import uuid

def sanitize\_id(id):

return id.strip().replace(" ", "")

(\_ADD, \_DELETE, \_INSERT) = range(3)

(\_ROOT, \_DEPTH, \_WIDTH) = range(3)

class Node:

def \_\_init\_\_(self, name, identifier=None, expanded=True):

self.\_\_identifier = (str(uuid.uuid1()) if identifier is None else

sanitize\_id(str(identifier)))

self.name = name

self.expanded = expanded

self.\_\_bpointer = None

self.\_\_fpointer = []

@property

def identifier(self):

return self.\_\_identifier

@property

def bpointer(self):

return self.\_\_bpointer

@bpointer.setter

def bpointer(self, value):

if value is not None:

self.\_\_bpointer = sanitize\_id(value)

@property

def fpointer(self):

return self.\_\_fpointer

def update\_fpointer(self, identifier, mode=\_ADD):

if mode is \_ADD:

self.\_\_fpointer.append(sanitize\_id(identifier))

elif mode is \_DELETE:

self.\_\_fpointer.remove(sanitize\_id(identifier))

elif mode is \_INSERT:

self.\_\_fpointer = [sanitize\_id(identifier)]

class Tree:

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

self.nodes = []

def get\_index(self, position):

for index, node in enumerate(self.nodes):

if node.identifier == position:

break

return index

def create\_node(self, name, identifier=None, parent=None):

node = Node(name, identifier)

self.nodes.append(node)

self.\_\_update\_fpointer(parent, node.identifier, \_ADD)

node.bpointer = parent

return node

def show(self, position, level=\_ROOT):

queue = self[position].fpointer

if level == \_ROOT:

print("{0} [{1}]".format(self[position].name, self[position].identifier))

else:

print("\t"\*level, "{0} [{1}]".format(self[position].name, self[position].identifier))

if self[position].expanded:

level += 1

for element in queue:

self.show(element, level) # recursive call

def expand\_tree(self, position, mode=\_DEPTH):

# Python generator. Loosly based on an algorithm from 'Essential LISP' by

# John R. Anderson, Albert T. Corbett, and Brian J. Reiser, page 239-241

yield position

queue = self[position].fpointer

while queue:

yield queue[0]

expansion = self[queue[0]].fpointer

if mode is \_DEPTH:

queue = expansion + queue[1:] # depth-first

elif mode is \_WIDTH:

queue = queue[1:] + expansion # width-first

def is\_branch(self, position):

return self[position].fpointer

def \_\_update\_fpointer(self, position, identifier, mode):

if position is None:

return

else:

self[position].update\_fpointer(identifier, mode)

def \_\_update\_bpointer(self, position, identifier):

self[position].bpointer = identifier

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

return self.nodes[self.get\_index(key)]

def \_\_setitem\_\_(self, key, item):

self.nodes[self.get\_index(key)] = item

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

return len(self.nodes)

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

return [node.identifier for node in self.nodes if node.identifier is identifier]

if \_\_name\_\_ == "\_\_main\_\_":

tree = Tree()

tree.create\_node("Harry", "harry") # root node

tree.create\_node("Jane", "jane", parent = "harry")

tree.create\_node("Bill", "bill", parent = "harry")

tree.create\_node("Joe", "joe", parent = "jane")

tree.create\_node("Diane", "diane", parent = "jane")

tree.create\_node("George", "george", parent = "diane")

tree.create\_node("Mary", "mary", parent = "diane")

tree.create\_node("Jill", "jill", parent = "george")

tree.create\_node("Carol", "carol", parent = "jill")

tree.create\_node("Grace", "grace", parent = "bill")

tree.create\_node("Mark", "mark", parent = "jane")

print("="\*80)

tree.show("harry")

print("="\*80)

for node in tree.expand\_tree("harry", mode=\_WIDTH):

print(node)

print("="\*80)

**anytree**

I recommend <https://pypi.python.org/pypi/anytree>

**Example**

from anytree import Node, RenderTree

udo = Node("Udo")

marc = Node("Marc", parent=udo)

lian = Node("Lian", parent=marc)

dan = Node("Dan", parent=udo)

jet = Node("Jet", parent=dan)

jan = Node("Jan", parent=dan)

joe = Node("Joe", parent=dan)

print(udo)

Node('/Udo')

print(joe)

Node('/Udo/Dan/Joe')

for pre, fill, node in RenderTree(udo):

print("%s%s" % (pre, node.name))

Udo

├── Marc

│ └── Lian

└── Dan

├── Jet

├── Jan

└── Joe

print(dan.children)

(Node('/Udo/Dan/Jet'), Node('/Udo/Dan/Jan'), Node('/Udo/Dan/Joe'))

node = { 'parent':0, 'left':0, 'right':0 }

import copy

root = copy.deepcopy(node)

root['parent'] = -1

left = copy

just to show another thought on implementation if you stick to the "OOP"

class Node:

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

self.data = data

self.child = {}

def append(self, title, child):

self.child[title] = child

CEO = Node( ('ceo', 1000) )

CTO = ('cto',100)

CFO = ('cfo', 10)

CEO.append('left child', CTO)

CEO.append('right child', CFO)

print CEO.data

print ' ', CEO.child['left child']

print ' ', CEO.child['right child']