Week 6 Monday
October 27, 2008
Scribe Notes

Computer Science 50
Fall 2008
Anjuli Kannan

Announcements (0:00 – 6:30)

File I/O (6:30 – 12:30)

- File I/O: reading bits from and writing them to memory
- A bitmap is a simple way to store an image. It consists of a sequence of bits specifiying the color of every single pixel.
- Although this is a very easy way to store an image, it results in very large files.
- File types such as jpg are compressed

The Data Structures We Already Know (12:30 – 20:30)

- What's a struct? A wrapper for a whole bunch of variables that have some conceptual relationship among themselves.
- In problem set 4, we put the global variables inside a struct called g.
- Why was this good? Easy to add more global variables, clear which variables were global, overall made for cleaner design.
- In this case, using a struct might not have made that much difference. But there are cases where it will.
- For instance, suppose we want to store information relating to students. Without structs, our only option is to use separate arrays for students' ID numbers, names, houses, etc., and assume that the ith element in the ID number array corresponds to the ith element in the name array.
- But if we use strcuts, we can just make one big array, each element of which is a student struct containing all the info relating to a particular student
- What are some problems with arrays?
- You have to pre-allocate memory. What if there are a different number of students this year than there were last year? Need to go back to code and change constant that indicates size of array.
- What if it keeps changing? Need to keep changing constant.
- But you don't actually need to commit to these details at compile time. You could use malloc to dynamically allocate at runtime based on number of students.
- But if we have malloc at our disposal, we could design a new data structure altogether.

Looking ahead to Hash Tables (20:30-24:00)

- As an example, consider Google. Suppose that they store all the webpages of the internet in an array.
- Again, this will present problems in the event that the internet grows
- Now, consider search. What if we want to search for, say, flamingos? Assuming that the array is sorted, we know we could do it in log n.

- log n seems fast, but if you think about the amount of data Google is dealing with, and the number of searches it handles every second, log n would actually be quite slow
- Ideally, we want to be able to do things in constant time
- Next time, we'll see how to do this using hash tables.
- In order to understand hash tables, we need to first understand linked lists

Introduction to Linked Lists (24:00-34:30)

- Imagine the registrar has an array to represent all the students. Suppose that the array is full and it needs to add a student.
- If we don't want to recompile, we can dynamically reallocate a larger array
- But allocating a huge chunk of memory like that is very, very slow
- Now suppose that instead of allocating an entire contiguous stretch of memory at once, we allocate memory on an as-needed basis
- When we start with one student, we allocate one struct
- When we need another student, we allocate one more struct, and so on.
- The problem with this situation is that we cannot assume that the second student will not be next to the first student, so we can't treat this set of structs like an array
- If our structs are all at different locations in memory, how can we remember where they all are?
- Make one member of each struct be a pointer containing the address of the next struct
- This is called a linked list. The structs are linked together by means of pointers.
- We can visualize this as a string of boxes chained together by arrows. Each box represents a struct, and the arrows represent the pointers.
- Each struct only knows about the next struct in the sequence, but if we start at the beginning and follow all the pointers, we will come up with every element in the "list"
- So all we need is a pointer to the first node in the list
- We will call this special type of struct, which contains a pointer to another one of itself, a "node" and define it in the following manner:

```
typedef struct _node
{
    student * student;
    struct _node *next;
}
node;
```

- A "node" contains two things: a pointer to a student, and a pointer to the next node

- In the last node, the node pointer would be set to NULL
- Notice in the definition above that we are sort of giving the structure two names: _node and then node.
- The reason for this is that we wish to make one of the data fields a pointer to another node. This results in a circular definition: we must refer to a node in the definition of the node!
- We can't use the word node inside the definition because it's not completely defined until the curly braces close. Therefore, we give it a kind of termporary name _node before the curly braces and use this name to refer to it within the definition.
- After the curly braces close, the structure is now defined as node and we never use the name _node again.
- The convention of using the same name preceded by _ is not necessary, but is a convention in C.

Manipulating Linked Lists (34:30 – 40:30)

- We declare the pointer to our list as node * first = NULL;
- This is all you need to remember to keep the list around, just as with arrays, all you need to remember is the address of the first element or the name of it (say, a)
- In list1.h we declare a type of node which contains an integer instead of a student pointer
- Let's make a program to manipulate linked lists
- Here's a start:

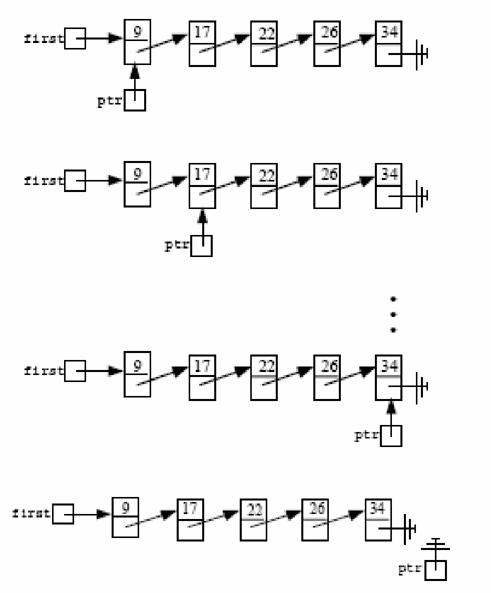
```
int
main(int argc, char * argv[])
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
               "1 - delete\n"
               "2 - find\n"
               "3 - insert\n"
               "4 - traverse\n"
               "0 - quit\n\n");

        // 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;
            }
        }
        return 0;
    }
```

- All this does is create a menu of options, display it to the user, read in the option that they choose, and then call the appropriate function
- We also declare our list at the top of the program (not shown): node * first;
- Now we have to write functions to accomplish the possible tasks the user might want to do

Traversing a Linked List (40:30 – 49:00)

- Alright, so first let's think about traverse(). What does it take to traverse the list?
    - Start at first.
    - While pointing at valid node, follow the pointer.
    - Print the value or the node you're at.
    - Now follow that node's pointer and repeat.
- Here's a graphical representation:



- Now let's think about the code.
- We'll need something to point with, a temporary pointer. Call it node * ptr.

- (We can't keep using first because that would cause to forget where the head of the list is.)
- Then we'll need a loop to iterate through the nodes. Start by pointing at first, and keep going until ptr is equal to NULL. While in the loop, just print out the value of the current node.
- ptr->n means that we want the n datum from ptr . We could have also said (*ptr).n but ptr->n is just concise and stylistically preferable.
- (The reason we're not using * here is that *ptr refers to the whole chunk of memory. We're also not using ptr.n because ptr is a pointer, whereas . syntax is for actual structures.)
- What's the update step? Go to the next field of the current pointer: ptr = ptr->next. Visually, this is like rearranging arrows so our temp pointer (ptr) now points to the next node in the list.
- Here's the code:

```
void
traverse()
{
    // traverse list
    printf("\nLIST IS NOW: ");
    node * ptr = first;
    while (ptr != NULL)
    {
        printf("%d ", ptr->n);
        ptr = ptr->next;
    }

    // flush standard output
    fflush(stdout);

    // pause before continuing
    sleep(1);
    printf("\n\n");
}
```

Searching a Linked List (49:00 – 57:00)

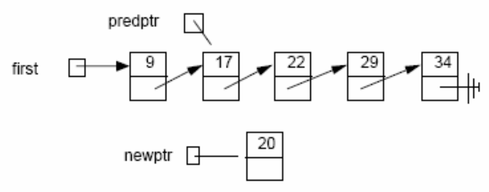
- Now let's write find()
- Our goal now is to take a list and search for an element. If it's there, print YES. Otherwise, print NO.
- First, we ask the user what they are looking for and read that into a variable.
- Again, we'll start with a temporary pointer that we initialize to first: node * ptr = first;
- Then we're going to have a loop that continues so long as we're not at a NULL pointer.

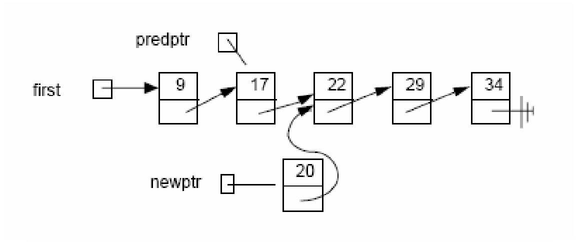
- If n equals the value of that node, print "YES!" and break out of the loop.
- Otherwise, go to the next node.
- Here's the code:

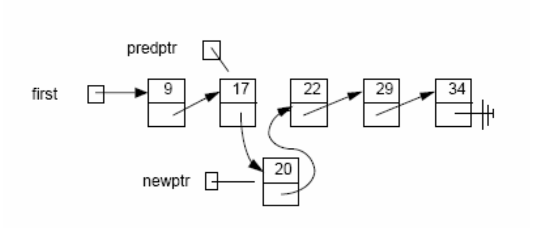
```
void
find()
{
    // prompt user for number
    printf("Number to find: ");
    int n = GetInt();

    // get list's first node
    node * ptr = first;

    // try to find number
    while (ptr != NULL)
```

Inserting an Element in a Linked List (57:00 – 65:00)

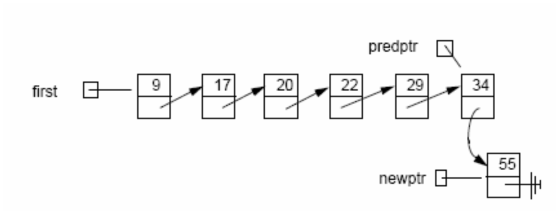
- From our point of view, the whole list is represented by the pointer first. So that's where our story begins.
- We declare a pointer node * predptr and point it to first
- Where do we go next? first->next. Then we go to that node's next, and so on. At each node, we check the value to see how it compares to the element we wish to insert.
- When we get to an element whose value is greater than the value we wish to insert, we make a new node for the guy we're inserting, but then we have to back up a step to insert him
- But wait, we don't have a backtrack mechanism, so what do we do? We have to have two pointers from the get go.
- That way, we can keep our pointer at one node and use another pointer to "look ahead". If the value that we look ahead to is still less, then we can update. Otherwise we insert in between.
- When we finally find a place to insert our node, we simply redirect pointers to fit him in:



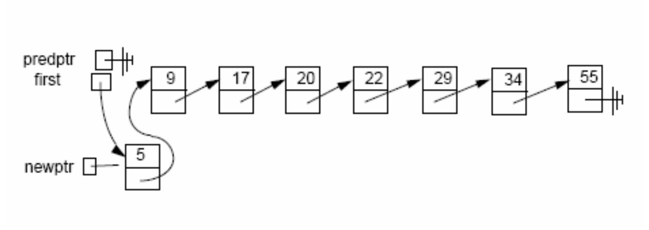




- What if we want to insert at the tail? This case is a bit easier. We again make a new node for the guy we are inserting and make the last element point to him.

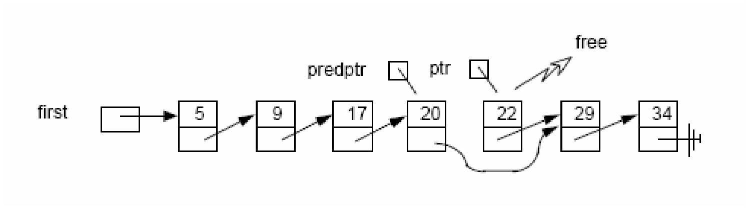


- What if we want to insert at the head? Make a new mode for the guy we're inserting , make him point to the node first is pointing to, and make first point to him. Order is critical!

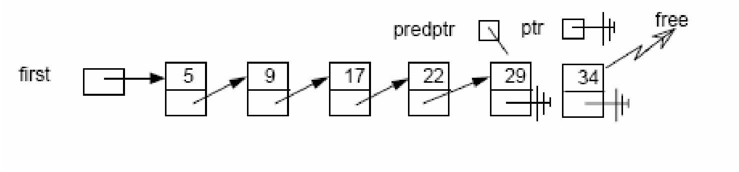


Deleting an Element from a Linked List (65:00 - 66:30)

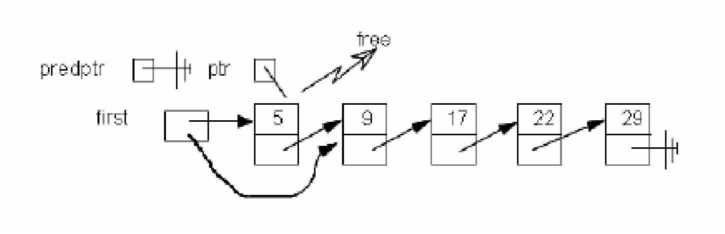
- Here, our goal is to get rid of an element
- Again, we walk through the list looking for the element using predptr
- Once we find him, we need to merge the list before him and the list after him
- Finally, we free the node we removed
- Order is once more very important! We must stitch the halves together, *then* free the node.
  Otherwise, we'll be following a pointer that is no long ours to follow



- What if we want to delete the tail? We must make the second to last element the new tail by making its next pointer NULL, and then free the tail:



- What if we want to delete the head? Make first point to the second element and free the head:



Implementing Insertion (66:30 – 76:00)

- Now take a look at the code for insert():

```
void
insert()
{
    // try to instantiate node for number
    node * newptr = (node *) 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;
        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();
    }
```

- We first have to instantiate a node for the element we're inserting.
- Then we can take advantage of the sizeof() function to determine how much memory to allocate. This way, we don't have to hardcode any assumptions about the size by specifying an exact number of bytes. We keep the code general so it will remain correct even if we changed something about the node structure.
- As usual, we make sure that malloc() was successful by checking that newptr is NULL
- Now we have to consider every possible case
- First, suppose we have an empty list. We must consider this case because if the lsit is empty, then first is NULL. What would happen if we tried to follow first→NULL? Seg fault!
- Fortunately, this case is the easiest to deal with. If the list is empty, simply make first point to the node we just made. That's what first = newptr accomplishes.

- Next case: we want to insert at the head. This happens if the new guy's n value is less than the first guy's n value.
- To deal with this case, we point our new guy to the head and point first to the new guy.
- Finally, our last case is if we have to insert the node at the middle or the tail. We walk through the list following nodes' next pointers and examining the n values of each node we visit. This also gets divided into a few cases
- If we find a node with the same value we are inserting, we stop, call off the whole operation, and free the node we just made.
- Otherwise, we have to insert. By seeing if predptr is pointing to NULL, we can determine if we're at the end of the list.