Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

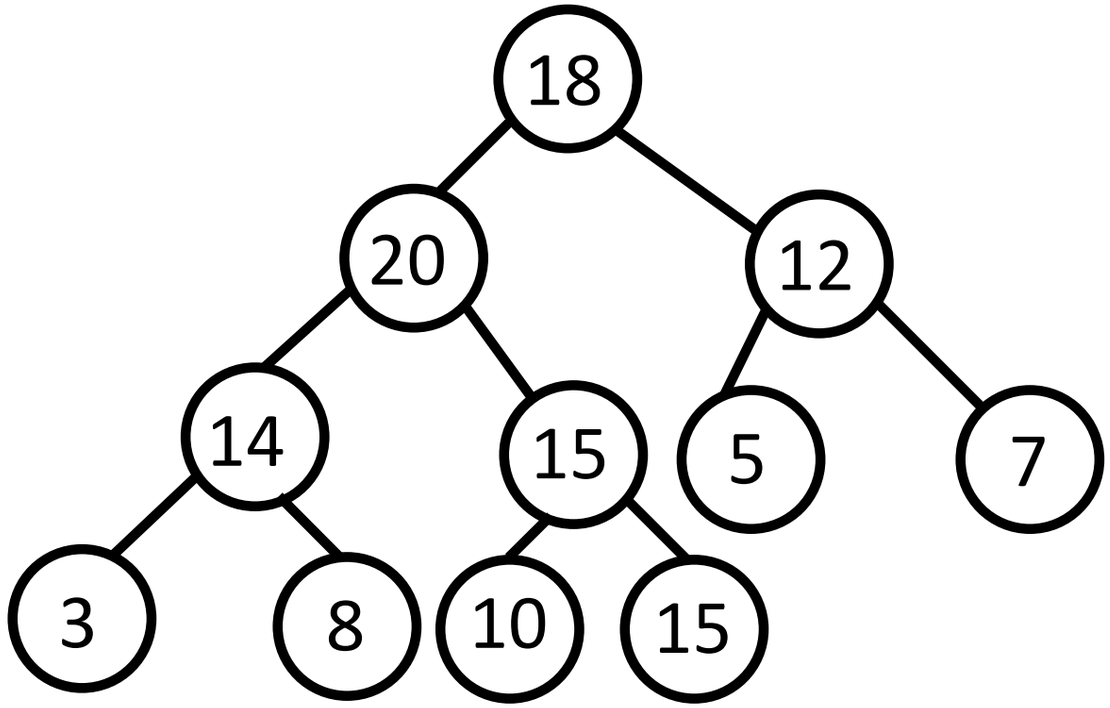
Lecture 36
Heaps and Heap sort

S

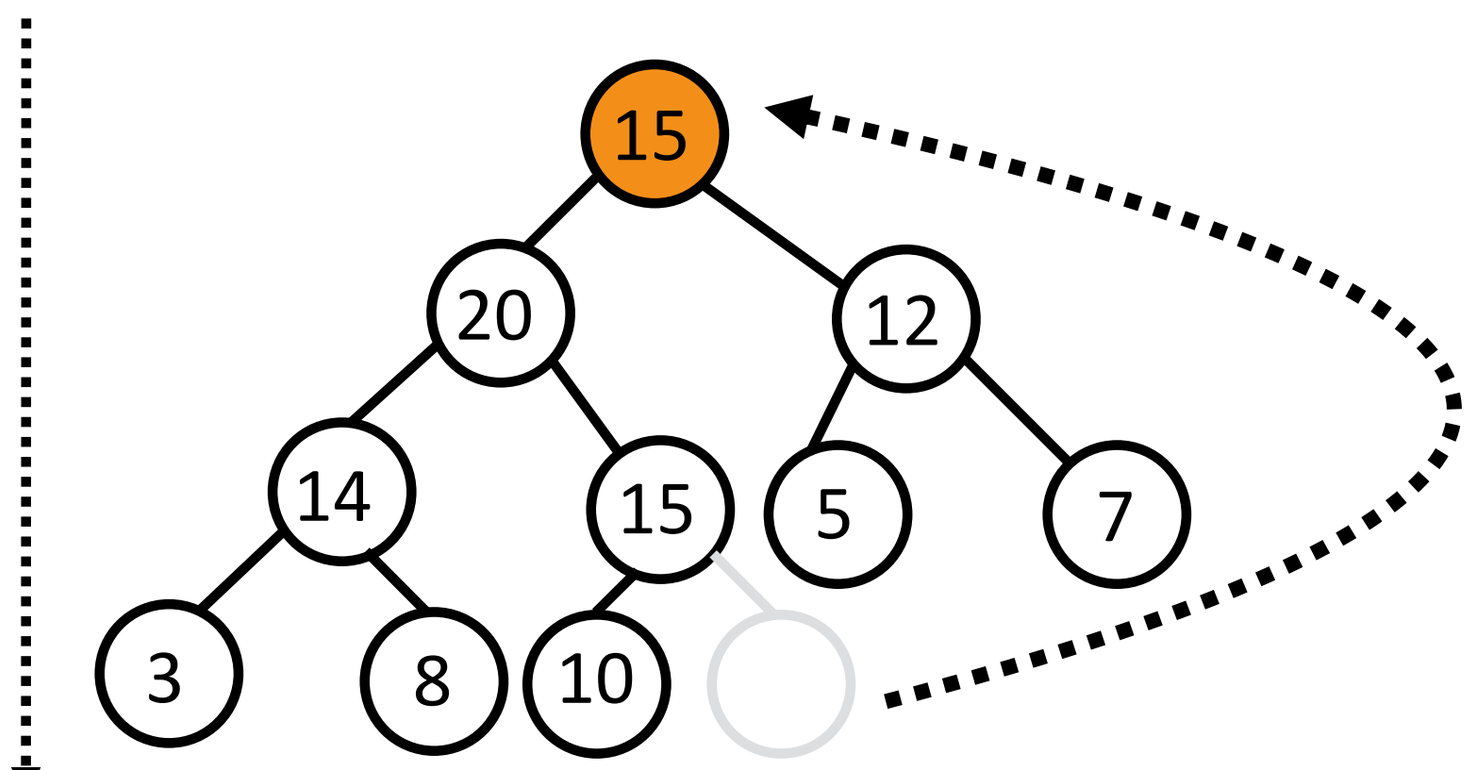
Operations

add:
- put at the bottom
- while order is broken, rise.

get_max:
- swap root with last item
- remove last item
- while order is broken, sink.


18
20
12
14
15
5
7
3
8
10
15

sink
(swapping with
largest child)



A concrete implementation

```
def largest_child(self, k):
    if self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1
```

what if k only has one child

```
# Make the item at index k sink to the correct position.
def sink(self,k):
    while 2*k <= self.count:
        child = self.largest_child(k)
        if self.array[k] >= self.array[child]:
            break
        self.swap(child,k)
        k = child
```

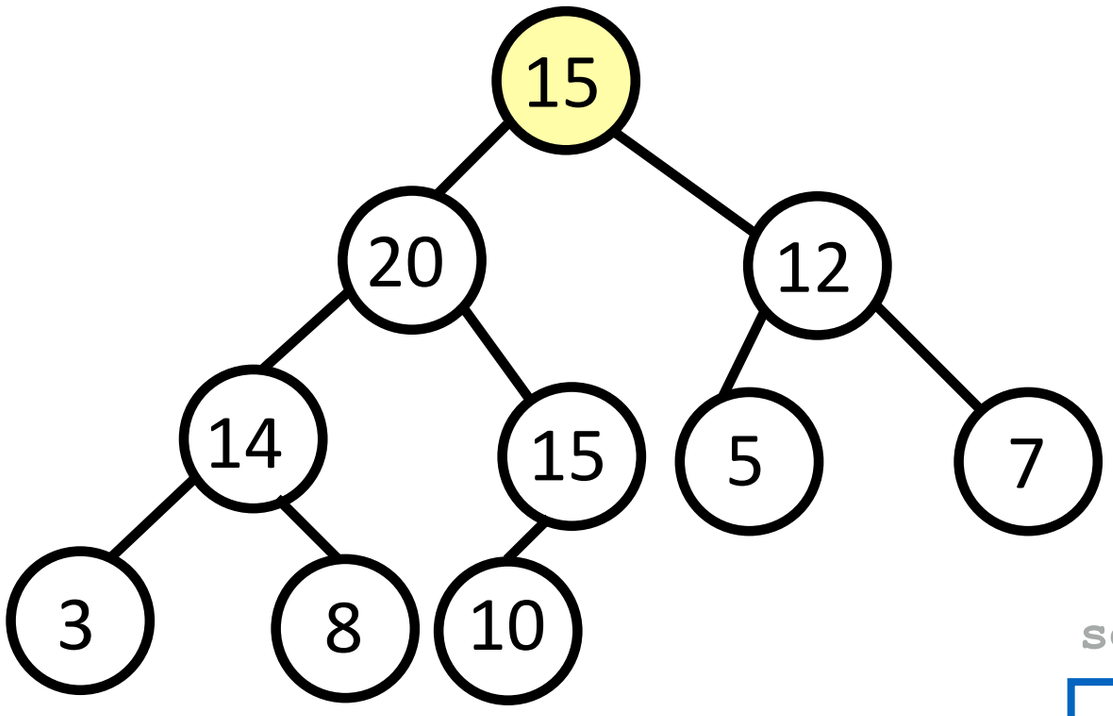
left child in last position means
k has only one child

```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1
```

```
# Make the item at index k sink to the correct position.
def sink(self,k):
    while 2*k <= self.count:
        child = self.largest_child(k)
        if self.array[k] >= self.array[child]:
            break
        self.swap(child,k)
        k = child
```

On subtle errors

- Errors like that are very easy to make, and hard to spot.
- Your armoury against them includes:
  - ➞ Thorough testing (recall us asking for test cases?)
  - ➞ Code review
  - ➞ Proofs of correctness



k=1

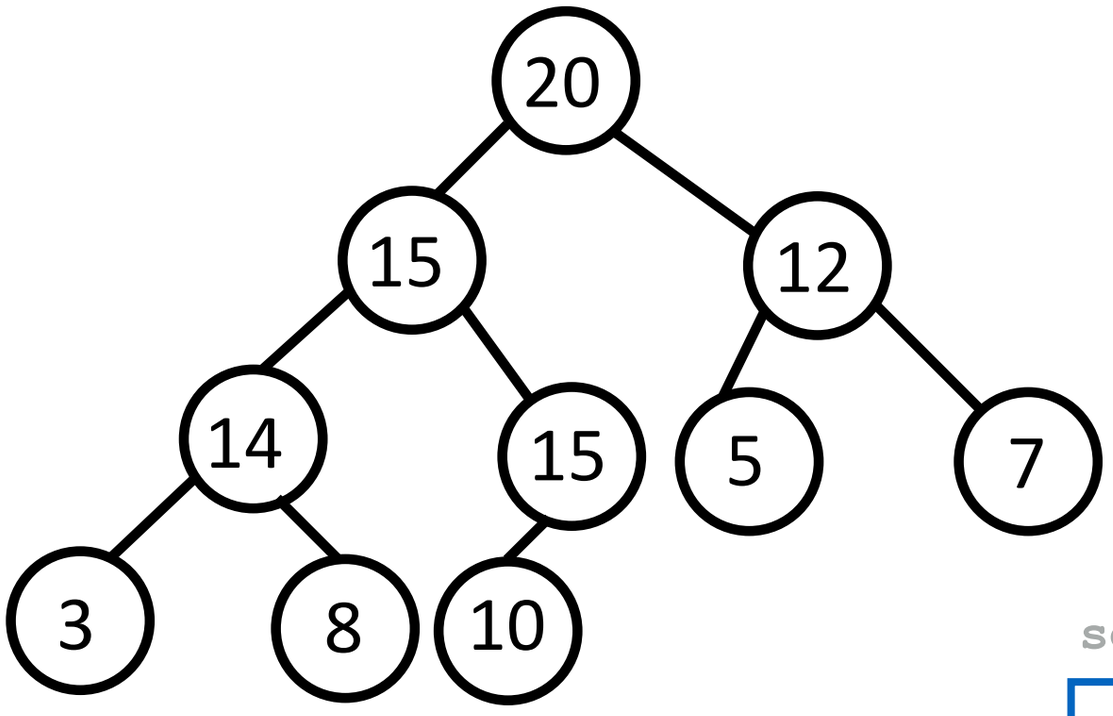
self.count = 10

self.array

	15	20	12	14	15	5	7	3	8	10
0	1	2	3	4	5	6	7	8	9	10

```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# Make the item at index k sink to the correct position.
def sink(self,k):
    while 2*k <= self.count:
        child = self.largest_child(k)
        if self.array[k] >= self.array[child]:
            break
        self.swap(child,k)
        k = child
```



child = 5

k = 2

self.count = 10

self.array

	20	15	12	14	15	5	7	3	8	10
0	1	2	3	4	5	6	7	8	9	10

```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# Make the item at index k sink to the correct position.
def sink(self,k):
    while 2*k <= self.count:
        child = self.largest_child(k)
        if self.array[k] >= self.array[child]:
            break
        self.swap(child,k)
        k = child
```

best case: O(1)

worst case: O(log N)

(may need to consider comparison operations)

Complexity of get_max

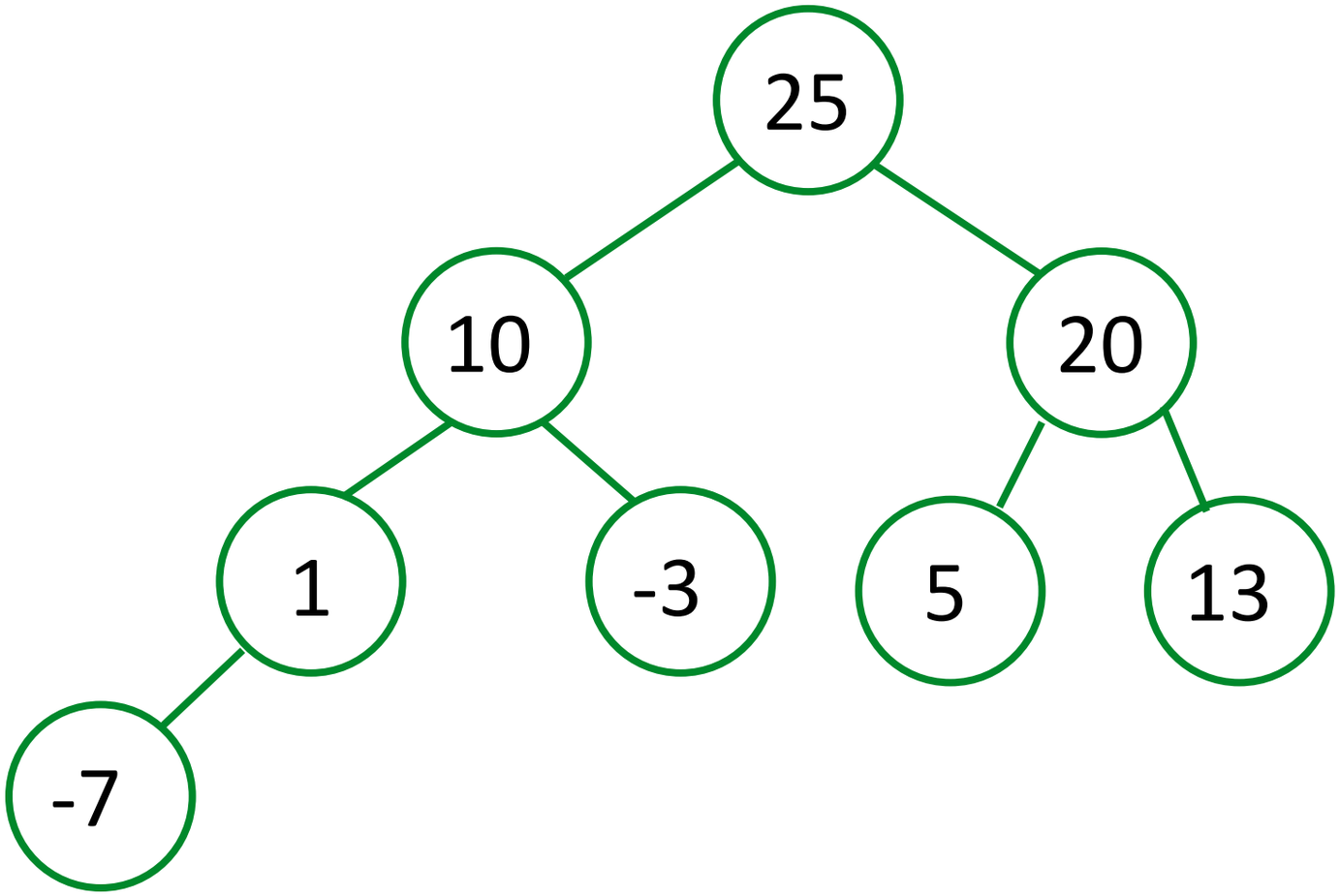
- Loop in sink can iterate at most depth times ≈ log(N)
  (after depth iterations, the new item is at the root)

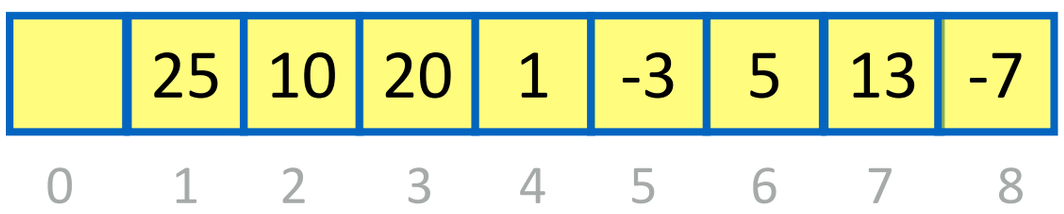
- Best case: O(1)*OCompare when the item is larger or equal than largest children.

- Worst case: O(log N)*OCompare when the item sinks all the way to the bottom.

Heap sort

[5, -7, 10, -3, 13, 20, 25,1]





[5, -7, 10, -3, 13, 20, 25,1]

	-7	-3	1	5	10	13	20	25
0	1	2	3	4	5	6	7	8

These numbers are sorted!

Heap sort

- For each element in the array:
  - Add it to the Heap: O(log(N))

N times

- While heap contains elements:
  - Get max item: O(log(N))
  - Put it in the “hole” made by previous step: O(1)

N times

worst case: O(N log N)

Summary

- A simple Heap implementation
  - rise
  - sink
  - largest_child

- Heap Sort