Name: Daniel Quiroga (dquiroga)
Assignment: Project 5: Balance in Life
Date Submitted: Wednesday, April 25, 2018 11:50:46 AM EDT
Current Grade: Needs Grading

Submission Field:
There is no student submission text data for this assignment.

Comments:
There are no student comments for this assignment.

Files:
Original filename: BST_Test.py
Filename: Project 5 Balance in Life_dquiroga_attempt_2018-04-25-11-50-46_BST_Test.py

Original filename: Writeup.pdf
Filename: Project 5 Balance in Life_dquiroga_attempt_2018-04-25-11-50-46_Writeup.pdf

Original filename: Fraction.py
Filename: Project 5 Balance in Life_dquiroga_attempt_2018-04-25-11-50-46_Fraction.py

Original filename: Binary_Search_Tree.py
Filename: Project 5 Balance in Life_dquiroga_attempt_2018-04-25-11-50-46_Binary_Search_Tree.py

TEST CASES:

I did the simple insertions that were the same for a regular BST. Just inserting and removing one element, this was to check to see that the function balanced did not impact the base cases of the tree and still worked as expected. Then I used a few test cases to check simple three node left left, left right, and vice versa rotations. This was to see that the None node floating variables were properly taken care of and no issues arose. Then instead of having variables of node none, I used actual nodes to see that the Balanced function did a good job with the rotations and was not impacted by either a none node or a regular node. I then began to insert just multiple values and checked up to see that balance was correctly fixing the tree and also the height. At first the height was not updating properly, which was impacting the way balance was working since the calc balance function depended on the height of the nodes being updated as rotations occurred. I then created a multi-level tree that had a few rotations in order to be built. The insertion aspect worked properly. Then I began by removing one node and checking to see that the height and balance worked. I continued to do so for three until we were back to a simple tree. By the multiple insertions and rotations needed. I am confident that the code has worked. The rotation of the tree kept working properly and the height also was updated whenever there was a rotation that may have lowered the height a total of one or two levels.

Performance Analysis:

Inside of the inner Node Class:

Constructor: I had references to the value of node, setting the new nodes right and left equal to None and by default whenever the node is created it has a height of one since nodes are always inserted at leaf positions.

Get_node_height: The two functions work in constant time. They do not enter any loops and just do basic arithmetic to replace the previous height of the node with a proper value.

The BST class:

Insert_element and __rins: this includes the private function, ***__rins which also calls on the balance function,*** which is the recursive function that helps find the correct position to insert the given element. In the worst case, the function will run in log(n) time. Since BST are balanced, in order to locate a specific location, the insertion is dependent on the height of the tree at that given point. Each time hacking away half of the tree. Inside, the recursive function, I call upon the get_node_height function in order to update the height of each node as the recursive call goes through, this has no impact on the time since the function, get node height, works in constant time to update the value of height for the nodes. The **__balanced** function also runs in constant time. It first calls on the calc_balance function to do a constant time computation. This then just uses if statements to see what type of imbalance has occurred and then calls on the constant function again. Each time the tree is balanced, I again call on get node height which is a constant time function to update the heights of all the nodes that have been moved or rotated.

Remove_element and __rrem: this includes the private function, **__rrem which also calls on the balance function**, which is the recursive call that helps locate the specified element and

remove and update the tree properly. In the worst case, that is removing from the root with an balanced tree, that has height h. The function would run in log(n) time. The function would enter a loop, using, ***__smallest_value,*** which would locate the leftmost node on the tree rooted to the right of the root. Each time the function goes to a new level, half of the tree is taken away. The balanced function is called as well which runs in constant time. In the balanced another constant time function, calculates the. Imbalance. This has no impact on the time it takes the function to run. The get_node_height is also called during the function, but this runs in constant time in order to update the node height during the recursive call and has no impact on the overall time. The get node height is also called in balance to update the nodes impacted by rotation during the call.

In_order, pre_order, post_order, to_list: All four traversals work in the same performance time. The worst case would be a balanced tree with height n. The function would run in linear time since it would continue to execute based off of the number of nodes in the tree since it has to visit every single one of them. Then it is a matter in how the string/array is created. As a node is added the time it takes for the traversals to function is also added.

Get_height: This function runs in constant time, since the height of the root position is updated every time an insertion or removal recursive call is used. Thus, all that is needed is to return the attribute n_height of the root which is the height of tree.

Fraction: This program first must insert all of the Fraction variables into a balanced BST. This will run in O(log(n)) time. After it must print out the sorted array. This will call on the function to_list() which runs in O(n) time. So for the program to run entirely, it would run in n + log(n) performance time.

The sorting strategy requires O(nlogn) time, as we insert n elements at a cost of O(logn) time each, and then return the sorted list in O(n) time: O(n*logn + n) = O(nlogn).

Writeup grade: 2.5 / 3

```
class Binary_Search_Tree:

    class __BST_Node:

        def __init__(self, value):
            self.value = value
            self.right = None
            self.left = None
            self.n_height = 1


        def get_node_height_ins(self):
            if self.right == None and self.left != None:
                self.n_height = self.left.n_height + 1

            elif self.right != None and self.left == None:
                self.n_height = self.right.n_height + 1

            elif self.right != None and self.left != None:

                if self.right.n_height > self.left.n_height:
                    self.n_height = self.right.n_height + 1

                else:
                    self.n_height = self.left.n_height + 1

        def get_node_height_rem(self):
            if self.right == None and self.left != None:
                self.n_height = self.left.n_height + 1

            elif self.right != None and self.left == None:
                self.n_height = self.right.n_height + 1

            elif self.right != None and self.left != None:

                if self.right.n_height > self.left.n_height:
                    self.n_height = self.right.n_height + 1

                else:
                    self.n_height = self.left.n_height + 1
            else:
                self.n_height -= 1

    def __init__(self):
        self.__root = None
```

```
def insert_element(self, value):

      x = value
      t = self.__root
      self.__root = self.__rins(x, t)

def __rins(self, value, position):
#recursive calls to reach base case
      t = position
      x = value
      if t == None:
            t = Binary_Search_Tree.__BST_Node(x)
            t.get_node_height_ins()
            return t
      if x < t.value:
            t.left = self.__rins(x, t.left)
      if x > t.value:
            t.right = self.__rins(x, t.right)
      if x == t.value:
            raise ValueError
      t.get_node_height_ins()
      return self.__balanced(t)

def remove_element(self, value):

      x = value
      t = self.__root
      self.__root = self.__rrem(x, t)

def __rrem(self, value, position):
#recursive call to remove specific value
      t = position
      x = value
      if t == None:
            raise ValueError

      if t.value == x:
            if t.right == None and t.left == None:
                  t.get_node_height_rem()
                  return None

            if t.right != None and t.left == None:
                  t.get_node_height_rem()
                  return t.right
```

```
            if t.right == None and t.left != None:
                  t.get_node_height_rem()
                  return t.left

            if t.right != None and t.left != None:
                  pos = t.right
                  m = self.__smallest_value(pos)
                  t.value = m
                  t.right = self.__rrem(m, pos)
                  t.get_node_height_rem()
                  return t

      if x < t.value:
            t.left = self.__rrem(x, t.left)

      if x > t.value:
            t.right = self.__rrem(x, t.right)

      t.get_node_height_rem()
      return self.__balanced(t)

def __smallest_value(self, position):
      t = position
      while t.left != None:
            t = t.left
      m = t.value
      return m

def __balanced(self, position):
      root = position
      t = position
      balance = self.__calc_balance(root)

      if t == None:
            return t

      if balance == -2:
            temp1 = t.left
            balance = self.__calc_balance(temp1)
            if balance == -1 or balance == 0:
                  temp = t.left.right
                  root = t.left
                  root.right = t
                  t.left = temp
                  if temp != None:
                        temp.get_node_height_ins()
```

```
                        t.get_node_height_rem()
                        root.left.get_node_height_ins()
                        if t.right == None and t.left == None:
                              t.n_height = 1
                        root.get_node_height_ins()
                        return root

                  if balance == 1:
                        root = t.left.right
                        temp = root.left
                        root.left = t.left
                        t.left = root
                        root.left.right = temp
                        temp = root.right
                        root.right = t
                        t.left = temp
                        if temp != None:
                              temp.get_node_height_ins()
                        t.get_node_height_ins()
                        if t.right == None and t.left == None:
                              t.n_height = 1
                        root.left.get_node_height_ins()
                        if root.left.right == None and root.left.left ==
None:
                              root.left.n_height = 1
                        root.get_node_height_ins()
                        return root

            if balance == 2:
                  temp1 = t.right
                  balance = self.__calc_balance(temp1)
                  if balance == 1 or balance == 0:
                        root = t.right
                        temp = root.left
                        root.left = t
                        t.right = temp
                        if temp != None:
                              temp.get_node_height_ins()
                        t.get_node_height_rem()
                        root.left.get_node_height_ins()
                        if t.right == None and t.left == None:
                              t.n_height = 1
                        root.get_node_height_ins()
                        return root

                  if balance == -1:
```

```
                        root = t.right.left
                        temp = root.right
                        root.right = t.right
                        t.right = root
                        root.right.left = temp
                        temp = root.left
                        root.left = t
                        t.right = temp
                        if temp != None:
                              temp.get_node_height_ins()
                        t.get_node_height_ins()
                        if t.right == None and t.left == None:
                              t.n_height = 1
                        root.right.get_node_height_ins()
                        if root.right.right == None and root.right.left ==
None:
                              root.right.n_height = 1
                        root.get_node_height_ins()
                        return root

            return root

      def __calc_balance(self,position):
            t = position
            bal = 0

            if t.left == None and t.right != None:
                  bal = int(t.right.n_height)

            if t.right == None and t.left != None:
                  bal -= int(t.left.n_height)

            if t.left != None and t.right != None:
                  bal = int(t.right.n_height - t.left.n_height)

            if t.left == None and t.right == None:
                  bal = 0

            return bal

      def in_order(self):

            if self.__root == None:
                  return '[ ]'
            else:
                  string = '[ '
```

```
            string += self.__rin_order(self.__root)
            string += ' ]'
            return string

    def __rin_order(self, position):
        t = position
        string = ''
        if t.left != None:
            string += self.__rin_order(t.left) + ', '
        string += str(t.value)
        if t.right != None:
            string += ', ' + self.__rin_order(t.right)
        return string

    def pre_order(self):

        if self.__root == None:
            return '[ ]'
        else:
            string = '[ '
            string += self.__rpre_order(self.__root)
            string += ' ]'
            return string

    def __rpre_order(self, position):
        t = position
        string = ''
        string += str(t.value)
        if t.left != None:
            string += ', ' + self.__rpre_order(t.left)
        if t.right != None:
            string += ', ' + self.__rpre_order(t.right)
        return string

    def post_order(self):

        if self.__root == None:
            return '[ ]'
        else:
            string = '[ '
            string += self.__rpost_order(self.__root)
            string += ' ]'
            return string

    def __rpost_order(self, position):
        t = position
```

```
            string = ''
            if t.left != None:
                  string += self.__rpost_order(t.left) + ', '
            if t.right != None:
                  string += self.__rpost_order(t.right) + ', '
            string += str(t.value)
            return string

      def get_height(self):

            if self.__root == None:
                  return 0
            else:
                  return self.__root.n_height # TODO replace pass with your
implementation

      def __str__(self):
            return self.in_order()

      def to_list(self):
            vals = list()
            if self.__root == None:
                  return vals
            else:
                  vals = self.__rlist(self.__root)
                  return vals

      def __rlist(self, position):
            t = position
            vals = list()
            if t.left != None:
                  vals += self.__rlist(t.left)
            vals.append(t.value)
            if t.right != None:
                  vals += self.__rlist(t.right)
            return vals

if __name__ == '__main__':
      pass #unit tests make the main section unnecessary.
```

```python
import unittest
from Binary_Search_Tree import Binary_Search_Tree


class BSTTester(unittest.TestCase):

    def setUp(self):
        self.tree = Binary_Search_Tree()

    def test_empty_tree(self):
        self.assertEqual('[ ]', self.tree.pre_order())
        self.assertEqual('[ ]', self.tree.in_order())
        self.assertEqual('[ ]', self.tree.post_order())
        self.assertEqual([], self.tree.to_list())
        self.assertEqual(0, self.tree.get_height())

    def test_empty_tree_insert_remove(self):
        self.tree.insert_element(6)
        self.tree.remove_element(6)
        self.assertEqual('[ ]', self.tree.pre_order())
        self.assertEqual('[ ]', self.tree.in_order())
        self.assertEqual('[ ]', self.tree.post_order())
        self.assertEqual([], self.tree.to_list())
        self.assertEqual(0, self.tree.get_height())

    def test_one_insertion(self):
        self.tree.insert_element(7)
        self.assertEqual('[ 7 ]', self.tree.pre_order())
        self.assertEqual('[ 7 ]', self.tree.in_order())
        self.assertEqual('[ 7 ]', self.tree.post_order())
        self.assertEqual([7], self.tree.to_list())
        self.assertEqual(1, self.tree.get_height())

    def test_single_left_rotation(self):
        self.tree.insert_element(5)
        self.tree.insert_element(15)
        self.tree.insert_element(25)
        self.assertEqual('[ 15, 5, 25 ]', self.tree.pre_order())
        self.assertEqual('[ 5, 15, 25 ]', self.tree.in_order())
        self.assertEqual('[ 5, 25, 15 ]', self.tree.post_order())
        self.assertEqual([5, 15, 25], self.tree.to_list())
        self.assertEqual(2, self.tree.get_height())

    def test_single_right_rotation(self):
        self.tree.insert_element(20)
        self.tree.insert_element(10)
```

```
        self.tree.insert_element(5)
        self.assertEqual('[ 10, 5, 20 ]', self.tree.pre_order())
        self.assertEqual('[ 5, 10, 20 ]', self.tree.in_order())
        self.assertEqual('[ 5, 20, 10 ]', self.tree.post_order())
        self.assertEqual([5, 10, 20], self.tree.to_list())
        self.assertEqual(2, self.tree.get_height())

    def test_double_rotation_simple_right_left(self):
        self.tree.insert_element(25)
        self.tree.insert_element(10)
        self.tree.insert_element(20)
        self.assertEqual('[ 20, 10, 25 ]', self.tree.pre_order())
        self.assertEqual('[ 10, 20, 25 ]', self.tree.in_order())
        self.assertEqual('[ 10, 25, 20 ]', self.tree.post_order())
        self.assertEqual([10, 20, 25], self.tree.to_list())
        self.assertEqual(2, self.tree.get_height())

    def test_double_rotation_simple_left_right(self):
        self.tree.insert_element(10)
        self.tree.insert_element(25)
        self.tree.insert_element(20)
        self.assertEqual('[ 20, 10, 25 ]', self.tree.pre_order())
        self.assertEqual('[ 10, 20, 25 ]', self.tree.in_order())
        self.assertEqual('[ 10, 25, 20 ]', self.tree.post_order())
        self.assertEqual([10, 20, 25], self.tree.to_list())
        self.assertEqual(2, self.tree.get_height())

    def test_double_rotation_simple_right_left_remove_root(self):
        self.tree.insert_element(25)
        self.tree.insert_element(10)
        self.tree.insert_element(20)
        self.tree.remove_element(20)
        self.assertEqual('[ 25, 10 ]', self.tree.pre_order())
        self.assertEqual('[ 10, 25 ]', self.tree.in_order())
        self.assertEqual('[ 10, 25 ]', self.tree.post_order())
        self.assertEqual([10, 25], self.tree.to_list())
        self.assertEqual(2, self.tree.get_height())

    def test_double_rotation_simple_right_left_remove_right(self):
        self.tree.insert_element(25)
        self.tree.insert_element(10)
        self.tree.insert_element(20)
        self.tree.remove_element(25)
        self.assertEqual('[ 20, 10 ]', self.tree.pre_order())
        self.assertEqual('[ 10, 20 ]', self.tree.in_order())
        self.assertEqual('[ 10, 20 ]', self.tree.post_order())
```

```
            self.assertEqual([10, 20], self.tree.to_list())
            self.assertEqual(2, self.tree.get_height())

      def test_double_rotation_simple_right_left_remove_left(self):
            self.tree.insert_element(25)
            self.tree.insert_element(10)
            self.tree.insert_element(20)
            self.tree.remove_element(10)
            self.assertEqual('[ 20, 25 ]', self.tree.pre_order())
            self.assertEqual('[ 20, 25 ]', self.tree.in_order())
            self.assertEqual('[ 25, 20 ]', self.tree.post_order())
            self.assertEqual([ 20, 25], self.tree.to_list())
            self.assertEqual(2, self.tree.get_height())

      def test_multiple_insert_simple(self):
            self.tree.insert_element(20)
            self.tree.insert_element(10)
            self.tree.insert_element(40)
            self.tree.insert_element(5)
            self.tree.insert_element(15)
            self.assertEqual('[ 20, 10, 5, 15, 40 ]',
self.tree.pre_order())
            self.assertEqual('[ 5, 10, 15, 20, 40 ]', self.tree.in_order())
            self.assertEqual('[ 5, 15, 10, 40, 20 ]',
self.tree.post_order())

      def test_multiple_insert_simple_rotation(self):
            self.tree.insert_element(20)
            self.tree.insert_element(10)
            self.tree.insert_element(40)
            self.tree.insert_element(5)
            self.tree.insert_element(15)
            self.tree.insert_element(7)
            self.assertEqual('[ 10, 5, 7, 20, 15, 40 ]',
self.tree.pre_order())
            self.assertEqual('[ 5, 7, 10, 15, 20, 40 ]',
self.tree.in_order())
            self.assertEqual('[ 7, 5, 15, 40, 20, 10 ]',
self.tree.post_order())
            self.assertEqual([5, 7, 10, 15, 20, 40], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())

      def test_multiple_insert_simple_rotation_remove_root(self):
            self.tree.insert_element(20)
            self.tree.insert_element(10)
            self.tree.insert_element(40)
```

```
            self.tree.insert_element(5)
            self.tree.insert_element(15)
            self.tree.insert_element(7)
            self.tree.remove_element(10)
            self.assertEqual('[ 15, 5, 7, 20, 40 ]', self.tree.pre_order())
            self.assertEqual('[ 5, 7, 15, 20, 40 ]', self.tree.in_order())
            self.assertEqual('[ 7, 5, 40, 20, 15 ]',
self.tree.post_order())
            self.assertEqual([5, 7, 15, 20, 40], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())

      def test_multiple_insert_simple_rotation_remove_right(self):
            self.tree.insert_element(20)
            self.tree.insert_element(10)
            self.tree.insert_element(40)
            self.tree.insert_element(5)
            self.tree.insert_element(15)
            self.tree.insert_element(7)
            self.tree.remove_element(40)
            self.assertEqual('[ 10, 5, 7, 20, 15 ]', self.tree.pre_order())
            self.assertEqual('[ 5, 7, 10, 15, 20 ]', self.tree.in_order())
            self.assertEqual('[ 7, 5, 15, 20, 10 ]',
self.tree.post_order())
            self.assertEqual([5, 7, 10, 15, 20], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())

      def test_multiple_insert_simple_rotation_remove_left(self):
            self.tree.insert_element(20)
            self.tree.insert_element(10)
            self.tree.insert_element(40)
            self.tree.insert_element(5)
            self.tree.insert_element(15)
            self.tree.insert_element(7)
            self.tree.remove_element(5)
            self.assertEqual('[ 10, 7, 20, 15, 40 ]',
self.tree.pre_order())
            self.assertEqual('[ 7, 10, 15, 20, 40 ]', self.tree.in_order())
            self.assertEqual('[ 7, 15, 40, 20, 10 ]',
self.tree.post_order())
            self.assertEqual([7, 10, 15, 20, 40], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())

      def test_double_rotation_right_left(self):
            self.tree.insert_element(20)
            self.tree.insert_element(25)
            self.tree.insert_element(26)
```

```
            self.tree.insert_element(45)
            self.tree.insert_element(40)
            self.assertEqual('[ 25, 20, 40, 26, 45 ]',
self.tree.pre_order())
            self.assertEqual('[ 20, 25, 26, 40, 45 ]',
self.tree.in_order())
            self.assertEqual('[ 20, 26, 45, 40, 25 ]',
self.tree.post_order())
            self.assertEqual([20, 25, 26, 40, 45], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())

      def test_multiple_insertions_with_rotations(self):
            self.tree.insert_element(25)
            self.tree.insert_element(30)
            self.tree.insert_element(31)
            self.tree.insert_element(50)
            self.tree.insert_element(45)
            self.tree.insert_element(55)
            self.tree.insert_element(34)
            self.assertEqual('[ 45, 30, 25, 31, 34, 50, 55 ]',
self.tree.pre_order())
            self.assertEqual('[ 25, 30, 31, 34, 45, 50, 55 ]',
self.tree.in_order())
            self.assertEqual('[ 25, 34, 31, 30, 55, 50, 45 ]',
self.tree.post_order())
            self.assertEqual([25, 30, 31, 34, 45, 50, 55],
self.tree.to_list())
            self.assertEqual(4, self.tree.get_height())

      def test_multiple_insertions_with_rotations_with_remove_1(self):
            self.tree.insert_element(30)
            self.tree.insert_element(35)
            self.tree.insert_element(36)
            self.tree.insert_element(55)
            self.tree.insert_element(50)
            self.tree.insert_element(60)
            self.tree.insert_element(39)
            self.tree.remove_element(55)
            self.assertEqual('[ 36, 35, 30, 50, 39, 60 ]',
self.tree.pre_order())
            self.assertEqual('[ 30, 35, 36, 39, 50, 60 ]',
self.tree.in_order())
            self.assertEqual('[ 30, 35, 39, 60, 50, 36 ]',
self.tree.post_order())
            self.assertEqual([30, 35, 36, 39, 50, 60], self.tree.to_list())
            self.assertEqual(3, self.tree.get_height())
```

```
    def test_multiple_insertions_with_rotations_with_remove_2(self):
        self.tree.insert_element(25)
        self.tree.insert_element(30)
        self.tree.insert_element(31)
        self.tree.insert_element(50)
        self.tree.insert_element(45)
        self.tree.insert_element(55)
        self.tree.insert_element(34)
        self.tree.remove_element(50)
        self.tree.remove_element(25)
        self.assertEqual('[ 31, 30, 45, 34, 55 ]',
self.tree.pre_order())
        self.assertEqual('[ 30, 31, 34, 45, 55 ]',
self.tree.in_order())
        self.assertEqual('[ 30, 34, 55, 45, 31 ]',
self.tree.post_order())
        self.assertEqual([30, 31, 34, 45, 55], self.tree.to_list())
        self.assertEqual(3, self.tree.get_height())

    def
test_multiple_insertions_with_rotations_with_remove_3_with_balance0(se
lf):
        self.tree.insert_element(25)
        self.tree.insert_element(30)
        self.tree.insert_element(31)
        self.tree.insert_element(50)
        self.tree.insert_element(45)
        self.tree.insert_element(55)
        self.tree.insert_element(34)
        self.tree.remove_element(50)
        self.tree.remove_element(25)
        self.tree.remove_element(30)
        self.assertEqual('[ 45, 31, 34, 55 ]', self.tree.pre_order())
        self.assertEqual('[ 31, 34, 45, 55 ]', self.tree.in_order())
        self.assertEqual('[ 34, 31, 55, 45 ]', self.tree.post_order())
        self.assertEqual([31, 34, 45, 55], self.tree.to_list())
        self.assertEqual(3, self.tree.get_height())

if __name__ == '__main__':
  unittest.main()
```

```python
from Binary_Search_Tree import Binary_Search_Tree


class Fraction:

  def __init__(self, numerator, denominator):
    # use caution here... In most languages, it is not a good idea to
    # raise an exception from a constructor. Python is a bit different
    # and it shouldn't cause a problem here.
    if denominator == 0:
      raise ZeroDivisionError
    self.__n = numerator
    self.__d = denominator
    self.__reduce()

  @staticmethod
  def gcd(n, d):
    while d != 0:
      t = d
      d = n % d
      n = t
    return n

  def __reduce(self):
    if self.__n < 0 and self.__d < 0:
      self.__n = self.__n * -1
      self.__d = self.__d * -1

    divisor = Fraction.gcd(self.__n, self.__d)
    self.__n = self.__n // divisor
    self.__d = self.__d // divisor

  def __add__(self, addend):
    num = self.__n * addend.__d + self.__d * addend.__n
    den = self.__d * addend.__d
    return Fraction(num, den)

  def __sub__(self, subtrahend):
    num = self.__n * subtrahend.__d - self.__d * subtrahend.__n
    den = self.__d * subtrahend.__d
    return Fraction(num, den)

  def __mul__(self, multiplicand):
    num = self.__n * multiplicand.__n
    den = self.__d * multiplicand.__d
    return Fraction(num, den)
```

```
def __truediv__(self, divisor):
  if divisor.__n == 0:
    raise ZeroDivisionError
  num = self.__n * divisor.__d
  den = self.__d * divisor.__n
  return Fraction(num, den)

def __lt__(self, other):
  #TODO replace pass with your implementation,
  #returning True if self is less than other and
  #False otherwise.
  if self.to_float() < other.to_float():
    return True
  else:
    return False
def __gt__(self, other):
  #TODO replace pass with your implementation,
  #returning True if self is greater than other and
  #False otherwise.
  if self.to_float() > other.to_float():
    return True
  else:
    return False

def __eq__(self, other):
  #TODO replace pass with your implementation,
  #returning True if self equal to other and
  #False otherwise. Note that fractions are
  #stored in reduced form.
  if self.to_float() == other.to_float():
    return True
  else:
    return False

def to_float(self):
  #this is safe because we don't allow a
  #zero denominator
  return self.__n / self.__d

def __str__(self):
  return str(self.__n) + '/' + str(self.__d)

# the __repr__ method is similar to __str__, but is called
# when Python wants to display these objects in a container like
# a Python list.
```

```
  def __repr__(self):
    return str(self)

if __name__ == '__main__':
  #TODO create a bunch of fraction objects and store them in an array.
  #Then insert each item from the array into a balanced BST.
  #Then get the in-order array representation of the BST using
  #the new to_list() method, which you must implement.
  #print the original and in-order traversal arrays to show that
  #the fractions have been sorted.
  arr = [Fraction(3,4), Fraction(5,10), Fraction(100,240), Fraction(5,6),
Fraction(8,40), Fraction(33,99), Fraction(34,99), Fraction(17,19),
Fraction(-2,50), Fraction(-1,2), Fraction(-2,-5), Fraction(5,-18)]
  tree = Binary_Search_Tree()
  for i in arr:
    tree.insert_element(i)
  print(arr, tree.to_list())


  arr1 = [Fraction(-6,10), Fraction(2,3),Fraction(-3,9),
Fraction(-10,25), Fraction(-2,7), Fraction(-5,13), Fraction(-1,-2),
Fraction(9,13), Fraction(13,17)]
  tree1 = Binary_Search_Tree()
  for i in arr1:
    tree1.insert_element(i)
  print(arr1, tree1.to_list())
```

```
test_empty_height (__main__.BSTTester) ... ok
test_empty_str (__main__.BSTTester) ...
ok
test_empty_traversals (__main__.BSTTester) ...
ok
test_five_left_elbow_non_root_height (__main__.BSTTester) ...
ok
test_five_left_elbow_non_root_str (__main__.BSTTester) ...
ok
test_five_left_elbow_non_root_traversals (__main__.BSTTester) ...
ok
test_five_left_non_root_height (__main__.BSTTester) ...
ok
test_five_left_non_root_str (__main__.BSTTester) ...
ok
test_five_left_non_root_traversals (__main__.BSTTester) ...
ok
test_five_right_elbow_non_root_height (__main__.BSTTester) ...
ok
test_five_right_elbow_non_root_str (__main__.BSTTester) ...
ok
test_five_right_elbow_non_root_traversals (__main__.BSTTester) ...
ok
test_five_right_non_root_height (__main__.BSTTester) ...
ok
test_five_right_non_root_str (__main__.BSTTester) ...
ok
test_five_right_non_root_traversals (__main__.BSTTester) ...
ok
test_one_height (__main__.BSTTester) ...
ok
test_one_str (__main__.BSTTester) ...
ok
test_one_traversals (__main__.BSTTester) ...
ok
test_six_left_double_height (__main__.BSTTester) ...
ok
test_six_left_double_str (__main__.BSTTester) ...
ok
test_six_left_double_traversals (__main__.BSTTester) ...
ok
test_six_left_floater_height (__main__.BSTTester) ...
ok
test_six_left_floater_str (__main__.BSTTester) ...
ok
test_six_left_floater_traversals (__main__.BSTTester) ...
ok
test_six_remove_double_left_height (__main__.BSTTester) ...
ok
test_six_remove_double_left_str (__main__.BSTTester) ...
ok
test_six_remove_double_left_traversals (__main__.BSTTester) ...
ok
test_six_remove_double_right_height (__main__.BSTTester) ...
```

```
ok
test_six_remove_double_right_str (__main__.BSTTester) ...
ok
test_six_remove_double_right_traversals (__main__.BSTTester) ...
ok
test_six_remove_no_rotate_height (__main__.BSTTester) ...
ok
test_six_remove_no_rotate_str (__main__.BSTTester) ...
ok
test_six_remove_no_rotate_traversals (__main__.BSTTester) ...
ok
test_six_remove_root_height (__main__.BSTTester) ...
ok
test_six_remove_root_str (__main__.BSTTester) ...
ok
test_six_remove_root_traversals (__main__.BSTTester) ...
ok
test_six_remove_root_twice_height (__main__.BSTTester) ...
ok
test_six_remove_root_twice_str (__main__.BSTTester) ...
ok
test_six_remove_root_twice_traversals (__main__.BSTTester) ...
FAIL
test_six_remove_single_left_height (__main__.BSTTester) ...
ok
test_six_remove_single_left_str (__main__.BSTTester) ...
ok
test_six_remove_single_left_traversals (__main__.BSTTester) ...
ok
test_six_remove_single_right_height (__main__.BSTTester) ...
ok
test_six_remove_single_right_str (__main__.BSTTester) ...
ok
test_six_remove_single_right_traversals (__main__.BSTTester) ...
ok
test_six_right_double_height (__main__.BSTTester) ...
ok
test_six_right_double_str (__main__.BSTTester) ...
ok
test_six_right_double_traversals (__main__.BSTTester) ...
ok
test_six_right_floater_height (__main__.BSTTester) ...
ok
test_six_right_floater_str (__main__.BSTTester) ...
ok
test_six_right_floater_traversals (__main__.BSTTester) ...
ok
test_three_left_elbow_height (__main__.BSTTester) ...
ok
test_three_left_elbow_str (__main__.BSTTester) ...
ok
test_three_left_elbow_traversals (__main__.BSTTester) ...
ok
test_three_left_height (__main__.BSTTester) ...
```

```
ok
test_three_left_str (__main__.BSTTester) ...
ok
test_three_left_traversals (__main__.BSTTester) ...
ok
test_three_perfect_height (__main__.BSTTester) ...
ok
test_three_perfect_str (__main__.BSTTester) ...
ok
test_three_perfect_traversals (__main__.BSTTester) ...
ok
test_three_right_elbow_height (__main__.BSTTester) ...
ok
test_three_right_elbow_str (__main__.BSTTester) ...
ok
test_three_right_elbow_traversals (__main__.BSTTester) ...
ok
test_three_right_height (__main__.BSTTester) ...
ok
test_three_right_str (__main__.BSTTester) ...
ok
test_three_right_traversals (__main__.BSTTester) ...
ok
test_two_left_height (__main__.BSTTester) ...
ok
test_two_left_str (__main__.BSTTester) ...
ok
test_two_left_traversals (__main__.BSTTester) ...
ok
test_two_right_height (__main__.BSTTester) ...
ok
test_two_right_str (__main__.BSTTester) ...
ok
test_two_right_traversals (__main__.BSTTester) ...
ok

======================================================================
FAIL: test_six_remove_root_twice_traversals (__main__.BSTTester)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "AVL_Grade.py", line 614, in test_six_remove_root_twice_traversals
    self.assertEqual('[ 50, 30, 80, 55 ]', self.__tree.pre_order())
AssertionError: '[ 50, 30, 80, 55 ]' != '[ 80, 50, 30, 55 ]'
- [ 50, 30, 80, 55 ]
?           ----
+ [ 80, 50, 30, 55 ]
?  ++++


----------------------------------------------------------------------
Ran 72 tests in 0.000s

FAILED (failures=1)
```

[3/4, 1/2, 5/12, 5/6, 1/5, 1/3, 34/99, 17/19, -1/25, -1/2, 2/5, -5/18] [-1/2, -5/18, -1/25, 1/5, 1/3, 34/99, 2/5, 5/12, 1/2, 3/4, 5/6, 17/19] [-3/5, 2/3, -1/3, -2/5, -2/7, -5/13, 1/2, 9/13, 13/17] [-3/5, -2/5, -5/13, -1/3, -2/7, 1/2, 2/3, 9/13, 13/17]