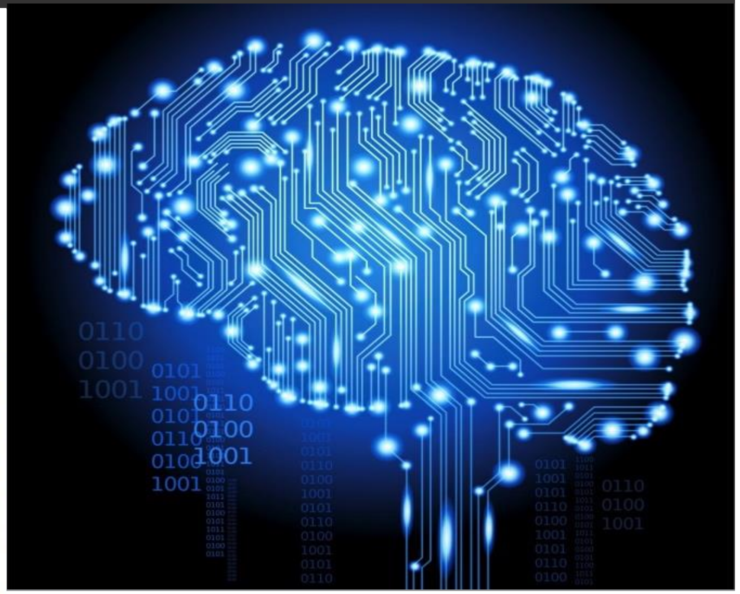
MONASH University

Information Technology

Priority Queues and Heaps

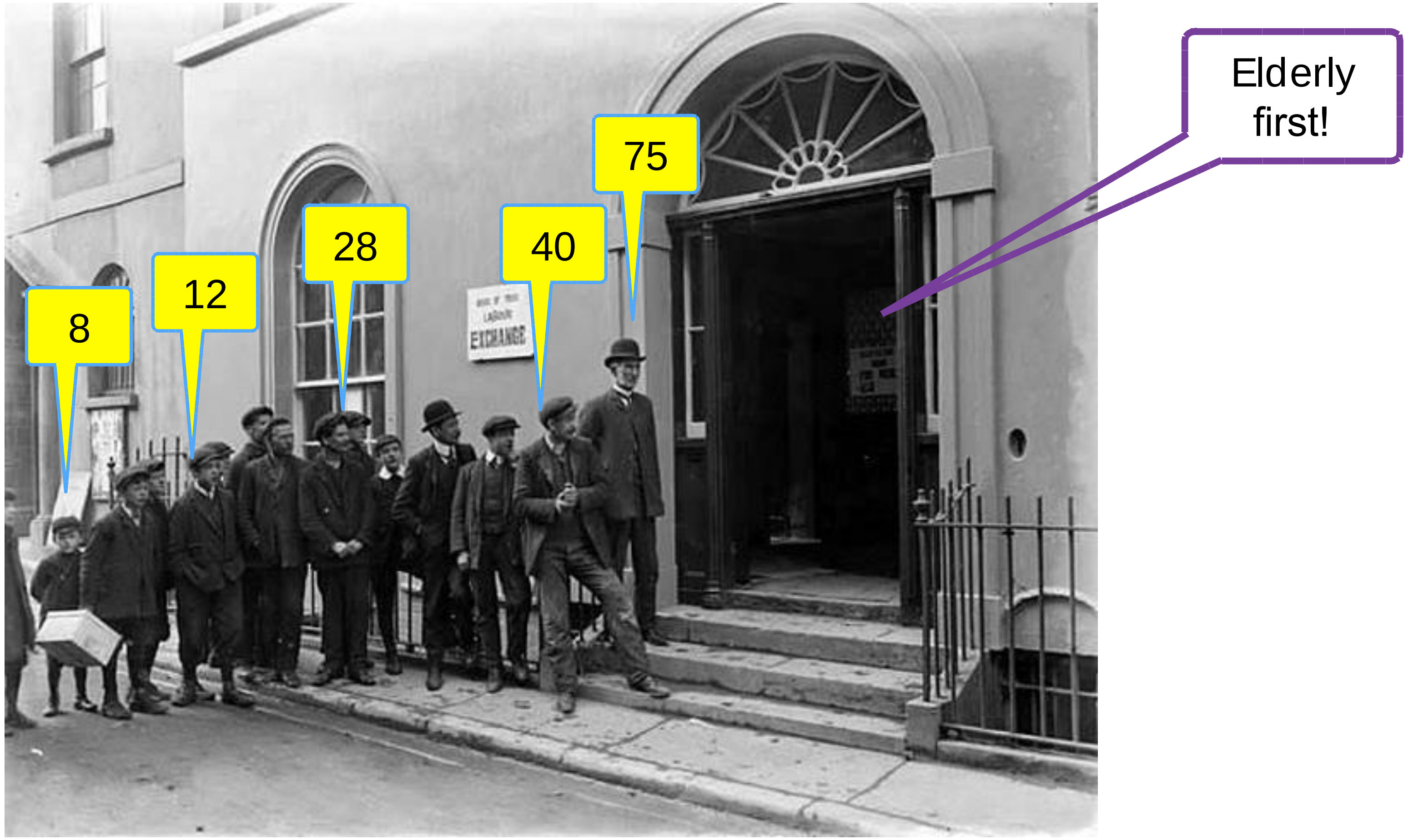
Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor



Objectives for this lecture

- Learn about Priority Queues
- Consider different implementations of Priority Queues
- Start to consider using Heaps to implement Priority Queues

“Form an orderly queue to the left..”



"Form an orderly queue to the left.."

Priority Queues

Priority Queue

Or lowest, they are dual

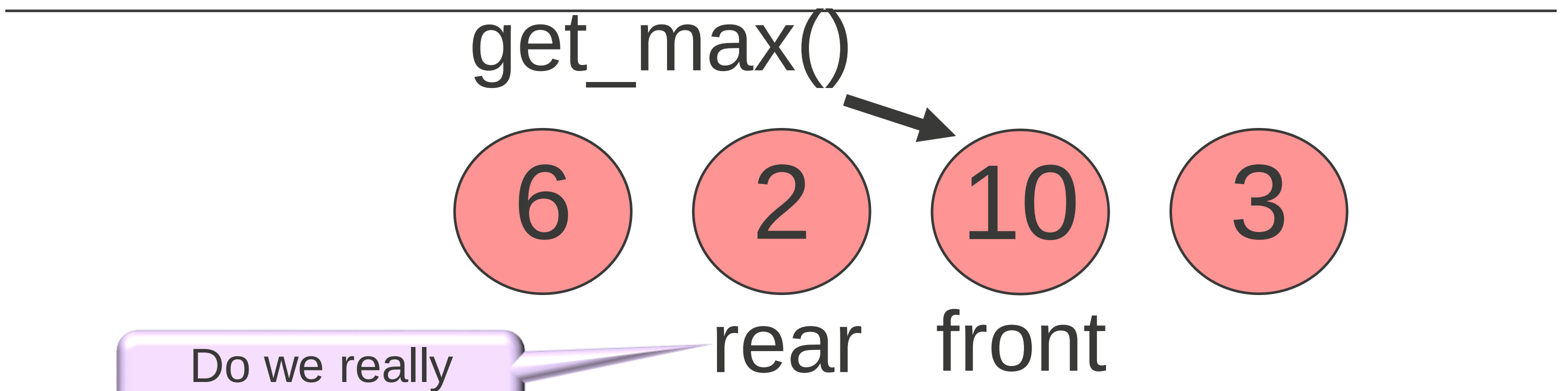
- **Each element has a numeric priority**
  - We process first the element with the highest priority
- **FIFO queue can be seen in this light: the priority is the amount of time spent in the queue**

Isn't this just a (reversed) sorted list?

No! Different operations.

- **Two main operations:**
  - `add(element)`: adds an element (which has some priority)
  - `get_max()`: serves the element with highest priority

Or `get_min()`, they are dual



Do we really need `rear`?

Uses of Priority Queues

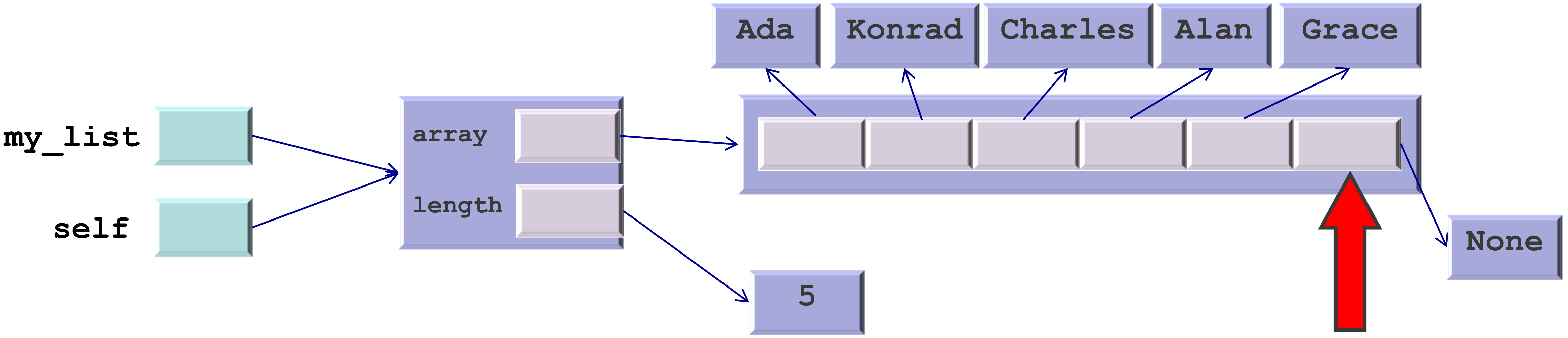
- Hospital emergency rooms
- Job scheduling
- Discrete event simulations
- Graph algorithms
- Genetic algorithms

Implementing Priority Queues

- **We need to implement the usual boring operations**
  - is_empty, __len__, __init__
- **Plus the two interesting ones**
  - `get_max()`: deletes the max element and returns it
  - `add(element)`: adds element to the priority queue
- **Many possible ways, for example:**
  - Arrays lists (sorted and unsorted)
  - Linked lists (sorted and unsorted)
  - Binary search trees

Unsorted array list implementation

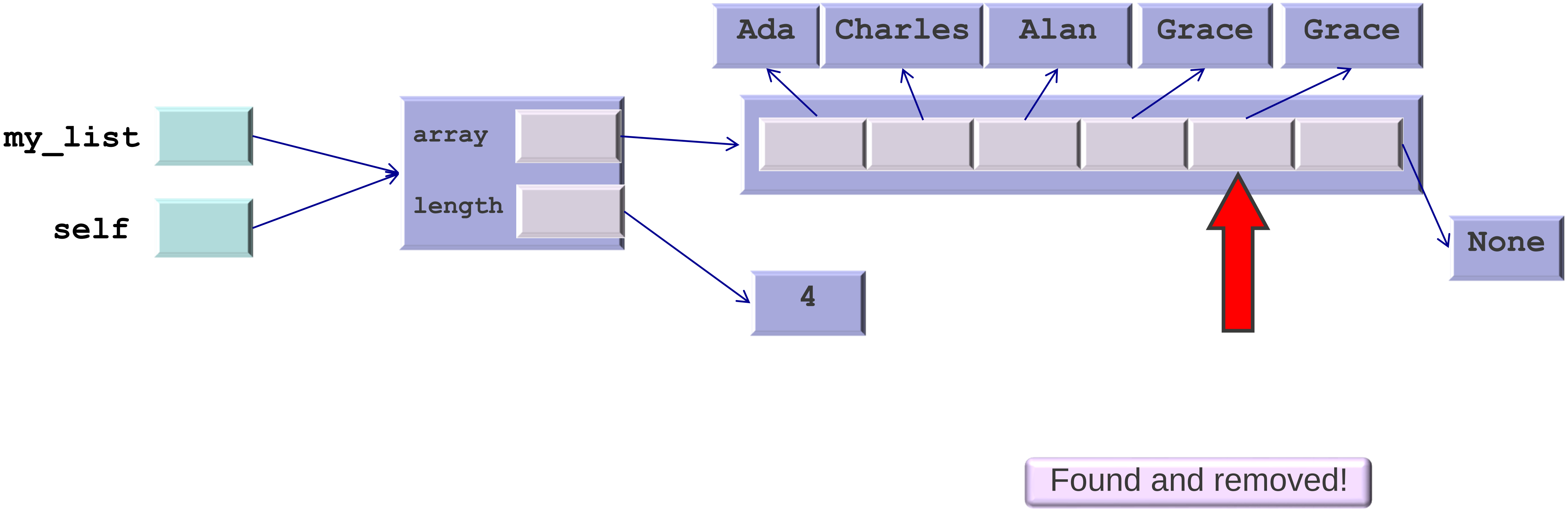
- **get_max: find maximum element, remove it, and shuffle remaining elements**



Find max (Konrad) and remove it

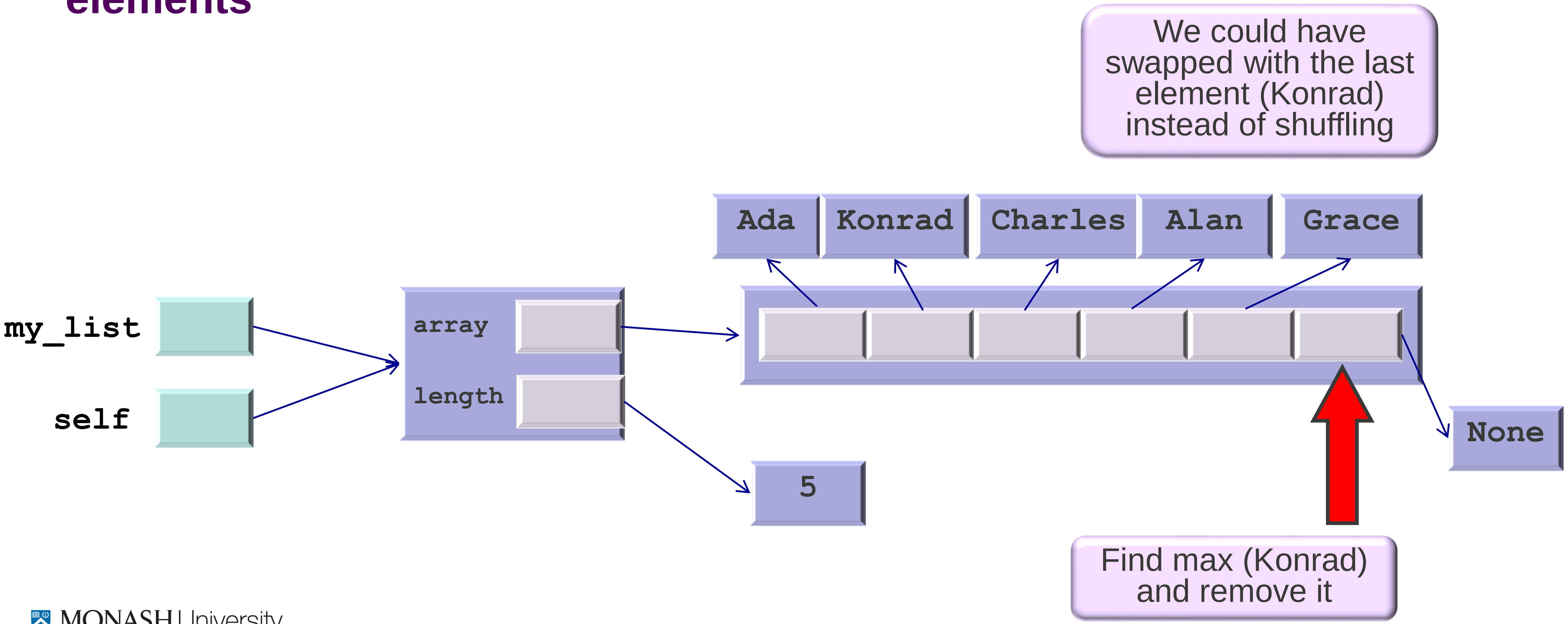
Unsorted array list implementation

- get_max: find maximum element, remove it, and shuffle remaining elements



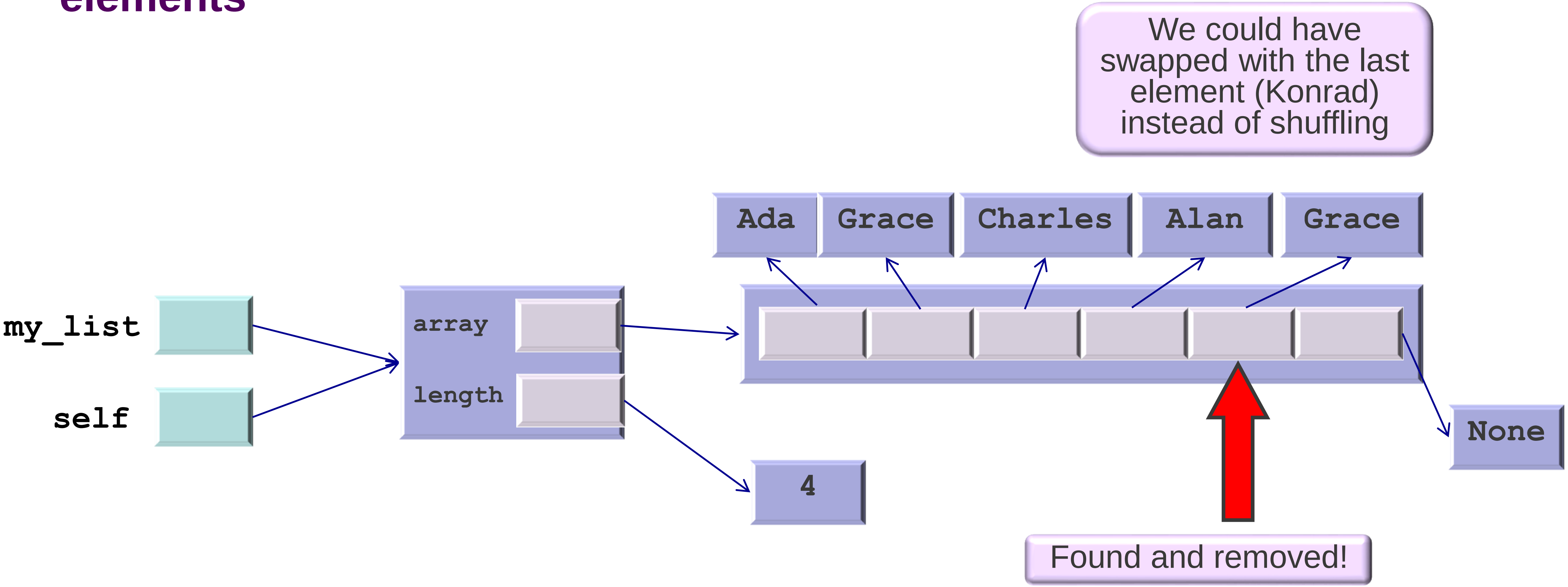
Unsorted array list implementation

- **get_max: find maximum element, remove it, and shuffle remaining elements**



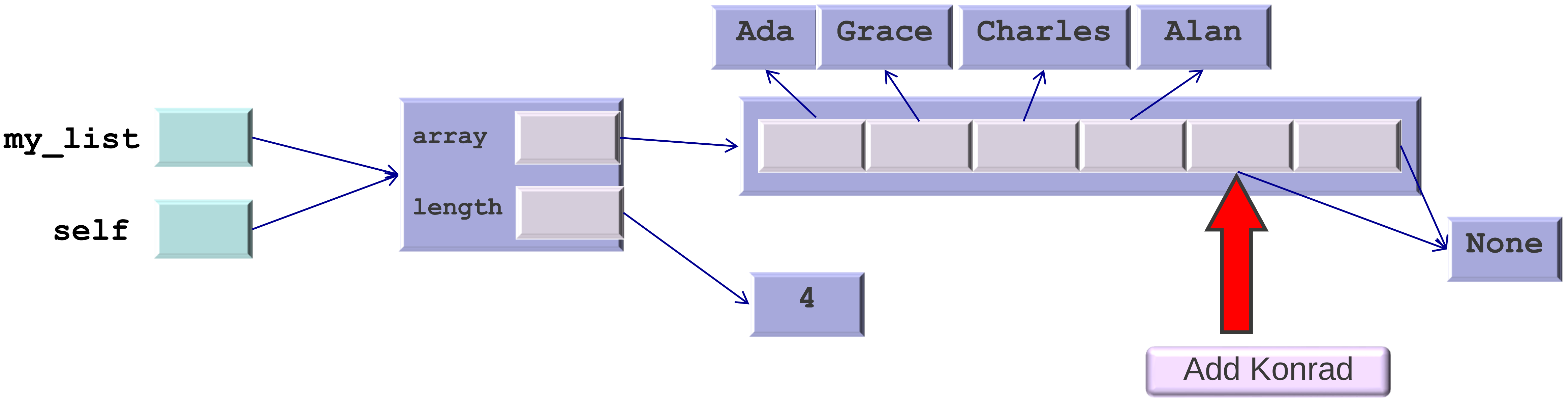
Unsorted array list implementation

- **get_max: find maximum element, remove it, and shuffle remaining elements**



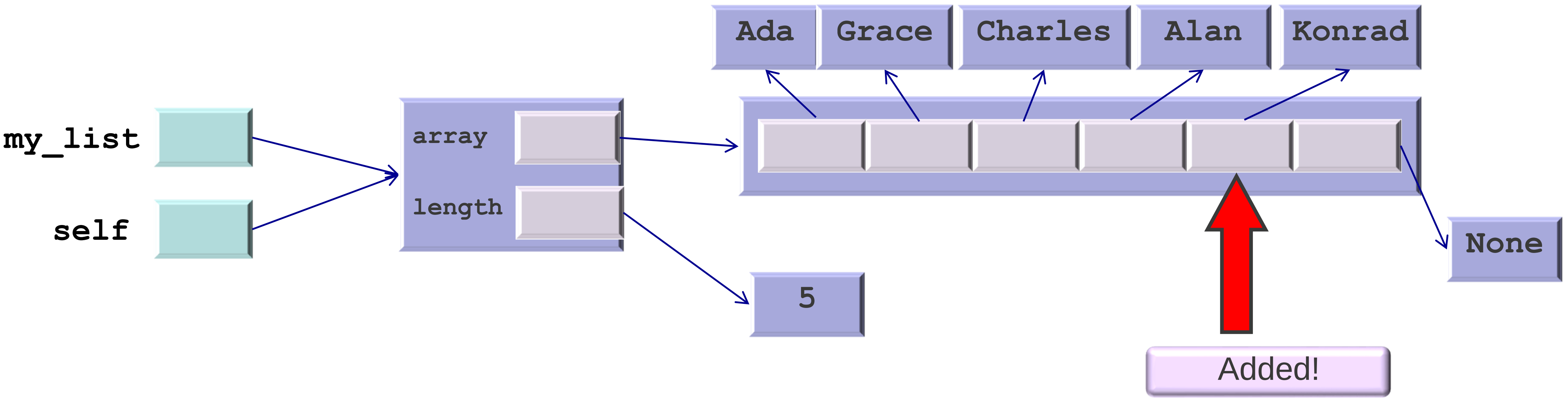
Unsorted array list implementation

- get_max: find maximum element, remove it, and shuffle remaining elements
- add: place at the back



Unsorted array list implementation

- get_max: find maximum element, remove it, and shuffle remaining elements
- add: place at the back



Complexity analysis for unsorted lists

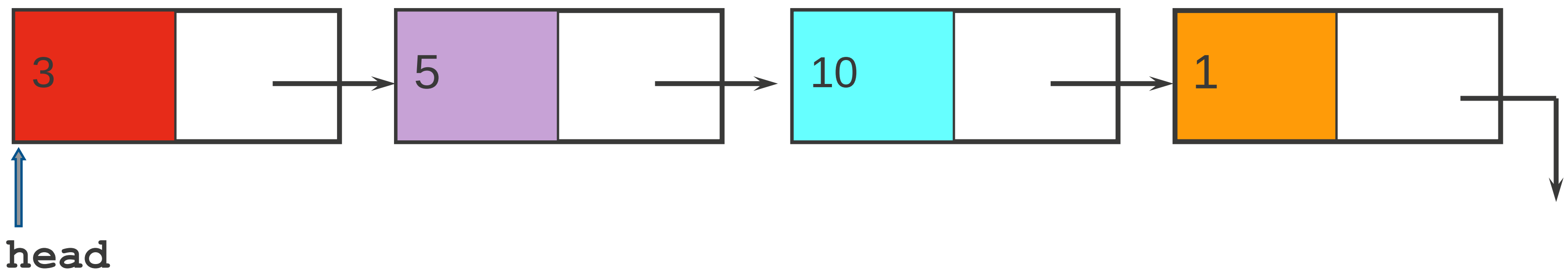
- **add:**
  - Worst (and best): O(1)
  - Since we just copy to last cell (pointed by length) and increment
- **get_max**
  - Worst: O(N)*OComp, where N is the size of the queue
  - Since we traverse to compute max O(N)*OComp + shuffle O(N)

And best is also O(N)*OComp, since we have to find the max, which means we need to traverse all elements, comparing for each.

O(N)*OComp even if we swap with the last element rather than shuffle

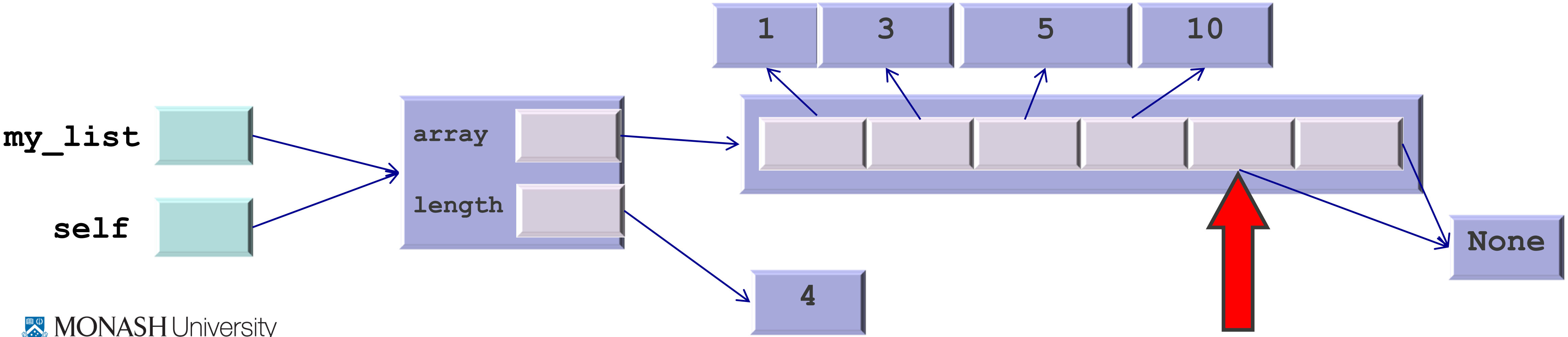
Unsorted linked list implementation

- **`add`: create node at the head of the list**
- **`get_max`: find the max element and bypass the node**
- **Same complexity as unsorted array list**
  - add: O(1)
  - get_max: O(N)*OComp traversal to compute the max + O(1) to bypass the node



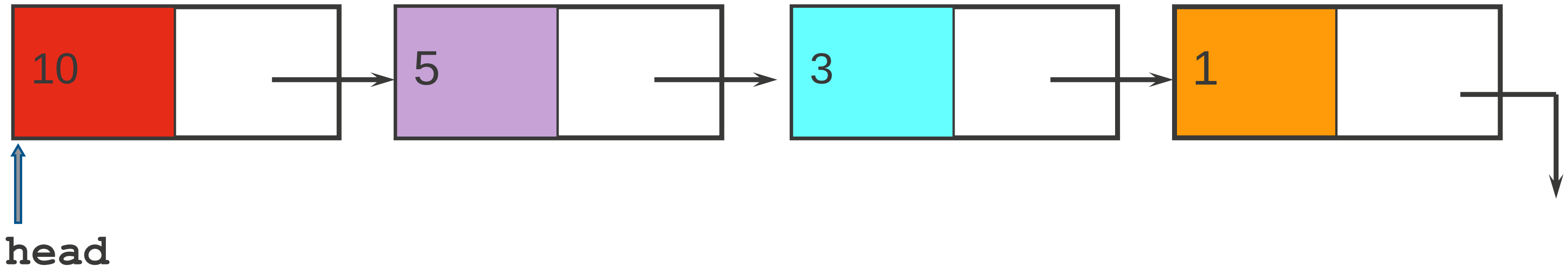
Sorted array list implementations

- **add**
  - Best O(log N)*OComp (largest element): since O(log N)* OComp for search + O(1) to add it as last
  - Worst O(N) (smallest one): since O(log N)*OComp for the search + O(N) shuffle (assuming OComp < O(N))
- **get_max**
  - O(1): since max element is always at **`N-1`**



Sorted linked list implementation

- **Should it be increasing or decreasing order?**
  - We want to remove greatest: so decreasing order
    - Note that increasing order would require traversing the list!
    - Could we just add a pointer to the last node? Not enough! (need back links)
- **add:**
  - Best: O(1)*OComp position found at head (largest element)
  - Worst: O(N)*OComp search to find the last place (smallest one)
- **get_max:**
  - Best and worst: O(1) to take out the head node



Priority Queues using linear structures…

- **Summary: the worst case time complexity of the operations (*OComp) is:**

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)

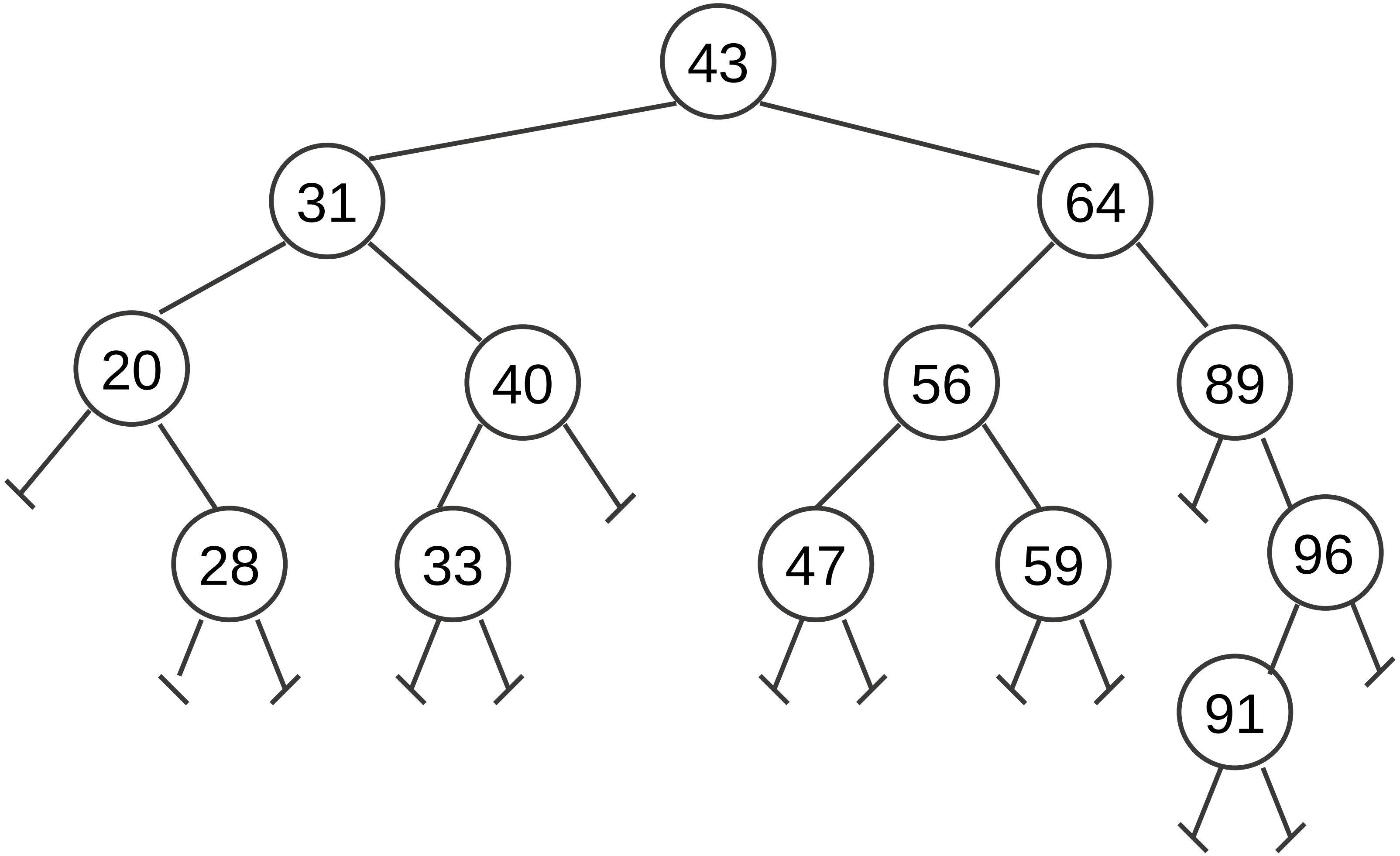
- **Let's try non-linear structures..**

Binary search tree implementation

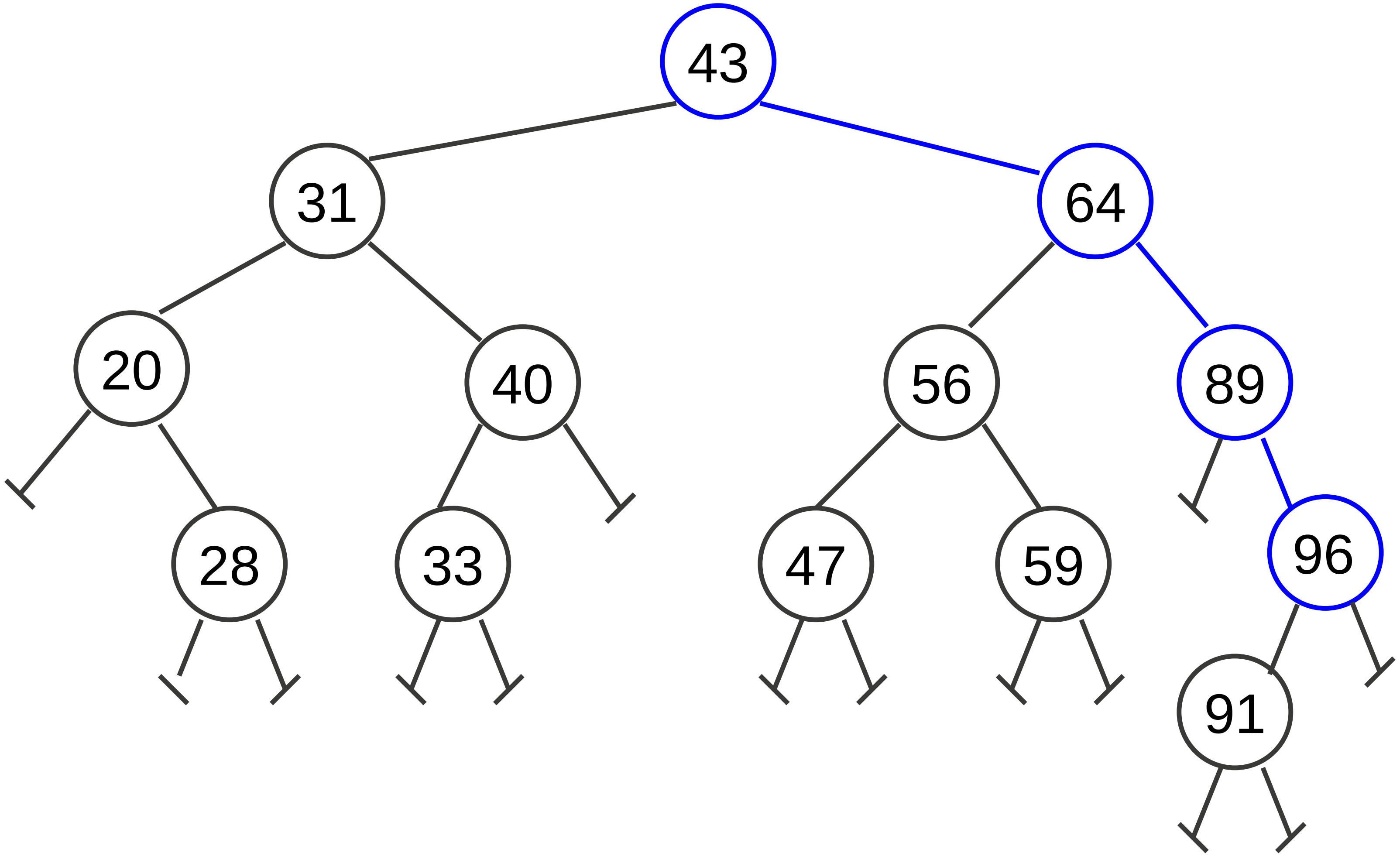
- **Given the following binary search tree node class:**

```
class BinarySearchTreeNode(Generic[T]):
    def __init__(self, item: T = None) -> None:
        self.item = item
        self.left = None
        self.right = None
class BinarySearchTree(Generic[T]):
    def __init__(self) -> None:
        self.root = None
```

- **Implement the `get_max` operation**
  – Where is the max element in a binary search tree?



In this particular example the max element is 96

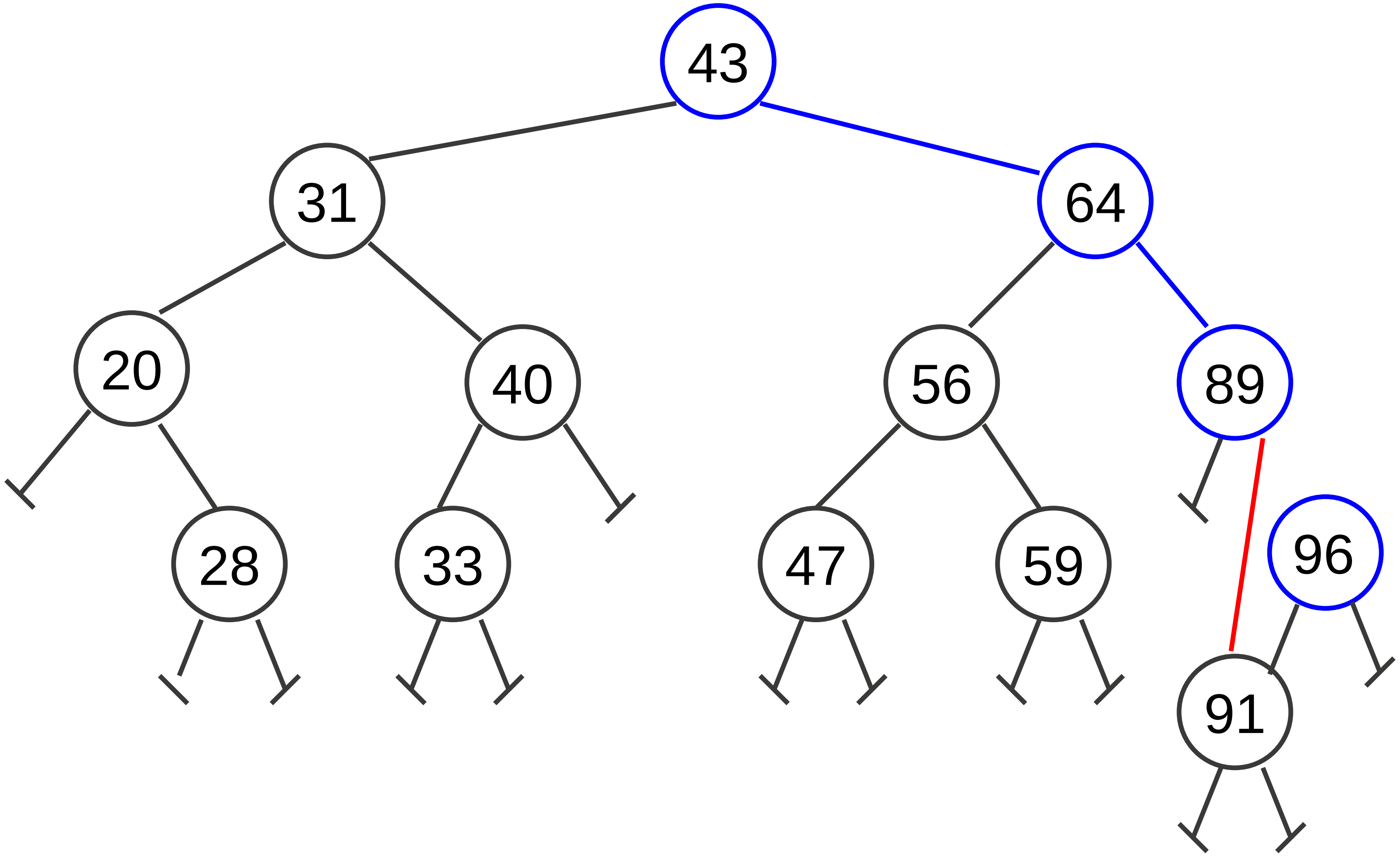


In this particular example the max element is 96

In general, it is the rightmost node (always go right until the node does not have a right child)

Complexity?

O(Depth)



In this particular example the max element is 96

Complexity?

In general, it is the rightmost node (always go right until the node does not have a right child)

O(Depth)

Deleting it is easy: point parent to child of deleted node

Lets start by simply returning the max without deleting

If priority queue is empty, raise an exception

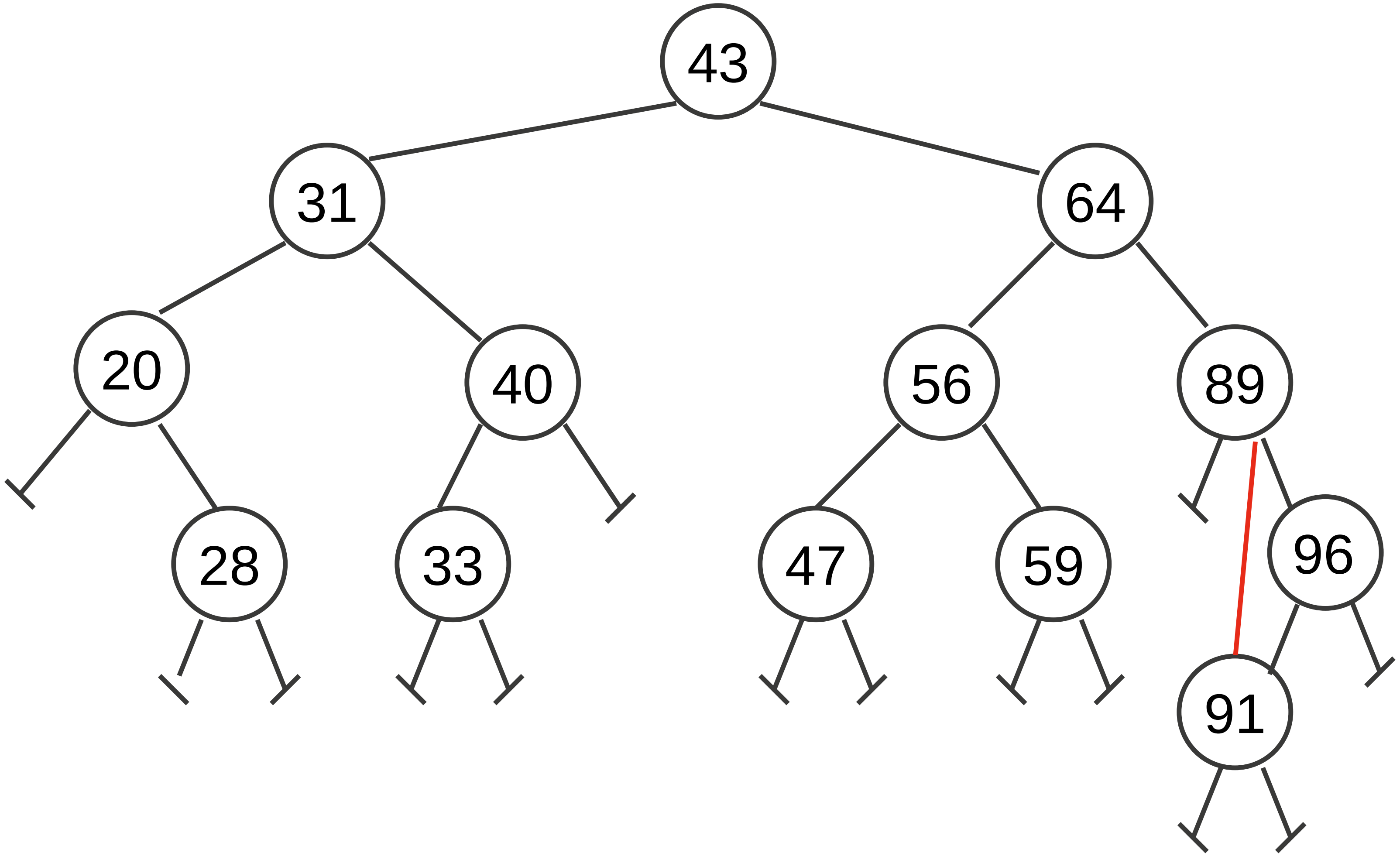
```
def get_max(self) -> T:
    if self.root is None:
       raise ValueError("Priority queue is empty")
    else:
        return self.get_max_aux(self.root)


def get_max_aux(self, current: BinarySearchTreeNode[T]) -> T:
    if current.right is None: # base case: at max
        return current.item
    else:
        return self.get_max_aux(current.right)
```

Complexity?

O(Depth)

But remember, this does not delete! Let's delete



What do we need to do to remove the max?

Set **`parent.right`** to **`max.left`**

Let's pass the parent as parameter

```
def get_max(self) -> T:
    if self.root is None:
        raise ValueError("Heap 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)
```

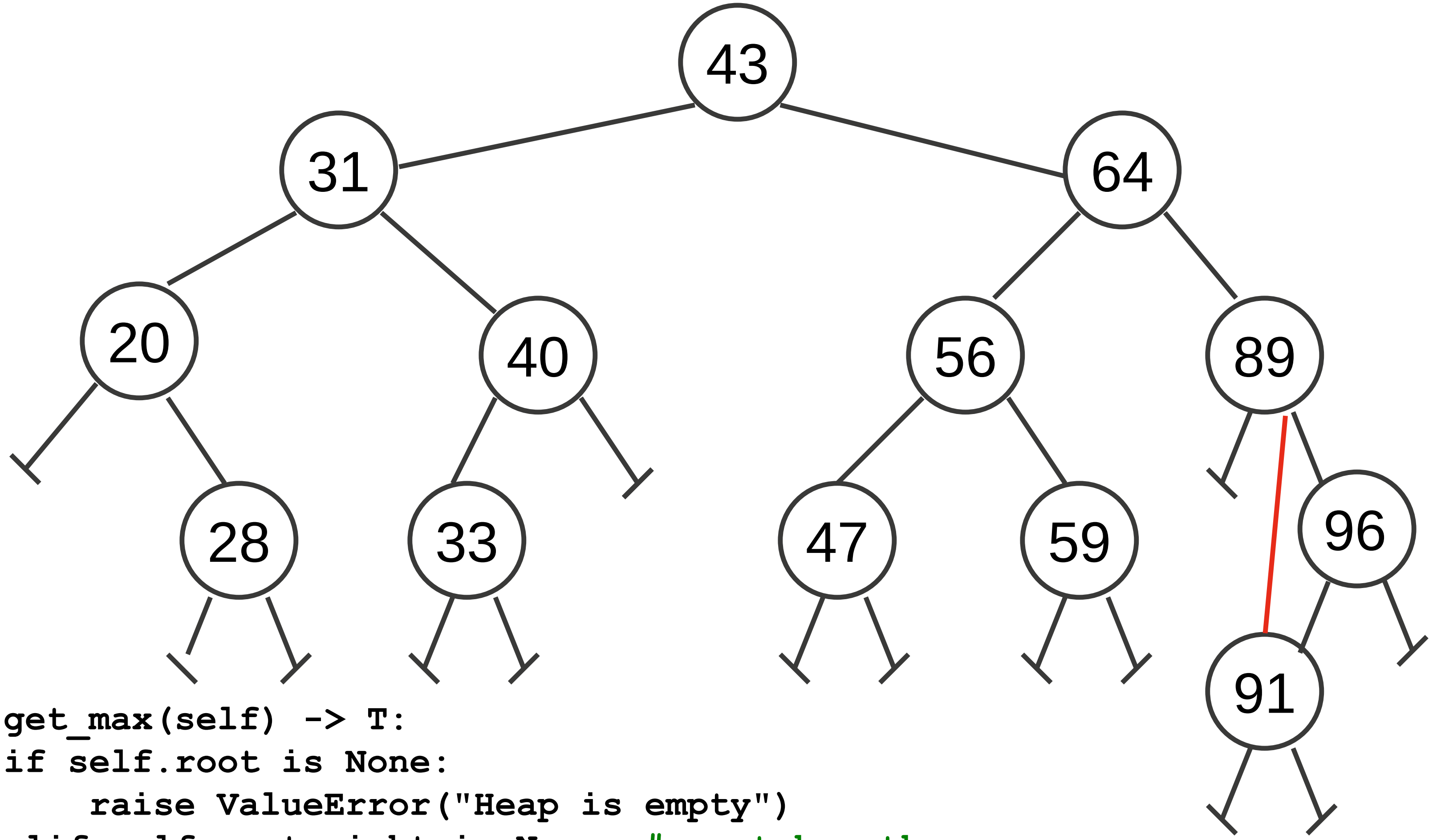
What if we swap these two?

We need the parent to delete the Max node

Removed type hinting for lack of space

```
def get_max_aux(self, current, parent) -> T:
    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)
```

It is an easy deletion case: leaf with only one child.




```
def get_max(self) -> T:
    if self.root is None:
        raise ValueError("Heap is empty")
    elif self.root.right is None: # root has the max
        …
    else:
        return self.get_max_aux(self.root.right, self.root)

def get_max_aux(self, current, parent) -> T:
    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)
```

```python
def get_max(self) -> T:
    if self.root is None:
        raise ValueError("Heap 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)
```

We can instead pass only the parent

```python
def get_max_aux(self, parent: BinarySearchTreeNode[T]) -> T:
    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)
```

A better implementation

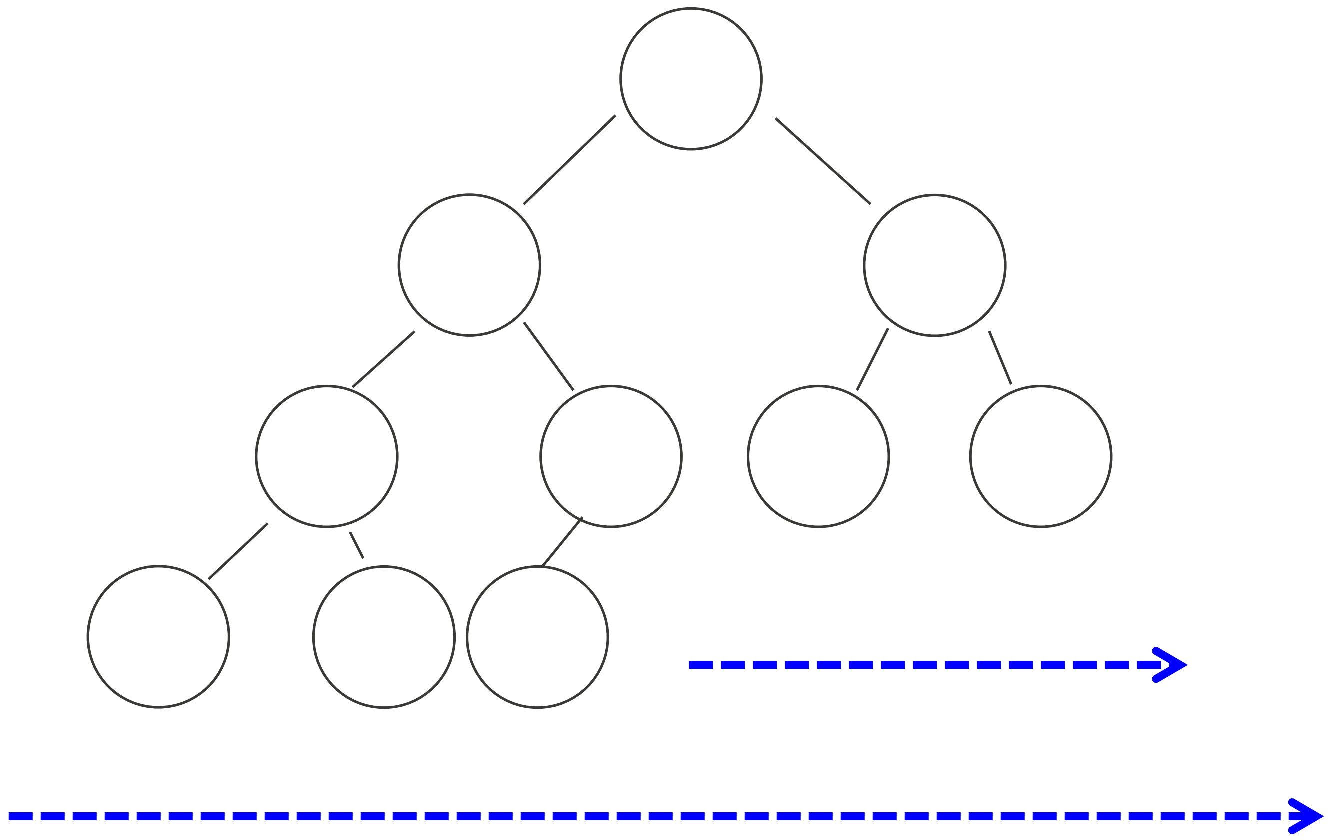
- **Each of the previous choices has one O(N) operation**
  - O(Depth) for the binary tree with Depth being N-1 if unbalanced
- **Can we do better?**
  - Of course
- **Use a (max) heap**
  - One could also use a min-heap, where the lower the number the more important the item
  - In this unit we use max-heaps but the ideas are the same for a min-heap

Heaps

Basics of heaps

- **View the elements as a binary tree with two special properties:**
  - Complete
  - Heap-ordered
- **In a complete binary tree:**
  - Every level -- except possibly the last one -- is full
  - The last level is filled from the left

Complete binary tree

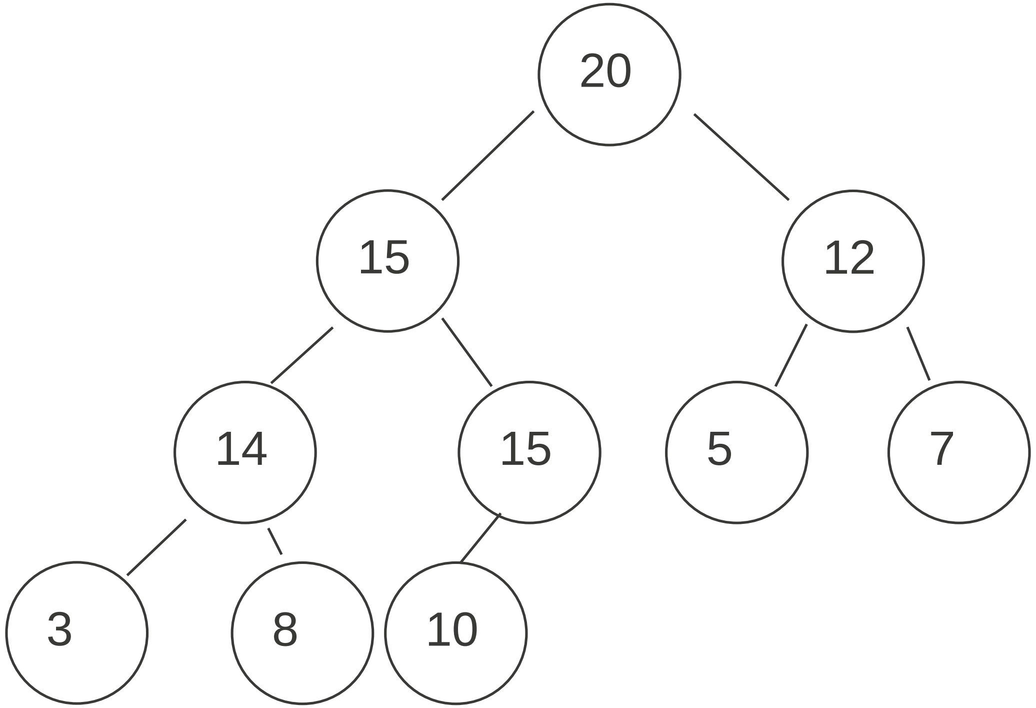


Heap-ordered (max-heap)

- **Heap-ordered**
  - Each child is smaller than (or equal to) its parent
- **Note that this imposes no conditions on siblings!**
  - Do binary search tree invariants impose conditions on siblings?

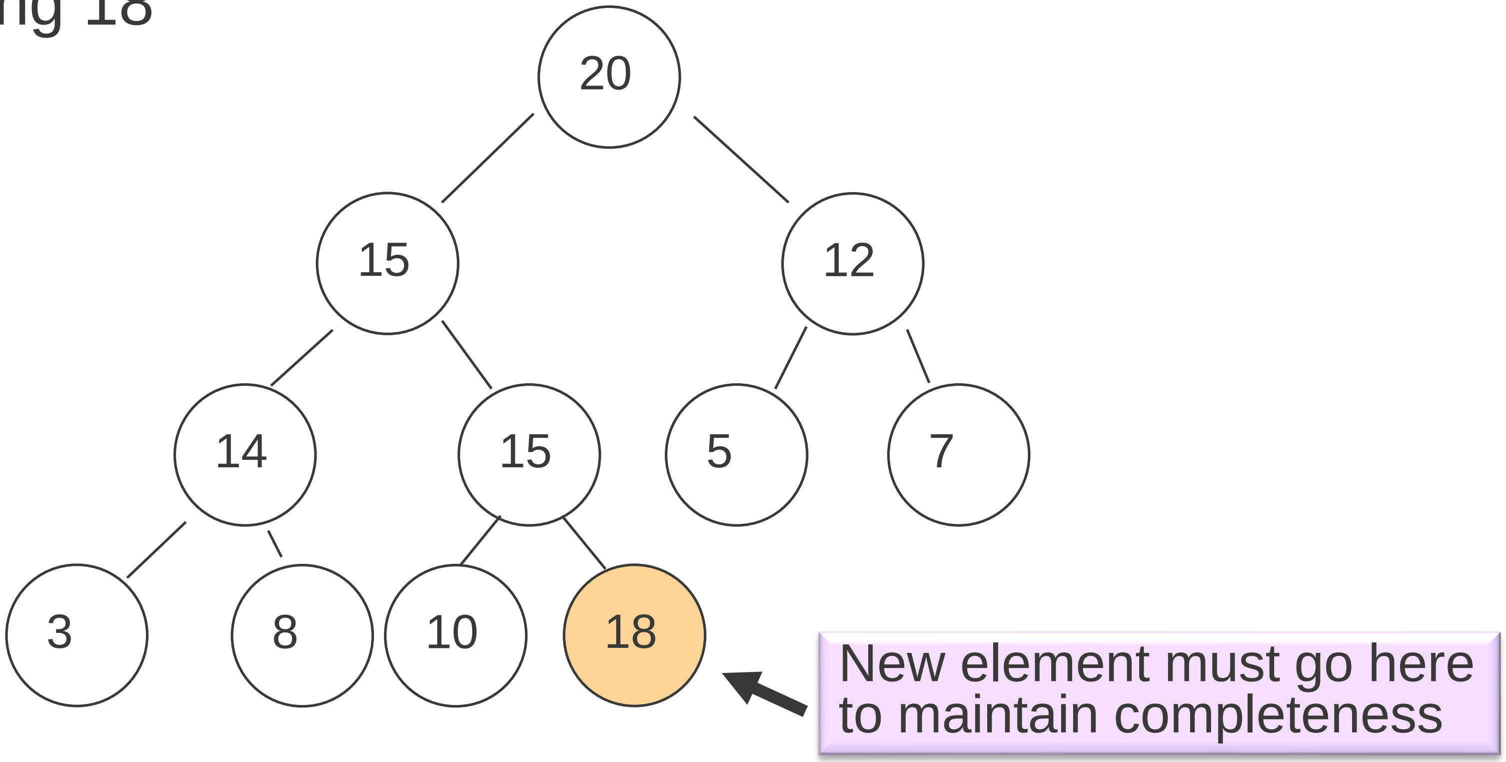
Yes! The keys of all nodes on the right tree are known to be greater than those on the left.

Complete, heap-ordered binary tree



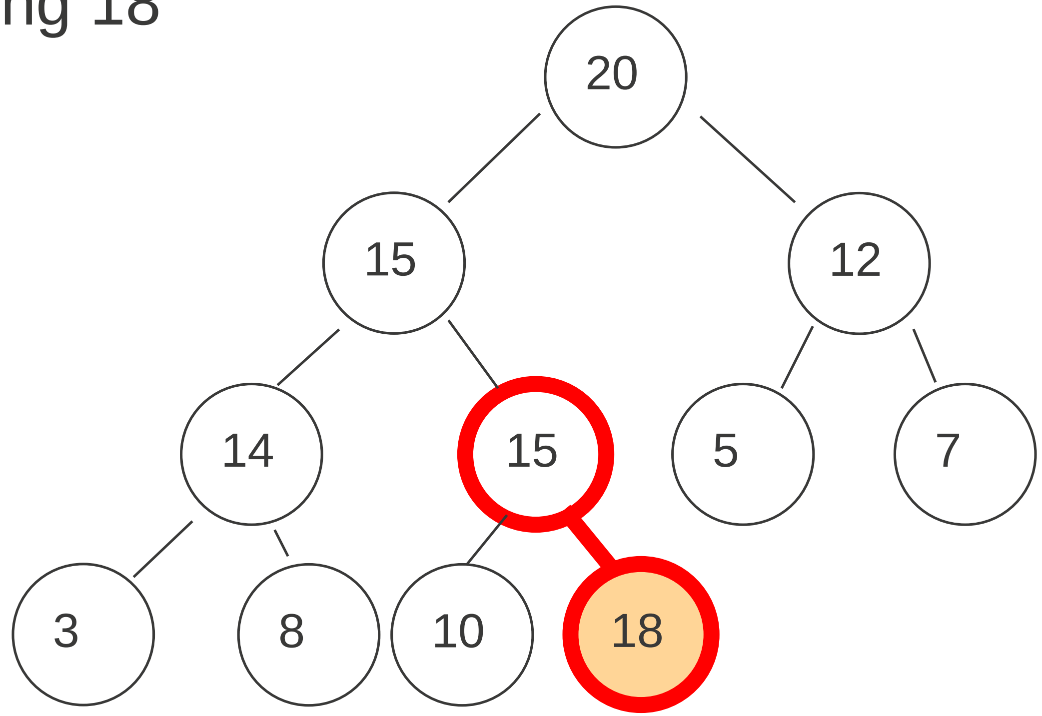
Insertion (`add`)

adding 18



Insertion (`add`)

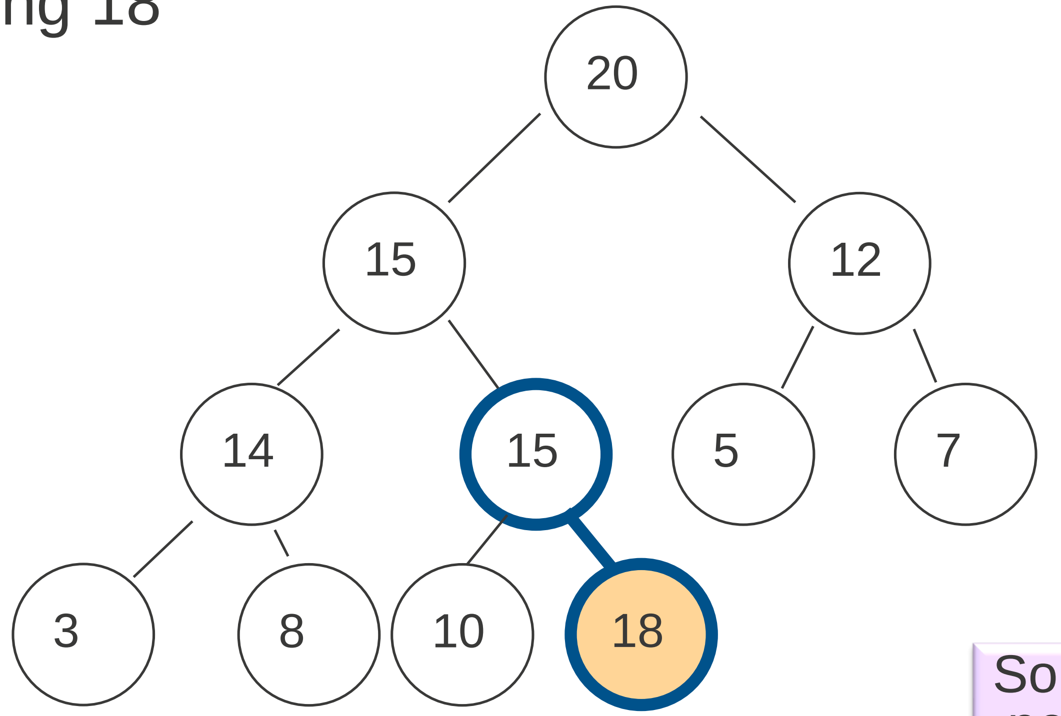
adding 18



But we’ve broken the heap-ordering

Insertion (`add`)

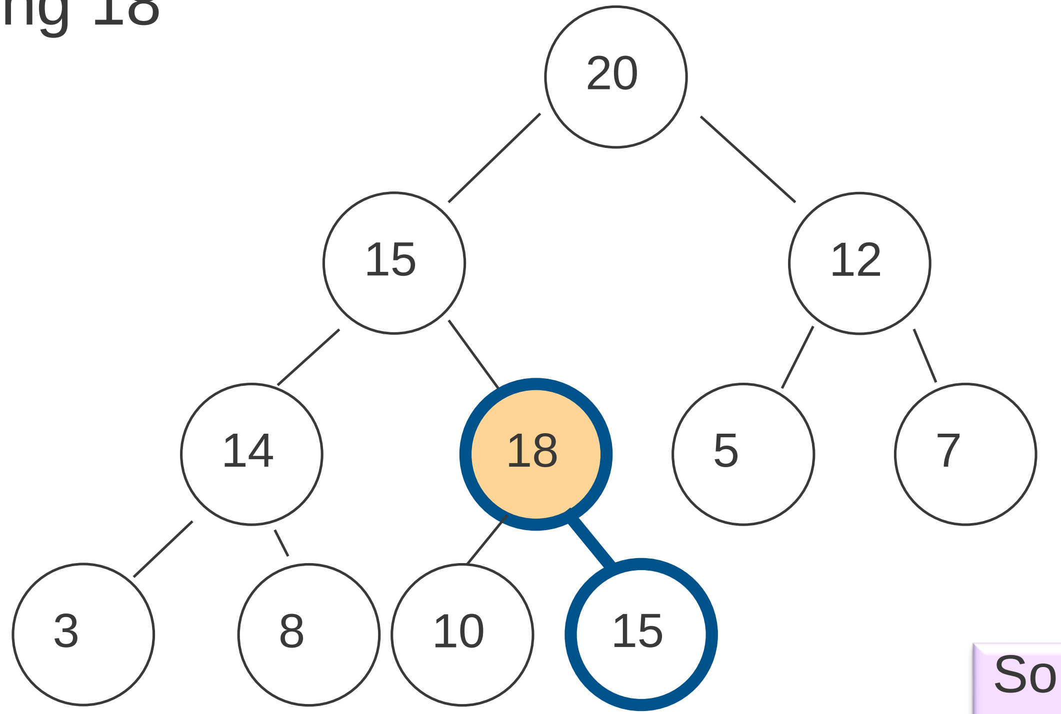
adding 18



Solution: raise the new element by swapping

Insertion (`add`)

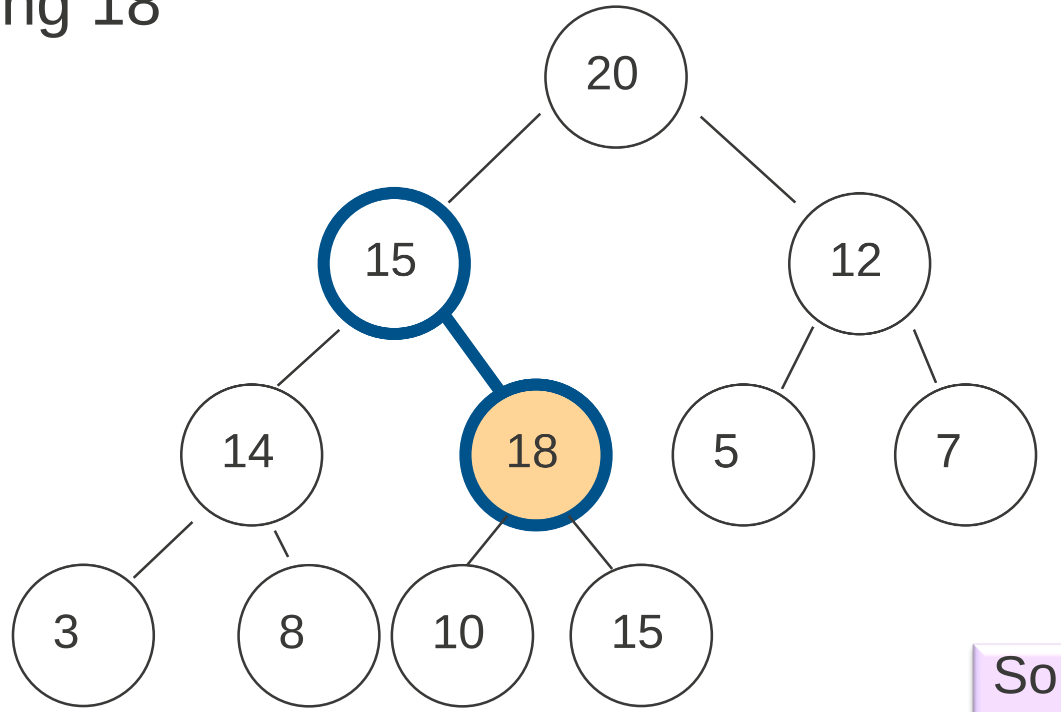
adding 18



Solution: raise the new element by swapping

Insertion (`add`)

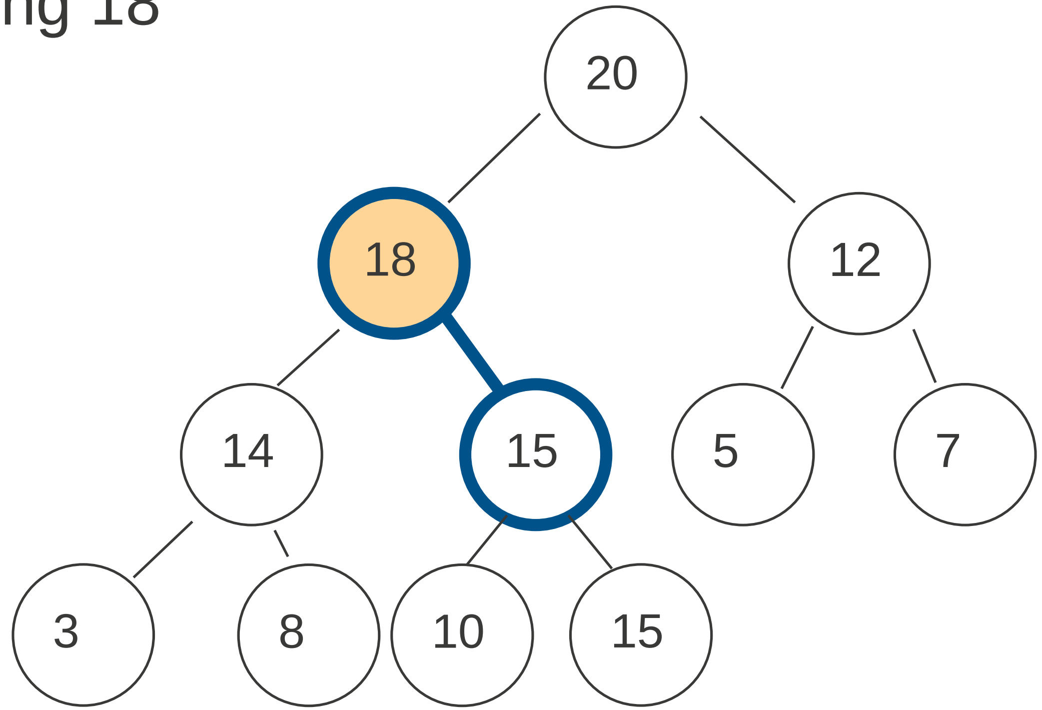
adding 18



Solution: raise the new element by swapping

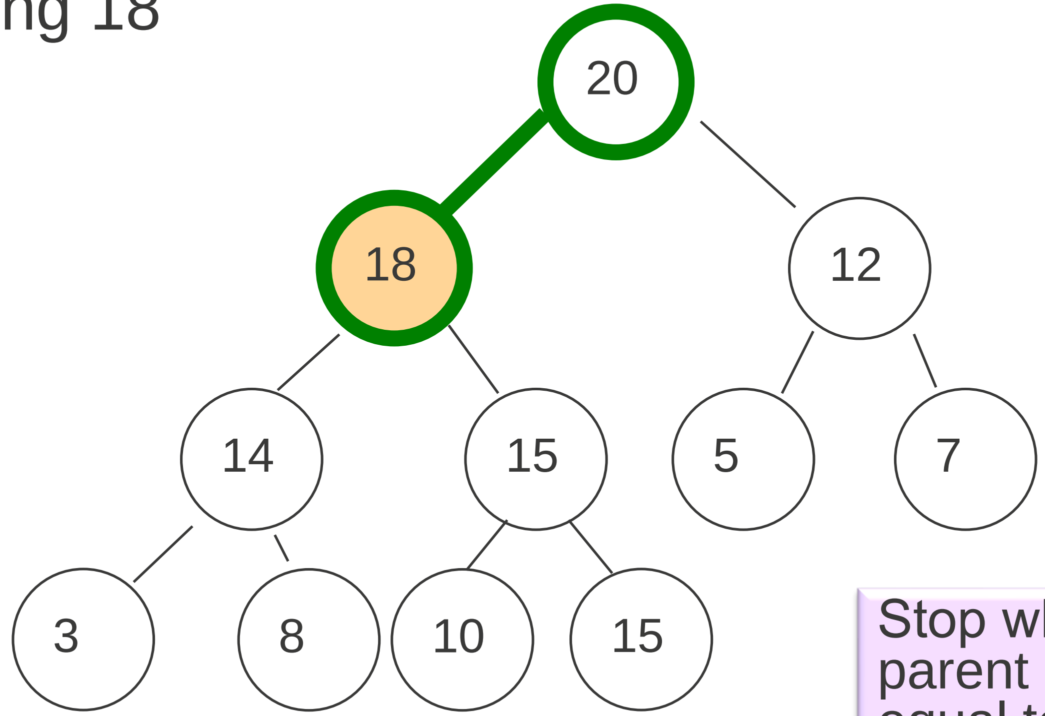
Insertion (`add`)

adding 18



Insertion (`add`)

adding 18

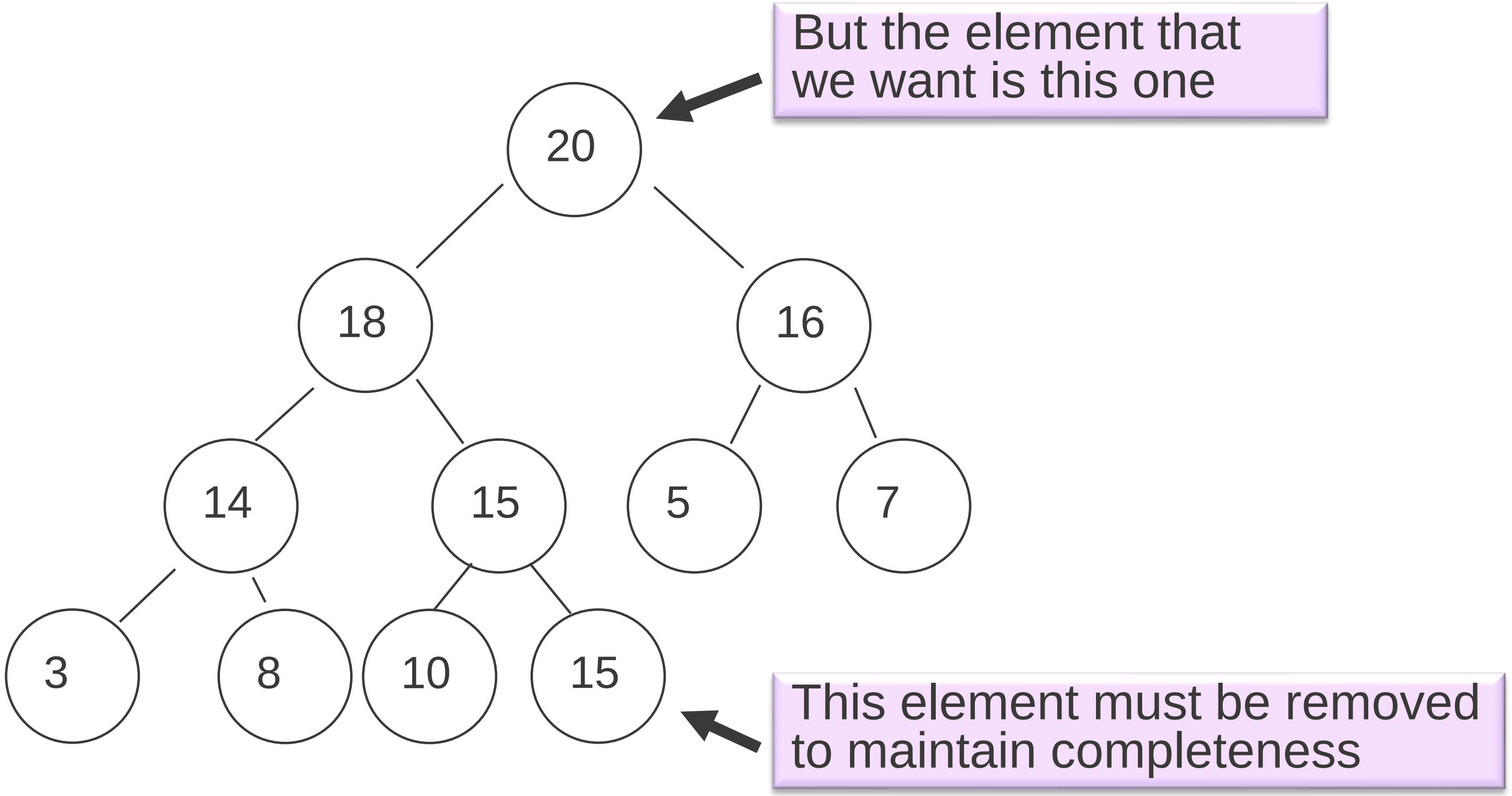


Stop when the element's parent is greater than or equal to the new one

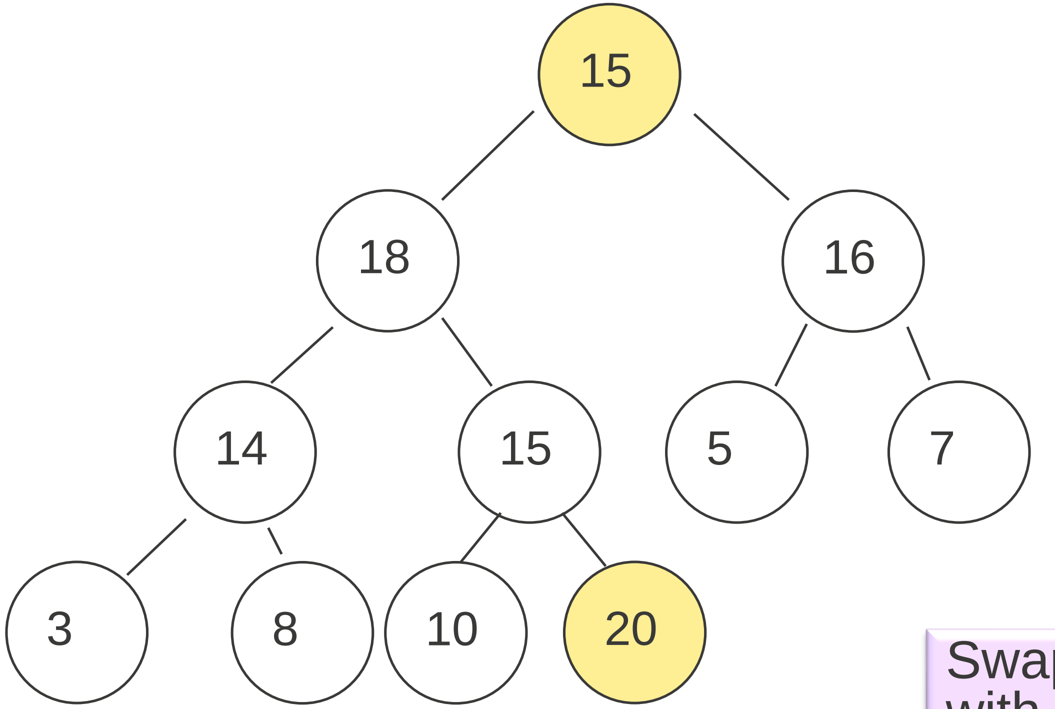
Correctness of insertion

- **Is this algorithm provably correct?**
- **What are the invariants of the heap class implemented with trees?**
  - the tree is binary,
  - complete, and
  - heap sorted
- **Remember: to prove the correctness of add we have to prove that it maintains the invariants**
  - When can the tree become non-binary?
    - When connecting a new node
  - When can completeness be affected?
    - When connecting a new node
  - When can heap sortedness be affected?
    - When raising (swapping) nodes (after connecting a new one)

Retrieval (`get_max`)

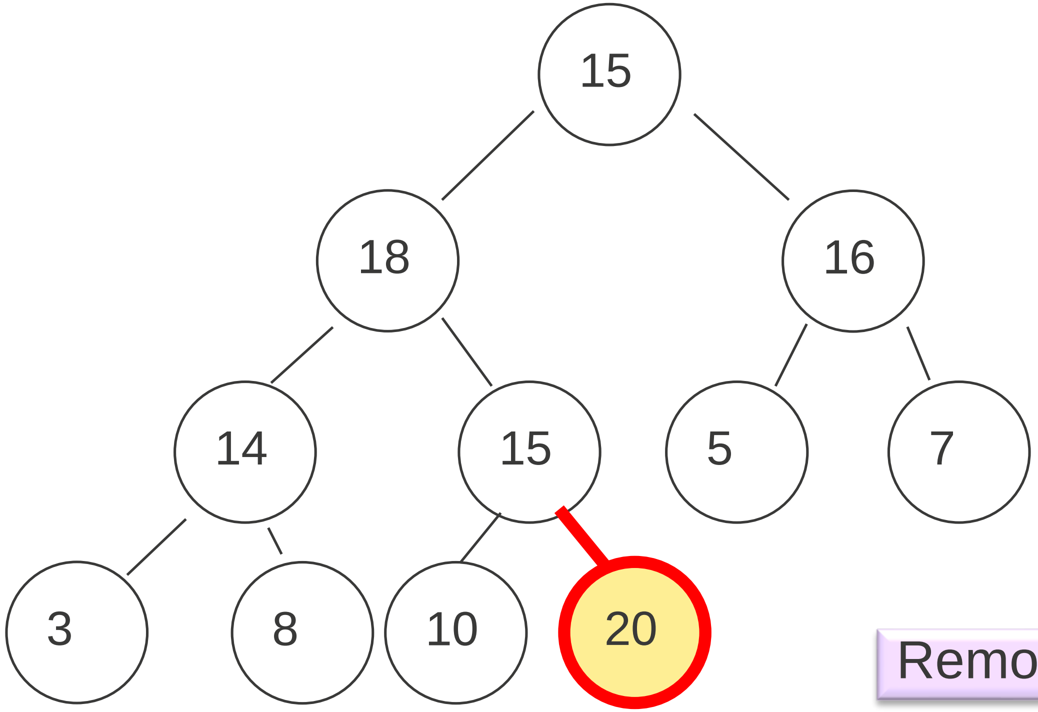


Retrieval (`get_max`)



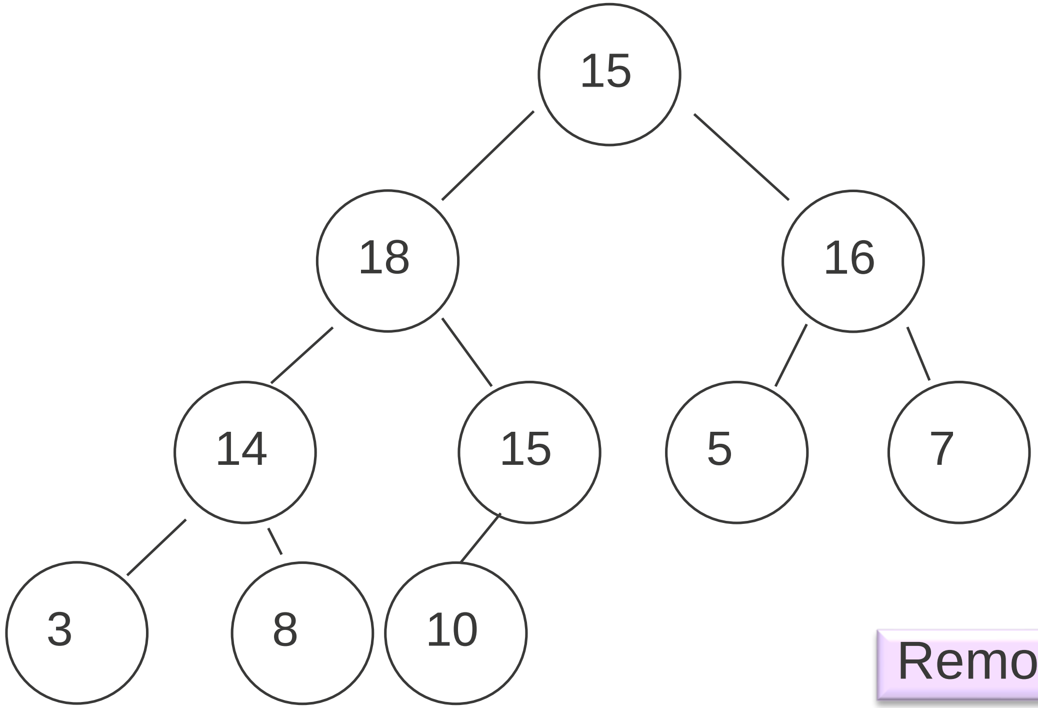
Swap this position with the maximum

Retrieval (`get_max`)



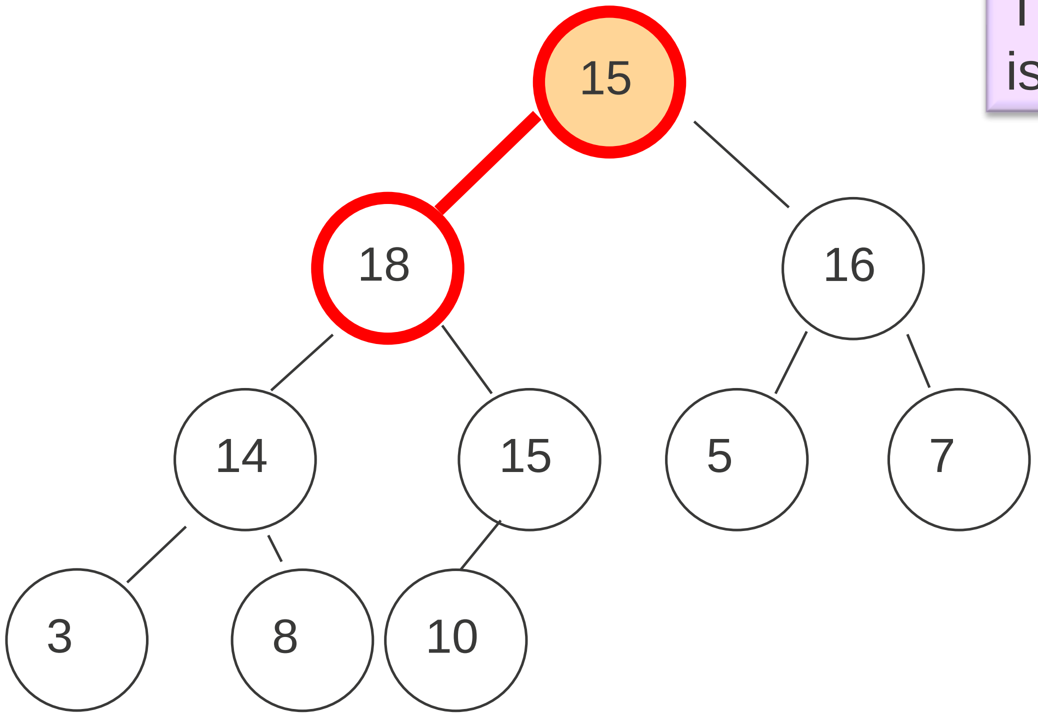
Remove the maximum

Retrieval (`get_max`)



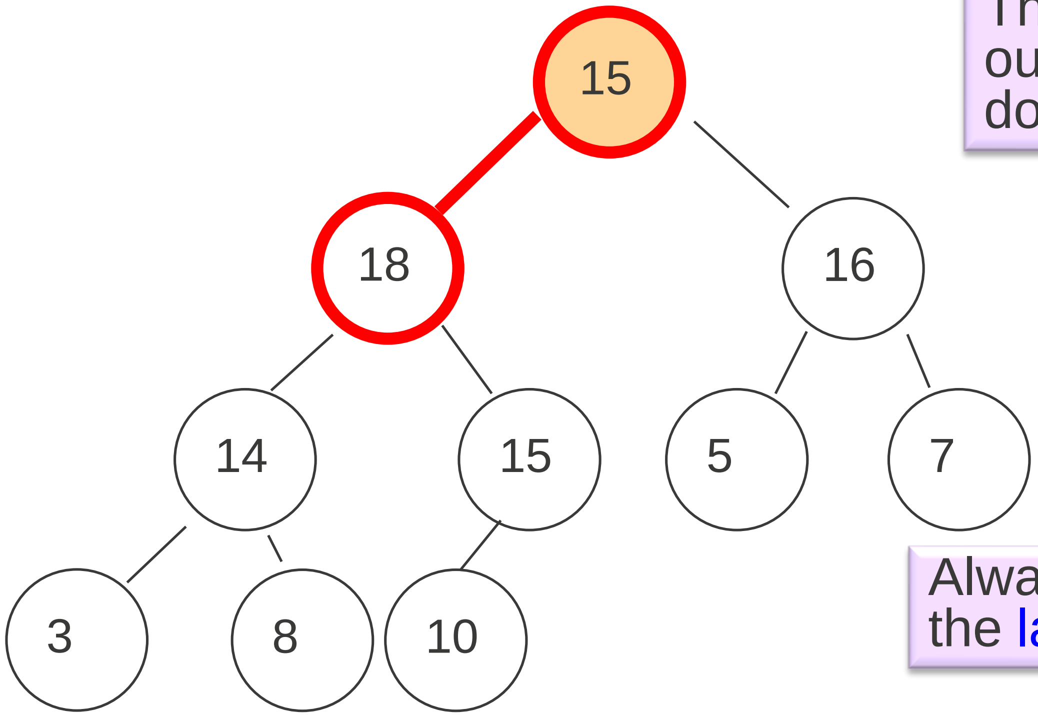
Remove the maximum

Retrieval (`get_max`)



The heap-ordering is broken again

Retrieval (`get_max`)

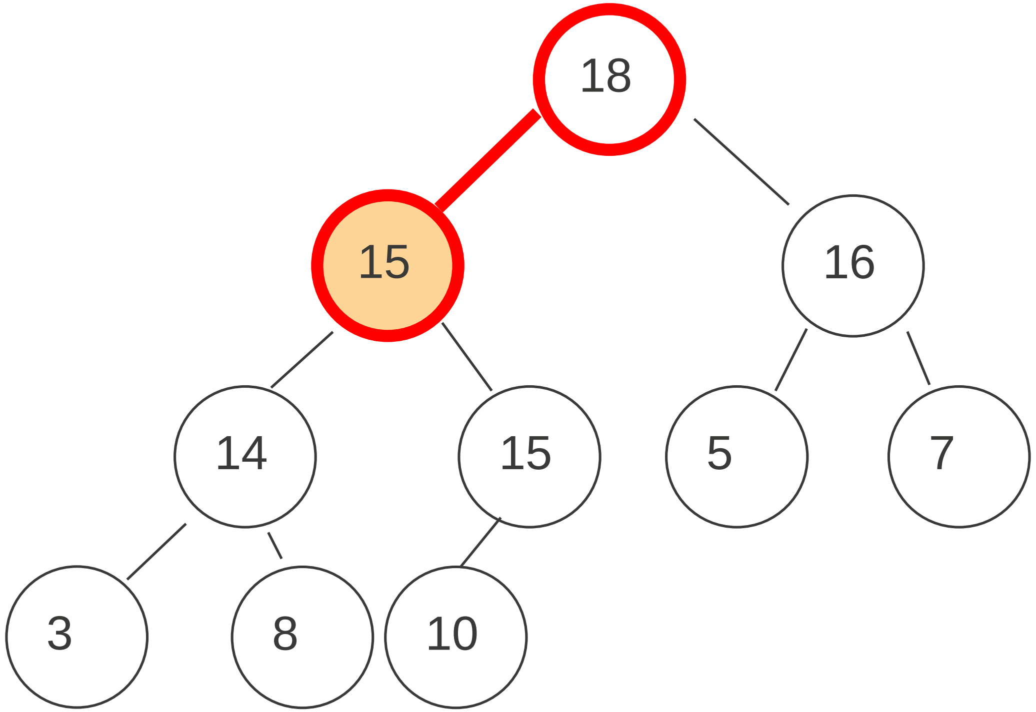


This time, move the out-of-place element downwards (sink)

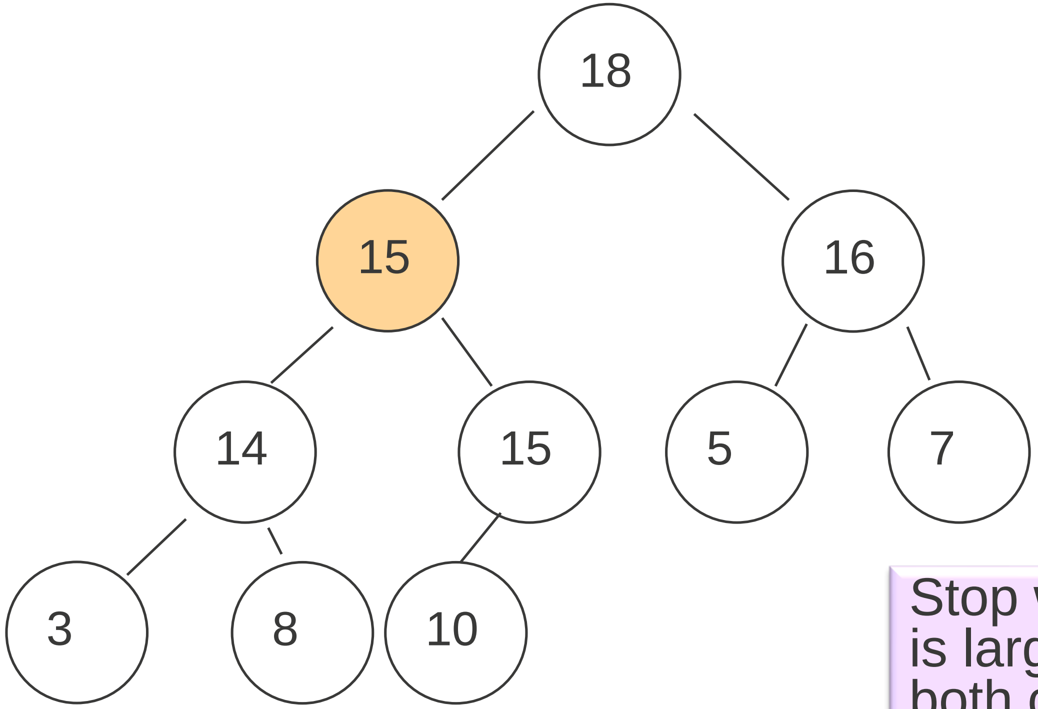
Always swapping with the larger child. Why?

Because otherwise we might not be heap-sorted (16 cannot be the root)

Retrieval (`get_max`)



Retrieval (`get_max`)



Stop when the element is larger than or equal to both of its children
(or if it hits the bottom)

Correctness again

- **Is this algorithm correct too?**
- **Again, what are the invariants of the heap class implemented with trees?**
  - the tree is binary,
  - complete, and
  - heap sorted
- **We have to prove it maintain the tree invariants**
  - When can the tree become non-binary?
    - It cannot if we do not add nodes
  - When can completeness be affected?
    - When disconnecting a new node
  - When can heap sortedness be affected?
    - When sinking (swapping) nodes

Summary

- **We now know what a Priority Queue is and its two main operations**
  - add
  - get_max
- **You know the pros and cons of several possible implementations**
- **You know what a Heap is, i.e., a binary tree that is:**
  - Complete
  - Heap ordered
- **And you know how to use a Heap for the operations of Priority Queues**
  - Complexity and correctness