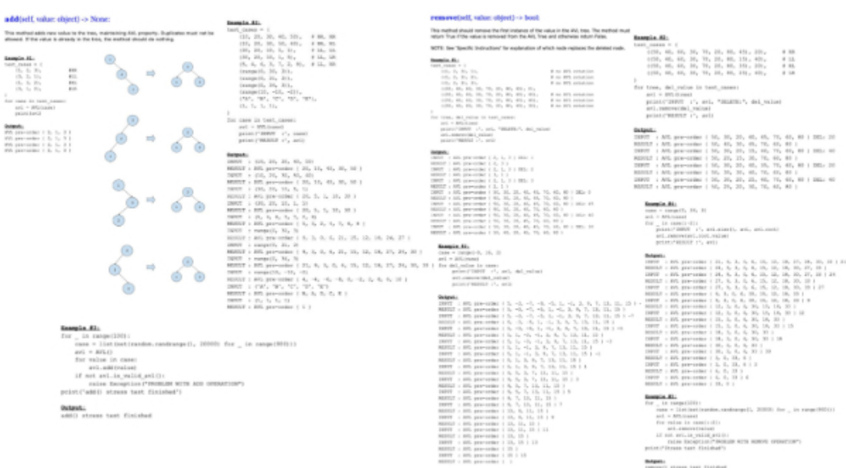
Question

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.



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

Chegg

```
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) + " }"
```

```
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()
```

```python
# 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:
```

```python
        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
```

```python

            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
```

```
        # 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
```

```

    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:
```

```python
        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
```


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

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

# Preorder Traversal
```


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


Practice with similar questions

Q: Implement the add() and remove() methods for an AVL Tree inPython. All needed classes, requirements, skeleton code, andexamples are provided below: When removing a node, replace it with the left most child of theright subtree(aka in-order successor). You do not need to'recursively' continue this process. If the deleted node only hasone subtree (either right or left), replace the deleted node withthe 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 beadded. NO BUILT-IN DATA STRUCTURES CAN BE USED. Instructions foradd() are attached along with skeleton code. Please make sure thatthe 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 = 0class AVL(BST): """ An AVL tree. Note: T...

A: See answer

Show more