

# **CS 241: Systems Programming**

## **Lecture 19. Linked Lists**

Fall 2019

Prof. Stephen Checkoway

# Announcements

Project proposal due tonight

Homework 3 due tonight

# Aside: returning multiple values

In Python, functions can return multiple values (it returns a tuple)

```
def example():  
    return "example", 5
```

In C, functions cannot; instead

- Return a struct

```
struct ret_val { char const *s; int i; };  
    struct ret_val example1(void) {  
    struct ret_val r = { .s = "example", .i = 5 };  
    return r;  
}
```

# Returning multiple values (cont)

- ▶ Add pointer parameters

```
char const *example2(int *out) {  
    *out = 5;  
    return "example";  
}
```

- ▶ Use global variables

```
int example_ret;  
char const *example3(void) {  
    example_ret = 5;  
    return "example";  
}
```

# Aside 2: Avoid globals

Avoid global variables whenever possible

## Globals

- ▶ make your code difficult to reason about
- ▶ make writing correct multi-threaded code extremely difficult
- ▶ make testing individual functions difficult
- ▶ pollute the namespace because they are available everywhere
- ▶ can cause implicit coupling between separate functions

Sometimes globals are fine...but they're usually not what you want

How should a function return multiple values (in most cases)

- A. Return a struct
- B. Using pointer parameters
- C. Using global variables
- D. A or B
- E. A, B, or C

# Review from Data Structures



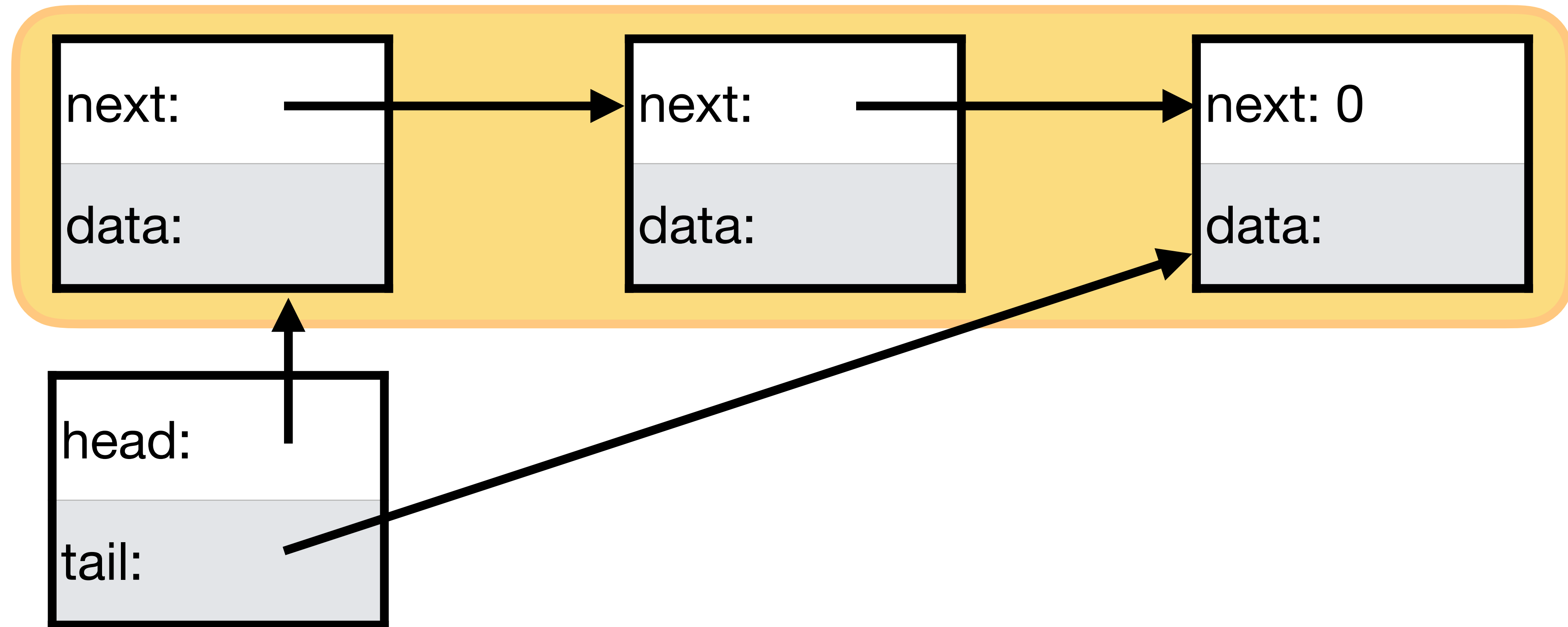
A (singly) linked list is a data structure that implements the List ADT

- ▶ Add, insert, remove elements
- ▶ Ordered by position in the list

Each node contains

- ▶ An element of the list
- ▶ A pointer to the next element in the list or 0 (**NULL**) for the last node

# Review from Data Structures



The list itself usually contains a pointer to the head of the list (first node) and the tail of the list (last node)

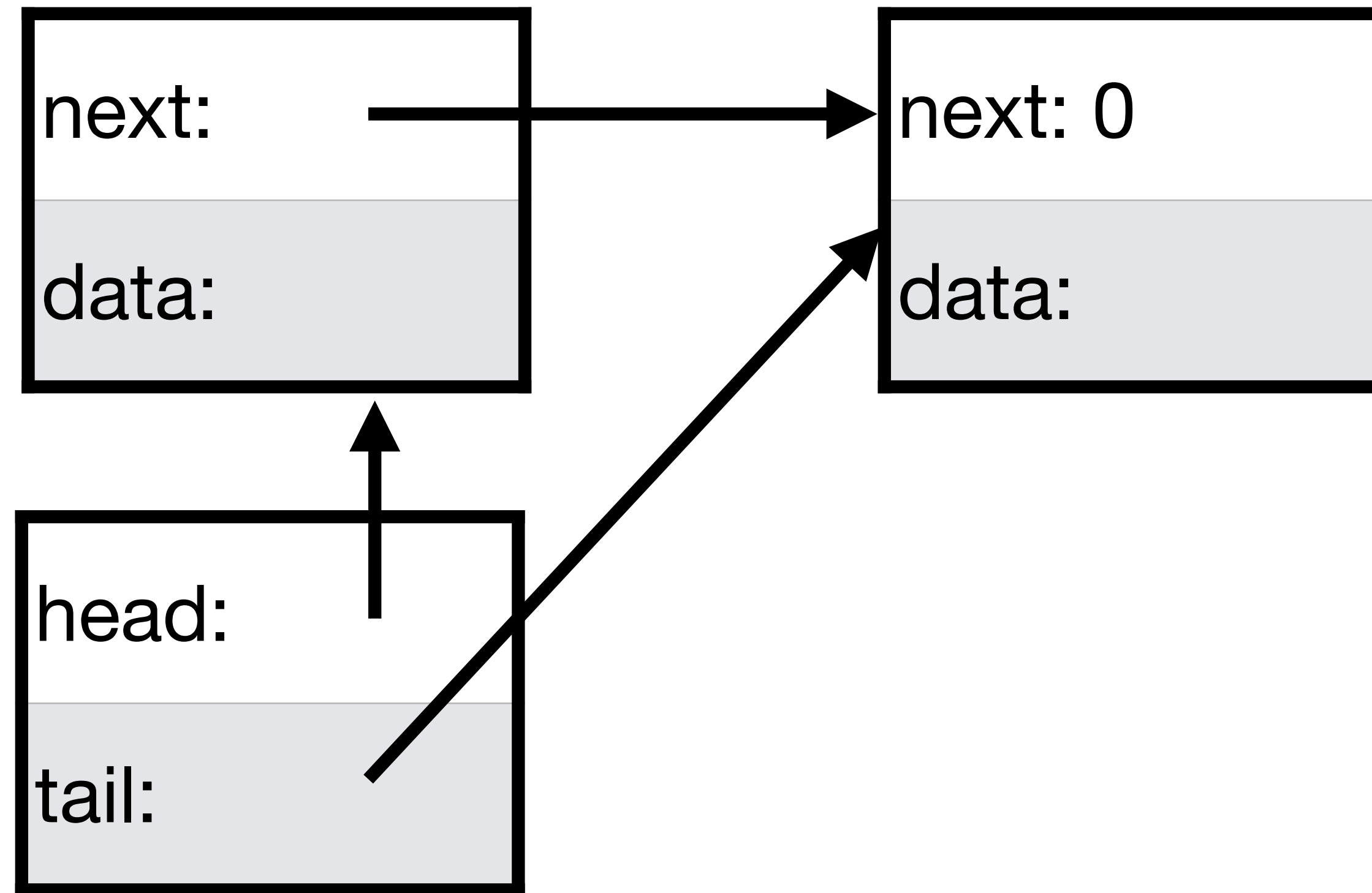


# Data types for a list of ints

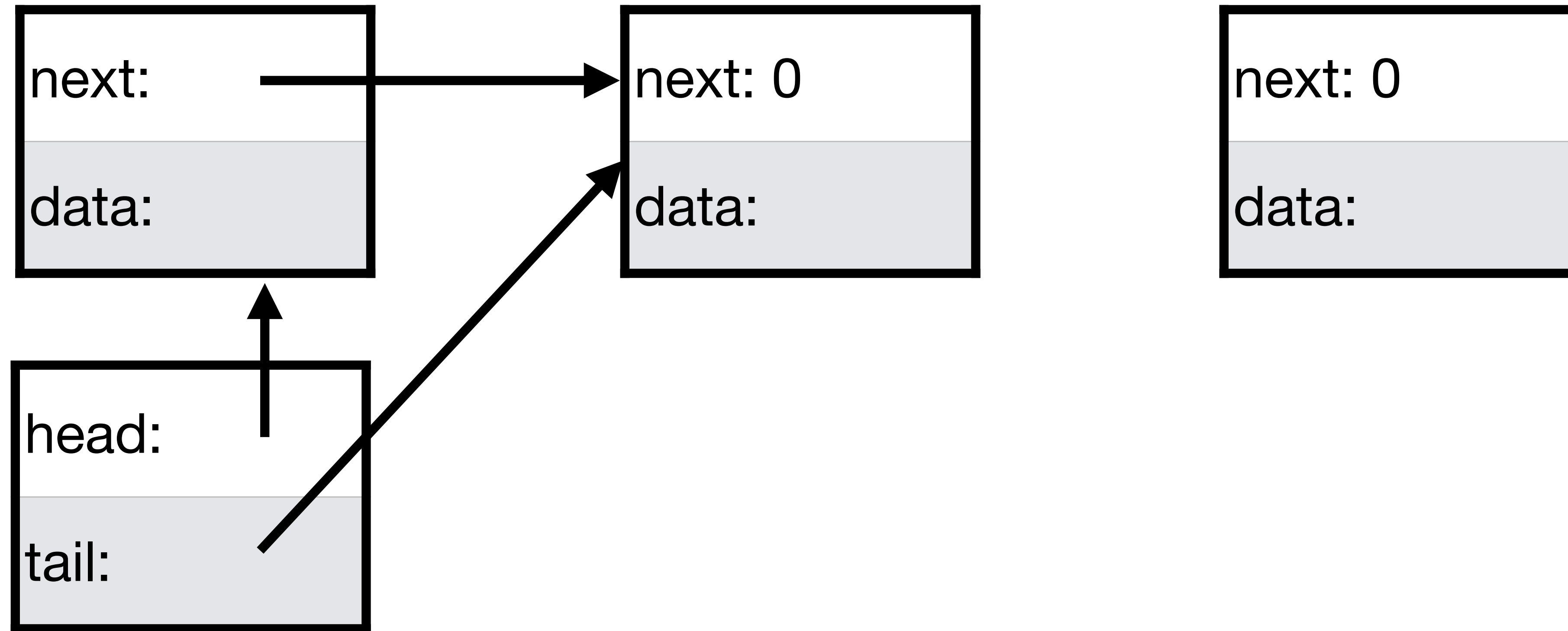
```
typedef struct Node {  
    struct Node *next;  
    int data;  
} Node;
```

```
typedef struct List {  
    Node *head;  
    Node *tail;  
} List;
```

# Appending to the list

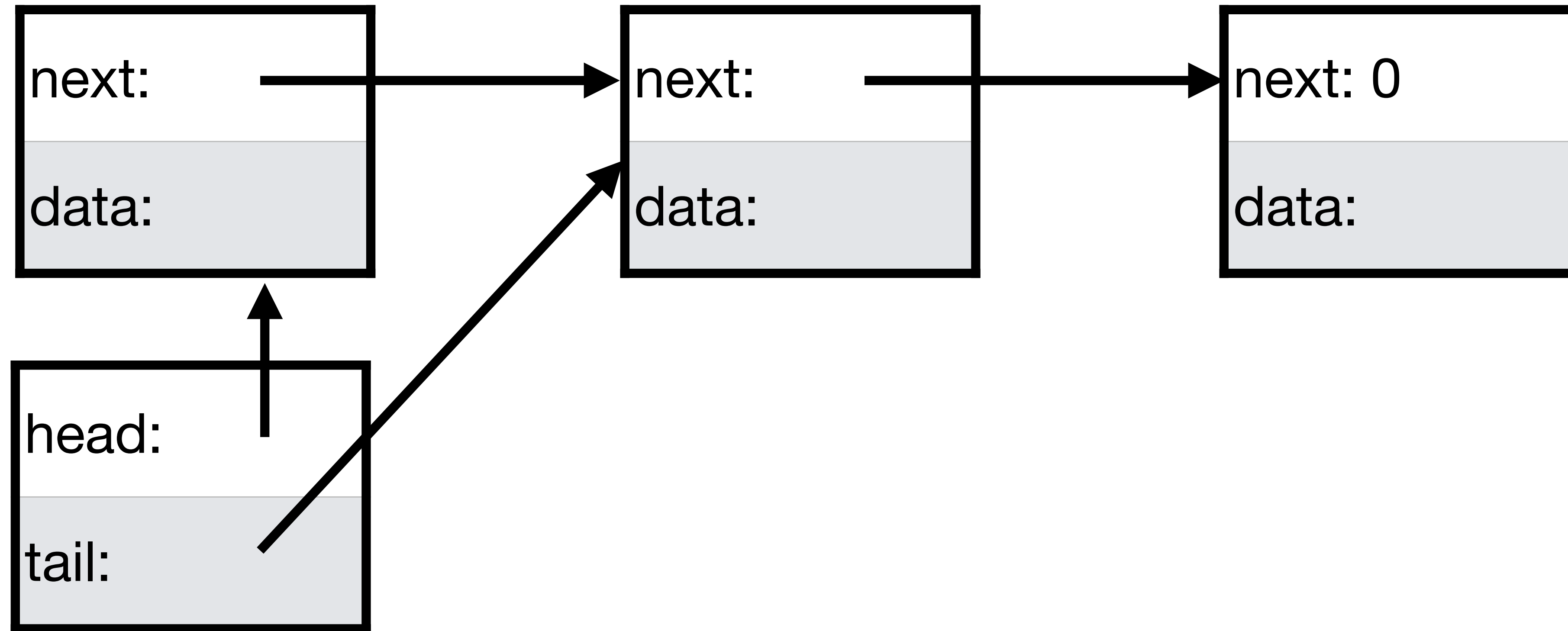


# Appending to the list



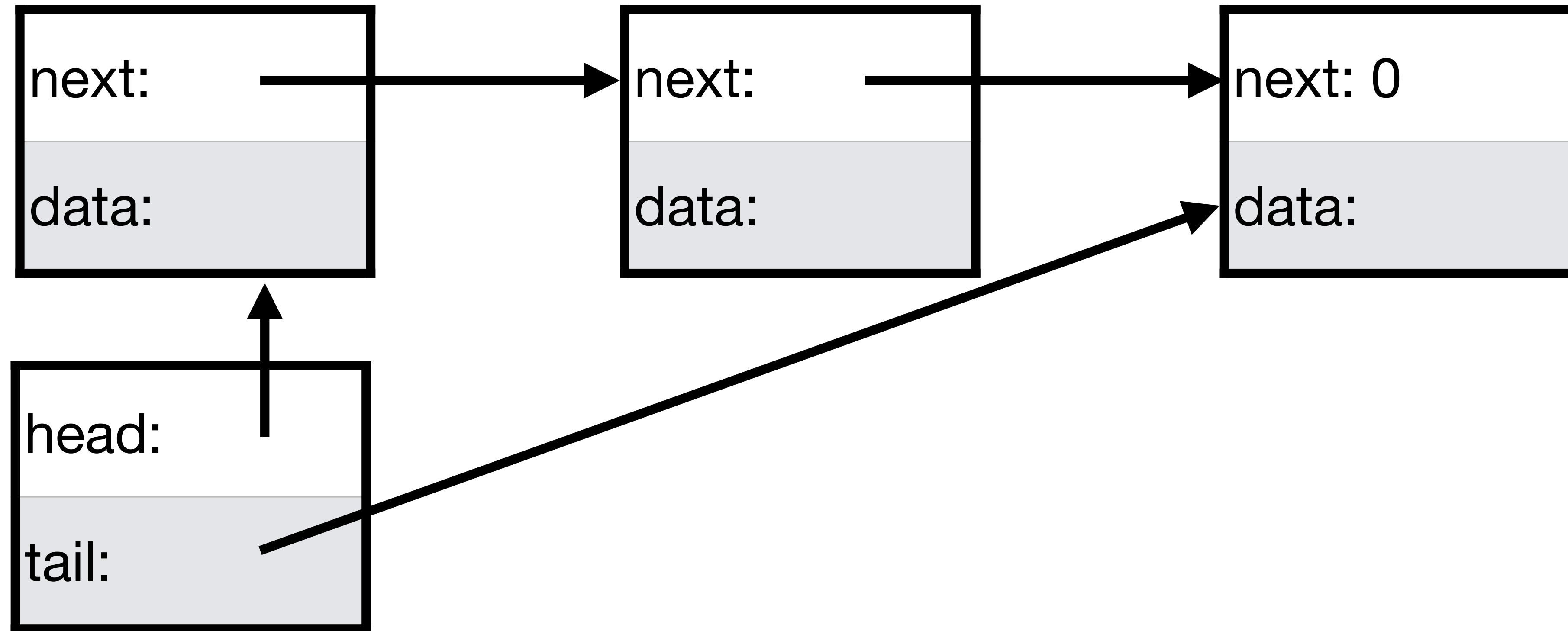
1. Create a new node with  $\text{next} = 0$  and data set to the new element

# Appending to the list



1. Create a new node with next = 0 and data set to the new element
2. Update tail->next to point to the new node

# Appending to the list



1. Create a new node with next = 0 and data set to the new element
2. Update tail->next to point to the new node
3. Update tail to point to the new node

# Appending to the list

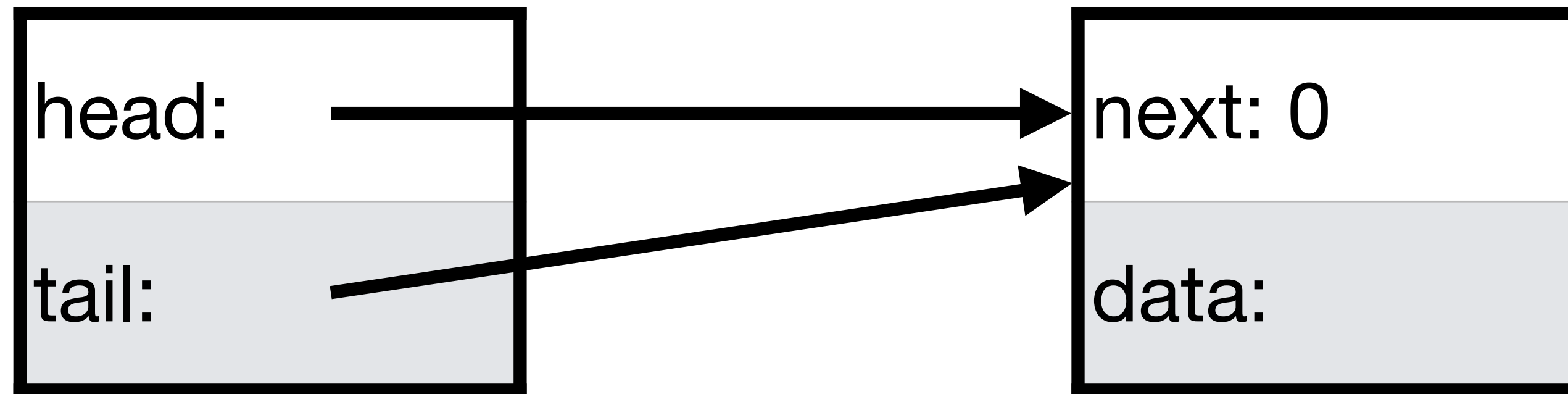
```
void list_append(List *list, int data) {  
    // Create a new node.  
    Node *node = malloc(sizeof *node);  
    node->next = 0;  
    node->data = data;  
    // Update tail->next to point to the new node.  
    list->tail->next = node;  
    // Update tail to point to the new node.  
    list->tail = node;  
}
```

What happens if we append to an empty list using this code?

```
void list_append(List *list, int data) {  
    // Create a new node.  
    Node *node = malloc(sizeof *node);  
    node->next = 0;  
    node->data = data;  
    // Update tail->next to point to the  
    // new node.  
    list->tail->next = node;  
    // Update tail to point to the new node.  
    list->tail = node;  
}
```

- A. head and tail both point to the new node
- B. head points to the new node and tail is 0
- C. tail points to the new node and head is 0
- D. head and tail are both 0
- E. Undefined behavior

# Appending the first element



Set the head and tail pointers to point to the new node



# Appending to the list

```
void list_append(List *list, int data) {
    // Create a new node.
    Node *node = malloc(sizeof *node);
    node->next = 0;
    node->data = data;
    if (list_isempty(list)) {
        // Insert the first element in the list.
        list->head = node;
        list->tail = node;
    } else {
        // Update tail->next to point to the new node.
        list->tail->next = node;
        // Update tail to point to the new node.
        list->tail = node;
    }
}
```

# isempty and size

```
// Returns true if the list is empty.  
bool list_isempty(List const *list) {  
    return list->head == 0;  
}
```

```
// Return the list size.  
size_t list_size(List const *list) {  
    size_t size = 0;  
    for (Node const *node = list->head; node; node = node->next)  
        ++size;  
    return size;  
}
```

What steps should we follow to prepend an element to the beginning of a nonempty linked list

```
void list_prepend(List *list, int data);
```

- A. – Create a new node `n` containing the element
  - Set `n->next` to `list->head`
  - Set `list->head` to `n`
  
- B. – Create a new node `n` containing the element
  - Set `list->head` to `n`
  - Set `n->next` to `list->head`
  
- C. – Create a new node `n` containing the element
  - Set `list->head` to `n`
  - Set `list->tail` to `n`

# In-class exercise

<https://checkoway.net/teaching/cs241/2019-fall/exercises/Lecture-19.html>

Grab a laptop and a partner and try to get as much of that done as you can!