Linked Lists and Mutable Trees Solutions

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:

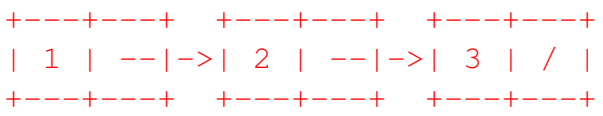
```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    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`:

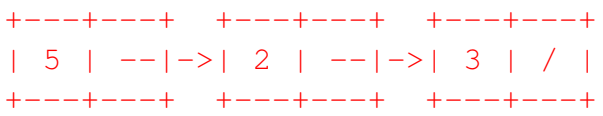
```
if link is Link.empty:
    print('This linked list is empty!')
```

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

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

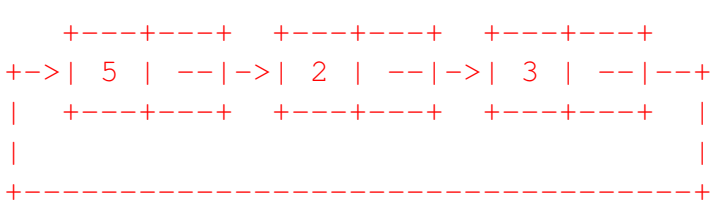



```
>>> a.first
1

>>> a.first = 5
```




```
>>> a.first
5

>>> a.rest.first
2

>>> a.rest.rest.rest.rest.first
Error: tuple object has no attribute rest (Link.empty has no rest)
```

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




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

2


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

"Link(1, Link(2, Link(3)))"


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

Link(1, Link(2, Link(3)))


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

'<1 2 3>'


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

<<1> 2 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 _____:

        _____

    _____
```

```
    if lst is Link.empty
        return Link.empty
     elif lst.rest is Link.empty:
        return Link(lst.first)
    return Link(lst.first, skip(lst.rest.rest))
```

Base cases:

- When the linked list is empty, we want to return a new Link.empty.
- If there is only one element in the linked list (aka the next element is empty), we want to return a new linked list with that single element.

Recursive case:
All other longer linked lists can be reduced down to either a single element or empty linked list depending on whether it has odd or even length. Therefore, we want to keep the first element, and recurse on the element after the next (skipping the immediate next element with `lst.rest.rest`). To build a new linked list, we can add new links to the end of the linked list by calling skip recursively inside the `rest` argument of the `Link` constructor.

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))
    """
```

```
def skip(lst): # Recursively
    if lst is Link.empty or lst.rest is Link.empty:
        return
    lst.rest = lst.rest.rest
    skip(lst.rest)
```

```
def skip(lst): # Iteratively
    while lst is not Link.empty and lst.rest is not Link.empty
        :
        lst.rest = lst.rest.rest
        lst = lst.rest
```

Because this problem is mutative, we should never be creating a new list - we should never have `Link(x)`, or the creation of a new Link instance, anywhere in our code! Instead, we'll be reassigning `lst.rest`.

In order to skip a node, we can assign `lst.rest = lst.rest.rest`. If we have lst assigned to a link list that looks like the following:
`1 -> 2 -> 3 -> 4 -> 5`

Setting `lst.rest = lst.rest.rest` will take the arrow that points form 1 to 2 and change it to point from 1 to 3. We can see this by evaluating `lst.rest.rest`. `lst.rest` is the arrow that comes from 1, and `lst.rest.rest` is the link with 3.

Once we've created the following list:
`1 -> 3 -> 4 -> 5`

we just need to call skip on the rest of the list. If we call skip on the list that starts at 3, we'll skip over the link with 4 and set the pointer from 3 to point to the link with 5. This is the behavior that we want! Therefore, our recursive call is `skip(lst.rest)`, since `lst.rest` is now the link that contains 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
    """

    if s is Link.empty:
        return False
    slow, fast = s, s.rest
    while fast is not Link.empty:
        if fast.rest is Link.empty:
            return False
        elif fast is slow or fast.rest is slow:
            return True
        slow, fast = slow.rest, fast.rest.rest
    return False
```

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
    """

    for b in t.branches:
        t.label += tree_sum(b)
    return t.label
```

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 ______________________:

            ____________________________________

    ____________________________________________
```

```
    def helper(t, seen_so_far):
        if t.label in seen_so_far:
          t.label = -1
        else:
            seen_so_far = seen_so_far + [t.label]
        for b in t.branches:
            helper(b, seen_so_far)
    helper(t, [])
```

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:

            ____________________________________

    ________________________________________________
```

```
def replace_leaves_sum(t):
    def helper(t, total):
        if t.is_leaf():
            t.label = total + t.label
        else:
            for b in t.branches:
                helper(b, total + t.label)
    helper(t, 0)
```

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 ____________________________________
```

```
if n == 0:
    return True
elif t.is_leaf():
    return n == 1 and t.label == elem
elif t.label == elem:
    return True in [contains_n(elem, n - 1, b) for b in
      t.branches]
else:
    return True in [contains_n(elem, n, b) for b in
      t.branches]
```

Base cases: The simplest case we have is when `n == 0`, or when we want at least 0 instances of `elem` in `t`. In this case, we always return `True`. The other simple case we consider is when the tree is only a leaf — there is nothing left to recurse on. In that case, we simply check to see that both `n == 1` and that `t.label == elem`, meaning that we have one element left to satisfy, and the leaf label satisfies the final element we are looking for. If we have more elements to search for (ie. n ¿ 1), then we will not satisfy that many elements at the leaf node; conversely, if we have fewer (ie. `n == 0`), then the case would already be covered by the first base case.

Recursive cases: If the current node isn't a leaf, then there's two different cases we should consider. Either the label of the current node is equal to `elem` or the label is not equal to `elem`. For the former, we would have to search for `n` more `elems` in each branch of `t` and return `True` if any of the branches contain `n` elems. For the latter, we would have `(n - 1)` elements remaining, so we would search for `(n - 1)` more `elems` in each branch of `t` and return `True` if any of the branches contain `(n - 1)` `elems`. Since there is not room to do a for loop, we can use a list comprehension to recursively call the function on each branch. Thus, our two list comprehension statements would be `[contains_n(elem, n, b)for b in t.branches]` and `[contains_n(elem, n - 1, b)for b in t.branches]`. To determine if any of the branches contain either `n` elems or `(n - 1)` elems, we can check if there's a `True` element in the respective lists.