DPST1091 / CPTG1391

Introduction to Programming

Week 9 – Lecture 2

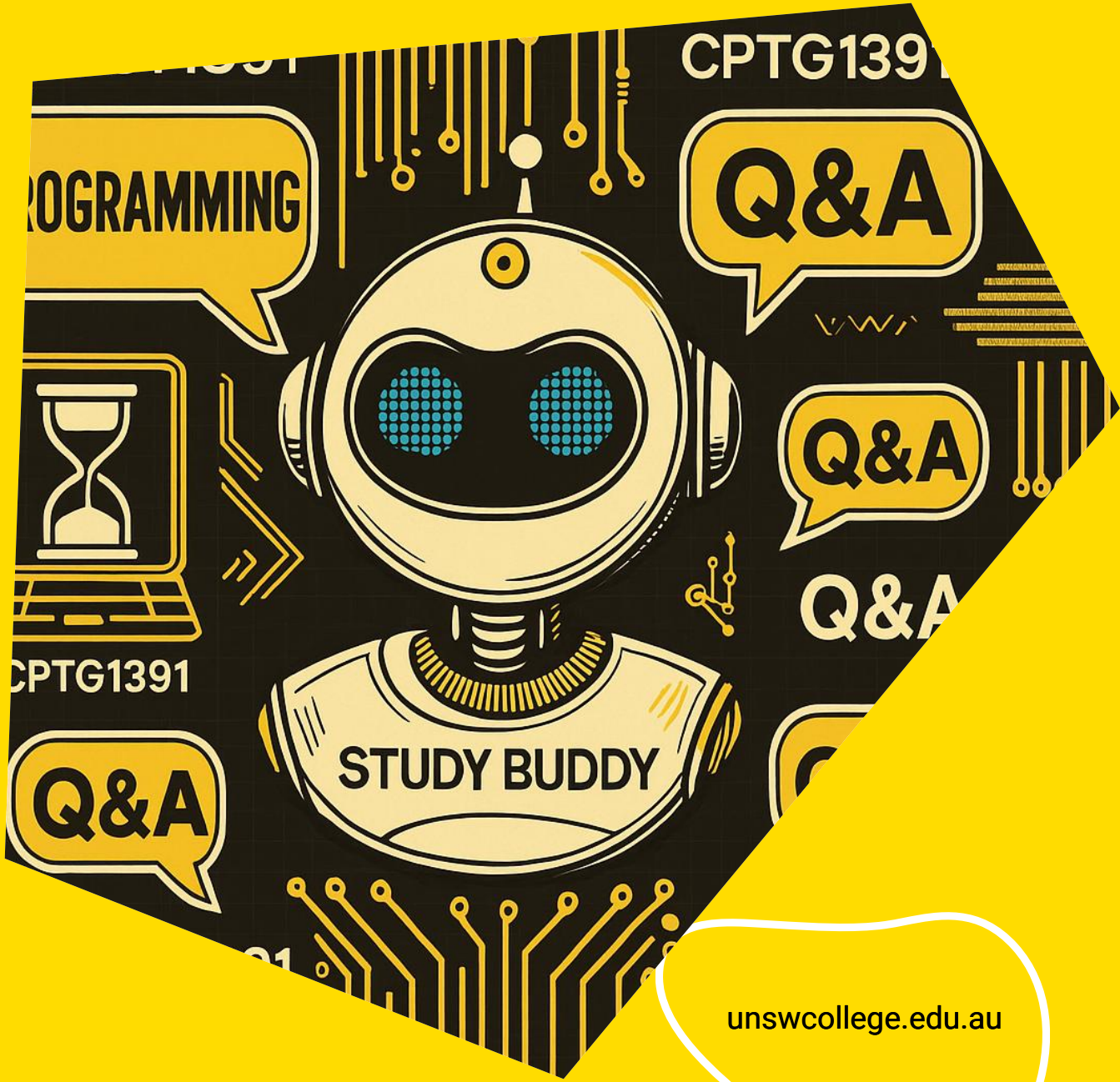
Lecturer and Course Convener:

Dr Pantea Aria

UNSW College

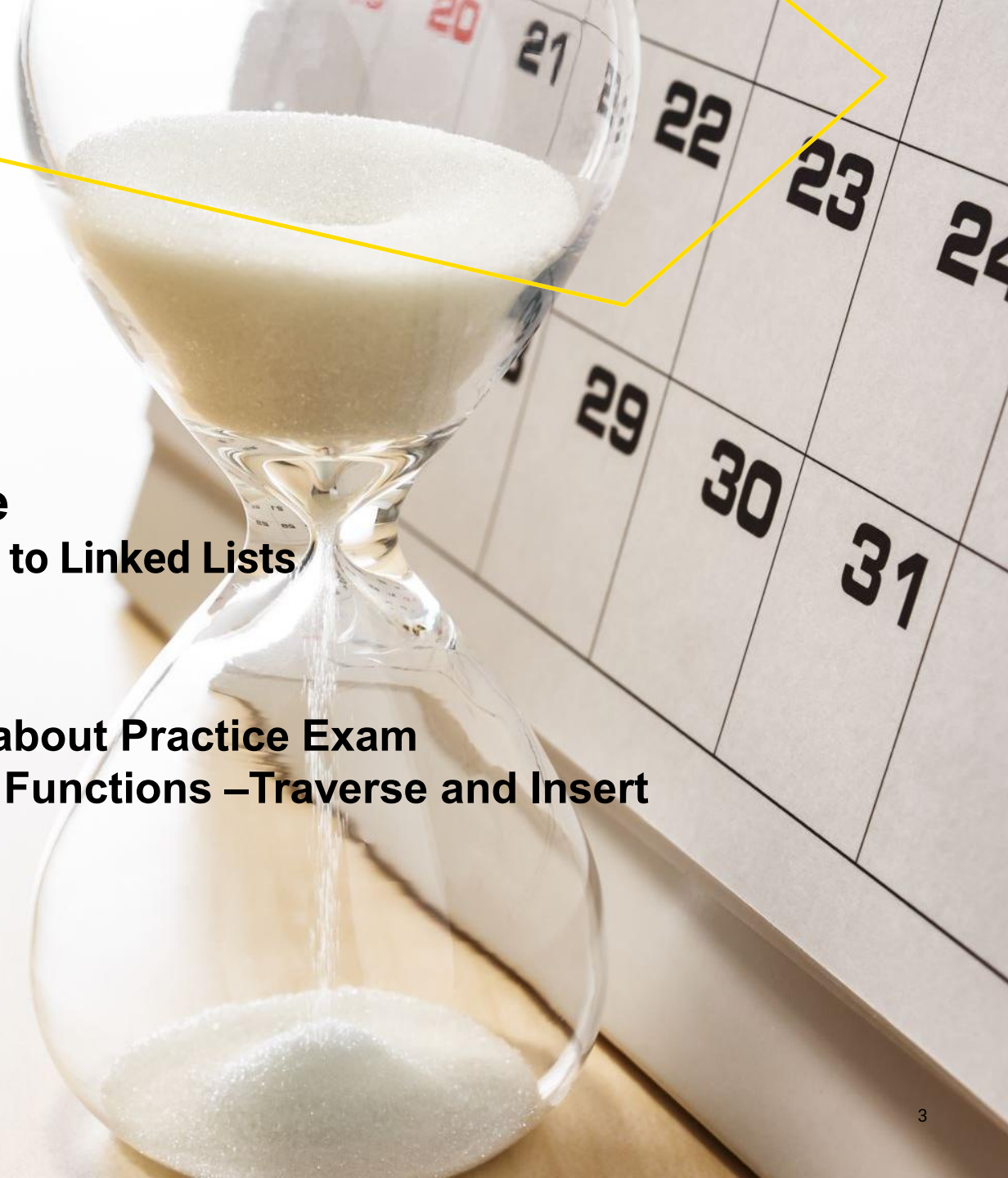
Linked Lists Functions

(part 1)



unswcollege.edu.au

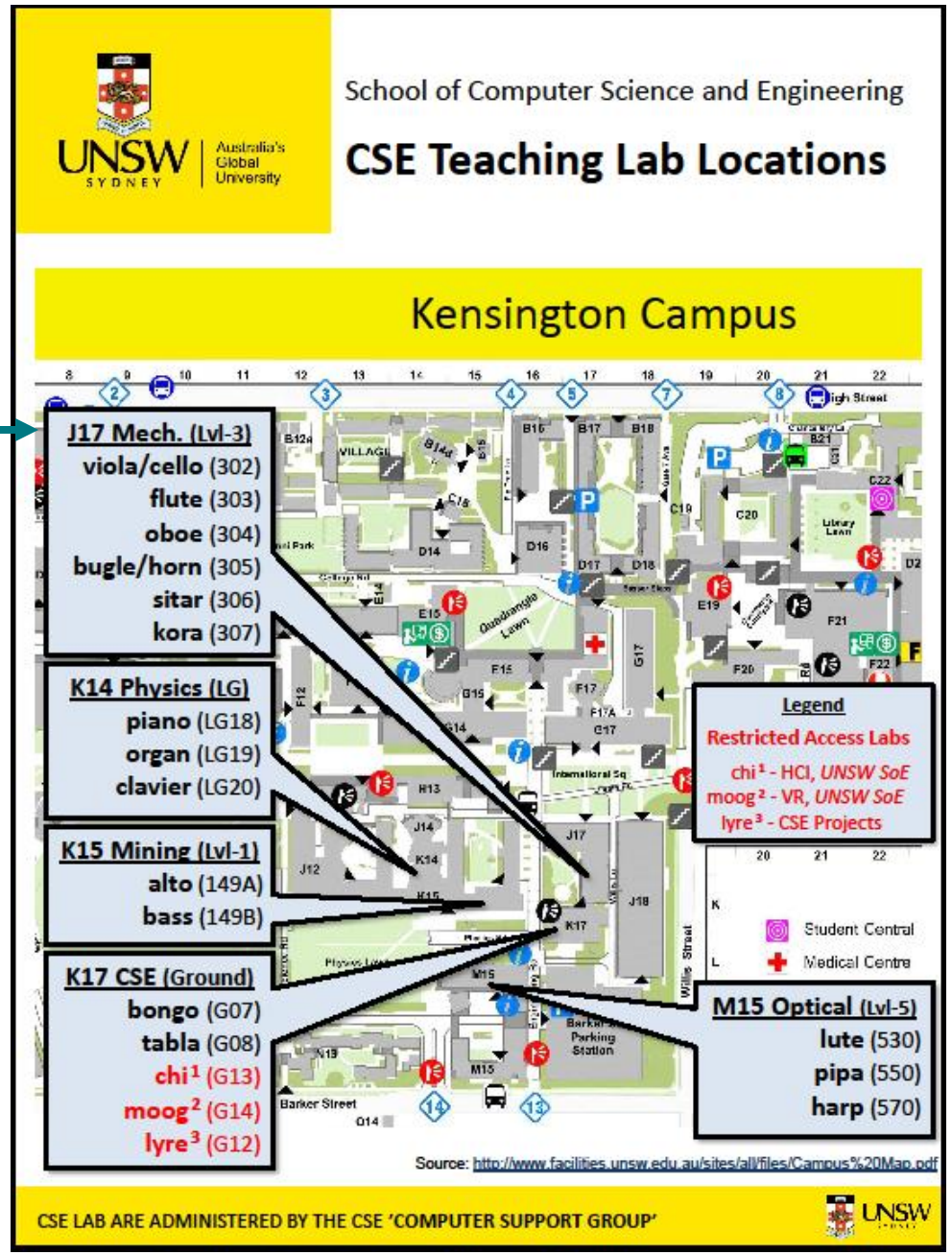
Agenda



- **Last lecture**
  - **Introduction to Linked Lists**
- **Today**
  - **Information about Practice Exam**
  - **Linked Lists Functions –Traverse and Insert**

Practice Exam information

here



There will be no lecture on Monday Week 10

- It will be running during **next Monday lecture**
- Held in **CSE Labs** for **2 hours (Actual exam is about 3 hours)**

  When: Monday 23rd March, 9:00 AM to 11:00 AM (be there 10 min early)
  Where: J17 Ainsworth Building
  Level3 https://www.learningenvironments.unsw.edu.au/physical-spaces/k-j17
- Don’t miss the chance to see what the exam environment is like and get used to using it. And try out some hurdle questions and other exam questions in an exam environment.
- Don’t worry, I won’t mark it. It is only about you getting familiar with the exam environment.

Linked Lists, What are they?

UNSW College

An alternative way to store a collection of data is to use a **linked list.**

Arrays are extremely **useful** and we are definitely not replacing them.

A **linked list** is simply **another option** that may be more suitable in **certain situations**.

Linked lists work well for **sequential data**, such as:

- a to-do list
- a train made up of connected carriages
- a queue of customers waiting in line
- a chain of connected tasks in a workflow

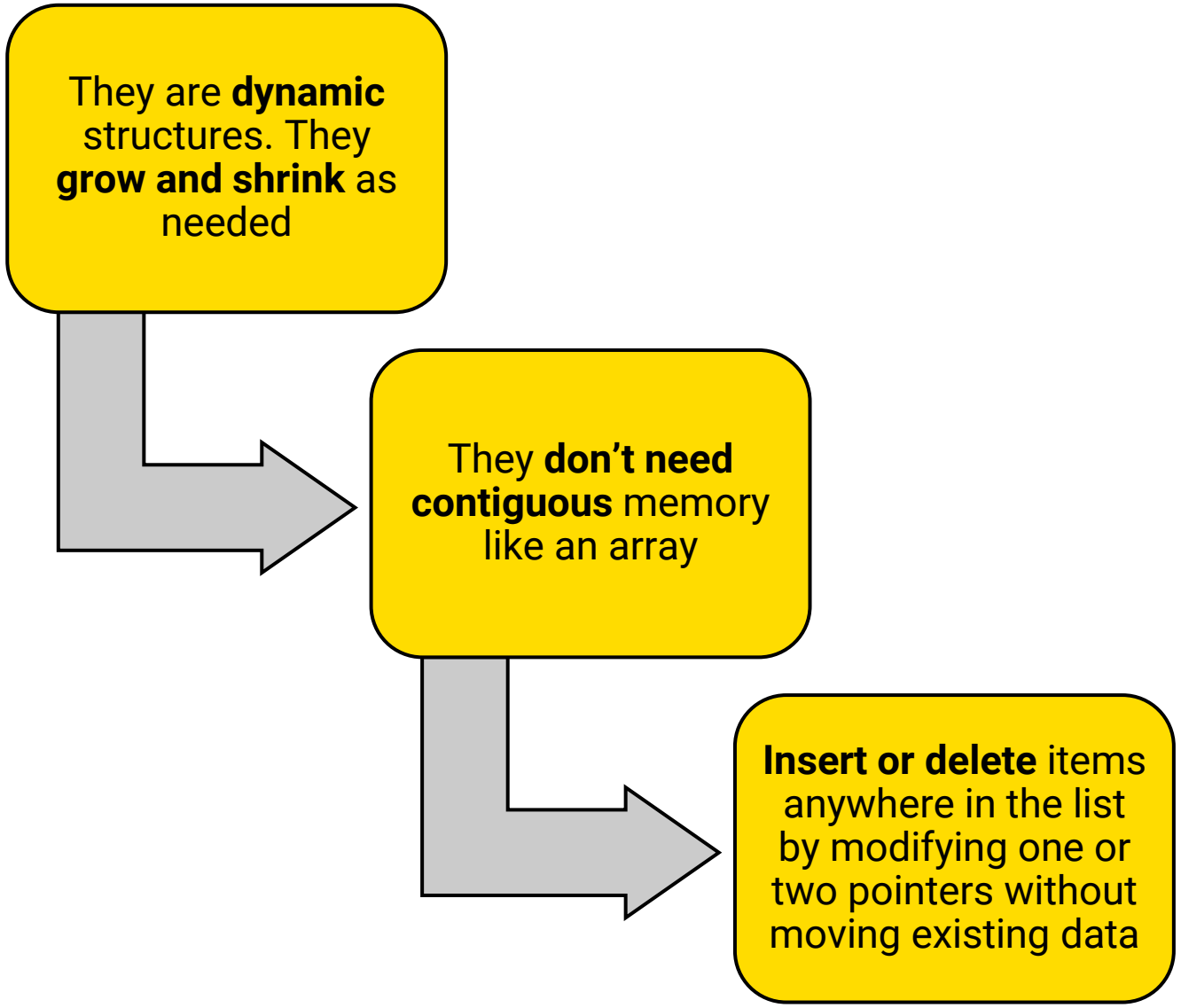
Arrays Advantages

- Data elements are stored next to each other in a continuous (contiguous) section of memory.
- They are efficient for both sequential access and direct (random) access.
- Adding or removing elements at the end of the structure is simple and efficient.

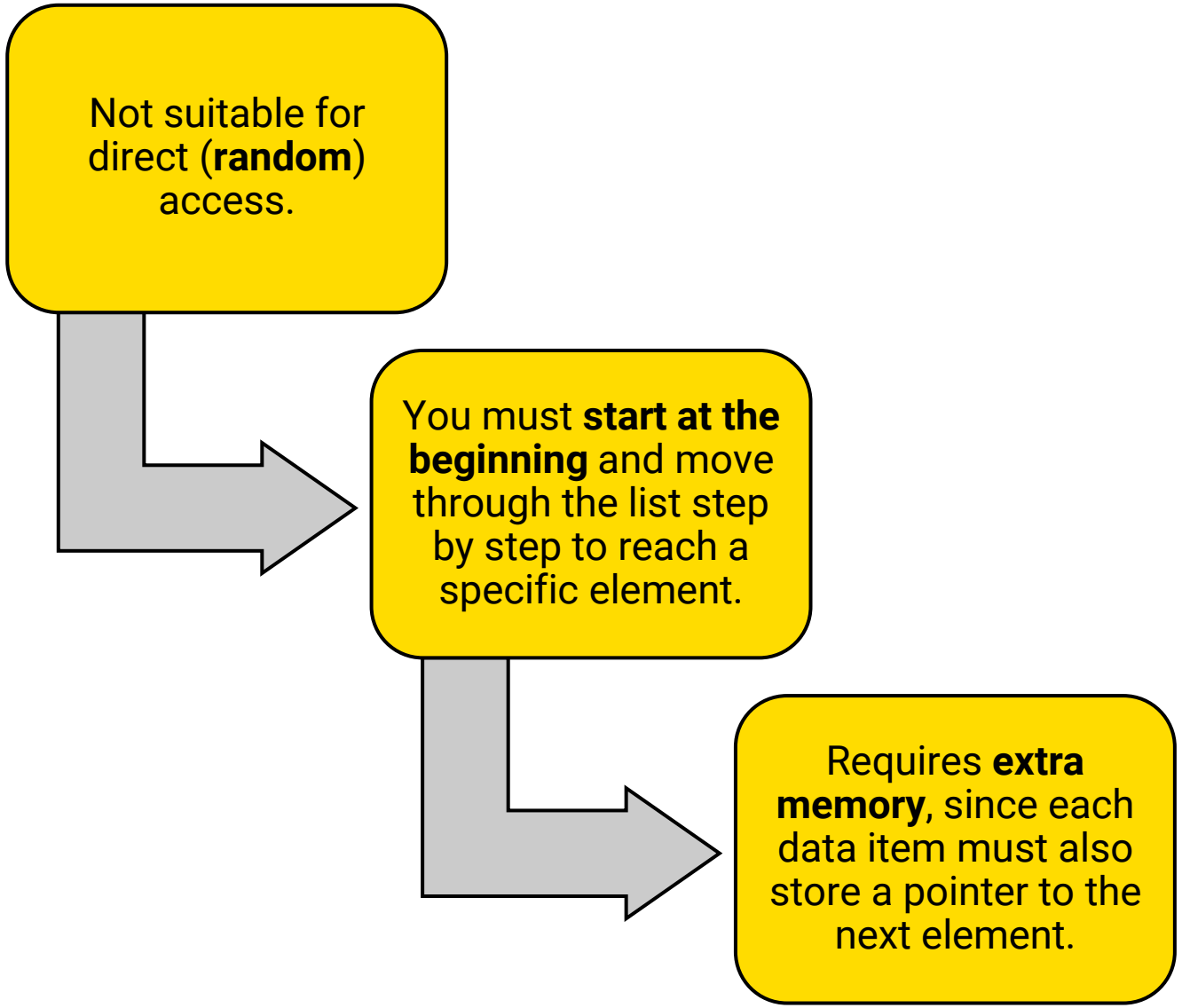
Arrays Disadvantages

- Inserting or deleting elements in the middle can be inefficient and cumbersome.
- All the elements that come after the insertion or deletion point must be shifted to make space or close the gap.
- If the array is static (fixed size), we actually cannot insert a new element once it is full.

Linked lists Advantages



Linked lists Disadvantages



Arrays and Linked Lists in the Memory

The elements in a linked list are not stored in contiguous memory locations.

Instead, they are distributed (scattered) across different areas of memory and connected using pointers

```
int numbers[] = {10, -4, 0, 12, 56};
```

0x30	
0x34	
0x38	10
0x3c	-4
0x40	0
0x44	12
0x48	56
0x4c	
0x50	
0x54	

list

0x30	
0x34	
0x38	10
0x3c	0x6c
0x40	
0x44	12
0x48	0x78
0x4c	
0x50	
0x54	

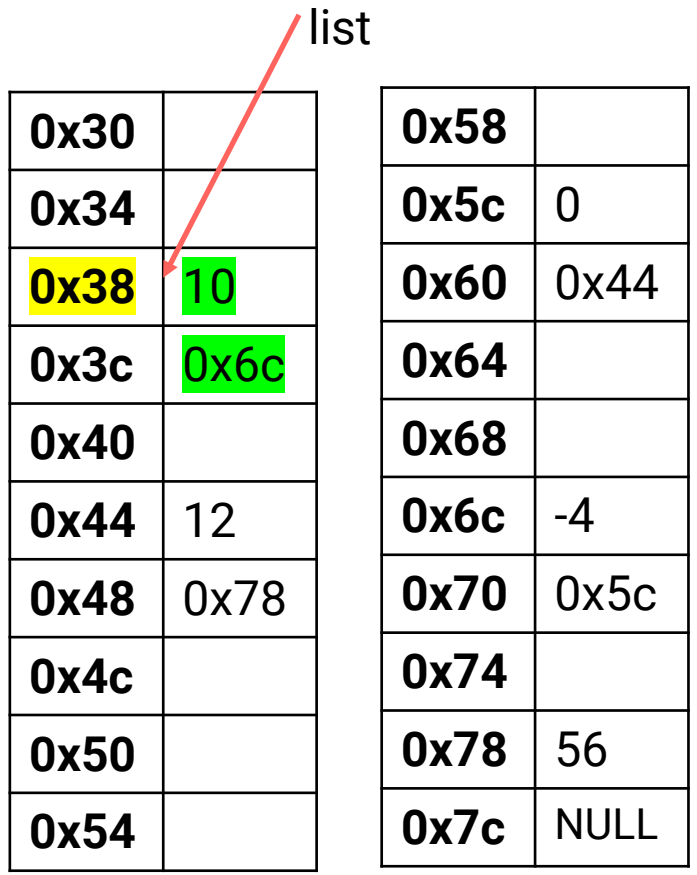
0x58	
0x5c	0
0x60	0x44
0x64	
0x68	
0x6c	-4
0x70	0x5c
0x74	
0x78	56
0x7c	NULL

Linked Lists in the Memory

You need a pointer to the **first node** in the list (called the **head**).

Each node stores its data along with a **pointer** to the **next node** in the sequence.

To access the full list, you follow each pointer **step by step**, like moving from one train carriage to the next in a connected train.

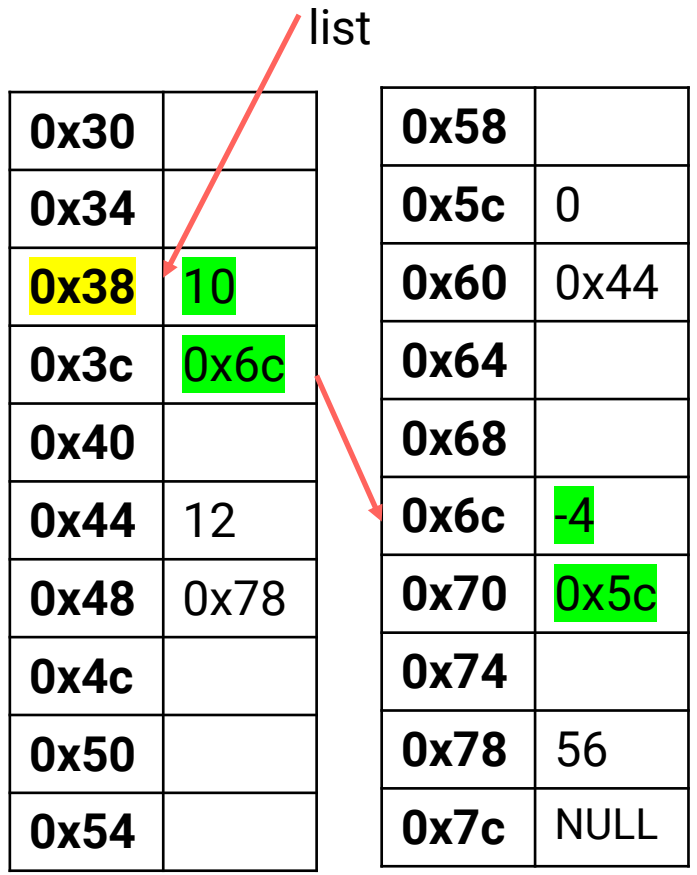


Linked Lists in the Memory

You need a pointer to the **first node** in the list (called the **head**).

Each node stores its data along with a **pointer** to the **next node** in the sequence.

To access the full list, you follow each pointer **step by step**, like moving from one train carriage to the next in a connected train.

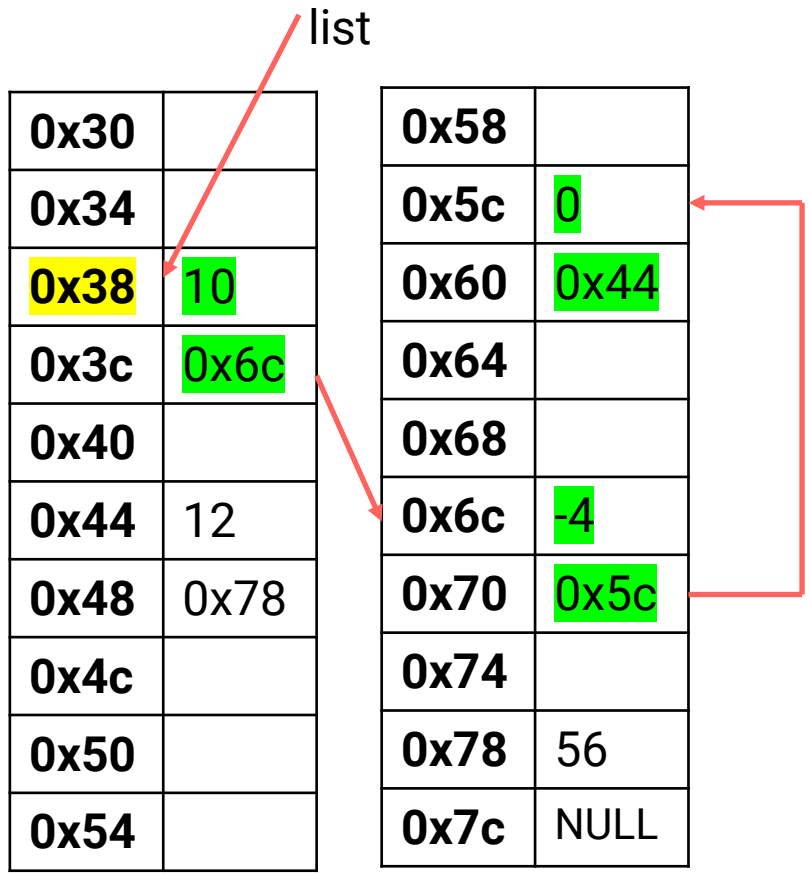


Linked Lists in the Memory

You need a pointer to the **first node** in the list (called the **head**).

Each node stores its data along with a **pointer** to the **next node** in the sequence.

To access the full list, you follow each pointer **step by step**, like moving from one train carriage to the next in a connected train.

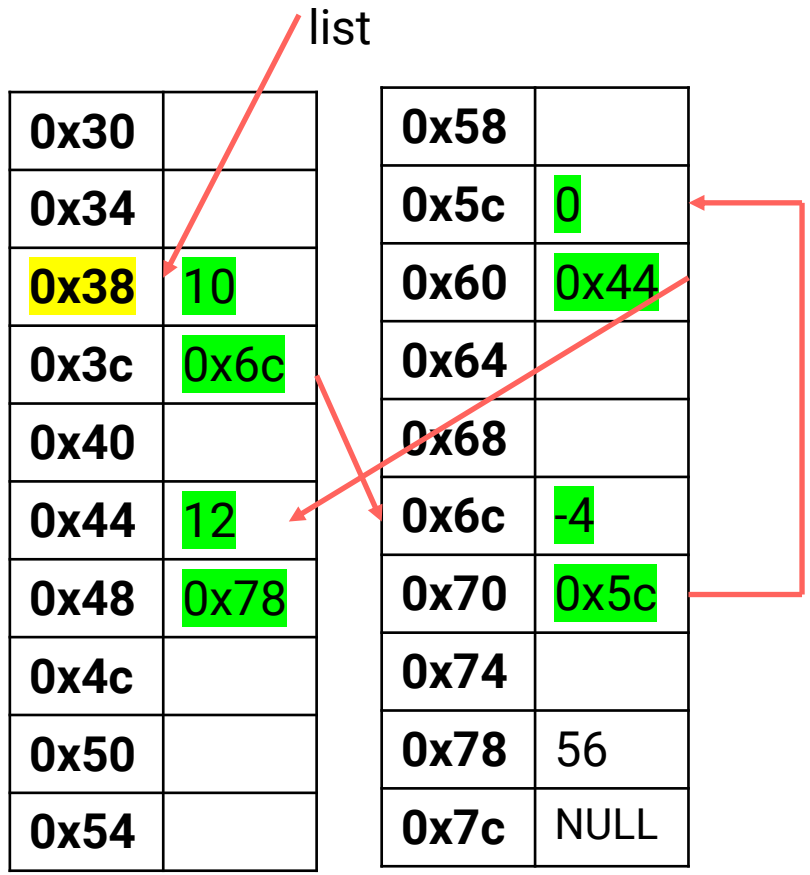


Linked Lists in the Memory

You need a pointer to the **first node** in the list (called the **head**).

Each node stores its data along with a **pointer** to the **next node** in the sequence.

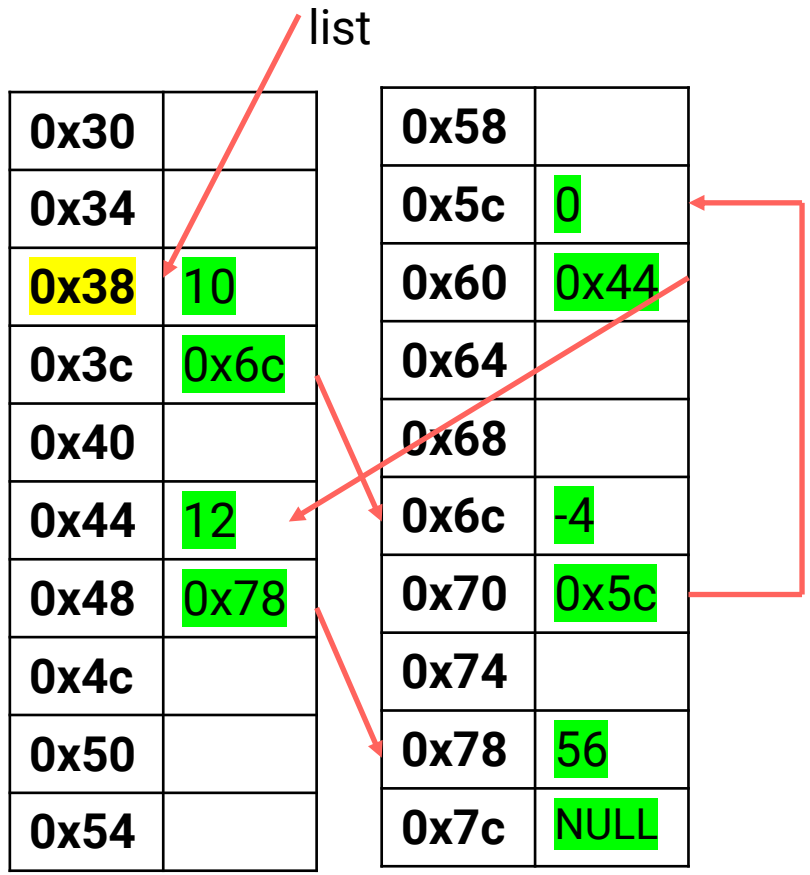
To access the full list, you follow each pointer **step by step**, like moving from one train carriage to the next in a connected train.



Linked Lists in the Memory

When the pointer to the next node contains **NULL**, it indicates that there are no more nodes to follow and you have reached the end of the list.

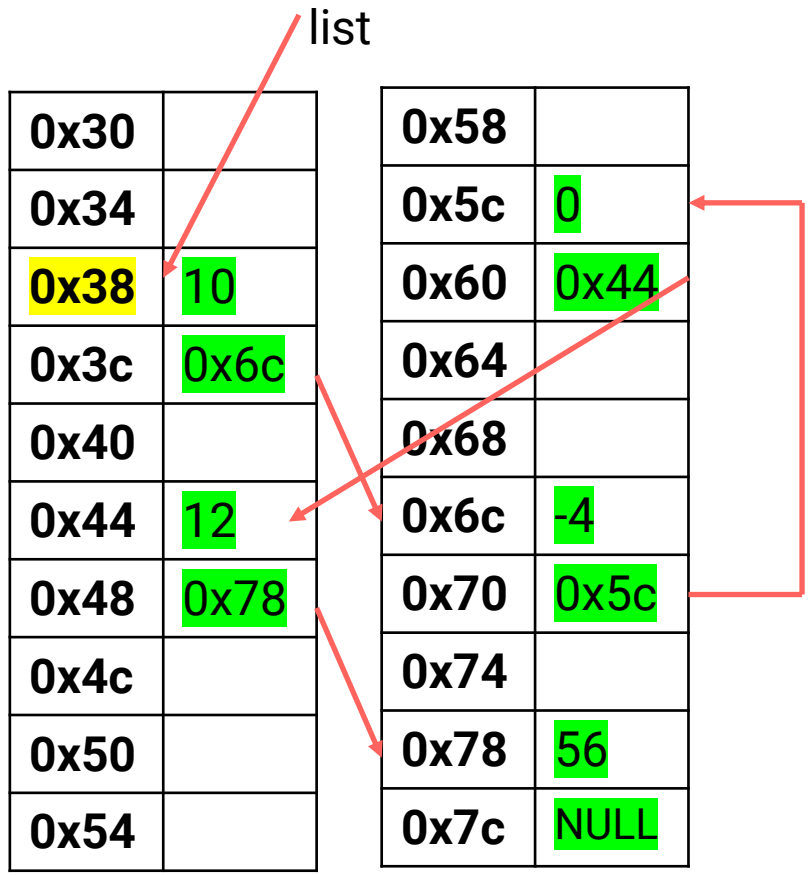
At this point, you have successfully traversed the entire linked list.



Linked Lists in the Memory

A linked list is described as sequential because we must begin at the first node and move through each node one by one to access elements.

Unlike arrays, we cannot directly access a specific element using an index; instead, we must traverse the list to reach it.

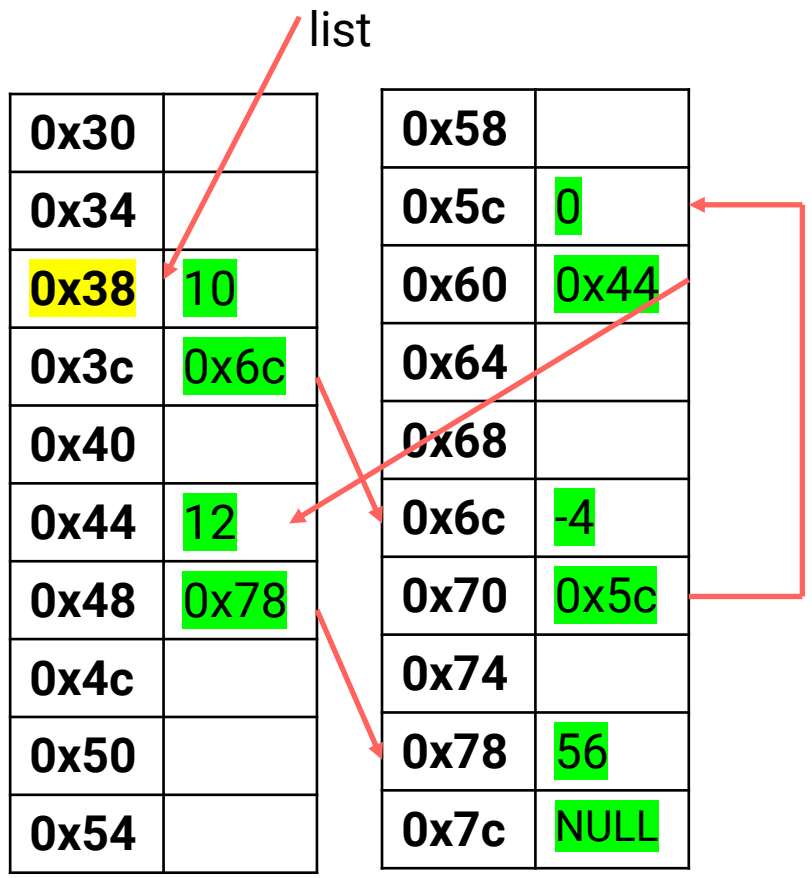


Nodes

In C, we can define a structure (**struct**) that allows us to **store both**:

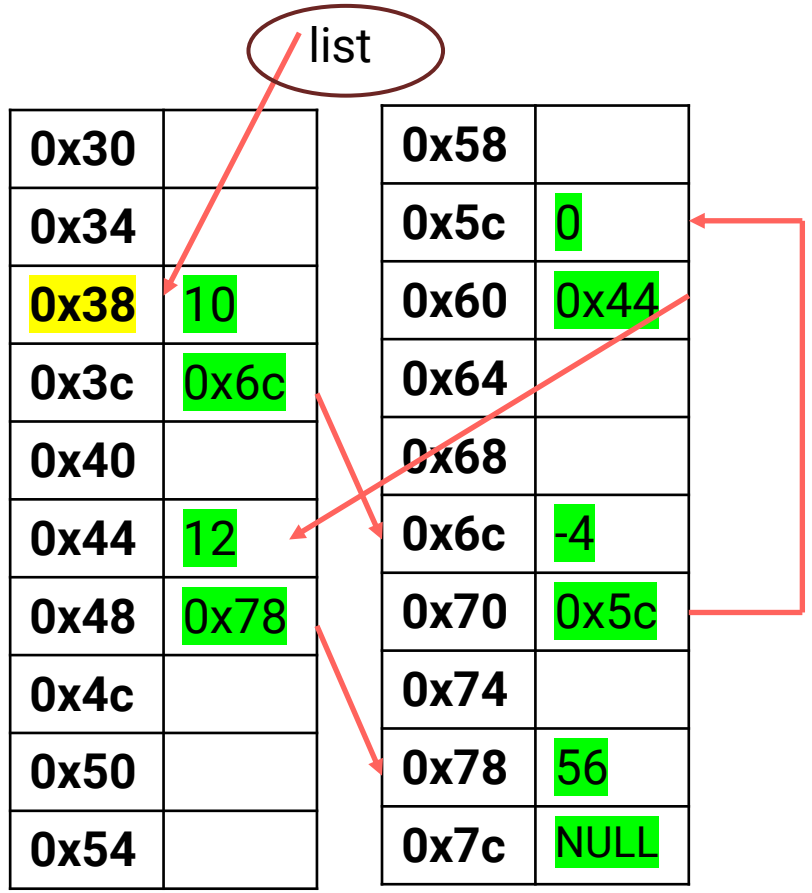
- ⟶ an **int value** (the data), and
- ⟶ a **pointer containing the address of the next node** in the list.

```
struct node {

    int data;

    struct node *next;
};
```



The variable list stores the address of the first node in the linked list

```
struct node *list;

struct node {

    int data;

    struct node *next;
};
```



```
struct node *list;

 struct node {

       int data;

       struct node *next;
      };
```

Each node contains a **data** field.

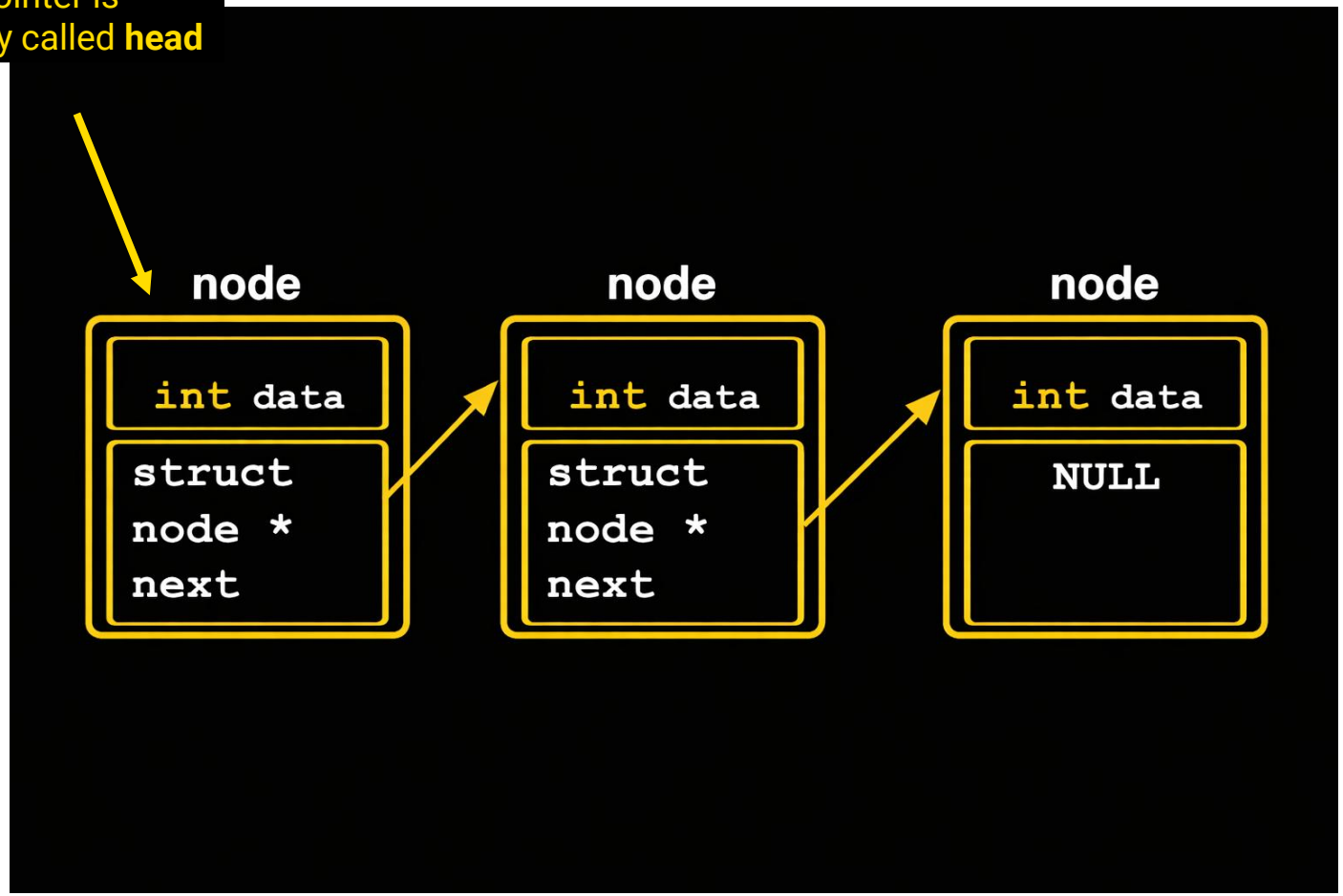
In this example, the **data** is a single **int**, but it can be **any data type** depending on your needs.

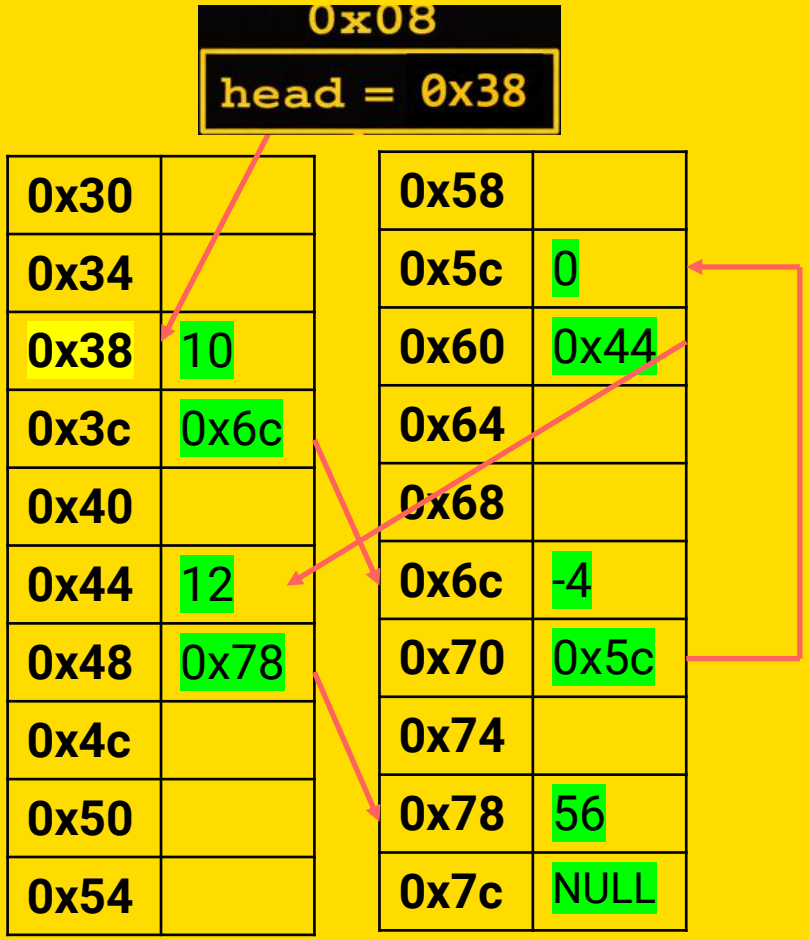
Later, we will explore linked lists that store **different types** of data.

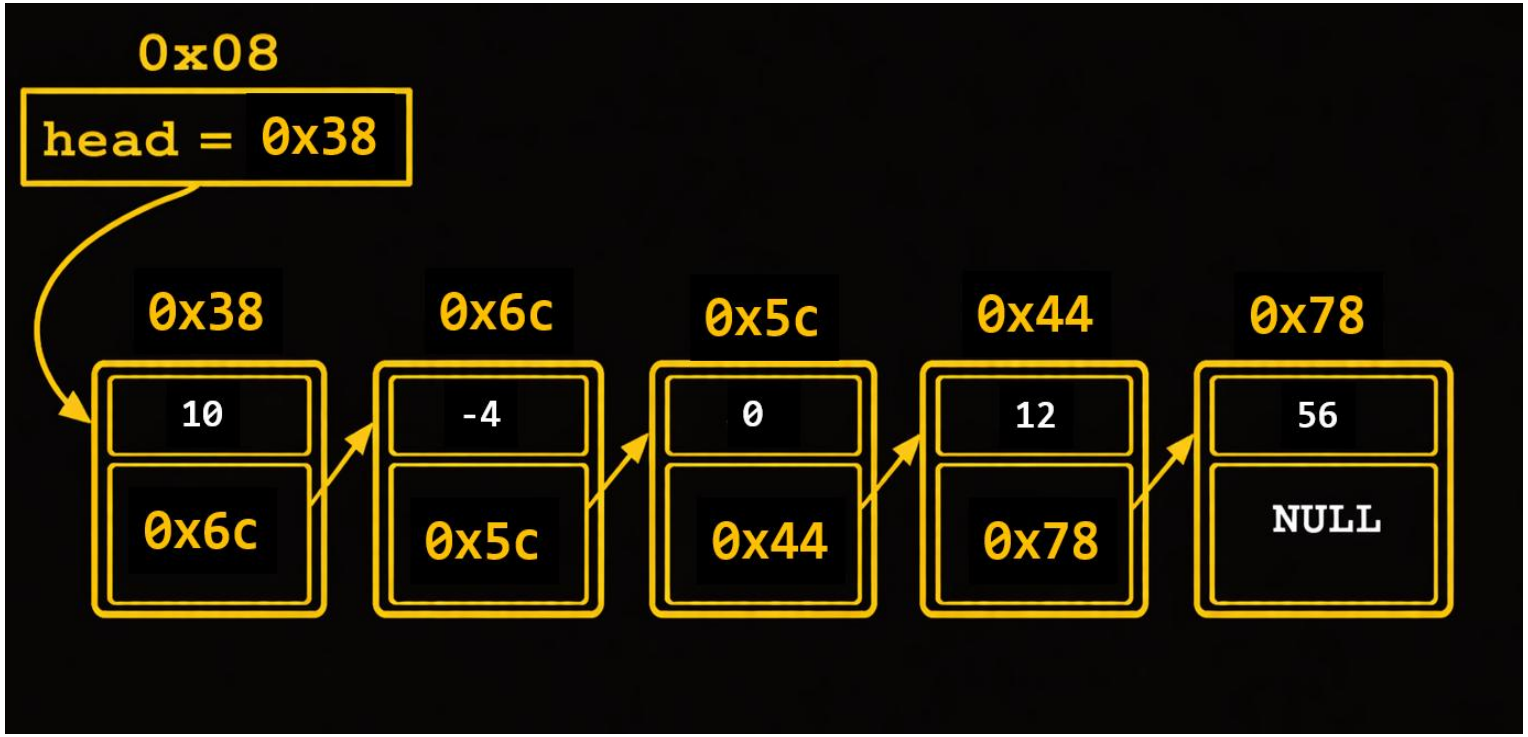
Each node also includes a **pointer to the next node** in the list (which is of the **same type**).

Visualising a Linked List

A pointer to the first node in the linked list(this pointer is commonly called **head**




0x08
head = 0x38
0x30
0x34
0x38
10
0x3c
0x6c
0x40
0x44
12
0x48
0x78
0x4c
0x50
0x54
0x58
0x5c
0
0x60
0x44
0x64
0x68
0x6c
-4
0x70
0x5c
0x74
0x78
56
0x7c
NULL


0x08
head = 0x38
0x38
10
0x6c
0x6c
-4
0x5c
0x5c
0
0x44
0x44
12
0x78
0x78
56
NULL

Creating a Linked List

Here is the code to create a linked list with nothing in it.

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

0x08

Head = NULL

Creating a Node

We use **malloc** to **allocate new nodes** on the **heap**.

This gives us full control over **memory management**, allowing us to:

- **create new nodes** whenever needed, and
- **free them** when they are **no longer required.**

Steps to Create a Node:

- **Allocate memory** for a struct node using **malloc**.
- **Assign a value** to the node's **data** field.
- Set the **next pointer** to point to the **appropriate** node (or **NULL**).

Creating a List with 1 Node

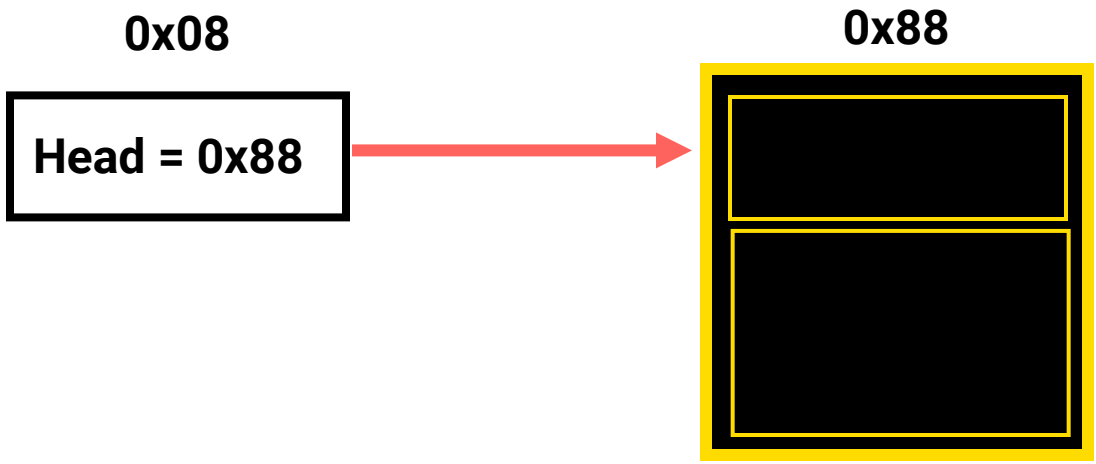
```c
struct node {
        int data;
        struct node *next;
};

struct node *head = NULL;
```

0x08

Head = NULL

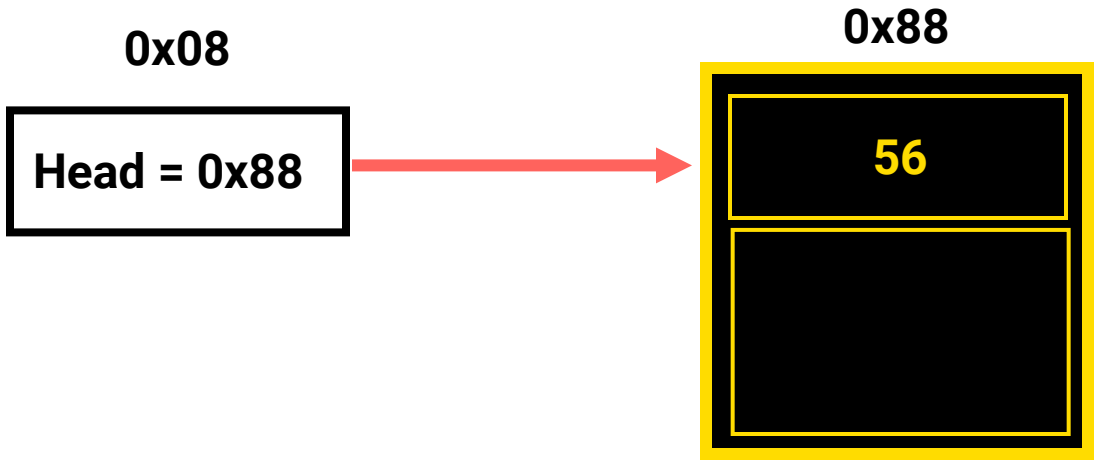
Creating a List with 1 Node

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

struct node *head = NULL;
head = malloc(sizeof(struct node));
```



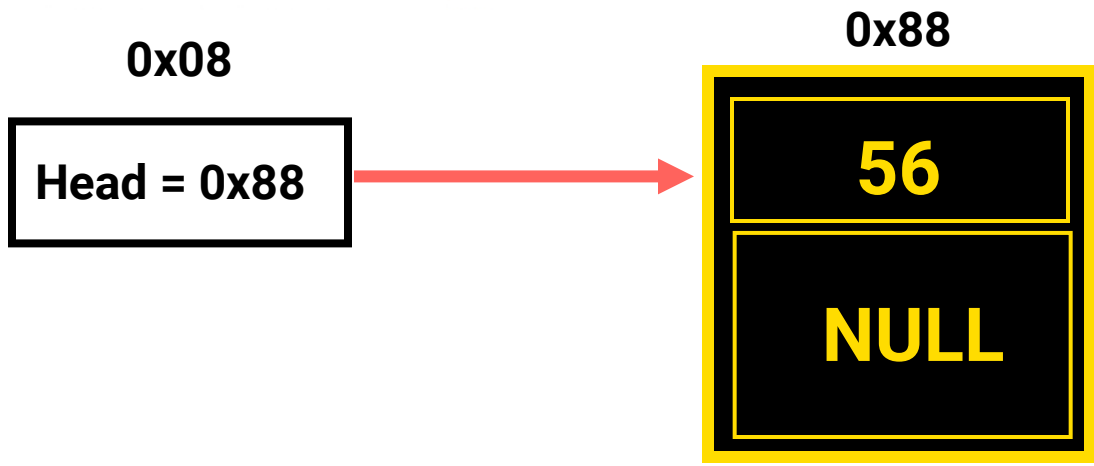
Creating a List with 1 Node

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

struct node *head = NULL;
head = malloc(sizeof(struct node));
head->data = 56;
```



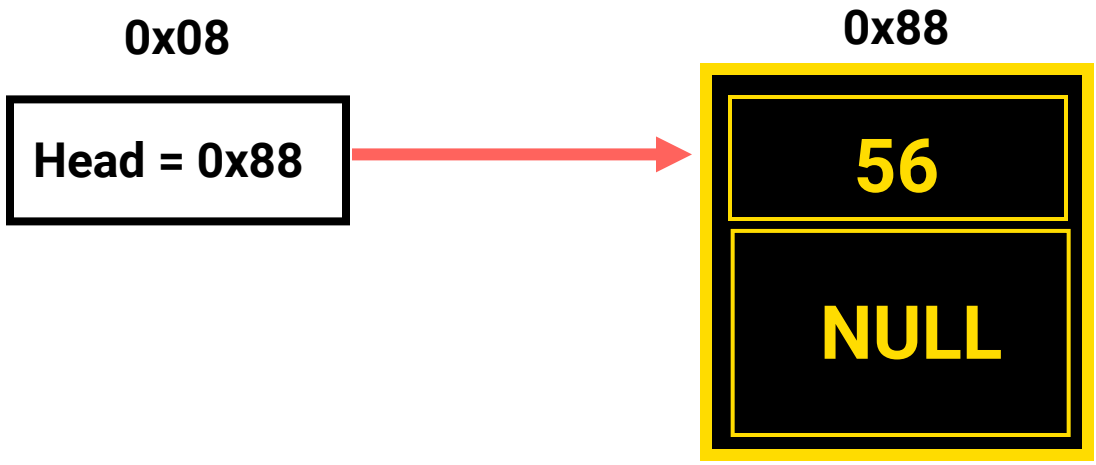
Creating a List with 1 Node

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

struct node *head = NULL;
head = malloc(sizeof(struct node));
head->data = 56;
head->next = NULL;
```



This is a linked list of size 1.

Adding a new node to the end of our list

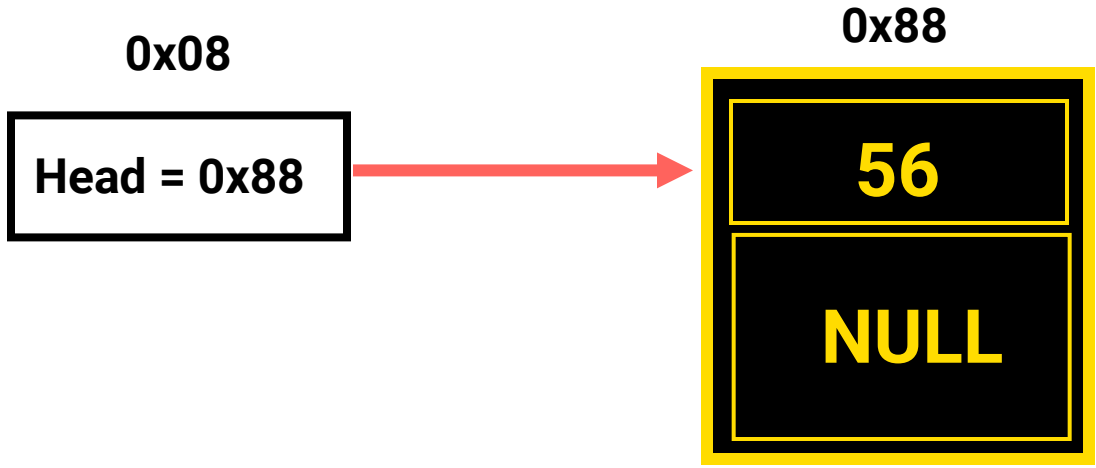


We will create a new node and attach it to the **end of the existing list**.

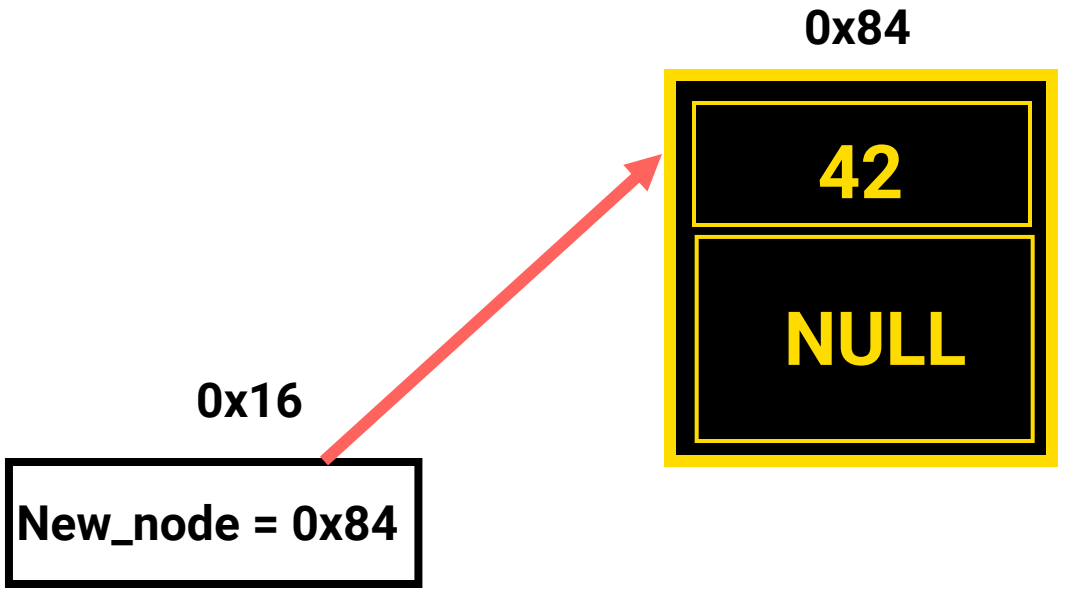
The **last node** in a linked list is commonly referred to as the **tail**.

Adding a new node to the end of our list

is the current node connected to the new node? (not yet)

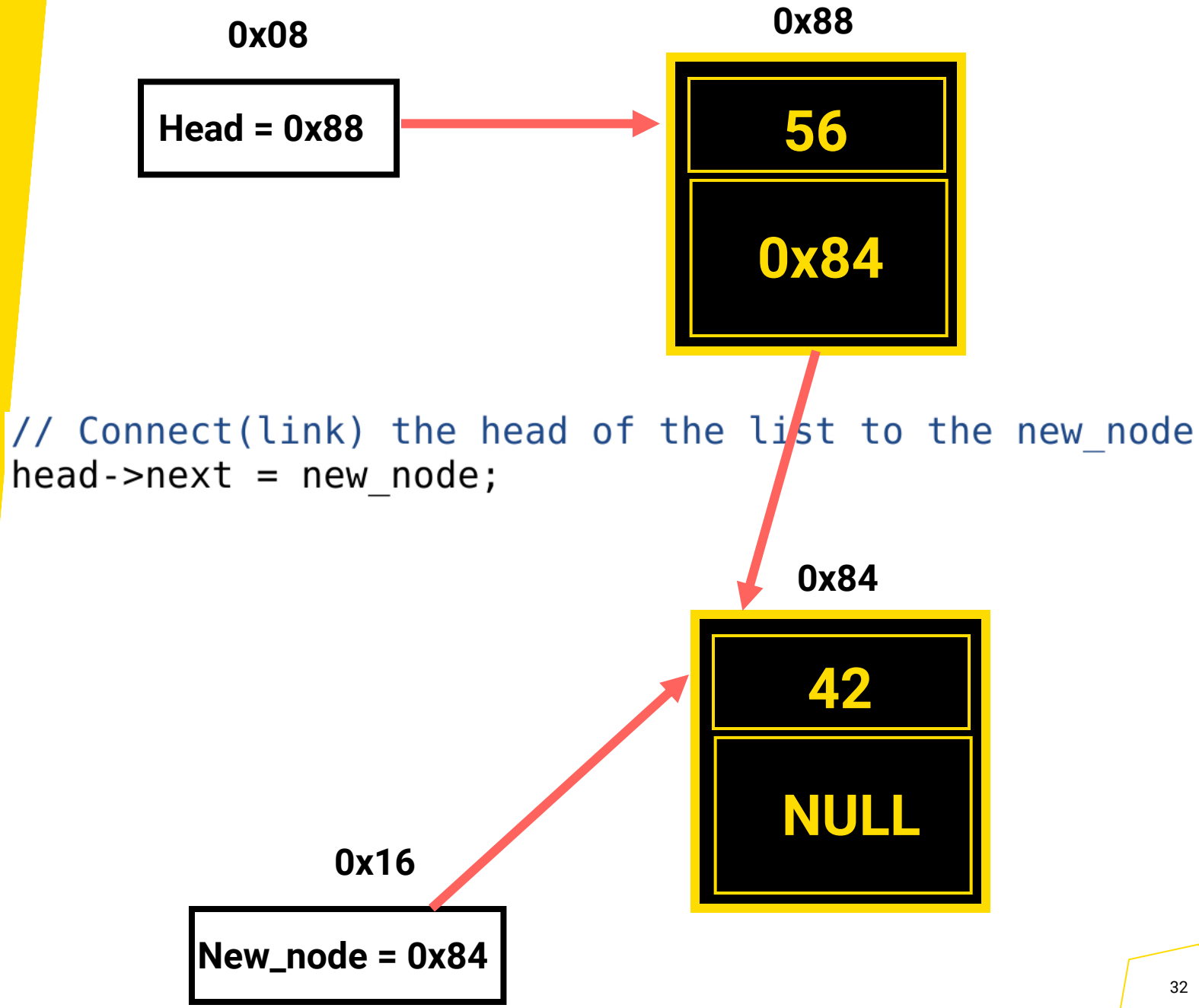



```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 42;
new_node->next = NULL;
```

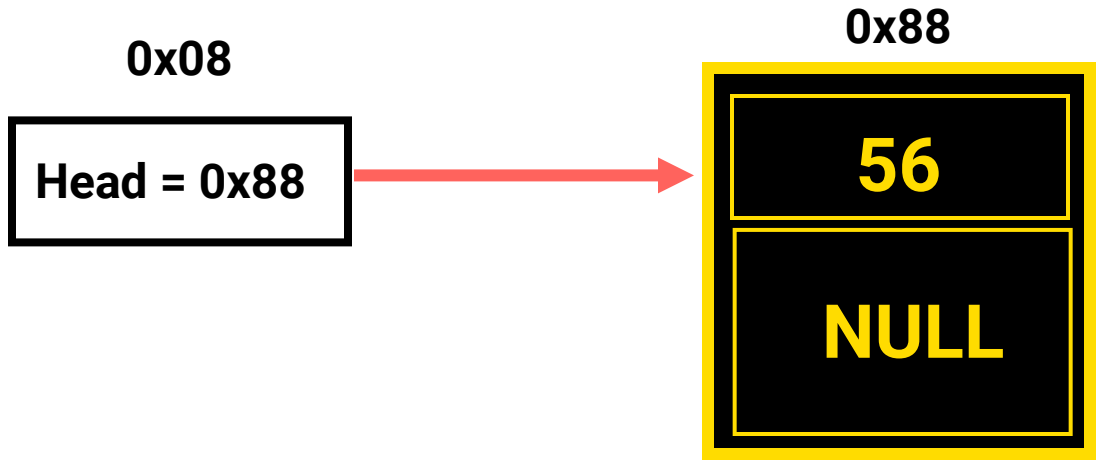


Adding a new node to the end of our list

are they connected now? (YES!!)



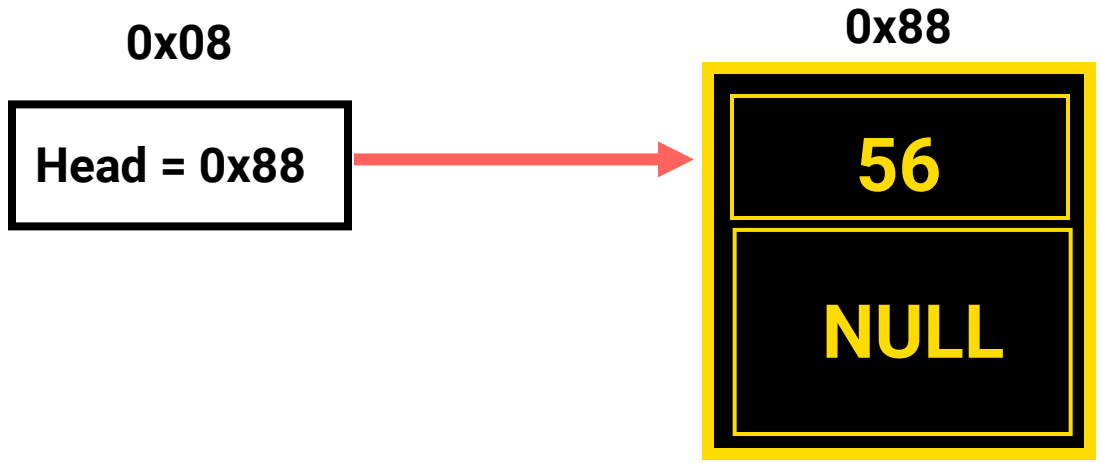
Adding a new node to the start of our list



We will create a new node and **insert it at the beginning** of the list.

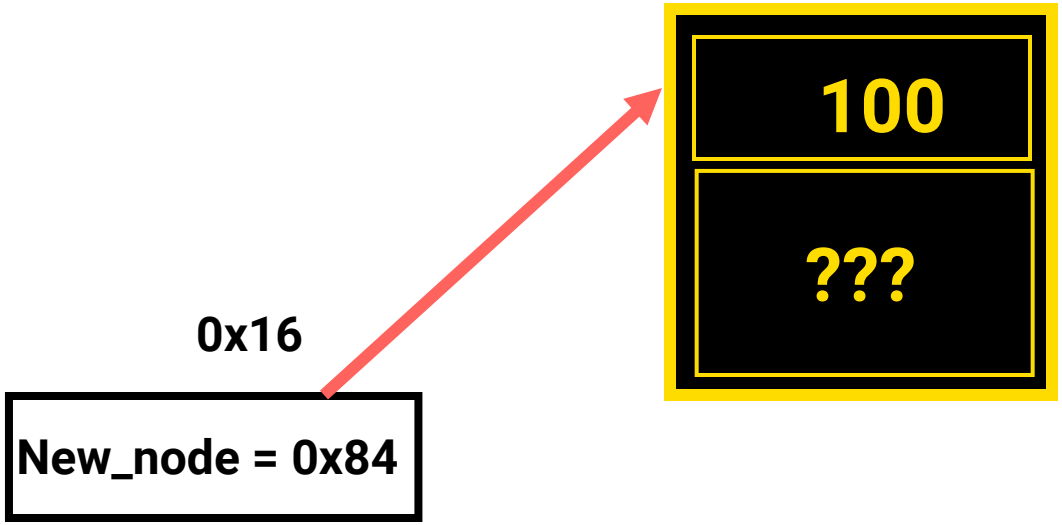
The **first node** in a linked list is commonly referred to as the **head**.

Adding a new node to the start of our list




```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 100;
new_node->next = ???;
```

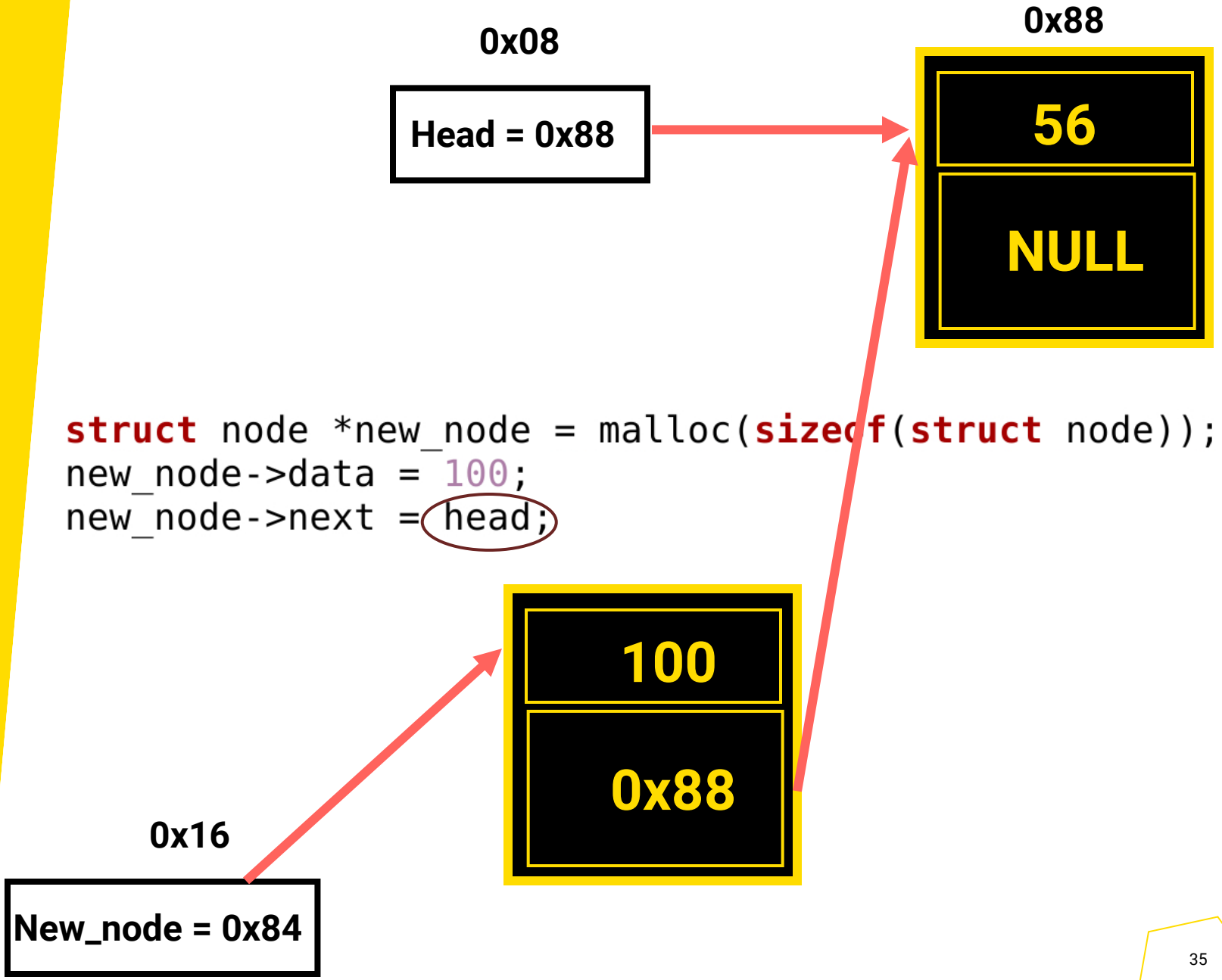
is the new node connected to the head? (not yet)



Adding a new node to the start of our list

is the new node connected to the head? YES!!!

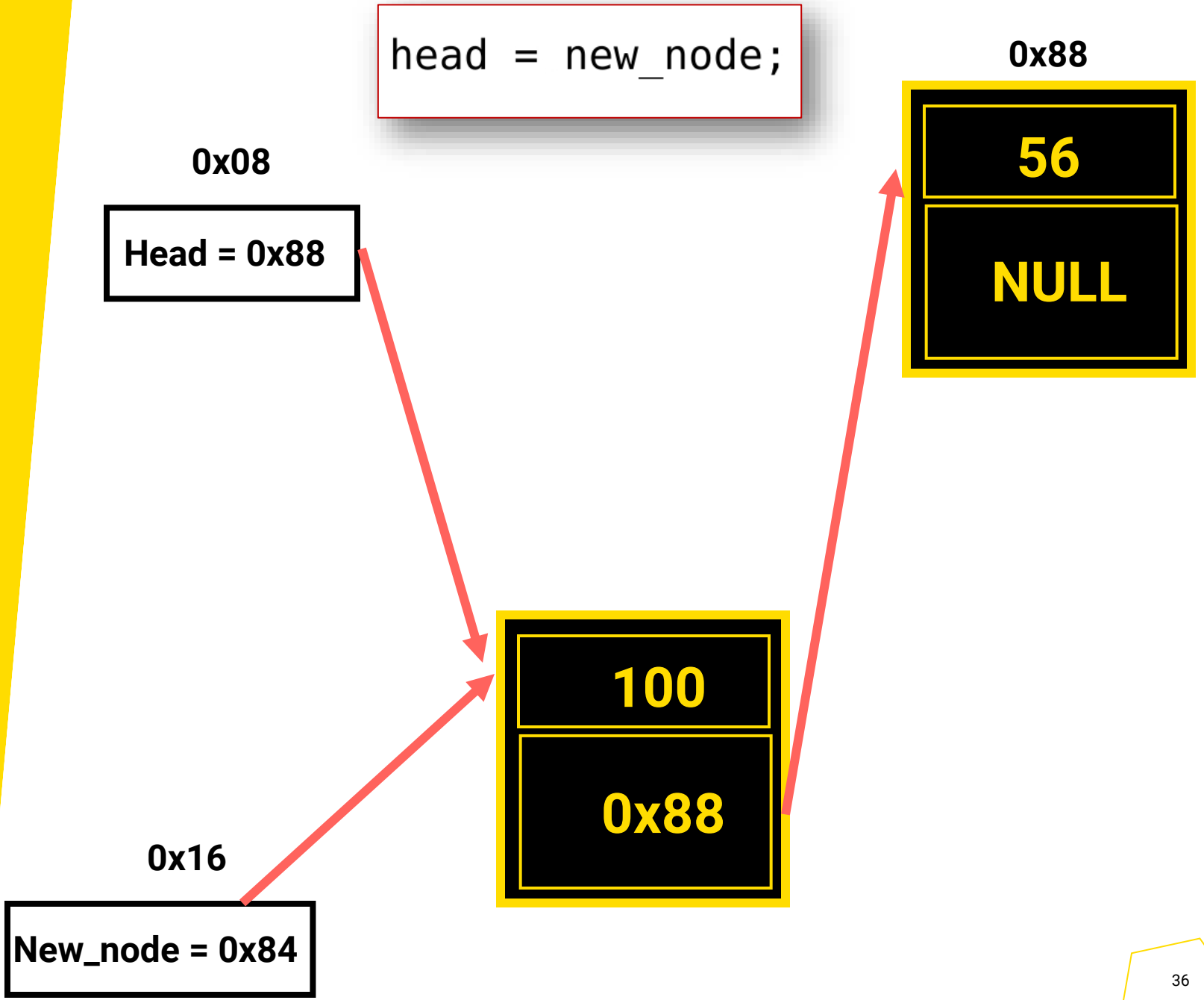
Is the head pointing to the correct node? (not yet)



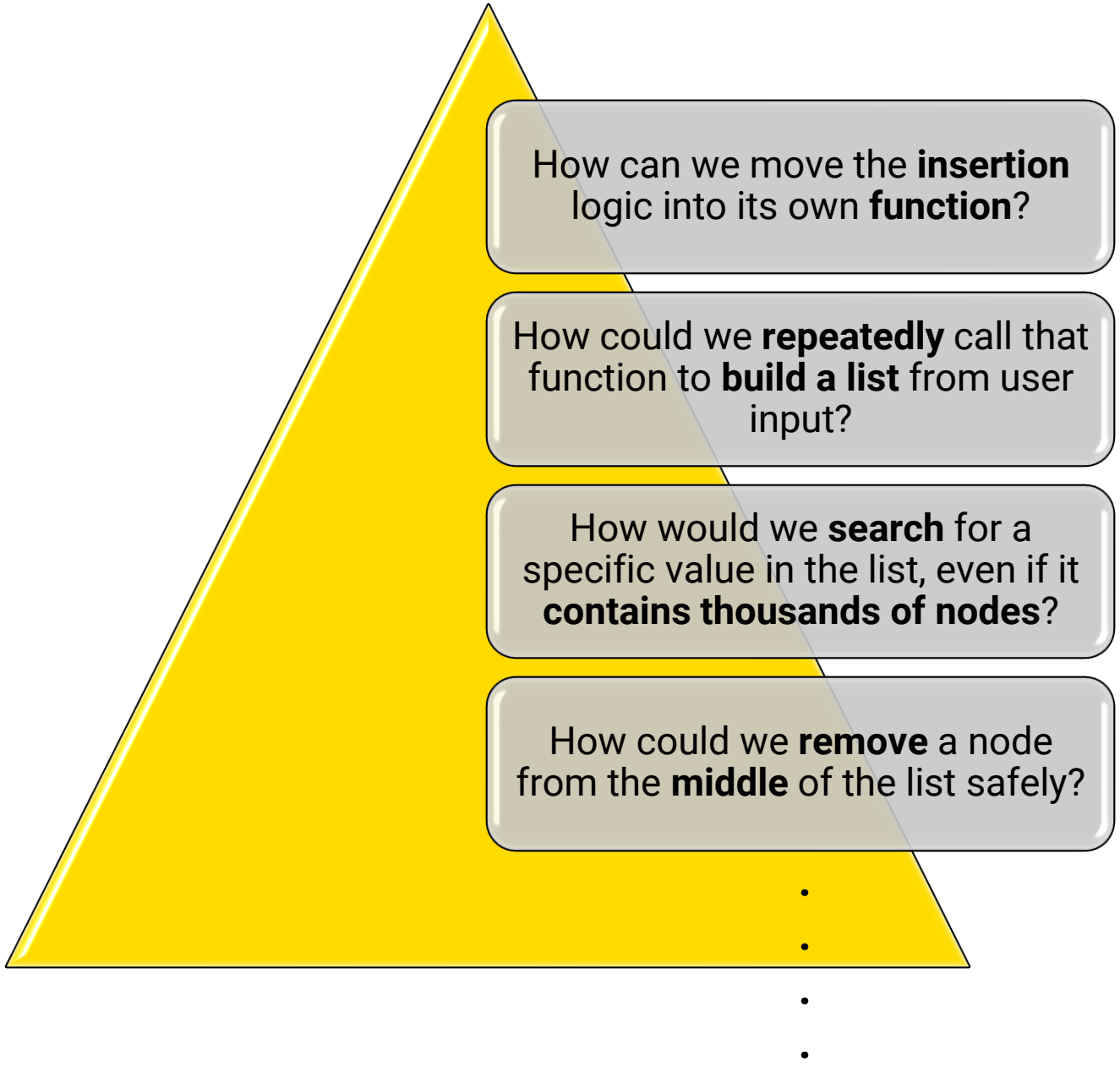
Adding a new node to the start of our list

is the new node connected to the head? YES!!!

Is the head pointing to the correct node? (YES!!!)



Linked Lists Functions



We can improve our program by writing functions

e.g. create_node

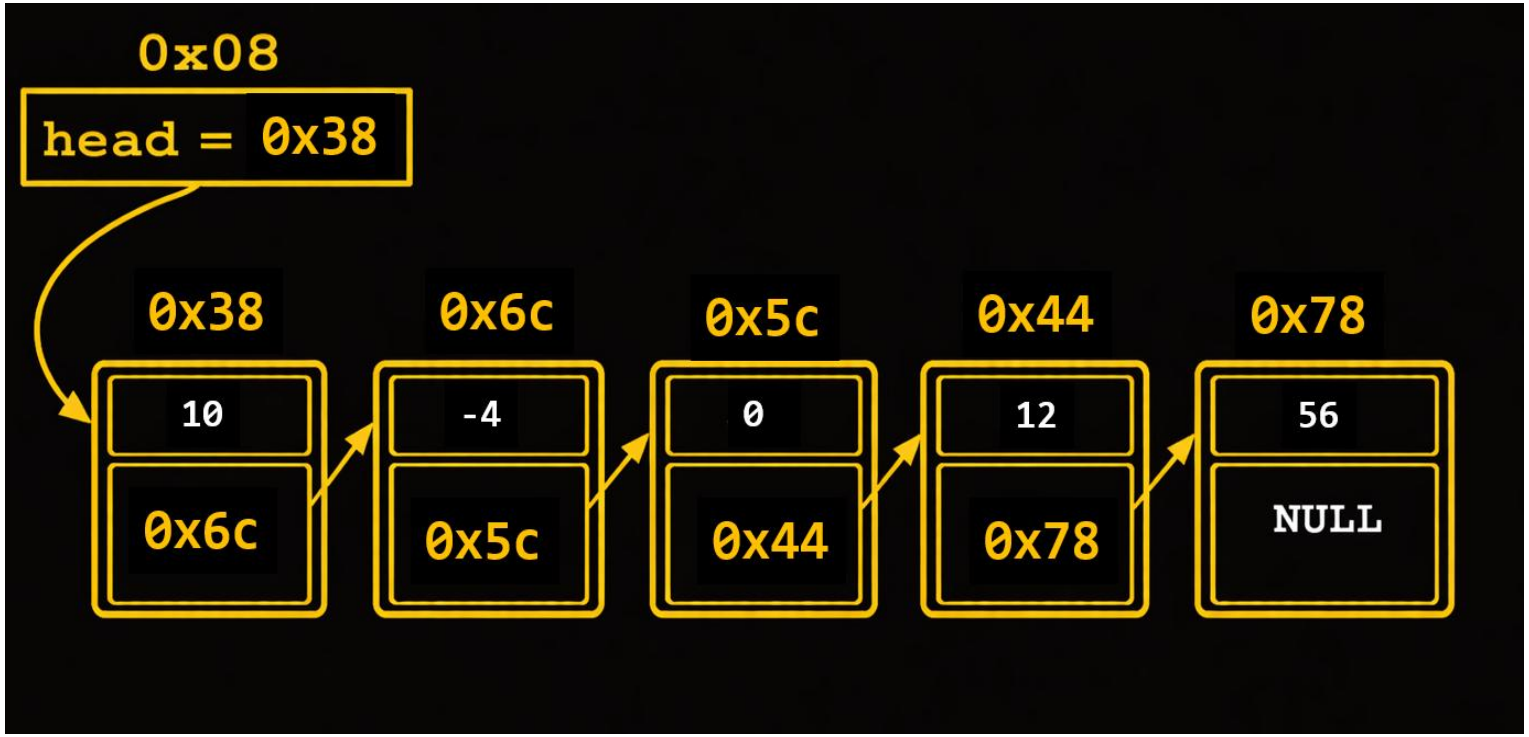
```c
// Allocates memory for a new node and initialises it
// with the given data value and next pointer.
// Returns NULL if memory allocation fails.

struct node *create_node(int data, struct node *next) {

    // Allocate memory for one struct node
    struct node *new_node = malloc(sizeof(struct node));

    // Check if memory allocation was successful
    if (new_node == NULL) {
        return NULL;
    }

    // Assign the data value to the node
    new_node->data = data;

    // Set the next pointer
    new_node->next = next;

    // Return the address of the newly created node
    return new_node;
}
```

Create a Linked List Inserting at Head

```c
int main(void) {

    struct node *head = NULL;
    int i = 0;

    while (i < 10) {
        struct node *new_node = create_node(i, head);
        head = new_node;
        i++;
    }

    return 0;
}
```

Printing a Node

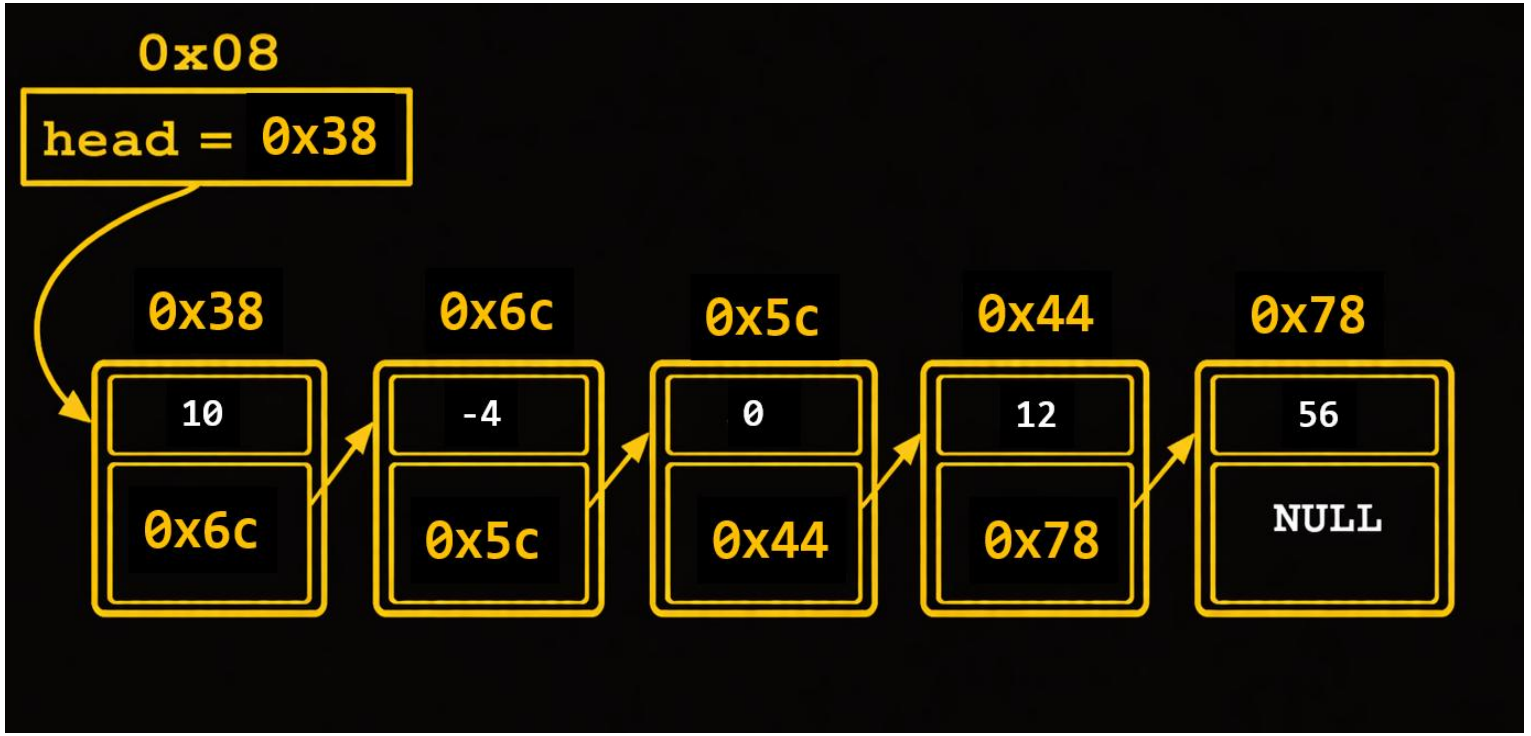
How could we print the data from the **first** node in this linked list?



Printing a Node

```
printf("%d", head->data);
```

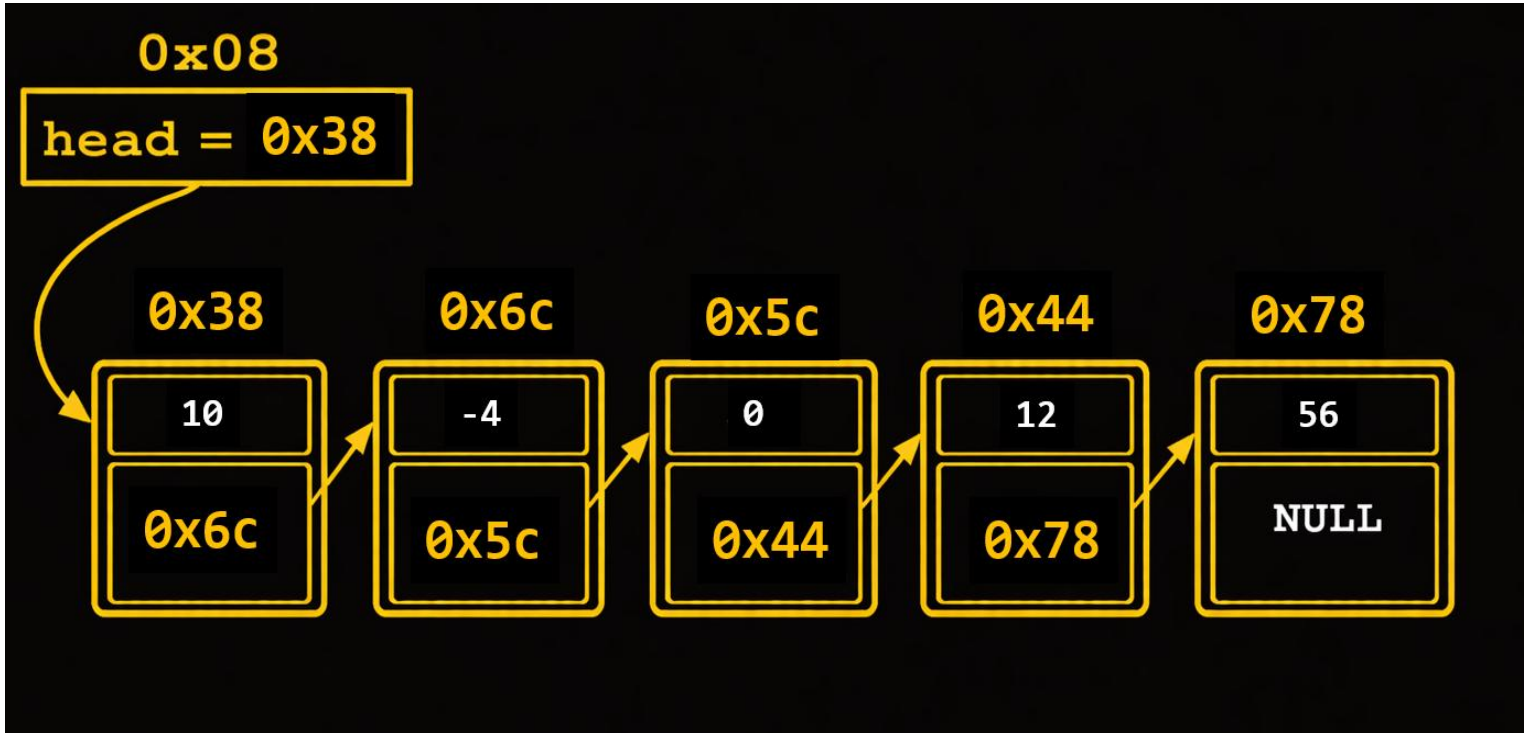
How could we print the data from the **first** node in this linked list?



Printing a Node

How could we print the data from **each** node in this linked list?

We need to traverse the list



Traversing a Linked List

Traversing a list means **starting at the head node** and **moving** through **each node** one **at a time** until reaching the end (**NULL**).

We commonly **traverse** a linked list to:

- **print the data** stored in each node,
- determine **how many nodes** are in the list, or
- **search** for a **specific** value within the list.

Traversing a Linked List

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

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

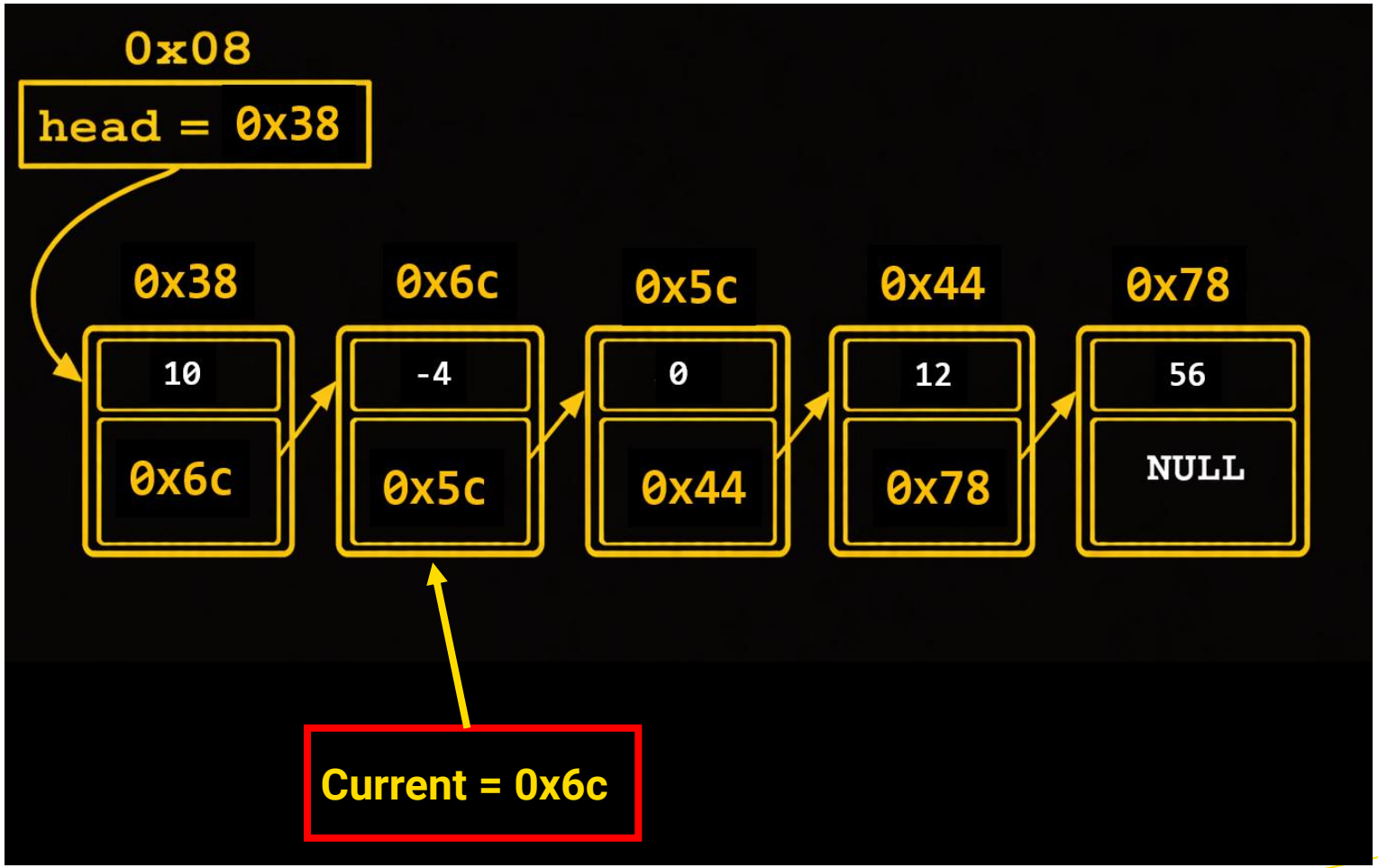
Traversing a Linked List

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

Traversing a Linked List

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

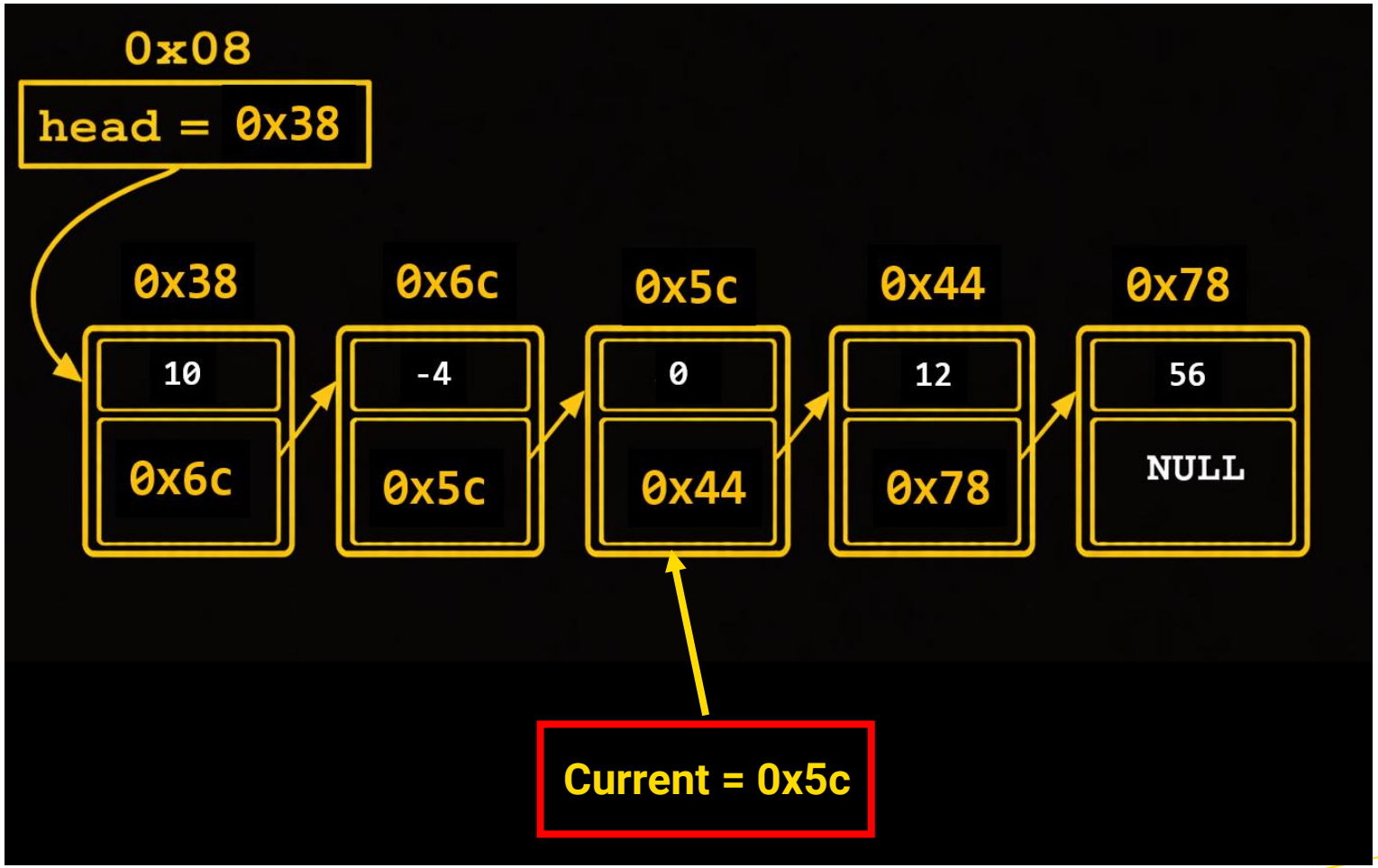
Then we should **move current** to the next nodes one by one



Traversing a Linked List

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

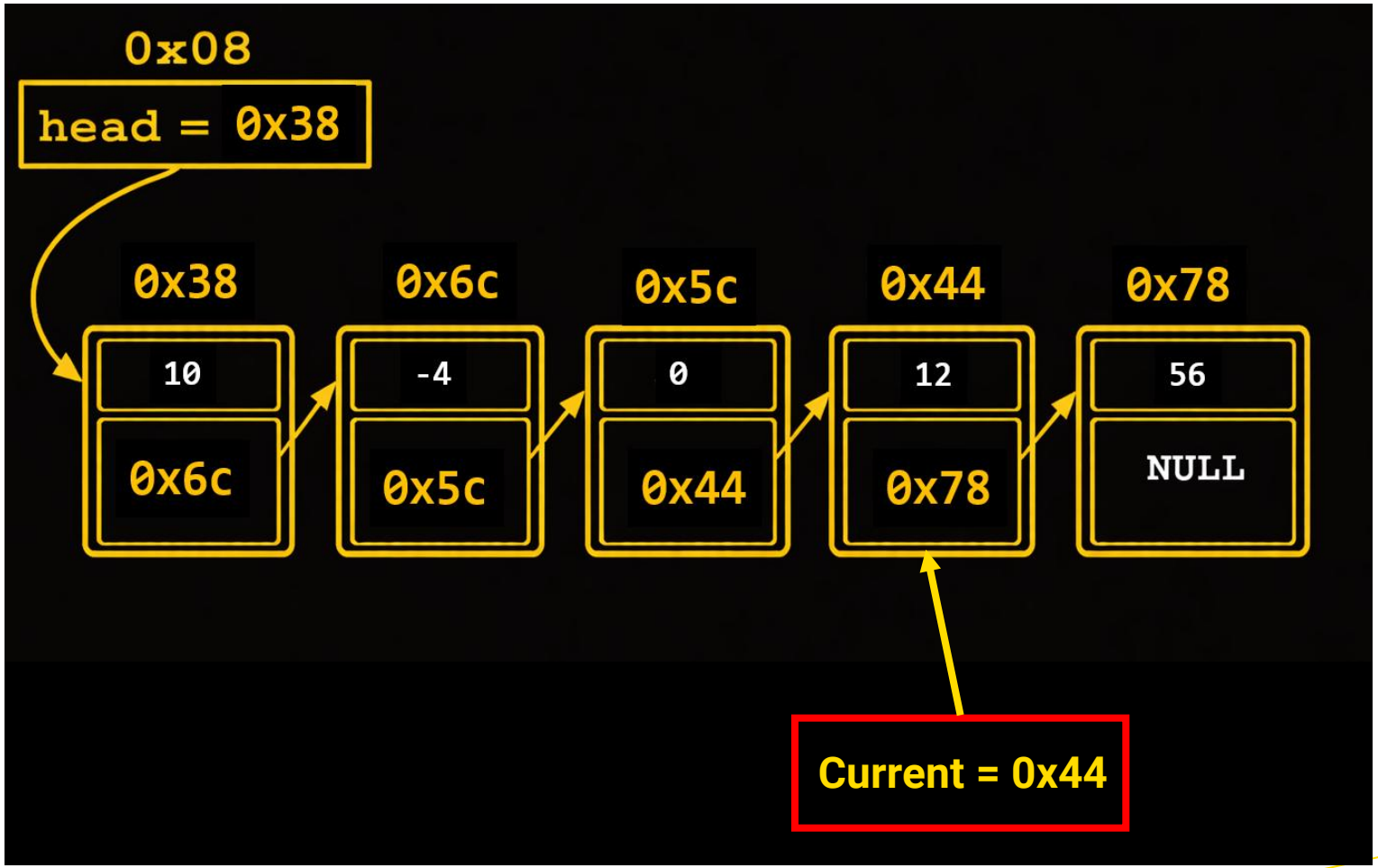
Then we should **move current** to the next nodes one by one



Traversing a Linked List

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

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



Traversing a Linked List

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

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

Traversing a Linked List

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

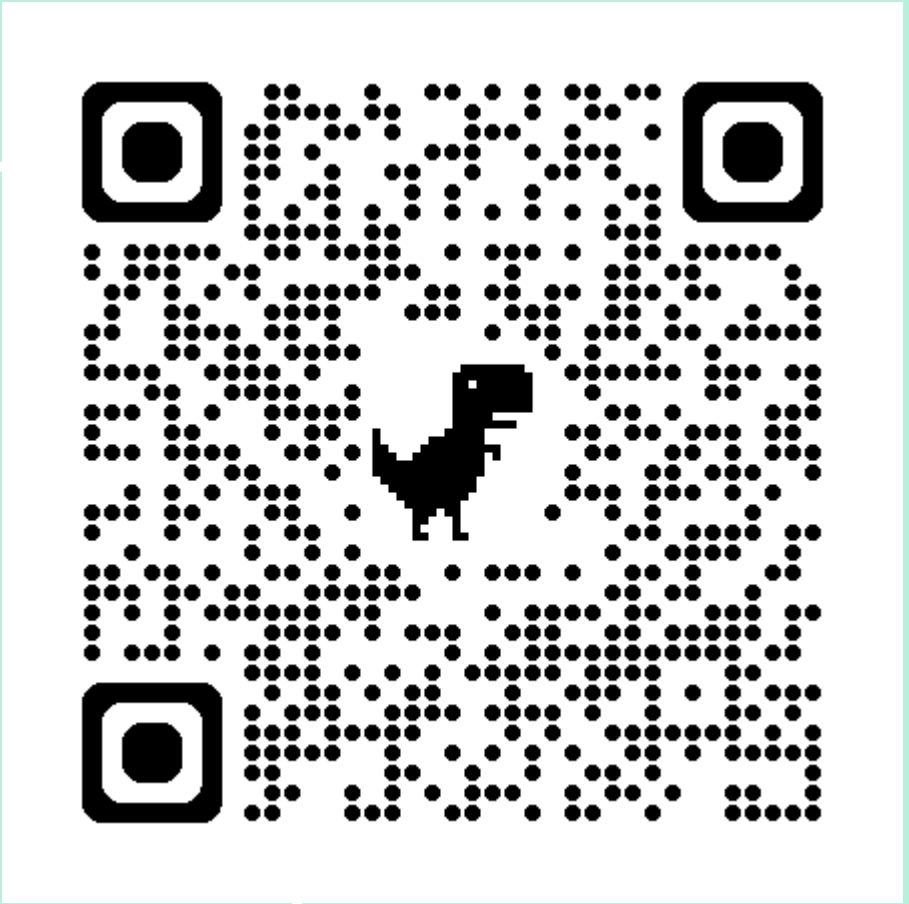
We should **stop** when **current = NULL**

Printing a Linked List

```c
// Traverse the list and print the
// data stored in each node
void print_list(struct node *head) {

    struct node *current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Demo

Print_list_of_three_nodes.c



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

UNSW College

Inserting nodes in a Linked List

Where can we insert in a list?

- **At the head** (what we just did! Slide 37)
- **Between any two nodes** that exist (next lecture!)
- **After the tail** as the last node (now!)

Insert at the tail

To add a new node to the **end of a linked list**, we must:

Locate the **last node** in the list

Update its **next pointer** so that it **points to the new node**

How can I move through the list to find the last node?

We should traverse the list

But if we continue **traversing** until `current == NULL`, we have moved **beyond** the **last node** (the tail) of the list!!!

How can I move through the list to find the last node?

<u>We want to stop at the last node</u>

How can we tell if we are at the last node?

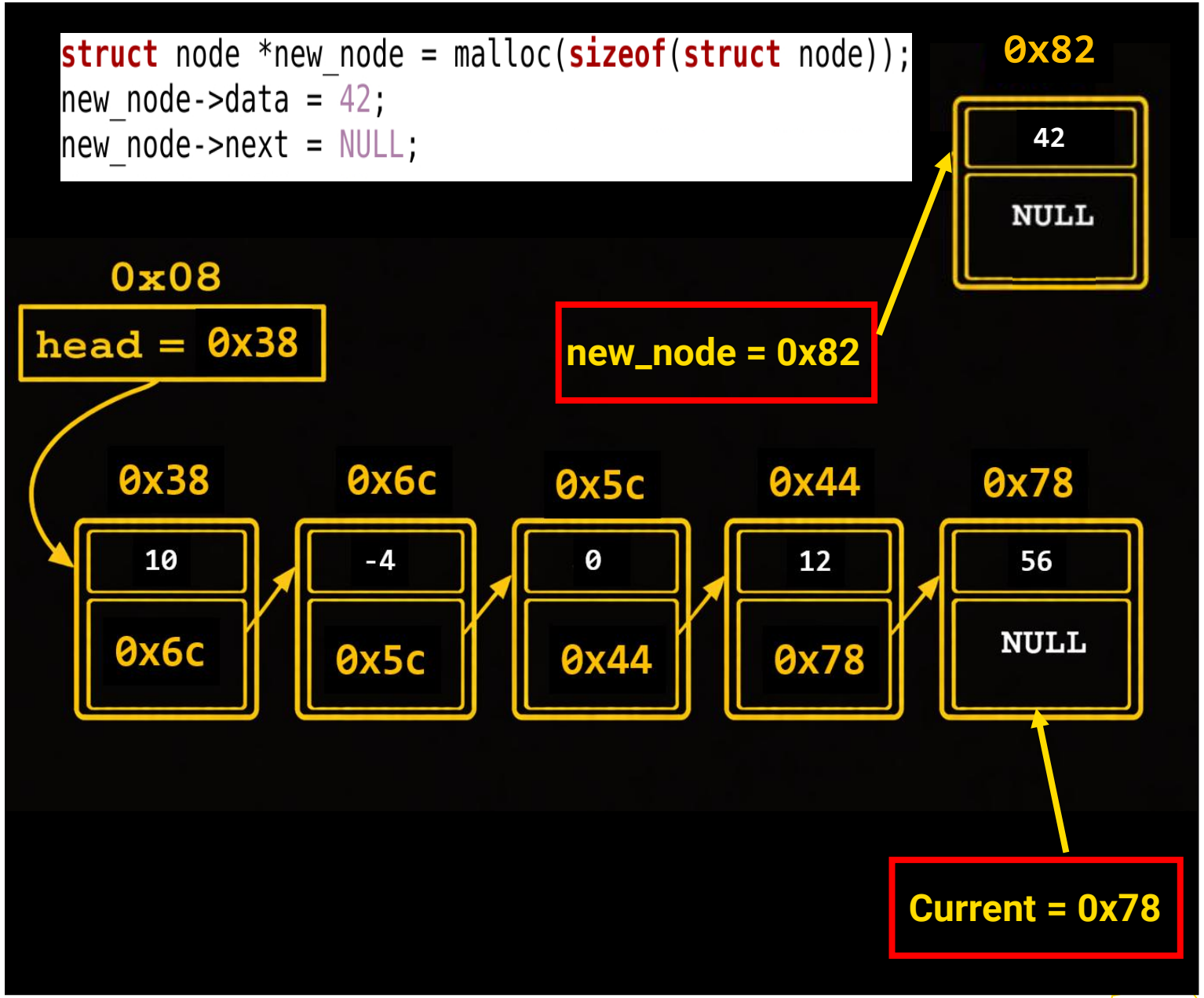
How can I move through the list to find the last node?

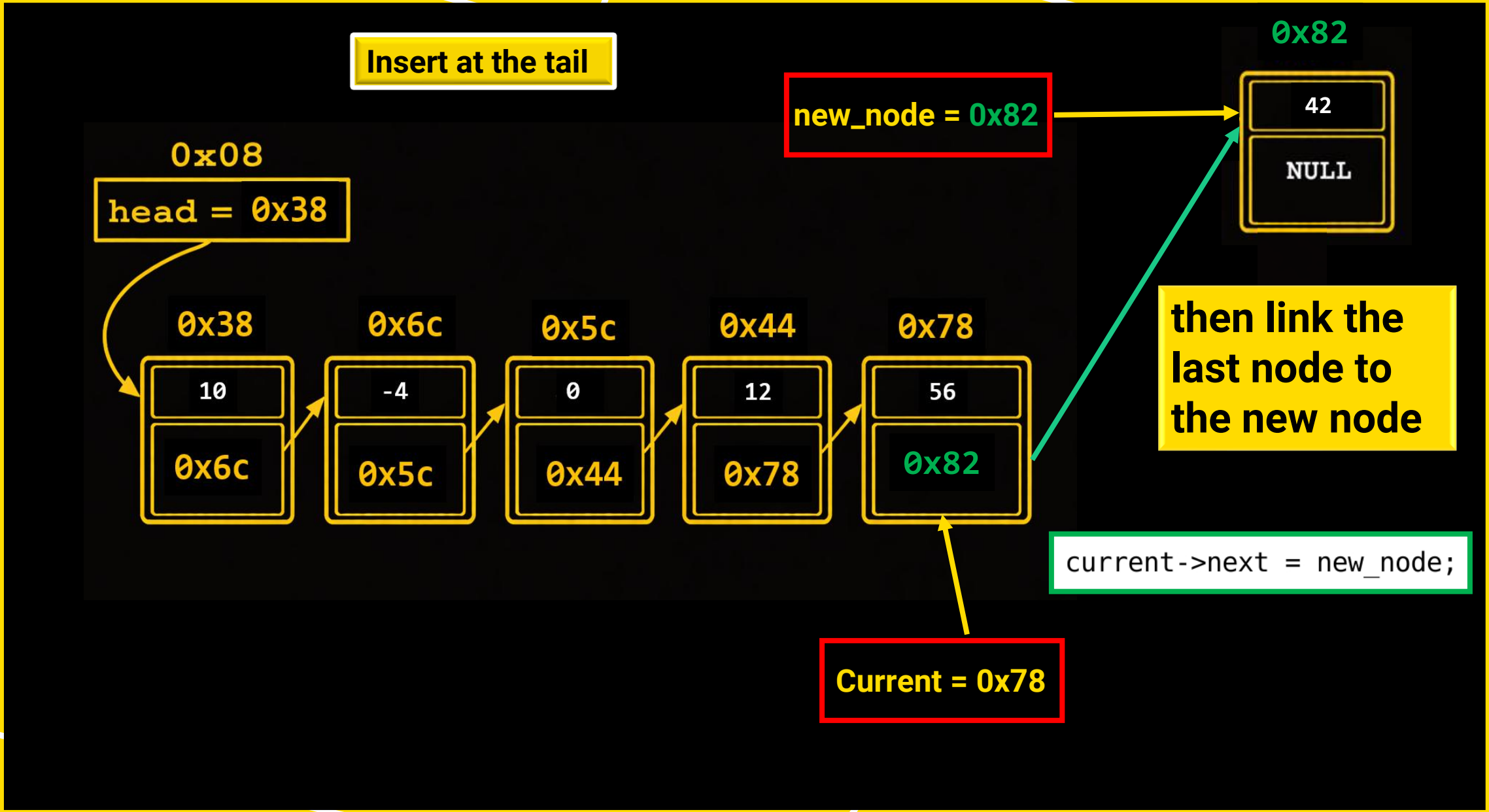
We should stop when `current->next == NULL`

We want to stop at the last node

Insert at the tail

Now we can make the the new node first,




Insert at the tail
0x08
head = 0x38
0x38
10
0x6c
0x6c
-4
0x5c
0x5c
0
0x44
0x44
12
0x78
0x78
56
0x82
new_node = 0x82
0x82
42
NULL
then link the last node to the new node
current->next = new_node;
Current = 0x78

Insert at the tail function

Big issue

```c
// What would cause this function to break?
void insert_at_tail(struct node *head, int data){

    struct node *current = head;

    // Find the tail of the list
    while (current->next != NULL) {
       current = current->next;
    }

    // Create the new node
    struct node *new_node = create_node(data, NULL);

    // link the new node to the tail of the list
    current->next = new_node;

}
```

Testing Linked List Functions

It is essential to **consider different scenarios**:

- An empty list
- A list containing a single node
- A list containing multiple nodes

Our function only inserts nodes **at the end of the list**. If we were developing a function that **inserts a node at any position,** we would **also** need to test:

- Inserting at the beginning of the list
- Inserting in the middle of the list
- Inserting at the end of the list

Inserting At Tail Code Bug

If we have an **empty list then,**

`head == NULL;`

then `current == NULL;`

it means that `current->next` will be **dereferencing** a `NULL` **pointer** and result in a **run time error**

```
void insert_at_tail(struct node *head, int data){

    struct node *current = head;

    // Find the tail of the list
    while (current->next != NULL) {
```

Inserting At Tail

Is this a better code?

```c
void insert_at_tail(struct node *head, int data) {

    // Create a new node with the given data
    struct node *new_node = create_node(data, NULL);

    // Handle the situation where the list is currently empty
    if (head == NULL) {
        head = new_node;
    } else {

        // Start from the head to traverse the list
        struct node *current = head;

        // Move through the list until we reach the final node
        while (current->next != NULL) {
            current = current->next;
        }

        // link the new node to the tail of the list
        current->next = new_node;
    }
}
```

Inserting At Tail

The program no longer crashes. However, the list is **still empty** after calling the function. Why does this happen??

```c
int main(void) {

    struct node *head = NULL;

    insert_at_tail(head, 42);

    // local variable head is in main is still NULL
    return 0;
}
```

How to fix it?

We need to change the function prototype so that it returns the head of the list and then update our local variable by assigning it to the returned value.

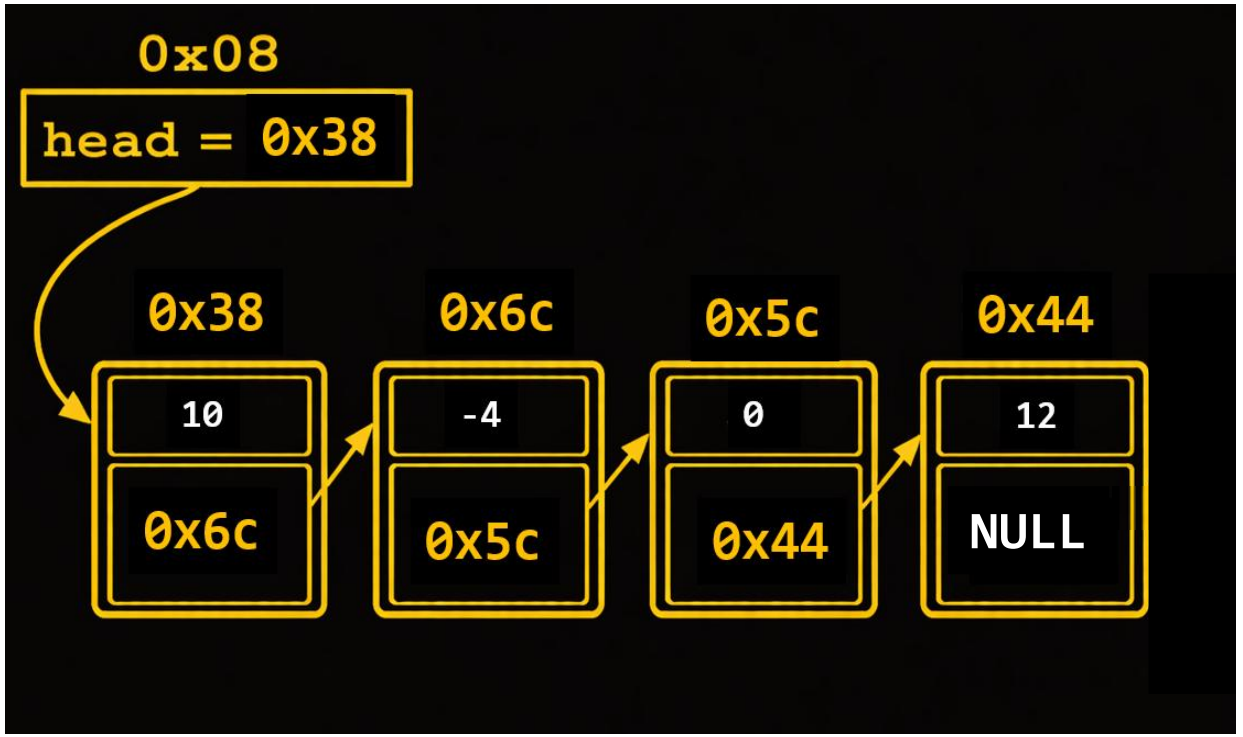
```c
struct node *insert_at_tail(struct node *head, int data);
int main(void) {

    struct node *head = NULL;

    // updating local variable head
    head = insert_at_tail(head, 42);

    return 0;
}
```

So, the final Insert at Tail function would be

```c
struct node *insert_at_tail(struct node *head, int data) {

    // Create a new node with the given data
    struct node *new_node = create_node(data, NULL);

    // Handle the situation where the list is currently empty
    if (head == NULL) {
        head = new_node;
    } else {

        // Start from the head to traverse the list
        struct node *current = head;

        // Move through the list until we reach the final node
        while (current->next != NULL) {
            current = current->next;
        }

        // link the new node to the tail of the list
        current->next = new_node;
    }
    return head;
}
```

Inserting in the Middle of the List

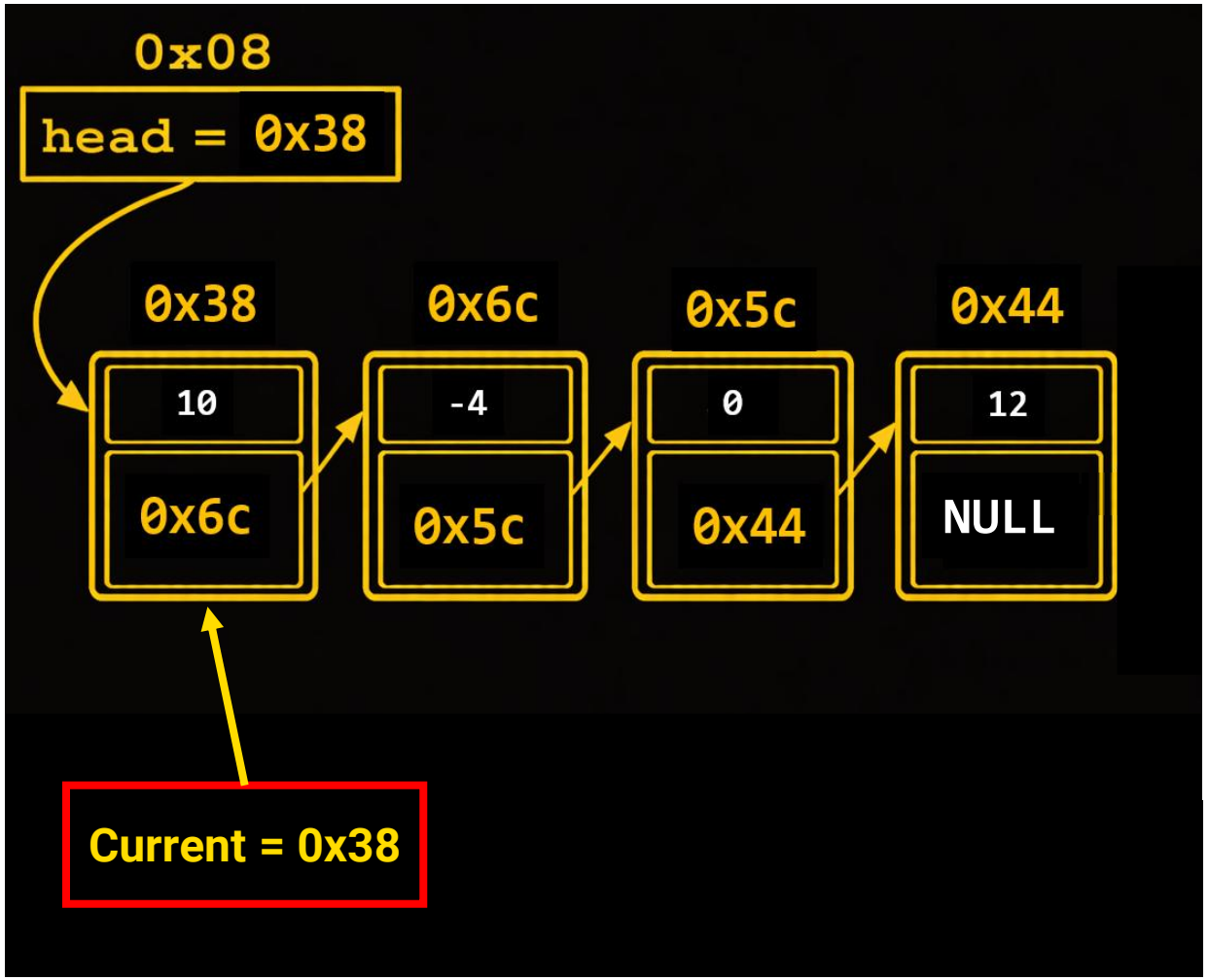
We want to **insert a new node** at the **position given by size / 2**, assuming positions **begin counting from 0**. In this example, that means the new node should be inserted at position 2.



This means, we should **use a counter** and **stop traversing** when we get to the node **before the position** we want to insert at **(size/2 - 1).** In this case position 1.

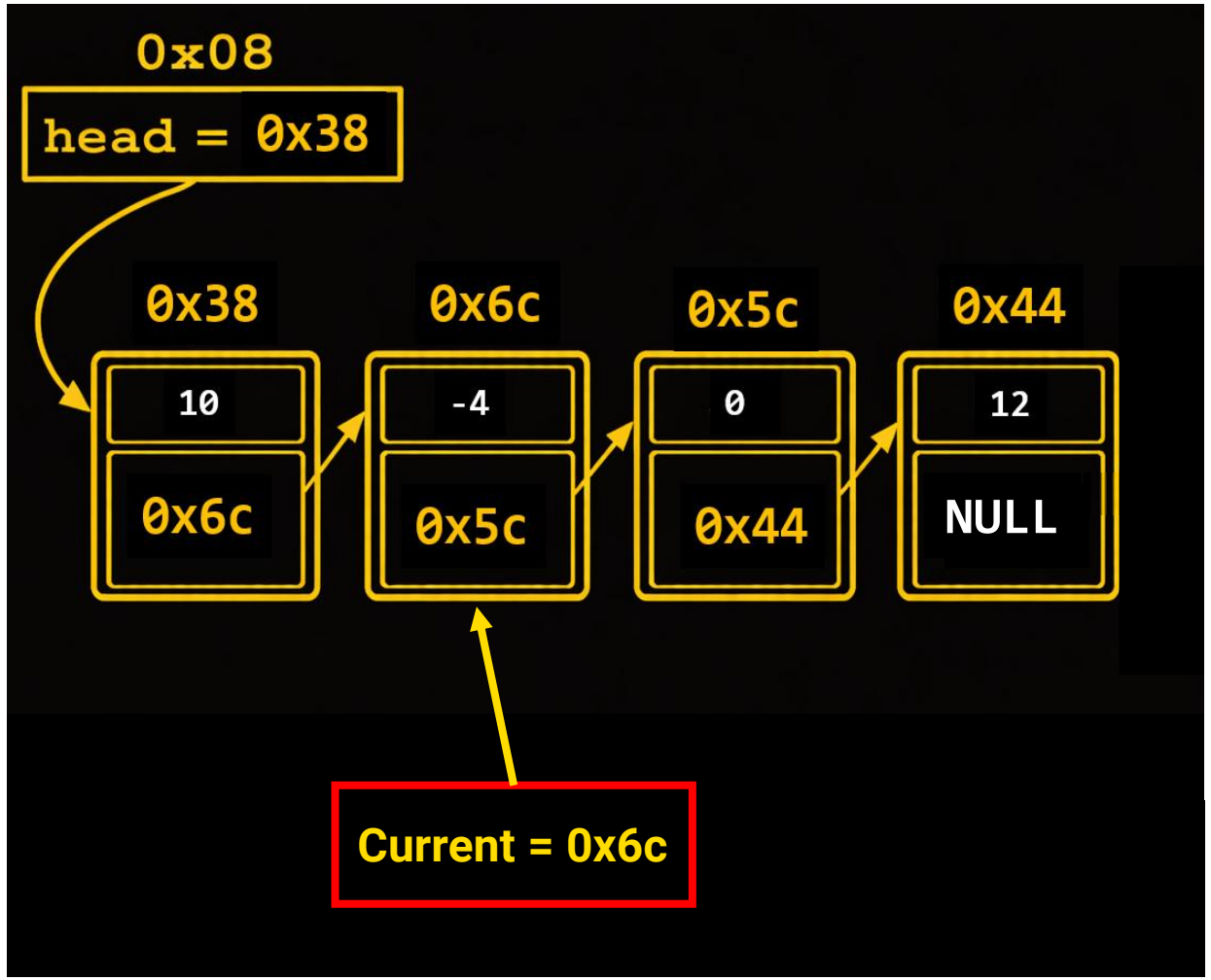
Inserting in the Middle of the List

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



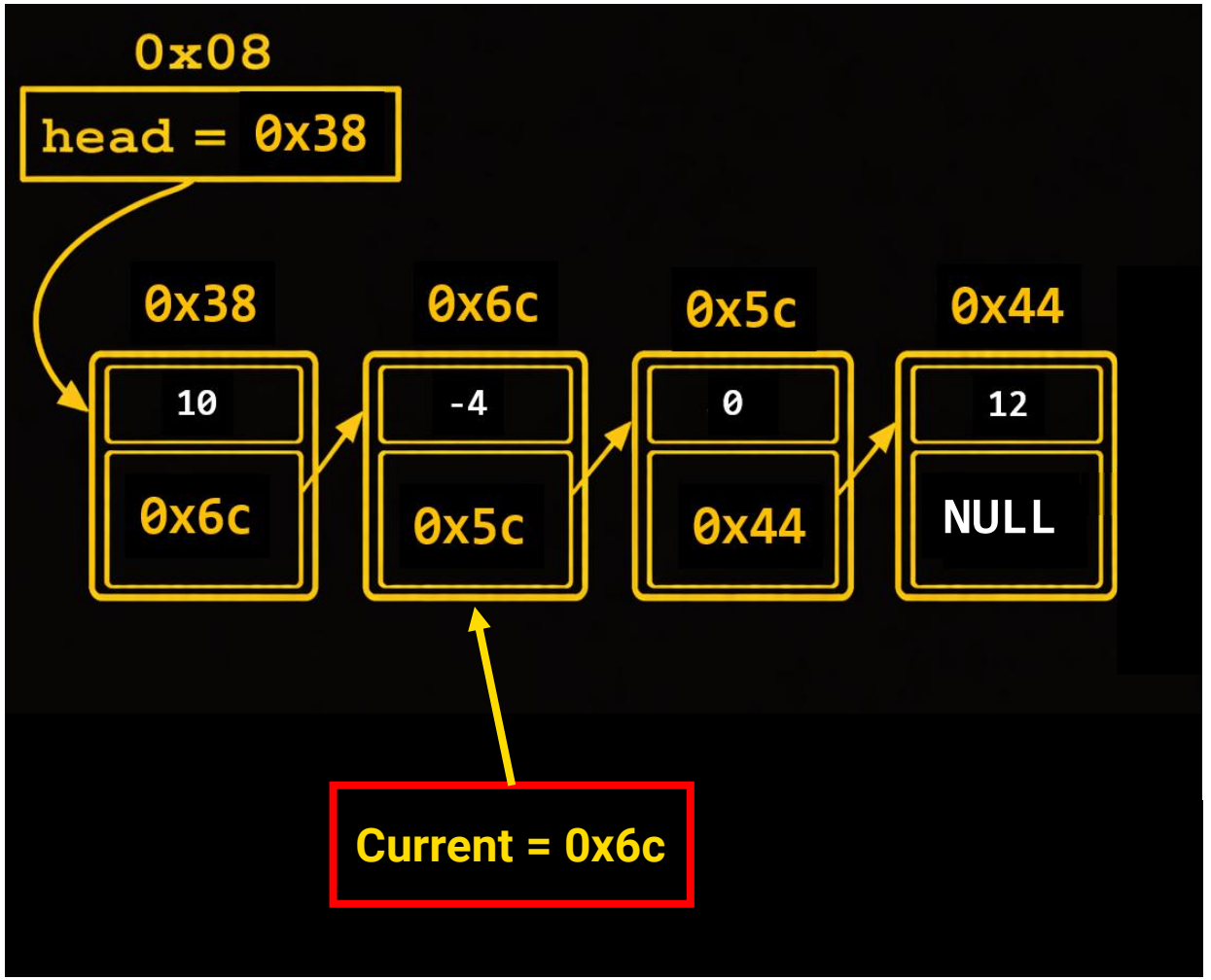
Inserting in the Middle of the List

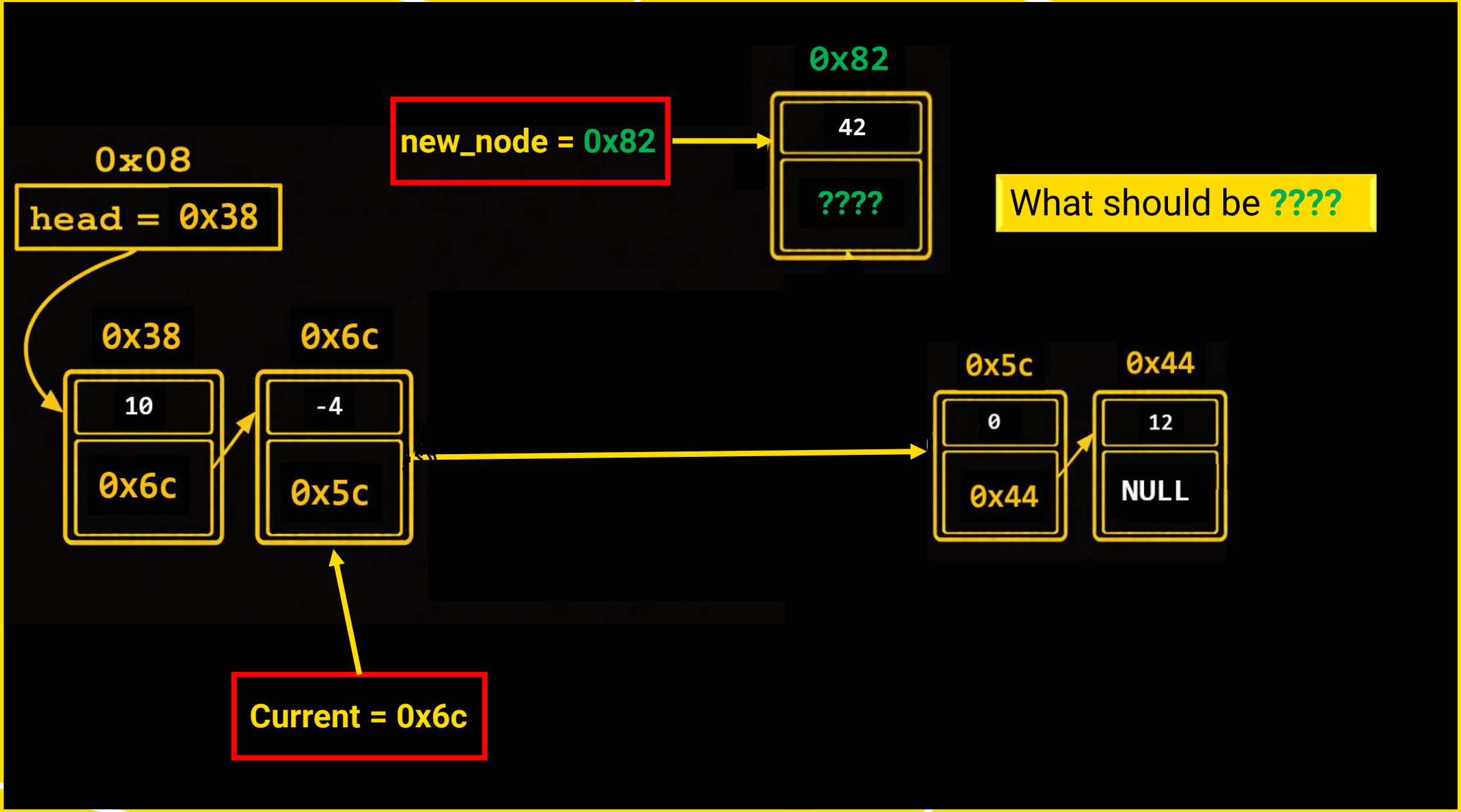
```
while (counter < size/2 - 1) {
    current = current->next;
    counter++;
}
```

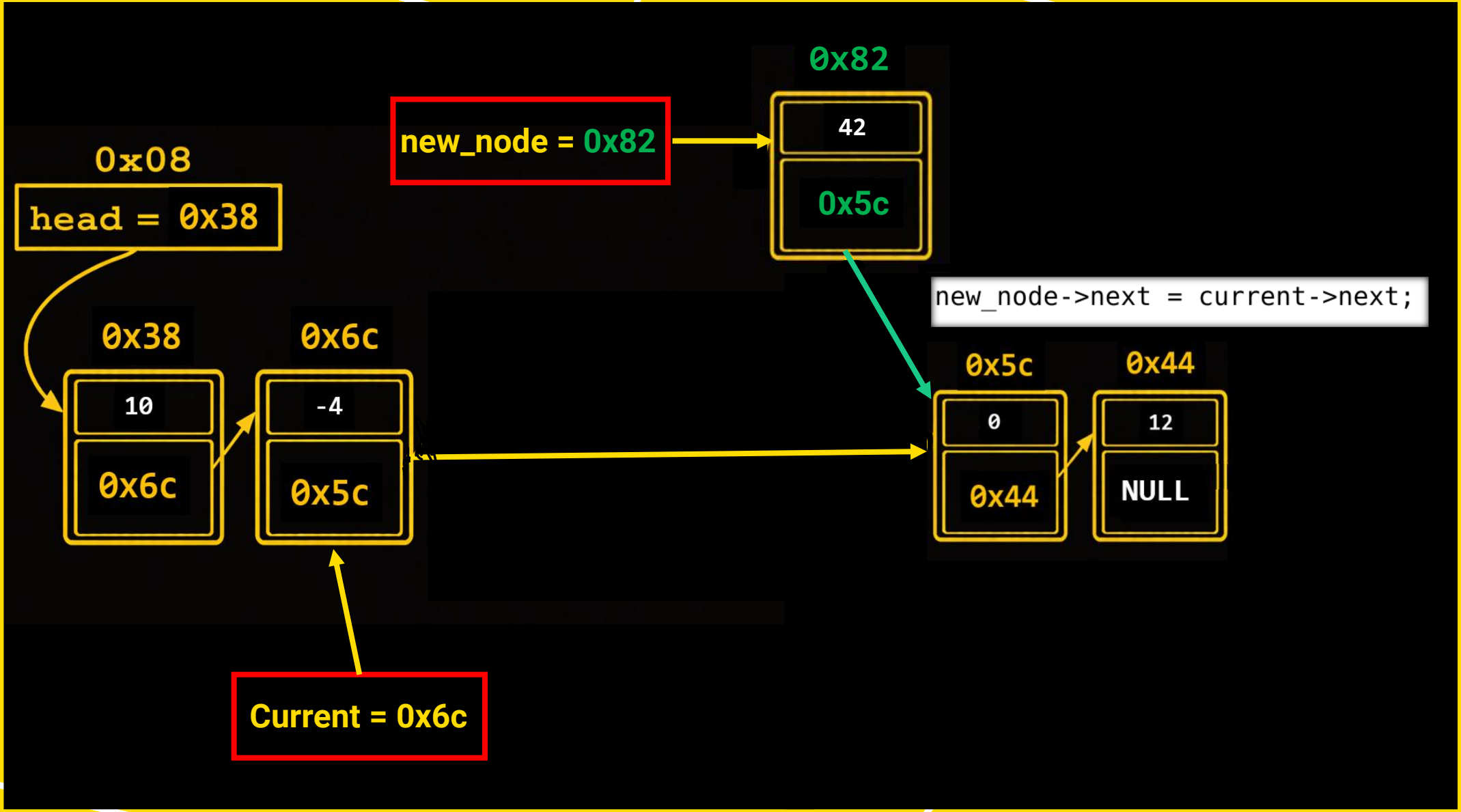


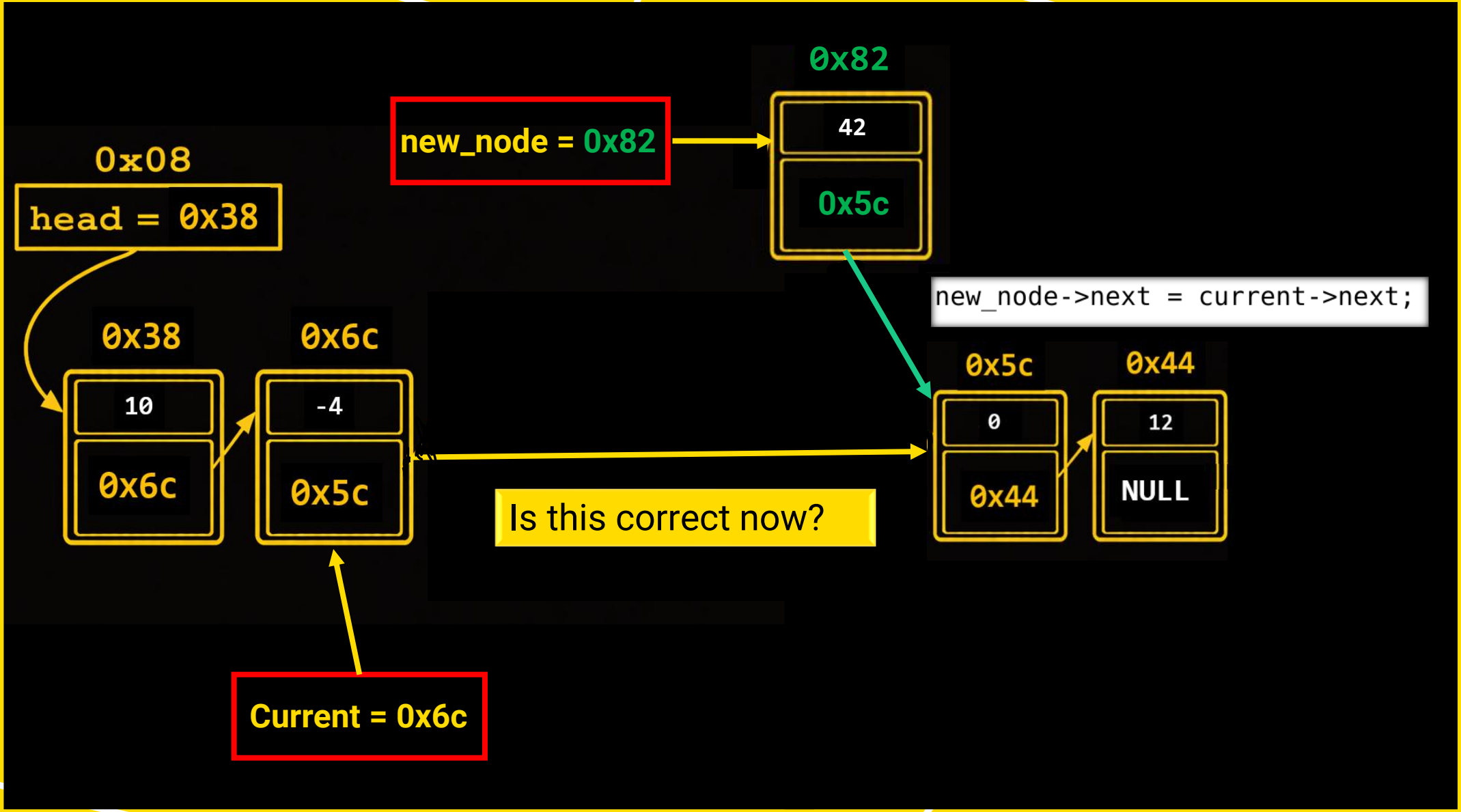
Inserting in the Middle of the List

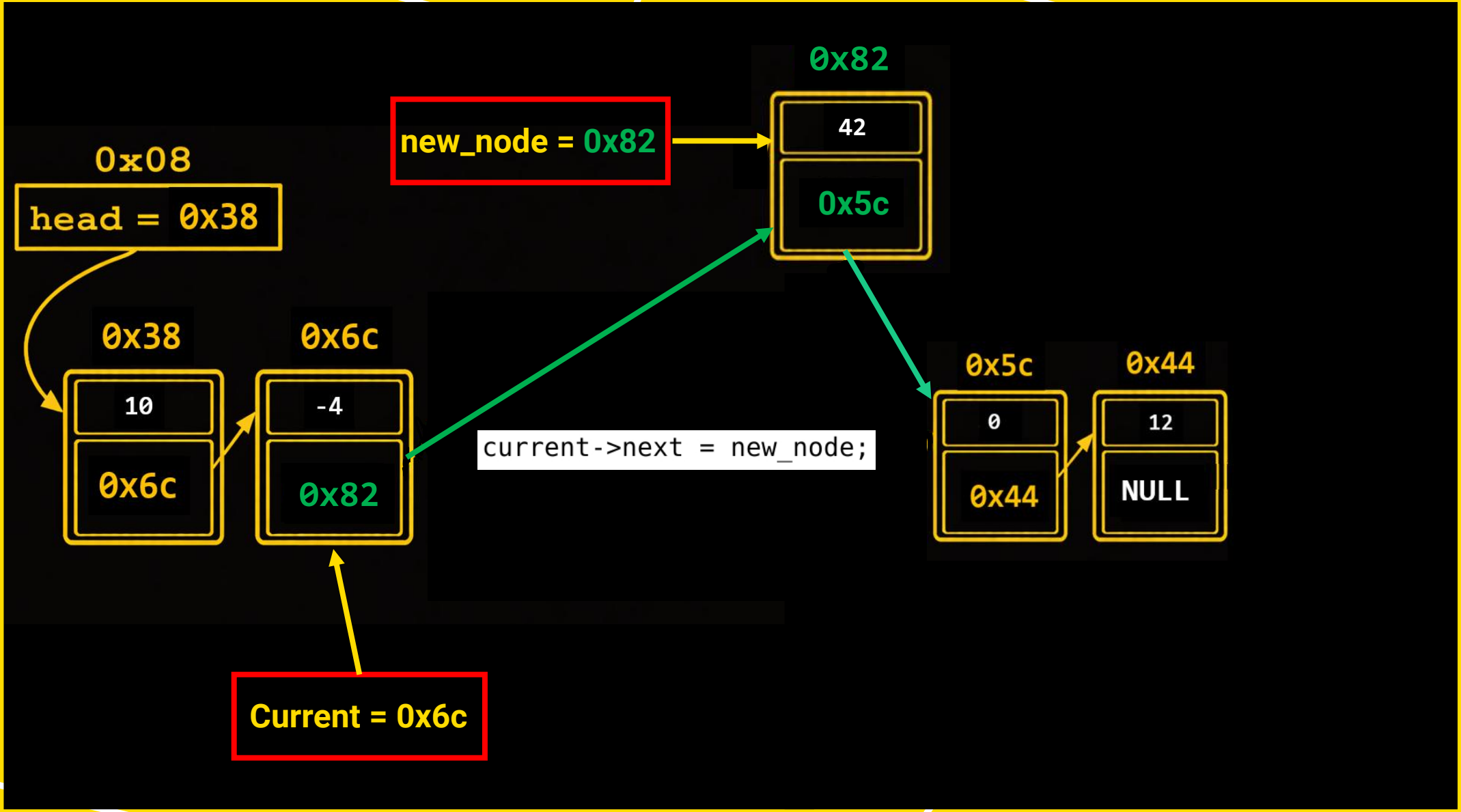
Now we want to connect our **new node**. It should come **after the current node**, but **before `current->next`**




0x82
42
????
new_node = 0x82
0x08
head = 0x38
What should be ????
0x38
0x6c
0x5c
0x44
10
0x6c
-4
0x5c
0
0x44
12
NULL
Current = 0x6c


0x82
42
0x5c
new_node = 0x82
0x08
head = 0x38
new_node->next = current->next;
0x38
0x6c
10
0x6c
-4
0x5c
0x5c
0x44
0
0x44
12
NULL
Current = 0x6c


0x82
42
0x5c
new_node = 0x82
0x08
head = 0x38
new_node->next = current->next;
0x38
0x6c
10
0x6c
-4
0x5c
0x5c
0x44
0
0x44
12
NULL
Is this correct now?
Current = 0x6c


0x82
42
0x5c
new_node = 0x82
0x08
head = 0x38
0x38
10
0x6c
0x6c
-4
0x82
current->next = new_node;
0x5c
0
0x44
0x44
12
NULL
Current = 0x6c

What would be the 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?

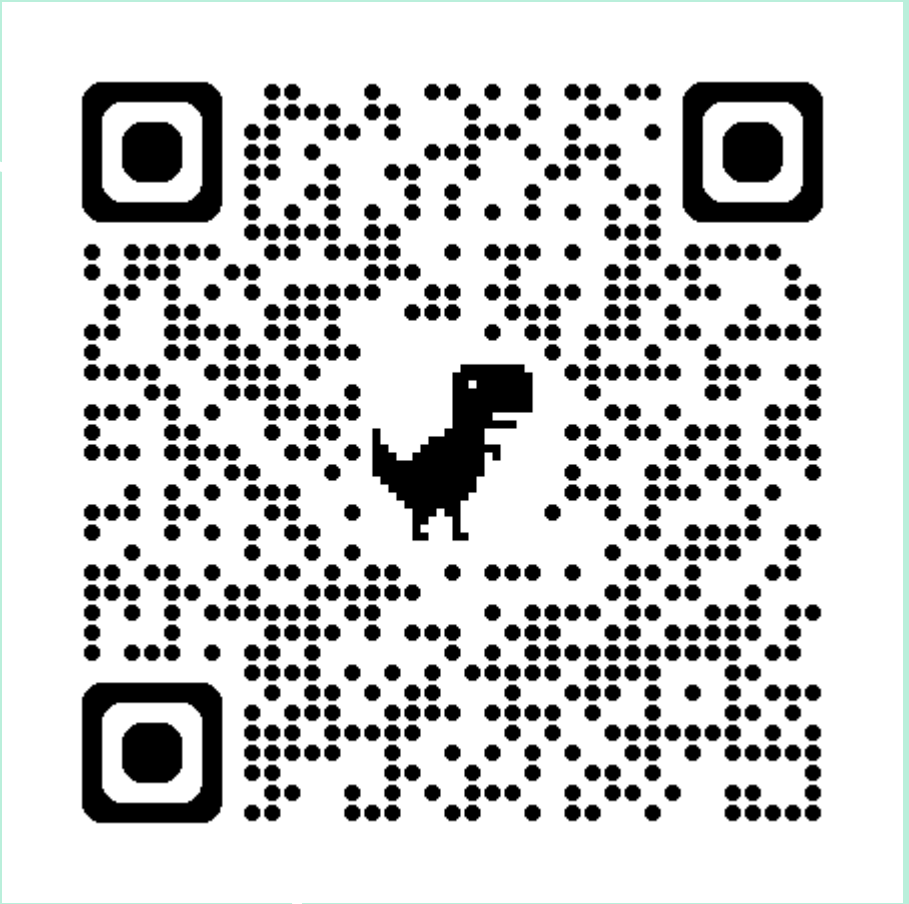
Inserting Into a Linked List Test Cases

Draw a diagram!!!

- Inserting into an empty list
- Inserting at the beginning (head) of the list
- Inserting after the first node when there is only one node
- Inserting somewhere in the middle of the list
- Inserting at the end of the list

Demo

list_create_insert_print_length.c



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

UNSW College

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me

26T1 Voice of the Student – Fill out form

See you soon …

UNSW College