CS 61A
Fall 2021

Linked Lists, Trees

Discussion 8: October 20, 2021

Representation: Repr, Str

Q1: WWPD: Repr-esentation

```
class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret
```

Given the above class definitions, what will the following lines output?

```
>>> A('one')
```

```
>>> print(A('one'))
```

```
>>> repr(A('two'))
```

```
>>> b = B()
```

```
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
```

Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

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

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

You can find an implementation of the `Link` class below:

```
class Link:
    """A linked list."""
    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:
            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) + '>'
```

Q2: The Hy-rules of Linked Lists

In this question, we are given the following Linked List:

```
ganondorf = Link('zelda', Link('link', Link('sheik', Link.empty)))
```

What expression would give us the value `'sheik'` from this Linked List?

What is the value of `ganondorf.rest.first`?

What would be the value of `str(ganondorf)`?

What expression would mutate this linked list to `<zelda ganondorf sheik>`?

Q3: Sum Nums

Write a function that takes in a linked list and returns the sum of all its elements. You may assume all elements in `s` are integers. Try to implement this recursively!

```
def sum_nums(s):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
    "*** YOUR CODE HERE ***"




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

Q4: Multiply Links

Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Implementation Note: you might not need all lines in this
    skeleton code
    ________________ = __________
    for _____________________________________:
        if _____________________________________:
            _________________________________
        ___________________
    ____________________________________________________
    ____________________________________________________
```

Q5: Flip Two

Write a recursive function `flip_two` that takes as input a linked list `s` and mutates `s` so that every pair is flipped.

```
def flip_two(s):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
    "*** YOUR CODE HERE ***"


    # For an extra challenge, try writing out an iterative approach
    as well below!
    "*** YOUR CODE HERE ***"









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

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 the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

With this implementation, we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists. That's why we sometimes call these objects "mutable trees."

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

Q6: Make Even

Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
    "*** YOUR CODE HERE ***"









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

Q7: Leaves

Write a function `leaves` that returns a list of all the label values of the leaf nodes of a `Tree`.

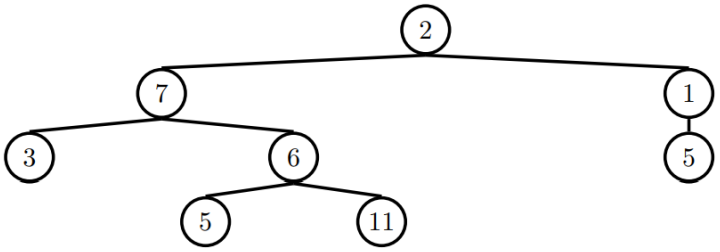
```
def leaves(t):
    """Returns a list of all the labels of the leaf nodes of the
    Tree t.

    >>> leaves(Tree(1))
    [1]
    >>> leaves(Tree(1, [Tree(2, [Tree(3)]), Tree(4)]))
    [3, 4]
    """
    "*** YOUR CODE HERE ***"


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

Q8: Find Paths

Hint: This question is similar to `find_path` on Discussion 05.

Define the procedure `find_paths` that, given a Tree `t` and an `entry`, returns a list of lists containing the nodes along each path from the root of `t` to `entry`. You may return the paths in any order.

For instance, for the following tree `tree_ex`, `find_paths` should behave as specified



in the function doctests.

```
def find_paths(t, entry):
    """
    >>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree
    (11)])]), Tree(1, [Tree(5)])])
    >>> find_paths(tree_ex, 5)
    [[2, 7, 6, 5], [2, 1, 5]]
    >>> find_paths(tree_ex, 12)
    []
    """

    paths = []
    if ____________________________:
        _________________________________
    for _________________________________:
        ________________________:
            ________________________________________
    ______________________
```