Linked Lists

References to same type

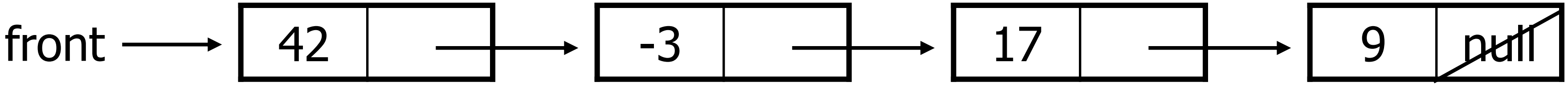
- What would happen if we had a class that declared one of its own type as a field?

```
public class Strange {
    private String name;
    private Strange other;
}
```

  - Will this compile?
    - If so, what is the behavior of the `other` field? What can it do?
    - If not, why not? What is the error and the reasoning behind it?

Linked data structures

- a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
  - `LinkedList`, `TreeSet`, `TreeMap`

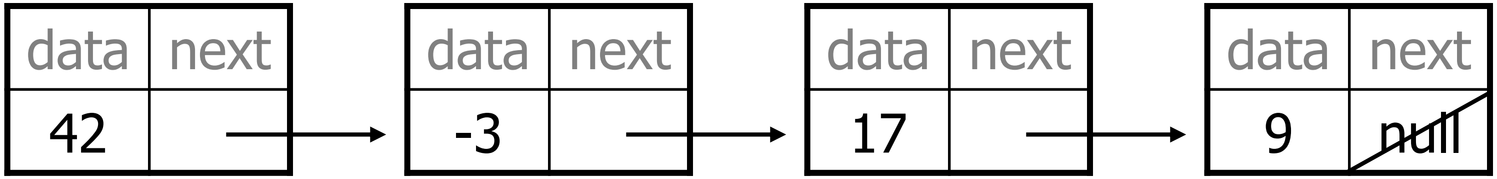


A list node class

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

- Each list node object stores:
  - one piece of integer data
  - a reference to another list node

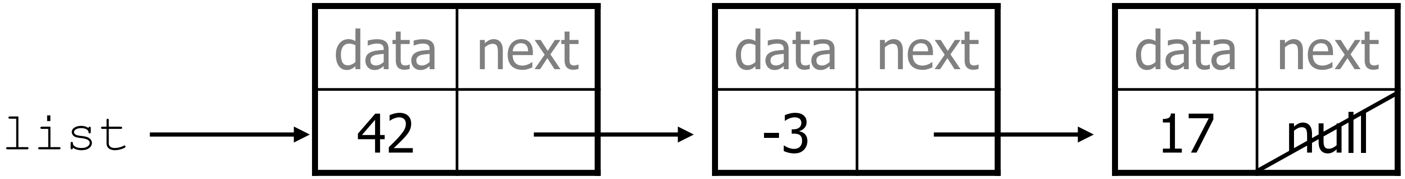
- ListNodes can be "linked" into chains to store a list of values:



data next
42
data next
-3
data next
17
data next
9 ~~null~~

List node client example

```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 17
    }
}
```



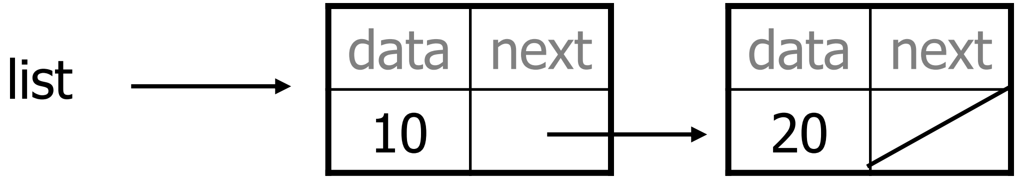
List node w/ constructor

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

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

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

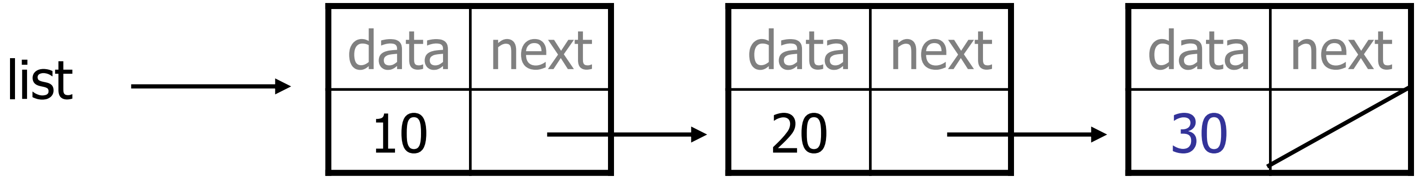
– Exercise: Modify the previous client to use these constructors.

Linked node problem 1

- What set of statements turns this picture:

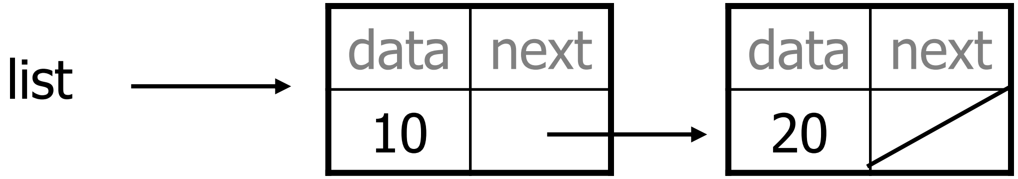


- Into this?

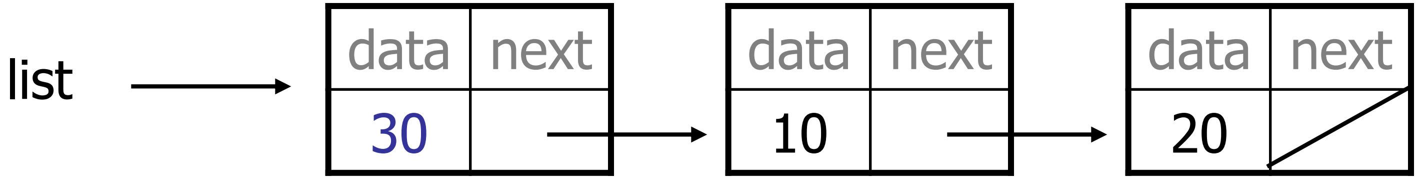


Linked node problem 2

- What set of statements turns this picture:

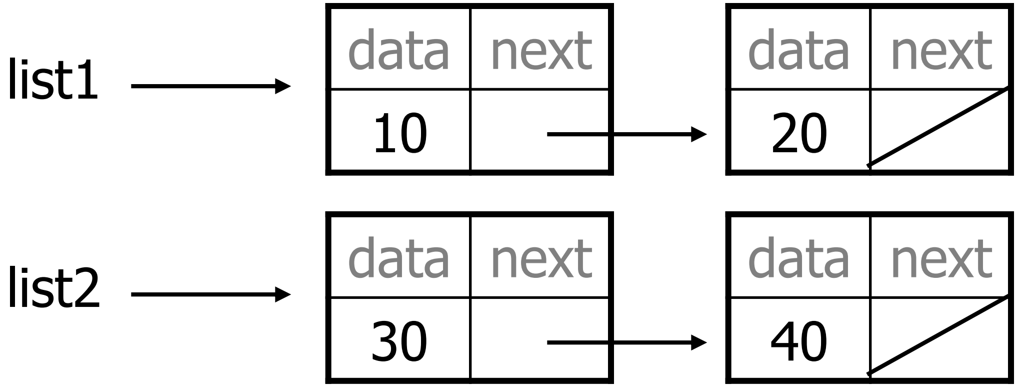


- Into this?

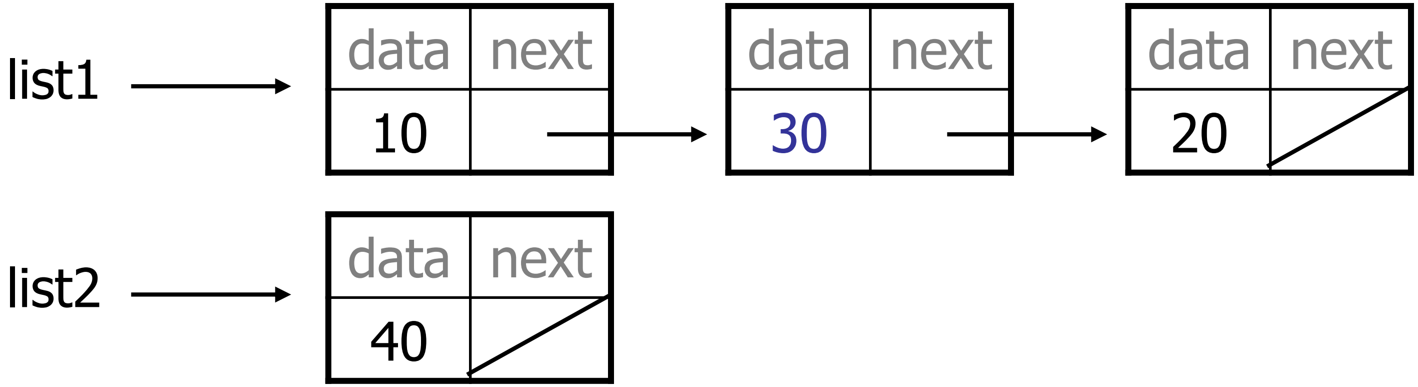


Linked node problem 3

- What set of statements turns this picture:

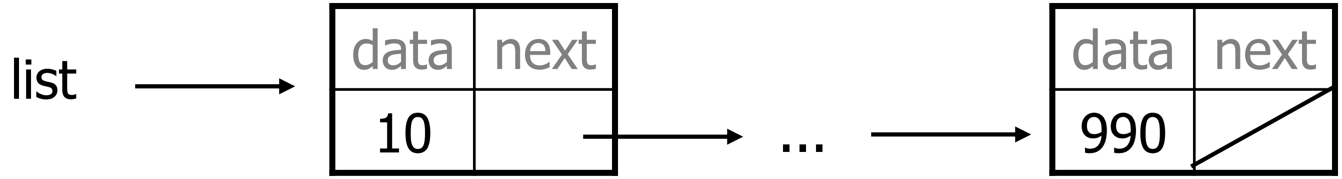


- Into this?

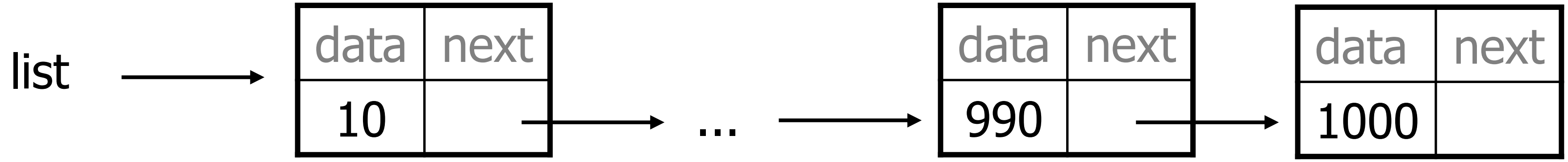


Linked node problem 4

- What set of statements turns this picture:



- Into this?



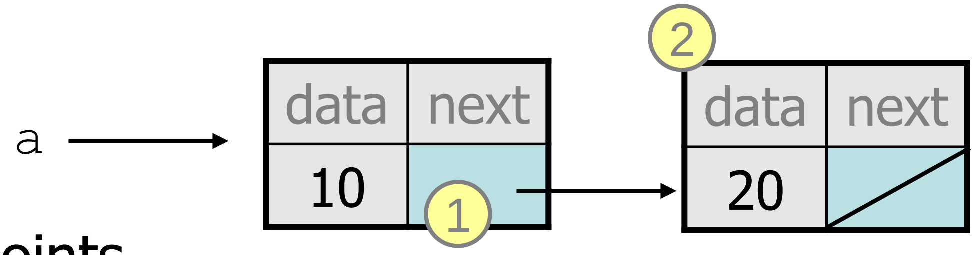
References vs. objects

variable `=` **value**`;`

a *variable* (left side of `=`) is an arrow (the base of an arrow)

a *value* (right side of `=`) is an object (a box; what an arrow points at)

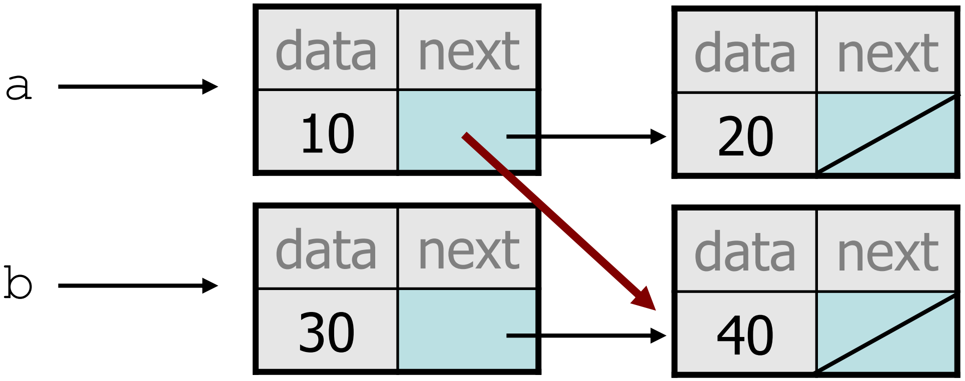
- For the list at right:
  - `a.next =` **value**`;`
    means to adjust where ① points
  - **variable** `= a.next;`
    means to make **variable** point at ②



Reassigning references

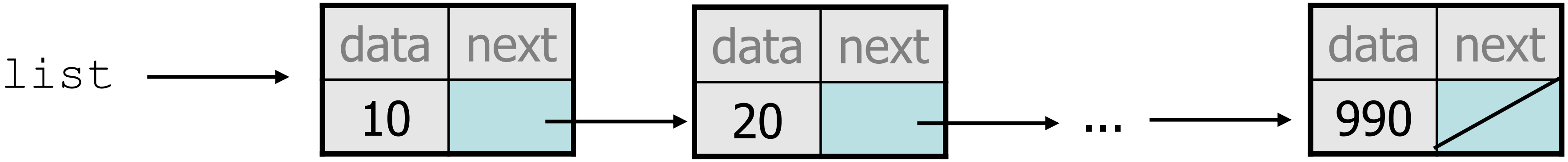
- when you say:
  - `a.next = b.next;`

- you are saying:
  - "Make the *variable* `a.next` refer to the same *value* as `b.next`."
  - Or, "Make `a.next` point to the same place that `b.next` points."



Linked node question

- Suppose we have a long chain of list nodes:



  - We don't know exactly how long the chain is.

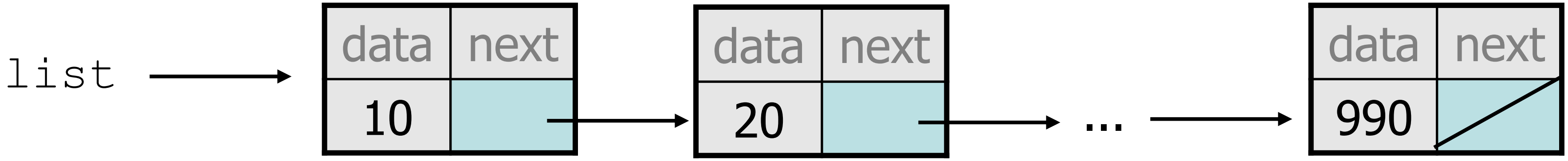
- How would we print the data values in all the nodes?

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?
```

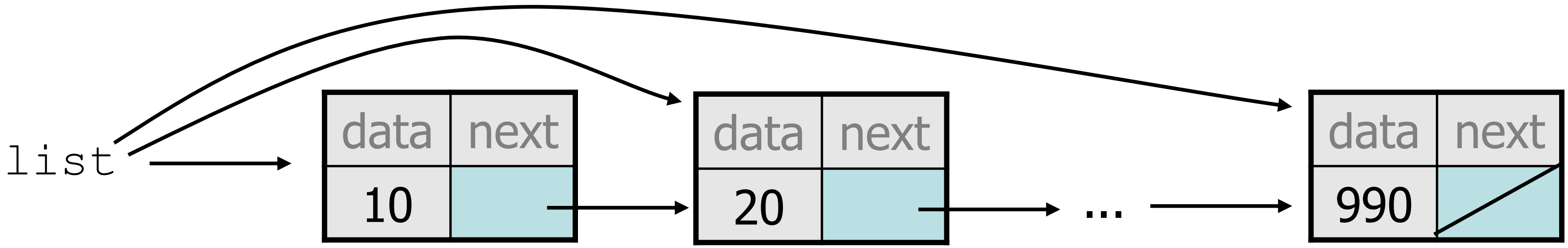


Traversing a list?

- 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
}
```

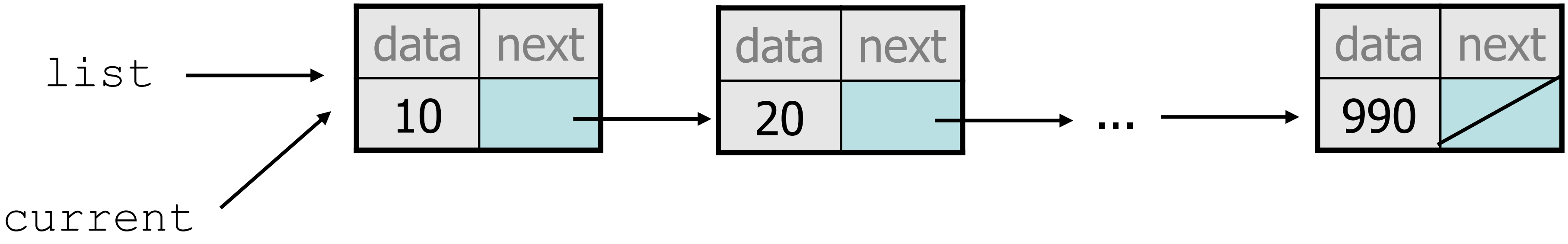
  - What's wrong with this approach?
    - (It loses the linked list as it prints it!)



A `current` reference

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

```
ListNode current = list;
```



- What happens to the picture above when we write:

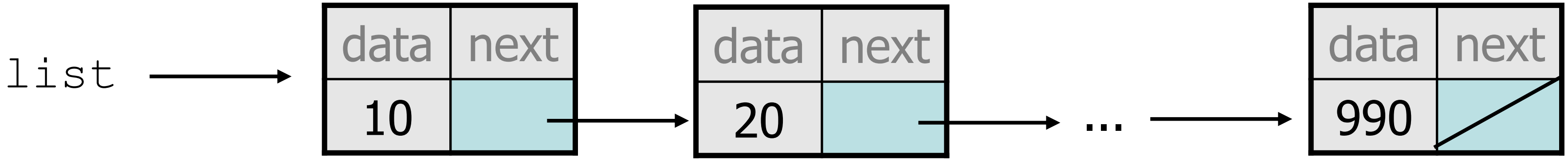
```
current = current.next;
```

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.



Linked list vs. array

- Algorithm to print list values:

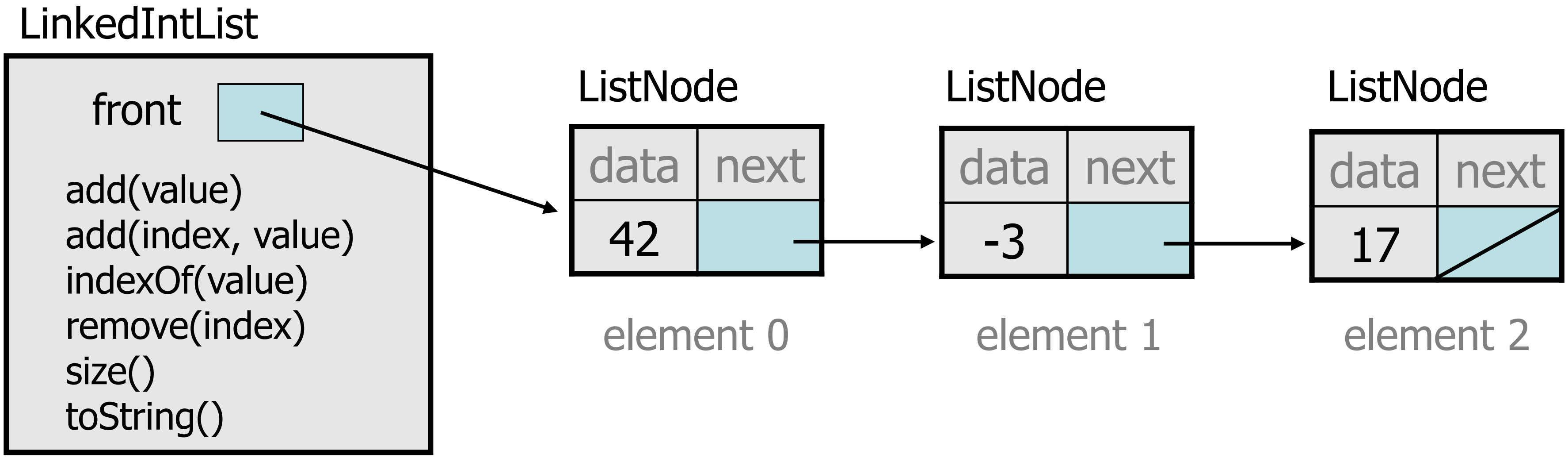
```
ListNode front = ...;

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

- Similar to array code:

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

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

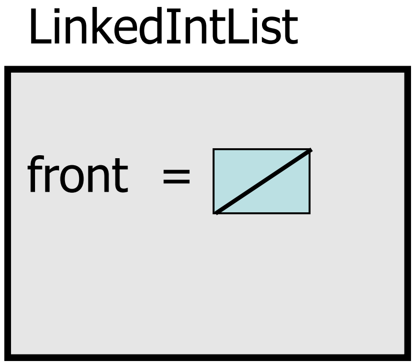
A LinkedIntList class

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



LinkedIntList class v1

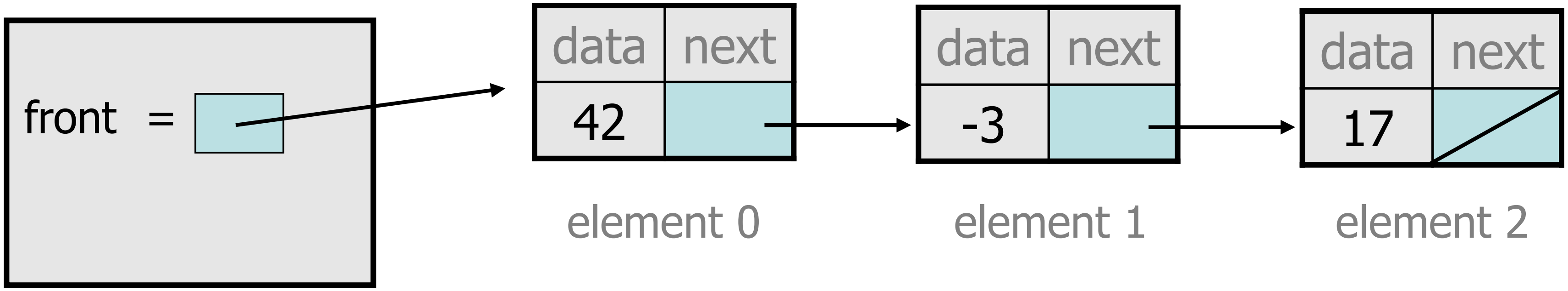
```
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }


    methods go here

}
```



Implementing add

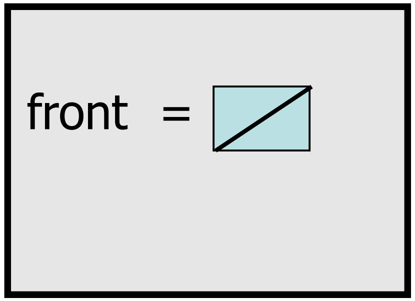
```
// Adds the given value to the end of the list.
public void add(int value) {
    ...
}
```

- How do we add a new node to the end of a list?

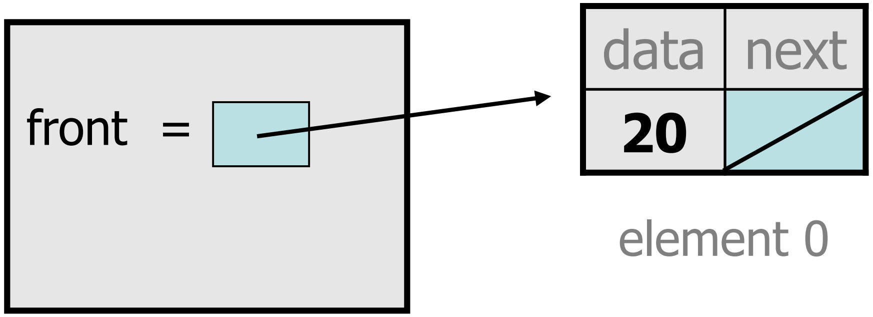


Adding to an empty list

- Before adding 20:



After:



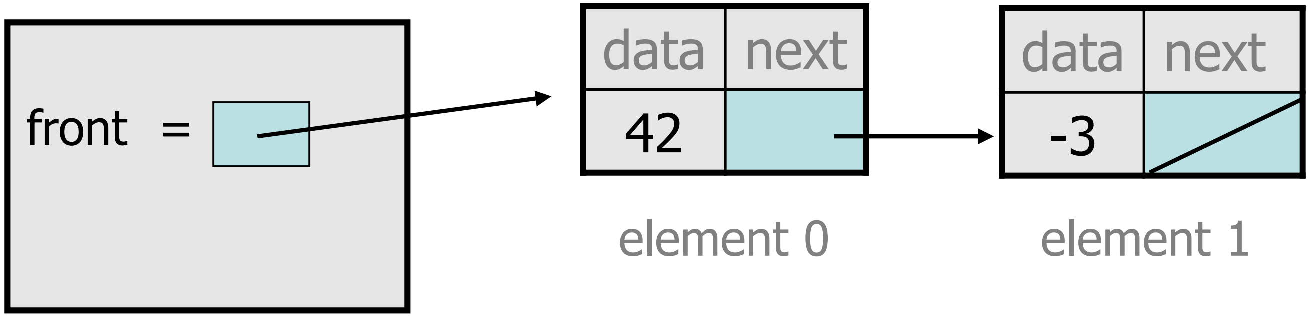
  - We must create a new node and attach it to the list.

The `add` method, 1st try

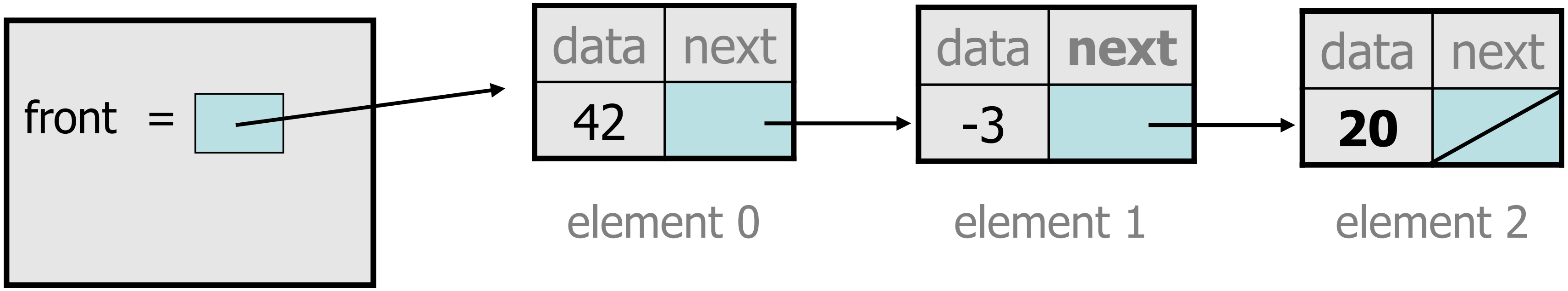
```
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list

        ...

    }
}
```

Adding to non-empty list

- Before adding value 20 to end of list:

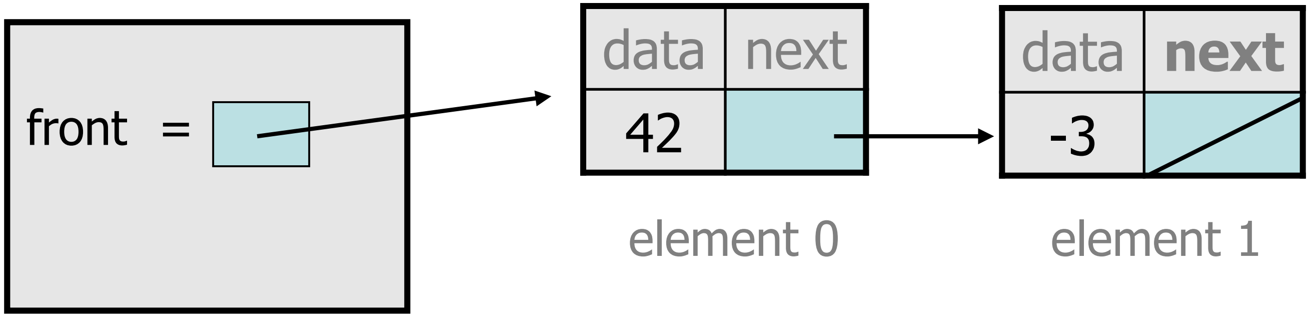


- After:



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?

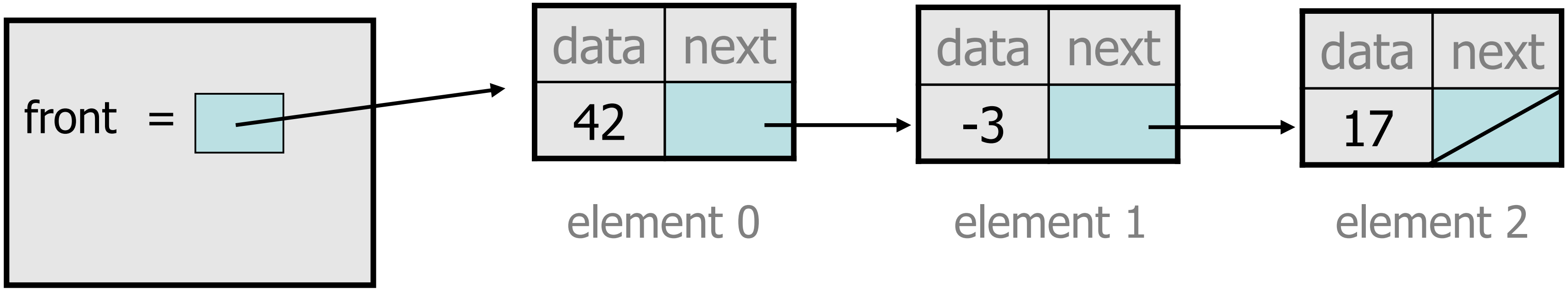
The add method

```
// 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);
    }
}
```

Implementing `get`

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

- Exercise: Implement the `get` method.



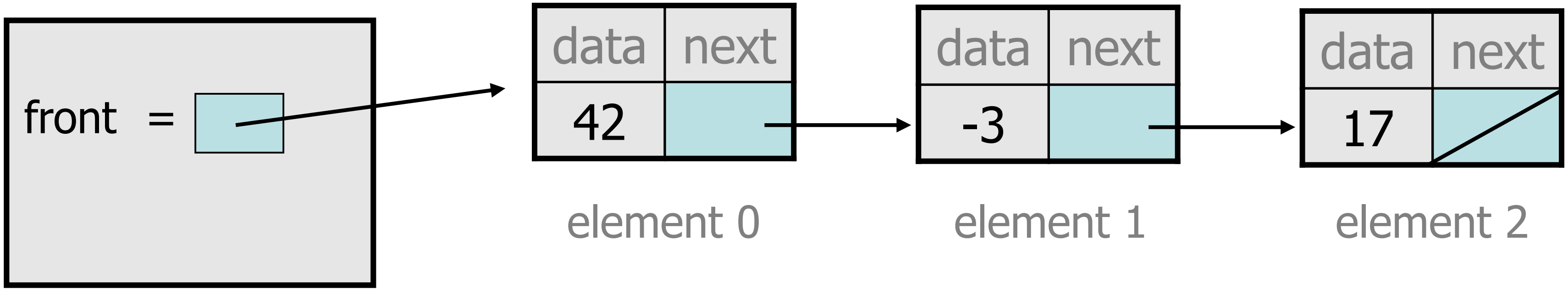
The `get` method

```
// 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);
    }
}
```

Conceptual questions

- What is the difference between a `LinkedIntList` and a `ListNode`?

- What is the difference between an empty list and a `null` list?
  - How do you create each one?

- Why are the fields of `ListNode` public? Is this bad style?

- What effect does this code have on a `LinkedIntList`?

```
ListNode current = front;
current = null;
```

Conceptual answers

- A list consists of 0 to many node objects.
  - Each node holds a single data element value.

- null list: `LinkedIntList list = null;`
  empty list: `LinkedIntList list = new LinkedIntList();`

- It's okay that the node fields are public, because client code never directly interacts with `ListNode` objects.

- The code doesn't change the list.
  You can change a list only in one of the following two ways:
  - Modify its `front` field value.
  - Modify the `next` reference of a node in the list.

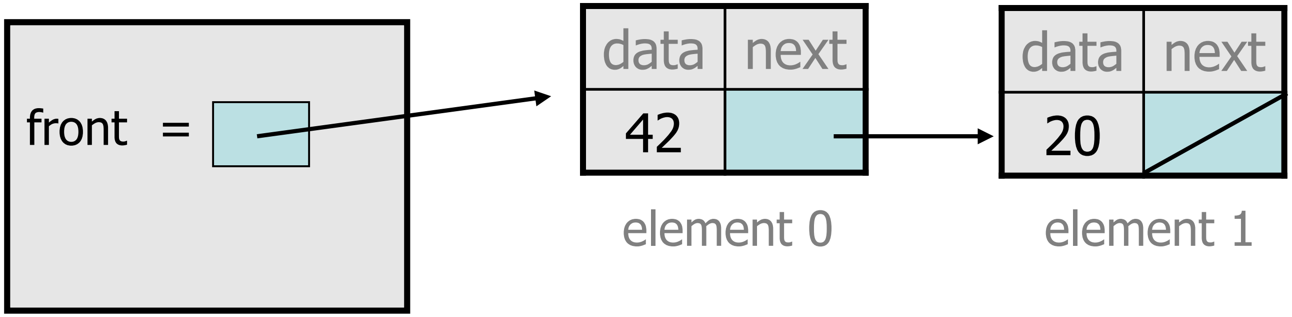
Implementing remove

```
// Removes and returns the list's first value.
public int remove() {
    ...
}
```

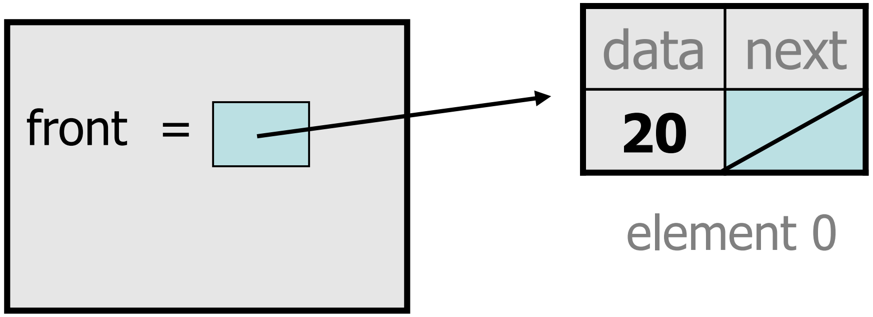
- How do we remove the front node from a list?

Removing front element

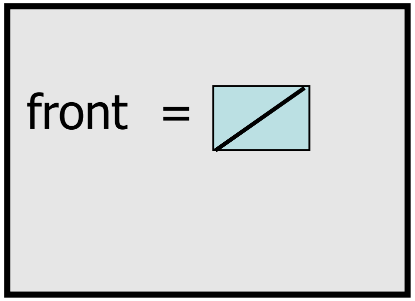
- Before removing front element:



- After first removal:



After second removal:



remove solution

```
// Removes and returns the first value.
// Throws a NoSuchElementException on empty list.
public int remove() {
    if (front == null) {
        throw new NoSuchElementException();
    } else {
        int result = front.data;
        front = front.next;
        return result;
    }
}
```

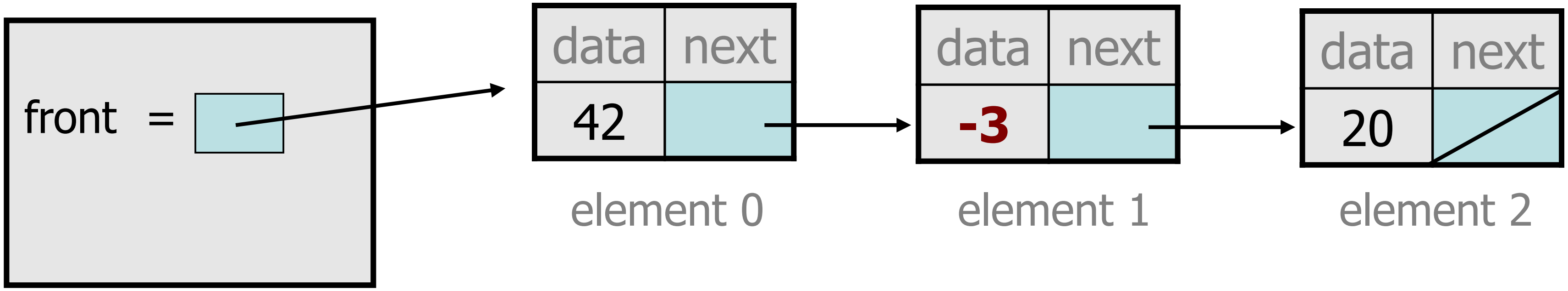
Implementing `remove` (2)

```
// Removes value at given index from list.
// Precondition: 0 <= index < size
public void remove(int index) {
    ...
}
```

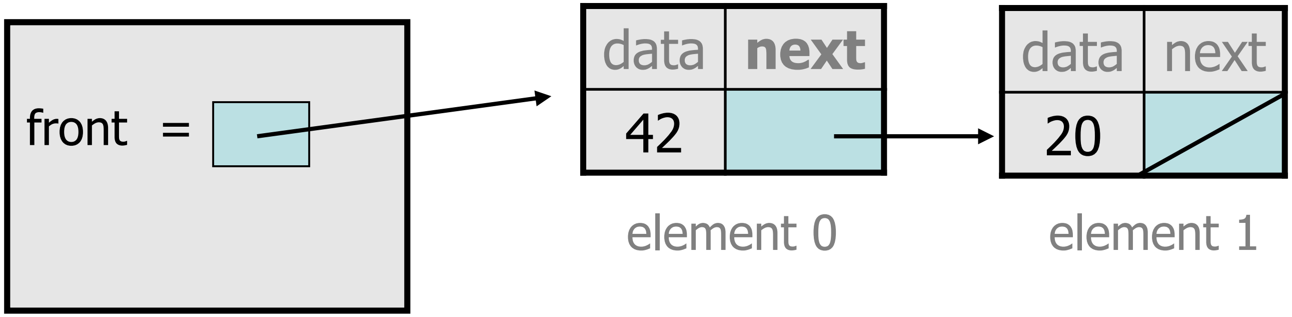
– How do we remove any node in general from a list?

Removing from a list

- Before removing element at index 1:

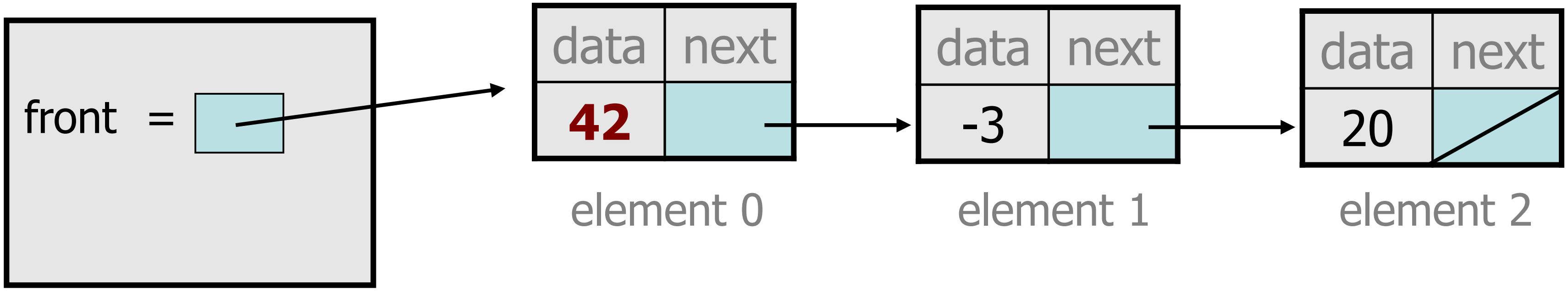


- After:

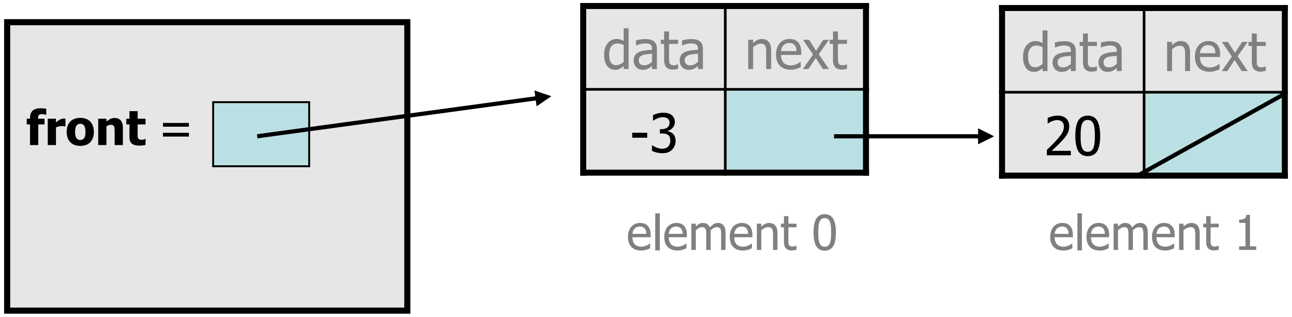


Removing from the front

- Before removing element at index 0:

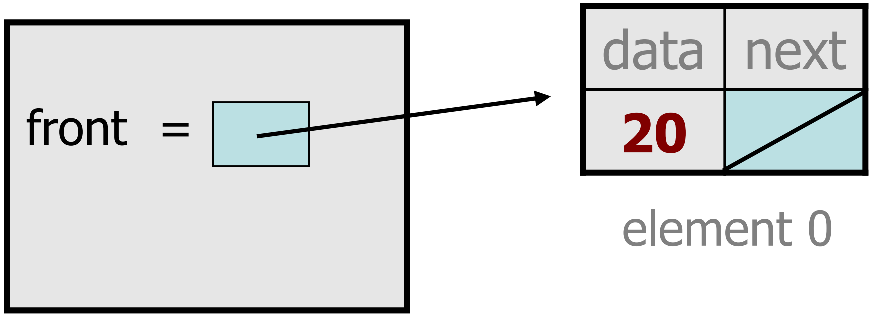


- After:

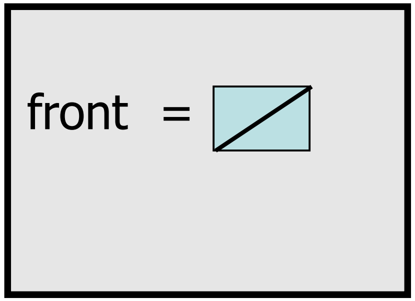


Removing the only element

- Before:



After:



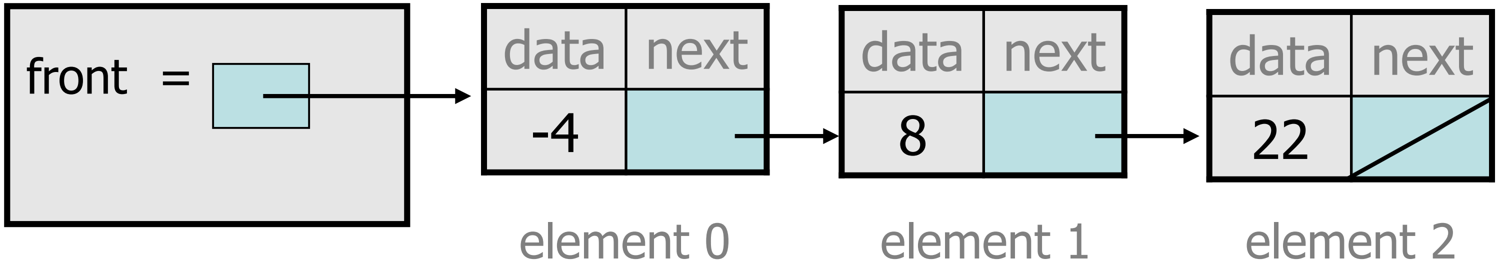
  - We must change the front field to store `null` instead of a node.
  - Do we need a special case to handle this?

remove (2) solution

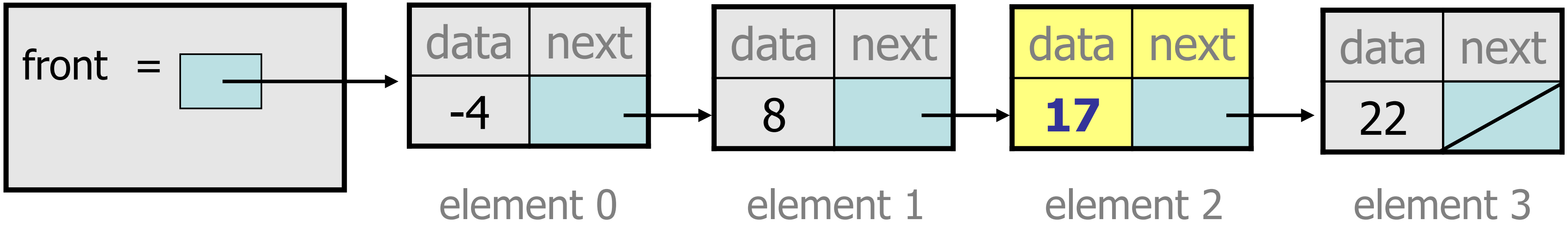
```
// Removes value at given index from list.
// Precondition: 0 <= index < size()
public void remove(int index) {
    if (index == 0) {
        // special case: removing first element
        front = front.next;
    } else {
        // removing from elsewhere in the list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = current.next.next;
    }
}
```

Exercise

- Write a method `addSorted` that accepts an integer value as a parameter and adds that value to a sorted list in sorted order.
  - Before `addSorted(17)` :



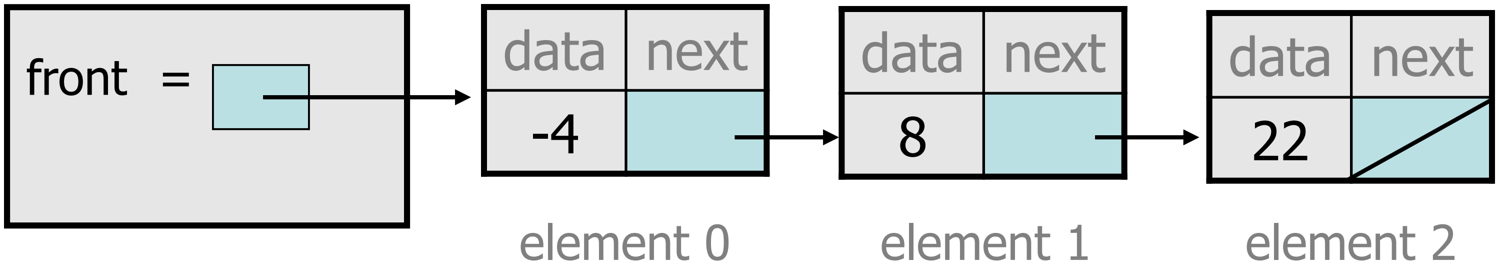
  - After `addSorted(17)` :



The common case

- Adding to the middle of a list:

```
addSorted(17)
```

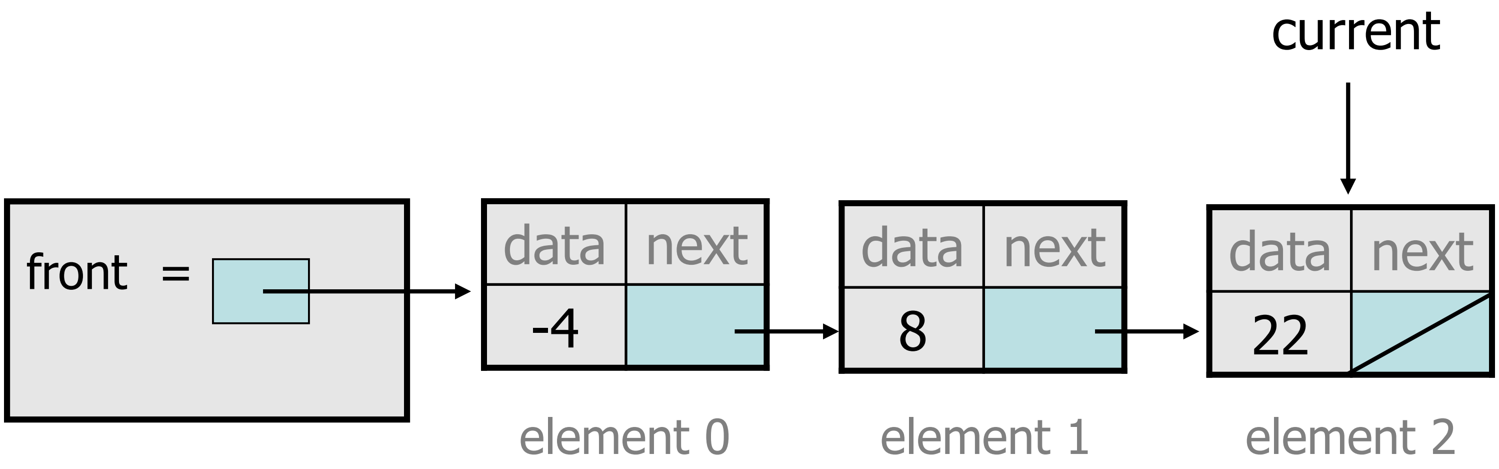


  - Which references must be changed?
  - What sort of loop do we need?
  - When should the loop stop?

First attempt

- An incorrect loop:

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

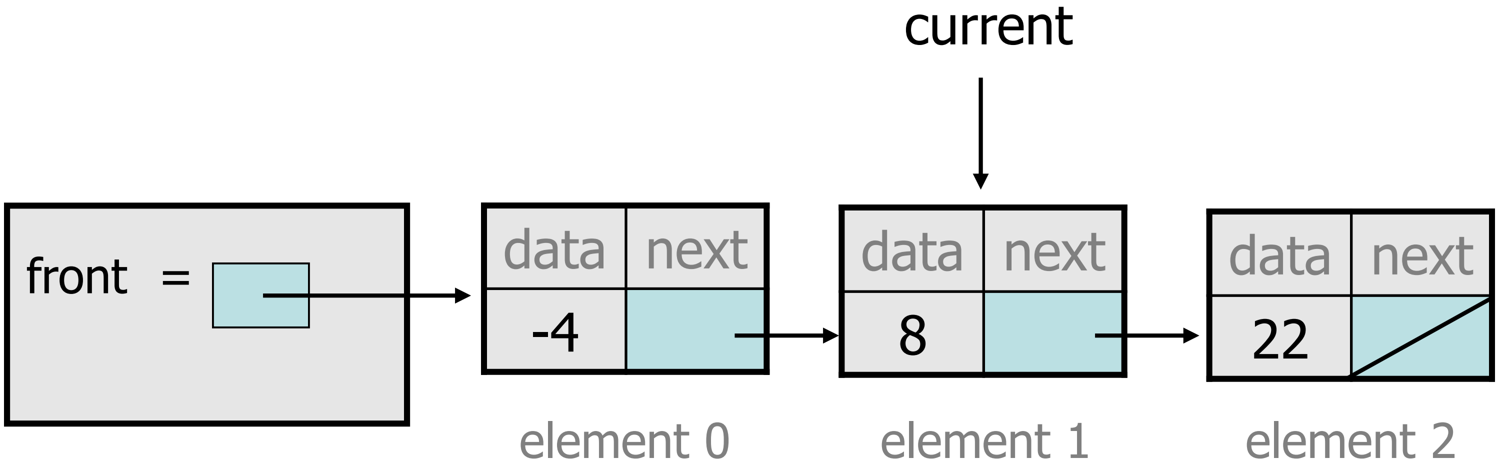


- What is wrong with this code?
  - The loop stops too late to affect the list in the right way.

Key idea: peeking ahead

- Corrected version of the loop:

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

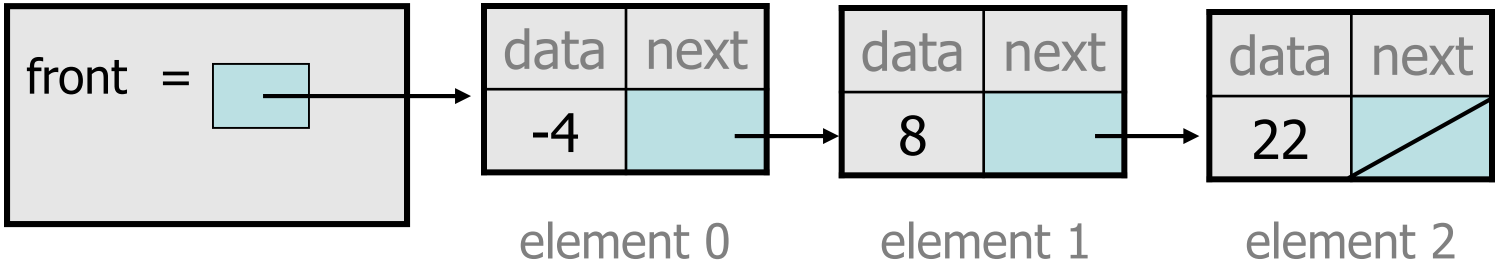


  - This time the loop stops in the right place.

Another case to handle

- Adding to the end of a list:

```
addSorted(42)
```



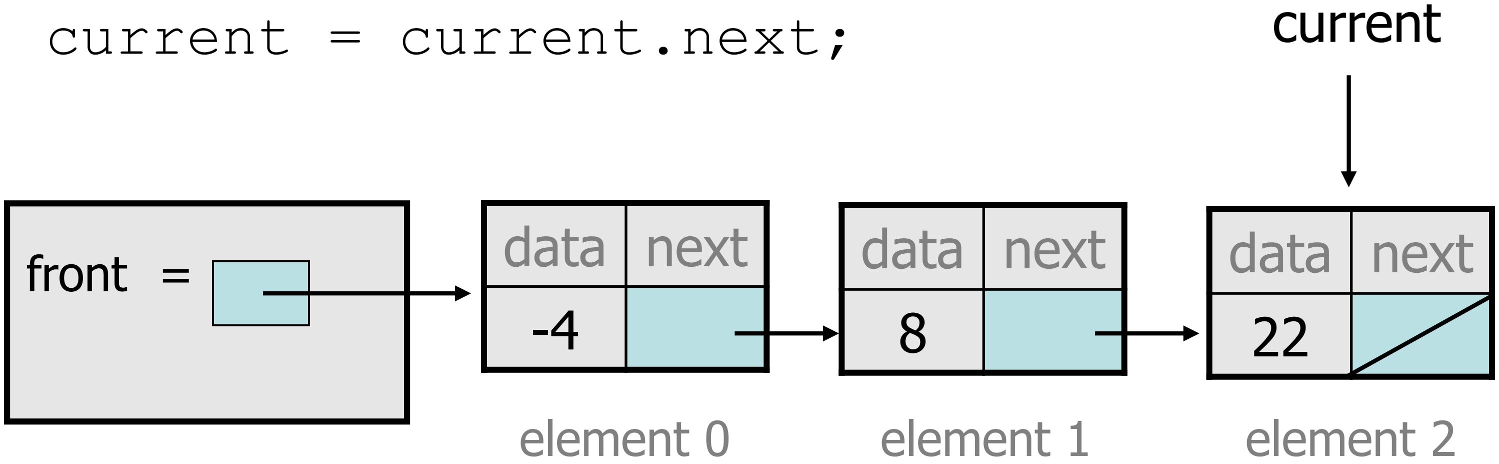
```
Exception in thread "main": java.lang.NullPointerException
```

- Why does our code crash?
- What can we change to fix this case?

Multiple loop tests

- A correction to our loop:

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

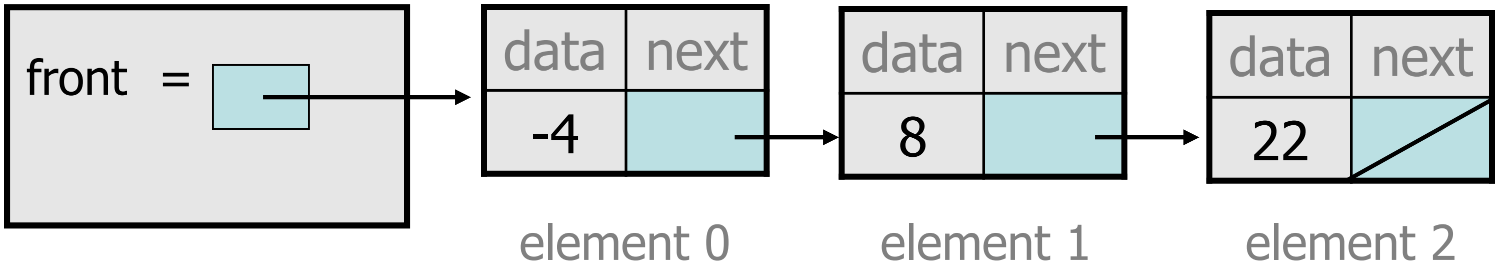


  - We must check for a `next` of `null` *before* we check its `.data`.

Third case to handle

- Adding to the front of a list:

```
addSorted(-10)
```



  - What will our code do in this case?
  - What can we change to fix it?

Handling the front

- Another correction to our code:

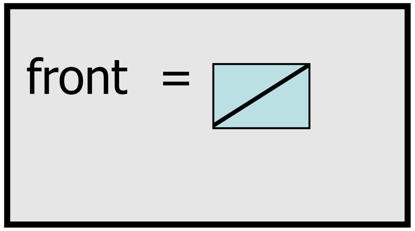
```
if (value <= front.data) {
    // insert at front of list
    front = new ListNode(value, front);
} else {
    // insert in middle of list
    ListNode current = front;
    while (current.next != null &&
           current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value,
                                 current.next);
}
```

  - Does our code now handle every possible case?

Fourth case to handle

- Adding to (the front of) an empty list:

```
addSorted(42)
```



  - What will our code do in this case?
  - What can we change to fix it?

Final version of code

```
// Adds given value to list in sorted order.
// Precondition: Existing elements are sorted
public void addSorted(int value) {
    if (front == null || value <= front.data) {
        // insert at front of list
        front = new ListNode(value, front);
    } else {
        // insert in middle of list
        ListNode current = front;
        while (current.next != null &&
               current.next.data < value) {
            current = current.next;
         }
        current.next = new ListNode(value,
                                   current.next);
    }
}
```

Other list features

- Add the following methods to the `LinkedIntList`:
  - `size`
  - `isEmpty`
  - `clear`
  - `toString`
  - `indexOf`
  - `contains`

- Add a `size` field to the list to return its size more efficiently.

- Add preconditions and exception tests to appropriate methods.

Abstract classes

- **abstract class**: A hybrid between an interface and a class.
  - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes cannot be instantiated (cannot use `new` to create any objects of their type)

- What goes in an abstract class?
  - implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - declare generic behaviors that subclasses must implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {
    ...

    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters);

}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

- Exercise: Introduce an abstract class into the list hierarchy.

Abstract and interfaces

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements IntList {}  // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements IntList {} // ok

public class Child extends Empty {}             // error
```

Abstract class vs. interface

- Why do both interfaces and abstract classes exist in Java?
  - An abstract class can do everything an interface can do and more.
  - So why would someone ever use an interface?

- Answer: Java has single inheritance.
  - can extend only one superclass
  - can implement many interfaces
  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.