COMP1511/1911 Programming Fundamentals

Week 8 Lecture 1

Linked Lists
Recap and Deletion

Week 8 Check off: Mini Exam

- Full details in forum post
- **In person attendance**
  - If you are in an online class OR if you cannot make your timetabled in-person class
    - sign up for an in-person lab time and attend in the second hour
    - attend your regular tlb as well if you are an online student
  - Email course account if for some reason you are sick or can't attend on the day cs1511@unsw.edu.au

Week 8 Check off: Mini Exam

- **Format: 1 hour**
  - A one dot array question
  - A two dot array question
  - Two debugging questions
- Attempt and submit at least one questions to get **check off**
- There are also regular lab exercises to do for the lab

This is a great opportunity to try out the exam environment and attempt questions that will be the same style as the array hurdles and debugging questions in the final exam!

Revision Sessions This Week

Wednesday 22/07/2026 4PM-6PM in **Kora Lab J17-307.**

Sign up and get more details on forum.

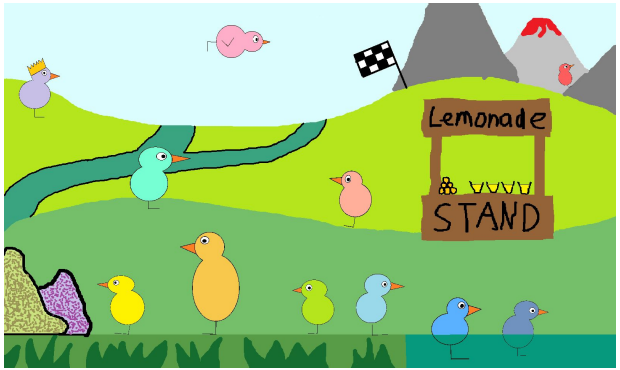
Assignment 2: CS Duck Life 🦢

Assignment Due Date: 7th August 6:00 PM (Friday Week 10)

We have an Assignment 2 Video 📽️ to help you get started.

We have Help Sessions and labs to help you too!

If you would like to organise a **drop in** session to extra 1-1 help with a tutor
please email **cs1511@unsw.edu.au**



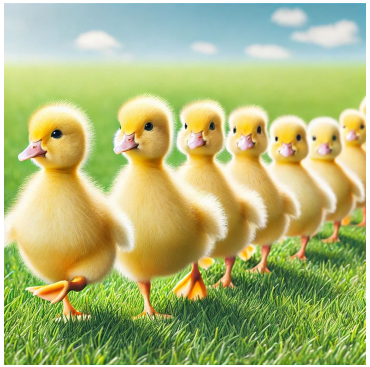
Last Lecture with Sofia

- Linked List Basics
- Creating nodes
- Printing a List
- Inserting nodes
  - At beginning
  - At tail
  - Insert after position n

Today's Lecture

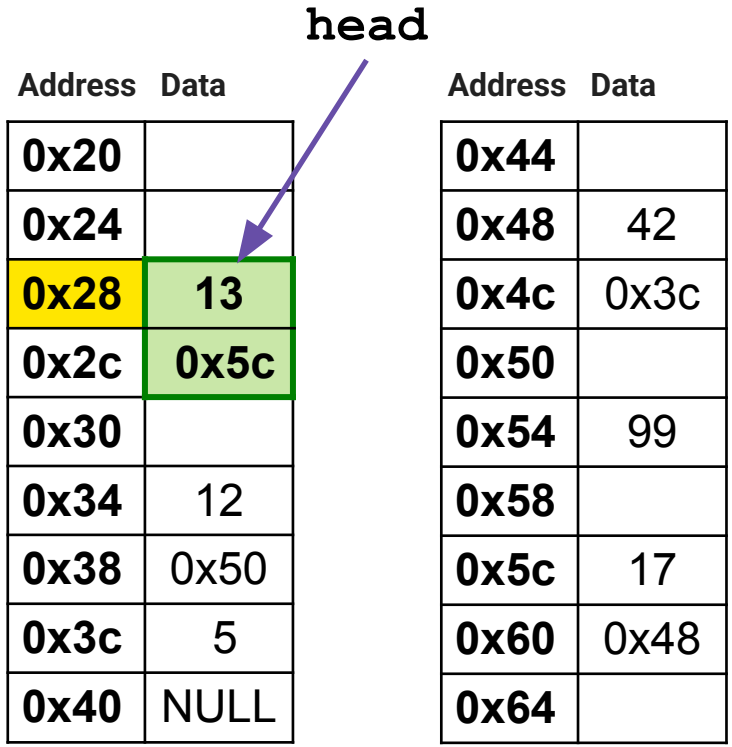
- https://cgi.cse.unsw.edu.au/~cs1511/26T2/code/week_8/
- Recap:
  - List basics, creating nodes, printing lists
  - Insert at head, Insert at tail, Insert after nth position
- Deletion:
  - Deleting the first node
  - Freeing the whole list
  - Search and delete
  - Memory leaks, use after free errors

Linked List Recap



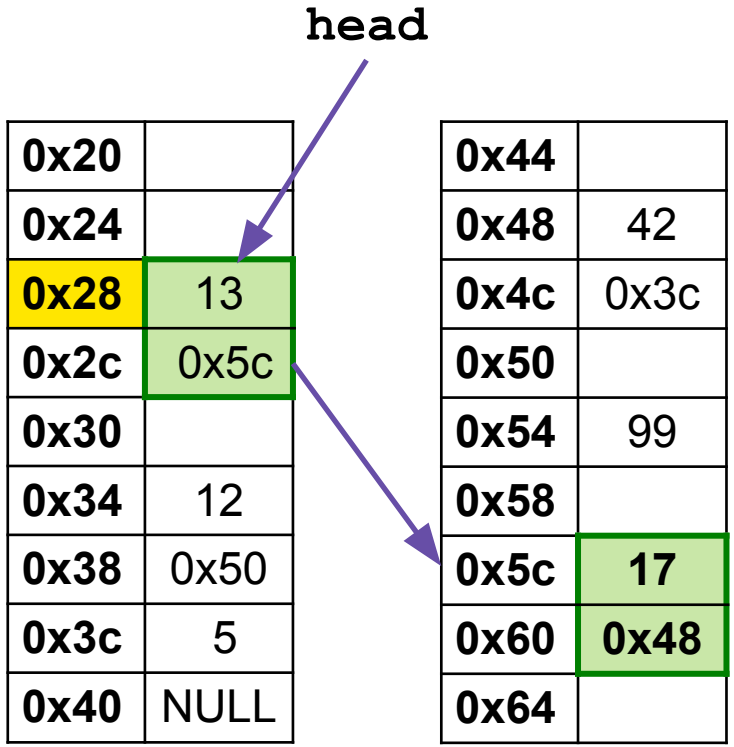
Linked Lists in Memory

- We have a pointer to the first piece of data in the list
- For every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.



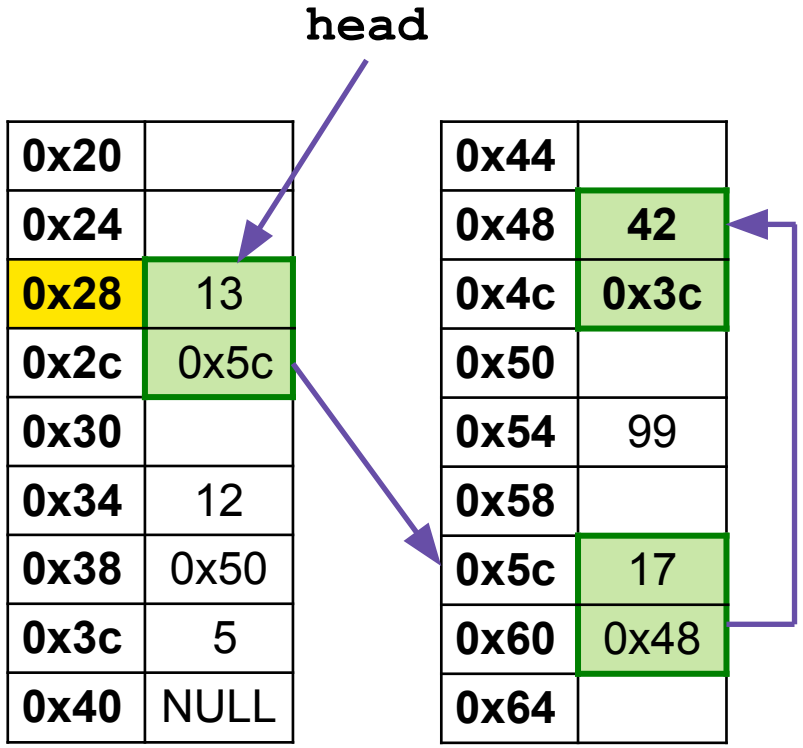
Linked Lists in Memory

.
- We have a pointer to the first piece of data in the list
- For every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.



Linked Lists in Memory

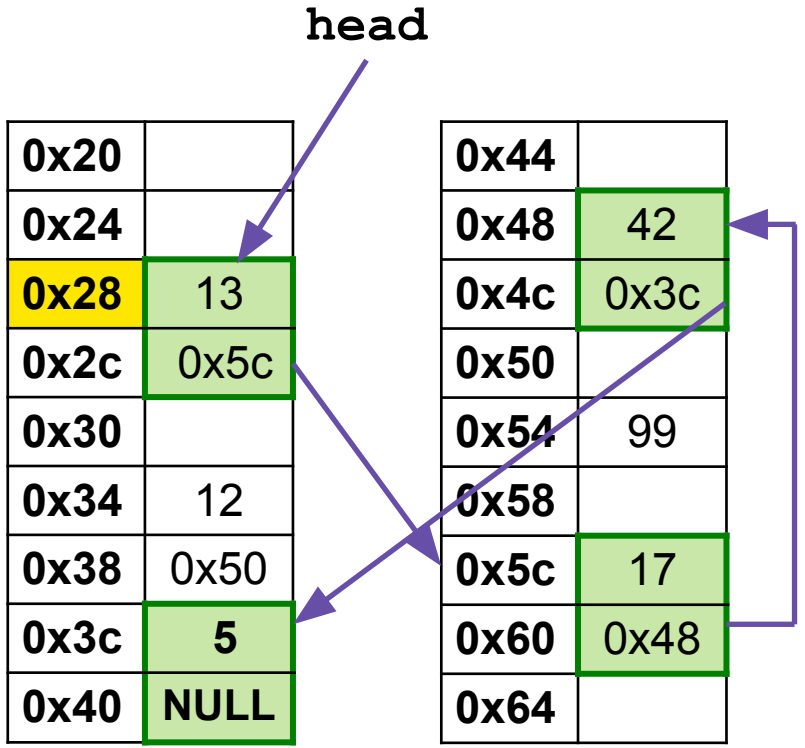
- We have a pointer to the first piece of data in the list
- For every piece of data you store in the list you need to store a link (pointer containing the address) to the next item in the list.
- Like a scavenger hunt.



Linked Lists in Memory

- When the value of the pointer to the next piece of data is NULL you have reached the end of the list.

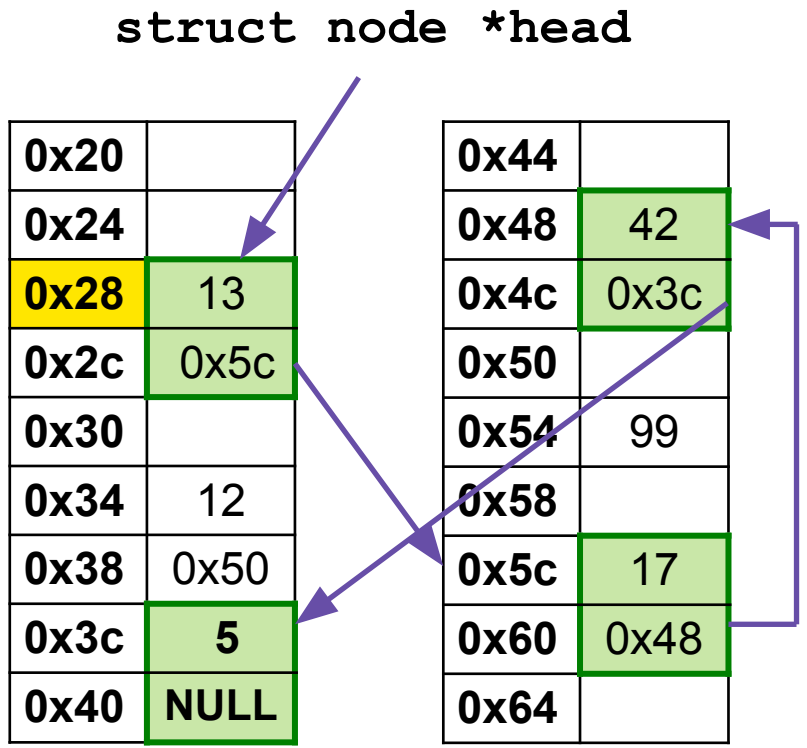
- **Congratulations!**
  You have just traversed a linked list.



Linked Lists Nodes

- We store our data and a pointer together in a struct.
- The list variable is a pointer to the first node in the list

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



Creating a Node

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

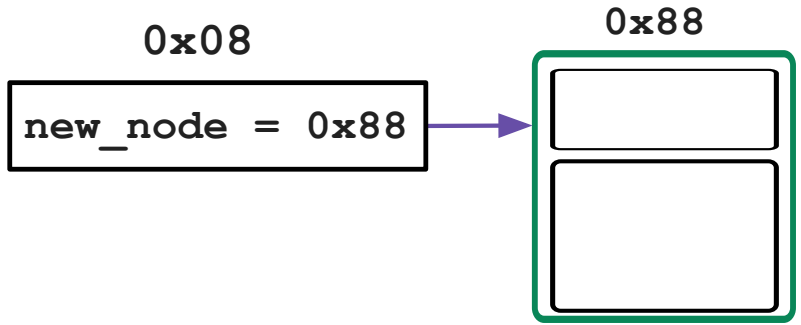
```
struct node *create_node(int num) {



}
```

Creating a Node

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



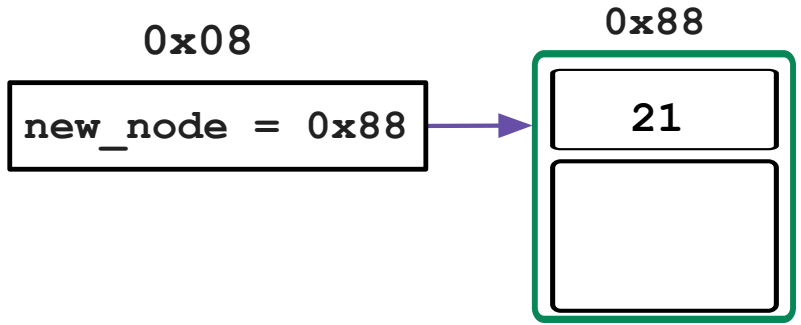

```
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));



}
```

Creating a Node

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



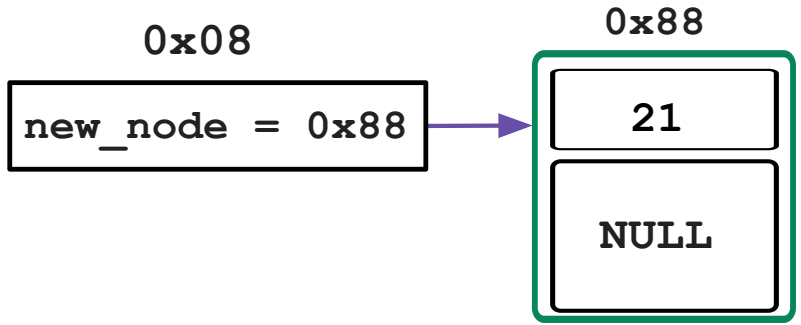

```
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = num;



}
```

Creating a Node

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

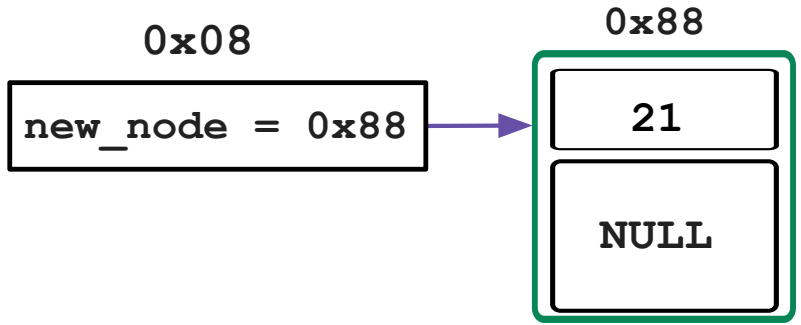



```
struct node *create_node(int num) {
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = NULL;

}
```

Creating a Node

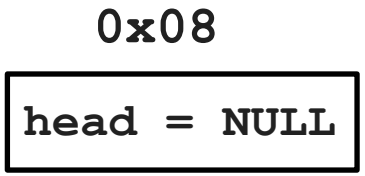
```
struct node {
    int data;
    struct node *next;
};
```




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

Creating a List with 1 Node in C

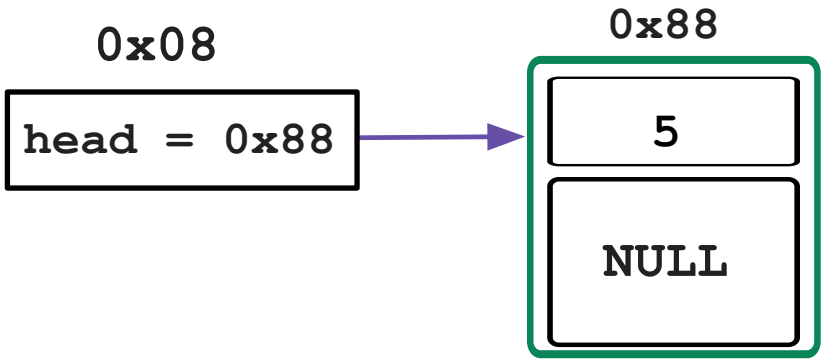
```
struct node {
    int data;
    struct node *next;
};
```




```
struct node *head = NULL;
```

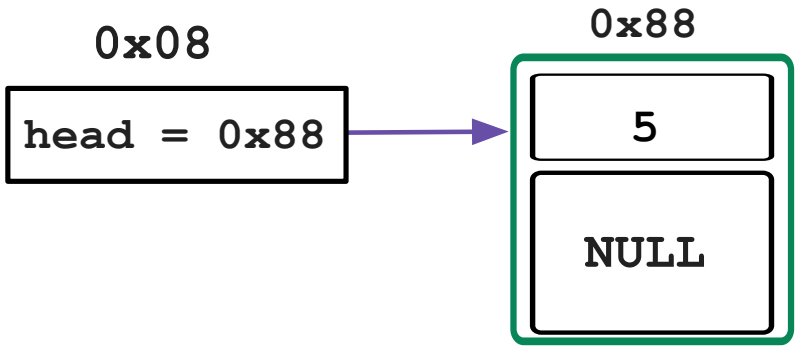
Creating a List with 1 Node in C

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

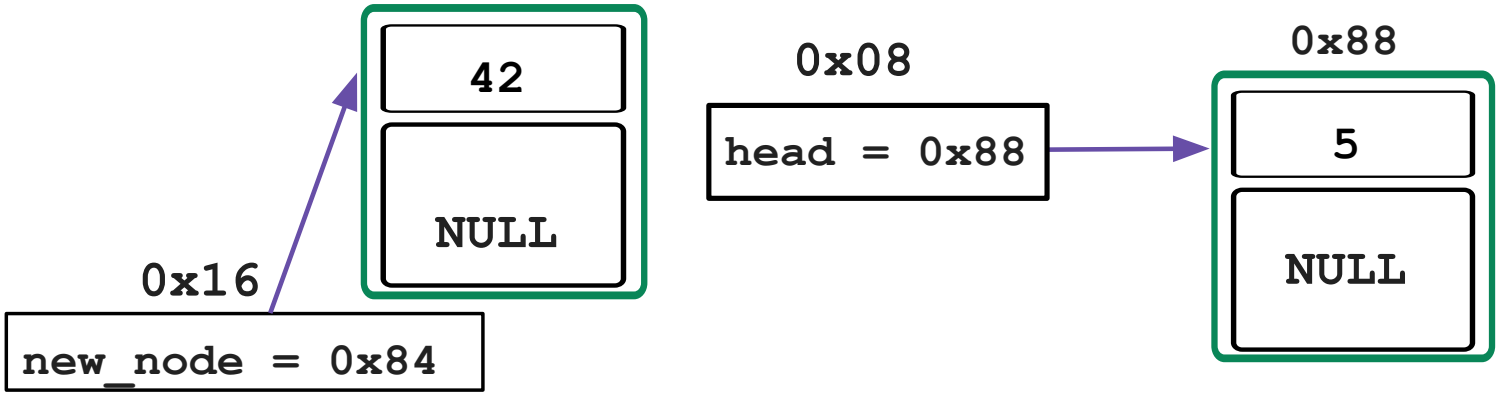



```
struct node *head = NULL;
head = create_node(5);
```

Adding a node to the front of the list

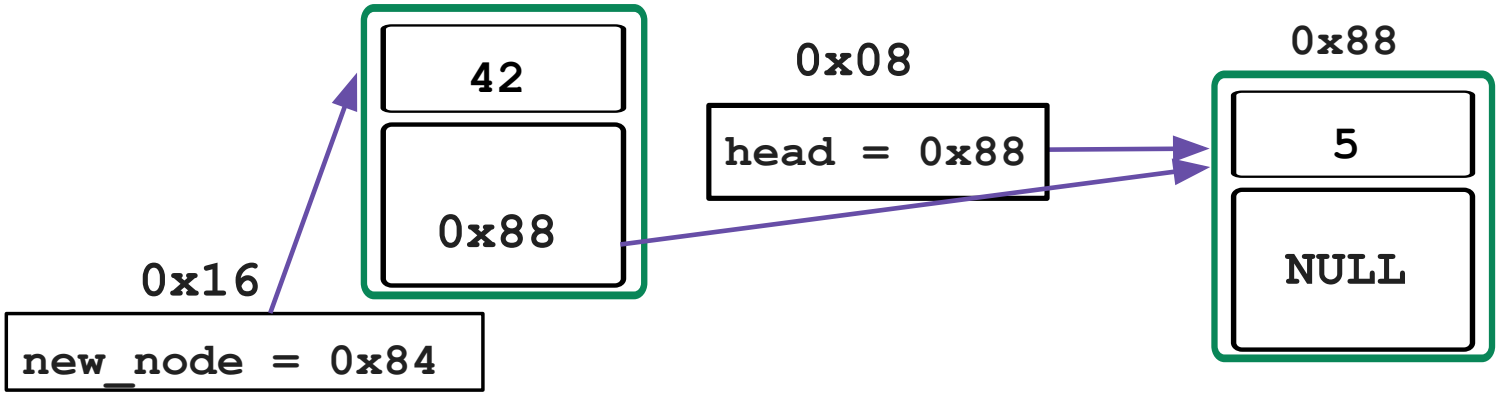


Adding a node to the front of the list



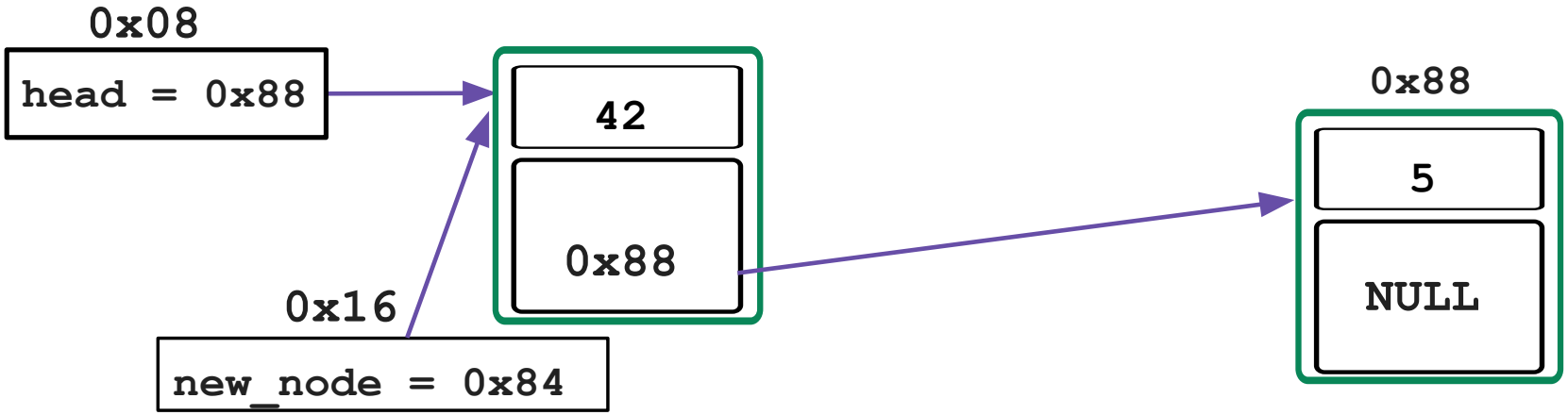

```c
struct node *new_node = create_node(42);
```

Adding a node to the front of the list



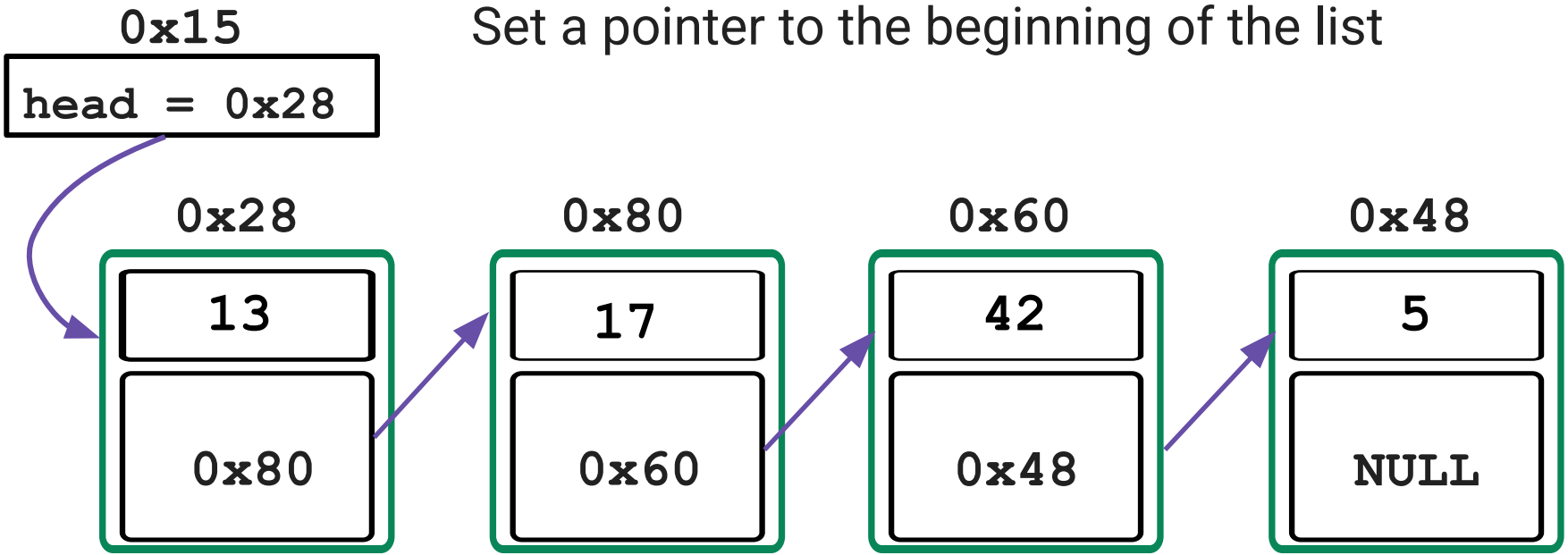

```
struct node *new_node = create_node(42);
new_node->next = head;
```

Adding a node to the front of the list

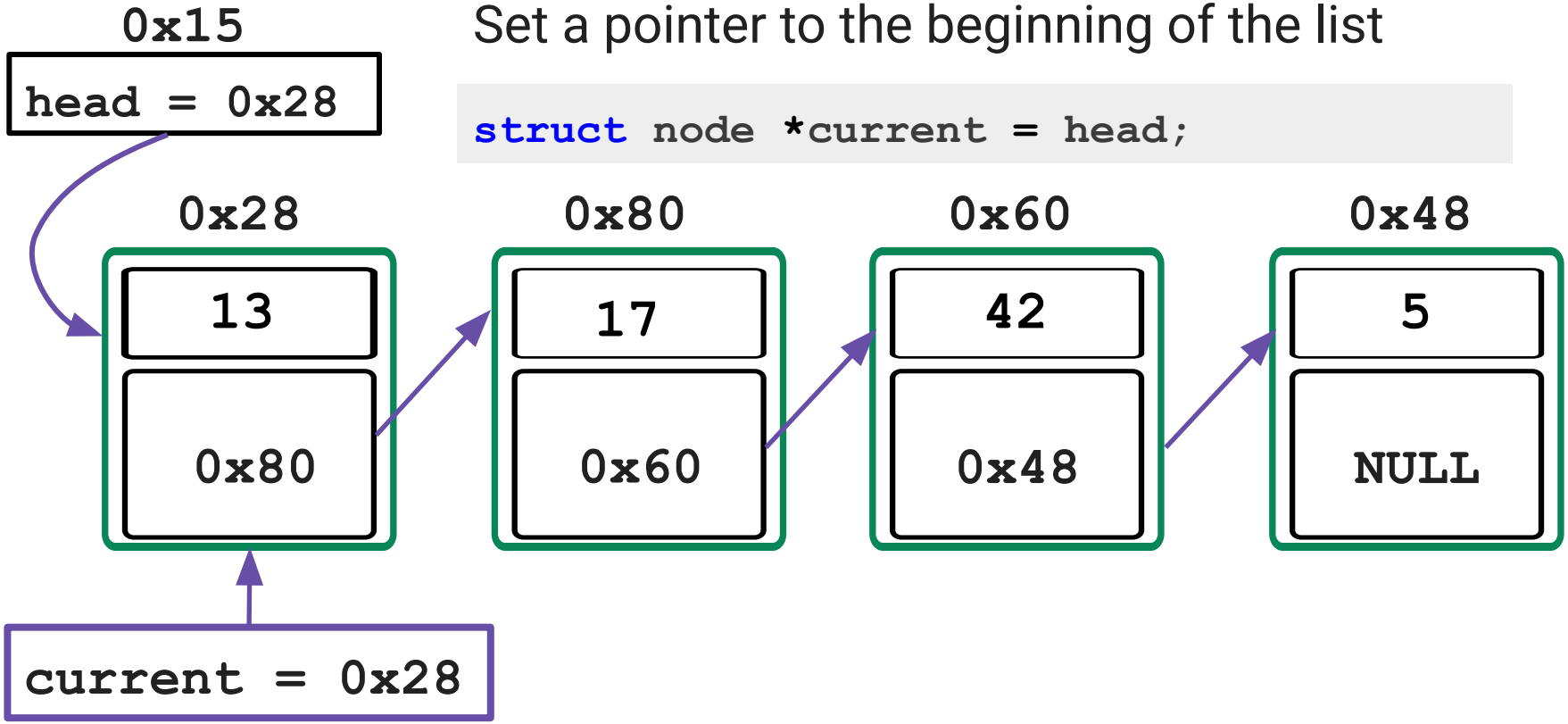



```
struct node *new_node = create_node(42);
new_node->next = head;
head = new_node;
```

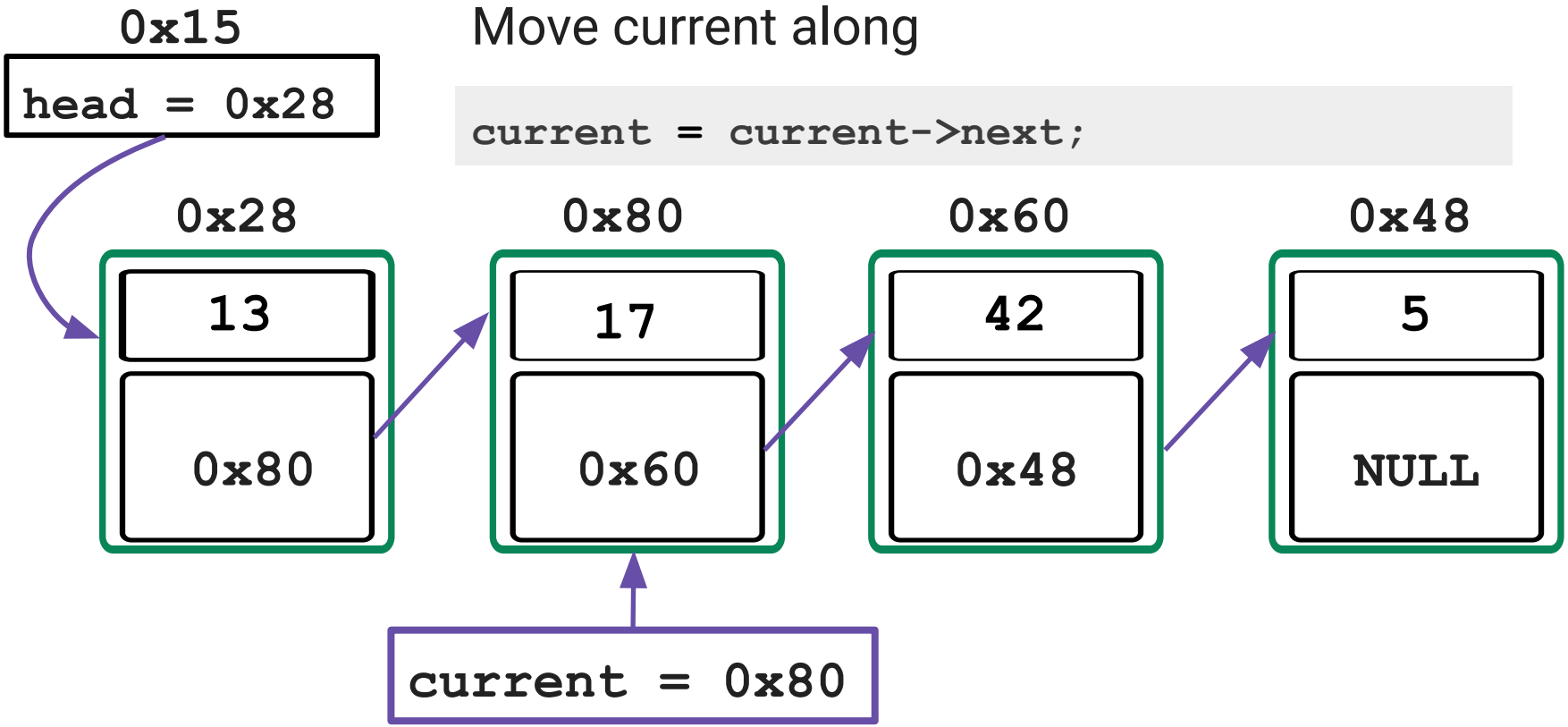
Traversing a List



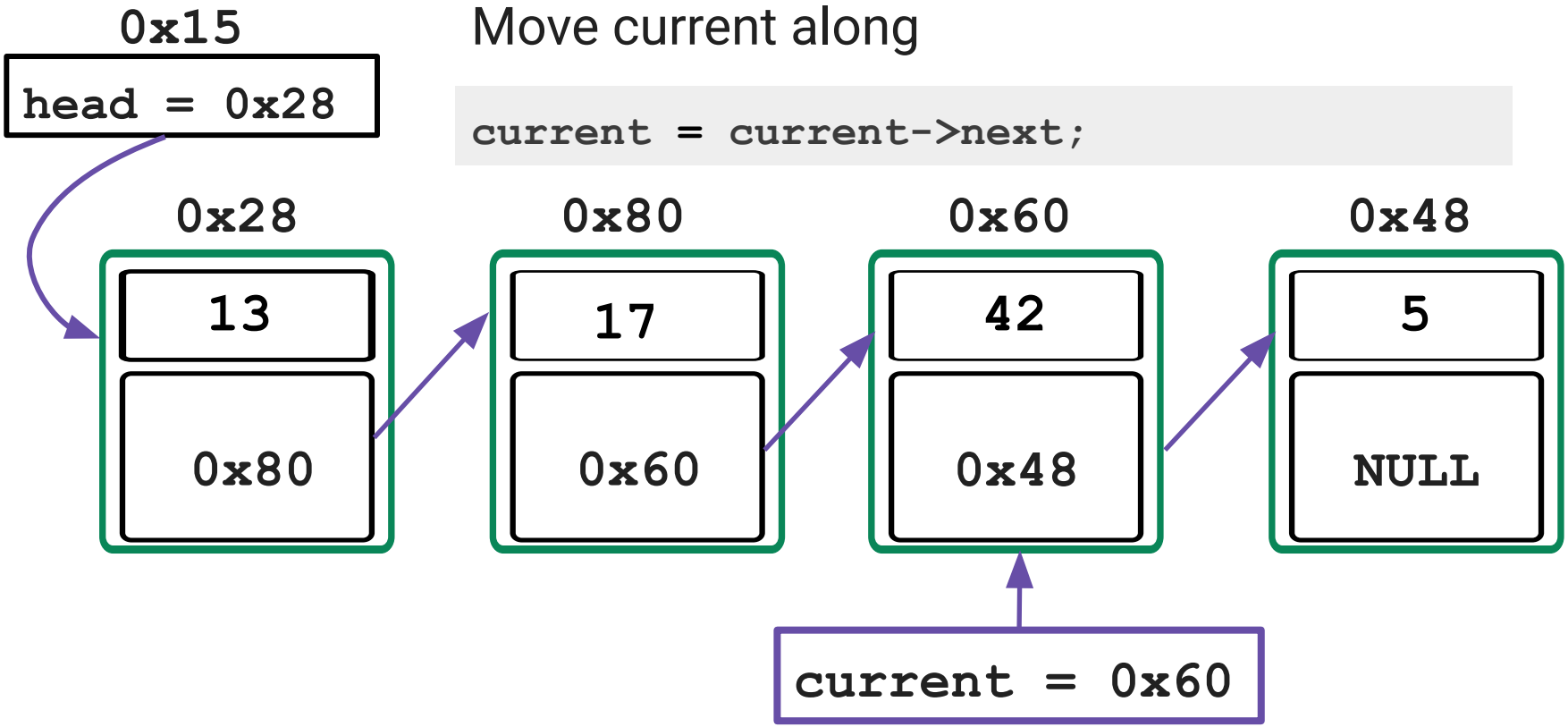
Traversing a List



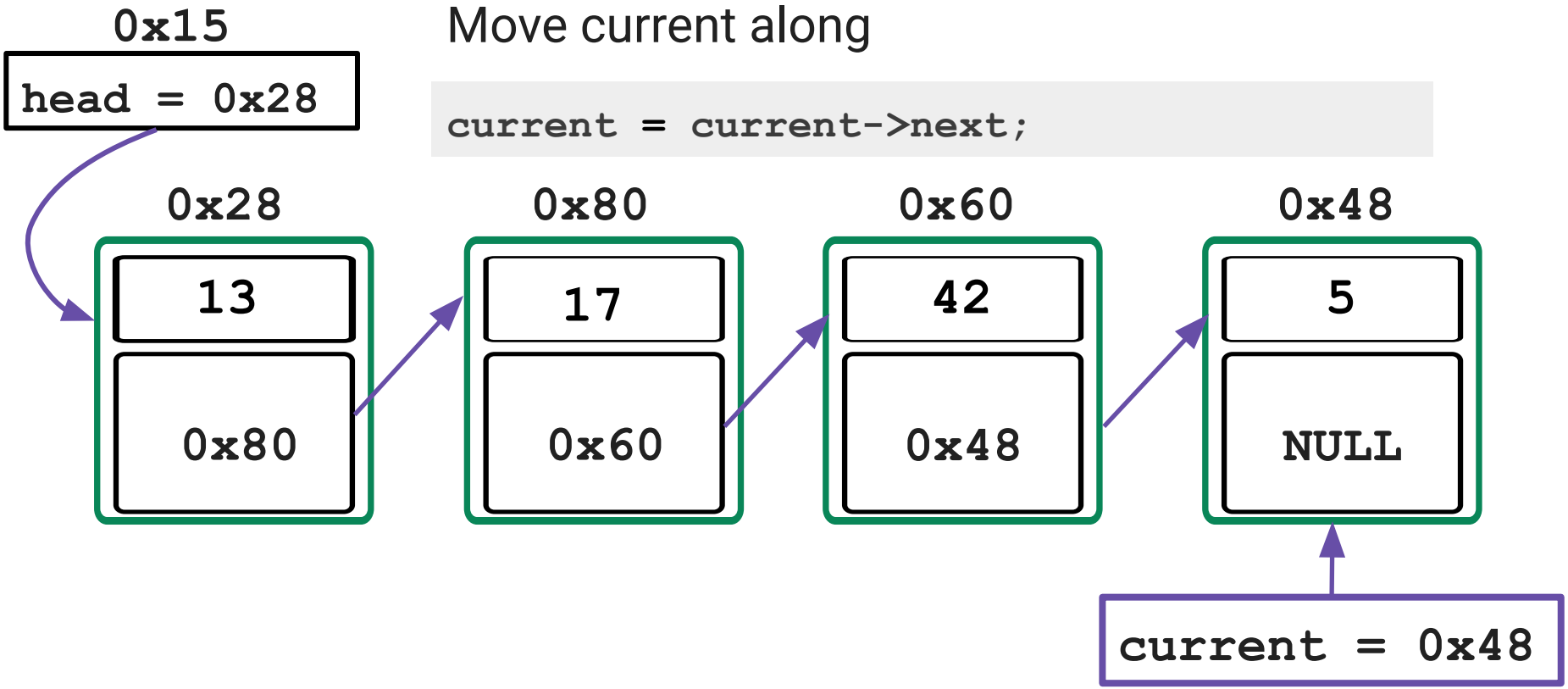
Traversing a List



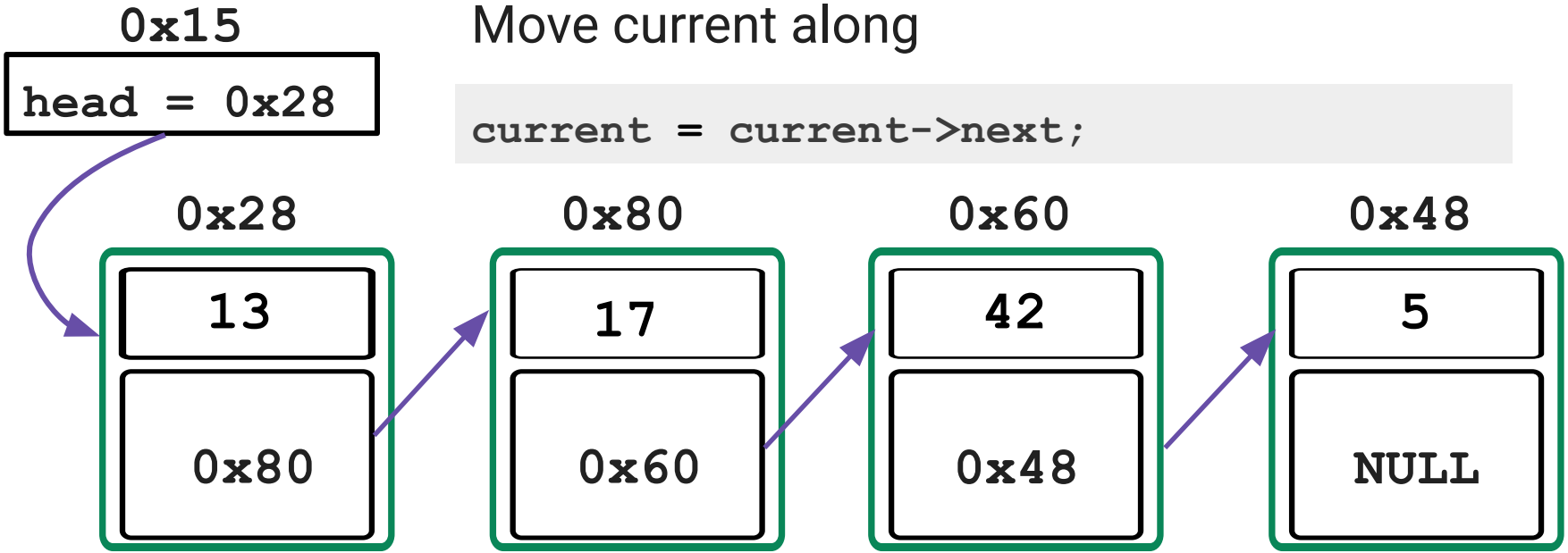
Traversing a List



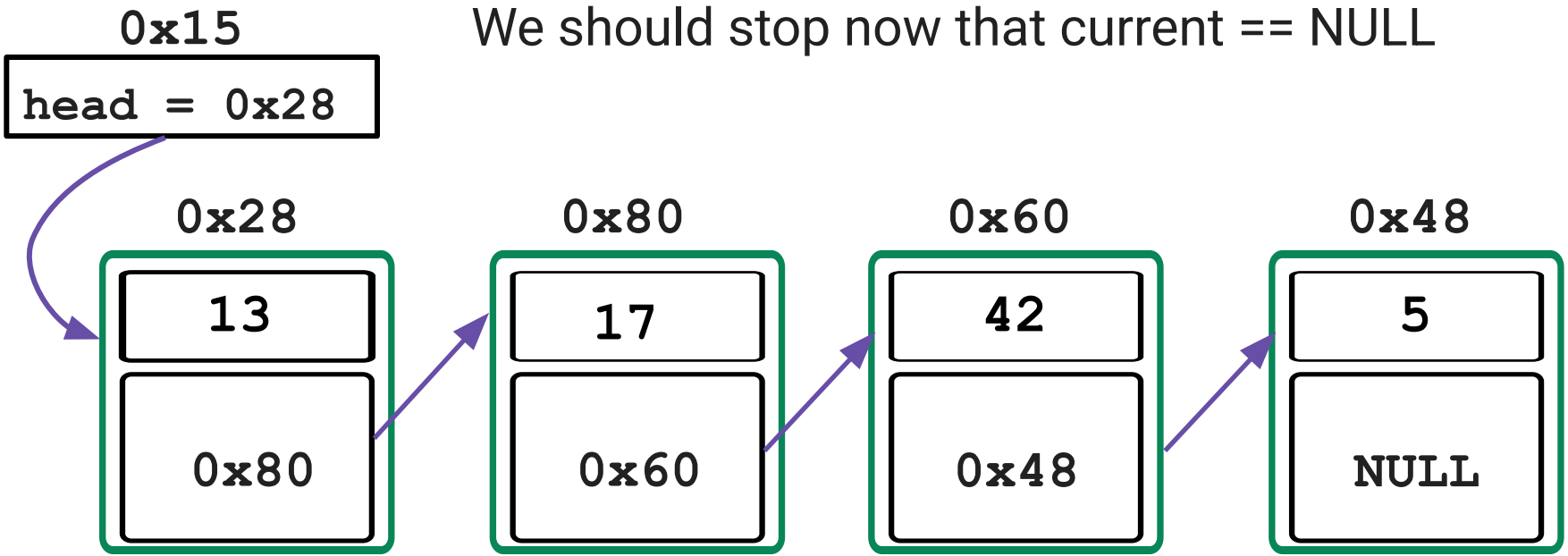
Traversing a List



Traversing a List



Traversing a List

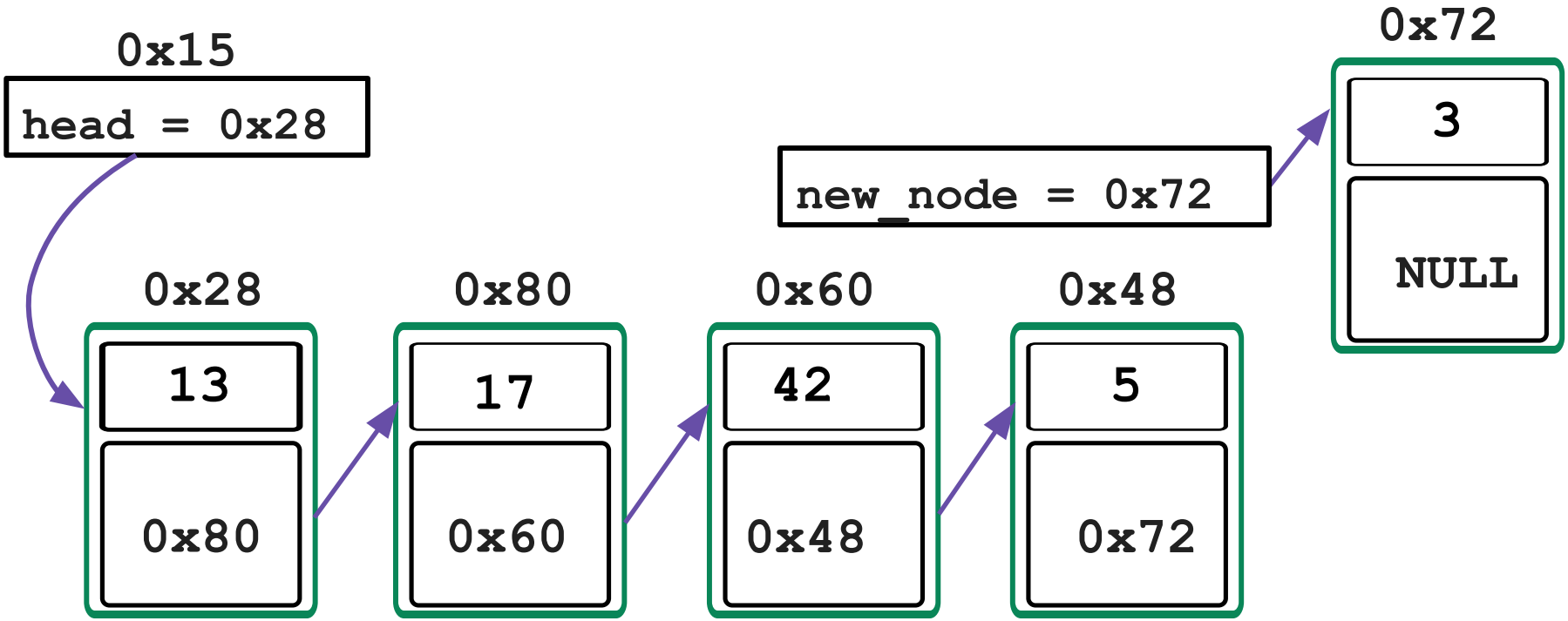


Inserting at the tail (end) of a list

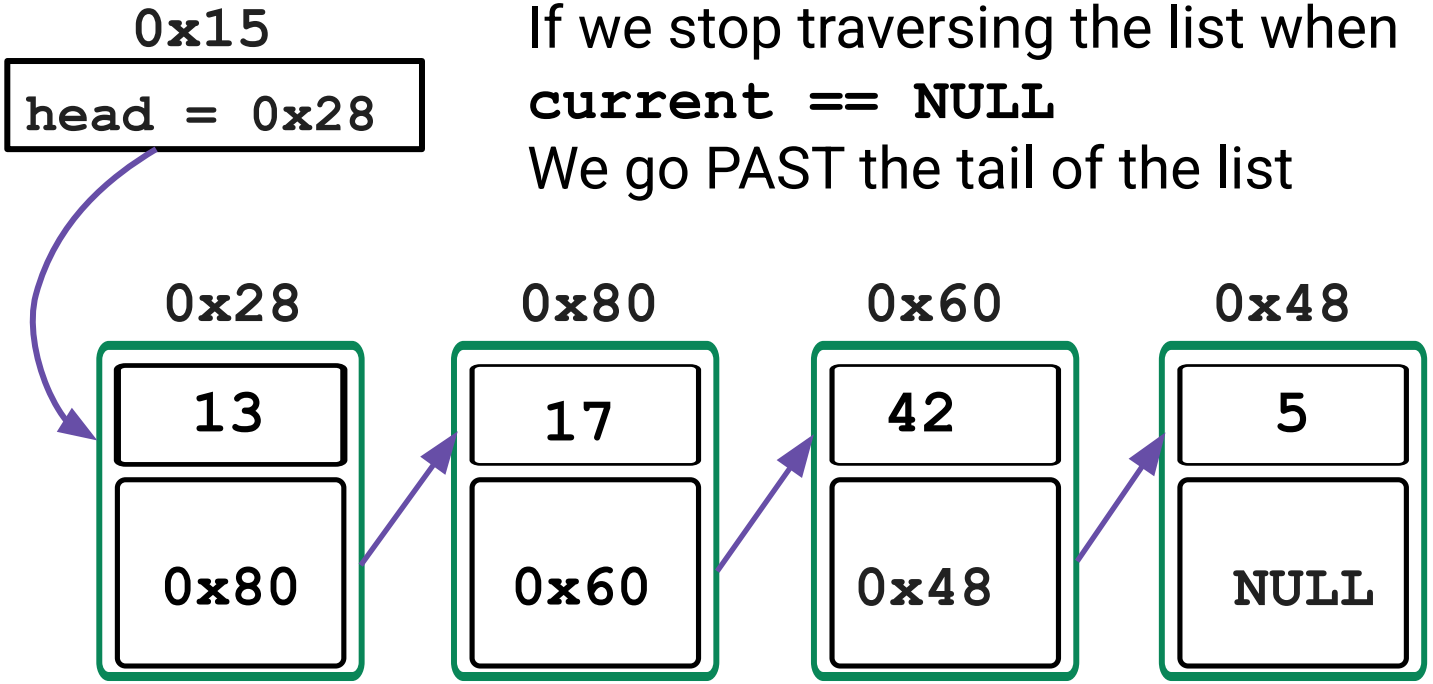
To insert a node at the end of the list we need to

- Find the last node in the list
- Connect the last node in the list to the new node

Insert at the Tail of the list



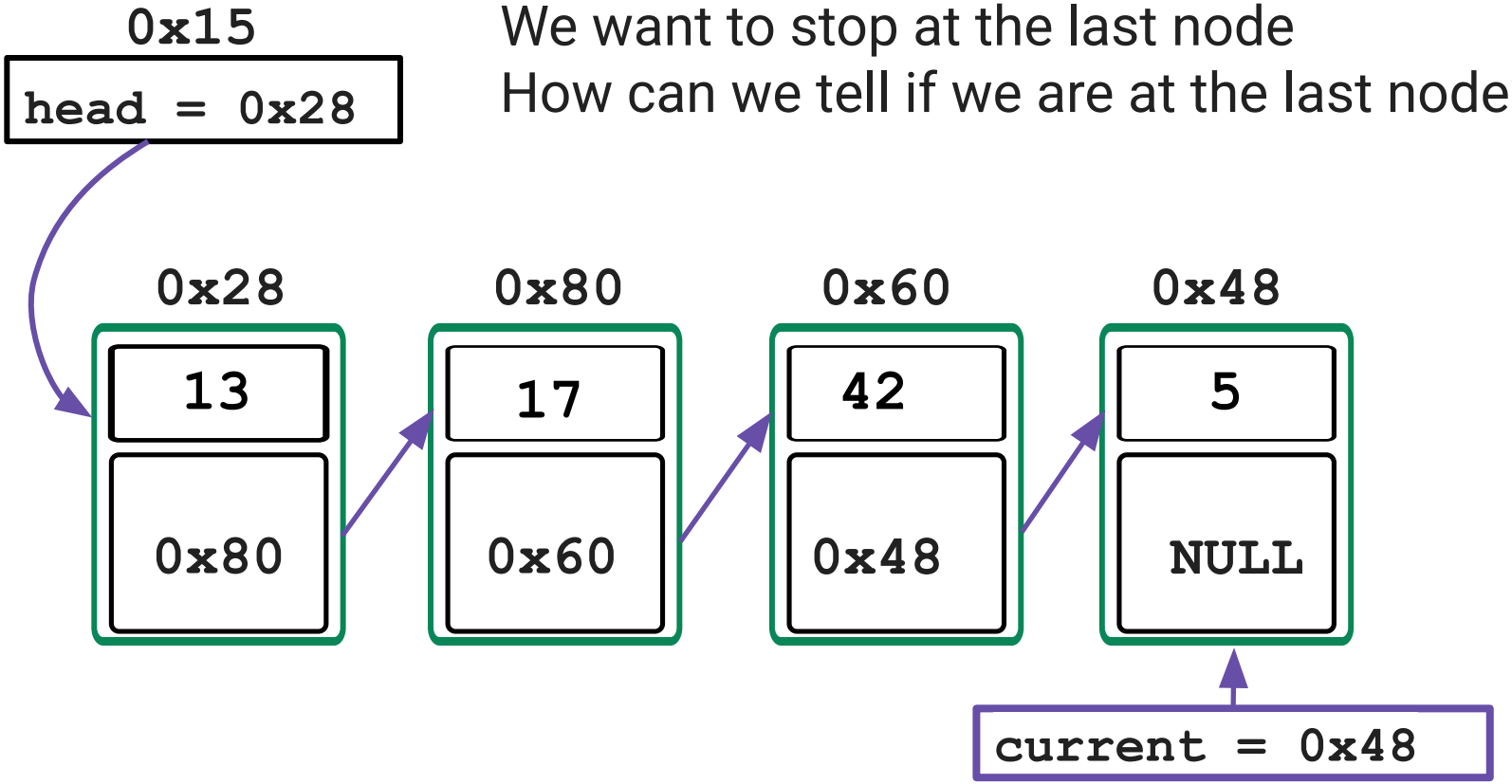
Finding the Tail of the list



If we stop traversing the list when
`current == NULL`
We go PAST the tail of the list

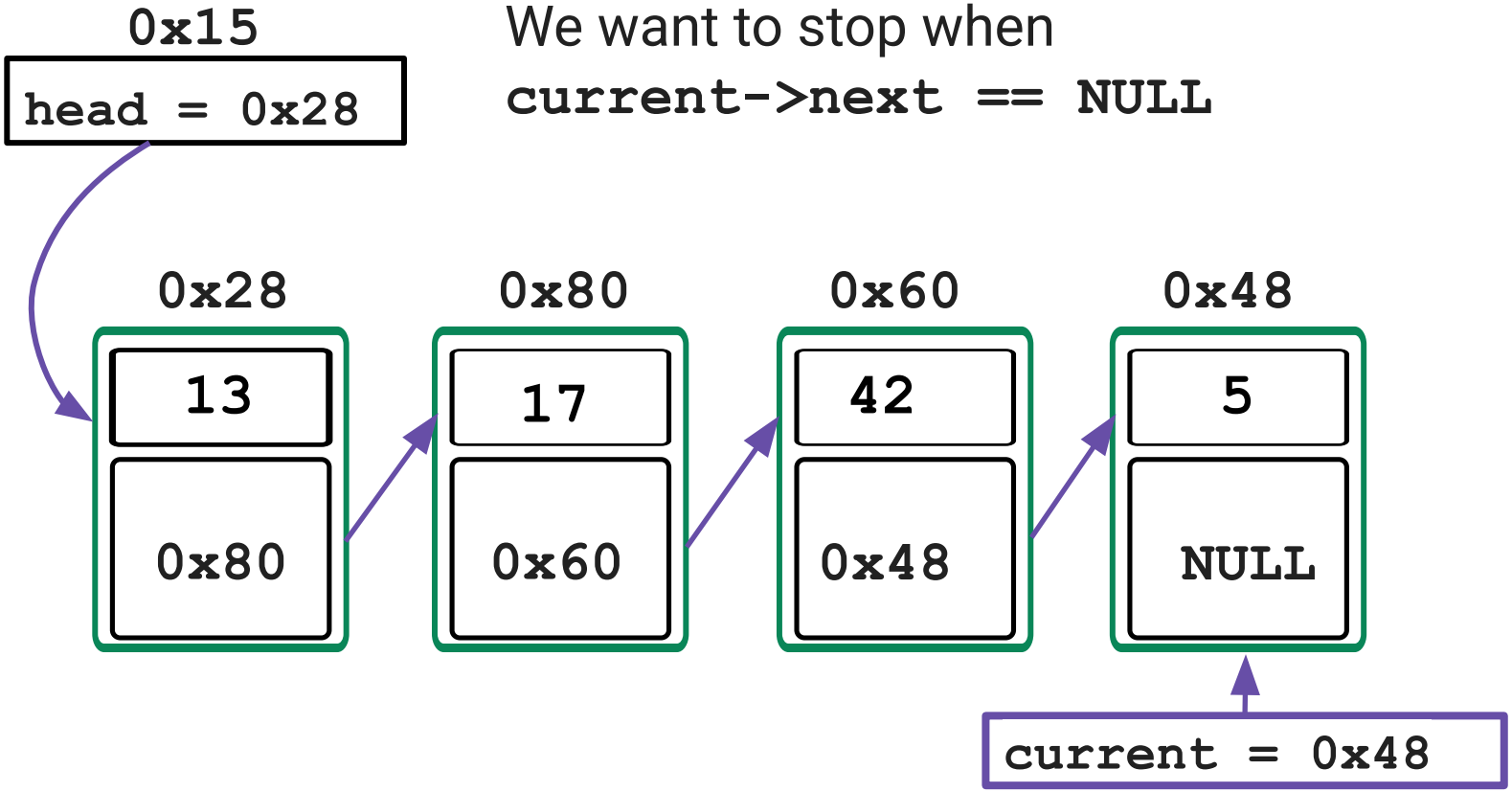
`current = NULL`

Finding the Tail of the list

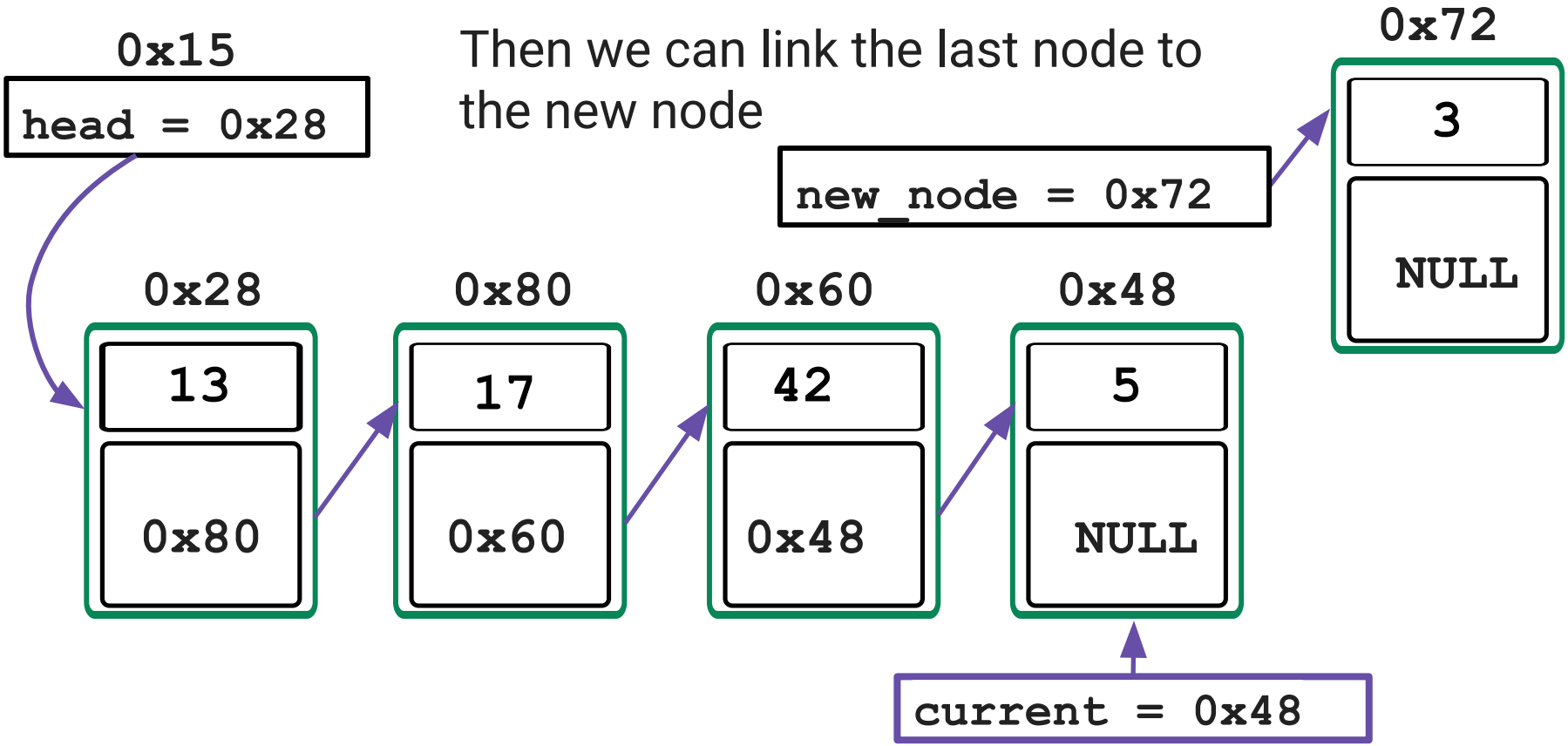


We want to stop at the last node
How can we tell if we are at the last node?

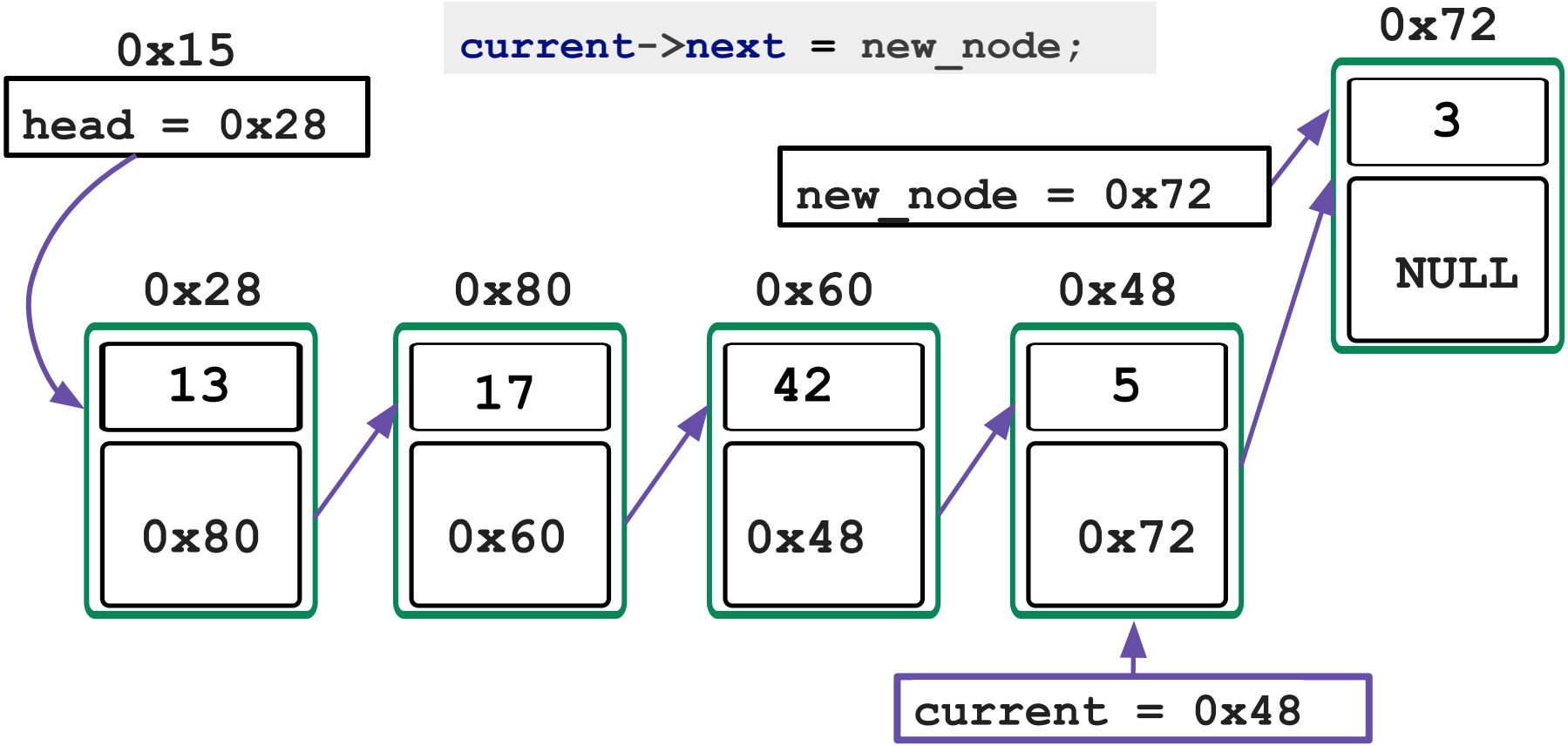
Finding the Tail of the list



Insert at the Tail of the list



Insert at the Tail of the list



Inserting at Tail

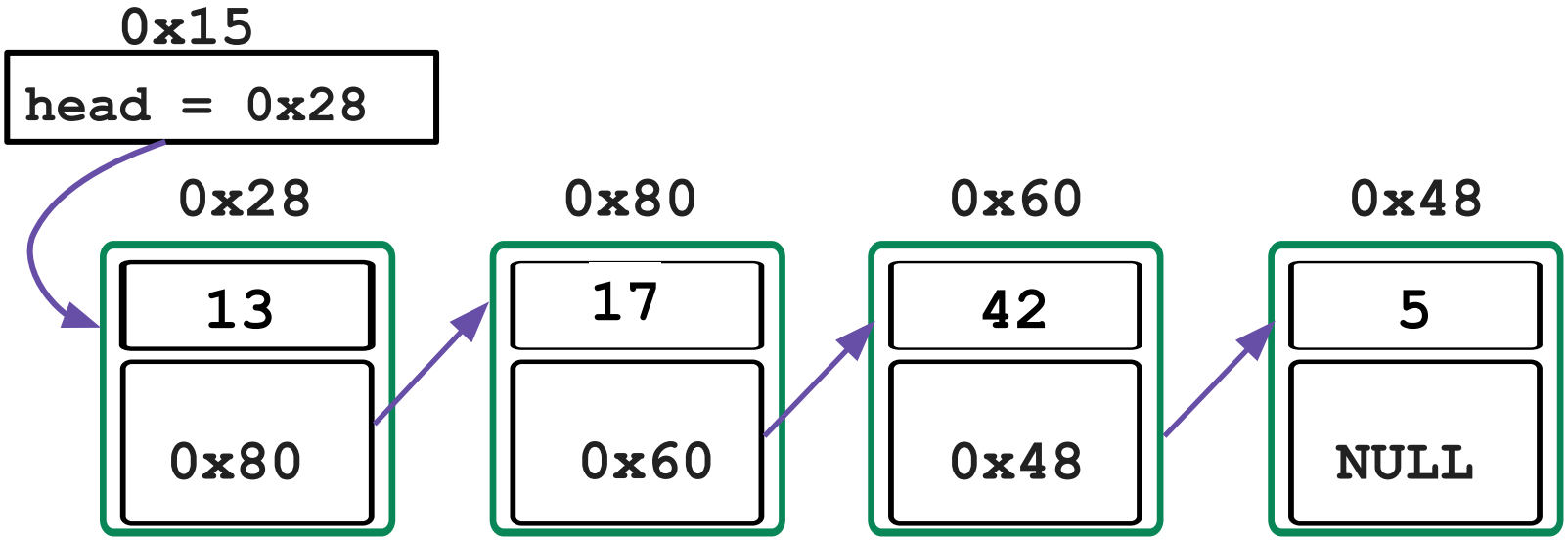
Warning: If we have an empty list

- head == NULL;
- so then current == NULL;
- so `current->next`
  will be dereferencing a NULL pointer and result in a run time error

```
struct node *insert_tail(struct node *head, int num){
    struct node *new_node = create_node(num);
    struct node *current = head;
     // Find the tail of the list
     while (current->next != NULL) {
```

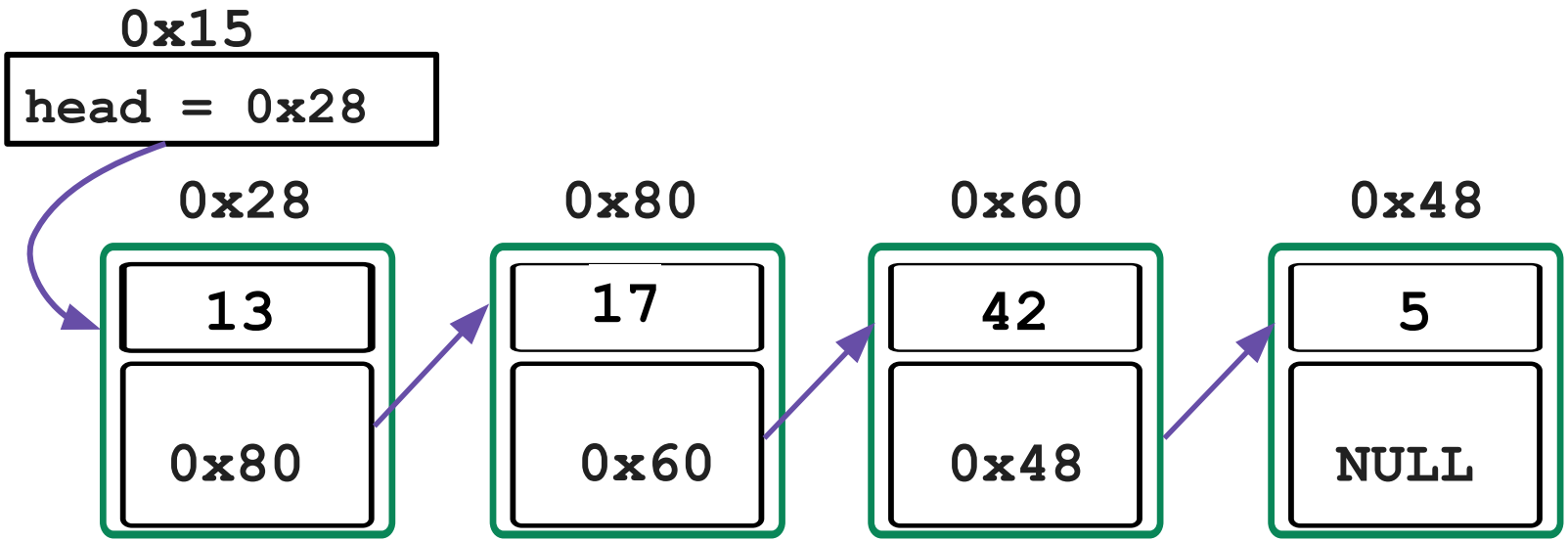
Inserting node at a given position

We want to insert a new node after position 2, assuming positions start at 1.



Inserting node at a given position

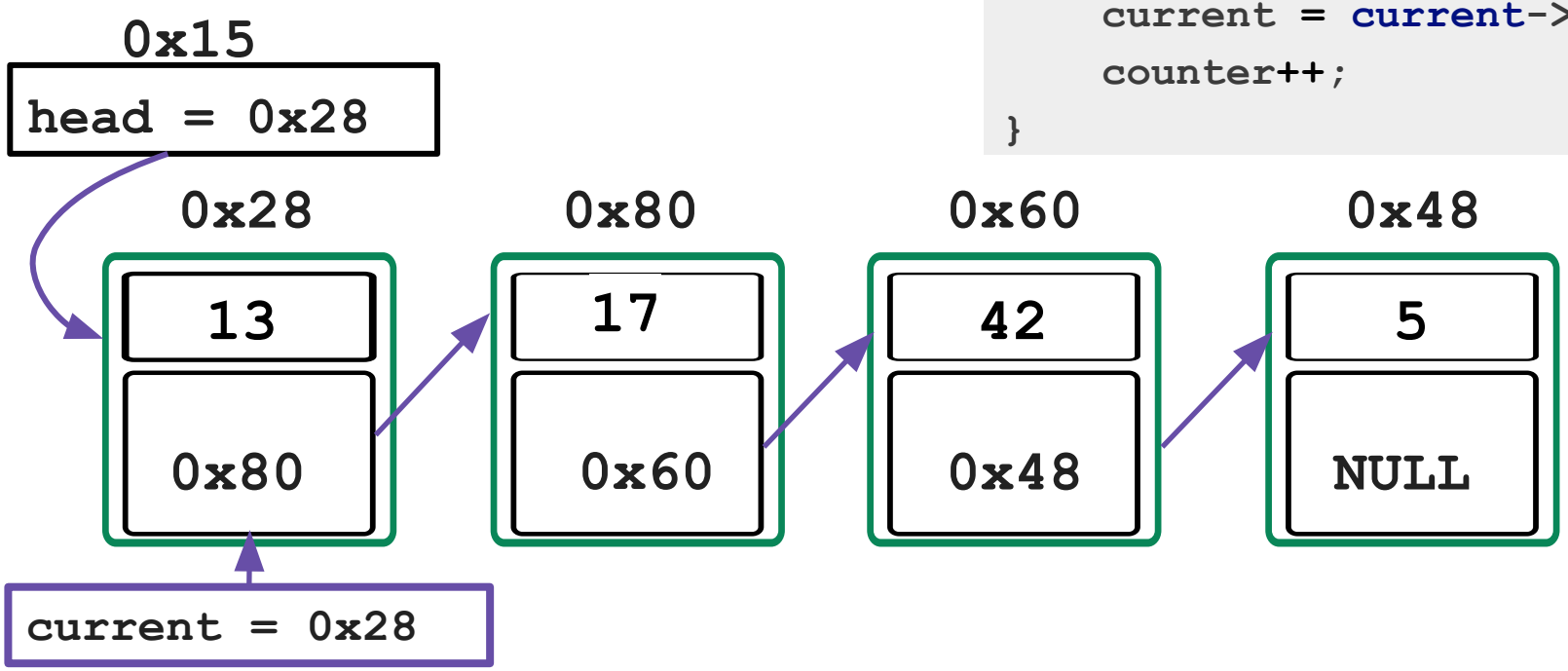
Use a counter to find the correct position to insert after



Inserting at Position

```
struct node *current = head;
int counter = 1;
while (counter < position) {
    current = current->next;
    counter++;
}
```

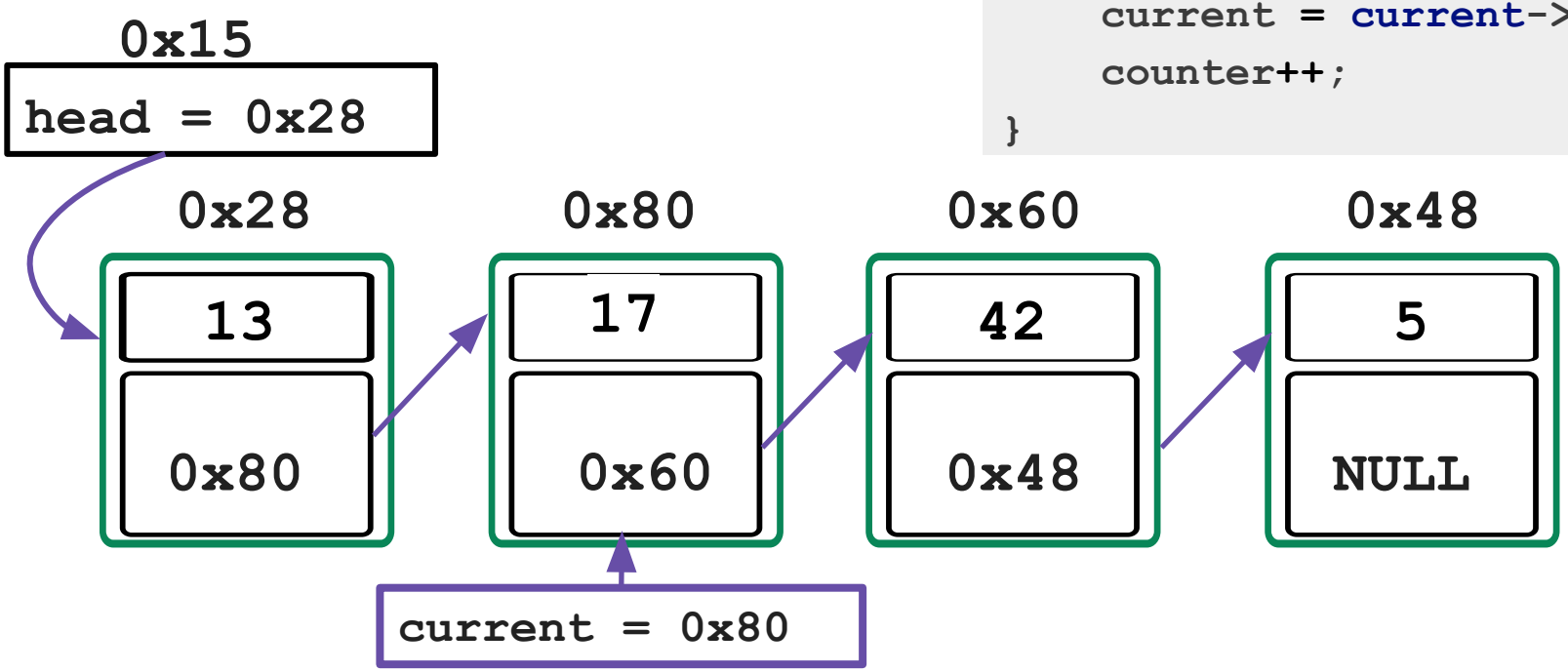
counter = 1



Inserting at position

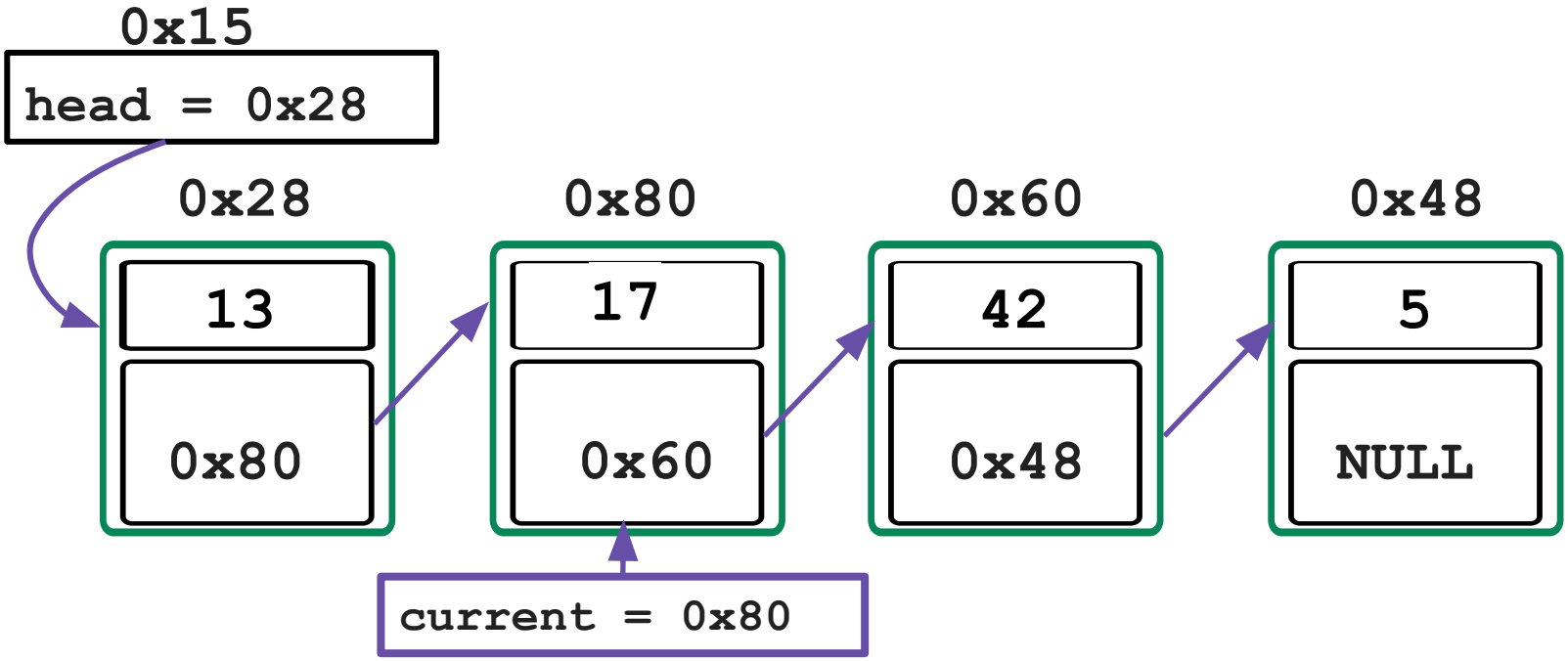
```
struct node *current = head;
int counter = 1;
while (counter < position) {
    current = current->next;
    counter++;
}
```

counter = 2

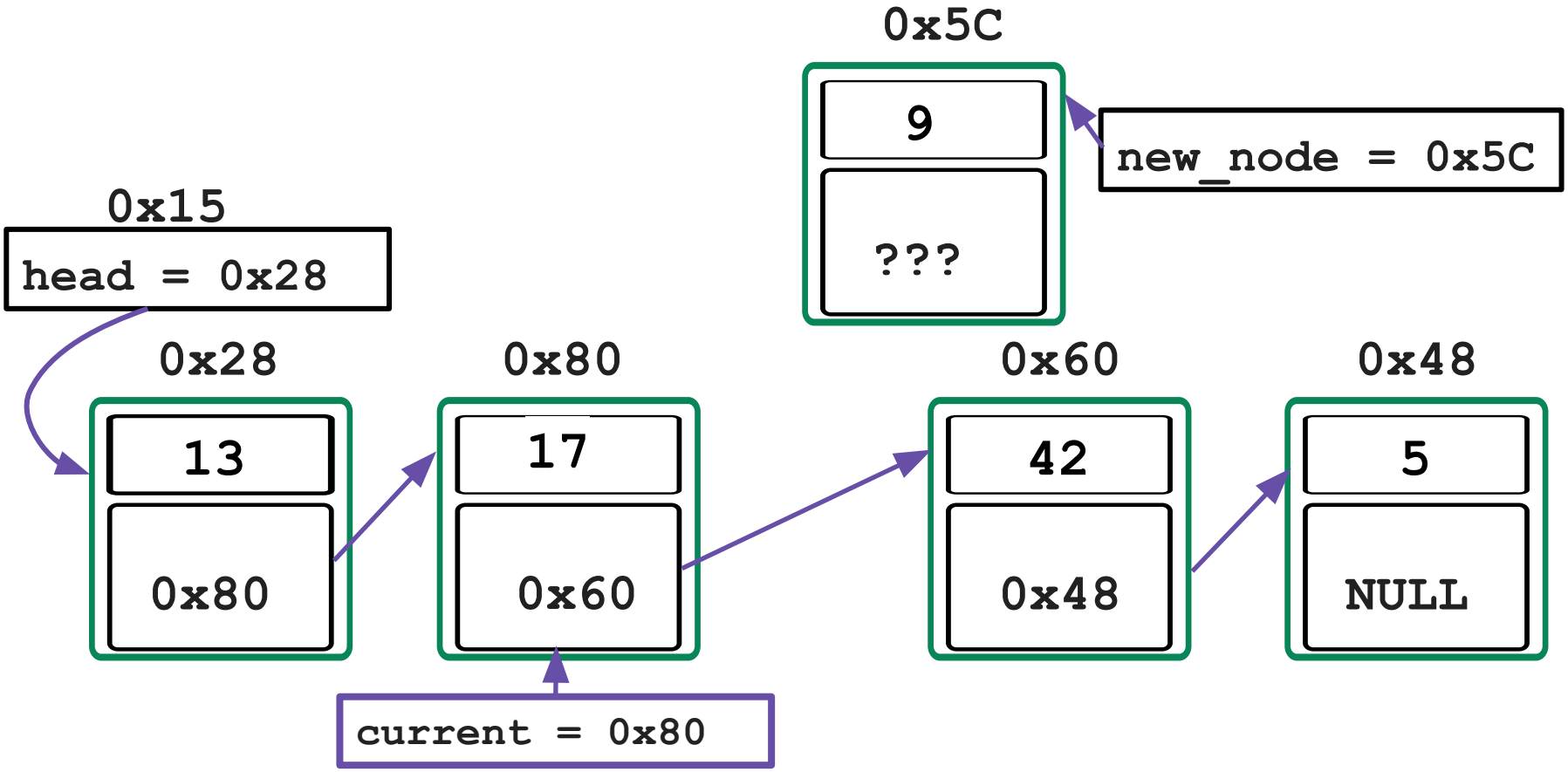


Inserting at position

Now we want to connect our new node. It should come after the current node, but before current->next

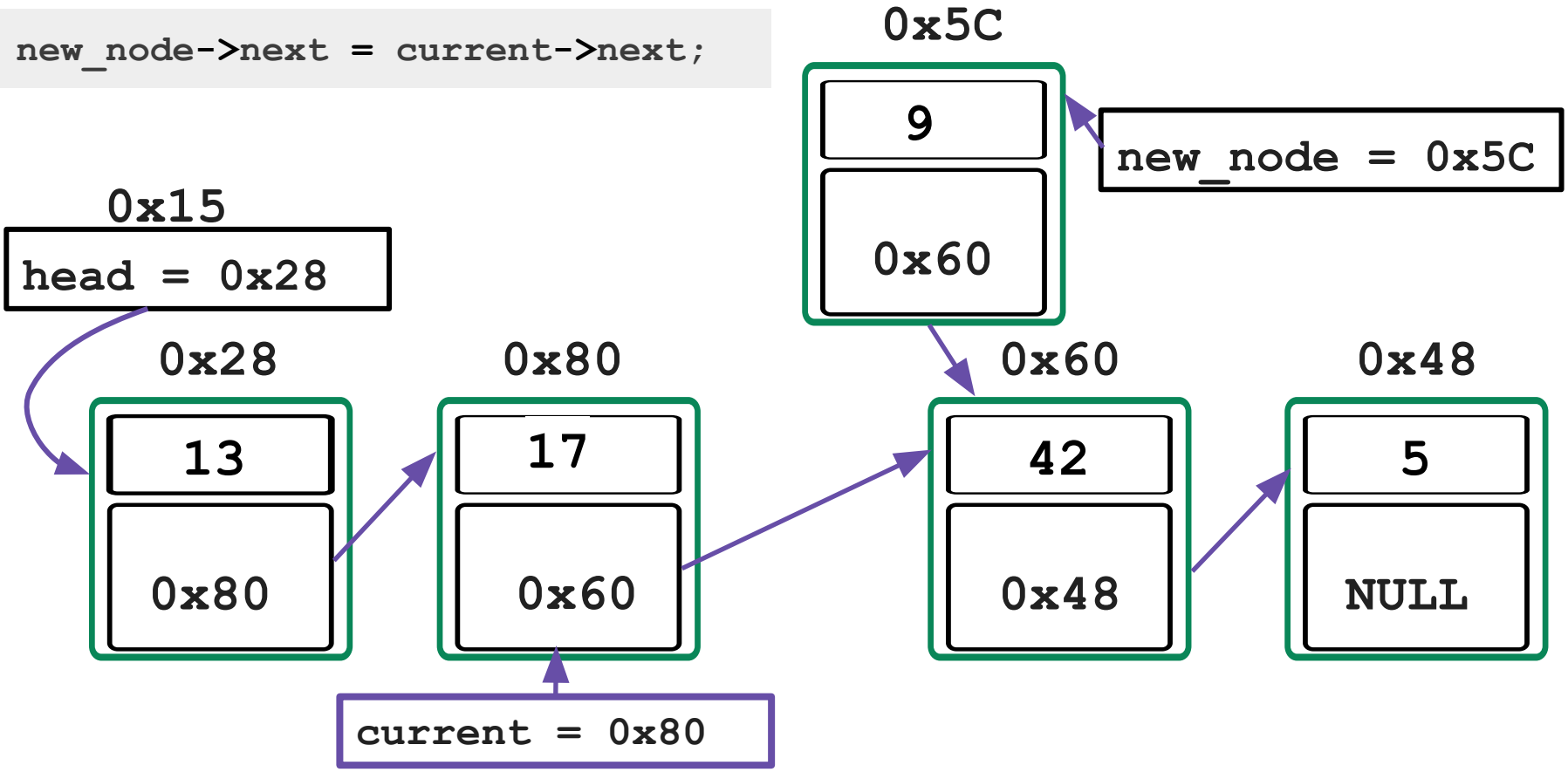


Inserting at Position



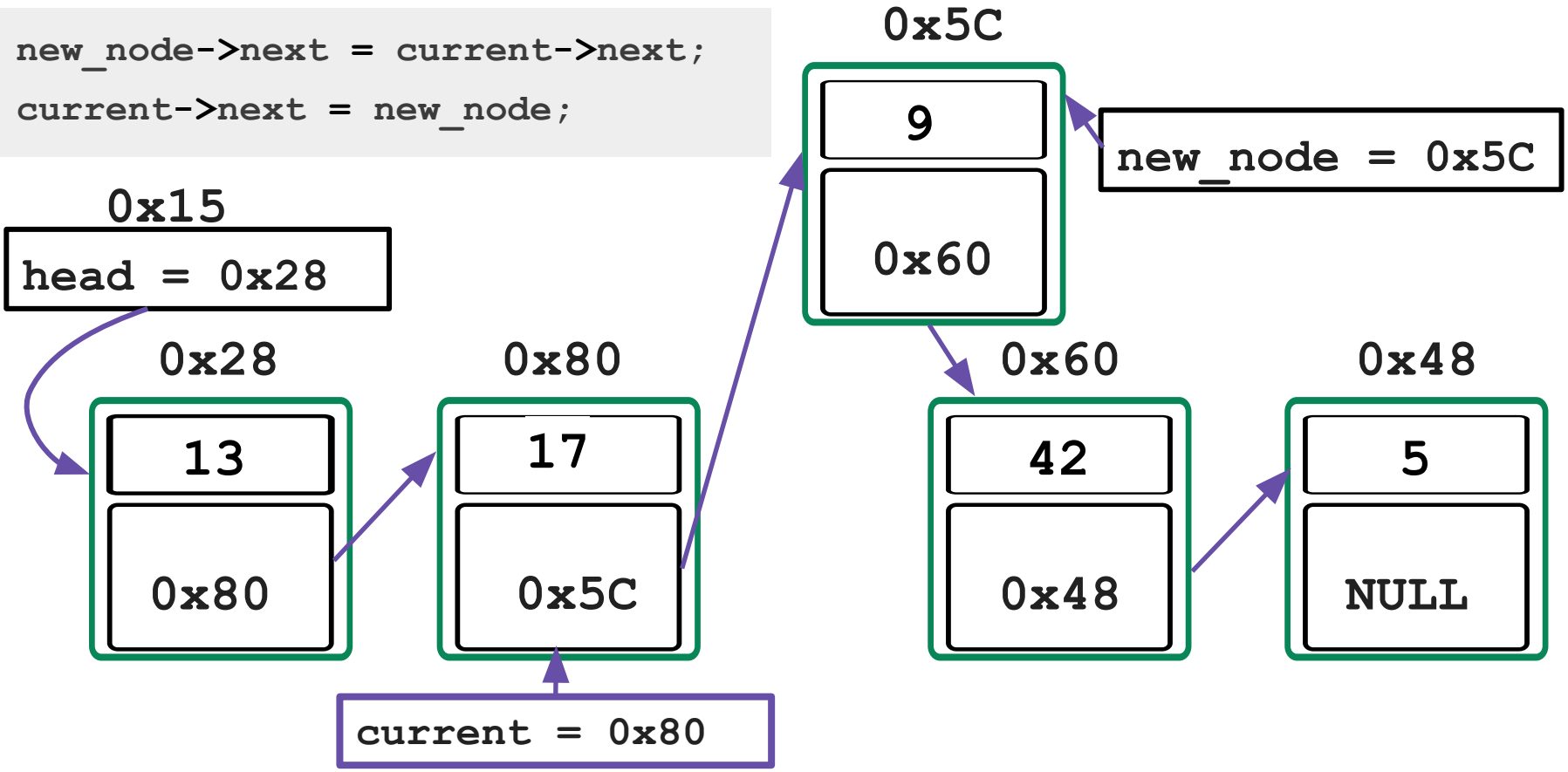
Inserting at Position

```
new_node->next = current->next;
```



Inserting at Position

```
new_node->next = current->next;
current->next = new_node;
```



Coding: Inserting at Position

- What other special cases are there?

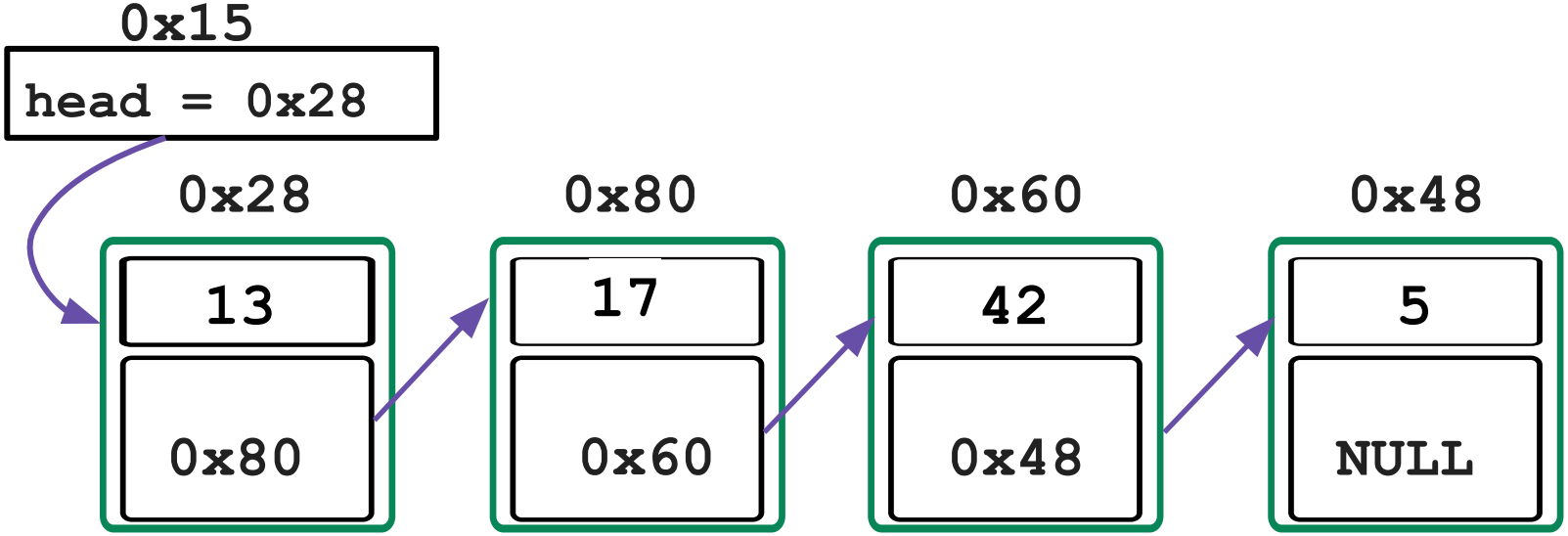
Coding: Inserting at Position

- What other special cases are there?
  - Empty list
  - Insert at the start (position 0 or less).
  - Insert at tail
  - Insert past tail
  - Anything else we should check?

Linked List Deletion

Deleting the First Node in a Linked List

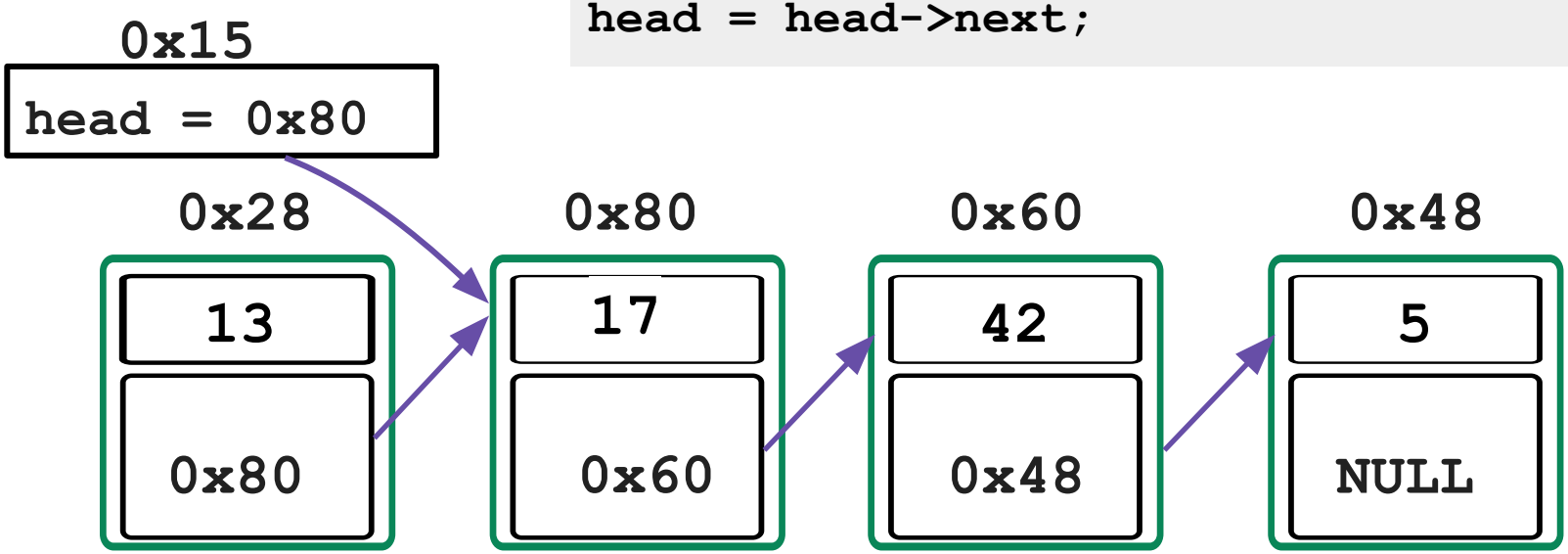
If our list is not empty, we want to make the second node the new head of the list and free the first node that we want to delete.



Deleting the First Node in a Linked List

What would be the problem with updating head first?

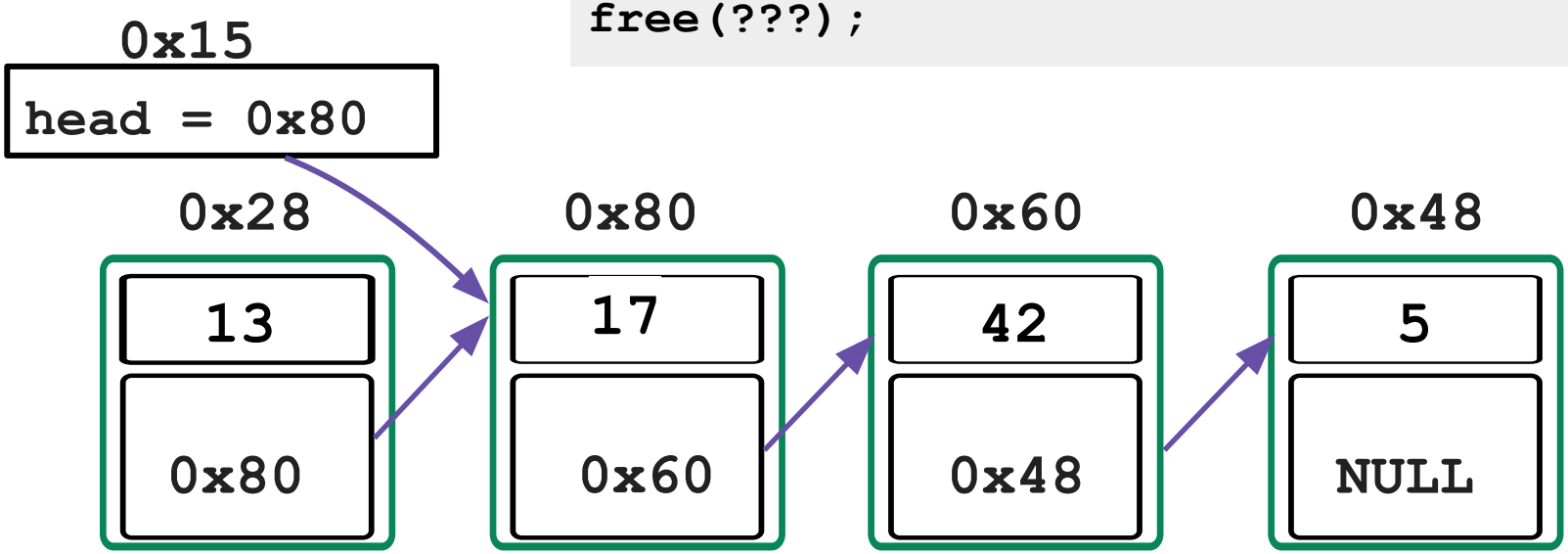
```
head = head->next;
```



Deleting the First Node in a Linked List

We now have no pointer to the first node so we can't free it!

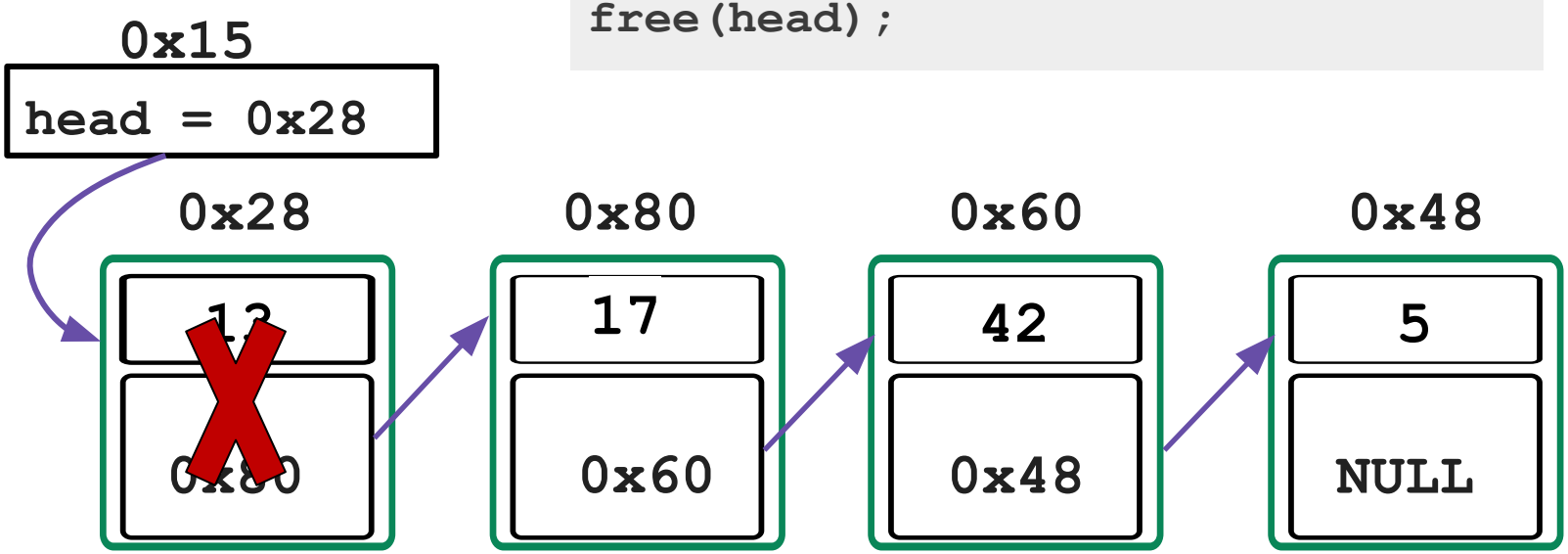
```
free(???);
```



Deleting the First Node in a Linked List

What would be the problem calling free on head first?

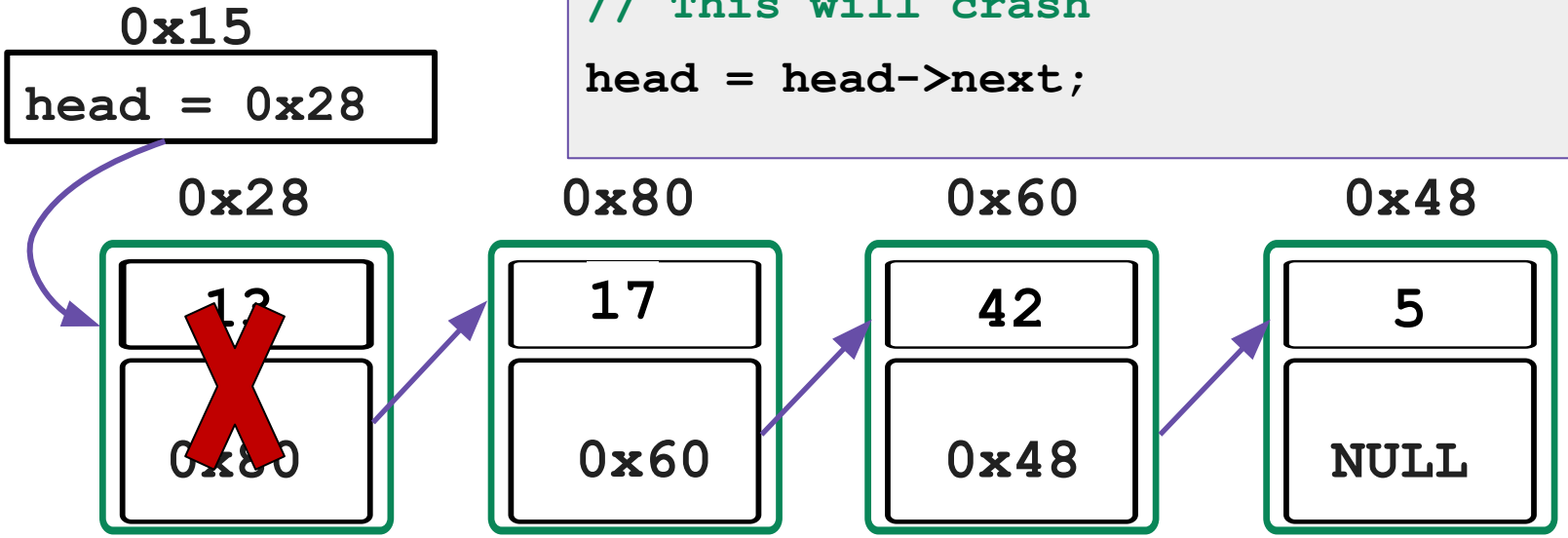
```
free(head);
```



Deleting the First Node in a Linked List

We can't access memory that has been freed. We have lost the rest of the list

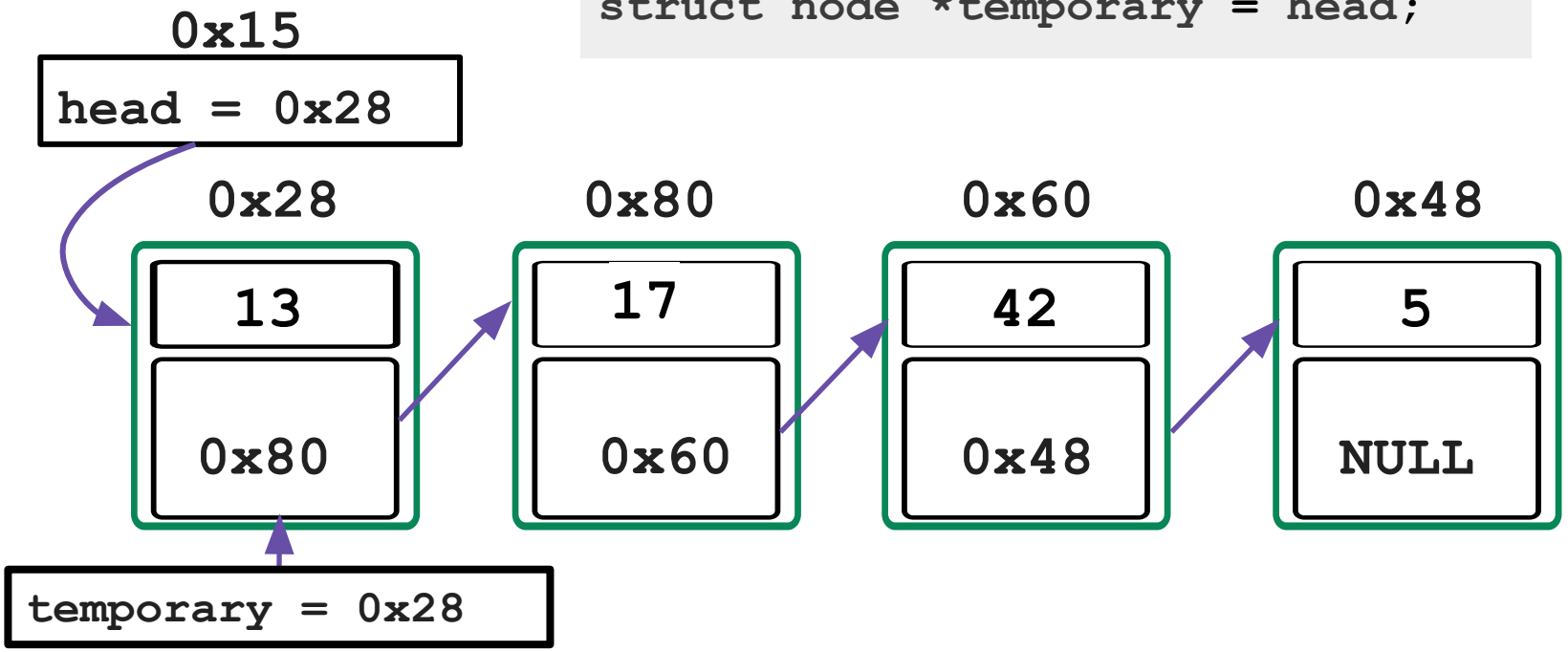
```
// This will crash
head = head->next;
```



Deleting the First Node in a Linked List

Let's create a pointer to the first node

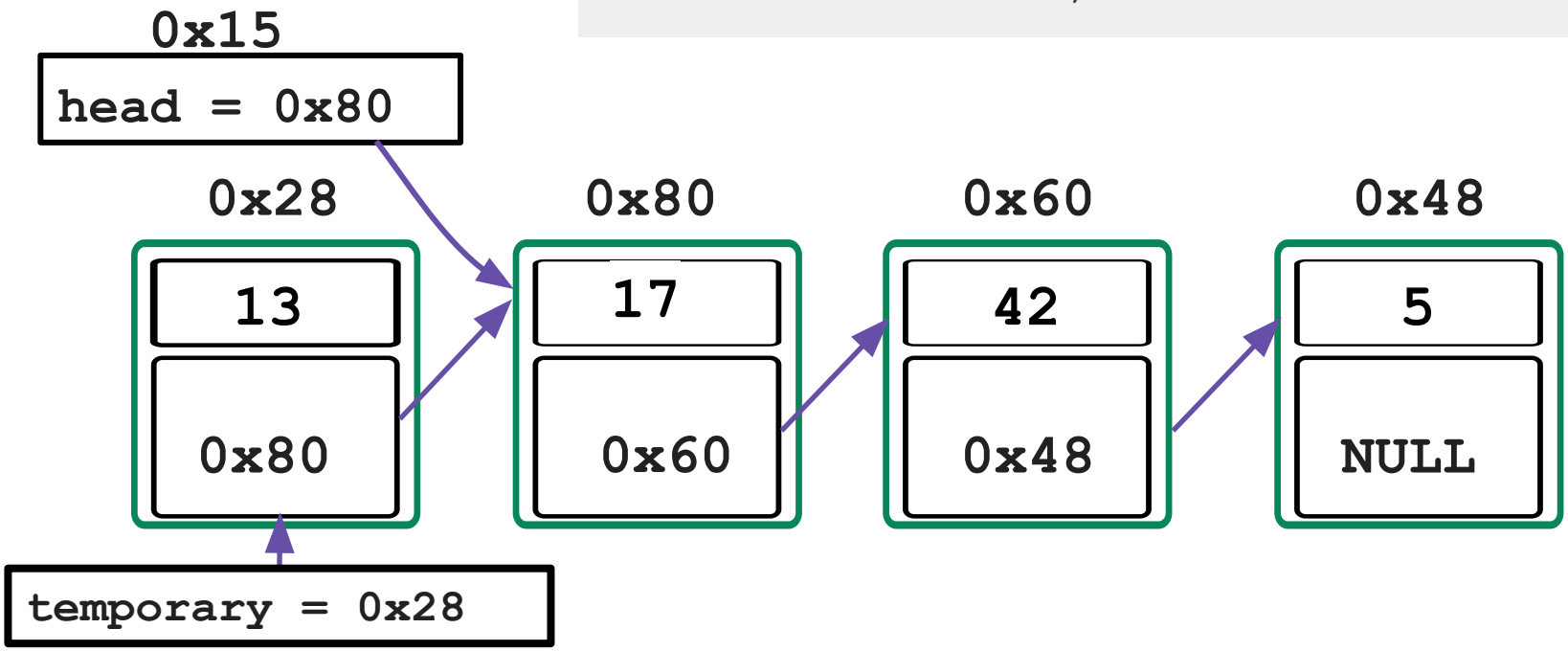
```
struct node *temporary = head;
```



Deleting the First Node in a Linked List

Now we can update head

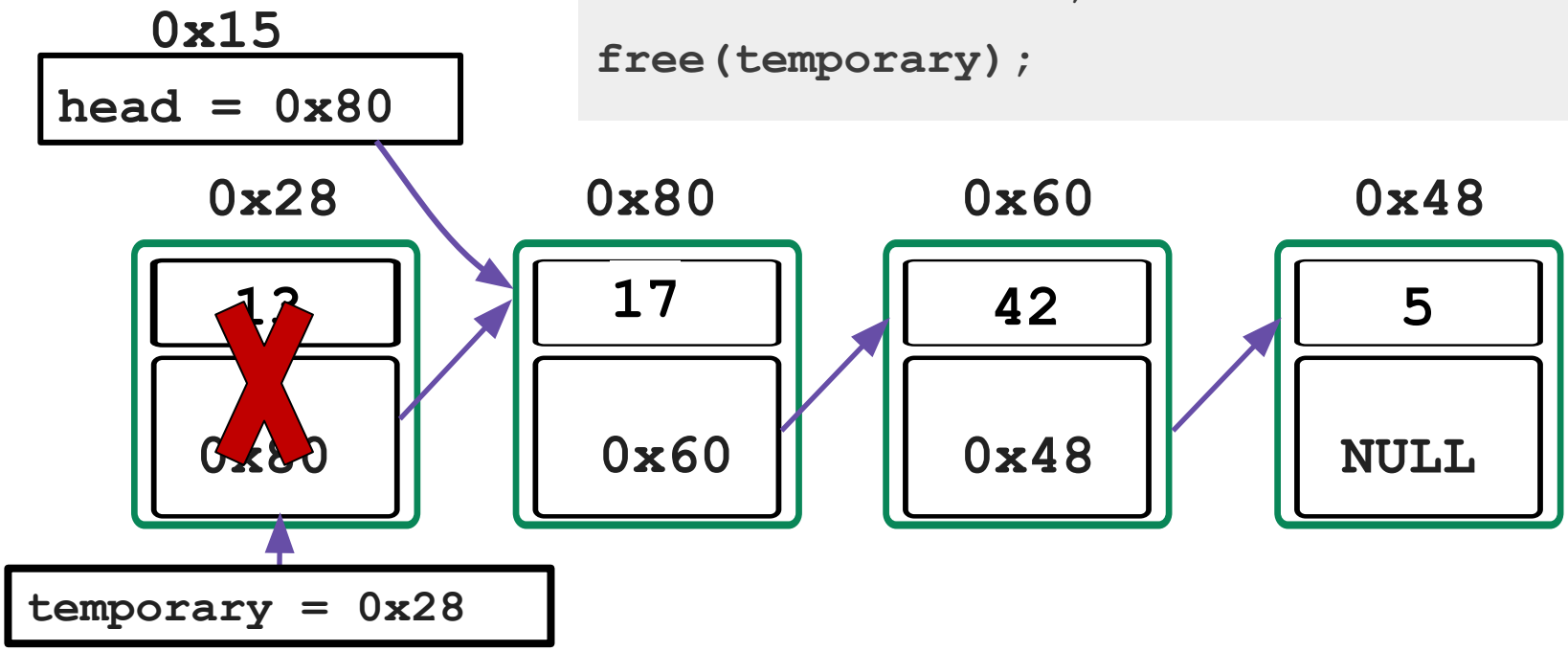
```
head = head->next;
```



Deleting the First Node in a Linked List

Now we can free the first node

```
head = head->next;
free(temporary);
```



Deleting the First Node in a Linked List

What should the prototype of the function look like?

How should we call the function?

Are there any cases we have not considered?

Deleting the First Node in a Linked List

We need to consider the case when the list is empty

- If it is empty we can't delete anything
- We just return the head of the list which would be NULL

```
if (head == NULL) {
    return head; //or return NULL;
}
```

Memory Leaks

- We malloc for every node in our lists.
- For small programs that do not run for a long time, not freeing the nodes may not cause any issues.
- For programs that run for a long time
  - For example
    - Web servers (e.g. used by Youtube, or Amazon etc)
    - Game servers (e.g. Minecraft server)
    - Your own web browser or local games
  - Memory leaks can gradually increase memory usage, leading to degraded performance, crashes, restarts and potentially service outages.

Free List the wrong way

What is wrong with this code?

```
// Free all nodes in a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        free(current);
        current = current->next;
    }
}
```

Free List the wrong way

Don’t forget that if you free memory, you can’t use it!

```
// Free all nodes in a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        free(current);
        // Accessing memory that has just been freed
        current = current->next;
    }
}
```

Warning: Memory Use After Free 📢🔔

- Using memory after freeing is a serious bug ⚠️
- On dcc our program will crash at runtime and will helpfully tell us the problem
- In production code, it can cause:
  - Crashes
  - Bugs
  - **Serious security Vulnerabilities** 🚨

    Real world example: Google Chrome's WebAudio component 2019. It allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page. NVD - CVE-2019-13720

Free List the Correct Way

Let's test it and check it with dcc --leak-check

```
// Delete all nodes from a given list
void free_list(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        head = head->next;
        free(current);
        current = head;
    }
}
```

Revision Exercise

Write a function to search for a given value in a linked list
return 1 if it exists and 0 otherwise

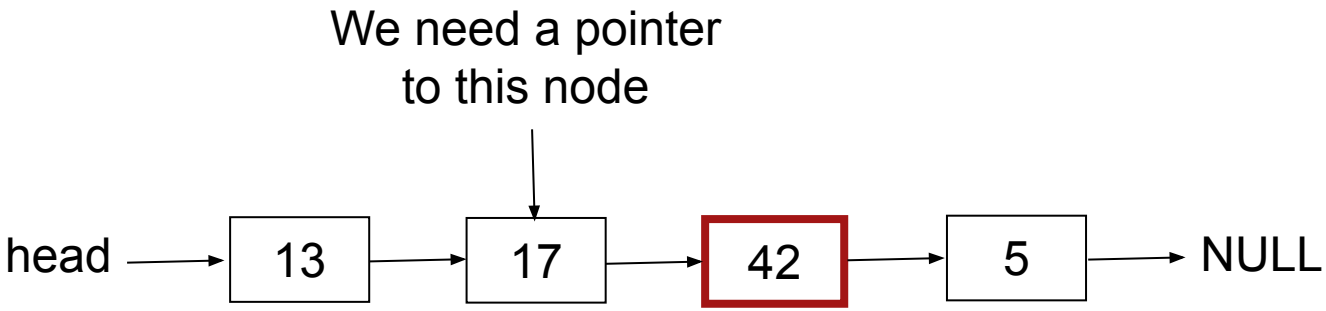
What should the prototype be?
What cases should we make sure we test?

Search and Delete

- We want to search for a node with a particular value in it and then delete it
- Where could the item be
  - Nowhere - if it is an empty list or the list does not contain the value
  - At the head (deleting the first node in the list)
  - Between any 2 nodes in the list
  - At the tail (deleting the last node in the list)
  - There could be multiple occurrences! For now let’s just consider the first occurrence

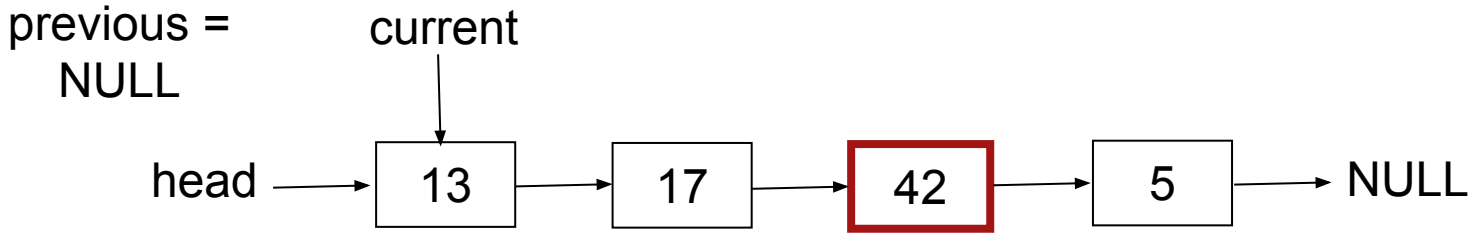
Search and delete: between 2 nodes

- To delete a node we need to link the previous node to the next node
  - E.g. if we want to delete the node with 42, we need to find the node before it



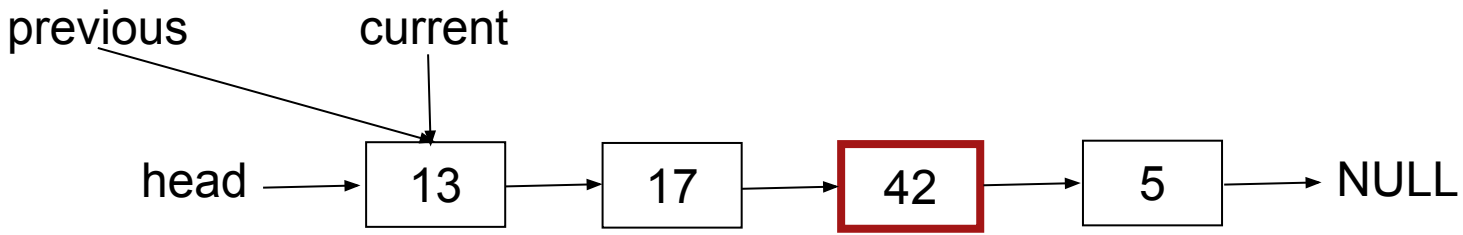
Search and delete: between 2 nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



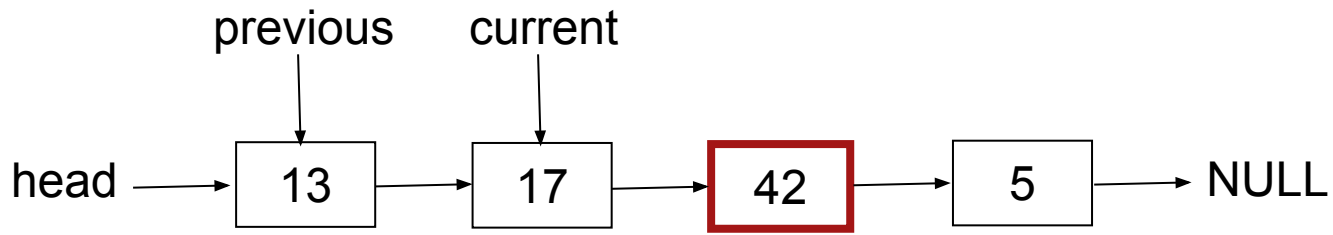
Search and delete: between 2 nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



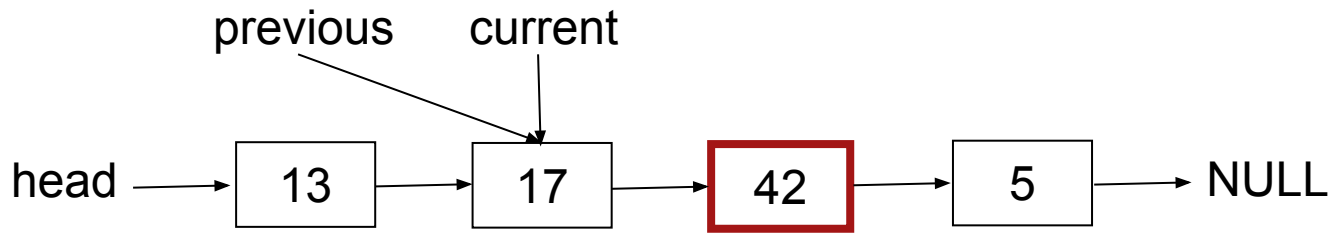
Search and delete: between 2 nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



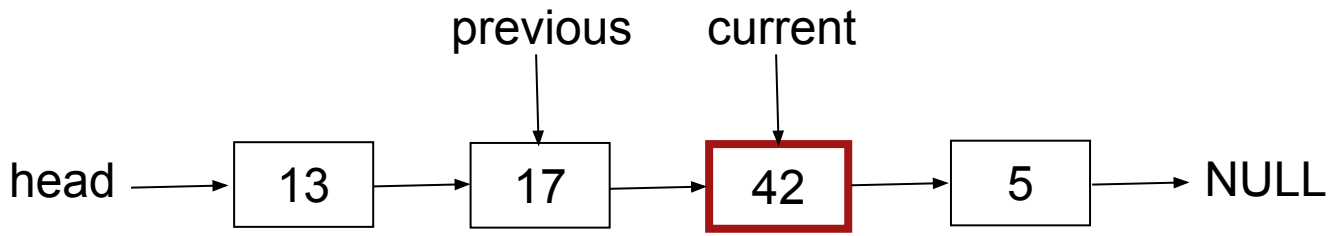
Search and delete: between 2 nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



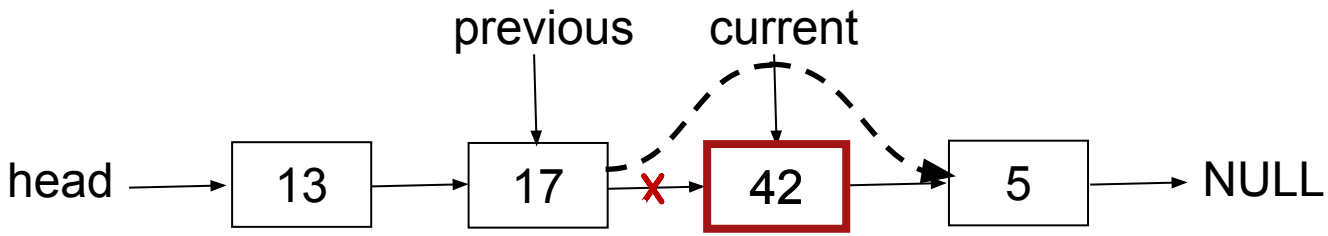
Search and delete: between 2 nodes

```
// Approach 1: Have a previous node pointer
struct node *previous = NULL;
struct node *current = head;
while (current != NULL && current->data != value) {
    previous = current;
    current = current->next;
}
```



Search and delete: Approach 1

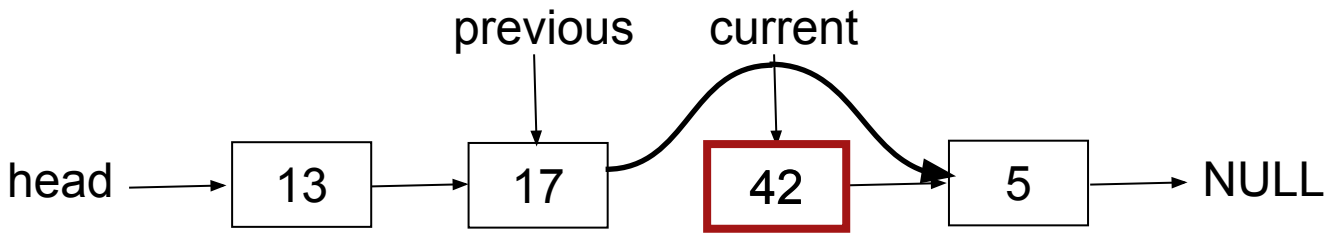
Then we need to connect previous node to the one after the one we are deleting.



Search and delete: Approach 1

Then we need to connect previous node to the one after the one we are deleting.

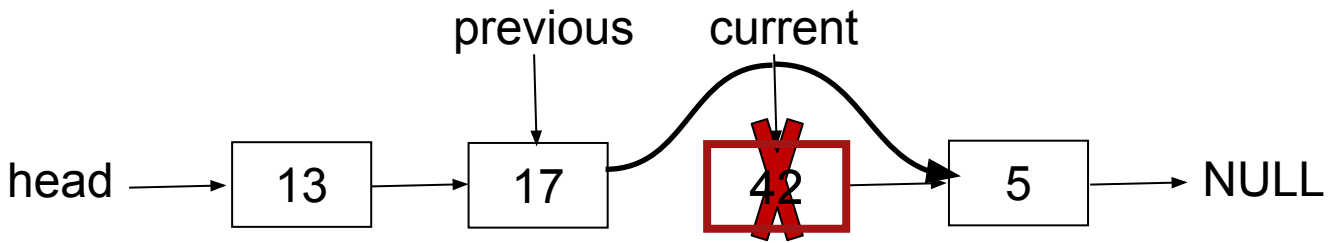
```
previous->next = current->next;
```



Search and delete: Approach 1

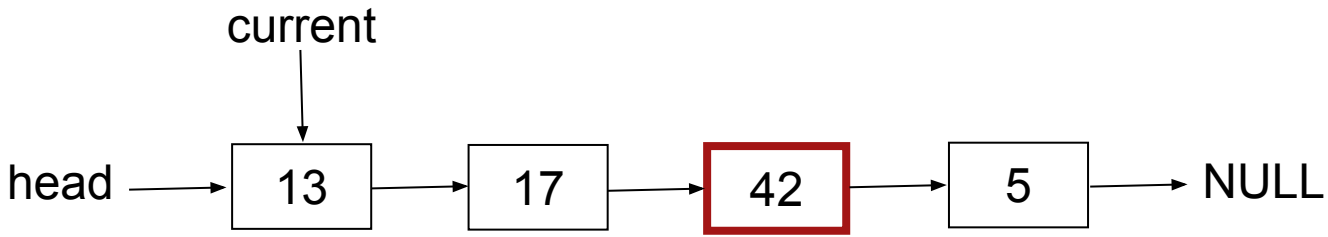
Now we can free the node we want to delete

```
free(current);
```



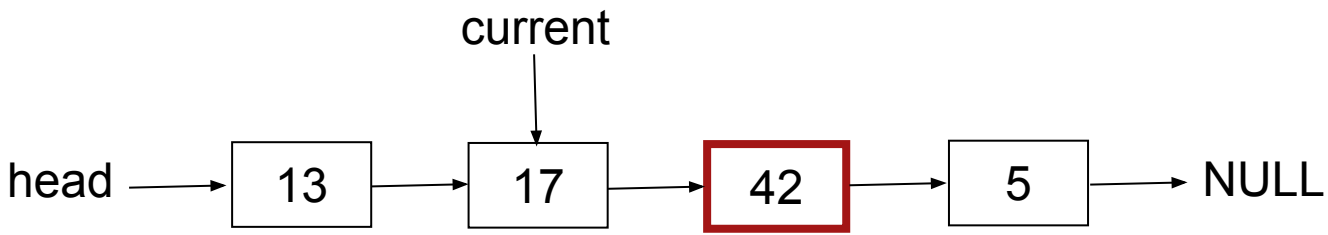
Search and delete Approach 2

```
// Approach 2: Just use 1 pointer to traverse
// but check the next node
struct node *current = head;
while (current->next != NULL &&
       current->next->data != value) {
    current = current->next;
}
```



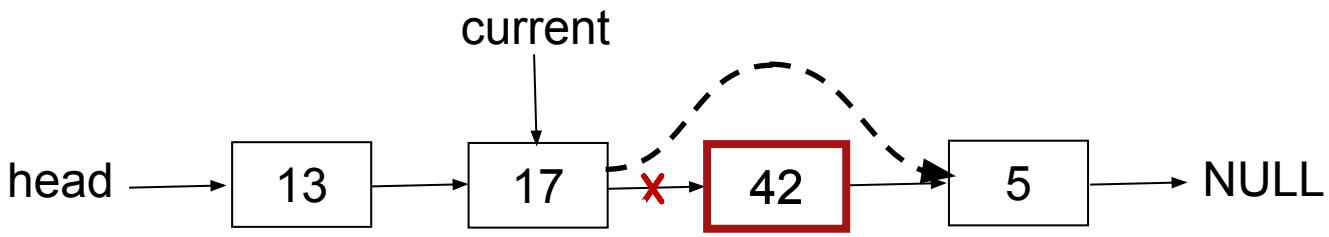
Search and delete Approach 2: general case

```
// Approach 2: Just use 1 pointer to traverse
// but check the next node
struct node *current = head;
while (current->next != NULL &&
       current->next->data != value) {
    current = current->next;
}
```



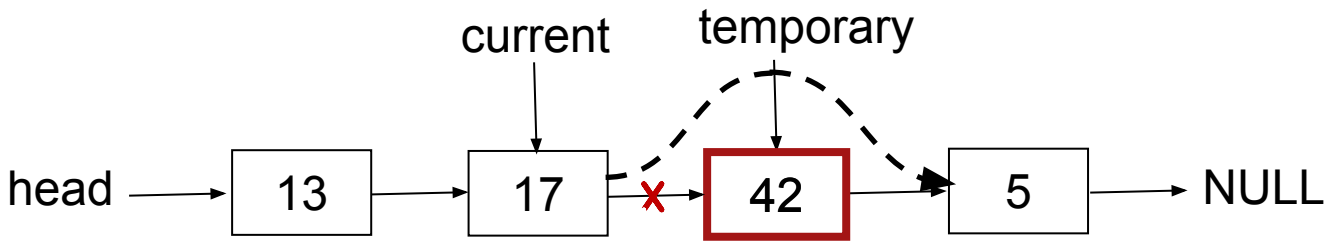
Search and delete: Approach 2

Then we need to connect current node to the one after the one we are deleting. But we still need a pointer to the node we want to free. How can we do that?



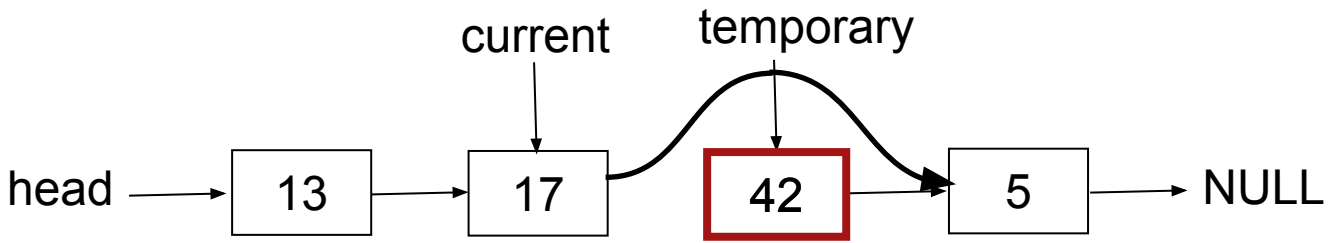
Search and delete: Approach 2

```
struct node *temporary = current->next;
```



Search and delete: Approach 2

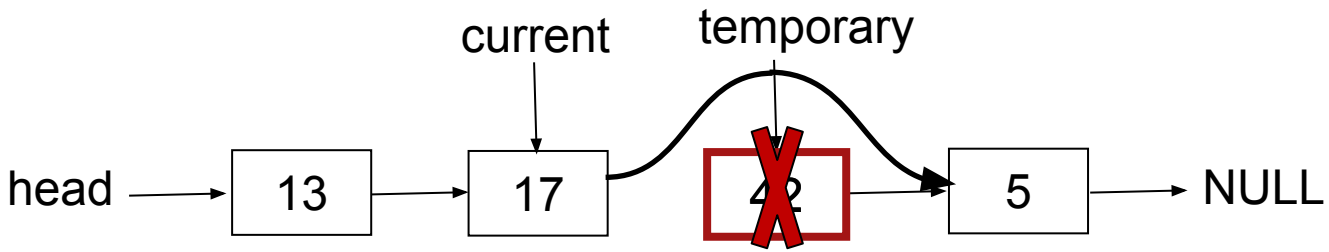
```
struct node *temporary = current->next;
current->next = temporary->next;
```



Search and delete: Approach 2

Now we can free the node we want to delete

```
free(temporary);
```



Coding

Let’s code up the first approach

Extension Exercises:

Code up the other approach

Extend our first approach to delete all occurrences.

What did we learn today?

- Recap on Basics of Linked Lists
- Recap of Inserting into Linked Lists at any position
- Deleting elements
  - First node
  - Freeing All nodes
  - Search and Delete
  - Issues:
    - Memory Leaks
    - Accessing memory after being freed

Next Lecture

- Searching and deleting
  - Deleting from anywhere in a linked list recapped and/or continued
- Linked Lists a Larger Application.
  - Linked Lists as members in other structs
  - Linked Lists with more complex data (other than just int)
  - Helpful for assignment 2

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/pXdtYN4xgE

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1511@unsw.edu.au