Lecture 7: Linked Nodes

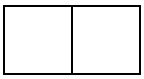
- We've looked at an array-based structure (ArrayList)
    - Why did we need ArrayList in the first place?
        - Can't expand an array - why?
        - Because arrays are stored as chunks of memory
    - So why even store data as chunks at all?
        - It makes it fast - O(1) lookup
        - (random access)
    - What other things to arrays do badly?
        - adding/removing in the middle
        - Ex. you have 10,000 values and want to remove the first value - have to scoot all 9,999 other values
    - So what we're going to do - break up the single chunk, and in so doing be able to add/remove efficiently
        - Give up random access
        - "opposite" benefits of ArrayList
- The changes to make
    - An array:

0	2	4	2	1	7

    - But now we don't need everything contiguous in memory - scattered
        - (little boxes everywhere: 23, 2, 40, 0, 14, 72)
    - But a list needs ORDER, so how can we keep track of order?
        - Essentially, add an "arrow" from each box to the one that is next
        - Each bit of data "points" to the next bit of data
    - How do we keep track of the front?
        - A "pointer" to the first element of the list
        - Is this enough information? YES
        - Can follow pointers to every other element of the list
        - (like a VHS tape rather than a CD)
    - We call each element a NODE
        - Like a lego building block
        - Consists of two parts: the DATA and an arrow (REFERENCE)
        - The difference between a NODE and a REFERENCE to a node is a central one that we'll talk about
        - Like the difference between a phone number and an actual person
            - I store PHONE NUMBERS, not PEOPLE in my phone
- We're going to develop a LinkedIntList of nodes

- We draw nodes like this:

data next



    - How many fields? What types?
    - We'll write the ListNode class - nodes are independent objects

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

    - Private fields?
        - Not in this case - I'll explain why later in the week why this is ok to do
    - ListNode is a "recursive" type - it is defined with a field of its own type
        - This is OK to do
- Let's write some code to build up a list with values 3, 7, 12
    - First, create the variable list of type ListNode
        - We draw this as a single box - stores a REFERENCE (arrow) to a node
            - IT IS NOT A NODE ITSELF
        - What does it store initially? NULL
            - The absence of anything
    - Then, set it equal to a new node
    - Changes the picture

```
                  +------+------+
     +---+        | data | next |
list | +-+--->    |      |      |
     +---+        +------+------+
```

    - What do we want to have the node store as data? 3
    - What do we want it's "next" reference to point to? A new node

```
list.data = 3;
list.next = new ListNode();
```

```
                  +------+------+       +------+------+
     +---+        | data | next |       | data | next |
list | +-+--->    |   3  |   +--+--->   |      |      |
     +---+        +------+------+       +------+------+
```

    - Have to be careful about what you're talking about
        - We get "inside" a node by using the dot notation - "follow the arrow"
        - Then we give the name of the field that we want to modify

```
list.next.data = 7;
list.next.next = new ListNode();
```

    - (draw picture, point out what list, list.next, and list.next.next refer to)
    - Set the final data

```
list.next.next.data = 12;
list.next.next.next = null;
```

    - The final box (reference) is NULL - meaning the absence of a value (“terminator”)
    - Final assignment to null is actually unnecessary - the default is null
- This is obviously tedious
    - Bad way to manipulate a list
    - We’d need a list class to hide these details, have the “add”, “remove”, “get” operations
    - But for today, we’re not going to worry about those - nodes are hard enough
- But we can make some improvements - add constructors

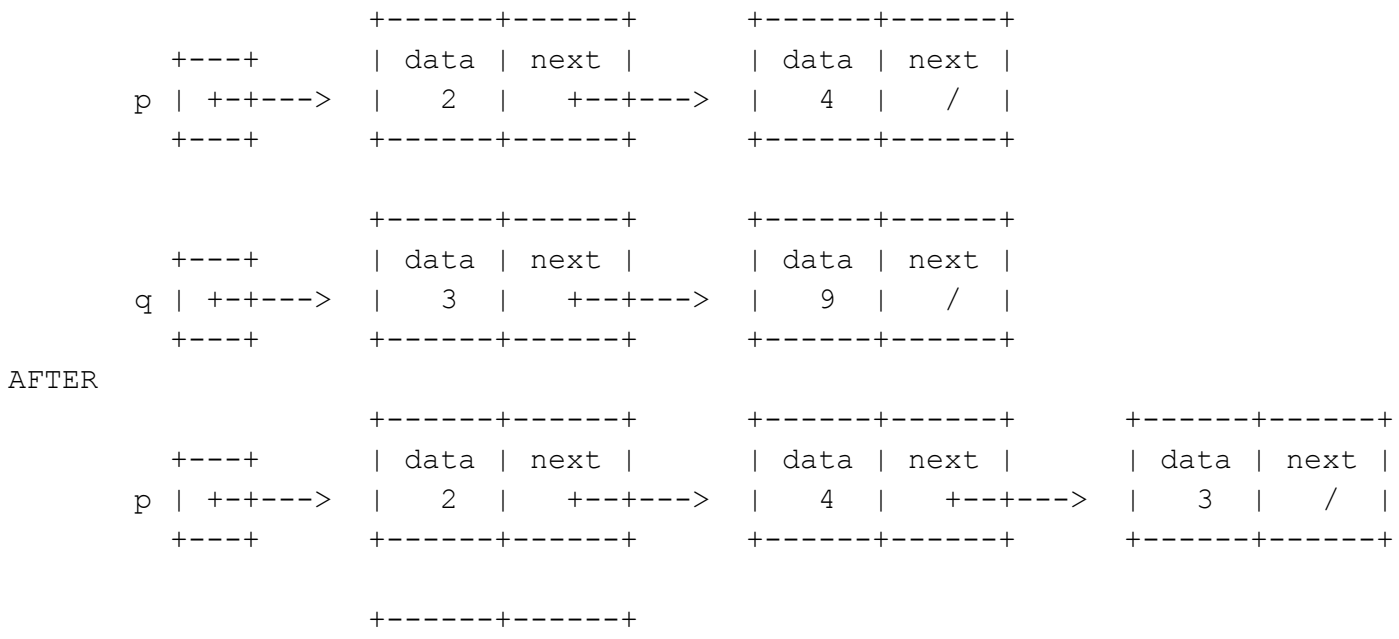
```
public class ListNode {
    public int data;
    public ListNode next;

    public ListNode() {
        this(0, null);
    }

    public ListNode(int data) {
        this(data, null);
    }

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

    - Note, we use the “this” notation - all constructors call one “main” constructor
    - We could write the previous creation in a single line of code
- Section tomorrow, lots of problems manipulating these nodes, understanding the data/node/reference distinction
- A problem:



```
  +---+       | data | next |
q | +-+--->  |   9  |   /  |
  +---+       +------+------+
```

- Solution
    - How many variables of type ListNode do we have?
        - SIX
    - Number the boxes - which ones need to change?
    - But we have to be careful about the order of the changes
        - What if we changed the “q” box first to point to the 9? Then we’d “lose” the 3 because we’d have no way to refer to it
        - Like having helium balloons and losing the string - fly away
    - Which box is it “safe” to change”?
        - The one with “9” as data b/c the “next” is already “null”
- jGrasp debugger
- Very important to draw PICTURES
    - The variables can be very confusing
    - Only way to master LinkedList code
- Talk about NULL
    - What you can do with null
        - Store it
        - Test for it
        - Print it
        - Pass it
        - Return it
    - What you can’t do with it
        - Dereference it (NullPointerException)
    - jGrasp interactions pane
- More exercises
    - Turn [10, 20, 30] into [20, 30]
    - Turn [10, 20] into [30, 10, 20]
    - Turn [10, 20] into [10, 20, 30]
    - Turn [10, 20,....,990] into [10, 20, ..., 990, 1000]
        - “Stopping one early”
        - Use of a “current”

```
ListNode current = list;
while (current.next != null) {
    current = current.next;
}
current.next = new ListNode(1000);
```