CS61A NOTE11 Mutabel Tress

Mutabel Trees

We define a tree to be a recursive data abstraction that has a `label` (the value stored in the root of the tree) and `branches` (a list of trees directly underneath the root).

Previously we implemented trees by using a functional data abstraction, with the `tree` constructor function and the `label` and `branches` selector functions. Now we implement trees by creating the `Tree` class. Here is part of the class included in the lab.

纯文本

```
class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

Even though this is a new implementation, everything we know about the functional tree data abstraction remains true. That means that solving problems involving trees as objects uses the same techniques that we developed when first studying the functional tree data abstraction (e.g. we can still use recursion on the branches!). The main difference, aside from syntax, is that tree objects are mutable.

-	**Tree constructor and selector functions**	**Tree class**
Constructing a tree	To construct a tree given a `label` and a list of `branches`, we call `tree(label, branches)`	To construct a tree object given a `label` and a list of `branches`, we call `Tree(label, branches)` (which calls the `Tree.__init__` method).
Label and branches	To get the label or branches of a tree `t`, we call `label(t)` or `branches(t)` respectively	To get the label or branches of a tree `t`, we access the instance attributes `t.label` or `t.branches` respectively.
Mutability	The functional tree data abstraction is immutable because we cannot assign values to call expressions	The `label` and `branches` attributes of a `Tree` instance can be reassigned, mutating the tree.
Checking if a tree is a leaf	To check whether a tree `t` is a leaf, we call the convenience function `is_leaf(t)`	To check whether a tree `t` is a leaf, we call the bound method `t.is_leaf()`. This method can only be called on `Tree` objects.

注意 t.is_leaf()等价于 is_leaf(t)

lab9 WWPD

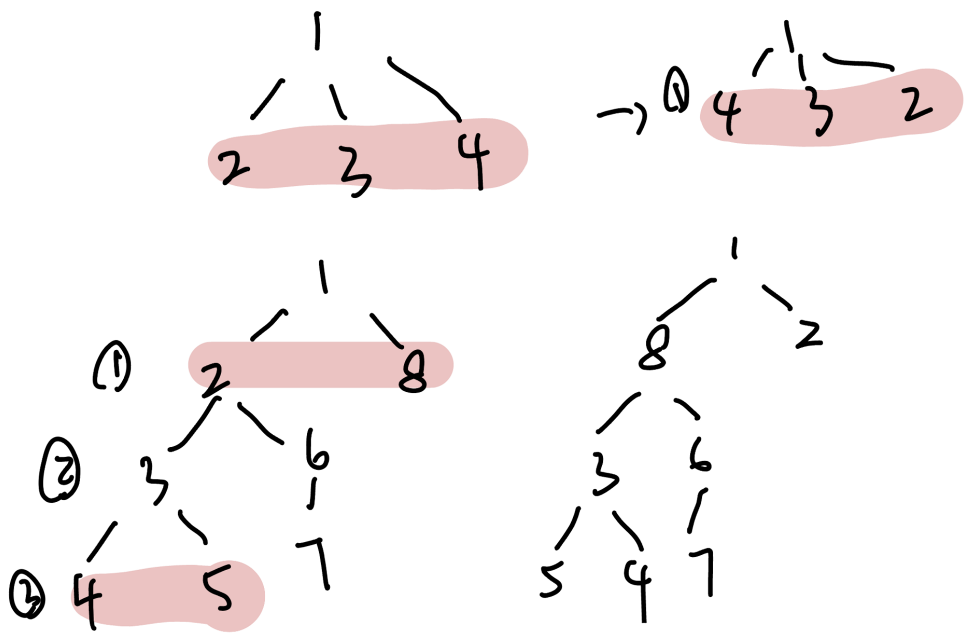
纯文本

```
>>> from lab09 import *
>>> t = Tree(1, Tree(2))
Error #后面必须是[]

>>> t = Tree(1, [Tree(2)])
>>> t.label
1

>>> t.branches[0]
Tree(2) #易错，不是2，而是树
```

```
>>> t.branches[0].label
2

>>> t.label = t.branches[0].label
>>> t
Tree(2, [Tree(2)]) #注意空格，可变

>>> t.branches.append(Tree(4, [Tree(8)]))
>>> len(t.branches)
2 #append是在[]里加el，只加了1

>>> t.branches[0]
Tree(2)

>>> t.branches[1]
Tree(4, [Tree(8)])
```

Q10: Reverse Other 过于难了

Write a function `reverse_other` that mutates the tree such that **labels** on *every other* (odd-depth) level are reversed. For example, `Tree(1,[Tree(2, [Tree(4)]), Tree(3)])` becomes `Tree(1,[Tree(3, [Tree(4)]), Tree(2)])`. Notice that the nodes themselves are *not* reversed; only the labels are.



```
def reverse_other(t):
    """Mutates the tree such that nodes on every other (odd-depth)
    level have the labels of their branches all reversed.

    >>> t = Tree(1, [Tree(2), Tree(3), Tree(4)])
    >>> reverse_other(t)
    >>> t
    Tree(1, [Tree(4), Tree(3), Tree(2)])
    >>> t = Tree(1, [Tree(2, [Tree(3, [Tree(4), Tree(5)]), Tree(6, [Tree(7)
    >>> reverse_other(t)
    >>> t
    Tree(1, [Tree(8, [Tree(3, [Tree(5), Tree(4)]), Tree(6, [Tree(7)])]), Tr
    """
    "*** YOUR CODE HERE ***"
    def reverse_helper(t, need_reverse):
        if t.is_leaf():
            return
        new_labs = [child.label for child in t.branches][::-1]
        for i in range(len(t.branches)):
            child = t.branches[i]
            reverse_helper(child, not need_reverse)
            if need_reverse:
                child.label = new_labs[i]
    reverse_helper(t, True)
```

disc Q7: Find Paths(Additional Practice: Trees)有点小难

Define the procedure `find_paths` that, given a Tree `t` and an `entry`, returns a list of lists containing the nodes along each path from the root of `t` to `entry`. You may return the paths in any order.找出到 entry 的所有路

For instance, for the following tree `tree_ex`, `find_paths` should behave as specified in the function doctests.

纯文本

```
def find_paths(t, entry):
    """
    >>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])])
    >>> find_paths(tree_ex, 5)
    [[2, 7, 6, 5], [2, 1, 5]]
    >>> find_paths(tree_ex, 12)
    []
    """
    paths=[]#里面添路线元素
    if t.label=entry#也可不只这一个，跟之前不完全一样
        paths.append([t,label]) #别写else
    for i in t.branches:
        for path in find_paths(i, entry):  #也可能有多条所以要for
            paths.append([t.label]+path)
    return paths
```

lab9 Q2: Make Even

Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.输入树 t，改变之使得基数全 +1 偶数不变

纯文本

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
    "*** YOUR CODE HERE ***"
    if t.label%2!=0: #不用定义leaf吗？
        t.label+=1
    for b in t.branches:
        make_even(b)
    return             #确实不返回任何，但为何要加return

    # Alternate Solution
    t.label += t.label % 2
    for branch in t.branches:
        make_even(branch)
    return
```

Q3: Cumulative Mul

Write a function `cumulative_mul` that mutates the Tree `t` so that each node's label becomes the product of its label and all labels in the subtrees rooted at the node.

> **Hint**: Consider carefully when to do the mutation of the tree and whether that mutation should happen before or after processing the subtrees.利用可变性，仔细考虑变化时间

纯文本

```
def cumulative_mul(t):
    """Mutates t so that each node's label becomes the product of all label
    the corresponding subtree rooted at t.

    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> cumulative_mul(t)
    >>> t
    Tree(105, [Tree(15, [Tree(5)]), Tree(7)])
    >>> otherTree = Tree(2, [Tree(1, [Tree(3), Tree(4), Tree(5)]), Tree(6,
    >>> cumulative_mul(otherTree)
    >>> otherTree
    Tree(5040, [Tree(60, [Tree(3), Tree(4), Tree(5)]), Tree(42, [Tree(7)])]
    """
    "*** YOUR CODE HERE ***" #有意思
    for b in t.branches:
        cumulative_mul(b)
    result=t.label
    for b in t.branches:
        result *= b.label
    t.label=result
```

Q4: Prune Small 修剪小的

Complete the function `prune_small` that takes in a `Tree` `t` and a number `n` and prunes `t` mutatively. If `t` or any of its branches has more than `n` branches, the `n` branches with the smallest labels should be kept and any other branches should be *pruned*, or removed, from the tree.输入树 t 和数字 n，修剪 t，若 t 或 t 的 branches 有大于 n 的 branches，有最小 label 的 n 个 branches 保留，其余修剪掉

> *Hint*: The `max` function takes in an `iterable` as well as an optional `key` argument (which takes in a one-argument function). For example, `max([-7, 2, -1], key = abs)` would return `-7` since `abs(-7)` is greater than `abs(2)` and `abs(-1)`.

纯文本

```
def prune_small(t, n):
    """Prune the tree mutatively, keeping only the n branches
    of each node with the smallest labels.

    >>> t1 = Tree(6)
    >>> prune_small(t1, 2)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_small(t2, 1)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2), Tree(3)]), Tree(5
    >>> prune_small(t3, 2)
    >>> t3
    Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2)])])
    """
    while len(t.branches)>n:
        large=max(t.branches, key=lambda x: x.label)
        t.branches.remove(large)
    for b in t.branches:
        prune_small(b, n)
```

lab9 Q6 Is BST

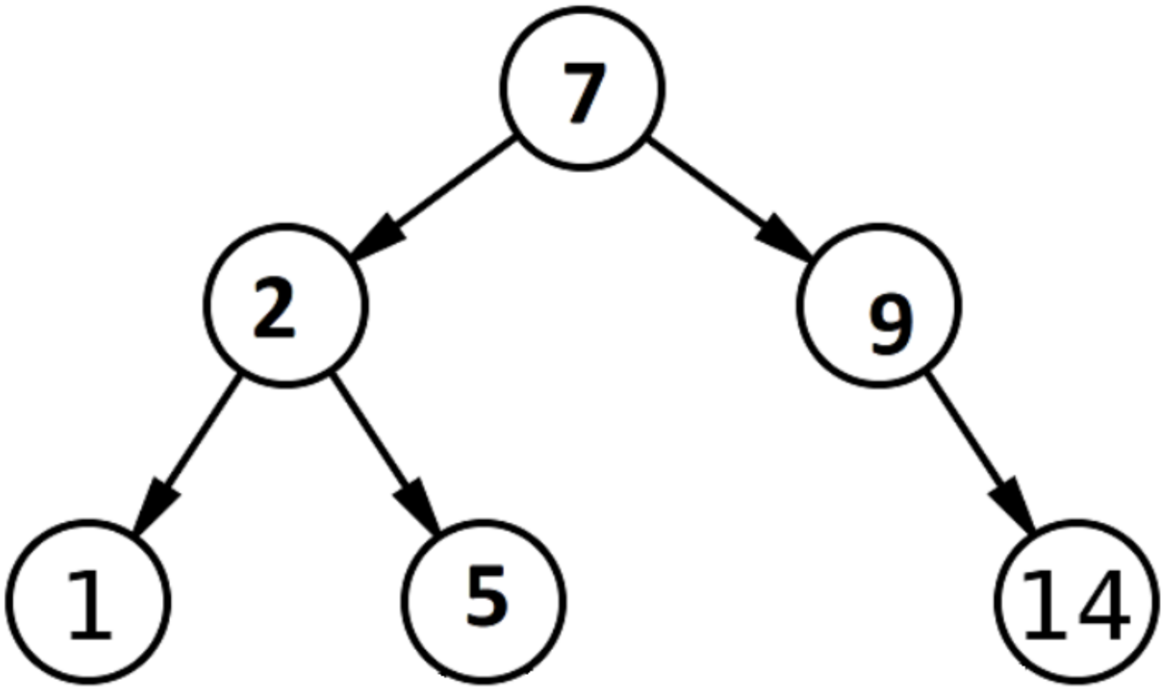
Write a function `is_bst`, which takes a Tree `t` and returns `True` if, and only if, `t` is a valid binary search tree, which means that:

- Each node has at most two children (a leaf is automatically a valid binary search tree) 每个节点至多有 2 孩子
- The children are valid binary search trees 有效二叉搜索树
- For every node, the entries in that node's left child are less than or equal to the label of the node 左边所有的孩子小于等于 label，max(branches[0])≤label
- For every node, the entries in that node's right child are greater than the label of the node 右边所有孩子大于 label, min(branches[-1])>label

Note that, if a node has only one child, that child could be considered either the left or right child. You should take this into consideration.

Hint: It may be helpful to write helper functions `bst_min` and `bst_max` that return the minimum and maximum, respectively, of a Tree if it is a valid binary search tree.

An example of a BST is:



纯文本

```
def is_bst(t):
    """Returns True if the Tree t has the structure of a valid BST.

    >>> t1 = Tree(6, [Tree(2, [Tree(1), Tree(4)]), Tree(7, [Tree(7), Tree(8
    >>> is_bst(t1)
    True
    >>> t2 = Tree(8, [Tree(2, [Tree(9), Tree(1)]), Tree(3, [Tree(6)]), Tree
    >>> is_bst(t2)
    False
    >>> t3 = Tree(6, [Tree(2, [Tree(4), Tree(1)]), Tree(7, [Tree(7), Tree(8
    >>> is_bst(t3)
    False
    >>> t4 = Tree(1, [Tree(2, [Tree(3, [Tree(4)])])])
    >>> is_bst(t4)
    True
    >>> t5 = Tree(1, [Tree(0, [Tree(-1, [Tree(-2)])])])
    >>> is_bst(t5)
    True
    >>> t6 = Tree(1, [Tree(4, [Tree(2, [Tree(3)])])])
    >>> is_bst(t6)
    True
    >>> t7 = Tree(2, [Tree(1, [Tree(5)]), Tree(4)])
    >>> is_bst(t7)
    False
    """
    "*** YOUR CODE HERE ***"
    def bst_min(t): #在bst的基础下找最小，简化过程
        if t.is_leaf():
            return t.label
        return min(t.label,bst_min(t.branches[0])) #0严谨

    def bst_max(t): #在bst的基础下找最大，简化过程
        if t.is_leaf():
            return t.label
        return max(t.label,bst_min(t.branches[-1])) #-1严谨

    if t.is_leaf():
        return True
    elif len(t.branches)==1:
        a=t.branches[0]   #要取出来[0]，得到一个新树
        return is_bst(a) and (bst_min(a)>t.label or bst_max(a)<t.label)
    elif len(t.branches)==2:
        a1,a2=t.branches[0],t.branches[1]
        return (is_bst(a1) and is_bst(a2)) and bst_max(a1) <= t.label and b
    else:
        return False  #排除其他树影响
```

lab9 Q7: Add trees 合并树

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well. 相同位置相加，只在一个树存在也仍写出

纯文本

```
def add_trees(t1, t2):
    """
    >>> numbers = Tree(1,
    ...                [Tree(2,
    ...                      [Tree(3),
    ...                       Tree(4)]),
    ...                 Tree(5,
    ...                      [Tree(6,
    ...                            [Tree(7)]),
    ...                       Tree(8)])])
    >>> print(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print(add_trees(Tree(2), Tree(3, [Tree(4), Tree(5)])))
    5
      4·
      5
    >>> print(add_trees(Tree(2, [Tree(3)]), Tree(2, [Tree(3), Tree(4)])))
    4
      6
      4
    >>> print(add_trees(Tree(2, [Tree(3, [Tree(4), Tree(5)])]), \
    Tree(2, [Tree(3, [Tree(4)]), Tree(5)])))
    4
      6
        8
        5
      5
    """
    if not t1:
        return t2
    if not t2:
        return t1
    new_label = t1.label+t2.label
    t1_branches, t2_branches = list(t1.branches), list(t2.branches) #为什么
    length_t1, length_t2 = len(t1_branches),len(t2_branches)
    if length_t1<length_t2:
        t1_branches += [None for i in t2_branches[length_t1:]] #非常好的融合
    elif length_t1>length_t2:
        t2_branches += [None for i in t1_branches[length_t2:]]
    return Tree(new_label, [add_trees(i,j)for i,j in zip(t1_branches, t2_br
```

Q4: Add Leaves 可变加叶子，每层不一样可考虑 helper

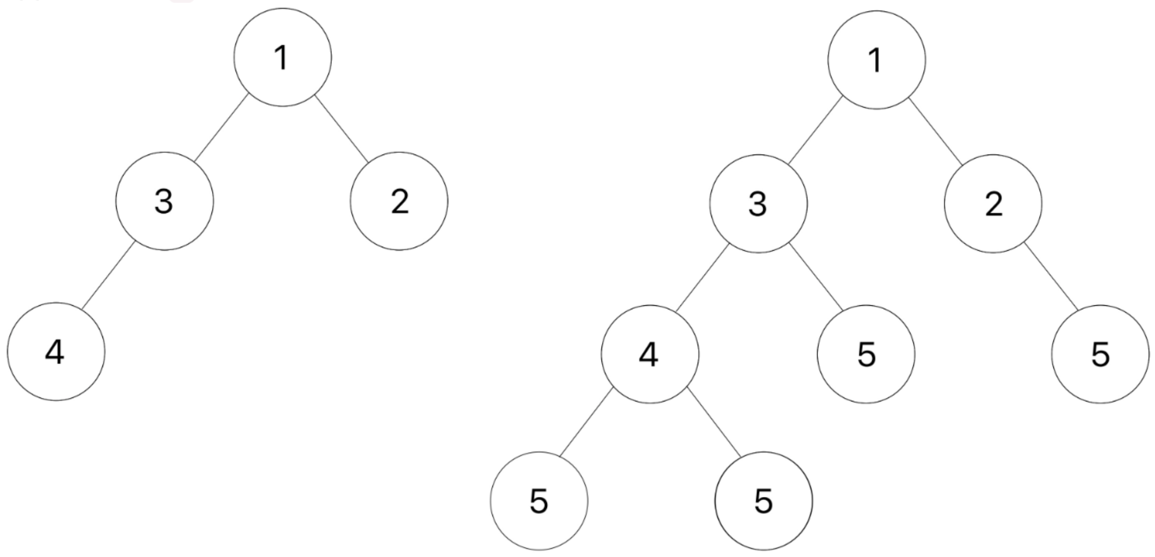
Implement `add_d_leaves`, a function that takes in a `Tree` instance `t` and a number `v`.

We define the depth of a node in `t` to be the number of edges from the root to that node. The depth of root is therefore 0.

For each node in the tree, you should add `d` 个数与层数有关 leaves to it, where `d` is the depth of the node. Every added leaf should have a label of `v`. If the node at this depth has existing branches, you should add these leaves to the end of that list of branches.

For example, you should be adding 1 leaf with label `v` to each node at depth 1, 2 leaves to each node at depth 2, and so on.

Here is an example of a tree `t` (shown on the left) and the result after `add_d_leaves` is applied with `v` as 5.



纯文本

```
def add_d_leaves(t, v):
    """Add d leaves containing v to each node at every depth d.

    >>> t_one_to_four = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> print(t_one_to_four)
    1
      2
      3
        4
    >>> add_d_leaves(t_one_to_four, 5)
    >>> print(t_one_to_four)
    1
      2
        5
      3
        4
          5
          5
        5

    >>> t1 = Tree(1, [Tree(3)])
    >>> add_d_leaves(t1, 4)
    >>> t1
    Tree(1, [Tree(3, [Tree(4)])])
    >>> t2 = Tree(2, [Tree(5), Tree(6)])
    >>> t3 = Tree(3, [t1, Tree(0), t2])
    >>> print(t3)
    3
      1
        3
          4
      0
      2
        5
        6
    >>> add_d_leaves(t3, 10)
    >>> print(t3)
    3
      1
        3
          4
            10
            10
            10
          10
          10
        10
      0
        10
      2
        5
          10
          10
        6
```

```
            10
            10
          10
    """
    "*** YOUR CODE HERE ***"
    def helper(t,d):
        for b in t.branches:
            helper(b,d+1)
        t.branches.extend([Tree(v) for _ in range(d)]) #t.branches是树的列表
    helper(t,0)
```

apply_tree (variation from su_19_final)

```
def apply_tree(fn_tree, val_tree):
    """ Returns a new tree which is the result of applying each function stored
    in fn_tree to the corresponding labels in val_tree
    >>> double = lambda x: x*2
    >>> square = lambda x: x**2
    >>> identity = lambda x: x
    >>> t1 = Tree(double, [Tree(square), Tree(identity)])
    >>> t2 = Tree(6, [Tree(2), Tree(10)])
    >>> apply_tree(t1, t2)
    Tree(12, [Tree(4), Tree(10)])
    """
    new_label = fn_tree.label(val_tree.label)
    new_branches = []
    for i in range(len(fn_tree.branches)):
        new_branches.append(apply_tree(fn_tree.branches[i],
val_tree.branches[i]))
    return Tree(new_label, new_branches)
```

注意函数只对树递归，用 for 循环将算好的 branches append 到一起，树的写法直接 Tree（label，branches）即可

一行代码：return Tree(**fn_tree.label(val_tree.label)**,[apply_tree(fn_tree.branches[i], val_tree.branches[i] for i in range(len(val_tree.branches))])列表表达式可以一行

Trees Warm Up Question (sum_leaves)

```
def sum_leaves(t):
    """
    >>> t1 = Tree(2)
    >>> sum_leaves(t1)
    2
    >>> t2 = Tree(2, [Tree(0), Tree(1), Tree(6)])
    >>> sum_leaves(t2)
    9
    >>> t3 = Tree(2, [Tree(0), t2])
    >>> sum_leaves(t3)
    11
    """
    if t.is_leaf():
        return t.label
    else:
        return sum([sum_leaves(b) for b in t.branches])
```

注意 sum()的括号

Max_path (su19_mt1)

```
def max_path(t, k):
     """
     >>> t1 = Tree(6, [Tree(3, [Tree(8)]), Tree(1, [Tree(9),
     Tree(3)])])
     >>> max_path(t1, 3)
     [6, 3, 8]
     >>> max_path(t1, 2)
     [3, 8]
     >>> t2 = Tree(5, [t1, Tree(7)])
     >>> max_path(t2, 1)
     [9]
     >>> max_path(t2, 2)
     [5, 7]
     >>> max_path(t2, 3)
     [6, 3, 8] """
     def helper(t, k, on_path):
          if k == 0:
               return []
          elif t.is_leaf():
               return [t.label]
          a = [[t.label] + helper(b, k-1, True) for b in t.branches]
          if on_path:
               return max(a, key = sum)
          else:
               b = [helper(b, k, False) for b in t.branches]
```