ENGINEER A:
Study up on data structures and algorithms, working on sample problems (TopCoder , etc...), and practice with writing code on a whiteboard.

The point of interviews is that we're trying to see how you solve problems more than anything else, and that your knowledge of data structures and big-O notation would be tested. (Like why you would use one data structure over another or the running time of using a specific data structure or the general running time of your algorithm.) **these are websites with questions about the same scope/complexity of Google interview questions: leetcode.com, google-interview.com**

ENGINEER B:
(1) Make sure you treat each practice question while studying as if it was a real interview question: read the question, close the book / walk away from the computer), talk through the problem aloud, and write a solution on a whiteboard (or paper if you don't have a whiteboard).
(2) (FOR ONSITE INTERVIEWS) Make sure to reset yourself after each individual interview. Don't worry about your prior performance. Just stay calm, put a smile on your face, and do your best in the next interview.
(3) The interview is supposed to be a conversation. Be certain to talk about your ideas and answers.
(4) When given an interview question, (i) make sure you completely and fully understand the question, (b) write down an initial set of edge cases, (c) verbalize every solution or idea you have, and (d) code up a solution on the whiteboard, and (e) go through all your edge cases.

ENGINEER C:
Get comfortable writing code on a whiteboard, sharpen up the algorithm and data structures in your toolkit, be ready for open ended questions, ask questions when you need more context.
Review data structures and algorithms, and to talk about thinking process during the interview.

ENGINEER D:
Practice problems and try to think about and express the Google scale implications of decisions. For example, when designing Google Suggest, speed is very important, so if your solution starts at making a server request on each keypress, can you find a way to improve it into a solution that is fast on the back and frontend.
Think about things like internationalization or prioritizing data relevant to the logged in user.

ENGINEER E:
Practice speaking out loud while coding, practice writing on a whiteboard, know algorithms (like be able to find big-O of anything he writes on the board, recursive vs. iterative, common algorithms), data structures (like knowing pros/cons of stacks/queues/trees/graphs/hashes). Know one language well enough to code comfortably (his is C#). I also told him to expect testing questions.

ENGINEER F:
Basically exactly what's in this YouTube video: **http://www.youtube.com/watch?v=oWbUtlUhwa8**
- Make sure you clarify the problem so you're answering the same thing
- Ask for sample input / output if relevant
- Explain everything you're doing (which algorithm you're choosing, the code you're writing, etc) and why
- Run through the sample input again very quickly and verify it gives sample output

ENGINEER G:
Practice writing code by hand, get sharp with data structure and algorithms (by writing real code, not just reading the theoretical stuff), watch lectures online **(coursera.com),** try some of the available problems on TopCoder.

ENGINEER H:
Check out TopCoder archives and maybe spend some time checkout out online lectures one Coursera.

ENGINEER I:
- Definitely watch this YT video: http://www.youtube.com/watch?v=oWbUtlUhwa8
- Explain your every thought and why you're doing things the way you are
- Practice writing on a whiteboard while also explaining something to a person behind you
- Review: data structures (and access times, why you would use one over the other, etc); algorithms (tree traversal, recursion, string manipulation, etc)
- Mentioned sample problems (find most common character in a string, reverse words of a string, math problems analogous to fibonacci, etc)

ENGINEER J:
Speak out loud and describe your process - your interviewer will only know that you're thinking if you're vocal. If you walk through a suboptimal solution in your head and never vocalize it, the interviewer won't be able to assess how to help you with hints. **Like in the real world, requirements are not necessarily well defined so your questions may also need you to ask questions to figure out exactly what's going on.**

ENGINEER K:
Q: Which shoes should I wear? (For Onsite)
A: Wear whatever makes you more comfortable. Some people wear tshirt, shorts and flip flops, others are in full suit; so wear whatever makes you happy.

Q: What is the scope of the interview questions?
A: Kinda everything. Lots of google interview q's involve writing a function for string manipulation (e.g., reverse all words in a string, find the most common character in a string), mathematical problems (e.g., print the first 1000 prime numbers, print the first 1000 fibonnaci numbers), or find this property of a tree. So it's good to study your data structures (hashmap, stack, list, queue, etc) - know their access times/strengths/weaknesses; study algorithms (e.g., tree traversal algorithms) and running times. You may also get a design question (e.g., how would you design ms paint; what classes would you have, how would things interact, etc).

Q: What are the possible languages I'll be asked to program in.
A: Your choice really; program in whatever you're most comfortable with. If that's C, great. If that's python, fine. You can also choose to answer in python if properties of the language make writing a solution easier and vice versa.

I suggest the following websites:
- The youtube video on preparing for your technical interview from 2 Google engineers
- google-interview.com - Get a sense of people's actual interviews
- leetcode.com - Practice questions that get progressively harder (much like google interview questions do)

ENGINEER L:
Take hints from your interviewer.

All interviews are challenging at google. No one is expected to do perfect on every aspect. Don't let it fluster you.

Not being combative, being flustered or stubborn is very important

Do the easiest solution first and then make your code better. Don't jump directly into a dynamic programming solution until you've done brute force first.

Communicate as you go.

ENGINEER M:
- Overview of interview questions (fairly academic -- define some function to compute blah. mentioned sample q's -- find most common character in a string, reverse words of a string, find first 1000 prime numbers)
- Talk through your solution and why you're doing what you're doing etc
- Practice writing on a whiteboard (or in a google doc) while also explaining your solution to someone else
- Know data structures and why to use one over the other and their access times. Know running time of your algorithms

ENGINEER N:
we're looking more for how you solve the problem and the entire thought process. gave example of the type of question google asks -- write a function to do X (e.g., find most common character in a string, reverse words of a string, compute the first N numbers with a particular property (e.g., prime numbers, fibonnaci numbers, etc). practice writing out a solution while explaining it. practice explaining everything you're doing and why. watch the google interview prep youtube video for more advice.

ENGINEER O:
Study, study, study. Practice with datastructures and algorithms. Read through java library code (collections, searches, sorts). Read up on distributed computing, data security, cryptography, etc...

ENGINEER P:
Generally, all computer science questions are fair game
Try to clarify the question and make sure you understand what approach you're taking before you start writing code. You should know what problem you're trying to solve.
As you work, talk through what you're thinking so your interviewer knows what you're thinking and so that you know that what you're thinking makes sense.

ENGINEER Q:
1. Ask questions of your interviewer to be certain that you fully understand the problem you are expected to address.
2. Communicate with the interviewer as you work through your solution. Don't go silent for long stretches of time.
3. Test your solution with sample data/paramaters
4. Discuss how your solutions scale
5. Discuss the time & size complexity of your solutions

ENGINEER R:
I focused primarily on practicing interviews with another person and on a whiteboard. I stressed how this would help him in many ways to organize his ideas, to speak as he thinks and to get feedback.

I also mentioned more general points like polishing and shortening his resume, focusing on accomplishments and measurable results, etc.

ENGINEER S:
Always talk about what you are thinking, even if it doesn't make sense, because it helps us to see how you think and also if you are not going in the right track to put her back in the right track.

to ask as many questions as you need, the interviewers help with resolving the problem and also is a good way to see what doubts come to her head and how she solves problems.

a very through list of things that might appear in your interview, you don't have to know all of them or study everything that is in there, but is a good thing to keep mind
http://steve-yegge.blogspot.com/2008/03/get-that-job-at-google.html

and practice practice practice, here are some places where you can find example questions and remember to talk (aloud) while you solve them

[1] "Five Essential Phone Screen Questions" by Steve Yegge
[2] StackOverflow pages, http://stackoverflow.com/search?q=google+interview
[2] Careercup, “Sample Google Interview Questions”
[4] TopCoder (http://www.topcoder.com) (about 10 division-2 competitions)

ENGINEER T:
1. Practice writing code by first creating it on a whiteboard, paper, or in a non-IDE text editor and then copy your code into an IDE and see if it compiles/runs as expected.
2. Ask questions of your interviewer to be certain that you fully understand the problem you are expected to address.
3. Communicate with the interviewer as you work through your solution. Don't go silent for long stretches of time.
4. Test your solutions with sample data/paramaters
5. Discuss how your solutions scale
6. Discuss the time & size complexity of your solutions
7. Approach the problem as if you were trying to solve it with a colleague at work, in which case it would be more of a discussion rather than a test.

ENGINEER U:
I suggest practicing with a friend to do a mock interview, writing code on paper/gdocs to prepare for the phone screening and speaking out loud your thought process as you solve a problem.

ENGINEER V:
always ask questions.
practice, practice, practice.
talk aloud while you practice so he gets used to the idea of walking people through his solutions.
interviewers are here to help

ENGINEER W:
- Prepare, prepare, prepare!
- Prepare on the board, on the computer
- Dont freeze, let the interviewer see your chain of thoughts
- Review your book about algorithms (for example Introduction to Algorithms by Cormen)
- Think about problems of scale (for example read the papers about google file system and big table). All found at research.google.com
Bigtable: A Distributed Storage System for Structured Data, The Google File System

ENGINEER X:
Brush up on complexity analysis and basic data structures and algorithms and practice writing code on a whiteboard. He was a little bit weak when it came to talking about concurrency so I also advised him to take a look at some fundamental concurrent programming concepts.

ENGINEER Y:
No “trick” questions or brain teasers will be asked during interview.
Approach the interview as a problem to be solved, instead of being nervous about passing the interview.

ENGINEER Z:
Practice writing code on paper.
Study the basics of algorithms and data structures.
Don't freak out.

ENGINEER A1:
Pick up the Algorithm Design Manual by Steve Skiena
Focus on Data Structures, Algorithms, and polishing your understanding of runtime complexity/Big-O
Don't be afraid to throw exceptions in the code if it's the right thing to do
Expect to hear "can you do it better/faster/cheaper"

ENGINEER B1:
Prepare to code in plain Text like Google docs.
Practice coding algorithm questions.

ENGINEER C1:
Take time to think through the question. Interview questions are typically under-specified so you should ask clarifying questions and make sure you understands the behavior intended by the interviewer with examples.

ENGINEER D1:
- Learn about code readability (suggested "Refactoring" book by Martin Fowler)
- Encourages to find opportunities to contribute to large codebases like an open source project.

ENGINEER E1:
Read "Cracking the Coding Interview" problems from projecteuler.net

ENGINEER F1:
Practice writing code in Google Docs

Speak out your thought process as much as possible during the interview. This will allow the interviewer to stay on top of what you're doing and offer hints.

The interview style is definitely going to depend on the interviewer. Some might be more laid back, others might be straight to the point and ask the coding question.

ENGINEER G1:
Here's an outline of what I suggest for ONSITE interviews:

-Read and do all/most Java problems in “Cracking the Coding Interview”!!! (minus the database questions)
-For an onsite interview, you’ll be writing code on the board. That’s much different from on a computer.
-Practice on a board or with pen and paper. You’ll need to be fast.
-Don’t use “i” and “j” as iterators. “i” looks like “j” and “j” looks like a semicolon “;”. Using “i” and “k” is fine. Let your interviewer know why you are doing this.
-The overall flow of the interview should be:
Interviewer may give some background on himself/herself. Should be at most 1-3 min. The interviewer wants to give you as much of the allotted time as possible to work on the problem.
Interviewer describes the problem.
You ask questions about what assumptions you can make, clarifying questions, etc. The question may be intentionally vague.
Draw a diagram/plan of what you will do. The interviewer should let you know if you made an incorrect assumption or if you’re way off.
-Start coding on the board.
-Interviewer will add complexity to the problem if there is time and/or prod you to debug your code.
-Talk as much as possible.
-The interviewer is not going to give you the answer, but they’re also not going to let you go way off course.
-Sometimes the problem’s structure can give you a clue as to what NOT to consider.
-If you need to take a minute to just stare at the board and think without talking, that’s ok.
-Saying what you know will help you out with the interviewer. Specifically it will convey a lot of your knowledge while spending little interview time.
-ex: “I would normally use StringBuilder, but in the interest of time, I’ll use the + syntax because it is quicker to write .”
-ex: “Under the hood, string concatenation with + uses a StringBuilder. So using + is ok in a single line.”
-ex: “The time complexity of resizing an array can be amortized over the total number of insertions.”
-ALWAYS look at time/space complexity. Ask if one is more important than the other. In the real world, there are tradeoffs. Usually speed is more important.
-In practice, use descriptive variable names, but on the board, shorter can be better.
-If you are familiar with a more complex data structure that fits a particular situation, mention it. Be ready for follow up questions.
-Abstract away parts of the problem that are be very simple
-ex: array resizing.

-ex: array sorting if it is not the focus of the problem. Just state the runtime and space usage of whatever algorithm you choose (probably QuickSort).
-You want to show your interviewer what you know. Spending time implementing a simple but lengthy operation is not optimal. Just describe a method that does what you need, and tell your interviewer that you'll assume you have an implementation and you'll get back to it if there's time at the end.
-ETC
-If using Java, proper use of an interface, generics, final/static are all a plus.
-Consider using auxiliary data structures (esp. if space is not a primary concern). ie you can maintain multiple data structures to achieve some functional or performance goal. Mention the space used.
-Sometimes it may make sense to sort your data even if that is not a requirement, to make your code faster.
-HashSets, HashMaps are go-to data structures because they are fast. LinkedLists tend to be slow for some basic operations.
-**Beware of the runtime costs of classes that come with Java. Factor in these costs in analyzing your algorithm.
-ex: Don't use Collections.sort() and forget that there is a time and space space performance cost.
-ex: If you use an ArrayList, don't forget that insert and remove are not cheap.
-Discussing this is always good. It is good to show that you know the implications of what's going on under the hood.

ENGINEER H1:
Know sorting algorithms, how they work, their time complexities and if there is an in-place version. Your best bet is probably Quicksort. Also know MergeSort, InsertionSort. BubbleSort is VERY slow and should almost never be used.

ENGINEER I1:
Take 1-2 months to work daily on interview questions (there are many available, online and in books), while also revisiting the fundamentals (also pointed him to some academic books on the topic of algorithms and data structures).

ENGINEER J1:
1. Make sure you are comfortable with basic data structures from the Collections framework, including the complexity of their methods.
2. Make sure you know how to implement a Hashtable, linked list, etc. from scratch if necessary.
3. If you find yourself implementing an algorithm that is worse than O(n) and you're not sure it has to be, call this out and try to see if you can do better.
4. Continue to talk through your thought processes.

ENGINEER K1:
Relax, ask questions, think out loud.
Be prepared for questions about scale, e.g. concurrency or very large data.