

Building Java Programs

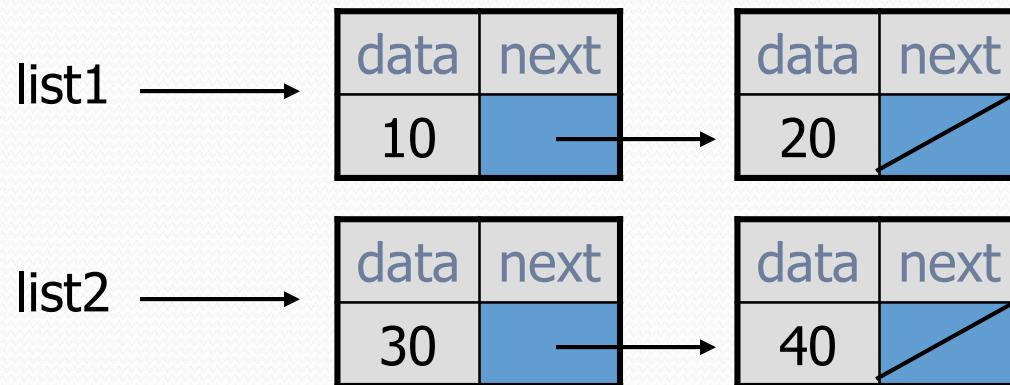
Chapter 16
Linked List Basics

reading: 16.2

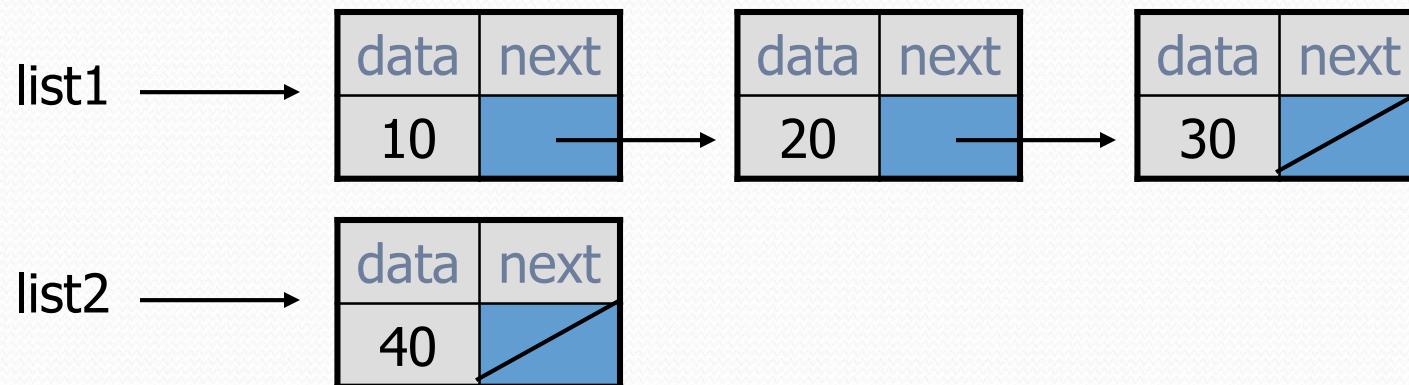


Linked node problem 3

- What set of statements turns this picture:

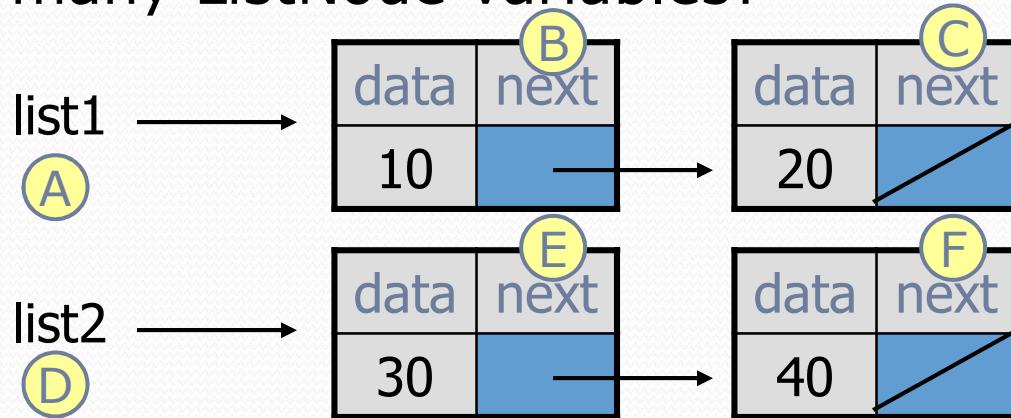


- Into this?

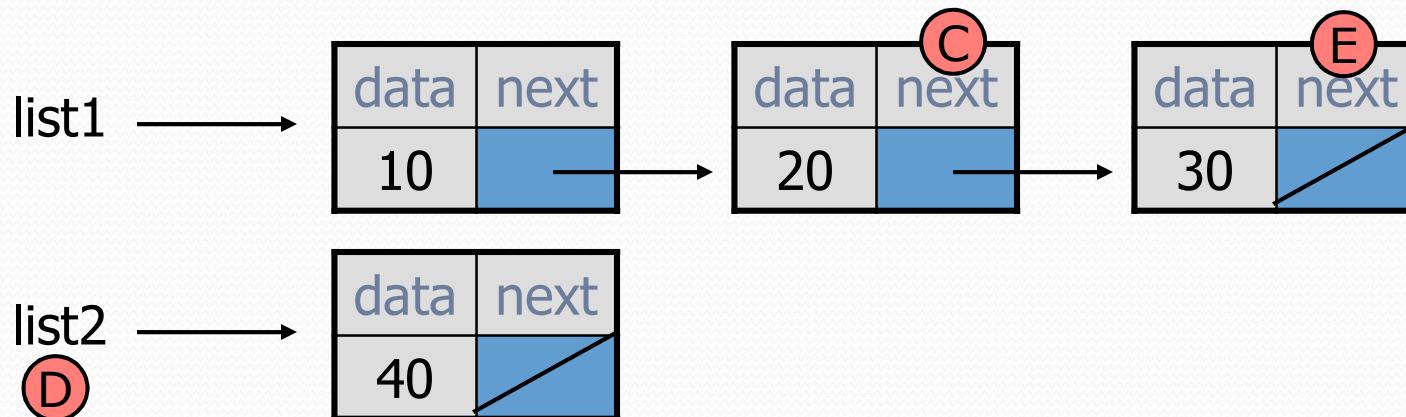


Linked node problem 3

- How many ListNode variables?



- Which variables change?



References vs. objects

variable = **value**;

a *variable* (left side of =) place to put a reference

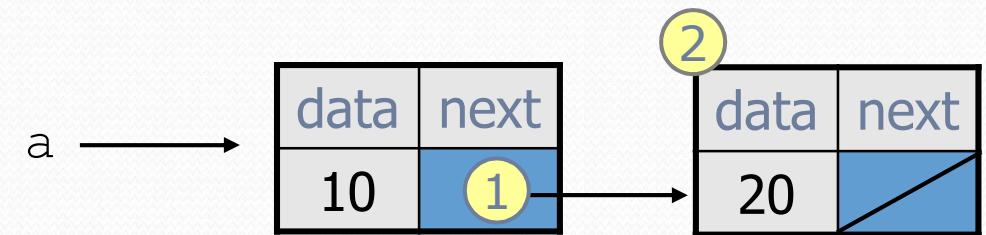
(where the phone number goes; where the base of the arrow goes)

a *value* (right side of =) is the reference itself

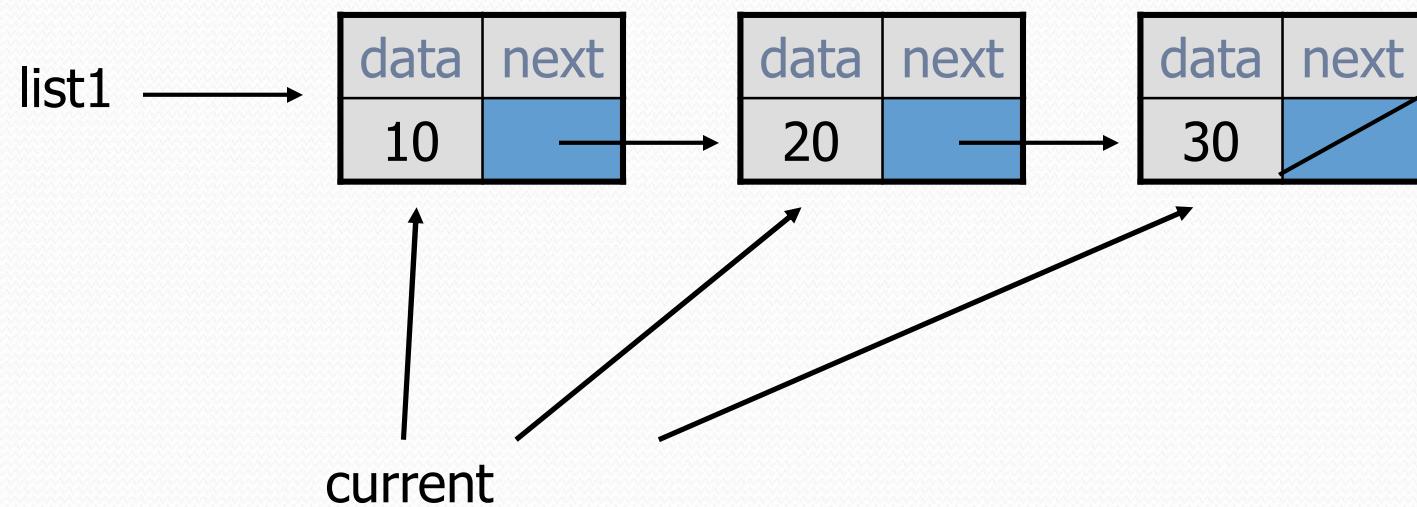
(the phone number; the destination of the arrow)

- For the list at right:

- `a.next = value;`
means to adjust where `a` points

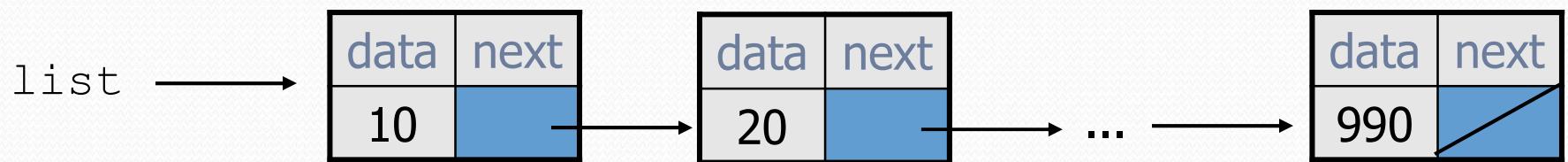


- `variable = a.next;`
means to make `variable` point at



Linked node question

- Suppose we have a long chain of list nodes:

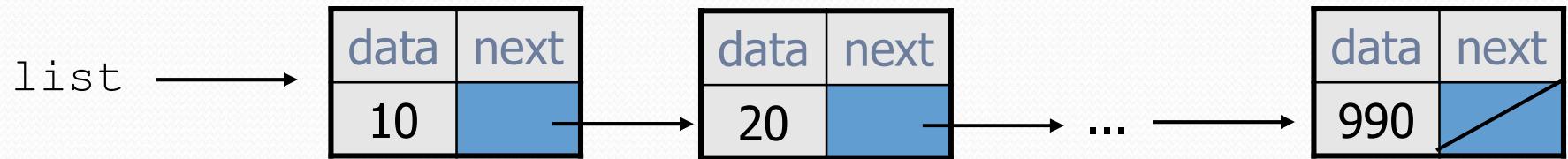


- We don't know exactly how long the chain is.
- How would we print the data values in all the nodes?

Algorithm pseudocode

- Start at the **front** of the list.
- While (there are more nodes to print):
 - Print the current node's **data**.
 - Go to the **next** node.
- How do we walk through the nodes of the list?

```
list = list.next; // is this a good idea?
```



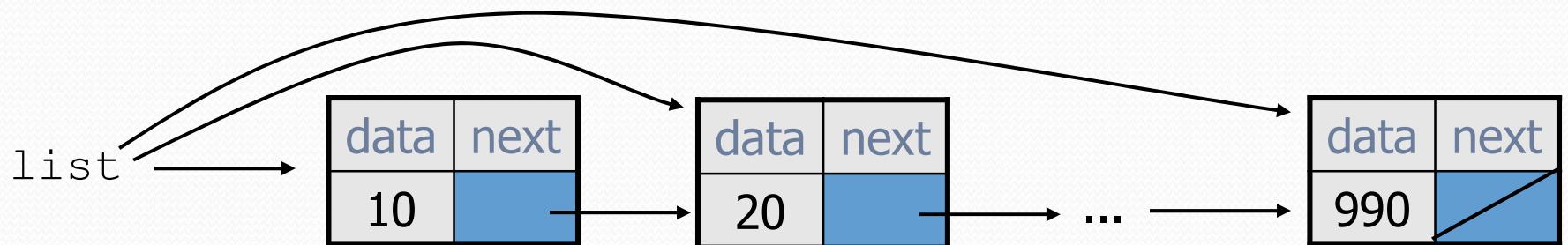
Traversing a list?

- One (bad) way to print every value in the list:

```
while (list != null) {  
    System.out.println(list.data);  
    list = list.next;      // move to next node  
}
```



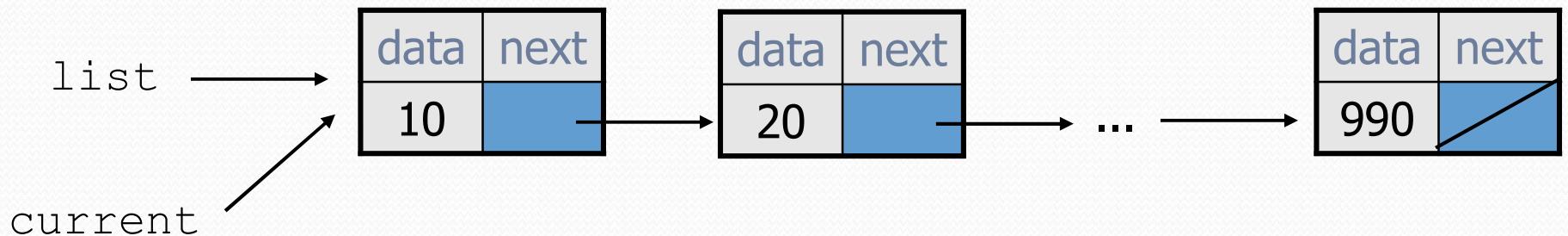
- What's wrong with this approach?
 - (It loses the linked list as it prints it!)



A current reference

- Don't change list. Make another variable, and change it.
 - A ListNode variable is NOT a ListNode object

```
ListNode current = list;
```



- What happens to the picture above when we write:

```
current = current.next;
```

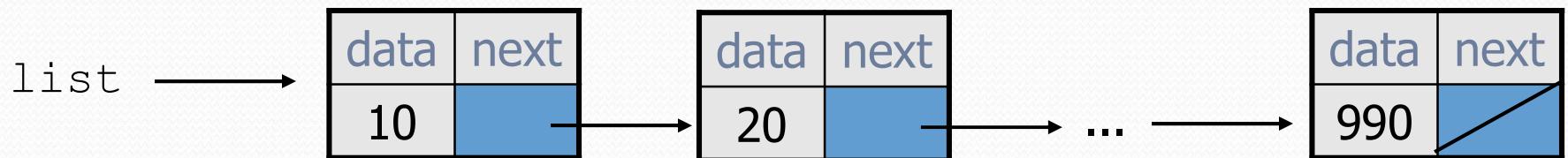
Traversing a list correctly

- The correct way to print every value in the list:

```
ListNode current = list;  
while (current != null) {  
    System.out.println(current.data);  
    current = current.next; // move to next node  
}
```



- Changing current does not damage the list.



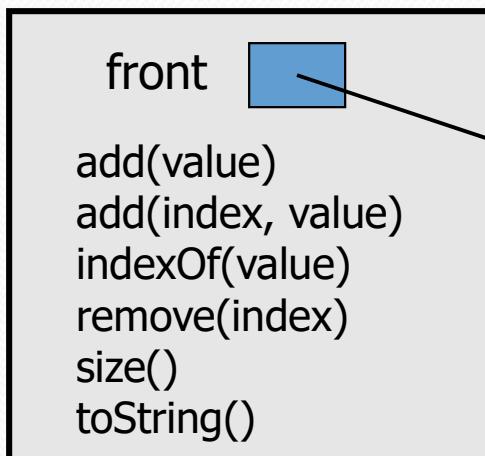
Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
 - Describes *what* a collection does, not *how* it does it
- Java's collection framework describes several ADTs:
 - Queue, List, Collection, Deque, List, Map, Set
- An ADT can be implemented in multiple ways:
 - ArrayList and LinkedList implement List
 - HashSet and TreeSet implement Set
 - LinkedList, ArrayDeque, etc. implement Queue
- The **same** external behavior can be implemented in many different ways, each with pros and cons.

A LinkedList class

- Let's write a collection class named `LinkedList`.
 - Has the same methods as `ArrayList`:
 - `add`, `add`, `get`, `indexOf`, `remove`, `size`, `toString`
 - The list is internally implemented as a chain of linked nodes
 - The `LinkedList` keeps a reference to its `front` as a field
 - `null` is the end of the list; a `null` `front` signifies an empty list

`LinkedList`



`ListNode`

<code>data</code>	<code>next</code>
42	

element 0

`ListNode`

<code>data</code>	<code>next</code>
-3	

element 1

`ListNode`

<code>data</code>	<code>next</code>
17	

element 2

LinkedList class v1

```
public class LinkedList {  
    private ListNode front;  
  
    public LinkedList() {  
        front = null;  
    }  
  
    }  
  
methods go here
```

LinkedList

front = 

Linked List vs. Array

- Print list values:

```
ListNode list= ...;                                int[] a = ...;

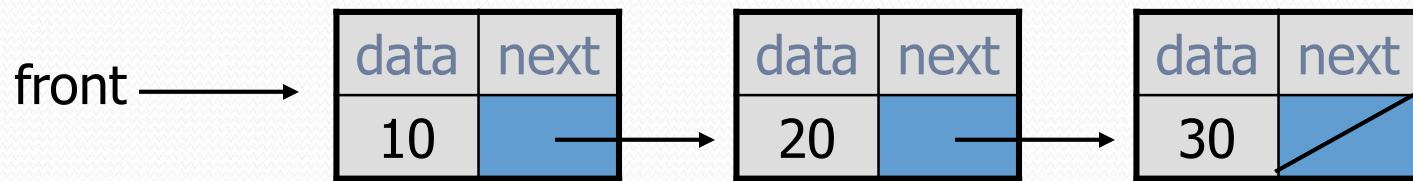
ListNode current = list;                            int i = 0;
while (current != null) {                          while (i < a.length) {
    System.out.println(current.data);           System.out.println(a[i]);
    current = current.next;                     i++;
}
```

- Similar to array code:

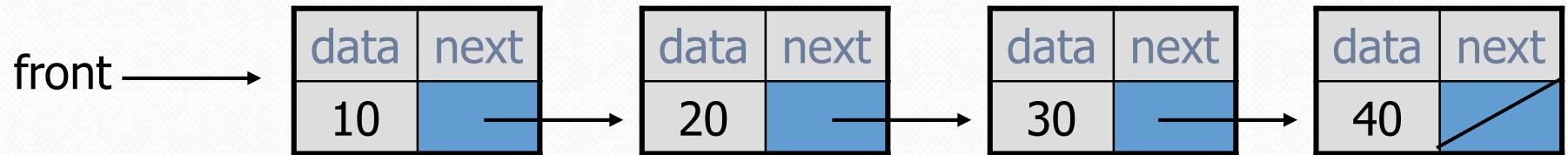
Description	Array Code	Linked List Code
Go to front of list	int i = 0;	ListNode current = list;
Test for more elements	i < size	current != null
Current value	elementData[i]	current.data
Go to next element	i++; // i=i+1	current = current.next;

Before/After

- Before



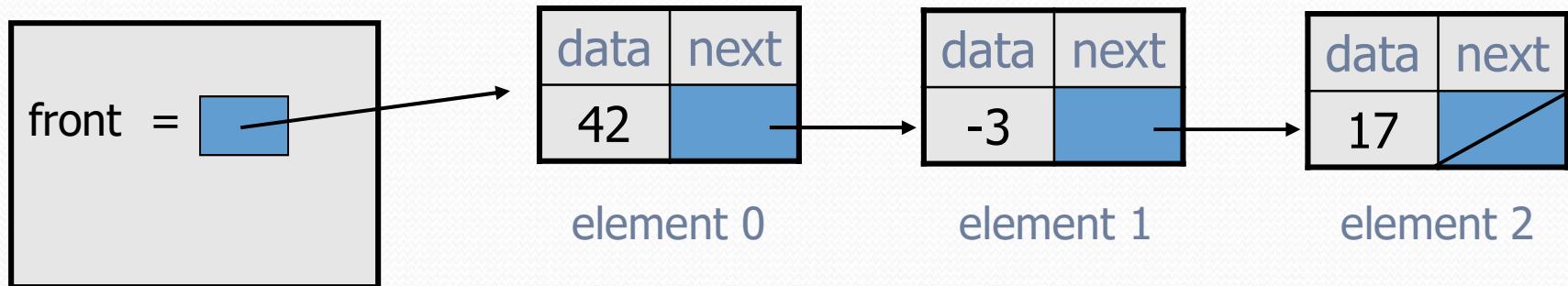
- After



Implementing add

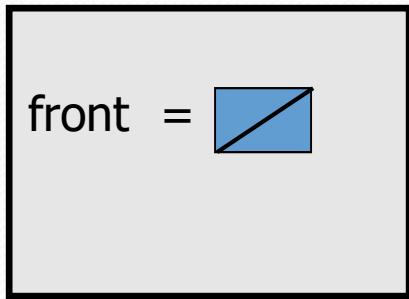
```
// Adds the given value to the end of the list.  
public void add(int value) {  
    ...  
}
```

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?

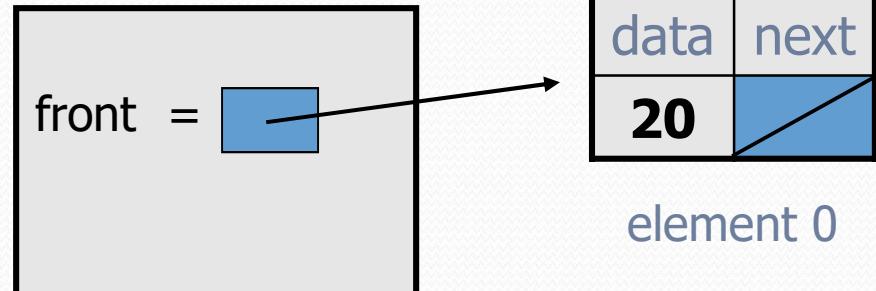


Adding to an empty list

- Before adding 20:



After:



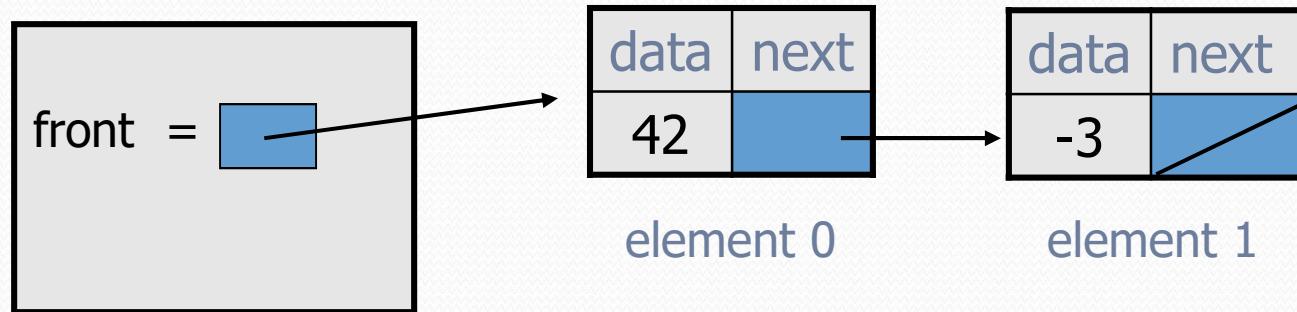
- We must create a new node and attach it to the list.

The add method, 1st try

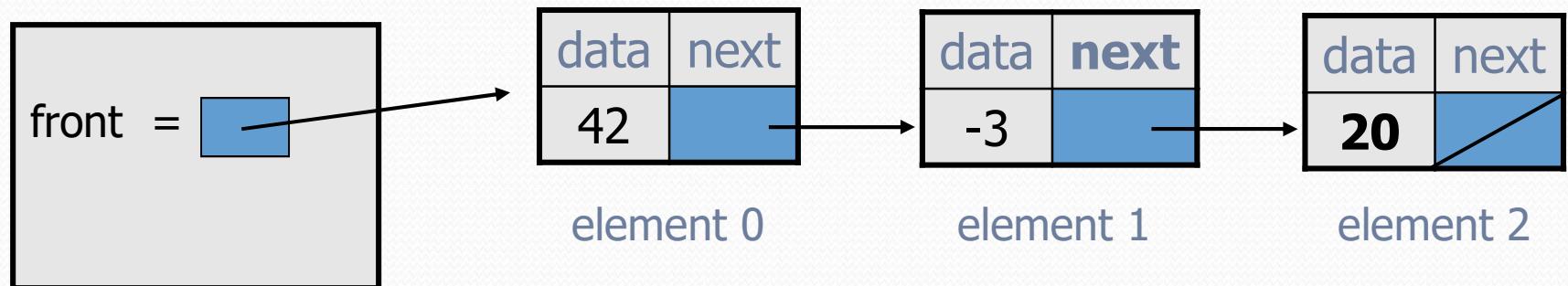
```
// Adds the given value to the end of the list.  
public void add(int value) {  
    if (front == null) {  
        // adding to an empty list  
        front = new ListNode(value);  
    } else {  
        // adding to the end of an existing list  
        . . .  
    }  
}
```

Adding to non-empty list

- Before adding value 20 to end of list:

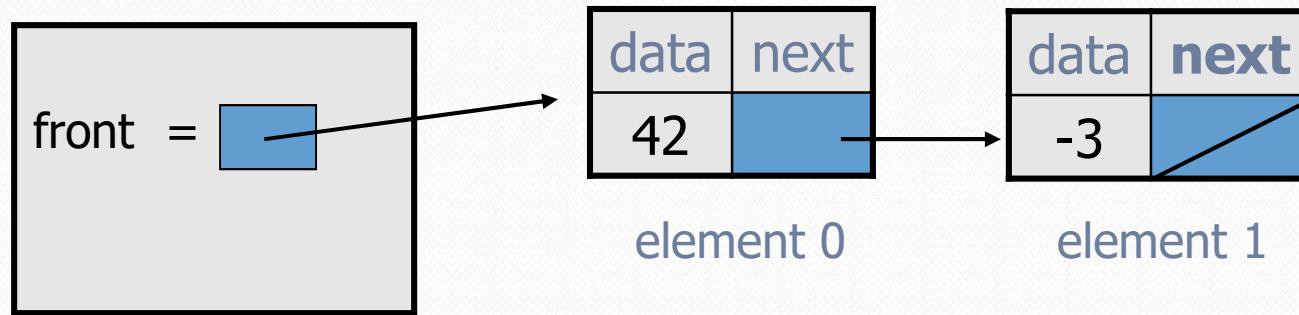


- After:



Don't fall off the edge!

- To add/remove from a list, you must modify the `next` reference of the node *before* the place you want to change.



- Where should `current` be pointing, to add 20 at the end?
- What loop test will stop us at this place in the list?

The add method

```
// Adds the given value to the end of the list.  
public void add(int value) {  
    if (front == null) {  
        // adding to an empty list  
        front = new ListNode(value);  
    } else {  
        // adding to the end of an existing list  
        ListNode current = front;  
        while (current.next != null) {  
            current = current.next;  
        }  
        current.next = new ListNode(value);  
    }  
}
```

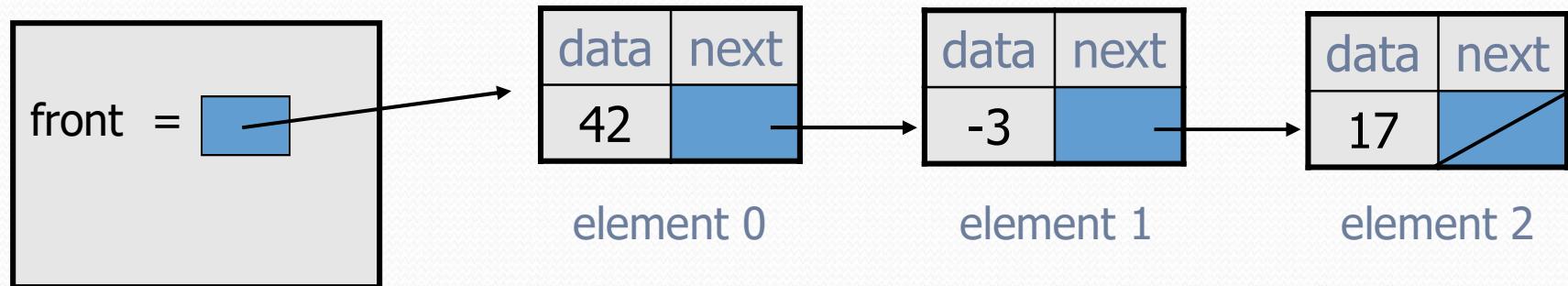
changing a list

- There are only two ways to change a linked list:
 - Change the value of `front` (modify the front of the list)
 - Change the value of `<node>.next` (modify middle or end of list to point somewhere else)
- Implications:
 - To add in the middle, need a reference to the *previous* node
 - Front is often a special case

Implementing get

```
// Returns value in list at given index.  
public int get(int index) {  
    ...  
}
```

- Exercise: Implement the `get` method.



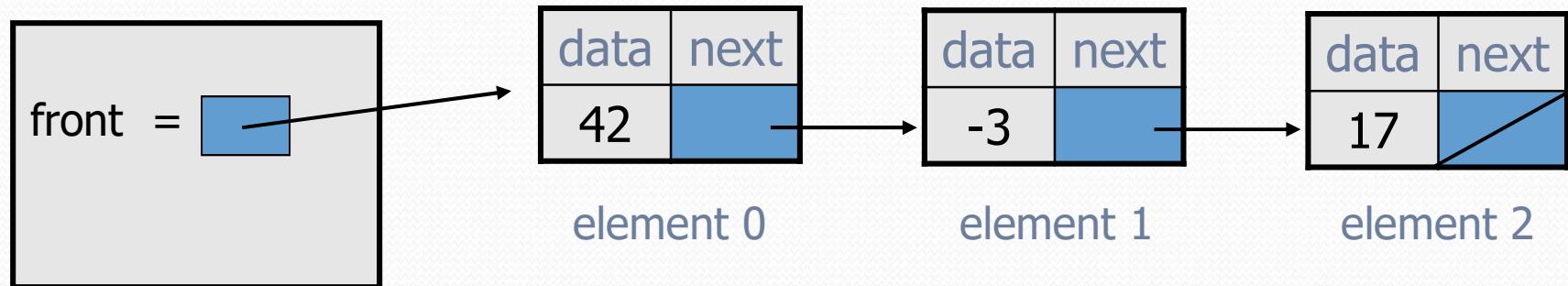
The get method

```
// Returns value in list at given index.  
// Precondition: 0 <= index < size()  
public int get(int index) {  
    ListNode current = front;  
    for (int i = 0; i < index; i++) {  
        current = current.next;  
    }  
    return current.data;  
}
```

Implementing add (2)

```
// Inserts the given value at the given index.  
public void add(int index, int value) {  
    ...  
}
```

- Exercise: Implement the two-parameter `add` method.



The add method (2)

```
// Inserts the given value at the given index.  
// Precondition: 0 <= index <= size()  
public void add(int index, int value) {  
    if (index == 0) {  
        // adding to an empty list  
        front = new ListNode(value, front);  
    } else {  
        // inserting into an existing list  
        ListNode current = front;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = new ListNode(value,  
                                    current.next);  
    }  
}
```