Python Intermediate Programming

Unit 3:Python Data Structures

CHAPTER 9: TREE II

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In the second lecture on trees we will learn how to use lists (sets, if ordering among nodes is not important) to store N-ary trees (trees where each node has an arbitrary number of children: 0 -for leaves- or more). We will also discuss traversal orders for binary trees (from the previous lecture) and general trees too.
- The most common example of an N-ary tree is a file system. There is a root folder which contain files and other folders (which can contain files and other folders, etc.).

Simple N-ary Trees

LECTURE 1

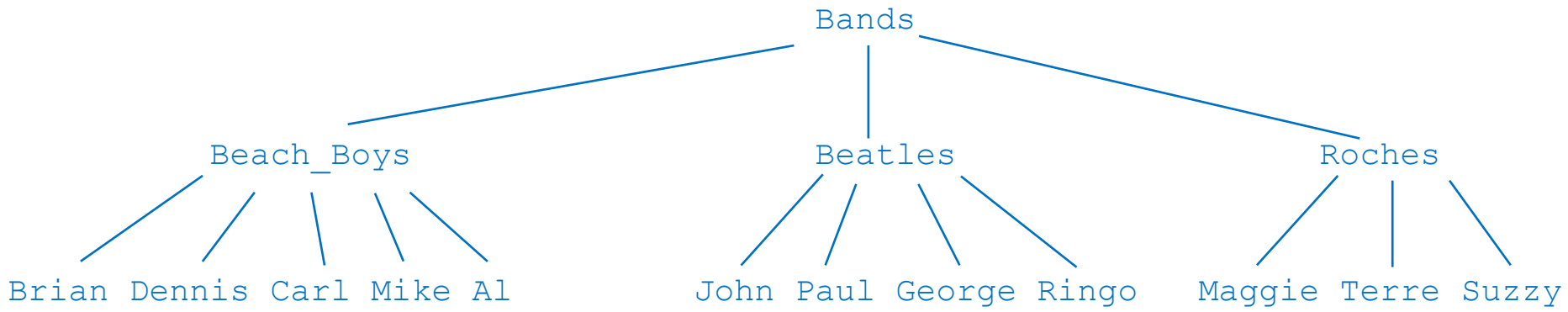
Simple N-ary Trees

- The main way we will store N-ary trees is as a node containing both a value and a list of non-empty subtrees (each itself a TN; no Nones). The length of each list is the number of children/subtrees of the node represented by each TN. We can use the following simple class to store this information.

```
class TN:
    def __init__(self, value, children = [] ):
        self.value     = value
    self.children  = children
    # set() when order of children is unimportant
```

Simple N-ary Trees

- We will discuss many operations on trees below, written as functions. We can also define methods that implement these operations as methods in the TN class, but prefer to uses functions here. First, here is an example of a simple N-ary tree for storing bands and their members. This tree, even if we added more bands and members, would always have a height of 2.



Simple N-ary Trees

- Using the TN class, we represent this tree as the following TN objects. Each TN object has a value and a list (potentially empty) of children/subtrees.

```
TN('Bands',
   [TN('Beach_Boys',
         [TN('Brian'), TN('Dennis'), TN('Carl'), TN('Mike'), TN('Al')]
    TN('Beatles',
         [TN('John'), TN'Paul'), TN('George'), TN('Ringo')]
    TN('Roches',
         [TN('Maggie'), TN('Terre'), TN('Suzzy')]
   ]
```

Simple N-ary Trees

- Notice that each TN constructor with two arguments is a string followed by a list of TN. The TN constructors with one argument is just a string, and no list of children (so the empty list of children -the default value in __init__) is used.
- Similarly, to how we computed the height and size of binary trees, we can compute these quantities for N-ary trees. The code for both mixes iteration (to produce all the children of a node) and recursion (to process each child's subtree).

Simple N-ary Trees

- Here are these method written with simple Python constructs. Note that we assume that atree is a TN: each recursive call is also guaranteed to be a TN, from the list of children (we don't need the conventional base cases needed in recursion; or, we can think of leaf nodes -those with no children- as base cases): e.g., in the size function, if atree has no children (the base case), then the loop executes 0 times and the count of that one TN is 1; in the height function, if atree has no children (the base case), then the loop executes 0 times and the height of that one TN is 0 (-1 + 1).
- So, in both cases, the base case is not explicit, but results in no loop executions.

Simple N-ary Trees

```
def size(atree):
    count = 1
    for c in atree.children:
        count += size(c)
    return count

def height(atree):
    h = -1
    for c in atree.children:
        h = max(h,height(c))
    return 1 + h
```

Simple N-ary Trees

- We can reduce the code in these functions (not sure if this simplifies them in terms of understanding them) by using Pythons sum/max functions (which operate on arguments to iterate over) and comprehensions which accumulate the values of the recursive calls on all the children/subtrees.

```
def size(atree):
    return 1 + sum( size(c) for c in atree.children )
def height(atree):
    if atree.children == []:
        return 0
    else:
       return 1 + max( height(c) for c in atree.children )
```

Simple N-ary Trees

- Note that the sum function returns 0 for an empty argument list (which produces the correct answer). But the max function raises a ValueError when its argument list is empty. If we defined emax to generalized max, allowing us to specify what value emax should return for an empty argument list

```
def emax(*args,empty = None):
    if args == ():
        return empty
    else:
        return max(*args)
```

Simple N-ary Trees

then we could likewise simplify height (whose if was needed in case that atree.children was empty). Here if atree.children is empty, emax returns 0 (so, it uses an if inside emax's defintion).

```
def height(atree):
    return emax( (height(c)+1 for c in atree.children), 0 )
```

- We can use nested lists in Python to represent trees as well. For the tree above we would represent it as follows.

Simple N-ary Trees

```
['Bands',
   ['Beach_Boys',
      ['Brian'], ['Dennis'], ['Carl'], ['Mike'], ['Al']
   ],
   ['Beatles',
      ['John'], ['Paul'], ['George'], ['Ringo']
   ],
   ['Roches',
      ['Maggie'], ['Terre'], ['Suzzy']
   ]
  ]
```

- Here, the value at the root of a subtree is always followed by a list of its subtrees (also in this list form). So 'Bands' is followed by a list of 3 bands; the first band, 'Beach_Boys', is followed by a list of its 5 member; each member is list that is a name followed by no other values (a 1-list)

Simple N-ary Trees

- We can write short recursive functions to translate between these two representations. Again, we see the combination of iteration (in a comprehension) to produce all the children of a node, with recursive calls of the function on each child. Note that if alist has just one value, alist[1:] is an empty list.

```
def list_to_tree(alist):
    if alist == []:
        return []
    else:
        return TN(alist[0], [list_to_tree(x) for x in alist[1:]])
def tree_to_list(atree):
    return [atree.value]+[tree_to_list(c) for c in atree.children]
```

Simple N-ary Trees

- The project file accompanying this lecture includes a version of print_tree for N-ary trees. For every node, it recursively prints 1/2 its children, then the node's value, then recursively prints the other 1/2 of its children. There is a problem though. For the above it prints as follows

```
. . . . Brian
. . . . Dennis
. . Beach_Boys
. . . . Carl
. . . . Mike
. . . . Al
Bands
. . . . John
. . . . Paul
. . Beatles
. . . . George
. . . . Ringo
. . . . Maggie
. . Roches
. . . . Terre
. . . . Suzzy
```

and when this tree is rotated 90 clockwise, it is not obvious where the (left) members of the Beatles ends and the (right) members of the Roches begins.

Simple N-ary Trees

- If we want to determine whether a value is in an N-ary we can define the following function. Because there is generally no order property on N-ary trees (unlike binary search trees) we must search all children of each node, if the value is not at the node. This requires many recursive calls.

```
def contains(atree,value):
    return atree.value == value or\
           any( contains(c,value) for c in atree.children )
```

Simple N-ary Trees

- The any function returns True if its argument iterator (here a tuple) produces one True value (in this case, any won't produce any more values to check); if the iterator terminates before producing any value, any returns False. So, value is contained in atree if it is the value at the root of the tree or if it is contained in any of the subtrees.
- The find function searches the N-ary atree and returns a reference to the node containing value, or None if value is not found anywhere in atree. Here we will assume that every value in the N-ary tree is unique.

Simple N-ary Trees

```
def find(atree,value):
    if atree.value == value:
        return atree
    else:
        for c in atree.children:
            answer = find(c,value)
            if answer != None:
                return answer
        return None
```

Simple N-ary Trees

- In find we search for the value in the tree, much like we did in contains. But, instead of returning a bool value, if we find a TN with value, we return the TN (and don't recur any deeper). If we don't find that value in this TN, we iterate through its children, trying to find the value in each subtree. For most subtrees (which don't contain the value) the recursive call will return None; but if a non-None value is returned in a recursive call, that value is found so it is returned for the current TN. If the answer for each recursive call on the subtrees is None, eventualy the loop terminates and this function returns
- None.
- Let's look at one more function: finding all the ancestors of a value in a tree. The ancestors of a node are its parent, its parent's parent, etc., all the way back to the root of a tree. To simplify this function, we will also consider a value to be an ancestor of itself. Here we will also assume that every value in the N-ary tree is unique.

Simple N-ary Trees

```
def ancestors(atree,value):
    if atree.value == value:
        return [value]
    else:
        for c in atree.children:
            answer = ancestors(c,value)
            if answer != []:
                return [atree.value] + answer
        return []
```

Simple N-ary Trees

- This function operates very similarly to the find function discussed above. In ancestors we search for the value in the tree, much like we did in contains and find. But instead of returning a bool value or the TN, if we find the value we return a list with just that value (and don't recur any deeper). If we don't find that value in this TN, we iterate through its children, computing the ancestors in each subtree.
- For most subtrees (which don't contain the value) the recursive call will return []; but if a non-empty list is returned, the value was in that subtree, so the value of the current TN is an ancestor, so a list containing the value of the current TN is prepended to the list of ancestors, and this new list is returned. If the answer for each recursive call on the subtrees is empty, eventualy the loop terminates and this function returns an empty list.

Traversal Orderings

LECTURE 2

Traversal Orderings

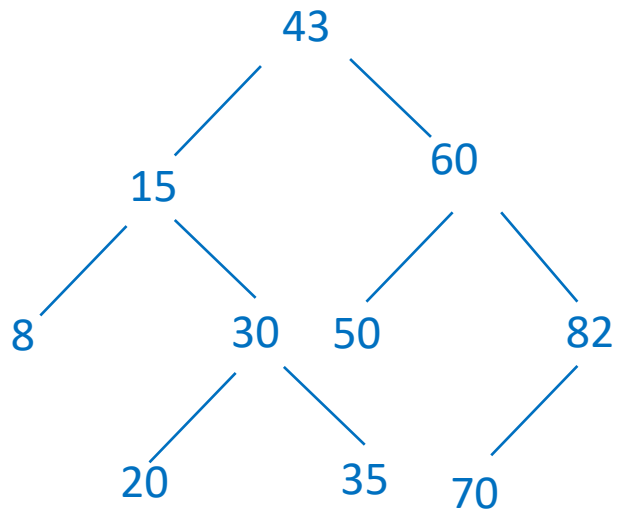
There are four traversal orderings that apply to binary trees: preorder, inorder, postorder, and breadth-first order. In the first three cases, the word before order determines when a parent node is processed compared to its children.

1. **Preorder:** process the value in the parent node first, then recursively process its left and right subtrees; typically we process the left subtree before the right subtree, although we can talk about reverse preorder, which processes the right subtree before the left subtree.
2. **Inorder:** recursively process the left subtree first, then process the value in the parent node, then recursively process its right subtree; ditto for standard and reverse ordering
3. **Postorder:** recursively process the left and right subtrees first, then process the value in the parent node; ditto for standard and reverse ordering (see Preorder)

Traversal Orderings

4. breadth-first order: process all the nodes at depth 0 (just the root), then all at depth 1, then all at depth 2, etc.

For the binary search tree:



Traversal Orderings

Here are these four different orders.

```
Preorder                : 43, 15,  8, 30, 20, 35, 60, 50, 82, 70
Inorder                 :  8, 15, 20, 30, 35, 43, 50, 60, 70, 82
Postorder               :  8, 20, 35, 30, 15, 50, 70, 82, 60, 43
Breadth-first order: 43, 15, 60,  8, 30, 50, 82, 20, 35, 70
```

Traversal Orderings

- The project downloaded for the previous trees lecture includes the methods generator_preorder, generator_inorder, generator_postorder, and generator_breadth_first, all are generators that can all be used with iterators trivially:

```
for v in generator_inorder(atree):
    print(v)
```

Traversal Orderings

- The preorder, inorder, and postorder generators all look the same: using two loops on recursive calls for iterating over subtrees: it is just WHERE the yield atree.value appears compared to these loops: pre/before the loops, in-between the loops, post/after the loops. The breadth-first order generator uses a single loop and a list of the nodes to visit (in depth-first order).
- The N-ary tree versions are similar, although there is no equivalent for inorder traveral when there are arbitrary number of children/subtrees. These appear in the project folder for this lecture