Behind the Scenes

Most companies conduct their interviews in very similar ways. We will offer an overview of how companies interview and what they're looking for. This information should guide your interview preparation and your reactions during and after the interview.

Once you are selected for an interview, you usually go through a screening interview. This is typically conducted over the phone. College candidates who attend top schools may have these interviews in-person.

Don't let the name fool you; the "screening" interview often involves coding and algorithms questions, and the bar can be just as high as it is for in-person interviews. If you're unsure whether or not the interview will be technical, ask your recruiting coordinator what position your interviewer holds (or what the interview might cover). An engineer will usually perform a technical interview.

Many companies have taken advantage of online synchronized document editors, but others will expect you to write code on paper and read it back over the phone. Some interviewers may even give you "homework" to solve after you hang up the phone or just ask you to email them the code you wrote.

You typically do one or two screening interviewers before being brought on-site.

In an on-site interview round, you usually have 3 to 6 in-person interviews. One of these is often over lunch. The lunch interview is usually not technical, and the interviewer may not even submit feedback. This is a good person to discuss your interests with and to ask about the company culture. Your other interviews will be mostly technical and will involve a combination of coding, algorithm, design/architecture, and behavioral/experience questions.

The distribution of questions between the above topics varies between companies and even teams due to company priorities, size, and just pure randomness. Interviewers are often given a good deal of freedom in their interview questions.

After your interview, your interviewers will provide feedback in some form. In some companies, your interviewers meet together to discuss your performance and come to a decision. In other companies, interviewers submit a recommendation to a hiring manager or hiring committee to make a final decision. In some companies, interviewers don't even make the decision; their feedback goes to a hiring committee to make a decision.

Most companies get back after about a week with next steps (offer, rejection, further interviews, or just an update on the process). Some companies respond much sooner (sometimes same day!) and others take much longer.

If you have waited more than a week, you should follow up with your recruiter. If your recruiter does not respond, this does *not* mean that you are rejected (at least not at any major tech company, and almost any

other company). Let me repeat that again: not responding indicates nothing about your status. The intention is that all recruiters should tell candidates once a final decision is made.

Delays can and do happen. Follow up with your recruiter if you expect a delay, but be respectful when you do. Recruiters are just like you. They get busy and forgetful too.

The Microsoft Interview

Microsoft wants smart people. Geeks. People who are passionate about technology. You probably won't be tested on the ins and outs of C++ APIs, but you will be expected to write code on the board.

In a typical interview, you'll show up at Microsoft at some time in the morning and fill out initial paper work. You'll have a short interview with a recruiter who will give you a sample question. Your recruiter is usually there to prep you, not to grill you on technical questions. If you get asked some basic technical questions, it may be because your recruiter wants to ease you into the interview so that you're less nervous when the "real" interview starts.

Be nice to your recruiter. Your recruiter can be your biggest advocate, even pushing to re-interview you if you stumbled on your first interview. They can fight for you to be hired–or not!

During the day, you'll do four or five interviews, often with two different teams. Unlike many companies, where you meet your interviewers in a conference room, you'll meet with your Microsoft interviewers in their office. This is a great time to look around and get a feel for the team culture.

Depending on the team, interviewers may or may not share their feedback on you with the rest of the interview loop.

When you complete your interviews with a team, you might speak with a hiring manager (often called the "as app", short for "as appropriate"). If so, that's a great sign! It likely means that you passed the interviews with a particular team. It's now down to the hiring manager's decision.

You might get a decision that day, or it might be a week. After one week of no word from HR, send a friendly email asking for a status update.

If your recruiter isn't very responsive, it's because she's busy, not because you're being silently rejected.

Definitely Prepare:

"Why do you want to work for Microsoft?"

In this question, Microsoft wants to see that you're passionate about technology. A great answer might be, "I've been using Microsoft software as long as I can remember, and I'm really impressed at how Microsoft manages to create a product that is universally excellent. For example, I've been using Visual Studio recently to learn game programming, and its APIs are excellent." Note how this shows a passion for technology!

What's Unique:

You'll only reach the hiring manager if you've done well, so if you do, that's a great sign!

Additionally, Microsoft tends to give teams more individual control, and the product set is diverse. Experiences can vary substantially across Microsoft since different teams look for different things.

The Amazon Interview

Amazon's recruiting process typically begins with a phone screen in which a candidate interviews with a specific team. A small portion of the time, a candidate may have two or more interviews, which can indicate either that one of their interviewers wasn't convinced or that they are being considered for a different team or profile. In more unusual cases, such as when a candidate is local or has recently interviewed for a different position, a candidate may only do one phone screen.

The engineer who interviews you will usually ask you to write simple code via a shared document editor. They will also often ask a broad set of questions to explore what areas of technology you're familiar with.

Next, you fly to Seattle (or whichever office you're interviewing for) for four or five interviews with one or two teams that have selected you based on your resume and phone interviews. You will have to code on a whiteboard, and some interviewers will stress other skills. Interviewers are each assigned a specific area to probe and may seem very different from each other. They cannot see the other feedback until they have submitted their own, and they are discouraged from discussing it until the hiring meeting.

The "bar raiser" interviewer is charged with keeping the interview bar high. They attend special training and will interview candidates outside their group in order to balance out the group itself. If one interview seems significantly harder and different, that's most likely the bar raiser. This person has both significant experience with interviews and veto power in the hiring decision. Remember, though: just because you seem to be struggling more in this interview doesn't mean you're actually doing worse. Your performance is judged relative to other candidates; it's not evaluated on a simple "percent correct" basis.

Once your interviewers have entered their feedback, they will meet to discuss it. They will be the people making the hiring decision.

While Amazon's recruiters are usually excellent at following up with candidates, occasionally there are delays. If you haven't heard from Amazon within a week, we recommend a friendly email.

Definitely Prepare:

Amazon cares a lot about scale. Make sure you prepare for scalability questions. You don't need a background in distributed systems to answer these questions. See our recommendations in the System Design and Scalability chapter.

Additionally, Amazon tends to ask a lot of questions about object-oriented design. Check out the Object-Oriented Design chapter for sample questions and suggestions.

What's Unique:

The Bar Raiser is brought in from a different team to keep the bar high. You need to impress both this person and the hiring manager.

Amazon tends to experiment more with its hiring process than other companies do. The process described here is the typical experience, but due to Amazon's experimentation, it's not necessarily universal.

The Google Interview

There are many scary rumors floating around about Google interviews, but they're mostly just that: rumors. The interview is not terribly different from Microsoft's or Amazon's.

A Google engineer performs the first phone screen, so expect tough technical questions. These questions may involve coding, sometimes via a shared document. Candidates are typically held to the same standard and are asked similar questions on phone screens as in on-site interviews.

On your on-site interview, you'll interview with four to six people, one of whom will be a lunch interviewer. Interviewer feedback is kept confidential from the other interviewers, so you can be assured that you enter each interview with blank slate. Your lunch interviewer doesn't submit feedback, so this is a great opportunity to ask honest questions.

Interviewers are typically not given specific focuses, and there is no "structure" or "system" as to what you're asked when. Each interviewer can conduct the interview however she would like.

Written feedback is submitted to a hiring committee (HC) of engineers and managers to make a hire / no-hire recommendation. Feedback is typically broken down into four categories (Analytical Ability, Coding, Experience, and Communication) and you are given an overall score from 1.0 to 4.0. The HC usually does not include any of your interviewers. If it does, it was purely by random chance.

To extend an offer, the HC wants to see at least one interviewer who is an "enthusiastic endorser." In other words, a packet with scores of 3.6, 3.1, 3.1 and 2.6 is better than all 3.1s.

You do not necessarily need to excel in every interview, and your phone screen performance is usually not a strong factor in the final decision.

If the hiring committee recommends an offer, your packet will go to a compensation committee and then to the executive management committee. Returning a decision can take several weeks because there are so many stages and committees.

Definitely Prepare:

As a web-based company, Google cares about how to design a scalable system. So, make sure you prepare for questions from System Design and Scalability.

Google puts a strong focus on analytical (algorithm) skills, regardless of experience. You should be very well prepared for these questions, even if you think your prior experience should count for more.

What's Different:

Your interviewers do not make the hiring decision. Rather, they enter feedback which is passed to a hiring committee. The hiring committee recommends a decision which can be—though rarely is—rejected by Google executives.

▸ The Apple Interview

Much like the company itself, Apple's interview process has minimal bureaucracy. The interviewers will be looking for excellent technical skills, but a passion for the position and the company is also very important. While it's not a prerequisite to be a Mac user, you should at least be familiar with the system.

The interview process usually begins with a recruiter phone screen to get a basic sense of your skills, followed up by a series of technical phone screens with team members.

Once you're invited on campus, you'll typically be greeted by the recruiter who provides an overview of the process. You will then have 6-8 interviews with members of the team with which you're interviewing, as well as key people with whom your team works.

You can expect a mix of one-on-one and two-on-one interviews. Be ready to code on a whiteboard and make sure all of your thoughts are clearly communicated. Lunch is with your potential future manager and appears more casual, but it is still an interview. Each interviewer usually focuses on a different area and is discouraged from sharing feedback with other interviewers unless there's something they want subsequent interviewers to drill into.

Towards the end of the day, your interviewers will compare notes. If everyone still feels you're a viable candidate, you will have an interview with the director and the VP of the organization to which you're applying. While this decision is rather informal, it's a very good sign if you make it. This decision also happens behind the scenes, and if you don't pass, you'll simply be escorted out of the building without ever having been the wiser (until now).

If you made it to the director and VP interviews, all of your interviewers will gather in a conference room to give an official thumbs up or thumbs down. The VP typically won't be present but can still veto the hire if they weren't impressed. Your recruiter will usually follow up a few days later, but feel free to ping him or her for updates.

Definitely Prepare:

If you know what team you're interviewing with, make sure you read up on that product. What do you like about it? What would you improve? Offering specific recommendations can show your passion for the job.

What's Unique:

Apple does two-on-one interviews often, but don't get stressed out about them–it's the same as a one-on-one interview!

Also, Apple employees are huge Apple fans. You should show this same passion in your interview.

The Facebook Interview

Once selected for an interview, candidates will generally do one or two phone screens. Phone screens will be technical and will involve coding, usually an online document editor.

After the phone interview(s), you might be asked to do a homework assignment that will include a mix of coding and algorithms. Pay attention to your coding style here. If you've never worked in an environment which had thorough code reviews, it may be a good idea to get someone who has to review your code.

During your on-site interview, you will interview primarily with other software engineers, but hiring managers are also involved whenever they are available. All interviewers have gone through comprehensive interview training, and who you interview with has no bearing on your odds of getting an offer.

Each interviewer is given a "role" during the on-site interviews, which helps ensure that there are no repetitive questions and that they get a holistic picture of a candidate. These roles are:

- Behavioral ("Jedi"): This interview assesses your ability to be successful in Facebook's environment. Would you fit well with the culture and values? What are you excited about? How do you tackle challenges? Be prepared to talk about your interest in Facebook as well. Facebook wants passionate people. You might also be asked some coding questions in this interview.
- Coding and Algorithms ("Ninja"): These are your standard coding and algorithms questions, much like what you'll find in this book. These questions are designed to be challenging. You can use any programming language you want.

- Design/Architecture ("Pirate"): For a backend software engineer, you might be asked system design questions. Front-end or other specialties will be asked design questions related to that discipline. You should openly discuss different solutions and their tradeoffs.

You can typically expect two "ninja" interviews and one "jedi" interview. Experienced candidates will also usually get a "pirate" interview.

After your interview, interviewers submit written feedback, prior to discussing your performance with each other. This ensures that your performance in one interview will not bias another interviewer's feedback.

Once everyone's feedback is submitted, your interviewing team and a hiring manager get together to collaborate on a final decision. They come to a consensus decision and submit a final hire recommendation to the hiring committee.

Definitely Prepare:

The youngest of the "elite" tech companies, Facebook wants developers with an entrepreneurial spirit. In your interviews, you should show that you love to build stuff fast.

They want to know you can hack together an elegant and scalable solution using any language of choice. Knowing PHP is not especially important, particularly given that Facebook also does a lot of backend work in C++, Python, Erlang, and other languages.

What's Unique:

Facebook interviews developers for the company "in general," not for a specific team. If you are hired, you will go through a six-week "bootcamp" which will help ramp you up in the massive code base. You'll get mentorship from senior devs, learn best practices, and, ultimately, get a greater flexibility in choosing a project than if you were assigned to a project in your interview.

The Palantir Interview

Unlike some companies which do "pooled" interviews (where you interview with the company as a whole, not with a specific team), Palantir interviews for a specific team. Occasionally, your application might be re-routed to another team where there is a better fit.

The Palantir interview process typically starts with two phone interviews. These interviews are about 30 to 45 minutes and will be primarily technical. Expect to cover a bit about your prior experience, with a heavy focus on algorithm questions.

You might also be sent a HackerRank coding assessment, which will evaluate your ability to write optimal algorithms and correct code. Less experienced candidates, such as those in college, are particularly likely to get such a test.

After this, successful candidates are invited to campus and will interview with up to five people. Onsite interviews cover your prior experience, relevant domain knowledge, data structures and algorithms, and design.

You may also likely get a demo of Palantir's products. Ask good questions and demonstrate your passion for the company.

After the interview, the interviewers meet to discuss your feedback with the hiring manager.

Definitely Prepare:

Palantir values hiring brilliant engineers. Many candidates report that Palantir's questions were harder than those they saw at Google and other top companies. This doesn't necessarily mean it's harder to get an offer (although it certainly can); it just means interviewers prefer more challenging questions. If you're interviewing with Palantir, you should learn your core data structures and algorithms inside and out. Then, focus on preparing with the hardest algorithm questions.

Brush up on system design too if you're interviewing for a backend role. This is an important part of the process.

What's Unique:

A coding challenge is a common part of Palantir's process. Although you'll be at your computer and can look up material as needed, don't walk into this unprepared. The questions can be extremely challenging and the efficiency of your algorithm will be evaluated. Thorough interview preparation will help you here. You can also practice coding challenges online at HackerRank.com.

Special Situations

(removed sections from Indian Edition)

Product (and Program) Management

These "PM" roles vary wildly across companies and even within a company. At Microsoft, for instance, some PMs may be essentially customer evangelists, working in a customer-facing role that borders on marketing. Across campus though, other PMs may spend much of their day coding. The latter type of PMs would likely be tested on coding, since this is an important part of their job function.

Generally speaking, interviewers for PM positions are looking for candidates to demonstrate skills in the following areas:

- *Handling Ambiguity:* This is typically not the most critical area for an interview, but you should be aware that interviewers do look for skill here. Interviewers want to see that, when faced with an ambiguous situation, you don't get overwhelmed and stall. They want to see you tackle the problem head on: seeking new information, prioritizing the most important parts, and solving the problem in a structured way. This typically will not be tested directly (though it can be), but it may be one of many things the interviewer is looking for in a problem.
- *Customer Focus (Attitude):* Interviewers want to see that your attitude is customer-focused. Do you assume that everyone will use the product just like you do? Or are you the type of person who puts himself in the customer's shoes and tries to understand how they want to use the product? Questions like "Design an alarm clock for the blind" are ripe for examining this aspect. When you hear a question like this, be sure to ask a lot of questions to understand *who* the customer is and *how* they are using the product. The skills covered in the Testing section are closely related to this.
- *Customer Focus (Technical Skills):* Some teams with more complex products need to ensure that their PMs walk in with a strong understanding of the product, as it would be difficult to acquire this knowledge on the job. Deep technical knowledge of mobile phones is probably not necessary to work on the Android or Windows Phone teams (although it might still be nice to have), whereas an understanding of security might be necessary to work on Windows Security. Hopefully, you wouldn't interview with a team that required specific technical skills unless you at least claim to possess the requisite skills.
- *Multi-Level Communication:* PMs need to be able to communicate with people at all levels in the company, across many positions and ranges of technical skills. Your interviewer will want to see that you possess this flexibility in your communication. This is often examined directly, through a question such as, "Explain TCP/IP to your grandmother." Your communication skills may also be assessed by how you discuss your prior projects.
- *Passion for Technology:* Happy employees are productive employees, so a company wants to make sure

that you'll enjoy the job and be excited about your work. A passion for technology—and, ideally, the company or team—should come across in your answers. You may be asked a question directly like, "Why are you interested in Microsoft?" Additionally, your interviewers will look for enthusiasm in how you discuss your prior experience and how you discuss the team's challenges. They want to see that you will be eager to face the job's challenges.

- *Teamwork / Leadership:* This may be the most important aspect of the interview, and—not surprisingly—the job itself. All interviewers will be looking for your ability to work well with other people. Most commonly, this is assessed with questions like, "Tell me about a time when a teammate wasn't pulling his / her own weight." Your interviewer is looking to see that you handle conflicts well, that you take initiative, that you understand people, and that people like working with you. Your work preparing for behavioral questions will be extremely important here.

All of the above areas are important skills for PMs to master and are therefore key focus areas of the interview. The weighting of each of these areas will roughly match the importance that the area holds in the actual job.

Acquisitions and Acquihires

During the technical due diligence process for many acquisitions, the acquirer will often interview most or all of a startup's employees. Google, Yahoo, Facebook, and many other companies have this as a standard part of many acquisitions.

Which startups go through this? And why?

Part of the reasoning for this is that their employees had to go through this process to get hired. They don't want acquisitions to be an "easy way" into the company. And, since the team is a core motivator for the acquisition, they figure it makes sense to assess the skills of the team.

Not all acquisitions are like this, of course. The famous multi-billion dollar acquisitions generally did not have to go through this process. Those acquisitions, after all, are usually about the user base and community, less so about the employees or even the technology. Assessing the team's skills is less essential.

However, it is not as simple as "acquihires get interviewed, traditional acquisitions do not." There is a big gray area between acquihires (i.e., talent acquisitions) and product acquisitions. Many startups are acquired for the team and ideas behind the technology. The acquirer might discontinue the product, but have the team work on something very similar.

If your startup is going through this process, you can typically expect your team to have interviews very similar to what a normal candidate would experience (and, therefore, very similar to what you'll see in this book).

How important are these interviews?

These interviews can carry enormous importance. They have three different roles:

- They can make or break acquisitions. They are often the reason a company does not get acquired.
- They determine which employees receive offers to join the acquirer.
- They can affect the acquisition price (in part as a consequence of the number of employees who join).

These interviews are much more than a mere "screen."

Which employees go through the interviews?

For tech startups, usually all of the engineers go through the interview process, as they are one of the core motivators for the acquisition.

In addition, sales, customer support, product managers, and essentially any other role might have to go through it.

The CEO is often slotted into a product manager interview or a dev manager interview, as this is often the closest match for the CEO's current responsibilities. This is not an absolute rule, though. It depends on what the CEO's role presently is and what the CEO is interested in. With some of my clients, the CEO has even opted to not interview and to leave the company upon the acquisition.

What happens to employees who don't perform well in the interview?

Employees who underperform will often not receive offers to join the acquirer. (If many employees don't perform well, then the acquisition will likely not go through.)

In some cases, employees who performed poorly in interviews will get contract positions for the purpose of "knowledge transfer." These are temporary positions with the expectation that the employee leaves at the termination of the contract (often six months), although sometimes the employee ends up being retained.

In other cases, the poor performance was a result of the employee being mis-slotted. This occurs in two common situations:

- Sometimes a startup labels someone who is not a "traditional" software engineer as a software engineer. This often happens with data scientists or database engineers. These people may underperform during the software engineer interviews, as their actual role involves other skills.
- In other cases, a CEO "sells" a junior software engineer as more senior than he actually is. He underperforms for the senior bar because he's being held to an unfairly high standard.

In either case, sometimes the employee will be re-interviewed for a more appropriate position. (Other times though, the employee is just out of luck.)

In rare cases, a CEO is able to override the decision for a particularly strong employee whose interview performance didn't reflect this.

Your "best" (and worst) employees might surprise you.

The problem-solving/algorithm interviews conducted at the top tech companies evaluate particular skills, which might not perfectly match what their manager evaluates in their employees.

I've worked with many companies that are surprised at who their strongest and weakest performers are in interviews. That junior engineer who still has a lot to learn about professional development might turn out to be a great problem-solver in these interviews.

Don't count anyone out—or in—until you've evaluated them the same way their interviewers will.

Are employees held to the same standards as typical candidates?

Essentially yes, although there is a bit more leeway.

The big companies tend to take a risk-averse approach to hiring. If someone is on the fence, they often lean towards a no-hire.

In the case of an acquisition, the "on the fence" employees can be pulled through by strong performance from the rest of the team.

How do employees tend to react to the news of an acquisition/acquihire?

This is a big concern for many startup CEOs and founders. Will the employees be upset about this process? Or, what if we get their hopes up but it doesn't happen?

What I've seen with my clients is that the leadership is worried about this more than is necessary.

Certainly, some employees are upset about the process. They might not be excited about joining one of the big companies for any number of reasons.

Most employees, though, are cautiously optimistic about the process. They hope it goes through, but they know that the existence of these interviews means that it might not.

What happens to the team after an acquisition?

Every situation is different. However, most of my clients have been kept together as a team, or possibly integrated into an existing team.

How should you prepare your team for acquisition interviews?

Interview prep for acquisition interviews is fairly similar to typical interviews at the acquirer. The difference is that your company is doing this as a team and that each employee wasn't individually selected for the interview on their own merits.

You're all in this together.

Some startups I've worked with put their "real" work on hold and have their teams spend the next two or three weeks on interview prep.

Obviously, that's not a choice all companies can make, but—from the perspective of wanting the acquisition to go through—that does increase your results substantially.

Your team should study individually, in teams of two or three, or by doing mock interviews with each other. If possible, use all three of these approaches.

Some people may be less prepared than others.

Many developers at startups might have only vaguely heard of big O time, binary search tree, breadth-first search, and other important concepts. They'll need some extra time to prepare.

People without computer science degrees (or who earned their degrees a long time ago) should focus first on learning the core concepts discussed in this book, especially big O time (which is one of the most important). A good first exercise is to implement all the core data structures and algorithms from scratch.

If the acquisition is important to your company, give these people the time they need to prepare. They'll need it.

Don't wait until the last minute.

As a startup, you might be used to taking things as they come without a ton of planning. Startups that do this with acquisition interviews tend not to fare well.

Acquisition interviews often come up very suddenly. A company's CEO is chatting with an acquirer (or several acquirers) and conversations get increasingly serious. The acquirer mentions the possibility of interviews at some point in the future. Then, all of a sudden, there's a "come in at the end of this week" message.

If you wait until there's a firm date set for the interviews, you probably won't get much more than a couple of days to prepare. That might not be enough time for your engineers to learn core computer science concepts and practice interview questions.

VIII

The Offer and Beyond

Just when you thought you could sit back and relax after your interviews, now you're faced with the post-interview stress: Should you accept the offer? Is it the right one? How do you decline an offer? What about deadlines? We'll handle a few of these issues here and go into more details about how to evaluate an offer, and how to negotiate it.

Handling Offers and Rejection

Whether you're accepting an offer, declining an offer, or responding to a rejection, it matters what you do.

Offer Deadlines and Extensions

When companies extend an offer, there's almost always a deadline attached to it. Usually these deadlines are one to four weeks out. If you're still waiting to hear back from other companies, you can ask for an extension. Companies will usually try to accommodate this, if possible.

Declining an Offer

Even if you aren't interested in working for this company right now, you might be interested in working for it in a few years. (Or, your contacts might one day move to a more exciting company.) It's in your best interest to decline the offer on good terms and keep a line of communication open.

When you decline an offer, provide a reason that is non-offensive and inarguable. For example, if you were declining a big company for a startup, you could explain that you feel a startup is the right choice for you at this time. The big company can't suddenly "become" a startup, so they can't argue about your reasoning.

Handling Rejection

Getting rejected is unfortunate, but it doesn't mean that you're not a great engineer. Lots of great engineers do poorly, either because they don't "test well" on these sort of interviewers, or they just had an "off" day.

Fortunately, most companies understand that these interviews aren't perfect and many good engineers get rejected. For this reason, companies are often eager to re-interview previously rejected candidate. Some companies will even reach out to old candidates or expedite their application *because* of their prior performance.

When you do get the unfortunate call, use this as an opportunity to build a bridge to re-apply. Thank your recruiter for his time, explain that you're disappointed but that you understand their position, and ask when you can reapply to the company.

You can also ask for feedback from the recruiter. In most cases, the big tech companies won't offer feedback, but there are some companies that will. It doesn't hurt to ask a question like, "Is there anything you'd suggest I work on for next time?"

Evaluating the Offer

Congratulations! You got an offer! And—if you're lucky—you may have even gotten multiple offers. Your recruiter's job is now to do everything he can to encourage you to accept it. How do you know if the company is the right fit for you? We'll go through a few things you should consider in evaluating an offer.

The Financial Package

Perhaps the biggest mistake that candidates make in evaluating an offer is looking too much at their salary. Candidates often look so much at this one number that they wind up accepting the offer that is *worse* financially. Salary is just one part of your financial compensation. You should also look at:

- *Signing Bonus, Relocation, and Other One Time Perks:* Many companies offer a signing bonus and/or relocation. When comparing offers, it's wise to amortize this cash over three years (or however long you expect to stay).
- *Cost of Living Difference:* Taxes and other cost of living differences can make a big difference in your take-home pay. Silicon Valley, for example, is 30+% more expensive than Seattle.
- *Annual Bonus:* Annual bonuses at tech companies can range from anywhere from 3% to 30%. Your recruiter might reveal the average annual bonus, but if not, check with friends at the company.
- *Stock Options and Grants:* Equity compensation can form another big part of your annual compensation. Like signing bonuses, stock compensation between companies can be compared by amortizing it over three years and then lumping that value into salary.

Remember, though, that what you learn and how a company advances your career often makes far more of a difference to your long term finances than the salary. Think very carefully about how much emphasis you really want to put on money right now.

Career Development

As thrilled as you may be to receive this offer, odds are, in a few years, you'll start thinking about interviewing again. Therefore, it's important that you think right now about how this offer would impact your career path. This means considering the following questions:

- How good does the company's name look on my resume?
- How much will I learn? Will I learn relevant things?
- What is the promotion plan? How do the careers of developers progress?
- If I want to move into management, does this company offer a realistic plan?
- Is the company or team growing?
- If I do want to leave the company, is it situated near other companies I'm interested in, or will I need to move?

The final point is extremely important and usually overlooked. If you only have a few other companies to pick from in your city, your career options will be more restricted. Fewer options means that you're less likely to discover really great opportunities.

Company Stability

All else being equal, of course stability is a good thing. No one wants to be fired or laid off.

However, all else isn't actually equal. The more stable companies are also often growing more slowly.

How much emphasis you should put on company stability really depends on you and your values. For some candidates, stability should not be a large factor. Can you fairly quickly find a new job? If so, it might be better to take the rapidly growing company, even if it's unstable? If you have work visa restrictions or just aren't confident in your ability to find something new, stability might be more important.

The Happiness Factor

Last but not least, you should of course consider how happy you will be. Any of the following factors may impact that:

- *The Product:* Many people look heavily at what product they are building, and of course this matters a bit. However, for most engineers, there are more important factor, such as who you work with.
- *Manager and Teammates:* When people say that they love, or hate, their job, it's often because of their teammates and their manager. Have you met them? Did you enjoy talking with them?
- *Company Culture:* Culture is tied to everything from how decisions get made, to the social atmosphere, to how the company is organized. Ask your future teammates how they would describe the culture.
- *Hours:* Ask future teammates about how long they typically work, and figure out if that meshes with your lifestyle. Remember, though, that hours before major deadlines are typically much longer.

Additionally, note that if you are given the opportunity to switch teams easily (like you are at Google and Facebook), you'll have an opportunity to find a team and product that matches you well.

Negotiation

Years ago, I signed up for a negotiations class. On the first day, the instructor asked us to imagine a scenario where we wanted to buy a car. Dealership A sells the car for a fixed $20,000—no negotiating. Dealership B allows us to negotiate. How much would the car have to be (after negotiating) for us to go to Dealership B? (Quick! Answer this for yourself!)

On average, the class said that the car would have to be $750 cheaper. In other words, students were willing to pay $750 just to avoid having to negotiate for an hour or so. Not surprisingly, in a class poll, most of these students also said they didn't negotiate their job offer. They just accepted whatever the company gave them.

Many of us can probably sympathize with this position. Negotiation isn't fun for most of us. But still, the financial benefits of negotiation are usually worth it.

Do yourself a favor. Negotiate. Here are some tips to get you started.

1. *Just Do It.* Yes, I know it's scary; (almost) no one likes negotiating. But it's so, so worth it. Recruiters will not revoke an offer because you negotiated, so you have little to lose. This is especially true if the offer is from a larger company. You probably won't be negotiating with your future teammates.
2. *Have a Viable Alternative.* Fundamentally, recruiters negotiate with you because they're concerned you may not join the company otherwise. If you have alternative options, that will make their concern much more real.
3. *Have a Specific "Ask":* It's more effective to ask for an additional $7000 in salary than to just ask for "more."

After all, if you just ask for more, the recruiter could throw in another $1000 and technically have satisfied your wishes.

4. *Overshoot:* In negotiations, people usually don't agree to whatever you demand. It's a back and forth conversation. Ask for a bit more than you're really hoping to get, since the company will probably meet you in the middle.
5. *Think Beyond Salary:* Companies are often more willing to negotiate on non-salary components, since boosting your salary too much could mean that they're paying you more than your peers. Consider asking for more equity or a bigger signing bonus. Alternatively, you may be able to ask for your relocation benefits in cash, instead of having the company pay directly for the moving fees. This is a great avenue for many college students, whose actual moving expenses are fairly cheap.
6. *Use Your Best Medium:* Many people will advise you to only negotiate over the phone. To a certain extent, they're right; it is better to negotiate over the phone. However, if you don't feel comfortable on a phone negotiation, do it via email. It's more important that you attempt to negotiate than that you do it via a specific medium.

Additionally, if you're negotiating with a big company, you should know that they often have "levels" for employees, where all employees at a particular level are paid around the same amount. Microsoft has a particularly well-defined system for this. You can negotiate within the salary range for your level, but going beyond that requires bumping up a level. If you're looking for a big bump, you'll need to convince the recruiter and your future team that your experience matches this higher level—a difficult, but feasible, thing to do.

On the Job

Navigating your career path doesn't end at the interview. In fact, it's just getting started. Once you actually join a company, you need to start thinking about your career path. Where will you go from here, and how will you get there?

Set a Timeline

It's a common story: you join a company, and you're psyched. Everything is great. Five years later, you're still there. And it's then that you realize that these last three years didn't add much to your skill set or to your resume. Why didn't you just leave after two years?

When you're enjoying your job, it's very easy to get wrapped up in it and not realize that your career is not advancing. This is why you should outline your career path before starting a new job. Where do you want to be in ten years? And what are the steps necessary to get there? In addition, each year, think about what the next year of experience will bring you and how your career or your skill set advanced in the last year.

By outlining your path in advance and checking in on it regularly, you can avoid falling into this complacency trap.

Build Strong Relationships

When you want to move on to something new, your network will be critical. After all, applying online is tricky; a personal referral is much better, and your ability to do so hinges on your network.

At work, establish strong relationships with your manager and teammates. When employees leave, keep in touch with them. Just a friendly note a few weeks after their departure will help to bridge that connection from a work acquaintance to a personal acquaintance.

This same approach applies to your personal life. Your friends, and your friends of friends, are valuable connections. Be open to helping others, and they'll be more likely to help you.

Ask for What You Want

While some managers may really try to grow your career, others will take a more hands-off approach. It's up to you to pursue the challenges that are right for your career.

Be (reasonably) frank about your goals with your manager. If you want to take on more back-end coding projects, say so. If you'd like to explore more leadership opportunities, discuss how you might be able to do so.

You need to be your best advocate, so that you can achieve goals according to your timeline.

Keep Interviewing

Set a goal of interviewing at least once a year, even if you aren't actively looking for a new job. This will keep your interview skills fresh, and also keep you in tune with what sorts of opportunities (and salaries) are out there.

If you get an offer, you don't have to take it. It will still build a connection with that company in case you want to join at a later date.

XI

Advanced Topics

When writing the 6th edition, I had a number of debates about what should and shouldn't be included. Red-black trees? Dijkstra's algorithm? Topological sort?

On one hand, I'd had a number of requests to include these topics. Some people insisted that these topics are asked "all the time" (in which case, they have a very different idea of what this phrase means!). There was clearly a desire—at least from some people—to include them. And learning more can't hurt, right?

On the other hand, I know these topics to be rarely asked. It happens, of course. Interviewers are individuals and might have their own ideas of what is "fair game" or "relevant" for an interview. But it's rare. When it does come up, if you don't know the topic, it's unlikely to be a big red flag.

> Admittedly, as an interviewer, I *have* asked candidates questions where the solution was essentially an application of one of these algorithms. On the rare occasions that a candidate already knew the algorithm, they did not benefit from this knowledge (nor were they hurt by it). I want to evaluate your ability to solve a problem you haven't seen before. So, I'll take into account whether you know the underlying algorithm in advance.

I believe in giving people a fair expectation of the interview, not scaring people into excess studying. I also have no interest in making the book more "advanced" so as to help book sales, at the expense of your time and energy. That's not fair or right to do to you.

(Additionally, I didn't want to give interviewers—who I know to be reading this—the impression that they can or should be covering these more advanced topics. Interviewers: If you ask about these topics, you're testing knowledge of algorithms. You're just going to wind up eliminating a lot of perfectly smart people.)

But there are many borderline "important" topics. They're not often asked, but sometimes they are.

Ultimately, I decided to leave the decision in your hands. After all, you know better than I do how thorough you want to be in your preparation. If you want to do an extra thorough job, read this. If you just love learning data structures and algorithms, read this. If you want to see new ways of approaching problems, read this.

But if you're pressed for time, this studying isn't a super high priority.

Useful Math

Here's some math that can be useful in some questions. There are more formal proofs that you can look up online, but we'll focus here on giving you the intuition behind them. You can think of these as informal proofs.

Sum of Integers 1 through N

What is 1 + 2 + ... + n? Let's figure it out by pairing up low values with high values.

If n is even, we pair 1 with n, 2 with n - 1, and so on. We will have $\frac{n}{2}$ pairs each with sum n + 1.

If n is odd, we pair 0 with n, 1 with n - 1, and so on. We will have $\frac{n+1}{2}$ pairs with sum n.

n is even			
pair #	**a**	**b**	**a + b**
1	1	n	n + 1
2	2	n - 1	n + 1
3	3	n - 2	n + 1
4	4	n - 3	n + 1
...	...	...	...
$\frac{n}{2}$	$\frac{n}{2}$	$\frac{n}{2}+1$	n + 1
total:	$\frac{n}{2}*(n+1)$		

n is odd			
pair #	**a**	**b**	**a + b**
1	0	n	n
2	1	n - 1	n
3	2	n - 2	n
4	3	n - 3	n
...	...	...	...
$\frac{n+1}{2}$	$\frac{n-1}{2}$	$\frac{n+1}{2}$	n
total:	$\frac{n+1}{2} * n$		

In either case, the sum is $\frac{n(n+1)}{2}$.

This reasoning comes up a lot in nested loops. For example, consider the following code:

```
1   for (int i = 0; i < n; i++) {
2     for (int j = i + 1; j < n; j++) {
3       System.out.println(i + j);
4     }
5   }
```

On the first iteration of the outer for loop, the inner for loop iterates n - 1 times. On the second iteration of the outer for loop, the inner for loop iterates n-2 times. Next, n - 3, then n - 4, and so on. There are $\frac{n(n-1)}{2}$ total iterations of the inner for loop. Therefore, this code takes $O(n^2)$ time.

Sum of Powers of 2

Consider this sequence: $2^0 + 2^1 + 2^2 + ... + 2^n$. What is its result?

A nice way to see this is by looking at these values in binary.

	Power	Binary	Decimal
	2^0	00001	1
	2^1	00010	2
	2^2	00100	4
	2^3	01000	8
	2^4	10000	16
sum:	2^5-1	11111	32 - 1 = 31

Therefore, the sum of $2^0 + 2^1 + 2^2 + ... + 2^n$ would, in base 2, be a sequence of (n + 1) 1s. This is $2^{n+1} - 1$.

Takeaway: The sum of a sequence of powers of two is roughly equal to the *next* value in the sequence.

Bases of Logs

Suppose we have something in $\log_2$ (log base 2). How do we convert that to $\log_{10}$? That is, what's the relationship between $\log_b k$ and $\log_x k$?

Let's do some math. Assume $c = \log_b k$ and $y = \log_x k$.

```
log_b k = c --> b^c = k          // This is the definition of log.
log_x (b^c) = log_x k            // Take log of both sides of b^c = k.
c log_x b = log_x k              // Rules of logs. You can move out the exponents.
c = log_b k = log_x k / log_x b  // Dividing above expression and substituting c.
```

Therefore, if we want to convert $\log_2 p$ to $\log_{10}$, we just do this:

$$\log_{10} p = \frac{\log_2 p}{\log_2 10}$$

Takeaway: Logs of different bases are only off by a constant factor. For this reason, we largely ignore what the base of a log within a big O expression. It doesn't matter since we drop constants anyway.

Permutations

How many ways are there of rearranging a string of n unique characters? Well, you have n options for what to put in the first characters, then n - 1 options for what to put in the second slot (one option is taken), then n - 2 options for what to put in the third slot, and so on. Therefore, the total number of strings is n!.

$$n! = \underline{n} * \underline{n - 1} * \underline{n - 2} * \underline{n - 3} * \ldots * \underline{1}$$

What if you were forming a k-length string (with all unique characters) from n total unique characters? You can follow similar logic, but you'd just stop your selection/multiplication earlier.

$$\frac{n!}{(n-k)!} = \underline{n} * \underline{n - 1} * \underline{n - 2} * \underline{n - 3} * \ldots * \underline{n - k + 1}$$

Combinations

Suppose you have a set of n distinct characters. How many ways are there of selecting k characters into a new set (where order doesn't matter)? That is, how many k-sized subsets are there out of n distinct elements? This is what the expression n-choose-k means, which is often written $\binom{n}{k}$.

Imagine we made a list of all the sets by first writing all k-length substrings and then taking out the duplicates.

From the above *Permutations* section, we'd have $n!/(n-k)!$ k-length substrings.

Since each k-sized subset can be rearranged k! unique ways into a string, each subset will be duplicated k! times in this list of substrings. Therefore, we need to divide by k! to take out these duplicates.

$$\binom{n}{k} = \frac{1}{k!} * \frac{n!}{(n-k)!} = \frac{n!}{k!(n-k)!}$$

Proof by Induction

Induction is a way of proving something to be true. It is closely related to recursion. It takes the following form.

Task: Prove statement P(k) is true for all k >= b.

- Base Case: Prove the statement is true for P(b). This is usually just a matter of plugging in numbers.
- Assumption: Assume the statement is true for P(n).
- Inductive Step: Prove that *if* the statement is true for P(n), then it's true for P(n+1).

This is like dominoes. If the first domino falls, and one domino always knocks over the next one, then all the dominoes must fall.

Let's use this to prove that there are 2^n subsets of an n-element set.

- Definitions: let $S = \{a_1, a_2, a_3, \ldots, a_n\}$ be the n-element set.

- Base case: Prove there are 2^0 subsets of `{}`. This is true, since the only subset of `{}` is `{}`.
- Assume that there are 2^n subsets of $\{a_1, a_2, a_3, ..., a_n\}$.
- Prove that there are 2^{n+1} subsets of $\{a_1, a_2, a_3, ..., a_{n+1}\}$.

  Consider the subsets of $\{a_1, a_2, a_3, ..., a_{n+1}\}$. Exactly half will contain a_{n+1} and half will not.

  The subsets that do not contain a_{n+1} are just the subsets of $\{a_1, a_2, a_3, ..., a_n\}$. We assumed there are 2^n of those.

  Since we have the same number of subsets with x as without x, there are 2^n subsets with a_{n+1}.

  Therefore, we have $2^n + 2^n$ subsets, which is 2^{n+1}.

Many recursive algorithms can be proved valid with induction.

Topological Sort

A topological sort of a directed graph is a way of ordering the list of nodes such that if `(a, b)` is an edge in the graph then a will appear before b in the list. If a graph has cycles or is not directed, then there is no topological sort.

There are a number of applications for this. For example, suppose the graph represents parts on an assembly line. The edge (Handle, Door) indicates that you need to assemble the handle before the door. The topological sort would offer a valid ordering for the assembly line.

We can construct a topological sort with the following approach.

1. Identify all nodes with no incoming edges and add those nodes to our topological sort.
   - We know those nodes are safe to add first since they have nothing that needs to come before them. Might as well get them over with!
   - We know that such a node must exist if there's no cycle. After all, if we picked an arbitrary node we could just walk edges backwards arbitrarily. We'll either stop at some point (in which case we've found a node with no incoming edges) or we'll return to a prior node (in which case there is a cycle).
2. When we do the above, remove each node's outbound edges from the graph.
   - Those nodes have already been added to the topological sort, so they're basically irrelevant. We can't violate those edges anymore.
3. Repeat the above, adding nodes with no incoming edges and removing their outbound edges. When all the nodes have been added to the topological sort, then we are done.

More formally, the algorithm is this:

1. Create a queue `order`, which will eventually store the valid topological sort. It is currently empty.
2. Create a queue `processNext`. This queue will store the next nodes to process.
3. Count the number of incoming edges of each node and set a class variable `node.inbound`. Nodes typically only store their outgoing edges. However, you can count the inbound edges by walking through each node n and, for each of its outgoing edges `(n, x)`, incrementing `x.inbound`.
4. Walk through the nodes again and add to `processNext` any node where `x.inbound == 0`.
5. While `processNext` is not empty, do the following:
   - Remove first node n from `processNext`.

- » For each edge `(n, x)`, decrement `x.inbound`. If `x.inbound == 0`, append `x` to `processNext`.
- » Append `n` to `order`.

6. If `order` contains all the nodes, then it has succeeded. Otherwise, the topological sort has failed due to a cycle.

This algorithm does sometimes come up in interview questions. Your interviewer probably wouldn't expect you to know it offhand. However, it would be reasonable to have you derive it even if you've never seen it before.

Dijkstra's Algorithm

In some graphs, we might want to have edges with weights. If the graph represented cities, each edge might represent a road and its weight might represent the travel time. In this case, we might want to ask, just as your GPS mapping system does, what's the shortest path from your current location to another point p? This is where Dijksta's algorithm comes in.

Dijkstra's algorithm is a way to find the shortest path between two points in a weighted directed graph (which might have cycles). All edges must have positive values.

Rather than just stating what Dijkstra's algorithm is, let's try to derive it. Consider the earlier described graph. We could find the shortest path from `s` to `t` by literally taking all possible routes using actual time. (Oh, and we'll need a machine to clone ourselves.)

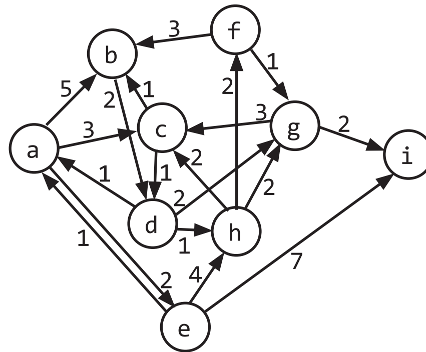
1. Start off at `s`.
2. For each of `s`'s outbound edges, clone ourselves and start walking. If the edge `(s, x)` has weight 5, we should actually take 5 minutes to get there.
3. Each time we get to a node, check if anyone's been there before. If so, then just stop. We're automatically not as fast as another path since someone beat us here from `s`. If no one has been here before, then clone ourselves and head out in all possible directions.
4. The first one to get to `t` wins.

This works just fine. But, of course, in the real algorithm we don't want to literally use a timer to find the shortest path.

Imagine that each clone could jump immediately from one node to its adjacent nodes (regardless of the edge weight), but it kept a `time_so_far` log of how long its path would have taken if it did walk at the "true" speed. Additionally, only one person moves at a time, and it's always the one with the lowest `time_so_far`. This is sort of how Dijkstra's algorithm works.

Dijkstra's algorithm finds the minimum weight path from a start node s to *every* node on the graph.

Consider the following graph.



Assume we are trying to find the shortest path from a to i. We'll use Dijkstra's algorithm to find the shortest path from a to all other nodes, from which we will clearly have the shortest path from a to i.

We first initialize several variables:

- `path_weight[node]`: maps from each node to the total weight of the shortest path. All values are initialized to infinity, except for `path_weight[a]` which is initialized to `0`.
- `previous[node]`: maps from each node to the previous node in the (current) shortest path.
- `remaining`: a priority queue of all nodes in the graph, where each node's priority is defined by its `path_weight`.

Once we've initialized these values, we can start adjusting the values of `path_weight`.

> A (min) **priority queue** is an abstract data type that—at least in this case—supports insertion of an object and key, removing the object with the minimum key, and decreasing a key. (Think of it like a typical queue, except that, instead of removing the oldest item, it removes the item with the lowest or highest priority.) It is an abstract data type because it is defined by its behavior (its operations). Its underlying implementation can vary. You could implement a priority queue with an array or a min (or max) heap (or many other data structures).

We iterate through the nodes in `remaining` (until `remaining` is empty), doing the following:

1. Select the node in `remaining` with the lowest value in `path_weight`. Call this node n.
2. For each adjacent node, compare `path_weight[x]` (which is the weight of the current shortest path from a to x) to `path_weight[n] + edge_weight[(n, x)]`. That is, could we get a path from a to x with lower weight by going through n instead of our current path? If so, update `path_weight` and `previous`.
3. Remove n from `remaining`.

When `remaining` is empty, then `path_weight` stores the weight of the current shortest path from a to each node. We can reconstruct this path by tracing through `previous`.

Let's walk through this on the above graph.

1. The first value of n is a. We look at its adjacent nodes (b, c, and e), update the values of `path_weight` (to 5, 3, and 2) and `previous` (to a) and then remove a from `remaining`.
2. Then, we go to the next smallest node, which is e. We previously updated `path_weight[e]` to be 2. Its adjacent nodes are h and i, so we update `path_weight` (to 6 and 9) and `previous` for both of those.

Observe that 6 is `path_weight[e]` (which is 2) + the weight of the edge `(e, h)` (which is 4).

3. The next smallest node is `c`, which has `path_weight` 3. Its adjacent nodes are `b` and `d`. The value of `path_weight[d]` is infinity, so we update it to 4 (which is `path_weight[c] + weight(edge c, d)`). The value of `path_weight[b]` has been previously set to 5. However, since `path_weight[c] + weight(edge c, b)` (which is 3 + 1 = 4) is less than 5, we update `path_weight[b]` to 4 and `previous` to `c`. This indicates that we would improve the path from a to b by going through `c`.

We continue doing this until `remaining` is empty. The following diagram shows the changes to the `path_weight` (left) and `previous` (right) at each step. The topmost row shows the current value for n (the node we are removing from `remaining`). We black out a row after it has been removed from `remaining`.

	INITIAL		n = a		n = e		n = c		n = b		n = d		n = h		n = g		n = f		FINAL	
	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr
a	0	-	removed																0	-
b	∞	-	5	a			4	c	removed										4	c
c	∞	-	3	a			removed												3	a
d	∞	-					4	c			removed								4	c
e	∞	-	2	a	removed														2	a
f	∞	-											7	h			removed		7	h
g	∞	-									6	d			removed				6	d
h	∞	-			6	e					5	d	removed						5	d
i	∞	-	∞	-	9	e									8	g			8	g

Once we're done, we can follow this chart backwards, starting at `i` to find the actual path. In this case, the smallest weight path has weight 8 and is `a -> c -> d -> g -> i`.

Priority Queue and Runtime

As mentioned earlier, our algorithm used a priority queue, but this data structure can be implemented in different ways.

The runtime of this algorithm depends heavily on the implementation of the priority queue. Assume you have `v` vertices and `e` nodes.

- If you implemented the priority queue with an array, then you would call `remove_min` up to `v` times. Each operation would take `O(v)` time, so you'd spend $O(v^2)$ time in the `remove_min` calls. Additionally, you would update the values of `path_weight` and `previous` at most once per edge, so that's `O(e)` time doing those updates. Observe that `e` must be less than of equal to v^2 since you can't have more edges than there are pairs of vertices. Therefore, the total runtime is $O(v^2)$.
- If you implemented the priority queue with a min heap, then the `remove_min` calls will each take `O(log v)` time (as will inserting and updating a key). We will do one `remove_min` call for each vertex, so that's `O(v log v)` (`v` vertices at `O(log v)` time each). Additionally, on each edge, we might call one update key or insert operation, so that's `O(e log v)`. The total runtime is `O((v + e) log v)`.

Which one is better? Well, that depends. If the graph has a lot of edges, then v^2 will be close to `e`. In this case, you might be better off with the array implementation, as $O(v^2)$ is better than $O((v + v^2) \log v)$. However, if the graph is sparse, then `e` is much less than v^2. In this case, the min heap implementation may be better.

Hash Table Collision Resolution

Essentially any hash table can have collisions. There are a number of ways of handling this.

Chaining with Linked Lists

With this approach (which is the most common), the hash table's array maps to a linked list of items. We just add items to this linked list. As long as the number of collisions is fairly small, this will be quite efficient.

In the worst case, lookup is `O(n)`, where `n` is the number of elements in the hash table. This would only happen with either some very strange data or a very poor hash function (or both).

Chaining with Binary Search Trees

Rather than storing collisions in a linked list, we could store collisions in a binary search tree. This will bring the worst-case runtime to `O(log n)`.

In practice, we would rarely take this approach unless we expected an extremely nonuniform distribution.

Open Addressing with Linear Probing

In this approach, when a collision occurs (there is already an item stored at the designated index), we just move on to the next index in the array until we find an open spot. (Or, sometimes, some other fixed distance, like the `index + 5`.)

If the number of collisions is low, this is a very fast and space-efficient solution.

One obvious drawback of this is that the total number of entries in the hash table is limited by the size of the array. This is not the case with chaining.

There's another issue here. Consider a hash table with an underlying array of size 100 where indexes 20 through 29 are filled (and nothing else). What are the odds of the next insertion going to index 30? The odds are 10% because an item mapped to any index between 20 and 30 will wind up at index 30. This causes an issue called *clustering*.

Quadratic Probing and Double Hashing

The distance between probes does not need to be linear. You could, for example, increase the probe distance quadratically. Or, you could use a second hash function to determine the probe distance.

Rabin-Karp Substring Search

The brute force way to search for a substring S in a larger string B takes `O(s(b-s))` time, where `s` is the length of S and `b` is the length of B. We do this by searching through the first `b - s + 1` characters in B and, for each, checking if the next `s` characters match S.

The Rabin-Karp algorithm optimizes this with a little trick: if two strings are the same, they must have the same hash value. (The converse, however, is not true. Two different strings can have the same hash value.)

Therefore, if we efficiently precompute a hash value for each sequence of `s` characters within B, we can find the locations of S in `O(b)` time. We then just need to validate that those locations really do match S.

For example, imagine our hash function was simply the sum of each character (where space = 0, `a` = 1, `b` = 2, and so on). If S is `ear` and B = `doe are hearing me`, we'd then just be looking for sequences where the sum is 24 (e + a + r). This happens three times. For each of those locations, we'd check if the string really is `ear`.

char:	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	24	20	6	19	24	23	13	13	14	24	28	41	30	21	20	18		

If we computed these sums by doing `hash('doe')`, then `hash('oe ')`, then `hash('e a')`, and so on, we would still be at `O(s(b-s))` time.

Instead, we compute the hash values by recognizing that `hash('oe ') = hash('doe') - code('d') + code(' ')`. This takes `O(b)` time to compute all the hashes.

You might argue that, still, in the worst case this will take `O(s(b-s))` time since many of the hash values could match. That's absolutely true—for this hash function.

In practice, we would use a better *rolling hash function*, such as the Rabin fingerprint. This essentially treats a string like `doe` as a base 128 (or however many characters are in our alphabet) number.

```
hash('doe') = code('d') * 128^2 + code('o') * 128^1 + code('e') * 128^0
```

This hash function will allow us to remove the `d`, shift the `o` and `e`, and then add in the space.

```
hash('oe ') = (hash('doe') - code('d') * 128^2) * 128 + code(' ')
```

This will considerably cut down on the number of false matches. Using a good hash function like this will give us expected time complexity of `O(s + b)`, although the worst case is `O(sb)`.

Usage of this algorithm comes up fairly frequently in interviews, so it's useful to know that you can identify substrings in linear time.

AVL Trees

An AVL tree is one of two common ways to implement tree balancing. We will only discuss insertions here, but you can look up deletions separately if you're interested.

Properties

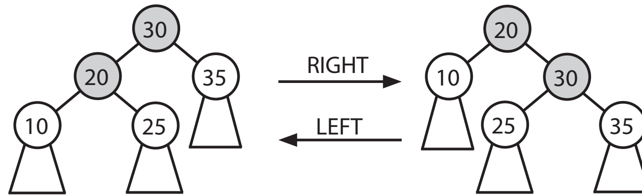
An AVL tree stores in each node the height of the subtrees rooted at this node. Then, for any node, we can check if it is height balanced: that the height of the left subtree and the height of the right subtree differ by no more than one. This prevents situations where the tree gets too lopsided.

```
balance(n) = n.left.height - n.right.height
-1 <= balance(n) <= 1
```

Inserts

When you insert a node, the balance of some nodes might change to -2 or 2. Therefore, when we "unwind" the recursive stack, we check and fix the balance at each node. We do this through a series of rotations.

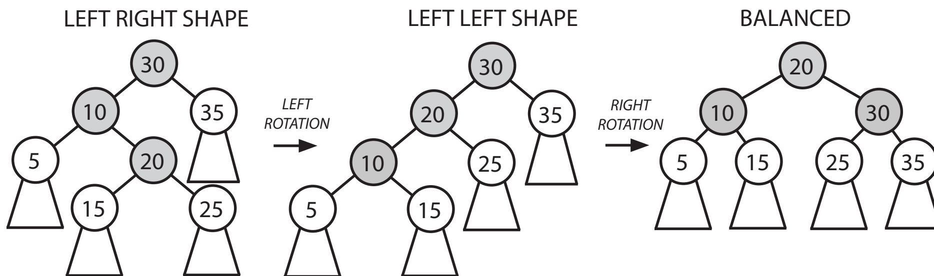
Rotations can be either left or right rotations. The right rotation is an inverse of the left rotation.



Depending on the balance and where the imbalance occurs, we fix it in a different way.

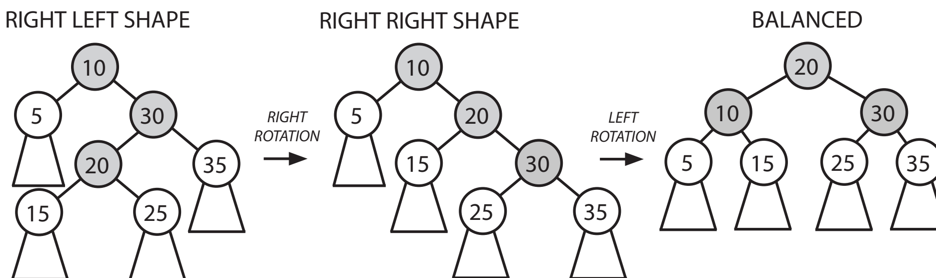
- *Case 1: Balance is 2.*

  In this case, the left's height is two bigger than the right's height. If the left side is larger, the left subtree's extra nodes must be hanging to the left (as in LEFT LEFT SHAPE) or hanging to the right (as in LEFT RIGHT SHAPE). If it looks like the LEFT RIGHT SHAPE, transform it with the rotations below into the LEFT LEFT SHAPE then into BALANCED. If it looks like the LEFT LEFT SHAPE already, just transform it into BALANCED.



- *Case 2: Balance is -2.*

  This case is the mirror image of the prior case. The tree will look like either the RIGHT LEFT SHAPE or the RIGHT RIGHT SHAPE. Perform the rotations below to transform it into BALANCED.



In both cases, "balanced" just means that the `balance` of the tree is between -1 and 1. It does not mean that the `balance` is 0.

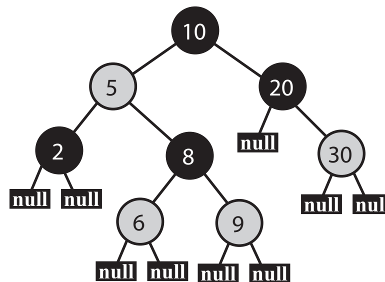
We recurse up the tree, fixing any imbalances. If we ever achieve a balance of 0 on a subtree, then we know that we have completed all the balances. This portion of the tree will not cause another, higher subtree to have a balance of -2 or 2. If we were doing this non-recursively, then we could break from the loop.

Red-Black Trees

Red-black trees (a type of self-balancing binary search tree) do not ensure quite as strict balancing, but the balancing is still good enough to ensure O(log N) insertions, deletions, and retrievals. They require a bit less memory and can rebalance faster (which means faster insertions and removals), so they are often used in situations where the tree will be modified frequently.

Red-black trees operate by enforcing a quasi-alternating red and black coloring (under certain rules, described below) and then requiring every path from a node to its leaves to have the same number of black nodes. Doing so leads to a reasonably balanced tree.

The tree below is a red-black tree (where the red nodes are indicated with gray):



Properties

1. Every node is either red or black.
2. The root is black.
3. The leaves, which are NULL nodes, are considered black.
4. Every red node must have two black children. That is, a red node cannot have red children (although a black node can have black children).
5. Every path from a node to its leaves must have the same number of black children.

Why It Balances

Property #4 means that two red nodes cannot be adjacent in a path (e.g., parent and child). Therefore, no more than half the nodes in a path can be red.

Consider two paths from a node (say, the root) to its leaves. The paths must have the same number of black nodes (property #5), so let's assume that their red node counts are as different as possible: one path contains the minimum number of red nodes and the other one contains the maximum number.

- Path 1 (Min Red): The minimum number of red nodes is zero. Therefore, path 1 has b nodes total.
- Path 2 (Max Red): The maximum number of red nodes is b, since red nodes must have black children and there are b black nodes. Therefore, path 2 has 2b nodes total.

Therefore, even in the most extreme case, the lengths of paths cannot differ by more than a factor of two. That's good enough to ensure an O(log N) find and insert runtime.

If we can maintain these properties, we'll have a (sufficiently) balanced tree—good enough to ensure O(log N) insert and find, anyway. The question then is how to maintain these properties efficiently. We'll only discuss insertion here, but you can look up deletion on your own.

Insertion

Inserting a new node into a red-black tree starts off with a typical binary search tree insertion.

- New nodes are inserted at a leaf, which means that they replace a black node.
- New nodes are always colored red and are given two black leaf (NULL) nodes.

Once we've done that, we fix any resulting red-black property violations. We have two possible violations:

- Red violations: A red node has a red child (or the root is red).
- Black violations: One path has more blacks than another path.

The node inserted is red. We didn't change the number of black nodes on any path to a leaf, so we know that we won't have a black violation. However, we might have a red violation.

In the special case that where the root is red, we can always just turn it black to satisfy property 2, without violating the other constraints.

Otherwise, if there's a red violation, then this means that we have a red node under another red node. Oops!

Let's call N the current node. P is N's parent. G is N's grandparent. U is N's uncle and P's sibling. We know that:

- N is red and P is red, since we have a red violation.
- G is definitely black, since we didn't *previously* have a red violation.

The unknown parts are:

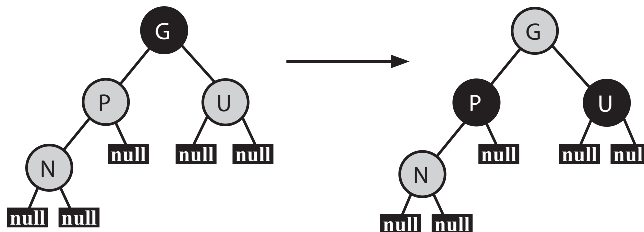
- U could be either red or black.
- U could be either a left or right child.
- N could be either a left or right child.

By simple combinatorics, that's eight cases to consider. Fortunately some of these cases will be equivalent.

- **Case 1: U is red.**

  It doesn't matter whether U is a left or right child, nor whether P is a left or right child. We can merge four of our eight cases into one.

  If U is red, we can just toggle the colors of P, U, and G. Flip G from black to red. Flip P and U from red to black. We haven't changed the number of black nodes in any path.



  However, by making G red, we might have created a red violation with G's parent. If so, we recursively apply the full logic to handle a red violation, where this G becomes the new N.

  Note that in the general recursive case, N, P, and U may also have subtrees in place of each black NULL (the leaves shown). In Case 1, these subtrees stay attached to the same parents, as the tree structure remains unchanged.

- **Case 2: U is black.**

  We'll need to consider the configurations (left vs. right child) of N and U. In each case, our goal is to fix up the red violation (red on top of red) without:

  » Messing up the ordering of the binary search tree.

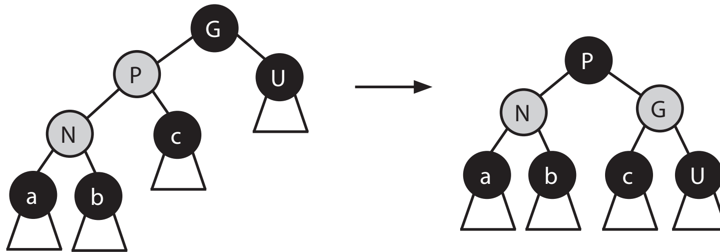
  » Introducing a black violation (more black nodes on one path than another).

  If we can do this, we're good. In each of the cases below, the red violation is fixed with rotations that maintain the node ordering.

  Further, the below rotations maintain the exact number of black nodes in each path through the affected portion of the tree that were in place beforehand. The children of the rotating section are either NULL leaves or subtrees that remain internally unchanged.

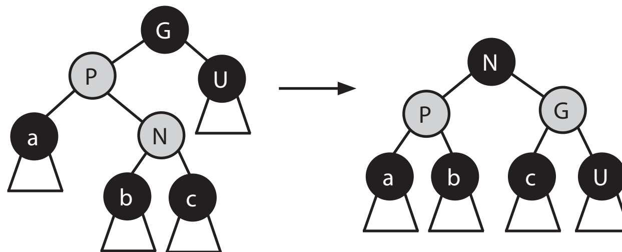
  Case A: N and P are both left children.

  We resolve the red violation with the rotation of N, P, and G and the associated recoloring shown below. If you picture the in-order traversal, you can see the rotation maintains the node ordering (`a <= N <= b <= P <= c <= G <= U`). The tree maintains the same, equal number of black nodes in the path down to each subtree a, b, c, and U (which may all be NULL) .



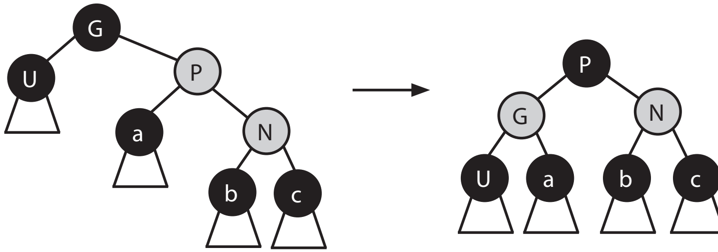
  Case B: P is a left child, and N is a right child.

  The rotations in Case B resolve the red violation and maintain the in-order property: `a <= P <= b <= N <= c <= G <= U`. Again, the count of the black nodes remains constant in each path down to the leaves (or subtrees).



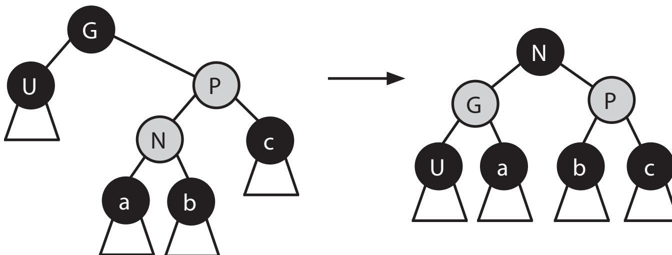
Case C: N and P are both right children.

This is a mirror image of case A.



Case D: N is a left child, and P is a right child.

This is a mirror image of case B.



In each of Case 2's subcases, the middle element by value of N, P, and G is rotated to become the root of what was G's subtree, and that element and G swap colors.

That said, do not try to just memorize these cases. Rather, study why they work. How does each one ensure no red violations, no black violations, and no violations of the binary search tree property?

▸ MapReduce

MapReduce is used widely in system design to process large amounts of data. As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

1. The system splits up the data across different machines.
2. Each machine starts running the user-provided `Map` program.
3. The `Map` program takes some data and emits a `<key, value>` pair.
4. The system-provided `Shuffle` process reorganizes the data so that all <key, value> pairs associated with a given key go to the same machine, to be processed by `Reduce`.
5. The user-provided `Reduce` program takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the `Reduce` program for more reducing.

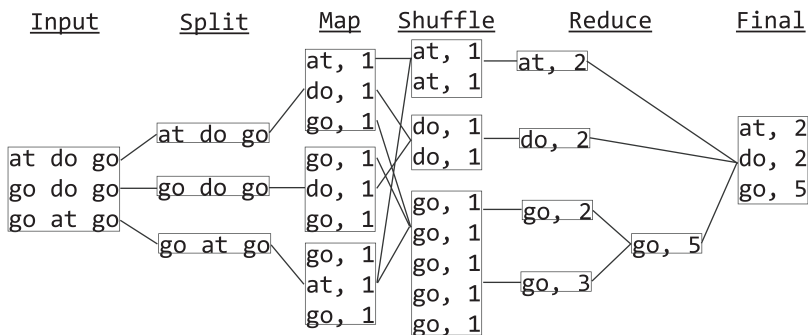
The typical example of using MapReduce—basically the "Hello World" of MapReduce—is counting the frequency of words within a set of documents.

Of course, you could write this as a single function that reads in all the data, counts the number of times each word appears via a hash table, and then outputs the result.

MapReduce allows you to process the document in parallel. The `Map` function reads in a document and emits just each individual word and the count (which is always 1). The `Reduce` function reads in keys (words) and associated values (counts). It emits the sum of the counts. This sum could possibly wind up as input for another call to `Reduce` on the same key (as shown in the diagram).

```
1   void map(String name, String document):
2     for each word w in document:
3       emit(w, 1)
4
5   void reduce(String word, Iterator partialCounts):
6     int sum = 0
7     for each count in partialCounts:
8       sum += count
9     emit(word, sum)
```

The diagram below shows how this might work on this example.



Here's another example: You have a list of data in the form {City, Temperature, Date}. Calculate the average temperature in each city every year. For example {(2012, Philadelphia, 58.2), (2011, Philadelphia, 56.6), (2012, Seattle, 45.1)}.

- **Map**: The Map step outputs a key value pair where the key is `City_Year` and the value is `(Temperature, 1)`. The '1' reflects that this is the average temperature out of one data point. This will be important for the `Reduce` step.
- **Reduce**: The Reduce step will be given a list of temperatures that correspond with a particular city and year. It must use these to compute the average temperature for this input. You cannot simply add up the temperatures and divide by the number of values.

  To see this, imagine we have five data points for a particular city and year: 25, 100, 75, 85, 50. The Reduce step might only get some of this data at once. If you averaged {75, 85} you would get 80. This might end up being input for another Reduce step with 50, and it would be a mistake to just naively average 80 and 50. The 80 has more weight.

  Therefore, our Reduce step instead takes in {(80, 2), (50, 1)}, then sums the *weighted* temperatures. So it does 80 * 2 + 50 * 1 and then divides by (2 + 1) to get an average temperature of 70. It then emits (70, 3).

  Another Reduce step might reduce {(25, 1), (100, 1)} to get (62.5, 2). If we reduce this with (70, 3) we get the final answer: (67, 5). In other words, the average temperature in this city for this year was 67 degrees.

We could do this in other ways, too. We could have just the city as the key, and the value be (Year, Temperature, Count). The Reduce step would do essentially the same thing, but would have to group by Year itself.

In many cases, it's useful to think about what the Reduce step should do first, and then design the Map step around that. What data does Reduce need to have to do its job?

Additional Studying

So, you've mastered this material and you want to learn even more? Okay. Here are some topics to get you started:

- **Bellman-Ford Algorithm**: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.
- **Floyd-Warshall Algorithm**: Finds the shortest paths in a weighted graph with positive or negative weight edges (but no negative weight cycles).
- **Minimum Spanning Trees**: In a weighted, connected, undirected graph, a spanning tree is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with minimum weight. There are various algorithms to do this.
- **B-Trees:** A self-balancing search tree (not a binary search tree) that is commonly used on disks or other storage devices. It is similar to a red-black tree, but uses fewer I/O operations.
- **A***: Find the least-cost path between a source node and a goal node (or one of several goal nodes). It extends Dijkstra's algorithm and achieves better performance by using heuristics.
- **Interval Trees**: An extension of a balanced binary search tree, but storing intervals (low -> high ranges) instead of simple values. A hotel could use this to store a list of all reservations and then efficiently detect who is staying at the hotel at a particular time.
- **Graph coloring**: A way of coloring the nodes in a graph such that no two adjacent vertices have the same color. There are various algorithms to do things like determine if a graph can be colored with only K colors.
- **P, NP, and NP-Complete**: P, NP, and NP-Complete refer to classes of problems. P problems are problems that can be quickly solved (where "quickly" means polynomial time). NP problems are those where, given a solution, the solution can be quickly verified. NP-Complete problems are a subset of NP problems that can all be reduced to each other (that is, if you found a solution to one problem, you could tweak the solution to solve other problems in the set in polynomial time).

  It is an open (and very famous) question whether P = NP, but the answer is generally believed to be no.
- **Combinatorics and Probability**: There are various things you can learn about here, such as random variables, expected value, and n-choose-k.
- **Bipartite Graph**: A bipartite graph is a graph where you can divide its nodes into two sets such that every edge stretches across the two sets (that is, there is never an edge between two nodes in the same set). There is an algorithm to check if a graph is a bipartite graph. Note that a bipartite graph is equivalent to a graph that can be colored with two colors.
- **Regular Expressions**: You should know that regular expressions exist and what they can be used for (roughly). You can also learn about how an algorithm to match regular expressions would work. Some of the basic syntax behind regular expressions could be useful as well.

There is of course a great deal more to data structures and algorithms. If you're interested in exploring these topics more deeply, I recommend picking up the hefty *Introduction to Algorithms* ("CLRS" by Cormen, Leiserson, Rivest and Stein) or *The Algorithm Design Manual* (by Steven Skiena).