CSE 143
Lecture 7

More Linked Lists

slides created by Marty Stepp
http://www.cs.washington.edu/143/

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;
```

2

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.

3

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
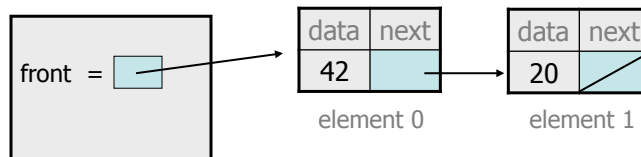   - example: **current.next = null;**

4

Implementing `remove`

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

- How do we remove the front node from a list?
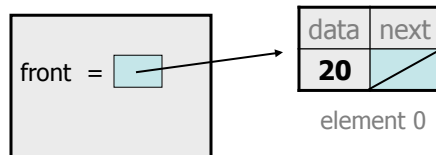- Does it matter what the list's contents are before the remove?
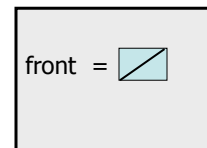
5

Removing front element

• Before removing front element:

data	next
42	

element 0

front =

data	next
20	

element 1

• After first removal:

front =

data	next
20	

element 0

After second removal:

front =

6

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;
    }
}
```

7
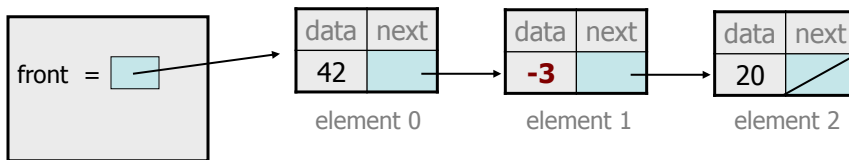
Implementing remove (2)

```
// 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?
– Does it matter what the list's contents are before the remove?
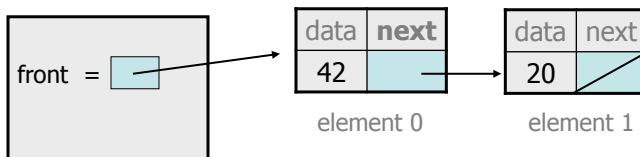
8

4

Removing from a list
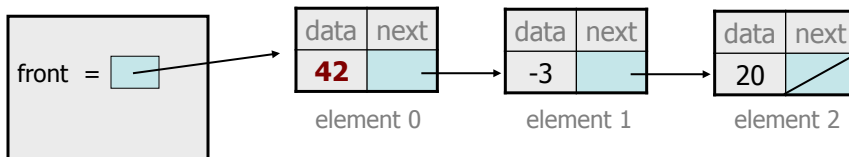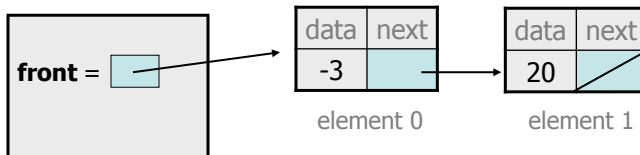
- Before removing element at index 1:

front = [ ] → | data | next |
42		→	data	next
-3		→	data	next
20	/

element 0   element 1   element 2

- After:

front = [ ] → | data | **next** |
| 42 | | → | data | next |
| 20 | / |

element 0   element 1

Removing from the front

- Before removing element at index 0:

front = [ ] → | data | next |
42		→	data	next
-3		→	data	next
20	/

element 0   element 1   element 2

- After:

front = [ ] → | data | next |
| -3 | | → | data | next |
| 20 | / |

element 0   element 1

Removing the only element

- Before: After:

data	next
20	

element 0
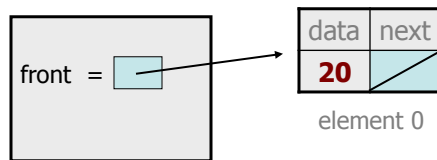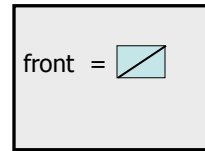
front =

front =

- We must change the front field to store `null` instead of a node.
- Do we need a special case to handle this?
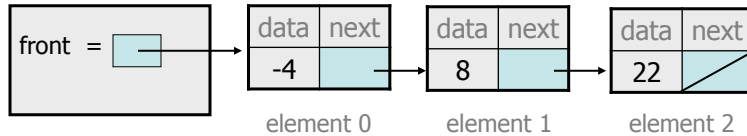
`remove` (2) solution

```java
// 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;
    }
}
```
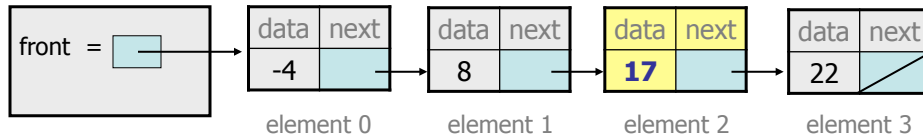
Exercise

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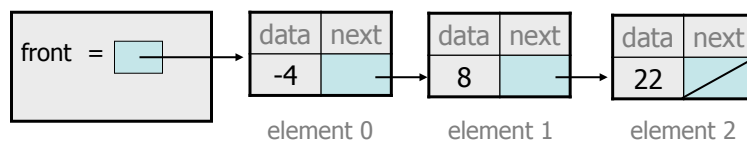
  - Before `addSorted(17)` :

front =		data	next		data	next		data	next
		-4			8			22	

  element 0   element 1   element 2

  - After `addSorted(17)` :

front =		data	next		data	next		data	next		data	next
		-4			8			**17**			22	

  element 0   element 1   element 2   element 3

13

The common case

- Adding to the middle of a list:

  `addSorted(17)`

front =		data	next		data	next		data	next
		-4			8			22	

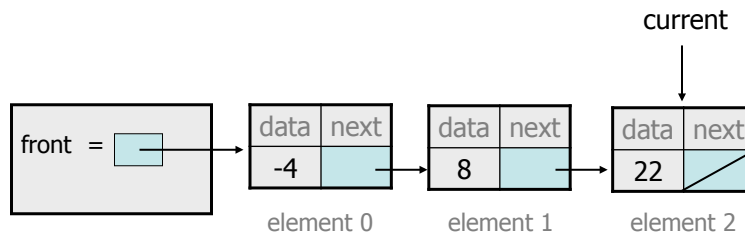 element 0   element 1   element 2

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

14

First attempt

- An incorrect loop:

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

current

data	next
-4	

front =

data	next
-4	

data	next
8	

data	next
22	

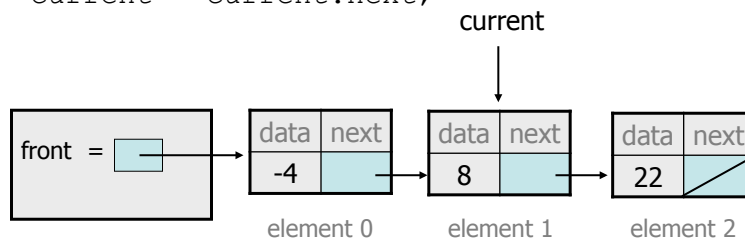element 0   element 1   element 2

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

15

Key idea: peeking ahead

- Corrected version of the loop:

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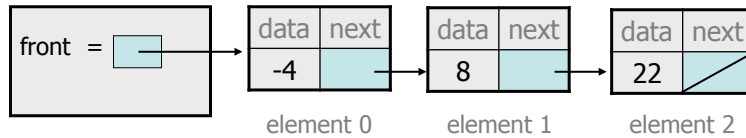
current

front =

data	next
-4	

data	next
8	

data	next
22	

element 0   element 1   element 2

  - This time the loop stops in the right place.

16

Another case to handle

- Adding to the end of a list:

  ```
  addSorted(42)
  ```


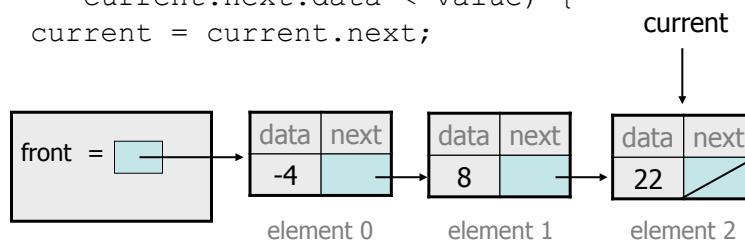 **Exception in thread "main": java.lang.NullPointerException**

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

17

Multiple loop tests

- A correction to our loop:

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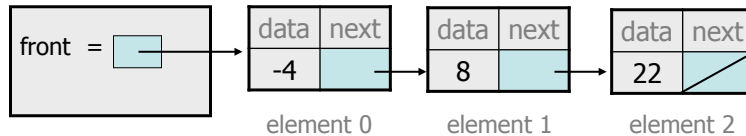

  - We must check for a `next` of `null` *before* we check its `.data`.

18

Third case to handle

- Adding to the front of a list:

```
addSorted(-10)
```

front = [ ]	→	data	next	→	data	next	→	data	next
		-4			8			22	

     element 0   element 1   element 2

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

19

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);
}
```
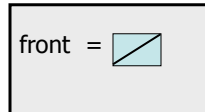
- – Does our code now handle every possible case?

20

10

Fourth case to handle

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

  `addSorted(42)`

  front =

  - 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);
    }
}
```

Exercise

- Write a method `reverse` that reverses the order of the elements in the list.

  For example, if the list initially stores this sequence of integers:

  ```
  [1, 8, 19, 4, 17]
  ```

  It should store the following sequence of integers after reverse is called:

  ```
  [17, 4, 19, 8, 1]
  ```

23

Solution

```
public void reverse() {
    ListNode current = front;
    ListNode previous = null;
    while (current != null) {
        ListNode nextNode = current.next;
        current.next = previous;
        previous = current;
        current = nextNode;
    }
    front = previous;
}
```

24