Advanced Data Structures
Assignment #3

There are 2 main goals of this assignment: first, to practice using priority queues and second, to compare the running times of different implementations of priority queues. As a result, you may notice that while complexity is a very important consideration, it is not the *only* consideration.

You are to implement a priority queue in 3 different ways: as an ordered linked list, as an array version of a min-heap, and as a binary tree version of a min-heap (i.e. nodes with pointers to children, *not* stored in an array). You can choose to either have all 3 implementations within a single program, or have 3 separate programs. Your textbook has a fairly complete implementation of the array version of a heap, and you are allowed to modify this according to your needs. Also, implementing a priority queue as an ordered linked list is a fairly simple task. Therefore, you should realize that you will need to concentrate most of your efforts on the binary tree version. In this version, you should (if at all possible) *avoid* the use of a "parent" pointer within each node; you should also try to think of the most efficient way to find the "last" node in the heap.

For each of the above implementations, each entry in the priority queue stores a single integer in the range [1..1000], which is its priority. The following operations must be supported: `insert`, `deleteMin`, and `traverse`. For the array version of the heap, the implementation must also support the `buildHeap` operation. In the case of the linked list implementation, `traverse` should print all items in the priority queue from front to rear. In the case of the heap implementations, `traverse` should print all items in the heap using a preorder traversal.

Part A (80 marks): This part simply tests that each of your implementations is working correctly. You are to read input from a file with the following format:

An integer value `n`, indicating that you will be using a priority queue with `n` elements
`n` integer values, which are the priorities of the items to be stored in the priority queue.

(i) For **each** of the three implementations, insert each of the items into the priority queue in the order given, and then print to a file the output obtained from the following sequence:
`traverse` (immediately after all items have been inserted)
`n` consecutive calls to `deleteMin` (in each case print the priority of the item removed)

(ii) For the array heap implementation **only**, **also** do the following (separately from part (i) above): insert each of the items into the array in the order given, and then use `buildHeap` to turn this array into a heap. Print to a file the output obtained from the following sequence:
`traverse` (prior to `buildHeap`, and after each iteration in `buildHeap`)
`n` consecutive calls to `deleteMin` (in each case print the priority of the item removed)

For each of the above, you must submit the results obtained from using the input file `assn3in.txt` available on Sakai. Your program *will* be tested using other input files; therefore you must make sure that your program is thoroughly tested and will work for a variety of input meeting the specifications.

Part B (40 marks): You are to compare the performance of the implementations for `n` = 50, 100, 1000, 5000 and 10000. For **each** of the three implementations, measure the **total** execution time required to first insert *the same* random sequence of `n` values in the range [1..1000] into the priority queue and then call `deleteMin` `n` times. For the array heap implementation **only**, **also** measure (separately) the **total** execution time required to insert each of the items into the heap array in the order given, use `buildHeap` to turn this array into a heap, and then call `deleteMin` `n` times.

For part B, no output needs to be printed other than the execution times for each run. Submit a table that presents the results for each implementation and each value of `n`. Also, write a commentary (two or more paragraphs) commenting on the results and what you think they mean. You should comment on complexity, representation, and the impact of `buildHeap`.

To ensure you are using the same random sequence, either generate it ahead of time and store it in a file to be read, or generate it at the start of the program and store it in memory, then use it for each implementation.

Note: you can use `System.nanoTime()` to obtain the current "time" of the system, and thereby use the difference between two such calls to measure elapsed time.