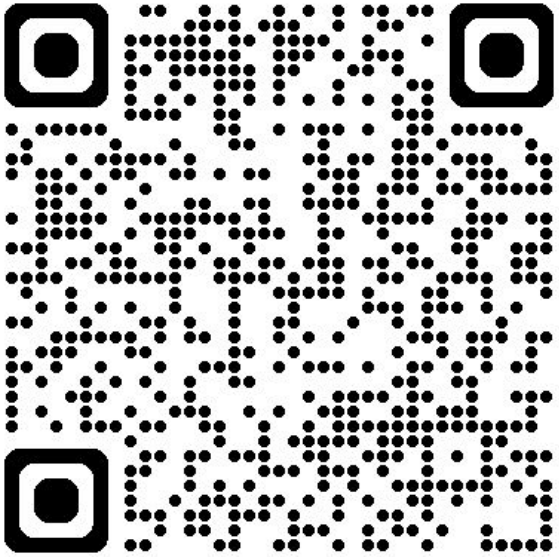
COMP1511/1911 Programming Fundamentals

Week 8 Lecture 2

Linked Lists
Deletion

Link to Week 8 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs 1511/25T3/code/week_8/



Revision Sessions This Week

Thursday 2-4pm (Online on Microsoft Teams)

Please sign up for the revision sessions and vote for your favourite topic on the forum.

Last Lecture

- Linked List recap
- List Length
- Inserting nodes
  - At the end(tail)
  - Inserting in the middle

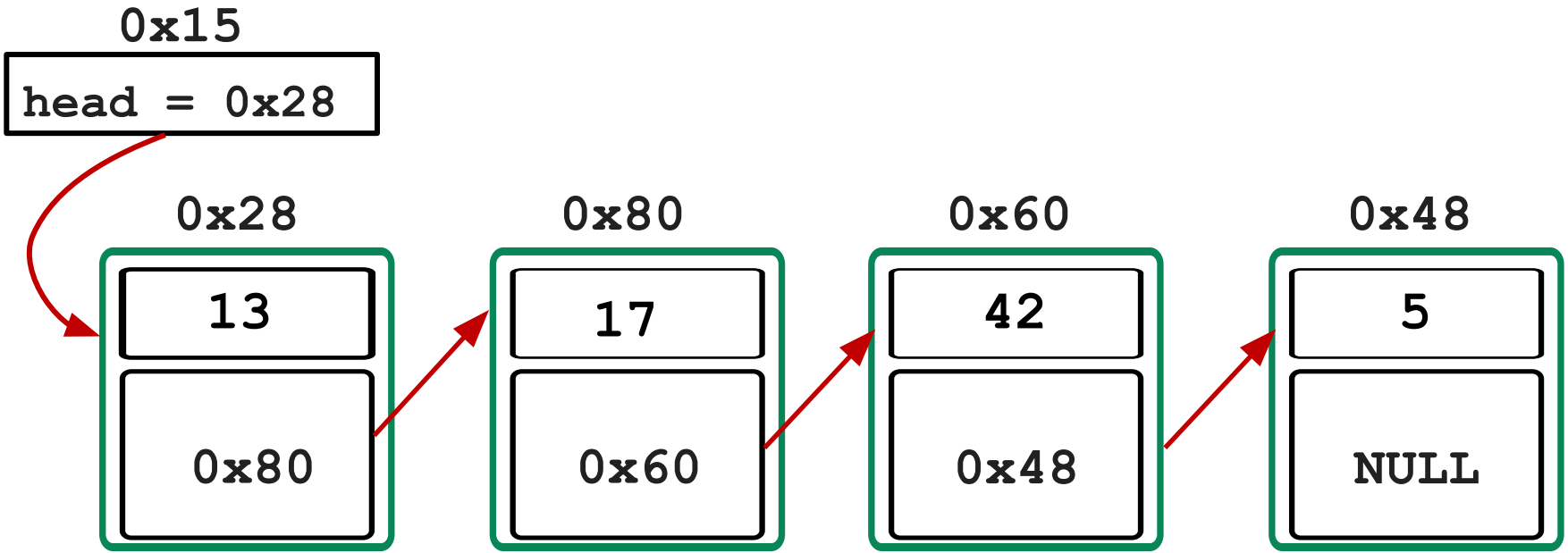
Today's Lecture

- Recap:
  - Traversing
  - Exercise: Search for a value
  - Exercise: Insert node at position
- Linked list deletion
  - First node
  - All nodes
  - Search and delete

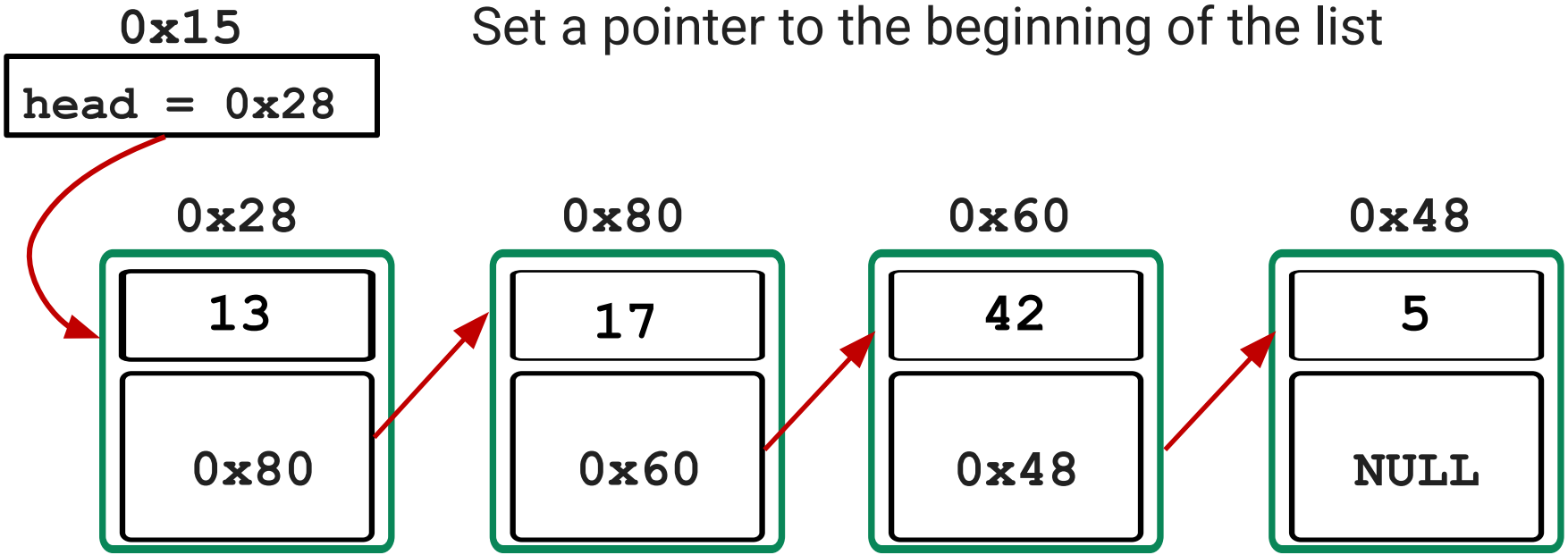
Linked List Recap

- Recap:
  - Traversal
  - Exercise: search for a value
  - Exercise: Insert at position

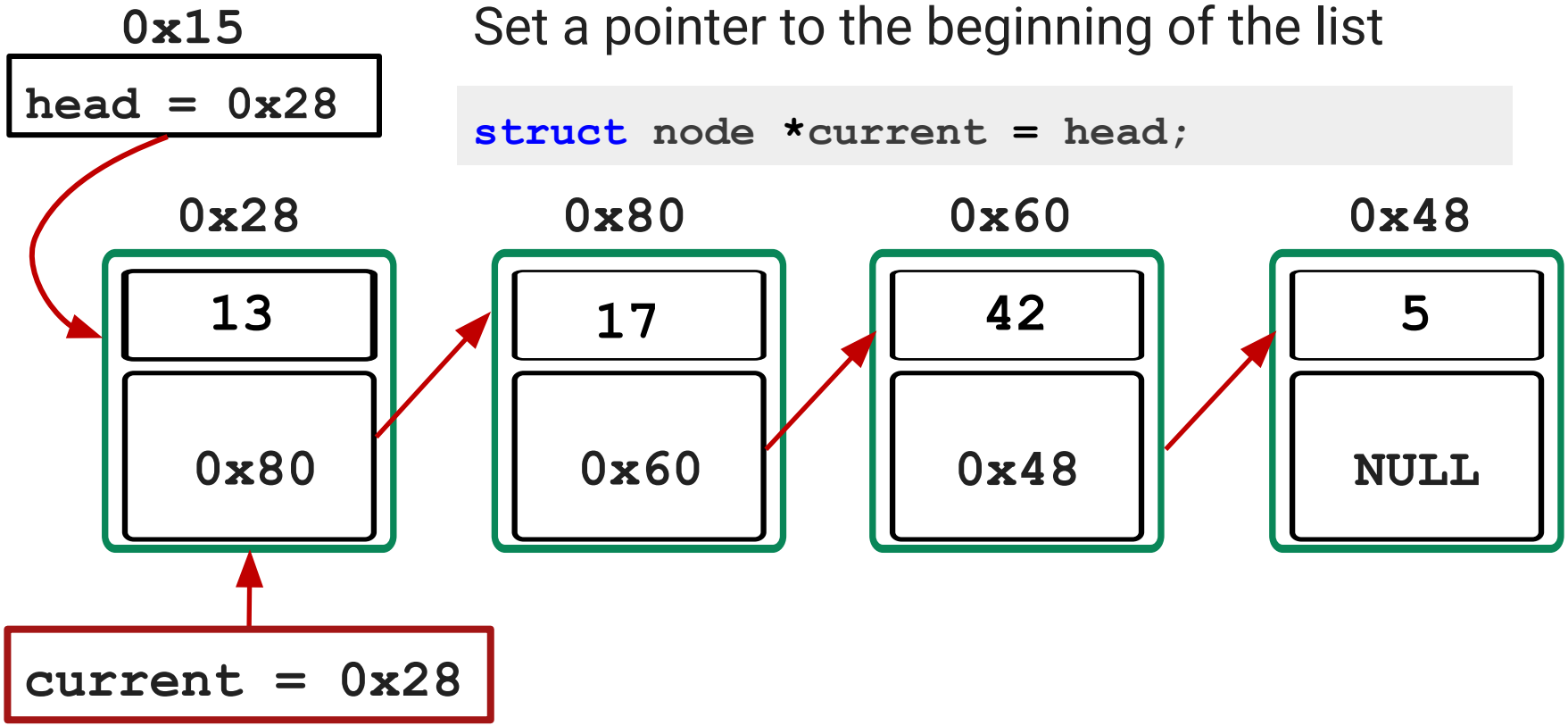
Traversing a List



Traversing a List



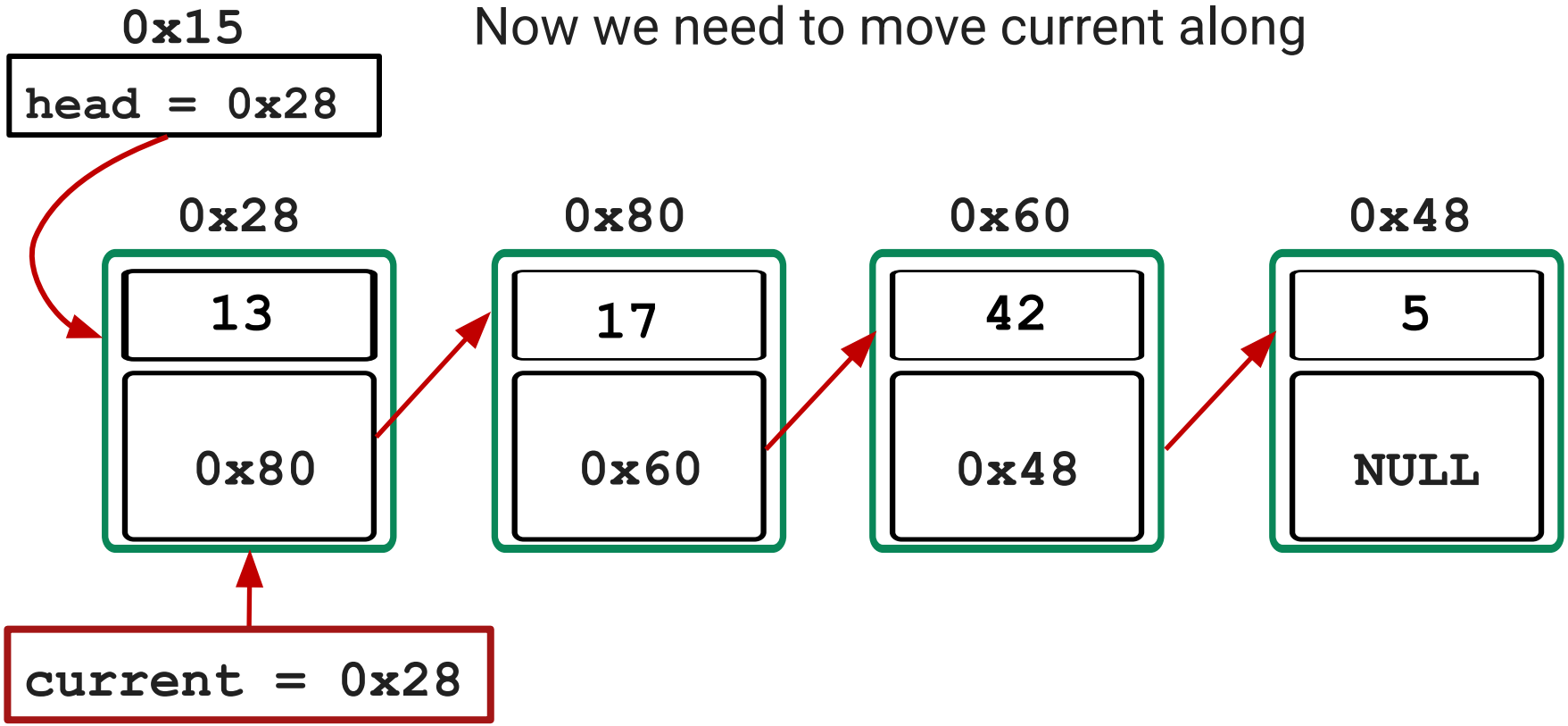
Traversing a List



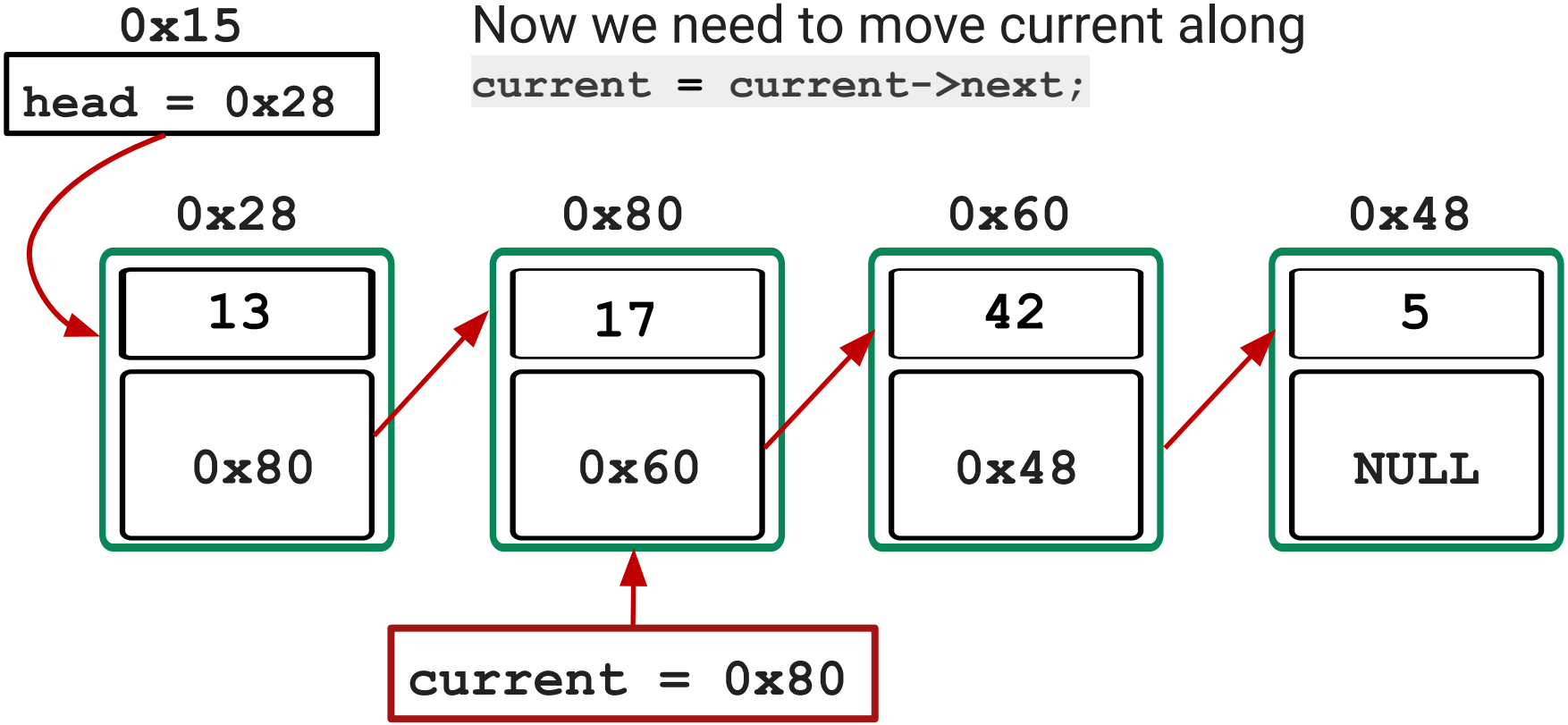
Set a pointer to the beginning of the list

```
struct node *current = head;
```

Traversing a List



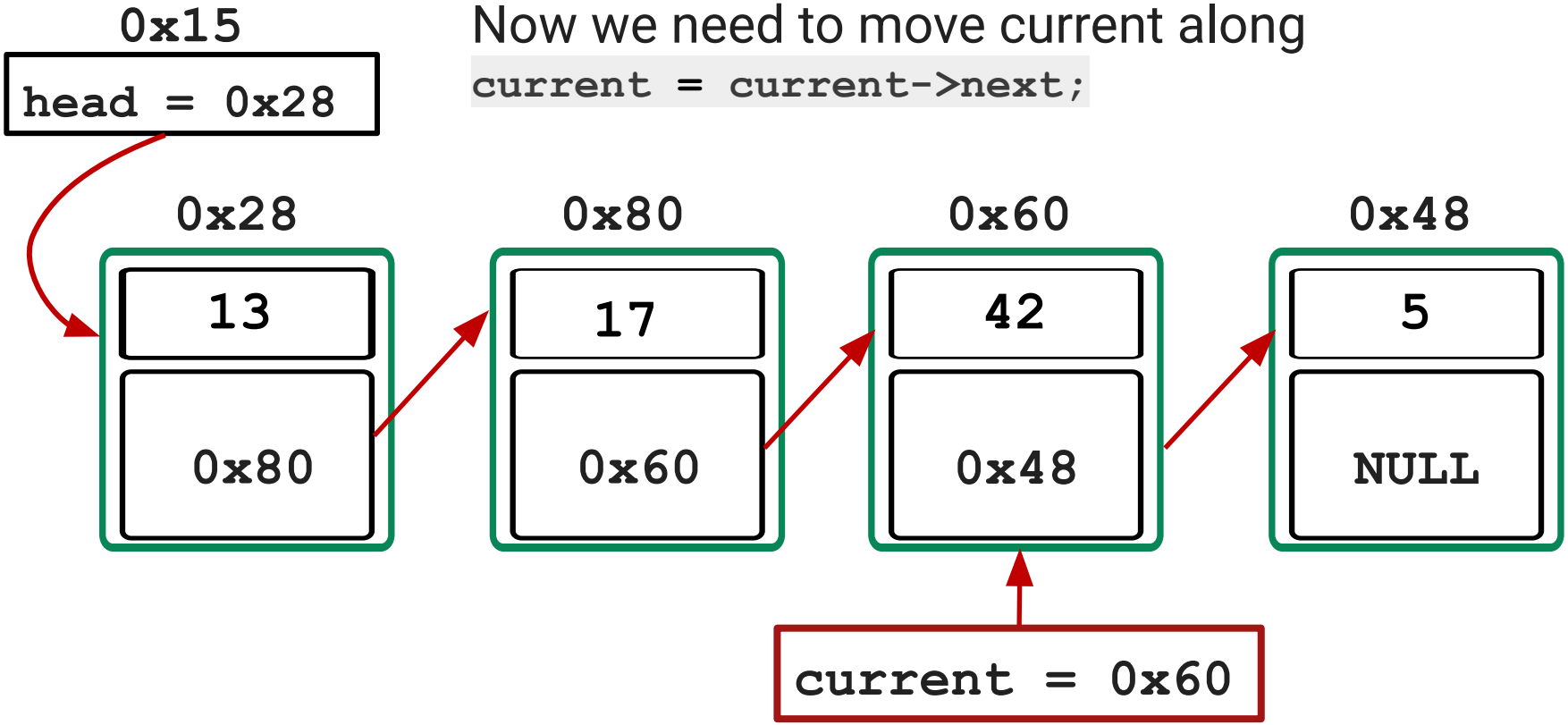
Traversing a List



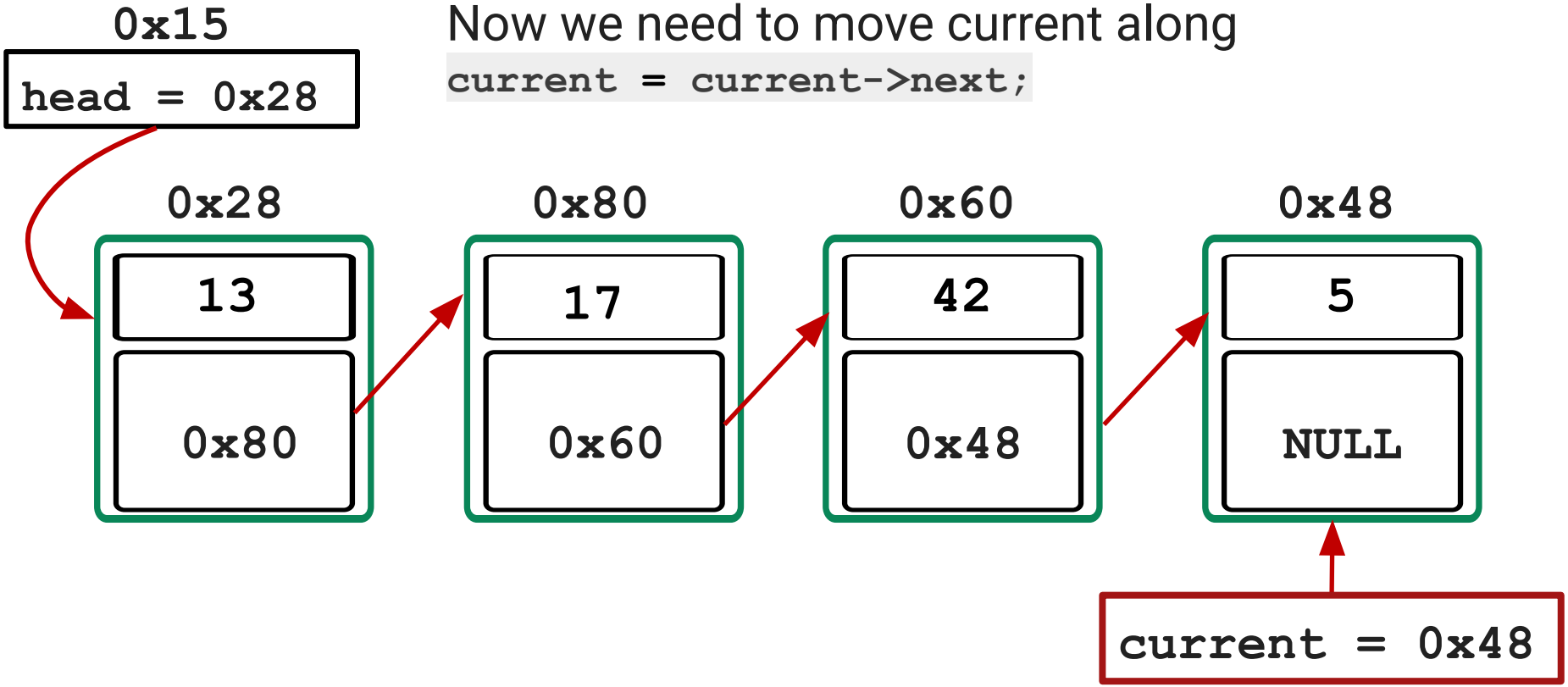
Traversing a List

Now we need to move current along

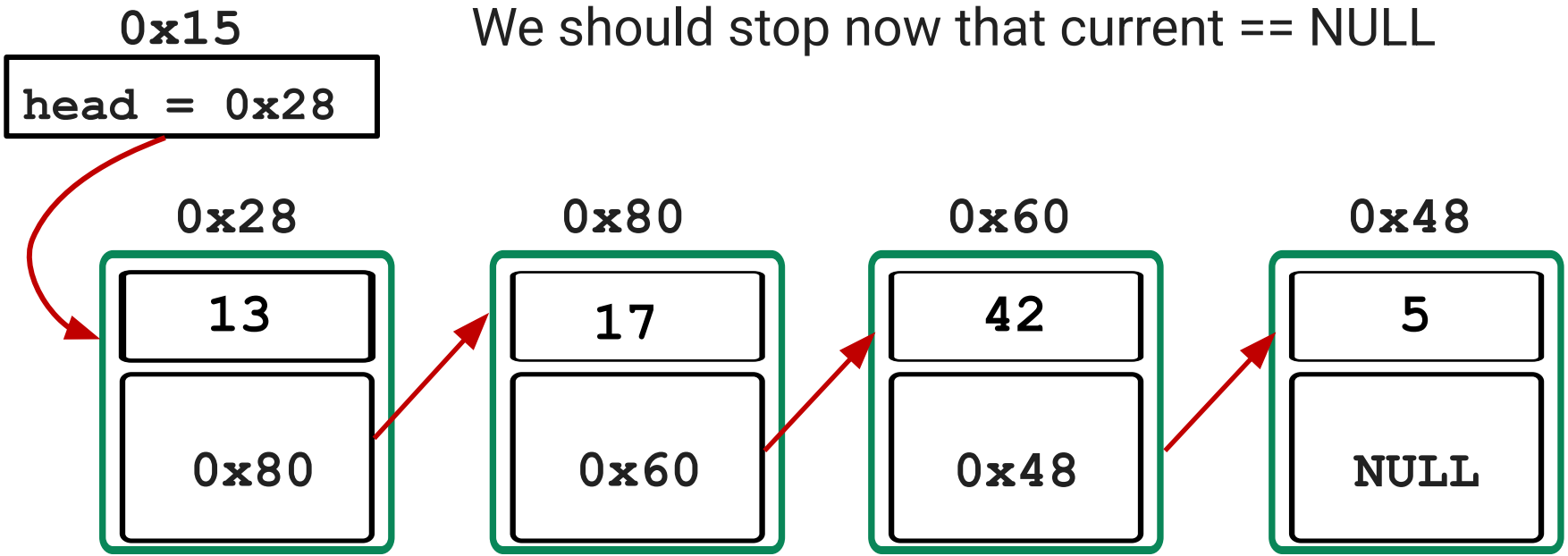
```
current = current->next;
```



Traversing a List



Traversing a List



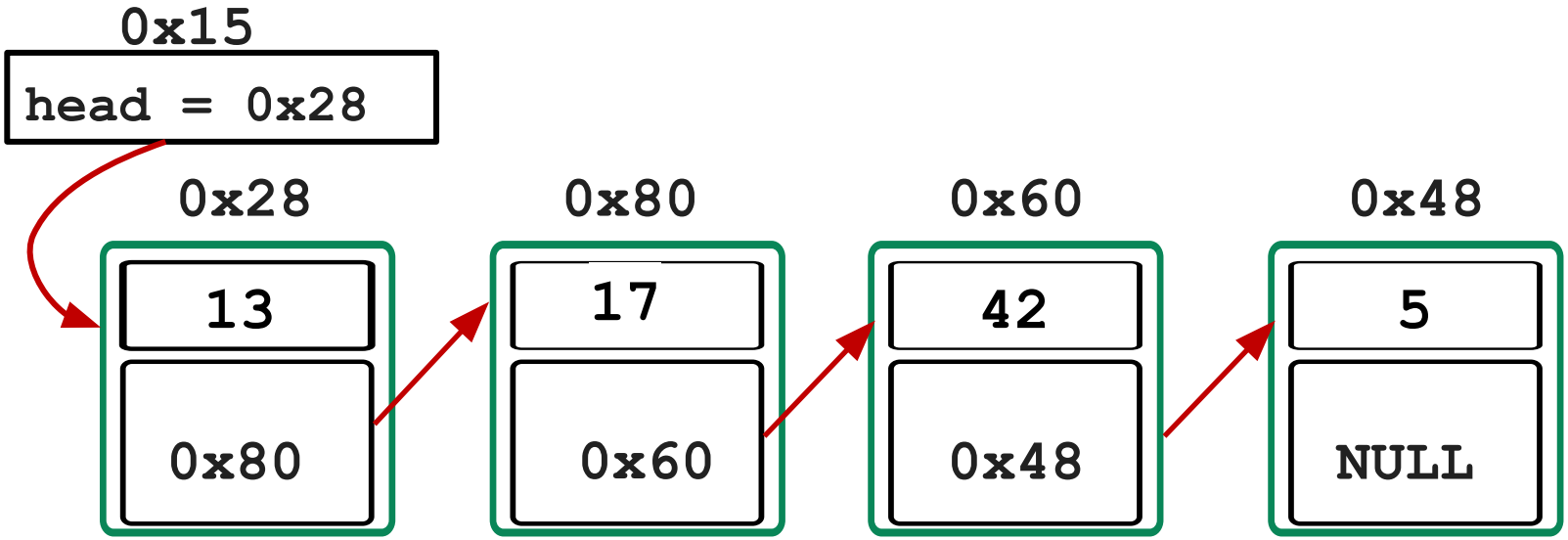
Exercise

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

What cases should we make sure we test?

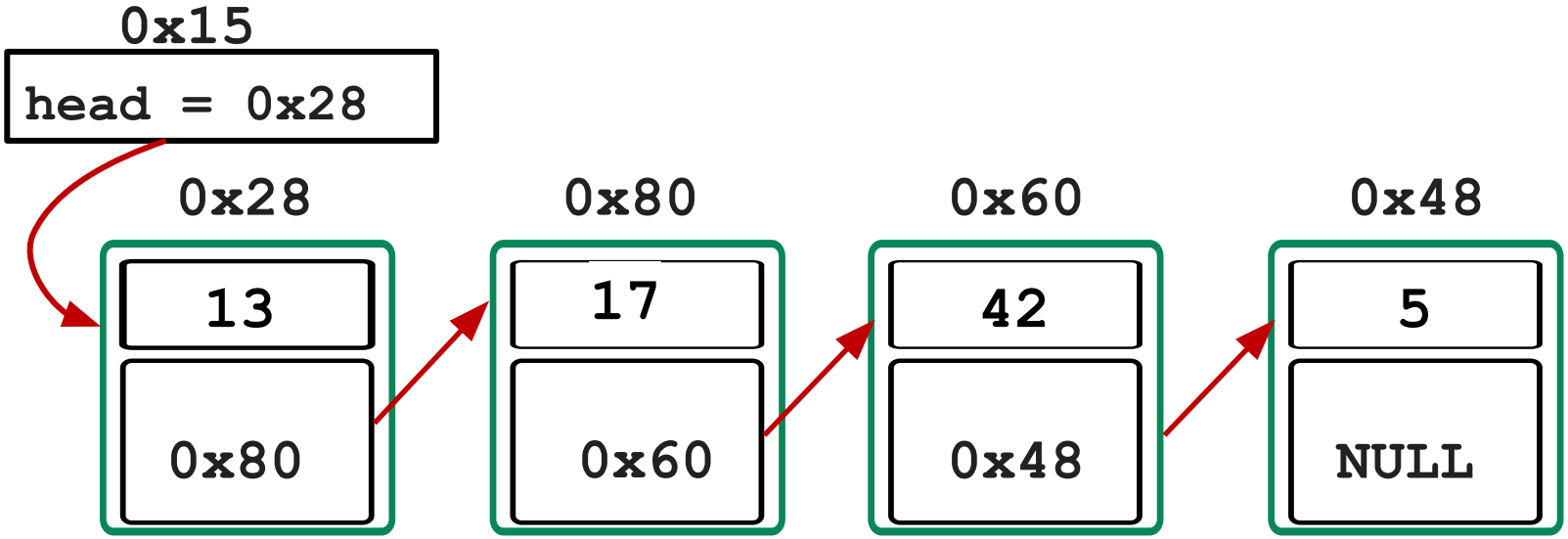
Inserting at a given position

We want to insert a new node at position 2, assuming positions start at 0.



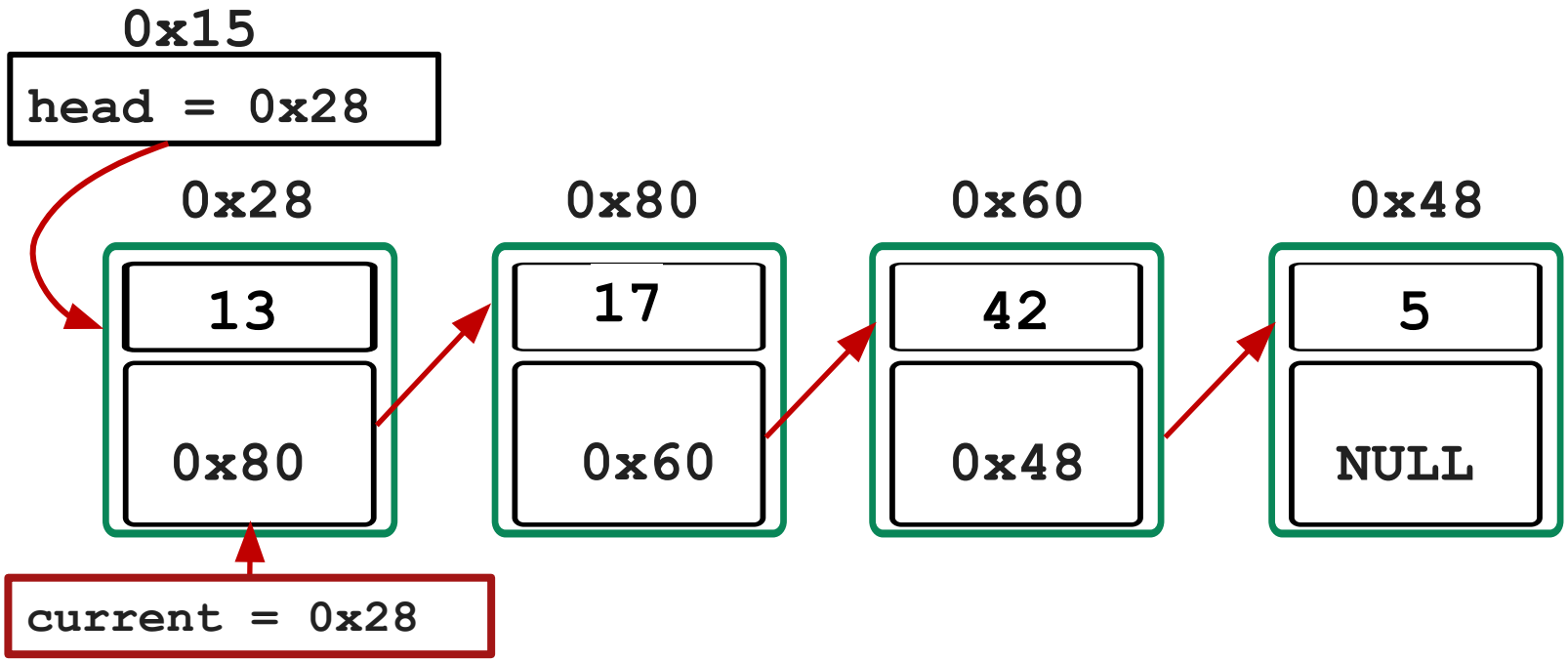
Inserting node at position

Use a counter and stop traversing when we get to the node **before** the position we want to insert at (position - 1). In this case position 1.



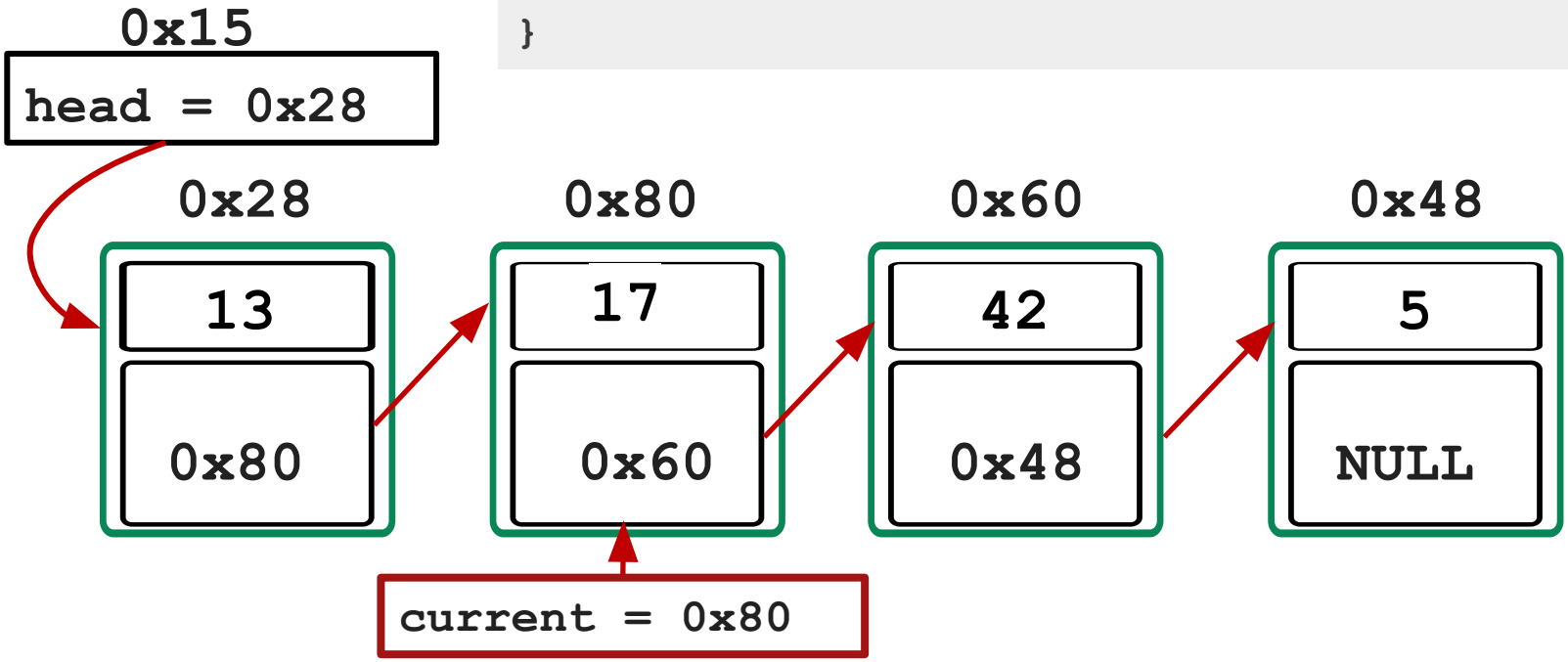
Inserting at Position

```
struct node *current = head;
int counter = 0;
```



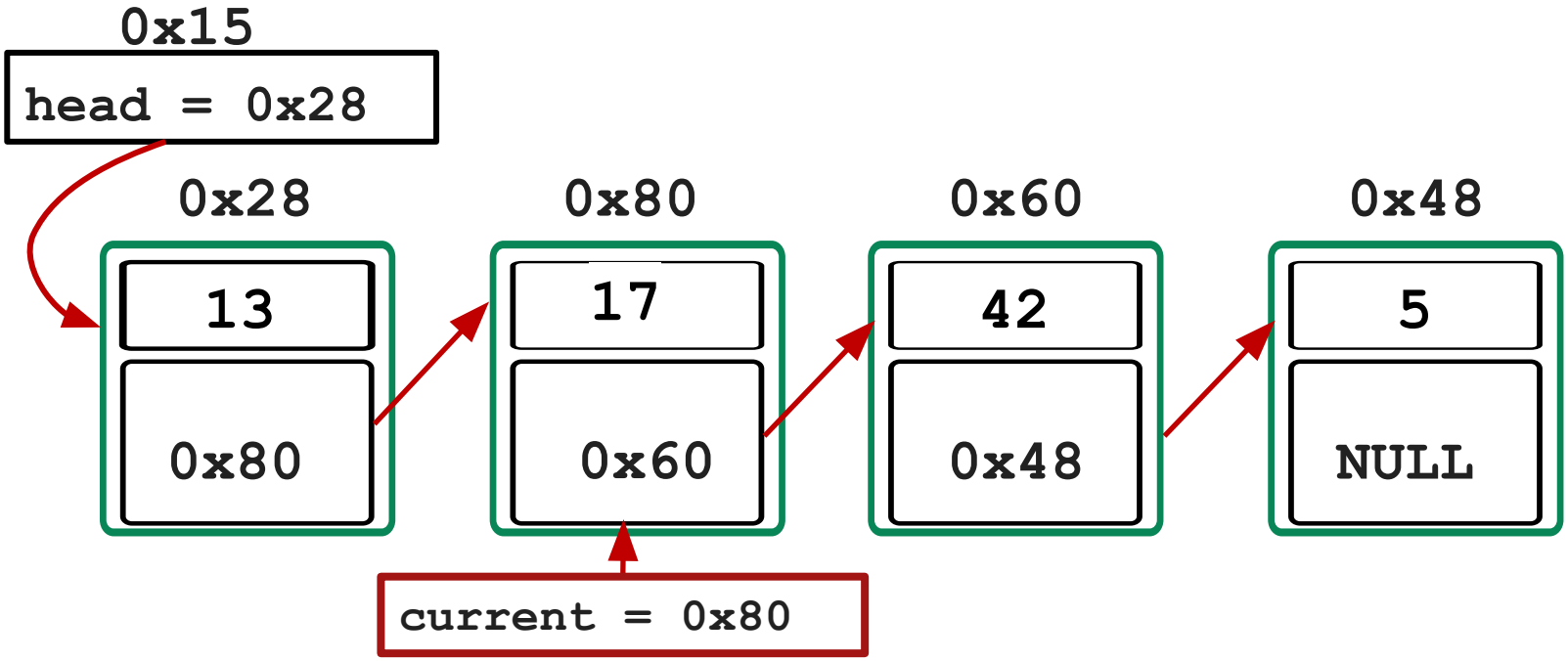
Inserting at position

```
while (counter < position - 1) {
    current = current->next;
    counter++;
}
```

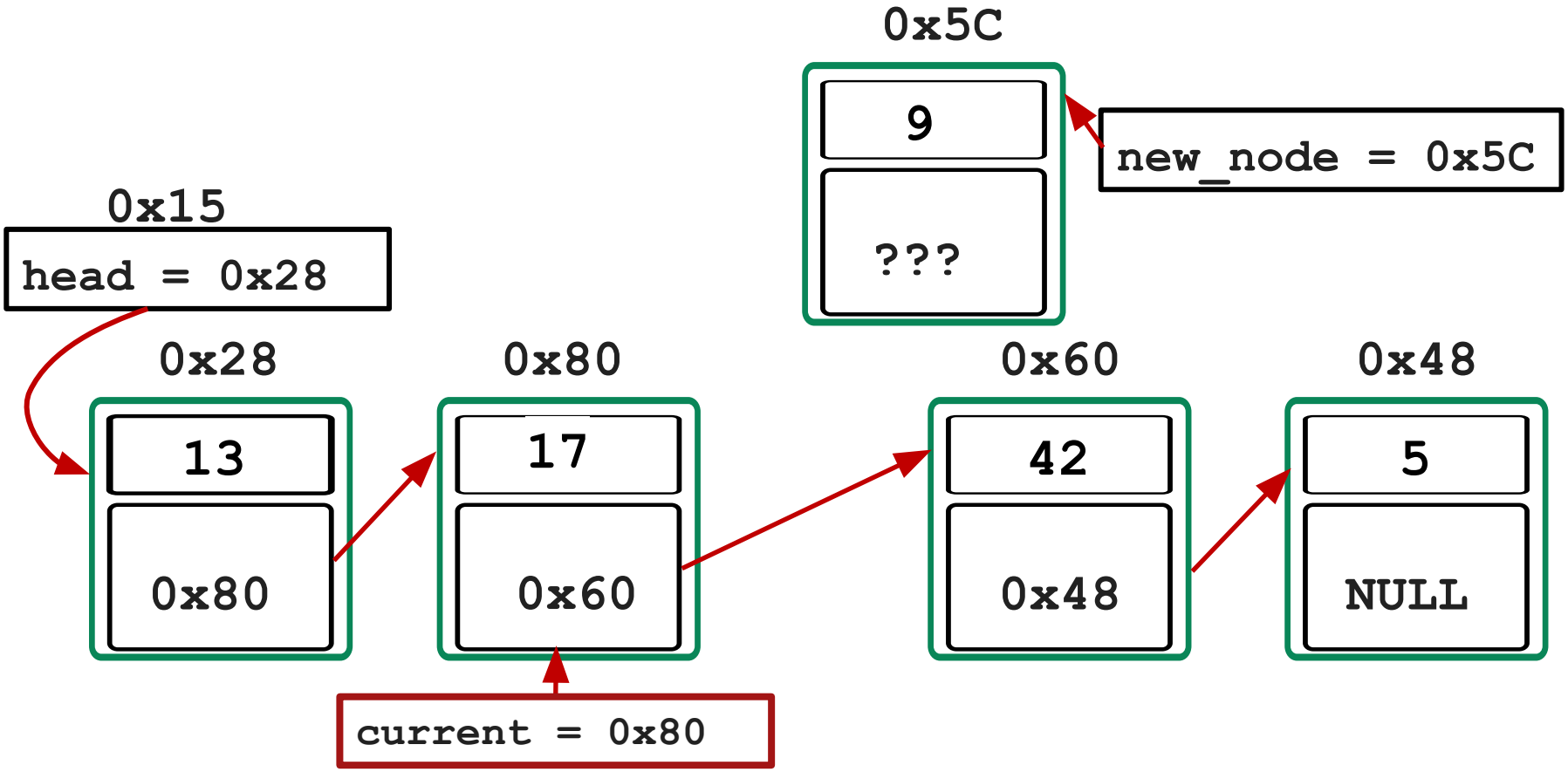


Inserting at Position

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

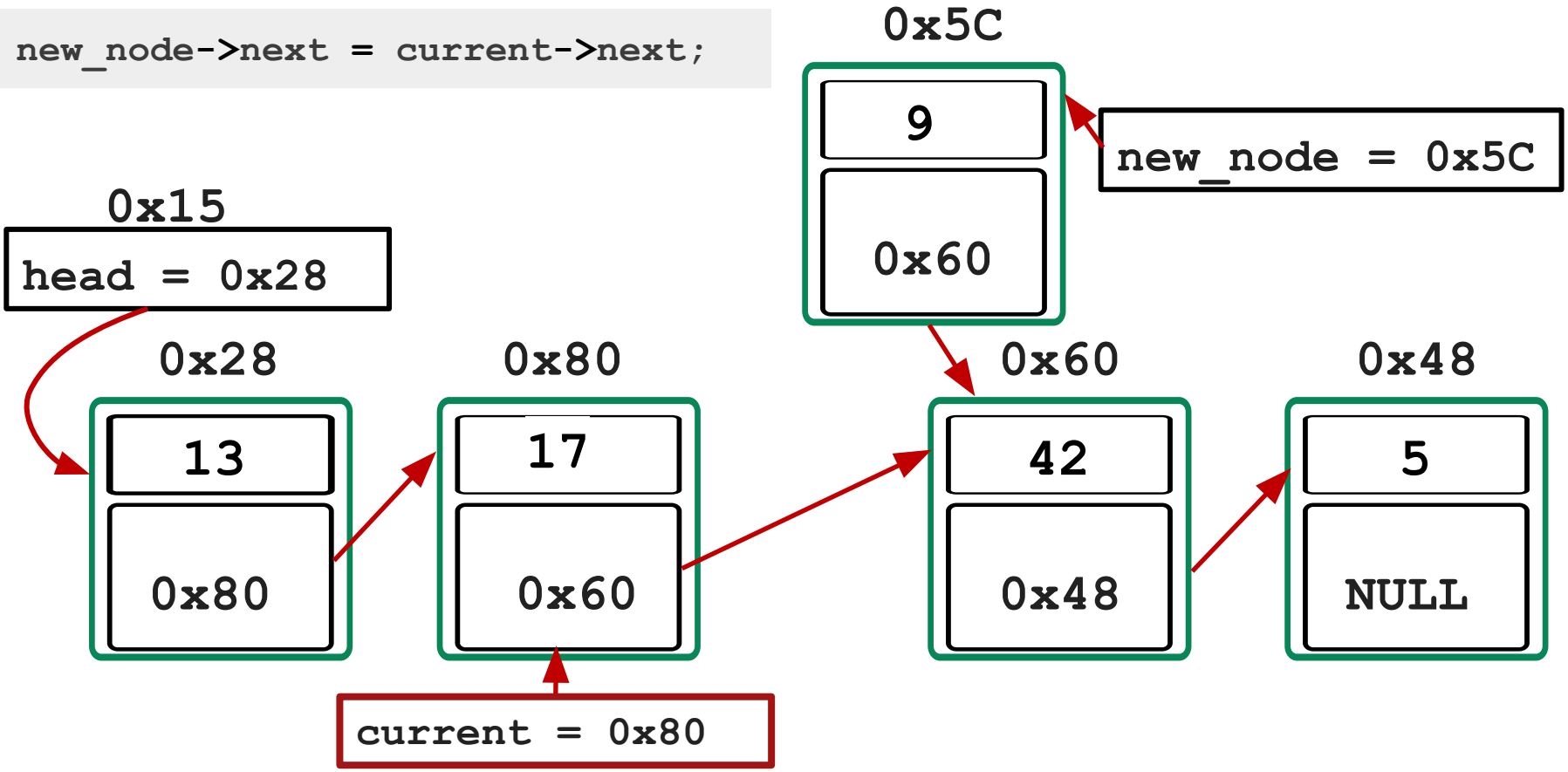


Inserting at Position



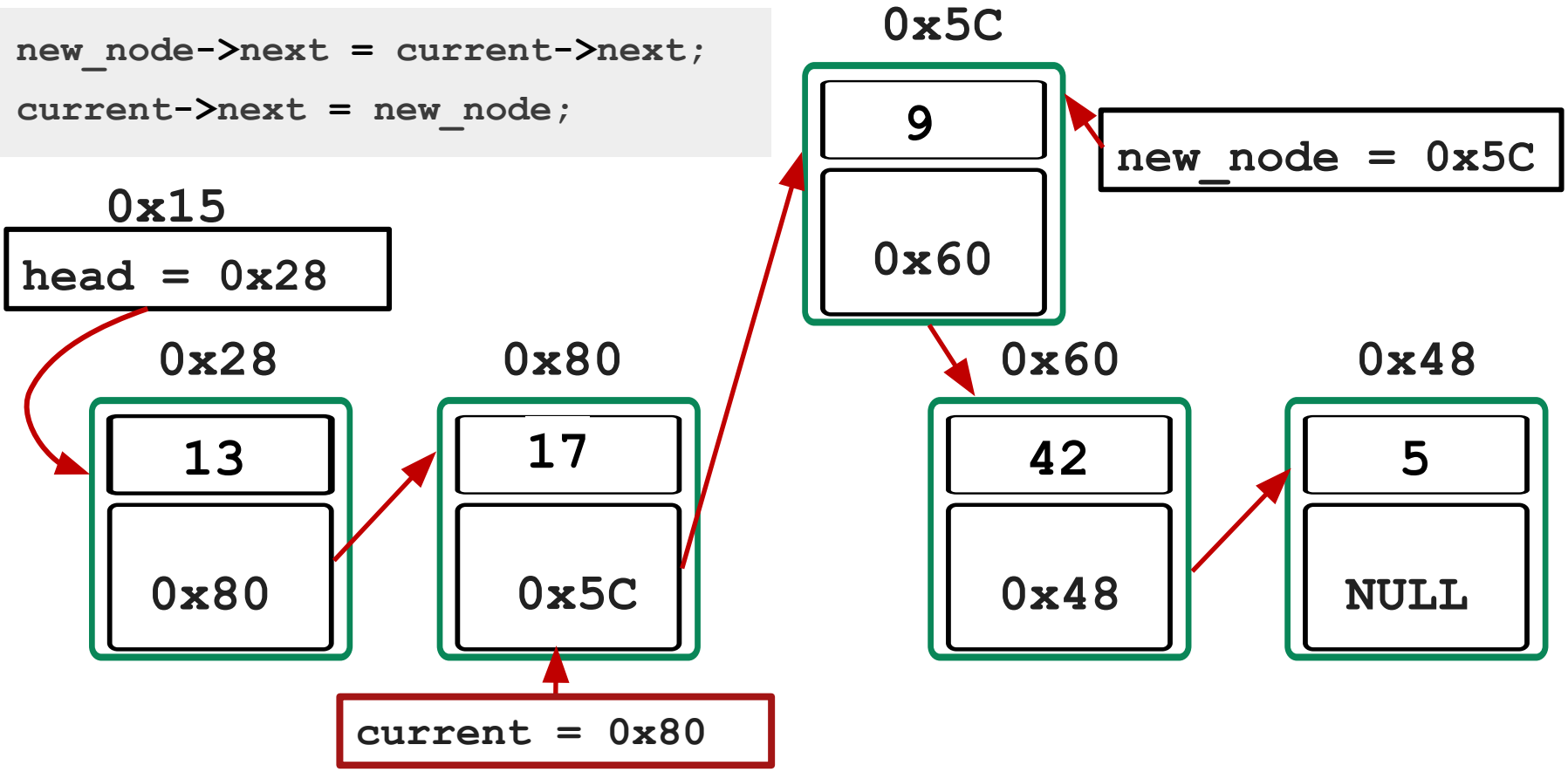
Inserting at Position

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



Inserting at Position

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



Coding: Inserting at Position

- What conditions will break this?
  - What happens if it is an empty list?
  - What happens if there is only 1 item in the list?
  - Anything else we should check?
- How can we modify our code to handle any of these situations that break it?

Deletion

Deleting the First Node in a Linked List

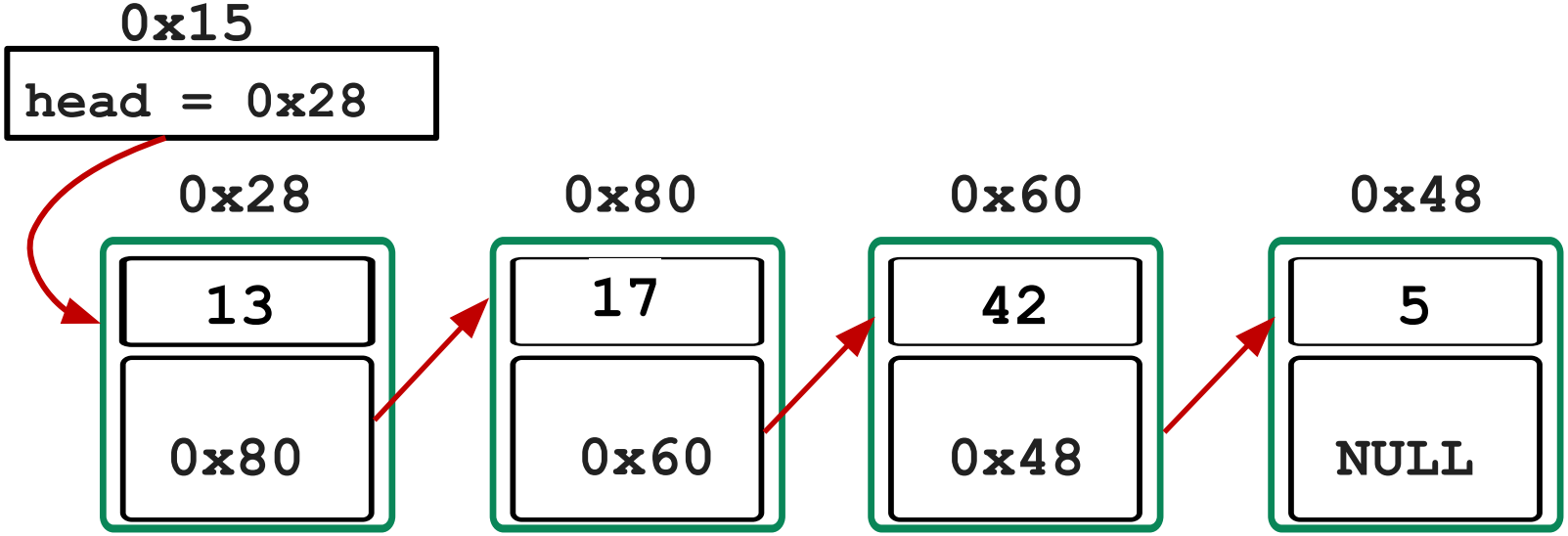
Let's write a function to delete the first node in a linked list.
We need to consider the case when the list is empty

- 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;
}
```

Deleting the First Node in a Linked List

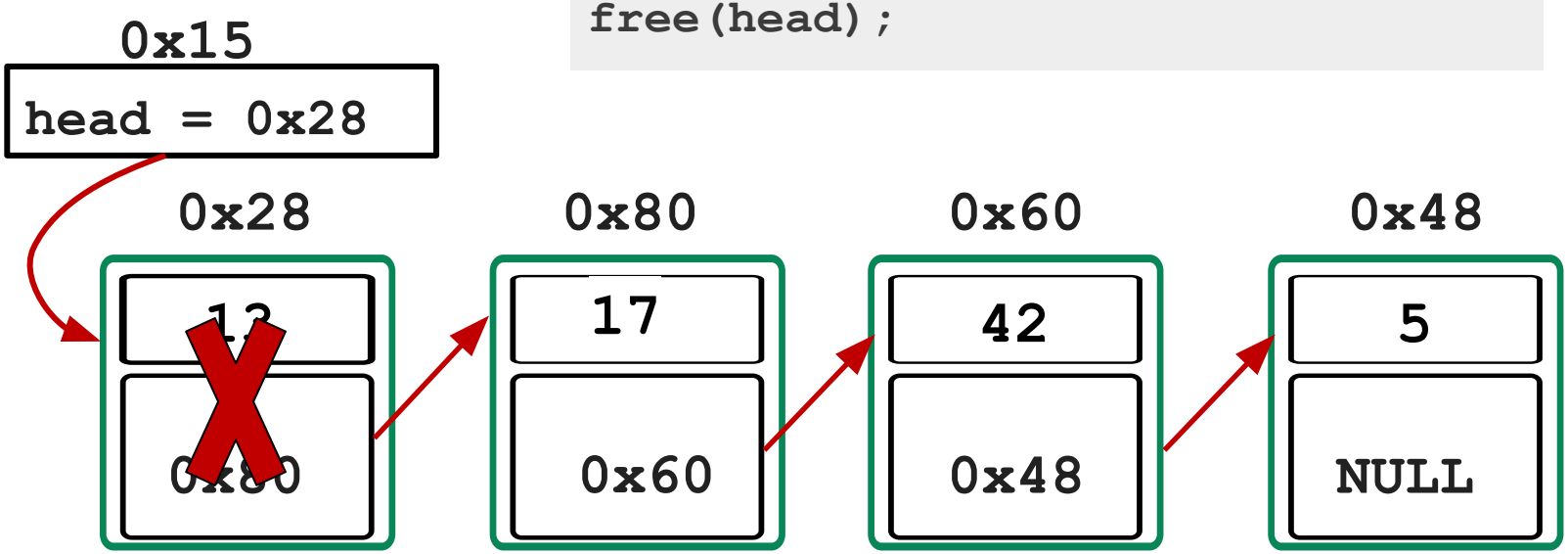
If our list is not empty, we want to make the second node the new head of the list and free the first node that we want to delete.



Deleting the First Node in a Linked List

What would be the problem calling free on head first?

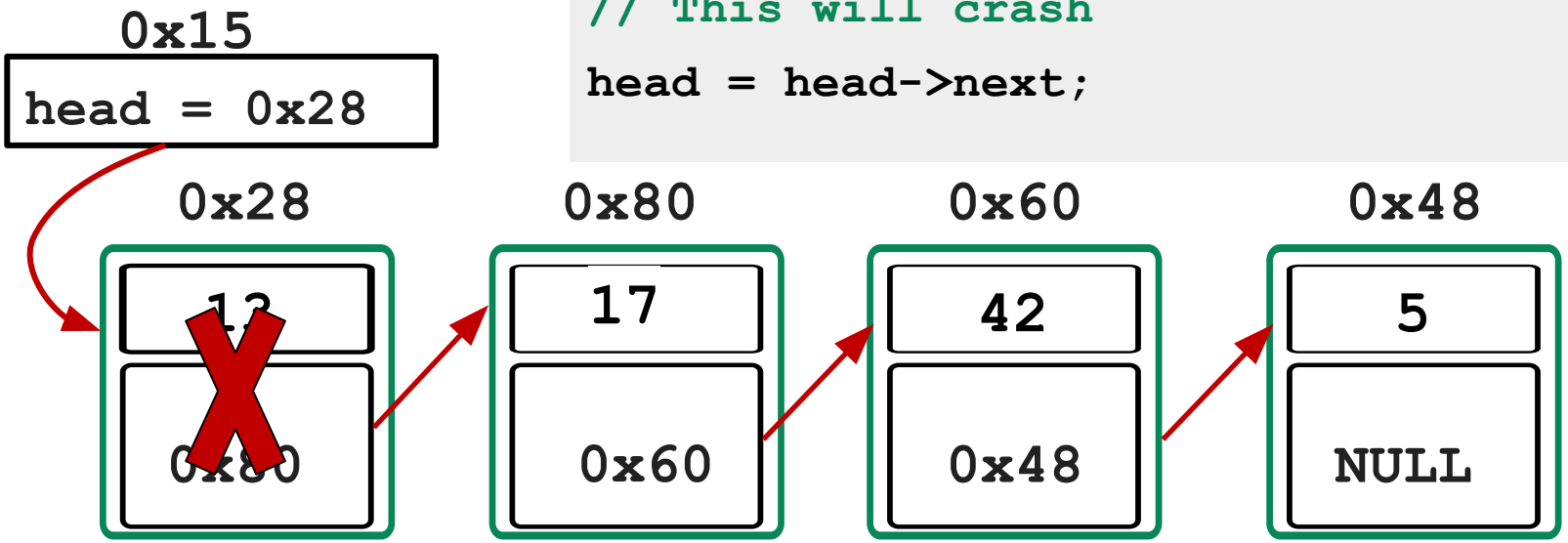
```
free(head);
```



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

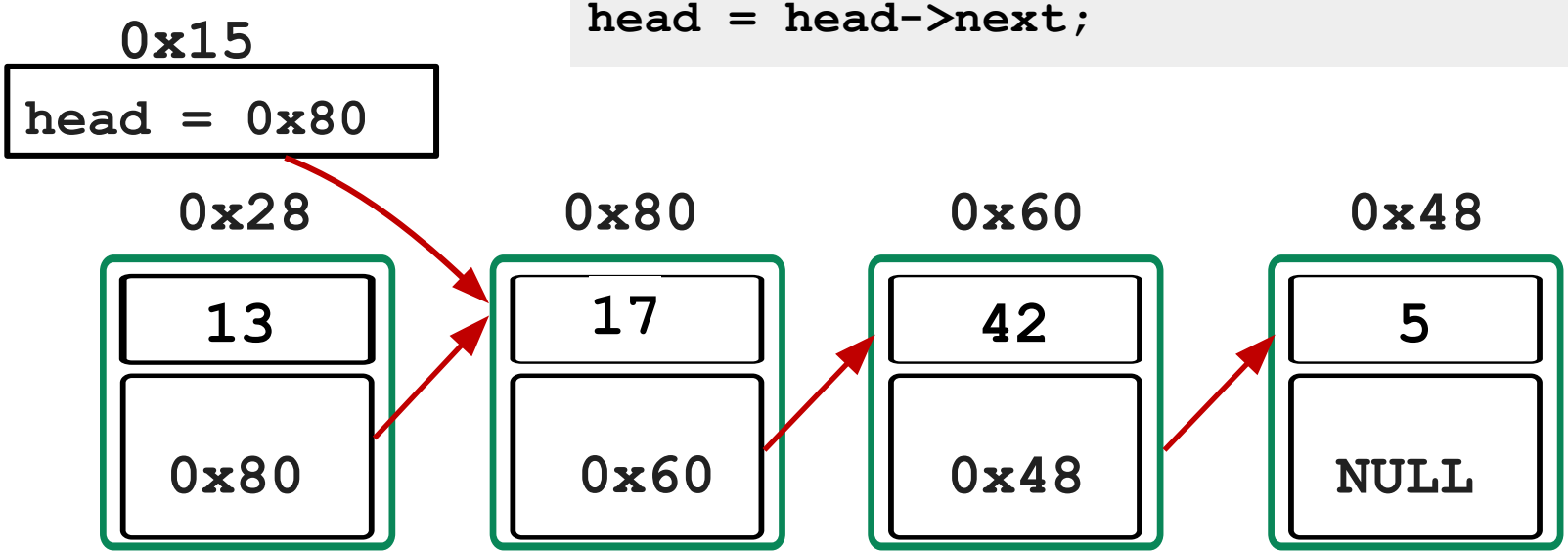
```
// This will crash
head = head->next;
```



Deleting the First Node in a Linked List

What would be the problem with updating head first?

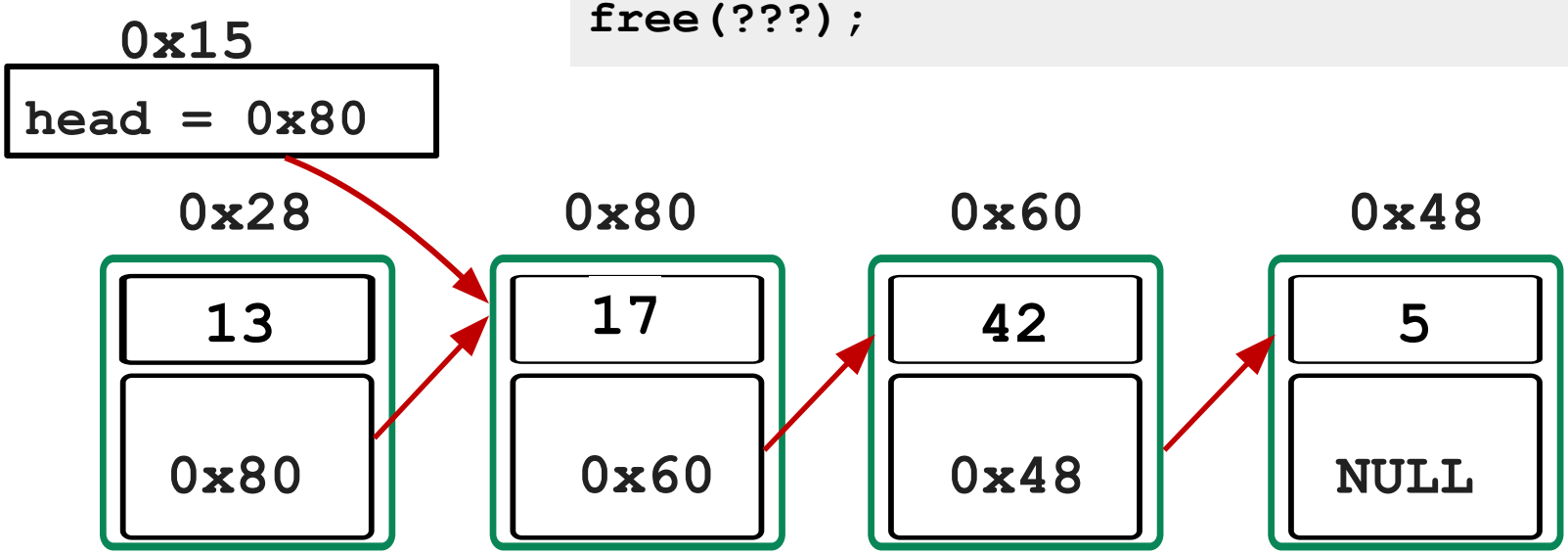
```
head = head->next;
```



Deleting the First Node in a Linked List

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

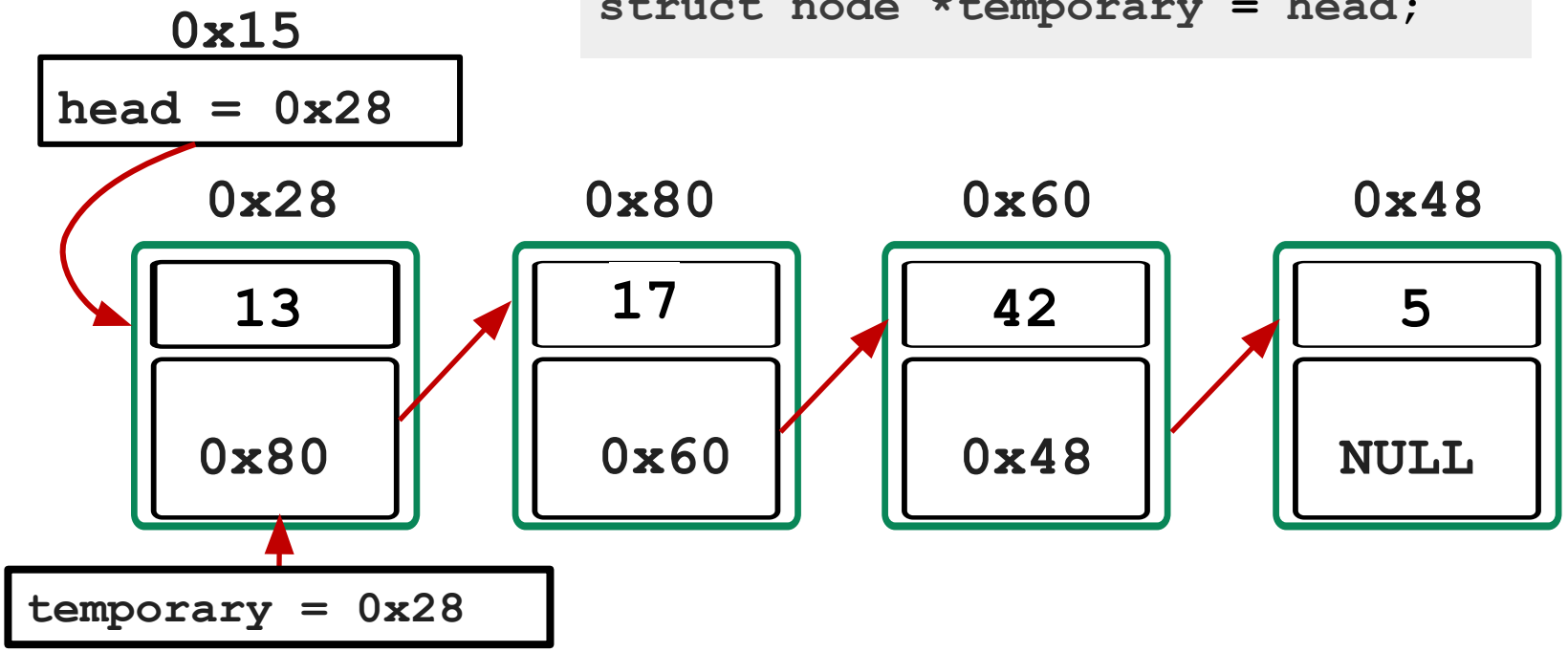
```
free(???);
```



Deleting the First Node in a Linked List

Let's create a pointer to the first node

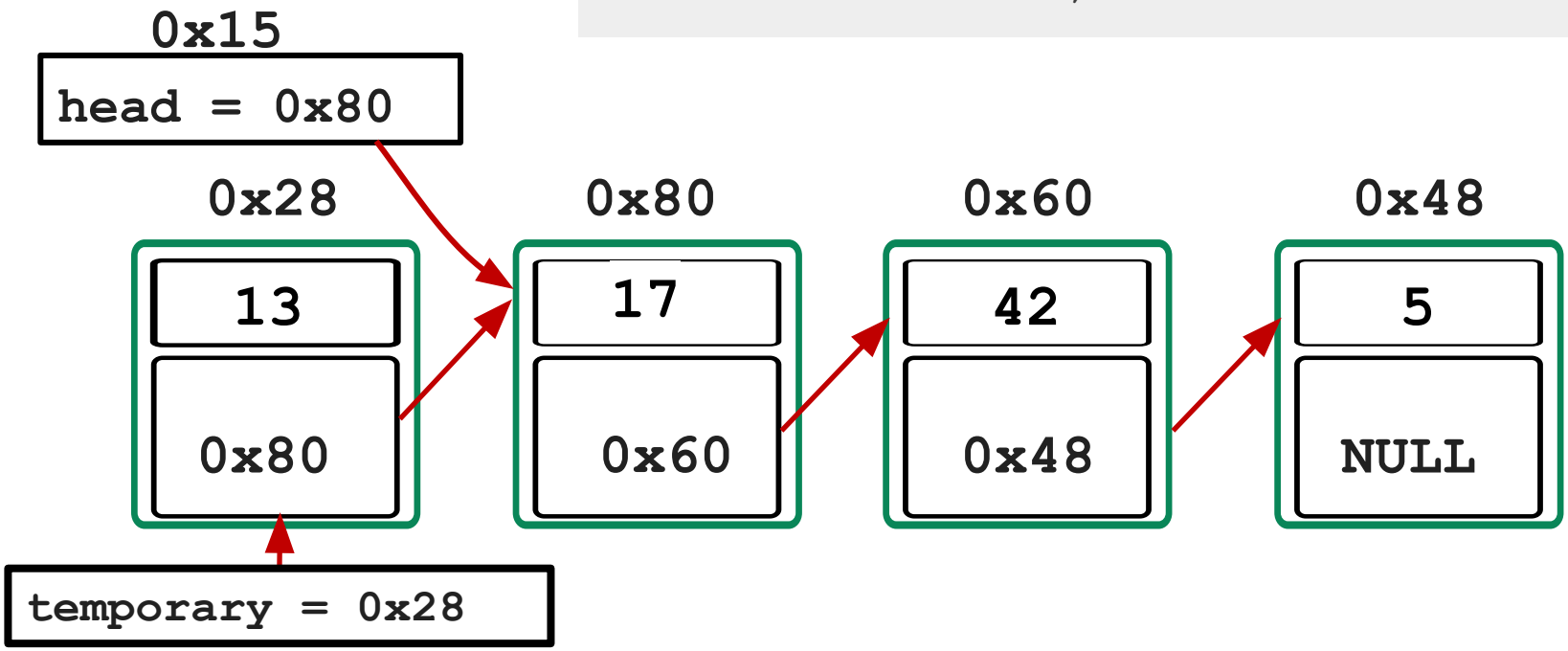
```
struct node *temporary = head;
```



Deleting the First Node in a Linked List

Now we can update head

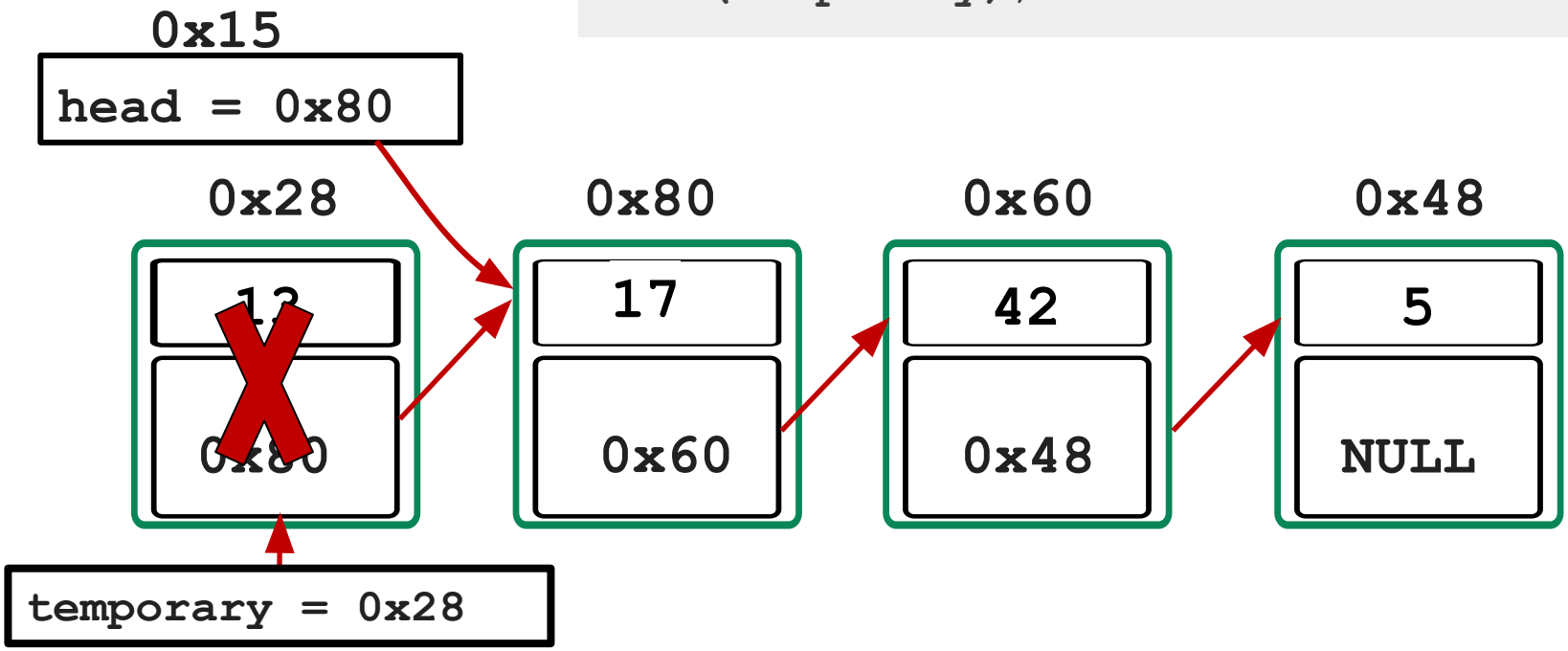
```
head = head->next;
```



Deleting the First Node in a Linked List

Now we can free the first node

```
free(temporary);
```



Deleting the First Node from a List

```
struct node *delete_first_node(struct node *head) {
    if (head == NULL) {
        return head;
    }
    struct node *temporary = head;
    head = head->next;
    free(temporary);
    return head;
}
```

Delete All Nodes the wrong way

What is wrong with this code?

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

Delete All Nodes the wrong way

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

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

Delete All Nodes the Correct Way

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

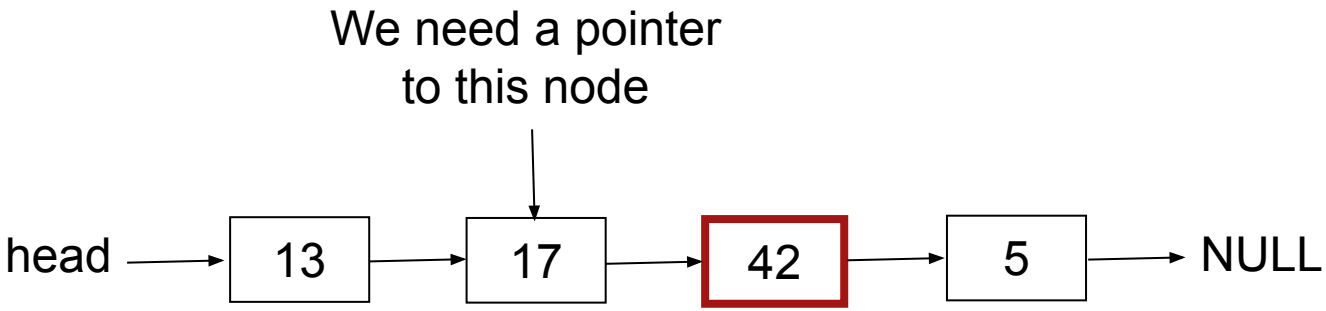
```
// Delete all nodes from a given list
void delete_all_nodes(struct node *head) {
    struct node *current = head;
    while (current != NULL) {
        head = head->next;
        free(current);
        current = head;
    }
}
```

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

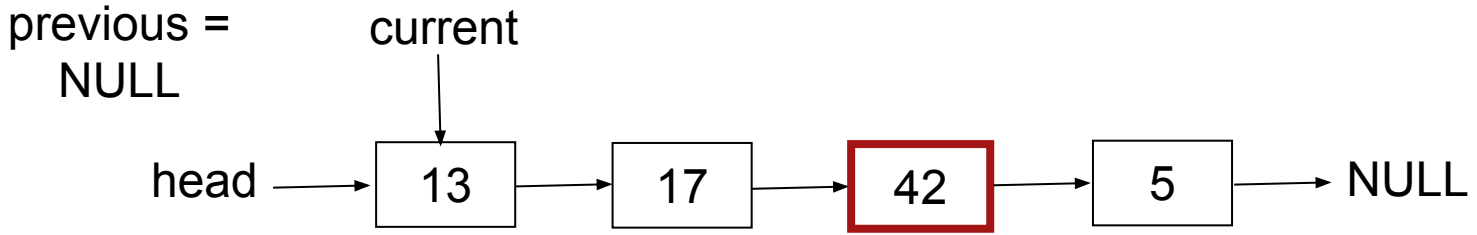
Search and delete: between 2 nodes

- To delete a node we need to link the previous node to the next node
  - If we want to delete the node with 42, we need to find the node before it



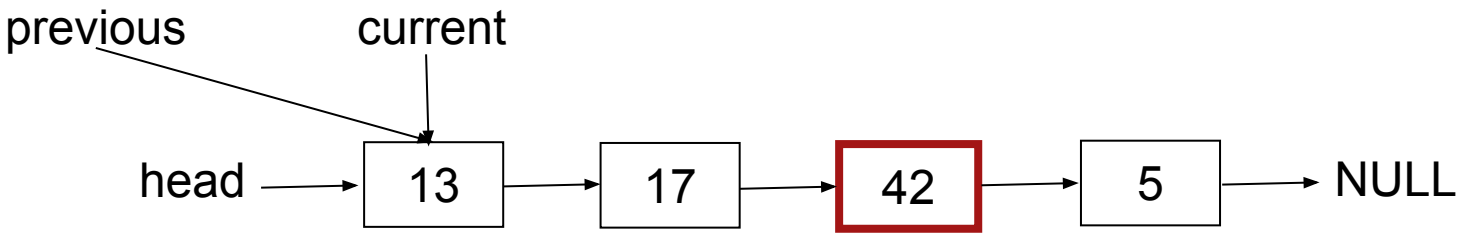
Search and delete: between 2 nodes

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



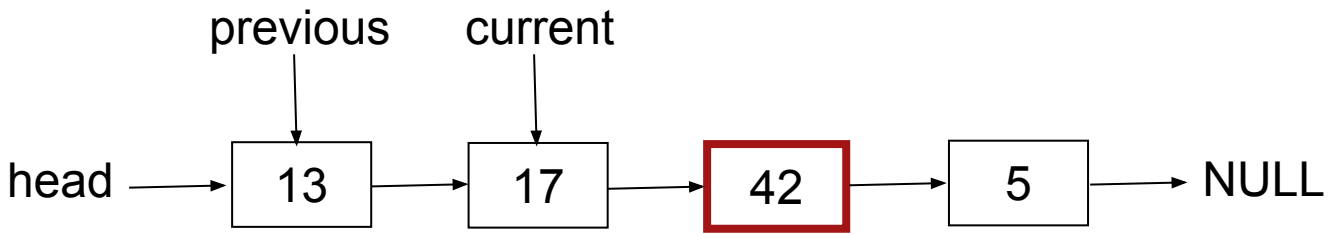
Search and delete: between 2 nodes

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



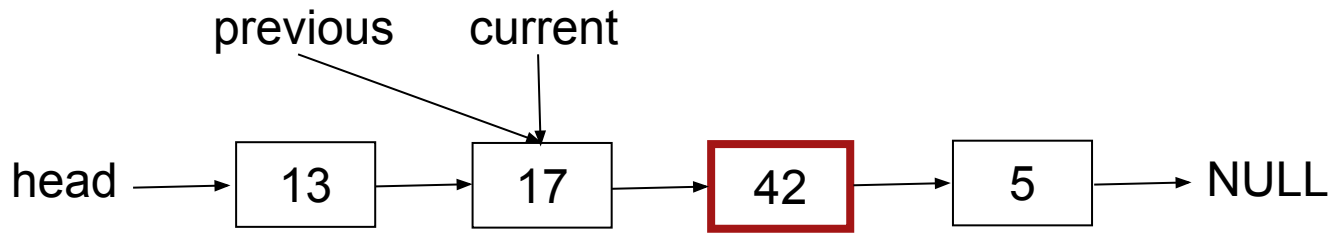
Search and delete: between 2 nodes

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



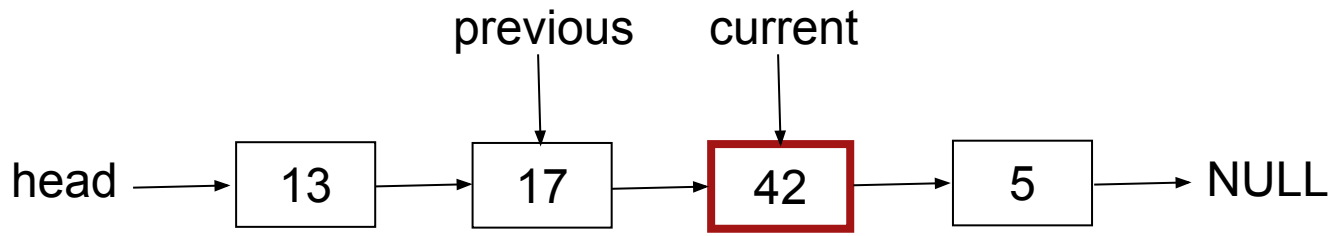
Search and delete: between 2 nodes

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



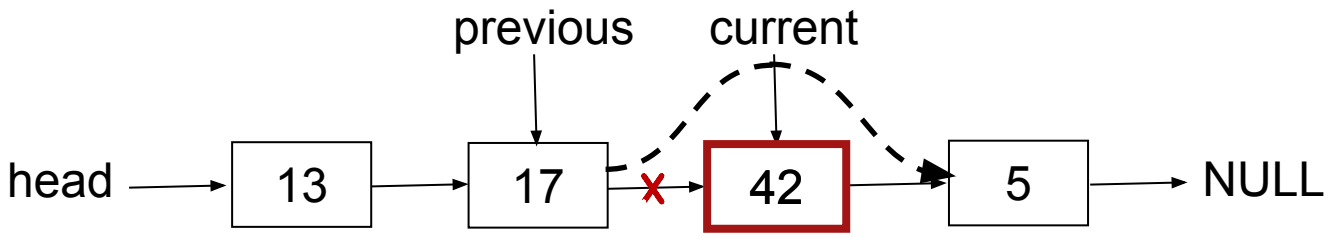
Search and delete: between 2 nodes

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



Search and delete: Approach 1

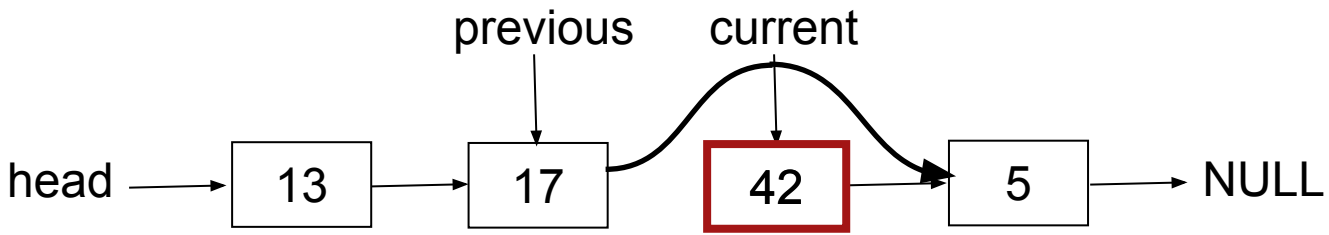
Then we need to connect current node to the one after the one we are deleting.



Search and delete: Approach 1

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

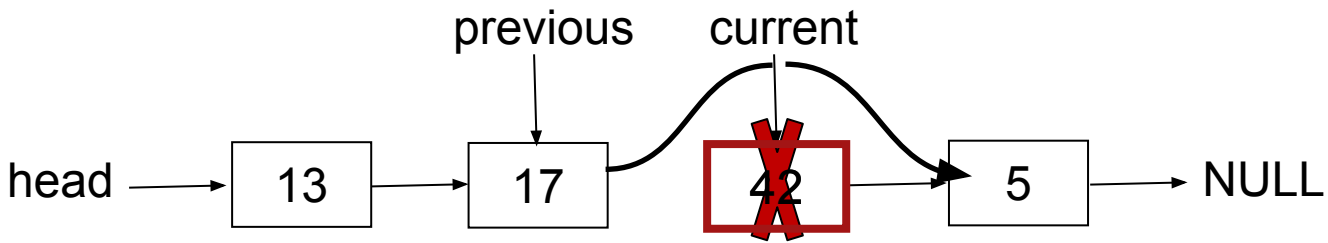
```
previous->next = current->next;
```



Search and delete: Approach 1

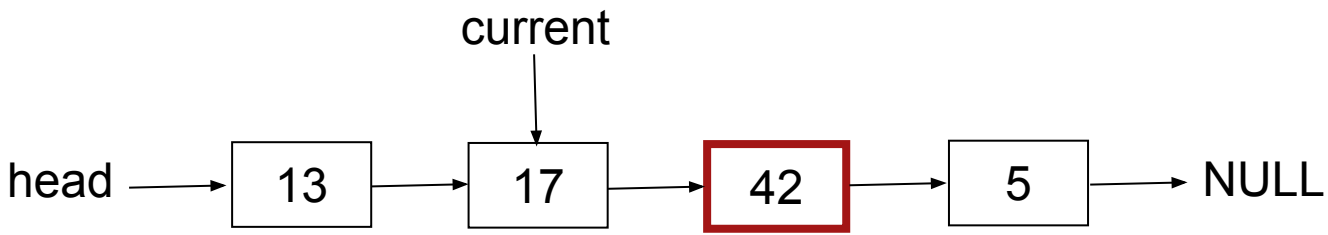
Now we can free the node we want to delete

```
free(current);
```



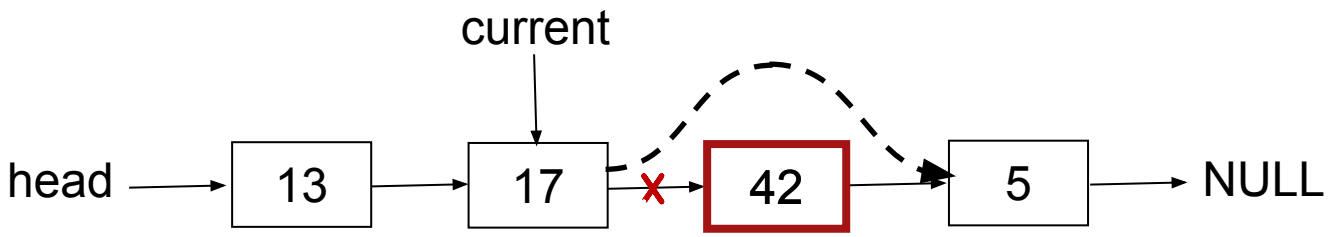
Search and delete Approach 2: general case

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



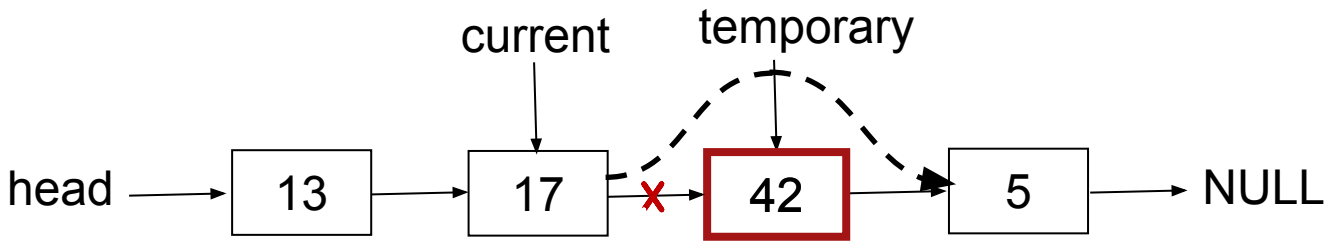
Search and delete: Approach 2

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?



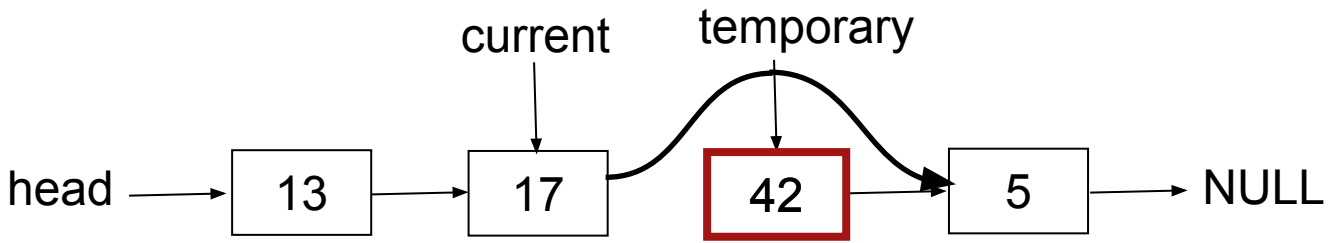
Search and delete: Approach 2

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



Search and delete: Approach 2

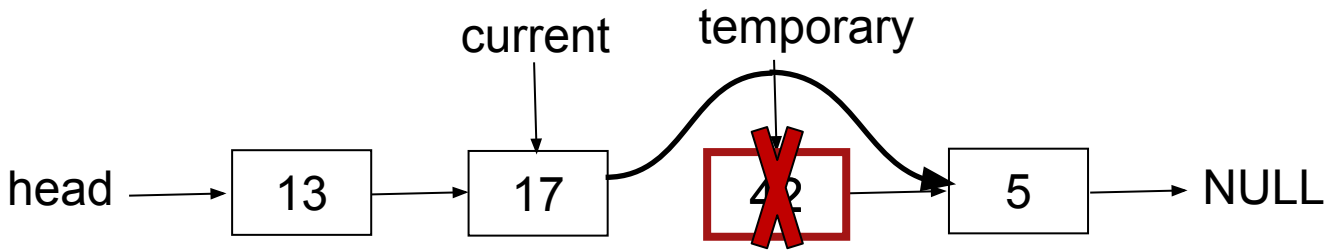
```
struct node *temporary = current->next;
current->next = temporary->next;
```



Search and delete: Approach 2

Now we can free the node we want to delete

```
free(temporary);
```



Coding

Let’s code up both of these approaches.

Let’s extend our first approach to delete all occurrences.

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/TDmCcARMMb

What did we learn today?

- Recap
- Inserting at any position
- Deleting elements
  - First node
  - All nodes
  - Search and delete

Next Lecture

- Linked Lists a Larger Application.
  - Linked Lists as fields in other structs
  - Linked Lists with more complex data (other than just int)
  - Multi-file Linked Lists
  - Helpful for assignment 2

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1511@unsw.edu.au