LINKED LISTS AND MUTABLE TREES

CS 61A

July 23, 2021

1 Linked Lists

Linked lists consists of a series of links which have two attributes: `first` and `rest`. The `first` attribute contains the value of the link (which can be an integer, string, list, even another linked list!). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just an empty linked list represented traditionally by an empty tuple (but not necessarily, so never assume that it is represented by an empty tuple otherwise you will break an abstraction barrier!).

Because each link contains another link or `Link.empty`, linked lists lend themselves to recursion (just like trees). Consider the following example, in which we double every value in linked list. We mutate the current link and then recursively double the rest.

```
def double_values(link):
     if link is not Link.empty:
        link.first *= 2 # we mutate the value inside of the link
        double_val(link.rest) # we mutate the values in the rest
                              # of the linked list
    # if the link is empty then do nothing
```

However, unlike with trees, we can also solve many linked list questions using iteration. Take the following example where we have written `double_values` using a while loop instead of using recursion:

```
def double_values_iter(link):
    while link is not Link.empty:
        link.first *= 2
        link = link.rest # Note that this does not mutate
                         # the original linked list;
                         # it changes what link the variable
                         # link is pointing to
```

For each of the following problems, assume linked lists are defined as follows:

```
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) + '>'
```

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

```
if link is Link.empty:
    print('This linked list is empty!')
```

1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```
>>> a = Link(1, Link(2, Link(3)))

>>> a.first




>>> a.first = 5

>>> a.first




>>> a.rest.first




>>> a.rest.rest.rest.rest.first
```

```
>>> a.rest.rest.rest = a

>>> a.rest.rest.rest.rest.first



>>> repr(Link(1, Link(2, Link(3, Link.empty))))



>>> Link(1, Link(2, Link(3, Link.empty)))



>>> str(Link(1, Link(2, Link(3))))



>>> print(Link(Link(1), Link(2, Link(3))))
```

2. Write a function `skip`, which takes in a `Link` and returns a new `Link` with every other element skipped.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
    if ____________________________________________:

        ____________________________________________

    elif __________________________________________:

        ____________________________________________

    ________________________________________________
```

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> skip(a)
    >>> a
    Link(1, Link(3))
    """
```

4. **(Optional)** Write `has_cycle` which takes in a `Link` and returns `True` if and only if there is a cycle in the `Link`. Note that the cycle may start at any node and be of any length. Try writing a solution that keeps track of all the links we've seen. Then try to write a solution that doesn't store those witnessed links (consider using two pointers!).

```
def has_cycle(s):
    """
    >>> has_cycle(Link.empty)
    False
    >>> a = Link(1, Link(2, Link(3)))
    >>> has_cycle(a)
    False
    >>> a.rest.rest.rest = a
    >>> has_cycle(a)
    True
    """
```

2 Mutable Trees

The difference between the Tree class and the Tree abstract data type (using functions)

- Using the constructor: Capital T for tree class and lower-case t for tree ADT

  `t = Tree(1)#class` vs. `t = tree(1)#adt functions`

- hi `label` and `branches` are now attributes and, `is_leaf()` is a method of the class instead of them all being functions.

  `t.label vs. label(t)`

  `t.branches vs. branches(t)`

  `t.is_leaf()vs. is_leaf(t)`

- A tree object is mutable while tree ADT is not mutable

  `t.label = 2` vs. `label(t)= 2 #this would error`

  This means we can mutate values in the tree object instead of making a new tree that we return. In other words, we can solve tree class problems non-destructively and destructively, but can only solve tree ADT problems non-destructively

Besides these differences, we use the same approach and ideas from ADT trees and apply them to Tree class including problem solving (base case, recursive calls, how to solve) and respecting abstraction barrier. For the following problems, use this definition for the Tree class:

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

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

1. Implement `tree_sum` which takes in a Tree object and replaces the label of the tree with the sum of all the values in the tree. `tree_sum` should also return the new label.

```
def tree_sum(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4)])
    >>> tree_sum(t)
    10
    >>> t.label
    10
    >>> t.branches[0].label
    5
    >>> t.branches[1].label
    4
    """
```

2. Define delete_path_duplicates, which takes in t, a tree with non-negative labels. If there are any duplicate labels on any path from root to leaf, the function should mutate the label of the occurrences deeper in the tree (i.e. farther from the root) to be the value -1.

```
def delete_path_duplicates(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(1), Tree(1)])])
    >>> delete_path_duplicates(t)
    >>> t
    Tree(1, [Tree(2, [Tree(-1), Tree(-1)])])
    >>> t2 = Tree(1, [Tree(2), Tree(2, [Tree(2, [Tree(1, [Tree
       (5)])])])])
    >>> delete_path_duplicates(t2)
    >>> t2
    Tree(1, [Tree(2), Tree(2, [Tree(-1, [Tree(-1, [Tree(5)])])
       ])])
    """
    def helper(_______________, _______________):

        if ________________________________:

            ________________________________

        else:

            ________________________________

        for _______ in ____________________:

            ________________________________

    ________________________________________
```

3. Given a tree t, mutate the tree so that each leaf's label becomes the sum of the labels of all nodes in the path from the leaf node to the root node.

```
def replace_leaves_sum(t):
    """
```

```
    >>> t = Tree(1, [Tree(3, [Tree(2), Tree(8)]), Tree(5)])
    >>> replace_leaves_sum(t)
    >>> t
    Tree(1, [Tree(3, [Tree(6), Tree(12)]), Tree(6)])
    """
    def helper(______________ , _________________):

        if t.is_leaf():

            _____________________________________

        for b in t.branches:

            _________________________________

    _____________________________________________
```

4. Write a function that returns true only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ___________________________________________:

        return ______________________________________

    elif ___________________________________________:

        return ______________________________________

    else:

        return ______________________________________
```