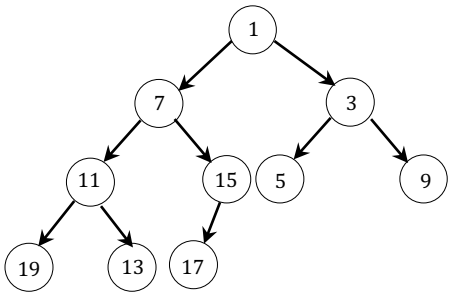
Homework #10

Submission instructions:
1. You should **not submit** this assignment.
2. Use the questions here for practicing priority queues.

Question 1:

Given the following minimum heap `h`:



We are executing the following sequence of operations (one after the other):

```
h.insert(6, None)
h.insert(8, None)
h.insert(0, None)
h.delete_min()
h.delete_min()
h.delete_min()
h.delete_min()
```

For each one of the operations above, draw the resulting heap **both** in its tree representation and in its array representation.

Note: In this question, we are focusing only on the structure of the heap and on the order that the priorities are set in it. We are less interested in the values associated to the priorities (as defined in the *priority queue ADT*).
Therefore, we associate `None` as the value for all keys.

Question 2:

a) Implement the *FIFO queue ADT* (`q.enqueue(elem)`, `q.dequeue()`, `q.first()`, `len(q)` and `q.is_empty()`), using only: a priority queue and one additional integer, as data members.

b) Professor Idle suggests the following solution to the previous section. Whenever an element is inserted into the queue, it is assigned a priority that is equal to the current size of the queue.
Does such a strategy result in *FIFO* semantics? Prove that it is so or provide a counterexample.

Question 3:

Add the following method to the `ArrayMinHeap` class:

```
def find_less_than_or_equal_to(self, k)
```

This method is given an integer `k`. When called, it creates and returns a list containing all the priorities in the heap that are less than or equal to `k`.

For example, if `h` is the heap you **started with** in question 1, the call:
`h.find_less_than_or_equal_to(11)`, could return:
`[1, 7, 3, 11, 5, 9]`.

Implementation requirements: Your method should run in time proportional to the size of the returned list, and it should **not** modify the heap.

Question 4:

Implement the following function:

```
def k_largest_elements(lst, k)
```

This function is given a list of integers `lst` and an additional integer `k`. When called, it will create and return a list containing the `k` largest elements in `lst`.

For example, the call:
`k_largest_elements([3, 9, 2, 7, 1, 7, 1, 3], 4)` could create and return the list `[7, 9, 7, 3]`.

Implementation requirements:

1. If `lst` contains *n* items, the run time of your function should be $\Theta(n \cdot log(k))$.
2. Your function can use $O(k)$ auxillary space (that is, you may use $O(k)$ space in addition to `lst`).