LECTURE 11 & 12

COMP(1511|1911) 26T2

SOFIA DE BELLIS

MULTI-FILE PROJECTS

INTRODUCTION TO LINKED LISTS

INSERTING INTO LINKED LISTS

TODAY

- Multi-file Projects
- Multi-File Demo Program
- Memory Revision Kahoot
- Introduction to Linked Lists

Week 8
“Mini Exam”

https://www.tickettailor.com/events/comp1511unsw/2308753

Week 8
Revision Sessions

https://www.tickettailor.com/events/comp1511unsw/2306134

MULTI-FILE PROJECTS

WHAT ARE THEY?

- Big programs are often spread out over multiple files. There are a number of benefits to this:

1. Improves readability (reduces length of program).
2. You can separate code by subject (modularity).
3. Modules can be written and tested separately.

- So far we have already been using the multi-file capability. Every time we #include, we are actually borrowing code from other files.
- We have been only including C standard libraries.

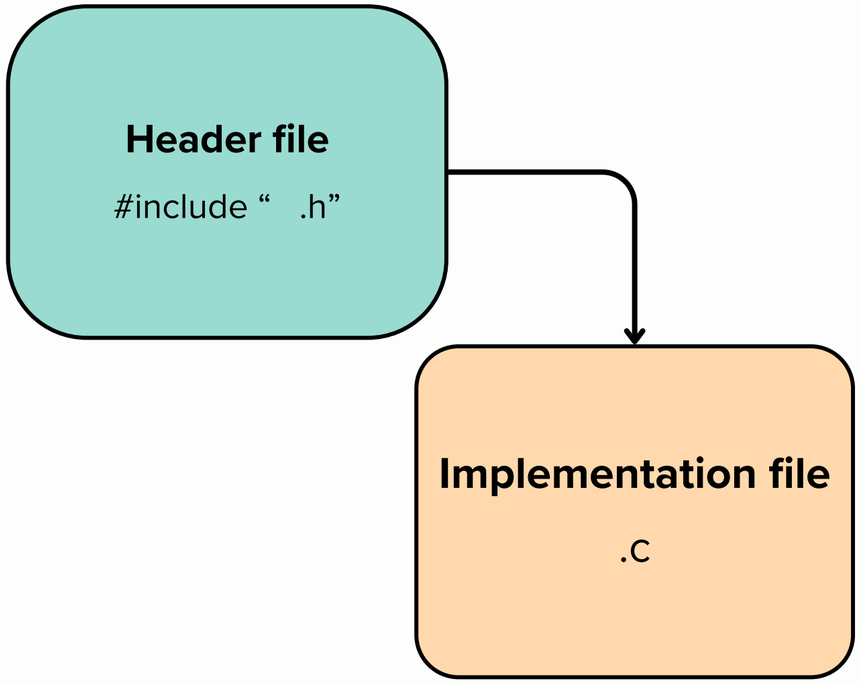
MULTI-FILE PROJECTS

WHAT ARE THEY?

- You can also #include your own libraries!
- This allows us to join projects together.
- It also allows multiple people to work together on projects out in the real world.
- We will also often produce code that we can then use again in other projects
    - This is all that the C standard libraries that we have been using are - functions that are useful in multiple instances.

MULTI-FILE PROJECTS

.H AND .C



- In a multi file project we might have:
  - (multiple) header file - this is the .h file that you have been using from standard libraries already
  - (multiple) implementation file - this is a .c file, it implements what is in the header file.
- Each header file that you write, will have its own implementation file
- A main.c file
  - this is the entry to our program
  - contains the main function
  - we try and have as little code here as possible

MULTI-FILE PROJECTS

THE HEADER FILE

Header file

#include “ .h”

- Typically contains:
  - function prototypes for all functions that will be implemented in the implementation file
  - comments that describe how the functions will be used
  - #defines
  - enum definitions
  - the file bascially SHOWS the programmer all they need to know to use the code
  - NO RUNNING CODE
  - This is like a definition file

MULTI-FILE PROJECTS

THE IMPLEMENTATION FILE

Implementation file

.c

This is where you implement the functions that you have defined in your header file.

MAIN.C

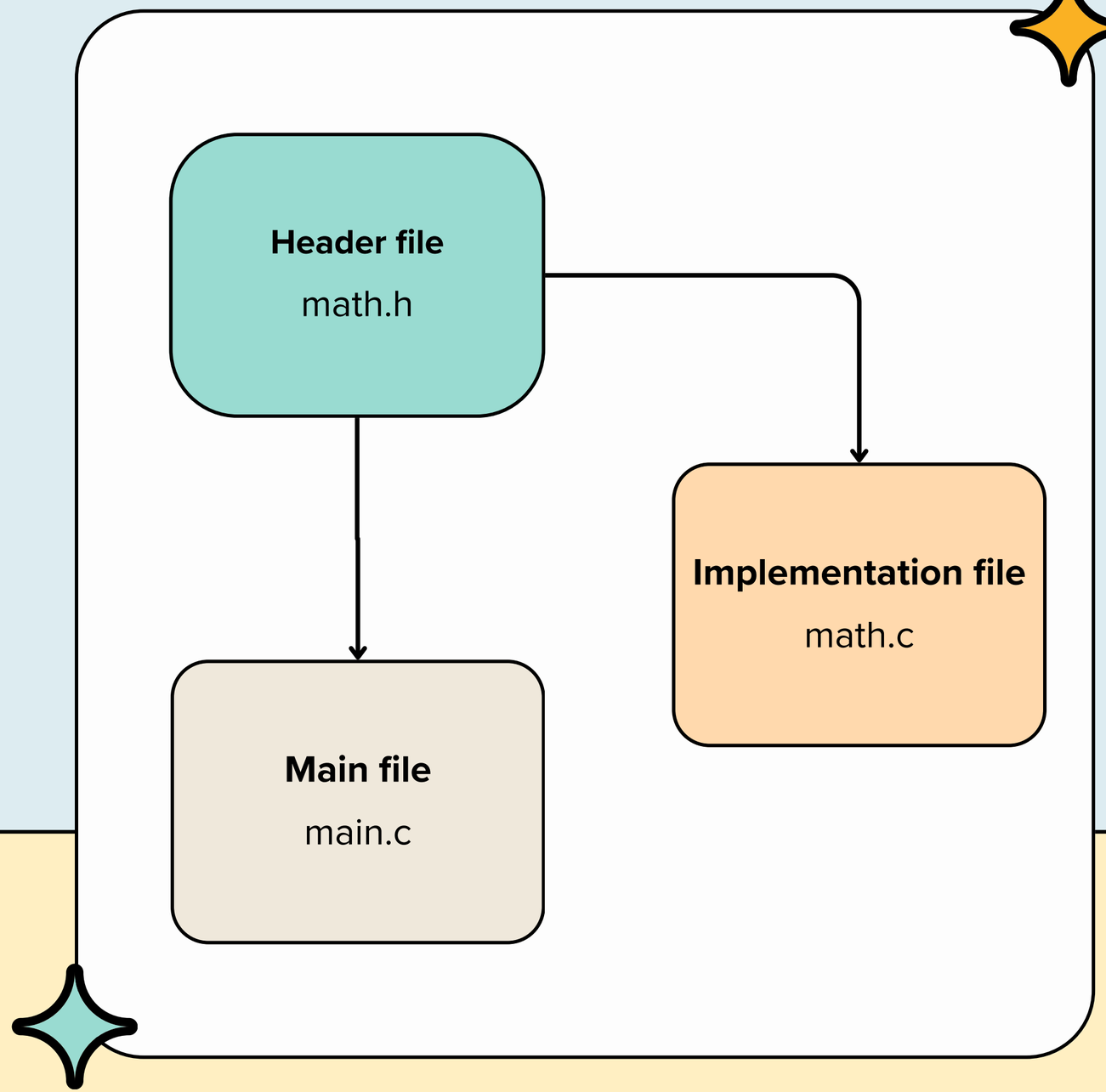
The entry point for you program, where you call functions from that may exist in other modules.

MULTI-FILE PROJECTS

CODE DEMO

We will have three files:

- Header file - math.h
- Implementation file - math.c
  - #include “math.h”
- Main file - main.c
  - #include “math.h”



MULTI-FILE PROJECTS

COMPILING

To compile a multi file project, you list any .c files you have in your project. In the case of our example, we have maths.c and main.c files.

```
sofiad@vx13 ~ dcc maths.c main.c -o maths
sofiad@vx13 ~ ./maths
The area of the rectangle is 13.950000
The sum of the two numbers is 12
sofiad@vx13 ~
```

The program will always enter the main.c, so there should only be one main.c when compiling.

MEMORY RECAP

REVISITING MEMORY

A helper function cannot return a pointer of a stack variable! So how can we deal with this?

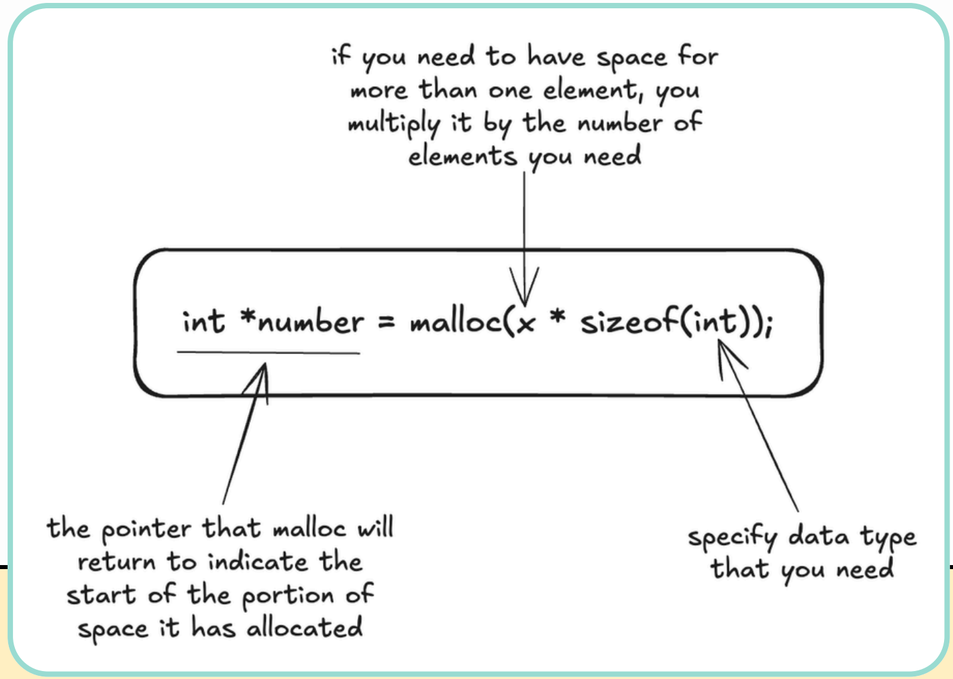
You can return by copying it or putting it into a more permanent storage - yay the heap!

Unlike stack memory, heap memory is allocated by the programmer and won't be deallocated until it is explicitly freed by the programmer also! You have a great power now... but with great power comes great responsibility!

SAVING MEMORY

We do have the wonderful opportunity to allocate some memory by calling the `**malloc()**` function and letting this function know how many bytes of memory we want this is the stuff that goes on the heap!

- This function **returns a pointer** to the piece of memory we created based on the number of bytes we specified as the input to.
- This function also allows us to dynamically create memory as we need it!



RETURNING MEMORY

It would be very impolite to keep requesting memory to be made, without giving some back when we no longer need it. This piece of memory is ours to control and it is important to remember to kill it or you will eat up all the memory you computer has... slow down the machine, and often result in crashing... often called a memory leak...

- **A memory leak occurs when you have dynamically allocated memory (with `malloc()`) that you do not free.**
- As a result, memory is lost and can never be free causing a memory leak.
- You can free memory that you have created by using the function `**free()**`.

```
free(number);
```

KAHOOT

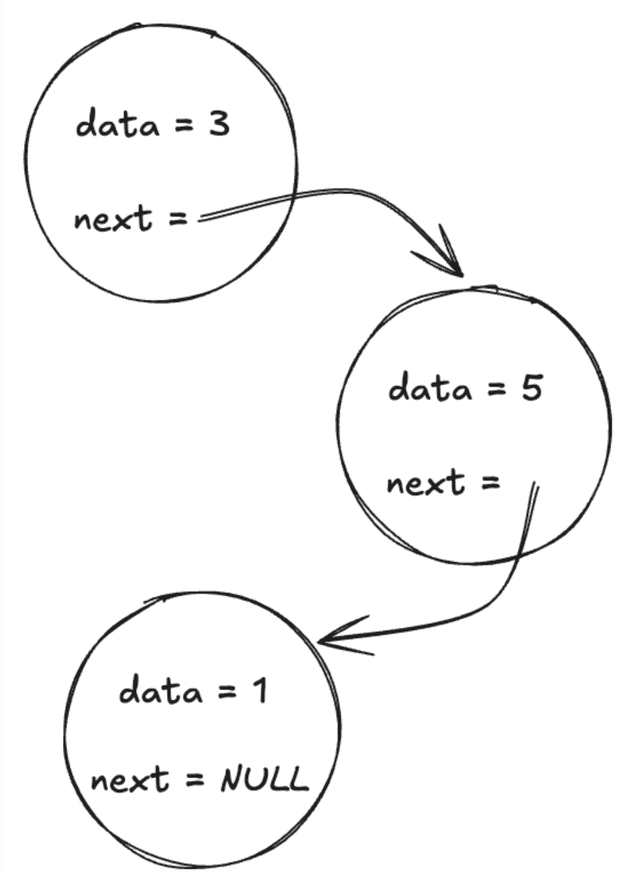
STRUCTS AND POINTERS

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

struct dog {
    int age;
    double weight;
    char *name;
};

int main(void) {
    // declare struct variable
    struct dog my_dog;

    // pointer to the struct
    struct dog *my_dog_ptr = &my_dog;


    // initialise the variable
    // deference, then access the member
    (*my_dog_ptr).age = 12;

    // alternative syntax
    my_dog_ptr->weight = 20;
    strcpy(my_dog_ptr->name, "jax");

    return 0;
}
```

- Remember that when we access members of a struct we use a .
- What happens if we make a pointer of type struct? How do we access it then?
- Those brackets can get quite confusing, so there is a shorthand way to do this with an ->
- There is no need to use (*my_dog_ptr) and instead can just straight my_dog_ptr->

LINKED LISTS

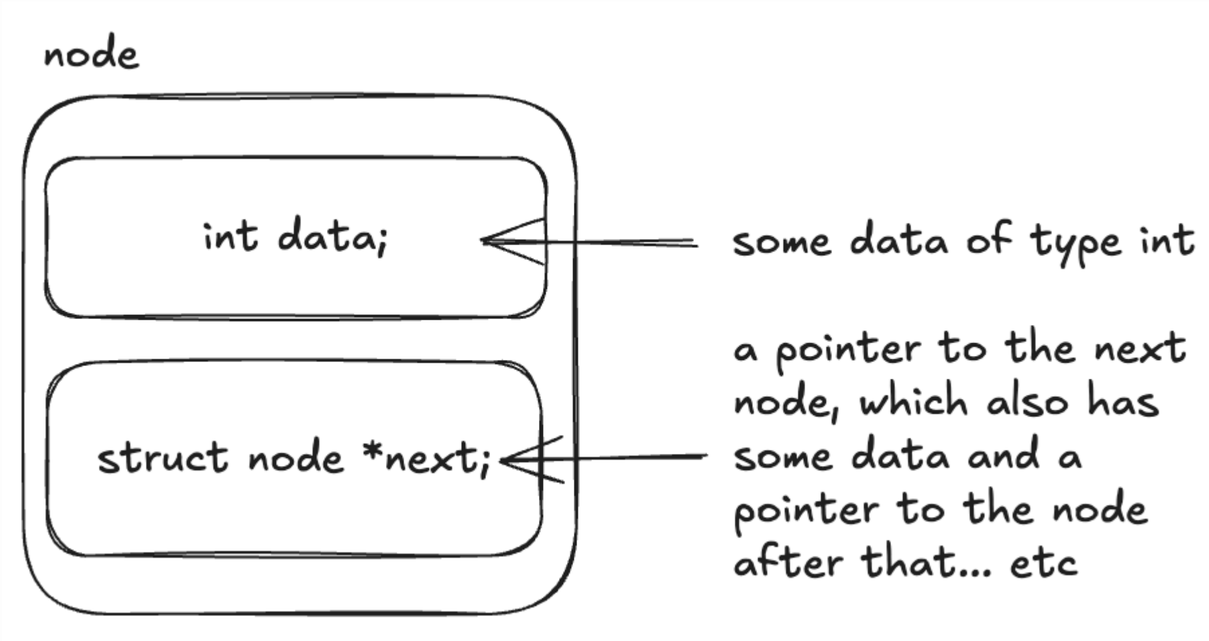
LINKED LISTS



- 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 “NODE”

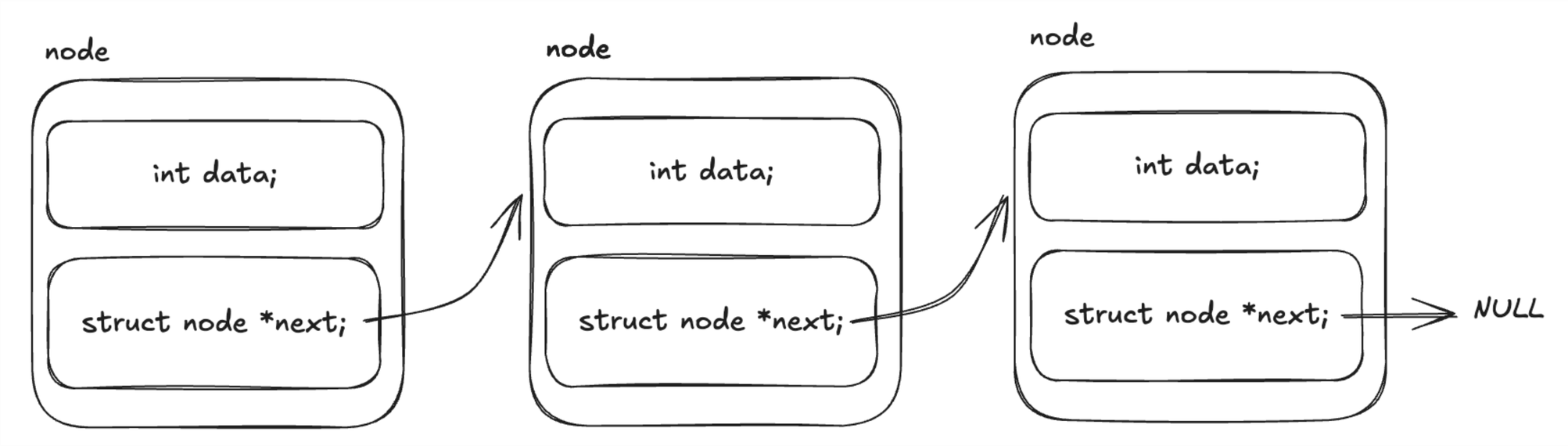
Each node has some data and a pointer to the next node (of the same data type), creating a linked structure that forms the list.




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

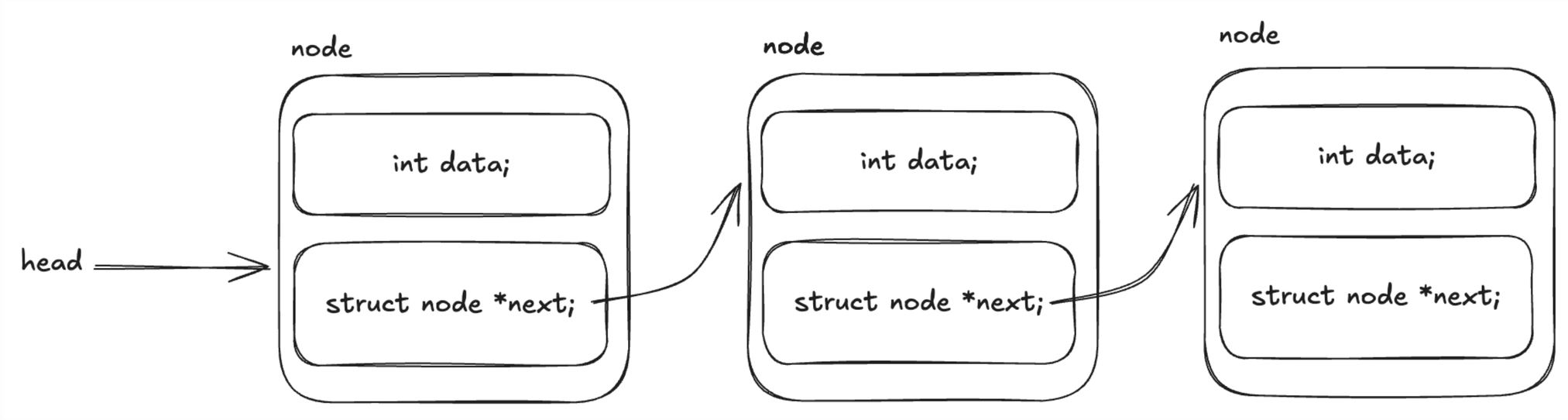
NODES AND LINKED LISTS

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.



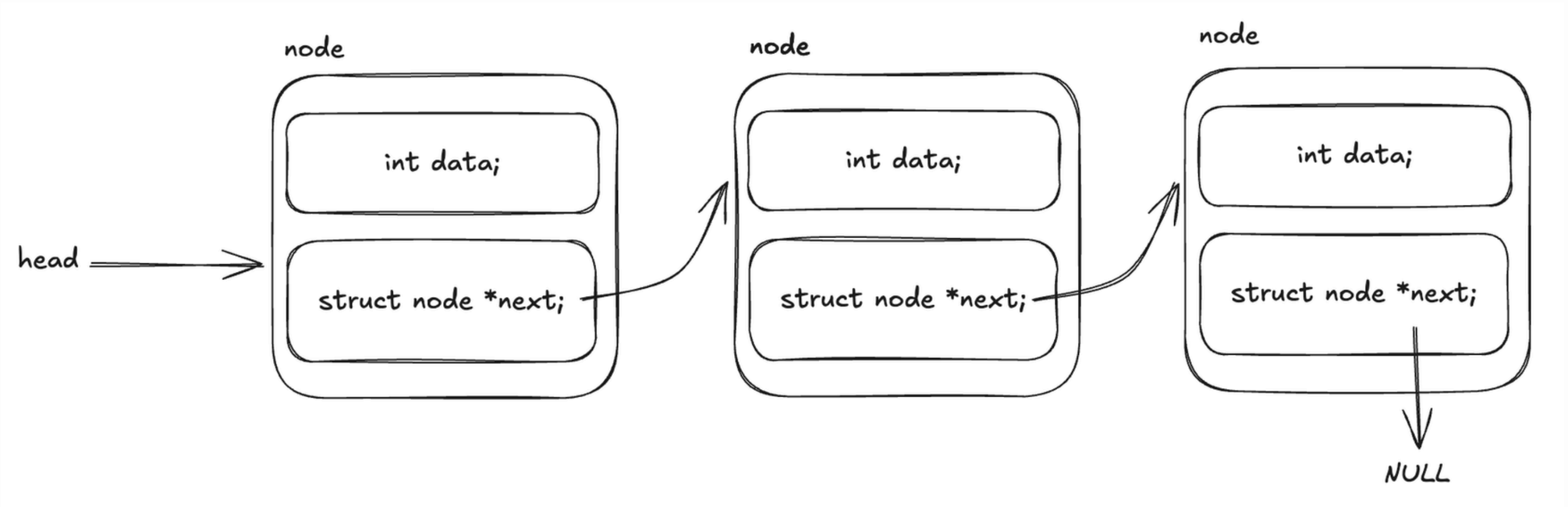
THE HEAD

To keep track of our list we keep a “head” pointer, which is a pointer to the first node in the list. The head pointer is not a node itself, but the memory address of where the first node is.

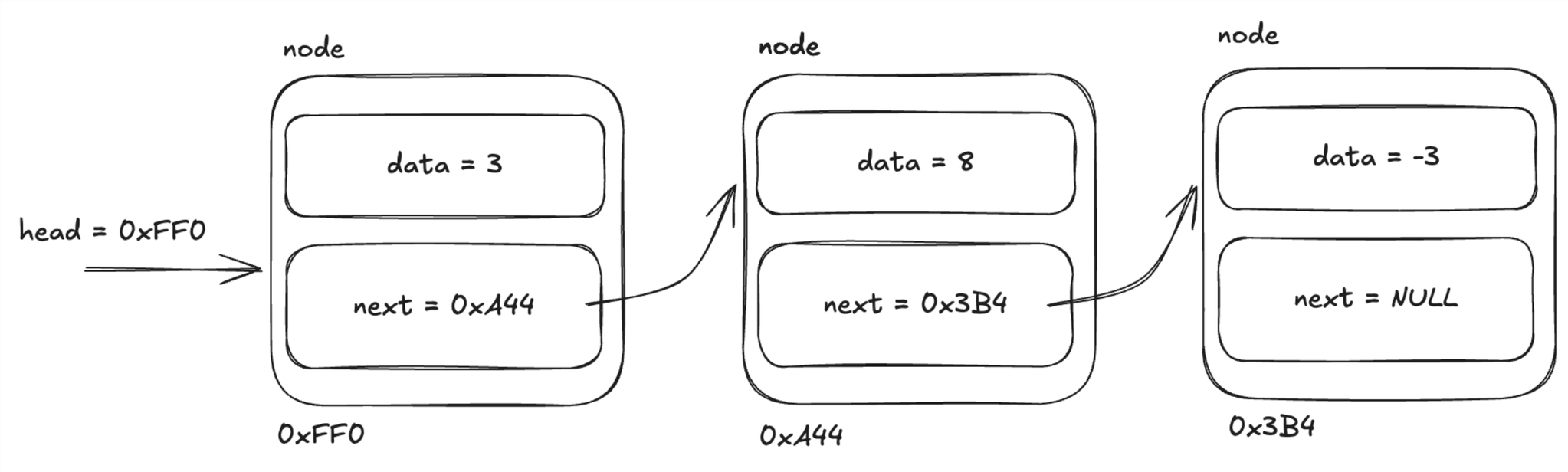


THE TAIL

The last node of a linked list is known as the “tail” node and it identified as the node thats next pointer, is pointing to NULL.



EXAMPLE LIST



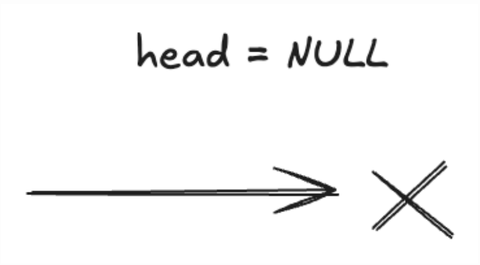
EXAMPLE LIST

In order to create a linked list, we would need to

1. Define struct for a node
2. A pointer to keep track of where the start of the list is
3. A way to create a node and then connect it into our list

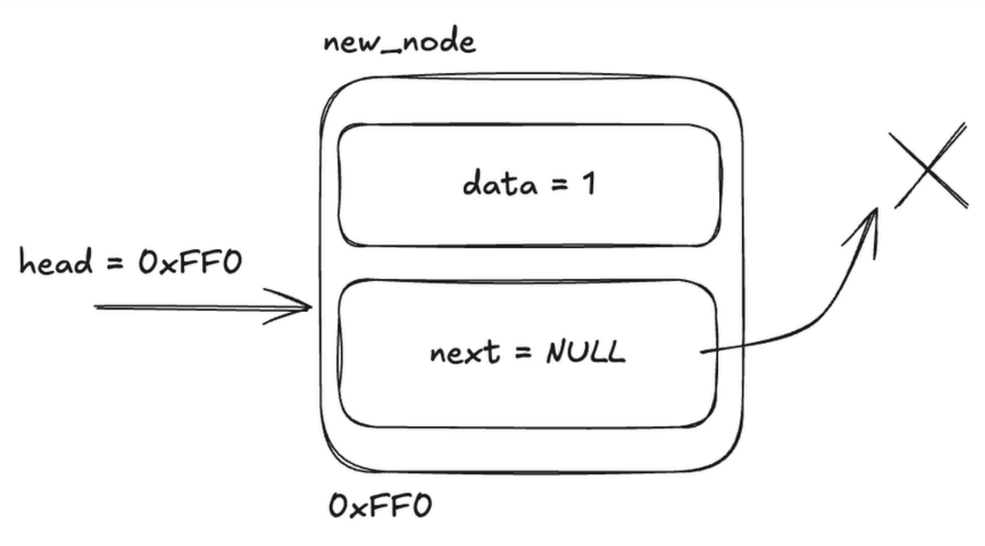
Let's say we wanted to create a linked list with 5, 3, 1

1. Let's create the first node to start the list.
2. 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.



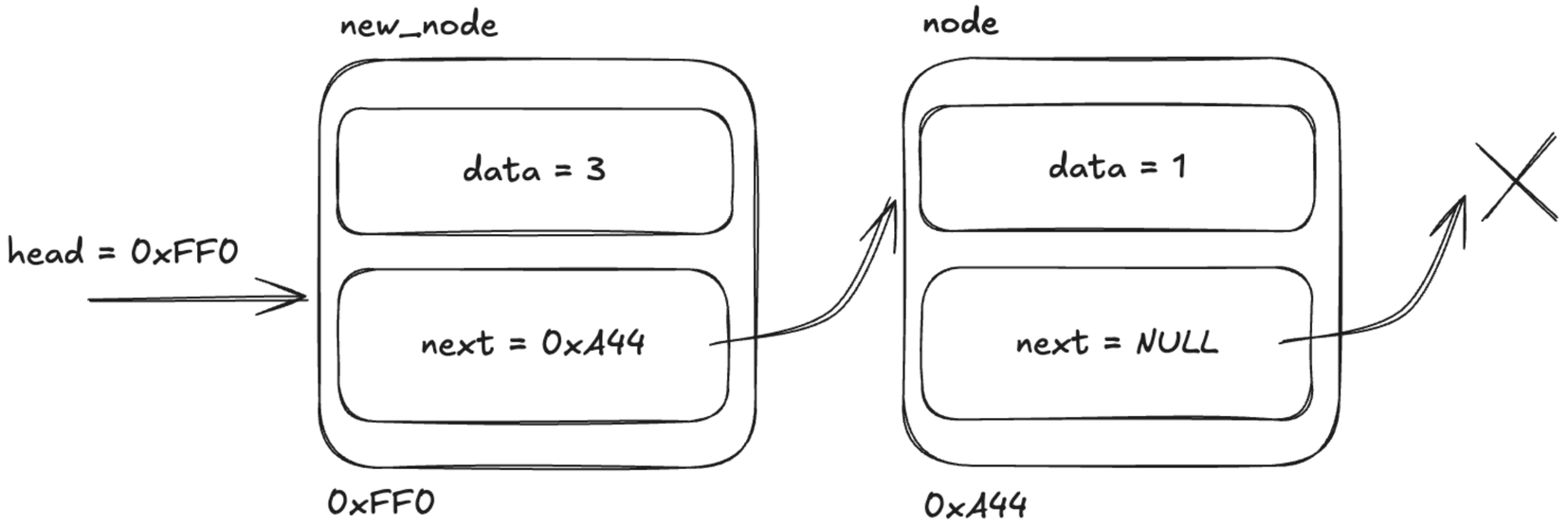
EXAMPLE LIST

1. Create the next node to store 1 into (you need memory).
2. Assign 1 to data.
3. Make its next pointer point to NULL.
4. Insert it at the beginning so the head would now point to this new node.



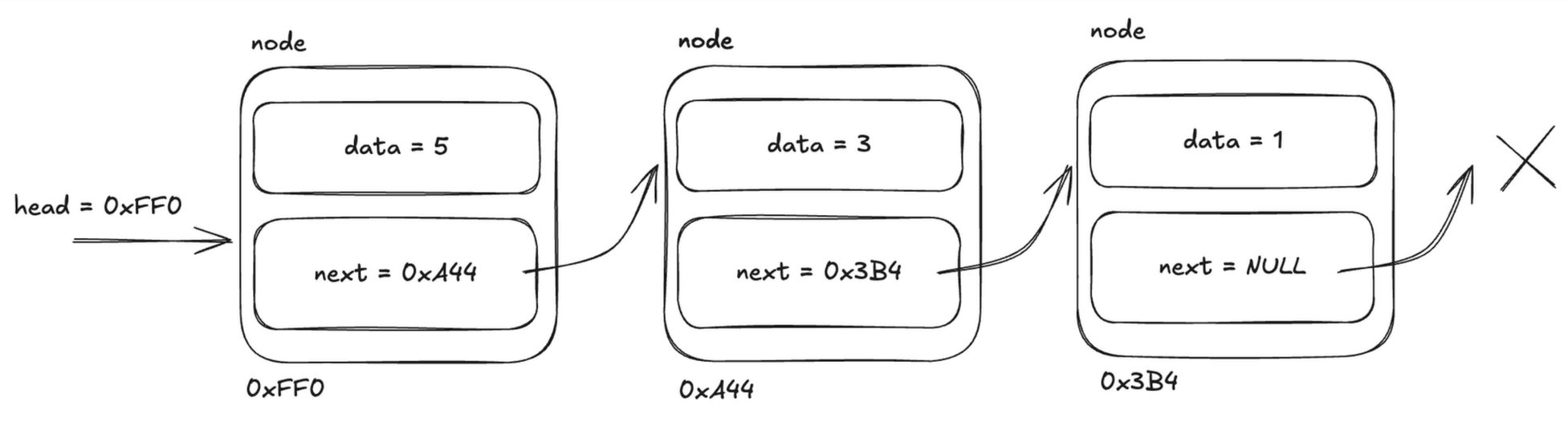
EXAMPLE LIST

1. Create the next node to store 3 into (you need memory)
2. Assign 3 to data.
3. Make its next pointer point to NULL.
4. Insert it at the beginning so the head would now point to it and the new node would point to the old head.



EXAMPLE LIST

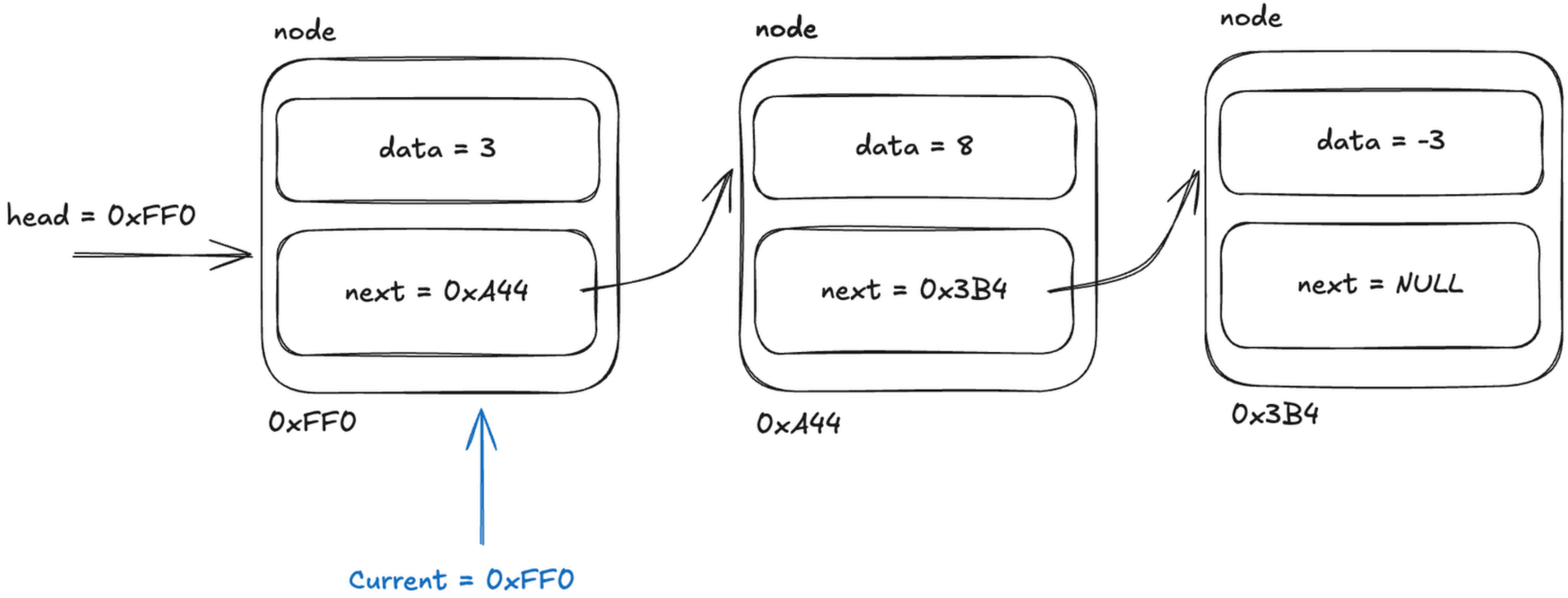
1. Create the next node to store 5 into (you need memory).
2. Assign 5 to data.
3. Make its next pointer point to NULL.
4. Insert it at the beginning so the head would now point to it and the new node would point to the old head



MOVING THROUGH A LIST

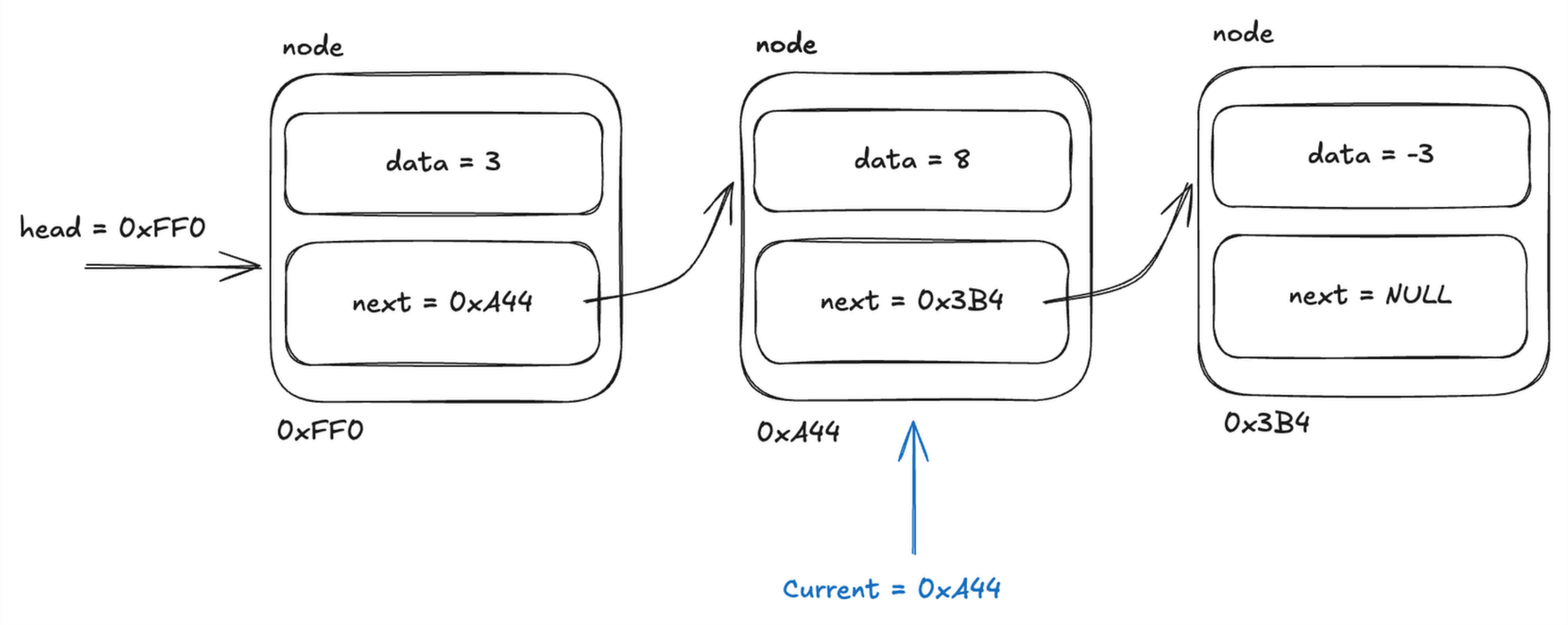
Set your current pointer to the head pointer to keep track of where you are currently located

struct node *current = head;



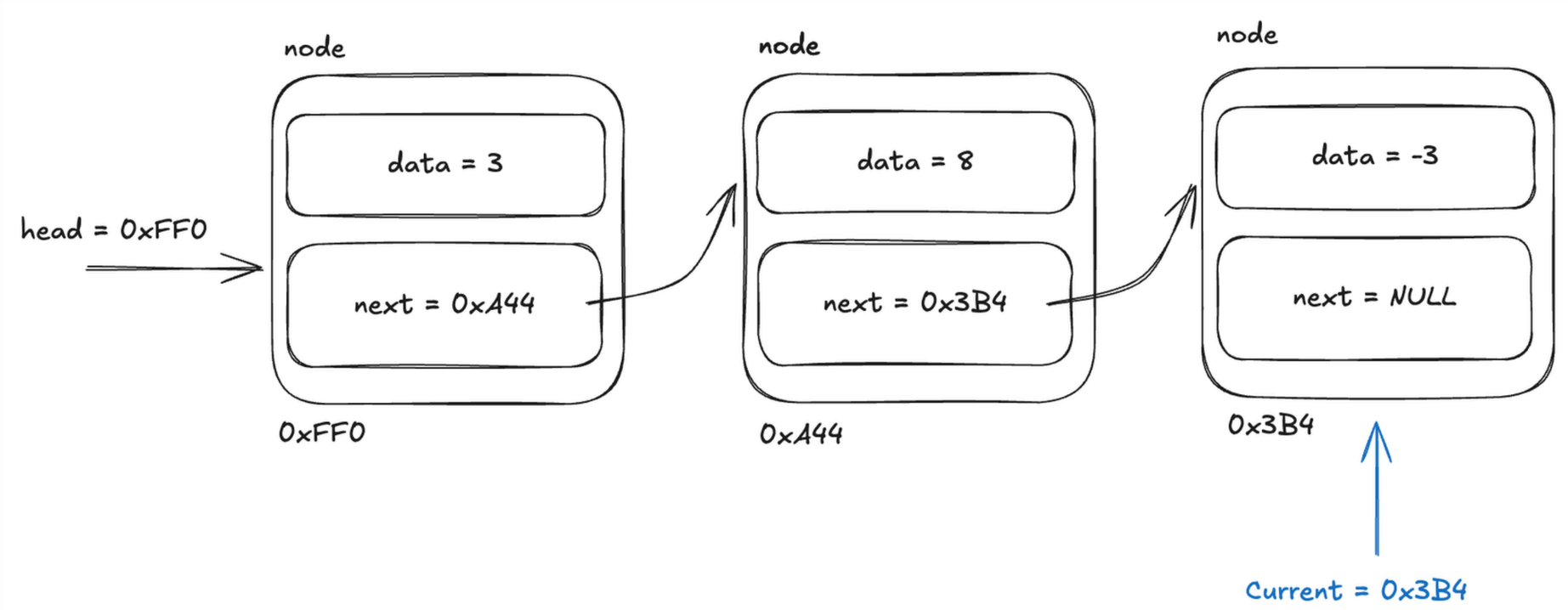
MOVING THROUGH A LIST

Now how would we move the current along?
current = current->next;



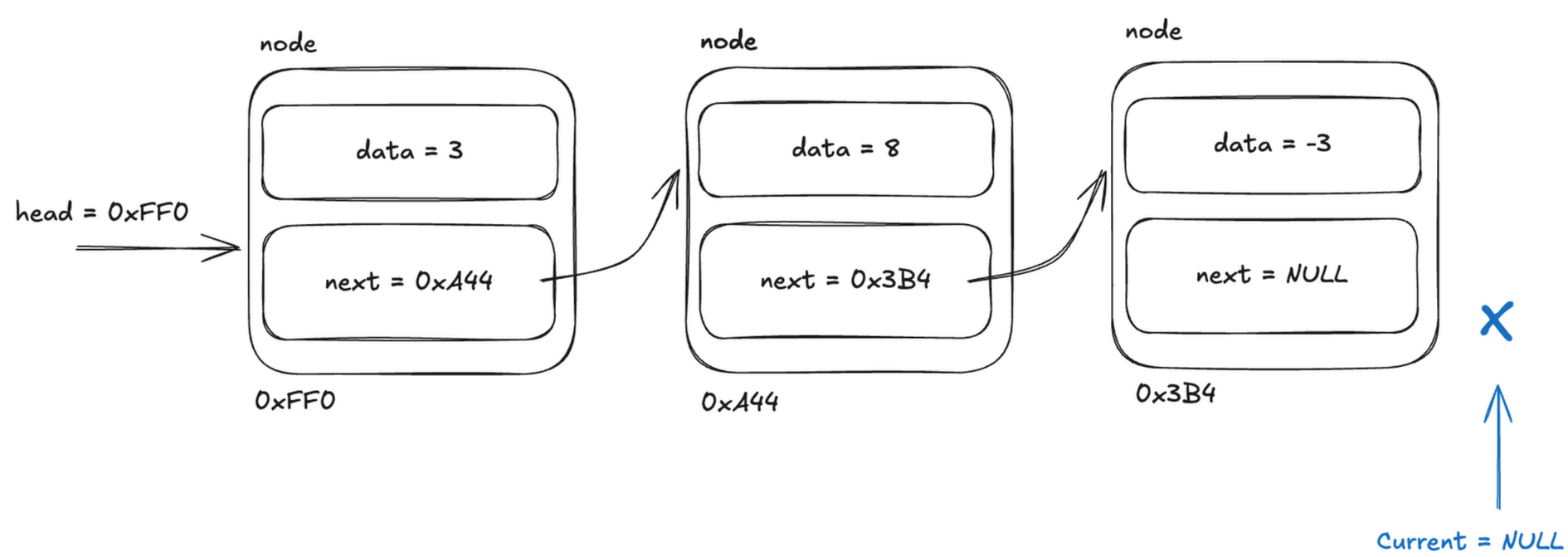
MOVING THROUGH A LIST

Now how would we move the current along?
current = current->next;



MOVING THROUGH A LIST

When should I be stopping?
while (current != NULL)



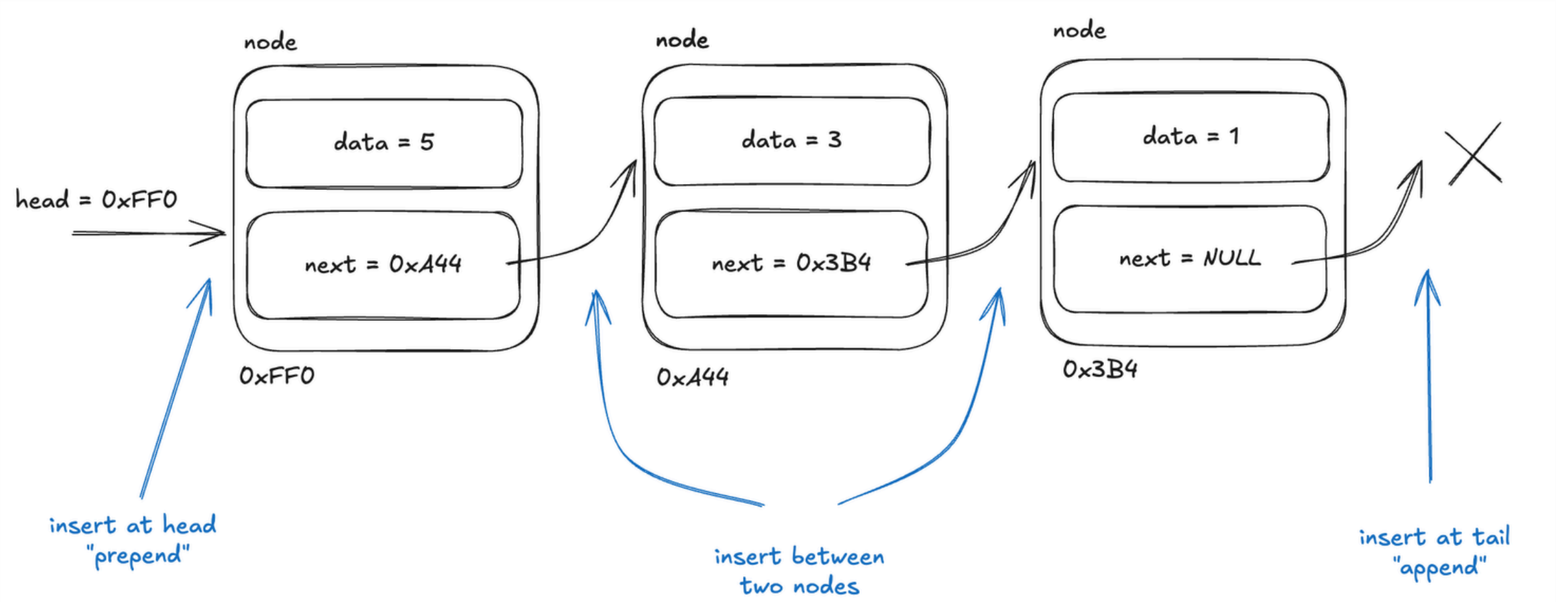
TRAVERSING A LINKED LIST

- 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.
- This is known as “traversing” a linked list.

INSERTING INTO A LINKED LIST

Where can I insert in a linked list?

- At the head (what we just did!)
- Between any two nodes that exist
- After the tail as the last node



THANK YOU

https://forms.office.com/r/5x6daiL5De