Name: Yangchen Ye (yye02)
Assignment: Project 4
Date Submitted: Sunday, April 7, 2019 1:55:59 PM 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 4_yye02_attempt_2019-04-07-13-55-59_BST_Test.py

    Original filename: Writeup.pdf
    Filename: Project 4_yye02_attempt_2019-04-07-13-55-59_Writeup.pdf

    Original filename: Binary_Search_Tree.py
    Filename: Project 4_yye02_attempt_2019-04-07-13-55-59_Binary_Search_Tree.py

CSCI 241 Data Structures
Project 4: It's Just a Jump to the Left
and a Step to the Right

Your submission for project 4 will be graded according to the following rubric. You will receive a score in each category. Your total project score will be the sum of all category scores. Categories and their weights may change with each project.

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Adherence to Specifications 20%	There are no deviations from the specifications. Items are named correctly. All files are present in the submission. The implementation and writeup both reflect a deep understanding of the project's purpose and how the specifications reinforce the relevant material.	Most components of the submission adhere to specifications. Some issues remain. Examples may include spelling or case of specified names or other editorially minor issues.	Many components of the submission conform to specifications, but key elements are incorrect. Examples may include incorrect performance measurements, algorithmic deviations from what we cover in class, or use of built-in Python features instead of custom implementations.	The submission attempts to follow the instructions, but fundamental issues persist. There is a clear misunderstanding as to the specifications and how they relate to the material at hand.	The submission may substantially incomplete. It may be missing specified components. It may implement items in ways other than specified or covered. The submission conveys either illiteracy or ignorance of the specifications.
Your Score 20 /20					

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Functionality / Testing 30%	The submission passes all test cases. Output is formatted correctly. There is no unspecified output. Main code is conditioned to be separate from class or function definitions. There is evidence of extensive thought behind testing of code. Applications based on the project's data structures work perfectly. Unit tests are well-designed with consideration to coverage and overlap. Repeated concepts from throughout the semester are now implemented correctly, such as string formatting.	The submission produces the correct result, but minor issues such as formatting. A reasonable attempt was made to cover most test case possibilities, and those tests take advantage of provided examples. Applications based on the project's data structures are mostly correct. Unit tests cover most cases well, but miss some possibilities.	There is evidence that the implementation does not consider problematic cases or special cases. The program may crash on a few test cases. Some testing is present, but it is incomplete and may not be based on the provided examples. There are some significant issues in the applications based on the project's data structures. Unit tests are sparse and missing significant coverage, but a reasonable effort is evident.	The implementation is fundamentally incorrect. Multiple cases fail, or the program crashes in multiple scenarios. The submission fails to convey a sufficient understanding of the material. There is insufficient testing present to demonstrate any meaningful correctness of the code. Applications do not function correctly. Unit tests consider only basic possibilities or test too many things at a time. Repeated concepts from throughout the semester are not correct, such as string formatting.	The program crashes on most inputs. It contains errors that imply it could not have run during development. Of the test cases that do not crash, many fail with incorrect results. Little to no effort is evident with respect to confirming functionality. Application(s) are missing. Unit tests, if present at all, vary significantly from the examples provided from project 2. Main sections other than BST_test contain code. Insufficient time spent on testing is apparent.
Your Score 30 **/30**					

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Performance 10%	The implementation takes advantage of all opportunities to improve performance.	The implementation is generally efficient, but some steps could be taken to improve performance	The implementation misses many opportunities to improve performance. Steps are taken without regard to their cost.	The implementation is naïve with respect to performance. Some sections of code take significantly longer than the algorithms described in class.	The implementation takes so long to run as to interfere with grading. it contains unnecessary loops or repetition of steps. If sections of code perform at all, they are extremely inefficient.
Your Score 10 **/10**					

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Readability 10%	Variable, function, and class names are descriptive. Comments explain sections of code where necessary. Indentation is consistent and deliberate. Test code is well structured like the provided examples.	Code is well-structured, but names could be more descriptive. Some sections of code are not obvious and could benefit from additional comments.	There is repeated code that should be moved to functions. Names are not reflective of what they reference. Inconsistent white space. The implementation contains unnecessary special cases.	The code uses inappropriate features for the problem at hand. Complex implementations are not commented. Names are poorly chosen. Repeated code degrades reliability. Functions return values inappropriately.	The implementation conveys a fundamental misunderstanding of basic programming structures, regardless of language. Loops and conditionals are inappropriately interchanged. Incorrect variables are used for computation. Sections of code are unreachable
Your Score 9 **/10**					

	Exemplary 100%	Good 90%	Satisfactory 75%	Marginal 60%	Unacceptable 0%
Writeup 30%	The writeup indicates a deep understanding of the material at hand. It employs illustrations, prose explanations, and examples to justify claims. It describes in detail the algorithmic steps that lead to performance variations (or for projects 2 and later, formal performance analysis). The document contains few if any grammatical errors.	The writeup is content-complete and generally correct, but presentation is lacking. It may fail justify some claims with illustrations or examples. It indicates that the general concepts are clearly understood, but minor improvements in presentation would benefit the reader.	The writeup convers the specified material, but in a minimal way. It draws some incorrect conclusions, or offers incorrect or incomplete justifications. Minimal effort to distinguish this submission among others is present.	The writeup is not structured as a prose document, or contains only cursory responses to the required items. The writeup appears to have been an afterthought of the implementation instead of an integral component of the project. No effort to distinguish this submission among others is present.	The writeup is missing or fails to address the required items. If it is present, it is so brief as to indicate a lack of sufficient time, analysis, and construction of rational argument.
Your Score 30/**30**					

Worst-case Performance Analysis of Methods

Class __BST_Node:

__init__():
The constructer of my *__BST_Node* class that creates a node with a given value has constant time worst-case performance, *O(1)*. What's done by the constructer is to set four spots in the memory, one to store the value given as the parameter and reference to that value by attribute *value;* two references *left* and *right* initialized to point to None; and another attribute *height* initialized to be 1. All of the above work is done in constant steps, so the whole constructer performs in constant time *O(1)*.

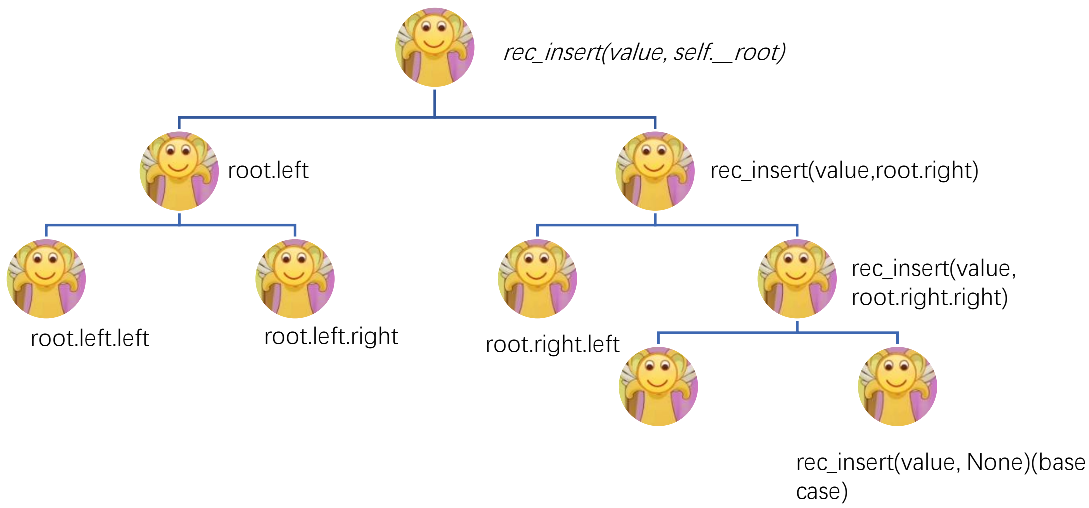
update_height(): Great level of detail for this update height method.
The *update_height()* method in the node class performs in constant time *O(1)* in its worst-case. This function updates the height of a node to be one added to the height of its left or right child depending on which one has the bigger height. Considering the possibility of left/right being None, my function catches five possible cases: left None, right None; left None, right not None; left not None, right None; left bigger than right; right bigger than left. One and only one of these cases should pass the *if* conditional and within each case the work is constant, just setting the *height* of node to be 1 + (*height* of the appropriate node), because we have constant-time access to the *height* of every node. Therefore, the worst-case performance of the *update_height()* method is also constant time *O(1)*.

Class Binary_Search_Tree:

__init__():
The constructer of the *Binary_Search_tree* class performs in constant time in the worst case. The work is just to initialize a reference *self.__root* to point to None, which is done in constant steps, making the whole constructer constant time *O(1)*.

__recursive_insert(value, node):
The worst-case asymptotic performance of my *__recursive_insert()* method is *O(h)* where *h* is the height of the binary search tree, and this is *O(n)* when the tree is not balanced, and *O(log(n))* if the tree is balanced.
First, it can be observed that the time of recursion in the worst case is always proportional to the height *h* of my binary search tree, which is the longest path from root to the None which is the child of a leaf node. This is justified because the function continues recursively as long as the control reference is not pointing at None, which has to be met by calling the child of a leaf node. Therefore, the total time of recursion is just proportional to the path starting from root to a leaf node, and the worst case is just the longest path.

Fantastic discussion and use of illustration for your insert method analysis.



Above is an illustration of the activation record when inserting a value to a tree whose height is 3. It can be seen clearly that ($1+h$) recursive functions are called, which is a function of h. Further, the work inside each recursive function is constant: when the base case is reached, constant time work is done to create and return a node object with the given value (constant time node constructer), and in the recursive case, constant operation is done to set the left (right) subtree to be the return value of further recursive function, update the height, and return the sub-root itself to higher recursive functions. Since all the work above is constant time as has been justified before, the total performance of the *__recursive_insert()* function is just a function of the height h of the tree, which is $O(h)$.

Having established that, we can determine the worst-case performance of the insertion function by exploring the relations between the height h of a tree and the total number of elements, n, of the tree. There're two possible cases:

1. When the tree is unbalanced as is the case in my implementation, in the worst-case, the height h just equals the number of elements n, which is when the tree goes all the way left or right and performs just like a linked list. In this case, the worst-case performance of my insertion function is linear time $O(n)$ since $n=h$.
2. When the tree is balanced, i.e., for every node position in the tree, the difference between the height of its left subtree and right subtree is at most 1, then it can be proved that h is at most a function of $log(n)$. Therefore, the worst-case performance of my function is logarithmic time $O(log(n))$. The proof goes as follows:

Proof for h being at most a function of $log(n)$:

To prove that h is less than a logarithmic function of n, it suffices to show that n is greater than an exponential function of h. To demonstrate this, suppose $n(h)$ is the least possible number of nodes for the balanced binary search tree of height h. Therefore, $n(1) = 1$, $n(2)$

$= 2$ since a balanced tree has to have at least one element to be height 1 and at least 2 element to be height 2. Then, for $h >= 3$, it can be observed that $n(h) = 1 + n(h-1) + n(h-2)$.

Then, since $n(h-1) > n(h-2)$, it follows that $n(h) > 2n(h-2) > 4n(h-4) > \ldots > 2^i n(h-2i)$ where $i = |h/2| + 1$ that makes $h-2i$ reach one of the base cases $n(1)$ or $n(2)$, and since the smaller one of them is $n(1)$, which is one, then it follows that the above expression is bigger than or equal to $2^{\left|\frac{h}{2}\right|+1}$

Therefore, for the best, $n(h) > 2^i n(h-2i) >= 2^{\left|\frac{h}{2}\right|+1}$, which is an exponential function of the height h, and in fact, $n(h)$ would be larger than this to a considerable degree because we've neglected a considerable number by the process of recursively reducing $1 + n(h-1) + n(h-2)$ to $2n(h-2)$. Since in this actually impossible and overoptimized case, n is still an exponential function of h, it follows that in the actual case of a balanced binary search tree, n will be an exponential function of h. Consequently, h would be a logarithmic function of n, so the $O(h)$ performance means $O(log(n))$ performance. Q. E. D.

insert_element(value):

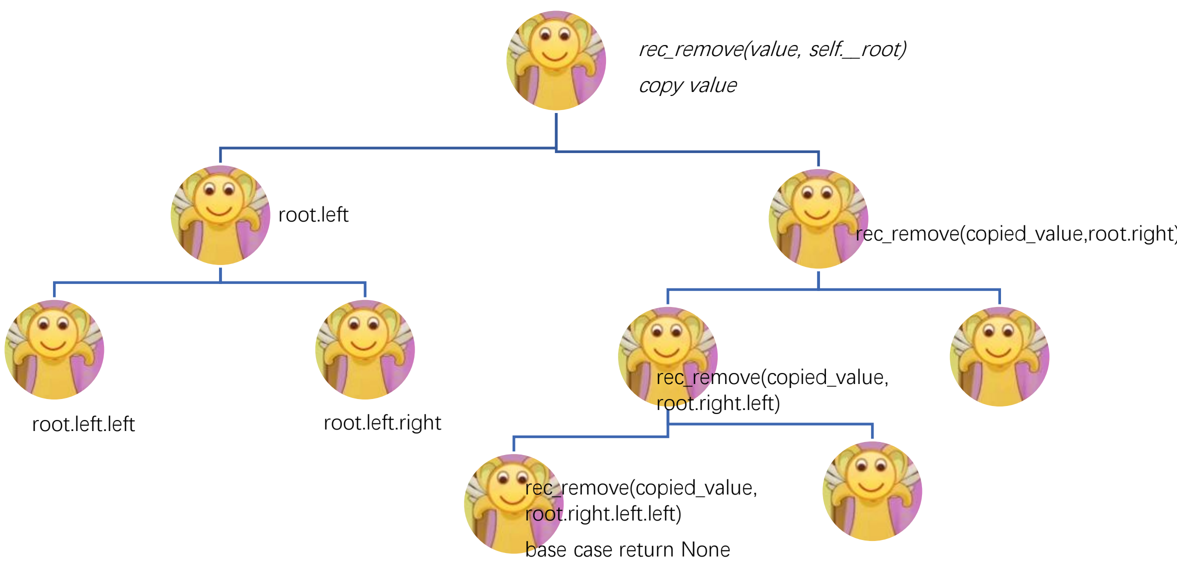
The worst-case performance of the public *insert* method is identical to the -*__recursive_insert()* method discussed above, because the public method just calls the recursive method and set the return value of that method to be the *self.__root,* which can be done in constant step while the worst-case performance of the recursive method is at least logarithmic time. Therefore, the performance of the public insert method is the same as that of the recursive one: $O(log(n))$ if the tree is balanced, $O(n)$ if there's no guarantee that the tree is balanced.

__recursive_remove(value, node):

The worst-case asymptotic performance of my *__recursive_remove()* method is $O(h)$ where h is the height of the binary search tree, and this is $O(n)$ when the tree is not balanced, and $O(log(n))$ if the tree is balanced.

First, it can be observed that the *__recursive_remove()* method would always end up in deleting a node which has no children or has one child: if the deleted node has two children, then it will not be deleted, it just copies the value of one of its child (with no more than one child) and that child will be deleted recursively. Therefore, the time of recursive calls is proportionate to the path from root to a node with one or no children. Since the longest path possible in a tree is its height, which is from root to one of its leaf, and that leaf certainly satisfies the condition of having one or no child (because it must have no children), then it follows that in the worst case, the time of recursive call is just the height of the tree. Further, the work inside each recursive call, apart from recursion, depends on the particular cases: if the control variable reaches a node with one or zero child, then constant work is done to return None or its only child; if the control variable reaches the node with two children, then the worst-case work is linear (proportional to the height) because it has to loop through the whole tree to locate the leftmost node in its right subtree, and do the remaining constant work of copying the value, setting the return value of a recursive remove to be the node's right child, update height, and return itself; and in the recursive cases, constant work is done to just set what is returned to be the node's left or right child, update the height and return the node itself.

Therefore, in the worst case, there are *h* recursive calls, one of which does a linear work proportional to the height *h* and the rest of which do constant works. Since *h* is pushed to infinity, the one recursive call doing *O(h)* work is not adequate to increase the total level of performance, leaving the overall performance a function of *h,* which is *O(h)* performance. The following graph illustrates such a worst case: the removed value is at the root position, and the leftmost node in its right subtree is at a leaf position and it takes *h* paths to go from root to that leaf.



The above example shows how a *__recursive_remove()* works in a tree of height 4: a total 4 recursive calls are made, one of which contains a *4* times while loop and the rest of which do constant work. Substituting 4 with an *h* denoting the height of the tree, and the overall performance of the *__recursive_remove()* method is *O(h).*

Based on the same discussion made above about the relations between *h* and *n,* the same argument for the performance of *__recursive_remove()* can be made here: if there's no guarantee that the tree is kept balanced, then the worst-case performance of the remove is linear time *O(n)* when the tree goes like a linked list; while if the tree is balanced, then the worst-case performance is logarithmic time *O(log(n)).*

remove_element(value):

Great job discussing the different cases that arise for the remove method!

Since this public remove method just calls the recursive method and set what is returned by that method to be the new *self.__root,* the performance of this method is strictly identical to the recursive one plus a constant step, which makes no difference on the performance when *n* is pushed to infinity. Therefore, the worst-case performance of *remove_element()* method is logarithmic time *O(log(n))* if the tree is balanced and is linear time *O(n)* if the tree is not balanded.

__recursive_inorder(node):
The worst-case performance of the *__recursive_inorder()* method is linear time *O(n)*. By the structure *if node.left is not None: recursion(left); if node.right is not None: recursion(right)*, and that every single node in a binary search tree can be reached from the root position using a combination of *left* and *right,* it is guaranteed that every node in a binary search tree is traversed by a recursive call of this function and no more recursive call is made (the recursion ends at leaf positions), making a total *n* recursive calls. Since there are a total *n* recursive calls and the work inside each call is constant, only involving concatenating a string and return that string, then the overall performance of the method is linear time *O(n).*

in_order():
The worst-case performance of the public in-order traversal is identical to its private recursive function, which is linear time *O(n).* Since it just calls the linear time recursive function and adds some constant work, involving a string concatenation and string slicing, both of which are constant time operations in python, they don't affect the overall performance of the public method. Therefore, the worst-case performance is linear time *O(n)*

__recursive_preorder(node):
For the same reason as in *__recursive_inorder(),* there are a total *n* recursive calls made in my *__recursive_preorder()* method, and the work inside each call is constant, only involving a string concatenation. Therefore, the overall performance of the recursive pre-order traversal is also linear *O(n).*

pre_order():
For the same reason as in the *in_order()* method, the worst-case performance of *pre_order()* is linear time *O(n)* identical to the private pre-order traversal function.

__recursive_postorder(node):
For the same reason as in *__recursive_inorder(),* there are a total *n* recursive calls made in my *__recursive_postorder()* method, and the work inside each call is constant, only involving a string concatenation. Therefore, the overall performance of the recursive post-order traversal is also linear *O(n).*

Nice discussion of
these traversals

post_order():
For the same reason as in the *in_order()* method, the worst-case performance of *post_order()* is linear time *O(n)* identical to the private post-order traversal function.

get_height():
The worst-case performance of my *get_height()* method is constant time *O(1),* since the *height* is an attribute kept track of in my private node class which is updated in each step of updating. Therefore, when asked to return the height, the performance is constant, either looking at the kept attribute and return it, or return 0 when there is no node.

__str__():
The performance of *__str__()* method is strictly identical to the *in_order()* method because it just calls the *in_order()* method and returns whatever is returned by the in-order traversal. Therefore, the worst-case performance is linear time *O(n).*

Purpose and Efficacy of Test Cases

Since my binary search tree structure contains two update method, *insert_element()* and *remove_element,* and to test that these update methods correctly update the tree, four things have to be tested: the in-order, pre-order, post-order traversal is right and the height is correctly updated. These four dimensions constitute a complete test, i.e., an update is successful if and only if all these four things are correct after that update. Therefore, for each insertion and removal, I have four consecutive test cases:

def test_empty_tree_string_inorder(self):
def test_empty_tree_preorder(self):
def test_empty_tree_post_order(self):
def test_empty_tree_height(self):

If the update is not expected to raise an error, then I expect to see all these four tests succeed and the tree is correctly updated; and if the update is expected to raise an error, I'd expect to see that these tests detects an error and the tree is left completely unchanged, i.e., all the four things are completely unchanged.

Note that the tests of the traversals and height are accomplished by calling respective functions, so the test cases will be passed only if (1): the update methods update the binary search tree in the correct way, and (2): the *in_order(), pre_order(), post_order(), __str__(), get_height()* functions are correct so as to present the tree in the right way. Conversely, if my test failed, it can be either that (1): the update methods are wrong, or (2): the traversal functions are wrong. However, this is not a problem for my test because (1): there is no other way to test the correctness of a tree than relying on manually written traversals, and (2): actually it can be determined by observing the pattern of the failure whether the update methods go wrong or the traversal functions go wrong. To explain more of this: in fact, two distinct traversals are able to uniquely determine a tree; therefore, if two of my traversal tests are successful and the other fails, then it shows clearly that the problem is with the other traversal function; while if the update methods are mistaken and my traversal functions are right, then it has to be the case that all three traversals fail, but fail in such a way that they still determine a consistent and unique tree, only that that tree is not what's supposed to be. To sum up, if the results of my traversal functions are consistent but still fail the test, it is a strong indicator that the problem is with my update methods; while on the other hand, the traversal functions are probably wrong if they fail to determin a consistent tree.

Another thing I have noted in my test cases is the multiple shapes a binary search tree can have. This issue is most significant when testing the *remove* method: I cannot just test *remove_leaving_empty, remove_leaving_one, remove_leving_two, etc.,* because there're a variety of possibilities inside each size of binary search tree. At very least, I have to test three different cases: remove a node with no children; remove a node with one child; and remove a node with two children. Therefore, for the remove test, I adopted the approach of *remove_root, remove_inner, remove_leaf,* corresponding to the cases of two children, one child, and no children. Since it is nearly impossible to exhaust the possible shapes and positions of insertion/removal when the size of the binary search tree reaches four or five, what I try to do is just to ensure that every possible conditionals in the insertion and removal functions are tested and as many different shapes are tested as possible.

Testing *insert_element():*

Therefore, my specific test blocks for insertion is as follows:

```
def test_empty_tree_string_inorder(self):
    #test the in_order format of an empty tree
    self.assertEqual('[ ]', str(self.__bst))
def test_empty_tree_preorder(self):
    self.assertEqual('[ ]', self.__bst.pre_order())
def test_empty_tree_post_order(self):
    self.assertEqual('[ ]', self.__bst.post_order())
def test_empty_tree_height(self):
    self.assertEqual(0, self.__bst.get_height())
```

For an empty binary search tree, I expect to see '[ ]' in all three traversals and 0 as its height.

```
def test_insert_one_inorder(self):
def test_insert_one_preorder(self):
def test_insert_one_postorder(self):
def test_insert_one_height(self):
…
def test_insert_two_left_inorder(self):
def test_insert_two_left_preorder(self):
def test_insert_two_left_postorder(self):
```

```
def test_insert_two_left_height(self):
…
def test_insert_two_error_inorder(self):
    self.__bst.insert_element(5)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(5)
    self.assertEqual('[ 5 ]', str(self.__bst))
def test_insert_two_error_preorder(self):
def test_insert_two_error_postorder(self):
def test_insert_two_error_height(self):
```

Above are testing the possible cases when inserting a binary search tree of one or two element with the possibility of error. The following are just extending this testing paradigm to the tree of three and four element, creating as many different shapes as possible and classifying the possibilities by the resulting height of the tree and the direction of insertion:

```
def test_insert_three_height_three_left_inorder(self):
…
def test_insert_three_height_three_right_inorder(self):
…
def test_insert_three_height_three_left_right_inorder(self):
…
def test_insert_three_error_inroder(self):
…
```

Testing *remove_element():*

My specific testing blocks for the remove method are as follows:

You should not be putting source code into your write up as it just adds extra bloat since the reader can just go to your source files themselves.

```
def test_remove_empty_inorder(self):
    with self.assertRaises(ValueError):
        self.__bst.remove_element(4)
    self.assertEqual('[ ]', str(self.__bst))
def test_remove_empty_preorder(self):
    with self.assertRaises(ValueError):
        self.__bst.remove_element(4)
    self.assertEqual('[ ]', self.__bst.pre_order())
def test_remove_empty_postorder(self):
    with self.assertRaises(ValueError):
        self.__bst.remove_element(4)
    self.assertEqual('[ ]', self.__bst.post_order())
def test_remove_empty_height(self):
    with self.assertRaises(ValueError):
        self.__bst.remove_element(4)
    self.assertEqual(0, self.__bst.get_height())
```

For an empty binary search tree, any remove is expected to raise an exception and leave the tree exactly unchanged, because there's no valid value to be removed.

```
def test_remove_leaving_empty_inorder(self):
…
def test_remove_one_error_preorder(self):
…
```

For a tree of one element, remove is straightforward, only involving two possibilities: successfully remove the tree to be an empty tree, or raise an exception and leave the tree unchanged.

```
def test_remove_leaf_leaving_one_inorder(self):
…
def test_remove_root_leaving_one_inorder(self):
…
def test_remove_two_error_inorder(self):
```

Here comes the distinction of different removes: for *remove_leaf*, the remove is done on a node with no children, and for *remove_root*, it is done on a node with one child, and there's also the possibility of error and leave the tree unchanged.

```
def test_remove_root_leaving_two_inorder(self):
…
def test_remove_leaf_leaving_two_inorder(self):
…
def test_remove_three_error_inorder(self):
```

For a tree of three elements, there is the first cases where the removed node has two children, which I tested in *remove_node*. Then I just follow this testing paradigm to the binary search tree of four elements and ensure that all three different removes are tested.

Overall, great job with this writeup! Things to improve are just making sure not to bloat your writeup with source code since you already provide that in your other files it is redundant.

```python
class Binary_Search_Tree:
  # TODO.I have provided the public method skeletons. You will need
  # to add private methods to support the recursive algorithms
  # discussed in class
```

Remove TODO comments

```python
  class __BST_Node:
    # TODO The Node class is private. You may add any attributes and
    # methods you need. Recall that attributes in an inner class
    # must be public to be reachable from the the methods.
    __slots__ = 'value', 'left', 'right', 'height'
    def __init__(self, value):
      self.value = value
      self.left = None
      self.right = None
      self.height = 1

    def update_height(self):
    #update the node's height by adding one to the bigger height of its
children if both are not None
    #if left is None, right is not None, then add one to the right's height
    #if right is None, left is not None, then add one to the left's height
    #if both are None, do nothing.
      if self.left is None and self.right is not None:
        self.height = self.right.height + 1
      elif self.right is None and self.left is not None:
        self.height = self.left.height + 1
      elif self.right is not None and self.left is not None:
        if self.right.height > self.left.height:
          self.height = self.right.height + 1
        else:
          self.height = self.left.height + 1
      else:
        self.height = 1

  def __init__(self):
    self.__root = None

  def insert_element(self, value):
    # Insert the value specified into the tree at the correct
    # location based on "less is left; greater is right" binary
    # search tree ordering. If the value is already contained in
    # the tree, raise a ValueError. Your solution must be recursive.
    # This will involve the introduction of additional private
    # methods to support the recursion control variable.
      self.__root = self.__recursive_insert(value, self.__root)  #might
raise an error
```

```
  def __recursive_insert(self, value, node):
    if node is None:
      return self.__BST_Node(value)  #base case, return a newly created node
    elif value == node.value:
      raise ValueError('Value Exist')
    elif value < node.value:
      node.left = self.__recursive_insert(value, node.left)  #update the left subtree to insert the value
      node.update_height()
      return node
    else:
      node.right = self.__recursive_insert(value, node.right)  #update the right subtree to insert the value
      node.update_height()
      return node

  def remove_element(self, value):
    # Remove the value specified from the tree, raising a ValueError
    # if the value isn't found. When a replacement value is necessary,
    # select the minimum value to the from the right as this element's
    # replacement. Take note of when to move a node reference and when
    # to replace the value in a node instead. It is not necessary to
    # return the value (though it would reasonable to do so in some
    # implementations). Your solution must be recursive.
    # This will involve the introduction of additional private
    # methods to support the recursion control variable.
    self.__root = self.__recursive_remove(value, self.__root)

  def __recursive_remove(self, value, node):
    if node is None:
      raise ValueError('Value does not exist')
    elif value > node.value:
      node.right = self.__recursive_remove(value, node.right)
      node.update_height()
      return node
    elif value < node.value:
      node.left = self.__recursive_remove(value, node.left)
      node.update_height()
      return node
    else:  #find the node, begin removing
      if node.left is None and node.right is None:
        return None
      elif node.left is not None and node.right is None:
        return node.left
```

```
      elif node.left is None and node.right is not None:
        return node.right
      else:
        current_node = node.right  #locating the leftmost node in the 
right subtree
        while current_node.left is not None:
          current_node = current_node.left
        node.value = current_node.value
        node.right = self.__recursive_remove(current_node.value, 
node.right) #delete the current_node whose value has been duplicated
        node.update_height()
        return node

  def in_order(self):
    # Construct and return a string representing the in-order
    # traversal of the tree. Empty trees should be printed as [ ].
    # Trees with one value should be printed as [ 4 ]. Trees with more
    # than one value should be printed as [ 4, 7 ]. Note the spacing.
    # Your solution must be recursive. This will involve the introduction
    # of additional private methods to support the recursion control
    # variable.
    if self.__root is None:
      result = '[ ]'
    else:
      result = '[ '
      result = result + self.__recursive_inorder(self.__root)[:-2]
      result = result + ' ]'
    return result

  def __recursive_inorder(self, node):
    result = ''
    if node.left is not None:
      result = result + self.__recursive_inorder(node.left)
    result = result + str(node.value) + ', '
    if node.right is not None:
      result = result + self.__recursive_inorder(node.right)
    return result

  def pre_order(self):
    # Construct and return a string representing the pre-order
    # traversal of the tree. Empty trees should be printed as [ ].
    # Trees with one value should be printed in as [ 4 ]. Trees with
    # more than one value should be printed as [ 4, 7 ]. Note the spacing.
    # Your solution must be recursive. This will involve the introduction
    # of additional private methods to support the recursion control
    # variable.
```

```
    if self.__root is None:
      result = '[ ]'
    else:
      result = '[ '
      result = result + self.__recursive_preorder(self.__root)[:-2]
      result = result + ' ]'
    return result

  def __recursive_preorder(self, node):
    result = ''
    result = result + str(node.value) + ', '
    if node.left is not None:
        result = result + self.__recursive_preorder(node.left)
    if node.right is not None:
        result = result + self.__recursive_preorder(node.right)
    return result

  def post_order(self):
    # Construct an return a string representing the post-order
    # traversal of the tree. Empty trees should be printed as [ ].
    # Trees with one value should be printed in as [ 4 ]. Trees with
    # more than one value should be printed as [ 4, 7 ]. Note the spacing.
    # Your solution must be recursive. This will involve the introduction
    # of additional private methods to support the recursion control
    # variable.
    if self.__root is None:
      result = '[ ]'
    else:
      result = '[ '
      result = result + self.__recursive_postorder(self.__root)[:-2]
      result = result + ' ]'
    return result

  def __recursive_postorder(self, node):
    result = ''
    if node.left is not None:
        result = result + self.__recursive_postorder(node.left)
    if node.right is not None:
        result = result + self.__recursive_postorder(node.right)
    result = result + str(node.value) + ', '
    return result

  def get_height(self):
    # return an integer that represents the height of the tree.
    # assume that an empty tree has height 0 and a tree with one
    # node has height 1. This method must operate in constant time.
```

```python
    if self.__root is None:
      return 0
    else:
      return self.__root.height

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

if __name__ == '__main__':
  test_tree = Binary_Search_Tree()
  test_tree.insert_element(5)
  test_tree.insert_element(1)
  test_tree.insert_element(2)
  test_tree.insert_element(6)
  test_tree.insert_element(4)
  test_tree.insert_element(0)
  test_tree.remove_element(1)
  print(test_tree.in_order())
  print(test_tree.pre_order())
  print(test_tree.get_height())
  print(test_tree.post_order())
  print(str(test_tree))
  test_tree2 = Binary_Search_Tree()

  try:
    test_tree2.remove_element(2)
  except ValueError:
    print('successpython3 Binary_Search_Tree.py')
```

```
import unittest
from Binary_Search_Tree import Binary_Search_Tree

class BSTTester(unittest.TestCase):

	#test cases are built through three dimensions:
	#Through the x-axis, cases are built to test the functionality of update methods, insert and remove (Error)
	#Through the y-axis, the testing of functionality is constructed in terms of in_order, pre_order, post_order and height
	#Through the z-axis, the size of the tested tree are increasing from empty to 4.

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

	#Four element of empty tree
	def test_empty_tree_string_inorder(self):
		#test the in_order format of an empty tree
		self.assertEqual('[ ]', str(self.__bst))

	def test_empty_tree_preorder(self):
		self.assertEqual('[ ]', self.__bst.pre_order())

	def test_empty_tree_post_order(self):
		self.assertEqual('[ ]', self.__bst.post_order())

	def test_empty_tree_height(self):
		self.assertEqual(0, self.__bst.get_height())

	#Four element of one_element tree
	def test_insert_one_inorder(self):
		self.__bst.insert_element(5)
		self.assertEqual('[ 5 ]', str(self.__bst))

	def test_insert_one_preorder(self):
		self.__bst.insert_element(5)
		self.assertEqual('[ 5 ]', self.__bst.pre_order())

	def test_insert_one_postorder(self):
		self.__bst.insert_element(5)
		self.assertEqual('[ 5 ]', self.__bst.post_order())

	def test_insert_one_height(self):
		self.__bst.insert_element(5)
		self.assertEqual(1, self.__bst.get_height())
```

```python
#Four element of two_element tree, with possibility of ValueError
def test_insert_two_left_inorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(3)
      self.assertEqual('[ 3, 5 ]', str(self.__bst))

def test_insert_two_left_preorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(3)
      self.assertEqual('[ 5, 3 ]', self.__bst.pre_order())

def test_insert_two_left_postorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(3)
      self.assertEqual('[ 3, 5 ]', self.__bst.post_order())

def test_insert_two_left_height(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(3)
      self.assertEqual(2, self.__bst.get_height())

def test_insert_two_right_inorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.assertEqual('[ 5, 8 ]', str(self.__bst))

def test_insert_two_right_preorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.assertEqual('[ 5, 8 ]', self.__bst.pre_order())

def test_insert_two_right_postorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.assertEqual('[ 8, 5 ]', self.__bst.post_order())

def test_insert_two_right_height(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.assertEqual(2, self.__bst.get_height())

def test_insert_two_error_inorder(self):
      #should be ValueError, value exists
      self.__bst.insert_element(5)
      with self.assertRaises(ValueError):
```

```python
            self.__bst.insert_element(5)
        self.assertEqual('[ 5 ]', str(self.__bst))

    def test_insert_two_error_preorder(self):
        #should be ValueError, value exists
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(5)
        self.assertEqual('[ 5 ]', self.__bst.pre_order())

    def test_insert_two_error_postorder(self):
        #should be ValueError, value exists
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(5)
        self.assertEqual('[ 5 ]', self.__bst.post_order())

    def test_insert_two_error_height(self):
        #should be ValueError, value exists
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(5)
        self.assertEqual(1, self.__bst.get_height())

    #Four element of three_element tree with possibility of error
    #Test the insert works fine with a perfect tree with three element
    def test_insert_three_height_two_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.assertEqual('[ 3, 5, 8 ]', str(self.__bst))

    def test_insert_three_height_two_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.assertEqual('[ 5, 3, 8 ]', self.__bst.pre_order())

    def test_insert_three_height_two_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.assertEqual('[ 3, 8, 5 ]', self.__bst.post_order())

    def test_insert_three_height_two_height(self):
        self.__bst.insert_element(5)
```

```python
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.assertEqual(2, self.__bst.get_height())

    #Test that the insert works fine when the tree acts like a linked list
    def test_insert_three_height_three_left_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 2, 3, 5 ]', str(self.__bst))

    def test_insert_three_height_three_left_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 5, 3, 2 ]', self.__bst.pre_order())

    def test_insert_three_height_three_left_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 2, 3, 5 ]', self.__bst.post_order())

    def test_insert_three_height_three_left_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual(3, self.__bst.get_height())

    def test_insert_three_height_three_right_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(7)
        self.__bst.insert_element(9)
        self.assertEqual('[ 5, 7, 9 ]', str(self.__bst))

    def test_insert_three_height_three_right_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(7)
        self.__bst.insert_element(9)
        self.assertEqual('[ 5, 7, 9 ]', self.__bst.pre_order())

    def test_insert_three_height_three_right_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(7)
        self.__bst.insert_element(9)
        self.assertEqual('[ 9, 7, 5 ]', self.__bst.post_order())
```

```python
def test_insert_three_height_three_left_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(7)
    self.__bst.insert_element(9)
    self.assertEqual(3, self.__bst.get_height())

#Test that the insert works fine with a tree that goes left then right
def test_insert_three_height_three_left_right_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(4)
    self.assertEqual('[ 3, 4, 5 ]', str(self.__bst))

def test_insert_three_height_three_left_right_preorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(4)
    self.assertEqual('[ 5, 3, 4 ]', self.__bst.pre_order())

def test_insert_three_height_three_left_right_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(4)
    self.assertEqual('[ 4, 3, 5 ]', self.__bst.post_order())

def test_insert_three_height_three_left_right_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(4)
    self.assertEqual(3, self.__bst.get_height())

#Test the cases of error
def test_insert_three_error_inroder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual('[ 3, 5 ]', str(self.__bst))

def test_insert_three_error_preroder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual('[ 5, 3 ]', self.__bst.pre_order())
```

```python
def test_insert_three_error_postroder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual('[ 3, 5 ]', self.__bst.post_order())

def test_insert_three_error_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual(2, self.__bst.get_height())

#Test the four element updated by insert with a four-element tree
def test_insert_four_height_three_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(2)
    self.assertEqual('[ 2, 3, 5, 8 ]', str(self.__bst))

def test_insert_four_height_three_preorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(2)
    self.assertEqual('[ 5, 3, 2, 8 ]', self.__bst.pre_order())

def test_insert_four_height_three_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(2)
    self.assertEqual('[ 2, 3, 8, 5 ]', self.__bst.post_order())

def test_insert_four_height_three_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(2)
    self.assertEqual(3, self.__bst.get_height())

def test_insert_four_height_four_inorder(self):
    self.__bst.insert_element(5)
```

```python
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.assertEqual('[ 2, 3, 4, 5 ]', str(self.__bst))

    def test_insert_four_height_four_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.assertEqual('[ 5, 2, 4, 3 ]', self.__bst.pre_order())

    def test_insert_four_height_four_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.assertEqual('[ 3, 4, 2, 5 ]', self.__bst.post_order())

    def test_insert_four_height_four_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.assertEqual(4, self.__bst.get_height())

    def test_insert_four_left_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 2, 3, 4, 5 ]', str(self.__bst))

    def test_insert_four_left_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 5, 4, 3, 2 ]', self.__bst.pre_order())

    def test_insert_four_left_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.assertEqual('[ 2, 3, 4, 5 ]', self.__bst.post_order())
```

```python
def test_insert_four_left_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(4)
    self.__bst.insert_element(3)
    self.__bst.insert_element(2)
    self.assertEqual(4, self.__bst.get_height())

def test_insert_four_error_height_two_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual('[ 3, 5, 8 ]', str(self.__bst))

def test_insert_four_error_height_two_preorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(8)
    self.assertEqual('[ 5, 3, 8 ]', self.__bst.pre_order())

def test_insert_four_error_height_two_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(5)
    self.assertEqual('[ 3, 8, 5 ]', self.__bst.post_order())

def test_insert_four_error_height_two_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
    self.assertEqual(2, self.__bst.get_height())

def test_insert_four_error_height_three_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(4)
    with self.assertRaises(ValueError):
        self.__bst.insert_element(3)
```

```python
        self.assertEqual('[ 3, 4, 5 ]', str(self.__bst))

    def test_insert_four_error_height_three_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(4)
        self.assertEqual('[ 5, 3, 4 ]', self.__bst.pre_order())

    def test_insert_four_error_height_three_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(5)
        self.assertEqual('[ 4, 3, 5 ]', self.__bst.post_order())

    def test_insert_four_error_height_three_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.insert_element(3)
        self.assertEqual(3, self.__bst.get_height())

    #Test Remove
    #Test remove on an empty tree
    def test_remove_empty_inorder(self):
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ ]', str(self.__bst))

    def test_remove_empty_preorder(self):
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ ]', self.__bst.pre_order())

    def test_remove_empty_postorder(self):
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ ]', self.__bst.post_order())

    def test_remove_empty_height(self):
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
```

```
        self.assertEqual(0, self.__bst.get_height())

    #Test remove on an one-element tree
    def test_remove_leaving_empty_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.remove_element(5)
        self.assertEqual('[ ]', str(self.__bst))

    def test_remove_leaving_empty_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.remove_element(5)
        self.assertEqual('[ ]', self.__bst.pre_order())

    def test_remove_leaving_empty_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.remove_element(5)
        self.assertEqual('[ ]', self.__bst.post_order())

    def test_remove_leaving_empty_height(self):
        self.__bst.insert_element(5)
        self.__bst.remove_element(5)
        self.assertEqual(0, self.__bst.get_height())

    def test_remove_one_error_inorder(self):
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(3)
        self.assertEqual('[ 5 ]', str(self.__bst))

    def test_remove_one_error_preorder(self):
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(3)
        self.assertEqual('[ 5 ]', self.__bst.pre_order())

    def test_remove_one_error_postorder(self):
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(3)
        self.assertEqual('[ 5 ]', self.__bst.post_order())

    def test_remove_one_error_height(self):
        self.__bst.insert_element(5)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(3)
        self.assertEqual(1, self.__bst.get_height())
```

```python
#Test remove on a two-element tree
def test_remove_leaf_leaving_one_inorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(8)
      self.assertEqual('[ 5 ]', str(self.__bst))

def test_remove_leaf_leaving_one_preorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(8)
      self.assertEqual('[ 5 ]', self.__bst.pre_order())

def test_remove_leaf_leaving_one_postorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(8)
      self.assertEqual('[ 5 ]', self.__bst.post_order())

def test_remove_leaf_leaving_one_height(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(8)
      self.assertEqual(1, self.__bst.get_height())

def test_remove_root_leaving_one_inorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(5)
      self.assertEqual('[ 8 ]', str(self.__bst))

def test_remove_root_leaving_one_preorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(5)
      self.assertEqual('[ 8 ]', self.__bst.pre_order())

def test_remove_root_leaving_one_postorder(self):
      self.__bst.insert_element(5)
      self.__bst.insert_element(8)
      self.__bst.remove_element(5)
      self.assertEqual('[ 8 ]', self.__bst.post_order())

def test_remove_root_leaving_one_height(self):
      self.__bst.insert_element(5)
```

```python
        self.__bst.insert_element(8)
        self.__bst.remove_element(5)
        self.assertEqual(1, self.__bst.get_height())

    def test_remove_two_error_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(12)
        self.assertEqual('[ 5, 8 ]', str(self.__bst))

    def test_remove_two_error_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(12)
        self.assertEqual('[ 5, 8 ]', self.__bst.pre_order())

    def test_remove_two_error_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(12)
        self.assertEqual('[ 8, 5 ]', self.__bst.post_order())

    def test_remove_two_error_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(12)
        self.assertEqual(2, self.__bst.get_height())

    #Test remove on a three-element tree
    def test_remove_root_leaving_two_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.remove_element(5)
        self.assertEqual('[ 3, 8 ]', str(self.__bst))

    def test_remove_root_leaving_two_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.remove_element(5)
        self.assertEqual('[ 8, 3 ]', self.__bst.pre_order())
```

```
def test_remove_root_leaving_two_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(5)
    self.assertEqual('[ 3, 8 ]', self.__bst.post_order())

def test_remove_root_leaving_two_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(5)
    self.assertEqual(2, self.__bst.get_height())

def test_remove_leaf_leaving_two_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(3)
    self.assertEqual('[ 5, 8 ]', str(self.__bst))

def test_remove_leaf_leaving_two_preorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(3)
    self.assertEqual('[ 5, 8 ]', self.__bst.pre_order())

def test_remove_leaf_leaving_two_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(3)
    self.assertEqual('[ 8, 5 ]', self.__bst.post_order())

def test_remove_leaf_leaving_two_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.remove_element(3)
    self.assertEqual(2, self.__bst.get_height())

def test_remove_three_error_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
```

```python
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ 3, 5, 8 ]', str(self.__bst))

    def test_remove_three_error_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ 5, 3, 8 ]', self.__bst.pre_order())

    def test_remove_three_error_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual('[ 3, 8, 5 ]', self.__bst.post_order())

    def test_remove_three_error_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(4)
        self.assertEqual(2, self.__bst.get_height())

    #Test remove on a four-element tree
    def test_remove_leaf_leaving_three_height_two_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(16)
        self.assertEqual('[ 3, 5, 8 ]', str(self.__bst))

    def test_remove_leaf_leaving_three_height_two_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(16)
        self.assertEqual('[ 5, 3, 8 ]', self.__bst.pre_order())
```

```python
def test_remove_leaf_leaving_three_height_two_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(16)
    self.assertEqual('[ 3, 8, 5 ]', self.__bst.post_order())

def test_remove_leaf_leaving_three_height_two_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(16)
    self.assertEqual(2, self.__bst.get_height())

def test_remove_inner_leaving_three_height_two_inorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(8)
    self.assertEqual('[ 3, 5, 16 ]', str(self.__bst))

def test_remove_inner_leaving_three_height_two_preorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(8)
    self.assertEqual('[ 5, 3, 16 ]', self.__bst.pre_order())

def test_remove_inner_leaving_three_height_two_postorder(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(8)
    self.assertEqual('[ 3, 16, 5 ]', self.__bst.post_order())

def test_remove_inner_leaving_three_height_two_height(self):
    self.__bst.insert_element(5)
    self.__bst.insert_element(3)
    self.__bst.insert_element(8)
    self.__bst.insert_element(16)
    self.__bst.remove_element(8)
```

```python
        self.assertEqual(2, self.__bst.get_height())

    def test_remove_root_leaving_three_height_two_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(5)
        self.assertEqual('[ 3, 8, 16 ]', str(self.__bst))

    def test_remove_root_leaving_three_height_two_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(5)
        self.assertEqual('[ 8, 3, 16 ]', self.__bst.pre_order())

    def test_remove_root_leaving_three_height_two_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(5)
        self.assertEqual('[ 3, 16, 8 ]', self.__bst.post_order())

    def test_remove_root_leaving_three_height_two_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(16)
        self.__bst.remove_element(5)
        self.assertEqual(2, self.__bst.get_height())

    def test_remove_leaf_leaving_three_height_three_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.remove_element(2)
        self.assertEqual('[ 3, 4, 5 ]', str(self.__bst))

    def test_remove_leaf_leaving_three_height_three_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
```

```
        self.__bst.insert_element(2)
        self.__bst.remove_element(2)
        self.assertEqual('[ 5, 4, 3 ]', self.__bst.pre_order())

    def test_remove_leaf_leaving_three_height_three_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.remove_element(2)
        self.assertEqual('[ 3, 4, 5 ]', self.__bst.post_order())

    def test_remove_leaf_leaving_three_height_three_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(4)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.remove_element(2)
        self.assertEqual(3, self.__bst.get_height())

    def test_remove_inner_leaving_three_height_three_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.remove_element(3)
        self.assertEqual('[ 2, 4, 5 ]', str(self.__bst))

    def test_remove_inner_leaving_three_height_three_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.remove_element(3)
        self.assertEqual('[ 5, 4, 2 ]', self.__bst.pre_order())

    def test_remove_inner_leaving_three_height_three_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.remove_element(3)
        self.assertEqual('[ 2, 4, 5 ]', self.__bst.post_order())

    def test_remove_inner_leaving_three_height_three_height(self):
        self.__bst.insert_element(5)
```

```
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        self.__bst.remove_element(3)
        self.assertEqual(3, self.__bst.get_height())

    def test_remove_four_error_height_three_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(8)
        self.assertEqual('[ 2, 3, 4, 5 ]', str(self.__bst))

    def test_remove_four_error_height_three_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(8)
        self.assertEqual('[ 5, 3, 2, 4 ]', self.__bst.pre_order())

    def test_remove_four_error_height_three_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(8)
        self.assertEqual('[ 2, 4, 3, 5 ]', self.__bst.post_order())

    def test_remove_four_error_height_three_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(2)
        self.__bst.insert_element(4)
        with self.assertRaises(ValueError):
            self.__bst.remove_element(8)
        self.assertEqual(3, self.__bst.get_height())

    def test_remove_root_leaving_four_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
```

```python
        self.__bst.insert_element(6)
        self.__bst.insert_element(10)
        self.__bst.remove_element(5)
        self.assertEqual('[ 3, 6, 8, 10 ]', str(self.__bst))

    def test_remove_root_leaving_four_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(10)
        self.__bst.remove_element(5)
        self.assertEqual('[ 6, 3, 8, 10 ]', self.__bst.pre_order())

    def test_remove_root_leaving_four_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(10)
        self.__bst.remove_element(5)
        self.assertEqual('[ 3, 10, 8, 6 ]', self.__bst.post_order())

    def test_remove_root_leaving_four_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(10)
        self.__bst.remove_element(5)
        self.assertEqual(3, self.__bst.get_height())

    def test_remove_inner_leaving_four_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(6)
        self.assertEqual('[ 3, 5, 7, 8 ]', str(self.__bst))

    def test_remove_inner_leaving_four_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
```

```
        self.__bst.insert_element(7)
        self.__bst.remove_element(6)
        self.assertEqual('[ 5, 3, 8, 7 ]', self.__bst.pre_order())

    def test_remove_inner_leaving_four_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(6)
        self.assertEqual('[ 3, 7, 8, 5 ]', self.__bst.post_order())

    def test_remove_inner_leaving_four_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(6)
        self.assertEqual(3, self.__bst.get_height())

    def test_remove_leaf_leaving_four_inorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(7)
        self.assertEqual('[ 3, 5, 6, 8 ]', str(self.__bst))

    def test_remove_leaf_leaving_four_preorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(7)
        self.assertEqual('[ 5, 3, 8, 6 ]', self.__bst.pre_order())

    def test_remove_leaf_leaving_four_postorder(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
```

```python
        self.__bst.remove_element(7)
        self.assertEqual('[ 3, 6, 8, 5 ]', self.__bst.post_order())

    def test_remove_leaf_leaving_four_height(self):
        self.__bst.insert_element(5)
        self.__bst.insert_element(3)
        self.__bst.insert_element(8)
        self.__bst.insert_element(6)
        self.__bst.insert_element(7)
        self.__bst.remove_element(7)
        self.assertEqual(3, self.__bst.get_height())



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

```
test_complex_left_remove_leaf_height (__main__.BSTTester) ... ok
test_complex_left_remove_leaf_str (__main__.BSTTester) ...
ok
test_complex_left_remove_leaf_traversals (__main__.BSTTester) ...
ok
test_complex_left_remove_right_child_height (__main__.BSTTester) ...
ok
test_complex_left_remove_right_child_str (__main__.BSTTester) ...
ok
test_complex_left_remove_right_child_traversals (__main__.BSTTester) ...
ok
test_complex_right_remove_leaf_height (__main__.BSTTester) ...
ok
test_complex_right_remove_leaf_str (__main__.BSTTester) ...
ok
test_complex_right_remove_leaf_traversals (__main__.BSTTester) ...
ok
test_complex_right_remove_right_height (__main__.BSTTester) ...
ok
test_complex_right_remove_right_str (__main__.BSTTester) ...
ok
test_complex_right_remove_right_traversals (__main__.BSTTester) ...
ok
test_empty_height (__main__.BSTTester) ...
ok
test_empty_str (__main__.BSTTester) ...
ok
test_empty_traversals (__main__.BSTTester) ...
ok
test_four_height (__main__.BSTTester) ...
ok
test_four_str (__main__.BSTTester) ...
ok
test_four_traversals (__main__.BSTTester) ...
ok
test_one_duplicate_root (__main__.BSTTester) ...
ok
test_one_height (__main__.BSTTester) ...
ok
test_one_remove_missing (__main__.BSTTester) ...
ok
test_one_remove_root_height (__main__.BSTTester) ...
ok
test_one_remove_root_str (__main__.BSTTester) ...
ok
test_one_remove_root_traversals (__main__.BSTTester) ...
ok
test_one_str (__main__.BSTTester) ...
ok
test_one_traversals (__main__.BSTTester) ...
ok
test_three_duplicate_leaf (__main__.BSTTester) ...
ok
test_three_left_height (__main__.BSTTester) ...
```

```
ok
test_three_left_remove_middle_height (__main__.BSTTester) ...
ok
test_three_left_remove_middle_str (__main__.BSTTester) ...
ok
test_three_left_remove_middle_traversals (__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_remove_root_height (__main__.BSTTester) ...
ok
test_three_perfect_remove_root_str (__main__.BSTTester) ...
ok
test_three_perfect_remove_root_traversals (__main__.BSTTester) ...
ok
test_three_perfect_str (__main__.BSTTester) ...
ok
test_three_perfect_traversals (__main__.BSTTester) ...
ok
test_three_remove_missing (__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_remove_left_height (__main__.BSTTester) ...
ok
test_two_remove_left_str (__main__.BSTTester) ...
ok
test_two_remove_left_traversals (__main__.BSTTester) ...
ok
test_two_remove_right_height (__main__.BSTTester) ...
ok
test_two_remove_right_str (__main__.BSTTester) ...
ok
test_two_remove_right_traversals (__main__.BSTTester) ...
ok
test_two_remove_root_height (__main__.BSTTester) ...
ok
test_two_remove_root_str (__main__.BSTTester) ...
ok
test_two_remove_root_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

----------------------------------------------------------------------
Ran 58 tests in 0.004s

OK
```