

# LINKED LISTS, MUTABLE TREES AND MIDTERM REVIEW

---

## COMPUTER SCIENCE MENTORS 61A

October 28 – November 1, 2024

### 1 Linked Lists

---

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) + '>'
```

**Linked lists** are a recursive data structure for representing sequences. They consist of a series of “links,” each of which has two attributes: `first` and `rest`. The `first` attribute contains the value of the link (which can hold any type of data, even another linked list!). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just a “None” type value.

For example, `Link(1, Link(2, Link(3)))` is a linked list representation of the sequence 1, 2, 3.

Like trees, linked lists naturally lend themselves to recursive problem solving. Consider the following example, in which we double every value in linked list. We double the value of 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
```

1. What will Python output? Draw box-and-pointer diagrams along the way.

```
>>> 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 `combine_two`, which takes in a linked list of integers `lnk` and a two-argument function `fn`. It returns a new linked list where every two elements of `lnk` have been combined using `fn`.

```

def combine_two(lnk, fn):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> combine_two(lnk1, add)
    Link(3, Link(7))
    >>> lnk2 = Link(2, Link(4, Link(6)))
    >>> combine_two(lnk2, mul)
    Link(8, Link(6))
    """
    if _____:

        return _____

    elif _____:

        return _____

    combined = _____

    return _____

```

3. Write a function `middle_node` that takes as input a linked list `lst`. `middle_node` should return the middle node of the linked list. If there are two middle nodes, return the second middle node.

```
def middle_node(lst):
    """
    >>> head = Link(1, Link(2, Link(3, Link(4, Link(5))))
    >>> middle_node(head)
    Link(3, Link(4, Link(5))) # The middle node of the list is node 3
    >>> head = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6)))))
    Link(4, Link(5, Link(6))) # Since the list has two middle nodes with
                             values 3 and 4, we return the second one
    """
    list_iter, middle = _____, _____

    length = _____

    while _____:

        length = _____

        list_iter = _____

    for _____:

        middle = _____

    if length % 2 == 1:

        middle = _____

    return middle
```

Challenge version (Optional):

```
def middle_node(lst):

    list_iter, middle = _____, _____

    while _____ and _____:

        list_iter = _____

        middle = _____

    return middle
```

4. Write a recursive function `insert_all` that takes as input two linked lists, `s` and `x`, and an index `index`. `insert_all` should return a new linked list with the contents of `x` inserted at index `index` of `s`.

```
def insert_all(s, x, index):
    """
    >>> insert = Link(3, Link(4))
    >>> original = Link(1, Link(2, Link(5)))
    >>> insert_all(original, insert, 2)
    Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> start = Link(1)
    >>> insert_all(original, start, 0)
    Link(1, Link(1, Link(2, Link(5))))
    """
    if _____ and _____:
        _____

    if _____ and _____:
        _____

    _____
```

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 self.branches == []

# Implementation omitted
```

Here are a few key differences between the `Tree` class and the `Tree` abstract data type, which we have previously encountered:

- Using the constructor: Capital `T` for the `Tree` class and lowercase `t` for tree ADT `t = Tree(1)` vs. `t = tree(1)`
- In the class, `label` and `branches` are instance variables and `is_leaf()` is an instance method. In the ADT, all of these were globally defined functions.

`t.label` vs. `label(t)`

`t.branches` vs. `branches(t)`

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

- A `Tree` object is mutable while the tree ADT is not mutable. This means we can change attributes of a `Tree` instance without making a new tree. In other words, we can solve tree class problems non-destructively and destructively, but can only solve tree ADT problems non-destructively.

`t.label = 2` is allowed but `label(t) = 2` would error.

Apart from these differences, we can take the same general approaches we used for the tree ADT and apply them to the `Tree` class!

1. 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 _____:
            _____
    _____
```

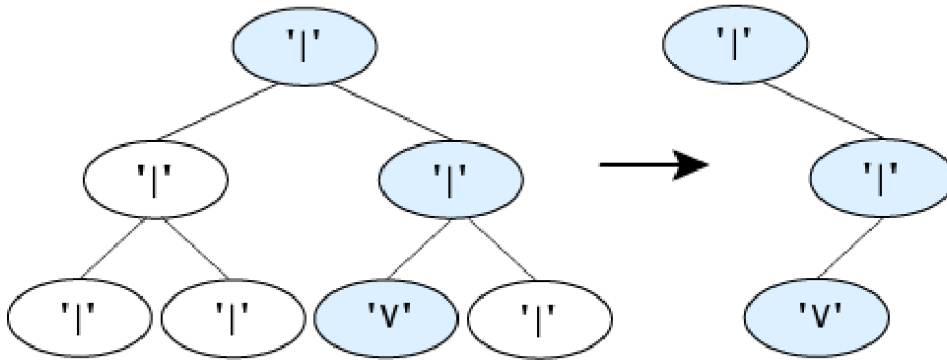
2. Given a tree `t`, mutate the tree so that each leaf's label becomes the sum of the labels of all nodes in the path from the leaf node to the root node.

```
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:  
            _____  
    _____
```

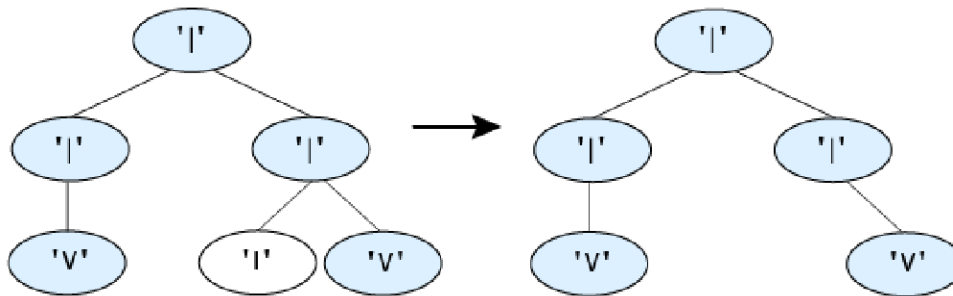


3. From Sp'22 MT2:

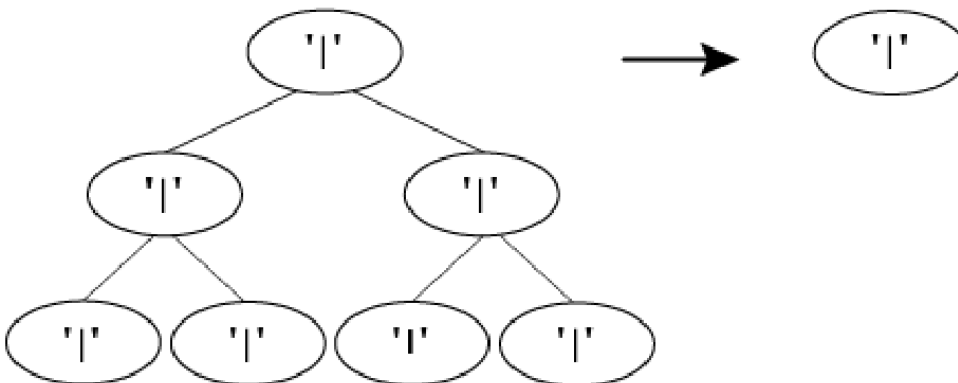
Implement *flower\_keeper*, a function that mutates a tree *t* so that the only paths that remain are ones which end in a leaf node with a Tulip flower ('V'). For example, consider this tree where only one path ends in a flower. After calling *flower\_keeper*, the tree has only three nodes left, the ones that lead to the flower:



The shaded nodes in the diagram indicate paths that end in flowers. For this tree where two paths end in flowers, the tree keeps both paths that lead to flowers.



For a tree where none of the nodes are flowers, the function removes every branch except the root node.



For a tree with only a single node that is a flower, the function does not remove anything.



```
def flower_keeper(t):
    """
    Mutates the tree T to keep only paths that end in flowers ('V').
    If a path consists entirely of stems ('|'), it must be pruned.
    If T has no paths that end in flowers, the root node is still kept.
    You can assume that a node with a flower will have no branches.
    >>> one_f = Tree('|', [Tree('|', [Tree('|'), Tree('|')]), Tree('|',
        [Tree('V'), Tree('|')])])
    >>> print(one_f)
    |
    |
    |
    |
    V
    |
    >>> flower_keeper(one_f)
    >>> one_f
    Tree('|', [Tree('|', [Tree('V')])])
    >>> print(one_f)
    |
    |
    V
    >>> no_f = Tree('|', [Tree('|', [Tree('|'), Tree('|')]), Tree('|',
        [Tree('|'), Tree('|')])])
    >>> flower_keeper(no_f)
    >>> no_f
    Tree('|')
    >>> just_f = Tree('V')
    >>> flower_keeper(just_f)
    >>> just_f
    Tree('V')
    >>> two_f = Tree('|', [Tree('|', [Tree('V')]), Tree('|', [Tree('|'),
        Tree('V')])])
    >>> flower_keeper(two_f)
    >>> two_f
    Tree('|', [Tree('|', [Tree('V')]), Tree('|', [Tree('V')])])
    """
    for b in _____:
        _____
        _____ = [_____ for b in _____ if _____]
```

### 3 Higher Order Functions

---

1. Write a function, `make_digit_remover`, which takes in a single digit `i`. It returns another function that takes in an integer and, scanning from right to left, removes all digits from the integer up to and including the first occurrence of `i`, starting from the ones place. If `i` does not occur in the integer, the original number is returned.

```
def make_digit_remover(i):  
    """  
    >>> remove_two = make_digit_remover(2)  
    >>> remove_two(232018)  
    23  
    >>> remove_two(23)  
    0  
    >>> remove_two(99)  
    99  
    """  
    def remove(_____) :  
  
        removed = _____  
  
        while _____ > 0:  
  
            _____  
  
            removed = removed // 10  
  
            if _____:  
  
                _____  
  
        return _____  
  
    return _____
```

1. Draw the environment diagram that results from running the following code.

```
bless, up = 3, 5
another = [1, 2, 3, 4]
one = another[1:]

another[bless] = up
another.append(one.remove(2))
another[another[0]] = one
one[another[0]] = another[1]
one = one + [another.pop(3)]
another[1] = one[1][1][0]
one.append([one.pop(1)])
```

2. Draw the box-and-pointer diagram.

```
>>> violet = [7, 77, 17]
>>> violet.append([violet.pop(1)])

>>> dash = violet * 2
>>> jack = dash[3:5]
>>> jackjack = jack.extend(jack)

>>> helen = list(violet)
>>> helen += [jackjack]
>>> helen[2].append(violet)
```

3. Write a function `duplicate_list`, which takes in a list of positive integers and returns a new list with each element  $x$  in the original list duplicated  $x$  times.

```
def duplicate_list(lst):
    """
    >>> duplicate_list([1, 2, 3])
    [1, 2, 2, 3, 3, 3]
    >>> duplicate_list([5])
    [5, 5, 5, 5, 5]
    """
    _____

    for _____:
        for _____:
            _____
    _____
```

4. Write a function that takes as input a number  $n$  and a list of numbers `lst` and returns `True` if we can find a subset of `lst` that sums to  $n$ .

```
def add_up(n, lst):
    """
    >>> add_up(10, [1, 2, 3, 4, 5])
    True
    >>> add_up(8, [2, 1, 5, 4, 3])
    True
    >>> add_up(-1, [1, 2, 3, 4, 5])
    False
    >>> add_up(100, [1, 2, 3, 4, 5])
    False
    """
    if _____:
        return True

    if lst == []:
        _____

    else:
        first, rest = _____, _____

        return _____
```

## 5 Iterators and Generators

---

1. Write a generator function `num_elems` that takes in a possibly nested list of numbers `lst` and yields the number of elements in each nested list before finally yielding the total number of elements (including the elements of nested lists) in `lst`. For a nested list, yield the size of the inner list before the outer, and if you have multiple nested lists, yield their sizes from left to right.

```
def num_elems(lst):  
    """  
    >>> list(num_elems([3, 3, 2, 1]))  
    [4]  
    >>> list(num_elems([1, 3, 5, [1, [3, 5, [5, 7]]]]))  
    [2, 4, 5, 8]  
    """  
  
    count = _____  
  
    for _____:  
        if _____:  
            for _____:  
                yield _____  
            _____  
        else:  
            _____  
  
    yield _____
```