 **Northwestern Polytechnic University**

**Python Programming**

**Homework Assignment #6**

**Due day: 11/23/2021**

**Instruction:**

1. **Push the source code to GitHub or answer sheet in word file**
2. **Please follow the code style rule like programs on handout.**
3. **Overdue homework submission could not be accepted.**

**4. Takes academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)**

**Link to execute programs:** <https://colab.research.google.com/drive/1rIKINSAqfKI3REFYkPSmRwFteXPIZDkq?usp=sharing>

1. Write a function to check if a tree contains some value.

***def******has\_itm(t, e):***

*"""*

*>>> has\_itm (tree(11, [tree(12), tree(13, [tree(14),tree(15)])] ), 11)*

*True*

*>>> has\_itm (tree(11, [tree(12), tree(13, [tree(14),tree(15)])] ), 16)*

*False*

*"""*

def branches (tree):

return tree[1:]

def root(tree):

return tree[0]

def tree(root, branches=[]):

for branch in branches:

assert isatree (branch), 'branches should be trees'

return [root] + list(branches)

def isatree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not isatree (branch):

return False

return True

def has\_itm(t, e):

if root(t)==None:

return False

if root(t)==e:

return True

for i in branches(t):

if has\_itm(i,e):

return True

return False

has\_itm (tree(11, [tree(12), tree(13, [tree(14),tree(15)])] ), 11)

has\_itm (tree(11, [tree(12), tree(13, [tree(14),tree(15)])] ), 16)

1. Create a function to calculate the average value at each node in a tree.

***def*** ***ave(t, e):***

*"""*

*>>> ave(tree(11, [tree(12), tree(13, [tree(14),tree(15)])]))*

*13.0 # (11+12+13+14+15)/5 = 13.0*

*"""*

def branches (tree):

return tree[1:]

def root(tree):

return tree[0]

def tree(root, branches=[]):

for branch in branches:

assert isatree (branch), 'branches should be trees'

return [root] + list(branches)

def isatree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not isatree (branch):

return False

return True

def no\_of\_nodes(t):

s=0

if root(t)==None:

return s

s+=1

for i in branches(t):

s+=no\_of\_nodes(i)

return s

def sum(t):

s=0

if root(t)==None:

return s

s+=root(t)

for i in branches(t):

s+=sum(i)

return s

def ave(t):

if root(t)==None:

return 0

total=sum(t)

no=no\_of\_nodes(t)

return total/no

ave(tree(11, [tree(12), tree(13, [tree(14),tree(15)])]))

1. Based on the tree construction functions on the handout, write a function to generate tree for *Fibonacci* series.

***def tree\_fib(n):***

*"""Construct a Fibonacci tree.*

*>>> tree\_fib (1)*

*[1]*

*>>> tree\_fib (3)*

*[2, [1], [1, [0], [1]]]*

*>>> tree\_fib(5)*

*[5, [2, [1], [1, [0], [1]]], [3, [1, [0], [1]], [2, [1], [1, [0], [1]]]]]*

*"""*

def tree\_fib(n):

if n<2:

return [n]

return [fib(n),tree\_fib(n-2), tree\_fib(n-1)]

def fib(x):

if x<2:

return x

return fib(x-2) + fib(x-1)

tree\_fib(1)

tree\_fib (3)

tree\_fib(5)

1. Generate a *def* function to apply math operational function for all leaves in a tree.

***def******app\_func\_leaves(t, g):***

*"""*

*>>> def triple(n):*

*return 3\*n*

*>>> def square(m):*

*return m\*m*

*>>> t= tree(1,*

*[tree(2),*

*tree(3,*

*[tree(4),*

*tree(5)])*

*])*

*>>> app\_func\_leaves(t, triple)*

*6 12 15 # leaves: 2, 4, 5*

*>>> app\_func\_leaves(t, square)*

*4 16 25*

*"""*

def triple(n):

return 3\*n

def square(m):

return m\*m

def tree(root, branches=[]):

for branch in branches:

assert is\_tree (branch), 'branches should be trees'

return [root] + list(branches)

def root(tree):

return tree[0]

def branches (tree):

return tree[1:]

def is\_tree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not is\_tree (branch):

return False

return True

def is\_leaf (tree):

return not branches(tree)

def app\_func\_leaves(t, g):

if branches(t) == []:

print(g(root(t)), end=' ')

else:

for b in branches(t):

app\_func\_leaves(b, g)

t= tree(1, [tree(2), tree(3, [tree(4),tree(5)])])

app\_func\_leaves(t, triple)

app\_func\_leaves(t, square)

1. Define a function to replace all leaves in a tree with new values as return, but don’t change original tree

***def******rpl\_leaves(t, old, new):***

*"""*

*t = tree('apple',*

*[tree('banana',*

*[tree('plum'),*

*tree('pear')]),*

*tree('peach',*

*[tree('plum')]),*

*tree('plum',*

*[tree('berry'),*

*tree('plum')]),*

*tree('plum')])*

>>> *s=rpl\_leaves (t, 'plum', 'fig'))*

>>> *s*

*tree('apple',*

*[tree('banana',*

*[tree('fig'),*

*tree('pear')]),*

*tree('peach',*

*[tree('fig')]),*

*tree('plum',*

*[tree('berry'),*

*tree('fig')]),*

*tree('fig')])*

"""

def tree(root, branches=[]):

for branch in branches:

assert is\_tree(branch),'branches should be trees'

return [root] + list(branches)

def root(tree):

return tree[0]

def branches (tree):

return tree[1:]

def is\_leaves(tree):

return not branches(tree)

def swap(a, b):

a[:], b[:] = b[:], a[:]

def is\_tree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not is\_tree (branch):

return False

return True

def rpl\_leaves(t, old, new):

if is\_leaves(t):

if root(t) == old:

return tree(new)

else:

return t

else:

st = [rpl\_leaves(b, old, new) for b in branches(t)]

return tree(root(t), st)

t = tree('apple',[tree('banana',

[tree('plum'),tree('pear')]),

tree('peach',[tree('plum')]),

tree('plum',[tree('berry'),

tree('plum')]),tree('plum')])

s=rpl\_leaves (t, 'plum', 'fig')

s

1. Write a function to count a tree’s height, which is the length of the longest path from the root to a leaf.

def branches (tree):

return tree[1:]

def tree(root, branches=[]):

for branch in branches:

assert is\_tree (branch), 'branches should be trees'

return [root] + list(branches)

def is\_tree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not is\_tree (branch):

return False

return True

def height(tree):

if branches(tree) == []:

return 0

else:

hite = [height(b) for b in branches(tree)]

return 1 + max(hite)

t = (tree(11, [tree(20), tree(10, [tree(5),tree(1),[tree(2), tree(3, )]])]))

height(t)

1. Return largest node value in a numeric tree by a *def* function

def root(tree):

return tree[0]

def branches (tree):

return tree[1:]

def tree(root, branches=[]):

for branch in branches:

assert is\_tree (branch), 'branches should be trees'

return [root] + list(branches)

def is\_tree (tree):

if type(tree) != list or len (tree) < 1:

return False

for branch in branches(tree):

if not is\_tree (branch):

return False

return True

def largest(t):

if root(t)==None:

return 0

large=root(t)

for i in branches(t):

if largest(i)>large:

large=largest(i)

return large

t=tree(15, [tree(25), tree(13, [tree(12),tree(10)])])

largest(t)