Computer Science 50
Fall 2009
Scribe Notes

Week 7 Wednesday: October 21, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 7 Wednesday: October 21, 2009
Andrew Sellergren

1 Announcements (0:00–10:00, 12:00–16:00)

- This is CS 50.
- 1 new handout.
- It might seem a little early,[1] but you should begin brainstorming for your final project. The climax of the course is the CS 50 Fair, which will give you an opportunity to display your project to friends, family, and random passersby. While for all the problem sets thus far, you've been handed a skeleton framework from which to create a complete program, for your final project, you'll most likely have to start from scratch. To help you do so, however, we'll be offering a number of seminars on a variety of topics.
- As mentioned on Monday, if you have a desire to develop a smartphone app, you should express interest (by e-mailing David)in obtaining one of the Android phones which have been generously donated by Google.
- Check out the Fair page for videos, music, pictures, and fun facts from last year's CS 50 Fair! Also RSVP for this year's fair.
- A sampling of funny comments from surveys that have already come in:
  - "I learn how to hack the world."
  - "The difficult of CS50 causes a buffer overflow in the array of memory called my mind."
- In seriousness, though, we do take your feedback. . . well, seriously. Allow us to address a few of the recurring serious comments:
  - We're aware that Sanders Theater can be a little daunting as a lecture space, so we really encourage you to stop David by raising your hand with questions. It would certainly be nice to hear someone else's voice for a change.
  - On workload: we know this course asks a lot of you, but only so that you can get the most out of it. That sounds cheesy, yes, but it's also true. If we were to make the problem sets half as long, you would probably get half as much out of this course. If you're feeling overwhelmed, by all means lean on the teaching staff, use your late days, and, most importantly, don't get discouraged. Trust us, no one finds this easy.[2]
  - On grades: we know that you're worried about them. You probably wouldn't be here at Harvard if you didn't care about grades.[3] But do realize that grades aren't everything. Moreover, they're subjective.

[1] It's not!
[2] Except that guy sitting next to you in lecture. Yeah, him. Give me a good evil eye. What a jerk.
[3] Except that guy sitting next to you. I bet he's a legacy. Psth.

The word "subjective" has often taken on a negative connotation in this context, but we don't think it's such a bad thing. Our focus is much more on providing written feedback that will help you improve your programming than on assigning numerical grades. The numbers are only one small part of the grading process. And as subjective as those are, they will be normalized across section at the end of the term. As well, each grade will be individually discussed between David and the grading TF. We put a lot of thought into it!

  - David apologizes for talking too quickly. It's been a common theme of comments in all the courses that he's taught. He's doing his best, though, to walk the fine line between too fast and too slow.

- More grandiosely, however, allow us to justify the very existence of lectures. After all, what's the point in coming to class when you can watch the video and read these notes? Well, we certainly hope that lectures are a way of piquing your interest in the material and engaging you as you learn it. That's why we have candy and fun demonstrations. But more than that, we hope that lectures will provide you with a mental framework that you can build upon as you attend sections and complete problem sets.

- A sampling of the more serious comments that we want to address:

  - "I think he's nice, but he expects that we catch on more quickly than we actually do."
  - "... much of what he says goes over my head."
  - "I've heard people coming out of lecture having no idea what was going on..."

  This is why we encourage you to raise your hand during lecture and to lean on the teaching staff. Please help us make lecture worthwhile for you to attend!

- And, to end our discussion of the surveys, two more funny comments:

  - "This comment would like to be shown on the projector in lecture. It's his birthday."
  - "I is dun with survey nao? kthxbai :)"

2 David's Auditory Hallucinations (11:00–12:00)

- I didn't hear anything, did you?

3 Where We've Been and Where We're Going (16:00–20:00)

- We started with the fundamentals: loops, conditions, statements, etc.

- Then we talked about how we might implement some of these fundamentals using C syntax.
- From there, we began our discussion of data structures, including arrays, structs, and linked lists.
- After we considered the shortcomings of these simple data structures, including slow insertion time and slow lookup time, we introduced a more sophisticated data structure, which we'll be exploring for the next two lectures: the hash table.
- Beyond these next two lectures, we'll see hash tables when we begin our segment on web programming. Both PHP and JavaScript make extensive use of hash tables, although they call them something different.
- That hash tables are utilized in multiple programming languages is merely one example of how the knowledge you gain in this course transcends any one particular programming language. That is, you're not learning C, or PHP, or JavaScript; you're learning how to program.

4 Hash Tables (20:00–55:00)

4.1 Motivation

- As we mentioned last time, the motivation for learning how to use hash tables is to resolve the issues of fixed length and slow lookup time posed by arrays and linked lists, respectively. In computer science, tradeoffs are a common theme. Running time can often be decreased at the expense of memory usage and vice versa. Efficiency can often be increased at the expense of effort on the part of the programmer. There's no such thing as a free lunch![4]
- Recall that searching a linked list runs in $O(n)$ because in the worst-case scenario, the value being searched for is at the end of the list, the entirety of which must be traversed in order to find it. And we can't use binary search on it because we don't have random access to the middle of the linked list as we do with an array.
- In an ideal world, our data structure would be a black box, which is to say that it takes input and gives output efficiently even if we don't know exactly how it does so. What we want is to be able to insert data into our structure, ideally in constant time, and retrieve the same data from that structure, ideally just as quickly.

[4] Except Lunch with David.

4.2 Implementation

- A hash table relies upon a well-designed hash function, which takes as input the value to be stored and deterministically returns as output the location where it should be stored, in order to spread out data evenly. Let's try writing a very simple hash function:

```
#define ERROR -1

int
hash(char *s)
{
    if (s == NULL)
        return ERROR;
}
```

  First, we do a sanity check to make sure that the value we've been passed isn't `NULL`. In the case that it is, we return -1, a sentinel value, because it's presumably not within the range of locations in our hash table, i.e. it's not a valid index into the array. We can also `#define` this sentinel value as a constant so that it's not a magic number. We're returning this value with the assumption that the caller function will check it before proceeding.

- Let's assume that the input string is composed only of alphabetic characters. We need to return a number based on that string.

```
#define ERROR -1

int
hash(char *s)
{
    if (s == NULL)
        return ERROR;

    int n = (int) s[0];
    return n;
}
```

  Now we're converting the input string to a number by explicitly casting it. One problem, however, will be that numbers won't begin with 0 since we'll get their ASCII values instead. We could either subtract 'a' or 'A' to fix this or we could simply mod by 26. Both approaches will ensure that we return a number 0 through 25, but the latter will not map A to 0, B to 1, etc. But we don't really care so long as we can reproduce this mapping when we want to look up values in our hash table. When we want to look up a value in our hash table, we simply pass it to the hash function to find its location and then jump to that location in the hash table.

4.3 Optimization

- One of the problems with this simple hash function is that it won't allow for more than 26 values. Beyond the 26th value, new values will clobber the old. This hints at the problem of *collisions*. Of course, we could fix this problem simply by making our hash table much larger than it needs to be. This has its own problems, however, among them using a large amount of memory which will make hardware caches less effective.

- For starters, a better hash function would be one that used both the first and second letters of the string in order to determine its location in the hash table. The probability that two students will have the same two letters of their last names, for example, is less than the probability that two students will have the same first letter of their last names. We might take into account the first two letters by adding them together:

```
#define ERROR -1

int
hash(char *s)
{
    if (s == NULL)
        return ERROR;

    int n = (int) s[0] - 'A' + (int) s[1] - 'A';
    return n;
}
```

  This has its own problems, however. For example, it will return the same hash index for two students who have last names beginning with the same letters but in different order, e.g. Babar and Abyssinian. We'll need a hash table that is twice the original size to accommodate this extra consideration.

- Generally, the more letters we take into account, the fewer collisions there will be. What if we include all the characters in the string:

```
#define ERROR -1

int
hash(char *s)
{
    if (s == NULL)
        return ERROR;

    int n = 0;
    for (int i = 0, length < strlen(s); i < length; i++)
```

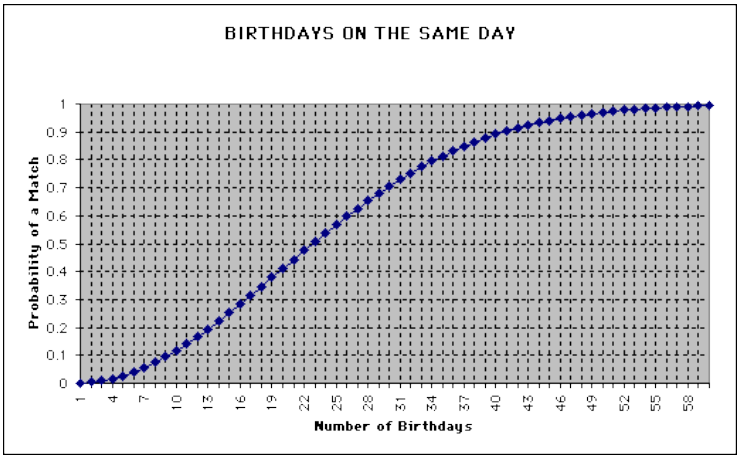
```
        n += s[i] - 'A';

    return n;
}
```

This won't eliminate collisions altogether, but it will make them less likely. But how much less likely? Let's start with a simpler example. If we were to return `rand() * 26`, or, in other words, a random number between 0 and 26, what would be the chances of a collision? This is another version of the birthday problem.

- The birthday problem: in a room of n CS 50 students, what's the probability that at least two students share the same birthday? First, we'll assume a uniform distribution of birthdays (which isn't really realistic, in fact). Then we can calculate the probability by considering the opposite event, namely that no two students have the same birthday:

$$\begin{aligned}\overline{p}(n) &= 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{n-1}{365}\right) \\ &= \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n} \\ &= \frac{365!}{365^n \, (365 - n)!}\end{aligned}$$

This math simply formalizes the following logic: the first student can have any one of the 365 available birthdays without any collisions; the second student, however, can only have 364 out of a possible 365 birthdays without there being a collision; the third student, 363 out of 365; and so on. Let's take a look at a graph of this probability since it's easier to visualize:



The takeaway message from this discussion is how high the probability of a collision is even for a small number n—for example, 90% in the case that $n = 40$. We need to come up with a way of dealing with collisions, it seems.

- In the case of a collision, we're better off chaining values together rather than clobbering the old value with the new. So if we have two students with the same birthday, we'll link them together. This way, our hash table is actually an array of linked lists rather than an array of simple values.
- But there's a problem. We're no longer running in constant time for lookups. In fact, if our hash function is poor enough or we're simply unlucky enough to have the maximum number of collisions, then our hash table will be no better than a single linked list of length n. In that case, our lookup will run in $O(n)$.
- Finally, though, we'll see that the real world trumps theory. On average, our linked lists will be of length $\frac{n}{k}$, where k is the number of locations in our hash table. In practice, dividing the number of iterations by a constant factor k will yield significant gains in performance.

4.4 Utilization

- If you decide to implement a hash table for Problem Set 6, then you'll discover the importance of a good hash function. You'll be tasked with loading up the 140,000 words of a dictionary into a data structure of some kind in order to spellcheck a number of texts. Your challenge will be to implement the fastest spellchecker possible! If you choose to opt in to this competition, your benchmark results will be posted on The Big Board.
- Let's map out a strategy for implementing Problem Set 6:

```
char *hashtable[100000];

void
insert(char *s)
{
    // find a location for s

    // insert s at that location; if collision, append to chain
}

bool
find(char *s)
{
   // find the location for s
```

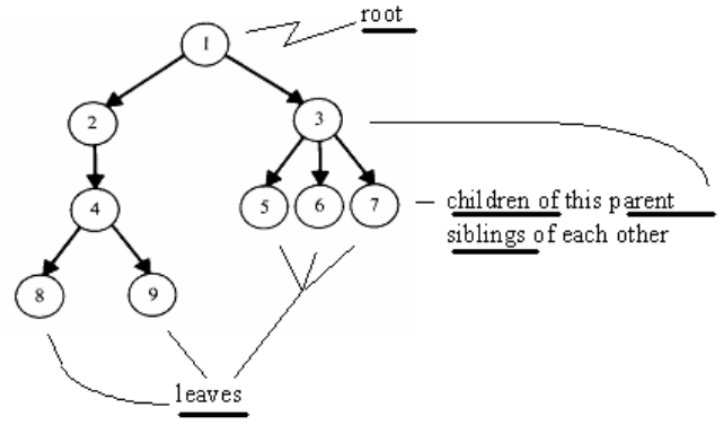
```
    //return true if found else false
}
```

That's it! Go ahead and type `submit`. Actually, wait, you should probably have some actual code in there, come to think of it.

- Realize that we've already handed you the code for implementing linked lists, which you'll need in order to handle collisions.

5 Tries and Trees (55:00–60:00)

- Tries are another data structure which might prove useful to you as you complete Problem Set 6. A trie is a type of tree (think of a family tree), which, for the purposes of Problem Set 6, will have arrays for its nodes. These arrays will contain as elements pointers to other arrays.
- Let's take a concrete example. If we want to look up a word beginning with the letters "pa," we begin by accessing the "p" index of our root node. This gives us a pointer to another array, which we follow. We then access the "a" index of this second array. And so on with the other letters in the word until we reach the end of the word, at which point we need to have a flag variable that signifies the end.
- Trees are data structures consisting of nodes, each of which may have any number of children.footnoteIf each leaf has a maximum of two children, the tree is a binary tree. Children of the same parent are siblings. Terminal nodes—those at the bottom that have no children—are called leaves. See the diagram below.



Later in the course, we'll see how we can use a specific type of tree called a *binary search tree* to achieve logarithmic lookup time.

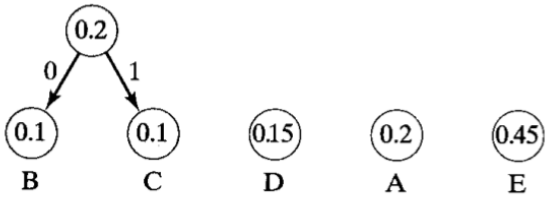
6 Compression and Huffman Coding (60:00–76:00)

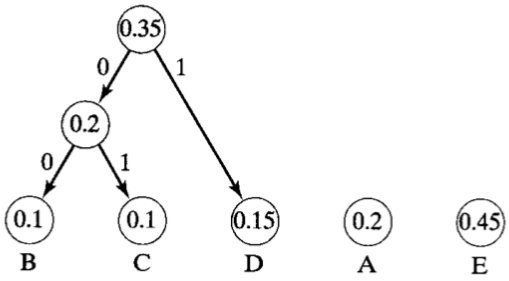
- If you've ever used a computer before, you've probably compressed a file at some point in your life. How is this actually implemented under the hood?
- If we wanted to compress an essay of yours, we could simply delete a few of the pages. This would be an example of *lossy* compression and while it might be less than ideal for text files, it actually words reasonably well for pictures and videos. Think of YouTube.[5]
- *Lossless* compression is useful for text files because none of the original data is lost. To implement it, we need only examine the text file and determine how we might represent the data with fewer bits. If we look at an ASCII chart, we can see that many of the characters will be used infrequently, if at all, in a text file. If we don't actually use 255 different characters in a text file, then we don't need a full 8 bits to represent each character. This is the idea behind Morse Code, which uses dots and dashes to represent characters.
- So we can take this idea of finding common patterns of characters and replacing them with shorthand to a lower level—the bit level. The compression utilities on computers generally analyze files to find bit patterns which can be represented more succinctly. You can, in fact, compress a file multiple times until a theoretical limit is reached. This limit is when there are no more (or only very short) patterns remaining.
- One problem of Morse Code is its ambiguity. If each letter isn't transmitted separately, then we might confuse a dot-dash letter with a dot letter followed by a dash letter. Thankfully, the algorithm which we'll be using—Huffman Coding—has the property of *immediate decodability*. This means that the code for any one character is never the prefix of the code for another character. More on that in a moment.
- The actual algorithm for Huffman Coding consists of 1) analyzing the frequency of characters in the text; 2) constructing a forest of childless trees, one for each unique character in the text; 3) joining the two trees with the smallest frequencies, creating a new parent with the sum of their frequencies; 4) assigning 0 and 1 to the left and right which lead from the new parent to each child; and 5) recursing from step 3.See these steps in action in these diagrams:

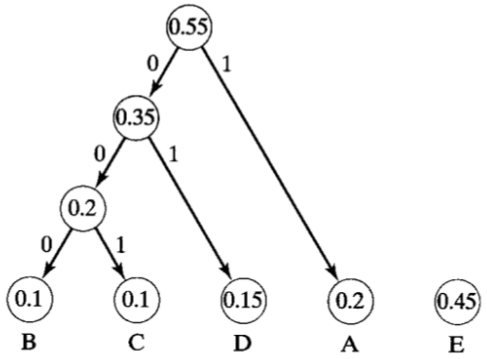
[5] If you've never heard of YouTube, just Google it. If you don't know what Google is, then ask someone via your Facebook status. Or Twitter about it. Or you could, like, go outside for a change.

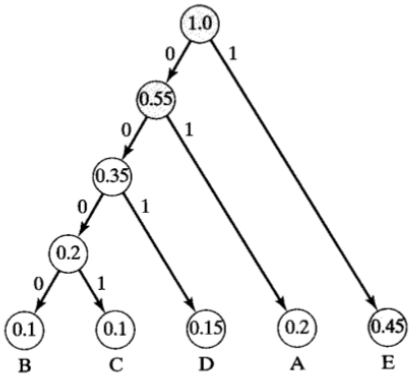
"ECEABEADCAEDEEEECEADEEEEEDBAAEABDBBAAEAAAC
DDCCEABEEDCBEEDEAEEEEEAEEDBCEBEEADEAEEDAEBC
DEDEAEEDCEEAEEE"

character	A	B	C	D	E
frequency	0.2	0.1	0.1	0.15	0.45









- So according to this tree, we have the following mappings:

  A 01
  B 0000
  C 0001
  D 001
  E 1

  Conveniently, the most frequently appearing character, E, is represented with the shortest bit sequence. And, toward the notion of immediate decodability, notice that none of the other characters begins with 1.

- How could we implement a tree node in C using a `struct`? We need some way of representing the frequency, which is displayed in each circle in the above diagrams. And, as with the nodes of a linked list, we need pointers to point to the next nodes. In the case of a binary tree, we'll need two pointers, one to the left child and one to the right child. So we might write this:

```
typedef struct node
{
    char symbol;
    int frequency;
    struct node *left;
    struct node *right;
}
node;
```

7 Geek Humor (76:00–80:00)

- We leave you with this.