DPST1091 / CPTG1391

Introduction to Programming

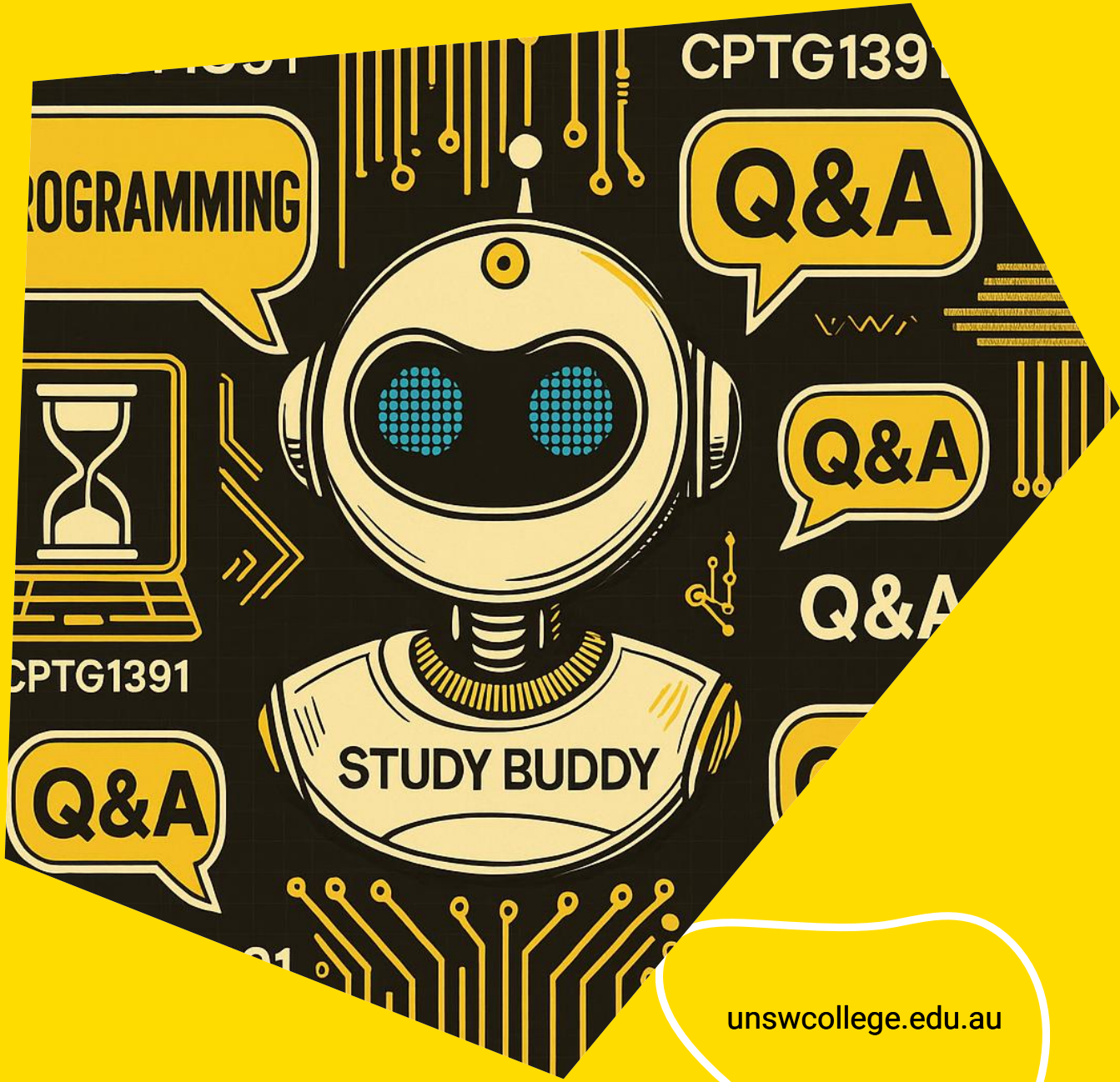
Week 9 – Lecture 1

Lecturer and Course Convener:

Dr Pantea Aria

UNSW College

Introduction to Linked Lists



unswcollege.edu.au

Agenda

- **Last lecture**
  - **Dynamic Memory, malloc and the heap**
  - **Multi-File Projects**
- y:

- **Today**
  - **Introduction to Linked Lists**

The Heap recap

We recall that when a stack frame is created, enough memory to store everything in the frame is allocated to the frame

What if the **amount of memory** needed **isn't known in advance**?

In this case, we **can't use stack memory**.

The program must know **exactly how much space** a stack frame requires **before** the function starts running.

Because the **size is unknown at compile time,** the stack cannot be used to allocate this memory.

Why do we need the heap?

- We want to **create arrays** inside functions and **return them**
- We also want to create arrays whose **size is only known at runtime**
- Unlike stack memory, heap memory is allocated **explicitly by the programmer**
- Heap memory **remains allocated** until the programmer **frees** it
- This gives you **full control over memory** allocation and lifetime
- **But remember with great power comes great responsibility**

C provides us some functions to interact with the heap.

malloc

Allocates memory **on the heap**

Returns a **pointer** to the allocated memory

Allows the programmer to **choose the size** of the allocation

```
#include <stdlib.h>
```

The NULL Pointer

What does malloc return?

- A **pointer to the allocated block of memory**, or
- **NULL** if the allocation fails (not enough memory available)
- Because allocation can fail, you should **always check that the returned pointer is not NULL** before using it.

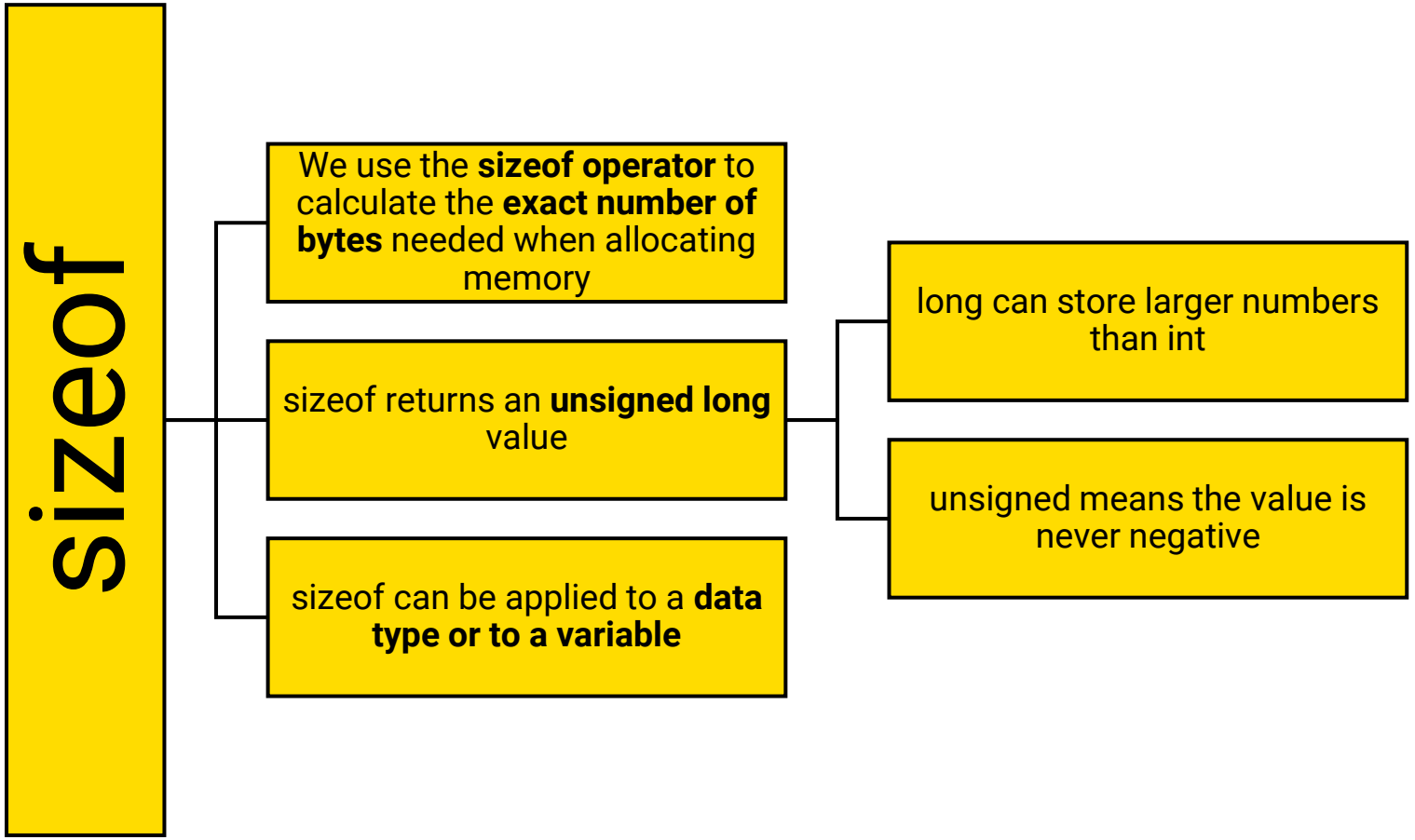
The NULL Pointer recap

- Sometimes we initialise a pointer with a **special value** to indicate that it is **not pointing to any valid memory yet**
- This special value is called **NULL**
- Dereferencing a **NULL** pointer will cause a **runtime error**

```c
int *my_ptr = NULL;

// Dereferencing a NULL pointer
// causes a runtime error
printf("%d\n", *my_ptr);
```

```c
// Safe usage
// Always check that a pointer is not NULL before
// dereferencing it

int *my_ptr = NULL;

if (my_ptr != NULL) {
    printf("%d\n", *my_ptr);
}
```

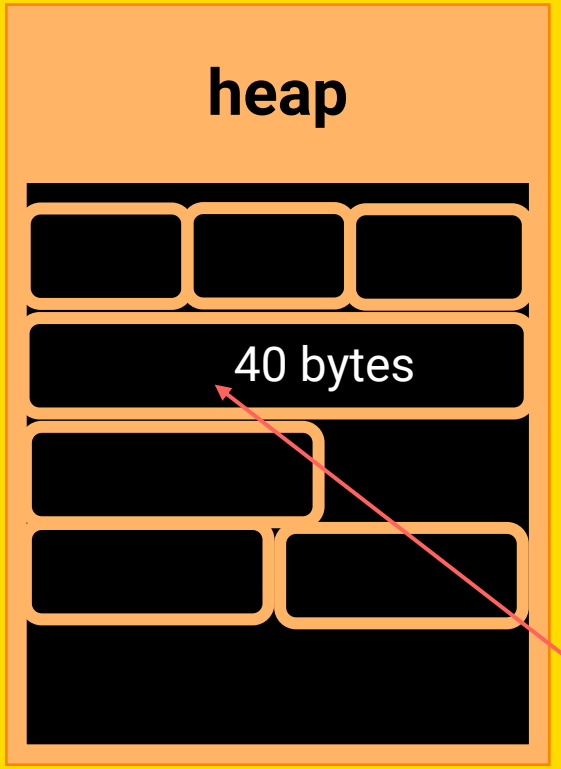
Sizeof Operator



Example

```c
#include <stdio.h>

int main(void) {
    double values[5];

    printf("Size of a double: %lu bytes\n", sizeof(double));
    printf("Size of 5 doubles: %lu bytes\n", 5 * sizeof(double));

    return 0;
}
```

Using malloc



To calculate how much memory to allocate, **multiply the number of elements** by **the size of each element's type** using sizeof.

This gives the total number of bytes that malloc should allocate.

malloc then returns a **pointer to the first byte** of the allocated block of memory.

```c
int *numbers = malloc(10 * sizeof(int));
```

The Free Function

- **free** tells the system that a **block of heap memory** is **no longer needed**
- **Every call to malloc** should have a **matching call to free**
- If memory is allocated but never freed, the program will **consume more and more memory** — this is known as a **memory leak**
- Memory leaks are especially problematic in **long-running programs**
- Although the operating system reclaims memory when a program exits, relying on this is **not good practice**
- Using memory **after it has been freed**, or **freeing it more than once**, can lead to **serious and hard-to-debug errors**

Example:

```c
// allocate space for values
double *values = malloc(count * sizeof(double));

// check malloc was successful
if (values == NULL) {
    return 1;
}

// use the array
// (e.g. values[0] = 3.14;)

// free the memory when done
free(values);
```

You can check for **memory leaks** using dcc with the flag `dcc --leak-check`

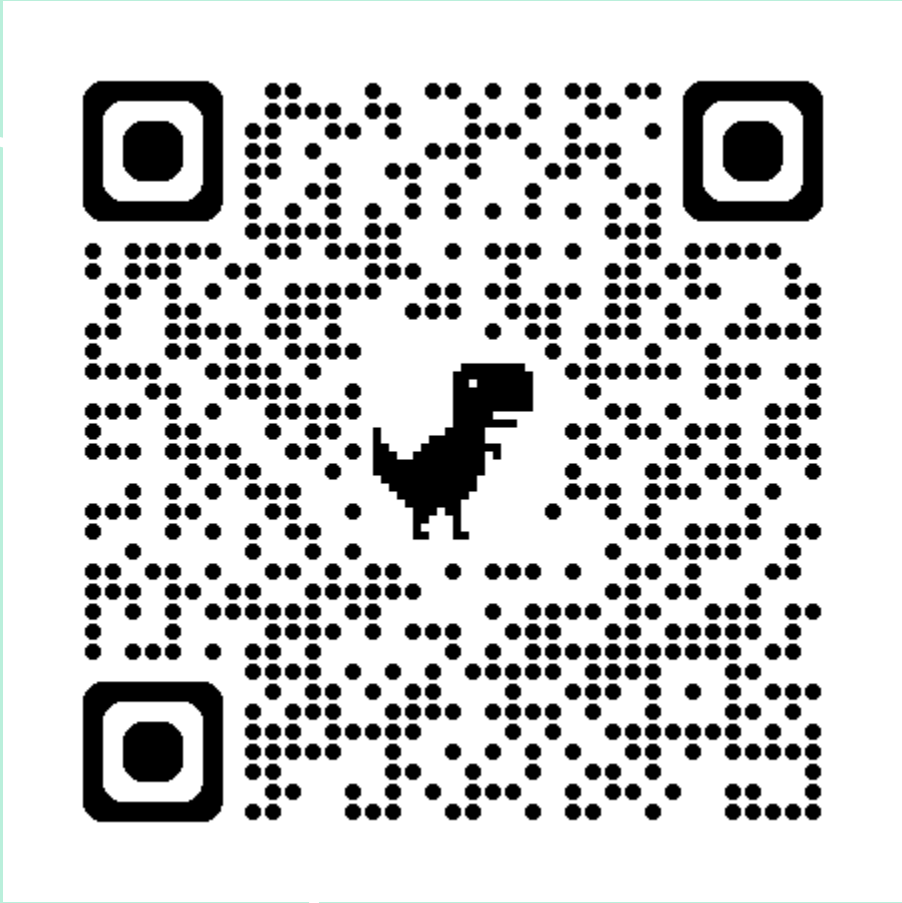
The realloc Function

If the array was created dynamically, we can resize it using **`realloc`**.

```c
// Allocate space for 10 integers
int *numbers = malloc(10 * sizeof(int));

// Later, we realise we need more space
// realloc increases the size without
// losing the existing data
numbers = realloc(numbers, 20 * sizeof(int));
```

Demo

malloc_recap.c



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

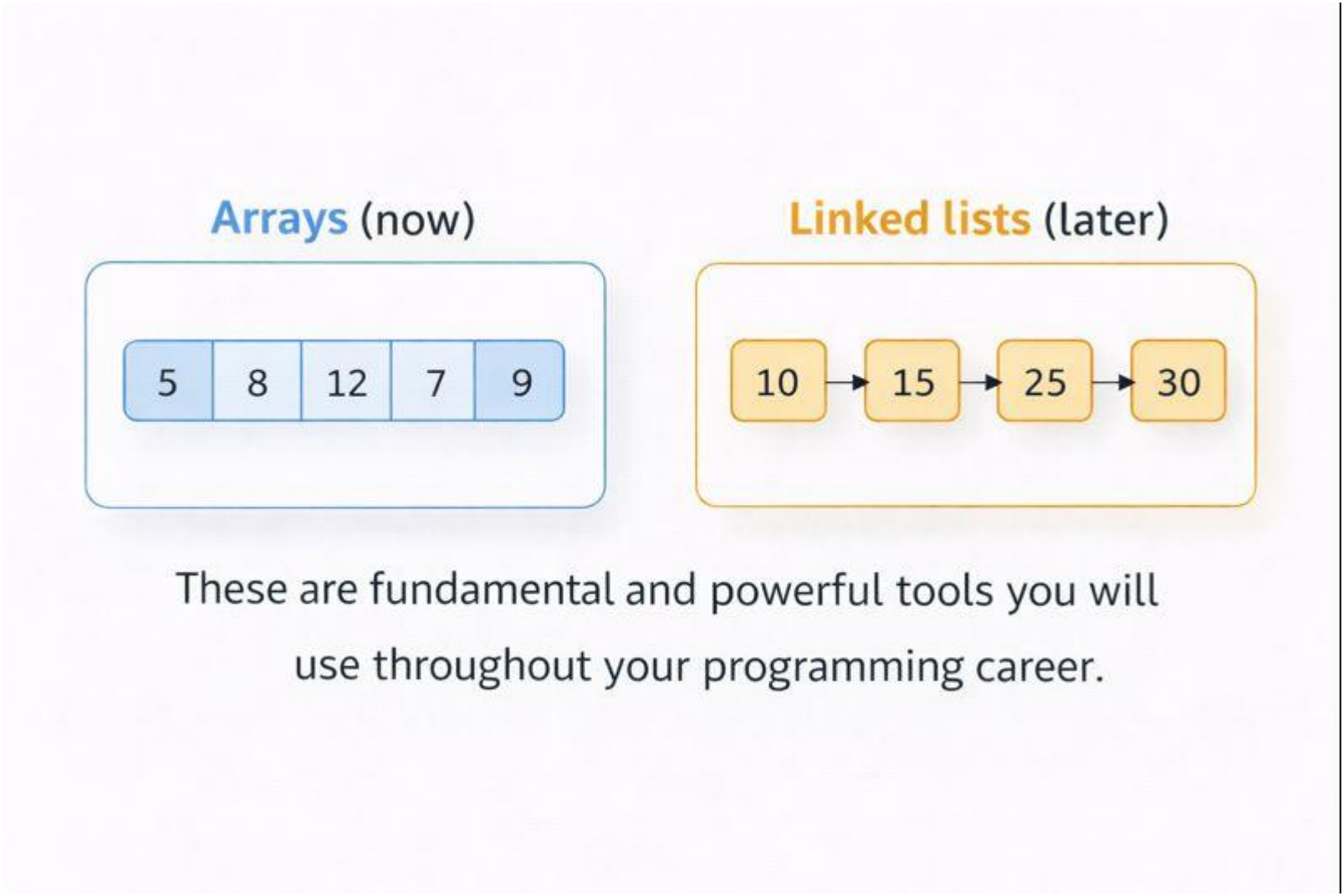
UNSW College

Linked Lists

UNSW College

Recall from week 3

Data structures we will use in our course:



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

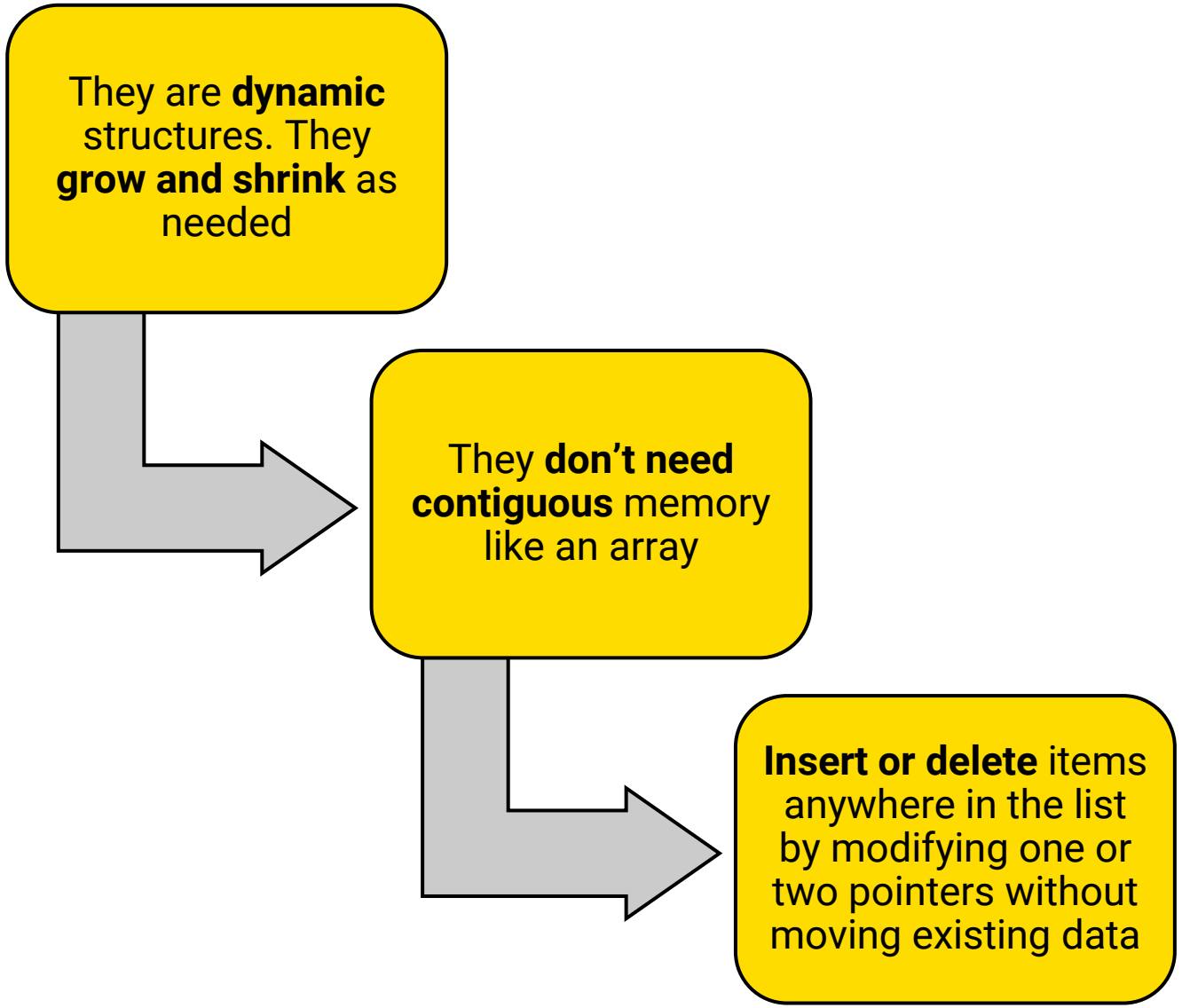
Arrays Advantages

- 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.

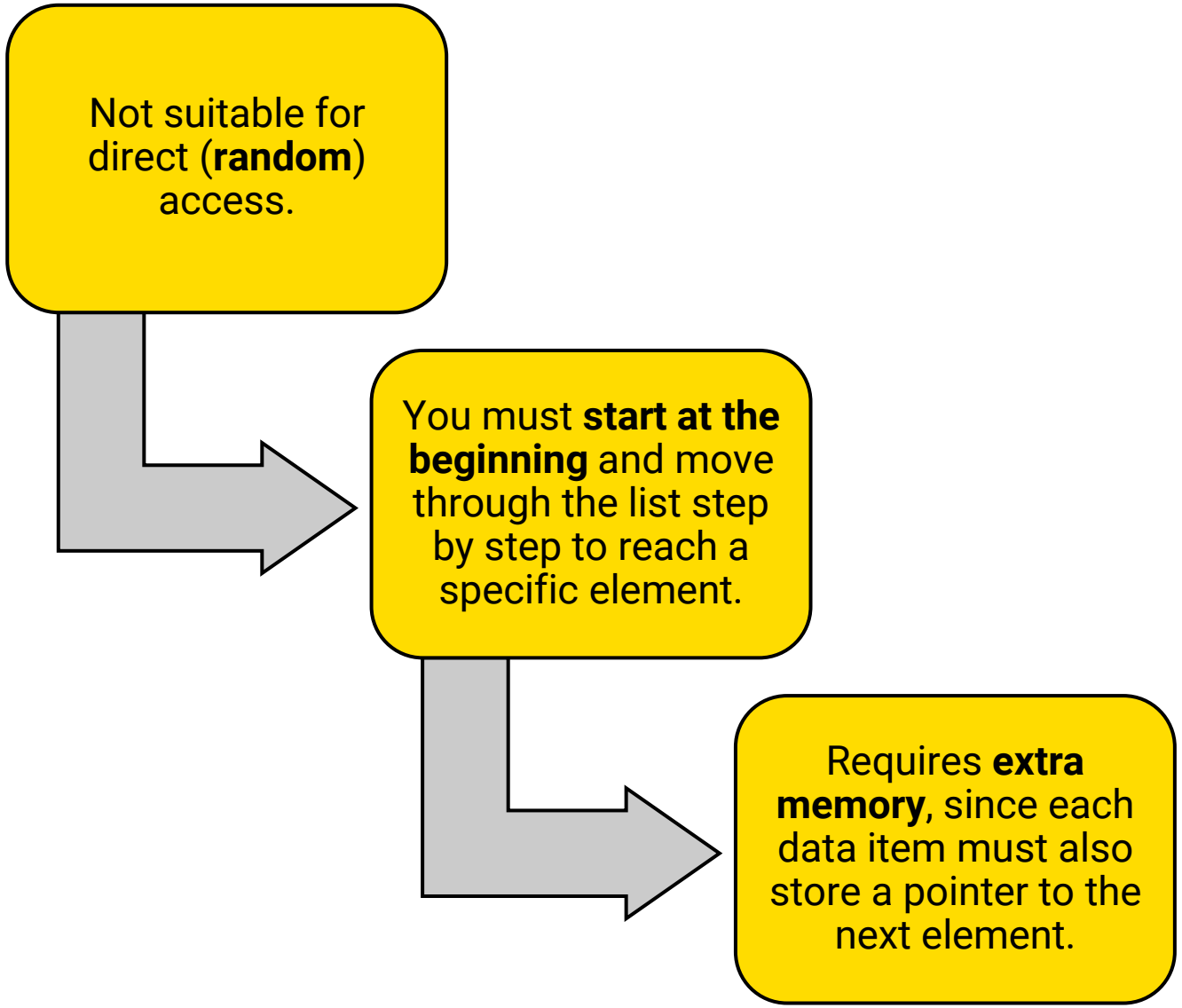
Arrays Disadvantages

- 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 in the Memory

The name of an array represents the address of the first element in memory.

Arrays are stored in contiguous (adjacent) memory locations, which enables us to use indexing for fast and efficient random access.

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

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

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	

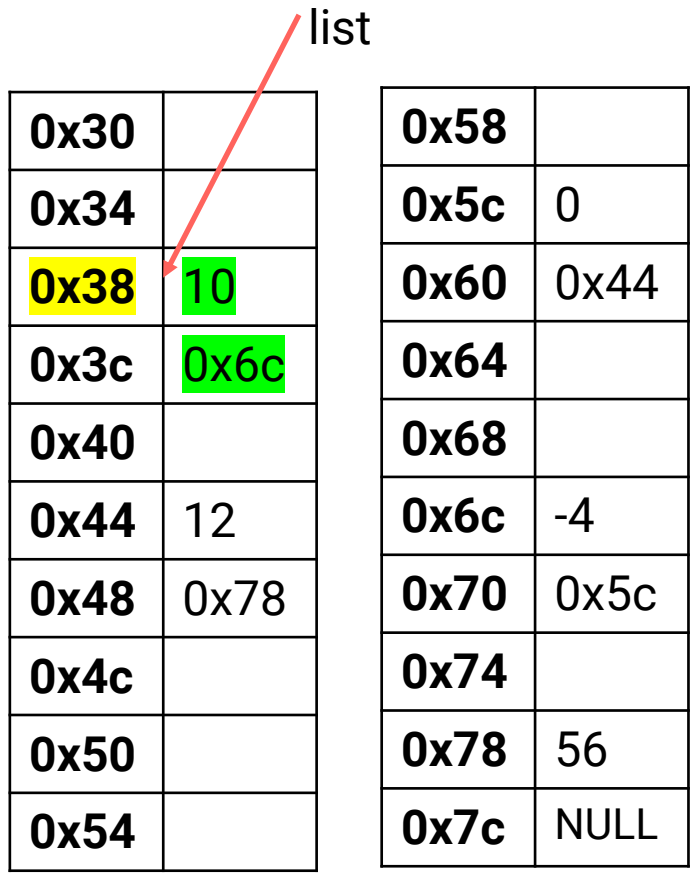
0x58	
0x5c	0
0x60	0x44
0x64	
0x68	
0x6c	-4
0x70	0x5c
0x74	
0x78	56
0x7c	NULL

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.

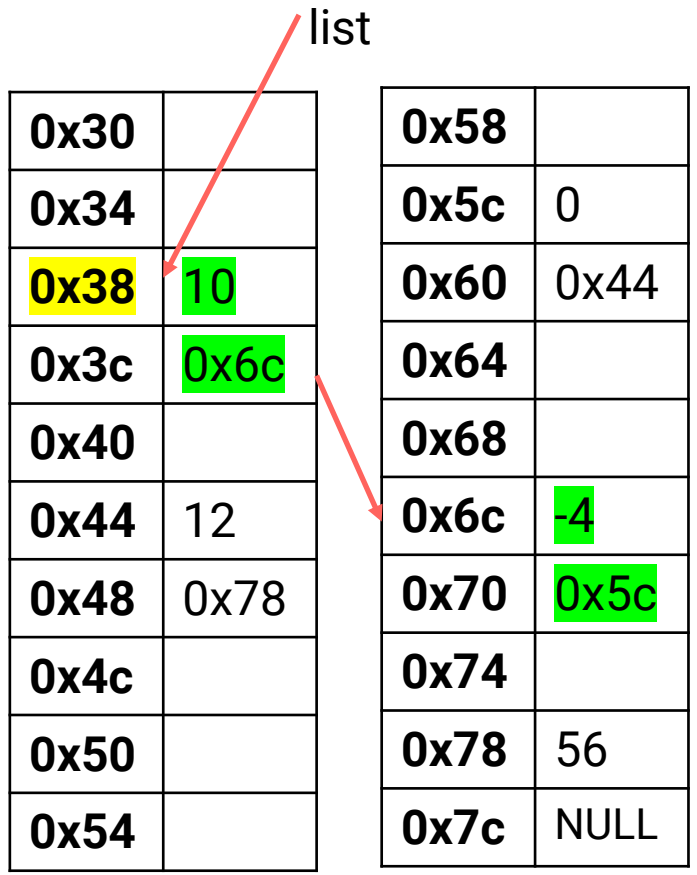


Linked Lists in the Memory

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.

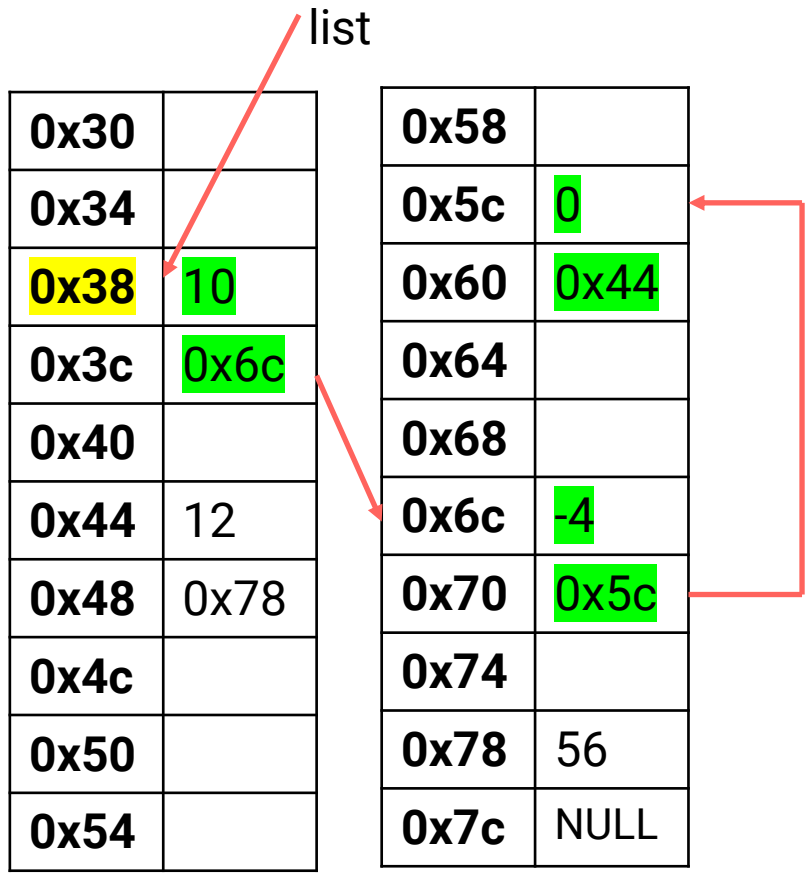


Linked Lists in the Memory

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.

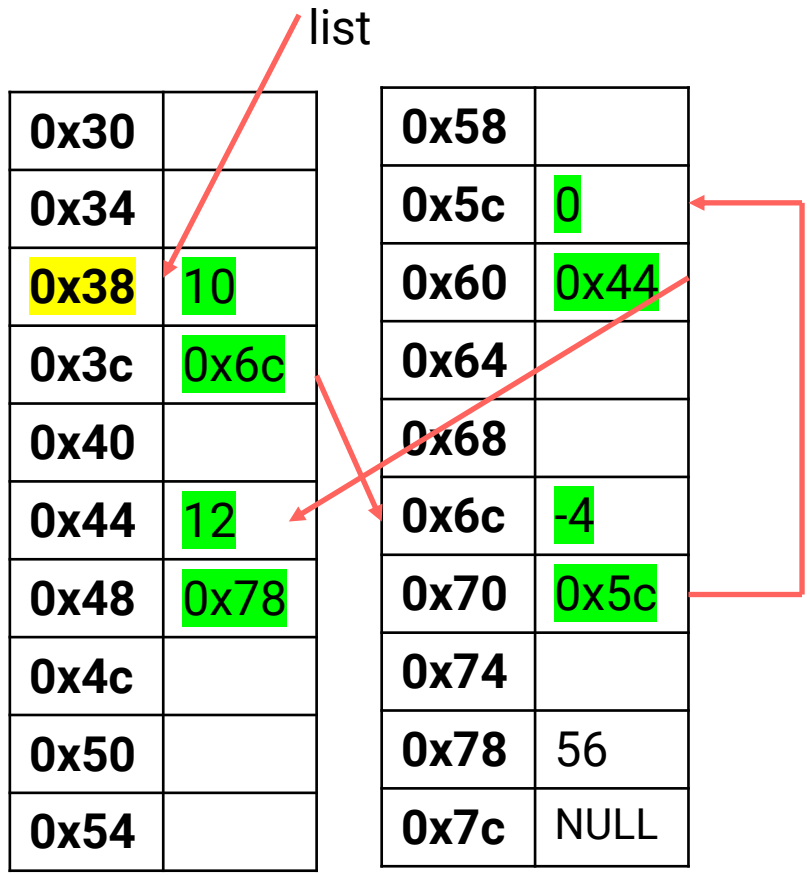


Linked Lists in the Memory

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.

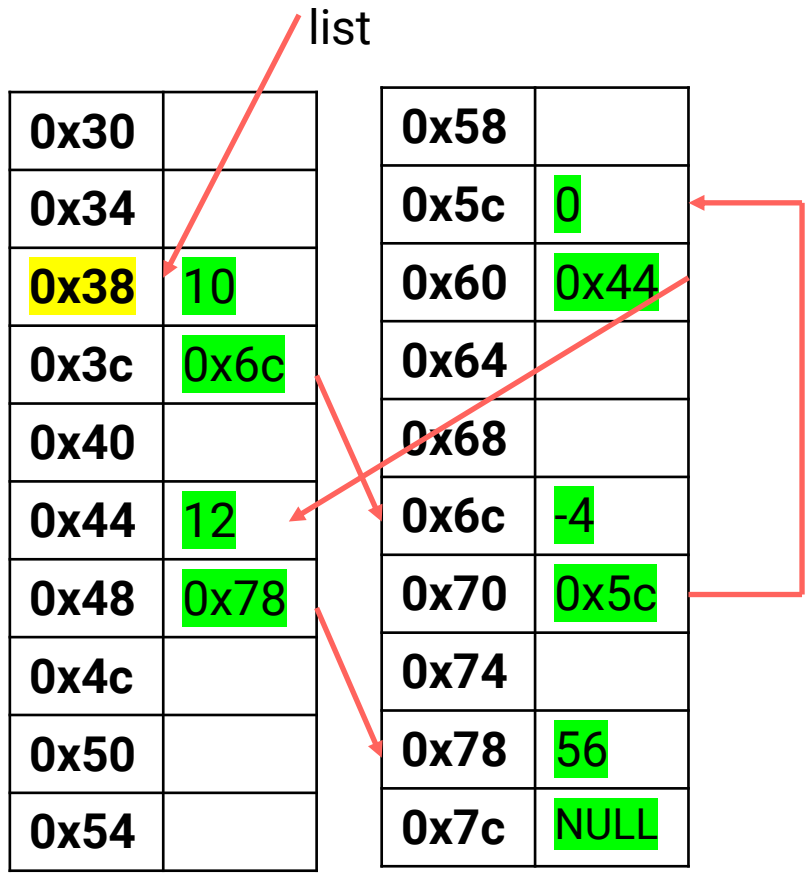
To access the full list, you follow each pointer **step by step**, like moving from one train carriage to the next in a connected train.



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.

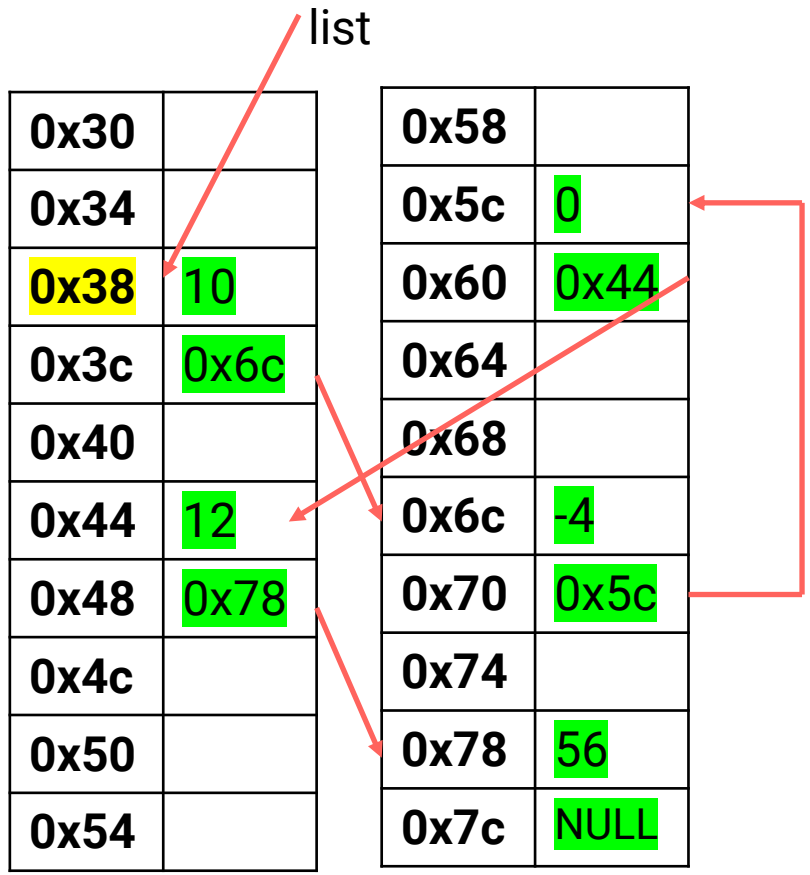
At this point, you have successfully traversed the entire linked list.



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.

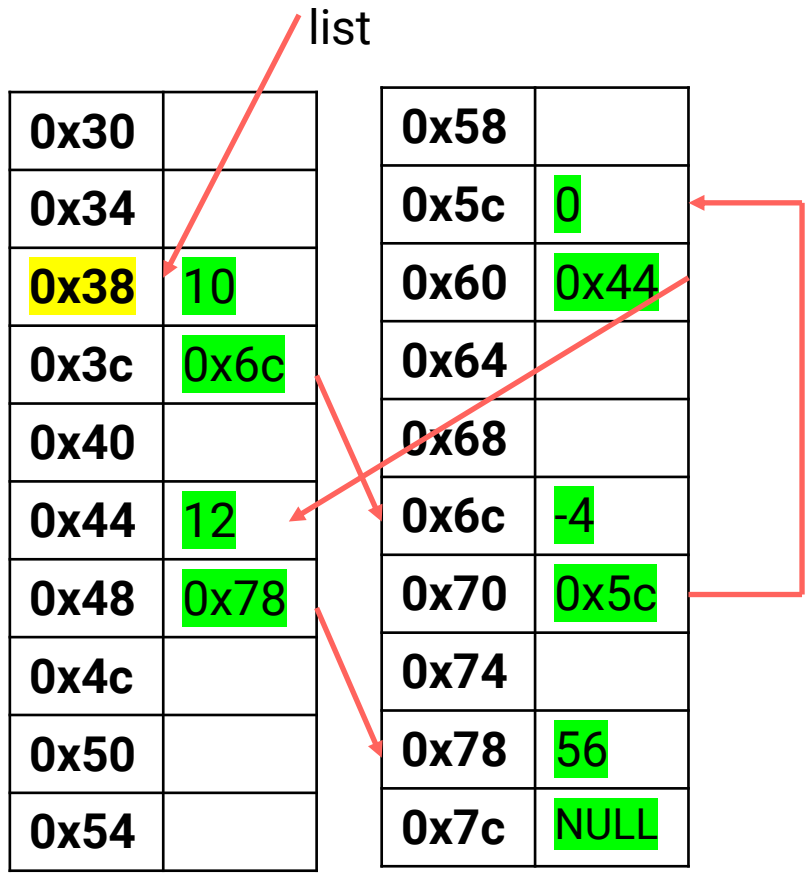
Unlike arrays, we cannot directly access a specific element using an index; instead, we must traverse the list to reach it.



Nodes

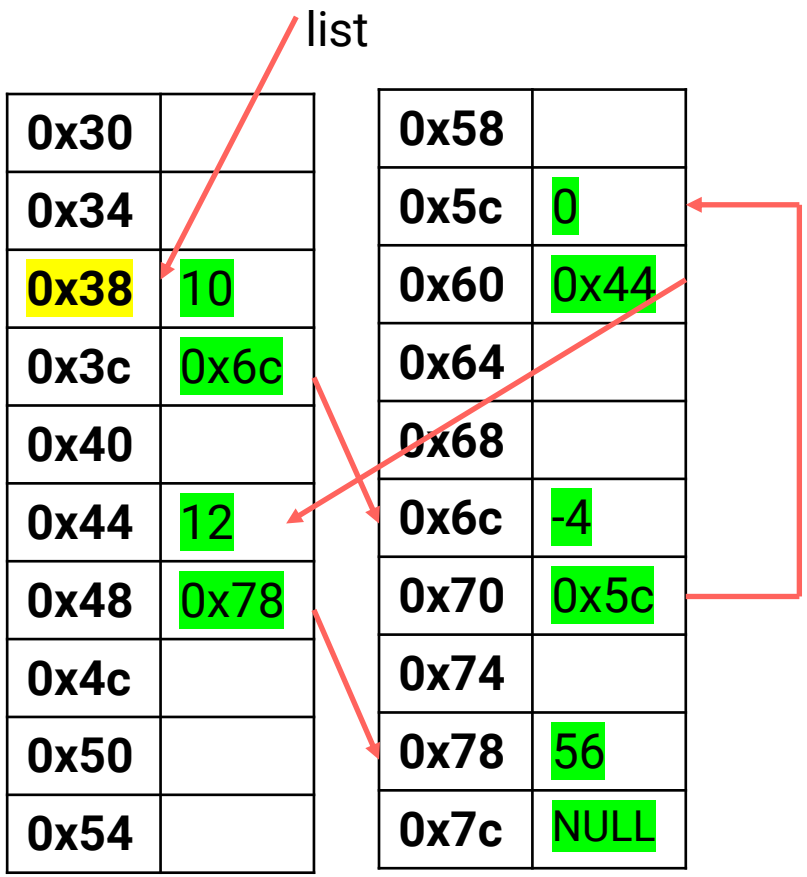
In C, we can define a structure (**struct**) that allows us to **store both**:

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



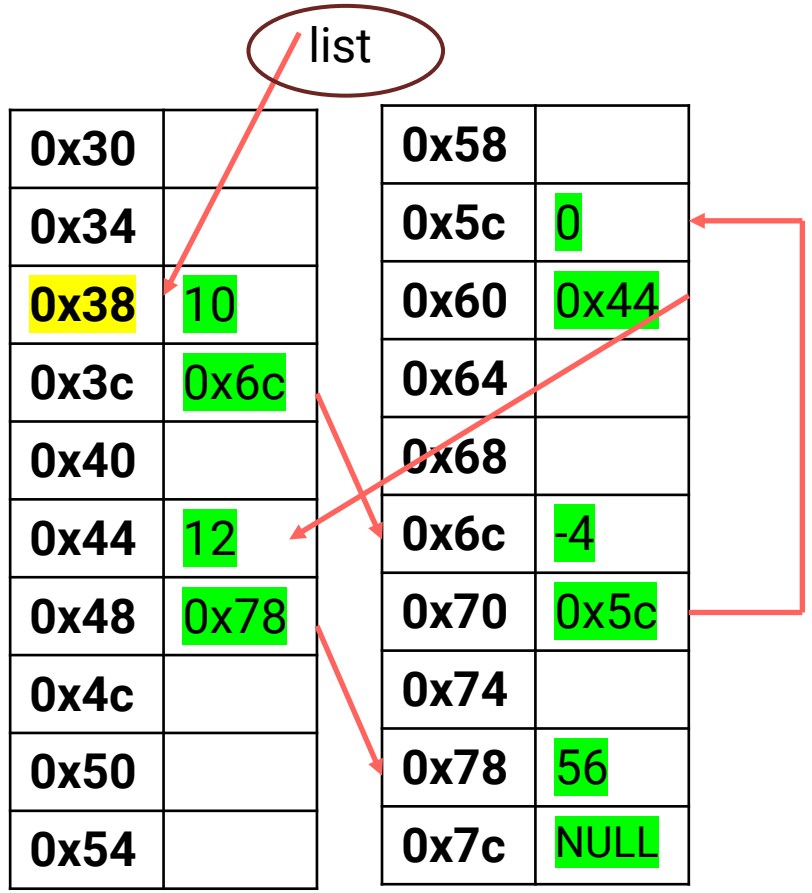
Nodes

```
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.

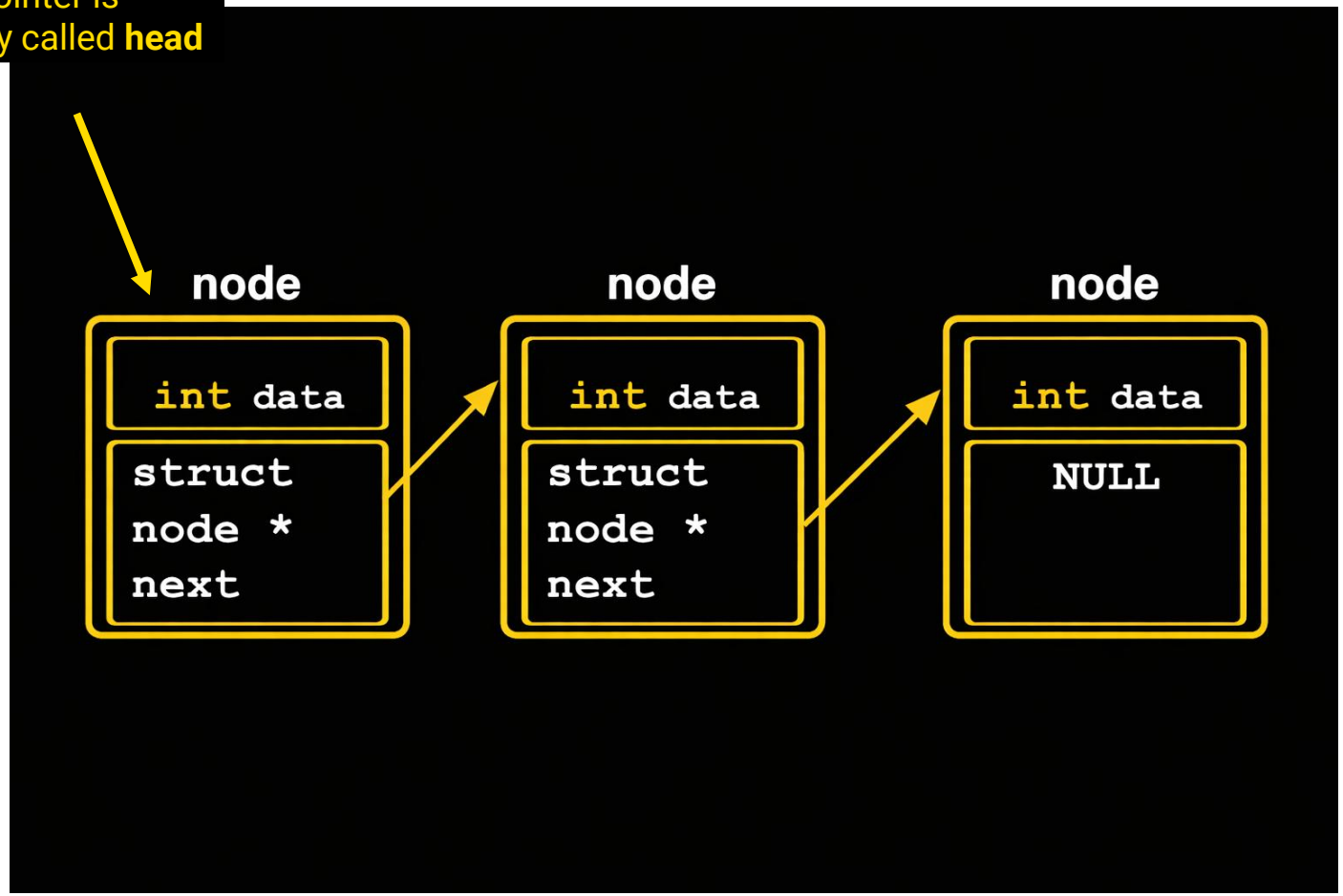
In this example, the **data** is a single **int**, but it can be **any data type** depending on your needs.

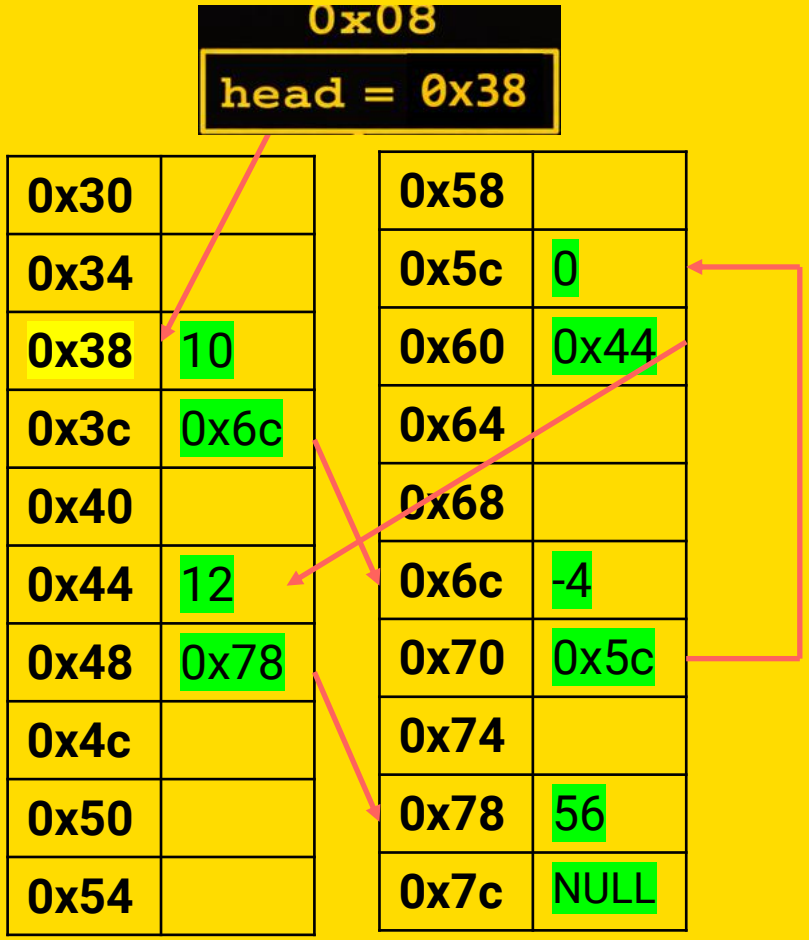
Later, we will explore linked lists that store **different types** of data.

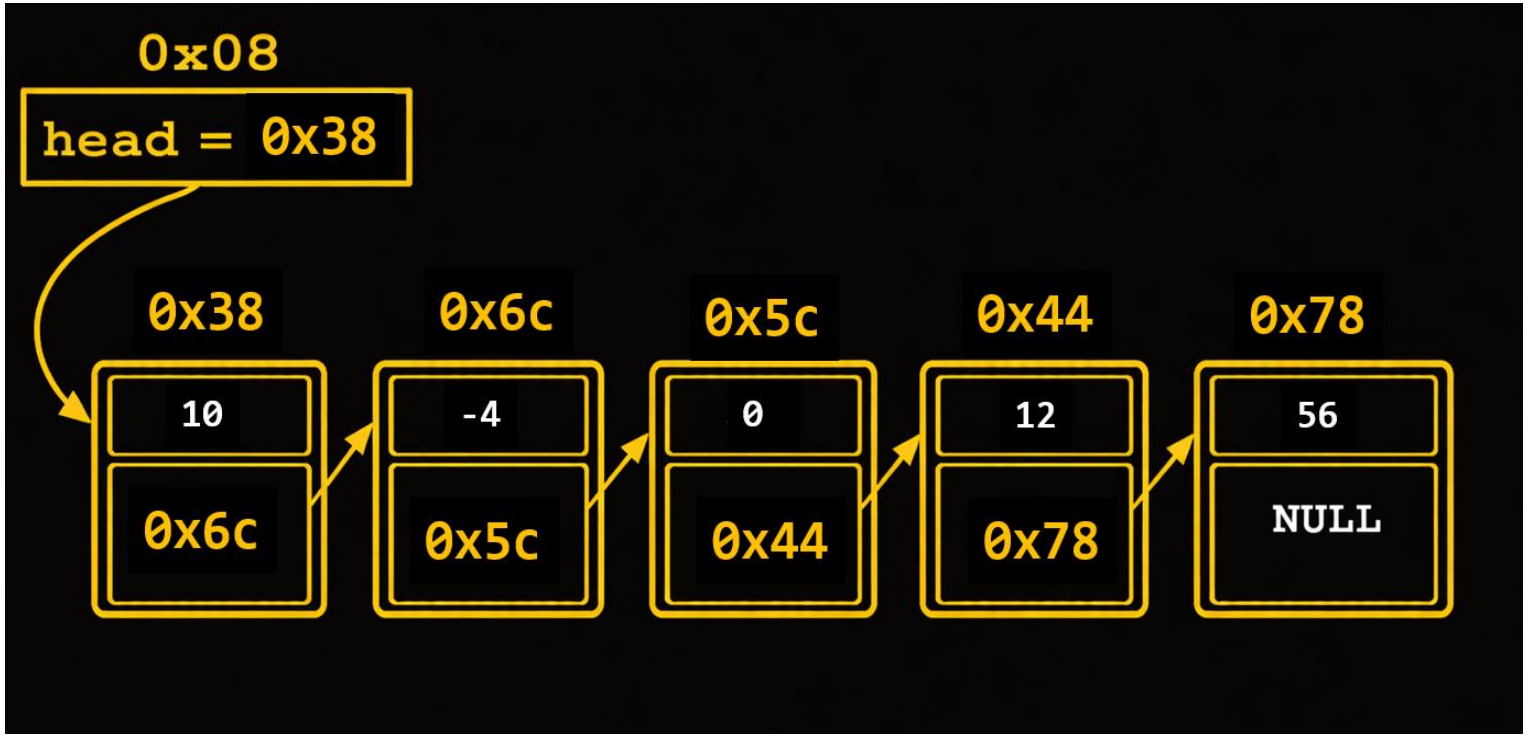
Each node also includes a **pointer to the next node** in the list (which is of the **same type**).

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

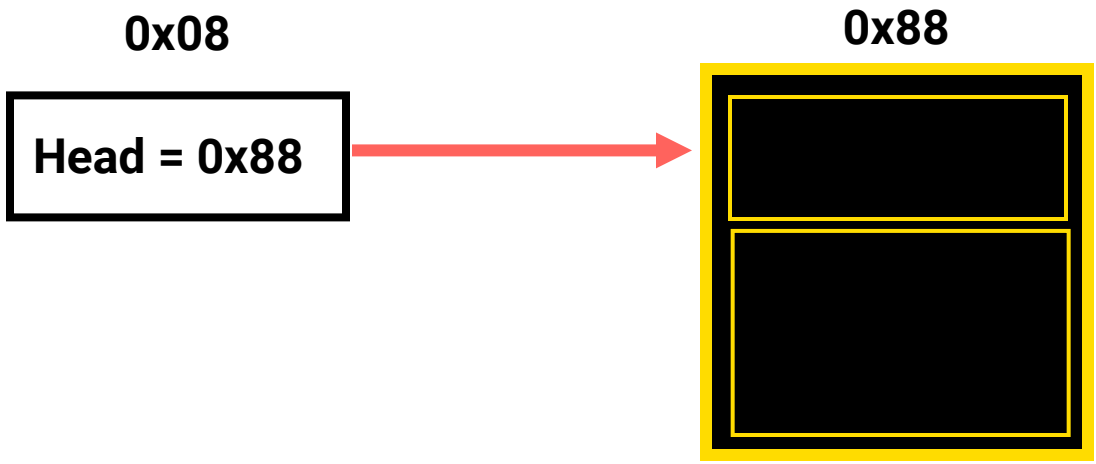
```c
struct node {
        int data;
        struct node *next;
};

struct node *head = NULL;
```

0x08

Head = NULL

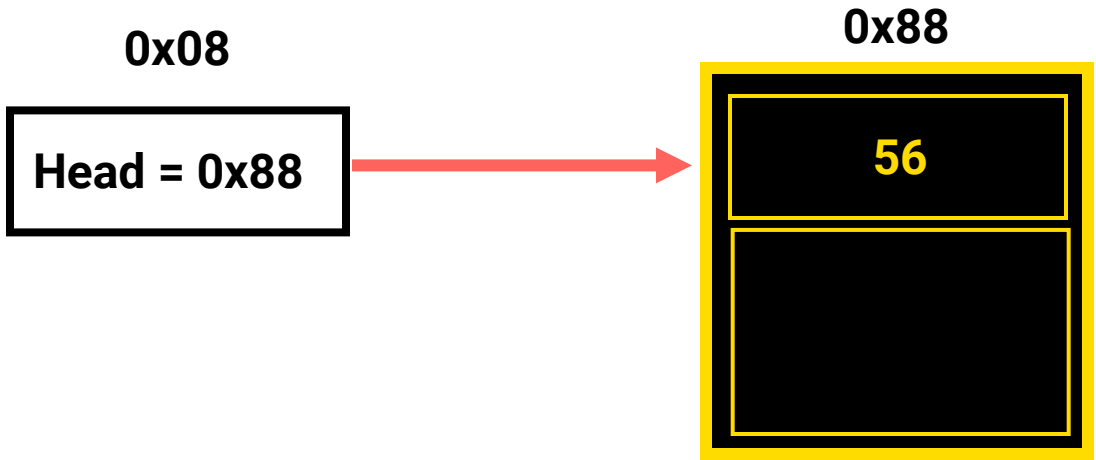
Creating a List with 1 Node

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

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



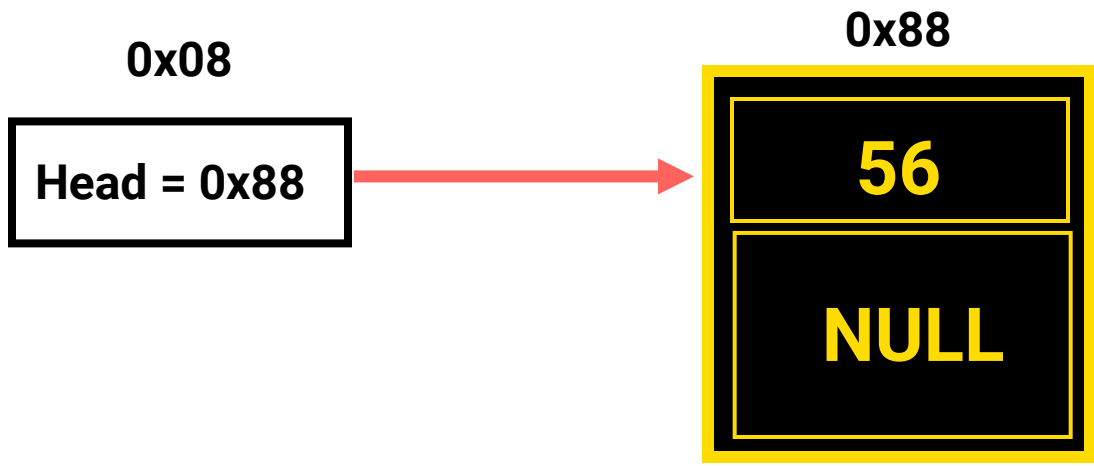
Creating a List with 1 Node

```
struct node {

        int data;

        struct node *next;
};

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



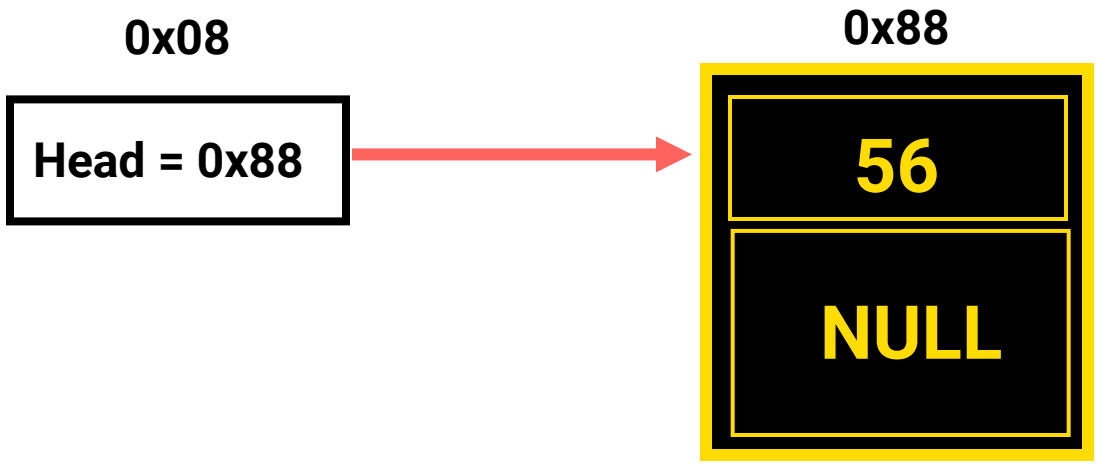
Creating a List with 1 Node

```
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

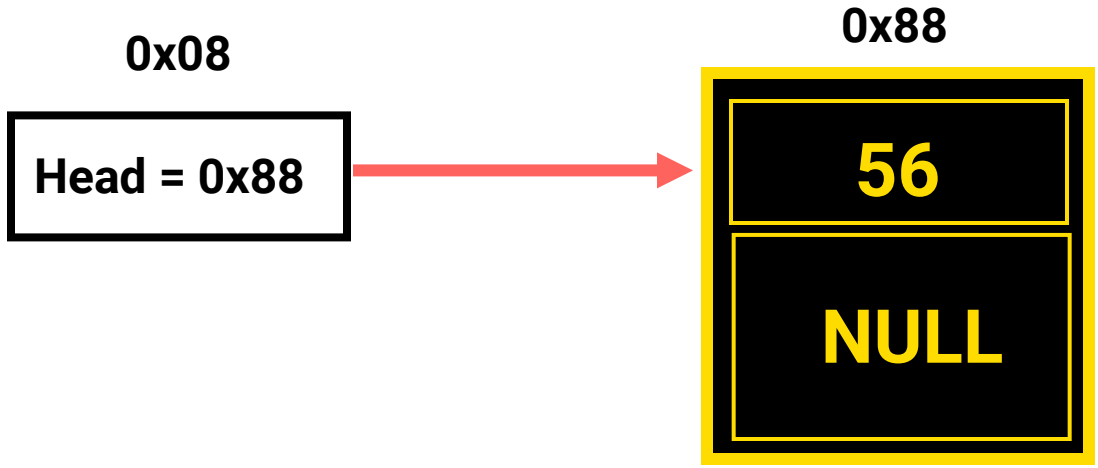


We will create a new node and attach it to the **end of the existing list**.

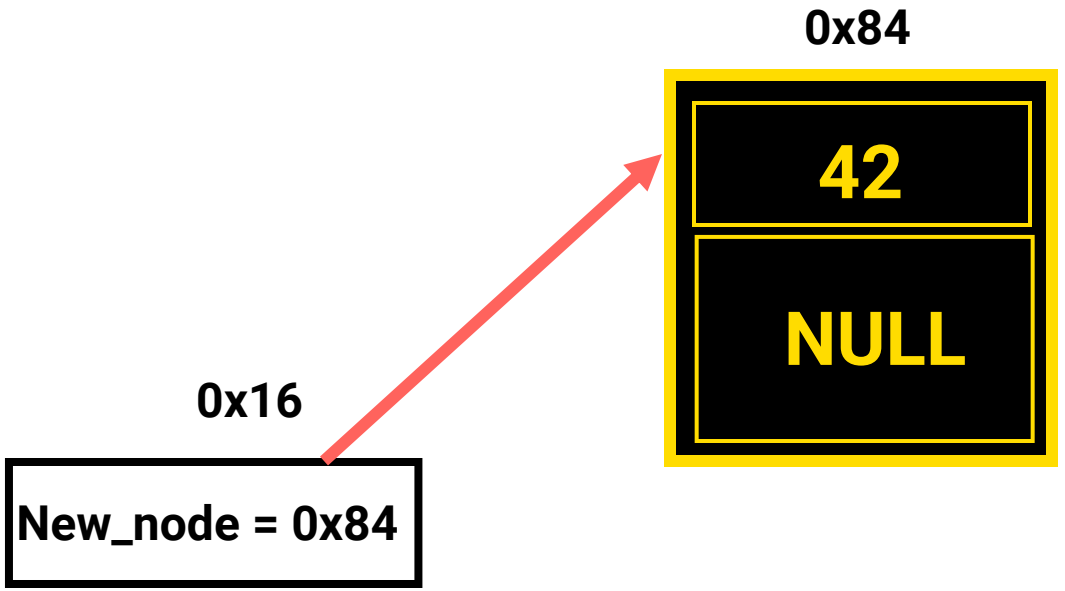
The **last node** in a linked list is commonly referred to as the **tail**.

Adding a new node to the end of our list

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

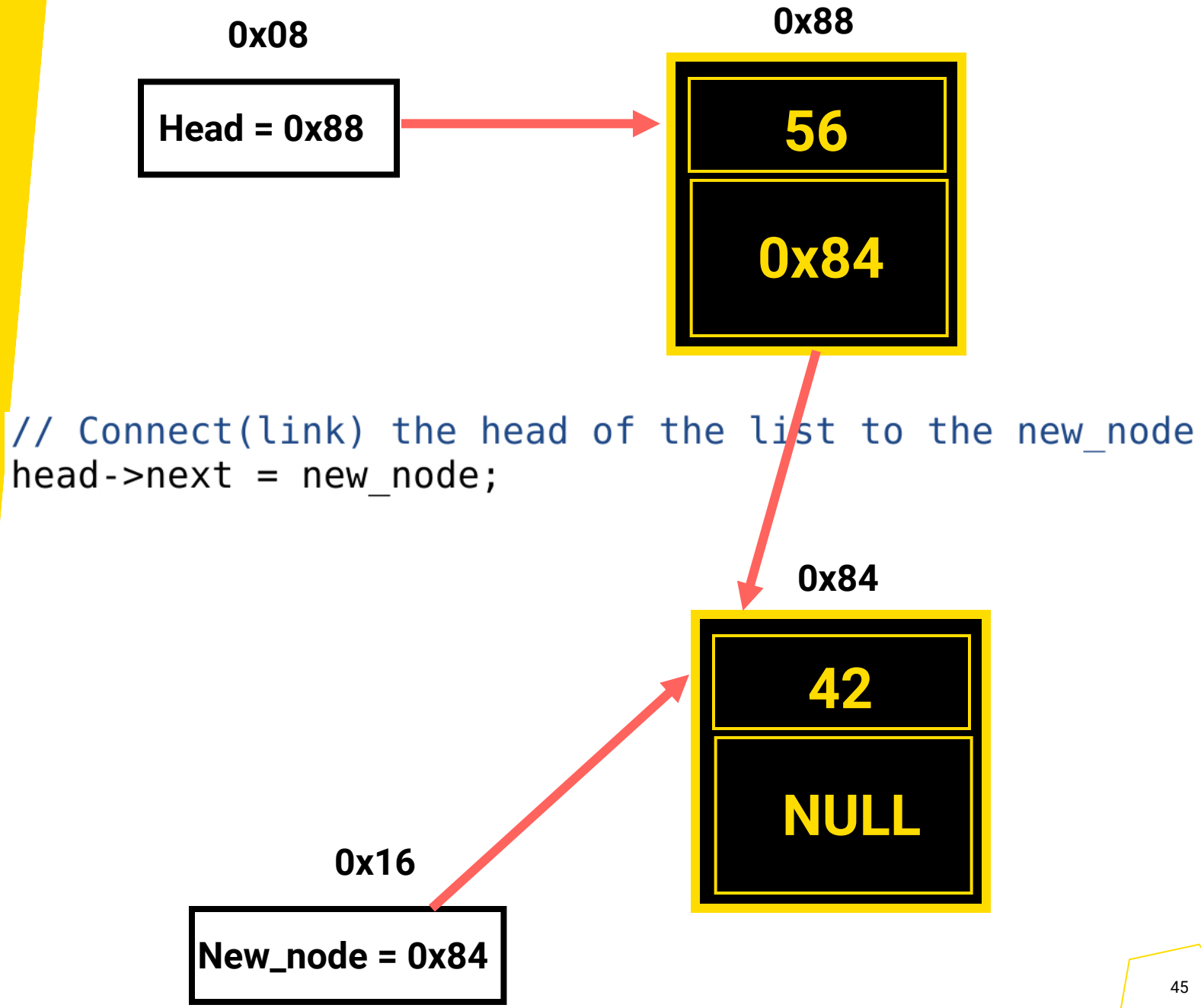



```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 42;
new_node->next = NULL;
```

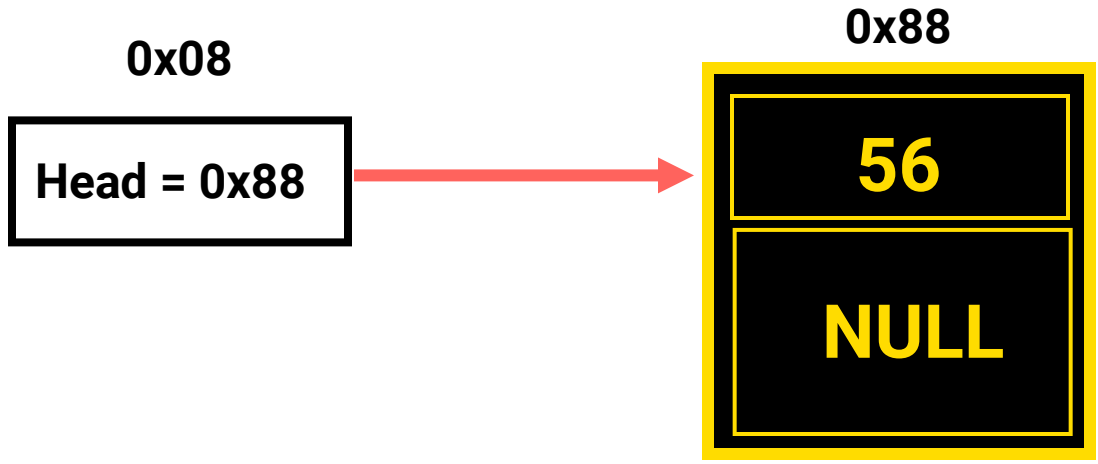


Adding a new node to the end of our list

are they connected now? (YES!!)



Adding a new node to the start of our list

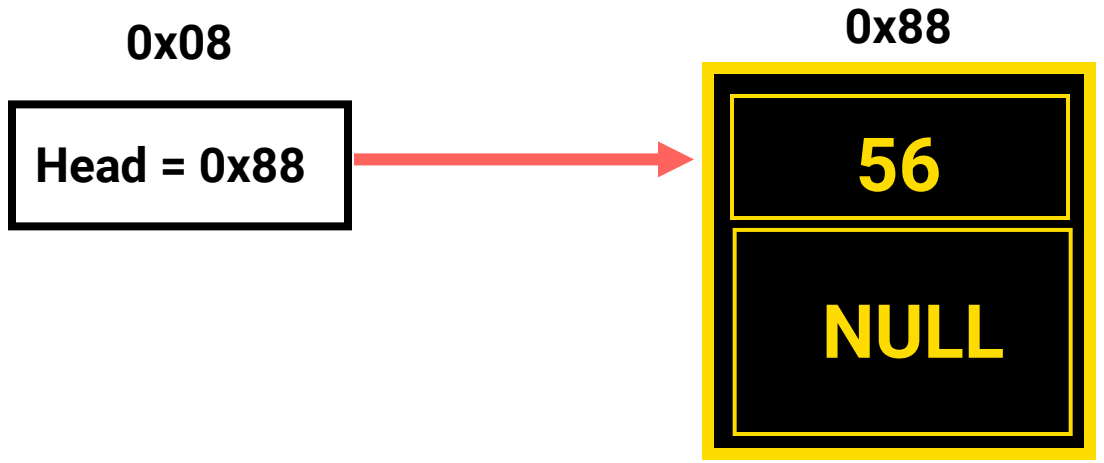


We will create a new node and **insert it at the beginning** of the list.

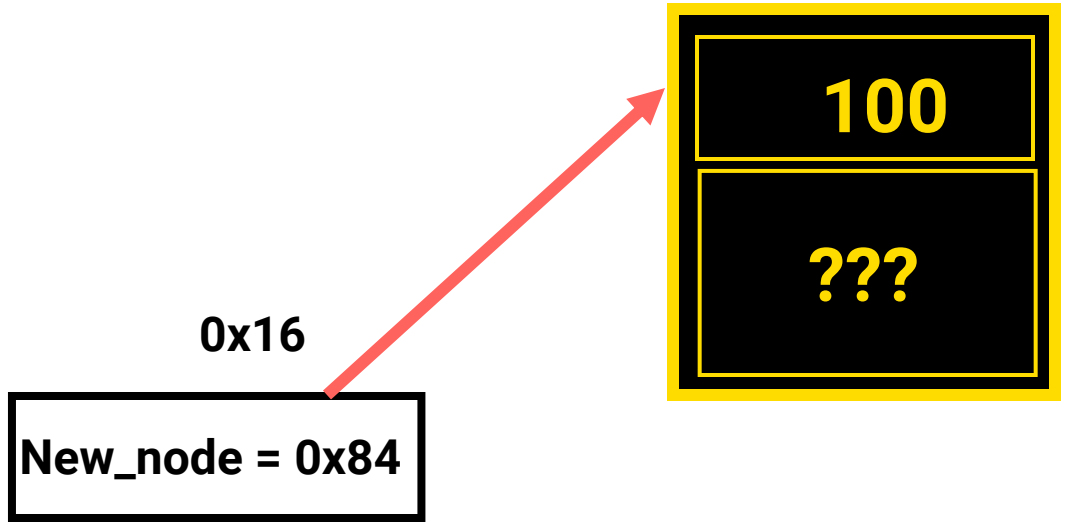
The **first node** in a linked list is commonly referred to as the **head**.

Adding a new node to the start of our list

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



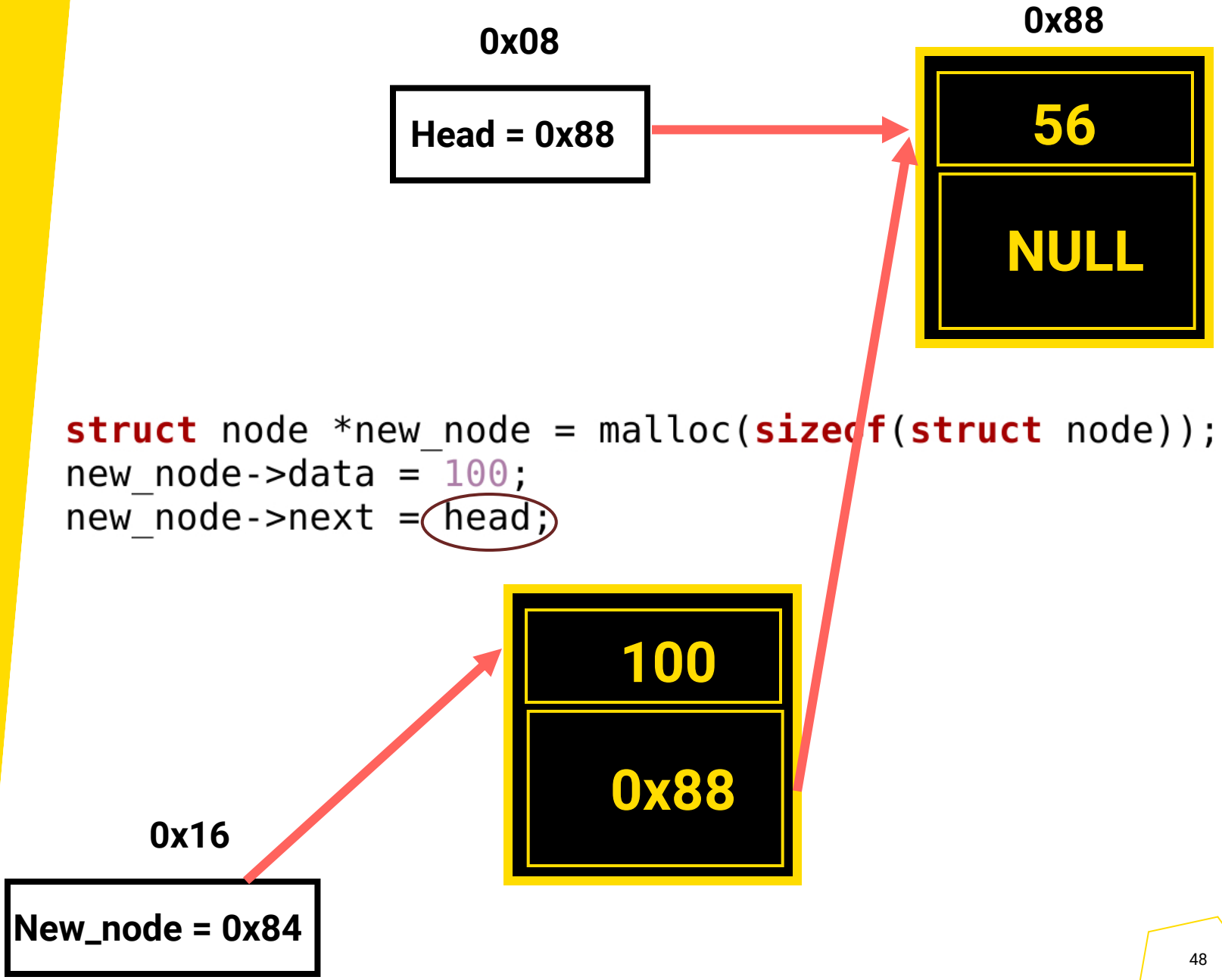

```
struct node *new_node = malloc(sizeof(struct node));
new_node->data = 100;
new_node->next = ???;
```



Adding a new node to the start of our list

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

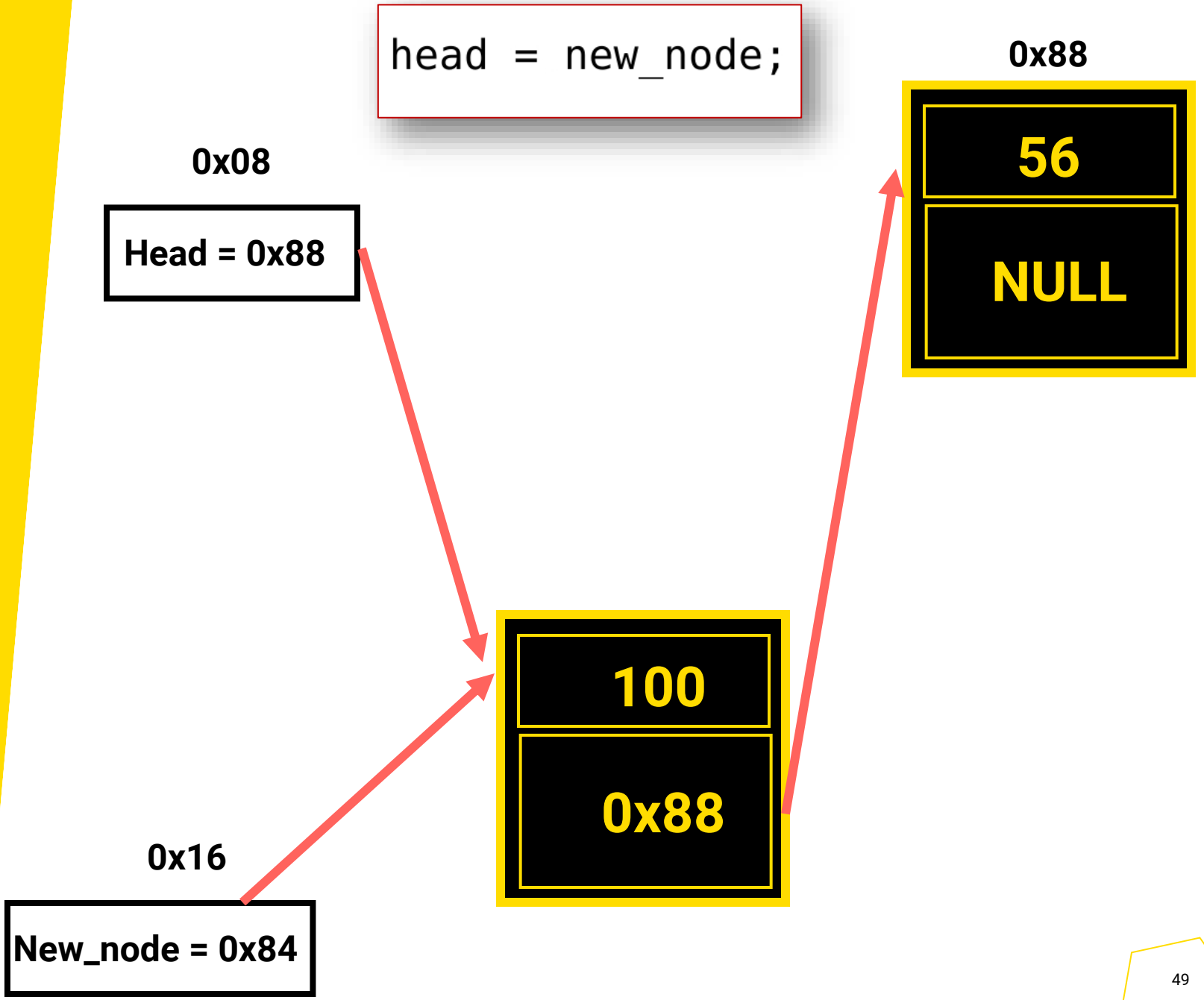
Is the head pointing to the correct node? (not yet)



Adding a new node to the start of our list

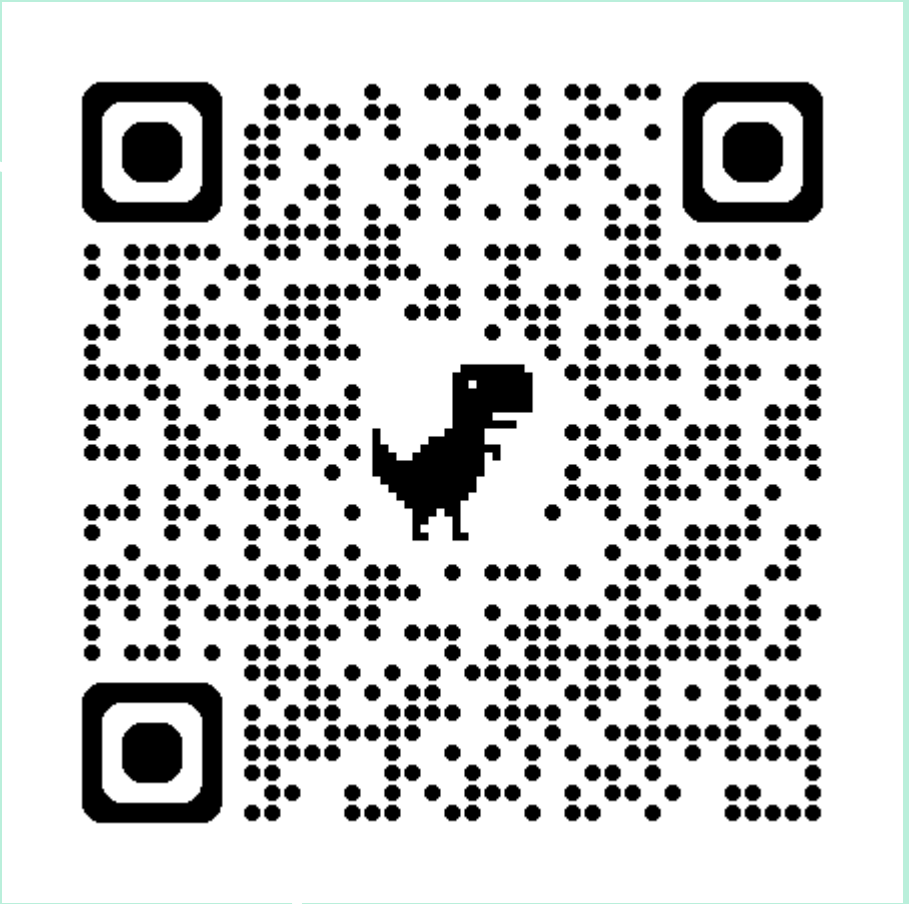
is the new node connected to the head? YES!!!

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



Demo

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

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