Courtesy of freeimages.com
https://www.freeimages.com/photo/digital-dreams-1155914

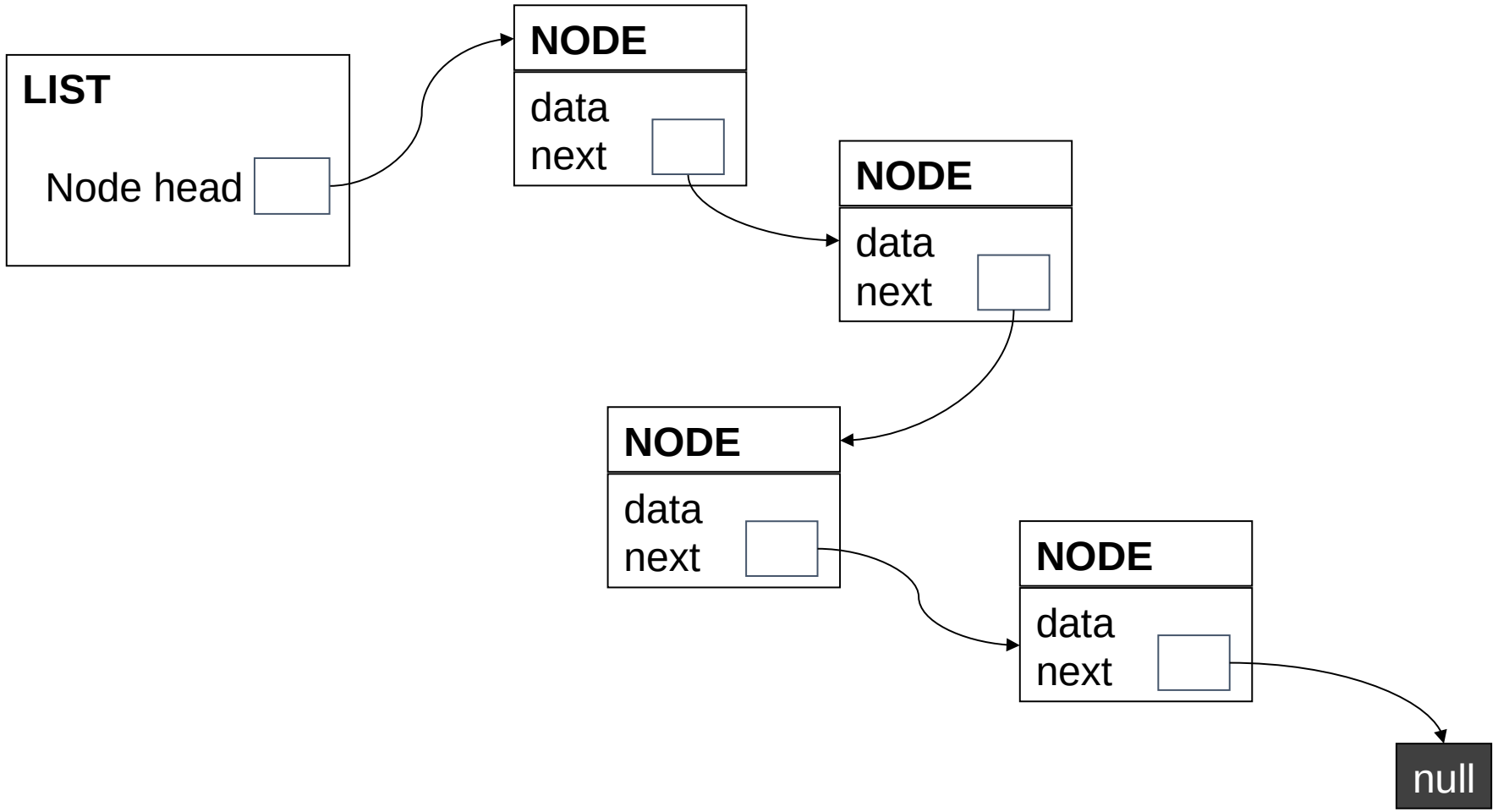
AC21008
Multi-paradigm programming and data structures

Data Structures
Doubly-linked list with iterator

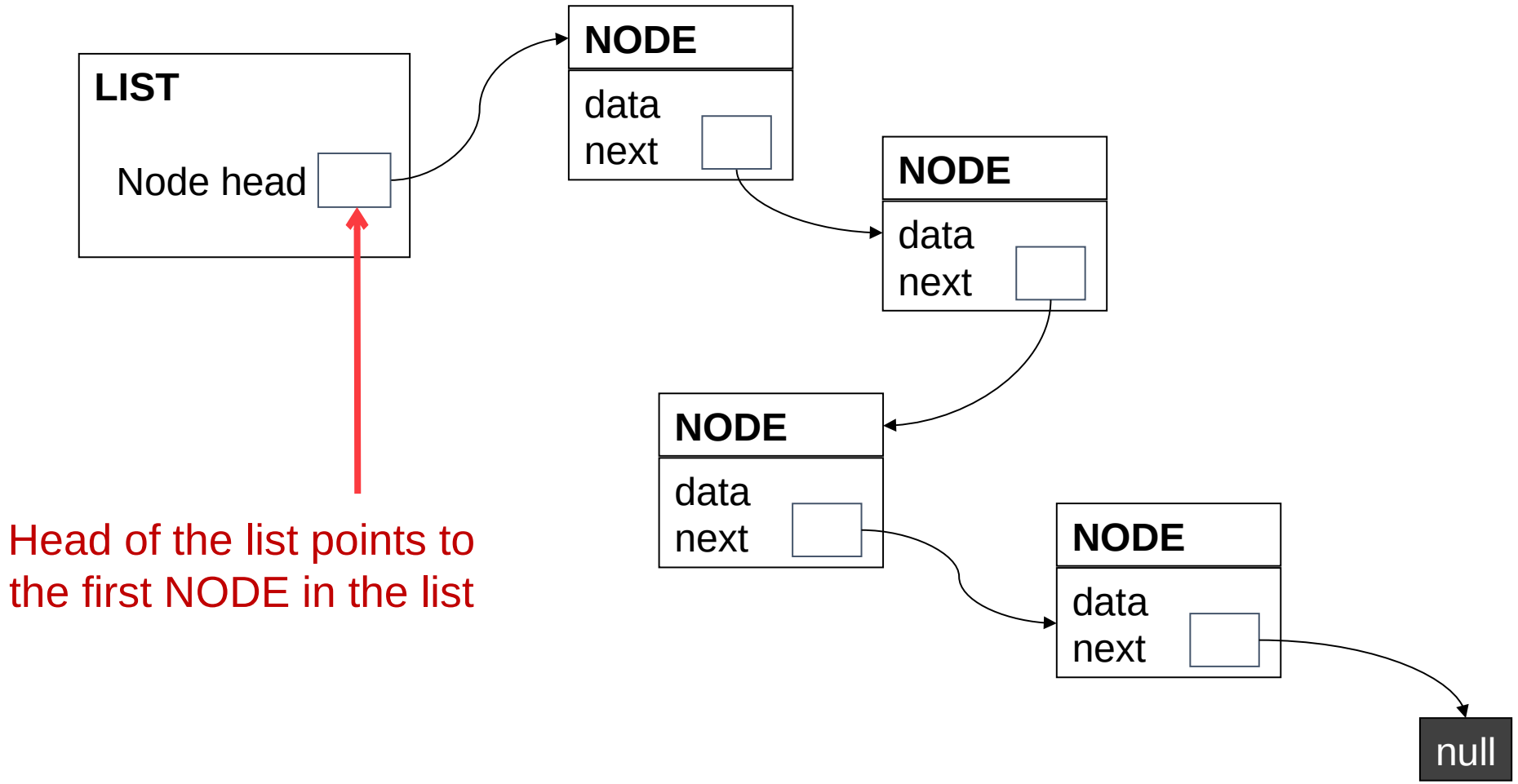
Overview

- Implementing a doubly-linked linked list with iterator!
- You will be implementing this for your next assignment!
- We will walk through the design together

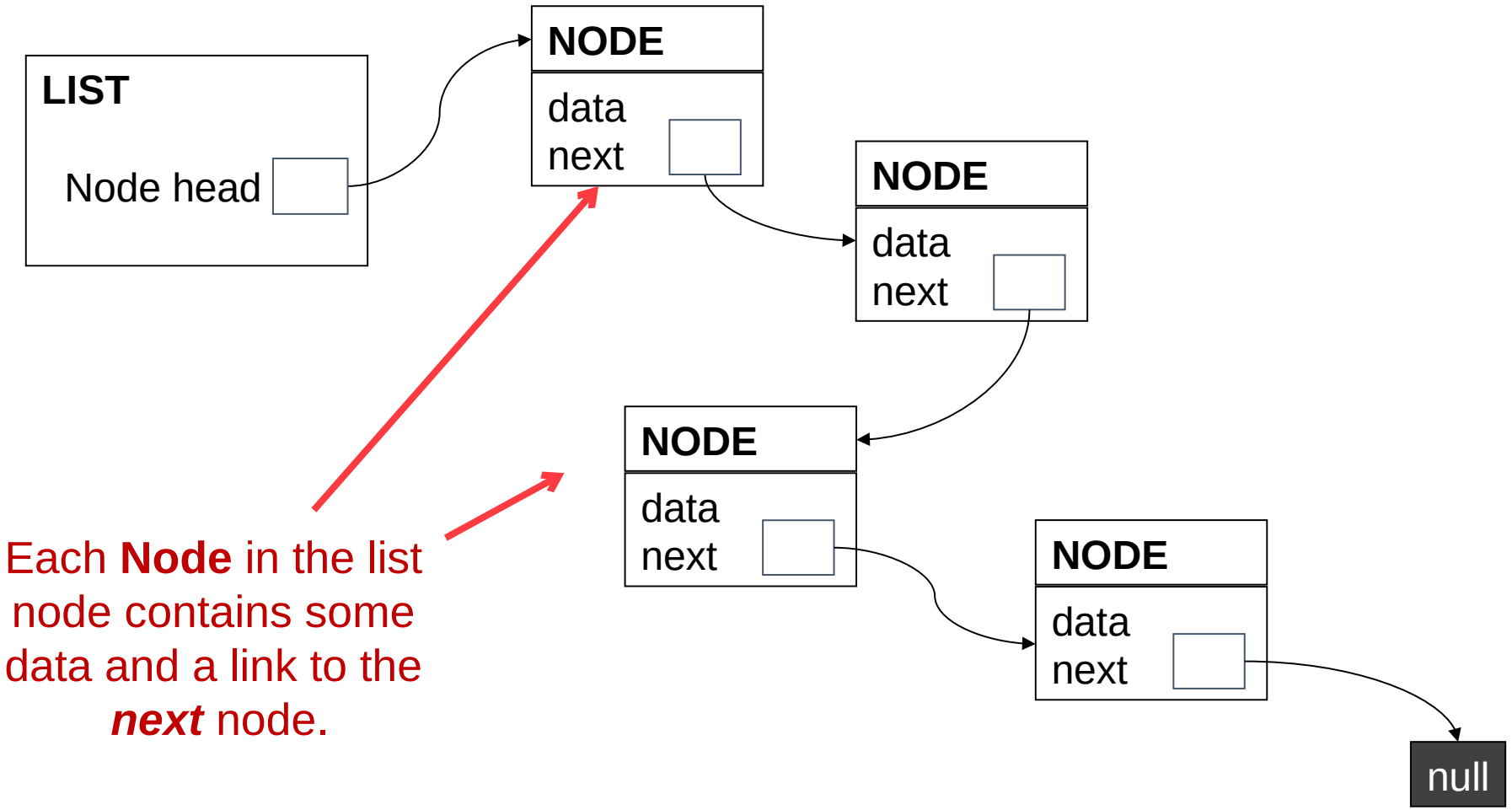
Linked List


LIST
Node head
NODE
data
next
NODE
data
next
NODE
data
next
NODE
data
next
null

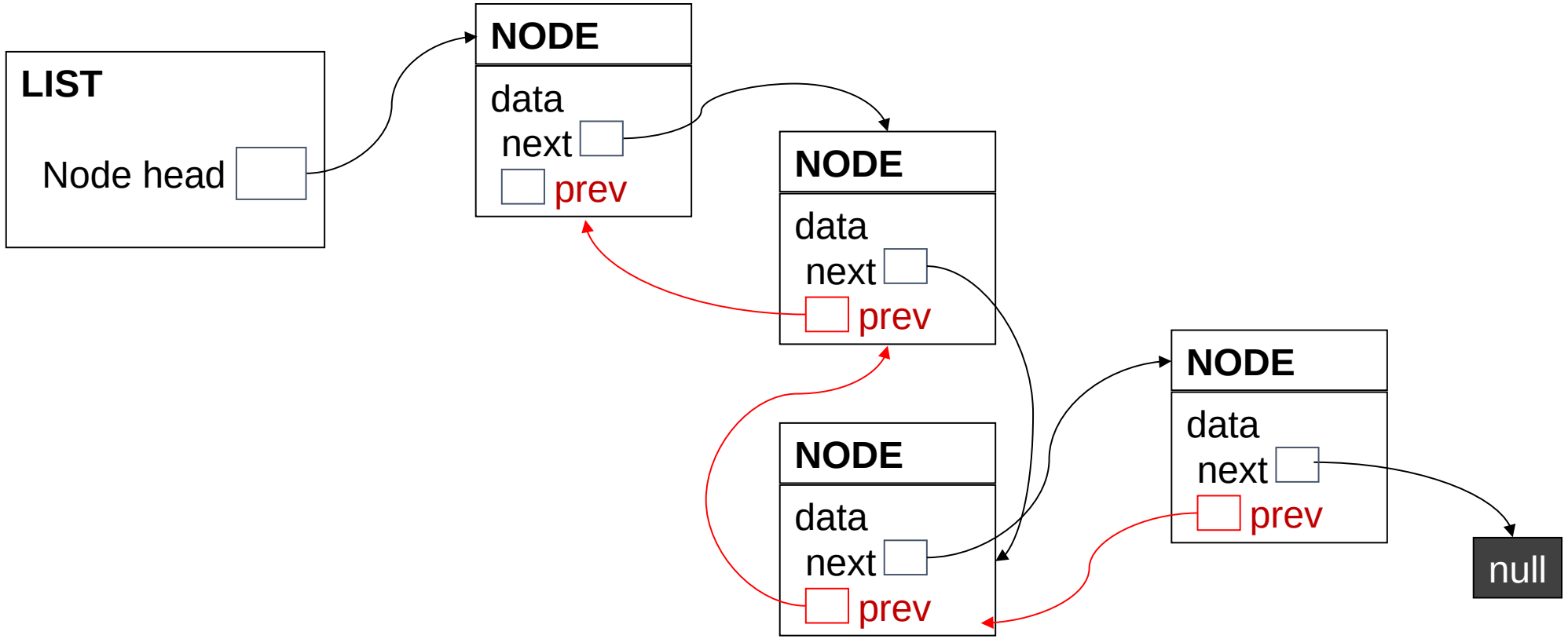
Linked List


LIST
Node head
NODE
data
next
NODE
data
next
NODE
data
next
NODE
data
next
null
Head of the list points to
the first NODE in the list

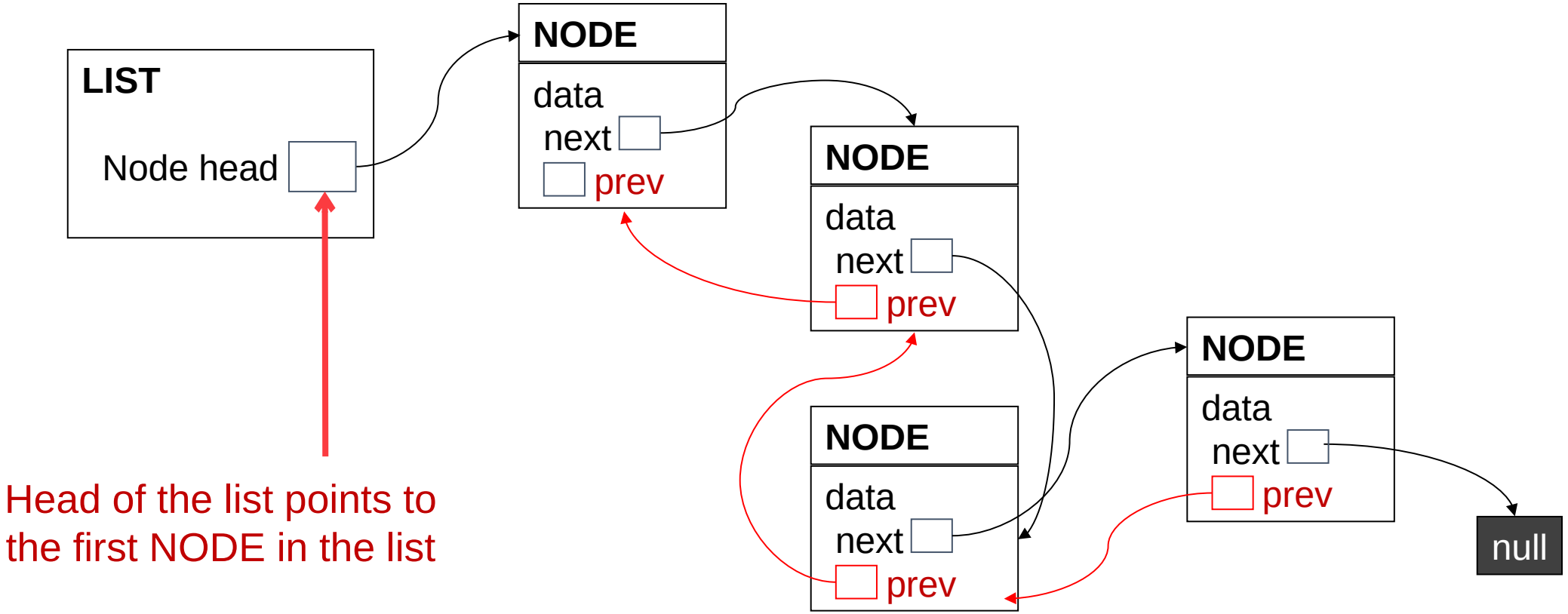
Linked List


LIST
Node head
NODE
data
next
NODE
data
next
NODE
data
next
NODE
data
next
null
Each **Node** in the list node contains some data and a link to the ***next*** node.

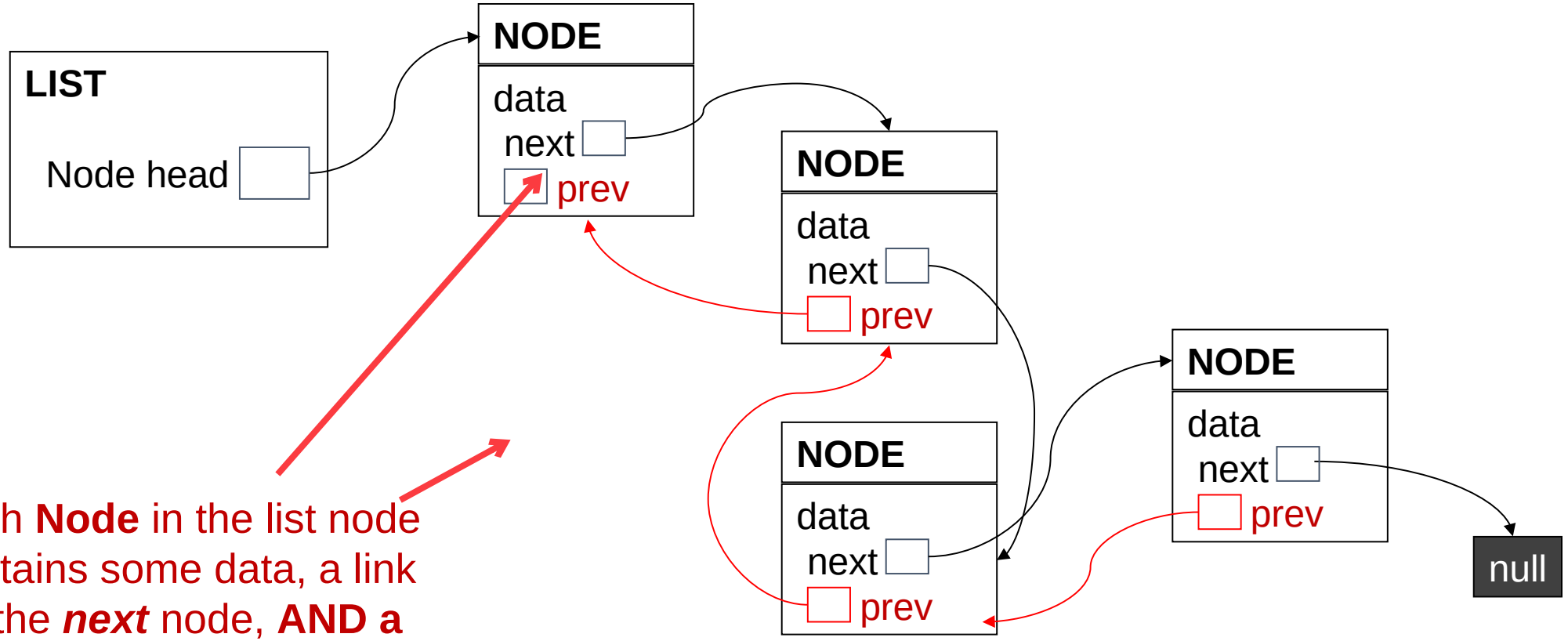
Doubly Linked List



Doubly Linked List



Doubly Linked List

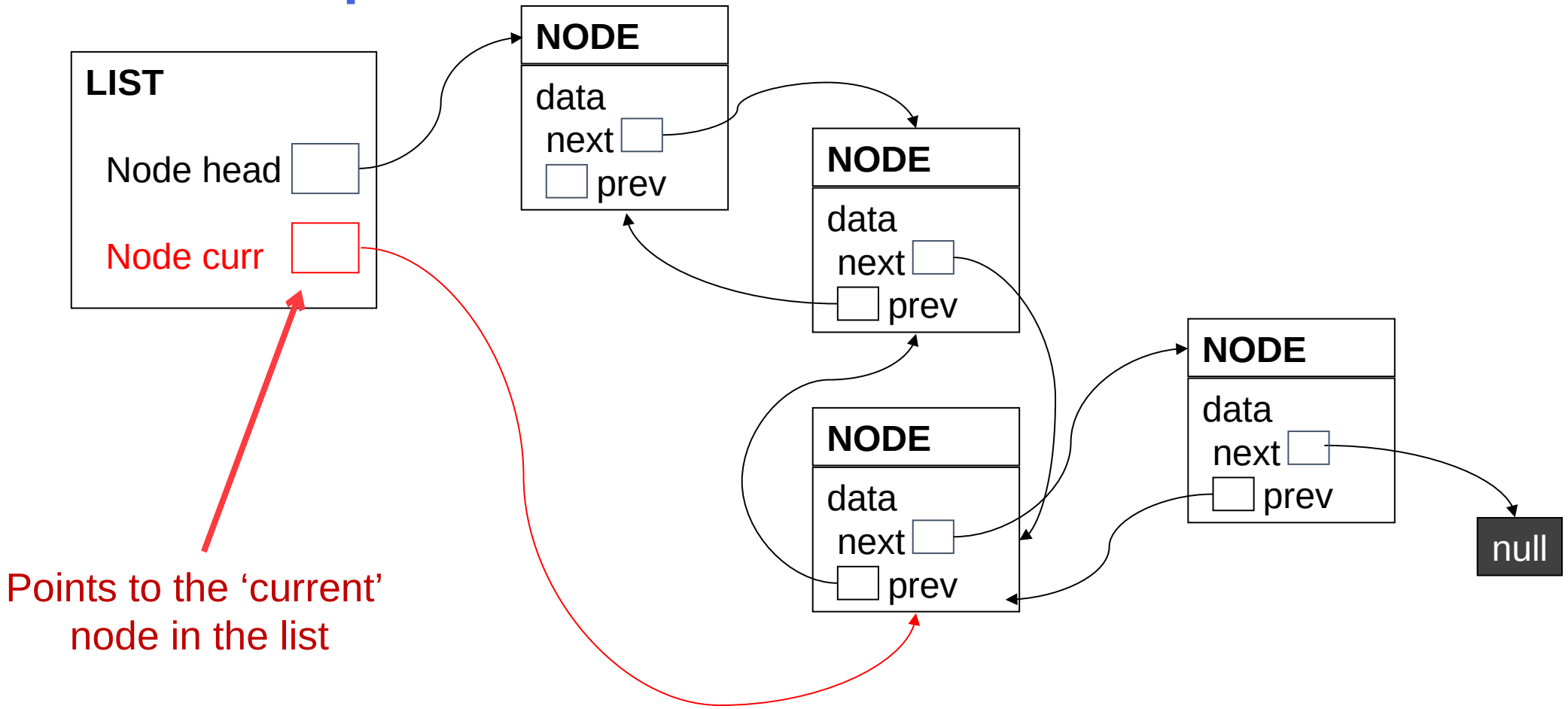


Each **Node** in the list node contains some data, a link to the ***next*** node, **AND a link to the previous node too.** Allows the list to be traversed in both directions

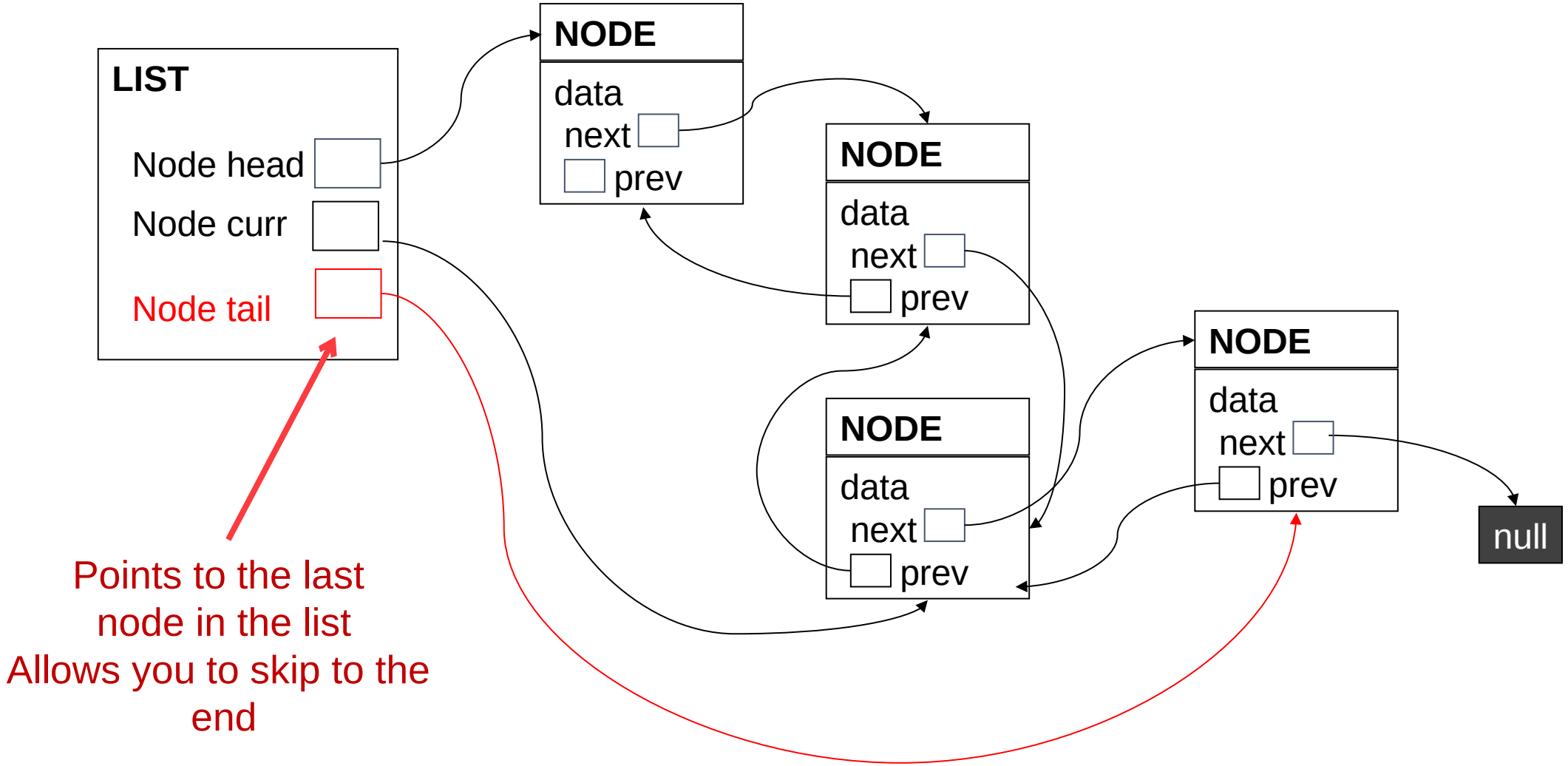
Applications of a doubly-linked list

- Think of a playlist in a music player
  - Shifting through tracks, backwards and forwards
- Think of the Back and Next keys on a web browser
- Also useful to know, what is the **current** position within the list?
  - Which is the current track in the playlist?
  - Which page is current in the web browser?

Doubly Linked List with a ‘current’ pointer



Also add a ‘tail’



Possible code structures

- ListNode and List

```
// defines a Node in a
// doubly-linked list
typedef struct listnode {
   int data;
   struct listnode *next;
   struct listnode *prev;
}; ListNode;
```

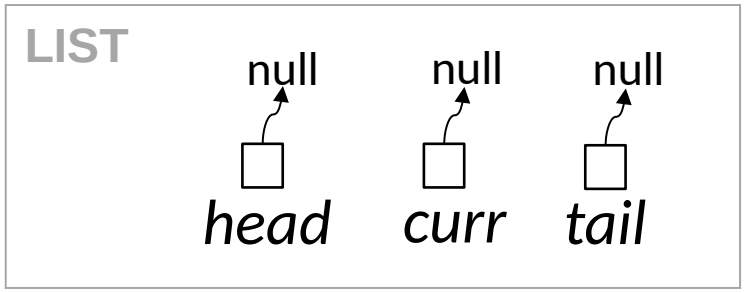
```
// defines a container for
// a doubly-linked list
// with ‘iterator’
typedef struct list {
   ListNode *head;
   ListNode *tail;
   ListNode *curr;
}; List;
```

Functions

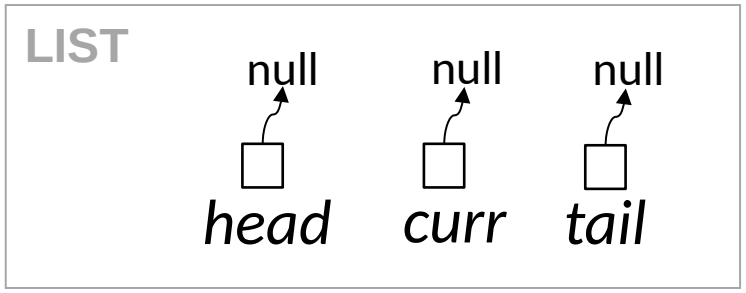
- Initialising an empty list
- Add new node after current position
- Add new node before current position
- Move current position forwards
- Move current position backwards
- Get data at current position
- Remove node at current position
- Free the list

Initialising the list

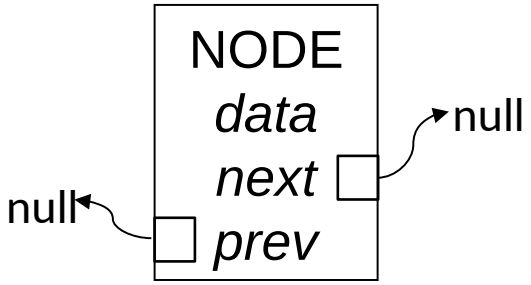
- An empty list
  - Nothing at the head, tail or current position



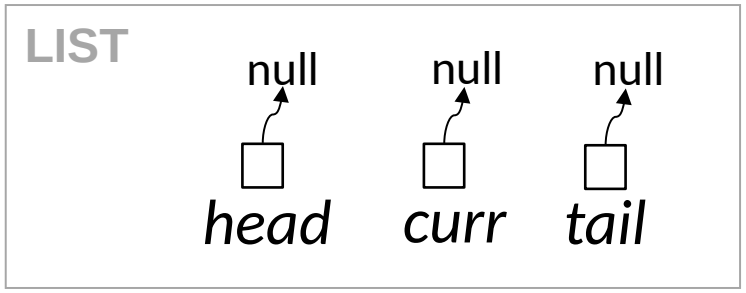
Adding a node to an empty list



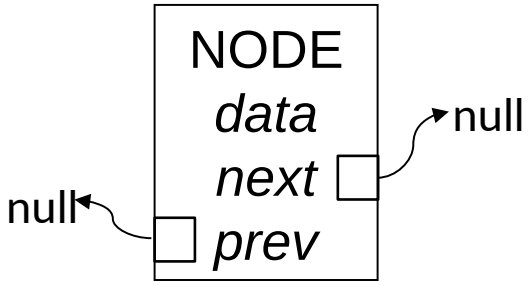
Adding a node to an empty list

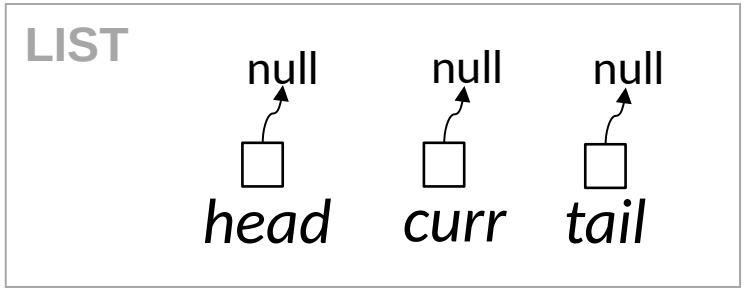


Create a new node, set it's data, etc.
Node's 'next' and 'prev' will be NULL, initially



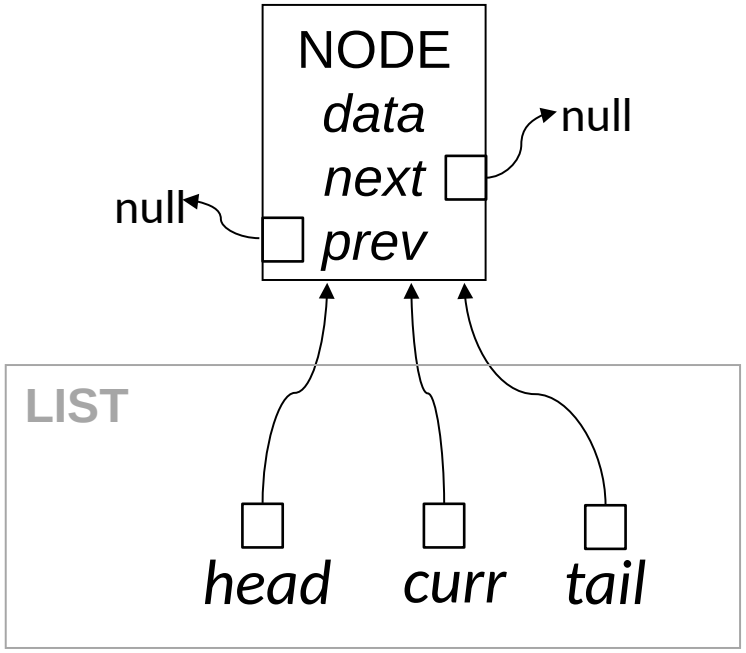
Adding a node to an empty list





new node becomes the new head, tail AND the current node
e.g. head = tail = curr = newNode

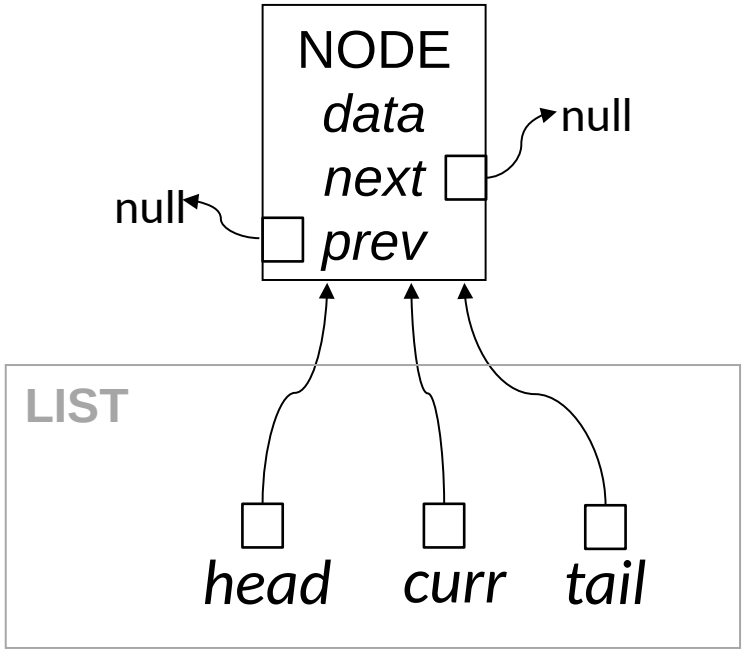
Adding a node to an empty list



Adding a node **after** the ‘current’ position

- Two cases to consider:
  - (i) ‘current’ is at the tail / end of the list – so, we will be inserting a new tail
  - (ii) ‘current’ is in the middle of the list (there’s at least one Node after it) – so, we will be inserting a new node in between two existing nodes

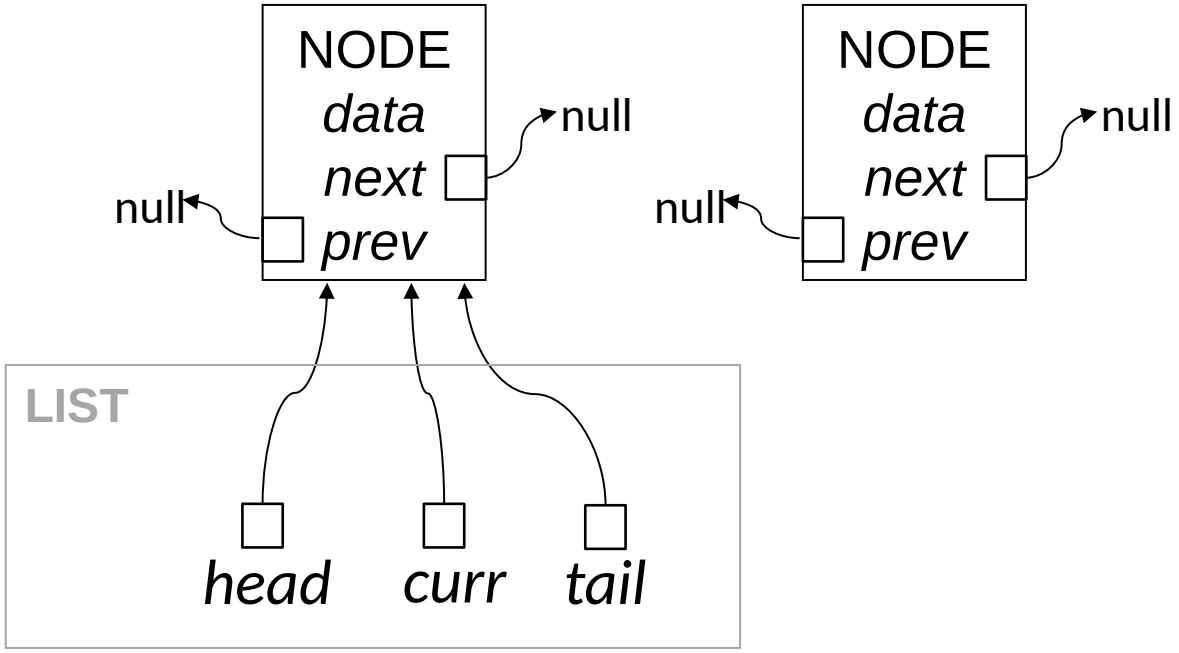
Add a node **after** the current position



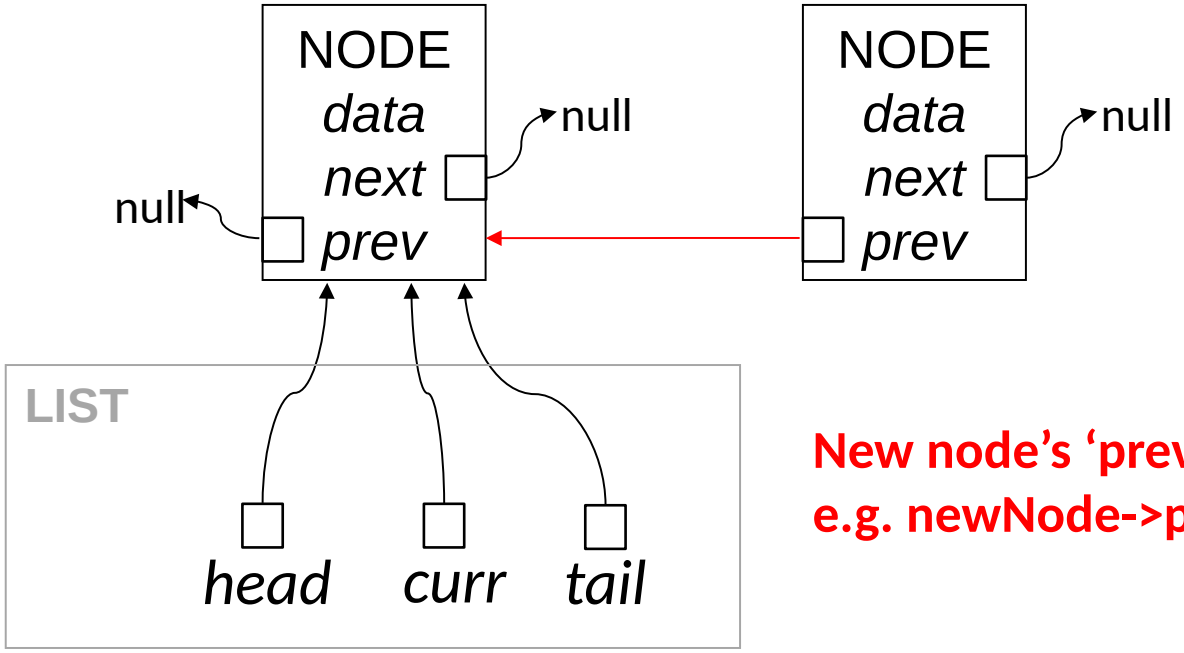
Case (i) – ‘current’ is at the ‘tail’ of the list

e.g. list->curr == list->tail

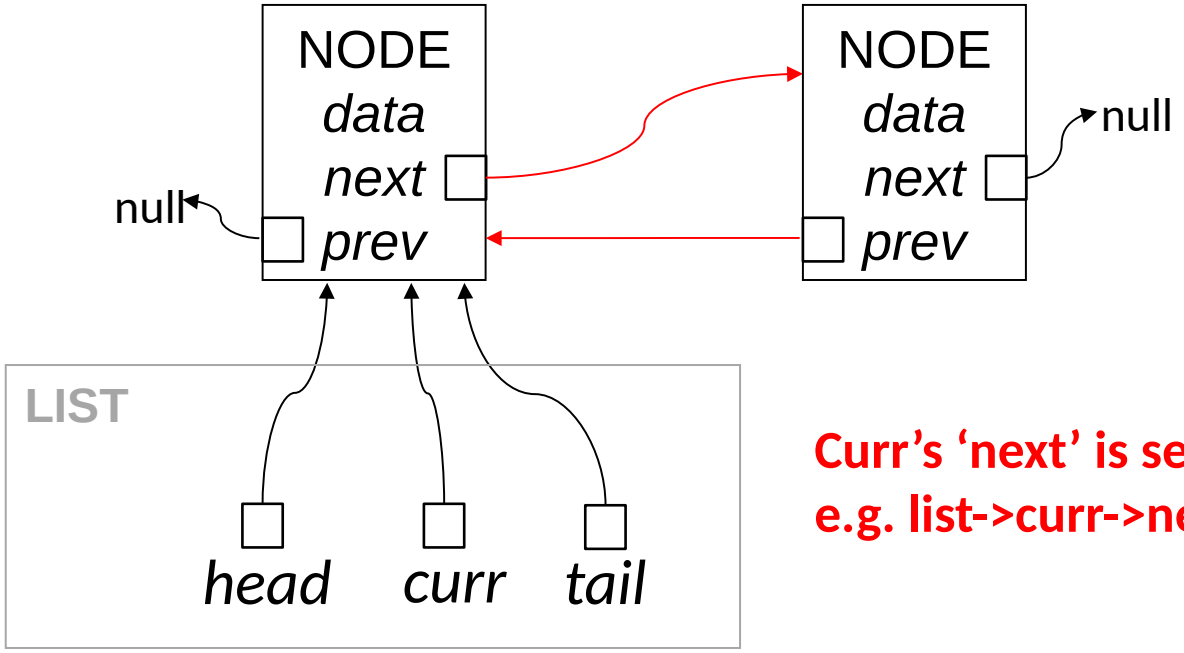
and list->curr->next == NULL



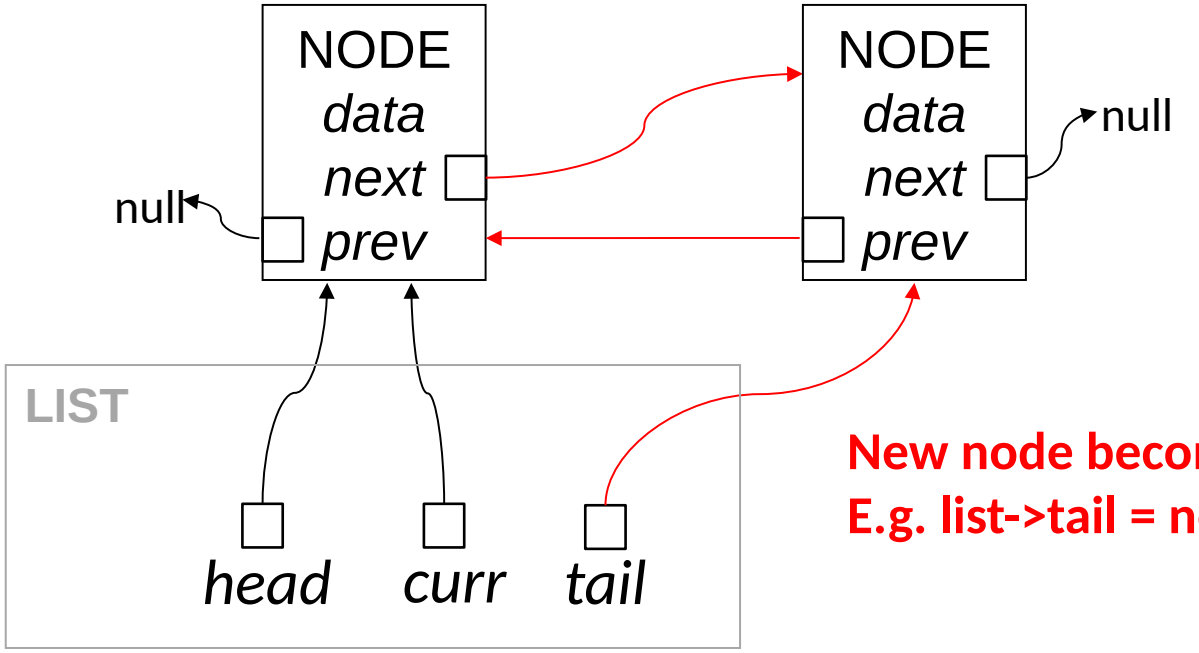
Create a new node, set it's data, etc.



New node's 'prev' is set to 'curr'
e.g. newNode->prev = list->curr

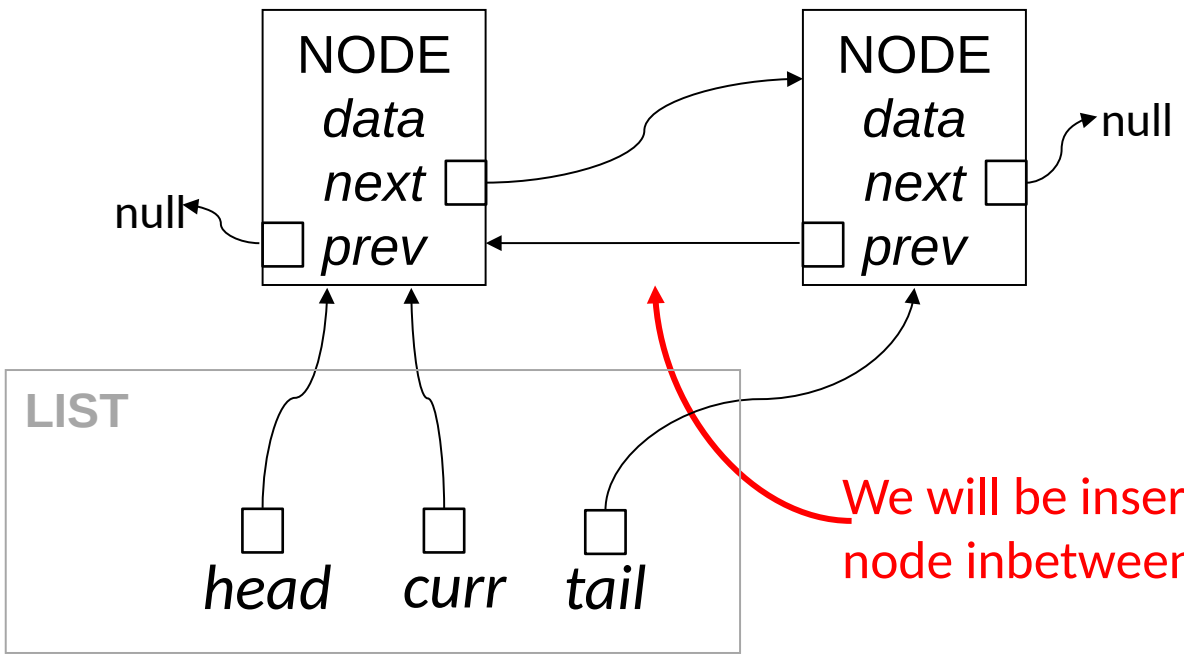


Curr's 'next' is set to new node
e.g. list->curr->next = newNode



New node becomes new tail of list
E.g. list->tail = newNode

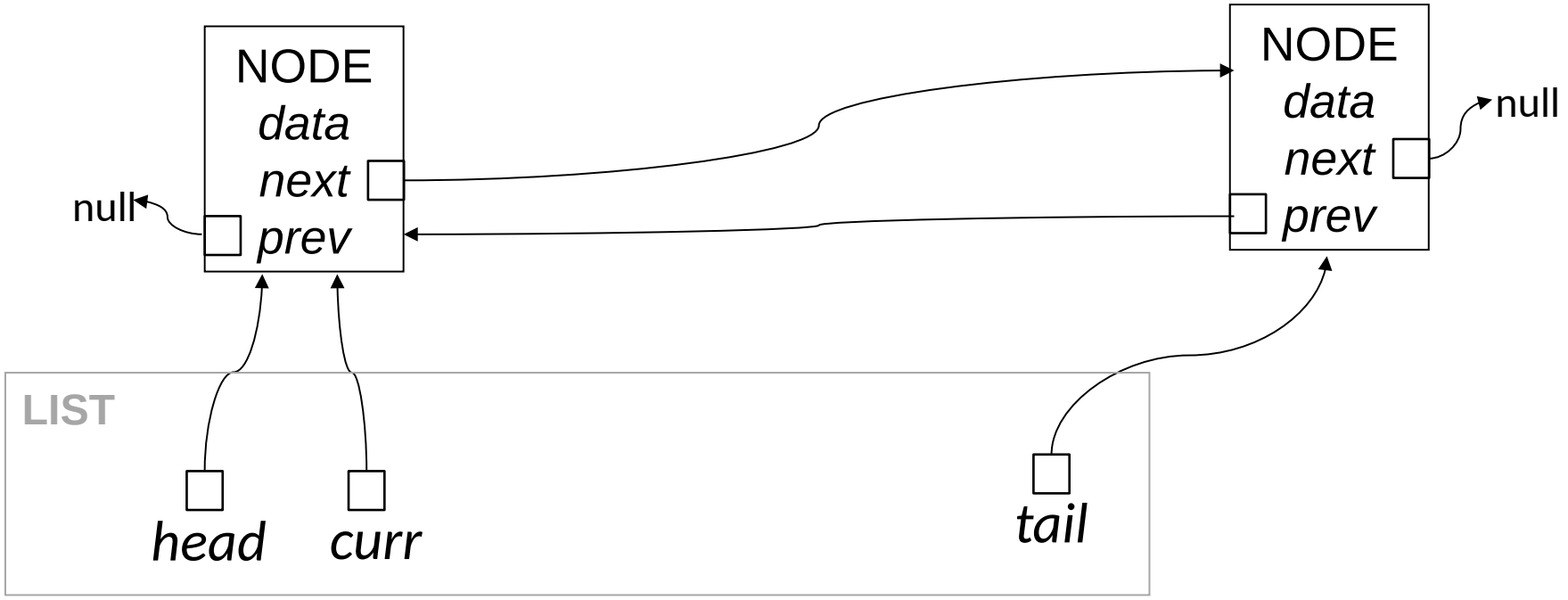
Add a node **after** the current position

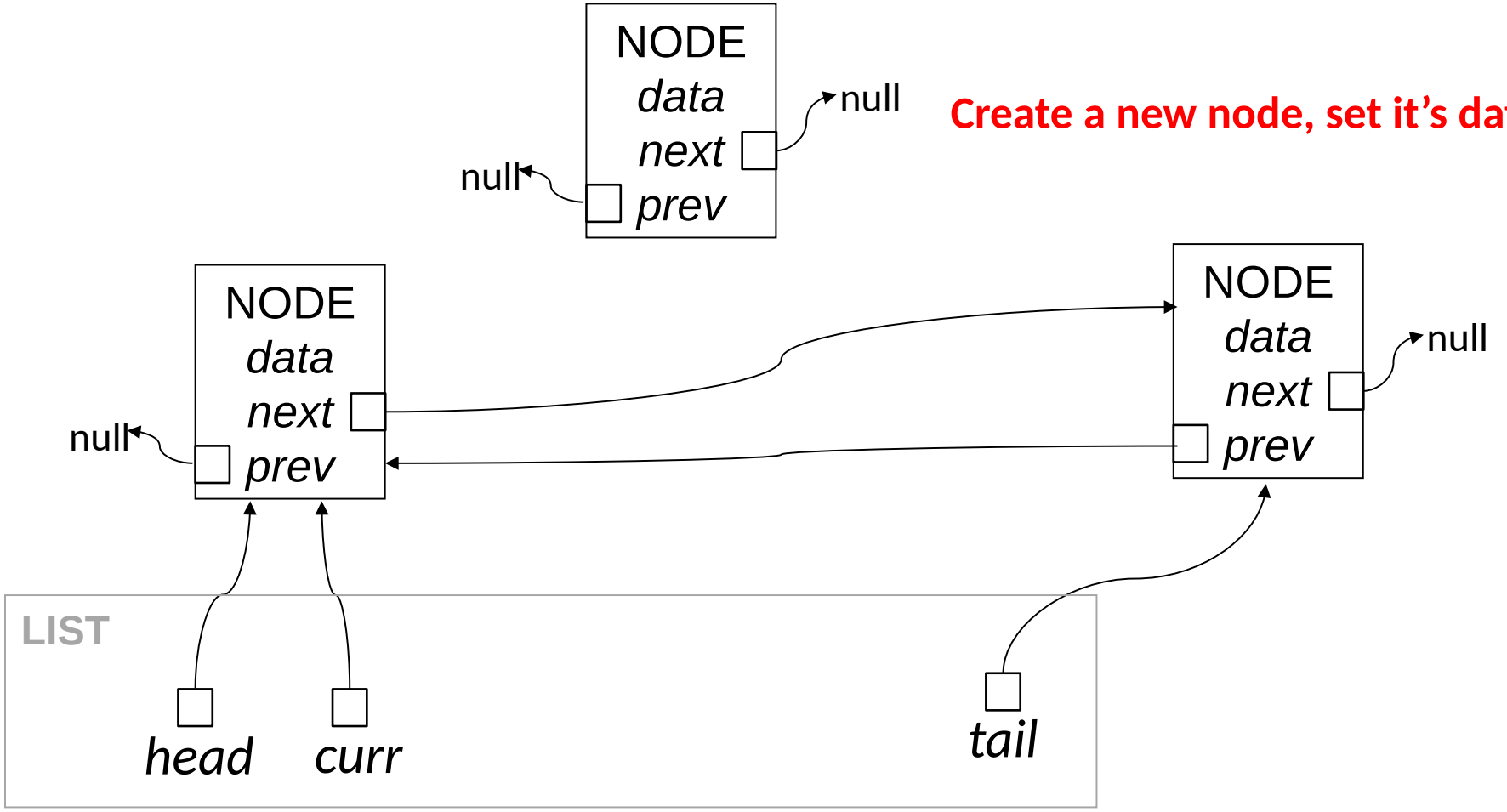


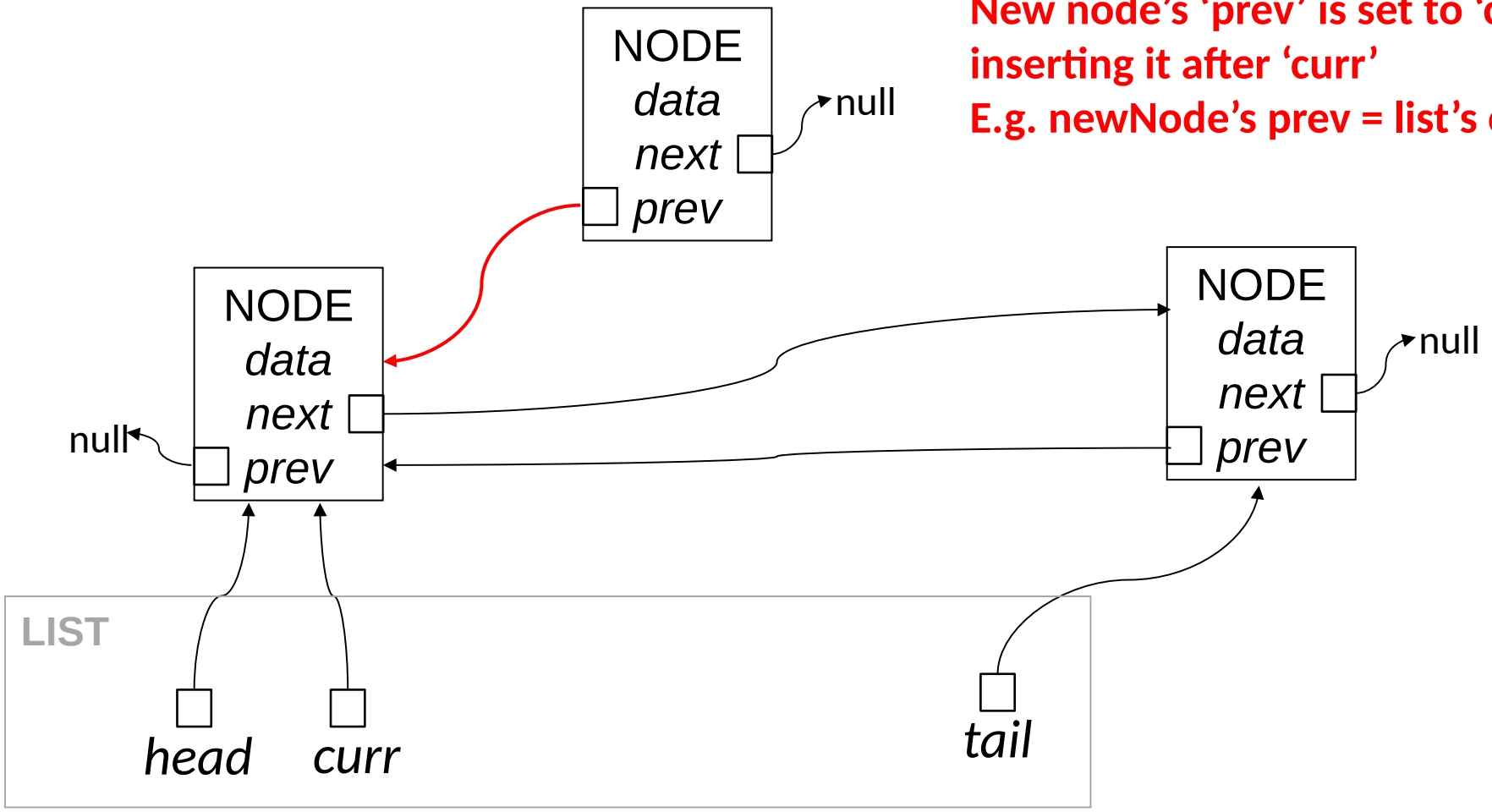
Case (ii) – 'current' is NOT the 'tail' of the list

list->curr->next != NULL

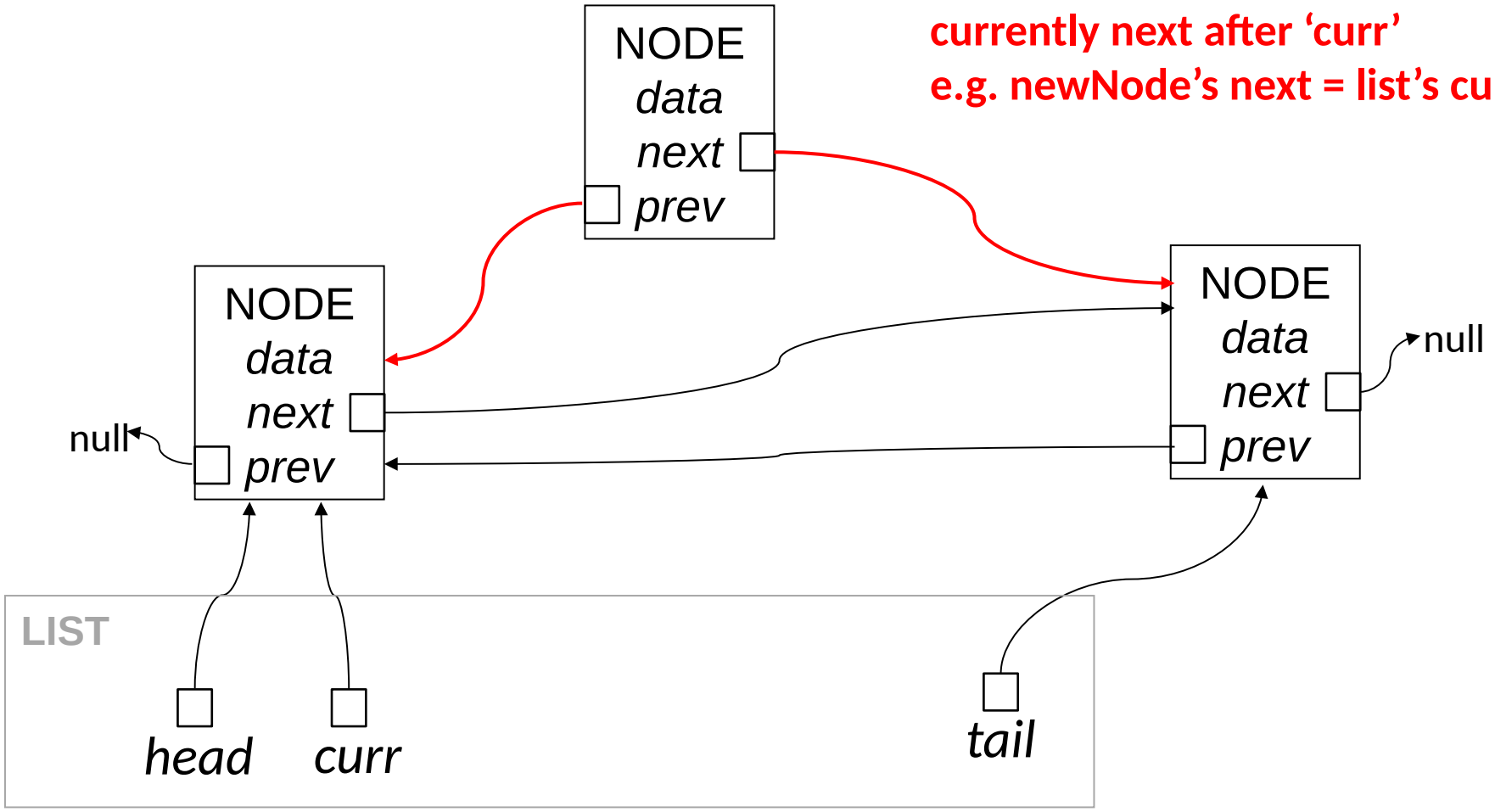
Let’s make some space (visually) between the existing nodes

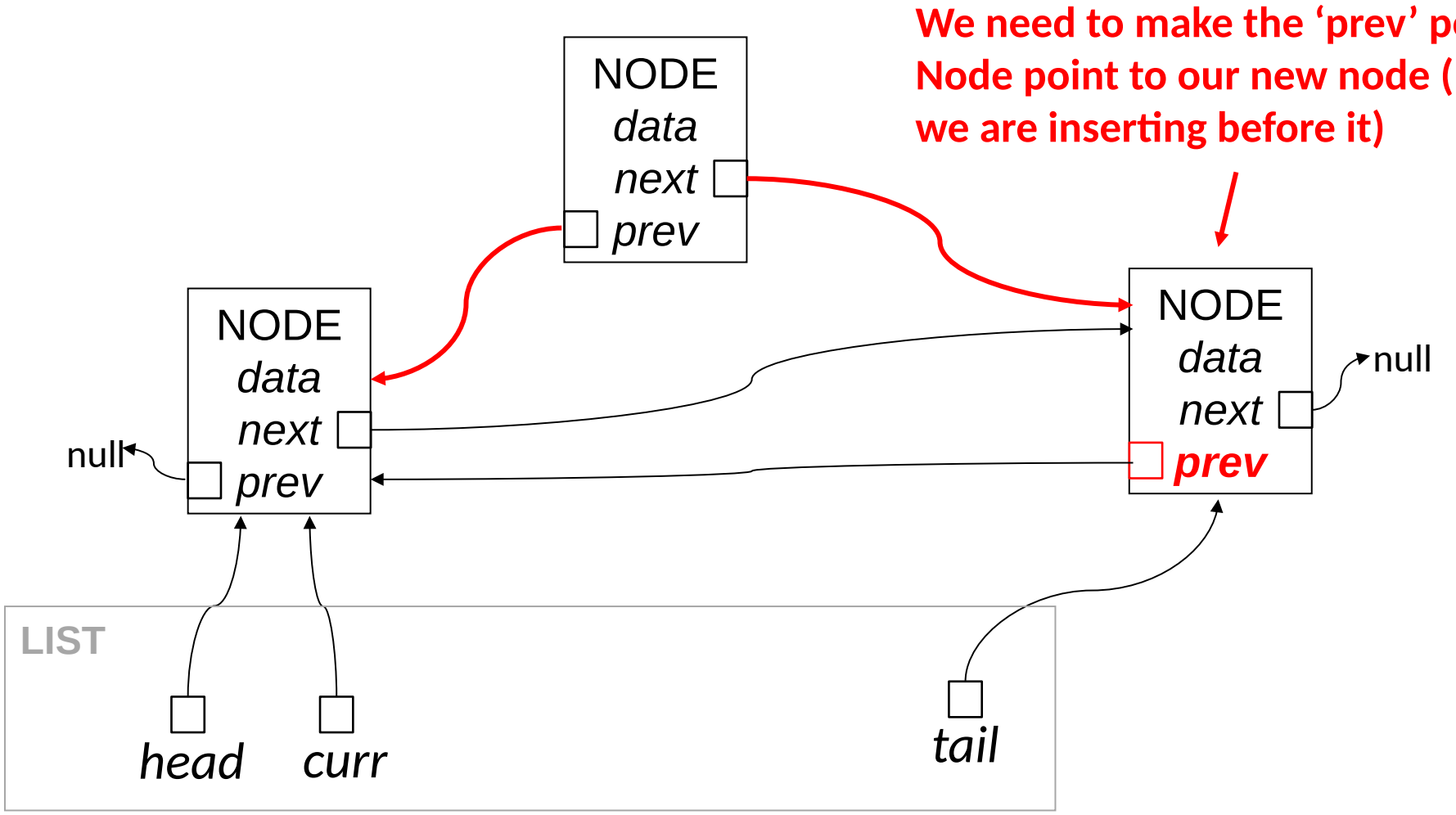


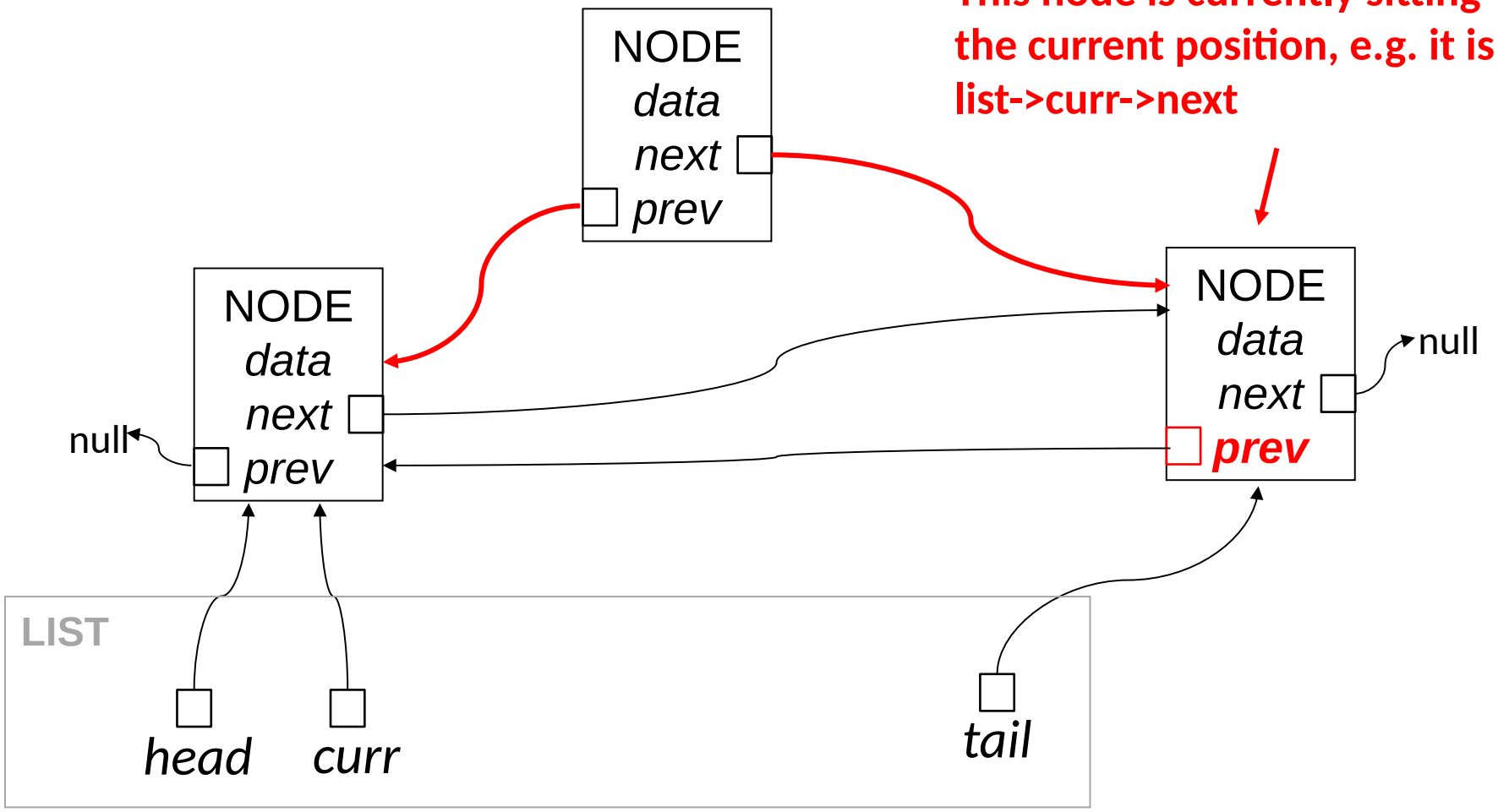

NODE
data
next
prev
null
null
Create a new node, set it's data, etc.
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail



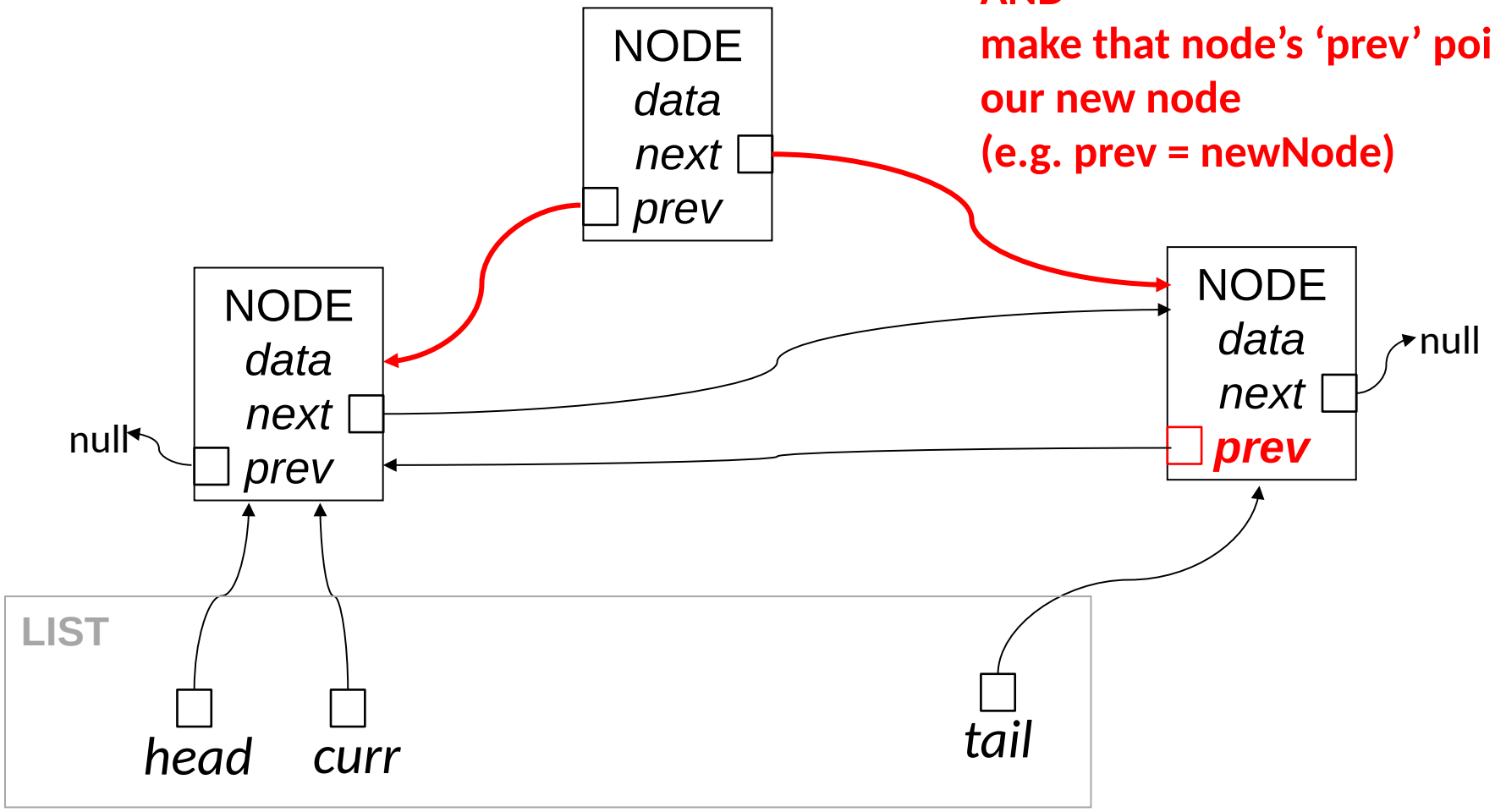
New node's 'prev' is set to 'curr' – we are inserting it after 'curr'
E.g. newNode's prev = list's curr

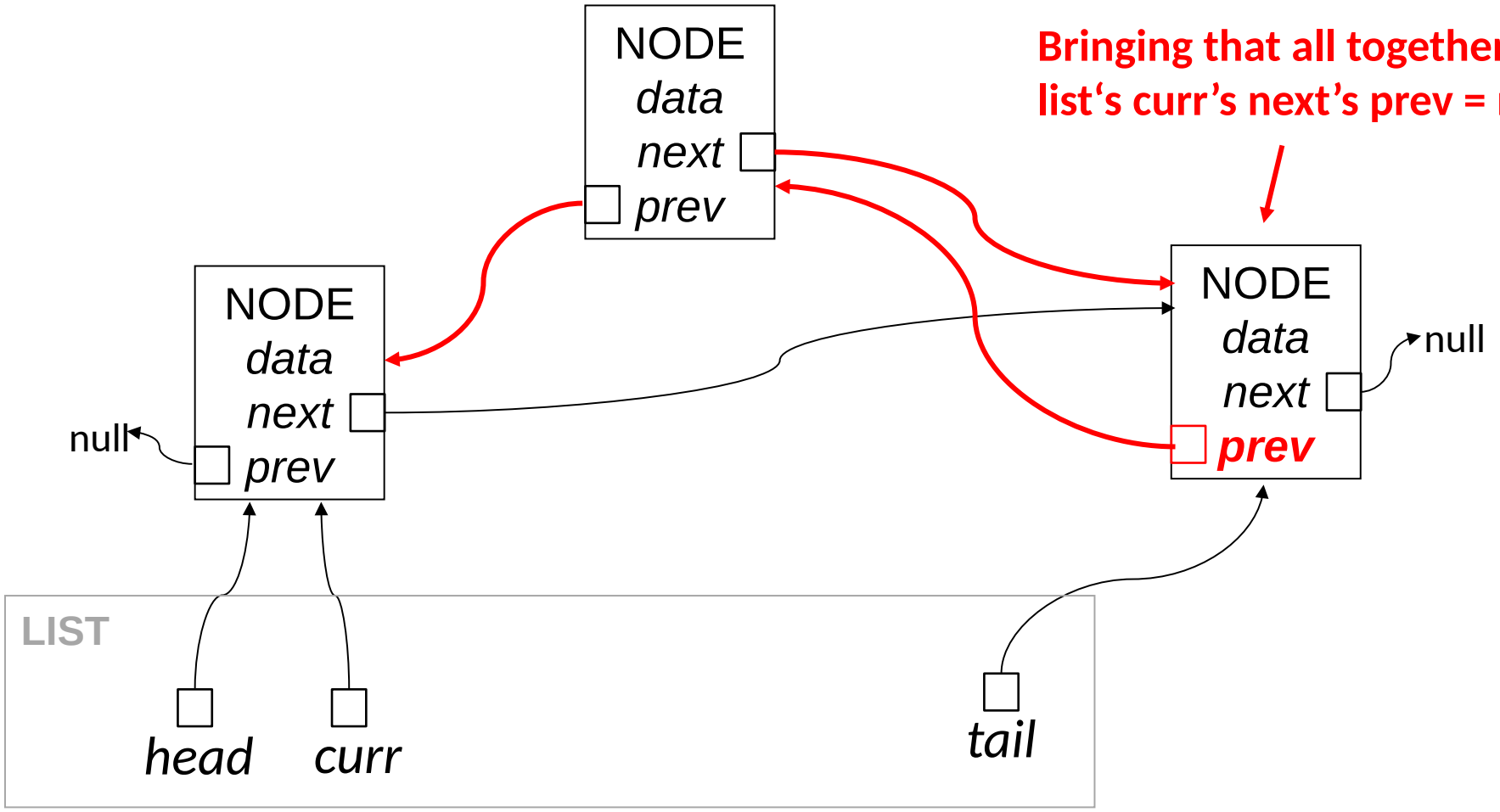

New node's 'next' is set to whatever is currently next after 'curr'
e.g. newNode's next = list's curr's next
NODE
data
next
prev
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail


We need to make the ‘prev’ pointer of this Node point to our new node (because we are inserting before it)
NODE
data
next
prev
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

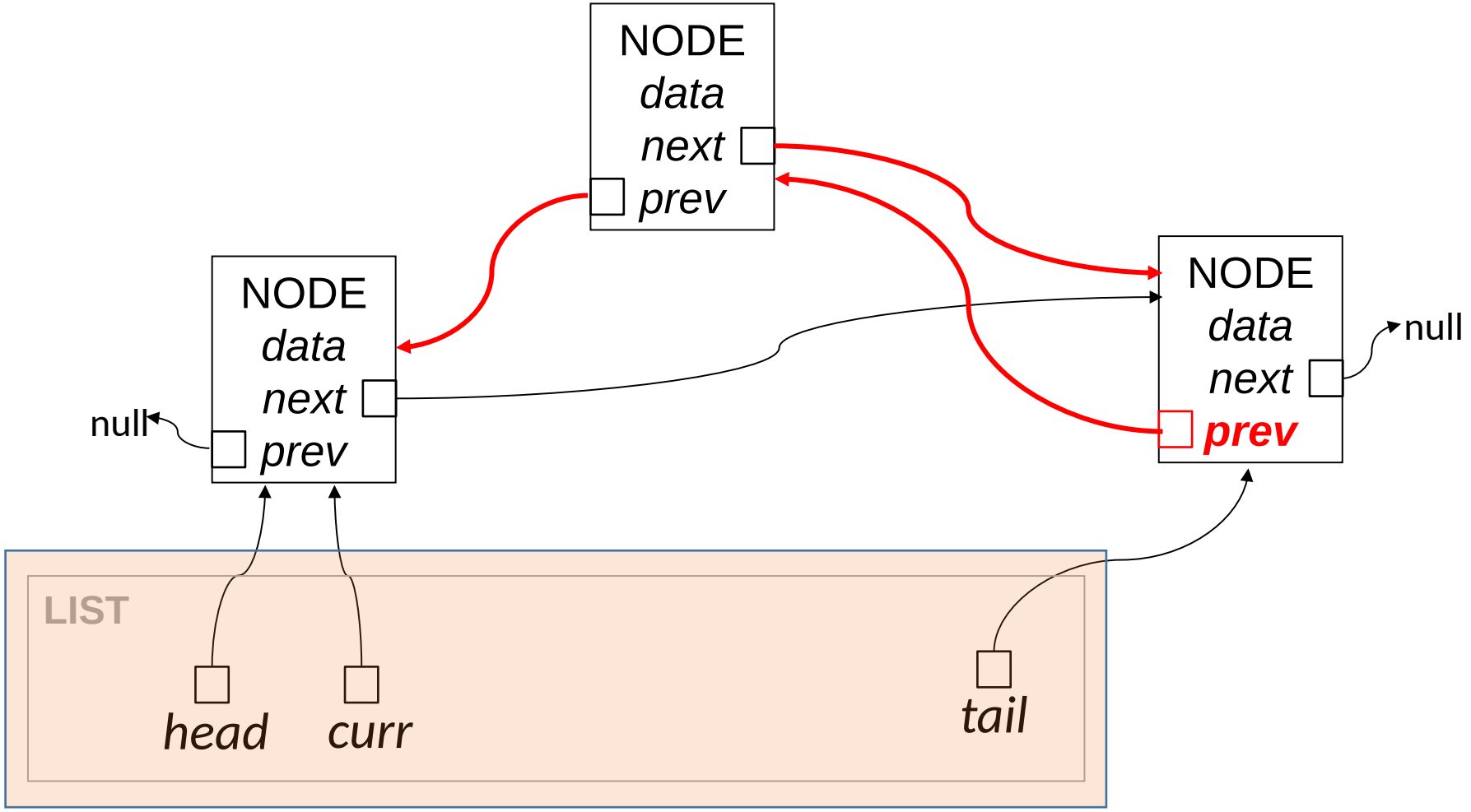

This node is currently sitting ‘next’ after the current position, e.g. it is list->curr->next
NODE
data
next
prev
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

We need to do the following:
Go to the next node after ‘curr’
(that’s: list’s curr’s next)
AND
make that node’s ‘prev’ point to
our new node
(e.g. prev = newNode)

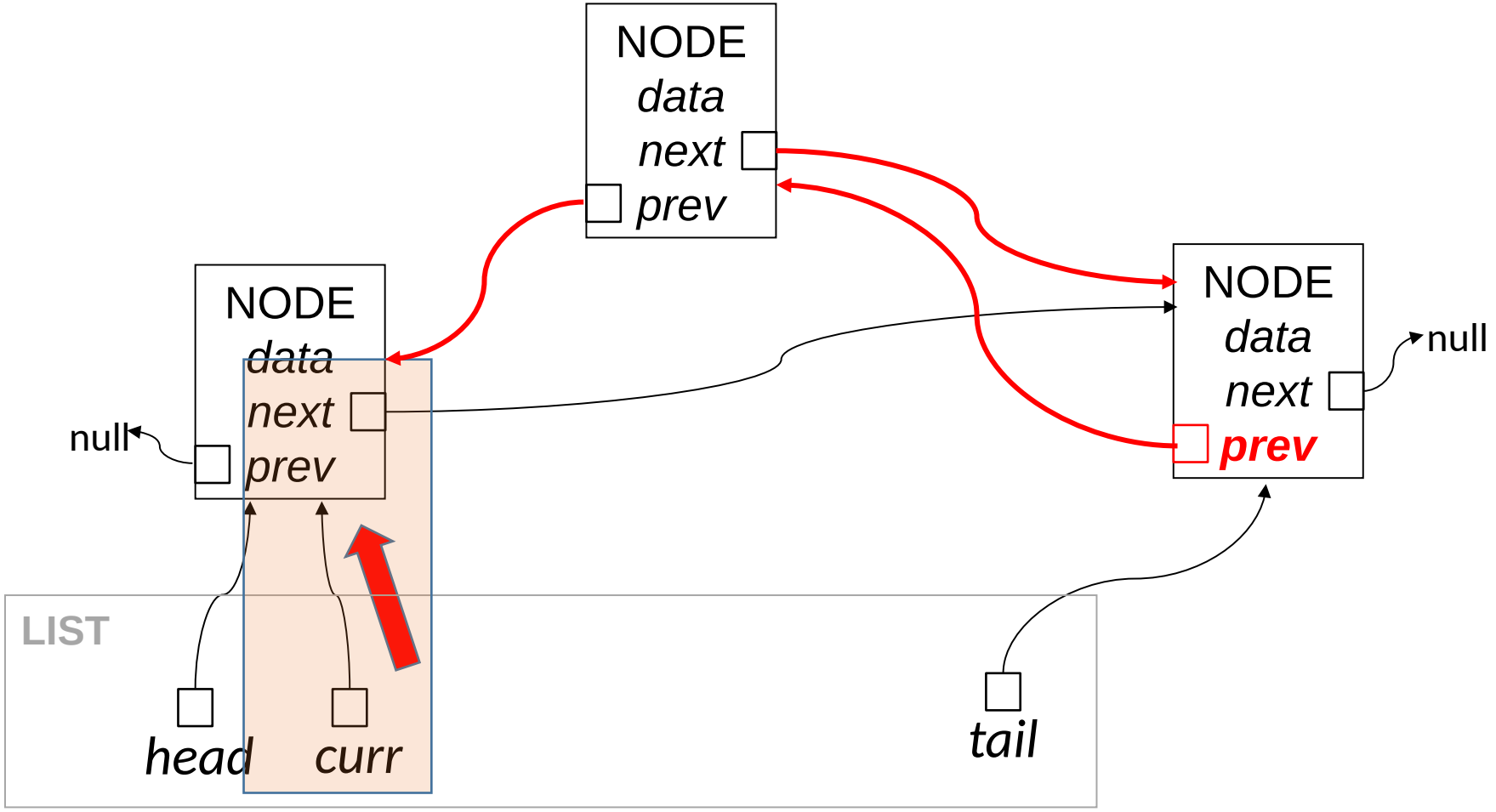



NODE
data
next
prev
Bringing that all together:
list‘s curr’s next’s prev = newNode!!
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

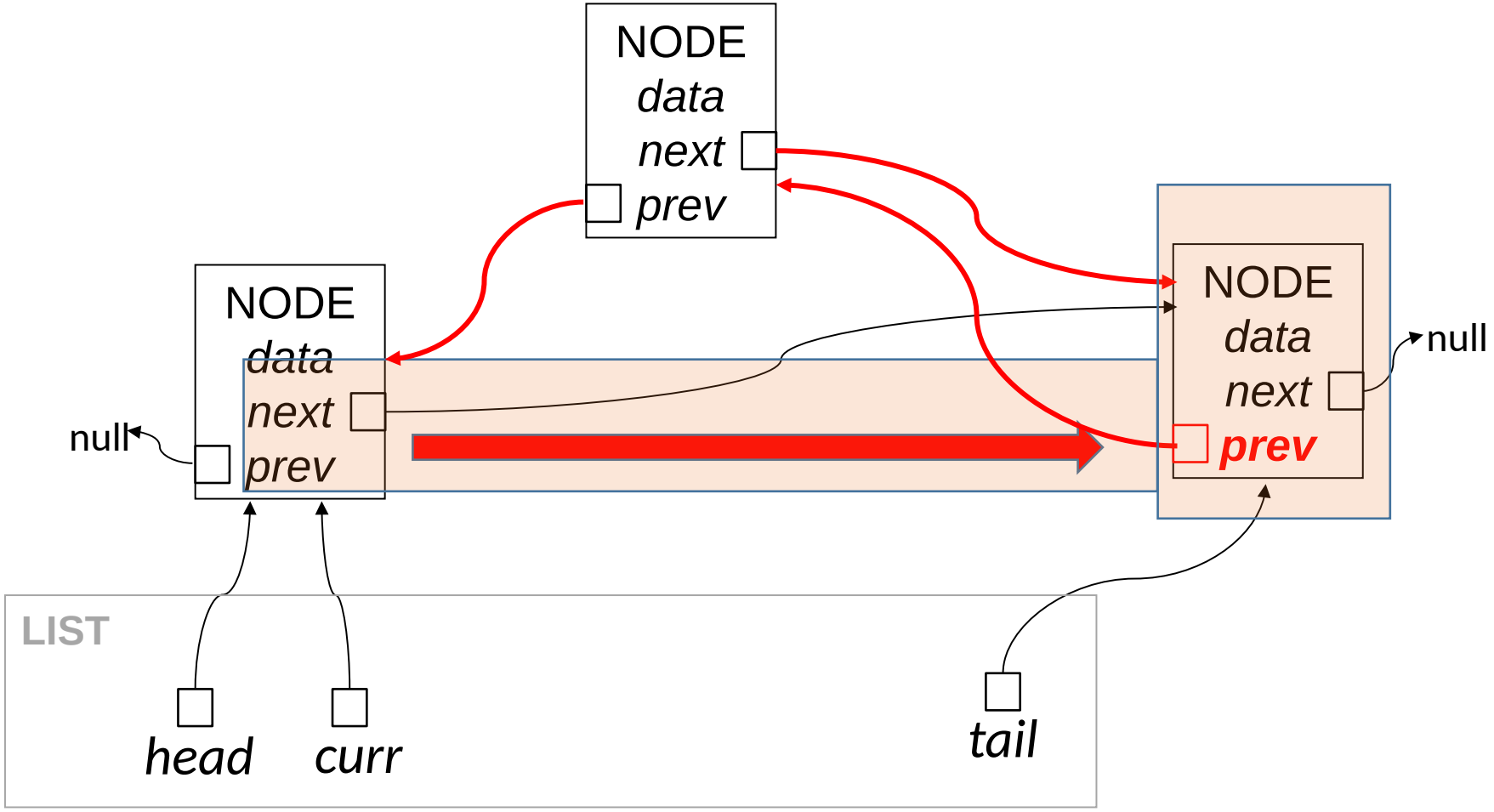
List's curr's next's prev = newNode!!



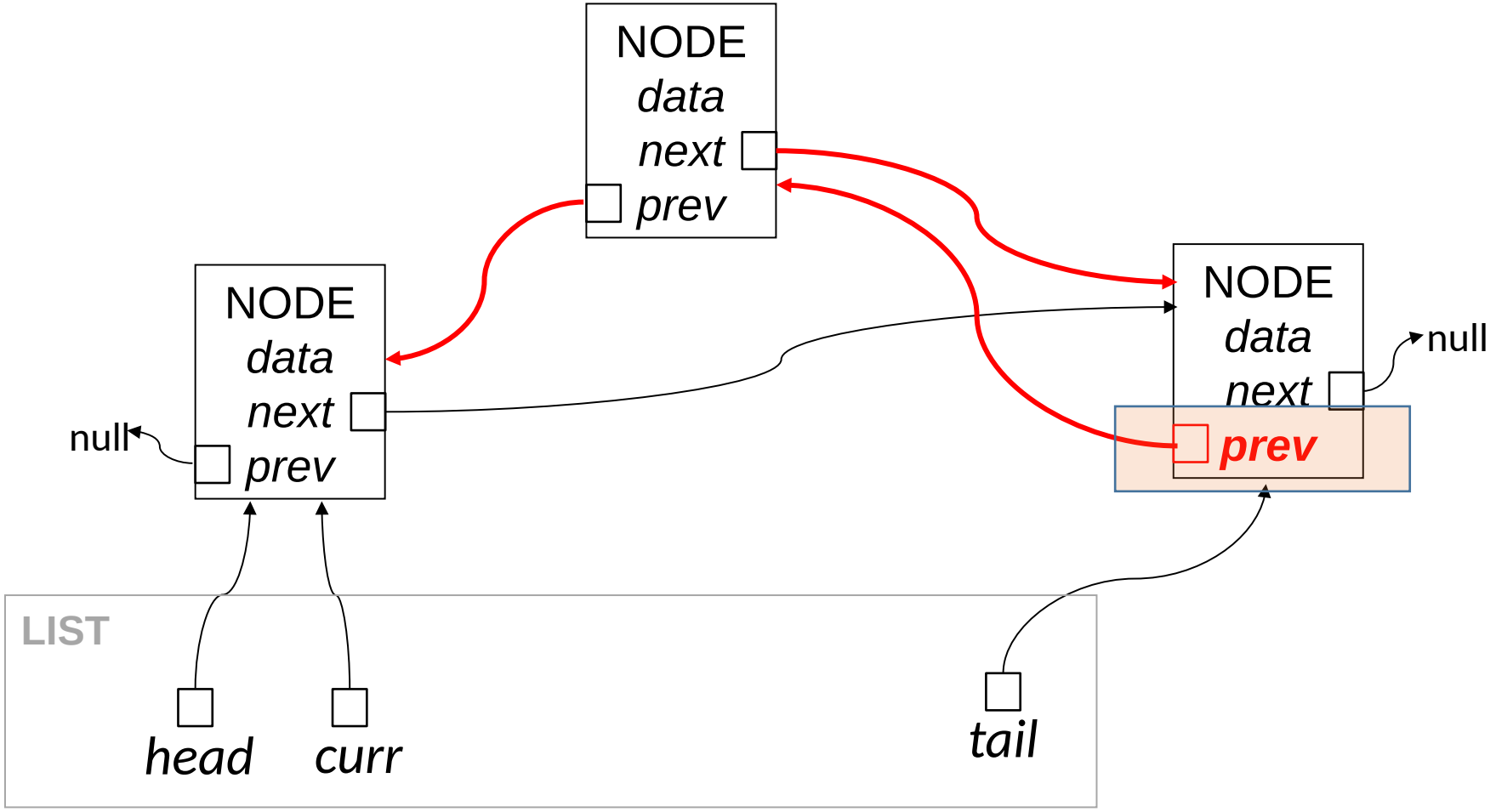
List's curr's next's prev = newNode!!



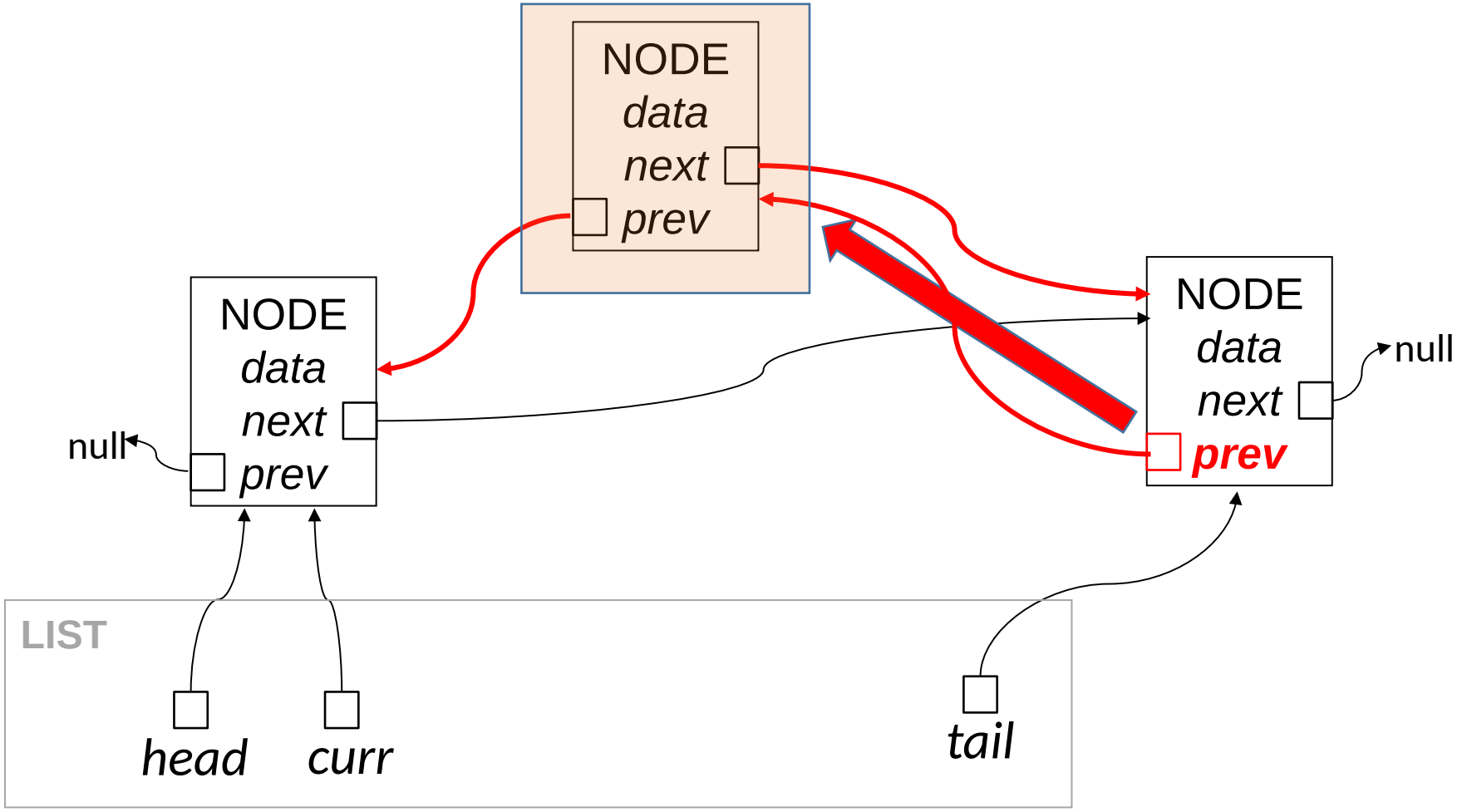
List's curr's **next's** prev = newNode!!



List's curr's next's **prev** = newNode!!

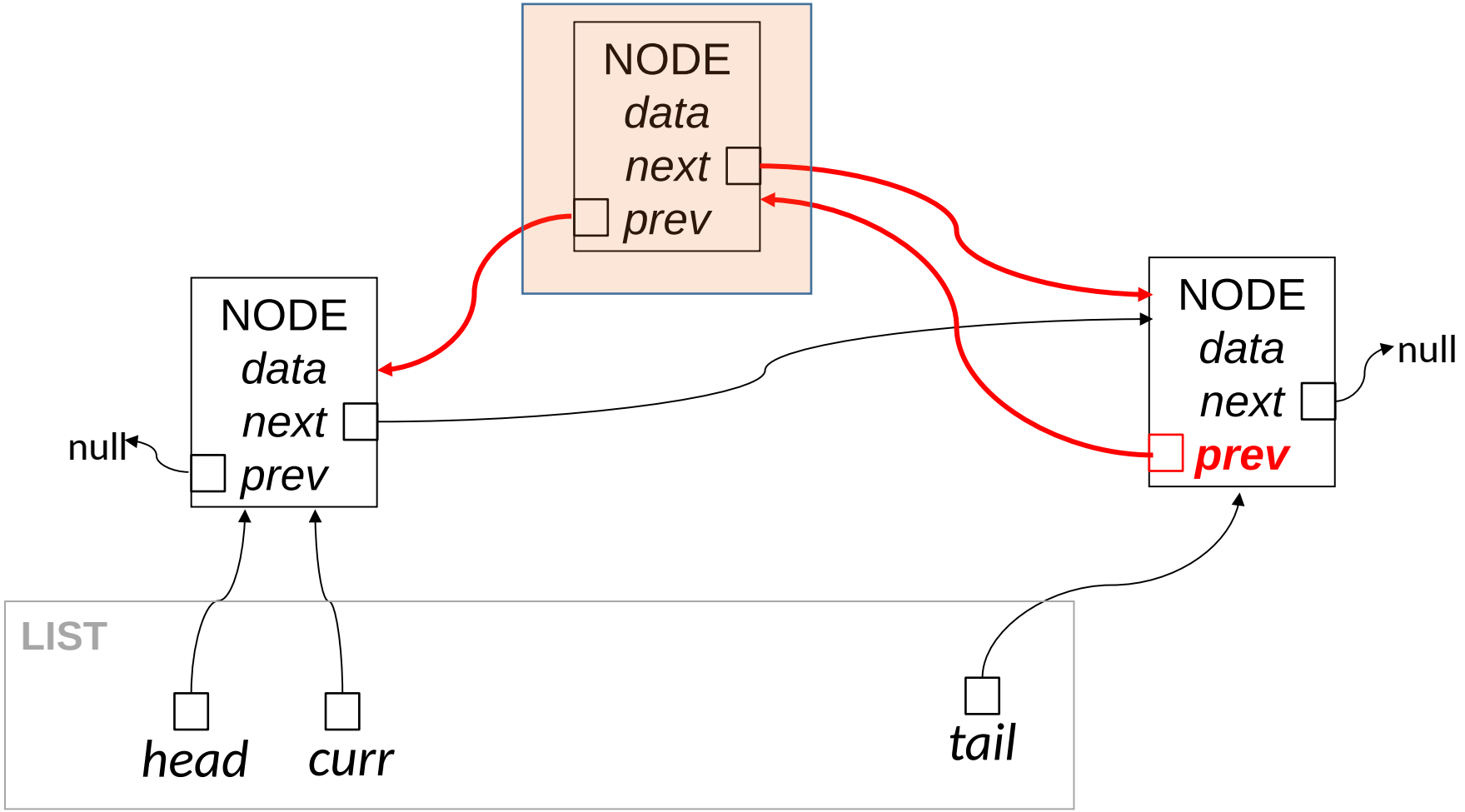


List's curr's next's prev = **newNode!!**

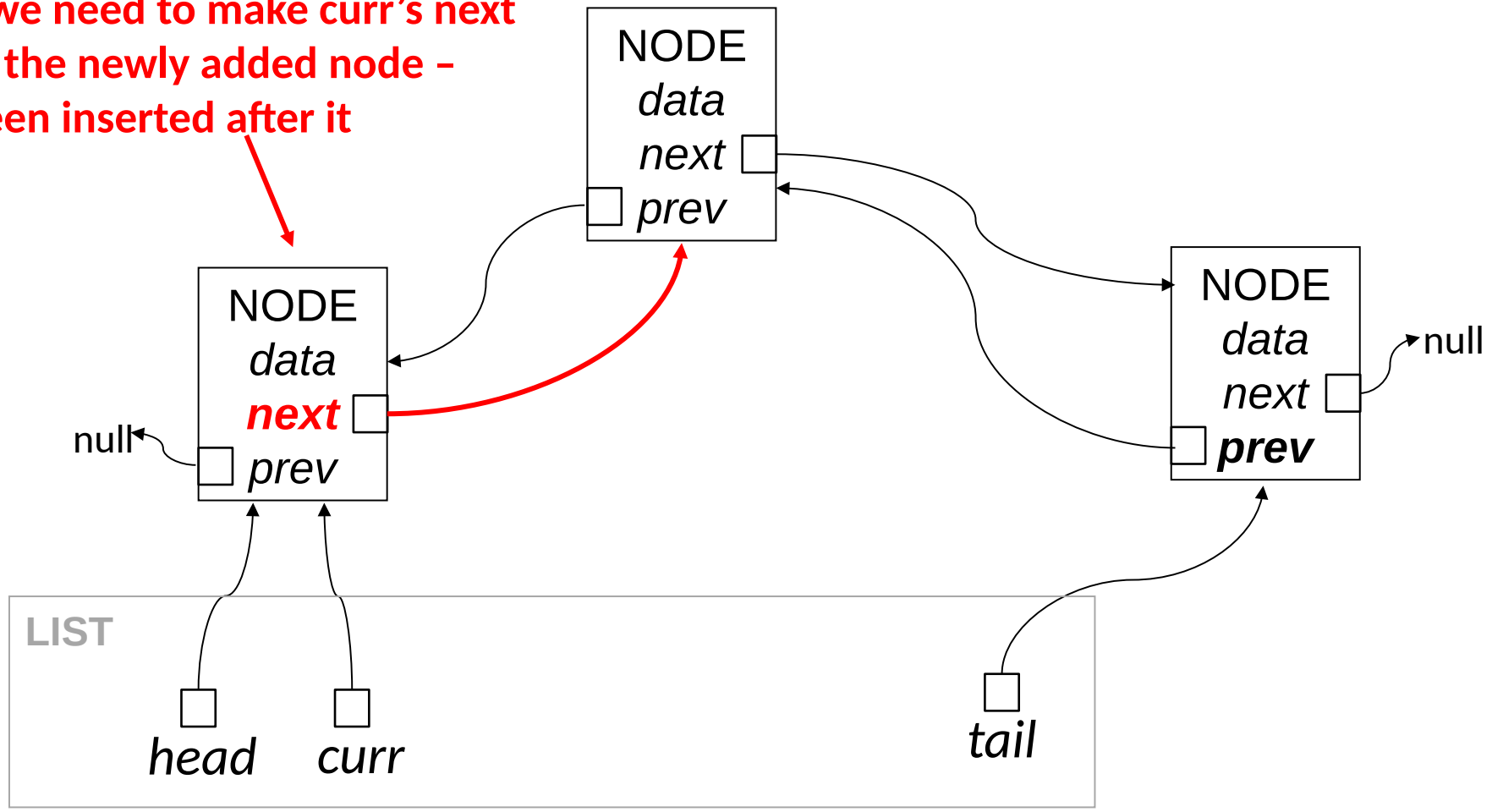


Set the list's, current node's, next node's prev field to point to the new node

list->curr->next->prev = ??



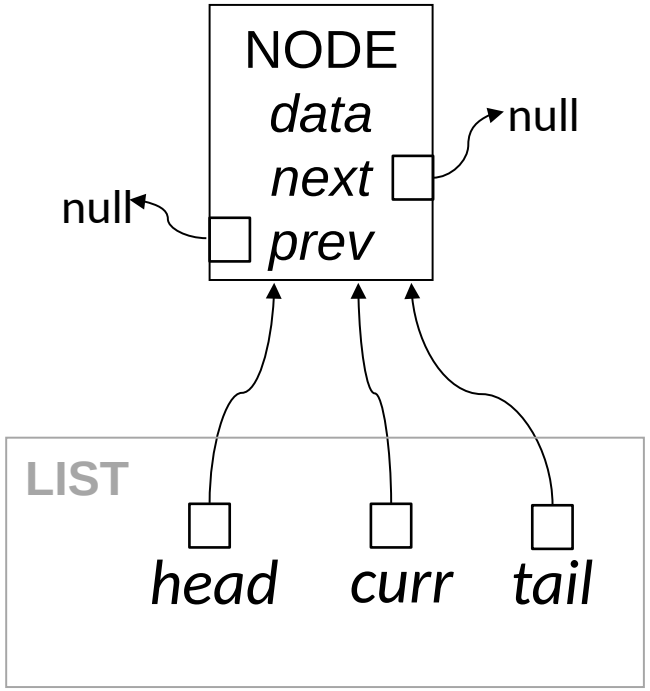
Finally, we need to make curr's next point to the newly added node – it has been inserted after it



Adding a node **before** the 'current' position

- Two cases to consider:
  - (i) 'current' is at the head of the list – so, we will be inserting a new head
  - (ii) 'current' is in the middle of the list (there's at least one Node prior to it) – so, we will be inserting a new node in between two existing nodes

Add a node **before** the current position

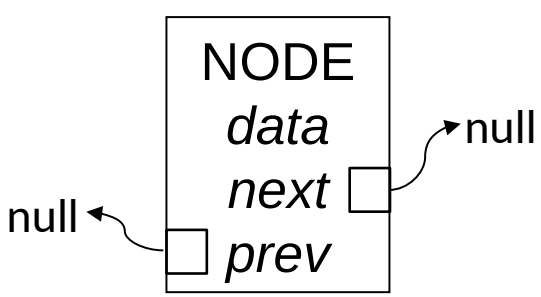


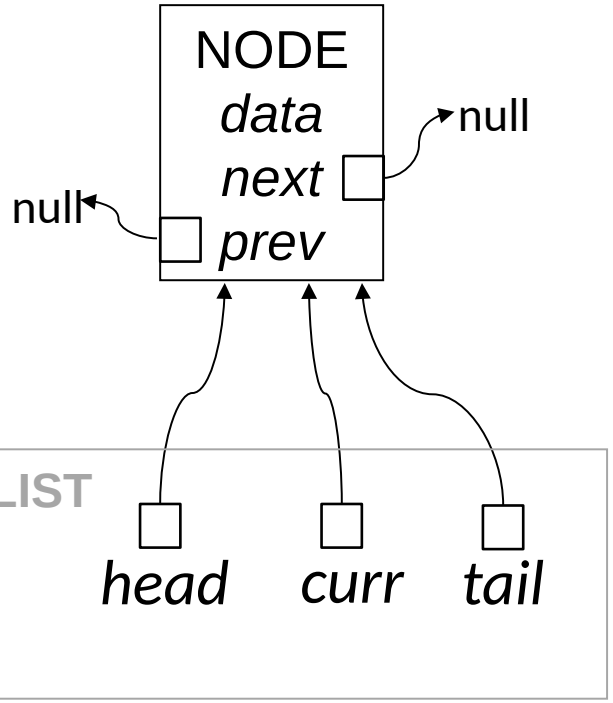
Case (i) – ‘current’ is at the ‘head’ of the list

e.g. list->curr == list->head

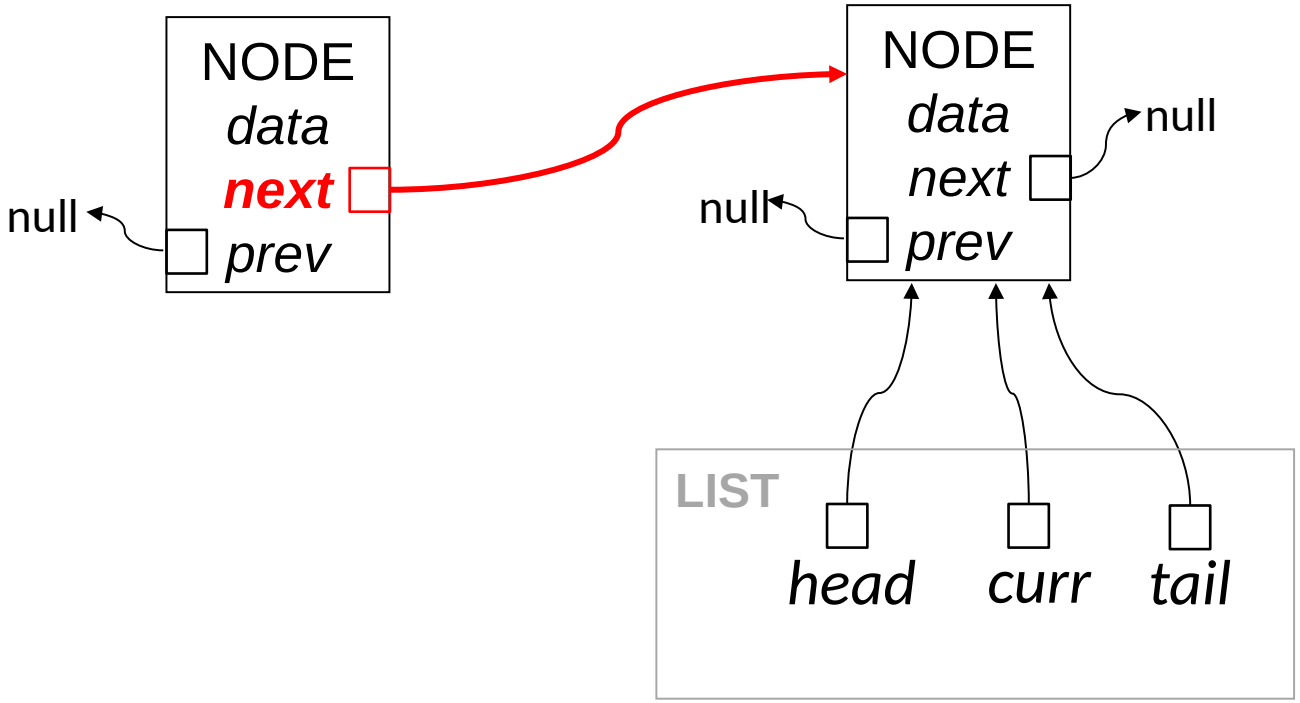
and list->curr->prev == NULL

Create a new node, set it's data, etc.

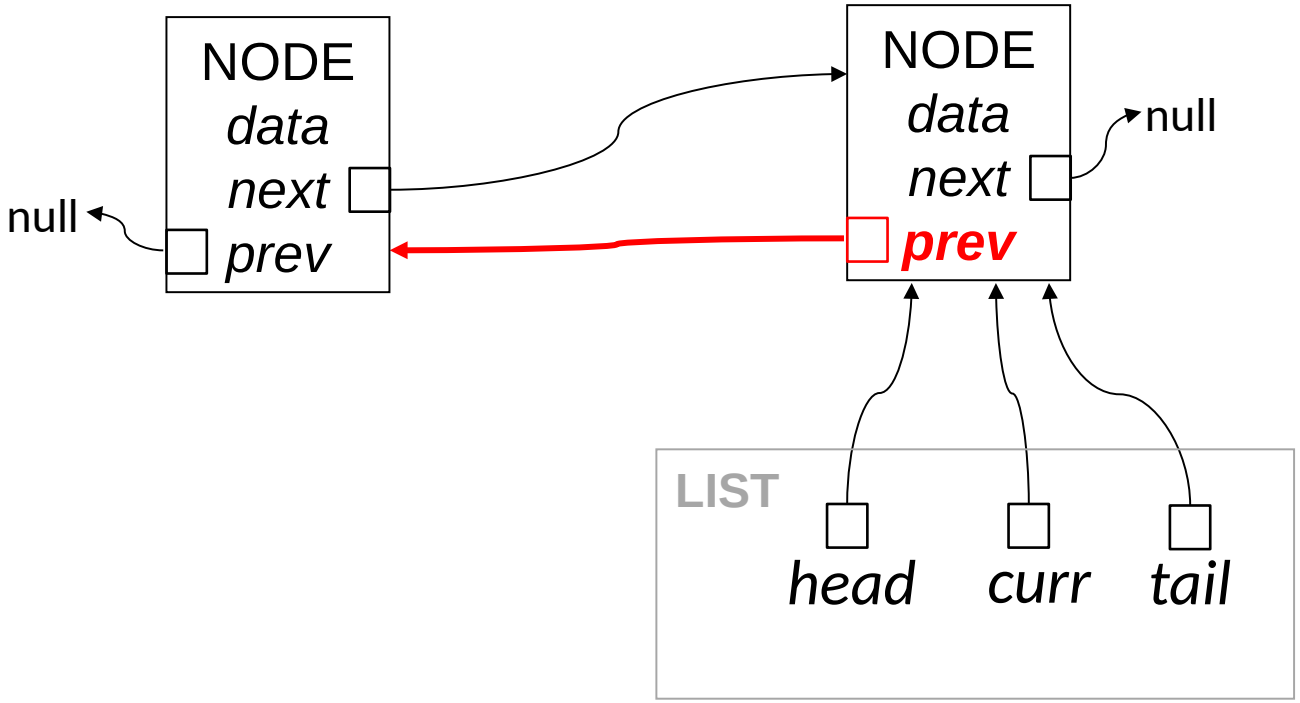




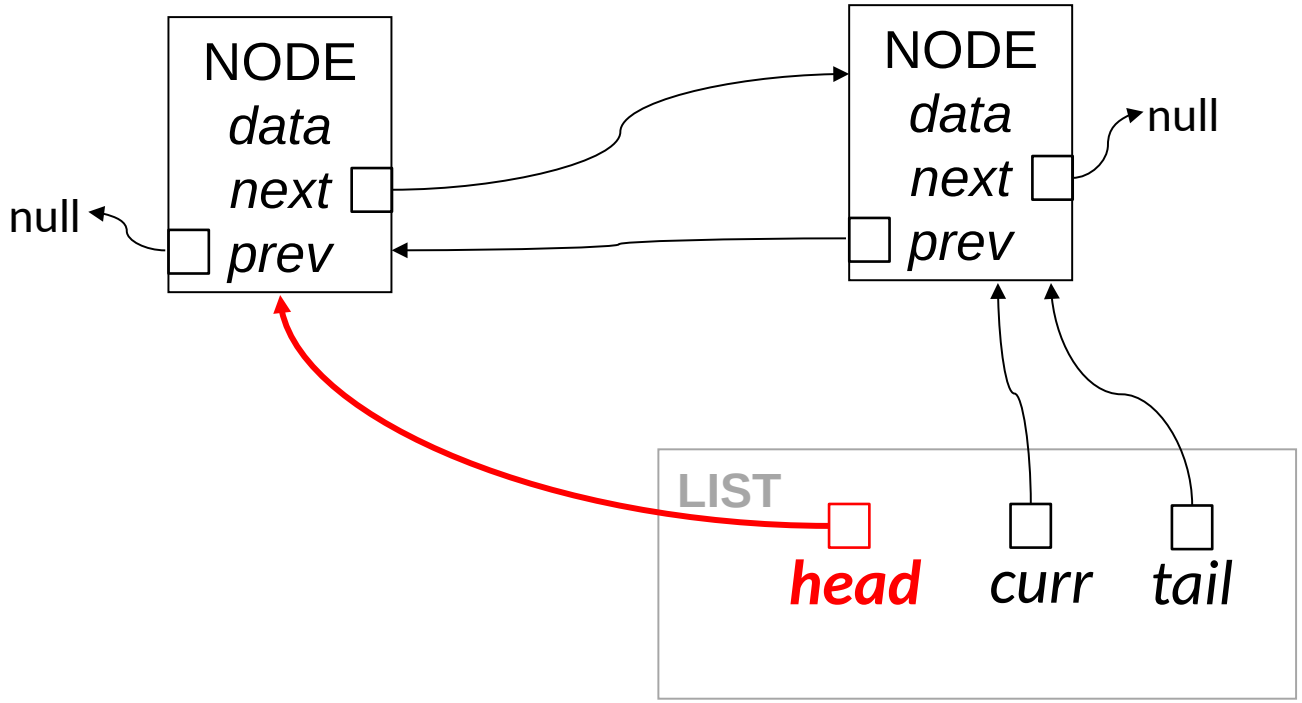
New node's next is set to 'curr' – we are inserting before 'curr'
E.g. make newNode's next point to the same node as list's curr



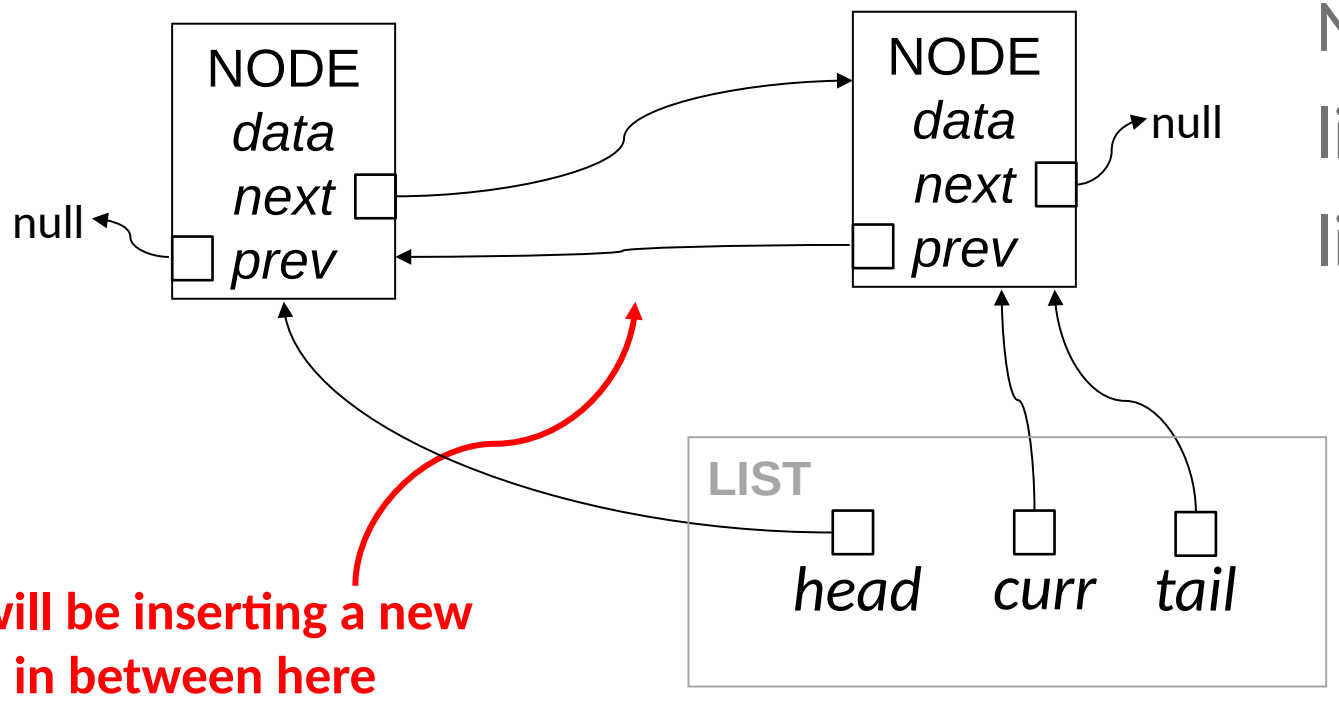
Curr's 'prev' is set to newNode
e.g. list's current node's previous = newNode



New node becomes the new head of the list
E.g. list's head = newNode



Add a node **before** the current position

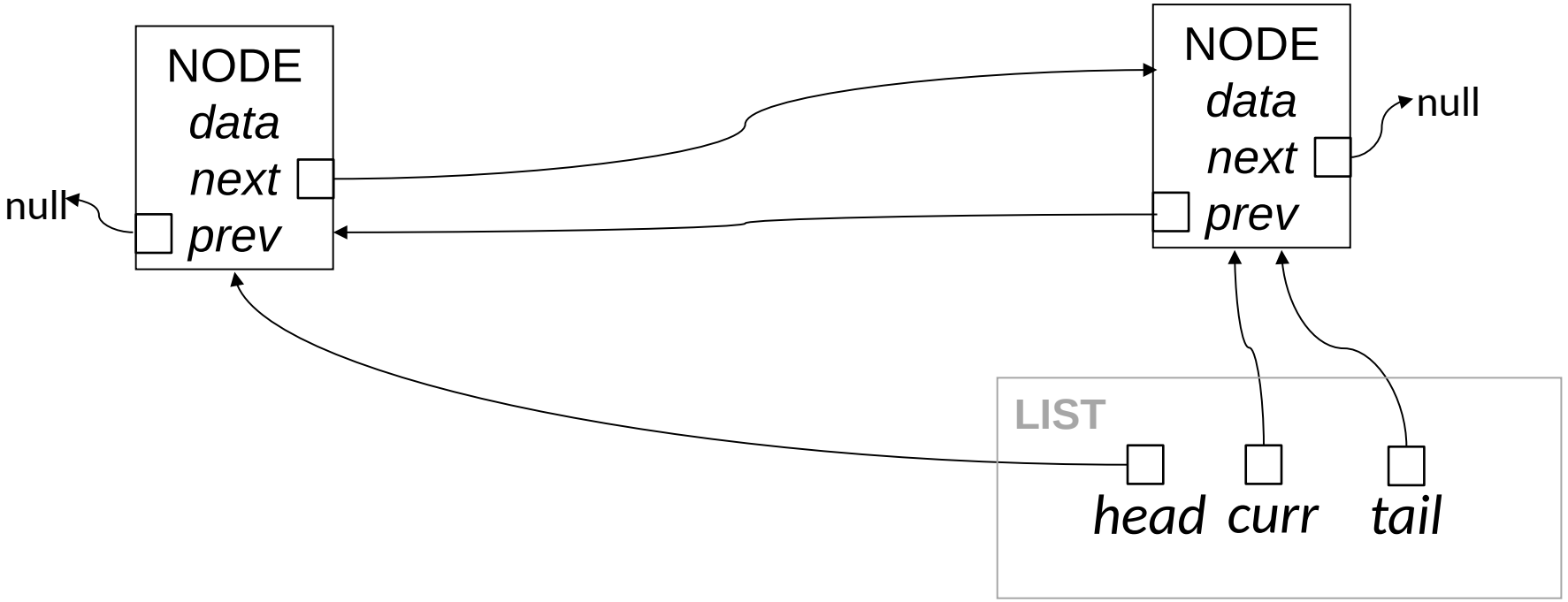


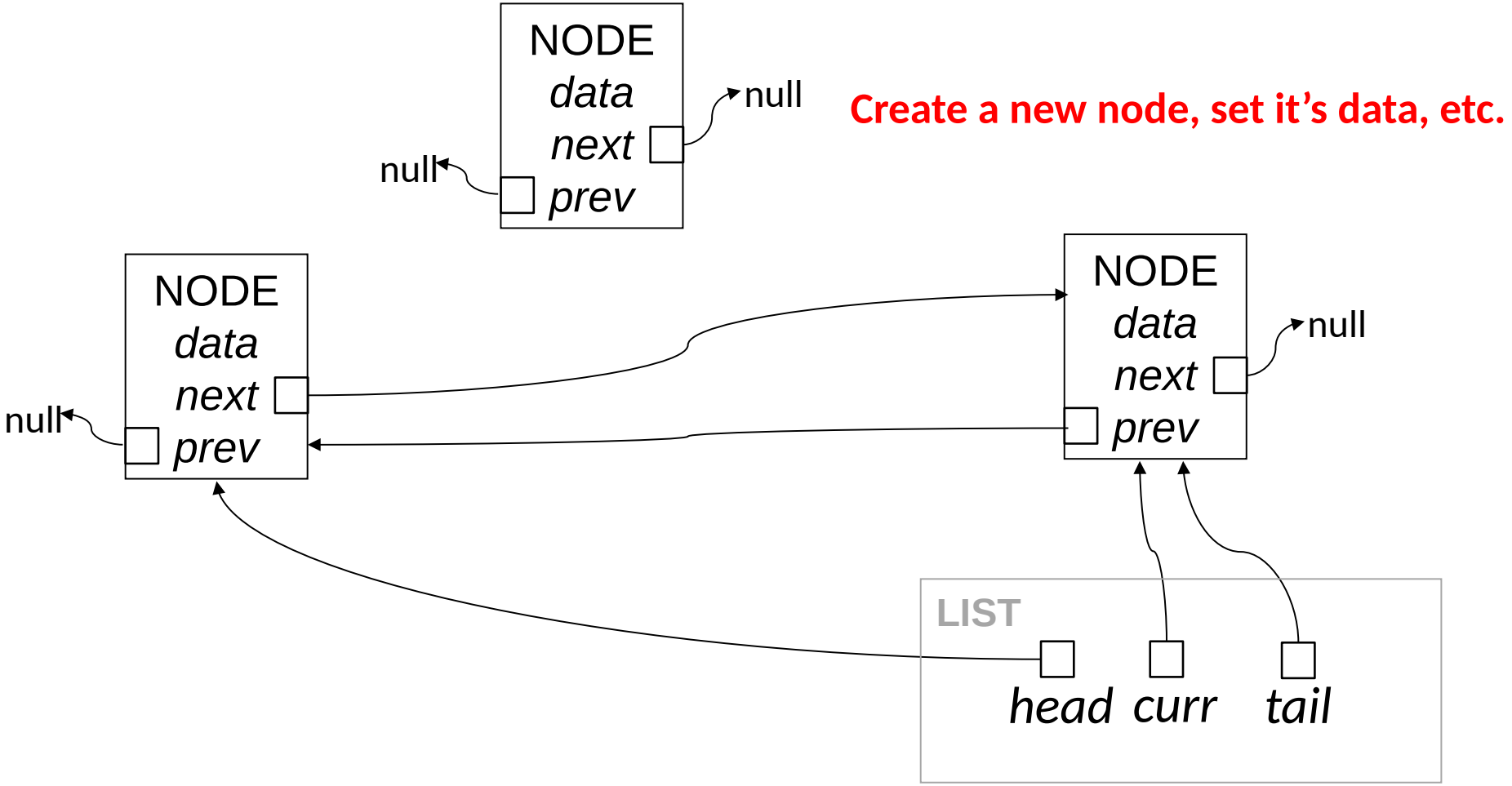
Case (ii) – 'current' is NOT the 'head' of the list

list->head != list->curr

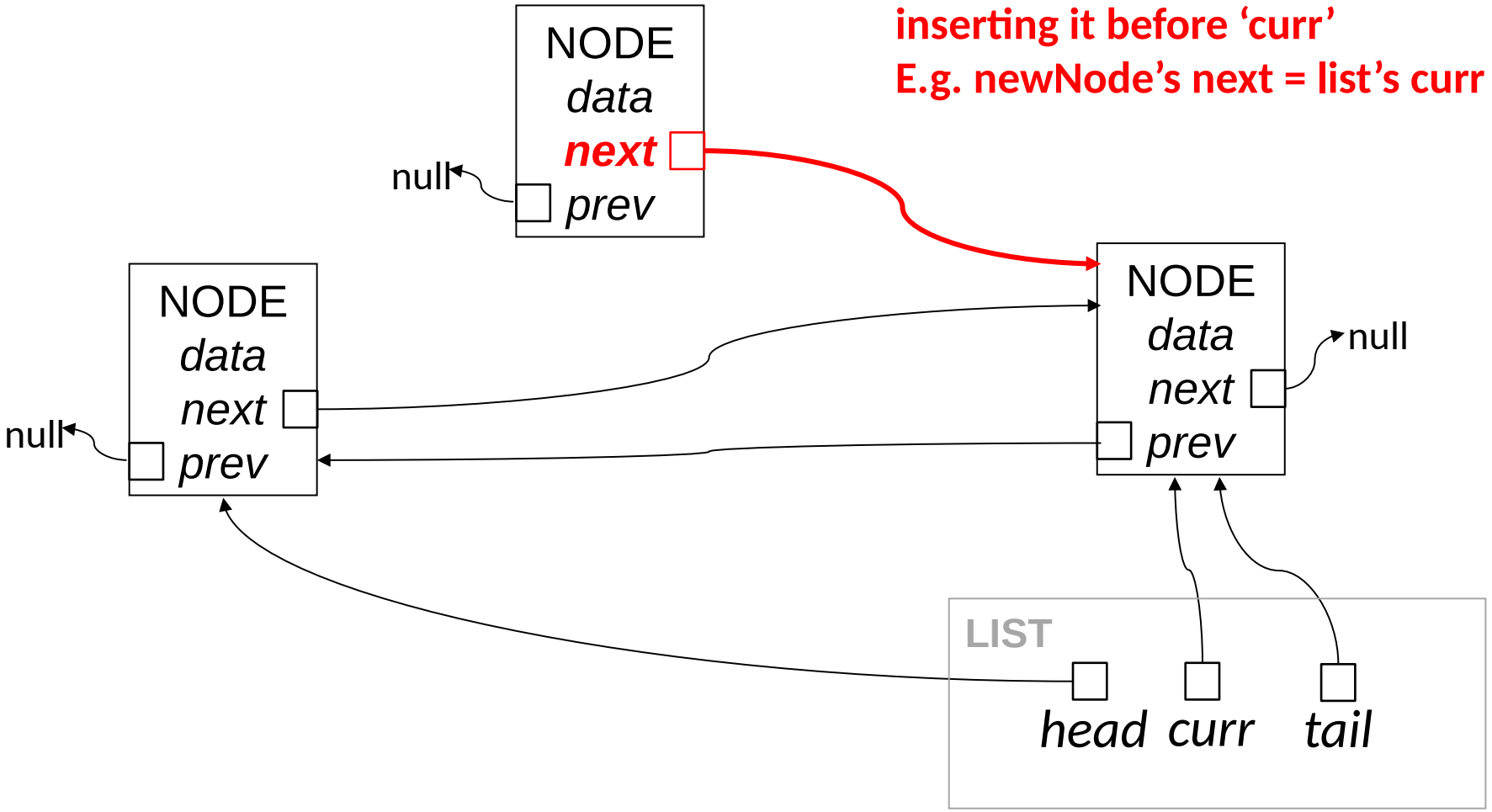
list->curr->prev != NULL

Let's make some space (visually) between the existing nodes

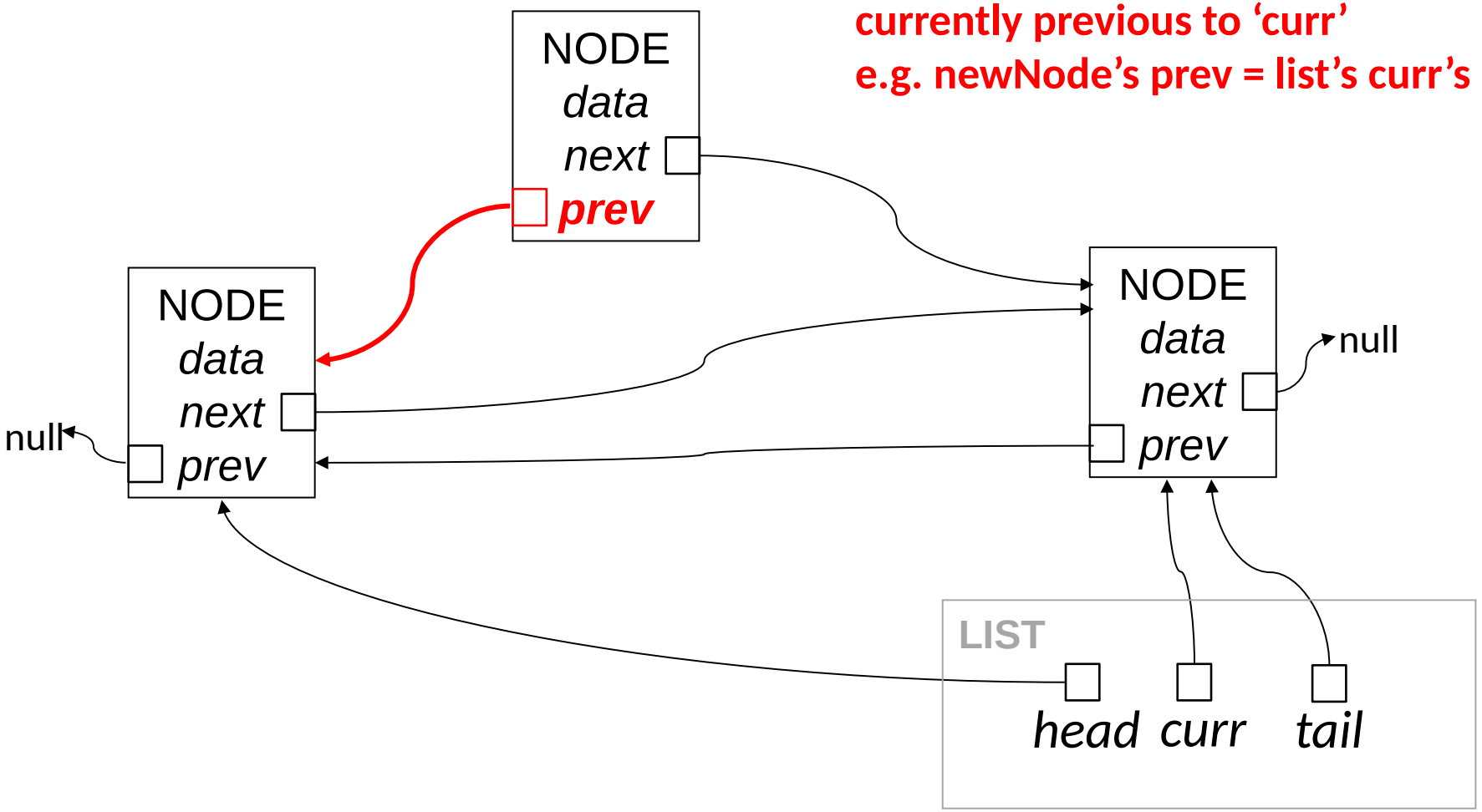



NODE
data
next
prev
null
null
Create a new node, set it's data, etc.
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

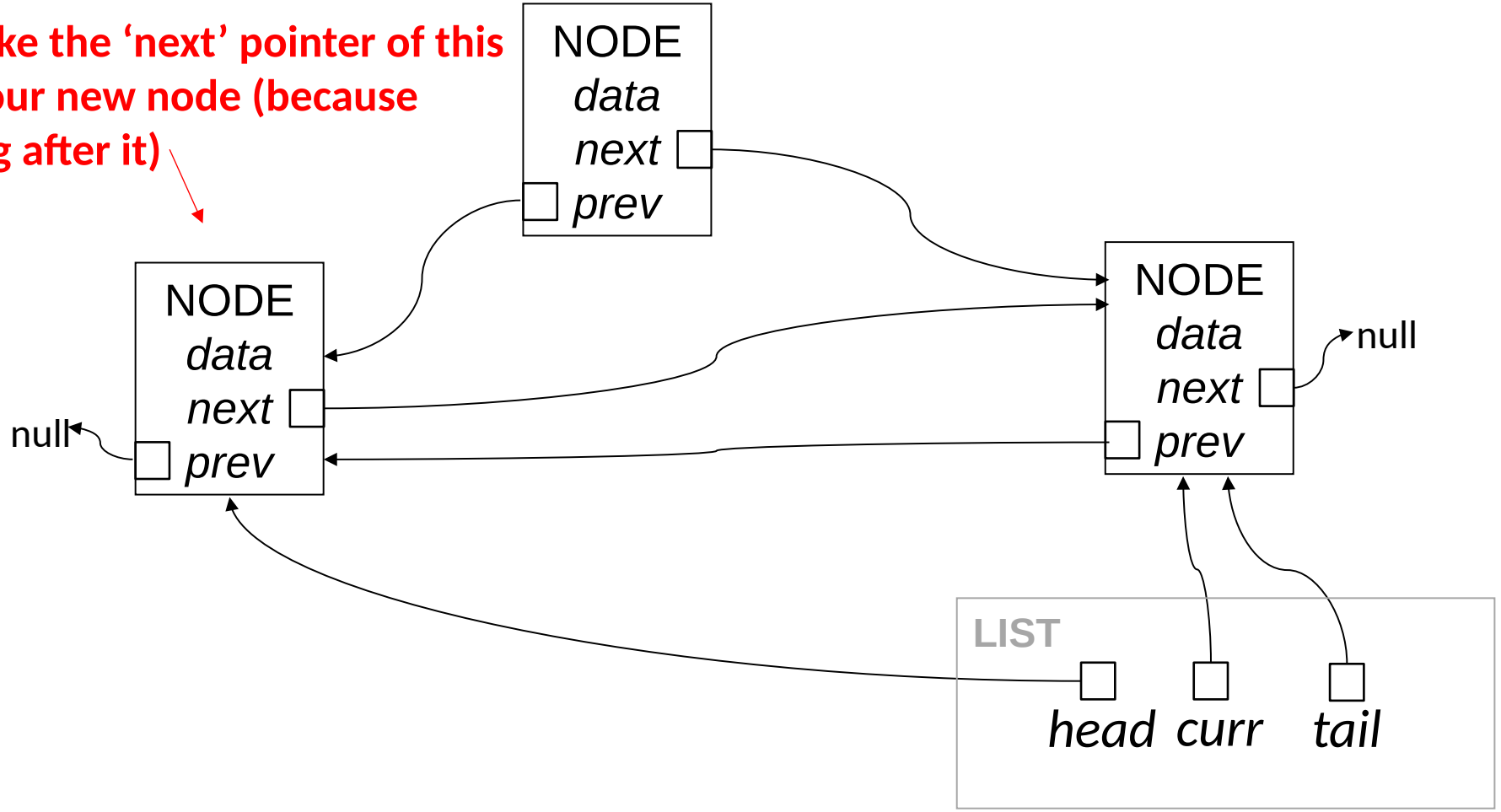
New node's 'next' is set to 'curr' – we are inserting it before 'curr'
E.g. newNode's next = list's curr



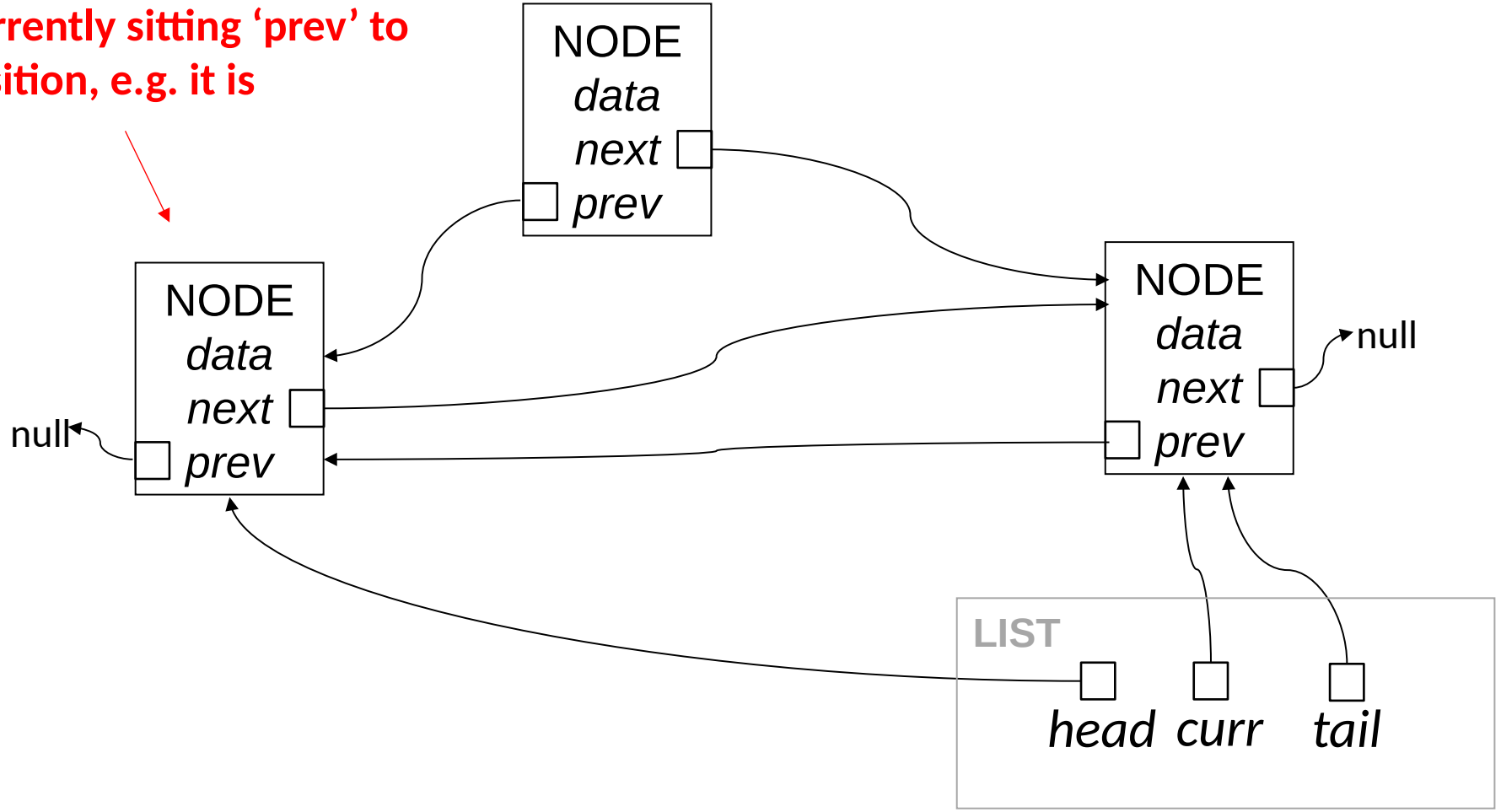
New node's 'prev' is set to whatever is currently previous to 'curr'
e.g. newNode's prev = list's curr's prev



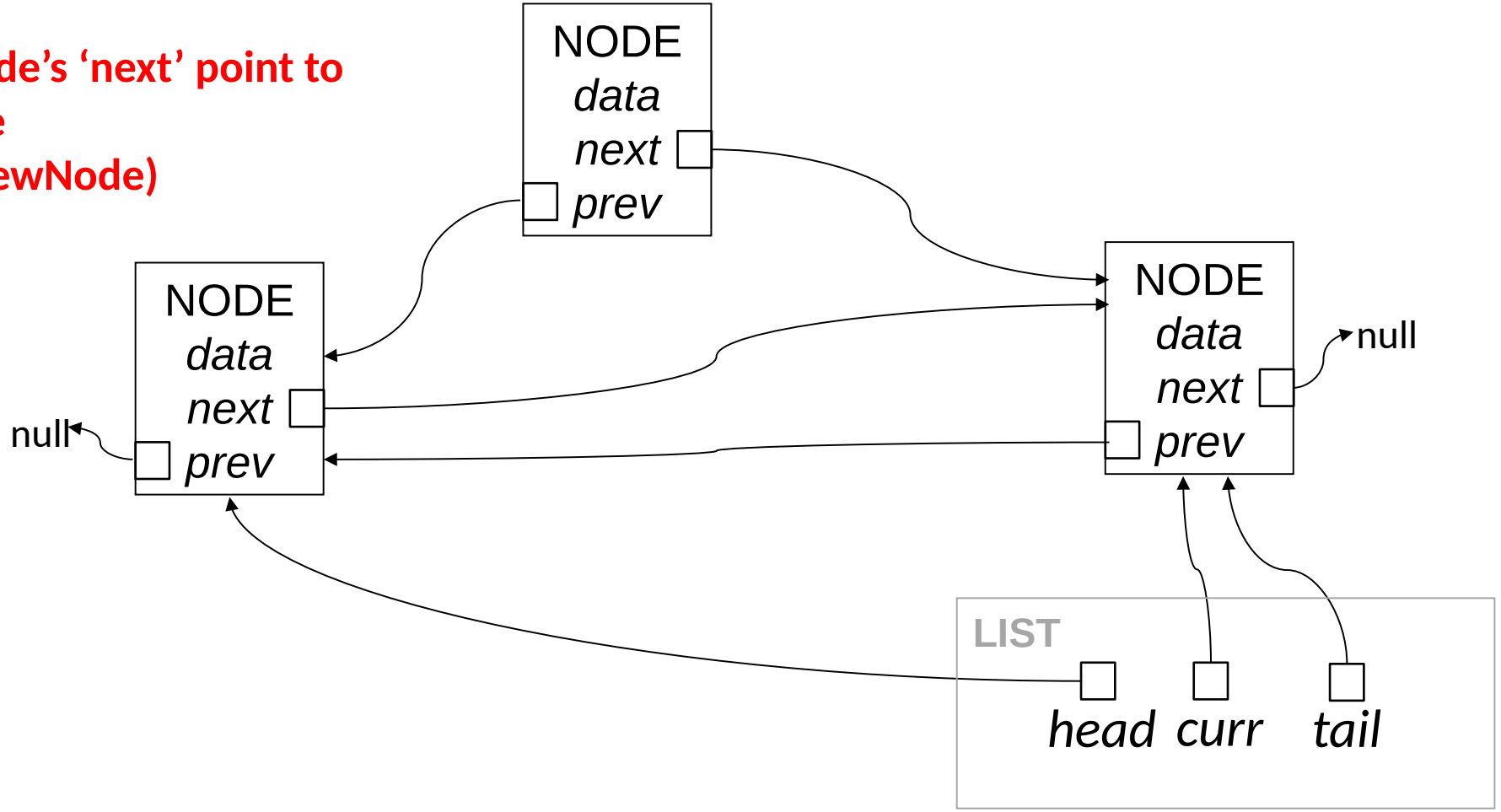
We need to make the 'next' pointer of this Node point to our new node (because we are inserting after it)



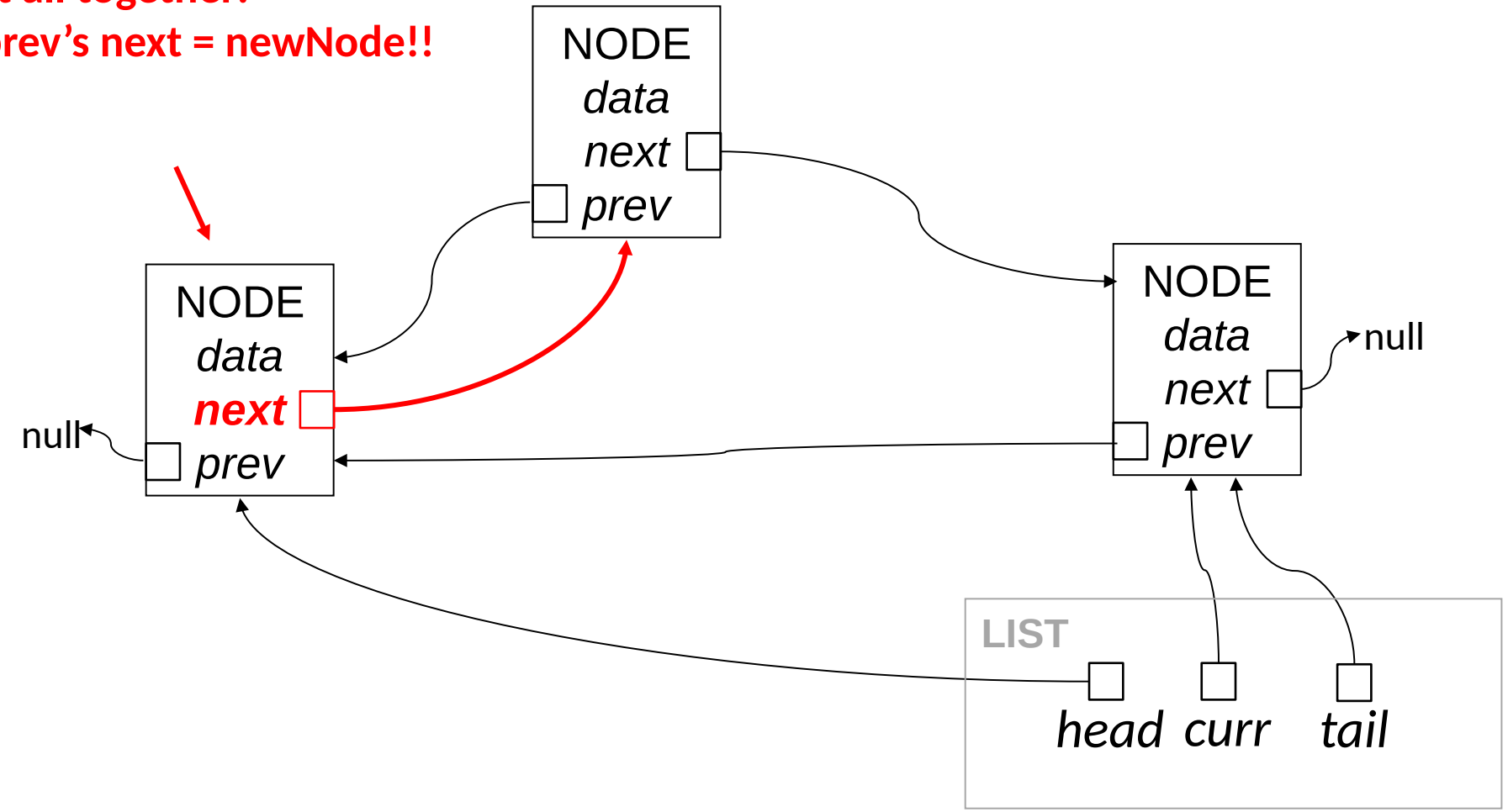
This node is currently sitting 'prev' to the current position, e.g. it is list->curr->prev

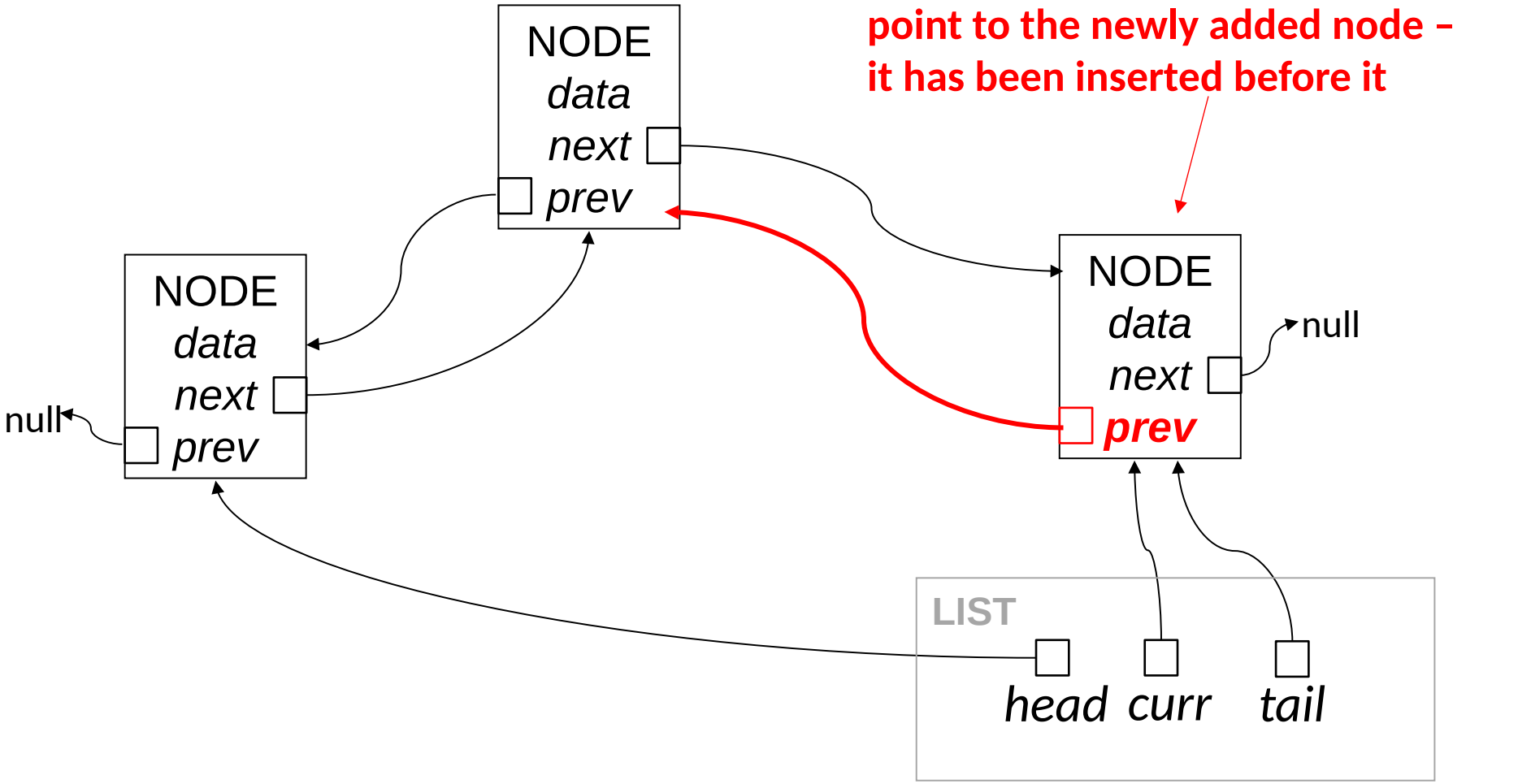


We need to do the following:
Go to the previous node before 'curr'
(that's: list->curr->prev)
AND
make that node's 'next' point to
our new node
(e.g. next = newNode)

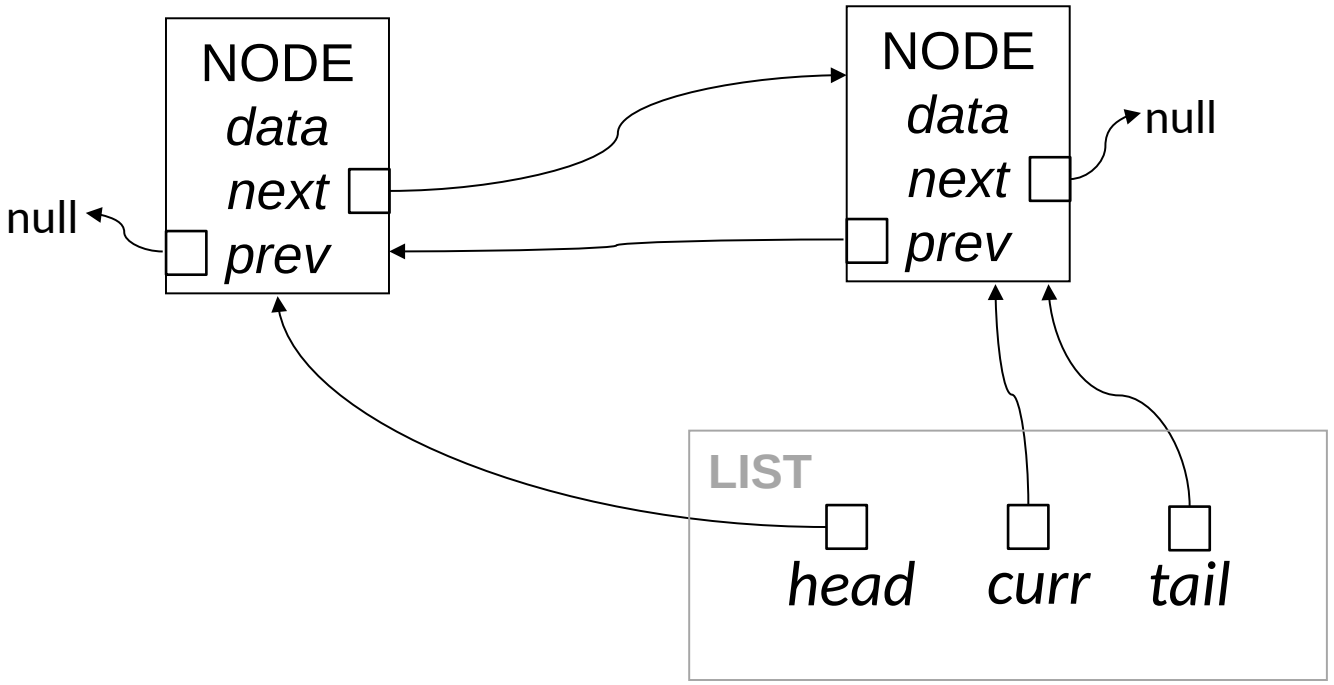


Bringing that all together:
list‘s curr’s prev’s next = newNode!!

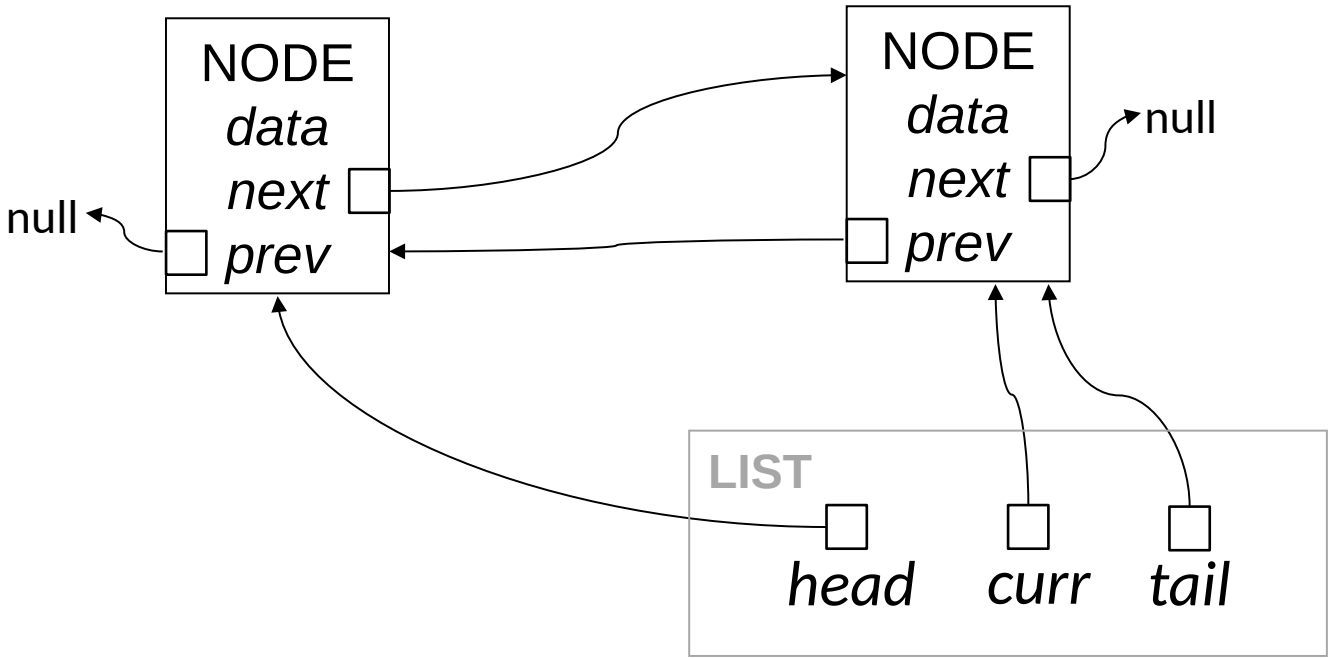



Finally, we need to make curr's prev point to the newly added node – it has been inserted before it
NODE
data
next
prev
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

Move ‘current’ one position back in the list

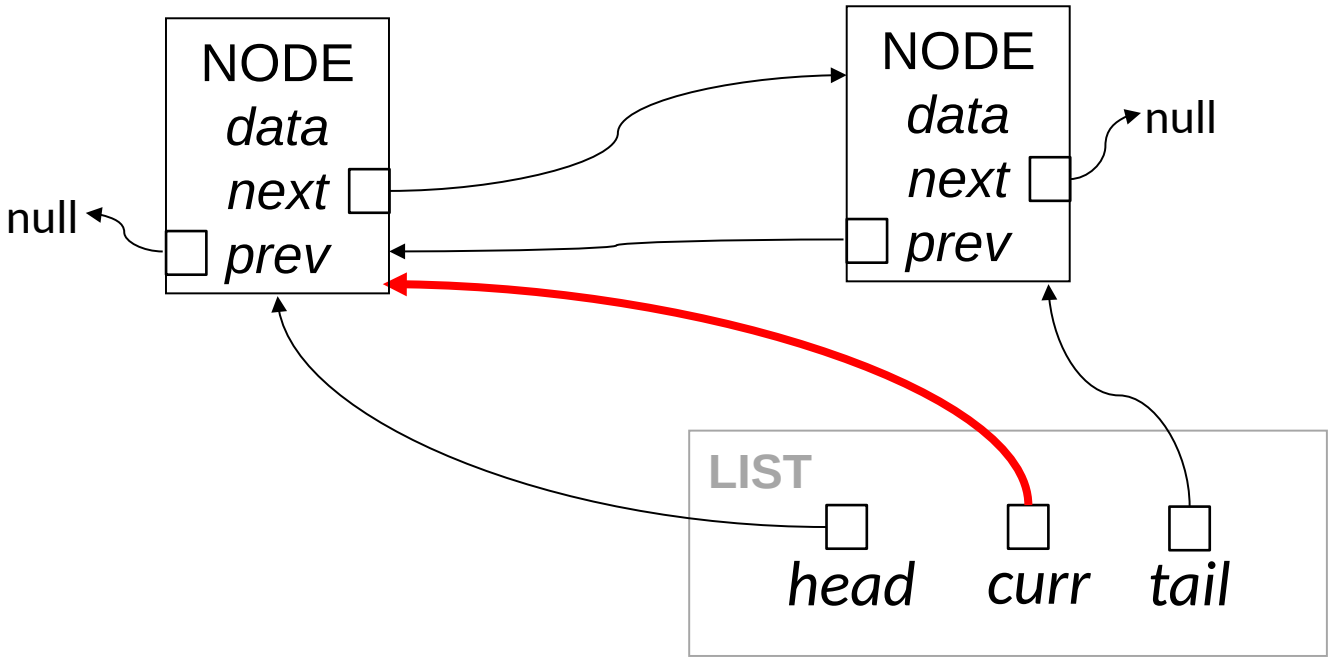


Move ‘current’ one position back in the list



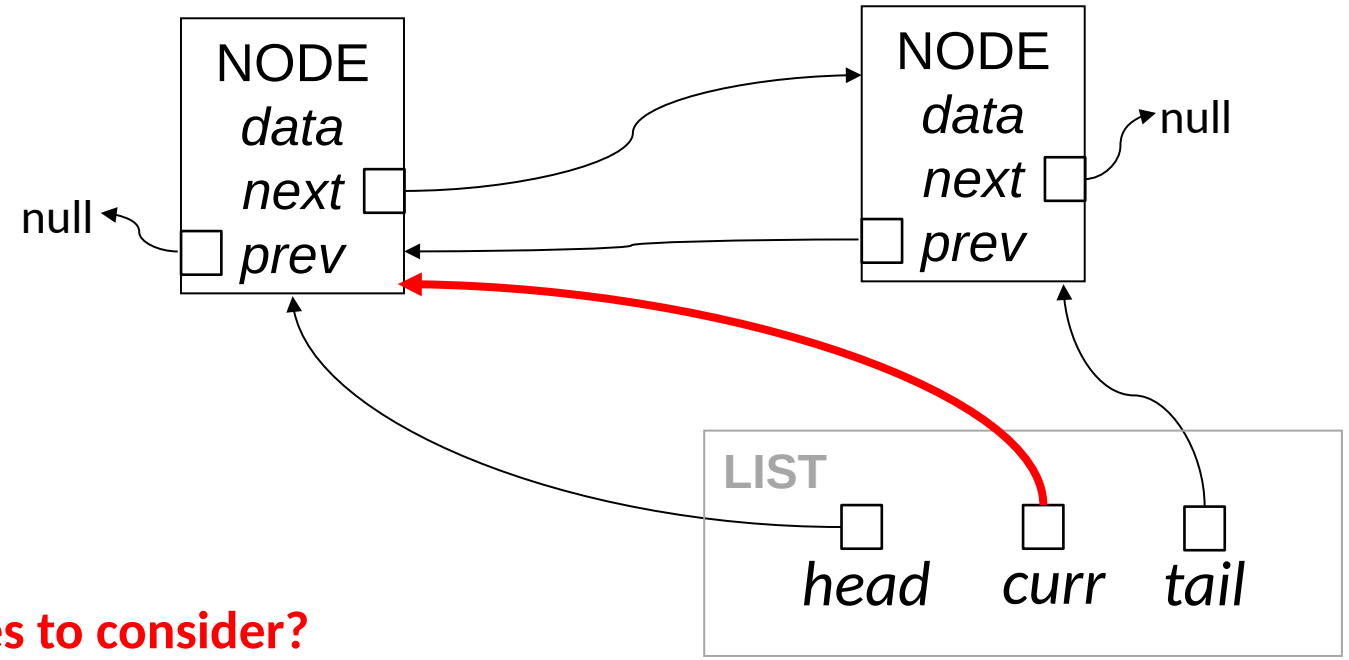
Shift curr to point to the node which is currently previous to it
E.g. list’s curr = list’s curr’s prev

Move ‘current’ one position back in the list



Shift curr to point to the node which is currently previous to it
E.g. list’s curr = list’s curr’s prev

Move ‘current’ one position back in the list



Special cases to consider?
What if the list is empty?
What if ‘curr’ is already at the ‘head’ / beginning of the list?

Move ‘current’ one position forward in the list

- Exercise for you

Get data at current position

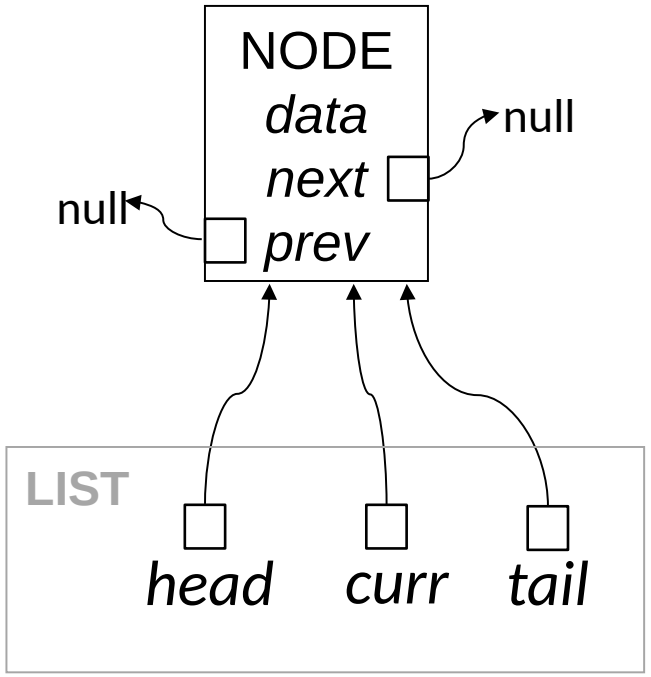
- An exercise for you
- See ‘peek’ methods in Stack example
- Remember to consider special cases

Remove node at current position

- Cases to consider
  - Empty list – nothing to do!
  - 1 item list
  - > 1 item list
    - Current node is head of list
    - Current node is tail of list
    - Current node is somewhere between head and tail

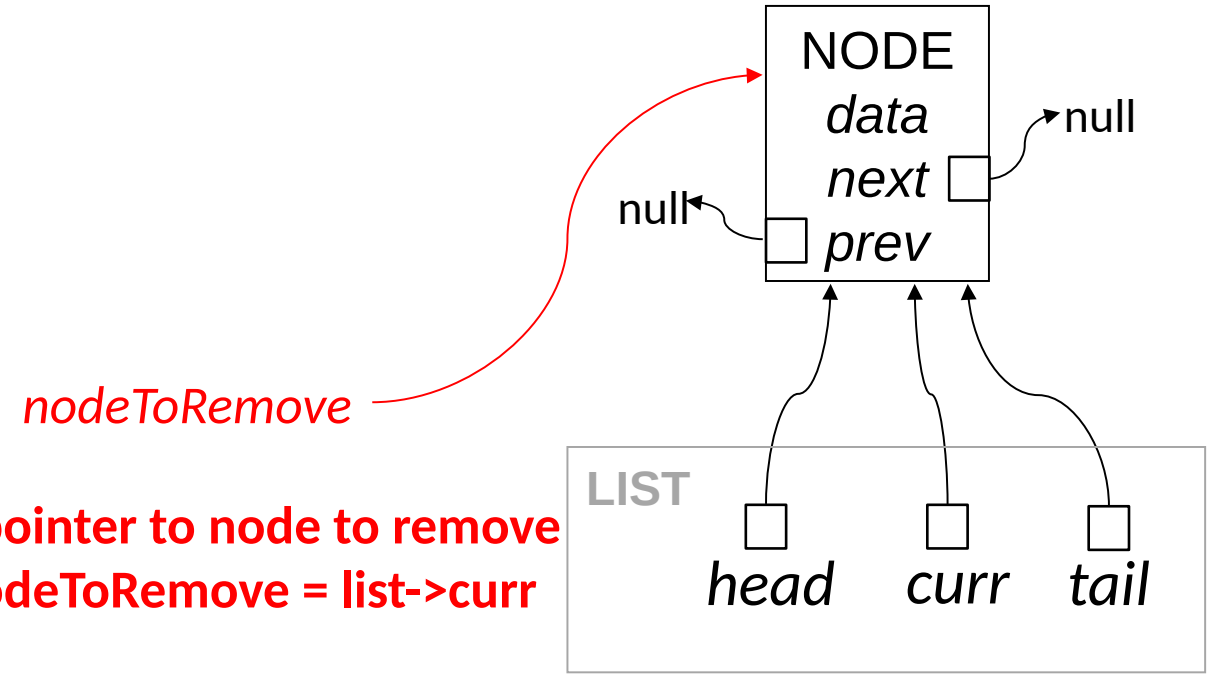
Remove node at current position

- 1 item list : head == tail == current



Remove node at current position

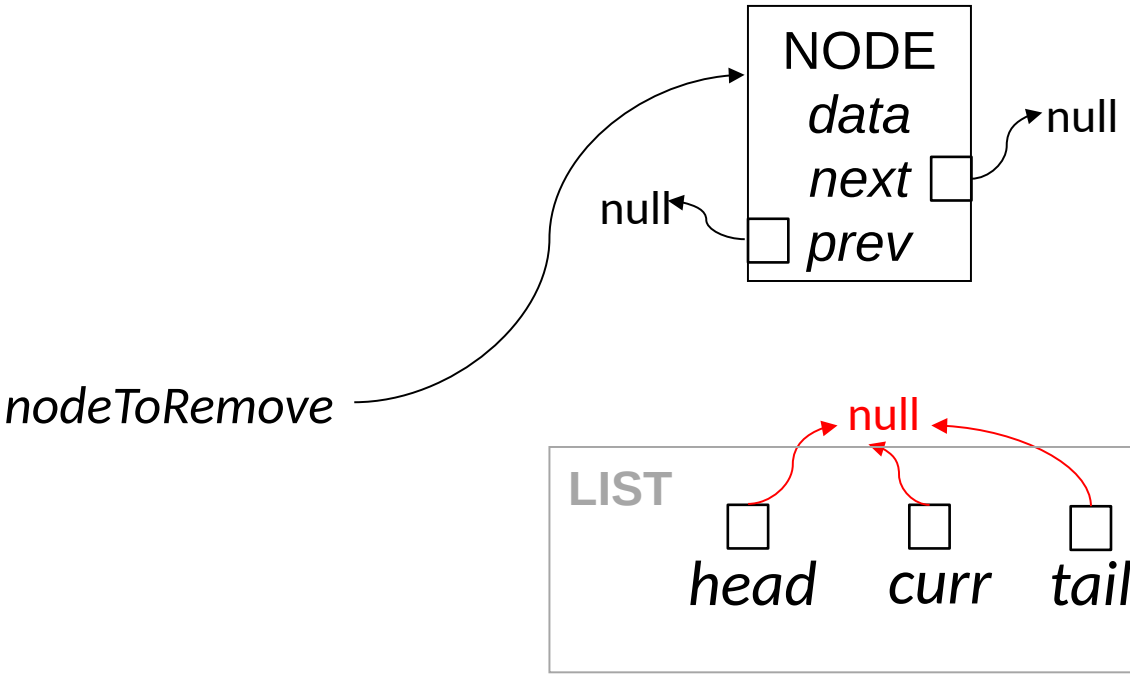
- 1 item list : head == tail == current



Set a pointer to node to remove
E.g. nodeToRemove = list->curr

Remove node at current position

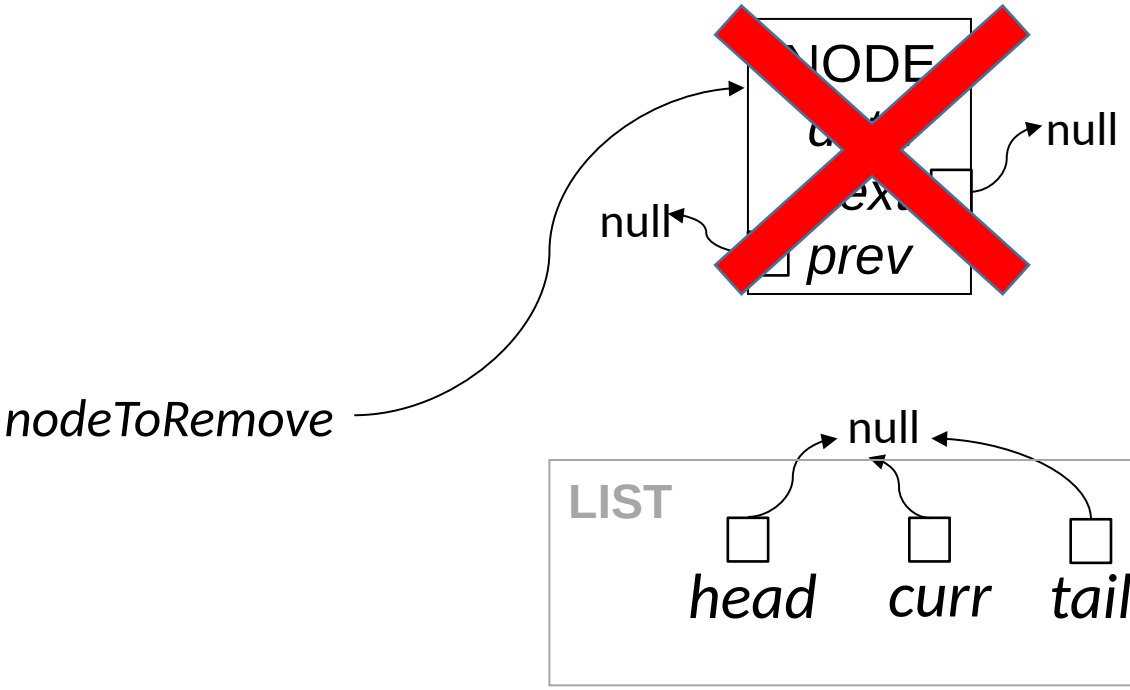
- 1 item list : head == tail == current



Set list to be empty
head = tail = curr = NULL

Remove node at current position

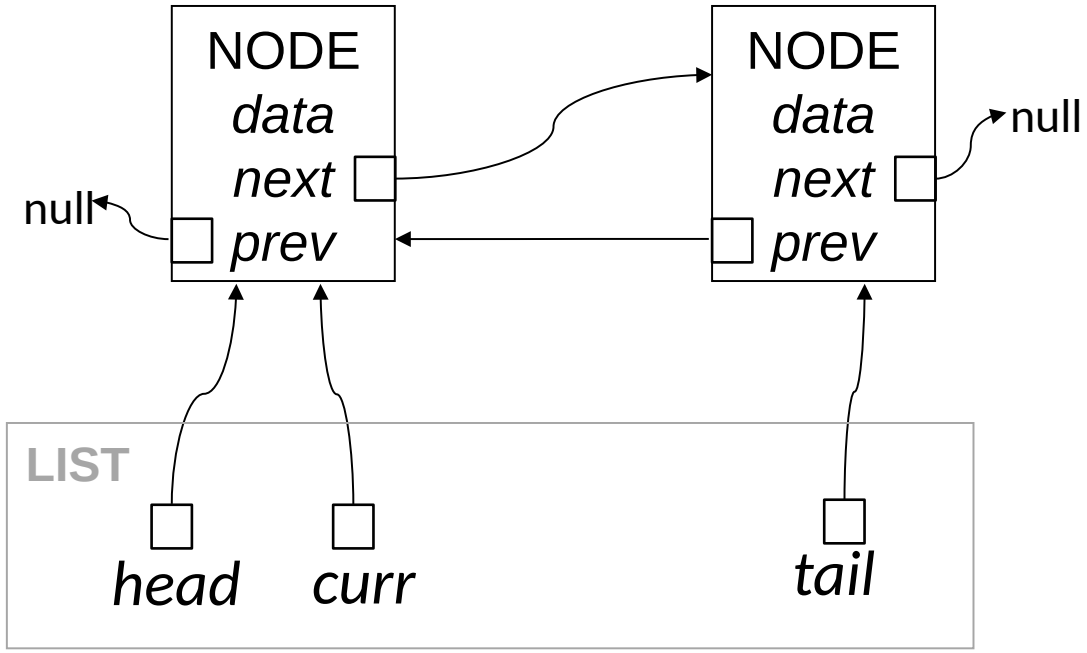
- 1 item list : head == tail == current



free node being removed
E.g. free(nodeToRemove)

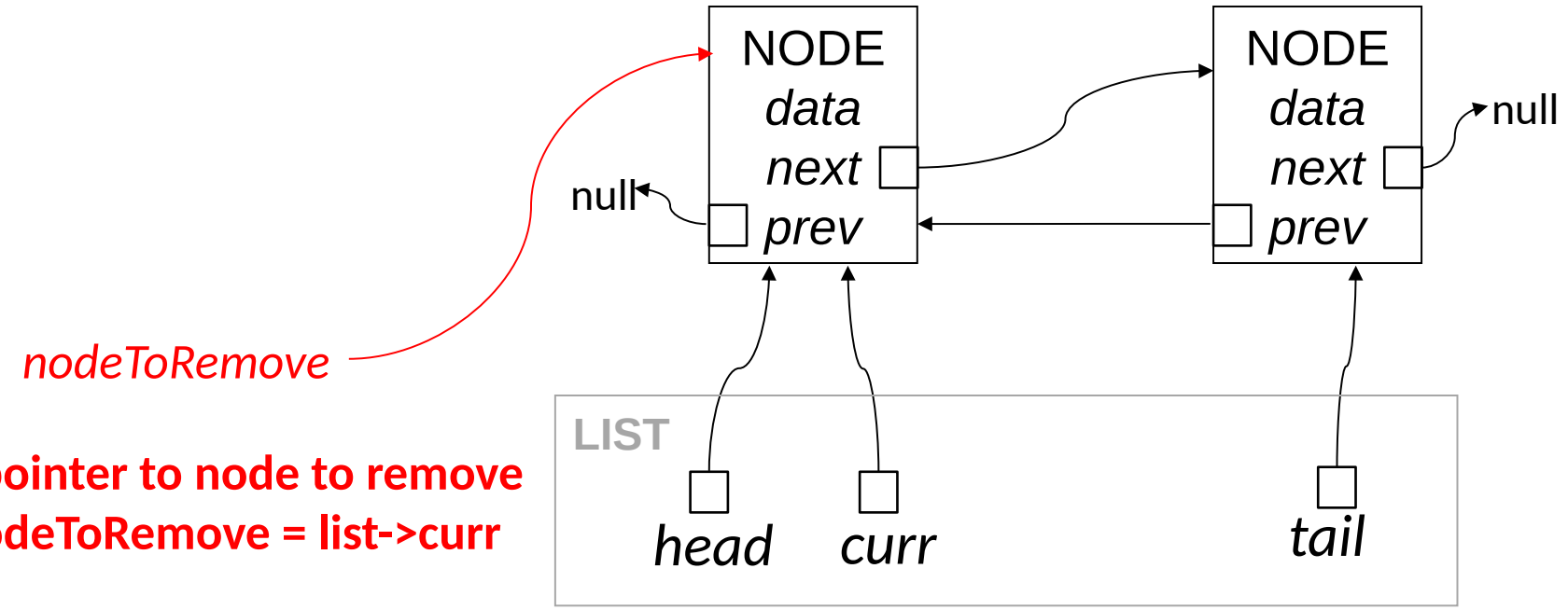
Remove node at current position

- More than 1 item in list, current node is at head of list
  - list->curr == list->head



Remove node at current position

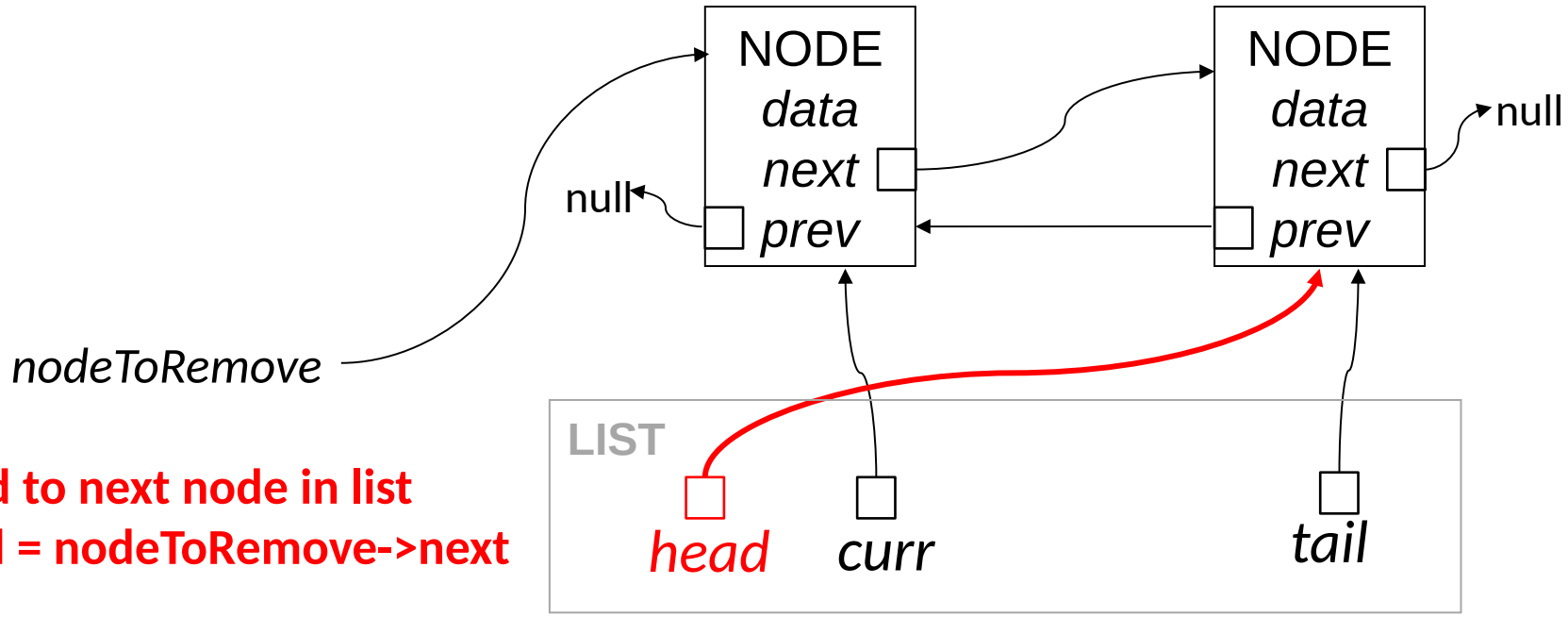
- More than 1 item in list, current node is at head of list



Set a pointer to node to remove
E.g. nodeToRemove = list->curr

Remove node at current position

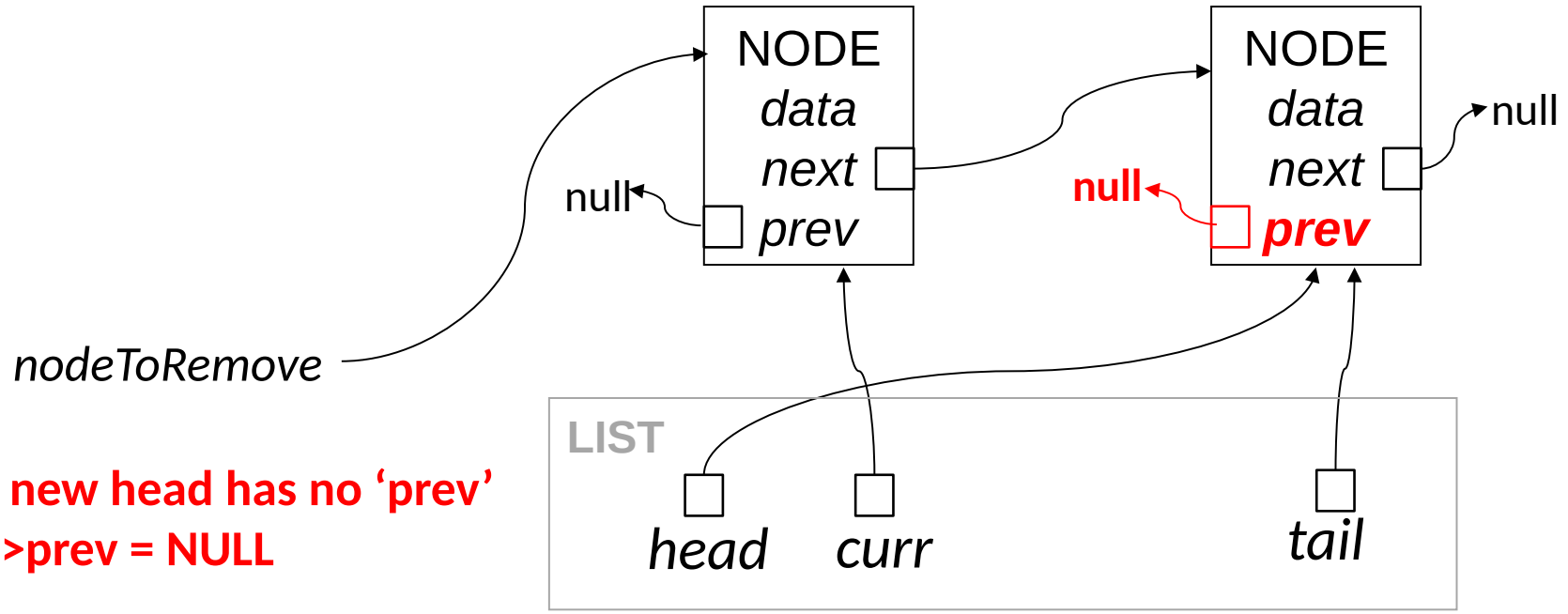
- More than 1 item in list, current node is at head of list



Shift head to next node in list
list->head = nodeToRemove->next

Remove node at current position

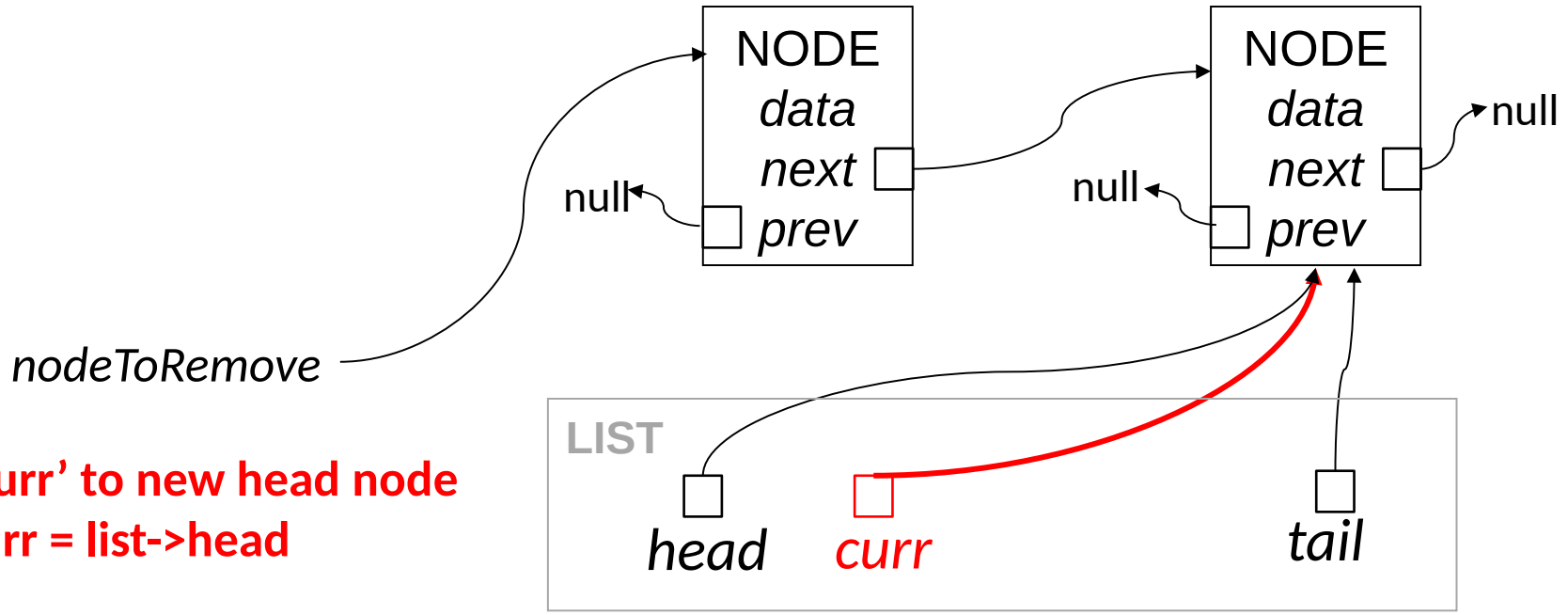
- More than 1 item in list, current node is at head of list



Make sure new head has no 'prev'
list->head->prev = NULL

Remove node at current position

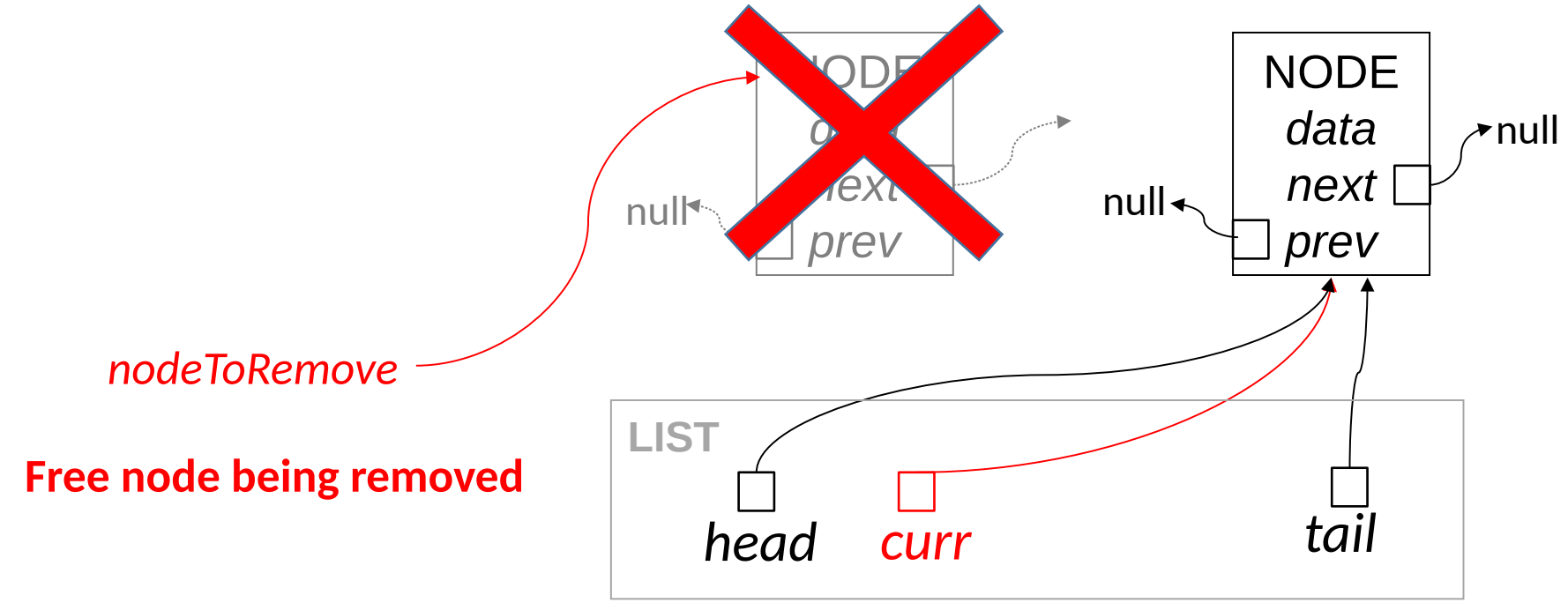
- More than 1 item in list, current node is at head of list



Shift 'curr' to new head node
list->curr = list->head

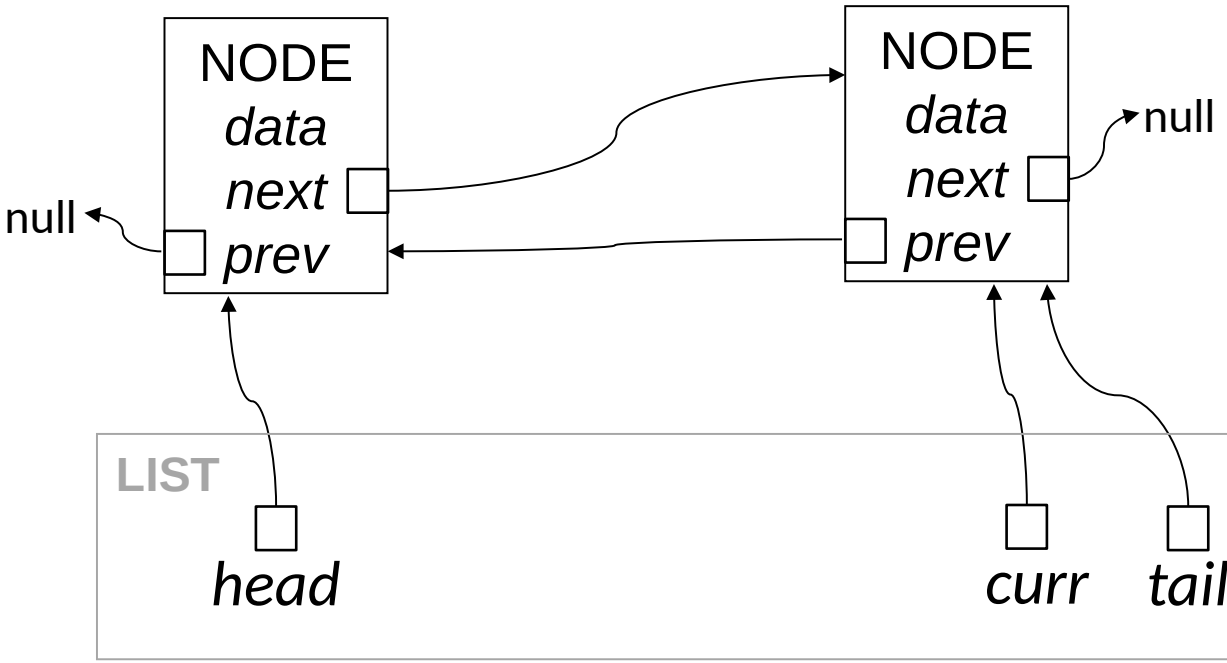
Remove node at current position

- More than 1 item in list, current node is at head of list



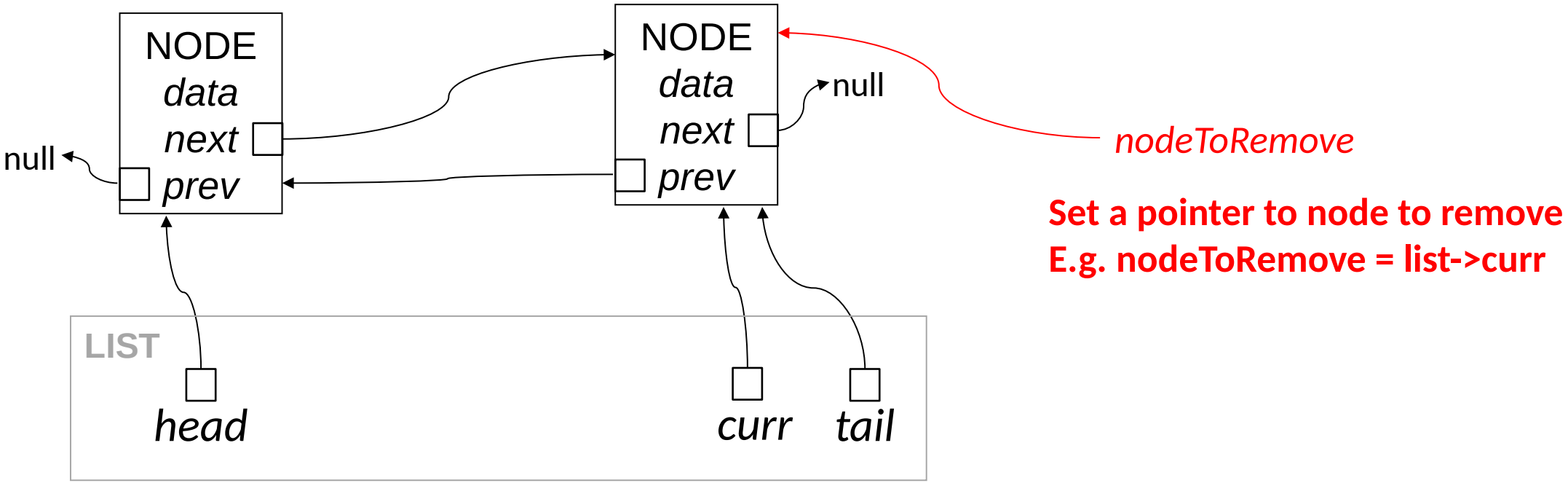
Remove node at current position

- More than 1 item in list, current node is at TAIL of list
  - list->curr == list->tail



Remove node at current position

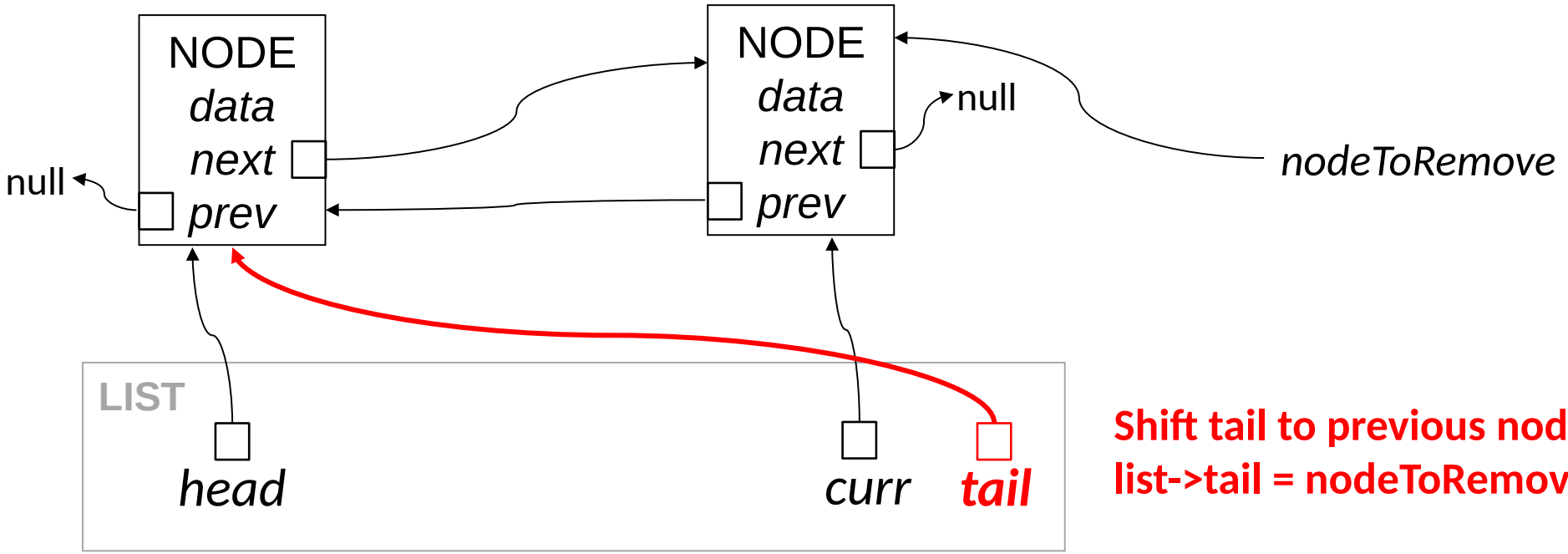
- More than 1 item in list, current node is at TAIL of list



Set a pointer to node to remove
E.g. nodeToRemove = list->curr

Remove node at current position

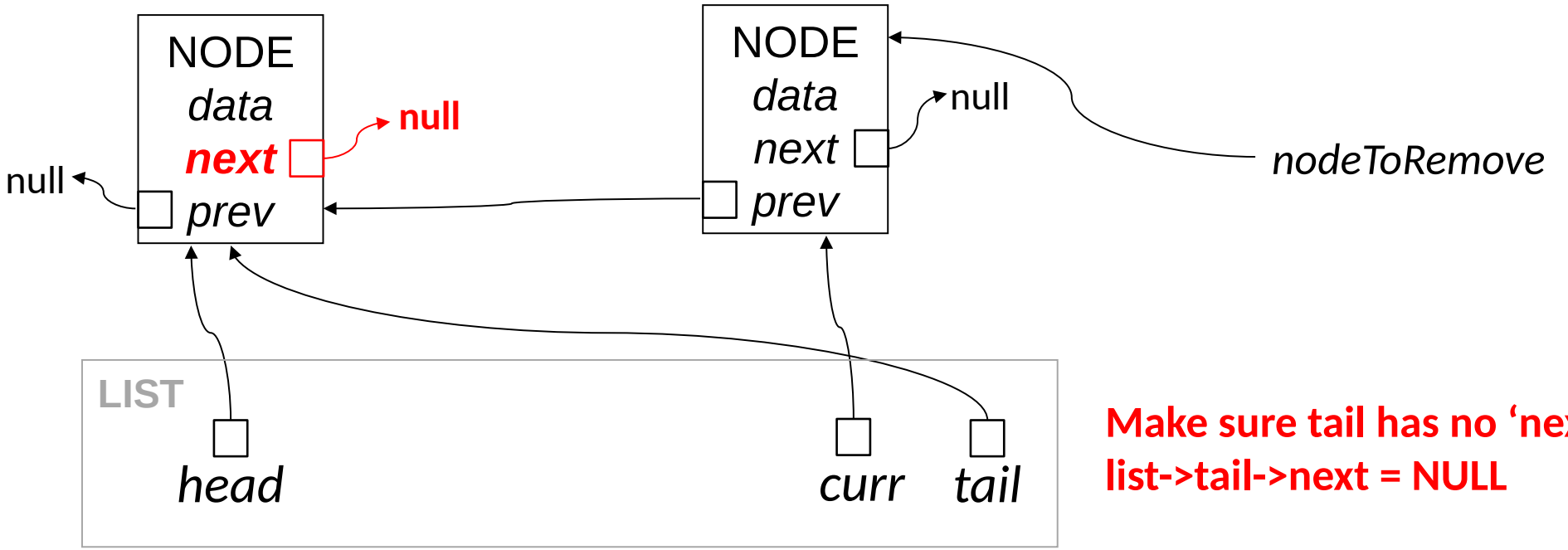
- More than 1 item in list, current node is at TAIL of list



Shift tail to previous node in list
list->tail = nodeToRemove->prev

Remove node at current position

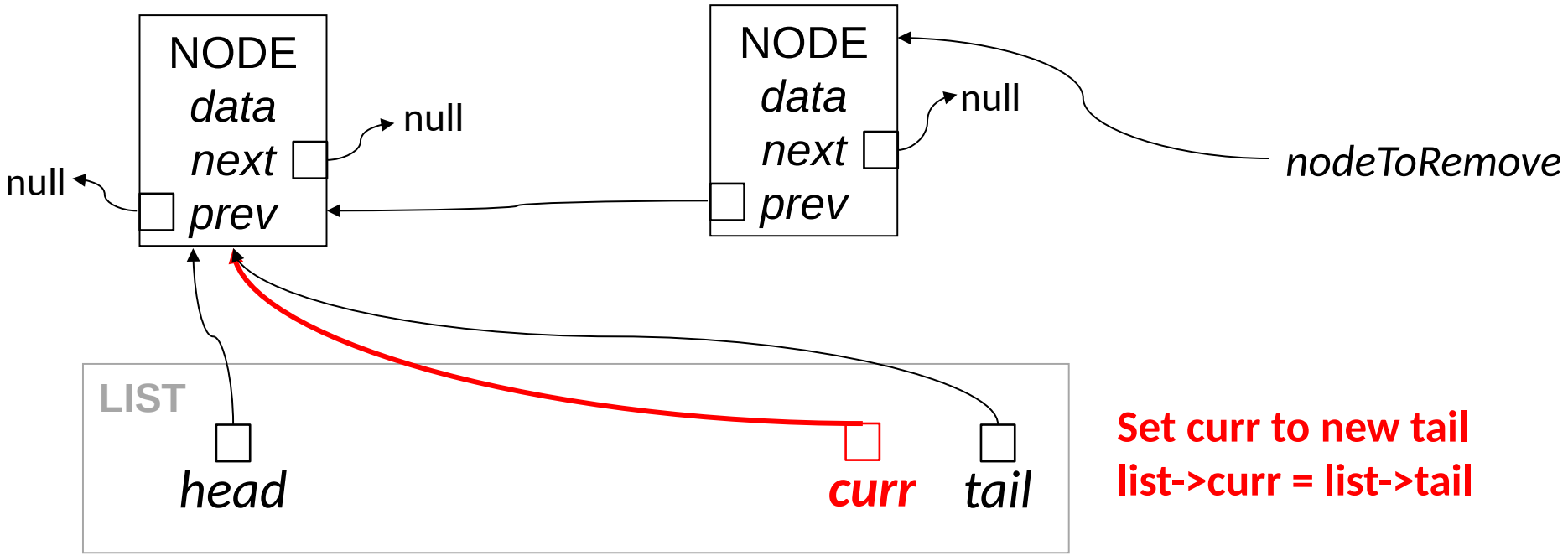
- More than 1 item in list, current node is at TAIL of list



Make sure tail has no 'next' node
list->tail->next = NULL

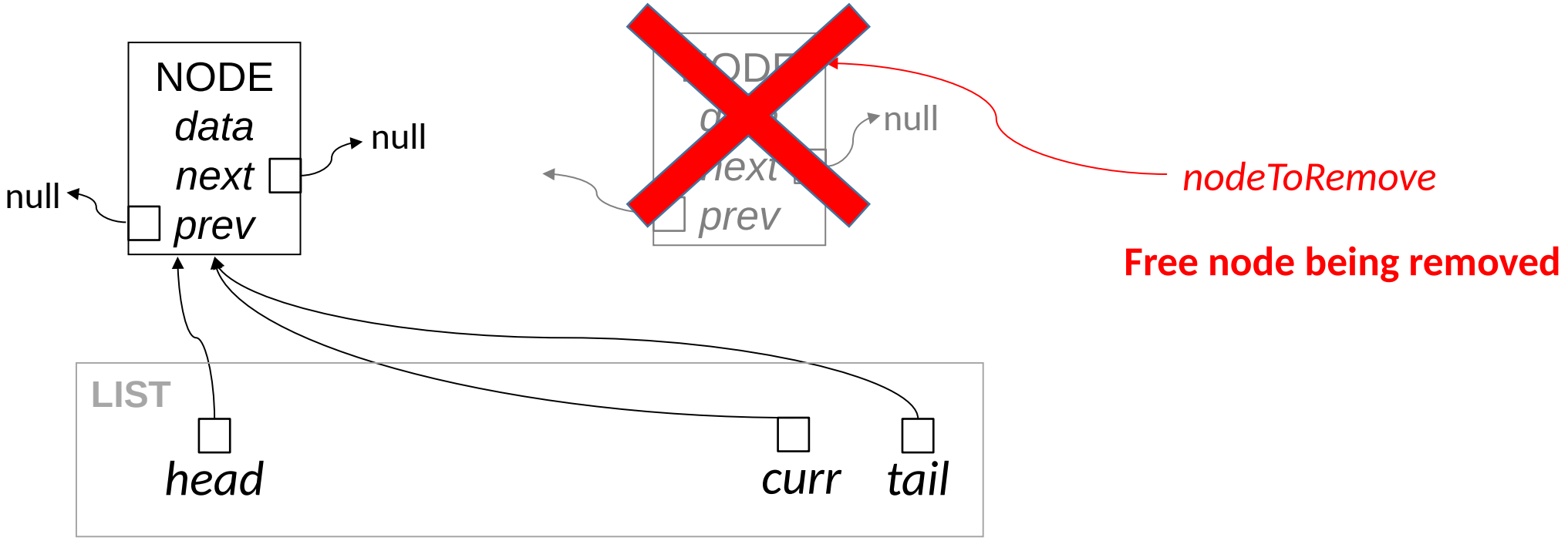
Remove node at current position

- More than 1 item in list, current node is at TAIL of list



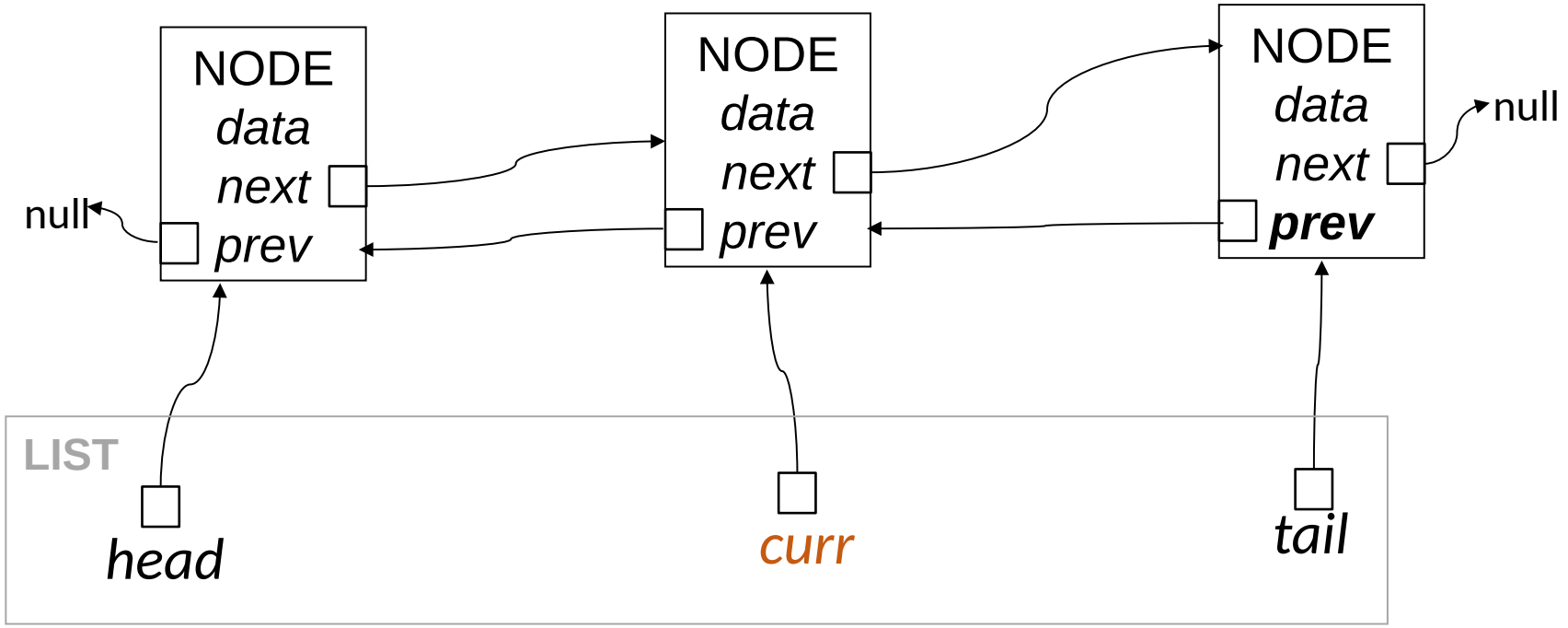
Remove node at current position

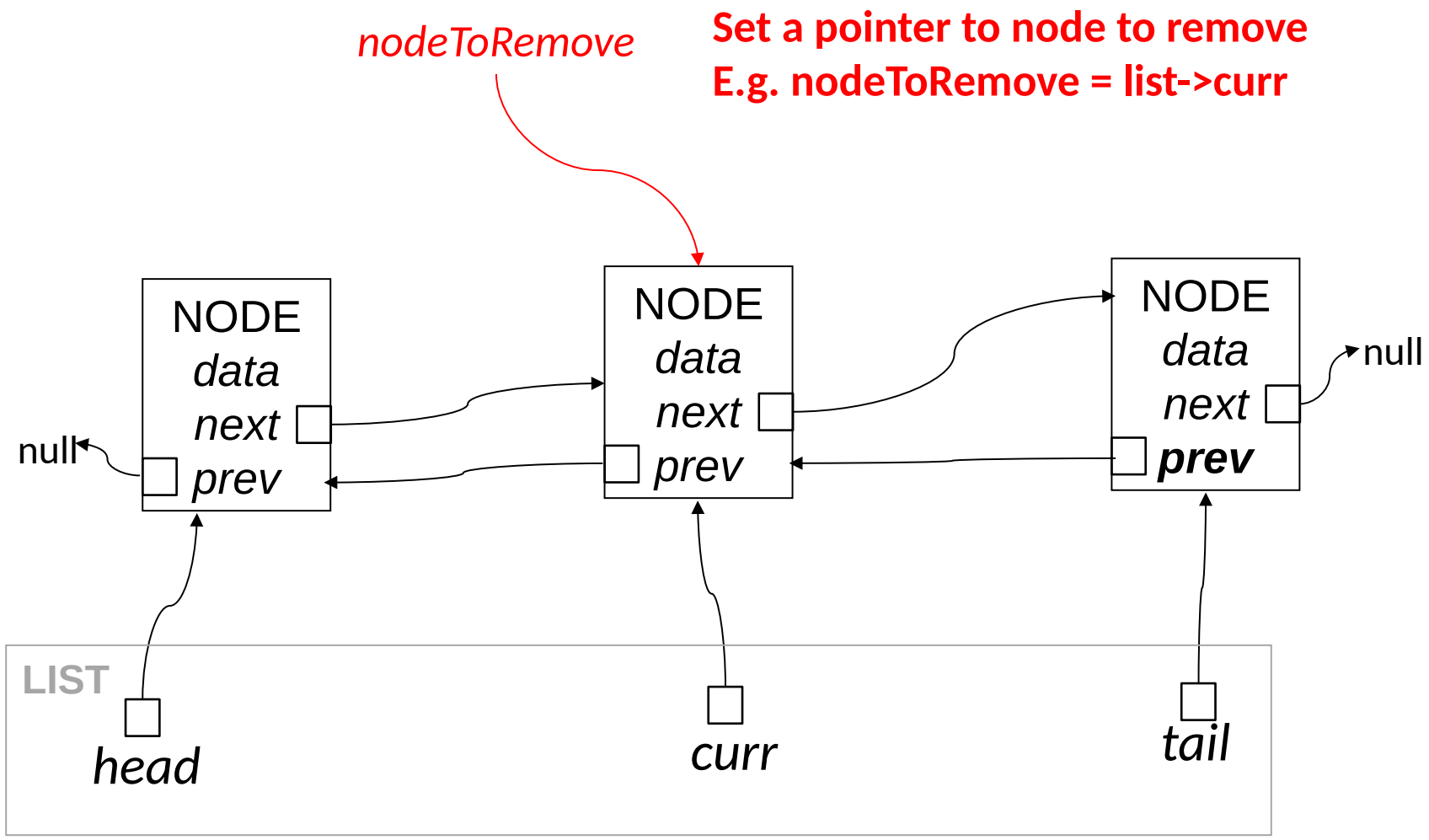
- More than 1 item in list, current node is at TAIL of list

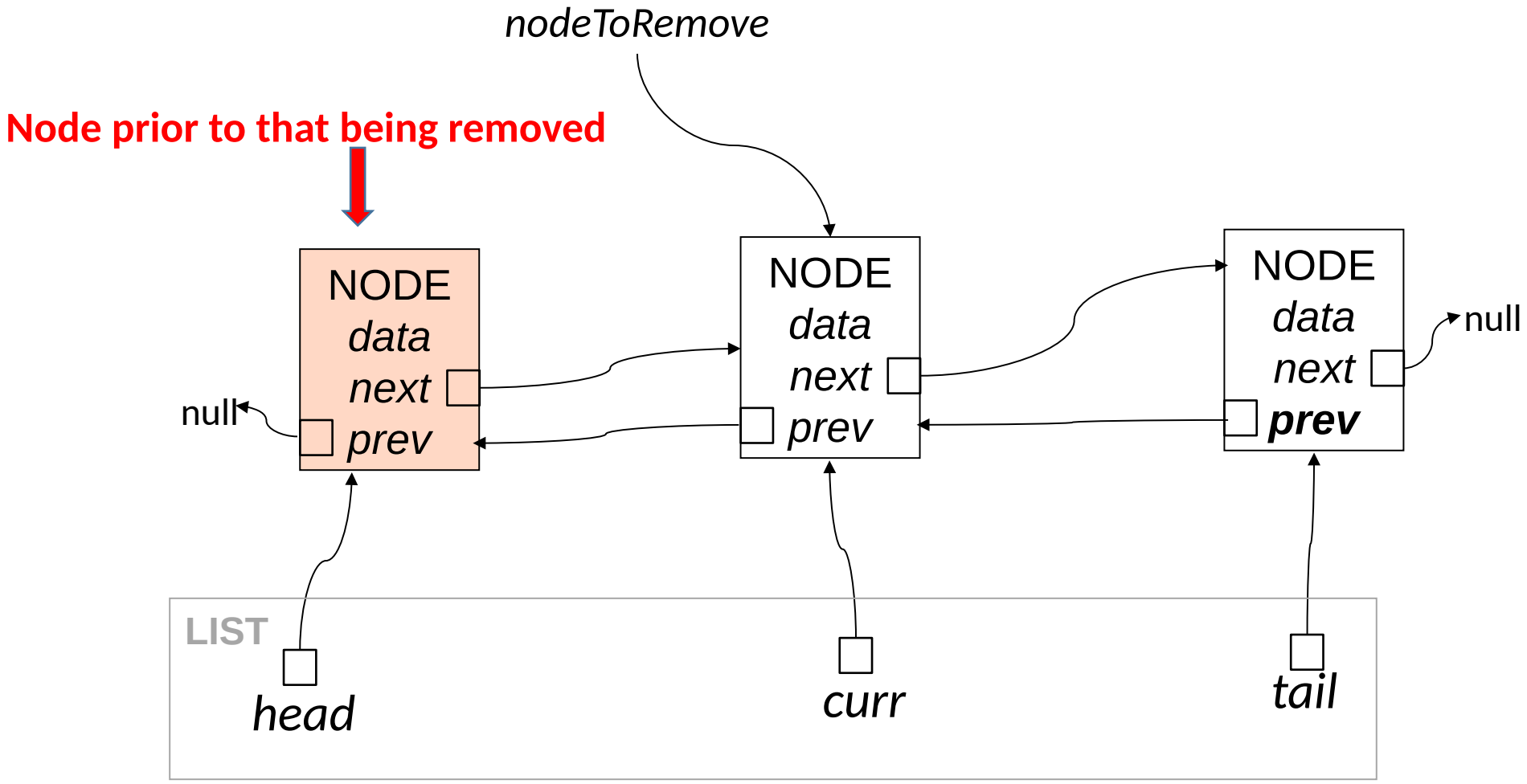


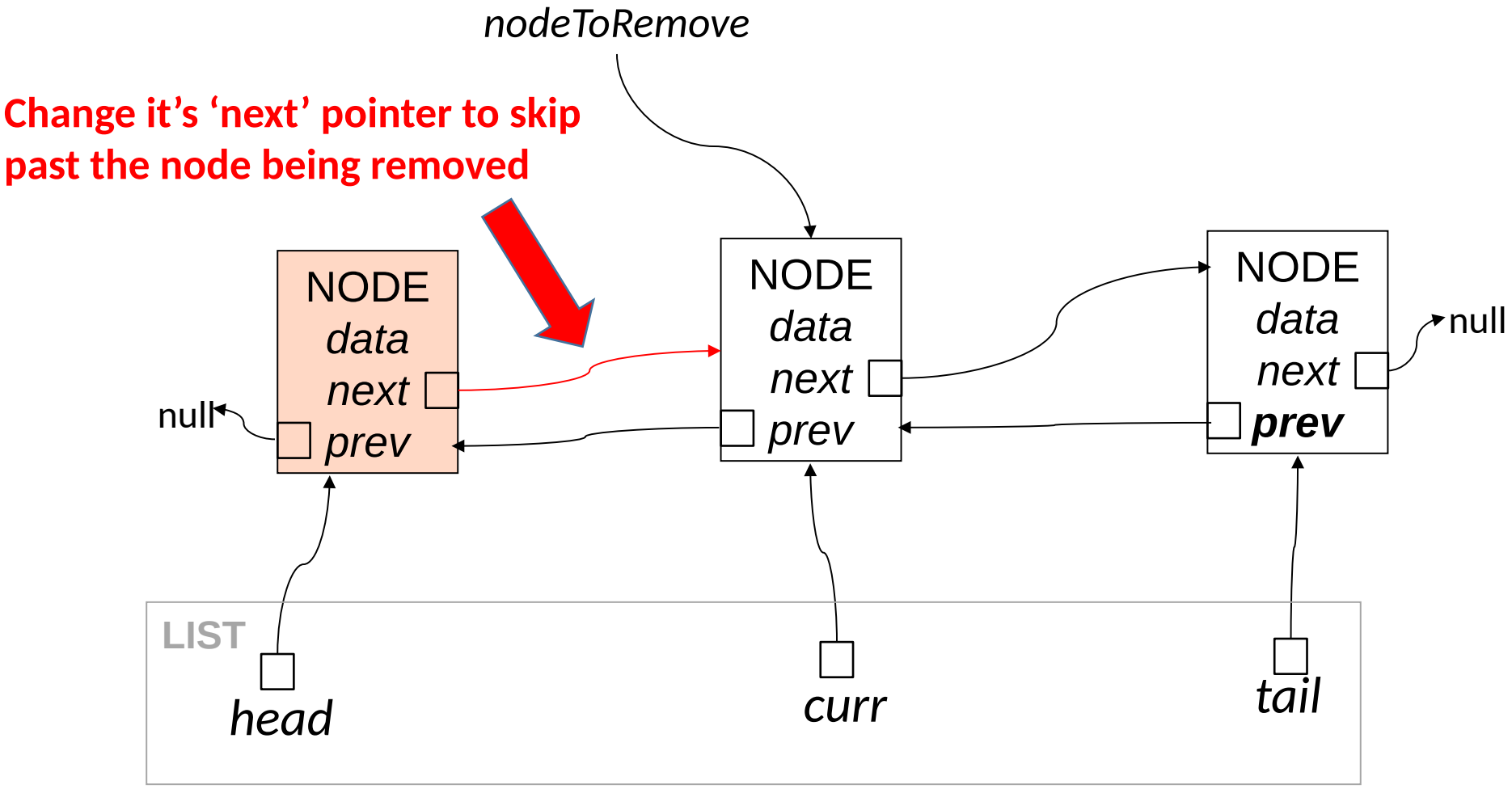
Remove node at current position

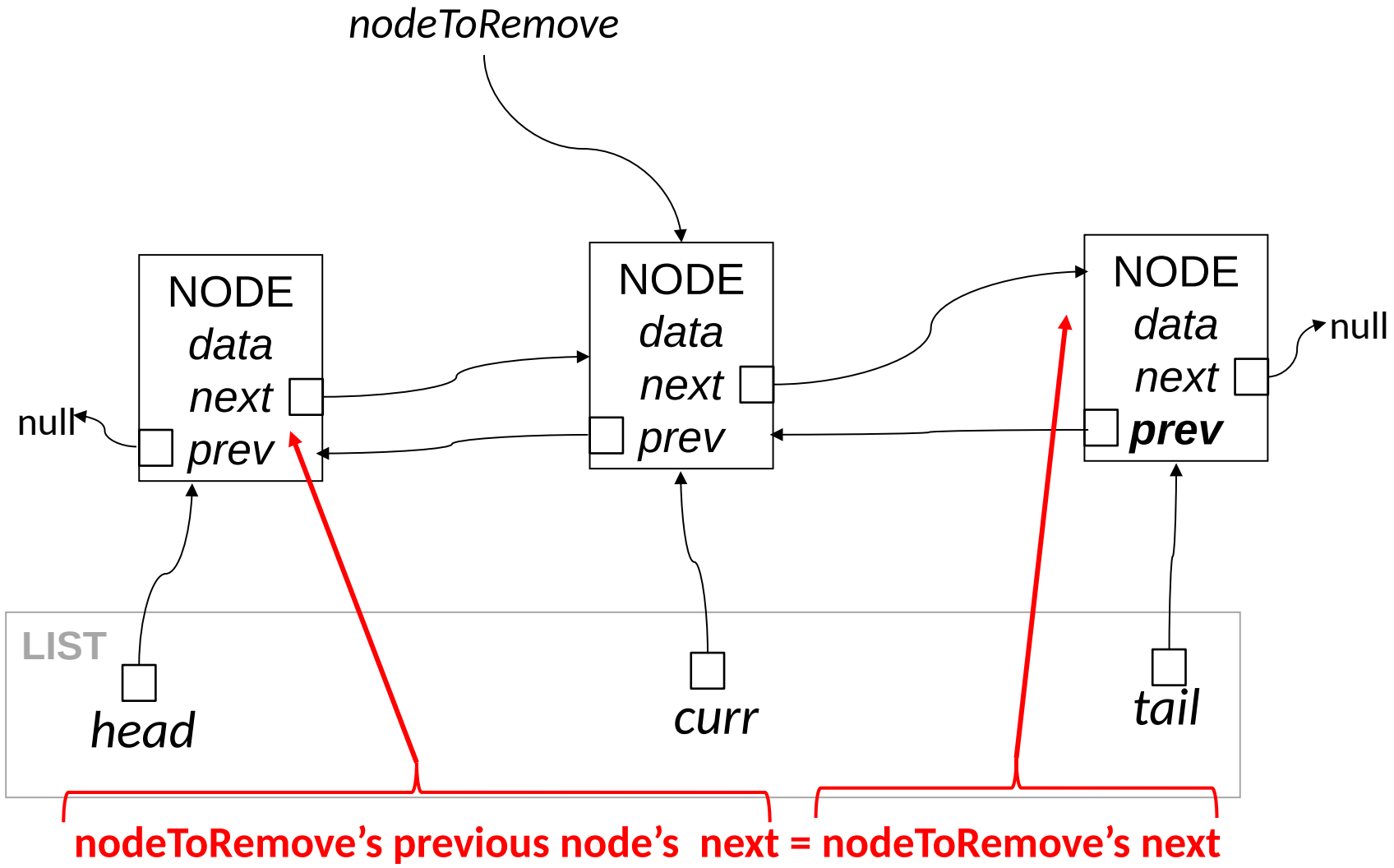
- More than 1 item in list, current node is somewhere in between head and tail
  - list->curr != list->head and list->curr != list->tail

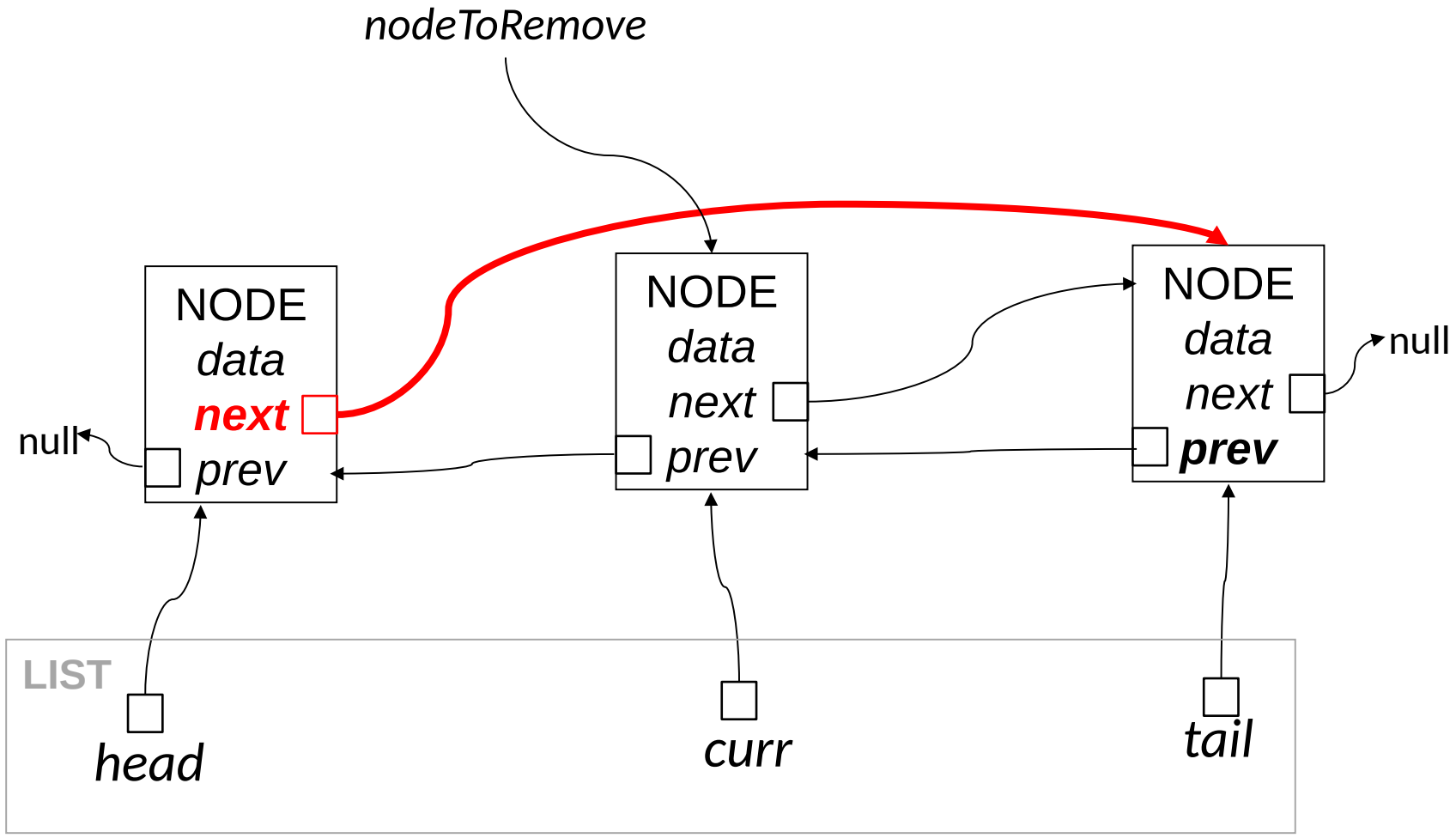



nodeToRemove
Set a pointer to node to remove
E.g. nodeToRemove = list->curr
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

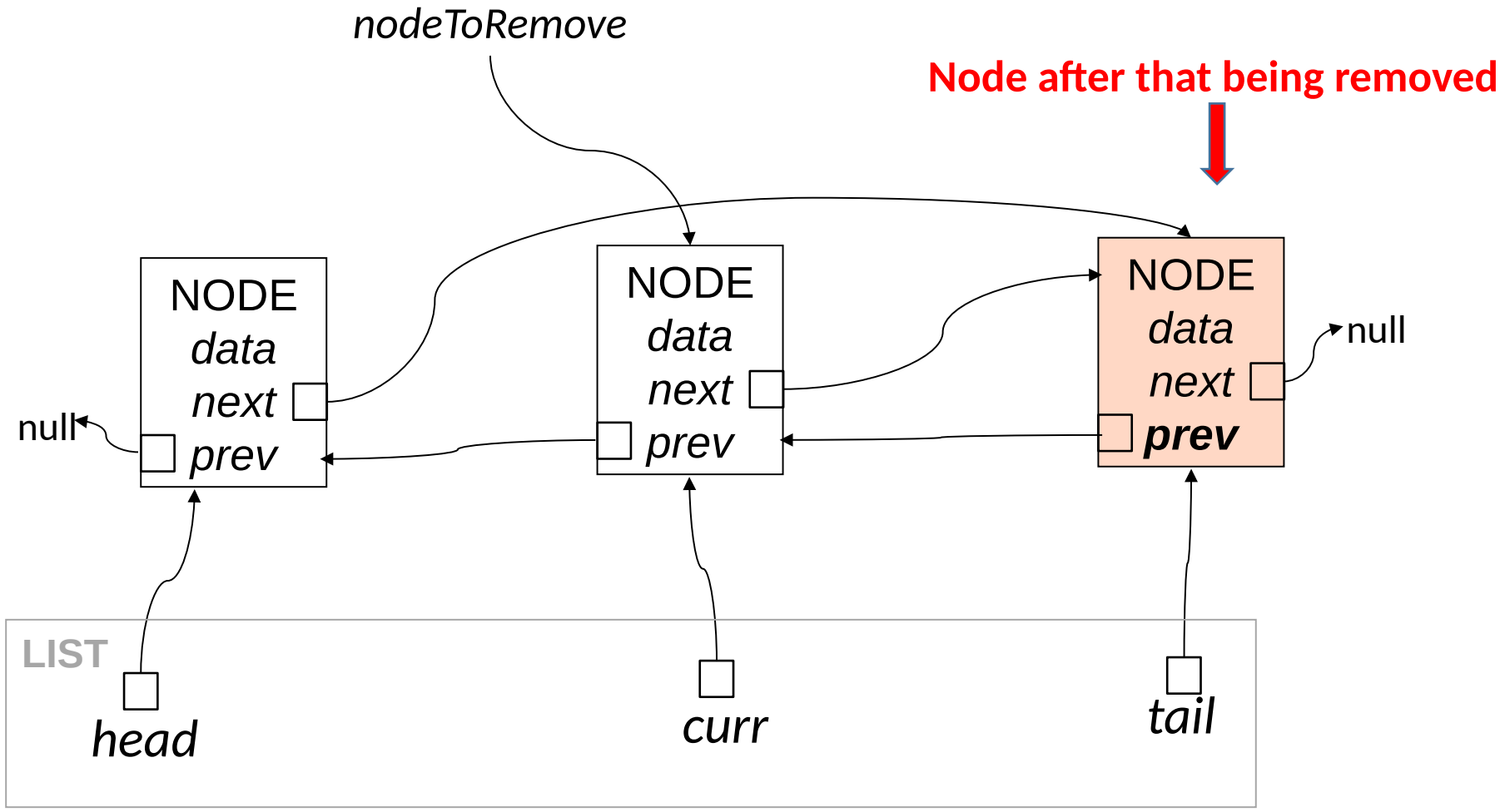

nodeToRemove
Node prior to that being removed
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

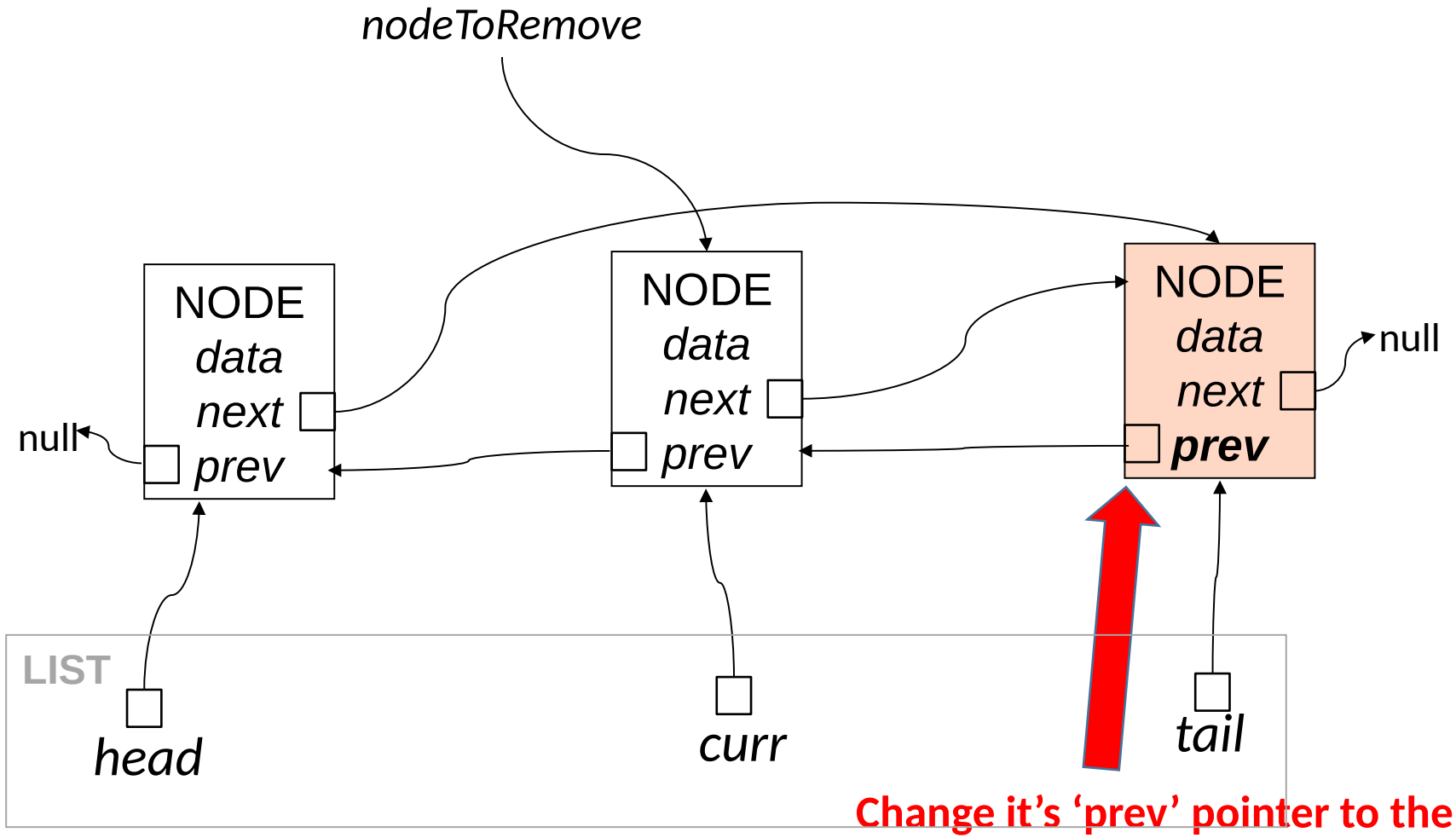

nodeToRemove
Change it's 'next' pointer to skip past the node being removed
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

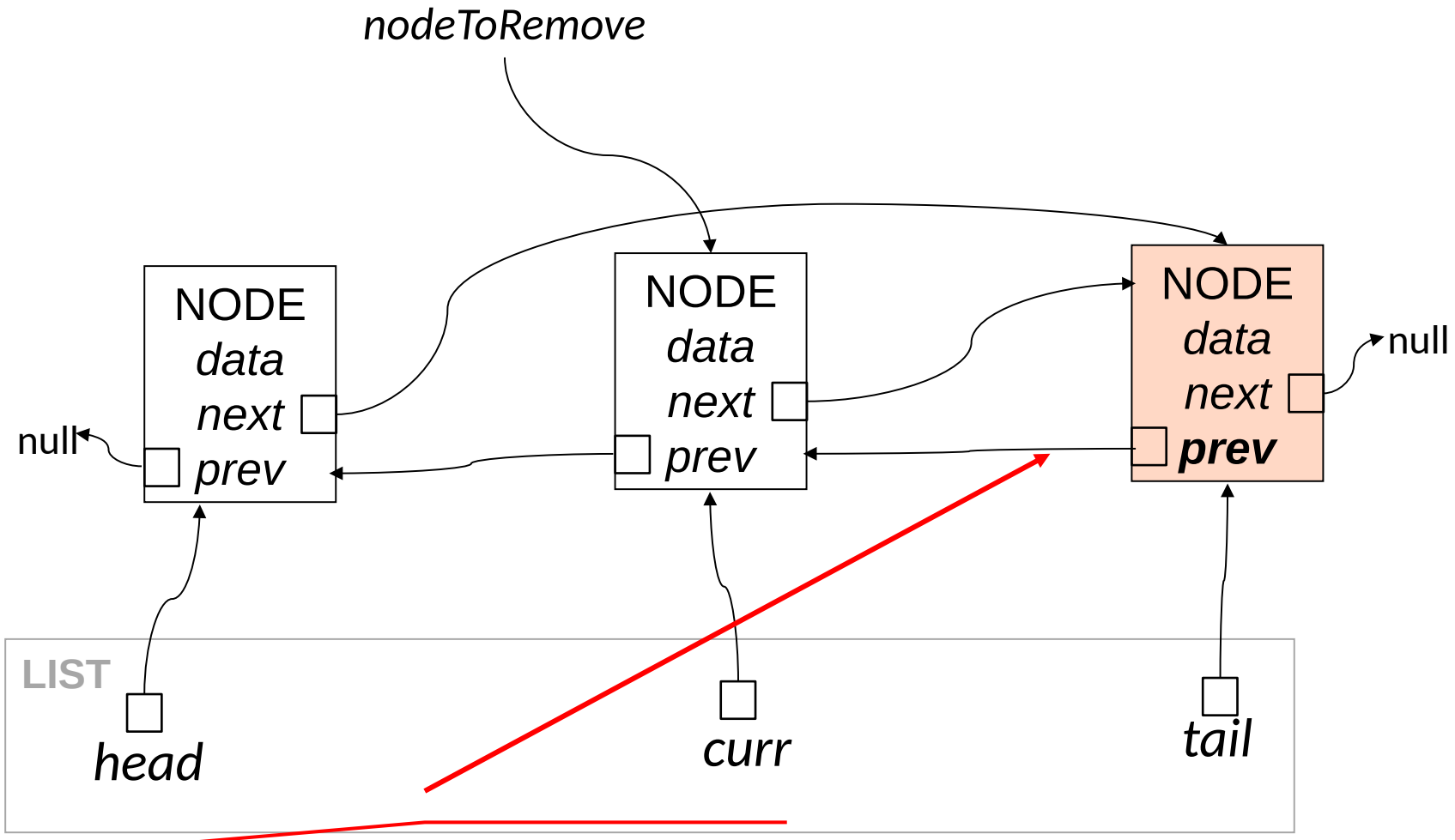

nodeToRemove
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail
nodeToRemove's previous node's next = nodeToRemove's next



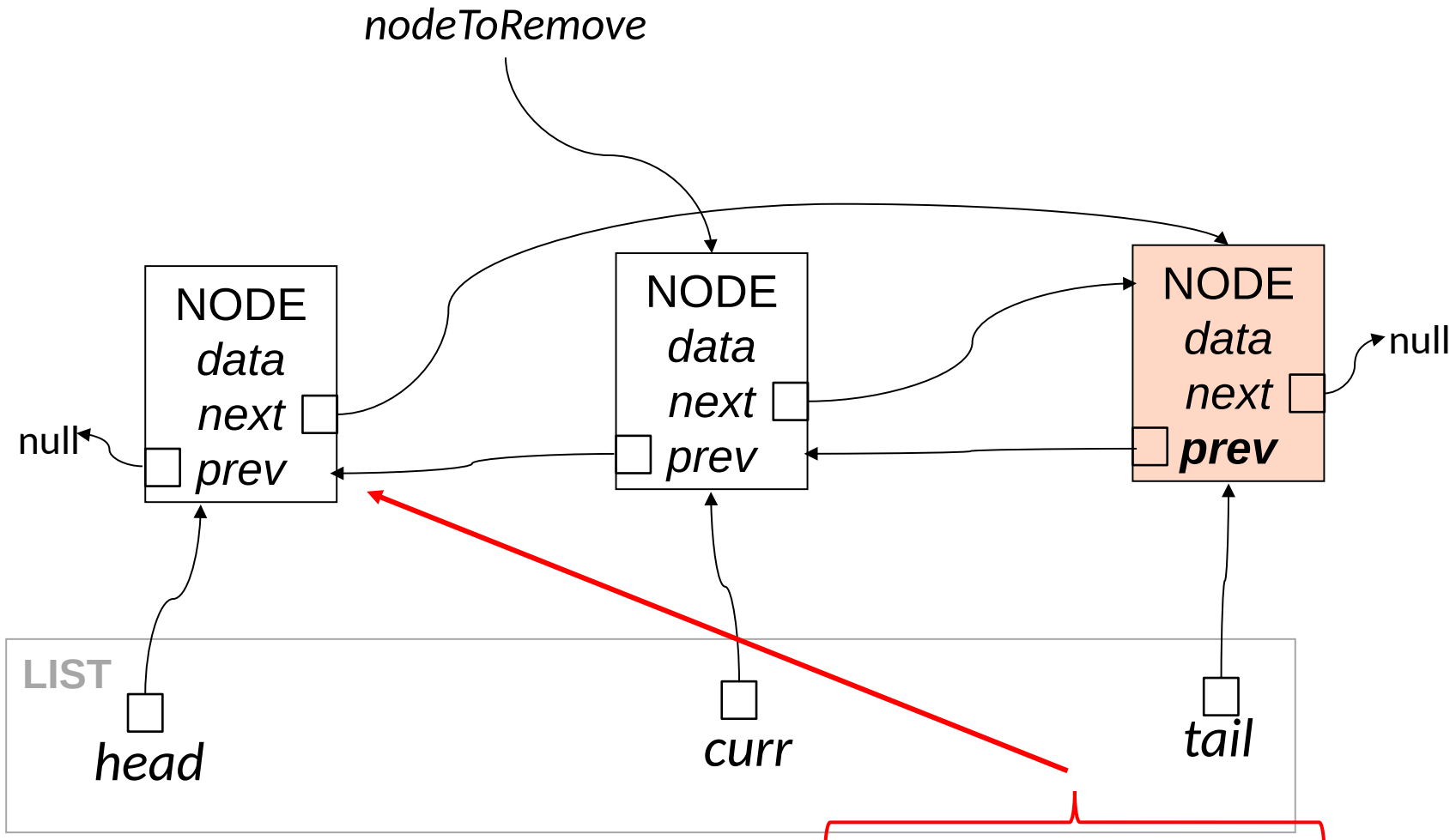
nodeToRemove's previous node's next = nodeToRemove's next


nodeToRemove
Node after that being removed
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

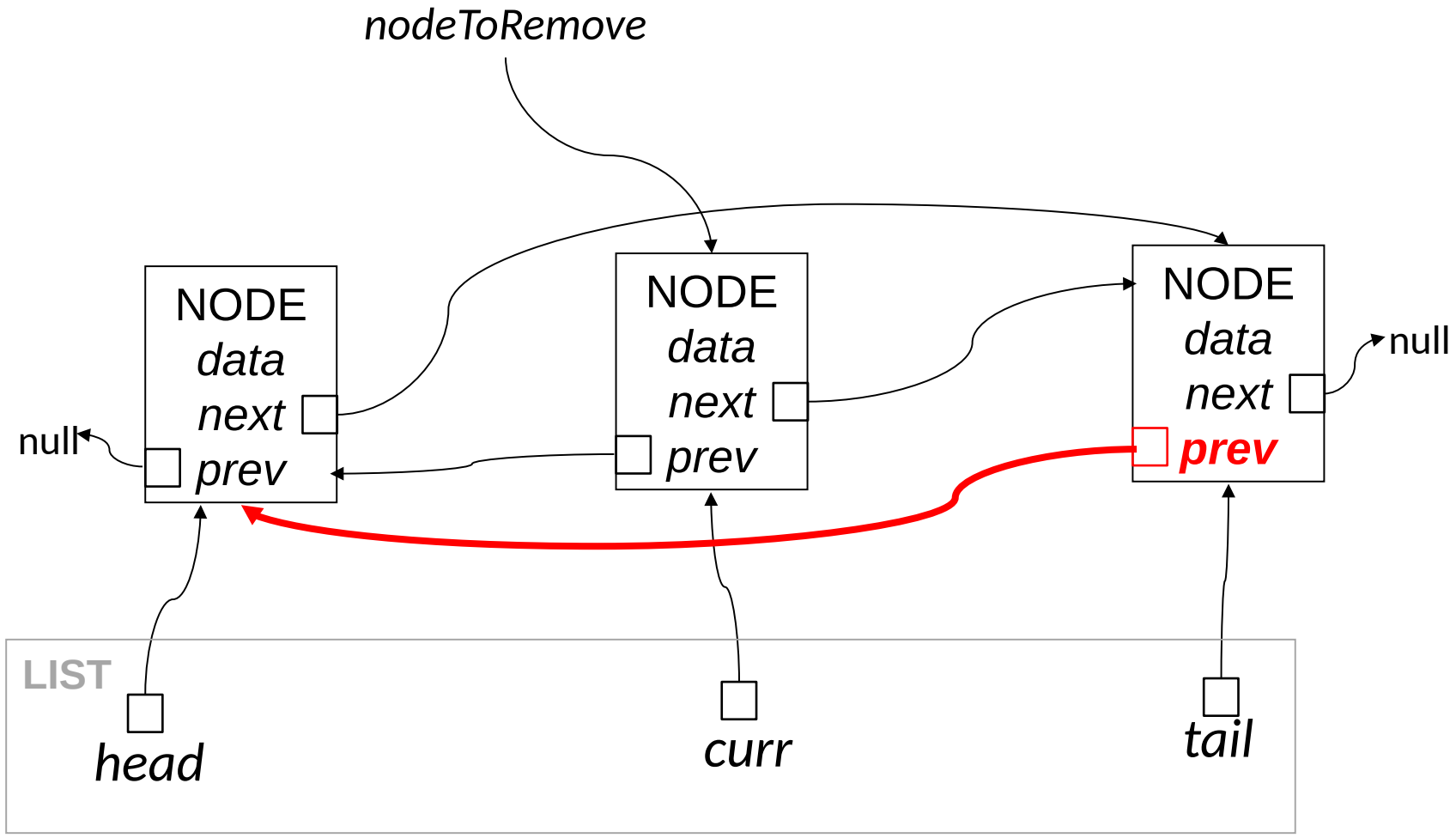

nodeToRemove
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail
Change it's 'prev' pointer to the
node behind the node being removed

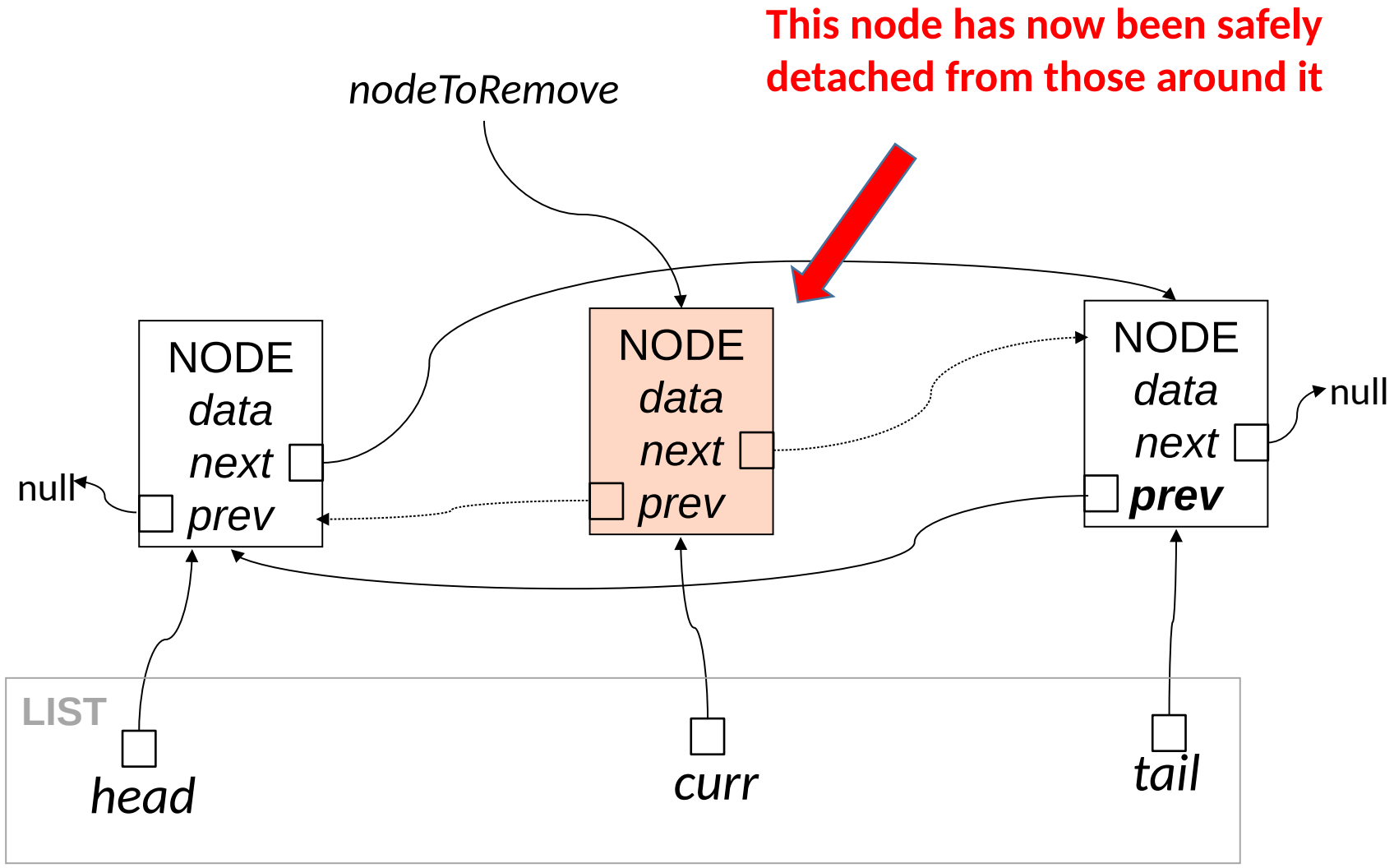


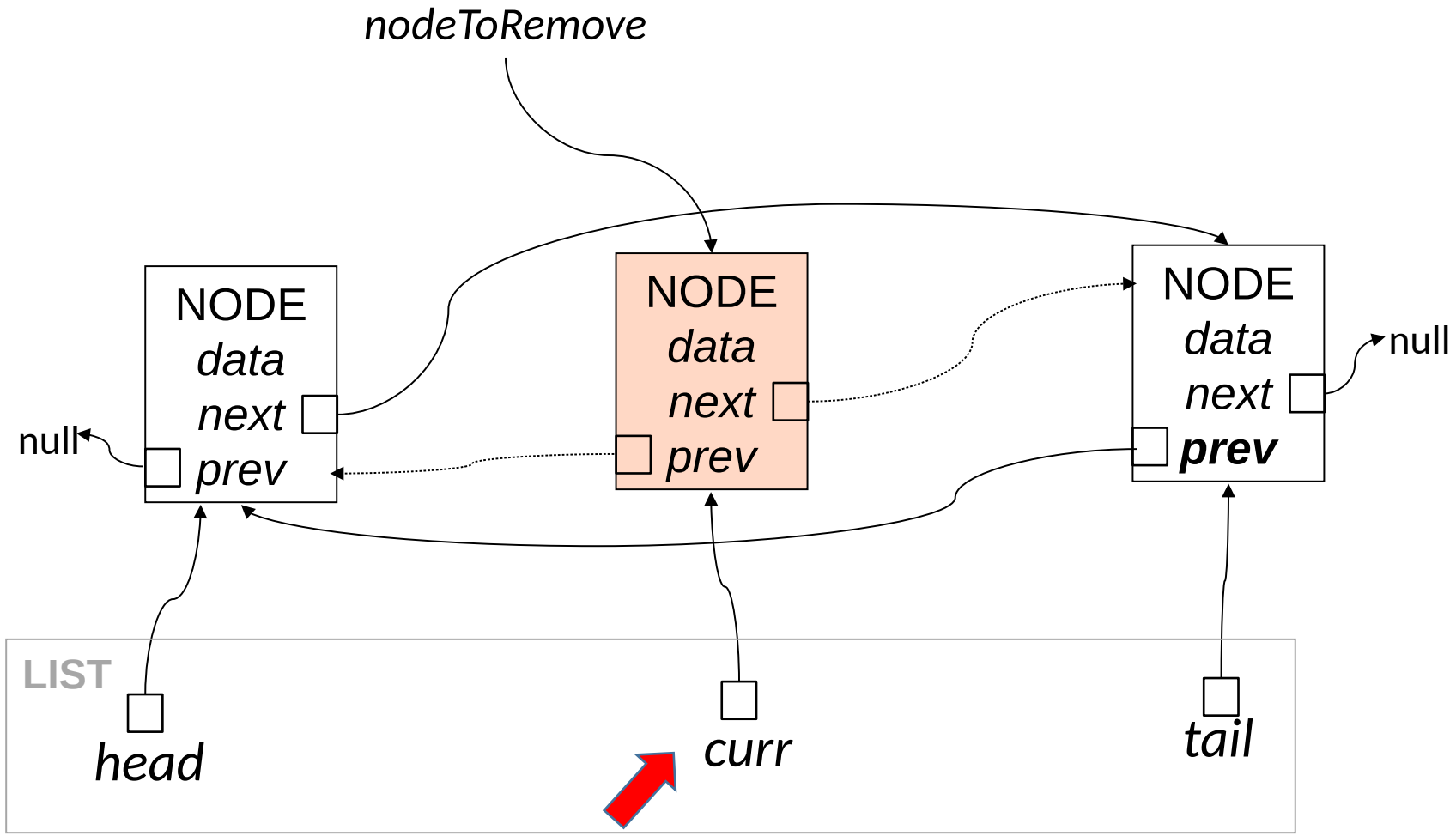
nodeToRemove's next node's previous = nodeToRemove's previous



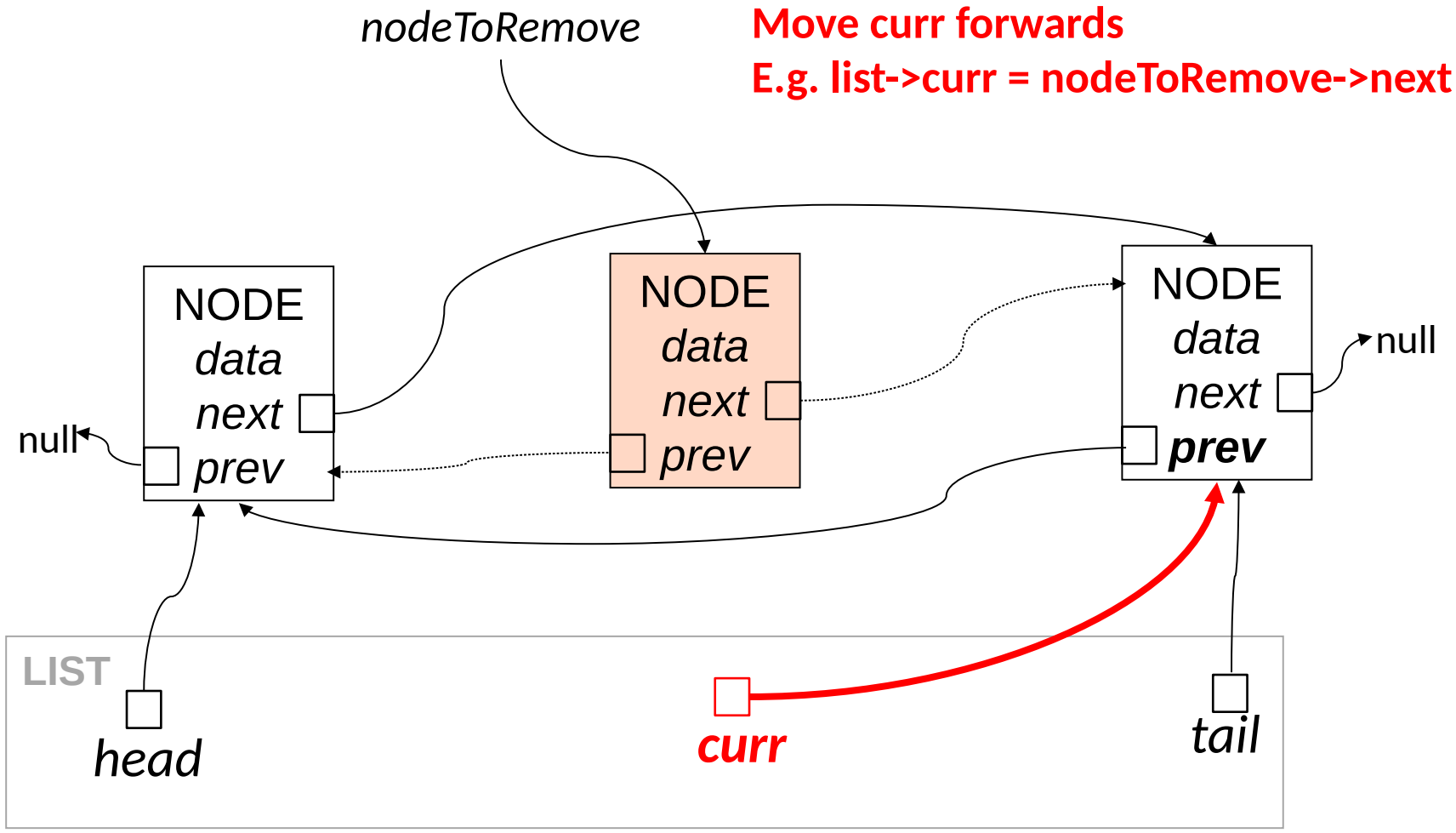
nodeToRemove's next node's previous = nodeToRemove's previous

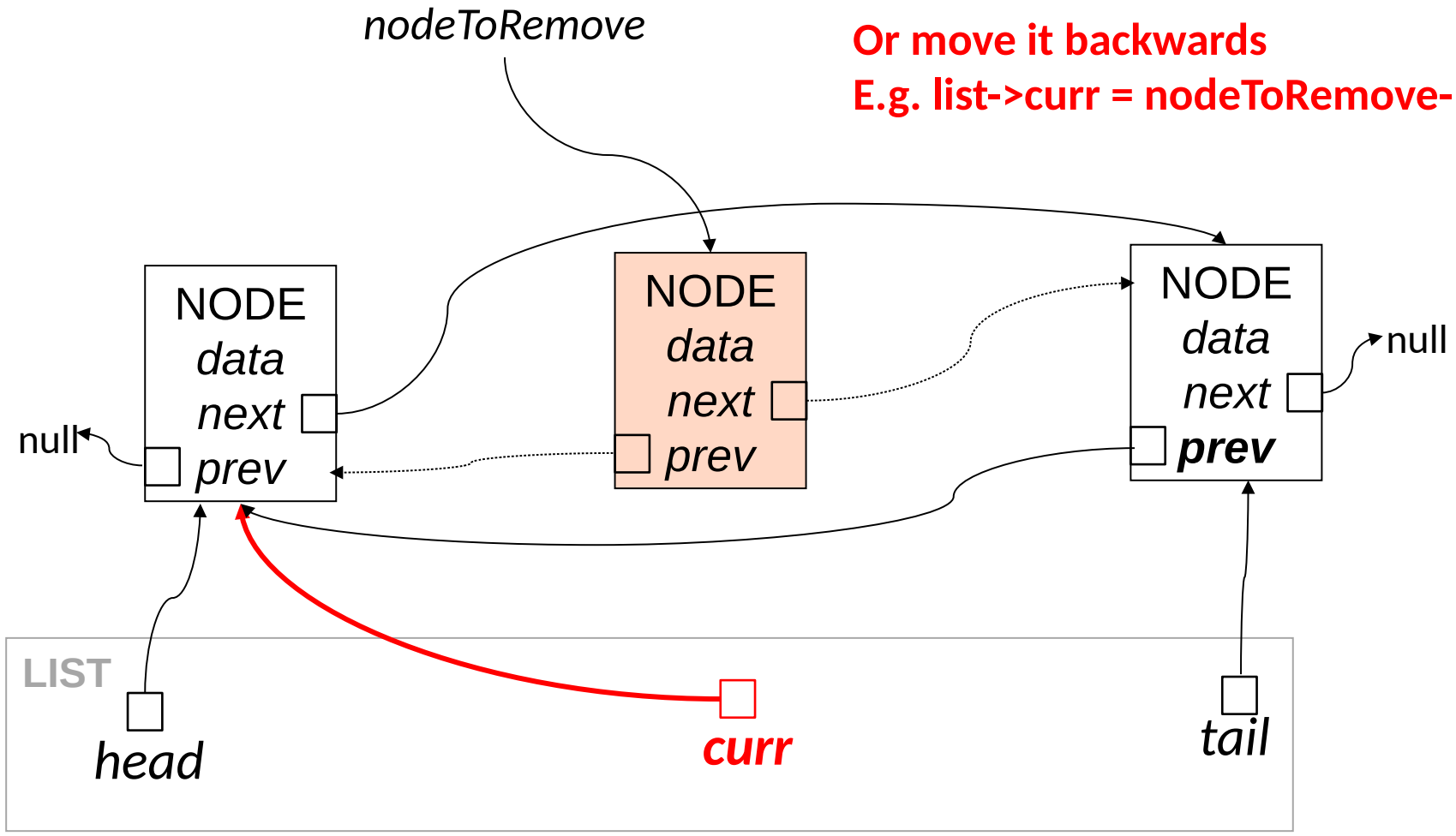

nodeToRemove
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

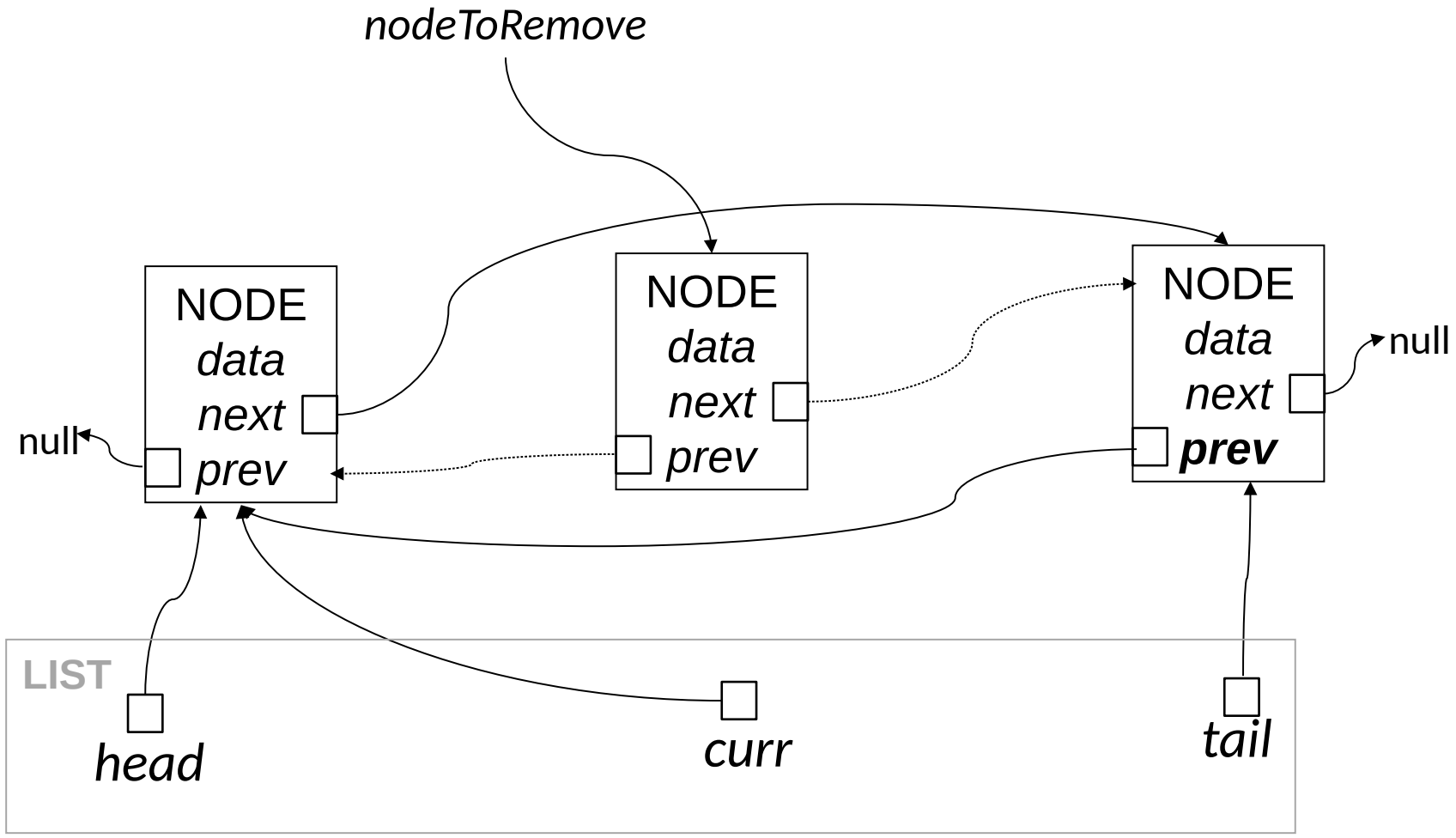

This node has now been safely detached from those around it
nodeToRemove
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail

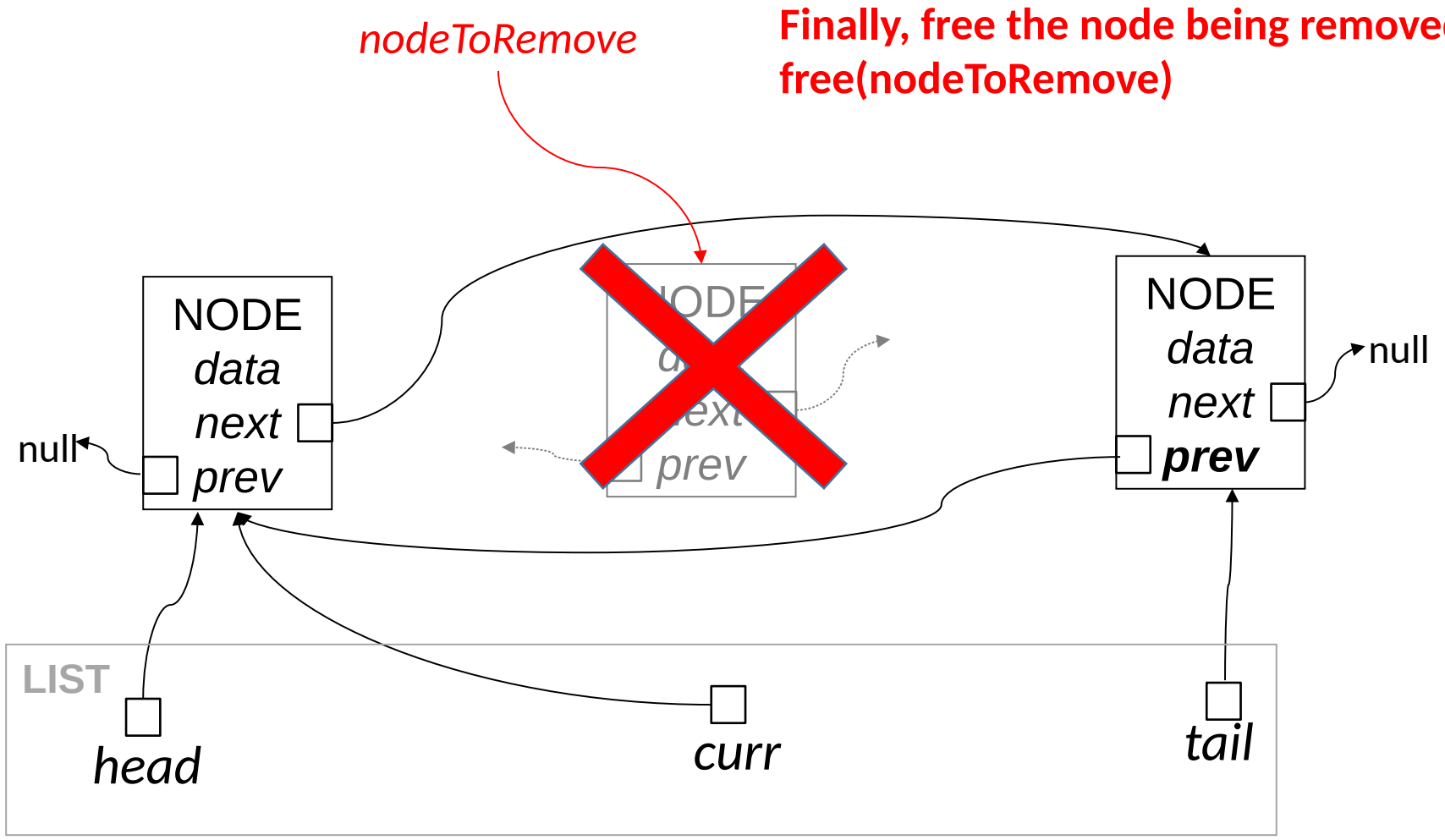


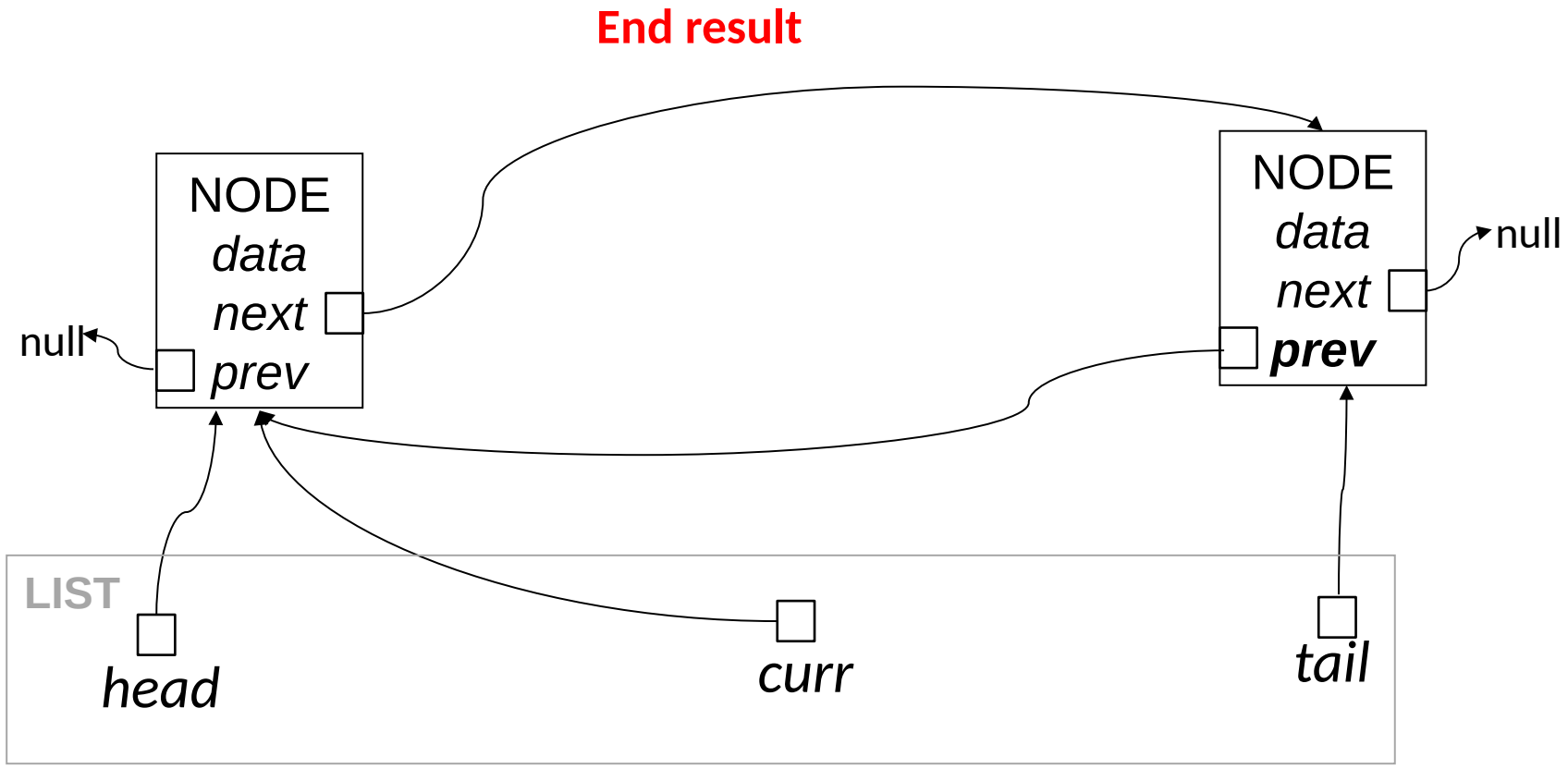
We need to move 'curr'
before we remove the node


nodeToRemove
Move curr forwards
E.g. list->curr = nodeToRemove->next
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail


nodeToRemove
Or move it backwards
E.g. list->curr = nodeToRemove->prev
NODE
data
next
prev
null
NODE
data
next
prev
NODE
data
next
prev
null
LIST
head
curr
tail


nodeToRemove
NODE
data
next
prev
NODE
data
next
prev
NODE
data
next
prev
null
null
LIST
head
curr
tail


nodeToRemove
Finally, free the node being removed
free(nodeToRemove)
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail


End result
NODE
data
next
prev
null
NODE
data
next
prev
null
LIST
head
curr
tail

Free the entire list

- See Stack example

End