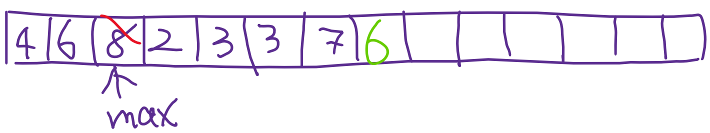
Priority Queue

- Priority Queue is a data structure that provides quick access to either the maximum item (or sometimes the minimum item) stored in it.
- Priority Queue is not a Queue. In a Queue, we enqueue an item to the tail and we dequeue the item that was the "oldest" in the queue (from the head). But in a Priority Queue, we do not really care where we insert an item to the structure, and we can only take out the maximum item in it.

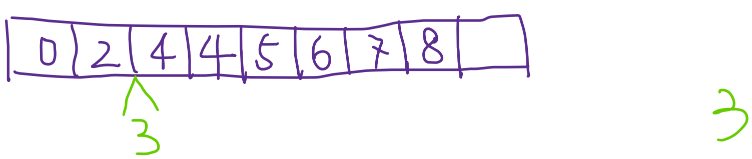
- The Priority Queue ADT provides at least the following operations:
  - $max()$: return the maximum item in a Priority Queue
  - $add(item)$: add an item to the Priority Queue; if it is larger than the maximum, it becomes the new maximum.
  - $pop\_max()$: pop out the maximum item; find the next maximum item to be the new maximum item.

- A naïve design of Priority Queue:
  - An ArrayList with a pointer pointing to the max.



    - $max()$ = return $self.list[max]$ $\quad // O(1)$
    - $add(item) = append(self.list, item)$ + one comparison with $self.list[max]$ to decide whether update the max pointer or not $\quad //O(1)$
    - $pop\_max() = del\ self.list[max]$ + move all items after max one spot to the left + linear search for the new max $\quad //\ O(n)$

- Another naïve design of Priority Queue:
  - A sorted ArrayList:



    - $max() = self.list[-1] \quad //O(1)$
    - $add(item)$ = binary search for the correct spot for the $item$ + move everything after the found spot one spot to the right + insert $item$ to the correct spot $\quad //O(n)$
    - $pop\_\text{max}\ () = del\ self.list[-1] \quad //O(1)$

1. Is possible that both $add(item)$ and $pop\_max()$ have a constant running time?

- [Theorem] Any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.
  - In other words, as long as a sorting algorithm is comparison-based, its running time is at least $\Theta(n \lg n)$ in the worst case.

- As an aside, here is a non-comparison-based sorting algorithm: how to sort a deck of cards? Create a pile for each rank.

- Using $add(item)$ and $pop\_max()$, we can create the following sorting algorithm:

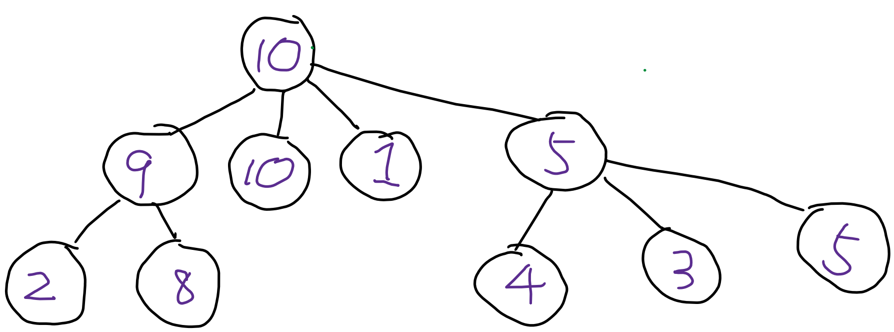
  Sort ($A[0 \dots n-1]$):
  1. pq = Priority Queue () //O(1)
  2. **for each** $item$ **in** A: //line 3 runs n times
  3. $\quad add(pq, item)$
  4. **for** i **in range** (n): //line 5and 6 run n times
  5. $\quad item = pop\_max(pq)$
  6. $\quad A[n-i-1] = item$ //O(1)

  If both line 3 and line 5 can be done with worst-case running time strictly less than $\Theta(\lg n)$, then we have a comparison-based sorting algorithm with worst-case time complexity strictly less than $\Theta(n \lg n)$, which is a contradiction to the above Theorem.

- Here, we introduce an implementation of Priority Queue using a *binary (maximum) heap.* In this implementation, both $add(item)$ and $pop\_max()$ has a worst-case time complexity $\Theta(\lg n)$.

<u>Heap</u>

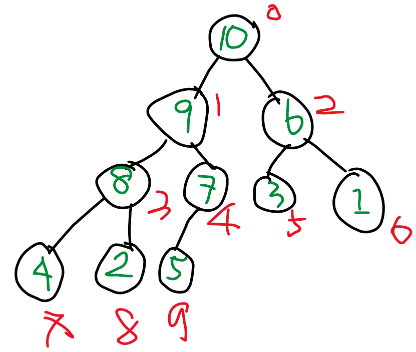
- A heap is a tree-shaped (or sometime forest-shaped) structure that satisfies "heap-property".
  - If the heap is a **max heap**, it satisfies the **max-heap-property**: subtree rooted at a node contains elements no larger than that contained at the node itself.
  - Similarly, If the heap is a **min heap**, it satisfies the **min-heap-property**: subtree rooted at a node contains elements no smaller than that contained at the node itself.
- In this class, without specification, we assume that all heaps we talk about are max heaps.



<u>Binary Heap</u>

- A **binary heap** is a heap stored on a **complete binary tree**.
  - "Binary" means that each node has at most two children.
  - "Complete" means only the lowest level of the tree can be not full, and all the nodes on the lowest level are on the left of this level.

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



- When the number of nodes of a binary heap is fixed, its shape is also fixed.
- From the above observation, we can see that it is possible to implement a binary heap using an ArrayList.

  - **__init__**$(self)$:

    $self.data = [\,]$

  - **max**$(self)$:

    return $self.data[0]$

  - **left**(i):

    return $2i + 1$

  - **right**(i):

    return $2i + 2$

  - **parent**(i):

    return $(i - 1)\ //\ 2$