Linked node problem 3

- What set of statements turns this picture:

  list1
  \[
  \begin{array}{ll}
  \text{data} & \text{next} \\
  10 & \text{\_}
  \end{array}
  \quad \rightarrow \quad
  \begin{array}{ll}
  \text{data} & \text{next} \\
  20 & \text{\_}
  \end{array}
  \]

  list2
  \[
  \begin{array}{ll}
  \text{data} & \text{next} \\
  30 & \text{\_}
  \end{array}
  \quad \rightarrow \quad
  \begin{array}{ll}
  \text{data} & \text{next} \\
  40 & \text{\_}
  \end{array}
  \]

- Into this?

  list1
  \[
  \begin{array}{ll}
  \text{data} & \text{next} \\
  10 & \text{\_}
  \end{array}
  \quad \rightarrow \quad
  \begin{array}{ll}
  \text{data} & \text{next} \\
  20 & \text{\_}
  \end{array}
  \quad \rightarrow \quad
  \begin{array}{ll}
  \text{data} & \text{next} \\
  30 & \text{\_}
  \end{array}
  \]

  list2
  \[
  \begin{array}{ll}
  \text{data} & \text{next} \\
  40 & \text{\_}
  \end{array}
  \]
Linked node problem 3

- How many ListNode variables?

- Which variables change?
Linked node problem 3

- How many ListNode variables?

- Which variables change?
Linked node problem 3

- How many ListNode variables?
- Which variables change?

```python
list1.next.next = list2
```
Linked node problem 3

- How many ListNode variables?

- Which variables change?
Linked node problem 3

- How many ListNode variables?

- Which variables change?

```
list1.next.next = list2
list2 = list2.next
list1.next.next.next = null
```
References vs. objects

\[\text{variable} = \text{value};\]

a \textit{variable} (left side of \(=\)) is an arrow (the base of an arrow)
a \textit{value} (right side of \(=\)) is an object (a box; what an arrow points at)

- For the list at right:
  - \texttt{a.next = value;} means to adjust where \(\textcircled{1}\) points
  - \texttt{variable = a.next;} means to make \texttt{variable} point at \(\textcircled{2}\)
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?

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

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

  ```java
  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

**Print list values:**

```java
ListNode list = ...;

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

**Similar to array code:**

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

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Array Code</th>
<th>Linked List Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to front of list</td>
<td><code>int i = 0;</code></td>
<td><code>ListNode current = list;</code></td>
</tr>
<tr>
<td>Test for more elements</td>
<td><code>i &lt; size</code></td>
<td><code>current != null</code></td>
</tr>
<tr>
<td>Current value</td>
<td><code>elementData[i]</code></td>
<td><code>current.data</code></td>
</tr>
<tr>
<td>Go to next element</td>
<td><code>i++;</code></td>
<td><code>current = current.next;</code></td>
</tr>
</tbody>
</table>
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.
### Table

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

### Diagram

The diagram shows a linked list with the following nodes:
- **Front**: 10
- 20
- 30

The arrows indicate the direction of the links.
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

front

add(value)
add(index, value)
indexOf(value)
remove(index)
size()
toString()
```

```
ListNode

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

element 0
```

```
ListNode

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
</tr>
</tbody>
</table>

element 1
```

```
ListNode

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 2
```
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }

    methods go here
}
Implementing `add`

```java
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?

```
front = data: 42  next: element 0
          data: -3  next: element 1
              data: 17  next: element 2
```
Adding to an empty list

- Before adding 20:

  ![Diagram](image1)

  `front = null`

- After:

  ![Diagram](image2)

  `front = new Node(data, next)`

  `element 0`

- We must create a new node and attach it to the list.
The `add` method, 1st try

```java
// 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:

  front = 42

  element 0

  data | next
  --- | ---
  42  |   

  element 1

  data | next
  --- | ---
  -3  |   

• After:

  front = 42

  element 0

  data | next
  --- | ---
  42  |   

  element 1

  data | next
  --- | ---
  -3  |   

  element 2

  data | next
  --- | ---
  20  |   
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

```java
// 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 \texttt{add (2)}

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

- Exercise: Implement the two-parameter \texttt{add} method.
The `add` method (2)

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