Linked Structures

Labs 3, 4, and 5 all build structures dynamically, one block at a time. This is a very useful programming technique, but it is one where you can get into a lot of trouble with sloppy coding. I can't emphasize enough how important it is to draw pictures of the structures you create. You should always have pen and paper with you when you are doing this kind of coding. Professional programmers , even people who have been doing this for years, draw pictures to go with their code. If you can make your pictures work, writing code to go with your pictures is easy.

Consider a simple Person class, where people have names and friends:

```
public class Person {
        String name;
        Person friend;

        public Person(String name) {
                this.name = name;
                this.friend = null;
        }
        …..
}
```

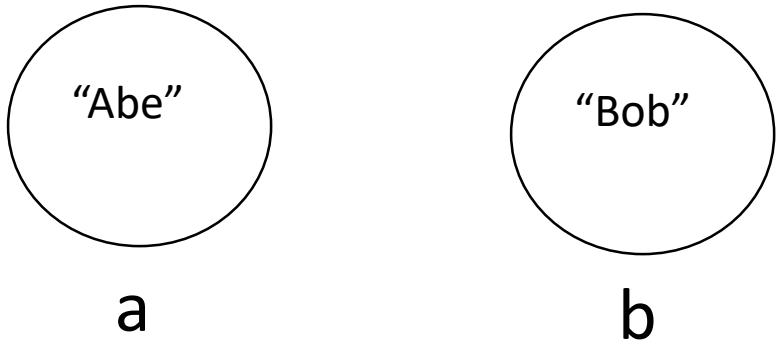
Don't be surprised that a Person can have an attribute *friend* that is also a Person. Just remember that the friend needs to be constructed. A friend is not a String "Mary" but a Person whose name is "Mary".

I am going to represent constructed people by circles.

The code

```
Person a = new Person( “Abe” );
Person b = new Person( “Bob” );
```

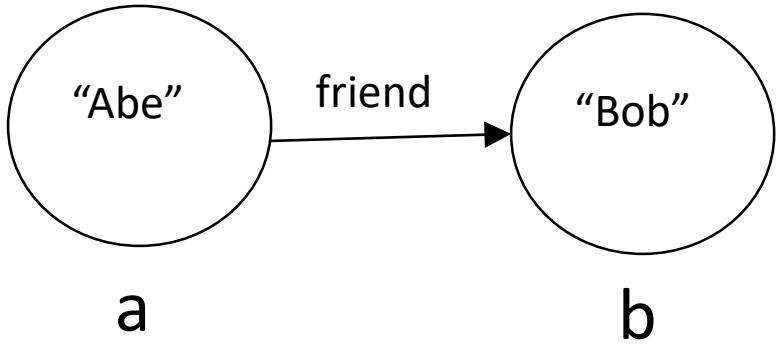
gives two objects:



If we then say

```
a.friend = b;
```

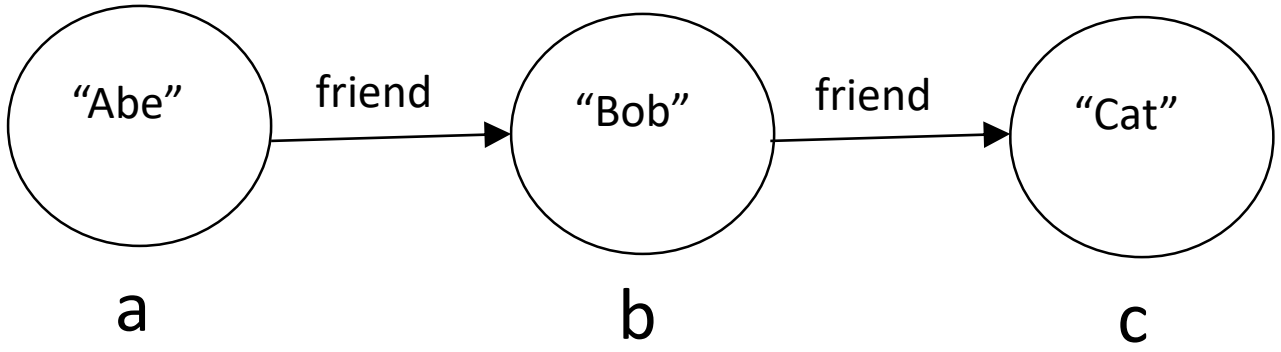
we get the following picture:



We could add more people to this. For example

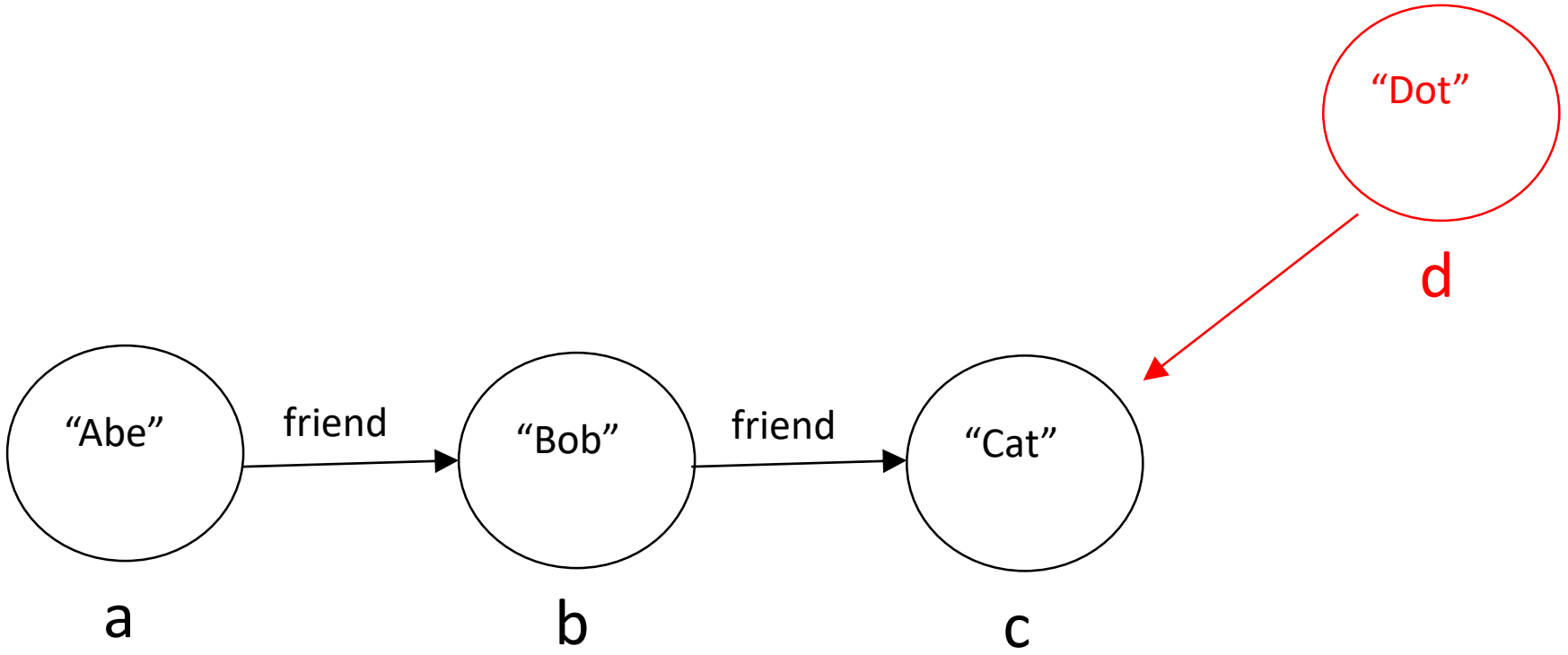
```
Person c = new Person("Cat");
b.friend = c;
```

will result in



Alternatively, we could go from a picture to code.

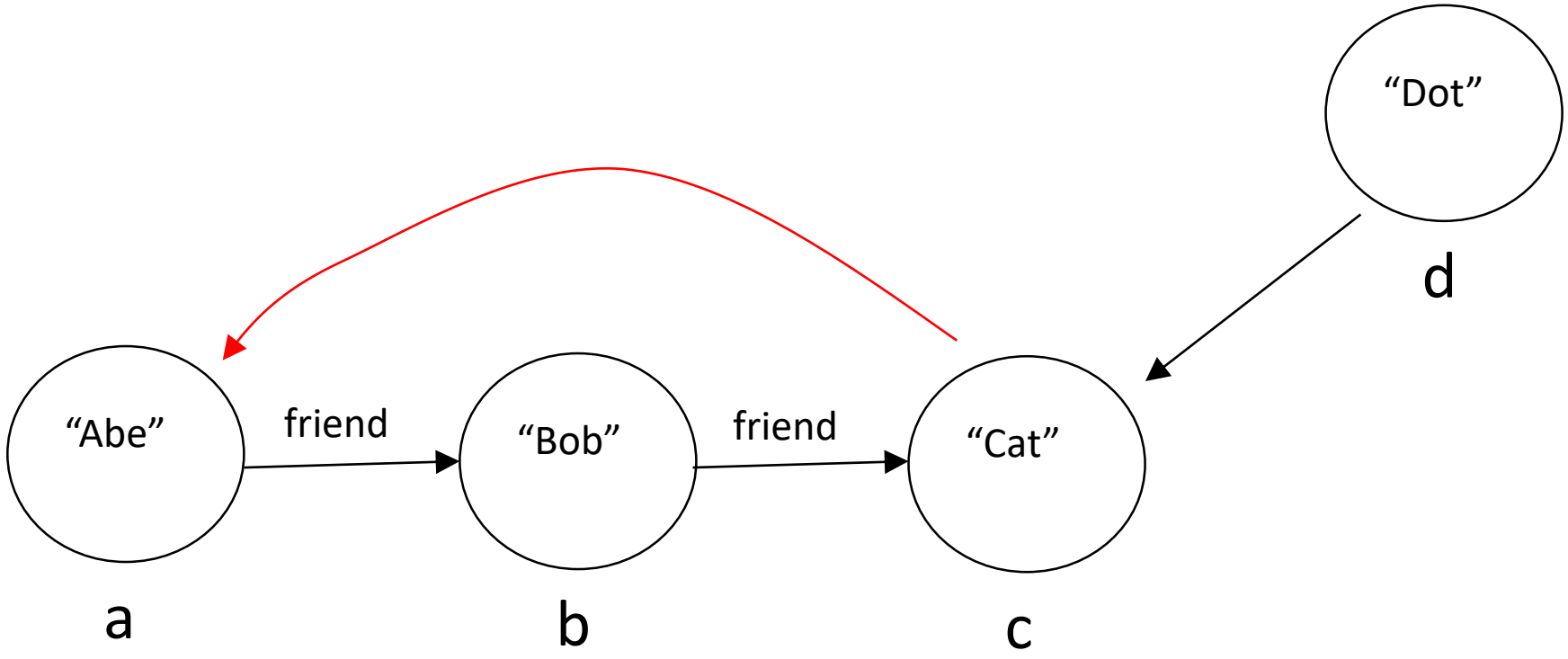
What would we need to add to our code to produce the following?



Answer:

```
Person d = new Person(“Dot”);
d.friend = c;
```

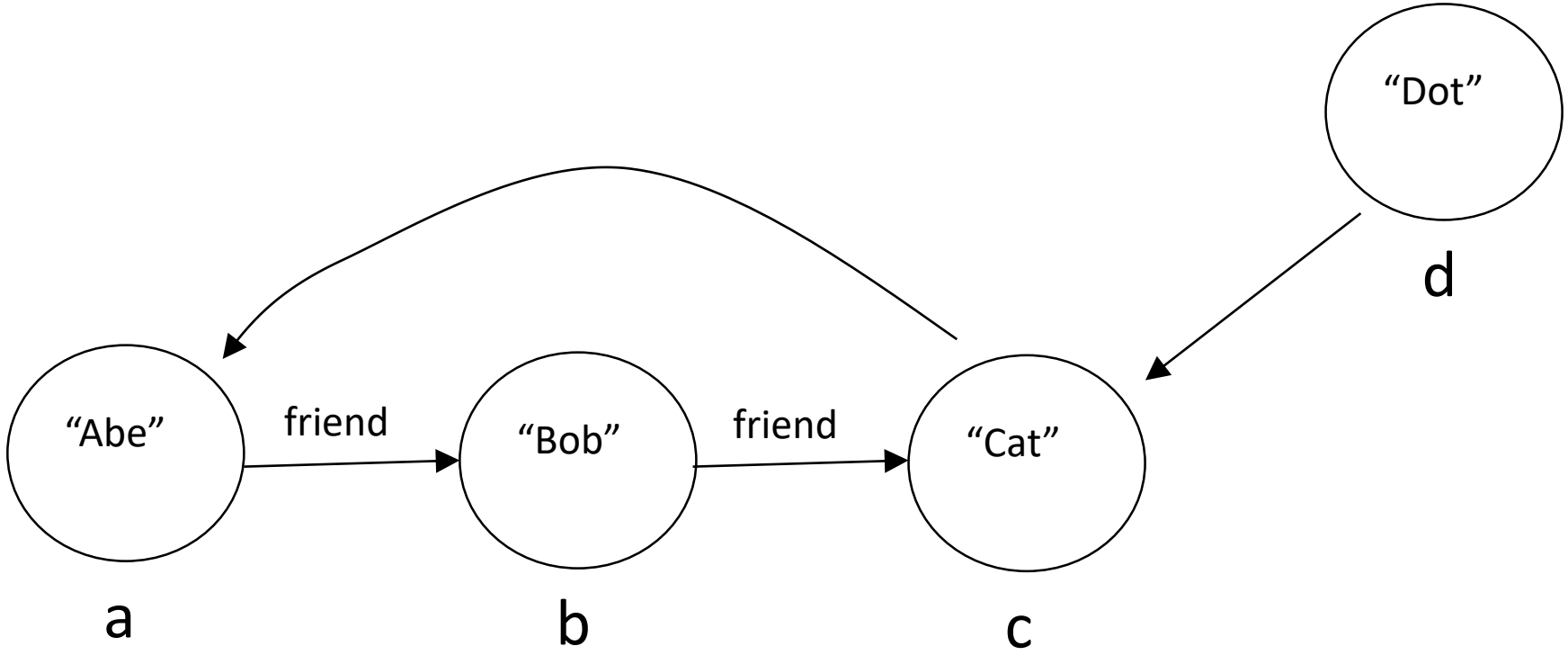
And this picture?



Answer: c.friend = a;

Even with this simple Person class we could make some very complex relationships. This flexibility is what makes this approach attractive for storing data structures.

Here is the complete code we used to create this structure:



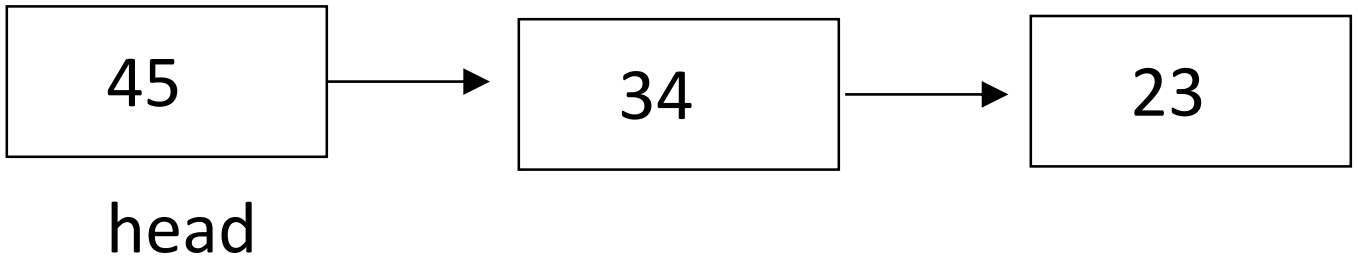

```
Person a = new Person( “Abe” );
Person b = new Person( “Bob” );
a.friend = b;
Person c = new Person(“Cat”);
b.friend = c;
Person d = new Person(“Dot”);
d.friend = c;
c.friend = a;
```

Now let’s think about storing integers. Instead of Person (with name and friend attributes) we will make a structure that I call Node (with data and next attributes):

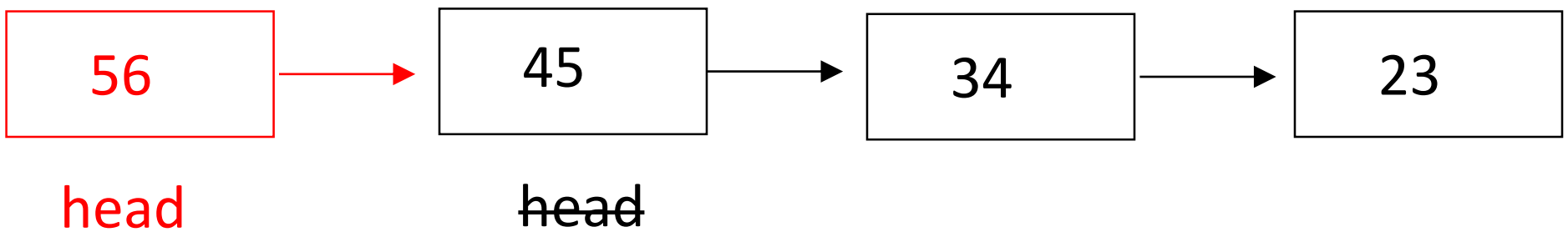
```
class Node {
        int data;
        Node next;

        public Node (int data) {
                this.data = data;
                this.next = null;
        }
        ….
}
```

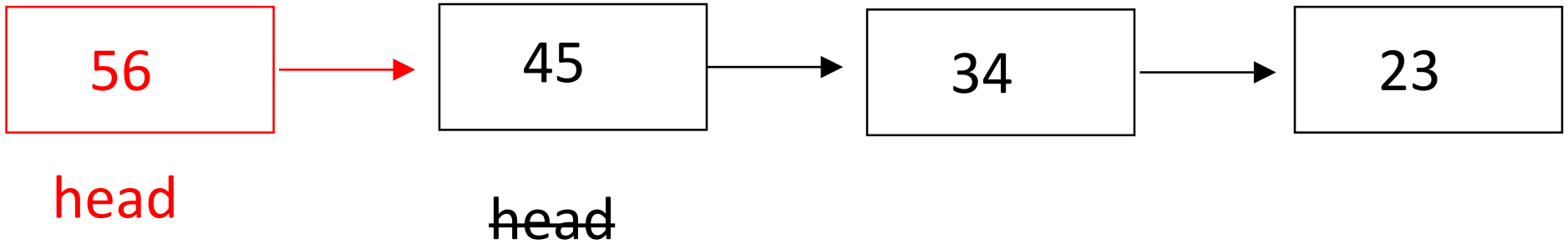
I will draw Nodes as rectangles instead of circles. To save time I will stop labeling my arrows; they all represent *next* fields.

Suppose we have the following structure:



What code will turn it into this? The things in red are new:

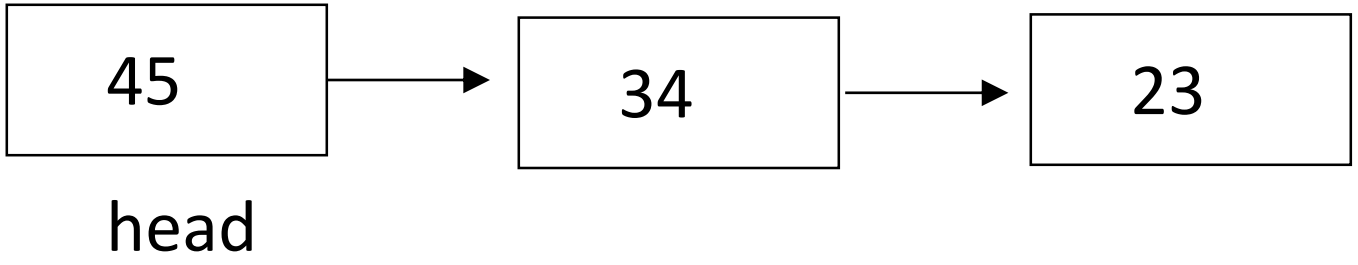




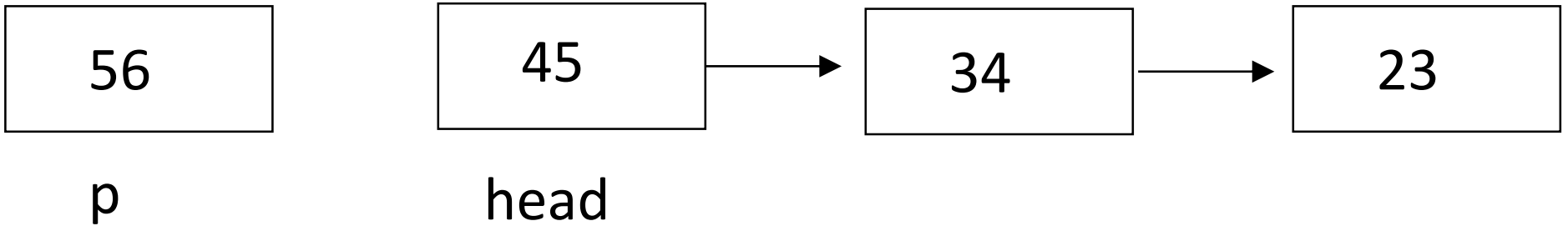
Answer:

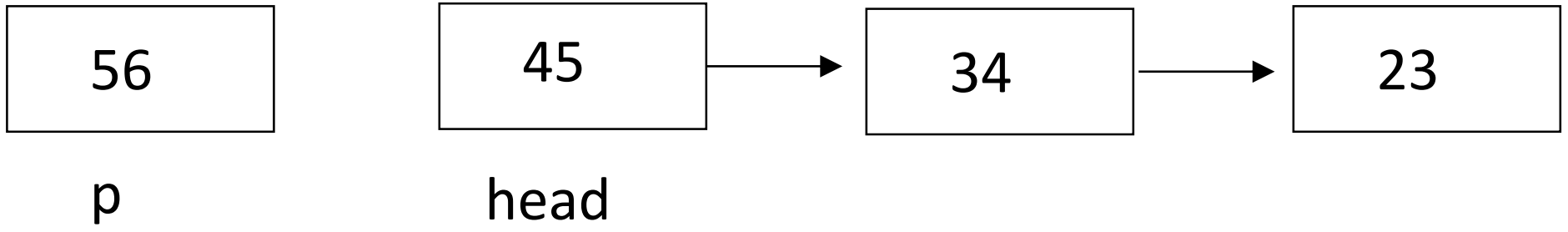
```
Node p = new Node(56);
p.next = head;
head = p;
```

This is important, so let's go through it one step at a time.
We start with this:

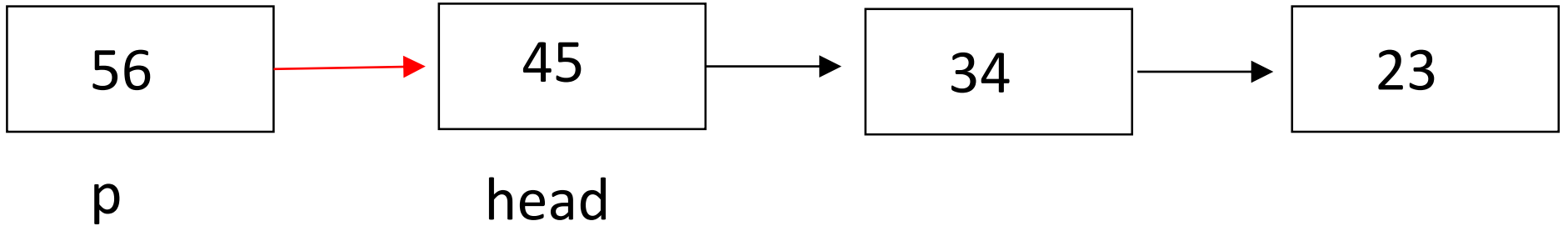


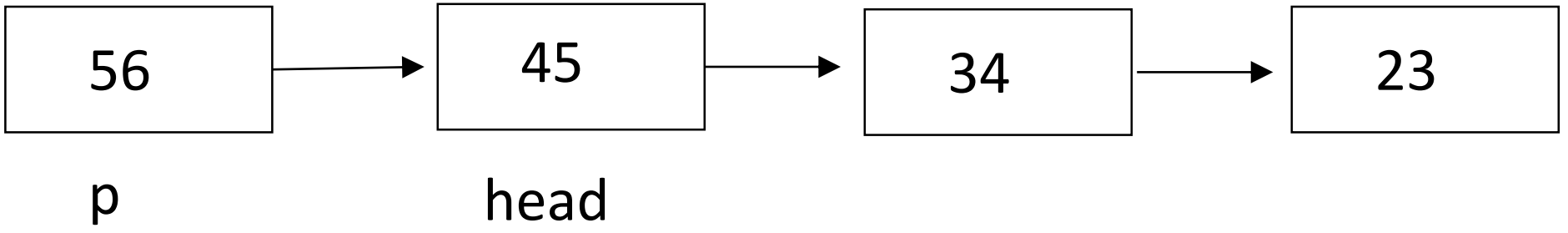
Node p = new Node(56); Does this:



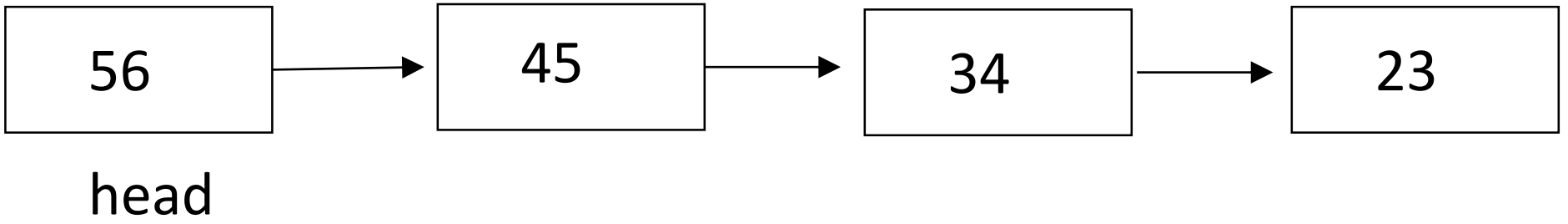


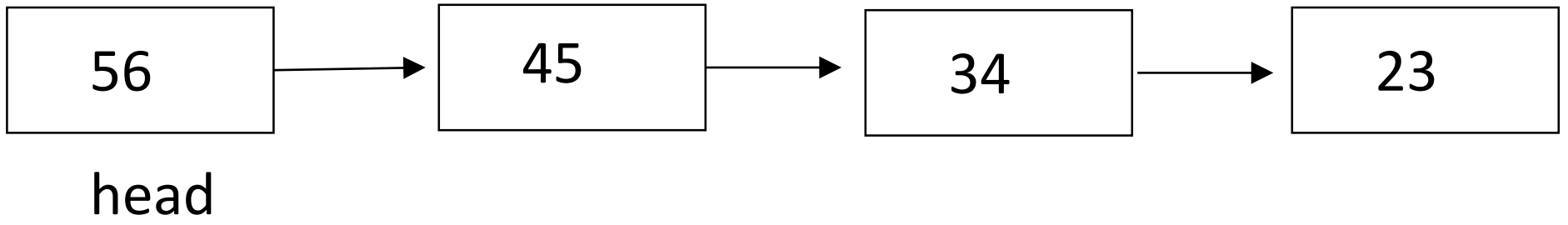
Saying p.next = head; does this; it is just like saying a.friend = b;



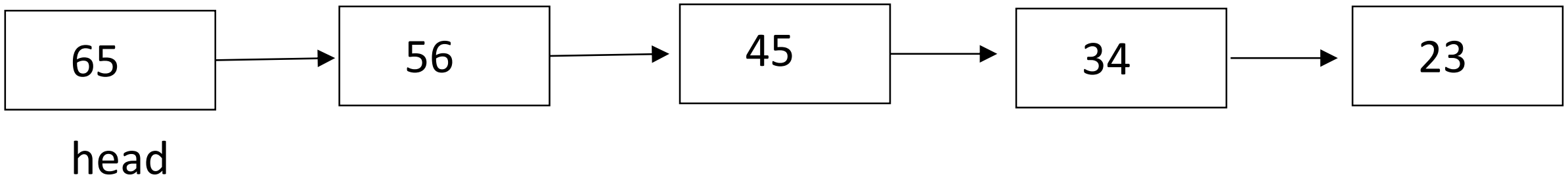


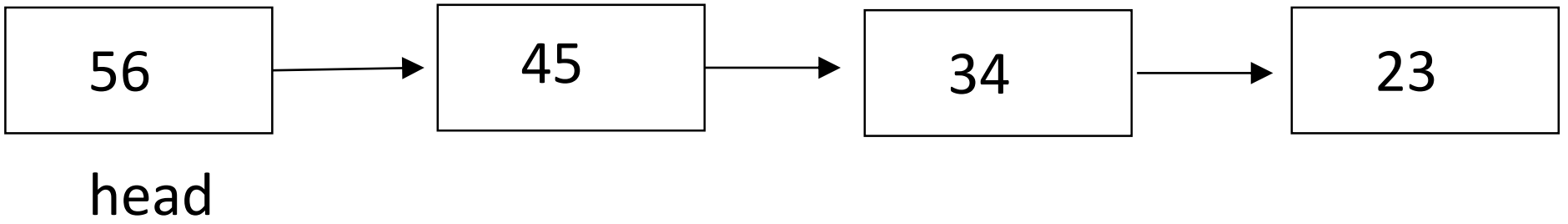
Finally, we want variable *head* to refer to the first node, which is now variable p. So we say head = p;





Okay, let's do it again. What code will add a node with 65 onto this structure, producing this?

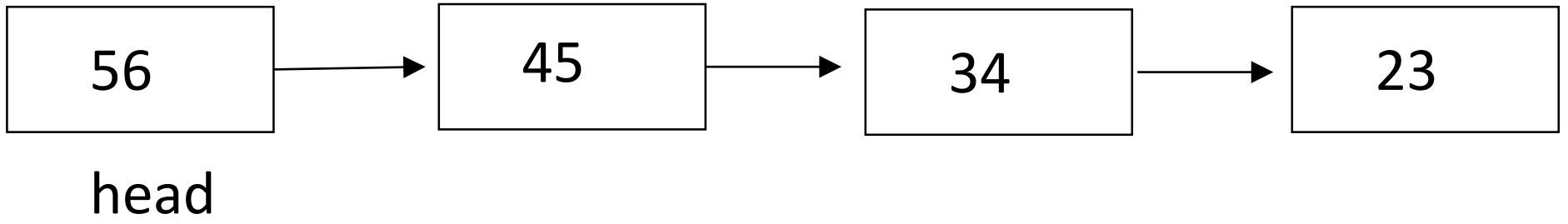




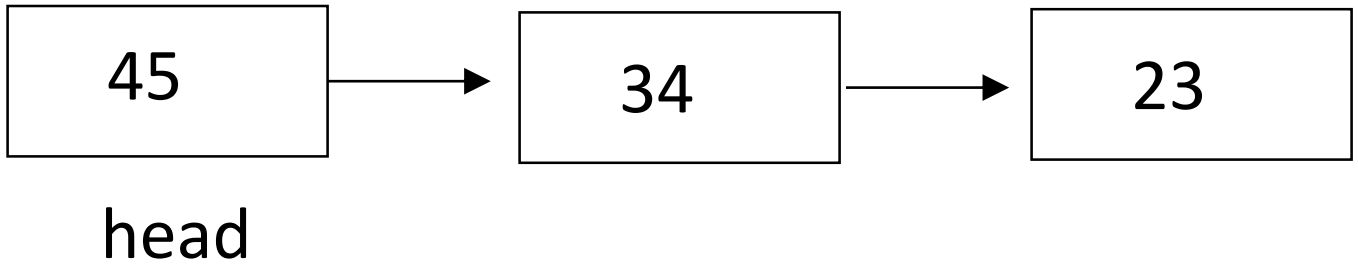
Answer:

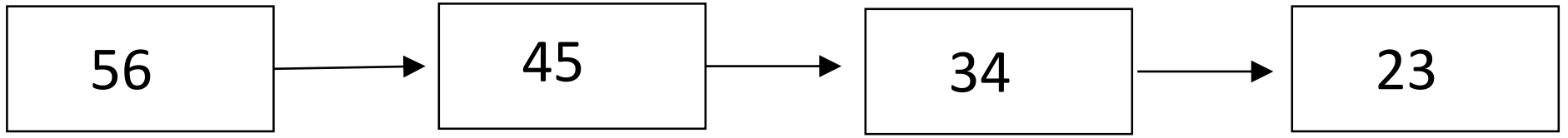
```
p = new Node(65);
p.next = head;
head = p;
```

We could make this structure as long as we want. But could we make it smaller? Could we turn



into



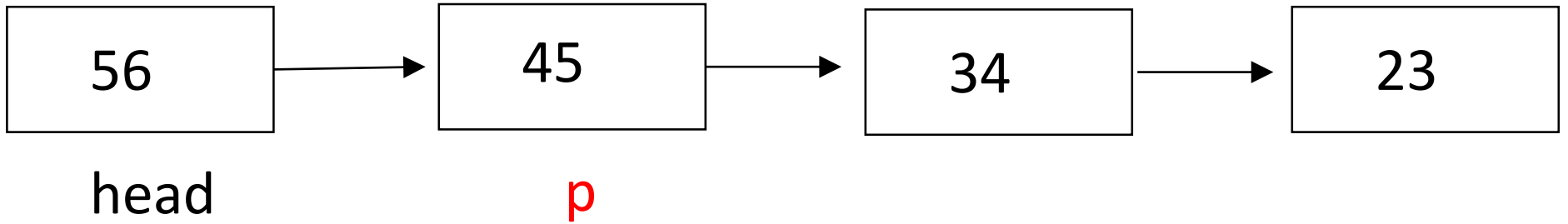


Sure. This code will do it:

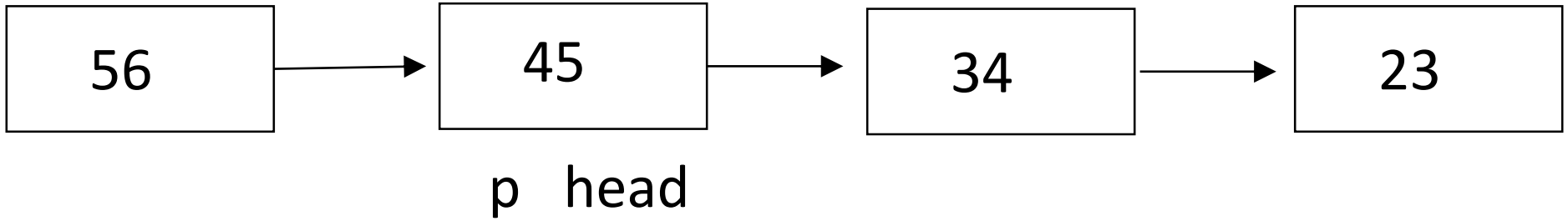
```
Node p = head.next;
head = p;
```

Be sure you understand that.

Node p = head.next; does this:



Then head = p; does this:



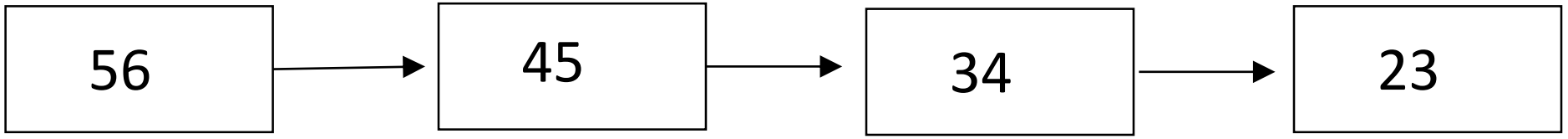
It doesn't matter that there is a box pointing at the box stored in both p and head. There is no way to access that box; it will eventually be garbage-collected.

Note that the two statements

```
Node p = head.next;
head = p;
```

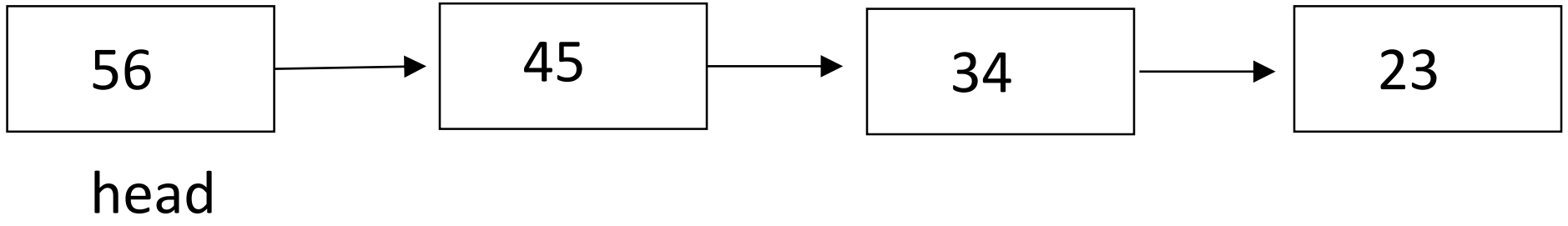
could be simplified into one:

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



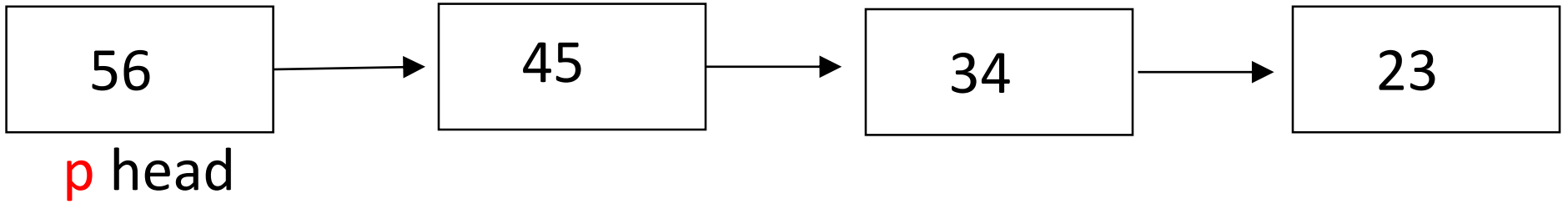
So we know how to delete the node at the head of this structure. Could we delete the node with value 45?

Sure. The idea is to put a pointer *p* before the node we want to delete and another pointer *q* after that node, and then say p.next = q;

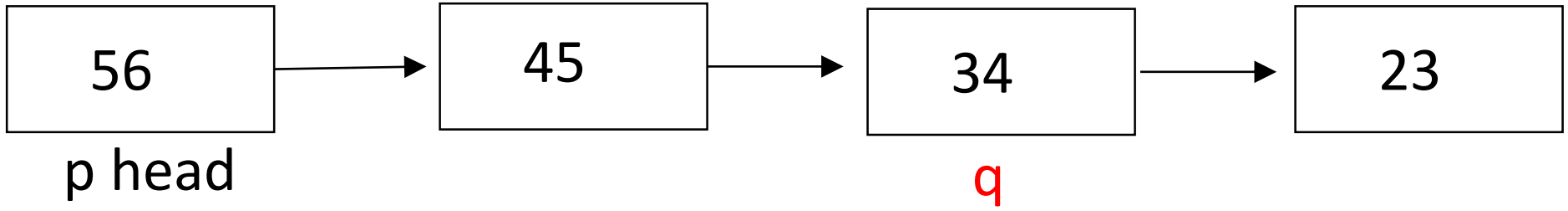


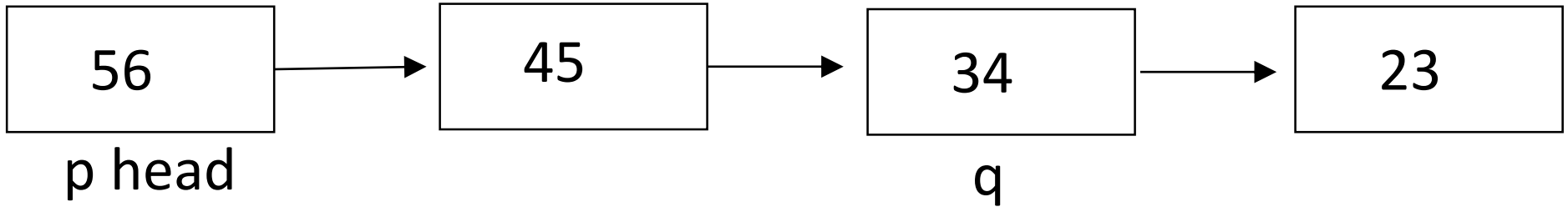
Here is the code:

```
Node p = head;
```

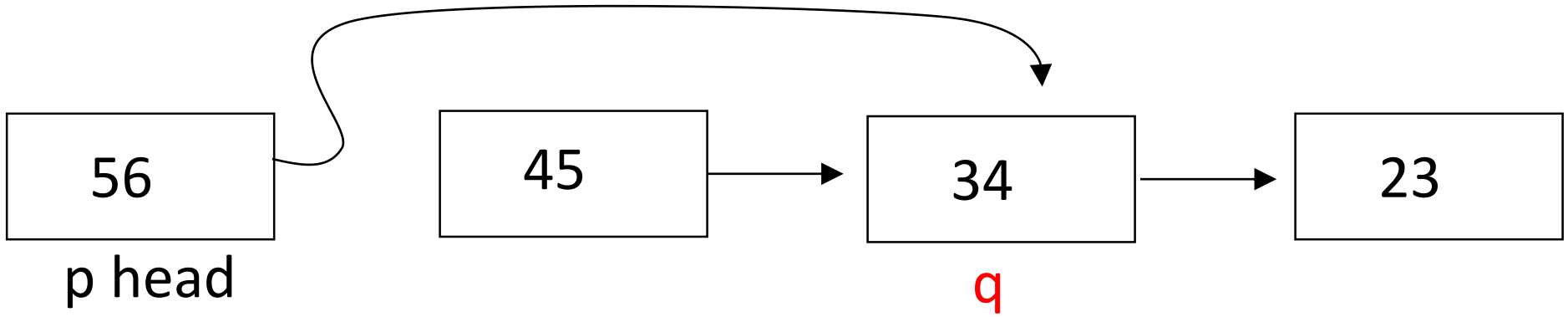



```
Node q = head.next.next;
```

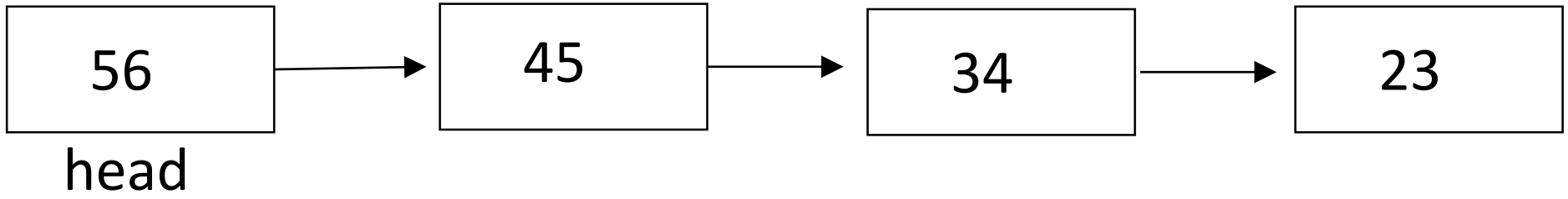




p.next = q;

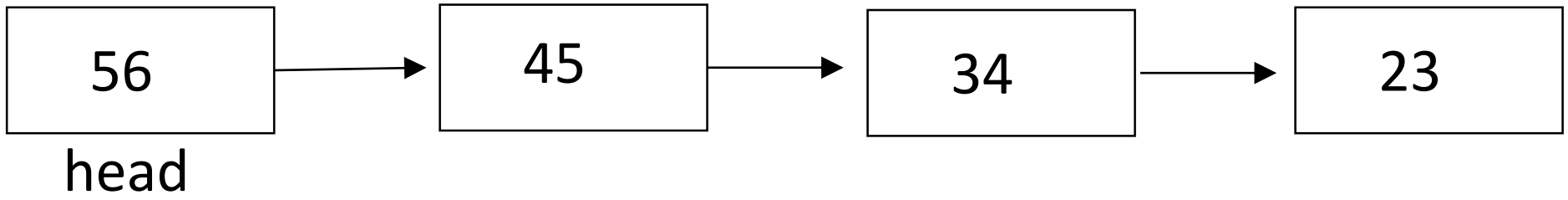


Now the linked nodes in the structure go from 56 to 34 to 23. There is no way to refer to that node containing 45; it will eventually be garbage-collected.



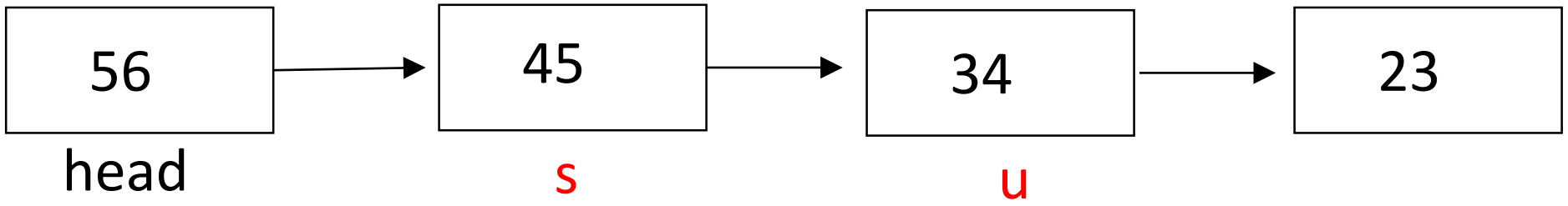
Let’s add a node with value 39 between 45 and 34.

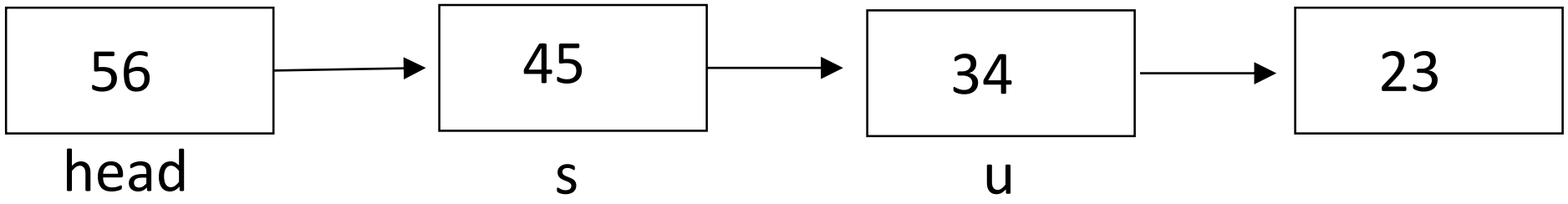
The idea is again to get variables pointing at the node before and the node after the place we want to do the insertion. I will call those nodes *s* and *u*, and the new node *t*.



First we set up pointers before and after the point of insertion:

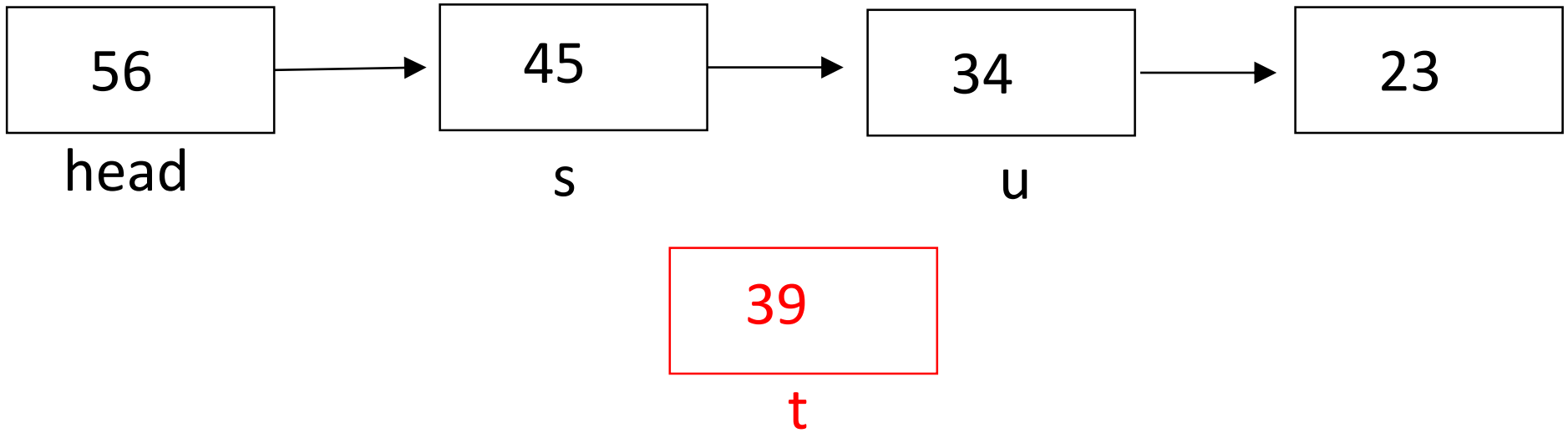
```
Node s = head.next;
Node u = s.next;
```

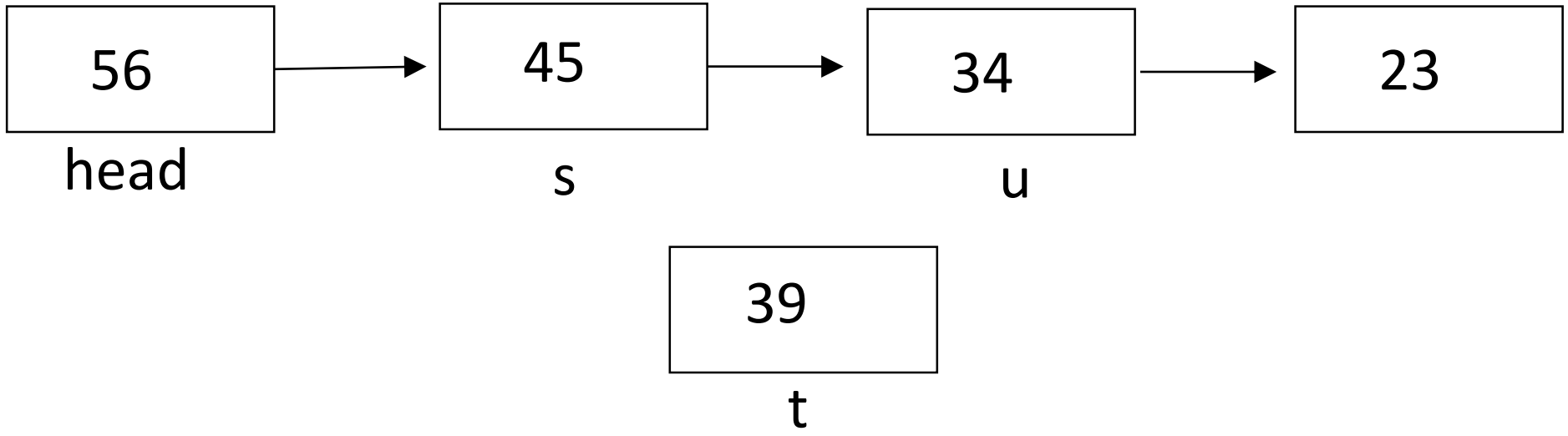




Now we make a new box and put 39 in it:

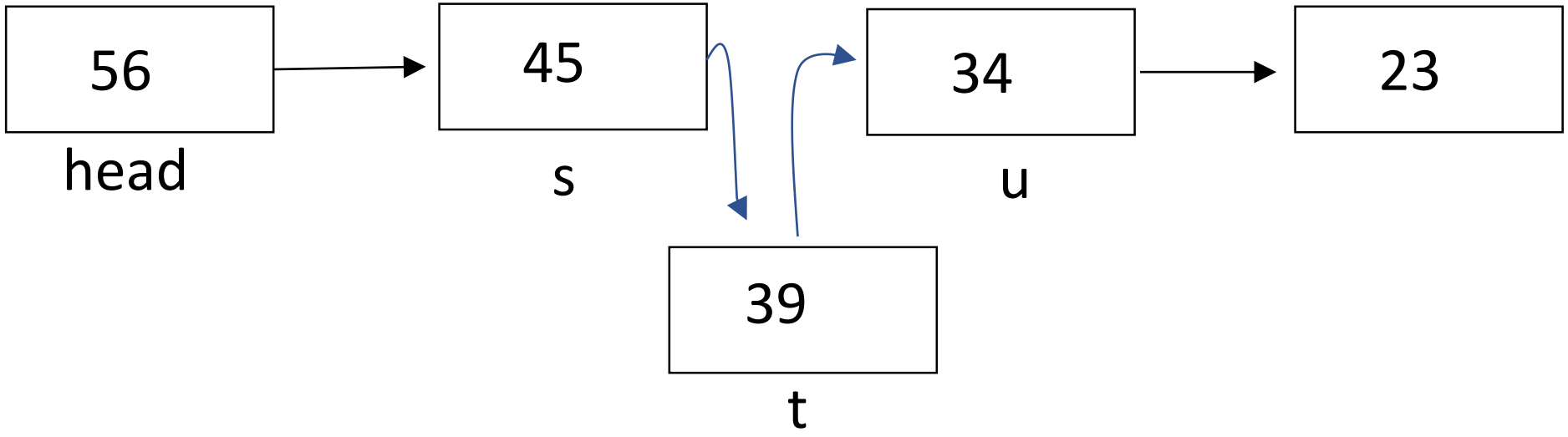
Node t = new Node(39);



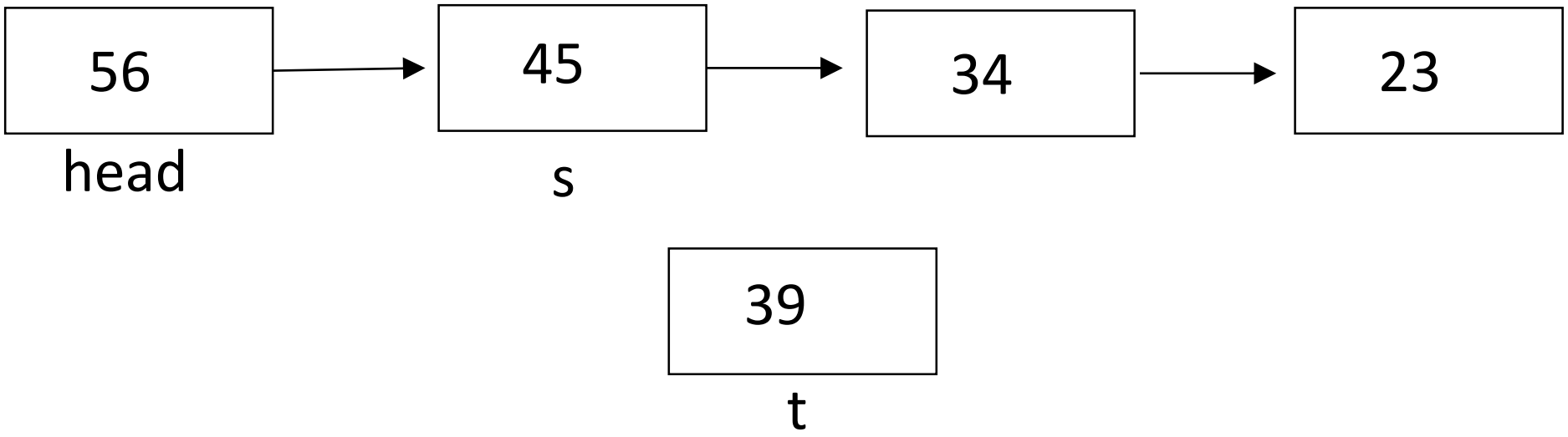


Finally, we adjust the links so s points to t and t points to u:

```
s.next = t;
t.next = u;
```



Note that we could do this without variable u if we do things in the right order:



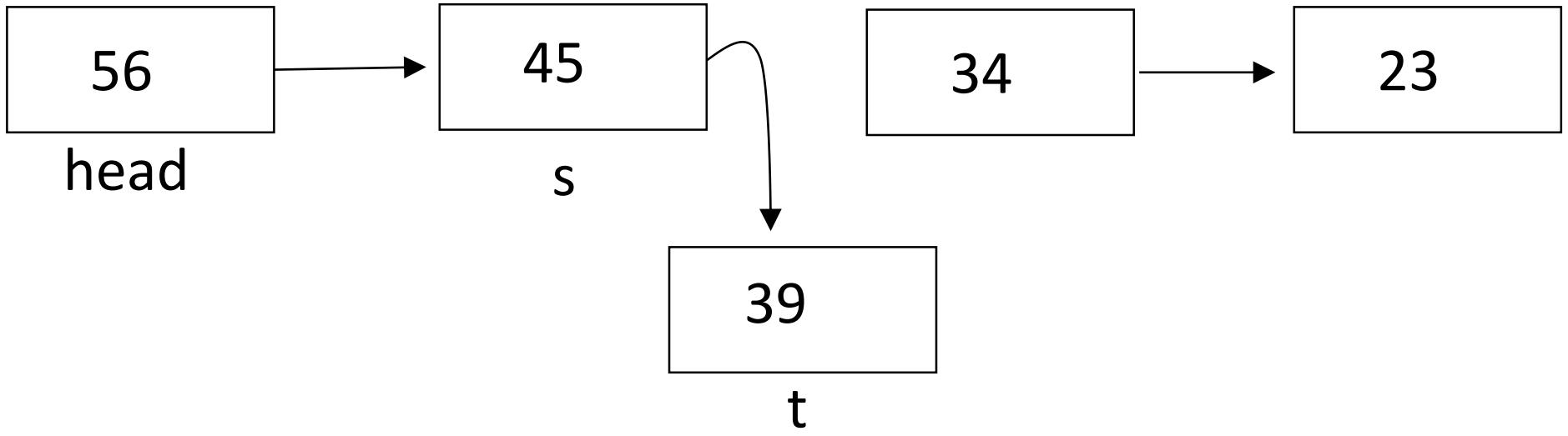
This works:

```
t.next = s.next;
s.next = t;
```

But this doesn't work:

```
s.next = t;
t.next = s.next;
```

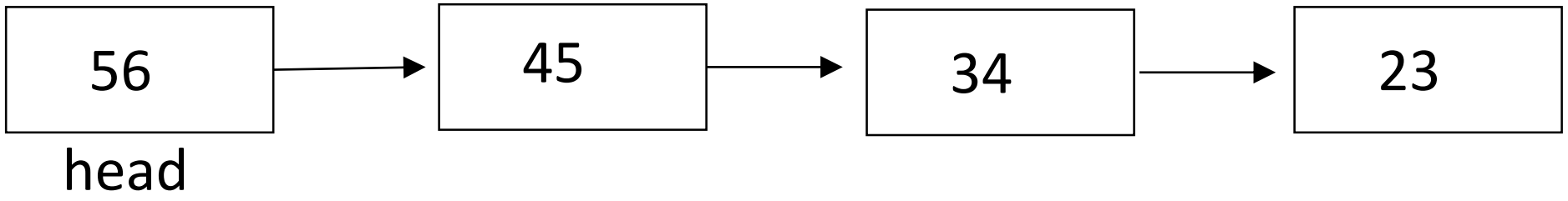
After we do the first statement `s.next = t;` we have the following picture:



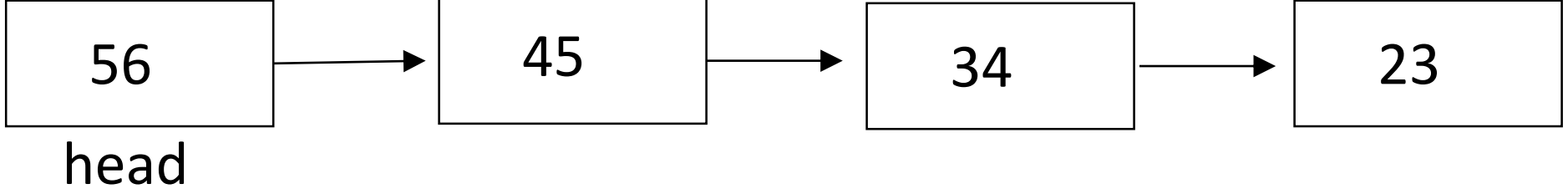
There is nothing pointing at the end of the list with values 34 and 23; we have lost part of our structure. Using 3 variables is easier.

The moral here is simple: be careful how you code; draw pictures to guide your code. Don’t try to memorize this code; just work it out from a picture.

Now you do it. Starting with this structure:



write code that will remove 34 from the structure. Then write code that will insert value 51 between 56 and 45.



Here's how I would do this:

```
// Remove 34
        Node p = head.next;
        Node q = p.next.next;
        p.next = q;


// Add 51
        Node a = head;
        Node c = a.next;
        Node b = new Node(51);
        a.next = b;
        b.next = c;
```