CS 50: Introduction to Computer Science I
Harvard College
Fall 2007

Scribe Notes
Week 6 Wednesday
Anjuli Kannan

Conducting a Raffle (0:00-13:00)

- Thanks to Microsoft, we're raffling off a copy of Office 2007
- Let's write some code to do the raffle for us
- We've got an array containing the names of 30 or so students – this is placed in students.h to provide some level of privacy to these students
- First, we want to determine the number of students. Can we just say int n = sizeof(students)?
- No, that will give the total number of bytes of the array. But each student's name takes up four bytes in the array. (A name is a char * and a pointer is just an int)
- To avoid making assumptions about the elements of the array, we find the number of elements by dividing the total size by the size an element:
  int n = sizeof(students) / sizeof(students[0])
- Next, we seed the random number generator with the time: srand(time(NULL));
- Then, to choose a random student, we generate a random number using rand(). This number will be between 0 and the maximum, which is probably about 2 billion, so we then have to take it modulo the total number of students.
- To make things more interesting, we generate 100 different random numbers in a for loop and make the winner the last number to be generated. We tell the computer to "sleep" in between each number to give it the effect of animation.
- Yet when we run this, we don't get any numbers printed out.
- Let's add the following line after printf: fflush(stdout)
- This takes whatever is on the print stream and literally flushes it out.
- For performance reasons, printf() doesn't immediately print to the screen. We force it to actually print out by calling fflush()
- Now the numbers print out, but they just appear "on top" of one another. So if 22 gets overwritten with 6, we see 62!
- To fix this, we tell each number to take up 3 places: printf("\r%-3d", rand() % n);
- Here's the final code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include "realstudents.h"

int
main(int argc, char * argv[])
{
    int n = sizeof(students) / sizeof(students[0]);
    srand(time(NULL));
    printf("Rolling a %d-sided die...\n", n);
    for (int i = 0; i < 10; i++)
```

```
        {
            printf("\r%-3d", rand() % n);
            fflush(stdout);
            sleep(1);
        }
        printf("\nThe winner is... %s!\n", students[rand() % n]);
    }
```

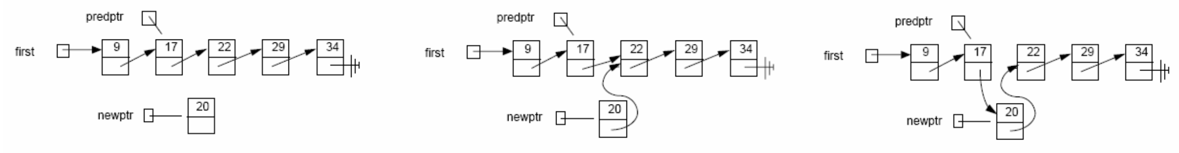
- We run it again, and this time the last number to get generated is the winner.

Linked Lists (13:00-18:00)

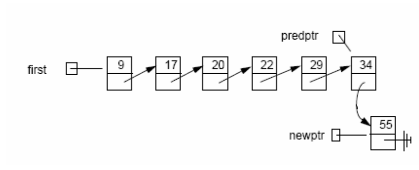
- What's the motivation again?
- Arrays are convenient for lookups, but insertions and deletions are time expensive.
- Insertions and deletions are time costly because we have to iterate over the array finding the correct location, shift elements down, and grow/shrink our array.
- Linked lists provide a much more practical structure for these operations.
- In linked lists, instead of having one big contiguous chunk of memory that we walk through, we have many pieces of memory that we instantiate one at a time and move between by following pointers.
- Last time, recall, we wrote a framework for dealing with linked lists and wrote traverse() and find().
- What if we want to insert an element?

Inserting an Element in a Linked List (18:00-25:00)

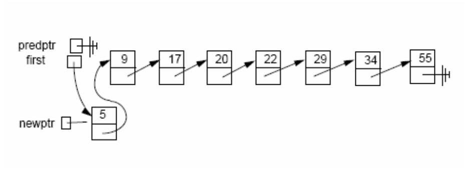
- From our point of view, the whole list is represented by the pointer first. So that's where our story begins.
- We declare a pointer node * predptr and point it to first
- Where do we go next? first->next. Then we go to that node's next, and so on. At each node, we check the value to see how it compares 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.

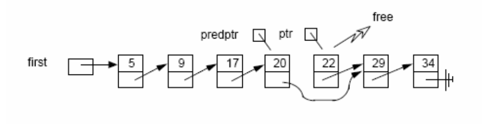


- What if we want to insert at the head? Make a new mode for the guy we're inserting , make him point to the node first is pointing to, and make first point to him. Order is critical!

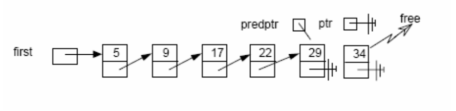


Deleting an Element from a Linked List (25:00-27:30)

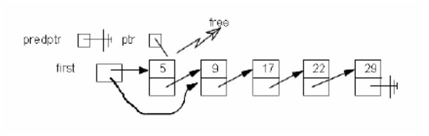
- Here, our goal is to get rid of an element
- Again, we walk through the list looking for the element using predptr
- Once we find him, we need to merge the list before him and the list after him
- Finally, we free the node we removed
- Order is once more very important! We must stitch the halves together, *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 Deletion (27:30-34:00)

- Here's the code for deletion:

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

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

    // try to delete number from list
    node * predptr = NULL;
    while (ptr != NULL)
    {
        // check for number
        if (ptr->n == n)
        {
            // delete from head
            if (ptr == first)
            {
                first = ptr->next;
                free(ptr);
            }

            // delete from middle or tail
            else
            {
                predptr->next = ptr->next;
                free(ptr);
            }

            // all done
            break;
        }
        else
        {
            predptr = ptr;
            ptr = ptr->next;
        }
    }

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

- Let's walk through this a bit. (You'll get more comfortable with this when you implement it yourself in ps5!)
- First, we ask for the number to delete and store that in n
- Then, we make two pointers so that we can eventually "stitch together" two halves. Call these ptr and predptr.
- Now, we iterate through the list and check to see if ptr→n is equal to the value we're looking for
- If we find that ptr==first, we are deleting the head. So we update first to point to the element after the head, and then we free the head.

- But if the element we are deleting is at the tail or the middle, we can handle these cases in the same way. We point ptr to the guy we're deleting and point predptr to the guy before him. Then we just update predptr→next to be ptr→next to achieve the stitching.
- Notice that in the case that we are deleting the tail, this just makes predptr→next be equal to NULL, which is exactly what we wanted. So we don't have to break this into two cases.
- Now, if we're not at the desired element, we just move on. We set predptr = ptr and then update ptr to ptr→next
- If we never find the right guy, nothing happens. We just complete the while loop, print out the contents of the list, and exit the function.

Implementing Insertion (34:00-45:00)

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

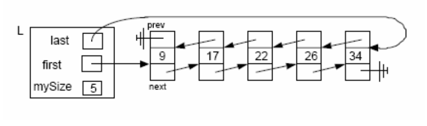
```
void
insert()
{
    // try to instantiate node for number
    node * newptr = (node *) malloc(sizeof(node));
    if (newptr == NULL)
        return;

    // 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();
}
```

- This is very similar, but we first have to instantiate a node for the element we're inserting.
- Again, 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.
- That's pretty much it for insert(). You can raed through the rest of it on your own, or, better yet, try implementing it yourself!
- But here's a question. Wouldn't it be useful if we also had a pointer last that always pointed to the last element of the list?

- Then if we had a particularly large element to insert, we could start at the end of the list and move backward. Well, we'd also need a way to move backward because right now our pointers only point forward…

Doubly-Linked Lists (45:00-50:00)

- In a doubly-linked list, each node has a pointer to the next guy *and* the previous guy
- To refer to the list, we keep around a pointer to the first guy, a pointer to the last guy, and a size (number of elements)



- Advantage of having prev and next pointers: for insertions and deletions we don't need to have two pointers walking through the list. When we get to the place where we need to "stitch" we just look at the current guy's previous and next pointers to get the two halves to be merged.
- Advantage of keeping track of size: running time. There are instances in which we'd want to know the size, and actually calculating this would be an O(n) operation.
- Note, however, that now that we are keeping track of size, there is an extra step to insertions and deletions
- OK, let's talk run times.
- For linked lists, what are the worst case running times of find(), insert(), and delete()? O(n) in all cases, assuming a sorted list
- But we can actually do better. With another data structure called a hash table, we can whittle down the runtime of all of these to constant runtime!!!! Sound to good to be true? Well, there's a cost, but we'll see that on Friday…