CS21003 Algorithms-1
Tutorial 4
Solution Sketch

1. Prove or disprove:

(a) The minimum value of any max-heap must be present in a leaf.

→ Prove by contradiction. Assume that the minimum value is a node, which is not the leaf. Then, it has atleast one child, and that has a value smaller than the node. Contradiction.

(b) The second minimum value of any max-heap must be present in a leaf.

→ Disprove by counter-example: Take a heap with 2 nodes.

2. Let us add the facility to a priority queue that the priority of an item may change after insertion. Provide an algorithm *changePriority* that, given the index of an element in the supporting array and a new priority value, assigns the new priority value to the element, and reorganizes the array so that heap ordering is restored. Your algorithm should run in O(log n) time for a heap of n elements.

→ The priority may either be increased or decreased. If it is increased, we might have to traverse up to the root, while swapping with the parent. If it is decreased, we might have to traverse down to a leaf node (= calling Heapify at that node). In both cases, the complexity is O(log n).

3. Design an O(nlogk)-time algorithm to merge k sorted linked lists having a total of n items.

→ We first create a max-heap for these k linked lists, where the nodes in the heap point to the first elements of these linked lists.

for i=1 to n

Output the max value. Move the pointer to next in the root linked list. Call Heapify at the root.

Since heap is of size k, each iteration takes O(log k) time, thus a total complexity of O(nlogk).

4. Answer the following questions:

(a) How can you find the second and the third maximum elements of a max-heap in constant time?

(b) Generalize the result for the **k**-th maximum element, where **k** $\in$ N is a constant.

→ The largest element is root, the second largest will be one of the two children {x,y}. Let x be the larger among these. The third largest will be from {y,x-left,x-right}, where x-left and x-right correspond to the left and right children of x. Why? If it is any other node z, its parent will have a larger value than z. But z is the third largest, so the parent must be either first or second largest. Thus, either the root or x. Hence, the only possibilities are {y,x-left,x-right}.

You can use this argument for k-th maximum elements as well, where you have to select the max from k remaining elements only. Since k is a constant, and the complexity would be polynomial in k, you can find the element in constant time.

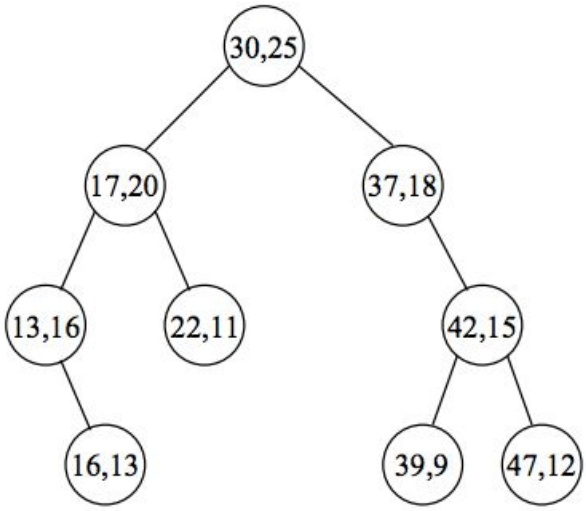
5. Propose an algorithm for printing 'k' largest elements in an array of 'n' elements in O (n + klogn) time. Can you now make it more efficient by doing it in O (n+klogk) time?

→ O (n + klogn) algo:

Create a heap out of n elements in O(n) time
Now, perform deleteMax operation k times → O(klogn)

O (n + klogk) algorithm: Idea is similar to Solution for Q4. At any time, the set of elements from which you have to select the next max (e.g., k elements for k-th max), should be in a heap structure. The nodes in the heap store the value (to select and delete the max element) as well as a pointer to the node in the original heap (to be able to find the two children and add to the smaller heap).

6. A treap T is a binary search tree with each node storing (in addition to a value) a priority. The priority of any node is not smaller than the priorities of its children. The root is the node with the highest priority. Unlike heaps, a treap is not forced to satisfy the heap-structure property. An example of a treap is given in the adjacent figure, where the pair (x, y) stored in a node indicates that x is the value of the node, and y is the priority of the node. The x values satisfy binary-search-tree ordering, and the y values satisfy heap ordering. An example of a treap is given below.



Design an O(h(T))-time algorithm to insert a value x with priority y in a treap. (Hint: Use rotations.) Insert (18,25).

→ First, insert x in the tree using the standard BST-insertion procedure. This insertion may violate heap ordering, that is, we may encounter a situation where a node u has a priority larger than the priority of its parent p. Depending upon whether u is the left or the right child of p, we make a right or left rotation making u the parent of p. The larger priority value moves by one level up the tree, and may again be larger than the priority of its new parent. This violation of heap ordering is repaired by another rotation. This process is repeated until heap ordering is restored, or the node with a large priority reaches the root of the treap.