CSc 120
Introduction to Computer Programming II

Linked Lists

Name Confusion

Before we start...

Python calls this sort of data structure a **`list.`**
Nearly all other programming languages call this an **array.**

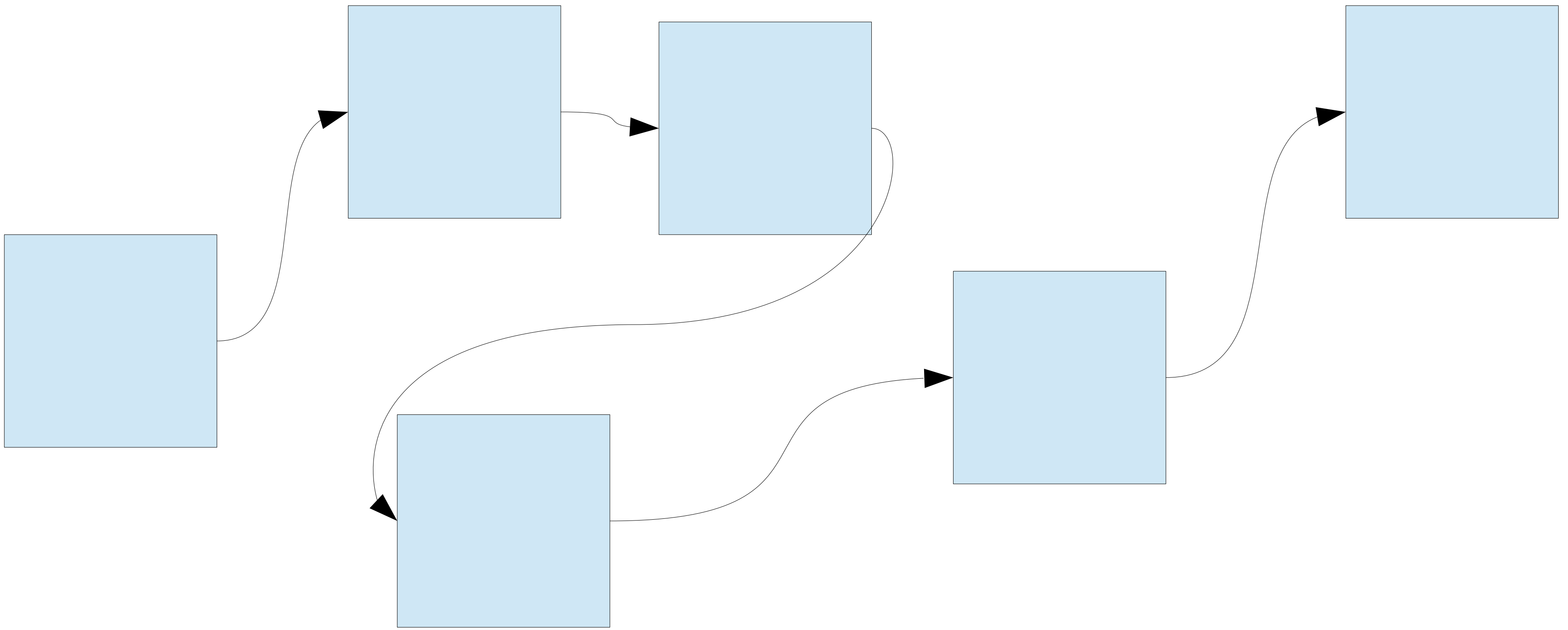
1	-17	23	0	1	2	2	10

NOTE:
I often mark key terminology in **red.**

Name Confusion

Before we start...

This slide deck deals with **linked lists.** They store data in a different way than a Python list.



Challenge

Group Exercise (setup):

Draw a picture of an array, using our standard drawing style. (An array is a bunch of boxes in a line.)

Put three random integers in the array, sorted. **(Use pencil.)**

-17	1	23

Choose any numbers that you feel like.

Challenge

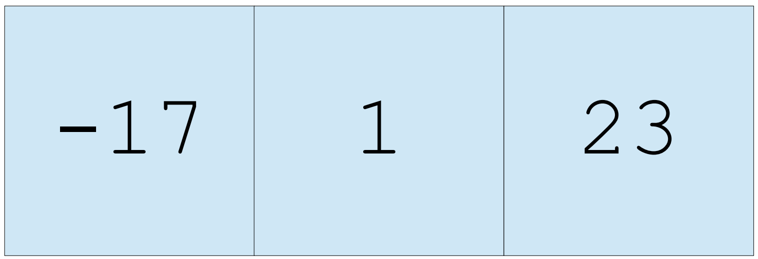
Group Exercise (insert):

Now, choose 7 other numbers (not sorted). Add them one at a time to the array. Keep the array sorted at all times.

How many steps did this take? How many times did you have to erase something and change a value?

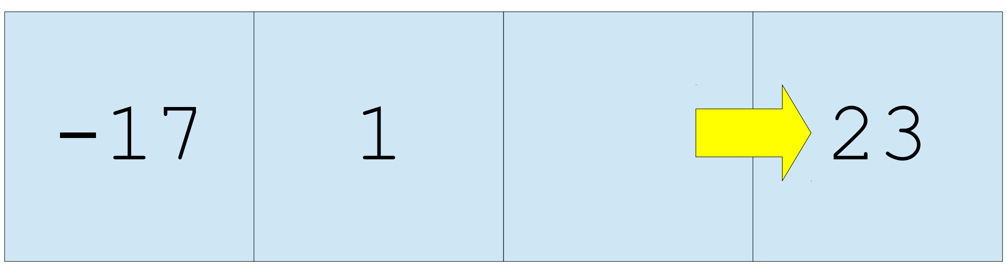
(live demo)

Inserting into a Sorted Array

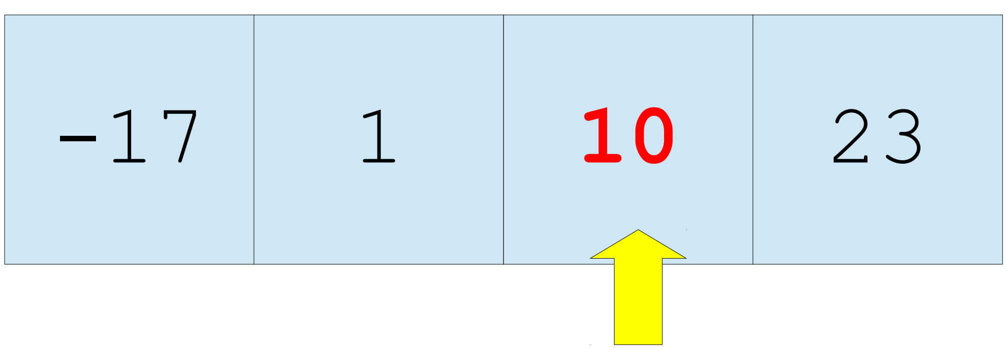


want to insert: **10**

need to make some space...



now it's possible to insert

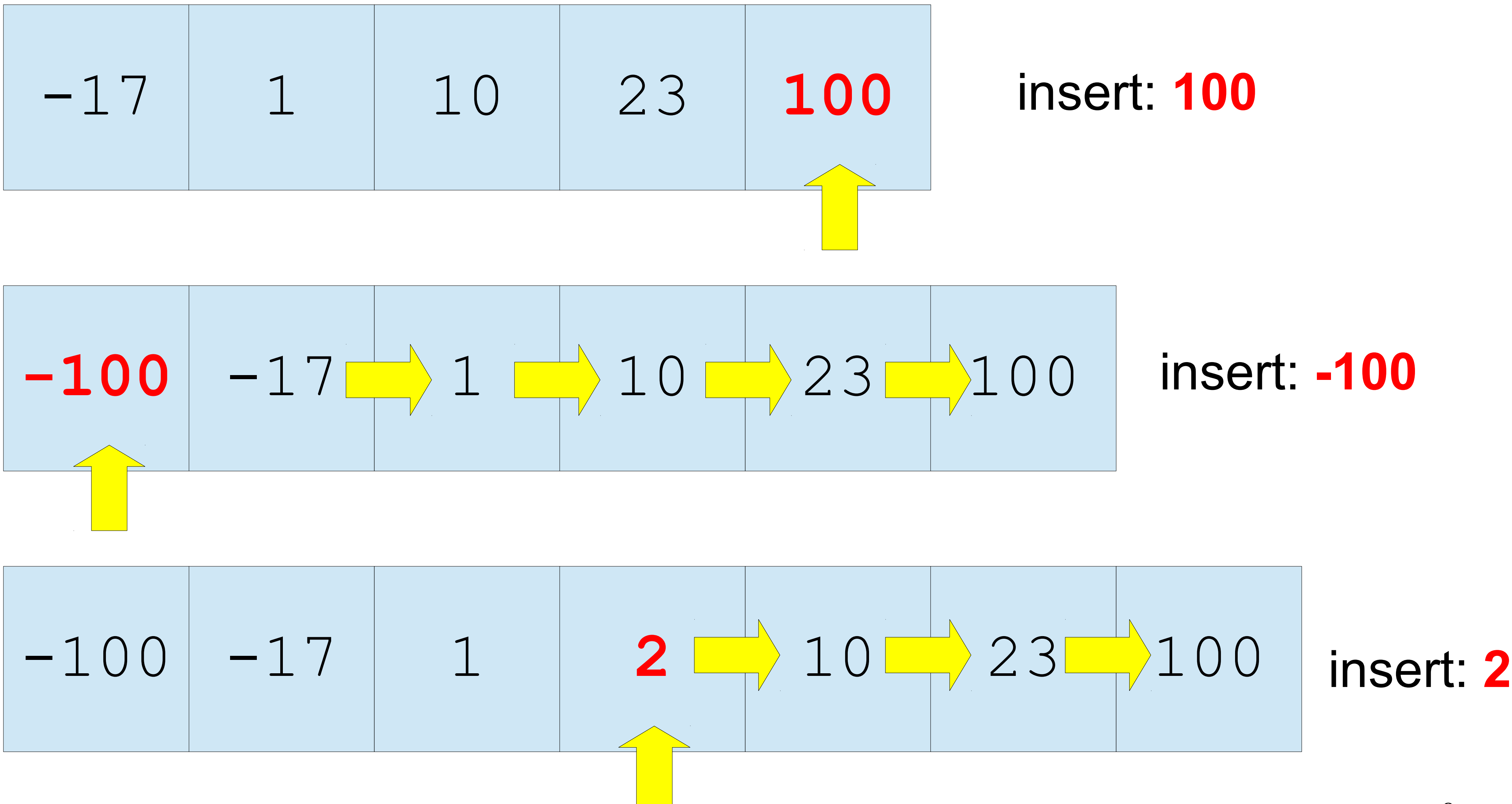


Inserting into a Sorted Array

To insert into a sorted array:

1) Search for the correct position
2) Shift values to make space
3) Insert into the empty hole

Inserting into a Sorted Array



Inserting into a Sorted Array

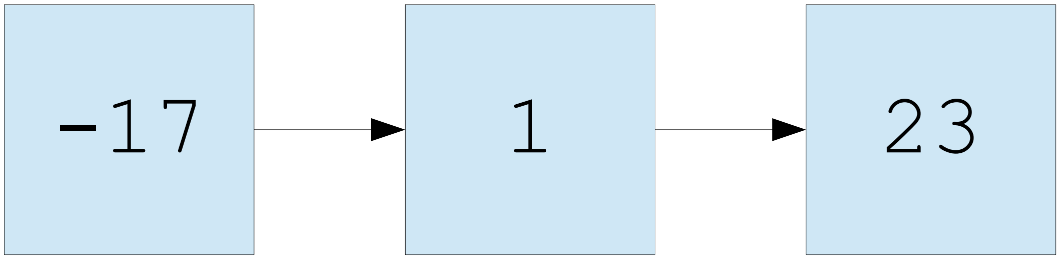
- When inserting into a sorted array, we (on average) move half the elements.
  - We call this cost `O(n)` – meaning that the cost is proportional to the number of elements
  - Inserting many items takes $O(n^2)$ cost

- Wouldn't it be nice to have a data structure which makes it cheaper?

References to the Rescue

Insight:

- What if the values were linked with **references,** instead of being next to each other in an array?



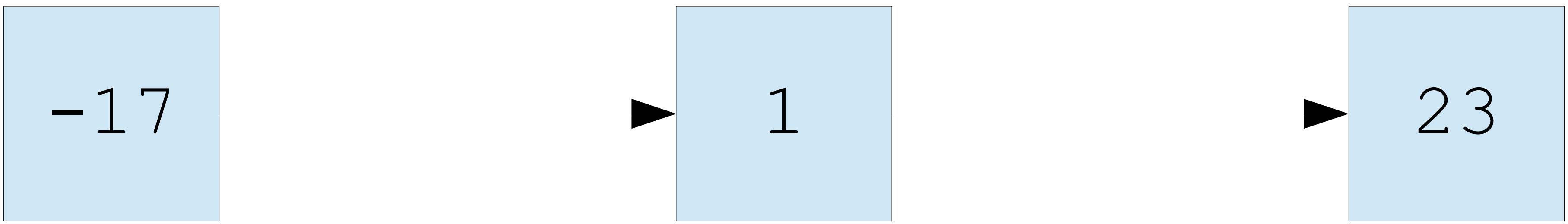
Our First Linked List

Group Exercise:

Re-run the insertion exercise, but now:

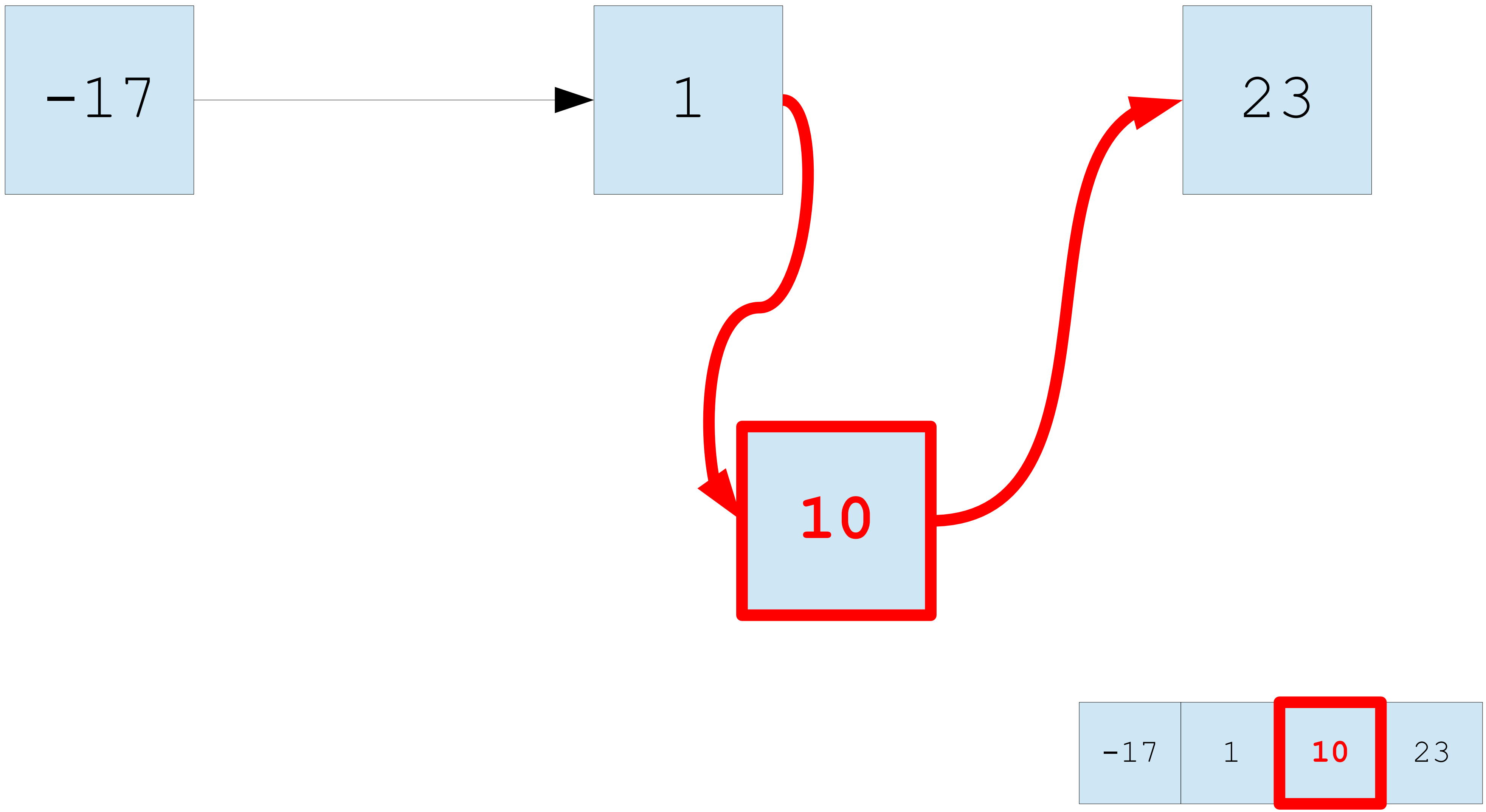
- Put each value inside its own box
- *Never* move a value
- Instead, only change references!

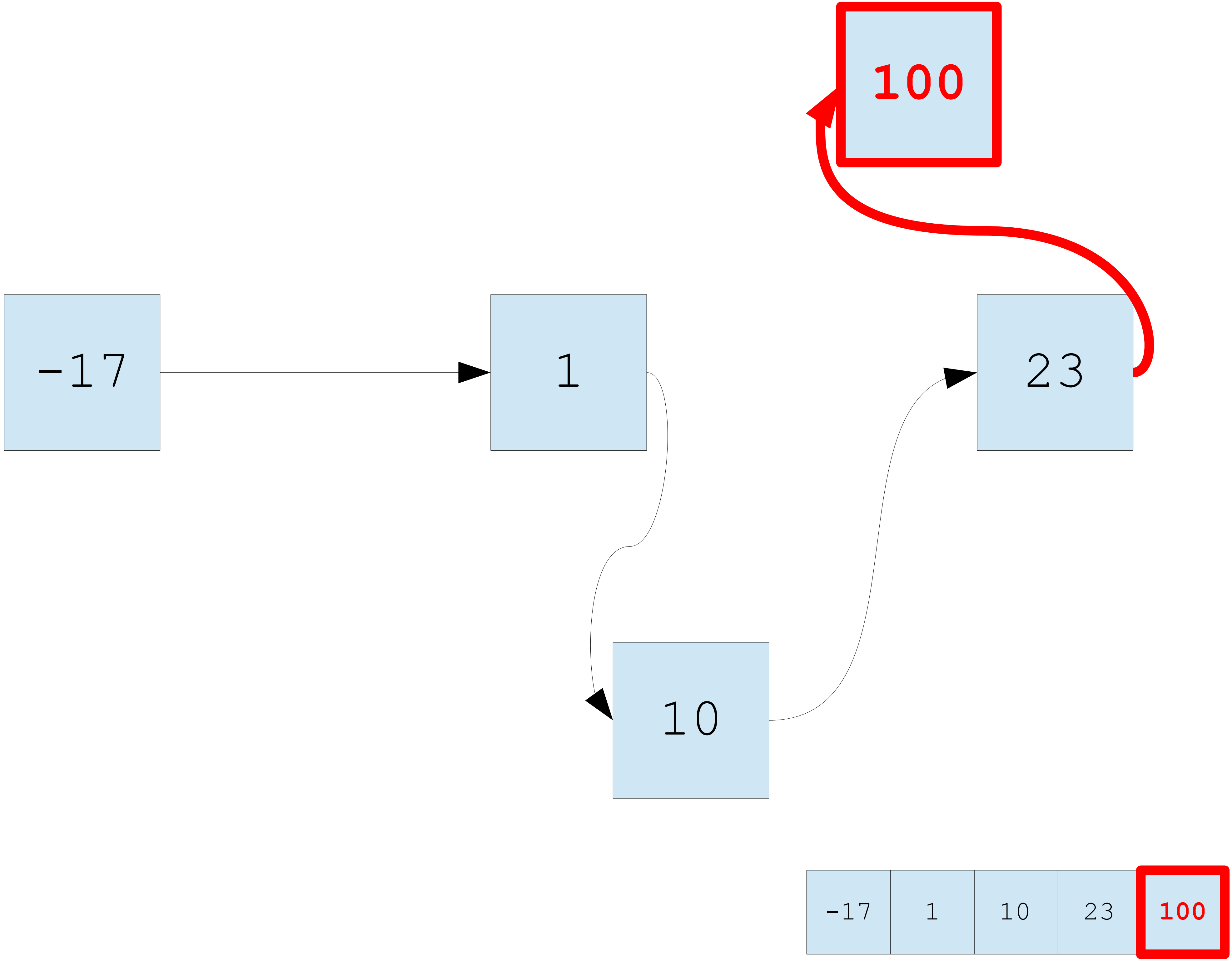
(live demo)

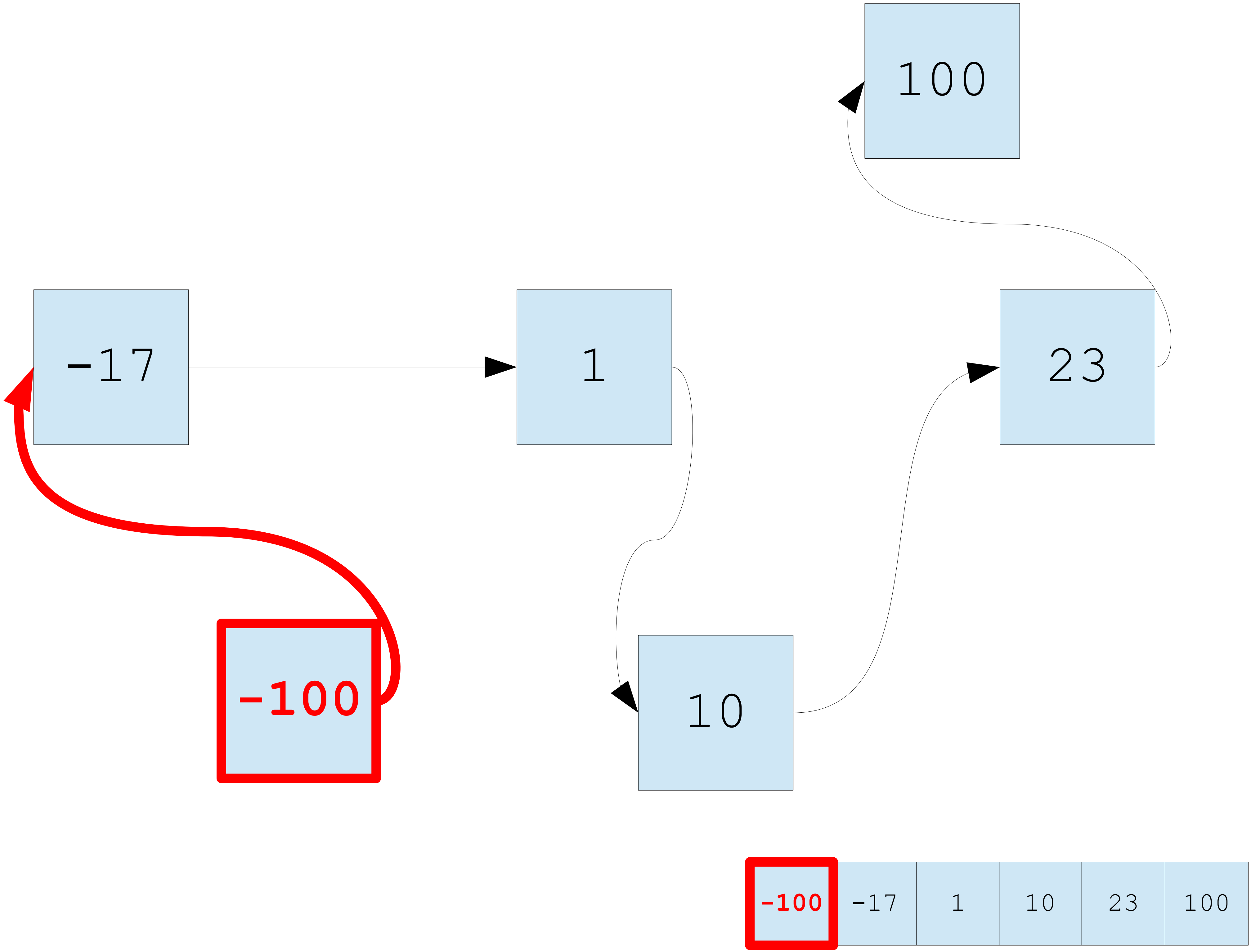


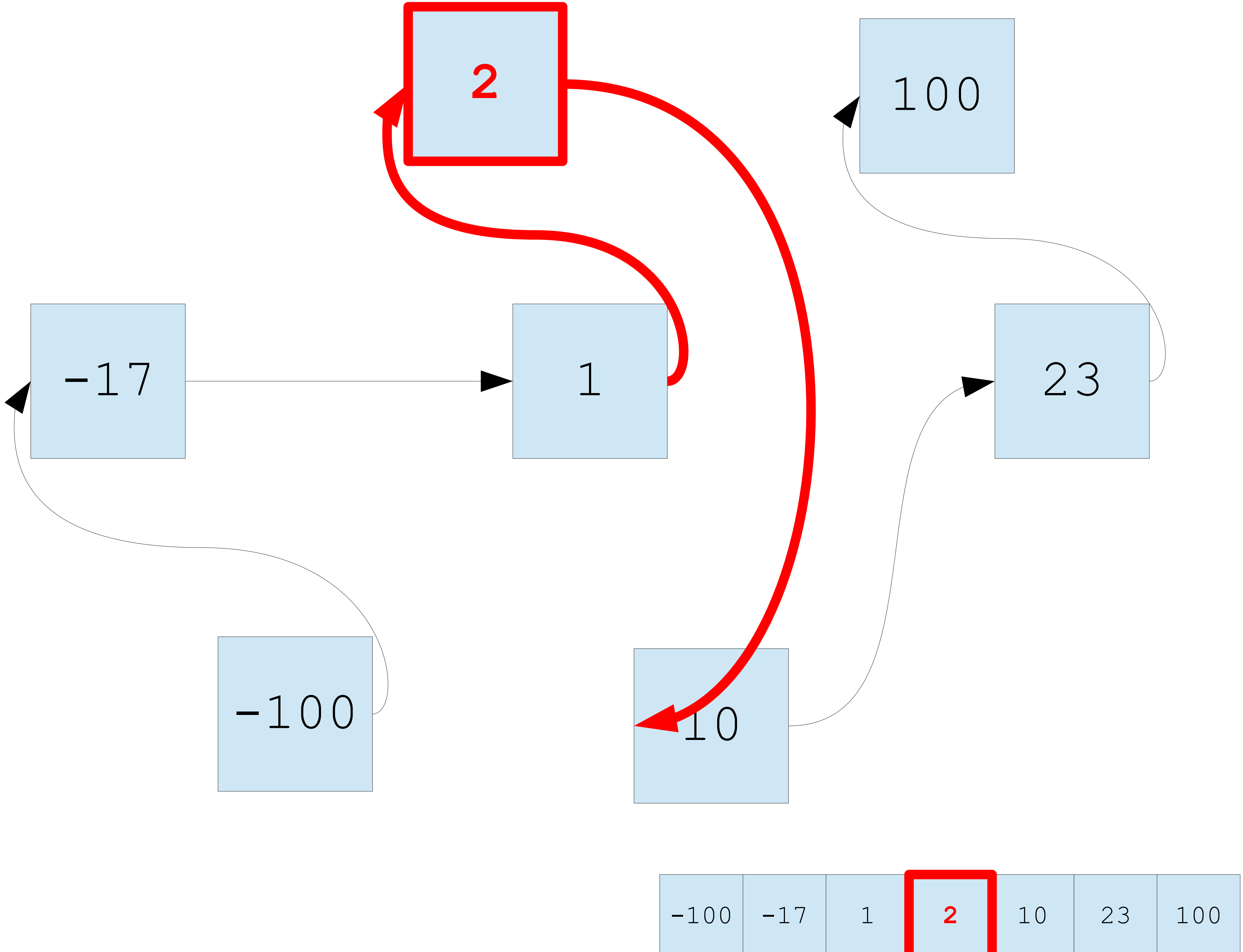
The same,
as an array

-17	1	23


-17
1
23
10
-17
1
10
23

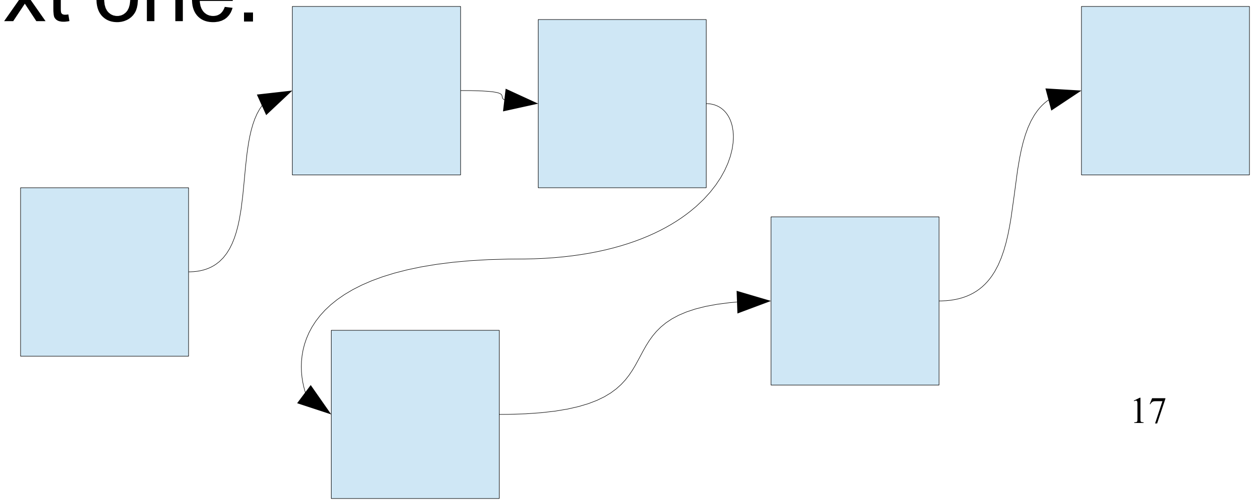

100
-17
1
23
10
-17
1
10
23
100


100
-17
1
23
-100
10
-100
-17
1
10
23
100


2
100
-17
1
23
-100
10
-100
-17
1
2
10
23
100

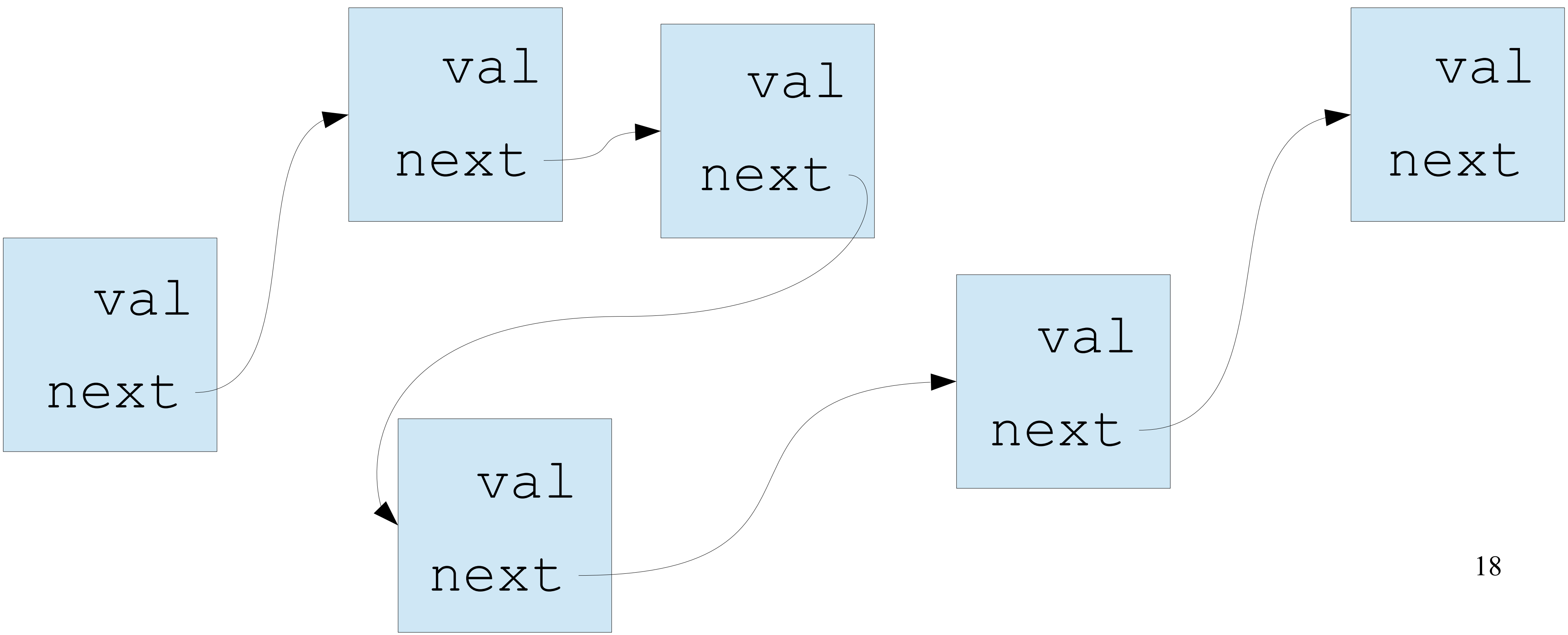
Linked Lists

- We just simulated a **linked list.**
- A **linked list** keeps each value in a separate container, called a **node;** once created, this container never moves.
- But each node has a link, which gives the location of the next one.



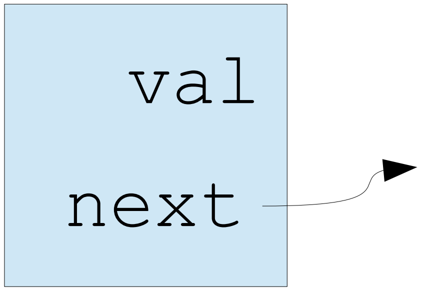
Linked Lists

- Each **node** of a list holds one data item, and also has a `next` pointer.



Linked Lists

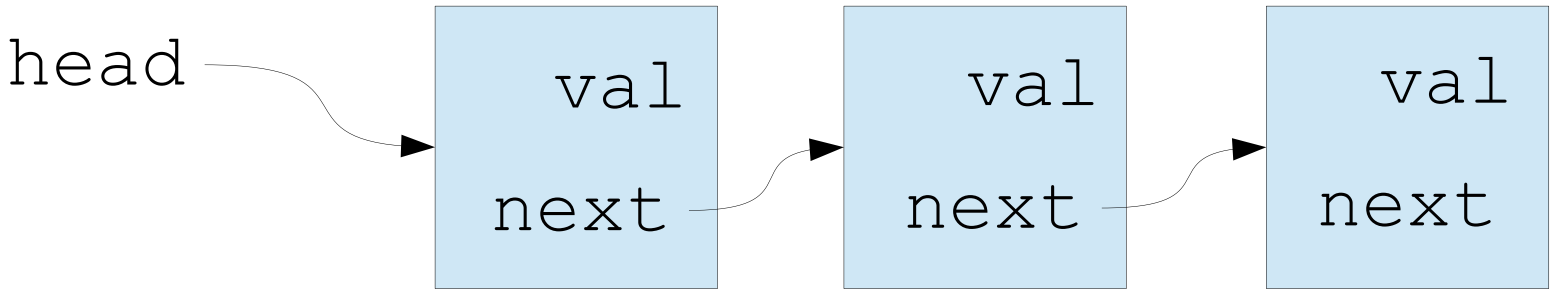
- Soon, we'll show you how to implement this **object** with Python **classes.**



- But for now, let's just focus on the concepts.
  - That's portable to *any* language!

`head` Pointer

- Every list needs a starting point
- Use a variable *outside* the list, which points to the first node.
  - Traditionally, named `head`



Empty Lists

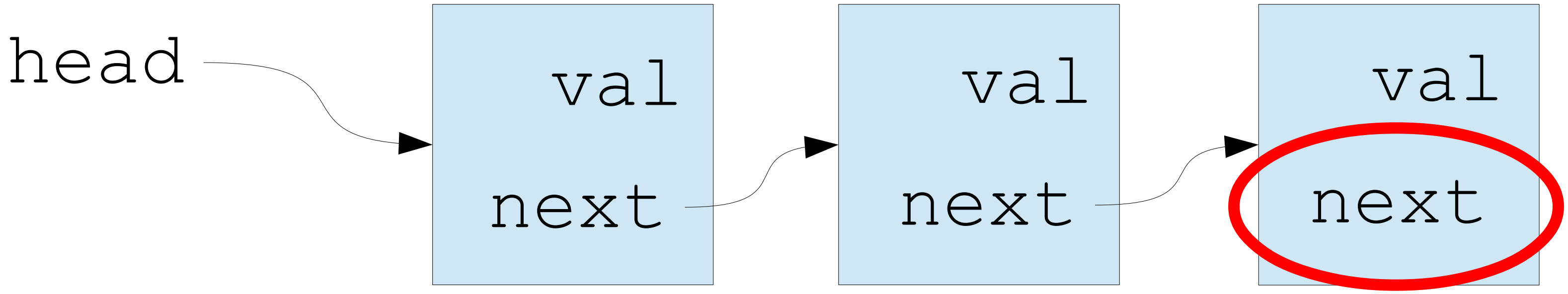
- If a list has no nodes at all, then the head pointer doesn't point to anything.
- In Python, we use the keyword `None` to represent “points at nothing.”

Sample Code

```
head = None
```

End of a List

- We also use `None` to represent the end of the list, since "there is no next node."



This reference is `None`.

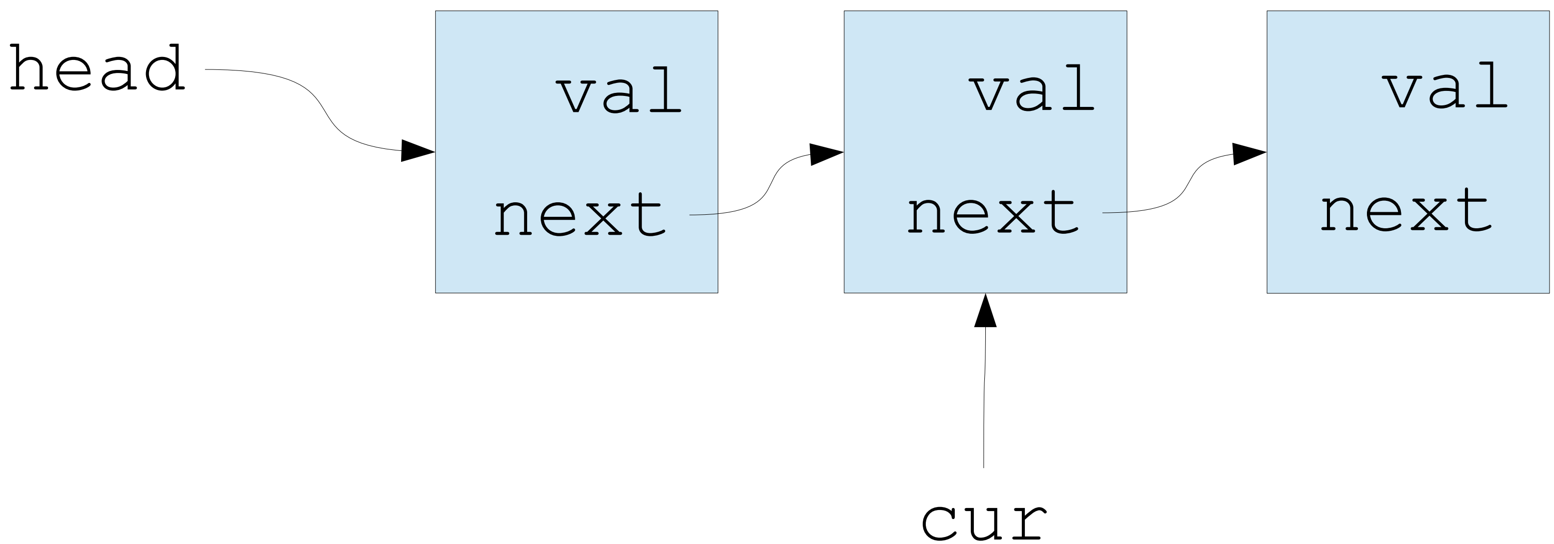
How to draw `None`?

Sometimes, a bare arrow. Sometimes, no arrow.

Use your best judgment.

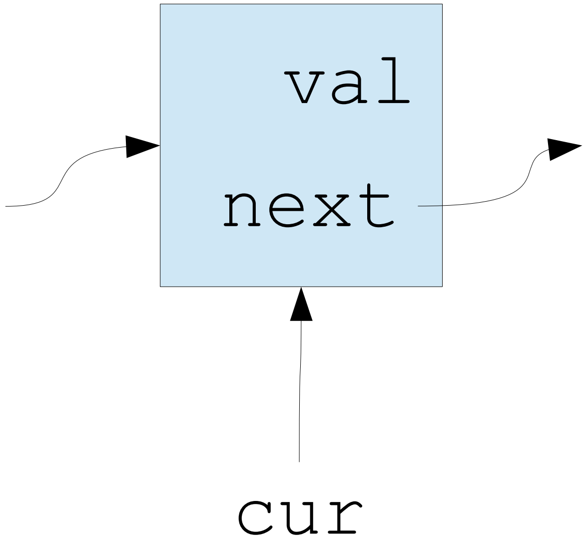
Traversing a List

- When iterating through a list, we need to have a variable (different than `head`), which points to the "current" node.



Traversing a List

- Using `cur`, you can read the value of the node that you're looking at, or move to the next one.



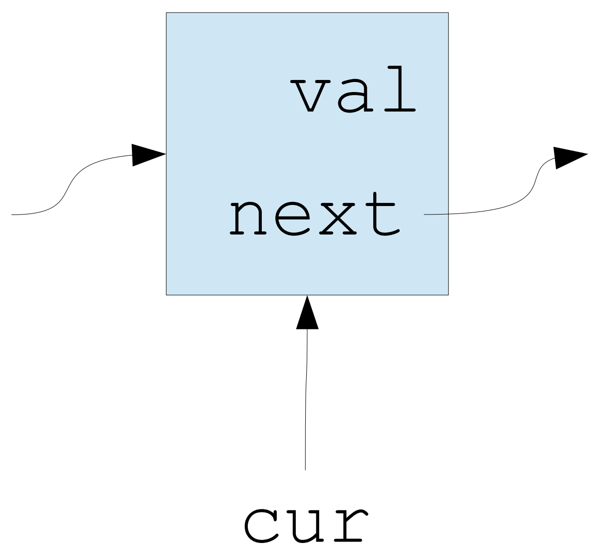
Sample Code

`cur.val`
(the value)

`cur.next`
(the next node)

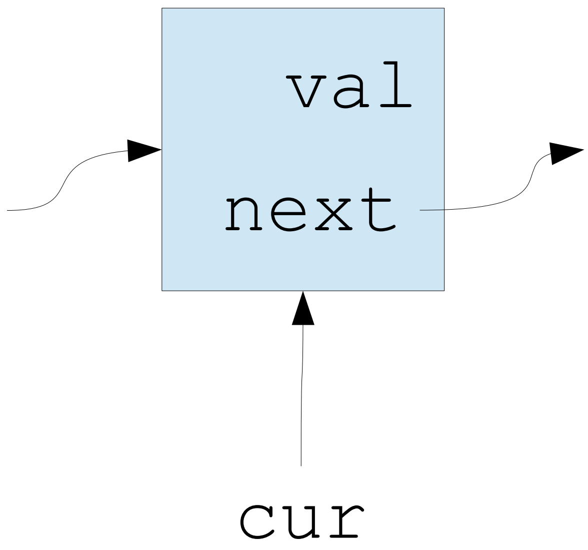
Traversing a List

- Using `cur`, it is *impossible* to move to the previous node.
- Why is this?



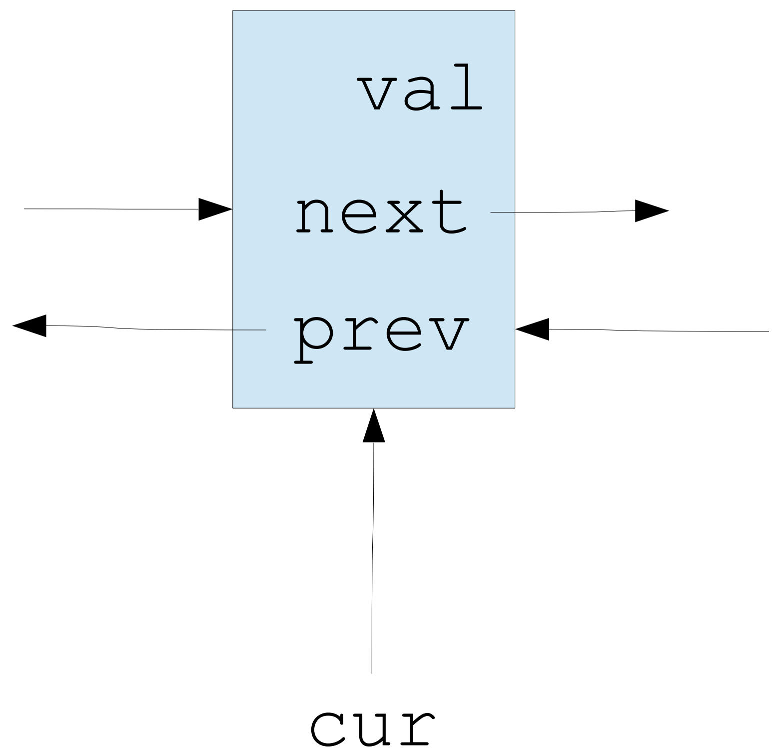
Traversing a List

- Remember, you can only traverse a **reference** in the "forward" direction. Moving backward isn't possible.



Traversing a List

- Later, we'll study **doubly-linked lists,** which solve this problem.

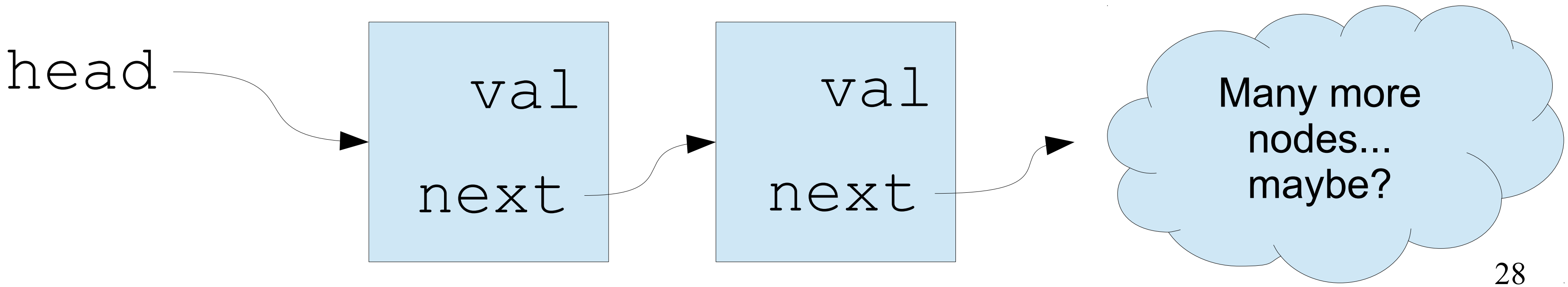


Traversing a List

Group Exercise:

Suppose we have a linked list, starting at `head`. (It might be empty.)

Write some Python code which will iterate through the list, and print out every value.



Traversing a List

```
cur = head

while cur is not None:
    print(cur.val)
    cur = cur.next
```

Easy Python Mistake:
Using `==` instead of "`is not`" when comparing something against `None`.

Traversing a List

```
cur = head

while cur is not None:
    print(cur.val)
    cur = cur.next
```

Group Exercise:

Draw a small linked list (4 nodes or more). Then use that drawing to simulate this code. **Make sure to keep track of the** `cur` **variable.**

Traversing a List

```
cur = head

while cur is not None:
    print(cur.val)
    cur = cur.next
```

Class Discussion:

How does this code handle the case of an empty list?

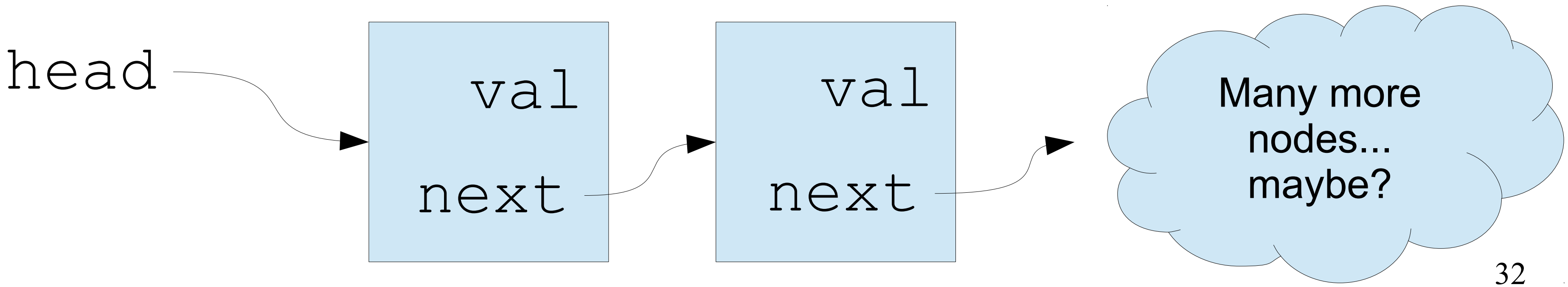
Does it have a bug? If so, what?
If it works, why does it work?

Traversing a List

Group Exercise:

Rewrite the previous code, but now do it as a *function*. What parameter(s) should it use?

Return the **length** of the list (that is, the number of nodes it contains).



Traversing a List

```
def print_list(head):
    cur = head
    count = 0

    while cur is not None:
        count += 1
        print(cur.val)
        cur = cur.next

    return count
```

Traversing a List

Group Exercise:

Redo the previous exercise, but in a strange way:

- If the list is empty:
  - Do all the work in your new function
- If the list is **not** empty:
  - Handle **first node** in your new function
  - Handle the rest of the list in your old one

Traversing a List

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```

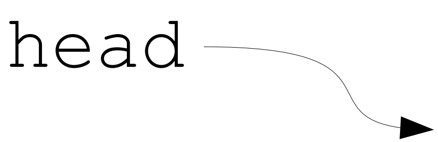
Group Discussion:

Discuss in your group:

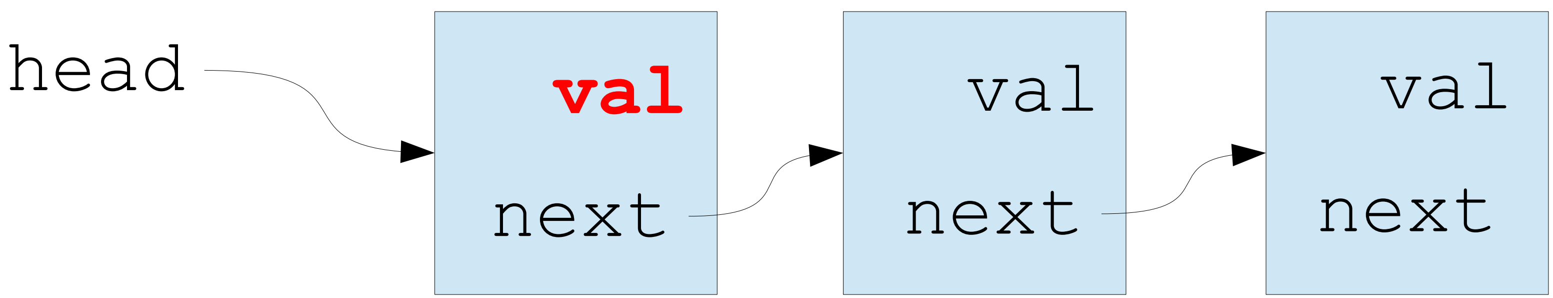
- What happens if the list is empty?
- What happens if the list has exactly 3 nodes?
  - What does `print_list()` return?
  - What does `print_list_2()` return?
- What happens if the list has exact 1 node?

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```

If the list is empty, our new function does all the work.



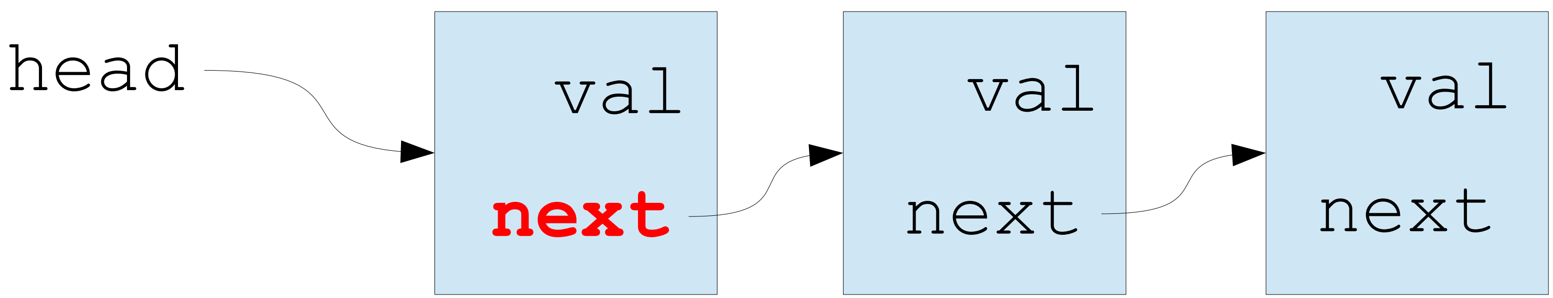
```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```



If the list has 3 nodes:

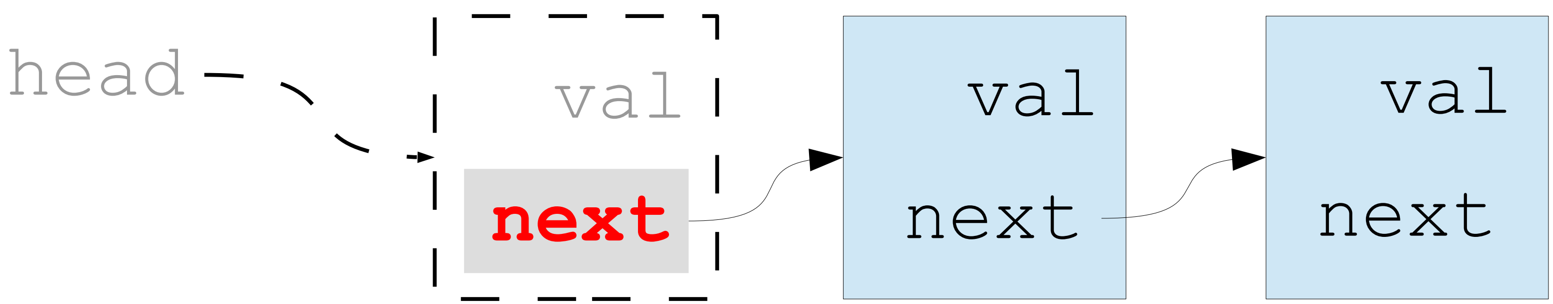
- We print the first value

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```



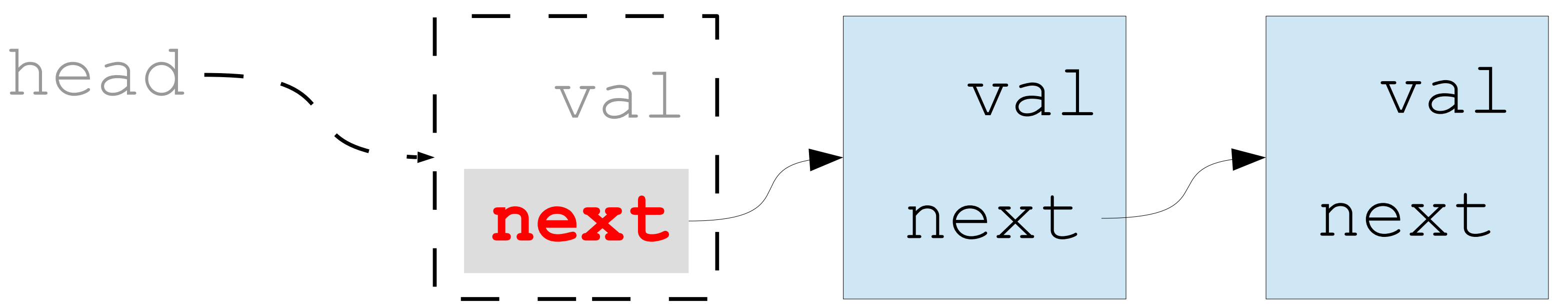
- We call `print_list()` and pass it the pointer to the 2nd node.

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```



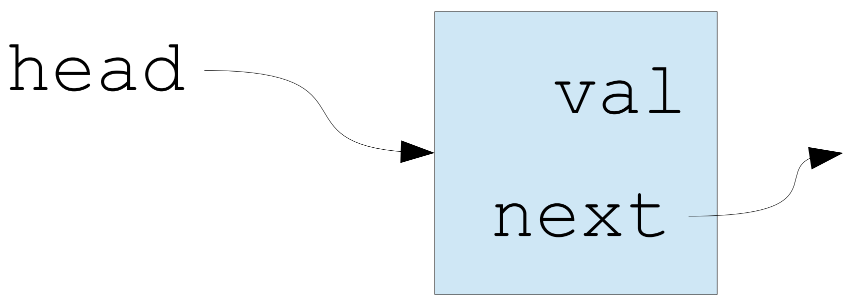
- `print_list()` works fine, because it thinks of its parameter as the head of a list.

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```



- `print_list()` will return 2, and we will return 3.

```python
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```



If the list has 1 node:

- We print the first value
- `print_list()` sees an **empty list,** and returns 0.
- We return 1.

Recursion

- A **recursive** function is one that *calls itself*.
  - Must pass a “simpler” parameter
  - Must have a “base case”

(more details later!)

```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list(head.next)
```

Group Discussion:

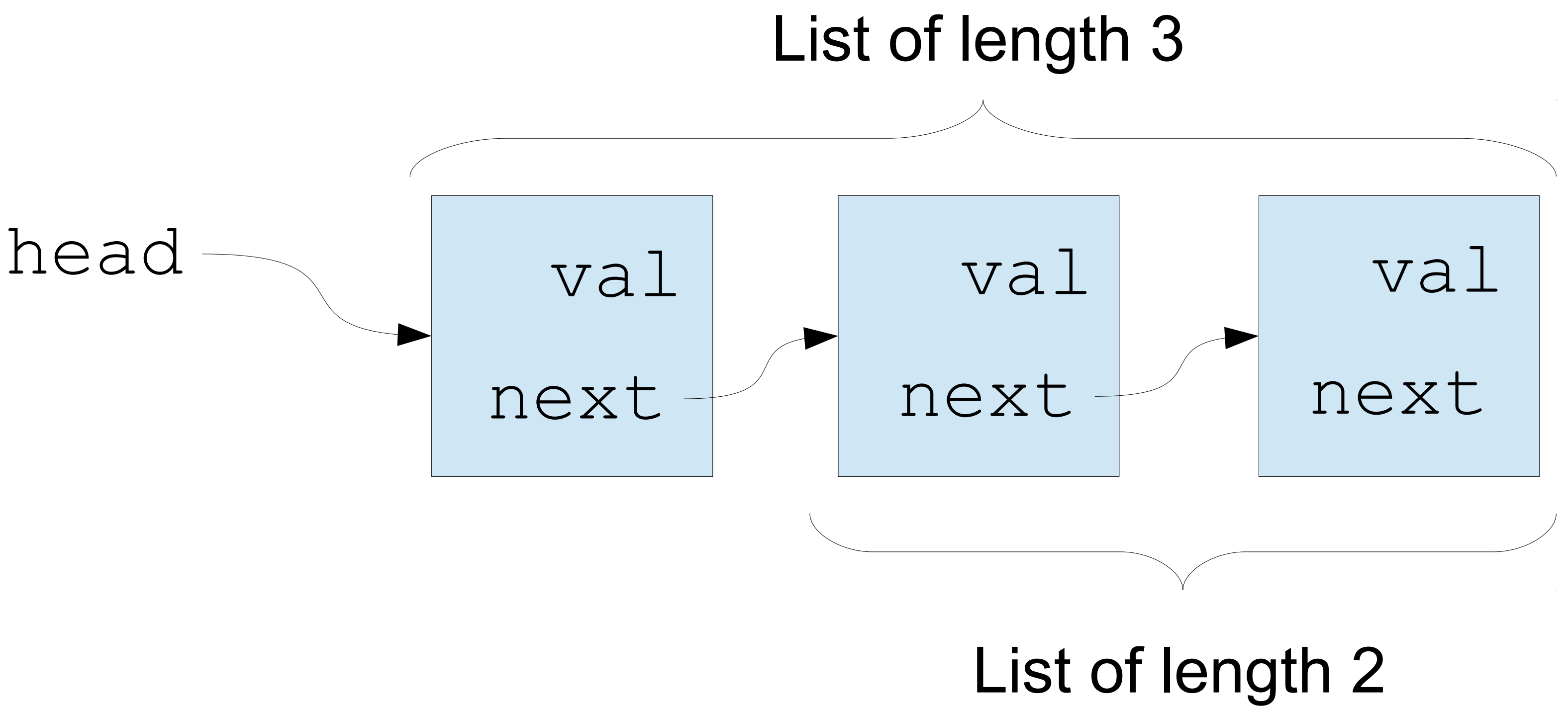
Suppose that we alter this function to call `print_list_2()` instead of `print_list()`. Do we need to make any other changes?

```python
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list_2(head.next)
```

This **recursive call** calls the same function, but with a simpler parameter.

In this case, we realize that the “next” pointer really points to a **slightly shorter list.**

```python
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list_2(head.next)
```



```
def print_list_2(head):
    if head is None:
        return 0
    else:
        print(head.val)
        return 1+print_list_2(head.next)
```

The **base case** is an empty list.

Thinking about Insertion

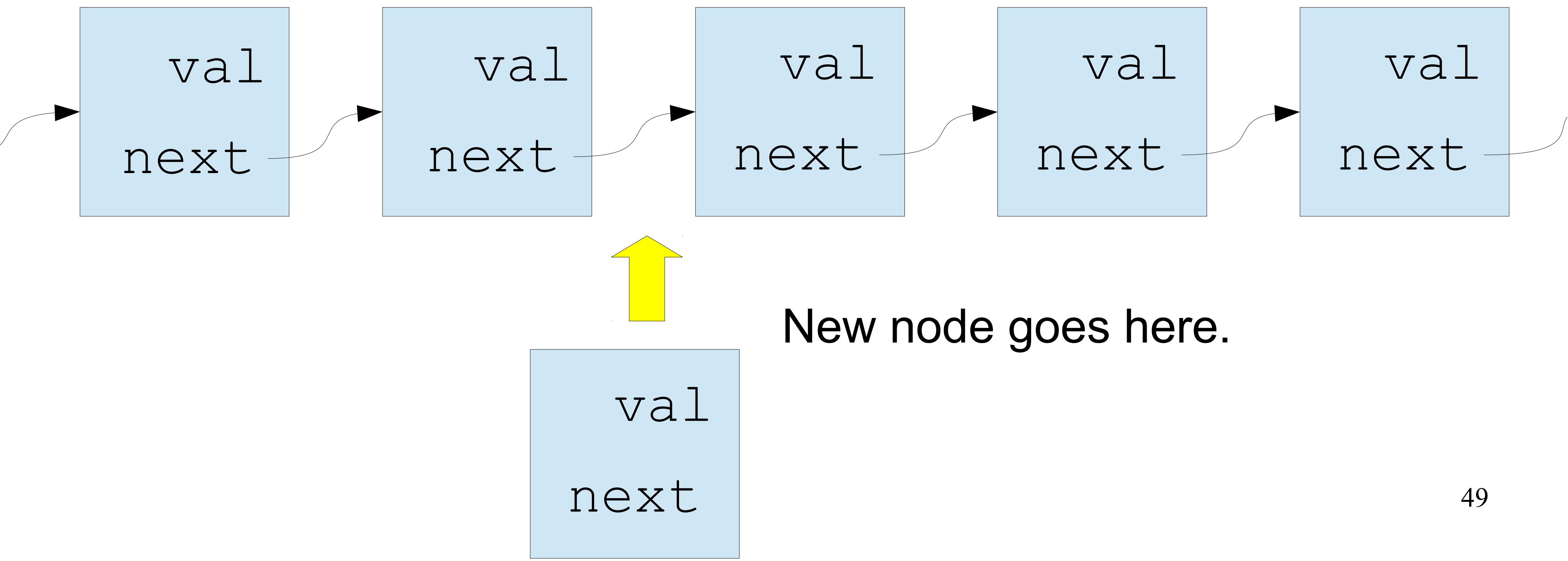
- We're about to try to **insert** a new node into a list. How do we think about this?

- Scan the list for the proper location
- Update references to add the node to the list

- But there's a trick – we need to update **two** references.

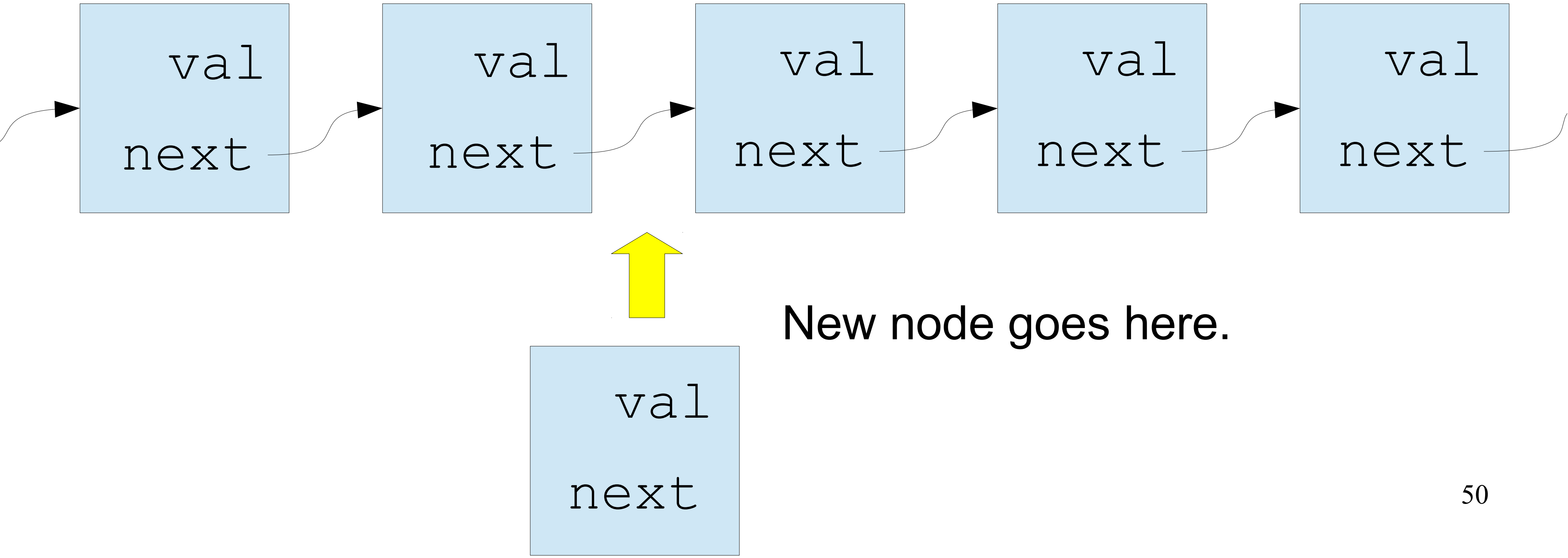
Thinking about Insertion

- Suppose we have a long list. We've found the correct place to put our new node.

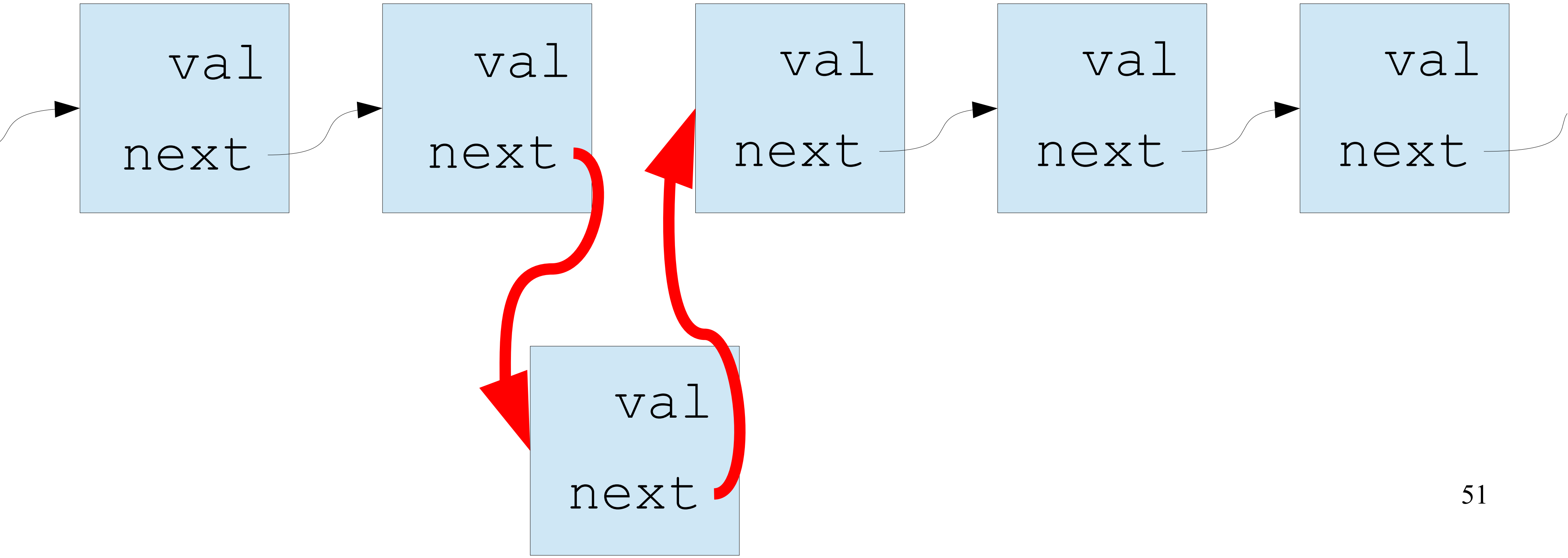


Group Exercise:

Which reference(s) need to be changed in order to insert this new node into the list?

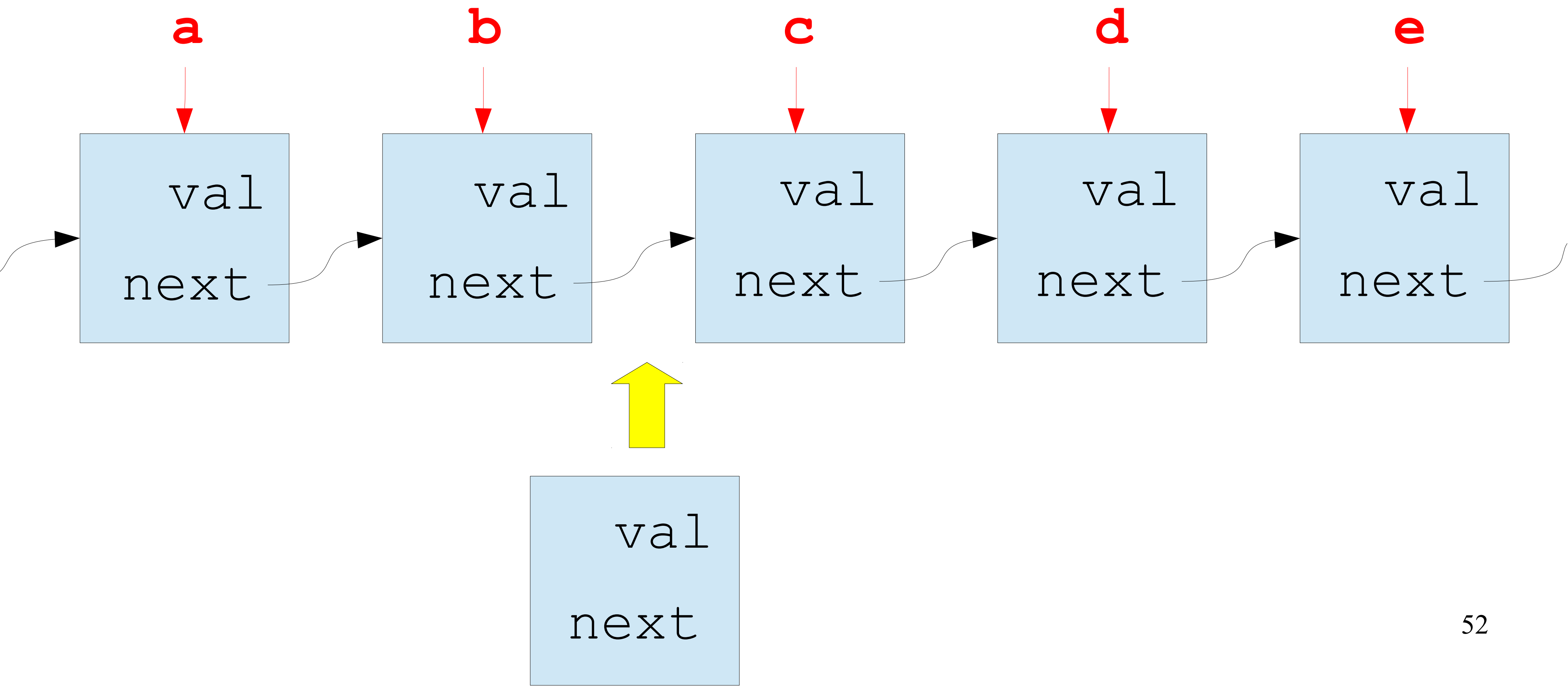


Thinking about Insertion

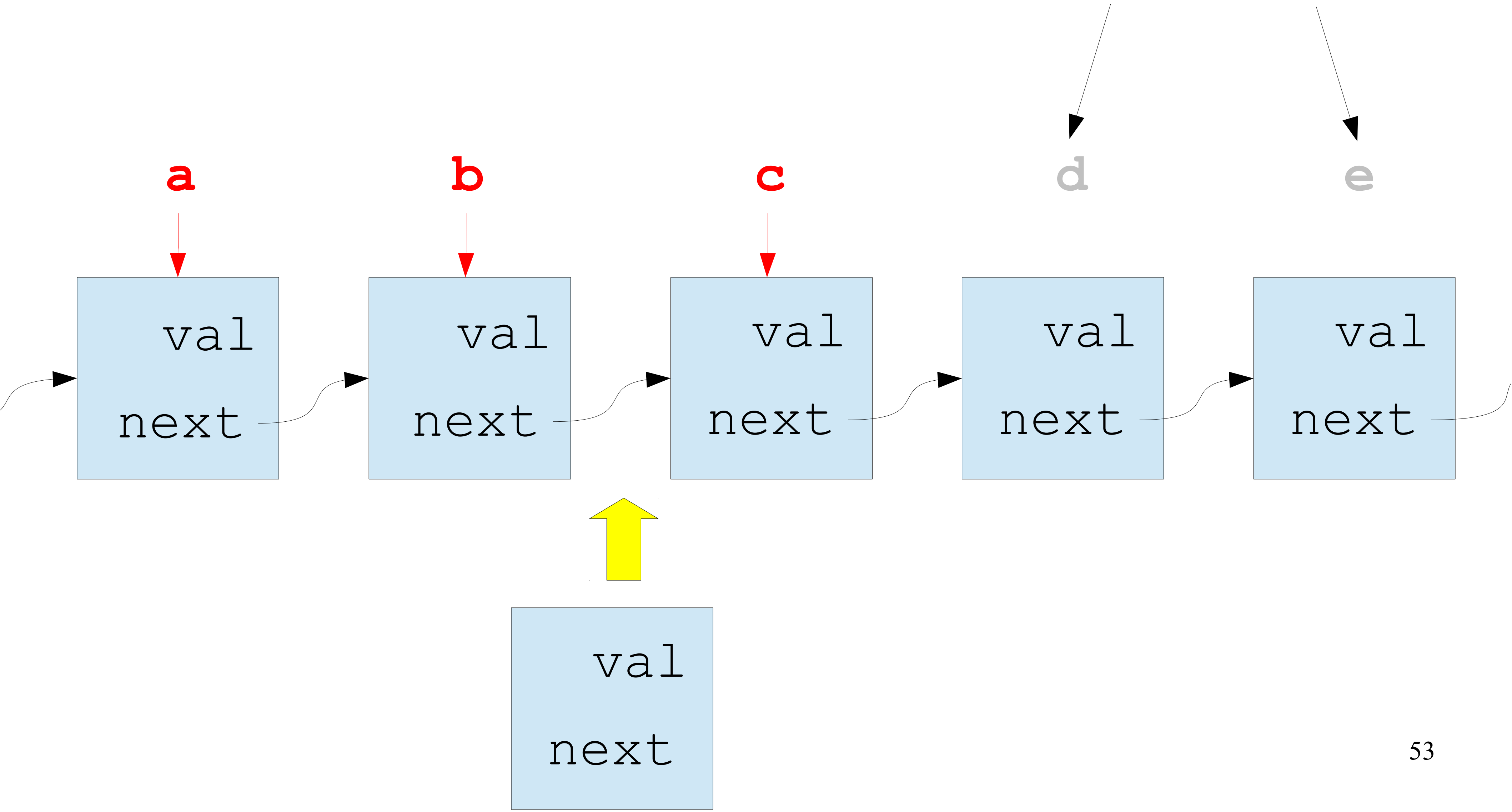


Group Exercise:

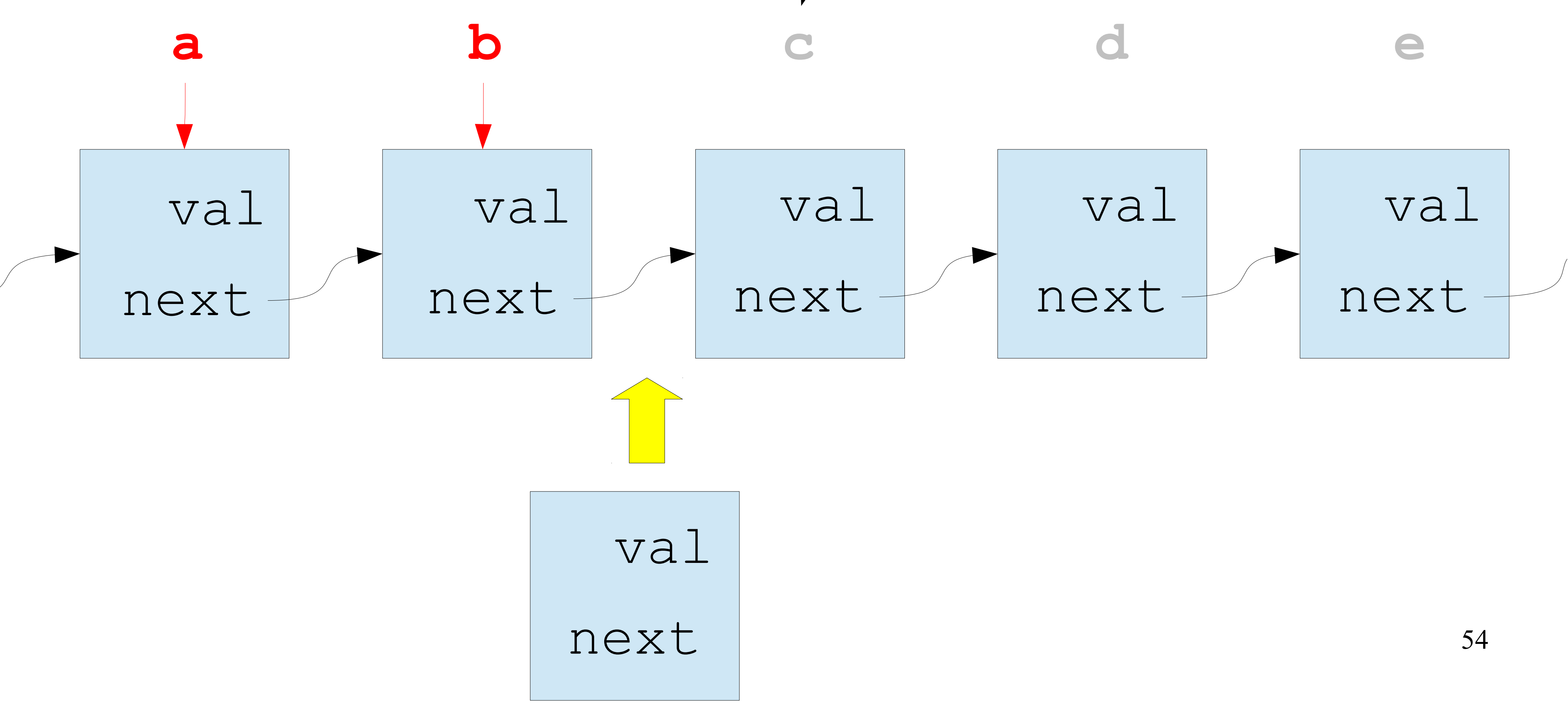
Consider these possible positions for `cur`. Which one is the correct one, which allows us to update the necessary references?



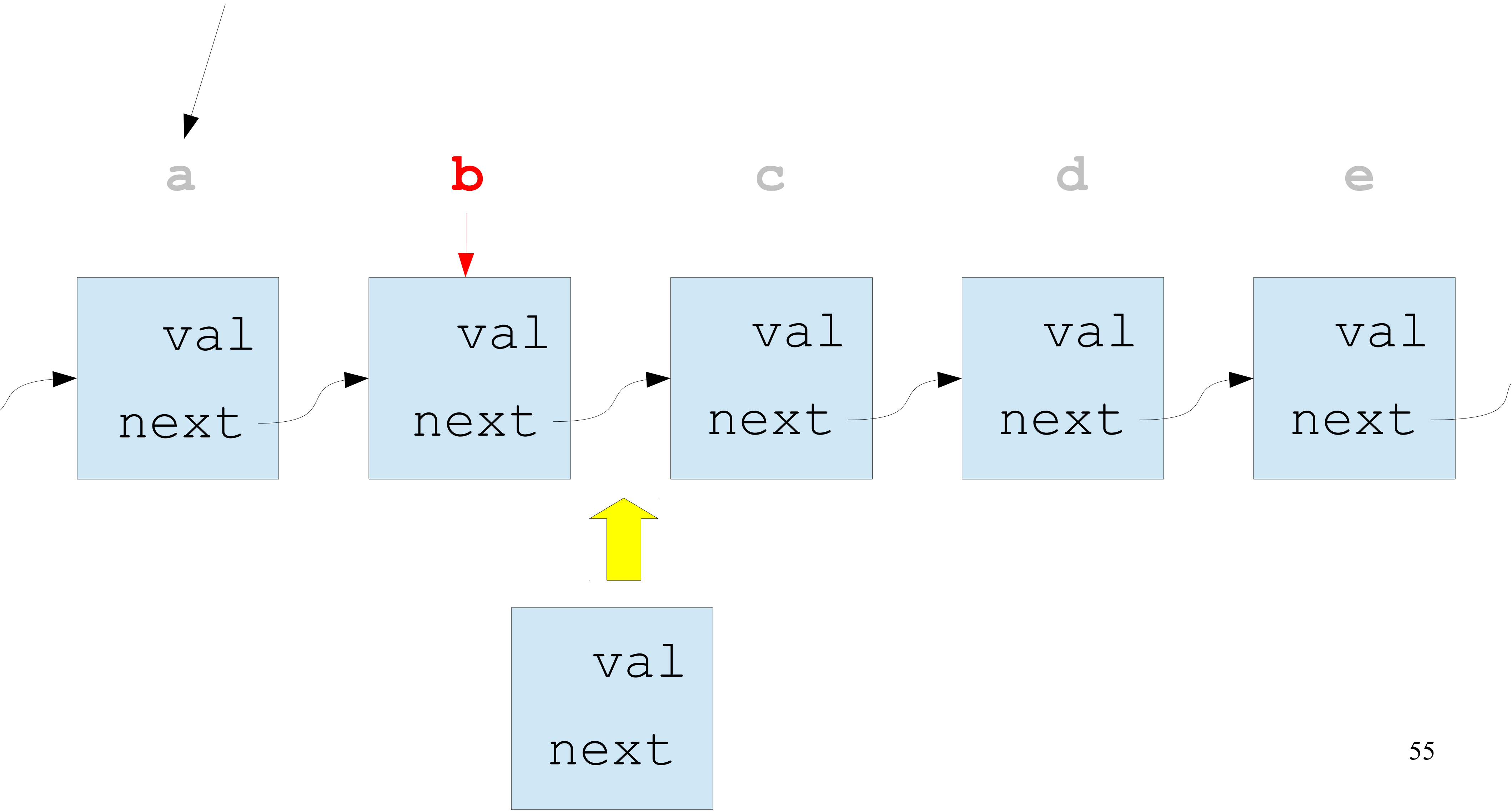
These certainly are far too late...



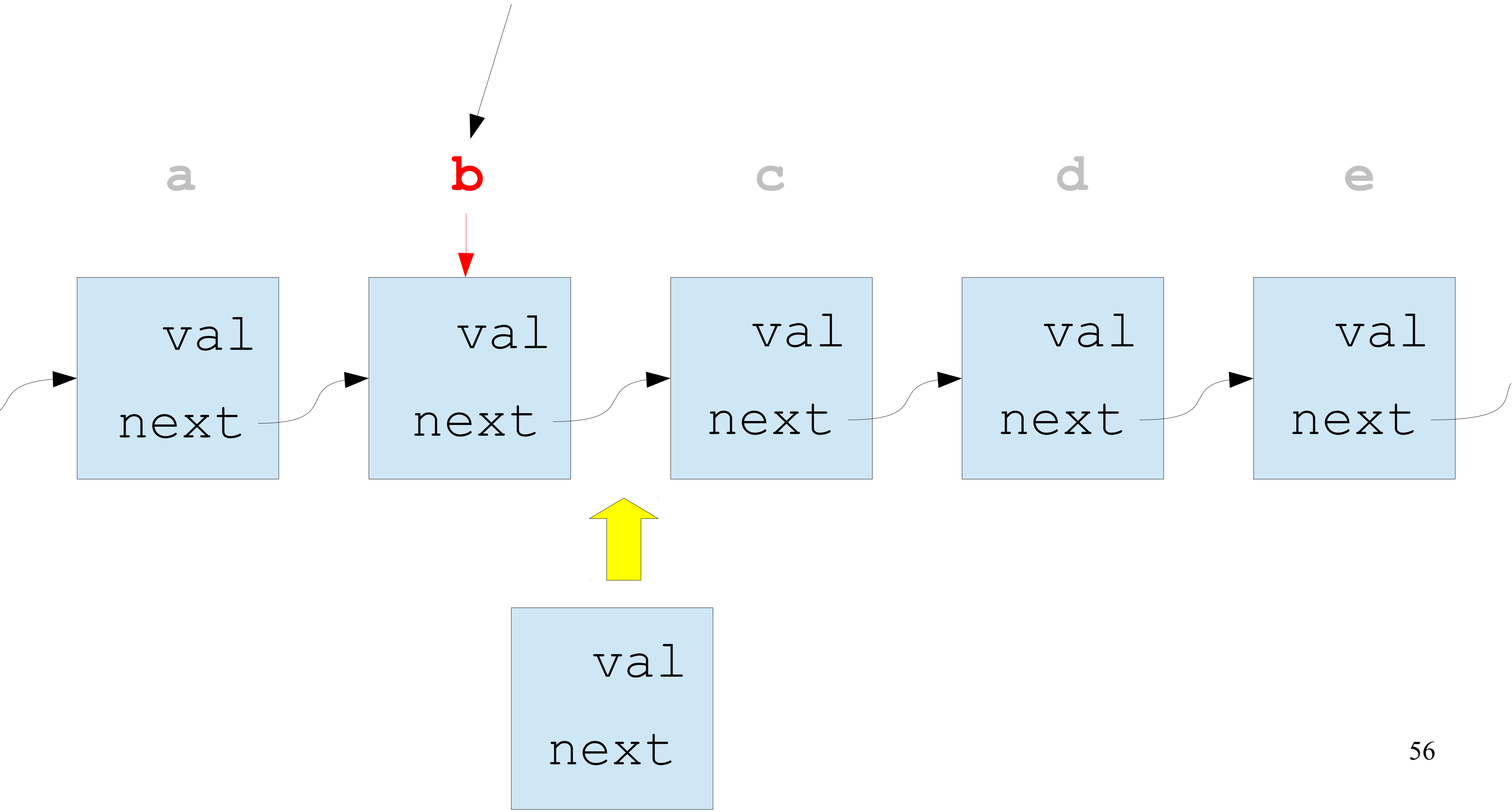
This is close, but it won't be able to set the `b.next` reference



You *could* use this, but it would be awkward.



This has direct access. Inserting is easy if we ***insert after*** a node.

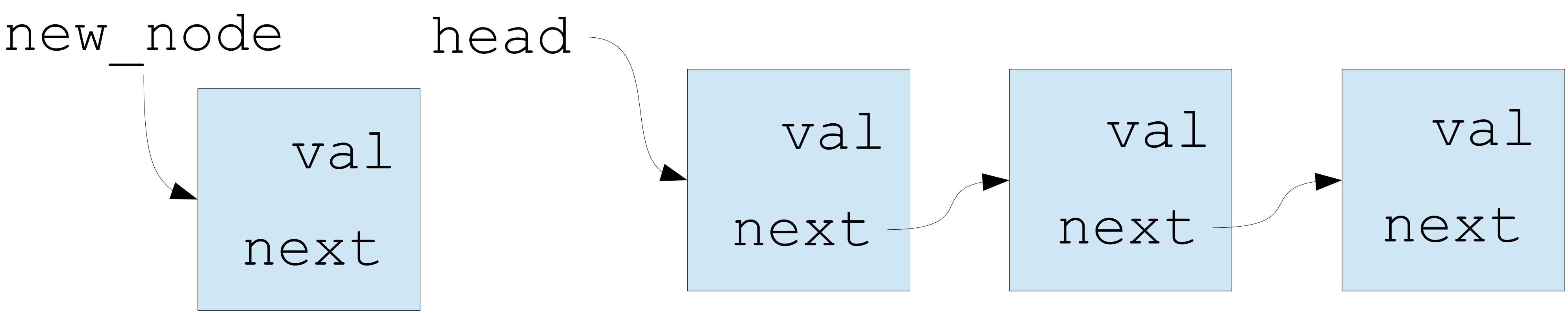


Insertion

Group Exercise:

Write Python code to insert. Assume we have a (sorted) linked list. `head` stores the head pointer.

Assume that I've given you a node, named `new_node`. Insert it into the list at the proper place. Update `head` if necessary.



How to Think About Loops

Did you have trouble writing the insert loop?

Do you think it might be hard to write more complex operations?

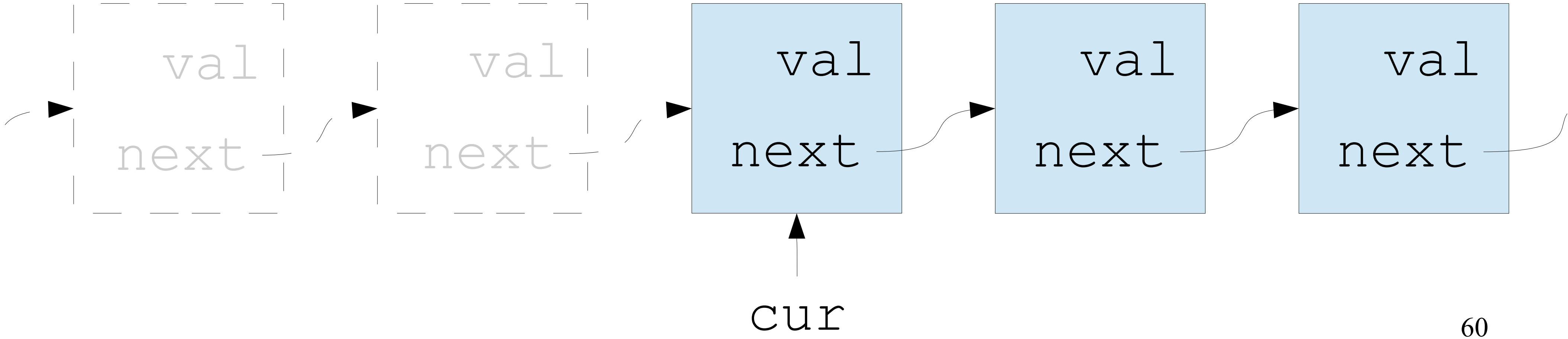
How can we think about loops *in general?*

How to Think About Loops

- When implementing a loop over a data structure, try to think of each pass *completely in isolation*.
  - What do we know now?
  - Where do we go next?
  - Try to ignore what happened before!

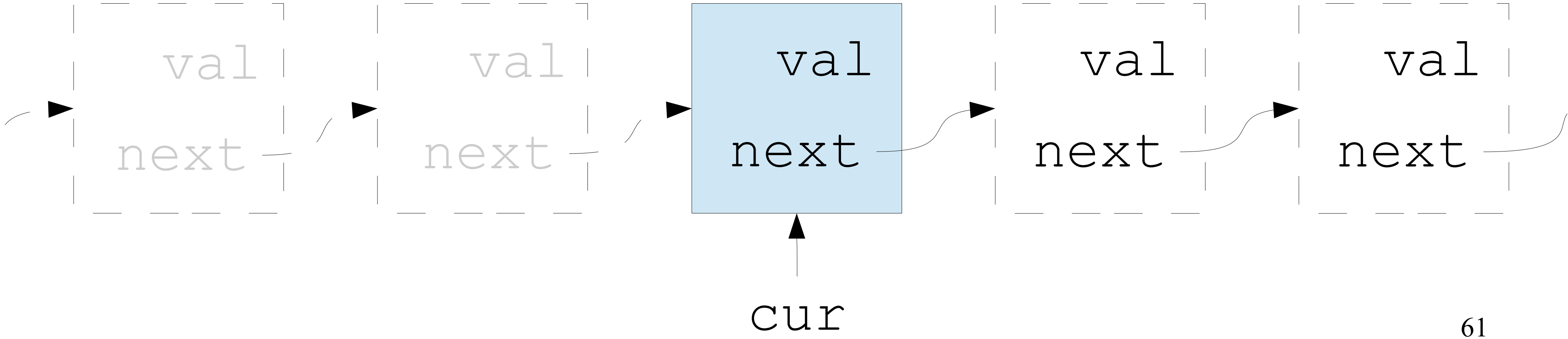
How to Think About Loops

- Even though a linked list has many nodes, you have to *focus on a single node at a time.*
  - Pretend that you “just woke up” here, and you don't remember anything that happened before.



How to Think About Loops

- As much as possible, ignore what comes next
  - Sometimes, it's critical
  - But do as little as you can.

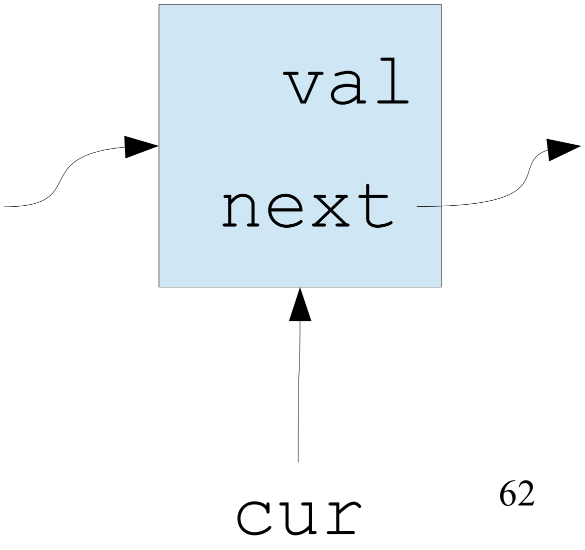


How to Think About Loops

- Use this template for most loops over lists:

```
if head is None:
    ... special case ...

cur = head
while cur is not None:
    ... handle cur ...
    cur = cur.next
```



```
head     = ... some list ...
new_node = ... a node ...

if head is None:
    head = new_node
elif head.val > new_node.val:
    new_node.next = head
    head = new_node
else:
    cur = head
    while cur.next is not None and
          cur.next.val <= new_node.val:
        cur = cur.next
    new_node.next = cur.next
    cur.next = new_node
```

```
if head is None:
    head = new_node
elif head.val > new_node.val:
    new_node.next = head
    head = new_node
else:
    cur = head
    while cur.next is not None and
          cur.next.val <= new_node.val:
        cur = cur.next
    new_node.next = cur.next
    cur.next = new_node
```

Group Exercise:

Draw a list, and simulate this algorithm using the picture.

Make sure to keep track of `cur`.

```
if head is None:
    head = new_node
elif head.val > new_node.val:
    new_node.next = head
    head = new_node
else:
    cur = head
    while cur.next is not None and
          cur.next.val <= new_node.val:
        cur = cur.next
    new_node.next = cur.next
    cur.next = new_node
```

Group Exercise:

Try simulating:

- Insert into an empty list
- Insert at head of non-empty list
- Insert at end of the list

A `ListNode` class

- It's time to implement a class to model our list nodes.
- A **class** defines a pattern for our objects
- Each **instance** of the class uses the same basic code

A `ListNode` class

```
class ListNode:
    def __init__(self, val):
        self.val  = val
        self.next = None
```

- This defines a class named `ListNode`

A `ListNode` class

```
class ListNode:
    def __init__(self, val):
        self.val  = val
        self.next = None
```

- The **constructor** runs each time that a new object has been created.

A `ListNode` class

```
class ListNode:
    def __init__(self, val):
        self.val  = val
        self.next = None
```

- The **constructor** runs each time that a new object has been created. It defines all of the **fields** of the object.

Difference from C/Java: We don't explicitly define object fields. Instead, we just *set* a few of them.

A `ListNode` class

```
a = ListNode(3)
b = ListNode(10)
c = ListNode(3)
d = ListNode("foo")
```

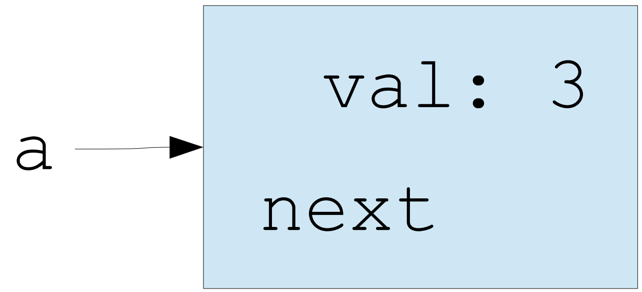
- To create **instances** of a class, call the class name, like a function.

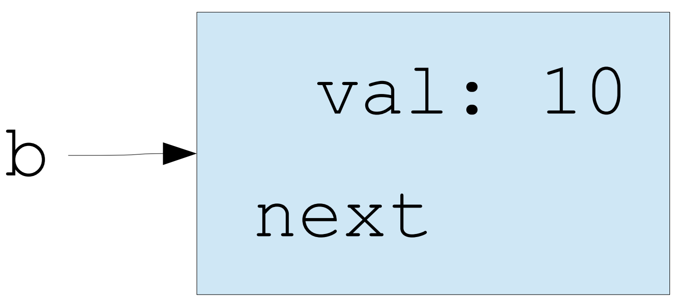
Group Exercise:

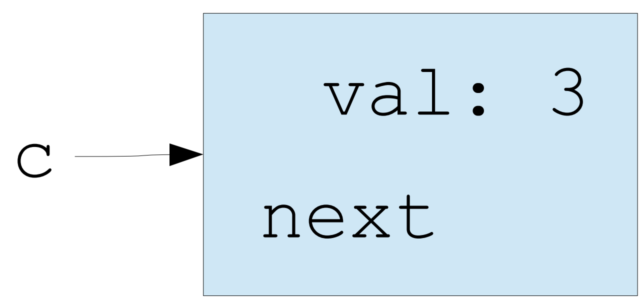
Draw the picture for the objects created above.

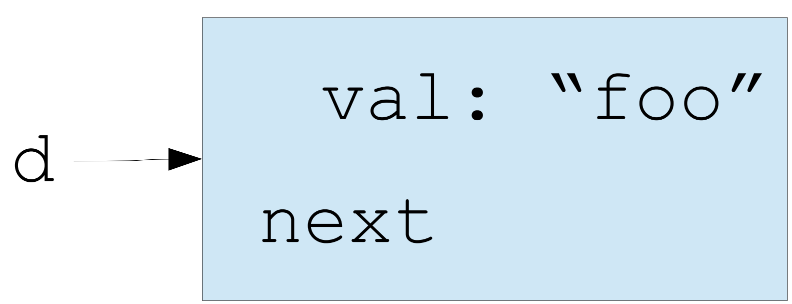
A `ListNode` class

```
a = ListNode(3)
b = ListNode(10)
c = ListNode(3)
d = ListNode("foo")
```









Difference from C/Java: Python data structures can hold data of any type, even mixed!

Building a List

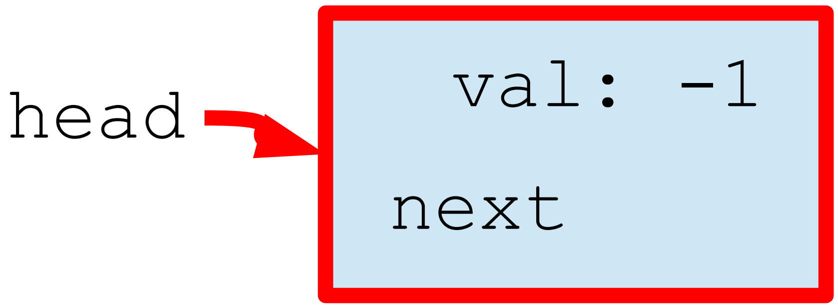
```
head                     = ListNode(-1)
head.next                = ListNode(3)
head.next.next           = ListNode(100)
head.next.next.next      = ListNode(101)
```

- Use the 'dot' syntax to read/write the fields of an object.

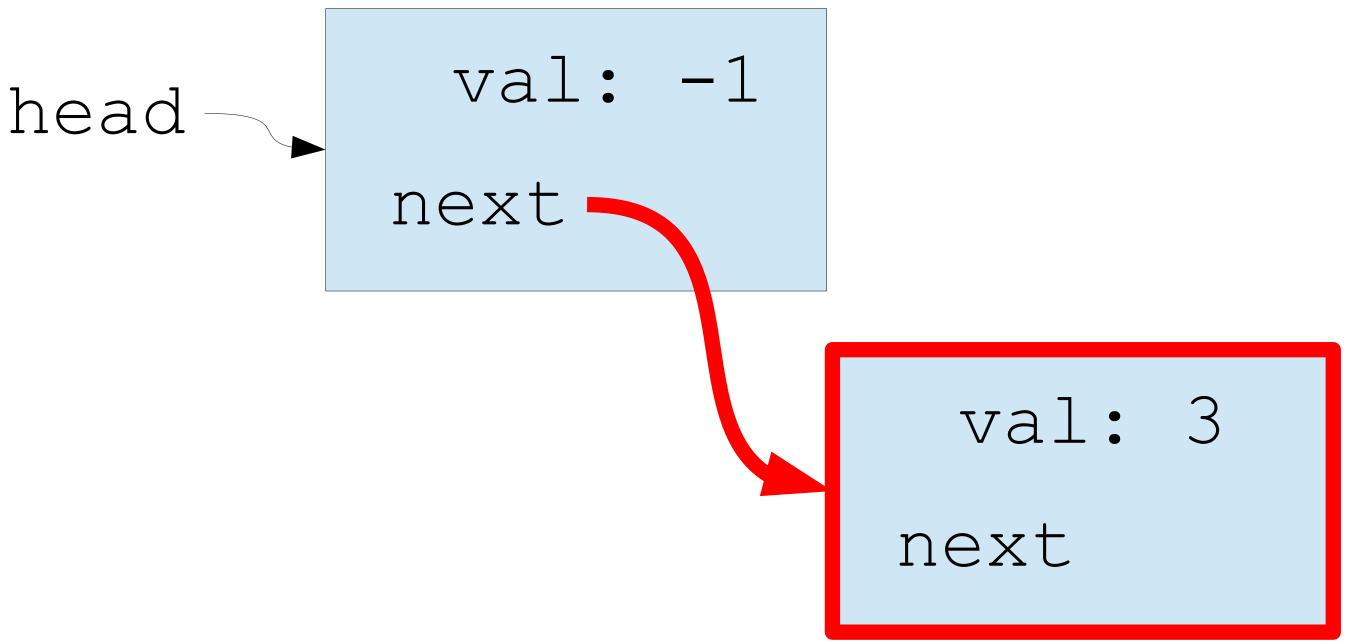
Group Exercise:

Draw the picture for the objects created above.

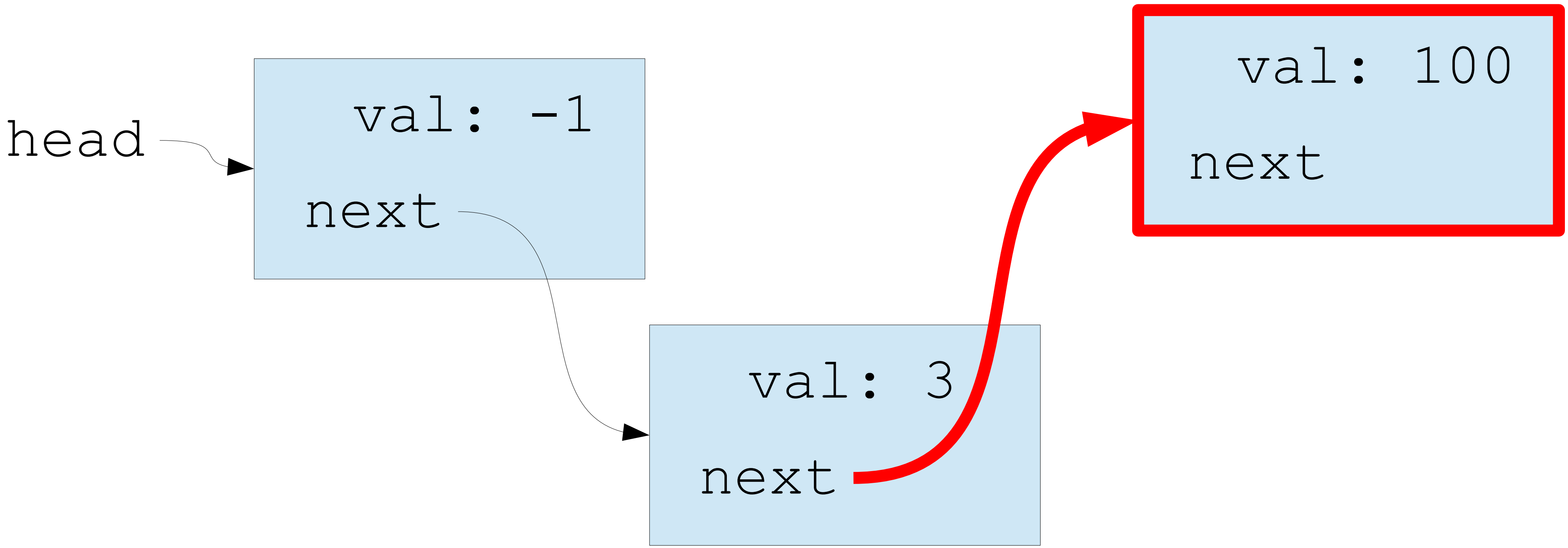
```
head                     = ListNode(-1)
head.next                = ListNode(3)
head.next.next           = ListNode(100)
head.next.next.next      = ListNode(101)
```



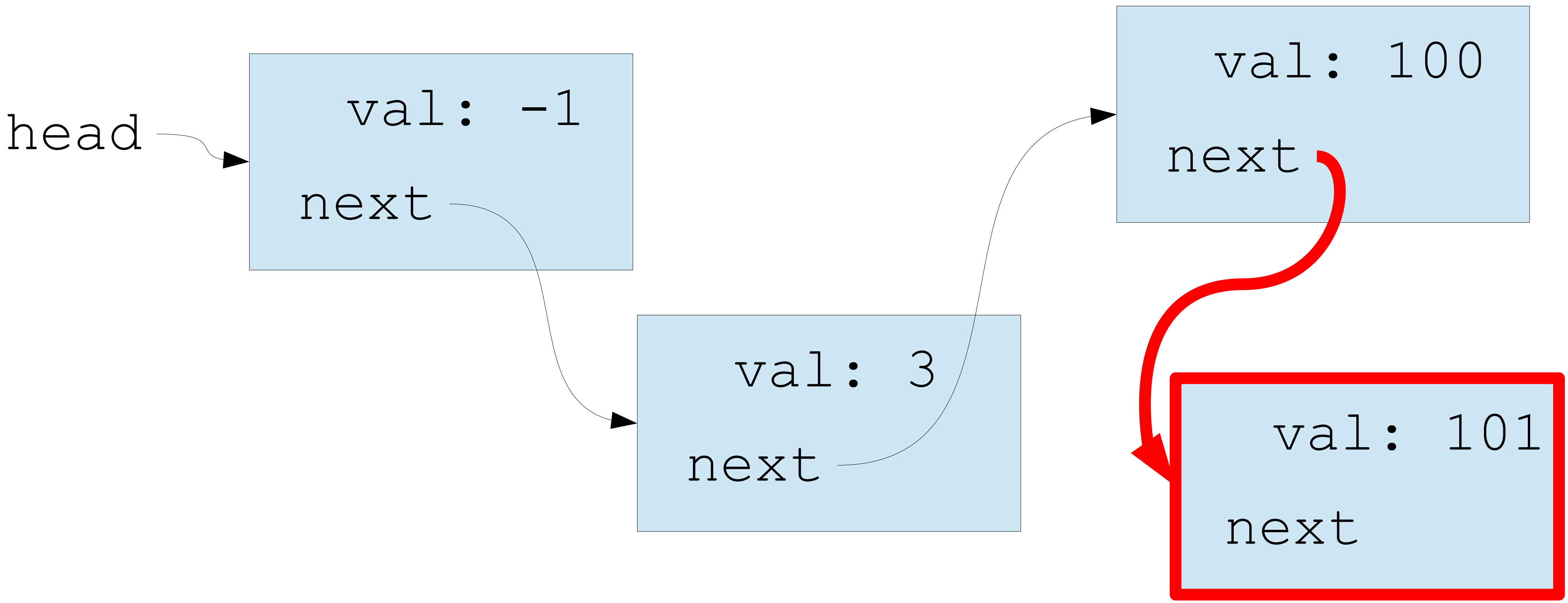
```
head                     = ListNode(-1)
head.next                = ListNode(3)
head.next.next           = ListNode(100)
head.next.next.next      = ListNode(101)
```



```
head                = ListNode(-1)
head.next           = ListNode(3)
head.next.next      = ListNode(100)
head.next.next.next = ListNode(101)
```



```
head                = ListNode(-1)
head.next           = ListNode(3)
head.next.next      = ListNode(100)
head.next.next.next = ListNode(101)
```



Building a List

```
head = ListNode("fred")
cur = head

cur.next = ListNode("wilma")
cur = cur.next

cur.next = ListNode("barney")
```

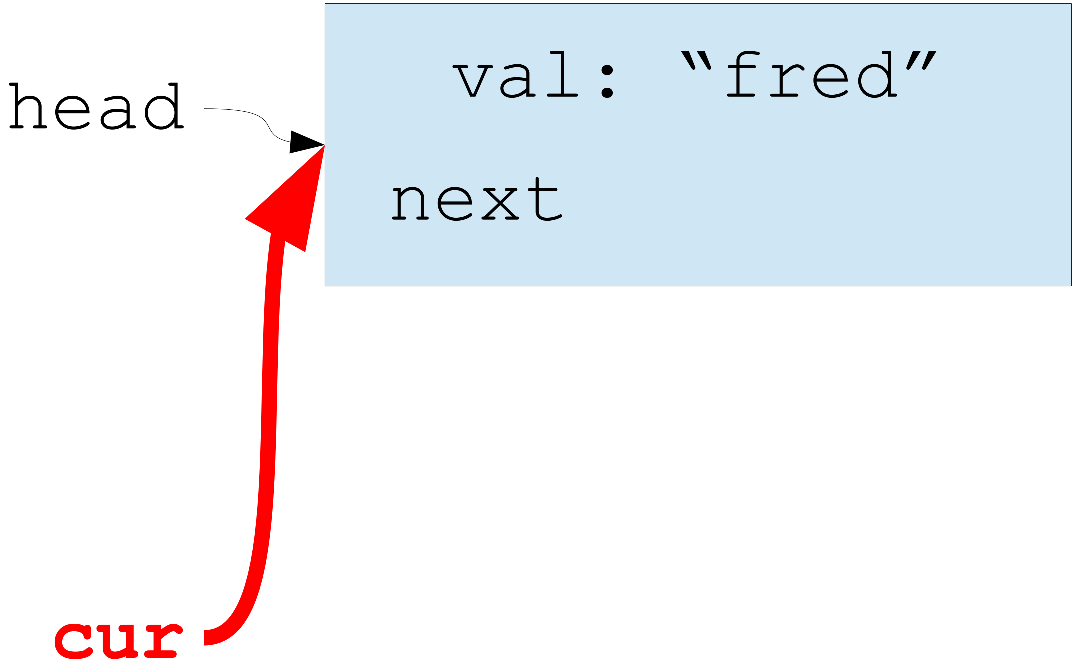
Group Exercise:

Draw the picture

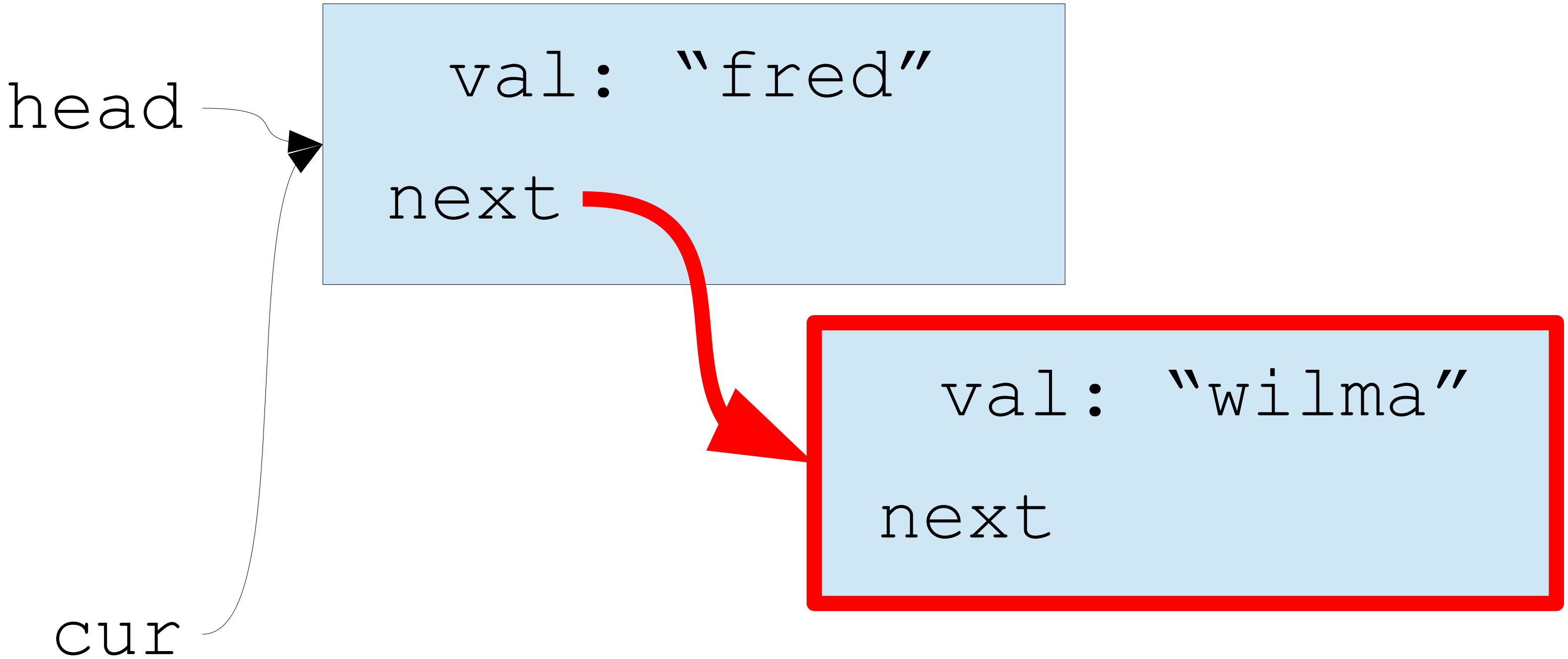
- `cur` pointers are not always required when building lists
- But sometimes, they make it easier.

```
head = ListNode("fred")
cur = head
cur.next = ListNode("wilma")
cur = cur.next
cur.next = ListNode("barney")
```

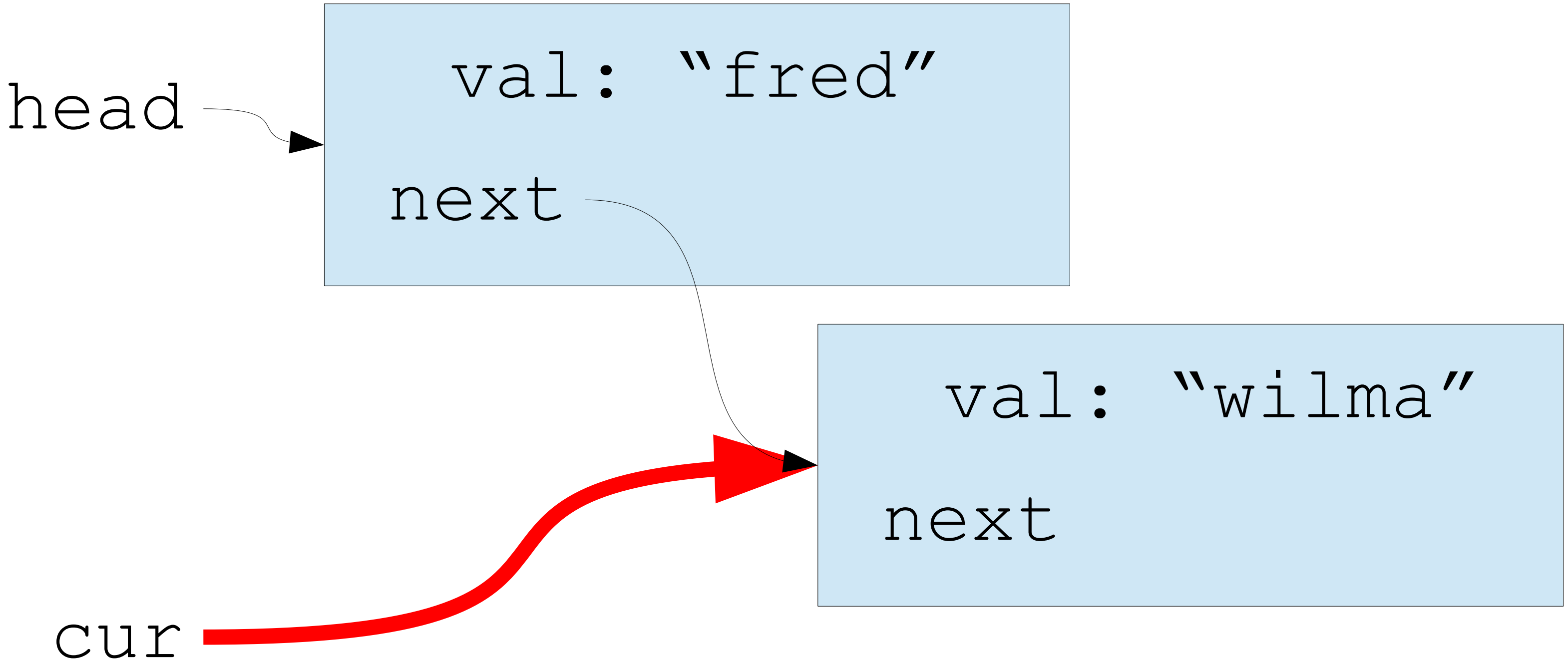
```
head = ListNode("fred")
cur = head
cur.next = ListNode("wilma")
cur = cur.next
cur.next = ListNode("barney")
```



```
head = ListNode("fred")
cur = head
cur.next = ListNode("wilma")
cur = cur.next
cur.next = ListNode("barney")
```

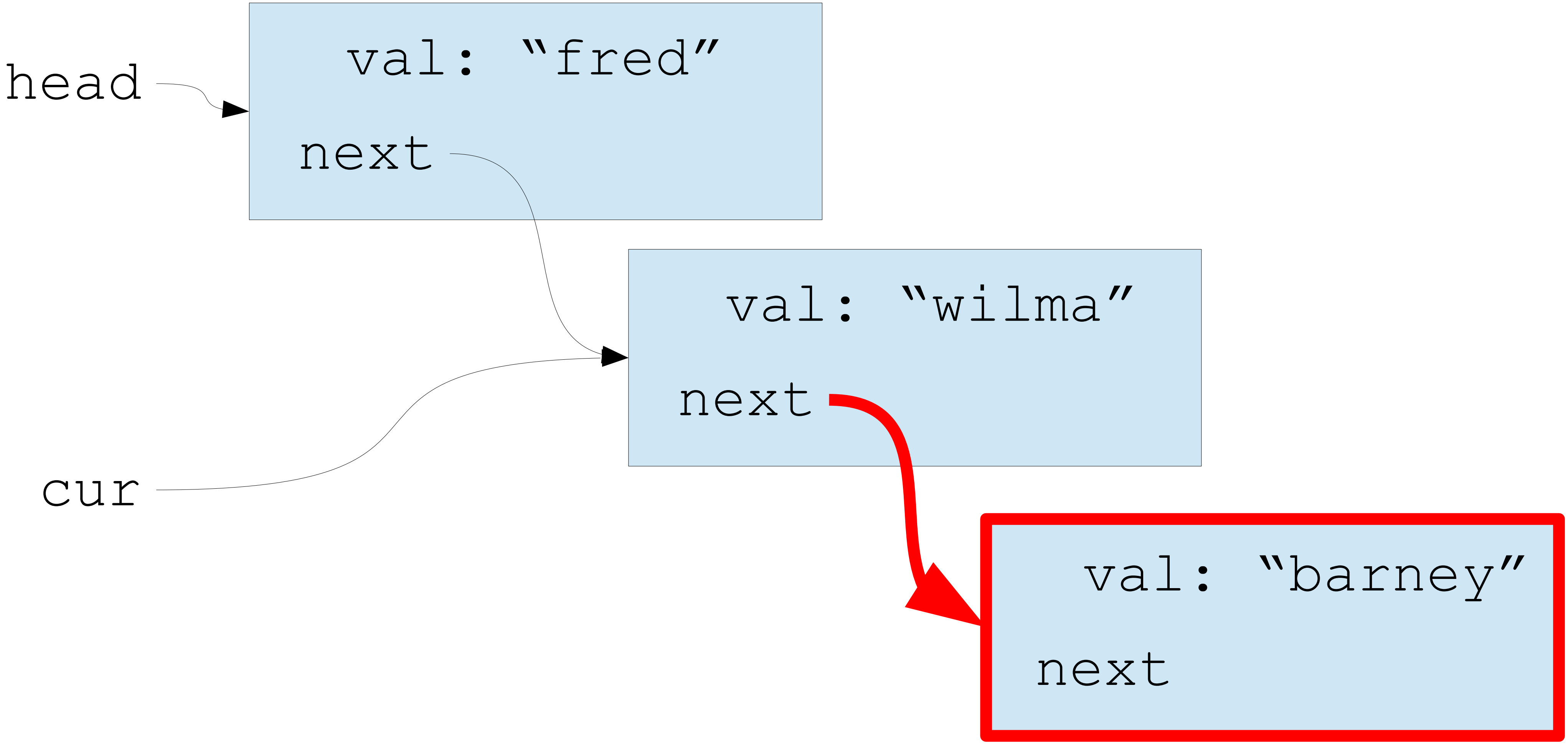


```
head = ListNode("fred")
cur = head
cur.next = ListNode("wilma")
cur = cur.next
cur.next = ListNode("barney")
```



```
head = ListNode("fred")
cur = head
cur.next = ListNode("wilma")
cur = cur.next
```

cur.next = ListNode("barney")



Final Notes

- Often, we use two classes:
  - `ListNode` to represent the nodes
  - `List` to represent an entire list as one object
    - Holds the `head` pointer
    - Has many methods to model operations on the list

```
first_list = List()     # starts empty
first_list.insert_sorted(10)
print(len(first_list))

another = List()        # starts empty
```

Final Notes

- There are many variants on lists
  - Doubly-linked
  - Circular
  - Sorted or not
  - Many possible “extra” features, like faster searching

- Many more advanced data structures use references
  - Example: Binary trees have **two** pointers per node.

Closing Exercise

Group Exercise:

Write a function that takes an array of values as input, and builds a linked list which contains the same values – in exactly the same order. You will need to build all of the nodes.

Return the list. (That is, return the reference to the head of the list.)

Extra Challenge:
Can you write a recursive version?

Closing Exercise

```
def arr2list(vals):
    if len(vals) == 0:
        return None

    head = ListNode(vals[0])
    tail = head

    for v in vals[1:]:
        tail.next = ListNode(v)
        tail = tail.next

    return head
```

Closing Exercise

```
def arr2list_recursive(vals):
    if len(vals) == 0:
        return None

    head      = ListNode(vals[0])
    head.next = arr2list_recusive(vals[1:])

    return head
```

Hidden Cost:
This code works, but doing lots of slicing, over and over, makes this code surprisingly slow.

A better version would never slice the array, and instead include an index into the array as a 2nd parameter.