LINKED LISTS AND TREES SOLUTIONS

CS Scholars

May 2, 2017

1 Linked Lists

Here is the `Link` class, provided for your reference:

```
class Link :
    empty = ()
    def __init__(self, first, rest = empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __repr__( self ):
        if self.rest is Link.empty :
            return "Link({})".format(self.first)
        else:
            return "Link({}, {})".format(self.first,self.rest)
```

Summary:

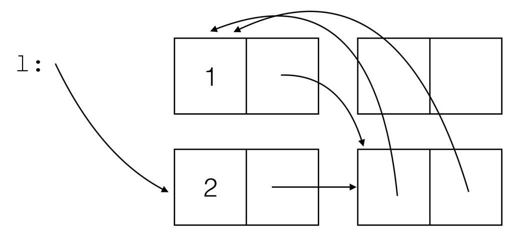
- `.first` : first element (can be number or another linked list)
- `.rest` : rest element (must be another linked list)
- `Link.empty` : empty linked list
- You can alter (mutate) a `Link` by changing a link's `first` value or `rest` pointer.
- Keep in mind if the function you are asked to write returns a *new* `Link` or alters the provided one.
- **Note:** Mutating does not necessarily imply that we return nothing!

1.1 Box and Pointer

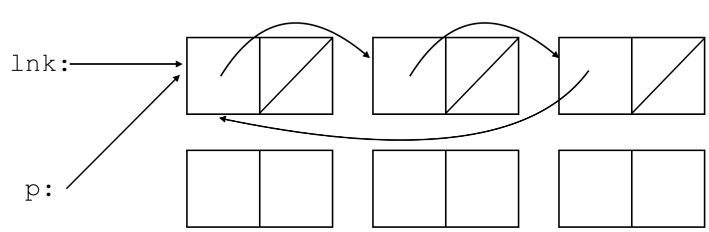
1. Draw a box and pointer diagram that results from executing the code below.

   1. From Brian Hou's Quiz 6

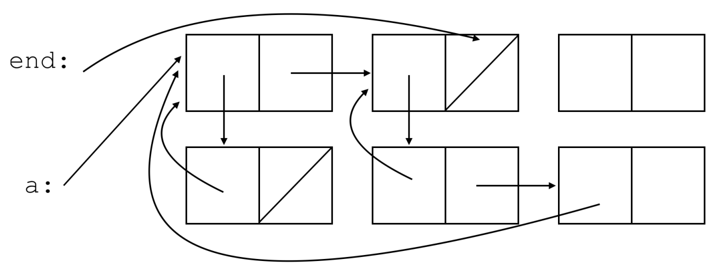
```
l = Link(0)
for e in range(1, 3):
    l = Link(e, Link(l, l))
l.rest.rest.rest = l.rest
```



   2.
```
lnk = Link(1, Link(2, Link(3)))
def m1():
    x = lnk
    def m2(lnk):
        nonlocal x
        if lnk is Link.empty:
            return x
        ret = m2(lnk.rest)
        lnk.first, lnk.rest = x, lnk.empty
        x = lnk
        return ret
    return m2
p = m1()(lnk)
```



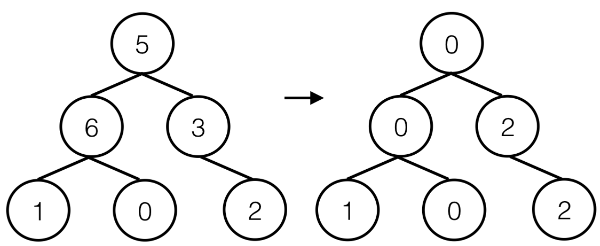
3.
```
a = Link(1, Link(2))
def x(lnk):
    if lnk is Link.empty:
        return lnk
    y(lnk)
    z = x(lnk.rest)
    lnk.first = Link(lnk, lnk.first)
    return z

def y(lnk):
    b = a
    lnk.first = Link.empty
    while b != lnk:
        lnk.first = Link(b, lnk.first)
        b = b.rest
    return lnk.first

end = x(a)
```



Here are the implementations of `Tree` and `Binary Tree`:

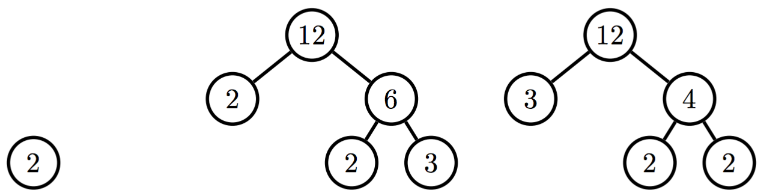
```
class Tree:
    def __init__(self, label, branches=[]):
        for c in branches:
            assert isinstance(c, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches

class BinTree:
    empty = ()
    def __init__(self, label, left=empty, right=empty):
        self.label = label
        self.left = left
        self.right = right
```

1. Implement a function `min_tree`, which takes a tree t. It returns a new tree with the exact same structure as `t`; at each node in the new tree, the entry is the **smallest** number that is contained in that node's subtrees or the corresponding node in `t`. Here is an example input and output:



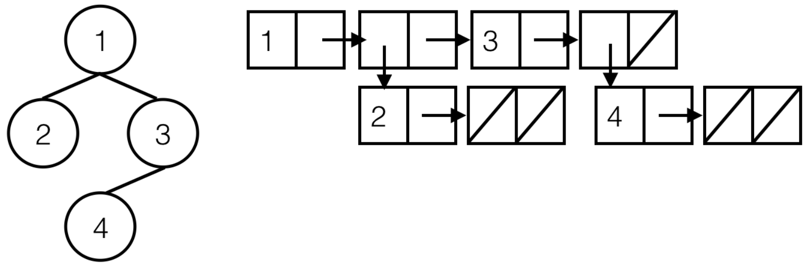

```
def min_tree(t):

   if t.is_leaf():

      return Tree(t.label)

   mins = [min_tree(b) for b in t.branches]

   return Tree(min([b.label for b in mins] + [t.label], mins)
```

2. (From Brian Hou's Quiz 6) We can represent the factorization of a number with a full binary tree, a tree that has either two subtrees or none at all. Implement make factor tree, which takes in an integer `n` that is greater than one and returns a tree that factors `n`.
   Example factor trees for 2 and 12 are shown below. The product of all leaves of a factor tree must be `n`. There may be multiple valid factor trees.




```
def factor(x):
    // returns a factor of x or False if the only factors are 1
        and x

def make_factor_tree(n):
    """
    >>> six = make_factor_tree(6)
    >>> print(six.branches[0].label, six.branches[1].label)
    2 3
    >>> two = make_factor_tree(2)
    >>> print(two.label, two.is_leaf())
    2 True
    """
    fact = factor(n)

    if fact:

        return BinTree(n, make_factor_tree(fact),
           make_factor_tree(n//fact)

    return BinTree(n)
```

3. Write a function that converts a Binary Tree to a Linked List, as shown:




```
def convert(t):

    if t is BinTree.empty:

        return Link.empty

    right = convert(t.right)

    left = convert(t.left)

    return Link(t.label, Link(left, right))
```

4. (From Summer 2016 Final) **Caught-Ya**
Implement the function catch up, which takes in two linked lists of integers lnk1 and lnk2 and mutates the linked list with the lower sum by repeatedly inserting 1 at the end until the sums are equal. See the doctests for details. You may assume that the two linked lists that are passed in are non-empty and have the same length. The Link class is provided for your reference. Hint: You may need the ternary operator if else.

```
def catch_up(lnk1, lnk2 ):
    """
    >>> odds = Link (1, Link(3, Link(5, Link(7))))
    >>> evens = Link(2, Link(4, Link(6, Link(8))))
    >>> catch_up(odds, evens )
    >>> print(odds) # odds is mutated
    <1 3 5 7 1 1 1 1 >
    >>> print(evens)
    <2 4 6 8 >
    """
    def catcher(link1, link2, sum1, sum2):

        sum1 = sum1 + link1.first

        sum2 = sum2 + link2.first

        if link1.rest is Link.empty:

            lower = link1 if sum1 < sum2 else link2

            for _ in range(abs(sum1 - sum2)):

                lower.rest = Link(1)

                lower = lower.rest

            ____________________________________

        else:
            catcher(link1.rest, link2.rest, sum1, sum2)

        catcher(lnk1, lnk2, 0, 0)
```

5. Define the function min leaf depth, which takes in a tree t and returns the minimum depth of any of the leaves in t. Recall that the depth of a node is defined as how far away that node is from the root. See the doctests for details.
Hint: You may find the built-in min function useful.

```
def min_leaf_depth ( t ):
    """
    >>> t1 = Tree(2)
    >>> min_leaf_depth (t1)
    0
    >>> t2 = Tree(2, [Tree(0), Tree(1), Tree(6)])
    >>> min_leaf_depth(t2)
    1
    >>> t3 = Tree(2, [Tree(0), t2])
    >>> min_leaf_depth(t3)
    1
    >>> t4 = Tree(2, [t2, t3])
    >>> min_leaf_depth(t4)
    2
    """
    if t.is_leaf():

        return 0

    else:

        c_depths = [min_leaf_depth(b) for b in t.branches]

        return 1 + min(c_depths)
```