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

Scribe Notes
Week 6 Friday
Anjuli Kannan

Generalizing Linked Lists (0:00-8:00)

- As you may recall, we had a new data structure last time called a linked list
- A linked list consists of nodes, each of which contains an int and a pointer to the next node
- But this is a bit inelegant (a) because we mix data and metadata by including a pointer inside the node and (b) because it's not reusable for data types besides ints
- We'd rather have a library that is versatile enough to be used in different contexts
- But oh, we can do better. In list2.h, we "factor out" the type of data we want to store by making the node contain a pointer to another node and a *pointer* to a student object:

```
typedef struct
{
    int id;
    char * name;
    char * house;
}
student;


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

- This way, not all the data is in the node itself. Rather, the node just contains a pointer to the data.
- The only thing this changes is some of the syntax. Particularly in traverse() we now need to print out each datum in the student structure—we can't just print a student as we could an int.
- To print out the data in the student structure, we have to follow a sequence of pointers. For example: ptr->student->name
- Problem: we can still only store one type of object in this linked list. To generalize, we could put a void pointer in the definition of the node:

```
typedef struct _node
{
    void * datum;
    struct _node *next;
}
node;
```

- Then when we declared a node, we'd have to cast the pointer appropriately.
- For now, we won't deal with that. We are satisfied with the layer of indirection provided by making a pointer to a student be one of the data fields.

Hash Tables (8:00-22:30)

- So next week, you'll have to make a dictionary
- We'll give you some framework, but you'll have to implement functions to (a) load in a dictionary and (b) determine whether a word is in a dictionary
- How might you do this? A linked list, perhaps. But you'll find that this might make searches, insertions, and deletions pretty slow.
- Can't we design a data structure where these operations are all constant time?
- A hash table is a structure that allows you to store pieces of data just like a linked list, but you insert and delete elements in constant time.
- Suppose you want a data structure to store values between 1 and 25.
- If we were using a linked list, we'd just make a node every time the user entered a number and insert it into the list
- If we wanted to determine whether, say, 9, was in the list, we'd walk through in linear time to return TRUE or FALSE
- What's a better way of remembering the numbers the user has typed in if we can assume some bounds on those numbers?
- Just make an array with 25 slots all initialized to zero. When the user enters, say, 9, put a 1 in the first slot of the array.
- Now runtimes of insertion, deletion, and lookup are constant! Sweet!
- But we made a huge assumption here. What if we don't have bounds? Now we've got a problem.
- Alright, we'll just make an array with a slot for every possible number. That'll be about… 4 billion cells or 16 gigabytes! All just to determine whether or not a user has entered some number. Probably not the most efficient approach.
- Well, we can't assume bounds, but we can assume sparseness. That is, the user is not going to type in all 4 billion numbers. They might only type in, say, 20 different numbers.
- One solution is to keep our array smaller, but take the input mod the size of the array to see which slot to change.
- For instance, we keep 25 slots in the array. If the user enters 25, we put a 1 in slot 0 (because 25 mod 25 = 0)
- Now our problem is ambiguity. When we look back at our array, how do we know whether the user entered 25 or 0 or 50? We now have collisions.
- Well, for starters, we could put the actual number they enter in the array, rather than just a 1 or 0.
- OK, so suppose the user enters 25 and we put a 25 in a[0]. Then they enter 50. What do we do?
- We'll discuss a few ways to address this problem in a minute. But first, let's talk about probability.
- We only have to think about what to do when two numbers hash to the same value if it's fairly likely to happen. But the probability of actually having a collision is really low, right?

The Birthday Problem (22:30-30:00)

- Actually, determining the probability of collisions is a lot like solving the famous birthday problem: given n people, what is the probability that 2 of them have the same birthday?
- The reason these two problems are the same is that we can think about the birthday problem as having an array of 365 slots and throwing students into slots that correspond to their birthdays.
- Assuming a random distribution of birthdays (which is arguable…) the probability that two students end up the same slot is the same as the probability that two randomly chosen numbers end up the same slot
- When a number corresponds to a particular slot we say that it *hashes* to that value. So if our hash function is simply the modulo operator, then 25 hashes to 0.
- OK, so how do we find the probability that two people from a group of n have the same birthday?
- This is easier calculating by considering the probability that all n people have different birthdays, and subtracting this probability from 1.
- So if n = 1 person, the probability that "all 1" students have different birthdays is 365/365 = 1. The probability that 2 have the same birthday is 1 – 0 = 1.
- If n=2 people, the probability that all 2 students have different birthdays is (365/365)(364/365) = 364/365. The probability that 2 have the same birthday is 1 – 364/365 = 1/365.
- If n=3 people, the probability that all 3 students have different birthdays is (365/365)(364/365)(363/365). The probability that 2 or more have the same birthday is 1 minus this product.
- For n people, the probability of having no people with the same birthday is

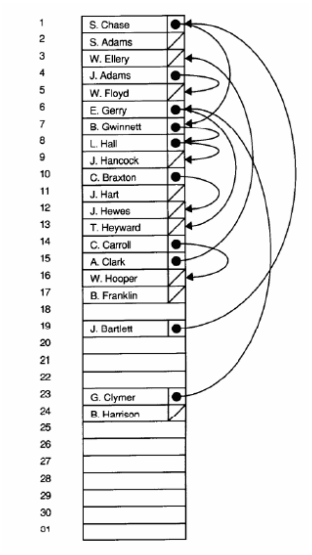
$$\bar{p}(n) = 1\cdot\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)\cdots\left(1-\frac{n-1}{365}\right) = \frac{365\cdot 364\cdots(365-n+1)}{365^n} = \frac{365!}{365^n(365-n)!}$$

- We can find the probability of a collision by subtracting p(n) from 1.
- By the time we get to n=23, 1-p(n is over a half!
- By the time we get to n=40 the probability is 90%!
- Recall that collisions in birthdays are analogous to collisions in our data structure.
- So this is bad news for our hash table. Apparently collisions are a lot more likely than we thought…

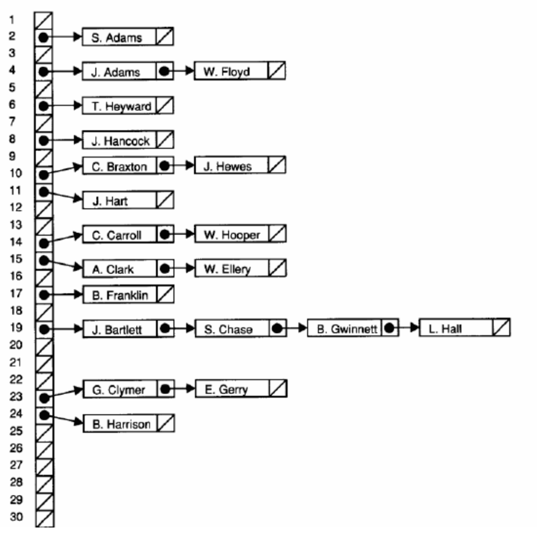
Dealing with Collisions (30:00-40:00)

- One way to deal with the collision between 25 and 50 is to just put 50 in the next available slot—say, at index 1. And if that's taken, to go to the next one, and so on.
- Later on, when we want to see if 50 has been stored, we first go to index 0, and then keep checking slots until we find an empty one.

- We call this approach *linear probing*.
- The problem with linear probing is that, in the worst case, we might end up with linear time lookup. Which is pretty disappointing because we might have gone through a linear time algorithm just to find a slot for it!
- Wouldn't it be better if we could keep track of where we put each element?
- In *coalesced chaining*, after we find a location for an element, we put a pointer to that location in the spot where it was supposed to go :



- For smaller examples, it does not appear to be much of an improvement, but as tables grow in size, chaining allows us to skip around rather than checking every single slot in sequence hunting for a particular element
- Another solution is *separate chaining*. Here, each slot in the array contains a linked list of elements that belong in that slot:



- This is dynamic because the linked list will only be as long as the number of element you put there. This saves memory.
- If our hash function is good, the elements will be pretty well distributed among the different slots, so now chain will be that long
- What's the asymptotic running time of insertion for a hash table with separate chaining?
- Well, since in the worst case all the elements might go in the same list, the asymptotic runtime is O(n)
- But this is purely theoretical. In practice, a good hash function will pretty evenly distribute numbers, so we will never end up with a linear insertion time.
- In reality, each chain is likely to have an approximate length of n/m, where m is the size of the array
- This results in an insertion time of about n/m
- Again, this will technically give us an asymptotic runtime of O(n) since we disregard constants, but in practice it is more efficient than the insertion time for a linked list (which was also O(n))
- Now what do we do if we are hashing strings instead of ints? Or what if we had student objects?
- One idea: look at first letter of name and hash to corresponding number (A=1, B=2, etc.)
- Problem with this is that it is not randomly distributed. A, for instance, is a much more likely hash value than Q.
- Still, it is passable, and is a valid design decision if you aren't too concerned about imbalance in your hash table.