The objective of this lab is to explore heaps and their application in the context of heap sort.

Please take the following steps in this lab. Download the file `Lab6.java`. This file contains partial implementations of a priority queue using a heap.

1. Take a look at interface `PriorityQueueADT` which contains basic operations of a priority queue. This interface is implemented using class `Heap` which maintains a set of integer keys in an array (complete tree) named `heap`. The implementation of class `Heap` is not complete. Write the missing methods from `PriorityQueueADT`. These methods are `isEmpty()`, `getSize()`, and `getMax()`. Each of these methods take 1 or 2 lines of code. In the space below, write down what your code returns in each of these methods.

   **Answer:**
   - `isEmpty()` returns whether `size == 0`.
   - `getSize()` returns `size`.
   - `getMax()` returns `heap[0]` (the root).

2. Class `Heap` has two constructors. While one of them initiates the heap as an empty heap, the other constructor receives an input array and `heapifies` it. More precisely, in the method `heapify`, all items are bubbled down from the bottom to the top of the tree. Right now, the `reheapDown` is called for all indices in the array. Do we need to call `reheapDown` for all indices? if not, update the code to make it more efficient. Write a sentence in the space below about your update.

   **Answer:** For a heap of size `n`, the last `n/2` items in the array are all leaves (and do not need to be bubbled down). So, we can replace "size-1" with "size/2" in the loop.

2. Recall that inserting a key `newKey` involves placing the key at the end of the array and bubbling it up. There is a bug in the `reheapUp` method. Indicate what the issue is, write a sentence about it in the space below, and fix it in your code.

   **Answer:** The recursive function’s base case is missing. We should add "index > 0" in the `if` condition.
3. Recall that `extractMax` involves returning the root (if the tree is not empty), and fixing the tree by replacing the root with the last element in the array. Complete the implementation of the `extractMax` operation.

4. In class Lab6, there is a method `heapSort1`. This method inserts keys in an input array `a` one by one into an initially empty heap with maximum size `maxSize`. After inserting all keys, the method should repeatedly call `extractMax` to get the items in the sorted order. Complete the implementation of `heapSort1`. The goal is to have the array `a` sorted in non-decreasing order (the extracted items should be copied reversely into the same input array `a`).
   The `main` function includes a commented block which tests `heapSort1`. Uncomment that block to test your code. Show your working code to the TA.

5. In class Lab6, there is a method `heapSort2`. This method sorts the input array by forming the heap from the array (rather than repeatedly calling `insert`). In order to do this, the constructor that calls `heapify` should be called. The second stage of the algorithm is similar to `heapSort1` and involves repeatedly calling `extractMax`. As before, the goal is to have `a` sorted in non-decreasing order. Complete the implementation of `heapSort2`.
   The `main` function includes a commented block which tests `heapSort2`. Uncomment that block to test your code. Show your working code to the TA.

6. In the main method, there is a commented block that creates a random array of large size and calls both `heapSort1` and `heapSort2` to sort this array. Uncomment this block and run the two algorithms for a few large values of `n` a few times. Which method runs faster? why is that? write a brief justification in the space below and show your numbers to the TA.

   **Answer:** Both algorithms form a heap on the input numbers and repeatedly call `extract max` to generate sorted output. In forming heaps, however, `heapSort1` calls `insert` `n` times and hence runs in $O(n \log n)$ time while `heapSort2` runs a linear-time heapify. Hence, `heapSort2` is just slightly faster. Here is a sample output: