Linked Lists

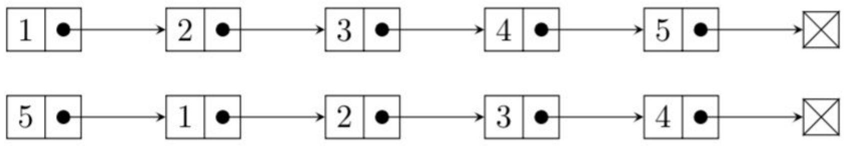
Remove All

Given a linked list, and a list of values, return a new linked list where all the links whose value exists in the list of values are removed.

```
def remove_all(lnk, vals):
      """
      >>> link = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
      >>> print(link)
      <1, 2, 3, 4, 5, 6>
      >>> print(remove_all(link, [2, 4]))
      <1, 3, 5, 6>
      >>> print(remove_all(link, []))
      <1, 2, 3, 4, 5, 6>
      """
      if lnk is Link.empty:
            return Link.empty
      elif lnk.first in vals:
            return remove_all(lnk.rest, vals)
      else:
            return Link(lnk.first, remove_all(lnk.rest, vals))
```

Reorder

Write a function `reorder` that takes in a linked list `lst` and mutates the linked list so that the last node is now the first node. Return the head of the linked list.



```
def reorder(lst):
      prev, last = Link.empty, lst
      while last.rest is not Link.empty:
            prev = last
            last = last.rest
      prev.rest = Link.empty
      last.rest = lst
      lst = last
      return lst
```

The giraffe of linked-lists

Given a linked-list, lst, of positive integers, mutate the linked-list such that each link is repeated the amount of times as the number in its first attribute.

```
def elongate(lst):
    >>> list = Link(2, Link(3, Link(1)))
    >>> print(list)
    <2 3 1>
    >>> elongate(list)
    >>> print(list)
    <2 2 3 3 3 1>
    if not lst is Link.empty:
          elongate(lst.rest)
          for _ in range(lst.first - 1):
                lst.rest = Link(lst.first, lst.rest)
```

Important: The function would not work correctly if instead of lst.rest = Link(lst.first, lst.rest), we had lst = Link(lst.first, lst). Why? Think of what the linked-list looks like in the global frame after we return from the function with the incorrect solution.

Trees

Remove Nodes

Implement `remove_nodes` which removes all nodes (excluding the root node) of a tree if the label of that root node returns True when the predicate `pred` is called on it. If a node is removed, all of its children are transfered to be its parents.

```
def remove_nodes(t, pred):
    """
    >>> t = Tree(1, [Tree(2), Tree(3), Tree(4, [Tree(6)]), Tree(5, [Tree(7)])])
    >>> remove_nodes(t, lambda x: x % 2 == 1)
    >>> t
    Tree(1, [Tree(2), Tree(4, [Tree(6)])])
    >>> t = Tree(6, [Tree(3, [Tree(5, [Tree(9), Tree(2, [Tree(7)])]), Tree(1)])])
    >>> remove_nodes(t, lambda x: x % 2 == 0)
    >>> t
    Tree(6, [Tree(3, [Tree(5, [Tree(9), Tree(7)]), Tree(1)])])
    """
    for b in list(t.branches):
        remove_nodes(b, pred)
        if pred(b.label):
            t.branches.extend(b.branches)
            t.branches.remove(b)
```

A Function for a Tree of Functions

Given a tree, t, that has one-parameter functions in its label attributes, a starting value n, and an ending value k, return a python list of linked-lists that contains all the paths from the root of the tree to it's leaves such that evaluating each level starting with the starting value n, results in k at the leaf.

```
def func_paths(t, n, k):
    >>> t = Tree(add_one , [Tree(square), Tree(cube)] )
    >>> func_paths(t, 3, 16)
    [Link(<function add_one>, Link(<function square>))]
    >>> func_paths(t, 0, 1)
    [Link(<function add_one>, Link(<function square>)), Link(<function add_one>,
    Link(<function cube>))]
    if t.is_leaf() :
          if t.label(n) == k:
                return [Link(t.label)]
          else:
                return []
    paths = []
    for b in t.branches:
          paths += [Link(t.label, path) for path in func_paths(b, t.label(n), k)]
    return paths
```

Lowest Hanging Fruit

Return the label at the lowest (greatest depth) leaf. If tie, return the leftmost one.
Calling lowest on the tree on the right would return 5.

```
def lowest(t):
      def lowest_pair(t):
            if t.is_leaf():
                  return [t.label, 0]
            max_pair = max([lowest_pair(b) for b in t.branches], key=lambda pair: pair[1])
            max_pair[1] += 1
            return max_pair
      return lowest_pair(t)[0]
```

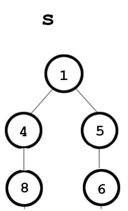
Fa12 Final (Adapted)

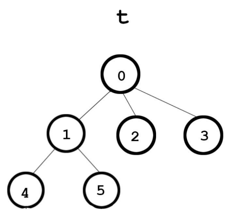
Various Longest Paths

A path through a tree is a sequence of connected nodes in which each node appears at most once. We will be solving three problems that deal with various longest paths. Below are the tree ADT functions that you'll need. Their implementations are abstracted away. We will also be using tree `s` and tree `t`. Make sure you understand their structures before you start on the questions.

```
def tree(label, branches=[]):
    ...

def label(tree):
    ...

def branches(tree):
    ...

def is_leaf(tree):
    ...

def print_tree(tree):
    ...
```

```
s = tree(1, [tree(4, [tree(8, [tree(9)])]),
              tree(5, [tree(6, [tree(7)])])])
t = tree(0, [s,
              tree(2),
              tree(3)])
>>> print_tree(s)
1
 4
  8
   9
 5
  6
   7
>>> print_tree(t)
0
 1
```



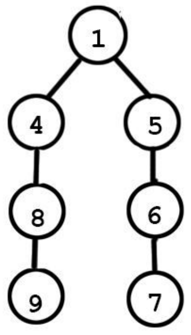


```
    4
      8
        9
    5
      6
        7
  2
  3
```

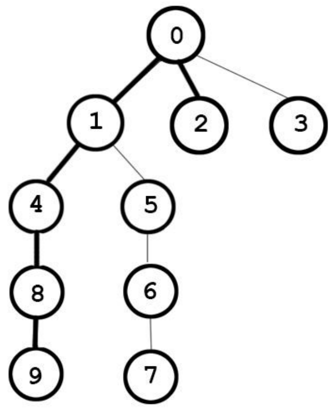
(a) Let's start with `longest_path_root_leaf`. Given a tree, return the number of nodes along the longest path from the root to a leaf. If there are multiple paths, pick the one that goes through the leftmost branch first.

```
def longest_path_root_leaf(tree):
      """ Given a tree, returns the number of nodes along the longest path from
      the root to a leaf. If there are multiple, pick the one that goes
      through the leftmost branch first.
      >>> longest_path_root_leaf(tree(1))
      1
      >>> longest_path_root_leaf(s)
      4
      >>> longest_path_root_leaf(t)
      5
      """

      if is_leaf(t): # We need base case because max([]) errors

          return 1

      return 1 + max([longest_path_root_leaf(b) for b in branches(t)])
```

(b) We will now implement `longest_path_root`. Given a tree, return the number of nodes along the longest path between two nodes, where the longest path must go through the root. You may assume `longest_path_root_leaf` works when implementing `longest_path_root`.



```
>>> longest_path_root(s)
7
```



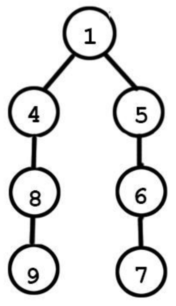
```
>>> longest_path_root(t)
6
```

The function `second_largest` takes in a list and returns the second largest element. If the list has less than two elements, `second_largest` returns 0. Assume `second_largest` is implemented correctly and you can call `second_largest` in `longest_path_root`.

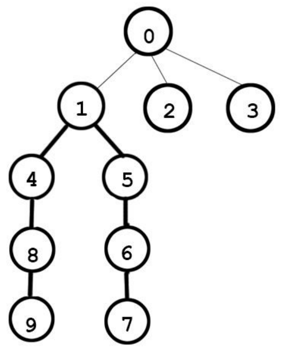
Hint: The longest path through the root can be composed of the two longest paths from the root to the leaf.

```
def second_largest(lst):
      """ Return the second largest item in a list, otherwise returns 0.
      Assume this function is already implemented and works correctly.
      >>> second_largest([3])
      0
      >>> second_largest([2, 2])
      2
      >>> second_largest([2,1,8,9,4])
      8
      """
      ...

def longest_path_root(tree):
      """ Given a tree, return the number of nodes along the longest path between
      any two nodes through the root.
      >>> longest_path_root(tree(1))
      1
      >>> longest_path_root(s)
      7
      >>> longest_path_root(t)
      6
      """
      if is_leaf(t): # We need base case because max([]) errors

          return 1

      longest_paths = [longest_path_root_leaf(b) for b in branches(t)]

      return 1 + max(longest_paths) + second_largest(longest_paths)
```

(c) Finally, let's implement `longest_path_tree`. Given a tree, return the total number of nodes along the longest path between any two nodes. The longest path does not necessarily have to go through the root. You may assume `longest_path_root` works when implementing `longest_path_tree`. If there are multiple longest paths, pick any of them.



```
>>> longest_path_root(s)
7
```



```
>>> longest_path_tree(t)
7
```

```
def longest_path_tree(tree):
    """ Given a tree, return the number of nodes along the longest path between
    any two nodes in the tree.
    >>> longest_path_tree(tree(1))
    1
    >>> longest_path_tree(s)
    7
    >>> longest_path_tree(t)
  7
  """
  if is_leaf(t):

      return 1

  longest_paths_without_root = [longest_path_tree(b) for b in branches(t)]

  longest_path_with_root = longest_path_root(t)

  return max(longest_path_with_root, max(longest_paths_without_root))
```

(d) Challenge Problem: Implement `list_nodes_root_leaf`, which is exactly the same as part a, except it returns a list with the labels of the nodes themselves instead of the total number of nodes in the path.

```
def list_nodes_root_leaf(tree):
    """ Given a tree, return a list with the labels of the nodes along the longest path
    from the root to a leaf. If there are multiple, pick the one that goes
    through the left most branch first.
    >>> list_nodes_root_leaf(tree(1))
    [1]
    >>> list_nodes_root_leaf(s)
    [1, 4, 8, 9]
    >>> list_nodes_root_leaf(t)
    [0, 1, 4, 8, 9]
    """
    if is_leaf(t): # We need base case because max([]) errors

        return [label(t)]

    return [label(t)] + max([list_nodes_root_leaf(b) for b in branches(t)], key=len)
```