Linked Lists

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

Array-Based List Review

- Array-based lists are what we've studied so far
  - `ArrayIntList`, `ArrayList`, `SortedIntList` all use arrays

- Arrays use a contiguous block of memory

- This means all elements are adjacent to each other

0	1	2	3	4	5	6	7	8
6	2	5	3	7	1	4	-9	-8

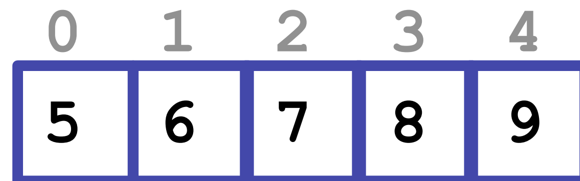
Advantages and Disadvantages

- Advantages of array-based lists
  - random access: can get *any* element in the entire array quickly
    - kind of like jumping to any scene on a DVD (no fast-forwarding required)
- Disadvantages of array-based lists
  - can't insert/remove elements at the front/middle easily
    - have to shift the other elements
  - can't resize array easily
    - have to create a new, bigger array

Linked Lists

- A linked list is a type of list

- But instead of a contiguous block of memory, like this:

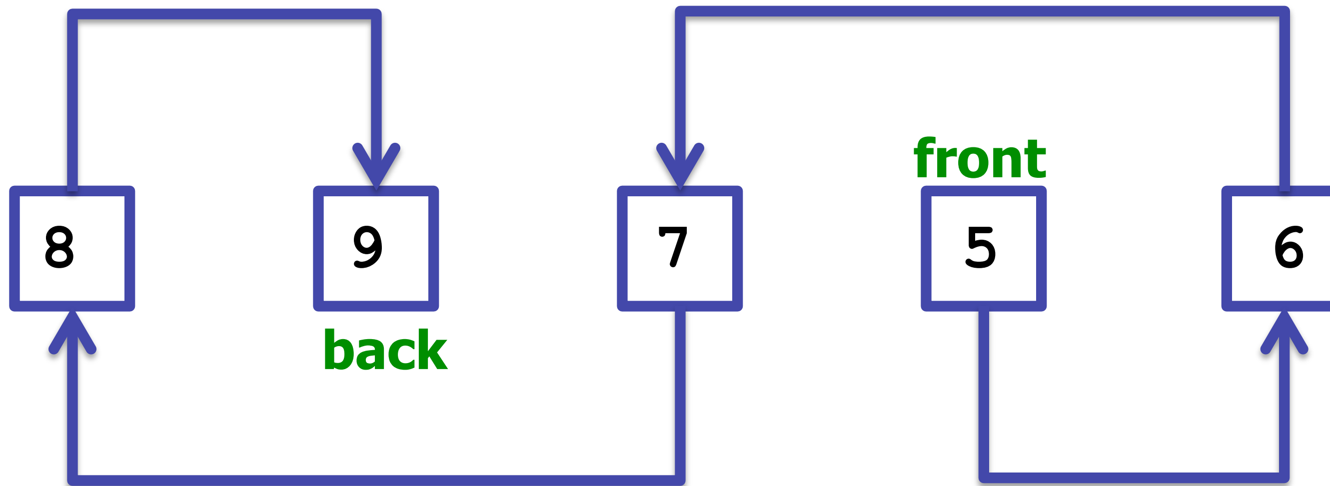


- Linked list elements are scattered throughout memory:

- But now the elements are unordered. How do linked lists keep track of everything?

Linked Lists

- Each element must have a reference to the next element:



- Now, so long as we keep track of the first element (the front), we can keep track of all the elements

Linked Lists

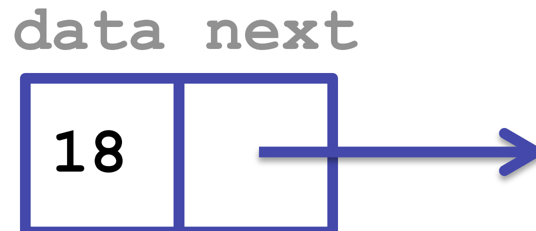
- These references to the next element mean that linked lists have sequential access
- This means that to get to elements in the middle of the list, we must first start at the front and follow all the links until we get to the middle:
  - kind of like fast-forwarding on a VHS tape
  - so getting elements from the middle/back is slow
- Linked lists also do some things well:
  - linked lists can insert elements quickly (no “shifting” needed)
  - linked lists can always add more elements (no set capacity)
- So there are tradeoffs between array lists and linked lists

List Nodes

- **List node**: an element in a linked list
  - an individual list nodes is very simple, but multiple list nodes can be used to build complex structures

- Each list node contains:
  - a piece of data
  - a reference to the next list node

- We typically draw list nodes like this:



`ListNode`

- Code for a simple `ListNode` class containing `int`s as data:

```
public class ListNode {
    public int data;
    public ListNode next;
}
```

- `ListNode` is poorly encapsulated (it has `public` fields)
  - but that's ok for now. We'll talk about it more, later.

`ListNode`

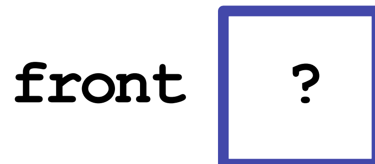
- **`ListNode`** is a recursive data structure
- This means a **`ListNode`** is defined in terms of itself
- A **`ListNode`** contains:
  - data
  - a reference to a **`ListNode`**
- A **`ListNode`** does *not* contain another **`ListNode`**
  - Instead, it contains a *reference* to another **`ListNode`**

Building a Small Linked List

- Let's make a short linked list containing 3, 7, and 12
- First, we need a reference to the front of the linked list:

```
ListNode front;
```

- The variable **`front`** is *not* a **`ListNode`**
  - it is just a variable that can refer to a **`ListNode`**
- We will draw **`front`** like this:



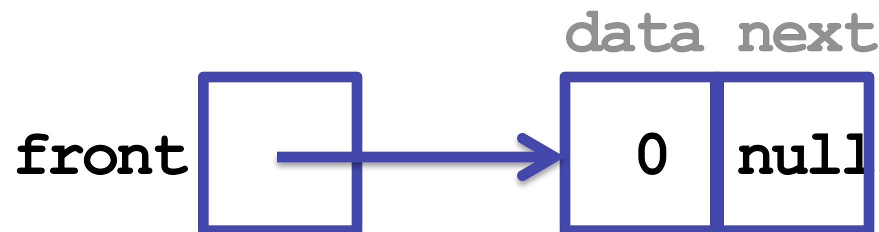
We'll replace the ? with an actual `ListNode` soon

Building a Small Linked List

- To make an actual **`ListNode`**, we must call **`new`**:

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

- This constructs a new node and makes **`front`** refer to it:



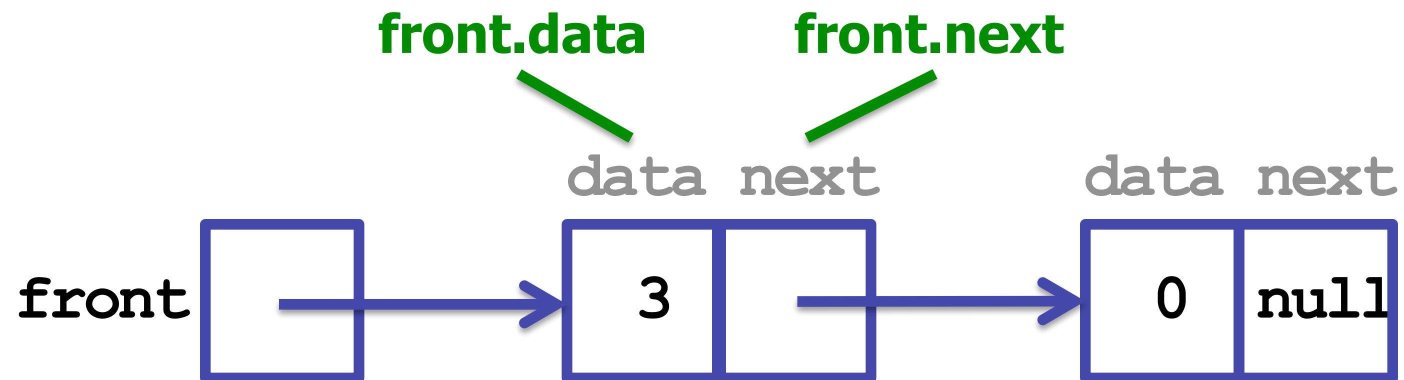
- Notice that Java automatically initialized **`data`** and **`next`** to their zero-equivalents
  - ints: 0
  - objects: null (means "no object")

Building a Small Linked List

- In this first node, we want to store a 3 and have it point to a new node. We can use dot notation for this:

```
front.data = 3;
front.next = new ListNode();
```

- And now our list looks like this:

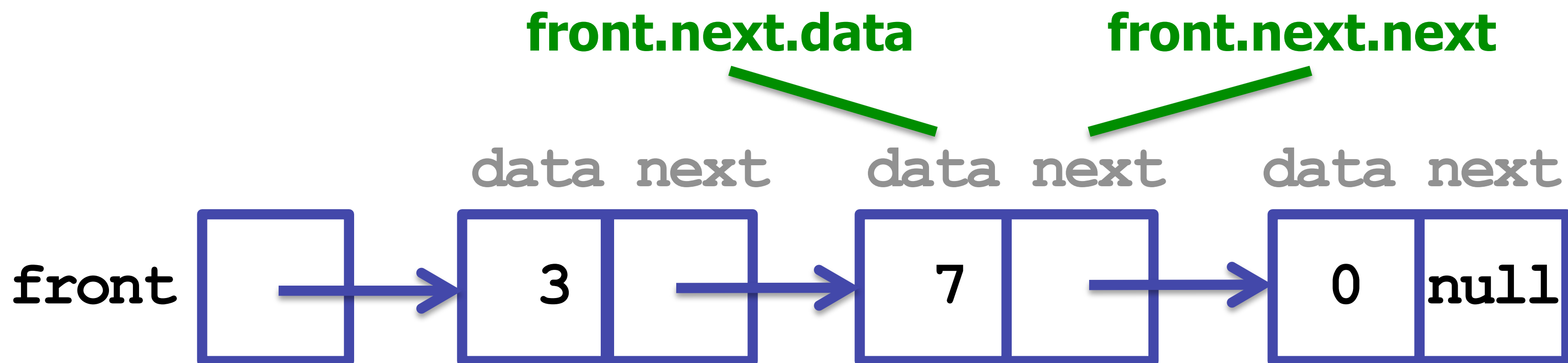


Building a Small Linked List

- In the second node, we want to store a 7 and have it point to a new node:

```
front.next.data = 7;
front.next.next = new ListNode();
```

- And now our list looks like this:

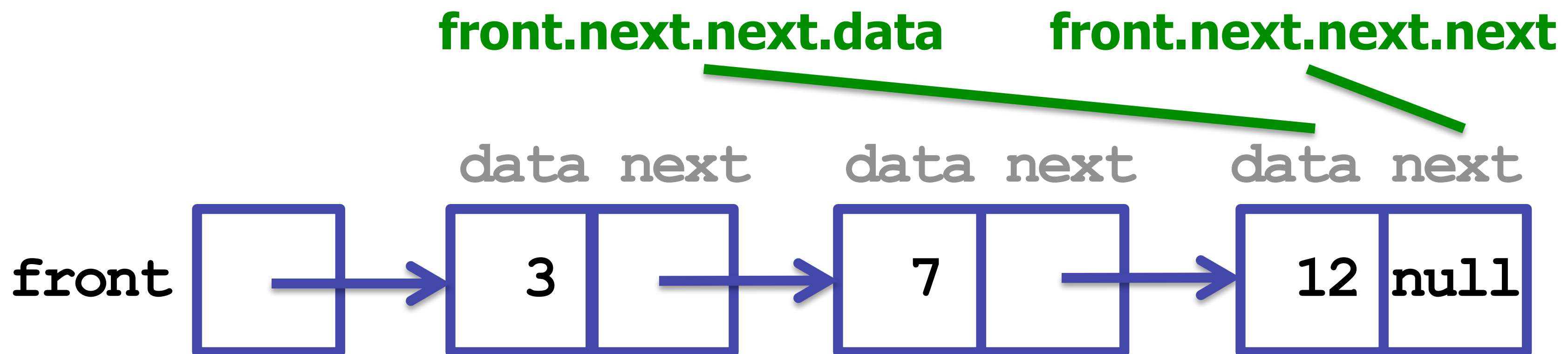


Building a Small Linked List

- In the last node, we want to store a 12 and terminate our list:

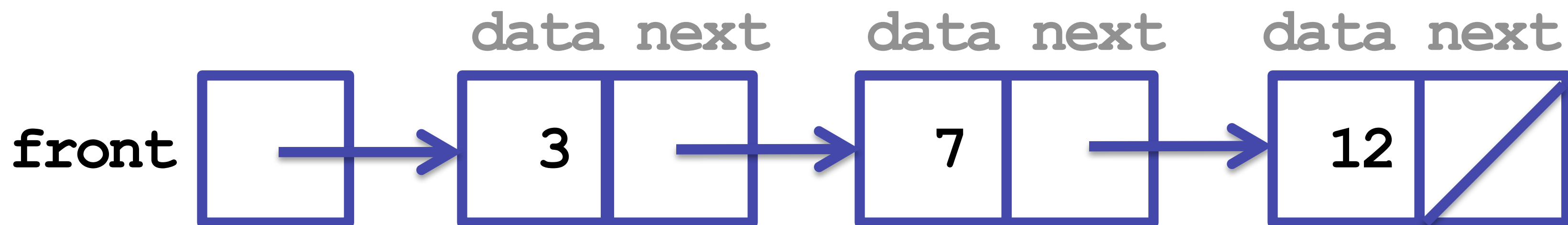
```
front.next.next.data = 12;
front.next.next.next = null;
```

- And now our completed list looks like this:



Building a Small Linked List

- It wasn't strictly necessary to set the last field to **`null`**
- Java had already done it, since **`null`** is a zero-equivalent
- But it's ok to be explicit about this kind of thing
- Also, we normally draw **`null`** as a diagonal line:



Improving `ListNode`

- Let's add some constructors to our **`ListNode`** class:

```
public class ListNode {
    public int data;
    public ListNode next;

    public ListNode() {
        this(0, null);
    }

    public ListNode(int data) {
        this(data, null);
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

Notice we still have one "main" constructor that is called by the other two

Better Linked List Code

- Our new **`ListNode`** constructors allows us to build our list containing 3, 7, and 12 in just one line of code:

```
ListNode front = new ListNode(3, new ListNode(7, new ListNode(12)));
```

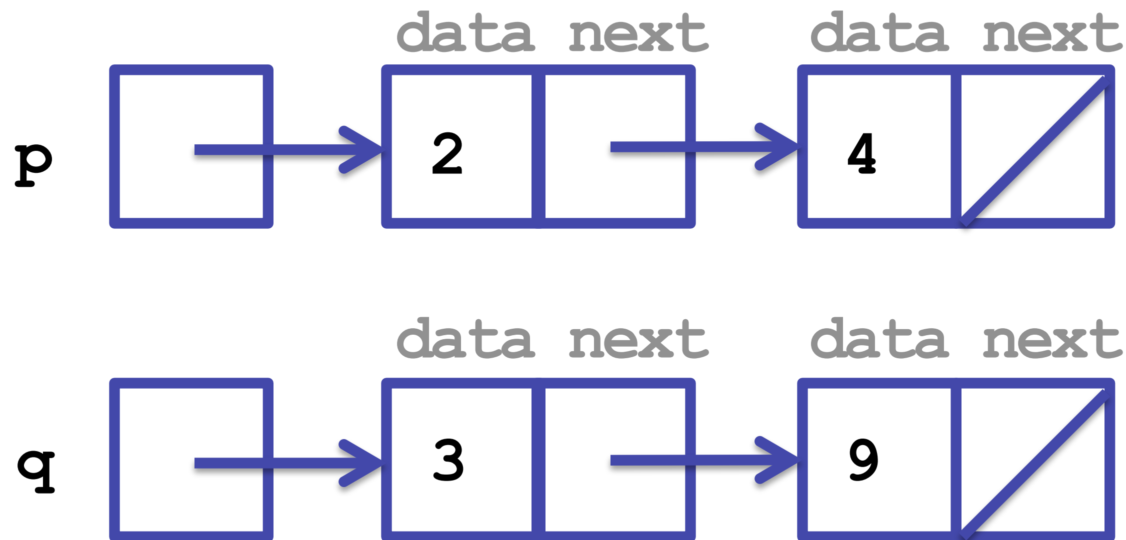
- This is a huge improvement
  - but it's still tedious and error-prone

- There is a better way! We can use loops on linked lists

- ...but we won't for a little longer
  - working with list nodes is challenging, so let's get more practice

Basic Linked List Questions

- Suppose you have two variables of type **`ListNode`** named **`p`** and **`q`**. Consider the following situation:

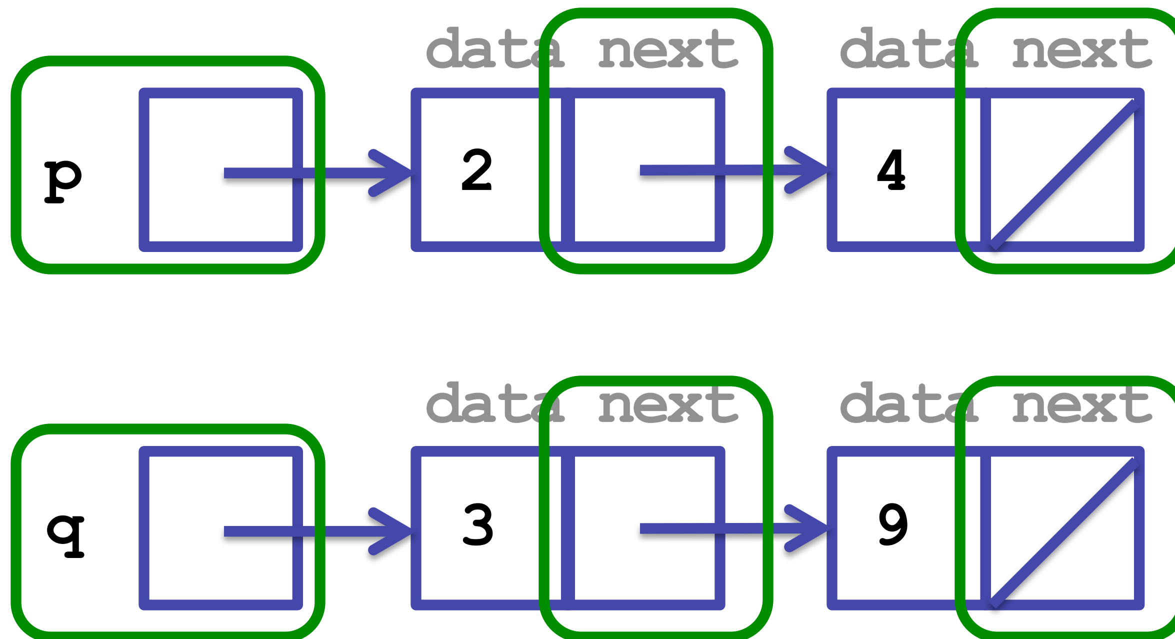


- How many variables of type **`ListNode`** are there?

- How many **`ListNode`** objects are there?

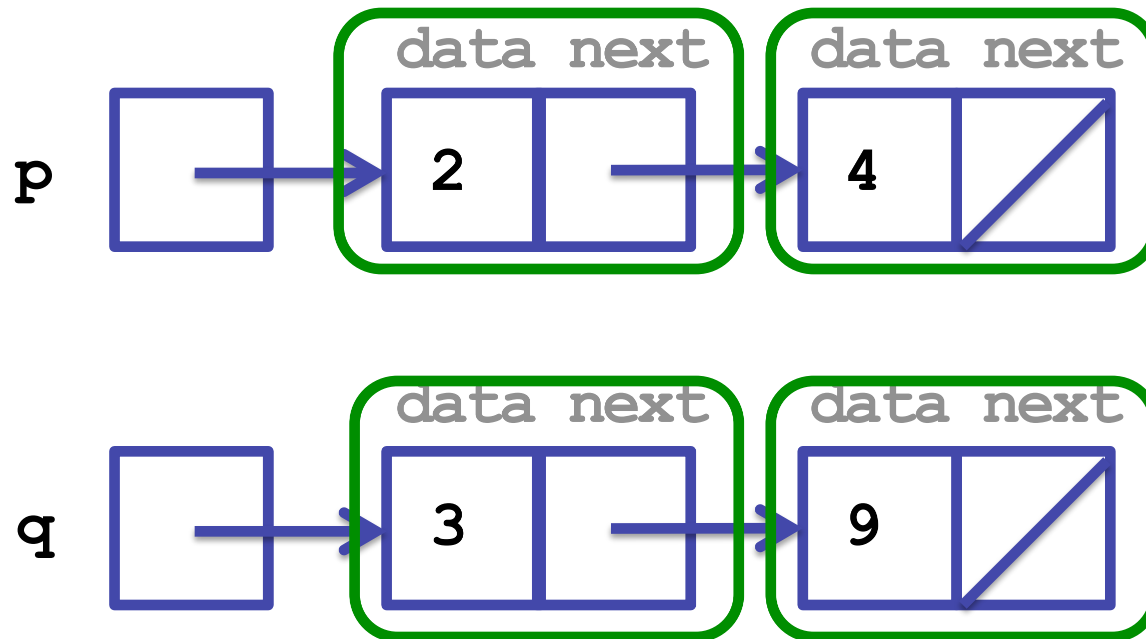
Basic Linked List Questions

- How many variables of type **`ListNode`** are there?
  - **6, circled in green**



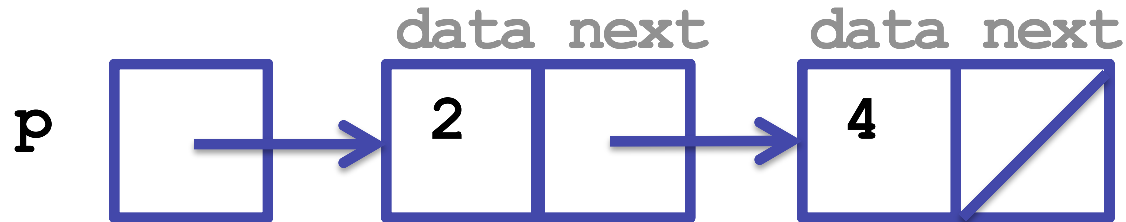
Basic Linked List Questions

- How many **`ListNode`** objects are there?
  - **4, circled in green**

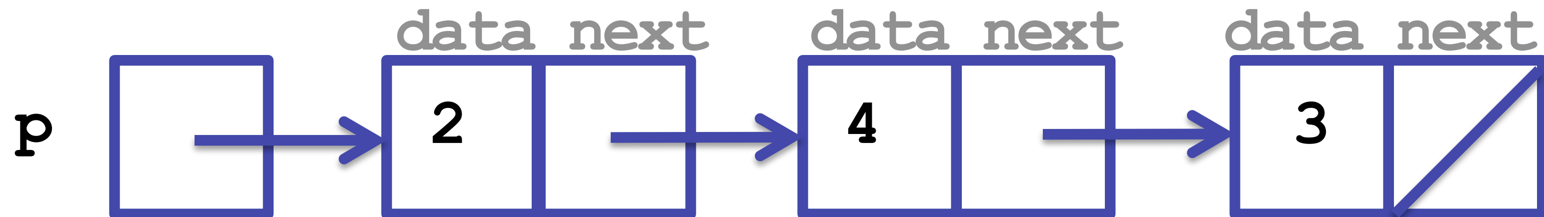


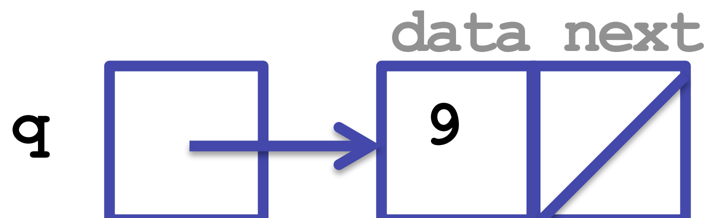
Before/After Problems

- Consider the same situation as before:



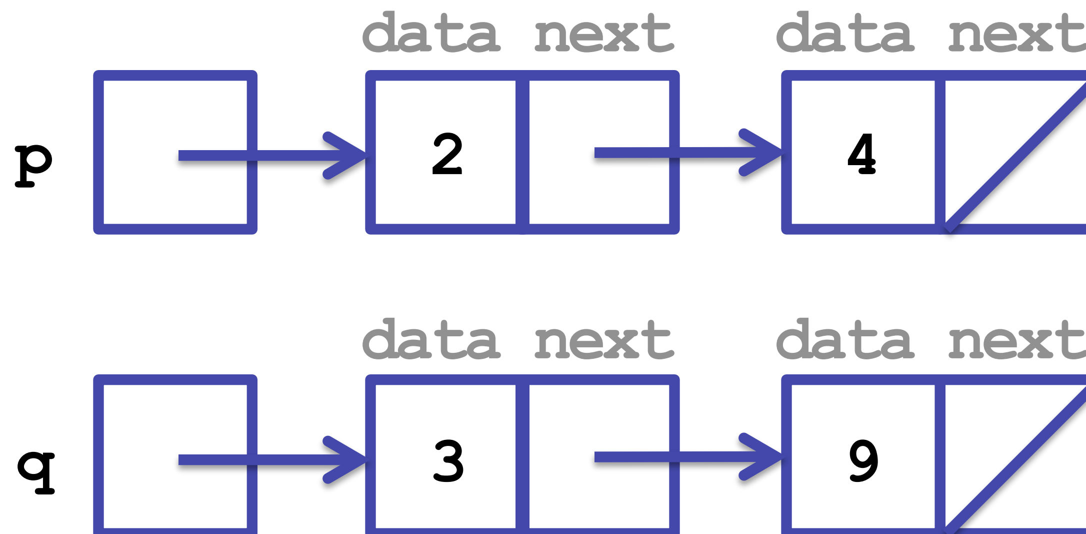
- How would you transform it to the following picture?





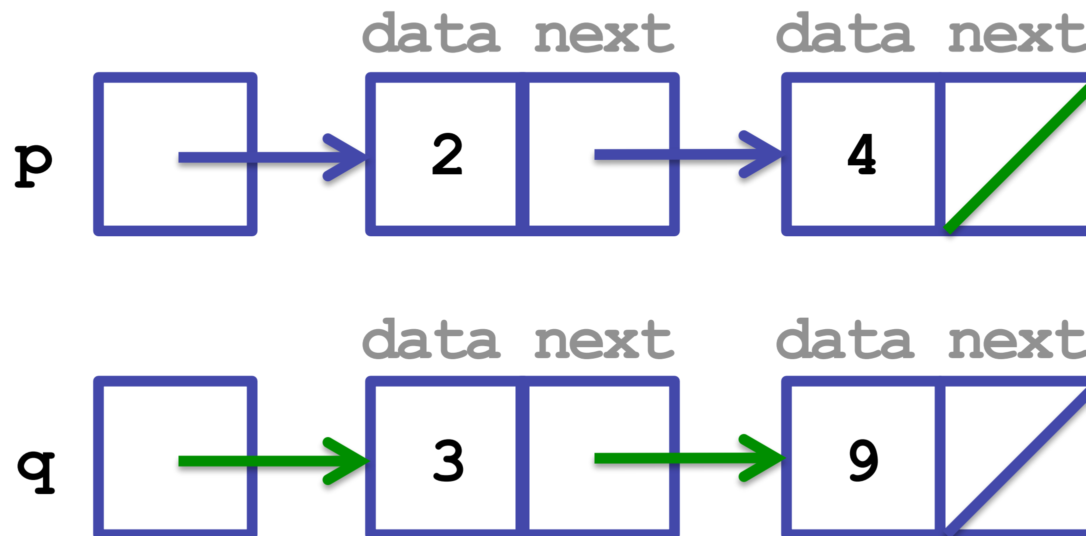
Before/After Problems

- Which variables need to change in order to arrive at the solution?



Before/After Problems

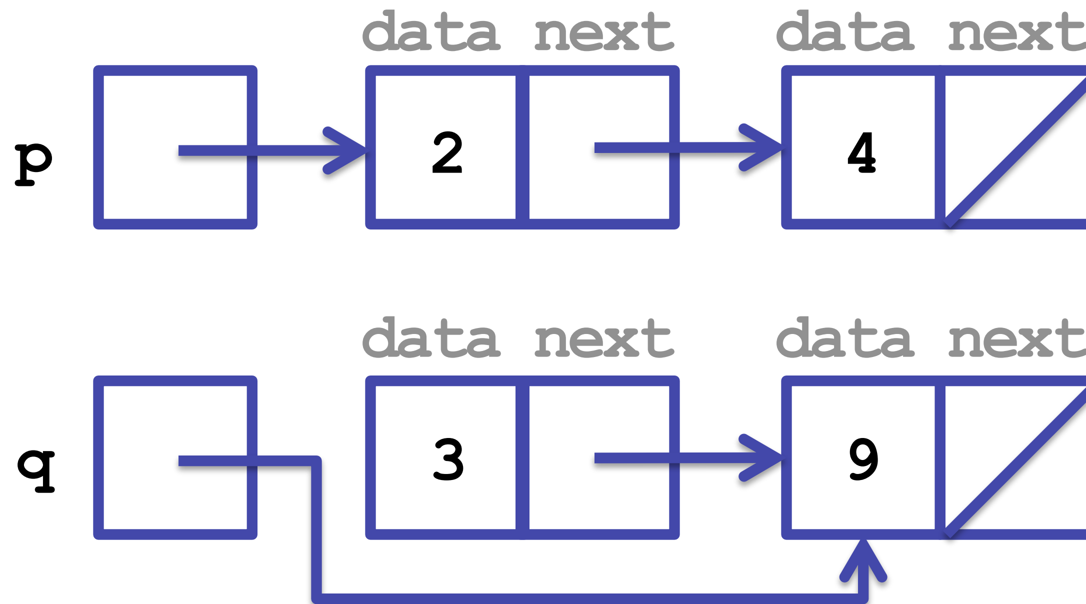
- Which variables/links need to change in order to arrive at the solution?
  - **3, colored green**



Before/After Problems

- But the order we change the links is also important

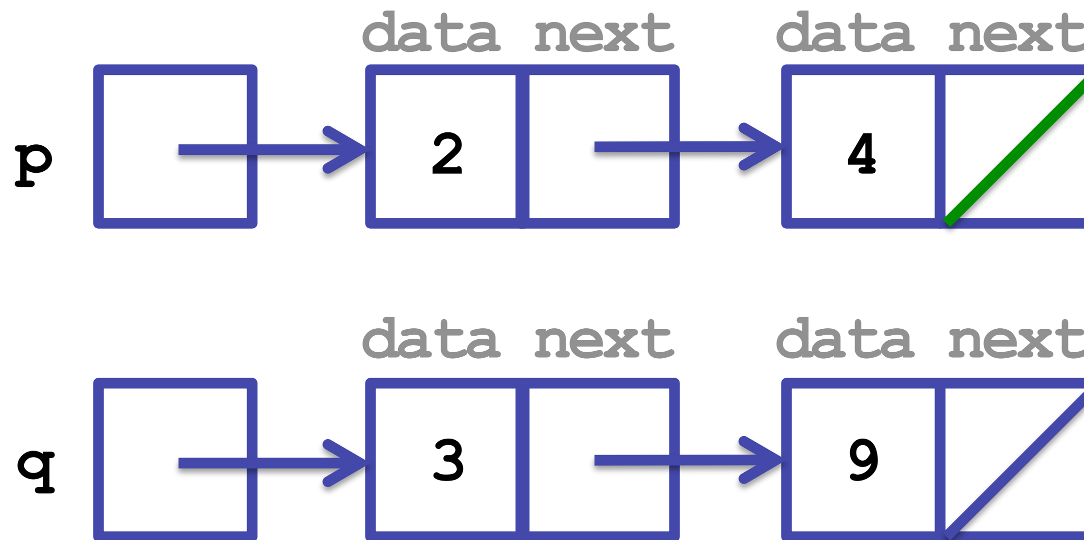
- In the final situation, **`q`** should point to the **`ListNode`** containing the 9. But if we do that first:



- ...we *permanently lose* the **`ListNode`** containing the 3

Before/After Problems

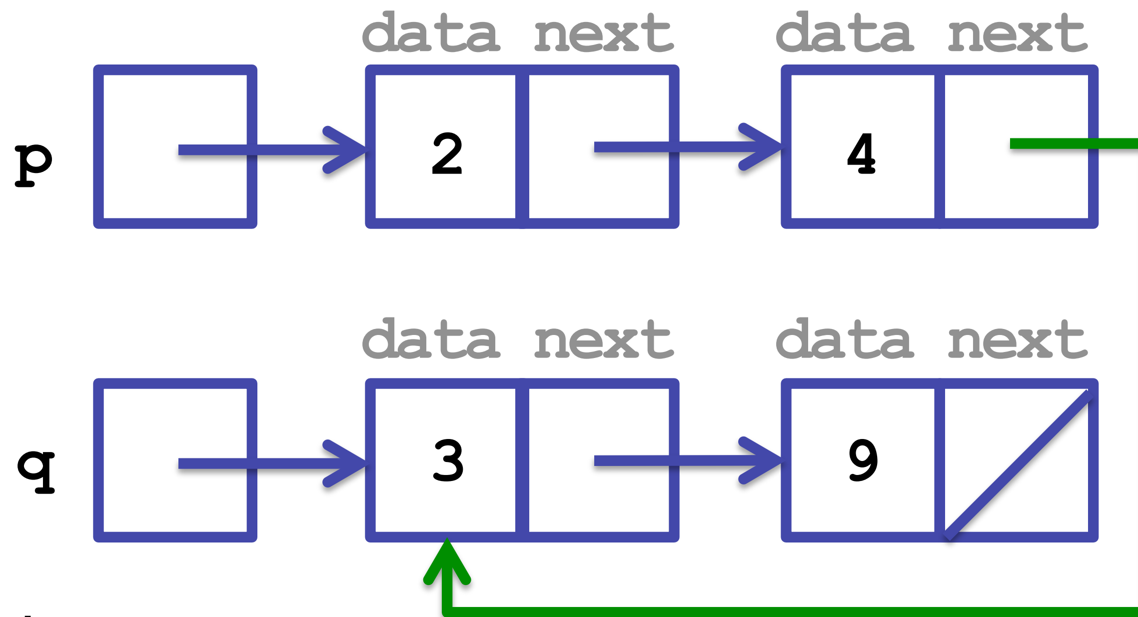
- So how do we actually solve it?



Change this one first. It points to `null`, so we can't lose anything by changing it

Before/After Problems

- Modifying `p.next.next`:

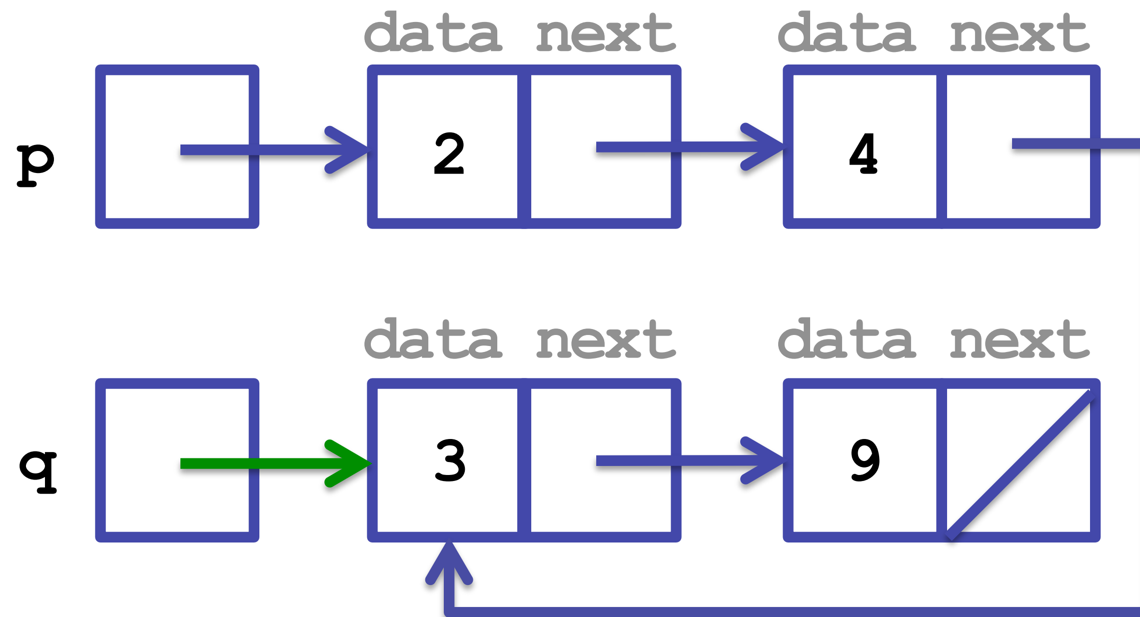


- Code

```
p.next.next = q;
```

Before/After Problems

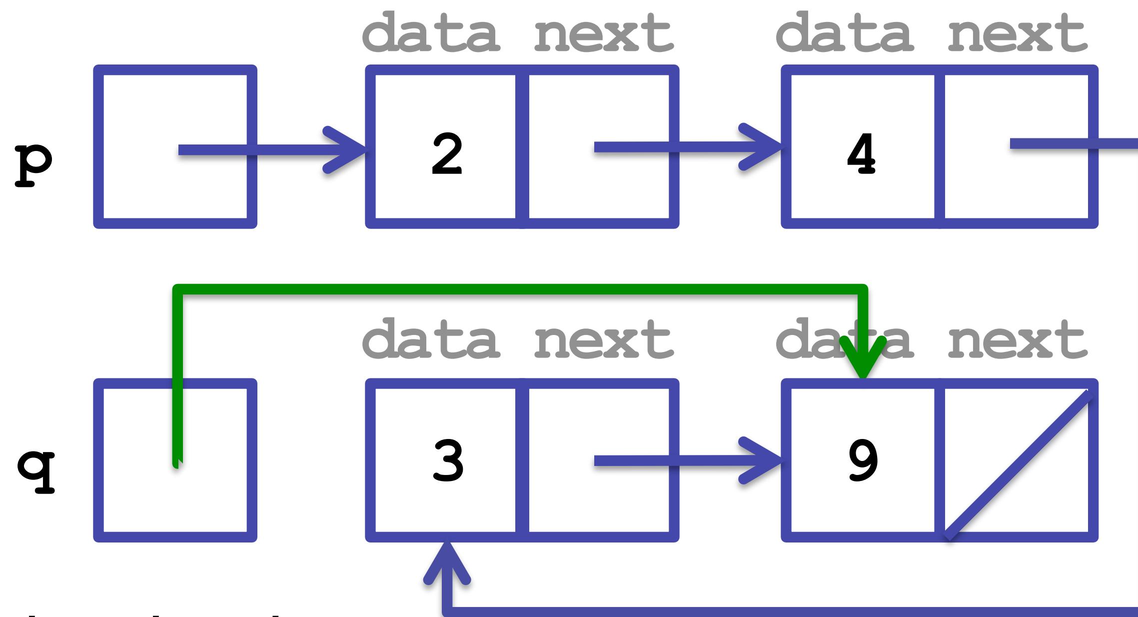
- What's next?



The `ListNode` referred to by `q` is now saved, so we can now safely change what `q` refers to

Before/After Problems

- Modifying **q**:

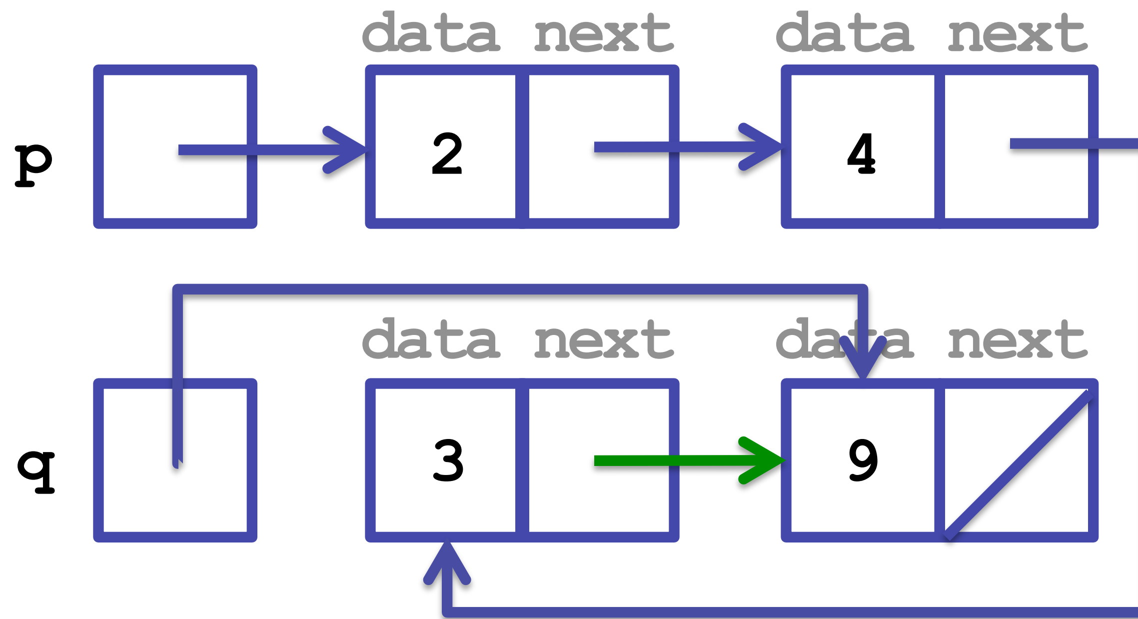


- Updated code

```
p.next.next = q;
q = q.next;
```

Before/After Problems

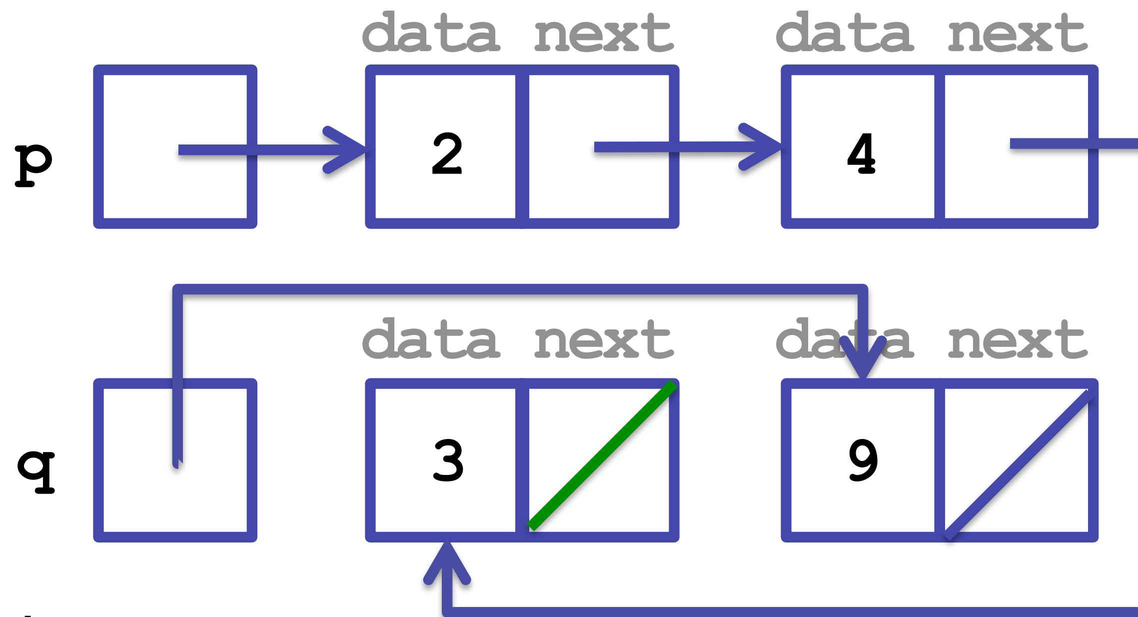
- What's next?



The `ListNode` containing the 3 should refer to `null`. It is now safe to change this link.

Before/After Problems

- Modifying **`p.next.next.next`**:

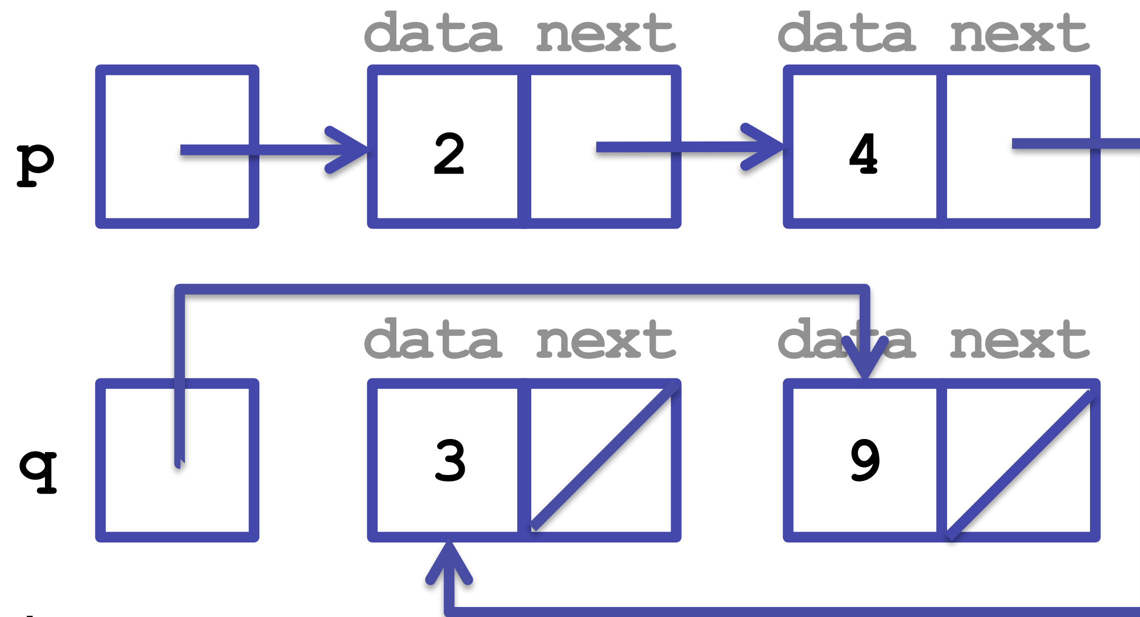


- Code

```
p.next.next = q;
q = q.next;
p.next.next.next = null;
```

Before/After Problems

- We're all done (even though it looks weird)



- Code

```
p.next.next = q;
q = q.next;
p.next.next.next = null;
```

Final Thoughts

- Working with linked lists can be hard
- So draw lots of pictures!
- jGRASP's debugger can also be helpful
  - but remember: you won't have jGRASP on the exams
  - and linked lists are *definitely* on the exams
- Sometimes, solving one of these problems requires a temporary variable:

```
ListNode temp = p;
```

This creates a `ListNode` variable. It does *not* create a new `ListNode` object (no call on `new`).