COMP1511 Week 9

Free and Linked List Exercises

Joanna Lin

Free

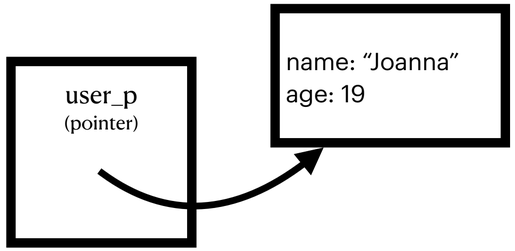
An equal and opposite reaction

- 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`**?

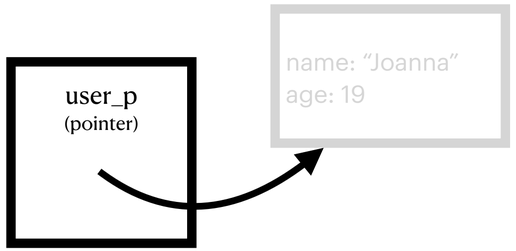
Free

Problem

We allocate some memory for a struct

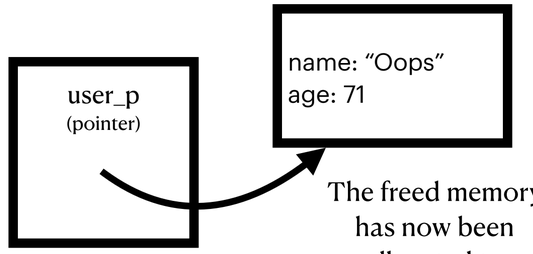


We free the piece of memory



... A bunch of mallocs later

We try to access the same piece of memory, but the data we retrieve isn’t what we expect.



The freed memory has now been allocated to something else

- **Issue:** 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

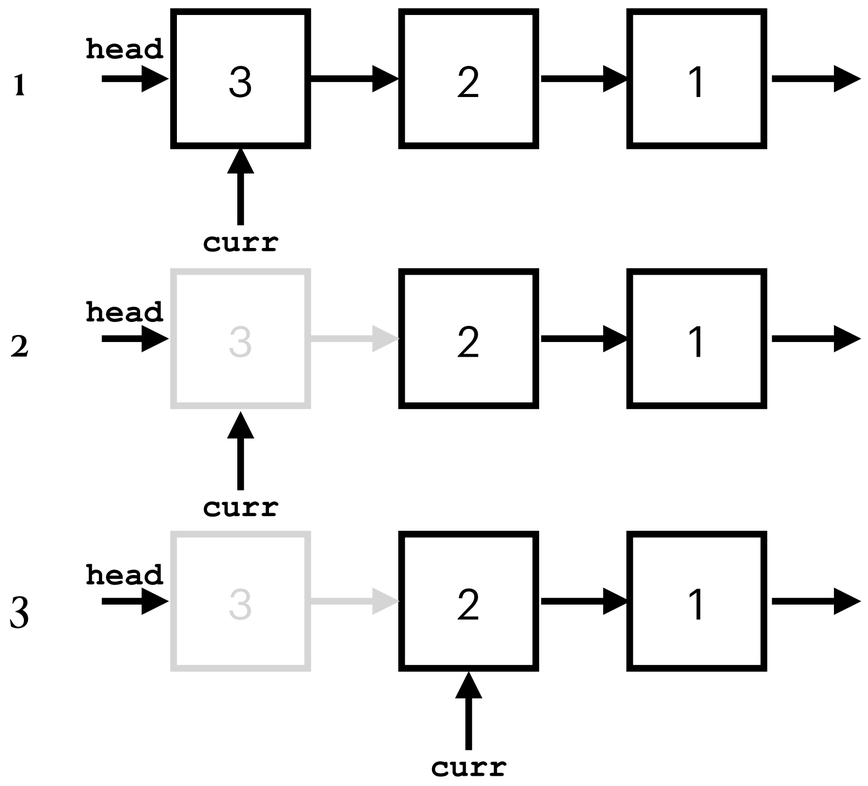
Attempt 1

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

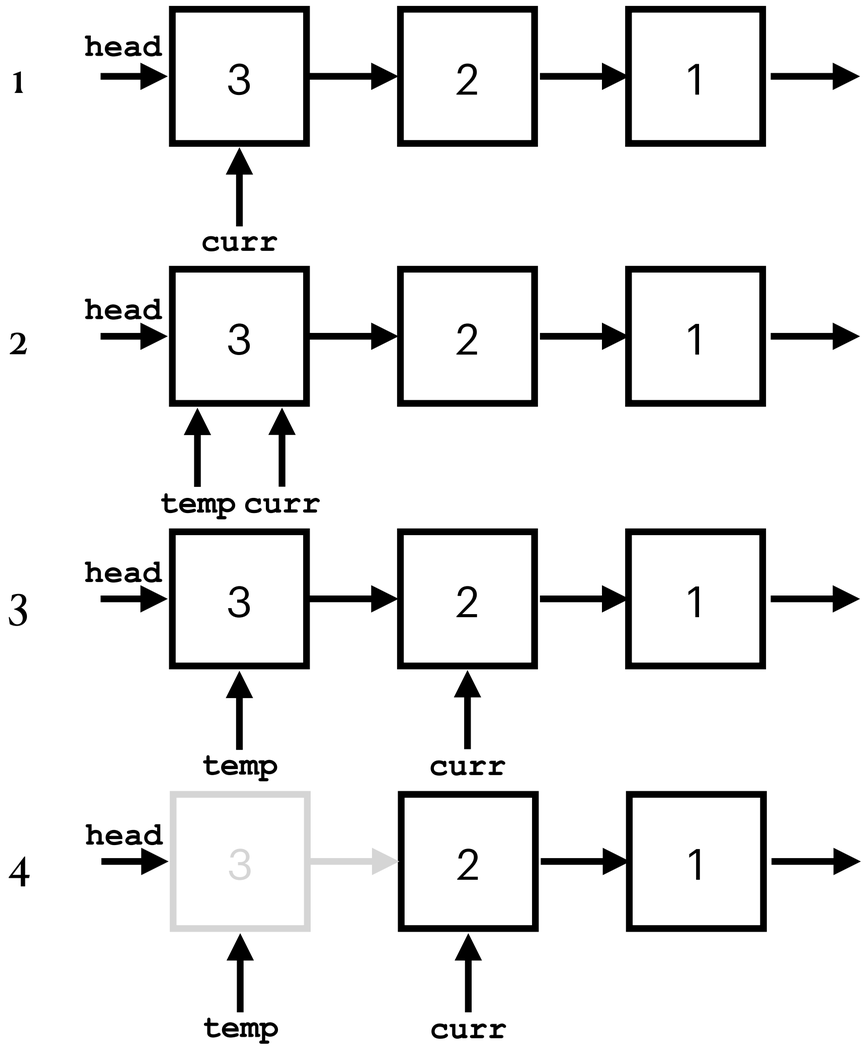
Correct Way

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.

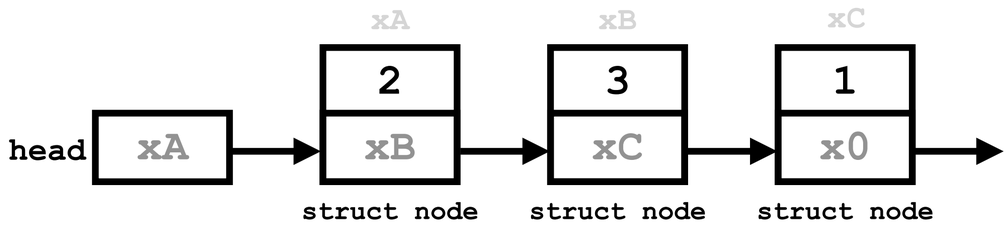
```c
struct node *curr = head;
while (curr != NULL) {
    struct node *temp = curr;
    curr = curr->next;
    free(temp);
}
```



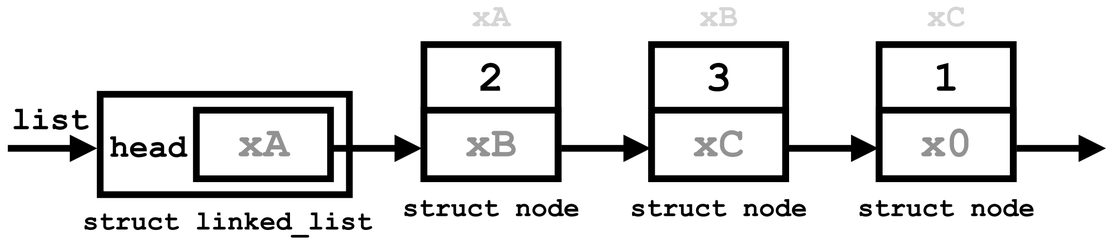
Linked List Problem Solving

Wrapper Around Head

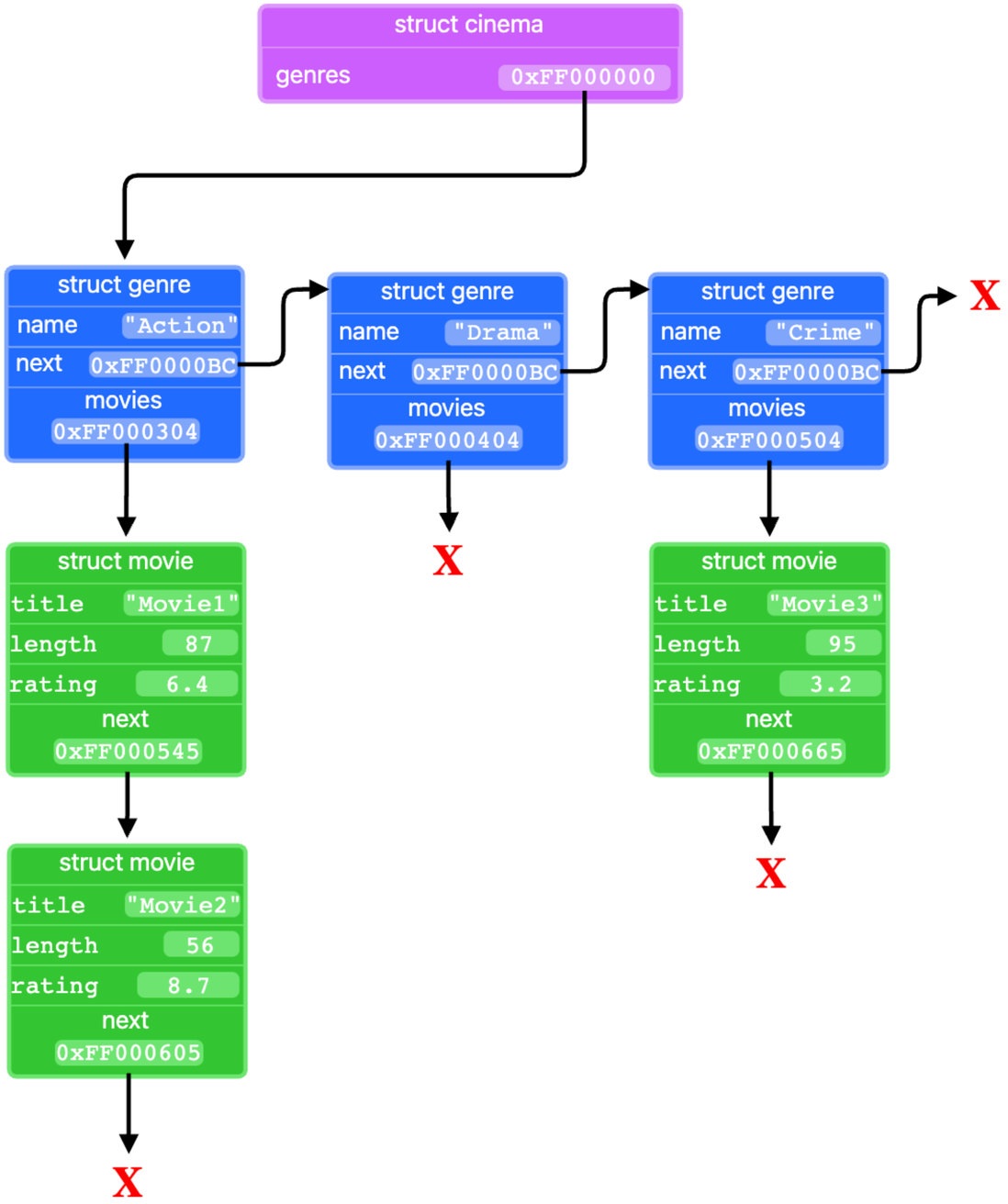
- So far we've only dealt with functions in which we pass in the head of a list directly.



- 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.