

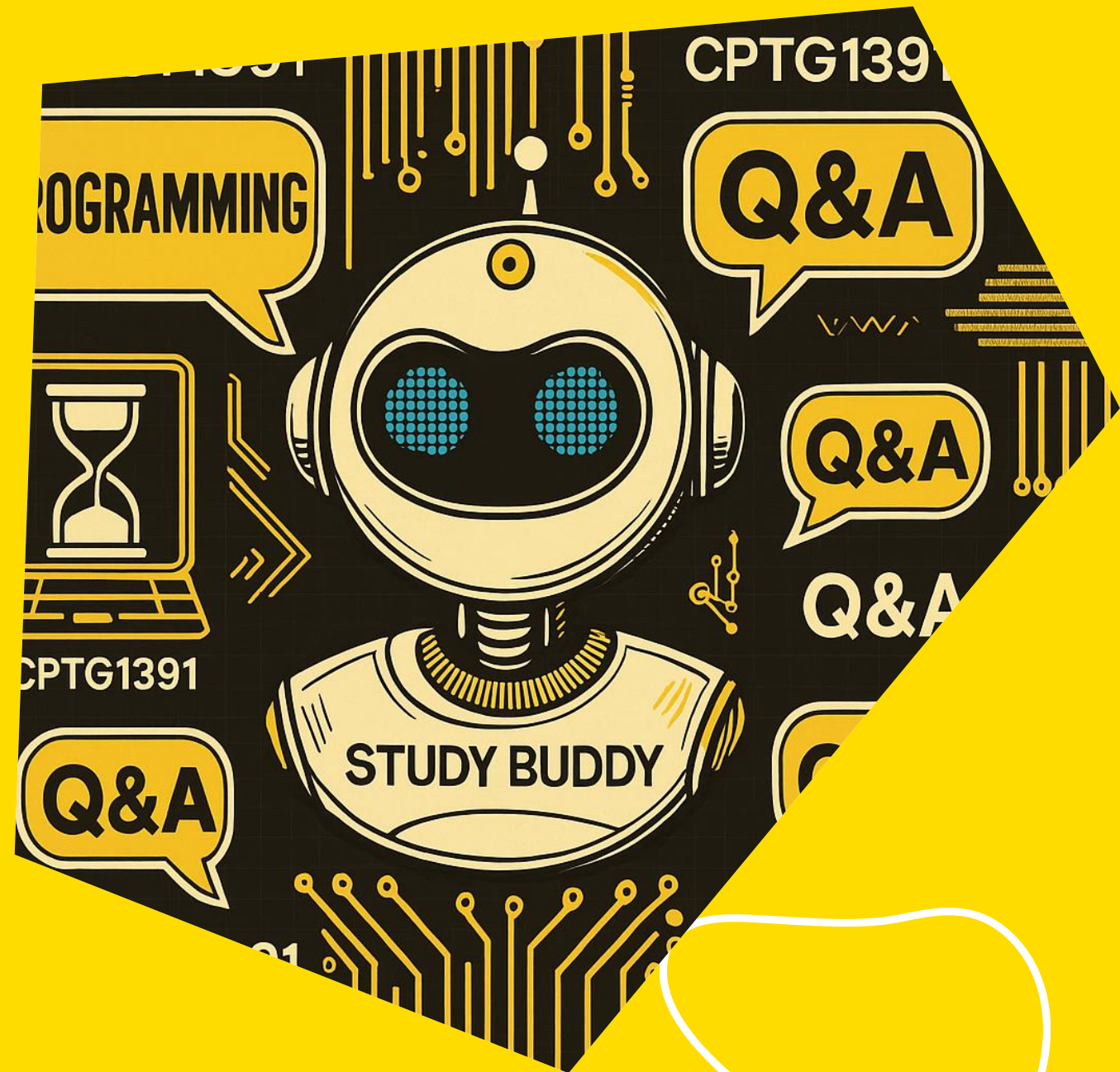
DPST1091 / CPTG1391
Introduction to Programming
Week 10 Lecture 2

Lecturer and Course Convener:

Dr Pantea Aria

Linked Lists Functions

(part 2)



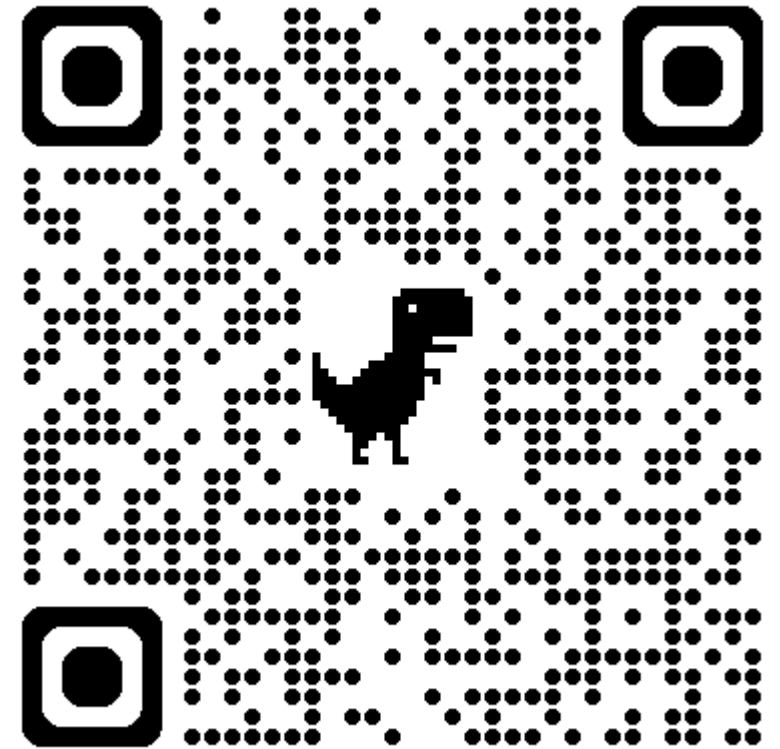
Agenda

- **Today**
 - **Linked Lists Functions –Delete**

Link to the Live Lecture Code

<https://cgi.cse.unsw.edu.au/~dp1091/26T2/code/10>

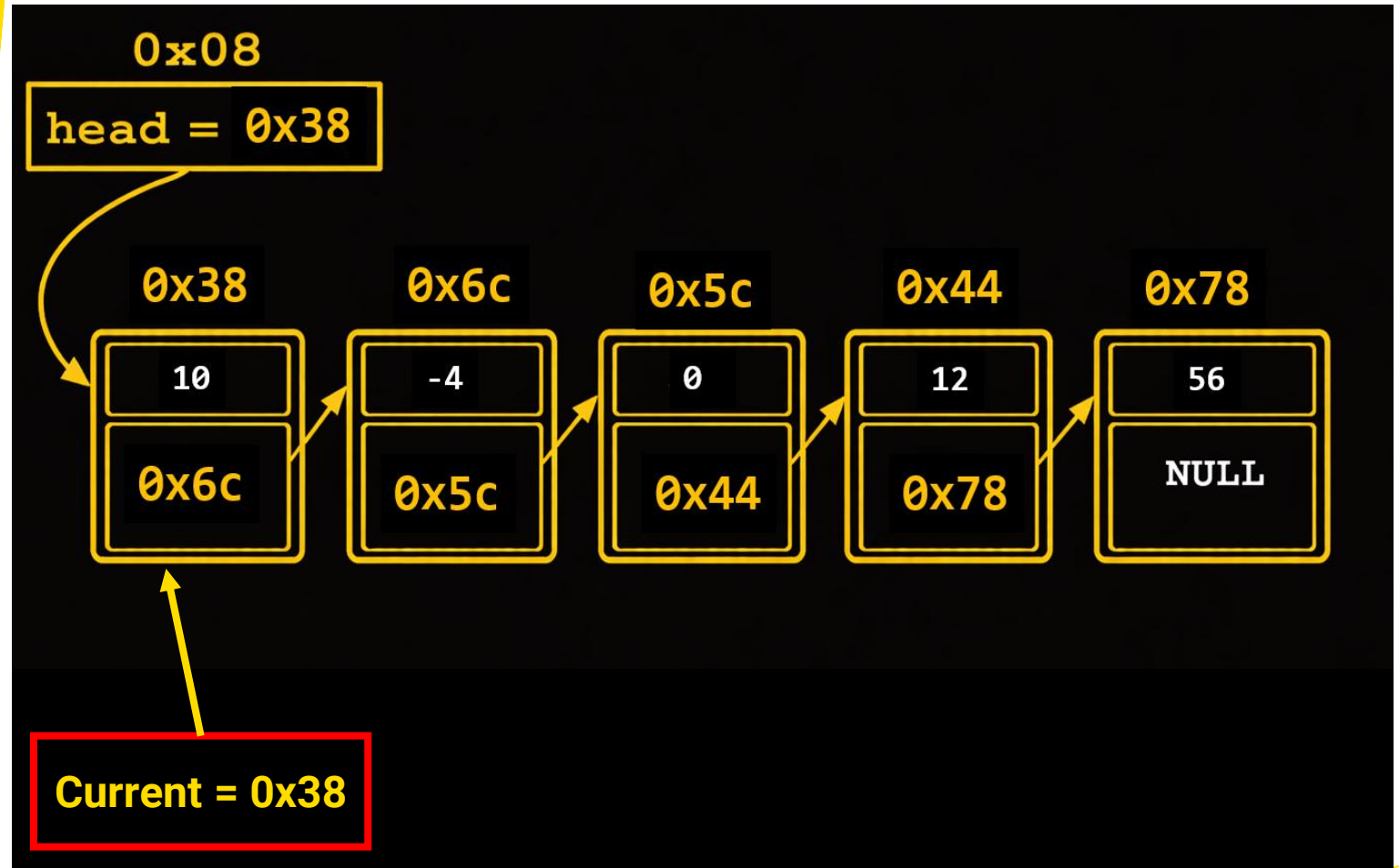
Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow
ideal coding style.



Traversing a Linked List recap

First, we should set a **pointer to the beginning of the list**

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



Traversing a Linked List

Then we should **move current** to the next nodes one by one



Traversing a Linked List

Then we should **move current** to the next nodes one by one

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



Traversing a Linked List

Then we should **move current** to the next nodes one by one

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



Traversing a Linked List

Then we should **move current** to the next nodes one by one

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



Traversing a Linked List

Then we should **move current** to the next nodes one by one

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



Traversing a Linked List

We should stop
when
current = NULL

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

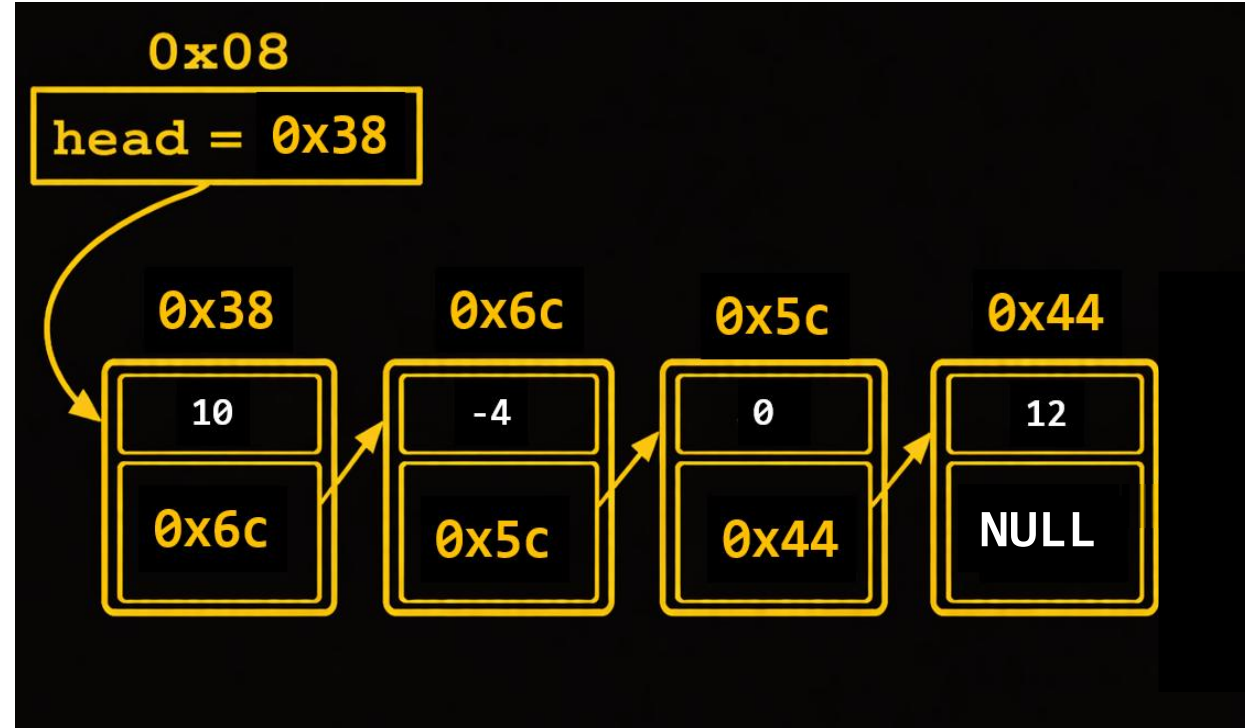


Demo

list_search_value.c

Inserting at a given position

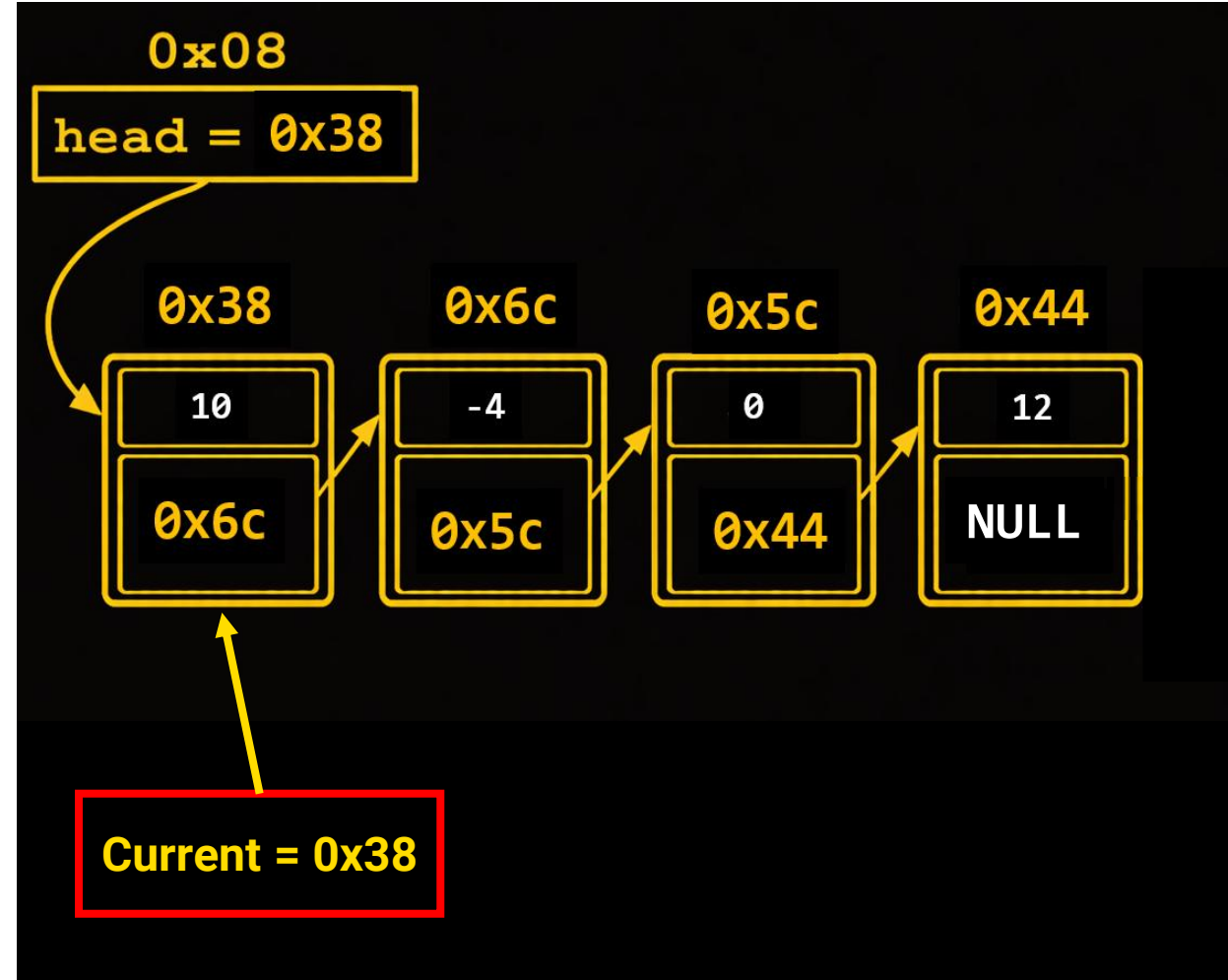
We want to **insert a new node** at the **position 2**, assuming positions **begin counting from 0**. In this example, that means the new node should be inserted at position 2.



This means, we should **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.

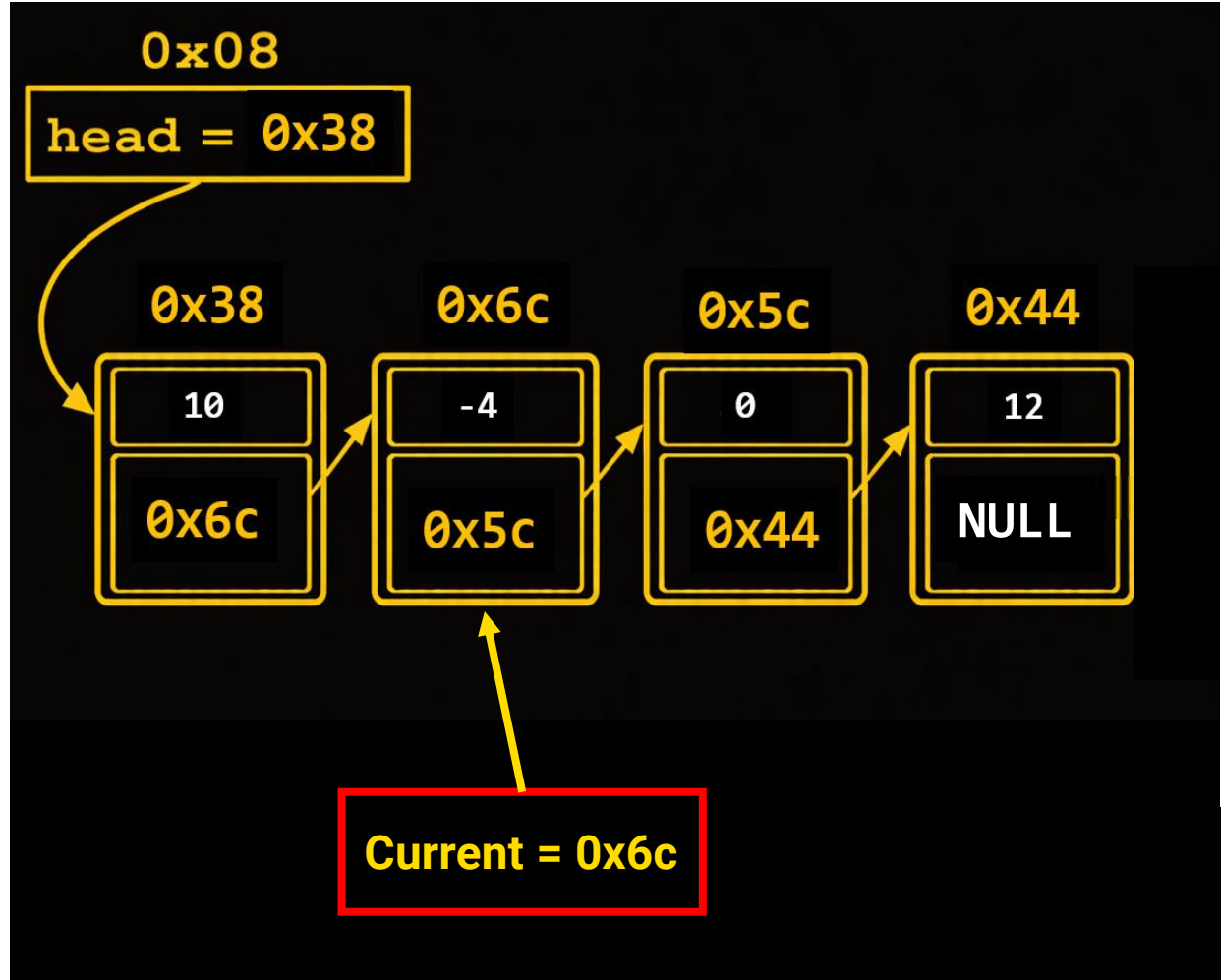
Inserting at Position

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



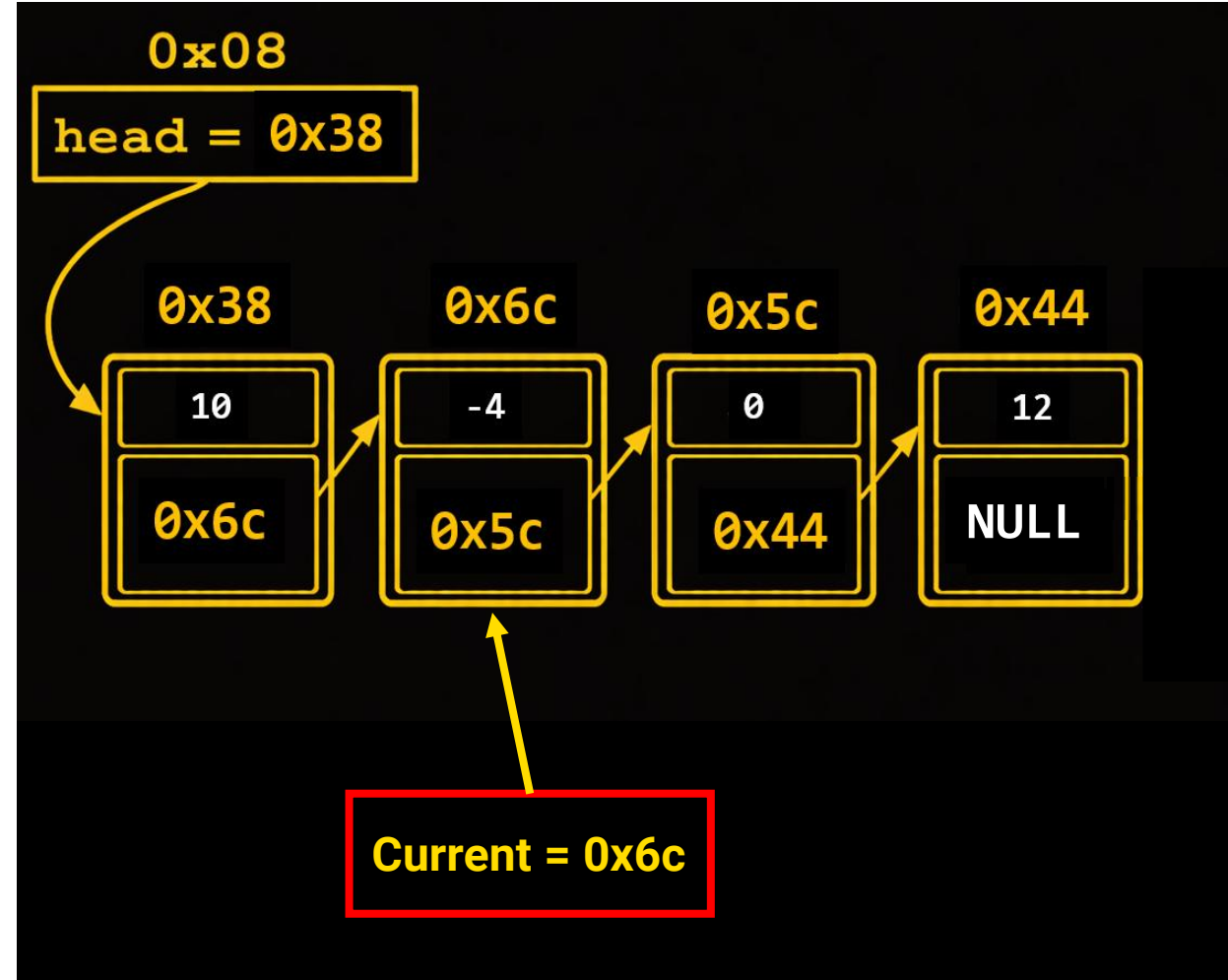
Inserting at Position

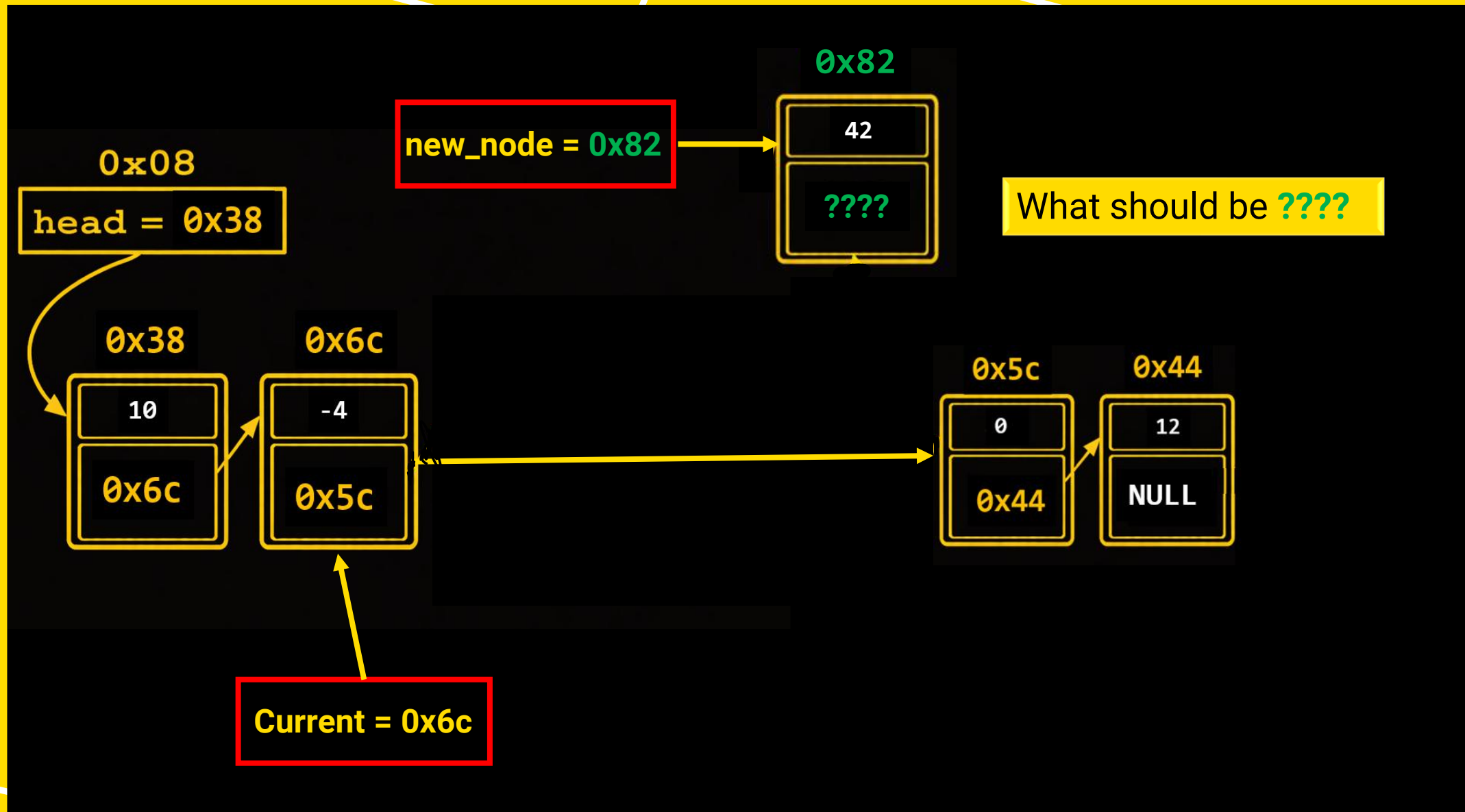
```
while (counter < position - 1) {  
    current = current->next;  
    counter++;  
}
```

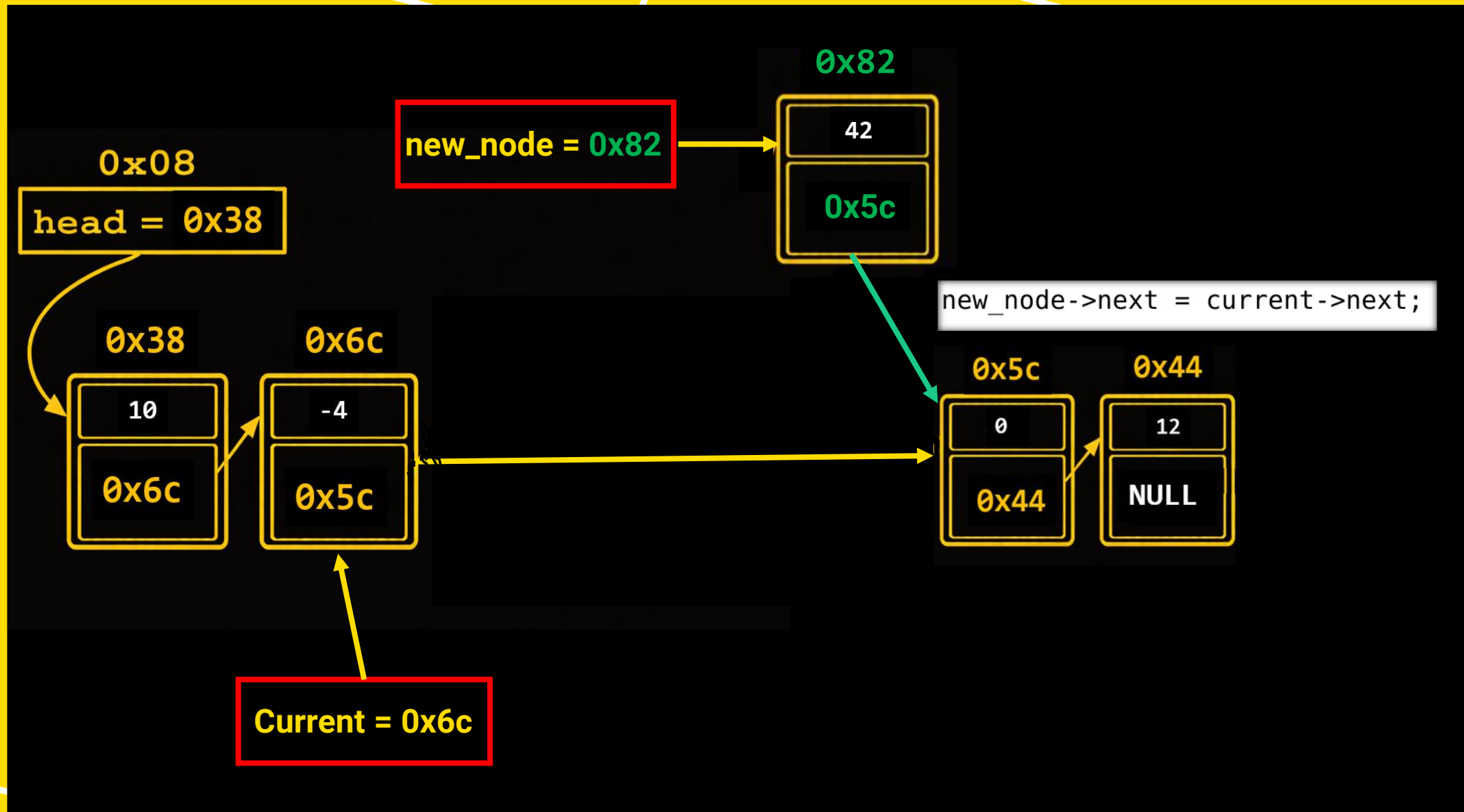


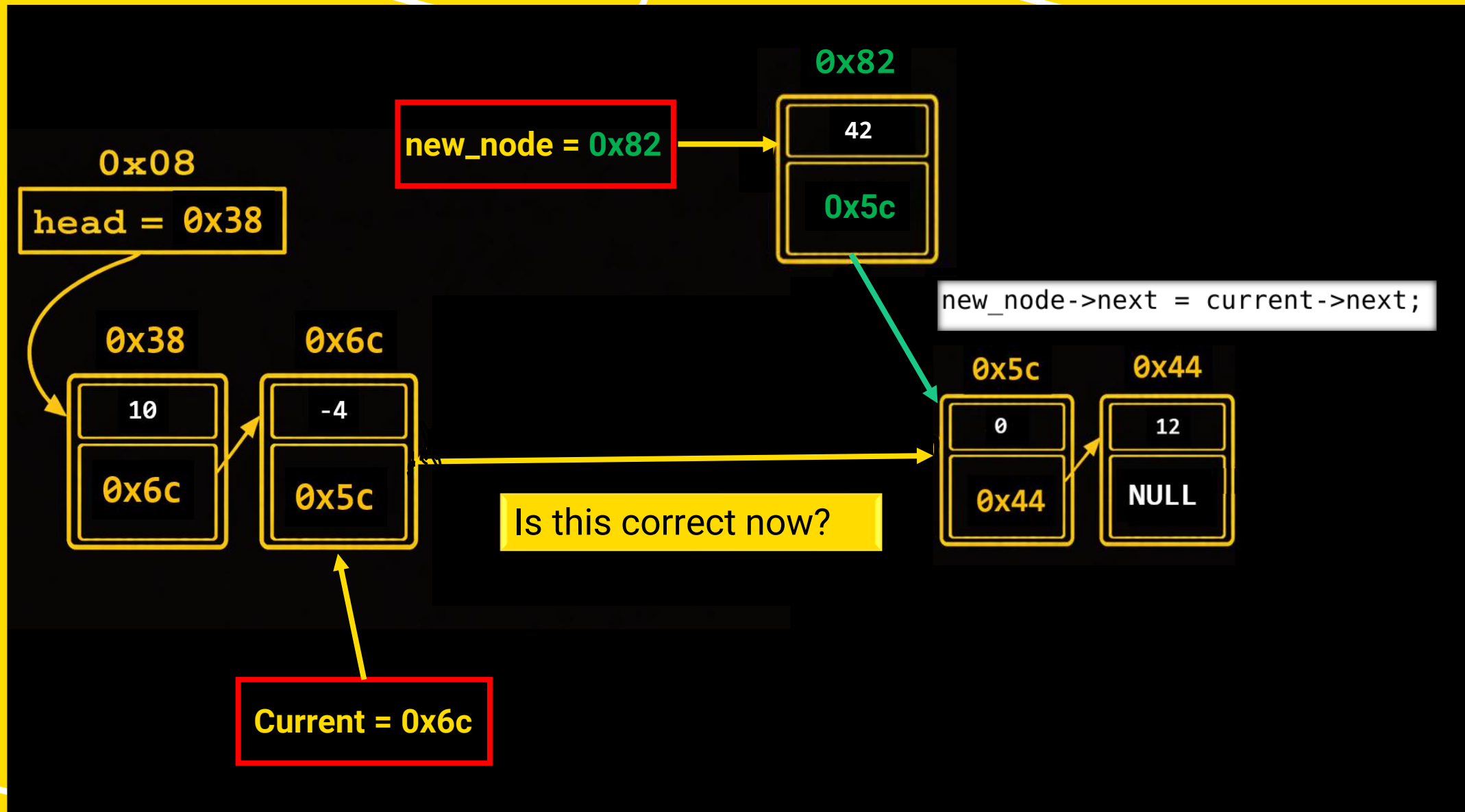
Inserting at Position

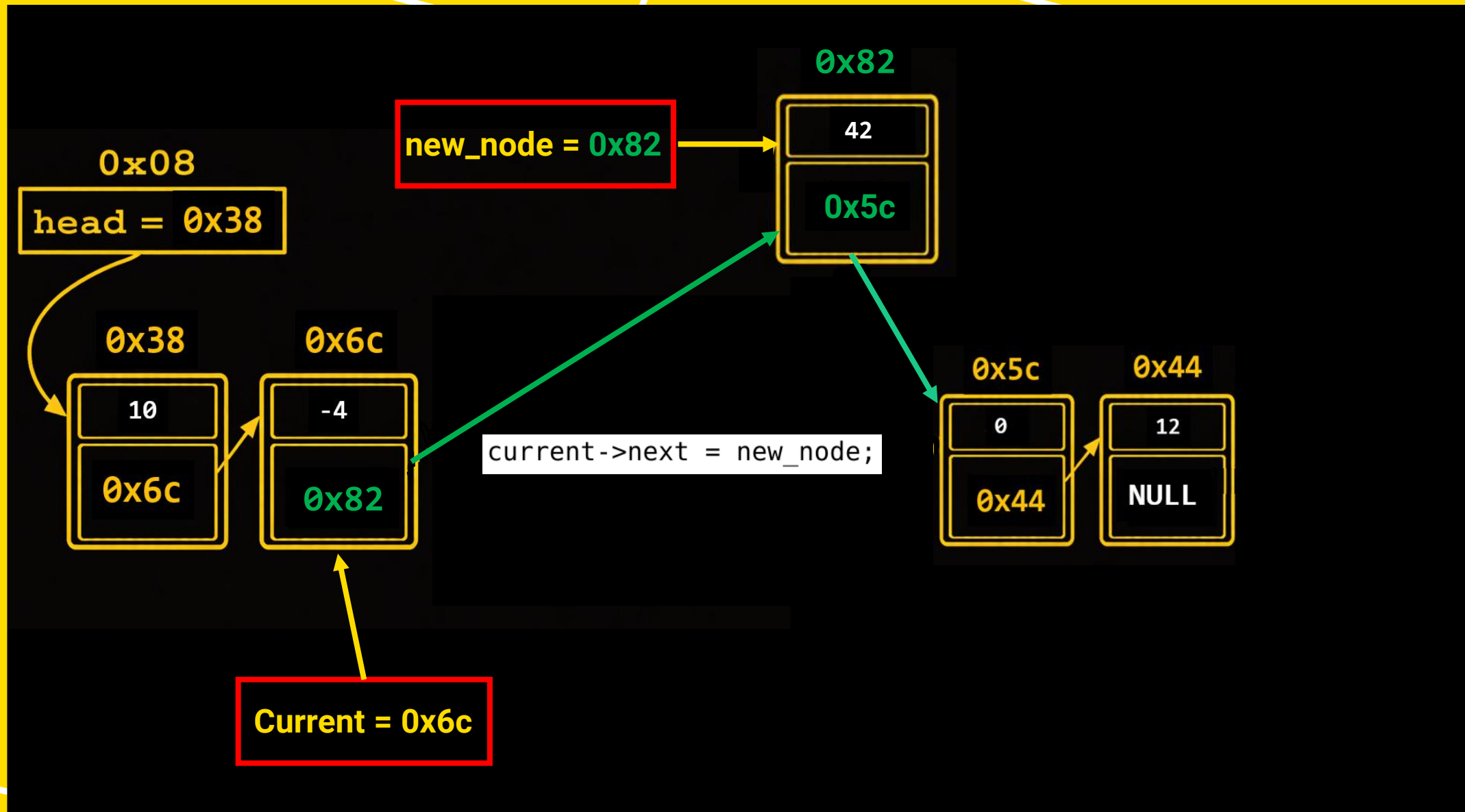
Now we want to connect our **new node**. It should come **after the current node**, but **before current ->next**











What would be the insert coding issues?

What conditions
will break this?

What happens if it is an
empty list?

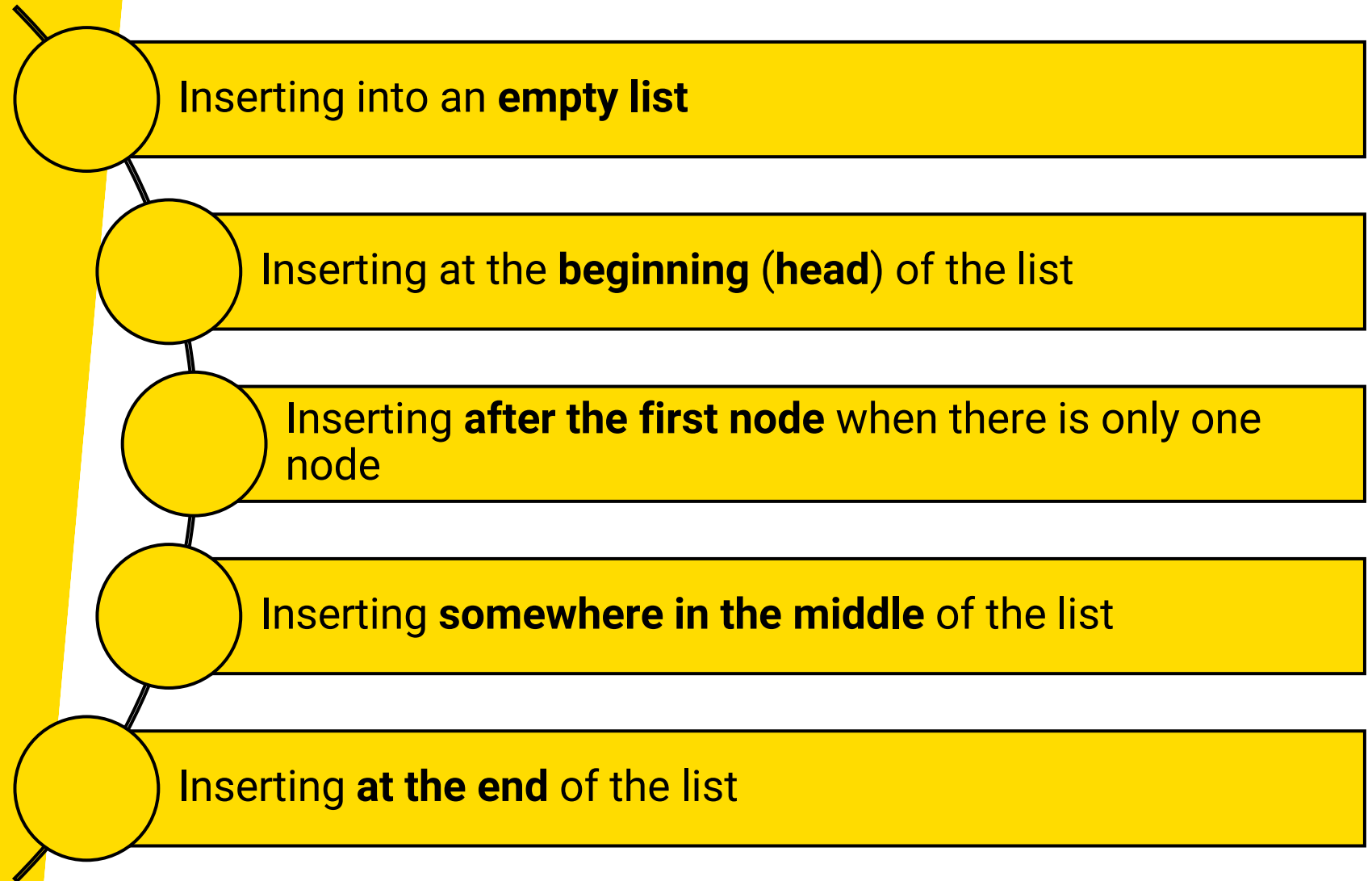
What happens if there is
only 1 item in the list?

Anything else we should
check?

How can we
modify our code
to handle any of
these situations
that break it?

Inserting Into a Linked List Test Cases

Draw a
diagram!!!



Deletion

Deleting the First Node in a Linked List

We
must
first
check
whether
the list
is
empty

If the list is empty,
there is no node to
remove.

In that case, we simply
return the head pointer,
which will 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 **issue** would arise if we called **free** on the head node before updating the head pointer?

```
free(head);
```



Deleting the First Node in a Linked List

Once memory has been freed, we can no longer access it. As a result, we **lose access** to the rest of the list.

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



Deleting the First Node in a Linked List

What problem could occur if we update the head pointer before handling the original first node properly?

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



Deleting the First Node in a Linked List

We no longer have a pointer to the first node, so there is **no way to free** its memory.

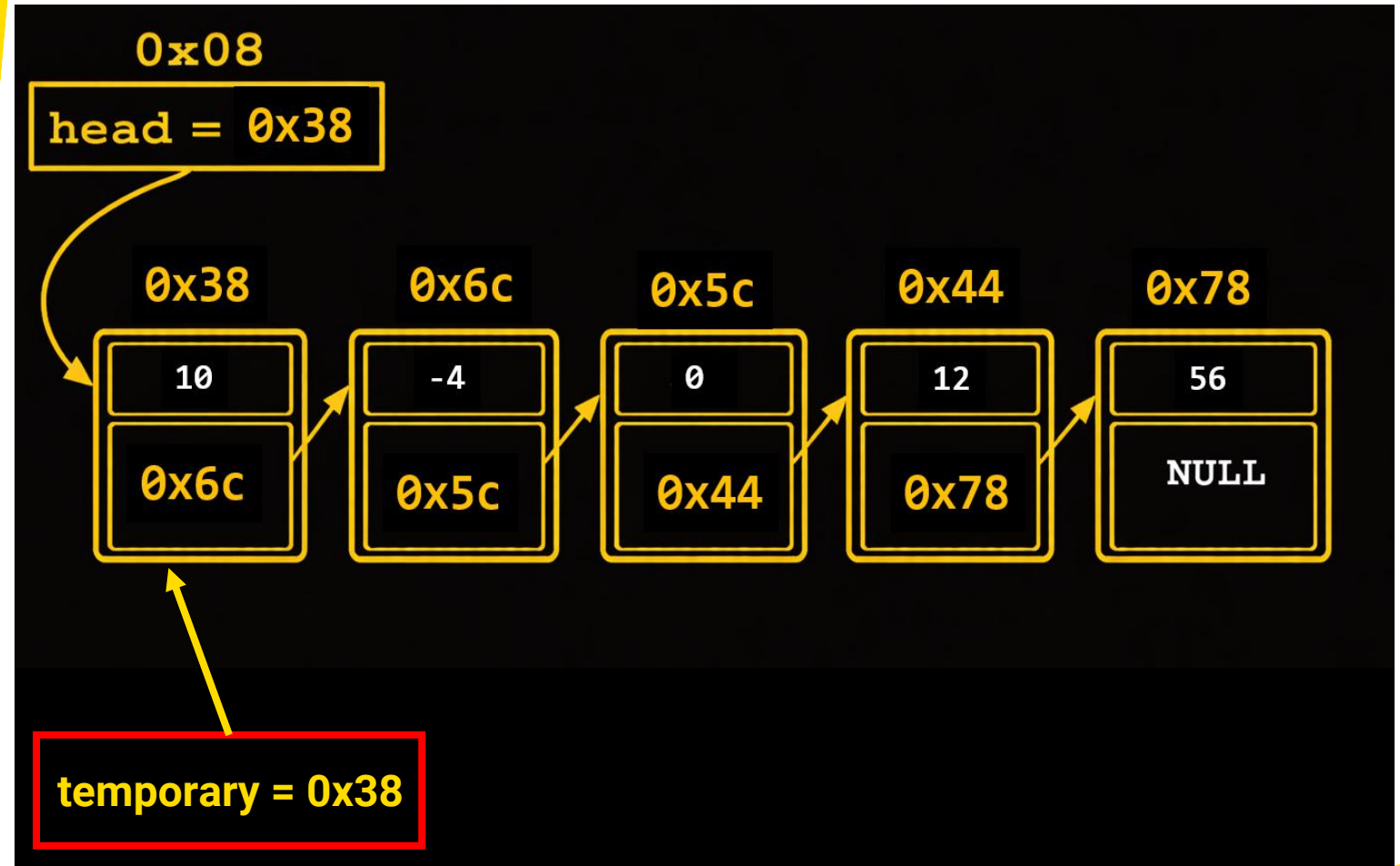
```
free(???);
```



Deleting the First Node in a Linked List

Let's introduce a separate pointer that refers to the first node.

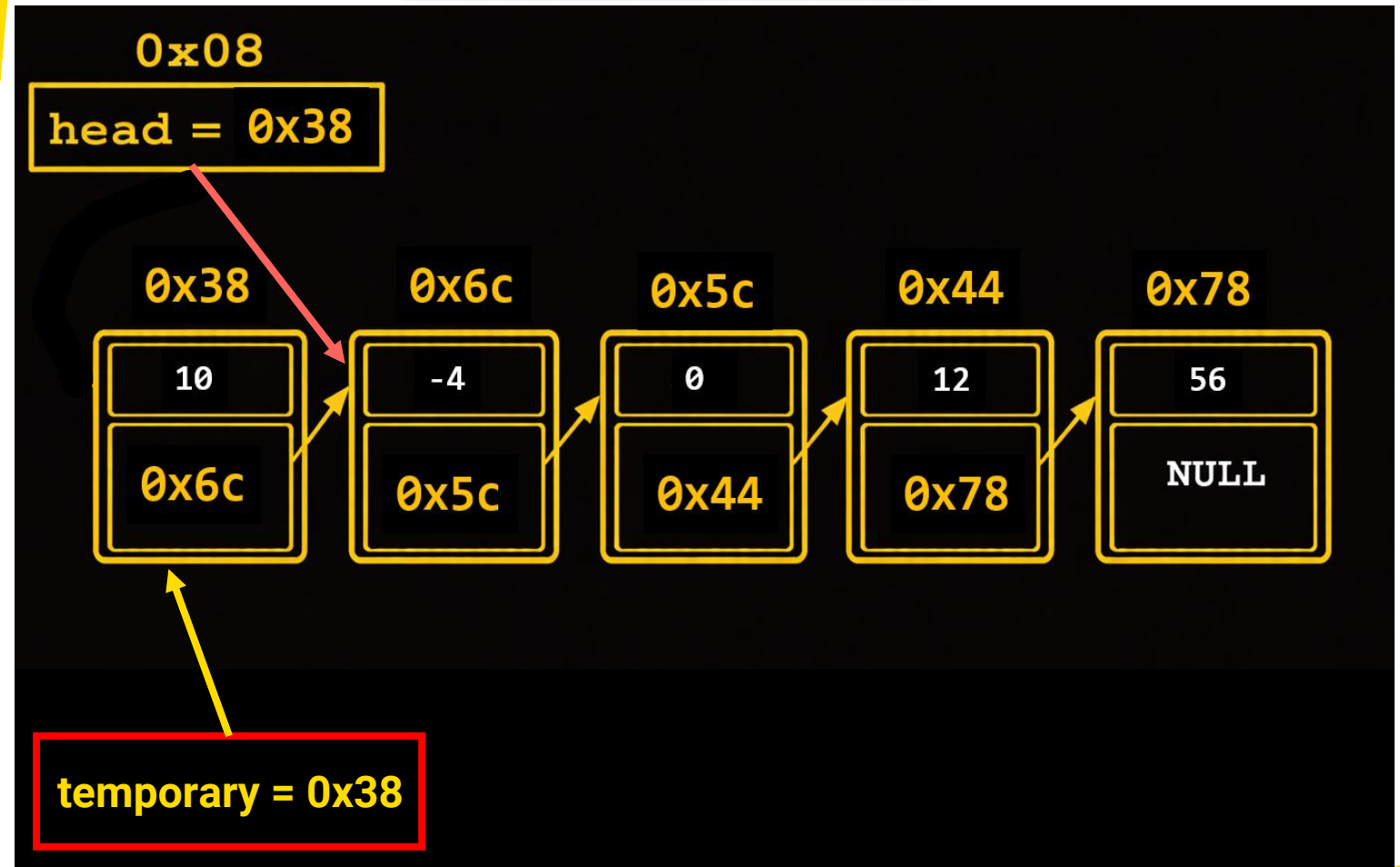
```
struct node *temporary = head;
```



Deleting the First Node in a Linked List

Now we can safely update the head pointer.

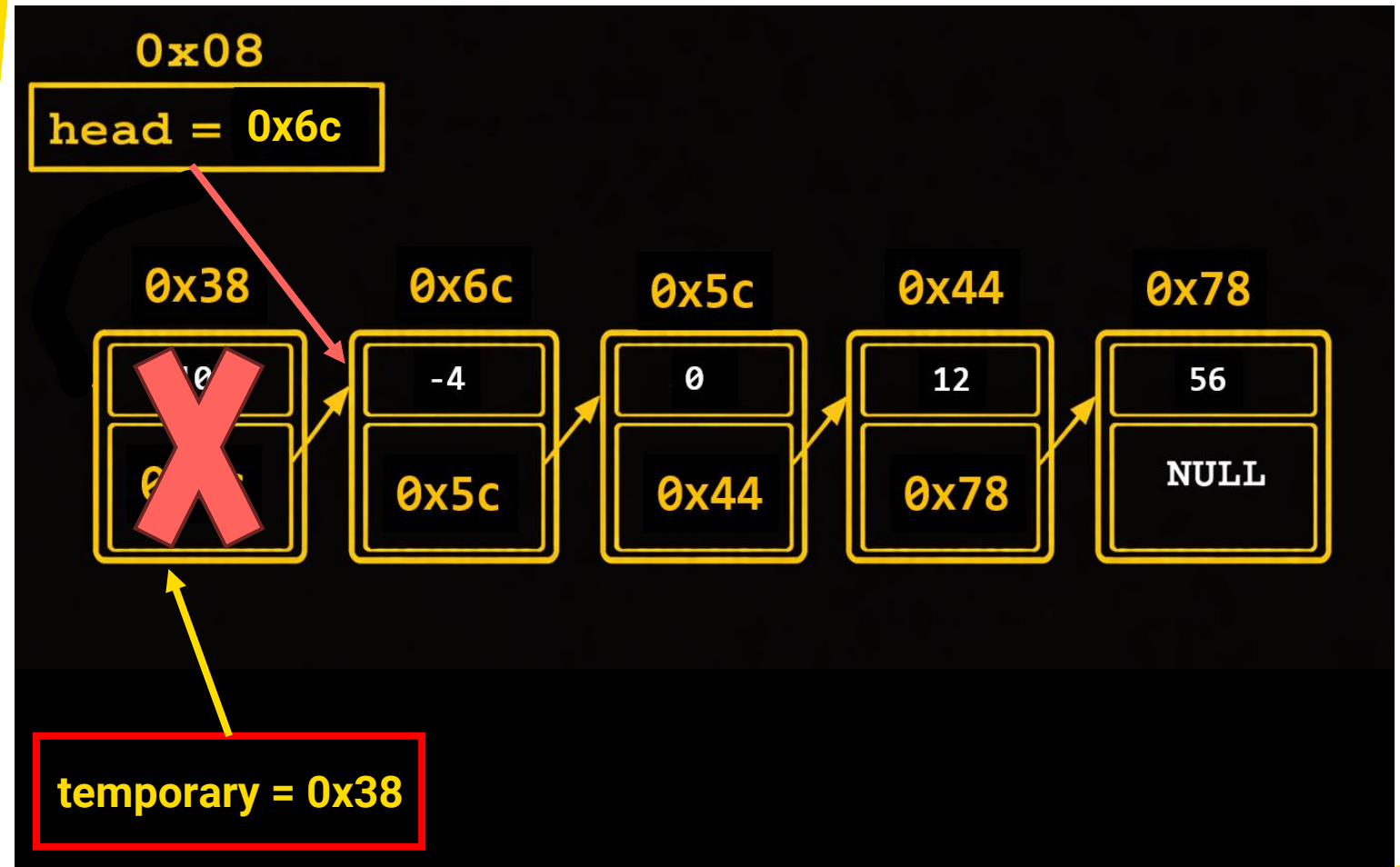
```
head = head->next;
```



Deleting the First Node in a Linked List

Now we can safely free the memory allocated to the first node.

```
free(temporary);
```



Deleting the First Node in a Linked 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;  
}
```

Deleting All Nodes

Wrong way

Is this code correct?

```
// 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;
    }
}
```

Deleting All Nodes

Wrong way

Remember, once you free a block of memory, **it must no longer be accessed or used.**

```
// 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 already been freed
        current = current->next;
    }
}
```

Deleting All Nodes

Correct way

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

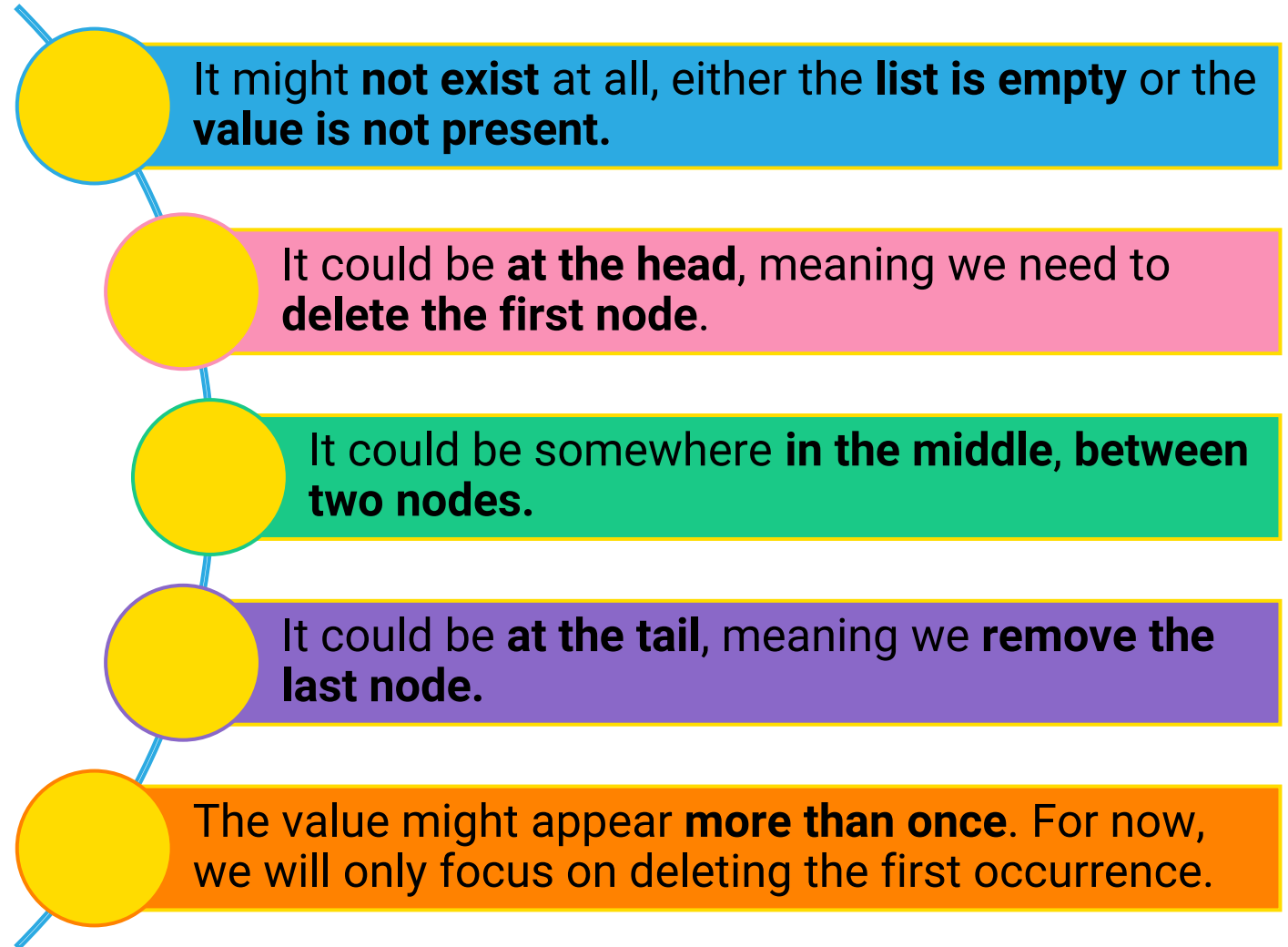
```
// Delete all nodes from a given list
void delete_all_nodes(struct node *head) {

    struct node *current = head;

    while (current != NULL) {
        head = head->next;
        free(current);
        current = head;
    }
}
```

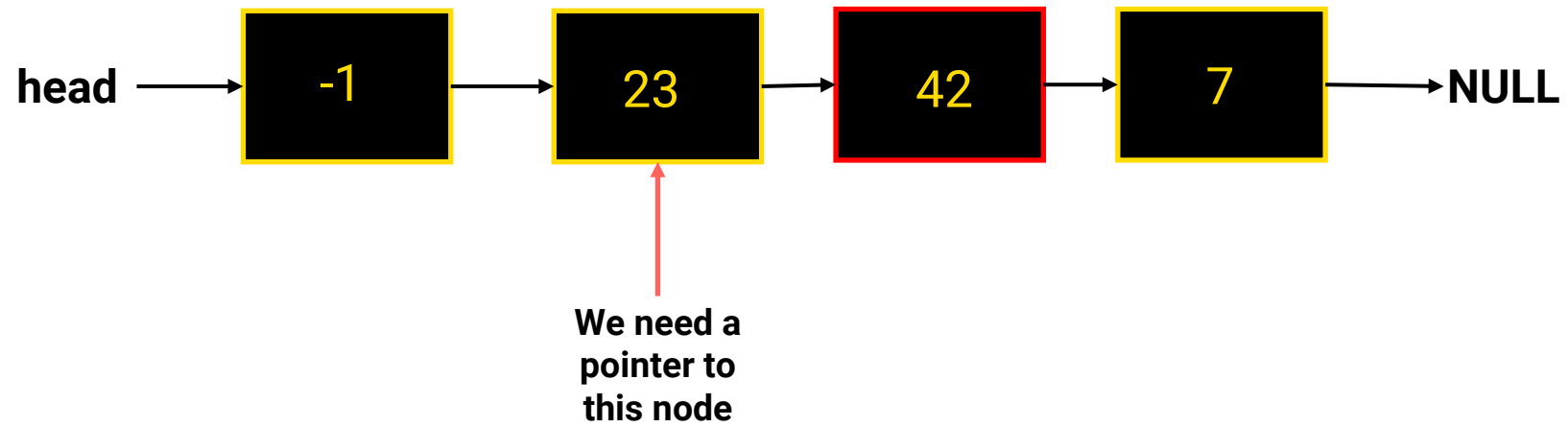
Search and Delete

We want to find a node that contains a specific value and then remove it from the list.
Where might this node be located?



Search and delete between 2 nodes

To delete a node, we must **connect the previous node directly to the next node**; so, if we want to remove the node containing 42, we first need to **locate the node that comes before it**.



Search and delete between 2 nodes

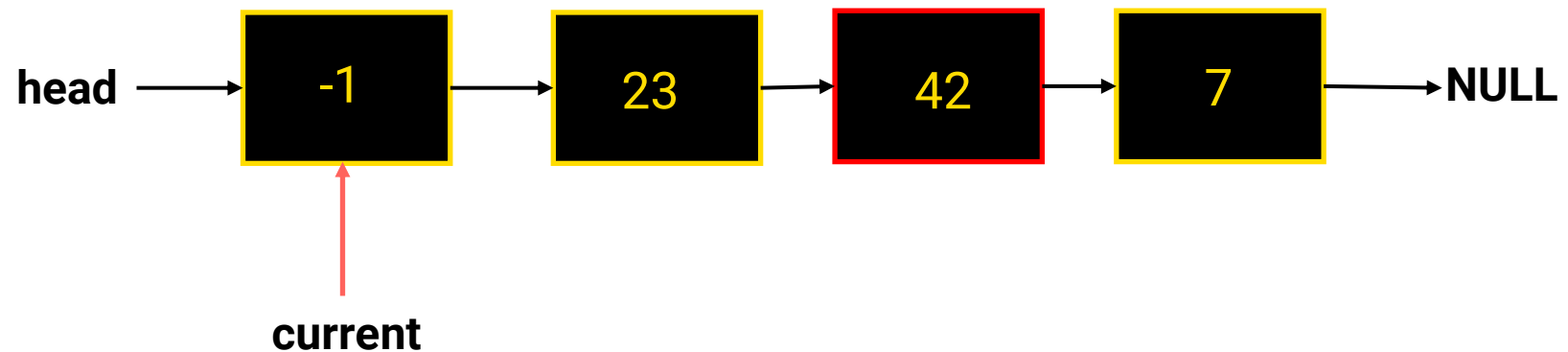
Method 1

```
// Method 1:  
// Use two pointers while traversing the list:  
// - 'current' moves through each node to find the target value.  
// - 'previous' keeps track of the node before 'current'.  
// This allows us to relink nodes properly when deleting.
```

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

```
// Traverse the list until:  
// 1. We reach the end (current == NULL), or  
// 2. We find the node containing search_key.
```

```
while (current != NULL && current->data != search_key) {  
    previous = current;    // Move previous forward  
    current = current->next; // Move current forward  
}
```



previous = NULL

Search and delete between 2 nodes

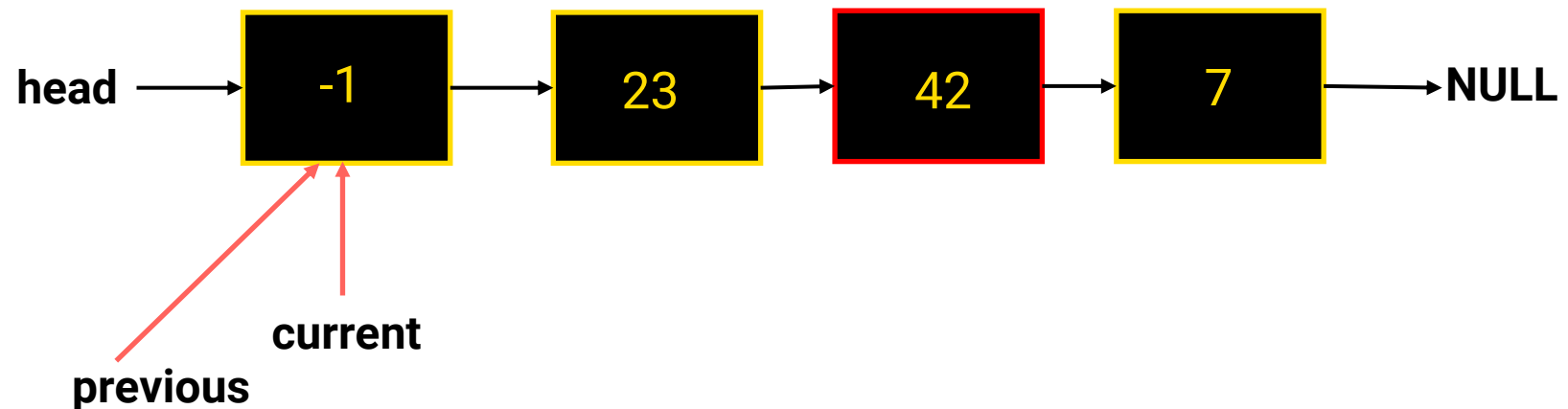
Method 1

```
// Method 1:  
// Use two pointers while traversing the list:  
// - 'current' moves through each node to find the target value.  
// - 'previous' keeps track of the node before 'current'.  
// This allows us to relink nodes properly when deleting.
```

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

```
// Traverse the list until:  
// 1. We reach the end (current == NULL), or  
// 2. We find the node containing search_key.
```

```
while (current != NULL && current->data != search_key) {  
    previous = current;    // Move previous forward  
    current = current->next; // Move current forward  
}
```



Search and delete between 2 nodes

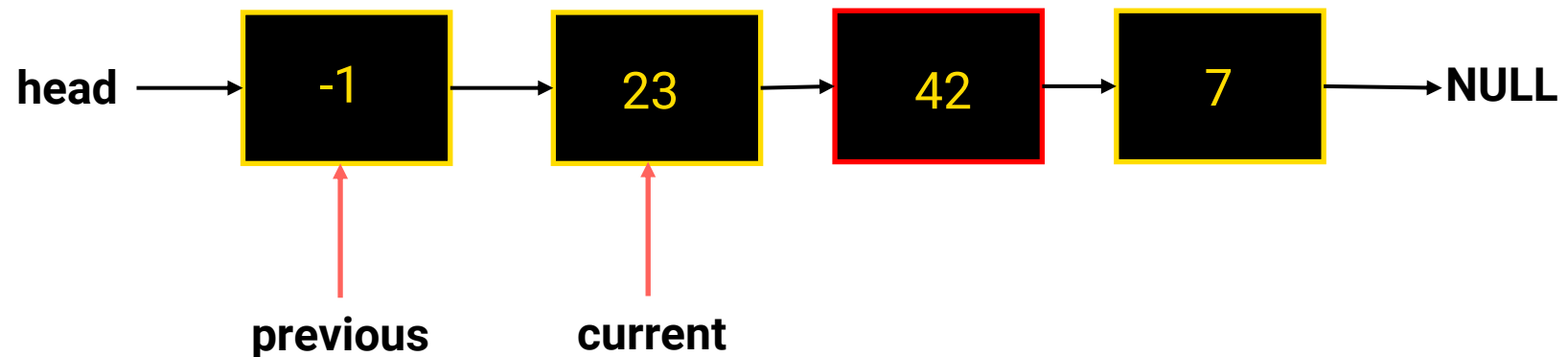
Method 1

```
// Method 1:  
// Use two pointers while traversing the list:  
// - 'current' moves through each node to find the target value.  
// - 'previous' keeps track of the node before 'current'.  
// This allows us to relink nodes properly when deleting.
```

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

```
// Traverse the list until:  
// 1. We reach the end (current == NULL), or  
// 2. We find the node containing search_key.
```

```
while (current != NULL && current->data != search_key) {  
    previous = current;    // Move previous forward  
    current = current->next; // Move current forward  
}
```



Search and delete between 2 nodes

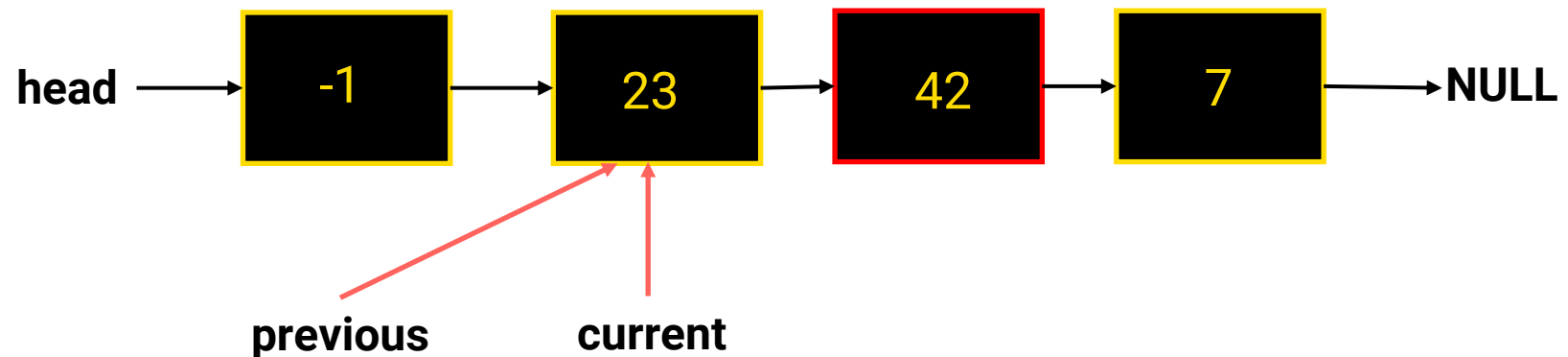
Method 1

```
// Method 1:  
// Use two pointers while traversing the list:  
// - 'current' moves through each node to find the target value.  
// - 'previous' keeps track of the node before 'current'.  
// This allows us to relink nodes properly when deleting.
```

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

```
// Traverse the list until:  
// 1. We reach the end (current == NULL), or  
// 2. We find the node containing search_key.
```

```
while (current != NULL && current->data != search_key) {  
    previous = current;    // Move previous forward  
    current = current->next; // Move current forward  
}
```



Search and delete between 2 nodes

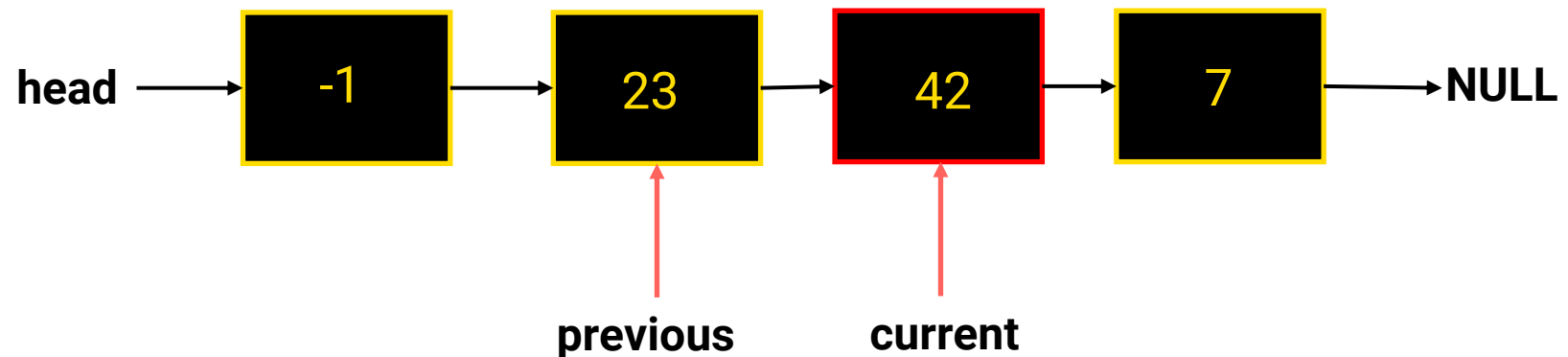
Method 1

```
// Method 1:  
// Use two pointers while traversing the list:  
// - 'current' moves through each node to find the target value.  
// - 'previous' keeps track of the node before 'current'.  
// This allows us to relink nodes properly when deleting.
```

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

```
// Traverse the list until:  
// 1. We reach the end (current == NULL), or  
// 2. We find the node containing search_key.
```

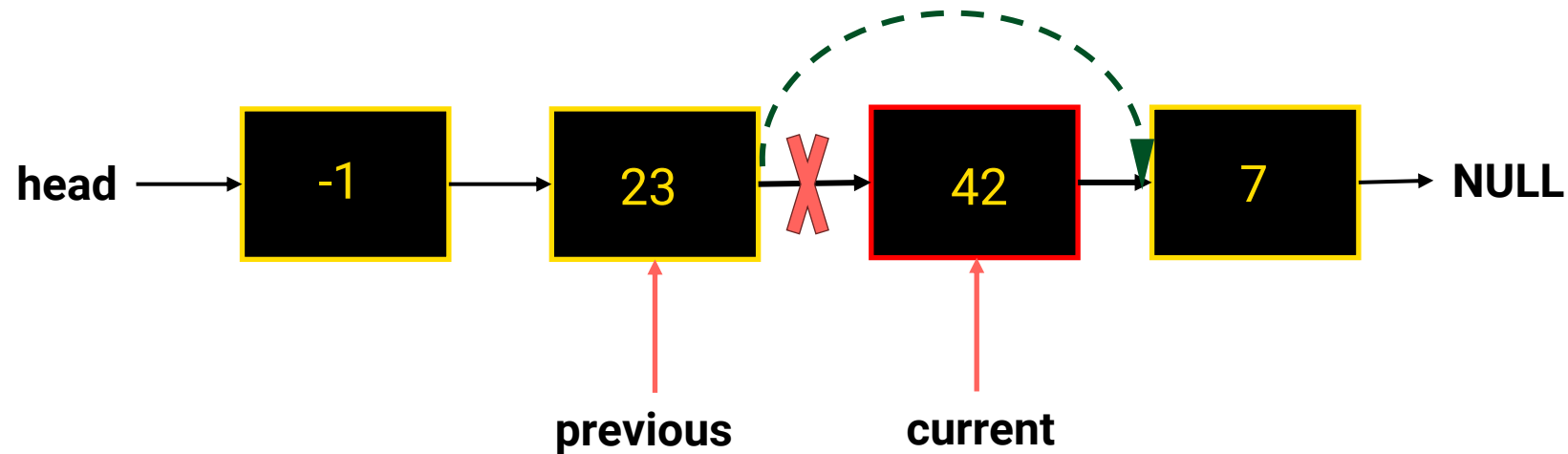
```
while (current != NULL && current->data != search_key) {  
    previous = current;    // Move previous forward  
    current = current->next; // Move current forward  
}
```



Search and delete between 2 nodes

Method 1

Now, we must **update the links** so that the **previous node points to the node that comes after the one being removed (current)**.

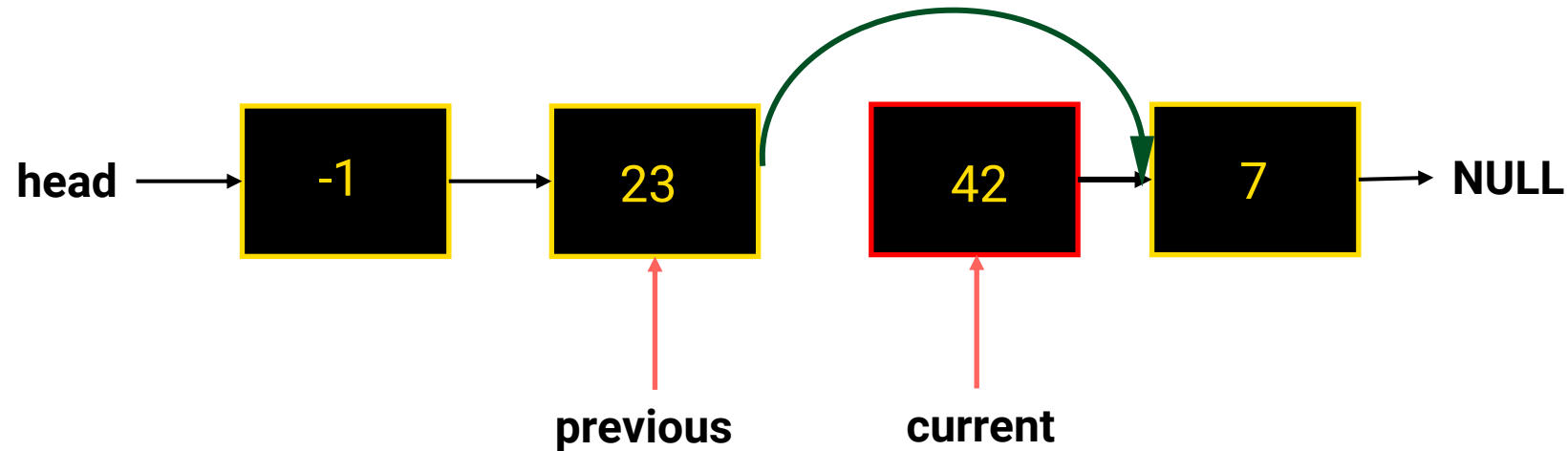


Search and delete between 2 nodes

Method 1

To connect **previous node** to the node that comes after the one being removed (**current**).

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

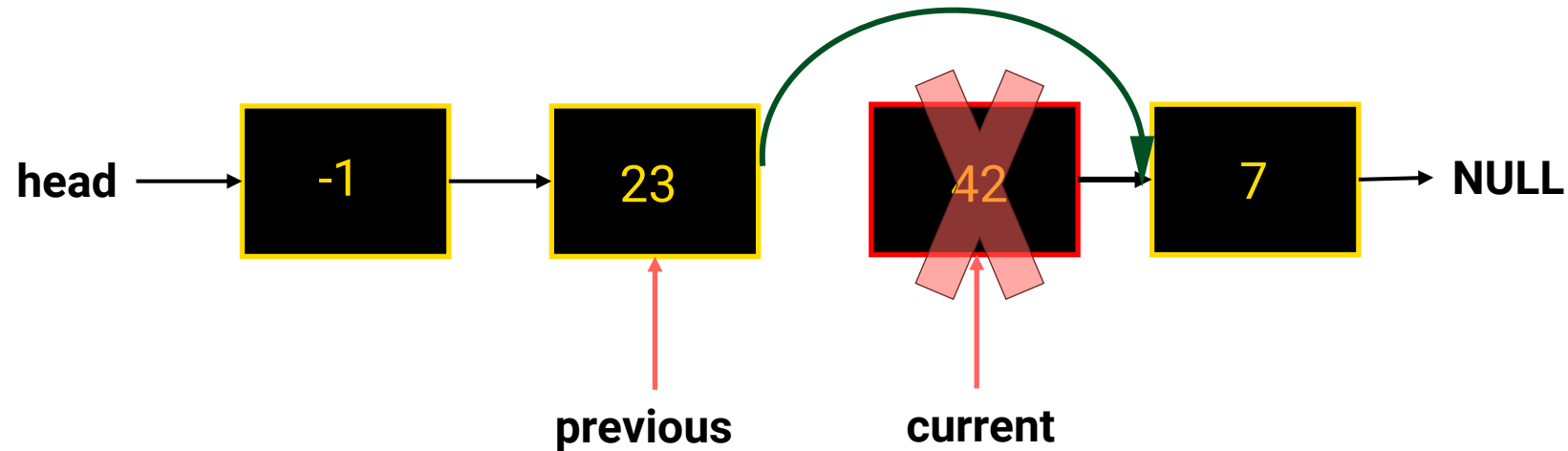


Search and delete between 2 nodes

Method 1

Now we can **free** the node we wanted to **remove** (current)

```
free(current);
```

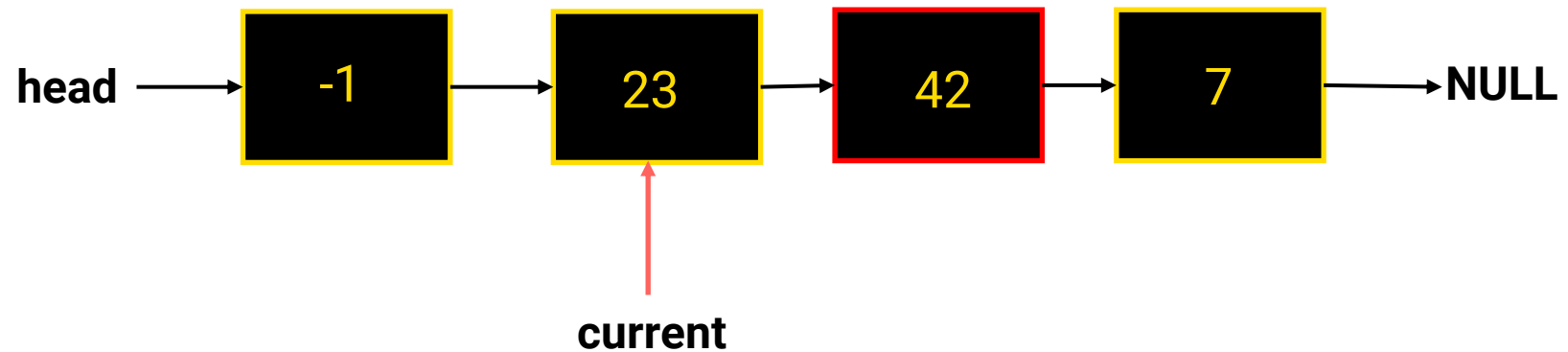


Search and delete between 2 nodes

Method 2

Use only one pointer (current) to move through the list.
Instead of looking at the current node's data, we examine the NEXT node's data (current->next).

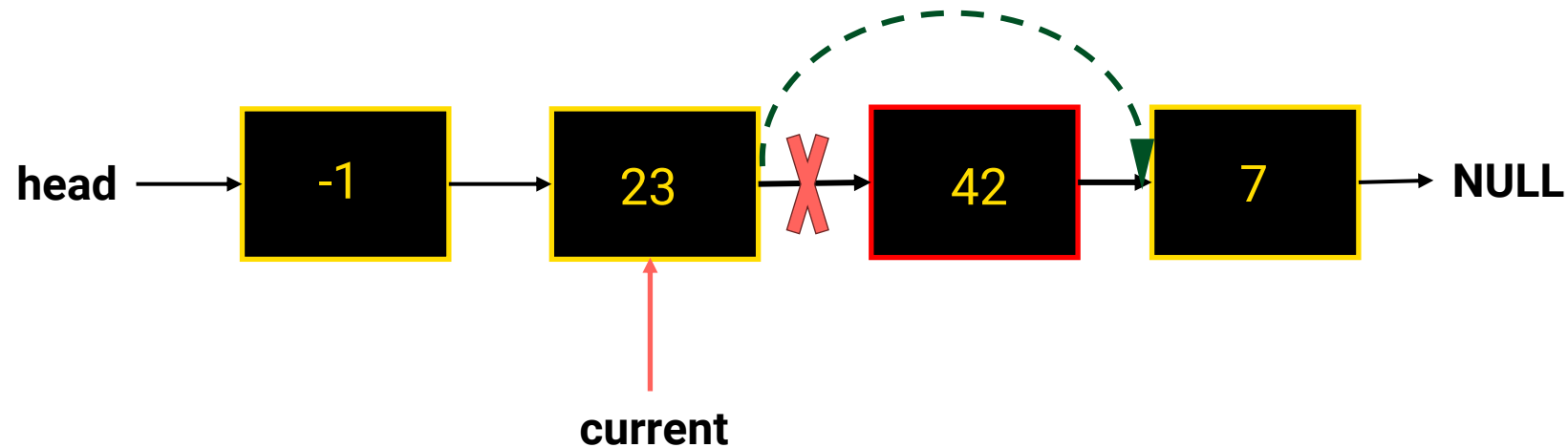
Why? Because if current->next contains the value we want to delete, then 'current' is the node right before the one we need to remove.



Search and delete between 2 nodes

Method 2

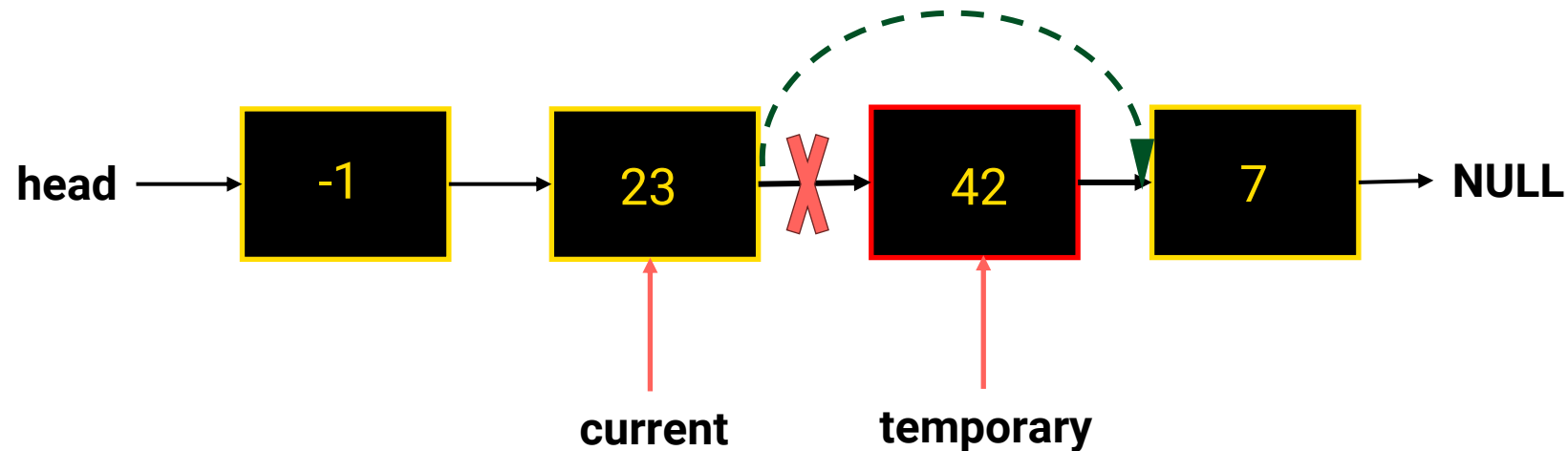
Next, we must update the link so that the current node points to the node after the one being removed. However, we still need a separate pointer to the node we intend to free. Otherwise, we lose it!!!!



Search and delete between 2 nodes

Method 2

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

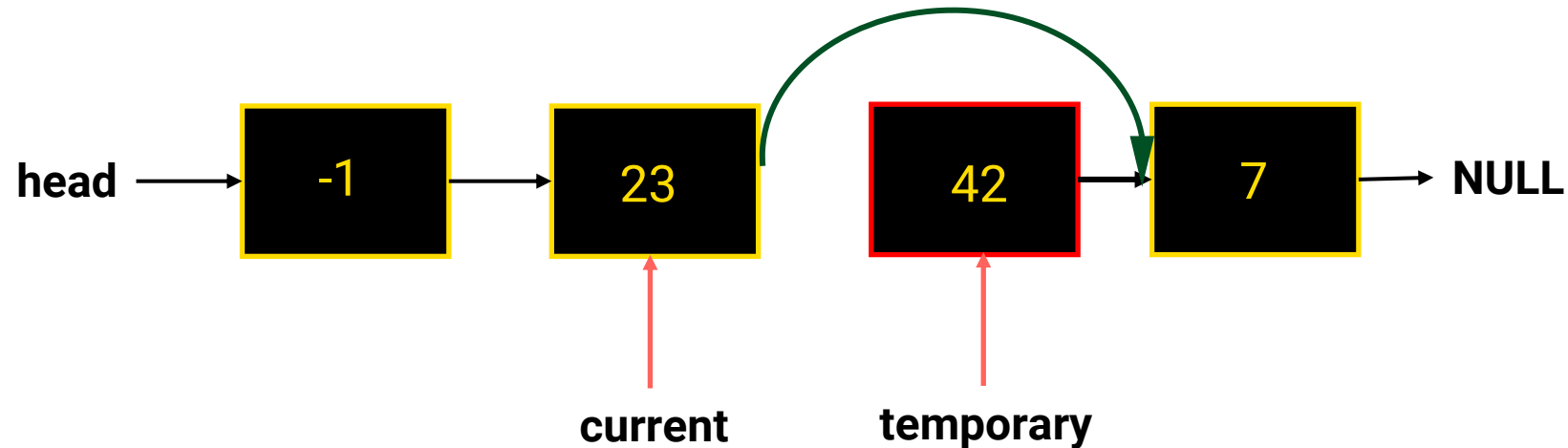


Search and delete between 2 nodes

Method 2

To connect **current node** to the node that comes after the one being removed (temporary):

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

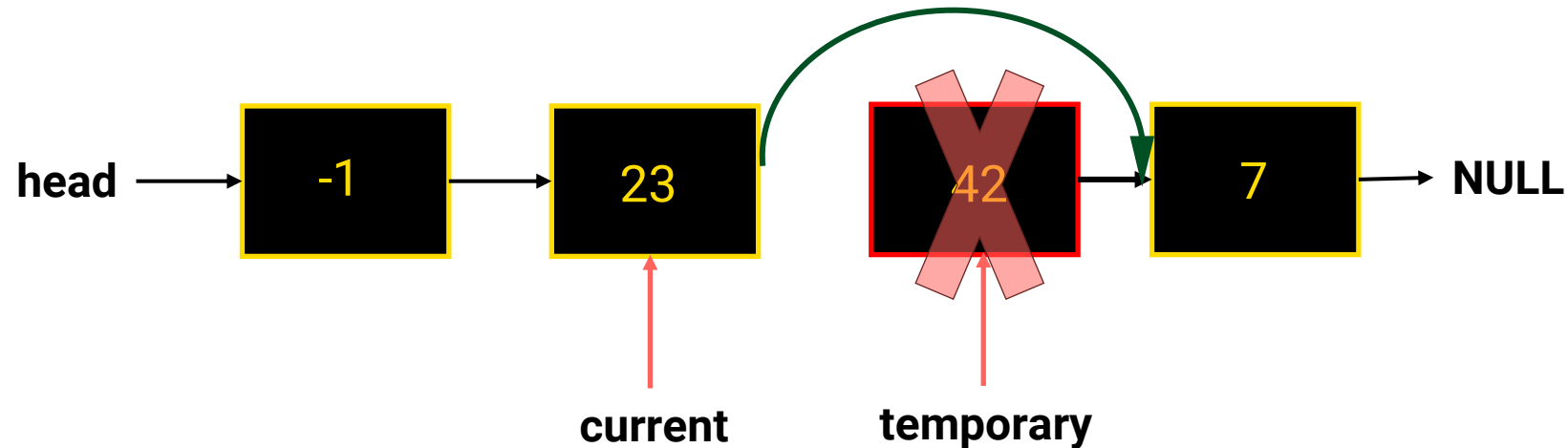


Search and delete between 2 nodes

Method 2

Now we can **free** the node we wanted to **remove** (**current**)

```
free(temporary);
```



Demo

list_delete.c

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me



Voice of the Student



[Voice of the Student – Fill out form](#)

See you soon ...