Worst-case Performance Analysis of Methods

Class __BST_Node:

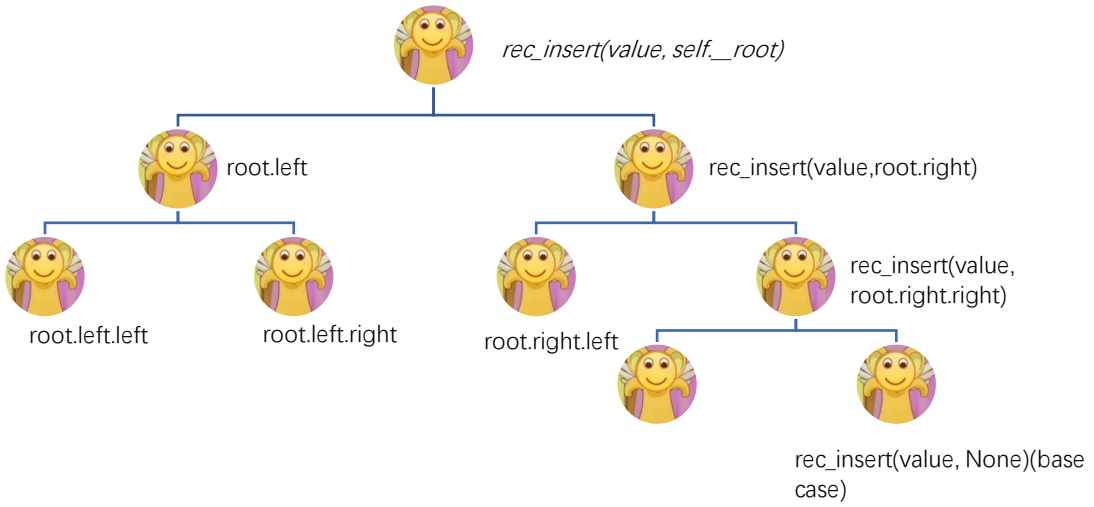
__init__():
The constructer of my *__BST_Node* class that creates a node with a given value has constant time worst-case performance, *O(1)*. What's done by the constructer is to set four spots in the memory, one to store the value given as the parameter and reference to that value by attribute *value;* 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():
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.



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)*

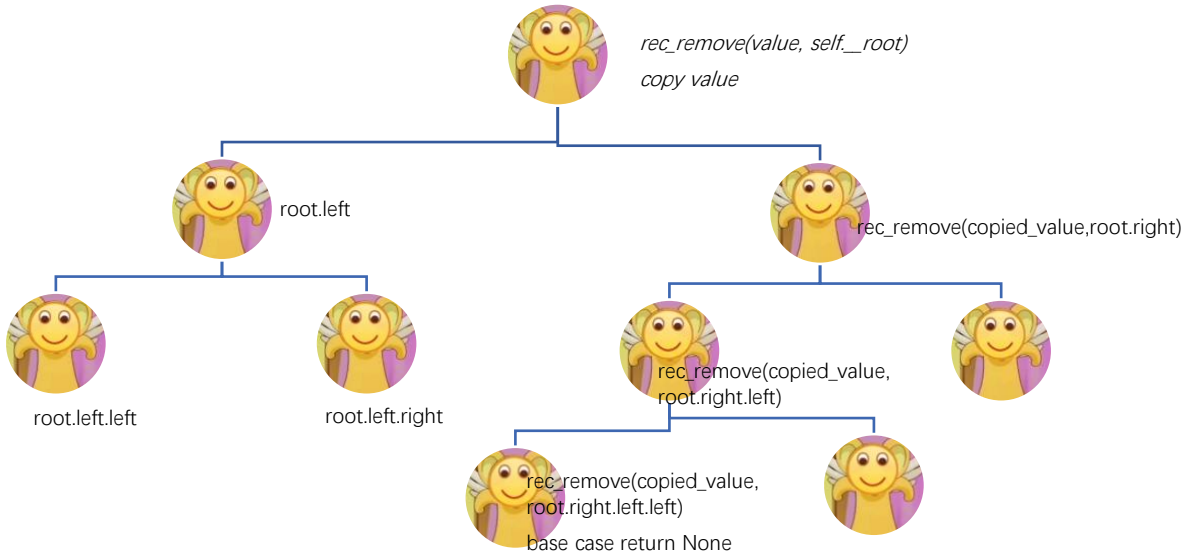
= 2 since a balanced tree has to have at least one element to be height 1 and at least 2 element to be height 2. Then, for *h >=3,* it can be observed that *n(h) = 1 + n(h-1) + n(h-2).*
Then, since *n(h-1) > n(h-2),* it follows that *n(h) > 2n(h-2) > 4n(h-4) > ... >* $2^i n(h-2i)$ where *i = |h/2| +1* 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):

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)*.

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:

```
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.