Quicker make_heap than looping over insert, which is all the text would say to do:

```
void make_heap (with arguments 0 and firstFree) {
    for (int j = firstFree - 1, j >= 0, j--) {
        fixDown(table.at(j));
    }
}
```

Fact (not in text! ☹) (not even in Ch. 10 Heapsort) This way of making an arbitrary vector into a heap runs in $O(n)$ time, more precisely $\Theta(n)$ time, with a pretty small constant! Hence better than:

```
void fixPQ(priority_queue& pq) {
    priority_queue other;
    while (! pq.empty()) {
        other.insert(pq.top());
        pq.pop();
    }
    // Still need to swap or copy other back into pq!
}  // Same dilemma as on Assignment 5, problem (2).
// Text sketches an in-place Build-Heap on pp 600-601, but still Θ(n log n)
```

NOT literal code - see next page

Takes $\Theta(n \log n)$ time

Conclusion: STL <algorithm>'s make_heap provides an efficient way to "re-heapify" that works also when many priorities have changed. "Dynamic."

It also gives the speediest general method for compiling "Top K" lists from initially-unsorted data:

① Put or keep data in a vector, any order.
② Call make_heap.
③ Pop (and save) K elements
④ Insert them back into heap, if needed

STL <algorithm> also has pop_heap and push_heap, which work with the same iterator args.

When n is the overall # of elements, the time is

$\Theta(n)$ for ① & ②
$+ \Theta(K \log n)$ for ③ & ④.

When K is relatively small – formally $K = O(n/\log n)$, this is $\Theta(n)$, and handily beats the $\Theta(n \log n)$ you'd get if you'd fully sorted all the data, then read off top K.

Importantly, STL's make_heap has a third parameter "Comp comp," which can take any function object that does a "bool greaterThan" style comparison. This also is templated, but you can just call it! Whereas literal code for "fixPQ" would need a template header. End