Binary Search Tree Sample Code

class TreeNode:

def \_\_init\_\_(self,key,val,left=None,right=None,parent=None):

self.key = key

self.payload = val

self.leftChild = left

self.rightChild = right

self.parent = parent

def hasLeftChild(self):

return self.leftChild

def hasRightChild(self):

return self.rightChild

def isLeftChild(self):

return self.parent and self.parent.leftChild == self

def isRightChild(self):

return self.parent and self.parent.rightChild == self

def isRoot(self):

return not self.parent

def isLeaf(self):

return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):

return self.rightChild or self.leftChild

def hasBothChildren(self):

return self.rightChild and self.leftChild

def replaceNodeData(self,key,value,lc,rc):

self.key = key

self.payload = value

self.leftChild = lc

self.rightChild = rc

if self.hasLeftChild():

self.leftChild.parent = self

if self.hasRightChild():

self.rightChild.parent = self

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

self.size = 0

def length(self):

return self.size

def \_\_len\_\_(self):

return self.size

def put(self,key,val):

if self.root:

self.\_put(key,val,self.root)

else:

self.root = TreeNode(key,val)

self.size = self.size + 1

def \_put(self,key,val,currentNode):

if key < currentNode.key:

if currentNode.hasLeftChild():

self.\_put(key,val,currentNode.leftChild)

else:

currentNode.leftChild = TreeNode(key,val,parent=currentNode)

else:

if currentNode.hasRightChild():

self.\_put(key,val,currentNode.rightChild)

else:

currentNode.rightChild = TreeNode(key,val,parent=currentNode)

def \_\_setitem\_\_(self,k,v):

self.put(k,v)

def get(self,key):

if self.root:

res = self.\_get(key,self.root)

if res:

return res.payload

else:

return None

else:

return None

def \_get(self,key,currentNode):

if not currentNode:

return None

elif currentNode.key == key:

return currentNode

elif key < currentNode.key:

return self.\_get(key,currentNode.leftChild)

else:

return self.\_get(key,currentNode.rightChild)

def \_\_getitem\_\_(self,key):

return self.get(key)

def \_\_contains\_\_(self,key):

if self.\_get(key,self.root):

return True

else:

return False

def delete(self,key):

if self.size > 1:

nodeToRemove = self.\_get(key,self.root)

if nodeToRemove:

self.remove(nodeToRemove)

self.size = self.size-1

else:

raise KeyError('Error, key not in tree')

elif self.size == 1 and self.root.key == key:

self.root = None

self.size = self.size - 1

else:

raise KeyError('Error, key not in tree')

def \_\_delitem\_\_(self,key):

self.delete(key)

def spliceOut(self):

if self.isLeaf():

if self.isLeftChild():

self.parent.leftChild = None

else:

self.parent.rightChild = None

elif self.hasAnyChildren():

if self.hasLeftChild():

if self.isLeftChild():

self.parent.leftChild = self.leftChild

else:

self.parent.rightChild = self.leftChild

self.leftChild.parent = self.parent

else:

if self.isLeftChild():

self.parent.leftChild = self.rightChild

else:

self.parent.rightChild = self.rightChild

self.rightChild.parent = self.parent

def findSuccessor(self):

succ = None

if self.hasRightChild():

succ = self.rightChild.findMin()

else:

if self.parent:

if self.isLeftChild():

succ = self.parent

else:

self.parent.rightChild = None

succ = self.parent.findSuccessor()

self.parent.rightChild = self

return succ

def findMin(self):

current = self

while current.hasLeftChild():

current = current.leftChild

return current

def remove(self,currentNode):

if currentNode.isLeaf(): #leaf

if currentNode == currentNode.parent.leftChild:

currentNode.parent.leftChild = None

else:

currentNode.parent.rightChild = None

elif currentNode.hasBothChildren(): #interior

succ = currentNode.findSuccessor()

succ.spliceOut()

currentNode.key = succ.key

currentNode.payload = succ.payload

else: # this node has one child

if currentNode.hasLeftChild():

if currentNode.isLeftChild():

currentNode.leftChild.parent = currentNode.parent

currentNode.parent.leftChild = currentNode.leftChild

elif currentNode.isRightChild():

currentNode.leftChild.parent = currentNode.parent

currentNode.parent.rightChild = currentNode.leftChild

else:

currentNode.replaceNodeData(currentNode.leftChild.key,

currentNode.leftChild.payload,

currentNode.leftChild.leftChild,

currentNode.leftChild.rightChild)

else:

if currentNode.isLeftChild():

currentNode.rightChild.parent = currentNode.parent

currentNode.parent.leftChild = currentNode.rightChild

elif currentNode.isRightChild():

currentNode.rightChild.parent = currentNode.parent

currentNode.parent.rightChild = currentNode.rightChild

else:

currentNode.replaceNodeData(currentNode.rightChild.key,

currentNode.rightChild.payload,

currentNode.rightChild.leftChild,

currentNode.rightChild.rightChild)

mytree = BinarySearchTree()

mytree[3]="red"

mytree[4]="blue"

mytree[6]="yellow"

mytree[2]="at"

print(mytree[6])

print(mytree[2])