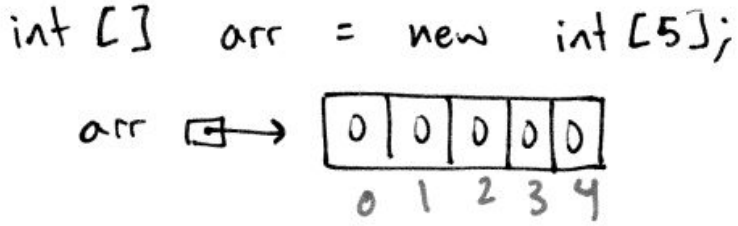
CSE143 Notes for Monday, 7/06/15

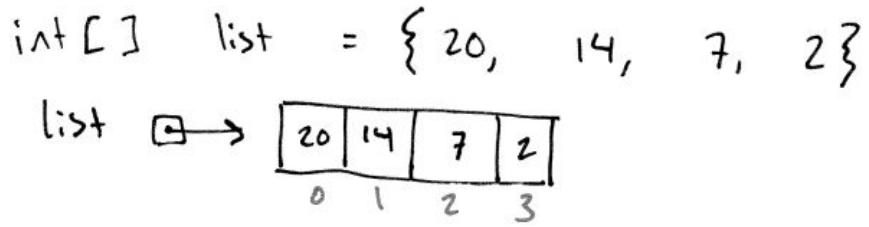
Arrays use contiguous memory:



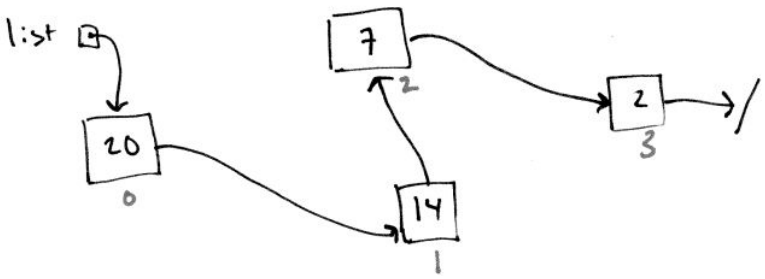
Arrays are what we call "random access" structures because we can quickly access any value within the array. If you use `arr[2]`, then you can access the element stored in index 2 by just jumping straight to it. However, one downside to that is that it is not easy to expand the array to store more values, and another downside is removing elements can cause expensive shifting. If you have something like 10 thousand values in an array and you want to get rid of the first one, you have to shift 9,999 values over to fill in the gap.

We explored how to fix this problem with a completely new data structure. The idea was suggested, that what if each element knew where to look for its neighbor, then we could spread out the elements all over memory. This is the idea behind a LinkedList.

We might store an array of 4 ints as follows. The internal representation of that list as an array is:



The internal representation of that list as a LinkedList is:



Each bit of data is going to point to the next bit of data and the final bit of data will have a special value that will indicate that we are at the end of the list. You might think that even with this interconnected structure, we'd have to keep track of where each value is stored. In fact, we just need a reference to the front of the list. So if we can get to the value that stores 20 in it, then from there we can get to every other value in the list.

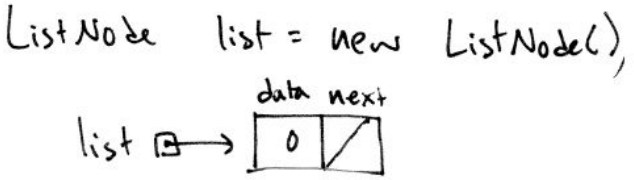
Linked lists have what we would call "sequential access." That means that it can be slow to access things in the middle of the list. However, with linked lists, you can add elements, insert new elements, and remove elements relatively quickly. Removing the first element would not require shifting all other elements over, instead you could just redirect the reference to the beginning of the list to instead be pointing at the second element.

In fact, we'll find that the things that arrays do particularly well linked lists tend to do badly and vice versa.

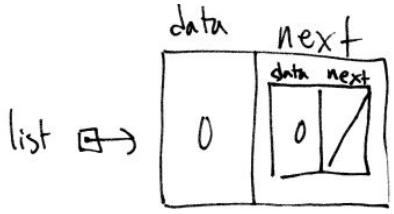
Linked lists are composed of individual elements called nodes. We will use the ListNode object defined in class, which is an object with two data fields: one for storing a single item of data and one for storing a reference to the next node in the list. For a list of int values, we'd declare this as follows:

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

This isn't a nicely encapsulated object because of the public data fields. It is okay to do this, because we will keep the entire ListNode object encapsulated inside of the LinkedIntList class we will be building later in the week.



This ListNode is a recursive data structure (a class that is defined in terms of itself in that the class is called ListNode and it has a data field of type ListNode). If we made two ListNodes that were linked together, one way to think about the internal representation is:



However, the way we think about a LinkedList in this class, is that the next field stores a reference to another ListNode object, not the entire object itself. The way we would draw this recursive structure is:

Then we wrote some code that would build up the list [3, 7, 12]. With linked lists, if you have a reference to the front of the list, then you can get to anything in the list. So we'll usually have a single variable of type ListNode that refers to (or points to) the front of the list. So we began with this declaration:

```
ListNode list;
```

The variable "list" is not itself a node. It's a variable that is capable of referring to a node. So we'd draw it something like this:

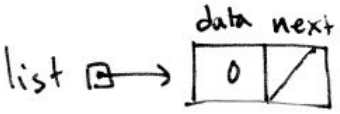
where the ListNode reference being declared is currently pointing to null instead of an actual object of type ListNode. So this box does not have a "data" field or a "next" field. It's a box where we can store a reference to such an object

.

We don't have an actual node until we call new:

```
list = new ListNode();
```

This constructs a new node and tells Java to have the variable "list" refer to it:



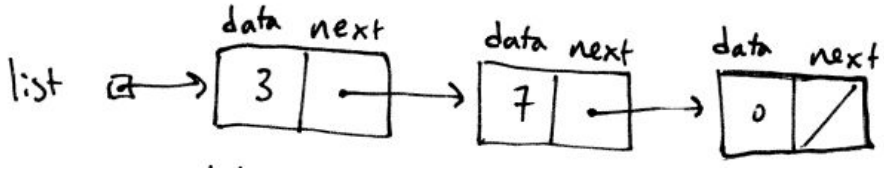
What do we want to do with this node? We want to store 3 in its data field and we want its next field to point to a new node:

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

When you program linked lists, you have to be careful to keep track of what you're talking about. The variable "list" stores a reference to the first node. We can get inside that node with the dot notation (list.data and list.next). So "list.next" is the way to refer to the "next" box of the first node. We wrote code to assign it to refer to a new node, which is why "list.next" is pointing at this second node.

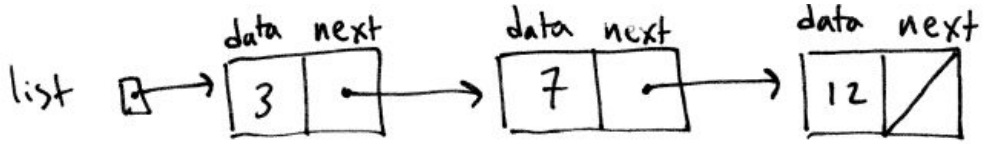
Now we want to assign the second node's data field (list.next.data) to the value 7 and assign the second node's next field to refer to a third node:

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



Finally, we want to set the data field of this third node to 12 (list.next.next.data) and we want to set its next field to null. The keyword "null" is a Java word that means "no object". This provides a "terminator" for the linked list (a special value that indicates that we are at the end of the list). So we'd execute these statements:

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



We draw a diagonal line through the last "next" field as a way to indicate that it's value is null. The assignment to null is actually unnecessary. Java will initialize all data fields to the "zero equivalent" for that particular type. For type int, that means initializing to 0. For double, it initializes to 0.0. For boolean, it initializes to false. For arrays and other objects, it initializes to null.

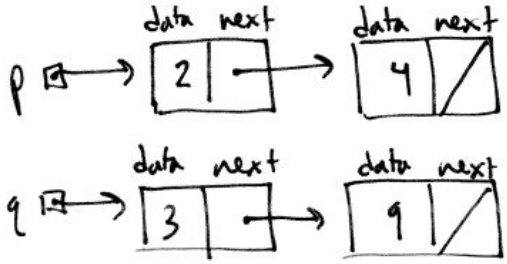
It is a good idea to add some constructors to the ListNode object to make them easier to work with. We added several constructors to the ListNode class, and discovered that the list we set up before can actually be created in a single line of code:

```
ListNode list = new ListNode(3, new ListNode(7, new, ListNode(12)));
```

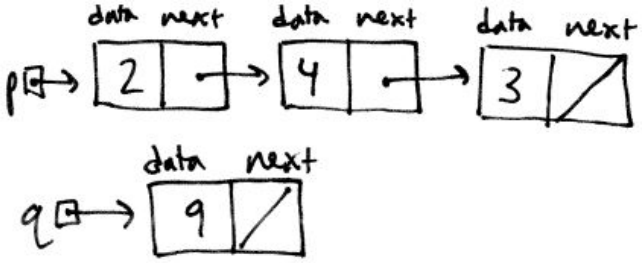
This sets up a list local variable, which is a ListNode reference to a new ListNode, which has a data field storing 3 and a next field storing a reference to a new ListNode object, which has a data field storing 7 and a next field storing a reference to a new ListNode object, which has a data field storing 12 and a next field storing a reference to the default ListNode object, which is null.

In section we will go over different exercises that involve list operations. Each will have a "before" picture and an "after" picture. The challenge is to write code that gets you from the one state to the other state.

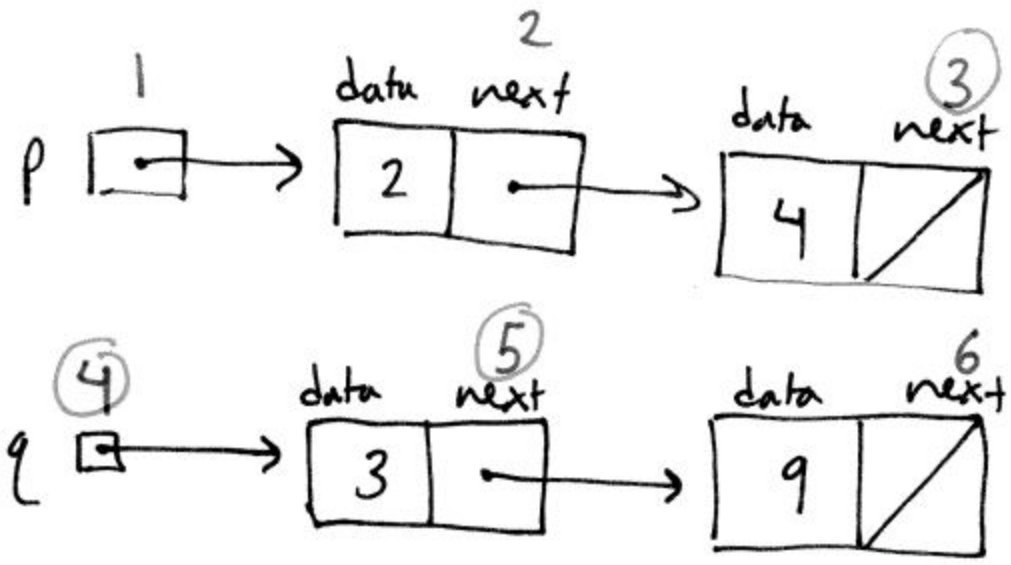
As an example, suppose that you have two variables of type ListNode called p and q and that you have the following situation:



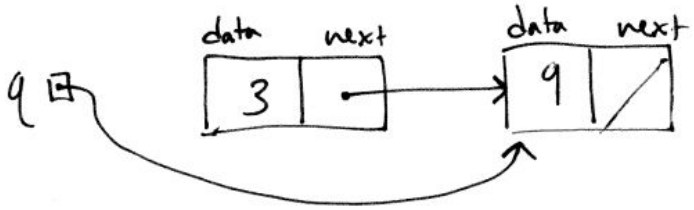
and you want to get to this situation:



How many variables of type ListNode do we have? The tempting answer is two (probably thinking of p and q). The other tempting answer is four (probably thinking of p, q and the two non-null links). But in fact, there are six different variables of type ListNode (numbered 1-6 in the following image). The names of those variables are: `p, p.next, p.next.next, q, q.next`, and `q.next.next`. In order to change the before picture to the after picture, we need to modify 3 of the references. These references are circled in the following image (3, 4, 5).

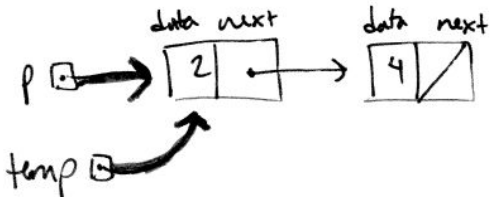


If we change them appropriately, we'll be done. But we have to be careful of how we do so. Order can be important. For example, suppose we were going to start by changing box 4. In the final situation, it's supposed to point at the node with 9 in it. But if we started with that change, then what would happen to the node with 3 in it?

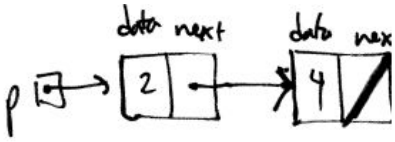


We would no longer have a reference to the ListNode storing 3 as the data. Java would come along and garbage collect that memory for us. This introduces an idea to us about references that are safe to change. In order for a reference to be safe to change, it should either be null, or it should be pointing to a ListNode that has multiple references pointing to it. In either case, we are not dropping a ListNode.

In this situation it is safe to change p or temp, because the other one will maintain the reference to the ListNode storing 2:

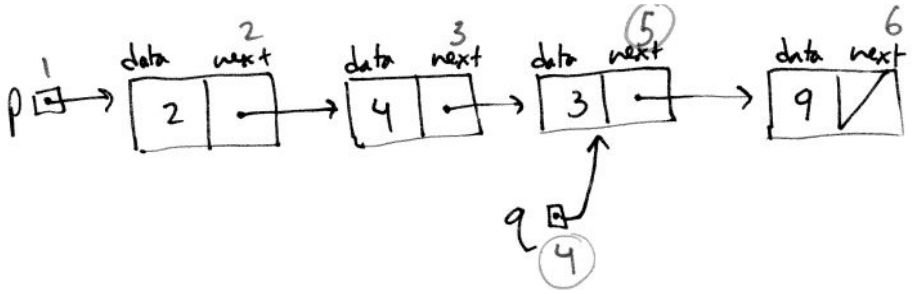


In this situation it is safe to change p.next.next, because it is null and therefore not storing a ListNode currently:



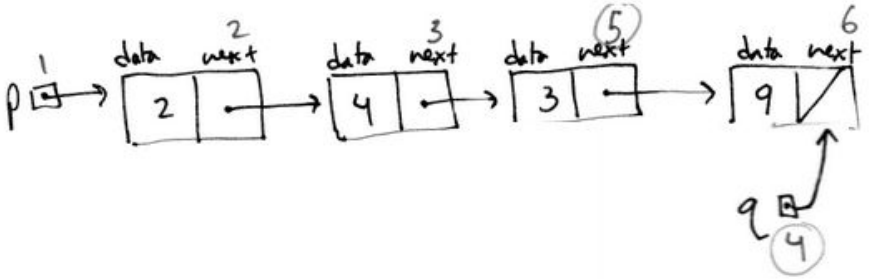
Of the three values we have to change to solve our problem, the one that is safe to change is reference 3 because it's currently null. So we begin by setting it to point to the node with 3 in it:

```
p.next.next = q;
```



Now that we've reset reference 3 to point at the box that reference 4 is pointing at, there are two references storing the same ListNode, and we can reset reference 4. It's supposed to point to the node that has 9 in it. We can do this by "leap frogging" over the current node it's pointing to:

```
q = q.next;
```



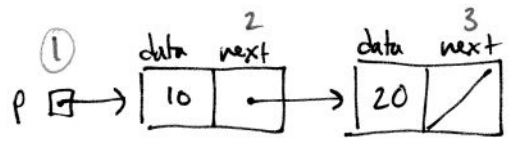
Now we just have to reset reference 5. But we can no longer refer to reference 5 as q.next because we've changed q. Now we have to refer to it this way:

```
p.next.next.next = null;
```

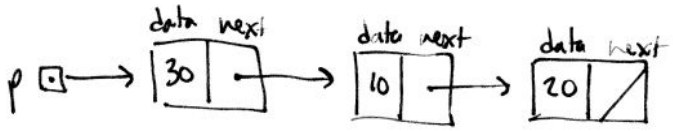
Putting these three lines together, we see the code that is needed to get from the initial state to the final state:

```
p.next.next = q;
q = q.next;
p.next.next.next = null;
```

We tried another example with a new before and after goal.
Before:

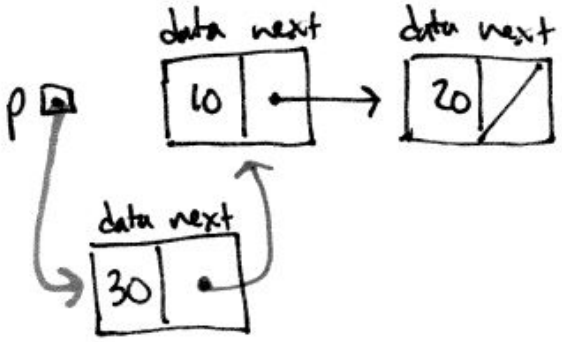


After:



In order to make this change, we have to modify the reference labelled as ‘1’ in the before picture. We will need to create a new ListNode to complete this problem, since we only have two ListNode objects in the before picture, and three ListNode objects in the after picture. Because of the way Java does assignment evaluation, we can actually do this in one line of code.

```
p = new ListNode(30, p);
```



The way that Java evaluates this, is first it makes a new ListNode, with 30 stored as the data and with the next field pointing to whatever ListNode object p is pointing to. Then it breaks the original reference for p, and reassigns p to point at the new ListNode object. We could also break this up into several lines of code, but we have to be careful of order to not lose nodes.

```
ListNode newNode = new ListNode(30);
newNode.next = p;
p = newNode;
```

If we switch the order of the last two lines, we would lose our original list.

It is essential that you draw pictures to keep track of what is pointing where and what is going on when this code executes. It's the only way to master linked list code. We'll practice these small problems in section so that in lecture on Wednesday we can turn to the question of how to use loops to do more generalized processing of linked lists.