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:
    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.

Q1: WWPD: Trees

What would Python display?

```
>>> t = Tree(1, Tree(2))
```

Error

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

1

"Tree(4, [Tree(2)])"

```
>>> t.label *= 4
>>> t
```

>>> obj ⇔ >>> print(repr(obj))

Tree(4, [Tree(2)])

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

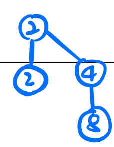
Tree(2)

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

2

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

Tree(2, [Tree(2)])




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

2

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

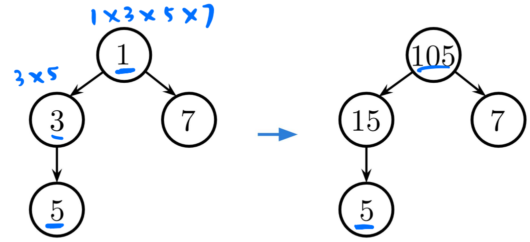
Tree(2)

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

Tree(4, [Tree(8)])

Q2: 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.

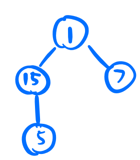


>>> cumulative_mul(t)

Think carefully about whether the mutation of tree 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 ***"
    for b in t.branches:
        cumulative_mul(b)
        t.label *= b.label
```




```
# You can use more space on the back if you want
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

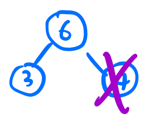
Q3: Prune Small

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.

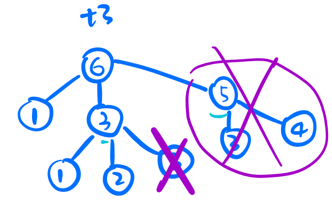
```
def prune_small(t, n):
    """Prune the tree mutatively, keeping only the n branches
    of each node with the smallest labels.

    >>> 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),
    Tree(4)])])
    >>> 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 b: b.label)
        t.branches.remove(largest)
    for b in t.branches:
        prune_small(b, n)
```

6



t3



lst.remove(elem) → remove the 1st occurence of elem from the lst

lst.pop(i) → remove & return the element at index i

← when to keep pruning?

← find the largest branch

prune branches of t

recursively prune branch's branches

max(iterable, key = lambda x: ____)

max([1, 2, 3], key = lambda x: -x) → 1

-1 -2 -3

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:
    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!

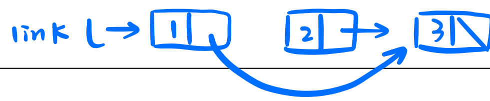
To check if a linked list is empty, compare it against the class attribute `Link.empty`.

Q4: WWPD: Linked Lists

What would Python display? If you get stuck, try drawing out or visualizing the box-and-pointer diagram!

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

1




```
>>> link.rest.first
```

2

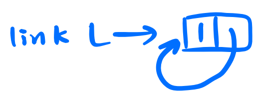
```
>>> link.rest.rest.rest is Link.empty
```

True

```
>>> link.rest = link.rest.rest
>>> link.rest.first
```

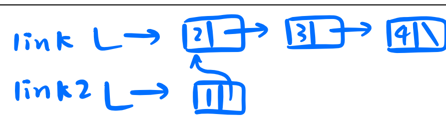
3

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



1

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



2

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

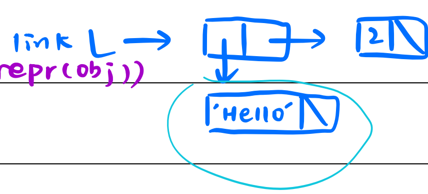
Error

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

Error

```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first
```

>>> obj <=> >>> print(repr(obj))



Link('Hello')

```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first.rest is Link.Empty
```

True

Q5: 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: Use the `type` built-in to implement a solution for the case where the list may not be shallow!

```
def convert_link(link):
    """Takes a linked list and returns a Python list with the same elements.
      link  [1]→ [2]→ [3]→ [4\]
    >>> link = Link(1, Link(2, Link(3, Link(4))))
    >>> convert_link(link)
    [1, 2, 3, 4]                    [1, 2] [3]
    >>> convert_link(Link.empty)
    []                 1, 2, 3, 4
    """
    "*** YOUR CODE HERE ***"
```

recursive :

```
if link is Link.empty:
    return []
res = convert_link(link.rest)
return [link.first] + res
```

[2, 3, 4]

[1]

iterative :

```
lst = []
while link is not Link.empty:
    lst.append(link.first)
    link = link.rest
return lst
```

```
# You can use more space on the back if you want
```

Q6: 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
    duplicated
    >>> z
    Link(1, Link(2, Link(2, Link(2, Link(2, Link(3))))))
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

Q7: Remove All

Implement a function `remove_all` that takes a `Link`, and a `value`, and remove any linked list node containing that value. You can assume the list already has at least one node containing `value` and the first element is never removed. Notice that you are not returning anything, so you should mutate the list.

Note: Can you create a recursive and iterative solution for `remove_all`?

```
def remove_all(link, value):
    """Remove all the nodes containing value in link. Assume that the
    first element is never removed.

    >>> l1 = Link(0, Link(2, Link(2, Link(3, Link(1, Link(2, Link(3)))))))
    >>> print(l1)
    <0 2 2 3 1 2 3>
    >>> remove_all(l1, 2)
    >>> print(l1)
    <0 3 1 3>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    """
    "*** YOUR CODE HERE ***"















# You can use more space on the back if you want
```