13.6 Mutating Binary Search Trees

Now that we have seen how searching works on binary search trees, we will study how they implement the two mutating operations of the Multiset ADT: value-based insertion and deletion. In both cases, we'll see how to adapt the same idea as `BinarySearchTree.__contains__` to take advantage of the BST property when recursing in the tree.

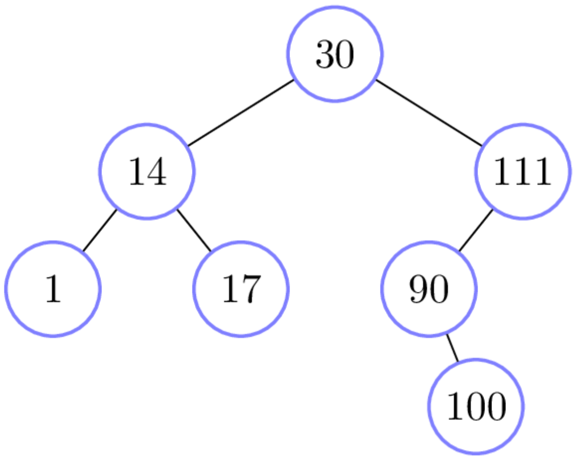
Insertion

The simplest approach for inserting into a binary search tree is to preserve the existing structure of the tree by putting the new item at the "bottom" of the tree—in other words, by making the new item a leaf.

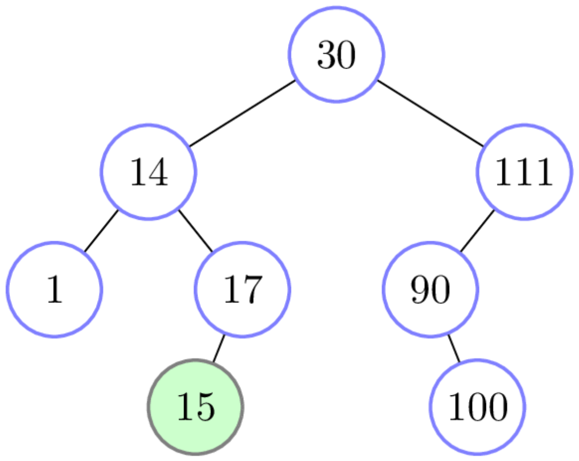
It turns out to be quite straightforward to implement this type of insertion recursively:

1. If the BST is empty, simply make the new item the root of the tree.
2. Otherwise, the item should be inserted into either the left subtree or the right subtree, while maintaining the binary search tree property.

For example, suppose we have the following binary search tree:



If we want to insert 15 into this tree, there's only one possible leaf position to put it: to the left of the 17.



The implementation of this insertion is very similar to `BinarySearchTree.__contains__`, and will be an exercise you complete in class this week. For now, let's turn our attention to deleting from a binary search tree, which is a bit more complex.

Deletion

The basic idea of binary search tree deletion is easy to state. given an item to delete, we take the same approach as `__contains__` to search for the item. If we find it, it will be at the root of a subtree (possibly a very small one—even just a leaf), where we delete it:

```
class BinarySearchTree:
    def remove(self, item: Any) -> None:
        """Remove *one* occurrence of <item> from this BST.

        Do nothing if <item> is not in the BST.
        """
        if self.is_empty():
            pass
        elif self._root == item:
            self._remove_root()
        elif item < self._root:
            self._left.remove(item)
        else:
            self._right.remove(item)

    def _remove_root(self) -> None:
        """Remove the root of this tree.

        Preconditions:
          - not self.is_empty()
        """
```

Just like we did for Tree.remove, we've pulled out a tricky part into a helper method BinarySearchTree._remove_root to implement separately. This keeps our methods from growing too long, and also helps us break down a larger task into smaller steps.

We now need to work on BinarySearchTree._remove_root. One thing we might try is to set self._root = None. Certainly this would remove the old value of the root, but this only works if the tree consists of *just* the root (with no subtrees), so removing the root makes the tree empty. In this case, we need to make sure that we also set _left and _right to None as well, to ensure the representation invariant is satisfied.

```python
class BinarySearchTree:
    def _remove_root(self) -> None:
        """..."""
        if self._left.is_empty() and self._right.is_empty():
            self._root = None
            self._left = None
            self._right = None
```

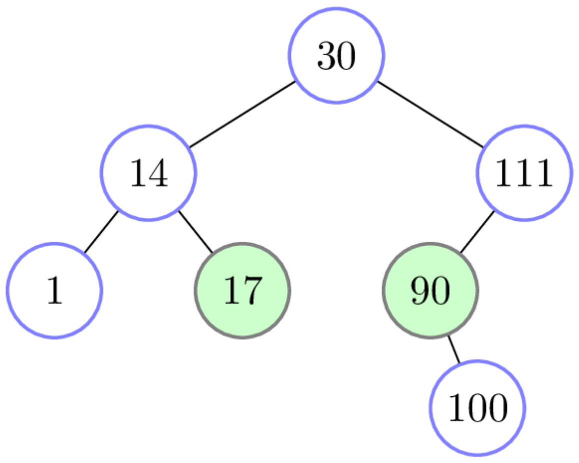
What about the case when the BST has at least one other item? We can't just set self._root = None, leaving a root value of None and yet a child that isn't None—this would violate our representation invariant. We can think of it as leaving a "hole" in the BST. We can analyse the tree structure to detect two "easy" special cases: when at least one of the subtrees is empty, but the other one isn't. In these cases, we can simply "promote" the other subtree up.

```python
class BinarySearchTree:
    def _remove_root(self) -> None:
        """..."""
        if self._left.is_empty() and self._right.is_empty():
            self._root = None
            self._left = None
            self._right = None
        elif self._left.is_empty():
            # "Promote" the right subtree.
            self._root, self._left, self._right = \
                self._right._root, self._right._left, self._right._right
        elif self._right.is_empty():
            # "Promote" the left subtree.
            self._root, self._left, self._right = \
                self._left._root, self._left._left, self._left._right
```

Finally, we need to handle the case that both subtrees are non-empty. Rather than restructure the entire tree, we can fill the "hole" at the root by *replacing* the root item with another value from the tree (and then removing that other value from where it was). The key insight is that there are **only two values** we could replace it with and still maintain the

BST property: the maximum (or, rightmost) value in the left subtree, or the minimum (or, leftmost) value in the right subtree.

Take a look at the following example: if we want to remove the root value 30, the two values we could replace it with are 17 and 90.



We'll choose to extract the maximum value from the left subtree to complete our implementation below.

```
class BinarySearchTree:
    def _remove_root(self) -> None:
        """..."""
        if self._left.is_empty() and self._right.is_empty():
            self._root = None
            self._left = None
            self._right = None
        elif self._left.is_empty():
            # "Promote" the right subtree.
            self._root, self._left, self._right = \
                self._right._root, self._right._left, self._right._right
        elif self._right.is_empty():
            # "Promote" the left subtree.
            self._root, self._left, self._right = \
                self._left._root, self._left._left, self._left._right
        else:
            self._root = self._left._extract_max()

    def _extract_max(self) -> Any:
        """Remove and return the maximum item stored in this tree.

        Preconditions:
          - not self.is_empty()
        """
```

We've once again kicked out the hard part to another helper, `BinarySearchTree._extract_max`. Finding the maximum item is easy: just keep going right to bigger and bigger values until you can't anymore. And removing that maximum is much easier than our initial problem of BST deletion because that maximum has at most one child, on the left.[1] Here's the method:

[1] How do we know that?

```python
class BinarySearchTree:
    def _extract_max(self) -> Any:
        """Remove and return the maximum item stored in this tree.

        Preconditions:
          - not self.is_empty()
        """
        if self._right.is_empty():
            max_item = self._root
            # Like remove_root, "promote" the left subtree.
            self._root, self._left, self._right = \
                self._left._root, self._left._left, self._left._right
            return max_item
        else:
            return self._right._extract_max()
```

The single base case here is actually handling two scenarios: one in which `self` has a left (but no right) child, and one in which it has *no* children (i.e., it is a leaf). Confirm for yourself that both of these scenarios are possible, and that the single base case handles both of them correctly.