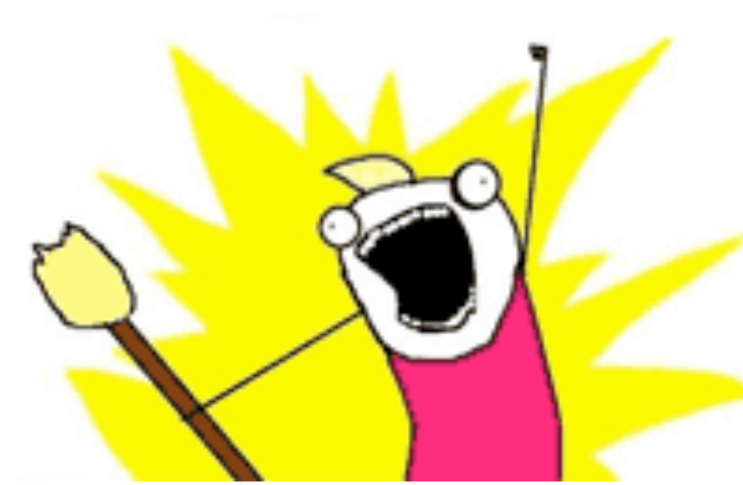
CSC263 Week 2

If you feel rusty with probabilities, please read the Appendix C of the textbook. It is only about 20 pages, and is highly relevant to what we need for CSC263.

Appendix A and B are also worth reading.

Problem Set 1 is due this Tuesday!

(Sept 29)



This week topic

➔ADT: **Priority Queue**

➔Data structure: **Heap**

An ADT we already know

First in first serve

Queue:

- → a collection of elements
- → supported operations
  - ◆ Enqueue(Q, x)
  - ◆ Dequeue(Q)
  - ◆ PeekFront(Q)

The new ADT

Oldest person first

Max-Priority Queue:

➔ a collection of elements **with priorities**, i.e., each element x has x.priority

➔ supported operations

- **Insert**(Q, x)
  - like enqueue(Q, x)
- **ExtractMax**(Q)
  - like dequeue(Q)
- **Max**(Q)
  - like PeekFront(Q)
- **IncreasePriority**(Q, x, k)
  - increase x.priority to k

Applications of Priority Queues

- ➔Job scheduling in an operating system
  - ◆Processes have different priorities (Normal, high...)
- ➔Bandwidth management in a router
  - ◆Delay sensitive traffic has higher priority
- ➔Find minimum spanning tree of a graph
- ➔etc.

Now, let’s implement a (Max)-Priority Queue

40 -> 33 -> 18 -> 65 -> 24 -> 25

Use an **unsorted linked list**

- ➔ **INSERT(Q, x)** # x is a node
  - Just insert x at the head, which takes **Θ(1)**
- ➔ **IncreasePriority(Q, x, k)**
  - Just change x.priority to k, which takes **Θ(1)**
- ➔ **Max(Q)**
  - Have to go through the whole list, takes **Θ(n)**
- ➔ **ExtractMax(Q)**
  - Go through the whole list to find x with max priority (O(n)), then delete it (O(1) if doubly linked) and return it, so overall **Θ(n)**.

65 -> 40 -> 33 -> 25 -> 24 -> 18

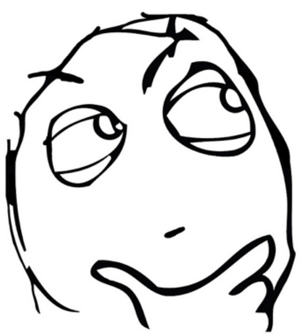
Use a reversely sorted linked list

- **Max(Q)**
  - Just return the head of the list, $\Theta(1)$
- **ExtractMax(Q)**
  - Just delete and return the head, $\Theta(1)$
- **INSERT(Q, x)**
  - Have to linearly search the correct location of insertion which takes $\Theta(n)$ in worst case.
- **IncreasePriority(Q, x, k)**
  - After increase, need to move element to a new location in the list, takes $\Theta(n)$ in worst case.

$\Theta(1)$ is fine, but $\Theta(n)$ is kind-of bad...

unsorted linked list
sorted linked list
...
Can we link these elements in a smarter way, so that we never need to do $\Theta(n)$?

Why does a sorted array also not work?

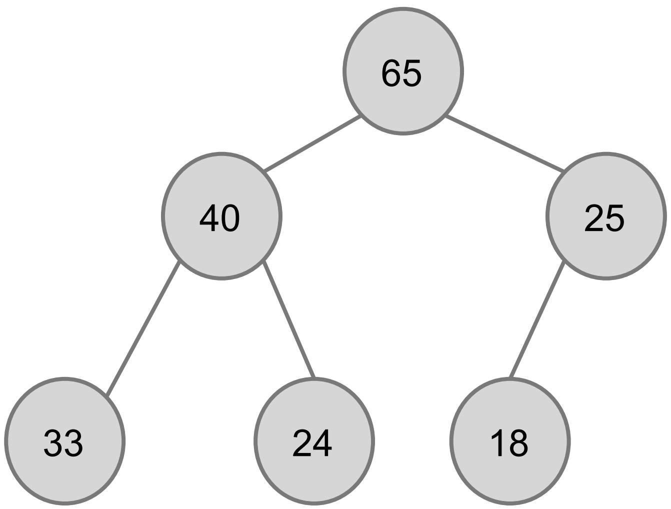


Yes, we can!

Worst case running times

	unsorted list	sorted list	Heap
Insert(Q, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Max(Q)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
ExtractMax(Q)	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
IncreasePriority (Q, x, k)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$

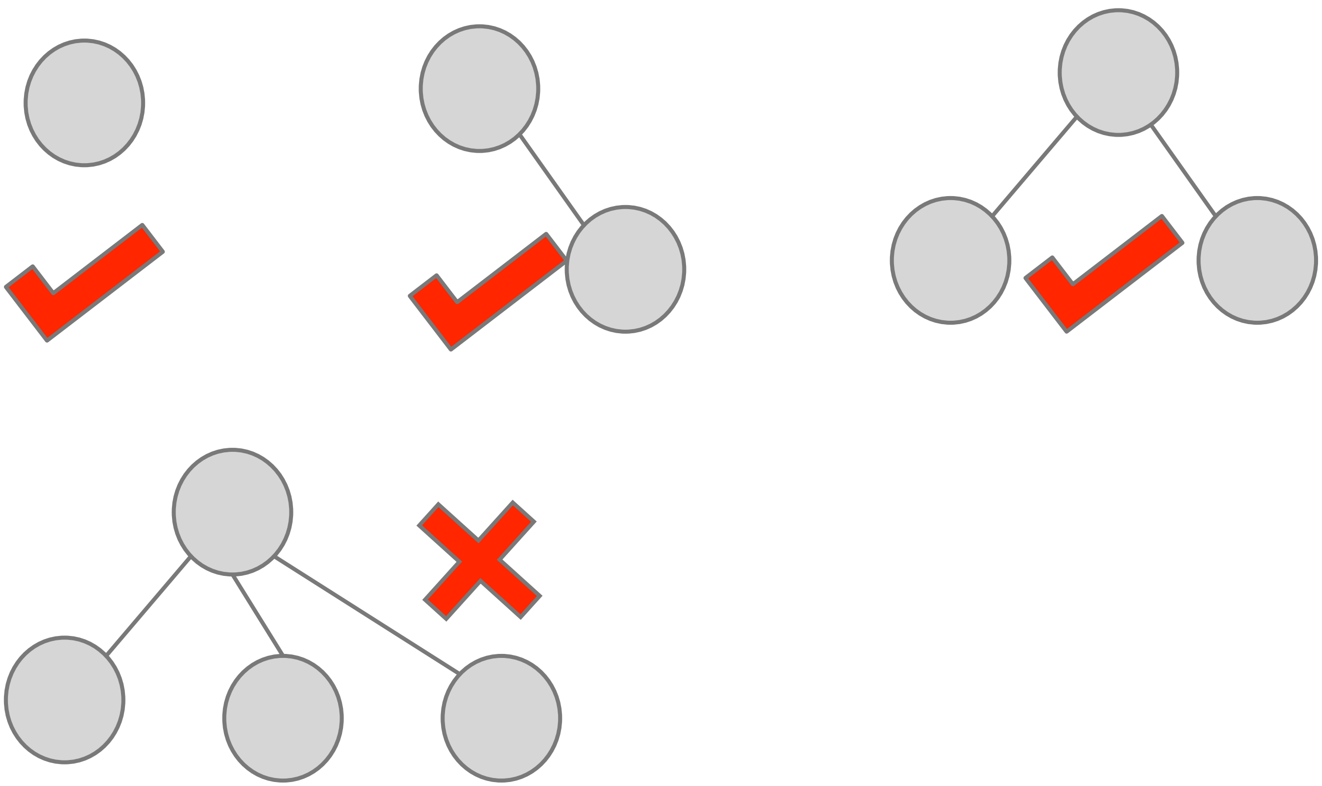
Binary Max-Heap



A **binary max-heap** is a **nearly-complete binary** tree that has the **max-heap property**.

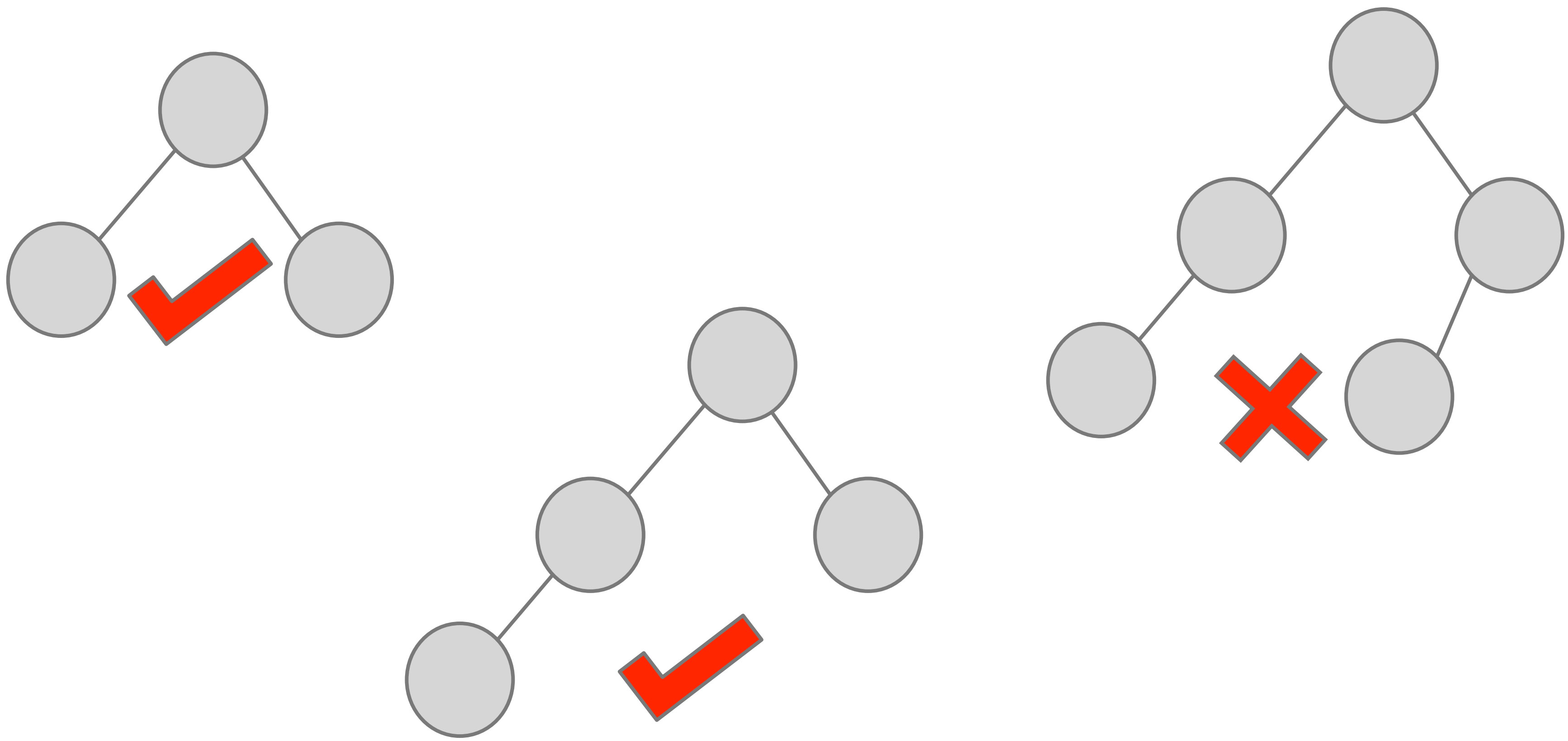
It's a binary tree

Each node has **at most** 2 children



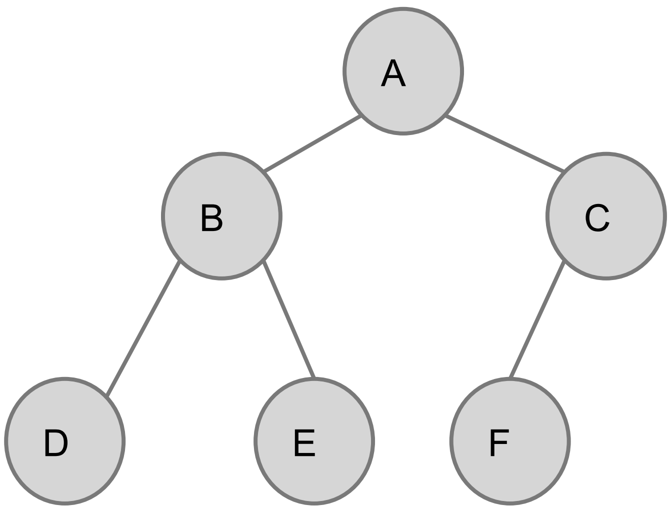
It's a nearly-complete binary tree

Each level is **completely filled**, except the bottom level where nodes are filled to as **far left** as possible



Why is it important to be a nearly-complete binary tree?

Because then we can **store** the tree in an **array**, and each node knows which **index** has its parent and its left/right child.



	A	B	C	D	E	F
index:	1	2	3	4	5	6

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2i + 1$$

$$\text{Parent}(i) = \text{floor}(i/2)$$

Assume index starts from 1

Why is it important to be a nearly-complete binary tree?

Another reason:

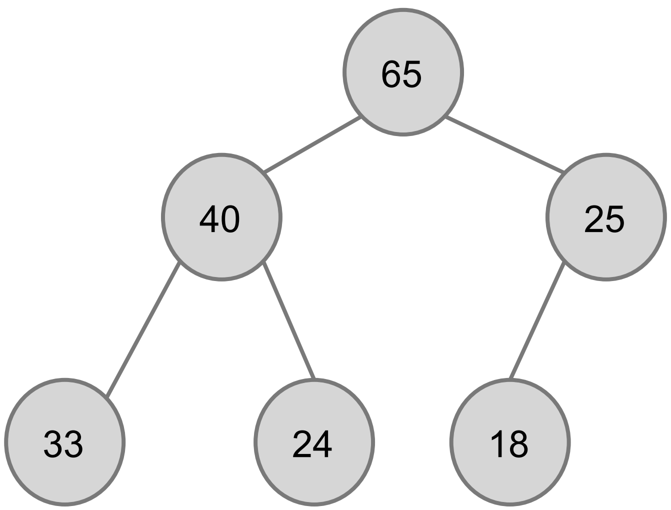
The **height** of a complete binary tree with **n** nodes is **$\Theta(\log n)$**.

This is essentially why those operations would have $\Theta(\log n)$ worst-case running time.

A thing to remember...

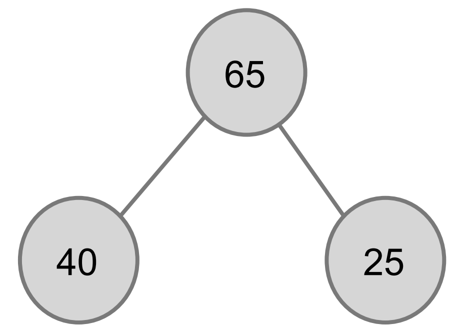
A heap is stored in an array.

Binary Max-Heap

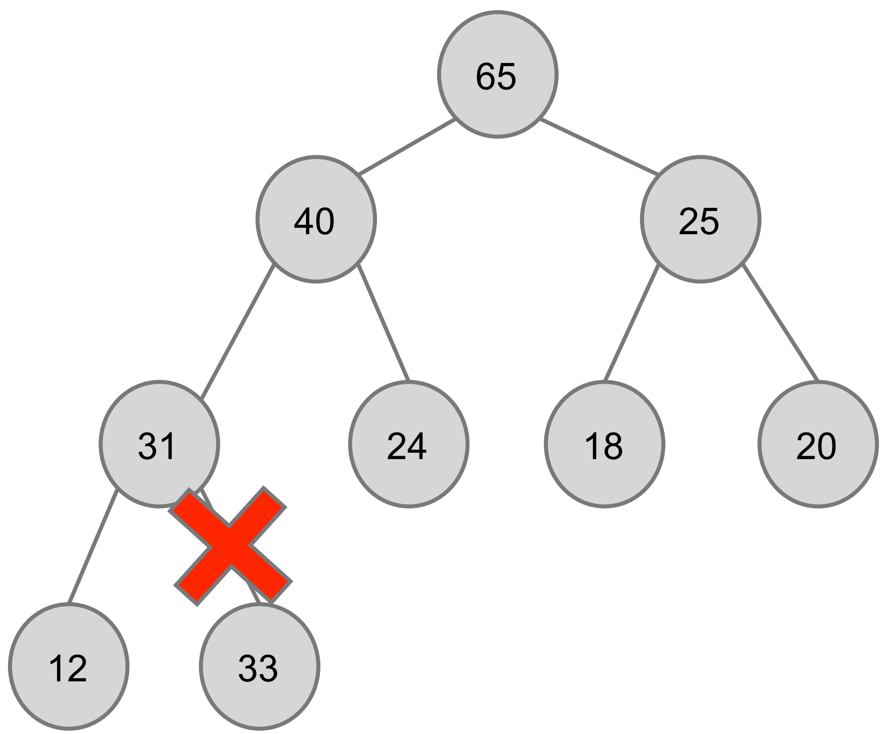


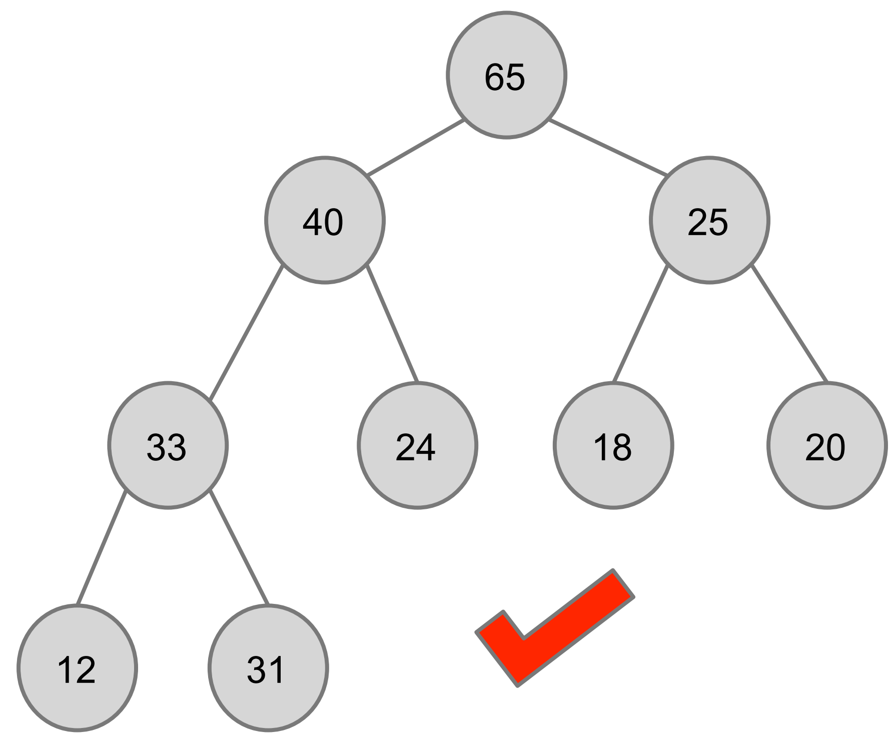
A **binary max-heap** is a **nearly-complete** **binary** tree that has the **max-heap property**.

The **max-heap property**

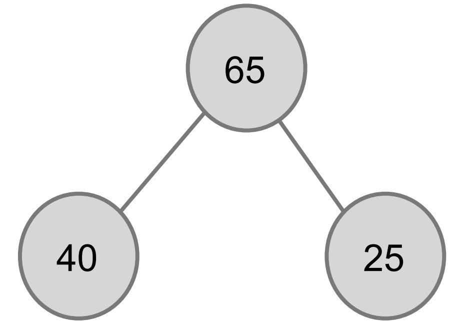


Every node has key (priority) greater than or equal to keys of its **immediate** children.



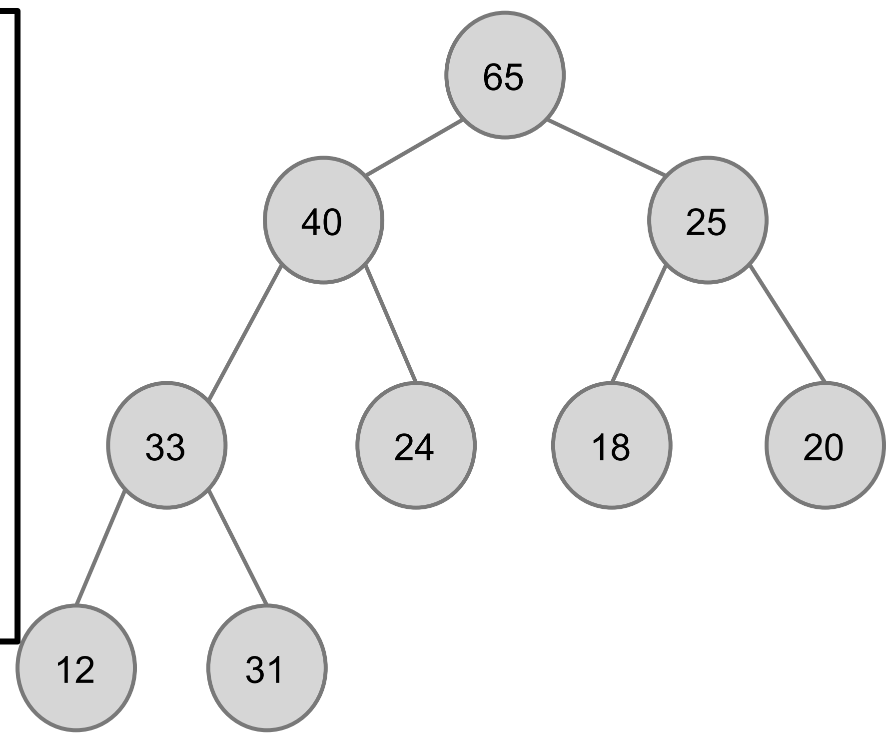


The **max-heap property**



Every node has key (priority) greater than or equal to keys of its **immediate** children.

Implication: every node is larger than or equal to **all its descendants**, i.e., every **subtree** of a heap is also a heap.



We have a binary max-heap defined, now let’s do operations on it.

- ➔Max(Q)
- ➔Insert(Q, x)
- ➔ExtractMax(Q)
- ➔IncreasePriority(Q, x, k)

Max(Q)

Return the largest key in Q,
in O(1) time

Max(Q): return the maximum element

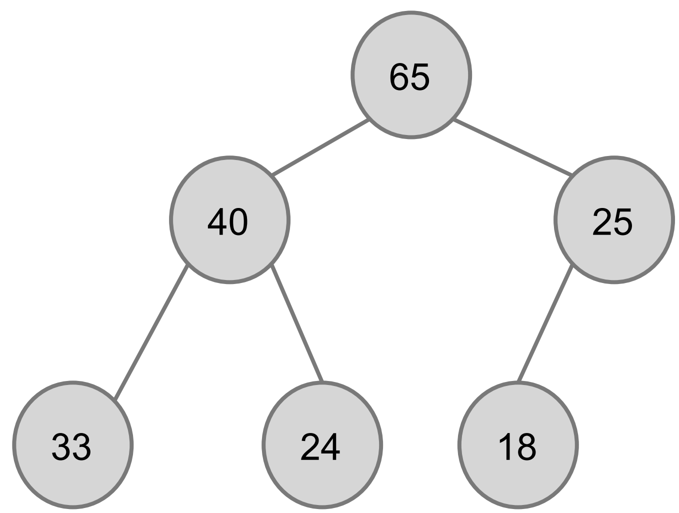
Return the **root** of the heap, i.e.,

just **return Q[1]**

(index starts from 1)

worst case $\Theta(1)$

Q	65	40	25	33	24	18



Insert(Q, x)

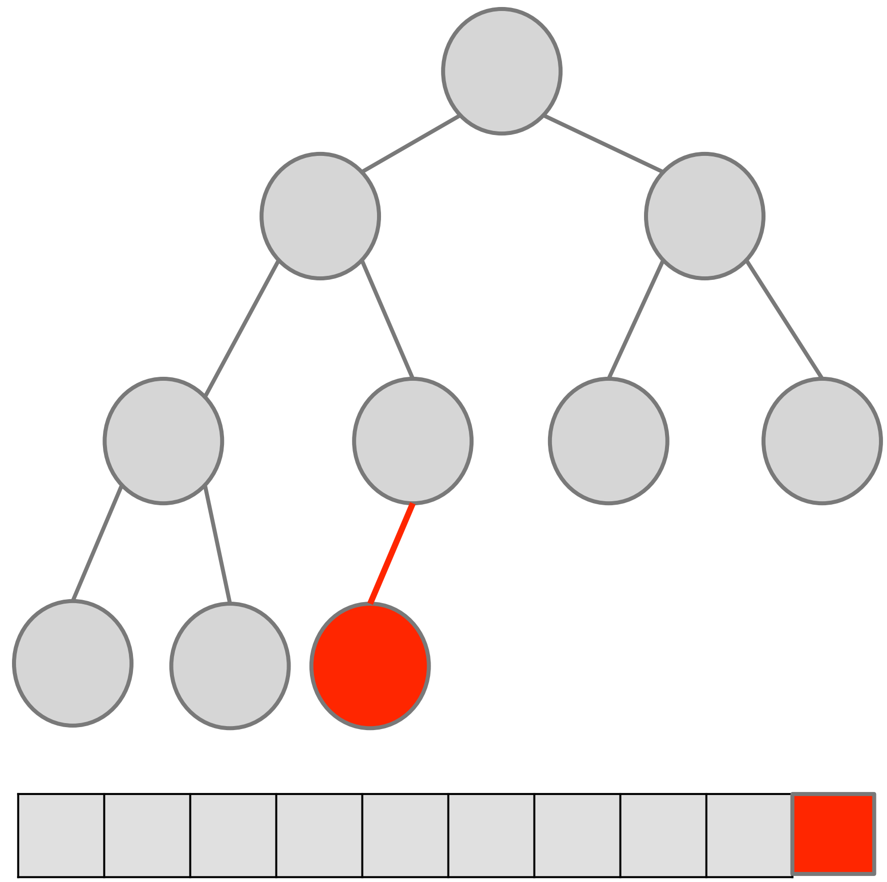
Insert node x into heap Q,
in O(logn) time

Insert(Q, x): insert a node to a heap

First thing to note:

Which spot to add the new node?

The only spot that keeps it a **complete** binary tree.

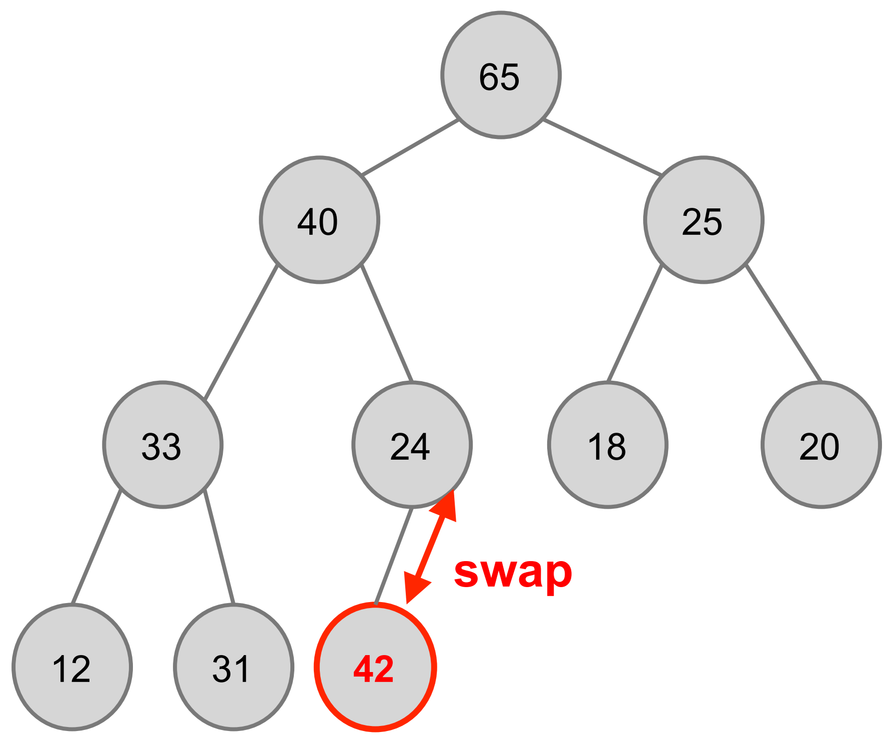


Increment heap size

Insert(Q, x): insert a node to a heap

Second thing to note: **Heap property** might be broken, how to fix it and **maintain** the heap property?

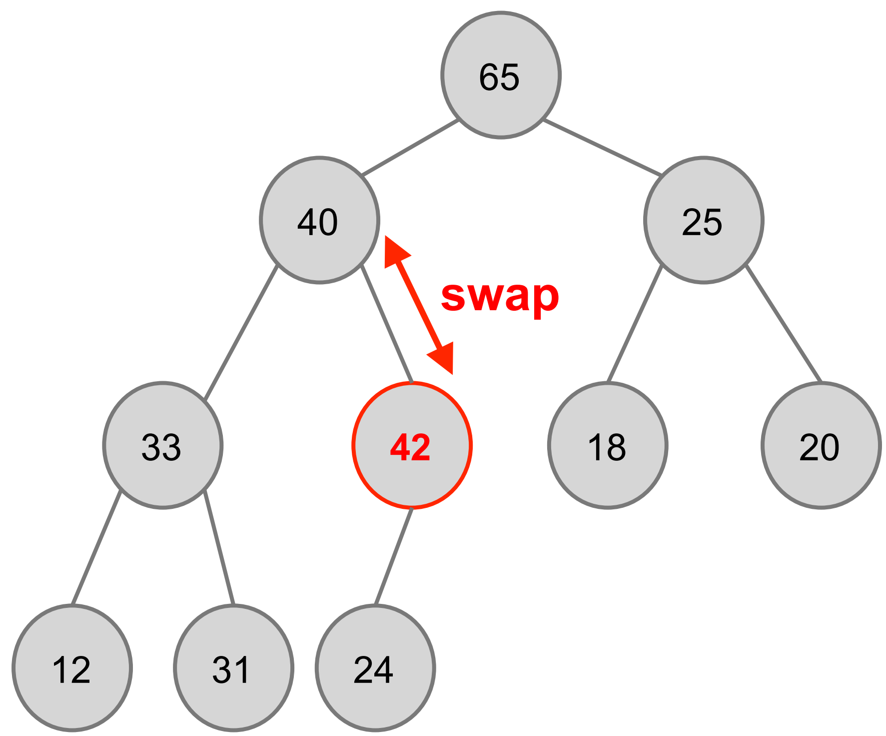
"**Bubble-up**" the new node to a proper position, by **swapping with parent.**



Insert(Q, x): insert a node to a heap

Second thing to note:
Heap property might be broken, how to fix it and **maintain** the heap property.

"**Bubble-up**" the new node to a proper position, by **swapping with parent.**

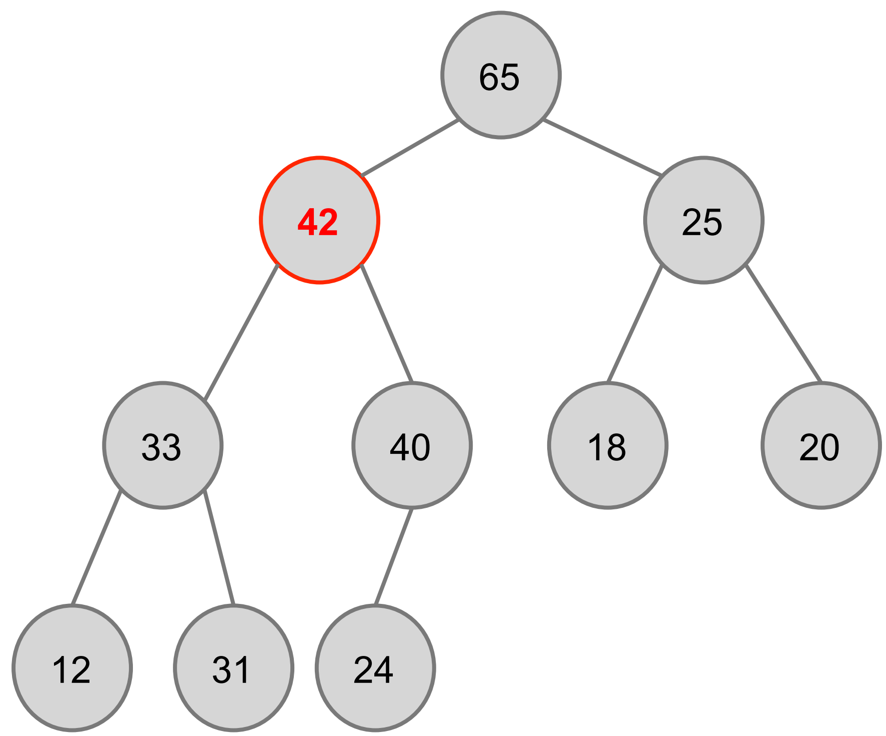


Insert(Q, x): insert a node to a heap

Second thing to note: **Heap property** might be broken, how to fix it and **maintain** the heap property.

"**Bubble-up**" the new node to a proper position, by **swapping with parent.**

Worst-case:
Θ(height) = Θ(log n)



ExtractMax(Q)

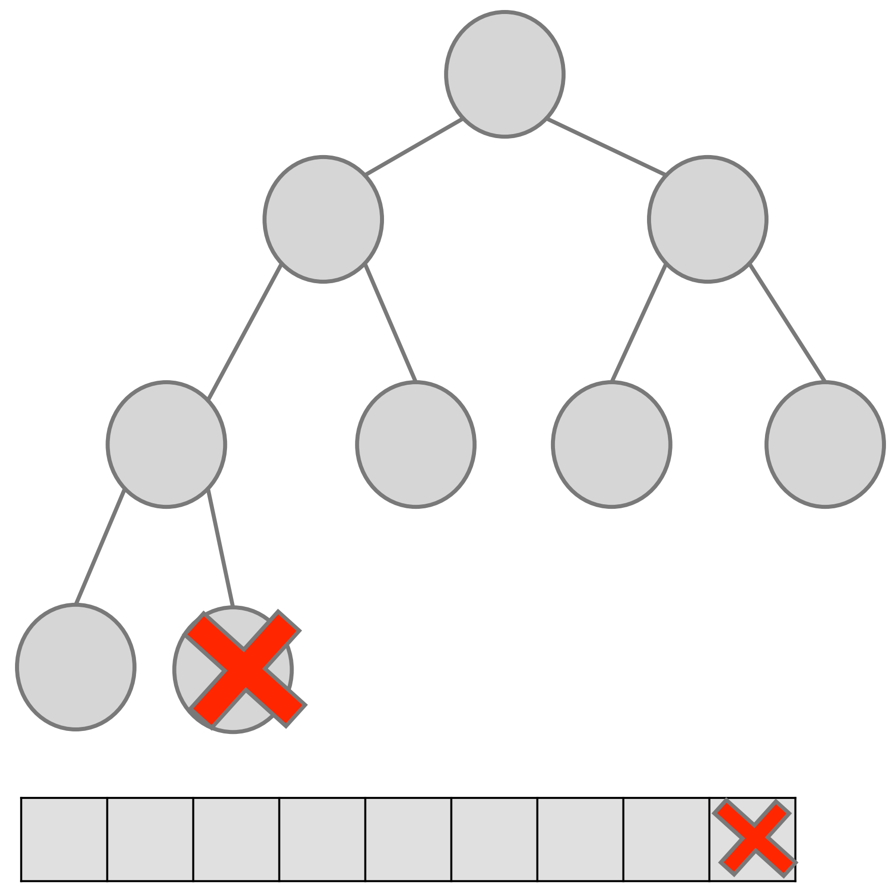
Delete and return the largest key in Q,
in O(logn) time

ExtractMax(Q): delete and return the maximum element

First thing to note:

Which **spot** to remove?

The only **spot** that keeps it a **complete** binary tree.



Decrement heap size

ExtractMax(Q): delete and return the maximum element

First thing to note:

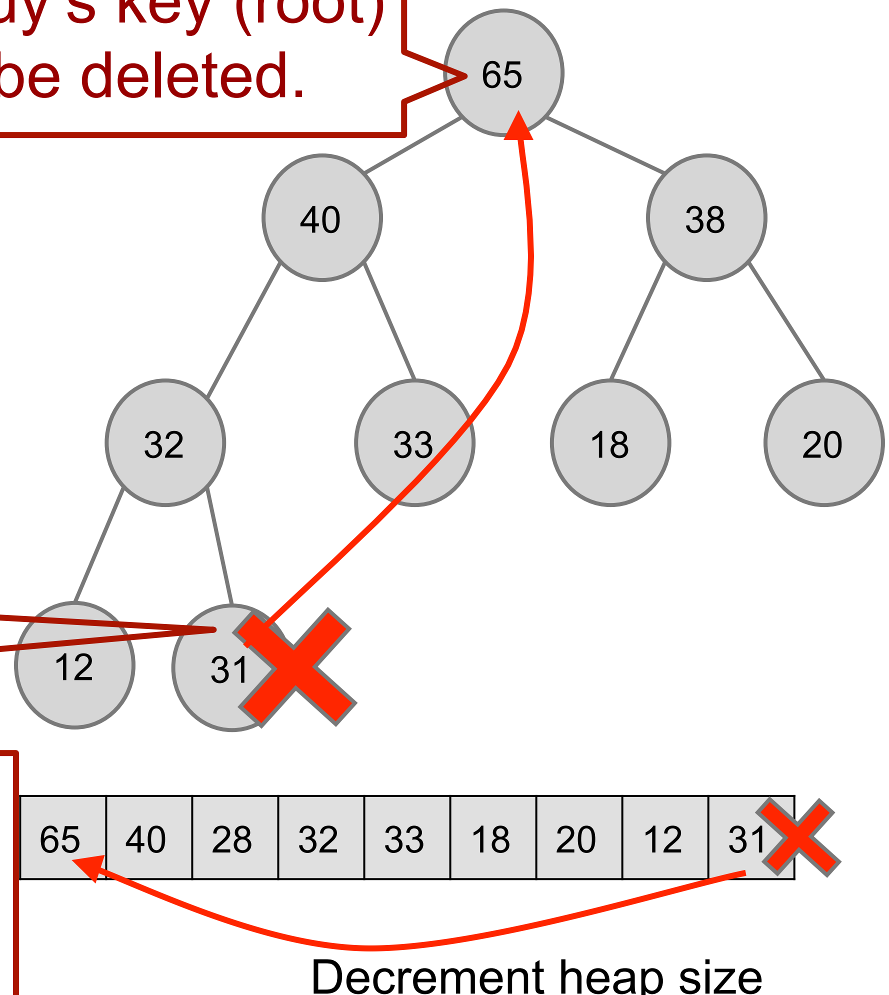
Which **spot** to remove?

The only **spot** that keeps it a **complete** binary tree.

THIS guy's key (root) should be deleted.

But the last guy's key should NOT be deleted.

Overwrite root with the **last guy**'s key, then **delete** the last guy (decrement heap size).

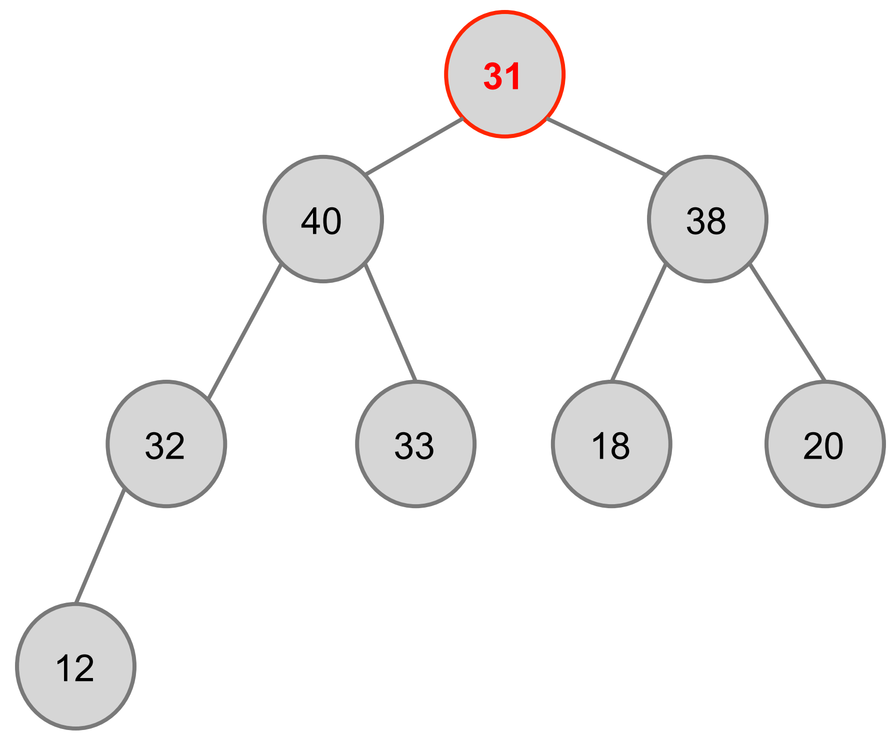


ExtractMax(Q): delete and return the maximum element

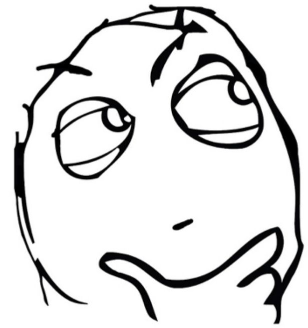
Now the **heap property** is broken again…, need to fix it.

"Bubble-down" by swapping with…

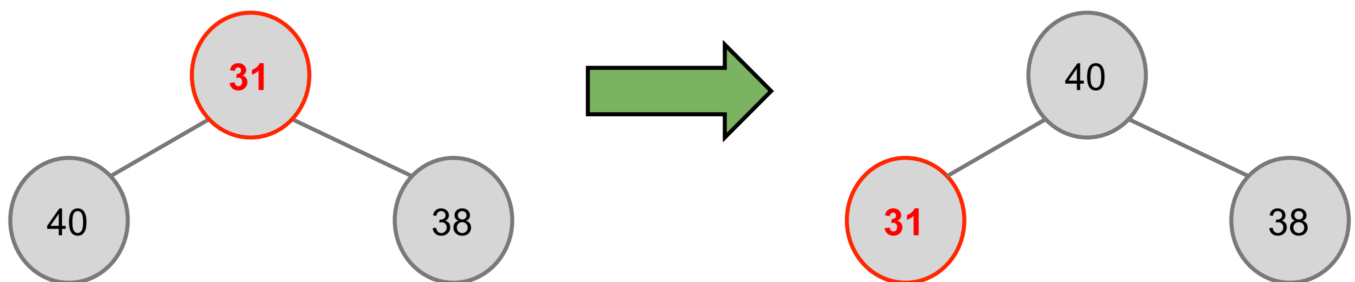
a child...



Which child to swap with?



so that, after the swap, **max-heap property** is satisfied

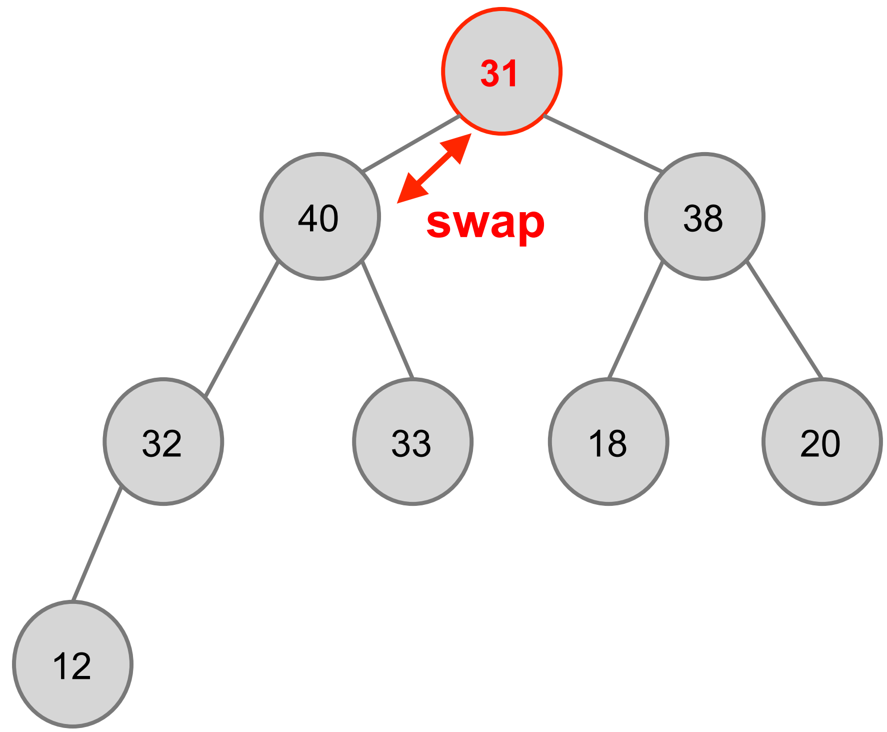


The “elder” child!

because it is the **largest** among the three

ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again…, need to fix it.

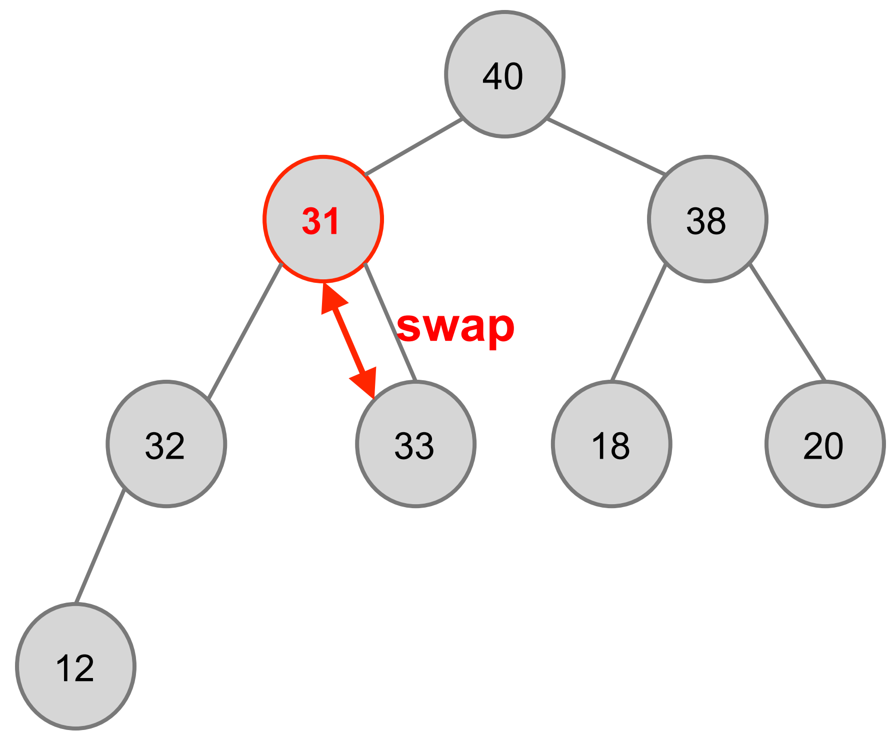


"Bubble-down" by swapping with

the elder child

ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again…, need to fix it.

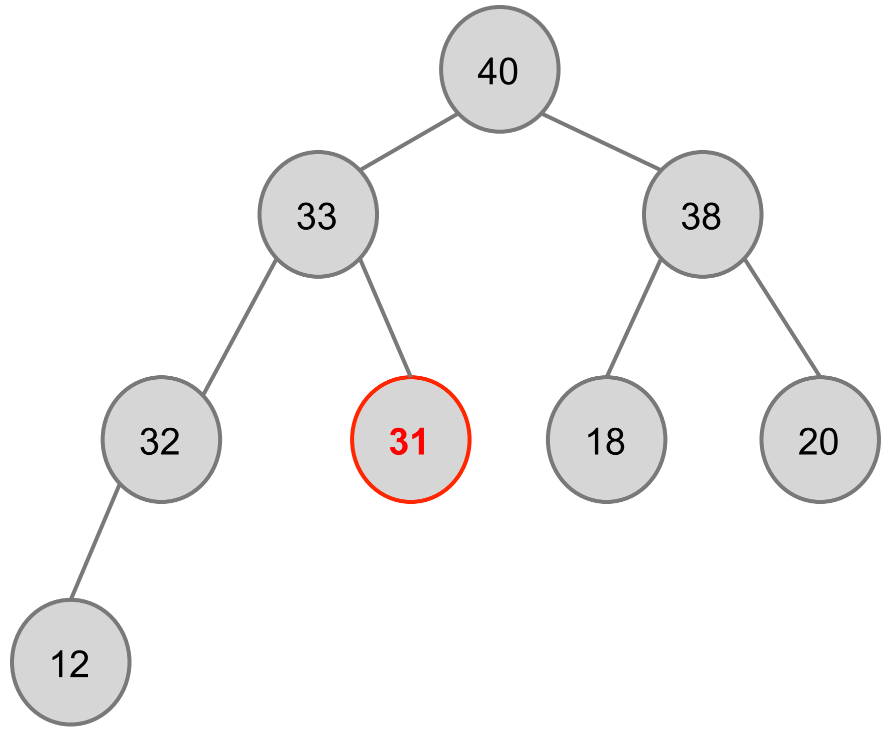


“**Bubble-down**” by swapping with...

the elder child

ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again…, need to fix it.



“Bubble-down” by swapping with

the elder child

Worst case running time: Θ(height) + some constant work

Θ(log n)

Quick summary

Insert(Q, x):
➔Bubble-up, swapping with parent

ExtractMax(Q)
➔Bubble-down, swapping elder child

Bubble up/down is also called **percolate** up/down, or **sift** up down, or **tickle** up/down, or **heapify** up/down, or **cascade** up/down.

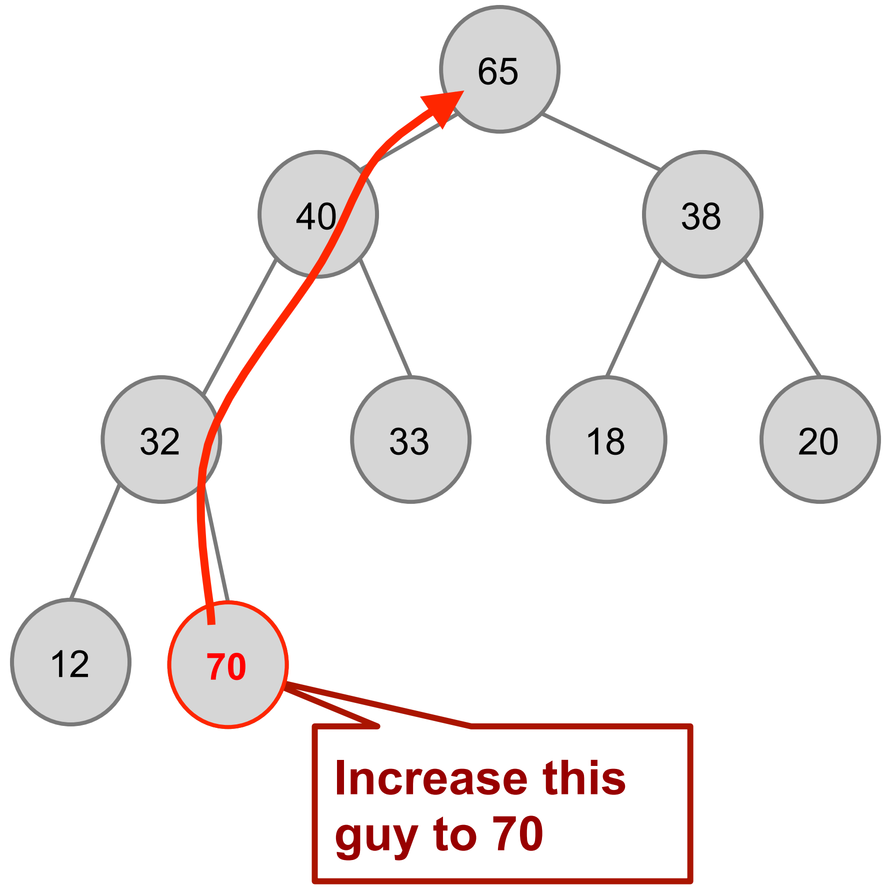
IncreasePriority(Q, x, k)

Increases the key of node x to k,
in O(logn) time

IncreasePriority(Q, x, k): increase the key of node x to k

Just increase the key, then...

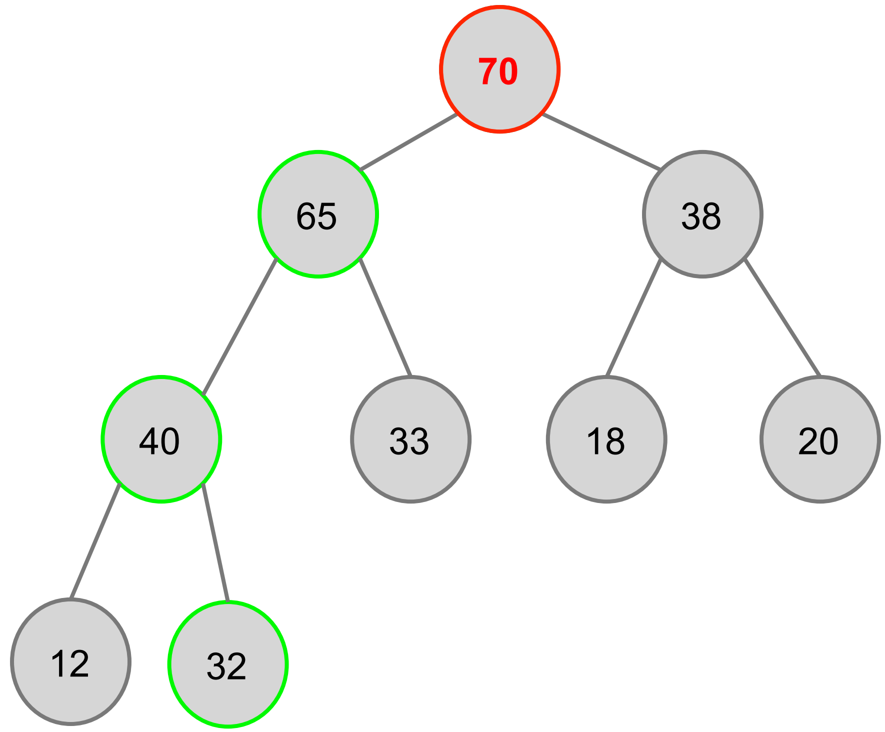
Bubble-up by swapping with parents, to proper location.



IncreasePriority(Q, x, k): increase the key of node x to k

Just increase the key, then...

Bubble-up by swapping with parents, to proper location.



Worst case running time: Θ(height) + some constant work

Θ(log n)

Now we have learned how implement a priority queue using a heap

- ➔ Max(Q)
- ➔ Insert(Q, x)
- ➔ ExtractMax(Q)
- ➔ IncreasePriority(Q, x, k)

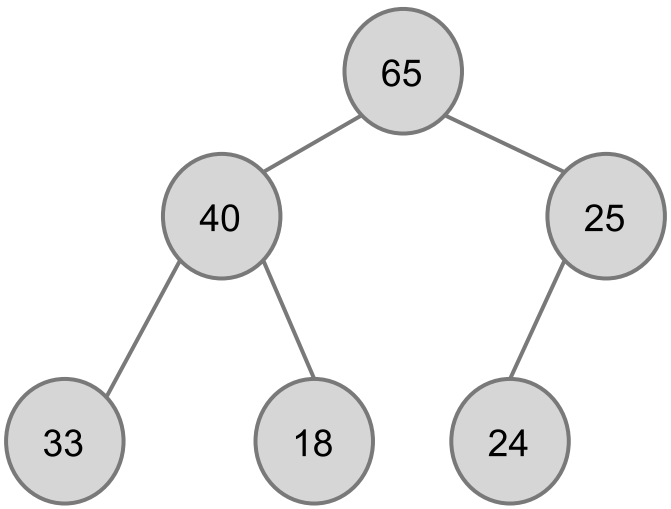
Next:

- ➔How to use heap for **sorting**
- ➔How to **build a heap** from an unsorted array

HeapSort

Sorts an array, in O(n logn) time

The idea



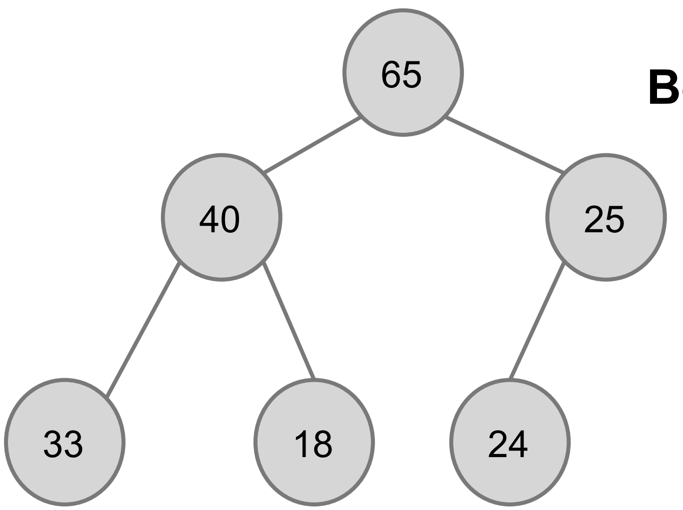
Worst-case running time: each ExtractMax is **O(log n)**, we do it **n** times, so overall it's...
O(n logn)

How to get a sorted list out of a heap with n nodes?

Keep extracting max for n times, the keys extracted will be sorted in non-ascending order.

Now let's be more precise

What's needed: modify a max-heap-ordered **array** into a **non-descendingly** sorted **array**



Before:

65	40	25	33	18	24

After:

18	24	24	25	40	65

We want to do this "**in-place**" without using any extra array space, i.e., just by **swapping** things around.

Valid heaps are green rectangled

65	40	25	33	18	24
24	40	25	33	18	**65**
24	40	25	33	18	**65**
40	33	25	**24**	18	**65**
18	33	25	24	**40**	**65**
33	25	**18**	24	**40**	**65**
25	**24**	18	**33**	**40**	**65**
24	**18**	**25**	**33**	**40**	**65**
18	**24**	**25**	**33**	**40**	**65**
18	**24**	**25**	**33**	**40**	**65**

Step 1: swap first (65) and last (24), since the tail is where 65 (max) belongs to.

Step 2: decrement heap size

This node is like deleted from the tree, not touched any more.

Step 3: fix the heap by **bubbling down** 24

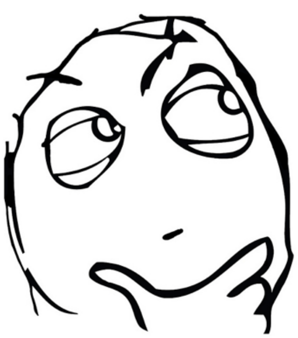
Repeat Step 1-3 until the array is fully sorted (at most n iterations).

HeapSort, the pseudo-code

```
HeapSort(A)
```
“’’sort any array A into non-descending order ’’”

Missing!

```
    BuildMaxHeap(A)  # convert any array A into a heap-ordered one
        swap A[1] and A[i]  # Step 1: swap the first and the last
        A.size ← A.size - 1  # Step 2: decrement size of heap
        BubbleDown(A, 1)  # Step 3: bubble down the 1st element in A
```



Does it work?

It works for an array A that is initially heap-ordered, it does work NOT for any array!

BuildMaxHeap(A)

Converts an array into a max-heap ordered array, in O(n) time

Convert any array into a heap ordered one

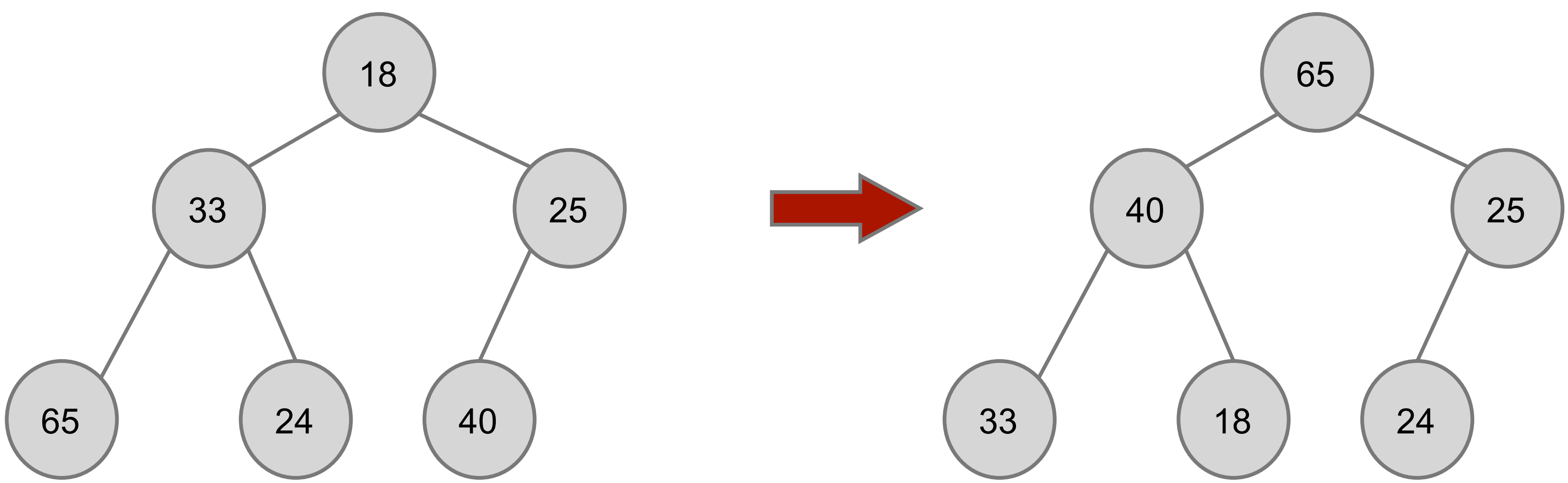
any array

18	33	25	65	24	40

heap ordered array

65	40	25	33	18	24

In other words...



Idea #1

```
BuildMaxHeap(A):

    B ← empty array  # empty heap
    for x in A:
        Insert(B, x)  # heap insert
    A ← B    # overwrite A with B
```

Running time:

Each Insert takes **O(log n)**, there are **n** inserts...

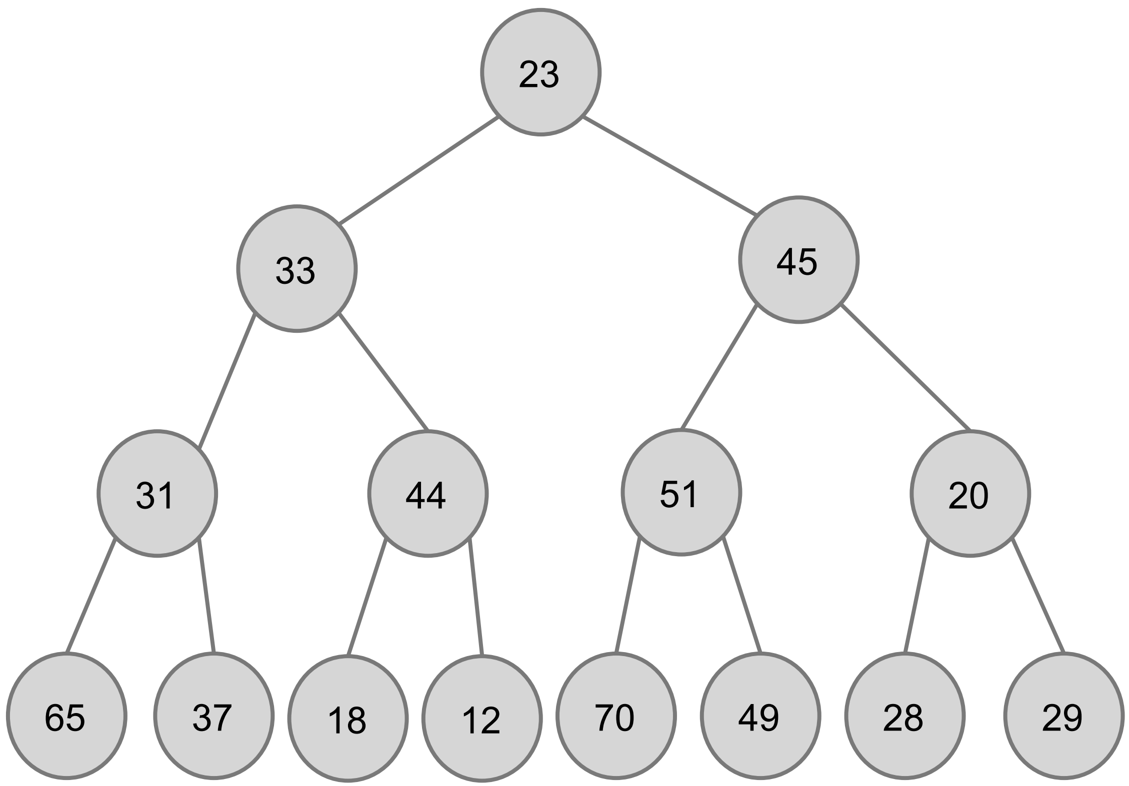
so it's **O(n log n)**, not very exciting.

Not **in-place**, needs a second array.


WHAT IF I TOLD YOU
YOU CAN DO BETTER THAN THIS
imgflip.com

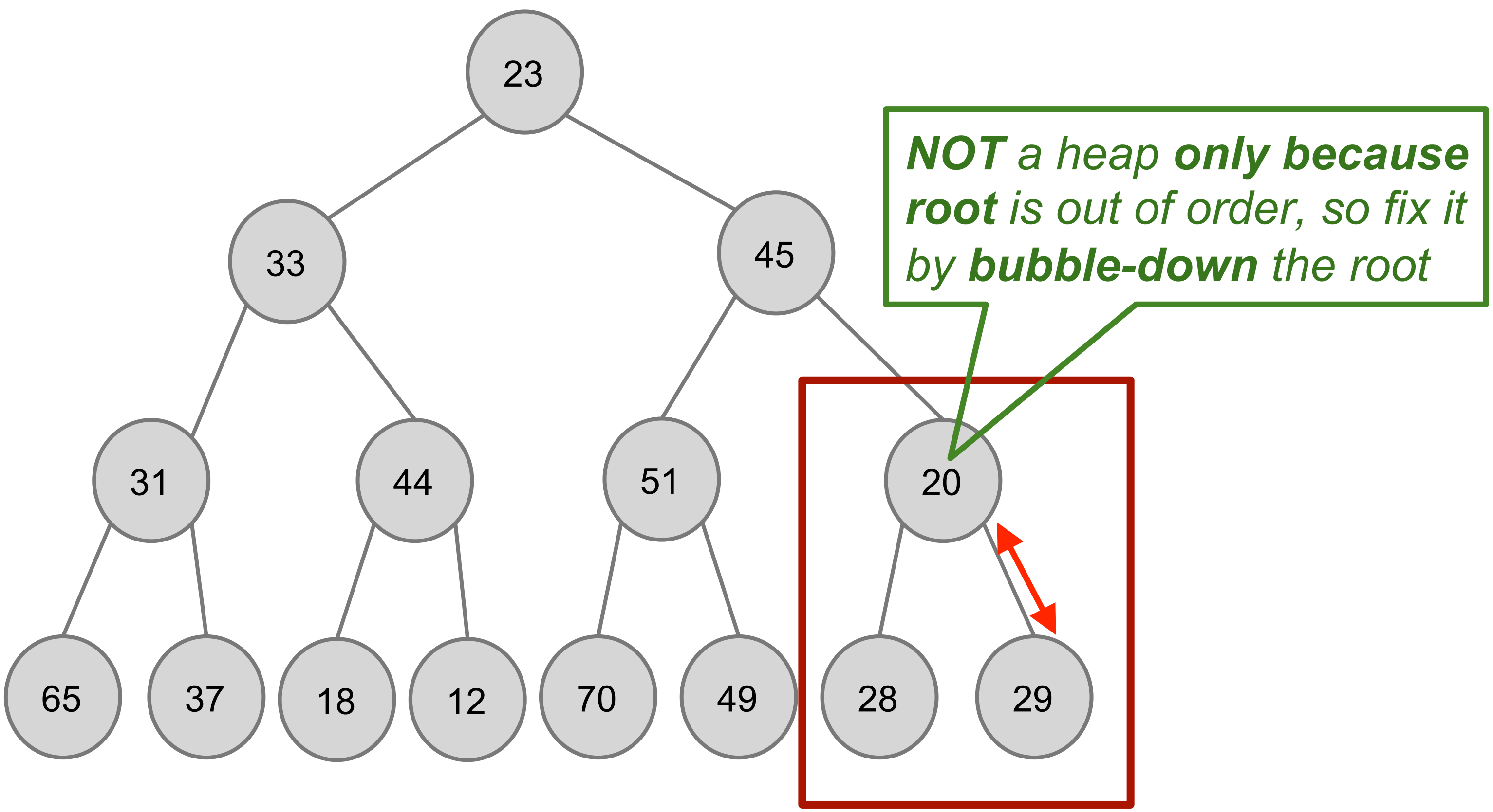
Idea #2

Fix heap order, from bottom up.



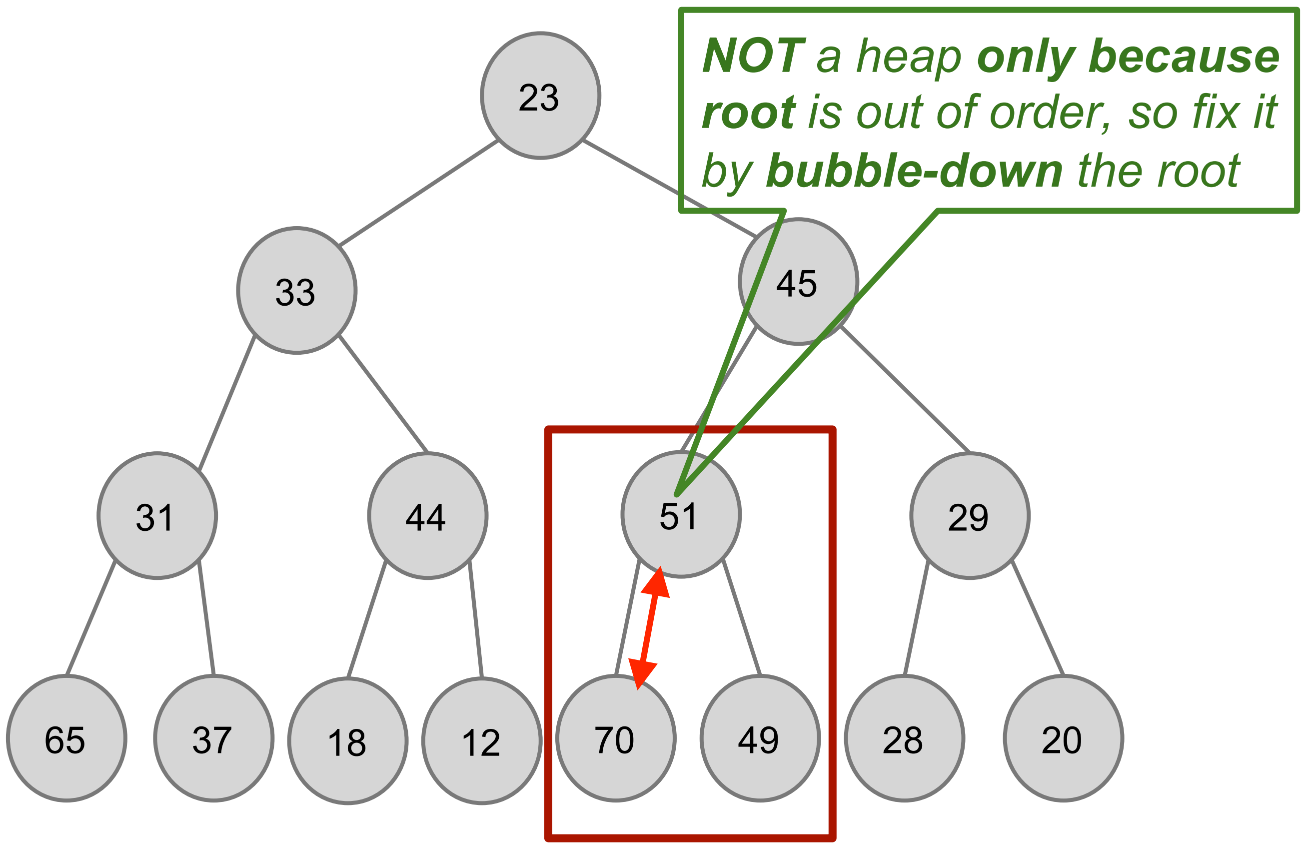
Idea #2

Adjust heap order, from bottom up.



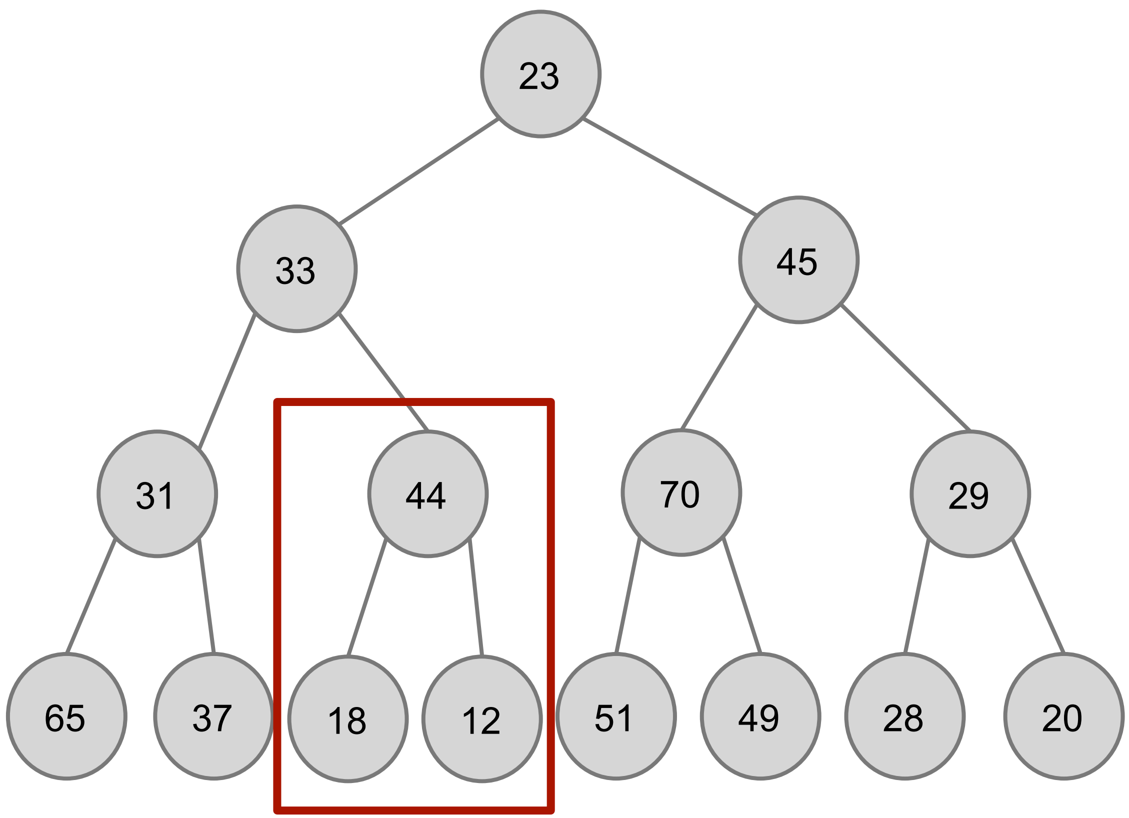
Idea #2

Adjust heap order, from bottom up.



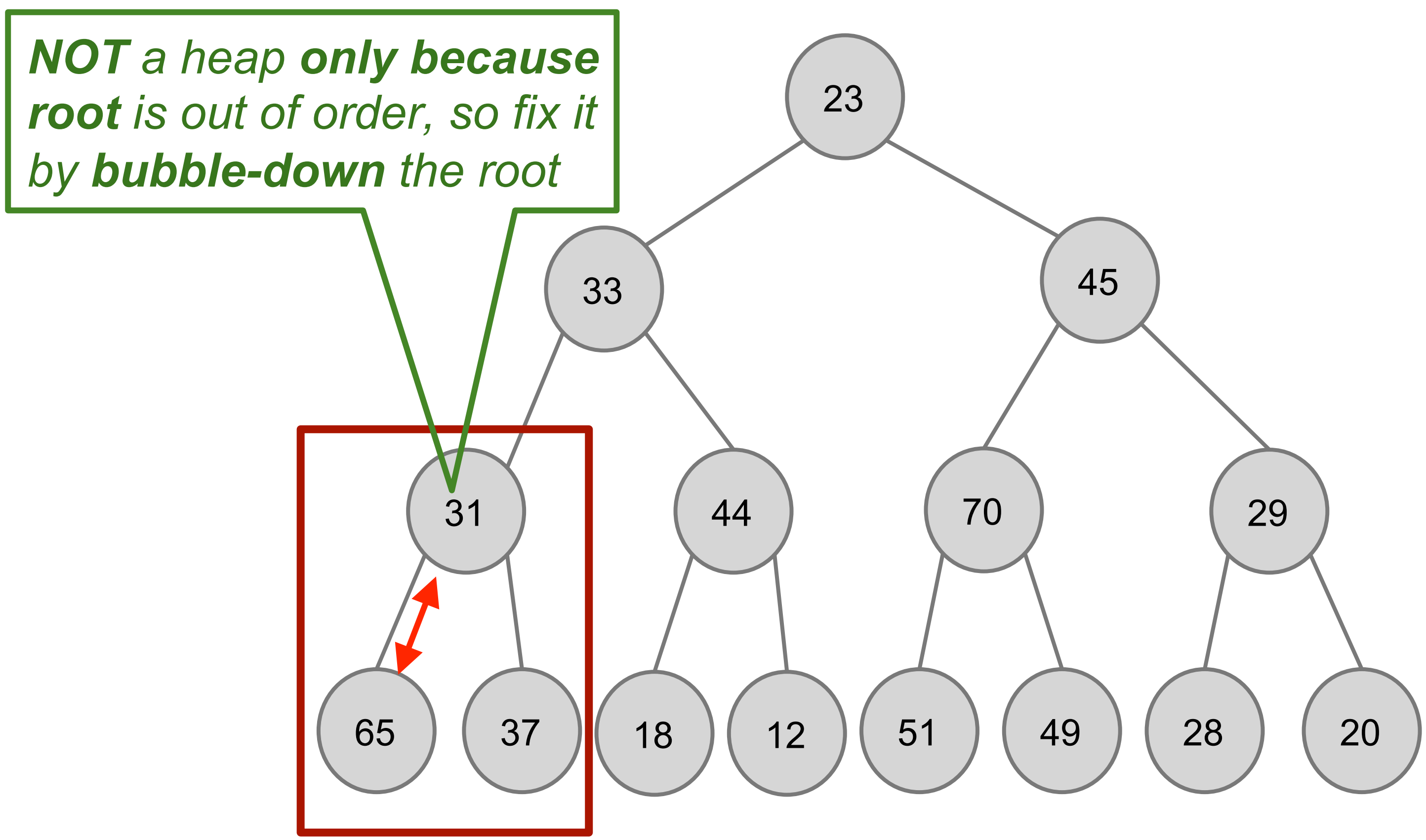
Idea #2

Adjust heap order, from bottom up.



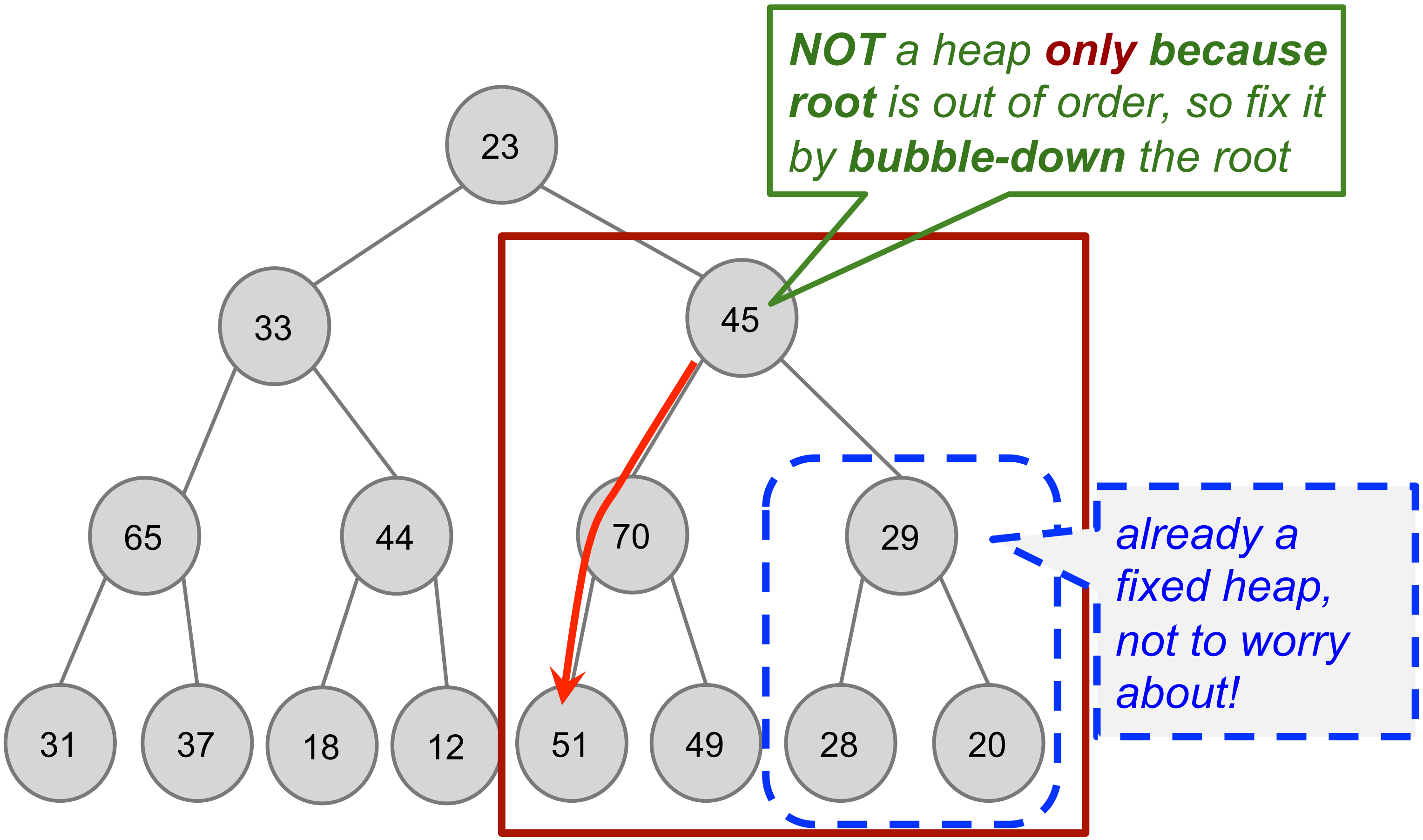
Idea #2

Adjust heap order, from bottom up.



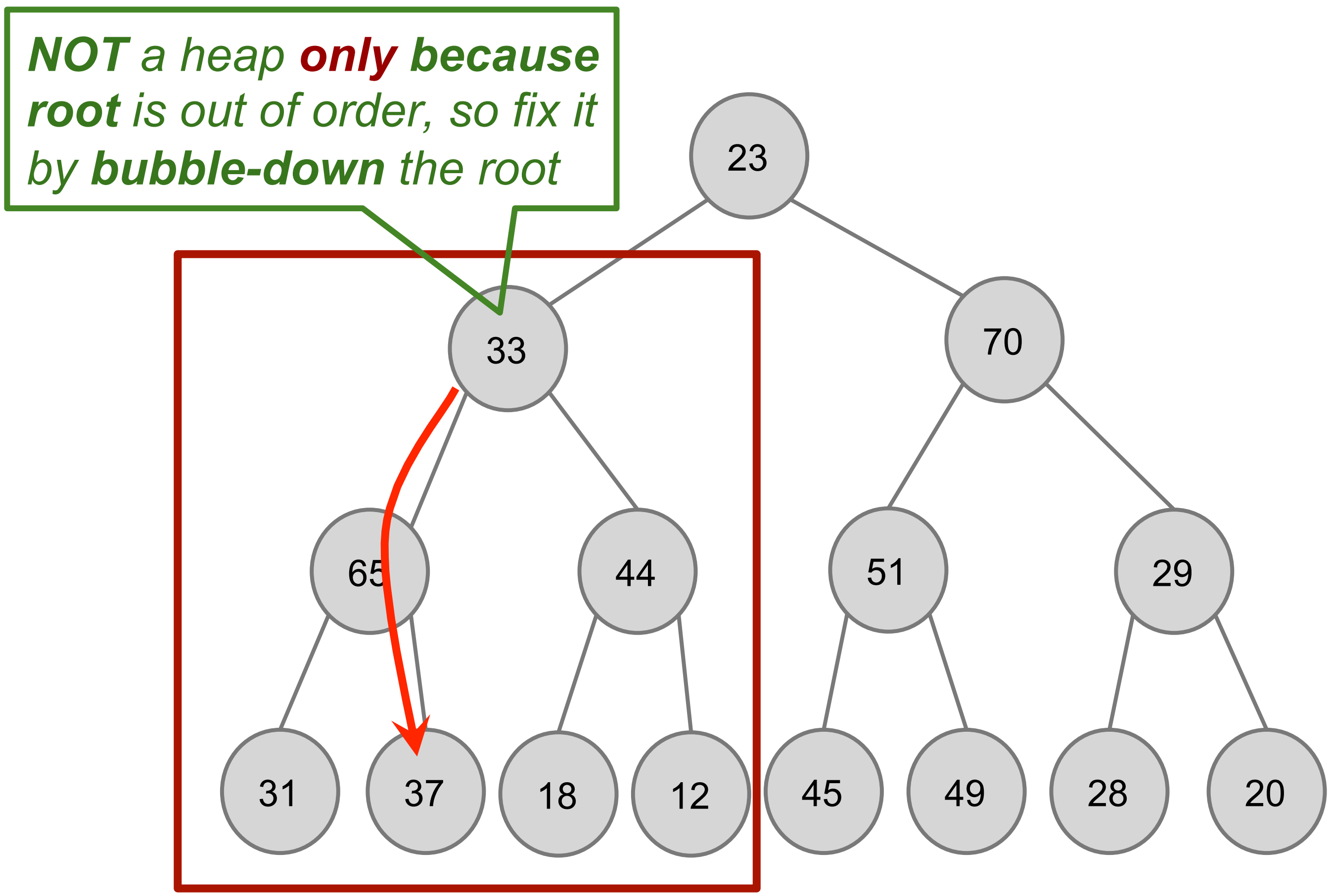
Idea #2

Adjust heap order, from bottom up.



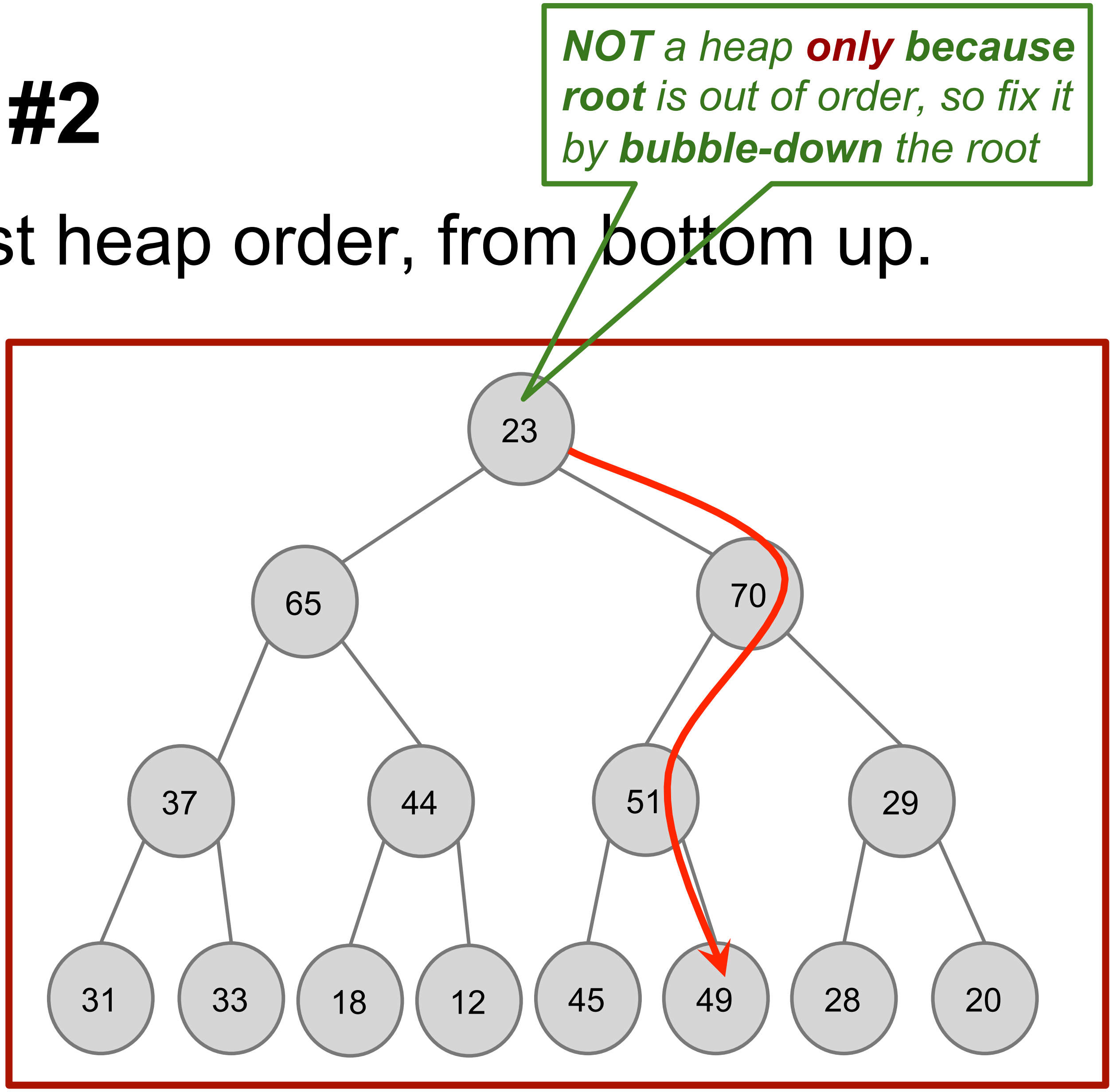
Idea #2

Adjust heap order, from bottom up.



Idea #2

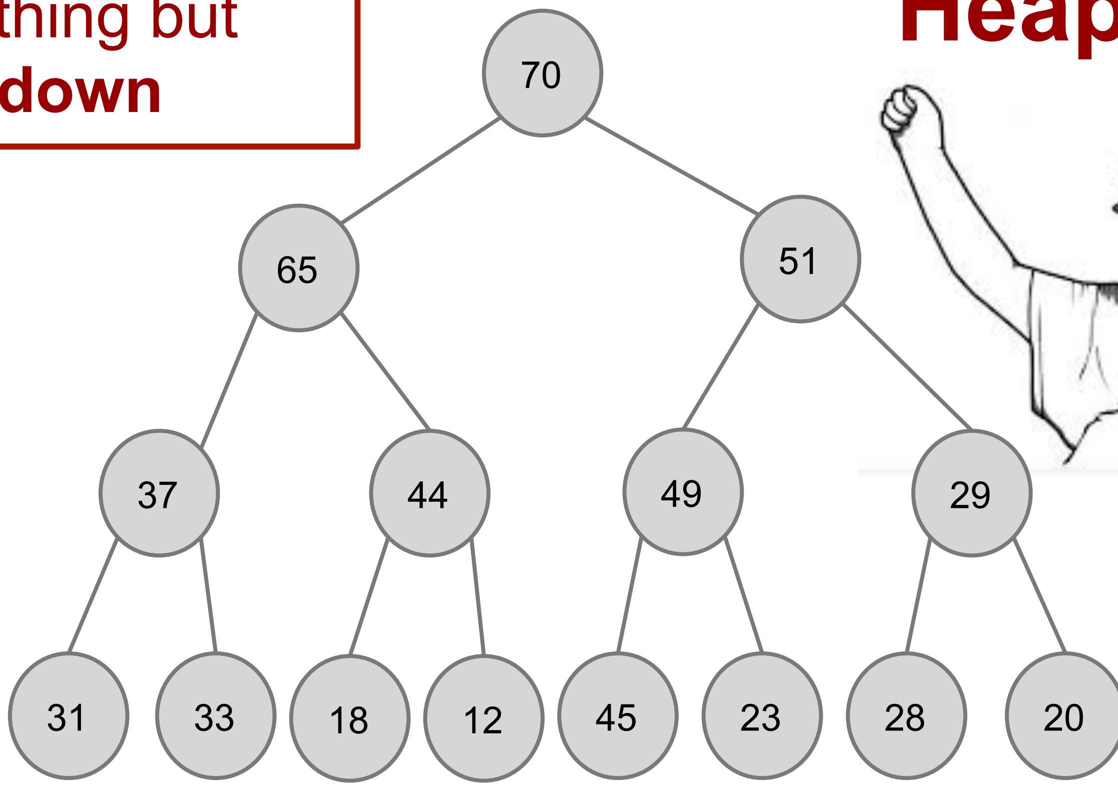
Adjust heap order, from bottom up.



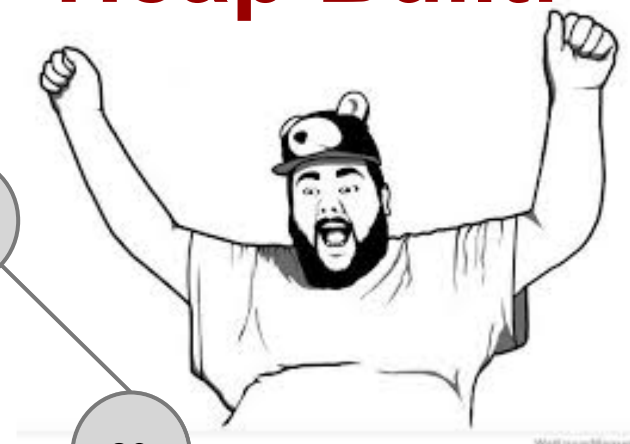
Idea #2

Adjust heap order, from bottom up.

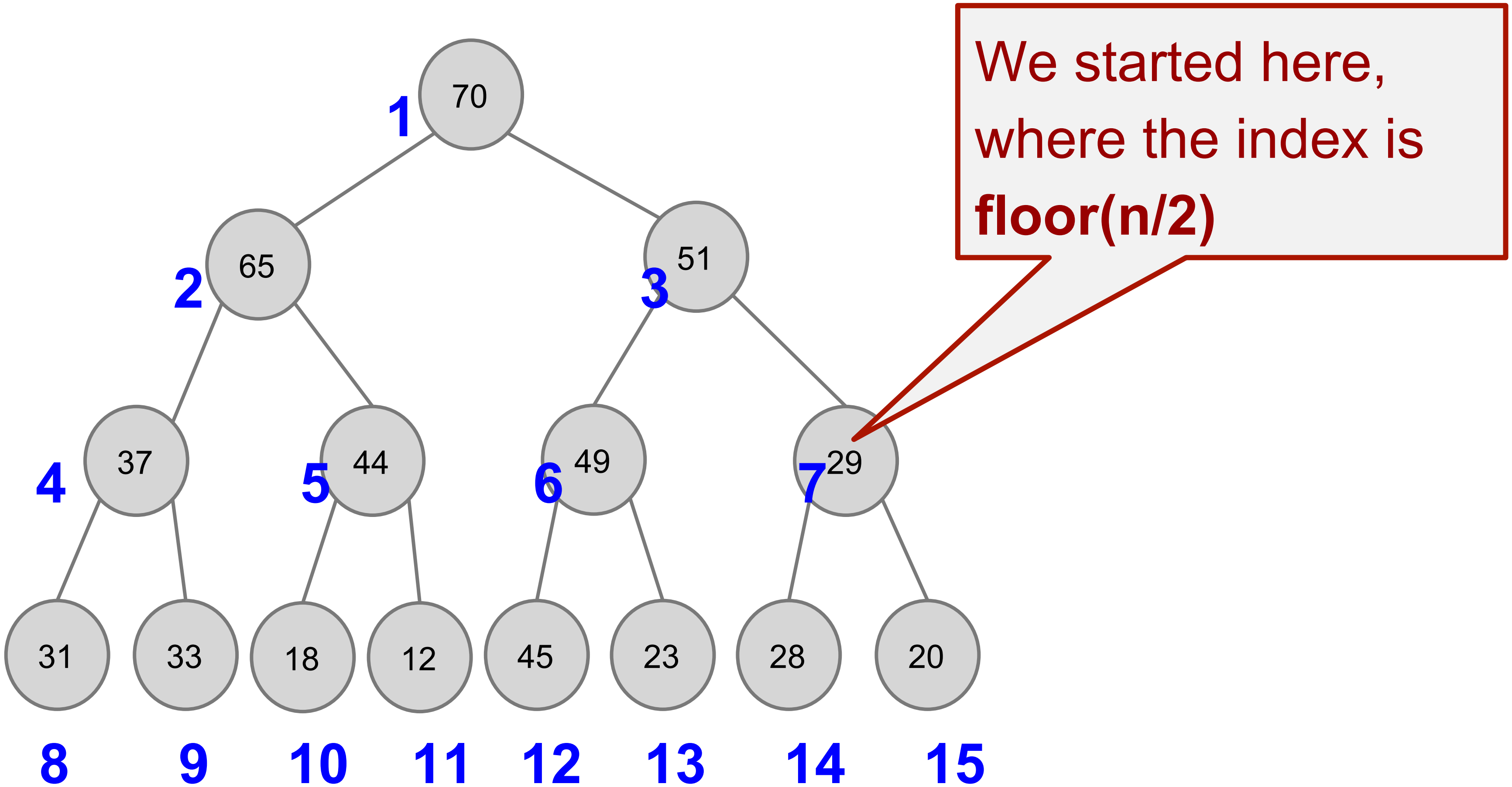
We did nothing but **bubbling-down**



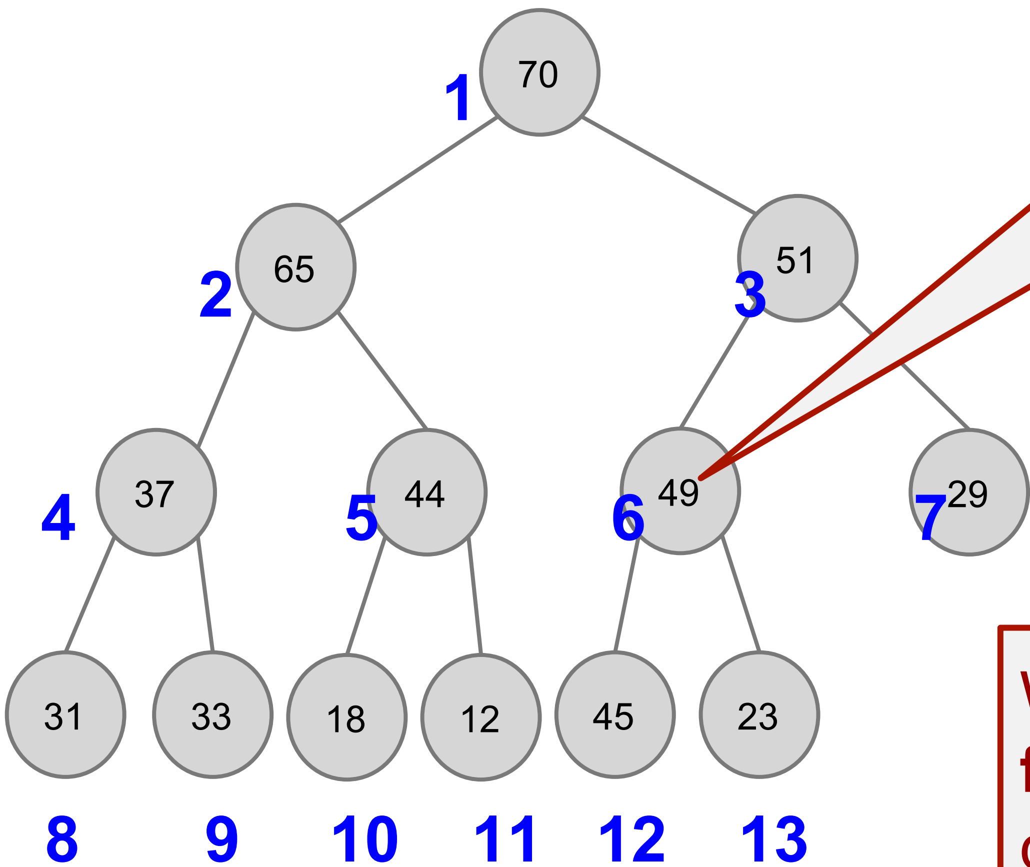
Heap Built!



Idea #2: The starting index



Idea #2: The starting index



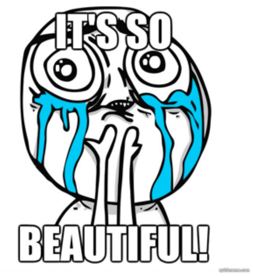
Even the bottom level is not fully filled, we still start from **floor(n/2)**

We **always** start from **floor(n/2)**, and go down to **1**.

Idea #2: Pseudo-code!

```
BuildMaxHeap(A):

   for i ← floor(n/2) downto 1:
      BubbleDown(A, i)
```



Advantages of Idea #2:

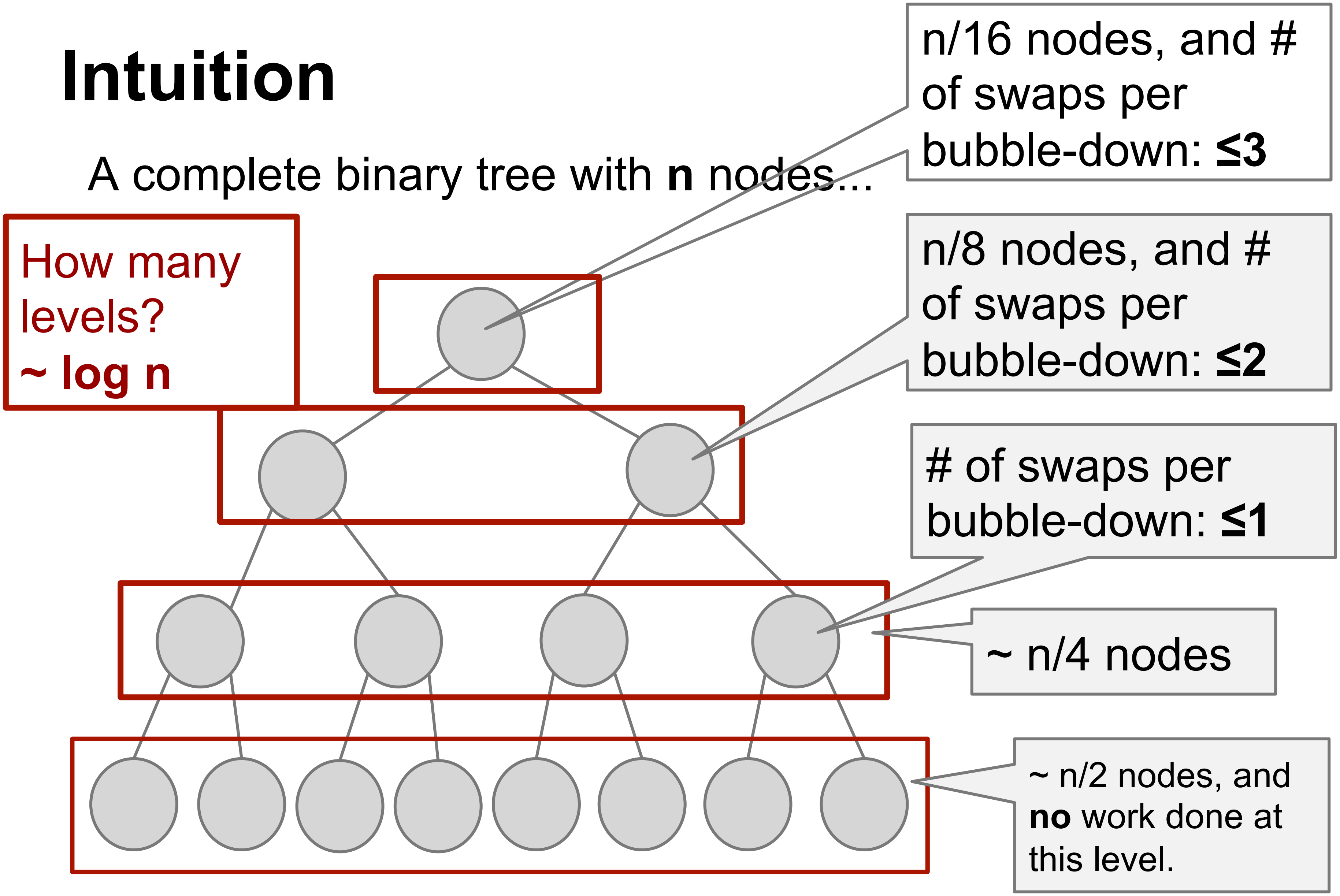
- ➔ It's in-place, no need for extra array (we did nothing but bubble-down, which is basically swappings).
- ➔ It's worst-case running time is **O(n)**, instead of O(n log n) of Idea #1.

Why?

Analysis:
Worst-case running time of
BuildMaxHeap(A)

Intuition

A complete binary tree with **n** nodes...



So, total number of swaps

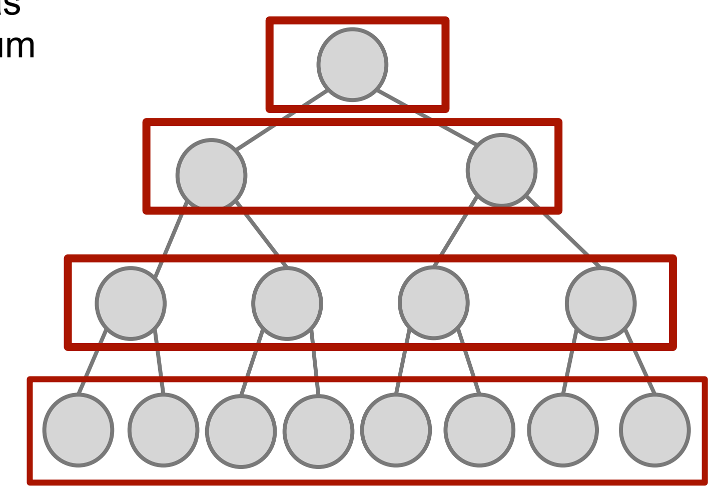
$$T(n) = 1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + 3 \cdot \frac{n}{16} + \dots$$

$$= \sum_{i=1}^{\log n} i \cdot \frac{n}{2^{i+1}} \quad \leq \sum_{i=1}^{+\infty} i \cdot \frac{n}{2^{i+1}}$$

$$= n \boxed{\sum_{i=1}^{+\infty} \frac{i}{2^{i+1}}} \; =1$$

same trick as
Week 1's sum

$$= n$$



$\Sigma_{i=0,1,..}$ $i/2^i = \Sigma_{k=0,1,..} k\ x^k$,
when $x=1/2$

$\Sigma_{k=0,1,..} k\ x^k = x/(1-x)$^2

So $\Sigma_{i=0,1,..} i/2^i = 1/2/(1-1/2)^2 = 2$


BUILD MAX HEAP
YOU CAN DO IN LINEAR TIME
imgflip.com

Summary

HeapSort(A):

- ➔Sort a heap-ordered array in-place
- ➔O(n log n) worst-case running time

BuildMaxHeap(A):

- ➔Convert an unsorted array into a heap, in-place
- ➔Fix heap property from bottom up, do bubbling down on each sub-root
- ➔O(n) worst-case running time

Algorithm visualizer

http://visualgo.net/heap.html

Next week

➔ADT: Dictionary

➔Data structure: Binary Search Tree