Lab 8 Solutions **lab08.zip (lab08.zip)**

Solution Files

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Required Questions

Linked Lists

Q1: WWPD: Linked Lists

Read over the `Link` class in `lab08.py`. Make sure you understand the doctests.

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q link -u
```

Enter `Function` if you believe the answer is `<function ...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

If you get stuck, try drawing out the box-and-pointer diagram for the linked list on a piece of paper or loading the `Link` class into the interpreter with `python3 -i lab08.py`.

```
>>> from lab08 import *
>>> link = Link(1000)
>>> link.first
------

>>> link.rest is Link.empty
------

>>> link = Link(1000, 2000)
------

>>> link = Link(1000, Link())
------
```

```
>>> from lab08 import *
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
------

>>> link.rest.first
------

>>> link.rest.rest.rest is Link.empty
------

>>> link.first = 9001
>>> link.first
------

>>> link.rest = link.rest.rest
>>> link.rest.first
------

>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
------

>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.first
------

>>> link2.rest.first
------
```

```
>>> from lab08 import *
>>> link = Link(5, Link(6, Link(7)))
>>> link                  # Look at the __repr__ method of Link
------

>>> print(link)          # Look at the __str__ method of Link
------
```

Q2: Convert Link

Write a function `convert_link` that takes in a linked list and returns the sequence as a Python list. You may assume that the input list is shallow; that is none of the elements is another linked list.

Try to find both an iterative and recursive solution for this problem!

```
def convert_link(link):
    """Takes a linked list and returns a Python list with the same elements.

    >>> link = Link(1, Link(2, Link(3, Link(4))))
    >>> convert_link(link)
    [1, 2, 3, 4]
    >>> convert_link(Link.empty)
    []
    """
    # Recursive solution
    if link is Link.empty:
        return []
    return [link.first] + convert_link(link.rest)

# Iterative solution
def convert_link_iterative(link):
    result = []
    while link is not Link.empty:
        result.append(link.first)
        link = link.rest
    return result
```

Video Walkthrough:
YouTube link (https://youtu.be/hdO9Ry8d5FU?t=25m6s)

Use Ok to test your code:

```
python3 ok -q convert_link
```

Walkthrough:

YouTube link (https://youtu.be/hdO9Ry8d5FU?t=25m6s)

Trees

Q3: WWPD: Trees

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q trees-wwpd -u
```

Enter `Function` if you believe the answer is `<function ...>`, `Error` if it errors, and `Nothing` if nothing is displayed. Recall that `Tree` instances will be displayed the same way they are constructed (/lab/lab08/#mutable-trees).

```
>>> from lab08 import *
>>> t = Tree(1, Tree(2))
------

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

>>> t.branches[0]
------

>>> t.branches[0].label
------

>>> t.label = t.branches[0].label
>>> t
------

>>> t.branches.append(Tree(4, [Tree(8)]))
>>> len(t.branches)
------

>>> t.branches[0]
------

>>> t.branches[1]
------
```

Q4: Square

Write a function `label_squarer` that mutates a `Tree` with numerical labels so that each label is squared.

```
def label_squarer(t):
    """Mutates a Tree t by squaring all its elements.

    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> label_squarer(t)
    >>> t
    Tree(1, [Tree(9, [Tree(25)]), Tree(49)])
    """
    t.label = t.label ** 2
    for subtree in t.branches:
        label_squarer(subtree)
```

Use Ok to test your code:

```
python3 ok -q label_squarer
```

Q5: 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.

```
def cumulative_mul(t):
    """Mutates t so that each node's label becomes the product of all labels in
    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)])
    """
    for b in t.branches:
        cumulative_mul(b)
    total = t.label
    for b in t.branches:
        total *= b.label
    t.label = total
```

Use Ok to test your code:

```
python3 ok -q cumulative_mul
```

Q6: Add Leaves

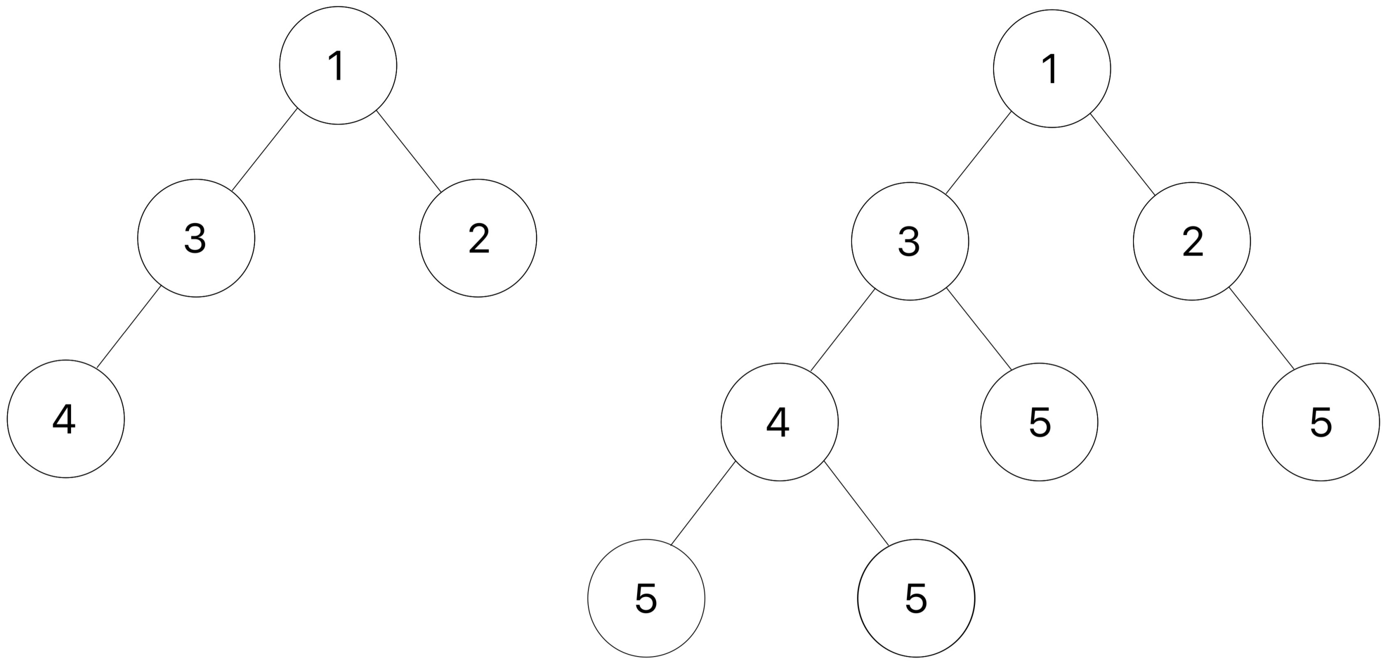
Implement `add_d_leaves`, a function that takes in a `Tree` instance `t` and a number `v`.

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` 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.



Hint: Use a helper function to keep track of the depth!

```
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
        """
        def add_leaves(t, d):
            for b in t.branches:
                add_leaves(b, d + 1)
            t.branches.extend([Tree(v) for _ in range(d)])
        add_leaves(t, 0)
```

Use Ok to test your code:

```
python3 ok -q add_d_leaves
```

A first thought is to recursively call `add_d_leaves` on each branch to add the leaves to the branches. However, notice that each recursive call would need to know what the current depth is in order to add that many leaves. We could try initializing a variable within the body of the function, but by now we know that in order to keep track of changing values across recursive calls we should use a helper function!

The helper function should take in a tree and a depth value, and we will define it as a function that adds `d` leaves to the branches of the root node, `d + 1` leaves to the branches of each of the root node's branches, and so on:

```
def add_leaves(t, d):
    """Adds a number of leaves to each node in t equivalent to the depth of
    the node, assuming that the root node is at depth d, the children of
    the root node are at depth d + 1, and so on."""
    ...
```

We don't need a parameter for `v` since that value won't change and we can access it from the parent frame. With this function defined as such, we can call `add_leaves` with arguments `t` and 0 to add leaves starting at depth 0.

```
def add_d_leaves(t, v):
    def add_leaves(t, d):
        """Adds a number of leaves to each node in t equivalent to the depth of
        the node, assuming that the root node is at depth d, the children of
        the root node are at depth d + 1, and so on."""
        ...
    add_leaves(t, 0)
```

Inside the helper function, we can now call it recursively on each branch. Each node's branch is one depth level greater than the node itself, so we should update `d` to `d + 1`:

```
def add_leaves(t, d):
    for b in t.branches:
        add_leaves(b, d + 1)
    ...
```

Now that we've made these recursive calls, let's take a step back and look at our progress. Taking the leap of faith, we know that each recursive call should've successfully added the correct number of leaves at each node in each branch. That means that the only step left is to add the correct number of leaves to the current node!

The parameter `d` tells us how many leaves to add at this node. Since we are mutating `t` to add these leaves, we need to mutate the list of `t`'s branches. We know a few different ways to mutatively add elements to a list: `insert`, `append`, and `extend`. Which one makes most sense to use here? Well, we know that we have to add `d` elements to the end of `t.branches`. Index doesn't matter so we can rule out `insert`. `append` is good for adding a single element, while `extend` is useful for adding multiple elements contained in a list, so let's use `extend`!

The input to `extend` should be a list, so how do we create a list with the leaves that we need? The most concise way is with a list comprehension. To create each leaf, we call `Tree(v)`:

```
[Tree(v) for _ in range(d)]
```

Now, we just have to extend `t.branches` by this list:

```
def add_leaves(t, d):
    for b in t.branches:
        add_leaves(b, d + 1)
    t.branches.extend([Tree(v) for _ in range(d)])
```

Do we need an explicitly base case? Let's take a look at what happens when `t` is a leaf. In that case, `t.branches` would be an empty list, so we would not enter the `for` loop. Then, the function will extend `t.branches`, which is an empty list, by a list containing the new leaves. This is exactly the desired result, so no base case is needed!

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q7: Every Other

Implement `every_other`, which takes a linked list `s`. It mutates `s` such that all of the odd-indexed elements (using 0-based indexing) are removed from the list. For example:

```
>>> s = Link('a', Link('b', Link('c', Link('d'))))
>>> every_other(s)
>>> s.first
'a'
>>> s.rest.first
'c'
>>> s.rest.rest is Link.empty
True
```

If `s` contains fewer than two elements, `s` remains unchanged.

> Do not return anything! `every_other` should mutate the original list.

```
def every_other(s):
    """Mutates a linked list so that all the odd-indiced elements are removed
    (using 0-based indexing).

    >>> s = Link(1, Link(2, Link(3, Link(4))))
    >>> every_other(s)
    >>> s
    Link(1, Link(3))
    >>> odd_length = Link(5, Link(3, Link(1)))
    >>> every_other(odd_length)
    >>> odd_length
    Link(5, Link(1))
    >>> singleton = Link(4)
    >>> every_other(singleton)
    >>> singleton
    Link(4)
    """
    if s is Link.empty or s.rest is Link.empty:
        return
    else:
        s.rest = s.rest.rest
        every_other(s.rest)
```

Use Ok to test your code:

```
python3 ok -q every_other
```

Q8: Prune Small

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.

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

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

Use Ok to test your code:

```
python3 ok -q prune_small
```