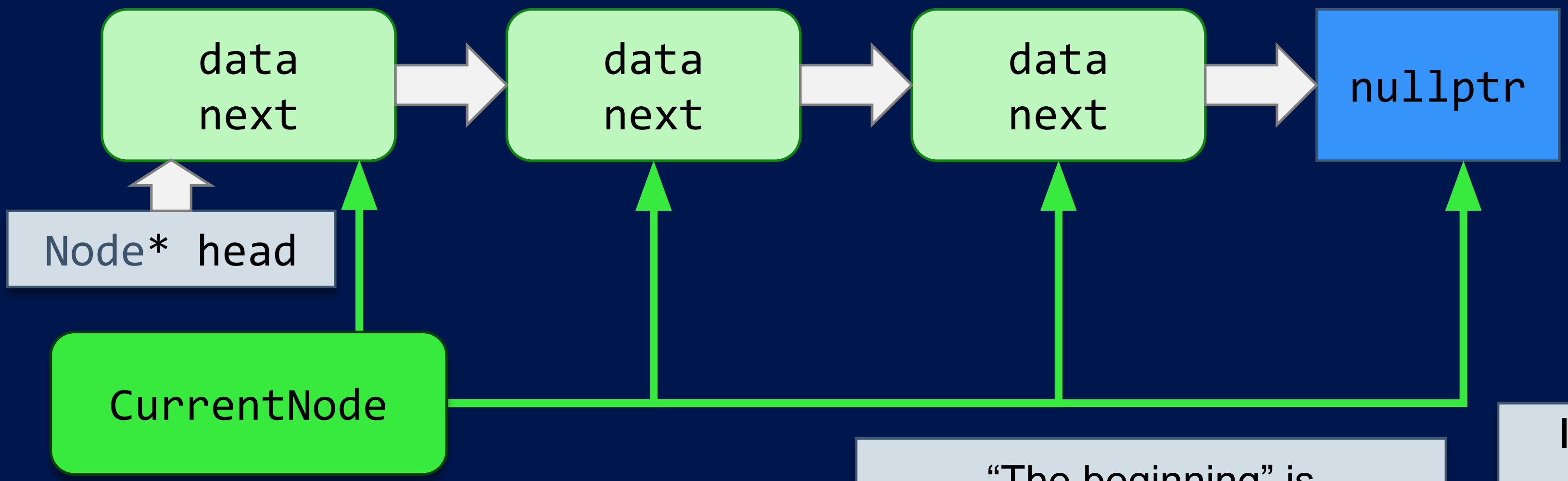
COP3503

Linked List Operations

Back to Our List of Numbers



"The beginning" is anywhere you want to start.

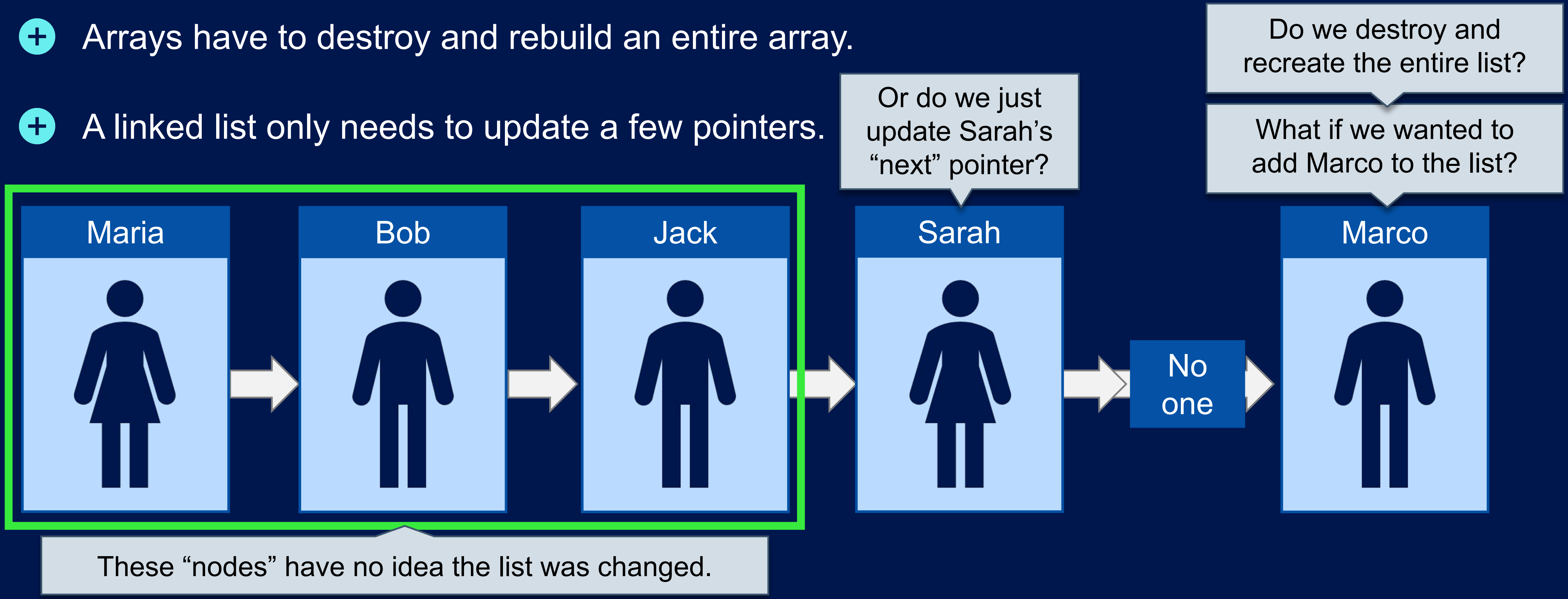
It's commonly the head of the list, but could be the tail, or somewhere in the middle!

```
//Start at the beginning (with a node*)
currentNode = head

// Check if we should keep iterating
while currentNode is valid (i.e. not nullptr)

    // Do something with current node
    // Move our "current" pointer to the next in the list
    currentNode = currentNode->next
```

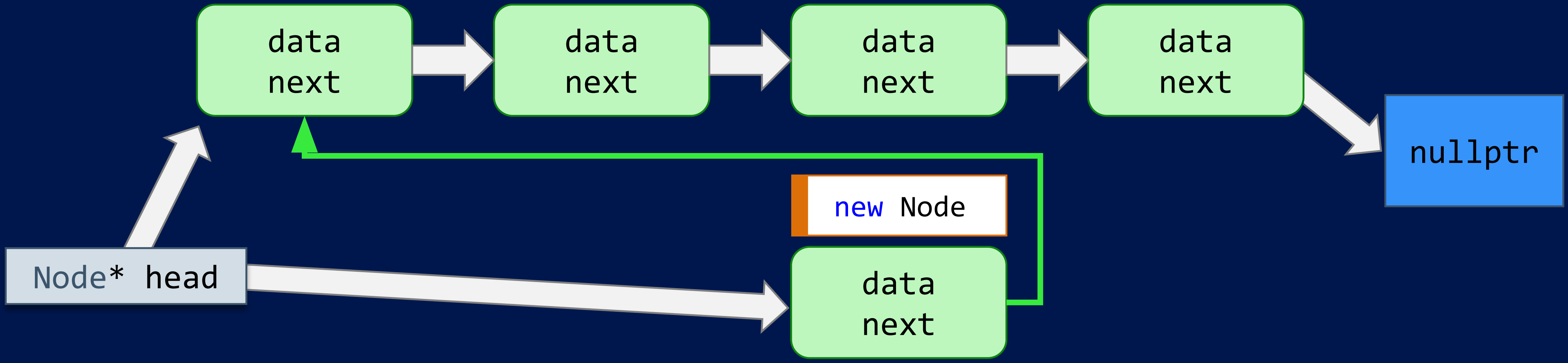
Iteration may not always be in a forward direction.

Inserting a Node Into an Existing List

- A big advantage a linked list has over arrays is the ability to add and remove nodes quickly.
- Arrays have to destroy and rebuild an entire array.
- A linked list only needs to update a few pointers.



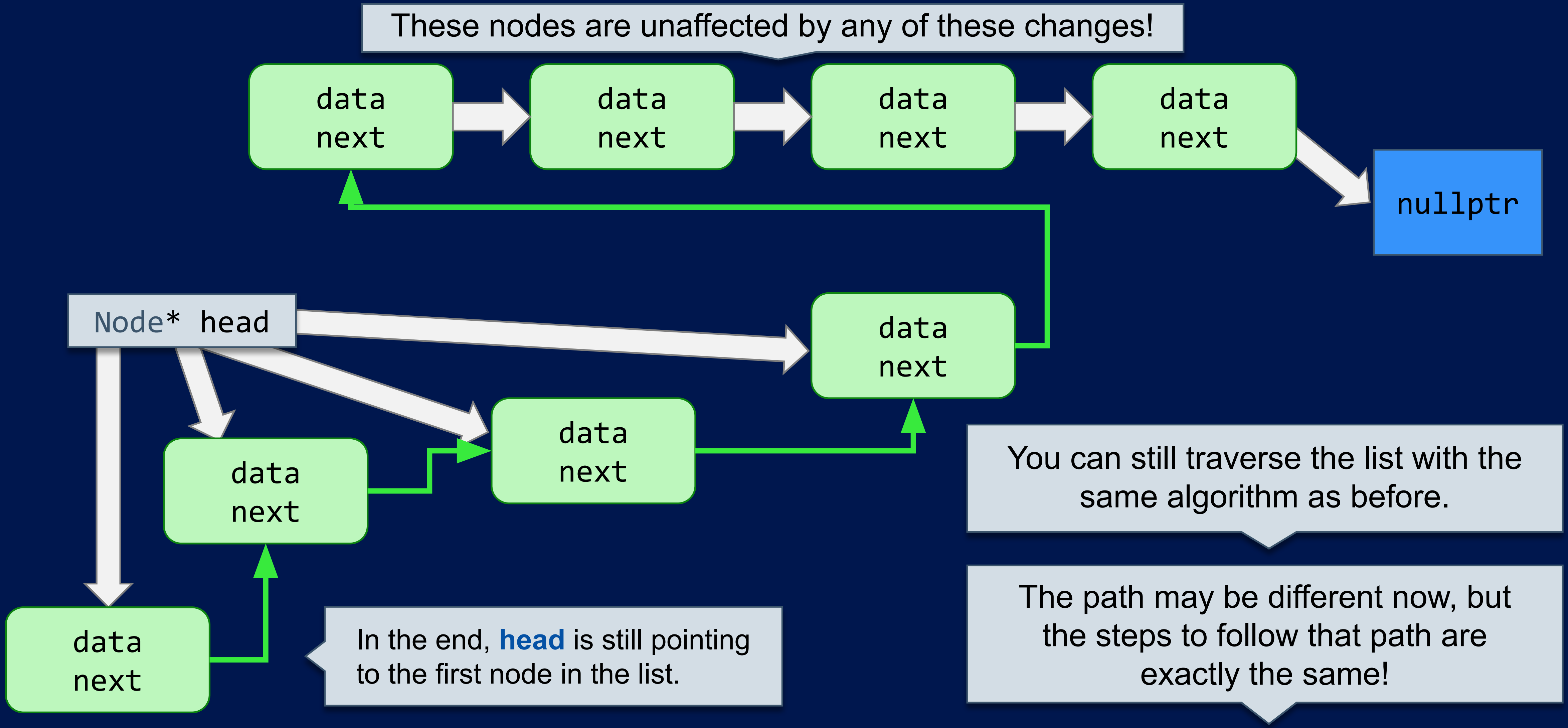
Adding to the Front of the List



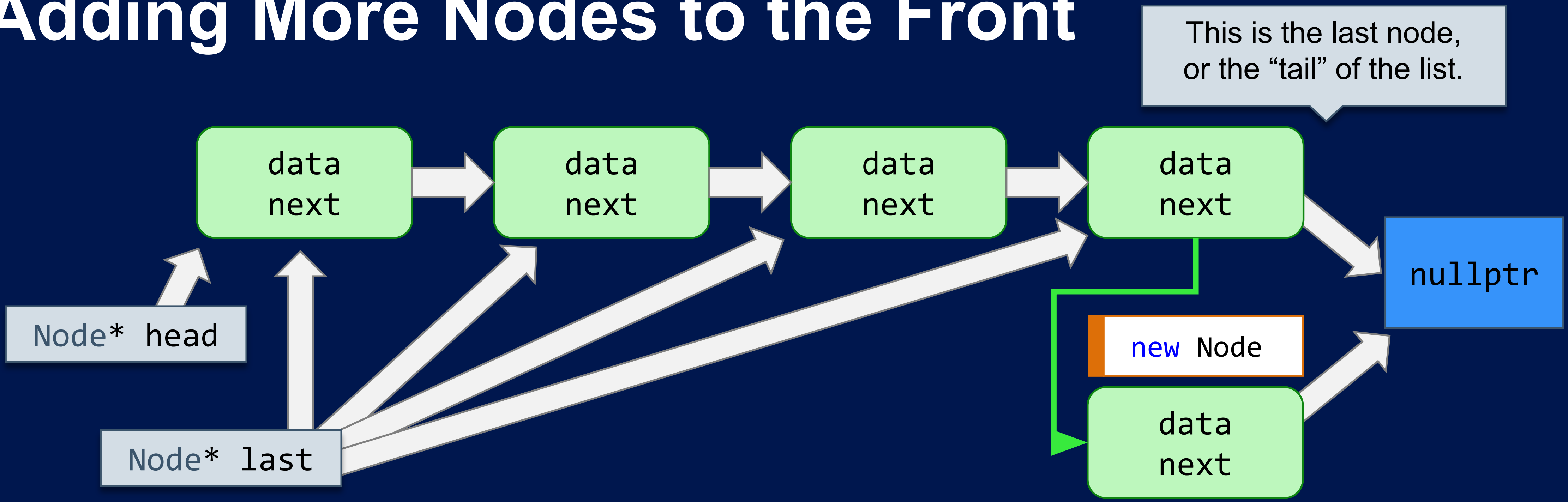
Head may point somewhere different now, but still represents the first node in the list.

1. Create a new node.
2. Set its next node to the start of the list (i.e., the head)
3. Set the current head pointer to the new node.

Adding More Nodes to the Front



Adding More Nodes to the Front




```
// Slightly different traversal
currentNode = head

// Keep going until we find node with a "next"
// pointer that is nullptr—that's the last
while currentNode->next is valid
    currentNode = currentNode->next
```

- Similar to adding to the front of the list:
  1. Create a new node.
  2. Set the next pointer of the "tail" to this new node.
  3. If you have a tail pointer, set the tail pointer itself to this new node (not applicable in this example).

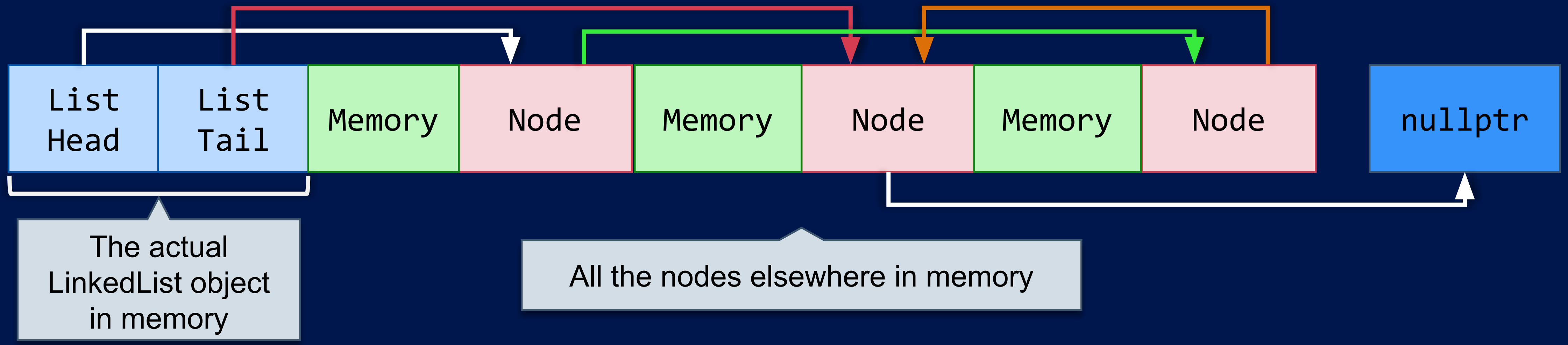
Storing Head and Tail Pointers

- A basic Linked List implementation might only store the first node:

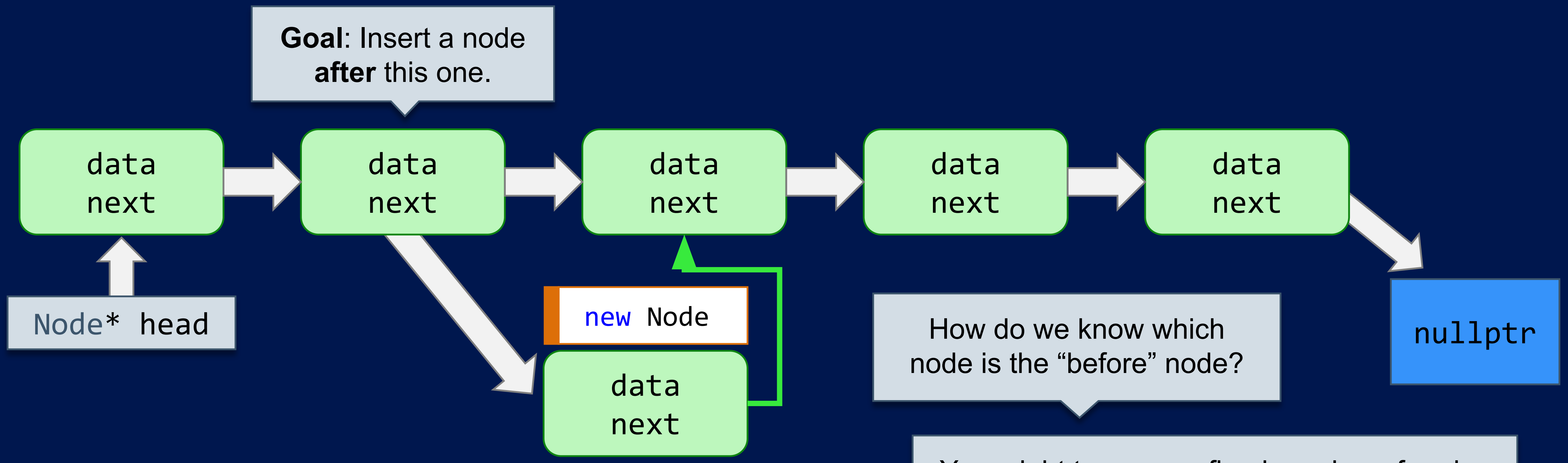
```
class LinkedList
{
    Node* head;
};
```

- You could also store a tail pointer.

```
class LinkedList
{
    Node* head;
    Node* tail;
};
```



Inserting a Node in Between Two Nodes



1. Create a new node.
2. Set its "next" pointer to the node that will ultimately follow it.
3. Set the "previous" node's next pointer to the new node.

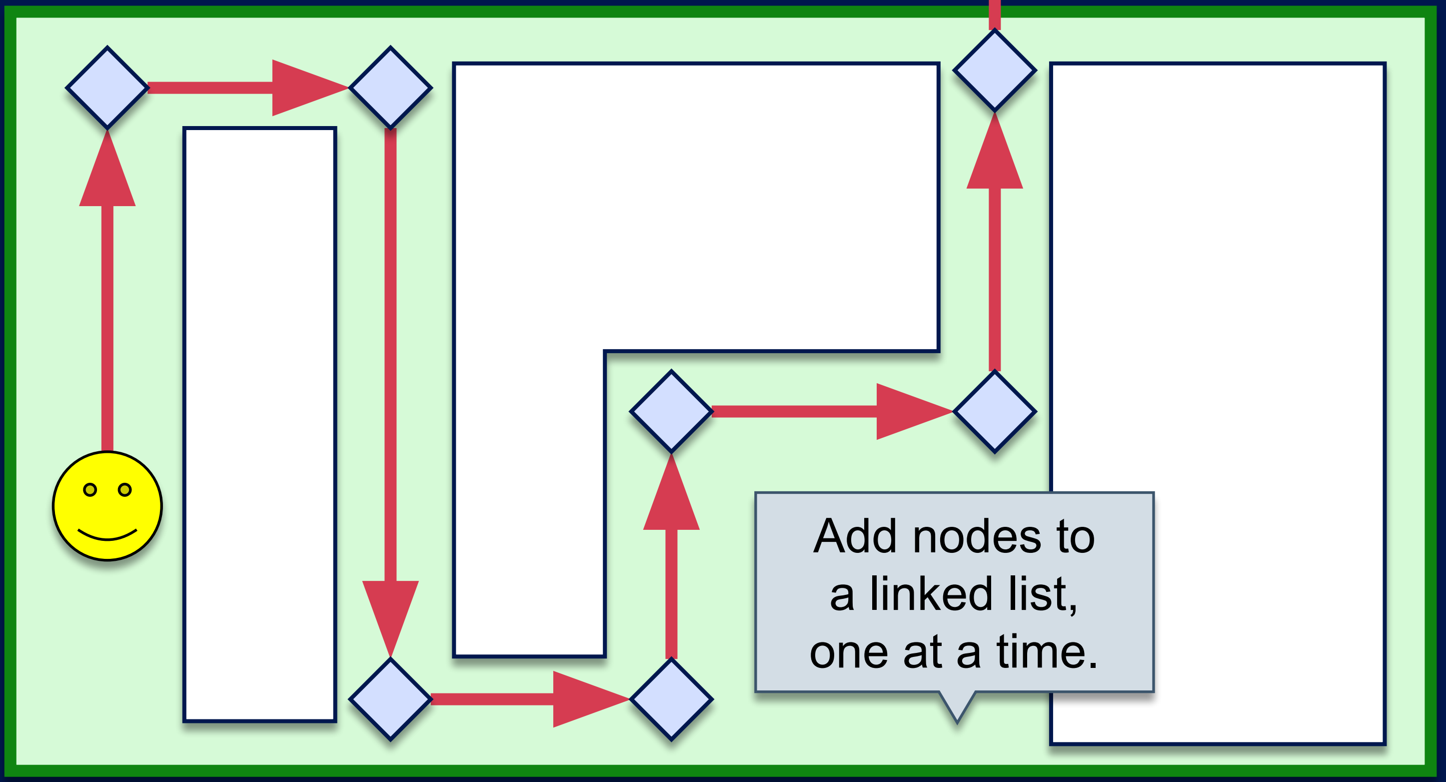
You might traverse a fixed number of nodes (i.e., InsertNodeAfterNodeIndex(1)).

You might traverse and search for a node with a specific value (i.e., InsertNodeAfterValue(200)).

Use Case: Movement / pathfinding

nullptr

Last waypoint has no next.




```
class Character
{
    Waypoint* destination;
}
```

```
class Waypoint
{
    float xPosition;
    float yPosition;
    Waypoint* next;
}
```

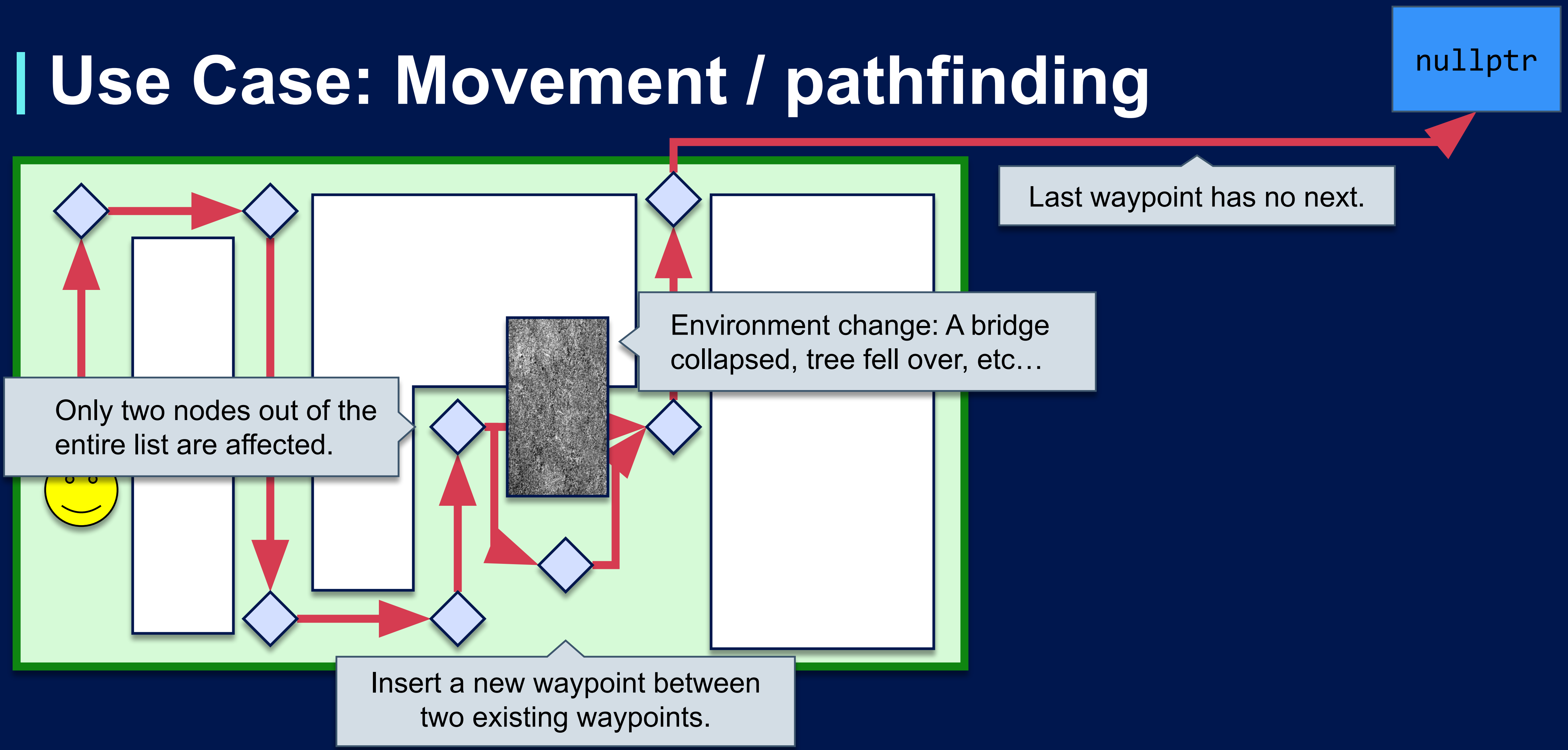
Player Movement Algorithm:

If destination is not null
and player is not at destination,
take a step toward destination.

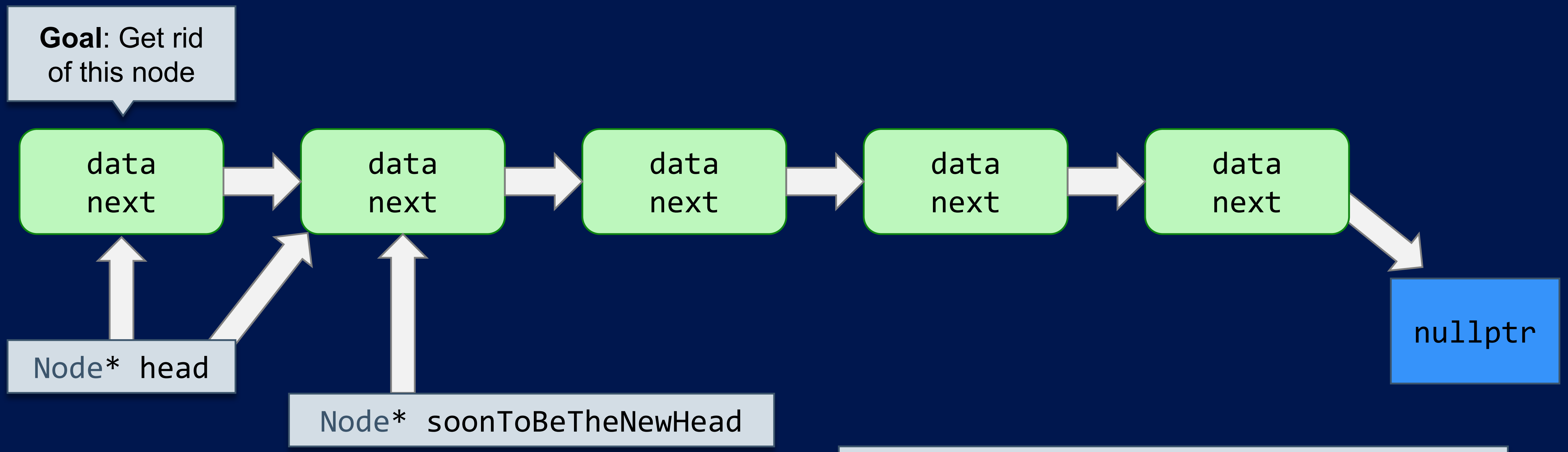
Then,
if player is at destination
destination = destination->next.

What if the environment changed
and blocked part of the path?

Use Case: Movement / pathfinding



Removing the Head Node

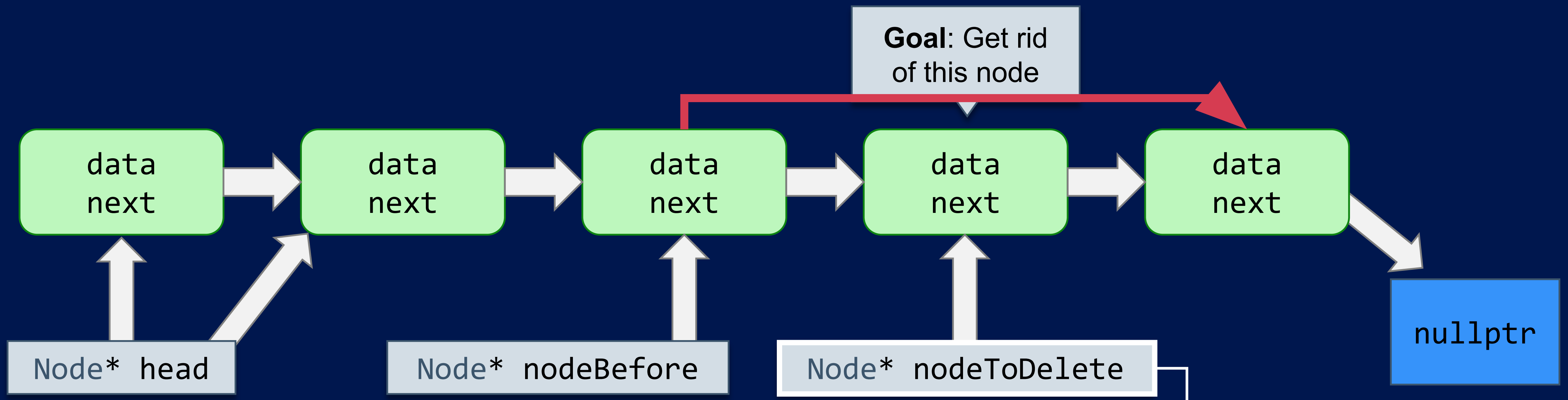


1. Create a pointer to the node **after** the head.
2. Delete the head node.
3. Set the head node to the pointer you created.

Why not just call delete Head and be done?

Once head is deleted, any data at that location (including the location of head->next) is lost!

Removing a Node



1. Get a pointer to the node before the one to delete.
2. Set its “next” pointer to the node that will ultimately follow it (the one **after** the deleted node).

```
nodeBefore->next = nodeToDelete->next; // OR…
nodeBefore->next = nodeBefore->next->next;
```

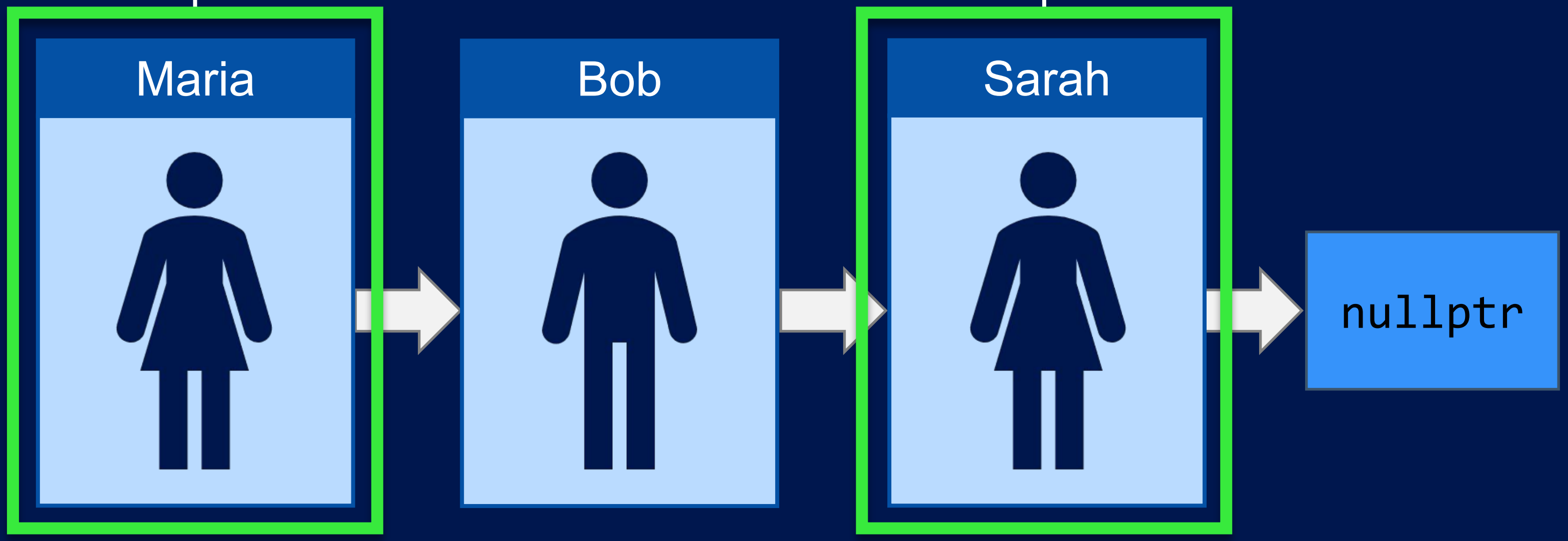
3. Delete the node.

If this is a temporary variable declared in a function somewhere, it will fall out of scope eventually.

Just don’t try to use it after deleting what it pointed to!

Checking for Node Validity!

What if we wanted to visit **every other** node, starting with the first?




```cpp
struct PersonNode
{
    Person thePerson;
    PersonNode* next;
};
```

```cpp
PersonNode* p = someList.GetFirst(); // Start at the beginning
p->thePerson.SomeOperation();

p = p->next->next;
p->thePerson.SomeOperation();

p = p->next->next;
```

p->next == Bob Node
(Bob Node)->next == Sarah Node

From Sarah, p->next is nullptr

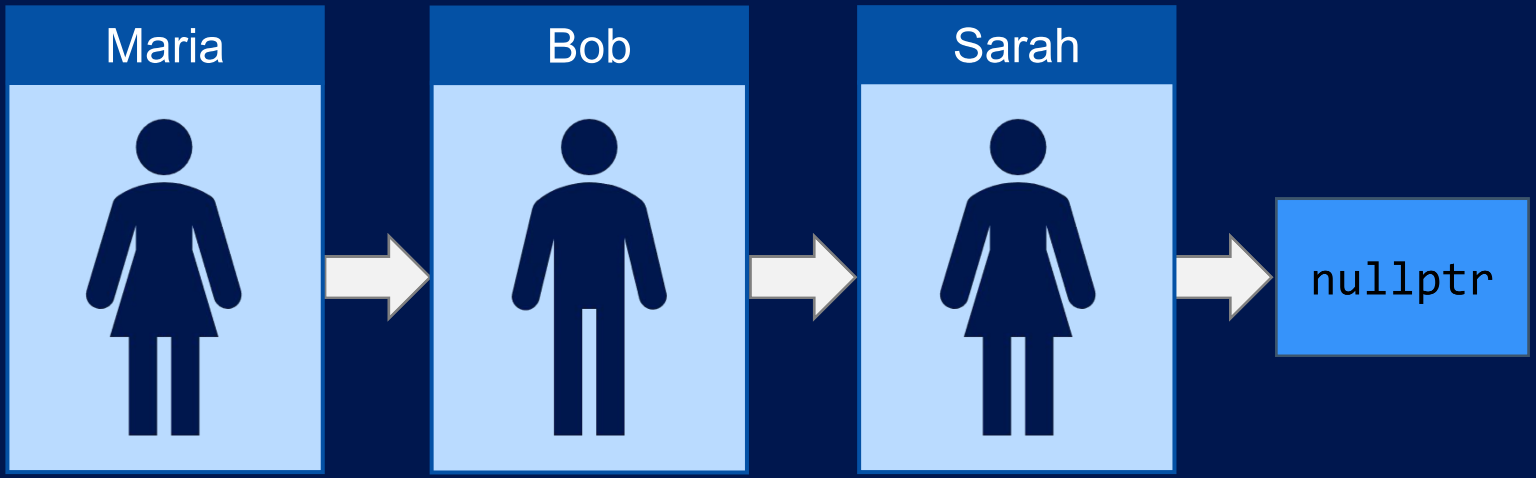
nullptr->next will likely crash your program

You can see this is a bad idea, looking at a diagram.

Your program doesn't have a diagram, and can't see "obviously bad" decisions.

Checking for Node Validity!

```
struct PersonNode
{
    Person thePerson;
    PersonNode* next;
};
```



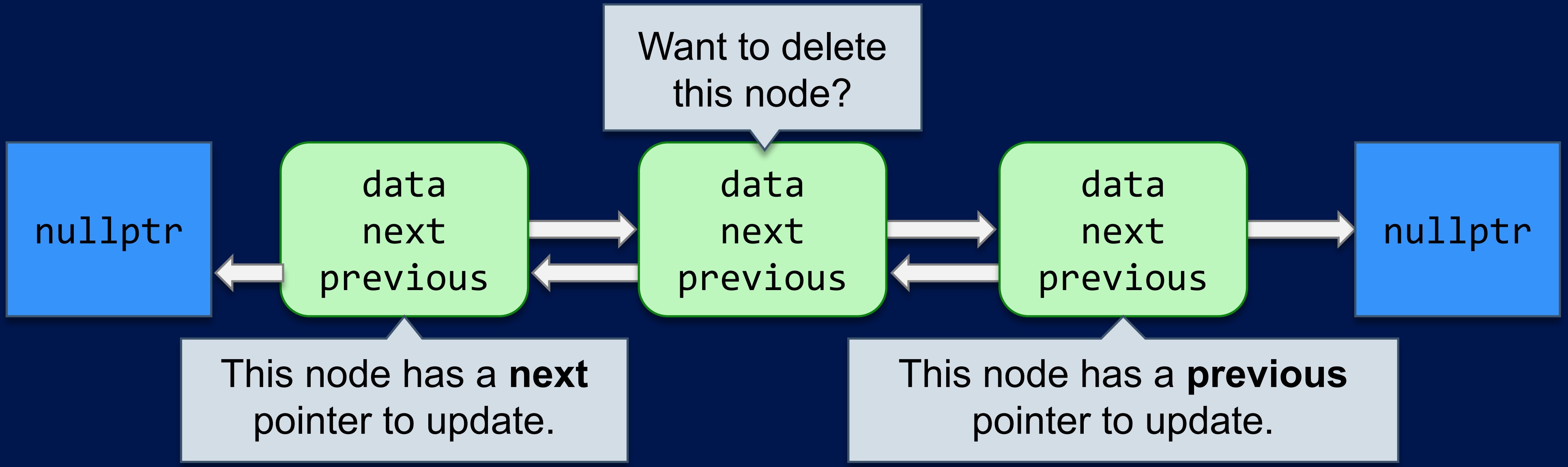

```
PersonNode* p = someList.GetFirst();
if (p != nullptr)
    p->thePerson.SomeOperation();

if (p->next != nullptr) // IF someone follows me...
{
    p = p->next->next; // Move to who follows THEM

    if (p != nullptr)    // If we actually have a valid pointer…
        p->thePerson.SomeOperation(); // THEN, FINALLY do something
}
```

Don’t make assumptions on the structure of the entire list.

Handle each node one at a time, and **check to make sure you have valid pointers**!

Changes With a Doubly-Linked List

- The overall algorithms for any operation is largely the same.
- You have to account for a “previous” pointer in nodes.
  - Initialize them—to `nullptr` if nothing else.
- Some algorithms may be easier to implement if you have previous pointers.



Edge Cases

- Many algorithms in our code work correctly in **almost all** cases.
- **Edge cases** are the special circumstances that need specific code.
- Common edge cases:
  - An empty list (or a full one!)
  - An operation on the first or last node
  - If a value is at some minimum (often zero), some maximum, or **`nullptr`**.

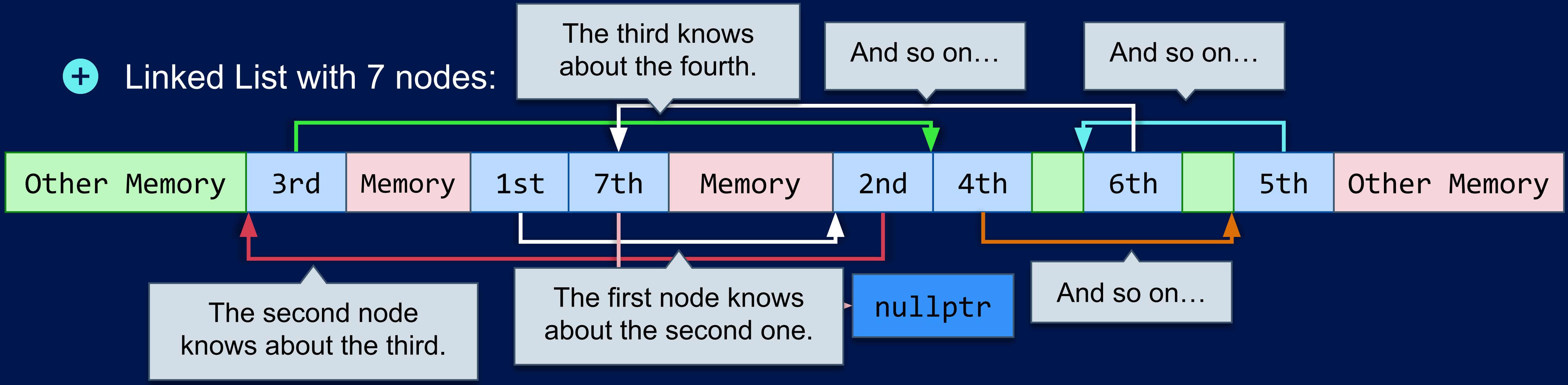
```
// Example: RemoveNode(Node* target)
if (target == head)
    // Edge case to remove the first node
else if (target == tail)
    // Edge case to remove the last node
else
    // Code for any "in-the-middle" nodes

// Any "universal" code that should always execute
```

Many algorithms will have a setup kind of like this.

Visualizing Memory Can Be a Challenge

- Array with 7 contiguous elements:

- Linked List with 7 nodes:

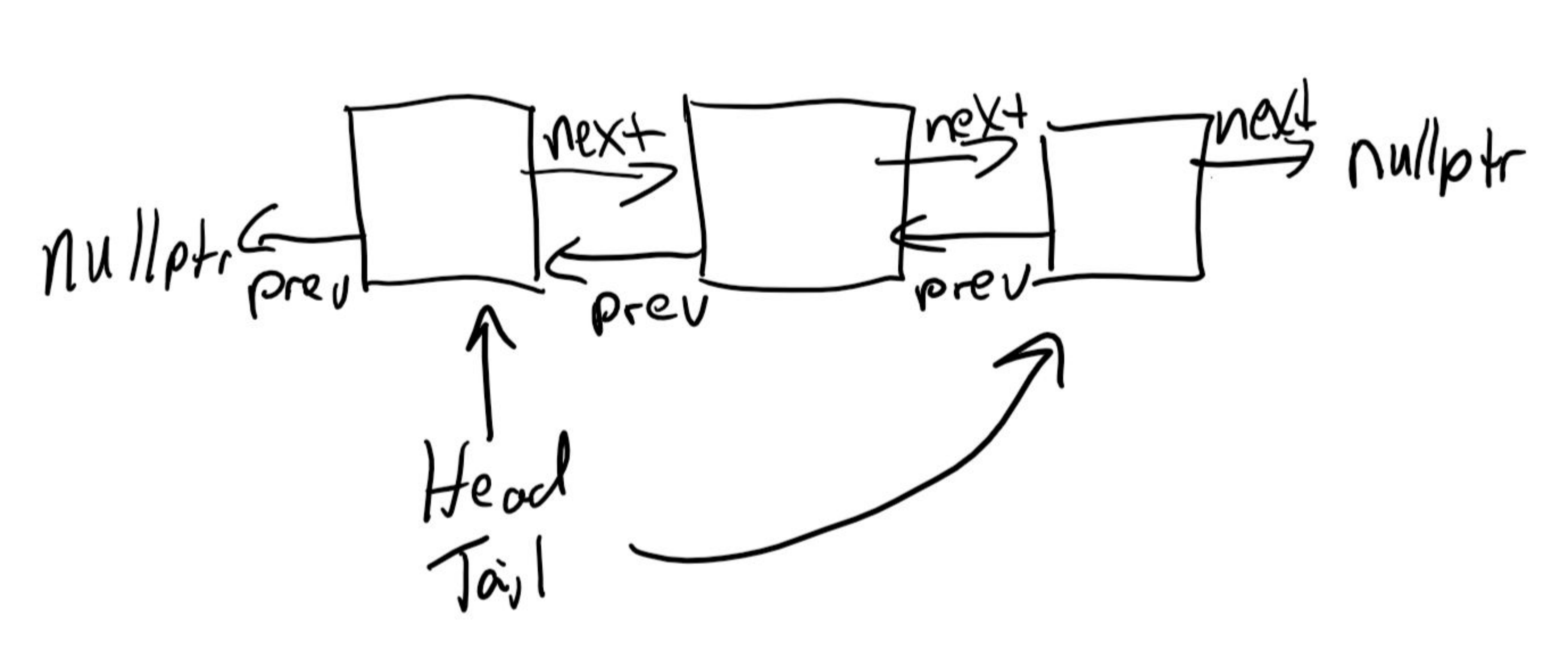


Keeping track of all of this as a human being can be difficult!

Diagrams are a necessity!

Draw Diagrams to Plan and Debug!

- You don't have to be "good enough" to remember all of it.
- Draw a diagram to represent the current state of your list.



- Use a diagram like this with **every** algorithm
- Walk through your code, updating this diagram each step of the way.
- If your code has a bug in it, you'll "break" your diagram.

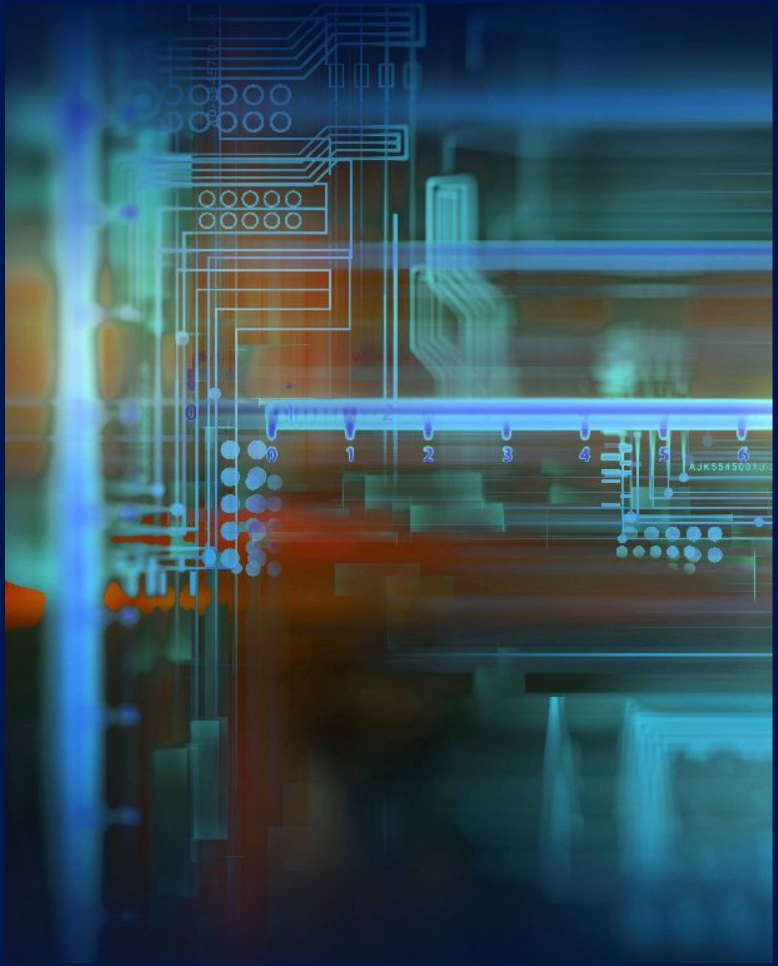
Other Operations Beyond These

- We’ve looked at the most common operations.

- You could create more, depending on what your program needs:
  - Combine two lists of nodes.
  - Split a list into multiple parts.
  - Remove every **other** node.
  - Insert <X> nodes after/before an existing node.

- These “advanced” operations can utilize existing functionality.
  - Set up the basic operations first, then build upon them.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.