

CSCI-SHU 210 Data Structures

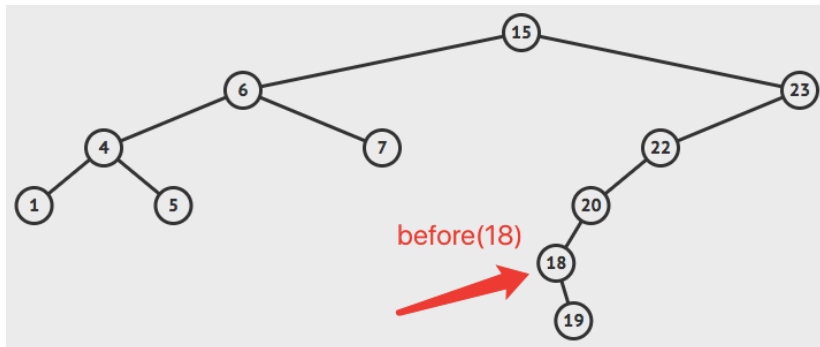
Recitation9 Worksheet Binary Search Trees

Today's recitation, we will first practice Binary Search Tree operations by hand, then we will start coding.

1. Practice the following Binary Search Tree operations:

➤ **Before(node)**

○ **Case 1**



a). What is before(18) for the BST above?

15

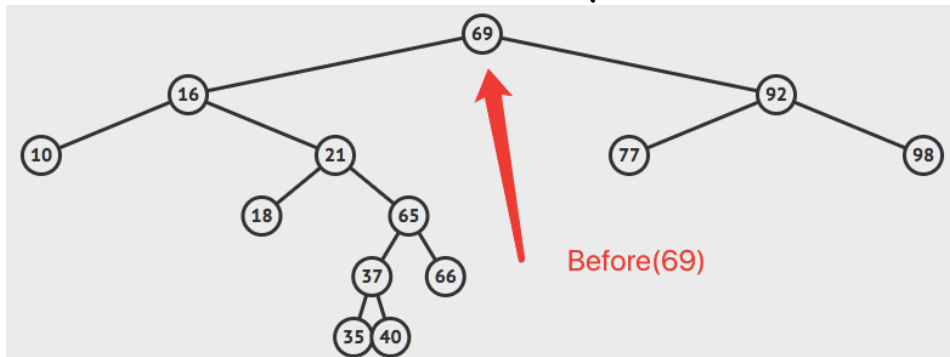
b). There are two cases in **before(self, node)** function on the next page. Which case is executed for before(18)? Highlight case 1 in the code.

case 1 highlighted in green

c). Briefly describe what to do, in order to find before(18) for the BST above.

To find the node traversed in-order before (18), since (18) is a left child of its parent, we go all the way upward, until we meet a node who is the right child of its parent, we

○ **Case 2** return that node's parent; Or, we return None as the one before (18).



a). What is before(69) for the BST above?

66

b). There are two cases in **before(self, node)** function on the next page. Which case is executed for before(69)? Highlight case 2 in the code.

Highlighted in Yellow.

c). Briefly describe what to do, in order to find before(69) for the BST above.

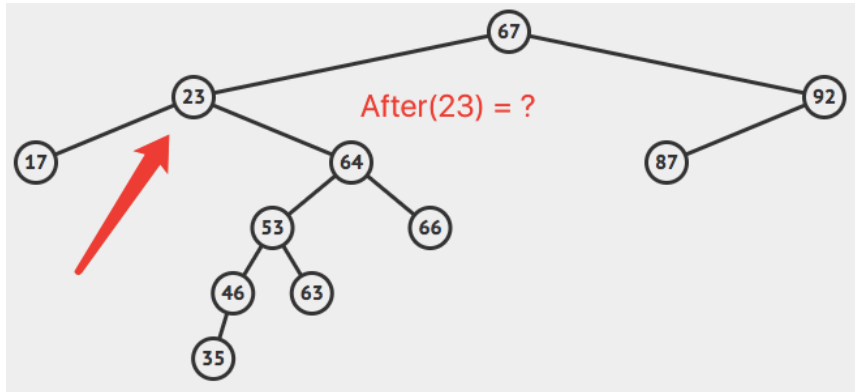
We could determine that the left child of (69) is not None, so we find the Last Item in its left subtree. To acquire this, we start by using (69)'s left child as root, and go all the way right, until we reach someone whose right child is None. We return that node as the one before (69).

- Code snippet for before(self, node)

```
1. def _subtree_last_position(self, node):
2.     """Return the node that contains the last item in subtree rooted at given node.
3.     """
4.     walk = node
5.     while walk._right is not None:          # keep walking right
6.         walk = walk._right
7.     return walk
8.
9. def before(self, node):
10.     """Return the node that is just before the given node in the natural order.
11.     Return None if the given node is the first position.
12.     """
13.     if node._left is not None:
14.         return self._subtree_last_position(node._left)
15.     else:
16.         # walk upward
17.         walk = node
18.         above = walk._parent
19.         while above is not None and walk == above._left:
20.             walk = above
21.             above = walk._parent
22.         return above
```

the highlighted green part is for case 1.
yellow part is for case 2

➤ After(node)
○ Case 1



if has right:
go all the way left of its right

if is right child.
go left upward,

until some one is a left child

/ None .

return that node's parent

a). What is after(23) for the BST above?

35

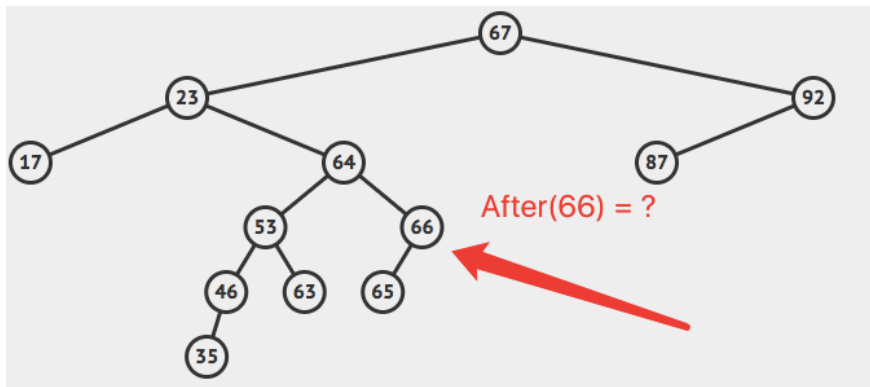
b). There are two cases in **after(self, node)** function on the next page. Which case is executed for after(23)? Highlight case 1 in the code.

highlighted in green

c). Briefly describe what to do, in order to find after(23) for the BST above.

The right of (23) is not None, we go all the way left from (23)'s right, and we go all the way left to find the last element in the sub-tree, in this case, we find (35). Its left child is None.

○ Case 2



a). What is after(66) for the BST above?

67

b). There are two cases in **after(self, node)** function on the next page. Which case is executed for after(66)? Highlight case 2 in the code.

Highlighted in yellow.

c). Briefly describe what to do, in order to find after(66) for the BST above.

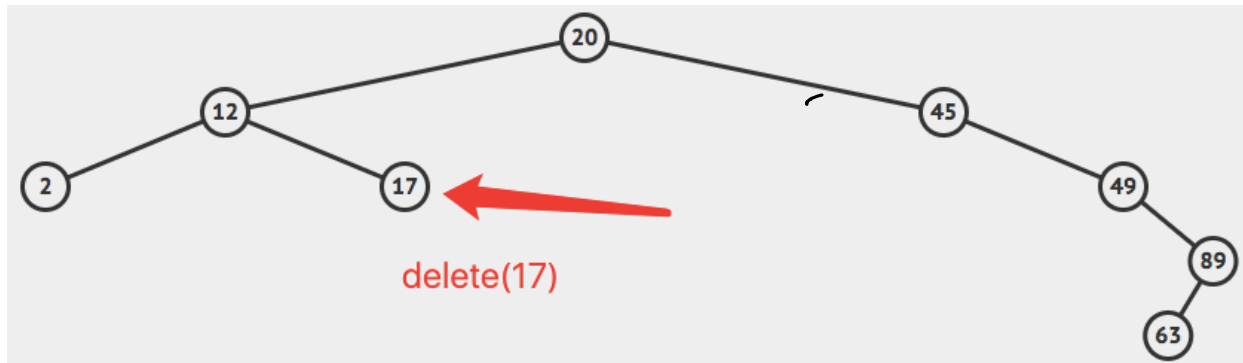
From In-order traversal, we could determine that (66) is a right child of its ancestor, so, we go all the way left upward, as long as the node's ancestor is a right child, we don't stop. Until we meet a node who is the left child of its parent, we return that node's parent. Or, we return None.

- Code snippet for `after(self, node)`

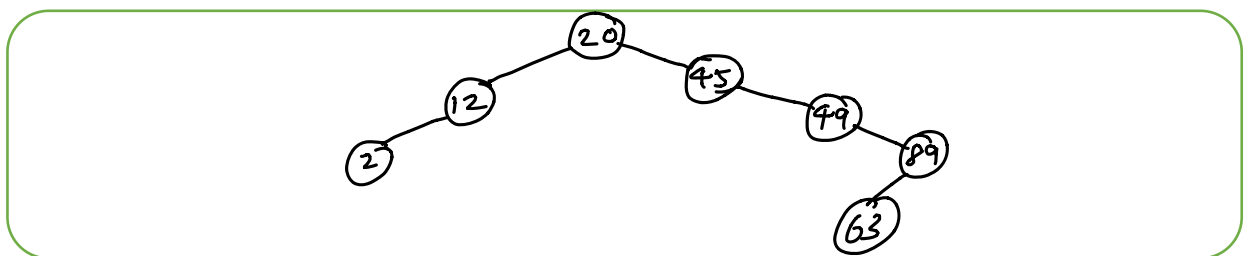
```
1. def _subtree_first_position(self, node):
2.     """Return the node that contains the first item in subtree rooted at given node
   ."""
3.     walk = node
4.     while walk._left is not None:          # keep walking left
5.         walk = walk._left
6.     return walk
7.
8. def after(self, node):
9.     """Return the node that is just after the given node in the natural order.
10.
11.     Return None if the given node is the last position.
12.     """
13.     if node._right is not None:
14.         return self._subtree_first_position(node._right)
15.     else:
16.         walk = node
17.         above = walk._parent
18.         while above is not None and walk == above._right:
19.             walk = above
20.             above = walk._parent
21.         return above
```

➤ **delete(node)**

○ **Case 1**



a). Suppose we have the BST above. Draw the BST after delete(17).

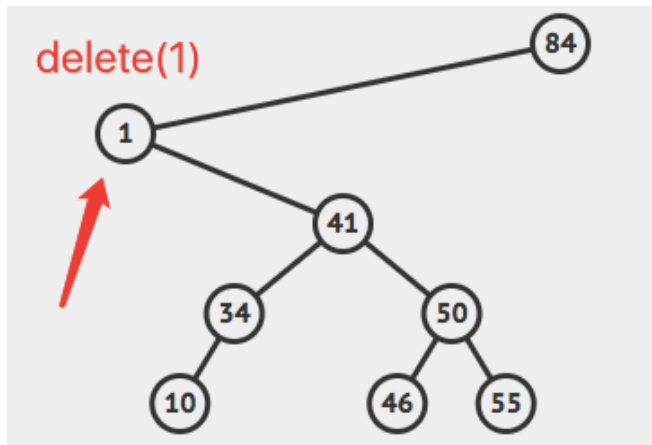


b). There are two cases in **delete(self, node)** function on the next page. Case 1 and case 2 are combined in the same function. Can you locate this function?

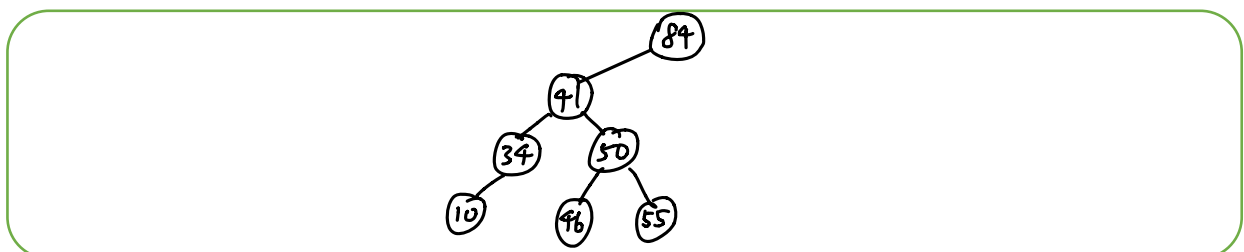
The function is: **self.delete**, it handles when the node has less than 2

c). Briefly describe what to do, in order to delete(17) from the BST above. child.
Since the left and right child are both None, and since (17) is a right child of its parent, we need simply to replace its parent's right child with None.

○ **Case 2**



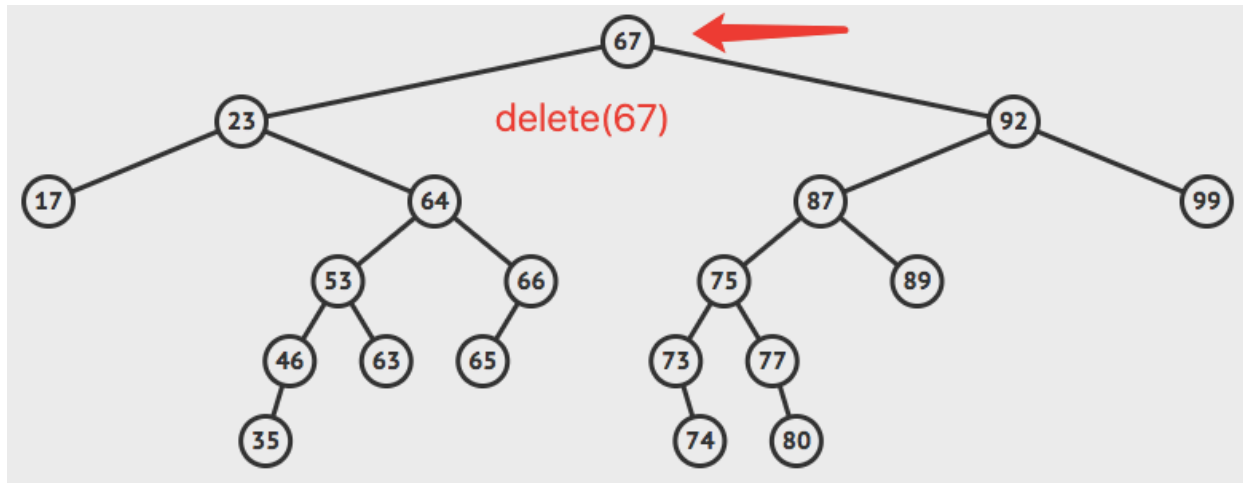
a). Suppose we have the BST above. Draw the BST after delete(1).



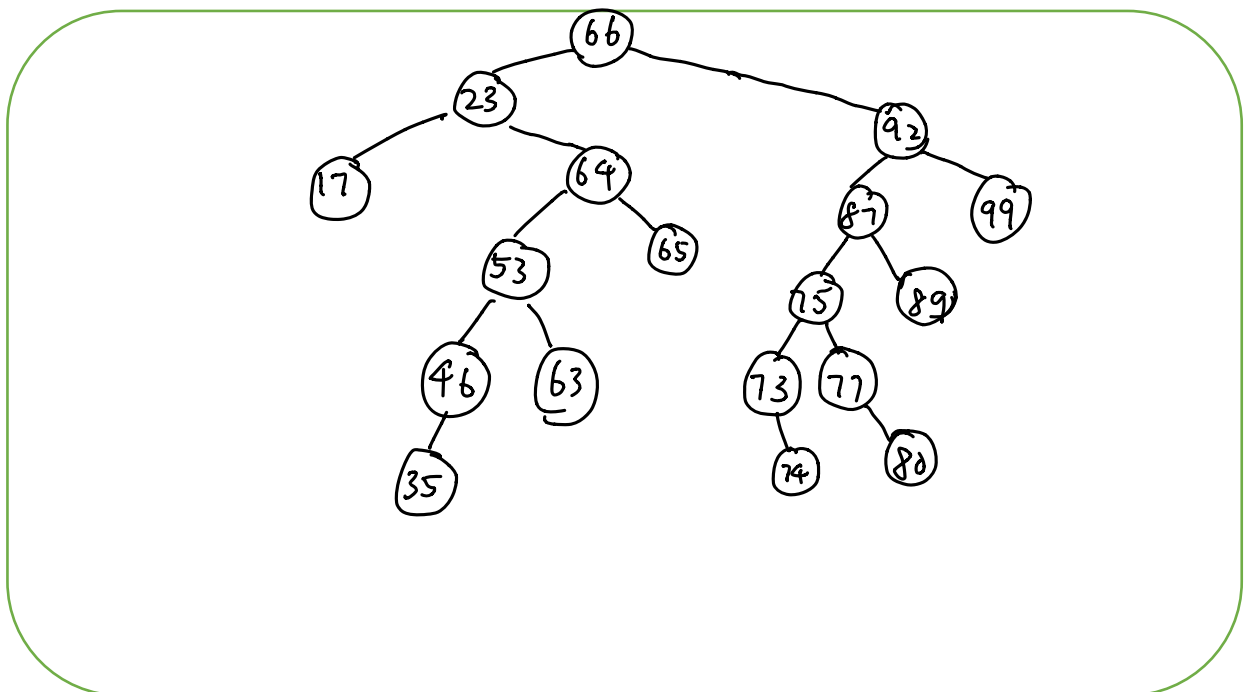
b). Briefly describe what to do, in order to delete(1) from the BST above.

node 1 only have 1 children. First, we set the parent of its children to be node (1)'s parent. Then, we determine whether (1) is a right or left child of its parent. we set its parent's left child to be (1)'s child if (1) is the left child originally. and vice versa.

○ Case 3



a). Suppose we have the BST above. Draw the BST after delete(67).



b). There are two cases in `delete(self, node)` function on the next page. Can you locate the code for case 3?

The code is in green. It belongs to the code when the node has 2 children.

c). Briefly describe what to do, in order to delete(67) from the BST above.

In order to delete 67, since 67 has two child, we want to first find out the nearest value to 67 that is smaller than 67. To do this, we need to find the last element in 67's left subtree. That's we go all the way right inside the subtree, until we reach 66, whose right is None. We then replace the value of Node(66) and Node(67). Now, 67 is at the bottom of the tree with only one child - (65), which is 66's left child. Now, the case is similar to case 2, we only need to connect (65) with Node(67)'s parent, and we delete it from the tree.

The key is to find the biggest value that's smaller than the node to delete, and do a swap.

- Code snippet for `delete(self, node)`

```
1. def _delete(self, node):
2.     """Delete the given node, and replace it with its child, if any.
3.
4.     Return the element that had been stored at the given node.
5.     Raise ValueError if node has two children.
6.     """
7.     if self.num_children(node) == 2:
8.         raise ValueError('Position has two children')
9.     child = node._left if node._left else node._right # might be None
10.    if child is not None:
11.        child._parent = node._parent # child's grandparent becomes parent
12.    if node is self._root:
13.        self._root = child # child becomes root
14.    else:
15.        parent = node._parent
16.        if node is parent._left:
17.            parent._left = child
18.        else:
19.            parent._right = child
20.    self._size -= 1
21.    return node._element
22.
23. def delete(self, node):
24.     """Remove the given node."""
25.     if node._left and node._right: # node has two children
26.         replacement = self._subtree_last_position(node._left)
27.         self._replace(node, replacement._element) # from BinaryTree(class Tree)
28.         node = replacement
29.     # now node has at most one child
30.     parent = node._parent
31.     self._delete(node) # inherited from BinaryTree(class Tree)
32.     self._rebalance_delete(parent) # (This line only works in AVL Tree)
33.
```

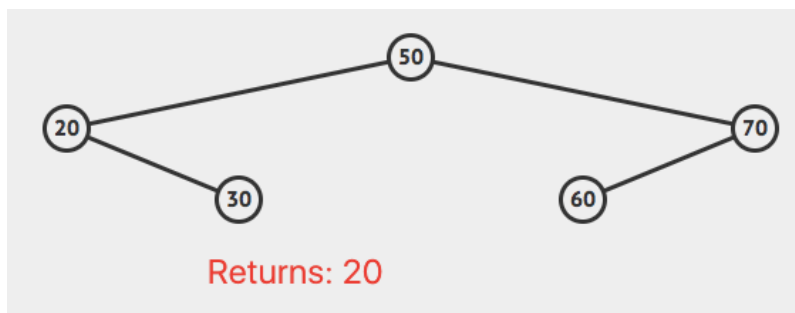
2. Coding exercises.

Your task 1: Try to perform some operations on our BinarySearchTree class.

1. Insert 0, 1, 2, 3, 4 into the tree.
2. Get the Node of 2 by calling `get_node(self, v)` function.
3. Use `before(self, node)` function to get node of 1.
4. Use `after(self, node)` function to get node of 3.
5. Delete 0, 1, 2, 3, 4 from the tree.

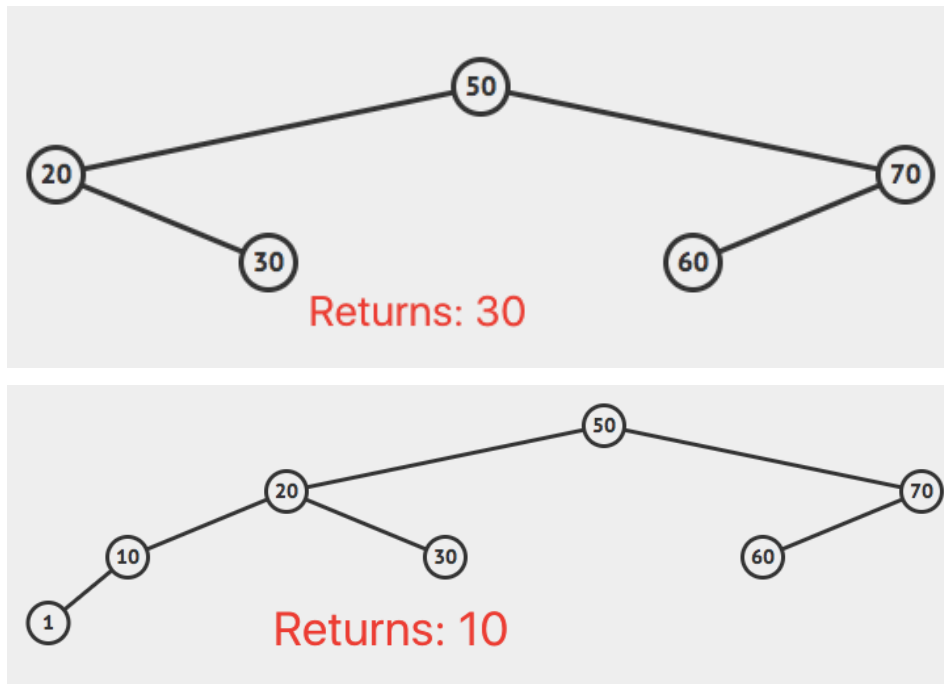
Your task 2: Implement function `minimum(self)` at line 405. When called, the minimum element within the tree is returned.

For example:



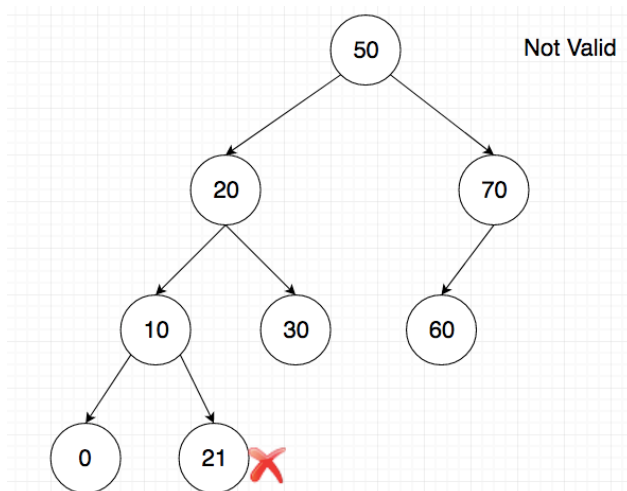
Your task 3: Implement function `second_minimum(self)` at line 412. When called, the second smallest element within the tree is returned.

For example:



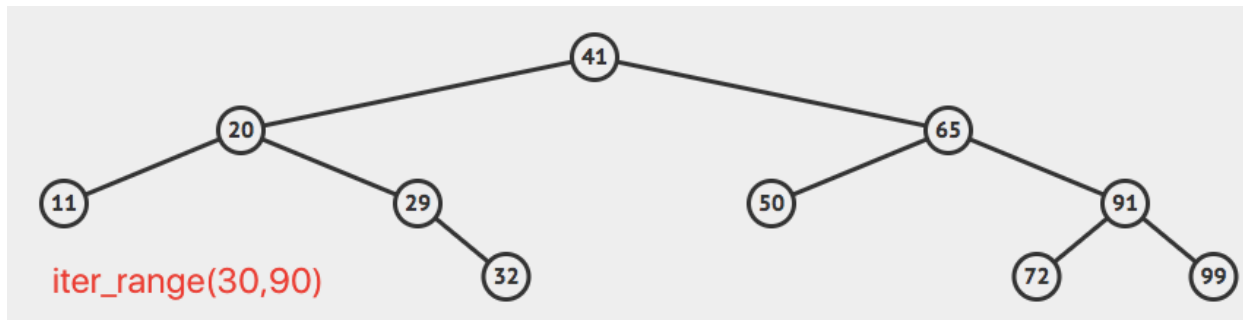
Your task 4: Implement function `is_valid(self)` at line 419. When called, returns True if the self tree is a valid binary search tree. Returns false otherwise.

For example:



Your task 5: Implement function `iter_range(self, start, stop)` at line 426. When called, Yield a generator that contains elements in order, such that $\text{start} \leq \text{elements} \leq \text{stop}$.

For example:



`iter_range(30, 90)` on the above BST should create an generator.

This generator object should contain:

32, 41, 50, 65, 72