



CS 151

Advanced C++ Programming

Module 14 – Linked Lists

Chris Merrill
Computer Science Dept.
cmerrill@miracosta.edu

Agenda

- Linked Lists – Nodes, Operations, and Templates
 - Singly-Linked List
 - Doubly-Linked Lists
- Review of Homework and Lab
- Quiz 4 (Chapters 14 – 16)

Topics

17.1 Introduction to the Linked List ADT

17.2 Linked List Operations

17.3 A Linked List Template

17.4 Recursive Linked List Operations

17.5 Variations of the Linked List

17.6 The STL `list` Container

Introduction to Linked Data Structures

A *linked data structure* consists of capsules of data known as *nodes* that are connected via *links*.

- Links can be viewed as arrows and thought of as one-way passages from one node to another.
- In C++, nodes are objects of a “node” structure:
 - The data in a node is stored via data members.
 - The links are realized as pointers.
 - *Therefore, a link in a node is a data member of the node structure itself.*

A Simple Linked List Class

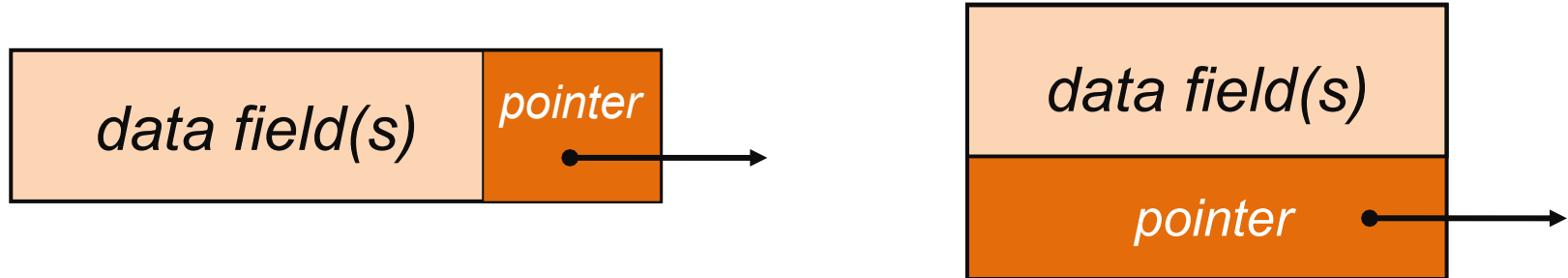
Each node is typically illustrated as a box containing one or more pieces of data and a link:

- Data is stored as a data member (or members) of the node, represented as information within the node "box".
- Links are illustrated as arrows that point to the next node in the list – the node to which they "link".

Node Organization

A node contains:

- data: one or more data fields – may be organized as variables, structures, objects, etc.
- a pointer that can point to another node
- two representations are used in this lecture:



Linked Lists Using C++

The simplest linked data structure is a *singly linked list*:

- a linked list consists of a single chain of nodes, each connected to the next by a link (pointer).
 - The first node is called the *head* node.
 - The last node serves as a kind of *end marker* usually implemented by using **nullptr**.

Detecting the End of a Linked List

By setting the link data member to `nullptr`:

- code can test whether a node is the last node in the linked list.
- note: this is tested using `==` and `!=`

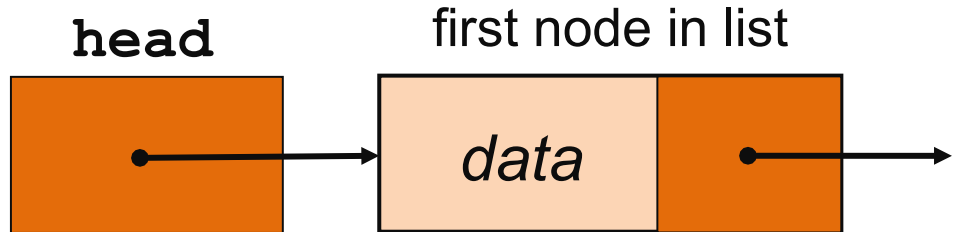
The “head” of a List

The “head” of a linked list is actually a *pointer* to the first node of the list. It is not a real node, as it does not have a data section.

- A list with no nodes is called the *empty list*.
- In this case the list’s head is set to **nullptr**.

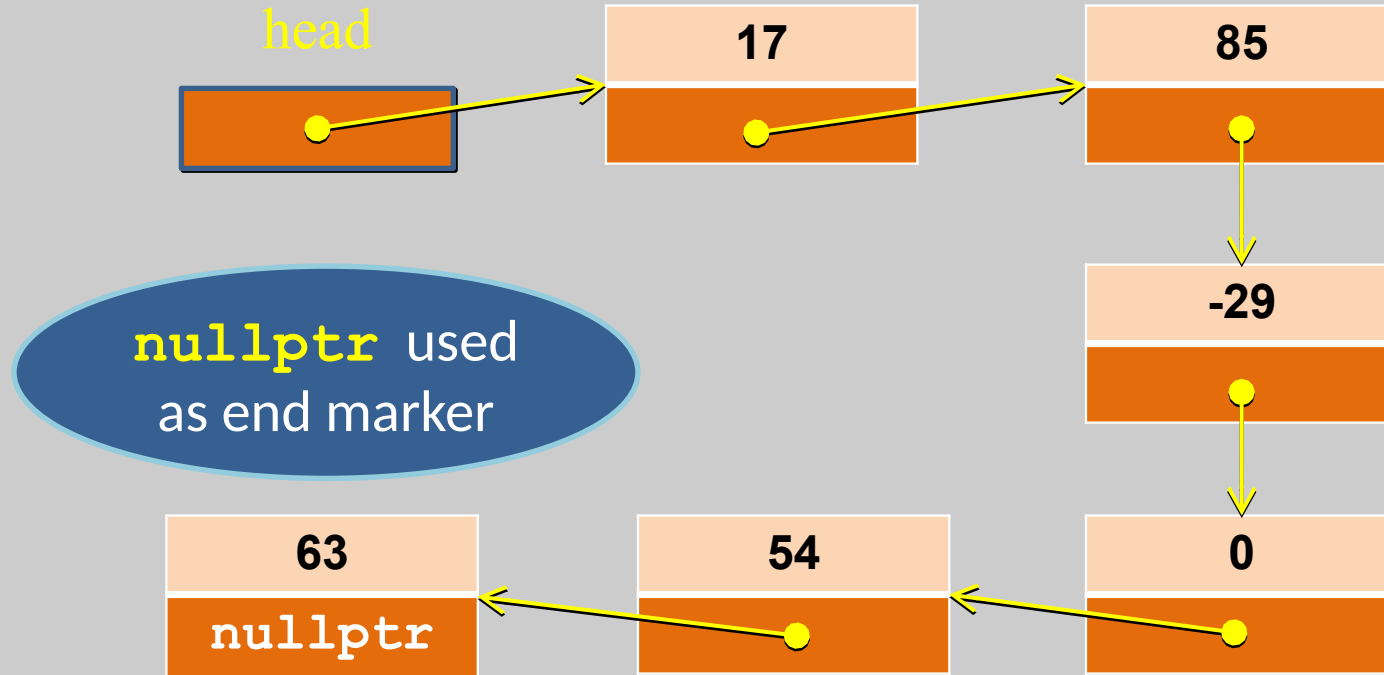


empty list



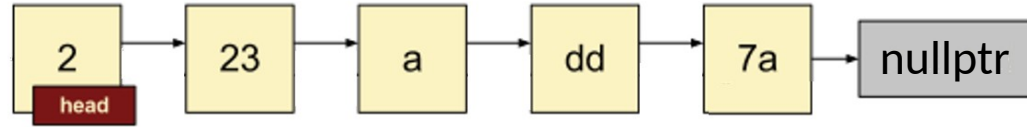
non-empty list

Nodes and Links in a Linked List



Linked Lists In C++ Are Not Contiguous In Memory

Singly Linked List



Array



A Simple Linked List Object?

Because linked lists are not continuous in memory, there is not a single linked list “object” which contains all the nodes in the linked list.

- Rather, a linked list is a connected group of node objects which point to the next node in the chain.
- Once the head node can be reached, every other node in the list can be reached.

C++ Implementation

Implementation of nodes in C++ uses a *structure* with a pointer to a structure of the same type, a *self-referencing* data structure:

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

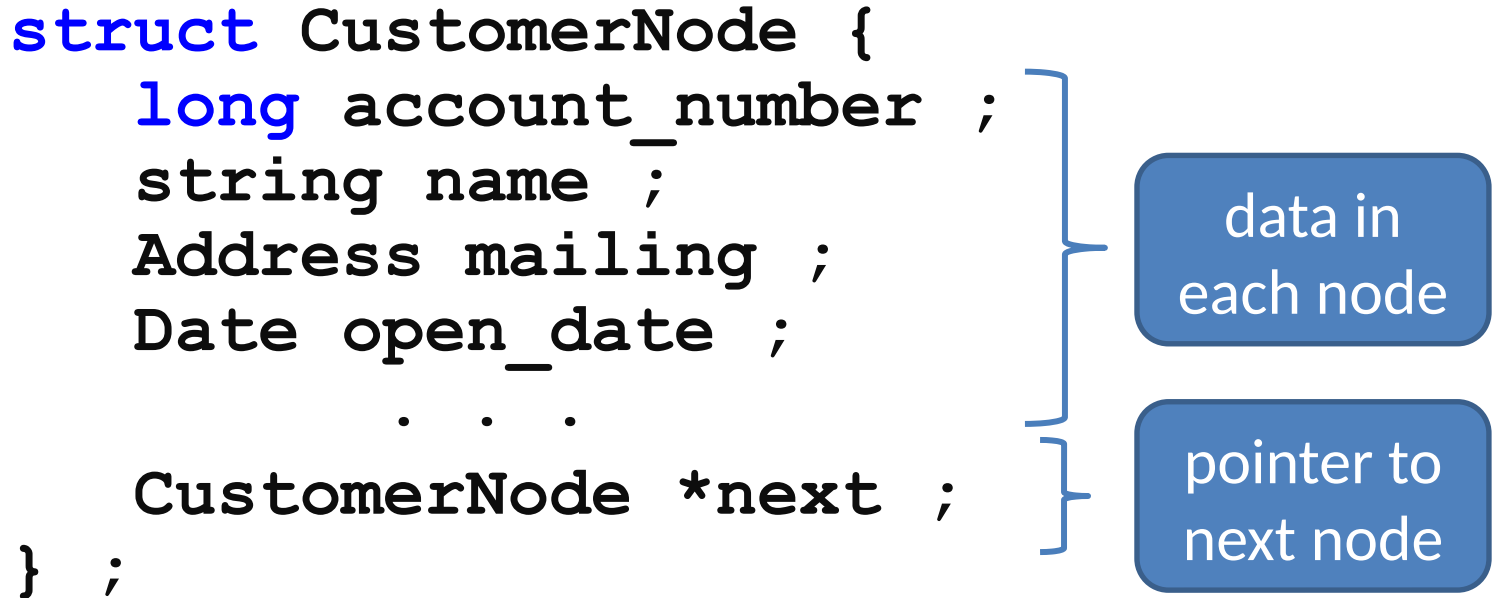
Huh?



C++ Implementation

The “data” in a node can be a number, character, class, structure, or any combination.

```
struct CustomerNode {  
    long account_number ;  
    string name ;  
    Address mailing ;  
    Date open_date ;  
    . . .  
    CustomerNode *next ;  
} ;
```



data in
each node

pointer to
next node

Creating an Empty List

Using our simple definition of a node, define a pointer for the head of the list as follows:

```
ListNode *head = nullptr ;
```

- The head pointer is initialized to `nullptr` to indicate that this is an empty list.

head



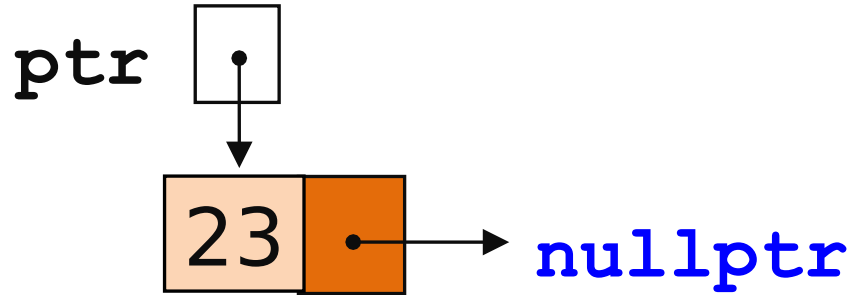
C++ Implementation

Nodes can be equipped with constructors:

```
struct ListNode {  
    int data ;  
    ListNode *next ;  
    ListNode(int d, ListNode *p = nullptr)  
        {data = d ; next = p ;}  
} ;
```


Creating a Node

```
ListNode *ptr ;  
int num = 23 ;  
ptr = new ListNode(num) ;
```



Traversing a Linked List

If a linked list already contains nodes, it can be traversed as follows:

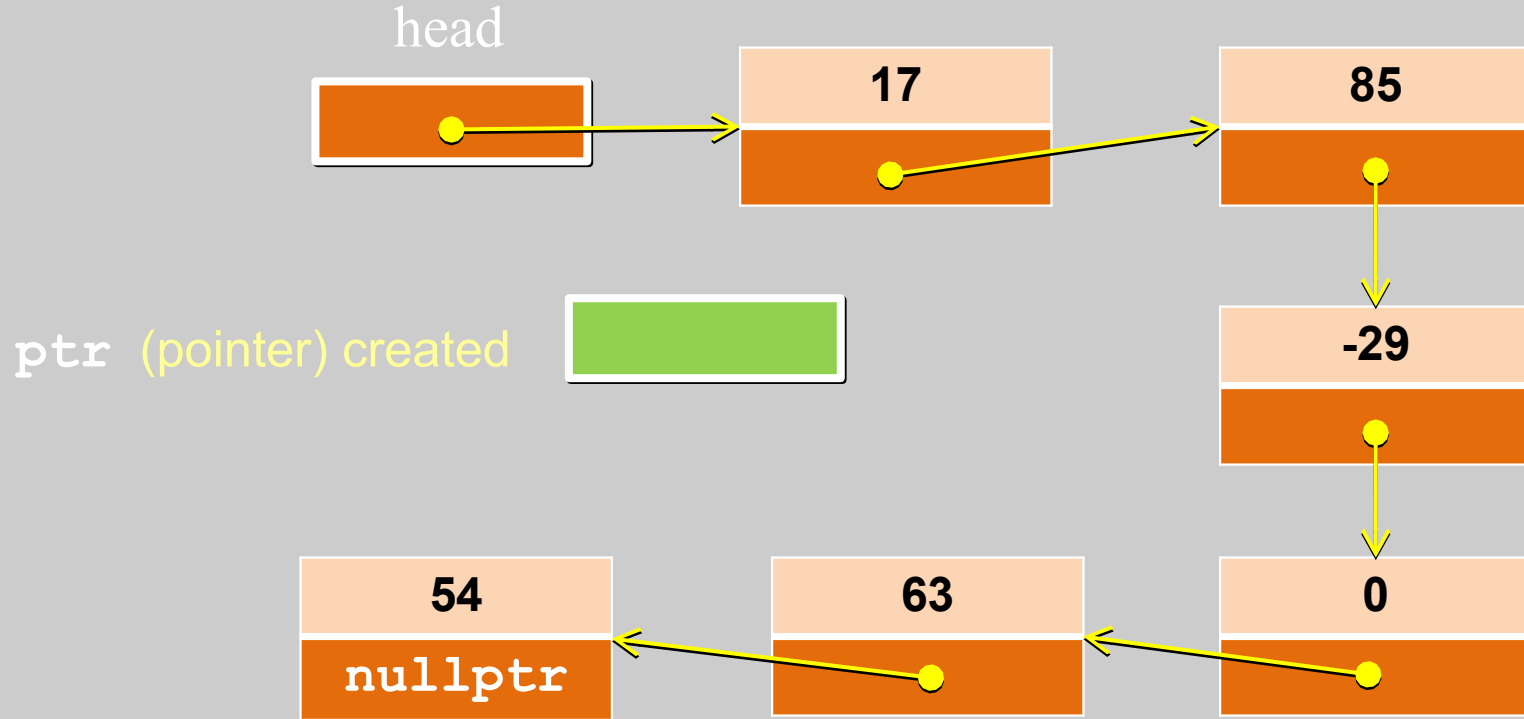
- Set a local variable (say **ptr** of type **ListNode***) to the value stored by the “head” node. After accessing the first node, the accessor function for the **next** data member will provide the location of the next node in the list.
- Repeat this until the location of the “next” node is equal to **nullptr**.

Traversing a Linked List

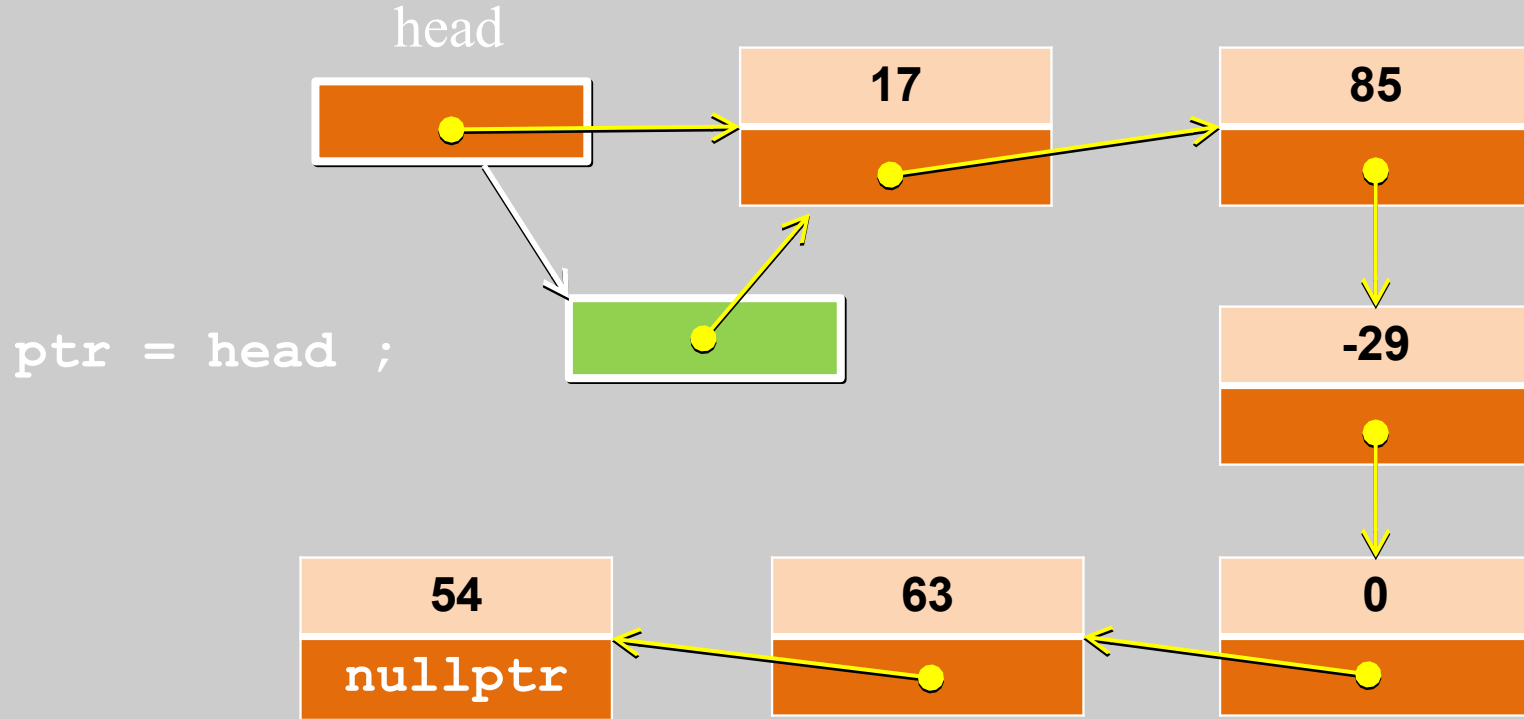
Basic algorithm (pseudocode) of traversal:

- *set a node pointer **ptr** to the **head** pointer*
- *while **ptr** is not **nullptr***
 - *process data*
 - *set **ptr** to the successor of current node*
- *end while*

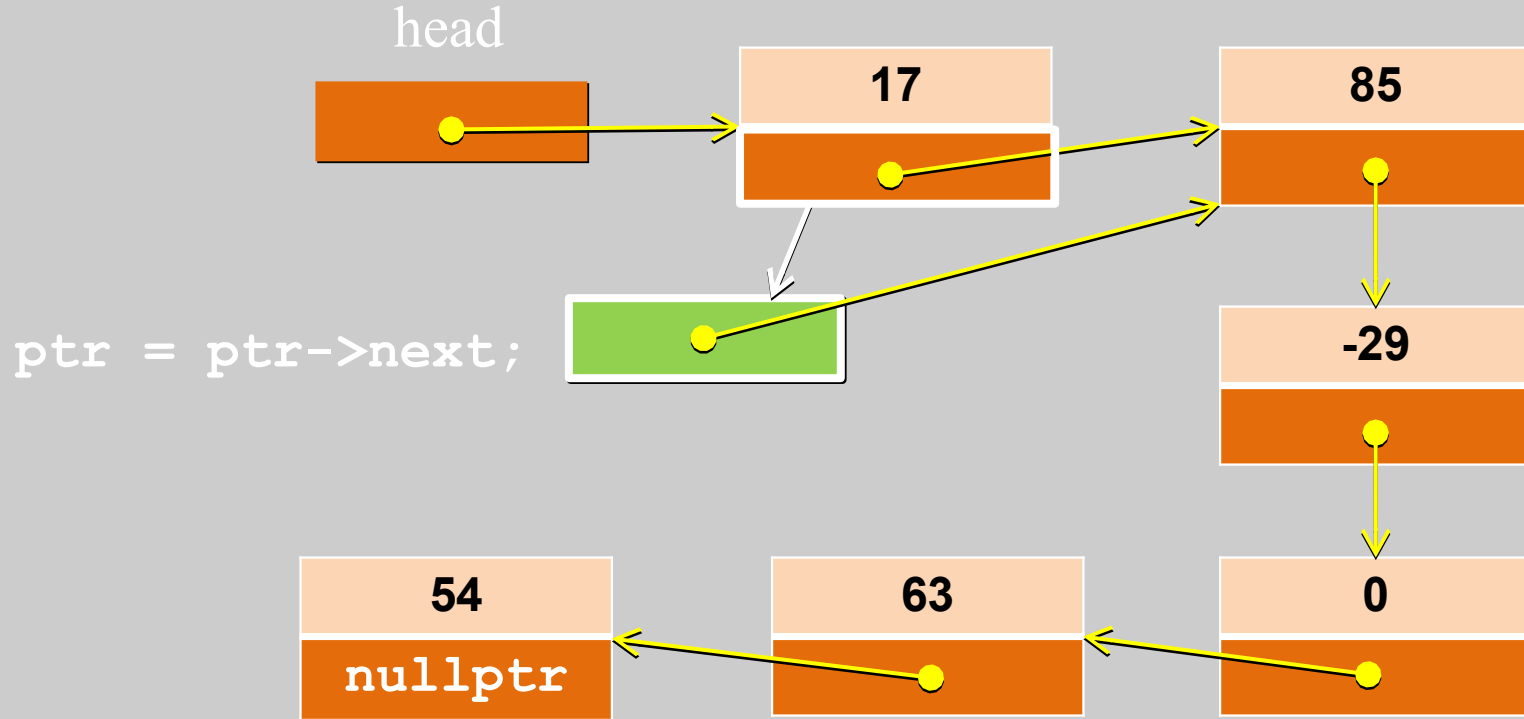
Traversing a Linked List



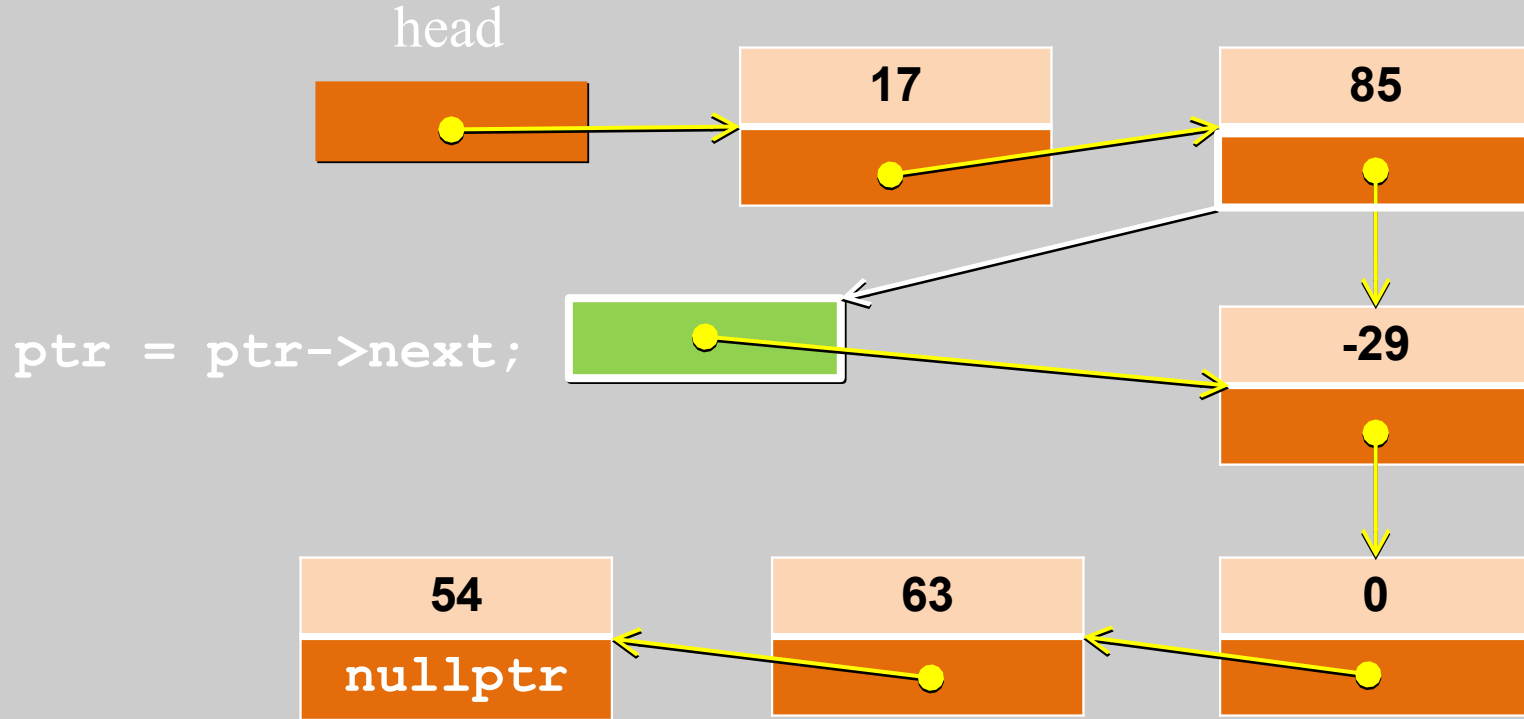
Traversing a Linked List



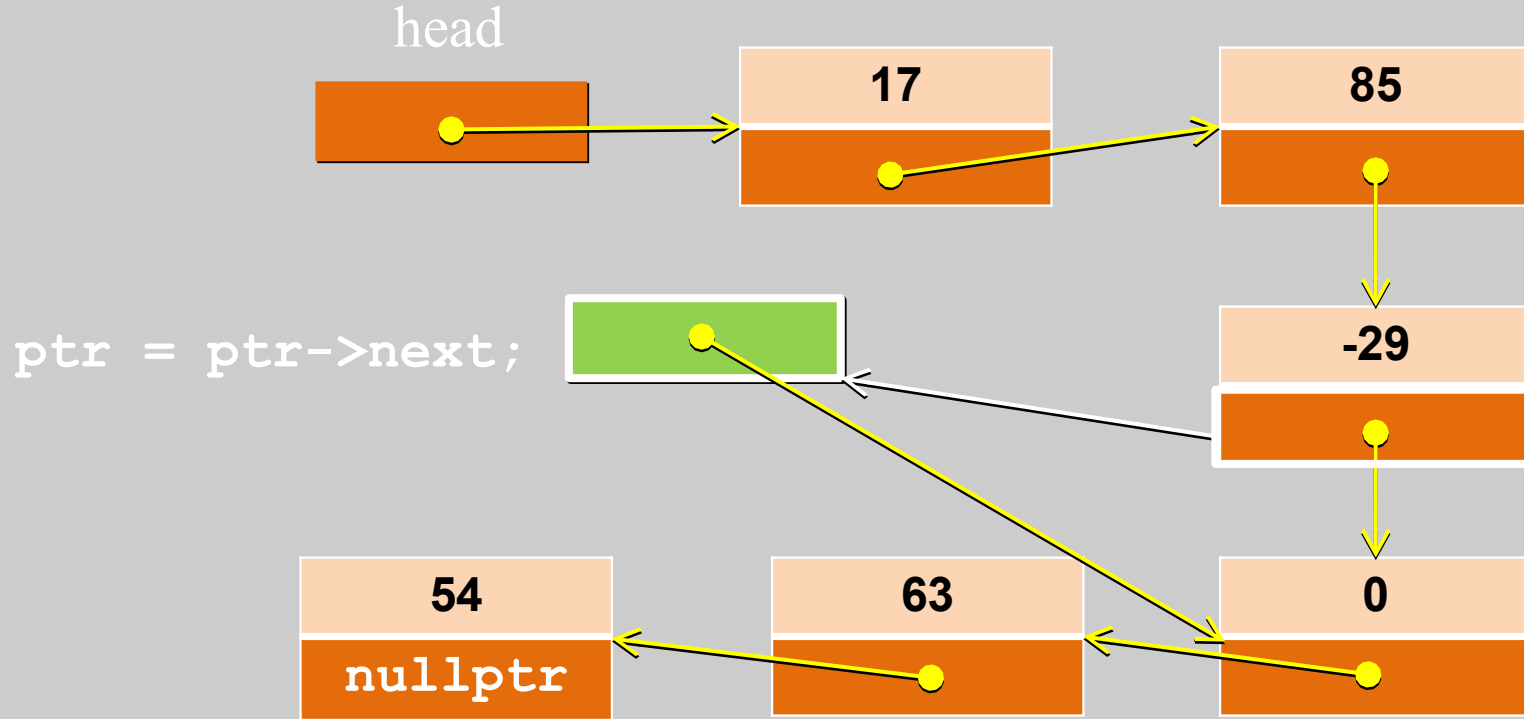
Traversing a Linked List



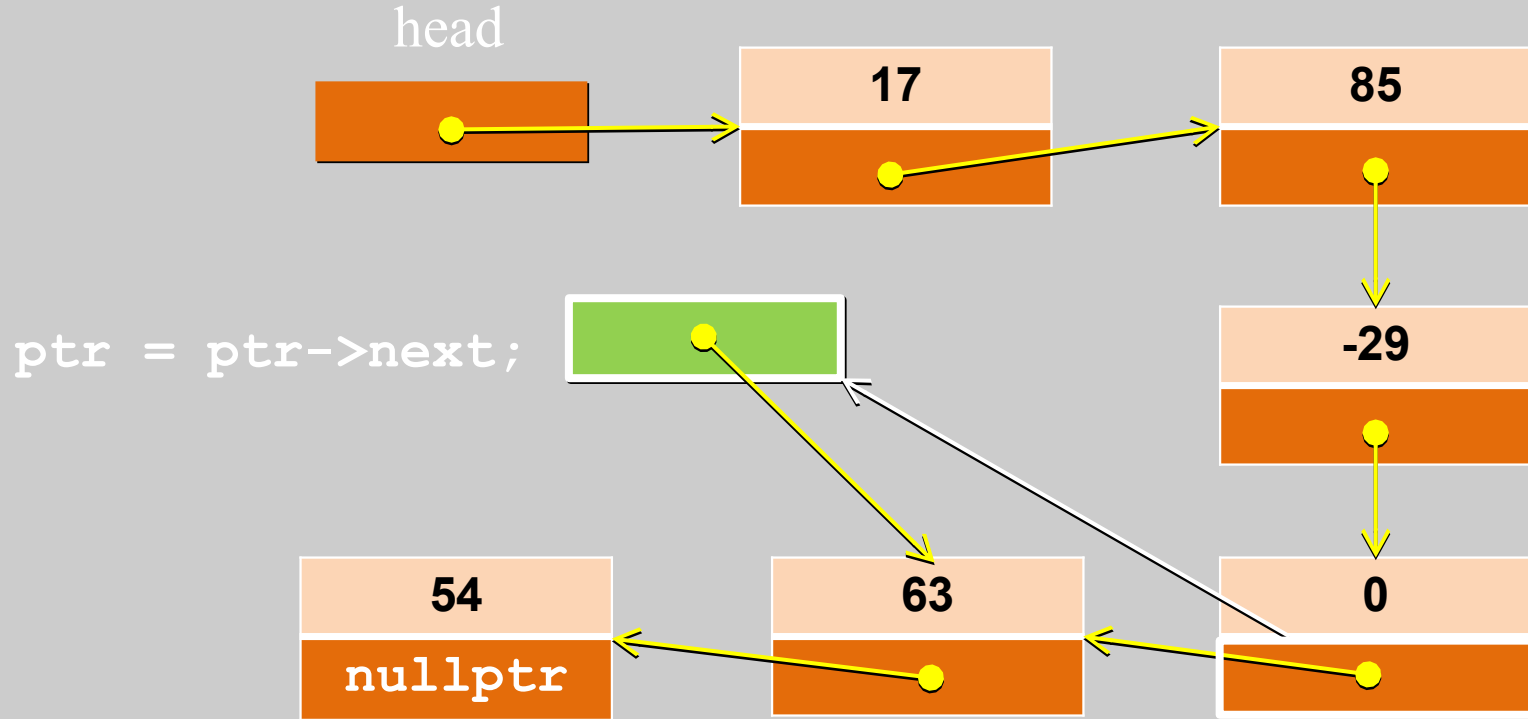
Traversing a Linked List



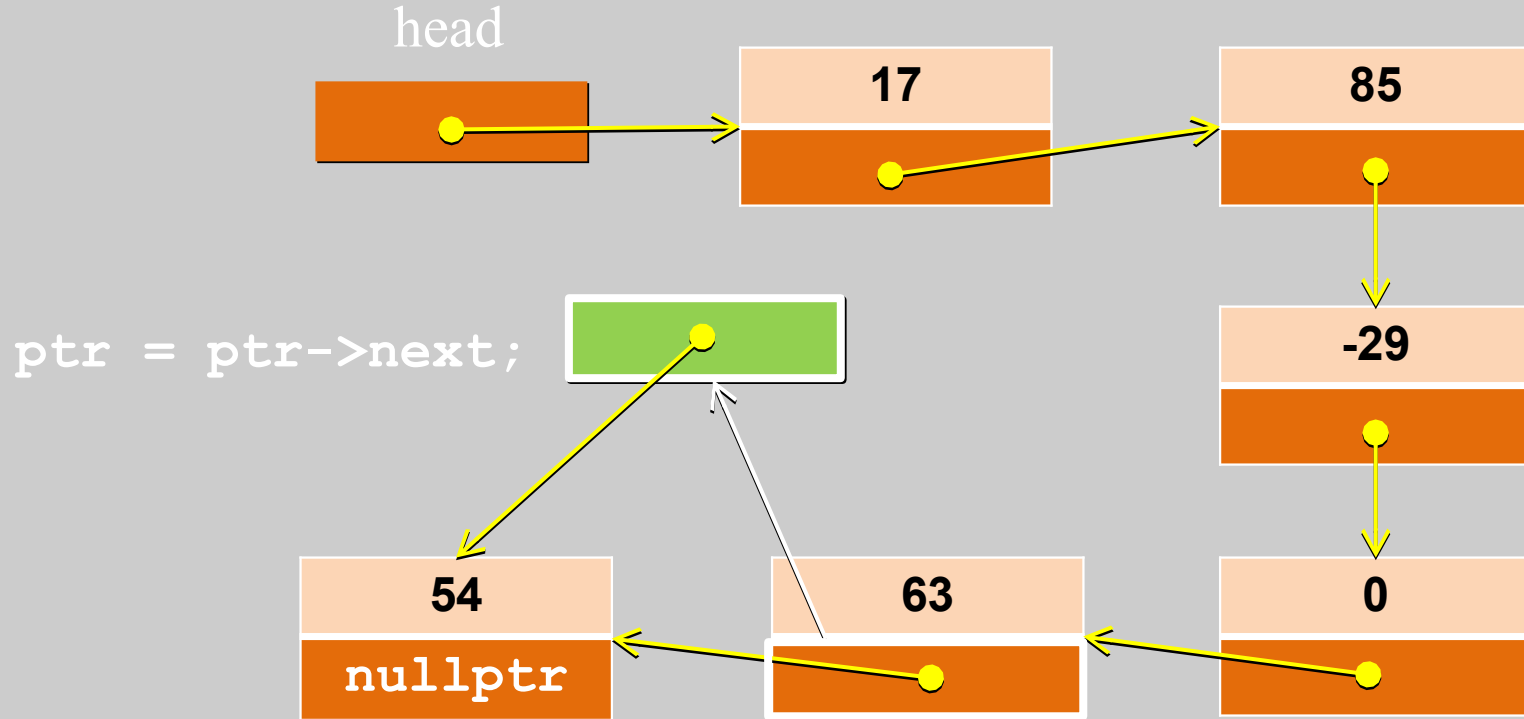
Traversing a Linked List



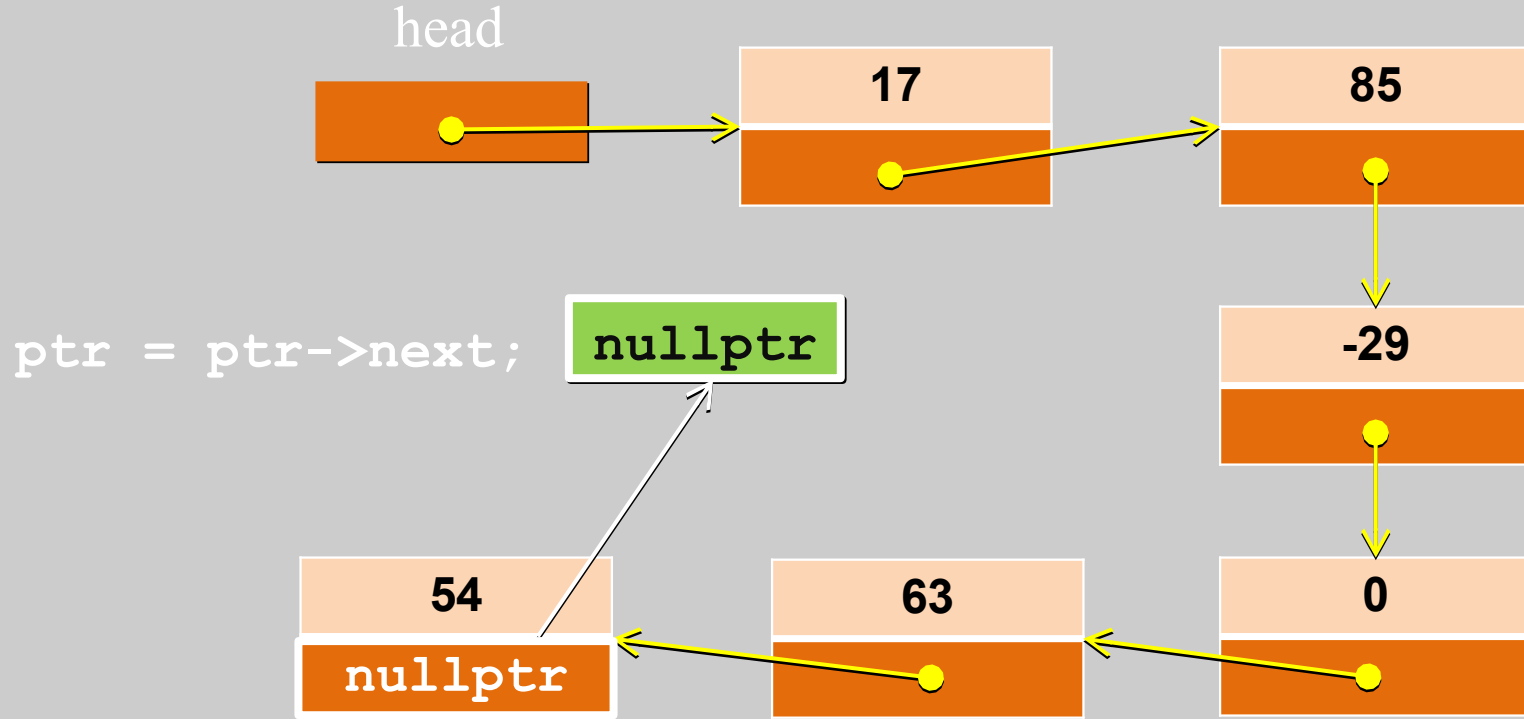
Traversing a Linked List



Traversing a Linked List



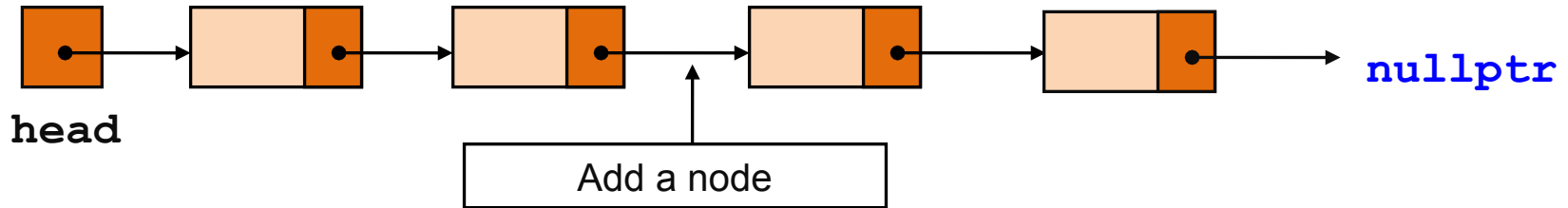
Traversing a Linked List



Linked Lists

Nodes can be added or removed from the linked list easily during a program's execution.

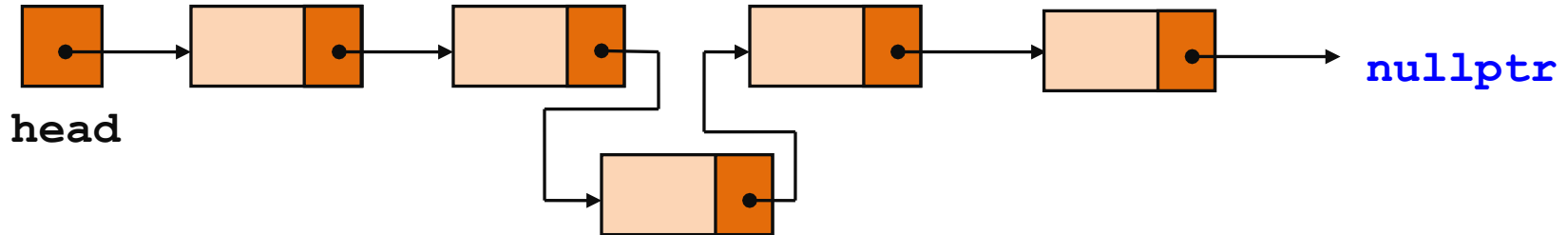
Addition or removal of nodes can take place at beginning, end, or middle of the list.



Linked Lists vs. Arrays and Vectors

Unlike arrays and vectors, linked lists can “grow” and “shrink” without moving lots of data, since they don’t occupy a contiguous area in memory.

- Insertion or removal of a node *anywhere* in the list can be done by simply manipulating pointers.



Adding a Node to the Front of a Linked List

To add a node to the start of the linked list (making the new node the first node on the list).

- Before the insertion, the variable **head** gives the location of the first node of the list.
- When the new node is created, its **next** link is set to the current contents of **head**.
- Then **head** is set to the address of the new node.

Adding a Node to the Front of a List

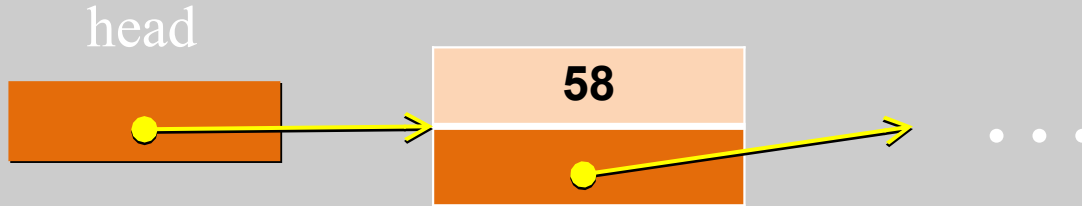
```
int value = 35 ;  
head = new LinkNode(value, head) ;
```

Does this work if the list is initially empty?

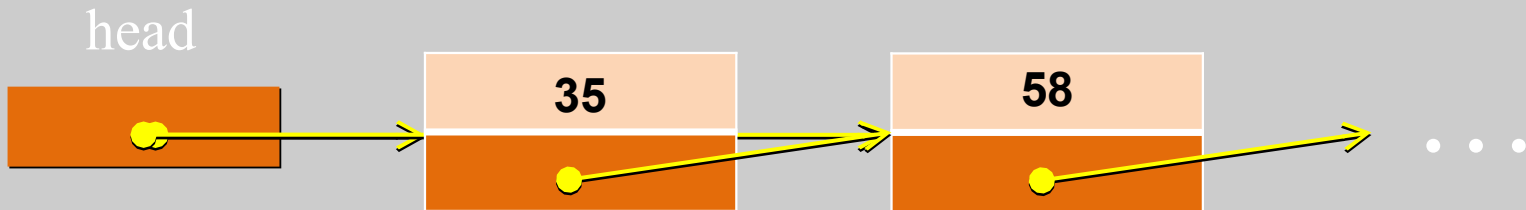
Adding a Node to the Front of a Sorted List

- Test to see if:
 - head pointer is **nullptr**, or
 - node value pointed at by head is greater than value to be inserted.
 - note: You must test in this order: The results are unpredictable if the second test is attempted on an empty list.
- Create new node, set its **next** pointer to **head**, then point head to it.

Adding a Node to the Front of a List



```
head = new ListNode(35, head) ;
```



Deleting a Node from the Front of a Linked List

To remove the first node from a linked list:

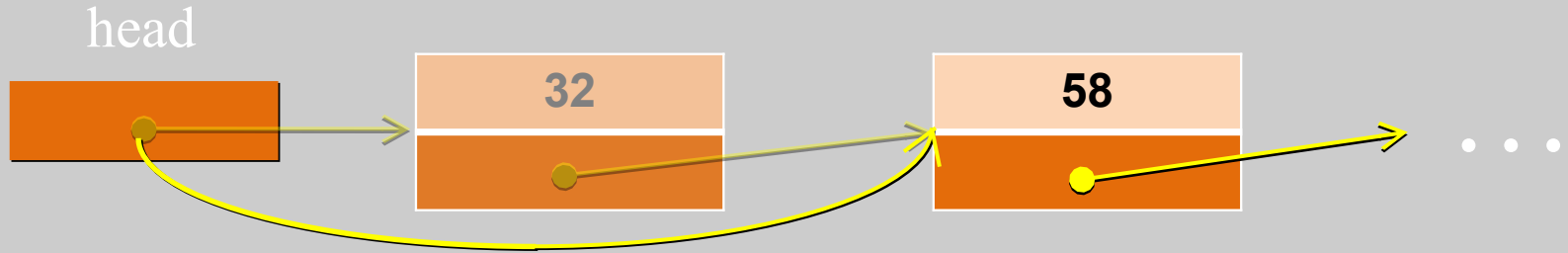
- Change the **head** variable to point to the node that is currently the *second* node in the list.
 - If there is no second node, then the **head** variable should be set to **nullptr**.

Deleting a Node from the Front of a List

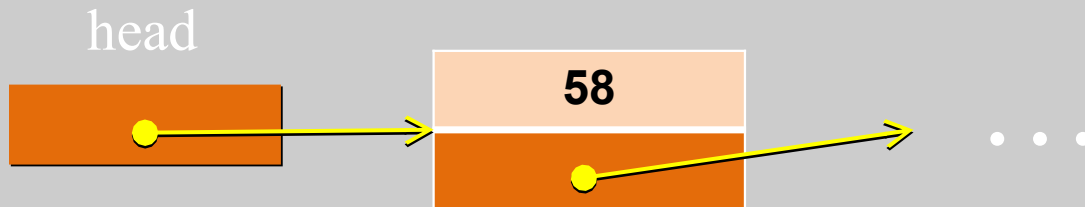
Why in
this
order?

```
if (head != nullptr) {  
    ListNode *ptr = head ;  
    int removed_data = ptr->data ;  
    head = head->next ;  
    delete ptr ;  
    (process removed_data ?)  
}
```

Deleting a Node from the Front of a List



```
int removed_data = head->data ;  
head = head->next ;  
(process removed_data ?)
```



Destroying (Erasing) a Linked List

If you want to remove *all* nodes appearing in the list:

- Use list traversal to visit each node.
- For each node,
 - Save the current value of head in a temporary pointer
 - Set the list's head to the next node in the list
 - Free the node's memory using the temporary pointer
- Loop until head is `nullptr`.

Linked List Operations

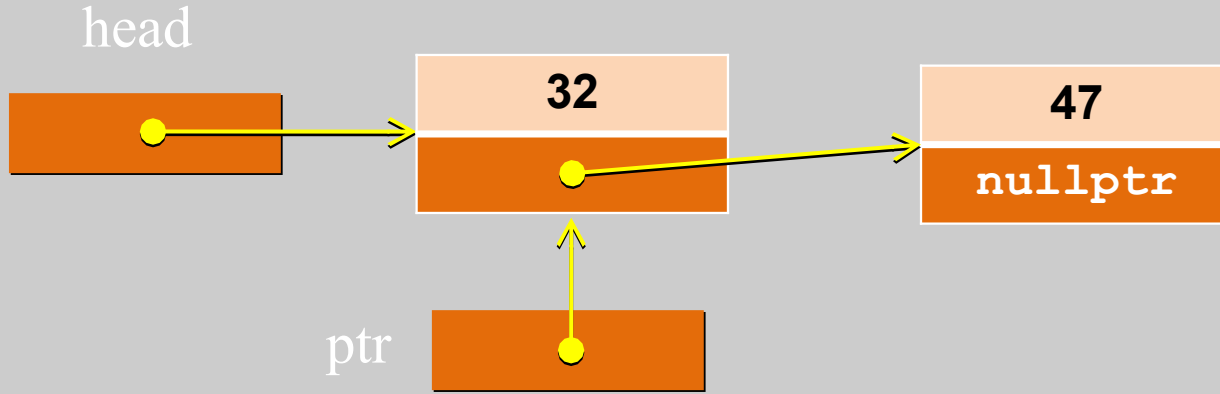
More basic operations:

- add a node to the list
- traverse the linked list
- delete/remove a node from the list
- delete/destroy the list

Adding an Item to the End of a List

- If the list is empty, set **head** to the new node:
`head = new ListNode(num) ;`
- If the list is not empty:
 - move a pointer **ptr** to the last node using a loop
`while(ptr->next != nullptr)`
`ptr = ptr->next ;`
 - then add a new node containing the item
`ptr->next = new ListNode(num) ;`

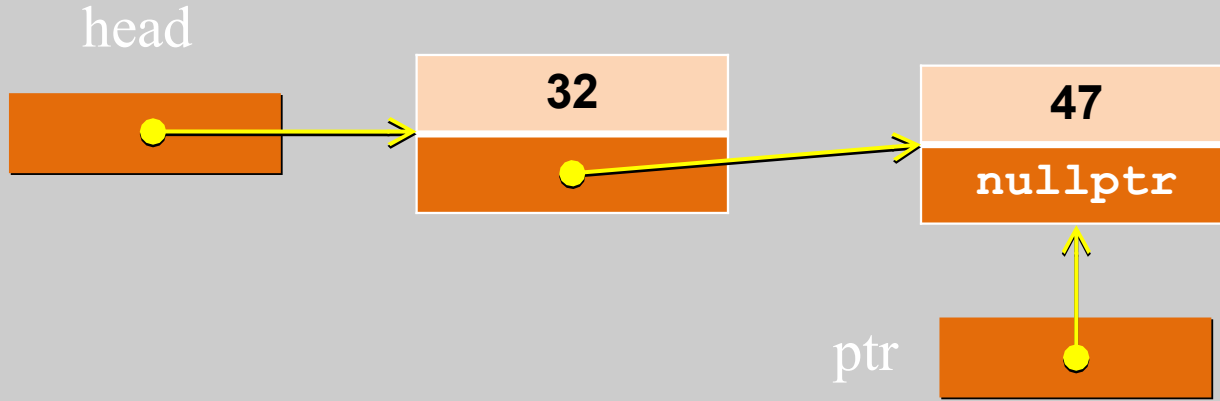
Adding a Node at End (Part 1 of 4)



```
ListNode *ptr = head ;
```

Find the last node in the list

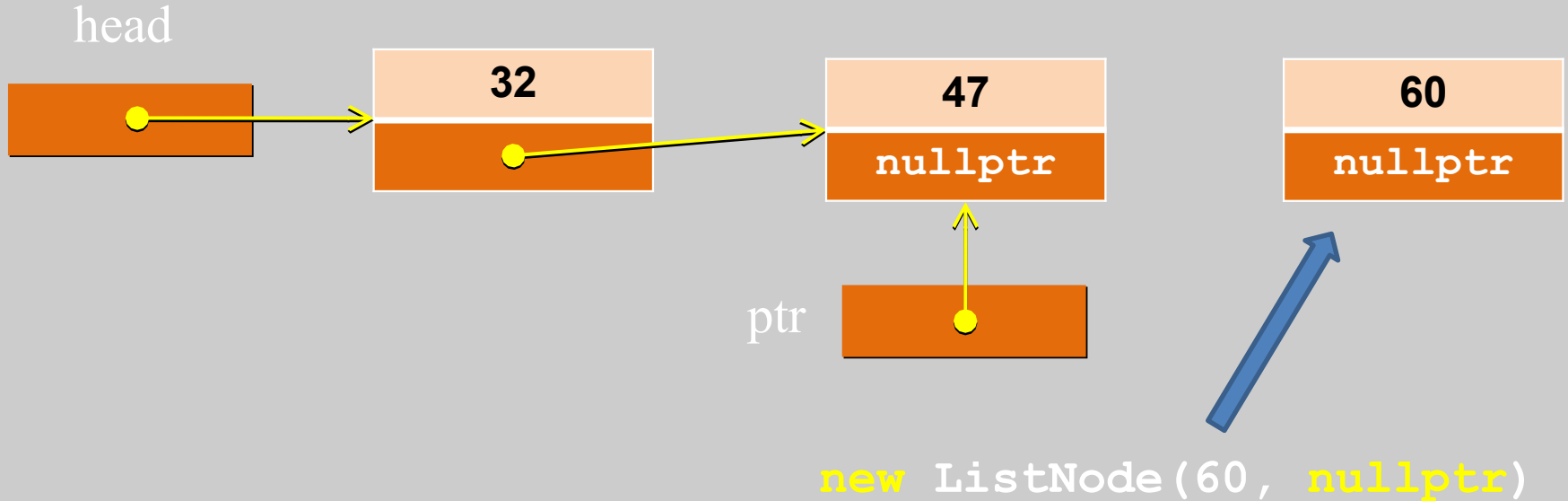
Adding a Node at End (Part 2 of 4)



```
while (ptr->next != nullptr)
    ptr = ptr->next ;
```

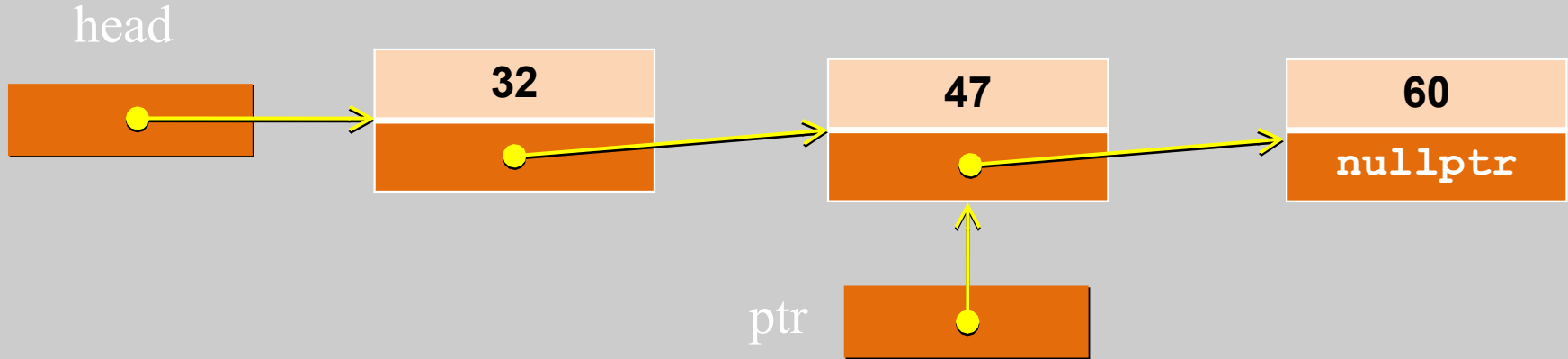
Find the last node in the list

Adding a Node at End (Part 3 of 4)



Create the new node

Adding a Node at End (Part 4 of 4)



```
ptr->next = new ListNode(60, nullptr) ;
```

Link *old* last new node to **new** last item

Maintaining a Sorted List

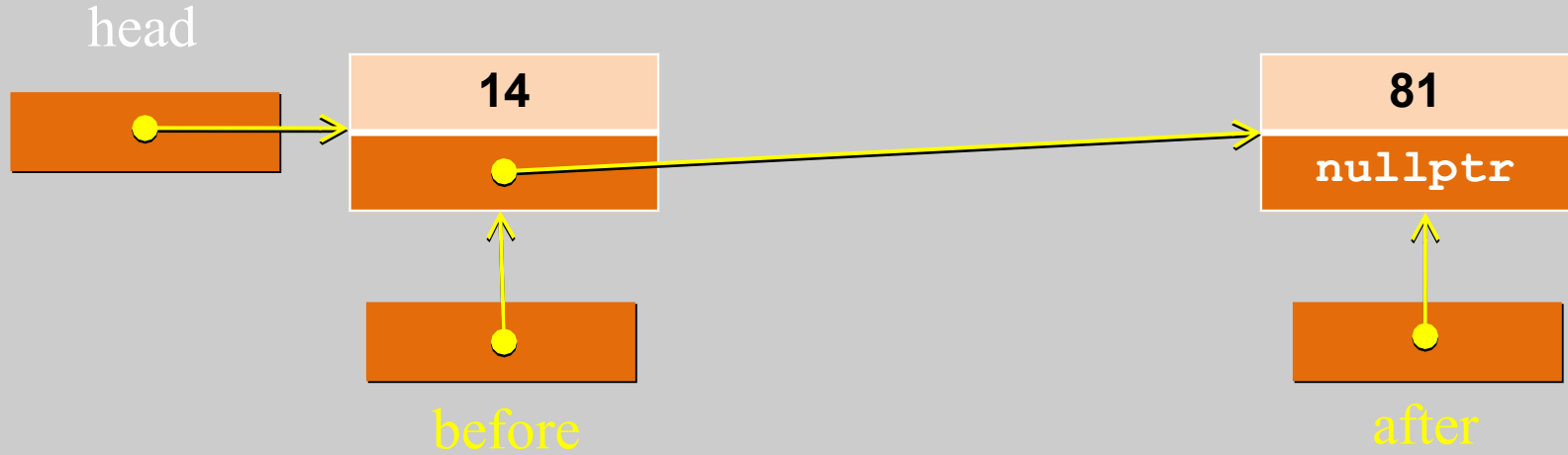
You may want to keep nodes in a linked list in sorted order according to their data fields.

- Adding or removing nodes from either end won't work.
- For adding, here are two possibilities (ascending order):
 - The add point is at the head of the list (because the item at the head is already greater than the item being added, or because the list is empty.
 - The add point is after an existing node in a non-empty list.

Inserting a Node after Head in a List

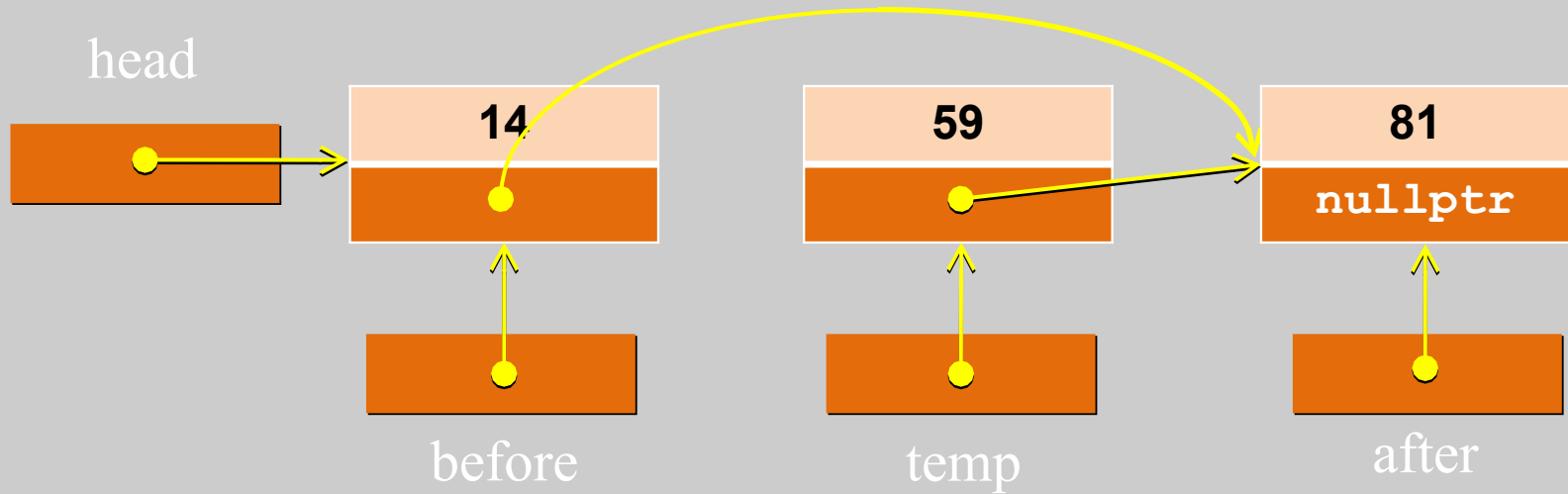
- This will require two pointers to traverse the list:
 - a pointer **after** to locate the node with data value greater than that of node to be inserted.
 - a pointer **before** to 'trail behind' **after** one node, pointing to node before point of insertion.
- The new node is inserted between the nodes pointed to by these pointers.

Inserting a Node (Part 1 of 3)



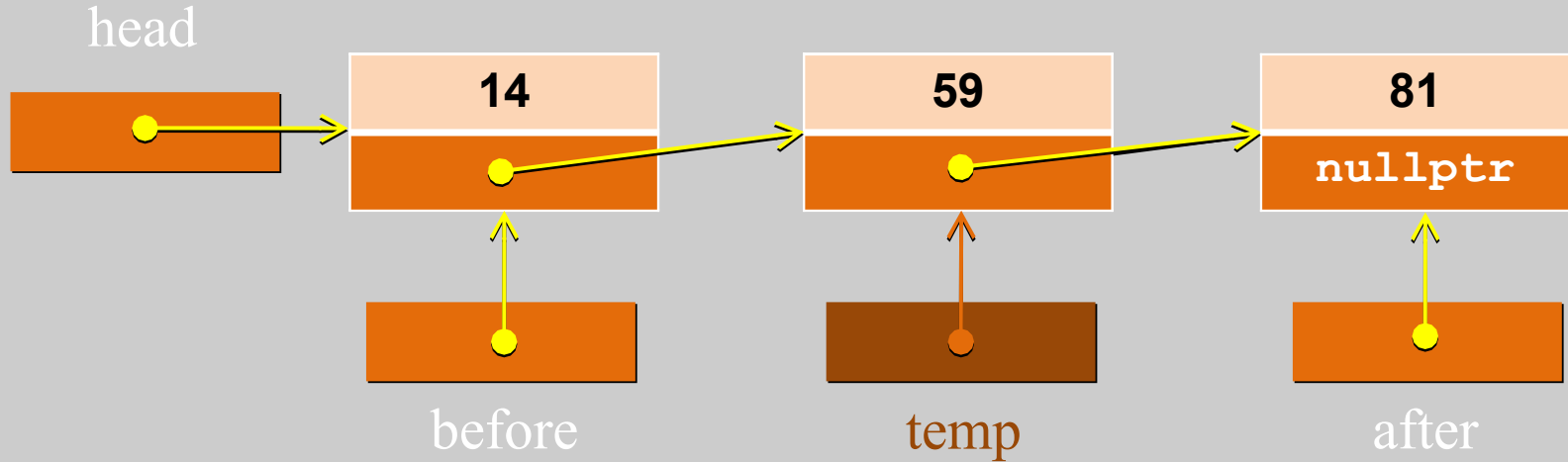
Preparing to add a **LinkNode** for 59

Inserting a Node (Part 2 of 3)



```
LinkNode *temp = new LinkNode(59, after) ;
```

Inserting a Node (Part 3 of 3)



```
before->next = temp ;  
Variable temp is then recycled.
```

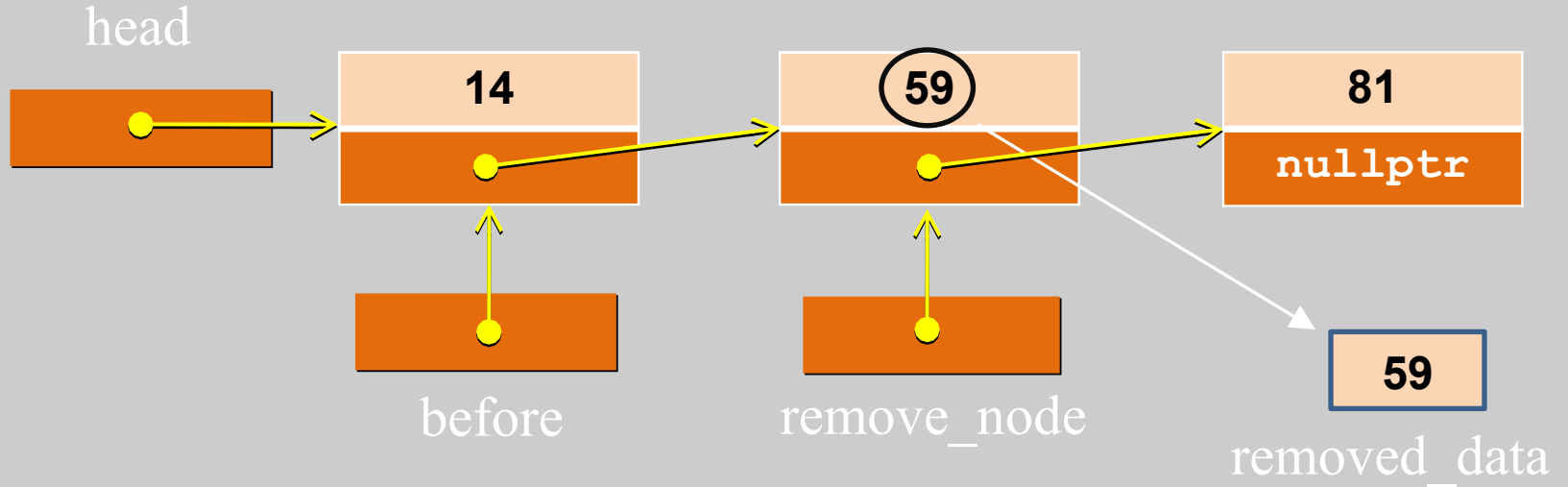

Removing an Element

Like adding, to remove a node from a linked list requires two pointers:

1. one, say **remove_node**, to point to the node to be removed, and
2. another, say **before**, to point to the node prior to the node to be removed.

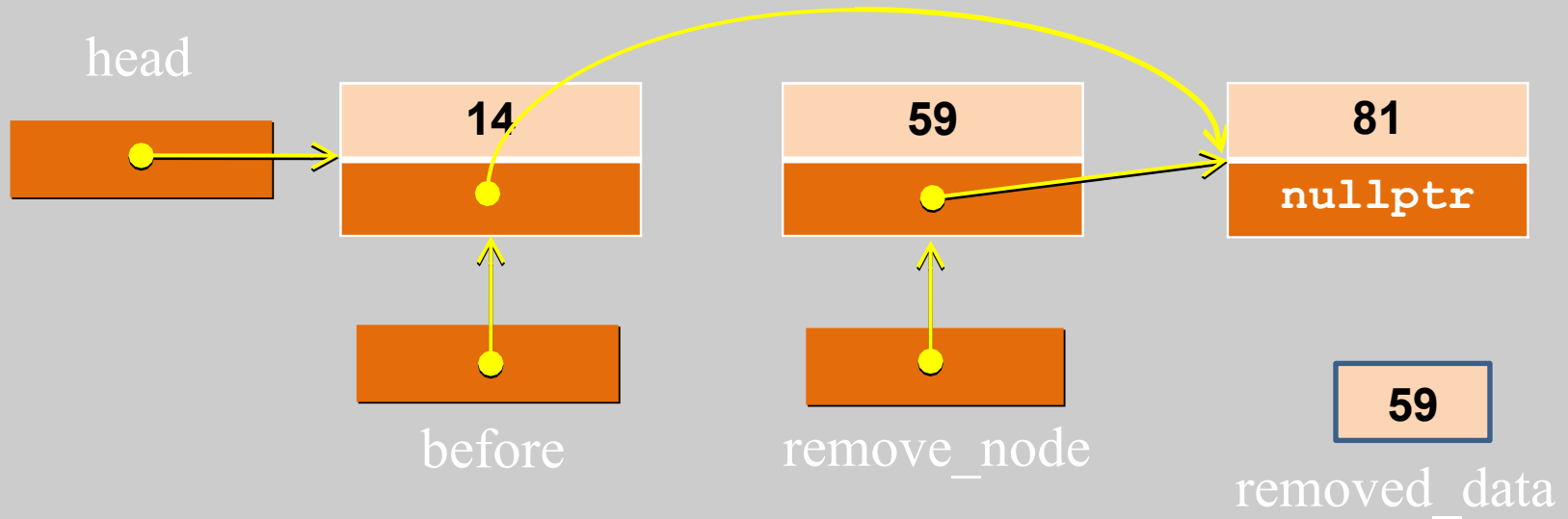
Save the data in the node to be removed, set the prior node's **next** pointer to the **next** pointer of the node to be removed, and delete the node.

Removing a Node (Part 1 of 3)



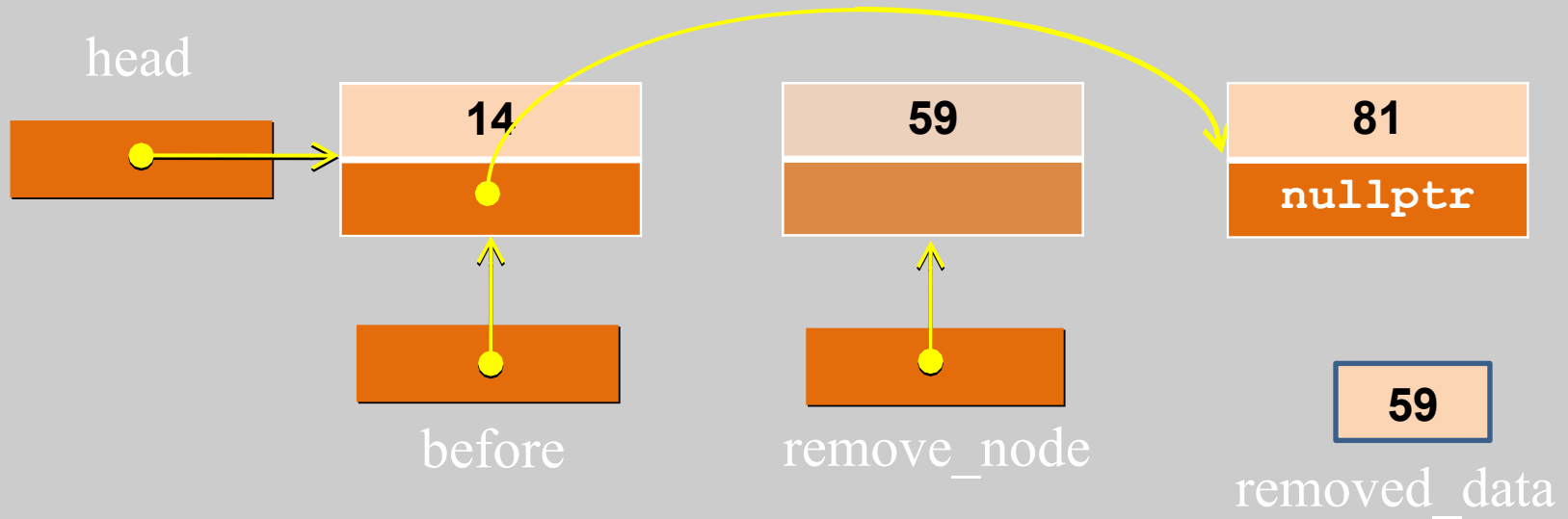
```
removed_data = remove_node->data
```

Removing a Node (Part 2 of 3)



Change pointer of prior node
`before->next = remove_node->next ;`

Removing a Node (Part 3 of 3)

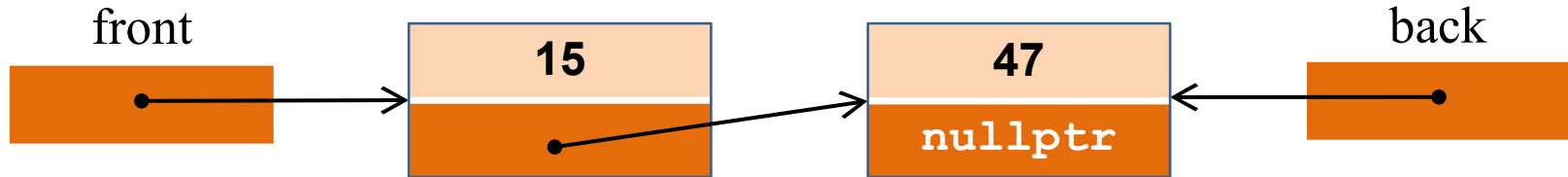


```
Delete victim from the heap  
delete remove_node ;
```

Adding a Pointer to the Back of a List

Sometimes another node is added to a linked list, this one at the back end of the list

- This allows easy additions at either end of the list
- The name “head” is usually changed to “front” and another called “back” is added:

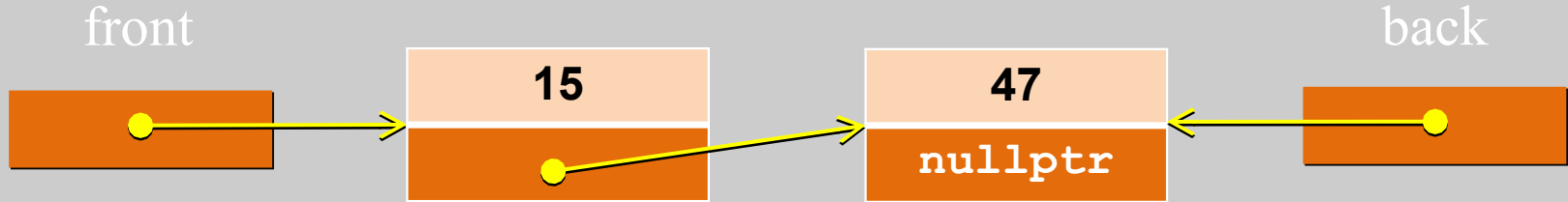


Adding a Back Pointer to a Linked List

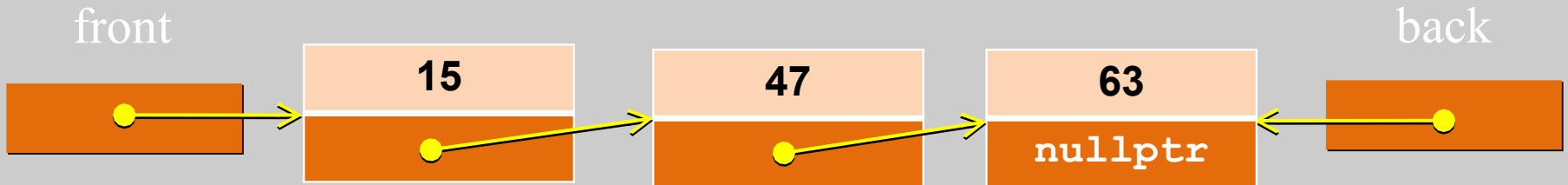
To add a node to the *end* or *back* of the linked list with a second pointer at the end:

- The variable **back** gives the location of the current last node of the list
 - Therefore, when the new node is created, the **next** of the current last node is set to the new node
 - Then **back** also is set equal to the new node

Adding to the Back of a List



```
back->link = newEntry  
back = back->link;
```

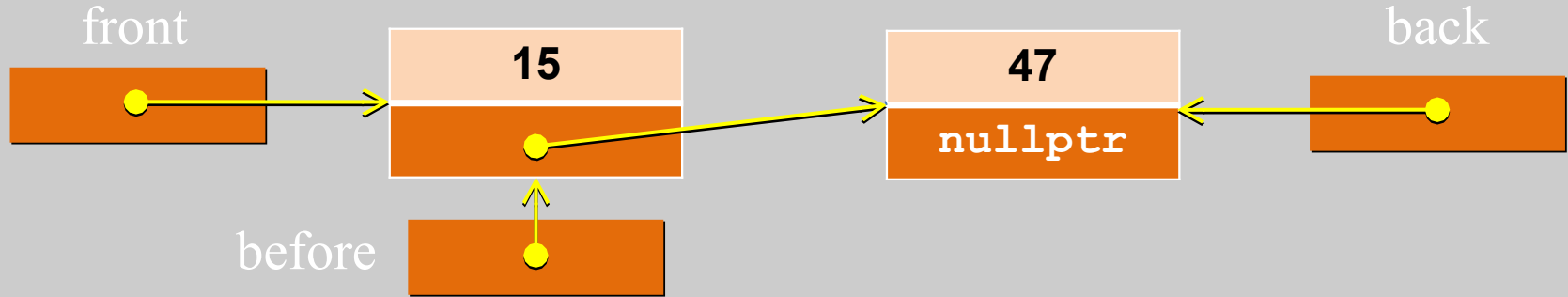


Deleting a Node from the Back of a List

Like deleting any node (except the front) from a singly linked list, we still require a pointer to the node before the node to be deleted.

- However, we don't need one to the end node, since that is what **back** is for.
- Set **before** to **head**.
- Loop until **before->next** is **nullptr**.

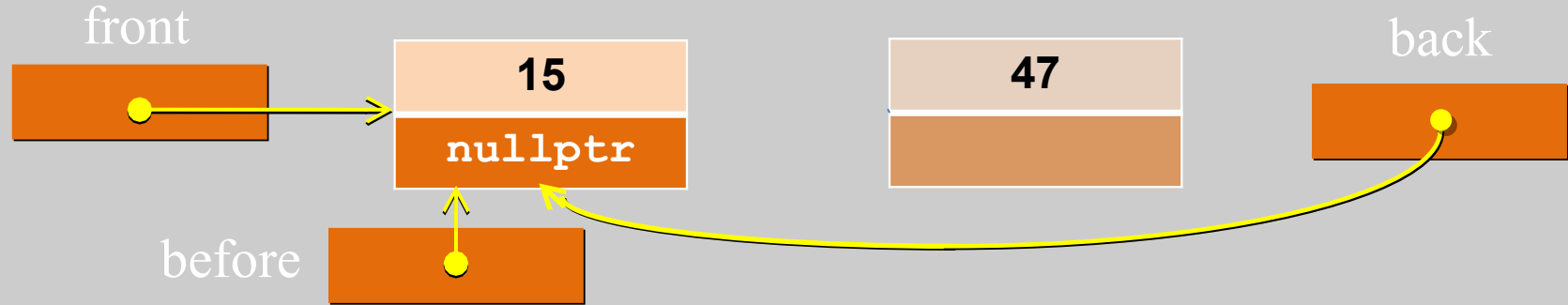
Deleting a Node From the End of a List (part 1)



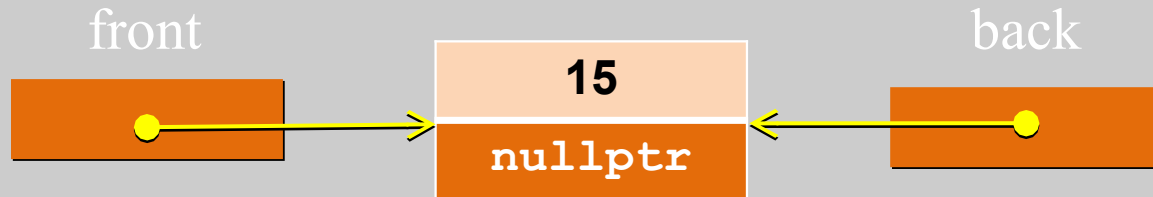
```
int removed_data = back->data ;  
before->next = nullptr ;  
delete back ;
```



Deleting a Node From the End of a List (part 2)



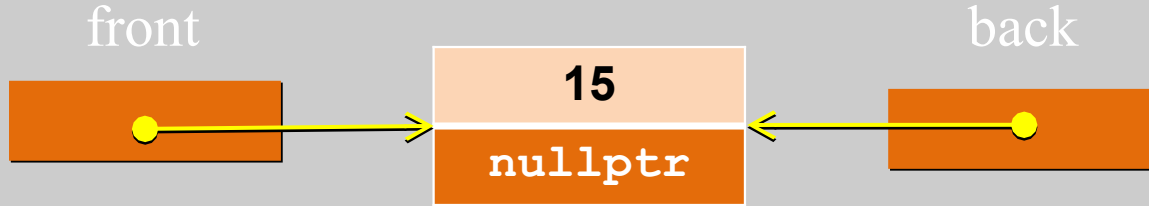
```
back = before ;  
(process? removed_data)
```



Removing the Only Node in a List with a Back

If there is only one item left in the list, then simply delete the node from the heap and set both **front** and **back** to **nullptr**

Deleting the Only Node from the List



```
int removed_data = front->data ;  
delete front ;  
front = back = nullptr ;  
    (process? removed_data)
```



A Linked List Template

A linked list template can be created by replacing the type of the data in the node with a type parameter, say T.

- If you are defining the linked list as a class template, then all member functions must be function templates.
- Implementation assumes use with data types that support comparison: `==` and `<=`.

Recursive Linked List Operations

- A non-empty linked list consists of a head node followed by the rest of the nodes.
- The rest of the nodes form a linked list that is called the tail of the original list.
- Many linked list operations can be broken down into the smaller problems by processing the head of the list and then recursively operating toward the tail of the list.

Recursive Linked List Operations

To find the *length* (number of elements) of a list:

- If the list is empty, the length is 0 (base case).
- If the list is not empty, find the length of the tail and then add 1 to obtain the length of the original list.

Other Recursive Linked List Operations

Add and remove operations can use recursion:

- General design considerations:
 - The base case is often when the list is empty.
 - The recursive case often involves the use of the tail of the list (*i.e.*, the list without the head). Since the tail has one fewer entries than the list that was passed into this call, the recursion eventually stops.

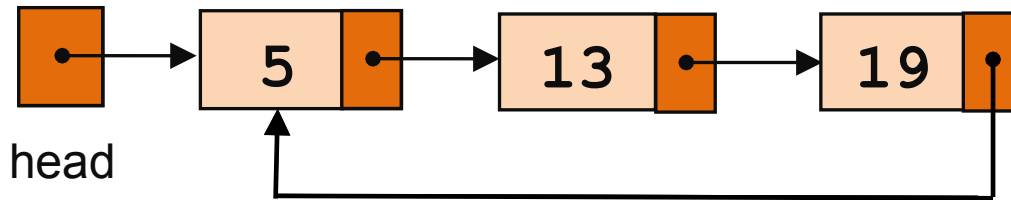
Limitations of Recursive Linked List Operations

Limitations of recursion:

- Recursive solutions typically require more processing and resources than iterative solutions.
- Recursive operations use a stack, which has a very limited size (restricting the number of calls) vs. iterative operations (no limits to number of iterations).

Variations of the Linked List

- *Circular linked list*: This is a singly-linked list in which the last node in the list points back to the first node in the list, not to **nullptr**.



Using a Circular List

When traversing a circular list:

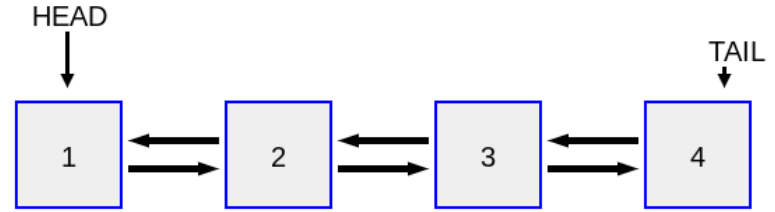
- the algorithm can't end with reaching a **nullptr** pointer, since there aren't any.
- instead, stop traversing when the pointer used to traverse the list returns back to the starting point of the traversal.

Doubly-Linked Lists

The linked lists we've seen so far allow movement in one direction – from the head to the back of the list.

- However, in *doubly-linked* lists, each node has two links:
 - one link that points to the *next* node in the list
 - another that points to the *previous* node
- This allows movement in both directions.

Doubly-Linked Lists

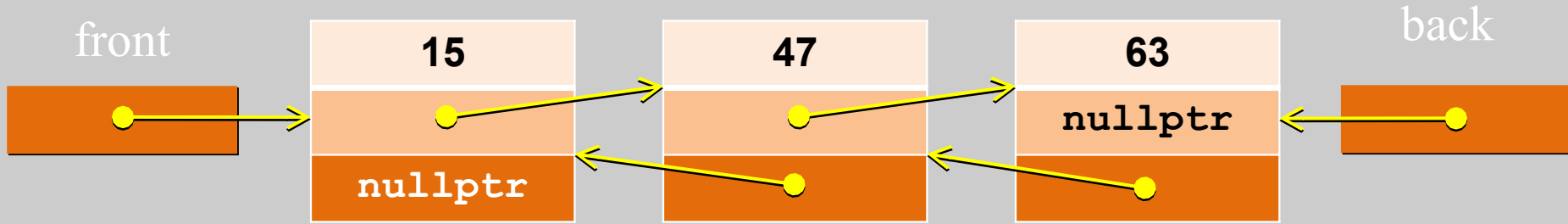


The node class for a doubly linked list can begin as follows:

```
struct TwoWayNode {  
    int data ;  
    TwoWayNode *previous ;  
    TwoWayNode *next ;  
}
```

- Functions in a doubly linked list class must be enhanced to accommodate the extra link.

A Doubly-Linked List Example



Doubly-linked list can be traversed in either direction

Adding a **TwoWayNode** to the Front of a Doubly-Linked List

Adding a node to the front looks a lot like the code we used to add a node to the front of a singly linked list:

```
TwoWayNode *new_node = back front  
    new TwoWayNode(data, nullptr, front) ;  
if (front != nullptr)  
    front->previous = newNode ;  
front = new_node ;
```

} new

Adding a **TwoWayNode** to the Front of a Doubly-Linked List

We also need to add code to update **back** if the list was originally empty:

```
if (back == nullptr)
    back = new_node ;
```

Adding To An Empty Doubly-Linked List (page 1 of 4)

front

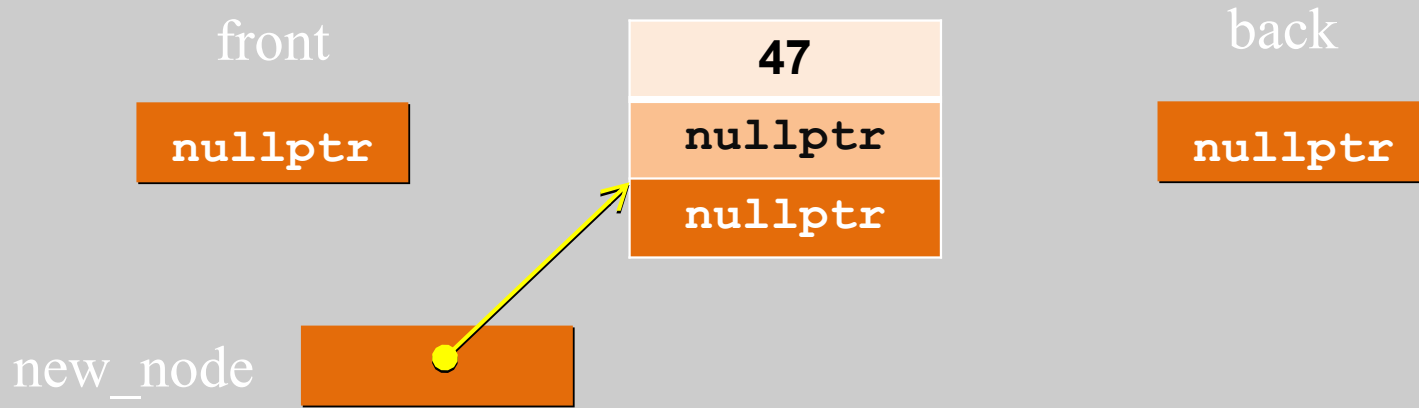
`nullptr`

back

`nullptr`

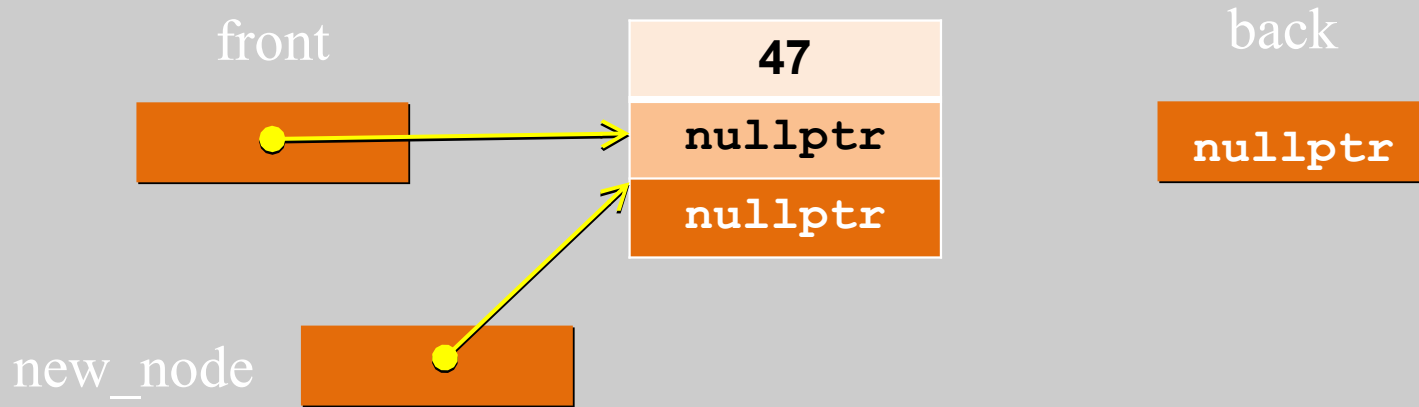
An empty doubly linked list

Adding To An Empty Doubly-Linked List (page 2 of 4)



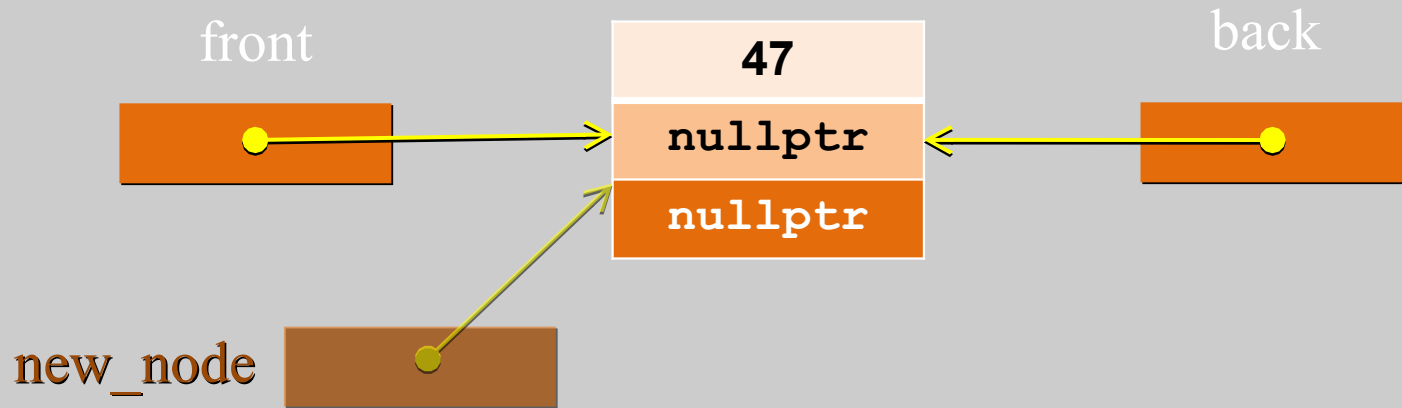
```
TwoWayNode *new_node = new TwoWayNode(47, nullptr, front) ;
```

Adding To An Empty Doubly-Linked List (page 3 of 4)



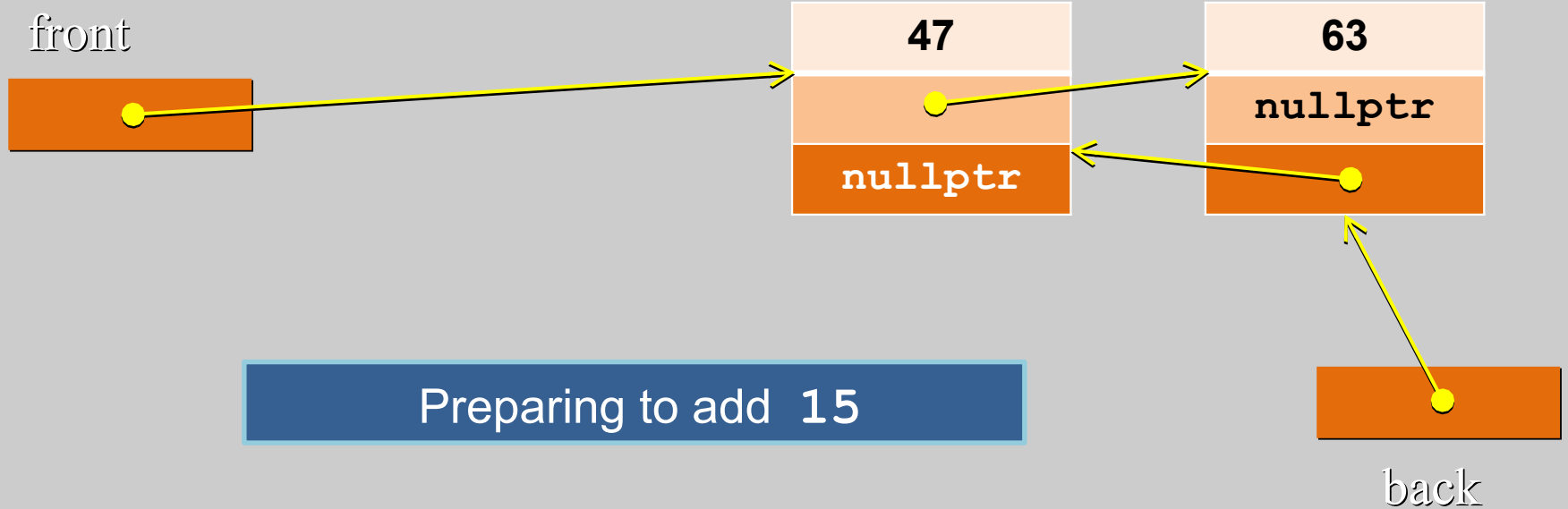
```
front = new_node ;
```

Adding To An Empty Doubly-Linked List (page 4 of 4)

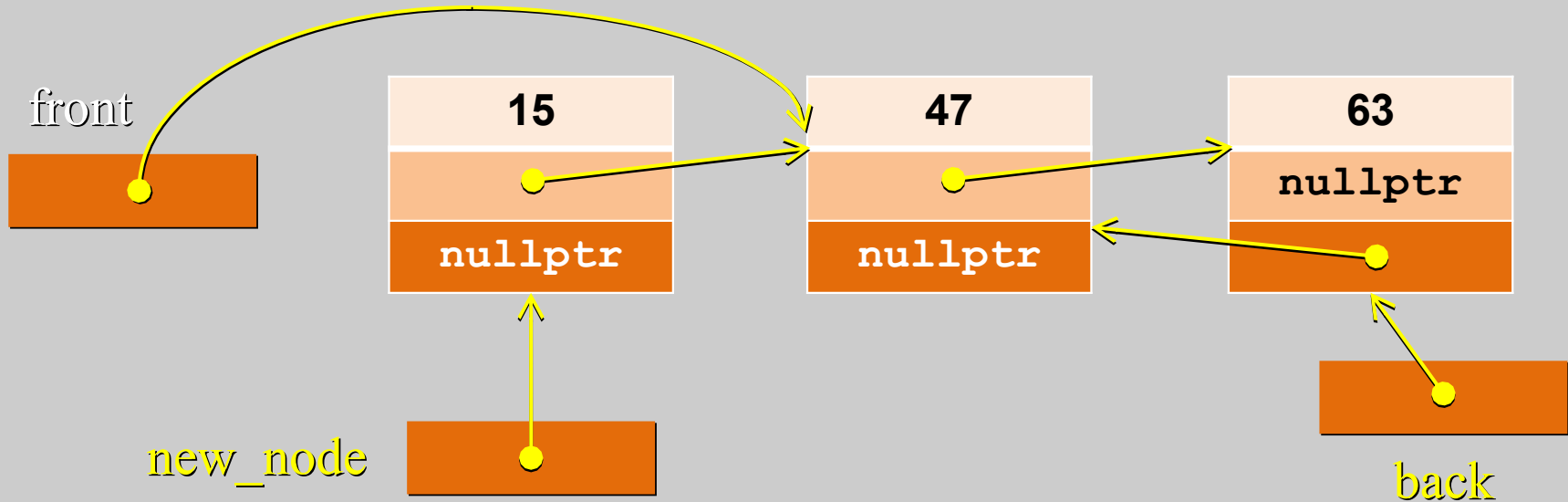


```
back = new_node ;
```

Adding to the Front of a Doubly Linked List (1 of 4)

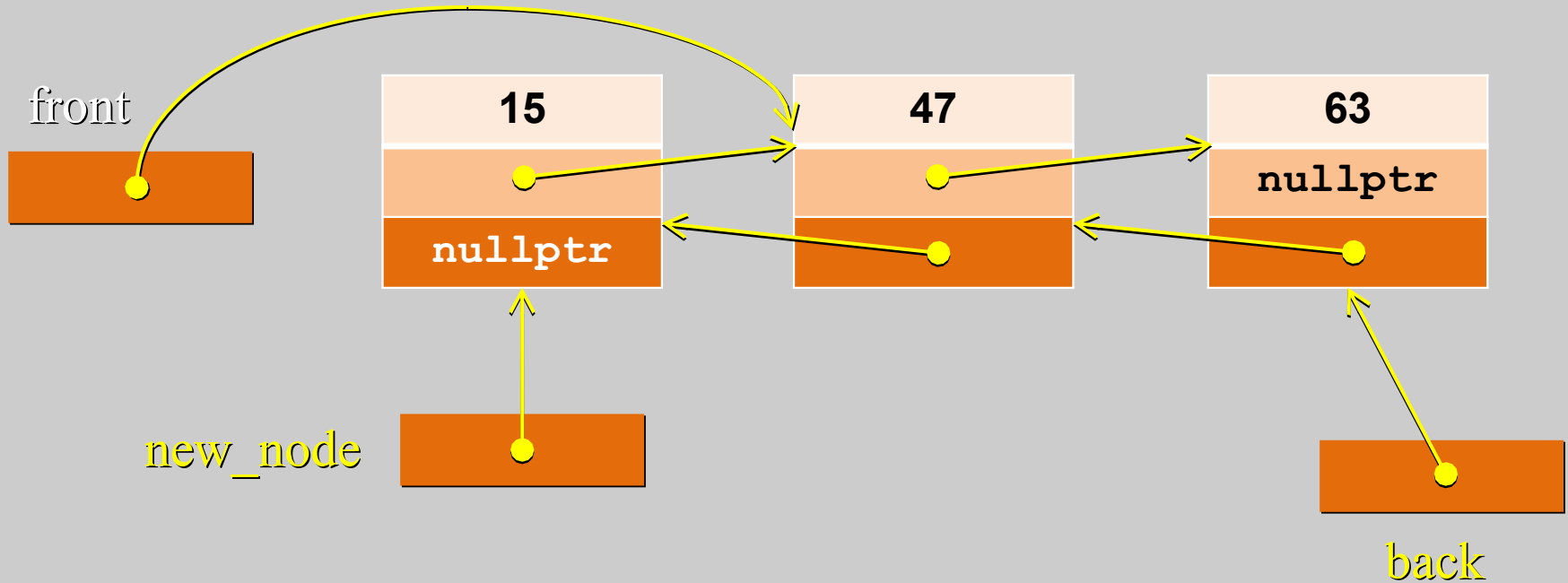


Adding to the Front of a Doubly Linked List (2 of 4)



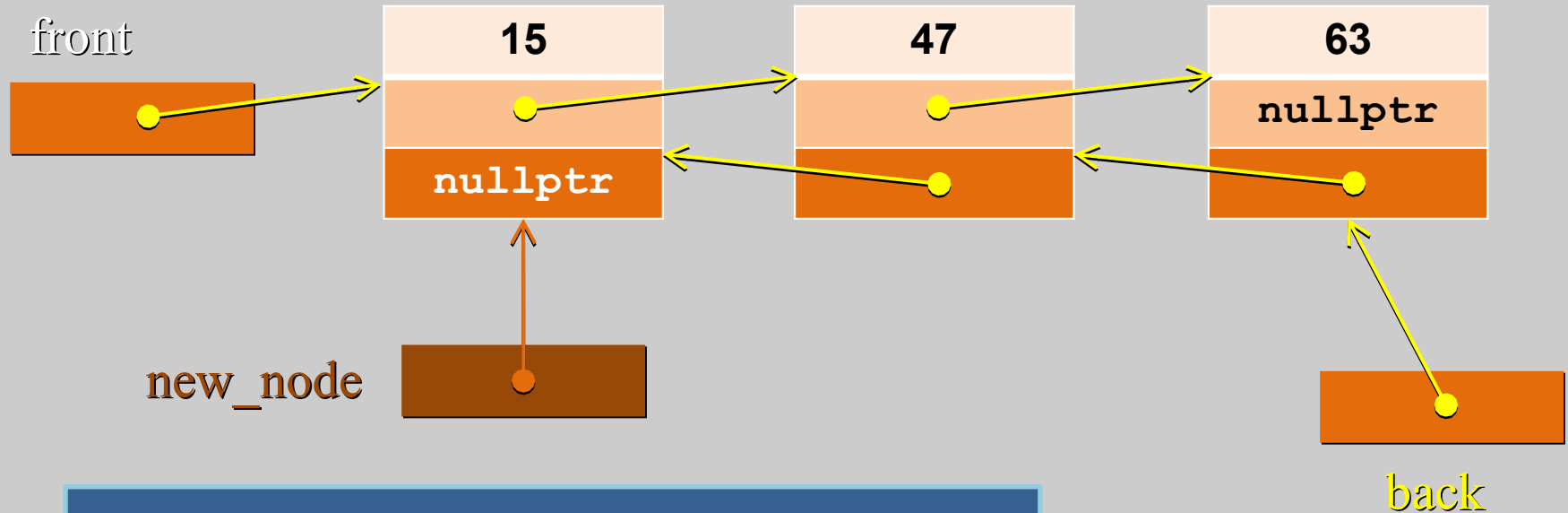
```
TwoWayNode *new_node =  
    new TwoWayNode (15, nullptr, front) ;
```


Adding to the Front of a Doubly Linked List (3 of 4)



```
front->previous = new_node ;
```

Adding to the Front of a Doubly Linked List (4 of 4)



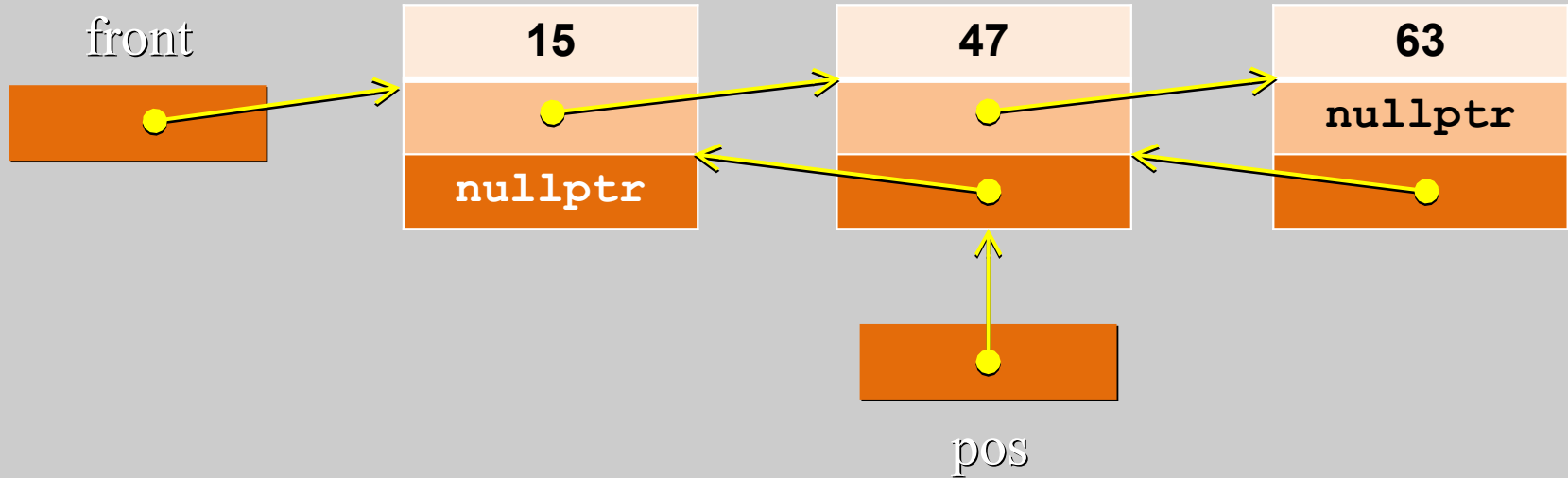
```
front = new_node ;  
Variable newNode is recycled.
```

Deleting a **TwoWayNode** from a Doubly-Linked List

Deleting a node from a doubly-linked list looks quite different from the code we used to delete a node from a singly linked list, since we need only one pointer:

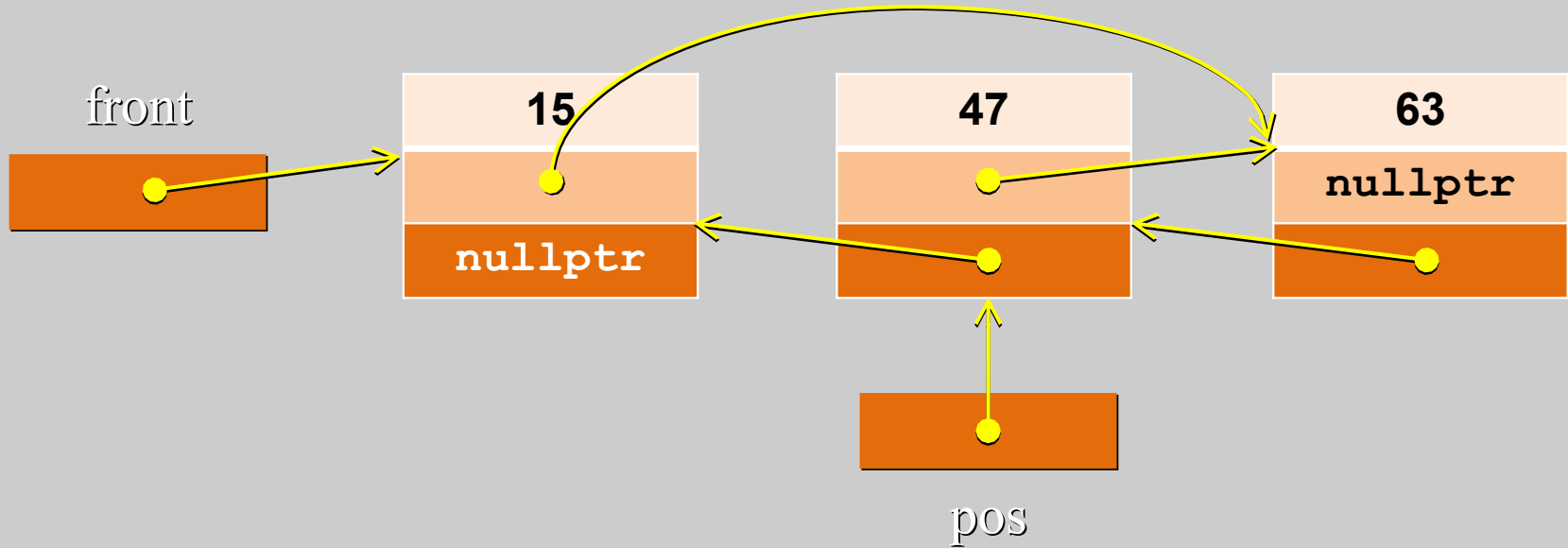
```
pos->previous->next = pos->next ;  
pos->next->previous = pos->previous ;  
pos = pos->next ;
```

Deleting from a Doubly Linked List (1 of 4)



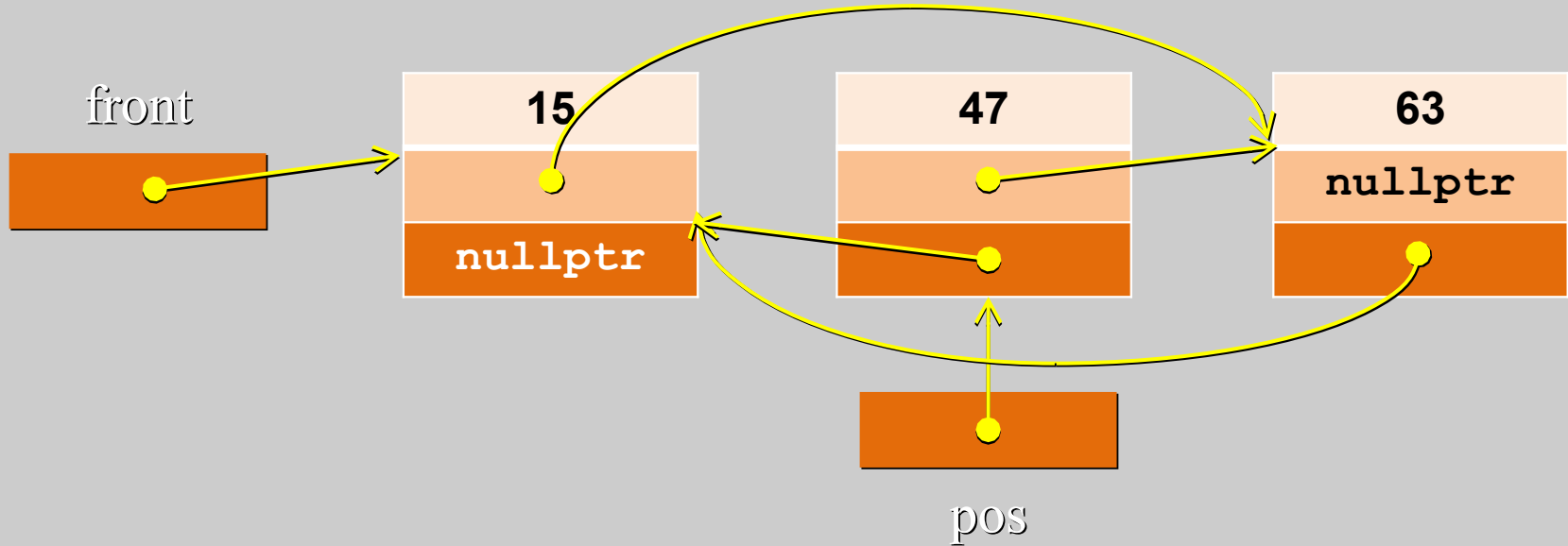
Prepare to remove **47** from the list

Deleting from a Doubly Linked List (2 of 4)



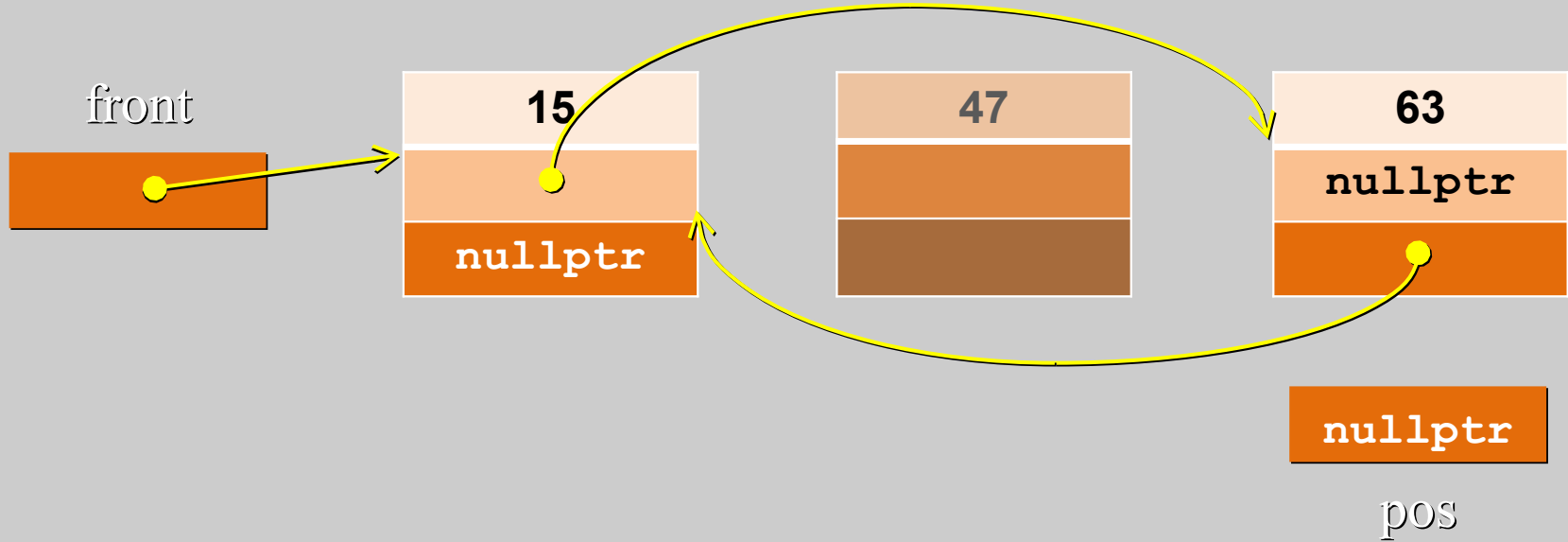
```
pos->previous->next = pos->next ;
```

Deleting from a Doubly Linked List (3 of 4)



```
pos->next->previous = pos->previous ;
```

Deleting from a Doubly Linked List (4 of 4)



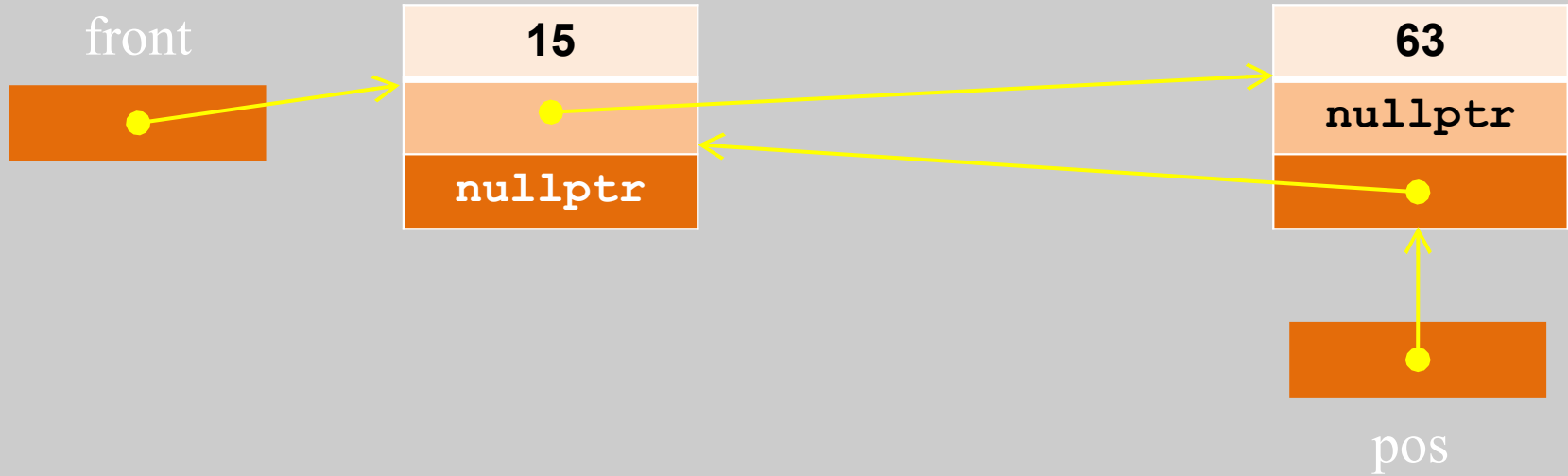
```
delete pos ;  
pos = nullptr ;
```

Inserting a **Node** into a Doubly-Linked List

Here's the code for inserting a node into a doubly-linked:

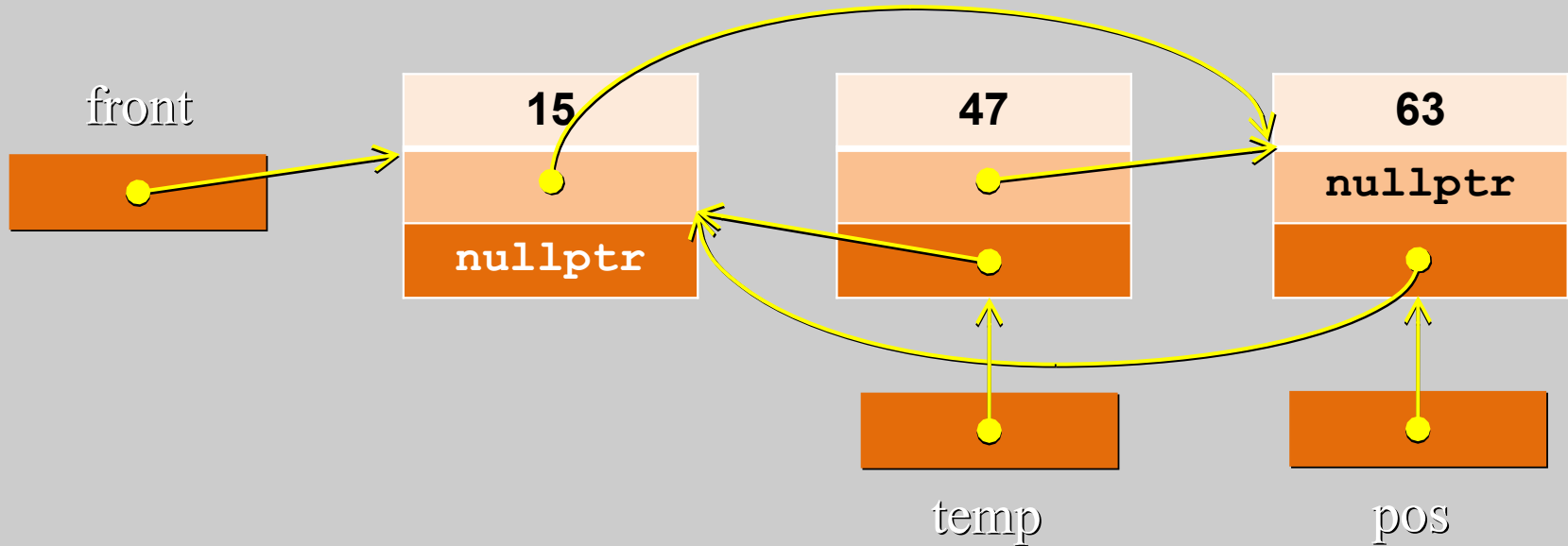
```
TwoWayNode *temp = new TwoWayNode(newData,  
                                   pos->previous, pos) ;  
pos->previous->next = temp ;  
pos->previous = temp ;
```


Inserting into a Doubly Linked List (1 of 4)



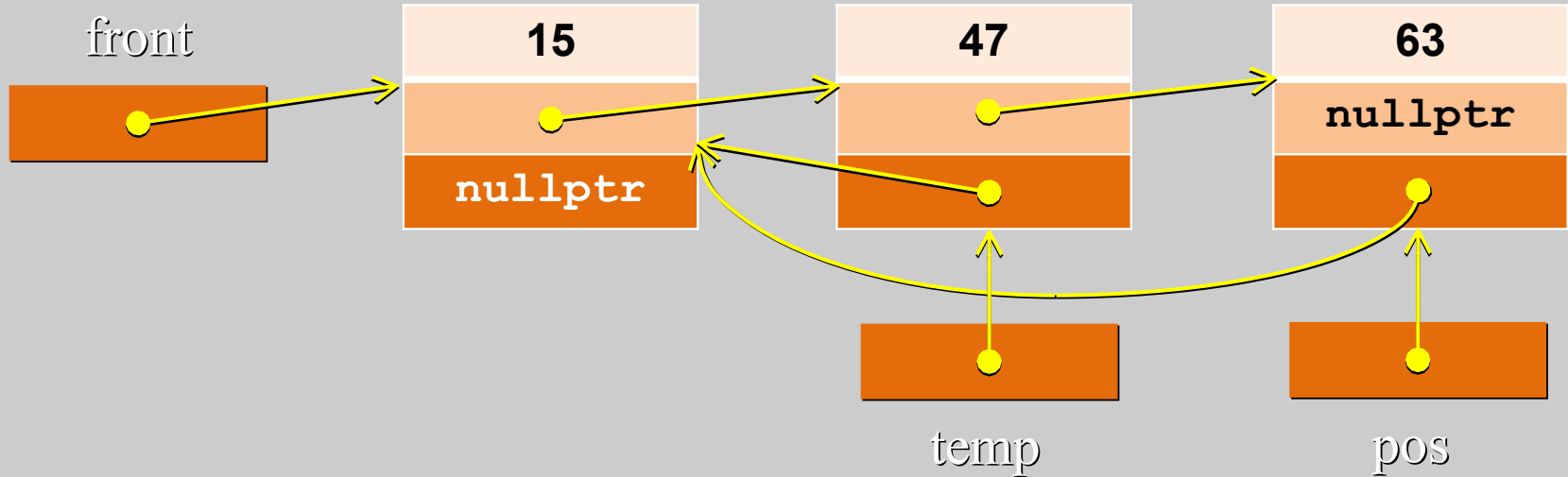
Prepare to insert **47** into the list

Inserting into a Doubly Linked List (2 of 4)



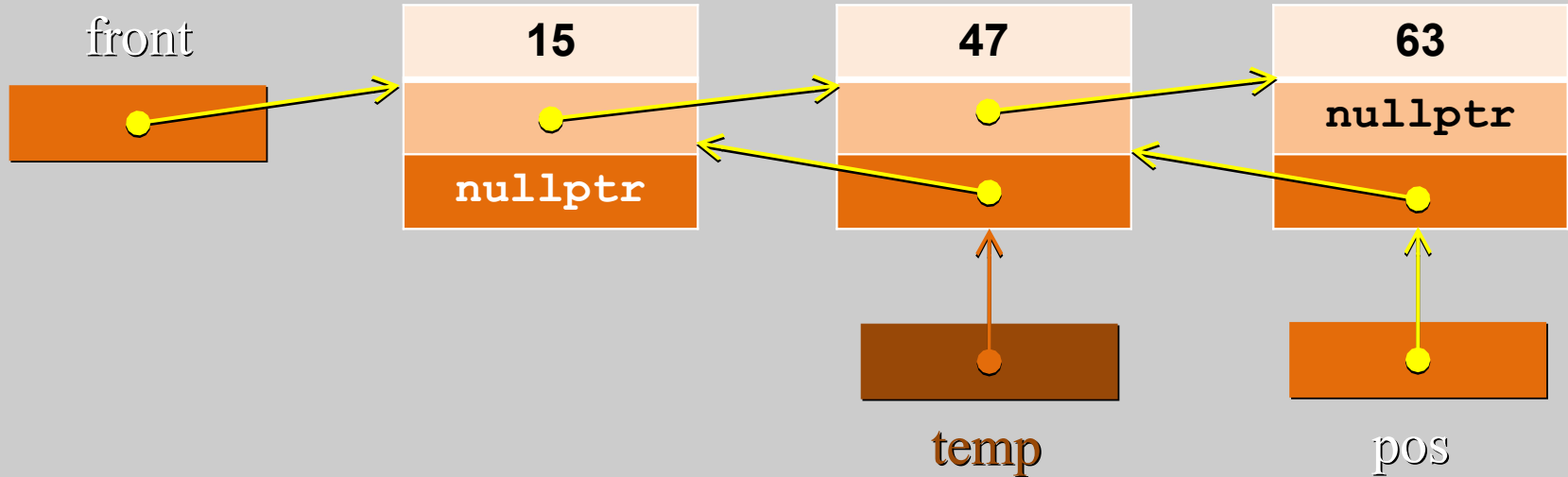
```
TwoWayNode *temp =  
    new TwoWayNode(47, pos->previous, pos) ;
```

Inserting into a Doubly Linked List (3 of 4)



```
pos->previous->next = temp ;
```

Inserting into a Doubly Linked List (4 of 4)



```
pos->previous = temp ;  
Pointer temp is recycled.
```


Some Observations

You've seen how a linked list works:

- Linked lists are always “full” (there are no empty slots).
- Adding/inserting to and removing from a linked list moves items after the insertion/deletion point to a new “position” in the list without actually moving elements in the list.
 - Once an insertion/deletion point is found the actual change simply involves moving pointers.

Observations (continued)

- However, when inserting or removing elements from the middle of a large list, that list must be traversed first to find the desired position in the list.
 - *True* random access data structures such as vectors do not have this problem.

Imagine performing a binary search using a list!!!

Vectors vs Linked Lists

An interesting video from the designer and original implementor of C++, Bjarne Stroustrup, comparing the two data structures.

<https://www.youtube.com/watch?v=YQs6IC-vgmo>

The STL **list** Container

The STL (Standard Template Library) contains a **list** class.

- It is generic (can accommodate a single data element of any type).
- It uses a doubly-linked list.
- Many of the functions in the class are identical in operation to other data structures in the STL.

The STL **list** Container

Member functions for the **list** container:

list.front() – returns the first element in the list

list.back() – returns the last element in the list

- If the list contains objects of a class or structure, these functions return a pointer to that object.
- If the list is empty, the behavior is undefined.

The STL **list** Container

list.begin() – returns an iterator to the beginning (the first element) in the list

list.end() – returns an iterator to “passed-the-end” of the list

- behavior is undefined if the list is empty.
- **end()** does not point to an actual element, and therefore should never be dereferenced.

The STL **list** Container

list.rbegin() – returns a *reverse* iterator which points to the last element in the list.

list.rend() – returns a *reverse* iterator to the *theoretical* item before the first element in the list.

- behavior is undefined if the list is empty.
- **rend()** does not point to an actual element, and therefore should never be dereferenced.

The STL `list` Container

`list.push_back(x)` – pushes the element `x` on the end of the list, after its current last element. Increases size by one.

`list.pop_back()` – deletes the last element in the list, effectively reducing the size of the list by one.

- Also `push_front(x)` and `pop_front()`

The STL **list** Container

list.size() – returns the number of elements in the list (0 means the list is empty)

list.reverse() – reverses the order in which items appear in the list

list.unique() – eliminates any item in the list which duplicates another item appearing before it.

The STL **list** Container

list.find(*iter first*, *iter last*, **x**) – returns an iterator to the first occurrence of the element **x** occurring between **first** and **last** (exclusive)

- If **x** is not found, **last** is returned.
- The function uses the **operator==** to determine equality between elements.

The STL **list** Container

list.insert(*iterator pos*, **x**) – inserts the element **x** at the position referenced by the iterator.

list.merge(*list &a*) – Merges list **a** into the calling list by moving all of its elements into their ordered positions into the calling list

- Both lists should already be in sorted order.
- When finished, list **a** will be empty.

The STL **list** Container

list.empty() – removes all elements in a list, setting its size to zero.

list.erase(*iterator pos*) – deletes the element currently pointed by to **pos** in the list.

- The iterator should be bi-directional.
- Other iterators and pointers which point to the deleted element become invalid.

The STL **list** Container

list.erase(*iterator start*, *iterator stop*) – deletes all element occurring in the list between **start** (inclusive) and **stop** (exclusive).

- Deletes the element referenced by **start**, but does not delete the element referenced by **stop**.
- Both iterators should be bi-directional.

Smart Pointers and Linked Lists

Smart pointers are generally helpful in preventing common problems such as memory leaks, inadvertent deletion, and multiple deletion.

- They may not be useful in implementing linked lists:
 - Unique pointers cannot be used due to ownership.
 - Shared pointers add the overhead of shared groups.
 - Using shared pointers in a circularly-linked list may be tricky to code.

Homework

Homework, Module 14

- Complete current homework assignment
- Complete lab assignment
- Complete any old homework or labs
- Start or continue working on final or research projects