CSCI-SHU 210 Data Structures

Assignment 8 Binary Search Trees & AVL Trees

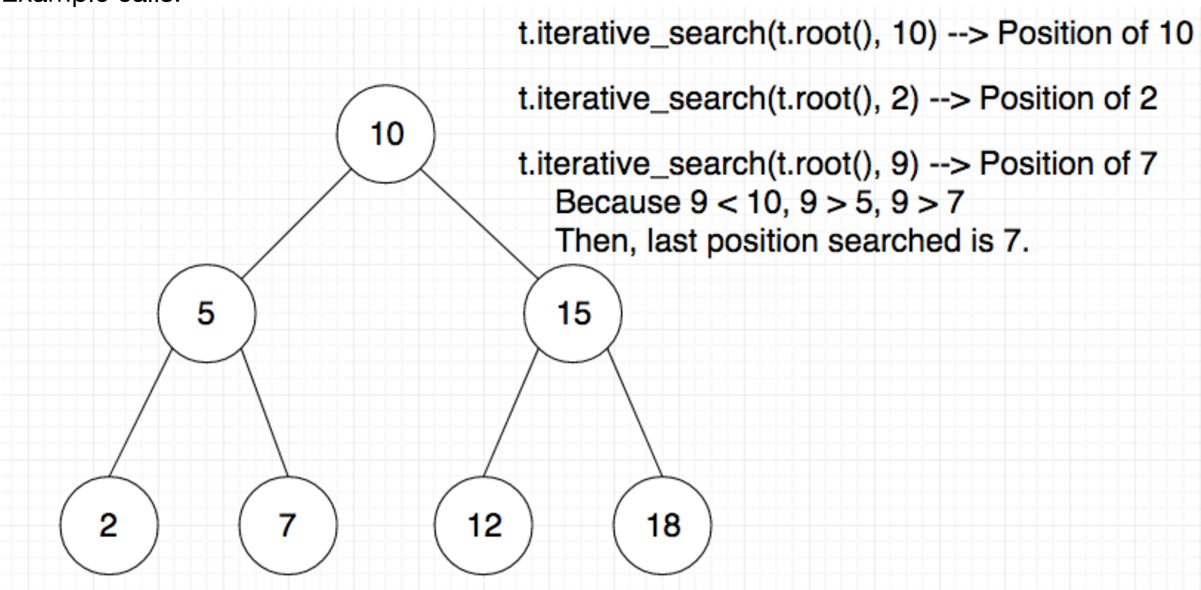
Problem 1: Iterative seach in BST

In class TreeMap, our search function `_subtree_search(self, p, k)` is implemented recursively.

"""Return Position of p's subtree having key k, or last node searched."""

Your task: Implement function `iterative_search(self, p, k)`, which performs same job as _subtree_search iteratively.

Example calls:



Important:

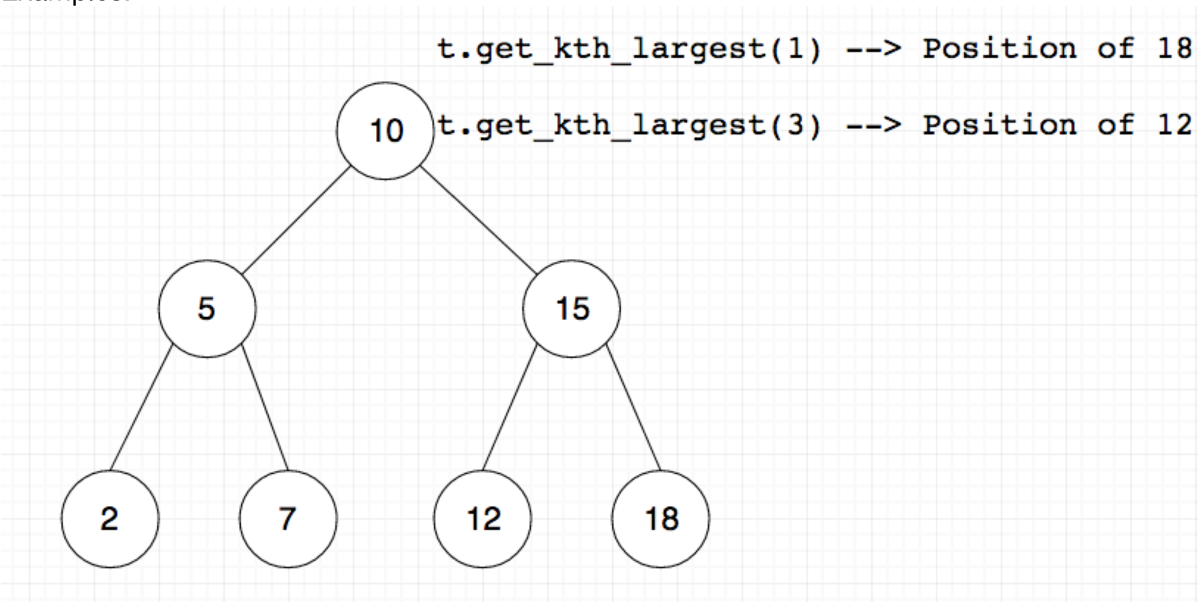
- Same job means, if same tree, same parameters are given, `iterative_search` should return the exact same position as `_subtree_search.`
- Your function should return a Position!

Problem 2: Find k-th largest element in BST

Implement function `get_kth_largest(self, k)`, which returns the position of k-th largest node within a Binary Search Tree.

If k is too large, return the largest element's position within the tree.
If k is too small, return the smallest element's position within the tree.

Examples:



Important:

- Your function should return a Position!

Problem 3: Convert a normal BST to Balanced BST

Implement function `to_balanced(self).`

Given a BST (Binary Search Tree) that may be unbalanced, convert it into a balanced BST that has minimum possible height.

```
Before calling bst.to_balanced():
          4                                   1
        /   \                                  \
       3     5                                   2
      /       \               or                  \
     2         6                                   3
    /           \                                   \
   1             7                                   4
After calling:                                        \
       4                                               5
    /    \                                              \
   2      6                                              6
 /  \    /  \                                             \
1    3  5    7                                             7
```

Important:

- You have two options to solve this problem, both options are valid:
  - No modification to original tree, return a new TreeMap object that is balanced.
  - Perform balancing on original tree, return nothing.

- For trees that might have multiple results, as long as the resulting tree has minimum possible height, then the resulting tree is acceptable.

Problem 4: Re-implement AVLTreeMap

AVLTreeMap._Node was implemented as the following:

```
class _Node(TreeMap._Node):
        """Node class for AVL maintains height value for balancing.

        We use convention that a "None" child has height 0, thus a leaf has height 1.
        """
        __slots__ = '_height'         # additional data member to store height

        def __init__(self, element, parent=None, left=None, right=None):
            super().__init__(element, parent, left, right)
            self._height = 0           # will be recomputed during balancing
```

Now, instead of using self._height, we are going to change this variable to self._balance_factor.

Definition for balance factor:

Balance factor of a node is the height of left subtree minus height of right subtree.

The balance factor of a node is always equal to −1, 0, or 1, except during an insertion or removal, when it may become temporarily equal to −2 or +2.

Implement functions

- update_balance_factor_for_add(self, p)
- update_balance_factor_for_delete(self, p)
- update_balance_factor_for_rotate(self, p)

So the new AVLTreeMap class updates heights correctly during insertion, deletion, and rotation.

Important:

- A lot of modifications have been done in class AVLTreeMap.
- You only need to implement those three functions to make class AVLTreeMap works correctly.