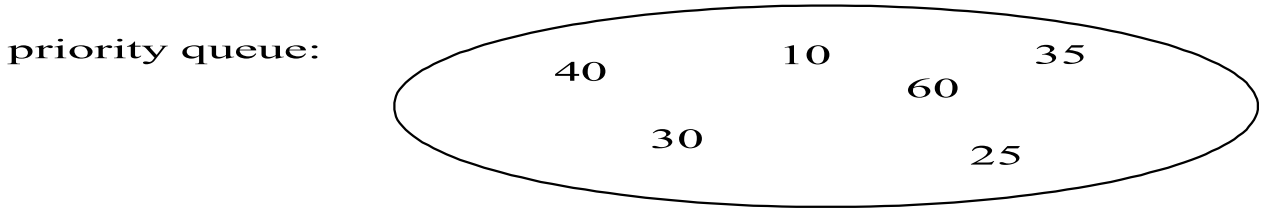
Objective: To understand priority queue implementations in Python including being able to determine the big-oh of each operation.

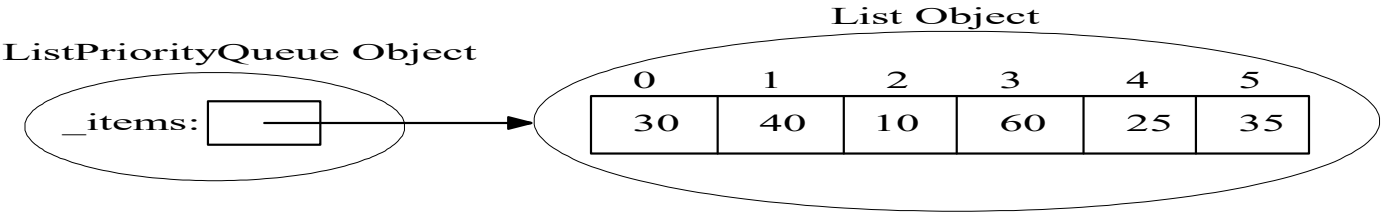
To start the lab: Download and unzip the lab4.zip file from eLearning.

Part A: Python list implementations of a Priority Queue

a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next? 10



b) The `ListPriorityQueue` implementation in `lab4/list_priority_queue.py` uses an **unorder Python list**.

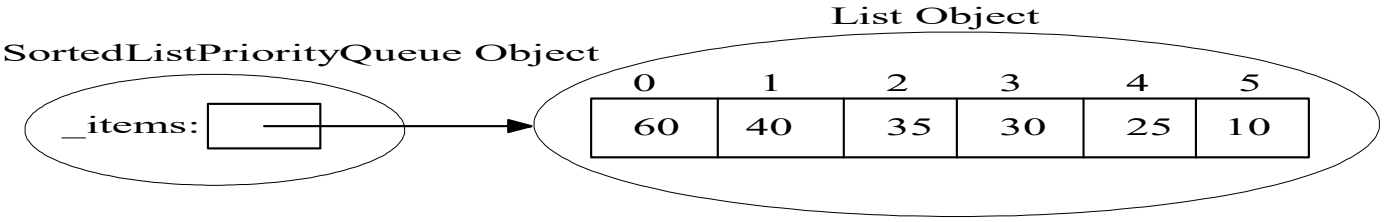


What would be the big-oh notation for each of the following methods: (justify your answer)

Ÿenqueue: O(1) because we aren't moving the items around at all, just adding one to the end

Ÿdequeue: O(n) because we have to go through all n items to find the item with the highest priority.

c) The `SortedListPriorityQueue` implementation in `lab4/sorted_list_priority_queue.py` uses a **Python list order by priorities** in decending order.

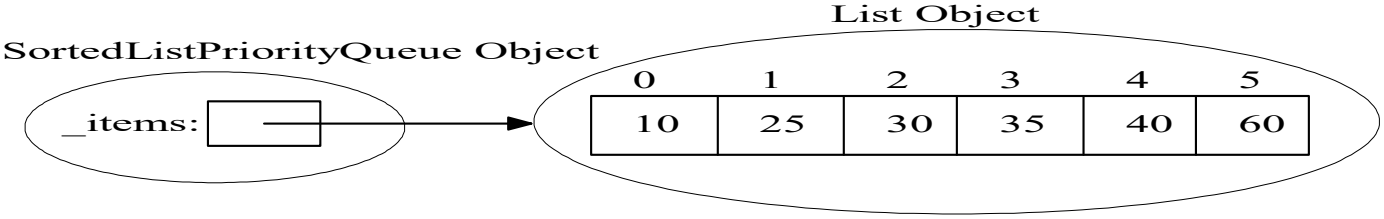


What would be the big-oh notation for each of the following methods: (justify your answer)

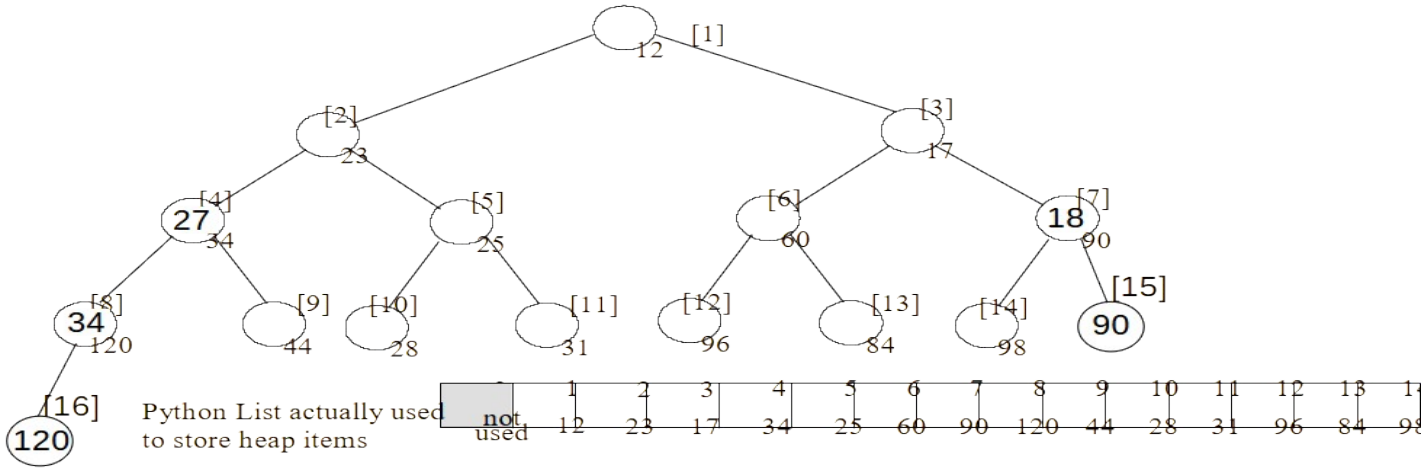
Ÿenqueue: O(n) because we are searching through n items to find where the new item should be

Ÿdequeue: O(1) because we are always popping from the end, which will always be constant.

d) Why would it be a bad idea to implement a priority queue using a **Python list order by priorities** in reverse (ascending) order? (HINT: What is the big-oh notations for enqueue and dequeue?) Both enqueue and dequeue would be O(n) because we are either searching through n items (enqueue) or moving n items (dequeue)



Part B: (Lecture 7 and) Section 6.6 discusses a very "non-intuitive", but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array is used to store a *complete binary tree* (a full tree with any additional leaves as far left as possible) with the items being arranges by *heap-order property*, i.e., each node is ≤ either of its children. An example of a *min* heap "viewed" an a complete binary tree would be:



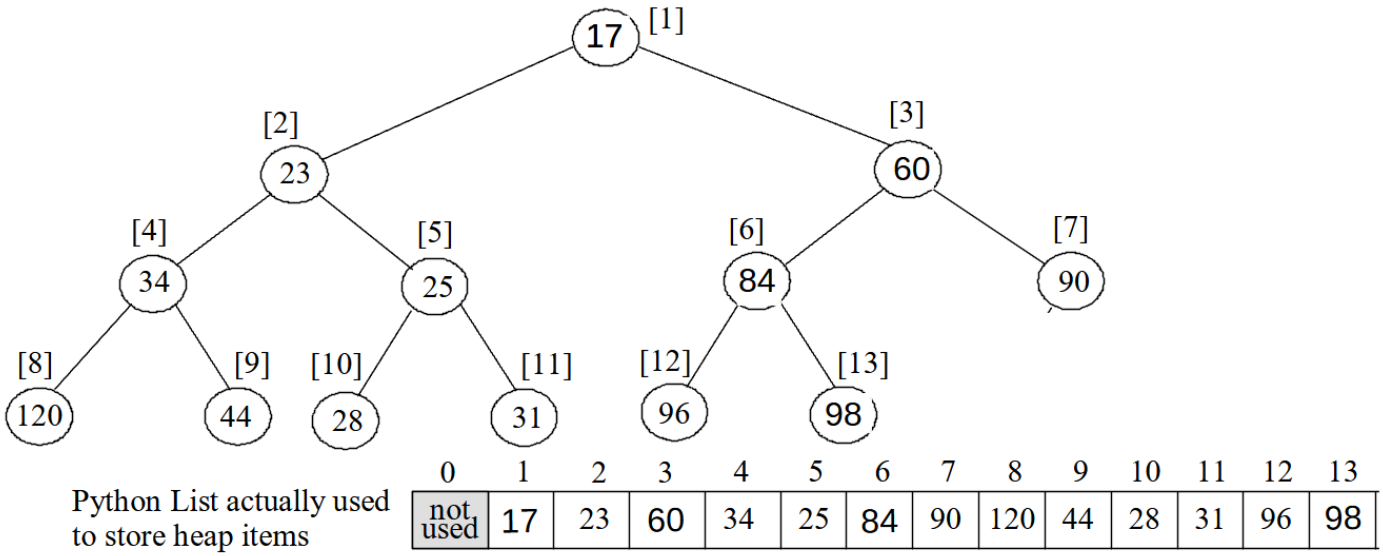
a) For the above heap, the list/array indexes are indicated in [ ]'s. For a node at index *i*, what is the index of:
Ÿits left child if it exists: 2*i
Ÿits right child if it exists: 2*i+1
Ÿits parent if it exists: i//2
Recall the General Idea of insert(newItem):
Ÿappend newItem to the end of the list (easy to do, but violates heap-order property)
Ÿrestore the heap-order property by repeatedly swapping the newItem with its parent until it *percolates up* to the correct spot

b) What would the above heap look like after inserting 18 and then 27? (show the changes on above tree)

c) What is the big-oh notation for inserting a new item in the heap?

Now let us consider the `delMin` operation that removes and returns the minimum item. Recall the General Idea of `delMin()`:

Ÿremember the minimum value so it can be returned later (easy to find - at index 1)
Ÿcopy the last item in the list to the root, delete it from the right end, decrement size
Ÿrestore the heap-order property by repeatedly swapping this item with its smallest child until it *percolates down* to the correct spot
Ÿreturn the minimum value



Python List actually used to store heap items

0	1	2	3	4	5	6	7	8	9	10	11	12	13
not used	17	23	60	34	25	84	90	120	44	28	31	96	98

d) What would the above heap look like after `delMin`? (show the changes on above tree)

Part C: (a) Run the `lab4/timePriorityQueues.py` program that enqueues 40,000 random integers followed by dequeueing all 40,000 integers from various priority queues discussed above. Complete the following timing table from the output of `timePriorityQueues.py`.

Priority Queue Implementation	Execution Time in Seconds	
	Enqueuing 40,000 Random ints	Dequeuing 40,000 ints
Unsorted Python list	0.008 seconds	27.412 seconds
Sorted Python list in descending order	30.034 seconds	0.003 seconds
"Reverse" sorted Python list in ascending order	30.057 seconds	35.682 seconds
Binary heap stored in a Python list	0.051 seconds	0.126 seconds

timings for (b) below

timings for (c) below

b) Why does it take more time to enqueue 40,000 items in the "unsorted" Python list version than dequeue 40,000 in the sorted Python list version?
In my case it didn't due to other factors and processes running.

c) Why does it take more time to dequeue 40,000 items in the heap version than enqueue 40,000 in the heap version?
Because dequeue is removing the minimum value, so some values have to shift positions every time.

d) Why is the heap implementation of the priority queue considered "better" than the other three?
It is more efficient

After you have answered all the questions, submit your answers on eLearning.

If you do not get done today, then submit it by next week's lab period.

If you have extra time, this would be a good chance to work on Homework #2!