COMP1511 PROGRAMMING FUNDAMENTALS

LECTURE 13/14

Insert anywhere in the linked list

Time to delete from a linked list

LAST WEEK...

- Linked Lists -
  - creating a list
  - inserting nodes at the head
  - traversing a list
  - inserting nodes at the tail

TODAY...

- Linked Lists -
  - inserting anywhere in a linked list
  - deleting nodes in a list
    - at the head
    - at the tail
    - in the middle
    - with only one item in a list

WHERE IS THE CODE?

Live lecture code can be found here:

HTTPS://CGI.CSE.UNSW.EDU.AU/~CS1511/23T1/LIVE/WEEK08/

A LINKED LIST

WHY?

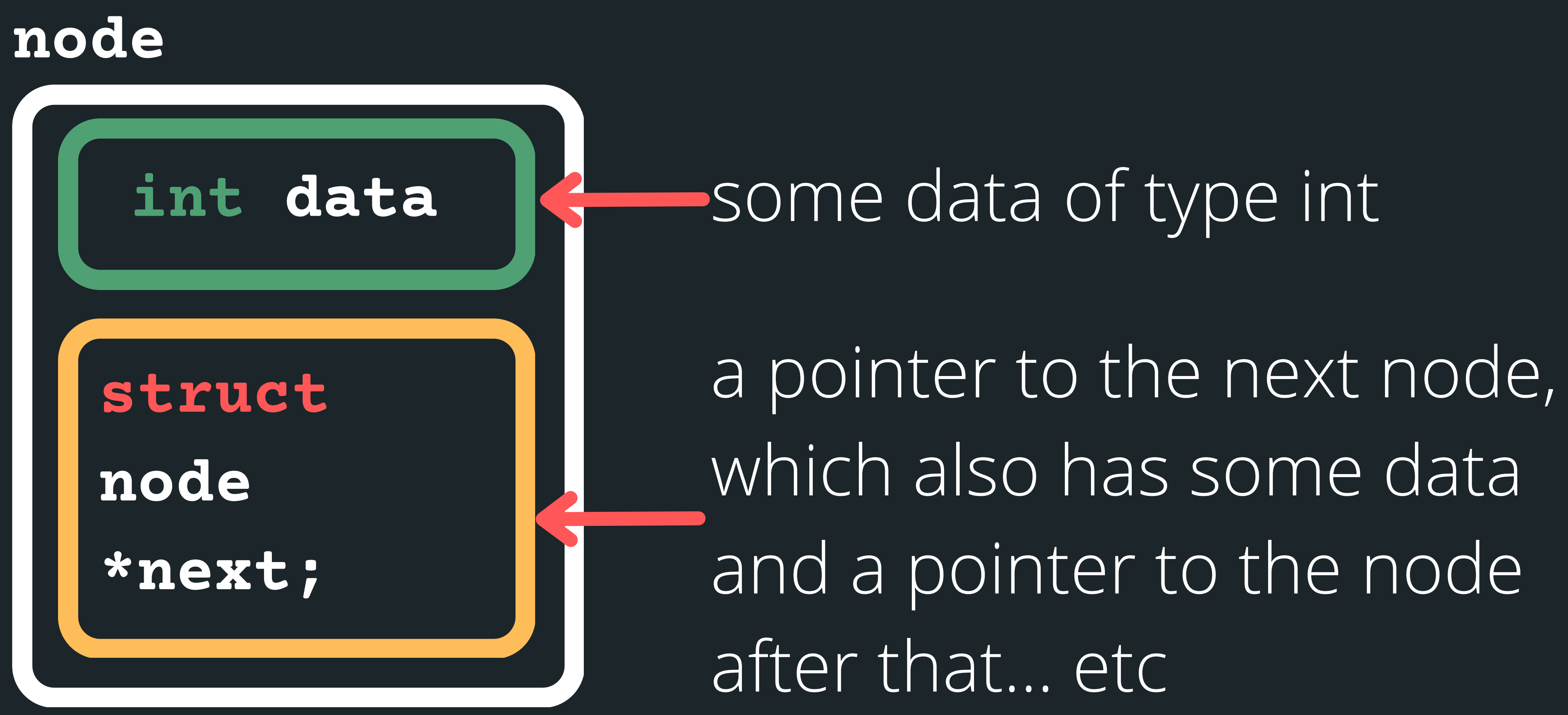
- Linked lists are dynamically sized, that means we can grow and shrink them as needed - efficient for memory!
- Elements of a linked list (called nodes) do NOT need to be stored contiguously in memory, like an array.
- We can add or remove nodes as needed anywhere in the list, without worrying about size (unless we run out of memory of course!)
- We can change the order in a linked list, by just changing where the next pointer is pointing to!
- Unlike arrays, linked lists are not random access data structures! You can only access items sequentially, starting from the beginning of the list.

A LINKED LIST IS MADE UP OF NODES

WHAT IS A NODE?

- Each node has some data and a pointer to the next node (of the same data type), creating a linked structure that forms the list
- Let me propose a node structure like this:

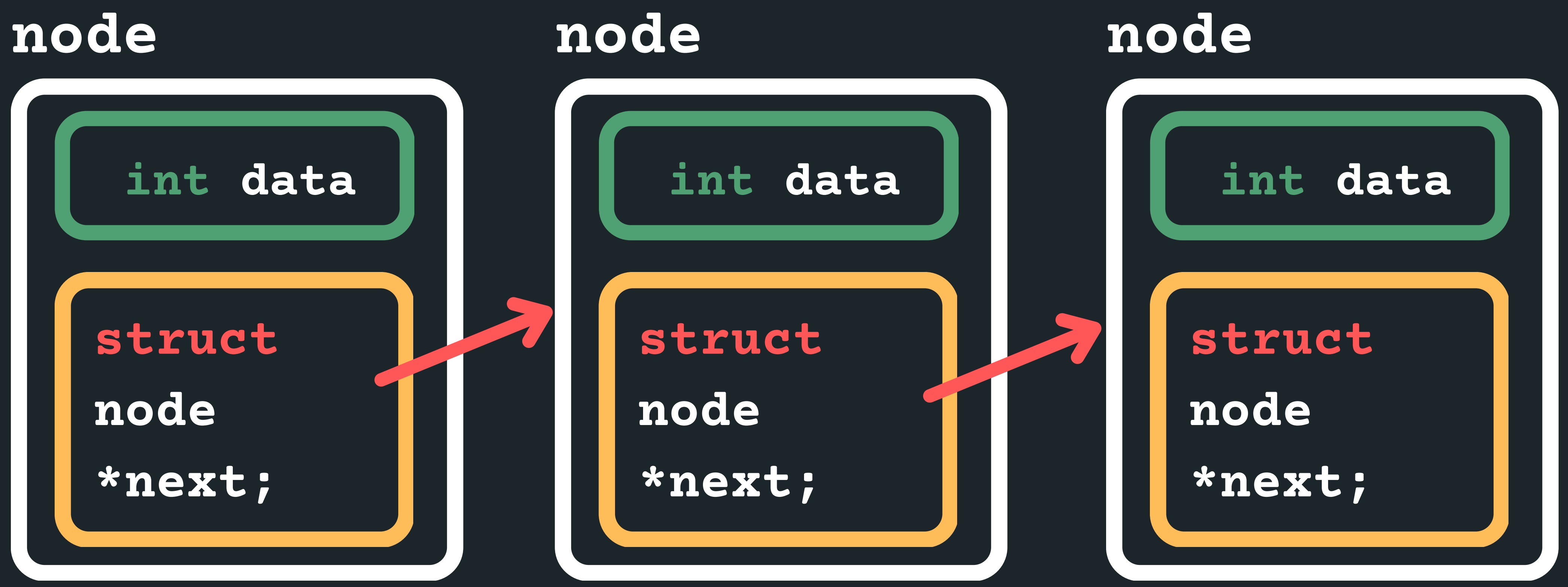
```
struct node {
    int data;
    struct node *next;
};
```



A LINKED LIST IS MADE UP OF MANY NODES

THE NODES ARE LINKED TOGETHER (A SCAVENGER HUNT OF POINTERS)

- We can create a linked list, by having many nodes together, with each struct node next pointer giving us the address of the node that follows it

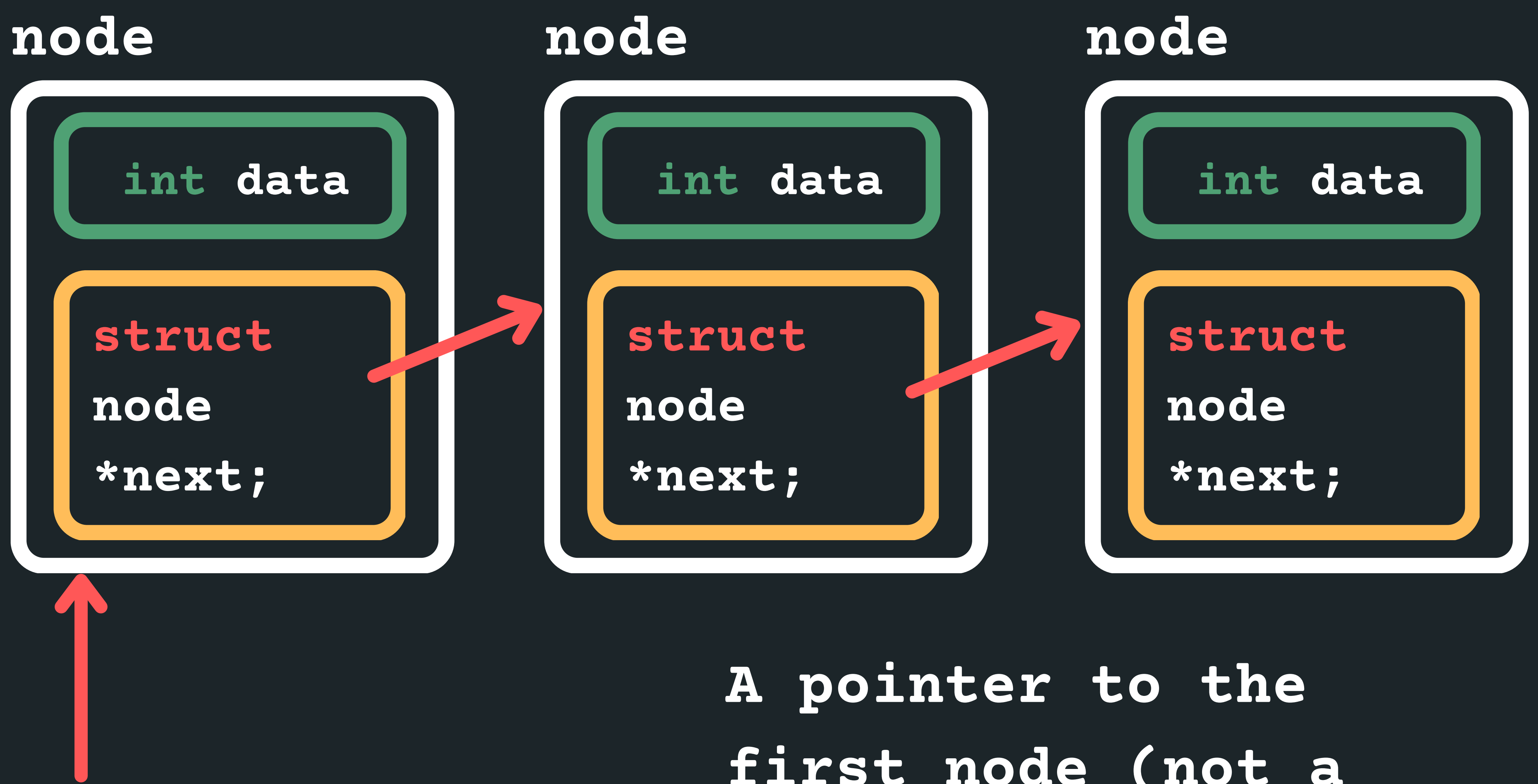


- But how do I know where the linked list starts?

A LINKED LIST IS MADE UP OF MANY NODES

THE NODES ARE LINKED TOGETHER (A SCAVENGER HUNT OF POINTERS)

- What about a pointer to the first node?



A pointer to the first node

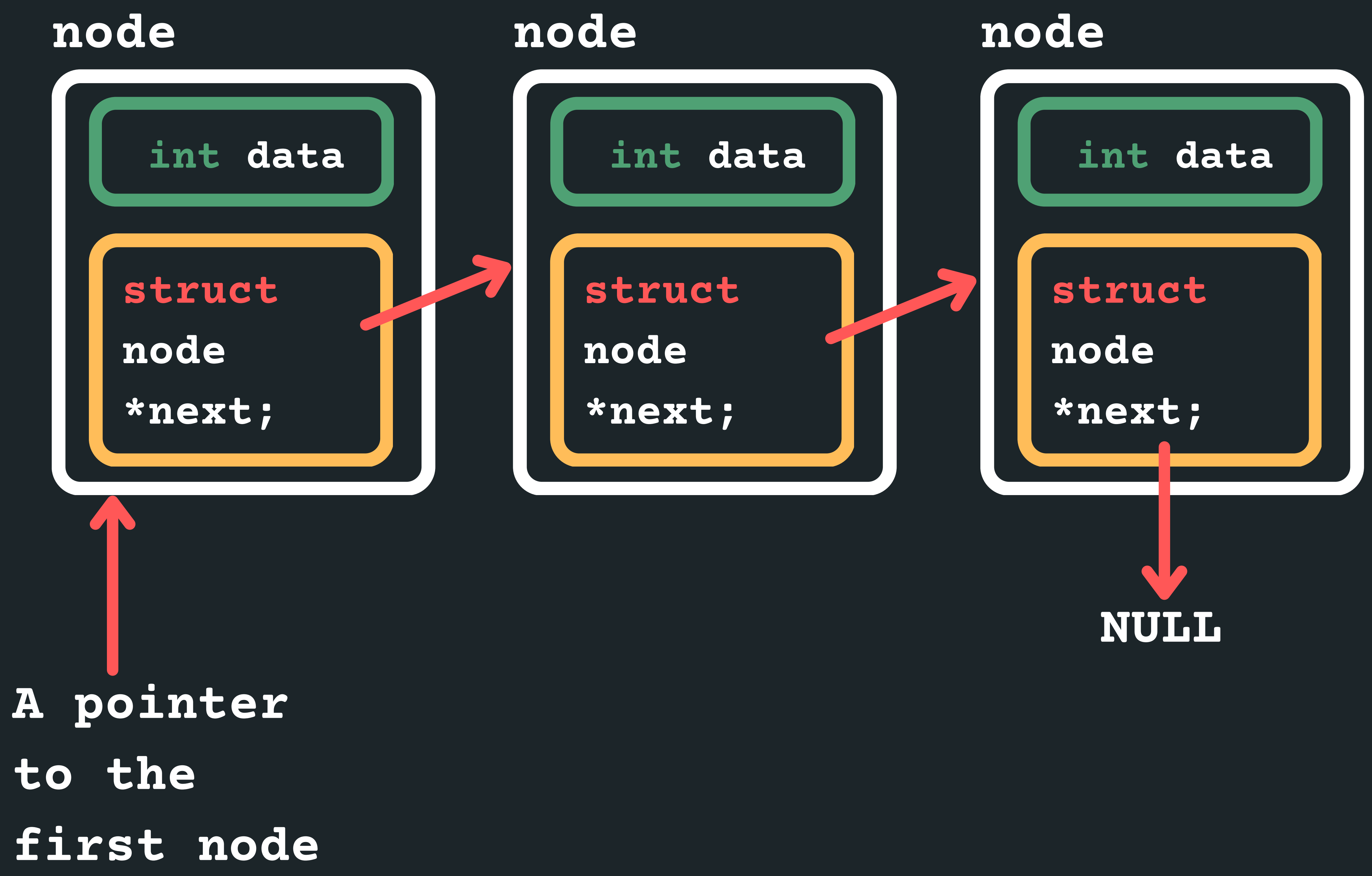
A pointer to the first node (not a node itself, but has the memory address of where the first node is!

- How do I know when my list is finished?

A LINKED LIST IS MADE UP OF MANY NODES

THE NODES ARE LINKED TOGETHER (A SCAVENGER HUNT OF POINTERS)

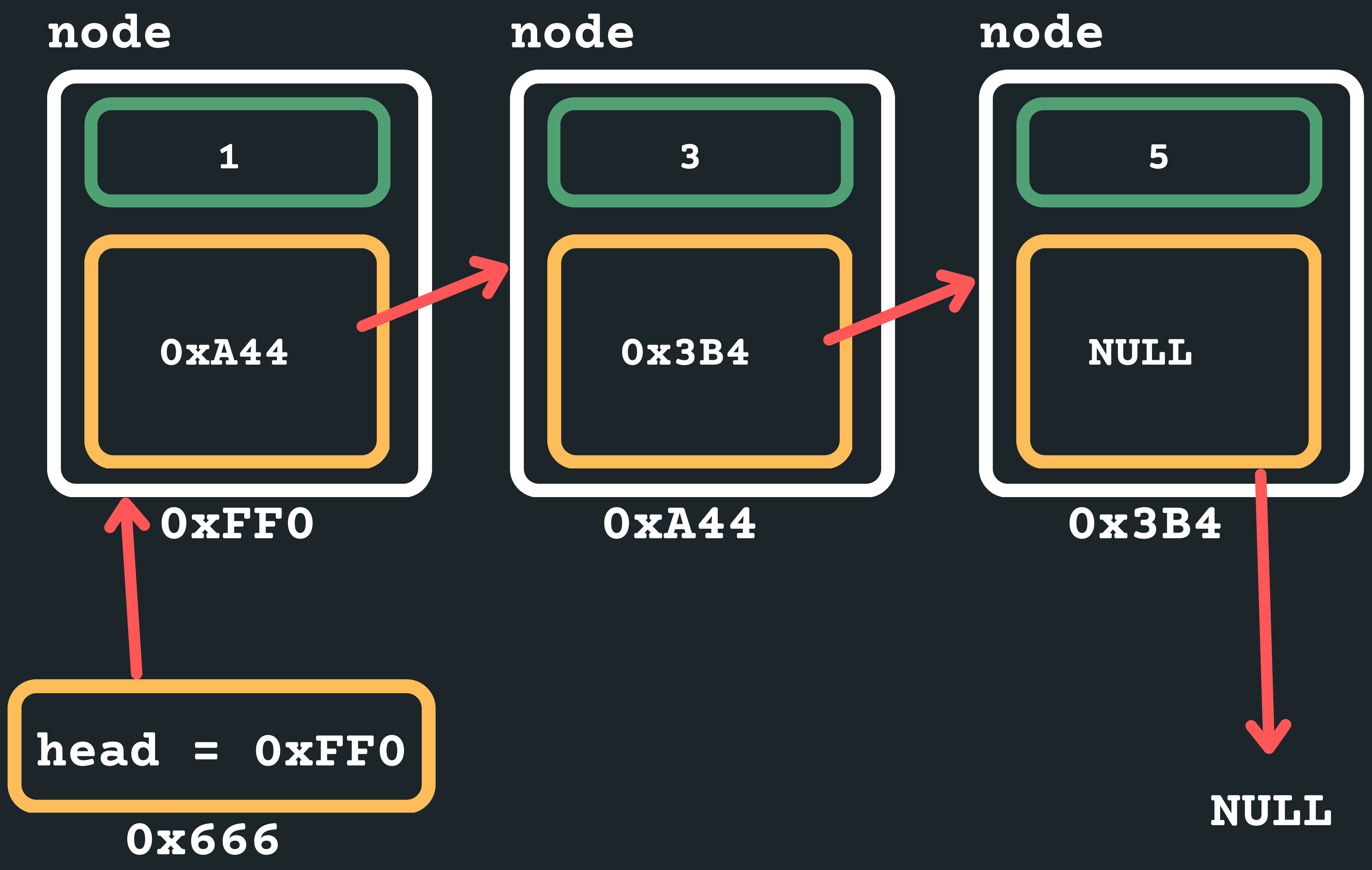
- Pointing to a NULL at the end!



A LINKED LIST IS MADE UP OF MANY NODES

THE NODES ARE LINKED TOGETHER (A SCAVENGER HUNT OF POINTERS)

- For example, a list with: 1, 3, 5



A LINKED LIST

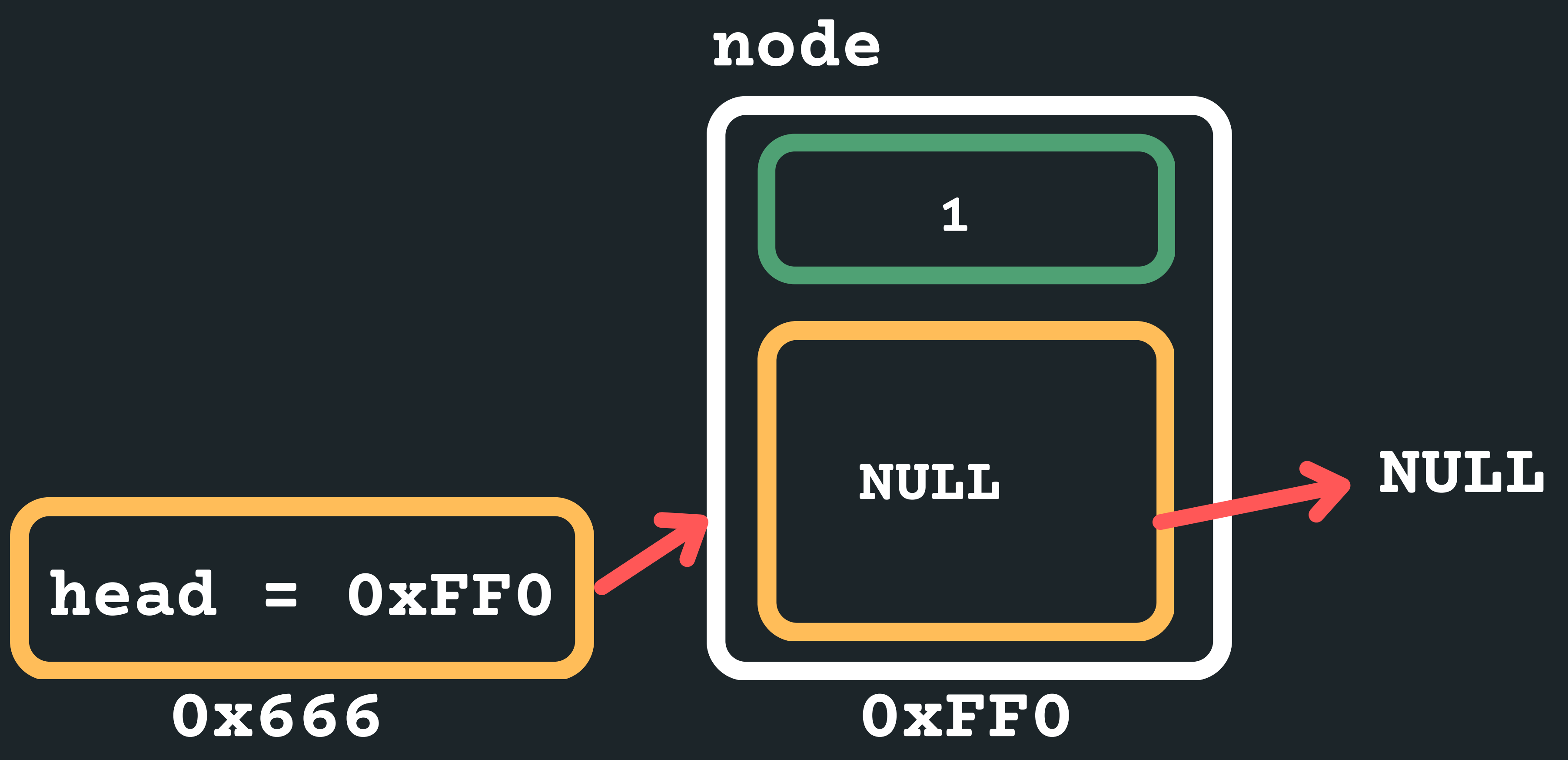
HOW DO WE CREATE ONE AND INSERT INTO IT?

- In order to create a linked list, we would need to
  - Define struct for a node,
  - A pointer to keep track of where the start of the list is and
  - A way to create a node and then connect it into our list…

A LINKED LIST

HOW DO WE CREATE ONE AND INSERT INTO IT?

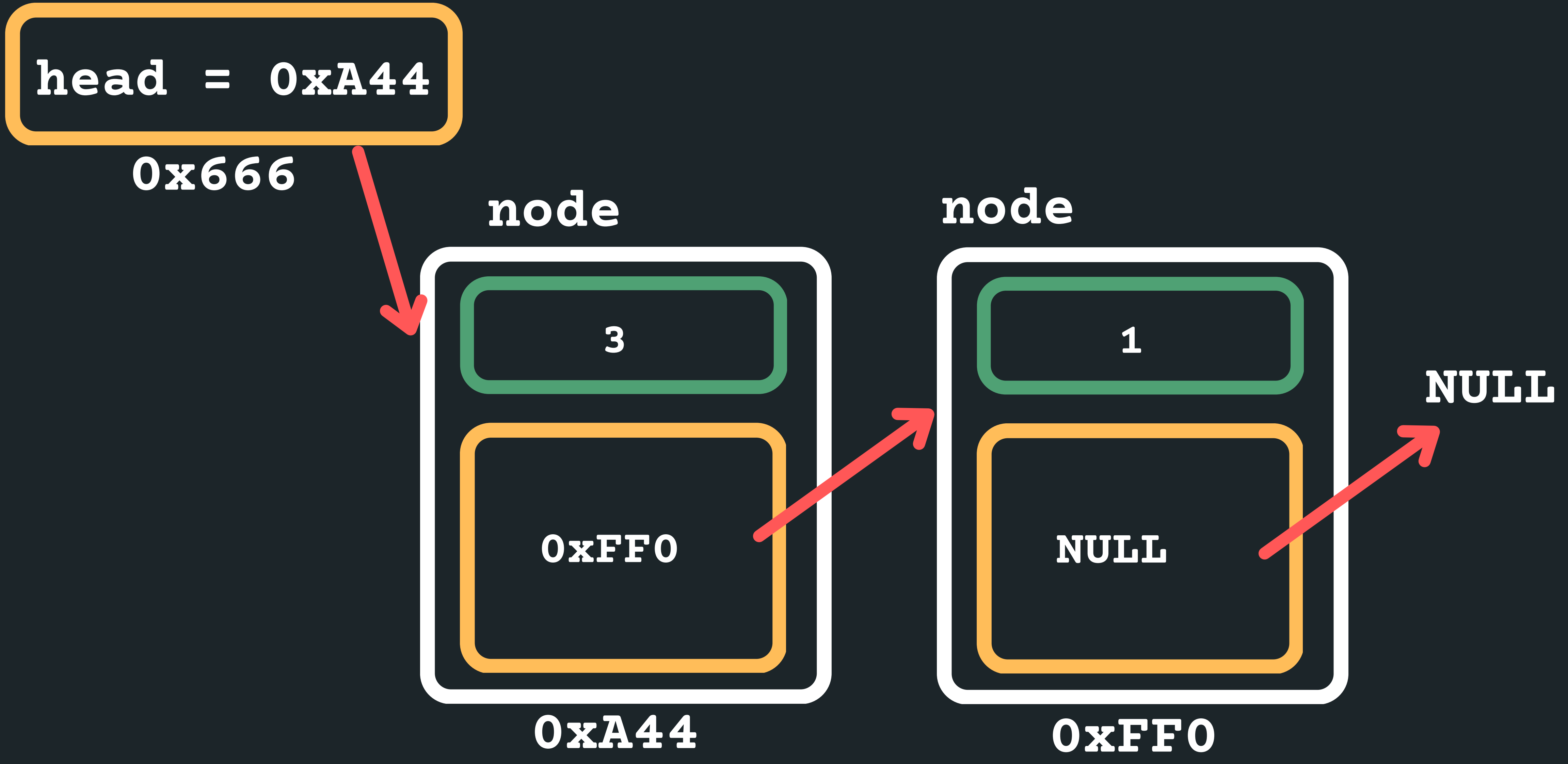
- Let's say we wanted to create a linked list with 5, 3, 1
  - Let's create the first node to start the list!
  - A pointer to keep track of where the start of the list is and by default the first node of the list
  - It will point to NULL as there are no other nodes in this list.



A LINKED LIST

HOW DO WE CREATE ONE AND INSERT INTO IT?

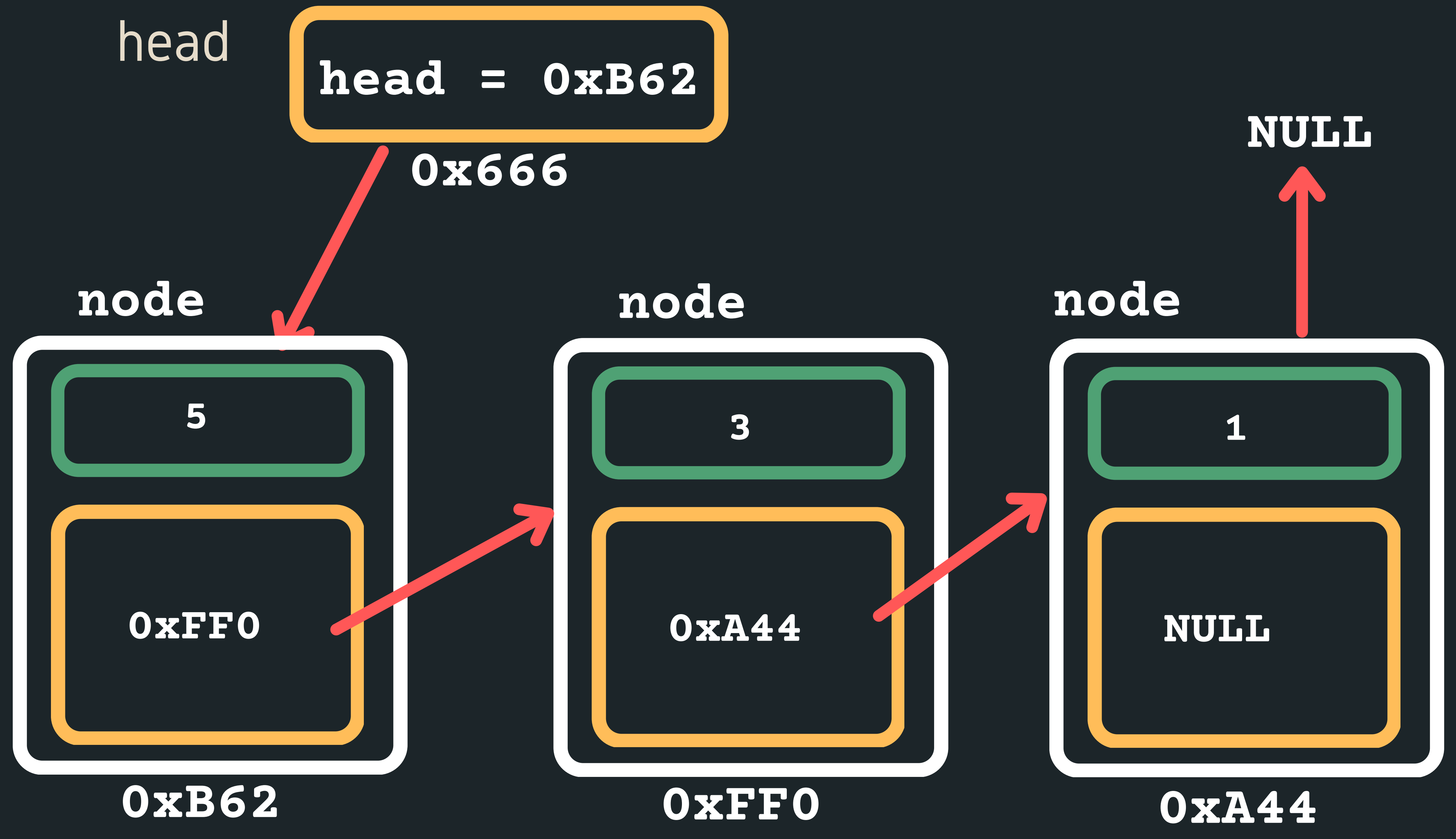
- Create the next node to store 3 into (you need memory)
- Assign 3 to data
- and insert it at the beginning so the head would now point to it and the new node would point to the old head



A LINKED LIST

HOW DO WE CREATE ONE AND INSERT INTO IT?

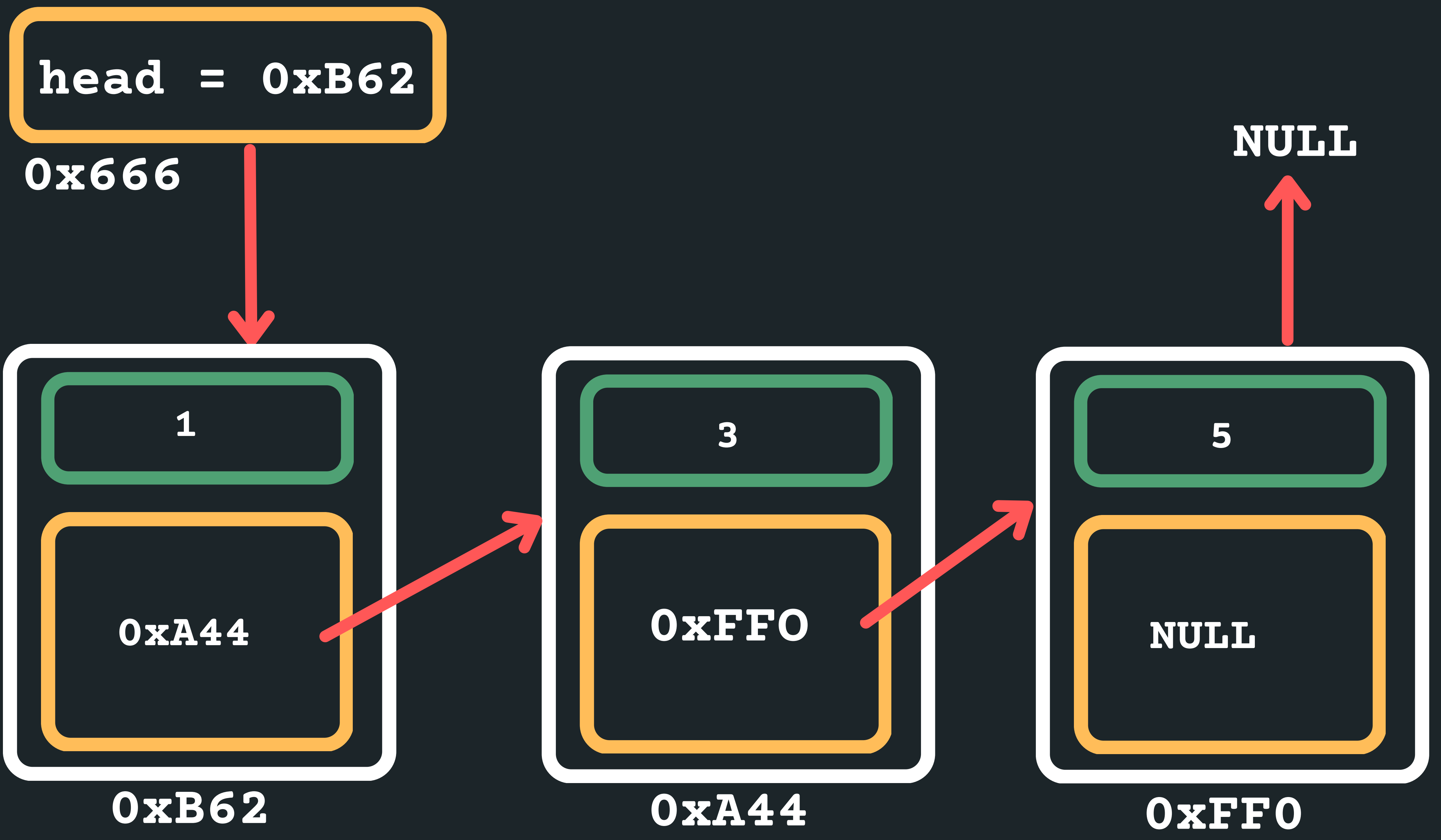
- Create the next node to store 5 into (you need memory)
- Assign 5 to data
- and insert it at the beginning so the head would now point to it and the new node would point to the old head



LINKED LISTS

INSERTING

- Where can I insert in a linked list?
  - At the head
  - Between any two nodes that exist
  - After the tail as the last node



A LINKED LIST

PUTTING IT ALL TOGETHER IN CODE

1. Define our struct for a node
2. A pointer to keep track of where the start of the list is:
   - The pointer would be of type struct node, because it is pointing to the first node
   - The first node of the list is often called the 'head' of the list (last element is often called the 'tail')
3. A way to create a node and then connect it into our list...
   - Create a node by first creating some space for that node (malloc)
   - Initialise the data component on the node
   - Initialise where the node is pointing to
4. Make sure last node is pointing to NULL

SO TRAVERSING A LINKED LIST...

- The only way we can make our way through the linked list is like a scavenger hunt, we have to follow the links from node to node (sequentially! we can't skip nodes)
- We have to know where to start, so we need to know the head of the list
- When we reach the NULL pointer, it means we have come to the end of the list.

LINKED LISTS

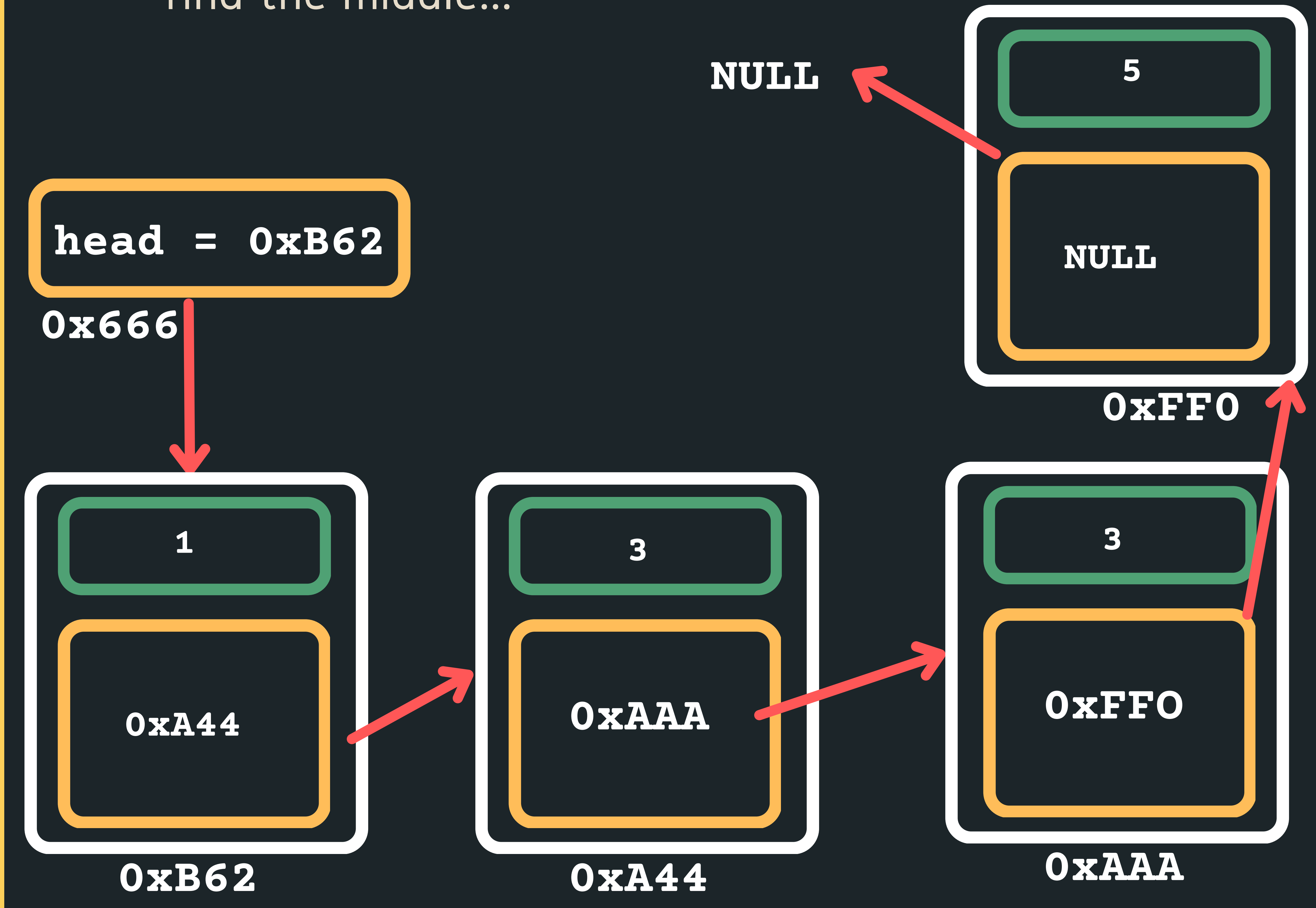
INSERTING

- You should always consider and make sure your solution works:
    - Inserting into an empty list
    - Inserting at the head of the list
    - Inserting after the first node if there is only one node
    - ...
- Draw a diagram!!!! It will allow you to easily see what are some potential pitfalls

LINKED LISTS

INSERT IN THE MIDDLE

- Let's consider an easy case to insert in the middle, find the size of the list and then divide that by 2 to find the middle...

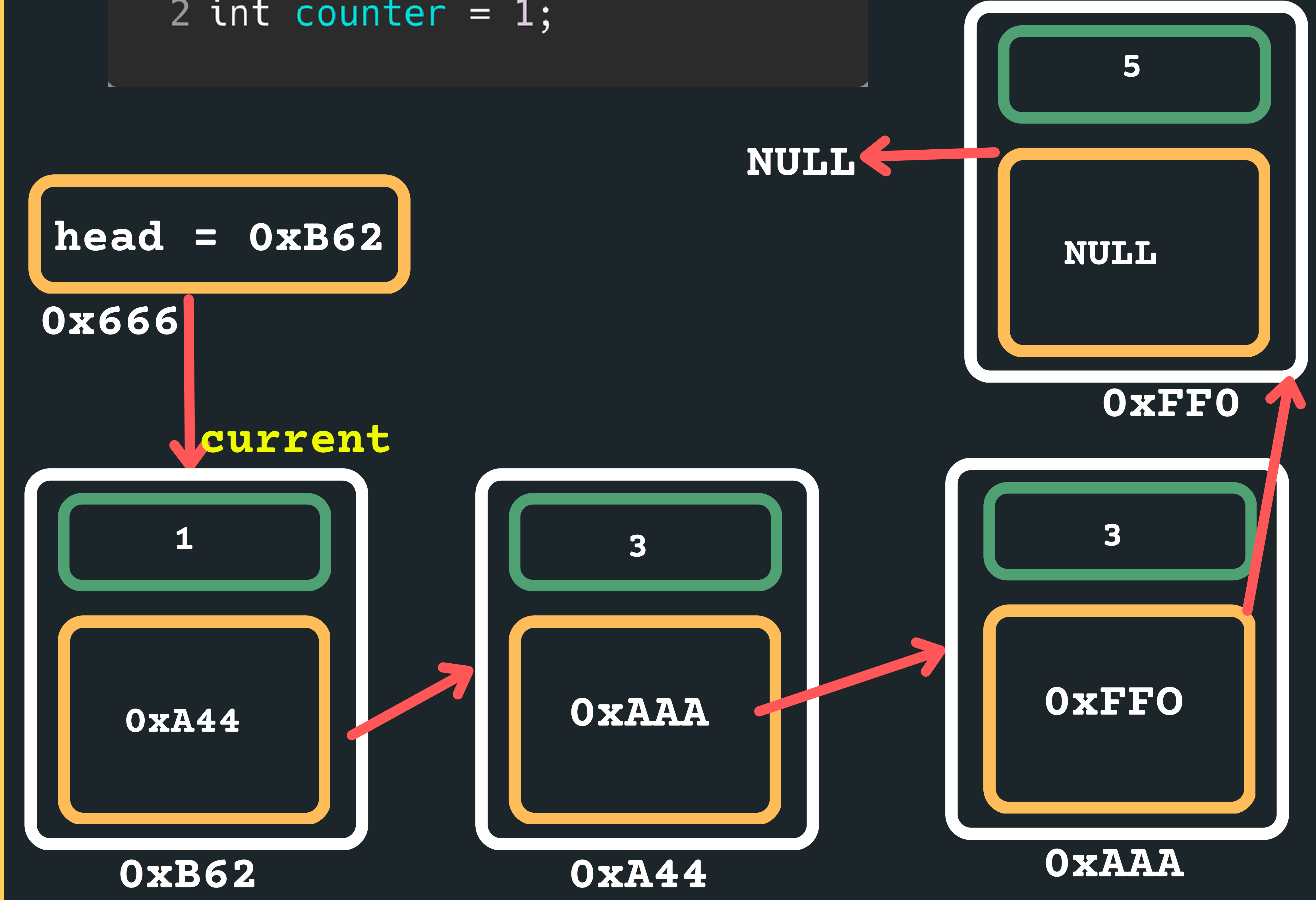


LINKED LISTS

INSERT IN THE MIDDLE

- Move through the list to get to the second node

```
struct node *current = head;
int counter = 1;
```

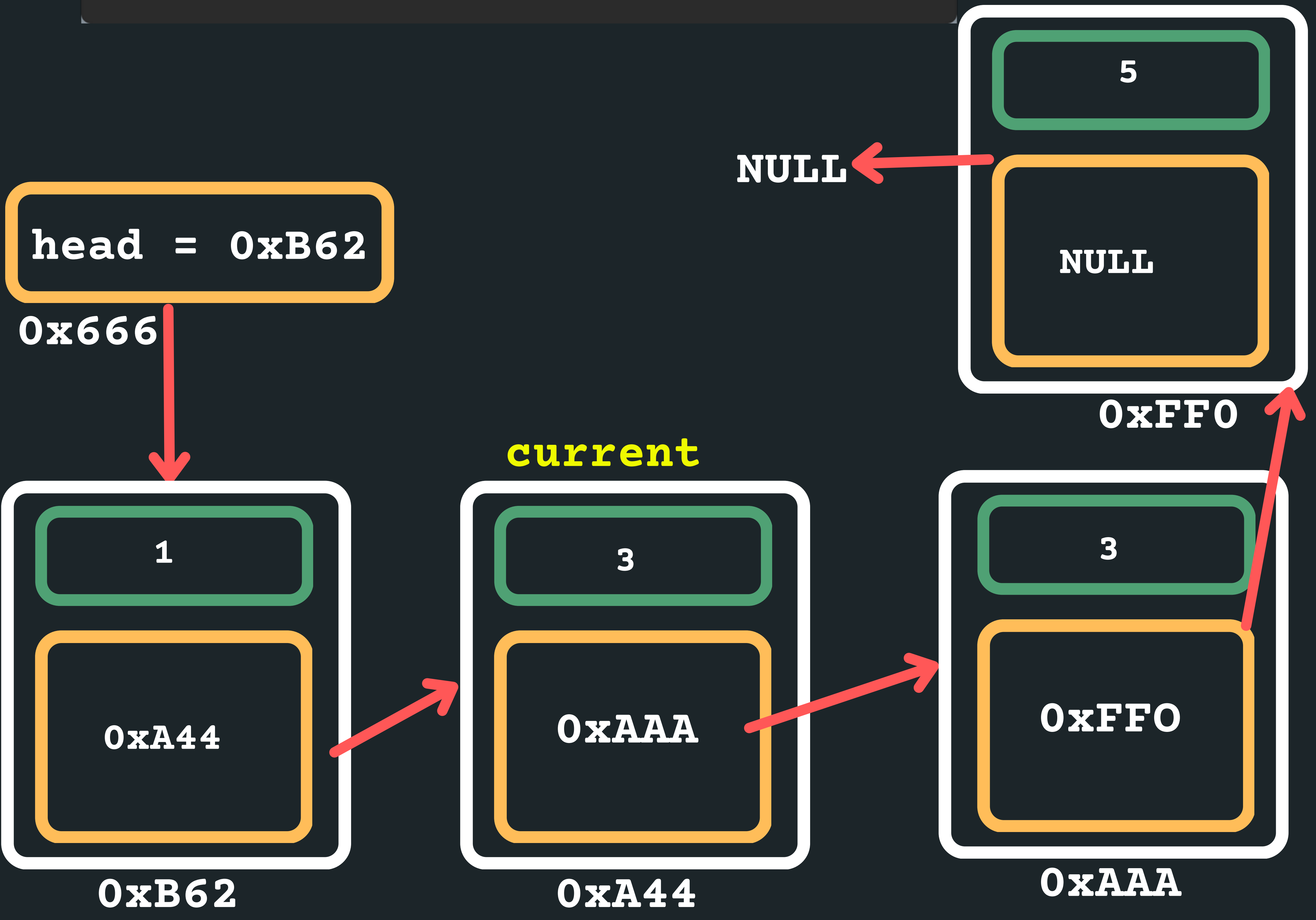


LINKED LISTS

INSERT IN THE MIDDLE

- Move through the list to get to the second node

```
while (counter != size_linked_list/2) {
    current = current->next;
}
```

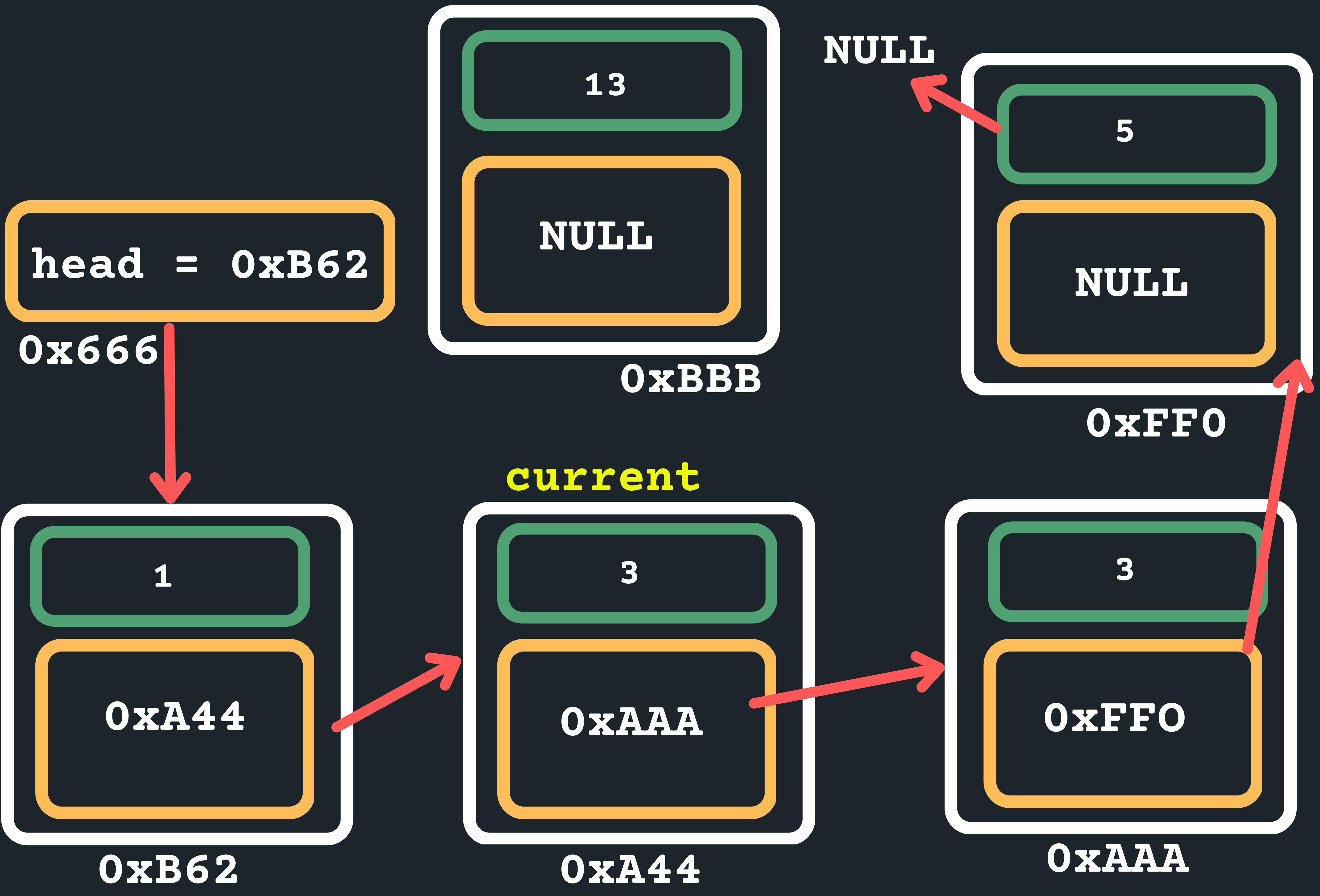


LINKED LISTS

INSERT IN THE MIDDLE

- Make a new node to insert

```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 13 //Example data!
new_node->next = NULL;
```

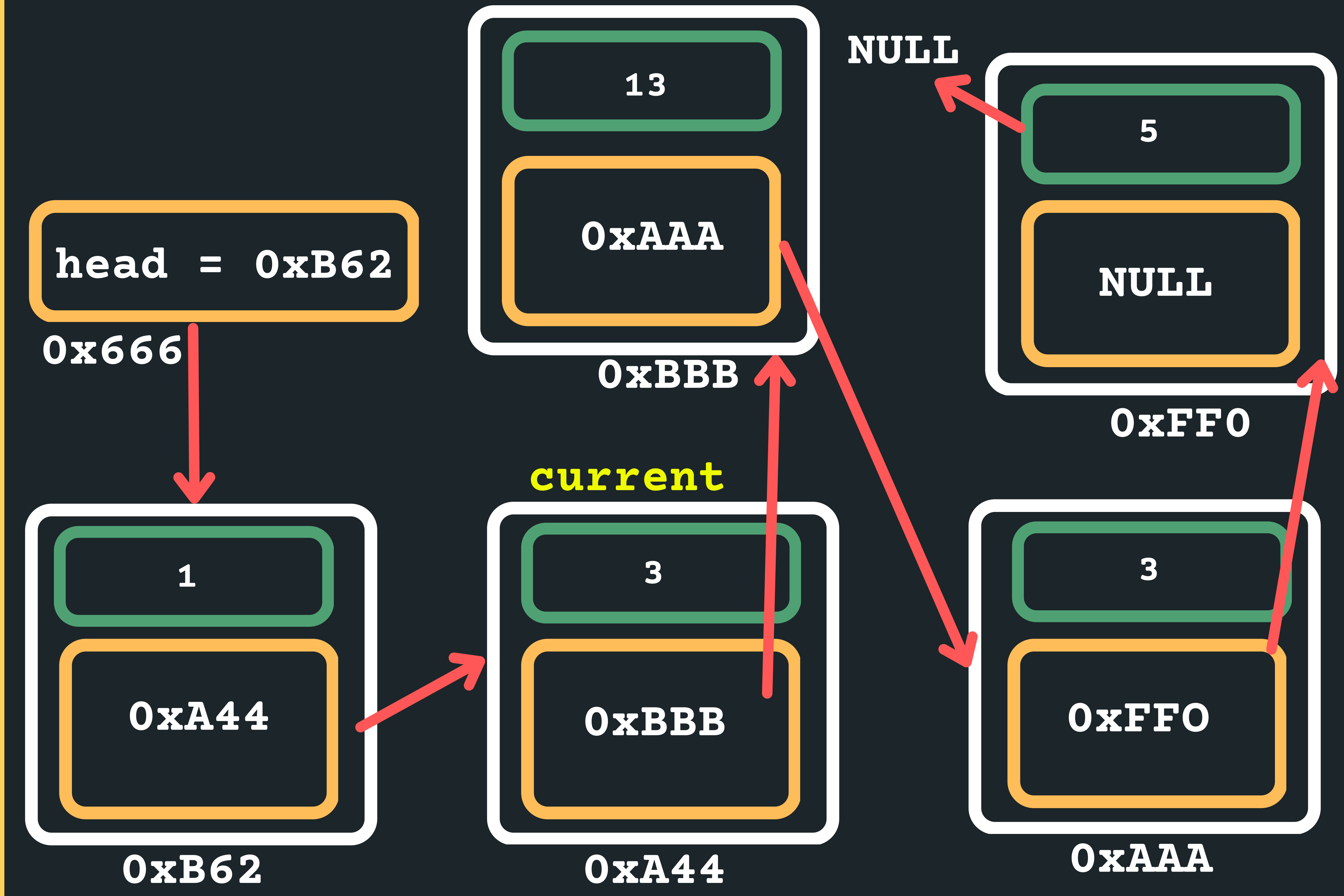


LINKED LISTS

INSERT IN THE MIDDLE

- Connect the node in between the two nodes

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



LET'S INSERT IN THE MIDDLE?

- Great!
- Let’t think of some conditions that may break this …
  - What happens if it is an empty list?
  - What happens if there is only one item in the list?
- How can we safeguard?

LET'S INSERT AFTER A PARTICULAR NODE?

- What about inserting in order into an ordered list? Let’s try that as a problem and then walk through the code...
- So for example, I have a list with 1, 3, 5 and I wanted to insert a 4 into this list - it would go after 3 ...
  - Let’s try it!

LINKED LISTS

INSERTING A NODE

- In all instances, we follow a similar structure of what to do when inserting a node. Please draw a diagram for yourself to really understand what you are inserting and the logic of inserting in a particular way.
- To insert a node in a linked list:
  - Find where you want to insert the node (stop at the node after which you want to insert)
  - Malloc a new node for yourself
  - Point the new_node->next to the current->next
  - Change the current->next to point to the new node
  - Consider possible edge cases, empty list, inserting at the head with only one item, etc etc.

BREAK TIME...

Can you determine how many times do the minute and hour hands of a clock overlap in a day?

LINKED LISTS

DELETING

- Where can I delete in a linked list?
  - Nowhere (if it is an empty list - edge case!)
  - At the head (deleting the head of the list)
  - Between any two nodes that exist
  - At the tail (last node of the list)

LINKED LISTS

DELETING EMPTY LIST

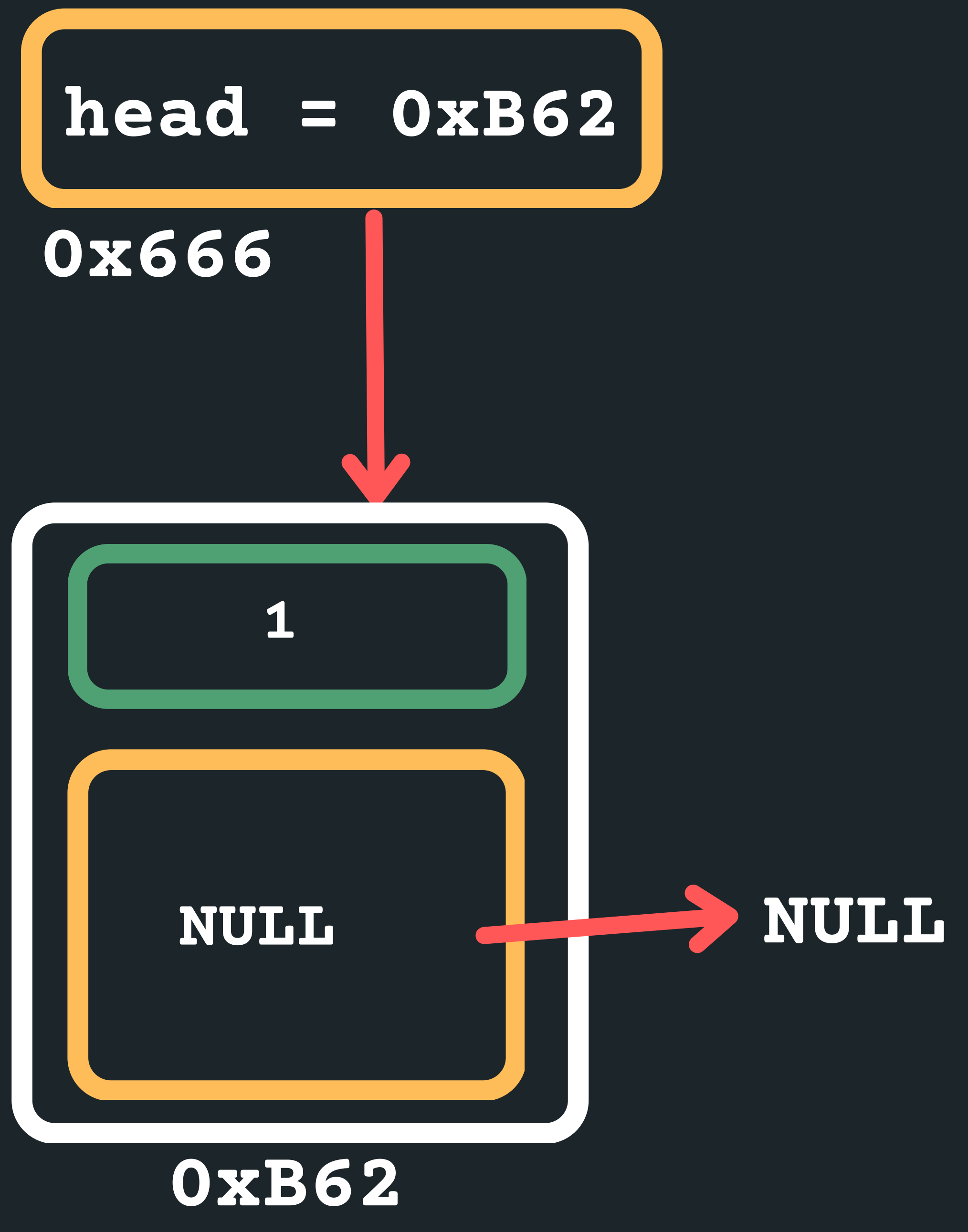
- Deleting when nowhere! (it is an empty list)
  - Check if list is empty
  - If it is - return NULL

```
struct node *current = head;
if (current == NULL){
    return NULL;
}
```

LINKED LISTS

DELETING ONE ITEM

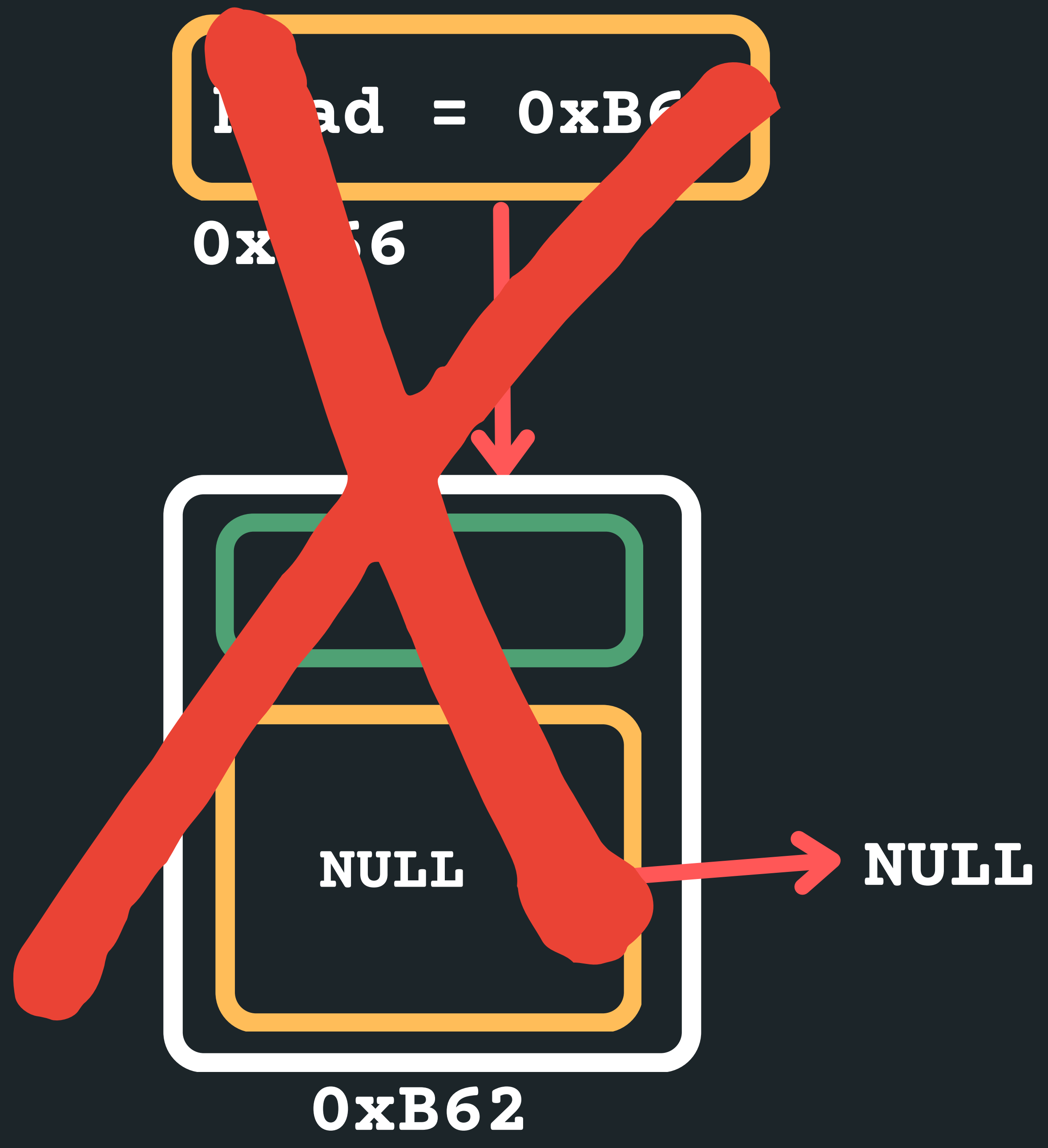
- Deleting when there is only one item in the list



LINKED LISTS

DELETING ONE ITEM

- Deleting when there is only one item in the list
  - free the head!

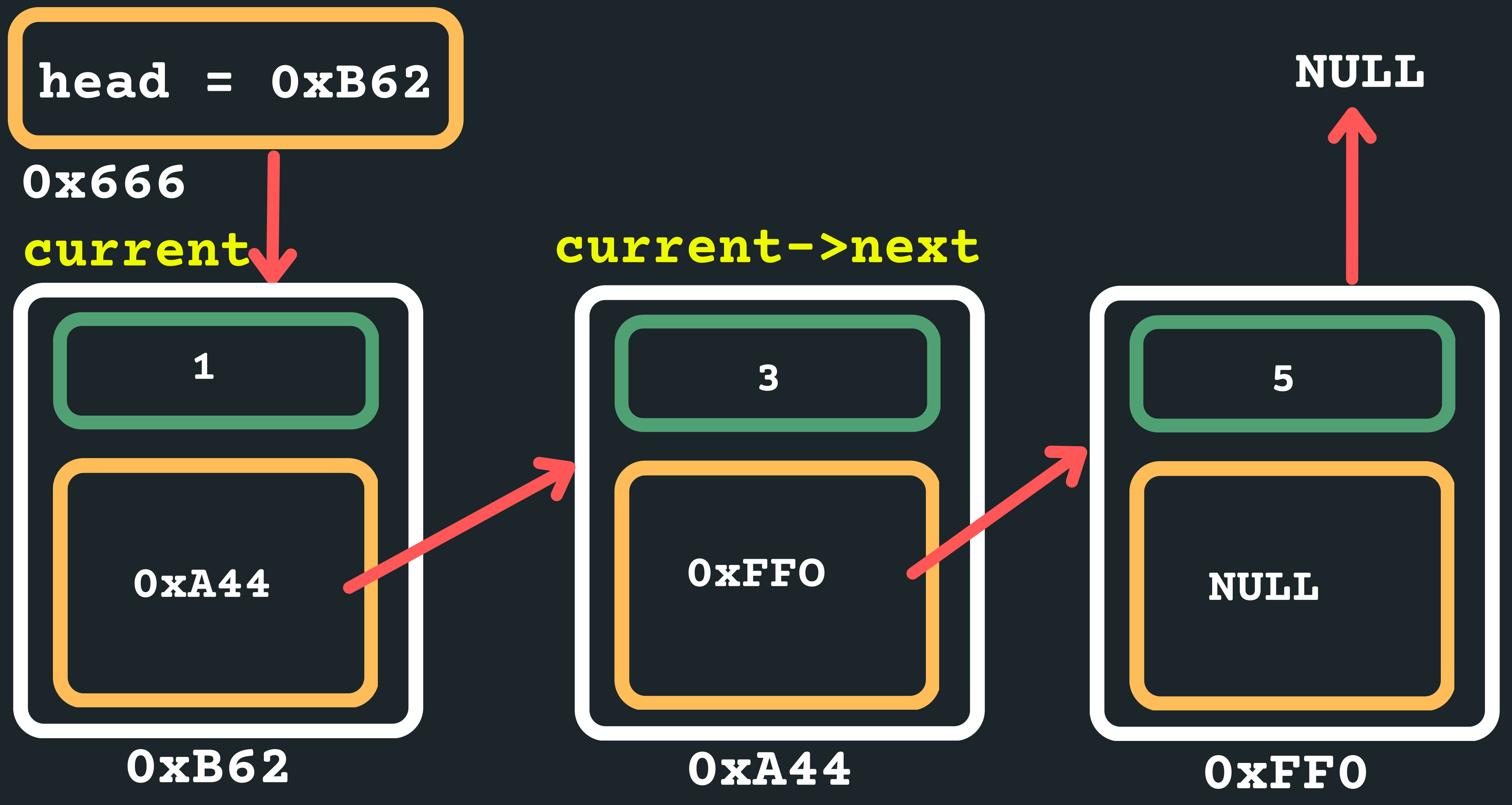


LINKED LISTS

DELETING THE HEAD WITH OTHER ITEMS

- Deleting when at the head of the list with other items in the list
    - Find the node that you want to delete (the head)

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

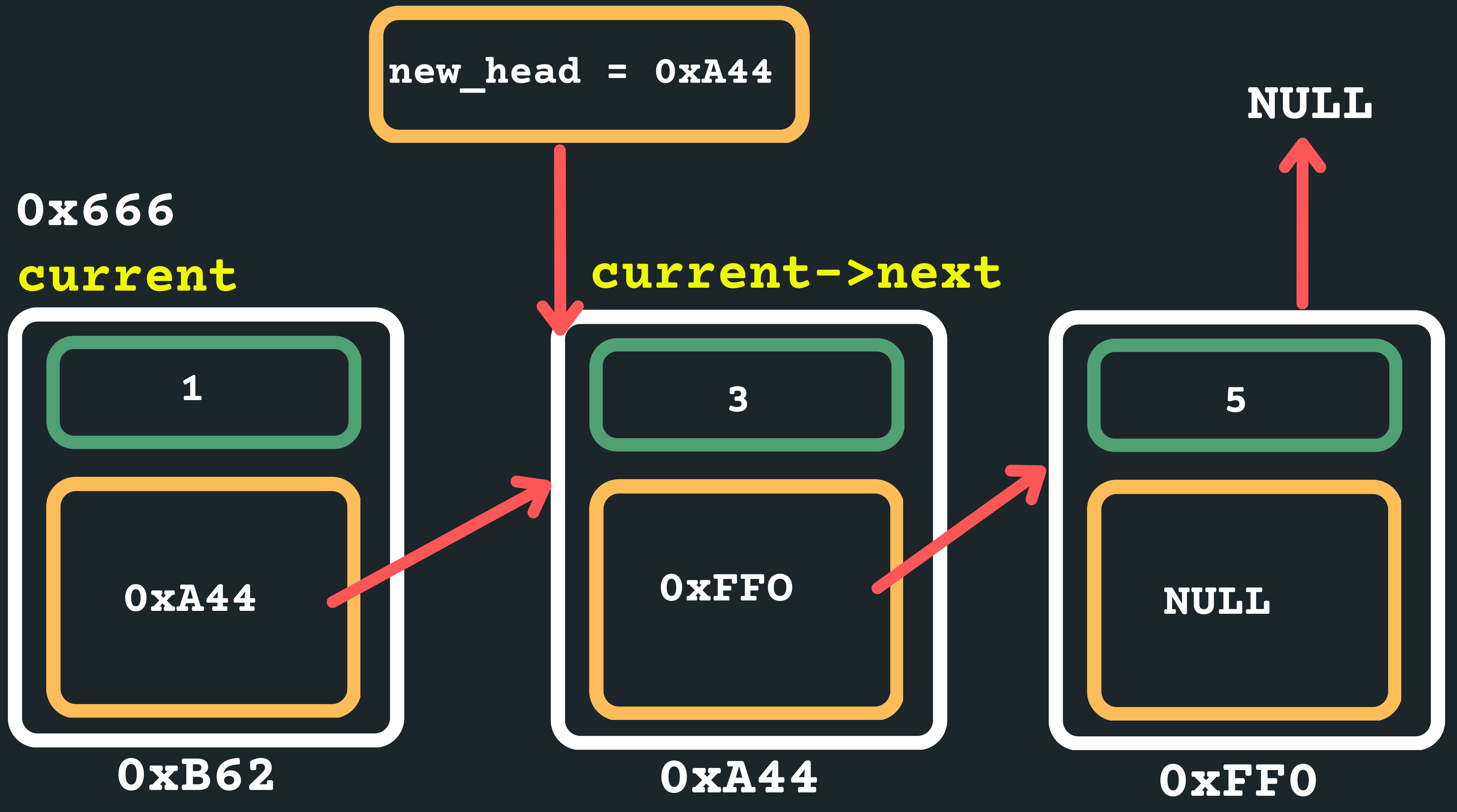


LINKED LISTS

DELETING THE HEAD WITH OTHER ITEMS

- Deleting when at the head of the list with other items in the list
  - Point the head to the next node

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

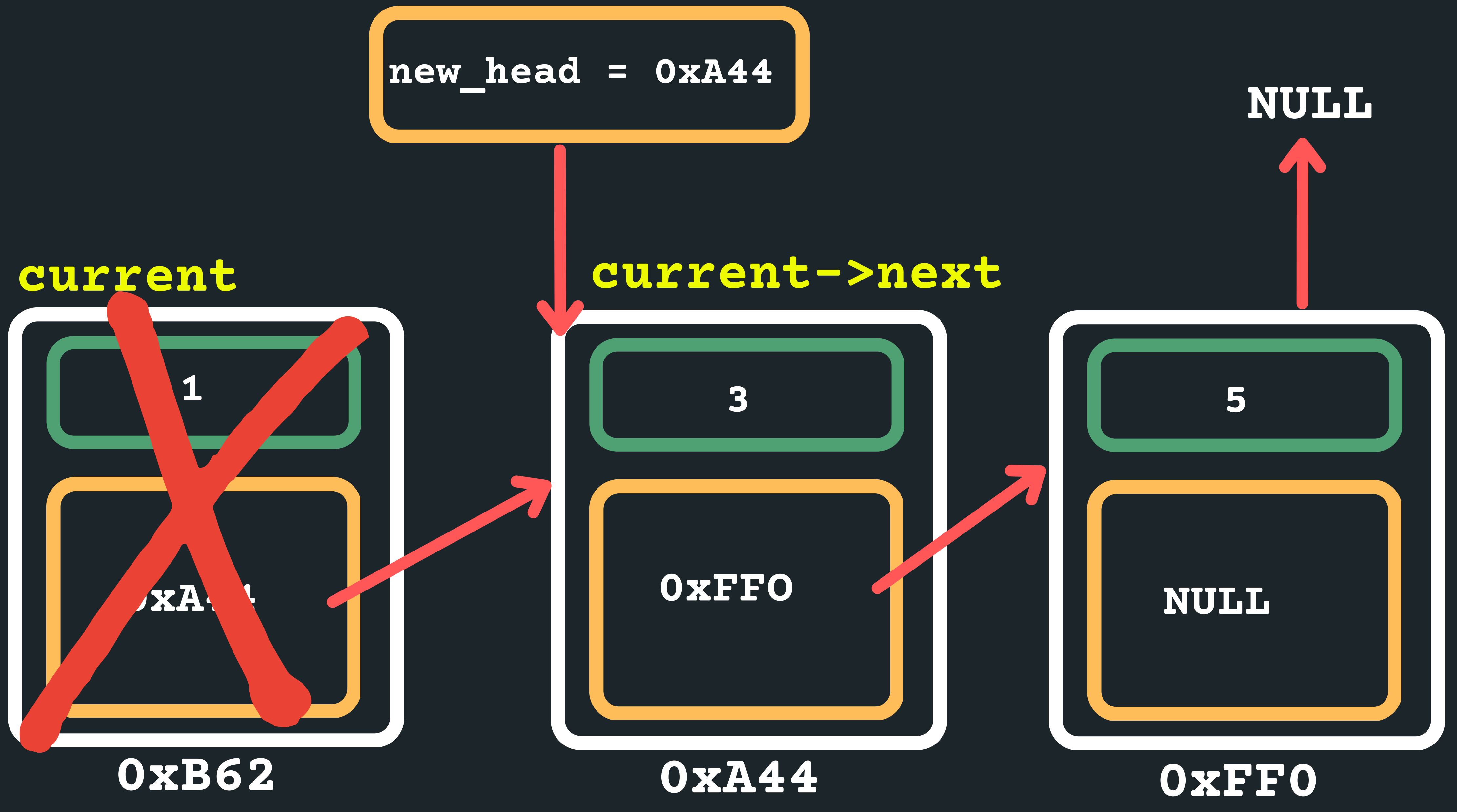


LINKED LISTS

DELETING THE HEAD WITH OTHER ITEMS

- Deleting when at the head of the list with other items in the list
    - Delete the current head

```
free(current);
```

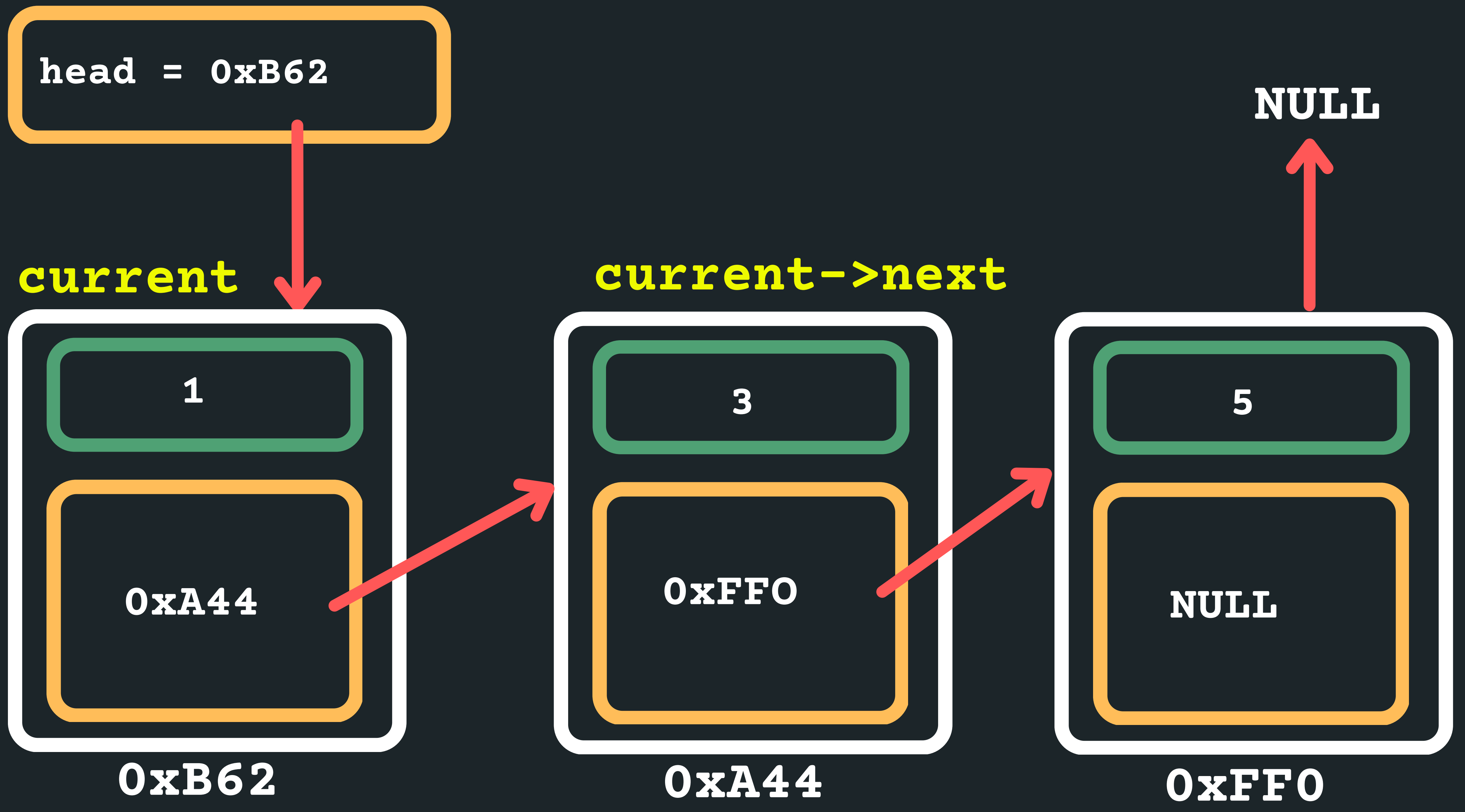


LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
  - Set the head to a variable current to keep track of the loop

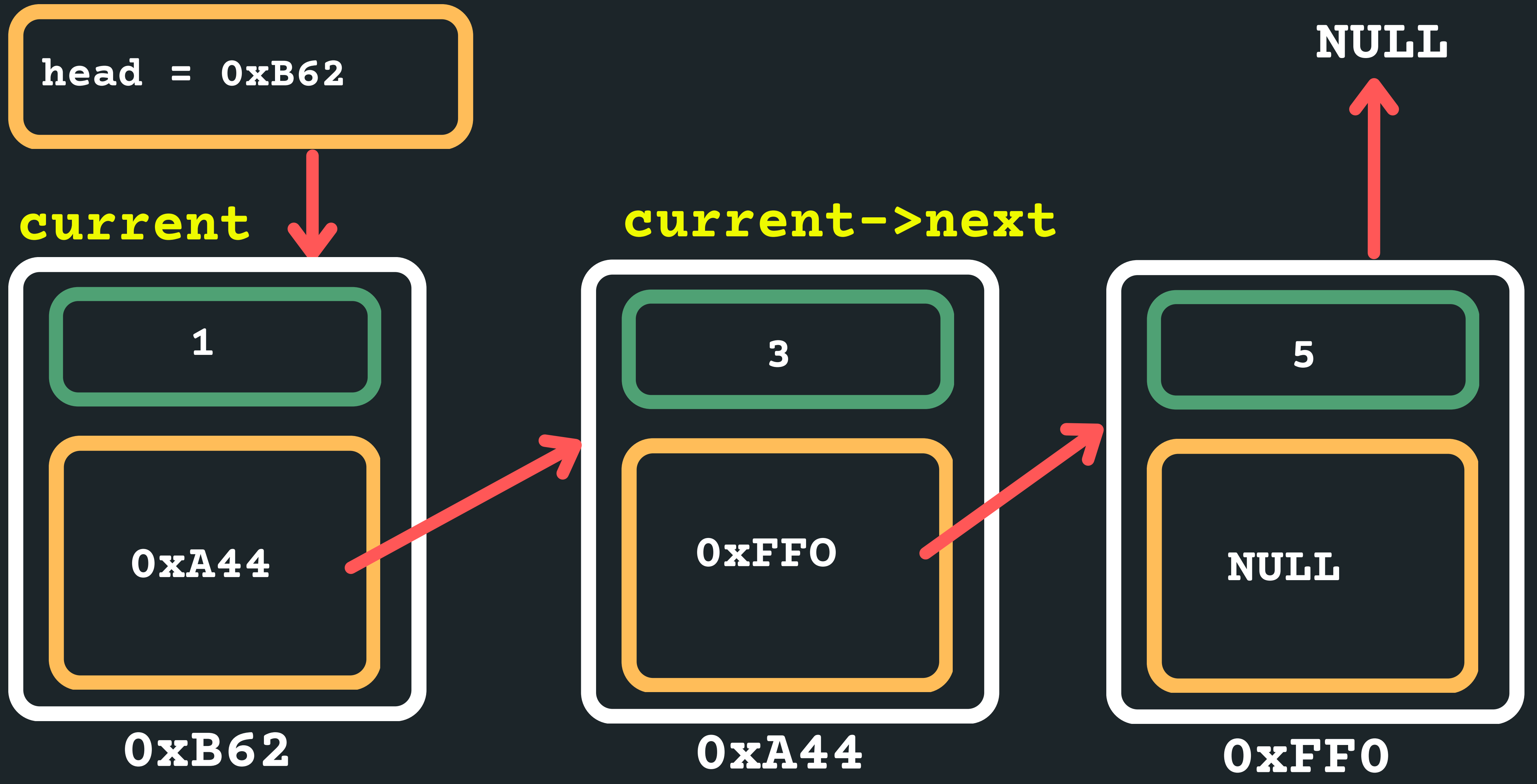
```
struct node *current = head
```



LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
  - Loop until you find the right node - what do we think loop until the node with 3 or the previous node? Remember that once you are on the node with 3, you have no idea what previous node was.

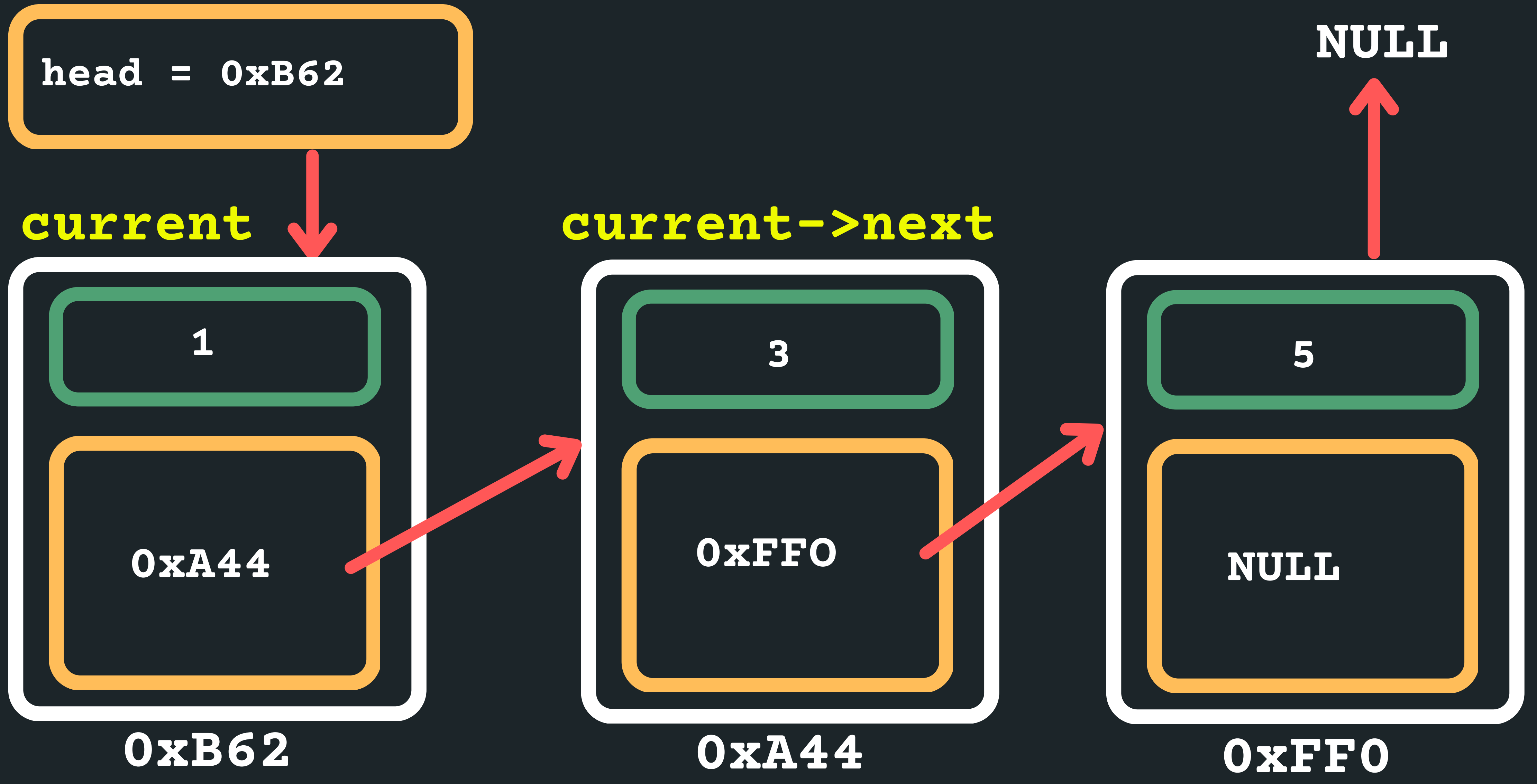


LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
  - So stop at a previous node (when the next is = 3)

```
while (current->next->data != 3){
    current = current->next;
}
```

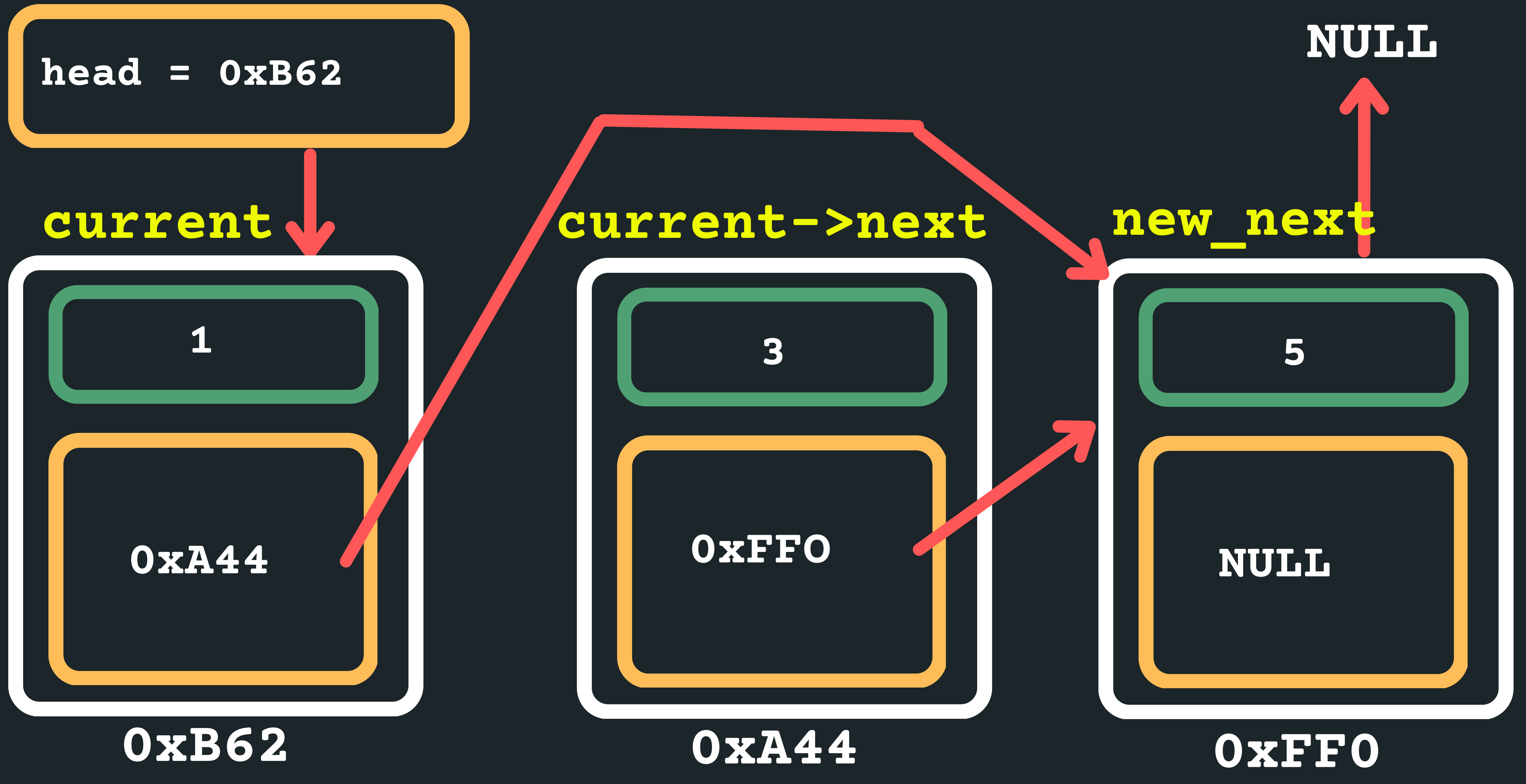


LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
  - Create new next node to store address

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

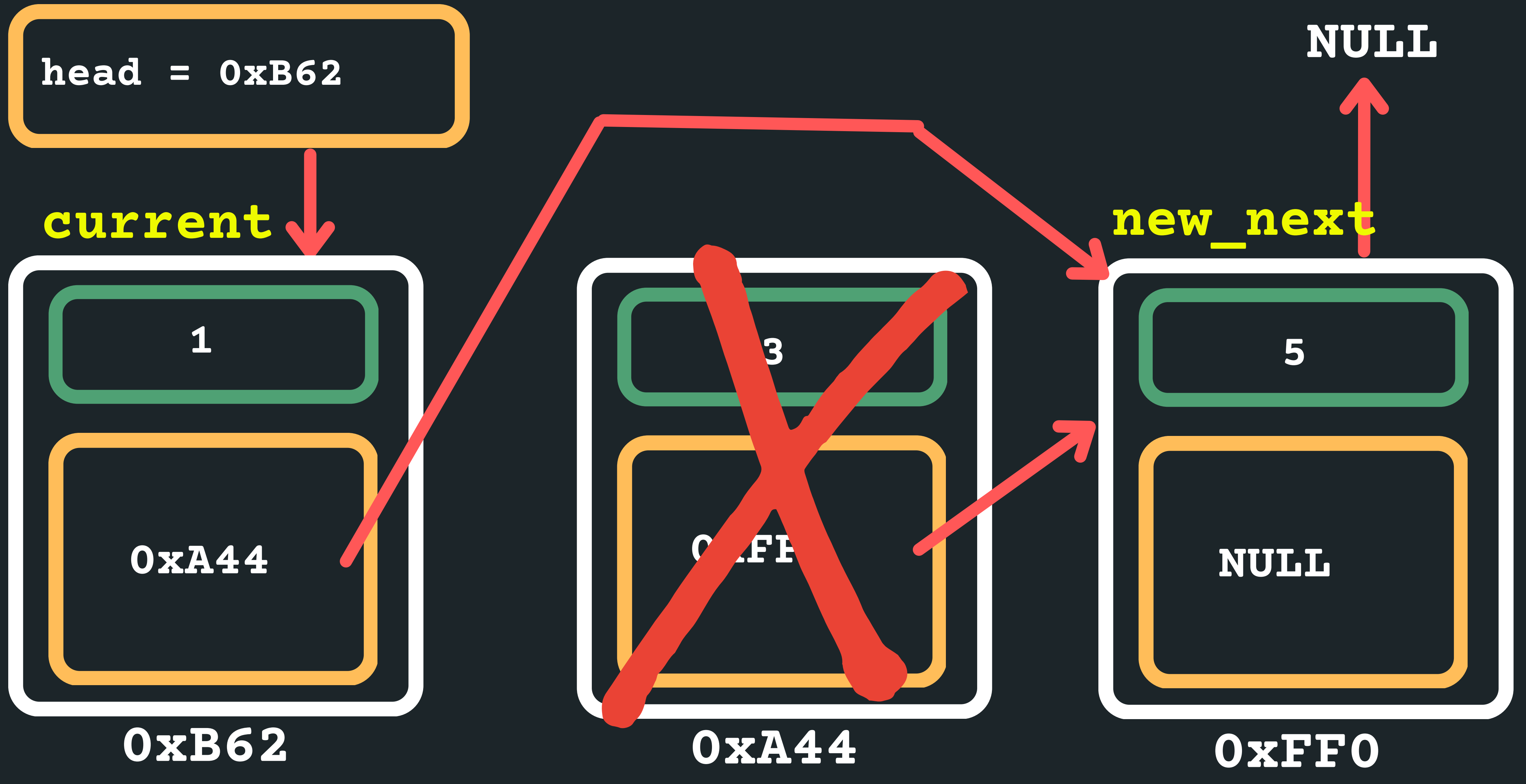


LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
    - Delete current->next

```
free(current->next);
```

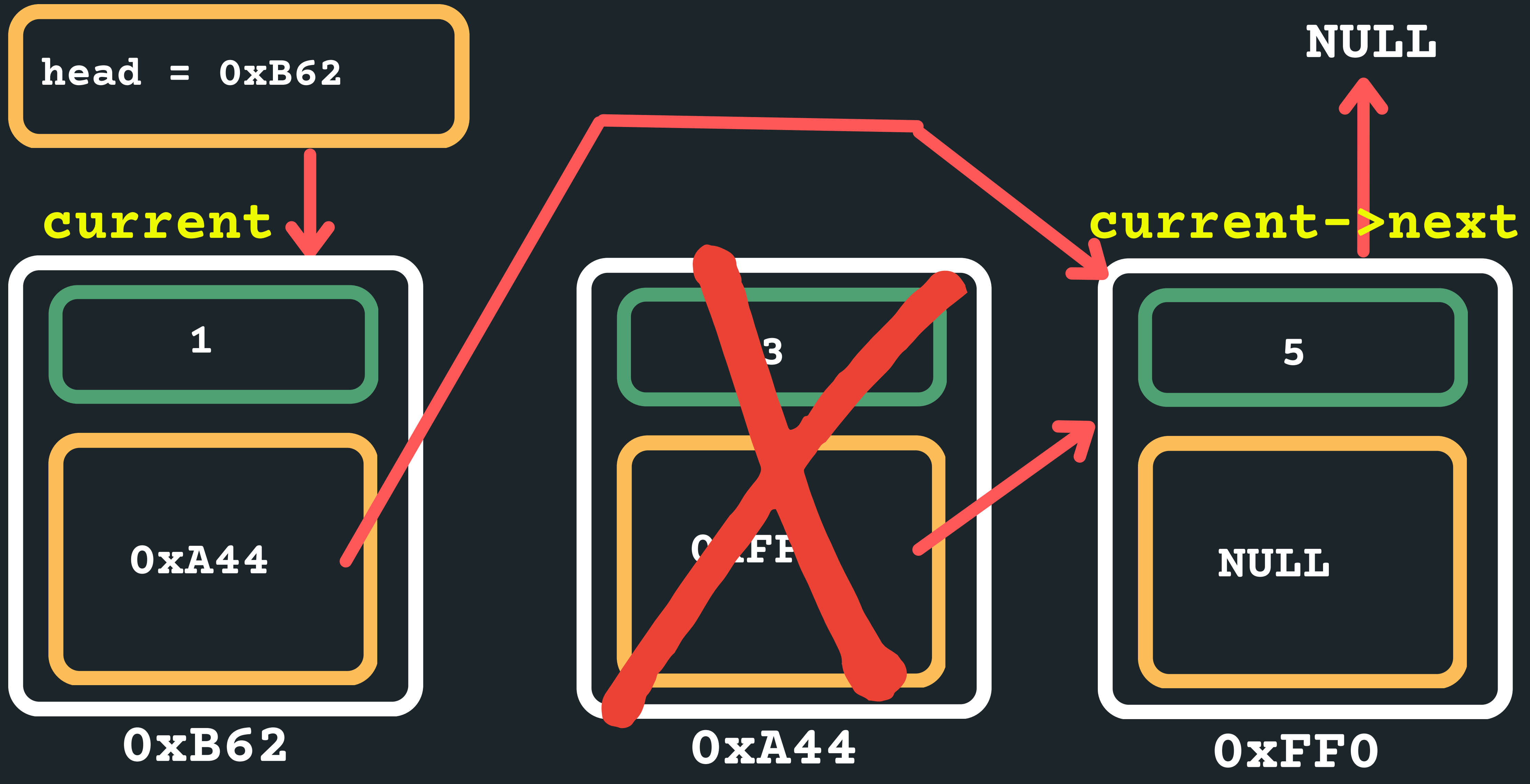


LINKED LISTS

DELETING IN MIDDLE OF TWO NODES

- Deleting when in the middle of two nodes (for example, node with 3)
  - Set the new current->next to the new_next node

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

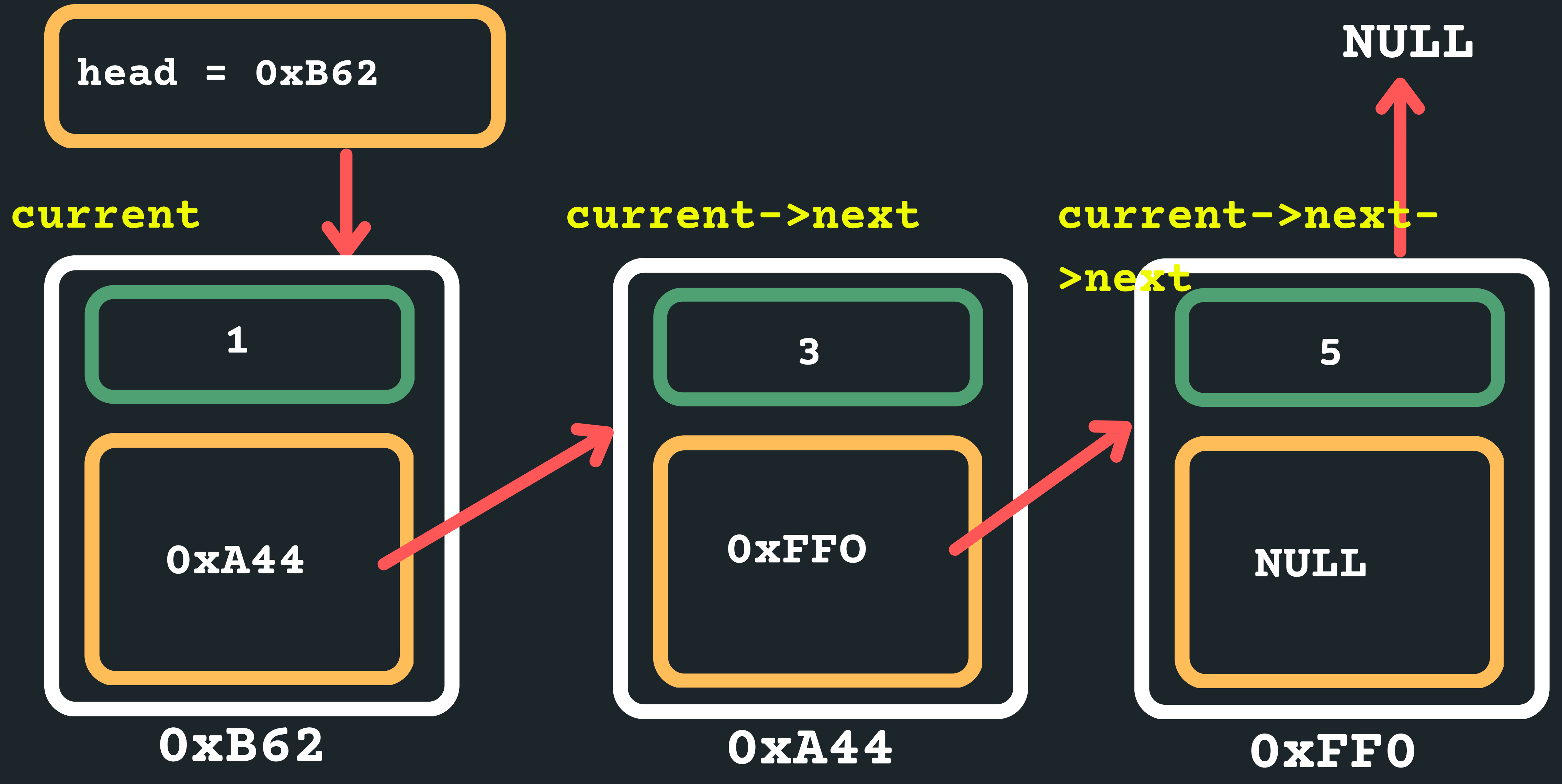


LINKED LISTS

DELETING THE TAIL

- Deleting when in the tail
    - Set the current pointer to the head of the list

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

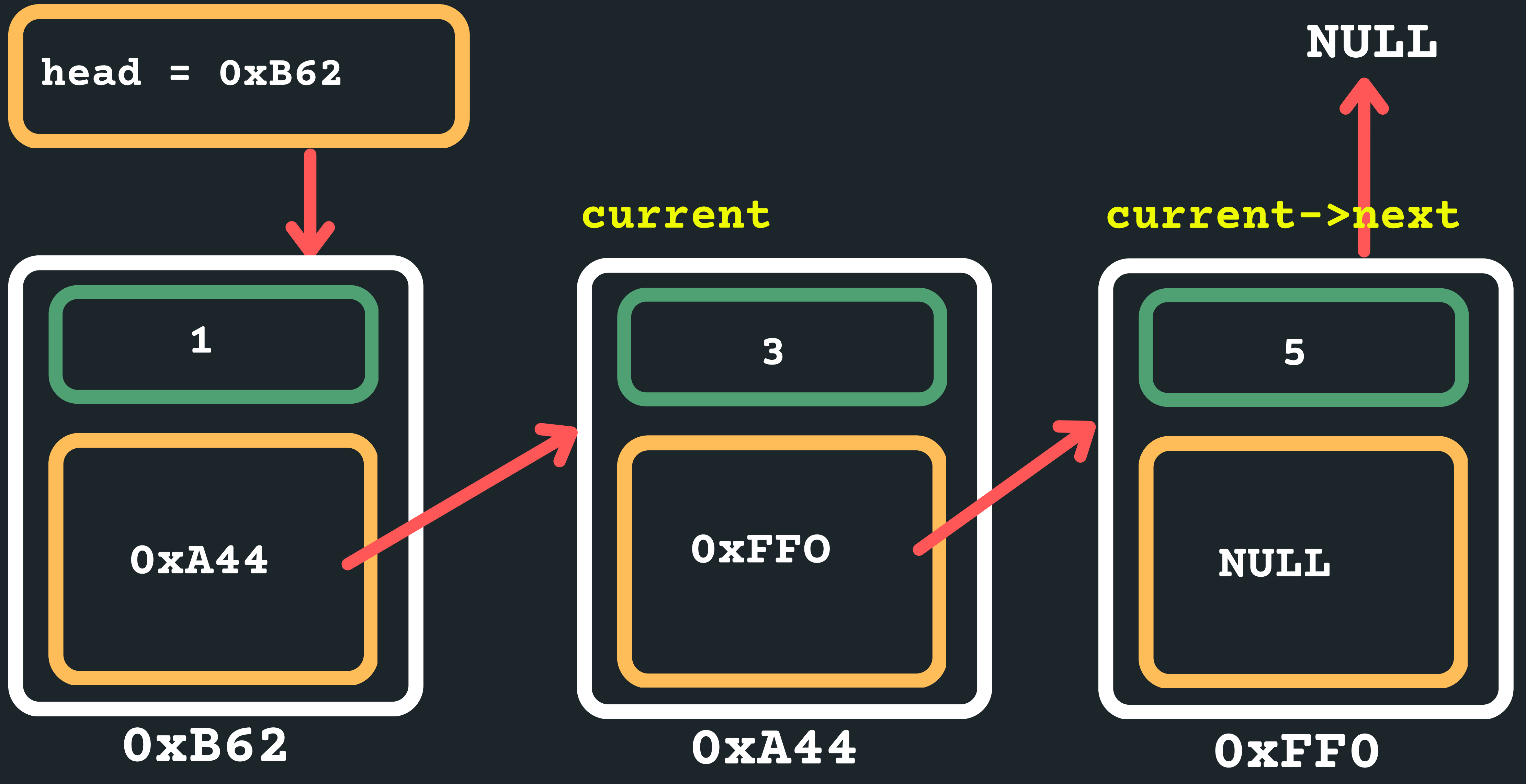


LINKED LISTS

DELETING THE TAIL

- Deleting when in the tail
  - Find the tail of the list (should I stop on the tail or before the tail?)
  - If the next is NULL than I am at the tail...

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

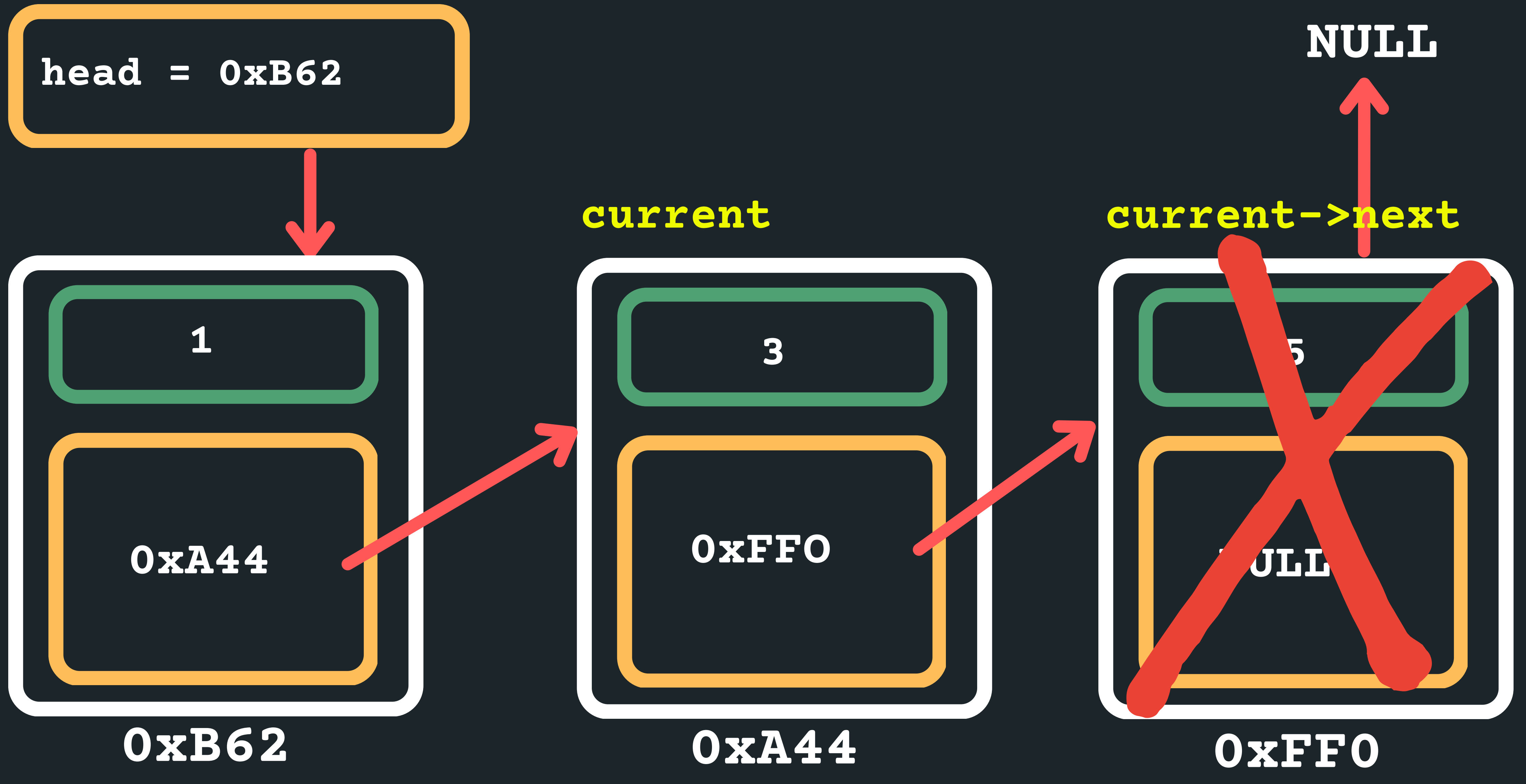


LINKED LISTS

DELETING THE TAIL

- Deleting when in the tail
    - Delete the current->next node

```
free(current->next);
```

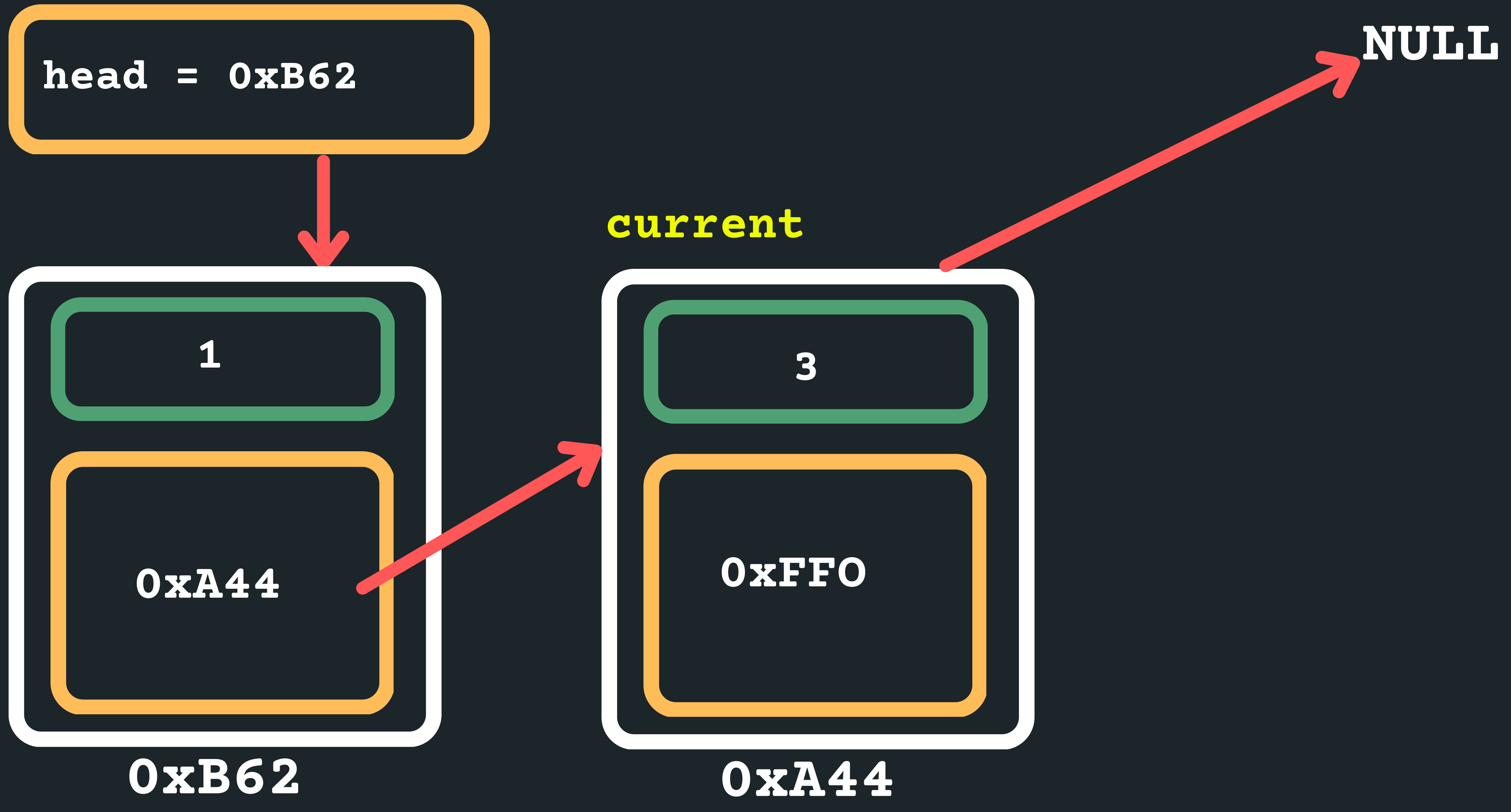


LINKED LISTS

DELETING THE TAIL

- Deleting when in the tail
    - Point my current->next node to a NULL

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



LINKED LISTS

DELETING A NODE

- In all instances, we follow a similar structure of what to do when deleting a node. Please draw a diagram for yourself to really understand what you are deleting and the logic of deleting in a particular way.
- To delete a node in a linked list:
    - Find the previous node to the one that is being deleted
    - Change the next of the previous node
    - Free the node that is to be deleted
    - Consider possible edge cases, deleting if there is nothing in the list, deleting when there is only one item in the list, deleting the head of the list, deleting the tail of the list, etc.

LINKED LISTS

DELETING A NODE

```c
struct node *delete_node (struct node *head, int data) {
    // Create a current pointer set to the head of the list
    struct node *current = head;
    // Sometimes it is helpful to keep track of a previous node
    // to the current as that means you won't lose it....
    struct node *previous = NULL; // If the current node is at head, that
                                  // means the previous node is at NULL

    // What happens if we have an empty list?
    if (current == NULL) {
        return NULL;
    } else if (current->data == data) {
    // What happens if we need to delete the item that is
    // the head of the list?
        struct node *new_head = current->next;
        free(current);
        return new_head;
        // This will return whatever was after current as the
        // new head. If there is only one node in the list and
        // it is the one to be deleted, it will capture this (NULL)
    }

    // Otherwise start looping through the list to find the data
    // 1. Find the previous node to the one you want to delete
    while (previous->next->data != data && current->next != NULL) {
        previous = current;
        current = current->next;
    }

    // 2. If the current node is the one to be deleted
    if (previous->next->data == data) {
        //point the next node to the new pointer
        previous->next = current_next;
        // 3. free the node to be deleted
        free(current);

    }
    return head;
}
```

Feedback please!

I value your feedback and use to pace the lectures and improve your overall learning experience. If you have any feedback from today's lecture, please follow the link below. Please remember to keep your feedback constructive, so I can action it and improve the learning experience.

https://www.menti.com/al6spfu1chi4

WHAT DID WE LEARN TODAY?

LINKED LISTS - INSERT ANYWHERE

linked_list.c

LINKED LISTS - DELETING

linked_list.c

REACH OUT

CONTENT RELATED QUESTIONS

Check out the forum

ADMIN QUESTIONS

cs1511@unsw.edu.au