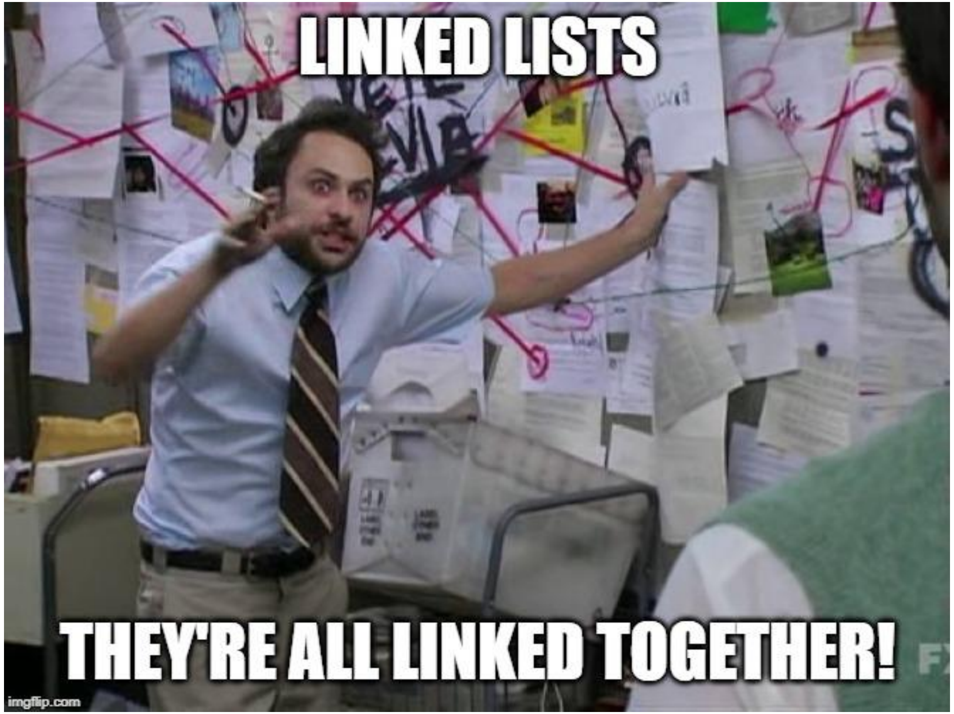
COMP1511 - Programming Fundamentals

Term 1, 2020 - Lecture 15

What did we learn last week?

Linked Lists

- A complete working implementation of Linked Lists
- Inserting nodes
- Removal of nodes
- Cleaning our memory



What are we covering today?

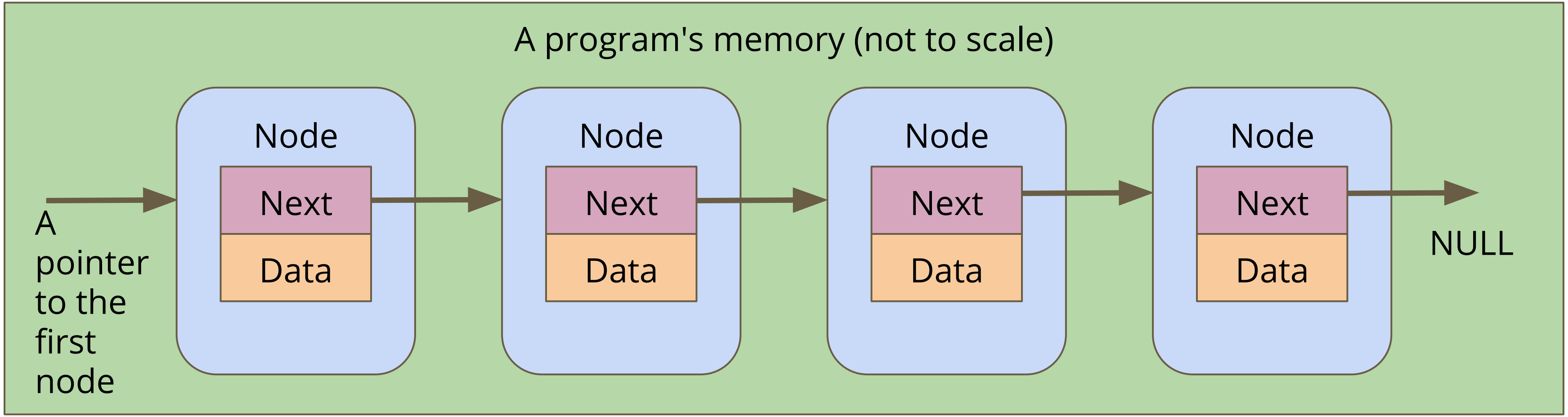
Abstract Data Types

- A recap of Multiple File Projects
- More detail on things like typedef
- The ability to present capabilities of a type to us . . .
- . . . without exposing any of the inner workings

Recap - Linked Lists

Structures with pointers to their own type

- We create Nodes in a Linked List by allocating memory
- We receive a pointer to a Node
- We connect Nodes together by aiming their pointers at each other



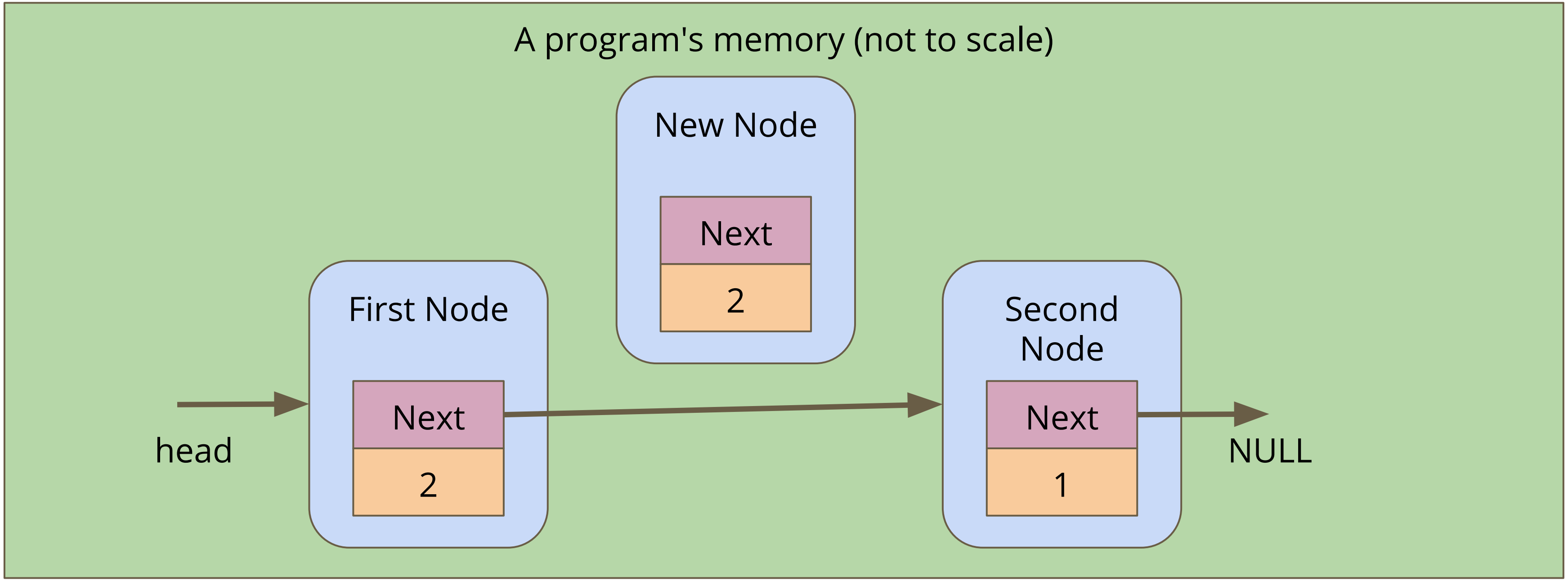
Insertion into Linked Lists

Insertion

- We create a new node
- We find two nodes we want to insert in between
- We aim our new node's next pointer at the second node
- We aim the first node's next pointer at our new node

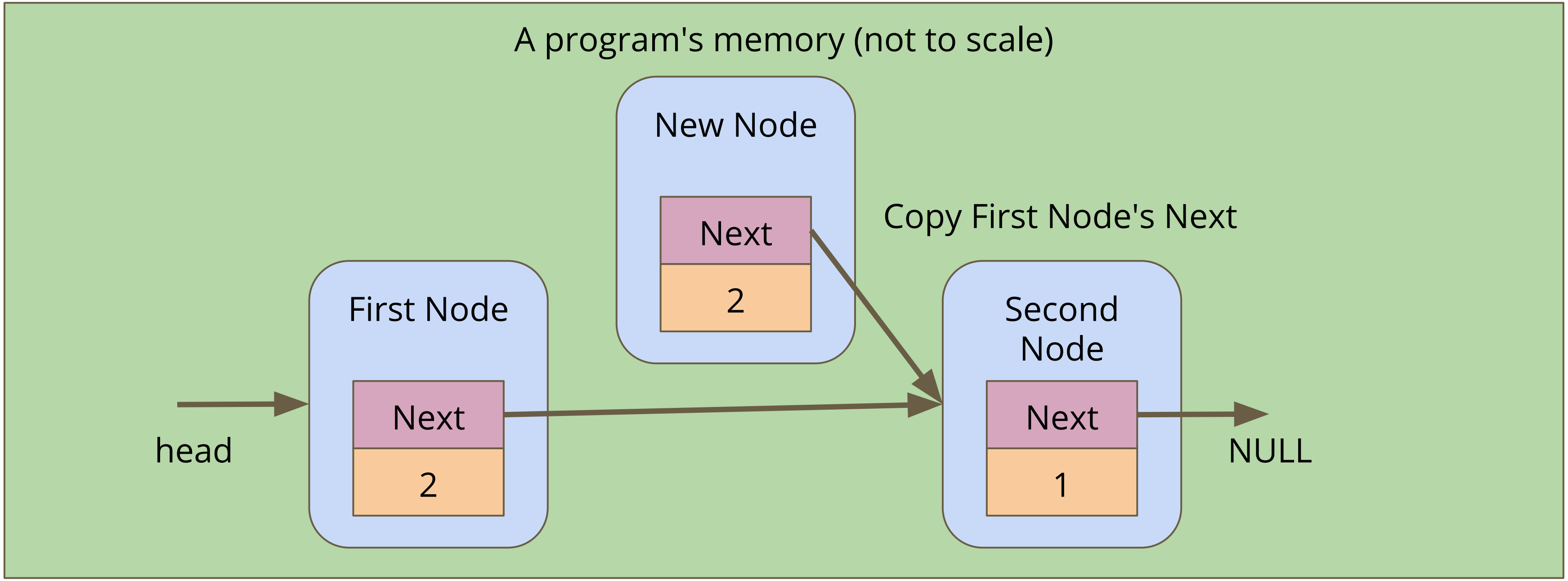
Create a node

A new node is made, it's not connected to anything yet



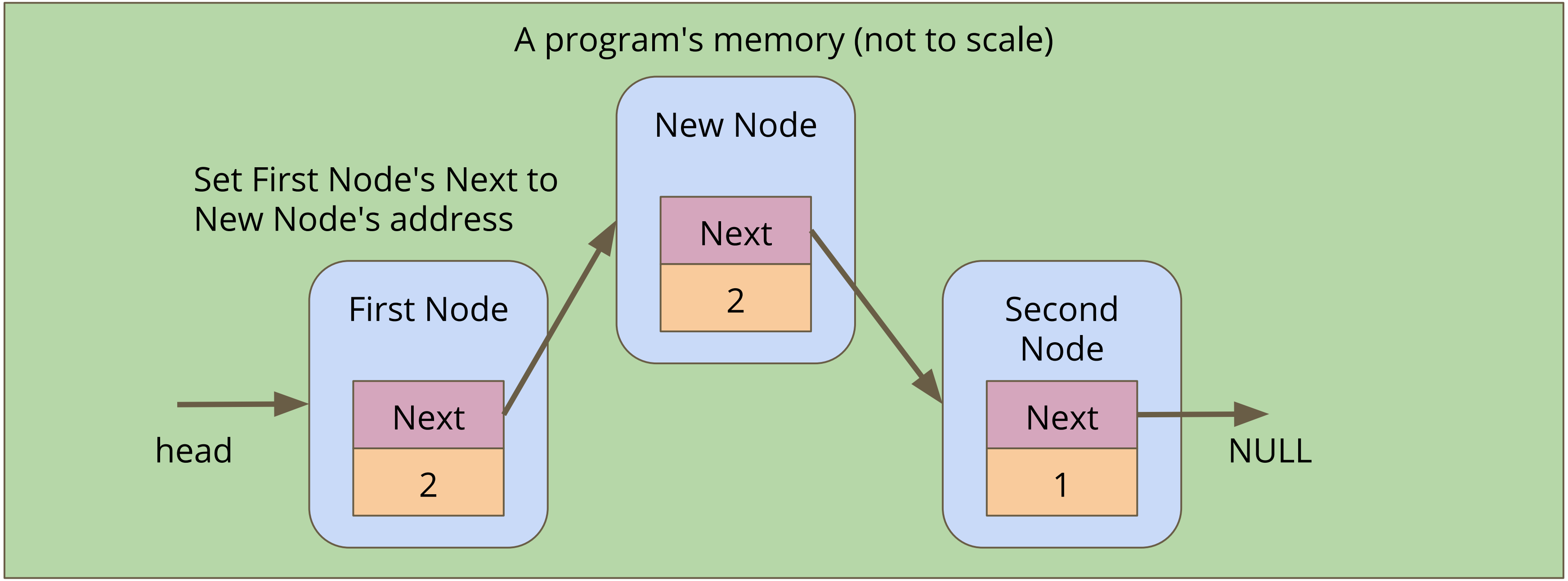
Connect the new node to the second node

Alter the **next** pointer on the New Node



Connect the first node to the new node

Alter the **next** pointer on the First Node



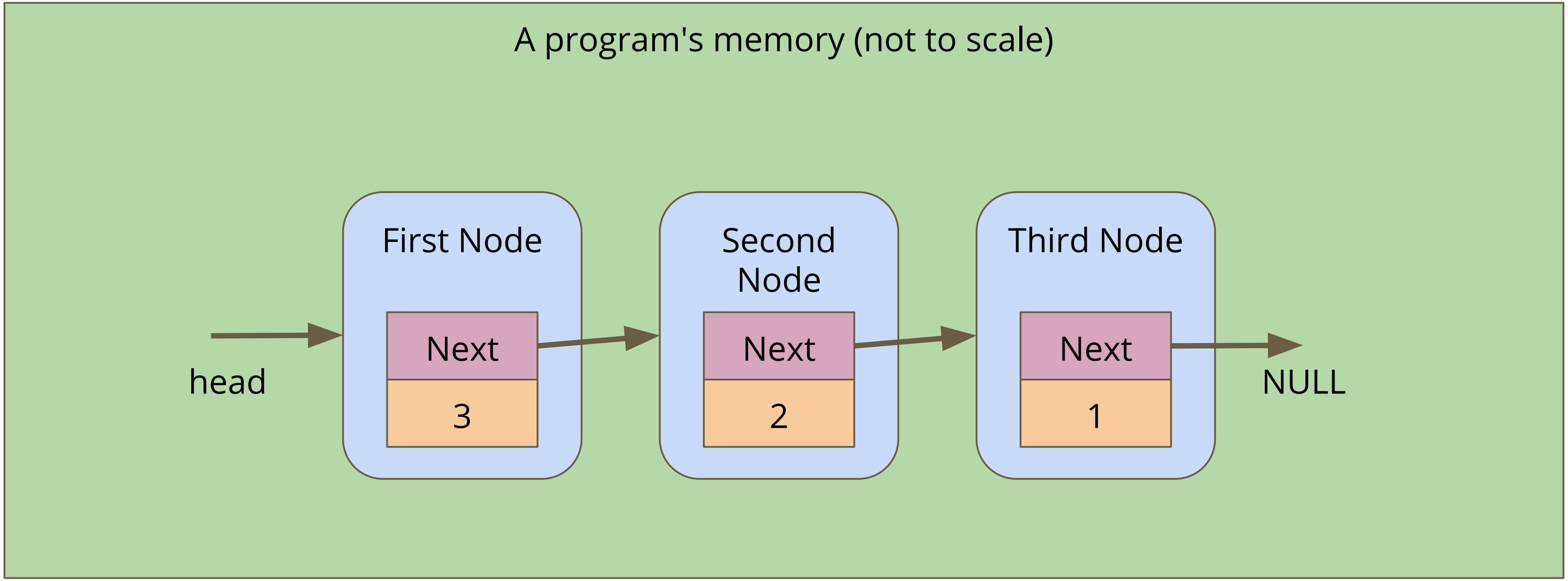
Removal from Linked Lists

Removal

- Find the node you wish to remove
- Bypass it by taking the previous node's next and aiming it at the node after the node you want to remove
- Then free the removed node and all its data

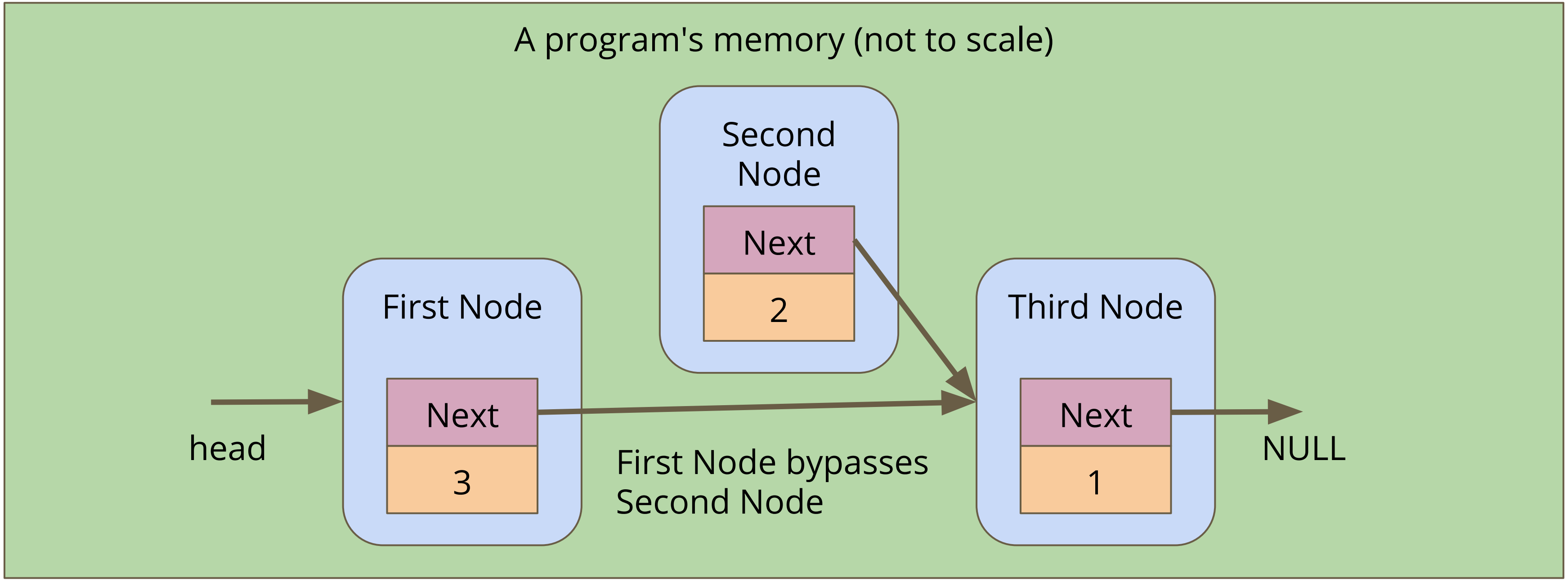
Removing a node

If we want to remove the Second Node



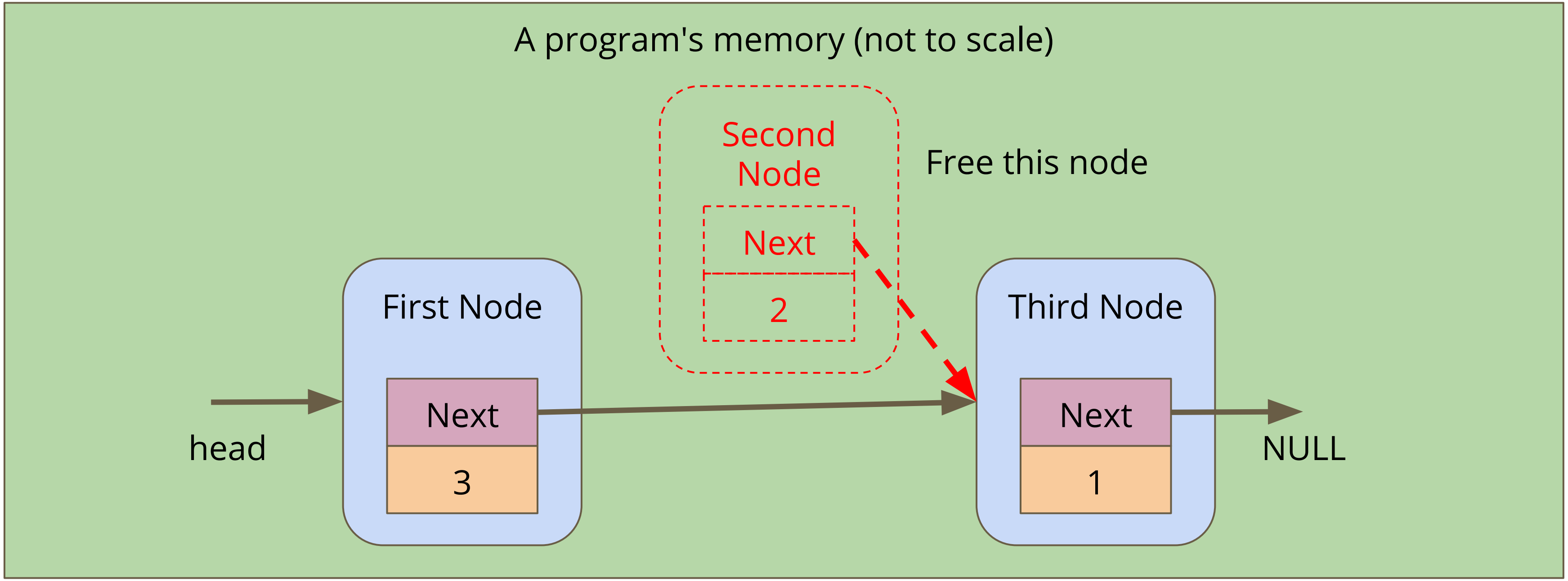
Skipping the node

Alter the First Node's **next** to bypass the node we're removing



Freeing the node

Free the memory from the now bypassed node



Recap - Multiple File Projects

Separating Code into Multiple files

- Header file (*.h) - Function Declarations
- Implementation file (*.c) - Majority of the running code
- Other files - can include a Header to use its capabilities

Separation protects data and makes functionality easier to read

- We don't have access to internal information we don't need
- We can't accidentally change something important
- We have a simple list of functions we can call

Using Multiple Files

Linking the Files

- A file that #includes the Header (*.h) file will have access to its functions
- It's own implementation (*.c) file will always #include it
- Implementation files are never included!

Compilation

- All Implementation files are compiled
- Header files are never compiled, they're included

An Example - A Beat

Assignment 2 - Beats by CSE is a nice example

beats.h

- Contains only defines, typedefs and function declarations
- Is commented heavily so that it's easy to know how to use it

beats.c

- Contains actual structs
- Contains implementation of beats.h's functions (once we've written them)

An Example - A Beat

How some of the other files interact . . .

main.c

- #includes beats.h
- Uses the functions in beats.h

test_beats.c

- #includes beats.h
- Is mutually exclusive with main.c because they both have main functions

Abstract Data Types

Types we can declare for a specific purpose

- We can name them
- We can fix particular ways of interacting with them
- This can protect data from being accessed the wrong way

We can hide the implementation

- Whoever uses our code doesn't need to see how it was made
- They only need to know how to use it

Typedef

Type Definition

- We declare a new Type that we're going to use
- `typedef <original Type> <new Type Name>`
- Allows us to use a simple name for a possibly complex structure
- More importantly, hides the structure details from other parts of the code

```
typedef struct beat *Beat;
```

- We can use **`Beat`** as a Type without knowing anything about the struct underlying it

Typedef in a Header file

The Header file provides an interface to the functionality

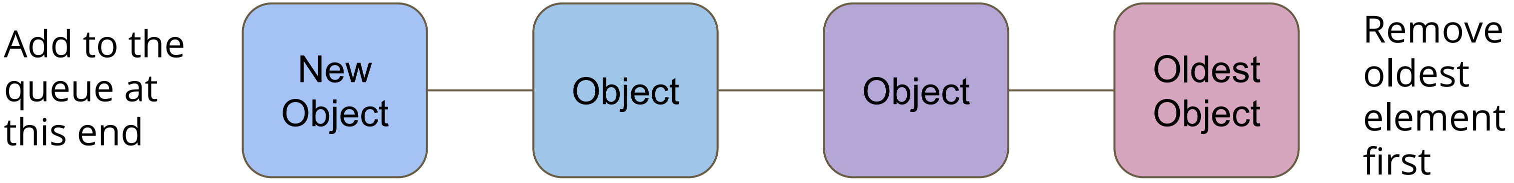
- We can put this in a **header** (*.h) file along with functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the *.c file which is not included directly
- We can also see the functions without knowing how they work

- We are able to see the **header** and use the information
- We hide the **implementation** that we don't need to know about

An Example of an Abstract Data Type - A Queue

What's a queue?

- You should be reasonably familiar with the concept
- In the human world, we sometimes line up for things
- New things join the back of the queue
- Whatever's been there the longest will be the first thing to leave the queue



What makes it Abstract?

A Queue is an idea

- An Array or a Linked List is a very specific implementation
- A Queue is just an idea of how things should be organised
- There's a structure, but there's no implementation!

Abstract Data Type for a Queue

- We can have a header saying how the Queue is used
- The Implementation could use an Array or a Linked List to store the objects in the Queue, but we wouldn't know!

Break Time

Programming Languages

- C++, Java, C# and many others are based on C
- There are too many programming languages to count or learn!

- Remember the fundamentals!
- C syntax is not as important as your plans and thinking
- You will encounter many programming languages, some will feel very different from C in their approach
- But if you learn how you want to communicate with computers, the actual language you use will never be a barrier for you

Let's build a Queue ADT

We're only concerned with how we'll use it, not what it's made of

- Our user will see a "Queue" rather than an Array or Linked List
- We will start with a Queue of integers
- We will provide access to certain functions:
    - Create a Queue
    - Destroy a Queue
    - Add to the Queue
    - Remove from the Queue
    - Count how many things are in the queue

A Header File for Queue

```c
// queue type hides the struct that is is
// implemented as
typedef struct queueInternals *Queue;

// functions to create and destroy queues
Queue queueCreate(void);
void queueFree(Queue q);

// Add and remove items from queues
// Removing the item returns the item for use
void queueAdd(Queue q, int item);
int queueRemove(Queue q);

// Check on the size of the queue
int queueSize(Queue q);
```

What does our Header (not) Provide?

Standard Queue functions are available

- We can join the end or take the element from the front of the Queue
- We are not given access to anything else inside the Queue!
- We cannot take more than one element
- We aren't able to loop through the Queue

The power of Abstract Data Types

- They stop us from accessing the data incorrectly!

Queue.c

Our *.c file is the implementation of the functionality

- The C file is like the detail under the "headings" in the header
- Each declaration in the header is like a title of what is implemented

- Let's start with a Linked List as the underlying data structure
- A Linked List makes sense because we can add to one end and remove from the other
- It also works because it can change length with no issues

The implementation behind a type definition

We can create a pair of structs

- queueInternals represents the whole Queue
- queueNode is a single element of the list

```
// Queue internals holds a pointer to the start of a linked list
struct queueInternals {
    struct queueNode *head;
};

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

Creation of a Queue

If we want our struct to be persistent, we'll allocate memory for it

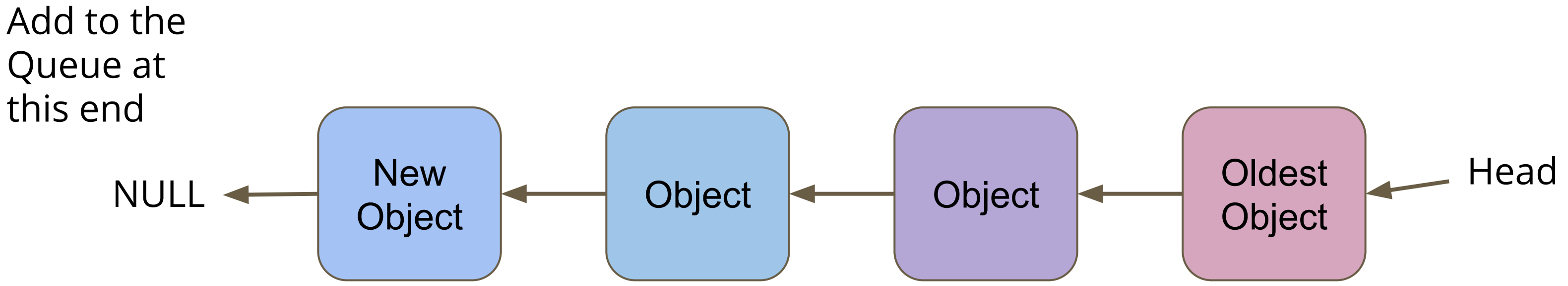
We create our Queue empty, so the pointer to the head is NULL

```
// Create an empty queue
Queue queueCreate(void) {
    Queue newQueue = malloc(sizeof(struct queueInternals));
    newQueue->head = NULL;
    return newQueue;
}
```

Adding items to the Queue

We add items to the end of the Queue

- We need to find the tail end of the Queue
- Then add an element at the end



Add Element at the end

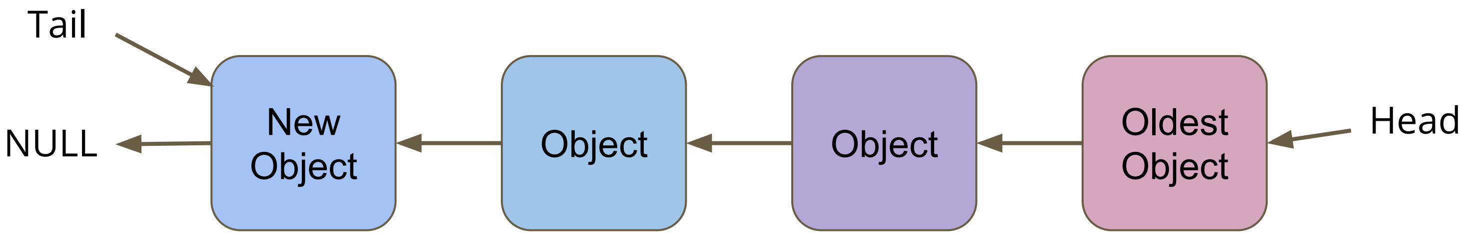
First option for adding an element at the tail end

- Loop through all the elements until the next pointer is NULL
- Add something to the end, pointing the NULL pointer at the new node

- Looping to find the end every time seems like a lot of extra work
- What if we keep track of the last element in the list using our **`queueInternals`** struct?

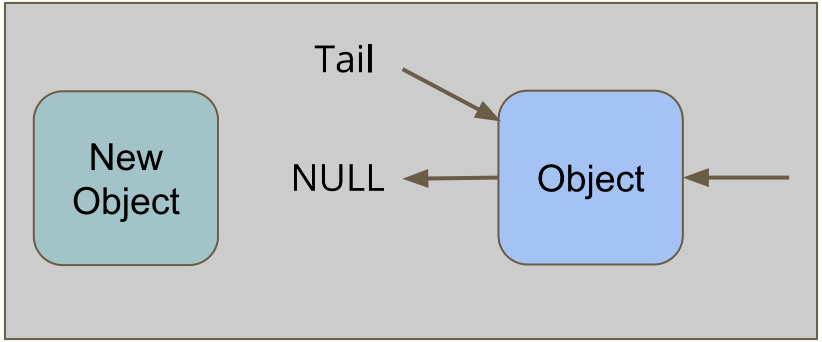
Keeping track of both ends

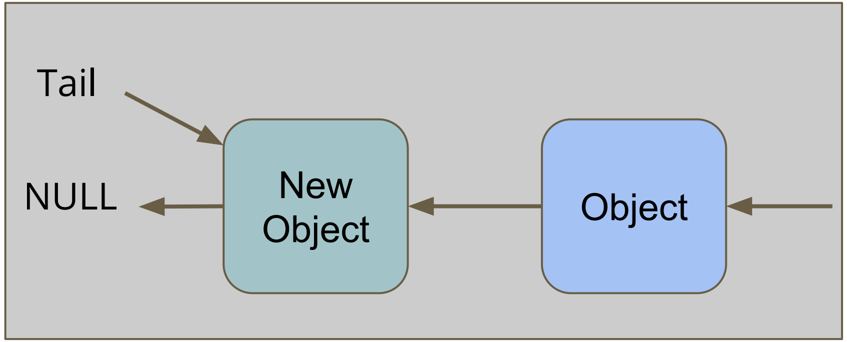
```
// Queue internals holds a pointer to the
// start and end of the linked list
struct queueInternals {
    struct queueNode *head;
    struct queueNode *tail;
};
```



Adding to the tail

- Connect the new object to the current tail
- Move the tail pointer to the new last object
- We no longer need to loop through the whole queue to find the tail





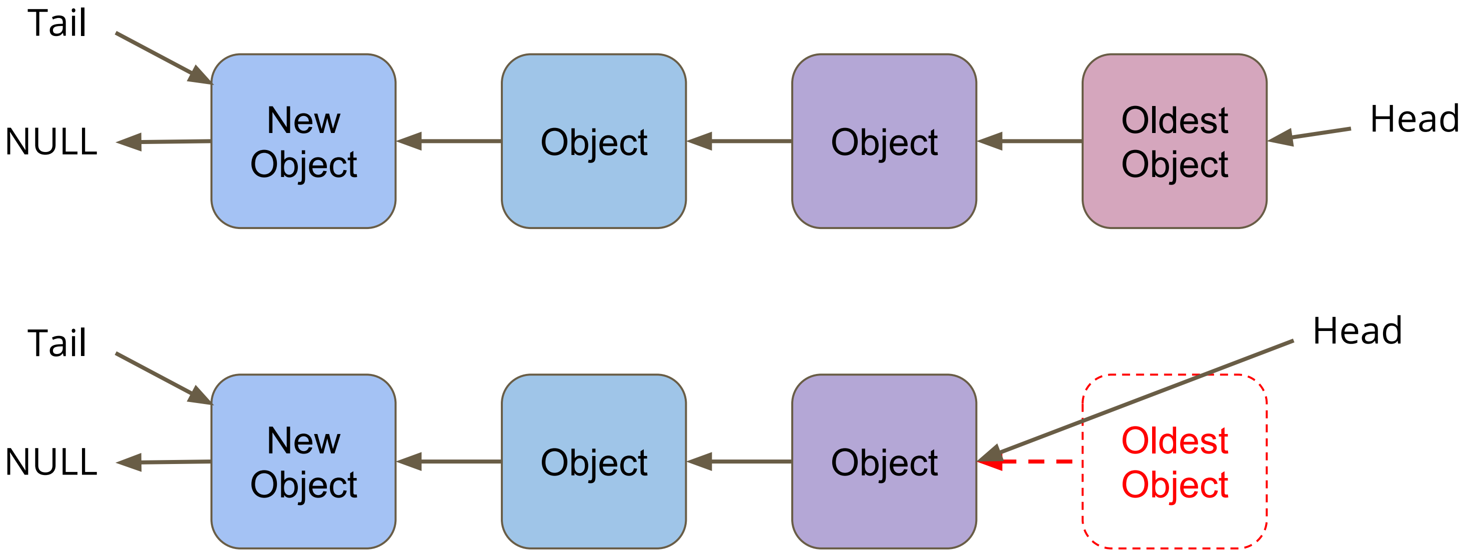
Code for Adding

```
void queueAdd(Queue q, int item) {
    struct queueNode *newNode = malloc(sizeof(struct queueNode));
    newNode->data = item;
    newNode->next = NULL;

    if (q->tail == NULL) {
        // Queue is empty
        q->head = newNode;
        q->tail = newNode;
    } else {
        q->tail->next = newNode;
        q->tail = newNode;
    }
}
```

Removing a Node

The only node that can be removed is the head (the oldest node)



Code for Removing

```
// Remove the head from the list and free the memory used
int queueRemove(Queue q) {
    if (q->head == NULL) {
        printf("Attempt to remove an element from an empty queue.\n");
        exit(1);
    }
    // Keep track of the old head
    int returnData = q->head->data;
    struct queueNode *remNode = q->head;

    // move the queue to the new head and free the old
    q->head = q->head->next;
    free(remNode);

    return returnData;
}
```

Testing Code in our Main.c

```
int main(void) {
    printf("Creating the Queue for Ice Cream.\n");
    Queue iceQueue = queueCreate();
    int id = 1;
    printf("Person %d joins the queue!\n", id);
    queueAdd(iceQueue, id);
    id = 2;
    printf("Person %d joins the queue!\n", id);
    queueAdd(iceQueue, id);
    id = 3;
    printf("Person %d joins the queue!\n", id);
    queueAdd(iceQueue, id);

    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
    printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
    return 0;
}
```

Other Functionality

There are some functions in the header we haven't implemented

- **Destroying and freeing the Queue**
- We're still at risk of leaking memory because we're only freeing on removal
- **Display the Number of Elements**
- This would be very handy because it would allow us to tell how many elements we can remove before we risk errors

We'll finish these and look at more tomorrow!

What did we cover today?

Abstract Data Types

- Makes use of Multi-file projects we discussed earlier
- typedef to protect a struct from open access
- Using multiple files to control how a type is used
- Hiding the implementation
- Providing a fixed interface
- Our demo is a partly implemented Queue