COMP1511 PROGRAMMING FUNDAMENTALS

LECTURE 11

Multi-file projects (finally)
Linked Lists - What is happening?
What is it? Inserting at the head, traversing it, inserting at the tail

LAST TIME...

- Pointers
- Malloc and free

TODAY...

- Multi-file projects (finally)
- Malloc and free rehash :)
- Linked Lists - what is it?
- Linked list - insert at the head
- Linked list - traversal
- Linked list - insert at the tail (if time?)

WHERE IS THE CODE?

Live lecture code can be found here:

HTTPS://CGI.CSE.UNSW.EDU.AU/~CS1511/25T1/CODE/WEEK_7

MULTI-FILE PROJECTS

WHAT ARE THEY?

- Big programs are often spread out over multiple files. There are a number of benefits to this:
  - Improves readability (reduces length of program)
  - You can separate code by subject (modularity)
  - 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

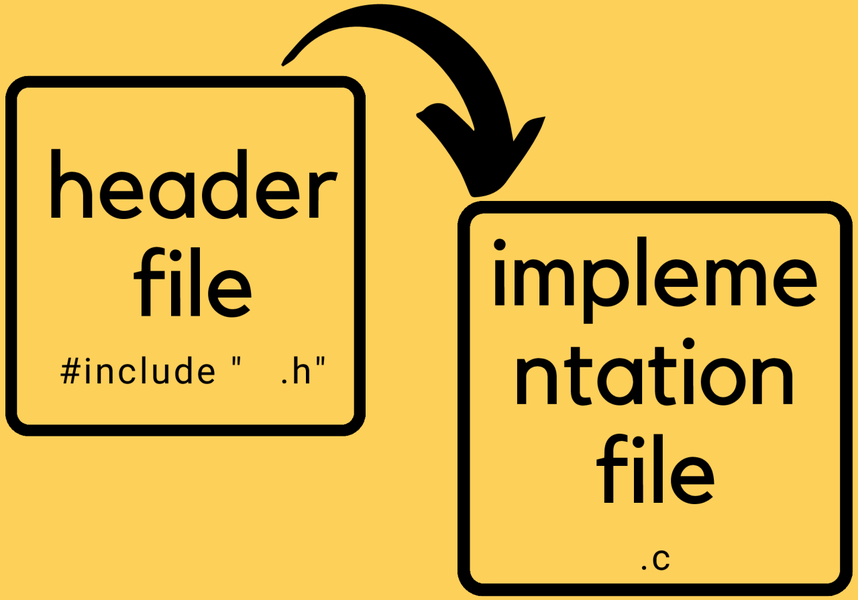
MULTI-FILE PROJECTS

WHAT ARE THEY?

- You can also #include your own! (FUN!)
- 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 (that is all that the C standard libraries 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, we try and have as little code here as possible

MULTI-FILE PROJECTS

.H HEADER FILE

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

MULTI-FILE PROJECTS

.C IMPLEMENTATION

implementation file

.c

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

MULTI-FILE PROJECTS

MAIN.C

This is where you call functions from that may exist in other modules.

MULTI-FILE PROJECTS

AN EXAMPLE

- We will have three files:
  - header file - maths.h
  - implementation file - maths.c
    - #include ”maths.h”
  - main file - main.c
    - #include ”maths.h”

MULTI-FILE PROJECTS

AN EXAMPLE HEADER FILE

```c
// This is the header file for the maths module
// example. The header file will contain:
// - any #defines
// - function prototypes and any comments

#define PI 3.14

// Function prototype for a function that
// calculate the square of a number:
int square(int number);

// Function prototype that calculates the sum of
// of two numbers
int sum(int number_one, int number_two);
```

MULTI-FILE PROJECTS

AN EXAMPLE IMPLEMENTATION FILE (NOTE TO INCLUDE THE HEADER THAT WE DEFINED!

```c
// This is the implementation file of maths.h
// We defined two functions in the header file (.h)
// and this is where we actually implement them

// Include your header file in the implementation file
// by using the below syntax:

#include "maths.h"

int square(int number) {
    return number * number;
}

int sum(int number_one, int number_two) {
    return number_one + number_two;
}
```

MULTI-FILE PROJECTS

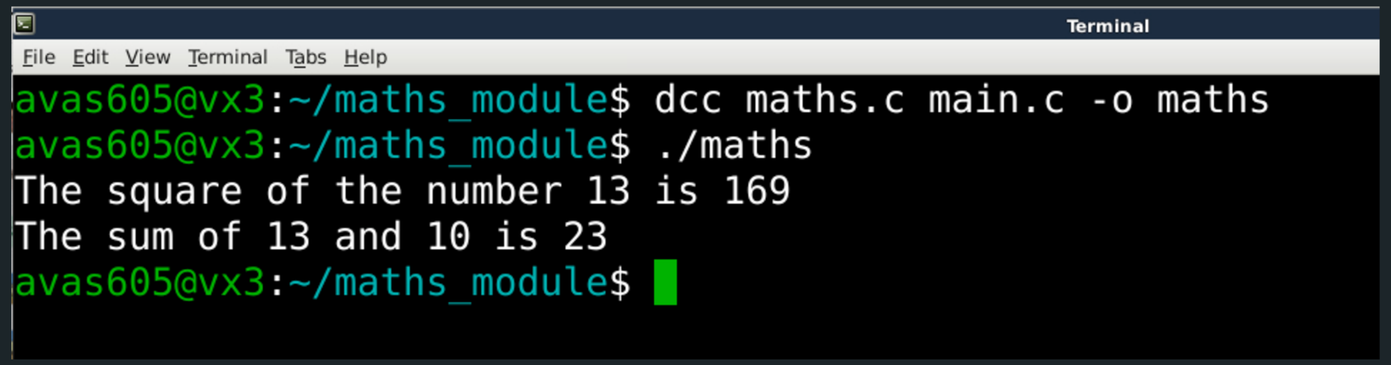
AN EXAMPLE OF MAIN THAT DRIVES OUR PROGRAM

```c
// This is the main file in our program.
// This is where we drive the program from
// and where we make calls to our modules. We
// need to inclide the header file for each
// module that we want to use functions from.

#include <stdio.h>
// Include our header file also
#include "maths.h"

int main(void) {
    int number_one = 13;
    int number_two = 10;

    printf("The square of the number %d is %d\n",
                                    number_one, square(number_one));
    printf("The sum of %d and %d is %d\n",
                    number_one, number_two, sum(number_one, number_two));
    return 0;
}
```

MULTI-FILE PROJECTS

COMPILING

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



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

REHASH

MALLOC()

- Allocate some memory by calling the function **`malloc()`** 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
  - this also allows us to dynamically create memory as we need it - neat!
  - This means that we are now in control of this memory (cue the evil laugh!)

REHASH

FREE()

It would be very impolite to keep requesting memory to be made (and hog all that memory!), without giving some back...

- 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()`**

HOW DO I KNOW HOW MUCH MEMORY TO ASK FOR WHEN I USE MALLOC()

SIZEOF()

- We can use the function **`sizeof()`** to give us the exact number of bytes we need to malloc (memory allocate)

```c
// This program demonstrates how sizeof() function works
// It returns the size of a particular data type
// We use the format specified %lu with it (long unsigned)
// if we want to print out the output of sizeof()

#include <stdio.h>

int main (void) {

    int array[10] = {0};

    // Example of using the sizeof() function
    printf("The size of an int is %lu bytes\n", sizeof(int));
    printf("The size of an array of int is %lu bytes\n", sizeof(array));
    printf("The size of a 10 ints is %lu bytes\n", 10 * sizeof(int));
    printf("The size of a double is %lu bytes\n", sizeof(double));
    printf("The size of a char is %lu bytes\n", sizeof(char));

    return 0;
}

```

FORMAT

MALLOC()

- Using the **`malloc()`** function:

```
1 int *ptr = malloc(x * sizeof(int));
```

if you need to have space for more than one element, you multiply it by the number of elements you need

the pointer that malloc will return to indicate the start of the portion of space it has allocated

using the function

specify data type that you need

FORMAT

MALLOC()

- Using the **`malloc()`** function example

```
1 int *ptr = malloc(10 * sizeof(int));
```

This will create a piece of memory of 10 * 4 bytes = 40 bytes and return the address of where this memory is in ptr

STRUCTS AND POINTERS

-> VERSUS .

- Remember that when we access members of a struct we use a .

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

#define MAX 15

// 1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
    // 2. Declare struct
    struct dog jax;

    // 3. Initialise struct (access memebers with .)
    // Remember we can't just do jax.name = "Jax"
    // So can use the function strcpy( ) in <string.h>
    // to copy the string over

    strcpy(jax.name, "Jax");
    jax.age = 6;

    printf("%s is an awesome dog, who is %d years old\n", jax.name, jax.age);
    return 0;
}

```

STRUCTS AND POINTERS

-> VERSUS .

- What happens if we make a pointer of type struct?

  How do we access it then?

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

#define MAX 15

// 1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
    // 2. Declare struct
    struct dog jax;

    // Have a pointer to the variable jax of type struct dog
    struct dog *jax_ptr = &jax;

    // How would we initialise it using the pointer?
    // Perhaps dereference the pointer and access the member?

    strcpy((*jax_ptr).name, "Jax");
    (*jax_ptr).age = 6;

    printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name, (*jax_ptr).age);
    return 0;
}

```

STRUCTS AND POINTERS

-> VERSUS .

- Those brackets can get quite confusing, so there is a shorthand way to do this with an ->
- There is no need to use (*jax_ptr) and instead can just straight jax_ptr ->

```
1
2     // INSTEAD OF THIS:
3     //strcpy((*jax_ptr).name, "Jax");
4     //(*jax_ptr).age = 6;
5     //printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name, (*jax_ptr).age);
6     // DO THIS:
7     strcpy(jax_ptr->name, "Jax");
8     jax_ptr->age = 6;
9
0     printf("%s is an awesome dog, who is %d years old\n", jax_ptr->name, jax_ptr->age);
1
```

WHY ARE YOU HURTING US WITH ALL THIS STUFF?

WE HAVE COME TO THE ULTIMATE REVEAL.

- Now that you have become comfortable with arrays, we are going to become acquainted with another important data structure (drum roll please 🥁):

- The one and only LINKED LIST 🥳

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

WHERE IS IT USED?

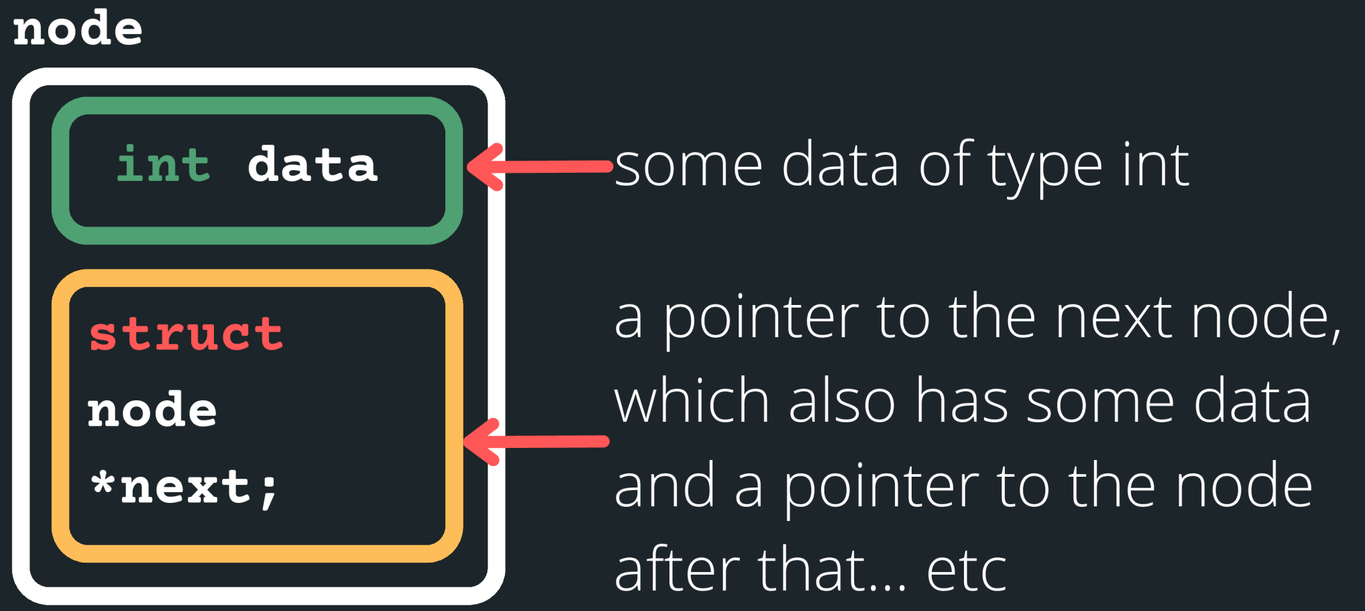
- Web browsers (think back buttons)
- Music Players (playlists)
- Can you think of some more?

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:

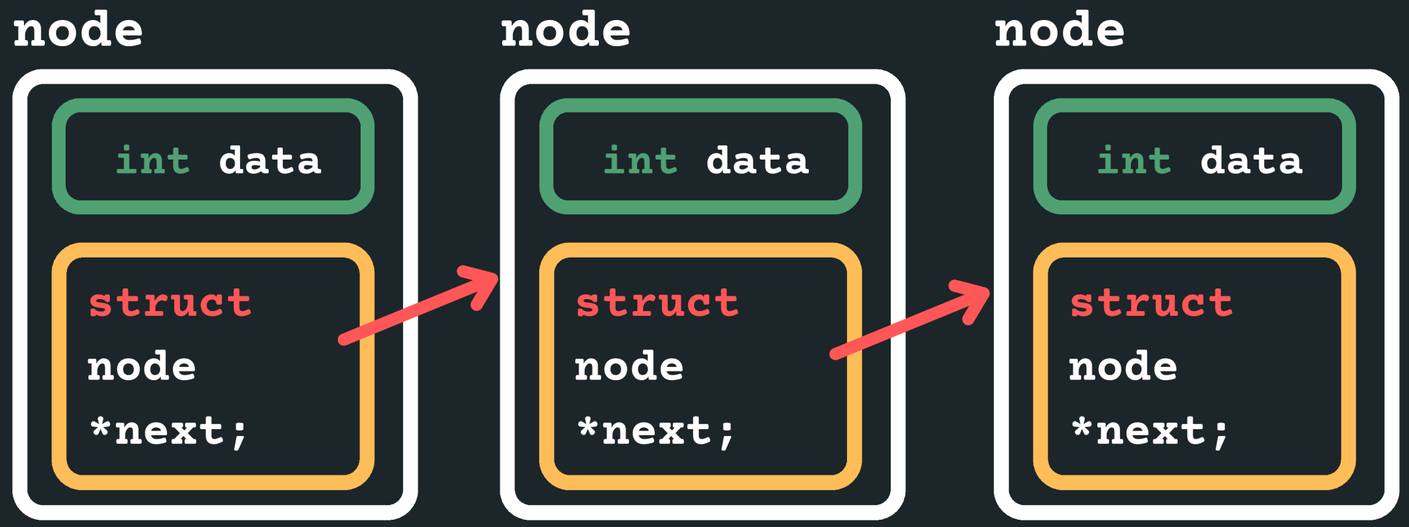
```
struct node {
    int data;
    struct node *next;
};
```



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

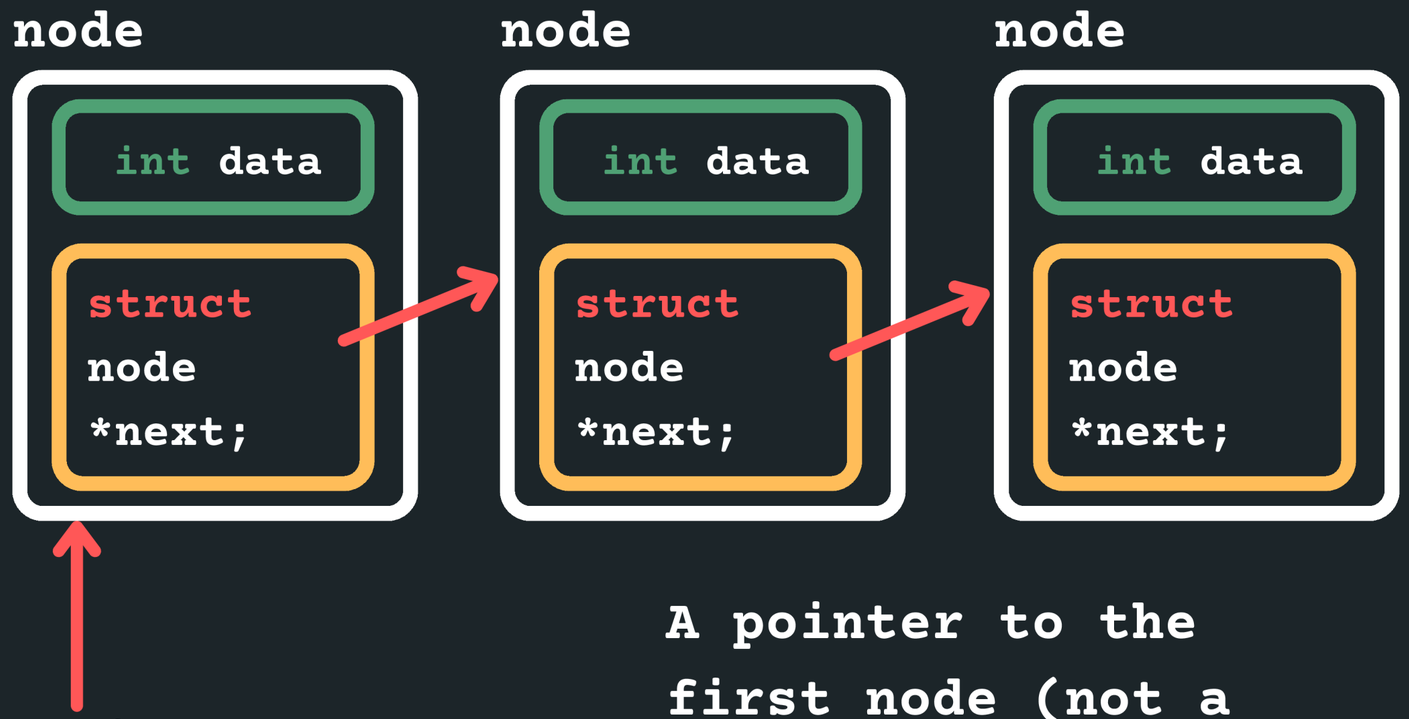


- But how do I know where the linked list starts?

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?

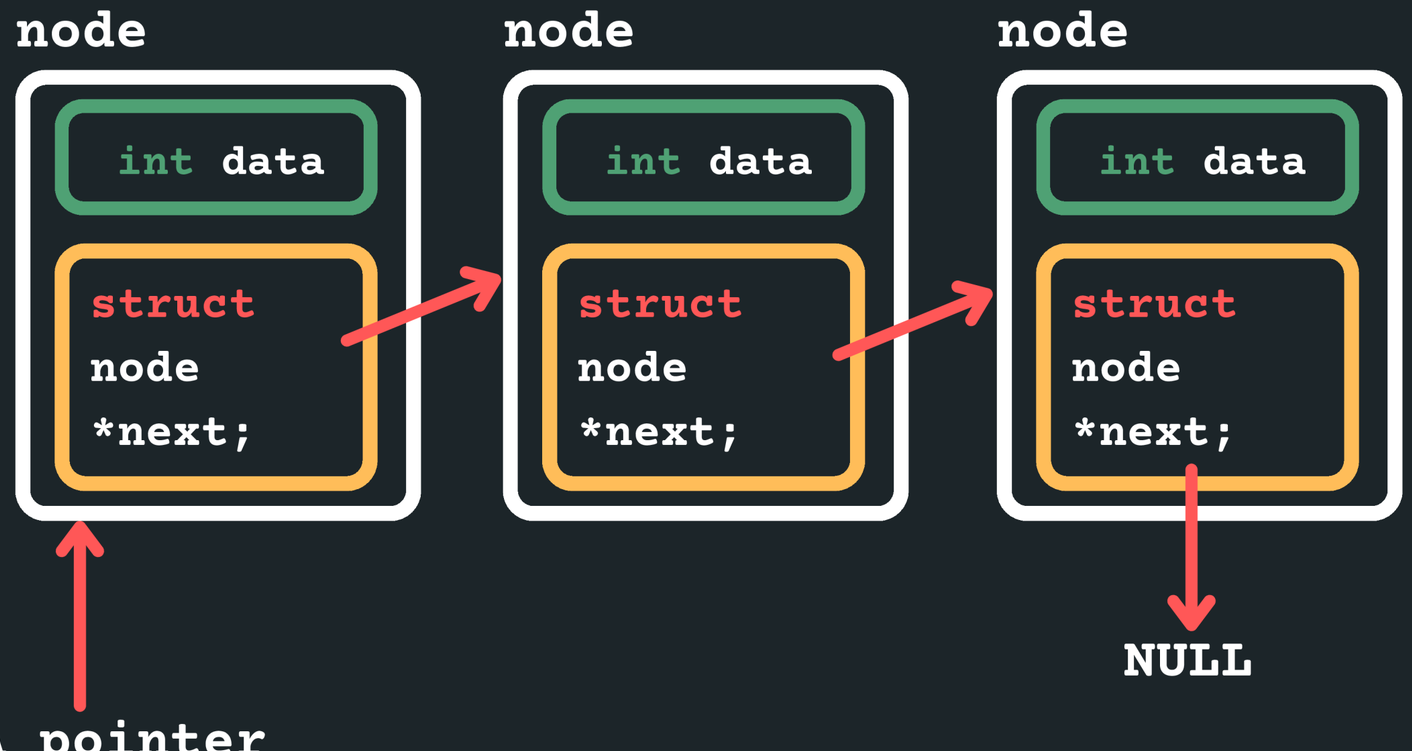


- How do I know when my list is finished?

A LINKED LIST IS MADE UP OF MANY NODES

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

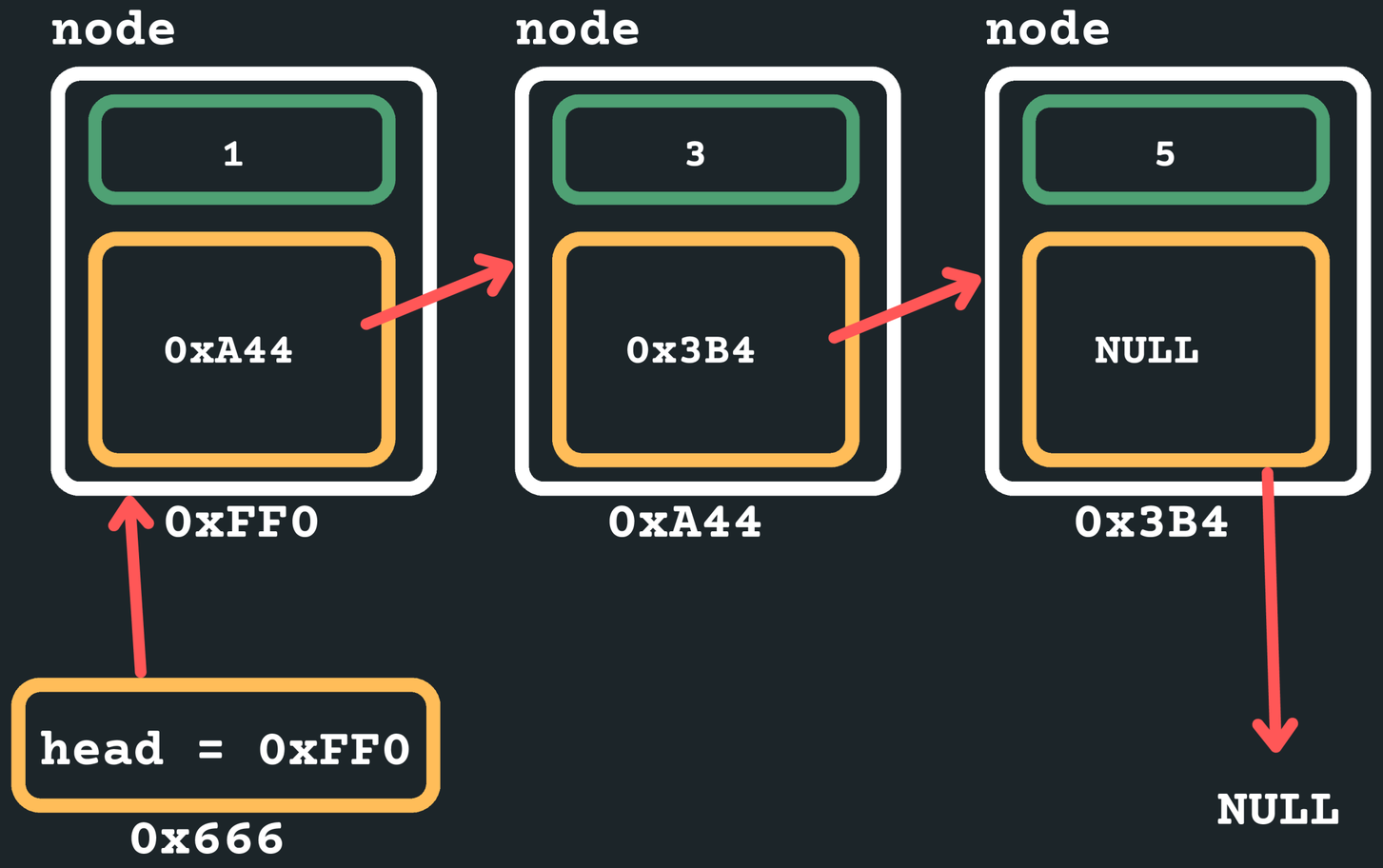
- Pointing to a NULL at the end!



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

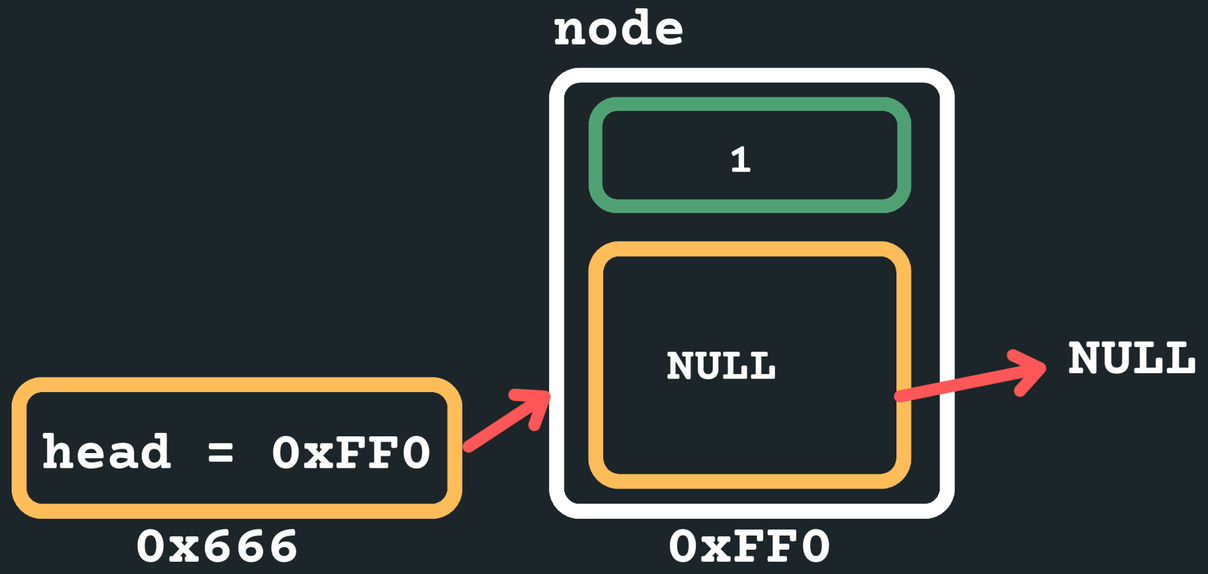
HOW DO WE CREATE ONE AND INSERT INTO IT?

- 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?

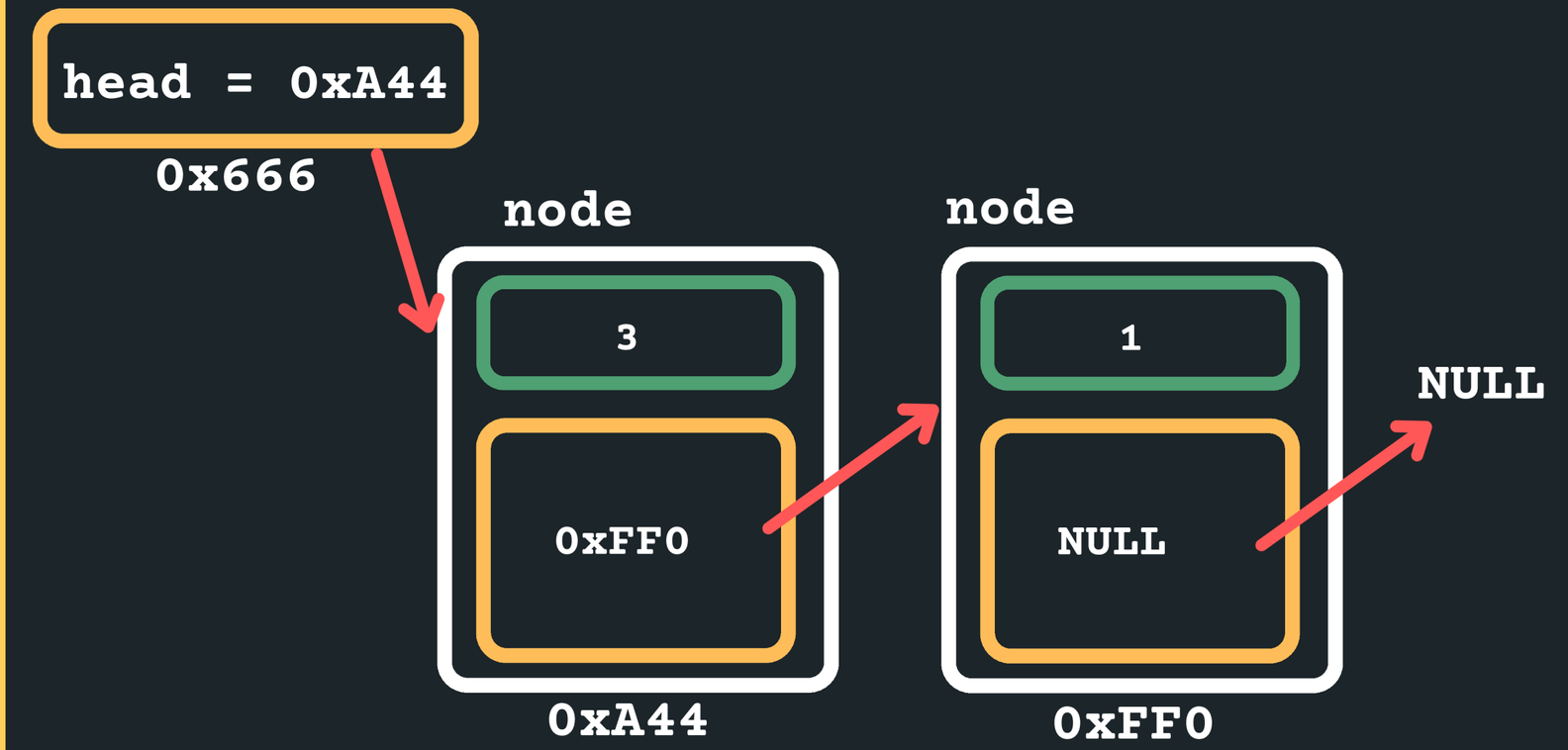
- Let’s say we wanted to create a linked list with 5, 3, 1
  - Let’s create the first node to start the list!
  - A pointer to keep track of where the start of the list is and by default the first node of the list
  - It will point to NULL as there are no other nodes in this list.



A LINKED LIST

HOW DO WE CREATE ONE AND INSERT INTO IT?

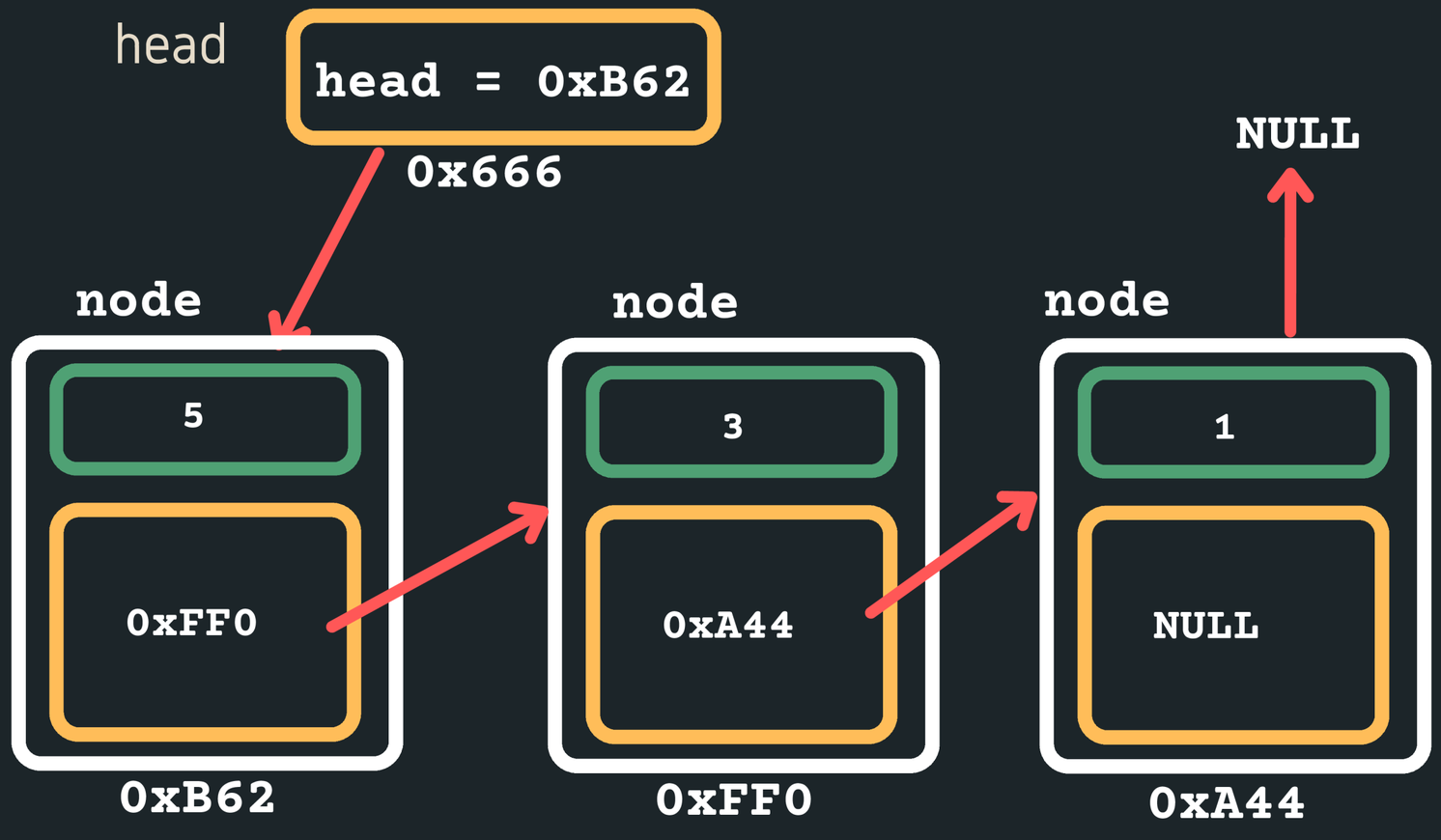
- Create the next node to store 3 into (you need memory)
- Assign 3 to data
- and insert it at the beginning so the head would now point to it and the new node would point to the old head



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



BREAK TIME...

You have five boxes in a row numbered 1 to 5, in one of which, a cat is hiding. Every night he jumps to an adjacent box, and every morning you have one chance to open a box to find him. How do you win this game of hide and seek - what is your strategy? What if there are n boxes?

A LINKED LIST

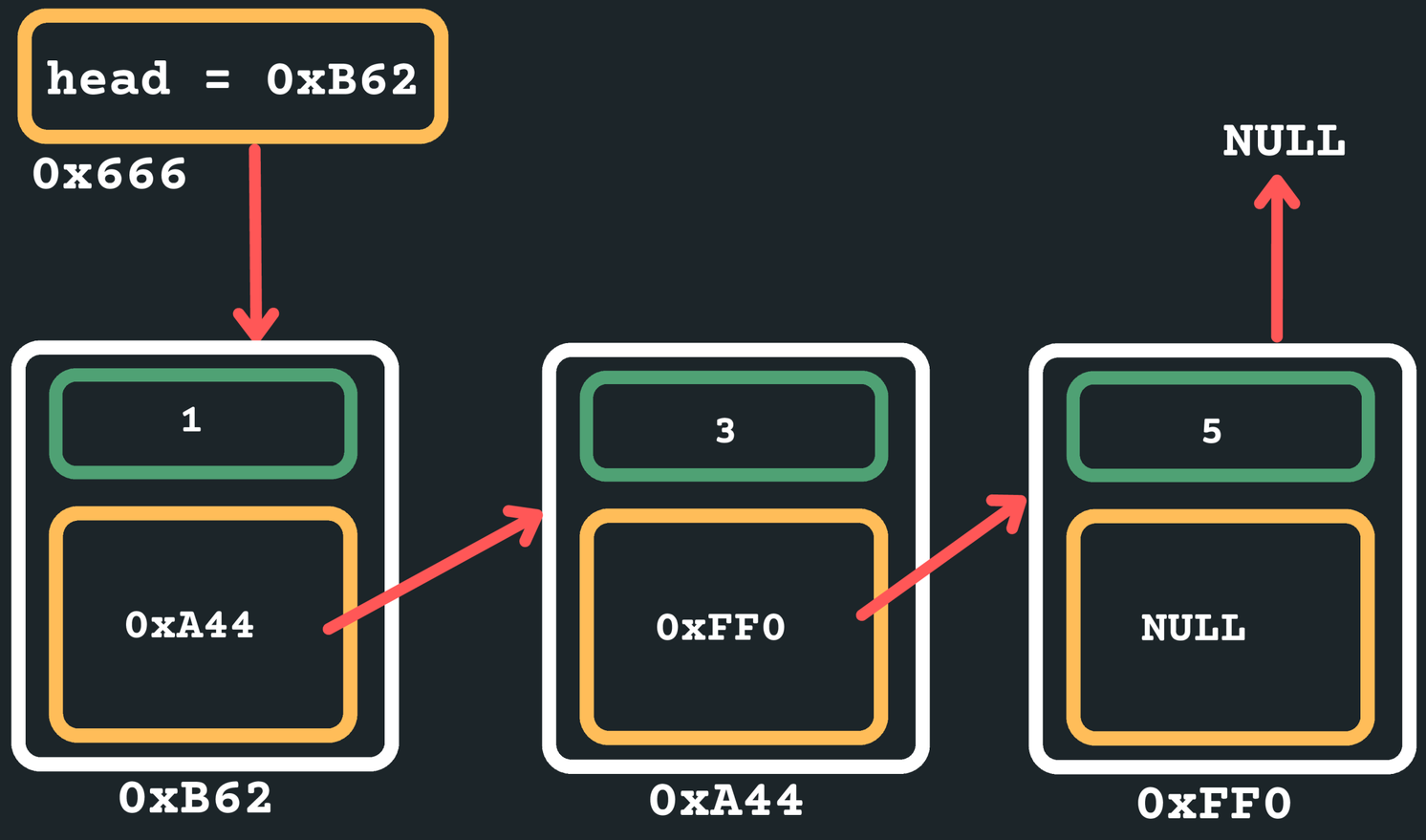
PUTTING IT ALL TOGETHER IN CODE

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

A LINKED LIST IS MADE UP OF MANY NODES

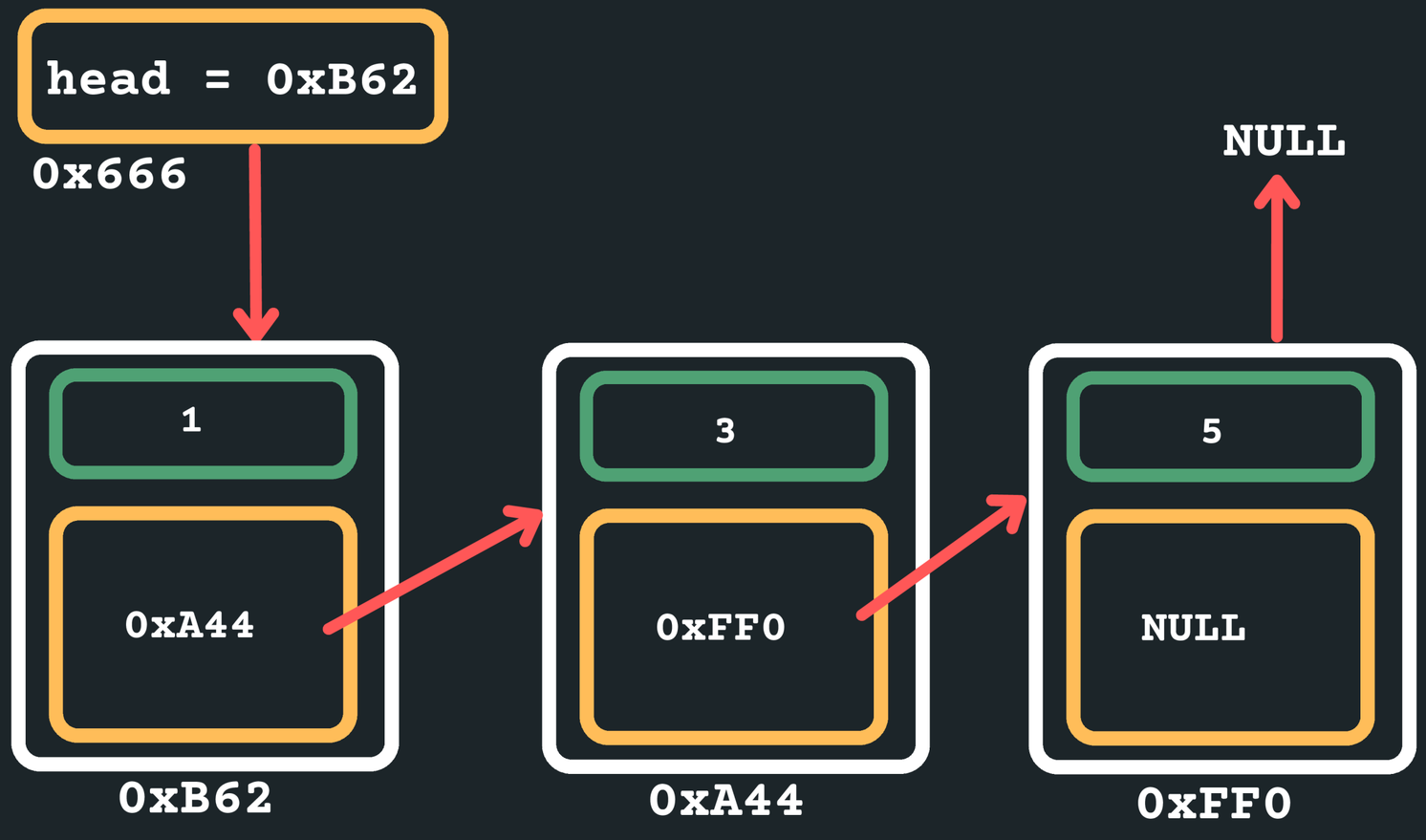
THE NODES ARE LINKED TOGETHER (A SCAVENGER HUNT OF POINTERS)

- For example a list with 1, 3, 5



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

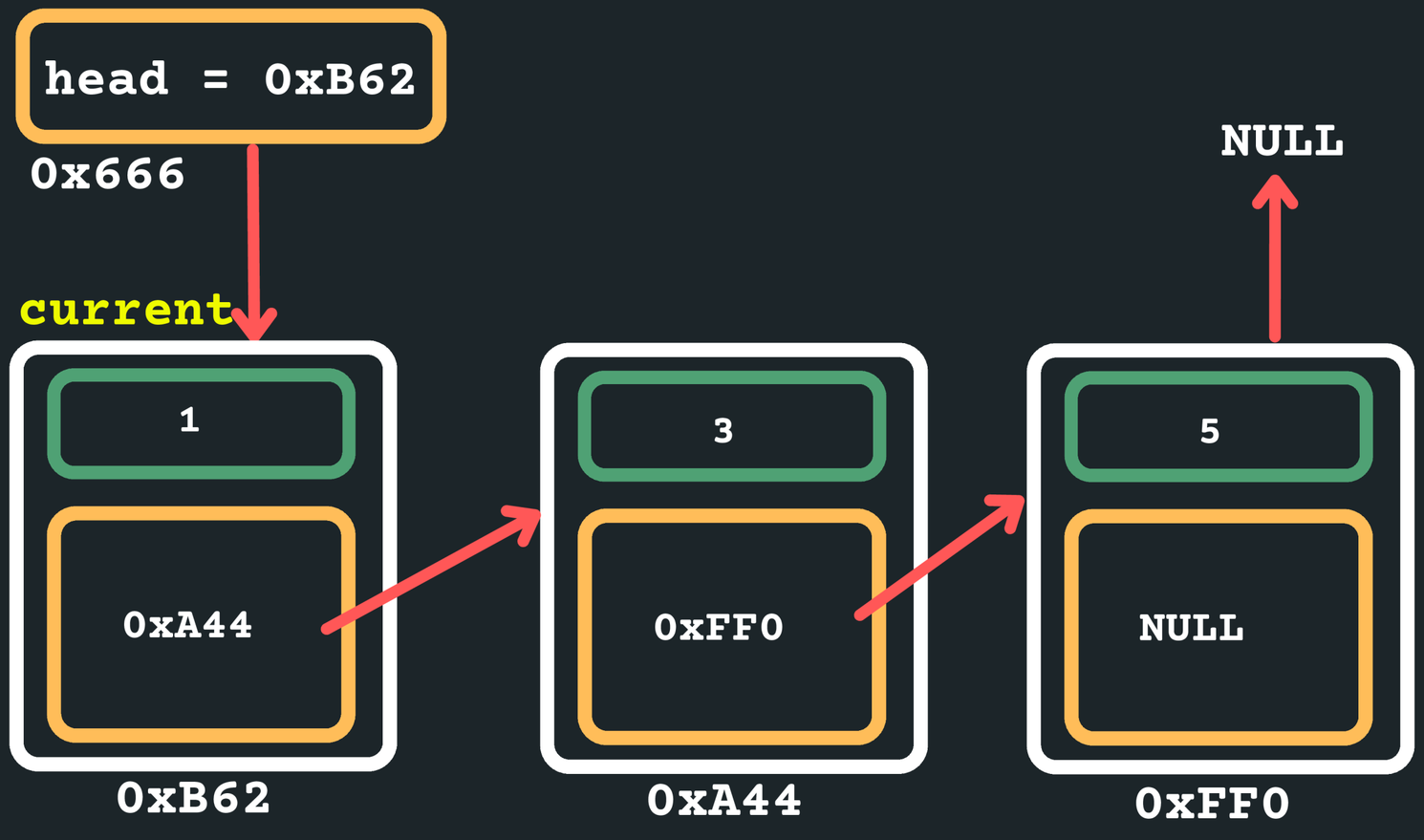
- How do you think we can move through the list to start a the head and then move to each subsequent node until we get to the end of the list...



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

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

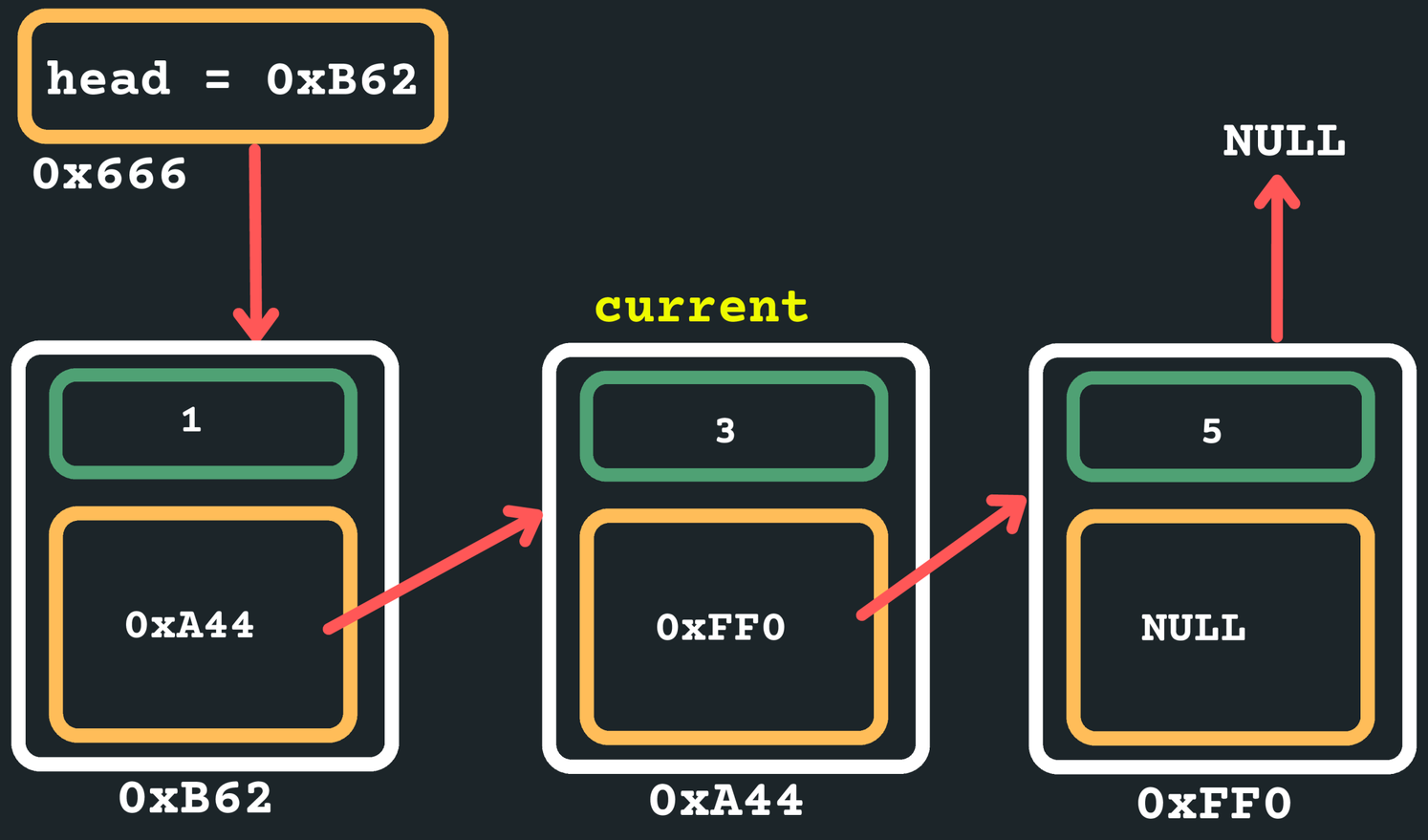
```
struct node *current = head
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

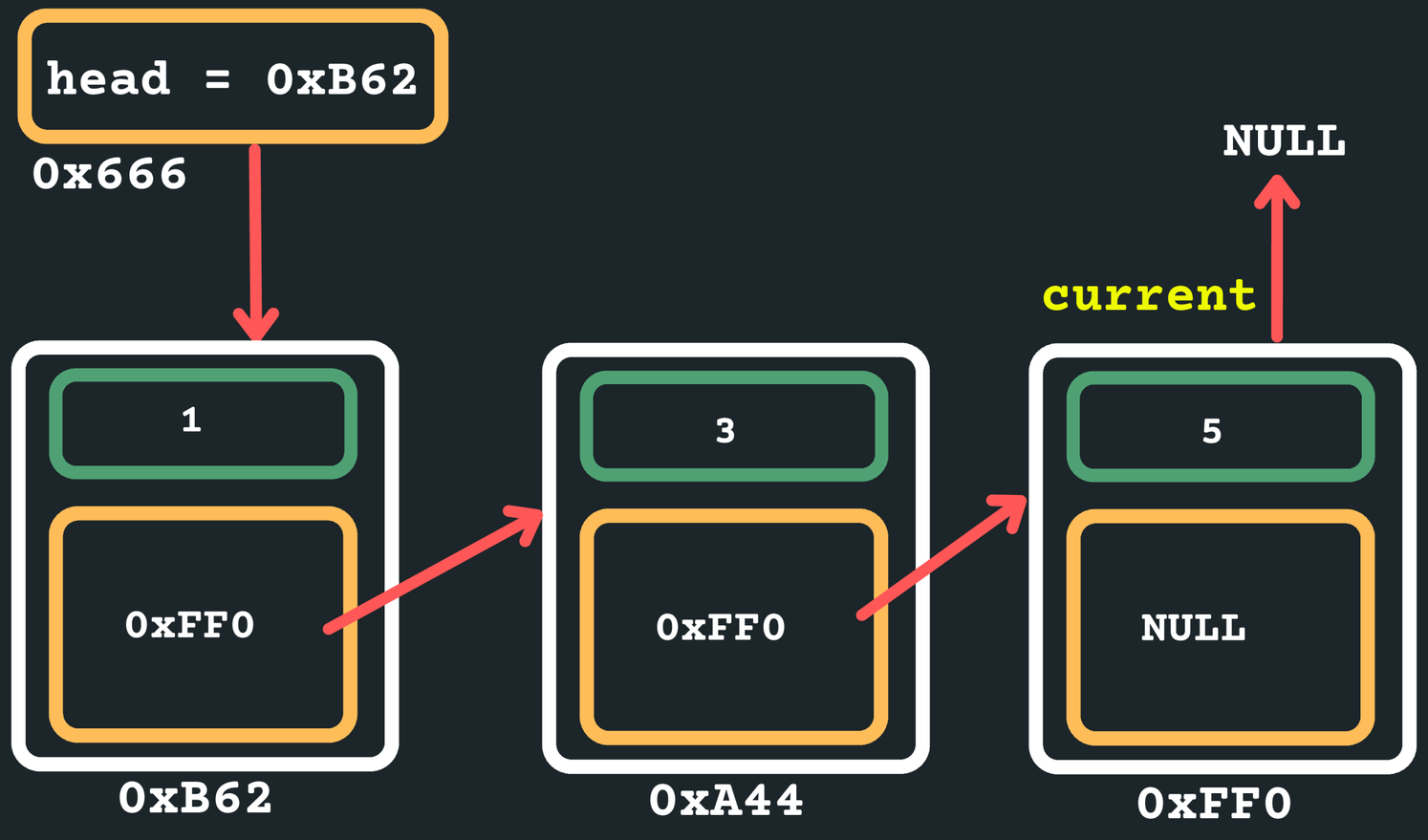
```
current = current->next
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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



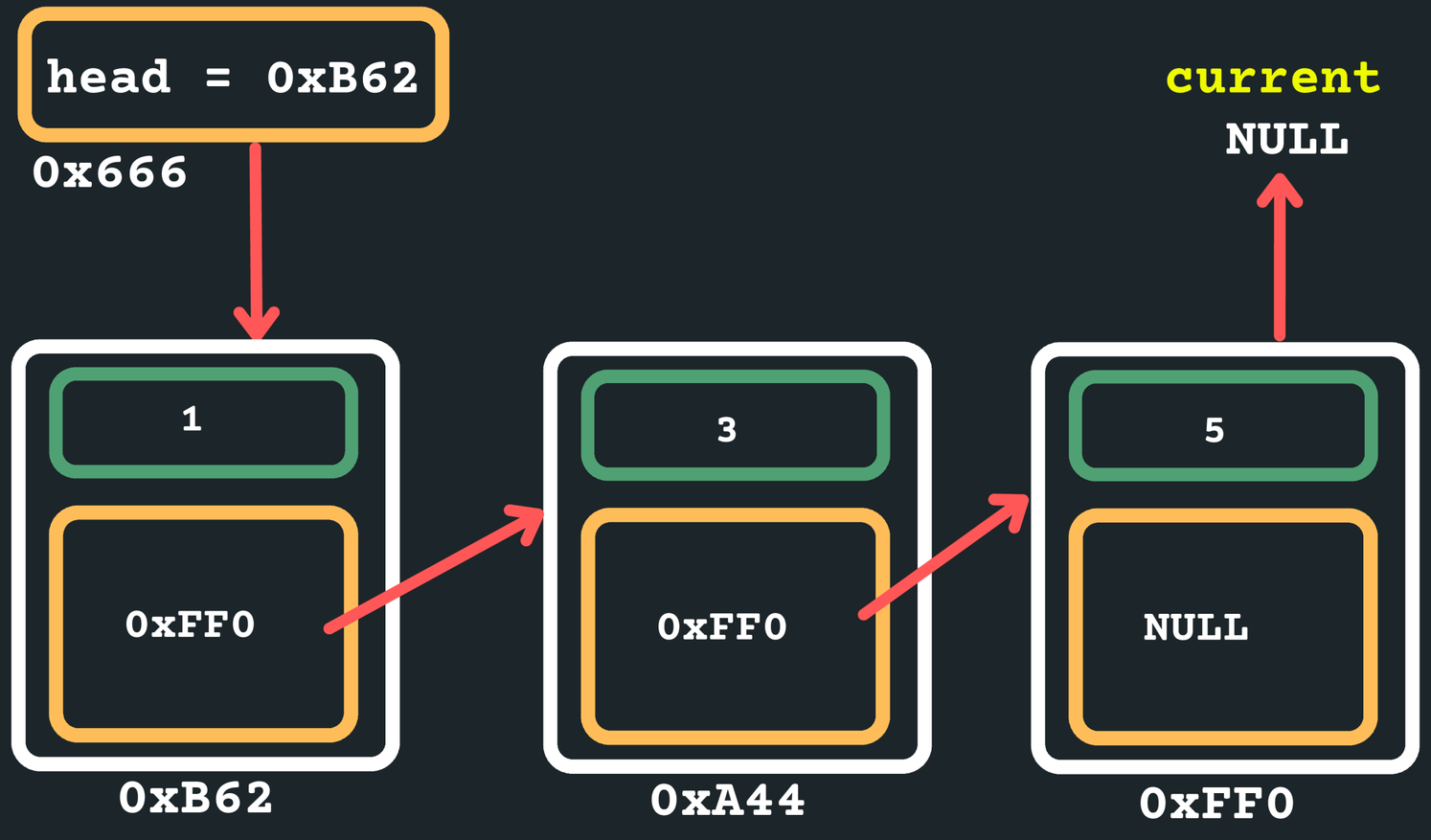
HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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

When should I be stopping?

```
while (current != NULL)
```



SO TRAVERSING A LINKED LIST...

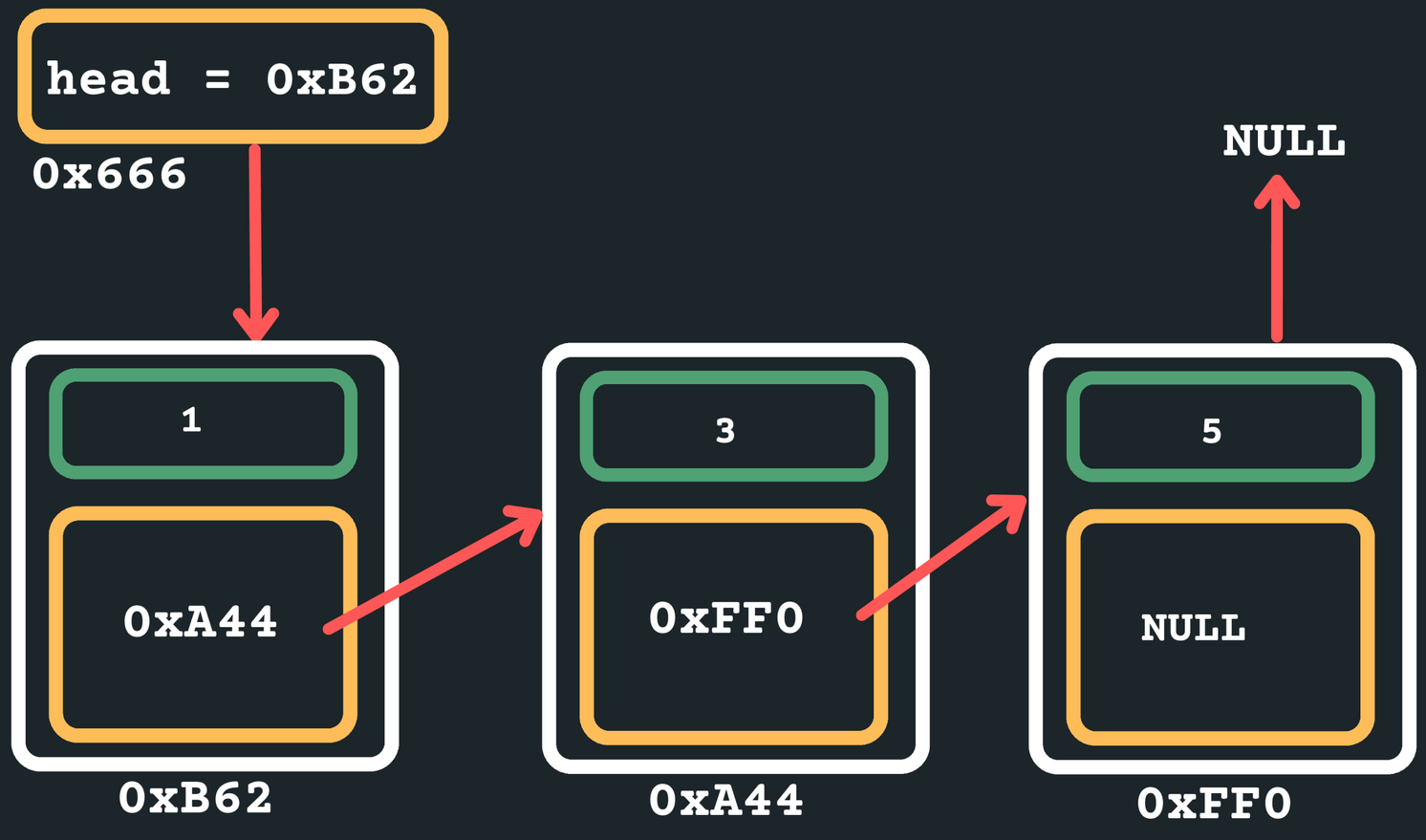
- The only way we can make our way through the linked list is like a scavenger hunt, we have to follow the links from node to node (sequentially! we can't skip nodes)
- We have to know where to start, so we need to know the head of the list
- When we reach the NULL pointer, it means we have come to the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

SO NOW, LET'S PRINT EACH NODE OUT...

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

INSERTING ANYWHERE IN A LINKED LIST...

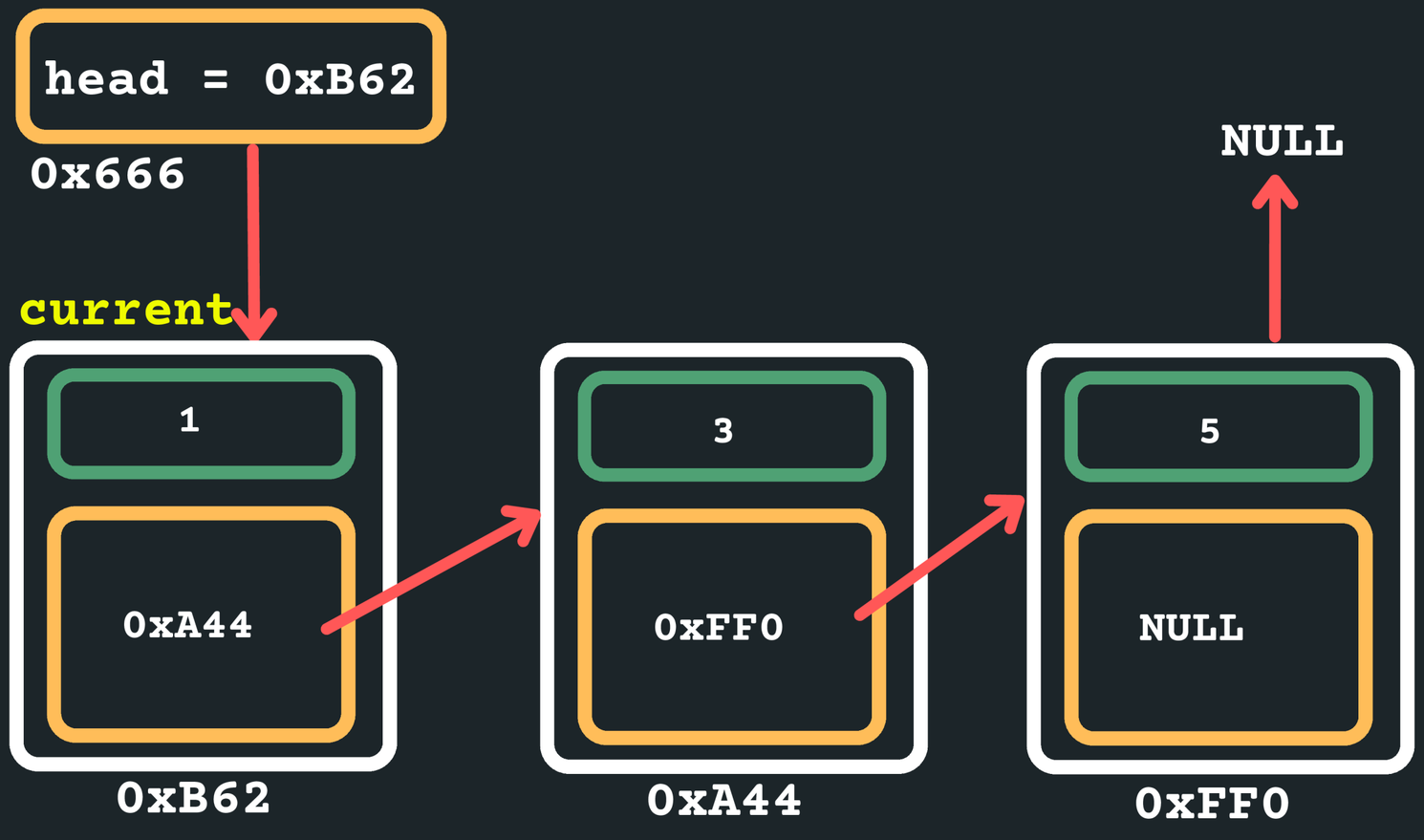
- Where can I insert in a linked list?
  - At the head (what we just did!)
  - Between any two nodes that exist (next lecture!)
  - After the tail as the last node (now!)



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

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

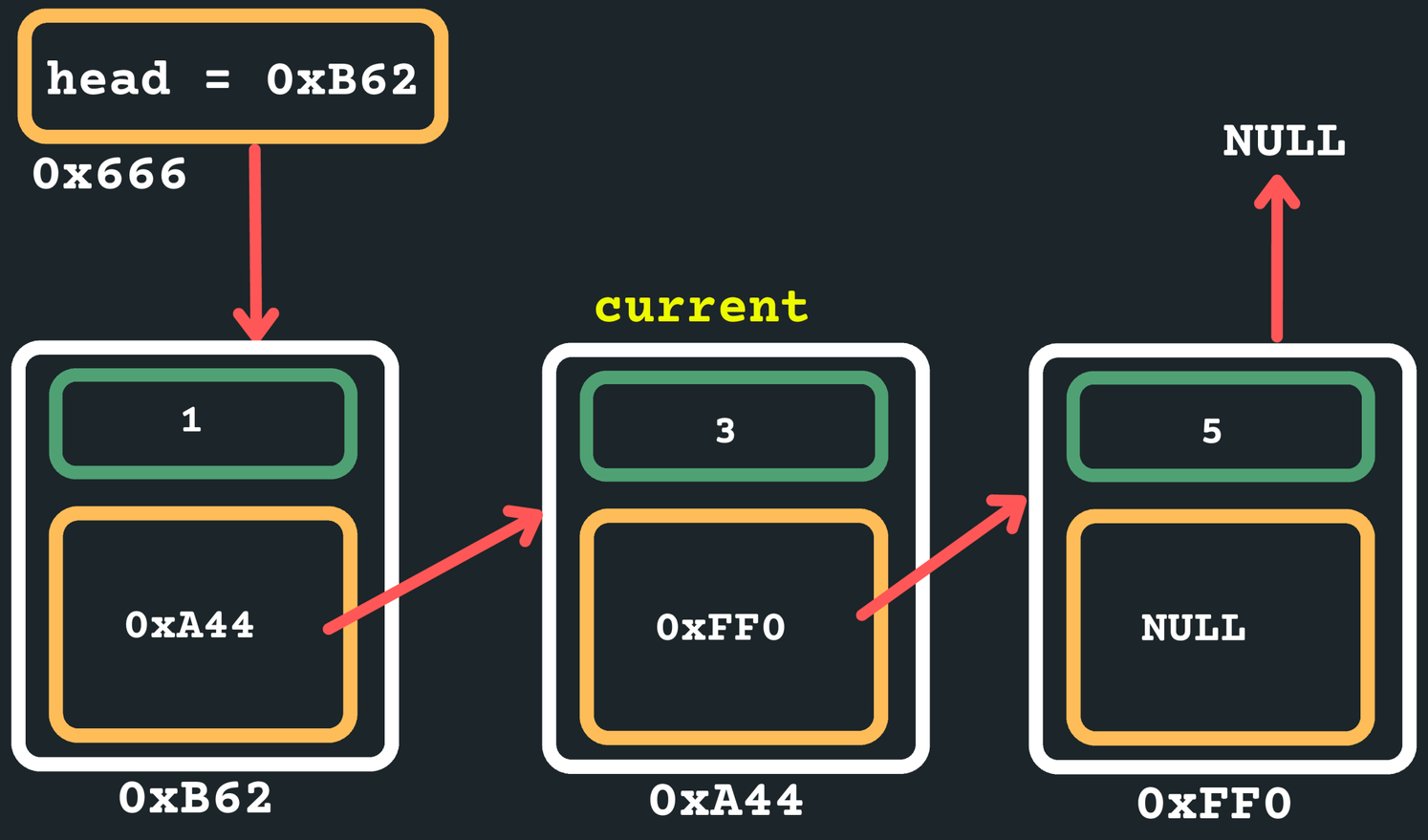
```
struct node *current = head
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

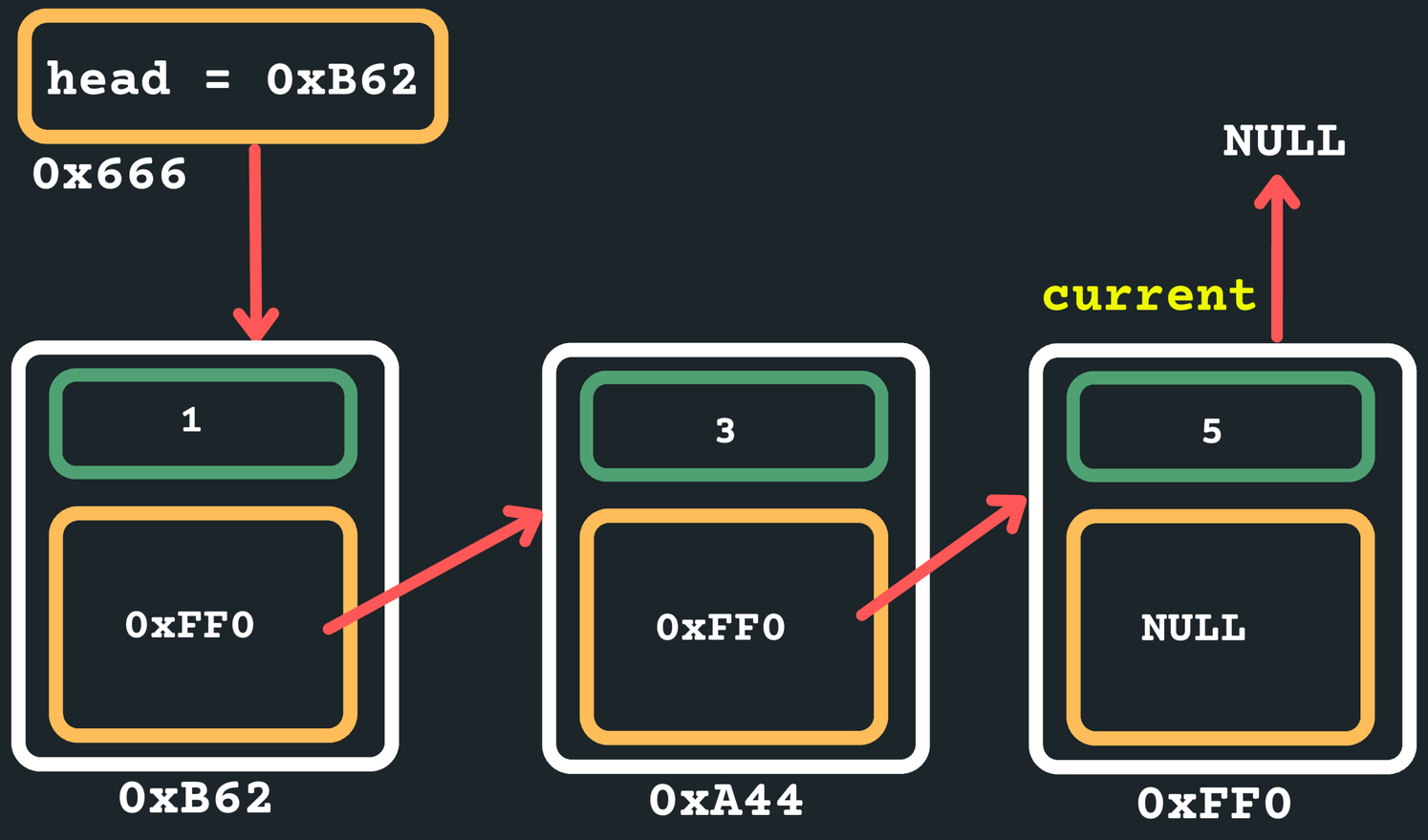
```
current = current->next
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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



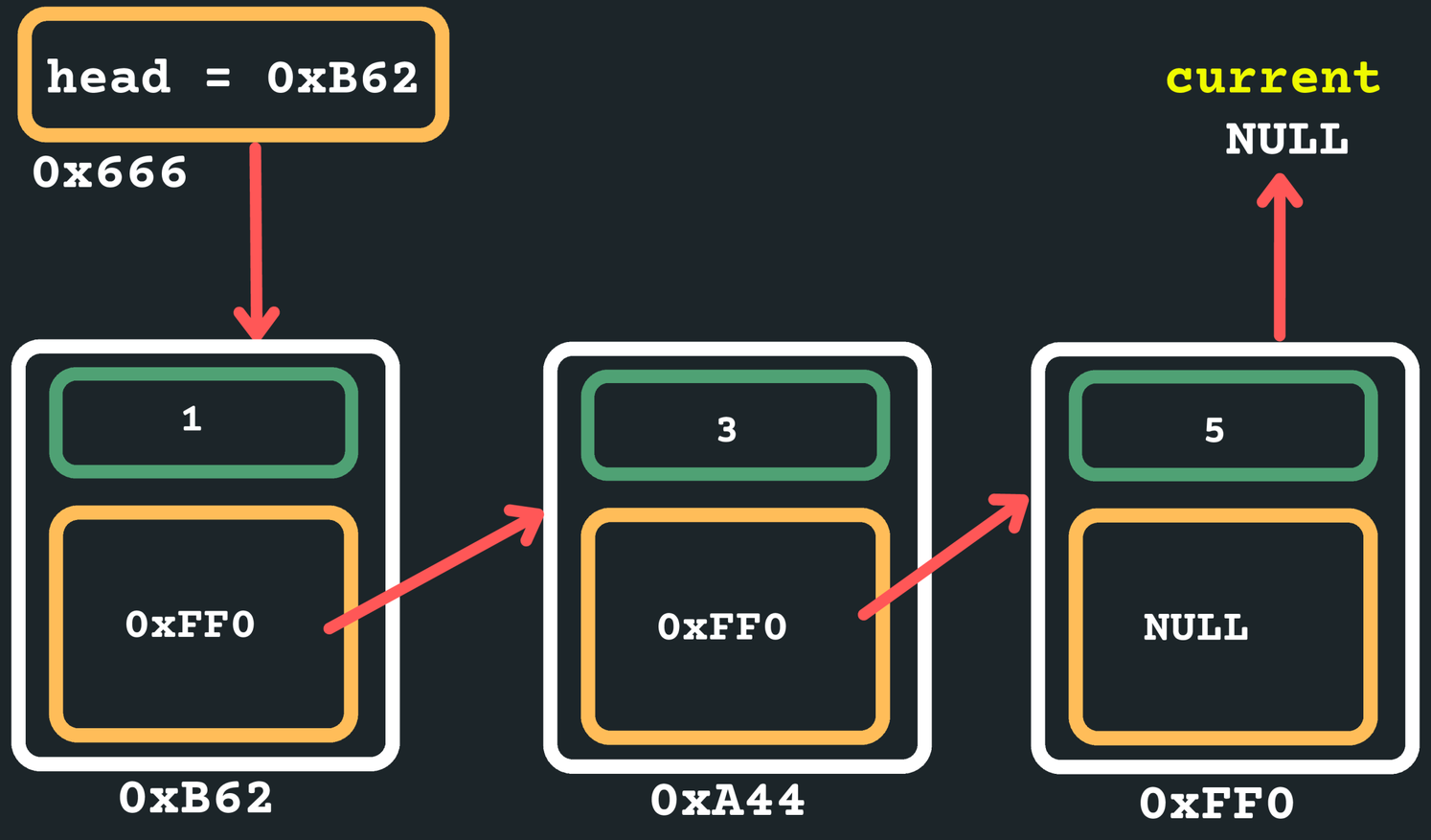
HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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

When should I be stopping?

```
while (current != NULL)
```



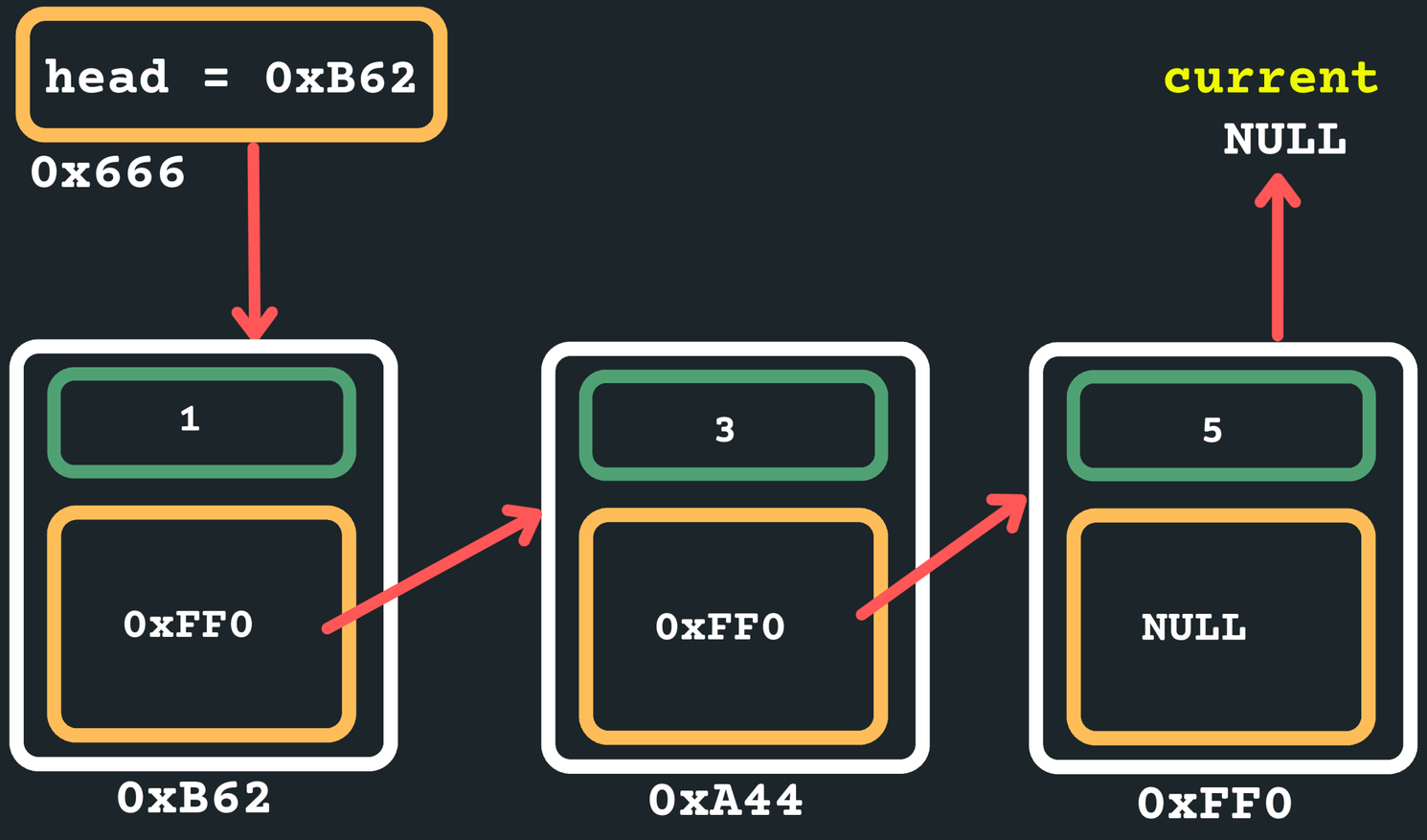
HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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

When should I be stopping? If you stop at current = NULL that means you won't know what the address of the previous node is!

```
while (current != NULL)
```



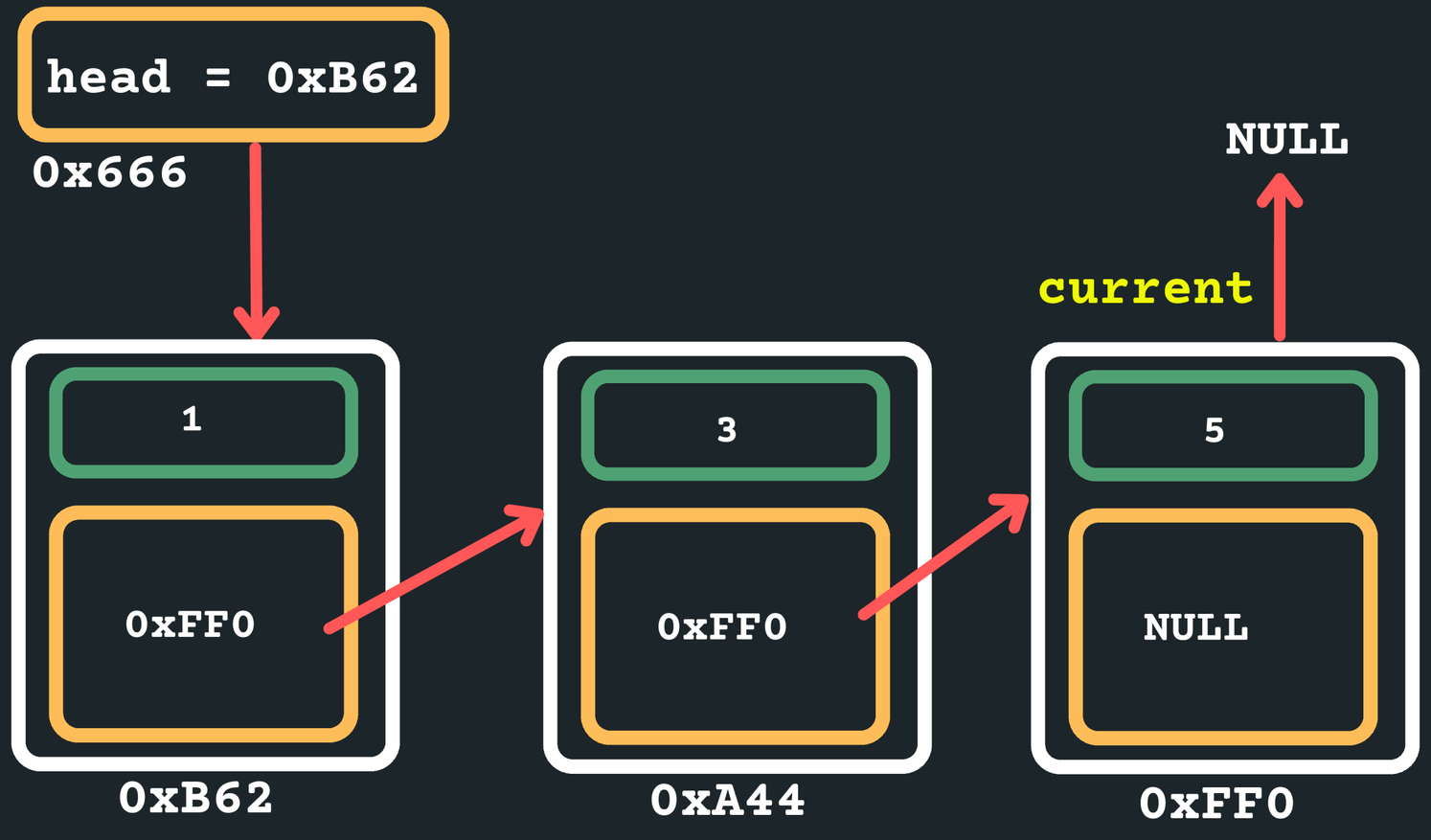
HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now how would we move the current along?

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

So let's stop at the last node...

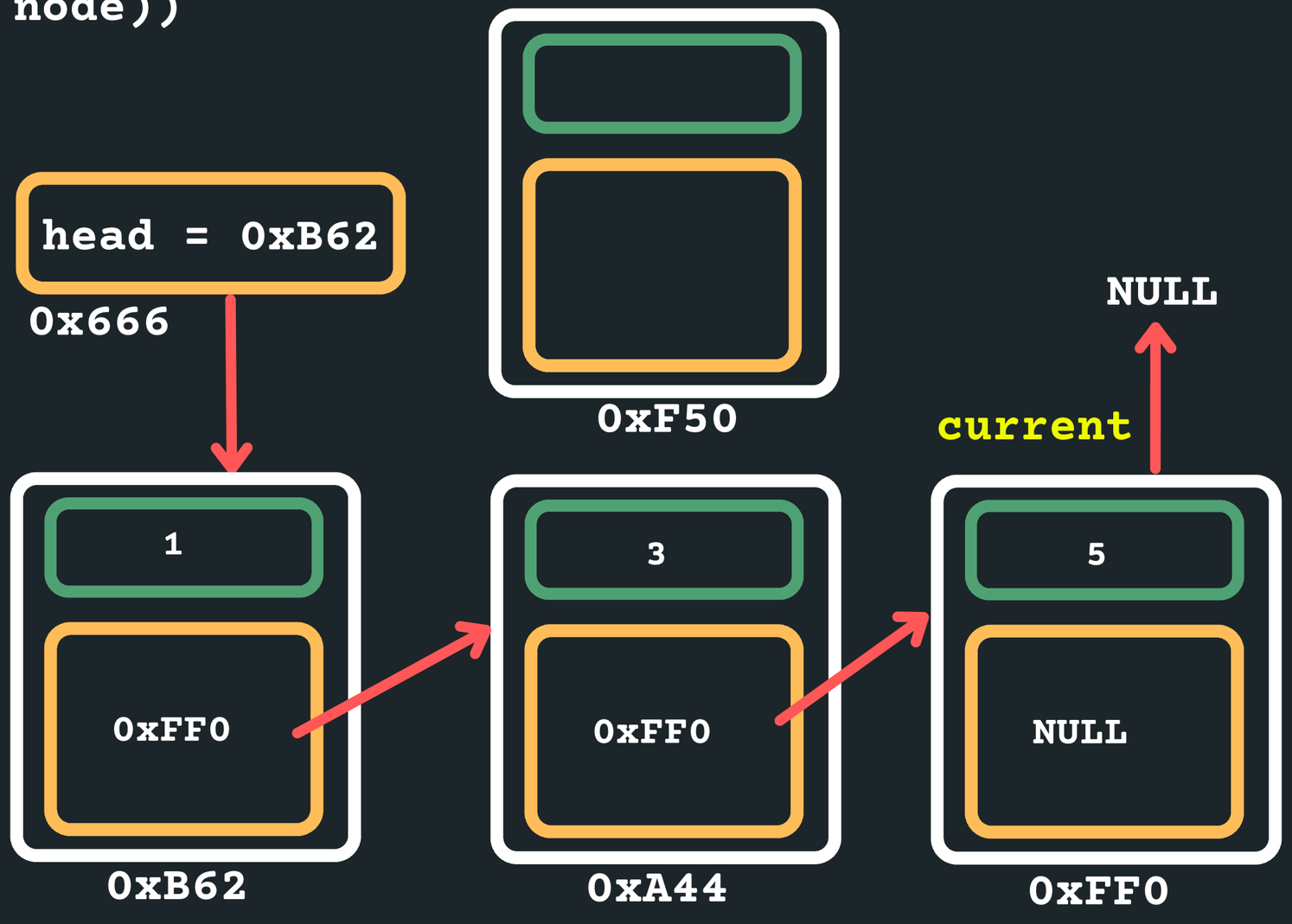
```
while (current->next != NULL)
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now we want to create a new node to insert:

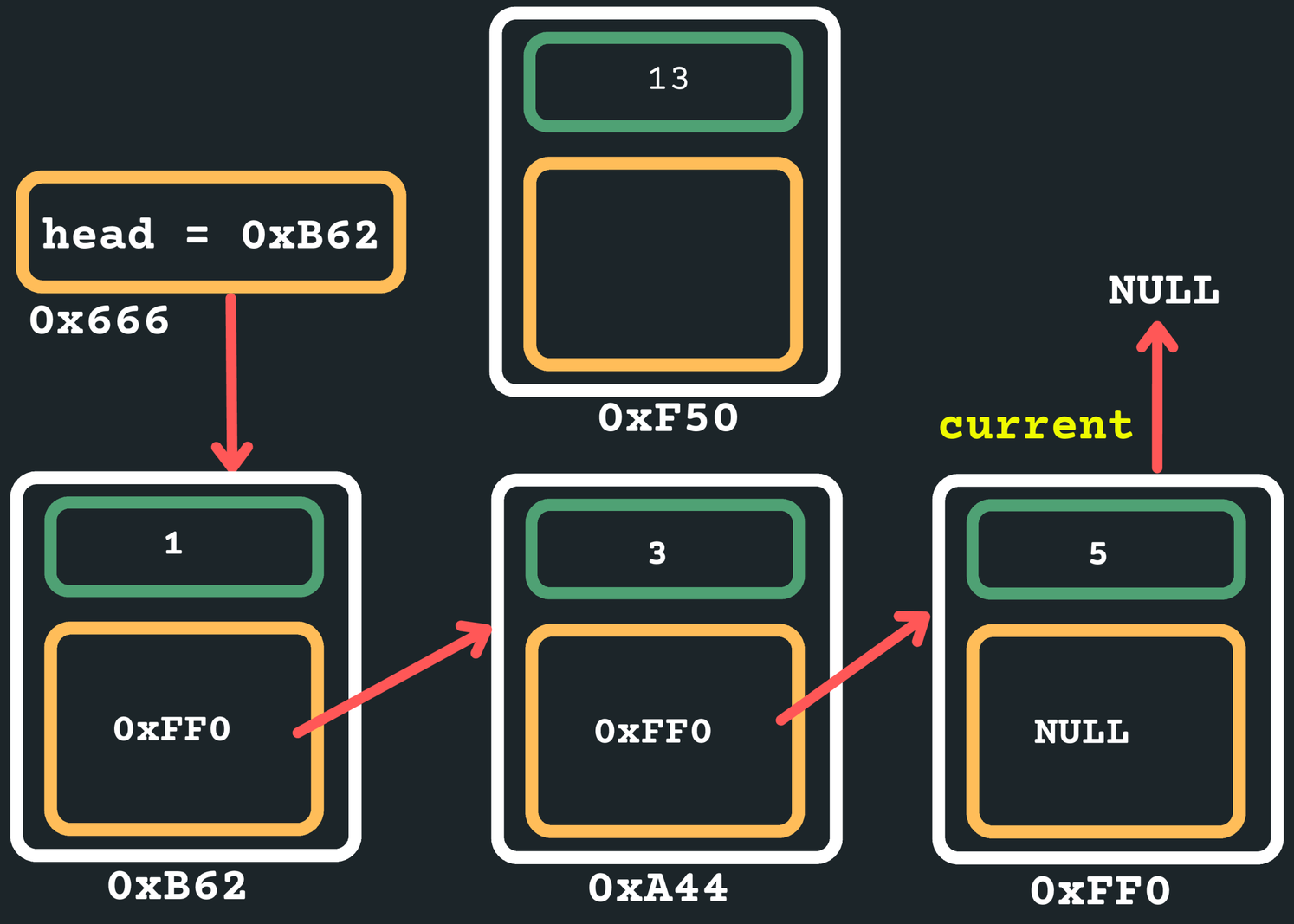
```
struct node new_node = malloc(sizeof(struct node))
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Assign values to new node:

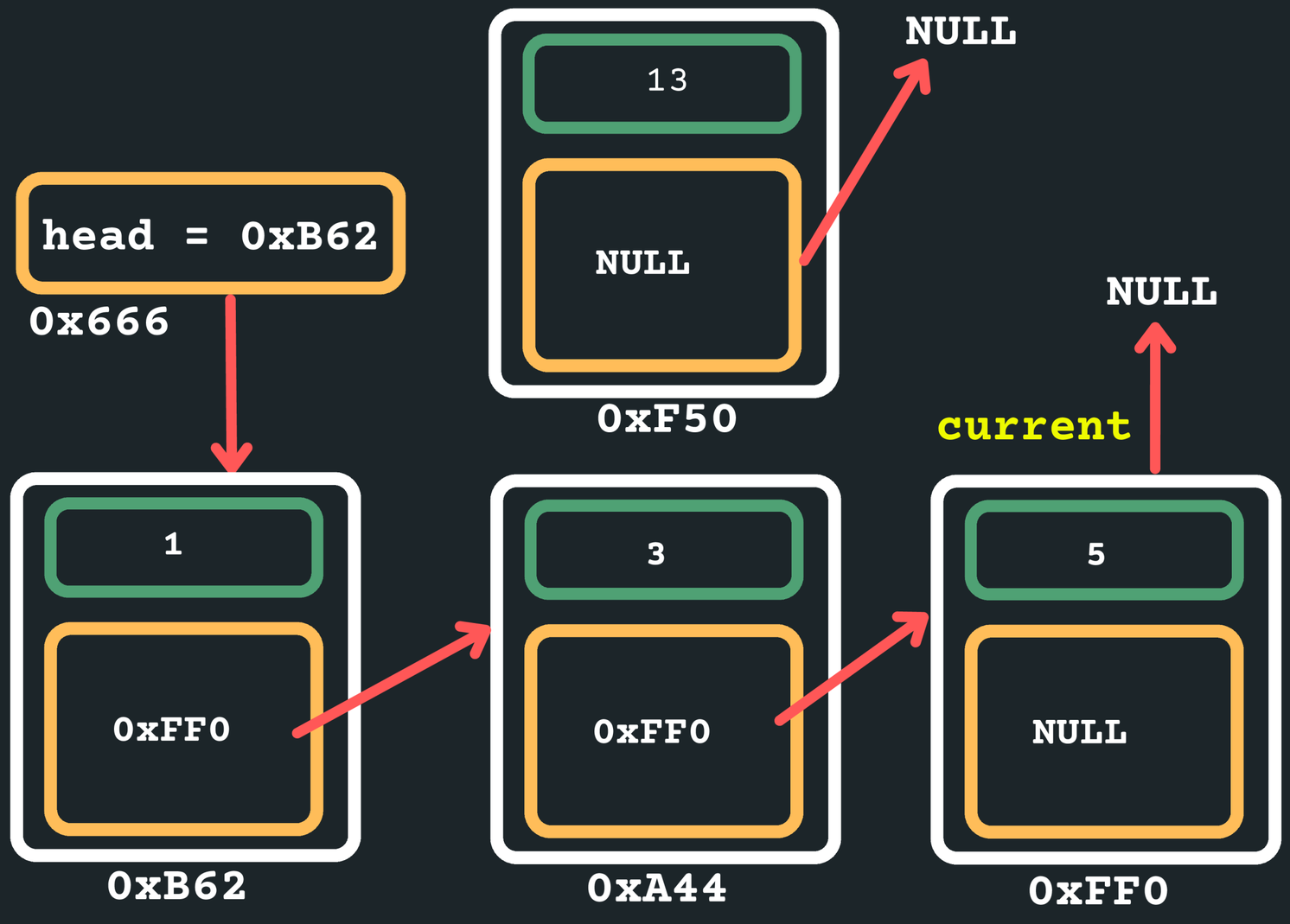
```
new_node->data = 13;
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Because this will be the last node point it to NULL

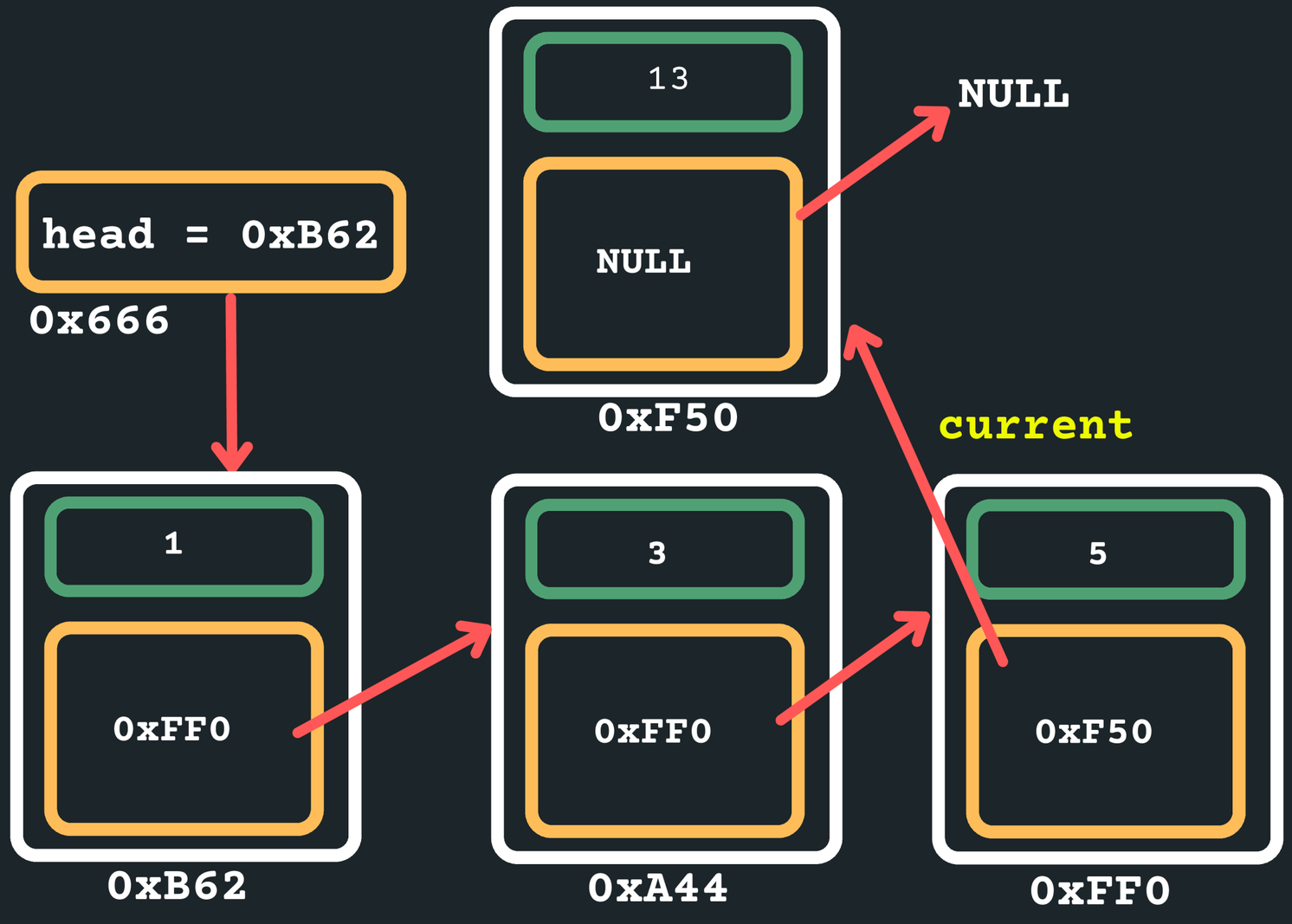
```
new_node->next = NULL;
```



HOW CAN WE MOVE THROUGH THIS LIST TO FIND NEXT NODE?

Now point our current last node to the new node

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



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://forms.office.com/r/SdwfGte8MK

WHAT DID WE LEARN TODAY?

MULTIFILE PROJECTS

maths.c

maths.h

main.c

LINKED LIST

What is it?

linked_list.c

LINKED LIST

Insert at the head

linked_list.c

LINKED LIST

Traverse a list

linked_list.c

LINKED LIST

Insert at the tail

linked_list.c

REACH OUT

CONTENT RELATED QUESTIONS

Check out the forum

ADMIN QUESTIONS

cs1511@unsw.edu.au