

# LINKED LISTS AND MIDTERM REVIEW

---

COMPUTER SCIENCE MENTORS 61A

March 12 to March 14, 2018

---

## 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
```

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

```
if link is Link.empty:
    print('This linked list is empty!')
```

1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```
>>> 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
```

2. Write a function `skip`, which takes in a `Link` and returns a new `Link` with every other element skipped.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
    if _____:
        _____:
    elif _____:
        _____
    _____
```

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    None
    >>> a
    Link(1, Link(3))
    """
```

4. Write a function `reverse`, which takes in a `Link` and returns a new `Link` that has the order of the contents reversed.

*Hint:* You may want to use a helper function if you're solving this recursively.

```
def reverse(lst):  
    """  
    >>> a = Link(1, Link(2, Link(3)))  
    >>> b = reverse(a)  
    >>> b  
    Link(3, Link(2, Link(1)))  
    >>> a  
    Link(1, Link(2, Link(3)))  
    """
```

---

## Midterm Review

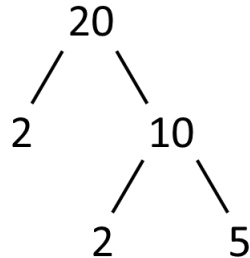
For each of the following problems, assume the `Tree` class is defined as follows:

```
class Tree:  
    def __init__(self, label, branches=[]):  
        self.label = label  
        self.branches = branches  
  
    def is_leaf(self):  
        return not self.branches
```

1. Write a function that returns true only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains(1, 2, t1)
    True
    >>> contains(2, 2, t1)
    False
    >>> contains(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains(1, 3, t2)
    True
    >>> contains(2, 2, t2) # Not on a path
    False
    """
    if n == 0:
        return True
    elif _____:
        return _____
    elif t.label == elem:
        return _____
    else:
        return _____
```

2. Define the function `factor_tree` which returns a *factor tree*. Recall that in a factor tree, multiplying the leaves together is the prime factorization of the root,  $n$ . See below for an example of a factor tree for  $n = 20$ .



```
def factor_tree(n):  
    for i in _____:  
        if _____:  
            return Tree(_____, _____)  
    _____
```

3. Draw the environment diagram that results from running the following code. If the code errors, draw the environment diagram up to the point that the error occurs.

```
earth = [0]
earth.append([earth])

def wind(fire, groove):
    fire[1][0][0] = groove
    def fire():
        nonlocal fire
        fire = lambda fantasy: earth.pop(1).extend(fantasy)
        return fire(groove)
    return fire()

sep = earth[1]
wind(earth, [earth[0]] + [earth.append(0)])
```