COMP1511 - Programming Fundamentals

Term 3, 2019 - Lecture 15

What did we learn last week?

Memory

- Allocating memory for use beyond the scope of functions

Multiple File Projects

- C files, H files and including our own files

Linked Lists

- structs, pointers and malloc all together!

What are we learning today?

Linked Lists

- Continuing our work from last week
- Insertion and Removal from Linked Lists
- Cleaning up memory from a linked list
- Finishing our Battle Royale example

Recap - Memory Allocation

Keeping variables available to us

- `malloc()` and `sizeof()` and also `free()`
- These functions allow us to declare variables as pieces of memory
- The memory is allocated to us and we access it via a pointer
- Unlike normal variables, these are never cleaned up by `{`curly brackets`}`
- We can keep using them until we don't need them
- We can `free()` them explicitly

Pointers and Structs

struct node *

A pointer that can aim at memory

A memory address that can aim at a node

struct node

A piece of memory that can store a node

Node in memory

Usually allocated by malloc()

```
struct node *n =
malloc(sizeof(struct node));
```

Node in memory

Recap - Multi-file Projects

Separating our code

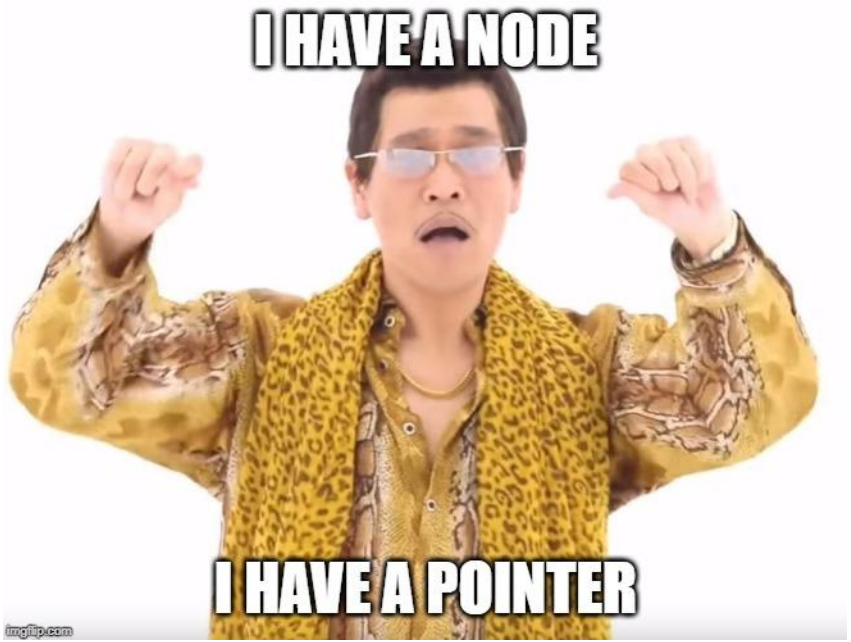
- Header files (*.h)
- Contain function declarations, but no running code
- Implementation files (*.c)
- Contain implementations of the Header's functions
- We `#include` header files
- This means the file that includes a header can't see the implementation, it trusts that it will be given certain functionality

Recap - Linked Lists

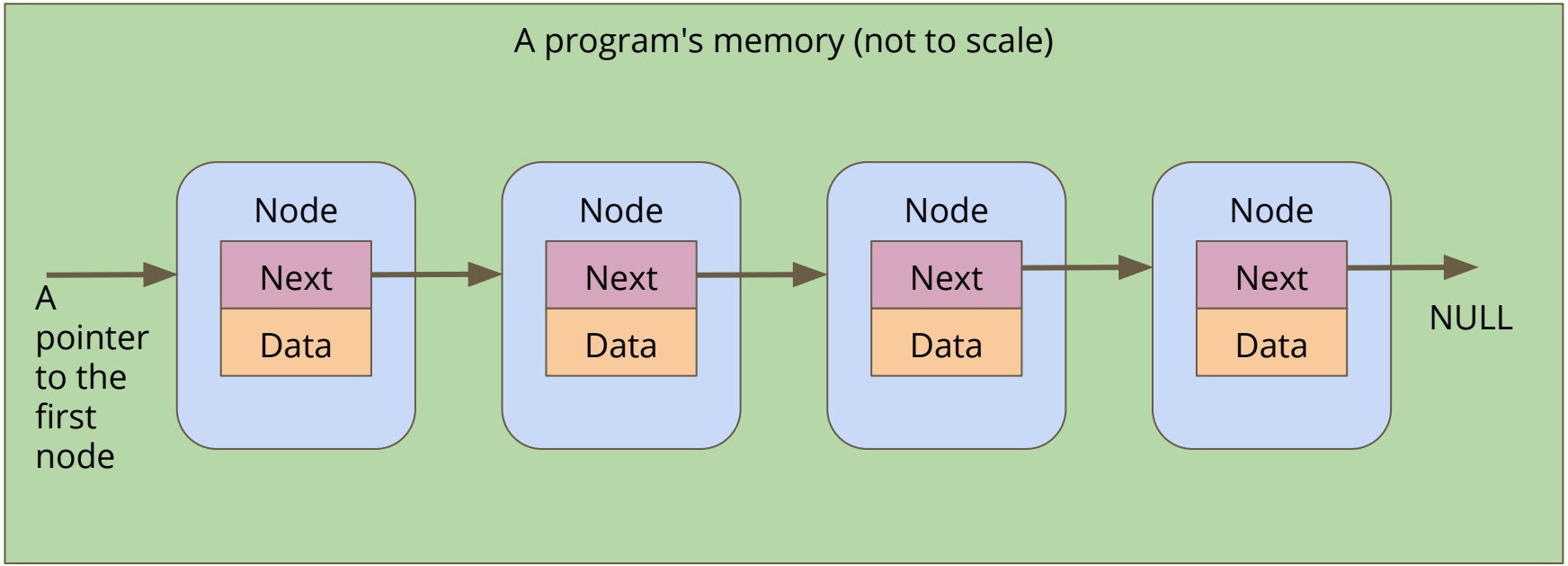
A chain of identical structs to hold information

- Pointers to the same type of struct so they can be chained together
- Some kind of information stored in the struct

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



A Linked List



Looping through a Linked List

Loop by using the next pointer

- We can jump to the next node by following the current node's next pointer
- We know we're at the end if the next pointer is NULL

```
// Loop through a list of nodes, printing out their data
void printData(struct node *n) {
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

Battle Royale - our unfinished example

Using a Linked List to track the players in a game

- We started by adding players to the game
- We were able to loop through and print the players in the game
- We might want to control the order of the list, so we need to be able to insert at a particular position
- We also want to be able to find and remove players from the list if they're knocked out of the round

Where are we up to?

What functionality do we have so far?

- createPlayer() makes a node
- printPlayers() can loop through a list and print it out
- We have code in our main that can build a list in a specific order
- What if we want an ordered list?

Inserting Nodes into a Linked List

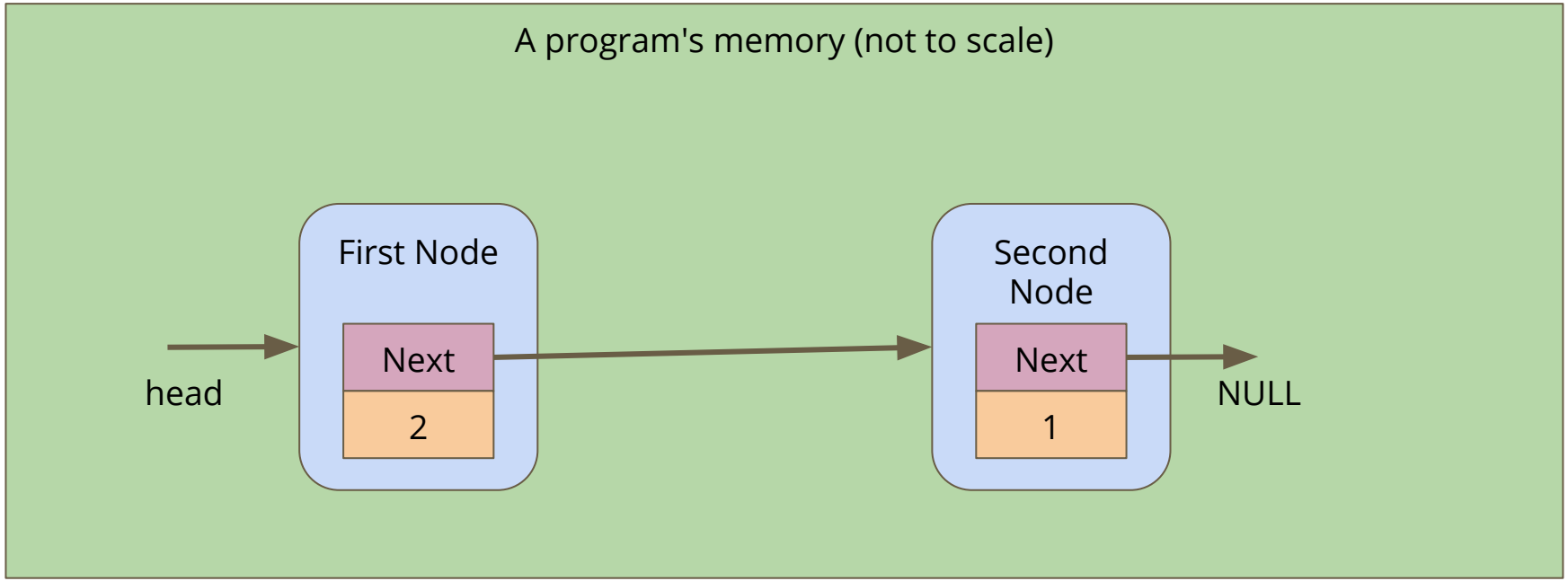
Linked Lists allow you to insert nodes in between other nodes

- We can do this by simply aiming next pointers to the right places
- We find two linked nodes that we want to put a node between
- We take the **next** of the first node and point it at our new node
- We take the **next** of the new node and point it at the second node

This is much less complicated with diagrams . . .

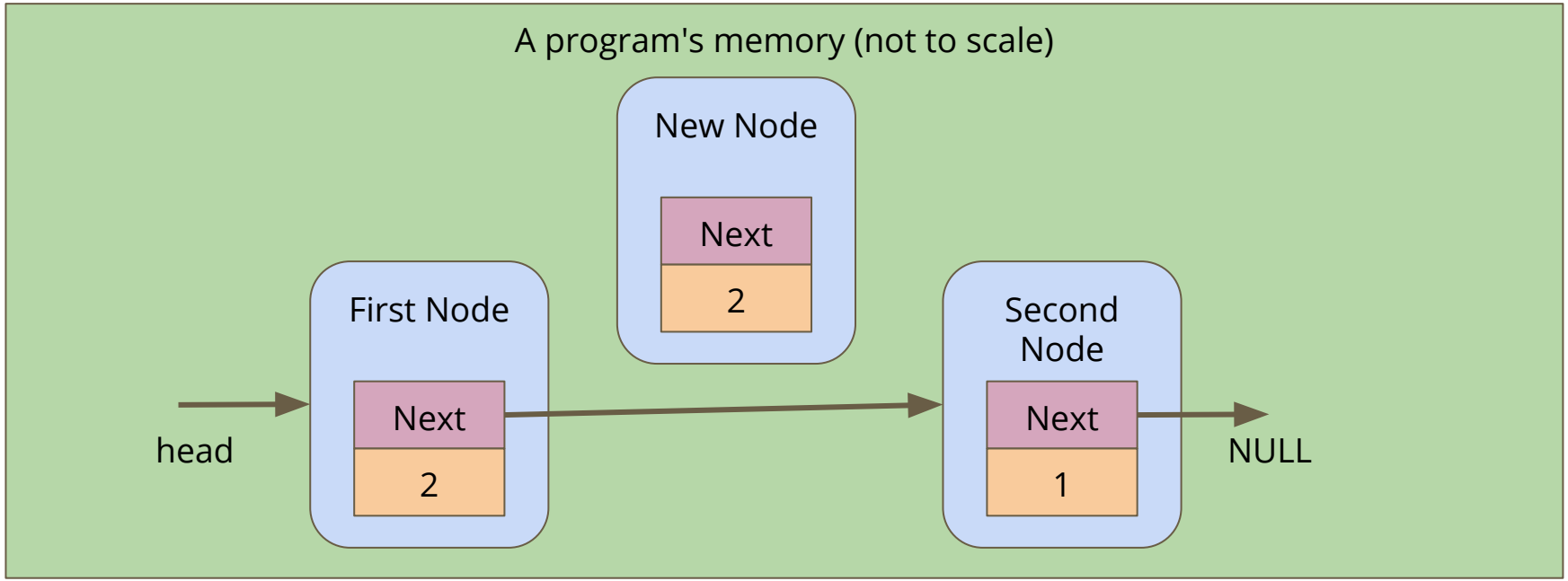
Our Linked List

Before we've tried to insert anything



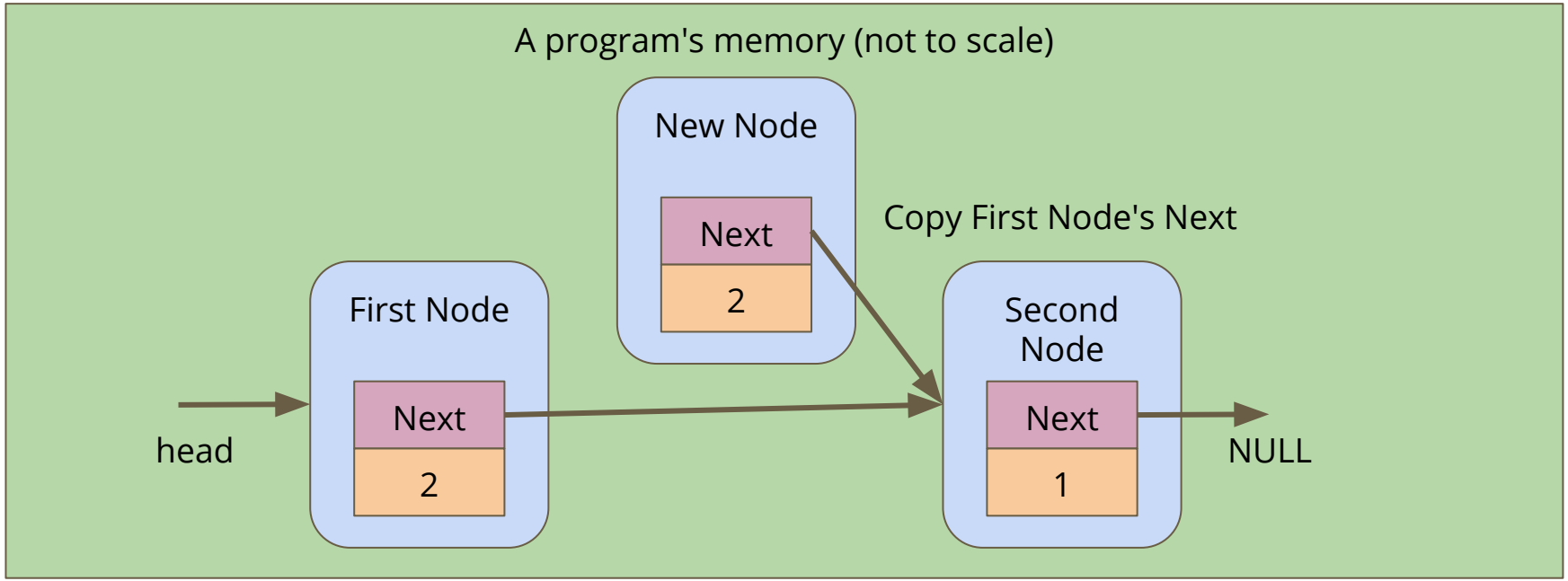
Create a node

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



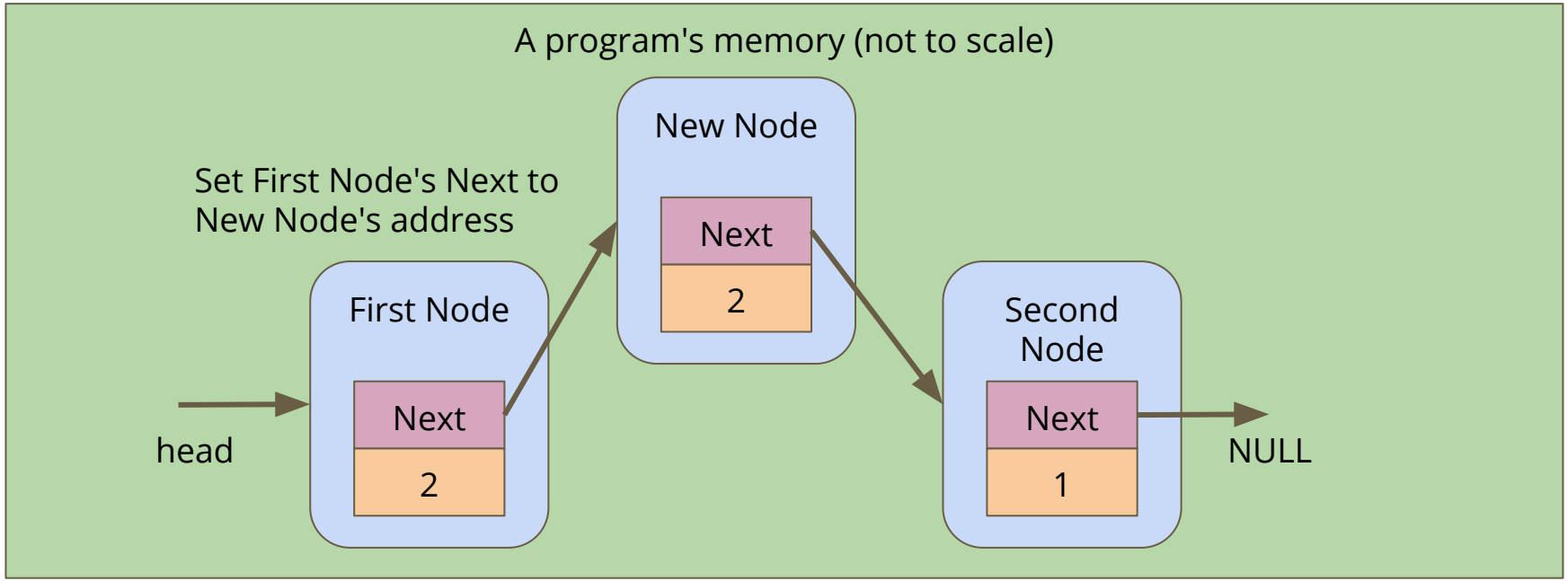
Connect the new node to the second node

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



Connect the first node to the new node

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



Code for insertion

```
// Create and insert a new node into a list after a given listNode
struct node *insert(struct node* listNode, char newName[]) {
    struct node *n = createNode(newName, NULL);
    if (listNode == NULL) {
        // List is empty, n becomes the only element in the list
        listNode = n;
        n->next = NULL;
    } else {
        n->next = listNode->next;
        listNode->next = n;
    }
    return listNode;
}
```

Inserting Nodes

We can use insertion to have greater control of where nodes are put in a list

```
int main(void) {
    // create the list of players
    struct node *head = createNode("Marc", NULL);
    insert("AndrewB", head);
    insert("Tom", head);
    insert("Aang", head);
    insert("Sokka", head);

    printPlayers(head);

    return 0;
}
```

Break Time

Homework - it's not real homework, just things that can inspire you

- *AlphaGo* Documentary (on Netflix)
- *I, Robot* Short Stories (Isaac Asimov)
- *Snow Crash* and *The Cryptonomicon* Novels (Neal Stephenson)
- *Human Resource Machine* Game (on Steam, iOS and Android)
- *Space Alert* Board Game (Vlaada Chvatil)

Insertion with some conditions

We can now insert into any position in a Linked List

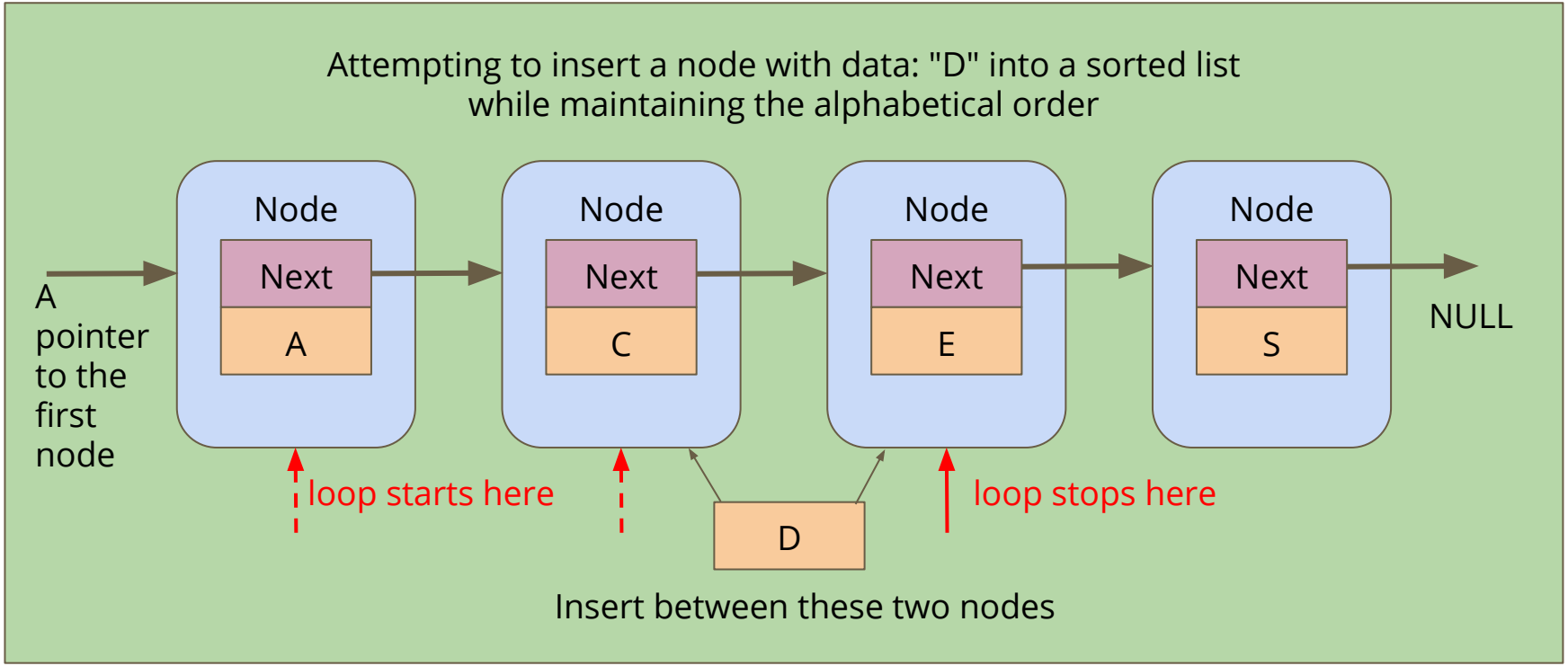
- We can read the data in a node and decide whether we want to insert before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to use a **`string.h`** function, **`strcmp()`** for this
- **`strcmp()`** compares two strings, and returns
  - 0 if they're equal
  - negative if the first has a lower ascii value than the second
  - positive if the first has a higher ascii value than the second

Finding a where to insert

We're going to loop through the list

- This loop assumes the list is already in alphabetical order
- Each time we loop, we're going to keep track of where we've been
- We'll test the name of each node using `strcmp()`
- We stop looping once we find the first name that's "higher" than ours
- Then we insert before that node

Finding the insertion point



Inserting into a list Alphabetically

```
struct node *insertAlphabetical(char newName[], struct node* head) {
    struct node *previous = NULL;
    struct node *n = head;
    // Loop through the list and find the right place for the new name
    while (n != NULL && strcmp(newName, n->name) > 0) {
        previous = n;
        n = n->next;
    }
    struct node *insertionPoint = insert(newName, previous);
    if (previous == NULL) {
        // we inserted at the start of the list
        insertionPoint->next = n;
        return insertionPoint;
    } else {
        return head;
    }
}
```

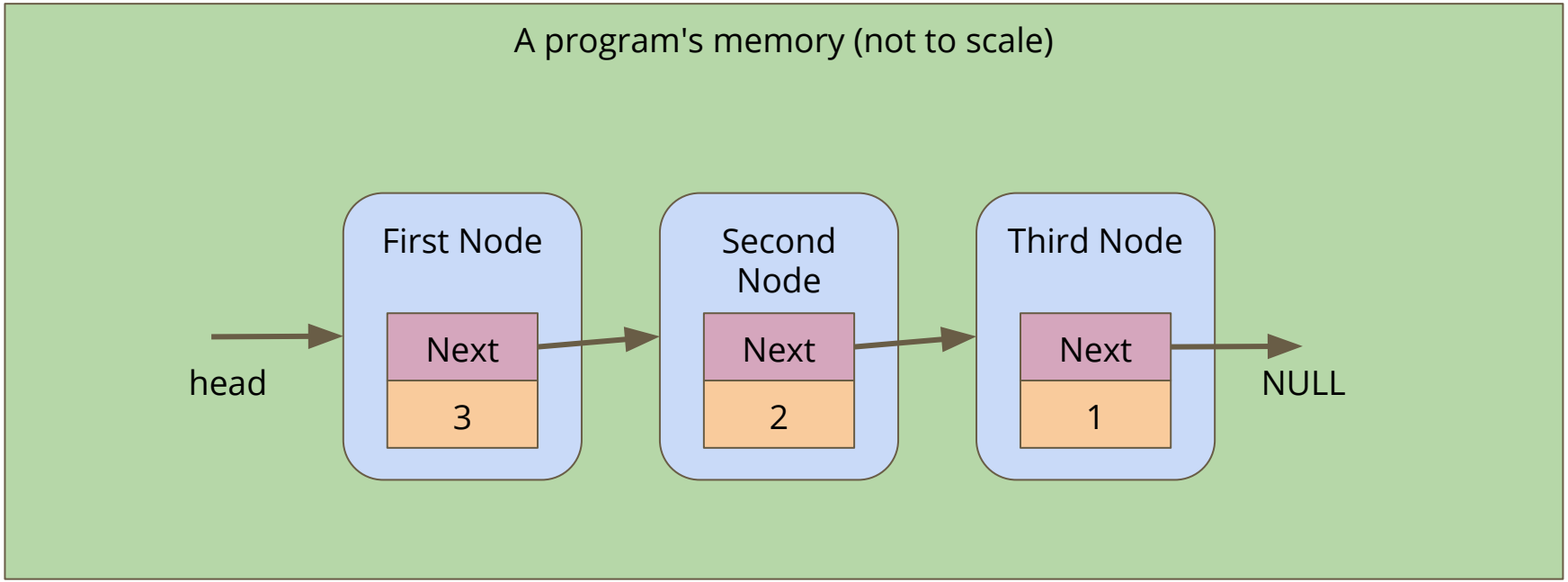
Removing a node

If we want to remove a specific node

- We need to look through the list and see if a node matches the one we want to remove
- To remove, we'll use **next** pointers to connect the list around the node
- Then, we'll free the node itself that we don't need anymore

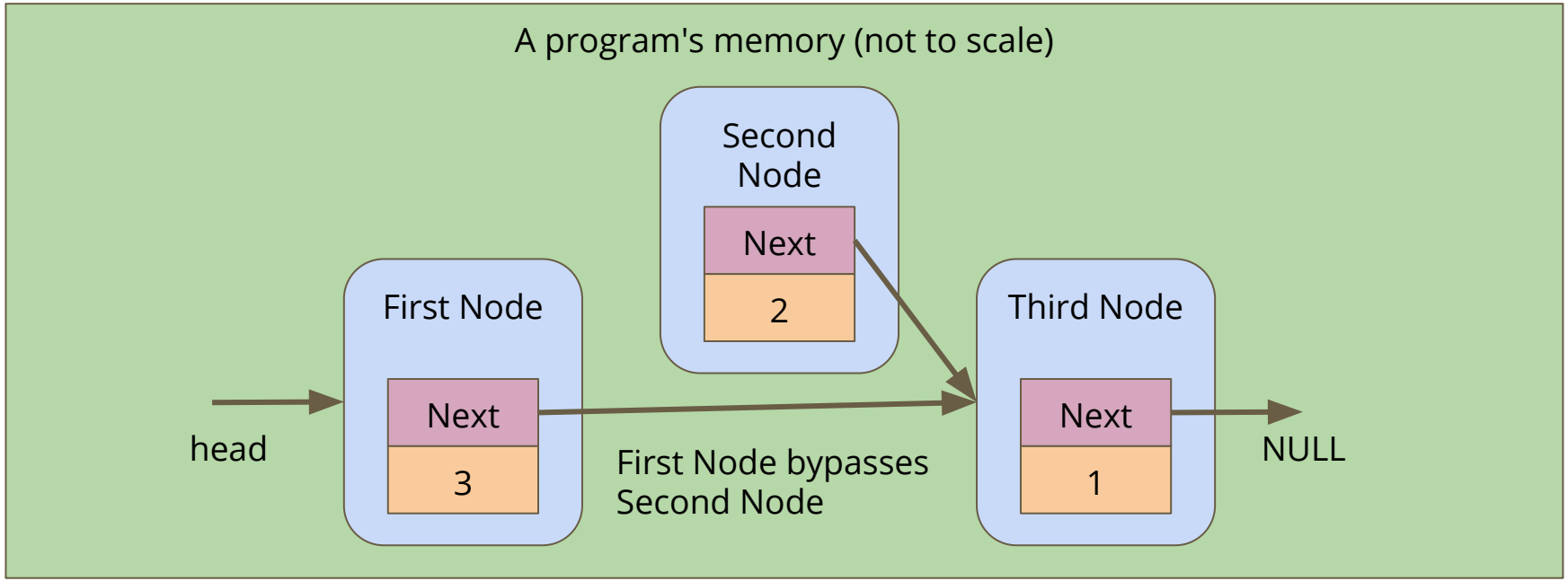
Removing a node

If we want to remove the Second Node



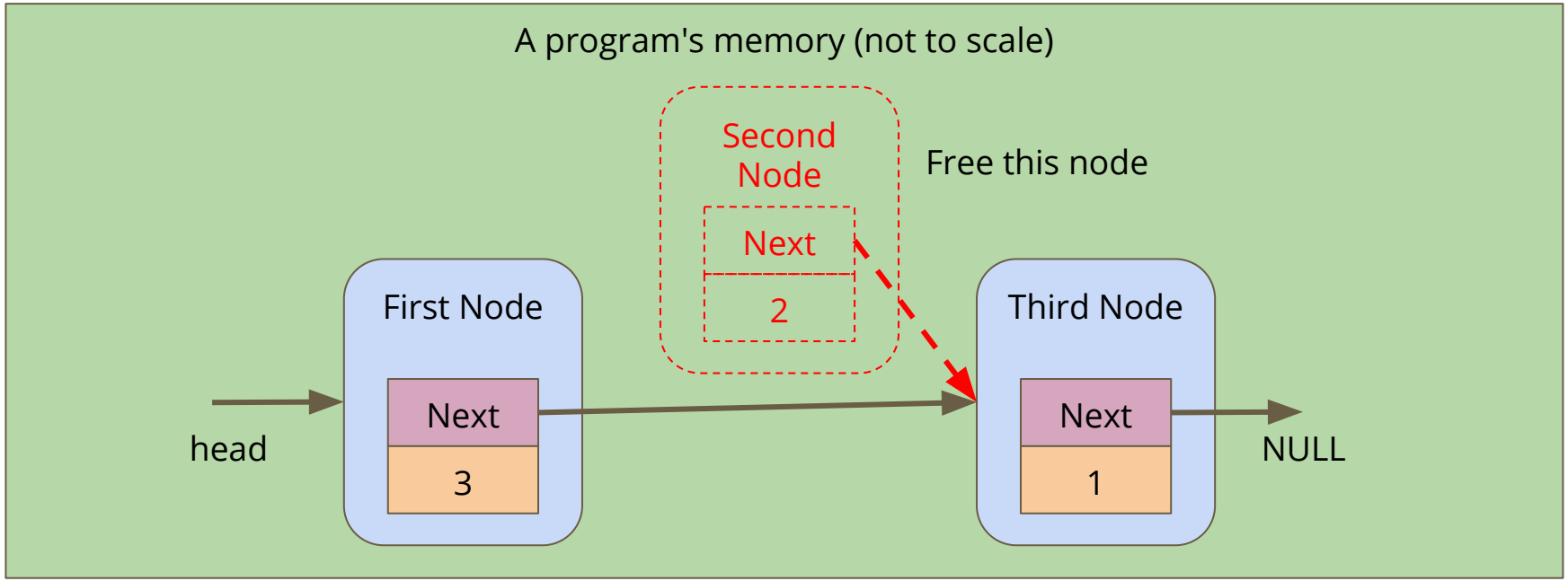
Skipping the node

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



Freeing the node

Free the memory from the now bypassed node



Finding the node

Loop until you find the right match

```
struct node *removeNode(char name[], struct node* head) {
    struct node *previous = NULL;
    struct node *n = head;
    // Keep looping until we find the matching name
    while (n != NULL && strcmp(name, n->name) != 0) {
        previous = n;
        n = n->next;
    }
    if (n != NULL) {
        // if n isn't NULL, we found the right node
```

Removing the node

Having found the node, remove it from the list

```
if (n != NULL) {
        // if n isn't NULL, we found the right node
        if (previous == NULL) {
            // it's the first node
            head = n->next;
        } else {
            previous->next = n->next;
        }
        free(n);
    }
    return head;
}
```

The Battle Royale Game

In a Battle Royale, people are removed from the game one at a time until only one person is left. They are the winner

- We can create a list of players
- We can make sure it's in a nice alphabetical order
- We can remove a single player from the list
- Now we need to remove players one at a time
- When there's only one left, they are the winner!

Game code

Once our list is created, we can loop through the game

- We print out the player list (we might want to modify that function!)
- Our user will tell us who was knocked out

```
// A game loop that runs until only one player is left
while (printPlayers(head) > 1) {
    printf("Who just got knocked out?\n");
    char koName[MAX_NAME_LENGTH];
    fgets(koName, MAX_NAME_LENGTH, stdin);
    koName[strlen(koName) - 1] = '\0';
    head = removeNode(koName, head);
    printf("----------\n");
}
printf("The winner is: %s\n", head->name);
```

What did we learn today?

Linked Lists

- Inserting nodes at a specific location
- Inserting nodes into an ordered list
- Finding nodes using a while loop
- Removing nodes