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

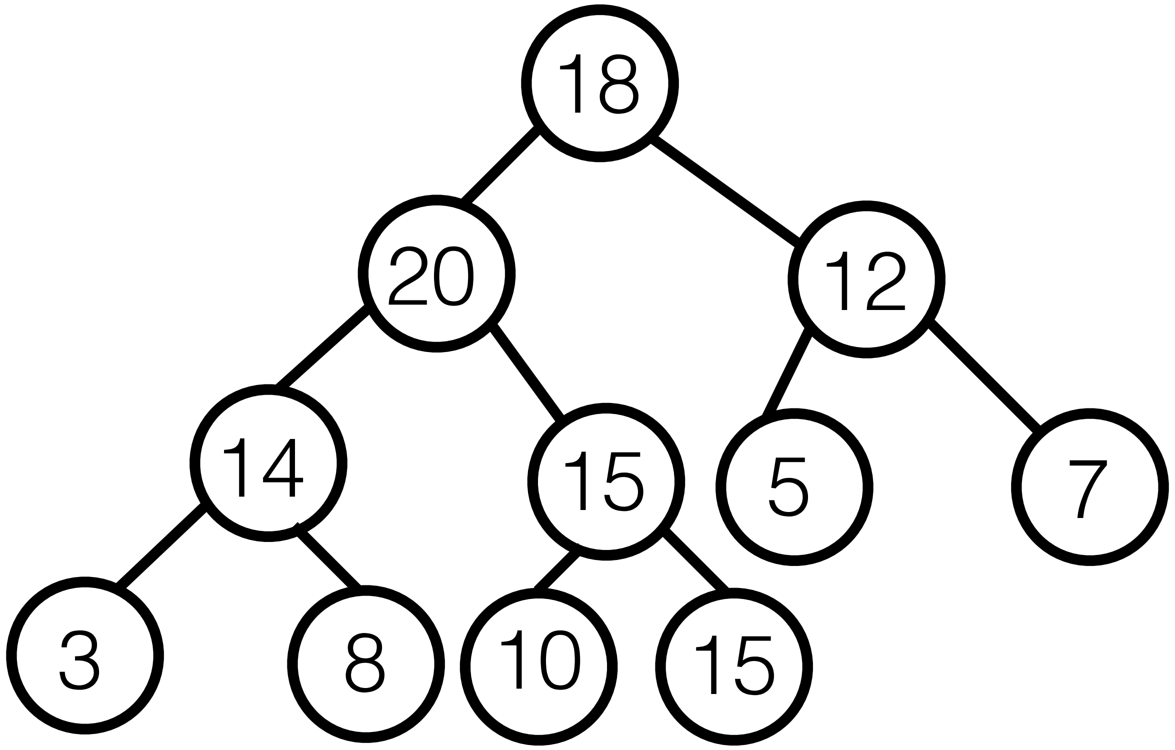
Lecture 34
Heaps and Heap sort

FIT 1008
Introduction to Computer Science

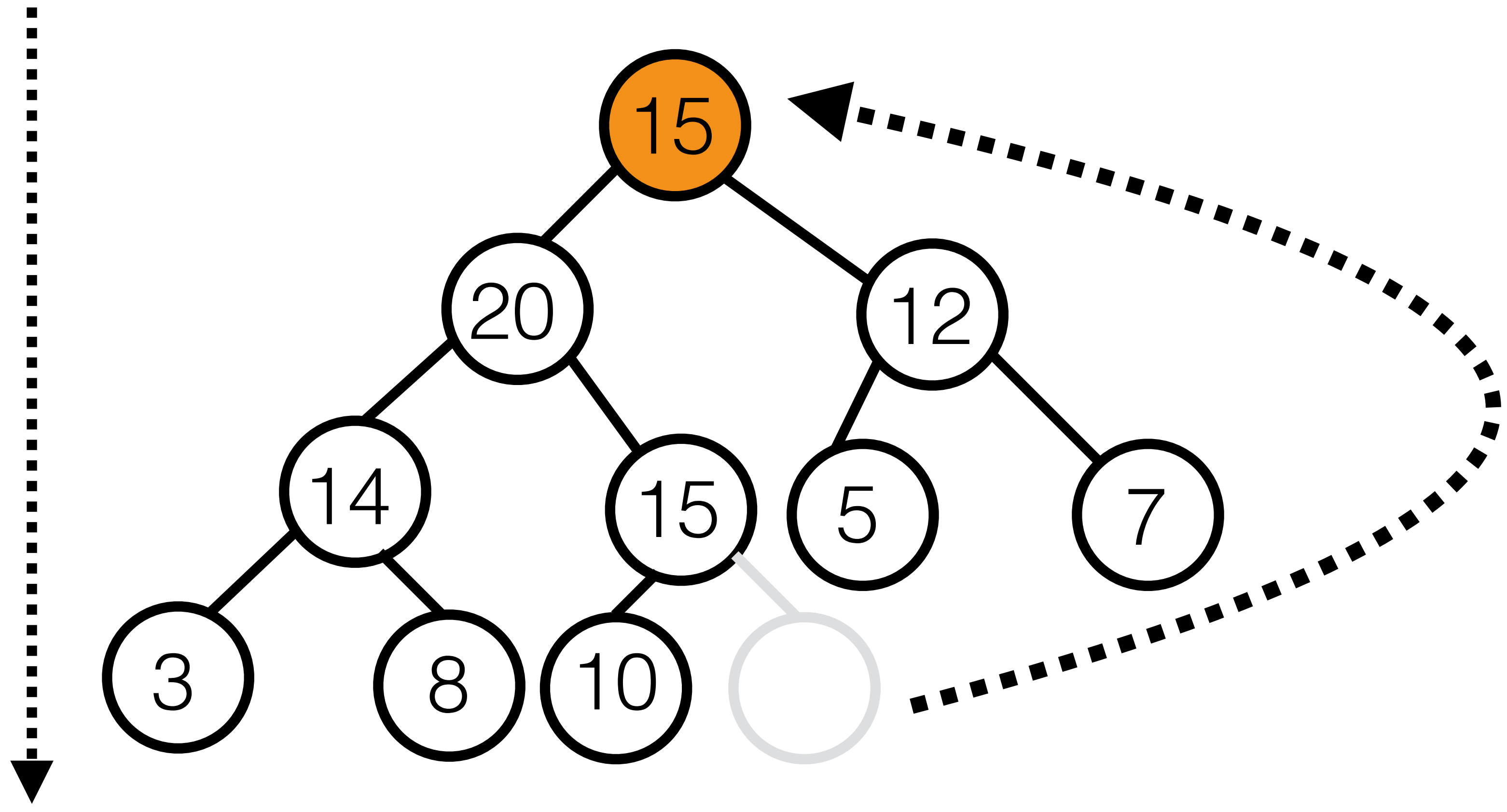
Operations

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.

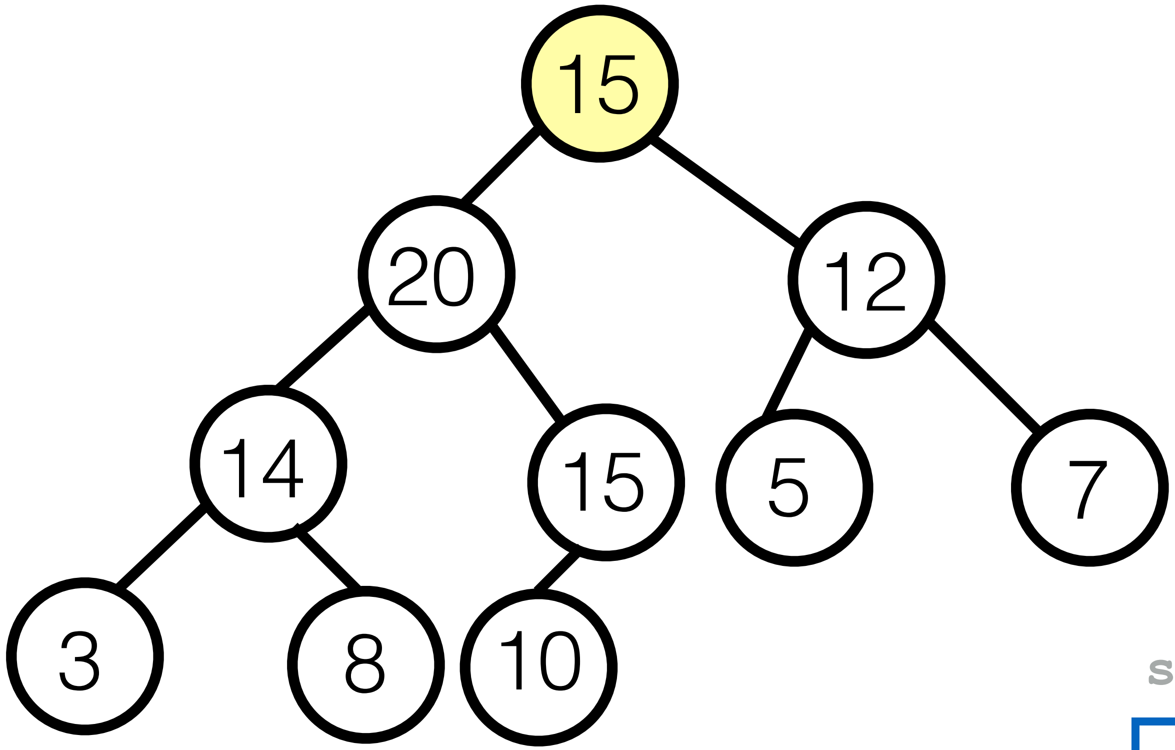

18
20
12
14
15
5
7
3
8
10
15

sink
(swapping with **largest child**)



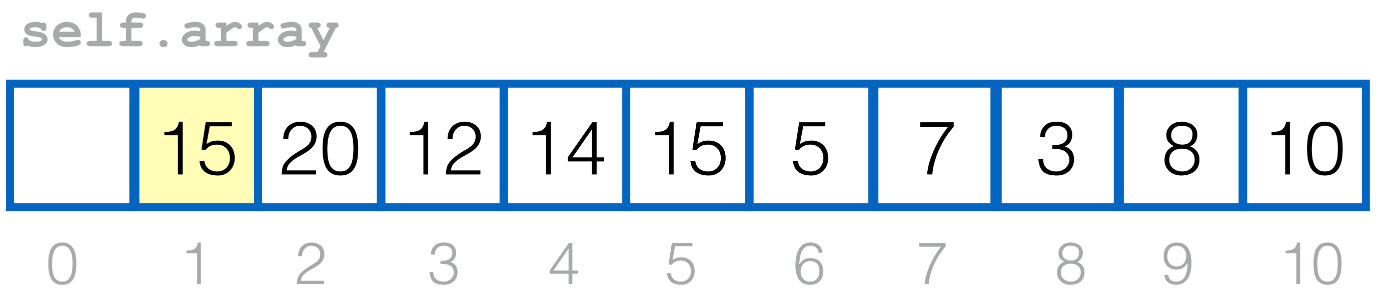
left child in last position means
k has only one child

```
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
```

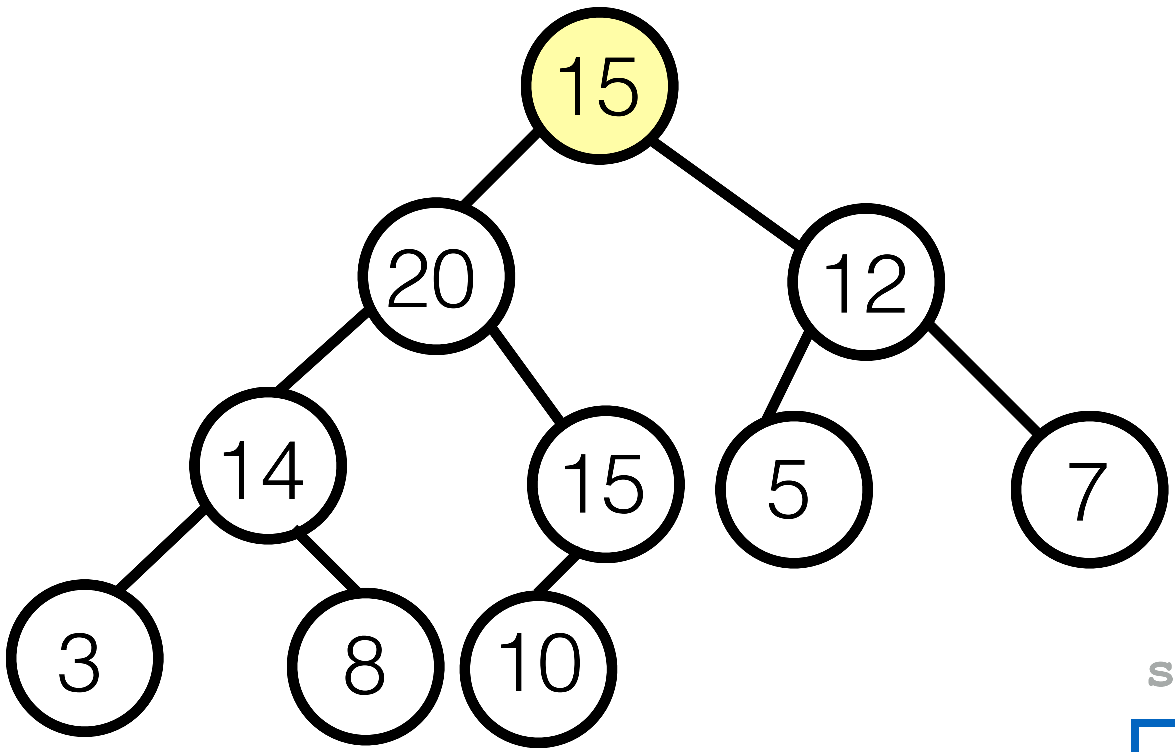


k=1

self.count = 10

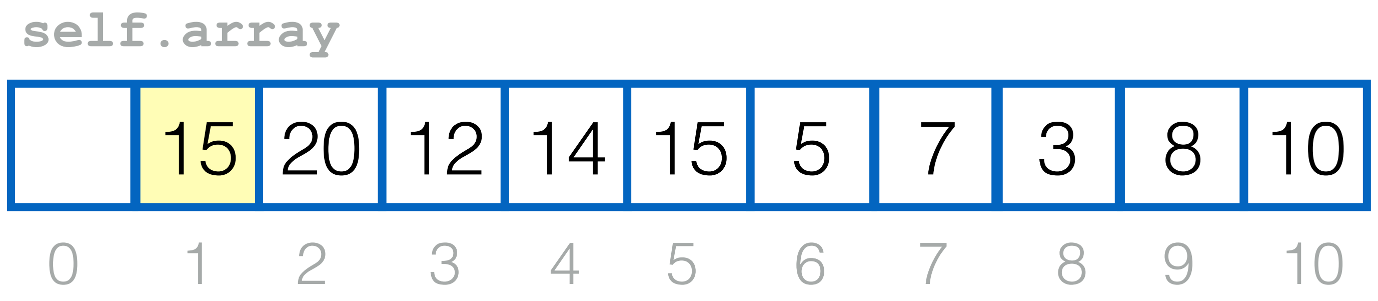


```python
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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
```



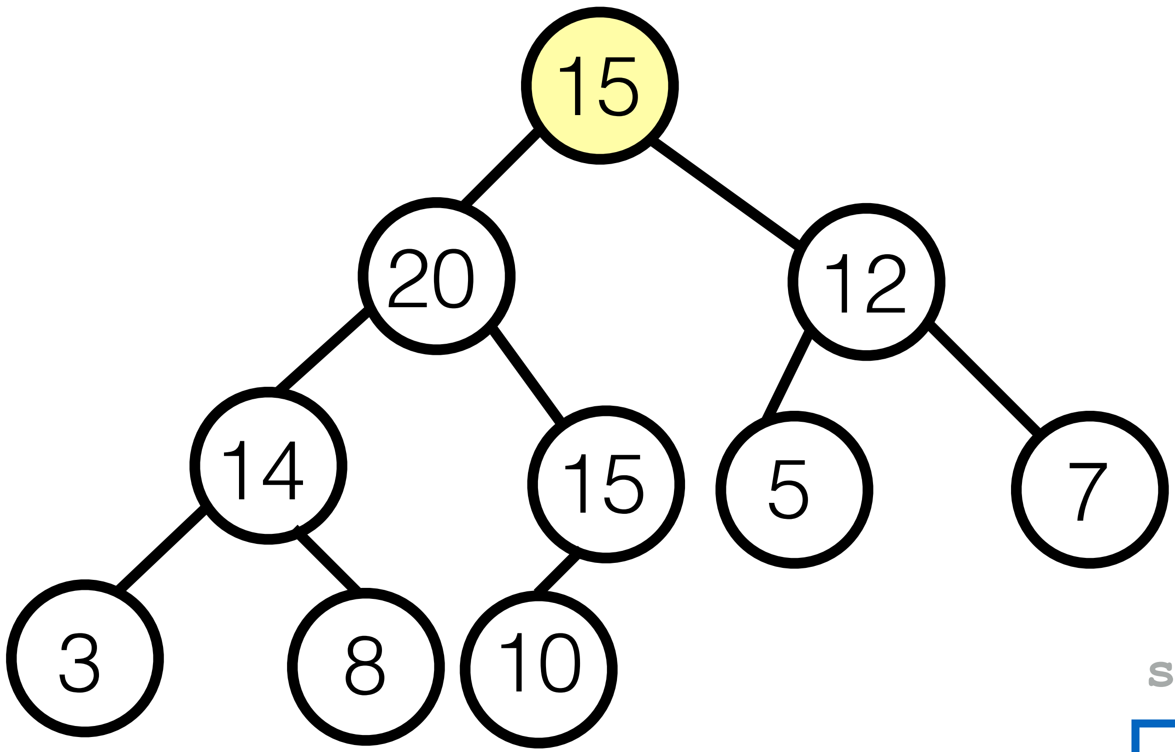
k=1

self.count = 10



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

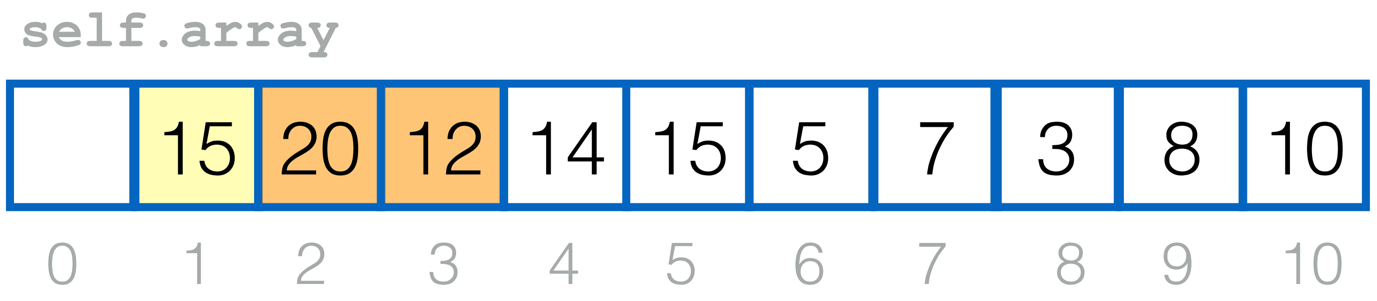
```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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
```



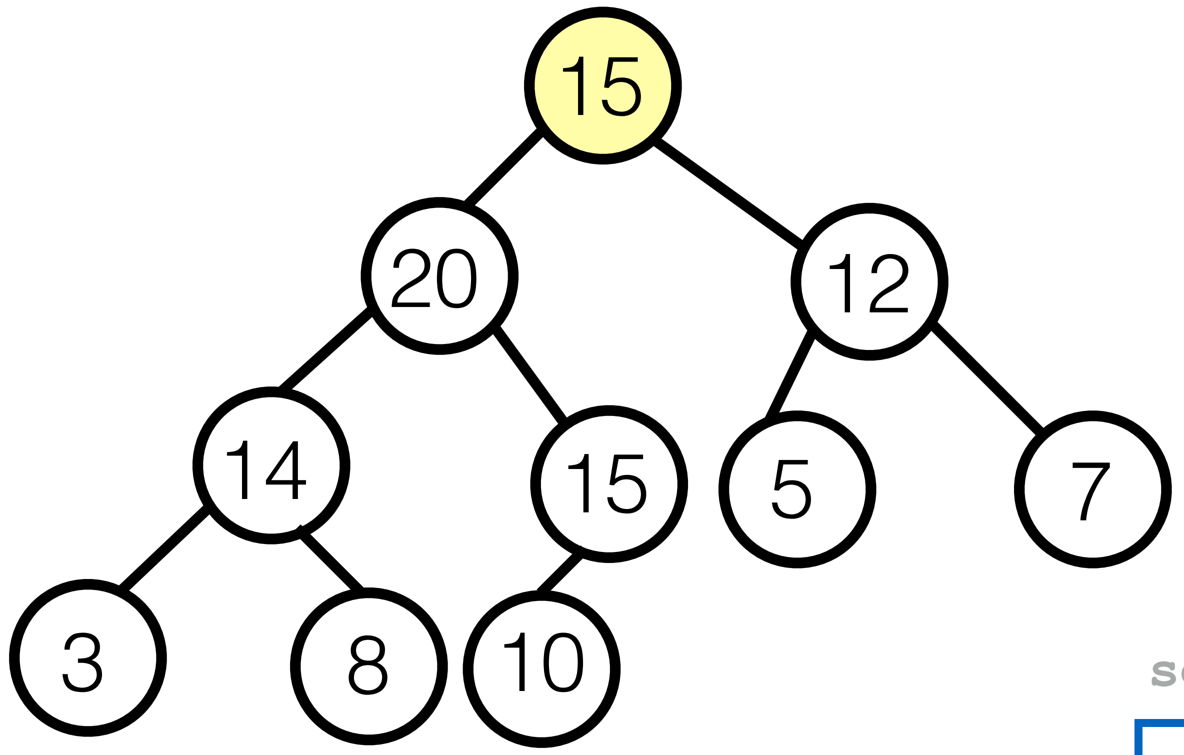
child = 2

k=1

self.count = 10



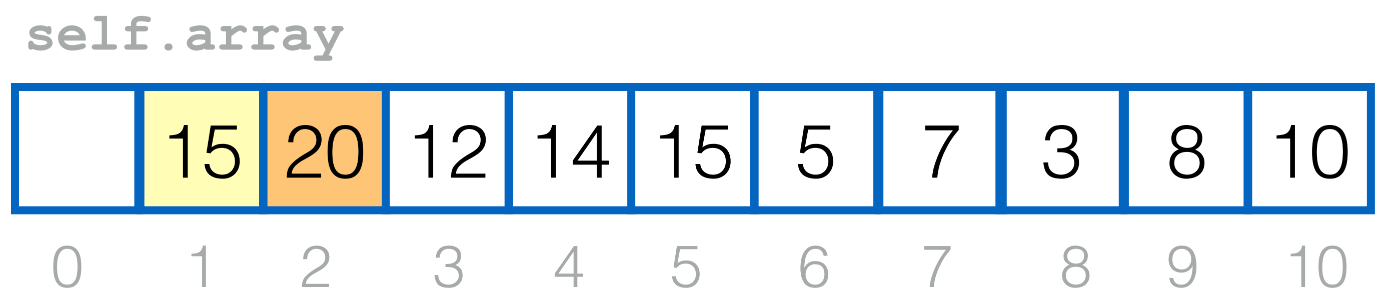
```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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 = 2

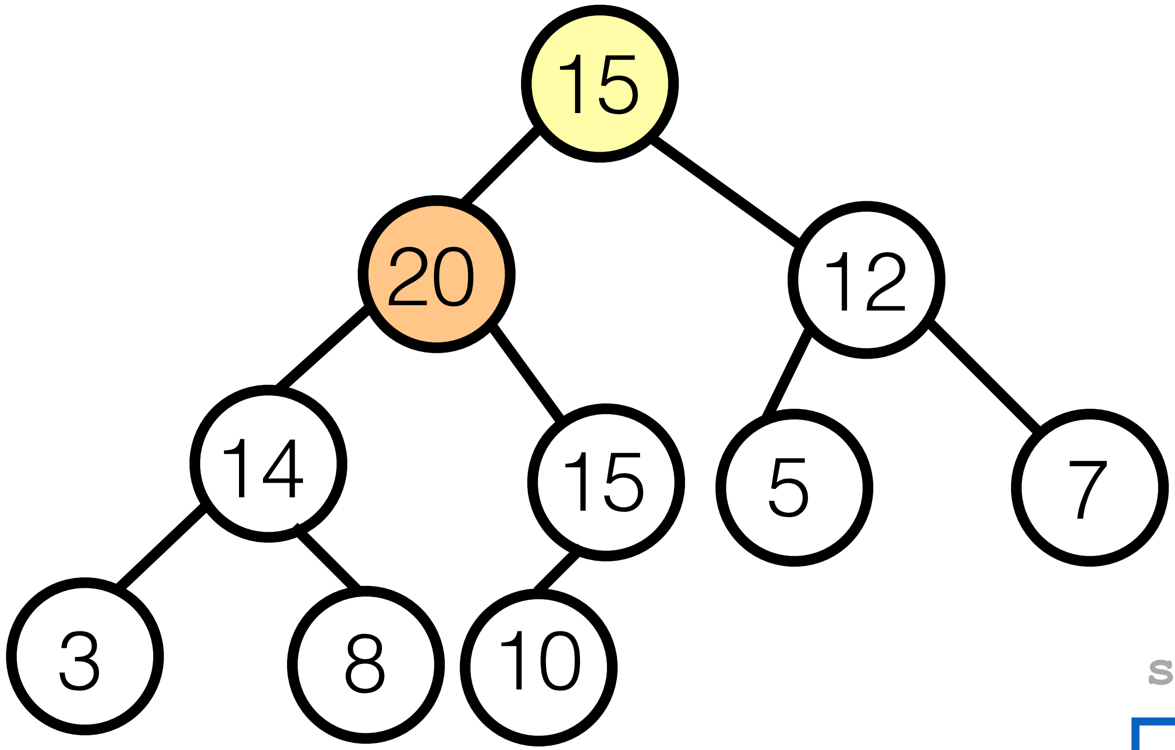
k=1

self.count = 10



	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
```

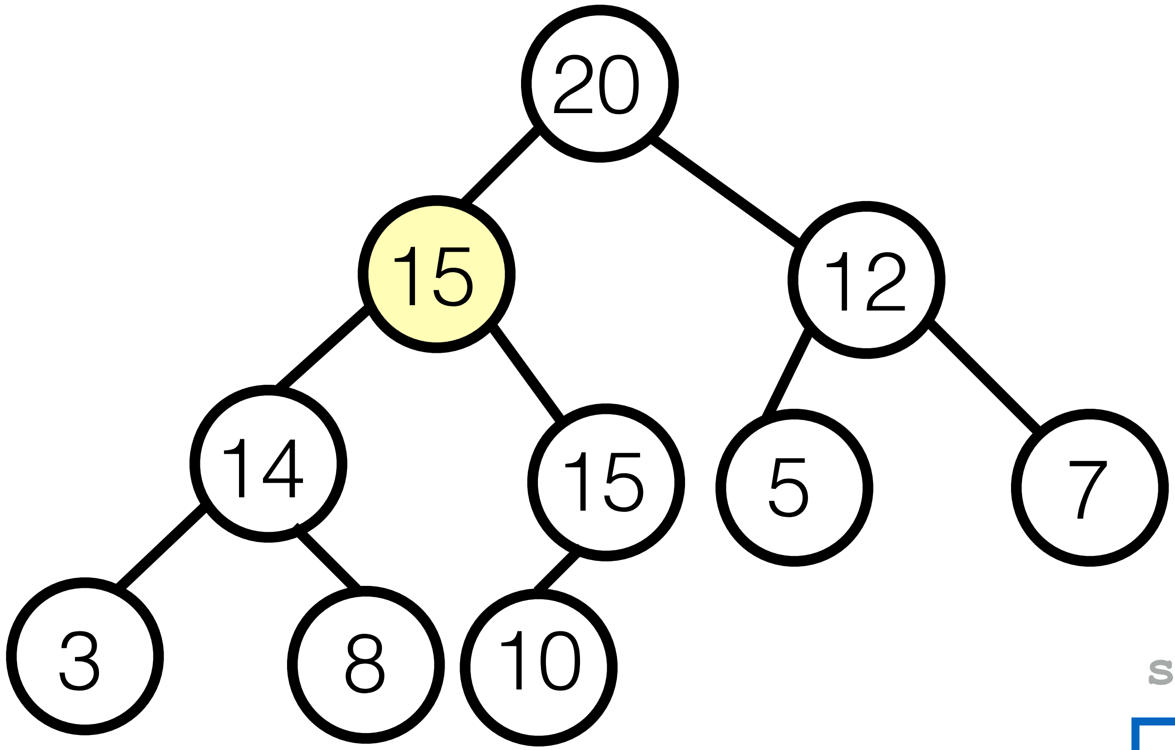


```
child =  2
k=1
self.count = 10
```

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 = 2

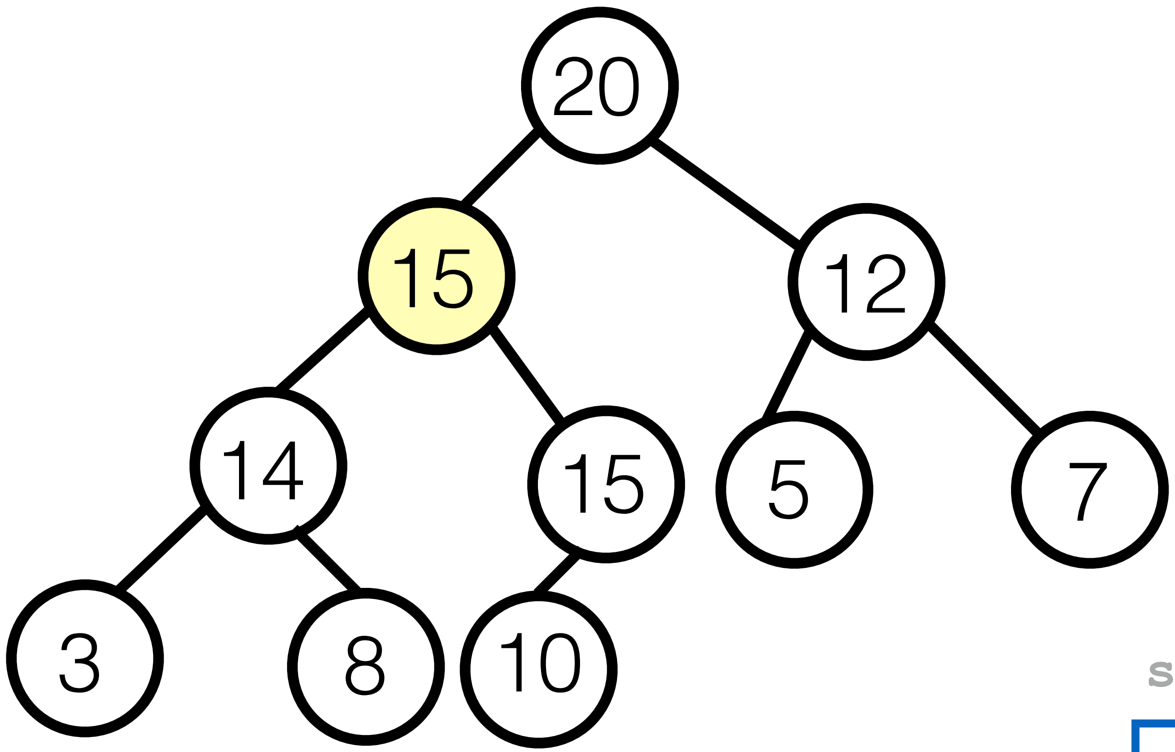
k=1

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

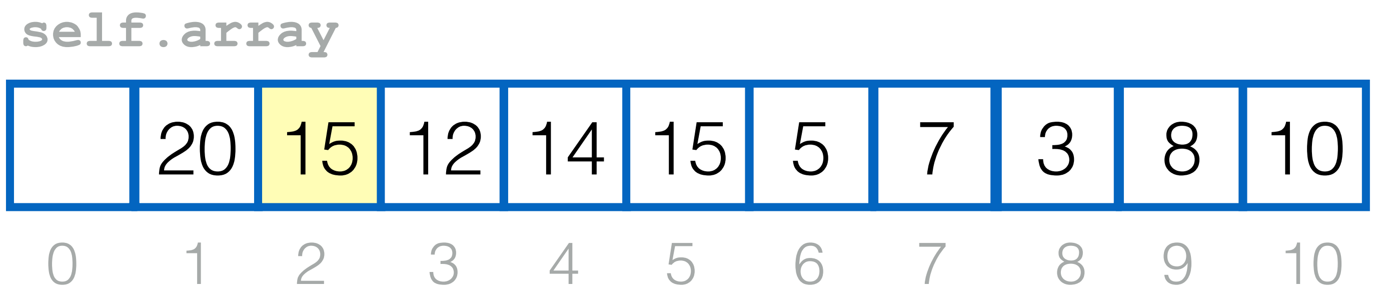
```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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
```



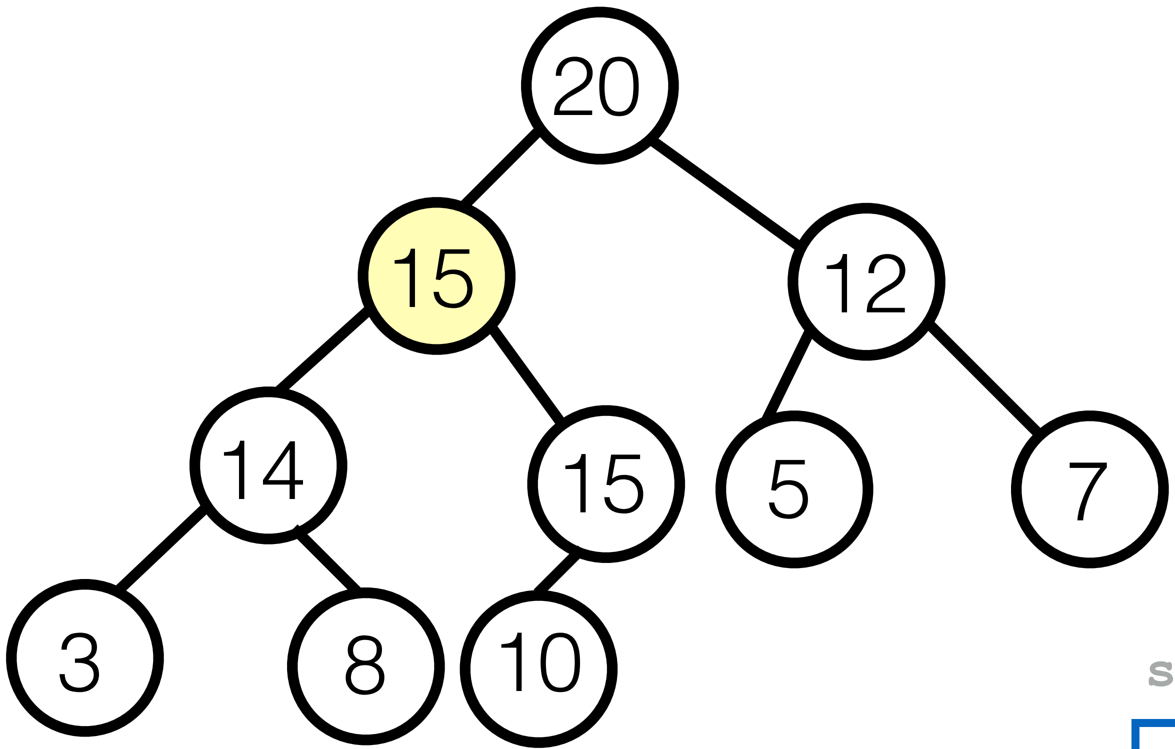
child = 2

k = 2

self.count = 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 = 2

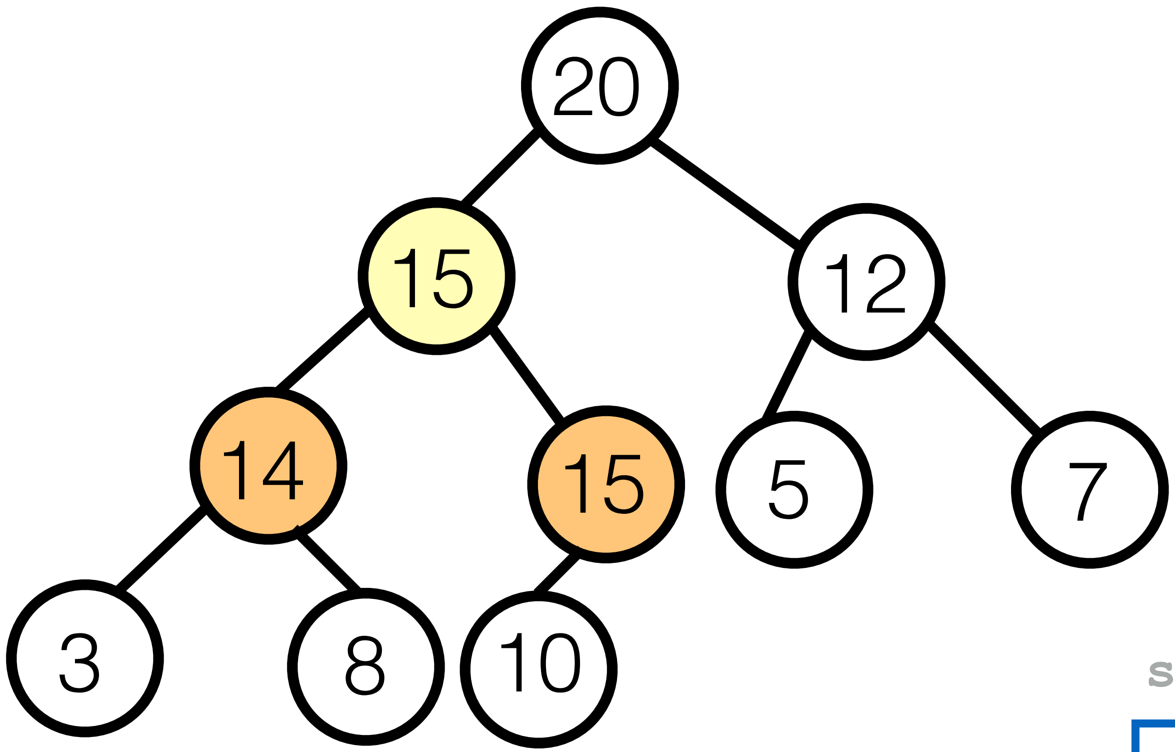
k = 2

self.count = 10

self.array

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

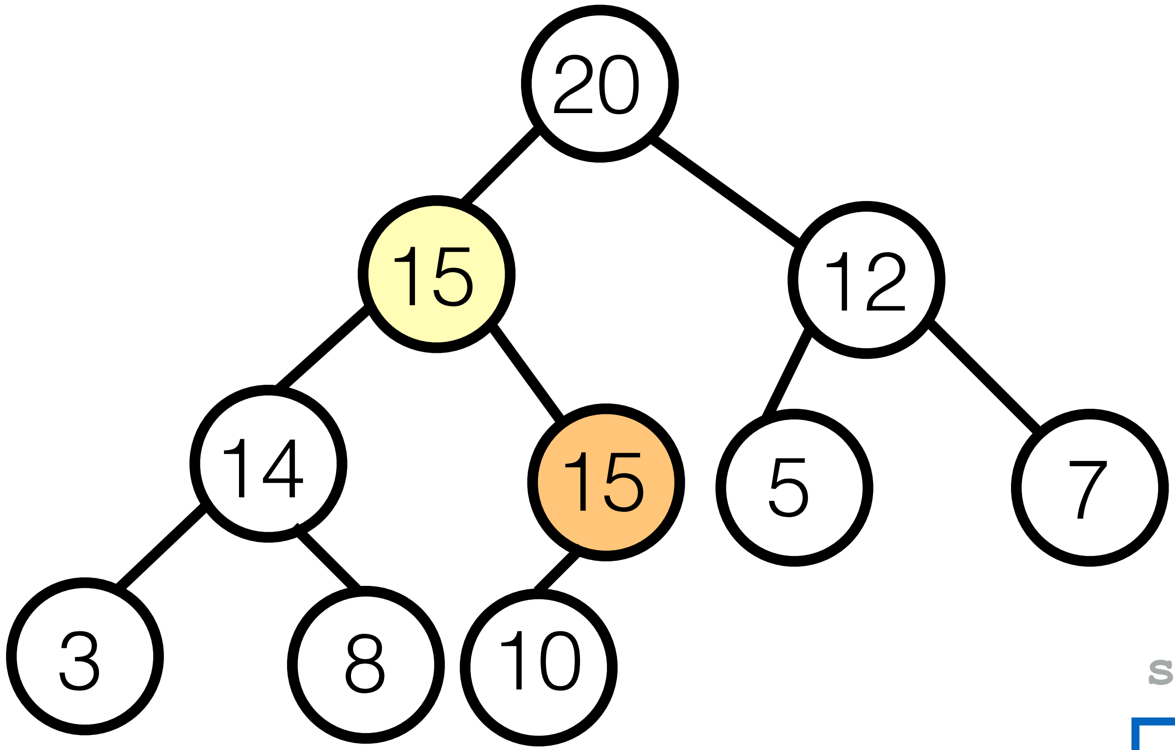
k = 2

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

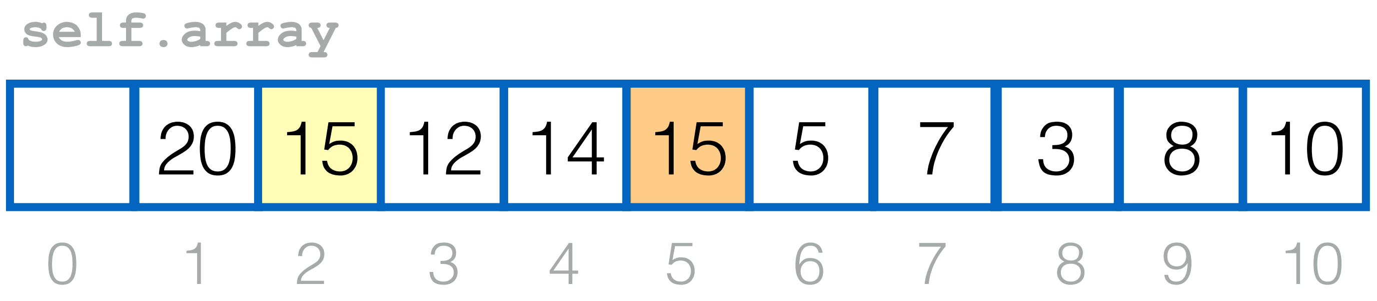
```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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
```



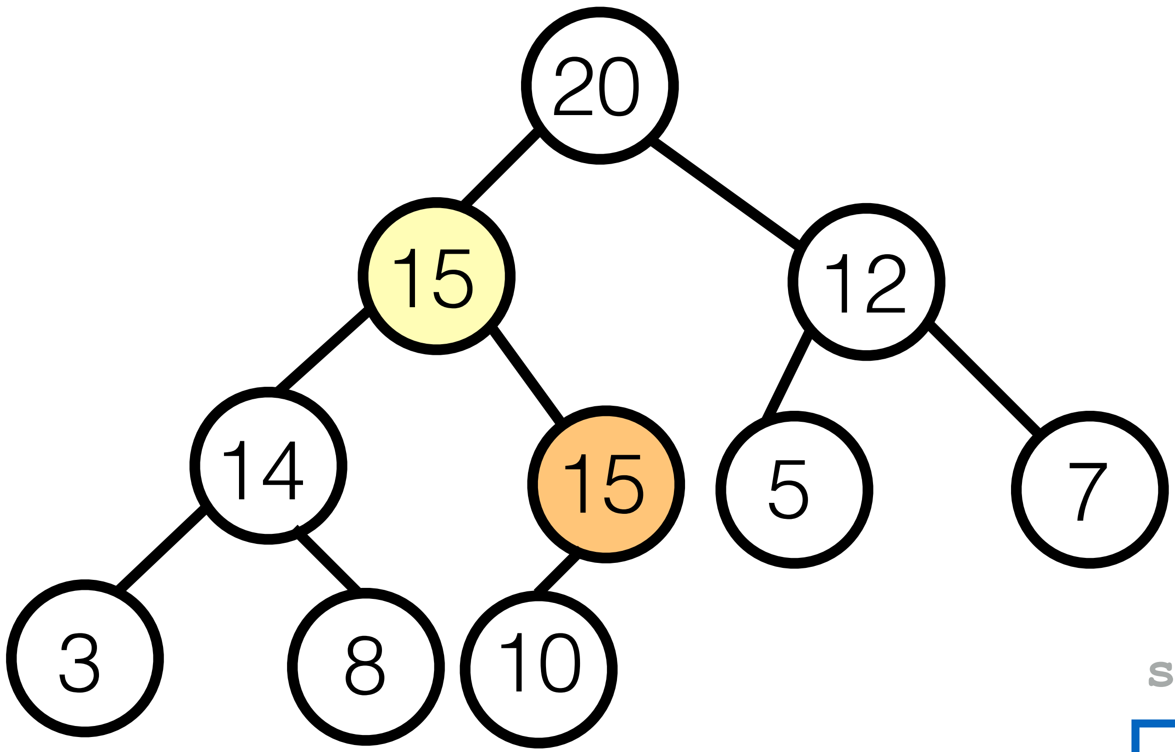
child = 5

k = 2

self.count = 10



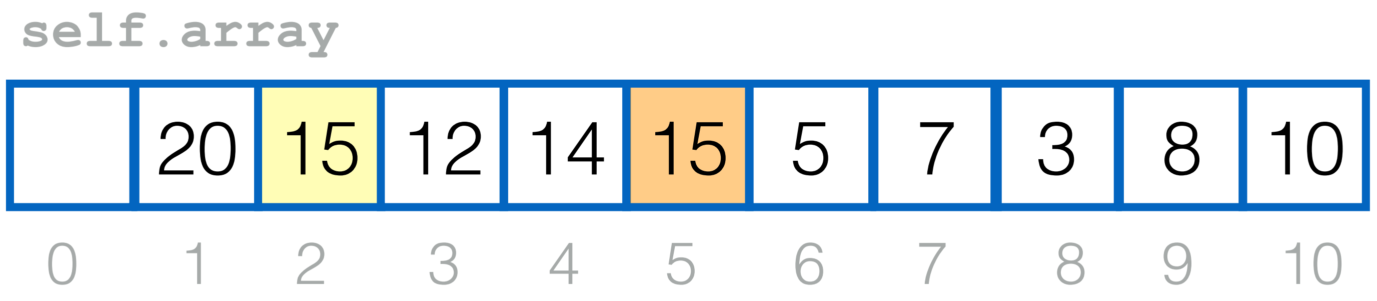
```
def largest_child(self, k):
    # Check for only one child.
    if 2*k == self.count or self.array[2*k] > self.array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

# 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
```



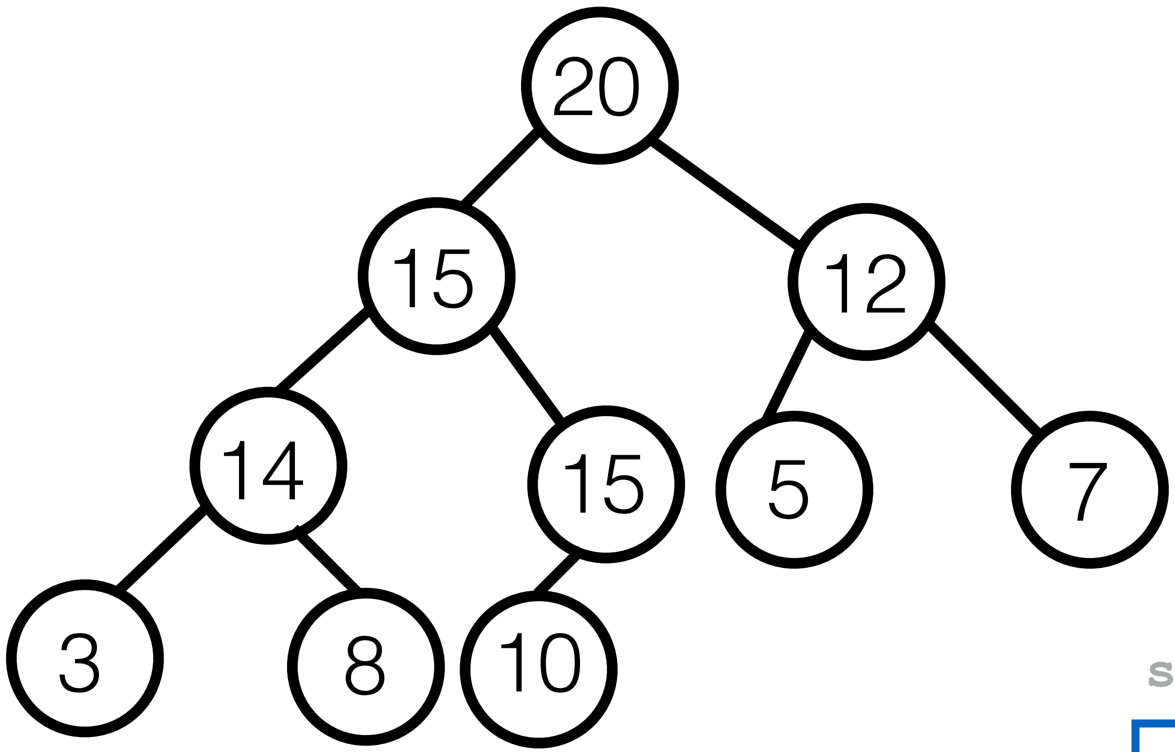
child = 5

k = 2

self.count = 10



```
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

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

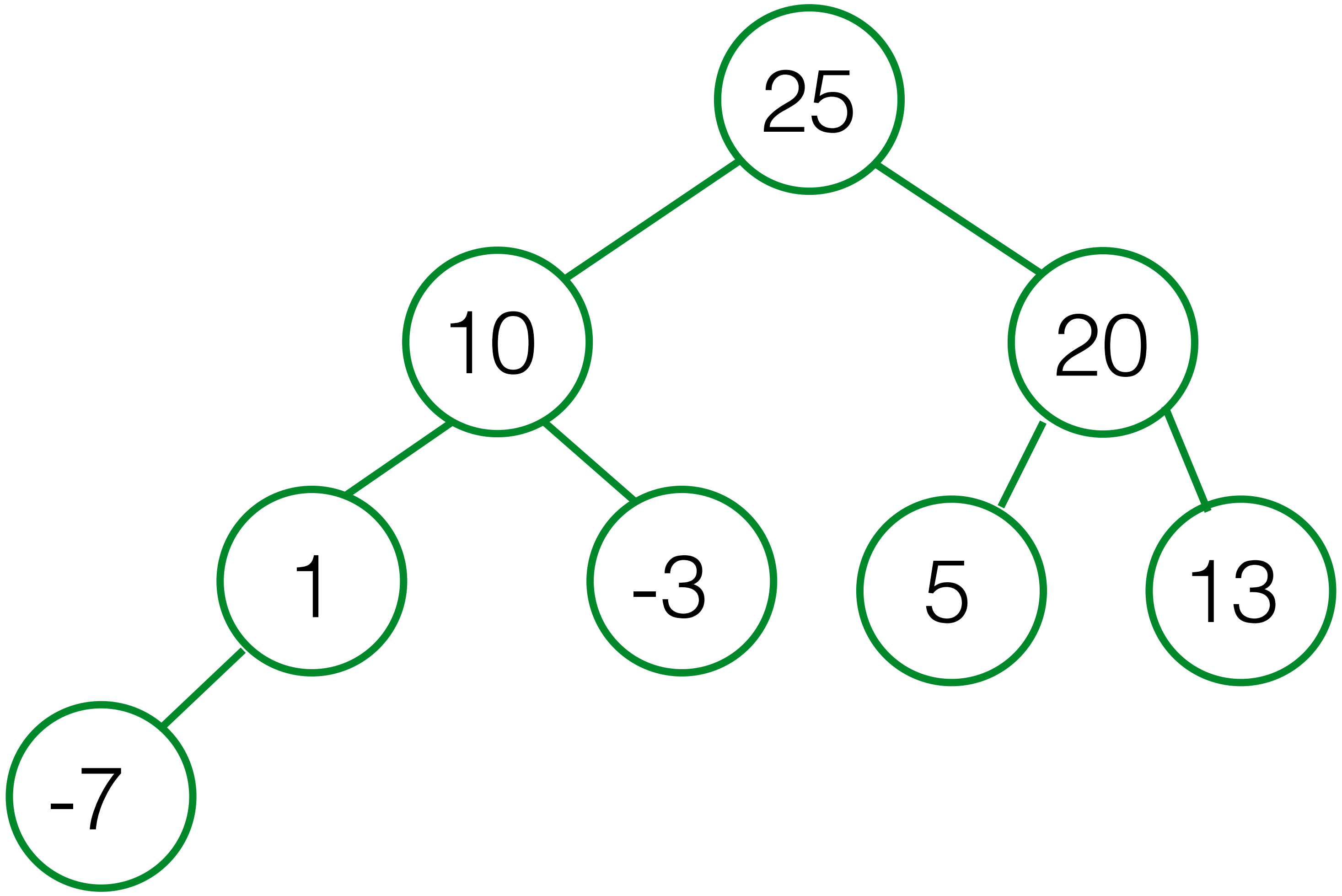
Homework: (key, value)

```
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
```

Just the key…

Heap sort

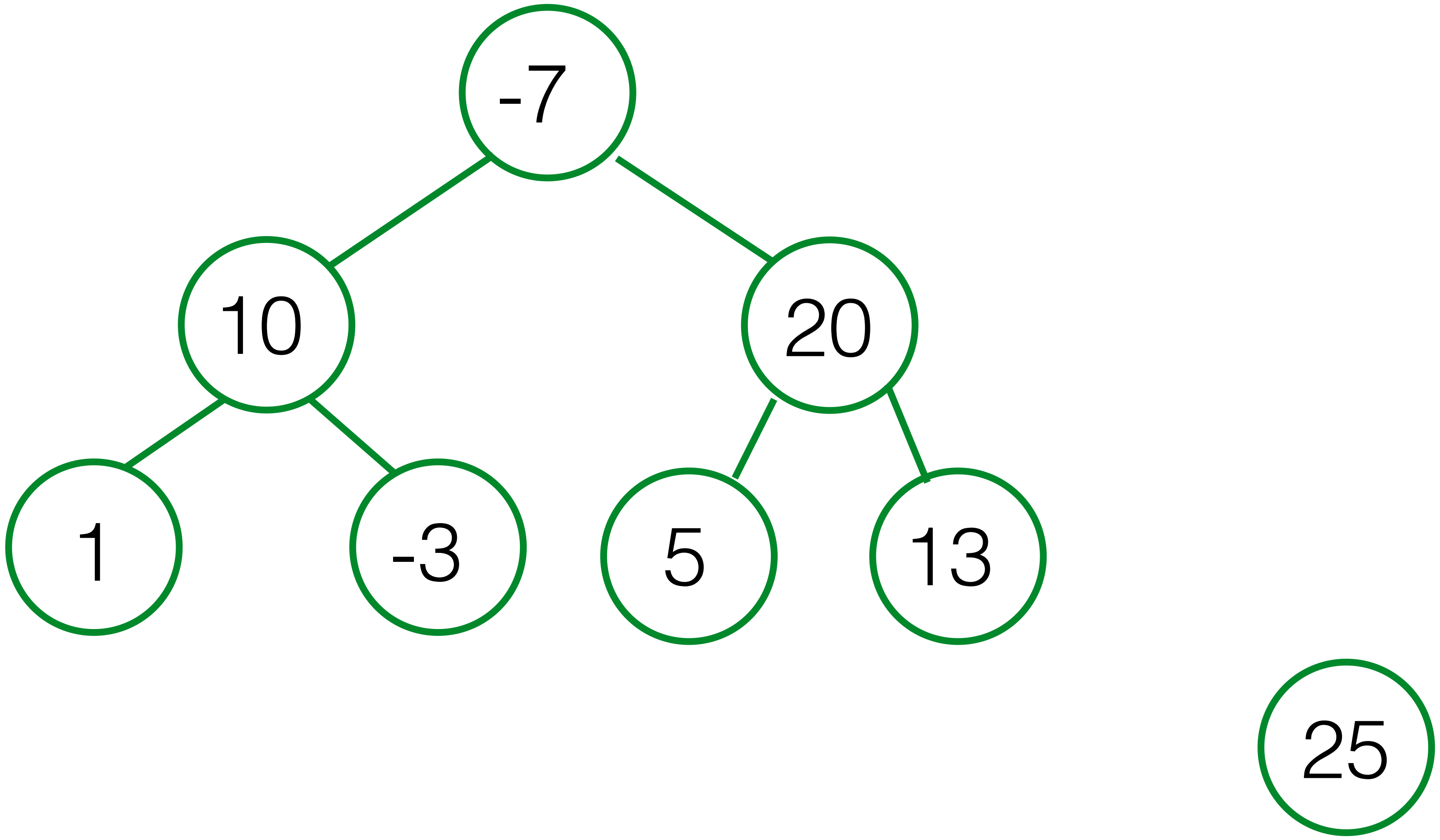
[5, -7, 10, -3, 13, 20, 25,1]



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

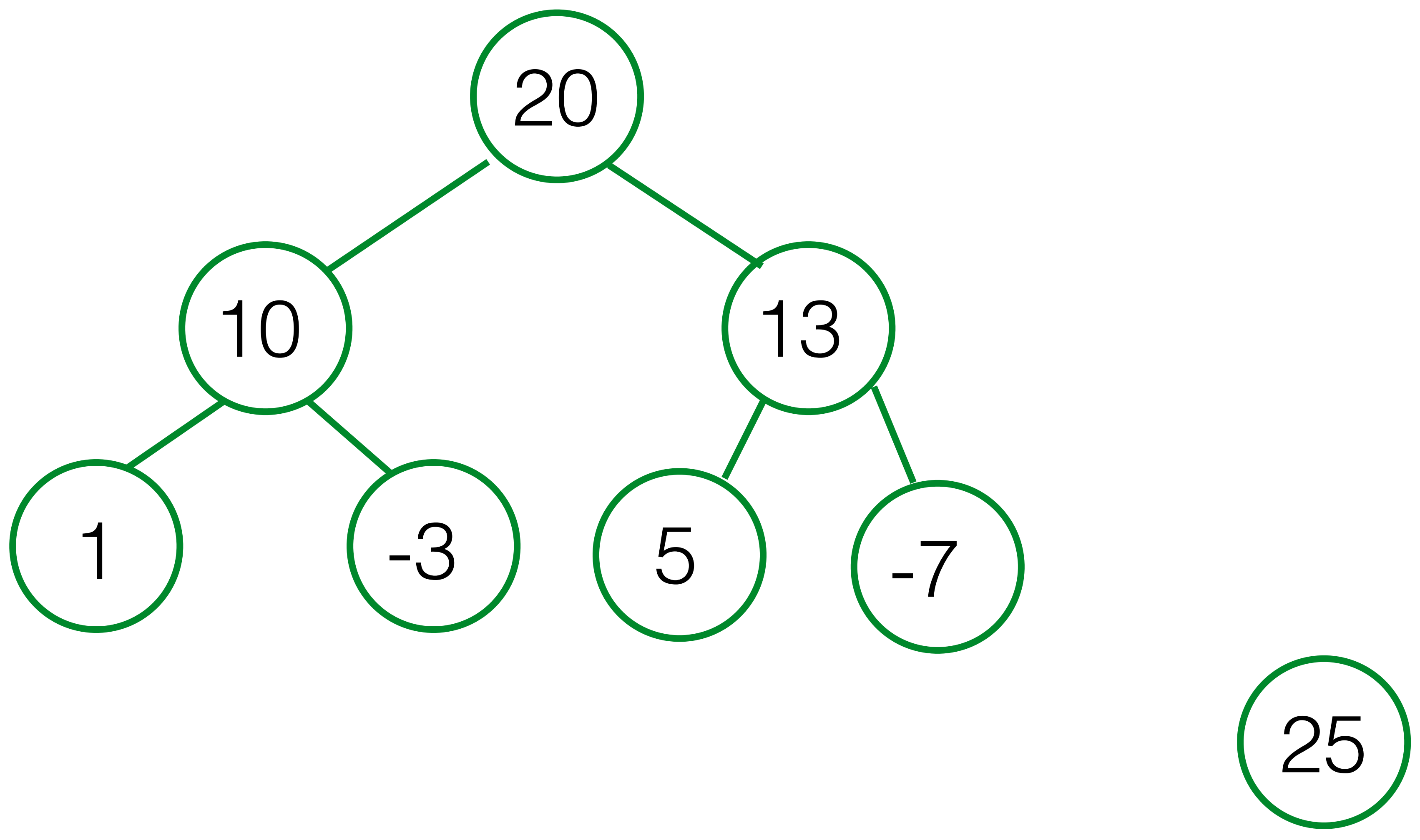
[5, -7, 10, -3, 13, 20, 25,1]

not a heap



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

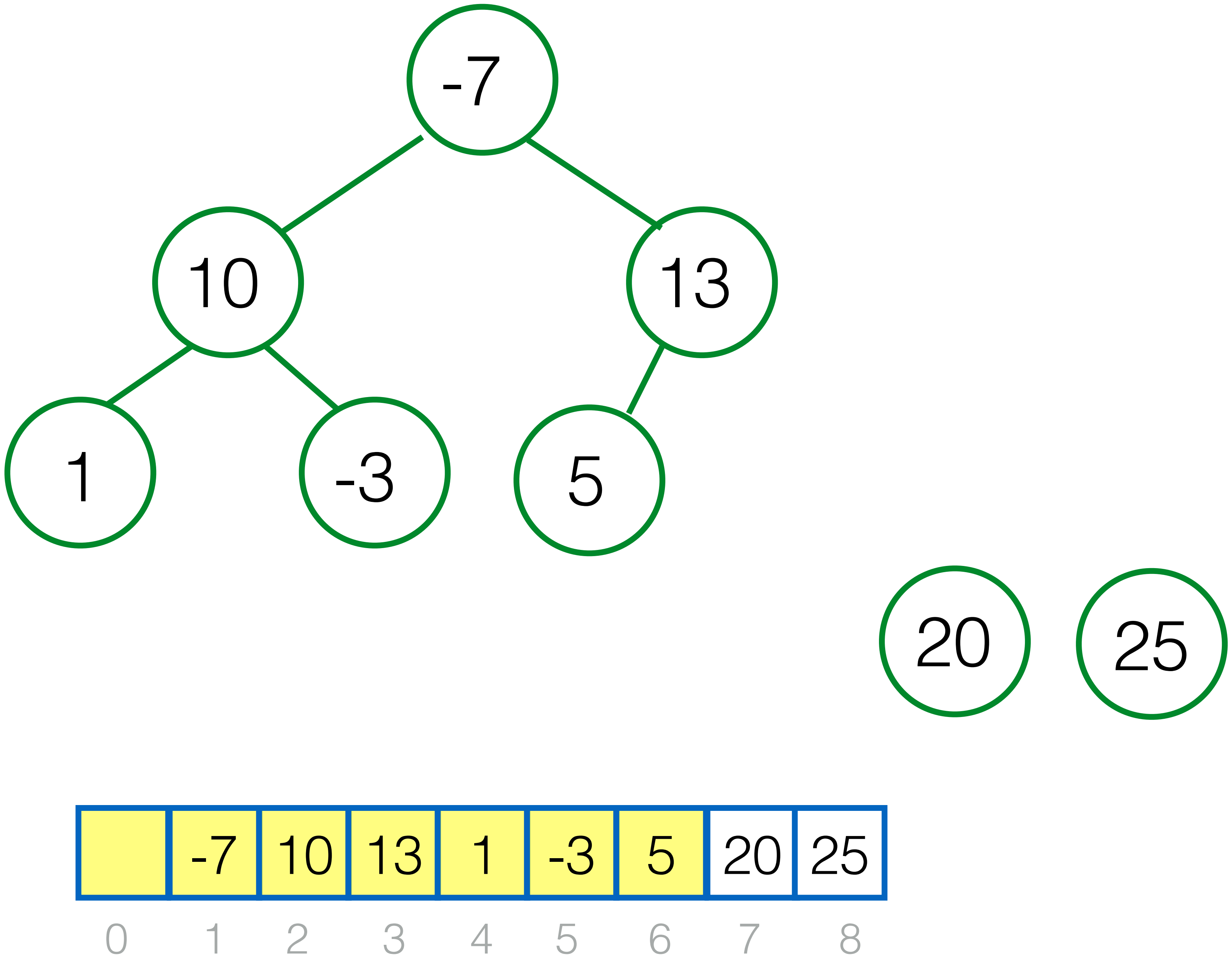
[5, -7, 10, -3, 13, 20, 25,1]



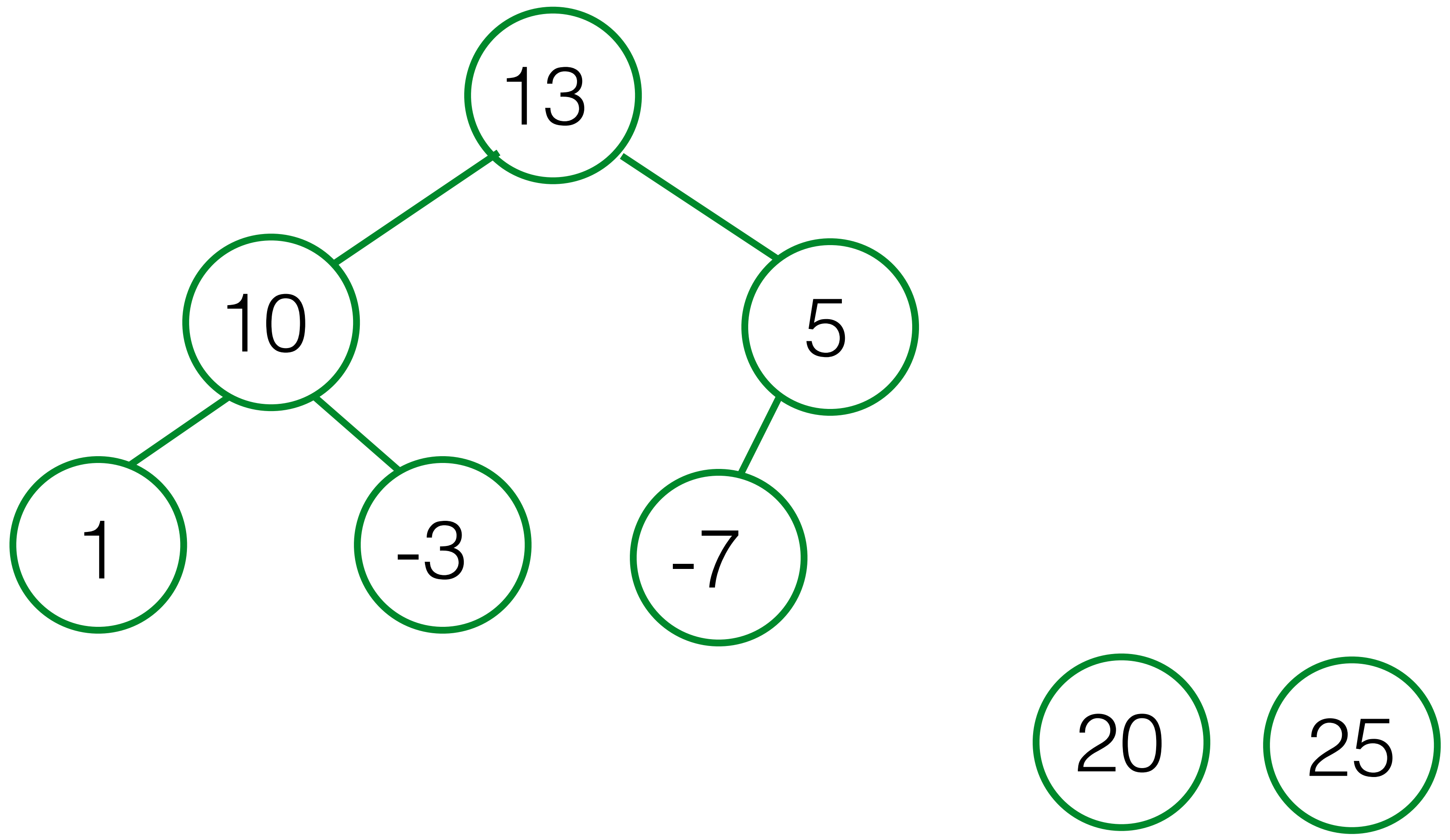
	20	10	13	1	-3	5	-7	25
0	1	2	3	4	5	6	7	8

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

not a heap



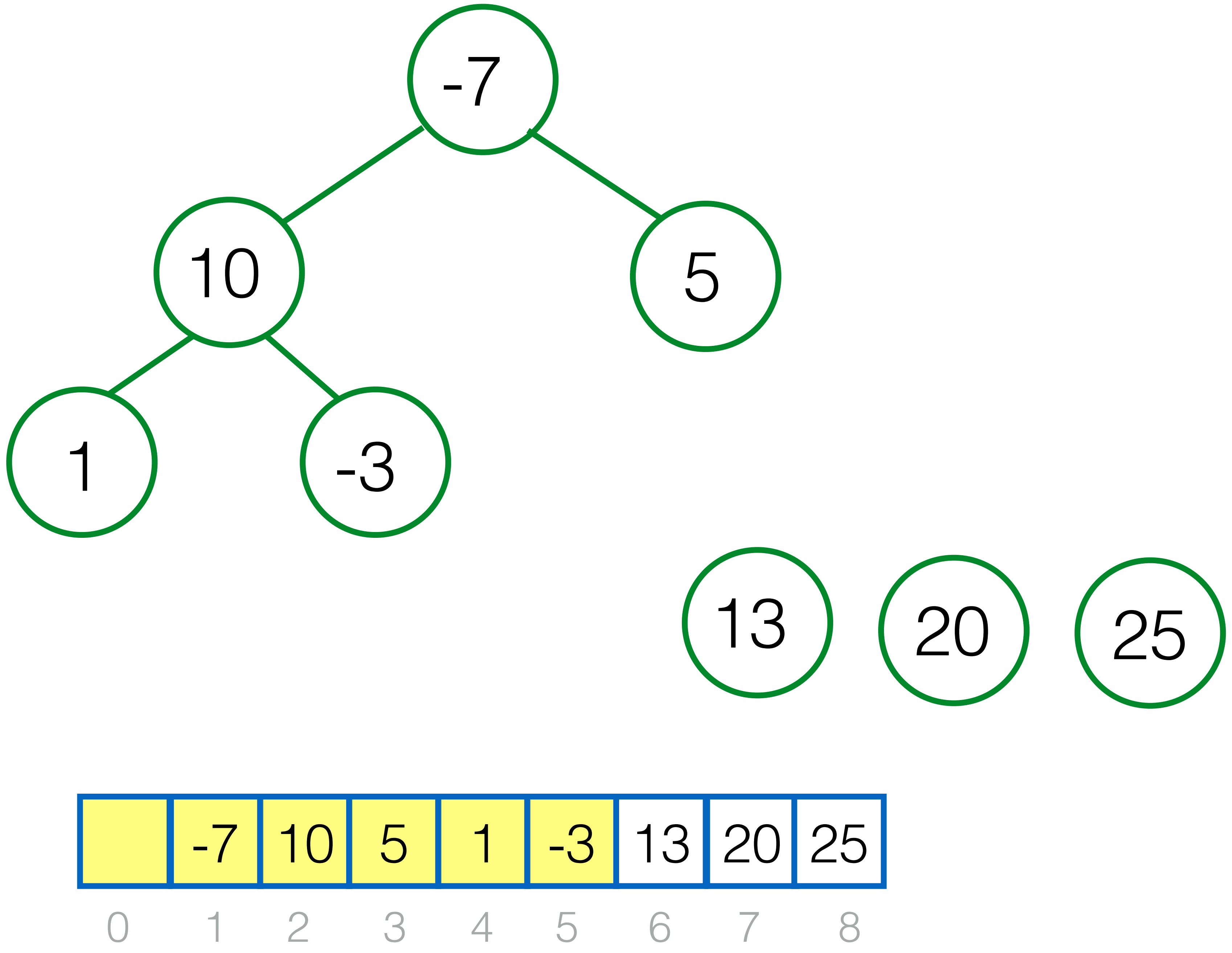
[5, -7, 10, -3, 13, 20, 25,1]



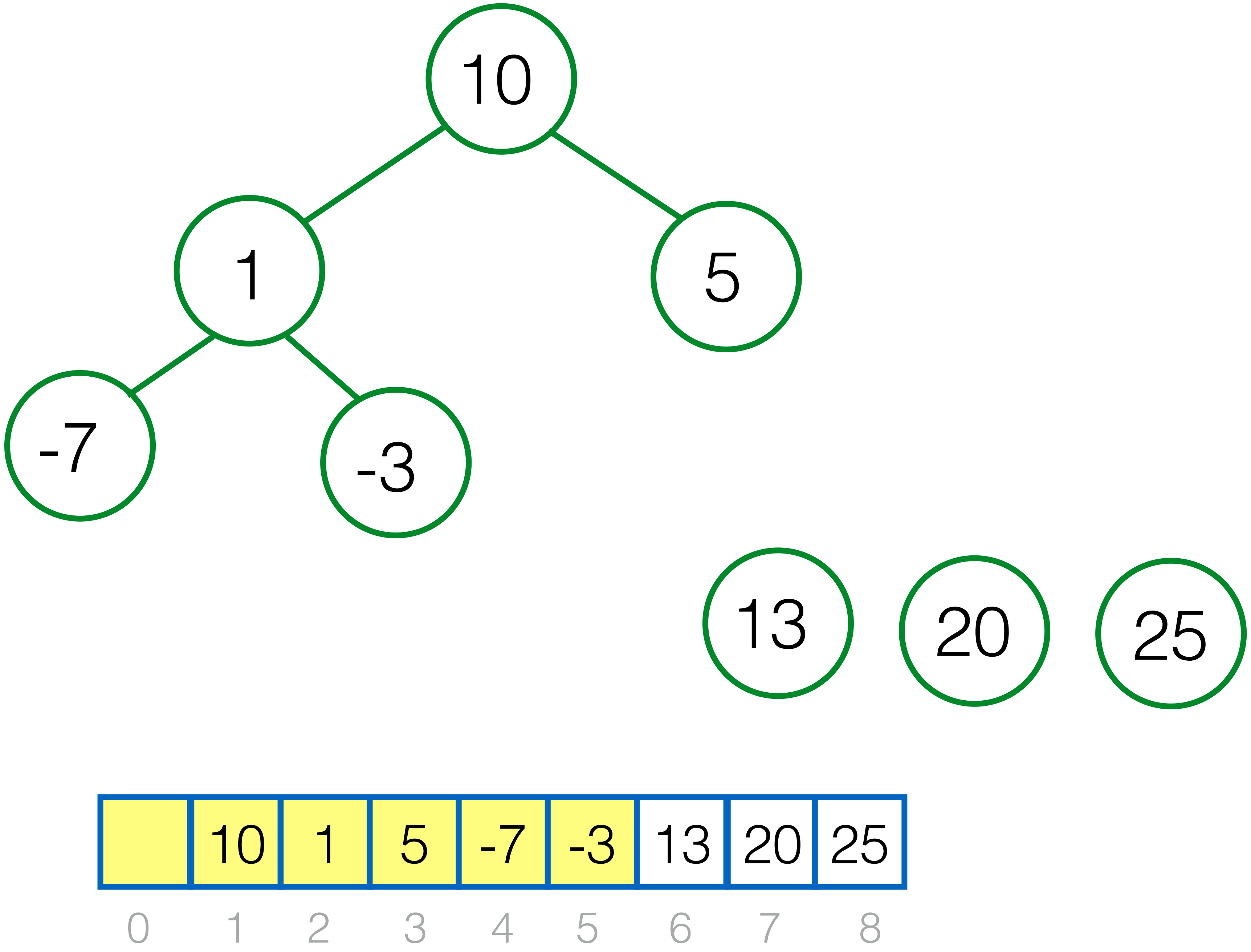
0	1	2	3	4	5	6	7	8
	13	10	5	1	-3	-7	20	25

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

not a heap

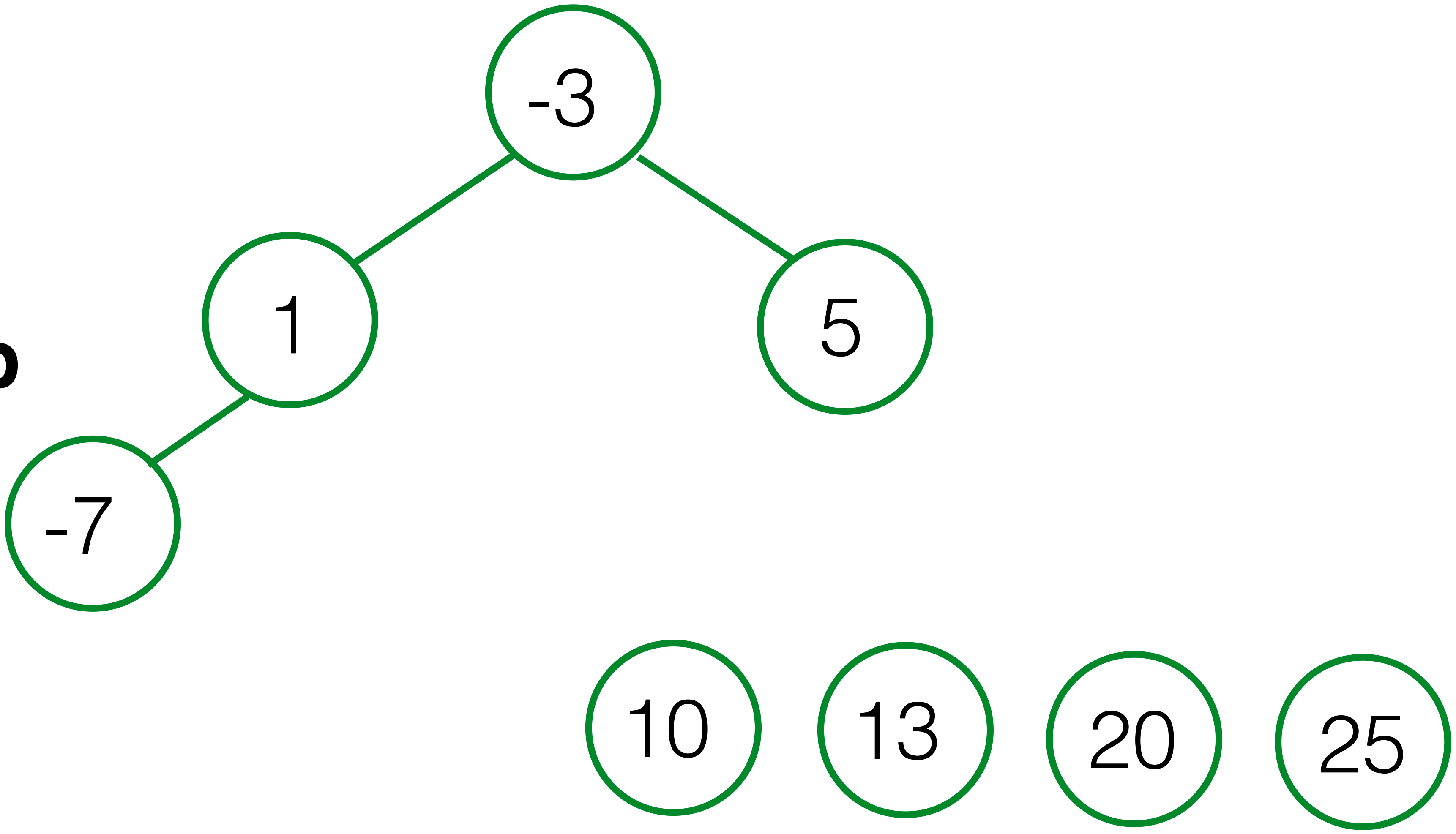


[5, -7, 10, -3, 13, 20, 25,1]



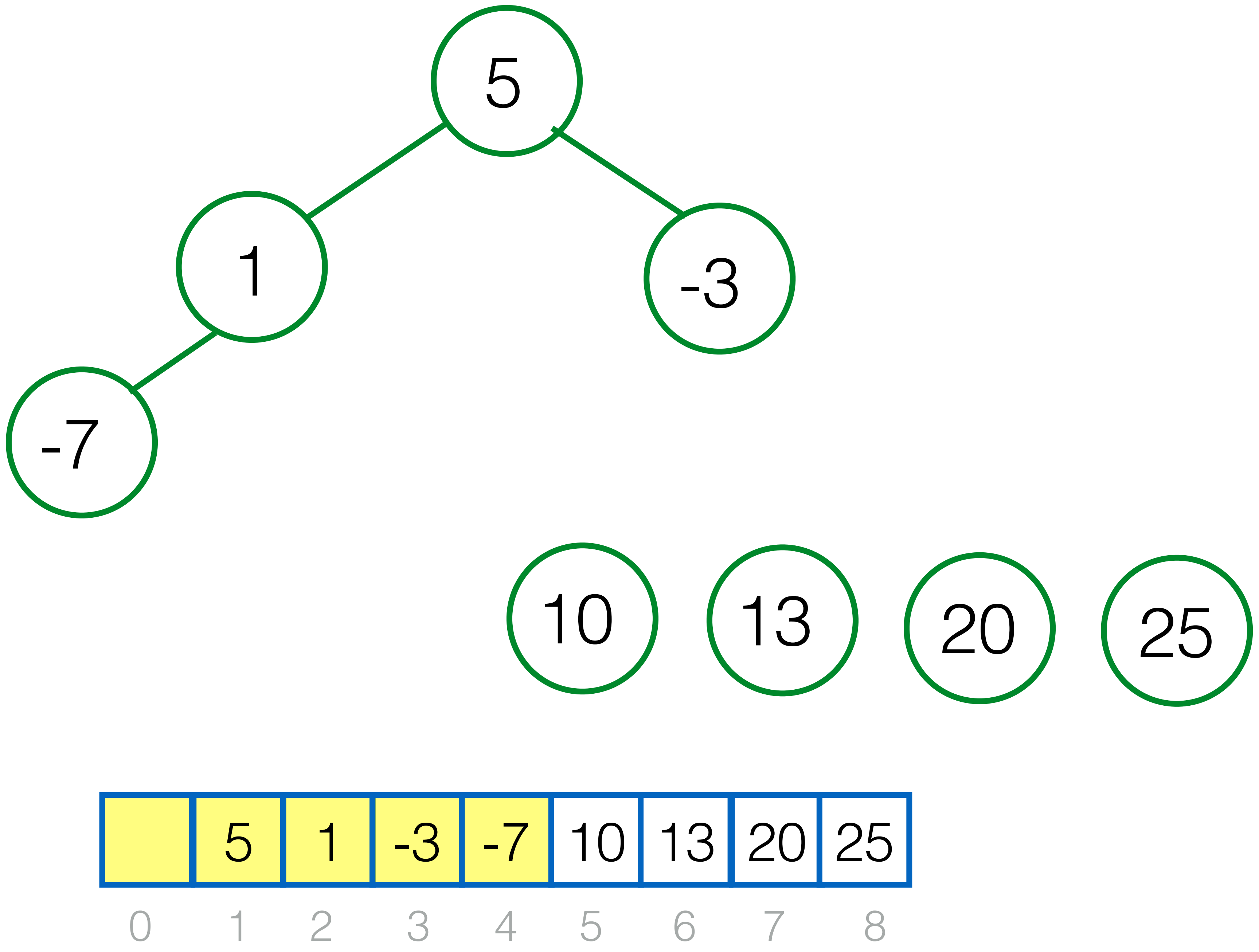
[5, -7, 10, -3, 13, 20, 25,1]

not a heap



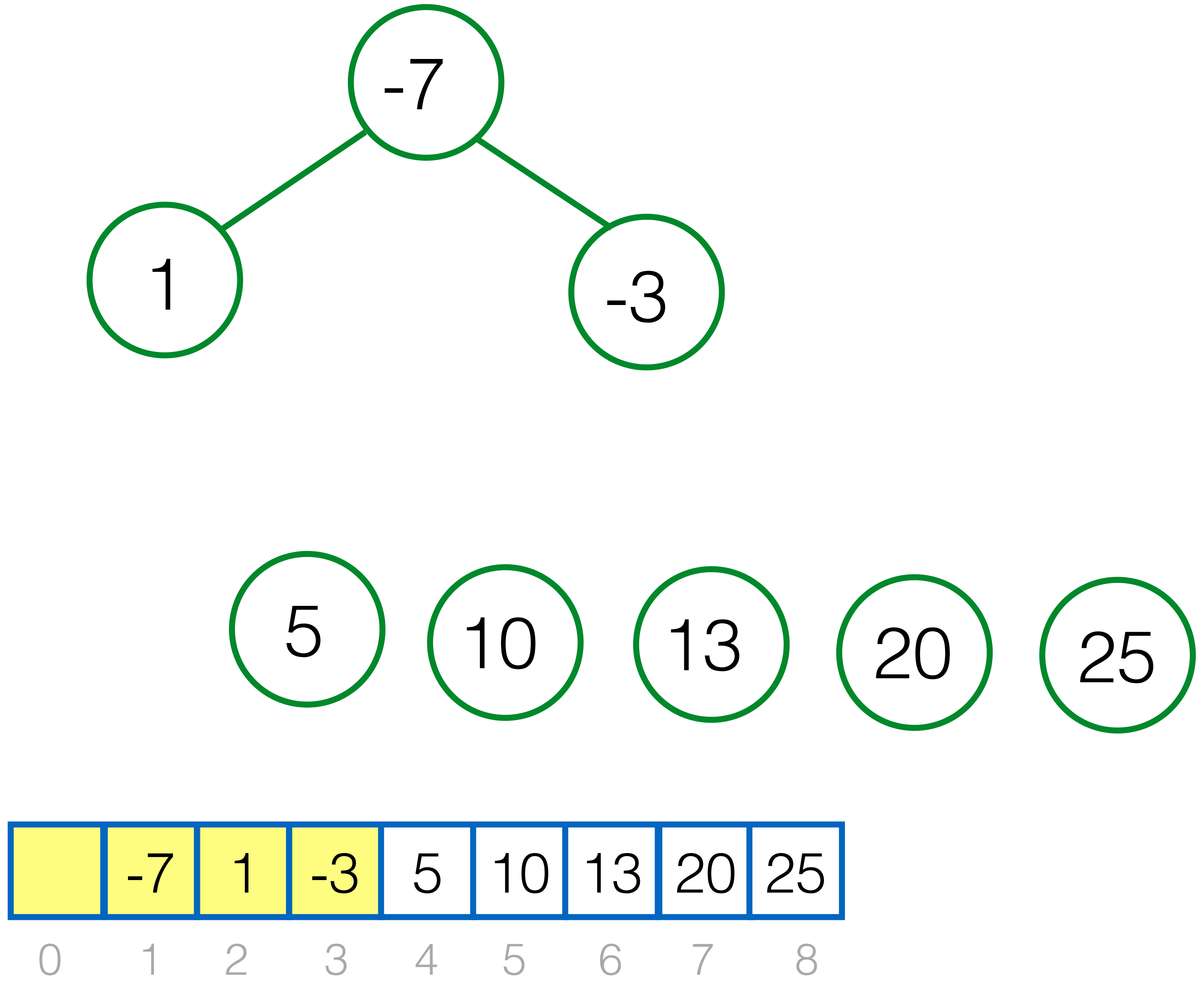
0	1	2	3	4	5	6	7	8
	-3	1	5	-7	10	13	20	25

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

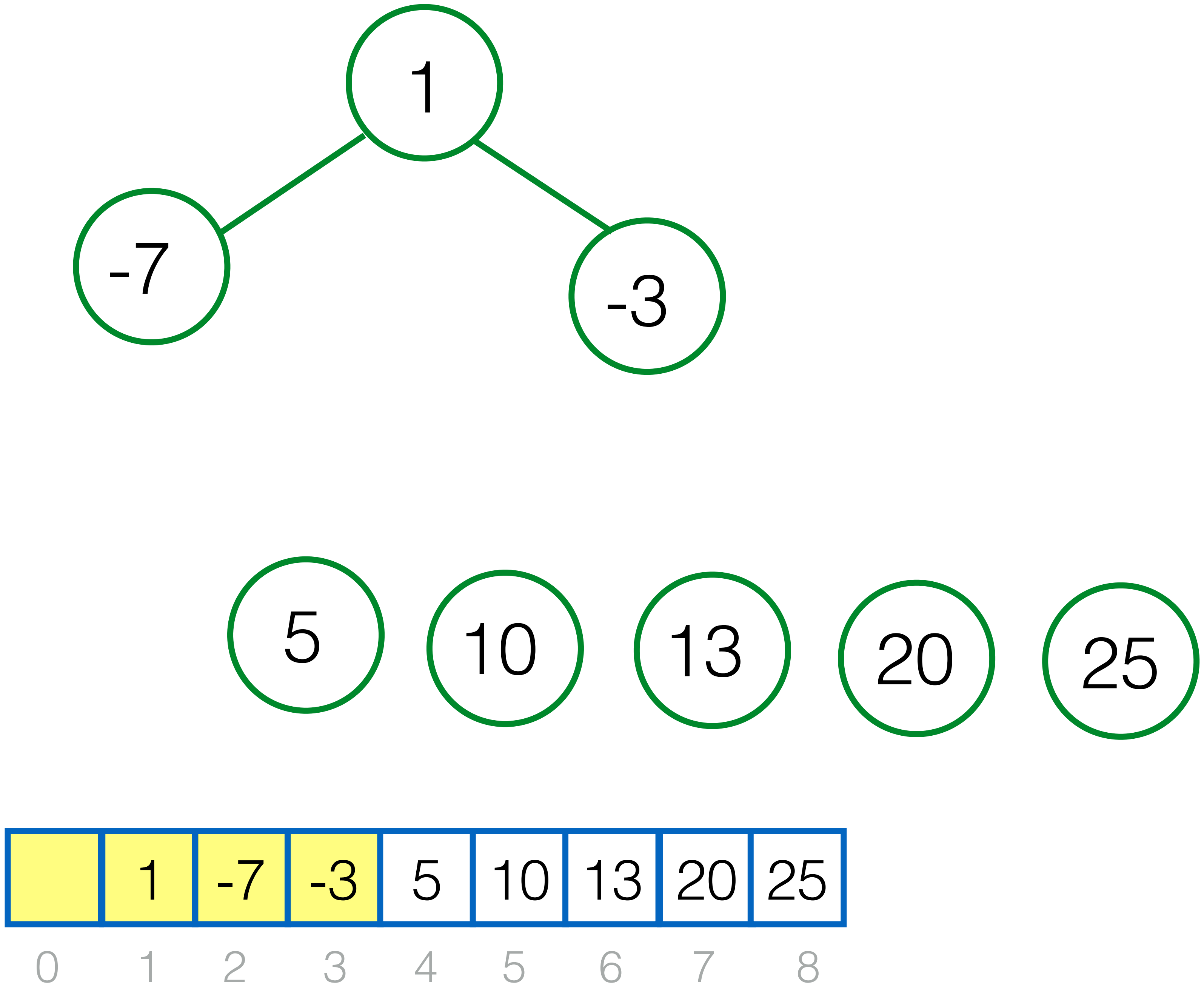


[5, -7, 10, -3, 13, 20, 25,1]

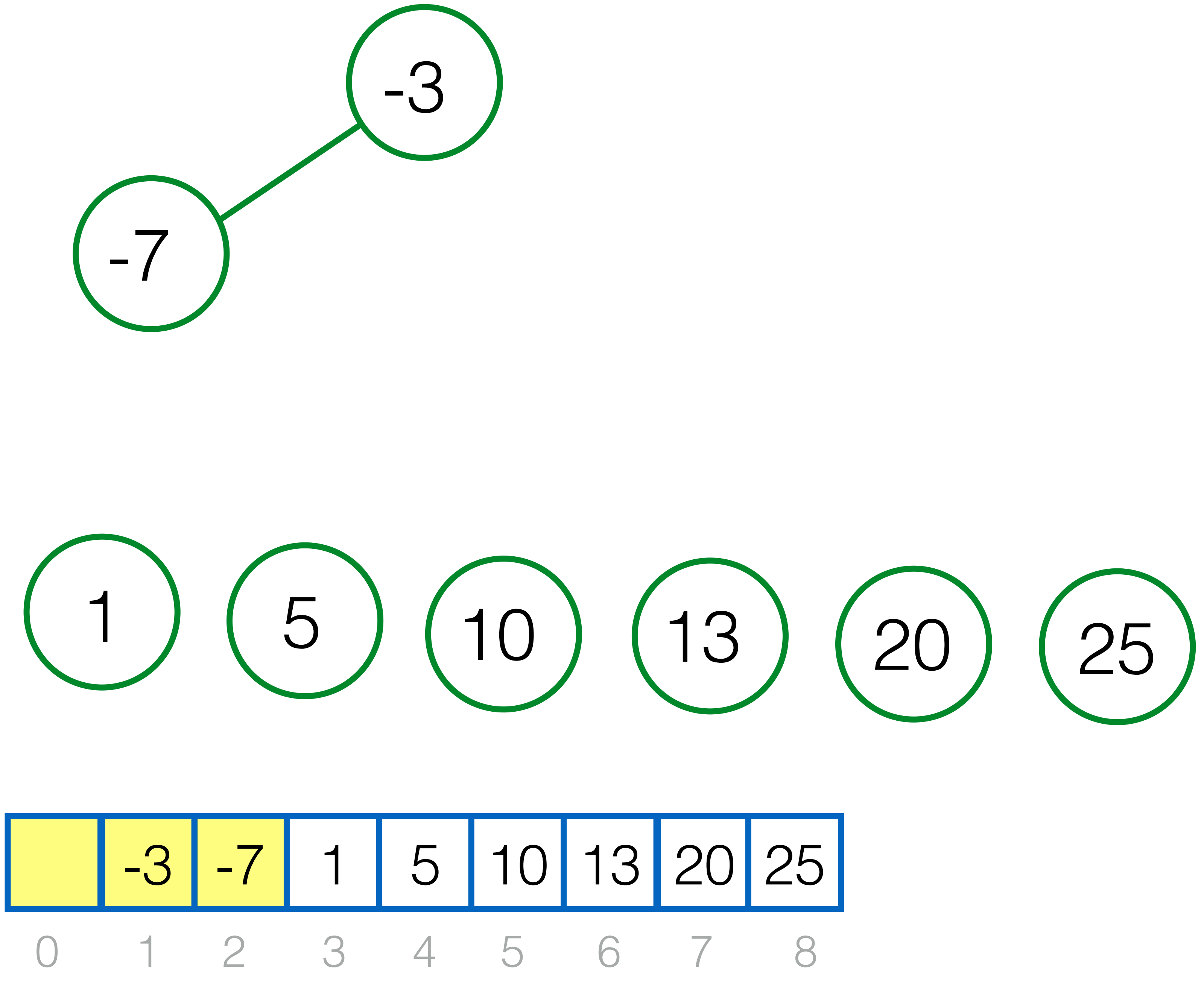
not a heap



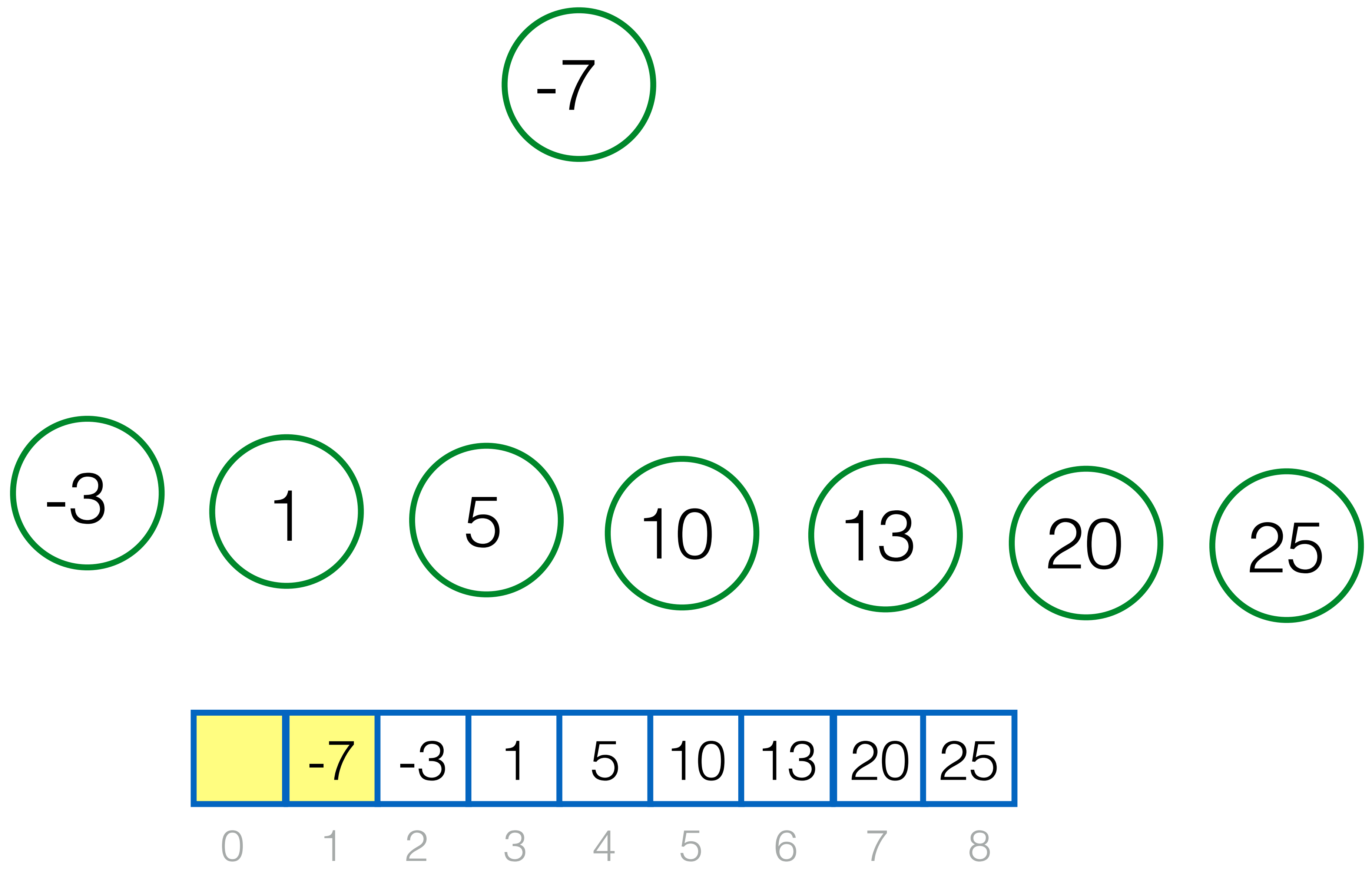
[5, -7, 10, -3, 13, 20, 25, 1]



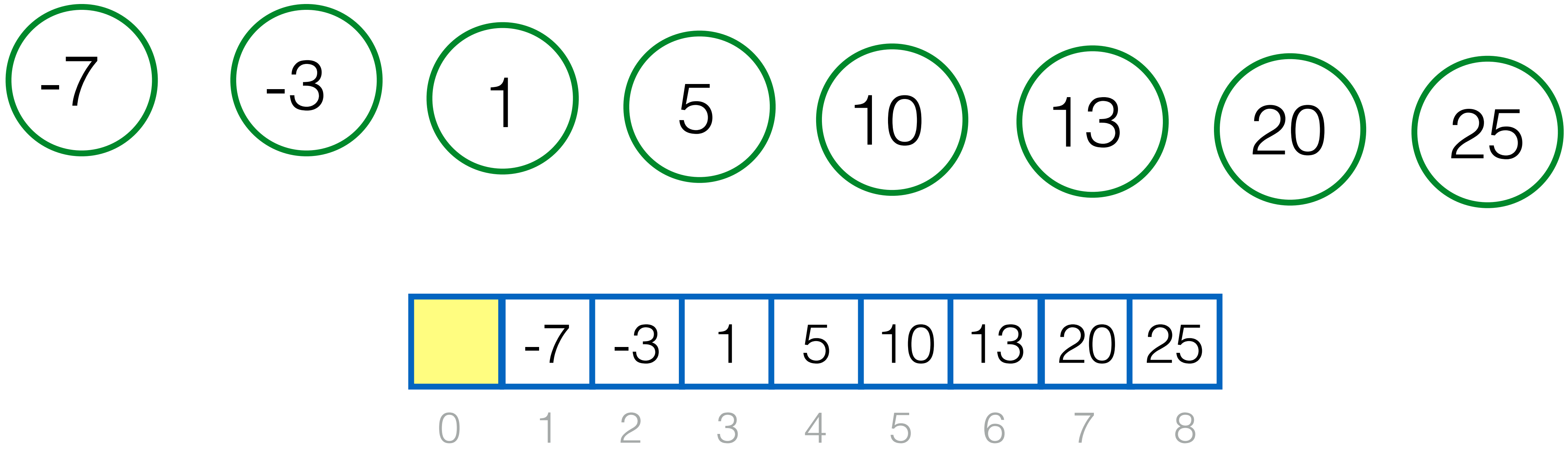
[5, -7, 10, -3, 13, 20, 25,1]



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



[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

Heap sort

$$\left.\begin{array}{l}\bullet\quad\text{For each element in the array:}\\ \qquad ➡\quad \text{Add it to the Heap: O(log(N))}\end{array}\right\}\rightarrow \textbf{N times}$$

$$\left.\begin{array}{l}\bullet\quad\text{While heap contains elements:}\\ ➡\quad \text{Get max item: O(log(N))}\end{array}\right\}\rightarrow \textbf{N times}$$

$$\begin{aligned}
\log n! &= \log(1 \cdot 2 \cdot 3 \cdot \;\cdots\; \cdot n) \\
&= \log 1 + \log 2 + \log 3 + \cdots + \log n \\
&\leq \log n + \log n + \log n + \cdots + \log n \\
&= n \log n
\end{aligned}$$

Heap sort

Construction can also be done in linear time

- 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))

N times

worst case: O(N log N)

Summary

- A simple Heap implementation
    - rise
    - sink
    - largest_child
- Heap Sort