Review of Lab07 and Hw07

lab07p1: Complex


```
class Complex:
    """
    >>> a = Complex(1, 2)
    >>> a
    Complex(real=1, imaginary=2)
    >>> print(a)
    1 + 2i
    """

    def __repr__(self):
        return f'Complex(real={self.real}, imaginary={self.imaginary})'


    def __str__(self):
        return f'{self.real} + {self.imaginary}i'
```

lab07p1: Complex


```
class Complex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    def __add__(self, c):
        return Complex(self.real + c.real, self.imaginary + c.imaginary)

    def __mul__(self, c):
        return Complex(self.real * c.real - self.imaginary * c.imaginary,
                       self.real * c.imaginary + self.imaginary * c.real)
```

lab07p2: store_digits


```
def store_digits(n):
    if n == 0:
        return Link(0)
    else:
        link = Link.empty
        while n > 0:
            link = Link(n % 10, link)
            n //= 10
        return link
```

边界情况

lab07p2: store_digits


```
def store_digits(n):
    rest=Link.empty
    while n>=0:
        f=n%10
        rest = Link(f, rest)
        n //= 10
    return rest
```

```
>>> store_digits(0)
>>> store_digits(1)
>>> store_digits(999999999999)
```

小心边界条件

lab07p2: store_digits


```
def store_digits(n):
    if n<10:
        return Link(n)
    tmp,d=n,0
    while tmp>=10:
        tmp//=10
        d+=1
    return  Link(tmp,store_digits(n-tmp*(10**d)))
```

```
>>> store_digits(10)
>>> store_digits(100)
```

lab07p4: cumulative_mul


```
def cumulative_mul(t):
    product = t.label
    for b in t.branches:
        cumulative_mul(b)
        product *= b.label
    t.label = product
```

递归调用保证了在这个时间点，
b.label是所有子节点的权重乘积

lab07p4: cumulative_mul


```
def cumulative_mul(t):
    if t.is_leaf():
        return t.label
    #[cumulative_mul(br) for br in t.branches]
    pro = 1
    for br in t.branches:
        cumulative_mul(br)
        pro *= br.label
    t.label *= pro
```

if返回label，else却没有返回值（返回None）
无论题目要求是什么，这个代码都不可能正确

lab07p5: prune_small


```
def prune_small(t, n):
    if len(t.branches) > n:
        label = sorted([b.label for b in t.branches])[n - 1]
        t.branches = list(filter(lambda b: b.label <= label, t.branches))
    for b in t.branches:
        prune_small(b, n)
```

lab07p5: prune_small


```
def prune_small(t, n):
    if t.is_leaf():
        return
    t.branches.sort(key=lambda tree: tree.label)
    while len(t.branches) > n:
        t.branches.pop(-1)
    for branch in t.branches:
        prune_small(branch, n)
```

branch的顺序发生了改变
实际进行的并不是“剪枝”操作

lab07p5: prune_small


```
def prune_small(t, n):
    while len(t.branches) > n:
        i = 0
        max_label = t.branches[0].label
        max_index = 0
        while i < len(t.branches) - 1:
            i += 1
            if t.branches[i].label > max_label and i != max_index:
                max_label = t.branches[i].label
                max_index = i
        t.branches.pop(i)
    for branch in t.branches:
        prune_small(branch, n)
```

这个条件并没有实际意义，但由于这里用了max_index的值，所以IDE不会提示你定义了一个没有使用过的变量

算了半天max_index结果没用到，白算了！

lab07p5: prune_small


```
def prune_small(t, n):
    for subtree in t.branches:
        prune_small(subtree,n)
    num_of_branches=len(t.branches)
    if(num_of_branches>n):
        helper=[]
        for i in range(num_of_branches):
            helper=helper+[(t.branches[i].label,i)]
        sorted(helper,key=lambda x:x[0])
        for i in range(n,num_of_branches):
            t.branches.pop(helper[i][1])
```

没用使用sorted的结果

“刻舟求剑”

lab07p5: prune_small


```
def prune_small(t, n):
    def core(t,count):
        if(t.is_leaf()):
            pass
        else:
            for unit in t.branches:
                count+=1
                if(count>n):
                    for a in range(0,len(t.branches)-n):
                        m=max([unit.label for unit in t.branches])
                        for i in range(0,len(t.branches)):
                            if t.branches[i].label==m:
                                t.branches.pop(i)
                                break
                core(unit,0)
    core(t,0)
    return
```

边遍历，边修改

hw07p1: Polynomial


```
class Polynomial:
    def __init__(self, lst):
        while len(lst) > 1 and lst[-1] == 0:
            lst = lst[:-1]
        self.args = lst

    def __repr__(self):
        return f'Polynomial({repr(self.args)})'

    def __str__(self):
        x_args = []
        for i in range(len(self.args)):
            x_args.append(f'{self.args[i]}' + ('' if i == 0 else f'*x^{i}'))
        return ' + '.join(x_args)
```

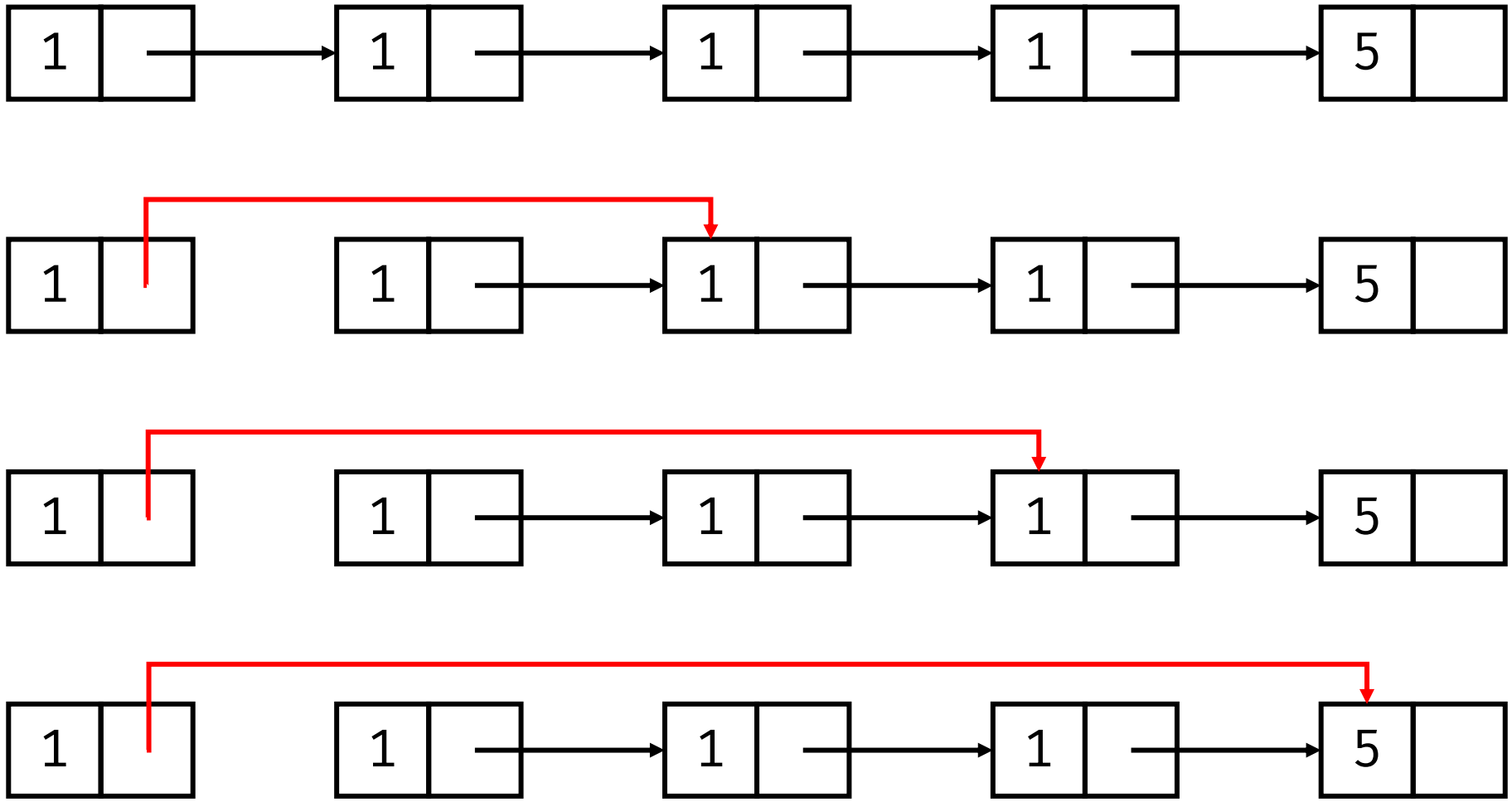
hw07p1: Polynomial


```
class Polynomial:
    def __add__(self, p):
        new_len = max(len(self.args), len(p.args))
        new_args = [0] * new_len
        for i in range(len(self.args)):
            new_args[i] += self.args[i]
        for i in range(len(p.args)):
            new_args[i] += p.args[i]
        return Polynomial(new_args)
```

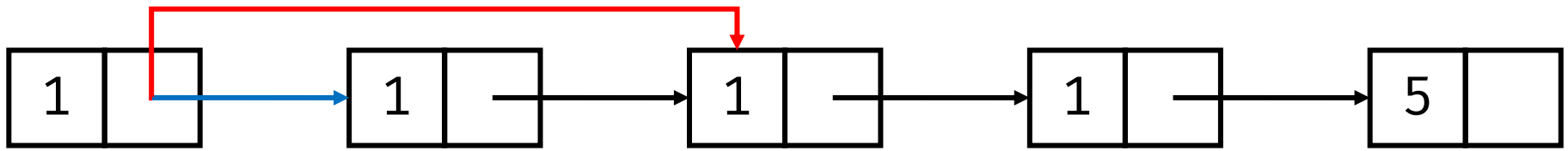
__init__会处理好各种输入

```
    def __mul__(self, p):
        new_len = len(self.args) + len(p.args)
        new_args = [0] * new_len
        for i in range(len(self.args)):
            for j in range(len(p.args)):
                new_args[i + j] += self.args[i] * p.args[j]
        return Polynomial(new_args)
```

hw07p2.1: remove_duplicates

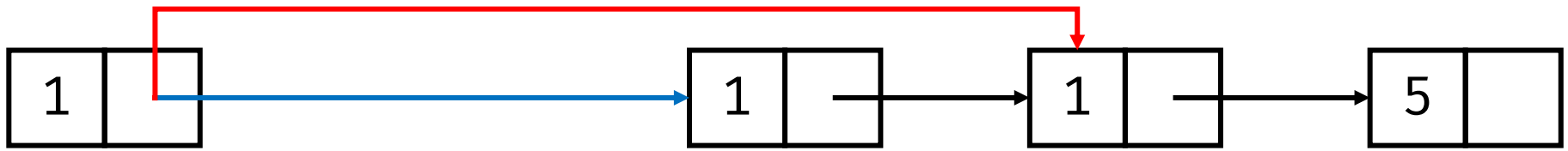


hw07p2.1: remove_duplicates




```
def remove_duplicates(lnk):
    while lnk != Link.empty:
        while lnk.rest != Link.empty and lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        lnk = lnk.rest
```

hw07p2.1: remove_duplicates

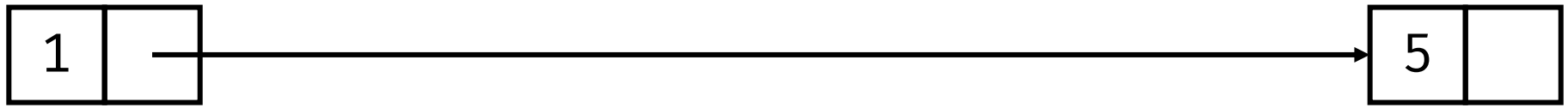



```
def remove_duplicates(lnk):
    while lnk != Link.empty:
        while lnk.rest != Link.empty and lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        lnk = lnk.rest
```

hw07p2.1: remove_duplicates

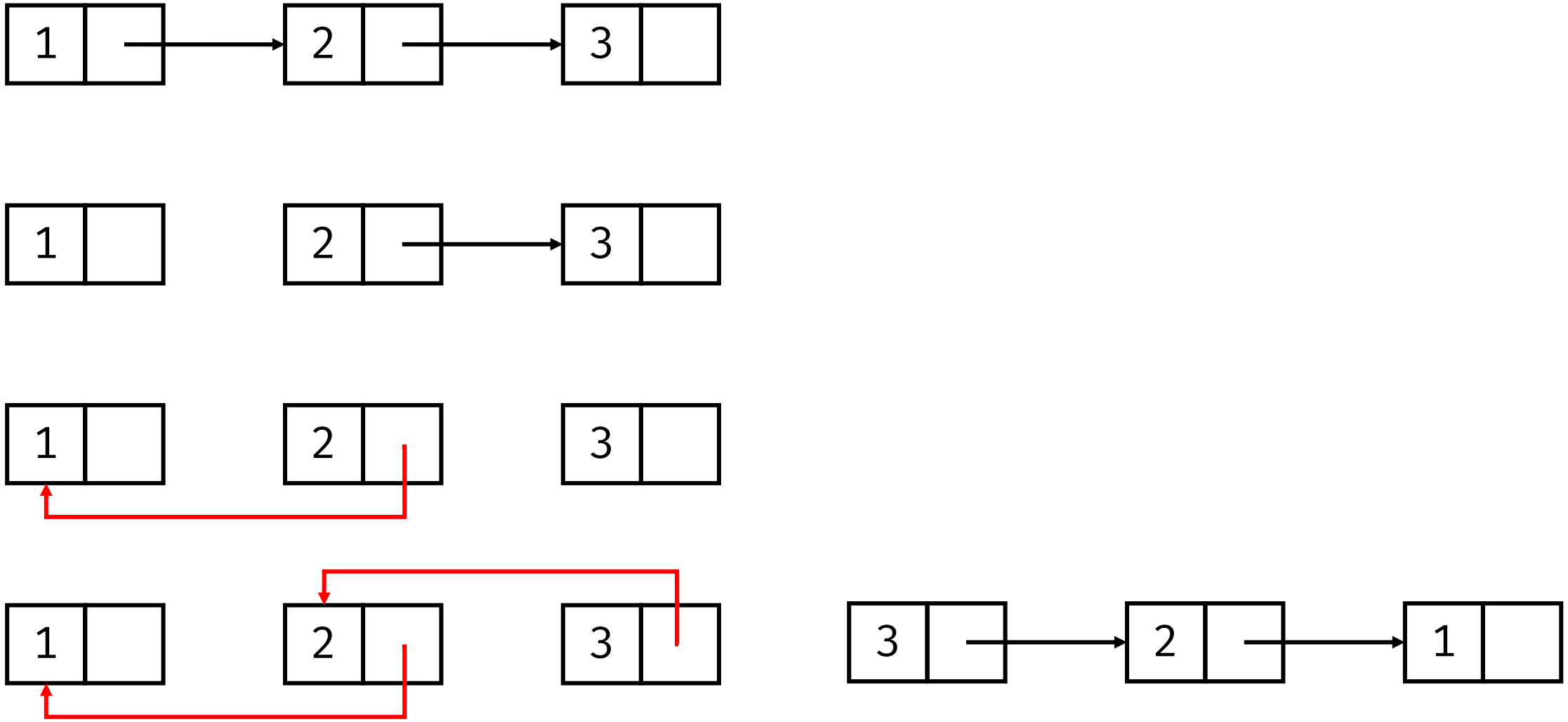

```
def remove_duplicates(lnk):
    while lnk != Link.empty:
        while lnk.rest != Link.empty and lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        lnk = lnk.rest
```

hw07p2.1: remove_duplicates



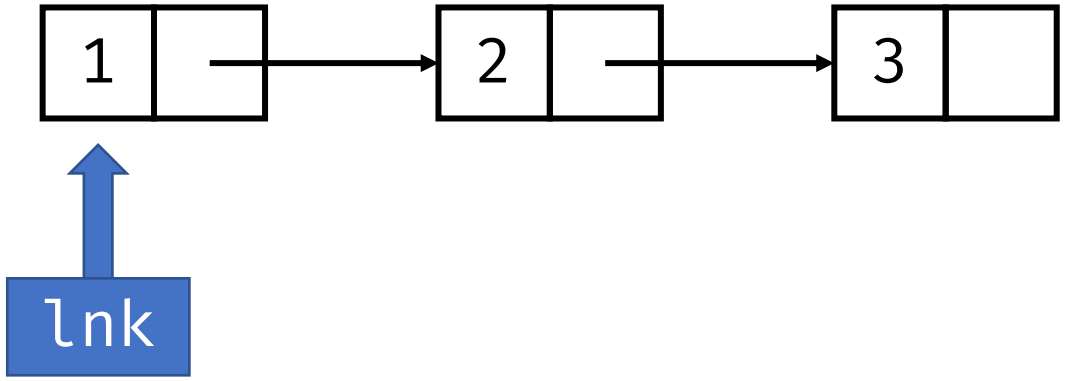

```
def remove_duplicates(lnk):
    while lnk != Link.empty:
        while lnk.rest != Link.empty and lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        lnk = lnk.rest
```

hw07p2.3: reverse



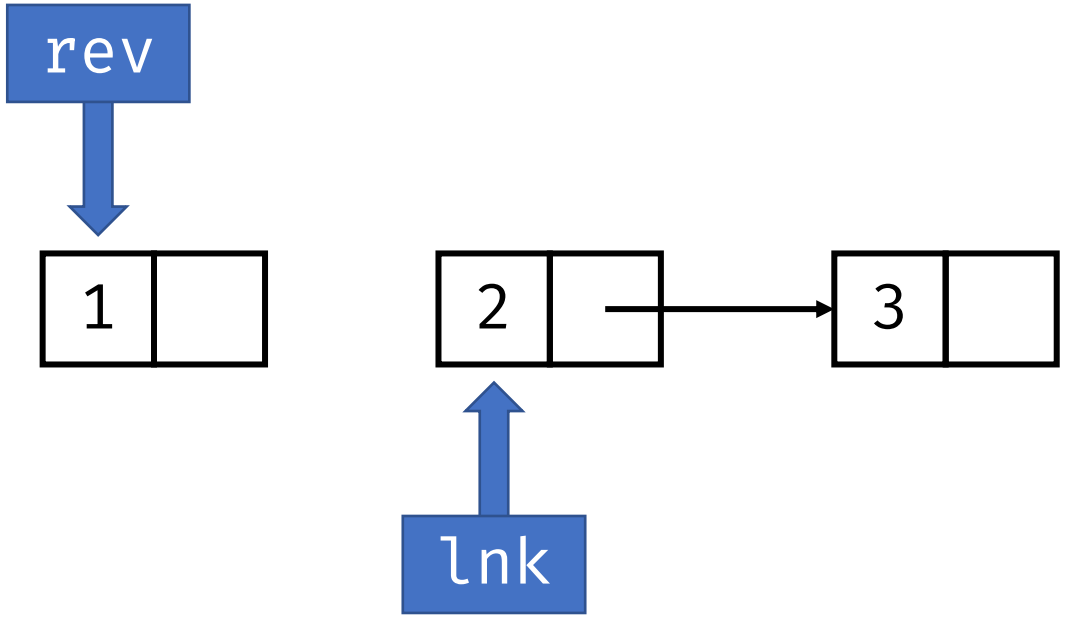
hw07p2.3: reverse

```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        rest = lnk.rest
        lnk.rest = rev
        rev = lnk
        lnk = rest
    return rev
```



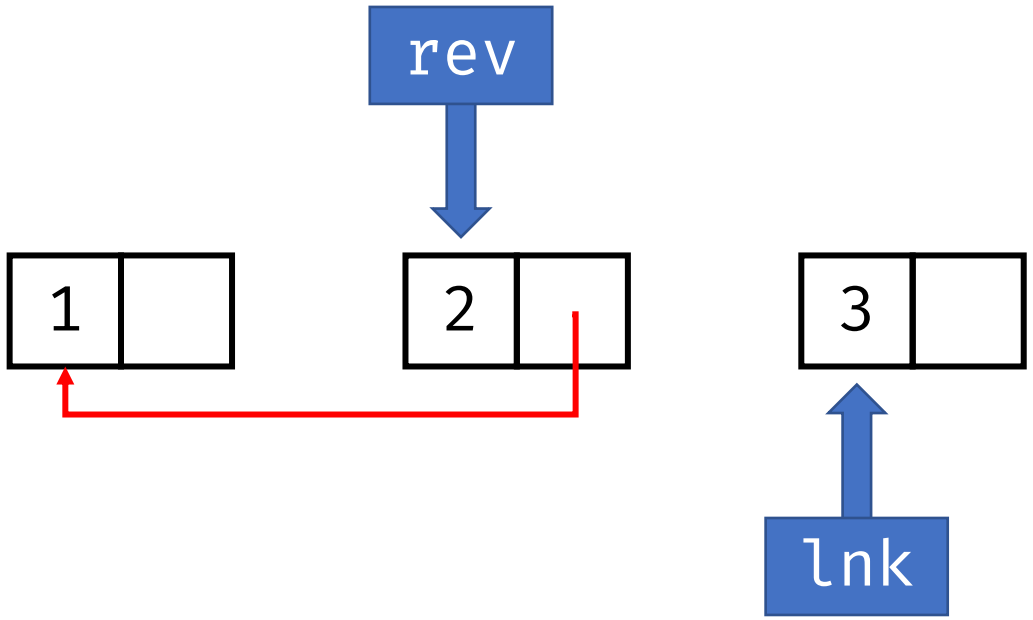
hw07p2.3: reverse

```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        rest = lnk.rest
        lnk.rest = rev
        rev = lnk
        lnk = rest
    return rev
```



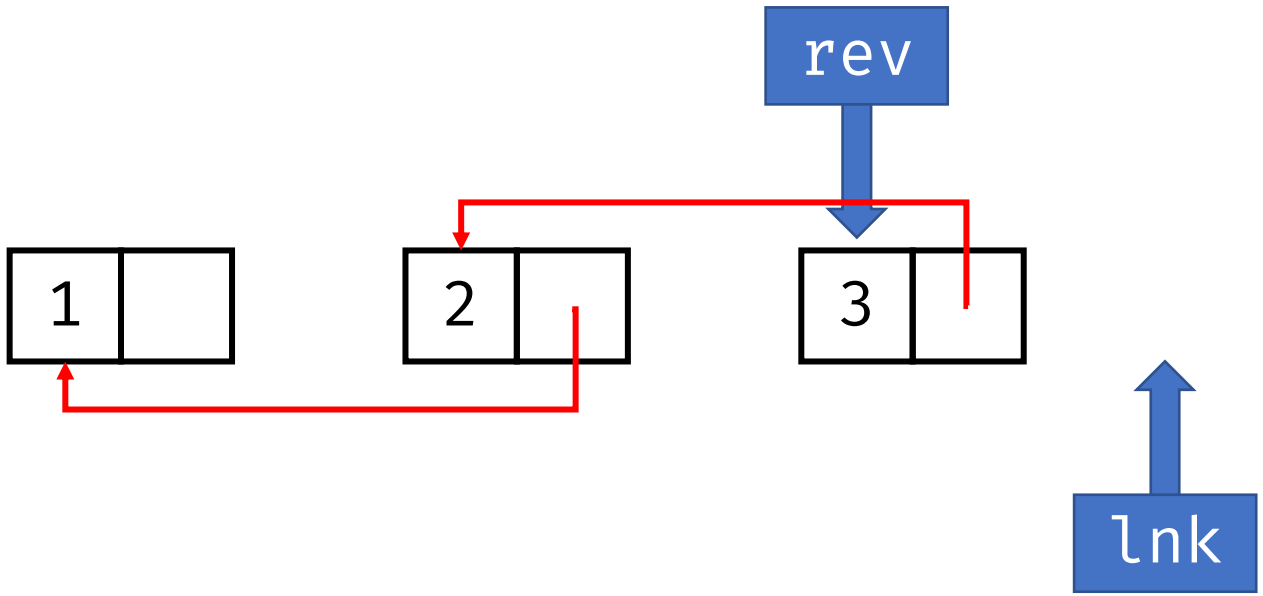
hw07p2.3: reverse

```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        rest = lnk.rest
        lnk.rest = rev
        rev = lnk
        lnk = rest
    return rev
```



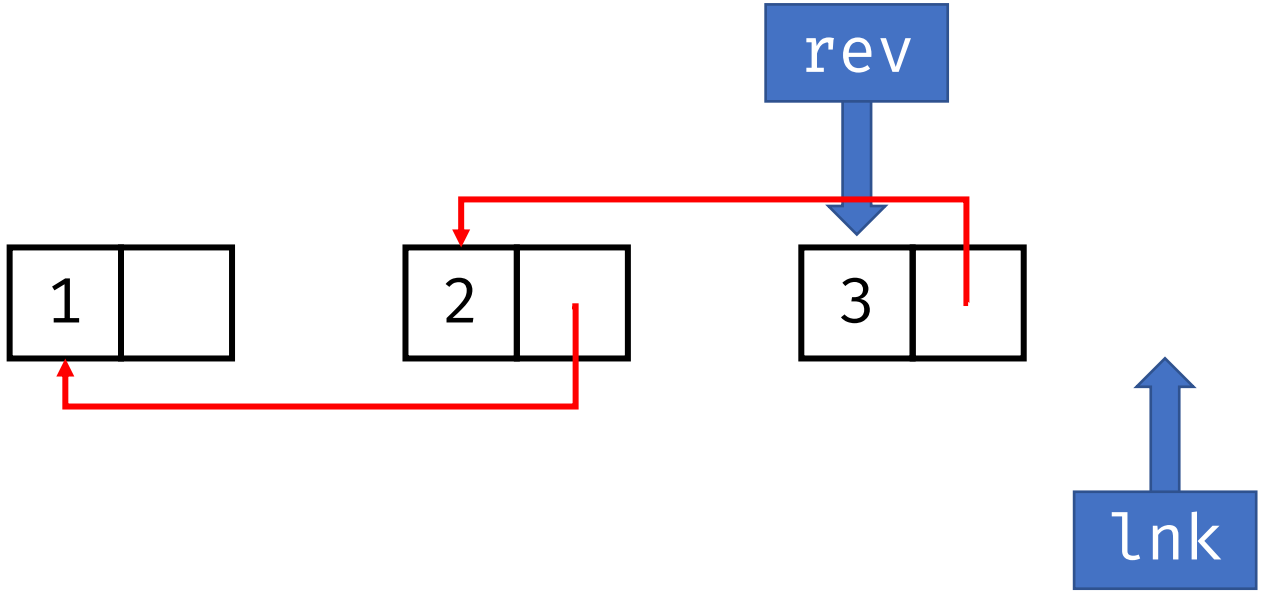
hw07p2.3: reverse

```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        rest = lnk.rest
        lnk.rest = rev
        rev = lnk
        lnk = rest
    return rev
```



hw07p2.3: reverse

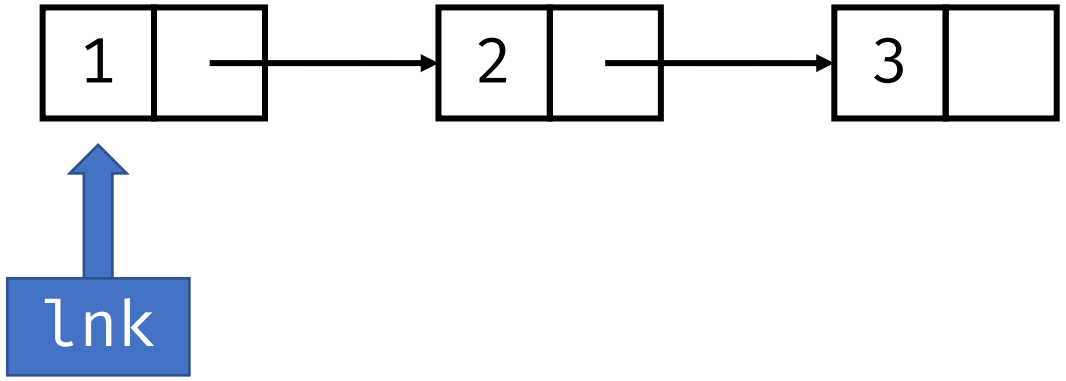
```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        rest = lnk.rest
        lnk.rest = rev
        rev = lnk
        lnk = rest
    return rev
```




```
def reverse(lnk):
    rev = Link.empty
    while lnk != Link.empty:
        lnk, lnk.rest, rev = lnk.rest, rev, lnk
    return rev
```

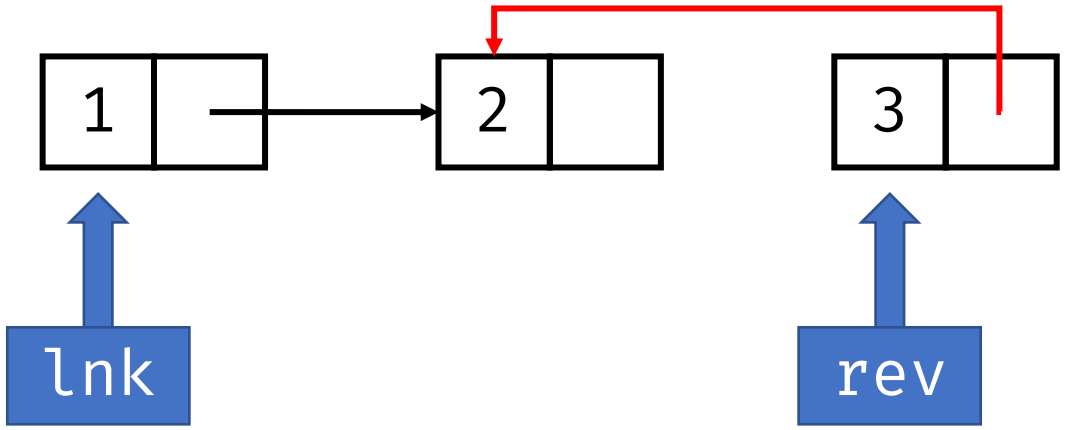
hw07p2.3: reverse

```
def reverse(lnk):
    if lnk == Link.empty:
        return Link.empty
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = empty
    return rev
```



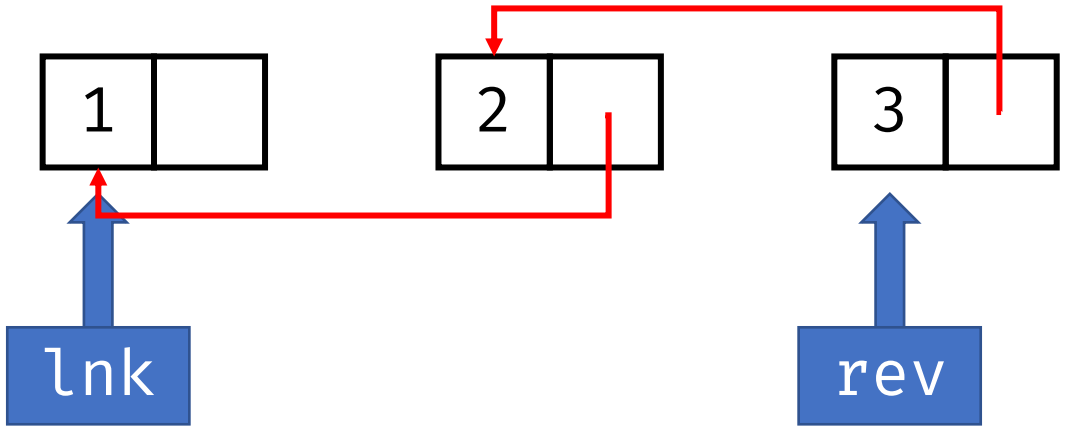
hw07p2.3: reverse


```
def reverse(lnk):
    if lnk == Link.empty:
        return Link.empty
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = empty
    return rev
```



hw07p2.3: reverse

```
def reverse(lnk):
    if lnk == Link.empty:
        return Link.empty
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = empty
    return rev
```



lnk.rest is ()

hw07p2.3: reverse

```
def reverse(lnk):
    if lnk == empty or lnk.rest == empty:
        return lnk
    rev = reverse(lnk.rest)
    lnk.rest.rest = lnk
    lnk.rest = empty
    return rev
```

hw07p3.1: generate_paths


```
def generate_paths(t, value):
    if t.label == value:
        yield [value]
    for b in t.branches:
        for p in generate_paths(b, value):
            yield [t.label] + p
```

hw07p3.2: funcs


```
def funcs(link):
    while link != Link.empty:
        yield (lambda i: lambda t: t.branches[i])(link.first)
        link = link.rest
    yield lambda t: t.label
```

hw07p3.3: count_coins_tree


```
def count_coins_tree(left, denos):
    if left == 0:
        return Tree('1')
    elif left < 0 or not denos:
        return None
    else:
        branches = [count_coins_tree(left, denos[1:]), \
                    count_coins_tree(left - denos[0], denos)]
        branches = list(filter(lambda x: x is not None, branches))
        if len(branches) == 0:
            return None
        return Tree(f'{left}, {denos}', branches)
```

hw07p3.4: balance_tree


```
def balance_tree(t):
    for b in t.branches:
        balance_tree(b)
    max_total = max([b.total for b in t.branches], default=0)
    for b in t.branches:
        b.label += max_total - b.total
    t.total = t.label + max_total * len(t.branches)
```

递归调用保证了在这个时间点
所有子树的权重都平衡，并且
b.total是子树的总权重

hw07p4: has_cycle

```
def has_cycle(lnk):
    p1, p2 = lnk, lnk
    while p1 != Link.empty and p2 != Link.empty:
        p1 = p1.rest
        if p2.rest == Link.empty:
            return False
        p2 = p2.rest.rest
        if p1 == p2:
            return True
    return False
```

p1每一次走1步，p2每一次走2步
第n次循环中，他们的距离是n

如果存在长度为c的环
当n%c==0的时候他们处于同一个位置

hw07p6: install_camera (top-down)


```
def install_camera(t):
    def helper(t, install, covered):
        if t.is_leaf():
            return 0 if covered else 1
        elif install:
            return sum([min(helper(b, True, True),
                            helper(b, False, True))
                        for b in t.branches]) + 1
        elif covered:
            return sum([min(helper(b, True, False),
                            helper(b, False, False))
                        for b in t.branches])
        else:
            return sum([helper(b, True, False) for b in t.branches])
    return min(helper(t, True, False), helper(t, False, False))
```

Install表示我愿不愿意装
Covered表示我是否被父节点的摄像头覆盖

叶子节点没被覆盖就得装一个摄像头

我装摄像头（+1），子节点可以爱装不装

我不装，但我被覆盖了，子节点爱装不装

我不装，且我没被覆盖，子节点必须全装

记忆化搜索（非本课程教学内容）

hw07p6: install_camera (bottom-up)


```
def install_camera(t):
    answer = 0
    def helper(t):
        nonlocal answer
        for b in t.branches:
            helper(b)
        if any(b.label == 0 for b in t.branches):
            answer += 1
            t.label = 1
        elif any(b.label == 1 for b in t.branches):
            t.label = 2
    helper(t)
    return answer + int(t.label == 0)
```

正确性证明:
对深度用数学归纳法
每次取深度最深的叶子
(非本课程教学内容)

如果有子节点没被覆盖，则我必须要装

如果有子节点装了，则我已被覆盖，就不装

如果根节点的所有子节点都没装，则还要+1