Project: full handout posted later today.
Test 2: returned during this week's labs.

Binary Search Tree performance/efficiency/complexity/running time.
* How fast does this code run?
* Assume a BST with n nodes.
* Code executes a constant amount of work for each node on one path from the root down to a leaf.
* As a function of n? In the worst-case?
  O(n)?
  O(log n)? This is what we _hope_.

Problem: BSTs can be unbalanced: left subtrees and right subtrees don't have to contain the same number of nodes...
In the worst-case: every operation is O(n)...

Needed: a way to keep BSTs balanced -- AVL trees, red-black trees, 2-3-4 trees.
For a balanced tree (one where for each node, height of left subtree is roughly equal to height of right subtree), overall height is always
So for balanced trees, worst-case running time is $O(\log n)$.

**ADT: Priority Queue**

* A sequence of elements, where each element has a "priority".
* Elements are removed in order of priority, breaking ties in standard "queue order" (first-in, first-out).
* Question: is priority 1 more important than priority 10?
  - "Min" priority queue: smaller priority values are more important
  - "Max" priority queue: larger ...

* Standard operations for a min priority queue:
  - `is_empty()`: return True iff the priority queue is empty
  - `insert(item, priority)`: some descriptions have just item and assume item has a "priority" attribute: insert item with priority into the queue
  - `get_min()`: return the first item with smallest priority, without changing the queue
  - `extract_min()`: remove the item with smallest priority and return it

* Question: how do we implement this?
  - Data structures: Queue, linked list, dictionary, BST, tree, list, tuple
    Pros and cons of each? Think of efficiency!
<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>get_min</th>
<th>extract_min</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>(builtin, unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted list</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(reverse order of priority)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>(O(\text{height}))</td>
<td>(O(\text{height}))</td>
<td>(O(\text{height}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

worst-case running times for each operation

Heap data structure:
* binary tree, partially ordered "top to bottom"
* more precisely, a _complete_ binary tree structure

- every level (except possibly the last) is full
- on the bottom level, every leaf is as far "left" as possible

consequence:
height of a complete binary tree is always \(O(\log n)\)

* "heap ordering": min-heaps and max-heaps are possible
  in a min-heap: every node's value is \(\leq\) its children's values
- In a min-heap, where is the smallest value? At the root.
- In a min-heap, where is the second-smallest value? At depth 1; meaning a child of the root.

Storing heaps...
- Standard trick to store a heap without using a linked structure...

Consider the index of each node according to a traversal by level (top to bottom and left to right)
* Property: a node with index i always has children with indices $2i + 1$ (left) and $2i + 2$ (right); conversely, a node with index i has its parent at index $(i - 1)//2$. 