COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 16
Stream B

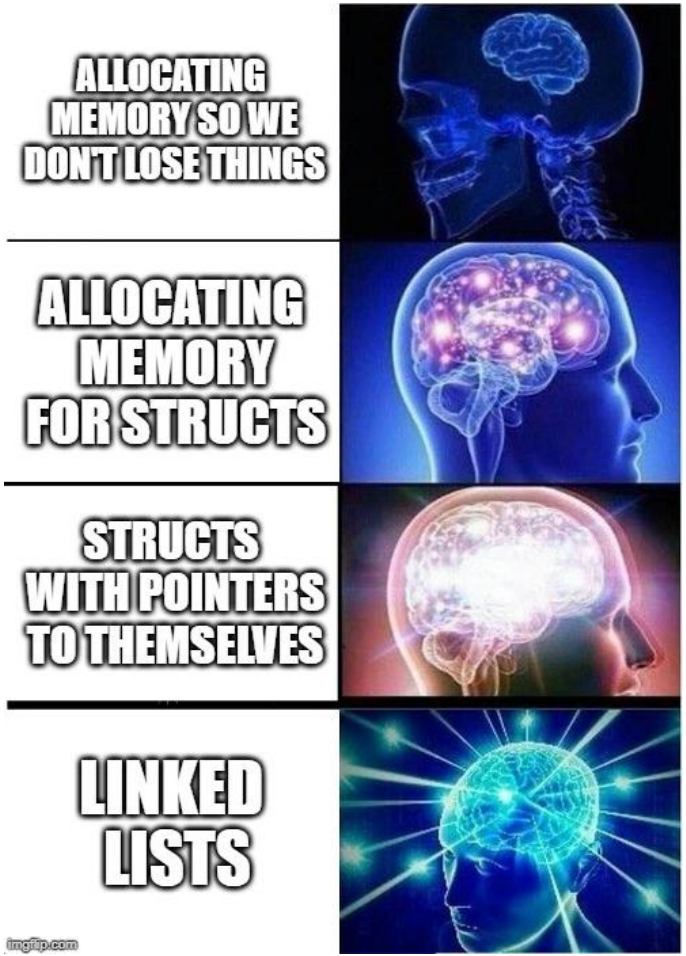
What did we cover on Tuesday?

Memory and Functions

- A brief look at how functions use memory
- Allocating and freeing memory

Linked Lists

- Node structs that hold pointers to other nodes
- Building chains of nodes called Linked Lists



What are we covering today?

Linked Lists

- A recap of how they work
- More complicated nodes
- Looking through a list . . . how do we loop through one?
- Inserting items into a list
- Removing items from a list

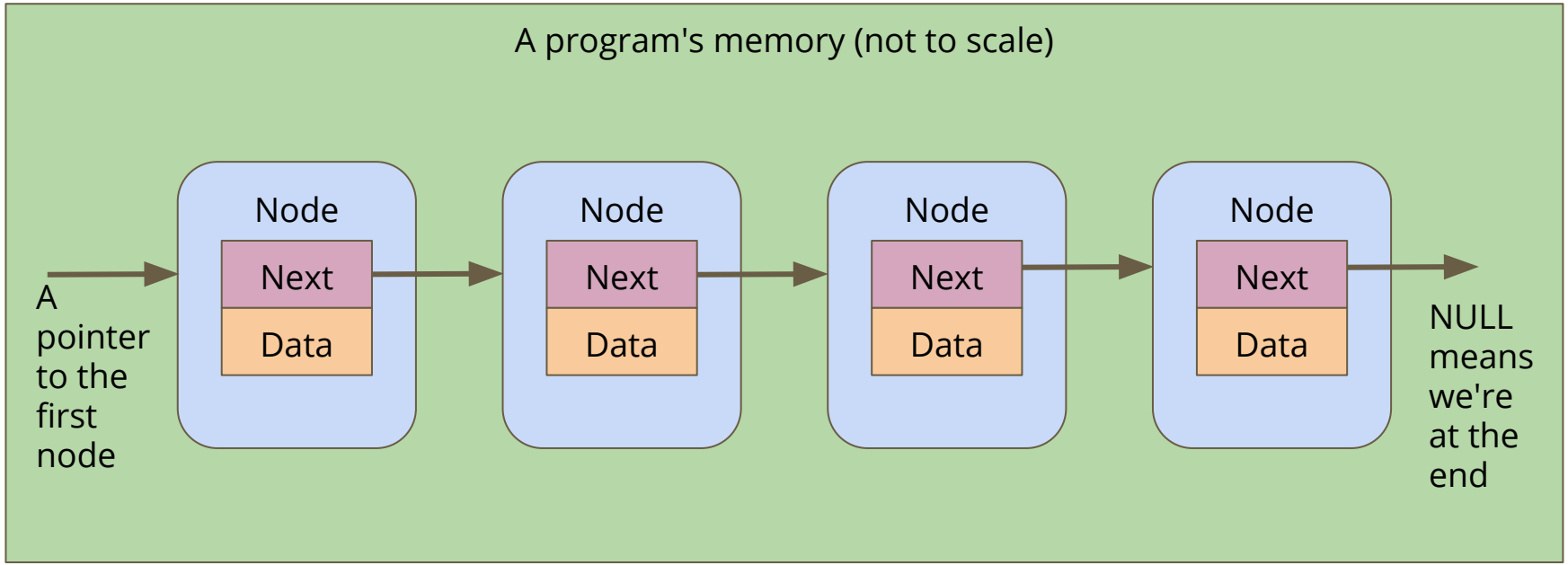
Recap - Linked Lists

Basic components

- A node struct
- Allocated memory for each created node (using **malloc**)
- Each node has a pointer to another node
- Nodes are chained together with these pointers

- We keep track of the list with a pointer to the first node
- The final node's next pointer will be NULL

Linked List diagram



What does a list node look like?

A node is a single element in a list

- A node is made using a struct
- It contains a piece of data (one or more variables)
- We can make nodes with more information if we want

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

```
struct node {
    struct node *next;
    char name[20];
    char type[20];
    int level;
}
```

Battle Royale

Let's use a Linked List to track the players in a game

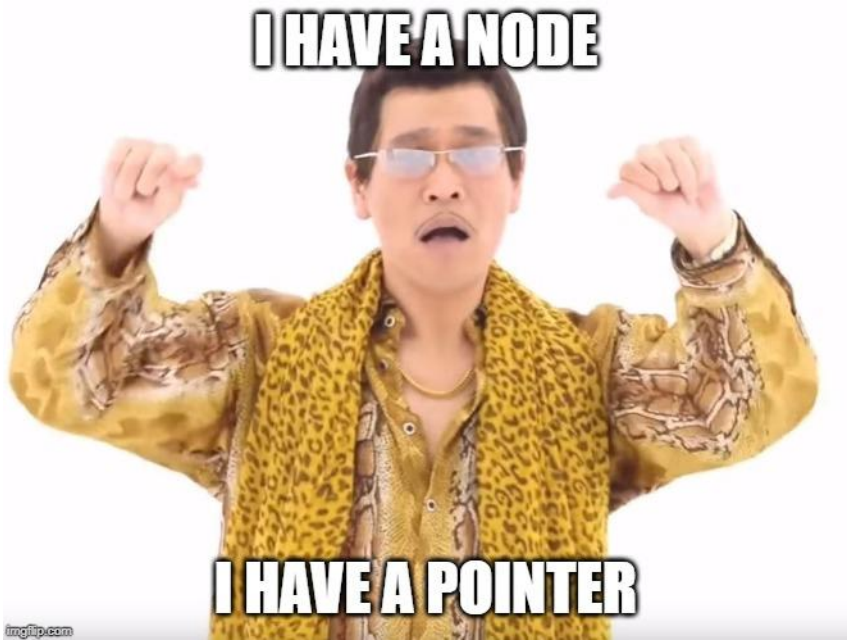
- We're going to start by adding players to the game
- We want to be able to list all the players that are currently in the game (this can change as we go on)
- Can we make sure our list is in order?
- We also want to be able to find and remove players from the list if they're knocked out of the round

What will our nodes look like?

We're definitely going to want a basic node struct

- Let's start with a name
- And a pointer

```
struct node {
    struct node *next;
    char name[50];
}
```



Creating nodes

We'll want a function that creates a node

```
// Create a node using the name and next pointer provided
// Return a pointer to this node
struct node *createNode(char newName[], struct node *newNext) {
    struct node *n;
    n = malloc(sizeof (struct node));
    if (n == NULL) {
        printf("Malloc failed, out of memory\n");
        exit(1);
    }
    strcpy(n->name, newName);
    n->next = newNext;
    return n;
}
```

Creating the list itself

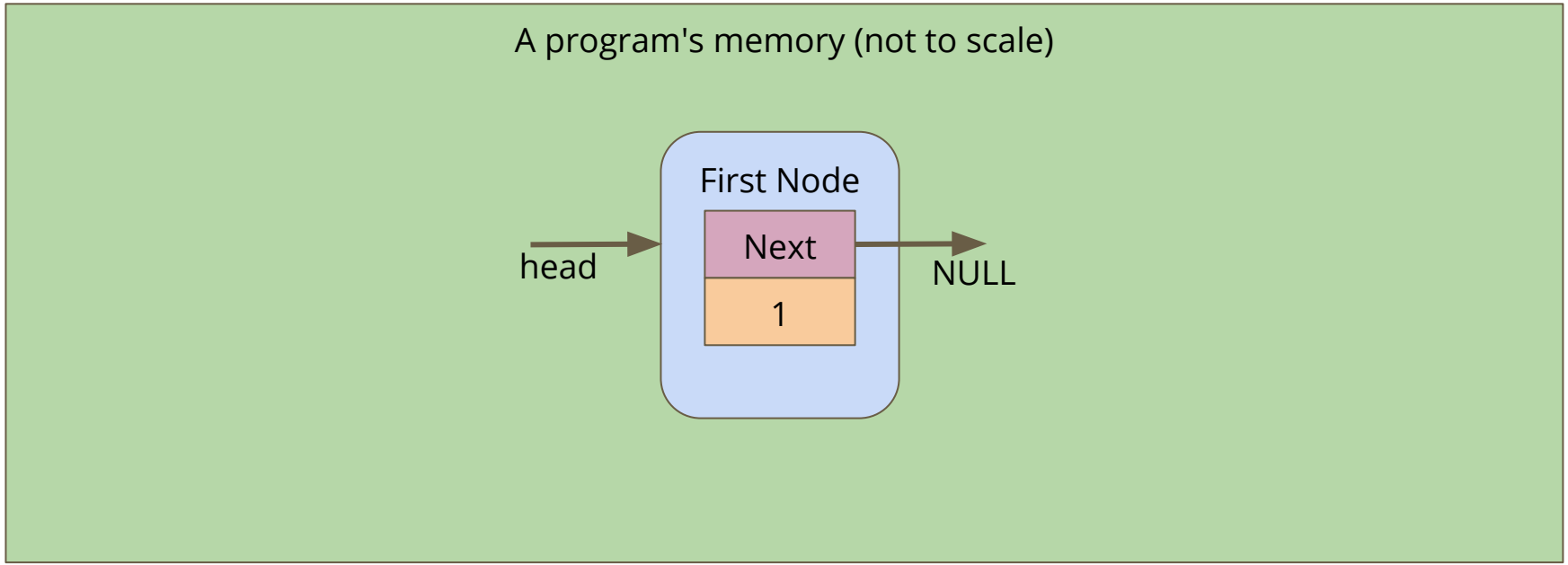
Note that we don't need to specify the length of the list!

```
int main(void) {
    // create the list of players
    struct node *head = createNode("AndrewT", NULL);
    head = createNode("Jashank", head);
    head = createNode("Marc", head);
    head = createNode("AndrewB", head);
    head = createNode("-BPINK- Someone", head);
    head = createNode("Tactical Marc", head);
    head = createNode("COMP1511 Pass Cut-off", head);

    return 0;
}
```

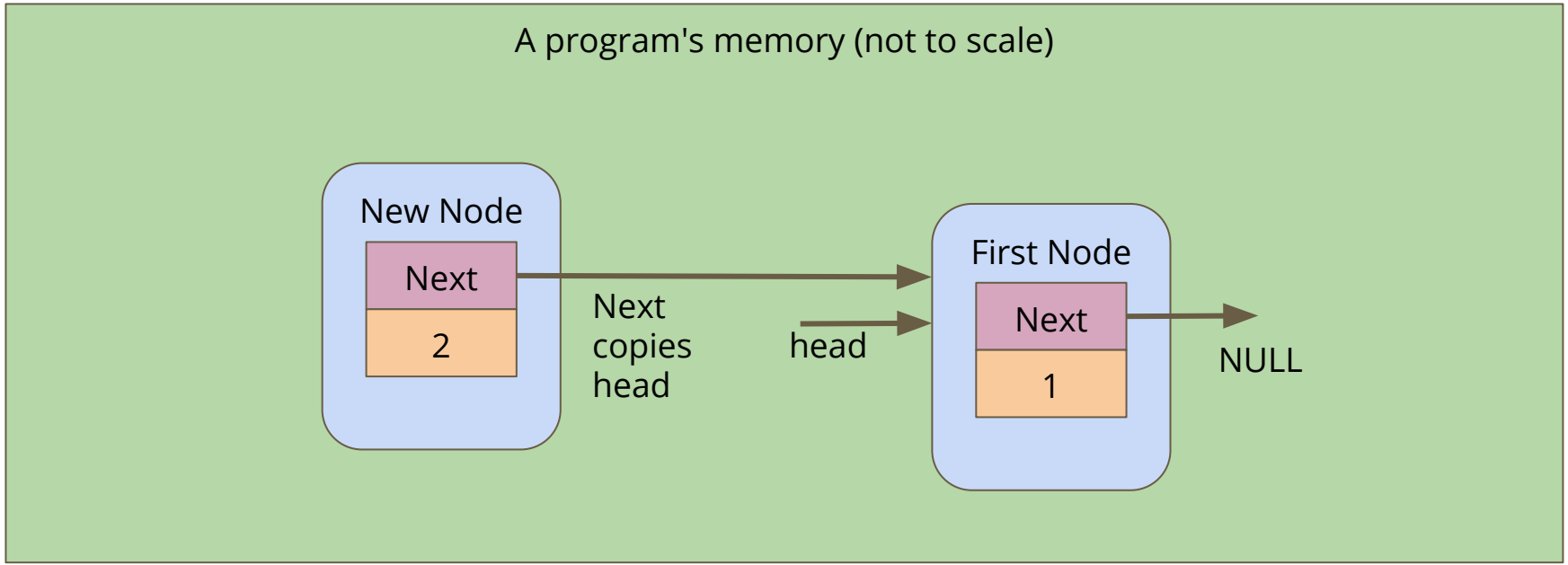
Simple linking of nodes

Head points at the First Node, its next is NULL



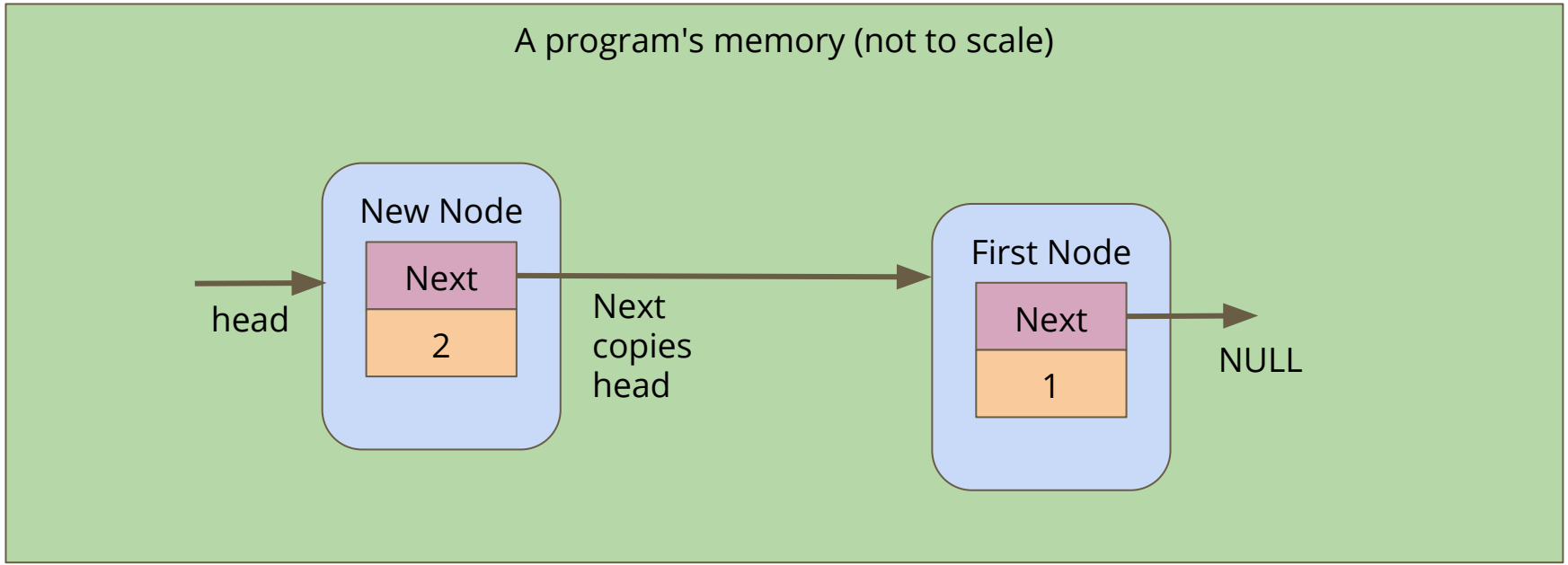
Simple linking of nodes

The New Node is created and copies the head pointer for its next



Simple linking of nodes

Head changes to aim at the new node instead of the first node



Printing out the list of players

How do we traverse a list to see all the elements in it?

- Loop through, starting with the pointer to the head of the list
- Use whatever data is inside the node
- Then move onto the next pointer from that node
- If the pointer is null, then we've reached the end of the list

```
// Loop through the list and print out the player names
void printPlayers(struct node* listNode) {
    while (listNode != NULL) {
        printf("%s\n", listNode->name);
        listNode = listNode->next;
    }
}
```

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 (on Steam, iOS and Android)
- Space Alert Board Game (Vlaada Chvatil)

The list is in a strange order

At the moment our list is just in reverse order of how we created nodes

- We'd like our list to be in some kind of useful order
- In order to do that, let's look at building a list that's ordered
- First we need to know how to insert nodes!

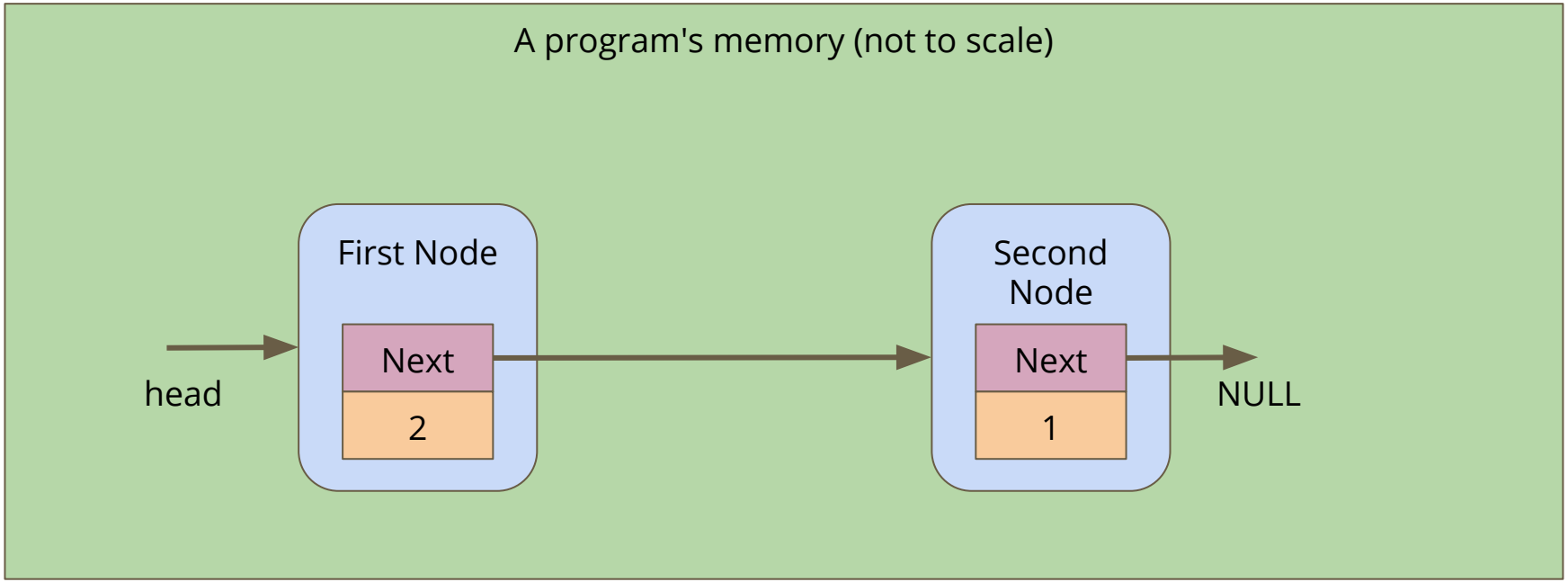
Inserting Nodes into a Linked List

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

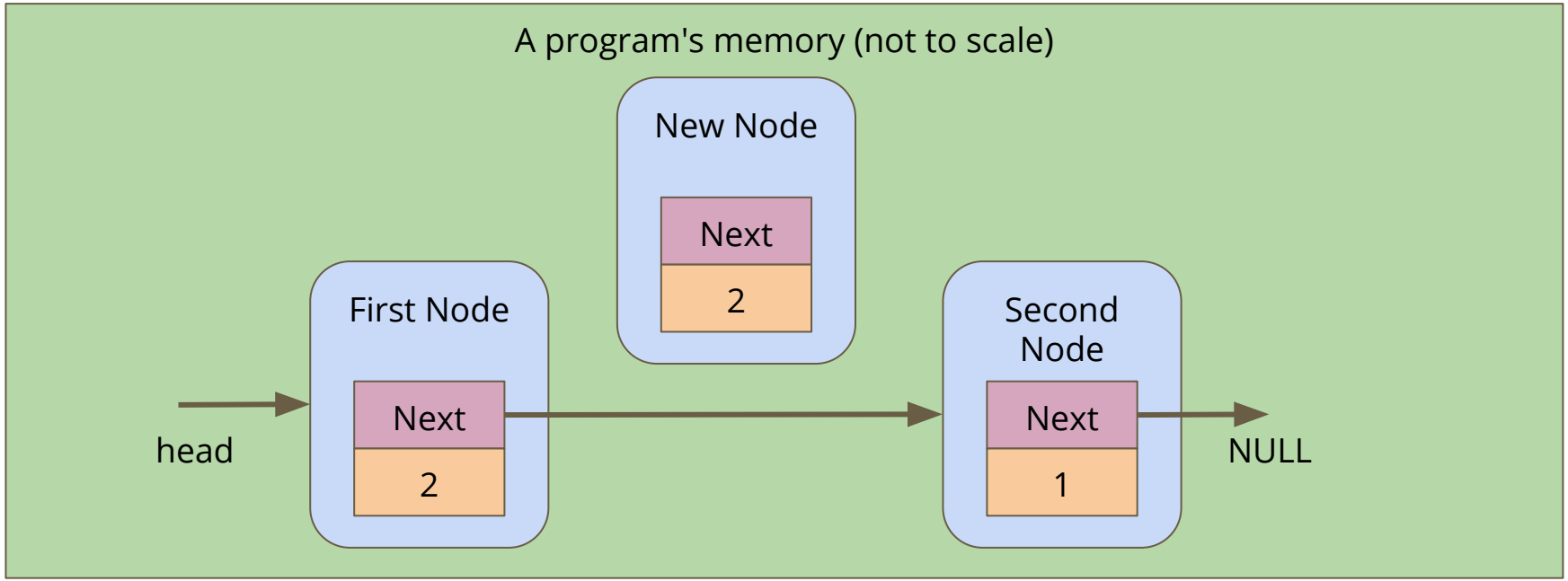
Our Linked List

Before we've tried to insert anything



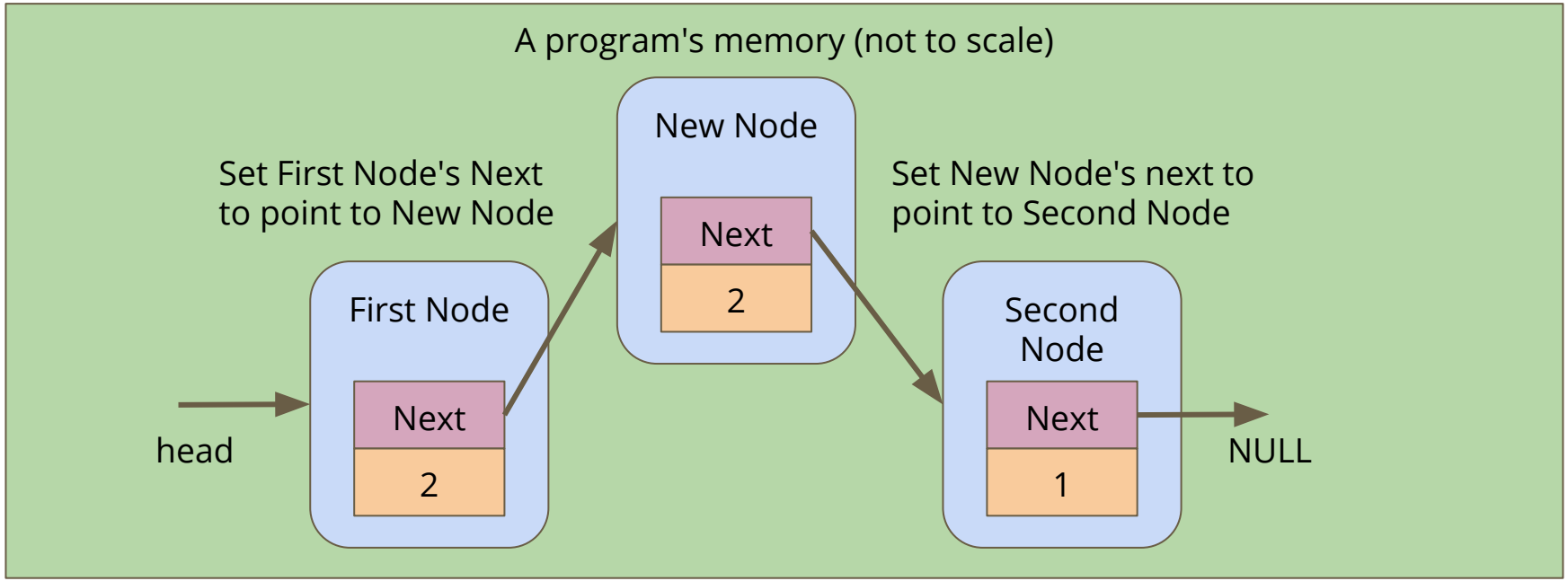
Create a node

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



Connect the new node

Alter the **next** pointers on the First Node and the New 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;
}
```

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 a rough alphabetical order
- We'll just take the first character and compare whether it's before or or after us in the alphabet

Inserting into a list Alphabetically

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

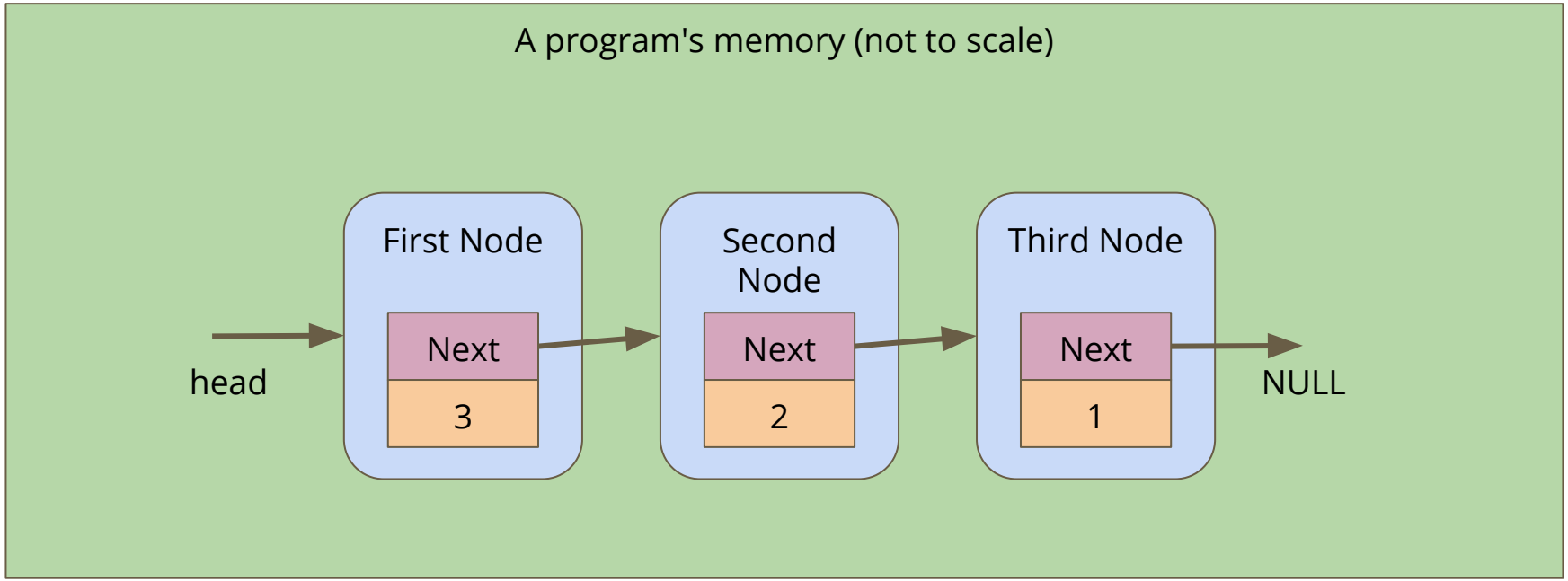
Removing a node

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

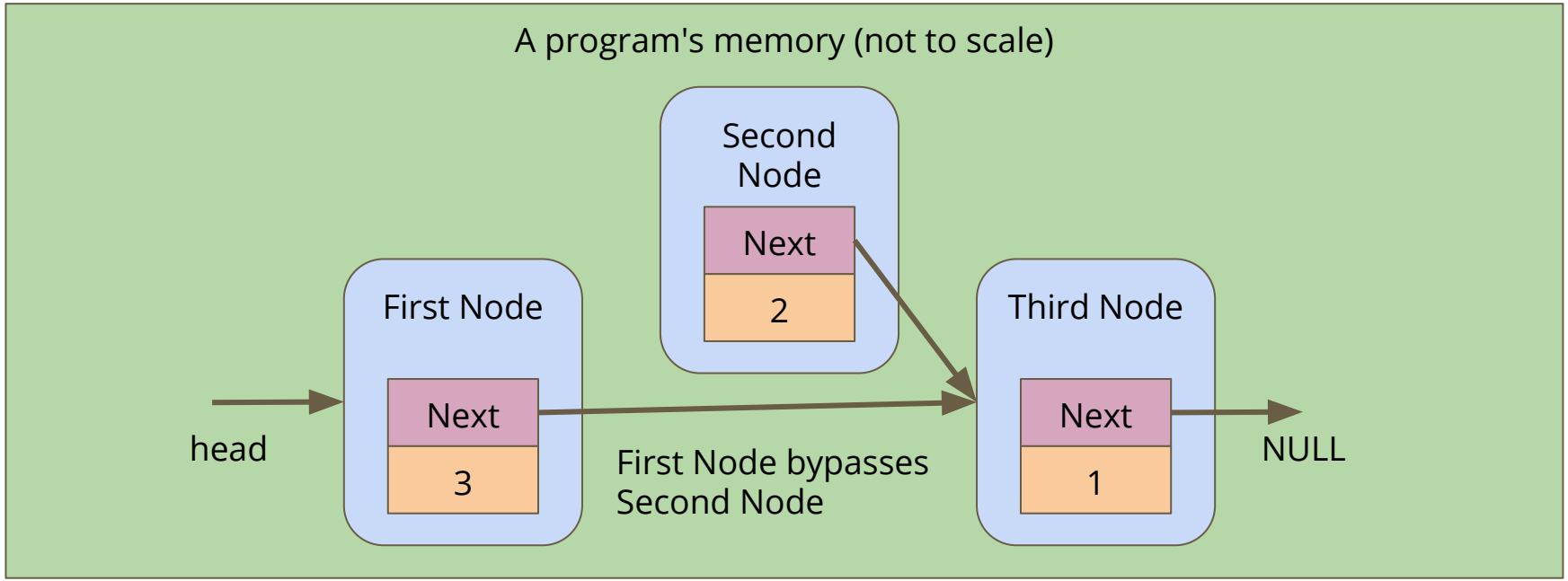
Removing a node

If we want to remove the Second Node



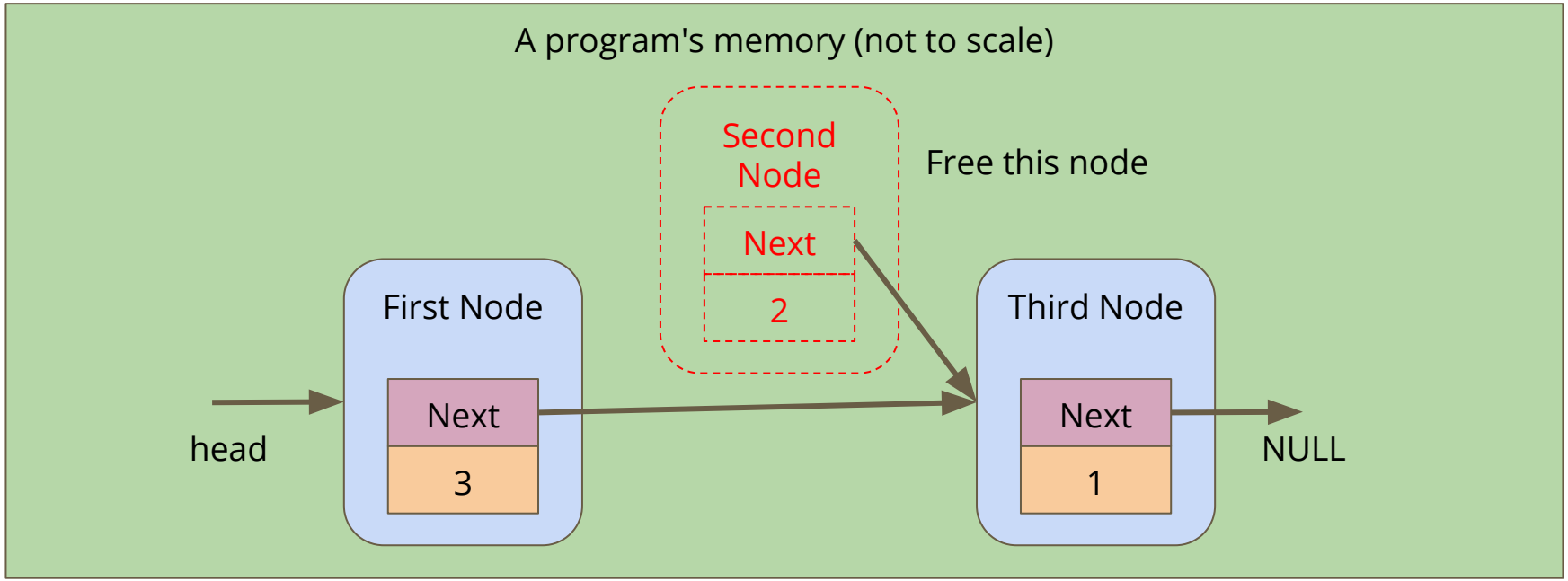
Removing a node

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



Removing a node

Free the memory from the now bypassed node



Removing a Node

```
struct node *removeNode(struct node* head, char name[]) {
    struct node *previous = NULL;
    struct node *n = head;
    // Loop through to try to find the correct node
    while (n != NULL && strcmp(name, n->name) != 0) {
        previous = n;
        n = n->next;
    }
    if (n != NULL) { // found the node
        if (previous == NULL) { // it's the first node
            head = n->next;
        } else {
            previous->next = n->next;
        }
        free(n);
    }
    return head;
}
```

Let's play a game

Once our list is created, we can play

- We'll tell the game who's been knocked out
- Then our program will find the person and remove them from the list

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

What did we learn today?

Linked Lists

- Nodes can have a variety of information in them
- Looping through the list
- Inserting nodes
- Inserting nodes into an ordered list
- Removing nodes