Lab 8: Linked Lists, Mutable Trees

lab08.zip (lab08.zip)

Due by 11:59pm on Thursday, July 22.

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

> **Minilecture Video: Linked Lists (https://youtu.be/pq8gEHEtvA4)**

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

> **Minilecture Video: Mutable Trees (https://youtu.be/FzOwLI-M-LY)**

Recall that a tree is a recursive abstract data type that has a `label` (the value stored in the root of the tree) and `branches` (a list of trees directly underneath the root).

We saw one way to implement the tree ADT -- using constructor and selector functions that treat trees as lists. Another, more formal, way to implement the tree ADT is with a class. Here is part of the class definition for `Tree` , which can be found in `lab07.py` :

```
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 tree ADT remains true. That means that solving problems involving trees as objects uses the same techniques that we developed when first studying the tree ADT (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 ADT implemented using functions and lists vs. implemented using a 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 tree ADT 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.

Required Questions

WWPD: Linked Lists

Q1: WWPD: Linked Lists

Minilecture Video: Linked Lists (https://youtu.be/pq8gEHEtvA4)

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`.

Hint Video

```
>>> 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
------
```

Linked Lists

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

Use Ok to test your code:

```
python3 ok -q convert_link
```

Hint Video

WWPD: 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 (/~cs61a/su21/lab/lab09/#displaying-tree-instances).

Hint Video

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

Trees

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

Use Ok to test your code:

```
python3 ok -q label_squarer
```

Hint Video

Q5: Cumulative Mul

> **Minilecture Video: Mutable Trees (https://youtu.be/FzOwLI-M-LY)**

Write a function `cumulative_mul` that mutates the Tree `t` so that each node's label becomes the product of all labels in the subtree 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)])
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q cumulative_mul
```

Hint Video

Q6: Add Leaves

Implement `add_d_leaves`, a function that takes in a `Tree` instance `t` and mutates it so that at each depth `d` in the tree, `d` leaves with labels `v` are added to each node at that depth. For example, we want to add 1 leaf with `v` in it to each node at depth 1, 2 leaves to each node at depth 2, and so on.

Recall that the depth of a node is the number of edges from that node to the root, so the depth of the root is 0. The leaves should be added to the end of the list of branches.

> **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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q add_d_leaves
```

Hint Video

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q7: Cycles

The `Link` class can represent lists with cycles. That is, a list may contain itself as a sublist.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.rest.rest.rest = s
>>> s.rest.rest.rest.rest.rest.first
3
```

Implement `has_cycle` ,that returns whether its argument, a `Link` instance, contains a cycle.

> *Hint*: Iterate through the linked list and try keeping track of which `Link` objects you've already seen.

```
def has_cycle(link):
    """Return whether link contains a cycle.

    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    >>> t = Link(1, Link(2, Link(3)))
    >>> has_cycle(t)
    False
    >>> u = Link(2, Link(2, Link(2)))
    >>> has_cycle(u)
    False
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q has_cycle
```

As an extra challenge, implement `has_cycle_constant` with only constant space (http://composingprograms.com/pages/28-efficiency.html#growth-categories). (If you followed the hint above, you will use linear space.) The solution is short (less than 20 lines of code), but requires a clever idea. Try to discover the solution yourself before asking around:

```
def has_cycle_constant(link):
    """Return whether link contains a cycle.

    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle_constant(s)
    True
    >>> t = Link(1, Link(2, Link(3)))
    >>> has_cycle_constant(t)
    False
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q has_cycle_constant
```

Q8: 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
```

Q9: 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
    >>> 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
```

Q10: Reverse Other

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)])]), Tre
    >>> reverse_other(t)
    >>> t
    Tree(1, [Tree(8, [Tree(3, [Tree(5), Tree(4)]), Tree(6, [Tree(7)])]), Tree(2)])
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q reverse_other
```