CS 61A Summer 2019 — Linked Lists, Interfaces, and Trees

Discussion 7: July 25, 2019

1 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!')
```

Implementation

```
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:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Questions

1.1 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
    """
```

1.2 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

2 Interfaces

In computer science, an **interface** is a shared set of attributes, along with a specification of the attributes' behavior. For example, an interface for vehicles might consist of the following methods:

- `def drive(self):` Drives the vehicle if it has stopped.
- `def stop(self):` Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a `Car` class and a `Train` can both implement the interface described above, but the `Car` probably has a different mechanism for `drive` than the `Train`.

The power of interfaces is that other programs don't have to know *how* each data type implements the interface – only that they *have* implemented the interface. The following `travel` function can work with both `Car`s and `Train`s:

```
def travel(vehicle):
    while not at_destination():
        vehicle.drive()
    vehicle.stop()
```

Magic Methods

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- `def __add__(self, other):` Allows objects to do `self + other`.
- `def __sub__(self, other):` Allows objects to do `self - other`.
- `def __mul__(self, other):` Allows objects to do `self * other`.

In addition, there is also an interface for sequences:

- `def __len__(self):` Allows objects to do `len(self)`.
- `def __getitem__(self, index):` Allows objects to do `self[i]`.

Questions

2.1 What would Python display?

```
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
```

```
>>> A("one")
```

```
>>> print(A("one"))
```

```
>>> repr(A("two"))
```

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

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

```
>>> c = A("c")
>>> b.add_a(c)
>>> str(b)
```

2.2 Write the function `is_palindrome` such that it works for any data type that implements the sequence interface.

Assume that the Link class has implemented the `\_\_len\_\_` method and a `\_\_getitem\_\_` method which takes in integers.

```
def is_palindrome(seq):
    """ Returns True if the sequence is a palindrome. A palindrome is a sequence
    that reads the same forwards as backwards
    >>> is_palindrome(Link("l", Link("i", Link("n", Link("k")))))
    False
    >>> is_palindrome(["l", "i", "n", "k"])
    False
    >>> is_palindrome("link")
    False
    >>> is_palindrome(Link.empty)
    True
    >>> is_palindrome([])
    True
    >>> is_palindrome("")
    True
    >>> is_palindrome(Link("a", Link("v", Link("a"))))
    True
    >>> is_palindrome(["a", "v", "a"])
    True
    >>> is_palindrome("ava")
    True
    """
```

3 Trees

Recall the tree abstract data type: a tree is defined as having a label and some branches. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```python
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
```

Notice that with this implementation we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists.

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

Questions

3.1 Assuming that every value in `t` is a number, let's define `average(t)`, which returns the average of all the values in `t`.

```
def average(t):
    """
    Returns the average value of all the nodes in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
```

3.2 Implement long_paths, which returns a list of all *paths* in a tree with length at least n. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
```