like a row in one of our 3-column tables.

CSC148 - Deletion from a Binary Search Tree

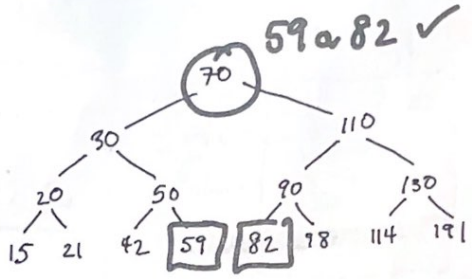
Part 1: Warm-Up

Before we write any more code for BST deletion, we will look at an example to gain some intuition about how we could delete values from a binary search tree.

1. Suppose you have to delete a value from the BST below. What would be an extremely easy value to delete, without changing the rest of the tree?

any leaf

59 or 82 ✓ leaf



2. Suppose instead you have to delete root value, 70. Ugh. One strategy is to find another value in the tree that can replace the 70 *but leave the rest of the BST unchanged.*

Could 110 replace the 70? X Could 20 replace the 70? X Could 98 replace the 70? ____

Exactly which values can replace the 70, but leave the rest of the BST unchanged? 59 or 82

3. Write the in-order traversal of this tree.

15 20 21 30 42 50 59 70 82 90 98 110 114 130 19

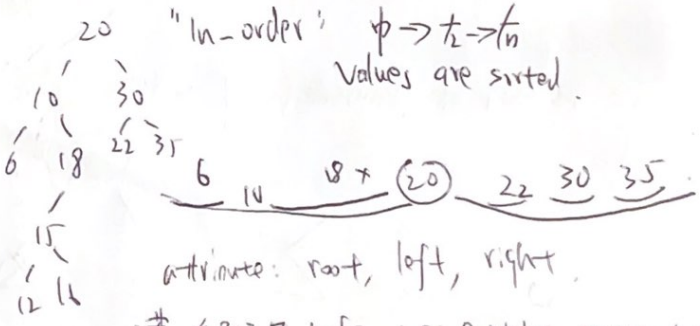
Part 2: Deleting the root of the tree

We saw in lecture that we can implement the `BinarySearchTree.delete` method as follows, using a helper to do the "hard" part:

```
class BinarySearchTree:
    def delete(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 item == self._root:
            self.delete_root()
        elif item < self._root:
            self._left.delete(item)
        else:
            self._right.delete(item)


    def delete_root(self) -> None:
        """Remove the root of this BST.

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

BST
- at most 2 kids per node
- value ordered in a sense (search faster)

"in-order": $t \to t_L \to t_R$
values are sorted.



6 10 15 18 20 22 30 35

attributes: root, left, right

有一个BST是 leaf, left & right empty tree

不是 empty

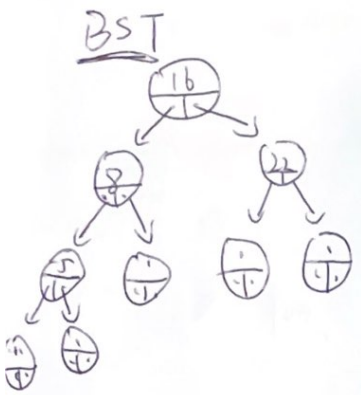
这样在recursion中不用"if"去 guard a None value

Now, we'll lead you through the cases to develop an implementation of the `BinarySearchTree.delete_root` method, in a similar fashion to what we did for `Tree.delete_root` last week.

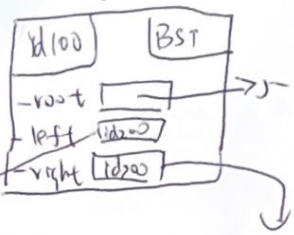
like a row in one of our 3-column tables.

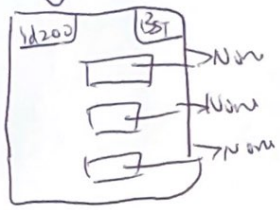
Case 1: self is a leaf

Suppose self is a leaf (i.e., its left and right subtrees are empty). Think
(1) fill in the if condition to check whether self is
for this case. Review the BST re



Memory model

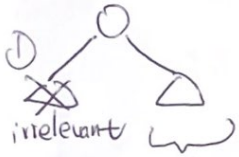


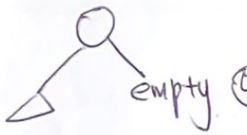


- smaller: return all items smaller than <item>

#13

- #max: 一直找right child





② empty: return None.

③ leaf: return self._root

④ return: self._root

合并

RI:
if root is not None, then left & right are BST. Even if this is a leaf

- find all items: (return all items)
Go both sides.

- count number of occurences of <item> in BST

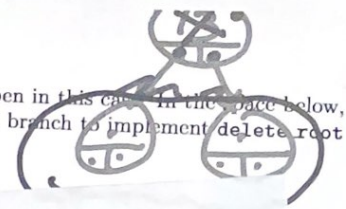
#13

like a row in one of our 3-column tables.

Case 1: self is a leaf

Suppose `self` is a leaf (i.e., its left and right subtrees are empty). Think about should happen in this case. In the space below, (1) fill in the `if` condition to check whether `self` is a leaf, and (2) fill in the body of the `if` branch to implement `delete_root` for this case. Review the BST reprsentation invariants!

```
def delete_root(self) -> None:
    if self._left is ...
```



做recursion 先要把情況分類

def delete(self, item: Any) -> None: (Remove item from one BST)

① like test cases.

Scenario

a) Item at root, not one node

call a helper & delete node

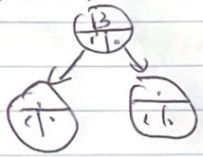
b) Item not at root, item < root

Recurse on left child.

child do all the work

c) Item not at root, item > root.

d) one-node tree, item == root



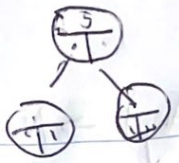
self._root = None,
不用recursion
self._left = None
self._right = None

e) empty tree — pass

a) b) c) e) 都很必要的，a) 可能可以handle d).

like a row in one of our 3-column tables.

Case 1: self is a leaf



Case1 self is a leaf.

~~if self~~ #leaf 有root, left & Right Nam

if self.root and (not self.left) and (not self.right):

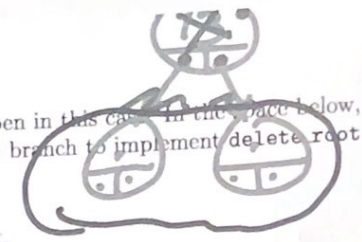
self.root = None.

Case 3: both subtrees are non-empty

like a row in one of our 3-column tables.

Case 1: self is a leaf

Suppose self is a leaf (i.e., its left and right subtrees are empty). Think about should happen in this case. In the space below, (1) fill in the if condition to check whether self is a leaf, and (2) fill in the body of the if branch to implement delete_root for this case. Review the BST reprsentation invariants!



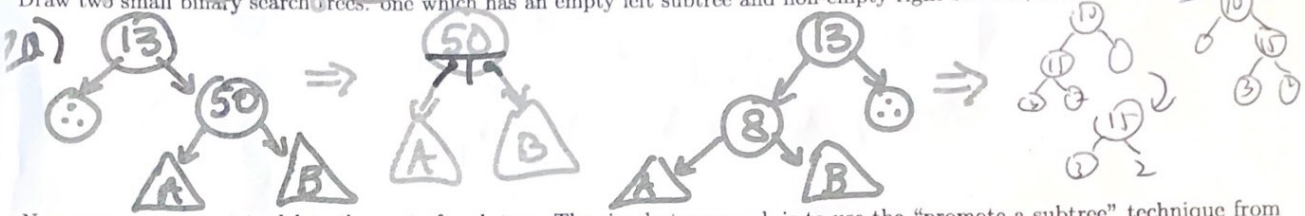
```
def delete_root(self) -> None:
    if self._left.is_empty() and
       self._right.is_empty():
        self._root = None
        self._left = None
        self._right = None
```

左空+右空.

```
# Case 1: this BST is a leaf
if self._left.is_empty() and self._right.
   is_empty():
    self._root = None
    self._left = None  左右的pointer
    self._right = None  object 没有被reference, memory free
```

Case 2: exactly one of self's subtrees are empty

Draw two small binary search trees: one which has an empty left subtree and non-empty right subtree, and vice versa.



Now suppose we want to delete the root of each tree. The simplest approach is to use the "promote a subtree" technique from last week. Review this idea, and then fill in the conditions and implementations of each elif.

```
def delete_root(self) -> None:
    if ...:  # Case 1
        ...
```

as above

2a) elif self._left.is_empty() and
    self._left, self._right, self._root
    =
    self._right._left, self._right._left, self._right._root

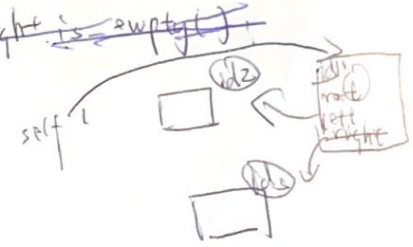
#know: not both empty

```
# Case 2a: empty left, non-empty right
```

2b) elif

```
# Case 2b: non-empty left, empty right
```

Homework: write this case



有child方法

elif self._left.is_empty() ~~and self._right~~ not ~~self._right.is_empty()~~:
    self._root = self._right._root
    self._left = self._right._left
    self._right = self._right._right

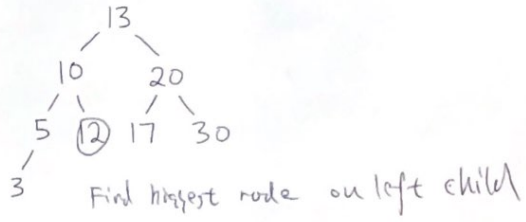
elif self._right.is_empty() and self._left.is_empty():

若写成 self = self._right:
self是local variable, 此举只会让 self 指向 right. 并没改变tree。并没有改变任何。

Case 3: both subtrees are non-empty

We'll finish this together

Suppose we have the following BST, whose left and right subtrees are *both* non-empty.



1. For this case, as we discovered earlier, we can *extract a value* from one of the subtrees and use it to replace the current root value. We need to do so carefully, to preserve the *binary search tree property*, since this is a representation invariant! Look at the sample BST above, and suppose we want to replace the root 13. Circle the value(s) in the subtrees that we could use to replace the root, and make sure you understand why these values (and *only* these values) work.

2. Since there are two possible values, you have a choice about which one you want to pick. In the space below, write a helper method that you could call on `self._left` or `self._right` to extract the desired value, and then use that helper to complete the implementation of `delete_root`.

```
def delete_root(self) -> None:
    ...
    else:                                  # Cases 1 and 2 omitted
                                           # Case 3: non-empty left, non-empty right
        largest_left_val = find_largest_node(self._left)
        self._root = largest_left_val

# Write your helper below:
```

We only call it when we have 2 childs.

pop_max,

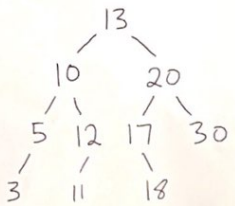
```
def find_largest_node(self) -> int:
    if self._right.is_empty():
        return self._root
    else:
        return find_largest_node(self._right)
```

base case 为 right 是空, return.

去掉 left child 最大值, 并且 return.

```
if self._right.is_empty(), and self._left.is_empty():
    largest_val = self._root
    self._root = None
    self._left = None
    self._right = None
    return largest_val
elif self._right.is_empty():
    largest_val = self._root
    self._root = self._left._root
    self._left = self._left._left
    self._right = self._left.right
    return largest_val
else:
    return find_largest_val(self._right)
```

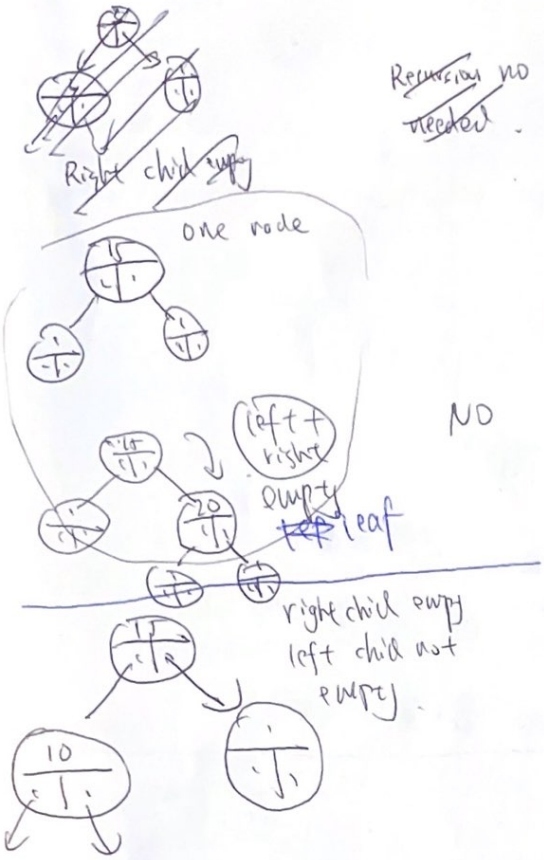
3. Check your assumptions: did you assume that the value you were extracting is a leaf? Consider the following tree...



def pop_max(self) -> Any:

"""Remove + return largest val in this BST"""

Scenarios Rec? Recursion + What they mutate / return? What's our job



~~Recursion no needed.~~

NO

remove right,
return right val.

mutate self to self.left.
return self.val.
promote left child

No Rhs, but Lhs.