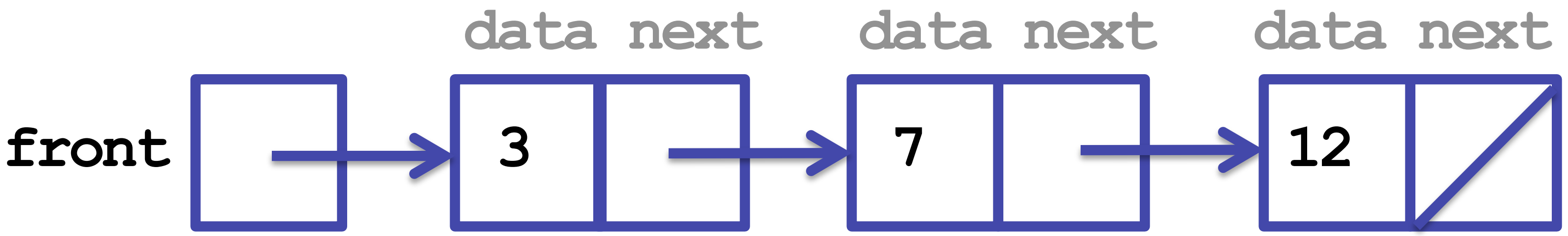
CSE 143
Lecture 7

Linked Lists and Loops

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Review

- Recall the linked list containing 3, 7, and 12:



- We can print all these elements without loops:

```
// prints first three elements on separate lines
System.out.println(front.data);
System.out.println(front.next.data);
System.out.println(front.next.next.data);
```

- But this is tedious, and we can't process a list containing thousands of nodes (reasonably)

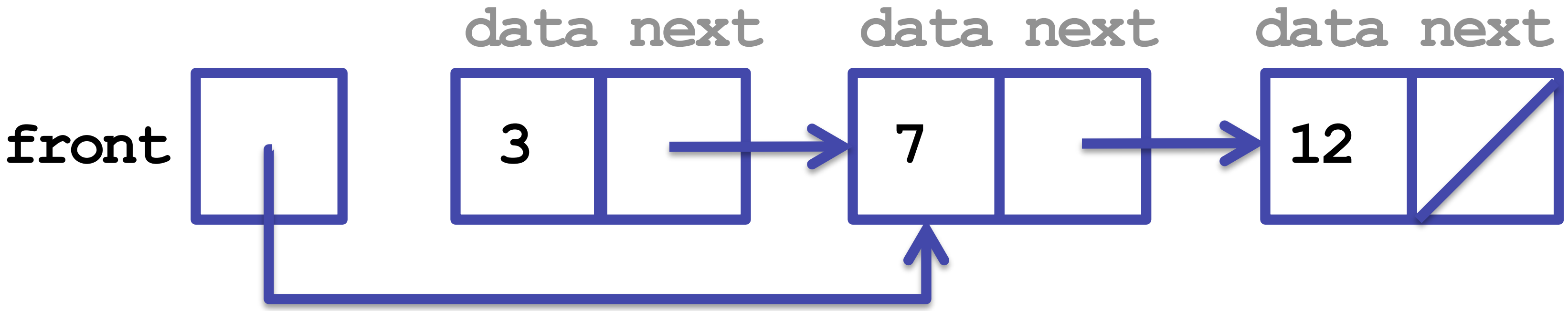
Basics of Linked List Loops

- As a first attempt, let's start with our only variable (**`front`**)

- How can we move **`front`** forward through the list?

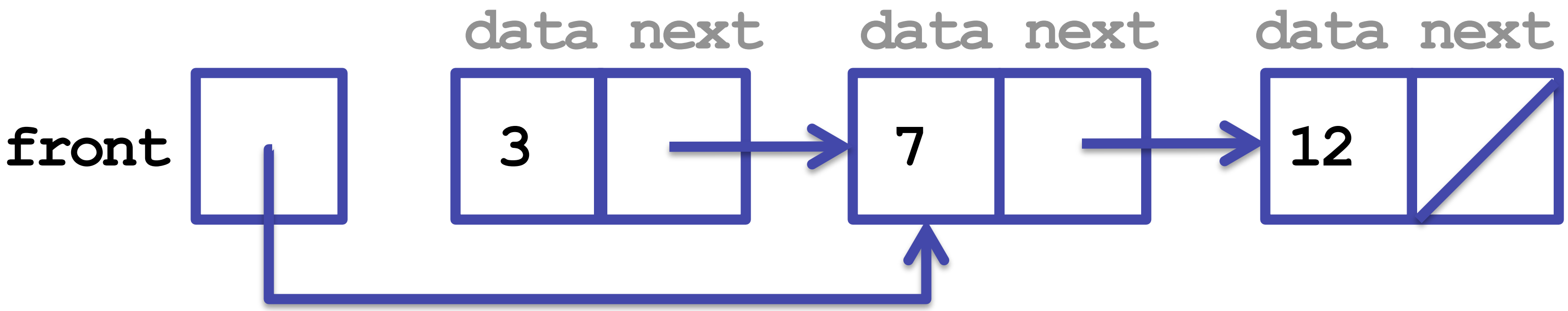
```
front = front.next;
```

- This code changes the list to look like this:

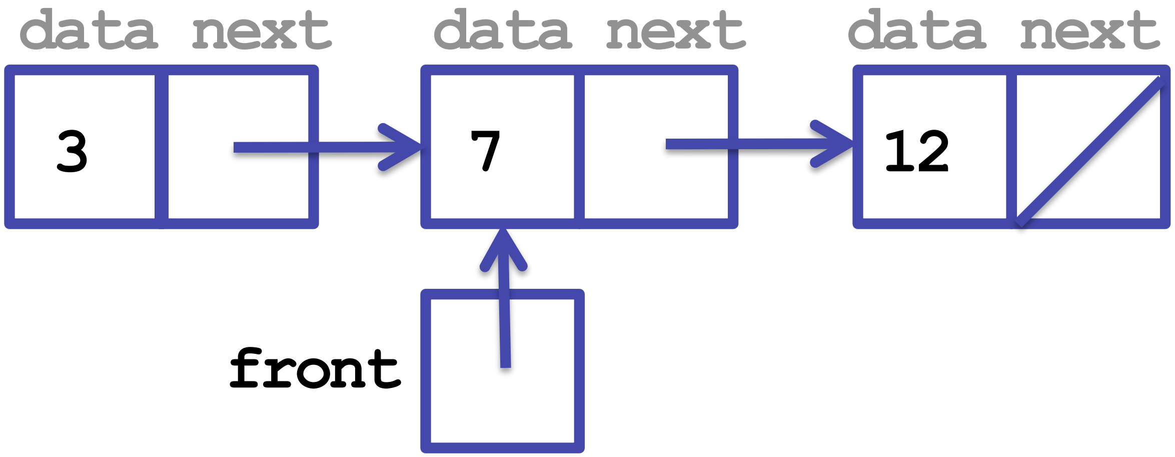


Quick Aside: Drawings

- Some people prefer to draw:



- Like this instead:



- Both ways are equally correct

Basics of Linked List Loops

- So, we can move **`front`** forward through the list

- But how long do we move **`front`** forward?
  - until there is no more data!

- When are we out of data?
  - when **`front`** refers to no **`ListNode`** (**`null`**)

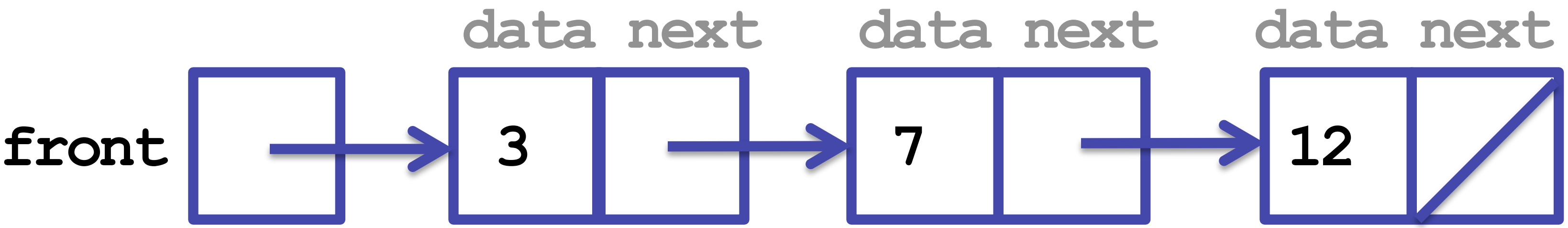
- Code to print all elements:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next; // moves front forward
}
```

Basics of Linked List Loops

- But does this code work? Let's follow it loop-by-loop:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next;
}
```

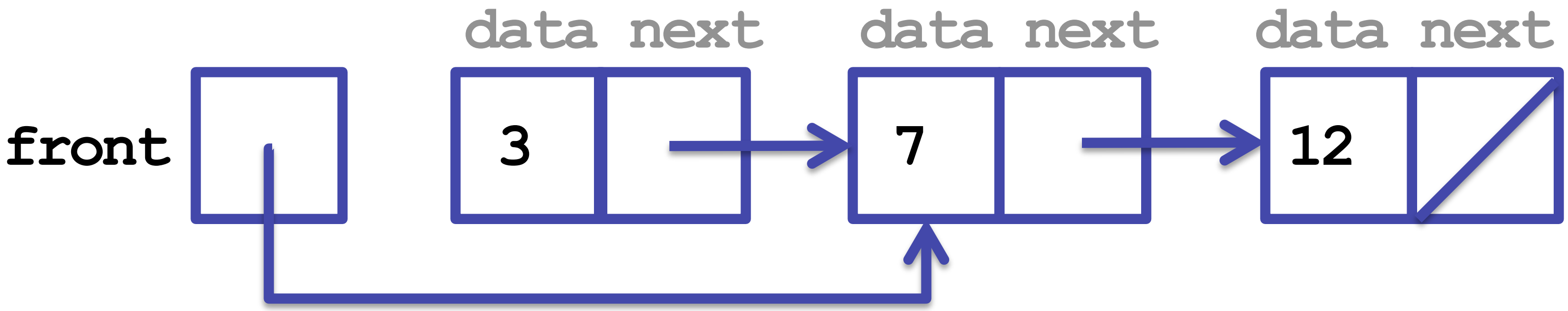


- Output:

Basics of Linked List Loops

- But does this code work? Let's follow it loop-by-loop:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next;
}
```



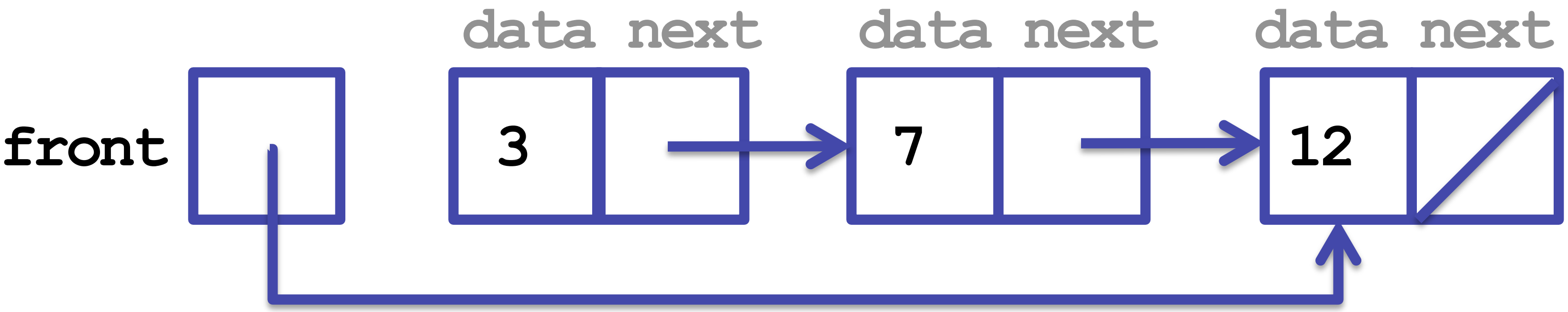
- Output:

```
3
```

Basics of Linked List Loops

- But does this code work? Let's follow it loop-by-loop:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next;
}
```



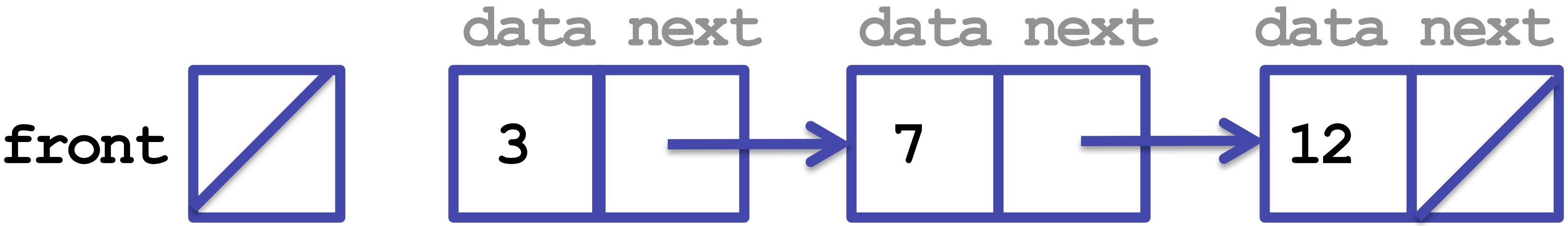
- Output:

```
3
7
```

Basics of Linked List Loops

- But does this code work? Let's follow it loop-by-loop:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next;
}
```



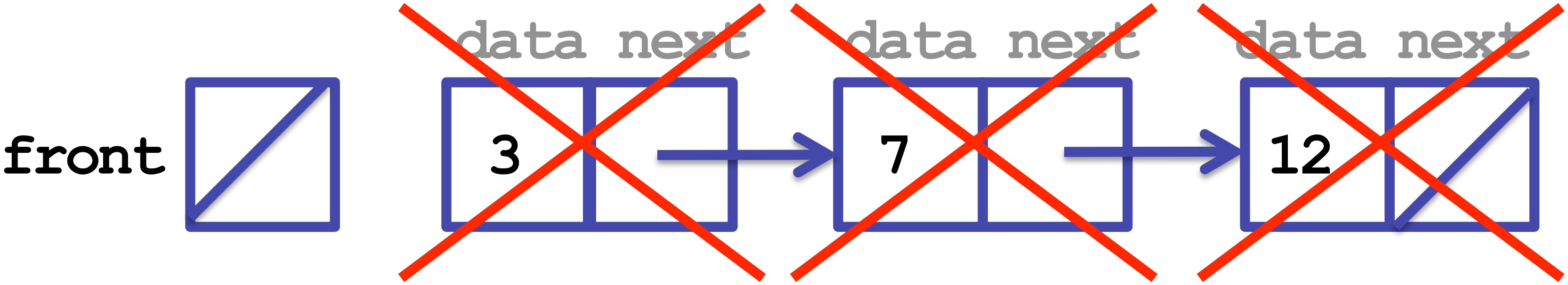
- Output:

```
3
7
12
```

Basics of Linked List Loops

- But does this code work? Let’s follow it loop-by-loop:

```
while (front != null) {
    System.out.println(front.data);
    front = front.next;
}
```



GARBAGE COLLECTED!

- Output:

```
3
7
12
```

Temporary Variables

- Moving `front` through the list destroyed our list

- We need `front` to stay at the front
  - so it can keep track of all the nodes

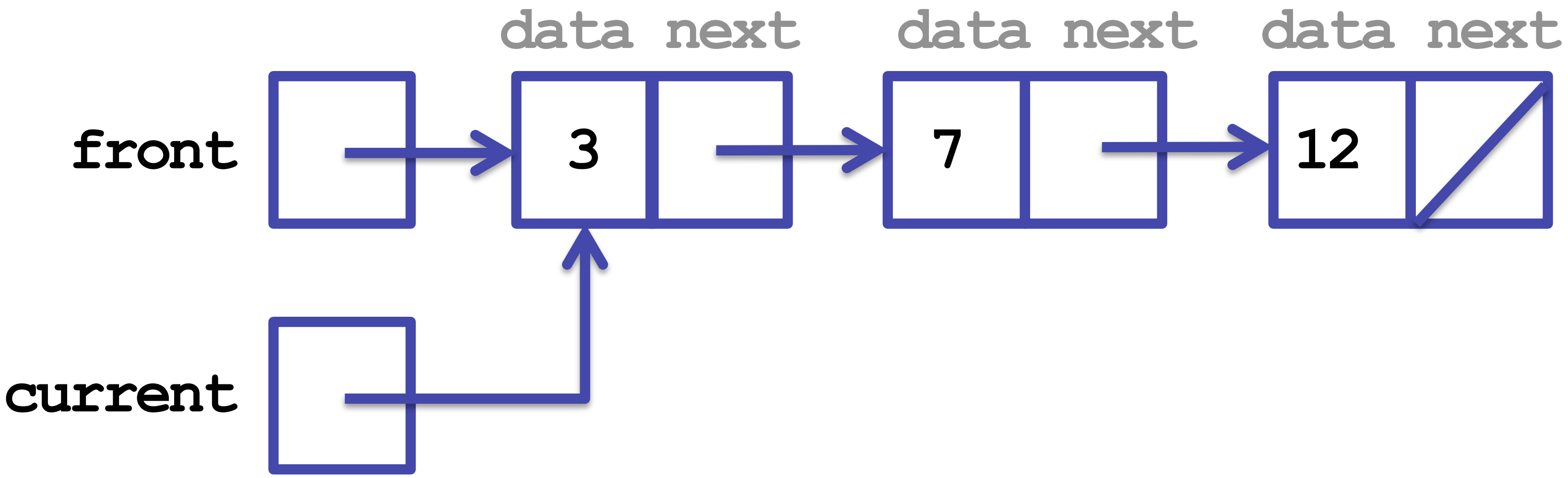
- We can create a temporary variable to move through the list
  - now `front` can stay at the front
  - and we still have a variable to move through the list

Temporary Variables

- Creating a temporary variable:

```
ListNode current = front;
```

- Which updates our picture to look like this:

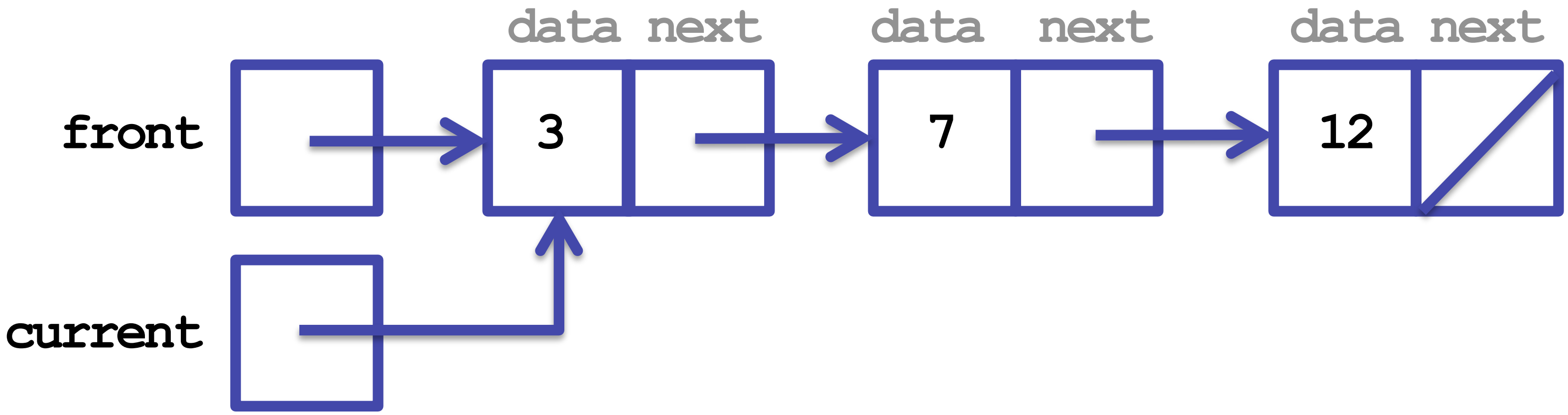


- Notice that we created a new variable. We did *not* create a new **`ListNode`** object.

Basics of Linked List Loops

- Let's update our code and follow it loop-by-loop:

```
ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

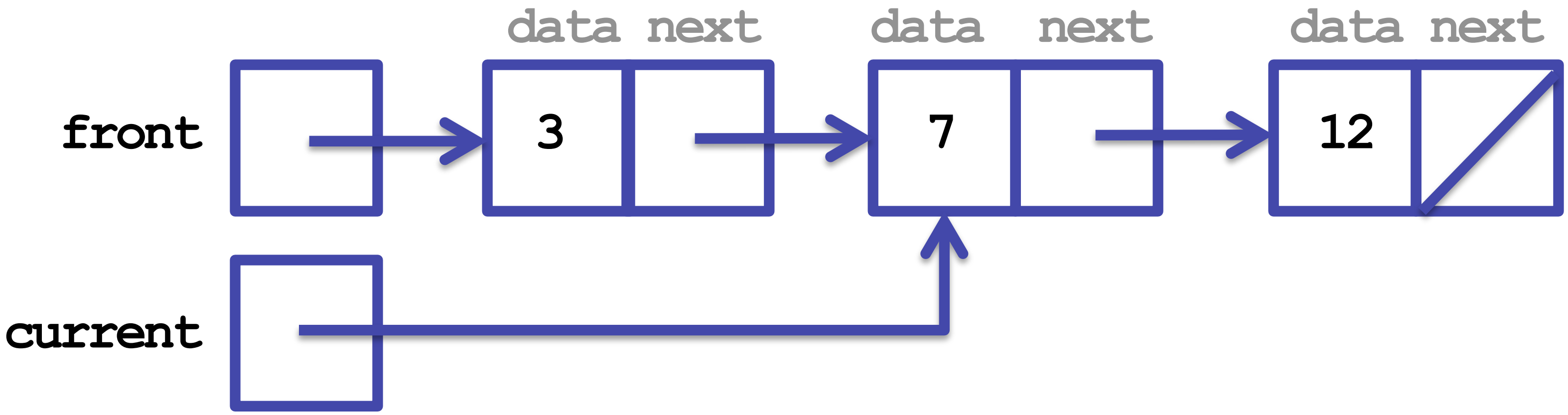


- Output:

Basics of Linked List Loops

- Let’s update our code and follow it loop-by-loop:

```
ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```



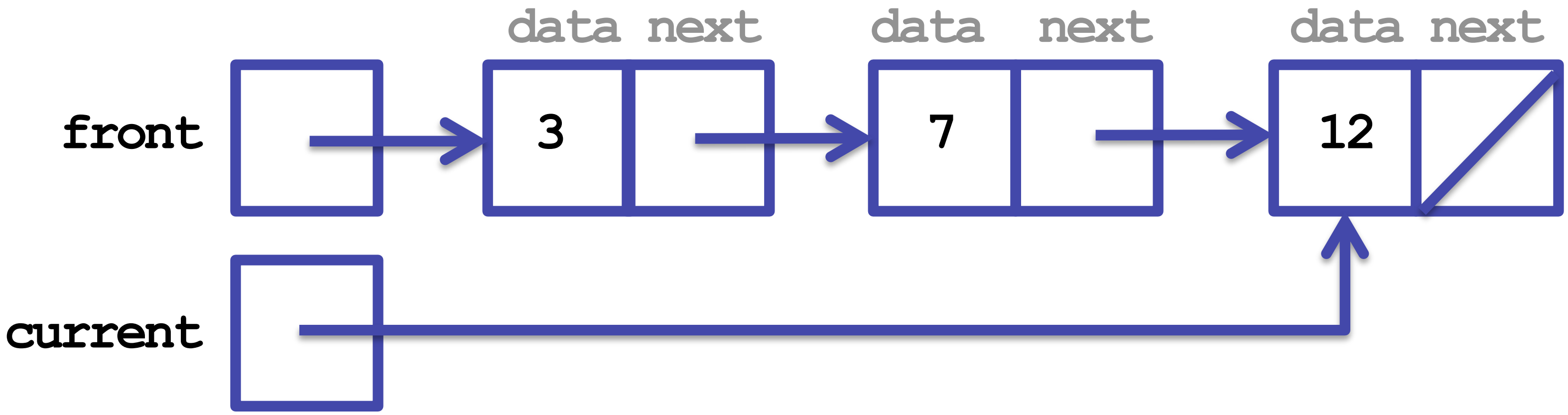
- Output:

  3

Basics of Linked List Loops

- Let’s update our code and follow it loop-by-loop:

```
ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```



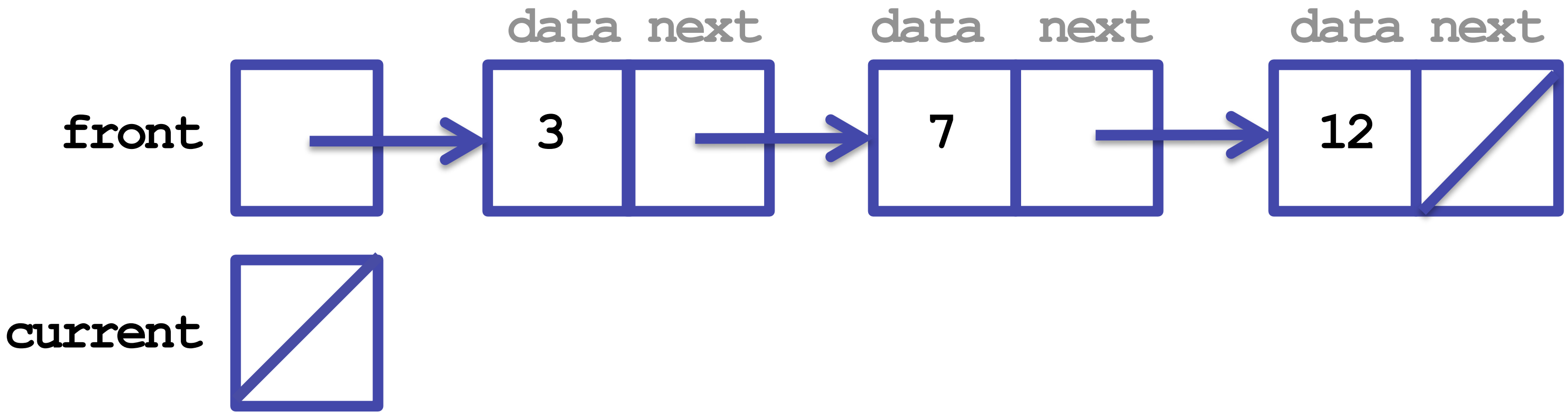
- Output:

```
3
7
```

Basics of Linked List Loops

- Let’s update our code and follow it loop-by-loop:

```
ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```



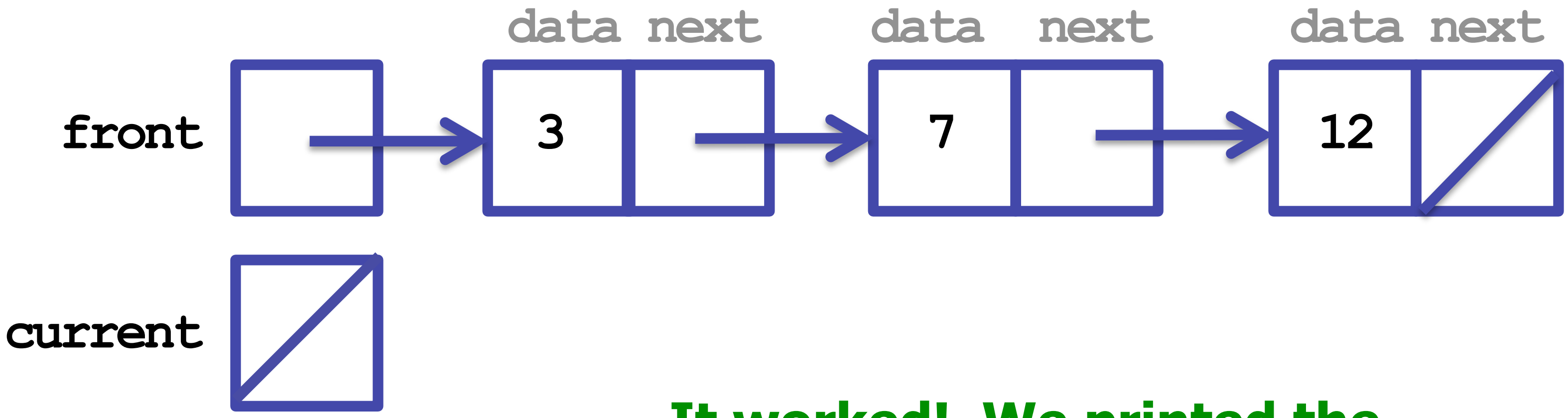
- Output:

```
3
7
12
```

Basics of Linked List Loops

- Let’s update our code and follow it loop-by-loop:

```
ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```



- Output:

```
3
7
12
```

It worked! We printed the list and didn’t destroy it in the process.

Relationship to Array Code

- If we had written the same kind of code for arrays, it would look like this:

```
int i = 0;
while (i < size) {
    System.out.println(elementData[i]);
    i++;
}
```

Relationship to Array Code

- A table explaining this relationship:

Description	Array Code	Linked List Code
go to front of list	`int i = 0;`	`ListNode current = front;`
test for more elements	`i < size`	`current != null`
get current value	`elementData[i]`	`current.data`
go to next element	`i++;`	`current = current.next;`

- This may be helpful if you are comfortable with arrays

For Loops

- Of course, we usually write the array code in a for loop:

```
    for (int i = 0; i < size; i++) {
        System.out.println(elementData[i]);
    }
```

- And we can still do this with the linked list code:

```
    for (ListNode current = front; current != null;
current = current.next) {
        System.out.println(current.data);
    }
```

- But I prefer using while loops with linked lists
  - the choice is yours

LinkedIntList

- **`LinkedIntList`** will have the exact same functionality as **`ArrayIntList`**:

```
add(int value)
add(int index, int value)
get(int index)
indexOf(int value)
remove(int index)
size()
toString()
```

- But it will be implemented with a linked list instead of with an array

LinkedIntList

- What data fields do we need?
  - at a bare minimum, we need the front of the list
  - we could also have others, like the size and the back of the list

- We're going to choose the bare minimum

- Code:

```
public class LinkedIntList {
    private ListNode front;

    ...
}
```

`ListNode` Style: Recap

- Recall that our **`ListNode`** class has public fields
  - instead of private fields with public methods

- Normally this is bad style. However, the client does *not* interact with our **`ListNode`** when using our **`LinkedIntList`**
  - they still get the nice interface of **`LinkedIntList`**'s methods

- So the client will never know the difference

- If we really wanted to write **`ListNode`** correctly:
  - we'd make it a **`private static class`** inside **`LinkedIntList`**
  - but because we're not really going to cover **`private static`** inner **`class`**es in this course, we'll keep **`ListNode`** as is

add

- Let’s write the appending add method (**add**)

- To write **add**, we need to get to the end of our list

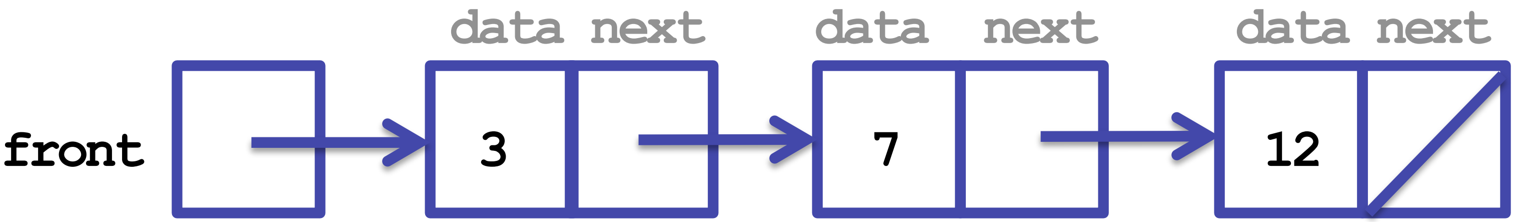
- Here’s a first attempt at getting to the end of our list:

```
ListNode current = front;
while (current != null) {
    current = current.next;
}
```

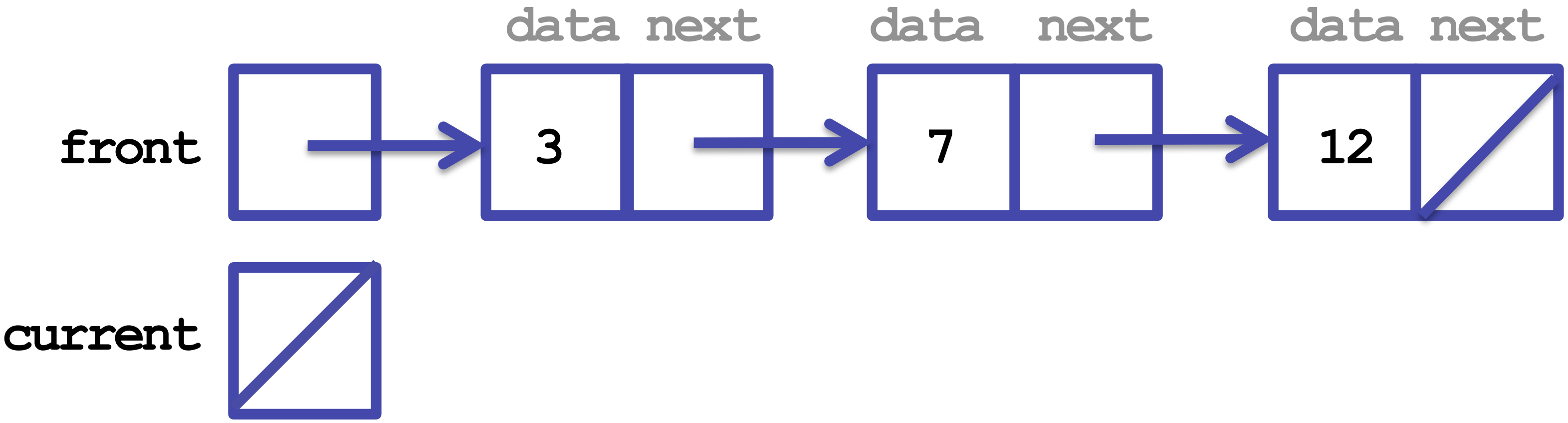
- But what’s wrong with this?

add

- Suppose we originally had a list of 3, 7, and 12:



- After executing our code, we'd have this situation:

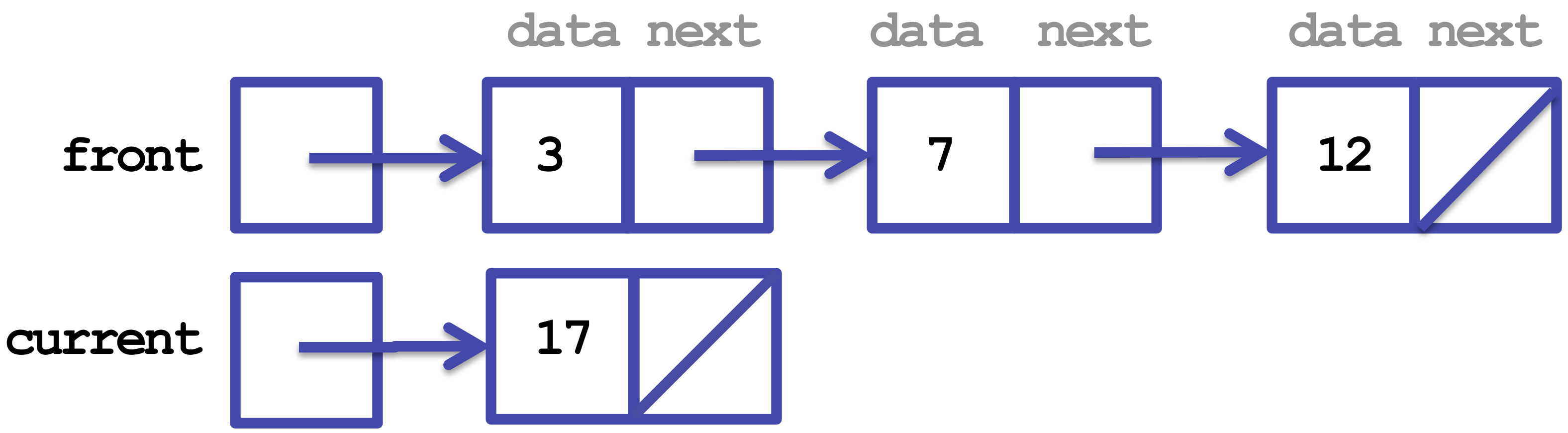


add

- We can try initializing **`current`** to a new node:

```
current = new ListNode(17);
```

- But this code leaves us with this situation:



- We have *not* added 17 to the end of our list
  - we've made a completely separate list instead!

IMPORTANT

- There are *only* two ways to change the structure of a linked list:

1) change the value of **`front`**
   - this changes the starting point of the list
   - example: **`front = null;`**

2) change the value of **`<something>.next`**, where **`<something>`** is a temporary variable that refers to a node in the list
   - this changes a link in the list
   - example: **`current.next = null;`**

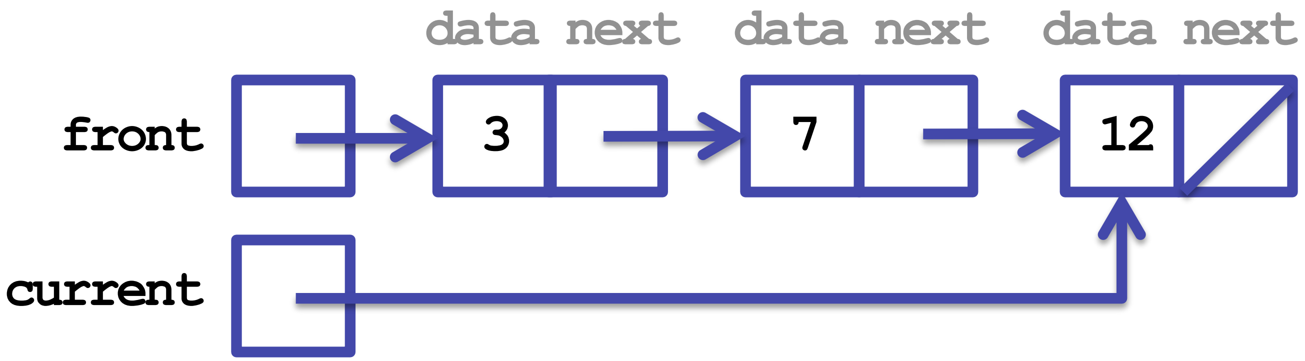
add

- In our first attempt, we fell off the end of the list
  - we continued looping until current was null
- We need to stop at the last node
  - the last node's **`next`** references **`null`**
- Let's update our test:

```
ListNode current = front;
while (current.next != null) {
    current = current.next;
}
```

add

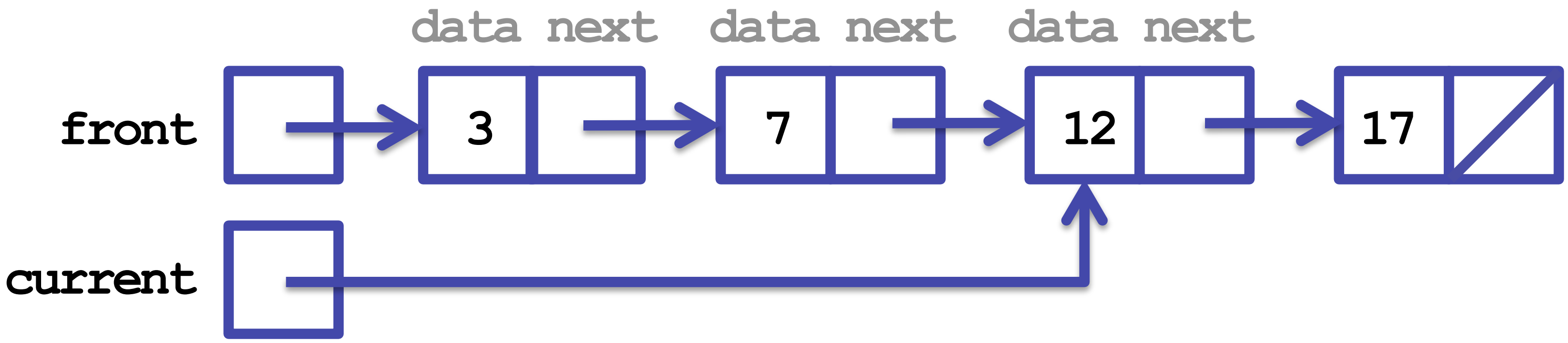
- This code leaves us with this situation:



- And now it's easy to see that this next line of code:

```
current.next = new ListNode(17);
```

- Correctly adds a new node to the list:



add

- Let’s now wrap this code in an actual add method:

```
public void add(int value) {
    ListNode current = front;
    while (current.next != null) {
        current = current.next;
    }
    current.next = new ListNode(value);
}
```

- But what happens if we have an empty list?

`NullPointerException`

- When our list is empty, **`front`** is **`null`**

- Our code sets **`current`** to **`front`** (which is **`null`**) and then asks for the value of **`current.next`**

- But **`current.next`** is the same as writing **`null.next`**

- What is the **`next`** field of **`null`**?
  - there isn't one, because there's no object!

- So Java throws a **`NullPointerException`**
  - you'll see a lot of these as you write linked list code

add

- So we have to make adding the first element to our list a special case:

```
public void add(int value) {
    if (front == null) {
        front = new ListNode(value);
    } else {
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```

- Usually, to change a linked list you'll need at least two cases
  - one for changing the first element, and one for all the others

addSorted

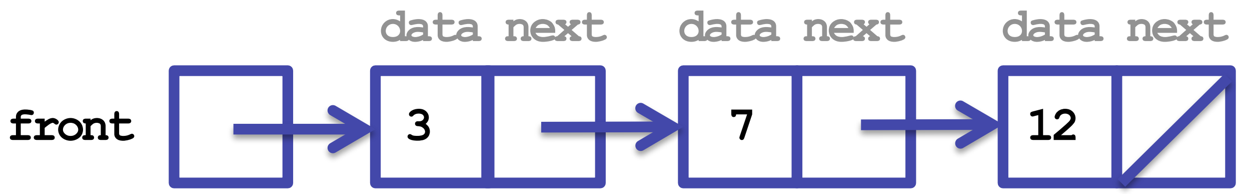
- Let's try something harder: let's write **addSorted**

- **addSorted** is just like the add method of **SortedIntList**:

```
// pre : list is in sorted (non-decreasing) order
// post: given value is inserted into list so as
//       to preserve sorted order
public void addSorted(int value) {
    ...
}
```

addSorted

- Assume we have a list containing 3, 7, and 12:



- Let’s try to write general code for adding a 10 to the list

- We need to stop one node early to change the link:

```
ListNode current = front;
while (current.next.data < value) {
    current = current.next;
}
```

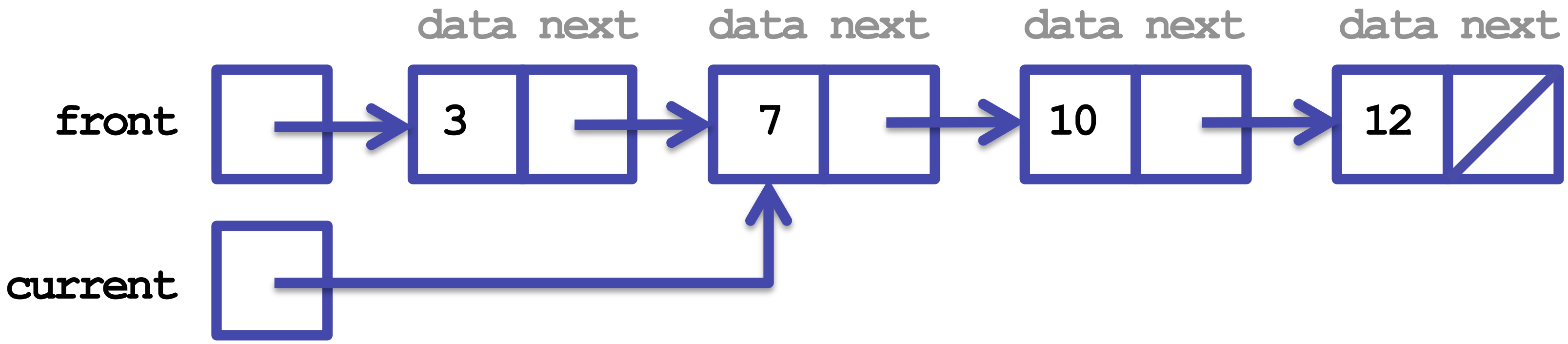
Continue looping until the next value in the list is >= to the value we want to insert

addSorted

- Now we need to insert our new node:

```
current.next = new ListNode(value, current.next);
```

- Which modifies our list to look like this:



- Some people prefer to use a temporary variable when inserting a new node into a list:

```
ListNode temp = new ListNode(value, current.next);
current.next = temp;
```

addSorted

- What if we try to use our code to add 13?
  - our loop test will continue forever!
  - or until `current.next` is `null`, which will make `current.next.data` throw a `NullPointerException`

- We can modify our loop test to check for this:

```
while (current.next != null && current.next.data < value)
```

- This works because the && operator short-circuits
  - this means if the first test is false, it won’t try the second test

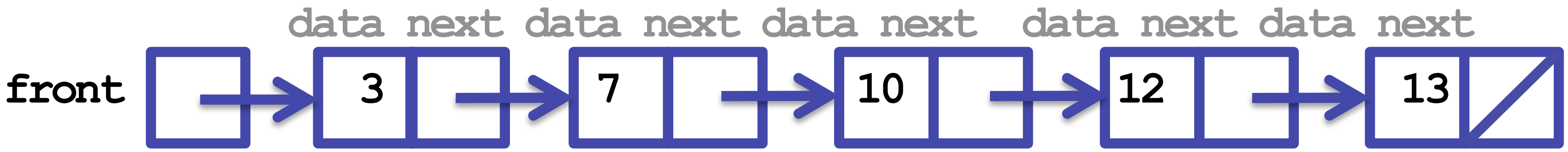
- Notice that the order of the loop test is important!
  - we can’t switch the tests. Why not?

addSorted

- So we can update our **addSorted** code:

```
public void addSorted(int value) {
    ListNode current = front;
    while (current.next != null && current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value, current.next);
}
```

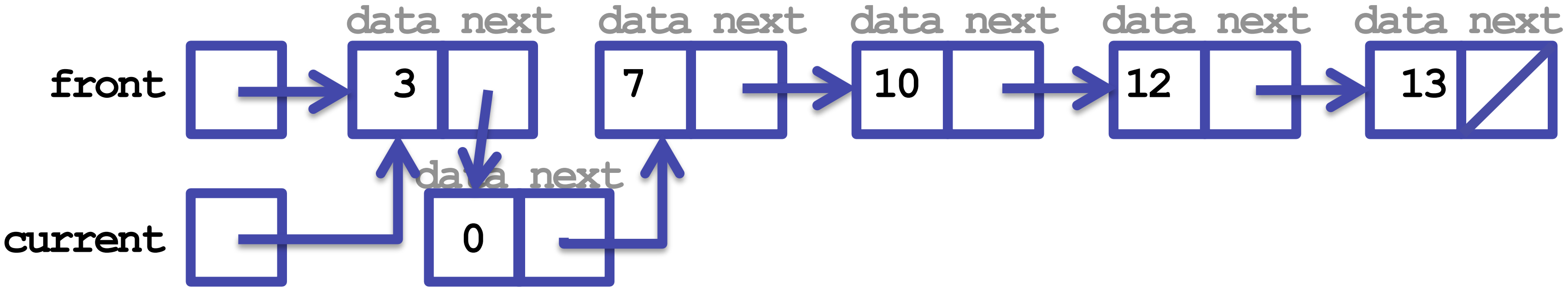
- And now we can successfully add 13 to the end:



- What happens if we try to add a 0 to our list?

addSorted

- If we try to add a 0, we add it in the wrong place:



- We need special code to add an element at the front:

```
front = new ListNode(value, front);
```

- And we need to know when to execute the above add code:

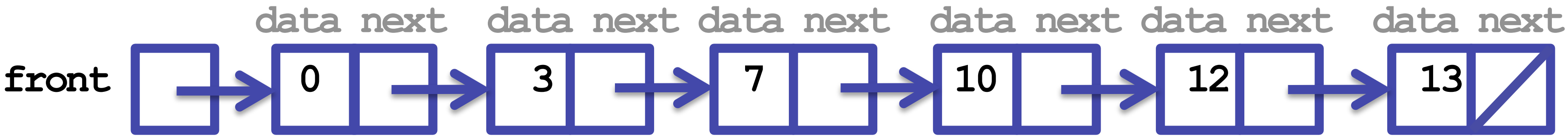
```
if (value <= front.data) {
    // add at front
}
```

addSorted

- Let’s update our **addSorted** code:

```
public void addSorted(int value) {
    if (value <= front.data) {
        front = new ListNode(value, front);
    } else {
        ListNode current = front;
        while (current.next != null && current.next.data < value) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}
```

- And now we can successfully add 0 to the front:



addSorted

- What happens if the list is empty when we call **addSorted**?

- When we have an empty list, **front** is **null**
  - our first line of code asks for **front.data**
  - **NullPointerException**!

- We need to update the first test to be more robust:

```
if (front == null || value <= front.data)
```

- Just like the && operator, the || operator also short-circuits
  - so, if **front** is **null**, we simply insert at the **front**
  - if **front** isn't **null**, we still check **front.data** to decide if we're still going to insert at the front

addSorted

- The final, correct version of **addSorted**:

```
public void addSorted(int value) {
    if (front == null || value <= front.data) {
        front = new ListNode(value, front);
    } else {
        ListNode current = front;
        while (current.next != null && current.next.data < value) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}
```

- That was surprisingly hard! It had four possible cases:
  - empty list
  - value <= [all values in list]
  - [some value in list] < value <= [some value in list]
  - value > [all values in list]