

Welcome to Data C88C!

Lecture 15: Linked Lists

Monday, July 21st, 2025

Week 5

Summer 2025

Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

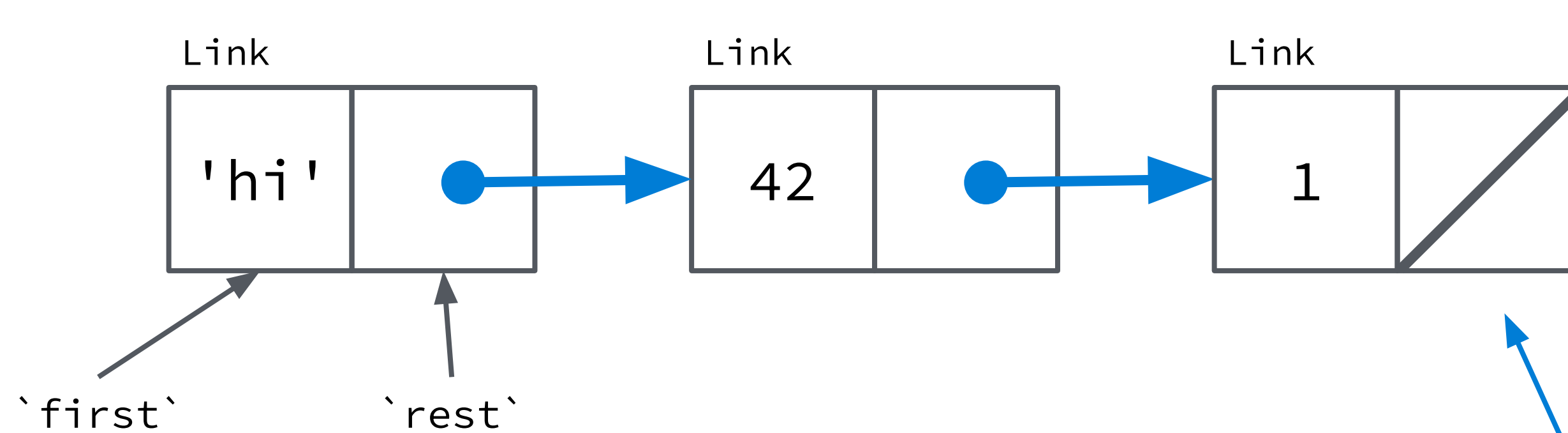
- Project 02 ("Ants") released today!
 - Checkpoint: due August 4th
 - All due: August 11th
- Mid-semester feedback form: [\[link\]](#)
 - Submit by Wednesday July 28th 11:59pm. If 75% of the class completes the survey by the deadline, we will give **1 point of extra credit** to everyone!
- Midterm grades will be released by Friday July 25th
 - "Change Grade Option" deadline: August 1st

Lecture Overview

- Linked Lists
 - Destructive vs non-destructive functions

Linked Lists: `Link` (Recap)

- A fundamental data structure. Consists of a value (`first`) and the remaining values (`rest`).
- **Recursively defined:** `rest` is itself a `Link` instance
- **Heterogeneous elements:** values in `Link` can be anything (eg a mix of ints and strs, or even other `Link` instances!)



At a high level, contains the values: 'hi', 42, 1
The above translated in terms of `Link`:

```
lst = Link('hi', Link(42, Link(1)))
```

```
class Link:
    """A linked list with a first element
    and the rest."""
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

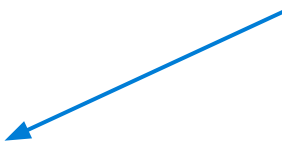
Default
argument
value

Tip: `empty` is a special value
("**sentinel**") that represents
the end of the list.

Exercise: Finding the Longest Song in a linked list

Question: Given a linked list of Songs, which Song is the longest? Implement it **recursively** first.

```
def longest_song(album):  
    if album == Link.empty:  
        return None  
  
    if album.rest == Link.empty:  
        return album.first  
    rest_longest = longest_song(album.rest)  
    if album.first.length > rest_longest.length:  
        return album.first  
    else:  
        return rest_longest
```

"Edge" case


```
class Song:  
    def __init__(self, name, artist, length):  
        self.name = name  
        self.artist = artist  
        self.length = length  
    def __repr__(self):  
        return f"Song({self.name},{self.artist},{self.length})"
```

```
>>> song1 = Song("Golden Slumbers", "The Beatles", 92)  
>>> song2 = Song("A Day In The Life", "The Beatles", 337)  
>>> album1 = Link(song1, Link(song2))  
>>> longest_song(album1)  
Song(A Day In The Life, The Beatles, 337)
```

Recursive approach:

Base case: the longest song of a one-song album is the song itself.

Recursive structure: to calculate the longest song of an N-song album, first calculate the longest song of the last (N-1) songs, then compare it with the first song.

Exercise: Finding the Longest Song in a linked list

Question: Given a linked list of Songs, which Song is the longest? Implement it **iteratively** (for/while loop).

```
def longest_song_iter(album):
    if album == Link.empty:
        return None
    cur_link, cur_longest = album.rest, album.first
    while cur_link != Link.empty:
        -----
        if cur_link.first.length > cur_longest.length:
        -----
            cur_longest = cur_link.first
        -----
        cur_link = cur_link.rest
        -----
    return cur_longest
```

`cur_link`: a "pointer" that steps through the linked list.
This is a common pattern.

```
class Song:
    def __init__(self, name, artist, length):
        self.name = name
        self.artist = artist
        self.length = length
    def __repr__(self):
        return f"Song({self.name},{self.artist},{self.length})"
```

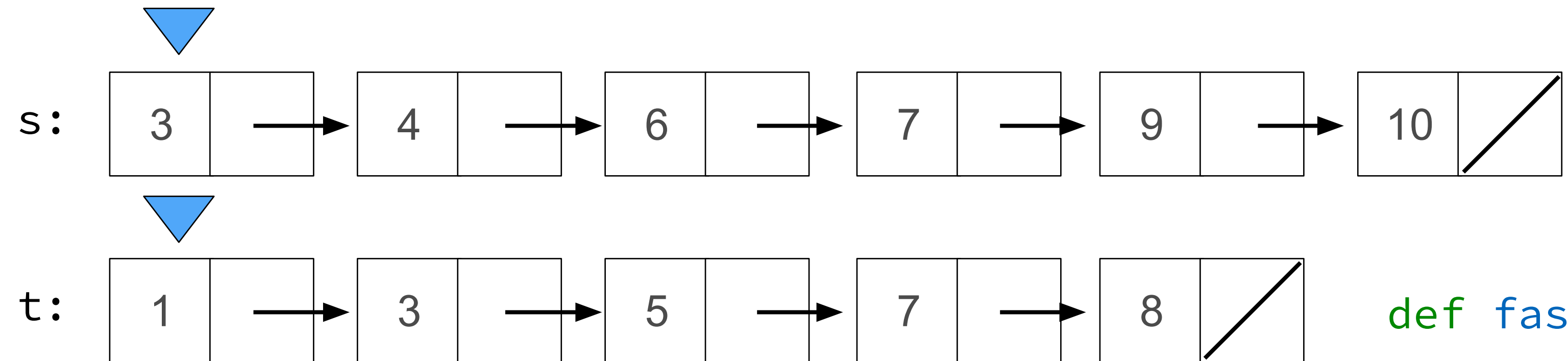
```
>>> song1 = Song("Golden Slumbers", "The Beatles", 92)
>>> song2 = Song("A Day In The Life", "The Beatles", 337)
>>> album1 = Link(song1, Link(song2))
>>> longest_song_iter(album1)
Song(A Day In The Life, The Beatles, 337)
```

Implementation idea: "walk" through the linked list, and keep track of the longest song.
When you reach the end of the list (`empty`), return the longest song encountered.

Discussion 8

Linear-Time Intersection of Sorted Linked Lists

Given two sorted **linked lists** with no repeats, return the number of elements that appear in both.



Implement it **recursively** and **iteratively**.

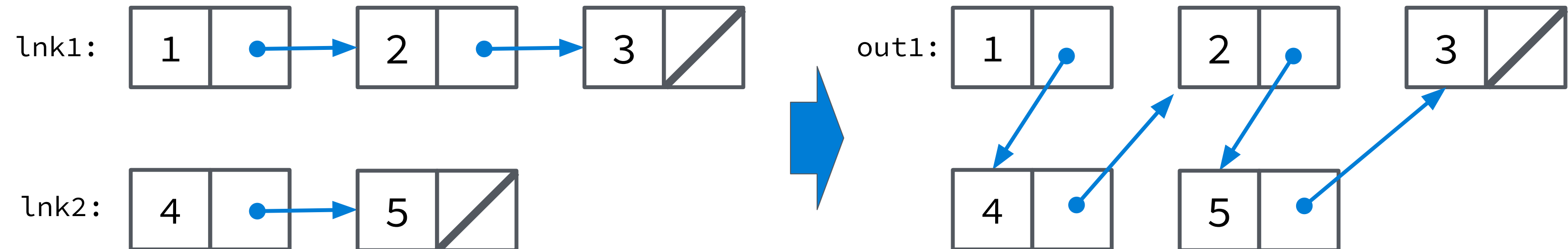
```
def fast_overlap(s, t):
    if s is Link.empty or t is Link.empty:
        return 0
    if s.first == t.first:
        return 1 + fast_overlap(s.rest, t.rest)
    elif s.first < t.first:
        return fast_overlap(s.rest, t)
    elif s.first > t.first:
        return fast_overlap(s, t.rest)
```

```
def fast_overlap(s, t):
    k = 0
    while s and t:
        if s.first == t.first:
            k, s, t = k + 1, s.rest, t.rest
        elif s.first < t.first:
            s = s.rest
        elif s.first > t.first:
            t = t.rest
    return k
```


Link interleave: "constructively"

Question: given two linked lists `lnk1, lnk2`, create a **new** linked list that contains the elements of `lnk1, lnk2` interleaved as follows:

```
>>> lnk1 = Link(1, Link(2, Link(3)))
>>> lnk2 = Link(4, Link(5))
>>> out1 = interleave(lnk1, lnk2)
>>> out1
<1 4 2 5 3>
```



Implement it **recursively**:

```
def interleave(lnk1, lnk2):
    if lnk1 == Link.empty:
        return link_copy(lnk2)
    elif lnk2 == Link.empty:
        return link_copy(lnk1)
    else:
        out_rest = interleave(lnk1.rest, lnk2.rest)
        return Link(lnk1.first, Link(lnk2.first, out_rest))
```

Question: why is it important for us to do `return link_copy(lst2)`? What if we instead did `return lst2`?

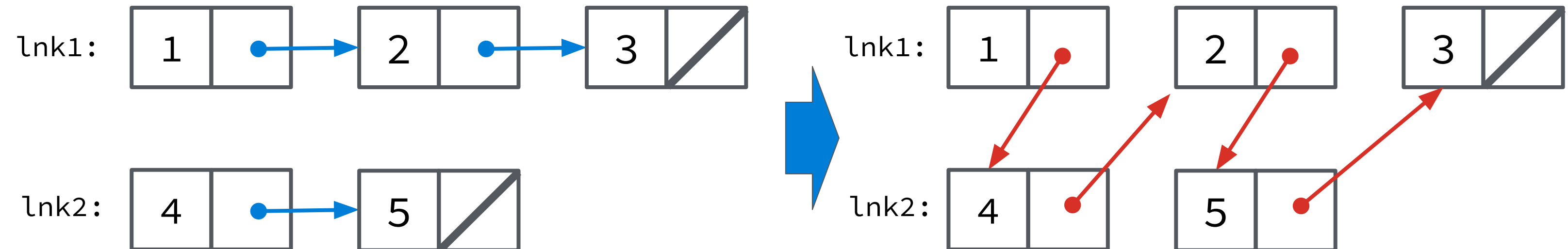
Answer: `return lst2` would appear to work, but it wouldn't be creating a **copy** of `lst2`: changes to `lst2` would propagate to `out1`, which violates our requirement to create a **new** linked list.

```
def link_copy(lnk):
    if lnk == Link.empty:
        return lnk
    return Link(lnk.first, link_copy(lnk.rest))
```

Link interleave: "destructively"

Question: given two linked lists `lnk1, lnk2`, **modify** `lnk1, lnk2` such that `lnk1` contains the elements of `lnk1, lnk2` interleaved as follows:

```
>>> lnk1 = Link(1, Link(2, Link(3)))
>>> lnk2 = Link(4, Link(5))
>>> interleave_mut(lnk1, lnk2)
>>> lnk1
<1 4 2 5 3>
```



You must not create any new `Link` instances, instead modify the input linked lists **"in place"**. Implement it **recursively**:

```
def interleave_mut(lnk1, lnk2):
    if lnk1 == Link.empty or lnk2 == Link.empty:
        return
    else:
        out_rest = interleave_mut(lnk1.rest, lnk2.rest)
        lnk1.rest = lnk2
        lnk2.rest = out_rest
```

Linked list "surgery": changing `rest` pointers!

Regarding "constructive" and "destructive" functions

| Type | Behavior | Pros | Cons |
|-------------------------------------|--|---|--|
| Constructive ("non-destructive") | Creates "new" things. Ex: "Given a list of integers, return a new list of integers with every integer squared" | Implementation is typically easier. Constructive code is often easier to understand. | Typically less performant than destructive functions, due to overhead of creating new instances (and additional memory usage). |
| Destructive | Modifies (mutates) inputs "in place". Ex: "Given a list of integers, mutate the list so that each value is squared" | Can be more performant than constructive functions. | Implementation is typically trickier to get right. Destructive code can be tricky to debug. |

(Personal advice) prioritize **legibility** and **ease of maintenance**, and prefer constructive code until you have to care about performance.
And even then: only optimize the code that is actually slow!
Requires benchmarking/profiling your code to identify slow spots. Often does not even boil down to "constructive" vs "destructive" code anyways...

"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**": Donald Knuth