Implement the PriorityQueue ADT

You can do these exercises in any order you wish. If you feel that implementing insert_ordered() and pop_back() or pop_front() in an array or linked list one more time will not be educational, then you can go straight to the heap implementation on the next page.
Also, if the heap implementation seems extremely complex then by all means start by finishing and understanding the binary tree and/or general tree exercises from previous classes, and take time getting acquainted with the PriorityQueue ADT through list implementations.
The heap exercises will be very helpful as a foundation for implementing complex tree structures but there is no immediate assignment where this implementation is needed, so just take your time to get these things right.

- ***The ADT:***
  - **add(value)**
    - Adds the value to the queue
  - **remove()**
    - Removes ***and returns*** the lowest value from the queue
- Use any method you like, including built in python functionality
  - Doesn't matter if you order while adding, or search while removing
    - Just get the ADT to return correctly
  - Using a built in python priority queue or a built in sort functionality is not really implementing it, so make sure you choose an implementation that is also good practice. It's still OK to do it first using built in methods, just to get it working.
  - What is the *time-complexity* of ***add()*** in your implementation?
  - What is the *time-complexity* of ***remove()*** in your implementation?
- Two recommended methods (implement both for practice):
  - Singly linked list
    - add() can be implemented like a insert_ordered() method
      - Floats each value recursively up the list
        - Recursive operation can return head or new node
    - remove() simply pop_front() (head_remove())
  - Array or python list
    - add() can be implemented like a insert_ordered() method
      - Floats each value down the list
        - Shifts each value one back while searching for location
    - remove() simple pop_back() (cheaper in array, as no shifting needed)
  - What is the time complexity of each operation?
- ***Now change the ADT so that it takes a priority and a value***
  - **add(value, priority)**
    - Adds the value to the queue
  - **remove()**
    - Removes ***and returns*** the *value* ***with the lowest priority*** from the queue
  - *This way you can store whatever class you like, with information and have the priority separate from the data.*

- ***Implement the PriorityQueue ADT using a <u>binary tree heap</u>***
    - Note that the implementation example in the book uses an array to implement the tree structure. In this way it is possible to use integer calculations to calculate the location of the parent or left/right child of each node. This is an efficient implementation, ***but we are more interested in*** getting students acquainted with ***programming with nodes***. *We therefore recommend using nodes and node references to implement the heap this time around*.
    - You can implement it like the teacher shows in the original video, using the reference ***last_node*** to keep track of which node was added last (or will be removed next). You can also use other ways to mark nodes so that you can traverse from the root directly to the correct location. These marks or values would still need to be bubbled up through the tree so it is uncertain they would be more efficient. ***Do what feels intuitive to you!***
    - ***Make sure to use subroutines (helper functions) to make your code clear and readable. The teacher's code in the video is a lot of jumble, so definitely take each idea and decide whether it would be easier to make sense of if you implement that idea as a separate operation and then call it with a clear and understandable name. This will make both understanding and maintaining your code much simpler.***
    - *What is the time complexity of each operation?*
- Implement ***heap-sort*** for any of your previous data structures
    - Use your *heap* implementation to process all the data
    - Simply add all the items to the heap and then remove them one by one and add them back to your original structure. (*What is the time-complexity of heap-sort?*)