Lab 8: Linked Lists, Mutable Trees

lab08.zip (lab08.zip)

Due by 11:59pm on Wednesday, October 19.

Starter Files

Download lab08.zip (lab08.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

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.

Linked Lists

Linked Lists

We've learned that a Python list is one way to store sequential values. Another type of list is a linked list. A Python list stores all of its elements in a single object, and each element can be accessed by using its index. A linked list, on the other hand, is a recursive object that only stores two things: its first value and a reference to the rest of the list, which is another linked list.

We can implement a class, `Link`, that represents a linked list object. Each instance of `Link` has two instance attributes, `first` and `rest`.

```
class Link:
    """A linked list.

    >>> s = Link(1)
    >>> s.first
    1
    >>> s.rest is Link.empty
    True
    >>> s = Link(2, Link(3, Link(4)))
    >>> s.first = 5
    >>> s.rest.first = 6
    >>> s.rest.rest = Link.empty
    >>> s                                    # Displays the contents of repr(s)
    Link(5, Link(6))
    >>> s.rest = Link(7, Link(Link(8, Link(9))))
    >>> s
    Link(5, Link(7, Link(Link(8, Link(9)))))
    >>> print(s)                             # Prints str(s)
    <5 7 <8 9>>
    """
    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 not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

A valid linked list can be one of the following:

1. An empty linked list (`Link.empty`)
2. A `Link` object containing the first value of the linked list and a reference to the rest of the linked list

What makes a linked list recursive is that the `rest` attribute of a single `Link` instance is another linked list! In the big picture, each `Link` instance stores a single value of the list. When multiple `Link`s are linked together through each instance's `rest` attribute, an entire sequence is formed.

> *Note*: This definition means that the `rest` attribute of any `Link` instance *must* be either `Link.empty` or another `Link` instance! This is enforced in `Link.__init__`, which raises an `AssertionError` if the value passed in for `rest` is neither of these things.

To check if a linked list is empty, compare it against the class attribute `Link.empty`. For example, the function below prints out whether or not the link it is handed is empty:

```
def test_empty(link):
    if link is Link.empty:
        print('This linked list is empty!')
    else:
        print('This linked list is not empty!')
```

Mutable Trees

Mutable 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.

```
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.**

Here is a summary of the differences between the tree data abstraction implemented as a functional abstraction vs. implemented as class:

-	**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.

Implementing trees as a class gives us another advantage: we can specify how we want them to be output by the interpreter by implementing the `__repr__` and `__str__` methods.

Here is the `__repr__` method:

```
def __repr__(self):
    if self.branches:
        branch_str = ', ' + repr(self.branches)
    else:
        branch_str = ''
    return 'Tree({0}{1})'.format(self.label, branch_str)
```

With this implementation of `__repr__`, a `Tree` instance is displayed as the exact constructor call that created it:

```
>>> t = Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> t
Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> t.branches
[Tree(3), Tree(5, [Tree(6)]), Tree(7)]
>>> t.branches[0]
Tree(3)
>>> t.branches[1]
Tree(5, [Tree(6)])
```

Here is the `__str__` method. You do not need to understand how this function is implemented.

```
def __str__(self):
    def print_tree(t, indent=0):
        tree_str = '  ' * indent + str(t.label) + "\n"
        for b in t.branches:
            tree_str += print_tree(b, indent + 1)
        return tree_str
    return print_tree(self).rstrip()
```

With this implementation of `__str__`, we can pretty-print a `Tree` to see both its contents and structure:

```
>>> t = Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> print(t)
4
  3
  5
    6
  7
>>> print(t.branches[0])
3
>>> print(t.branches[1])
5
  6
```

Required Questions

Getting Started Videos

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 is Link.empty
------

>>> 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!

Challenge: You may NOT assume that the input list is shallow. Hint: use the `type` built-in.

```
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)
    []
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q convert_link
```

Q3: Duplicate Link

Write a function `duplicate_link` that takes in a linked list `link` and a `value`. `duplicate_link` will mutate `link` such that if there is a linked list node that has a `first` equal to `value`, that node will be duplicated. **Note that** you should be mutating the original link list `link`; you will need to create new `Link`s, but you should not be returning a new linked list.

> **Note**: in order to insert a link into a linked list, you need to modify the `.rest` of certain links. We encourage you to draw out a doctest to visualize!

```
def duplicate_link(link, val):
    """Mutates `link` such that if there is a linked list
    node that has a first equal to value, that node will
    be duplicated. Note that you should be mutating the
    original link list.

    >>> x = Link(5, Link(4, Link(3)))
    >>> duplicate_link(x, 5)
    >>> x
    Link(5, Link(5, Link(4, Link(3))))
    >>> y = Link(2, Link(4, Link(6, Link(8))))
    >>> duplicate_link(y, 10)
    >>> y
    Link(2, Link(4, Link(6, Link(8))))
    >>> z = Link(1, Link(2, (Link(2, Link(3)))))
    >>> duplicate_link(z, 2) #ensures that back to back links with val are both duplic
    >>> z
    Link(1, Link(2, Link(2, Link(2, Link(2, Link(3))))))
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q duplicate_link
```

Trees

Q4: WWPD: Trees

Read over the `Tree` 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 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.

```
>>> 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]
------
```

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.

> **Hint**: Consider carefully when to do the mutation of the tree and whether that mutation should happen before or after processing the subtrees.

```
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)])
    >>> otherTree = Tree(2, [Tree(1, [Tree(3), Tree(4), Tree(5)]), Tree(6, [Tree(7)])])
    >>> cumulative_mul(otherTree)
    >>> otherTree
    Tree(5040, [Tree(60, [Tree(3), Tree(4), Tree(5)]), Tree(42, [Tree(7)])])
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q cumulative_mul
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q6: 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)
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q every_other
```

Q7: 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 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, [Tree(3),
    >>> prune_small(t3, 2)
    >>> t3
    Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2)])])
    """
    while _____________________________:
        largest = max(_________________, key=____________________)
        _________________________
    for __ in ______________:
        ____________________
```

Use Ok to test your code:

```
python3 ok -q prune_small
```