CS 61A Fall 2017

Linked Lists and Midterm Review

Discussion 6: October 11, 2017

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 agains 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 __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
    def __repr__(self):
        if self.rest is Link.empty:
            return 'Link({})'.format(self.first)
        else:
            return 'Link({}, {})'.format(self.first,
                                         repr(self.rest))
```

Questions

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

```
def flip_two(lnk):
    """
    >>> 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)))))
    """
```

1.2 Write a function `remove_duplicates` that takes as input a sorted linked list of integers, `lnk`, and mutates `lnk` so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> unique = remove_duplicates(lnk)
    >>> len(unique)
    2
    >>> len(lnk)
    2
    """
```

1.3 Define `reverse`, which takes in a linked list and reverses the order of the links. The function may *not* return a new list; it must mutate the original list. Return a pointer to the head of the reversed list.

```
def reverse(lnk):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> r = reverse(a)
    >>> r.first
    3
    >>> r.rest.first
    2
    """
```

1.4 Write `multiply_lnks`, which takes in a Python list of `Link` objects 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
    ()
    """
```

2 Midterm Review

2.1 Define a function `even_weighted` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """

    return [_________________________________________________]
```

2.2 The **quicksort** sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the **pivot** element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

First, implement the `quicksort_list` function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

```
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """

    if ____________________________________________________:

        ____________________________________________________

    pivot = lst[0]

    less = __________________________________________________

    greater = _______________________________________________

    return __________________________________________________
```

2.3 We can also use quicksort to sort linked lists! Implement the `quicksort_link` function, without constructing additional `Link` instances.

You can assume that the `extend_links` function is already defined. It takes two linked lists and mutates the first so that the ending node points to the second. `extend_link` returns the head of the first linked list.

```
>>> l1, l2 = Link(1, Link(2)), Link(3, Link(4))
>>> l3 = extend_links(l1, l2)
>>> l3
Link(1, Link(2, Link(3, Link(4))))
>>> l1 is l3
True

def quicksort_link(link):
    """
    >>> s = Link(3, Link(1, Link(4)))
    >>> quicksort_link(s)
    Link(1, Link(3, Link(4)))
    """
    if _____________________________________________________:

        return link

    pivot, _______ = ________________________________________

    less, greater = _________________________________________

    while link is not Link.empty:

        curr, rest = link, link.rest

        if _________________________________________________:

            _________________________________________________

        else:

            _________________________________________________

        link = ______________________________________________

    less = __________________________________________________

    greater = _______________________________________________

    _________________________________________________________

    return __________________________________________________
```

2.4 Implement the functions `max_product`, which takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(lst):
    """Return the maximum product that can be formed using lst
    without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
```

2.5 An **expression tree** is a tree that contains a function for each non-leaf node, which can be either '+' or '*'. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```
def eval_tree(tree):
    """Evaluates an expression tree with functions the root.
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """
```

2.6 Implement `widest_level`, which takes a `Tree` instance and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]

    while __________________________________________________:

        ____________________________________________________

        __________ = sum(_______________________________, [])

    return max(levels, key=_________________________________)
```

2.7 Complete `redundant_map`, which takes a tree `t` and a function `f`, and applies `f` to the node (2^d) times, where d is the depth of the node. The root has a depth of 0.

```
def redundant_map(t, f):
  """
  >>> double = lambda x: x*2
  >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
  >>> print_levels(redundant_map(tree, double))
  [2] # 1 * 2 ˆ (1) ; Apply double one time
  [4, 8] # 1 * 2 ˆ (2), 2 * 2 ˆ (2) ; Apply double two times
  [16] # 1 * 2 ˆ (2 ˆ 2) ; Apply double four times
  [256] # 1 * 2 ˆ (2 ˆ 3) ; Apply double eight times
  """
  t.label = _______________________________

  new_f = ______________________________________________

  t.branches = ___________________________________________

  return t
```