Hacking a Google Interview – Handout 3

Course Description

Instructors: Bill Jacobs and Curtis Fonger
Time: January 12 – 15, 5:00 – 6:30 PM in 32-124
Website: http://courses.csail.mit.edu/iap/interview

Question: Deck Shuffling

Given an array of distinct integers, give an algorithm to randomly reorder the integers so that each possible reordering is equally likely. In other words, given a deck of cards, how can you shuffle them such that any permutation of cards is equally likely?

Good answer: Go through the elements in order, swapping each element with a random element in the array that does not appear earlier than the element. This takes O(n) time.

Note that there are several possible solutions to this problem, as well as several good-looking answers that are incorrect. For example, a slight modification to the above algorithm whereby one switches each element with any element in the array does not give each reordering with equally probability. The answer given here is, in our opinion, the best solution. If you want to see other solutions, check the "Shuffling" page on Wikipedia.

Binary Search Trees

A binary search tree is a data structure that keeps items in sorted order. It consists of a binary tree. Each node has a pointer to two children (one or both of which may be null), an optional pointer to its parent (which may be null), and one element that is being stored in the tree (perhaps a string or an integer). For a binary search tree to be valid, each node's element must be greater than every element in its left subtree and less than every element in its right subtree. For example, a binary tree might look like this:

```
               17
           /        \
        6             46
      /   \              \
   3       12             56
 /       /    \          /
1       9        15    48
```

To check whether an element appears in a binary search tree, one need only follow the appropriate links from parent to child. For example, if we want to search for 15 in the above tree, we start at the root, 17. Since 15 < 17, we move to the left child, 6. Since 15 > 6, we move to the right child, 12. Since 15 > 12, we move to the right child again, 15. Now we have found the number we were looking for, so we're done.

To add an element to a binary search tree, we begin as if we were searching for the element, following the appropriate links from parent to child. When the desired child is null, we add the element as a new child node. For example, if we were to add 14 to the above tree, we would go down the tree. Once we reached 15, we would see that the node has no left child, so we would add 14 as a left child.

To remove an element from a binary search tree, we first find the node containing that element. If the node has zero children, we simply remove it. If it has one child, we replace the node with its child. If it has two children, we identify the next-smaller or next-larger element in the tree (it doesn't matter which), using an algorithm which we do not describe here for the sake of brevity. We set the element stored in the node to this value. Then, we splice the node that contained the value from the tree. This will be relatively easy, since the node will have at most one child. For example, to remove 6 from the tree, we first change the node to have the value 3. Then, we remove the node that used to have the value 3, and we make 1 the left child of the node that used to have the value 6.

A small modification to a binary search tree allows it to be used to associate keys with values, as in a hash table. Instead of storing a single value in each node, one could store a key-value pair in each node. The tree would be ordered based on the nodes' keys.

Interviewers sometimes ask about binary search trees. In addition, binary search trees are often useful as a component of an answer to interview questions. The important thing to remember is that insertion, removal, and lookup take O(log n) time (where n is the number of elements in the tree), since the height of a well-balanced binary search tree is O(log n). Although in the worst case, a binary search tree might have a height of O(n), there are "self-balancing" binary search trees that periodically reorganize a BST to ensure a height of O(log n). Many self-balancing BST's guarantee that operations take O(log n) time. If you want to learn more about particular types binary search trees, such as red-black trees, we recommend looking them up.

Question: Path Between Nodes in a Binary Tree

Design an algorithm to find a path from one node in a binary tree to another.

Good Answer: There will always be exactly one path: from the starting node to the lowest common ancestor of the nodes to the second node. The goal is to identify the lowest common ancestor.

For each node, keep track of a set of nodes in the binary tree (using a hash table or a BST) as well as a current node. At each iteration, for each of the two current nodes, change the current node to be its parent and add it to the appropriate set. The first element that is added to one set when it is already present in the other set is the lowest common ancestor. This algorithm takes O(n) time, where n is the length of the path. For example, if we were finding the lowest common ancestor of 3 and 15 in the above tree, our algorithm would do the following:

Current node 1	Current node 2	Set 1	Set 2
3	15	3	15
6	12	3, 6	15, 12
17	6	3, 6, 17	15, 12, 6

To improve the solution, we actually only need to use one set instead of two.

Bitwise Operations

Integers are represented in a computer using base two, using only 0's and 1's. Each place in a binary number represents a power of two. The rightmost bit corresponds to 2^0, the second digit from the right corresponds to 2^1, and so on. For example, the number 11000101 in binary is equal to 2^0 + 2^2 + 2^6 + 2^7 = 197. Negative integers can also be represented in binary; look up "two's complement" on Wikipedia for more details.

There are a few operations that a computer can perform quickly on one or two integers. The first is "bitwise and", which takes two integers and returns an integer that has a 1 only in places where both of the inputs had a 1. For example:

```
  00101011
& 10110010
----------
  00100010
```

Another operation is "bitwise or", which takes two integers and returns an integer that has a 0 only in places where both of the inputs had a 0. For example:

```
  00101011
| 10110010
----------
  10111011
```

"Bitwise xor" has a 1 in each place where the bits in the two integers is different. For example:

```
  00101011
^ 10110010
----------
  10011001
```

"Bitwise negation" takes a number and inverts each of the bits. For example:

```
~ 00101011
----------
  11010100
```

"Left shifting" takes a binary number, adds a certain number of zeros to the end, and removes the same number of bits from the beginning. For example, 00101011 << 4 is equal to 10110000. Likewise, right shifting takes a binary number, adds a certain number of zeros to the beginning, and removes the same number of bits from the end. For instance, 00101011 >>> 4 = 00000010. Actually, there is a more common form of right shifting (the "arithmetic right shift") that replaces the first few bits with a copy of the first bit instead of with zeros. For example, 10110010 >> 4 = 11111011.

Interviewers like to ask questions related to bitwise logic. Often, there is a tricky way to solve these problems. Bitwise xor can often be used in a tricky way because two identical numbers in an expression involving xor will "cancel out". For example, 15 ^ 12 ^ 15 = 12.

Question: Compute 2^x

How can you quickly compute 2^x?

Good answer: 1 << x (1 left-shifted by x)

Question: Is Power of 2

How can you quickly determine whether a number is a power of 2?

Good answer: Check whether x & (x - 1) is 0. If x is not an even power of 2, the highest position of x with a 1 will also have a 1 in x - 1; otherwise, x will be 100...0 and x - 1 will be 011...1; and'ing them together will return 0.

Question: Beating the Stock Market

Say you have an array for which the ith element is the price of a given stock on day i. If you were only permitted to buy one share of the stock and sell one share of the stock, design an algorithm to find the best times to buy and sell.

Good answer: Go through the array in order, keeping track of the lowest stock price and the best deal you've seen so far. Whenever the current stock price minus the current lowest stock price is better than the current best deal, update the best deal to this new value.

Program Design

Although it sometimes may seem like it, interviewers aren't always interested in little programming tricks and puzzles. Sometimes they may ask you to design a program or a system. For example, it's not uncommon to be asked to sketch out what classes you would need if you were to write a poker game program or a simulation of car traffic at an intersection. There are many different questions the interviewer could ask about design, so just keep in mind that if you need to design the classes for a program, try to keep your design simple and at the same time allow for future extensions on your design.

For example, if you were designing a five-card draw poker game program, you could have a GameMode interface or superclass and have a FiveCardDraw subclass to encapsulate the particular rules for that version of the game. Therefore if the interviewer then asks you how you could extend the system to allow a Texas hold 'em game, you could simply again make a TexasHoldEm subclass of GameMode.

Design Patterns

A design pattern is a useful design technique that programmers have discovered to be so useful over the years that they give a specific name to it. Interviewers sometimes ask you to list some design patters you know, and may even ask you to describe how a particular one works. But besides questions that directly test your knowledge of them, design patters are very useful as building blocks for solving other questions, especially the ones that focus on program design. There are several design patterns that exist, and we recommend that you take a look at the "Design Pattern" page on Wikipedia to get an idea of several of the best-know ones, but there are a few that truly stand out in their popularity and usefulness.

Listener/Observer Pattern:

This may be the most popular design pattern out there. The idea is this: suppose there were an e-mail list for Hacking a Google Interview (unfortunately there isn't one, but if we had been a bit more forward-thinking, perhaps we would have made one). This list would allow for important announcements to be sent to anyone who cared about the class. Every student who put themselves on the list would be a "listener" (or "observer"). The teacher would be the "announcer" (or "subject" in some texts). Every time the teacher wanted to let the students know something, they would go though the e-mail list and send an announcement e-mail to each listener.

In a program, we would have a Listener interface that any class could implement. That Listener interface would have some sort of "update()" method that would be called whenever the announcer wanted to tell the listeners something. The announcer would store a list of all the listeners. If we wanted to add an object "foo" as a listener, we would call "announcer.addListener(foo)", which would cause the announcer to add foo to its list of listeners. Whenever the announcer did something important that it wanted to tell the listeners about, it would go though its list of listeners and call "update()" on each of those objects.

Going back to the poker game program, you might mention to the interviewer that you could use the listener design pattern for several things. For example, you could have the GUI be a listener to several objects in the game (such as player hands, the pot, etc.) for any changes in game state for which it would need to update the display.

Singleton Pattern:

The singleton pattern is used when you want to make sure there is exactly one instance of something in your program. If you were making a Lord of the Rings game, you would want to make sure that the One Ring was only instantiated once! We have to give Frodo a chance!

In Java, for instance, to make sure there is only one of something, you can do something like this:

```
public class OnlyOneOfMe {
    private static OnlyOneOfMe theOneInstance = null;

    private OnlyOneOfMe() {
        // do stuff to make the object
    }

    public static OnlyOneOfMe getInstance() {
        if (theOneInstance == null) {
            theOneInstance = new OnlyOneOfMe();
        }
        return theOneInstance;
    }
}
```

Notice that there is no public constructor. If you want an instance of this class, you have to call "getInstance()", which ensures that only one instance of the class is ever made.

Model-View-Controller:

Model-view-controller (MVC) is a design pattern commonly used in user interfaces. Its goal is to keep the "data" separate from the user interface. For example, when designing a program that displays stock information, the code that downloads the stock information should not depend on the code that displays the information.

The exact meaning of model-view-controller is a bit ambiguous. Essentially, a program that uses model-view-controller uses separate programming entities to store the data (the "model"), to display the data (the "view"), and to modify the data (the "controller"). In model-view-controller, the view usually makes heavy use of listeners to listen to changes and events in the model or the controller.

Model-view-controller is a good buzzword to whip out when you're asked a design question relating to a user interface.

Classic Question #7: Ransom Note

Let's say you've just kidnapped Alyssa Hacker you want to leave a ransom note for Ben Bitdiddle, saying that if he ever wants to see her again, he needs to swear to never use Scheme again. You don't want to write the note by hand, since they would be able to trace your handwriting. You're standing in Alyssa's apartment and you see a million computer magazines. You need to write your note by cutting letters out of the magazines and pasting them together to form a letter. Here's the question: given an arbitrary ransom note string and another string containing all the contents of all the magazines, write a function that will return true if the ransom note can be made from the magazines; otherwise, it will return false. Remember, every letter found in the magazine string can only be used once in your ransom note.

For example, if the ransom note string was "no scheme" and the magazine string was "programming interviews are weird", you would return false since you can't form the first string by grabbing and rearranging letters from the second string.

Pretty-good answer: Make a data structure to store a count of each letter in the magazine string. If you're programming in C, you can make an array of length 256 and simply use the ASCII value for each character as its spot in the array, since characters are 1 byte. If you're programming in Java, you can just use a hash table instead (since characters are 2 bytes in Java). Then go through the magazine string and for each character, increment the value for that letter in your data structure. After you go though the whole magazine string, you should have an exact count of how many times each character appears in the magazine string. Then go through each character in the ransom note string and for every character you encounter, decrement the value for that letter in your data structure. If you ever find that after you decrement a count to something less than 0, you know you can't make the ransom note, so you immediately return false. If however you get though the entire ransom note without running out of available letters, you return true.

Even better answer: Because the magazine string may be very large, we want to reduce the time we spend going through the magazine string. We use the same idea as above, except we go through the ransom note and the magazine string at the same time. Keep one pointer for our current character in the ransom note and another pointer for our current character in our magazine string. First, check to see if the count in our data structure for our current ransom note character is greater than 0. If it is, decrement it and advance the pointer in our ransom note. If it isn't, start going through the characters in the magazine string, updating the counts in the data structure for each character encountered, until we reach the character we need for our ransom note. Then stop advancing the magazine string pointer and start advancing the ransom note pointer again. If we get to the end of the ransom note, we return true. If we get to the end of the magazine string (meaning we didn't find enough letters for our ransom note), we return false.