Computer Science 50
Fall 2010
Scribe Notes

Week 7 Wednesday: October 20, 2010
Andrew Sellergren

Contents

Computer Science 50
Fall 2010
Scribe Notes

Week 7 Wednesday: October 20, 2010
Andrew Sellergren

1 Announcements and Demos (0:00–19:00)

- This is CS50.
- To the end of helping you with your preterm planning, we're releasing new features and bug fixes for HarvardCourses today! First, we made it easier to clear your selection for a faculty member. Second, we made it more intuitive, via dropdown menus in the upper left, to narrow down courses by specific fields. Third, we ripped out the Facebook comment feature because no one was using it. Fourth, we made it possible to add courses to different lists, e.g. Courses I'm Taking, Courses I'm Shopping. Fifth, we added a Random Suggestions box on the lefthand side that populates with courses based on your previous selections and other students' selections. Finally, we're looking to add the ability to search for all courses that count toward a given concentration. However, because this data doesn't exist in any central database but rather must be cleaned from the course catalog, we're hoping that you can help us out by solving this problem with crowdsourcing! If you'd like to pick off a given concentration, e-mail David and he'll share a Google doc with you.
- If you're looking for ideas for your final project, check out ideas.cs50.net which contains a whole slew of user submissions. Keep in mind that the course wiki also has descriptions of various APIs (Application Programming Interfaces) that allow you to work with code from CS50 apps such as HarvardCourses. Why not tackle the Android course catalog app or HarvardTravel, a website to connect students who have traveled or studied abroad? How about a program to download all Facebook photos in which you're tagged? Or a website to reserve practice rooms from FDO?
- We're also happy to announce that we'll be making a lot of changes to our website and our apps based on your feedback from Problem Set 4. It's interesting to read about all your gripes regarding technology that you've seen at banks, convenience stores, and within Harvard itself.

2 From Last Time (19:00–38:00)

- Bitwise operators allow us to manipulate data on the bit level. The AND operator (`&`) returns 1 if both bit operands are 1. The OR operator (`|`) returns 1 if either one of the bit operands is 1. The XOR operator (`^`) returns 1 if and only if **exactly** one of the bit operands is 1.
- The XOR operator is useful in implementing RAID arrays in which multiple hard drives store data. In an array of three drives, one of the drives stores the result of XOR'ing the bits of the other two drives so that the total capacity of the array is equal to the size of those two drives combined and if any one of the drives fails, the missing data can be quickly rebuilt.

2.1 swap2.c

- In fact, XOR can be used to swap the values of two variables without using any temporary storage, as `swap2.c` demonstrates:

```
/**************************************************************************
 * swap2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Swaps two variables' values.
 *
 * Demonstrates (clever use of) bitwise operators.
 **************************************************************************/

#include <stdio.h>


// function prototype
void swap(int *a, int *b);


int
main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %d\n", x);
    printf("y is %d\n", y);
    printf("Swapping...\n");
    swap(&x, &y);
    printf("Swapped!\n");
    printf("x is %d\n", x);
    printf("y is %d\n", y);
}


/*
 * Swap arguments' values.
 */

void
swap(int *a, int *b)
{
```

```
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}
```

2.2 Problem Set 5

- For Problem Set 5, you'll be asked to recover a series of JPEGs from a flash drive that has been formatted. To do so, you'll leverage the fact that JPEGs are stored on disk with the same 4-byte sequence as a header. When you scan through the data that we've given you, you'll write out a new JPEG file each time you encounter that 4-byte sequence.
- One subtlety that we'll mention briefly now is the concept of Endianness. Although we tend to think of numbers as reading from left to right, they may be stored on disk in different directions. Memory addresses on Big Endian architectures read from left to right. Memory addresses on Little Endian architectures read from right to left.

2.3 `pointers1.c`

- `pointers1.c` takes a string as input and iterates over all of the characters in it, printing them out one per line:

```
/**************************************************************************
 * pointers1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a string, one character per line.
 *
 * Demonstrates strings as arrays.
 **************************************************************************/

#include <cs50.h>
#include <stdio.h>
#include <string.h>


int
main(void)
{
    // prompt user for string
    printf("String please: ");
    char *s = GetString();
    if (s == NULL)
```

```
        return 1;

    // print string, one character per line
    for (int i = 0, n = strlen(s); i < n; i++)
        printf("%c\n", s[i]);

    // free string
    free(s);

    return 0;
}
```

The only thing "new" worth mentioning is the call to `free`. Now that we know that `GetString` calls `malloc`, we need to make sure to explicitly return that memory to the operating system before the program closes.

2.4 pointers2.c

- More interestingly, because we know that strings are actually implemented as pointers under the hood, we can access the characters in the string using pointer arithmetic instead of bracket notation:

```
/**************************************************************************
 * pointers2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Prints a string, one character per line.
 *
 * Demonstrates pointer arithmetic.
 **************************************************************************/

#include <cs50.h>
#include <stdio.h>
#include <string.h>


int
main(void)
{
    // prompt user for string
    printf("String please: ");
    char *s = GetString();
    if (s == NULL)
        return 1;
```

```
    // print string, one character per line
    for (int i = 0, n = strlen(s); i < n; i++)
        printf("%c\n", *(s+i));

    // free string
    free(s);

    return 0;
}
```

This program's output is identical to `pointers1.c`, but it achieves it in a slightly different way. Recall that `*s` will access the memory that stores the first character of `s`. If we write `*(s+1)`, we're accessing the memory directly next to the first character, that of the second character. Because each character is a single byte, `s+1` represents the address of the second character, `s+2` represents the address of the third character, and so on. In fact, the compiler is smart enough to do this arithmetic even when the pointer points to an array of elements each of which is larger than a single byte. For example, `*(s+1)` will access the second element in an array of integers named `s`.

- Because it allows low-level access to memory, C is one of the few programming languages that offers pointer arithmetic. However, generally you'll find that the only difference between many programming languages is their syntax. Once you've learned one or two, as you will in this course, you'll be empowered to learn any number of others.

2.5 Stacks

- Hopefully to help clear up some confusion from Monday, here is the final definition of a struct that implements a stack:

```
typedef struct
{
    int numbers[CAPACITY];
    int size;
    int top;
}
stack;
```

Although a stack of infinite size is theoretically possible, a stack of fixed size is more practical. Thus, we can implement it using an array. `CAPACITY` is the maximum size of the stack, defined as a constant elsewhere. `size` is a piece of metadata that keeps track of how many values are currently on the stack. `top` stores the index of the value that's next to be popped off

the stack. We store this because this index won't necessarily be 0 if other values have already been popped off.

- We need both `size` and `top` because of a single corner case. In almost all cases, `size` will be the value of `top` plus one. However, if there are no values in the stack, then `top` will be zero and `size` will be zero. Alternatively, we could fix this by returning some sentinel value when the stack was empty.

2.6 Queues

- The struct to implement a queue is very similar to that to implement a stack:

```
typedef struct
{
    int head;
    int numbers[CAPACITY];
    int size;
}
queue;
```

Here, `head` keeps track of the value that was inserted first which will be the first to be removed. `head` mostly likely starts as 0, but as values are removed, it becomes 1, 2, 3, 4, and so on. In this way, we don't have to shift all the elements of the array, which is expensive. If we wanted to maximize our use of space, we might even wrap the queue around so that when we run out of space at the end of the array, we can fill the spaces at the beginning. In that case, we'd also need to keep track of the tail of the queue.

3 Hash Tables (38:00–64:00)

- To create a dictionary of words with which to look up misspellings, you certainly could use an array sorted alphabetically in combination with binary search. Binary search is in $O(\log n)$ which is theoretically fast, but not the absolute fastest. What would be even faster than binary search is something in $O(1)$, implying constant-time lookup.

- One data structure that affords constant-time lookup is a *hash table*. A hash table can actually be implemented as an array, albeit one that is larger than it needs to be to store all of your data. To insert data into a hash table, you will effectively throw the input at the hash table like a dart. Your hope is that it will stick to the hash table without "colliding" with any other inputs. That is, your hope is that you will find an empty element in the array. If you happen to find an empty element, then it only takes a single step to insert this input into your hash table.

- There's one qualification we need to make, however. We're not actually throwing the input at the hash table randomly. If we did, how would we know where to find it later? We need to insert the input in such a way that we can find it quickly in the future. More on this in a moment.

- If we did insert inputs randomly into the hash table, what would be the probability that we would collide with another input, i.e. we would hit a non-empty element in the array? Surprisingly high, it turns out. We can rephrase this problem as an instance of the birthday paradox: in a room of n CS50 students, what's the probability that at least 2 students have the same birthday? Here, the hash table has a size of 365 for the number of days in the year. Let's assume an even distribution of birthdays throughout the year. We can better answer this problem if we consider the opposite question: what's the probability that no 2 students have the same birthday in a room of n students?

  $$\bar{p}(n) = 1 \times \left(1 - \tfrac{1}{365}\right) \times \left(1 - \tfrac{2}{365}\right) \times \cdots \times \left(1 - \tfrac{n-1}{365}\right)$$

  The probability that no 2 students have the same birthday is 1 when n is 1. The probability that no 2 students have the same birthday is $1 \times \frac{364}{365}$ when n is 2 because there are 364 possibilities for the second person's birthday that don't collide with the first person's. The probability that no 3 students have the same birthday is $1 \times \frac{364}{365} \times \frac{363}{365}$. And so on. This series actually reaches high percentages fairly quickly. For example, when n is 40, the probability of a collision is almost 90%.

- The act of "throwing a dart" at our hash table to find where to put a value is called *hashing*. This is really just implemented as a function like so:

  ```
  int
  hash(int n)
  {
      return rand();
  }
  ```

  This will return us a random number which corresponds to an index in our hash table. Of course, we'd actually need to normalize this number so that it wasn't larger than the size of our hash table. But we won't bother doing that because we won't really be using this as our hash function anyway. As we said before, it doesn't really make sense to have a hash function that returns random numbers because we won't be able to look up numbers after we've inserted them.

- Just as simple is the following hash function:

```
int
hash(int n)
{
    return n;
}
```

If we want to store a number in a hash table, we can simply use the number itself as an index into the hash table. But we'll run into problems when the number to insert is larger than the size of the hash table. We can fix this using the modulus operator:

```
int
hash(int n)
{
    return n % 26;
}
```

Here, we're assuming that the hash table is of size 26 (perhaps for the number of letters in the alphabet). If our number n is larger than 26, it will be wrapped around to return a valid index into our hash table.

3.1 Linear Probing

- Given that collisions are likely to occur even if our hash table is several times as large as the number of values we're inputting (e.g. 365 versus 40 in the case above), we need to decide what do to when we have a collision. We could simply place the value in the bucket directly next to the one we originally intended to insert it in. This is called linear probing. If `hash` returned index 0 and we found that index 0 in our hash table already had a value in it, we could look at index 1.
- The linear probing approach for dealing with collision introduces problems with search, however. If the hash function returns index 0 but we don't find the value we're searching for at index 0, do we immediately return false? No, in fact, because the value we're searching for might be at index 1, or index 2, or worst case index 25. As you can see, searching a hash table that uses linear probing for collisions is in $O(n)$ which is not what we want at all.
- If the values we want to store in our hash table are strings rather than numbers, we need to rewrite our hash function. Because all we need is an index for each string, we could easily use the first character of the string cast to an integer like so:

```
int
hash(string s)
{
```

```
    return (int) s[0] - 'A';
}
```

We need to subtract the value of A in order to normalize the value of the letter to a number between 0 and 25 (assuming again that our hash table is of size 26). This hash function also assumes that its input will consist of all capital letters.

- Given that there aren't many words, names, or strings that start with Z but there are many that start with A, this hash function is going to produce a lot of collisions. The ideal hash function, it would seem, is one that uses resources as efficiently as possible, minimizing collisions without making search expensive.

3.2 Separate Chaining

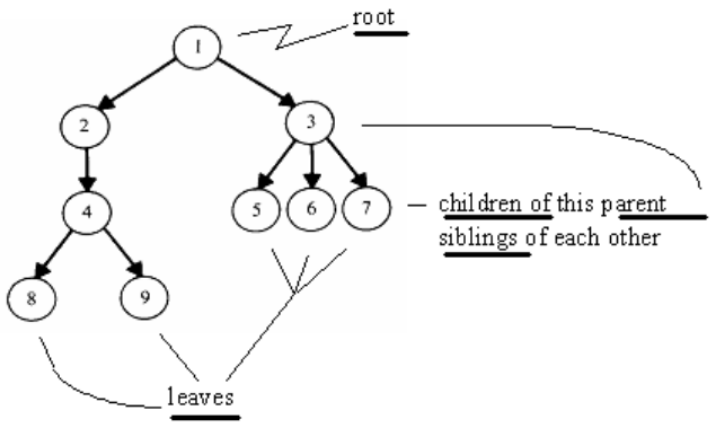
- Clearly linear probing wasn't a feasible solution to the problem of collisions given that it results in $O(n)$. A better approach is *separate chaining* which implements each element of the hash table as a linked list. When the hash function returns an index in the hash table which isn't empty, the value to be inserted becomes the head of a linked list at that index.
- Searching for a value in a hash table that uses separate chaining is in $O(n/k)$ where k is the size of the hash table and n is the number of values it stores. If we assume that our hash function achieves perfectly uniform distribution, then we'll end up with a linked list of length n/k at each index. If in the worst case, the value we're searching for is at the end of one of these linked lists, then we'll have to execute n/k steps in order to find it.
- Theoretically, $O(n/k)$ is actually the same as $O(n)$ because in the worst case, every input into the hash table causes a collision and thus our hash table is just one linked list of length n. However, in practice, this isn't the case, and our runtime will be faster.
- Separate chaining also solves another problem induced by linear probing: our hash table can now be smaller than the total number of values. In fact, there is no theoretical limit on the number of values that can be stored in a hash table since we can simply keep adding them as the heads of our linked lists. In practice, however, searching for a value will get faster as our hash table grows in size (assuming our hash function gives us a fairly even distribution).
- Incidentally, "node" and "bucket" are terms that computer scientists use to refer to a generic container for data. You'll hear them thrown around pretty liberally.
- To store strings in a hash table using separate chaining, we'll need to define a node of a linked list:

```
typedef struct node
{
    char word[LENGTH + 1];
    struct node *next;
}
node;
```

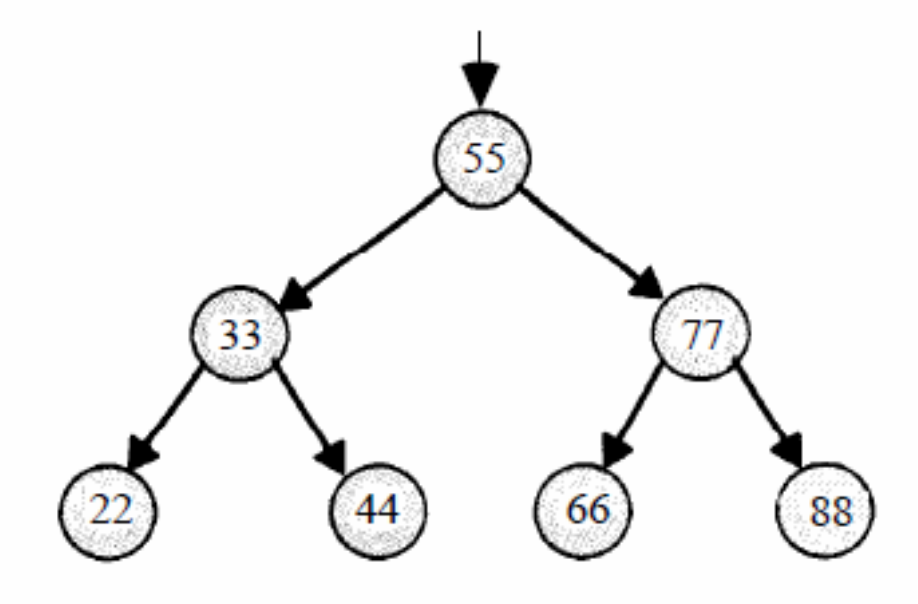
`LENGTH` is a constant which we'll use to represent the maximal length of a word that we want to store, plus one for the null terminator. Inside a node, we also need a pointer that will eventually point to the next node. In this implementation, each index in our array will actually be a pointer to the first node in the linked list that lives there.

4 Trees and Tries (64:00–75:00)

- Trees are data structures consisting of nodes, each of which may have any number of children. If each node 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.



- A special kind of tree called a binary search tree allows us to use binary search to do lookups:



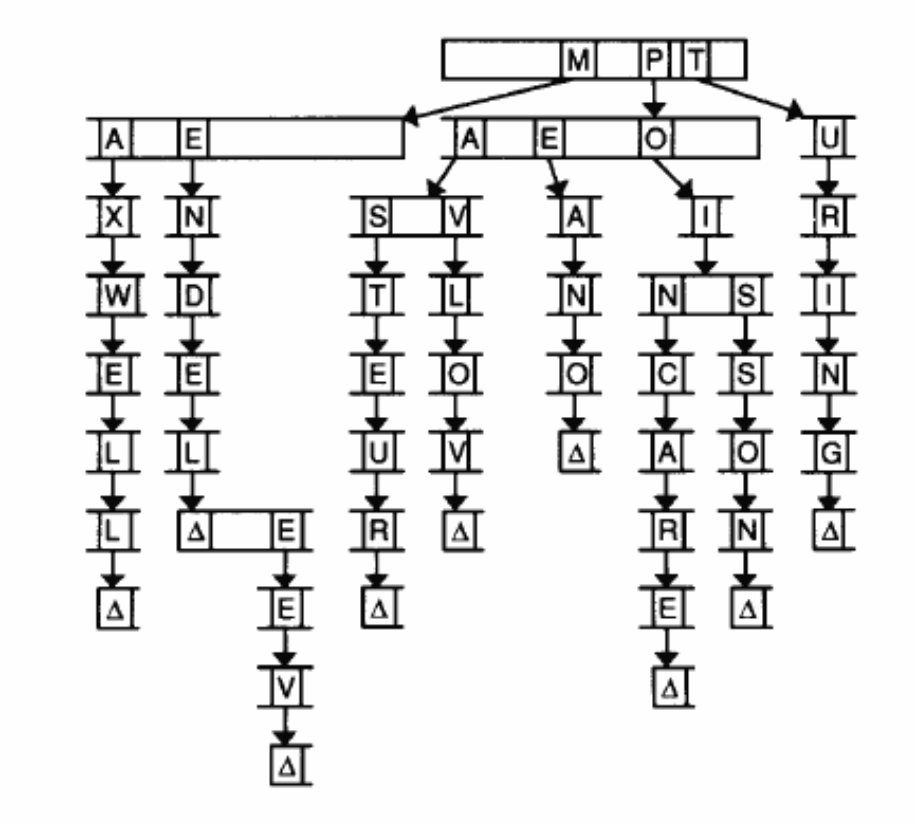
Binary search trees are specially structured so that each parent node is greater than its left child but less than its right child. If we were searching for the number 44 in the above tree, we would first check the root node, 55. Because 44 is less than 55, we would go left. Because 44 is greater than 33 we would then go right and reach the desired result.

- A tree is similar to a linked list in that one need only store a pointer to the root node.
- The node containing 33 is the child of the node containing 55, but it is also the root of a smaller tree consisting of the nodes containing 22 and 44. As you might have guessed, trees are good candidates for recursion because of these repeated relationships.
- Each node of a binary search tree might be implemented as follows:

```
typedef struct node
{
    int n;
    struct node *left;
    struct node *right;
}
node;
```

Here, we have space in `n` for the actual value that the node stores as well as two pointers that point to the left and right child nodes.

- Tries are another data structure which might prove useful to you as you complete Problem Set 6. A trie is a type of 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. Take a look at this visualization of a trie used to store names in which each array is of size 26 for the number of letters in the alphabet:



We walk through a trie much the same way we walk through a hash table with separate chaining. Each letter in the word we're inserting (converted to a number between 0 and 25) is also its index into the next level of the trie. So if we're inserting the name Maxwell, we first hash to M, then to A, then to X, etc. What happens when we get to the end of a word? We need some sort of flag (represented as a triangle in the diagram above) that marks the end of a word. That way, if two words share a prefix (e.g. Max and Maxwell), we will know that both of them are in our trie if this end-of-word flag is set at both the X and the last L.

- What's interesting about a trie is that it never actually stores any letters or words, but only pointers. The words are implicit.

- The compelling case for tries is that the running time of search is $O(m)$, where m is the length of the longest word, and is thus independent of n, the number of words in the trie. Woohoo, constant-time lookup! Go forth and solve Problem Set 6!