Heap operations

* insert():

concerns: preserve complete tree structure, preserve heap ordering

First step: add the value to the end of the list used to store the heap (conceptually, as the "next" node in the complete tree structure).

Second step: fix the heap order, by "percolating up" ("bubbling up") the new value: repeatedly swap the value with its parent, as long as the value is smaller than its parent. This stops when (1) the parent of the new value is smaller, or (2) the new value reaches the top.

Running time? O(log n).

* extract_min():

\[
\begin{array}{ccccccccc}
2 & 7 & 5 & 8 & 10 & 6 & 9 & 12 & 15 & 3 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\]
First step: take the last element and move it into the root position, to guarantee we are left with a complete tree structure.

Second step: "percolate down" ("bubble down") the value at the root until it is smaller than its children (or becomes a leaf).

```
<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```

Runtime? $O(\log n)$

Relationship to sorting...

"Heap Sort":
* Start with list $L$.
* First, put every value from $L$ into a heap.
* Next, take the values out of the heap (with $\text{extract\_min}$) and put them back into $L$, one by one, in order.
* Runtime? $O(n \log n)$ — twice: once to create the heap, then to sort the list.