, just ignore it.

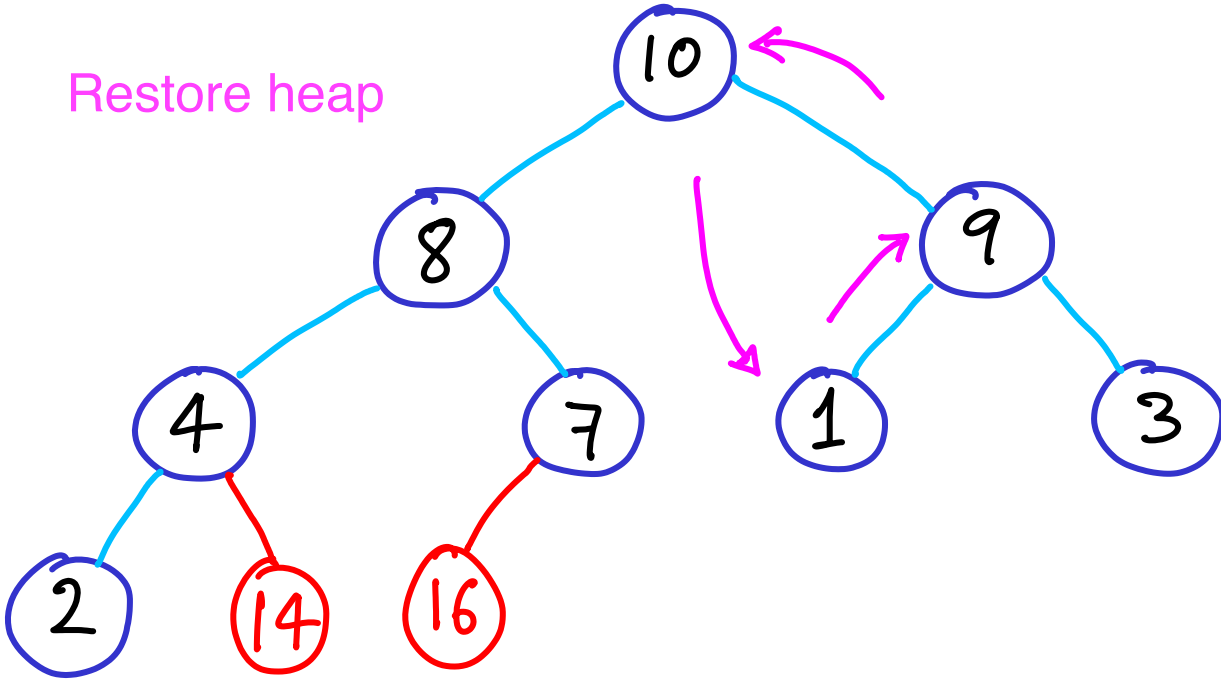
Notice MAX is stored at the max index of our array.



To work in-place  
when extracting MAX  
we can swap it w/ leaf.

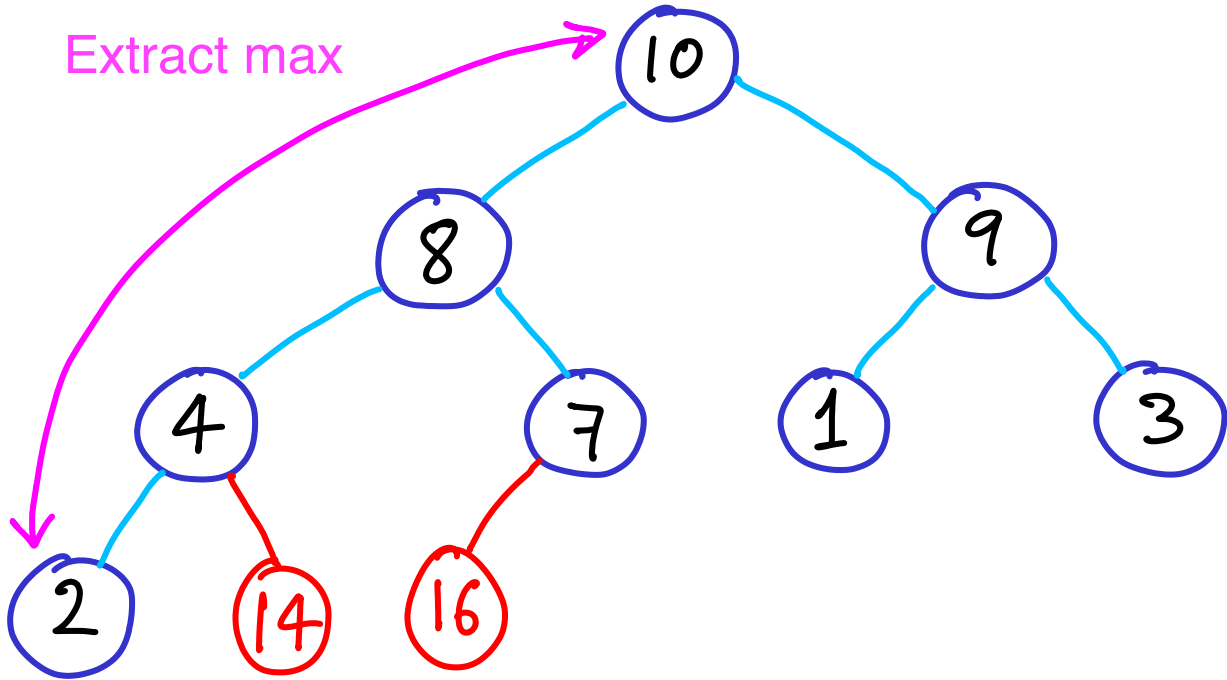


Restore heap

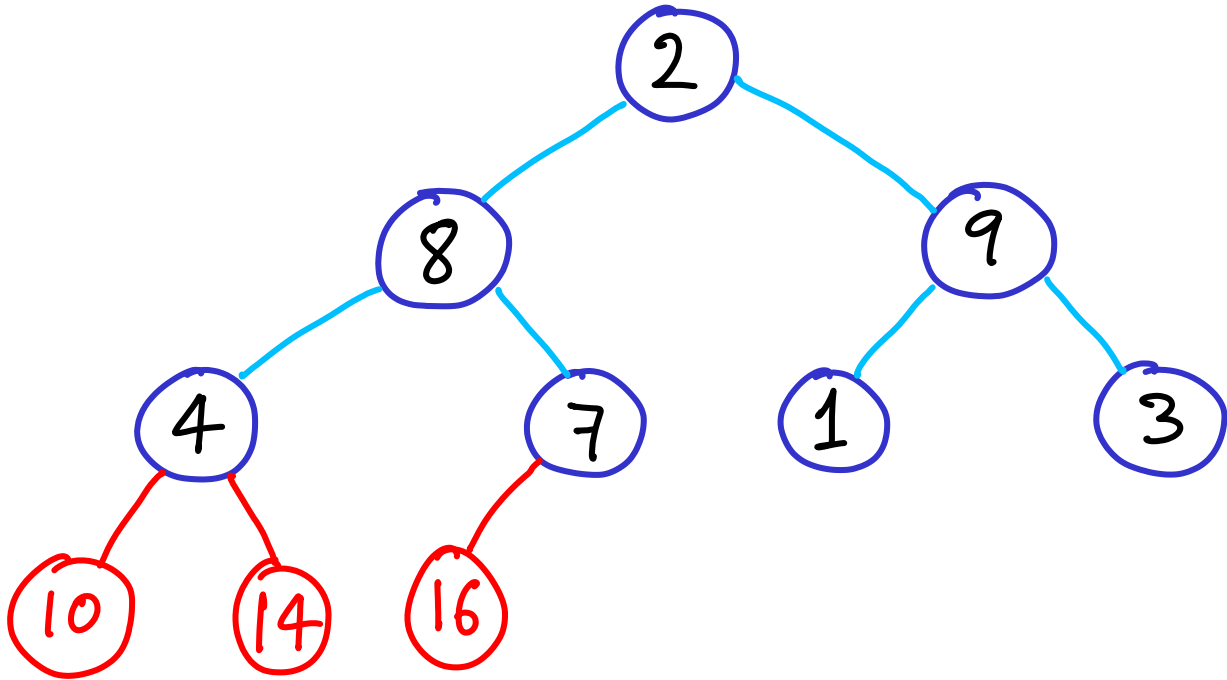


To work in-place  
when extracting MAX  
we can swap it w/ leaf.



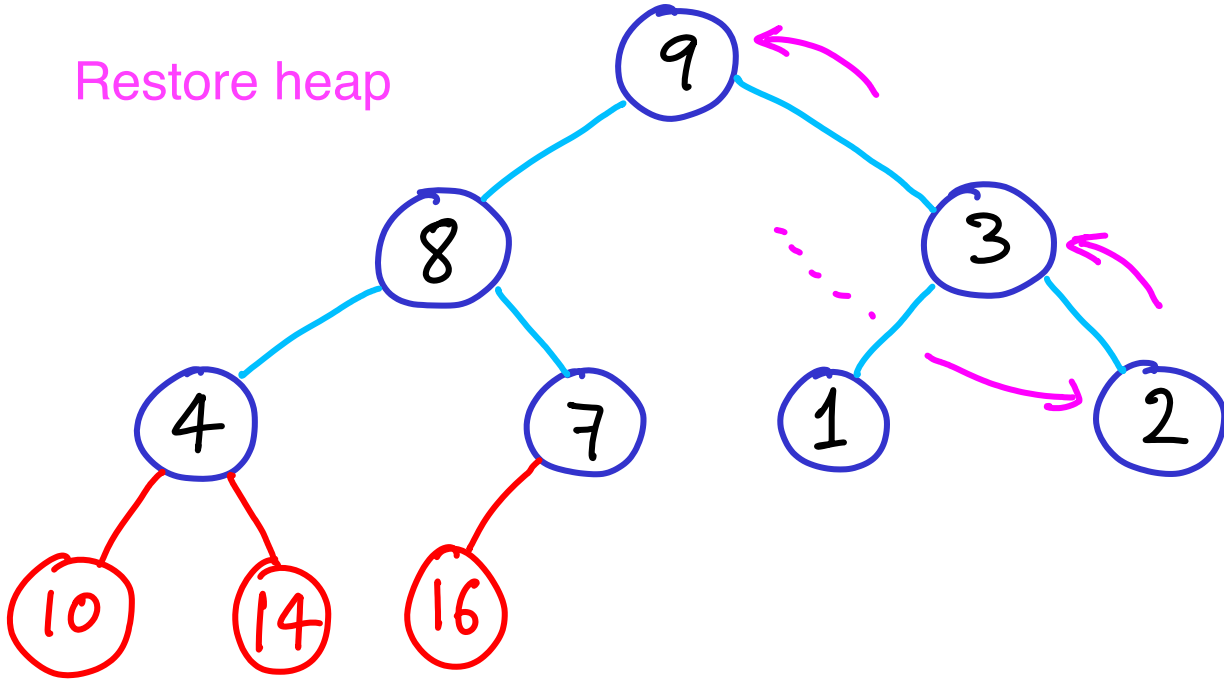


To work in-place  
when extracting MAX  
we can swap it w/ leaf.

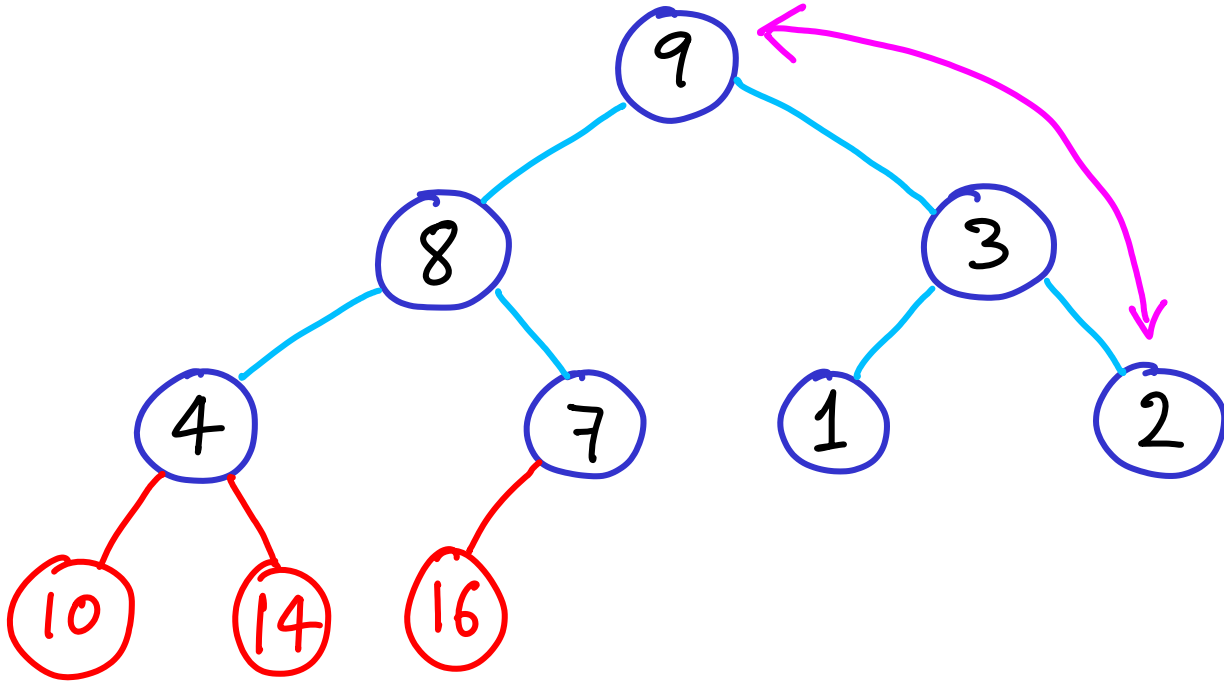


To work in-place  
when extracting MAX  
we can swap it w/ leaf.

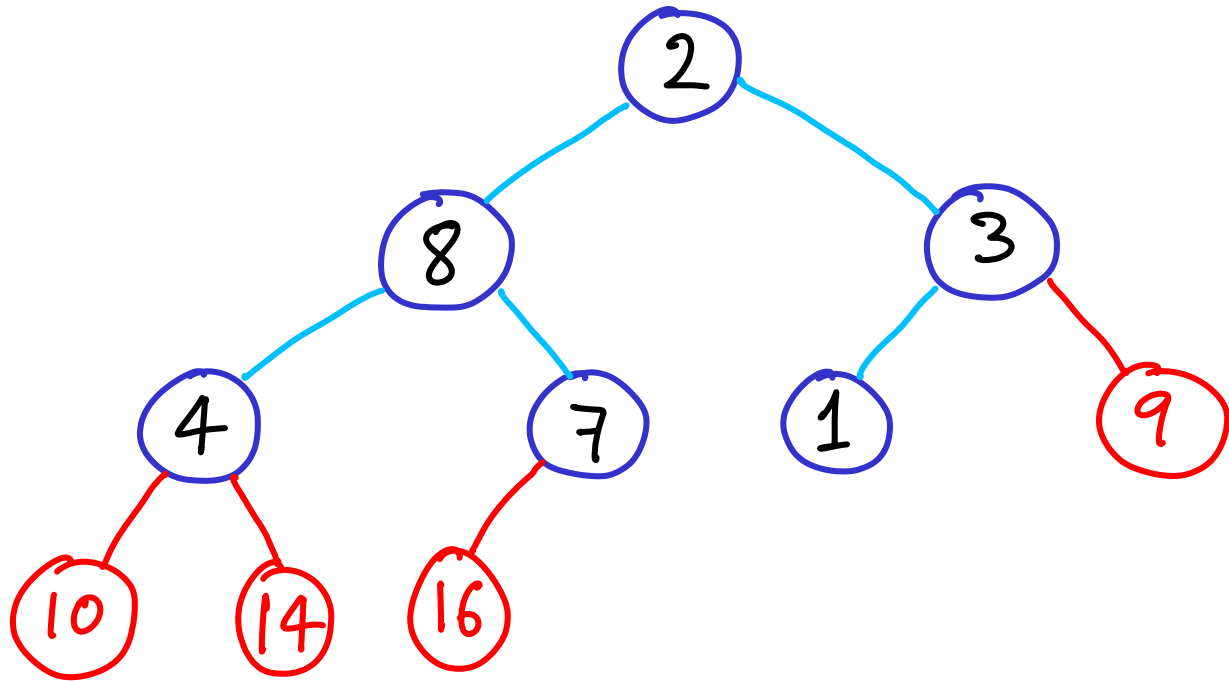
Restore heap



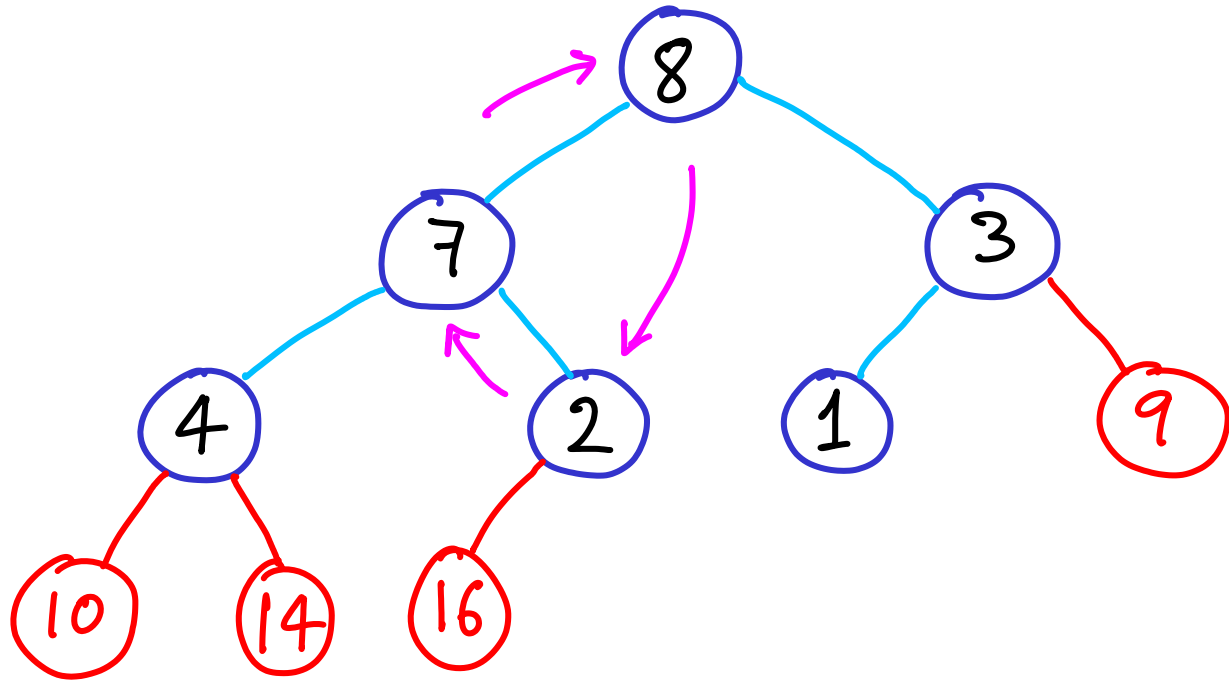
To work in-place  
when extracting MAX  
we can swap it w/ leaf.



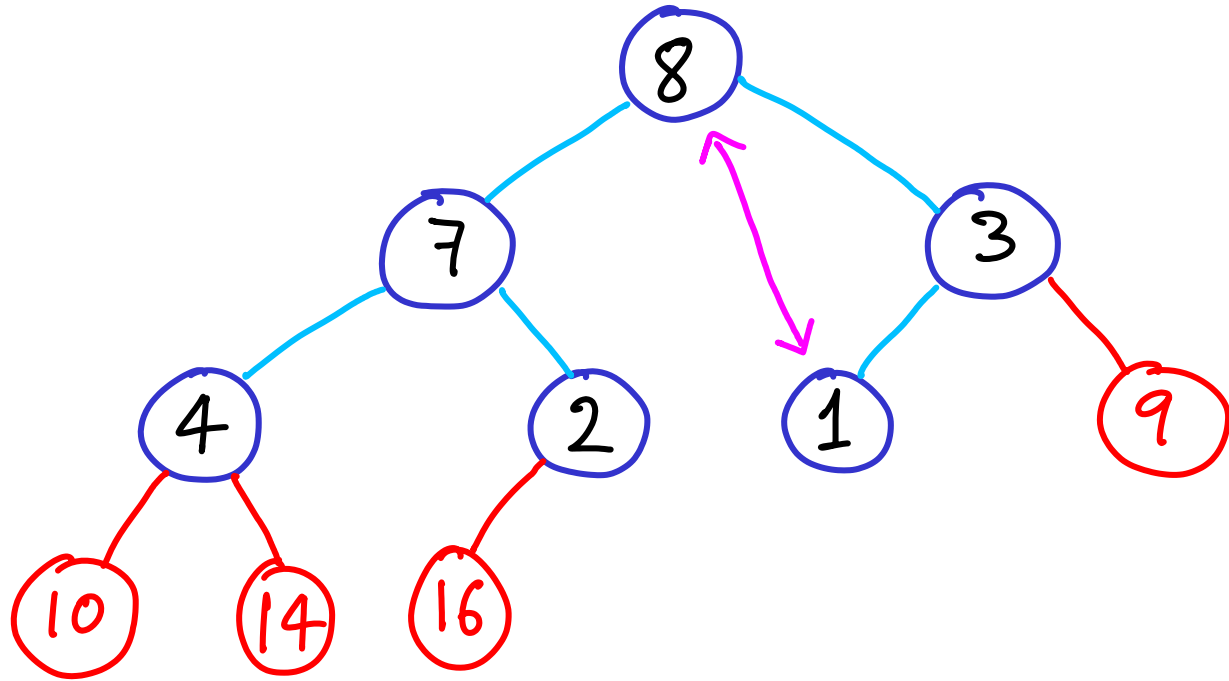
To work in-place  
when extracting MAX  
we can swap it w/ leaf.



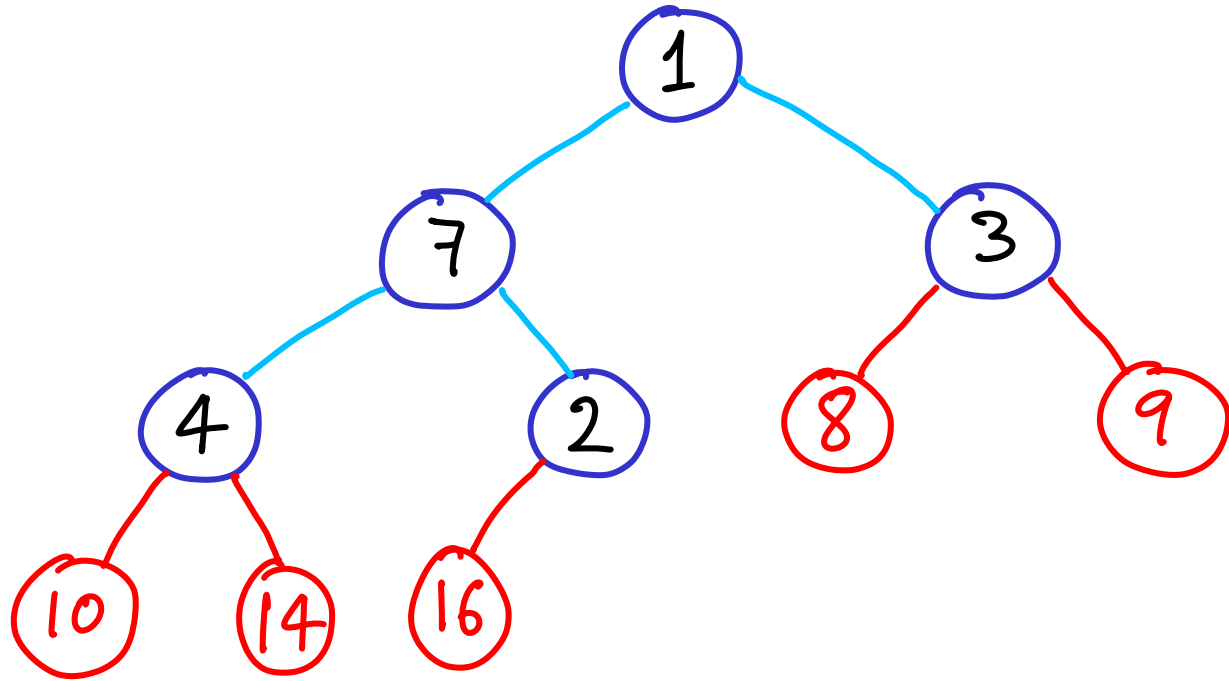
To work in-place  
when extracting MAX  
we can swap it w/ leaf.



To work in-place  
when extracting MAX  
we can swap it w/ leaf.

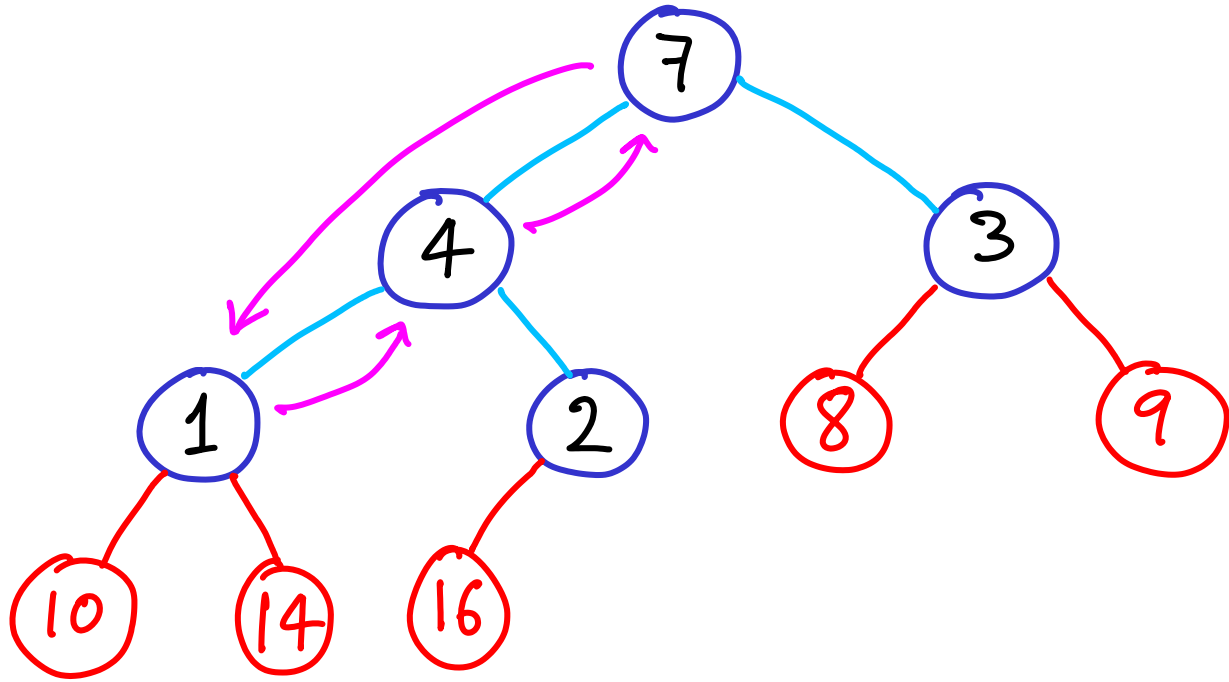


To work in-place  
when extracting MAX  
we can swap it w/ leaf.

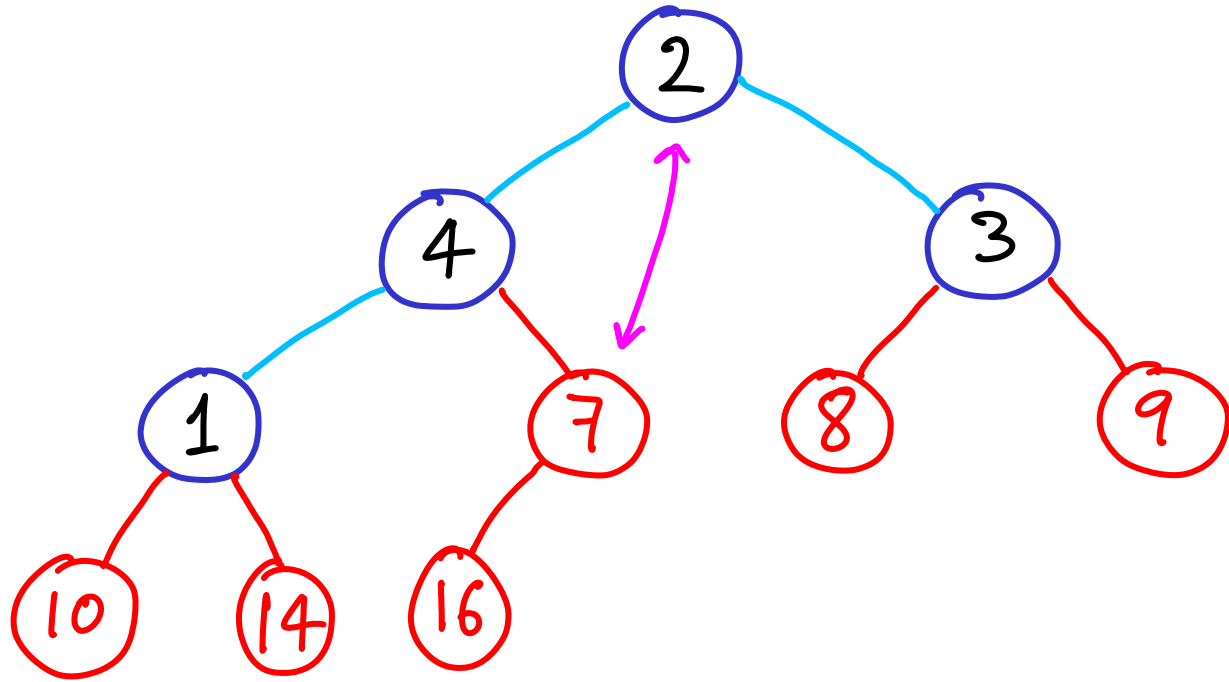


To work in-place  
when extracting MAX  
we can swap it w/ leaf.

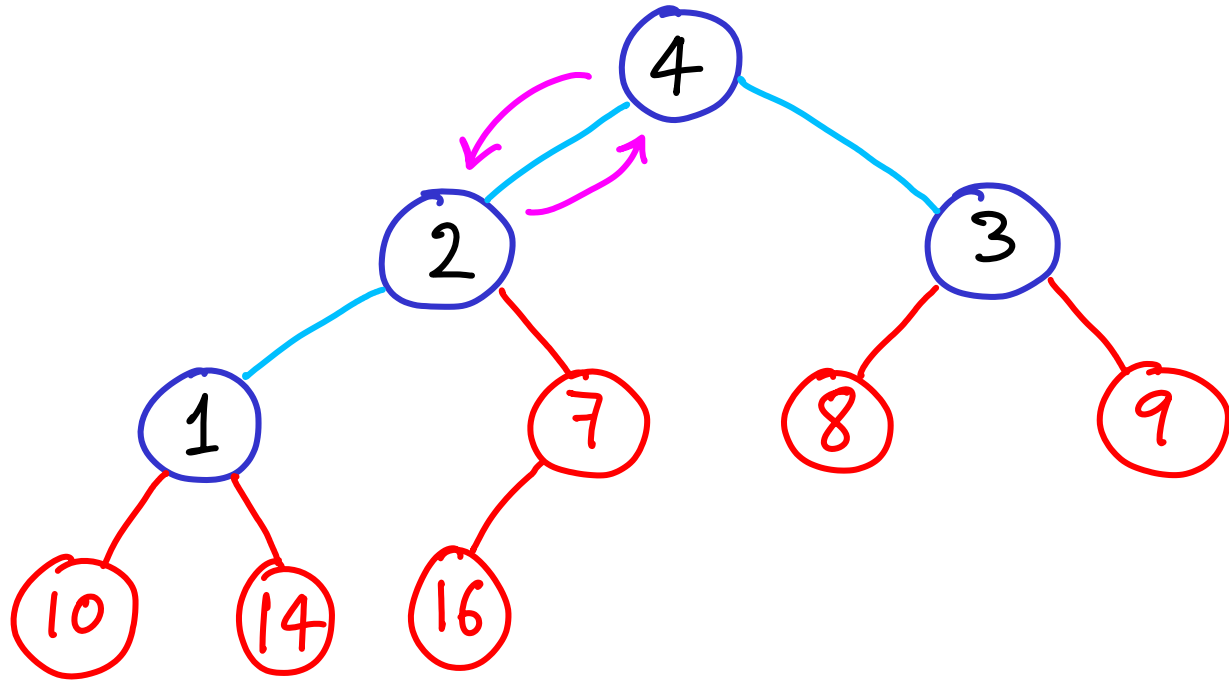




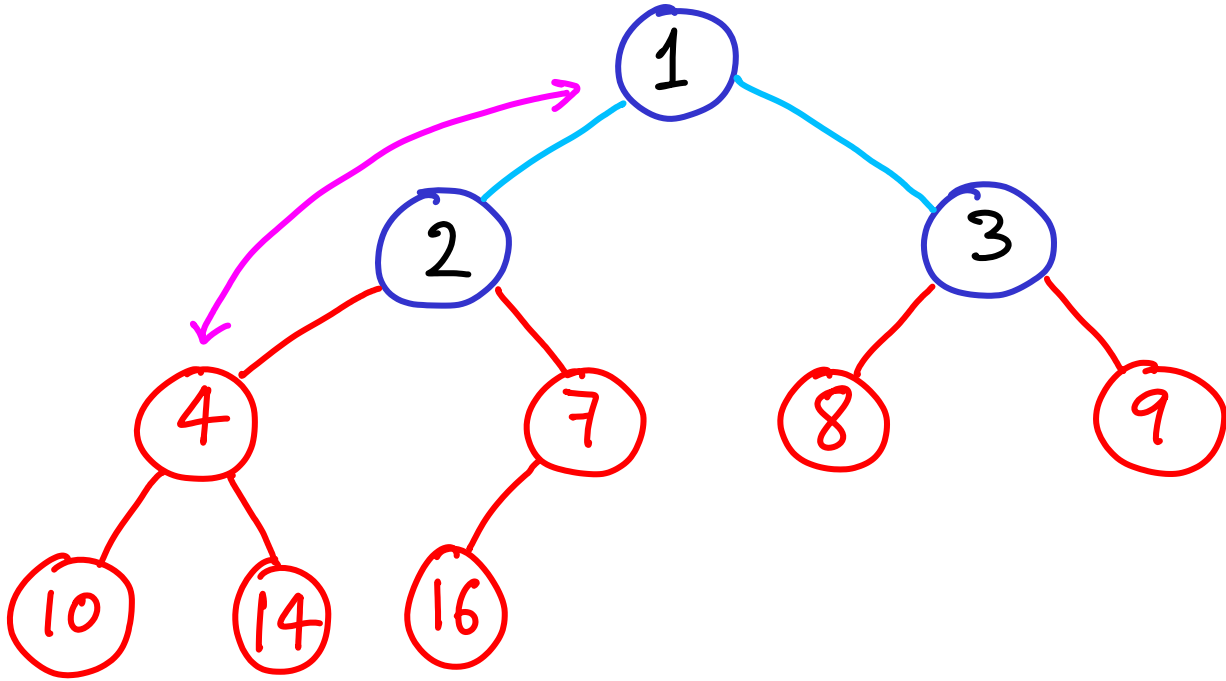
To work in-place  
when extracting MAX  
we can swap it w/ leaf.



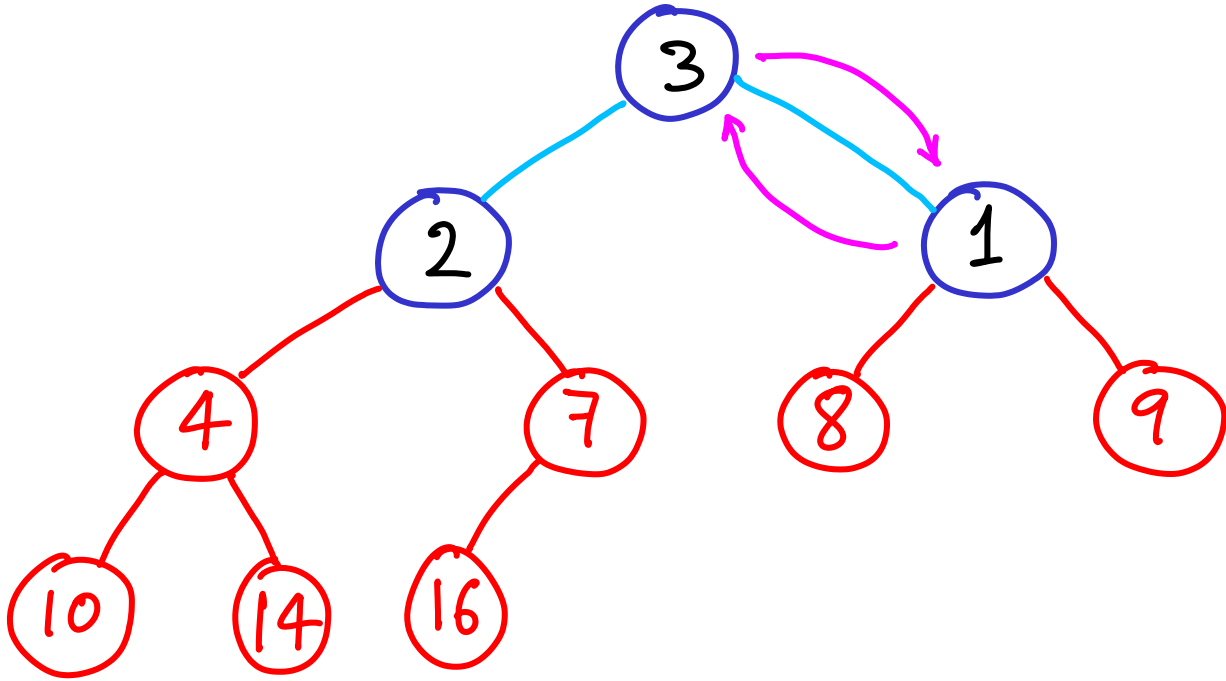
To work in-place  
when extracting MAX  
we can swap it w/ leaf.



To work in-place  
when extracting MAX  
we can swap it w/ leaf.

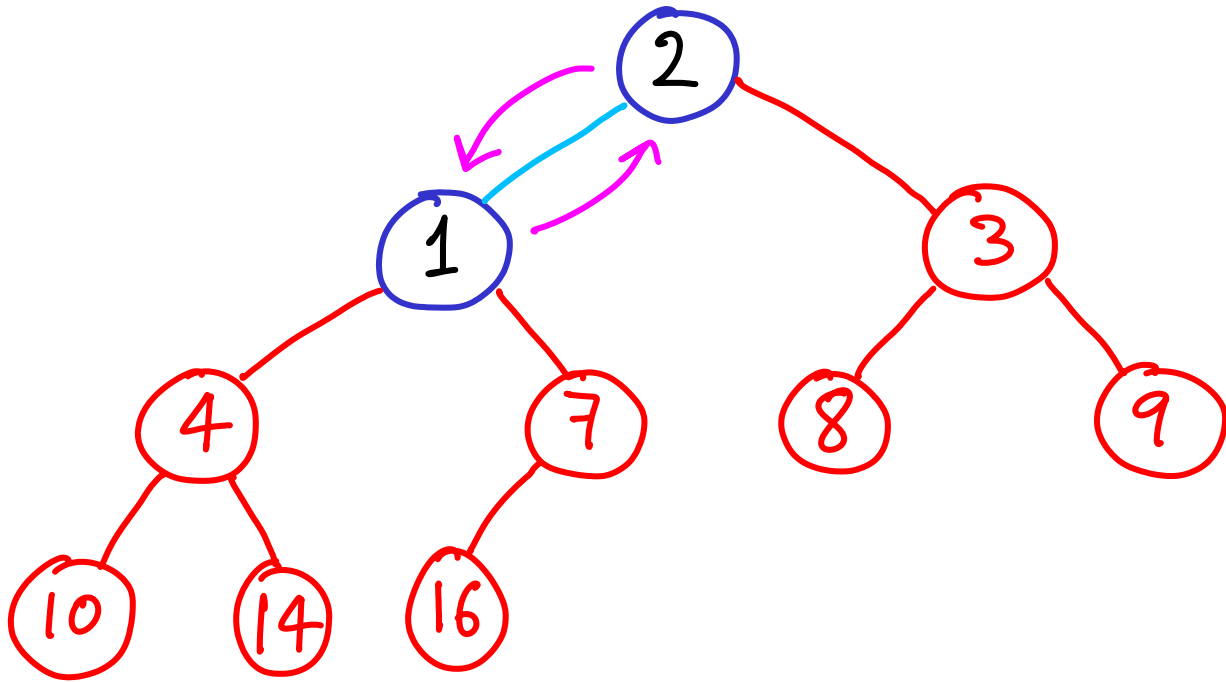


To work in-place  
when extracting MAX  
we can swap it w/ leaf.

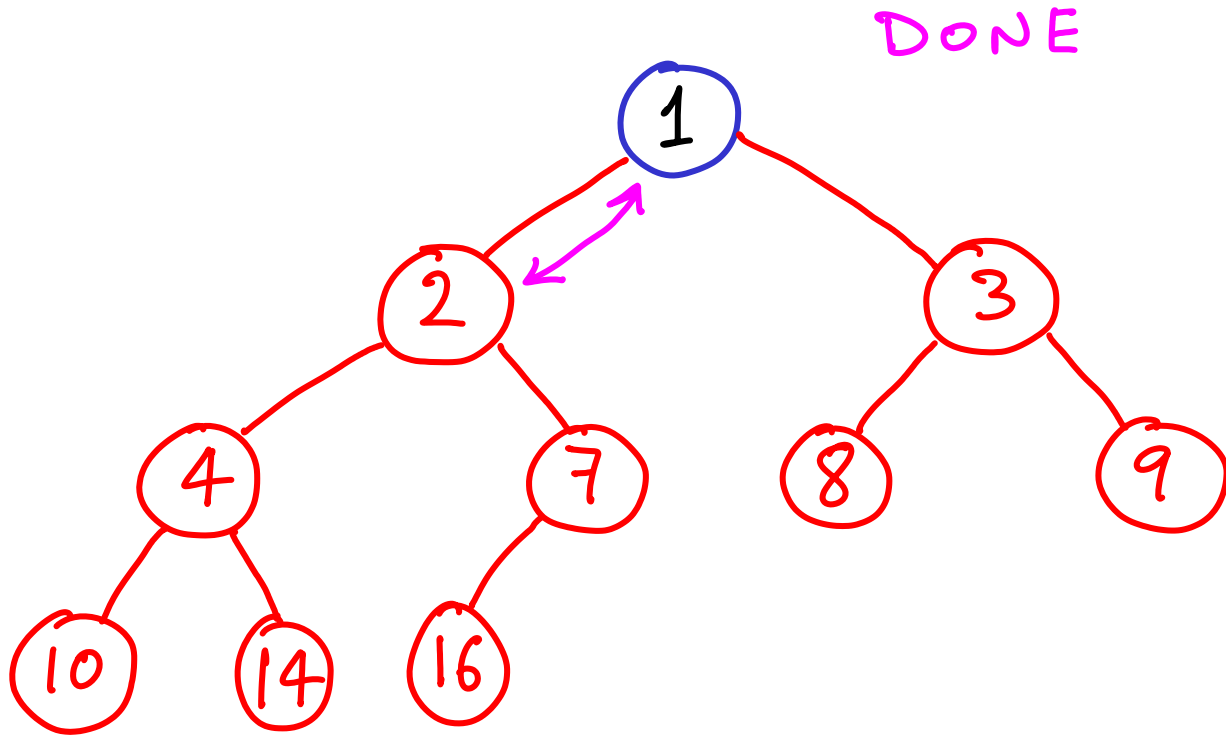


To work in-place  
when extracting MAX  
we can swap it w/ leaf.





To work in-place  
when extracting MAX  
we can swap it w/ leaf.



To work in-place  
when extracting MAX  
we can swap it w/ leaf.



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

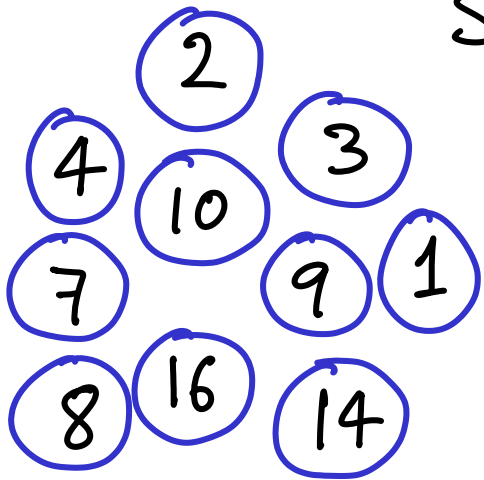
↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

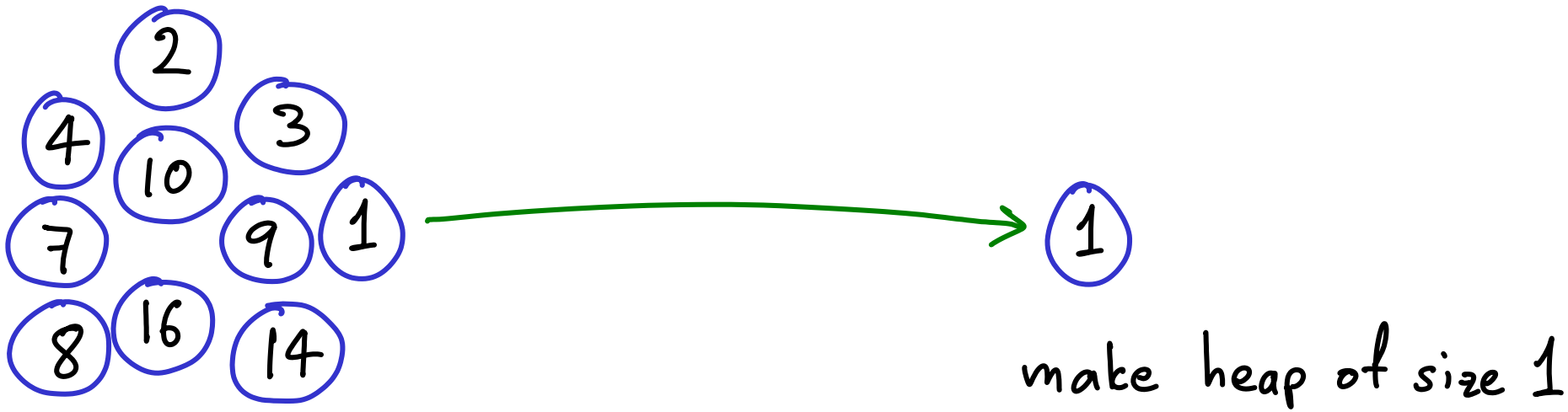


Start with unsorted  
elements

So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

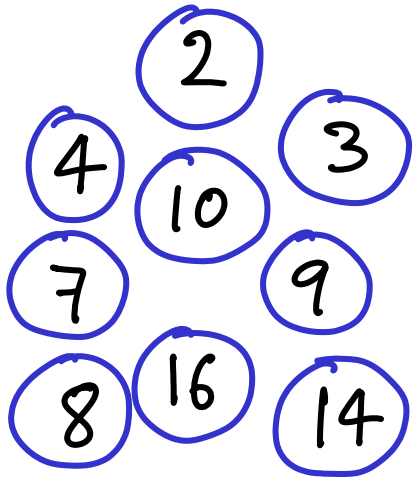
But how did we have a heap in the first place?



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

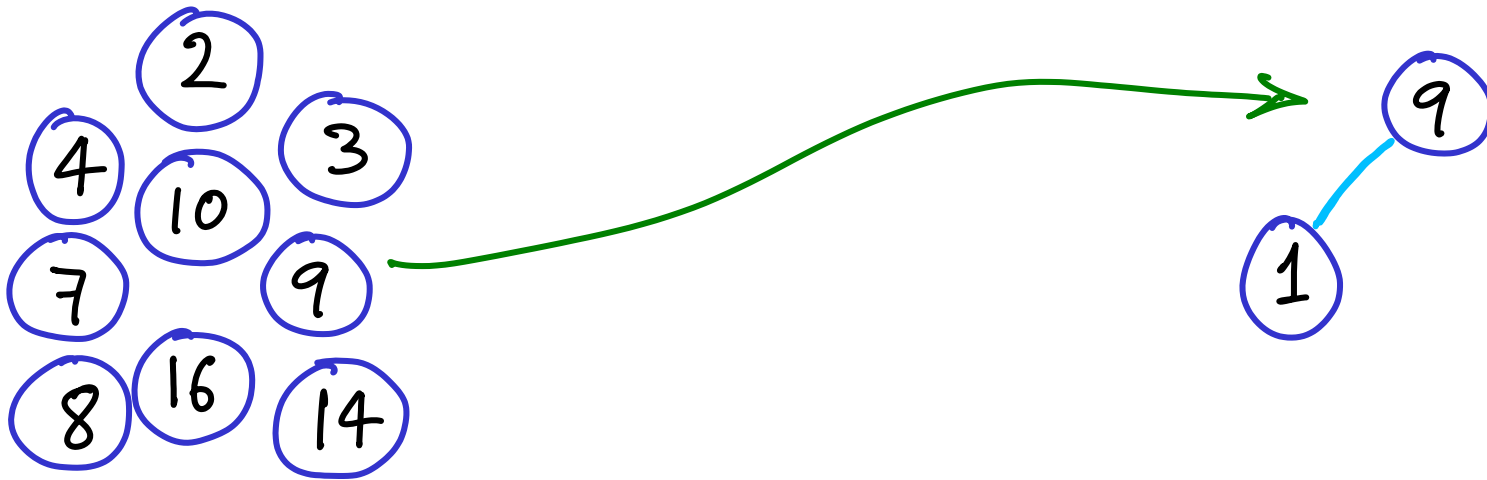
But how did we have a heap in the first place?



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

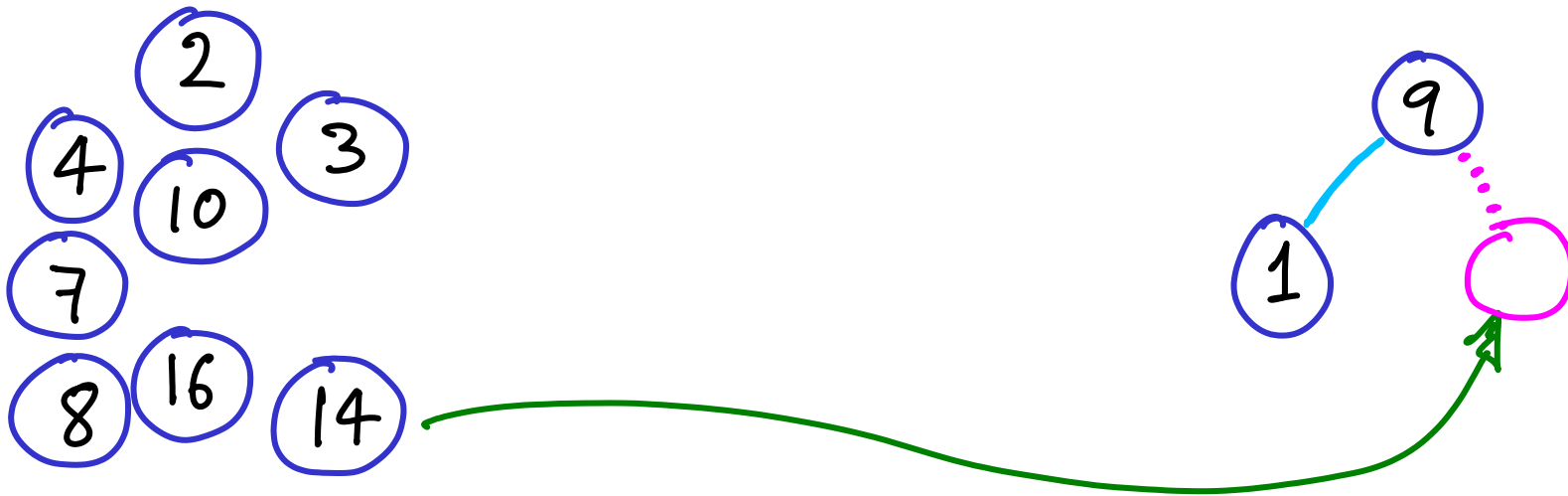


trivially get heap  
of size 2,  
possibly w/ a swap

So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

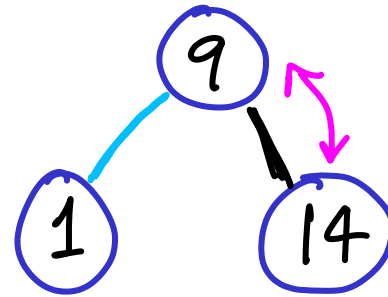
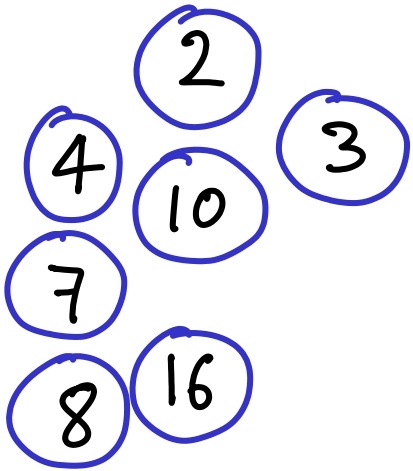


generally, insert  
new element as  
rightmost leaf  
in lowest level

So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

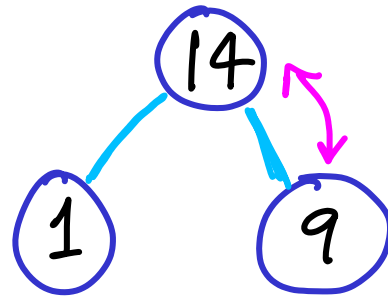
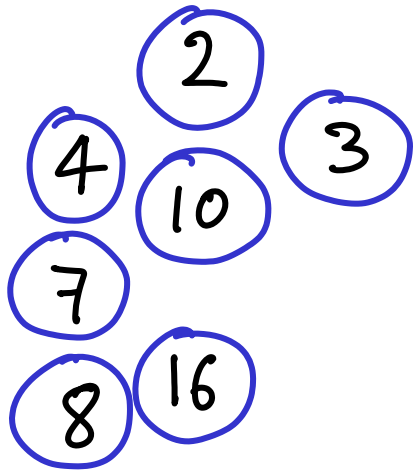


then repeat  
swapping  
while required

So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?



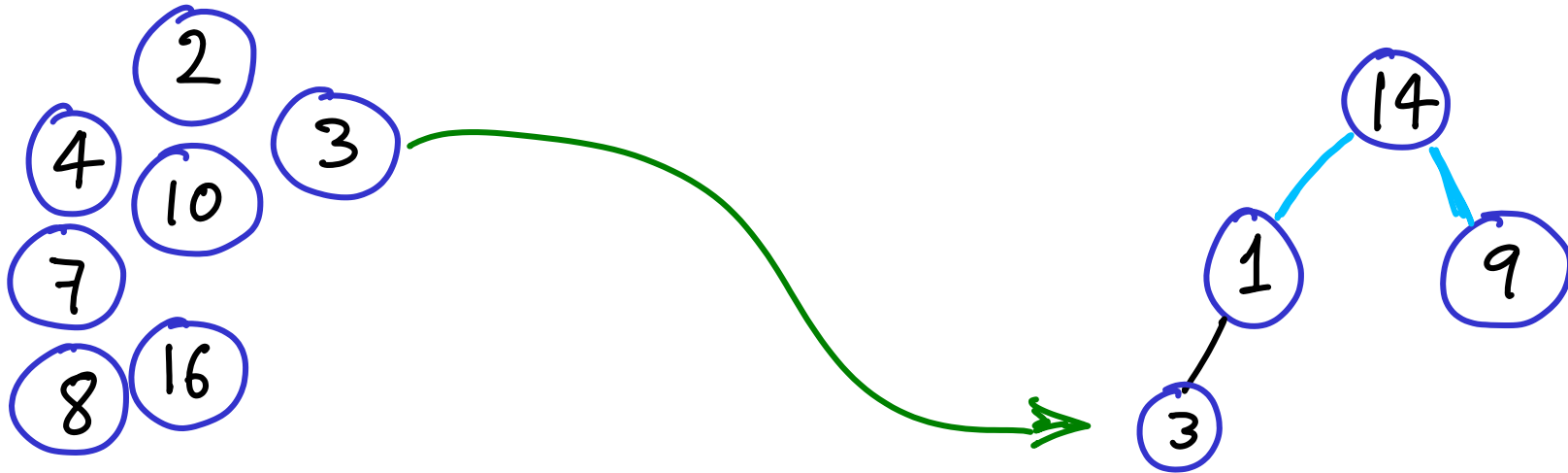
then repeat  
swapping  
while required



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

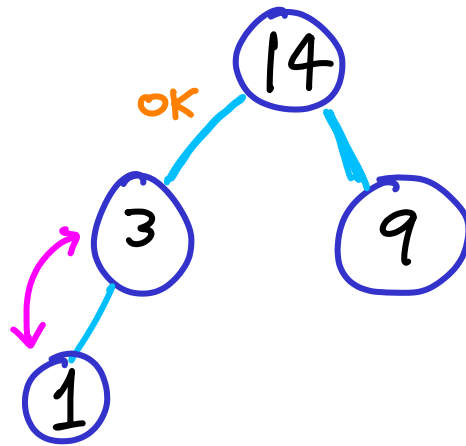
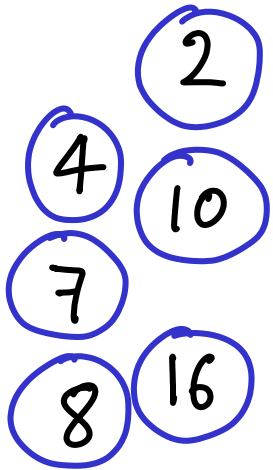
But how did we have a heap in the first place?



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

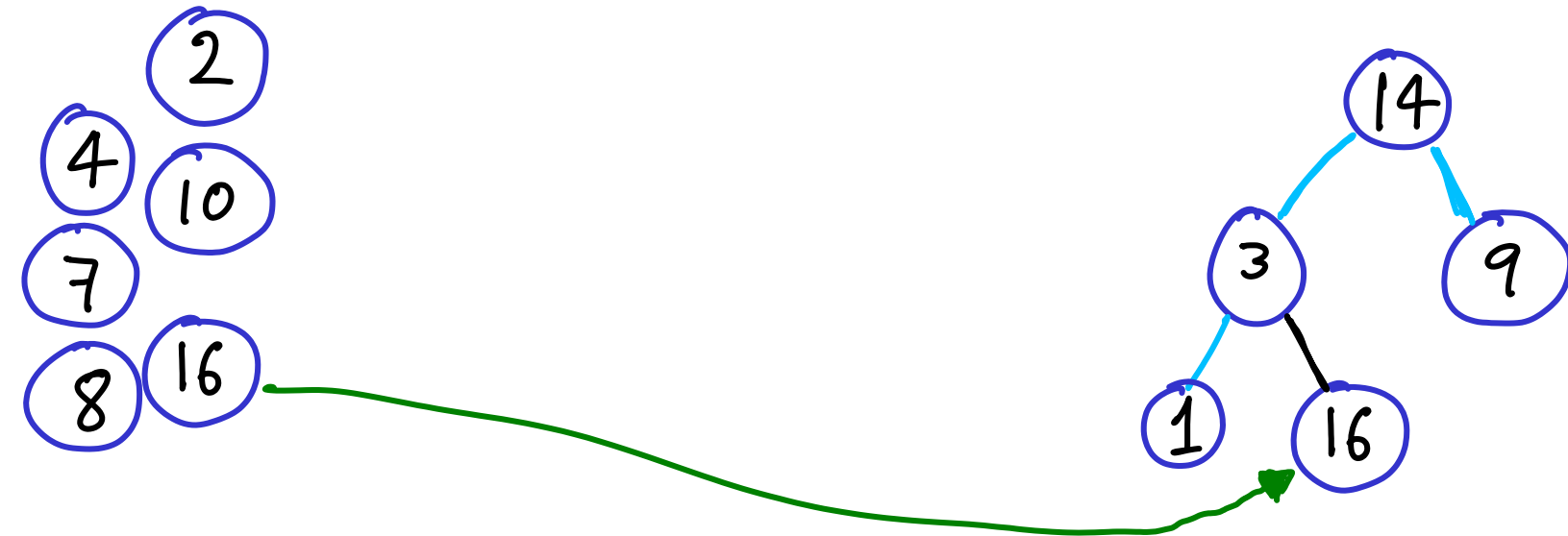
But how did we have a heap in the first place?



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

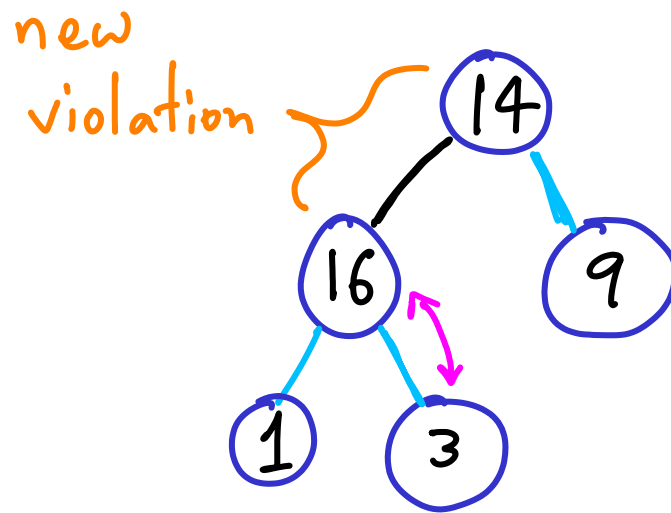
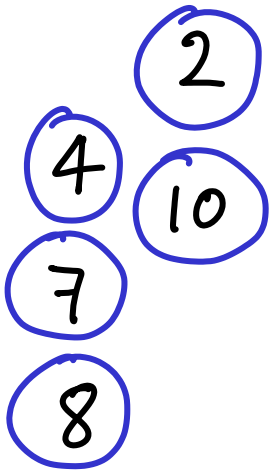
But how did we have a heap in the first place?



So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

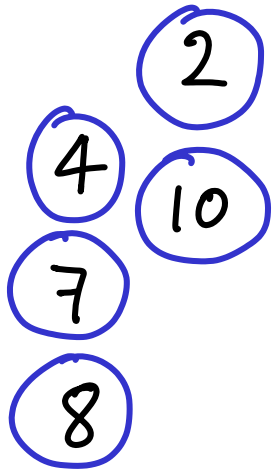
But how did we have a heap in the first place?



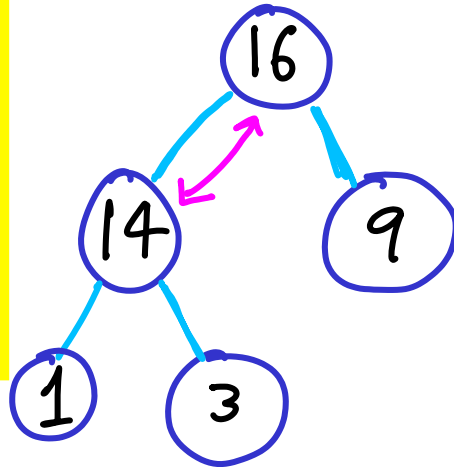
So we can extract MAX & maintain a heap, in  $O(\log n)$  time.

↳ do this  $n$  times  $\rightarrow O(n \log n)$

But how did we have a heap in the first place?

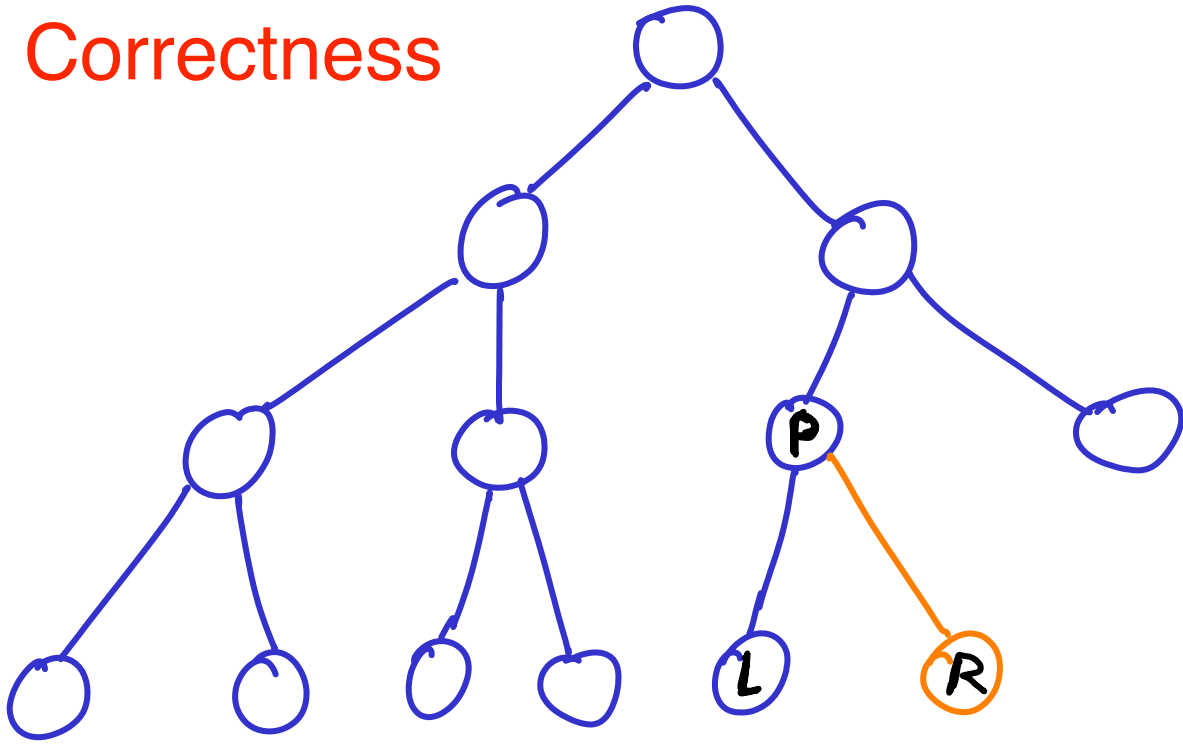


**"Forward" method**  
read input array left to right  
Also works "online":  
(streaming data)



etc

# Correctness

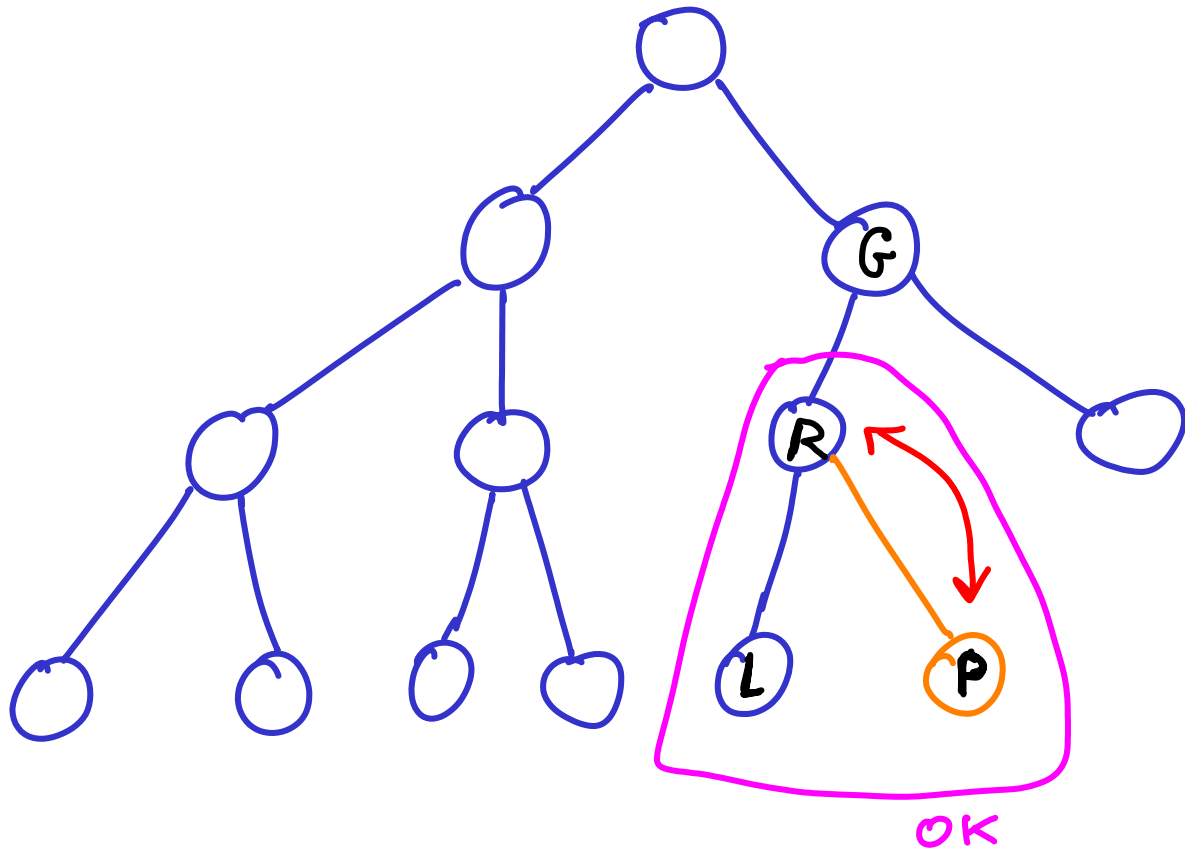


When inserting a new leaf

(wlog  $R$ )

there is a problem iff

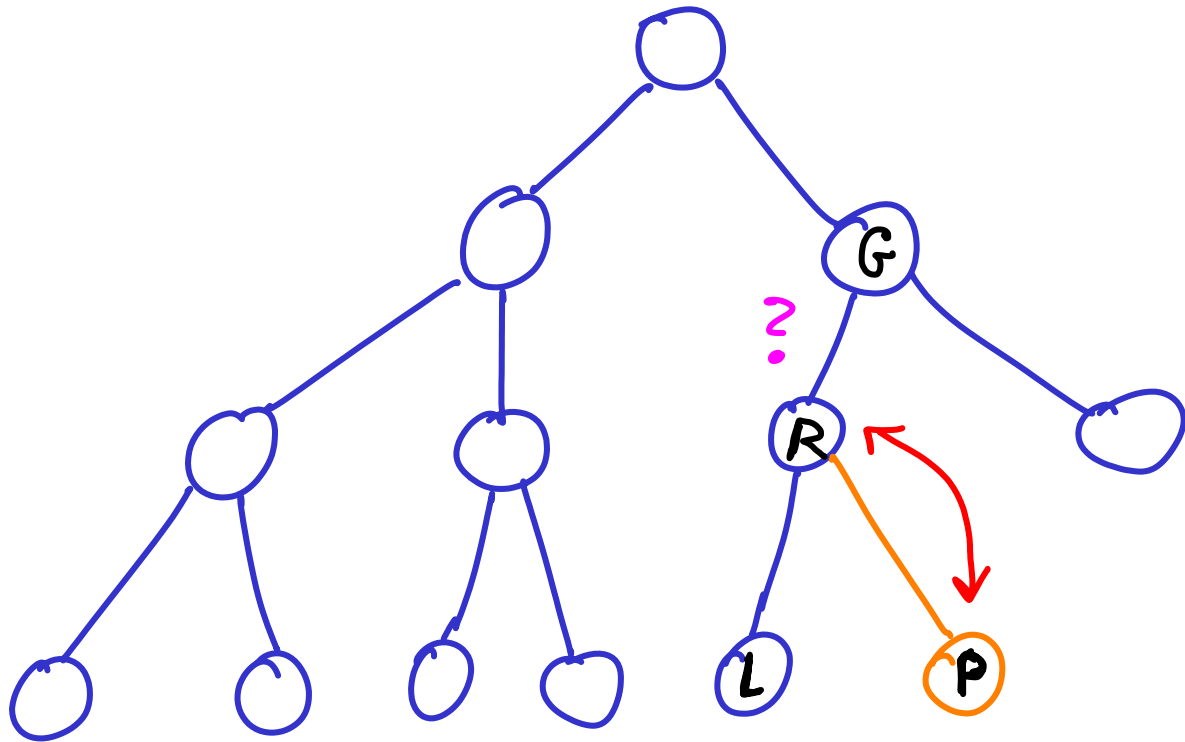
$$R > P$$



When inserting a new leaf  
(wlog  $R$ )

there is a problem iff  
 $R > P$

By swapping  $R \leftrightarrow P$   
we have a heap in  $\text{subtree}(R)$   
 $(R > P > L)$



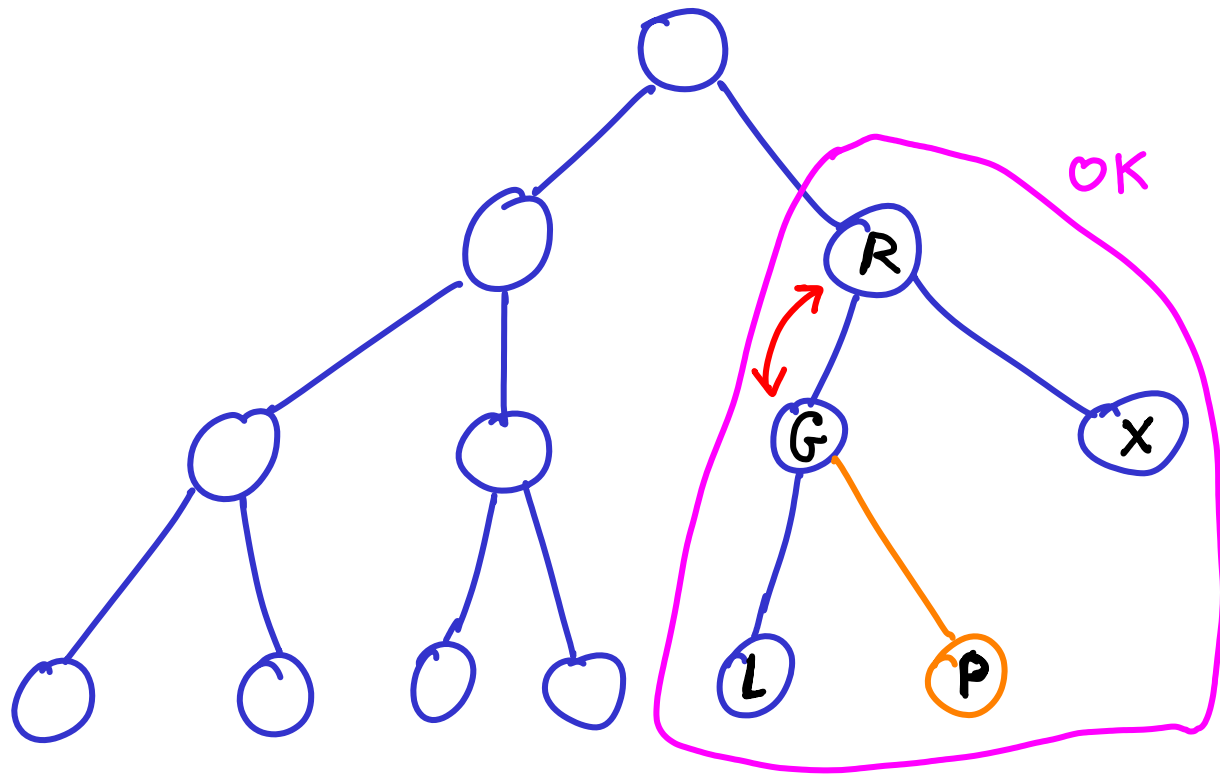
When inserting a new leaf  
(wlog R)

there is a problem iff  
 $R > P$

By swapping  $R \leftrightarrow P$   
we have a heap in  $\text{subtree}(R)$   
 $(R > P > L)$

...but we may have a new problem iff  $R > G$



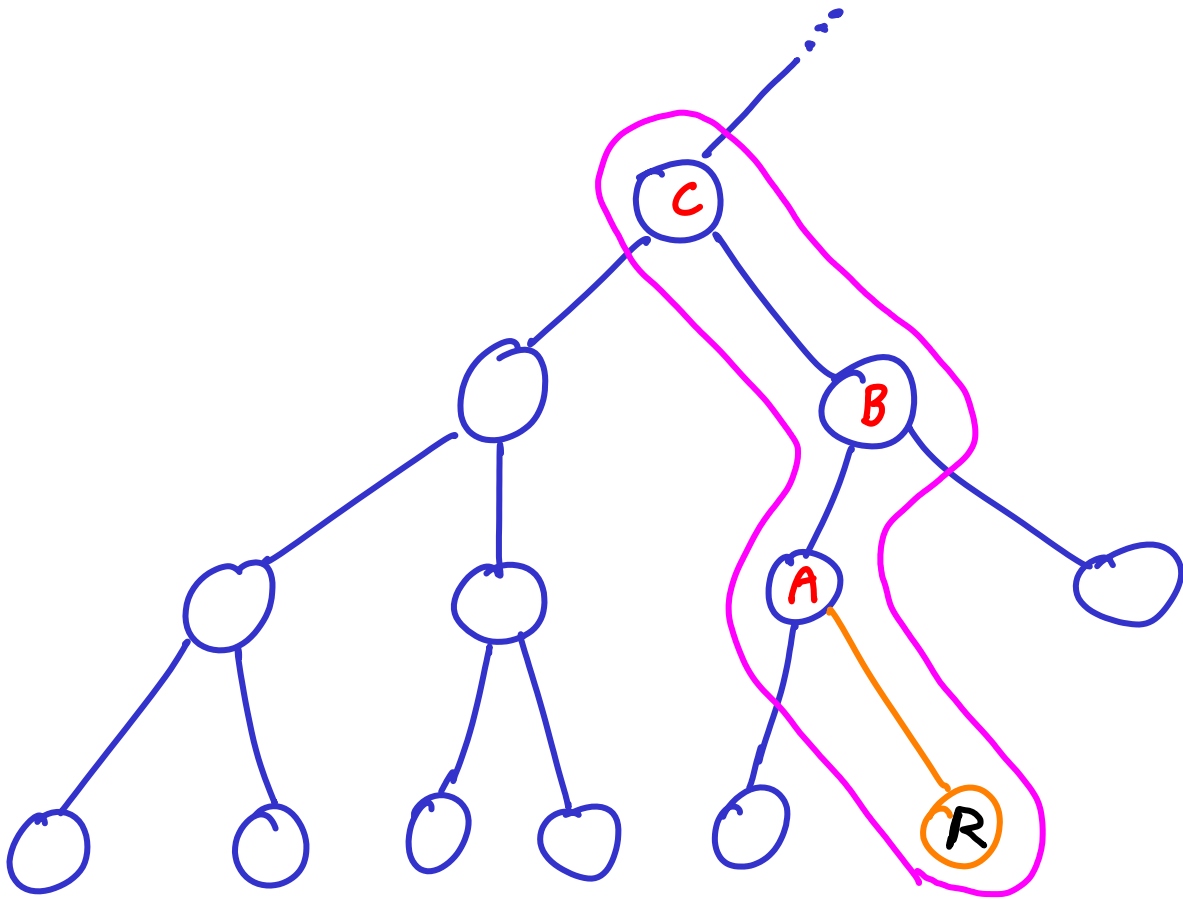


When inserting a new leaf  
(wlog  $R$ )

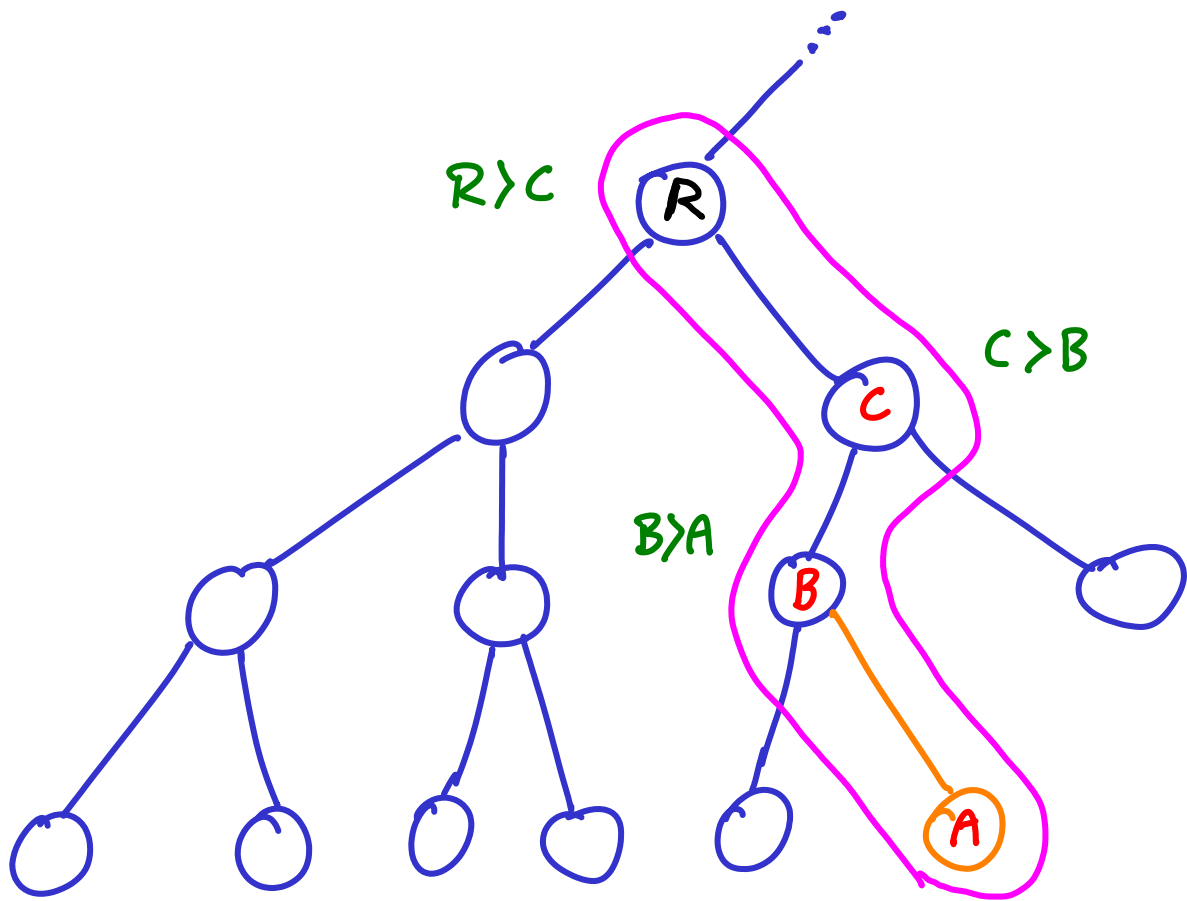
there is a problem iff  
 $R > P$

By swapping  $R \leftrightarrow P$   
we have a heap in  $\text{subtree}(R)$   
 $(R > P > L)$

...but we may have a new problem iff  $R > G$   
then  $R \leftrightarrow G$ .  $G > P > L$  : OK &  $R > G > X$  : OK

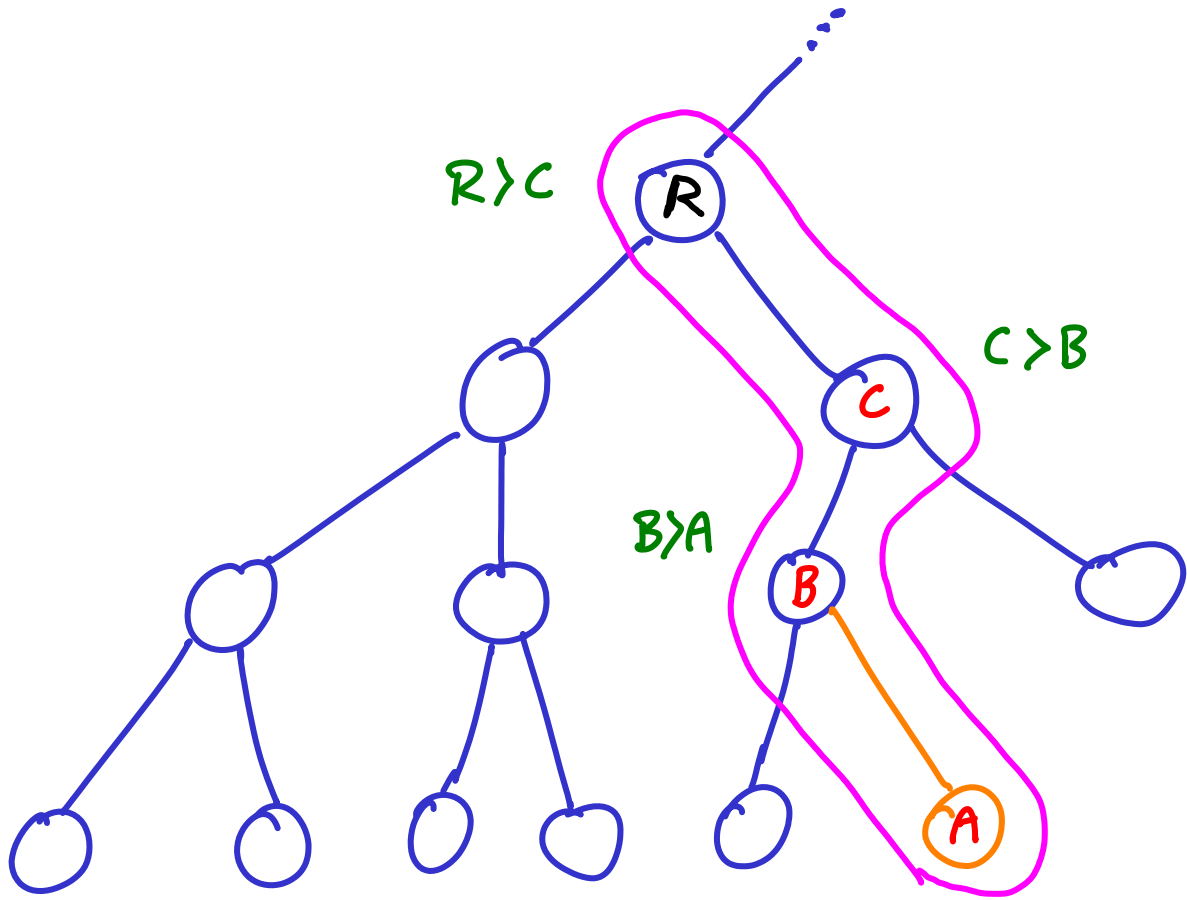


R will move up some path until smaller than node above.



R will move up some path until smaller than node above.

That path will "shift down" so every subtree has a larger root.

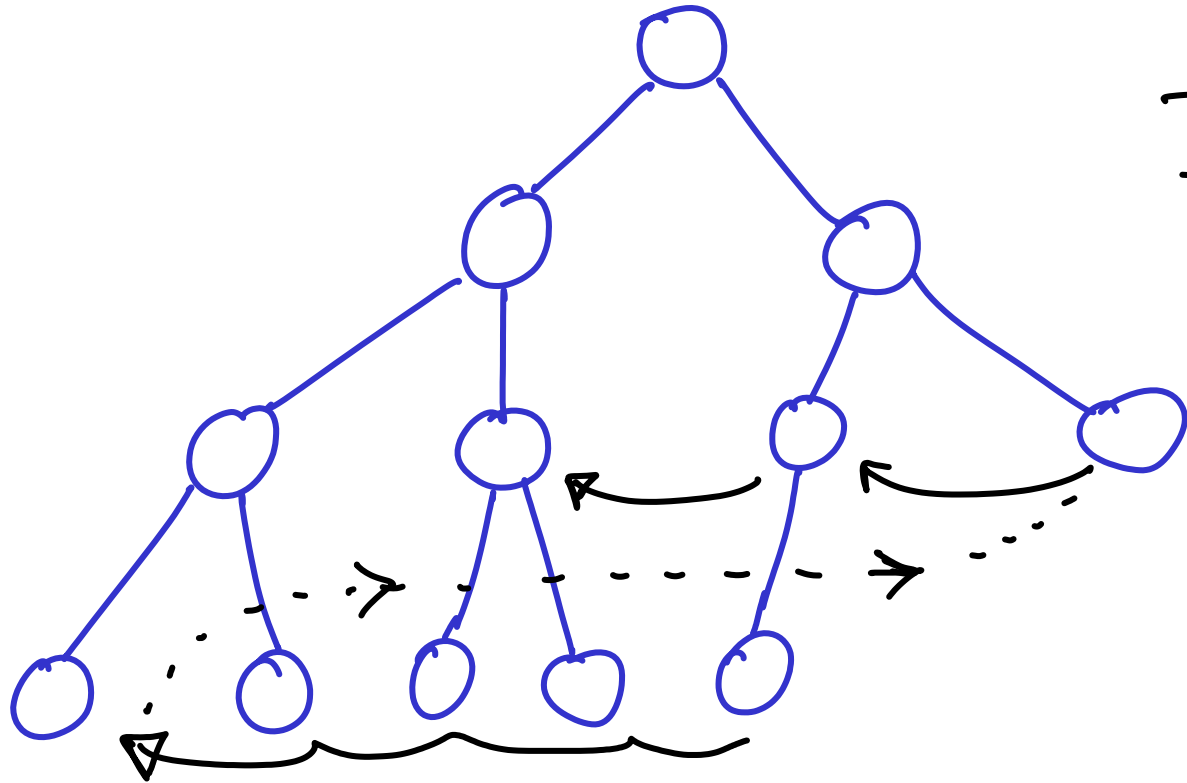


R will move up some path until smaller than node above.

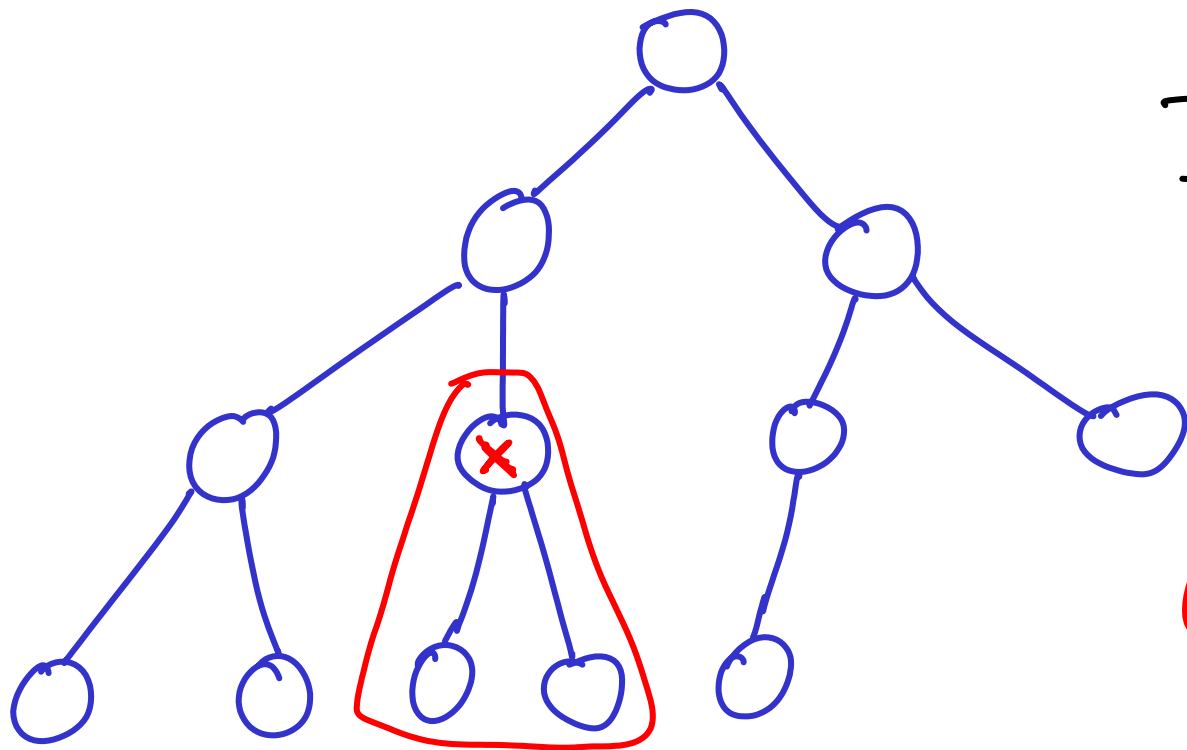
That path will "shift down" so every subtree has a larger root.

$O(n \log n)$  time  
 per node  
 and in-place : iterate on array.  
 Only swaps used.



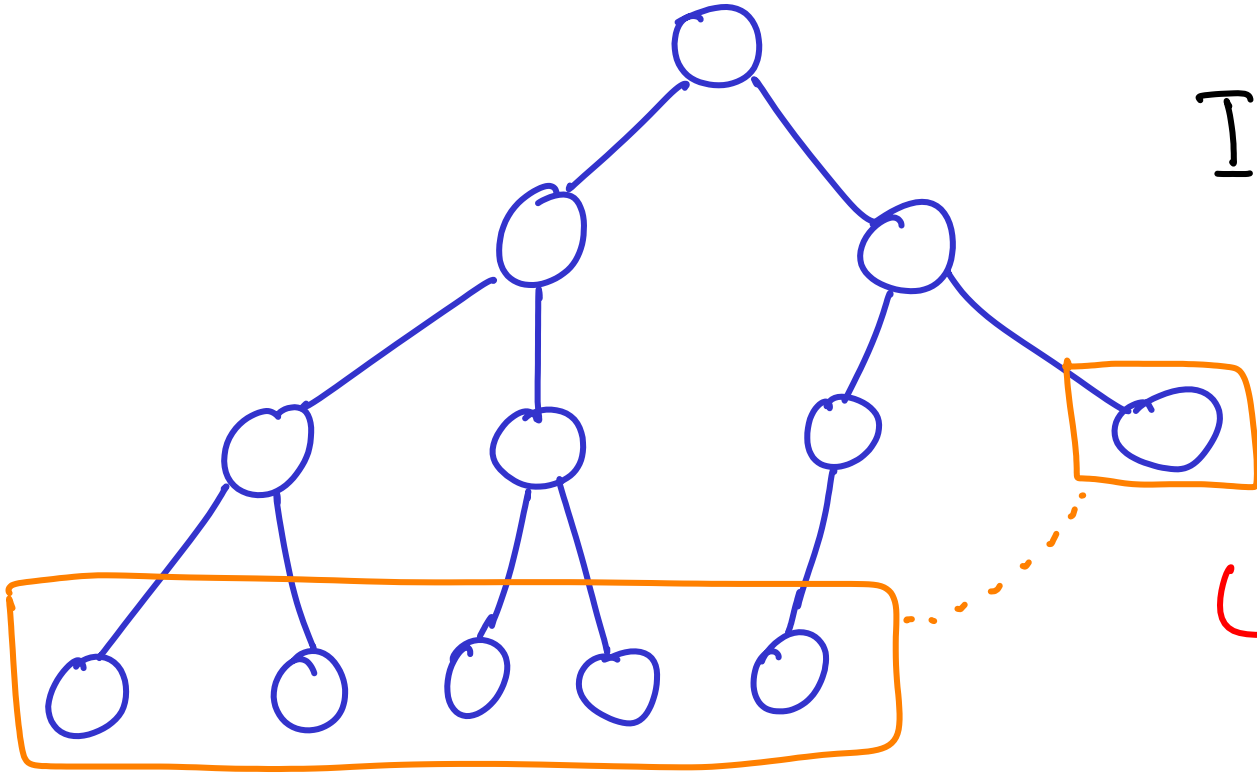


Iterate from end of array  
↳ from right to left  
on each level,  
starting at bottom



Iterate from end of array  
↳ from right to left  
on each level,  
starting at bottom

heapify each node  $x$   
↳ i.e. subtree at  $x$ .



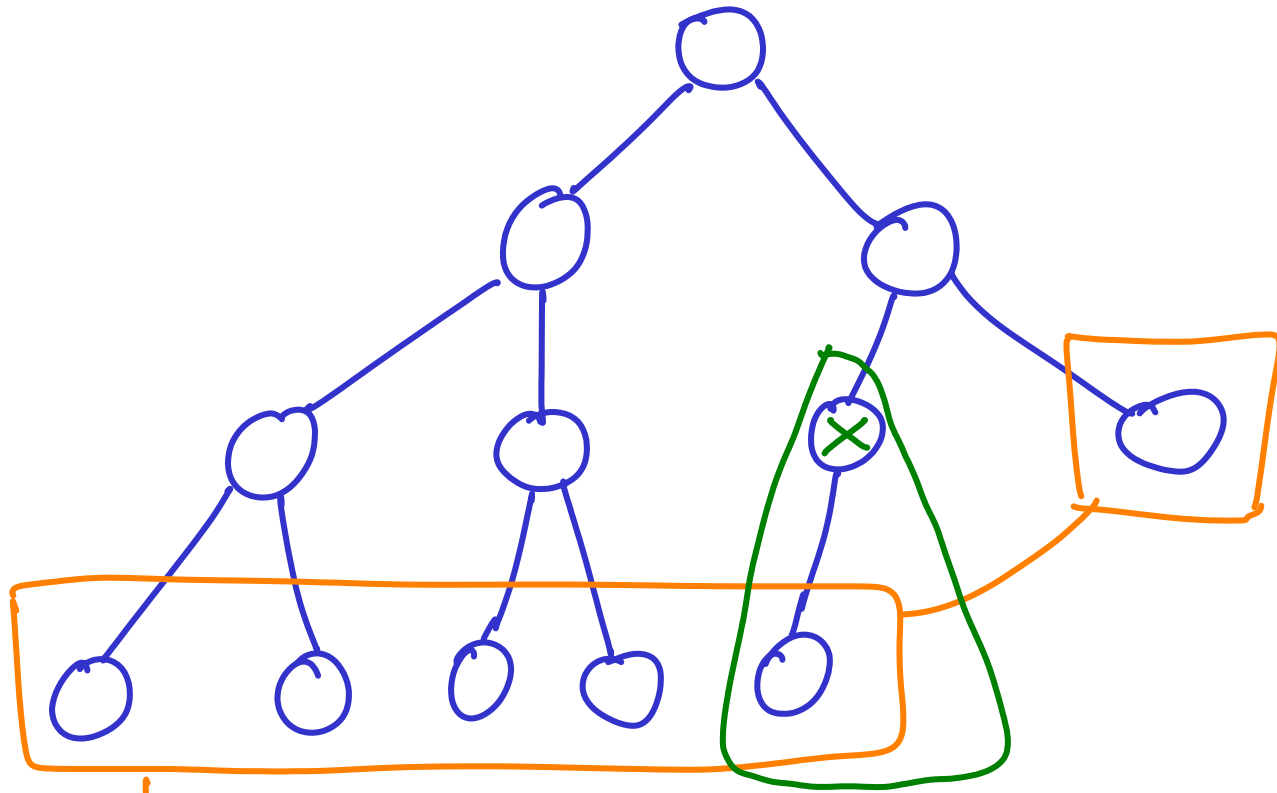
↳ already heaps

Iterate from end of array  
↳ from right to left  
on each level,  
starting at bottom



heapify each node  $x$   
↳ i.e. subtree at  $x$ .

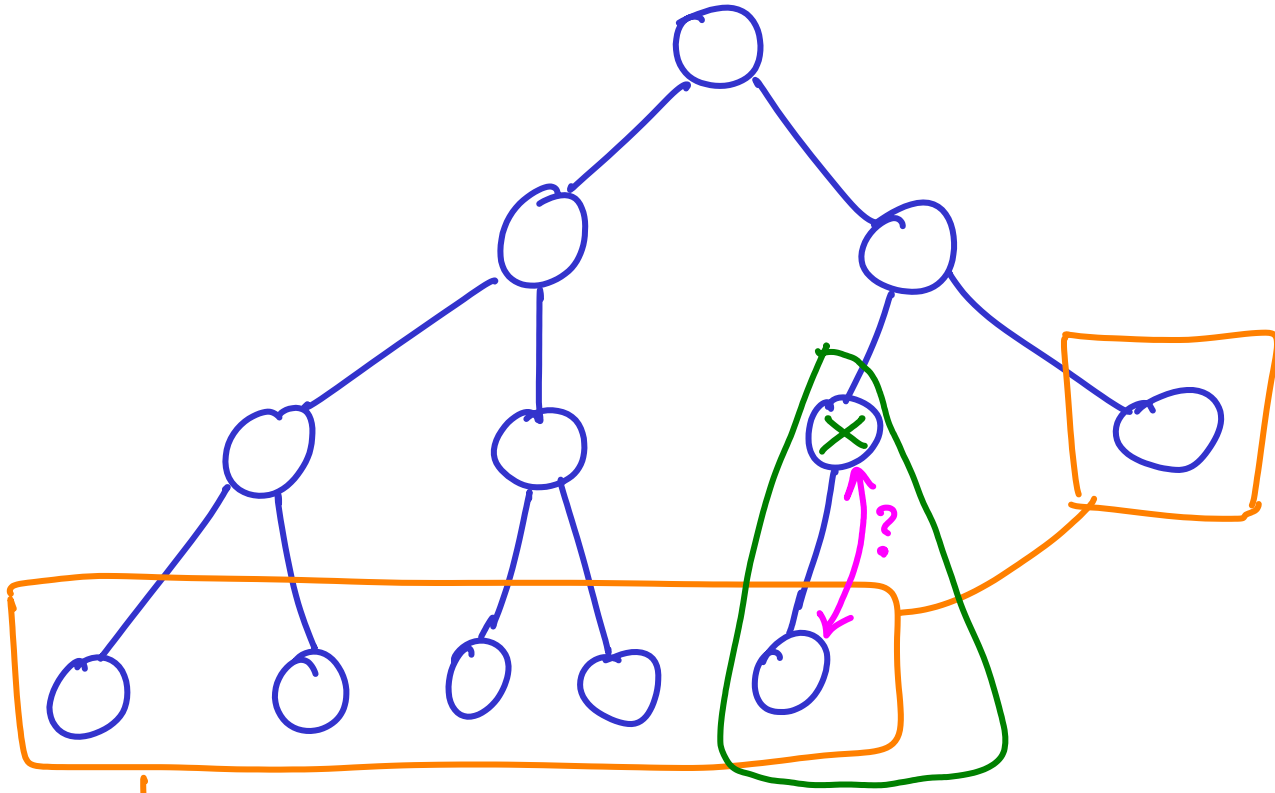




↳ already heaps

Heapify  $x$ :

|| can assume each child  
|| is a heap.

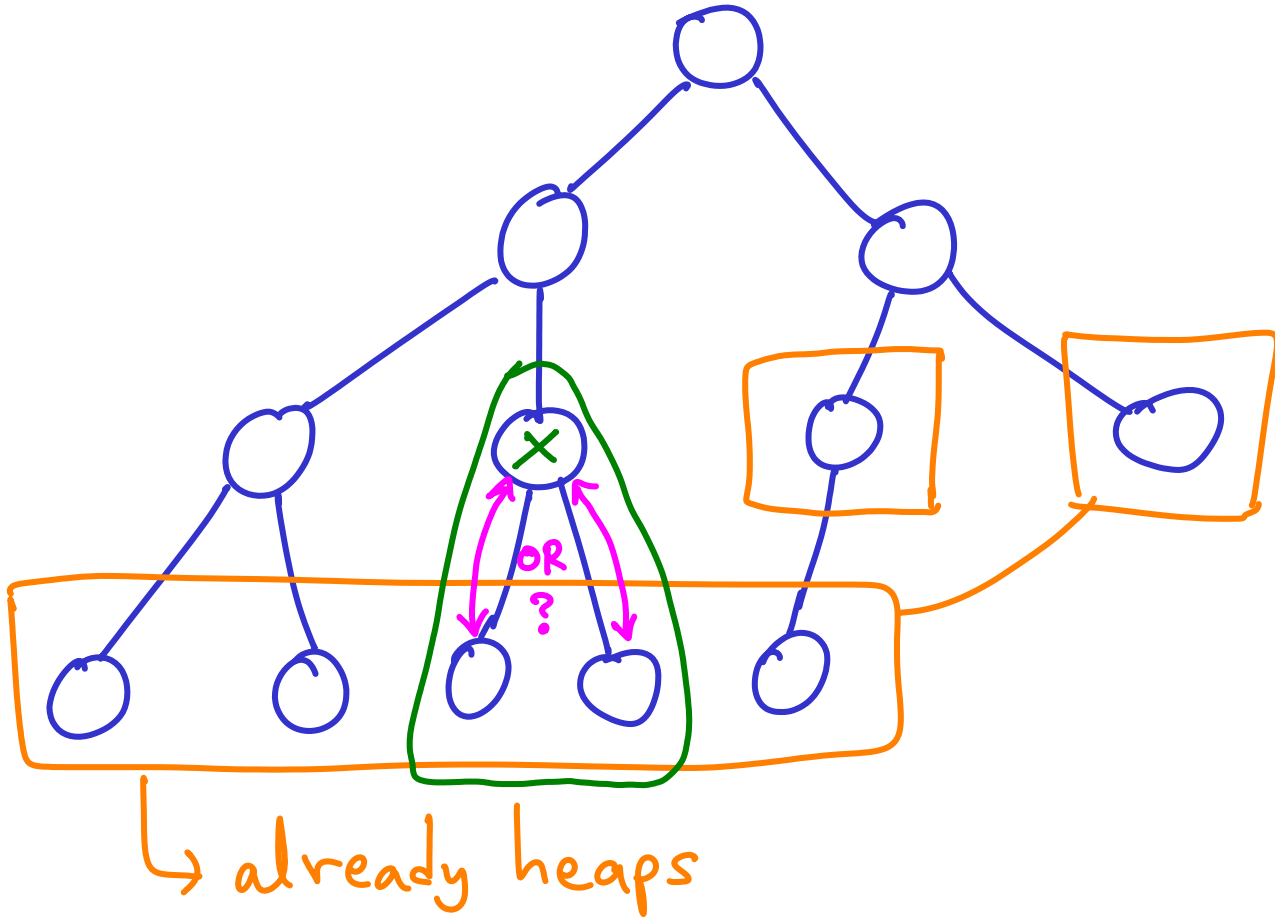


↳ already heaps

Heapify x:

|| can assume each child  
|| is a heap.

Might have to swap  
x w/ one of its children  
& further down levels

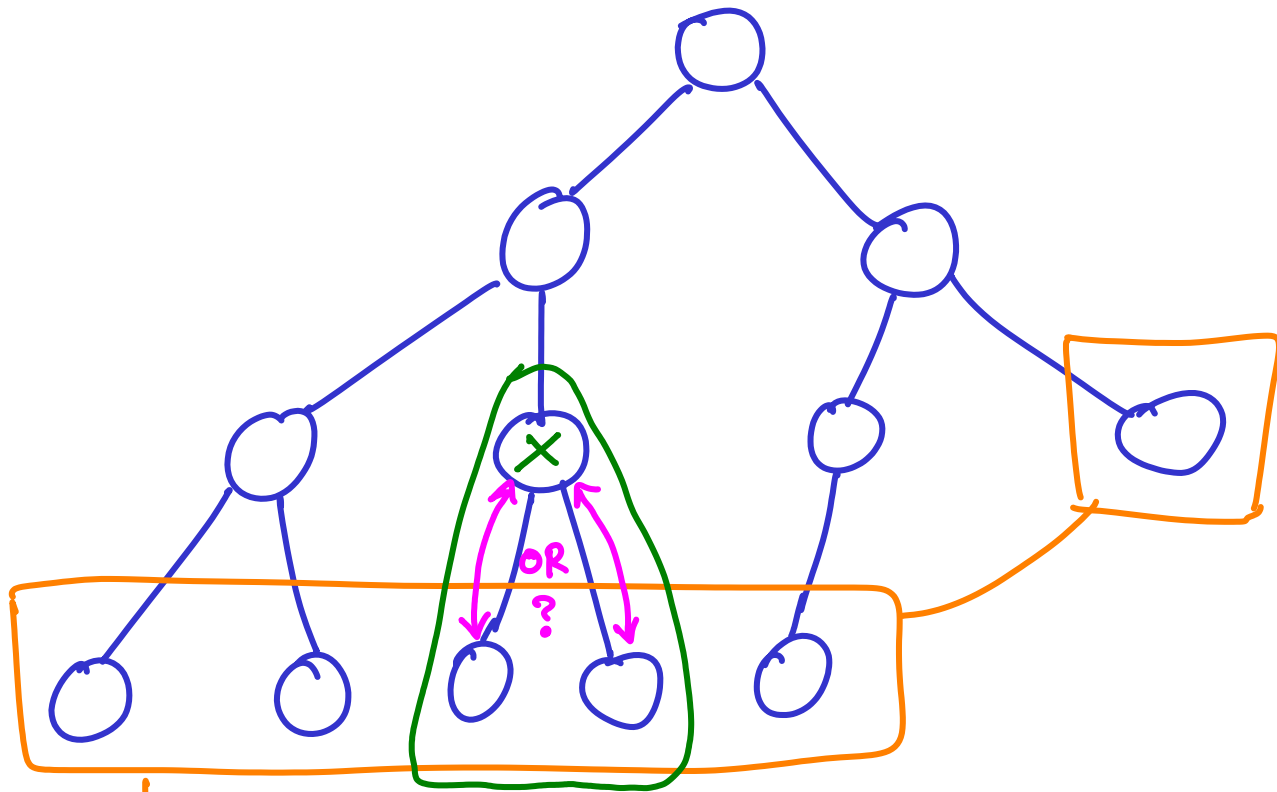


Heapify  $x$ :

|| can assume each child  
|| is a heap.

Might have to swap  
 $x$  w/ one of its children  
& further down levels

Compare two children, determine max.  
Then compare  $x$  with max, swap if needed



↳ already heaps

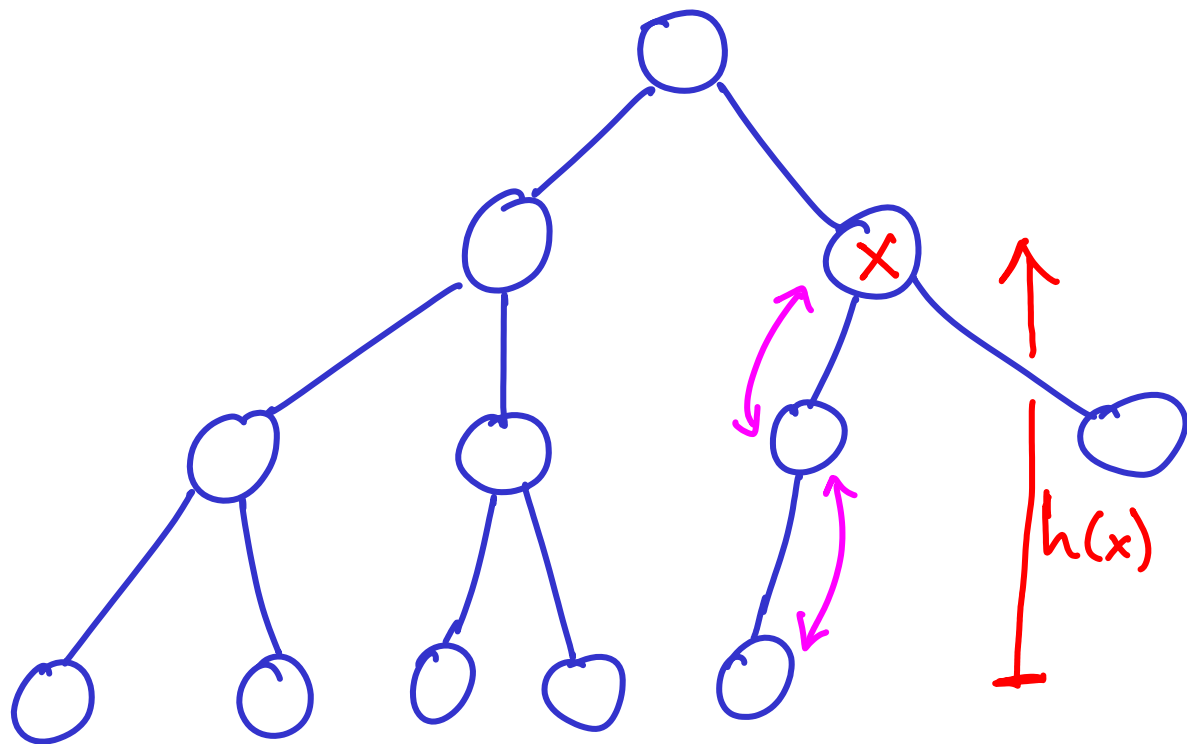
Heapify  $x$ :

|| can assume each child  
|| is a heap.

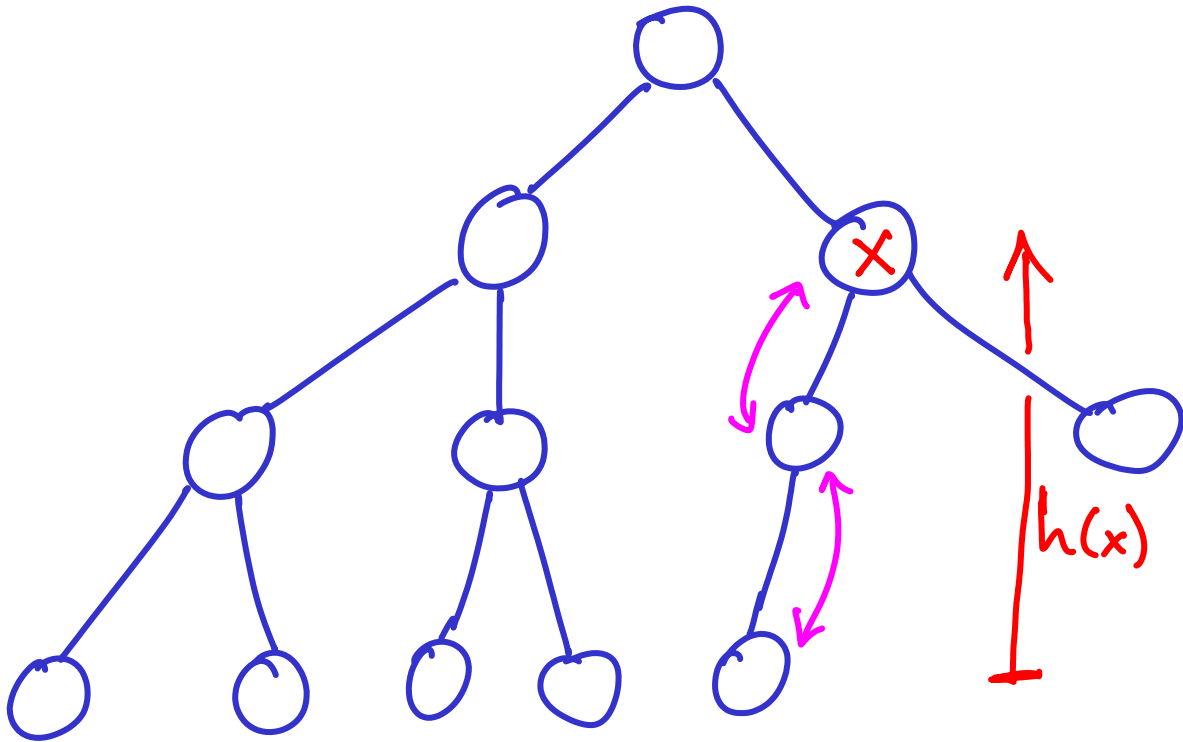
Might have to swap  
 $x$  w/ one of its children  
& further down levels

Time for  $x = O(\text{height}(x))$

↳ overall  $O(n \log n)$

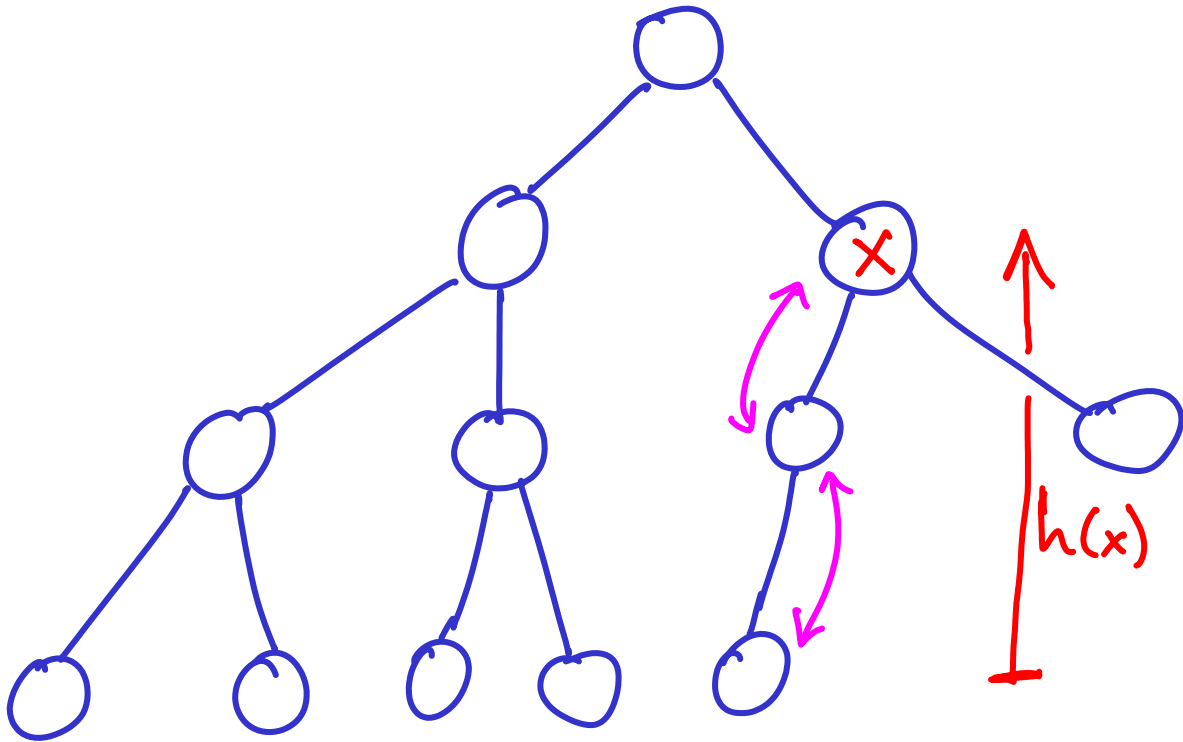


$$\text{Time} = O(h(x))$$



$$\text{Time} = O(h(x))$$

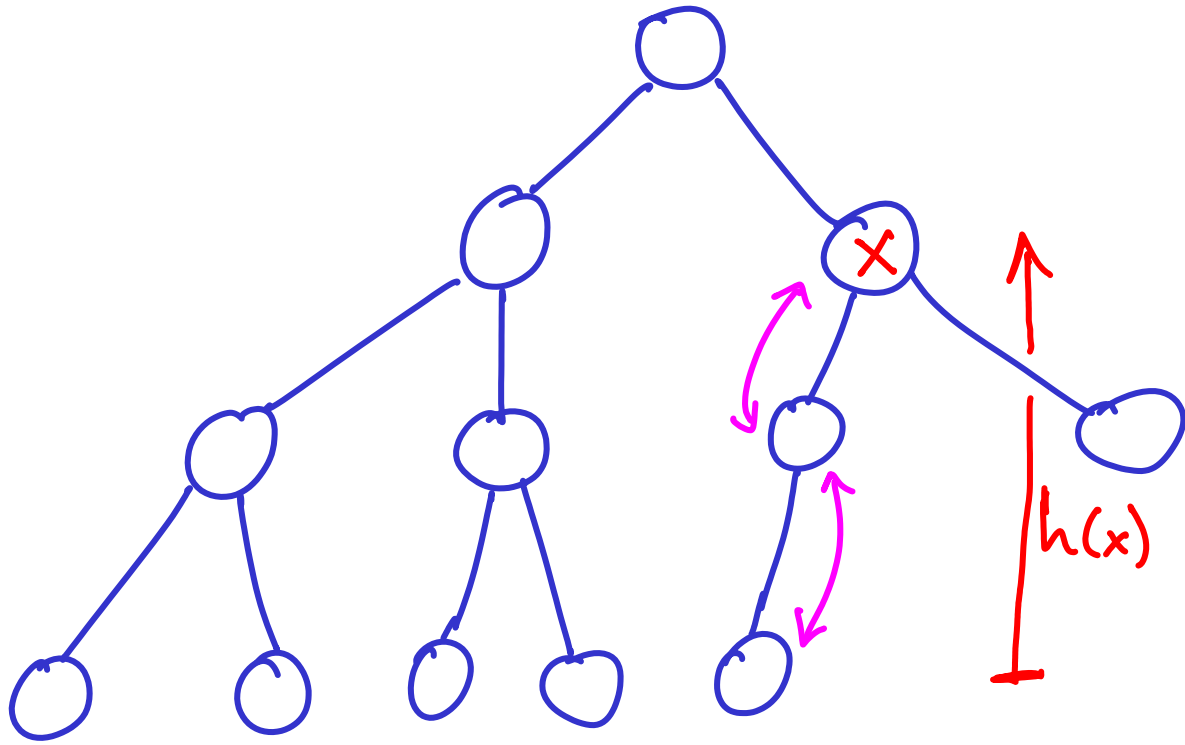
$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$



$$\text{Time} = O(h(x))$$

$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$

$$\sum \leq \underbrace{\frac{n}{2} \cdot 1}_{\substack{\text{nodes} \\ \text{lowest level}}} + \underbrace{\frac{n}{4} \cdot 2}_{\substack{\text{nodes} \\ \text{height}}} + \underbrace{\frac{n}{8} \cdot 3}_{\substack{\text{nodes} \\ \text{height}}} + \dots + \underbrace{2 \cdot ((\log n) - 1)}_{\substack{\text{nodes} \\ \text{height}}} + \underbrace{1 \cdot \log n}_{\text{root}}$$



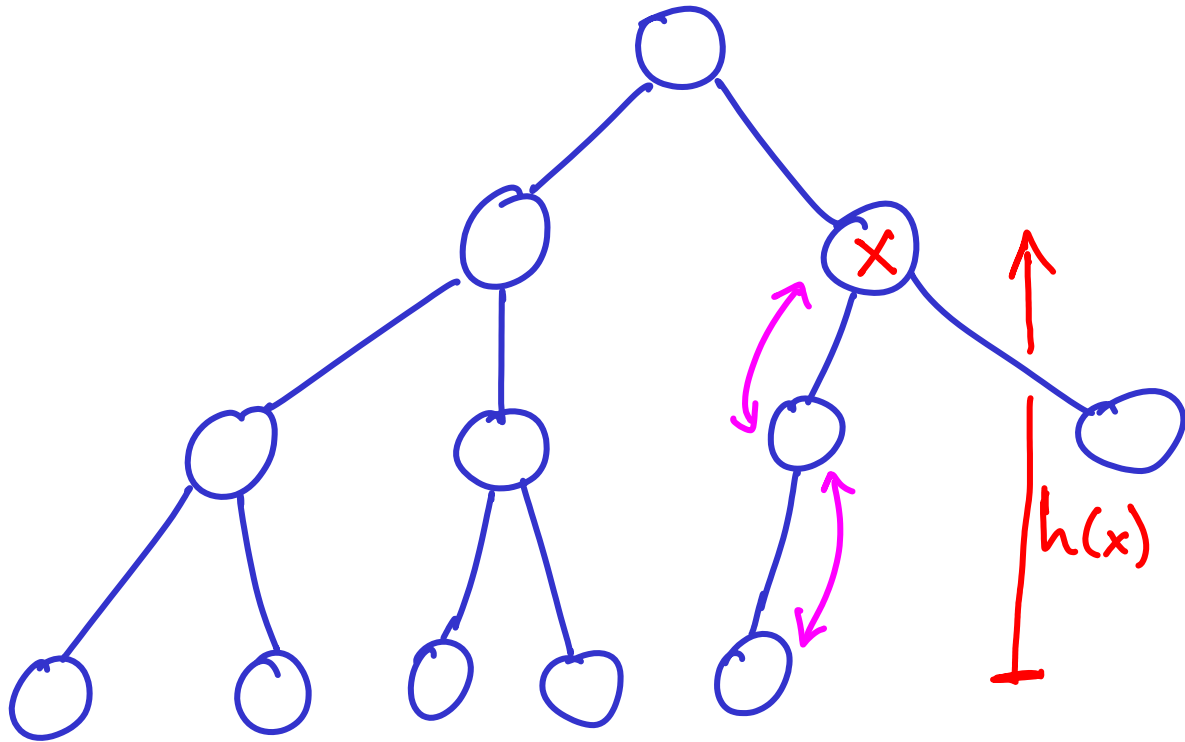
$$\text{Time} = O(h(x))$$

$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$

$$\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n$$

$$= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h$$



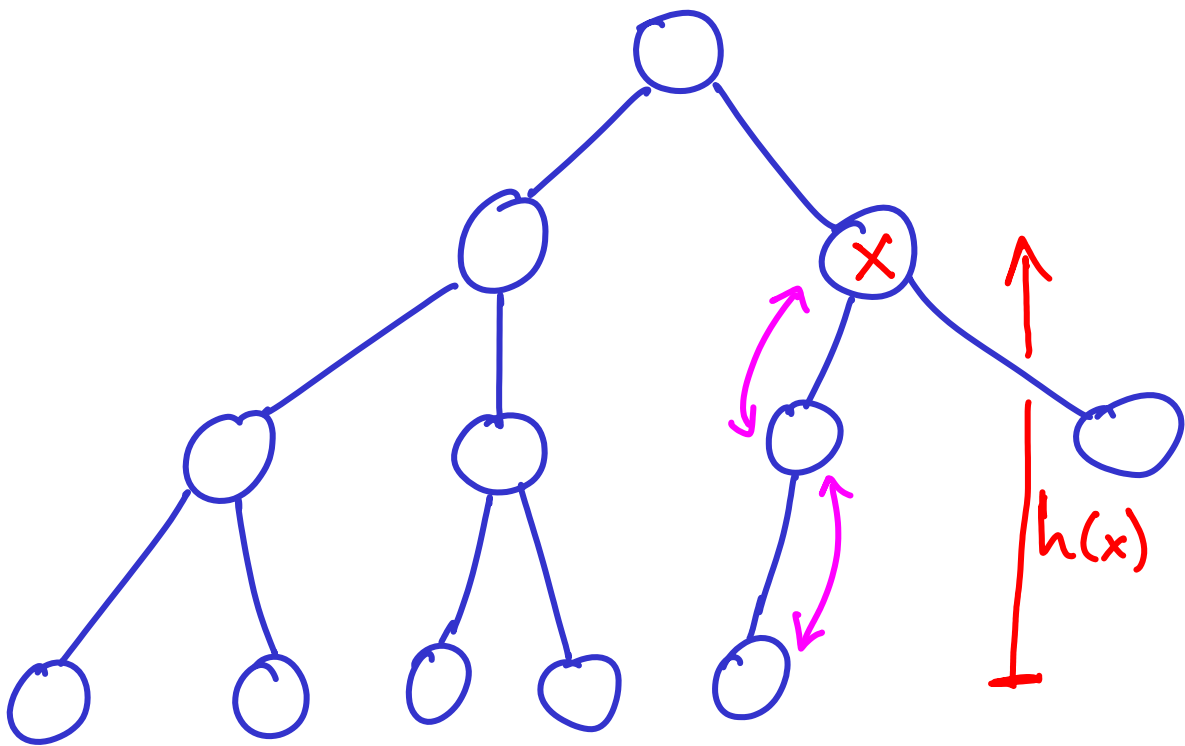


$$\text{Time} = O(h(x))$$

$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$

$$\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n$$

$$= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h}$$



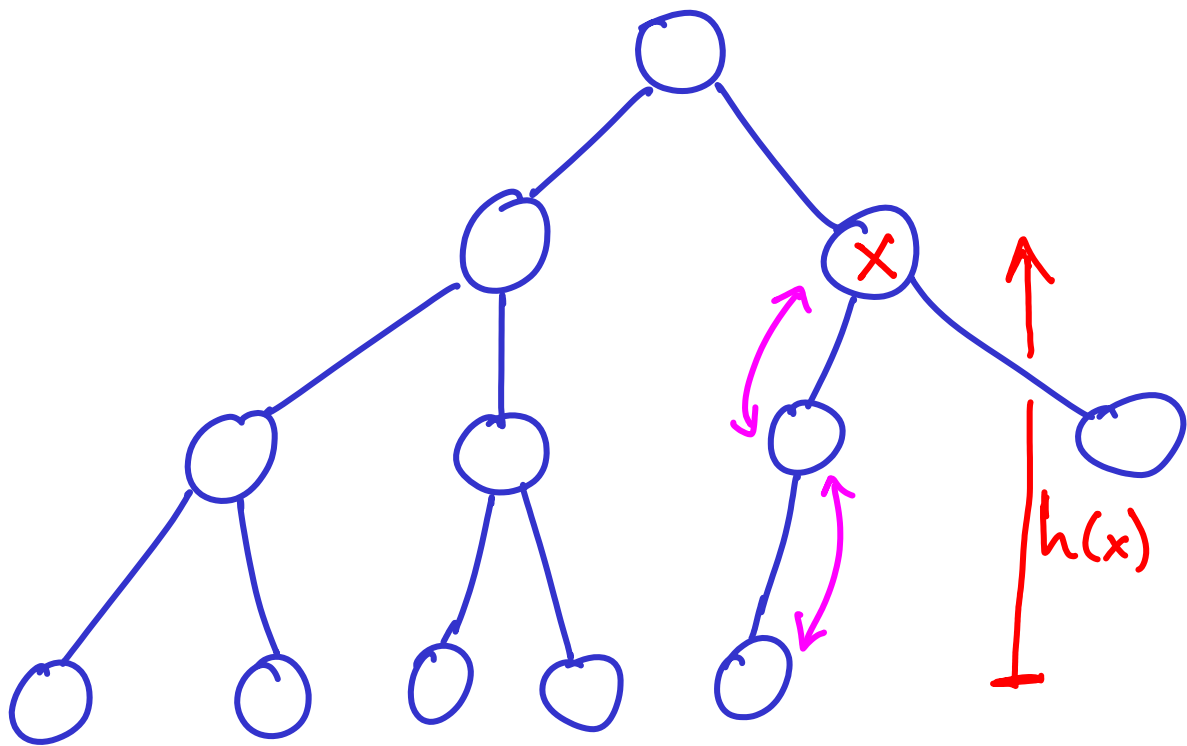
$$\text{Time} = O(h(x))$$

$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$

$$\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n$$

$$= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h} \leq n \frac{1/2}{(1-1/2)^2}$$

CLRS 1148  
 use  $\sum_{k=0}^{\infty} kx^k$



$$\text{Time} = O(h(x))$$

$$\text{Total time: } O\left(\sum_{\text{all } x} h(x)\right)$$

$$\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n$$

$$= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h} \leq n \frac{1/2}{(1-1/2)^2} = O(n)$$

CLRS 1148  
 use  $\sum_{k=0}^{\infty} kx^k$