Ex. After Build-Heap(4):

20 10 15 8 3 12

12 10 15 8 3 20

15 10 12 8 3 20

3 10 12 8 15 20

12 10 3 8 15 20

8 10 3 12 15 20

10 8 3 12 15 20

3 8 10 12 15 20
The call to build-heap takes $\Theta(n)$ time in worst case. Each of the $n-1$ calls to heapify takes $\Theta(\log n)$ time in worst case.

Therefore, the worst case run time of Heapsort is

$$T(n) = \Theta(n) + (n-1) \Theta(\log n) = \Theta(n \log n).$$

Note: This is as good as Mergesort (asymptotically speaking), but Heapsort sorts in-place, while Mergesort requires extra memory.
6.5 Priority Queues

A Priority Queue is an ADT which maintains a finite set $S$ of elements, each with an associated value called a key.

$$x = (\ldots, \text{key}) \in S$$

$$S = \{ \ldots, x, \ldots \}$$

The attribute $\text{key}[x]$ defines the "priority" of $x$ in $S$.

A Priority Queue supports the following operations:

- **Insert** $(S, x)$: Insert new element $x$ into $S$.
- **Maximum** $(S)$: Return element with largest key.
- **ExtractMax** $(S)$: Return and delete max element.
- **IncreaseKey** $(S, x, k)$: Change $\text{key}[x]$ to $k \geq \text{key}[x]$.
More generally the above ADT would be called a max priority queue. The dual notion of a min priority queue would have operations Insert(S, x), Minimum(S), ExtractMin(S), DecreaseKey(S, k).

Note: From our point of view regarding ADTs, ExtractMax is a hybrid operation. We could define a pure manipulation procedure called DeleteMax(S), which just deletes the maximum element. ExtractMax(S) is then the sequence Maximum(S), DeleteMax(S).

A max (resp. min) priority queue can be implemented efficiently as a max (resp. min) heap.

\[
\text{Heap Maximum}(A) \\
1.) \text{ If head-size}[A] < 1 \\
2.) \text{ error "head underflow"} \\
3.) \text{ return } A[1] \\
\]

Run time \( \Theta(1) \).
Heap Extract Max (A)
1. if heap-size [A] < 1
2. error 'heap underflow'
3. max = A[1]
5. heap-size [A] = (heap-size [A] - 1)
6. Heapify (A, 1)
7. return max

To obtain the Pure Manipulation Process, Heap Delete Max (A), just leave out lines (3) and (7).

The run time is $O(\log n)$ since all lines take constant time except line 6, which takes time $O(\log n)$.

Heap Increase Key (A, i, k)
1. if $k \geq A[i]$
2. $A[i] \leftarrow k$
3. while $i > 1$ and $A[parent(i)] < A[i]$
4. $A[i] \leftrightarrow A[parent(i)]$
5. $i \leftarrow Parent(i)$
HeapIncreaseKey \((A, i, k)\) increases the priority of \(A[i]\) to \(k\), then traverses an upward path from parent to parent, repairing the heap property along the way. Since this path can have length at most \(\log n\), where \(n = \text{heap-size}[A]\), the run time of HeapIncreaseKey \((A, i, k)\) is, in worst case, \(\Theta(\log n)\).

\[
\text{HeapInsert}(A, k) \\
1.) \text{head-size}[A] \leftarrow \text{head-size}[A] + 1 \\
2.) A[\text{head-size}[A]] \leftarrow -\infty \\
3.) \text{HeapIncreaseKey}(A, \text{head-size}[A], k)
\]

Run Time: \(\Theta(\log n)\)

Exercise: Implement a Priority Queue in both C and Java. Add appropriate constructors, destructors as appropriate. Also add isFull and isEmpty as well.

Remark:
A heap is a data structure while a priority queue is an ADT. Can a priority queue be implemented in other ways?