Linked Structures

See Section 3.2 of the text.

First, notice that Java allows classes to be recursive, in the sense that a class can have an element which is itself an object of that class:

```
class Person {
        String name;
        Person spouse;
        ….
}
```

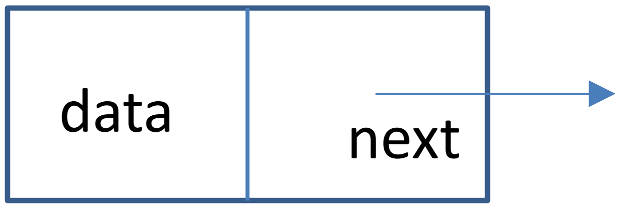
We can use this to build linked structures of almost any shape or complexity.   Each element of the structure is a node. A node can hold data, and it also has links to other nodes.

For example, for a linear structure we might use a node class like this:

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

        public Node() {
                ...
        }
}
```

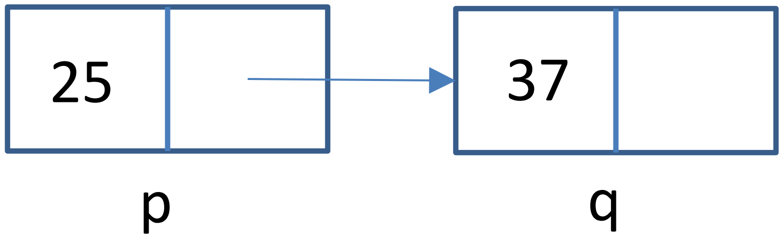
We can think of a Node as a box with 2 fields:



The code

```
p = new Node();
p.data = 25;
q = new Node();
q.data = 37;
p.next = q;
```

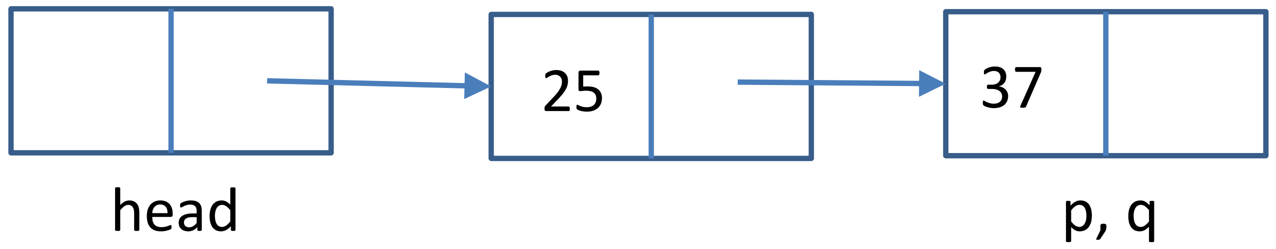
creates the following structure:



The code

```
head = new Node();
p = head;
q = new Node();
q.data = 25;
p.next = q;
p = q;
q = new Node();
q.data = 37;
p.next = q;
p = q;
```

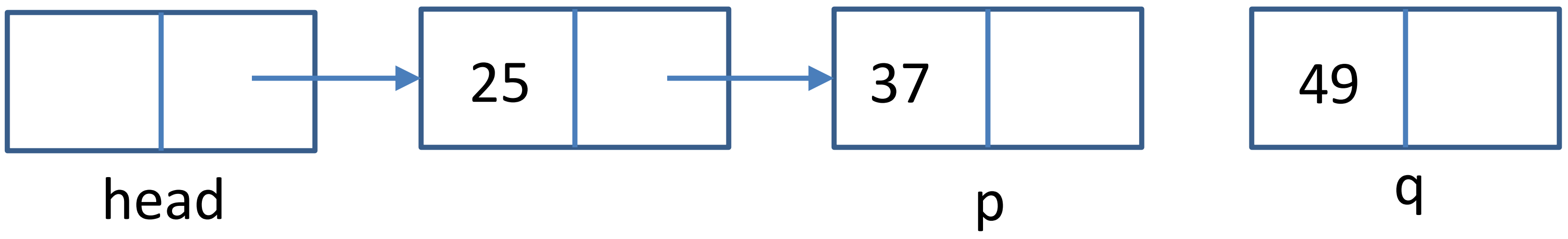
creates the following structure:



If we now say

```
        q = new Node();
        q.data = 49;
```

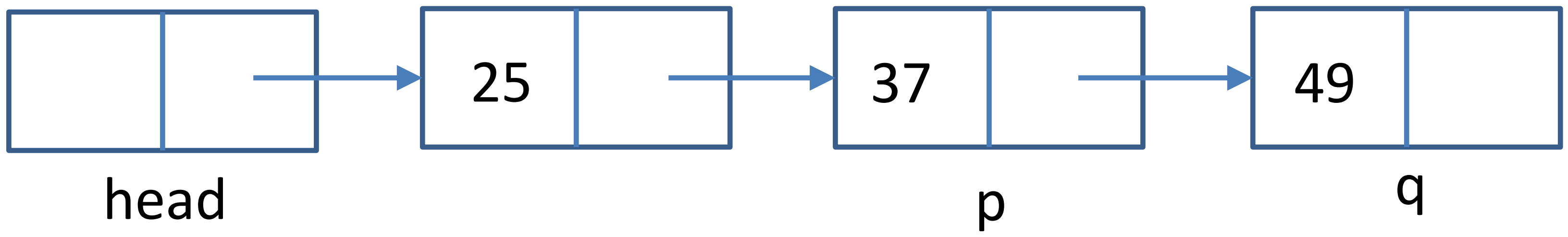
the picture becomes



The statement

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

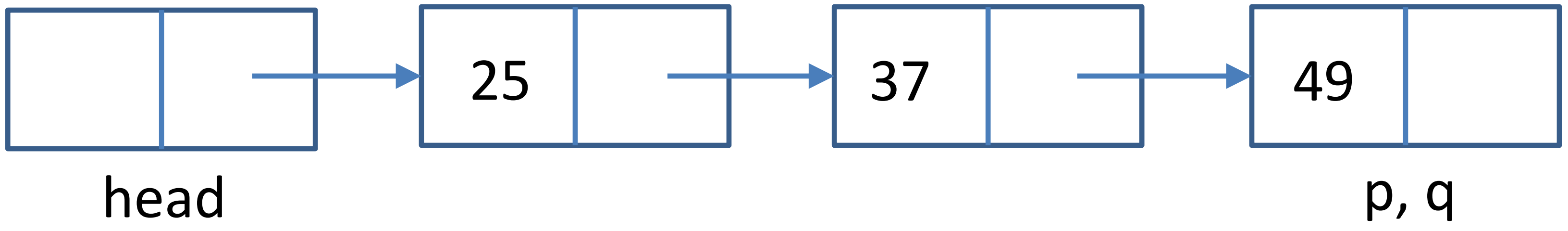
links p's box to q's box



The statement

```
p = q;
```

doesn't alter the nodes at all, but it makes variable p refer to the same box as variable q:

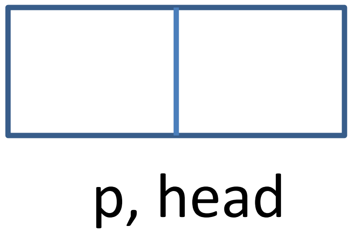


Of course, we can keep going; there is no limit to the number of nodes you can construct and link together.

Here is an example of a circular structure using the same Node class.

We start off

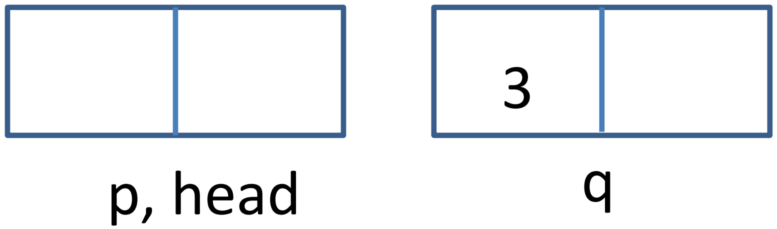
```
head = new Node();
p = head;
```



The code

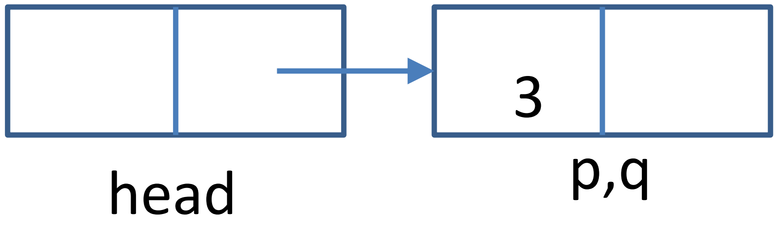
```
q = new Node();
q.data = 3;
```

makes this:



We now set some links. First

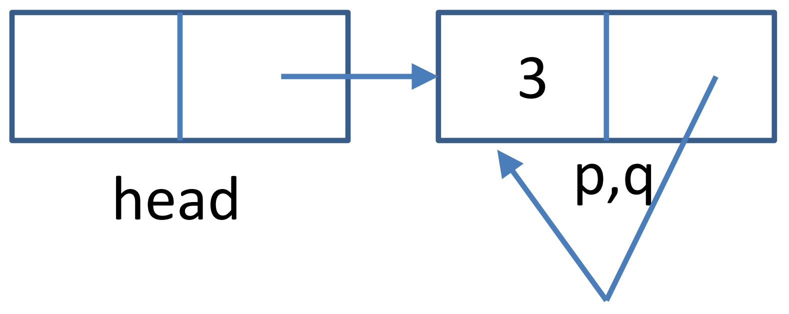
```
p.next = q;
p = q;
```



We now say

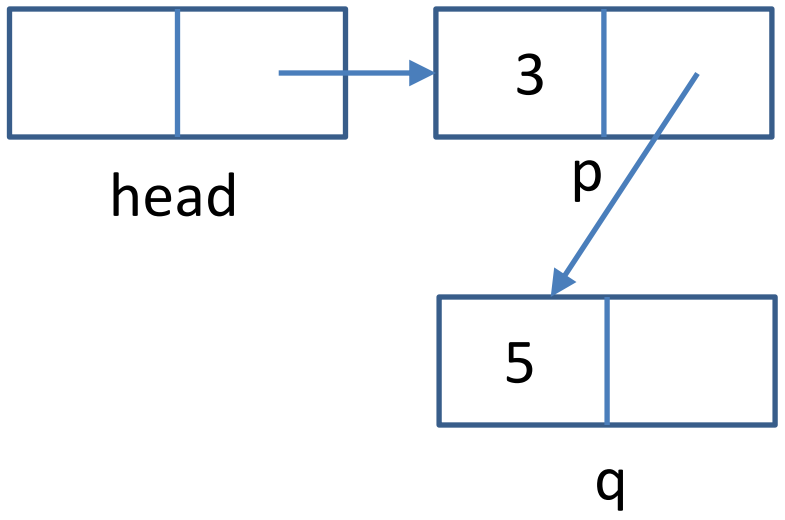
```
        q.next = head.next;
```

Since head.next is currently the same node as q, this makes q point to itself:



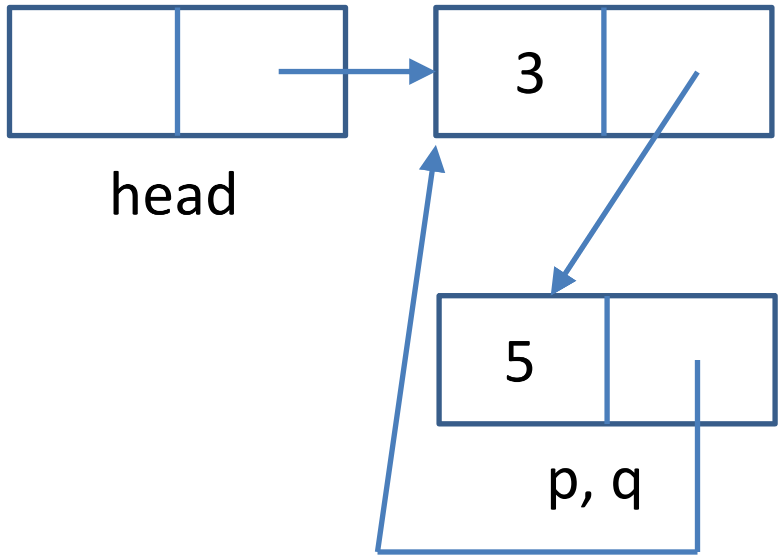
Now do these four statements again (with new data):

```
q = new Node();
q.data = 5;
p.next = q;
q.next = head.next;
```



Two more statements complete the cycle:

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

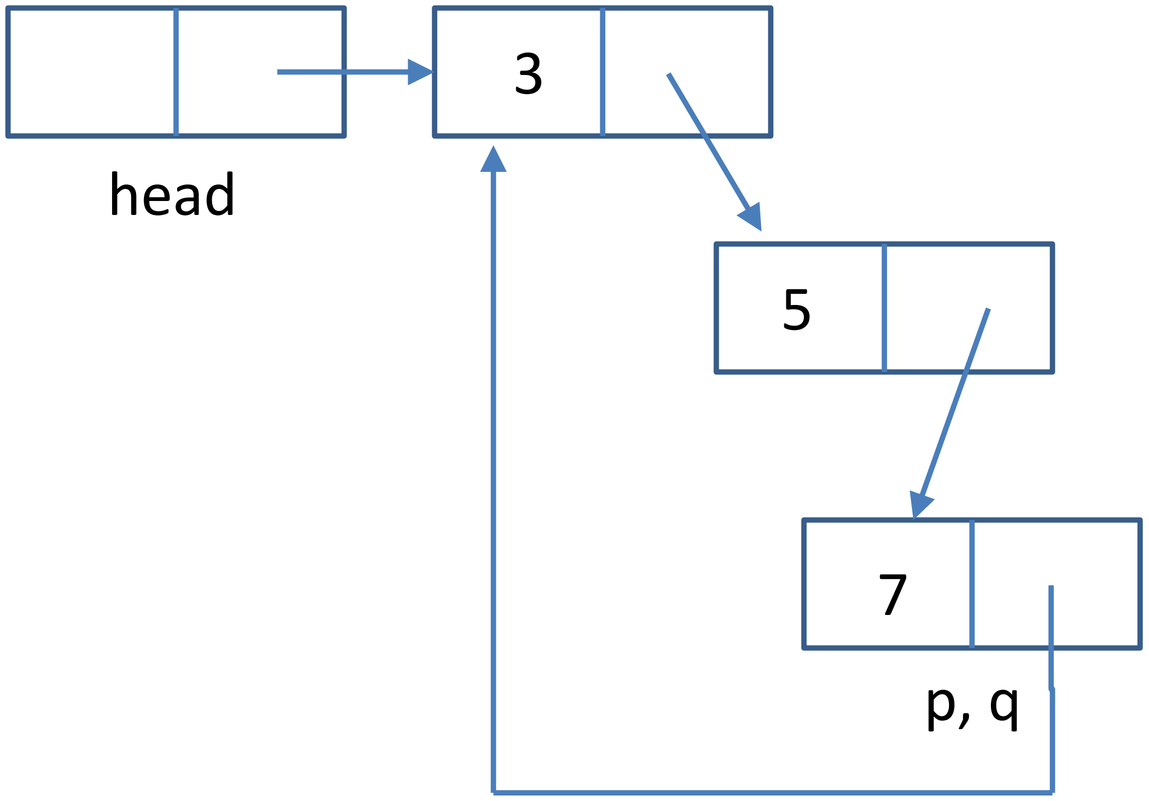

3
head
5
p, q

Once more. First we make a new node, put data in it, and connect our structure to it:

```
q = new Node();
q.data = 7;
p.next = q
```

q is now the last box in the list; we make it point back around at the first:

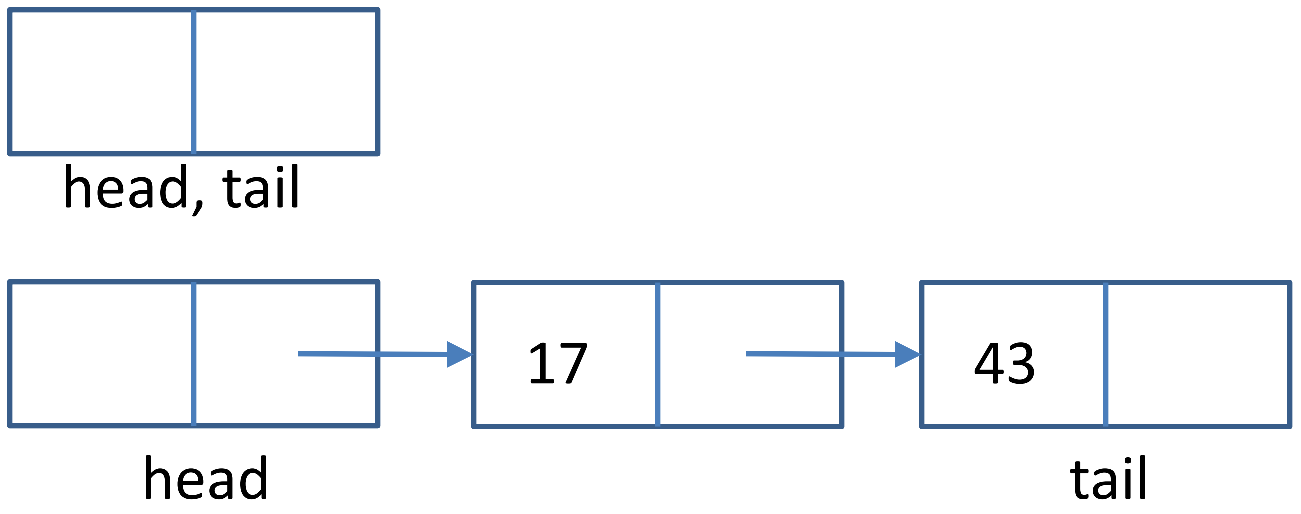
```
q.next = head.next;
p = q;
```



Note the role of the "head" node in these examples. It is a node with no data whose only role is to point at a feature of the structure, which in these examples is the first node of the structure. Such a node is called a "sentinel".

Good use of sentinels is one of the keys to successful programming with linked structures. Among other things, they can help you avoid special cases, a likely source of bugs.

For example, suppose you are making a simple linked list. Here are pictures of an empty list and a list with two data values:



In either case the code to add a node with datum 25 at the end of the list is

```
q = new Node();
q.data = 25;
tail.next = q;
tail = q;
```

If we want to add a node with datum 16 at the start of the list there is a special case to worry about, because if the list is initially empty we will need to update the variable *tail*:

```
q = new Node();
q.data = 16;
if (head==tail) { /* the list was empty */
        tail = q;
        head.next = q;
}
else {
        q.next = head.next;
        head.next = q;
}
```

You should form the habit of always drawing pictures of your linked structures and using those pictures as a guide to your coding. If the picture works your code will work.

Now, why do we do this (other than it is great fun)??

For one thing, this gives us natural implementations of trees and graphs and other non-linear structures that aren't easy to model in other ways.

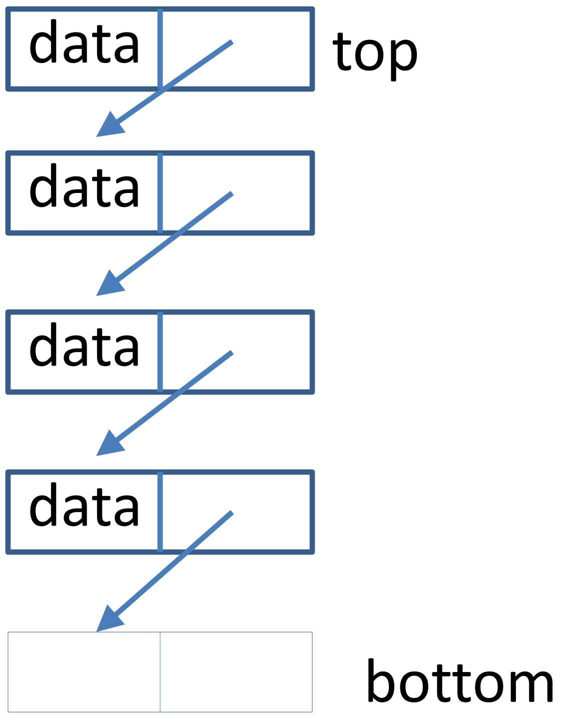
Sometimes, when we are dealing with large amounts of data, linked structures can be implemented more efficiently than array-based or ArrayList-based structures.

For example, suppose we have a queue with n elements based on an ArrayList.

How would you estimate the worst-case running time of the enqueue() and dequeue() operations?

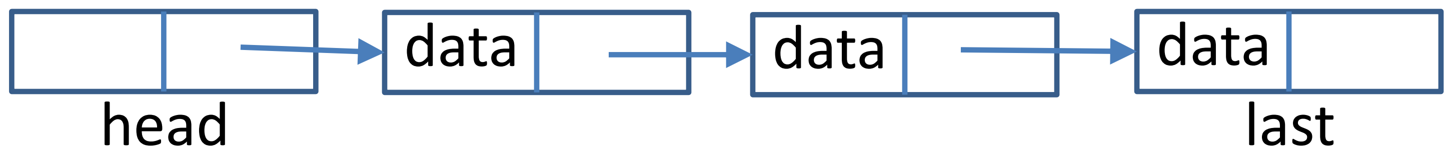
With a good linked structure they are both O(1).

Here is a picture for a Stack structure



How would you initialize or construct a Stack? How would you write Push(), Pop(), Top() and IsEmpty()??

Here is a picture of a Queue structure:



How would you construct an empty Queue, and write Enqueue(), Dequeue(), Front() and IsEmpty() to go with this picture?