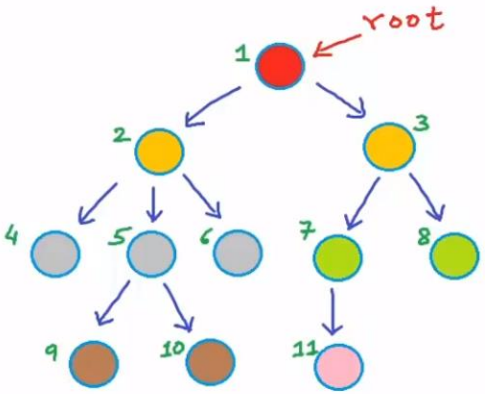
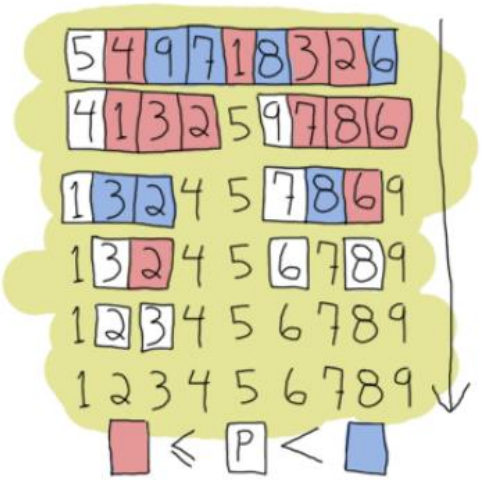
ECE 250 Data Structures & Algorithms



Linked Lists



Ziqiang Patrick Huang

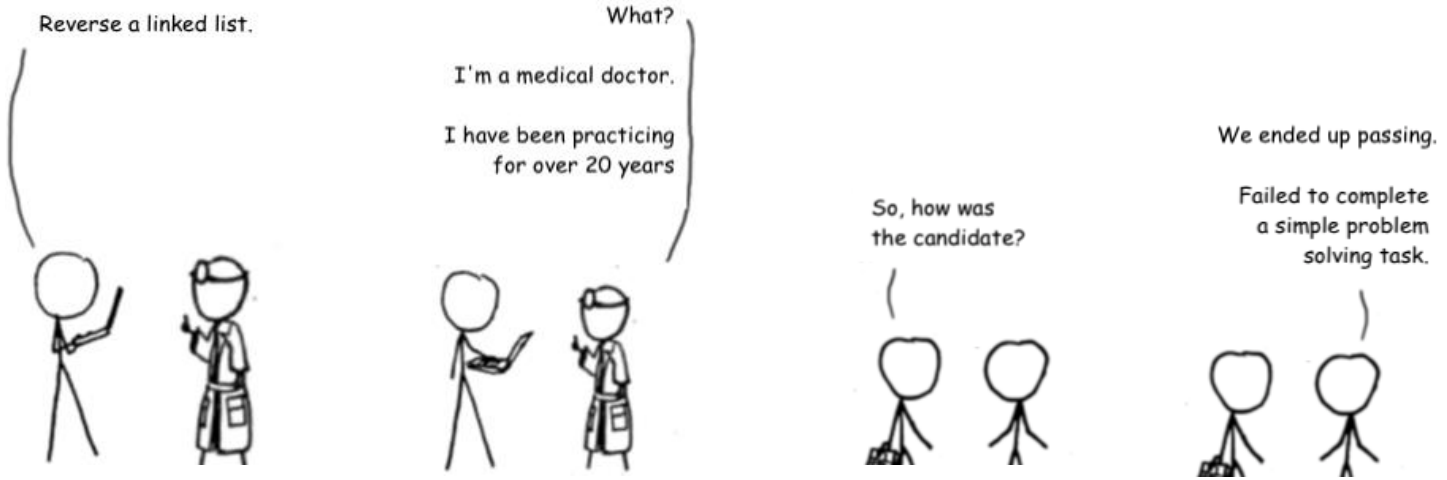
Electrical and Computer Engineering

University of Waterloo

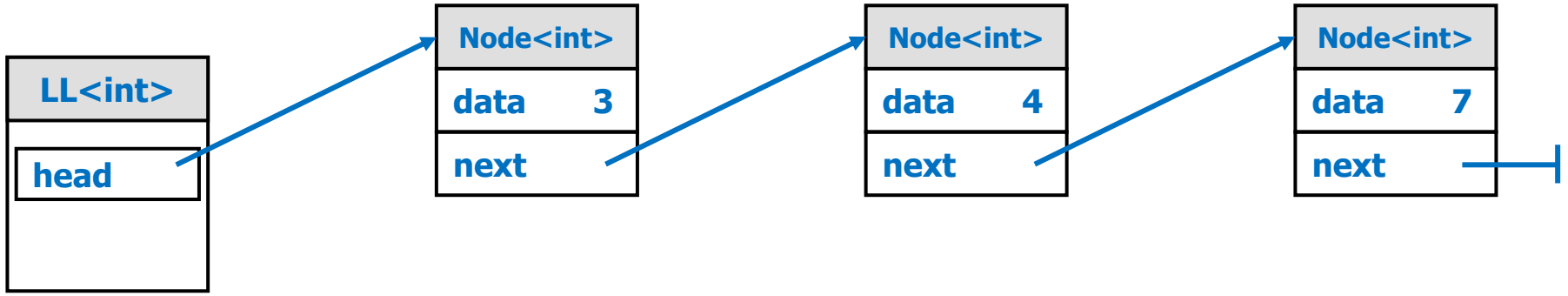
WATERLOO | ENGINEERING

What is a Linked List?

- Dynamic data structure: Linked Lists
  - Comprised of nodes, each node has
    - Data: whatever we want it to be: can template over this
    - Next: a pointer to the next node in the list
      - NULL for the last node

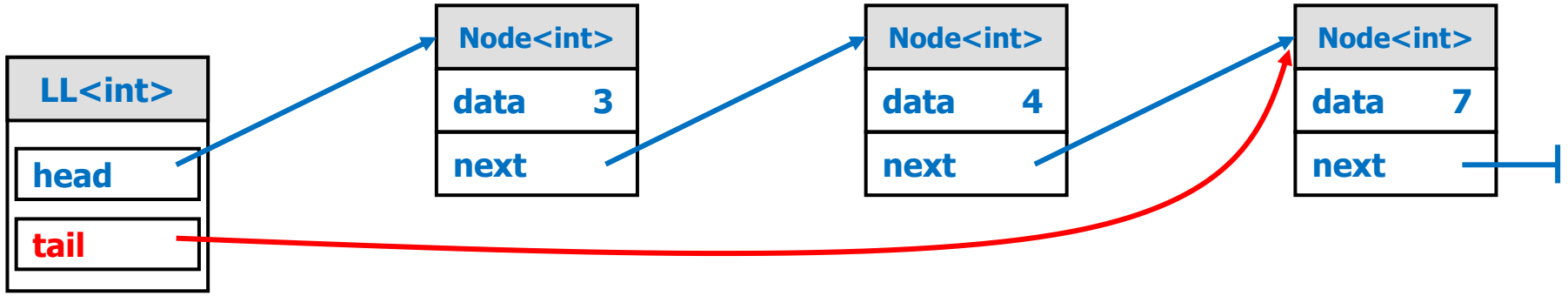


Singly-Linked List



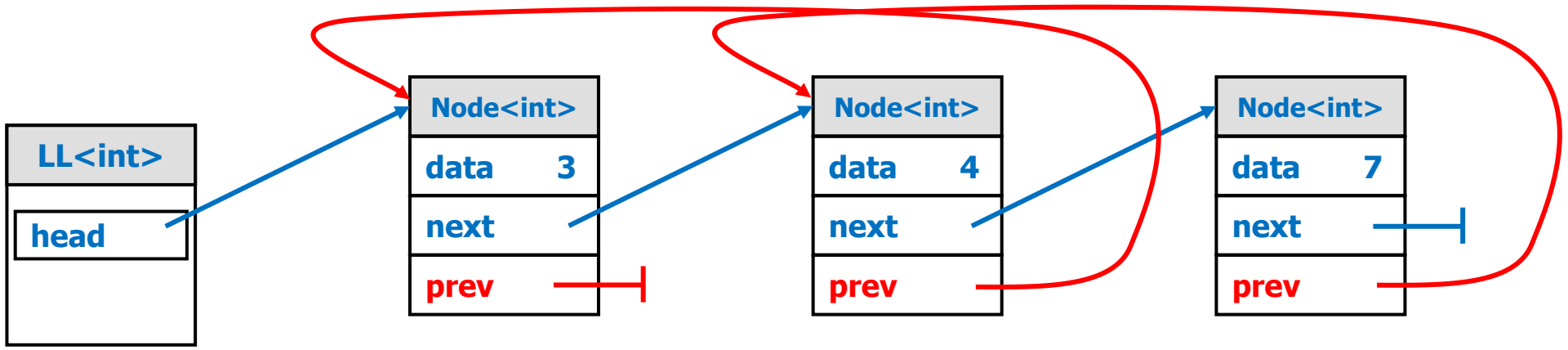
- Example Linked List
  - Has ints in it
  - 3 elements (3, 4, 7)
  - Made up of nodes (1 per element), linked together by next
  - LinkedList itself has a head pointer, points at first element

Singly-Linked List with Tail Pointer



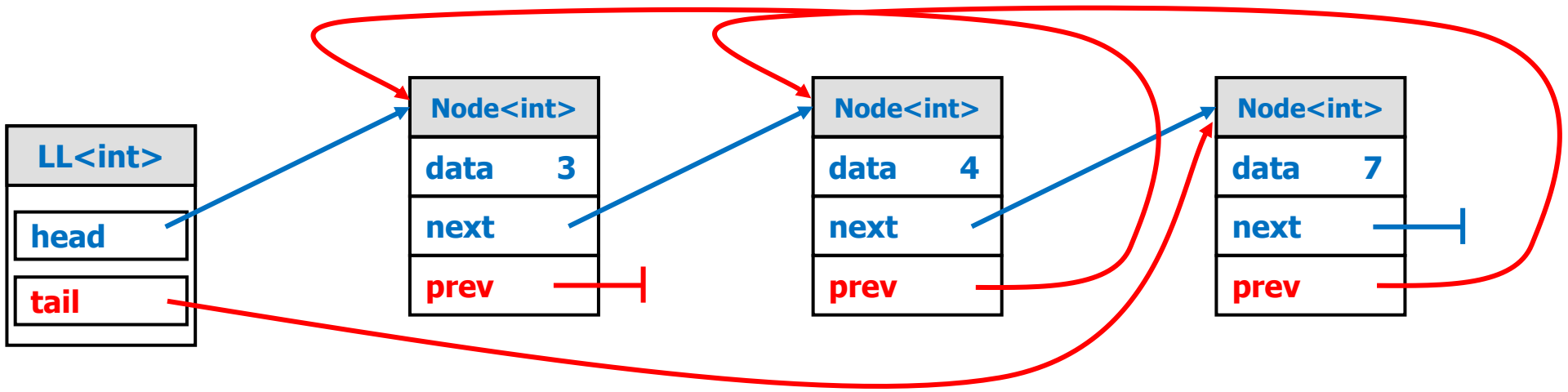
- Many variations on a Linked List
  - Sometimes we want to store the tail (last node)

Doubly-Linked List



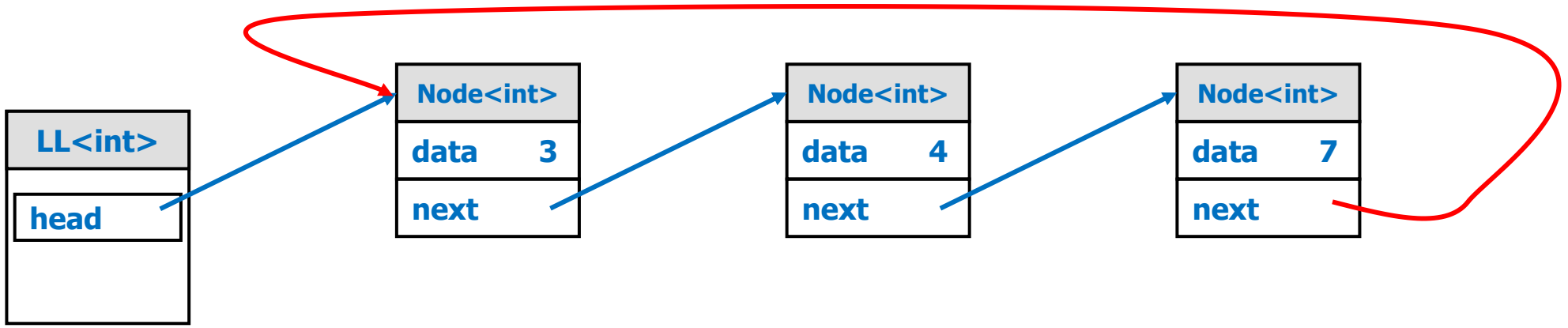
- Many variations on a Linked List
  - Sometimes we want to store the tail (last node)
  - Sometimes we doubly link the list: each node has a previous

Doubly-Linked List with Tail Pointer



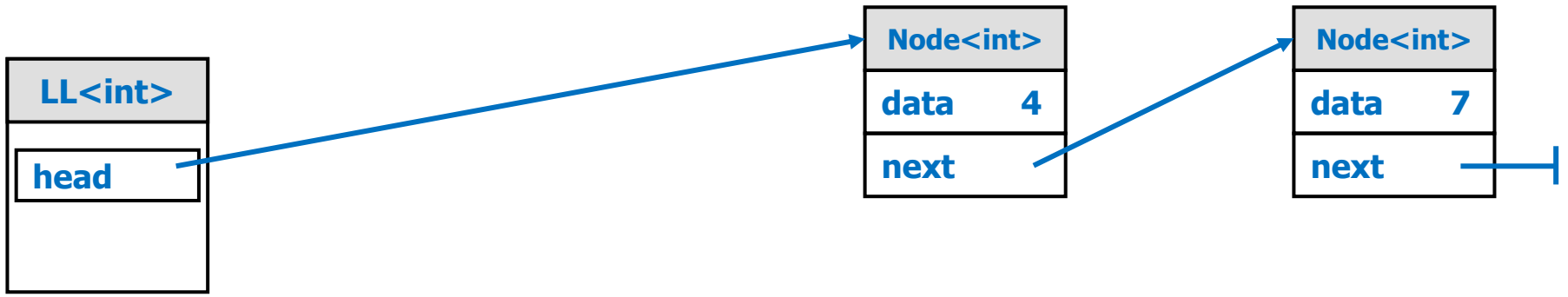
- Many variations on a Linked List
  - Sometimes we want to store the tail (last node)
  - Sometimes we doubly link the list: each node has a previous
  - We can do both of these together

Circularly-Linked List



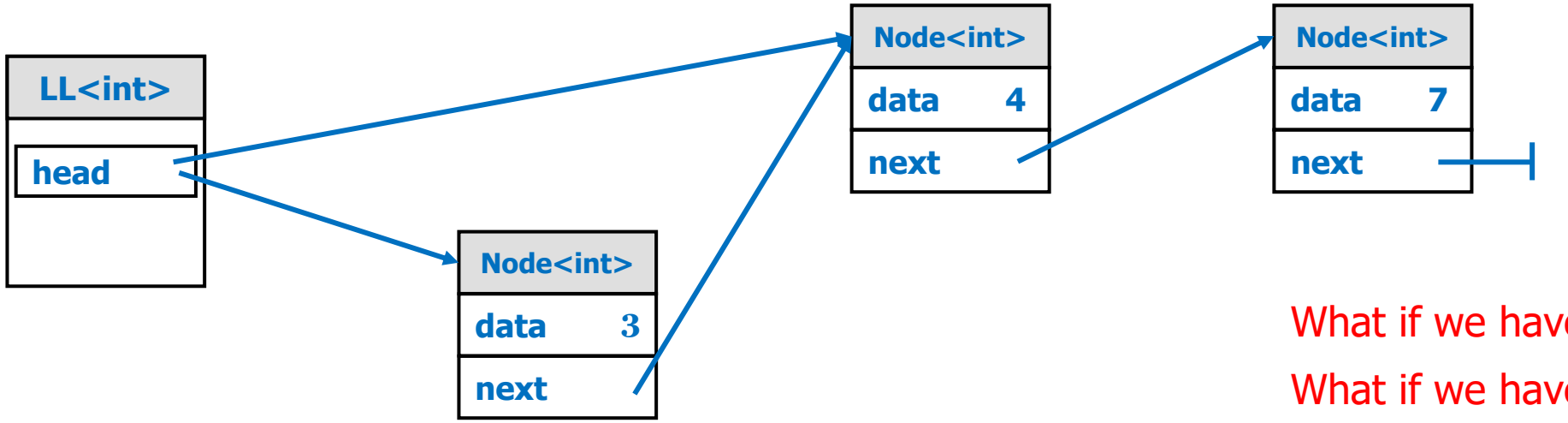
- Many variations on a Linked List
  - Sometimes we want to store the tail (last node)
  - Sometimes we doubly link the list: each node has a previous
  - We can do both of these together
  - Sometimes we might want a circular list (when?)

Linked List Basic Operations: Add to Front



- Our current Linked List with 2 nodes

Linked List Basic Operations: Add to Front



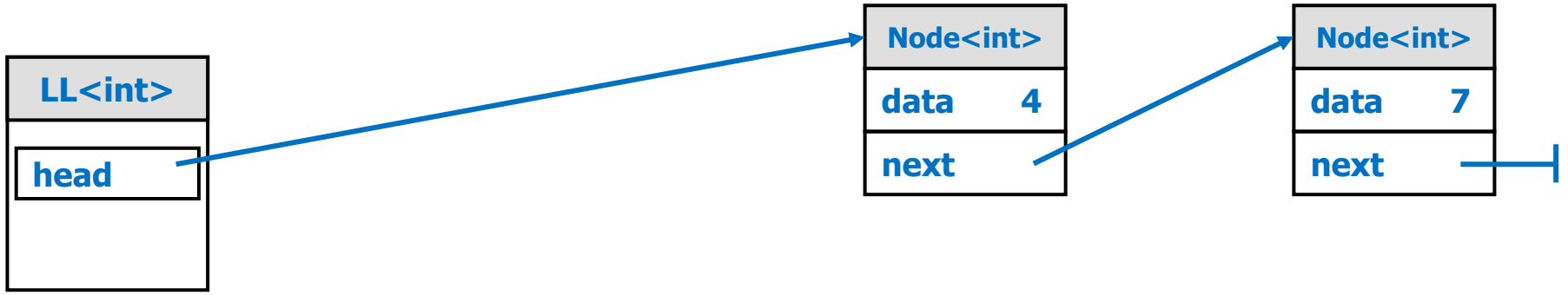
What if we have a tail pointer?

What if we have a doubly-linked list?

- Now we want to add 3 to the front of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … head
  - We need to point head at the node we just made

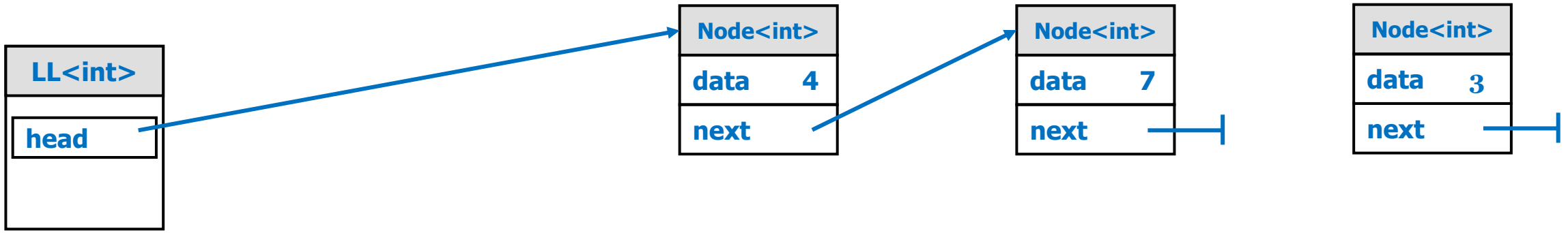
O(1) 😀

Linked List Basic Operations: Add to Back



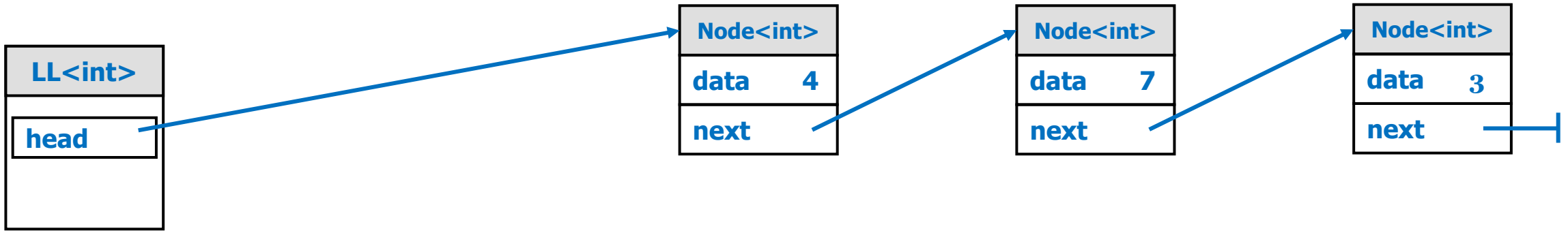
- Our current Linked List with 2 nodes

Linked List Basic Operations: Add to Back



- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL

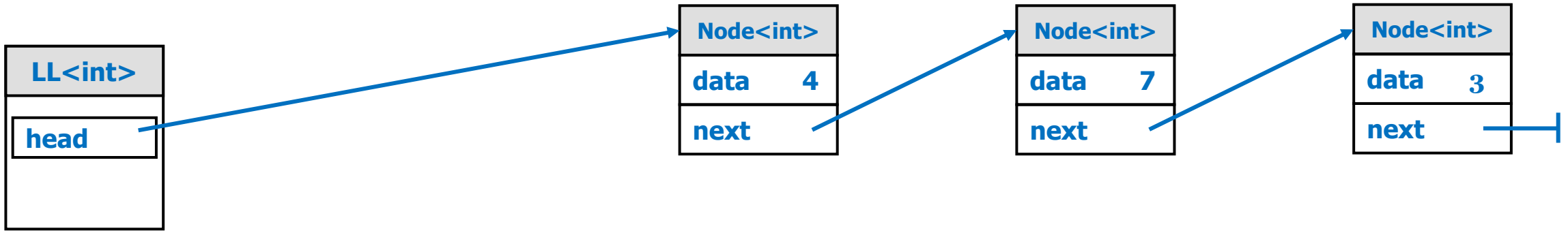
Linked List Basic Operations: Add to Back



- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL
  - We need to point the current last node's next at the node we just made

How do we get to this? Need to iterate through the entire linked list!

Linked List Basic Operations: Add to Back



What if we have a tail pointer?

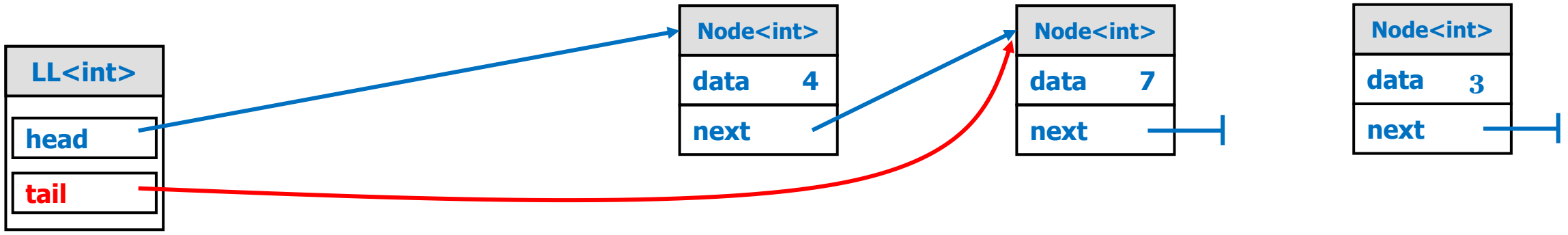
- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL

O(1) 😃

  - We need to point the current last node's next at the node we just made O(n) 😔

How do we get to this? Need to iterate through the entire linked list!

Linked List Basic Operations: Add to Back

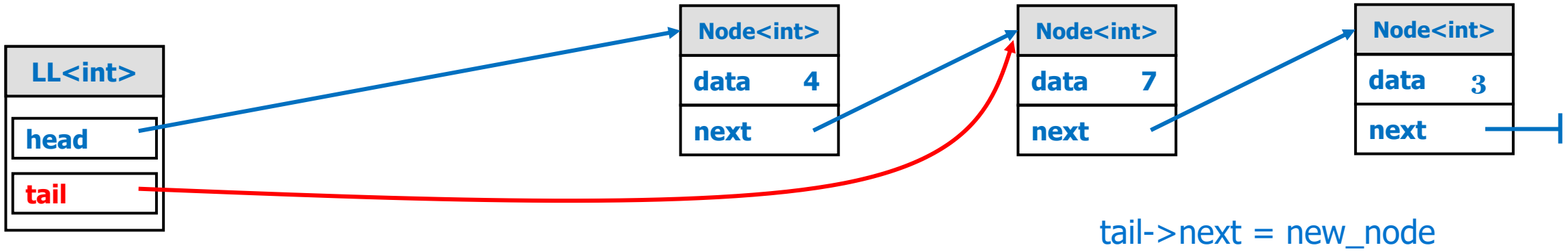


What if we have a tail pointer?

- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL
  - We need to point the current last node's next at the node we just made

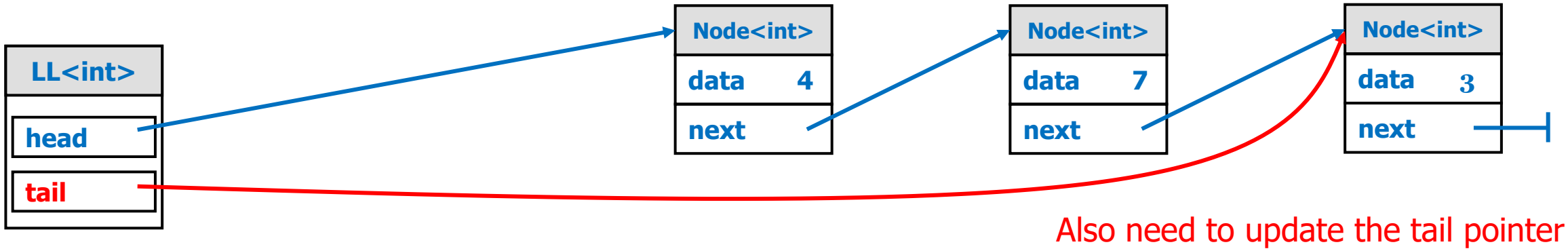
Easy, have access to current last node with tail pointer

Linked List Basic Operations: Add to Back



- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL
  - We need to point the current last node's next at the node we just made

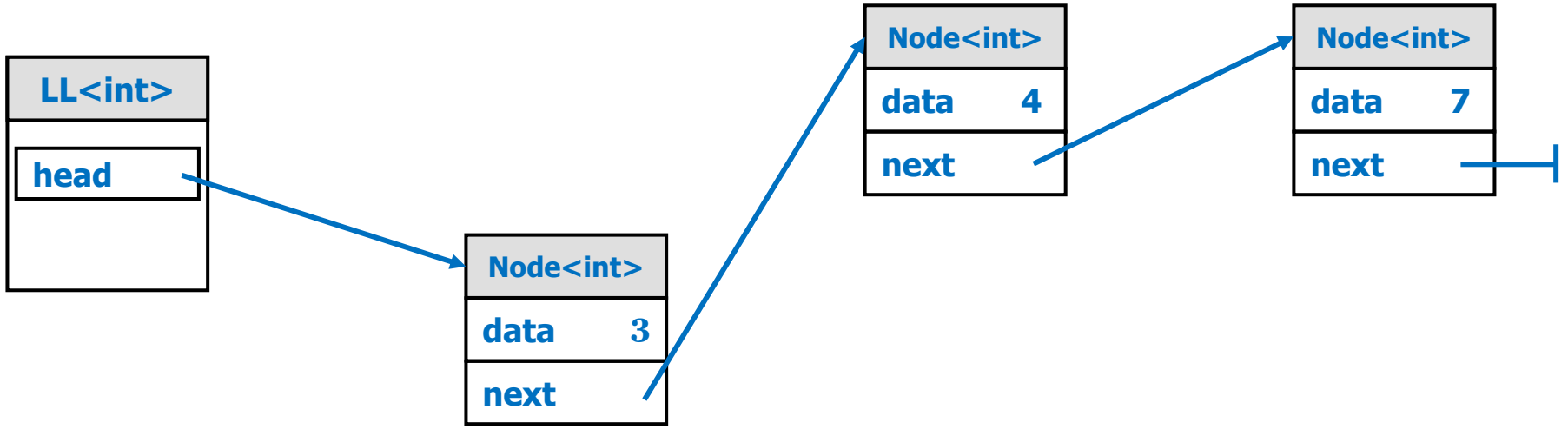
Linked List Basic Operations: Add to Back



- Now we want to add 3 to the back of the list
  - We need to make a new node
  - We need to make its data be 3
  - We need to make its next be … NULL
  - We need to point the current last node's next at the node we just made

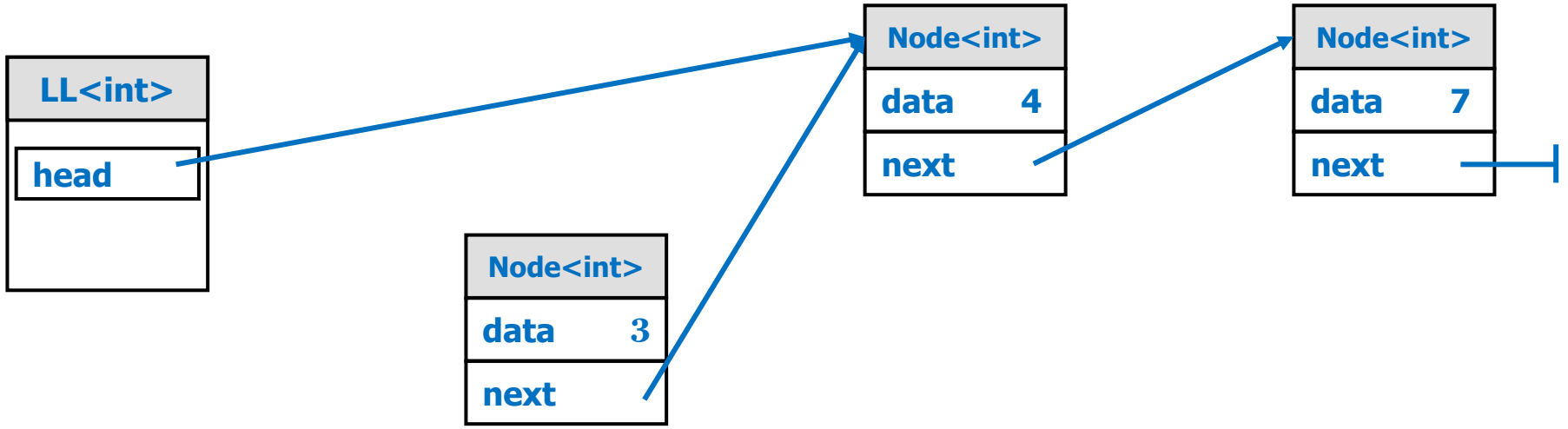
O(1) 😃

Linked List Basic Operations: Remove from Front



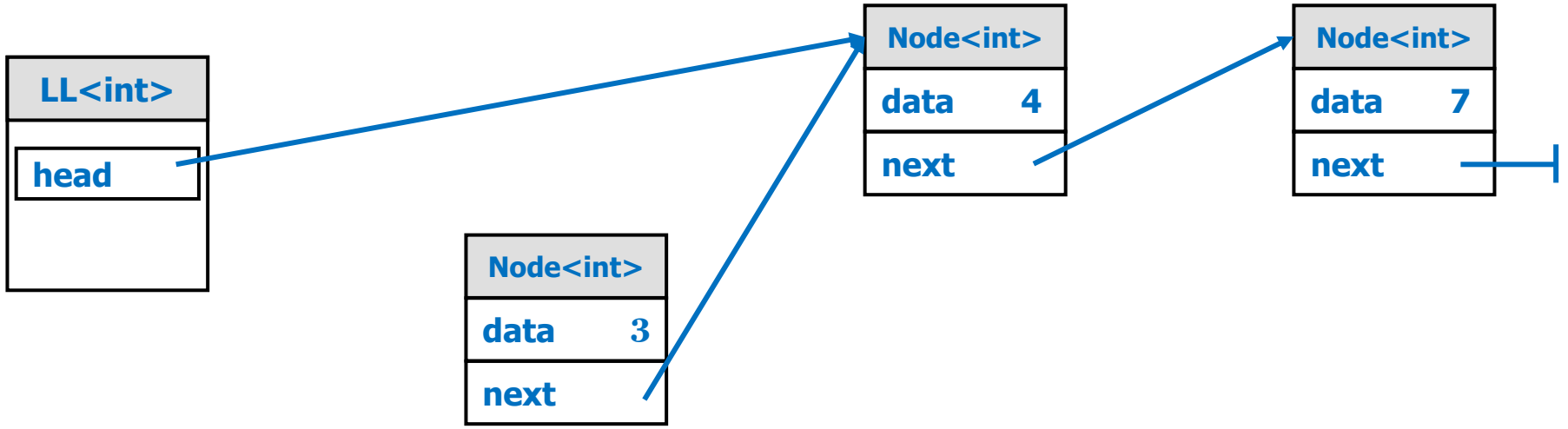
- Now we want to remove 3 from the front of the list

Linked List Basic Operations: Remove from Front



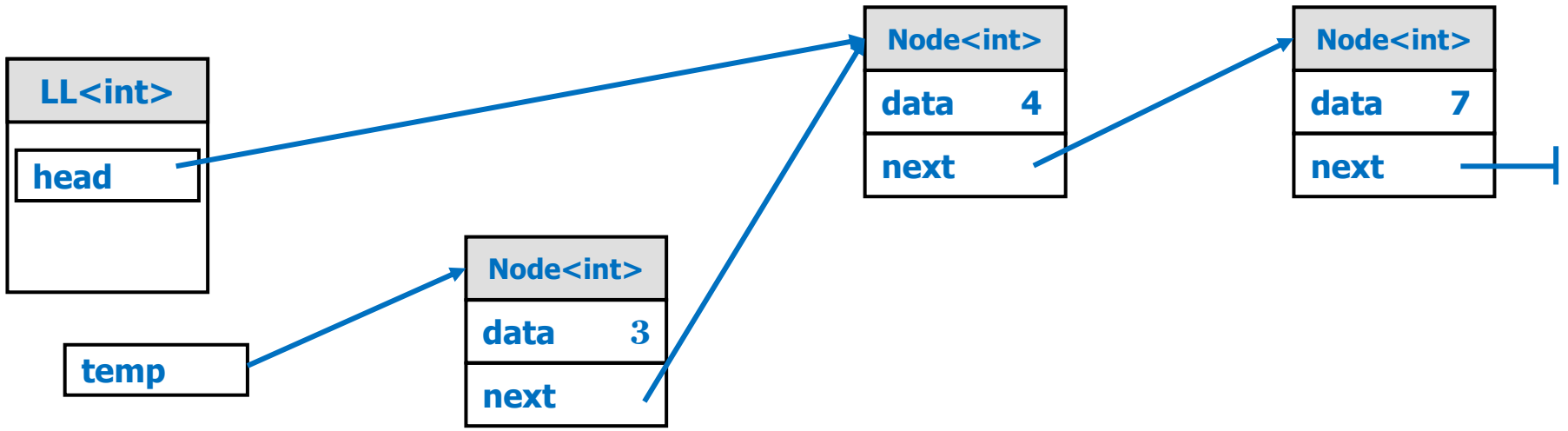
- Now we want to remove 3 from the front of the list
  - Pretty straight forward, just need to change the head pointer

Linked List Basic Operations: Remove from Front



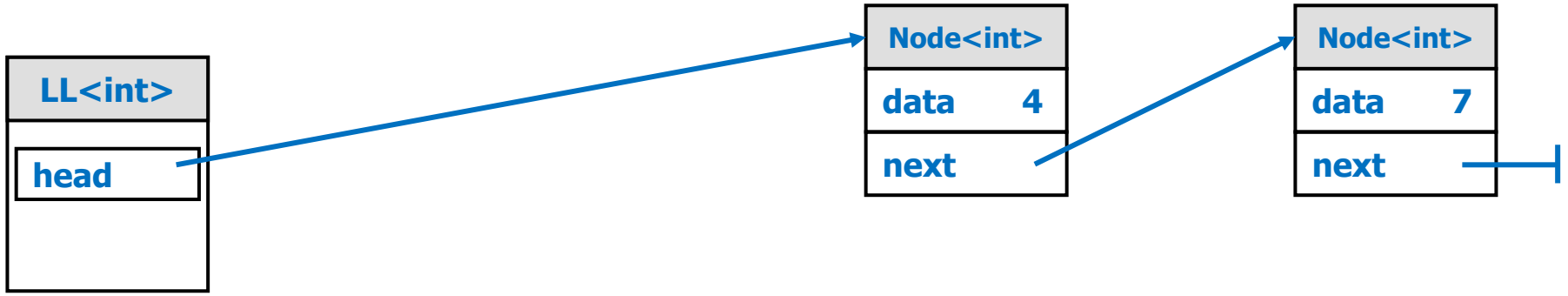
- Now we want to remove 3 from the front of the list
  - Pretty straight forward, just need to change the head pointer
  - Also need to delete the node (not necessary in garbage collected language, e.g., Java)

Linked List Basic Operations: Remove from Front



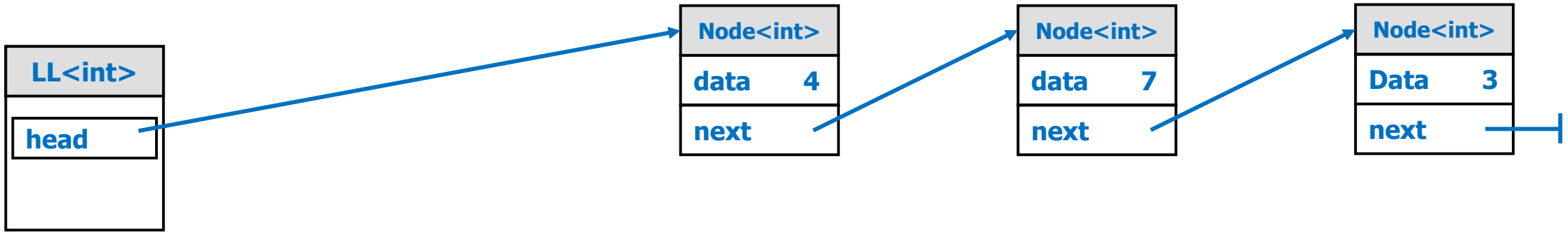
- Now we want to remove 3 from the front of the list
  - Pretty straight forward, just need to change the head pointer
  - Also need to delete the node (not necessary in garbage collected language, e.g., Java)
  - First need to store head in a temp pointer

Linked List Basic Operations: Remove from Front



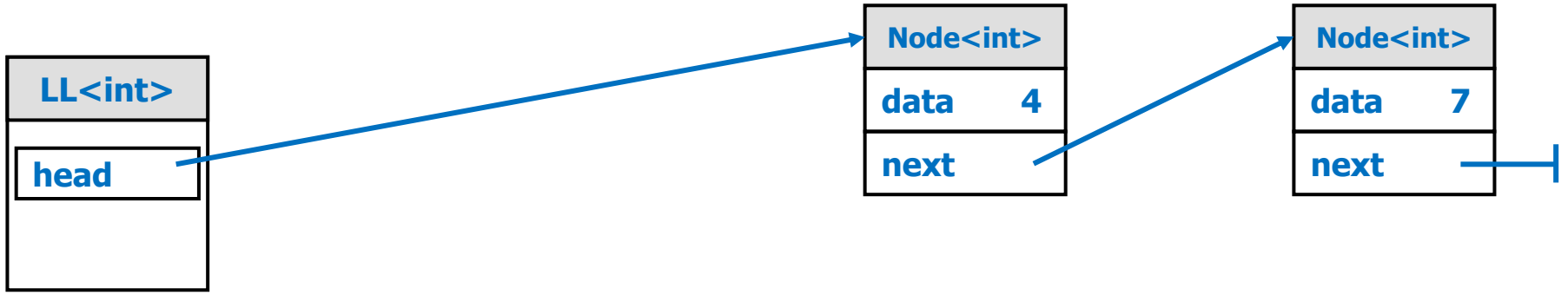
- Now we want to remove 3 from the front of the list
  - Pretty straight forward, just need to change the head pointer
  - Also need to delete the node (not necessary in garbage collected language, e.g., Java)
  - First need to store head in a temp pointer
  - Complexity same as add to front: O(1)

Linked List Basic Operations: Remove from Back



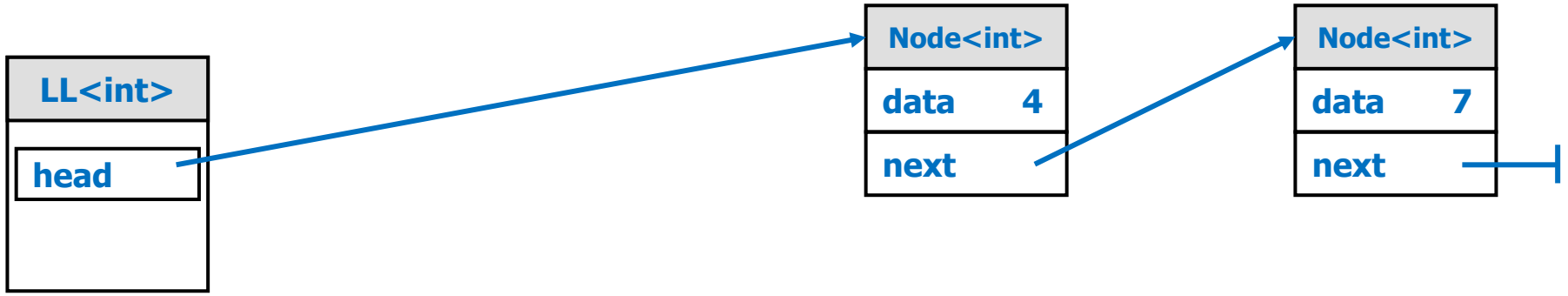
- Now we want to remove 3 from the back of the list

Linked List Basic Operations: Remove from Back



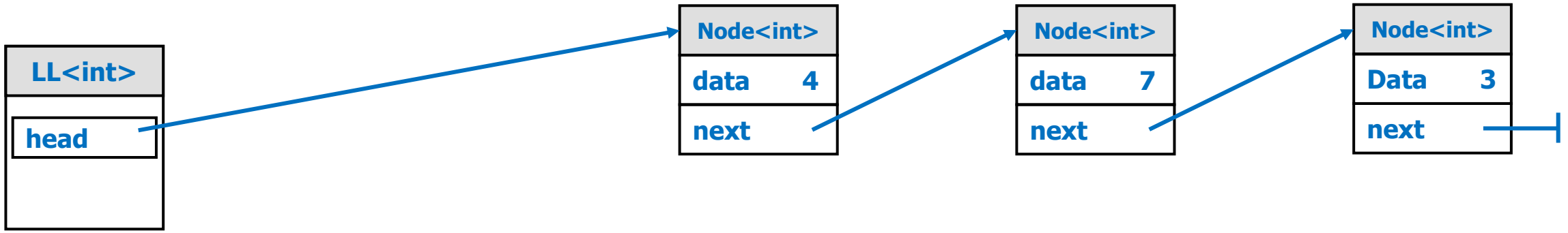
- Now we want to remove 3 from the back of the list
  - Need delete the last node and set the second last node's next to NULL

Linked List Basic Operations: Remove from Back



- Now we want to remove 3 from the back of the list
  - Need delete the last node and set the second last node's next to NULL
  - But again, need to iterate through the entire linked list → O(n)

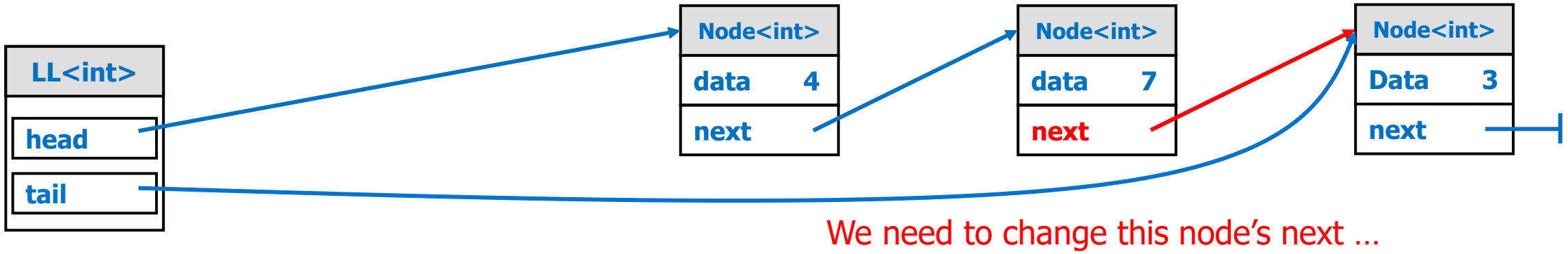
Linked List Basic Operations: Remove from Back



What if we have a tail pointer?

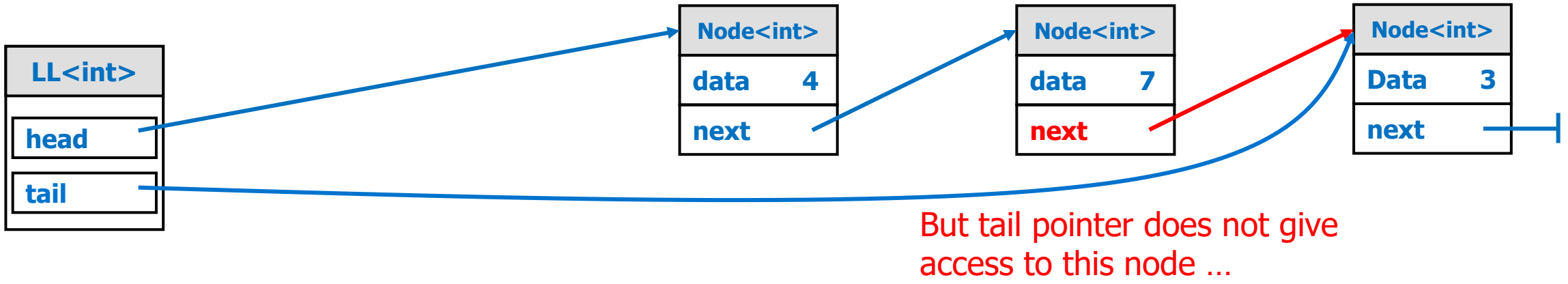
- Now we want to remove 3 from the back of the list
  - Need delete the last node and set the second last node's next to NULL
  - But again, need to iterate through the entire linked list → O(n)

Linked List Basic Operations: Remove from Back



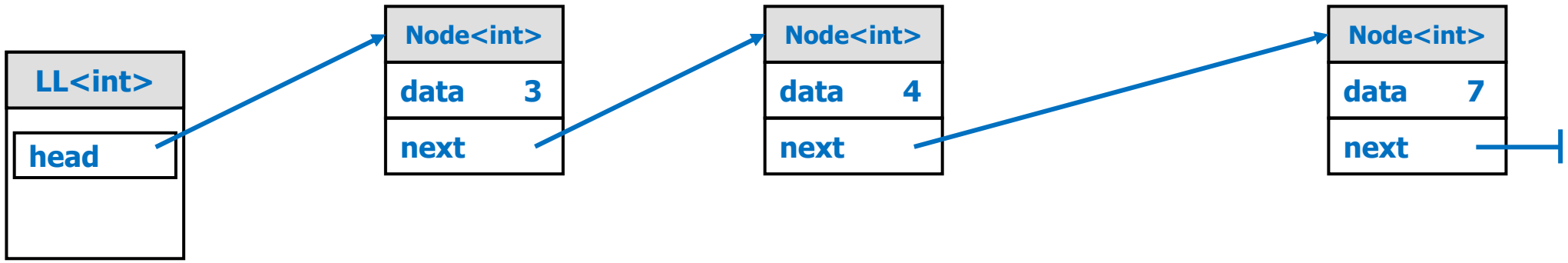
- Now we want to remove 3 from the back of the list
  - Need delete the last node and set the second last node's next to NULL
  - But again, need to iterate through the entire linked list → O(n)

Linked List Basic Operations: Remove from Back



- Now we want to remove 3 from the back of the list
  - Need delete the last node and set the second last node's next to NULL
  - But again, need to iterate through the entire linked list → O(n)
  - Having a tail pointer alone would NOT help
  - Doubly-linked list + tail pointer would make this O(1)

Linked List Basic Operations: Searching

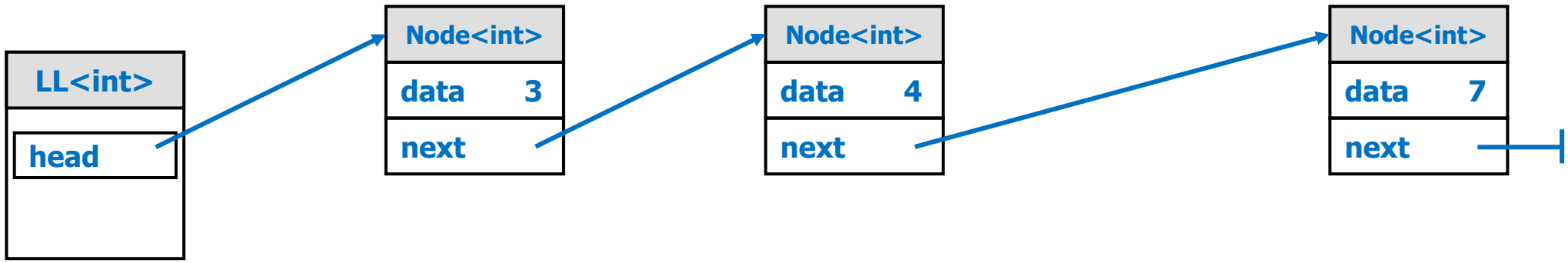


- Can have many variations
  - Checking if a specific item is in the LL
  - Remove a specific item from the LL
  - Insert an item at a specific place in the LL
    - e.g., before/after a specific item

All of these involve searching for a specific item in LL

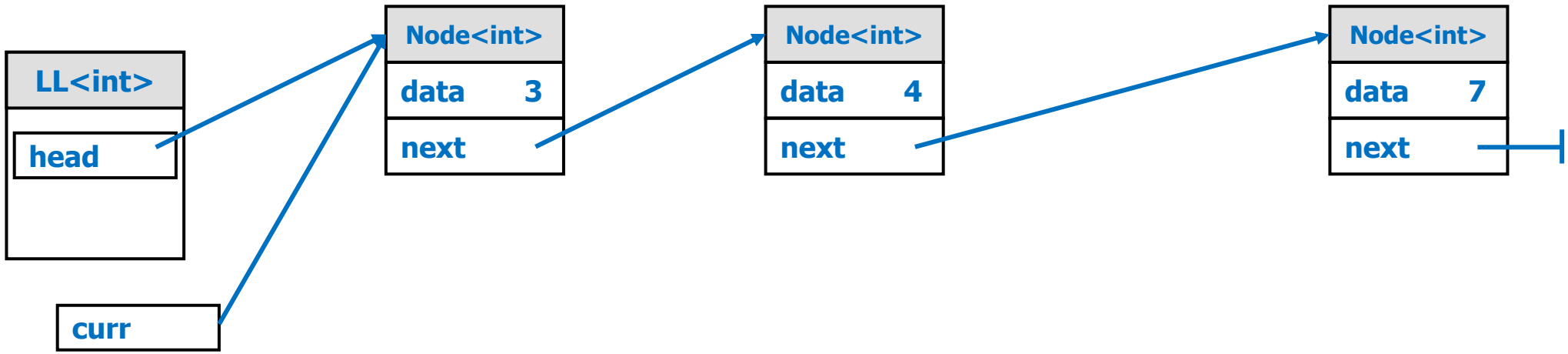
O(n)

Searching Example: Insert in Sorted Order



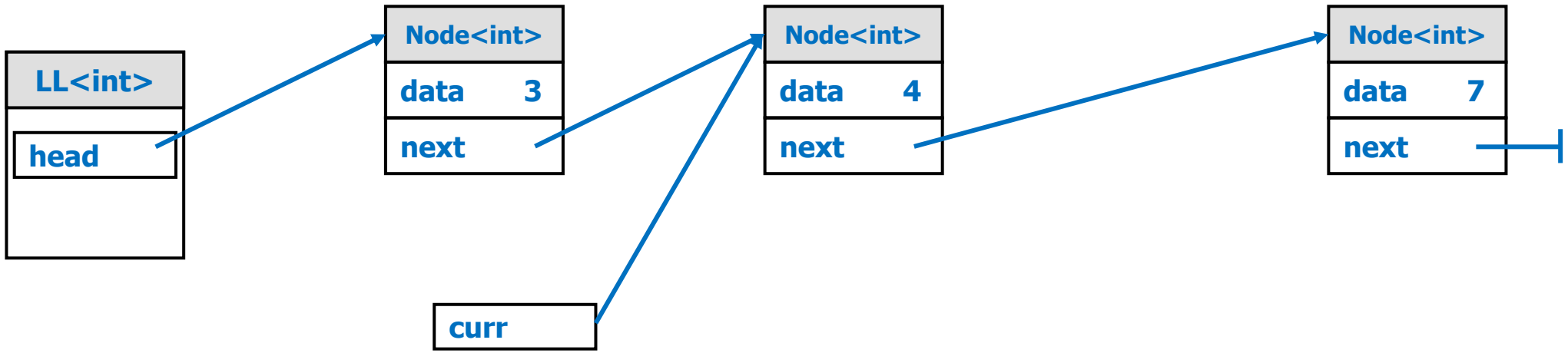
- Now we want to insert 5 between 4 and 7

Common Approach: Pointer to Node Before



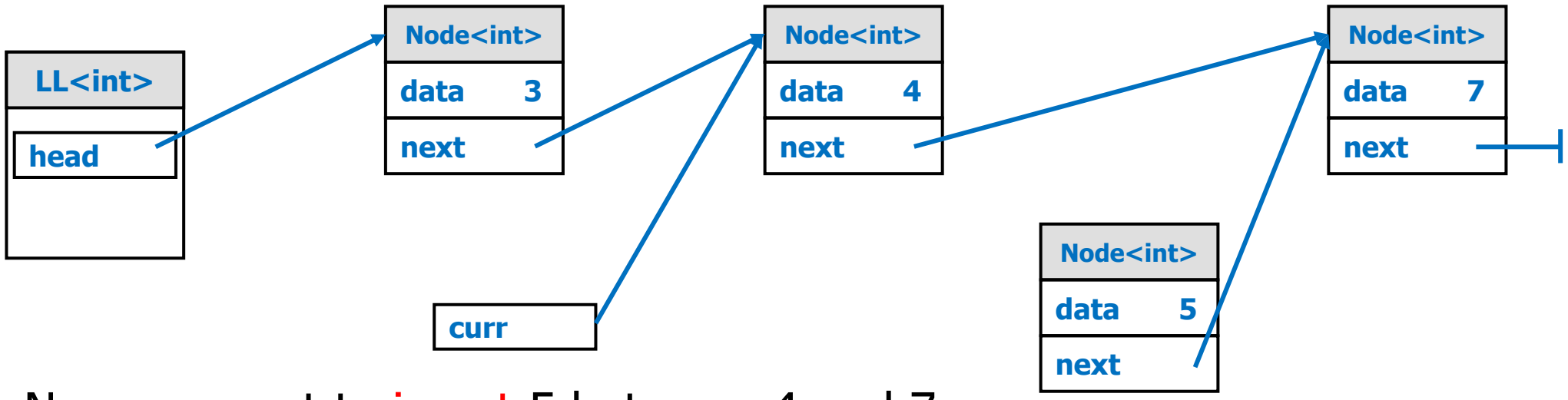
- Now we want to insert 5 between 4 and 7
  - Search for the node before using a "current pointer"

Common Approach: Pointer to Node Before



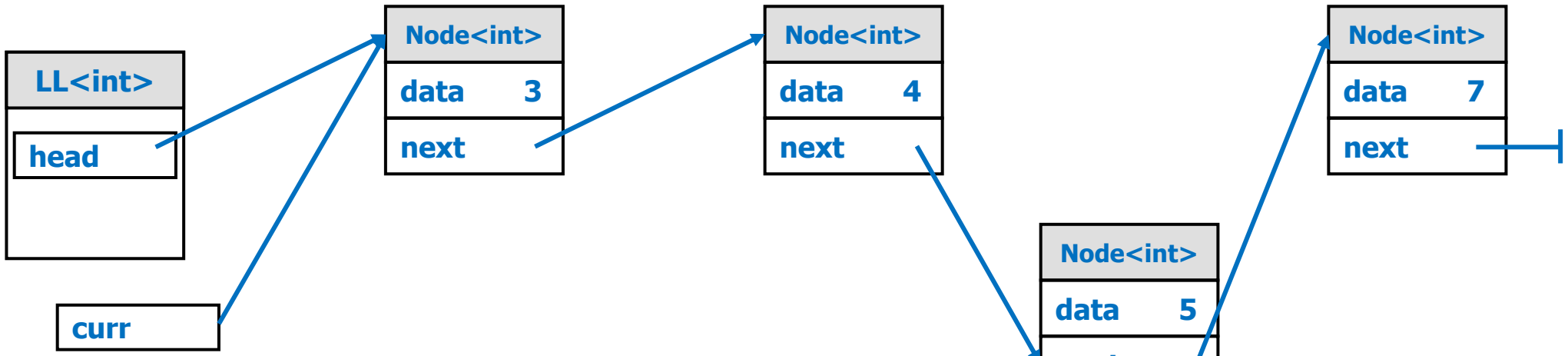
- Now we want to insert 5 between 4 and 7
  - Search for the node before using a "current pointer"
  - Once found (what is the condition here?) data(5) < curr->next->data

Common Approach: Pointer to Node Before



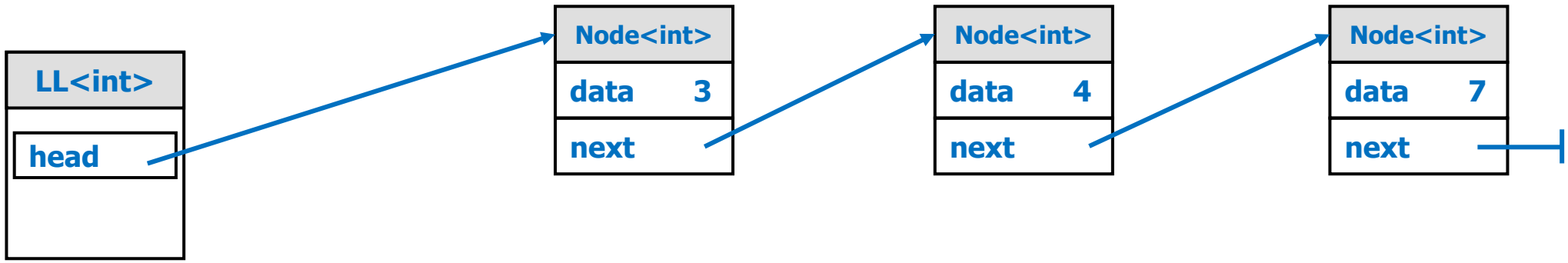
- Now we want to insert 5 between 4 and 7
  - Search for the node before using a "current pointer"
  - Once found (what is the condition here?) data(5) < curr->next->data
    - Create a new node
    - Data set to 5; next set to … curr -> next

Common Approach: Pointer to Node Before



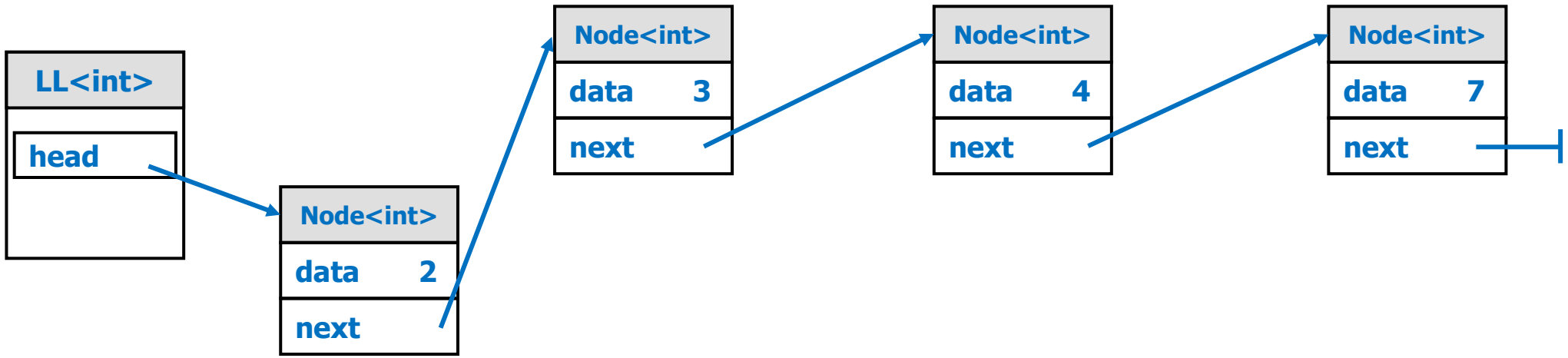
- Now we want to insert 5 between 4 and 7
  - Search for the node before using a "current pointer"
  - Once found (what is the condition here?) data(5) < curr->next->data
    - Create a new node
    - Data set to 5; next set to … curr -> next
  - Set the curr ->next to the new node
  - Corner case: add to front, need to be handled separately

Pointer to Node Before Corner Case



- Now we want to insert 2
  - The previous approach would have inserted 2 between 3 and 4
  - What is the corner case condition we are looking for here? data < head->data

Pointer to Node Before Corner Case



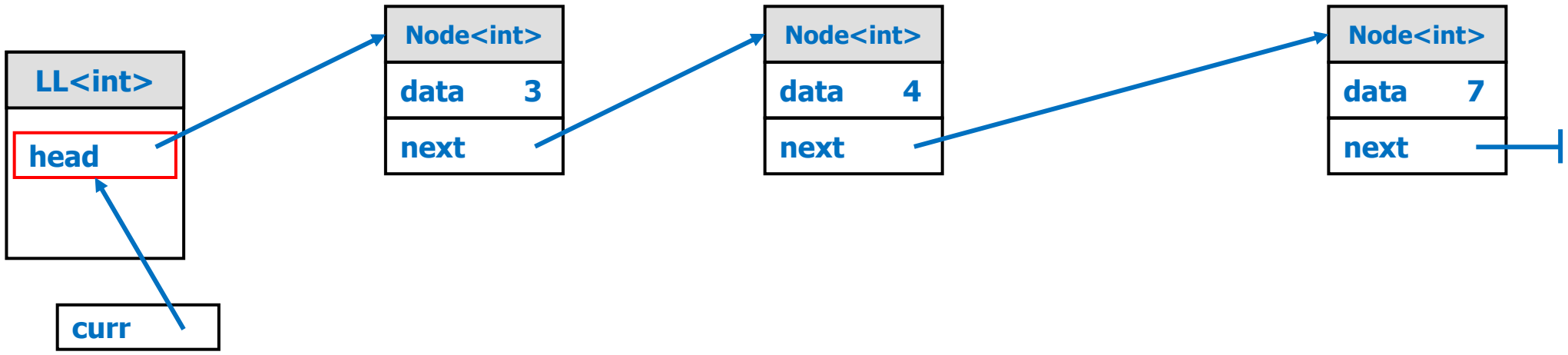
- Now we want to insert 2
  - The previous approach would have inserted 2 between 3 and 4
  - What is the corner case condition we are looking for here? data < head->data
  - What to do here? head = new Node(data, head)

Unify Common & Corner Cases

- What do the common case and the corner case have in common?
  - Corner case: we have to change `head`
  - Common case: change the `next` field of a node
  - They are both `Node *`s

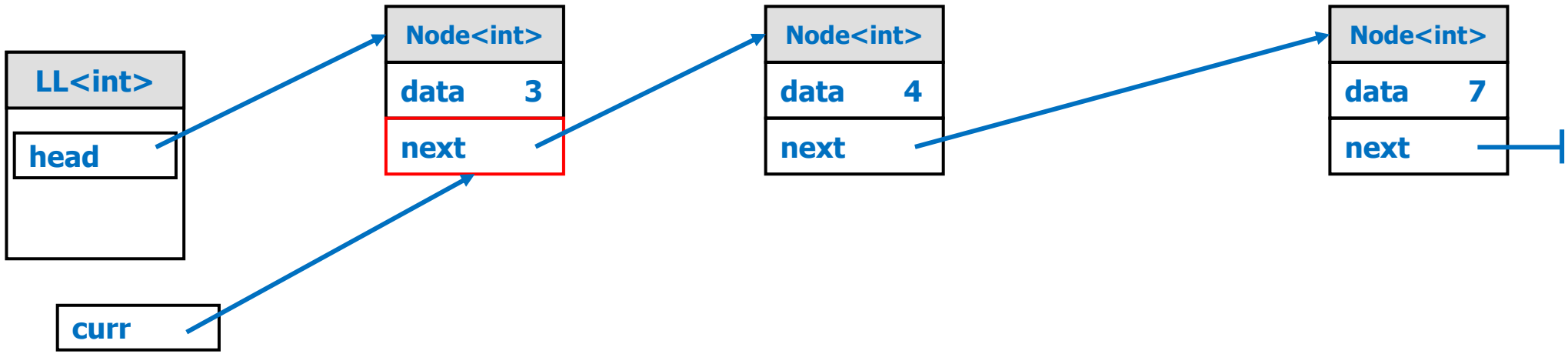
- Can have a `Node **` that points at "the box we might want to change"
  - Can point it at either `head` or the `next` field of a node
  - More elegant approach, but a little bit more pointer sophistication

Pointer to a Pointer Approach



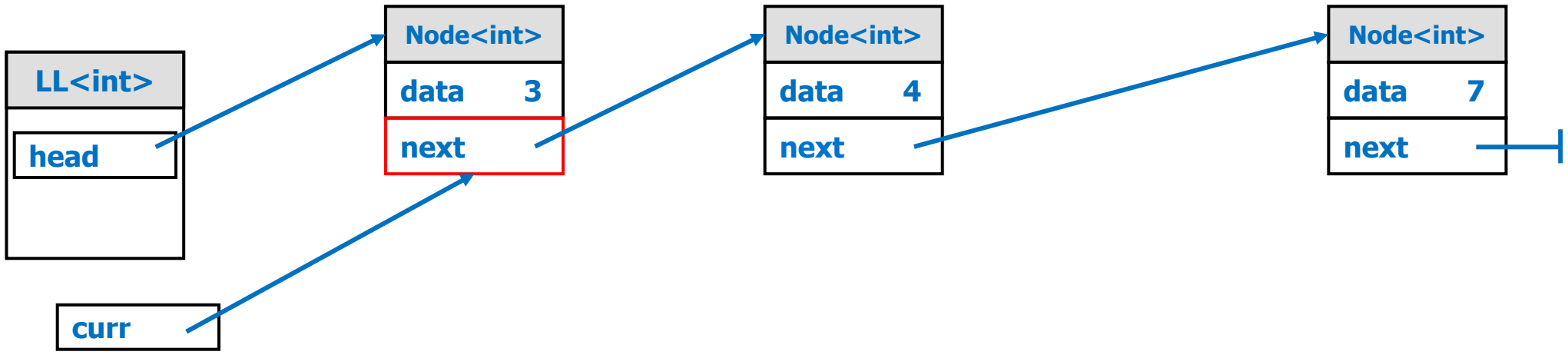
- Again we want to insert 5 between 4 and 7
  - curr is a Node **
  - Start it at &head (pointing at head's box)

Pointer to a Pointer Approach



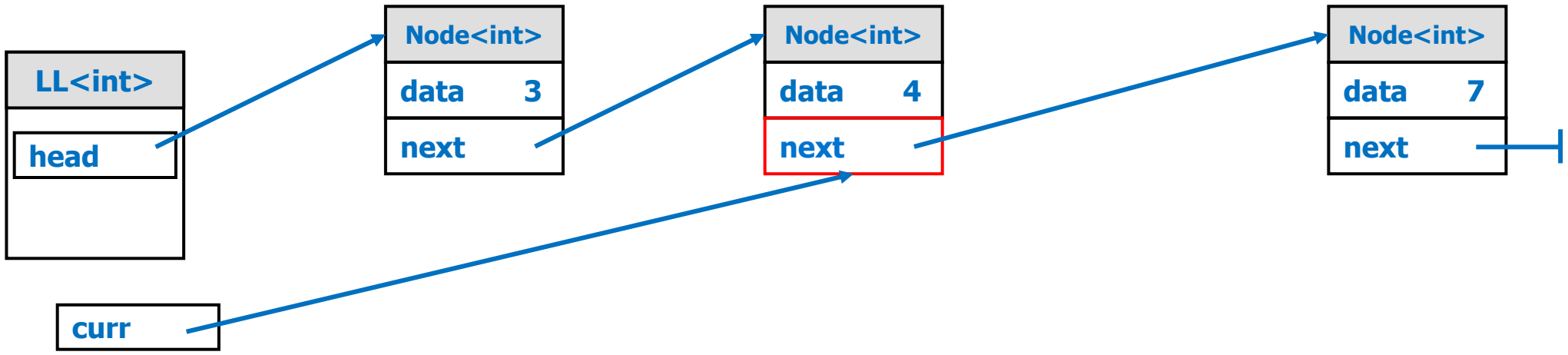
- Again we want to insert 5 between 4 and 7
  - curr is a Node **
  - Start it at &head (pointing at head's box)
  - Now as we search, we point it at the "next" fields

Pointer to a Pointer Approach



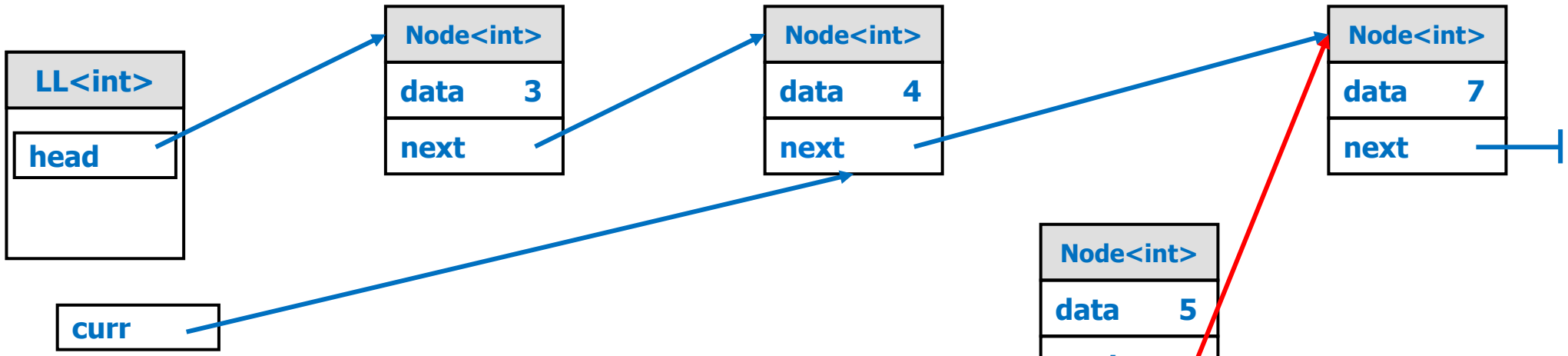
- Again we want to insert 5 between 4 and 7
  - What is the condition we are looking for? (*current)->data > data(5)
  - Is this the "box" we want to change? No, because (*current)->data = 4 (4 < 5)

Pointer to a Pointer Approach



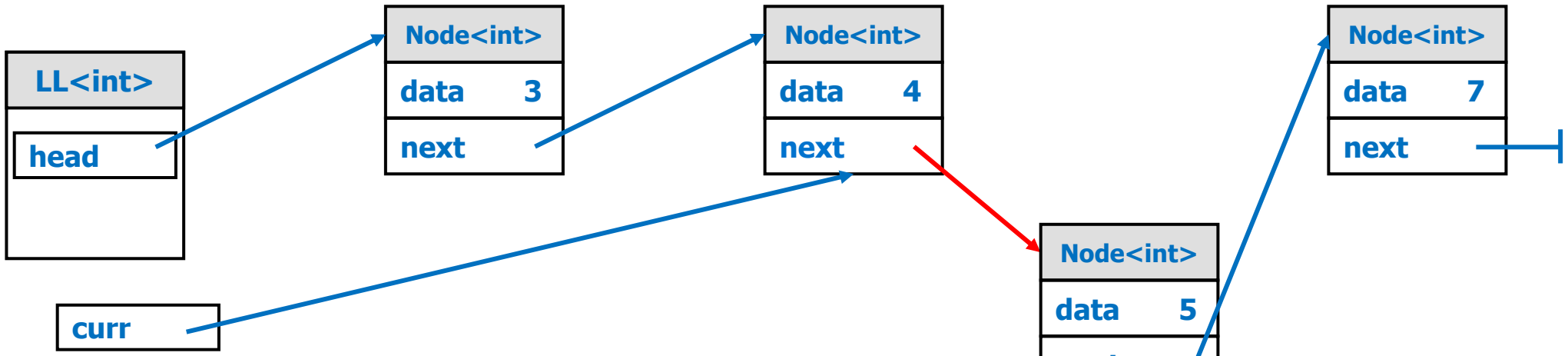
- Again we want to insert 5 between 4 and 7
  - Advance curr
  - Now is this the "box" we want to change? Yes, because (*current)->data = 7 (7 > 5)

Pointer to a Pointer Approach



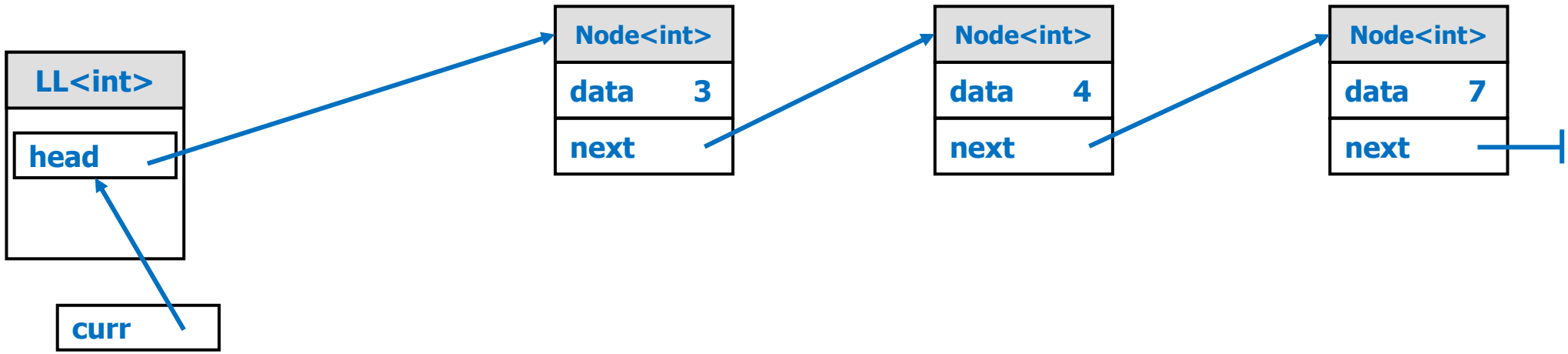
- Again we want to insert 5 between 4 and 7
  - Make a new Node with data = 5, next = ? *current

Pointer to a Pointer Approach



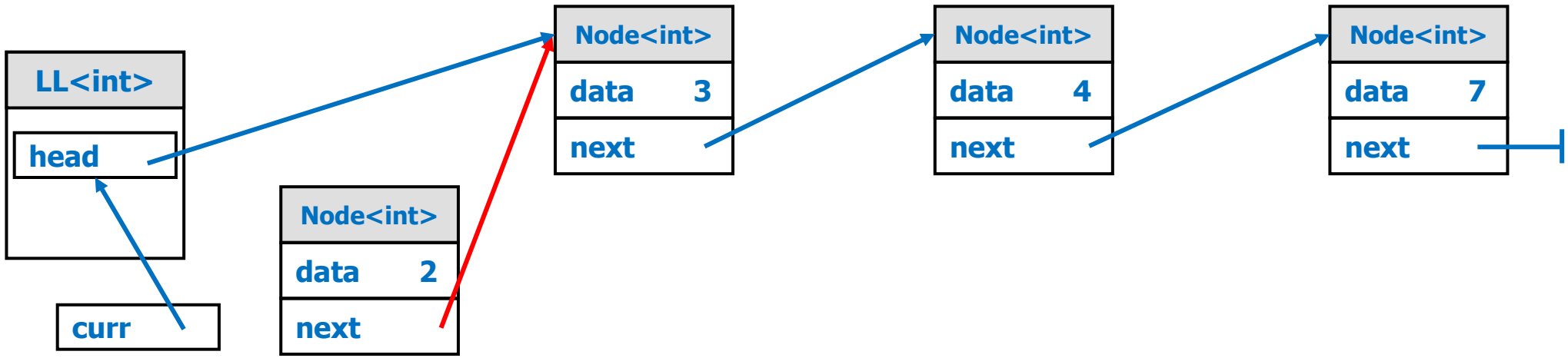
- Again we want to insert 5 between 4 and 7
  - Make a new Node with data = 5, next = ? *current
  - Set *current to that new node

Pointer to a Pointer Approach



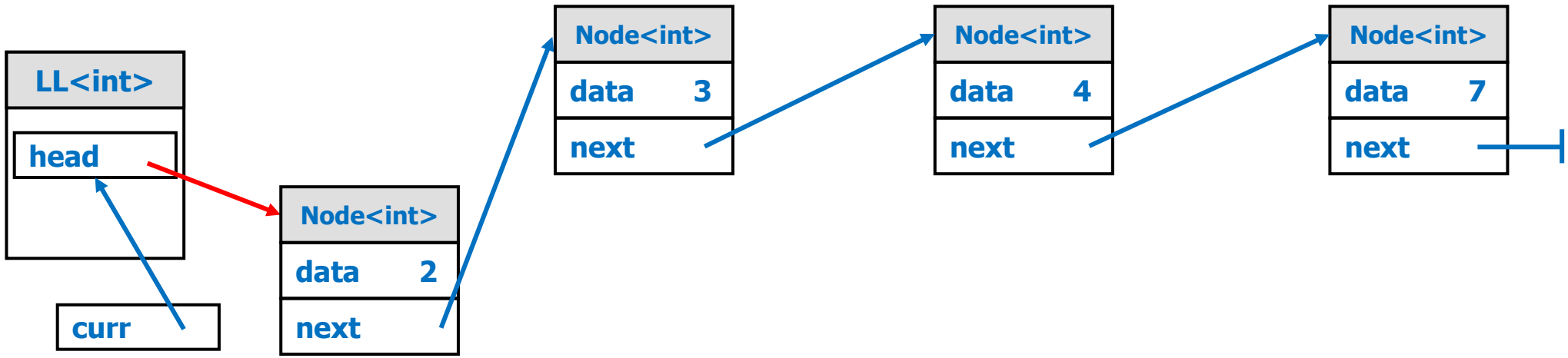
- Now revisit inserting 2
  - Start with curr at &head
  - Is this the "box" we want to change? Yes, because (*current)->data = 3 (3 > 2)

Pointer to a Pointer Approach



- Now revisit inserting 2
  - Make a new Node with data = 2, next = *current

Pointer to a Pointer Approach



- Now revisit inserting 2
  - Make a new Node with data = 2, next = *current
  - Set *current to that new node

Pointer to a Pointer: Generalized Algorithm

Steps to add a node (with dataToAdd) to a sorted linked list:

Start with `current` pointing at the box for `head`

As long as `(*current)->data < dataToAdd`

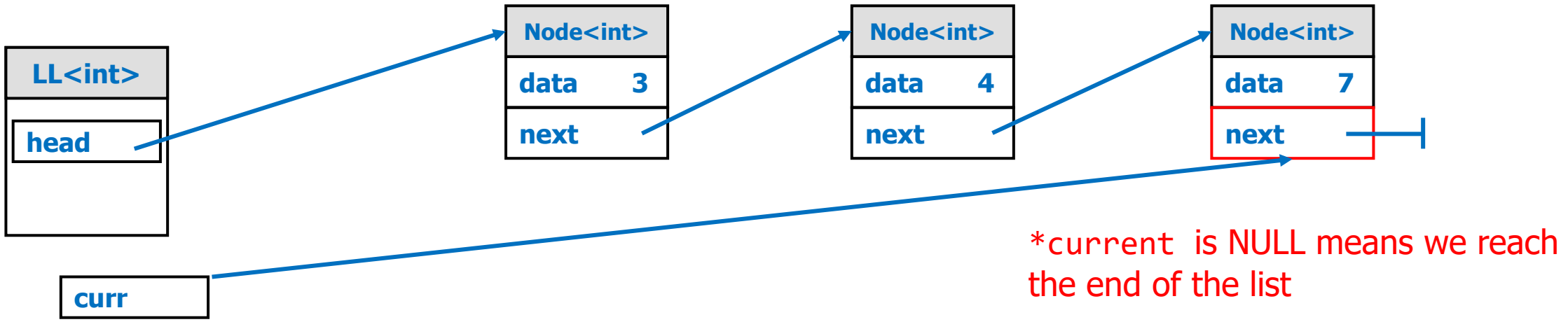
    Update current to point at the box for `(*current)->next`

Make a new Node with `data = dataToAdd, next = *current`

Set `*current` to that new node

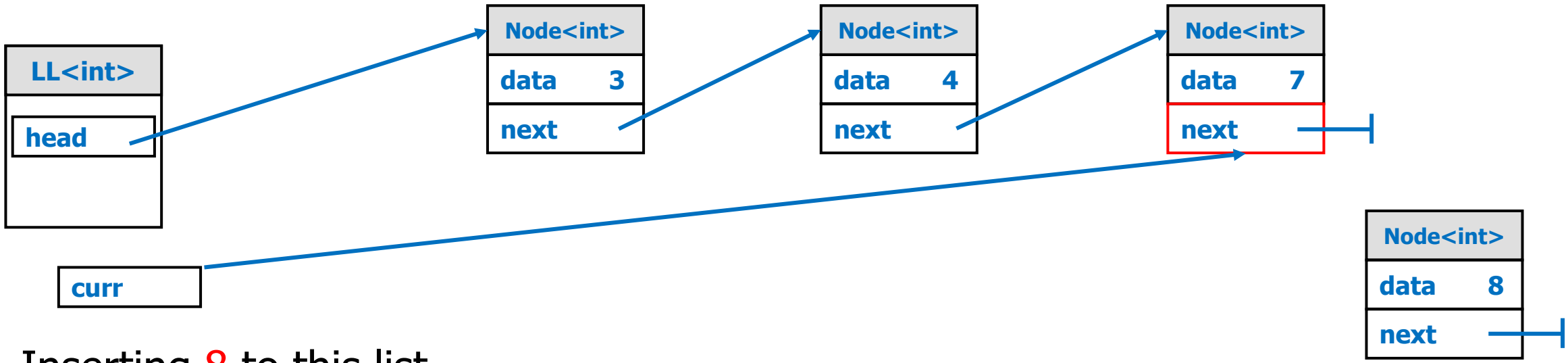
What if `*current` is NULL? When will that happen?

Pointer to a Pointer Approach



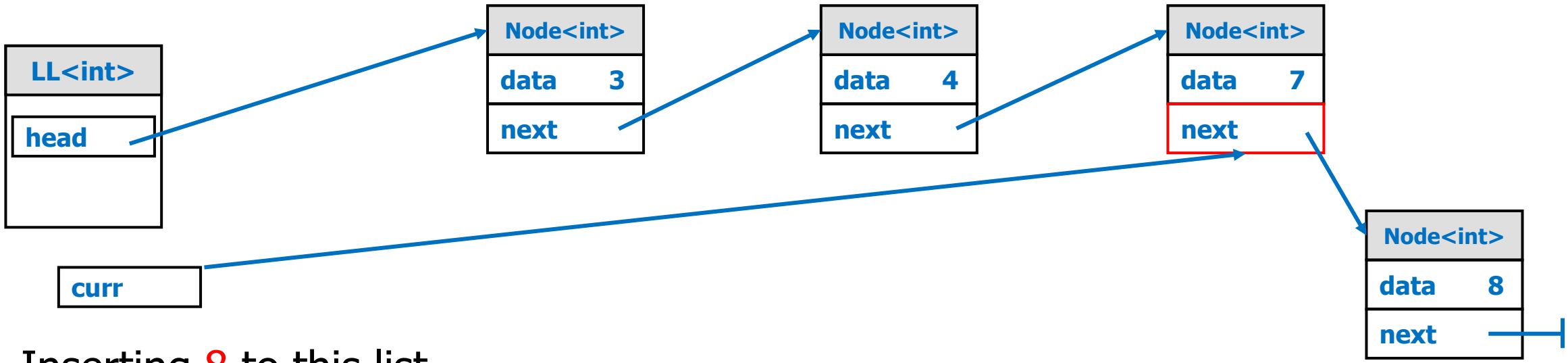
- Inserting 8 to this list
  - Is this the "box" we want to change? Yes
  - Same way …

Pointer to a Pointer Approach



- Inserting 8 to this list
  - Is this the "box" we want to change? Yes
  - Same way …
    - Make a new Node with data = 8, next = *current

Pointer to a Pointer Approach



- Inserting 8 to this list
  - Is this the "box" we want to change? Yes
  - Same way …
    - Make a new Node with data = 8, next = *current
    - Set *current to that new node

Pointer to a Pointer: Generalized Algorithm

Steps to add a node (with dataToAdd) to a sorted linked list:

Start with `current` pointing at the box for head

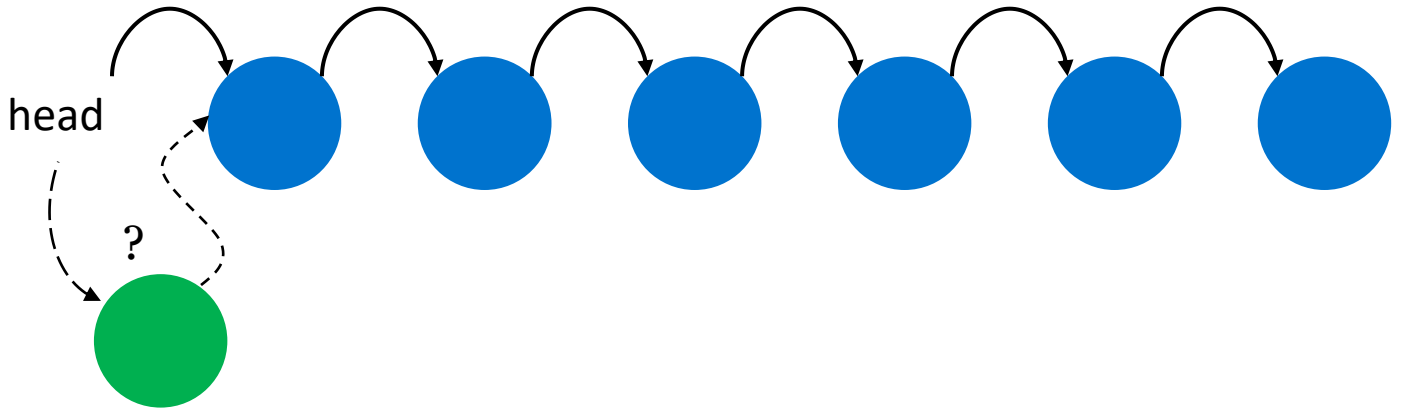
As long as `(*current)` is not NULL and `(*current)->data < dataToAdd`

    Update current to point at the box for `(*current)->next`

Make a new Node with `data = dataToAdd, next = *current`

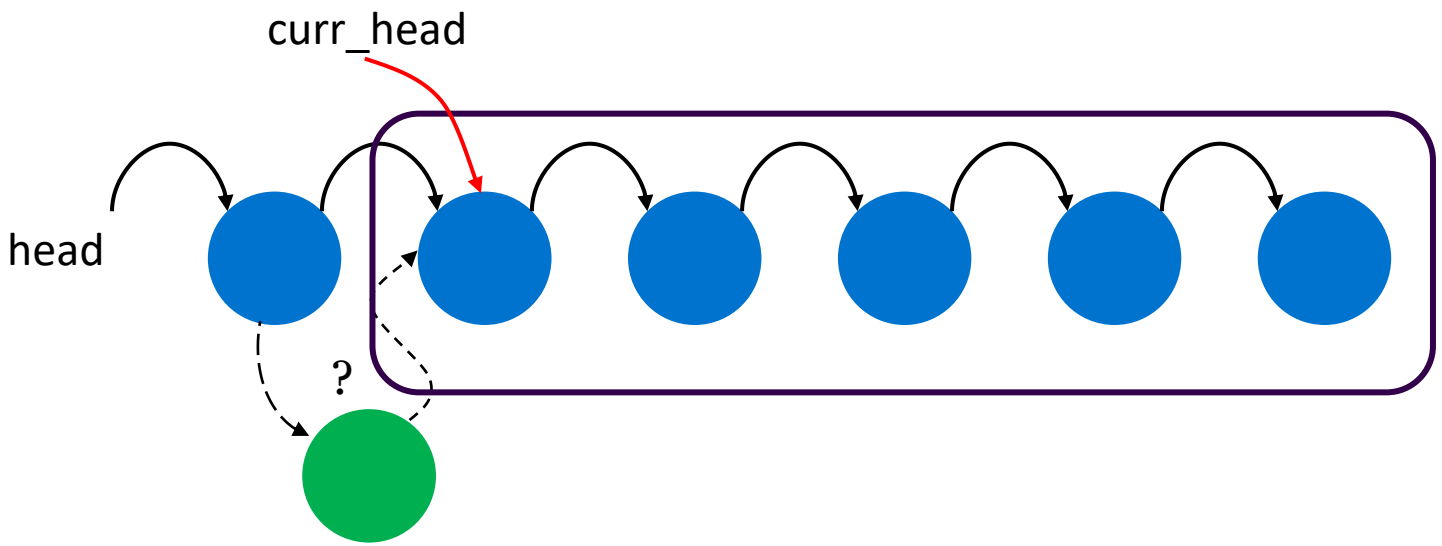
Set `*current` to that new node

Insert in Sorted Order: Recursive Approach



New node belong at the front of the list?

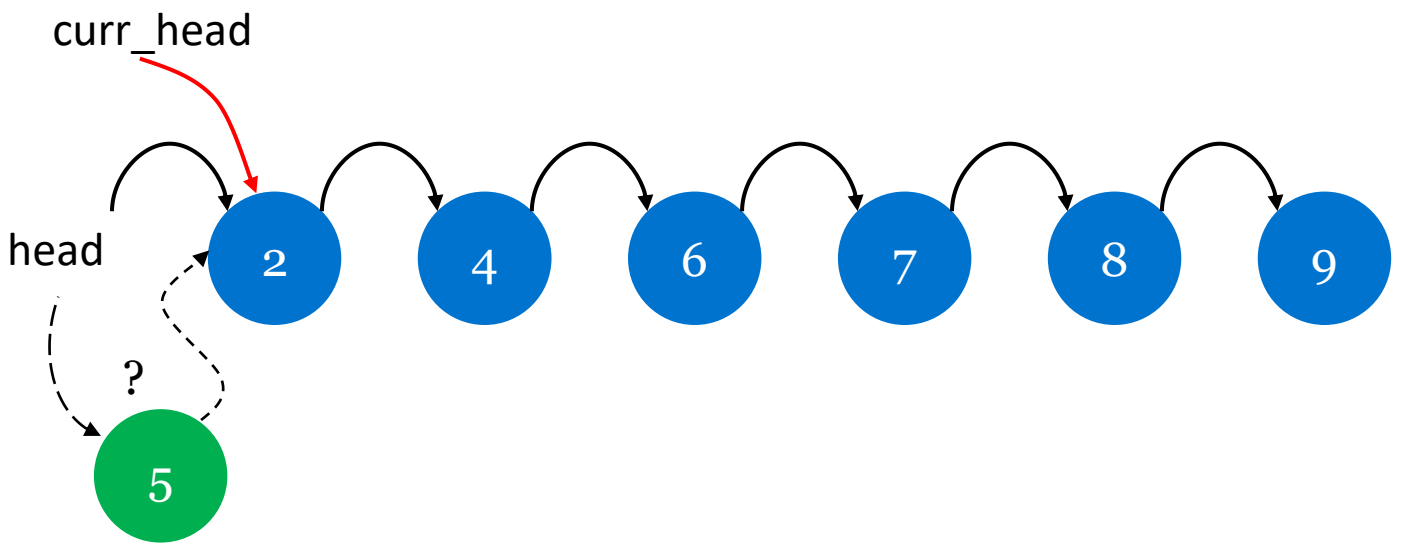
Insert in Sorted Order: Recursive Approach



If not, recursive call on a smaller linked list

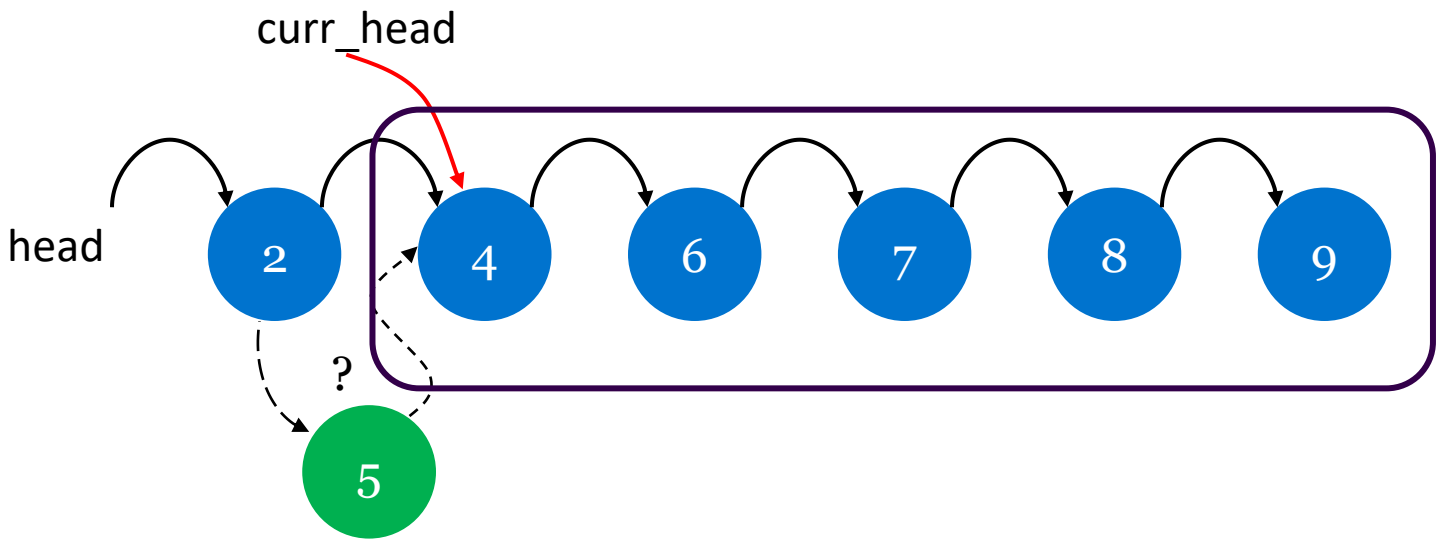
New node belong at the front of this list?

Recursive Approach Example



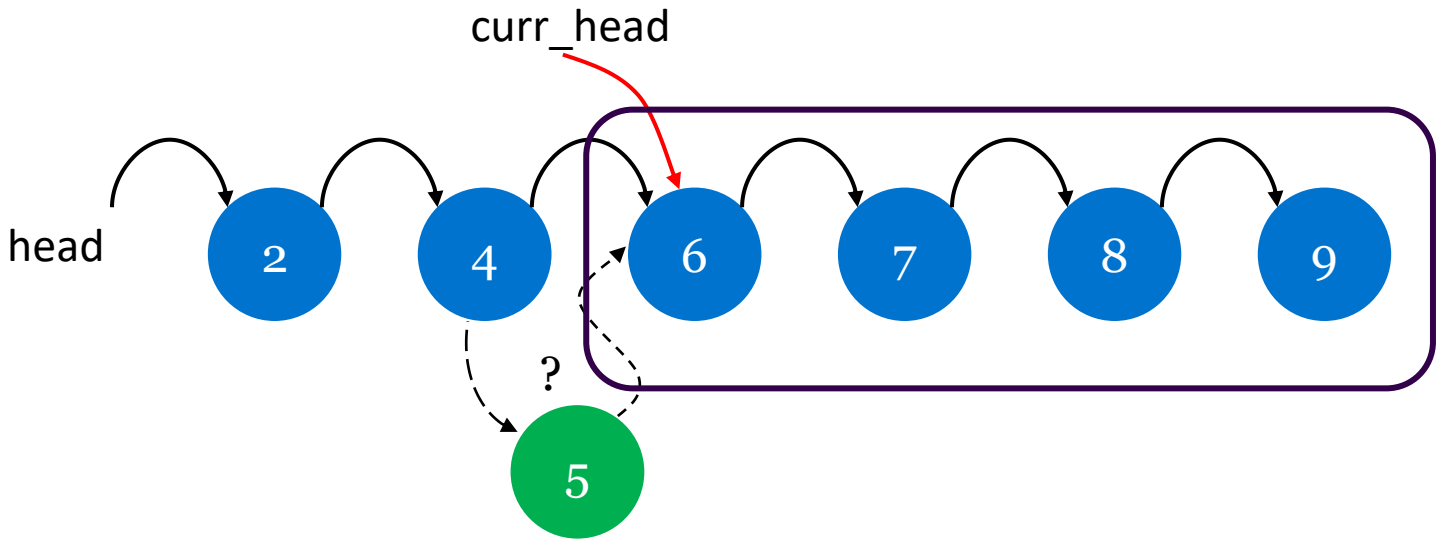
New node belong at the front of the list? No

Recursive Approach Example



New node belong at the front of the list? No

Recursive Approach Example



New node belong at the front of the list? Yes

Insert in Sorted Order: Recursive Approach

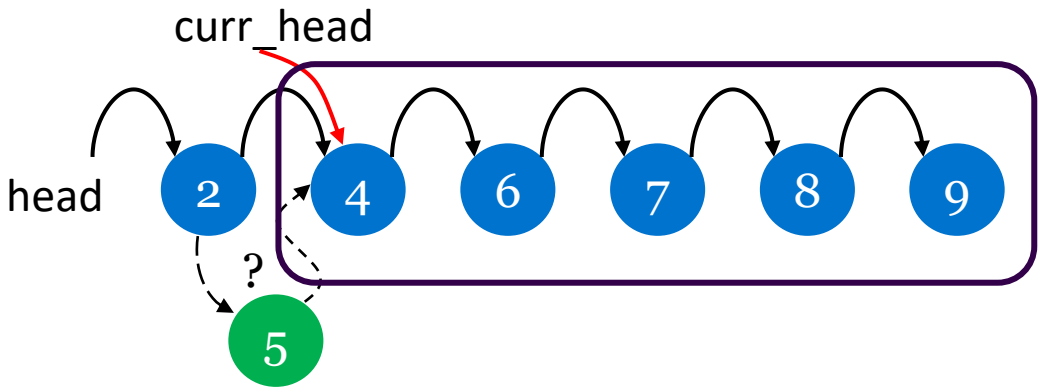
- Helper function: Node * **addSorted**(const T & data, Node * curr_head)
- Base cases
  - curr_head is NULL?
  - data < curr_head->data?

  } new Node goes at front return new Node(data, curr_head)

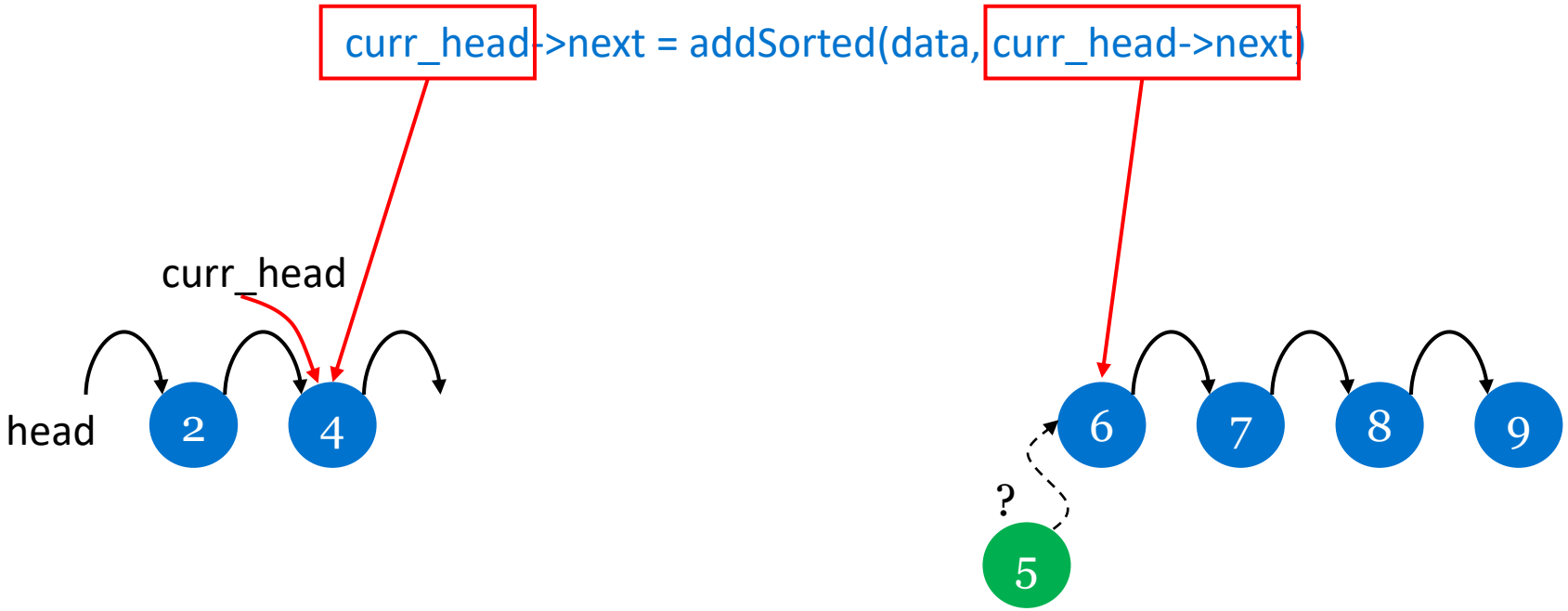
- Otherwise recurse with a smaller list, update curr_head->next
  - curr_head->next = addSorted(data, curr_head->next)

Insert in Sorted Order: Recursive Approach

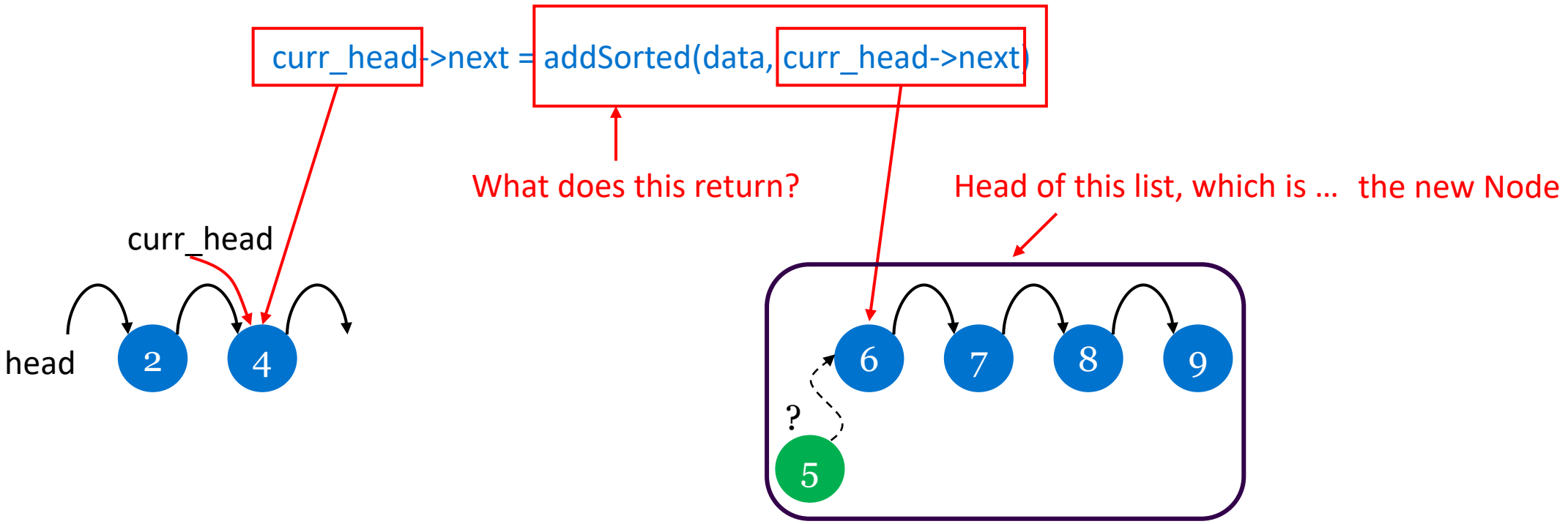
curr_head->next = addSorted(data, curr_head->next)



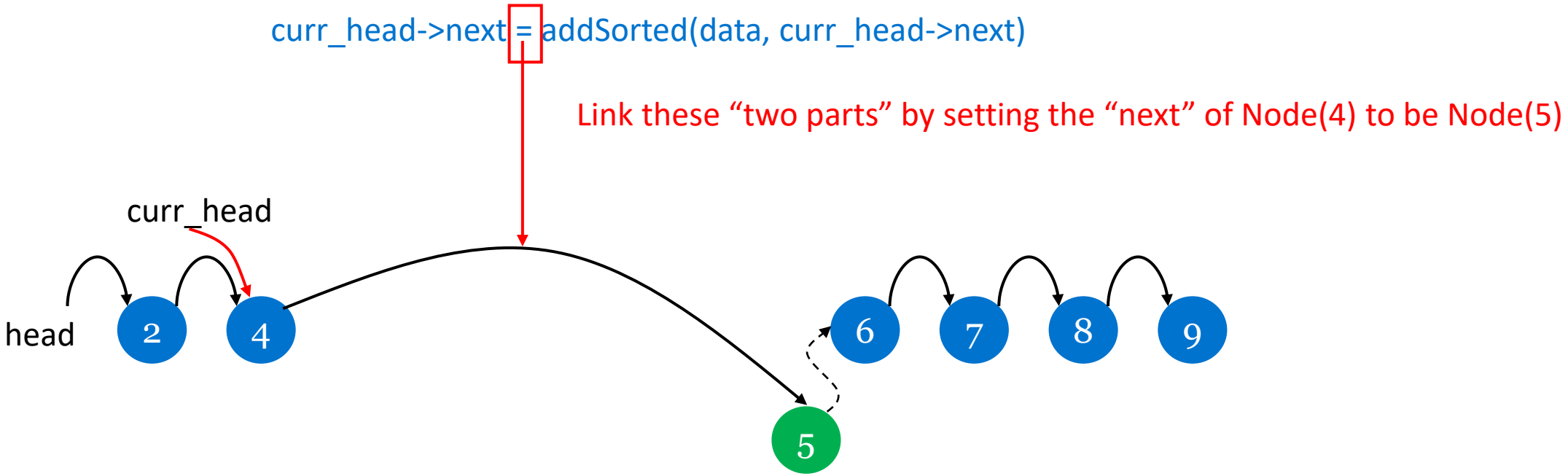
Insert in Sorted Order: Recursive Approach



Insert in Sorted Order: Recursive Approach



Insert in Sorted Order: Recursive Approach



Insert in Sorted Order: Recursive Approach

```
Node * addSorted(const T & data, Node * curr) {
      if (curr == NULL || data < curr->data) {
            return new Node(data, curr);
      }
      curr->next = addSorted(data, curr->next);
      return curr;
}

void addSorted(const T & data) {
      head = addSorted(data, head);
}
```

Linked List Class Declaration

```
template <typename T>
class LinkedList {
        class Node {
        public:
                T data;
                Node * next;
                Node (const T & d, Node * n) : data(d), next(n) {}
        };
        Node * head;
public:
        LinkedList() : head(NULL) {}
        …
};
```

Node class is an inner class of LinkedList

Linked List Class Declaration

```
template <typename T>
class LinkedList {
        class Node {
        public:
                T data;
                Node * next;
                Node (const T & d, Node * n) : data(d), next(n) {}
        };
        Node * head;
public:
        LinkedList() : head(NULL) {}
};
```

Node type is recursively defined! A node has a pointer to a Node!

Linked List Class Declaration

```
template <typename T>
class LinkedList {
        class Node {
        public:
                T data;
                Node  next;
                Node (const T & d, Node * n) : data(d), next(n) {}
        };
        Node * head;
public:
        LinkedList() : head(NULL) {}
};
```

Trying to make a Node have a Node (rather than a pointer) does not work

sizeof(Node) = sizeof(T) + sizeof(Node)…

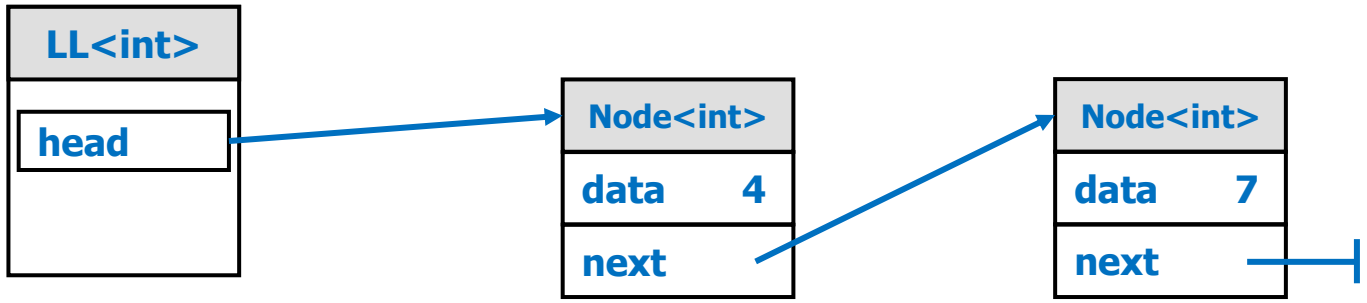
Linked List Class Declaration

```
template <typename T>
class LinkedList {
        class Node {
        public:
                T data;
                Node * next;
                Node (const T & d, Node * n) : data(d), next(n) {}
        };
        Node * head;
public:
        LinkedList() : head(NULL) {}
};
```

These are public, so anything that can access Node can access them …

but only LinkedList can access Node

Linked List Destruction



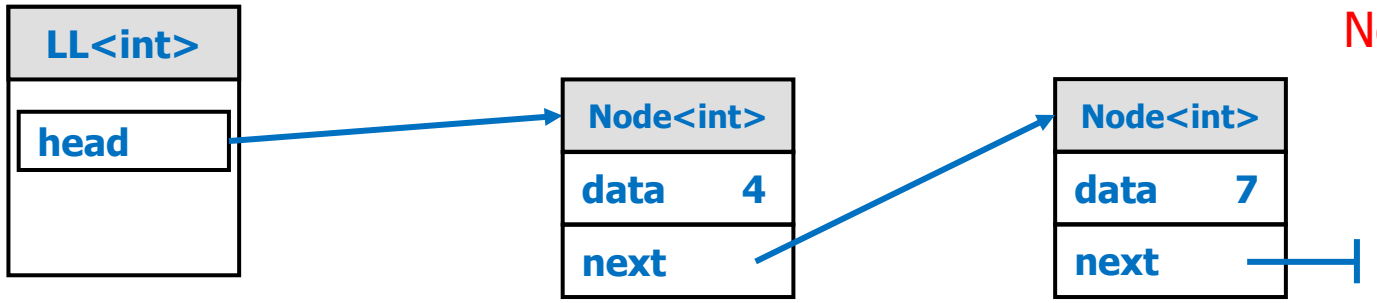
- LinkedList news nodes, so it is responsible for deleting them
  - Node should not delete anything (it does not create it)
  - We would want to delete Nodes when:
    - We remove things from the list
    - Or we destruct the list

Bad Destructor

```
template <typename T>
class LinkedList {
        class Node {…};
        Node * head;
public:
        …
        ~LinkedList()  {
                while (head != NULL) {
                        delete head;
                        head = head -> next;
                }
        }
};
```

A bad destructor: think about why …

Linked List Destruction



No way to reach this node anymore!

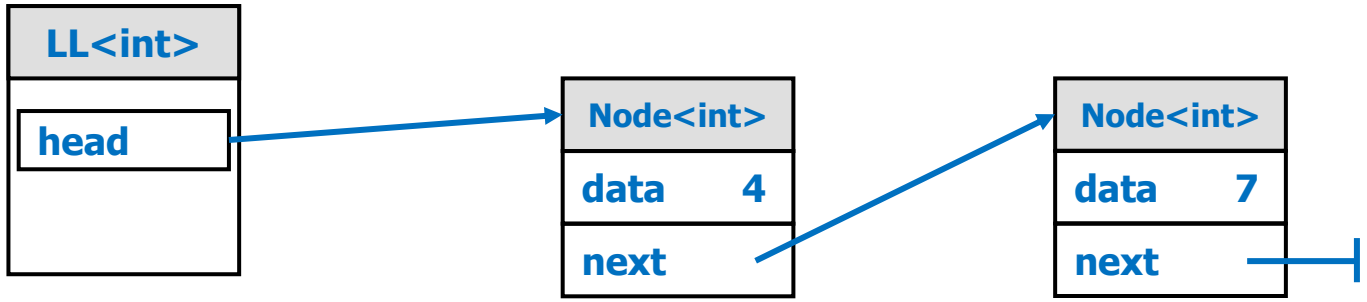
```
~LinkedList()  {
        while (head != NULL) {
                delete head;
                head = head -> next;
        }
}
```

Use memory after it has been freed!

Correctly Destructing LinkedList

- We can do this correctly a couple of ways
  - Keep another pointer to remember where to go next
    - Similar to the temp pointer we saw earlier
  - Use recursion
    - Write a "helper" function: destroy(Node * n)
      - Destroys the list starting at n, recursively
      - Base case: NULL

Linked List Destruction: Recursion




```
~LinkedList()  {
        destroy(head);
}
void destroy(Node * n) {
        if (n != null) {
                destroy(n->next);
                delete n;
        }
}
```

Won't start deleting any node until … it reaches the last node

Essentially delete backwards

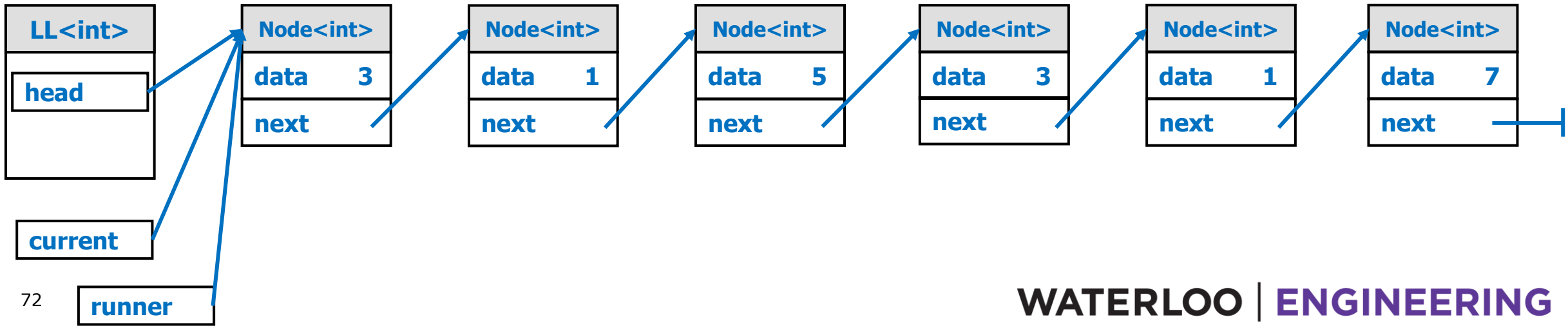
Linked List Coding Interview Question

- Q: Write code to remove duplicates from a linked list

- Step 1: Ask for clarifications
  - What kind of linked list are we assuming?
  - Is the linked list sorted?
  - What type of data are we assuming the linked list is holding?
- Step 2: If still not clear, use an example to confirm
  - "Do you mean with an input like X, the output should be Y ?"
- Step 3: Develop an algorithm
- Step 4: Translate your algorithm into code (with your most comfortable PL)

Remove Duplicates in Linked List

- Assume an unsorted singly-linked list of integers
- Example: Input = 3 -> 1 -> 5 -> 3 -> 1 -> 7 ; output = 3 -> 1-> 5 -> 7
- Algorithm: similar to our "countDuplicates" example
  - Use a "current" pointer to iterate through the linked list
  - Use another "runner" pointer to check all subsequent nodes for duplicates
    - current -> data == runner -> next -> data ?



Follow Up Questions

- What's the Big-O of our previous algorithm?
  - Same as "countDuplicates" : O(n^2)

- Can we do better?
  - Recall in step 1 we asked if the linked list is sorted …
  - If it is sorted, we can easily do it in O(n)
  - But still depends on the Big-O of the sorting algorithm
  - Later we will learn a few sorting algorithms that has O(nlogn)

Follow Up Questions

- Can we do even better (than O(nlogn))?
  - Think about what is really good at detecting duplicates …
    - Hint: one of the ADTs we saw earlier … Set!
  - Algorithm with Set
    - Iterate through the linked list, for each node, check if the set contains the data
      - No → First-time encounter, add to the set
      - Yes → duplicate found, remove it
    - Big-O depends on efficiency of `set.contain()` operation
    - Later we will learn about hash table and how we can use that to implement set and achieve amortized O(1) for `set.contain()` operation

Linked List Complexity (Compared with Array)

Data Structure	Search	Access			Add			Remove		
		Front	Back	Index	Front	Back	Index	Front	Back	Index
Array	O(n)	O(1)	O(1)	O(1)	O(n)	O(1)*	O(n)	O(n)	O(1)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

Require copying O(n) items

O(1) with Doubly Linked List + tail pointer

Occasionally O(n) when array is full

O(1) with tail pointer

Takeaway: LinkedList trade the ability to easily access a particular index for the ability to modify the structure without copying elements around

Wrap Up

- In this lecture we talked about
  - Variations of linked lists
  - Basic operations: add to front/back; remove from front/back, search
  - Example interview question on linked list
  - Complexity comparison with array
- My advice for dealing with linked list problems:
  - Draw pictures!
- Next up
  - Implementations of some ADTs (e.g., stacks & queues)
  - And their complexities

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 3.1 – 3.3
- Introduction to Algorithms: Chapter 10.2

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)