

Find solutions to your homework

Search

## Question

Implement the add() and remove() methods for an AVL Tree in Python. All needed classes, requirements, skeleton code, and examples are provided below:

1. When removing a node, replace it with the left most child of the right subtree(aka in-order successor). You do not need to 'recursively' continue this process. If the deleted node only has one subtree (either right or left), replace the deleted node with the root node of that subtree.
2. Variables in TreeNode and AVL classes are not private. You are allowed to access and change their values directly. You do not need to write any getter or setter methods for them.
3. RESTRICTIONS: You are not allowed to use ANY built-in Python data structures and/or their methods. In case you need 'helper' data structures in your solution, skeleton code includes prewritten implementation of Queue and Stack classes. You are allowed to create and use objects from those classes in your implementation.
4. You are not allowed to directly access any variables of the Queue or Stack classes. All work must be done only by using class methods.

```

add(self, value obj): -> None:
    """
    This method adds new value to the tree, maintaining AVL property. It returns None.
    """
    # ... (code for add method) ...

remove(self, value obj): -> None:
    """
    This method removes the node with the value of the value in the AVL tree. The method must
    return None. If the value is not found in the tree, the method should return None.
    """
    # ... (code for remove method) ...

class AVL:
    """
    AVL Tree class
    """
    # ... (code for AVL class) ...

class Stack:
    """
    Stack class
    """
    # ... (code for Stack class) ...

class Queue:
    """
    Queue class
    """
    # ... (code for Queue class) ...
    
```

import random

class Stack:

Class implementing STACK ADT.  
Supported methods are: push, pop, top, is\_empty

DO NOT CHANGE THIS CLASS IN ANY WAY  
YOU ARE ALLOWED TO CREATE AND USE OBJECTS OF THIS CLASS IN

## Post a question

Answers from our experts for your tough homework questions

20 questions left - Renews Aug. 12, 2022

Enter question

Continue to post

**Chegg**

**Writing lab closed?**

Get plagiarism checks, expert paper help & more

Use Chegg Writing

## My Textbook Solutions

**Fundamental...**

Solutions →

**Fundamental...**

Solutions →

View all solutions

**Chegg**

**Writing lab**

## YOUR SOLUTION

"""

```
def __init__(self):
    """ Initialize empty stack based on Python list """
    self._data = []

def push(self, value: object) -> None:
    """ Add new element on top of the stack """
    self._data.append(value)

def pop(self):
    """ Remove element from top of the stack and return its value """
    return self._data.pop()

def top(self):
    """ Return value of top element without removing from stack """
    return self._data[-1]

def is_empty(self):
    """ Return True if the stack is empty, return False otherwise """
    return len(self._data) == 0

def __str__(self):
    """ Return content of the stack as a string (for use with print) """
    data_str = [str(i) for i in self._data]
    return "STACK: { " + " , ".join(data_str) + " }"
```

class Queue:

"""

Class implementing QUEUE ADT.

Supported methods are: enqueue, dequeue, is\_empty

DO NOT CHANGE THIS CLASS IN ANY WAY

YOU ARE ALLOWED TO CREATE AND USE OBJECTS OF THIS CLASS IN  
YOUR SOLUTION

"""

```
def __init__(self):
    """ Initialize empty queue based on Python list """
    self._data = []

def enqueue(self, value: object) -> None:
    """ Add new element to the end of the queue """
    self._data.append(value)

def dequeue(self):
    """ Remove element from the beginning of the queue and return its value """
    return self._data.pop(0)

def is_empty(self):
    """ Return True if the queue is empty, return False otherwise """
    return len(self._data) == 0

def __str__(self):
    """ Return content of the stack as a string (for use with print) """
    data_str = [str(i) for i in self._data]
    return "QUEUE { " + " , ".join(data_str) + " }"
```



**writing lab  
closed?**

Get plagiarism checks,  
expert paper help & more

[Use Chegg Writing](#)



**Chegg**

**Writing lab  
closed?**

Get plagiarism checks,  
expert paper help & more

[Use Chegg Writing](#)

```

class TreeNode:
    """
    AVL Tree Node class
    DO NOT CHANGE THIS CLASS IN ANY WAY
    """

    def __init__(self, value: object) -> None:
        """
        Initialize a new AVL node
        DO NOT CHANGE THIS METHOD IN ANY WAY
        """

        self.value = value
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0

    def __str__(self):
        return 'AVL Node: {}'.format(self.value)


class AVL:
    def __init__(self, start_tree=None) -> None:
        """
        Initialize a new AVL tree
        DO NOT CHANGE THIS METHOD IN ANY WAY
        """

        self.root = None

        # populate AVL with initial values (if provided)
        # before using this feature, implement add() method
        if start_tree is not None:
            for value in start_tree:
                self.add(value)

    def __str__(self) -> str:
        """
        Return content of AVL in human-readable form using pre-order traversal
        DO NOT CHANGE THIS METHOD IN ANY WAY
        """

        values = []
        self._str_helper(self.root, values)
        return "AVL pre-order { " + ", ".join(values) + " }"

    def _str_helper(self, cur, values):
        """
        Helper method for __str__. Does pre-order tree traversal
        DO NOT CHANGE THIS METHOD IN ANY WAY
        """

        if cur:
            values.append(str(cur.value))
            self._str_helper(cur.left, values)
            self._str_helper(cur.right, values)

    def is_valid_avl(self) -> bool:

```

```
"""
```

Perform pre-order traversal of the tree. Return False if there are any problems with attributes of any of the nodes in the tree.

This is intended to be a troubleshooting 'helper' method to help find any inconsistencies in the tree after the add() or remove() operations. Review the code to understand what this method is checking and how it determines whether the AVL tree is correct.

DO NOT CHANGE THIS METHOD IN ANY WAY

```
"""
```

```
s = Stack()
s.push(self.root)
while not s.is_empty():
    node = s.pop()
    if node:
        # check for correct height (relative to children)
        l = node.left.height if node.left else -1
        r = node.right.height if node.right else -1
        if node.height != 1 + max(l, r):
            return False

    if node.parent:
        # parent and child pointers are in sync
        if node.value < node.parent.value:
            check_node = node.parent.left
        else:
            check_node = node.parent.right
        if check_node != node:
            return False
    else:
        # NULL parent is only allowed on the root of the tree
        if node != self.root:
            return False
    s.push(node.right)
    s.push(node.left)
return True
```

```
# -----
```

```
def add(self, value: object) -> None:
```

```
"""
```

TODO: Write your implementation

```
"""
```

```
pass
```

```
def remove(self, value: object) -> bool:
```

```
"""
```

TODO: Write your implementation

```
"""
```

```
pass
```

[Show transcribed data](#)

## Expert Answer



Anonymous answered this

90 answers

```
# Python code to delete a node in AVL tree
# Generic tree node class
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

# AVL tree class which supports insertion,
# deletion operations
class AVL_Tree(object):

    def insert(self, root, key):

        # Step 1 - Perform normal BST
        if not root:
            return TreeNode(key)
        elif key < root.val:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                               self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    # Recursive function to delete a node with
    # given key from subtree with given root.
    # It returns root of the modified subtree.
    def delete(self, root, key):

        # Step 1 - Perform standard BST delete
```

```

        if not root:
            return root

        elif key < root.val:
            root.left = self.delete(root.left, key)

        elif key > root.val:
            root.right = self.delete(root.right, key)

        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp

            elif root.right is None:
                temp = root.left
                root = None
                return temp

            temp = self.getMinValueNode(root.right)
            root.val = temp.val
            root.right = self.delete(root.right,
temp.val)

        # If the tree has only one node,
        # simply return it
        if root is None:
            return root

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and self.getBalance(root.left) >= 0:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and self.getBalance(root.right) <= 0:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and self.getBalance(root.left) < 0:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and self.getBalance(root.right) > 0:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    def leftRotate(self, z):

        y = z.right
        T2 = y.left

```

```

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
self.getHeight(y.right))

        # Return the new root
        return y

    def rightRotate(self, z):

        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
self.getHeight(y.right))

        # Return the new root
        return y

    def getHeight(self, root):
        if not root:
            return 0

        return root.height

    def getBalance(self, root):
        if not root:
            return 0

        return self.getHeight(root.left) -
self.getHeight(root.right)

    def getMinValueNode(self, root):
        if root is None or root.left is None:
            return root

        return self.getMinValueNode(root.left)

    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.val), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)

```

```
myTree = AVL_Tree()
```

```

root = None
nums = [9, 5, 10, 0, 6, 11, -1, 1, 2]

for num in nums:
    root = myTree.insert(root, num)

# Preorder Traversal
print("Preorder Traversal after insertion -")
myTree.preOrder(root)
print()

# Delete
key = 10
root = myTree.delete(root, key)

# Preorder Traversal
print("Preorder Traversal after deletion -")
myTree.preOrder(root)
print()

```

```

1  # Python code to delete a node in AVL tree
2  # Generic tree node class
3  class TreeNode(object):
4      def __init__(self, val):
5          self.val = val
6          self.left = None
7          self.right = None
8          self.height = 1
9
10 # AVL tree class which supports insertion,
11 # deletion operations
12 class AVL_Tree(object):
13
14     def insert(self, root, key):
15
16         # Step 1 - Perform normal BST
17         if not root:
18             return TreeNode(key)
19         elif key < root.val:
20             root.left = self.insert(root.left, key)
21         else:
22             root.right = self.insert(root.right, key)
23
24         # Step 2 - Update the height of the
25         # ancestor node
26         root.height = 1 + max(self.getHeight(root.left),
27                               self.getHeight(root.right))
28
29         # Step 3 - Get the balance factor
30         balance = self.getBalance(root)
31
32         # Step 4 - If the node is unbalanced,
33         # then try out the 4 cases
34         # Case 1 - Left Left
35         if balance > 1 and key < root.left.val:
36             return self.rightRotate(root)
37
38         # Case 2 - Right Right
39         if balance < -1 and key > root.right.val:
40             return self.leftRotate(root)
41
42

```

```

41
42     # Case 3 - Left Right
43     if balance > 1 and key > root.left.val:

```



```

44         root.left = self.leftRotate(root.left)
45         return self.rightRotate(root)
46
47     # Case 4 - Right Left
48     if balance < -1 and key < root.right.val:
49         root.right = self.rightRotate(root.right)
50         return self.leftRotate(root)
51
52     return root
53
54     # Recursive function to delete a node with
55     # given key from subtree with given root.
56     # It returns root of the modified subtree.
57     def delete(self, root, key):
58
59         # Step 1 - Perform standard BST delete
60         if not root:
61             return root
62
63         elif key < root.val:
64             root.left = self.delete(root.left, key)
65
66         elif key > root.val:
67             root.right = self.delete(root.right, key)
68
69         else:
70             if root.left is None:
71                 temp = root.right
72                 root = None
73                 return temp
74
75             elif root.right is None:
76                 temp = root.left
77                 root = None
78                 return temp
79
80             temp = self.getMinValueNode(root.right)
81             root.val = temp.val

```

```

79
80         temp = self.getMinValueNode(root.right)
81         root.val = temp.val
82         root.right = self.delete(root.right,
83                                 temp.val)
84
85         # If the tree has only one node,
86         # simply return it
87         if root is None:
88             return root
89
90         # Step 2 - Update the height of the
91         # ancestor node
92         root.height = 1 + max(self.getHeight(root.left),
93                               self.getHeight(root.right))
94
95         # Step 3 - Get the balance factor
96         balance = self.getBalance(root)
97
98         # Step 4 - If the node is unbalanced,
99         # then try out the 4 cases
100        # Case 1 - Left Left
101        if balance > 1 and self.getBalance(root.left) >= 0:
102            return self.rightRotate(root)
103
104        # Case 2 - Right Right

```

```

104         # Case 2 - Right Right
105         if balance < -1 and self.getBalance(root.right) <= 0:
106             return self.leftRotate(root)
107
108         # Case 3 - Left Right
109         if balance > 1 and self.getBalance(root.left) < 0:
110             root.left = self.leftRotate(root.left)
111             return self.rightRotate(root)
112
113         # Case 4 - Right Left
114         if balance < -1 and self.getBalance(root.right) > 0:
115             root.right = self.rightRotate(root.right)
116             return self.leftRotate(root)
117         return root
118

```

```

119     def leftRotate(self, z):
120
121         y = z.right
122         T2 = y.left
123
124         # Perform rotation
125         y.left = z
126         z.right = T2
127
128         # Update heights
129         z.height = 1 + max(self.getHeight(z.left),
130                           self.getHeight(z.right))
131         y.height = 1 + max(self.getHeight(y.left),
132                           self.getHeight(y.right))
133
134         # Return the new root
135         return y
136
137     def rightRotate(self, z):
138
139         y = z.left
140         T3 = y.right
141
142         # Perform rotation
143         y.right = z
144         z.left = T3
145
146         # Update heights
147         z.height = 1 + max(self.getHeight(z.left),
148                           self.getHeight(z.right))
149         y.height = 1 + max(self.getHeight(y.left),
150                           self.getHeight(y.right))
151
152         # Return the new root
153         return y
154
155     def getHeight(self, root):
156         if not root:
157             return 0
158

```

```

160
161     def getBalance(self, root):
162         if not root:
163             return 0
164
165         return self.getHeight(root.left) - self.getHeight(root.right)
166
167     def getMinValueNode(self, root):
168         if root is None or root.left is None:

```

```

169         return root
170
171         return self.getMinValueNode(root.left)
172
173     def preOrder(self, root):
174
175         if not root:
176             return
177
178         print("{} ".format(root.val), end="")
179         self.preOrder(root.left)
180         self.preOrder(root.right)
181
182
183 myTree = AVL_Tree()
184 root = None
185 nums = [9, 5, 10, 0, 6, 11, -1, 1, 2]
186
187 for num in nums:
188     root = myTree.insert(root, num)
189
190 # Preorder Traversal
191 print("Preorder Traversal after insertion -")
192 myTree.preOrder(root)
193 print()
194
195 # Delete
196 key = 10
197 root = myTree.delete(root, key)
198
199 # Preorder Traversal

```

```

190 # Preorder Traversal
191 print("Preorder Traversal after insertion -")
192 myTree.preOrder(root)
193 print()
194
195 # Delete
196 key = 10
197 root = myTree.delete(root, key)
198
199 # Preorder Traversal
200 print("Preorder Traversal after deletion -")
201 myTree.preOrder(root)
202 print()
203
204

```

Was this answer helpful?



1



1

## Practice with similar questions

**Q:** Implement the add() and remove() methods for an AVL Tree in Python. All needed classes, requirements, skeleton code, and examples are provided below: When removing a node, replace it with the left most child of the right subtree (aka in-order successor). You do not need to 'recursively' continue this process. If the deleted node only has one subtree (either right or left), replace the deleted node with the root node of that subtree. Variables in TreeNode...

**A:** [See answer](#)

## Questions viewed by other students

**Q:** PYTHON 3-- AVL tree- please do the add() function.Existing skeleton code cannot be altered, but helper methods can be added. NO BUILT-IN DATA STRUCTURES CAN BE USED. Instructions for add() are attached along with skeleton code. Please make sure that the given test code works.Skeleton Code:  

```
class Stack: """ Class implementing STACK ADT. Supported methods are: push, pop, top, is_empty DO NOT CHANGE THIS CLASS IN ANY WAY YOU ARE ALLOWED T...
```

**A:** [See answer](#)  100% (4 ratings)

**Q:** I need some help with the implementation of this AVL tree. I have tried to implement some of the functions but I don't know how to finish them and put everything together.  

```
class AVLTreeNode(TreeNode): """ AVL Tree Node class DO NOT CHANGE THIS CLASS IN ANY WAY """ def __init__(self, value: object) -> None: super().__init__(value) self.parent = None self.height = 0  
class AVL(BST): """ An AVL tree. Note: T...
```

**A:** [See answer](#)

[Show more](#) 

### COMPANY

About Chegg  
Chegg For Good  
College Marketing  
Corporate Development  
Investor Relations  
Jobs  
Join Our Affiliate Program  
Media Center  
Site Map

### LEGAL & POLICIES

Advertising Choices  
Cookie Notice  
General Policies  
Intellectual Property Rights  
Terms of Use  
Global Privacy Policy  
DO NOT SELL MY INFO  
Honor Code  
Honor Shield  
Academic Integrity

### CHEGG PRODUCTS AND SERVICES

Cheap Textbooks  
Chegg Coupon  
Chegg Play  
Chegg Study Help  
College Textbooks  
eTextbooks  
Flashcards  
Learn  
Uversity  
Chegg Math Solver  
Mobile Apps  
Solutions Manual  
Study 101  
Textbook Rental  
Used Textbooks  
Digital Access Codes  
Chegg Life  
Chegg Writing

### CHEGG NETWORK

Easybib  
Internships.com  
Thinkful  
Busuu  
Mathway

### CUSTOMER SERVICE

Customer Service  
Give Us Feedback  
Manage Subscription



© 2003-2022 Chegg Inc. All rights reserved.

