COMP1511 Week 9

Free and Linked List Exercises

Joanna Lin

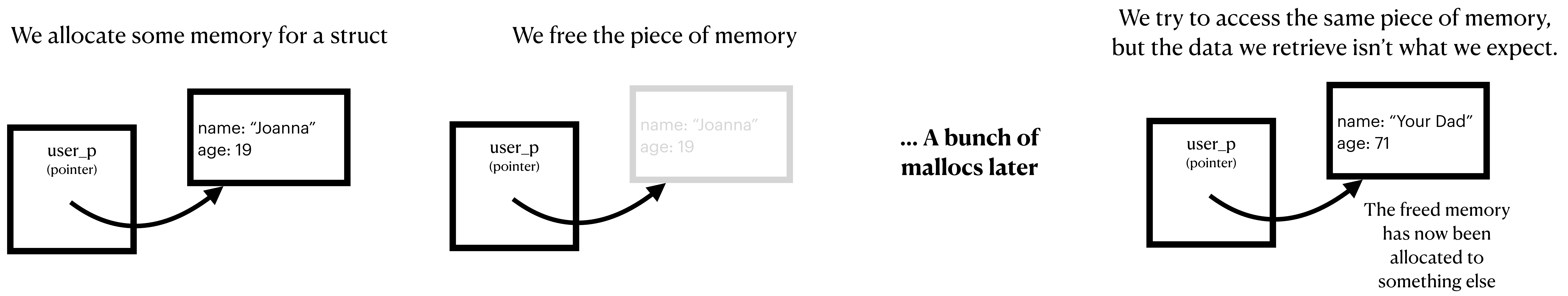
Free

An equal and opposite reaction

- Since the program doesn't manage heap memory for us, we need to do that ourselves.
- When a program terminates, the operating system reclaims all its memory.
- However, if we were to write larger programs that ran for longer periods of time, our memory usage will steadily grow with the amount of times we called **`malloc`**, causing performance issues.
- So, for every **`malloc`**, there must be a **`free`**.
- But… when do we call **`free`**?

Free

Problem



- **Warning:** You must always free, but you must never access freed memory.
  - When we return memory back to the system via `free`, it is now free to allocate that piece of memory to something else.
- An access-after-free error occurs when we try to *dereference* a pointer pointing to a freed address.
  - You can still change the address stored inside the pointer itself (reassign it)
- **Solution:** Only free memory you know you'll never need to use again.

Freeing a List

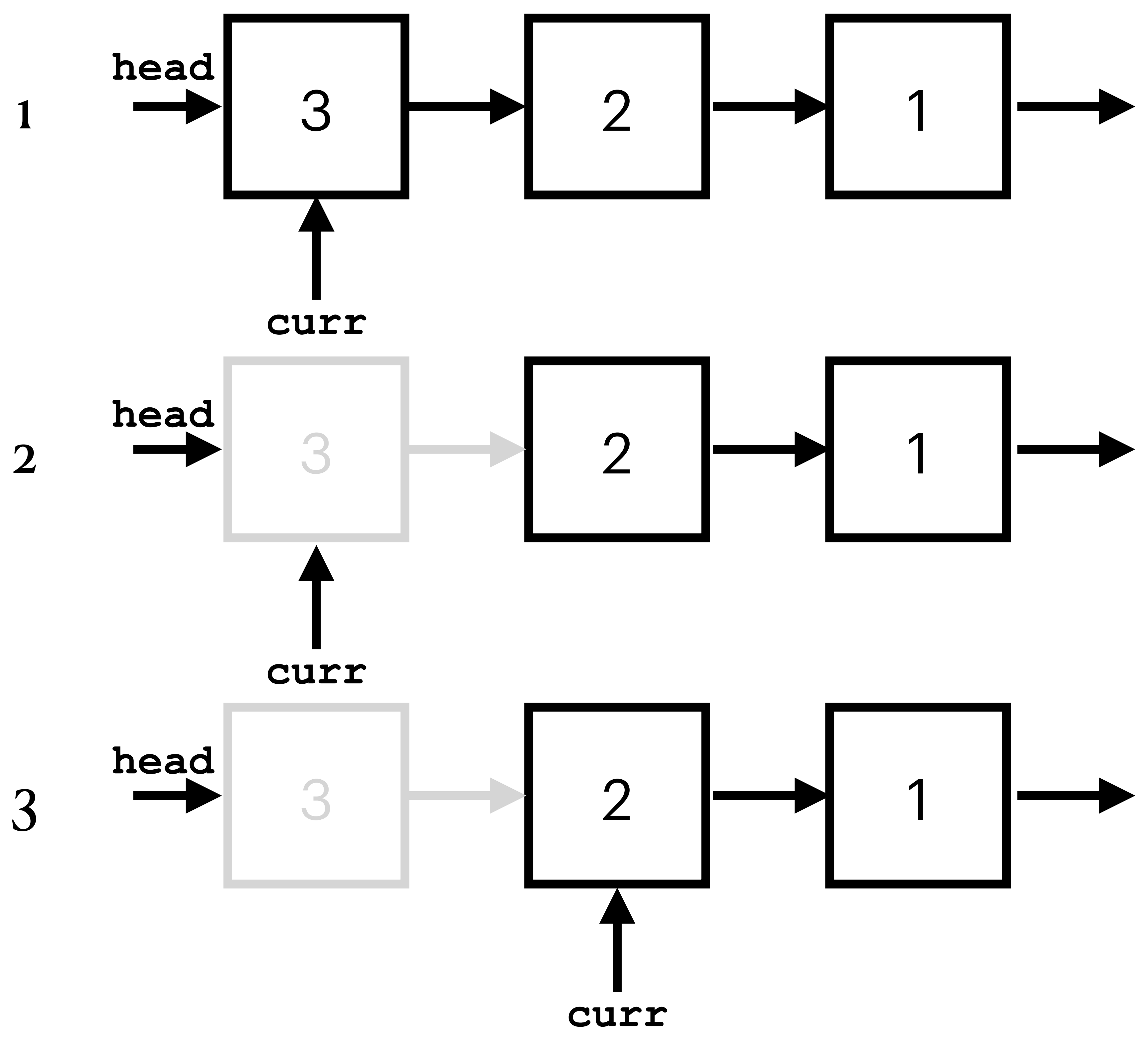
Attempt 1

Consider the following approach

1. We start at the head pointer.
2. We free the node.
3. We move onto the next node.
4. Repeat steps 2 and 3.

```c
struct node *curr = head;
while (curr != NULL) {
    free(curr);
    curr = curr->next;
}
```

- What's wrong with this approach?



Freeing a List

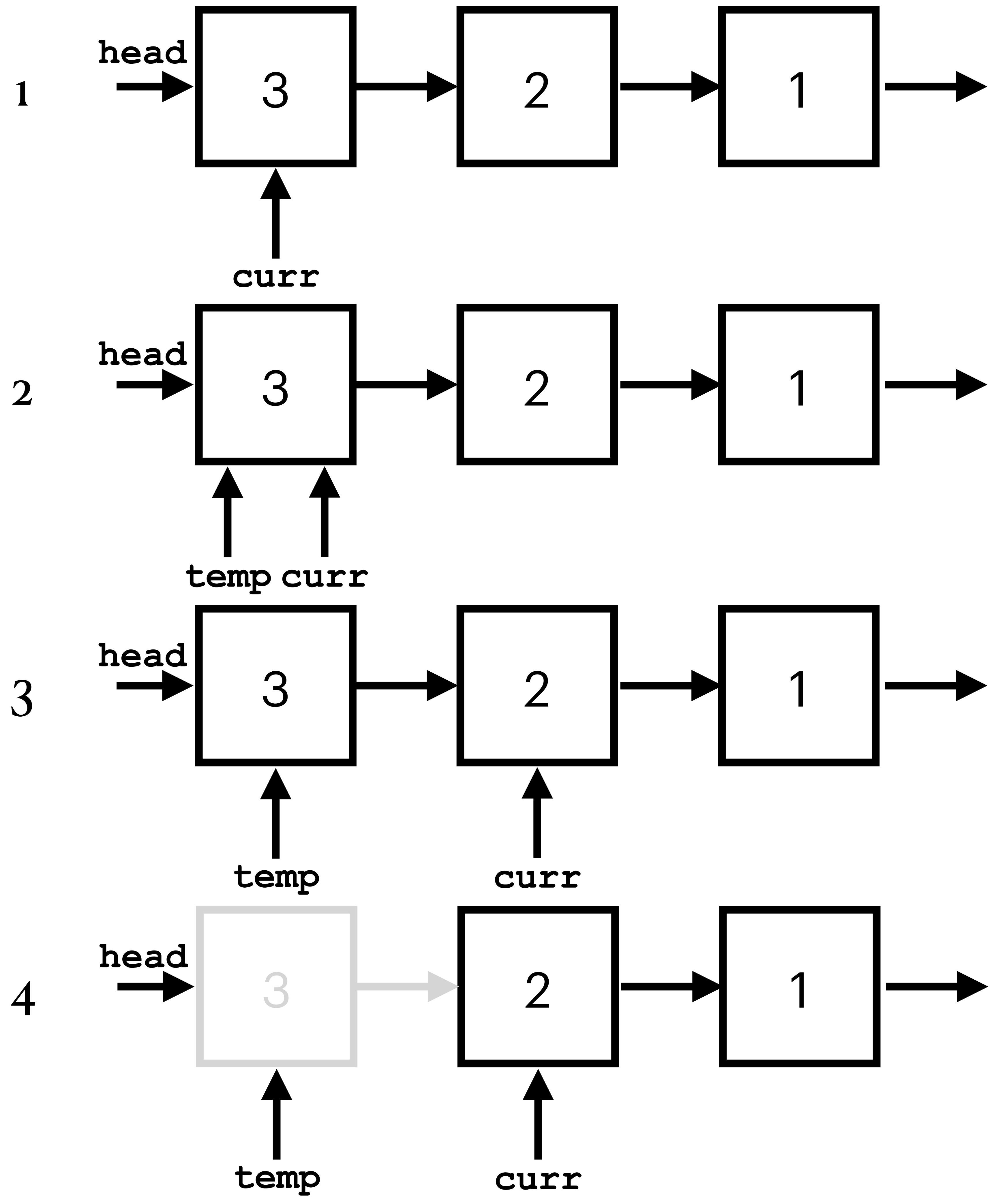
Correct Way

Trick: we must introduce a temporary variable — one that saves the node to be deleted.

Actual steps

1. We start at the head of the list
2. Save the node to be deleted in a temporary variable.
3. Shift the current pointer.
4. Free the node the temporary variable points at.
5. Repeat steps 2 to 4.

```c
struct node *curr = head;
while (curr != NULL) {
    struct node *temp = curr;
    curr = curr->next;
    free(temp);
}
```



Pop Quiz

What's wrong with this `create_node` function?

```
struct node {
    int data;
    struct node *next;
}

struct node *create_node(int data) {
    struct node *new_node = malloc(sizeof(struct node *));
    new_node->data = data;
    new_node->next = NULL
    return new_node;
}
```

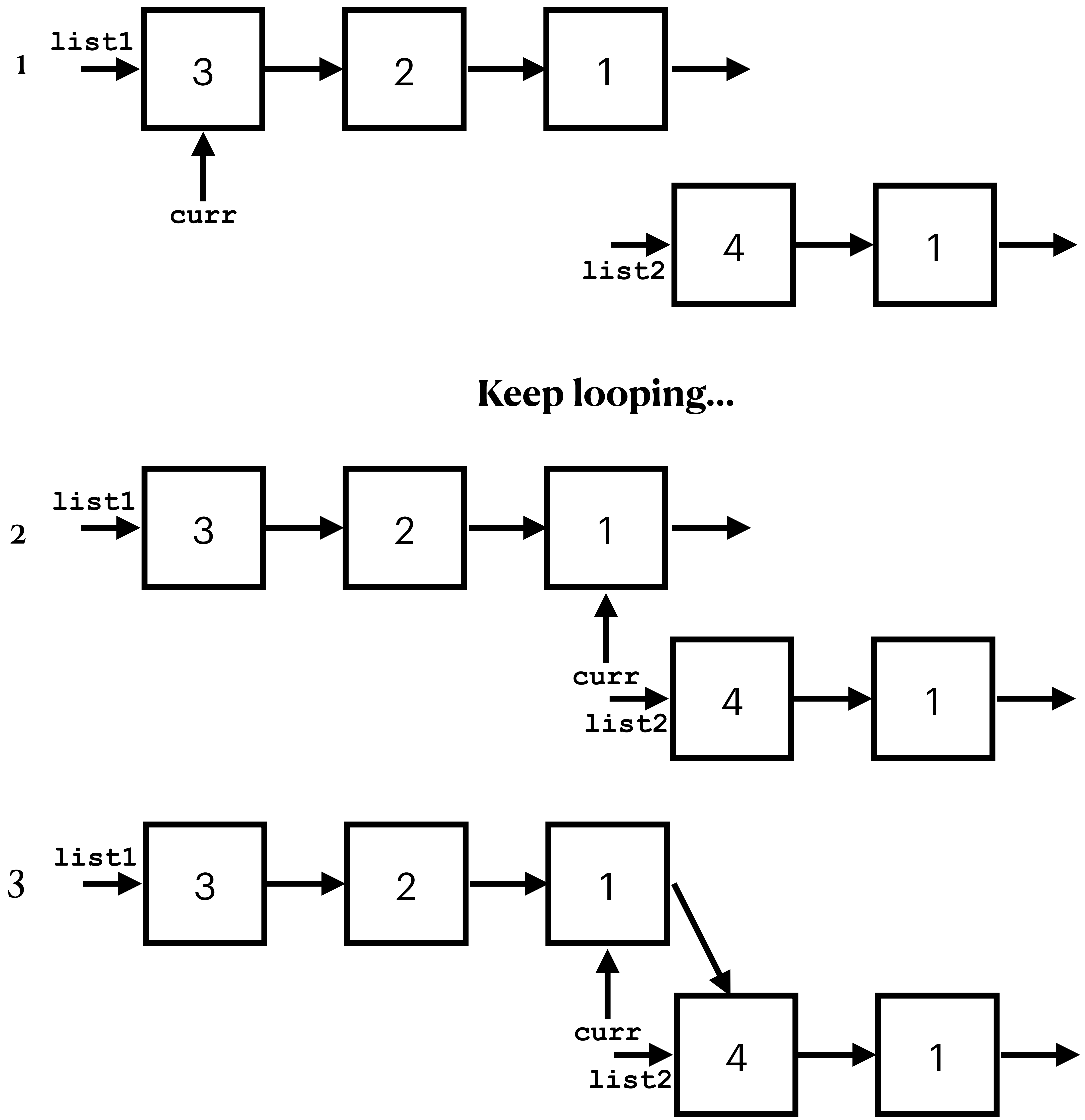
We want to allocate enough memory to store a `struct node`, not a `struct node` pointer!

To prevent uninitialised variable access, we should set the `next` pointer to `NULL`

Linked List Problems

Appending a list to another

- We want to append one list to another, and return the head of the new list.
- General logic
    1. We start our loop variable from the head of the first list.
    2. Loop until loop variable points at last node of first list.
    3. Make last node of first list point to start of second list.
    4. Return the head of the first list.
- Edge cases
    - When the first list is empty, return the second.



Linked List Problems

Copying a List: A Basic Approach

- We want to clone a list, and return the head of the clone.
- General logic
  1. We create an additional variables: one that stores the head of the clone
  2. We start our loop from the head pointer of our original list.
  3. We clone the current node using the value it stores.
  4. Use our append list function to add the node to the end of the copy. Store this return value into the head of the clone.
  5. Advance the loop variable.
  6. Repeat 3 to 5 until the end of the list.
  7. Return the head of the clone.

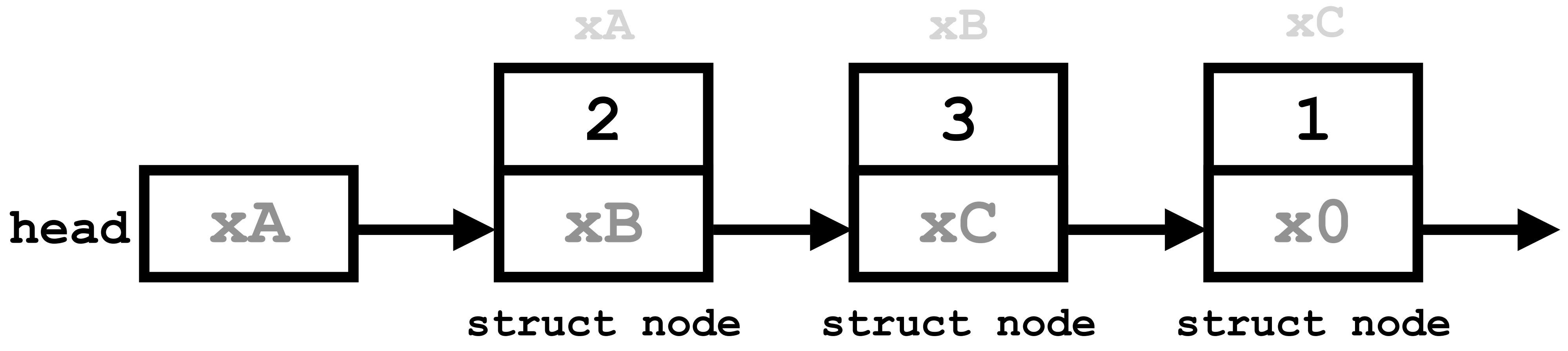
Linked List Problems

Copying a List: An Advanced Approach

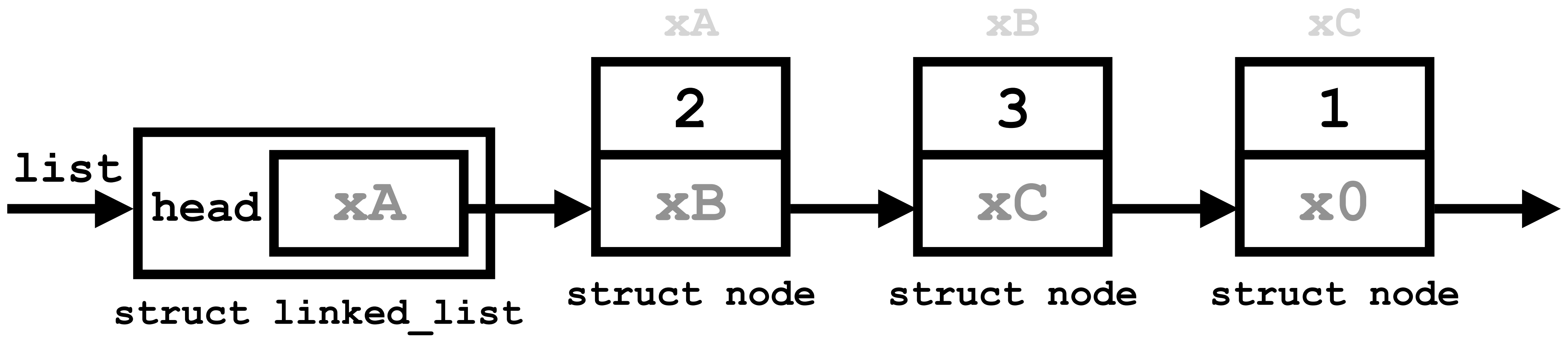
1. We create 2 variables: one that stores the head, another that points to the last node added to the list.
2. We start our loop from the head pointer.
3. We read node value and create a clone out of it.
4. a) If the copy's head is null, make the head point to the new node.
   b) Otherwise, make the last node that was added to the list point to the current node.
5. Shift the list copy's current pointer to the newly added node.
6. Advance the current pointer and repeat steps 3 to 5 to the end of the list.
7. Return the head of the clone.

Wrapper Around Head

- So far we've only dealt with functions in which we pass in the head of a list directly.

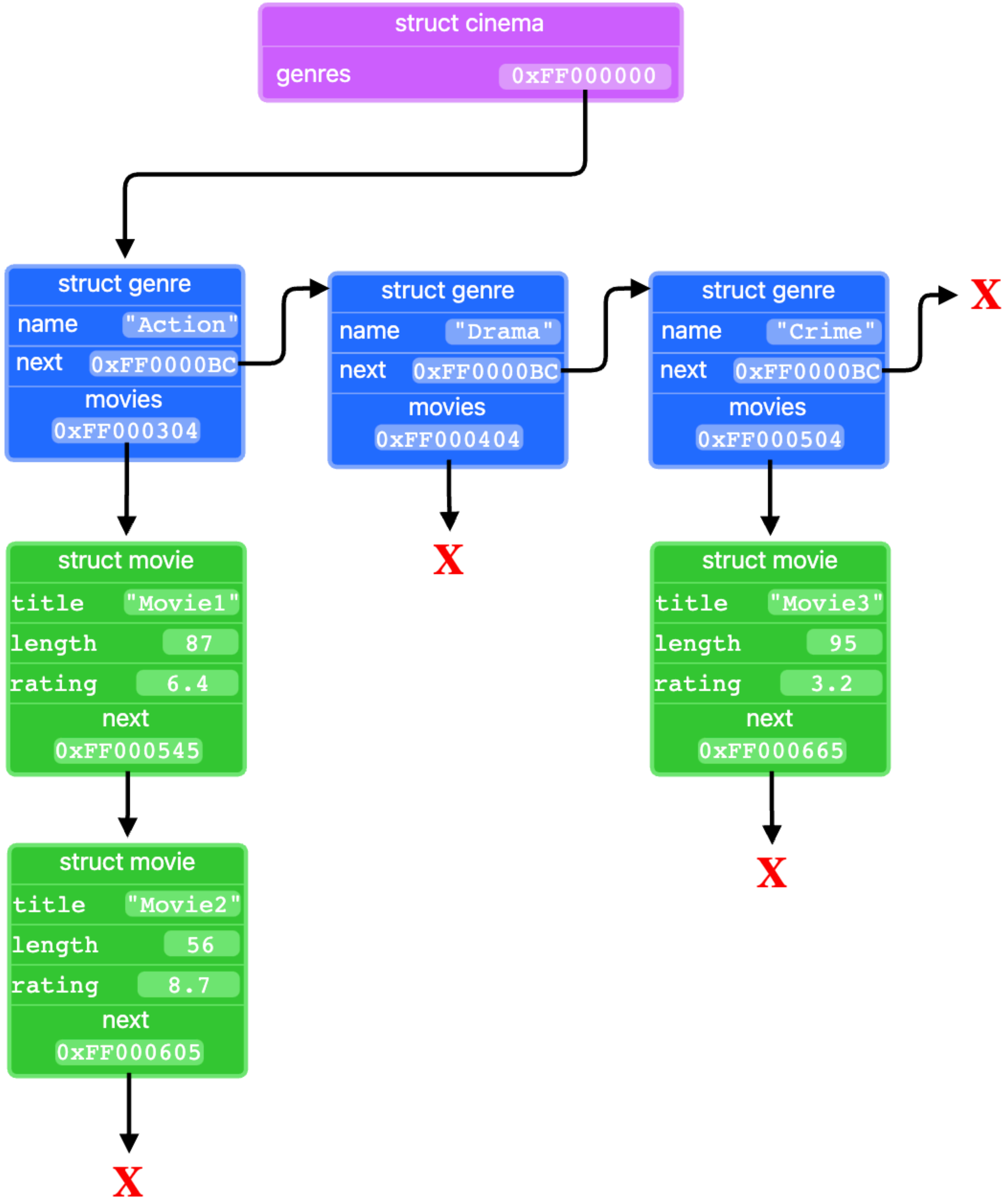


- In the assignment, functions are given a 'wrapper' around the head of the list. To access the head, we need to dereference the struct pointer.



- What are the advantages to this approach?

2D Linked List



- We have one 'wrapper' pointing to the head of a list — **`struct cinema`**
- Each **`struct genre`** node in this list
  - points to the next **`struct genre`** node in the list
  - is a 'wrapper' pointing to the head of a sub-list of **`struct movie`** nodes
- There are two types of lists here: lists of **`struct movie`** nodes and a list of **`struct genre`** nodes.