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`?
  ```java
  ListNode current = front;
  current = null;
  ```
A list consists of 0 to many node objects.
  - Each node holds a single data element value.

null list:       LinkedList list = null;
empty list:      LinkedList list = new LinkedList();

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?
- Does it matter what the list's contents are before the remove?
Removing front element

• Before removing front element:

- front = element 0

  - data: 42
  - next:

- element 1

  - data: 20
  - next:

• After first removal:

- front = element 0

  - data: 20
  - next:

After second removal:

- front = element 0

- data: 20
  - next:
// 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;
    }
}
// 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?
Removing from a list

• Before removing element at index 1:

  ![Diagram showing a list before removal](image)

  front = [42]

  element 0: [42, -3]

  element 1: [20]

• After:

  ![Diagram showing a list after removal](image)

  front = [42]

  element 0: [42]

  element 1: [20]
Removing from the front

• Before removing element at index 0:

  - front = [42]

  - Next data:
    - Element 0: data = 42
    - Element 1: data = -3
    - Element 2: data = 20

• After:

  - front = [-3]

  - Next data:
    - Element 0: data = -3
    - Element 1: data = 20
Removing the only element

• Before:

- We must change the front field to store `null` instead of a node.
- Do we need a special case to handle this?
// 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;
    }
}
• 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)`:**

  - **Before:**
    - `front = -4` → `data: -4` → `next`
    - `data: 8` → `next`
    - `data: 22` → `next`
    - `front = -4` → `next`
    - `data: 17` → `next`
    - `data: 22` → `next`

- **After `addSorted(17)`:**

  - **After:**
    - `front = -4` → `data: -4` → `next`
    - `data: 8` → `next`
    - `data: 17` → `next`
    - `data: 22` → `next`
    - `front = -4` → `next`
    - `data: 17` → `next`
    - `data: 22` → `next`
The common case

- Adding to the middle of a list: \texttt{addSorted(17)}

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

• An incorrect loop:

```java
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.
• Corrected version of the loop:

```java
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:
  \texttt{addSorted(42)}

\begin{itemize}
  \item \texttt{Exception in thread "main": java.lang.NullPointerException}
  \item Why does our code crash?
  \item What can we change to fix this case?
\end{itemize}
Multiple loop tests

• A correction to our loop:

```java
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:
  \texttt{addSorted(-10)}

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

```java
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;
    }
}
```

– Does our code now handle every possible case?
Fourth case to handle

- Adding to (the front of) an empty list:
  \texttt{addSorted(42)}

front = \[
\]

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