**Ace the coding interview, every time**

***Technical Preparation*** *Portions of the interviews will be extremely technical revolving around theory, algorithms, data structures, design patterns, languages, etc.****I can't stress enough the value of studying for the interview****as it can be very much like a CS exam, and its randomness around topics can catch folks by surprise.*

Why would any company do this, you ask? Because the fail/pass ratio for technical interviews is extraordinarily high (on average, **over 90% fail**), and companies want well-prepared candidates. Interviewing 100 people and hiring 10 is not an efficient use of anyone's time. Hiring takes a lot of time, and every company wants to do better.

OK. So with all of this material available on the Web and elsewhere, why are so many folks out there still bombing technical interviews? Simple. From my experience, and in speaking to countless colleagues in technology that are highly experienced interviewers, one thing is patently clear: Engineers of all types and educational backgrounds either passed or failed the tech screens and coding loops due to a simple root cause: **preparation or lack thereof.**So let's address and fix that, right now.

**Assumptions**

First and foremost, you already know how to code. You may or may not have a CS background - this is OK, but you're already a good coder, you just need a CS refresher, and a game plan for interview day.

This will cover the general coding interview blue-print. Candidates are, of course, expected to study and be prepared for technology specific questions around areas such as UX/UI development, embedded systems, mobile devices, etc.

**The Basics**

First, the good news. Virtually all software companies, especially the large ones, use the same interview process (in many cases to a great degree of similarity). We won't cover the process here, but these 3 blog posts cover what you should expect to encounter in 90% or more of your technical interviews at most companies:

1. ["Get that job at Google"](http://steve-yegge.blogspot.com/2008/03/get-that-job-at-google.html) - Steve Yegge. The original article, and by far the best reference on the topic
2. ["Get that job at Facebook"](https://www.facebook.com/notes/facebook-engineering/get-that-job-at-facebook/10150964382448920) - Carlos Bueno
3. ["Get that job at Microsoft"](http://www.booleanzen.com/blog/get-that-job-at-microsoft/) - Nick Ciubotariu (shameless plug)

If you only read one article, **read the first one.** Then read it again. It's must know for anyone who will go through a code-intensive technical interview, now or in the future.

Now, the irritating news. The articles are great, but also broad. That's a ton of material to cover. And they just tell you to study everything. And it gets worse. In Computer Science, there are an infinite amount of questions that can be asked. So we need specifics.

**What to Study**

First, the **absolute must-haves,** in order:

1. Trees (**especially** Binary Search Trees)
2. Trees (**especially** Binary Search Trees) - again
3. Big O Notation
4. Hash Tables
5. Object Oriented Design/System Design
6. Algorithms: Breadth First Search/Depth First Search, Binary Search, Merge Sort and Quick Sort

Let's dispose of Captain Obvious (Trees, and specifically, BST's). In every interview I've been through, with any team, product group or company, I've gotten a Binary Tree or BST question. **Every single one.** I have also been through interviews where every single one of the interviewers asked a BST question.

Speculating on reasons for this is neither here nor there. Personally, I posit that it's due to the versatility of Binary Search Trees - Insertion, Deletion and Search all take O(log n) time on average, and O(n) time worst case. They are very elegant data structures with lots of practical applications in Software Engineering, which is why, in any technical interview, you're going to see a BST-related question.

The rest of the study topics are self explanatory. Big O notation is absolutely necessary - you will be asked to determine time/space complexity in almost all your interview questions, and how to optimize your code for better time/space complexity. You are almost guaranteed to be given time/space complexity guidelines for designing your code. You are extremely likely to see a question where the solution will involve the use of hash tables. You are guaranteed to see at least one OO Design question or Systems Design question. And you are guaranteed at least one algorithmically driven question.

So what's left? Again, in order of priority:

1. Arrays
2. Recursion
3. Linked Lists
4. Stacks/Queues
5. Bit Manipulation

You will want to know recursion (remember the termination clause!!) and linked lists. Same for Stacks and Queues. Questions involving bit manipulation are rare but are encountered from time to time.

**Mock interview guidelines**

**Use a white board, or better yet, pencil and paper. Follow this rule, and you'll be successful. Violate it at your own peril.**

In an interview setting, you will not be given the advantage of using an IDE and compiler, so prepare in the most realistic way possible. Thus, write down the interview question in the same way that an interviewer would be providing it to you. Break it down into small parts and an approach to solve the problem. Write some pseudo-code, then write real code. Do all of this on paper, "walking through the code" as you write it. See if you can find bugs in the code (they will be there). Test your code. Pay attention to nullchecks and corner cases. Take your time and **really understand** the problem, before you move on to the next one.

Using this approach ensures you will learn and internalize the subject matter, rather than commit certain coding questions and solutions to rote memory. Run a coding question through an IDE only when you feel like you have given it your best shot and have exhausted the solution on paper. Writing the question on paper has an added advantage - you can review it later.

**Eat the elephant one bite at a time (how to study)**

The topic of computer science is, by its very own nature, very dense, and the subject matter can be hard to digest. Pick a particular area, increasing the difficulty as you go, and try to solve 2-3 coding questions per day. Anymore than this, and you may start to lose your drive to study, begin skipping days, etc., and a good study plan will immediately unravel.

Dedicate one day to reviewing that week's questions and study topics. This is extremely important for material retention. Take one day off per week to give your brain a break. This is important, everyone needs to rest and reset.

Let's say you devote 6 weeks to interview preparation. At 6 days per week, with 1 day dedicated to review (leaving 5 days a week for coding questions), you're going to solve 60 questions (assuming 2 questions/day), or 90 questions (assuming 3 questions/day). That's pretty serious progress, and 6 weeks will go by in a blur! Again, dial the time up or down as needed.

Don't cram, unless absolutely necessary! Re-read the section on properly preparing if needed ☺. A crash course will yield marginal results at best, failure at worst.

**The week before the interview**

If you followed the blueprint above, you're not cramming this week. You may be reviewing study material, but mostly you're doing additional research on the company you're interviewing with, the team, culture fit, and position in general. You already put in the study sweat equity. Use this week to spend the time polishing other areas of the interview, such as soft skills, specific technology topics, etc.

**The day/night before the interview**

**Rule #1: get a good night's sleep!!** Nothing is more important than being at your best for what promises to be a very exhaustive day of programming acrobatics. You've done your homework and you're ready - **have confidence in yourself and the hard work and prep time you've put in to get here!**

Do not violate this rule unless a work/family emergency dictates otherwise. And if this rule is broken for subsequently-mentioned reasons, it is probably wise to postpone the interview. This happens more frequently than you think, and folks will understand.

**No, no and no ☺**

**Do not interview while you're sick!!** It's not only not good for your own personal sake of well being, no one wants you there to get them sick! I admit to having done this once - I was on the verge of pneumonia, and my interviews had to be halted and I had to be helped out of the building as I was about to pass out from dehydration. Thankfully, I interviewed with a very understanding and courteous company who offered me another chance to come on campus and interview once I was healthy again.

**The day of the interview**

Be on time and bring a bottle of water (I prefer Gatorade for energy). Most companies will offer you refreshments, but have something handy just in case.

Relax and listen closely. When you engage your interviewer after a coding question has been given to you, do not start coding right away! Ask clarifying questions - this is absolutely expected. For instance, you may be asked a question such as "Delete the nth to last item from a list". Would you start coding this immediately? I posit that you really couldn't - here are clarifying questions that I would ask prior to formulating an attack vector or designing an algorithm to solve the problem.

* What type of list are we discussing? "A linked list"
* Is this a singly-linked list or doubly-linked list? "Singly-linked list"
* Is space complexity important? "No, but time complexity is"
* In that case, is a recursive solution acceptable? "Yes, but it would be nice to see an iterative solution as well if we have time"
* Do you want me to code the helper Node class? "No, you can assume you have int data and Node next"

We now have a completely different question than we started with. "Delete the nth to last node from a singly-linked list". Since the interviewer isn't concerned with space complexity, we can use an elegant and simple recursive solution to start with (recursive calls normally incur a penalty from a space complexity perspective, since recursive calls are stored in the execution stack, with the exception of tail recursion). We know that the interviewer doesn't care about the Node class, and wants an iterative solution as definite bonus points.

**Always solve the problem algorithmically first.** Talk through your approach to the problem with the interviewer. Draw out a small sample set of data to serve as a guide. Write a bit of pseudocode, in step fashion, explaining each step. Then, begin to write real code. Talk through the code - it helps you think on your feet and formulate your thoughts as you go. Test your code against the data set you created (your interviewer will want you to "run" the code) anyway, so take the initiative and do so. Ask the interviewer if they would like for you to write test cases if there is time, and be prepared to do it if asked to; this will earn you major bonus points as most candidates do not do this.

**What to do if you're stuck**

**Always solve the problem algorithmically first.** **If your algorithm is solid, coding it will be easy.** If you get stuck on coding, code your skeleton functions and get the nullcheck/termination clauses and other test cases out of the way to start getting code flowing in the right direction (or at least getting some code on the whiteboard).

Ask for help if you've exhausted your train of thought. You may get docked a few points, you may not. A critical factor in assessing Engineers is how they collaborate with others, and it sure beats staring at a white board in silence. The interviewer won't solve the problem for you, but they will provide hints if needed, this is done very frequently. Build on the hints, and break down your problem into small pieces. Try various solutions, and don't be afraid of the brute force approach - you can optimize it later if needed.

**Additional pointers**

Use the whiteboard judiciously. Start at the top left and work your way down, then right. Write as legibly as possible. Clearly separate pseudo-code from real code. Use generally-accepted coding conventions (in Java, for instance, method names have mixed case letters, beginning with a lower case letter and starting each subsequent word with an upper case letter). Make sure your parentheses and brackets are balanced, and you're not missing a sea of semicolons in your functions. Stay away from one letter variable names if possible; of course, this excludes loop constructs (int i, j) and try/catch blocks (exception e). Loop counters always start with i for index.

In other words, ensure you're writing clean, compilable code - this is definitely expected of you, and you will get demerits for sloppy code.

Be sure to take at least one break. I've been through an interview with 7 interviewers on the panel where a bio break wasn't offered. Ask for one if needed. Stay hydrated and stay positive.

**And most important, relax and show confidence throughout the interview.** Treat this as a situation where you are meeting new people and making some new friends, with some fun challenges thrown in the mix, not a do-or-die pressure-cooker.

**To add:** If you have one bad interview round, shake it off immediately. Put it in the back of your mind and absolutely kill the others. It is very, very rare that candidates go through the entire interview process having aced every single session. Your hire decision will depend on the data points from everyone you interview with, which is why you'll have multiple coding sessions. One sub-par session won't necessarily sink you (and indeed, most of the time it does not, if the others are positive).

**Study Guides**

(for clarity, I have **no affiliation** with anyone or anything I will recommend in this article, and will receive **zero compensation** for the recommendations I make). These are, in my opinion, just great resources to help you along the way.

**Books:**

1. [Cracking the Coding Interview: 150 Programming Questions and Solutions](http://www.amazon.com/Cracking-Coding-Interview-Programming-Questions/dp/098478280X) – to date, there is not a better book out there for technical interview preparation

**Interactive Learning:**

1. [My Code School](http://www.youtube.com/mycodeschool) - One of the best resources I have found that comprehensively covers data structures, algorithms and CS concepts in general. Best of all, it's free!
2. [Udemy – Data Structures and Algorithms.](https://www.udemy.com/introduction-to-data-structures-algorithms-in-java/#/)) – For those unfamiliar, or those that need a good refresher. There is a cost for this course ($49)

**Online Training:**

1. [Leetcode](http://oj.leetcode.com/) – With over 150 questions and counting, this should be your absolute first destination for practicing programming questions and code katas.

**Useful Sites:**

1. [GeeksforGeeks](http://www.geeksforgeeks.org/) – everything but the kitchen sink is here :), including a very nice “interview corner” section, and countless programming questions
2. [careercup](http://www.careercup.com/) – Developed by the author of “Cracking the Coding Interview”, this site remains an invaluable resource to everyone looking to improve their technical interview skills

**Improve and maintain, over time**

Once the rust is gone, and you're fully prepped, it is dead easy to keep your CS and interview skills sharp: **solve one coding question per day.**I've been doing this for the past 3 years or so, and I've realized the following benefits:

* I'm never out of practice
* It helps flex mental muscles and improve critical thinking, the same way solving crossword puzzles, brain teasers or playing a game of chess might.
* The relative time investment is small - no more than 30-35 minutes per day, on average, and the benefit is immeasurable. Were I to find myself in an immediate technical interview need situation, for whatever reason, I am always well prepared.