Question 1.
Suppose you implement the functionality of a priority queue using a sorted array (e.g., from biggest to smallest). What is the worst-case running time of Insert and Extract-Min, respectively? (Assume that you have a large enough array to accommodate the Insertions that you face.)
If we happen to get an element bigger than the first element, we'd need to move all the other elements to the right. So that is worst case of O(n). Extract-Min will just involve a constant time lookup of the last element in the array.
O(n), O(1)

Question 2.
Suppose you implement the functionality of a priority queue using an unsorted array. What is the worst-case running time of Insert and Extract-Min, respectively? (Assume that you have a large enough array to accommodate the Insertions that you face.)
Insertion can be done anywhere, so if you insert it at the end you can do it in constant time. For Extract-Min you will need to visit all the elements in the array to find the smallest, so that is O(n).
O(1), O(n)

Question 3.
You are given a heap with n elements that supports Insert and Extract-Min. Which of the following tasks can you achieve in O(logn) time?
Our options are explained as follows:

1. Find the largest element stored in the heap - Could be done if we insert all the number by negating them first, but this isn't mentioned in the question.
2. Find the median of the elements stored in the heap - Could be done with a min-heap and a max-heap, and the max-heap can be built by inserting the negative of the numbers, but again, this is not mentioned in the question.
3. Find the fifth-smallest element stored in the heap - Finding the fifth largest element will take 5 calls, each taking O(logn)
4. None of these - False.

Find the fifth-smallest element stored in the heap

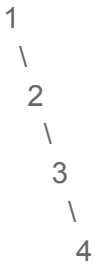
Question 4.
You are given a binary tree (via a pointer to its root) with n nodes. As in lecture, let size(x) denote the number of nodes in the subtree rooted at the node x. How much time is necessary and sufficient to compute size(x) for every node x of the tree?
For the lower bound, a linear number of nodes needs to be counted. For the upper bound, you can recursively compute the sizes of the right and left subtrees, then use the formula size(x) = 1 + size(y) + size(z) from the lecture.
n

Question 5.

Suppose we relax the third invariant of red-black trees to the property that there are no three reds in a row. That is, if a node and its parent are both red, then both of its children must be black. Call these relaxed red-black trees. Which of the following statements is not true?

Our options are as follows:

1. **Every red-black tree is also a relaxed red-black tree.** - Correct by definition
2. **The height of every relaxed red-black tree with n nodes is $O(\log n)$.** - Correct. In a red-black tree with n nodes, there is a root to NULL path with at most $\log_2(n+1)$ black nodes. Therefore, there are at most $2*\log_2(n+1)$ total nodes.
   We also know that a relaxed red-black tree may contain two red nodes for every black node. Therefore, the total number of nodes from a root to NULL path would be at most $3*\log_2(n+1)$. The height is therefore $O(\log n)$.
3. **There is a relaxed red-black tree that is not also a red-black tree.** - Correct. A regular red-black tree doesn't allow two red nodes in a row, but a relaxed one does. So, any relaxed red-black tree with two red nodes in a row is not a regular red-black tree.
4. **Every binary search tree can be turned into a relaxed red-black tree (via some coloring of the nodes as black or red).**
   False. Consider:

   In this case, no matter how we color the nodes, we cannot satisfy the property that all root to NULL paths must have the same number of black nodes. The left path has only one black node (the root itself), but on the right side there are at least two black nodes.

Every binary search tree can be turned into a relaxed red-black tree (via some coloring of the nodes as black or red).