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

Lecture 35
Heaps

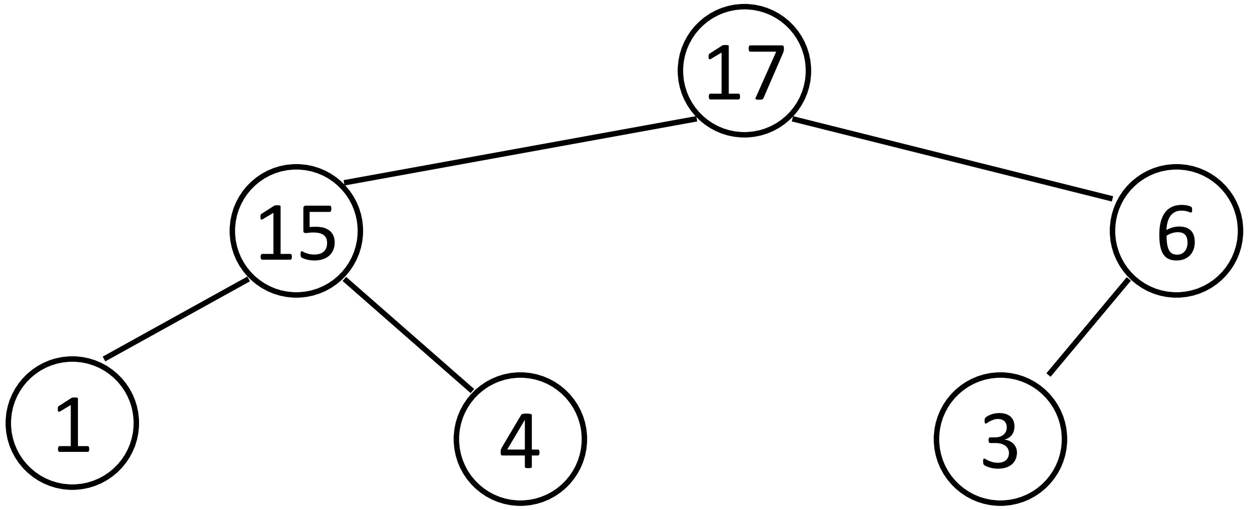
FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Objectives

- Revise basics of Heaps and Heap-based Priority Queue
- To understand a simple implementation of Heaps
- To be able to reason about the complexity of its operations
- Heap Sort

Heap (Max-Heap)

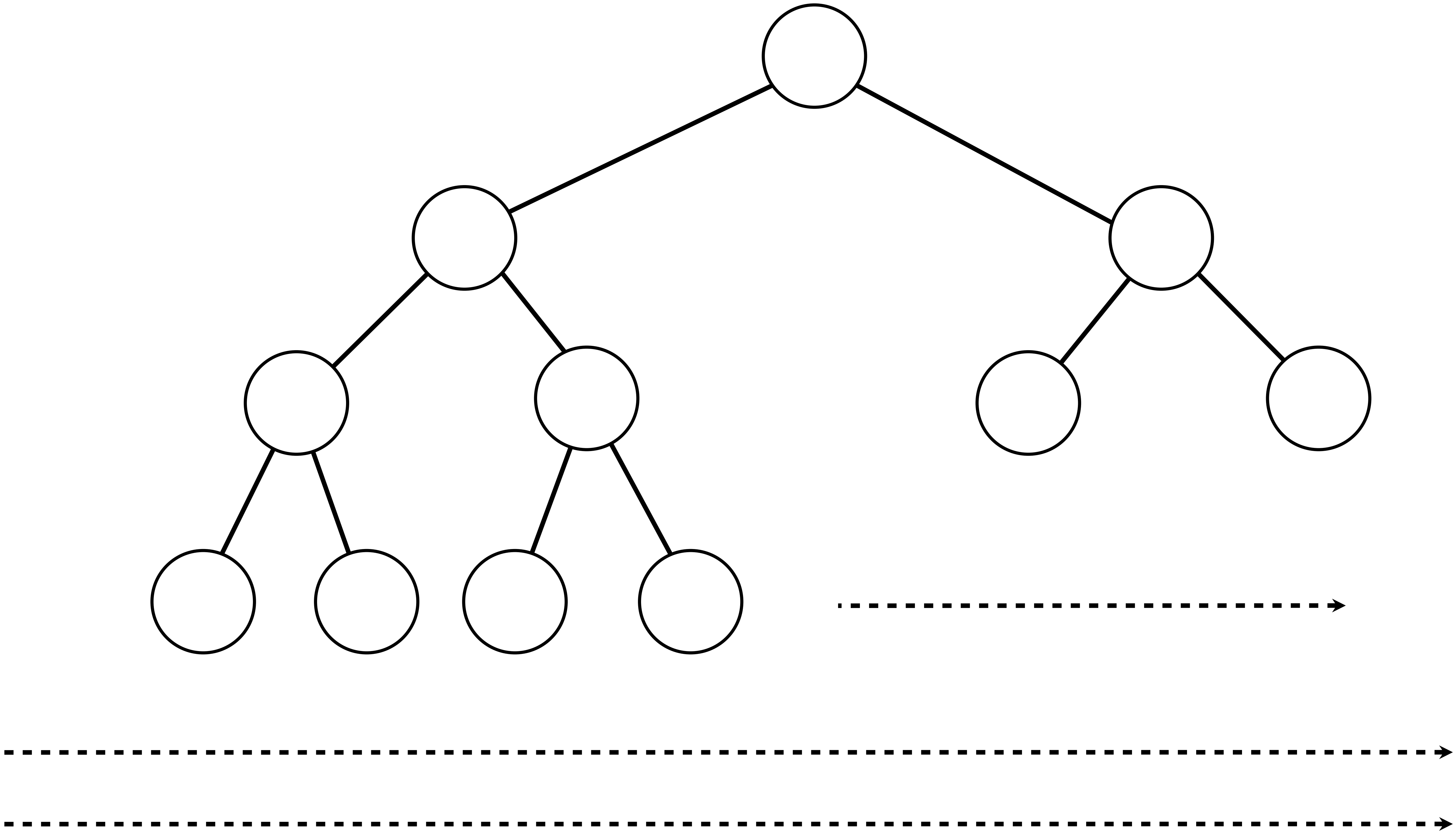


For **every** node:

- The values of the children are **<u>smaller or equal</u>** to its value.
- **All the levels are filled**, except possibly the last one, which is filled left to right.

Note: The **maximum** is always at the root of the tree.

Building a *binary* heap



Force the tree to be balanced…

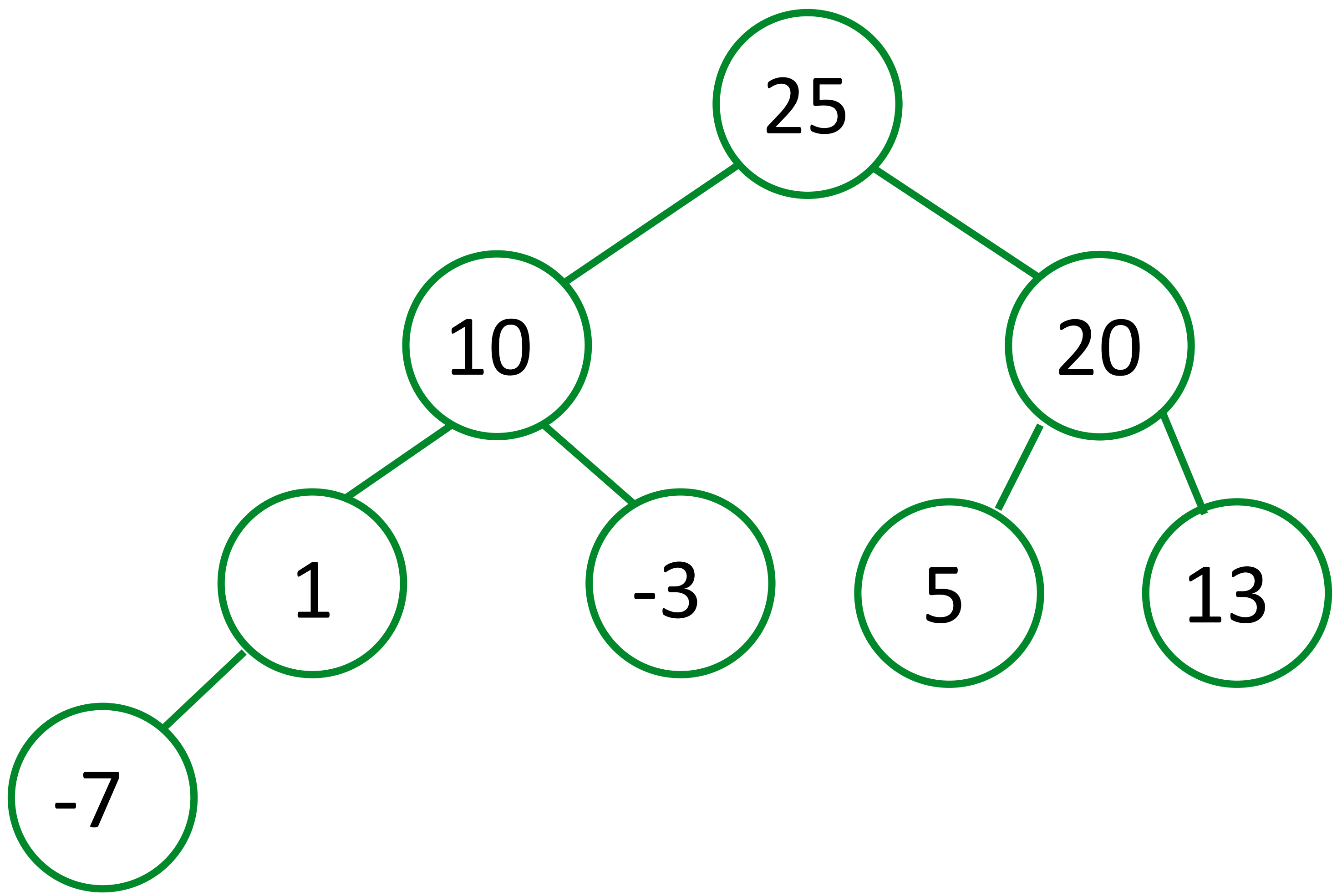
add:

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

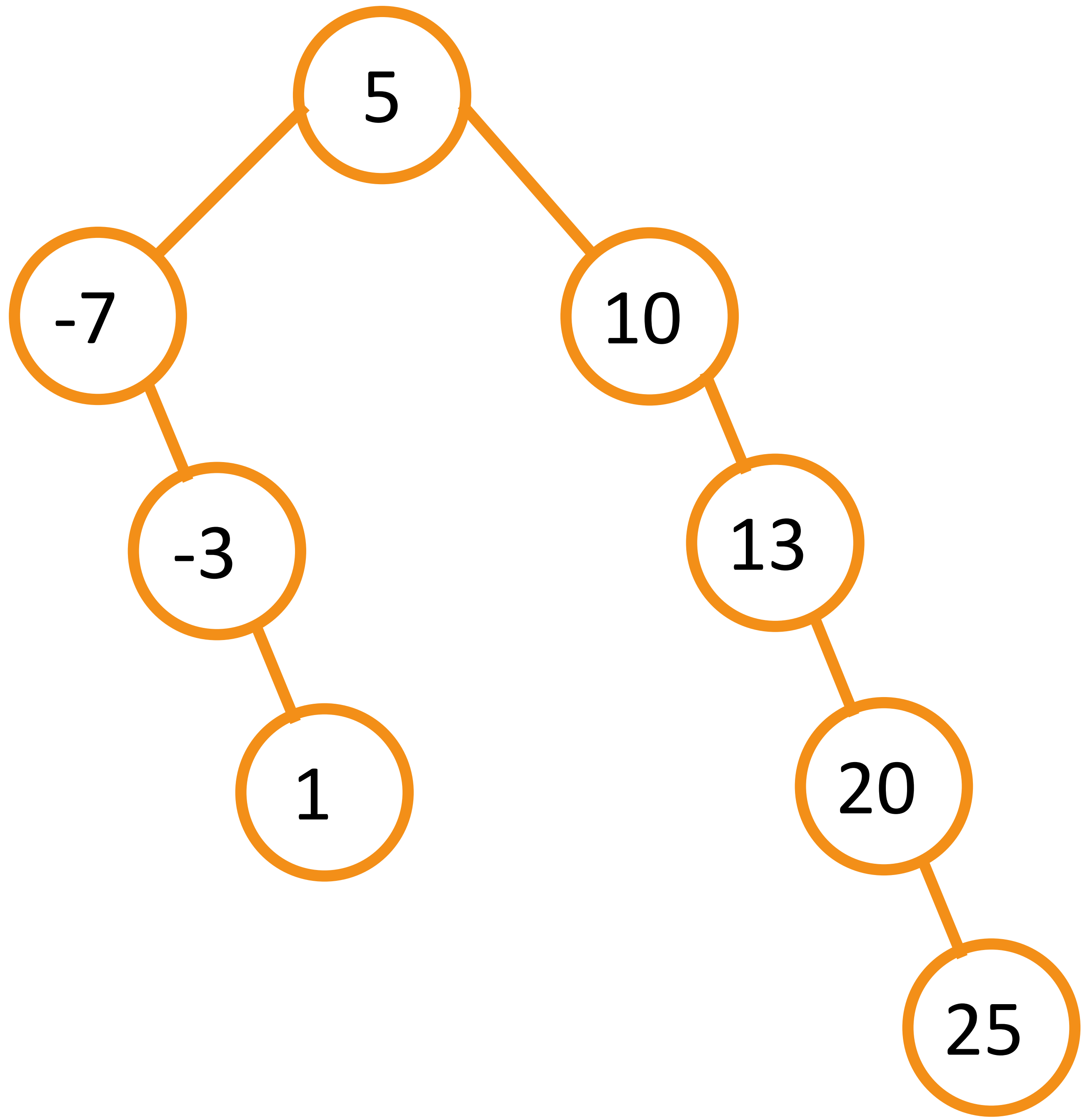
get_max:

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

Heap

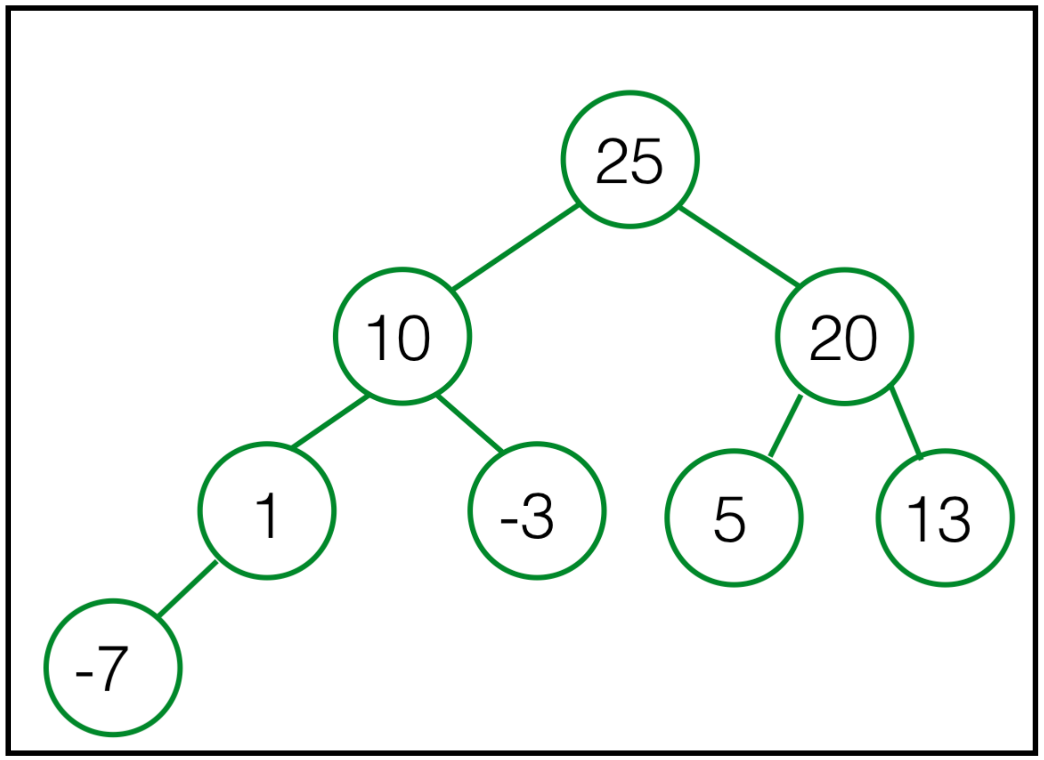


Lets insert the numbers [5, -7, 10, -3, 13, 20, 25,1] into an empty heap

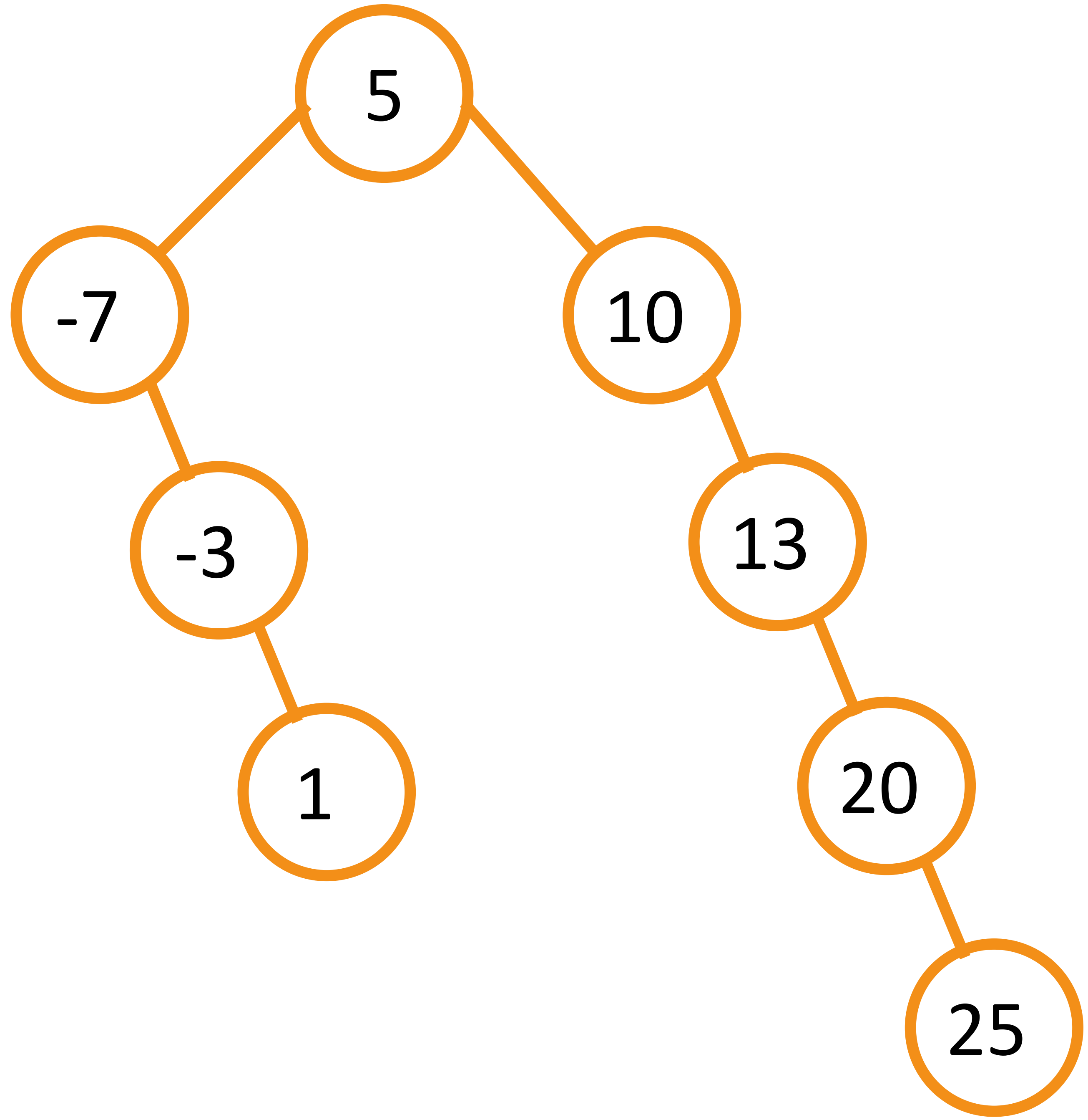


Lets insert the numbers [5, -7, 10, -3, 13, 20, 25,1] into an Binary Search Tree

Heap vs Binary Search Tree



Very different!



Implementation of Heaps?

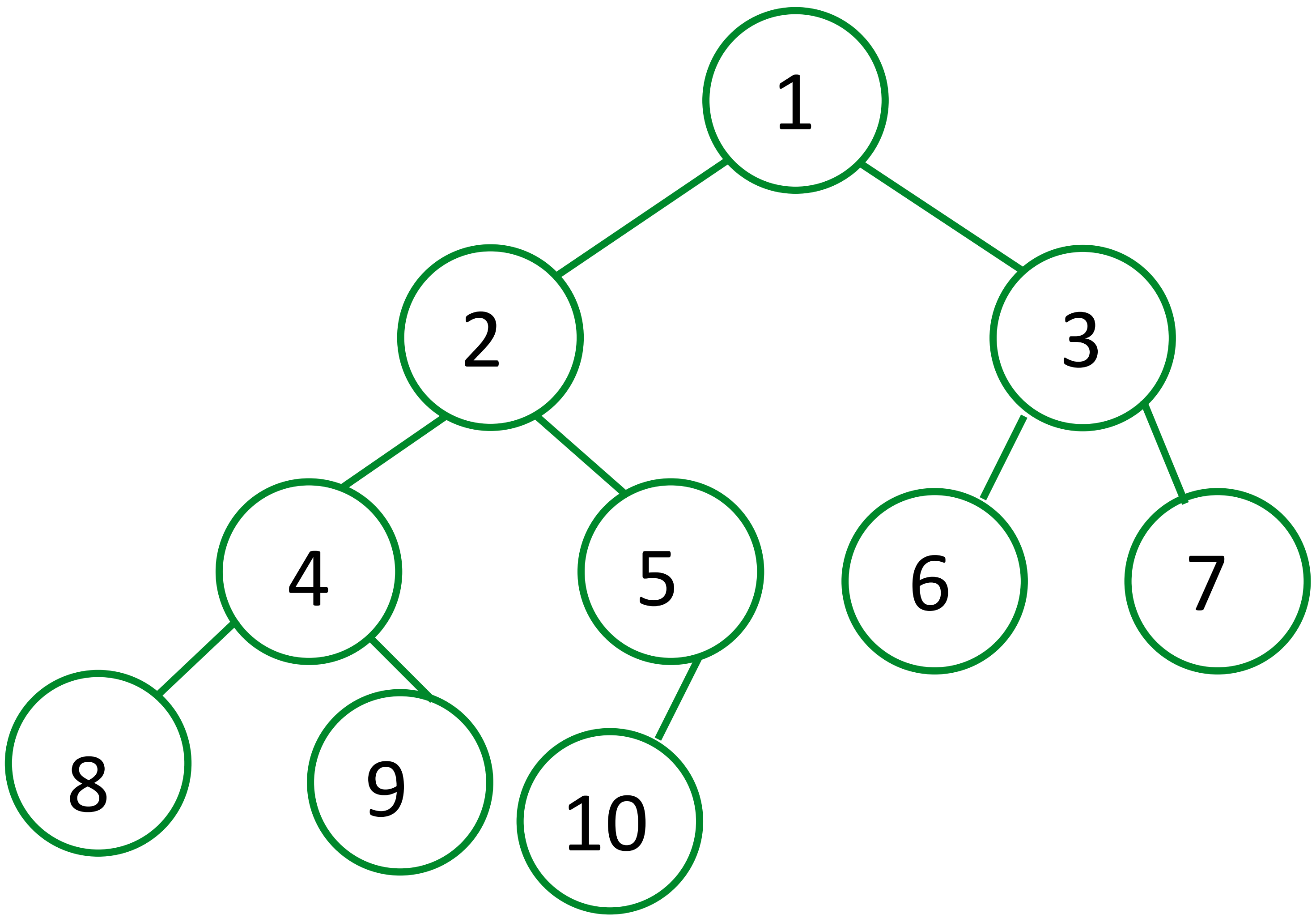
Implementation

Alternative 1: Binary tree of linked nodes

- → Downside: complex -- requires extra references to move up the tree (rise a node)
- → Extra memory.

Alternative 2: With an array

- → Possible due to completeness of the binary tree.
- → Advantages: Very compact



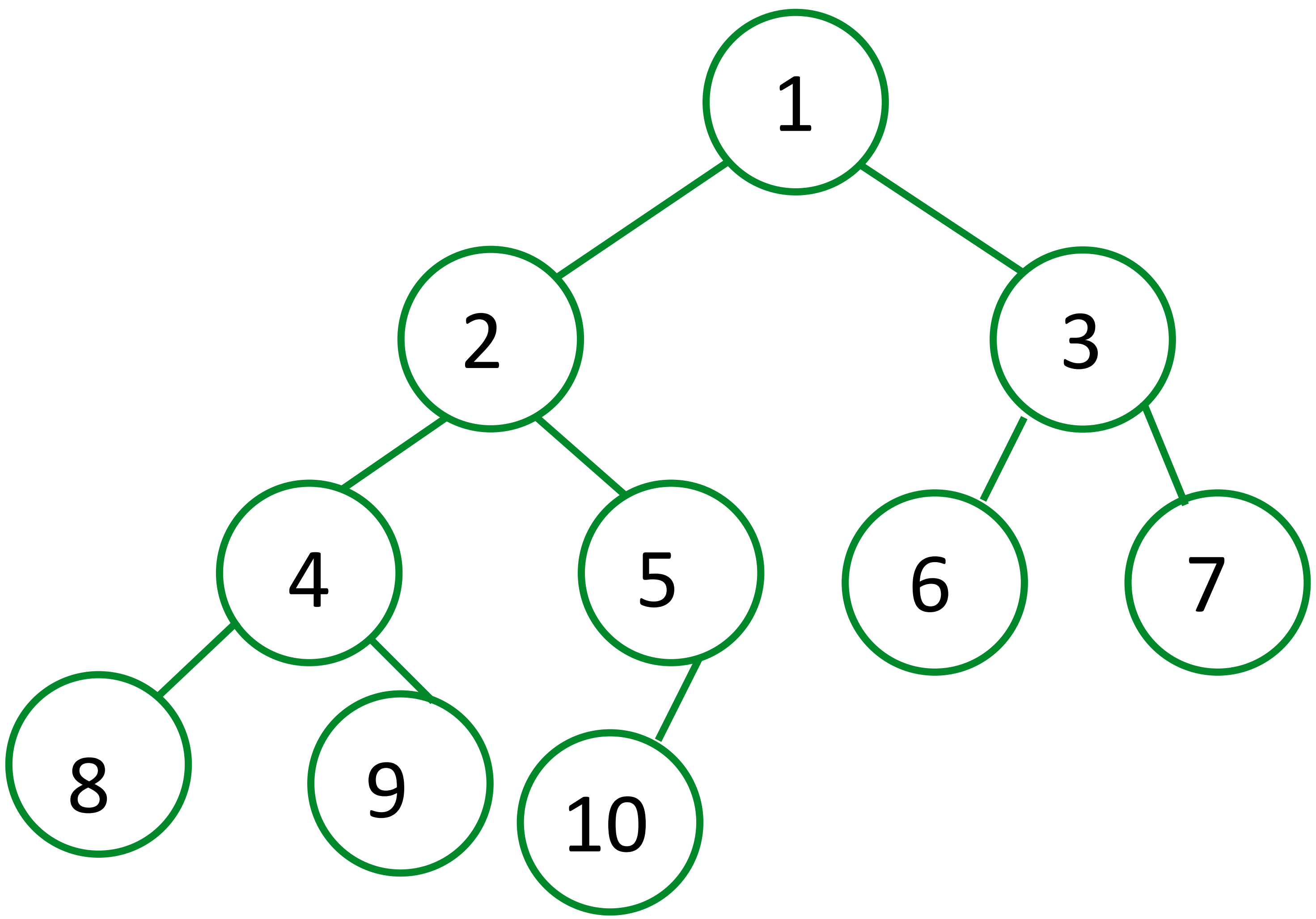
Parent Position	Child Left	Child Right
0	1	2
1	3	4
2	5	6
3	7	8
4	9	
k	?	?

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

shift

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10



Parent Position	Child Left	Child Right

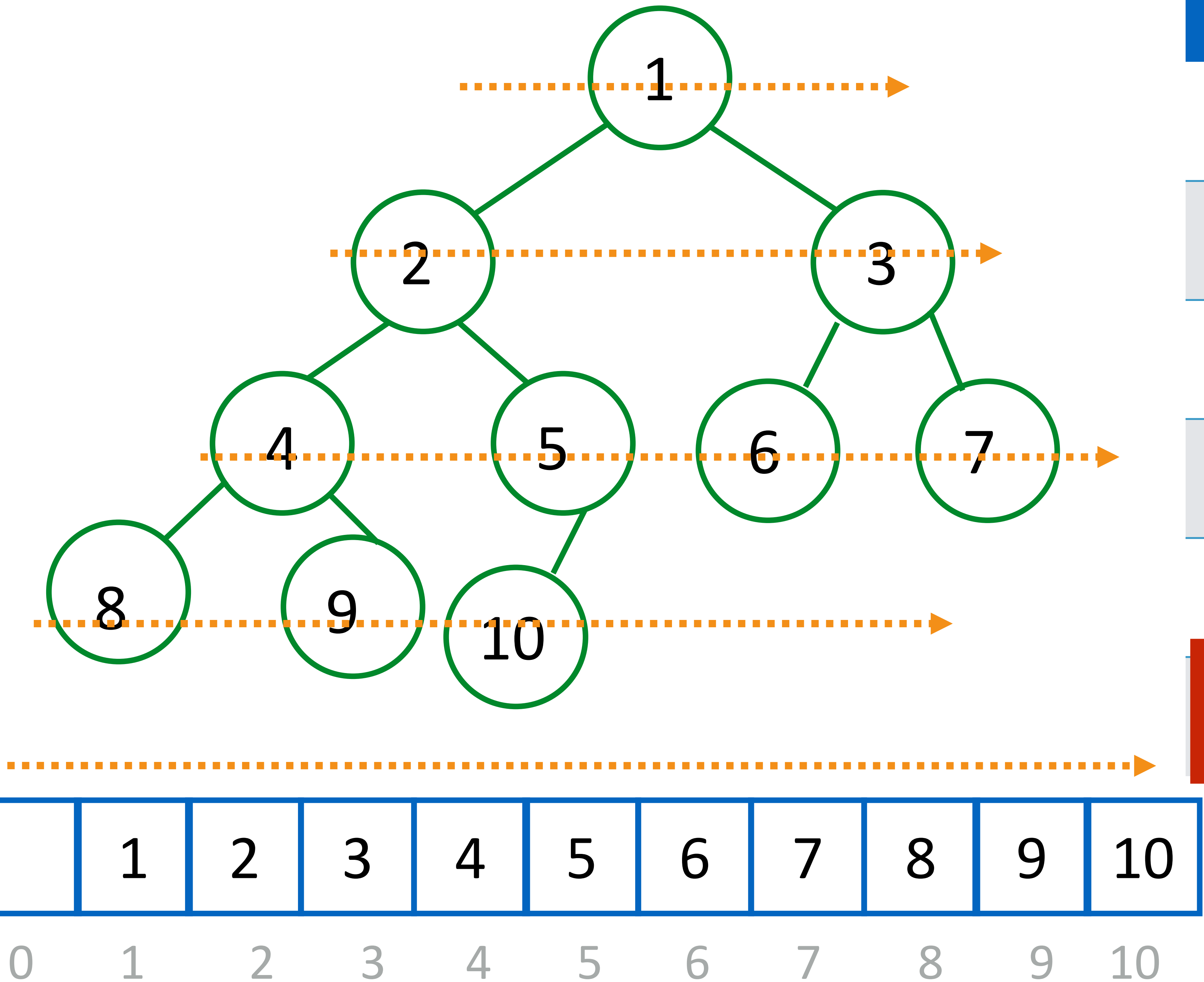
	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10

Root at position 1

Children of k: 2*k 2*k+1 (if they exist)

Parent of k: position k//2 (except for root)

Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7
4	8	9
5	10	
k	2*k	2*k+1



A concrete implementation

Start with a None so that we have items from position 1 onwards

```
class Heap:

    def __init__(self):
        self.count = 0
        self.array = [None]

    def __len__(self):
        return self.count
```

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.

```python
def add(self, item):
    if self.count + 1 < len(self.array):
        self.array[self.count+1] = item
    else:
        self.array.append(item)
    self.count += 1
    self.rise(self.count)
```

rise the last element -

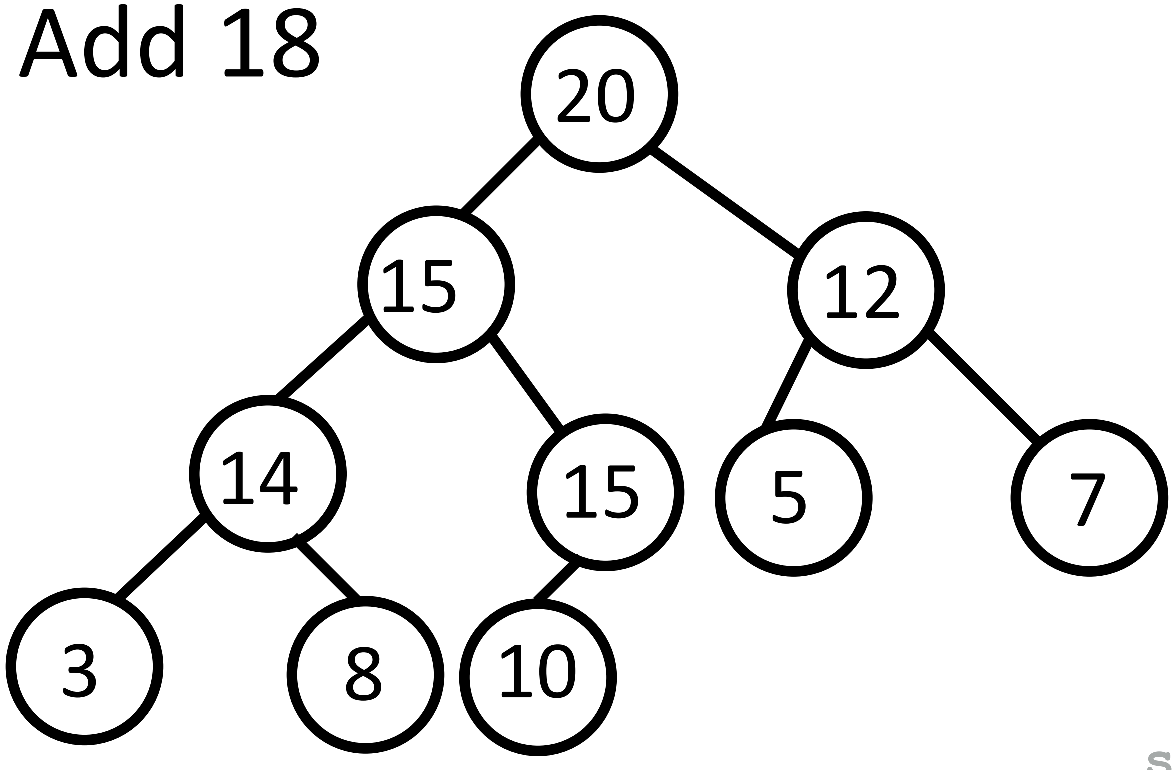
swap with parent while order is broken

```python
def swap(self, i, j):
    self.array[i], self.array[j] = self.array[j], self.array[i]

# Rise item at index k to its correct position
# Precondition: 1<= k <= self.count
def rise(self, k):
    while k > 1 and self.array[k] > self.array[k//2]:
        self.swap(k, k//2)
        k //= 2

def add(self, item):
    if self.count + 1 < len(self.array):
        self.array[self.count+1] = item
    else:
        self.array.append(item)
    self.count += 1
    self.rise(self.count)
```

Add 18



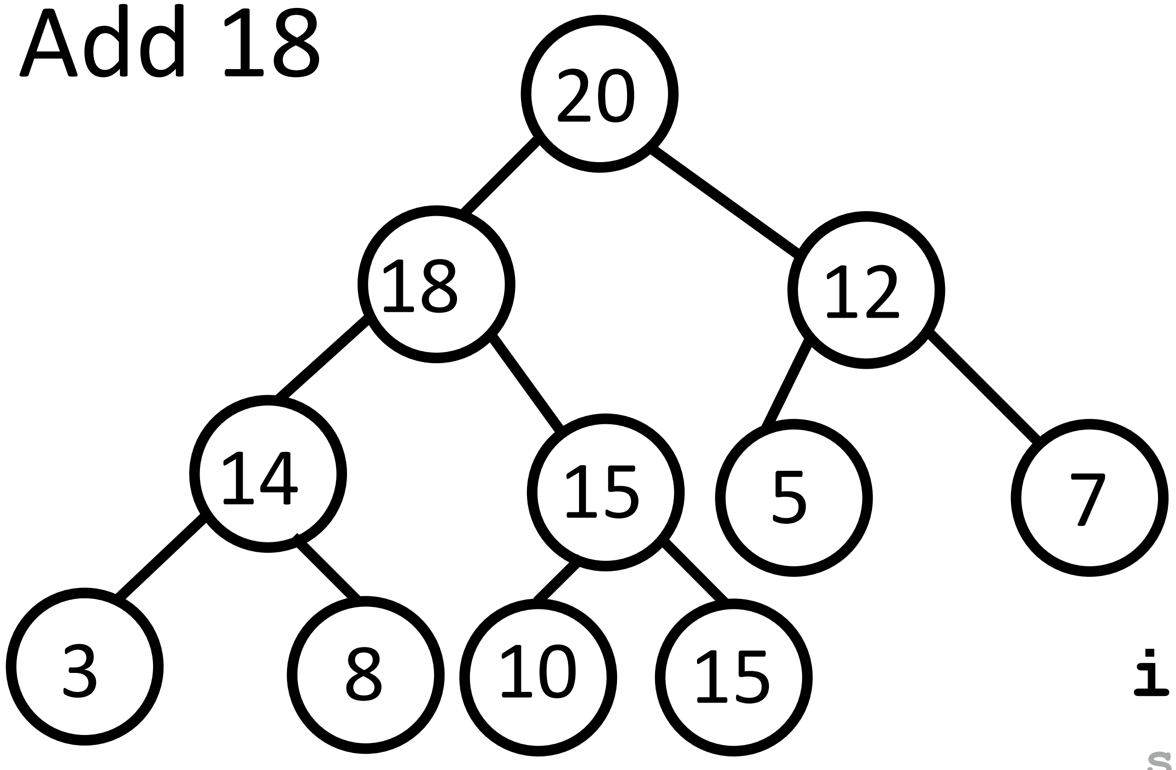
self.array

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

```
def rise(self, k):
    while k > 1 and self.array[k] > self.array[k//2]:
        self.swap(k, k//2)
        k //= 2

def add(self, item):
    if self.count + 1 < len(self.array):
        self.array[self.count+1] = item
    else:
        self.array.append(item)
    self.count += 1
    self.rise(self.count)
```

Add 18



item = 18 **self.count = 11**

self.array

	20	18	12	14	15	5	7	3	8	10	15
0	1	2	3	4	5	6	7	8	9	10	11

```
def rise(self, k):
    while k > 1 and self.array[k] > self.array[k//2]:
        self.swap(k, k//2)
        k //= 2

def add(self, item):
    if self.count + 1 < len(self.array):
        self.array[self.count+1] = item
    else:
        self.array.append(item)
    self.count += 1
    self.rise(self.count)
```

best case: O(1)

worst case: O(log N)

(may need to consider comparison operations)

Complexity of add

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

- Best case: O(1)*O(Compare) when the item is smaller or equal than its parent.

- Worst case: O(log N)*O(Compare) when the item rises all the way to the top.

Why log(N)? Heaps are always balanced
so there's only one path to explore

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.

Summary

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