Lecture 17

CSE 110

3 August 1992

1 Doubly-Linked Lists

We've seen how to use a linked list to keep track of an arbitrarily large amount of information, growing the list as we need to. We can search through the list and look for something, or print out all the data in the list.

As a further illustration of dynamic memory techniques, let's suppose that we need to be able to walk through a list backwards as well as forwards. That's the one thing we can't do well with the linked lists we've seen: Once we get to a node, we don't remember how we got there.

The solution is simple, of course: Instead of one pointer per node, we have two: One that points forwards and one that points back.

In class we developed code to read an arbitrarily long list of integers from the user, terminated by a `0`, print the numbers out in the order they were entered, and then print the numbers in reverse order. We needed to traverse the list in both directions.

1.1 A Program to Print a List of Numbers Forward and Backward

The code we developed in class appears at the end of these notes; these are some comments on various details of the code.

We had a big argument in class about one of these details: The problem of how to start the list off. There were two main camps: One camp said that we should start the list off with a 'bogus node', so that the functions that operated on lists would always have something to work on. The other camp held that the pointers to the head and tail nodes of the list should start out `NULL`, and that

the functions that operate on lists should check for that and behave slightly differently, if necessary, if they saw that they were operating on an empty list.

Neither method has advantages.

Disadvantages of the 'bogus node' solution include:[1]

- The head and tail of the list are now drastically different, because the head has a bogus node, but the tail doesn't. This means that the functions to walk through the list from head to tail will look substantially different, whereas they probably should look almost the same since they are doing almost the same thing.
- A list with no nodes is a perfectly reasonable entity. To avoid it for no reason smacks of superstition akin to the fear of the numeral zero prevalent in Europe in the Twelfth Century.

Disadvantages of the '`NULL` pointer' solution include:

- You might have to complicate the functions that manage lists to include special cases for empty lists.
- This complication will slow down the list-managing functions.

1.2 Notes on the Code

5 This is the definition of the bogus value that we're going to use to mark out bogus node. It's also the value that the user uses to indicate end-of-input. It's good to use the same value for both things: because the value marks the end-of-input, it can never be one of the data items we have to remember, and so therefore it's an ideal bogus node marker.

7–11 It might have been better to put the two pointers in this structure together, to emphasize the way they were related, but we put things in this order so that they would look good on the blackboard. Now that out program works, we can forget the blackboard, and we should probably reorder the structure members.

[1] We said in class that another disadvantage is that the bogus node must be distinguished in some way. In this program, we distinguish it by storing `BOGUS_VALUE` into its `number` member, but in some circumstances a ready-made bogus value may not be available. However, if no appropriate bogus value had been available, we could have distinguished the bogus node because it would be first in the list and therefore it would have a `NULL` pointer in its member for pointing to the previous node in the list.

13 We wrote the prototype for `getnum` last week, as soon as we knew what we wanted the function to do, before we wrote the function itself. If we'd been paying more attention, we could have used the prototype today to help remind us of how to write `getnum`. It would have been even better to hang a comment near the prototype.

24 `head` and `tail` are pointers to the first and last nodes in the list, respectively. We probably ought to add a comment to that effect.

26–29 These lines set up the bogus node at the head of the list. Note that we forgot to check for an error (`NULL`) return from `malloc`, and if `malloc` fails here we'll go and set the members of a nonexistent structure. Therefore this program has an error.

33 `store_input` is the function that appends a new node and a new datum to the end of the list. It returns a pointer to the new last node, so that `main` can keep track of where the last node is at all times.

It might have been more elegant to have `store_input` return `void`, and to pass in a pointer to `tail` itself instead of a copy of `tail`'s value. Then `store_input` could have changed `tail`'s value without intervention from `main`.

44–56 `store_input`, as promised, is simple; we have to manufacture a new node, fix its three members, and link it into the list by making the old `tail` node's `forward` member point to the new node. This last step, `tail->forward = newnode`, is the only one that would fail if the `tail` that were passed in were `NULL`; to make `store_input` work when `tail` is `NULL`, you need just skip this single statement. (This is not supposed to be obvious; please work through the code and see what happens.) So to convert `store_input` to operate on an empty list is a matter of adding one line of code: `if (tail != NULL)` just before `tail->forward = newnode`.

48 We forgot to check `malloc` here for an error return. This can result in disaster. A full solution would be to have `store_node` return `NULL` on failure, and have `main` check for a `NULL` return from `store_node` and abort if there was one.

58–66 `print_forward` uses the `NULL`-termination method to decide when to stop; it traverses the list, following the `forward` members, and stops after the `forward` member of the node it just processed was `NULL`. I think it's worth pointing out that this code works perfectly if the `head` that is passed in is a `NULL` pointer.

60 The line `head = head->forward` is there to skip past the bogus node at the head of the list before we start printing numbers.

70 If no appropriate bogus value had been available to distinguish the bogus node, we could have written `while(tail->back != NULL)` instead, so the possible lack of a bogus value is not a serious deficiency in the method we've used.

79 This is a quick hack on `getnum`; we should have added error checking and soforth. In particular, `getnum` should return `0` if it sees `EOF`, to signal end-of-input to `main`. As it is, our program goes into an infinite loop on `EOF`.

We stuck in this minimal `getnum` so we could get the program working and see if it had serious errors right away. This is perfectly legitimate. You should aim to get the program into a minimally working state as soon as possible, because that allows you to concentrate on major design decisions while postponing little details, such as `getnum`'s error handling, as long as possible.

A function like `getnum` that stands in for a better-developed version of the same function is called a *stub*.

2 Recursion

This weekend, I made a list of topics I thought were important enough to cover, but which we somehow missed along the way. Most of the things on the list are miscellaneous trivia. Recursion, however, is powerful stuff, and you shouldn't get out of a programming class without seeing how it works.

Recursion is a programming methodology. The idea is that when we describe how to perform a certain task (which is what programming is), it is sometimes most natural to describe how to reduce the task to a simpler task of the same type. Then you can reduce the simpler task to a still simpler task, and soforth, until finally the task is so simple it's trivial. Such methods are called *divide-and-conquer* methods or *recursive* methods.

2.1 The Factorial Function

The factorial function is a function from elementary mathematics. The function takes n nonnegative integer argument and yields a positive integer result. We write the factorial of n this way: $n!$.

The definition of the factorial function is:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

except when $n = 0$: $0! = 1$.

This definition yields a natural implementation in C:

```
long int fact(unsigned int n)
{
  long int total = 1;

  while (n>1)
    total *= n--;

  return total;
}
```

2.2 The Factorial Function

There's another way that mathematicians often define the factorial function:

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot (x-1)! & \text{otherwise} \end{cases}$$

This definition is equivalent to the earlier definition, but it does a different thing: Instead of defining the factorial of a number explicitly, it defines the factorial in terms of a simpler factorial. To compute the simpler factorial we will need to compute the factorial of a number smaller still, and soforth, until finally we discover that all we need is the factorial of 0, which is defined to be 1.

This formulation also leads to a natural implementation in C:

```
long int fact(unsigned int n)
{

  if (n == 0)
    return 1L;
  else
    return n * fact(n-1);
}
```

And in fact this works. (The `1L` is a constant with value 1 and type <`long int`>.)

The factorial function needs to call itself, but that's not a problem. When a function calls another, the computer saves a description of the place to return to, and all the local variables and other information it will need in order to return safely, and jumps into the called function. The saved information is sometimes called an *environment*, but more often it's called a *frame*. The function call semantics that C adopts in order to allow this give C functions a property called *reentrancy*; a function can be called, called again before it returns the first time, can return, and then return again, and the returns will happen properly and will not interfere with one another.[2]

When the `fact` function calls itself, the same thing happens. All of `fact`'s local variables and state information are saved, and the computer calls `fact`, which means that a second invocation of `fact` is created, with its own local variables. The first `fact` is paused, and waits for the second `fact` to complete. If the second invocation of `fact` calls `fact` again, a second frame is created, with the second `fact`'s local variables saved in it; when the third `fact` returns, the second `fact` picks up where it left off.

There is no arbitrary limit on the number of such nested calls one can make. Each frame consumes a little memory, though, and so if you make to many calls without returning, you'll run out of memory and your program will fail. This rarely, if ever, happens in correct programs, because frames are not that big.

2.3 The Towers of Hanoi Problem

The factorial function was simple enough, but there was an obvious nonrecursive way to write it, and there's no reason to use recursion when an obvious iterative solution is available. In this section we'll see a problem that is difficult to solve iteratively, but trivial to solve recursively.

The puzzle is this: There are three vertical pegs, named $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. There are some number, say 7, of circular discs, each of a different size, and each with a hole in the middle so that they can be placed on the pegs. Initially, all the discs are on peg $\mathcal{A}$, with the largest at the bottom, the next largest on top of that, and so on, with the smallest disc at the top.

The object of the puzzle is to transfer the entire tower to peg $\mathcal{C}$, subject to the following constraints:

1. You may only move one disc at a time.
2. You may never place a larger disc on top of a smaller disc.

[2]Not all languages have reentrant functions. FORTRAN is the principal example. This is a serious deficiency because it means you can't write a recursive function in FORTRAN.

After some thought, we arrive at a solution: To transfer a tower of n discs from $\mathcal{A}$ to $\mathcal{C}$, we can break the problem into three parts:

1. Transfer the smaller tower of $n-1$ discs from $\mathcal{A}$ to $\mathcal{B}$.
2. Move the nth disc, the largest, from $\mathcal{A}$ to $\mathcal{C}$.
3. Transfer the smaller tower of $n-1$ discs from $\mathcal{B}$ to $\mathcal{C}$.

In steps 1 and 3, we can ignore the large disc; it doesn't complicate matters at all. If we are performing these three steps in order to solve the problem with 7 discs, then steps 1 and 3 are just like the corresponding problem with only 6 discs.

How do we move a tower of 6 discs from one peg to another? Just apply the method again: Move a smaller tower of the top 5 discs out of the way, move the 6th disc to the destination, and move the smaller tower on top of it.

How do we move a tower of 5 discs? Just...

Eventually we'll reduce the problem to the trivial case, where we're asking how to move towers of 0 discs. How do we move a tower of 0 discs? Do nothing![3]

This suggests a straightforward implementation in C, which we'll see tomorrow.

[3]If this bothers you, then take one fewer step: We'll eventually reduce the problem to one of moving towers of only 1 disc from one pin to another; that's clearly trivial.