Linked Lists

Linked Lists

A linked list is a data structure that lets us store information in a sequence. It is sort of like an array, but with several important differences:

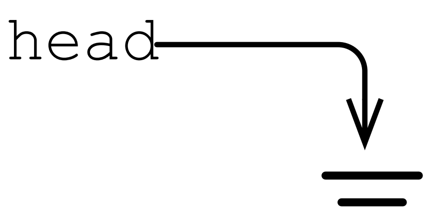
* The elements of the linked list can be spread out all over memory. They don't have to be right next to each other like the cells of an array.

* The elements of a linked lists are compound structures: they have *data* and *links*.

A linked list node has two members:
a **value** that holds user data, and
a **link** that refers to the next node.

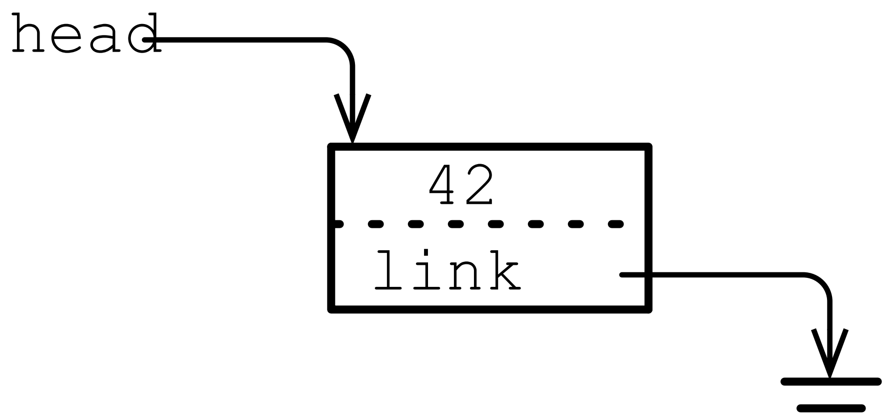
This is an individual
linked list node.

You need a pointer to the first node.
This variable is the same data type as
the 'link' member in a linked list node.
Call it 'head'. Initially 'head' points
to nothing, or NULL. Drawn as:



This means there's a pointer to null.
It signifies that there is no more data.

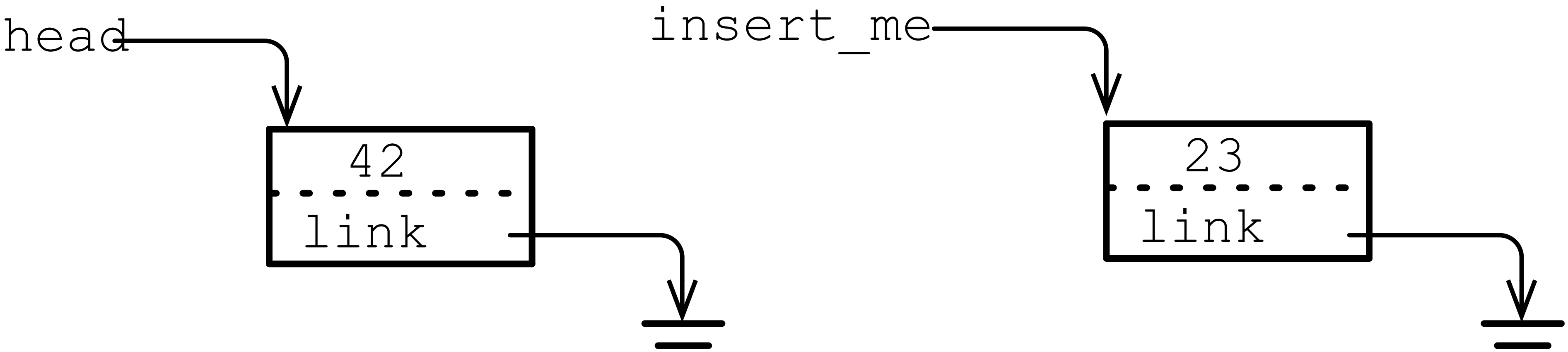
An empty list is not very interesting, but every list must start somewhere. Let's make a list that has a single node with the value '42'.
It looks like this:



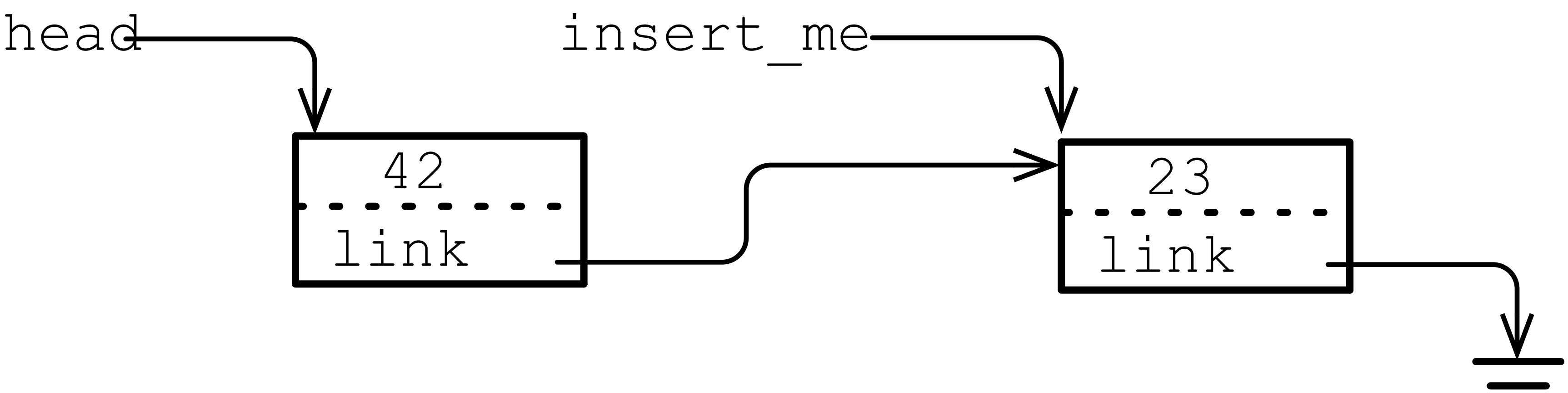
When you make a new node, you should specify the value for it to contain.

New nodes also have a null link. This means they are the end of the list.

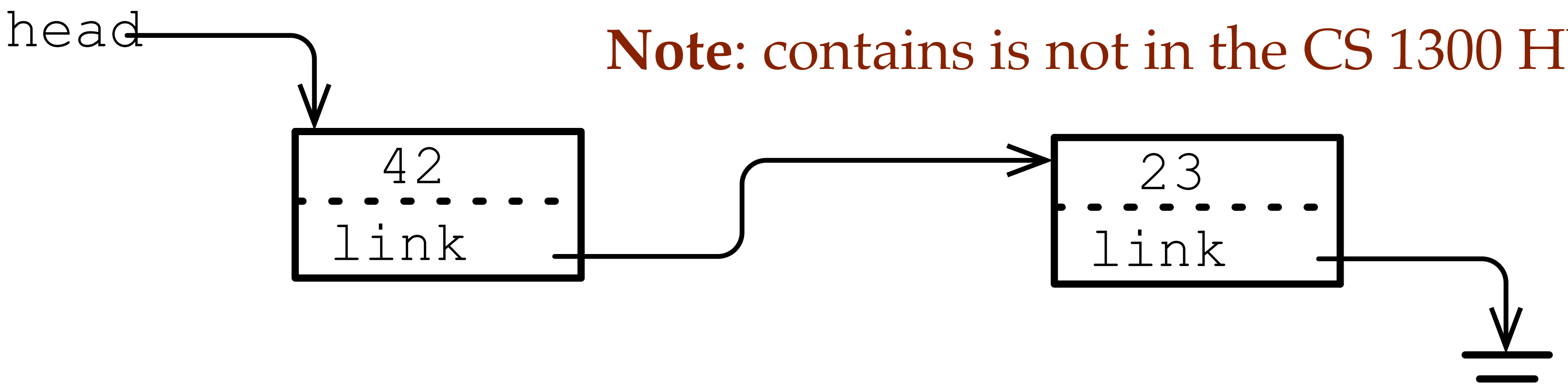
What if we want to **append** the number 23? We have to create a new node (call it 'insert_me').



But our list pointed to by 'head' still only has one element. We have to adjust the first node to point to our new 'insert_me' node:



Note: the 'insert_me' variable is no longer needed.

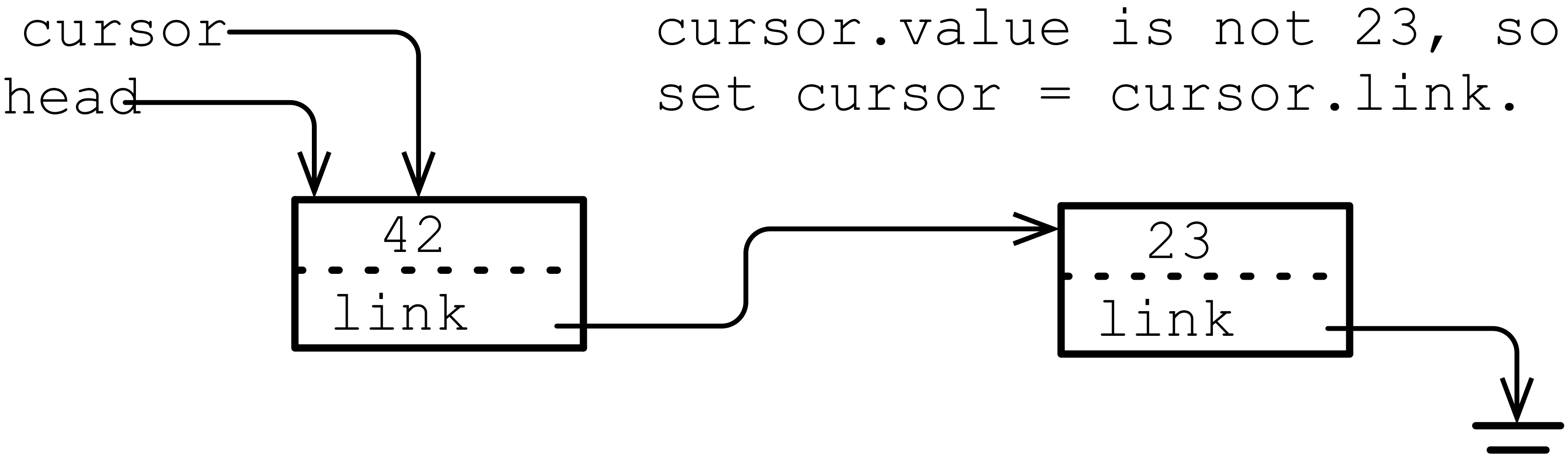


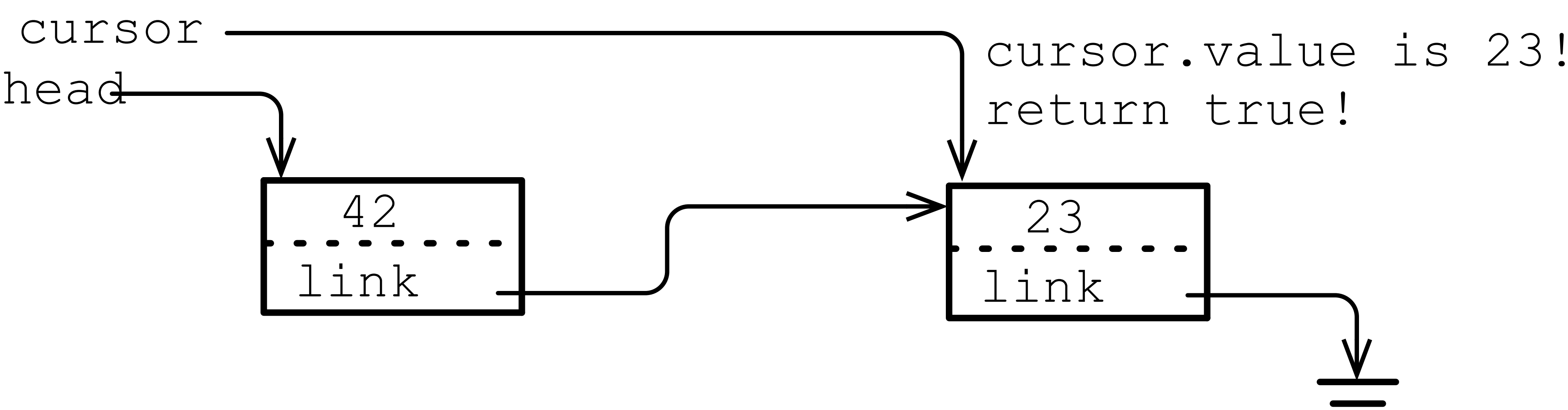
Note: contains is not in the CS 1300 HW

We can query the list pointed to by 'head' to see if it **contains** a certain datum. To do this, start at 'head', and examine a node value. If it is a match, declare success and return true. Otherwise, move to the next node if there is one. If this process reaches the end (indicated by a null link), we know the list does not have it, so the result is false.

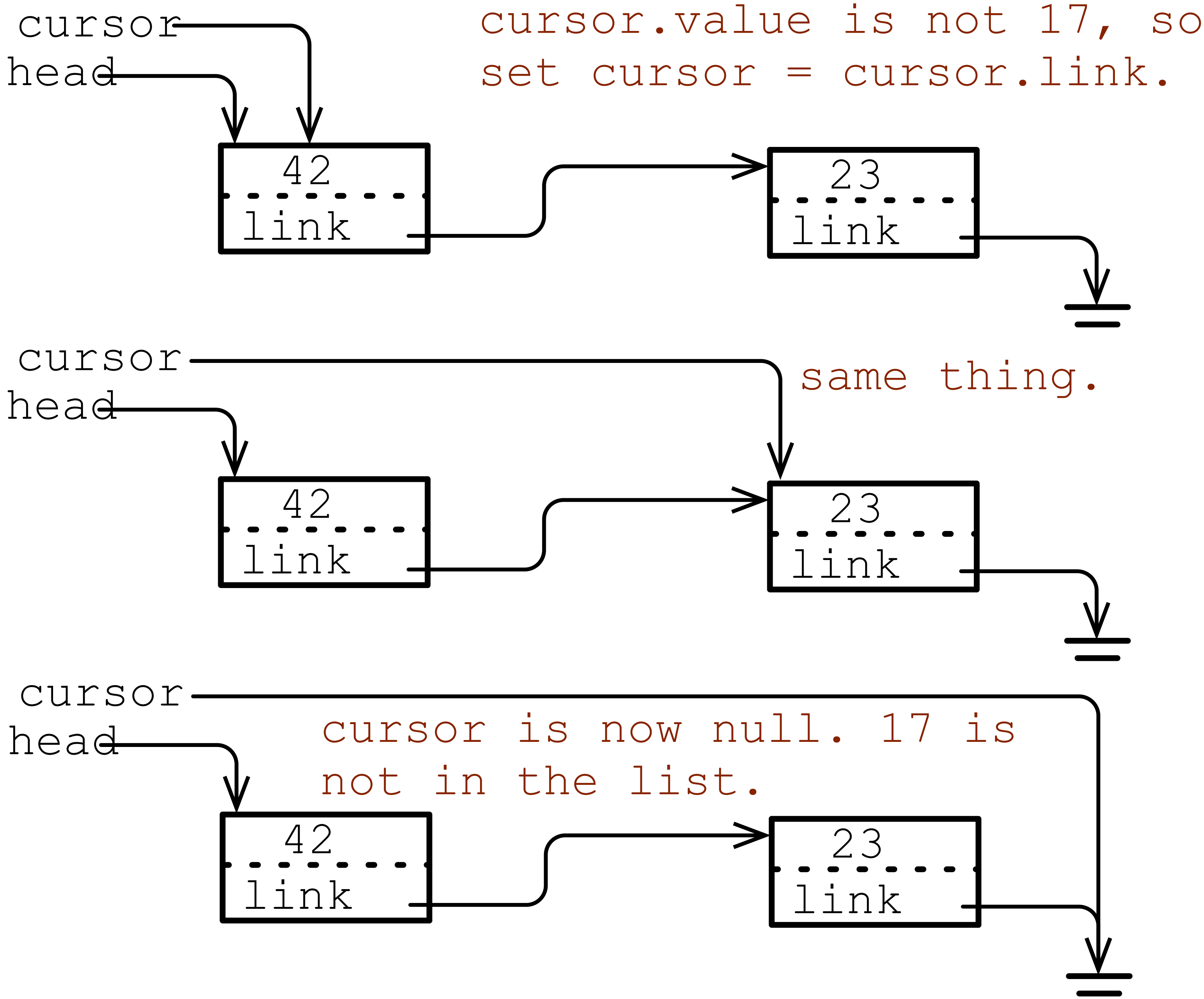
This is an example of a really important concept you'll need throughout data structures. This is called *iterating*.

To perform this scan, maintain a 'cursor' variable that points to the node we are currently inspecting. Say we are looking for '23':



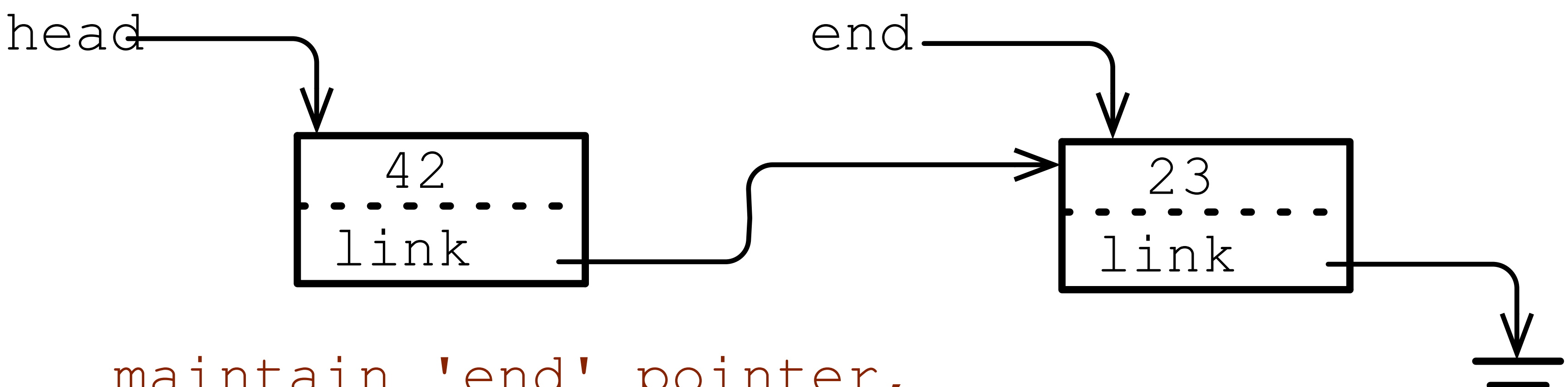


Now say we are looking for '17'.

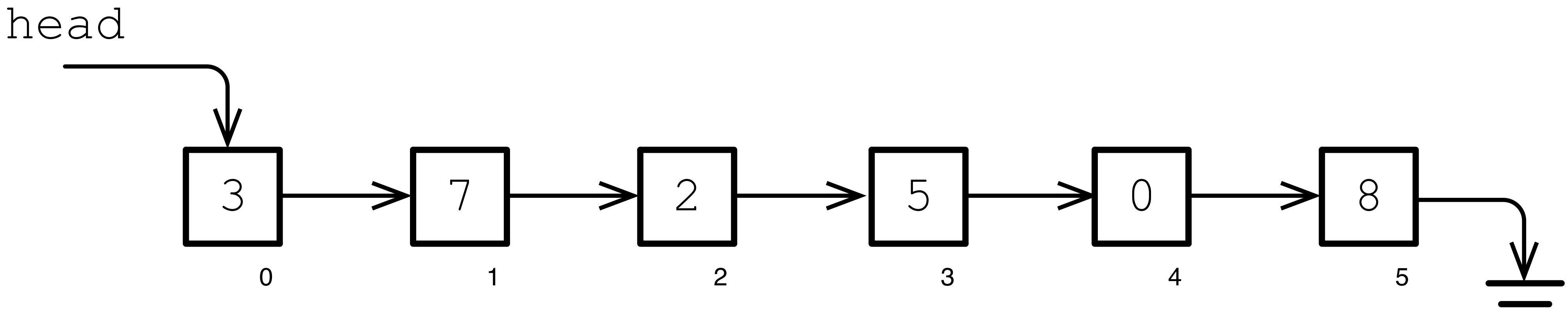


Remember when we **append**ed to the list? We cheated!
In general you can't tell how long a list is,
so you will need to use a cursor for appending.

The trick is to have a function that scans for
the end of the list, *or* to always keep a variable
that refers to the final node. Either way, you
will need a way to refer to the end.



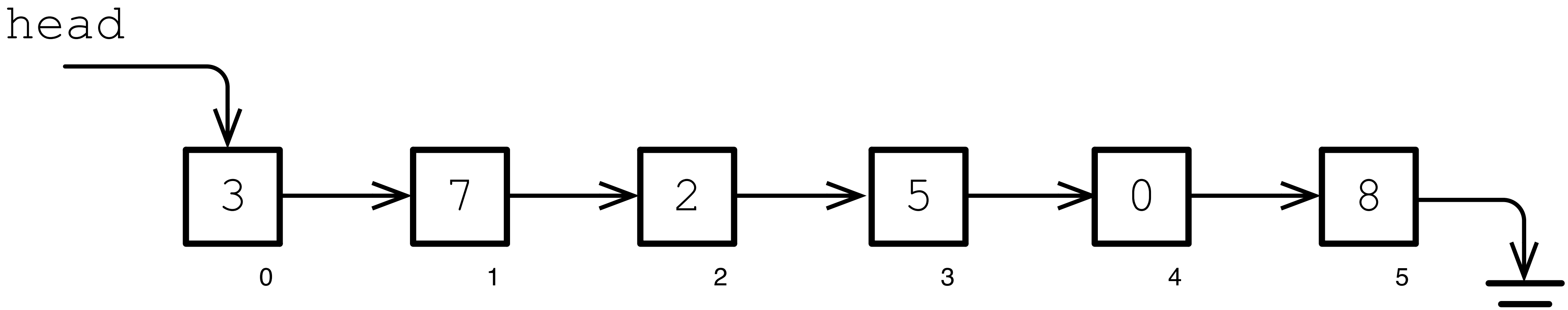
maintain 'end' pointer,
or,
write a 'scan_to_end' function.



We can **retrieve** a value by an offset. Say we want to get the number at index 3. Remember that we count starting from zero. So, the value at index number 3 is 5.

This is essentially a scan where we start at 'head' and follow the pointers until we've gone far enough.

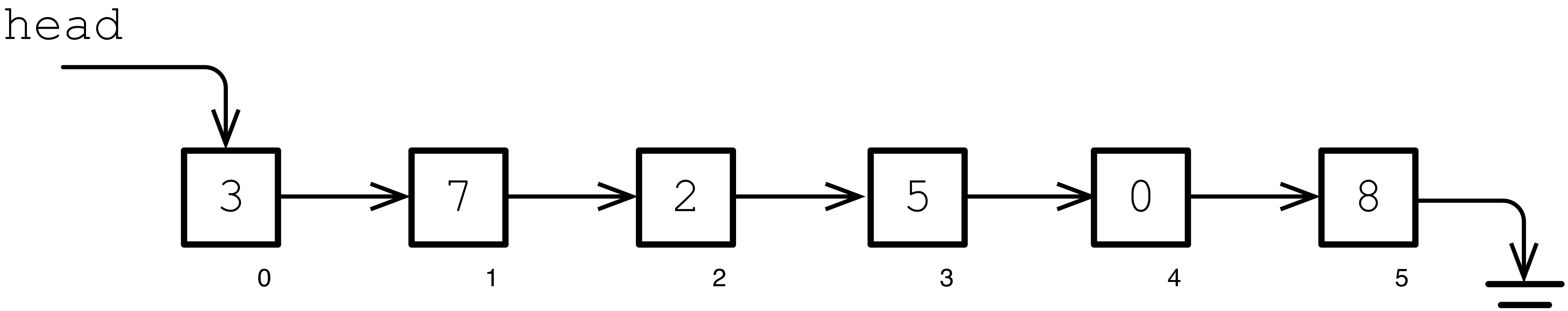
Keep in mind that the user might ask for an index that doesn't exist! Your program should not crash if this happens.



We can query the **size** of the linked list in a similar manner. Instead of looking for a particular index, we count nodes as we look for the null link that indicates the end of the list.

Remember: if the last index is 5, it means there are 6 nodes. If the last index is N, then there are N+1 nodes.

Drawing a picture helps make this sink in.

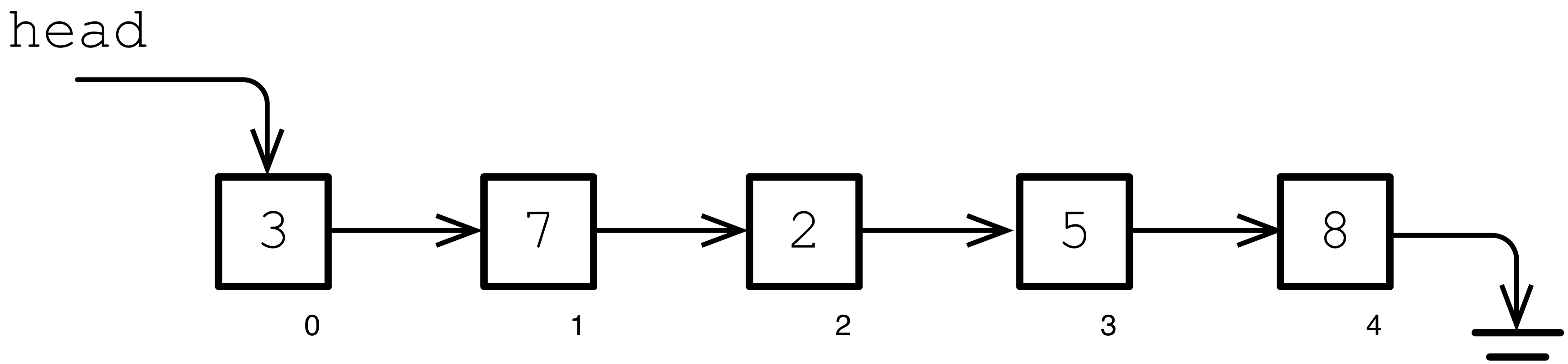


Another common operation is to **print** out the contents of the list. This is extremely helpful for debugging. It is similar to the size function in that we scan from the beginning to the end. But instead of counting nodes, we display each node's value.

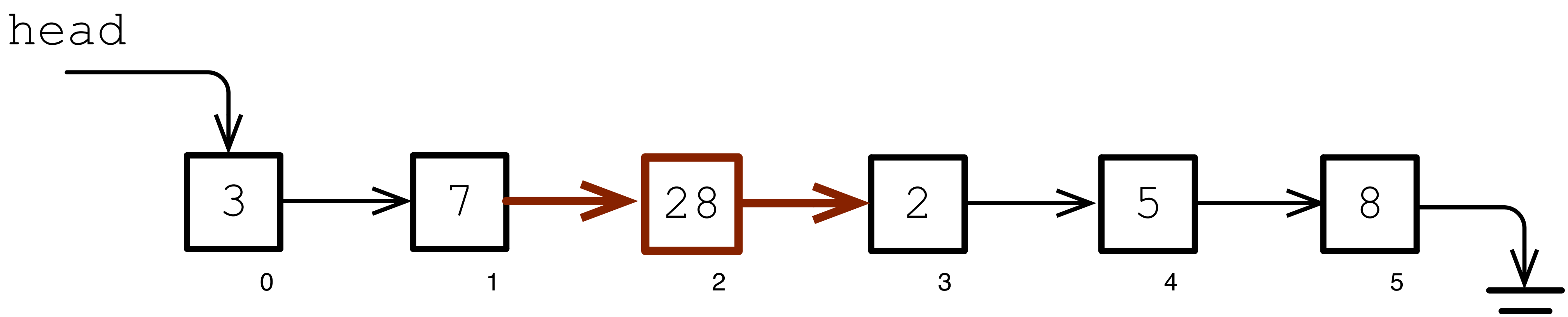
The list pointed to by 'head' would print out:

3 7 2 5 0 8

Say we wanted to **insert** a value a particular index. Say we want to put the number 28 at index 2. This is what it looks like at the start:

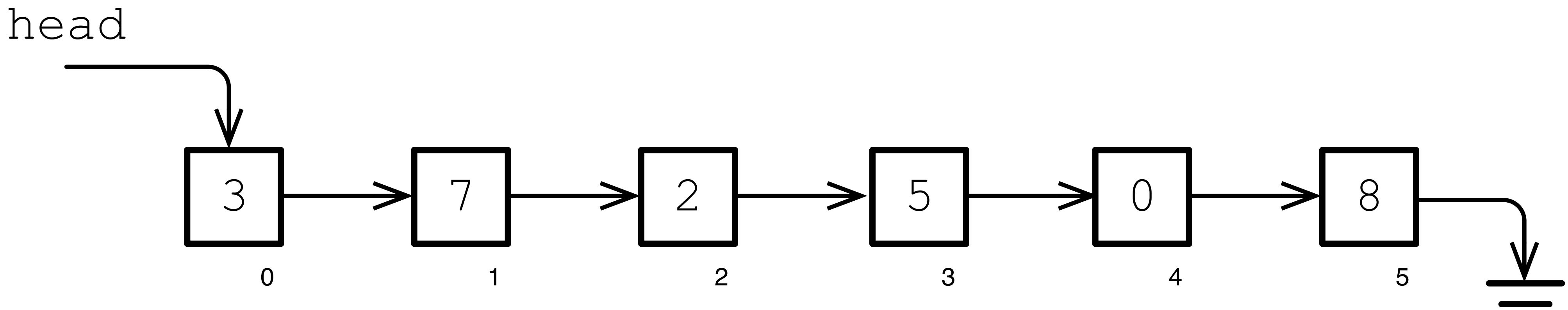


This is what we want it to look like at the end:

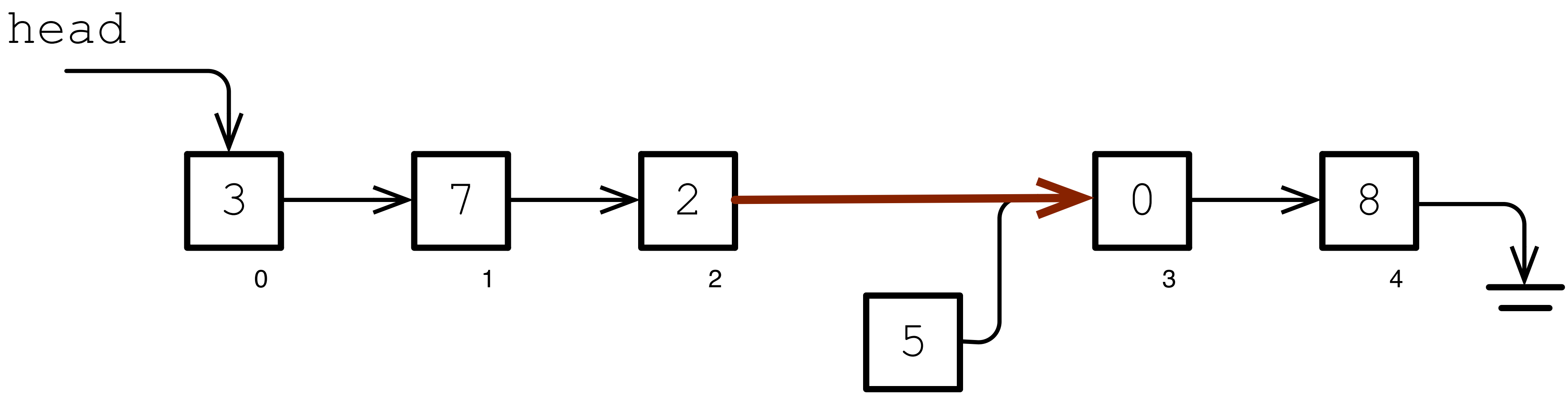


I've indicated the things that change: the link right before the new node, and the new node's link.

Say we wanted to **remove** a value a particular index. Say we want to remove the number 5 at index 3. This is what it looks like at the start:



This is what we want it to look like at the end:



Changed part indicated. Simply updated the preceding link to point beyond it. Note the '5' node persists.

Summay of Linked Lists Operations

```
init_node      create blank initialized node
append_data    append an int (defer to append)
append         append a node
insert_data    insert an int (defer to insert)
insert         insert a node
remove         remove node at index
size           gets size of list
contains       does list contain a value?
```

Note about node** head_ref

Most of these functions have a double-pointer. Don't be afraid. This is necessary. Here's most of the 'append' function to give you a sense of how to get started:

```
void append(node** head_ptr, node* new_node) {
  if (*head_ptr == NULL) {
    *head_ptr = new_node;
  } else {
    node* cursor = *head_ptr; // deref head_ptr
    // The rest was removed. All you do now
    // is scan for the last node starting at
    // cursor. The last node's link is NULL.
    // Have it point at new_node.
  }
}
```