Binary Search Trees: running time.

* What is the running time of each operation?
  (Worst-case, in big-Oh terms, as a function of n = size of the BST.)
* \( O(\log n) \)
* \( O(n) \)
* Actually, running time is \( O(\text{height}) \).

* What is height as a function of n?
- Worst-case:
- Best-case:

\[
\frac{n \text{ nodes, min height?}}{\text{flip it around:}} \quad \text{height } h \equiv \max n \?
\]

\[
\begin{align*}
  n &\le 2^{h+1} \\
  n+1 &\le 2^{h+1} \\
  \log_2(n+1) &\le h+1
\end{align*}
\]

A BST has height roughly \( \log n \) exactly when it is "balanced": for every node, height(left subtree) and height(right subtree) differ by at most 1.

* In practice, we need a way to keep BSTs balanced...
  AVL Trees, Red-black trees, 2-3-4 trees.

Priority Queues!
Conceptually, similar to a queue but where elements are have a "priority" that specifies how important each element is. When removing elements, those that are more important get removed first (elements with the same priority are processed in queue order: first-in, first-out).

What is a "priority", exactly? Any type of value that can be ordered. (For us, to keep things simple, we'll use integers.)

Question: which is more important? Priority 1, or priority 10?

"Min" priority queues treat smaller values as more important (priority 1 is more important than priority 10).
"Max" priority queues do it the other way around.

Operations (for min priority queue):
- is_empty(): return True iff the queue is empty
- insert(item, priority): add item with priority
  (some texts don't separate the two: they assume each item has a "priority" attribute)
- get_min(): return the item with smallest priority, without changing the queue
- extract_min(): remove and return the item with smallest priority

Data structures? Efficiency?

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>get_min</th>
<th>extract_min</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtin list, unsorted</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$ in addition to finding min</td>
</tr>
<tr>
<td>builtin list, sorted by priority</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(\text{height})$</td>
<td>$O(\text{height})$, $O(1)$ in addition to finding min</td>
<td>$O(\text{log}_2 n)$ if BST is balanced</td>
</tr>
</tbody>
</table>
Issues:
* BSTs don't handle duplicate values (really, they do, but we haven't seen how).
  - 1: store every item with same priority in a small queue, and use the BST to store these queues, along with their priority.
  - 2: add extra "time stamp" along with the priority to tell apart items with the same priority level.

Heap Data Structure:

* (1) conceptually, a "heap" is a "complete" binary tree:

- every level is full, except perhaps the last one
- on the last level, every node is as far left as possible

* consequence: complete binary trees always have height O(log n)
* (2) values are stored in "heap order" (min-heap or max-heap):
  - for a min-heap: each node's values is <= its children's values (top-to-bottom ordering instead of left-to-right like in a BST)
* property: the root of a min-heap always contains the smallest value

- second-smallest priority value? child of the root (or further below in case of repeated priorities)

Before we discuss how to implement insert and extract_min, a trick for storing heaps...

- node at index i:
  - left child's index? \(2i + 1\)
  - right child's index? \(2i + 2\)
  - parent's index? \((i - 1) // 2\)

```
5 10 7 20 15 7 20 21 32
0 1 2 3 4 5 6 7 8
```