COMP1511 - Programming Fundamentals

Week 8 - Lecture 13

What did we learn today?

Linked Lists

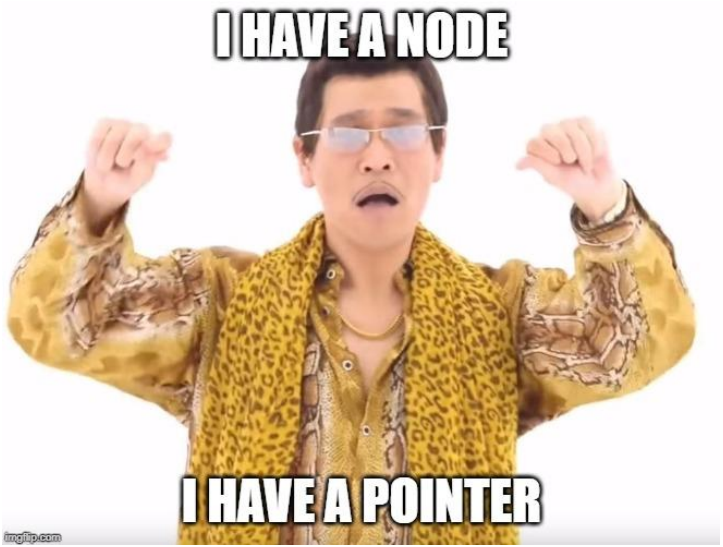
- Recap of Linked Lists
- Building the list
- Looping through the list
- Inserting nodes at a specific location
- Inserting nodes into an ordered list

Recap - Linked Lists

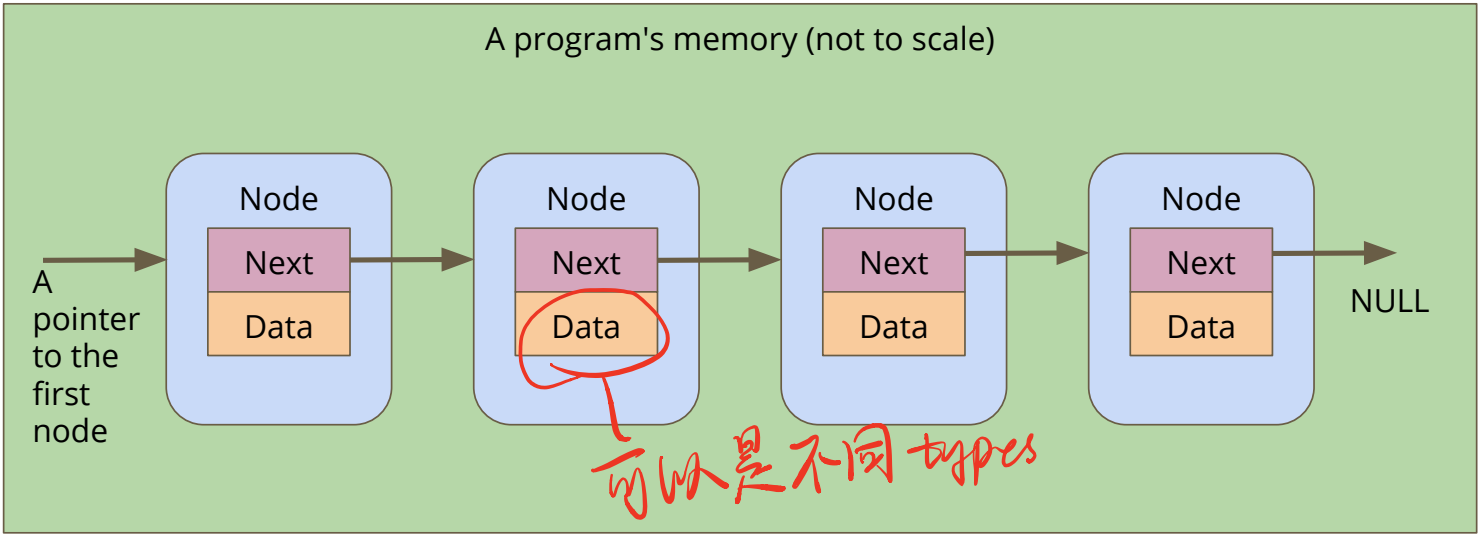
A chain of identical structs to hold information

- 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;
    }
}
```

What will our player nodes look like?

We're definitely going to want a basic node struct

- Let's start with a name
- And a pointer to the next node

```
struct player {
    char name[MAX_NAME_LENGTH];
    struct player *next;
};
```

Creating players

We'll want a function that creates a node

```
// Create a player node using the name and next pointer provided
// Return a pointer to this node
struct player *createPlayer(char newName[], struct player *newNext) {
    struct player *p;
    p = malloc(sizeof (struct player));
    strcpy(p->name, newName);
    p->next = newNext;
    return p;
}
```

Creating the list itself

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

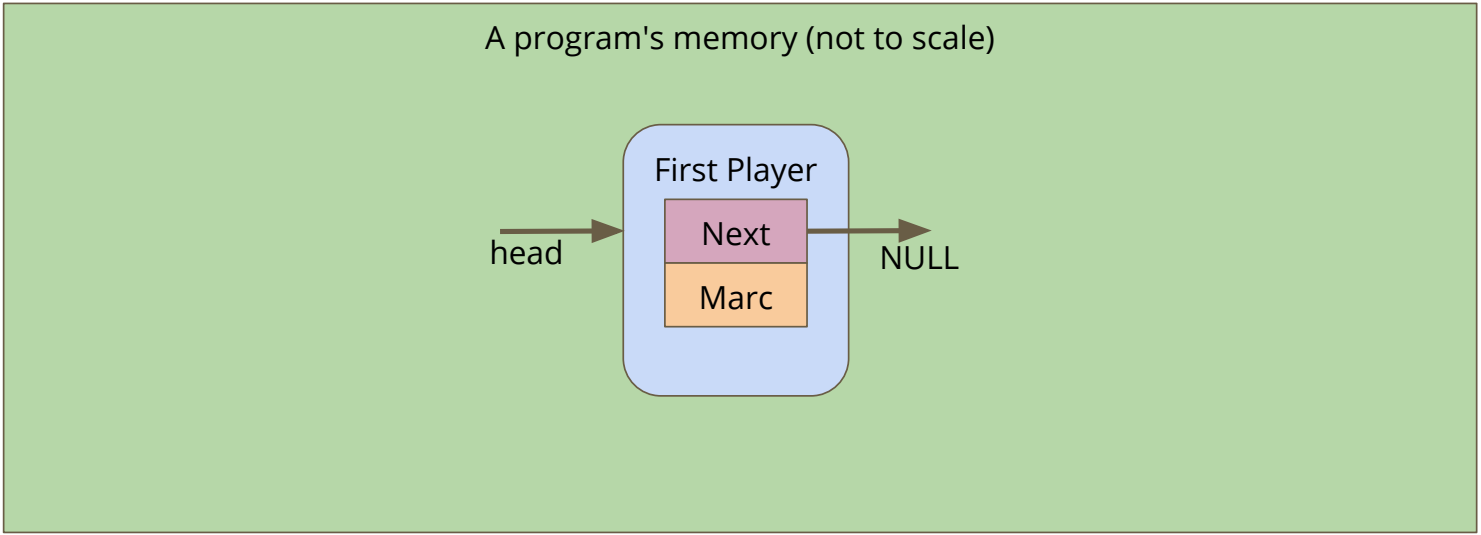
```
int main(void) {
    // create the list of players
    struct node *head = createPlayer("Marc", NULL);
    head = createPlayer("Tom", head);
    head = createPlayer("Goku", head);
    head = createPlayer("Bulma", head);
    head = createPlayer("Master Roshi", head);

    return 0;
}
```

This is one basic way of connecting player nodes together to make a list

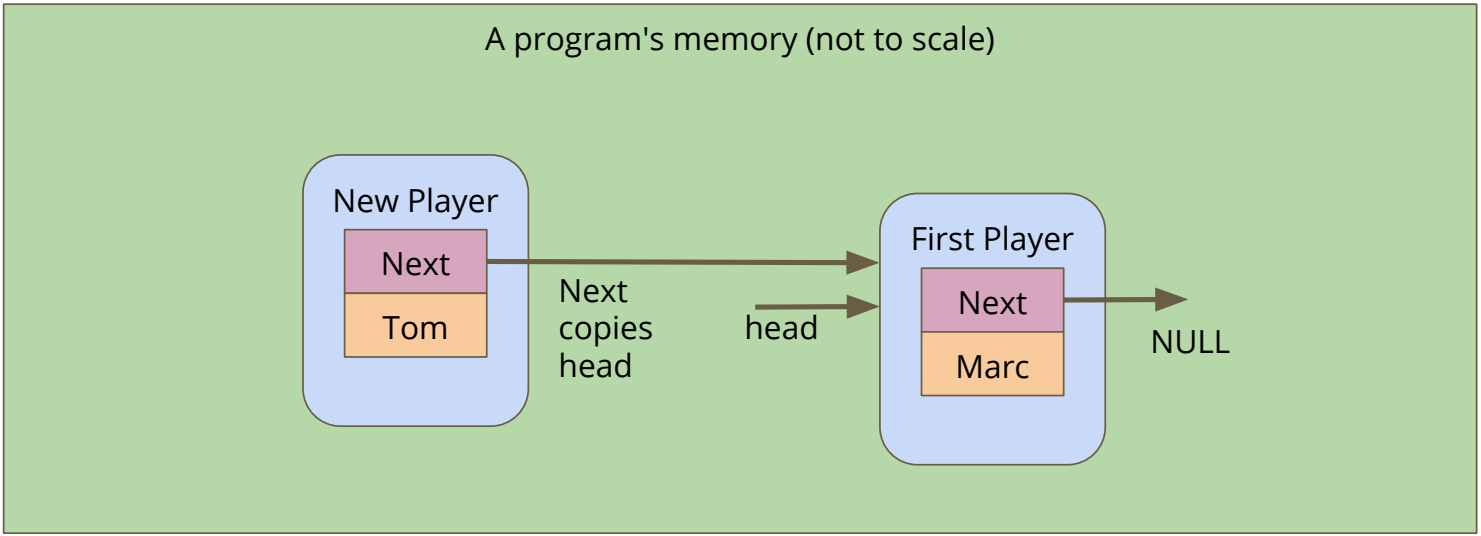
Using createPlayer

Head points at the First Player, its next is NULL



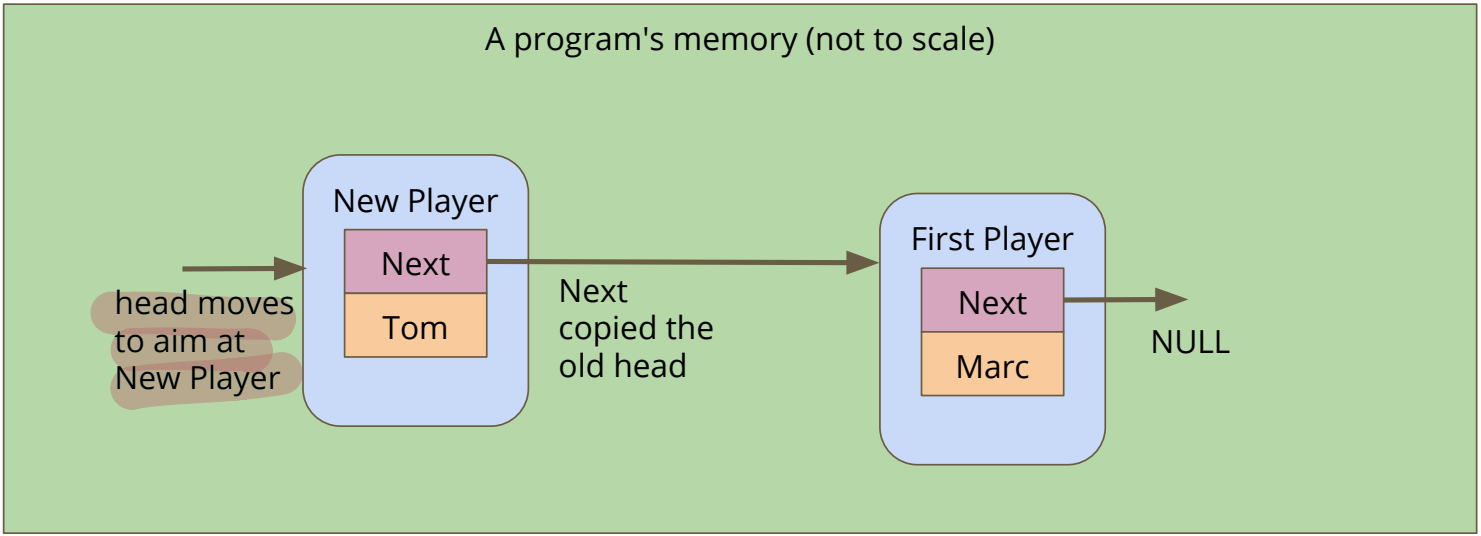
Adding another Player

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



Making sure the list is still valid

createPlayer returns a pointer to New Player, which is assigned to head



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 player node
- Then move onto the next pointer from that player 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 player* listPlayer) {
    while (listPlayer != NULL) {
        printf("%s\n", listPlayer->name);
        listPlayer = listPlayer->next;
    }
}
```

Inserting Nodes into a Linked List

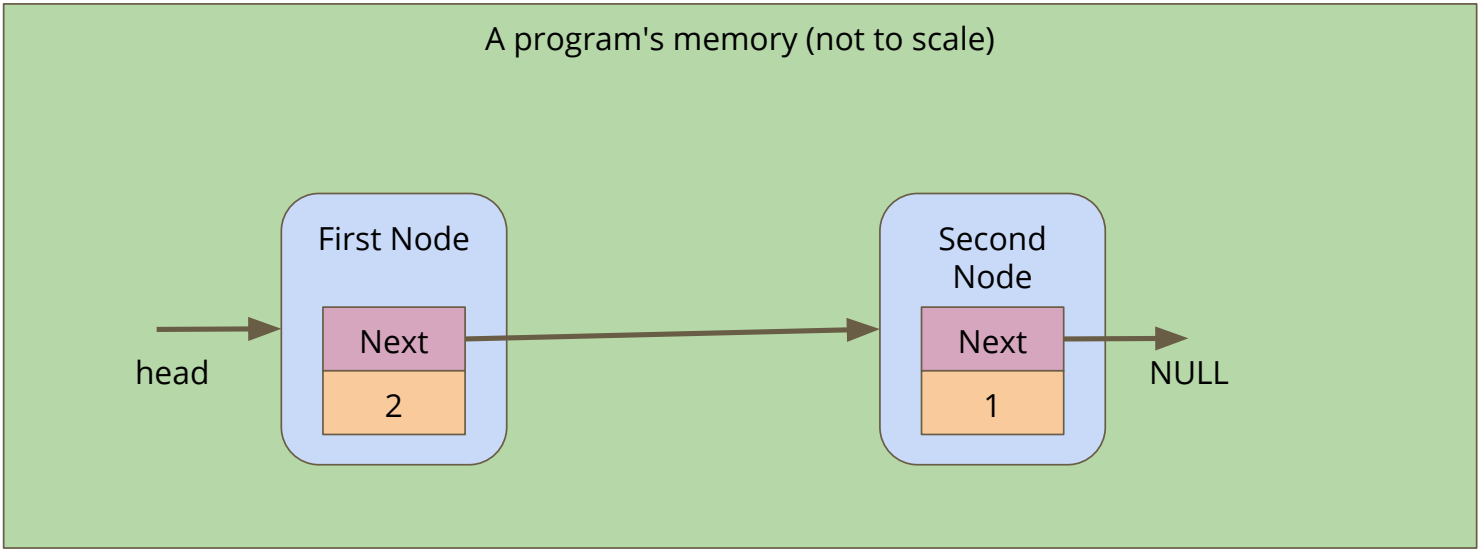
Linked Lists allow you to insert nodes in between other nodes

- 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 . . .

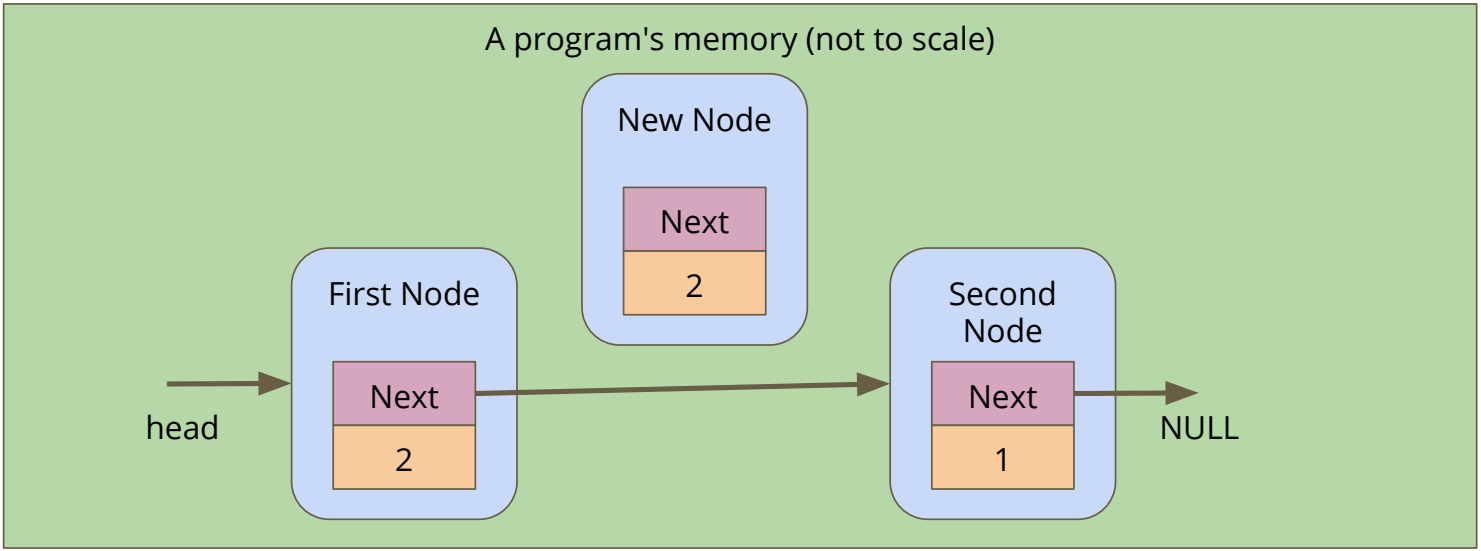
Our Linked List

Before we've tried to insert anything



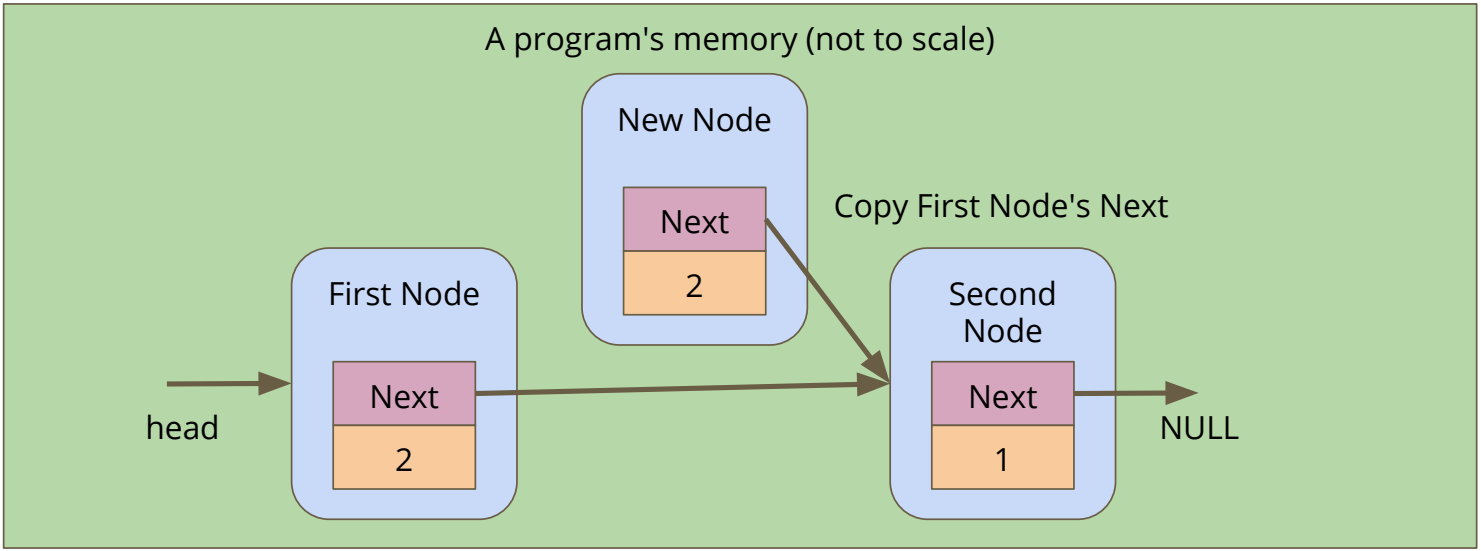
Create a node

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



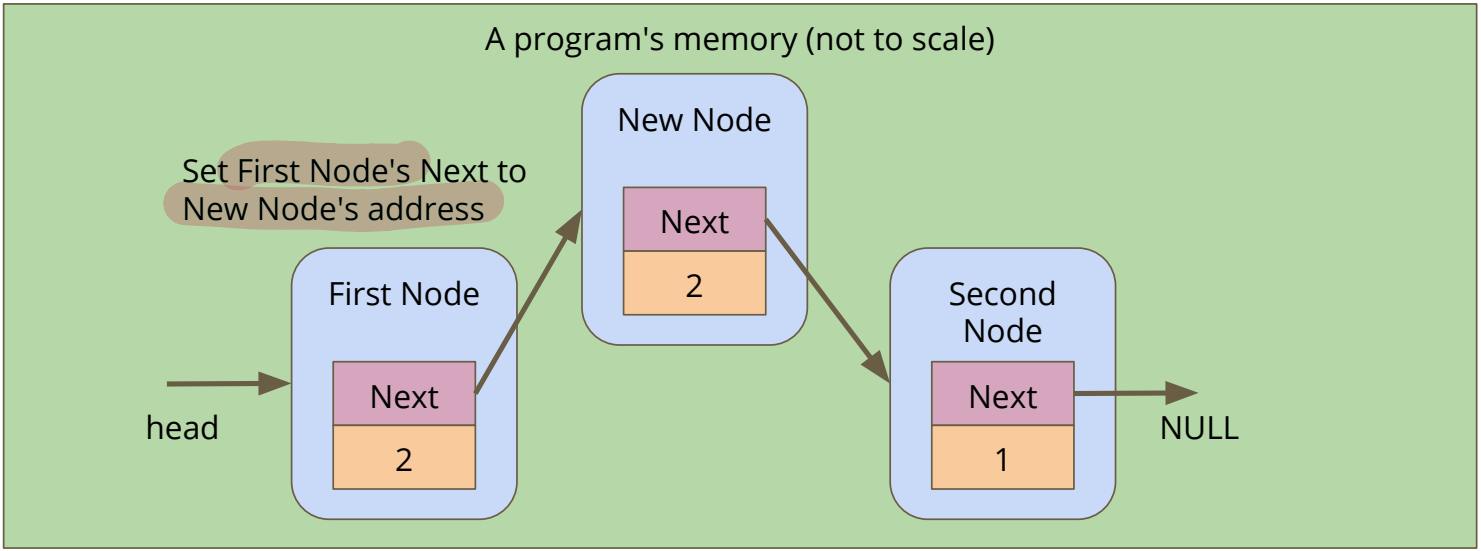
Connect the new node to the second node

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 of players

```
// Create and insert a new node into a list after a given insert position
struct player *insert(struct player* insertPos, char newName[]) {
    struct player *p = createPlayer(newName, NULL);
    if (insertPos == NULL) {
        // List is empty, p becomes the only element in the list
        insertPos = p;
        p->next = NULL;
    } else {
        // Set the new player (p)'s next to after the insertion position
        p->next = insertPos->next;
        // Set the insert position node's next to now aim at p
        insertPos->next = p;
    }
    return insertPos;
}
```

Inserting Players to create a list

We can use insertion to have greater control of where players end up in a list

```
int main(void) {
    // create the list of players
    struct node *head = createPlayer("Marc", NULL);
    insert("Tom", head);
    insert("Goku", head);
    insert("Bulma", head);
    insert("Master Roshi", head);

    printPlayers(head);

    return 0;
}
```

Insertion with some conditions

We can now insert into any position in a Linked List

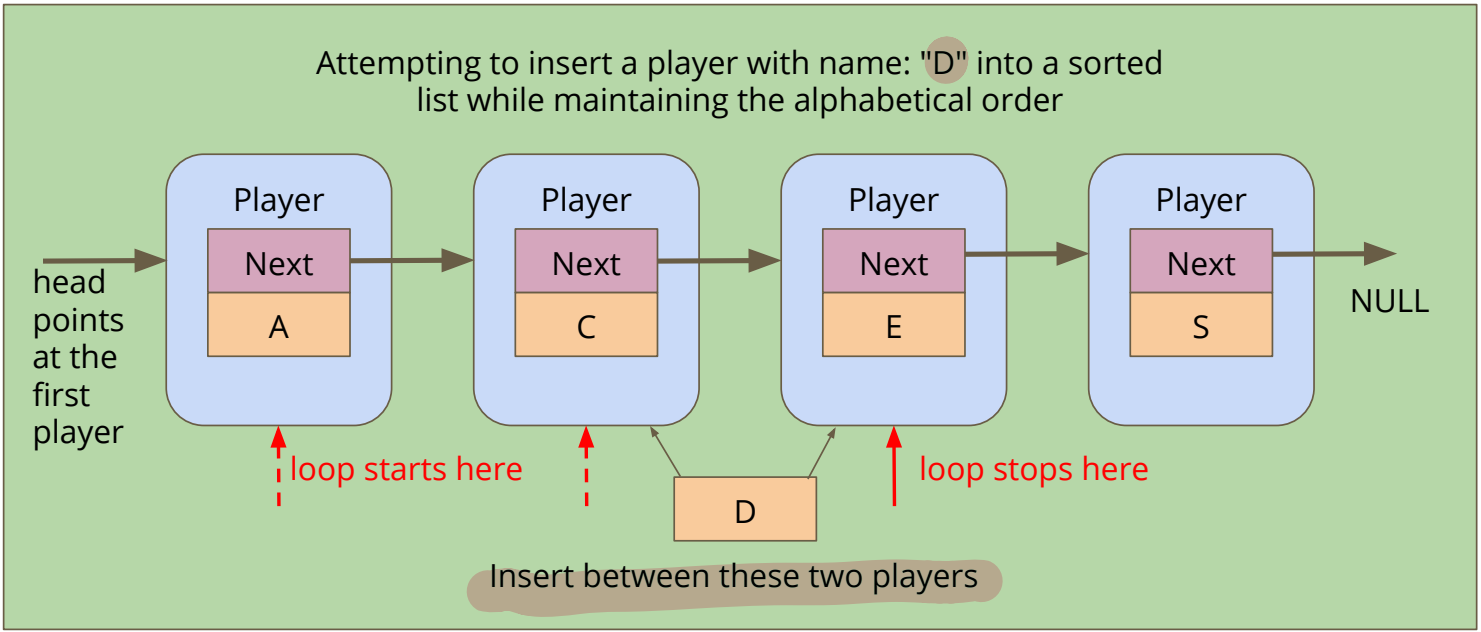
- We can read the data in a node and decide whether we want to insert before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to use a **`string.h`** function, **`strcmp()`** for this
- **`strcmp()`** compares two strings, and returns
  - 0 if they're equal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



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;
    }
}
```

first > second