COMP1511 - Programming Fundamentals

Term 3, 2020 - Lecture 14

What did we learn yesterday?

Linked Lists

- Some Revision
- Building Lists
- Insertion into Lists

What are we doing today?

More Linked Lists

- Finding something in a list
- Insertion to keep a list alphabetical
- Linked List Removal
- Freeing our Allocated Memory
- Playing the game

Insertion with some conditions

We can now insert into any position in a Linked List

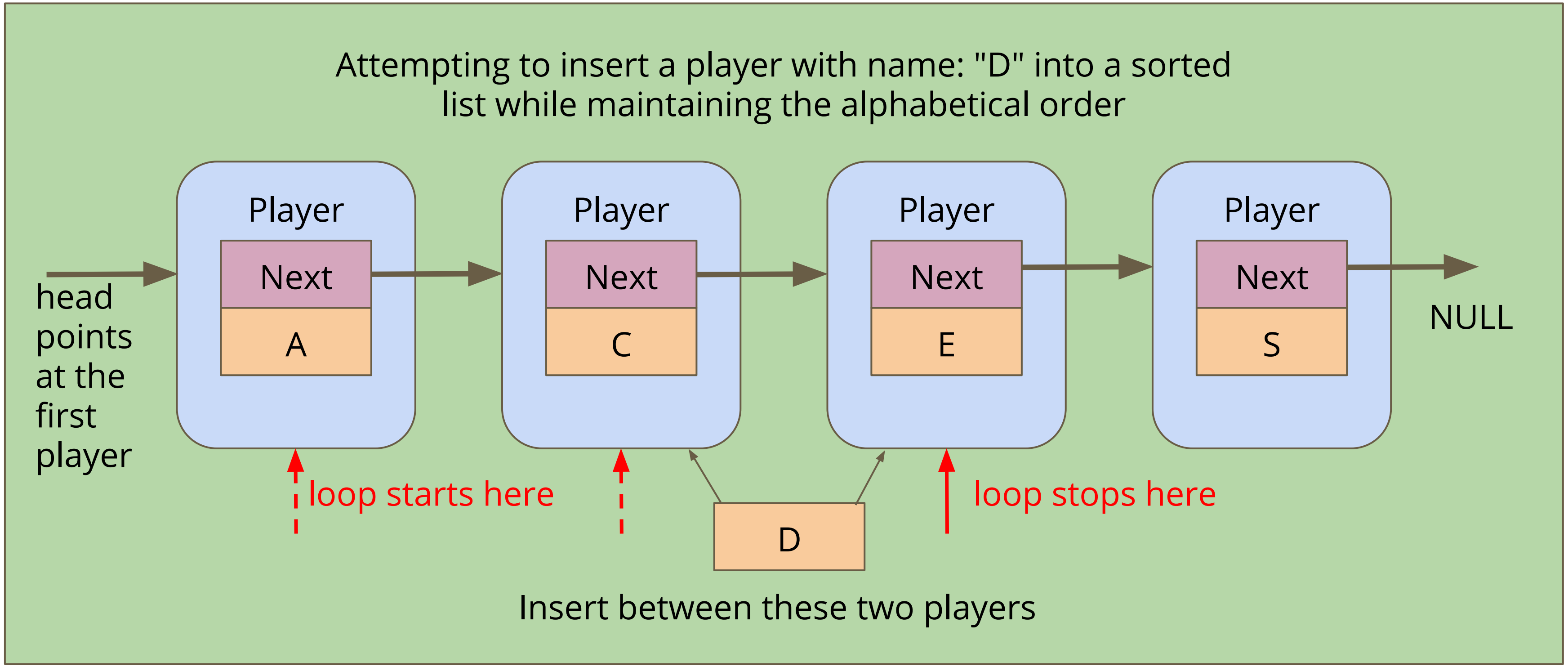
- We can read the data in a node and decide whether we want to insert before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to use a **`string.h`** function, **`strcmp()`** for this
- **`strcmp()`** compares two strings, and returns
  - 0 if they're equal
  - 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 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 the previous player
- We'll test the name of each player using **`strcmp()`**
- We stop looping once we find the first name that's "higher" than ours
- Then we insert before that player

Finding the insertion point


Attempting to insert a player with name: "D" into a sorted list while maintaining the alphabetical order
head points at the first player
Player
Next
A
Player
Next
C
Player
Next
E
Player
Next
S
NULL
loop starts here
D
loop stops here
Insert between these two players

Inserting into a list Alphabetically

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

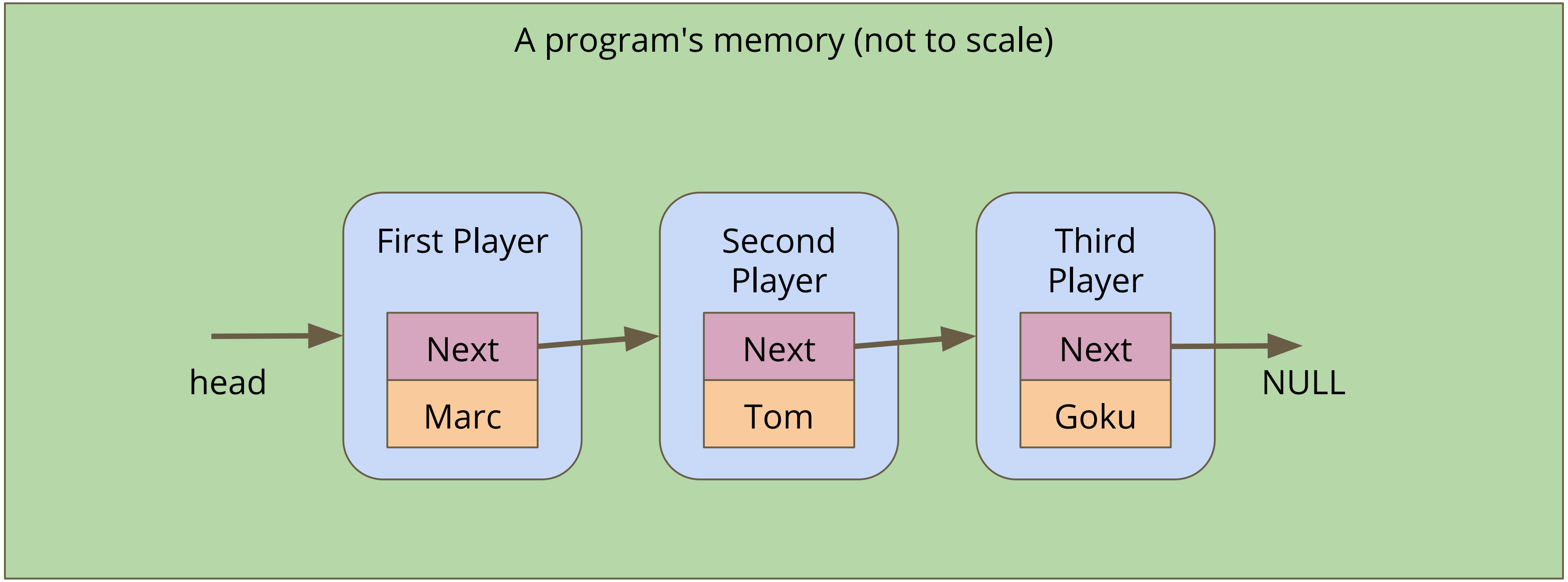
Removing a player

If we want to remove a specific player

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

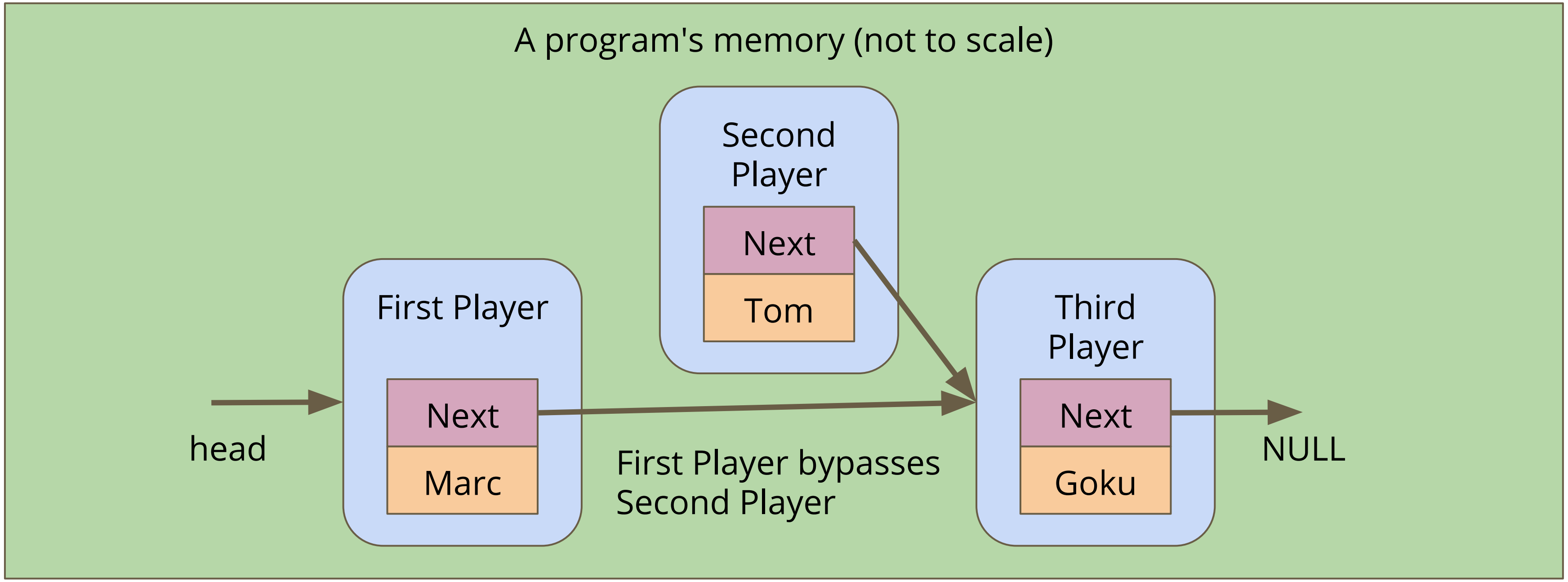
Removing a player node

If we want to remove the Second Player



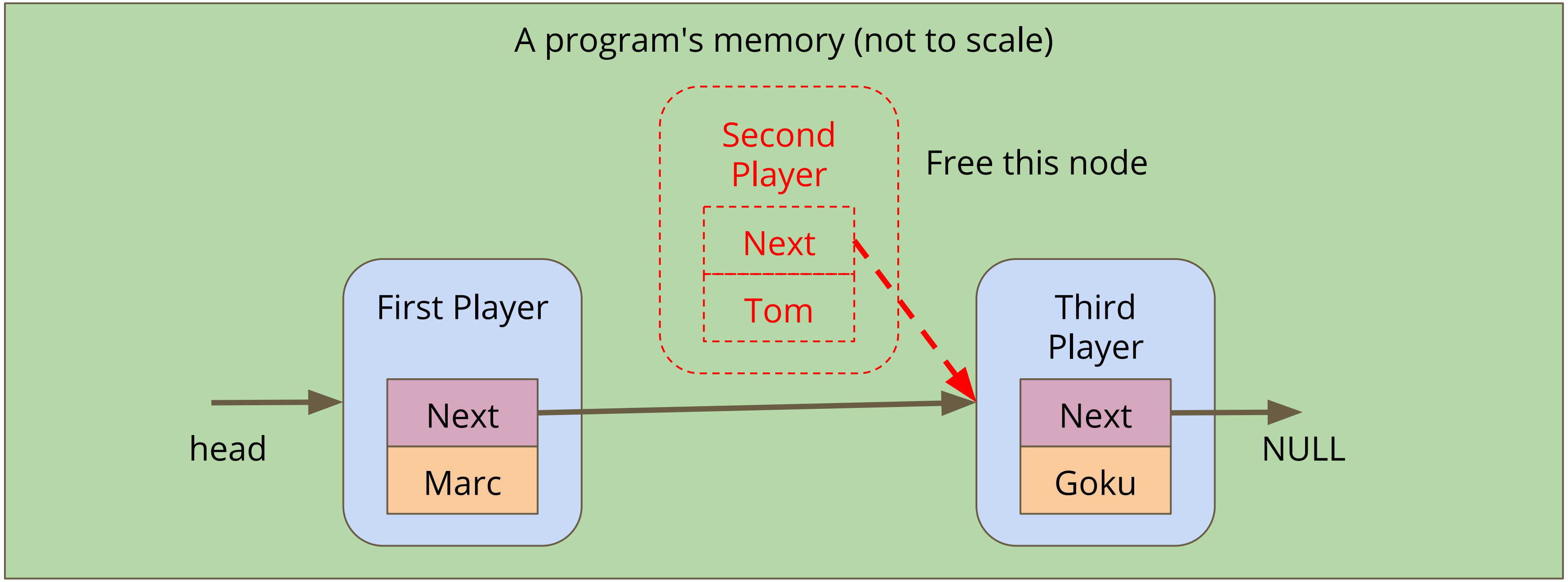
Skipping the player node

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


A program's memory (not to scale)
Second Player
Next
Tom
First Player
Next
Marc
Third Player
Next
Goku
head
NULL
First Player bypasses Second Player

Freeing the removed node

Free the memory from the now bypassed player node



Break Time

Keeping track of your own code projects

- Using **git** is a really handy way to keep backups of your work
- GitHub and BitBucket are two providers that will give you free online repositories to store your code
- Graphical Interfaces are available for git (GitHub Desktop and Sourcetree respectively)
- It takes some time to get familiar with how these work . . . but you can start practicing now!

Finding the right player

Loop until you find the right match

This is very similar to finding the insertion point earlier

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

Removing a player

Having found the player node, remove it from the list

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

The Battle Royale

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 = removePlayer(koName, head);
        printf("----------\n");
    }
    printf("The winner is: %s\n", head->name);
```

Cleaning Up

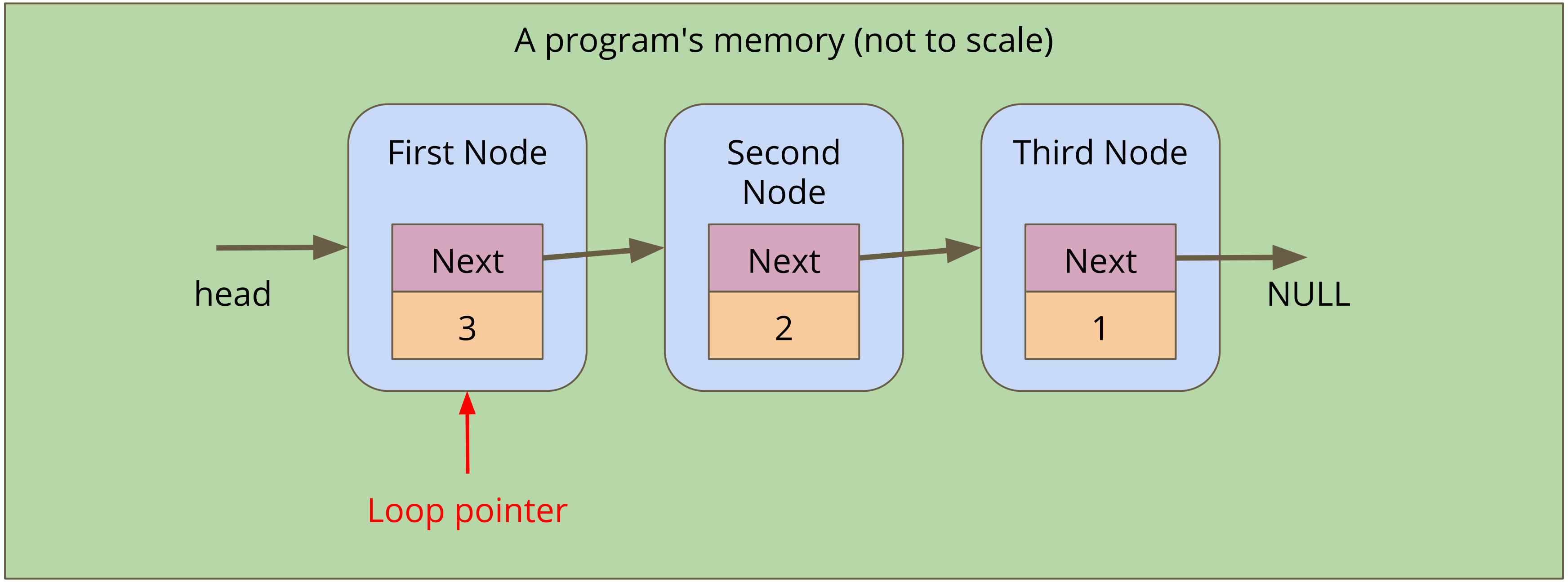
Remember, All memory allocated (malloc) needs to be freed

- We can run `dcc --leak-check` to see whether there's leaking memory
- What do we find?
- There are pieces of memory we've allocated that we're not freeing!

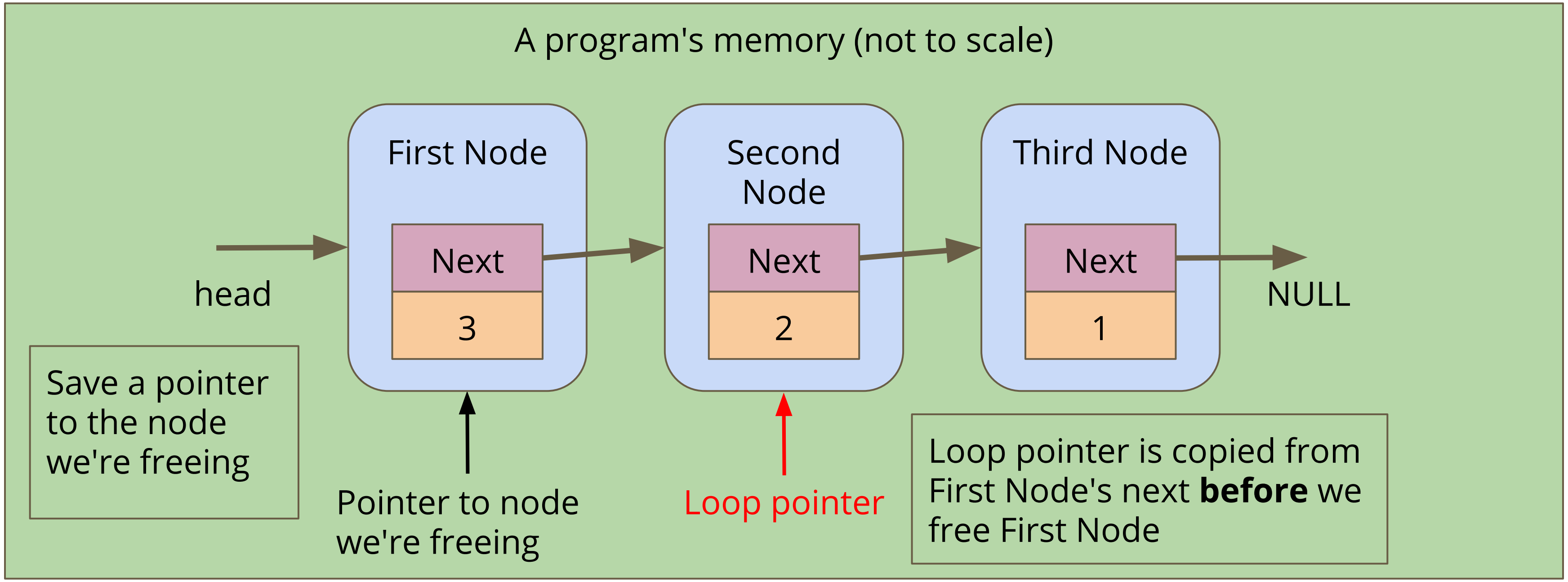
Let's write a function that frees a whole linked list

- Loop through the list, freeing the nodes
- Just be careful not to free one that we still need the pointer from!

Looping to free nodes



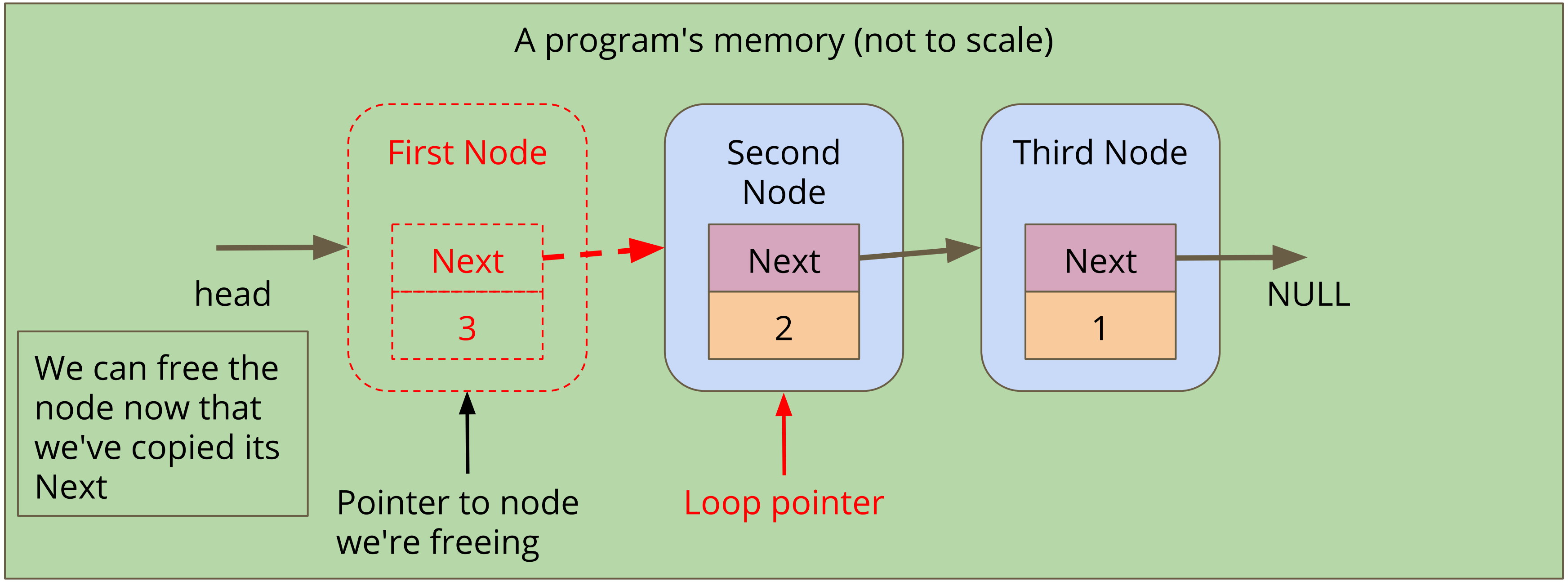
Looping to free nodes



Save a pointer to the node we're freeing

Loop pointer is copied from First Node's next **before** we free First Node

Looping to free nodes


A program's memory (not to scale)
head
First Node
Next
3
Second Node
Next
2
Third Node
Next
1
NULL
We can free the node now that we've copied its Next
Pointer to node we're freeing
Loop pointer

Code to free a linked list

```c
// Loop through a list and free all the allocated memory
void freeList(struct node *n) {
    while(n != NULL) {
        // keep track of the current node
        struct node *remNode = head;

        // move the looping pointer to the next node
        n = n->next;

        // free the current node
        free(remNode);
    }
}
```

Battle Royale, the Linked Lists demo

What have we written in this program?

- Creation of nodes
- Looping through a list
- Insertion of nodes into specific locations
- Finding locations using loops
- Removal of nodes
- Managing memory

What did we cover today?

Linked Lists

- Finding a particular node for insertion
- Removal
- Removing a specific node
- Memory cleaning