CS 61A Spring 2025

Linked Lists

Discussion 8: March 19, 2025

Switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Attendance

Fill out this discussion attendance form with the unique number you receive from your TA. As soon as you get your number, fill out the form, selecting *arrival* (not *departure* – that's later).

Getting Started

If there are fewer than 3 people in your group, feel free to merge your group with another group in the room.

Everybody say your name and your birthday and then tell the group about your favorite birthday party you've attended (either for your birthday or someone else's).

Pro tip: Groups tend not to ask for help unless they've been stuck for a looooooong time. Try asking for help sooner. We're pretty helpful! You might learn something.

Linked Lists

A linked list is a `Link` object or `Link.empty`.

You can mutate a `Link` object `s` in two ways: - Change the first element with `s.first = ...` - Change the rest of the elements with `s.rest = ...`

You can make a new `Link` object by calling `Link`: - `Link(4)` makes a linked list of length 1 containing 4. - `Link(4, s)` makes a linked list that starts with 4 followed by the elements of linked list `s`.

```
class Link:
    """A linked list is either a Link object or Link.empty

    >>> s = Link(3, Link(4, Link(5)))
    >>> s.rest
    Link(4, Link(5))
    >>> s.rest.rest.rest is Link.empty
    True
    >>> s.rest.first * 2
    8
    >>> print(s)
    <3 4 5>
    """
    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:
            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) + '>'
```

Drawing time: Pick a way for your group to draw diagrams. Paper, a whiteboard, or a tablet, are all fine. If you don't have anything like that, ask another group in the room if they have extra paper.

Q1: Strange Loop

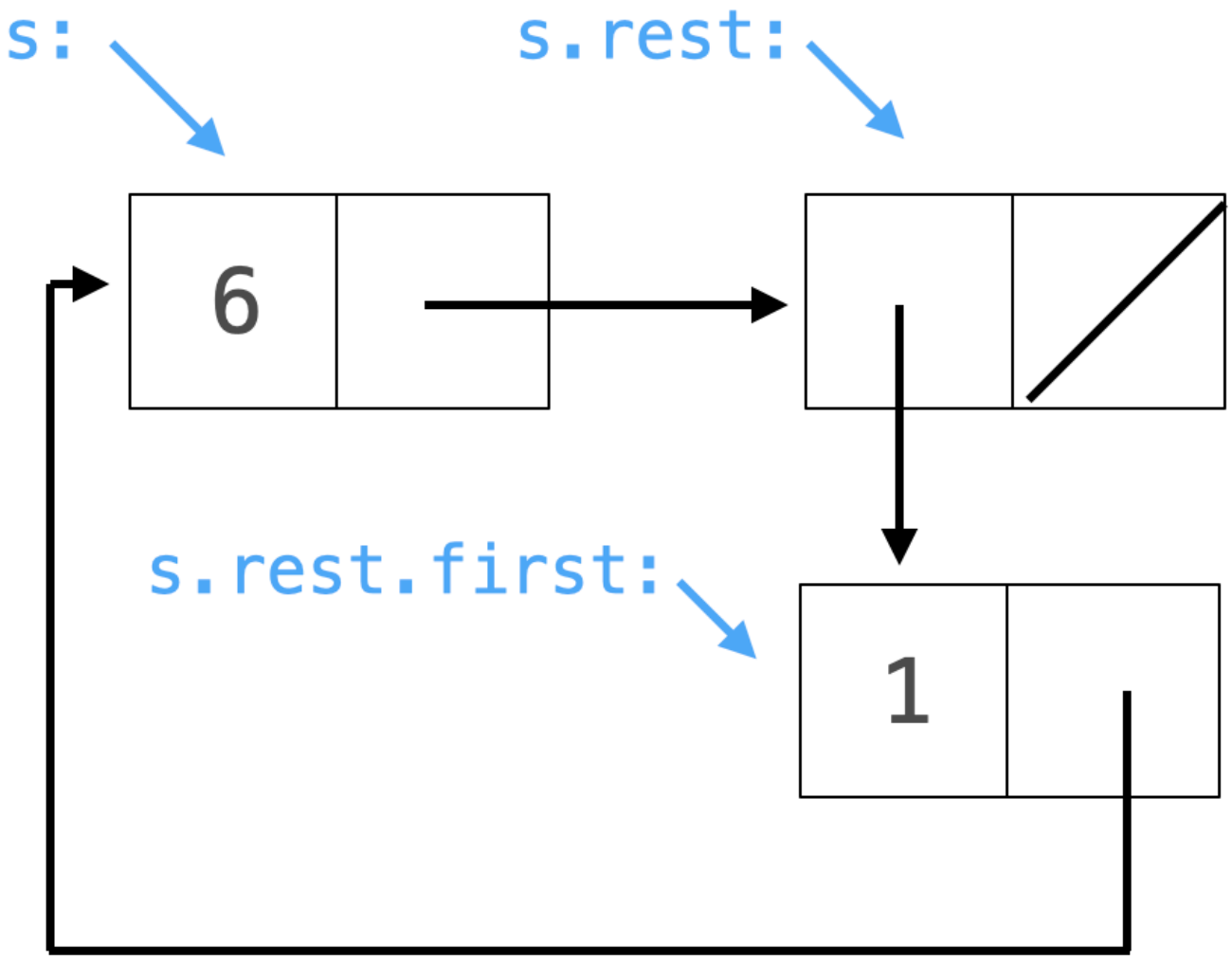
In lab, there was a `Link` object with a cycle that represented an infinite repeating list of 1's.

```
>>> ones = Link(1)
>>> ones.rest = ones
>>> [ones.first, ones.rest.first, ones.rest.rest.first, ones.rest.rest.rest.first]
[1, 1, 1, 1]
>>> ones.rest is ones
True
```

Implement `strange_loop`, which takes no arguments and returns a `Link` object `s` for which `s.rest.first.rest` is `s`.

Draw a picture of the linked list you want to create, then write code to create it.

For `s.rest.first.rest` to exist at all, the second element of `s`, called `s.rest.first`, must itself be a linked list.



Making a cycle requires two steps: making a linked list without a cycle, then modifying it. First create, for example, `s = Link(6, Link(Link(1)))`, then change `s.rest.first.rest` to create the cycle.

```
def strange_loop():
    """Return a Link s for which s.rest.first.rest is s.

    >>> s = strange_loop()
    >>> s.rest.first.rest is s
    True
    """
    s = Link(1, Link(Link(2)))
    s.rest.first.rest = s
    return s
```

Q2: Sum Two Ways

Implement both `sum_rec` and `sum_iter`. Each one takes a linked list of numbers `s` and a non-negative integer `k` and returns the sum of the first `k` elements of `s`. If there are fewer than `k` elements in `s`, all of them are summed. If `k` is 0 or `s` is empty, the sum is 0.

Use recursion to implement `sum_rec`. Don't use recursion to implement `sum_iter`; use a `while` loop instead.

```
def sum_rec(s, k):
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_rec(a, 2)
    7
    >>> sum_rec(a, 5)
    15
    >>> sum_rec(Link.empty, 1)
    0
    """
    # Use a recursive call to sum_rec; don't call sum_iter
    if k == 0 or s is Link.empty:
        return 0
    return s.first + sum_rec(s.rest, k - 1)

def sum_iter(s, k):
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_iter(a, 2)
    7
    >>> sum_iter(a, 5)
    15
    >>> sum_iter(Link.empty, 1)
    0
    """
    # Don't call sum_rec or sum_iter
    total = 0
    while k > 0 and s is not Link.empty:
        total, s, k = total + s.first, s.rest, k - 1
    return total
```

Add `s.first` to the sum of the first `k-1` elements in `s.rest`. Your base case condition should include `s is Link.empty` so that you're checking whether `s` is empty before ever evaluating `s.first` or `s.rest`.

Introduce a new name, such as `total`, then repeatedly (in a `while` loop) add `s.first` to `total`, set `s = s.rest` to advance through the linked list, and reduce `k` by one.

Discussion time: When adding up numbers, the intermediate sums depend on the order. `(1 + 3) + 5` and `1 + (3 + 5)` both equal 9, but the first one makes 4 along the way while the second makes 8 along the way. For the

same linked list `s` and length `k`, will `sum_rec` and `sum_iter` both make the same intermediate sums along the way?

For a summation, the order of additions doesn't affect the result, but for other operations this ordering matters. For example, `(2 ** 3) ** 5` is much smaller than `2 ** (3 ** 5)`.

Q3: Overlap

Implement `overlap`, which takes two linked lists of numbers called `s` and `t` that are sorted in increasing order and have no repeated elements within each list. It returns the count of how many numbers appear in both lists.

This can be done in *linear* time in the combined length of `s` and `t` by always advancing forward in the linked list whose first element is smallest until both first elements are equal (add one to the count and advance both) or one list is empty (time to return). Here's a lecture video clip about this (but the video uses Python lists instead of linked lists).

Take a vote to decide whether to use recursion or iteration. Either way works (and the solutions are about the same complexity/difficulty).

```
def overlap(s, t):
    """For increasing s and t, count the numbers that appear in both.

    >>> a = Link(3, Link(4, Link(6, Link(7, Link(9, Link(10))))))
    >>> b = Link(1, Link(3, Link(5, Link(7, Link(8)))))
    >>> overlap(a, b)  # 3 and 7
    2
    >>> overlap(a.rest, b)  # just 7
    1
    >>> overlap(Link(0, a), Link(0, b))
    3
    """
    if s is Link.empty or t is Link.empty:
        return 0
    if s.first == t.first:
        return 1 + overlap(s.rest, t.rest)
    elif s.first < t.first:
        return overlap(s.rest, t)
    elif s.first > t.first:
        return overlap(s, t.rest)


def overlap_iterative(s, t):
    """For increasing s and t, count the numbers that appear in both.

    >>> a = Link(3, Link(4, Link(6, Link(7, Link(9, Link(10))))))
    >>> b = Link(1, Link(3, Link(5, Link(7, Link(8)))))
    >>> overlap(a, b)  # 3 and 7
    2
    >>> overlap(a.rest, b)  # just 7
    1
    >>> overlap(Link(0, a), Link(0, b))
    3
    """
    res = 0
    while s is not Link.empty and t is not Link.empty:
        if s.first == t.first:
            res += 1
            s = s.rest
            t = t.rest
        elif s.first < t.first:
            s = s.rest
        else:
            t = t.rest
    return res
```

```
if s is Link.empty or t is Link.empty:
    return 0
if s.first == t.first:
    return ________________
elif s.first < t.first:
    return ________________
elif s.first > t.first:
    return ________________
```

```
k = 0
while s is not Link.empty and t is not Link.empty:
    if s.first == t.first:
        ________________
    elif s.first < t.first:
        ________________
    elif s.first > t.first:
        ________________
return k
```

Document the Occasion

Let your TA know you're done so that you can each get a **departure** number, and fill out the attendance form again (this time selecting *departure* instead of *arrival*). If your TA isn't in the room, go find them next door.

Extra Challenge

This last question is similar in complexity to an A+ question on an exam. Feel free to skip it, but it's a fun one, so try it if you have time.

Q4: Decimal Expansion

Definition. The *decimal expansion* of a fraction `n/d` with `n < d` is an infinite sequence of digits starting with the 0 before the decimal point and followed by digits that represent the tenths, hundredths, and thousands place (and so on) of the number `n/d`. E.g., the decimal expansion of 2/3 is a zero followed by an infinite sequence of 6's: 0.6666666….

Implement `divide`, which takes positive integers `n` and `d` with `n < d`. It returns a linked list with a cycle containing the digits of the infinite decimal expansion of `n/d`. The provided `display` function prints the first `k` digits after the decimal point.

For example, 1/22 would be represented as `x` below:

```
>>> 1/22
0.045454545454545456
>>> x = Link(0, Link(0, Link(4, Link(5))))
>>> x.rest.rest.rest.rest = x.rest.rest
>>> display(x, 20)
0.04545454545454545454...
```

```
def display(s, k=10):
    """Print the first k digits of infinite linked list s as a decimal.

    >>> s = Link(0, Link(8, Link(3)))
    >>> s.rest.rest.rest = s.rest.rest
    >>> display(s)
    0.8333333333...
    """
    assert s.first == 0, f'{s.first} is not 0'
    digits = f'{s.first}.'
    s = s.rest
    for _ in range(k):
        assert s.first >= 0 and s.first < 10, f'{s.first} is not a digit'
        digits += str(s.first)
        s = s.rest
    print(digits + '...')
```

```
def divide(n, d):
    """Return a linked list with a cycle containing the digits of n/d.

    >>> display(divide(5, 6))
    0.8333333333...
    >>> display(divide(2, 7))
    0.2857142857...
    >>> display(divide(1, 2500))
    0.0004000000...
    >>> display(divide(3, 11))
    0.2727272727...
    >>> display(divide(3, 99))
    0.0303030303...
    >>> display(divide(2, 31), 50)
    0.06451612903225806451612903225806451612903225806451...
    """
    assert n > 0 and n < d
    result = Link(0)  # The zero before the decimal point
    cache = {}
    tail = result
    while n not in cache:
        q, r = 10 * n // d, 10 * n % d
        tail.rest = Link(q)
        tail = tail.rest
        cache[n] = tail
        n = r
    tail.rest = cache[n]
    return result
```

Place the division pattern from the example above in a `while` statement:

```
>>> q, r = 10 * n // d, 10 * n % d
>>> tail.rest = Link(q)
>>> tail = tail.rest
>>> n = r
```

While constructing the decimal expansion, store the `tail` for each `n` in a dictionary keyed by `n`. When some `n` appears a second time, instead of constructing a new `Link`, set its original link as the rest of the previous link. That will form a cycle of the appropriate length.