Exercise 1. For any implementation of the set ADT, show how to modify/augment it so that it can return its size (number of elements) in O(1) in the worst case.

We initialize the set ADT with a parameter `size`, and increment it whenever we successfully insert a new value, and decrement it whenever we successfully delete a new value.

Exercise 2. Show that the BST property, "For each node x, the value at x is greater than the value at every left descendant of x, and less than the value at every right descendant of x", is equivalent to saying that "the tree's inorder traversal is strictly increasing".

We show that the BST property holds if and only if the tree's inorder traversal is strictly increasing.

[=>: If BST property, then inorder traversal strictly increases]

We prove by induction on the number of nodes, k.

[Base Case]

If the tree has 0 or 1 nodes, the inorder traversal outputs an empty or single-element sequence, which is strictly increasing.

[Inductive Step]

We assume that any BST with fewer than n nodes has a strictly increasing inorder traversal.

Consider a BST T with root r, left subtree L, and right subtree R.

Then, by the BST property, every value in L is < r, and every value in R is > r. By the inductive hypothesis, inorder(L) is strictly increasing, and inorder(R) is strictly increasing.

The inorder traversal goes as:

$$\text{inorder}(L), r, \text{inorder}(R)$$

Since all values in L < r < R, the inorder traversal for the BST T is strictly increasing.

[<=: "If inorder traversal strictly increases, then BST property holds"]

We prove by induction on the number of nodes, k.

[Base Case]
For 0 or 1 nodes, the BST property vacuously holds.

[Inductive Step]

We assume the inductive hypothesis: "Any tree with nodes fewer than k that has a strictly increasing inorder traversal satisfies the BST property."

We consider a tree T with root r, left subtree L, and right subtree R, whose inorder traversal is strictly increasing.

Because the sequence is strictly increasing, every value in L appears before r, and every value in R appears after r, implying that all values in L < r and all values in R > r, respectively.

By the inductive hypothesis, L and R themselves satisfy the BST property, therefore, the entire tree must satisfy the BST property.

Therefore, a binary tree satisfies the BST property if and only if its inorder traversal produces a strictly increasing sequence.

Exercise 3. Explain how to implement delete_leftmost in $\Theta(h)$ worst-case time. Note that the function returns two things: the leftmost value, and the updated tree.

We remove the smallest value in the subtree and return the smallest value and the updated subtree with the node removed.

```
def delete_leftmost(node):
    if node.l is None:
        return (node.value, node.r)

(value, node.l) = delete_leftmost(node)
return (value, node)
```

Exercise 4. Explain how to implement next_larger in $\Theta(h)$ worst-case time.

To implement next_larger(v) in a BST, we find the smallest value strictly greater than v by traversing from the root while keeping track of the best candidate successor.

Algorithm:

We initialize succ = None.
Starting at the root,

- If node.value > v, then record it as a candidate (succ = node.value) and move to the left child to look for a smaller valid value.
- Otherwise, (node.value <= v), move to the right child since any successor must be larger.

When traversal ends, return succ (or None if no larger element exists).

Correctness:
Whenever we move left after finding a value > v, we search for a smaller valid successor. Moving right discards values that cannot be larger than v. Thus, the smallest value greater than v is found.

Running time:
The procedure follows a single root-to-leaf path, so it runs in Θ(h) worst-case time, with h as the height of the tree.

Exercise 5. Prove that a tree is basically complete if and only if its height is floor(lg n).

A binary tree of height h can contain at most $2^{h+1} - 1$ nodes. If all levels, except possibly the last, are full, then the tree has at least 2^h nodes. Thus, for a basically complete tree:

$$2^h <= n <= 2^{h+1} - 1$$

[=> "If the tree is basically complete, then h = floor(lg n)"]

Taking log base 2 of both sides from the equation above:

$$h <= \lg n < h + 1$$

Therefore,

$$\text{floor}(\lg n) = h$$

[<= "If h = floor(lg n), then the tree is basically complete"]
From h = floor(lg n), we have

$$h <= \lg n < h + 1$$
$$2^h <= n < 2^{h+1}$$

This implies that all levels from 0 through h - 1 must be completely full, containing 2^h - 1 nodes. The remaining nodes, if any, lie on level h. Hence, the tree is filled level-by-level except possibly the last level. Therefore, the tree is basically complete.

Thus, a binary tree with n nodes is basically complete if and only if h = floor(log n).

```
(l, m, r) = split(x.l, v)
x.l = r
return (l, m, x)
```

Exercise 6. Assuming l and r are treaps, why does the new x in the above pseudocode result in a treap? In particular, why does attaching r to the left child of x maintain the BST and max heap properties?

It retains the BST property because all values in r' are < x.value since they came from the left subtree of x, and all values in x.r remain > x.value. Thus, the inorder ordering is preserved, and so is the BST property.

It also retains the max-heap property because before the split, x satisfied the heap property with its children. The recursive split does not change priorities. Since r' was originally before x in its left subtree, all nodes in r' have priority <= to that of x. Therefore, attaching r' under x preserves the heap property.

Hence, the resulting tree rooted at x remains a valid treap.

Exercise 7. Show that the running time of split is O(h), where h is the height of the treap.

The split operation divides a treap into two treap based on a value v by recursively descending from the root.

At each step, we compare v with the current node's value. Depending on the comparison, we recurse into either the left or right subtree, not both. After the recursive call returns, we perform constant-time pointer updates to reconnect subtrees.

Thus, each recursive step moves one level down the tree, following a single root-to-leaf path.

Since the longest such path has length h, the height of the treap, the recursion performs at most h steps.

Exercise 7. Show that the running time of merge is O(h), where h is the height of the treap.

The merge operation combines two treaps L and R where every key in L is less than every key in R.

It compares the priorities of the roots. The root with the higher priority becomes the new root to maintain the max heap property. Then, we recursively merge into one subtree only:

- If L.root wins, then merge L.r with R
- If R.root wins, merge L with R.l

Each recursive call moves one level downward in one of the treaps. Only one recursive call is made per step, and each step does O(1) work aside from recursion.

Thus, the recursion follows a single path whose length is at most the height h of the resulting treap. Therefore, merge runs in O(h) time.

Exercise 14. Show that $N(h) = F_{h+3} - 1$ where F_n is the nth Fibonacci number, defined as $F_0 = 0$, $F_1 = 1$, and for $n \geq 2$,

$$F_n = F_{n-1} + F_{n-2}.$$

Hint: Induction...

Let N(h) be the minimum number of nodes in an AVL tree of height h. From the AVL balance condition, the smallest tree of height h occurs when the two subtrees have heights h - 1 and h - 2. Thus,

$$N(h) = N(h-1) + N(h-2) + 1, h >= 2$$

With base cases N(0) = 1 and N(1) = 2.

Define M(h) = N(h) + 1. Then,

$$M(h) = M(h - 1) + M(h - 2)$$

With M(0) = 2 = F_3 and M(1) = 3 = F_4, where F_k is the Fibonacci sequence. By induction on h, this implies

$$M(h) = F_{h+3}$$

Therefore,

$$N(h) = F_{h+3} - 1.$$

This matches the base cases and satisfies the recurrence, completing the proof.