Part A: Update the java file names to include your initials at the end, send .java file only.

1. Write in-place heapSort, which takes an array of data, convert it to max heap using bottom-up algorithm. The expected run time should be O(n), where n is the total number of data. BubbleDown method is provided. You may test it in this Sort.
2. Write complete code for a MinHeapPriorityQueue method removeThirdMin that efficiently removes and returns the third minimum element from the data structure. It shouldn't have more than 3 comparisons to find the third minimum element. Your solution can make use of the methods bubbleUp, bubbleDown and swapData but must not apply the constructor or either of the methods add/enqueue and removeMin/dequeue.
   For example, if the array contains the following elements:
   1, 2, 3, 4, 5, 6, 7
   It should be changed by removeThirdMin to
   1, 2, 6, 4, 5, 7

Extra Credit: Modify quickSort method, to use insertionSort if sub-problem is small. Check with size 4, 8, 16, which give the best runtime?

Part B

1. Show how an initially empty min heap looks like after inserting following elements in the given order. Draw the min heap and the array.

   5, 2, 7, 8, 6, 1, 7

2. Show how a bottom-up min heap construction looks like from the above given values. Draw the min heap and the array.

3. What is the best-case and worst-case of insertionSort? Best: $O(n)$ Worst: $O(n^2)$
4. What is the best-case and worst-case of mergeSort? Best: $O(n \log n)$ Worst: $O(n \log n)$
5. What is the best-case and worst-case of quickSort? Best: $O(n \log n)$ Worst: $O(n^2)$
6. What can we do to lower, if not eliminate completely, the chances of worst-case of quickSort?
   We can use randomized QuickSort which is implemented randomly picking a Pivot, with randomized Algorithim, the worst case time complexity would be $O(n \log n)$, Compares $O(n^2)$ we have before which is improved.

Extra Credit:

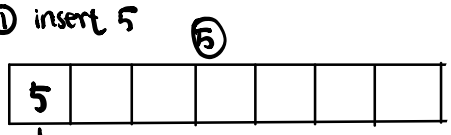
If we can access the data of a min-heap, how do we find the k-th minimum value efficiently? That means instead of O(log n), can we speed up to O(log k)?

Yes, finding k^{th} will means root to k-value.
so, we only need compare from elements from root to k, which will be 2^k-1, and the runtime can up to $O(\log k)$.

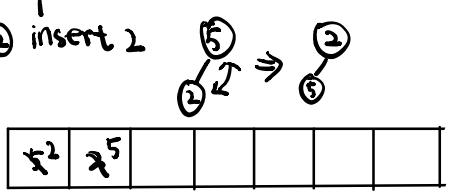
1). Min Heap

5, 2, 7, 8, 6, 1, 7

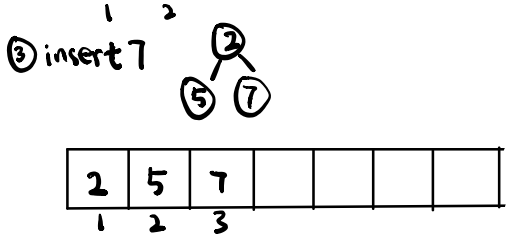
① insert 5



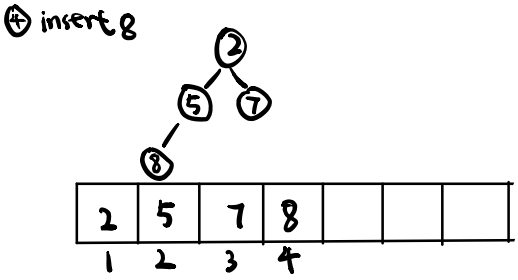
② insert 2



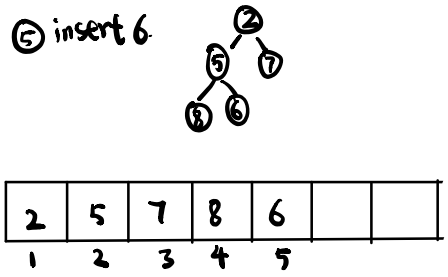
③ insert 7



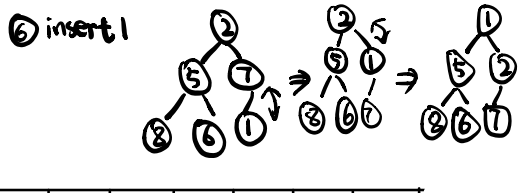
④ insert 8



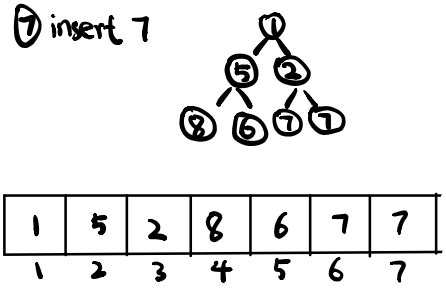
⑤ insert 6



⑥ insert 1



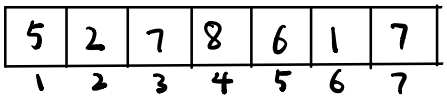
⑦ insert 7



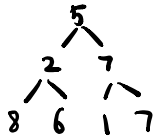
---

2). Bottom-up Min Heap

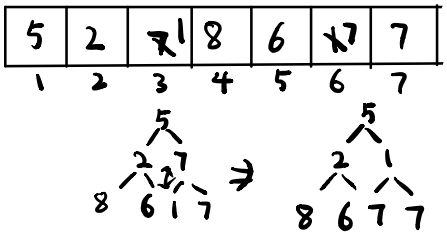
5, 2, 7, 8, 6, 1, 7



①



②



③