COMP_SCI 214: Data Structures and Algorithms

Linked Lists

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 1 due Thursday
  - If you downloaded code Thursday (1/4), make sure your code has the correct version of `in_class_modifier` (see @10 on Piazza)
- Recommendation: save late tokens later in the quarter
  - Can be used for one deadline on all non-exam assignments (resubmissions and main submissions are separate)
- Read Office hours guidelines!! (See top of canvas and OH page)
  - Makes OH smoother and efficient
  - Ensures you get practice with identifying and articulating issues

Wrapping up basics of DSSL2

Recap:
What are data structures made of?

- Two concrete building blocks for representing all data structures:
  - Vectors (or arrays)
  - Structs

Vectors

- Vectors contain sequences of data
  - *Indexed* by *integers* $0 \dots n-1$
  - Contents typically all of the same type
- Size fixed when *creating* the *specific vector*
  - A vector is created with size 10 → it has size 10 forever
  - But can create vector of different sizes
- Vectors take the same time to access an element…
  - Regardless of how far into the vector!
  - Regardless of how big the vector is!

0	1	2
15	1045	3

0	1	2	3	4	5
”red”	”yellow”	”puce”	”brown”	”black”	”purple”

Structs

- Structs contain collections of data
  - *Accessed* by field *names*
  - Fields can be (and often are) of different types
- Fields determined when *defining* the struct *type*
  - Accessing any field takes the same amount of time

Positions have an xand a y field, both numbers.

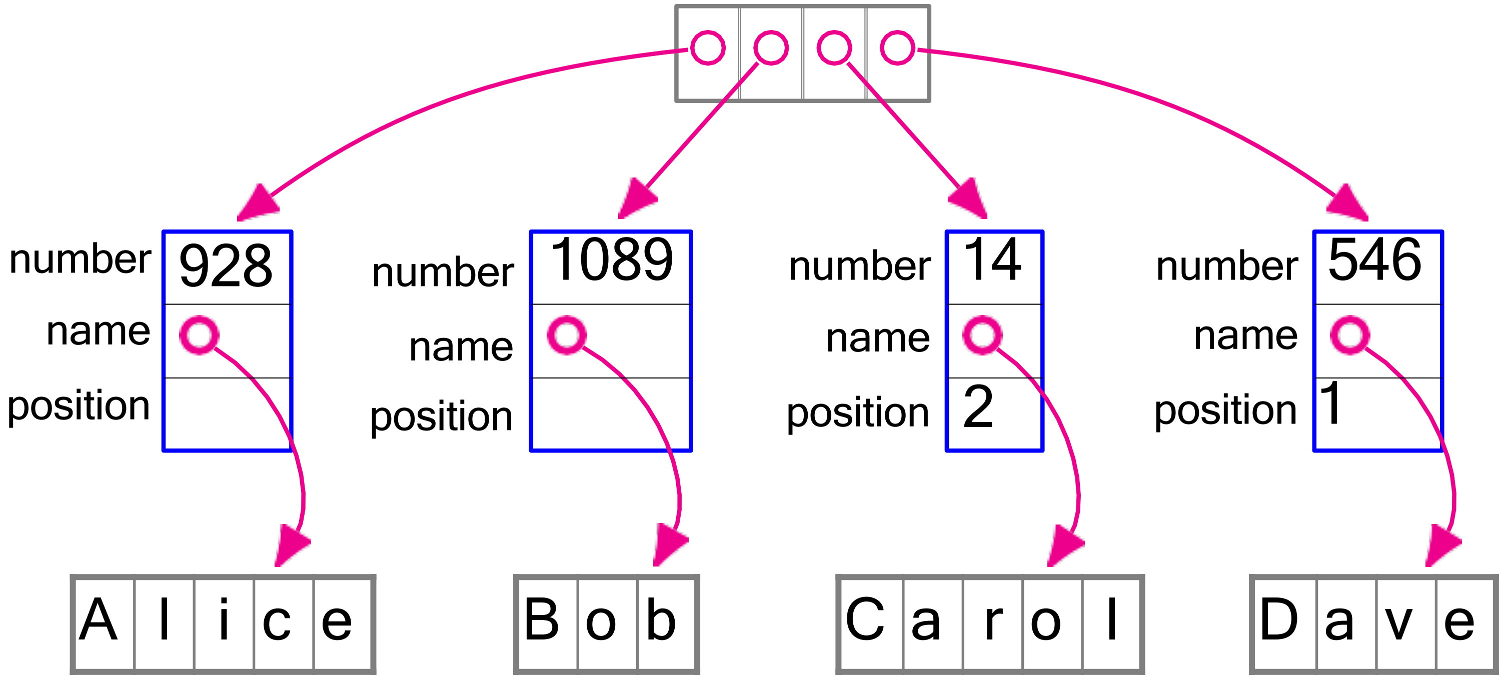
x. -3
y. 4

Runners have a number field (integer), a namefield (string), and a position field (integer).

number 928
name "Adam"
position 4

Structs and vectors can be combined through arrows

- A vector of structs



Structs and vectors can be combined

- A vector of structs

```
struct runner:
    let number; let name; let position

let runners = [ runner( 928, "Alice", 4),
                runner(1089, "Bob",   3),
                runner(  14, "Carol", 2),
                runner( 546, "Dave",  1) ]
```

- **QUIZ**: Suppose we want to find out Carol’s position:
  - **A:** runners[3].position
  - **B:** runners[2].position
  - **C:** runners.position[2]

Contracts

- DSSL2 (like Python) does not have static types. Instead, you can use contracts to check values.
  - And catch mistakes before they snowball!
- ```
  def add_elt (x: int?, i: nat?, v: VecC[int?]) -> int?:
    return x + v[i]
  ```
  - `# contract violation! expected `nat?``
  - `assert_error add_elt(10, -5, [2, 3, 4] )`
- We'll provide some contracts in assignments
- See supplementary video on Canvas (and docs) for details.
  ```

For more DSSL2 information

- See the DSSL2 reference (or help desk).
- To search the help desk for DSSL2-specific topics (instead of every package under the sun):
  - Prefix your query with "T:dssl2"
  - For example, "T:dssl2 error" to search for DSSL2's error function
- Look at the DSSL2 documentation (https://docs.racket-lang.org/dssl2/)
  - Or search "dssl2 <something>" in Google
- Larger example: see recipes.rkt on Canvas under Lecture 1

Vectors

What is a data structure?

- A scheme for organizing data, to use it efficiently
- Two parts:
  - **Representation:**
    - Conceptual pieces of data to concrete building blocks
  - **Operations:**
    - How a client accesses and manipulates these conceptual pieces

Data structures

- Data structures are made up of building blocks
  - represented by boxes and arrows

- A vector itself is a data structure
  - Represents a collection of data of usually the same type
  - Operations may relate to assigning or updating elements

Problems with vectors

2	3	4	5	7	8	9	10	11

- What if we need to add 1 at the beginning?
- What if we need to add 6 between 5 and 7?

Vectors are like bookshelves

Stuffed bookshelf

- What if we want to add a book at an end of the shelf?
  - Not enough space
  - Need a whole new bookshelf

Vectors are like bookshelves

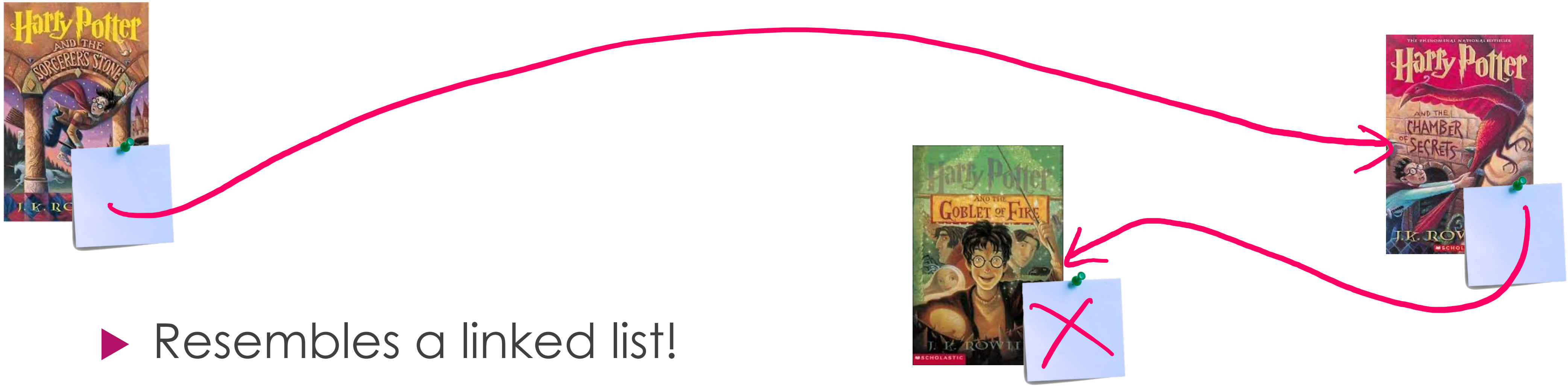
Bookshelf with space

- What if we want to add a book at in the middle of a shelf?
  - Need to shift over books on one side
  - Then insert
  - Can take a lot of time every time

Linked lists

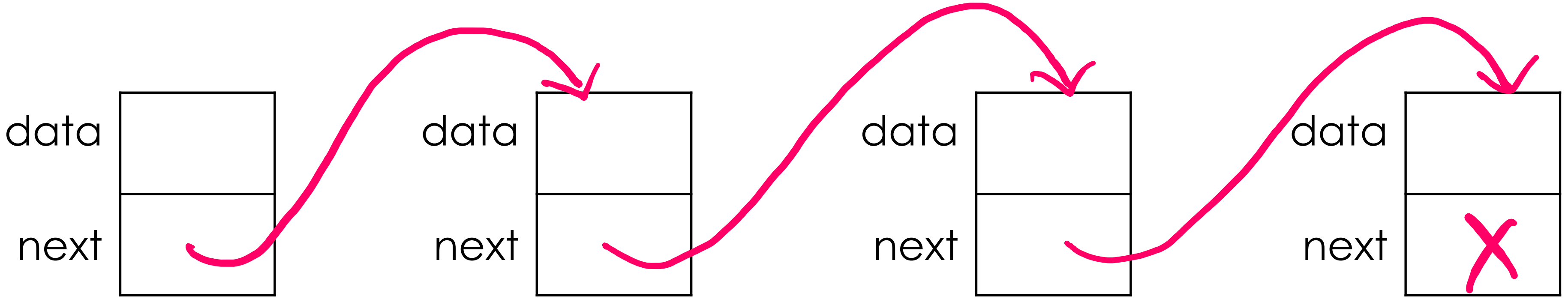
A treasure hunt for books?

- Leave books all over the house
- Each book contains a sticky at the end with location of next book



- Resembles a linked list!
  - You’ve seen this concept in 111 but they looked a bit different

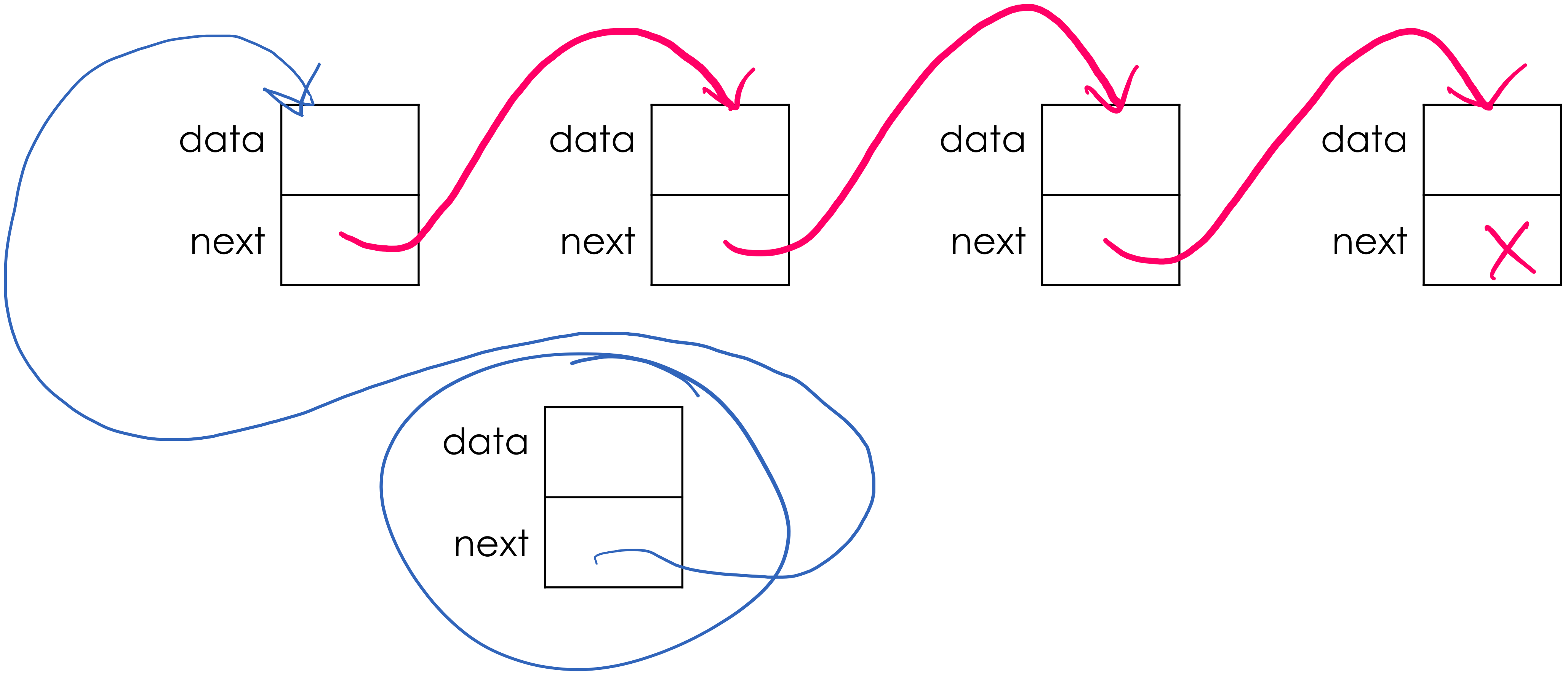
Representation of a linked list



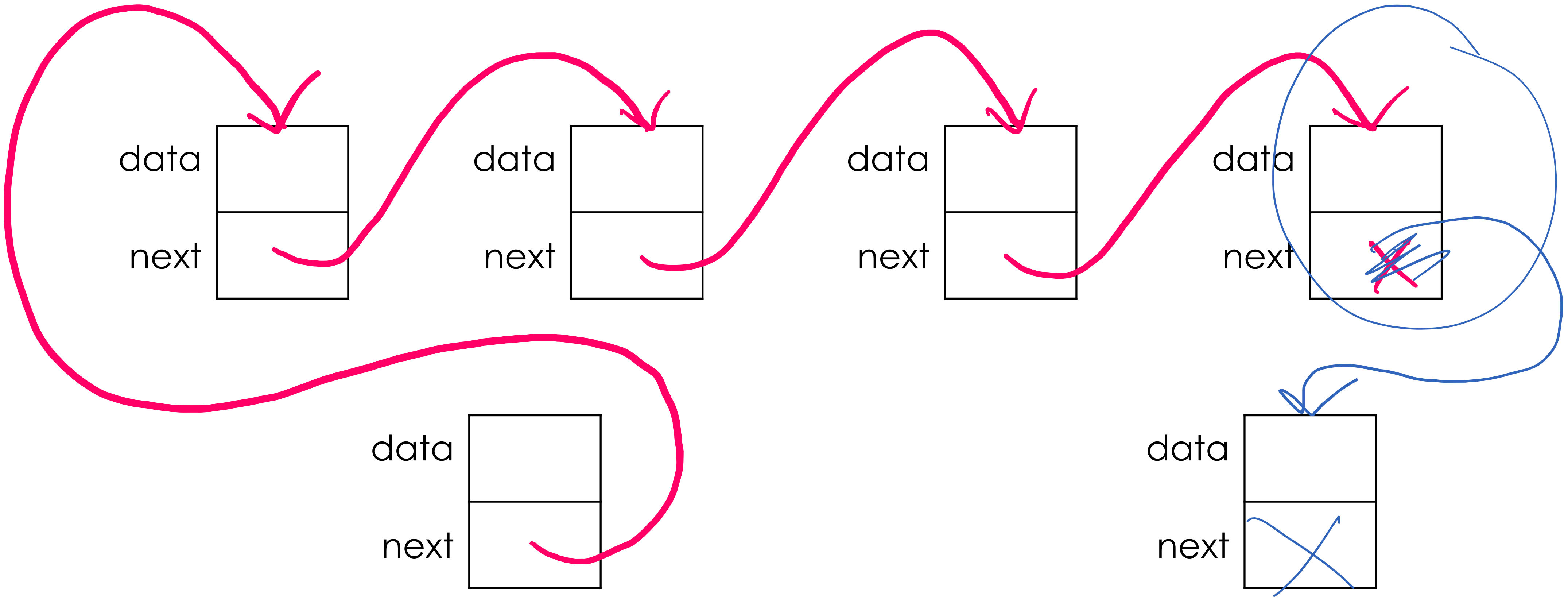
One element in the list is a `struct object` with 2 fields:

- The `data` field holds one data element
- The `next` field holds an arrow to the next node in the list

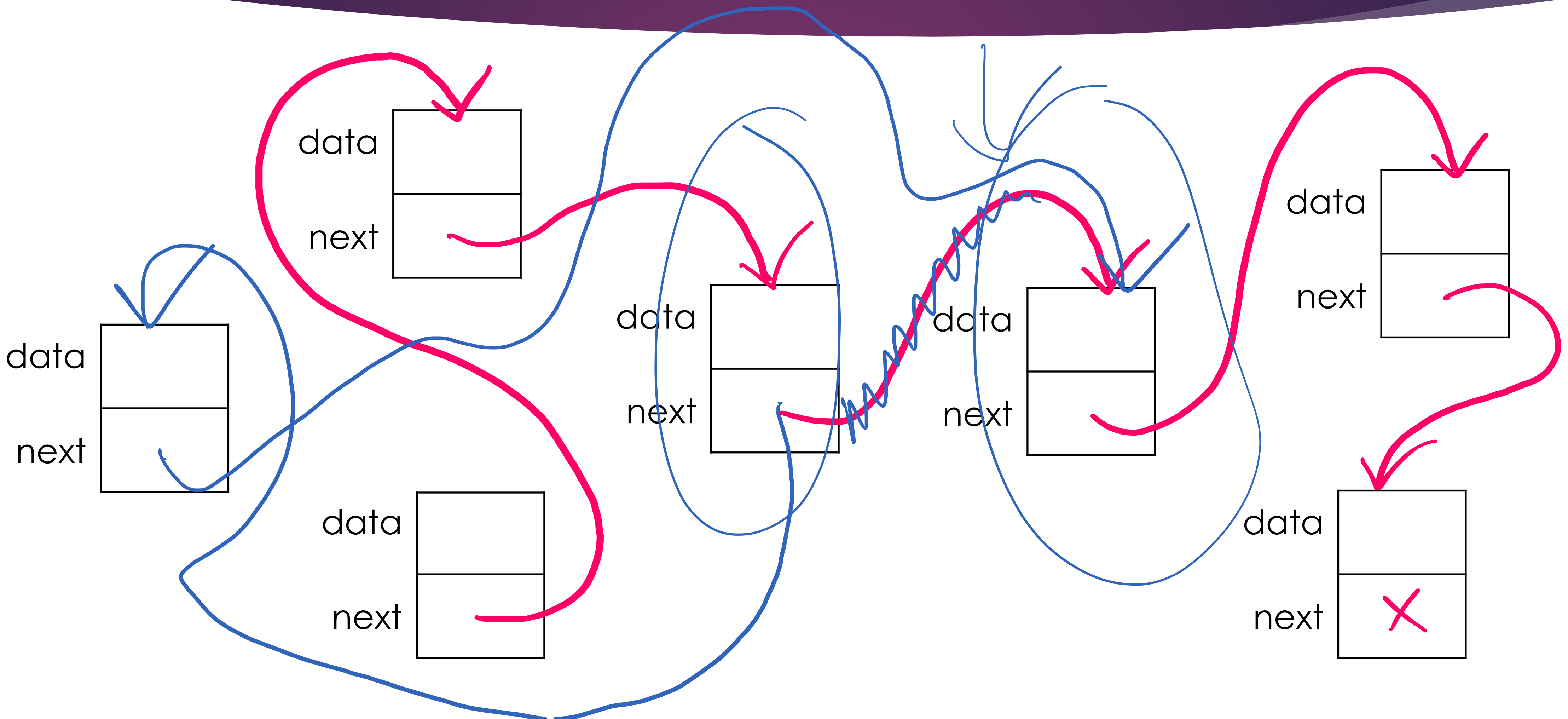
Inserting at the beginning



Inserting at the end



Inserting in the middle



Pause

- Any questions?
- Anything unclear?

Implementing linked lists

Atom of a linked list: one link

- Each link/node is a struct object that tells us: the value of the element, where the next element is

```
struct cons:    # recursive data structure
    let data
    let next
```

- A linked list is usually 1 or more `cons` objects strung together
- Often named `node` or `link` too

One link: the `cons` struct

```
# Link object is one of:
# - cons { data: Any, next: Link }
# - None
struct cons:
    let data
    let next
```

Possible `cons` values and representations

- Value: `None`
  - Empty list (visually: `[]`)
- Value: `cons(1, None)`
  - A list with one element `1` (visually: `[1]`)
- Value: `cons(1, cons(2, None))`
  - A list with two elements: `1` and `2` (visually: `[1, 2]`)
- Value: `cons(None, None)`
  - A list with one element: `None` (visually: `[None]`)
  - Also could be a list of lists with one empty list as its element (visually: `[[]]`)

```
struct cons:
    let data
    let next
```

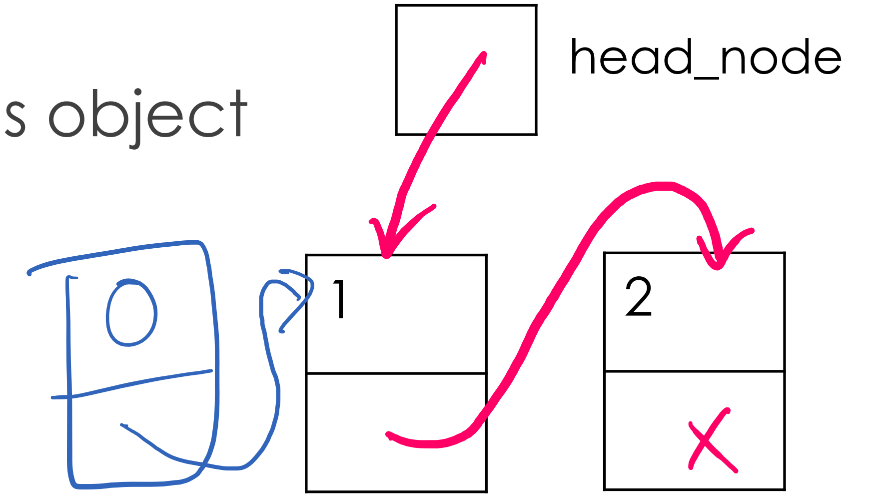
Representing and using a linked list

- Any client code that uses a linked list will want access to a variable representing the list
  - `let ll = <list>`

- Attempt #1: headerless linked list:
  - List variable is just the first node in the list

- Attempt #2: headerful linked list:
  - List variable contains a variable that is the first node in the list

Headerless linked list



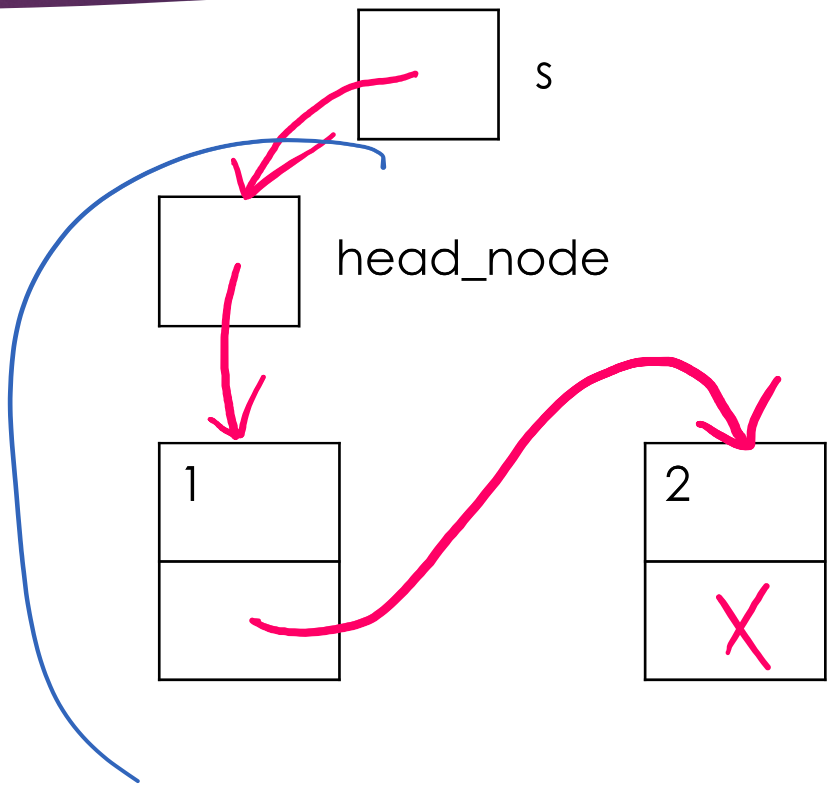
- List is represented as pointer to first cons object
  - All you need
  - Follow this first object to get to the rest
- Consider the client code:
  - `let head_node = cons(1, (2, None))`
  - `insert_front(head_node, 0)`
- What do you want `head_node` to contain after the insertion?
- Client burden (and no guarantees they'll do it or do it right)
  - Client needs to do `head_node = insert_front(head_node, 0)`

Headerful linked list – what we'll use

- Add a wrapper around a headerless linked list
- Linked list class:

```
class SLL: # SLL = Singly-Linked List
   let head
   # insert methods including __init__()
```

- Client code:
  - `let s = SLL()`
  - `s.remove_front()`
- Will `s` contain the updated head?
  - If the SLL class can guarantee it, then yes!



Headerful linked list

```
class SLL: # SLL = Singly-Linked List
    let head   # For representation

    def __init__(self):
        self.head = None

    def get_first(self):   # Operations
        if cons?(self.head):        not empty
            return self.head.data
        else: error('empty list')
```

Pause

- Any questions?
- Anything unclear?

More operations

- `get_nth`
- `set_nth`
- `get_last`
- `len`
- `insert_front`

Before we write operations

- How do we make sure our operations don't mess up the list?
- How can we check whether the list is still valid after the operation?

- We first need a notion of a "valid" list
- The list should be valid before and after each operation

Representation invariant

- All data structures have properties that always hold true for the DS to be be meaningful: **a representation invariant**

- Satisfying the invariant leads to correct data structures and can always ensure correct operations on data structures

Linked list invariant

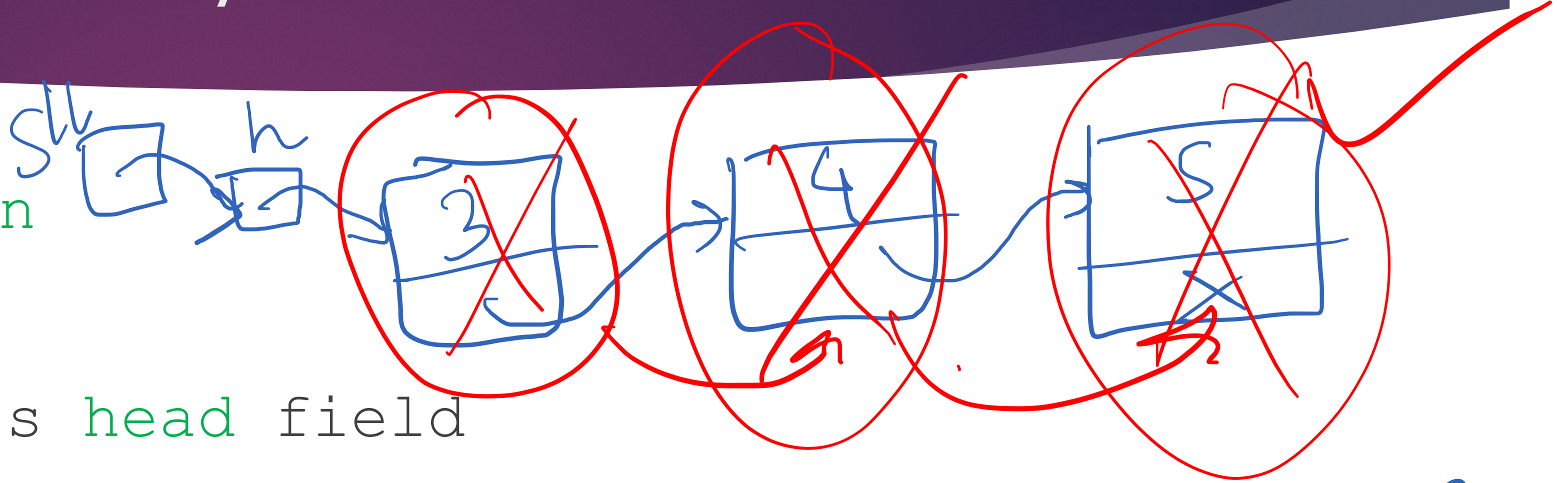
- Given the `SLL` class:
  - The `head` field contains the first node in the linked list
  - The `next` field of a node refers to the next node in the list
  - The last element in the list will always have a `next` field of `None`
  - Traversing the list from the head does not encounter any cycles and will always reach a `None` node
  - An empty list has a `head` value that is `None`

What to do with invariants?

- Often can implement these as check functions in code
  - Checked before and after each operation
  - Can get very complex and slow for production systems

- We will focus on theoretically satisfying the invariants and ensuring correctness when writing functions for operations, not checking for them programmatically

Algorithm for `get_nth` (this is pseudocode)



- Arguments: sll, n
  - count = 0
  - current = sll's head field
  - While current is not None:
    - If count == n: then return current
    - Increment count
    - current = current's next field

5

n = 2
Count = ~~0~~ ~~1~~ 2
curr =

get_nth

```
class SLL:
    …
    def get_nth(self, n):
        let count = 0
        let curr = self.head
        while not curr == None:
            if count == n:
                return curr.data
            count = count + 1
            curr = curr.next
        error('list too short')
```

Algorithm for `set_nth`

- Similar to `get_nth`
- Instead of returning, set the node's data field

set_nth

```
class SLL:
    …
    def set_nth(self, n, val):
        let count = 0
        let curr = self.head
        while not curr == None:
            if count == n:
                curr.data = val
                return
            count = count + 1
            curr = curr.next
        error('list too short')
```

More operations

- `get_nth` ✅
- `set_nth` ✅
- `get_last`
- `len`
- `insert_front`

In-class exercise up next

- Reminders:
  - Link is available on Canvas on the homepage (if on mobile: click on "Syllabus" to get to the homepage)
  - This is not an attendance quiz → graded based on engagement and specific criteria (which are specified)

- Questions are NOT to be shared with your classmates not here

In-class exercise (5 minutes)

1. Write a description in words (few sentences) or short pseudocode of how the function **`get_last`** could work given the current representation of SLL (remember you can't iterate through a linked list like a vector). Be specific about all steps.

```
class SLL: # SLL = Singly-Linked List
    let head # First node

def get_last(self):
    # returns the data in the last node in the list
```

Algorithm for `get_last`

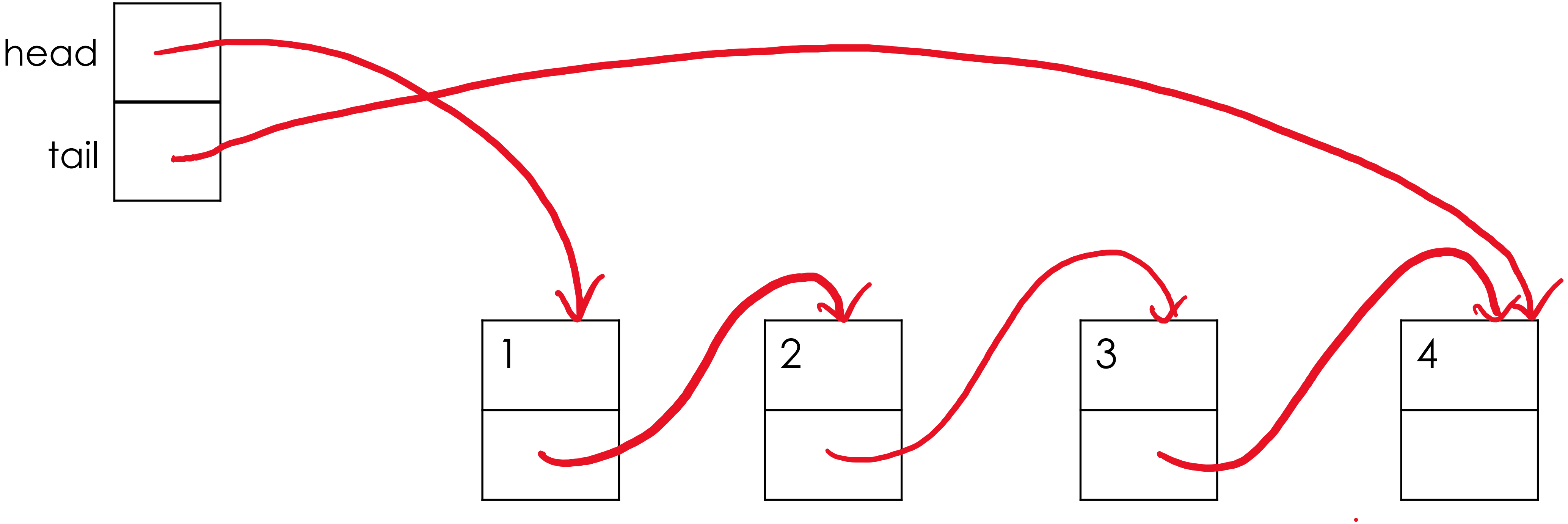
- `Arguments: sll`
  - `current = sll’s head field`
  - `While current is not None:`
    - `If current’s next field is None: then return current`
    - `current = current’s next field`

- We have to traverse the entire linked list to get to the end.
  - What if this is a function we need to keep calling?
  - Can we alter our algorithm or data about the list to make this quicker?

Let’s add more information to the SLL data structure

```
class SLL:
   let head
   let tail
```

SLL with tail field



New algorithm for `get_last`

- `Arguments: sll`
  - `If no elements in the sll:`
    - `Return the empty list (or None)`
  - `Return sll’s tail field`

Algorithm for `len`

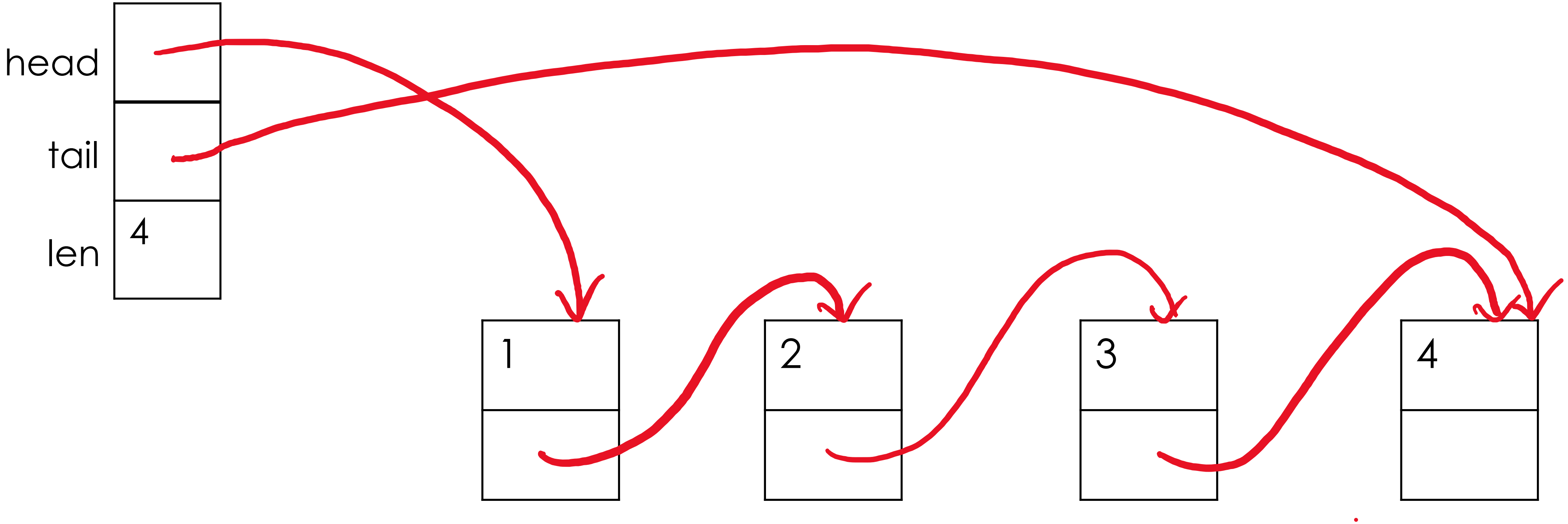
- `Arguments: sll`
  - `count = 0`
  - `current = sll's head field`
  - `While current is not None:`
    - `Increment count`
    - `current = current's next field`
  - `Return count`

- Again a full traversal
  - Can we alter our algorithm or data about the list to make this quicker?

Let’s add more information to the SLL data structure


```
class SLL:
   let head
   let tail
   let len
```

SLL with length field



New algorithm for `len`

- `Arguments: sll`
  - `Return ssl's len field`

Linked list invariants (additions in red)

- Given the `SLL` class:
  - The `head` field contains the first node in the linked list, the `tail` contains the last node in the list, the `len` field is the number of nodes in the list
  - The `next` field of a node refers to the next node in the list
  - The last node in the list (the `tail`) has a `next` field of `None`
  - Traversing the list from the head does not encounter any cycles and will always reach a `None` node; `tail` will be the last node visited
  - An empty list has a `head` and `tail` value that is `None` and a `len` of 0

Take a look at the linked list code

- Try to implement the functions yourself in the starter code
- Fill in the functions
- Only look at the completed version after you attempt

list-tail-starter.rkt
list-tail-complete.rkt

Dynamic Arrays

Dynamic arrays

- What if we want to have a resizable collection of elements?
  - Like Python's `list` or Java's `ArrayList`
- Can we implement this with any of the data structures we've seen so far?
  - Vector/Array
  - Linked list

Resizable list using vectors

dyn-array-vec.rkt

```
class DynamicArray:
    let data
    let len
    …
    def append(self, val):
        let resize_data = [None; self.len + 1]
        for i in range(self.len):
            resized_data[i] = self.data[i]
        resized_data[self.len] = value
        self.data = resized_data
        self.len = self.len + 1
```

Resizing the vector

- Here we just increased vector size by 1 each time we need to do an insert
- Doubling the vector size each time is more efficient
  - We'll see this later in the quarter!

Resizable list using linked lists

```
class DynamicArray:
    let head
    let tail
    let len
    …
    def append(self, val):
        …
```

```
struct cons:
    let data
    let next
```

Comparing implementations

- Both class implementations have the exact same operations (`get_ith`, `append`)
- The way a client would use both implementations is exactly the same

```
let arr = DynamicArray()
arr.append(2)
```

  - But each implementation is doing something else under the hood
- The dynamic array idea here is **abstract**
- The underlying vector or linked lists are **concrete** implementations
- More next time!