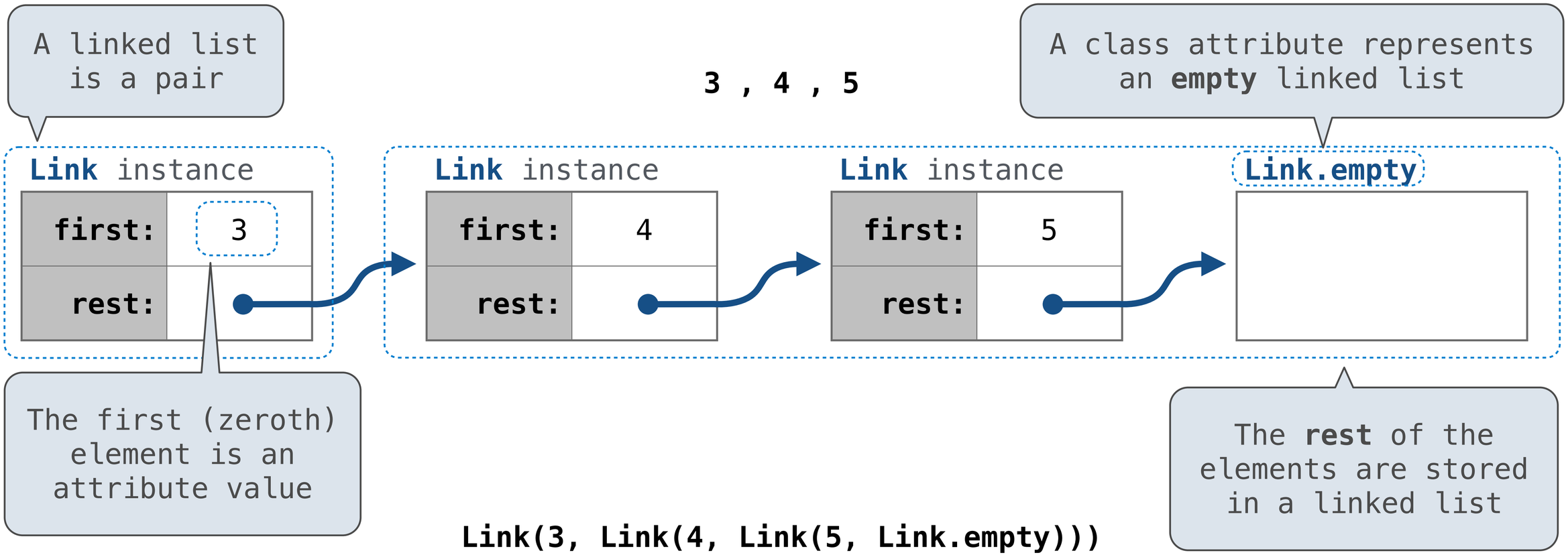
Linked Lists

Announcements

Linked Lists

Linked List Structure

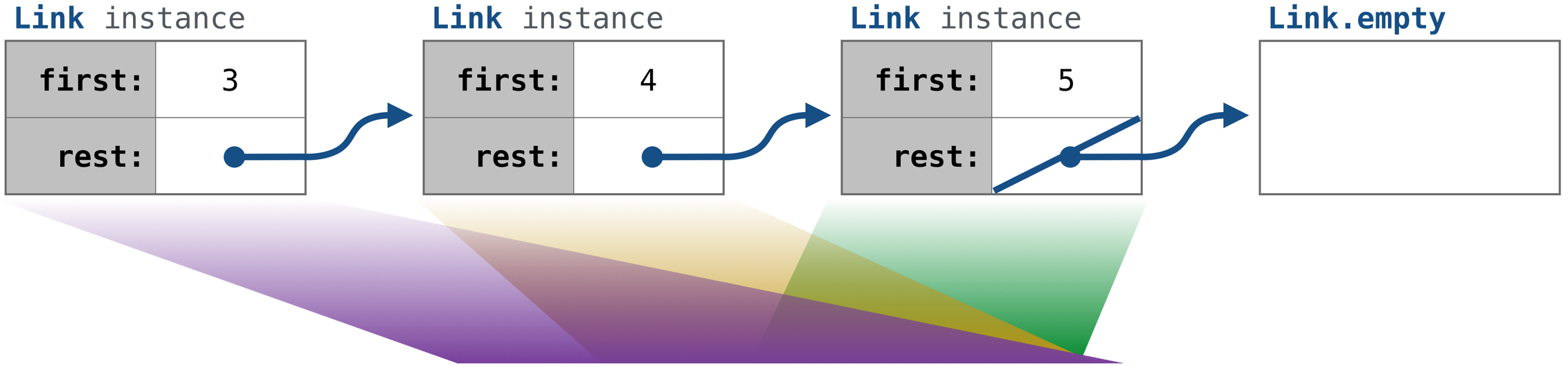
A linked list is either empty **or** a first value and the rest of the linked list



Linked List Structure

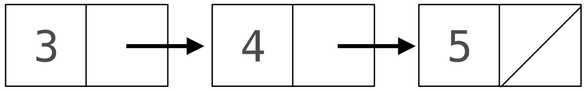
A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5




```
Link(3, Link(4, Link(5, Link.empty)))
```

<3 4 5>



Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Some zero-length sequence

Returns whether rest is a Link

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5              )))
```

(Demo)

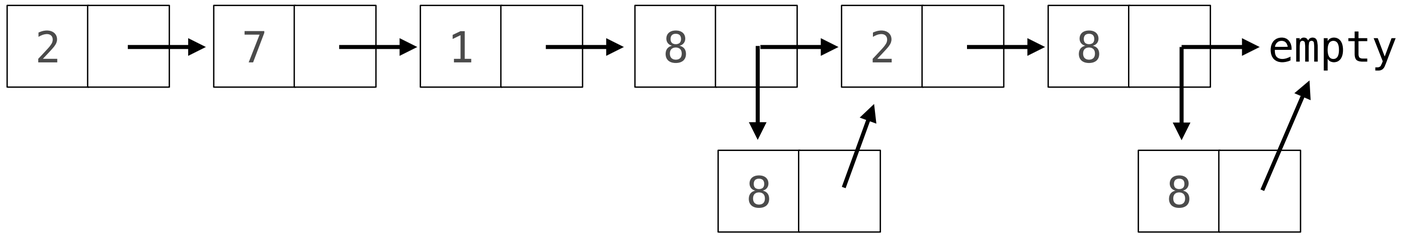
Repeated Inserts

Double a List

```
def double(s, v):
    """Insert another v after each v in list s.

    >>> s = [2, 7, 1, 8, 2, 8]
    >>> double(s, 8)
    >>> s
    [2, 7, 1, 8, 8, 2, 8, 8]
    """
    i = 0
    while i < len(s):
        if s[i] == v:
            s.insert(i+1, v)
            i += 2
        else:
            i += 1
```

Double a Linked List

```
def double_link(s, v):
    """Insert another v after each v in linked list s.

    >>> t = Link(2, Link(7, Link(1, Link(8, Link(2, Link(8))))))
    >>> double_link(t, 8)
    >>> print(t)
    <2 7 1 8 8 2 8 8>
    """
    while s is not Link.empty:
        if s.first == v:
            s.rest = Link(v, s.rest)
            s = s.rest.rest
        else:
            s = s.rest
```



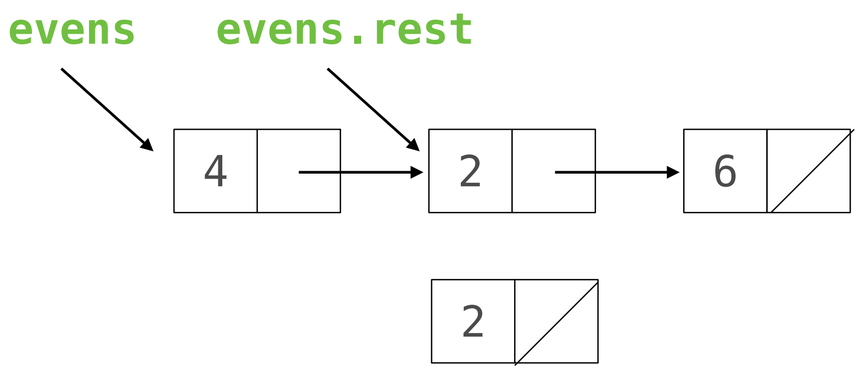
Break: 5 minutes

Linked List Practice

Slicing a Linked List

Normal slice notation (such as s[1:3]) doesn't work if s is a linked list.

```
def slice_link(s, i, j):
    """Return a linked list containing elements from i:j.

    >>> evens = Link(4, Link(2, Link(6)))
    >>> slice_link(evens, 1, 100)
    Link(2, Link(6))
    >>> slice_link(evens, 1, 2)
    Link(2)
    >>> slice_link(evens, 0, 2)
    Link(4, Link(2))
    >>> slice_link(evens, 1, 1) is Link.empty
    True
    """
    assert i >= 0 and j >= 0
    if  j == 0  or s is Link.empty:
        return Link.empty
    elif  i == 0 :
        return Link(s.first,  slice_link(s.rest, i, j-1) )
    else:
        return slice_link(s.rest,  i-1  ,  j-1  )
```



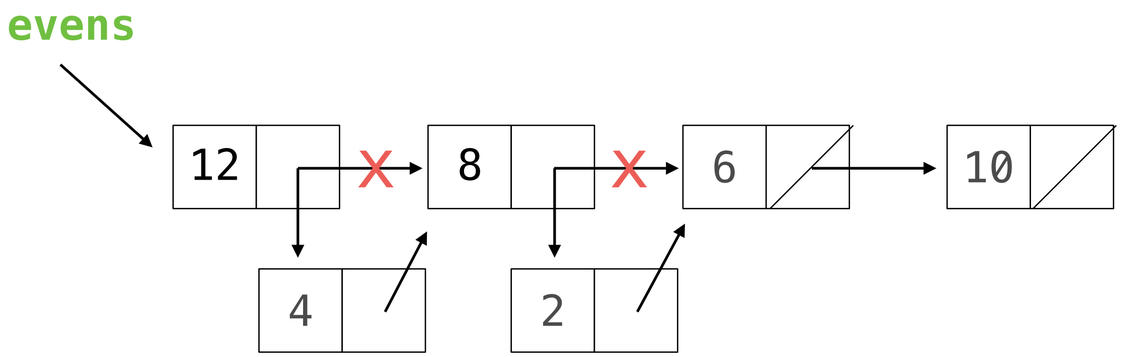
slice_link(evens, 1, 2) returns

slice_link(evens.rest, 0, 1) links 2 to

slice_link(evens.rest.rest, 0, 0) returns Link.empty

Inserting into a Linked List

```
def insert_link(s, x, i):
    """Insert x into linked list s at index i.

    >>> evens = Link(4, Link(2, Link(6)))
    >>> insert_link(evens, 8, 1)
    >>> insert_link(evens, 10, 4)
    >>> insert_link(evens, 12, 0)
    >>> insert_link(evens, 14, 10)
    Index out of range
    >>> print(evens)
    <12 4 8 2 6 10>
    """
    if s is Link.empty:
        print('Index out of range')
    elif i == 0:
        second = Link(s.first, s.rest)
        s.first = x
        s.rest = second
    elif i == 1 and s.rest is Link.empty:
        s.rest = Link(x)
    else:
        insert_link(s.rest, x, i-1)
```

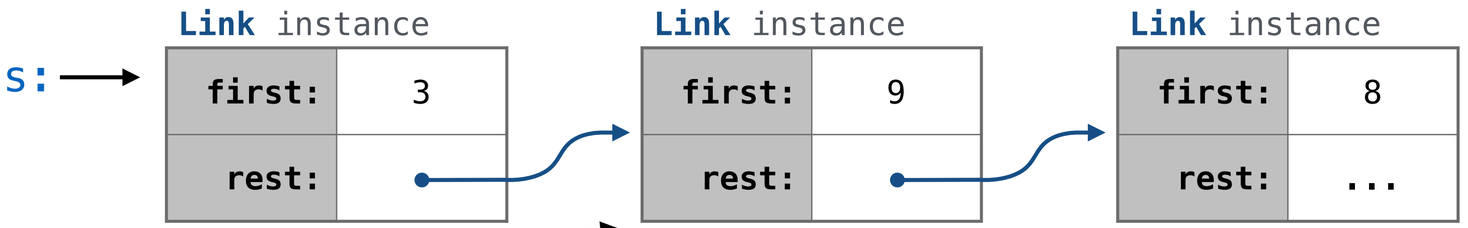


Spring 2023 Midterm 2 Question 3(b)

Definition. A *prefix sum* of a sequence of numbers is the sum of the first n elements for some positive length n.

Implement tens, which takes a non-empty linked list of numbers s represented as a Link instance. It prints all of the prefix sums of s that are multiples of 10 in increasing order of the length of the prefix.

```
def tens(s):
    """Print all prefix sums of Link s that are multiples of ten.
    >>> tens(Link(3, Link(9, Link(8, Link(10, Link(0, Link(14, Link(6)))))))
    20
    30
    30
    50
    """
    def f(suffix, total):
        if total % 10 == 0:
            print(total)

        if ____suffix is not Link.empty____:
            ____f(suffix.rest, total + suffix.first)____

    ____f(s.rest, s.first)____
```



More Linked Lists Practice

Recursion and Iteration

Many linked list processing functions can be written both iteratively and recursively

Recursive approach:

- What recursive call do you make?
- What does this recursive call do/return?
- How is this result useful in solving the problem?

```
def length(s):
    """The number of elements in s.

    >>> length(Link(3, Link(4, Link(5))))
    3
    """
    if s is Link.empty:
        return 0
    else:
        return 1 + length(s.rest)
```

Iterative approach:

- Describe a process that solves the problem.
- Figure out what additional names you need to carry out this process.
- Implement the process using those names.

```
def length(s):
    """The number of elements in s.

    >>> length(Link(3, Link(4, Link(5))))
    3
    """
    k = 0
    while s is not Link.empty:
        s, k = s.rest, k + 1
    return k
```

Constructing a Linked List

Build the rest of the linked list, then combine it with the first element.


```
s = Link.empty
s = Link(5, s)
s = Link(4, s)
s = Link(3, s)
```

```
def range_link(start, end):
    """Return a Link containing consecutive
    integers from start up to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    else:
        return Link(start, range_link(start + 1, end))
```

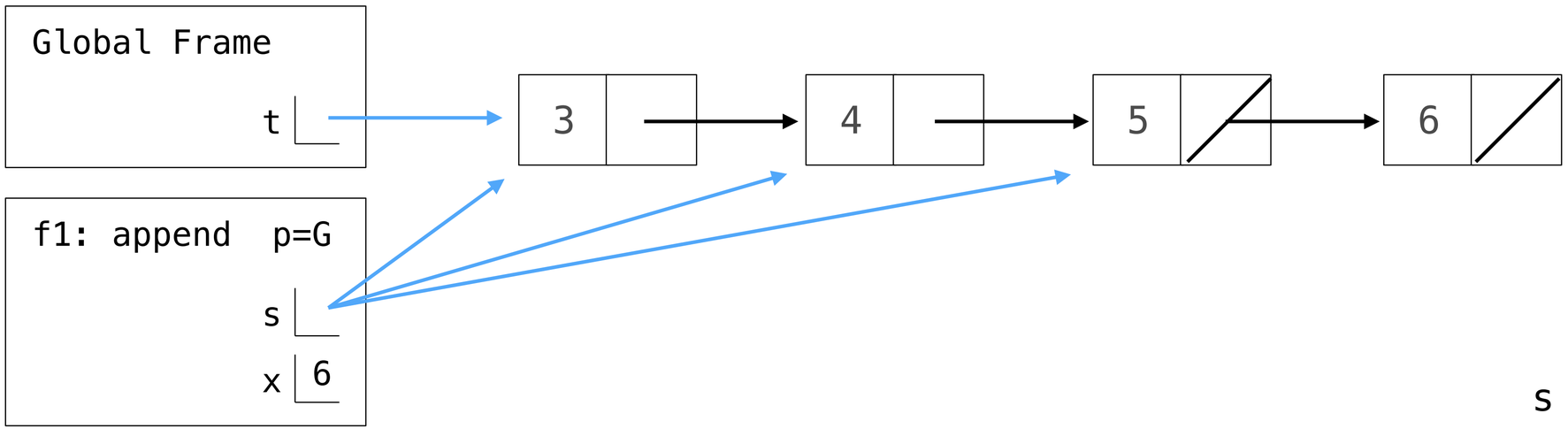
```
def range_link(start, end):
    """Return a Link containing consecutive
    integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    s = Link.empty
    k = end - 1
    while k >= start:
        s = Link(k, s)
        k -= 1
    return s
```

Linked List Mutation

To change the contents of a linked list, assign to first and rest attributes

Example: Append x to the end of non-empty s

```
>>> t = Link(3, Link(4, Link(5)))
>>> append(t, 6)
>>> t
Link(3, Link(4, Link(5, Link(6))))
```




```
s = s.rest

s.rest = Link(x)
```

Recursion and Iteration

Many linked list processing functions can be written both iteratively and recursively

Recursive approach:

- What recursive call do you make?
- What does this recursive call do/return?
- How is this result useful in solving the problem?

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)  # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    if ___s.rest is not Link.empty___:
        append(___s.rest___, ___x___)
    else:
        ___s.rest = Link(x)___
```

Iterative approach:

- Describe a process that solves the problem.
- Figure out what additional names you need to carry out this process.
- Implement the process using those names.

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)  # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    while ___s.rest is not Link.empty___:
        ___s = s.rest___
    ___s.rest = Link(x)___
```

Example: Pop

Implement pop, which takes a linked list s and positive integer i. It removes and returns the element at index i of s (assuming s.first has index 0).

```
def pop(s, i):
    """Remove and return element i from linked list s for positive i.
    >>> t = Link(3, Link(4, Link(5, Link(6))))
    >>> pop(t, 2)
    5
    >>> pop(t, 2)
    6
    >>> pop(t, 1)
    4
    >>> t
    Link(3)
    """
    assert i > 0 and i < length(s)
    for x in range(  i - 1  ):
        s = s.rest
    result = s.rest.first
    s.rest = s.rest.rest
    return   result
```