Plogramming Language(Python BST)

20163158 김남효

class Node:

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

self.leftchild = None

self.rightchild = None

self.data = data

self.size = 1

It is my Node class init source code. My code need to make root node like this root = Node(data) so its size is bigger than 1. But I only check the size with root.

1. Insertion\_function

def Insert\_function(self, data):

if(self.data == None):

self.data = data

else:

if(self.data>data):

if(self.leftchild == None):

self.leftchild = Node(data)

else:

self.leftchild.Insert\_function(data)

else:

if(self.rightchild == None):

self.rightchild = Node(data)

else:

self.rightchild.Insert\_function(data)

my code’s self is root. So if self.data == None(means root has no data) then input the data in root. If self.data>data then check self.leftchild is None or not. It means if self.leftchild is None then put or not none then do self.leftchild.insertion\_function(data) so check it base on self.leftchild.

If self.data <= data then check self.rightchild and do same like self.leftchild.

1. Also I make def find\_parent(self, data, parent) because it is useful when I do delete\_function.

if(self.data>data):

if(self.leftchild == None):

return None

return self.leftchild.find\_parent(data,self)

elif(self.data<data):

if(self.rightchild == None):

return None

return self.rightchild.find\_parent(data,self)

else:

return parent

it means find the node that equal the node.data = data and if it must do recursive then I save the before node so I can get parent node.

1. I make def find\_childeren(self,data)

It is almost same def find\_parent. The change is it is return node when node.data = data

def find\_childeren(self, data):

if(self.data>data):

if(self.leftchild == None):

return None

return self.leftchild.find\_childeren(data)

elif(self.data<data):

if(self.rightchild == None):

return None

return self.rightchild.find\_childeren(data)

else:

return self

1. I make function that counts of childeren nodes. It is useful when I use delete\_function

def childeren(self):

check = 0

if self.leftchild != None: # if leftchild is not none

check += 1

if self.rightchild != None: #if rightchild is not none

check += 1

return check

1. delete\_function. I have to check the 3types of counts of children nodes. (0, 1, 2)

def delete\_function(self, data):

ch\_child = 0#count of child nodes

child = self.find\_childeren(data) #child = node. Node is when node.data = data

parent = self.find\_parent(data, None)#when node is node.data = data, parent is its node’s parent node. It starts with None because root node’s parent is None

if(child == None):

return #if node.data!=data in all of node then there is no data in BST. So don’t have to do delete

else:

ch\_child = child.childeren()

if(ch\_child==0): #means it doesn’t have any children node

if(parent != None): #means this child is root or not.

If child = root, then go else, if child != root, then do next code

if(parent.leftchild == child):

parent.leftchild = None

#parent node don’t have to connect the leftchild because it is delete.

else:

parent.rightchild = None

#parent node don’t have to connect the rightchild because it is delete.

del child

else:#now child = root

self.data = None #root has no data now

self.size = 0 # its size must be 0

elif(ch\_child==1): #node has only one children node

if(child.leftchild != None): #if that children node in child(node).leftchild

temp = child.leftchild

else: #if that children node in child(node).rightchild

temp = child.rightchild

if(parent!=None): # child != root

if(parent.leftchild == child):

parent.leftchild = temp

#parent.leftchild must connect the child’s leftchild so child is delete

else:

parent.rightchild = temp

#parent.rightchild must connect the child’s rightchild so child is delete

del child

else: #if child = root then the root’s children go to root

self.data = temp.data

self.leftchild = temp.leftchild

self.rightchild = temp.rightchild

else: # now child’s count of children is 2

temp = child

successor = child.rightchild #I use successor so it start in the node’s rightchild

while(successor.leftchild != None):

temp = successor# I use temp to successor’s parent

successor = successor.leftchild

if(temp.leftchild == successor):

child.data = successor.data

temp.leftchild = successor.rightchild

#so child.data is successor.data so child data is delete, and successor’s leftchild is successor’s rightchild.

else:#it means ‘while’ don’t loop any time.(한번도 while문을 돌지 않았다.) so temp don’t have leftchild.

child.data = successor.data

temp.rightchild = successor.rightchild

#so child.data = successor.data so child data is delete, and successor’s rightchild is connect to temp.rightchild

def preorder(self): #print root first and left and right so make like this

if(self.size == 0):

return

print(self.data)

if(self.leftchild != None):

self.leftchild.preorder()

if(self.rightchild != None):

self.rightchild.preorder()

def postorder(self): #print left and right and root so make like this

if(self.size == 0):

return

if(self.leftchild != None):

self.leftchild.postorder()

if(self.rightchild != None):

self.rightchild.postorder()

print(self.data)

def inorder(self): #print left and root and right so make like this

if(self.size == 0):

return

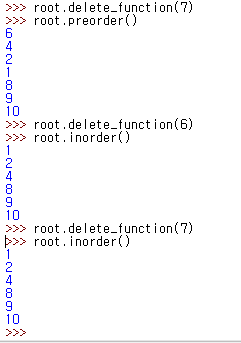
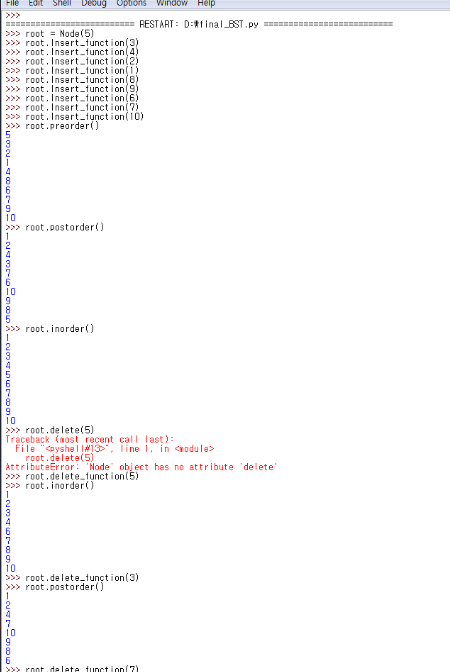
if(self.leftchild != None):

self.leftchild.inorder()

print(self.data)

if(self.rightchild != None):

self.rightchild.inorder()



Works like well