CSC263 Winter 2020

Priority Queues

Week 2

Announcements

Probability Worksheet (ungraded)

Assignment 1 out → Due Jan 27

Make Groups on MarkUs

https://mcs.utm.utoronto.ca/~263

Today

ADT Priority Queue

Data structure Heap

Use it to implement priority queues

...Also useful for other things, e.g. sorting

An ADT We already Know

First come first serve

Queue

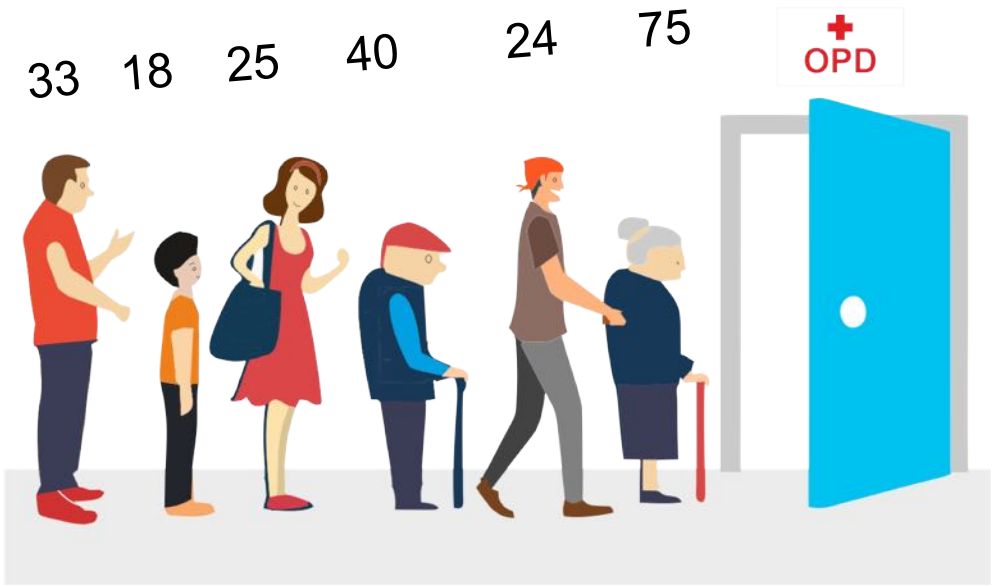
A collection of elements

Supported operations

- ❑ `Enqueue(Q, x)`
- ❑ `Dequeue(Q)`
- ❑ `PeekFront(Q)`

A 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

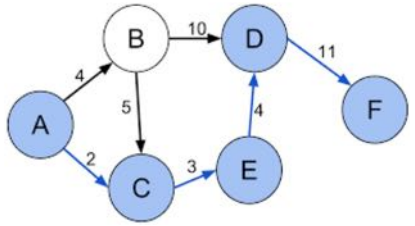
Job scheduling in an operating system

- Processes have different priorities (Normal, high...)

Bandwidth management in a router

- Delay sensitive traffic has higher priority

Several graph algorithms



- Find minimum spanning tree of a graph, find shortest path

Now, let's implement ADT (Max)-Priority Queue

40 -> 33 -> 18 -> 75 -> 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 (Θ(n)), then delete it (O(1) if double linked) and return it, so overall Θ(n).

75 -> 40 -> 33 -> 25 -> 24 -> 18

Use a linked list sorted in reverse

`Max(Q)`

- Just return the head of the list, $\Theta(1)$

`ExtractMax(Q)`

- Just delete and return the head, $\Theta(1)$

`Insert(Q,x)`

- 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.

Θ(1) is good,
but Θ(n) is bad...

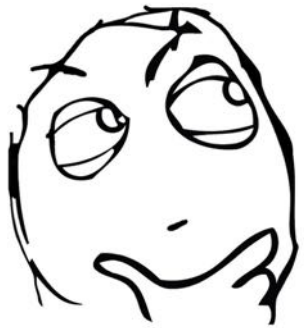
unsorted linked list

sorted linked list

...

Can we link these elements in a smarter way, so that we never need to do Θ(n)?

	unsorted list	sorted list
`Insert(Q,x)`	Θ(1)	Θ(n)
`Max(Q)`	Θ(n)	Θ(1)
`ExtractMax(Q)`	Θ(n)	Θ(1)
`IncreasePriority(Q,x,k)`	Θ(1)	Θ(n)



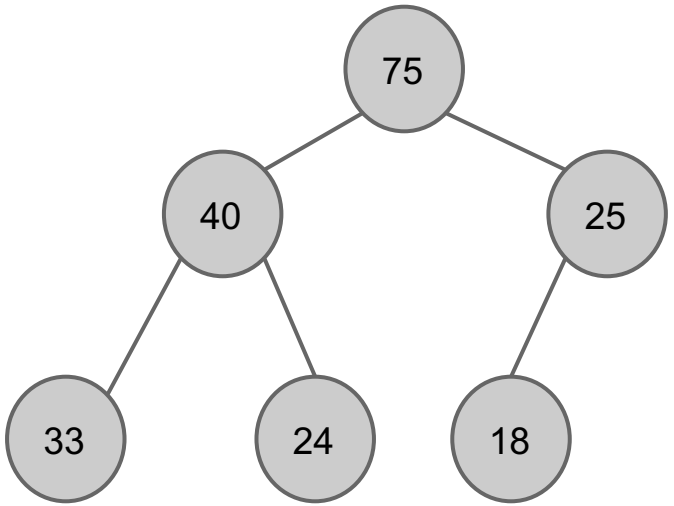
Yes, we can!

Worst case running times

	unsorted list	sorted list	**Heap**
`Insert(Q,x)`	Θ(1)	Θ(n)	Θ(log n)
`Max(Q)`	Θ(n)	Θ(1)	Θ(1)
`ExtractMax(Q)`	Θ(n)	Θ(1)	Θ(log n)
`IncreasePriority(Q,x,k)`	Θ(1)	Θ(n)	Θ(log n)

Some operations are worse, from Θ(1) to Θ(log n),
but that's a small price to pay for avoiding Θ(n) costs

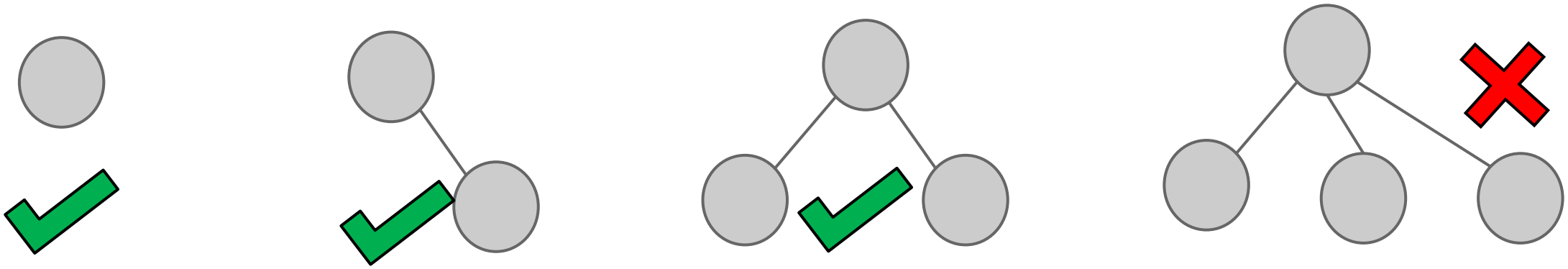
Binary Max-Heap



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

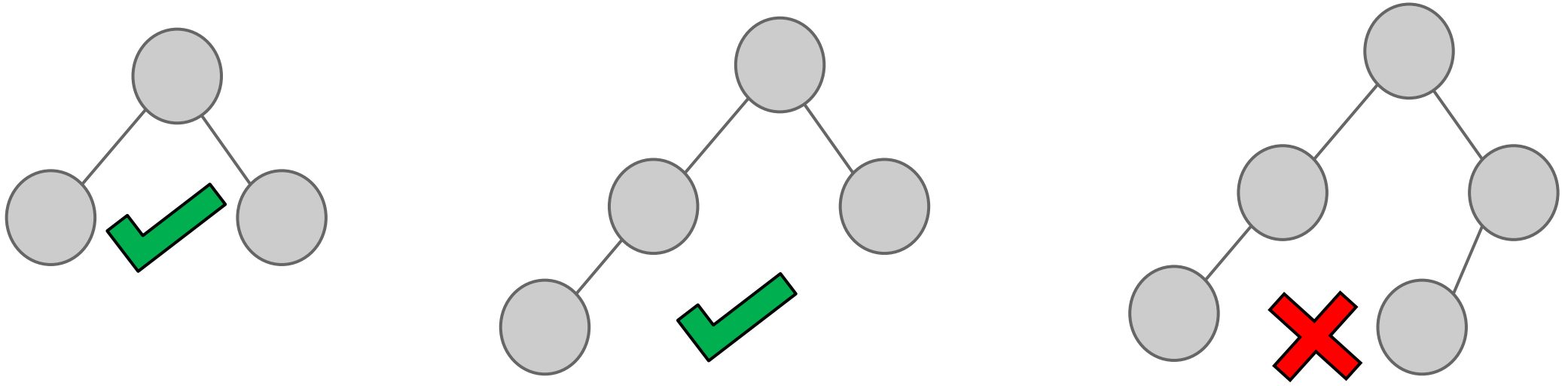
It’s a binary tree

Each node has at most 2 children



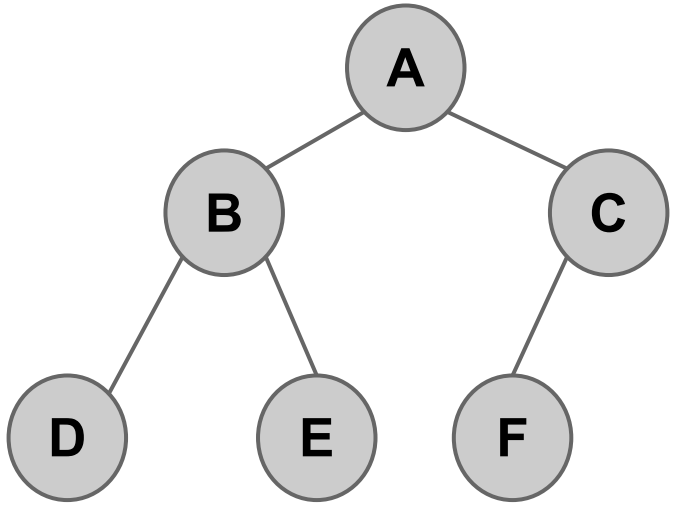
It's a nearly-complete binary tree

Each level is completely filled, except maybe 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 or left/right child.



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

```
Left(i) = 2i
Right(i) = 2i + 1 (if exists)
Parent(i) = floor(i/2)
```

Assume index starts at 1

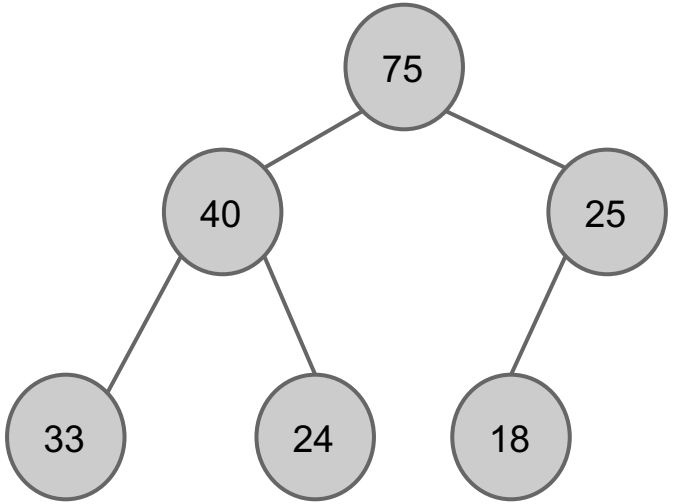
Why is it important to be a nearly-complete binary tree?

Another reason:

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

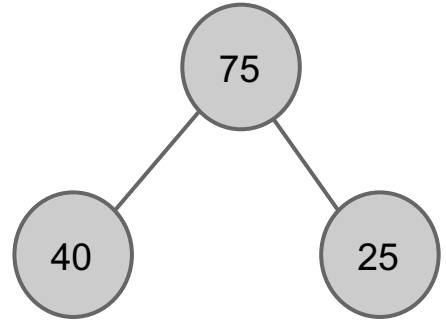
This is essentially why those operations would have Θ(log n) worst-case running time.

Binary Max-Heap

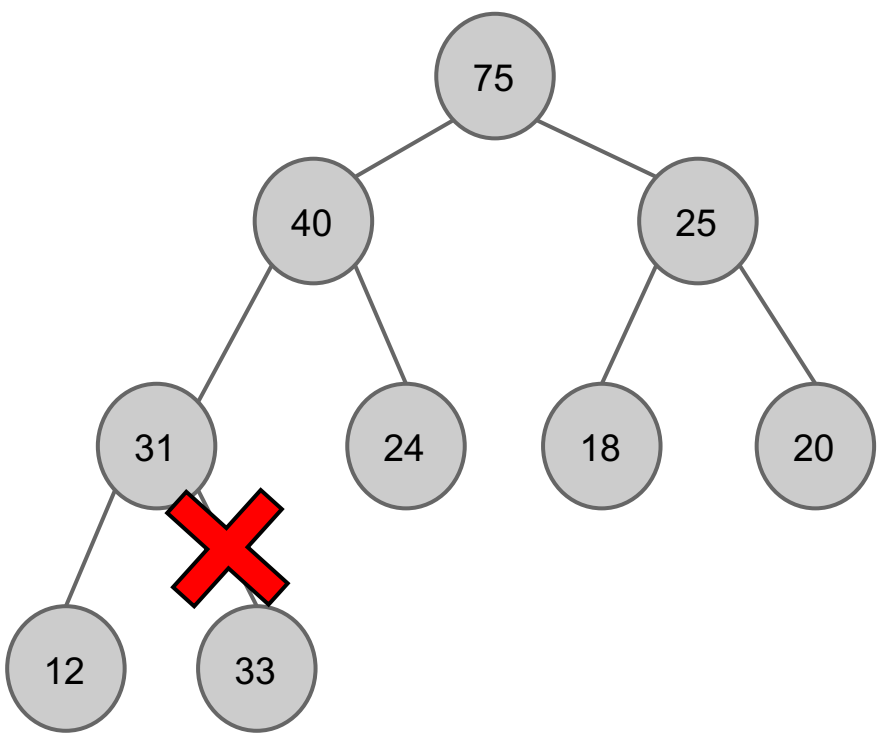


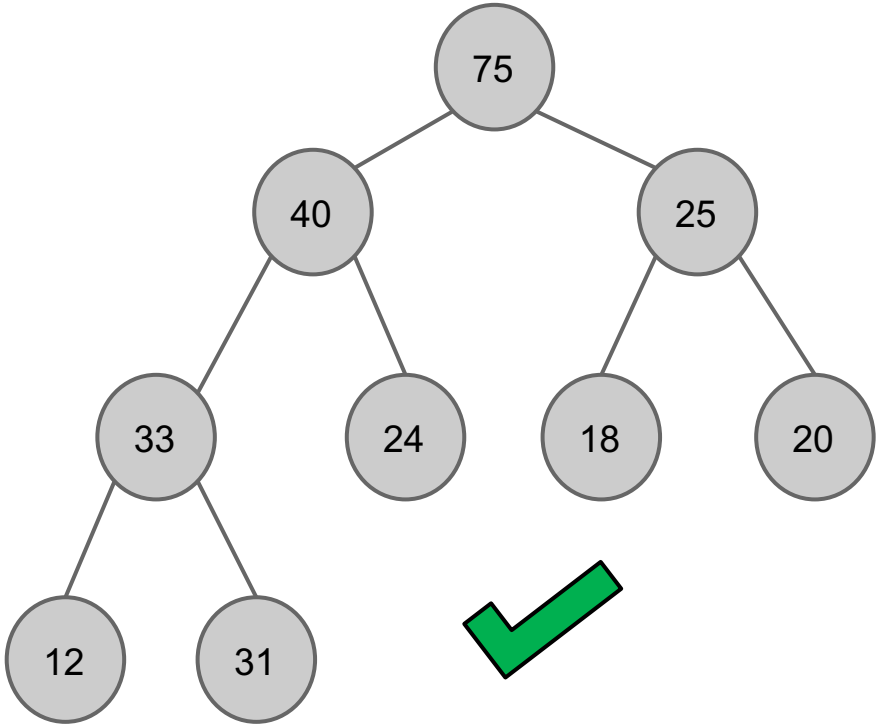
A binary max-heap is a **nearly-complete binary** tree that has the **max-heap property**.

The max-heap Property

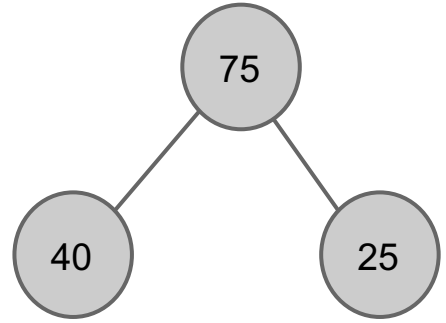


Every node has a key (priority) greater than or equal to keys of its immediate children.



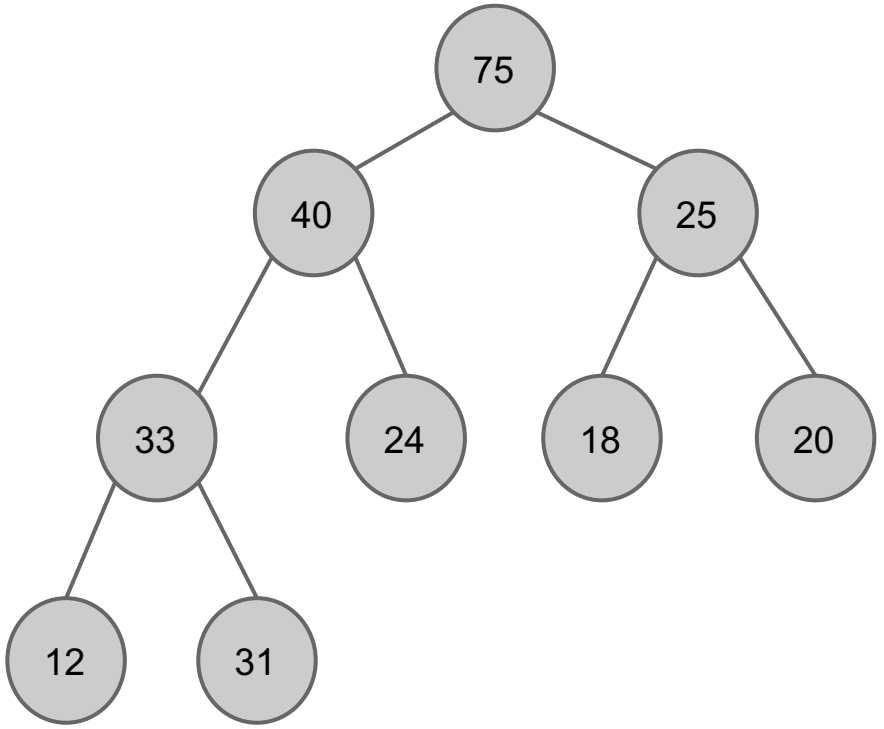


The max-heap Property



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.,

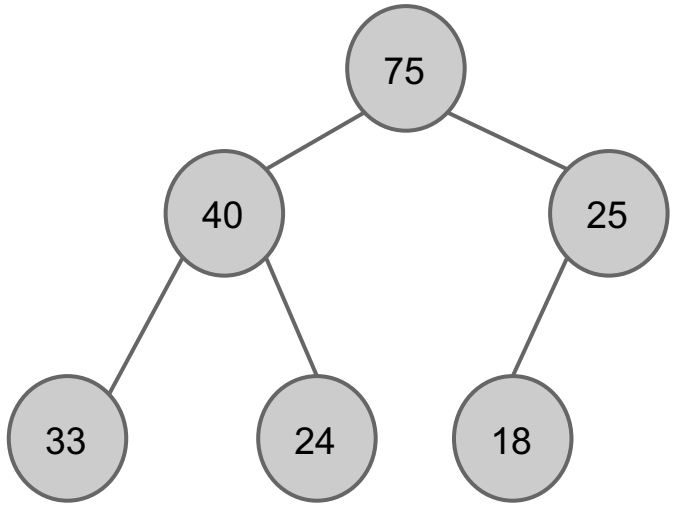
just return Q[1]

(index starts from 1)

worst case $\Theta(1)$

Q

75	40	25	33	24	18



`Insert(Q, x)`

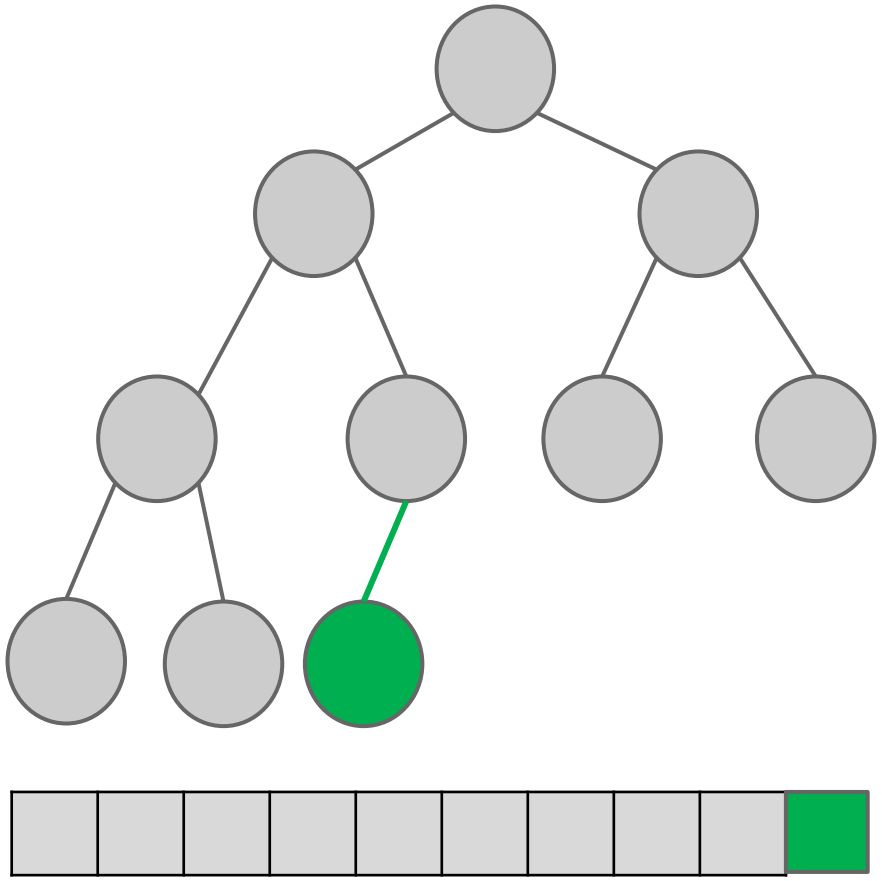
Insert node x into heap Q,
in O(logn) time

`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 **nearly complete** binary tree.



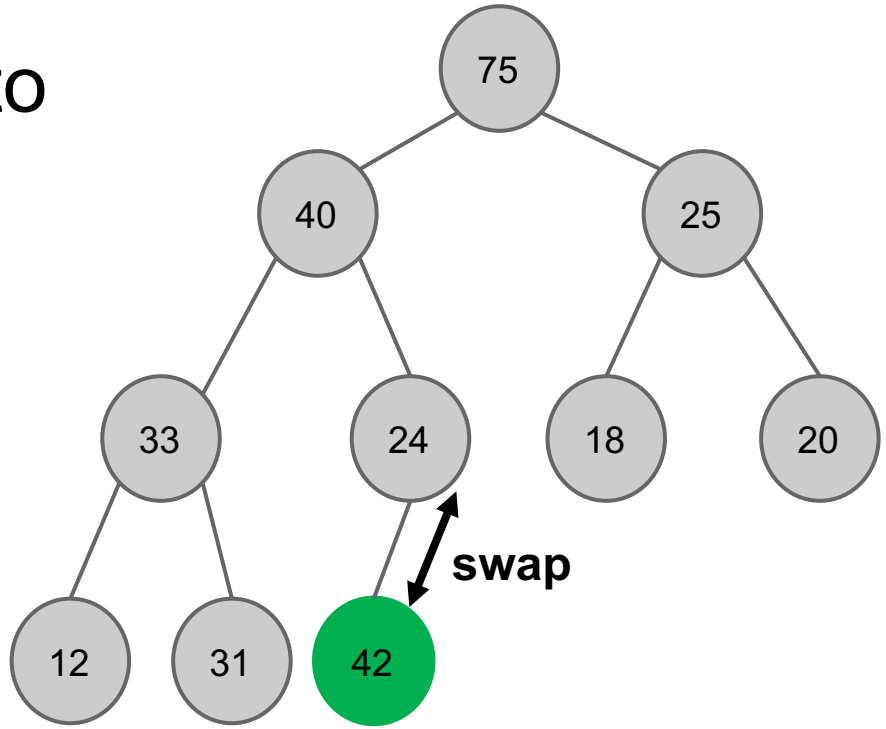
Increment heap size

`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.

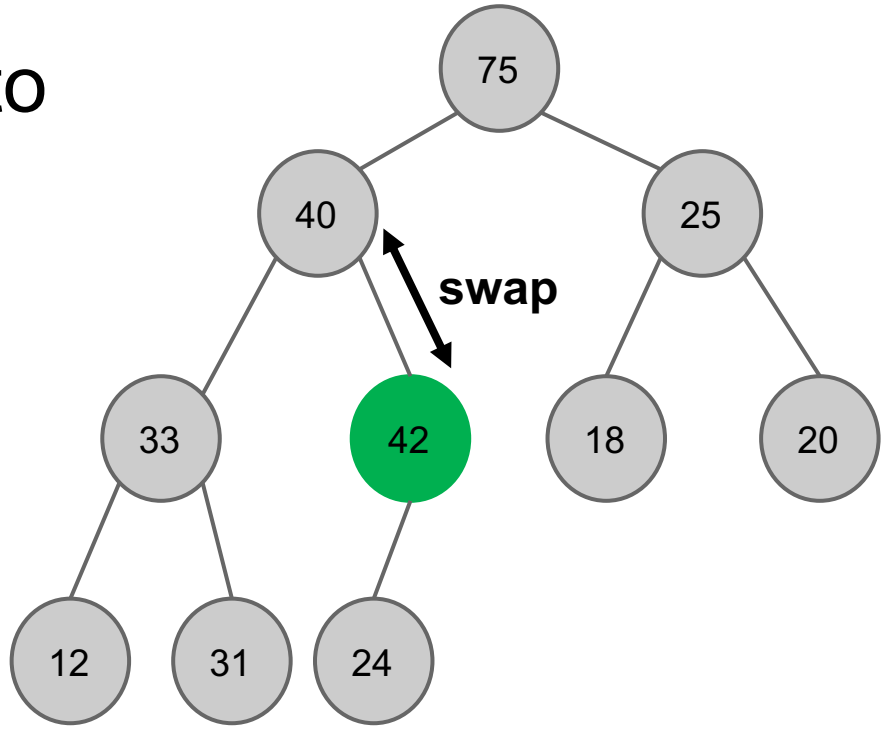


`Insert(Q,x)`
Insert a node to a heap

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.



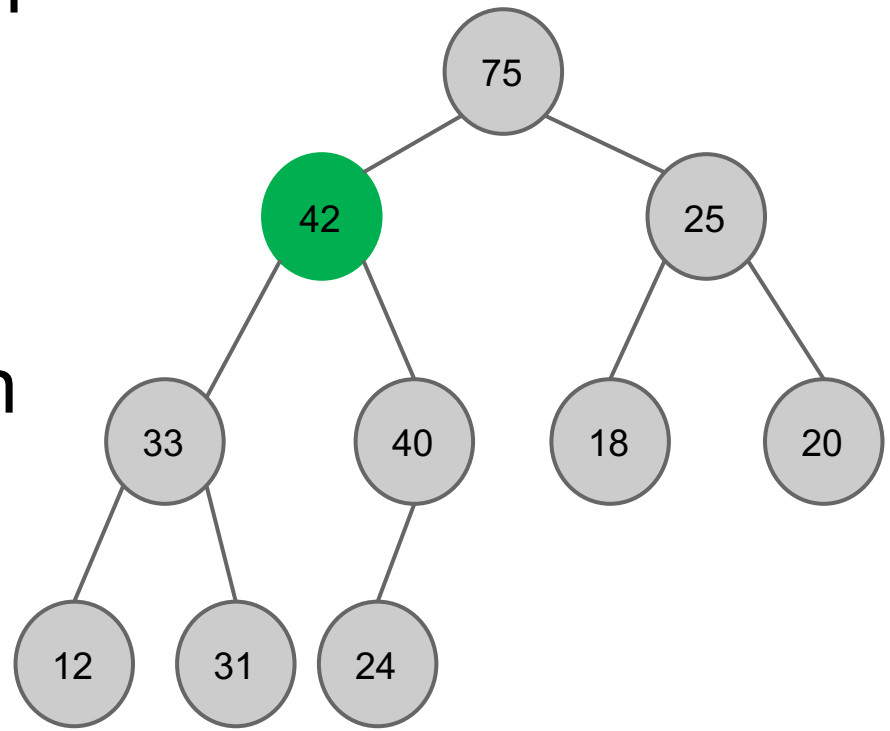
`Insert(Q,x)`
Insert a node to a heap

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)`

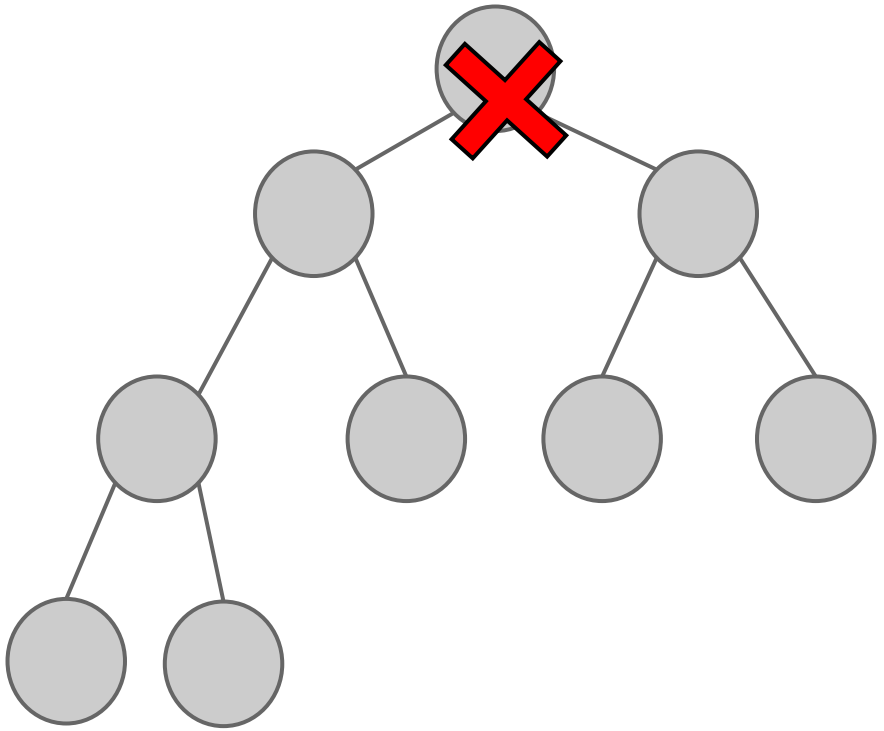
Delete and return the largest key in Q,
in O(logn) time

`ExtractMax(Q)`
Delete and return the maximum element

First thing to note:

Which spot to remove?

The root?



NO!

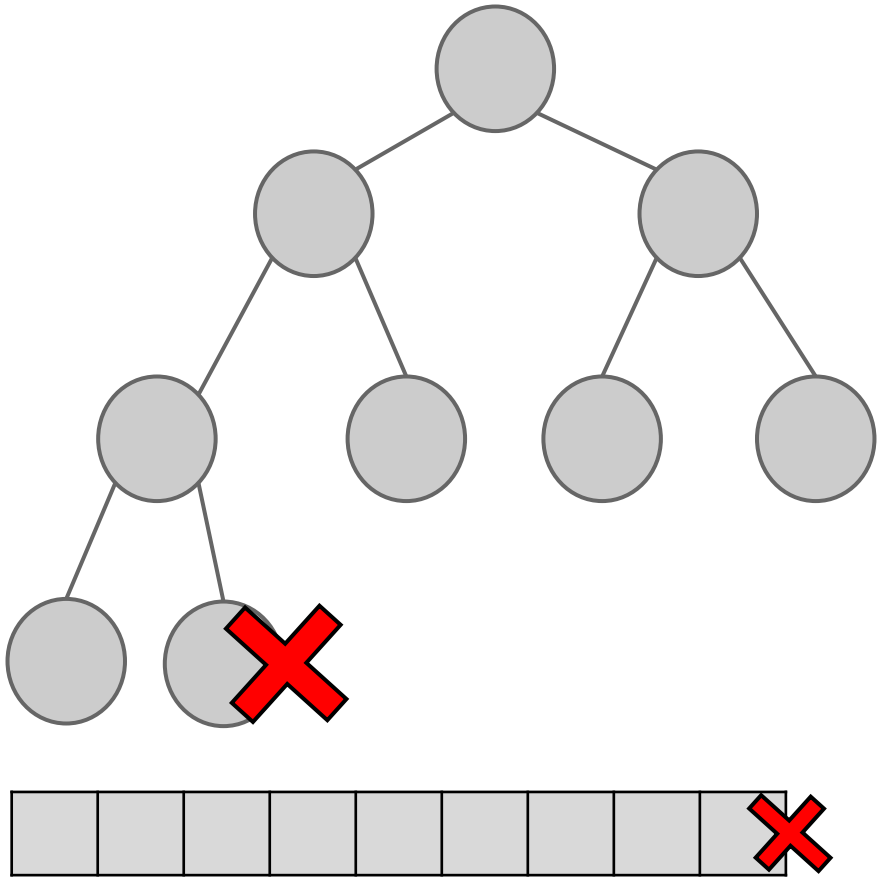
Will break into 2 heaps!

`ExtractMax(Q)`
Delete and return the maximum element

First thing to note:

Which spot to remove?

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



Decrement heap size

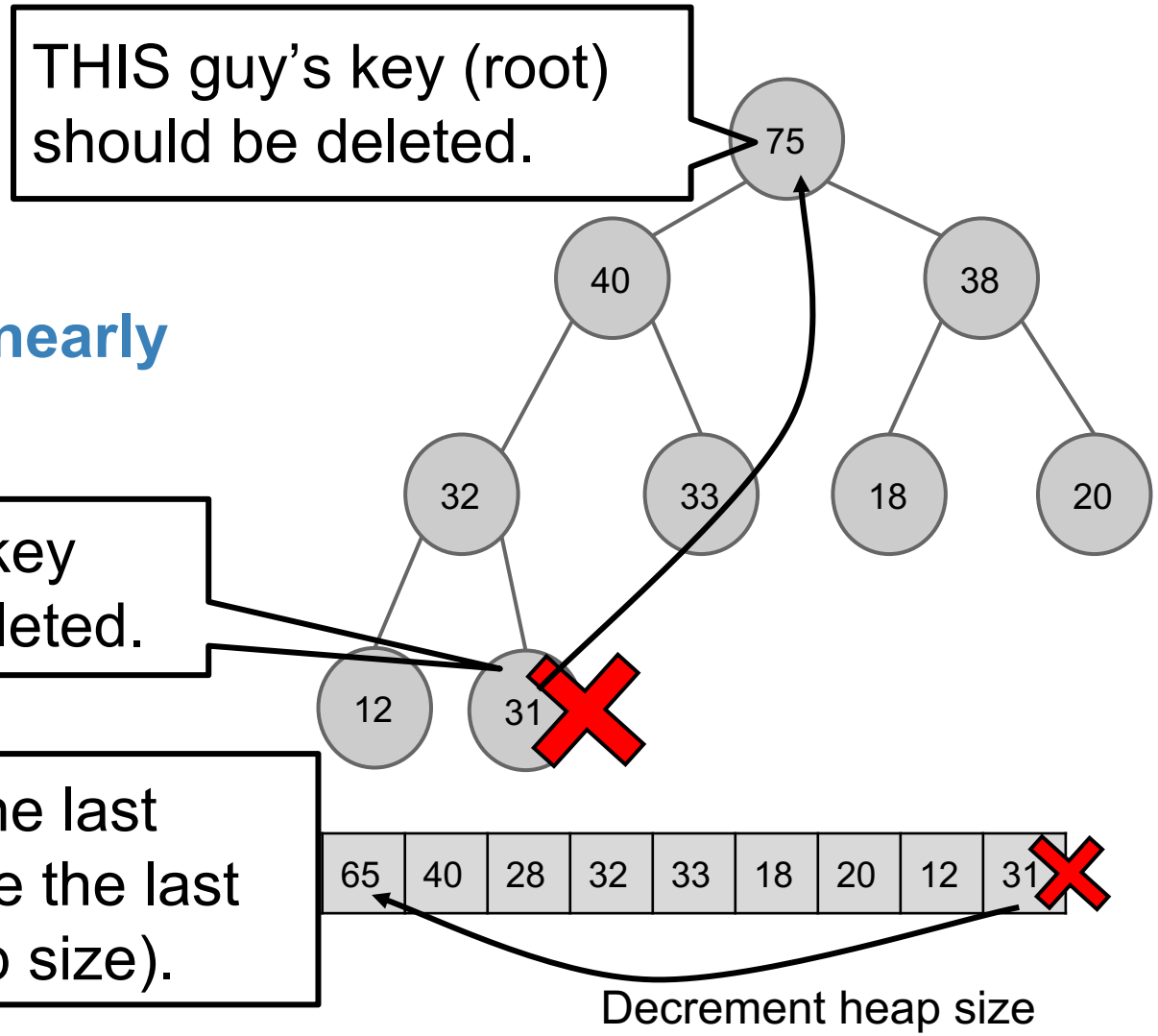
`ExtractMax(Q)`

Delete and return the maximum element

First thing to note:

Which spot to remove?

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

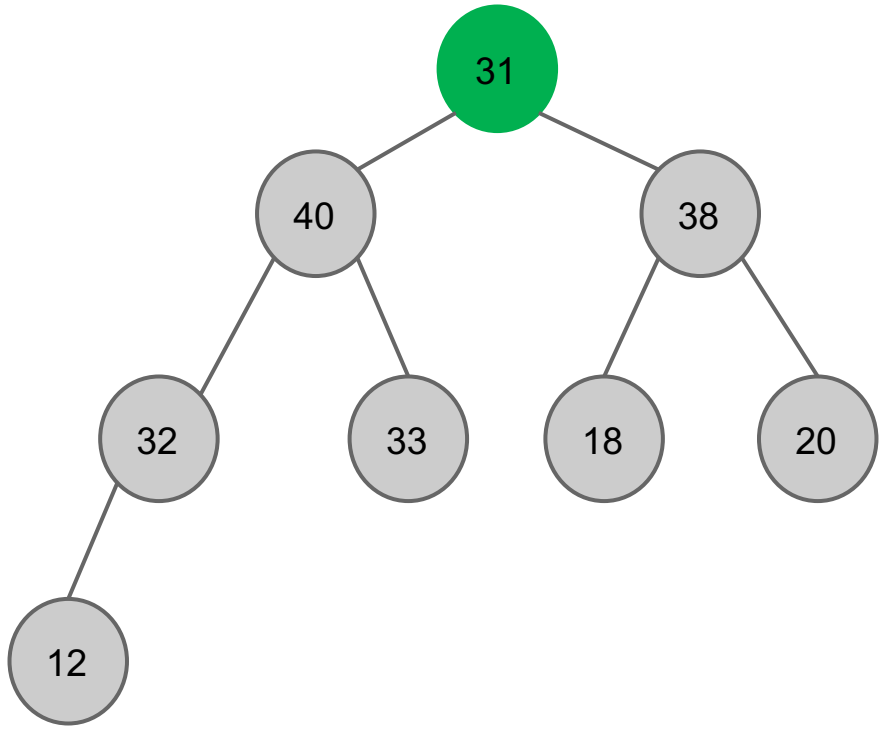


`ExtractMax(Q)`
Delete and return the maximum element

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

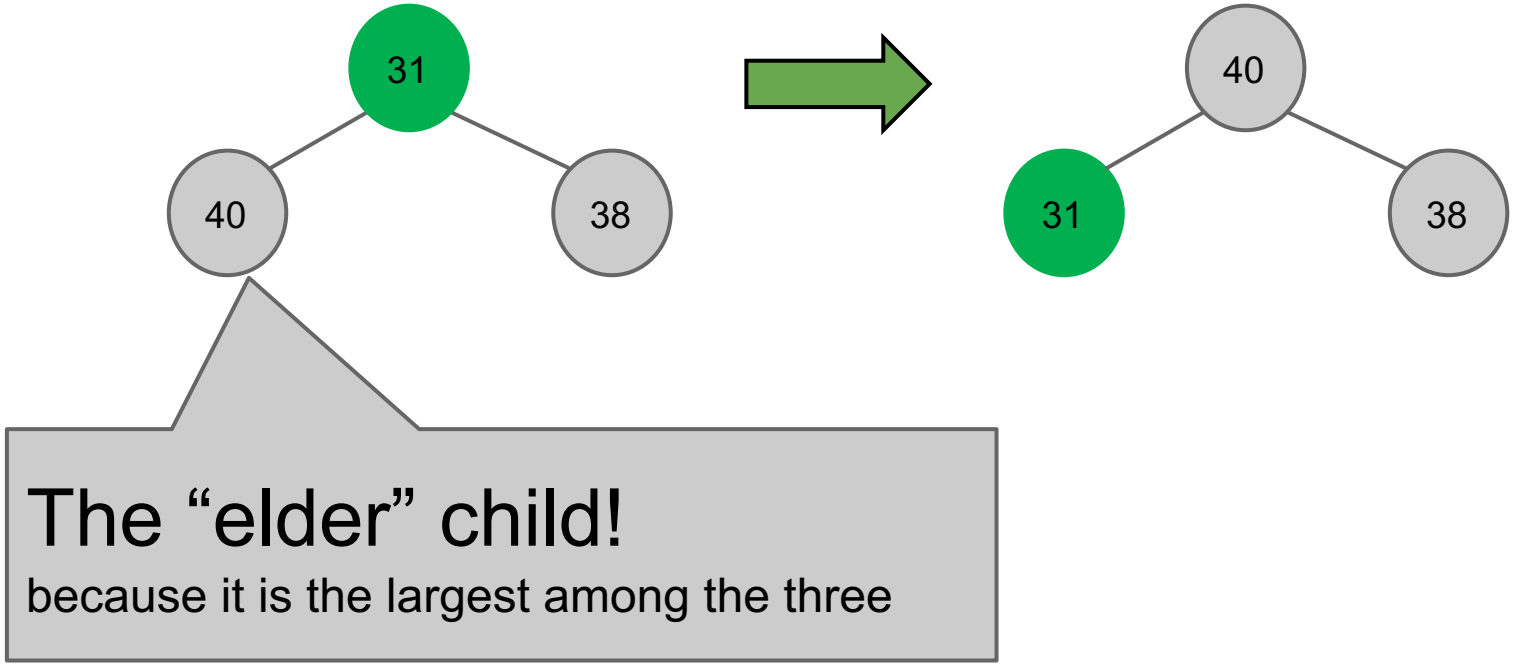
"Bubble-**down**" by swapping with

a child



Which child to swap with?

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

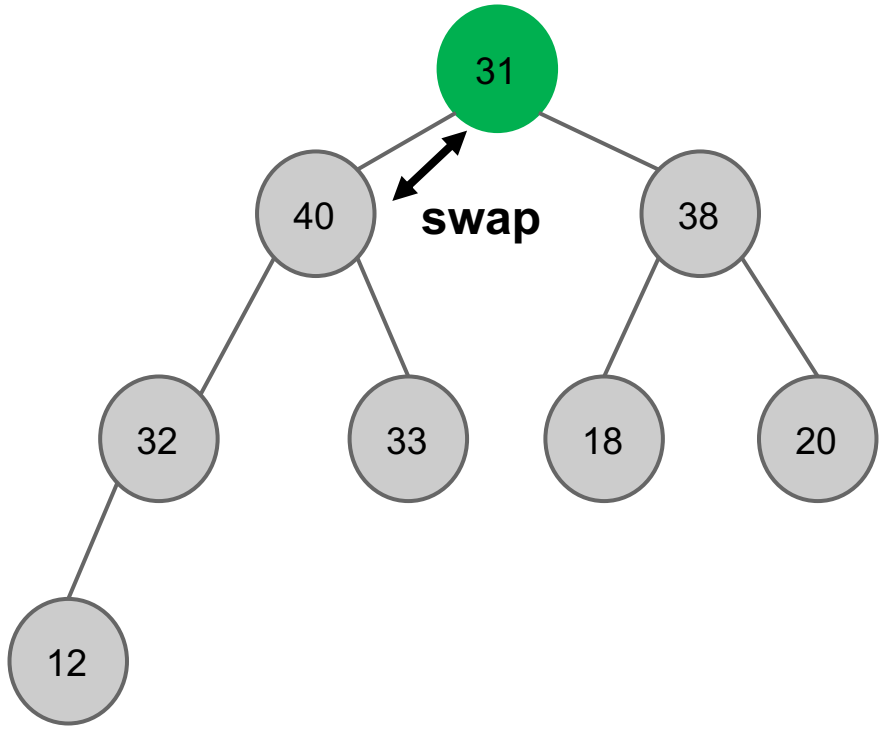


ExtractMax(Q):
Delete and return the maximum element

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

"Bubble-**down**" by swapping with

the elder child

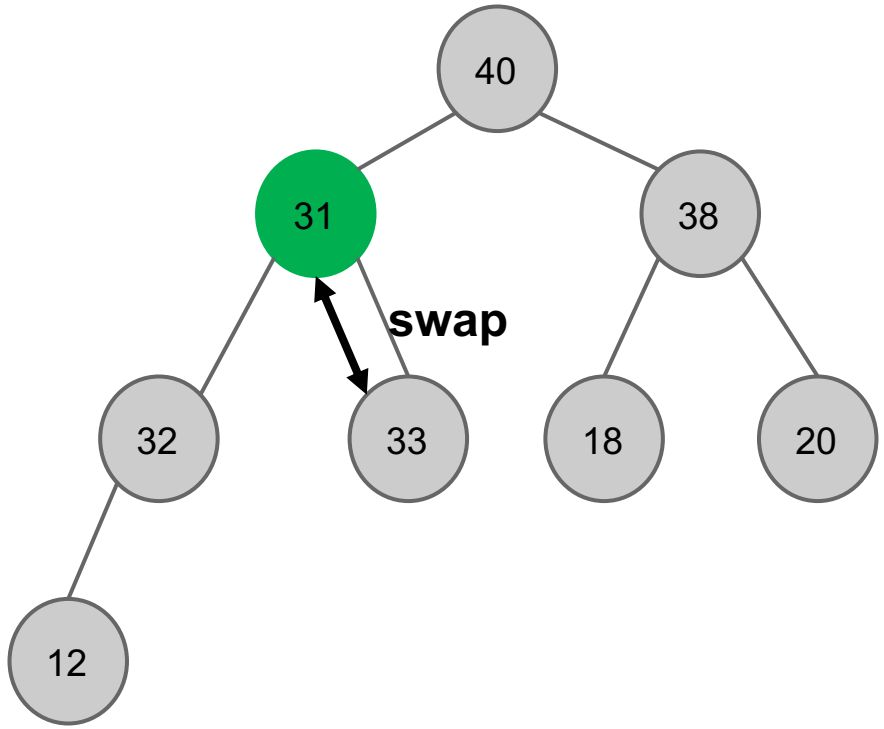


ExtractMax(Q):
Delete and return the maximum element

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

"Bubble-**down**" by swapping with...

the elder child

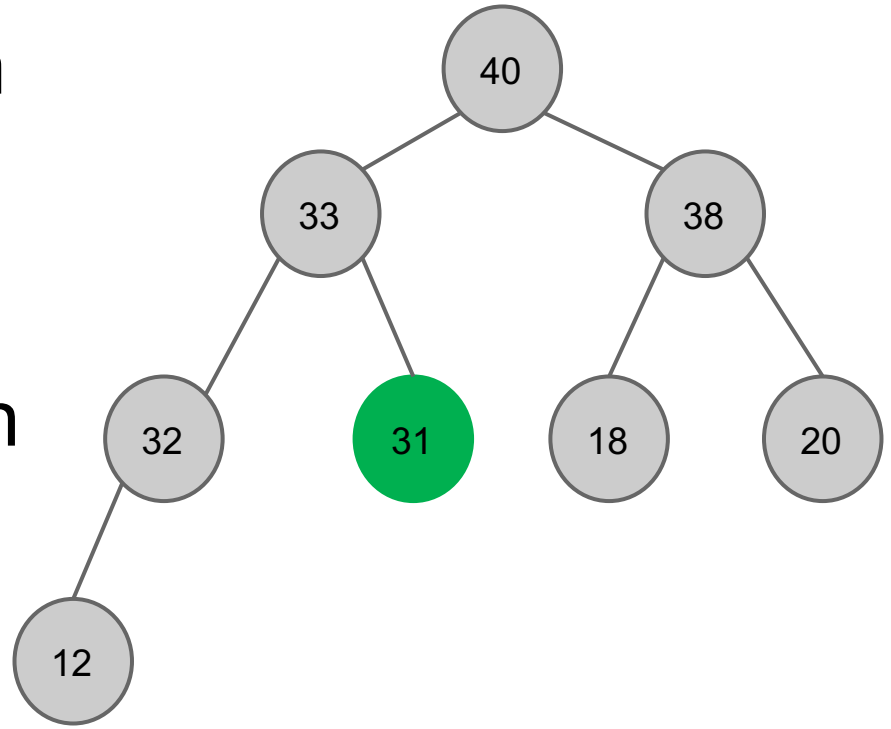


ExtractMax(Q):
Delete and return the maximum element

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 **trickle up/down**,
or **heapify up/down**, or **cascade up/down**.

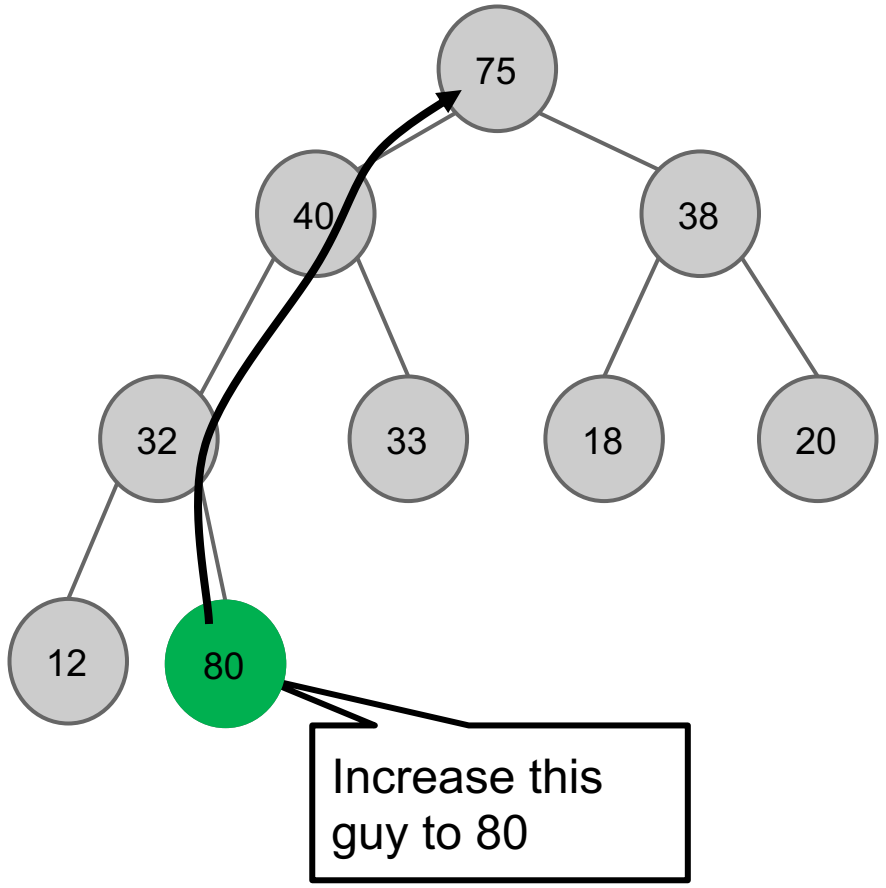
`IncreasePriority(Q, x, k)`

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...

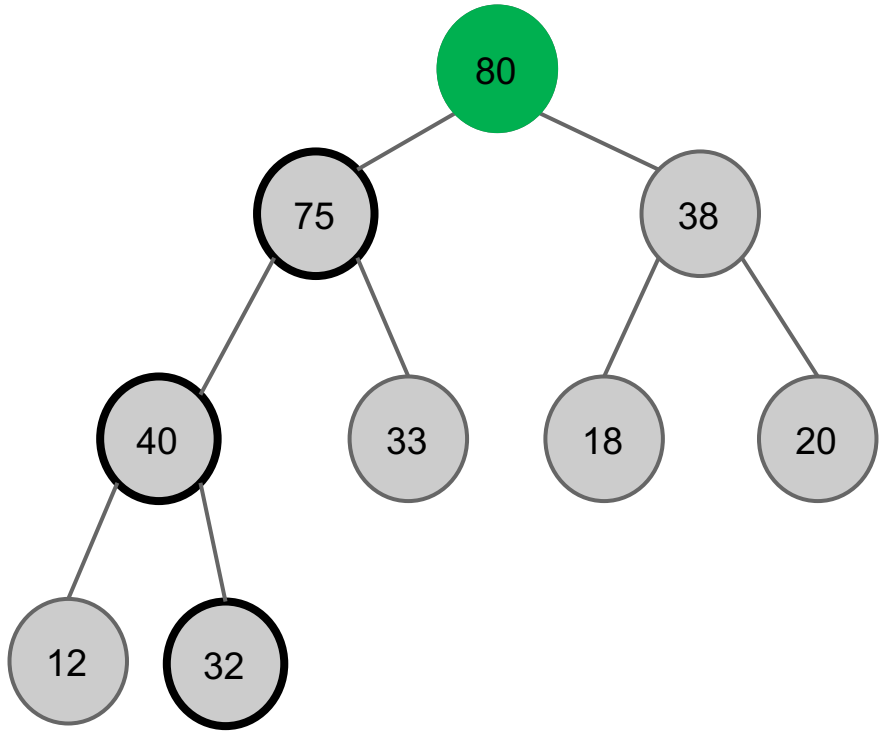
Bubble-up by swapping with parents, to proper location.



`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

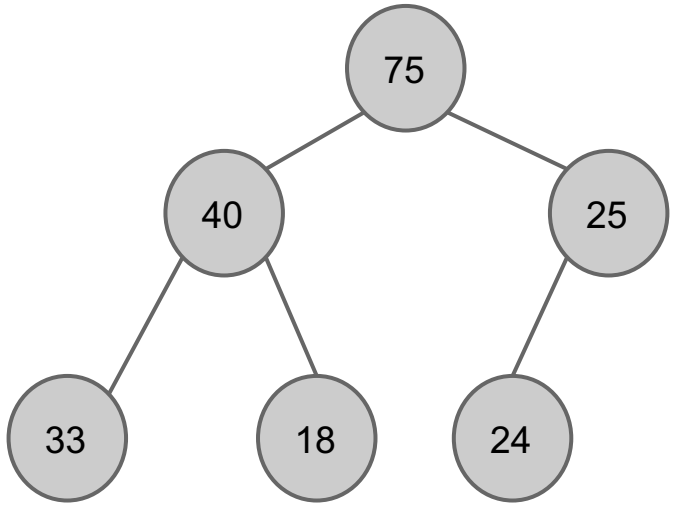
- ➔ How to use heap for **sorting**
- ➔ How to **build a heap** from an unsorted array

5 min

HeapSort

Sorts an array, in O(n logn) time

The Idea



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

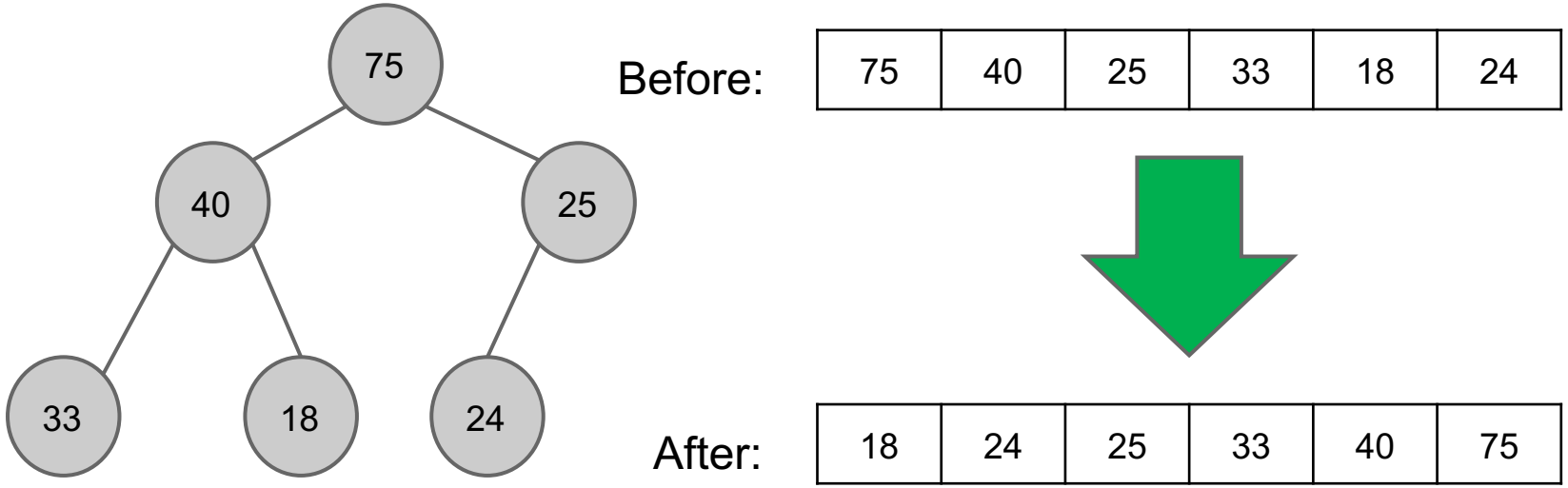
Keep extracting max for n times, the keys extracted will be sorted in non-ascending order.

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

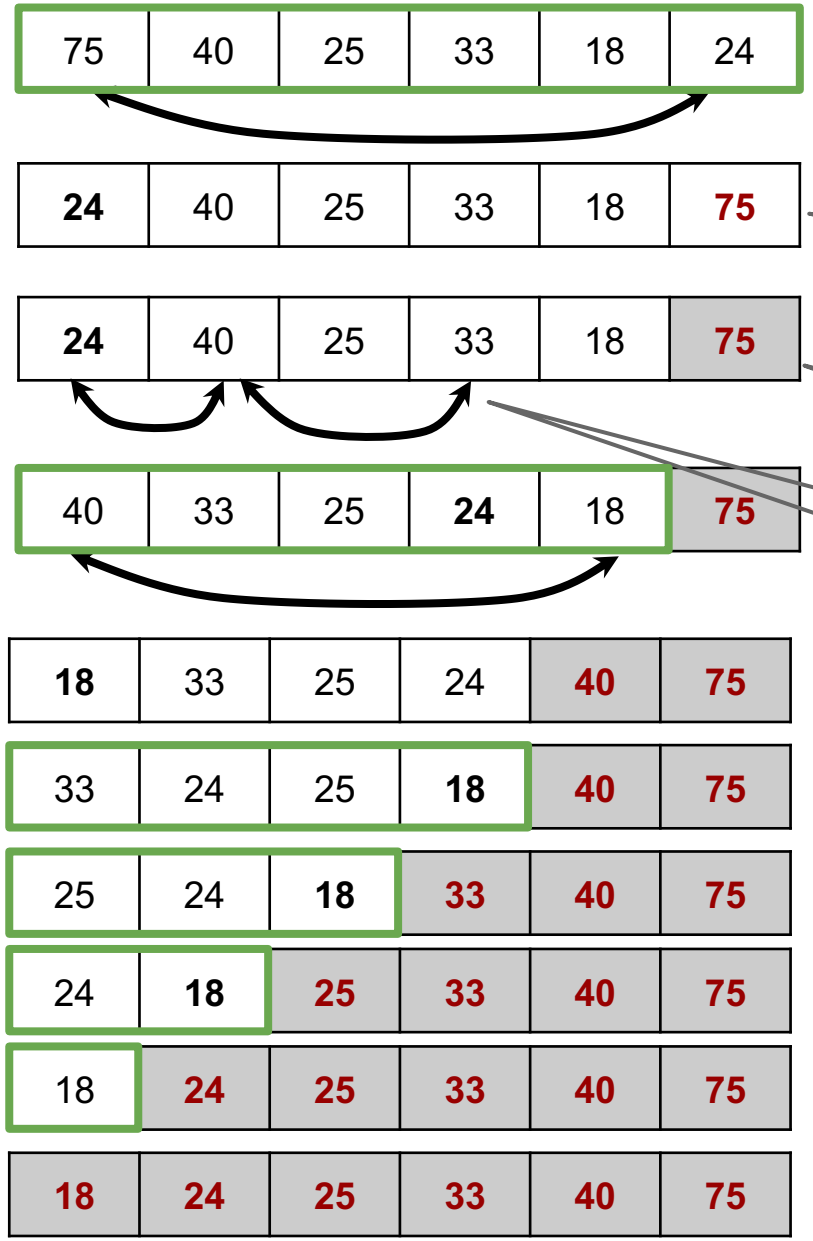
O(n log n)

Now let's be more precise

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



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



Step 1: swap first (75) and last (24), since the tail is where 75 (max) belongs to.

Step 2: decrement heap size

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

Step 3: fix the heap by bubbling down 24

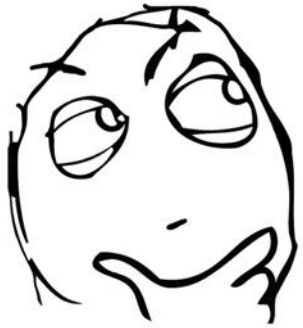
Repeat Step 1-3 until the array is fully sorted (at most n iterations).

Valid heaps are green rectangled

HeapSort - The Pseudo-code

```
HeapSort(A)
'''sort any array A into non-descending order '''
    BuildMaxHeap(A) # convert any array A into a heap-ordered one
    for i ← A.size downto 2:
        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
```

Missing!



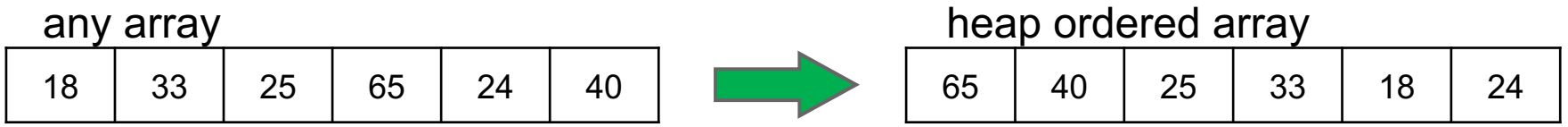
Does it work?

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

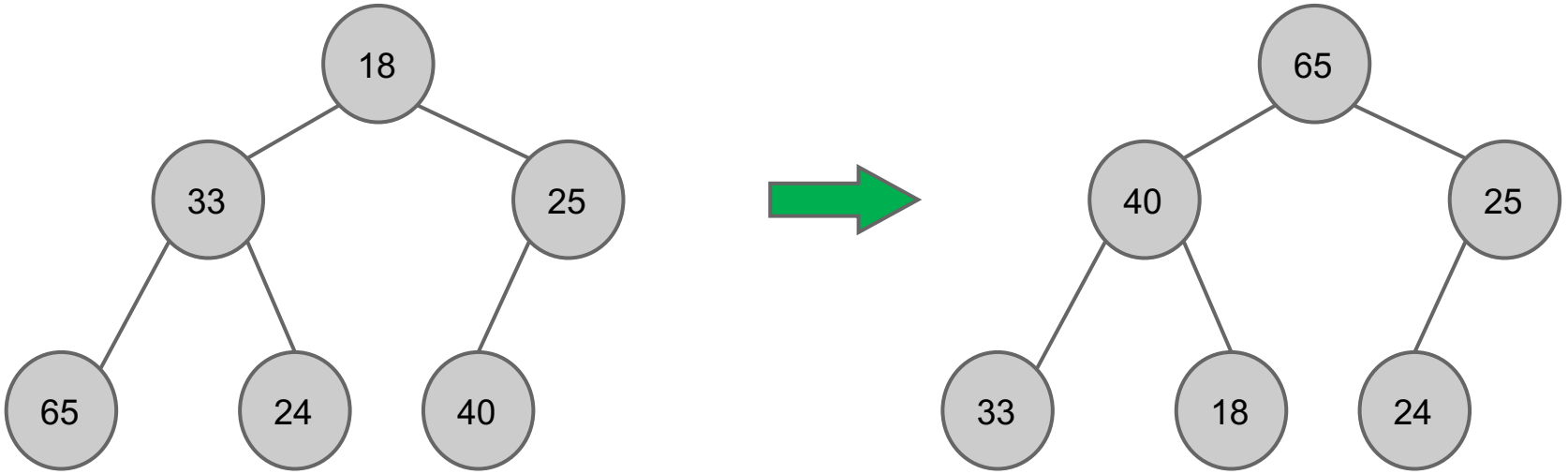
`BuildMaxHeap(A)`

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

Convert any Array into a Heap Ordered One



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:

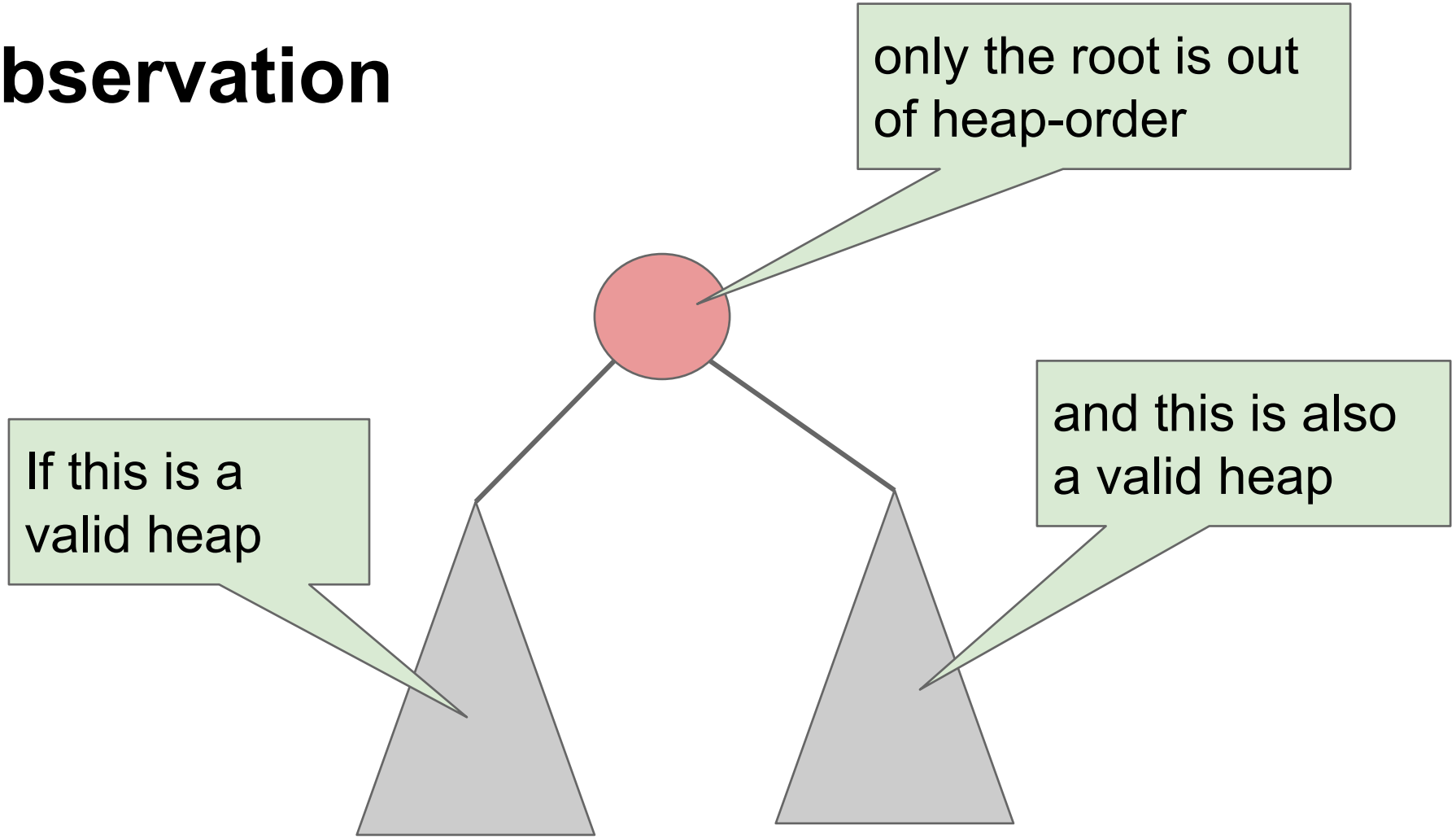
Each Insert takes O(log n), there are n inserts...

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

Not in-place, needs a second array.

a better idea for `BuildMaxHeap`

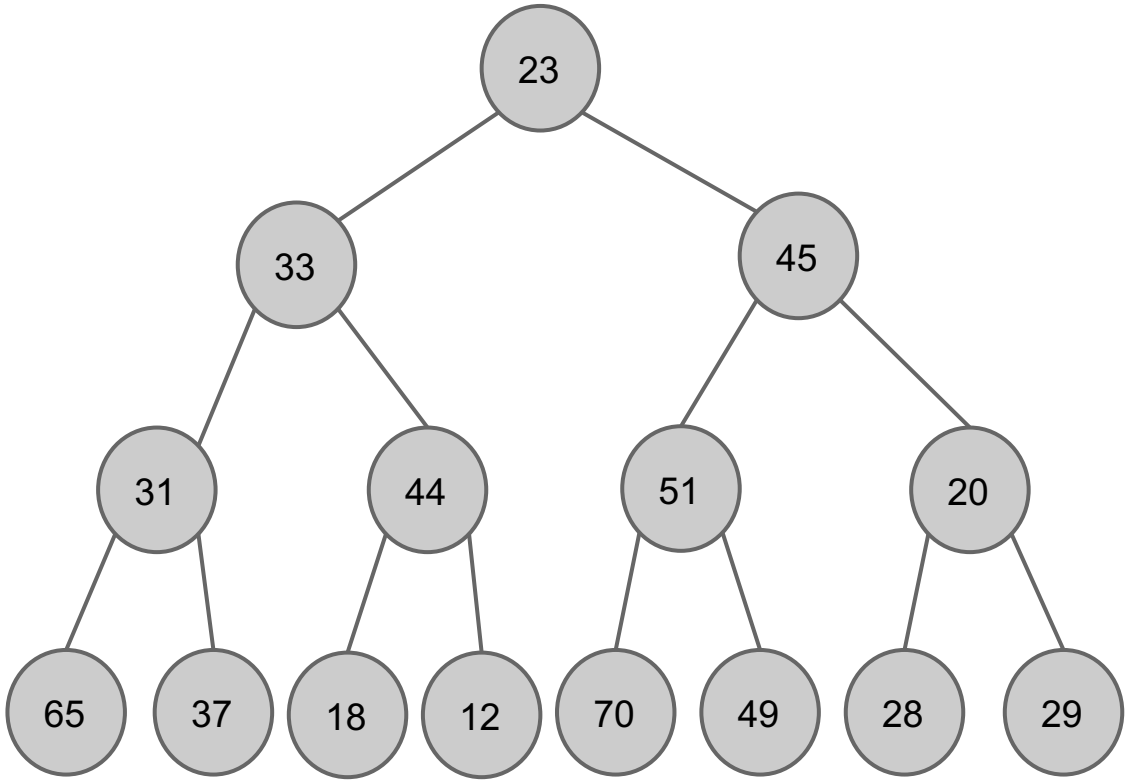
Key Observation



To make the whole thing a valid heap, all you need to do is …
bubbling-down the root.

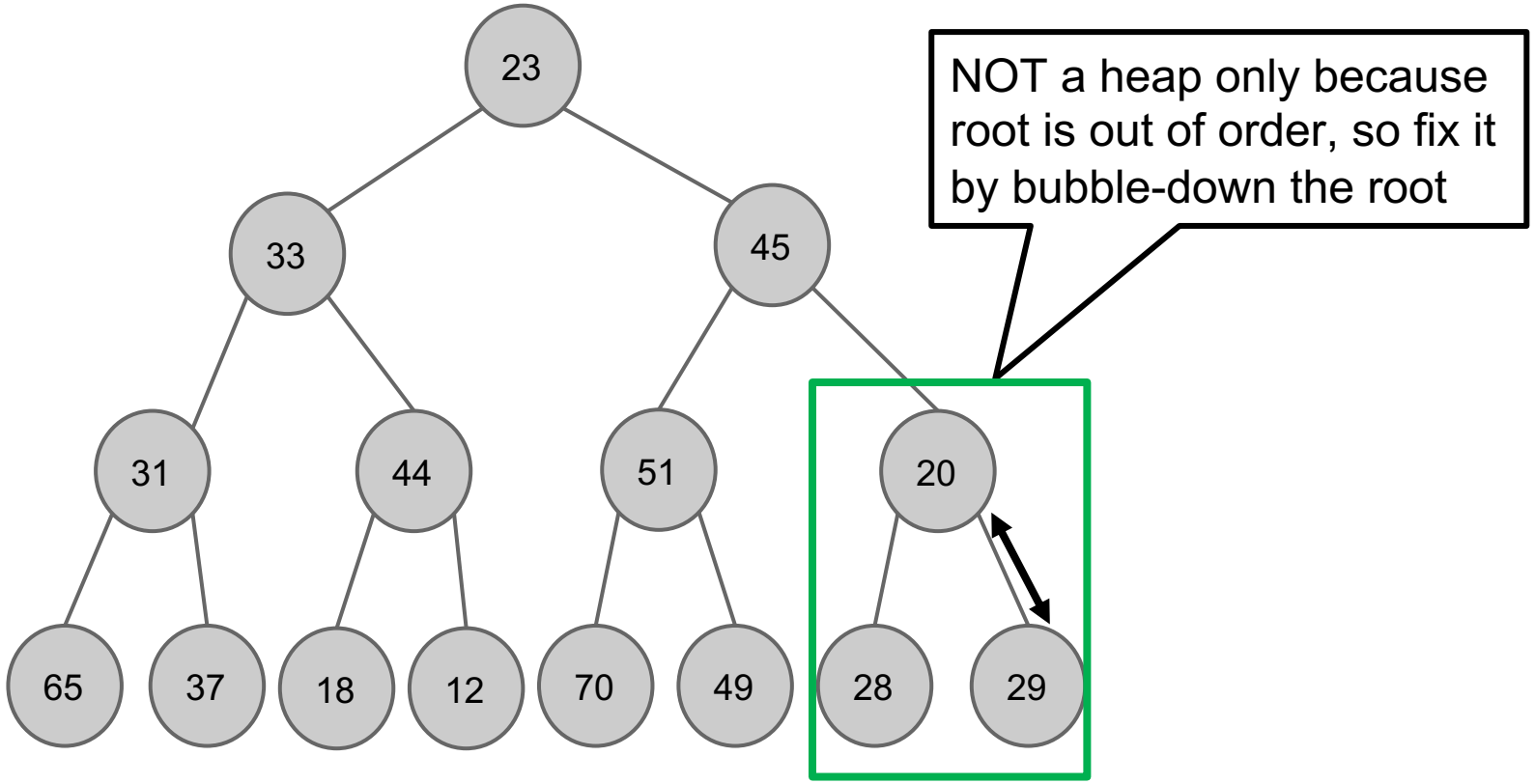
Idea #2

Fix heap order, from bottom up.



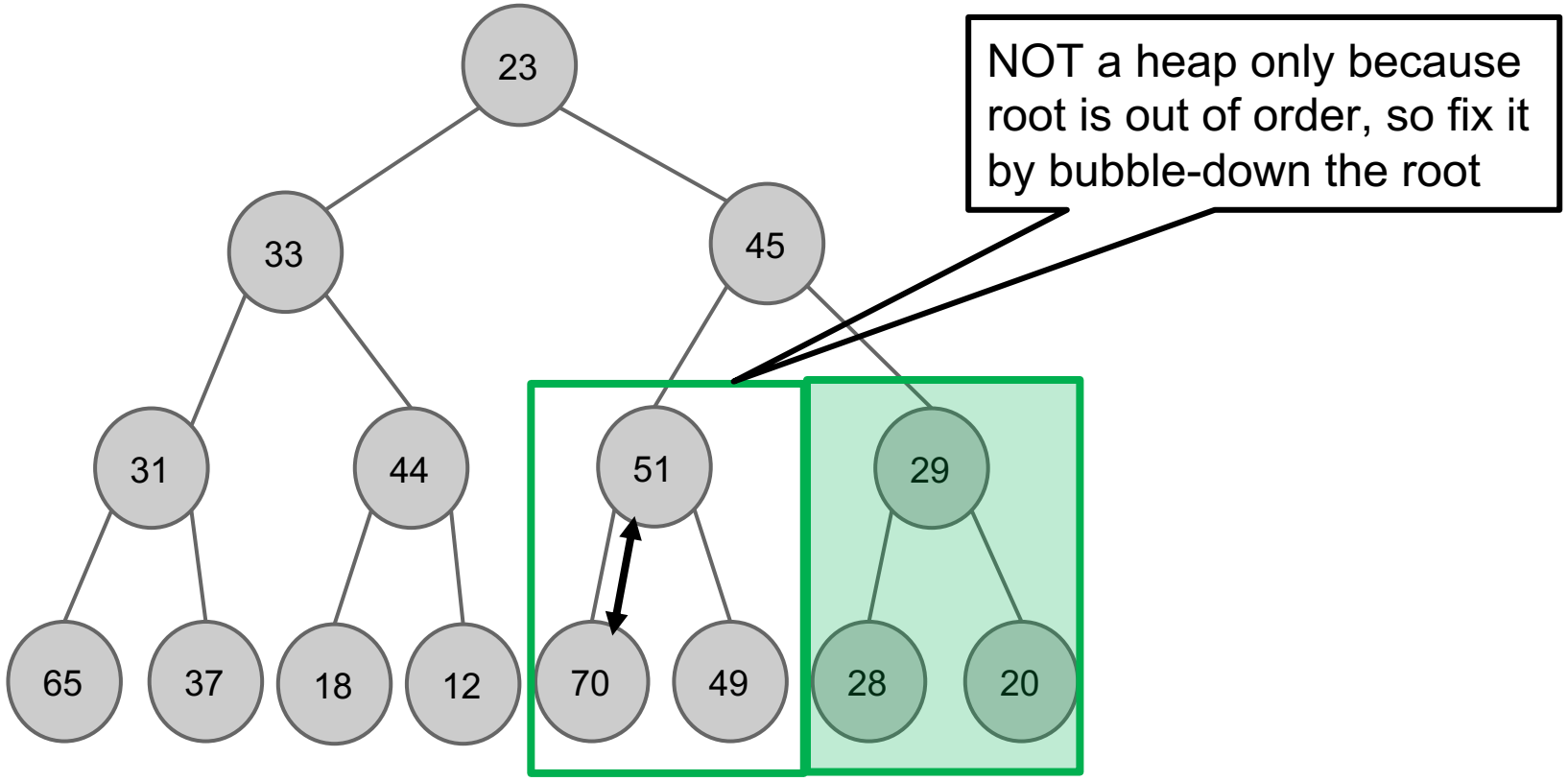
Idea #2

Adjust heap order, from bottom up.



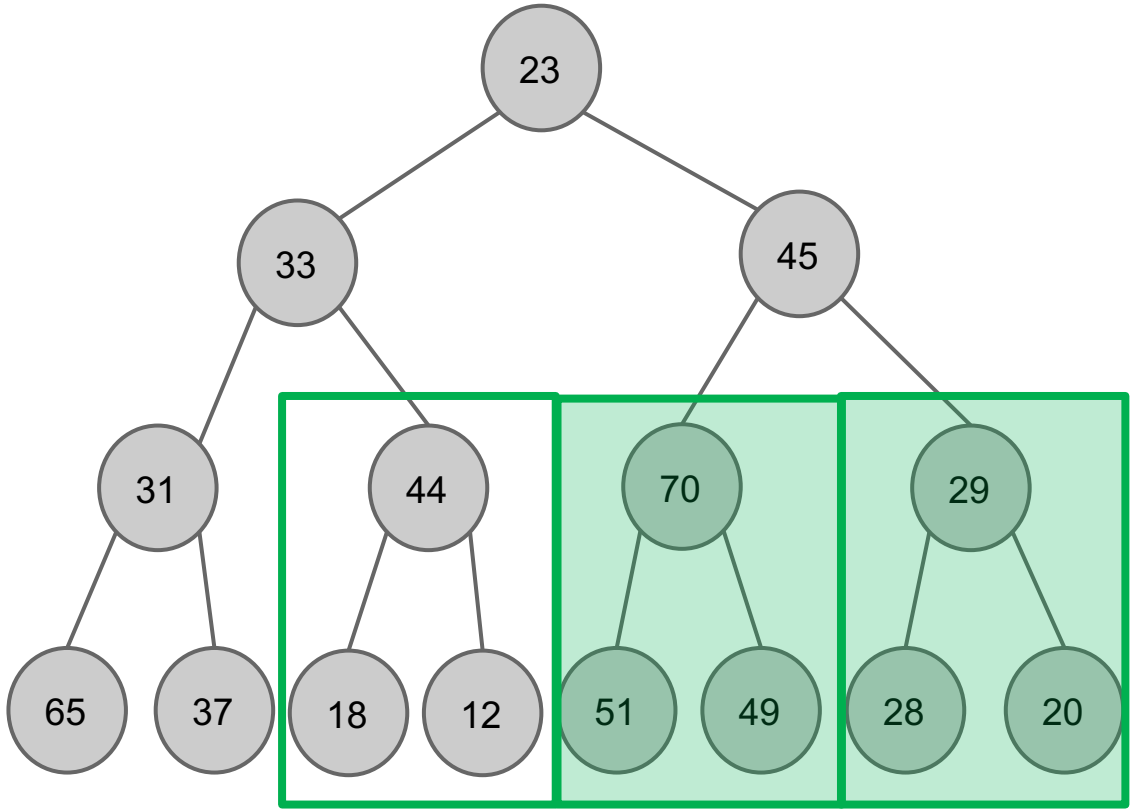
Idea #2

Adjust heap order, from bottom up.



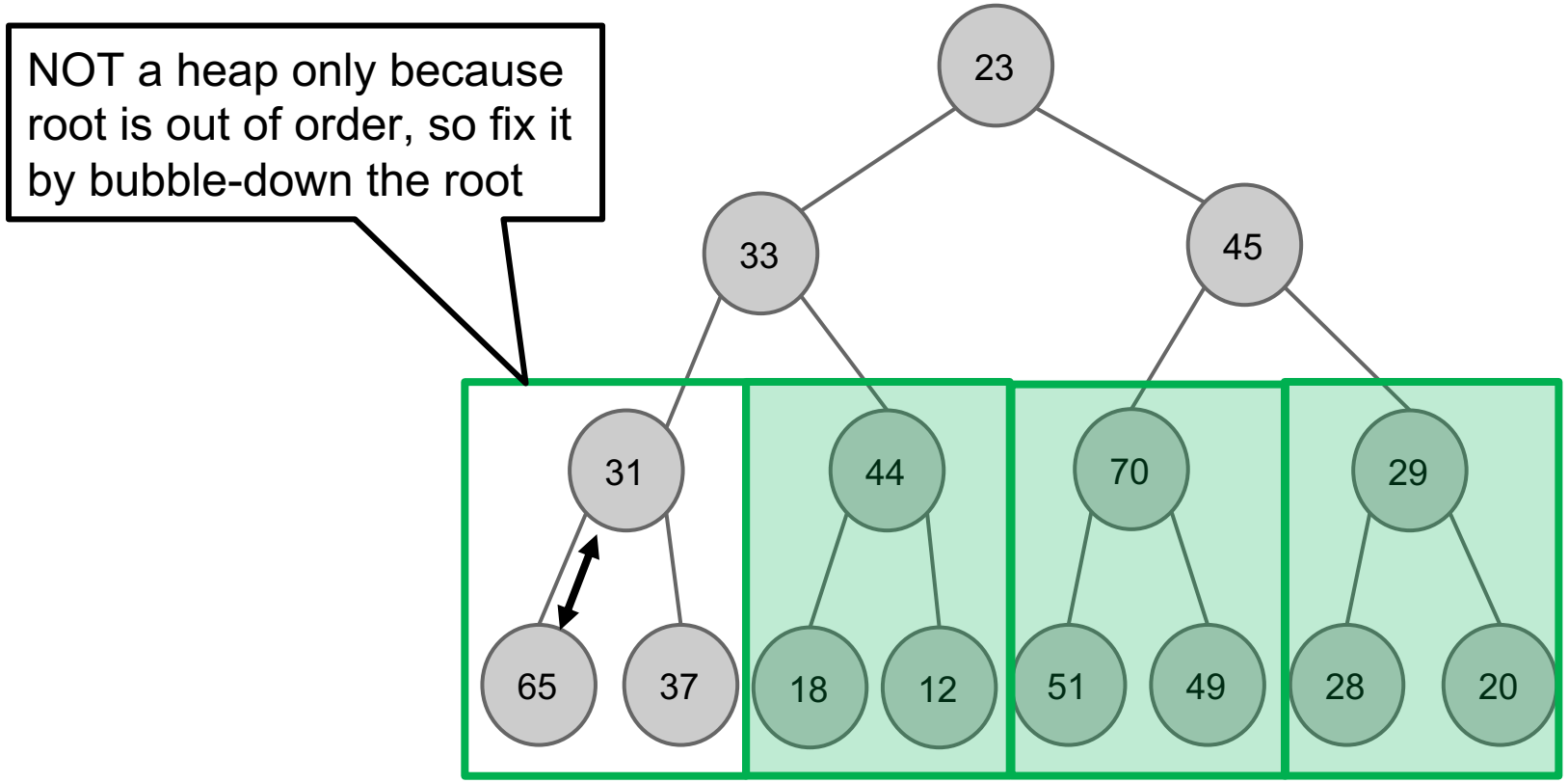
Idea #2

Adjust heap order, from bottom up.



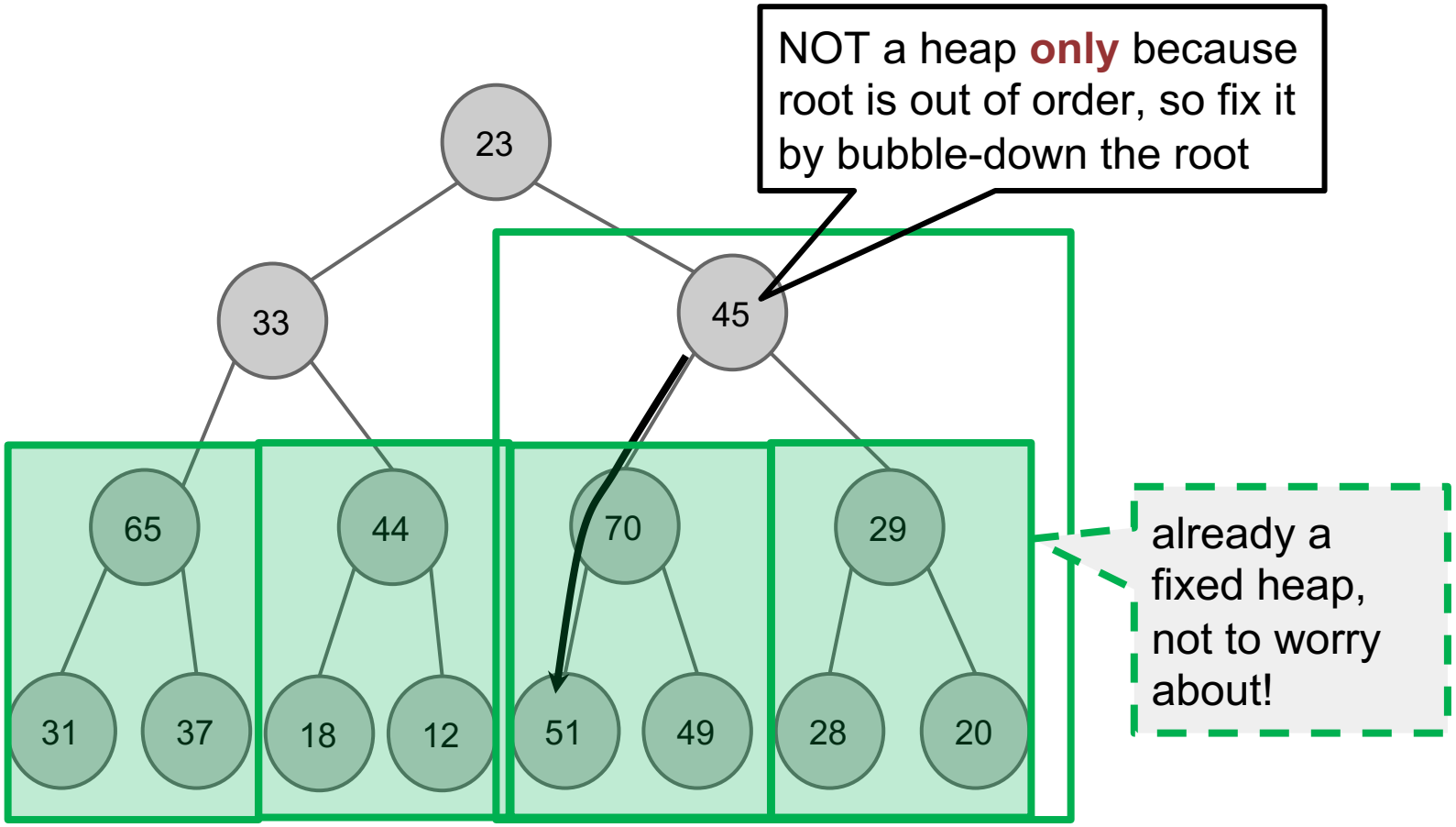
Idea #2

Adjust heap order, from bottom up.



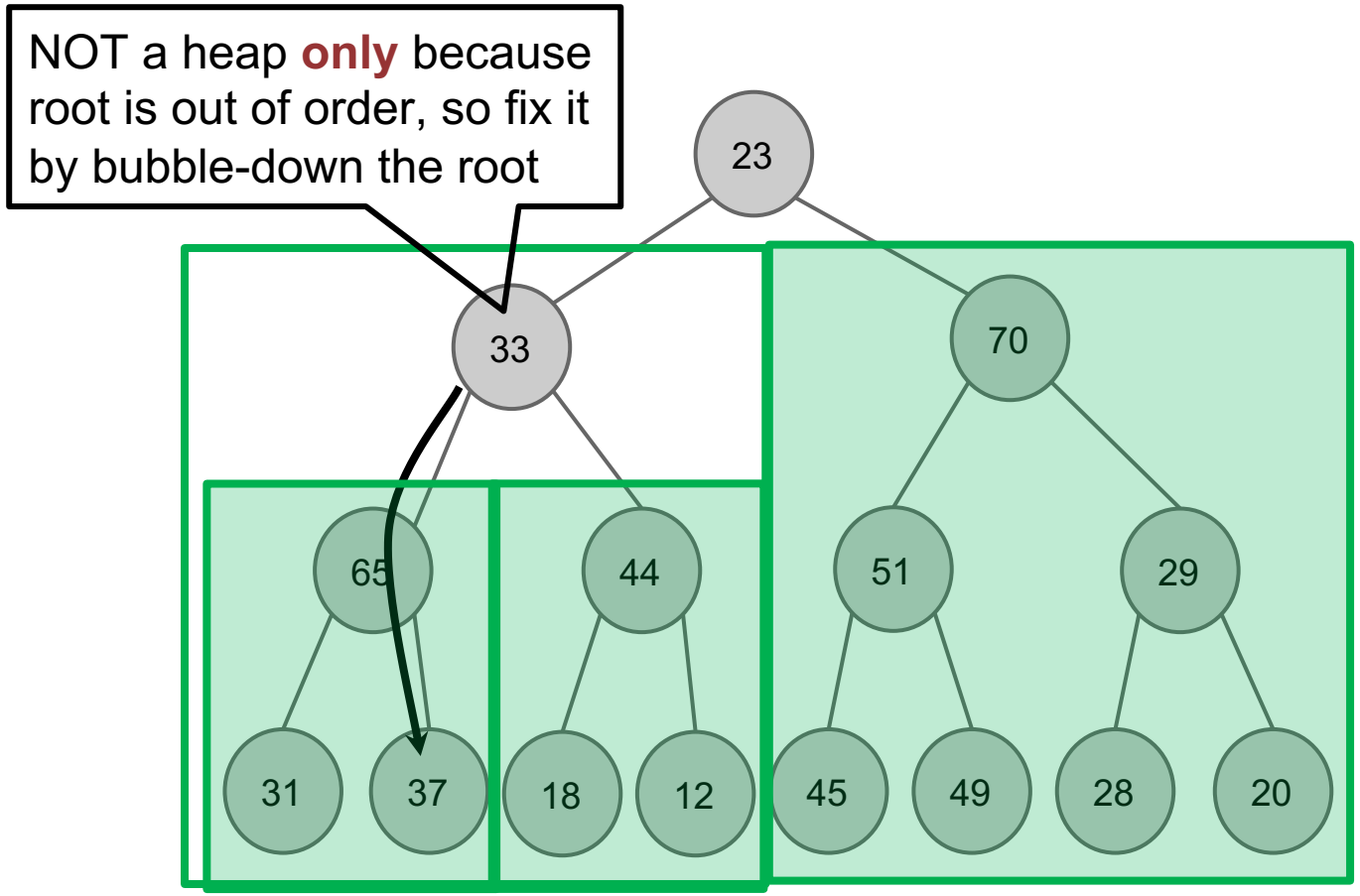
Idea #2

Adjust heap order, from bottom up.



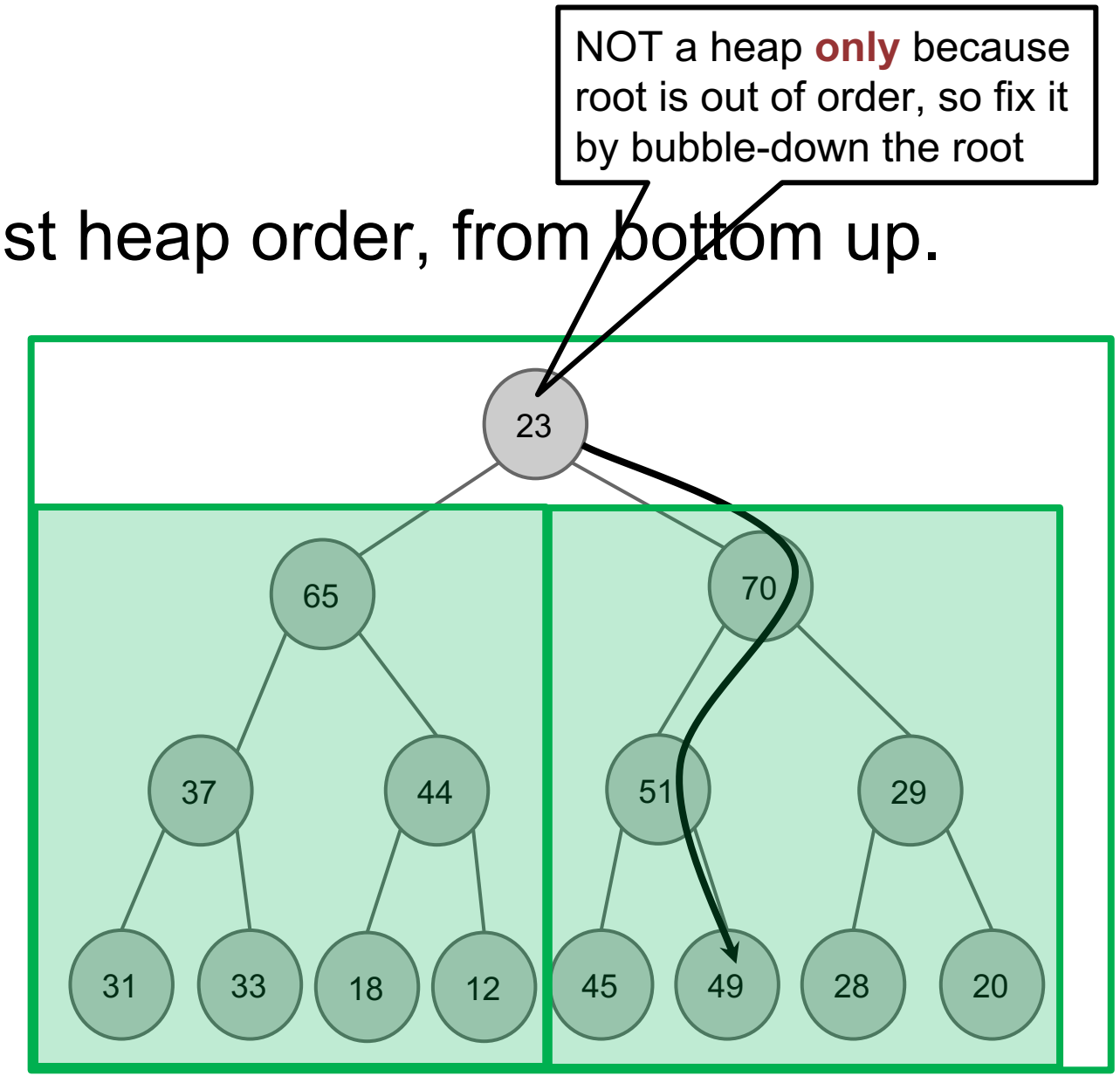
Idea #2

Adjust heap order, from bottom up.



Idea #2

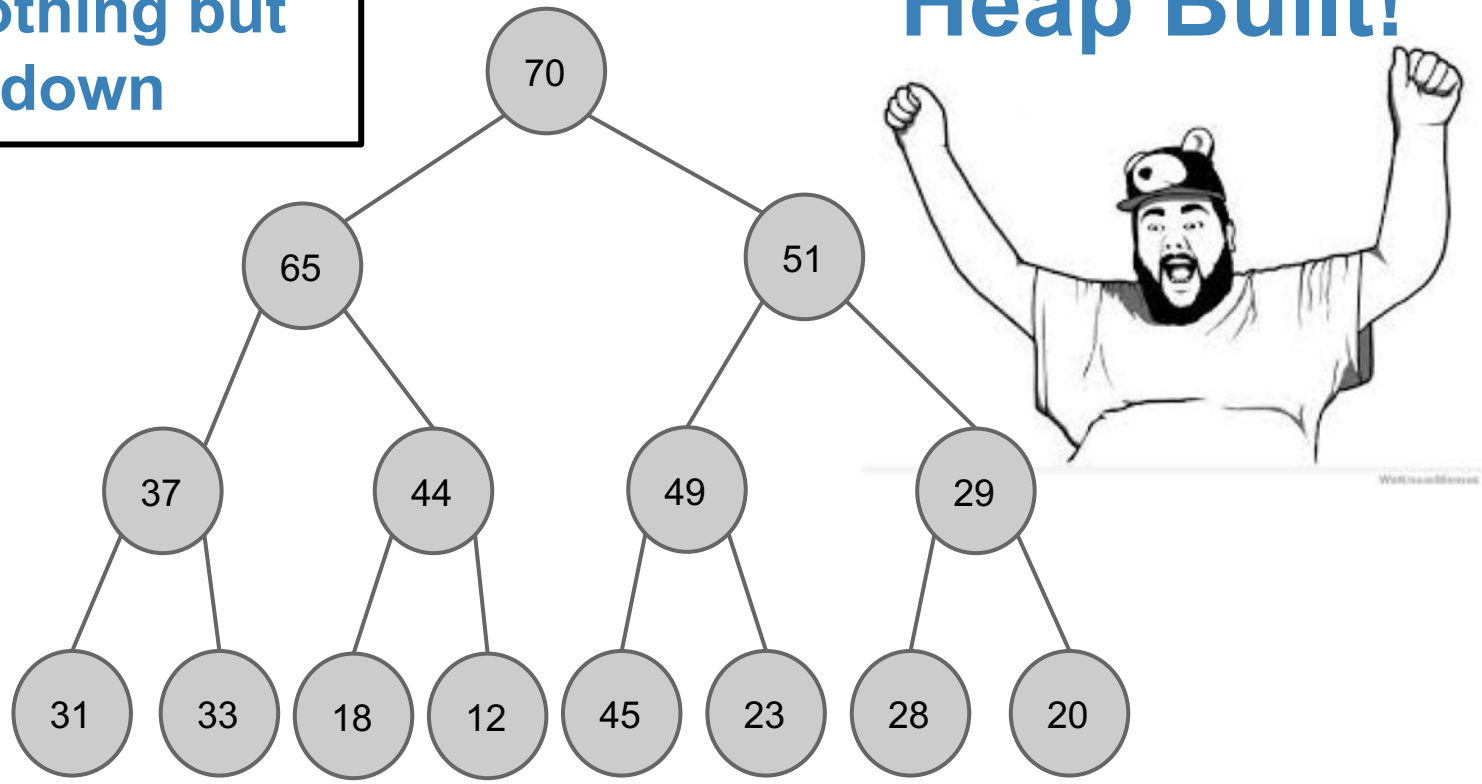
Adjust heap order, from bottom up.



Idea #2

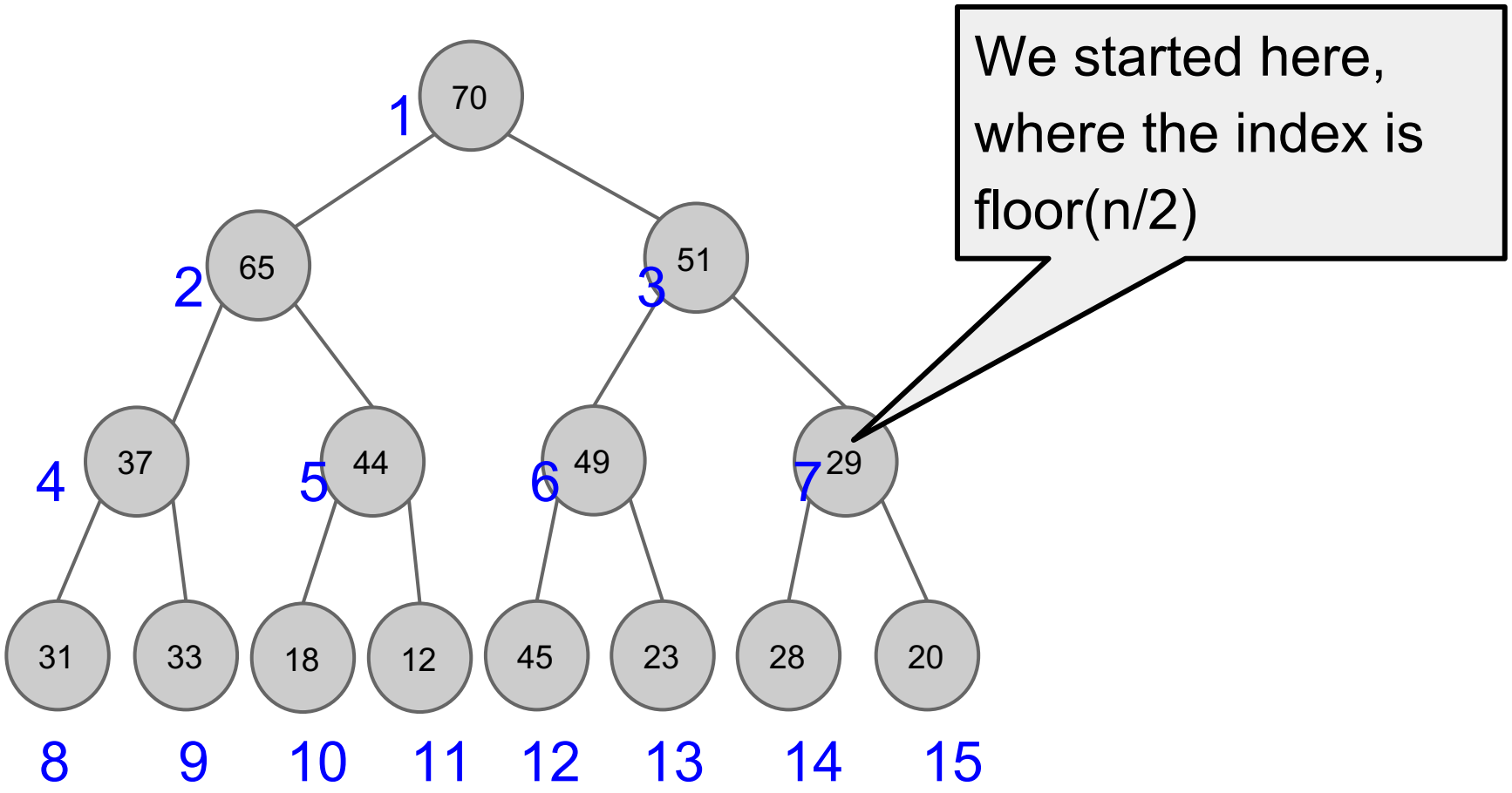
Adjust heap order, from bottom up.

We did nothing but bubbling-down

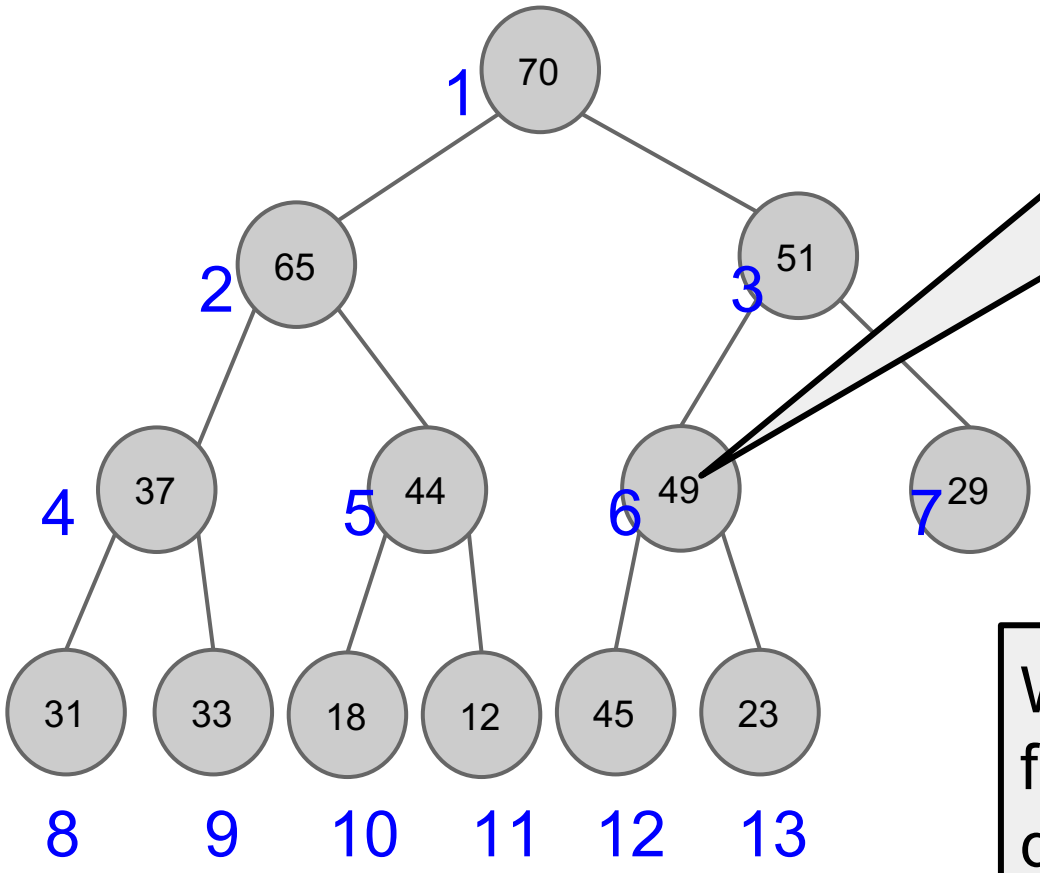


Heap Built!

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)
```

➔ It's in-place, no need for extra array (we did nothing but bubble-down, which is basically swappings).

➔ How about runtime?

- Each bubble down is O(log n)
- we do it roughly n/2 times
- so overall it is **O(n log n)**, … Right?

Analysis

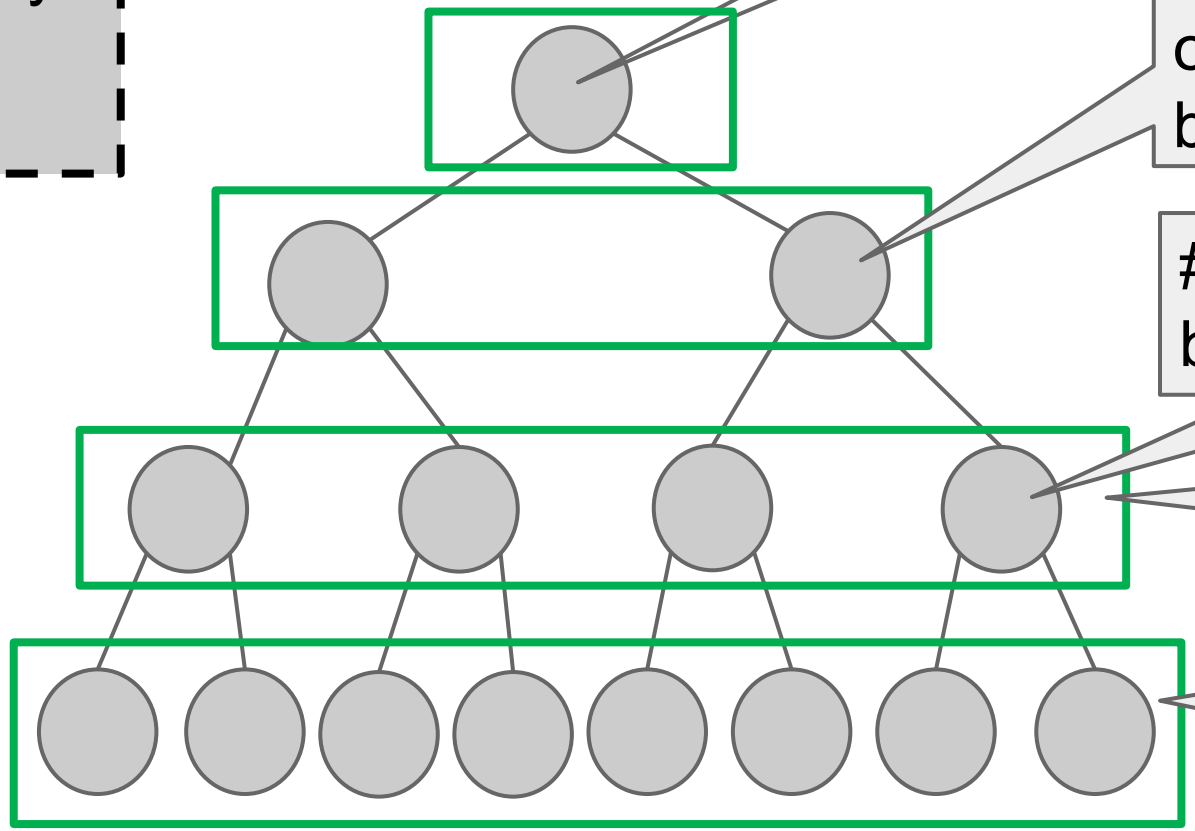
Worst-case running time of

`BuildMaxHeap(A)`

Intuition

A complete binary tree with n nodes...

How many levels?
~ log n



n/16 nodes, and # of swaps per bubble-down: ≤3

n/8 nodes, and # of swaps per bubble-down: ≤2

# of swaps per bubble-down: ≤1

~ n/4 nodes

~ n/2 nodes, and no work done at this level.

So, total number of swaps

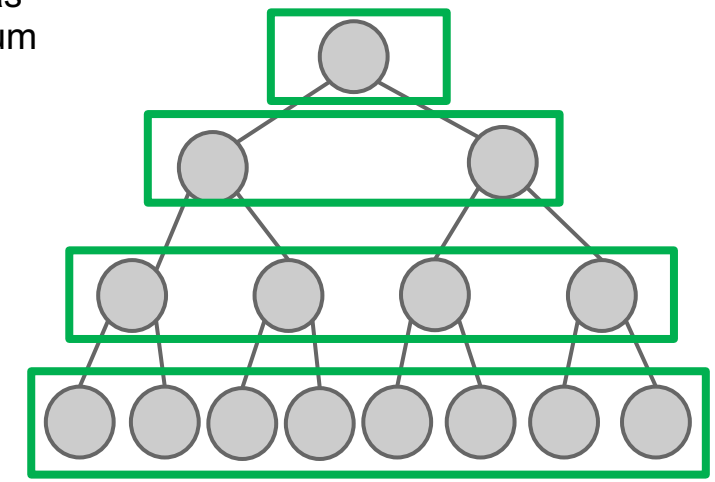
$$T(n) = 1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + 3 \cdot \frac{n}{16} + \dots$$

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

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

same trick as Week 1's sum

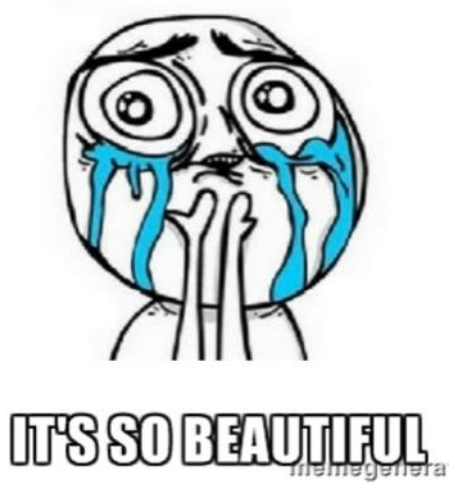
$$= n$$



The Power of Analysis

```
BuildMaxHeap(A):

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



This 2-line simple algorithm for BuildMaxHeap, which is easier to implement than the insert-n-times algorithm, by analysis, can be **proven** to be an order of magnitude faster (O(n) instead of O(n logn)).

One can never design such an elegant algorithm without the ability to perform **analysis**.

BuildMaxHeap: my second favourite algorithm in CLRS.

--Larry Zhang

Summary

`HeapSort(A)`

➔ Sort an unsorted 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

https://visualgo.net/en/heap

Today we learned

➔ ADT: Priority Queue

➔ Heap: a data structure for implementing priority queue efficiently.

➔ How to sort with heap

➔ How to build a heap, elegantly.

Next week

➔ ADT: Dictionary

➔ Data structure: Binary Search Tree