CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 3 Jan 18, 2013

Linked Lists

Lecture Goals

1. Announcements
2. Document Code = Happiness
3. Linked Lists

Upcoming Homework Assignment

HW #1 **Due: Friday, Jan 25**

Linked Lists (pointers)

Implement a Linked List data structure along with common operations that manipulate and query the structure, including (not limited to): appending, inserting, removing, searching, sizing, and testing for presence of values.

Details in the homework folder on GitHub.

Documenting Code

Documenting code is important not just for others but for your own sanity.

Pre and *Post*-condition contracts for functions give you some basic constraints about how to write code. It also saves time later on when you're working with code you wrote last week/month/year.

Pre-conditions

Pre-condition means what you expect to be the case when a function is called. Are variables initialized? Must they have specific values? Or be within a range of values?

e.g.

```
float read_pin(int pin) {
  // pin must be one of the constants defined
  // in PIN_CONSTANTS.h
  ..
}
```

Post-conditions

Post condition contracts define what the function must do and not do: what does it return, what are the restrictions on what it can change, etc.

e.g.

```
float read_pin(int pin) {
  // returns the value of the robot servo in the range
  // -1.0 .. 1.0. No robot state must be changed,
  // not even temporarily!
  ..
}
```

Pointers

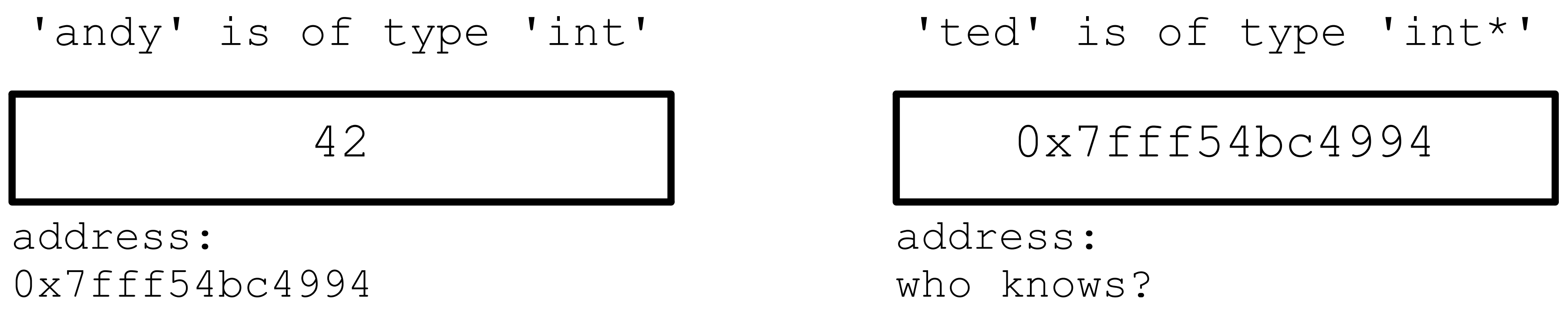
```
int main() {
  int andy = 42;
  cout << "andy: " << andy << endl;
  cout << "&andy: " << &andy << endl;
  int* ted = &andy;
  cout << "ted: " << ted << endl;
  cout << "*ted: " << *ted << endl;
}
```

This prints out:

```
andy: 42
&andy: 0x7fff54bc4994
ted: 0x7fff54bc4994
*ted: 42
```

Pointers: types and addresses

```
int andy = 42;
int* ted = &andy;
```



'Reference Operator', aka
'Address Of Operator': $\longrightarrow$ **&**

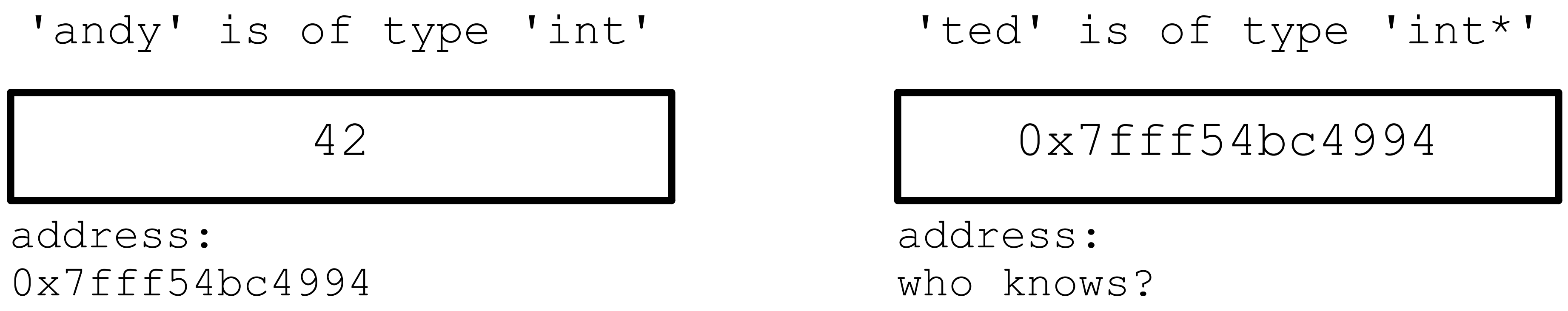
'Dereference Operator', aka
'Value Pointed By': $\longrightarrow$ *****

If we print things out…

```
andy: 42
&andy: 0x7fff54bc4994
ted: 0x7fff54bc4994
*ted: 42
```

Pointers: Operators and Values

```
int andy = 42;
int* ted = &andy;
```



'Reference Operator', aka
'Address Of Operator': $\longrightarrow$ **&**

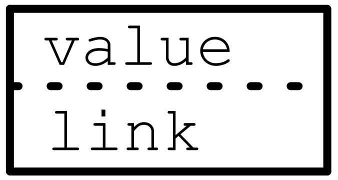
'Dereference Operator', aka
'Value Pointed By': $\longrightarrow$ *****

If we print things out…

```
andy: 42
&andy: 0x7fff54bc4994
ted: 0x7fff54bc4994
*ted: 42
```

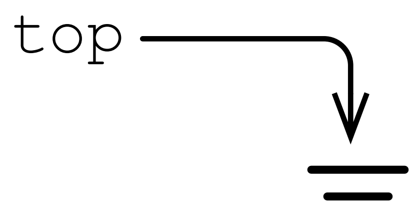
Linked List Node Data Structure

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.

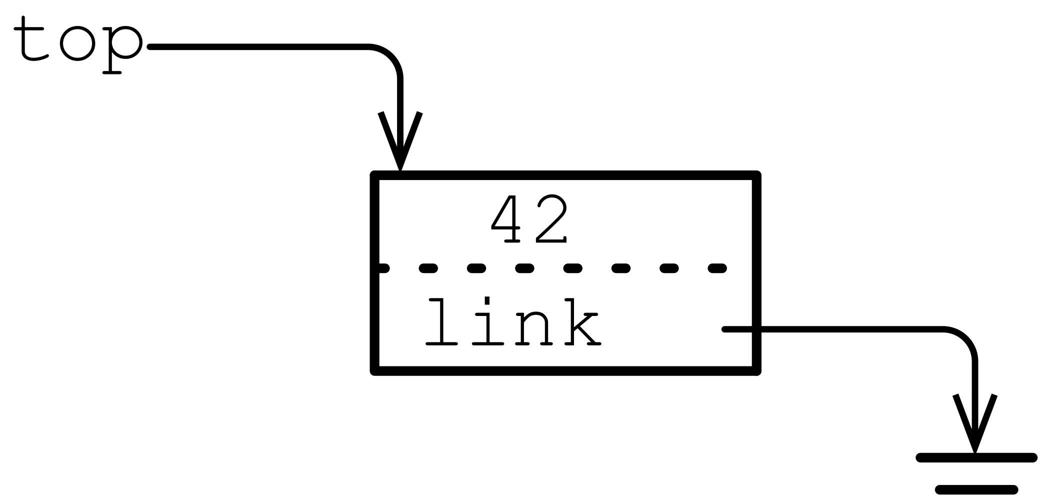
You need a pointer to the first node.
This variable is the same data type as
the 'link' member in a linked list node.
Call it 'top'. Initially 'top' points
to nothing, or NULL. Drawn as:



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

List With One Element

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:

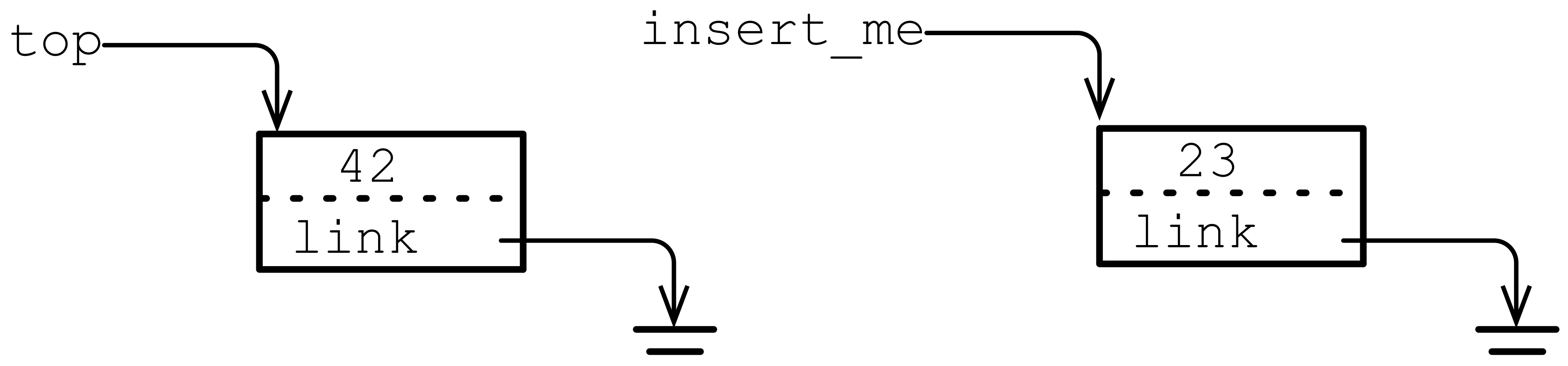


When you make a new node, you should specify the value for it to contain.

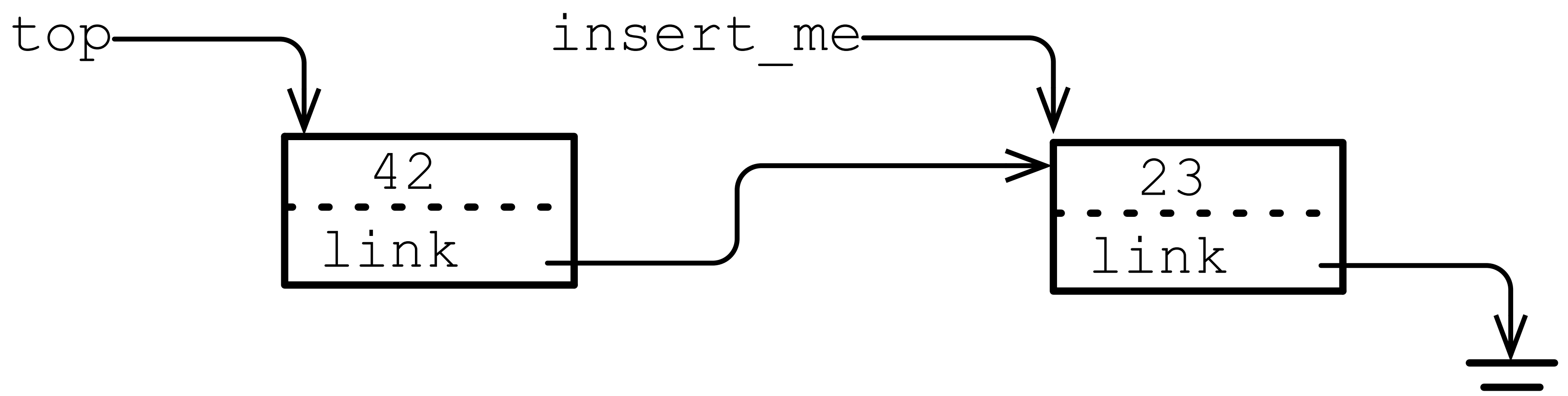
New nodes also have a null link. This means they are the end of the list.

Append

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

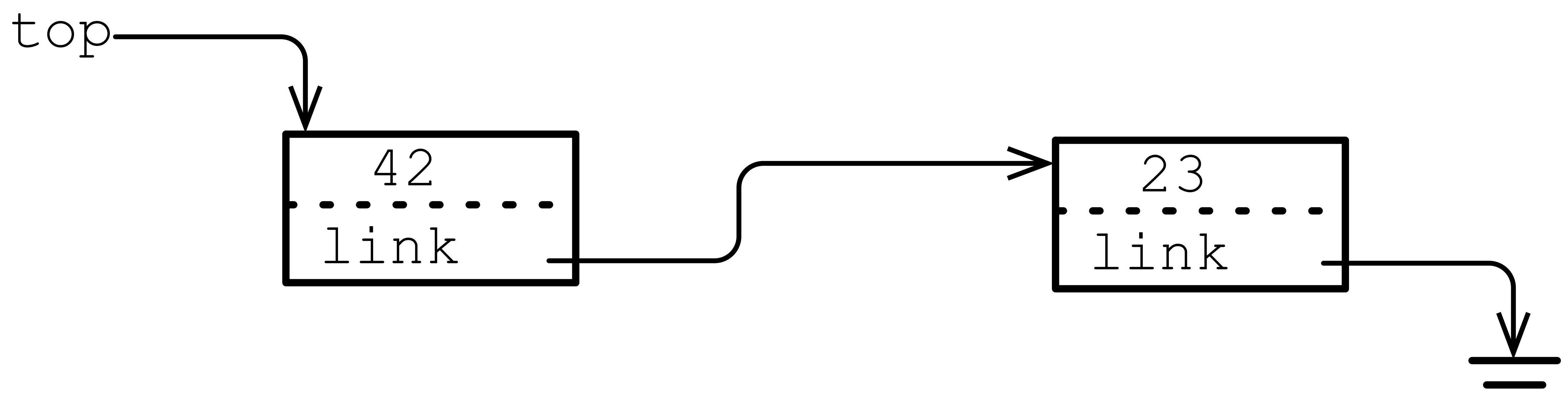


But our list pointed to by 'top' still only has one element. We have to adjust the first node to point to our new 'insert_me' node:



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

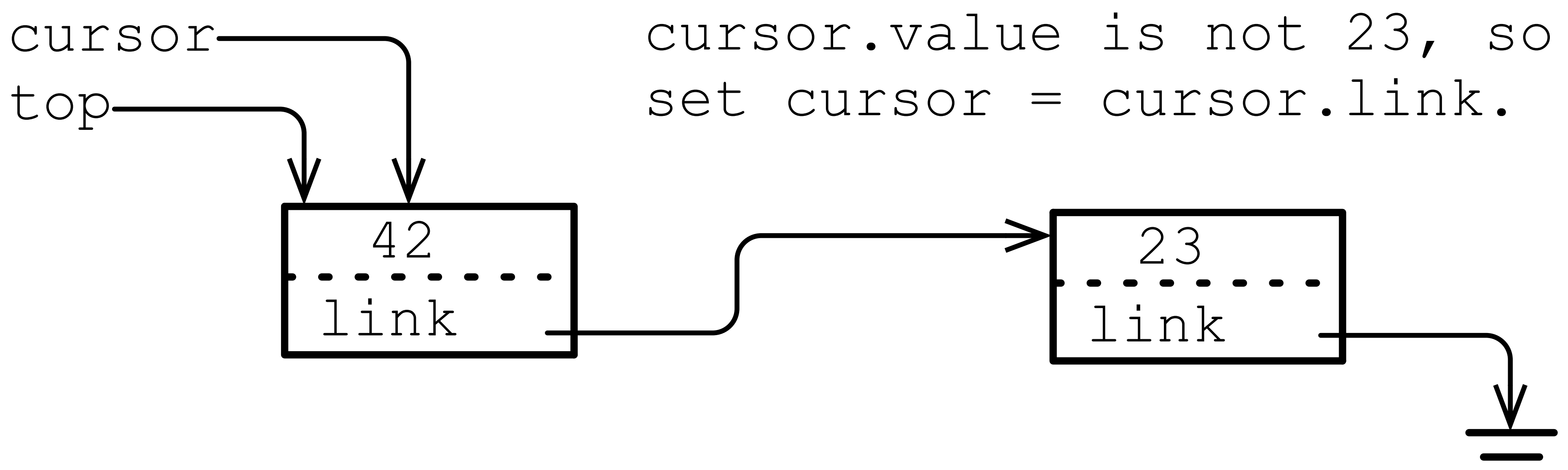
Contains Datum?



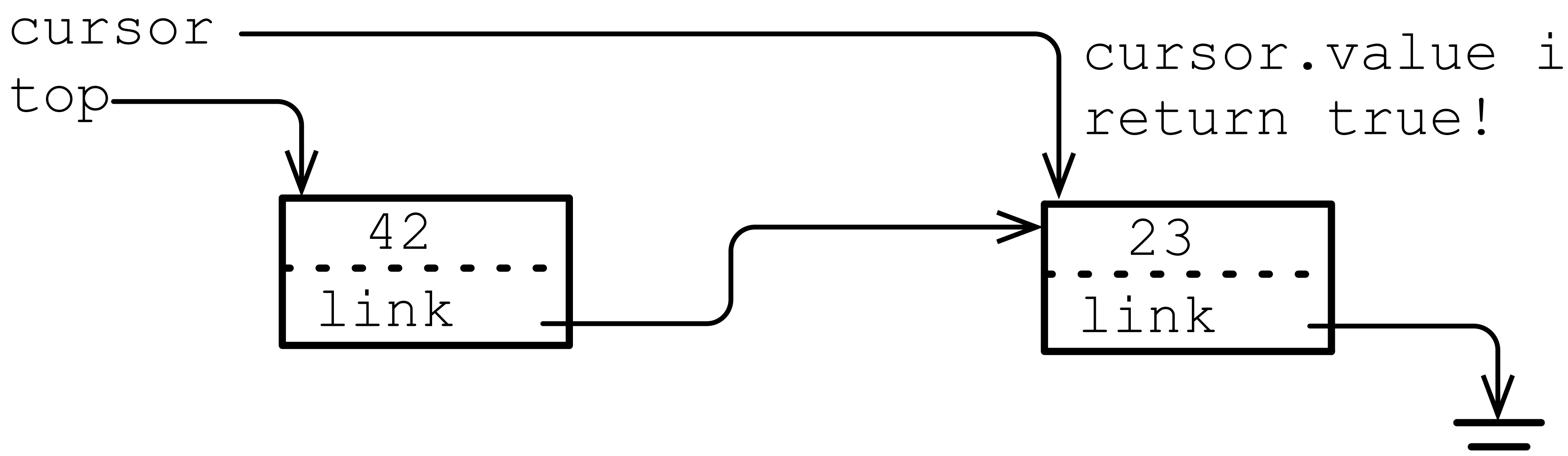
We can query the list pointed to by 'top' to see if it **contains** a certain datum. To do this, start at 'top', and examine a node value. If it is a match, declare success and return true. Otherwise, move to the next node if there is one. If this process reaches the end (indicated by a null link), we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

Using A Cursor

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



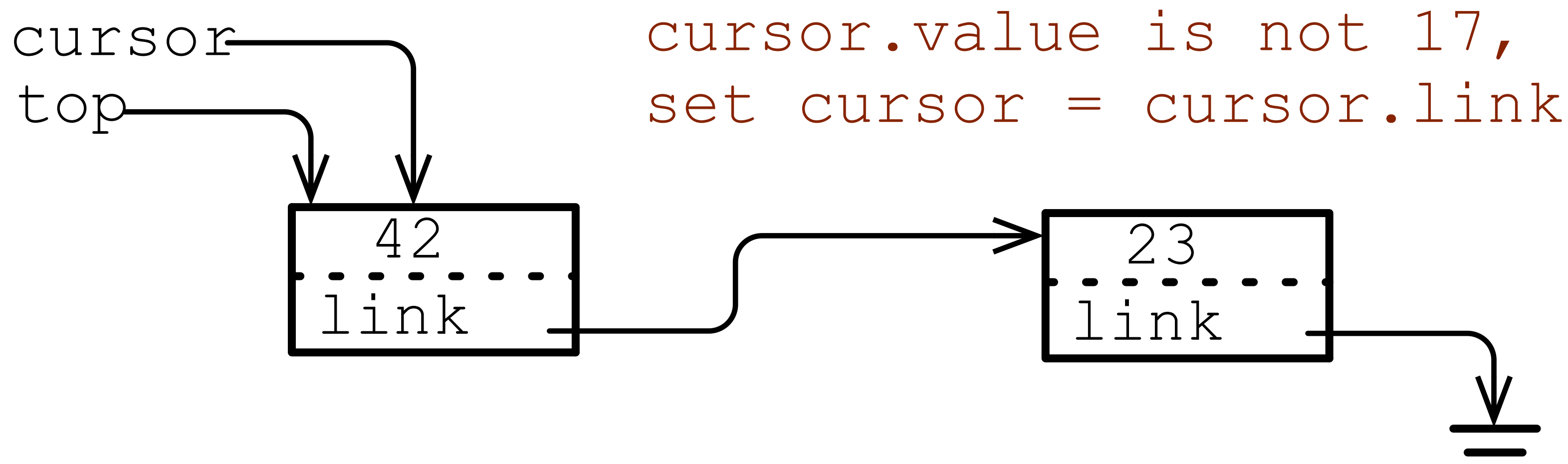
cursor.value is not 23, so set cursor = cursor.link.



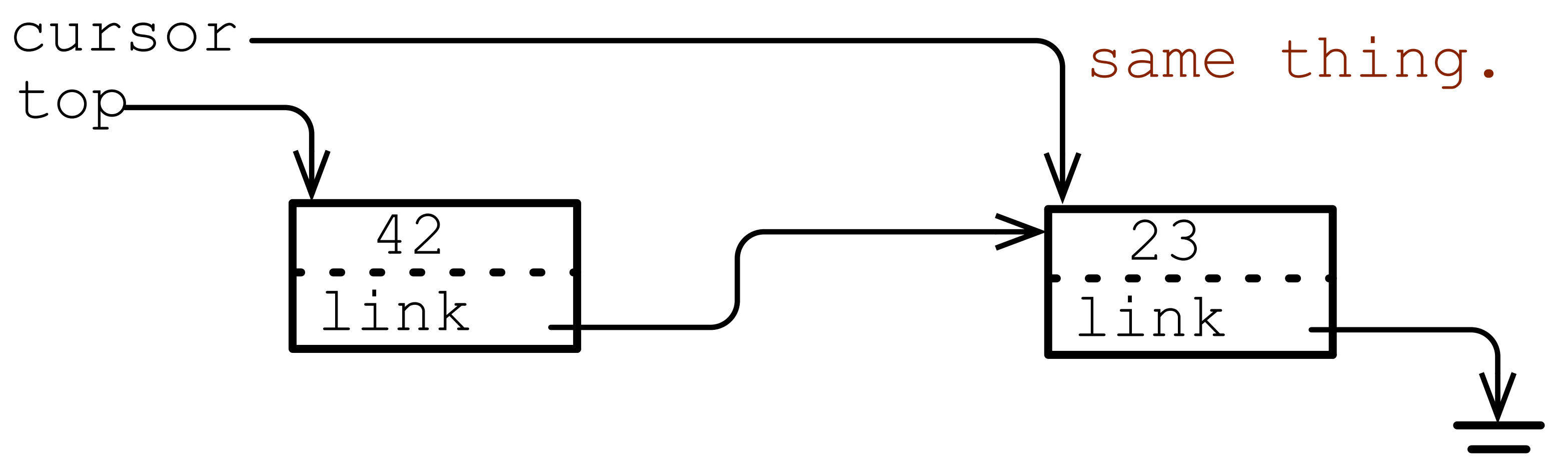
cursor.value is 23! return true!

Using A Cursor

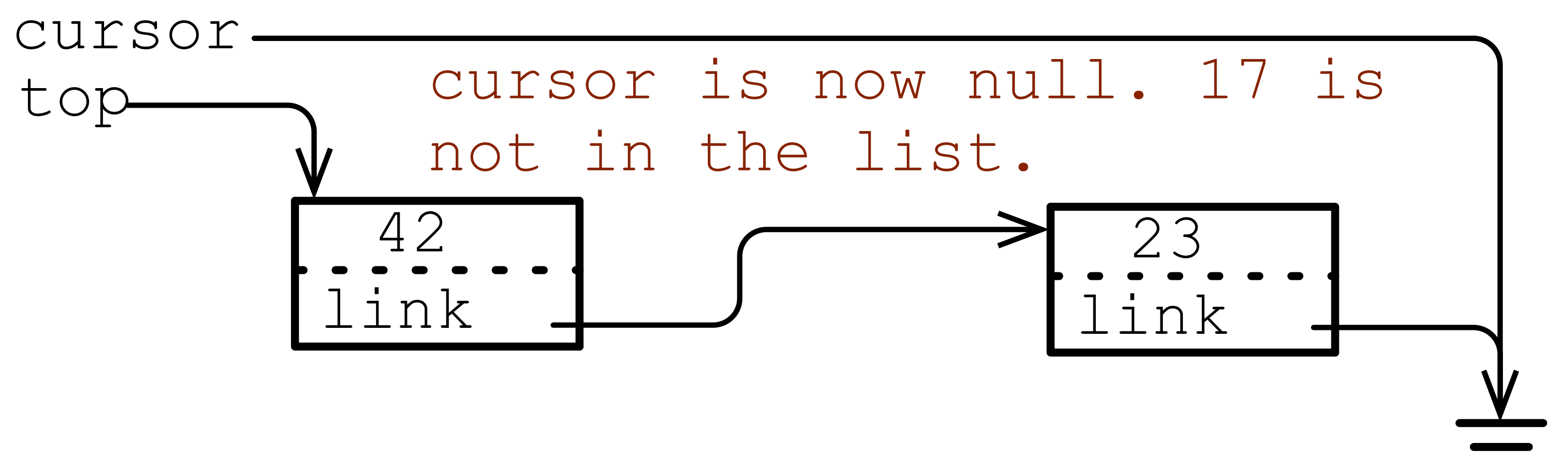
Now say we are looking for '17'.



cursor.value is not 17, so set cursor = cursor.link.



same thing.

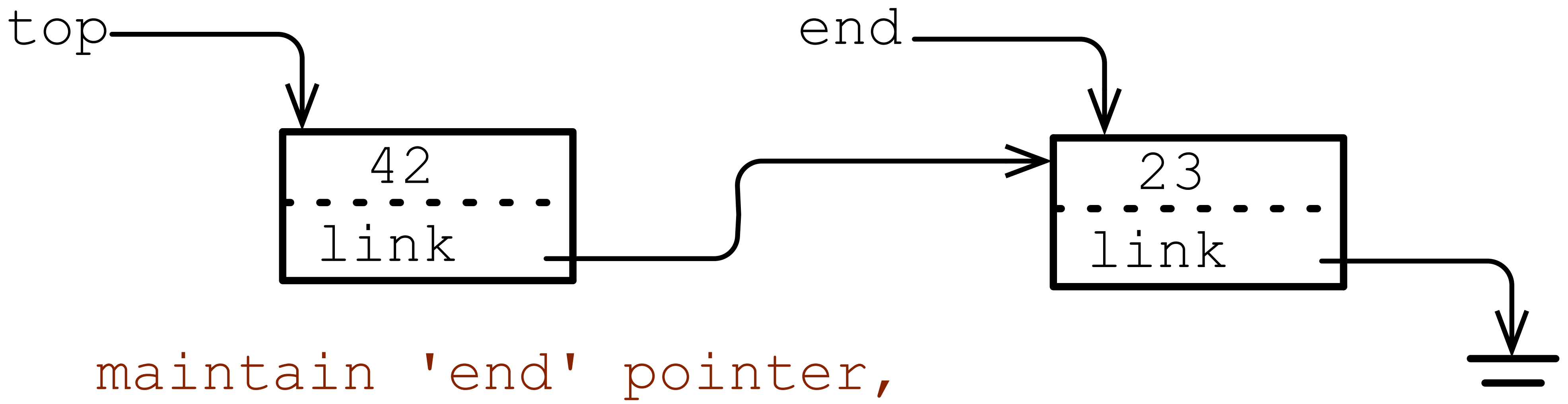


cursor is now null. 17 is not in the list.

Append, Revisited

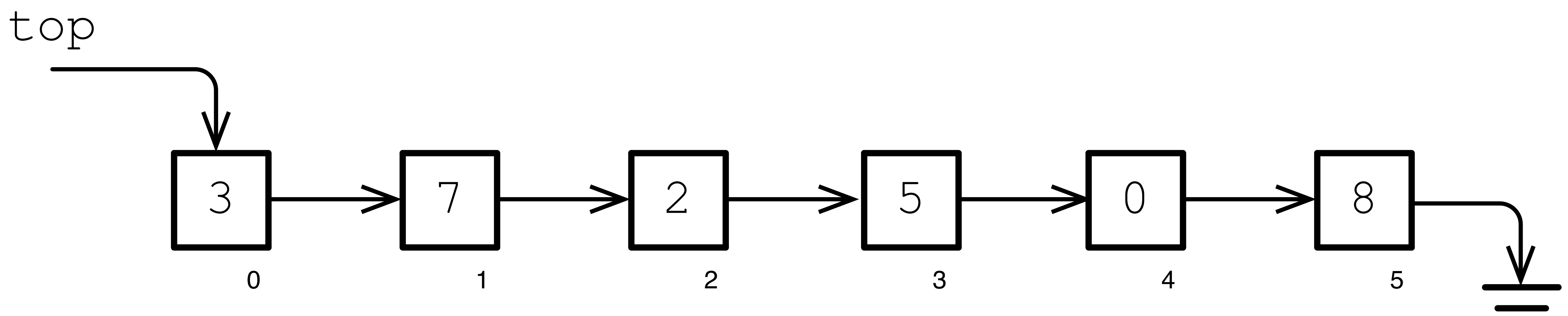
Remember when we **append**ed to the list? We cheated!
In general you can't tell how long a list is,
so you will need to use a cursor for appending.

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.

Indexed Access

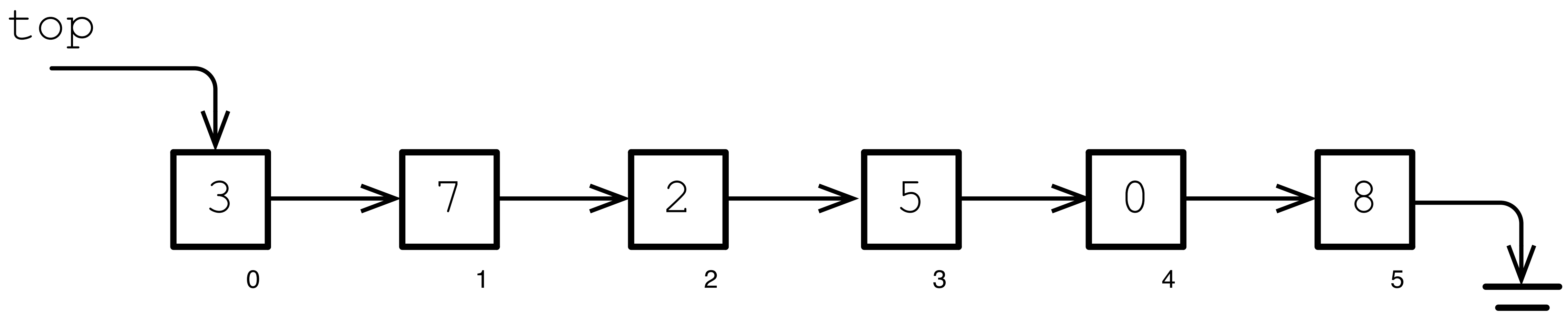


We can **retrieve** a value by an offset. Say we want to get the number at index 3. Remember that we count starting from zero. So, the value at index number 3 is 5.

This is essentially a scan where we start at 'top' 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.

Size Of List

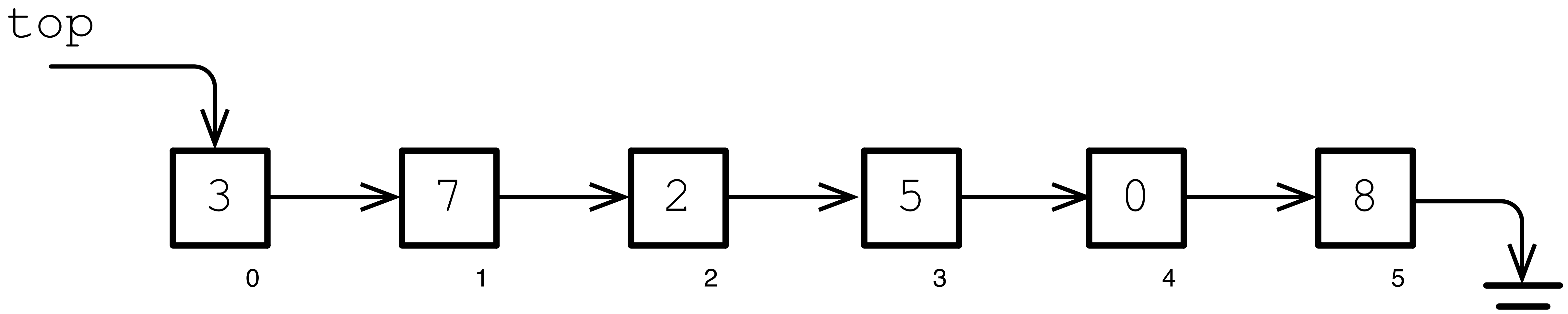


We can query the **size** of the linked list in a similar manner. Instead of looking for a particular index, we count nodes as we look for the null link that indicates the end of the list.

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.

Print List Contents



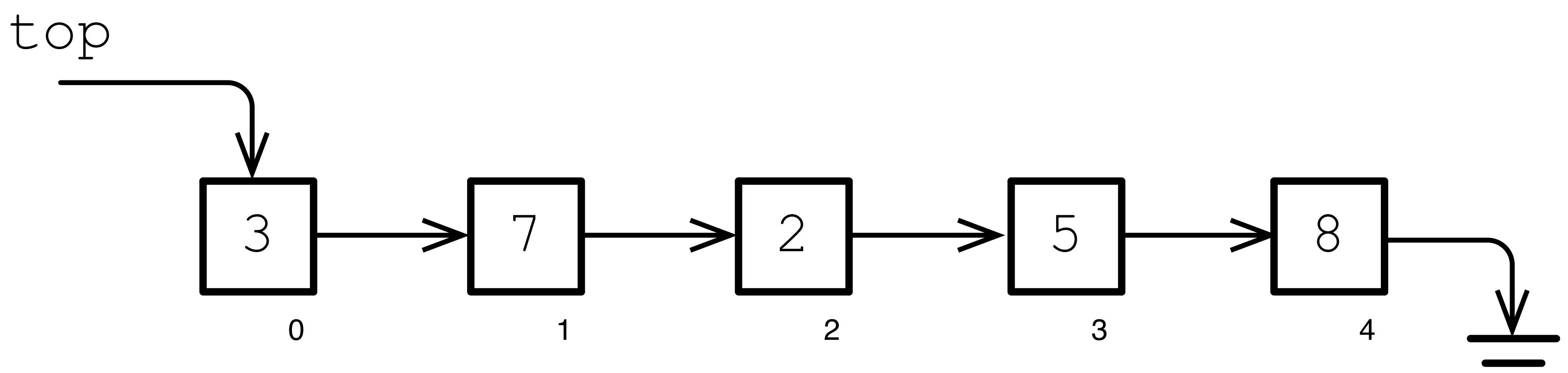
Another common operation is to **print** out the contents of the list. This is extremely helpful for debugging. It is similar to the size function in that we scan from the beginning to the end. But instead of counting nodes, we display each node's value.

The list pointed to by 'top' would print out as:

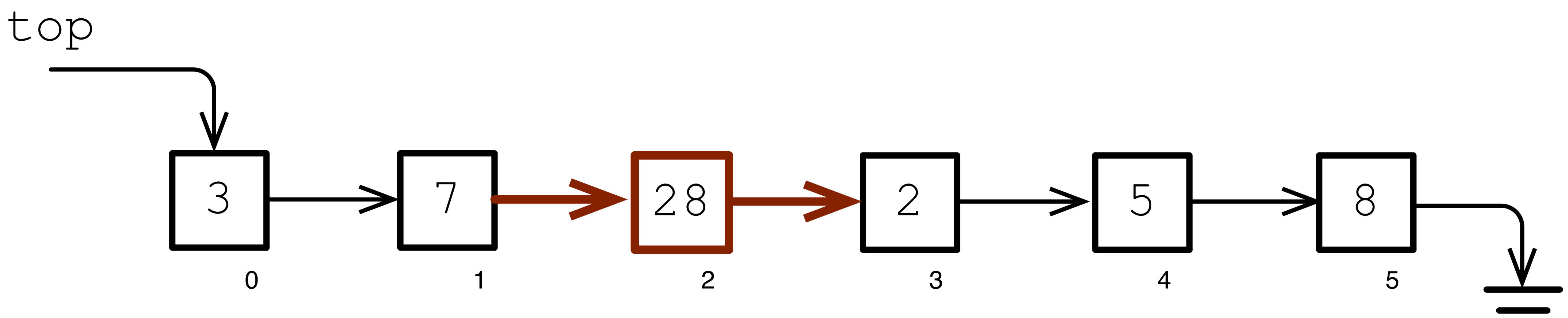
```
3 7 2 5 0 8
```

Insert At Index

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:



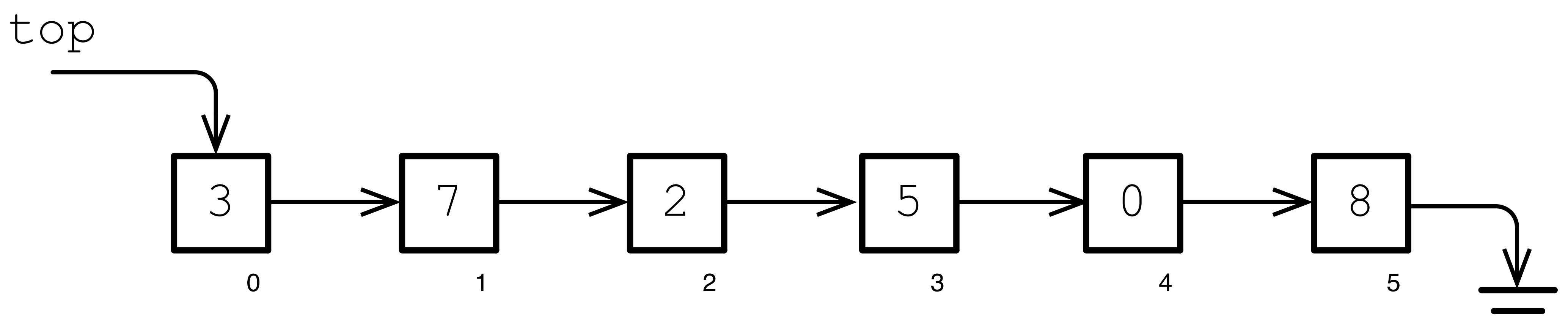
This is what we want it to look like at the end:



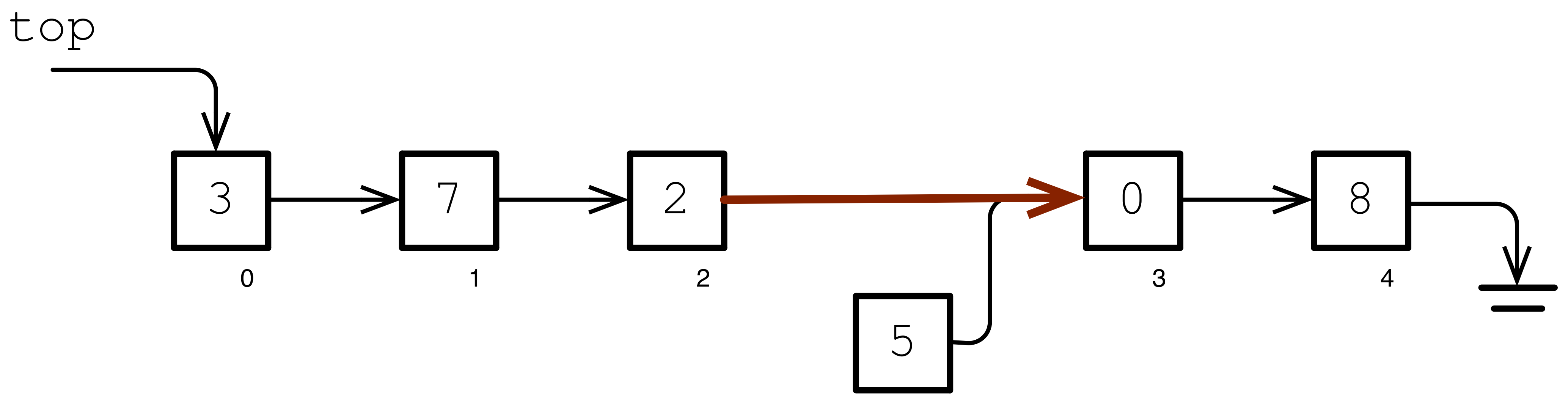
I've indicated the things that change: the link right before the new node, and the new node's link.

Remove At Index

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.