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

Lecture 34
Priority Queues

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Objectives

- To understand the Priority Queue ADT.
- Consider different implementations (advantages/disadvantages)
- To understand the Heaps, and the Heap-based implementation of Priority Queues.

The doctor will see you now…

http://www.nma.gov.au/__data/assets/image/0018/218250/News_of_day_17_480w.jpg

Probably should have seen the doctor earlier

The doctor will see you now…

http://www.nma.gov.au/__data/assets/image/0018/218250/News_of_day_17_480w.jpg

Priority Queue

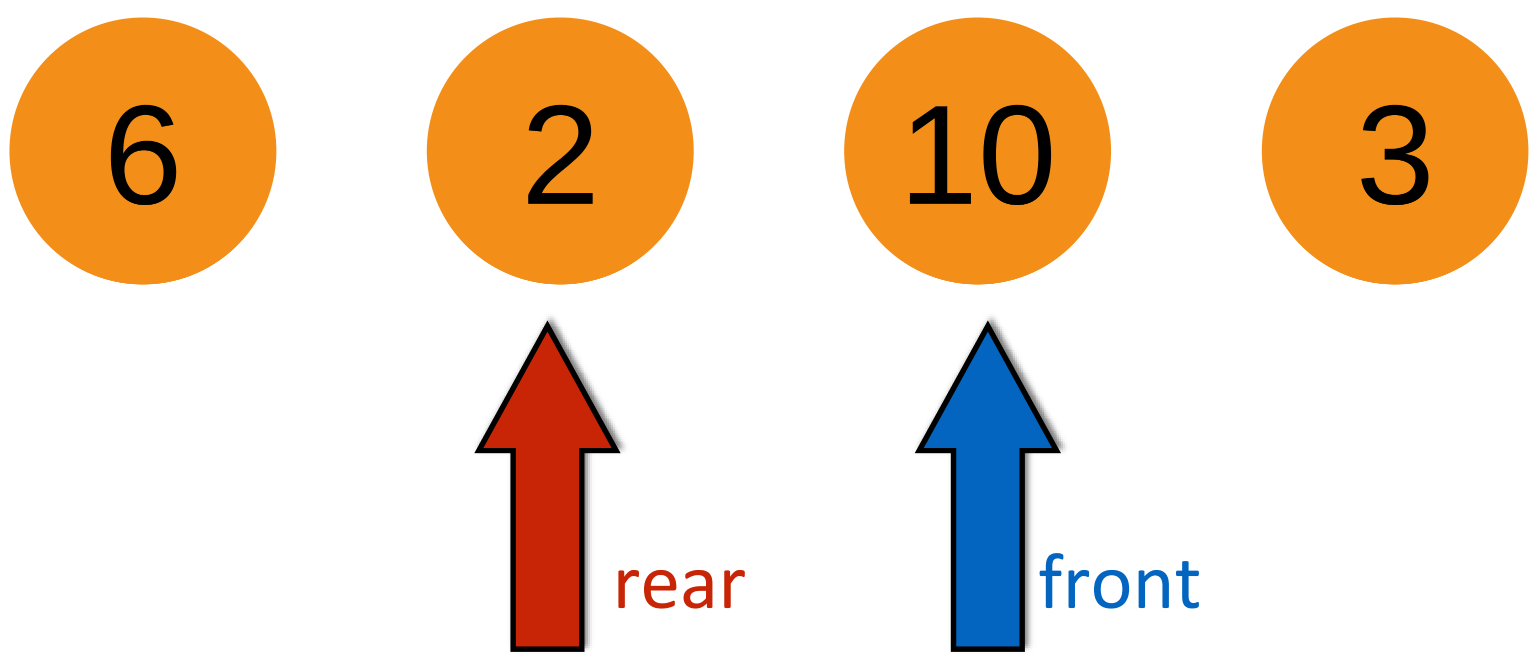
lowest in a dual implementation

- Each element has a numeric priority.
- Element with highest priority is processed first.

Operations:

`add(element)`

`get_max()`



The following data structures can be used to support the implementation of a Priority Queue ADT...

- Array-based List (sorted and unsorted)
- Linked List (sorted and unsorted)
- Binary Search Trees
- Heaps (Binary Tree-based)
- Heaps (Array-based)

Implementing Priority Queues

- Standard operations:
  `is_empty, __len__, __init__`

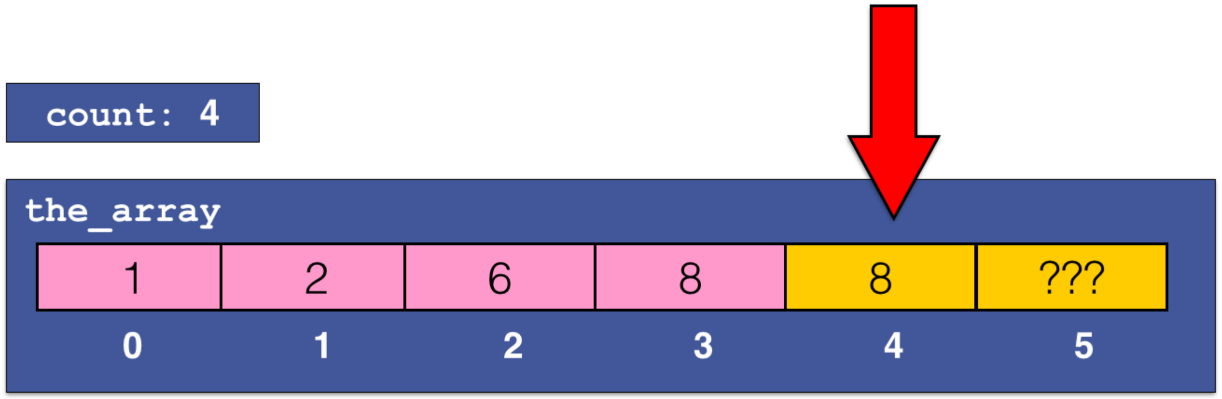
- Core operations:

  - **`get_max()`**:
    returns the max element (and removes it from the queue)

  - **`add(element)`**: adds element to the priority queue

Complexity

(Priority Queue using Unsorted Lists)



get_max()

- Find max item.
- Remove and reshuffle.

O(n)

---

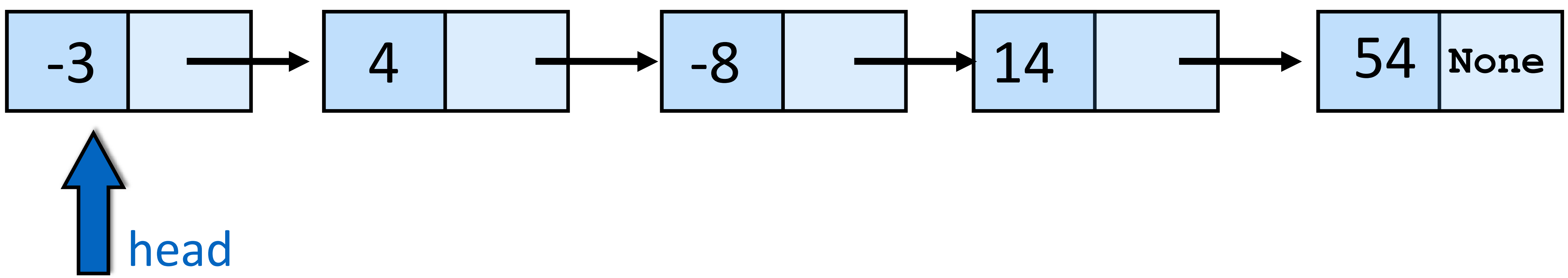
add()

O(1)

- Add item at the back.

Complexity

(Priority Queue using Unsorted Linked Lists)



`get_max()`

- Find max item.
- Remove.

O(n)

`add()`

O(1)

- Add item at the head.

Adding to an array-based sorted list

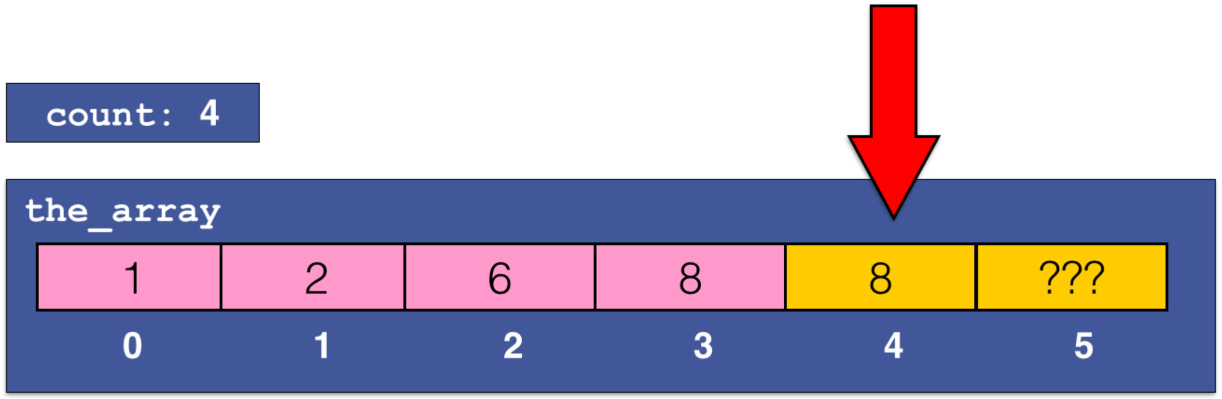
```
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```

Find correct position

Move things to make space

Complexity

(Priority Queue using Sorted Lists)



`get_max()`

- Find max item.
- Remove.

Position
`count-1`

O(1)

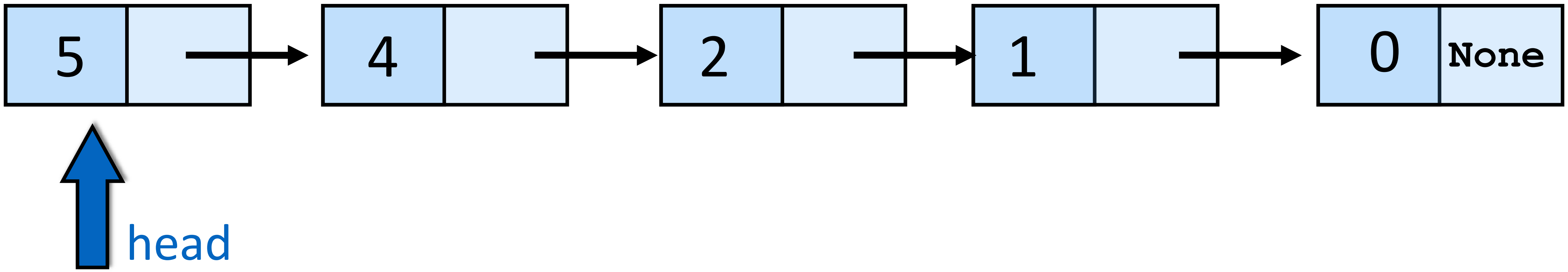
`add()`

- Add item.

O(n)

Complexity

(Priority Queue using Sorted Linked Lists)



`get_max()`

at head

- Find max item.
- Remove.

O(1)

`add()`

- Find correct position.

O(n)

Priority Queues using linear structures…

Implementation	`get_max()`	`add`
Unsorted array	O(n)	O(1)
Unsorted linked list	O(n)	O(1)
Sorted array	O(1)	O(n)
Sorted linked list	O(1)	O(n)

Consider comparisons complexity

Linear structure gives linear complexity

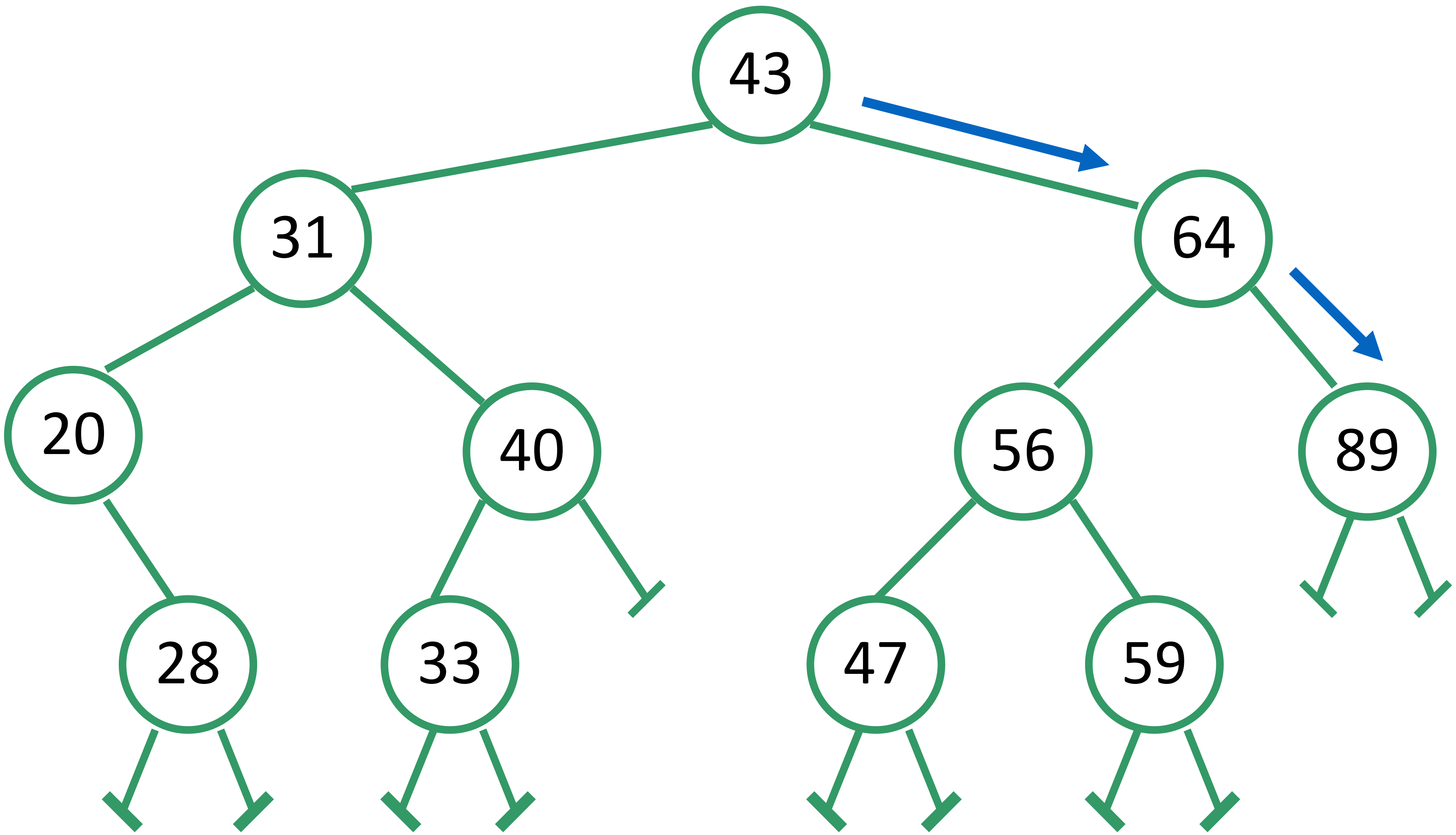
Let’s try a non-linear structure!

```python
class BinarySearchTreeNode:
    def __init__(self, key, item=None, left=None, right=None):
        self.key = key
        self.item = item
        self.left = left
        self.right = right


class BinarySearchTree:
    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None
```

`add()` trivial.

`get_max()` ?

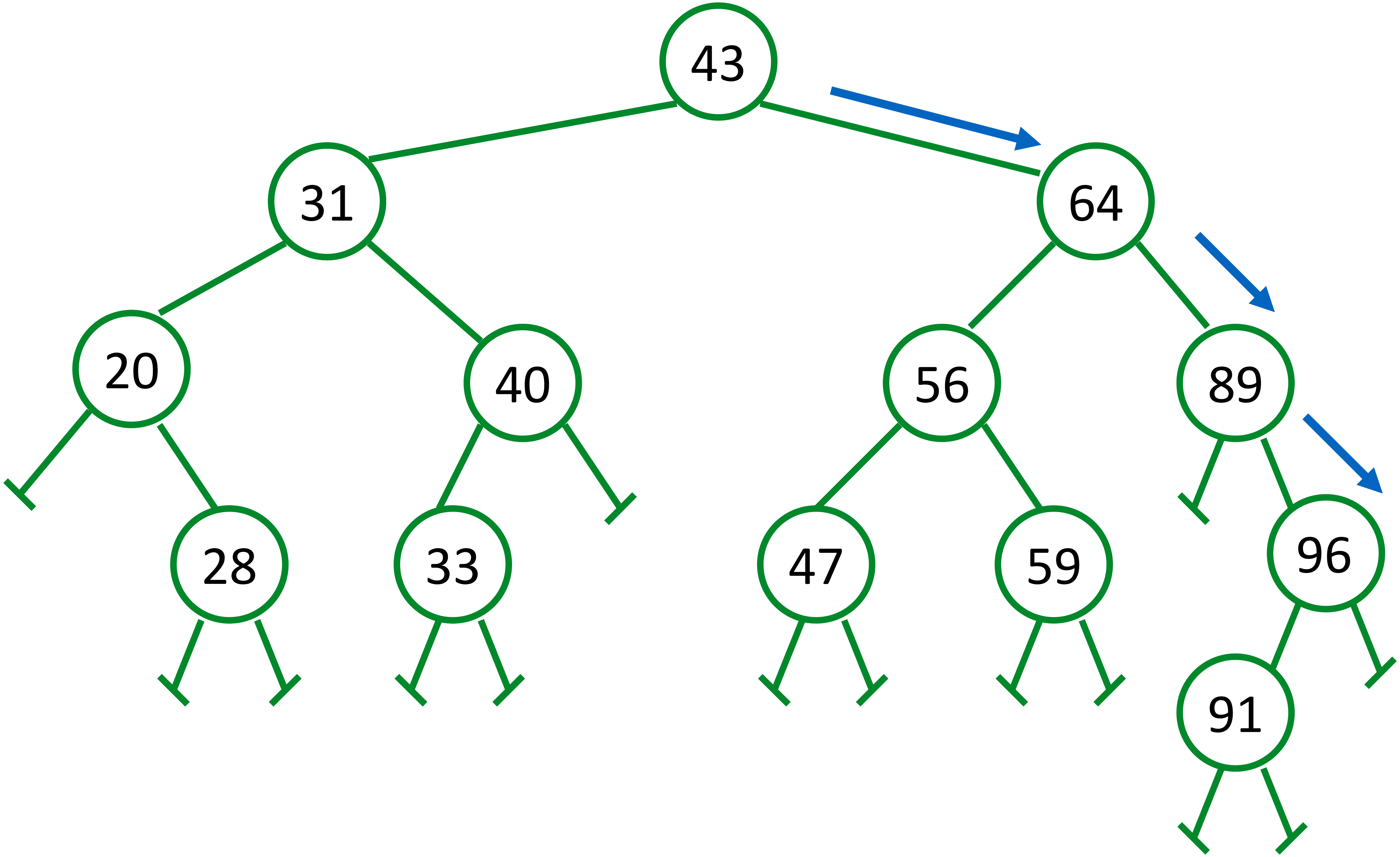
BST: **`get_max()`** ?



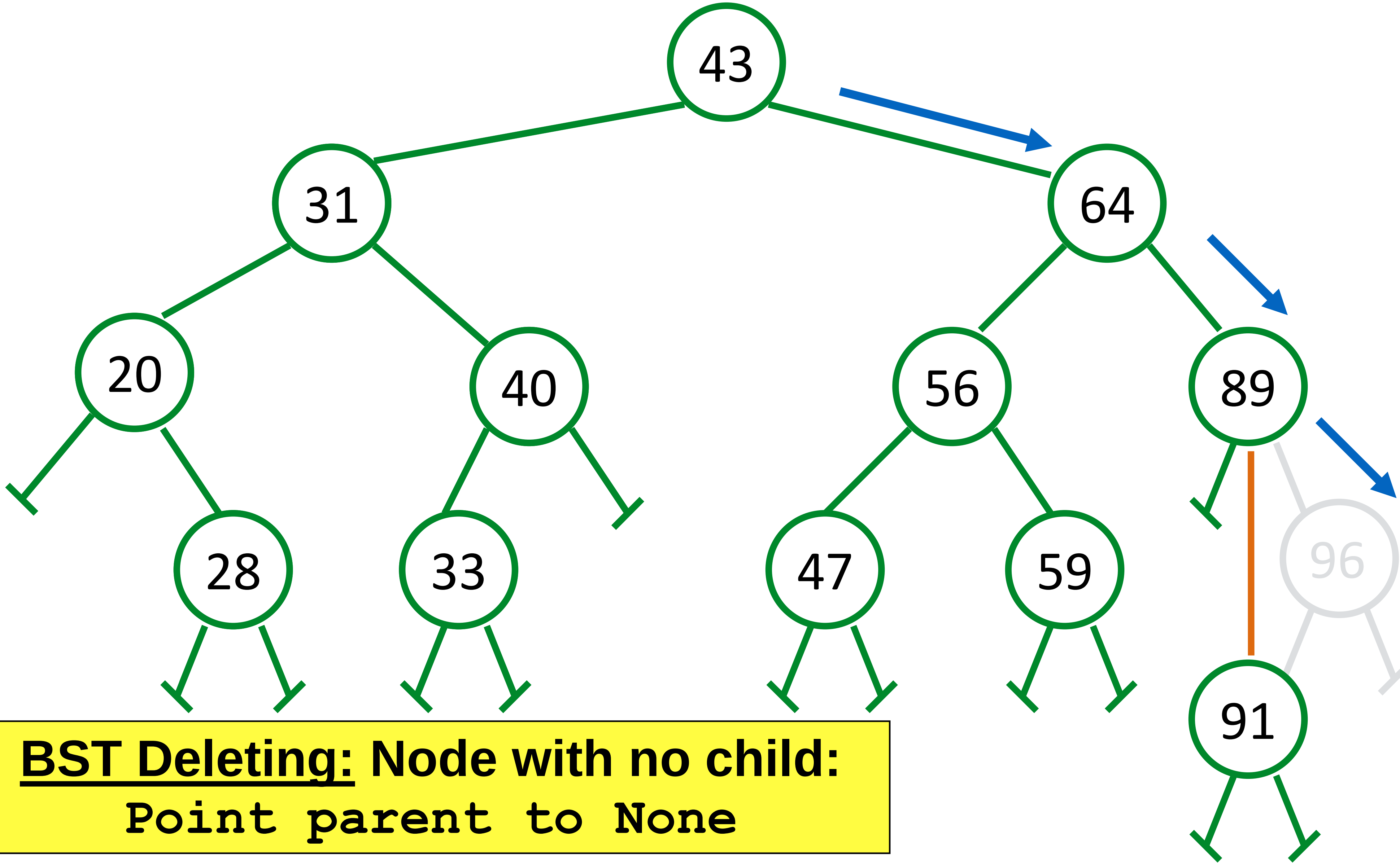
Right-most node: Go right until you can

Complexity depends on the height!

`get_max()`



`get_max()`



<u>BST Deleting:</u> Node with no child:
`Point parent to None`

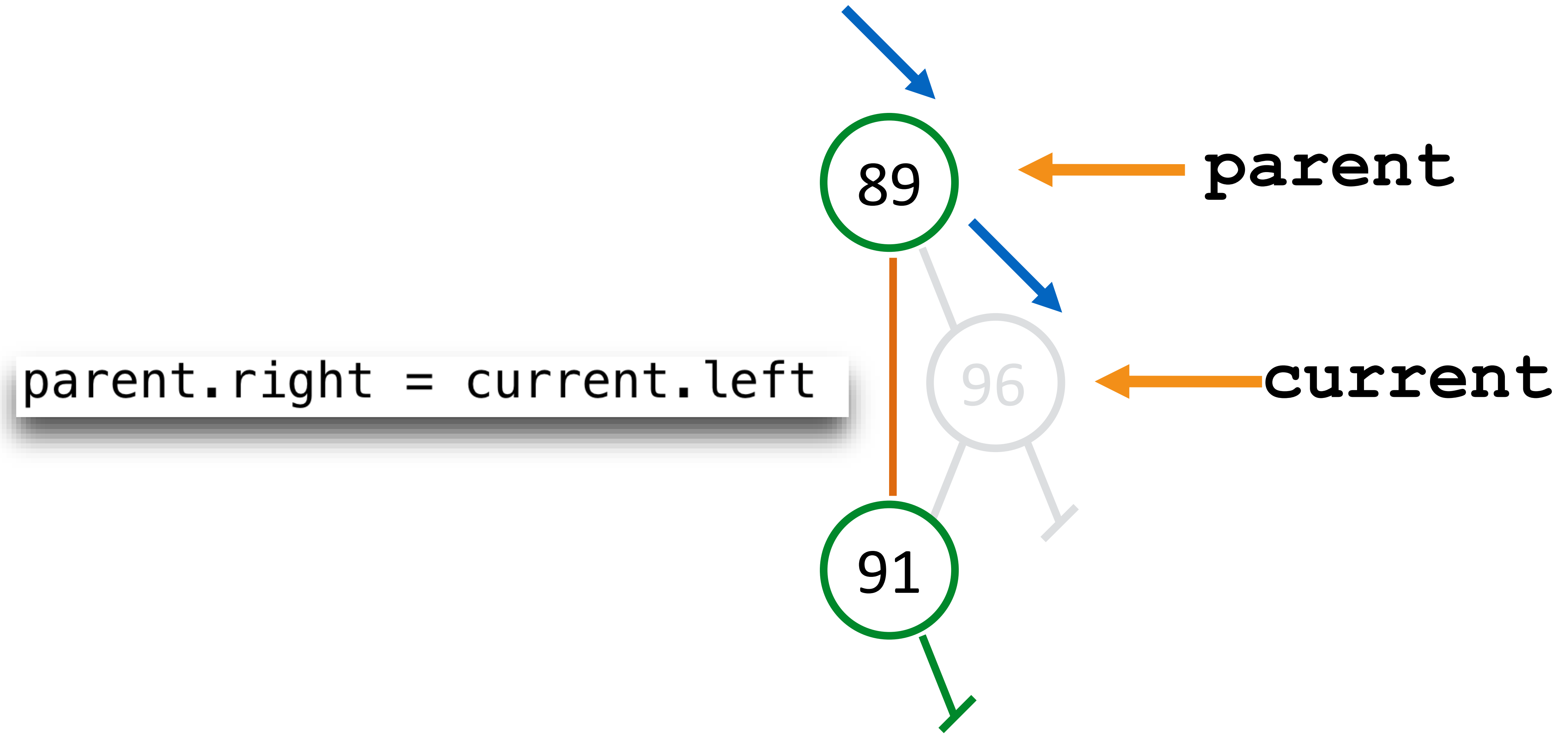
<u>BST Deleting:</u> Node with one child:
`Point parent to child of deleted node`

Simple version (without deleting)

```python
def get_max(self):
    if self.root is None:
        raise ValueError("Empty Priority Queue")
    else:
        return self.get_max_aux(self.root)

def get_max_aux(self, current):
    if current.right is None:  # base case: at max
        return current.item
    else:
        return self.get_max_aux(current.right)
```

With delete?

[Remember the parent]



Node with no child:
Point parent to None

Node with one child:
Point parent to child of deleted node

```python
def get_max(self):
    if self.root is None:
        raise ValueError("Priority Queue is empty")
    elif self.root.right is None: # root has the max
```

We can only call the aux method if we have a parent!

```python
def get_max_aux(self, current, parent):
    if current.right is None: # base case: at max
        parent.right = current.left
        return current.item
    else:
        return self.get_max_aux(current.right,current)
```

```
def get_max(self):
    if self.root is None:
        raise ValueError("Priority Queue is empty")
    elif self.root.right is None: # root has the max
        temp = self.root.item
        self.root = self.root.left # delete root
        return temp
    else:
        return self.get_max_aux(self.root.right, self.root)
```

```
def get_max_aux(self, current, parent):
    if current.right is None: # base case: at max
        parent.right = current.left
        return current.item
    else:
        return self.get_max_aux(current.right,current)
```

Alternative implementation

```python
def get_max(self):
    if self.root is None:
        raise ValueError("Priority Queue is empty")
    elif self.root.right is None: # root has the max
        temp = self.root.item
        self.root = self.root.left # delete root
        return temp
    else:
        return self.get_max_aux(self.root)


def get_max_aux(self, parent):
    if parent.right.right is None: # base case: at max
        temp = parent.right.item
        parent.right = parent.right.left
        return temp
    else:
        return self.get_max_aux(parent.right)
```

only passing the parent, but “looking down” two levels

A better implementation

- Each of the previous choices has one O(N) operation.
- O(Depth) for the binary tree with depth being N-1 if unbalanced.
- Can we do better? Use a (max) heap.
  - →One could also use a min-heap
  - →In FIT1008 we use max-heaps but the ideas are the same for a min-heap.

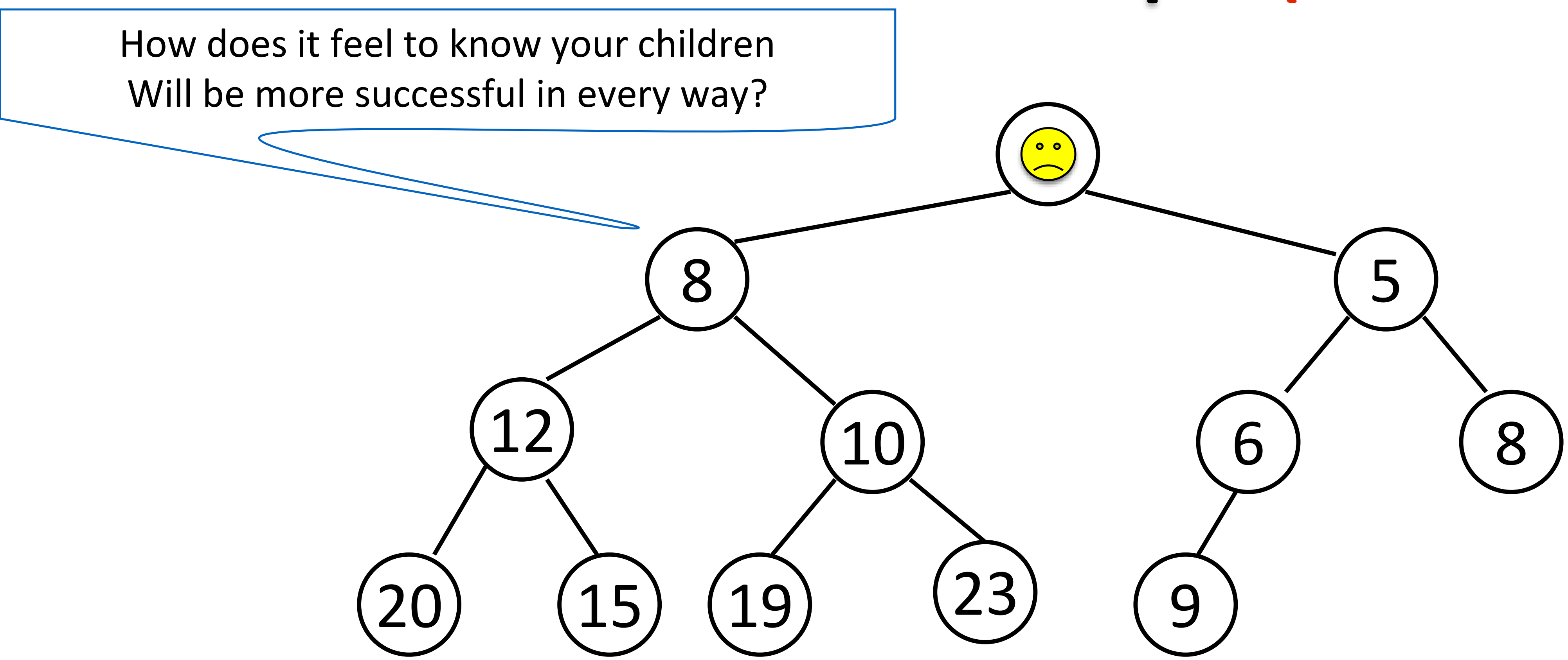
big bricks boss it up

Soil and silt sink through

Max Heaps!

http://static3.depositphotos.com/1003450/176/i/950/depositphotos_1760575-Heap-of-red-brick.jpg

Heap (Min-Heap)

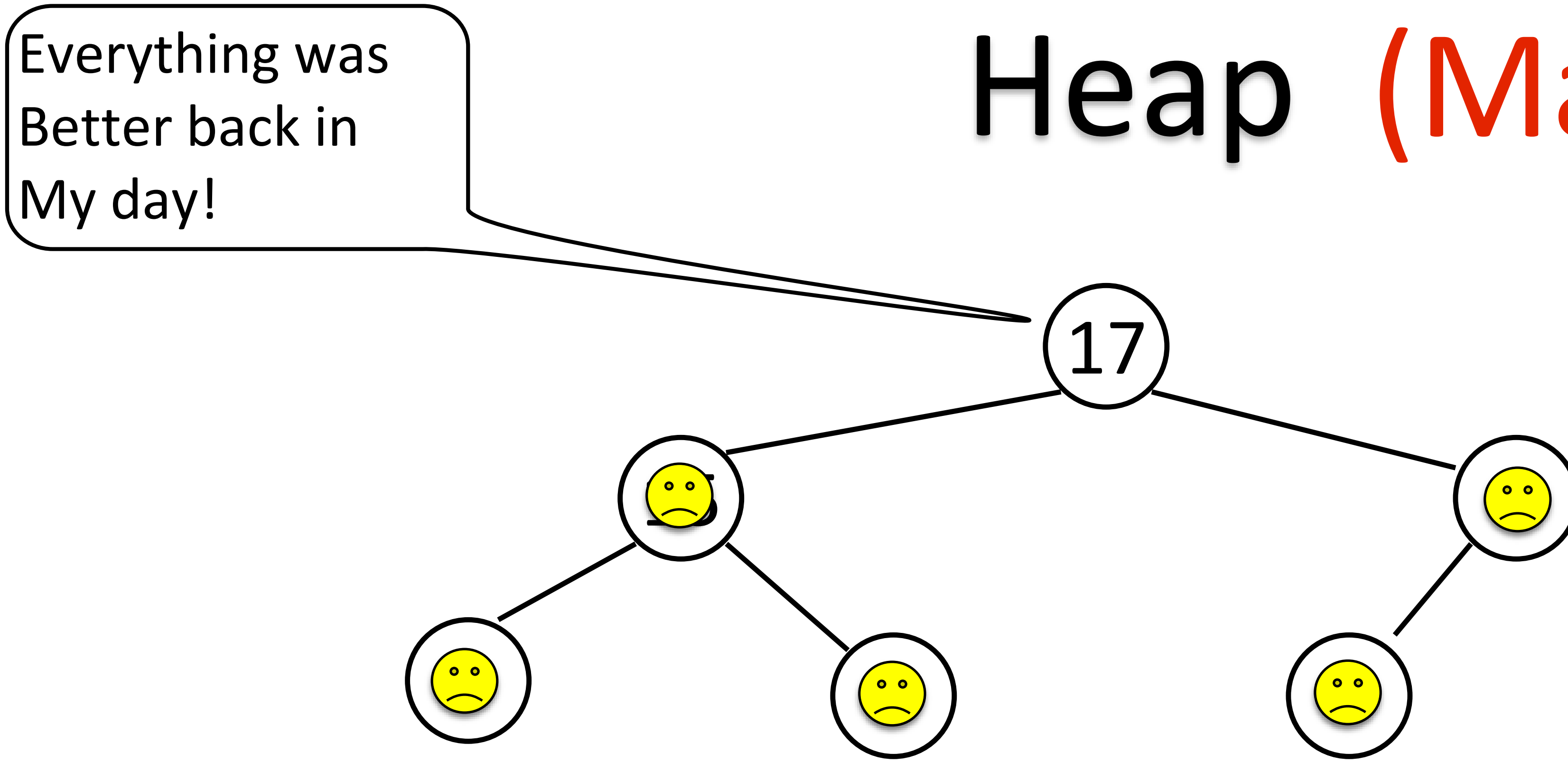


For **every** node:

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

Note: The minimum is always at the root of the tree.

Heap (Max-Heap)

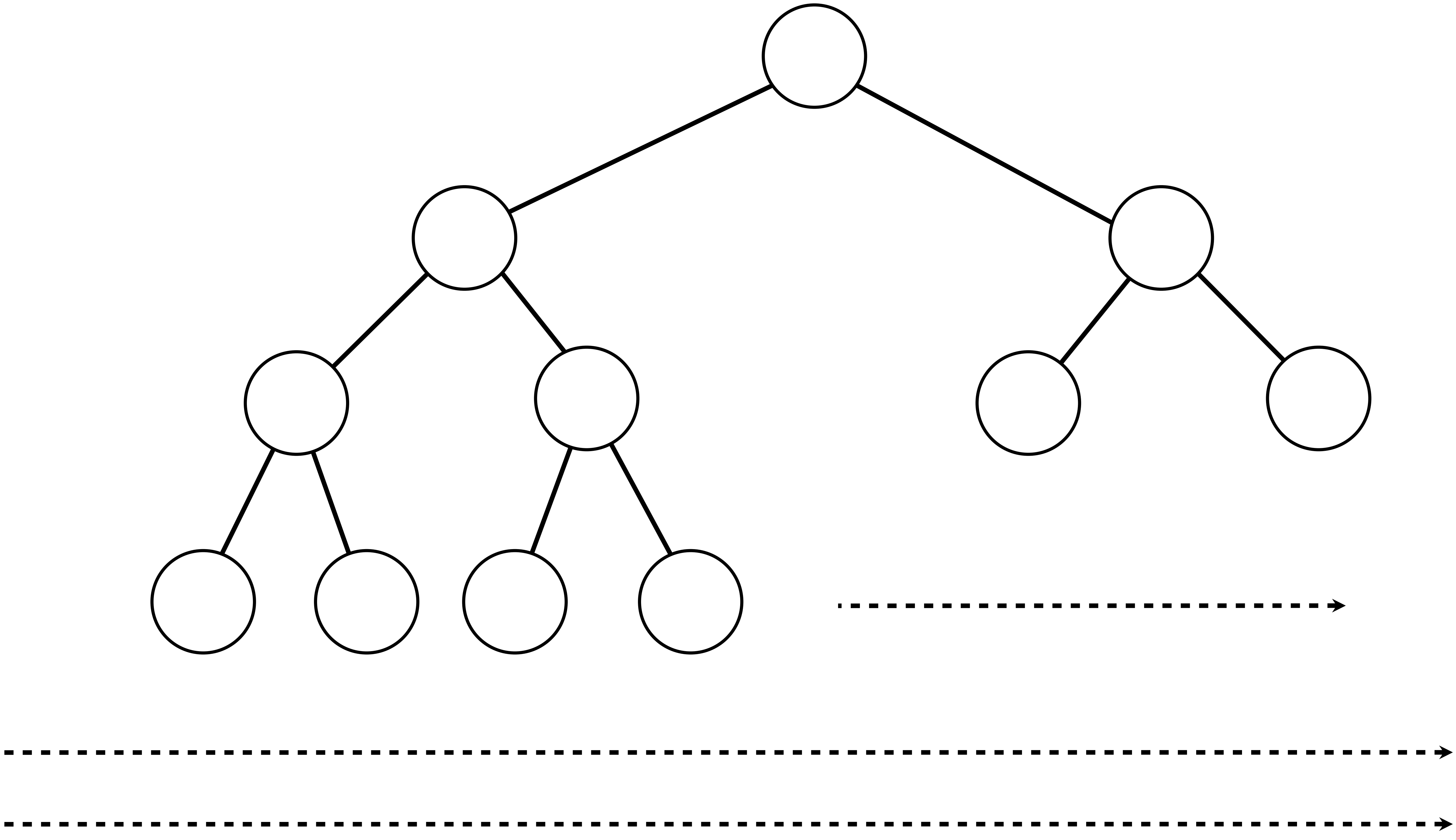


For **every** node:

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

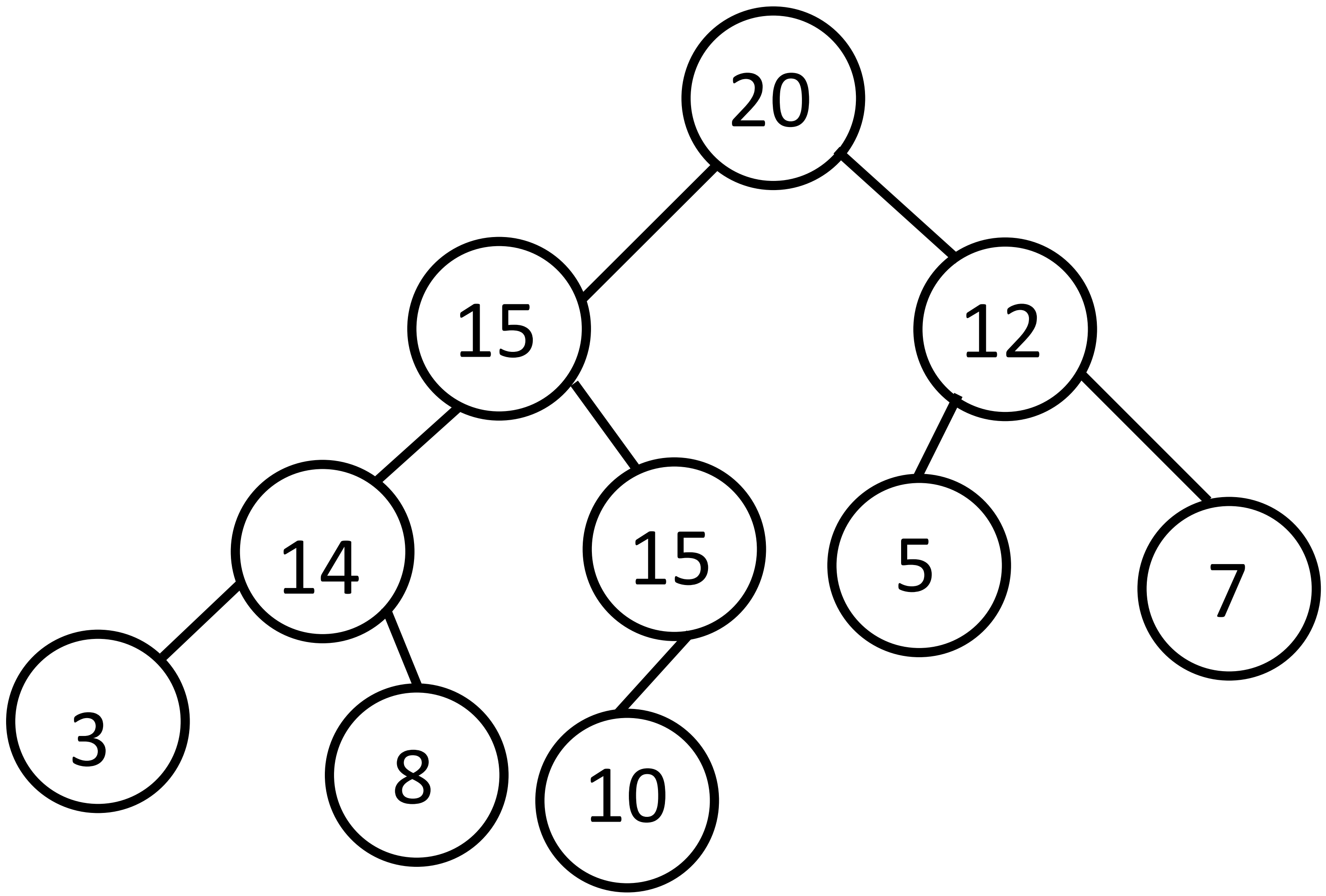
Note: The **maximum** is always at the root of the tree.

Building a *binary* heap



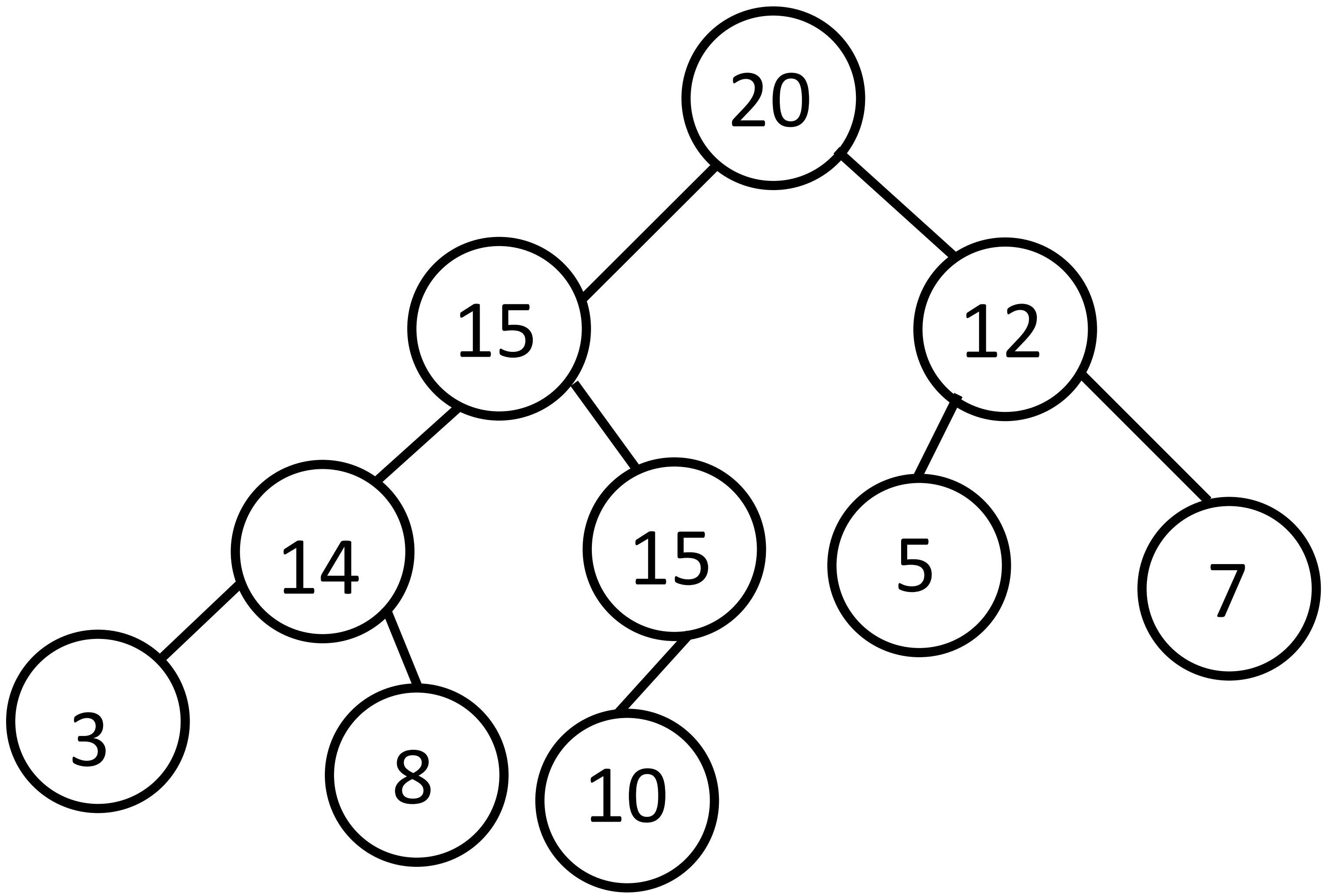
Force the tree to be balanced…

Example

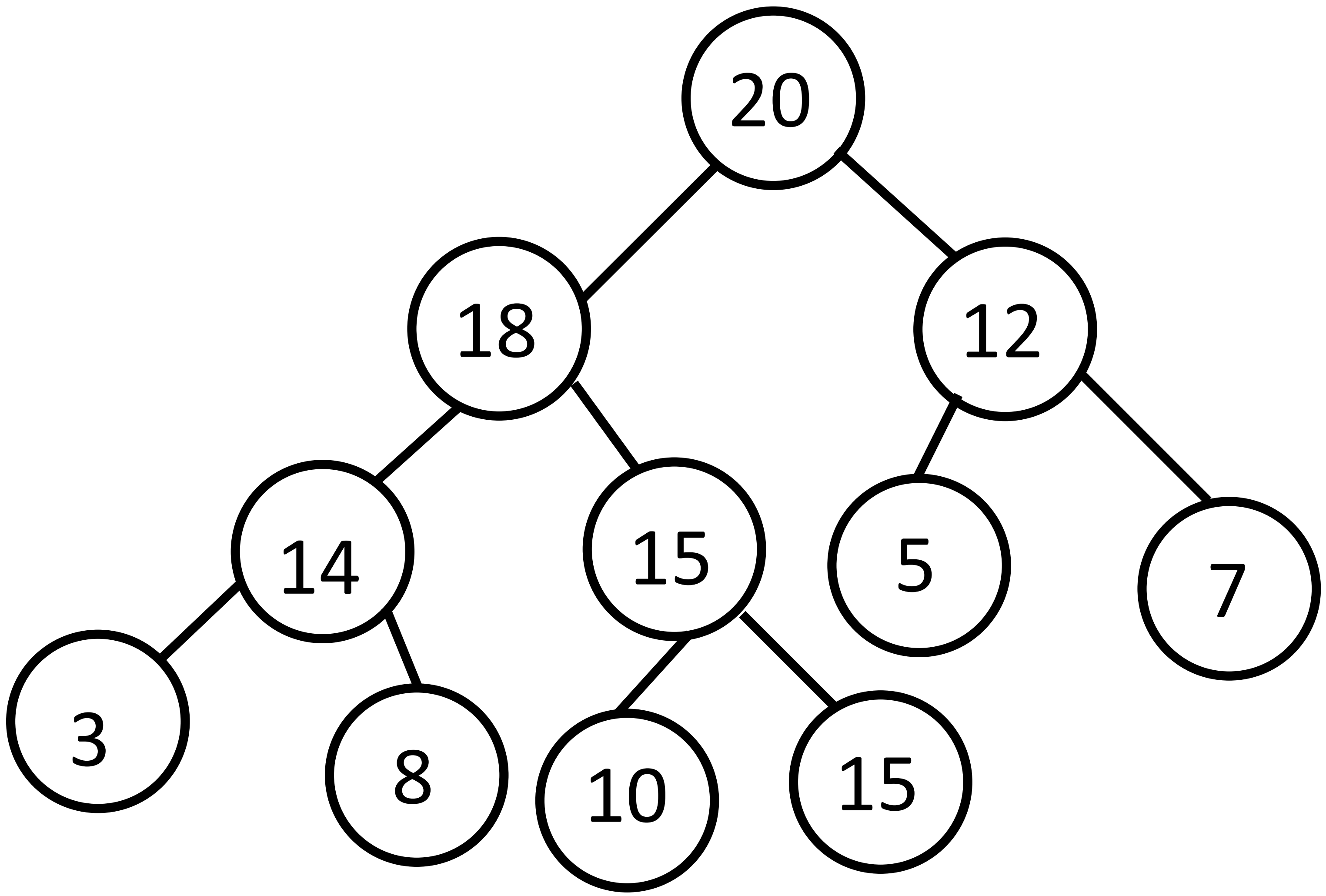


“Not a binary search tree”

Add 18



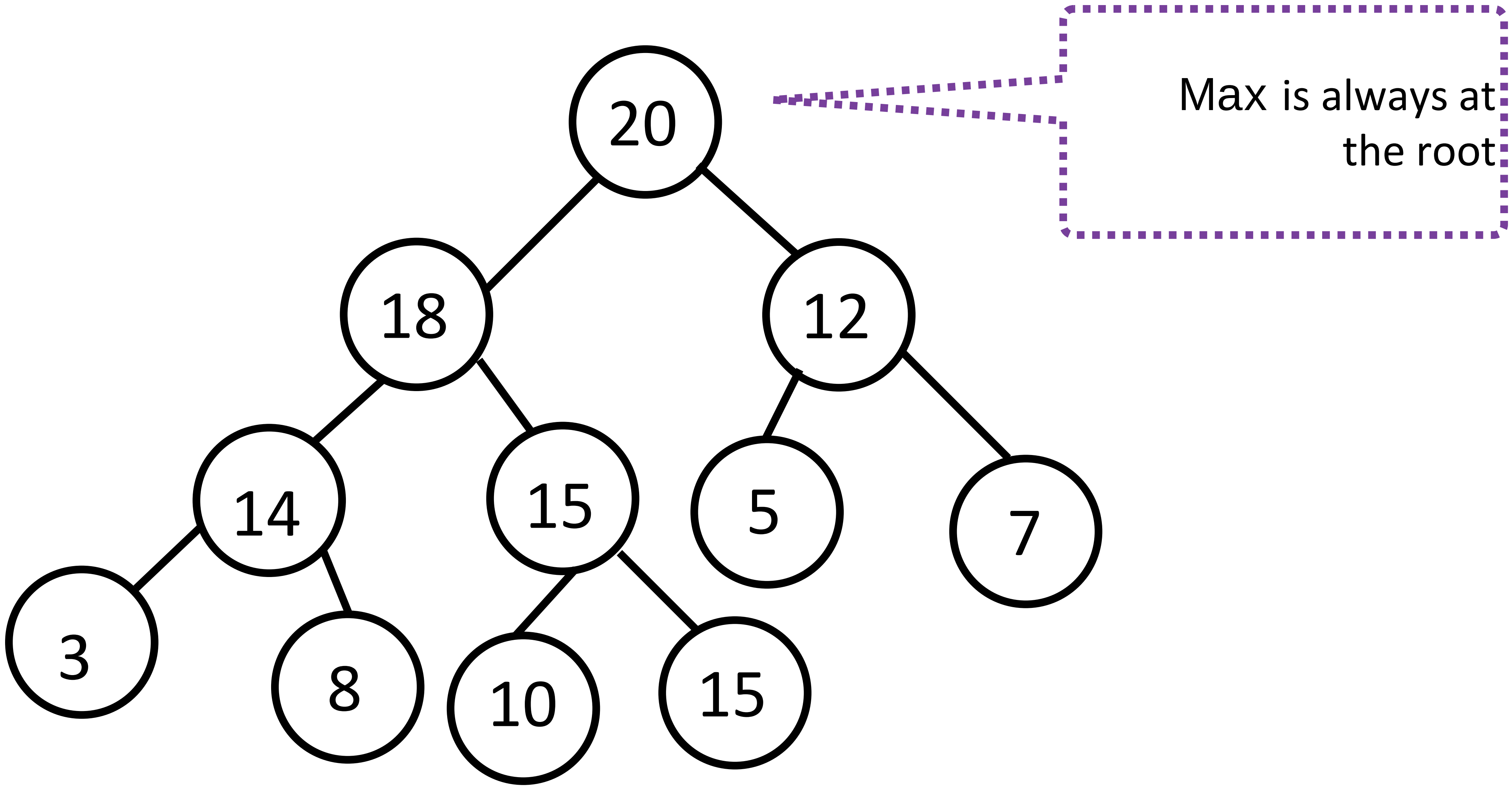
Add 18



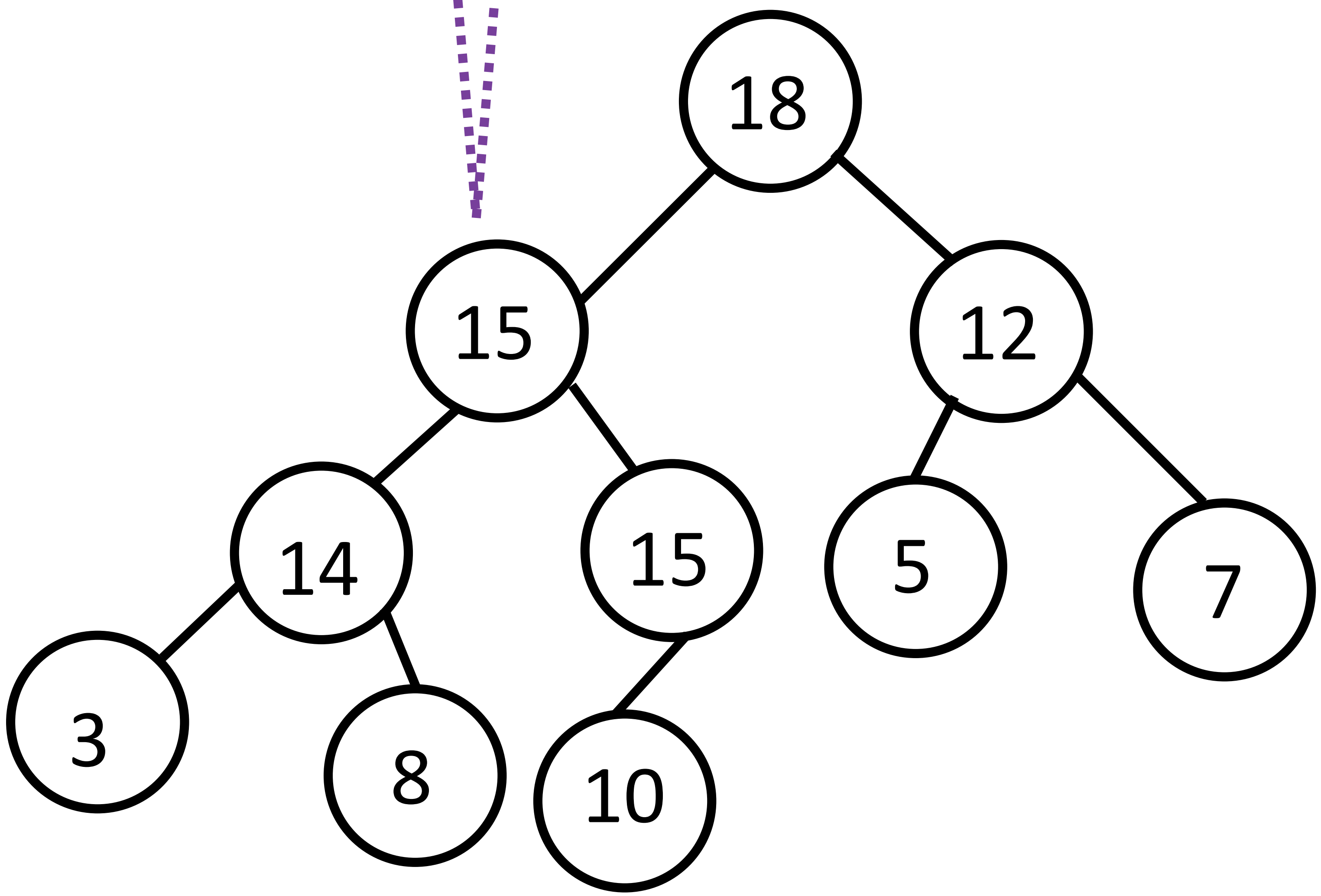
all good now!

issue: swaps need a reference to parent

```
get_max()
```


20
18
12
14
15
5
7
3
8
10
15
Max is always at the root

“Sink” until the element is larger than or equal to children



Complexity?

Implementation?

http://www.brainlazy.com/wp-content/uploads/Disgaea-3-Absence-of-Detention-14.jpg

I don't know what this is… but we'll cover complexity and implementation next time

Summary

- Priority Queues:
  - **`add`**
  - **`get_max`**

- Possible implementations: Lists, BST.

- Heaps: binary tress that are
  - Complete
  - Heap ordered

- Heap operations for Priority Queues: Complexity and correctness