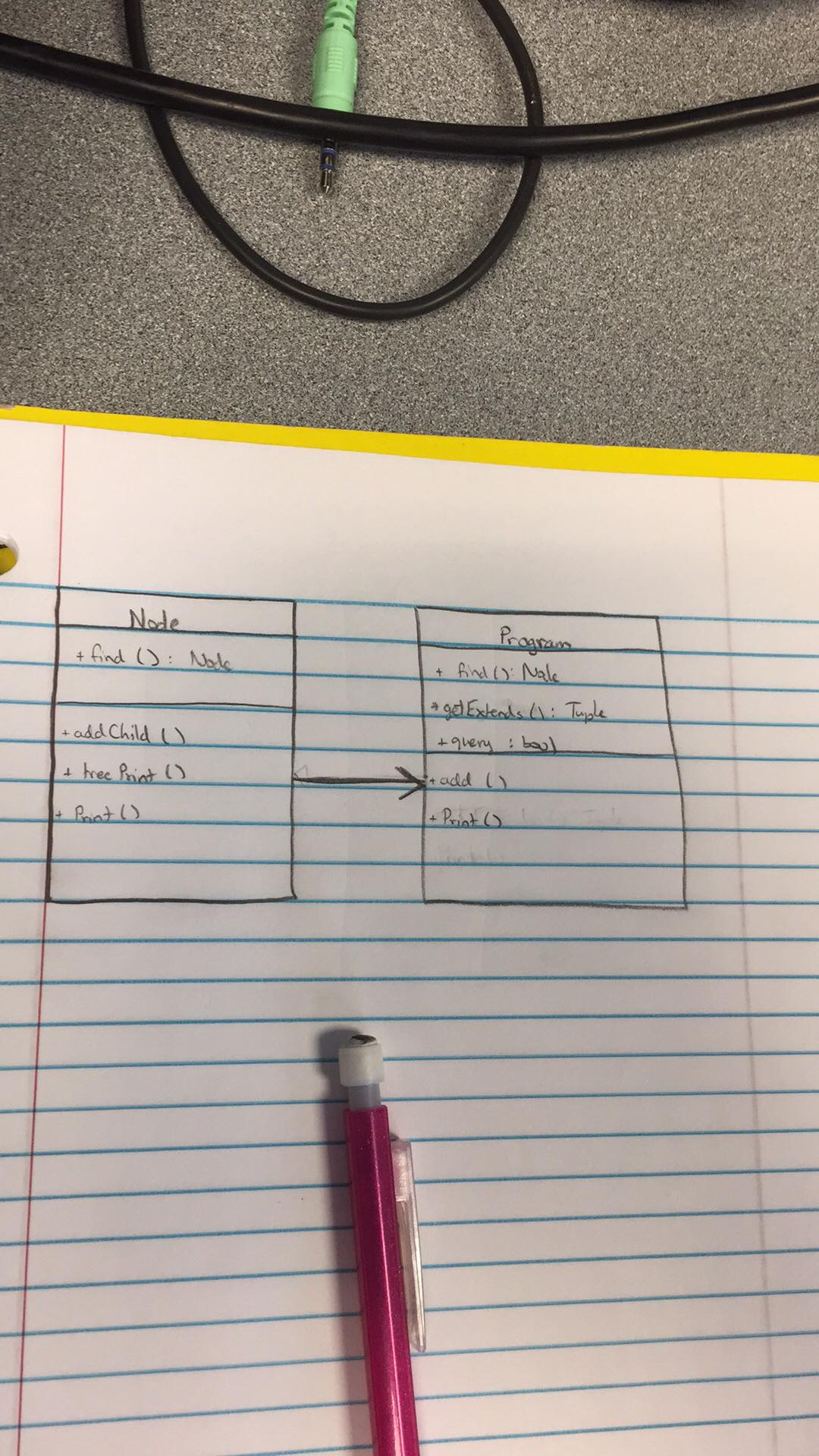
**MEMO**

TIME & SPACE ANALYSIS:

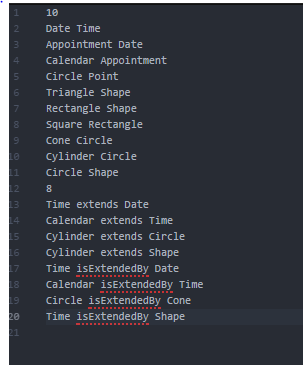
|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **Function** | **Time** | **Space** |
| Node | addChild(self, childPtr) | O(1) | O(n) |
| Node | find(self, target) | O(n) | O(n) |
| Node | treePrint(self, level) | O(n^2) | O(n) |
| Node | Print(self) | O(n^2) | O(n) |
| Program | find(self, target) | O(n) | O(n) |
| Program | Print(self) | O(n^2) | O(n^2) |
| Program | add(self, parent, child) | O(n) | O(n) |
| Program | getExtends(self, node, distance) | O(n) | O(n) |
| Program | query(self, c1, r, c2) | O(n) | O(n) |
| ------ | main() | O(n^2) | O(n^2) |

UML:

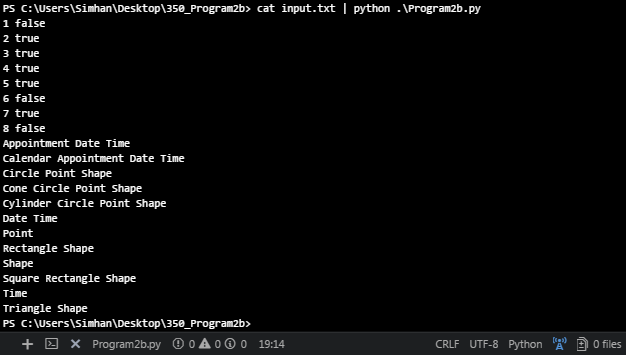


Screenshots:

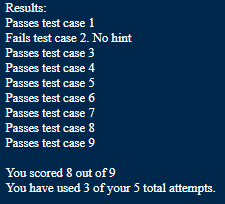
Input



Output



Test Cases

(case 2 gives me anxiety)

CODE:

# Srinivas Simhan

# CIS 350 - Elenbogen: Winter 2018

# Program 2b

# Due: 3/6/18

class Node:

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

self.children = []

self.extends = []

self.name = label

"""

PRE:

- the name of the child is given

POST:

- the name of the child is appended

DESCRIPTION:

- use this function to append a child to the a node, therefore extending the tree

"""

def addChild(self, childPtr): #Time O(1) #Space O(1)

self.children.append(childPtr) #Time O(1)

childPtr.extends.append(self) #Time O(1)

"""

PRE:

- the name of the target is given

POST:

- the function will return if you find the target or not

DESCRIPTION:

- the purpose of the function is to find the target node and return if you found it

"""

def find(self, target): #Time O(n) #Space O(n)

if self.name == target: #Time O(n) #Space O(1)

return self

else:

for node in self.children: #Time O(n) #Space O(n)

findPtr = node.find(target)

if findPtr is not None:

return findPtr

return None

"""

PRE:

- the level is given

POST:

- it will iterate through the levels and print the nodes

DESCRIPTION:

- this function is used to help print the tree

"""

#not needed anymore due to not doing it by bredth

def treePrint(self,level): #Time O(n^2) #Space O(n)

for i in range(1, 5 \* level): #Time O(n)

print(' ', end=""),

print(self.name)

for tree in self.children:

tree.treePrint( level + 1) #Time O(n)

"""

PRE:

- no givens

POST:

- it will call treePrint and print the tree

DESCRIPTION:

- calls treePrint function so that it can print the tree

"""

def Print(self):

self.treePrint(0) #Time O(n^2)

#/end of not needed code

class Program:

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

self.classes = []

self. allNodes = []

"""

PRE:

- the name of the target is given

POST:

- the function will return if you find the target or not

DESCRIPTION:

- the purpose of the function is to find the target node and return if you found it

"""

def find(self, target): #Time O(n) #Space O(n)

for tree in self.classes: #Time O(n) #Space O(n)

targetptr = tree.find(target)

if targetptr is not None: #Time O(1) #Space O(1)

return targetptr

return None

"""

PRE:

- no givens

POST:

- printed list of nodes (based on depth, not bredth)

DESCRIPTION:

- iterate through the list of nodes and print them out

"""

def Print(self): #Time O(n^2) #Space O(n^2)

self.allNodes.sort(key=lambda node: node.name) #Time O(nlogn) #Space O(1)

for node in self.allNodes: #Time O(n) #Space O(n)

extends = self.getExtends(node, 0) #Time O(n) #Space O(n)

# extends.sort(key=lambda extend: extend[1])

print(node.name, end='') #Time O(1) #Space O(1)

for extend in extends:

print('', extend[0].name, end='') #Time O(n) #Space O(1)

print()

"""

PRE:

- parent and child values given

POST:

- child is appended to the parent

DESCRIPTION:

- adds a child to the parent and appends it to the tree

"""

def add(self, parent, child): #Time O(n) #Space O(n)

parentPtr = self.find(parent) #Time O(n) #Space O(n)

childPtr = self.find(child) #Time O(n) #Space O(n)

if parentPtr is None:

parentPtr = Node(parent)

self.allNodes.append(parentPtr) #Time O(1) #Space O(1)

self.classes.append(parentPtr) #Time O(1) #Space O(1)

if childPtr is None:

childPtr = Node(child)

self.allNodes.append(childPtr) #Time O(1) #Space O(1)

parentPtr.addChild(childPtr) #Time O(1) #Space O(1)

else:

if childPtr is None:

childPtr = Node(child)

self.allNodes.append(childPtr) #Time O(1) #Space O(1)

parentPtr.addChild(childPtr)

for className in self.classes:

if className.name == child:

self.classes.remove(className) #Time O(n) #Space O(1)

"""

PRE:

- node and distance is given

POST:

- list of extends used for setting query and for printing

DESCRIPTION:

- creates a list of extends for use in query and printing functions

"""

#already set for depth first

def getExtends(self, node, distance): #Time O(n) #Space O(n)

extendsList = []

for extend in node.extends:

extendsList.append((extend, distance)) #Time O(1) #Space O(1)

extendsList += self.getExtends(extend, distance + 1) #Time O(n) #Space O(n)

return extendsList

"""

PRE:

- given c1, r, and c2: c1 and c2 are the names of two nodes, r is a

variable to choose if its extending or extended

POST:

- returns boolean if found and uses getExtends function

DESCRIPTION:

- used to implement a query to compare c1 and c2, and to return a bool value

"""

def query(self, c1, r, c2): #Time O(n) #Space O(n)

c1Node = self.find(c1) # find c1 #Time O(n) #Space O(n)

c2Node = self.find(c2) # find c2 #Time O(n) #Space O(n)

source = None

target = None

if c1 == c2:

return True

if r == "extends":

# does c1 extend c2 ?

source = c1Node

target = c2Node

elif r == "isExtendedBy":

# does c2 extend c1 ?\*

source = c2Node

target = c1Node

# get list of everything source extends. list is of tuples (Node, int distance)

extends = self.getExtends(source, 0) #Time O(n) #Space O(n)

for extend in extends: # look for target #Time O(n) #Space O(1)

# is the Node in extend target?

if extend[0] is target:

return True

return False # couldn't find :(

"""

PRE:

- no givens (inputs are in the main)

POST:

- list out the nodes based on relations

DESCRIPTION:

- this is what runs the program

"""

def main(): #Time O(n^2) #Space O(n^2)

prog = Program() #Time O(1) #Space O(1)

numPairs = int(input()) #Time O(1) #Space O(1)

for num in range(numPairs): #Time O(n^2) #Space O(n^2)

relation = input().split() #Time O(n) #Space O(n)

parent = relation[1]

child = relation[0]

prog.add(parent, child) #Time O(n) #Space O(n)

# how many queries?

numQuery = int(input())

for m in range(numQuery): #Time O(n^2) #Space O(n^2)

# read in and respond to each one

relation = input().split() #Time O(n) #Space O(n)

c1 = relation[0]

r = relation[1]

c2 = relation[2]

result = None

if prog.query(c1, r, c2): #Time O(n) #Space O(n)

result = "true"

else:

result = "false"

print(m + 1, result)

prog.Print()

main()