Name: _____

Please submit this homework by following the homework submission instructions on the class Web site. Reminder: You are welcome, and encouraged, to use any resources (e.g., Web sites) to help you with your work. However, *all such help must be clearly noted* in your submissions. Further, no matter what you use, *you must be able to explain* how and why it works. Refer to the class policy for details, and ask for clarifications if you are unsure if something is allowed. Questions marked with ⋆ are optional and may be answered for extra credit.

1. (1 pt.) Write your name in the space provided above.

2. (1 pt.) Package and submit your solutions to the programming questions as in previous assignments. After uploading your jar file to the FTP server, complete the following:
   File name: _____ Size, in bytes: _____

3. (5 pts.) A *priority queue* is a data structure that efficiently implements three operations:[1]

   - *insert*, which adds an element to the queue;
   - *findMin*, which returns the smallest element in the queue; and
   - *deleteMin*, which returns the smallest element in the queue and removes it from the queue.

   Describe an algorithm for sorting a list of elements efficiently by using a priority queue as the primary data structure. Suppose we have a priority queue implementation in which the insert, findMin, and deleteMin operations require time $t_i(q)$, $t_f(q)$, and $t_d(q)$, respectively, where q is the number of elements in the queue. Quantify the running time of your sorting algorithm on an input of n elements as accurately as possible.

[1] Mark Allen Weiss, *Data Structures and Problem Solving Using Java*, 3rd edition (Addison-Wesley, 2006), Section 6.9, p. 239.

4. (3 pts.) Explain how one may use a priority queue as the primary data structure to implement (1) a stack and (2) a FIFO (first-in, first-out) queue.[2]

5. Read pages 746–758 and 761–764 (Section 6.5) of the textbook and answer the following questions:

   (a) (5 pts.) The textbook's description of binary heaps[3] uses a dummy first item in the first array location. Using this scheme to store n nodes of a binary heap in

[2] *Idem*, Section 6.6.1, p. 225; Section 6.6.3, p. 227.
[3] *Idem*, p. 747.

an array A yields

$$\begin{aligned}
\text{parent}(A[i]) &= A[\lfloor i/2 \rfloor] \\
\text{left_child}(A[i]) &= \begin{cases} A[2i] & \text{if } 2i \leq n \\ \perp & \text{otherwise} \end{cases} \\
\text{right_child}(A[i]) &= \begin{cases} A[2i+1] & \text{if } 2i+1 \leq n \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

for $i \in [1, n]$, where $\perp$ denotes null. Provide similar expressions for parent, left-child, and right-child when no dummy item is used.

(b) (5 pts.) The textbook's method for implementing the *deleteMin* operation percolates a hole from a node to its child with the smaller key.[4] Would the heap-order property be satisfied if the child with the larger key were chosen instead? If so, explain why. If not, provide a counterexample.

[4] *Idem*, p. 754.

(c) (5 pts.) Explain, as precisely as possible, how the behavior of the textbook's `PriorityQueue` class[5] changes if the types of the formal parameters of the constructors on lines 11 and 13 are changed to `Comparator<AnyType>` and `Collection<AnyType>`, respectively.

(d) (5 pts.) The implementation of `buildHeap` and its accompanying description[6] suggest that the highest-numbered (by array index) nonleaf node is at position `currentSize/2`. Prove this claim.

[5] *Idem*, Figure 21.4, p. 750.
[6] *Idem*, p. 754.

6. (10 pts.) $\star$ Describe an algorithm for generating random trees (rooted, labeled, unordered) on a given set of nodes. The input is a positive integer n. The output is a random tree with n nodes labeled $1, 2, \ldots, n$. An important requirement is that the algorithm be unbiased: If $T(n)$ denotes the set of distinct trees on the n nodes $1, 2, \ldots, n$ and t is an arbitrary tree in $T(n)$, then the probability that the algorithm produces t as the output must be $1/|T(n)|$. Justify the correctness of your algorithm and quantify its running time.

7. (20 pts.) Using the standard `java.util.PriorityQueue` class, implement the algorithm you describe for Question 3 by providing a Java class `PQSorter` that implements the following interface:

```
public interface Sorter {
/**
     @param d the data to be sorted, as a list of Integers
     @return a new list containing the elements of d in sorted order.
*/
     public List<Integer> sort(List<Integer> d);
}
```

8. (40 pts.) Implement the `percDown` method used by the textbook's heapsort implementation[7] and and provide a Java class `HeapSorter` that implements the `Sorter` interface of Question 7.

9. (10 pts.) Study the performance of your two implementations of the `Sorter` interface by measuring the running times of their `sort` methods. You may use various test cases but, at a minimum, you should measure running times for inputs of varying sizes ($10 \ldots 10^6$ elements) and varying orders (randomly ordered, sorted, and sorted in reverse order). Refer to the documentation of `System.currentTimeMillis` for a simple method to measure running times.

   Summarize your results in tabular form in the README file of your submission. The file should also describe the exact steps that should be followed to reproduce these results using your submitted code.

10. (10 pts.) $\star$ Implement the algorithm you provide for Question 6 and submit your code with the rest of your assignment as usual. Perform experiments to quantify the running time of your implementation, varying any parameters used by your algorithm and implementation over as wide a range as possible. Also perform experiments to test your algorithm for bias in the output. As above, summarize your results in tabular form in the README file of your submission. The file should also describe the exact steps that should be followed to reproduce these results using your submitted code.

[7] *Idem*, pp. 763–764.