Computer Science 50
Fall 2009
Scribe Notes

Week 5 Wednesday: October 7, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 5 Wednesday: October 7, 2009
Andrew Sellergren

1 Announcements (0:00–3:00)

- This is CS 50.
- Quiz 0 will be held next Wednesday, 10/14, during normal lecture time. Do not come to Sanders Theater, however, as you will be assigned to one of three test-taking locations, as detailed in the About Quiz 0 handout.
- No class on Monday, 10/12, and no problem set next week!
- Also on the Quizzes page are 5 quizzes from past years. We highly recommend that you use these as practice! There are, however, some differences in material that was covered, so if you encounter a topic which is completely over your head, know that it might be because we haven't gone over it at all. Consult the syllabus or a TF if you have questions as to what will be covered.
- Sections this week on Sunday, Monday, and Tuesday will be quiz review. This Sunday from 7 to 8 PM in Emerson Hall 108 will be an optional course-wide review. This will be filmed and placed online by Tuesday. This review will take place in lieu of the weekly walkthrough.
- Next Friday, 10/16 at 1:15 PM, will be the next Lunch with David et al. RSVP at cs50.net/rsvp.
- Finally, for a bit of geek humor, check out Star Wars in ASCII.

2 Singly Linked Lists (3:00–64:00)

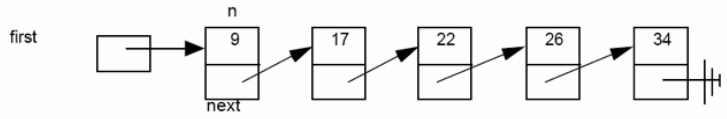
2.1 Introduction

- The syntax we introduced last week for handling pointers is really the last tricky syntax you'll be asked to learn. This week, however, we'll be introducing a new data structure called *singly linked lists*. Singly linked lists are an example of an *abstract data type* in C, which is to say that they have not only data associated with them, but also operations. Although `structs` are useful in that they encapsulate related information into a single object, they don't by themselves support any fundamental operations.
- What are some of the downsides of arrays? You need to know the size of an array when you declare it (if you do so statically). Even if we use a pre-processor directive to specify the size, whenever we change the size, we'll need to recompile and rerun the program for it to take effect. And this is, of course, something the user himself can't do.
- Of course, now that we know how to use `malloc`, we can also dynamically create an array, which is really just a chunk of memory.[1] However, if

[1] This is as good a time as any to remind you to always check for `NULL` pointers before you dereference them!

we want to add more elements to the array once the program is already running, we'll need to allocate a new, larger chunk of memory and then copy the old array into the new one, making sure to free the old one once we're done. This is, in fact, the exact approach that we took in implementing `GetString()` in the CS 50 library. One of the downsides is that the burden is on us to keep track of the length of the array and grow it if necessary.

- Singly linked lists redress this problem of fixed size. Just like our array of `structs`, each of which encapsulated a student, imagine a singly linked list of `structs`. Whereas the array could be represented as a series of contiguous boxes, the singly linked list would be represented by boxes which are not contiguous, but are "linked" in the sense that one leads to another, like a chain. Whereas with an array, we only need to keep track of the memory address of the first element—since all the other elements are contiguous in memory—with a singly linked list, we have to keep track of the memory location of each element. We do this by having each element in the list keep track of the memory location of the next element in the list in the form of a pointer. If we're clever about it, we can simply include this pointer in the definition of the `struct`. We should be careful that the last pointer in the list is a `NULL` pointer to signal the end of the list. We can visualize a linked list like so:



- Let's step back for a moment and simplify things. Instead of creating a singly linked list of students, let's create a singly linked list of `int`'s. To do this, we'll need to define each single element of the list as storing both an `int` and the location of the next element in the list. We'll create a `struct` to achieve this:

```
typedef struct node
{
    int n;
    struct node *next;
}
node;
```

Notice that the syntax is slightly different from what we used to declare a student `struct`. By using the above syntax, we can declare a new `node` instead of a new `struct node`, which are actually the same thing, the former simply being shorter. The pointer in this `struct` is going to be pointing to another one of itself, that is, another `node`.

- The various operations which are associated with linked lists are displayed on the start-up menu of `list1`:

```
MENU

1 - delete
2 - find
3 - insert
4 - traverse
0 - quit
```

  Insert and delete are trickier than they might appear at first. There are actually three cases for insert and delete: the beginning, the middle, and the end of the list. The way we've implemented insert in `list1` is in sorted order, thankfully.

- Although it might seem at first that learning this new data structure is purely pedagogical given that we can perform all the same operations with arrays, realize that for very large datasets, the time it takes to recopy an entire array in order to insert an element in it will vastly exceed the time it takes to insert the same element in an linked list.

- So how do we implement the insert operation for linked lists? Let's take a look first at the `main` method of `list1.c`:

```
/**************************************************************************
 * list1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demonstrates a linked list for numbers.
 **************************************************************************/


#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "list1.h"


// linked list
node *first = NULL;
```

```
// prototypes
void delete();
void find();
void insert();
void traverse();


int
main(int argc, char *argv[])
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
               "1 - delete\n"
               "2 - find\n"
               "3 - insert\n"
               "4 - traverse\n"
               "0 - quit\n\n");

        // get command
        printf("Command: ");
        c = GetInt();

        // try to execute command
        switch (c)
        {
            case 1: delete(); break;
            case 2: find(); break;
            case 3: insert(); break;
            case 4: traverse(); break;
        }
    }
    while (c != 0);

    // free list before quitting
    node *ptr = first;
    while (ptr != NULL)
    {
        node *predptr = ptr;
        ptr = ptr->next;
        free(predptr);
    }
    return 0;
}
```

First we have the prototypes and pre-processor directives. Then a do-while loop takes the user's input and calls the appropriate function via a switch statement. At the end, we free all the memory we've been using. Notice that the first call to `printf` has multiple strings passed to it, which is perfectly acceptable syntactically.

- So how do we represent an empty linked list? As you can see in the code above, we declare a global pointer-to-`node` and initialize it to `NULL`.

2.2 Insertion

- Before we take a look at the actual code, let's make some educated guesses as to what `insert()` is actually doing:

  1. Take in user's input:
     `n = GetInt();`
  2. Declare new `node` to store the user's input:
     `node *ptr = malloc(sizeof(node));`[2]
  3. Assign the user's input to the newly created `node`:
     `ptr->n = n;`[3]
  4. Make sure the new `node` points to `NULL`:
     `ptr->next = NULL;`
  5. Point our list pointer to the new `node`, the first and only element:
     `first = ptr;`

- We can visualize these steps using people! We'll start with Alex, who represents a `NULL` pointer. Then we allocate Olga, who represents an 8-byte chunk of memory. In her `n` field (held in her right hand), we'll put the number 2. Then we'll have Alex point to Olga, the only member of the list. Next we allocate Julie and give her the value 3 to hold in her right hand. Now we need to step back for a second.

- If the goal is a sorted linked list, we need another pointer to traverse the list and find where to insert the next value. So we point where Alex is pointing (at Olga) and check her value. It's 2, which is less than the value to be inserted, so we keep walking. Next is `NULL`, however, so we know that Julie goes at the end of the list. We assign to Olga's `next` pointer (represented by her left hand), the memory location of Julie.

- Finally, we allocate Robert and assign him the value 1. He's floating around somewhere in memory. When we traverse the list this time, now, we find that he should go before Olga. So what if we start by pointing

[2] We could ask for 8 bytes explicitly, but it's safer to dynamically determine the size of a `node` in case we change what it contains, for example.

[3] Despite the similar variable names, the compiler knows which one is the local variable and which one belongs to the `struct`.

Alex to Robert, now? We'll lose the rest of the linked list because we no longer know where in memory it is stored.

- Instead we'll have Robert point to Olga first. Now when we have Alex point to Robert, we won't lose our linked list. One of the compelling things about linked lists can be visualized now. Robert might be anywhere in memory, but so long as he points to the next element in the linked list, we can maintain the integrity of the list. So there's no shuffling around in memory when we insert a new element to the list. In contrast, if we want to insert a value into the middle of an array, we have to shift all the values down by one and we start increasing our running time.[4]
- Know that these operations are depicted in the lecture slides. For now, we'll concentrate on the actual C syntax that implements them. Take a look at `insert()`:

```
/*
 * void
 * insert()
 *
 * Tries to insert a number into list.
 */
void
insert()
{
    // try to instantiate node for number
    node *newptr = malloc(sizeof(node));
    if (newptr == NULL)
        return;

    // initialize node
    printf("Number to insert: ");
    newptr->n = GetInt();
    newptr->next = NULL;

    // check for empty list
    if (first == NULL)
        first = newptr;

    // else check if number belongs at list's head
    else if (newptr->n < first->n)
    {
        newptr->next = first;
```

[4] In the video, David inadvertently mentions quadratic running time. He was referring to insertion sort, **not** insertion into an array!

```
            first = newptr;
        }

        // else try to insert number in middle or tail
        else
        {
            node *predptr = first;
            while (true)
            {
                // avoid duplicates
                if (predptr->n == newptr->n)
                {
                    free(newptr);
                    break;
                }

                // check for insertion at tail
                else if (predptr->next == NULL)
                {
                    predptr->next = newptr;
                    break;
                }

                // check for insertion in middle
                else if (predptr->next->n > newptr->n)
                {
                    newptr->next = predptr->next;
                    predptr->next = newptr;
                    break;
                }

                // update pointer
                predptr = predptr->next;
            }
        }

        // traverse list
        traverse();
    }
```

To reiterate, we call `sizeof` when calling `malloc` not only because we might expand the definition of a `node` later on, but also because the compiler will not always return the exact number of bytes required to implement a `node`, but will often perform some optimizations in order to align it in memory along 4-byte chunks with some gaps in between.

- Once `malloc` returns, we do a sanity check to make sure it didn't return `NULL`. If it did and you were to try to dereference it, your program would seg fault. This is a feature of C.[5] Essentially, the compiler is preventing you from accessing the memory address 0x0, even though it does exist.
- In the next few steps, we follow those which we guessed a few moments ago. We take in the user input and assign it to the new `node` in addition to pointing its `next` pointer to `NULL`.
- Now we handle the cases we mentioned earlier. First, we check if the list is empty, which is the easiest case. Next, we handle the case where the new `node` belongs at the beginning of the list. Finally, the middle and end cases can be handled the same way.
- Recall our humans example. We pointed Robert to Olga and then Alex to Robert. But in our syntax above, we're assigning `first` to our `node` to be inserted. Why does this work? Well `first`, recall, is simply the memory address of the first element of the list, which would be Olga. So this assignment is equivalent to pointing Robert to Olga. Next we update Alex, telling him to point to Robert, the new first element of the list.
- When we tackle the middle and end cases, we'll need to bring in `predptr`, which plays the same role as we did when we traversed the list looking for the place to insert the new `node`. First we're checking for duplicates and if we find the value is already inserted, then we won't insert it again. This feels a little wasteful, though, since we've already allocated a new pointer and executed several other steps. What's the alternative? We could traverse the whole list searching for the value in question. But this, in effect, doubles our running time, so it's not ideal either. An improvement, then, might be to allocate a new pointer only when we're sure we need it. This ends up in a copy-paste job, which is not ideal, either.
- The end case, or tail case, is easier to handle than the middle case. All we need to do is point the `next` pointer of `predptr`, which is pointing at the last element of the list, to the new `node`.
- The middle case is perhaps the most complicated, yet it only actually requires two pointer updates. Thus, we've only seen two general cases: one that requires a single pointer update and one that requires two pointer updates.
- Note that the *singly* in singly linked list implies that we can only move in one direction down the list, in this case forward. If we want to traverse in both directions, we'll need to add a pointer to each element which points to the previous element in the list. This creates a *doubly linked list*. If we

[5] When you become an enterprise-level programmer, everything's a feature and nothing's a bug.

had implemented a doubly linked list to begin with, we wouldn't need the temporary `predptr` variable. So there's a tradeoff between performance and memory. We can improve our algorithm's running time at the cost of extra memory to store the additional pointers.

- How do we determine the length of a linked list? We have to traverse it. As soon as we encounter the `NULL` pointer, we know we're at the end of the list. This can be slow, of course, so why don't we do it once and store the result? There's no reason, after all, that the pointer to the first element of the list has to be *only* a pointer. We could make a separate `struct` that would encapsulate both the pointer to the first element and the size of the list. Each time we inserted an element in the list, then, we'd also need to update the stored size of the list.

2.3 Alternative Data Structures

- Another data structure we'll discuss is a stack. We might represent this visually as a stack of cafeteria trays. This conveys the idea that the last tray added is the first one out. We call this LIFO (last in, first out). This has some applications which are covered in CS 121. In addition, HTML tags have a hierarchical structure which can be described as LIFO. How could you implement this idea of a stack in C? It's just an array flipped vertically, at least conceptually. Of course, with a stack, you don't have random access, as you do with an array, but only access to the topmost element. Moreover, an array isn't an abstract data type, per se, because it has no operations associated with it. A stack, however, has push and pop operations associated with it.
- Queues are another data structure which we'll discuss. They exhibit a FIFO (first in, first out) policy, which is, frankly, much more "fair" than a stack. Queues, like linked lists, have insert and delete operations associated with them. Routers are one example of hardware which implement queues as a way of dealing with network traffic. Router queues also allow traffic to be prioritized: for example, voice traffic could be prioritized over movie downloads.

3 Bugs (64:00–70:00)

- The Blue Screen of Death (BSOD) is perhaps one of the more famous examples of bugs. This points to an error in a VXD (virtual device driver) at a memory address denoted in hexadecimal. Although it might seem useless, the error code given might actually be useful to some Microsoft programmer somewhere. Likewise with the Application Error alert window. Check out the slides for many more fun (or perhaps not so fun) bugs and errors.
- PC Load Letter?