Emma Bostian

De-Coding the Technical Interview Process

Tips and resources for nailing your next technical interview

Table Of Contents

CHAPTER 01

Introduction

Who This Book Is For

This book is for anyone who has ever felt like they didn't belong in the technical industry, those who have been told they're not good enough, and anyone who was recently laid off.

You are important. You are smart. And you can do this.

Why I Wrote This Book

Throughout my college degree and the first several years of my career, I was constantly bombarded with people telling me I wasn't good enough and that I wouldn't make it as a software engineer. For a long time, I let their words deplete my self-confidence. But somewhere along the way, I decided to pick myself up, study my ass off, and prove them all wrong.

I began my coding career in 2012 during my Sophomore year of college. I attended Siena College in Loudonville, New York, and after declaring a computer science major with a business minor, I was on my way to becoming a software engineer.

During my Junior year of college, I completed a six-month internship with IBM in Poughkeepsie, New York, where I automated the installation of WebSphere Application Server on z/OS using Python. During my internship, I met my future manager, who would bring me down to Austin, Texas.

After graduation, I picked up and moved to the south, where I joined the IBM Spectrum Control team. In my first year and a half with IBM, I focused on front-end development. It was a difficult few years as I studied Java and back-end development throughout college and was duly unprepared for the real world.

I focused my first year and a half on web accessibility. Still, I wanted something a bit more alluring than enterprise storage systems, so when I was presented with the opportunity to join the IBM Systems & Transformation design team, I jumped.

I cherished my time working with the design team and learned many skills that aided my career. I learned UX and visual design skills, I worked with IBM Support on a Salesforce support solution, and I worked with WordPress and PHP. But my most notable, and my favorite project, was with the quantum computing team. I single-handedly developed the IBMQ Network website using Vue.js, and that felt damn good.

Three years after beginning my career at IBM, I resigned, sold everything, and flew my two cats out to Germany, where I started a new role with LogMeIn. In my first year with LogMeIn, I worked on the GoToMeeting product team, but I've worked on building a design system from the ground up for the past year and a half. This was the same year my career took off. I started my blog, spoke at numerous conferences across Europe, the UK, and the United States, created three courses (two with LinkedIn Learning and one with Frontend Masters), started the Ladybug Podcast, joined JS Party, and more.

Throughout the past five years, I've had countless technical interviews.

Most of them were flamingly horrible, but most recently, I passed my Google technical interviews and accepted my dream job with Spotify in Stockholm, Sweden. I cannot believe I'm sitting here writing that I successfully passed a technical interview with Spotify. I am utterly proud of myself.

This book is the culmination of my learnings throughout the past five years.

When studying for my interviews, I struggled immensely to find resources that would help me achieve my dream job. Even with a computer science degree, I had trouble grasping algorithms and data structures concepts. I read Cracking The Coding Interview by Gayle Laakmann McDowell several times, each time frustrated that the solutions and concepts were written in Java.

Because I couldn't find the resources I needed to succeed, I wrote my own.

Front-End Focused

The code examples throughout this book are in JavaScript. While many of the concepts we'll discuss will be conceptual and relevant for all software developers, I am a JavaScript developer and will dive a bit deeper into web technologies.

This book will not be a deep-dive into HTML, CSS, or JavaScript but will touch on all the topics you should study before a technical interview. However, I will provide resources to learn more about these technologies and concepts.

CHAPTER 02

The Interview Process

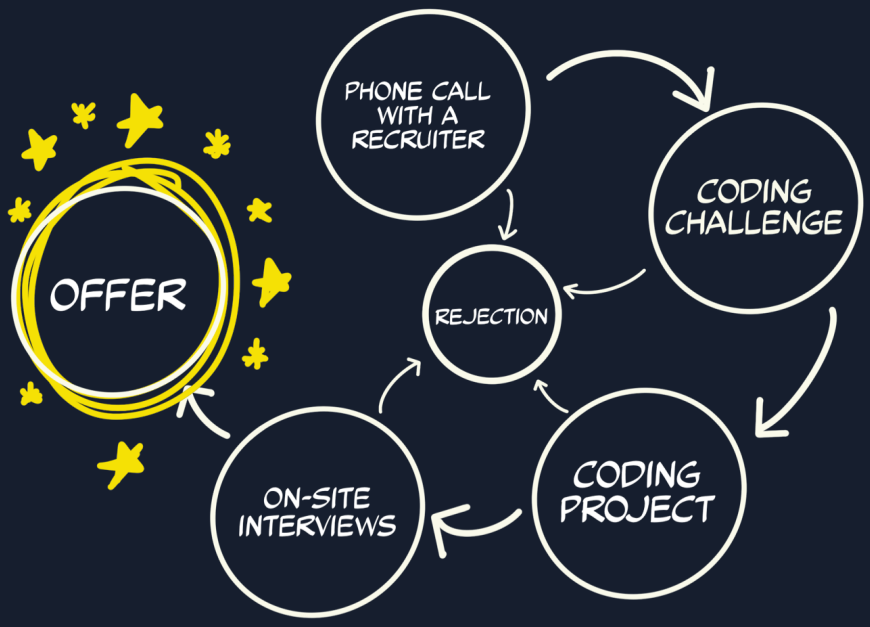
Overview

First of all, let's address the elephant in the room: the interview process is anxiety-inducing, exhausting, and stressful.

I struggle immensely with technical interviews and I know many of you do as well. It's okay if you're anxious or if you receive a rejection. You're not alone.

The interview process will change from company to company but at the bare minimum you can expect a recruiter phone interview and a technical phone interview.

The interview process has several steps. Let's discuss each of these steps in-depth.



Recruiter Phone Interview

During the phone interview the recruiter will tell you a bit more about the role and the interview process. Don't take this interview lightly.

Each step in the interview process is important and counts towards your overall performance.

Here are some tips for the recruiter phone interview.

Read Up On The Role And The Company

If you're applying for a job the recruiter will expect you to have a general understanding of the role you've applied for. So if you're mass-applying to every role you encounter, keep the applications organized.

When applying for multiple roles I like to create a Notion document with a table containing the company name, recruiter name and email address, a link to the application, and other pertinent details.

Before the recruiter phone interview, do some research on the company to see what their values are and what they're currently working on. Showing that you've done some investigation will impress your recruiter.

Be on time

Did you know that every culture has a different perception of time? I didn't either until I read The Culture Map by Erin Meyer. But regardless of whether or not it's appropriate to join a meeting five minutes late in your home culture, you should always be on-time for a job interview.

If you're conducting your recruiter call through an online video meeting, join a few minutes early to sort out any technical issues that may arise.

If you're conducting the interview through a phone call, ensure your phone is not on do not disturb mode, and you're in a quiet place.

Have A Few Questions Prepared

Towards the end of your meeting, the recruiter will ask if you have any questions. Throughout the interview process, if an interviewer, recruiter, or hiring manager asks if you have any questions, say yes.

You should have one or two questions prepared that are relevant to the role. They can include questions like:

- What is the culture like in the office?
- How is the diversity at your company?
- Are there opportunities for continuing education such as a conference allowance or subscriptions to learning platforms?
- How is the work/life balance?

Having questions prepared shows your interest in the role and may set you apart from other candidates.

Coding Challenge

If your recruiter phone interview goes well you'll move on to the technical phone interview. During this call you'll pair up with a developer who will give you a coding challenge.

I've had coding challenges where I was presented with two or three smaller questions, but I've also had coding challenges where it was one large question. The format of this challenge will vary from company to company but the skills you'll need are generally the same.

If you're applying for a front-end developer position you should focus on HTML, CSS, and JavaScript for the phone coding challenges (based on my experiences).

The questions are generally centered around your knowledge of DOM (Document Object Model) manipulation with JavaScript (your ability to add, remove, and alter HTML elements in the DOM).

You may have to do some CSS coding as well so don't forget to study the basics like specificity (the rules of how our styles are applied to elements), positioning, display properties (block versus inline), and responsive layouts with Flexbox and Grid.

You should also be comfortable with the basics of web accessibility. This include using semantic HTML (elements such as `<nav>` or `<header>` versus `<div>` and `<span>`) which allow screen readers to understand and parse the HTML document body as well as ARIA (accessible rich internet applications) which is a series of HTML attributes you can add to a DOM node, or HTML element, to provide more information about an element (i.e. is your modal visible or hidden).

You'll also want to feel comfortable writing code with JavaScript. It can be extremely nerve-wracking to memorize syntax for advanced topics like promises and async/await (trust me, I've bombed many of these questions), so be patient with yourself. If you can't remember the syntax, be honest. Honesty is always the best policy.

In my experience the coding challenges aim to test the breadth of your web development knowledge whilst the on-site interviews and/or coding projects aim to test vertical data structures and algorithms knowledge, however take this with a grain of salt. Every company will have a different interview process. You can ask your recruiter for more information about the topics you should study up on.

You may also receive some direct technical questions such as:

- Can you define a promise?
- What is a closure?
- How would you visually hide an element in the UI so it's still accessible for screen readers?
- Why do we use alt tags for images?
- How can we make our web applications more performant?

Sample Coding Challenge Questions

Here are some coding challenge questions. I encourage you to try solving them before referring to the solutions at the end of the book.

- Find the mammals
- Display sidebar
- Find the mammals (version 2)
- The social network

Find The Mammals

Given an array containing a list of animal objects, return a new array containing only the animals which are mammals. Each animal object contains an animal name (i.e. dog or snake) and a value, mammal, which is a boolean indicating whether the animal is a mammal.

What Skills Does This Test?

- Iterating through an array
- Accessing properties of an object

```
animals = [
  {
    type: 'Dog',
    mammal: true,
  },
  {
    type: 'Snake',
    mammal: false,
  },
  {
    type: 'Cheetah',
    mammal: true,
  },
]
```

See chapter 10 for the solution!

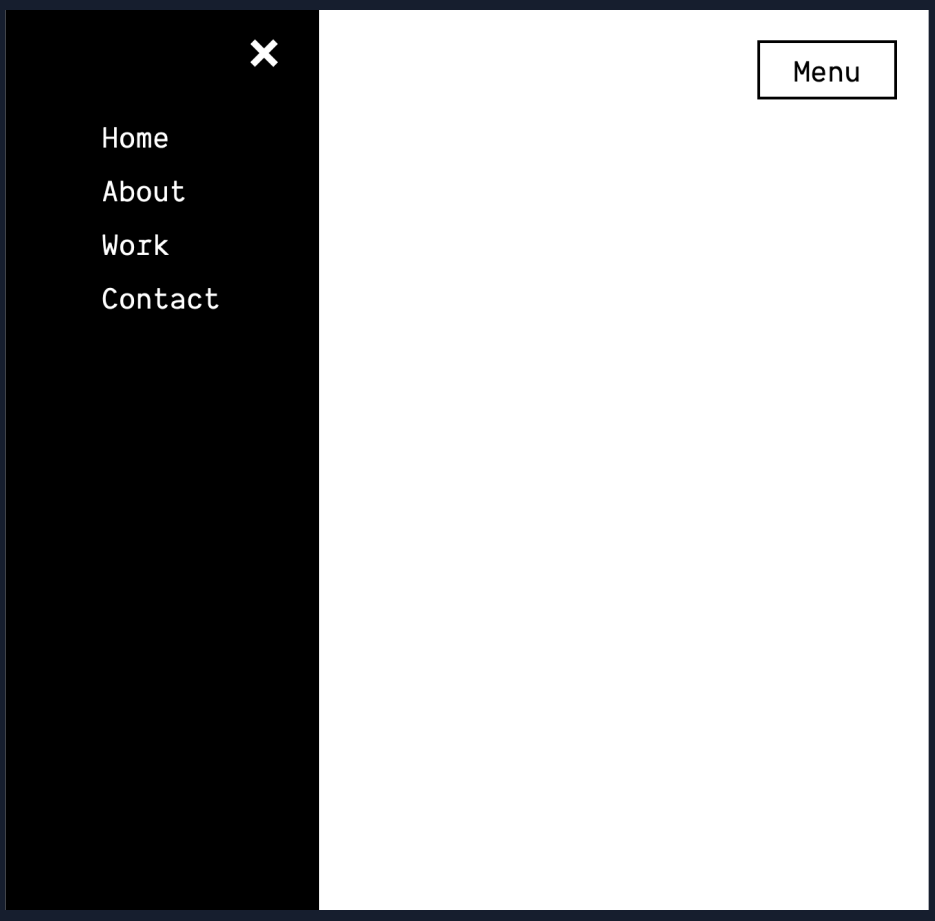
Display Sidebar

Create a mobile navigation which is hidden off-screen to start and when a “Show menu” button is clicked, the menu animates in from the left side of the viewport.

When a “Close” button is clicked the menu will slide back off the screen. You should only use vanilla JavaScript and CSS animations for this challenge.

What Skills Does This Test?

- DOM manipulation
- CSS animations
- Event handlers



See chapter 10 for the solution!

Find The Mammals (Version 2)

This question is the same prompt as the first question but instead of an array of animal objects, you have to query a DOM node, a section with a class of ‘all-animals’, which contains a list of animals.

Each animal will be encapsulated in a div with a class of ‘animal’. Inside this div are three nodes:

- An h2 indicating the type of animal
- A paragraph containing a child span with a class of ‘mammal-value’ that indicates whether the animal is a mammal

You should find all mammal nodes and append them as children of the section with a class of ‘only-mammals’.

What Skills Does This Test?

- DOM manipulation
- Iterating through nested DOM elements

(See the next page for a code snippet.)

Find The Mammals (Version 2)

```
<h1>All animals</h1>
<section class='all-animals'>
  <div class='animal'>
    <h2>Dog</h2>
    <p>mammal: <span class='mammal-value'>true</p>
  </div>
  <div class='animal'>
    <h2>Snake</h2>
    <p>mammal: <span class='mammal-value'>false</p>
  </div>
  <div class='animal'>
    <h2>Cheetah</h2>
    <p>mammal: <span class='mammal-value'>true</p>
  </div>
  <div class='animal'>
    <h2>Turtle</h2>
    <p>mammal: <span class='mammal-value'>false</p>
  </div>
  <div class='animal'>
    <h2>Frog</h2>
    <p>mammal: <span class='mammal-value'>false</p>
  </div>
  <div class='animal'>
    <h2>Cat</h2>
    <p>mammal: <span class='mammal-value'>true</p>
  </div>
  <div class='animal'>
    <h2>Badger</h2>
    <p>mammal: <span class='mammal-value'>true</p>
  </div>
</section>
<h1>Mammals</h1>
<section id='only-mammals'></section>
```

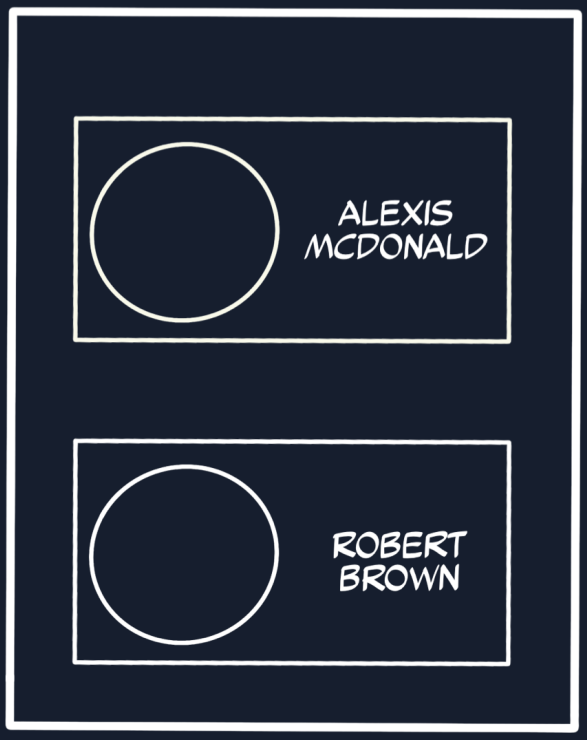
See chapter 10 for the solution!

The Social Network

Using the Random User Generator API, query for a list of 10 users and for each user display their profile photo and name.

What Skills Does This Test?

- DOM manipulation
- Asynchronous programming



See chapter 10 for the solution!

Coding Project

Some companies require a coding project in lieu of an on-site technical interview or a coding challenge. Others will require it in addition to technical interviews and coding challenges.

I love coding projects as they allow me to showcase my skills in an environment more conducive to how I work day-to-day.

A coding project is an encapsulated prompt that asks you to build a small application. Some companies might provide you with a few options to choose from that test different skills. Others might provide you with one prompt.

Coding Project Controversy

While I personally enjoy coding projects, many developers don't. While the intent behind coding projects may be innocent, they may put some candidates in a disadvantaged position.

Most candidates are not interviewing for fun; often candidates interview because they have lost their job or are currently in an unsafe work environment. Thus their time is valuable. Additionally, many candidates have families to care for and other commitments after working their day jobs so they cannot afford to spend upwards of four hours working on a coding project.

If you are going to require a coding project there are a few things you can do to ensure a fair interview process:

- Scope your projects appropriately. Your projects should take no more than a few hours and should have a finite scope. Your candidates should be clear about expected deliverables so they don't have to spend time communicating with the recruiter.
- Give your candidate a reasonable amount of time to complete the task. Don't expect your candidates to submit their project in 24-hours time. Give them a week or two weeks to submit their project. This will alleviate some of the stress your candidates are feeling.
- Your project should not contain real problems your company is facing. This is free labor (unless you're paying your candidates) and this is highly unethical. The project should contain a scoped problem that will demonstrate skills needed for the job.
- Pay your candidates for their time (if possible). If you're going to require hours of work from a candidate prior to extending them a job offer, you should be paying them for their time. I understand not all companies are in the financial position to be able to pay their candidates. If this is the case you should think about whether a coding project is a good fit for your interview process.

Additional Deliverables

Whenever I complete a coding project there are a few things I always include or do. If you have a bit of extra time, adding one or more of these things can help differentiate you from the other candidates, but by no means are these necessary to receive a job offer!

Clarify Requirements

Before starting a project, clarify the requirements. For example, does the company want you to use a specific technology stack, or are you free to use any JavaScript framework or library? I have made the mistake of assuming vanilla JavaScript was acceptable and only after receiving a rejection did I come to find out they wanted to see use of a popular framework or library like React.

A good coding project prompt will explicitly state whether they expect a particular framework or library, but some don't. Thus, it's always a good idea to ask!

Thoroughly Document

The project reviewer should be able to quickly and easily locate installation instructions for cloning and running your application. You can include this as a README, a PDF document, or a website, however you should clarify how the company prefers to receive these deliverables.

They may want it on a private GitHub repository. They may want it as a zip file sent through email. Or they may want it hosted as a website with a password. Be sure to submit in the proper format with complete documentation.

Create User Flows

I always sketch out user flows for any application I create. This is more of a UX design activity, however clarifying the information architecture (the hierarchy of information throughout the application) and the different flows the user can take (like sign in or edit preferences) will allow you to design your application to fit your users' needs.

You can include user flows or information architecture documents within your final deliverables. They will showcase your conscious attention to the end user.

You can use tools like Mural or Figma to complete this, or simply sketch it out on a piece of paper, take a photo and upload it.

Add Additional Enhancements

Including a few additional enhancements or features, if you have extra time, is always a great way to impress a potential employer. For example if you're asked to build a food tracker application you might want to add in the ability to contact your food delivery person through text in order to provide real-time updates.

The feature doesn't have to be fully-functional (it can be hard-coded data) but adding these enhancements shows your attention to detail.

If you don't have time to implement the features, you can list them with a high-level description of what the features do and why they enhance the user experience.

Note Areas For Improvement

I always include areas of improvement in my coding projects. They show that a candidate is self-aware and recognizes areas that can be improved. This is a great way to showcase your knowledge without having to spend hours implementing features or refactoring code.

For example if you decide to use a UI framework like Material Design or Bootstrap in your project to ensure you have a consistent and accessible UI, but it comes at the expense of application performance and not showcasing your CSS skills, this is one area you can note as a future improvement.

Tips For Nailing Your Coding Project

Here are a few tips for nailing your coding project.

Remove Code Comments

Once completing your project you should wait several hours (or better yet sleep on it) and then take a fresh read through your code. Are there any lingering code comments? Can you remove them?

Refactor Non-Performant Code

If there are areas of your code that aren't performant (a nested for-loop for example), try to optimize them. Show your reviewers that you prioritize performance.

Test For Accessibility

Before submitting, run your application through an accessibility tool like Lighthouse or Axe. Fixing small accessibility pitfalls and including a small paragraph explaining your mindful care of accessibility will set you apart from the crowd.

Design A Logical Project Architecture

Before jumping straight into coding take a moment to think through your project architecture. How do you want to organize your project files?

Do you want to use camelCase, kebab-case, or TitleCase for your file names? Do you want to include the CSS files or Sass files with the component file or in a separate styles folder? What are the benefits and drawbacks of each approach? Be consistent with your naming conventions and architect your code logically.

Sample Coding Projects

Here are two examples of coding projects you might be asked to build.

I won't be divulging specific interview questions or coding projects I've received from companies, but the following projects are examples of scoped applications that demonstrate various skills and would be a good candidate for a coding project.

Completing a few example projects will help you prepare for a coding project. It will help you determine the tools you're comfortable with for building an application.

To-Do Application

Before you roll your eyes, try to complete this challenge. Think about the enhancements you can make. Just because it's an over-used project doesn't mean you shouldn't practice.

Your task is to create a to-do application. Here are the system requirements. Users must be able to:

- Add tasks
- Complete tasks (mark them as done)
- Edit tasks
- Delete tasks
- See how many tasks are remaining

You can use any technology stack to complete this task. The UI (user interface) might look something like this.

A few additional enhancements might include

- Adding animations when tasks are deleted
- The ability to favorite a task and have it appear at the top of the list
- The ability to create groups or categories of tasks (i.e. home, work)
- Light and dark theme (for accessibility)

Your deliverables should include

- Your application should be hosted with a password to enter (you can do this with Netlify, their documentation is amazing).
- Your code should be minified and hosted in a private GitHub repository.
- You should provide a README containing instructions on how to install/clone and start your application locally as well as the password for the live site.

Food Delivery Application

Your task is to create a food delivery application. Here are the system requirements.

Users must be able to:

- See a list of five meal options
- See the prices of each meal option
- Add a meal to the cart
- Change the quantity of meals in the cart
- Remove a meal from the cart
- See their total bill
- “Check out” - this doesn’t have to process payment, just simply display a message stating their order for X price has been received and will arrive in X minutes.

There is no UI provided for this challenge so design your own! I recommend sketching out a low-fidelity mockup on paper or a higher-fidelity mockup in Figma,Sketch, or Adobe Illustrator before starting.

You don’t need design skills to sketch out a UI!

A few additional enhancements might include

- Having restaurants the user can navigate through
- Showing which meal options are vegetarian or vegan

- Show the wait time for each restaurant
- Filter by restaurant or meal type

Your deliverables should include

- Your application should be hosted with a password to enter (you can do this with Netlify, their documentation is amazing).
- Your code should be minified and hosted in a private GitHub repository.
- You should provide a README containing instructions on how to install/clone and start your application locally as well as the password for the live site.

On-Site Interviews

Not all companies require an on-site interview, however many of the larger companies do. In my experience the on-site interviews are the most nerve-wracking; you're thrown into four or five interviews each between 45 and 60 minutes. Additionally, you have more to think about than just coding.

You have to think about what clothing to wear and how to get to the building where the interview will be conducted. There are simply more variables.

There are several types of interviews that you may have during an on-site visit. Your recruiter can provide you with more detailed information about the individual interviews but here are a few interview types you may encounter.

- Data structures & algorithms interviews
- Front-end interviews
- Process & communication interviews
- Manager / team matching interviews

Let's take a look at each of these a bit more.

The Data Structures & Algorithms Interview

The data structures and algorithms interviews give many candidates anxiety; they're not something that a front-end developer would necessarily encounter in their day-to-day programming activities.

We live in a time where you don't need a computer science degree to get a development job. People take many paths to join the tech industry: bootcamps, related college degrees, unrelated college degrees, self-teaching.

But the data structures and algorithms interviews test knowledge that is primarily taught during a computer science degree. Thus many candidates are left cramming the days or weeks leading up to an interview and this is extraordinarily stressful.

In the data structures and algorithms section I'll cover a few of these topics more in-depth. But these will be the general topics you'll want to study up on.

I'll link some of my favorite courses and books for learning about data structures and algorithms in the resources at the end of this book. But here are the general topics you should study for this type of interview.

Potential Topics Covered

Data structures

- Stacks
- Queues
- Linked Lists
- Graphs
- Trees
- Tries

Sorting Algorithms

- Merge sort
- Quick sort
- Insertion sort

Searching Algorithms

- Binary search
- Depth-first search
- Breadth-first search
- Tree traversals

Concepts

- Big-O Notation
- Recursion

Front-End Interviews

Front-end interviews are the interviews I tend to struggle with the most. These interviews cover web technologies and the questions you'll receive will likely test practical situations you may encounter during your day-to-day responsibilities.

Here are some areas you will want to study up on for a front-end interview. For a comprehensive list, see the study guides at the back of the book.

Potential Topics Covered

HTML

- Semantic HTML
- Accessibility / ARIA

CSS

- Specificity
- Pseudo-elements / pseudo-selectors
- Position
- Display
- Media queries
- Animations

JavaScript

- Data structures (i.e. maps, sets, symbols, arrays)
- Closures
- Asynchronous programming
- DOM manipulation
- Event delegation

The Internet

- TCP/IP
- CORS
- Performance

UX / Visual Design

- Information architecture
- User flows
- UX heuristics

Sample On-Site Interview Question

Here is an example of on-site interview question you may be asked.

Infinite Scroll

How would you design and build an application which is a stream of continuously loading images?

Some areas you might want to touch on:

- Image performance: What image type should we store our images to reduce bundle size? Webp is a newer image format that is 26% smaller than PNG so perhaps it's a great choice for our application.
- Lazy loading: Lazy loading is an optimization technique where more images are loaded as the user scrolls down the page (as opposed to loading 1,000 images on first page load). Perhaps as the user nears the end of the web page (500px to the end of the page perhaps) we make another request for more images.

What other considerations should you keep in mind for this question?

The Process & Communication Interview

Another type of interview you may encounter during an on-site visit is the development process / communication interview. During this interview a hiring manager, engineer, scrum master, or someone else from the team will ask you questions about collaboration and workflow.

Do not take these interviews lightly; they count just as heavily in determining whether or not you'll receive an offer.

Here are some things to keep in mind for process and communication interviews.

- Sit up straight and make eye contact
- Don't fidget
- Think before answering; it's okay to take a quick pause to iron out your answer before jumping into it.
- Never speak porrly about your previous or current employer

Sample Process & Communication Interview Questions

Here are several examples of questions you might receive in the process and communication interview. Try to come up with your own answers to these questions. Write them down and practice speaking them out loud. You want your answers to be well-spoken but not sound rehearsed.

Tell me about a project that failed. Why did it fail and how did you handle it?

Potential Answer

Last year I started a project with the goal of improving mentorship in the tech industry. Unfortunately the project required a lot of my time and I wasn't able to commit 20 hours a week to keep it running so I had to put it on hold.

I'm still passionate about mentorship so I decided to mentor two developers and write blog posts to share my knowledge. Even though my project is currently on hold I'm able to mentor others through my writing.

I hope to continue the project one day.

How would you handle a situation in which your coworker has a different opinion on how to develop a new feature?

Potential Answer

I would first try to understand why my coworker is so adamant about their particular solution. I would schedule a time with them one on one where we can discuss the conflict. We would do this in private and at a scheduled time to ensure we've created a safe space for a discussion.

What benefits does this solution provide that mine does not? Maybe their solution ultimately is better. If I listen and still disagree I would explain my point of view. If we still can't come to an agreement we can bring in a neutral party like a team lead to make the final decision.

Why are you looking to leave your current job?

Potential Answer

I enjoy my current company but I'm looking to move into a role which allows me to accomplish X. I believe I can do that here.

Potential Answer

I'm looking for a smaller company where I can make more of an impact. Working in a large company has been great and I've learned many skills but I'm ready to take on more responsibility within a smaller team.

What are you looking for in your next role?

Potential Answer

I'm looking for a company where I am surrounded by diverse coworkers who are passionate about what they do.

Potential Answer

I'm looking for a company where I can grow my skills through mentorship and continued education (i.e. conferences, online learning platforms).

Tips For On-Site Interviews

While all interview processes are unique here are some tips to help you perform your best!

Wear Comfortable & Smart Clothing

You want to balance looking professional yet approachable with feeling comfortable. If you don't normally wear high heels or a suit, don't pick the day of your on-site interview to try it out for the first time! Choose clothing that will put you at ease while still conveying your professionalism.

Take Bathroom Breaks

This may sound strange but taking a moment to decompress in the bathroom will help you reset your mind between interviews, especially if you had an interview that didn't go as planned. Take two minutes and breathe. Clear your mind. It will help you begin the next interview with a clean canvas.

Visit The Building The Day Before The Interview

If you're able to visit the campus or building the day before I highly recommend doing so. The morning of your on-site interview you'll likely be a little stressed out so focus on removing all of the obstacles or stressers you have control over.

If you can't visit the site the day before, give yourself an extra 30 to 45 minutes to be early. Often these buildings and campuses can be a bit tricky to locate.

Manager & Team-Matching Interviews

During your interview process you may meet with a potential manager or team. During these meetings it's important to have a few questions prepared regarding the role.

Here are some sample questions you may want to ask.

- What technology stack is the team using?
- Does the team iterate on sprints? If so, how long are sprints? One week? Two weeks? Four?
- What opportunities are there to expand my knowledge? Will I have the opportunity to attend conferences or take online courses?
- Is there an opportunity for mentorship?

After The Interview

After the interview take a minute and recognize how much you've accomplished. You may have spent weeks or months interviewing for this role, and now you can relax. Whatever happens happens!

There are two potential paths after completing an interview process:

- You'll receive a job offer
- You'll receive a rejection

Let's take a quick look at how you can handle both situations.

Negotiating An Offer

If you've received an offer, congratu-freaking-lations! You, and only you, earned this and you deserve it.

When I received my first job offer at IBM, I did not negotiate; I didn't even realize that negotiation was a thing people did. But when we don't negotiate we can leave a lot of money and benefits on the table. And let me tell you a secret: your co-workers negotiated. They didn't feel guilty negotiating, and you shouldn't either.

You don't have to solely negotiate salary either; many benefits are up-for-grabs during a negotiation! Paid time off, stock options, flexible working hours. These are all benefits you can negotiate into your contract.

If You’re Happy With The Offer

If you’re already happy with the offer, you don’t *have* to negotiate, but typically a company doesn’t make their best offer first. I always say there’s never any shame in trying!

That being said, some companies provide their best offer up-front, and that’s okay! If you're happy with the offer, you still win!

If You’re Not Happy With The Offer

If you’re not happy with the offer and the company isn’t willing to negotiate, you have to decide whether to accept the offer. This isn’t something I can provide guidance on as everyone’s situation is different.

Some people have the luxury of being able to choose between multiple offers while others won’t be able to pay their bills if they don’t accept.

Whether or not you choose to accept an offer is something you and your family should decide together.

Job Offer Negotiation Template

Hi `recruiter`!

Thank you again for sending me the offer package for a Software Engineering position on `hiring manager's` team. I'm humbled and thrilled to be considered for this position!

Before I accept your offer, I want to discuss the proposed salary. `Company A` has also given me an offer, however I'm extremely excited about the idea of joining the team at `company` and would like to move forward with you.

I have a lot experience creating/doing `tasks`. Given my experience and expertise I'm seeking a salary in the range of `X` to `Y` gross per year.

I will bring my expertise bridging design and engineering to the team at `company` and will work hard to exceed the team's expectations and deliver great work!

Please let me know if this salary range is possible. I would love to sign the offer today.

Thank you so much!

Processing A Rejection

I cannot even count the number of times I've received a rejection email or phone call. Many of them came before I even had the initial recruiter phone call. Many of them came during the technical interview process.

Everyone gets rejected. Everyone.

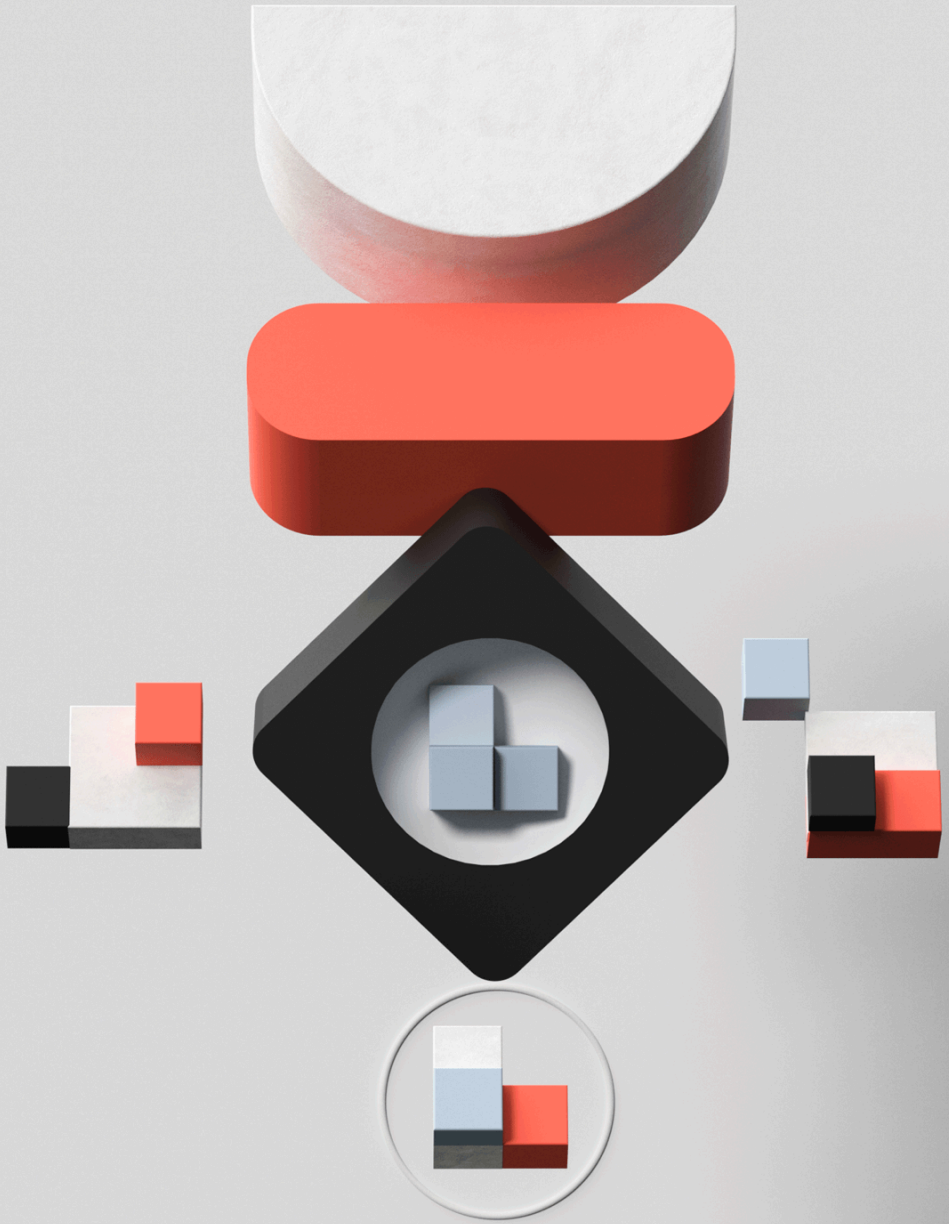
If you receive a rejection, it's okay, and it's a normal part of the process. It doesn't mean you're not good enough; it means you weren't a fit for this role or they had other candidates that more closely aligned with the teams' needs. You can always re-apply to a company in the future.

Take some time to process your rejection: they hurt. Let yourself feel those feelings and don't diminish them. Receiving a rejection is never a good feeling, however think about how proud of yourself you'll feel once you do receive an offer.

You'll get there. You can do this.

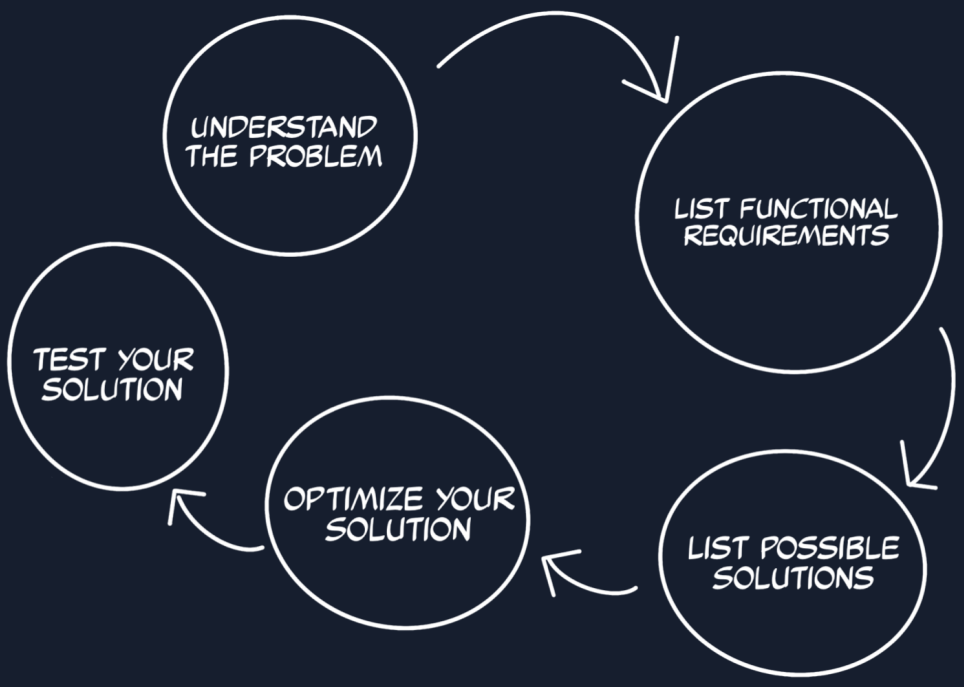
CHAPTER 03

Problem Solving



Where Do You Start?

Technical interview questions are generally meant to test your problem solving skills, so understanding how to solve problems can help you decode many problems.



Understand The Problem

Before jumping in to solving the problem, pause and ask yourself “do I understand what the problem is?” If the answer is no you should clarify.

Many questions are left intentionally incomplete as the interviewer wants to see your ability to deduce what information is incomplete or missing. A good way to ensure you understand the problem is to reiterate it to the interviewer.

“I have a graph with a cycle in it and I need to find the broken edge and remove it. Is this correct?”

If they say yes, you can move on to step two. If they say no ask for clarification.

List The Functional Requirements

Listing the functional requirements (or things your solution absolutely has to include) will help you understand the problem, ensure you don't get sidetracked on different aspects of the problem, and identify missing information.

For example if you're tasked with designing and coding an infinite scroll for an Instagram-style application, some of the functional requirements might be:

- Users must see their friends photos in reverse chronological order (the most recent first).
- We want to reduce the time to first paint (when the user first sees a "complete" UI) so lazy loading is a good option.
- We want to request a maximum of 30 photos at a time.

Thinking about these system requirements will allow you to focus on the tasks of most importance.

List Possible Solutions

Once you know what you need to design and code, think about a few different solutions. Are there any data structures you need to use? If so, what are the benefits and drawbacks of each? Speak your thoughts out loud and write down a few possible paths you can take.

Once you have two or three possibilities, choose the solution that you're most comfortable coding. If it's not the most performant or optimal solution that's okay. State that out loud.

“I know that using a nested for-loop to sort the array isn’t performant because it compares every element against every other element, but I’d like to start here and optimize once I get something down.”

There’s nothing wrong with coding a brute-force solution first and optimizing second.

Optimize Your Solution

If you have time, optimize your code. Can you remove the nested for-loop with an O(n^2) runtime and look for an O(n log n) solution?

If you don’t have time, explicitly state that you recognize this solution isn’t performant and you would have liked to refactor it to use merge sort or quick sort instead of bubble sort.

Test Your Solution

Even if you “know” your solution is 110% correct, test it. Think about the edge cases. What happens with your code when it’s passed an argument it didn't expect? Does it break? If so, refactor.

What To Do If You Get Stuck

There will likely come a time when you're solving a problem and you get stuck. You start to sweat, your stomach drops, and you freeze up. Getting stuck happens to the best engineers and it's nothing to beat yourself up about.

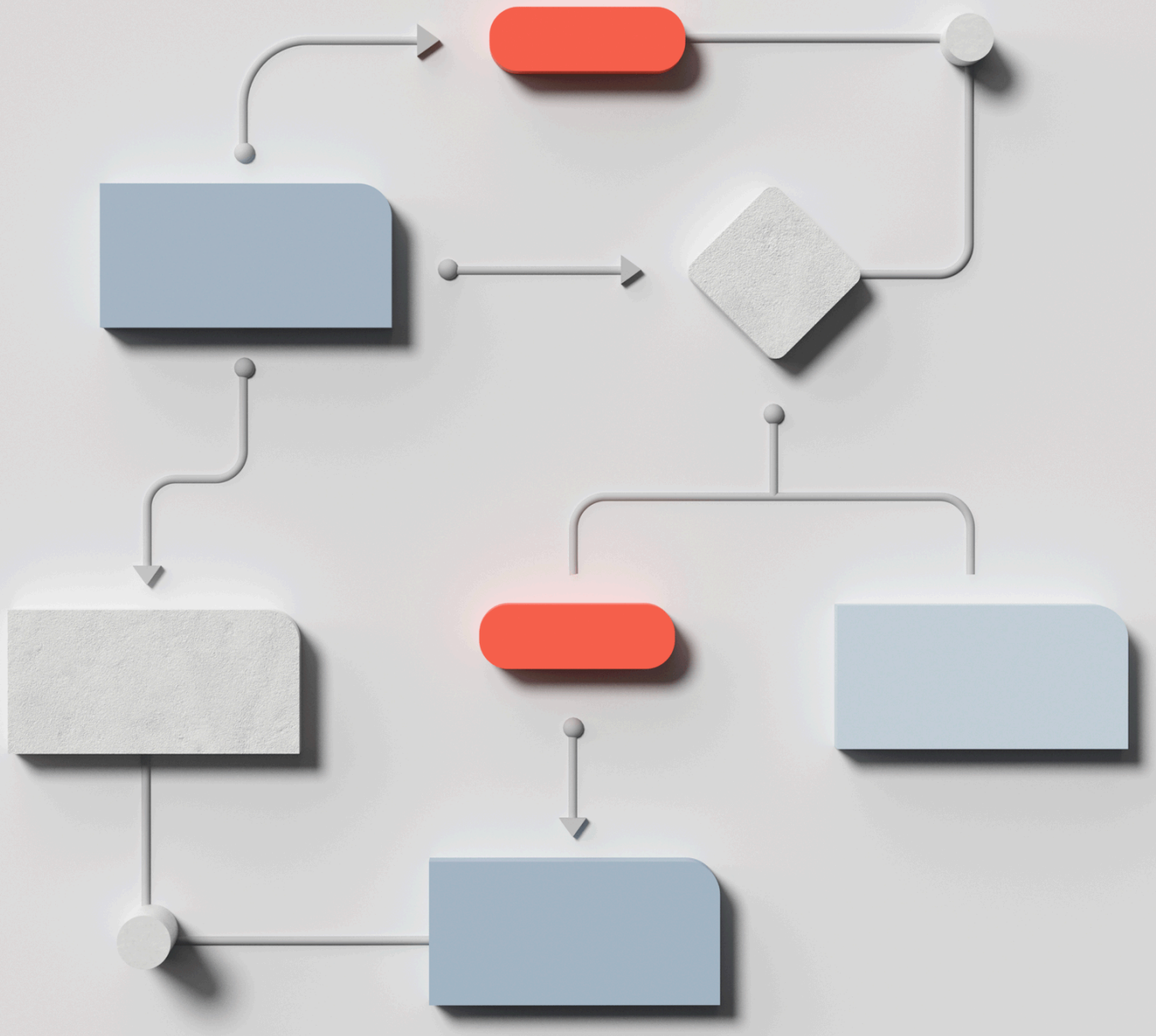
If you find yourself unable to move forward with a question, but honest with your interviewer. You can say something along the lines of "I know I need to do X but I'm not sure how to proceed."

Your interviewer most likely wants you to succeed. Therefore they'll be happy to give you a hint or two along the way if you get stuck or overlook an important aspect of the problem.

Try to stay calm, breathe, and take a drink of water. Everyone gets stuck, and you will be okay.

CHAPTER 04

Data Structures



The data structures and algorithms interviews give many candidates anxiety; they're not something that a front-end developer would necessarily encounter in their day-to-day programming activities.

We live in a time where you don't need a computer science degree to get a development job. People take many paths to join the technical industry: bootcamps, related college degrees, unrelated college degrees, self-teaching. But the data structures and algorithms interview tests knowledge that is primarily taught during a computer science degree. Thus many candidates are left cramming the days or weeks leading up to an interview and this is extraordinarily stressful.

What Is A Data Structure?

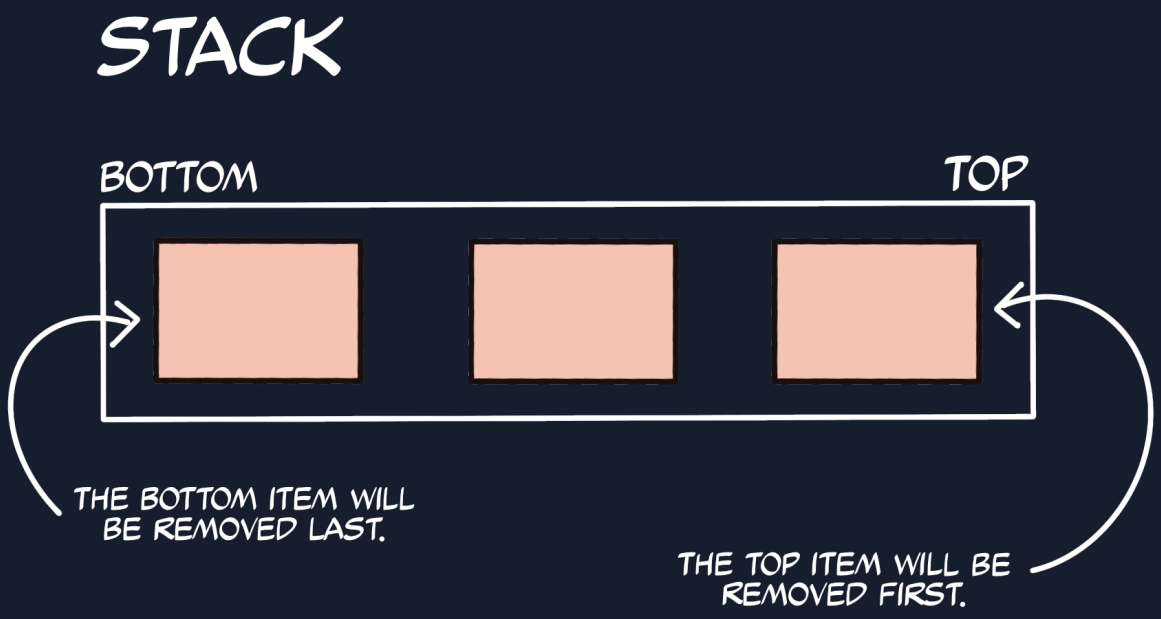
A data structure is an organization of data such that it can be accessed and modified in a particular manner. This is quite a vague definition but hopefully the definition becomes clearer as we progress through this section.

Many of the data structures you'll need for your technical interviews are not natively built into JavaScript. As a result they can be overwhelming to learn, especially when many of the resources for learning these data structures are geared towards back-end languages.

I will link a few additional resources for learning and practicing data structures in JavaScript in the resources section of this book. But let's take a quick look at some of the most common.

Stacks

A stack is a “last-in-first-out” data structure which means that the newest element (or the element which was added last) in the stack will be the first one removed. We can think of a stack like a stack of books. To get to the third book in the stack we have to remove the fifth book and then the fourth book (I know some of you are probably strong and just lift them all up at once but humor me).



Benefits Of Stacks

Stacks allow for constant-time adding and removing of the top item. Constant-time, or O(1), is a very performant runtime (we'll discuss Big-O notation and runtime in the next section).

These actions are constant-time because we don’t need to shift any items around to pop off the top item in the stack.

Downsides Of Stacks

Stacks, unfortunately don’t offer constant-time access to the nth item in the stack, unlike an array. So if we want to access the third book, we would have to pop off each item in our stack until we reach the 3rd book, and if we want to access the first item in the stack (the bottom element) we have to iterate through each book on top of it and pop it off; this has a worst-case runtime of O(n) where n is the number of books in the stack.

By contrast we can access specific indices in arrays with bracket notation. If we want the third item in an array we can access it in constant time with array[2].

Stack Methods

There are three primary methods (`push`, `pop`, `peek`) in a stack and a few additional methods (isEmpty, get length) you may add to help with different tasks.

- `pop()`: Removes the top item from the stack
- `push(item)`: Adds an item to the top of the stack
- `peek()`: Returns the item at the top of the stack (but does not remove it)
- `isEmpty()`: Returns true if the stack is empty
- `get length()`: Returns the number of items in the stack

Coding A Stack In JavaScript

I'll be coding this stack using JavaScript's class notation, but you can also create a stack with functions.

The first thing we'll do is create a class Stack and give it a constructor with one property: stack. We will build this stack using an array. The top of the stack will be the end of the array and the bottom of the stack will be the beginning of the array.

I'm structuring my stack in this manner so I can use the native `array.push()` and `array.pop()` methods. If we wanted to add and remove items from the beginnning of the array instead of the end we would have to use `array.unshift()` and `array.shift()`.

```
class Stack {
  constructor() {
    this.stack = []
  }
}
```

First let's create our length getter function. This will keep track of the number of items in our stack.

I'm using a getter function so I can access the stack's length with `stack.length` as opposed to creating a function `getLength()` which has to be called each time I want to check the length.

```
get length() {
  return this.stack.length;
}
```

Now let's create the push method which adds an element to the top of the stack. We can use the `array.push()` method (since the top of our stack is the end of the array). This method takes one argument: the item to add.

```
push(item) {
  return this.stack.push(item);
}
```

Next let's create the `pop` method which removes the top element from the stack and returns it. Since we've set the top of the stack to the end of the array we can use the `array.pop()` method.

```
pop() {
  return this.stack.pop();
}
```

To see which element lives at the top of the stack we can create a peek method. We just have to check which item lives in the last index of our stack array.

```
peek() {
  return this.stack[this.length - 1];
}
```

We may also want to have a helper function to check whether our stack is empty. You can of course omit this method and just check `this.length === 0`, however I personally like having this helper function.

```
isEmpty() {
  return this.length === 0;
}
```

Here is our finalized stack code. You can view the live code on my Code Sandbox.

```
class Stack {
  constructor() {
    this.stack = []
  }
  get length() {
    return this.stack.length
  }
  push(item) {
    return this.stack.push(item)
  }
  pop() {
    return this.stack.pop()
  }
  peek() {
    return this.stack[this.length - 1]
  }
  isEmpty() {
    return this.length === 0
  }
}
```

When Would You Use A Stack During An Interview?

Let’s try to think of a use case where a stack might come in handy during a technical interview. Take a few moments and see if you can come up with a scenario.

Keep in mind that stacks are a last-in-first-out data structure, meaning that the most recently added item will be the first one to be removed when the pop() function is called.

Were you able to come up with a technical interview question where a stack would be a good data structure to use? If so, great! If you struggled to think up a scenario, don't worry! The more you learn about and practice using data structures, the more quickly you'll be able to recognize questions that require certain data structures.

Here's an example question where a stack would be an optimal data structure:

Imagine you're building a navigation component for a product website we'll call "Rainforest". Users can navigate from the home page to different product pages within the site. For example, a user flow might be Home > Kitchen > Small Appliances > Toasters.

Build a navigation component that displays the list of previously visited pages as well as a back button which lets the user go back to the previous page.

A stack would be an optimal solution for this question because the top-most item in a stack is the most recently added item (in this case the last page that was visited). So when we want to navigate backwards, we can pop off the last item in the stack and render the state of the previous page.

Your solution should display a navigation with four links, a history list that displays the previously visited pages, a current page paragraph which displays the name of the currently displayed page, and a go back button which, when clicked, navigates to the previously visited page.

Take some time to develop your solution. Talk through your solution out loud to practice the traditional whiteboarding style solution. Even if you're confident in your communication skills, it's important to practice speaking your thoughts.

Have you developed a solution? If so, have you tested several user flows? Does anything break? If so, fix it!

Here are some additional error boundaries to think about:

- What happens if you don't have any previously visited pages?
- What happens if you're already on a page and you try navigating to it?

Now that you've accounted for some special use cases, are there any enhancements you could make to optimize performance or the user experience?

Follow-Up Question

Often during coding challenges, you'll be posed with an initial question and if you finish a piece of it the interviewer will ask some follow-up questions. Here are a couple of follow-ups you might encounter for this question. Often you're not expected to complete all pieces of the question or get them completely correct: these challenges are more to discover how you problem solve and communicate.

Expand your solution to be able to navigate forwards as well as backwards. Here are a few example user flows which will help you develop your solution:

User flow 1: Home Home > Kitchen Home > Kitchen > Bathroom *Back* Home > Kitchen Home > Kitchen > Living room

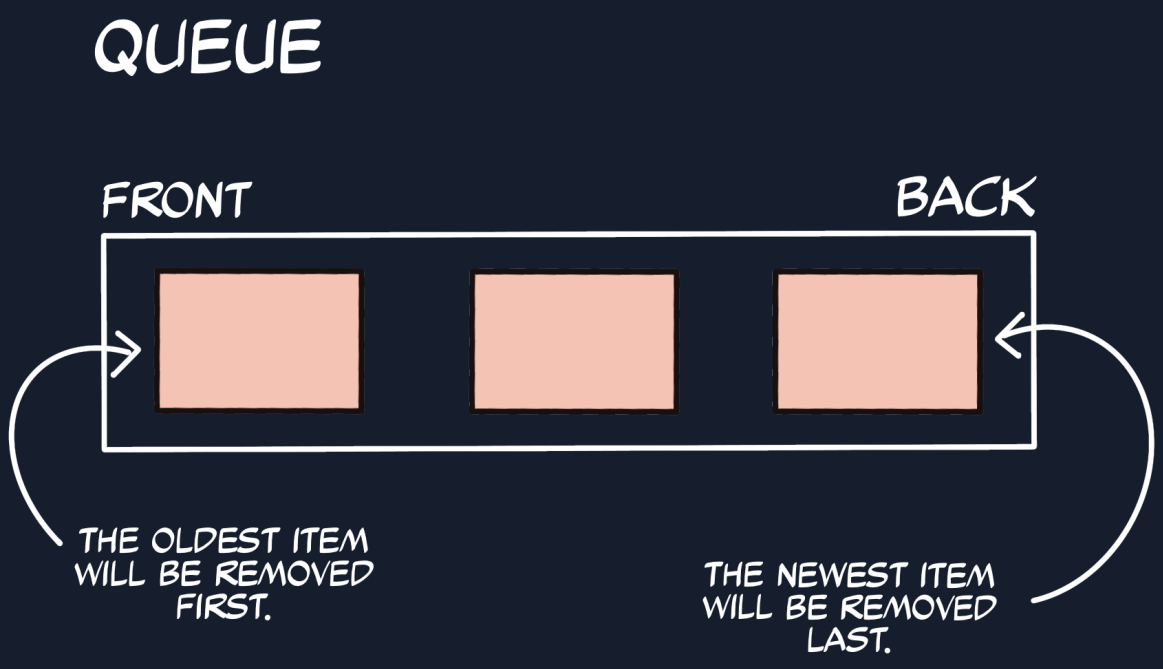
User flow 2: Home Home > Kitchen Home > Kitchen > Bathroom *Back* Home > Kitchen *Forward* Home > Kitchen > Bathroom Home > Kitchen > Bathroom > Home

User flow 3: Home Home > Kitchen Home > Kitchen > Bathroom *Back* Home > Kitchen *Back* Home *Forward* Home > Kitchen *Forward* Home > Kitchen > Bathroom

Queues

Queues are very similar to stacks however they use the “first-in-first- out” paradigm. This means that the oldest element (the element that was added first) is the next item to be removed.

We can picture a queue like a queue of people waiting to buy movie tickets. The person who has been waiting in line the longest is the next person to be serviced.



Use Cases For Queues

Queues are similar to linked lists (which we'll cover next) and are typically used in breadth-first searches for trees (which we'll also cover in a subsequent section). You may also see queues being used to implement cache.

Downsides Of Queues

Queues are much more difficult to update when adding and removing items than a stack because we're adding items to one side of the structure and removing them from the other side.

Queue Methods

Queues use three primary methods (`enqueue`, `dequeue`, `peek`) and several helper methods (`isEmpty`, `get length`).

- `enqueue()`: Add an item to the back of the queue
- `dequeue()`: Remove an item from the front of the queue
- `peek()`: Return the item at the front of the queue (but do not remove it)
- `isEmpty()`: Check whether the queue is empty
- `get length()`: Return the length of the queue

Coding A Queue In JavaScript

I'll be coding this queue using JavaScript's class notation, but you can also create a queue with functions.

The first thing we'll do is create a class Queue and give it a constructor with one property: queue. We will build this queue using an array. The front of the queue will be the front of the array and the back of the queue, where we add new elements, will be the end of the array.

```
export default class Queue {
  constructor() {
    this.queue = []
  }
}
```

First let's create a length property which will return the length of the queue. We'll use a getter function so we can access the length with `queue.length`.

```
get length() {
  return this.queue.length;
}
```

Now let's write the enqueue method which will take an item and add it to our queue. Remember we're adding items to the end of the array so we can use the native `array.push()` method.

```
enqueue(item) {
  this.queue.push(item);
}
```

The dequeue method will remove the element at the front of the queue. Since the front of our queue is the beginning of the array we can use the `array.shift()` method.

```
dequeue() {
  return this.queue.shift();
}
```

To check which item is at the front of the array we can create a peek method. The item at the front of the queue is the first element in the queue array so we can access it with array[0].

```
peek() {
  return this.queue[0];
}
```

Finally let's add a helper method, **`isEmpty`**, which returns a boolean value indicating whether or not the queue has items.

```
isEmpty() {
  return this.length === 0;
}
```

Here is our finalized queue code. You can view the live code on my Code Sandbox.

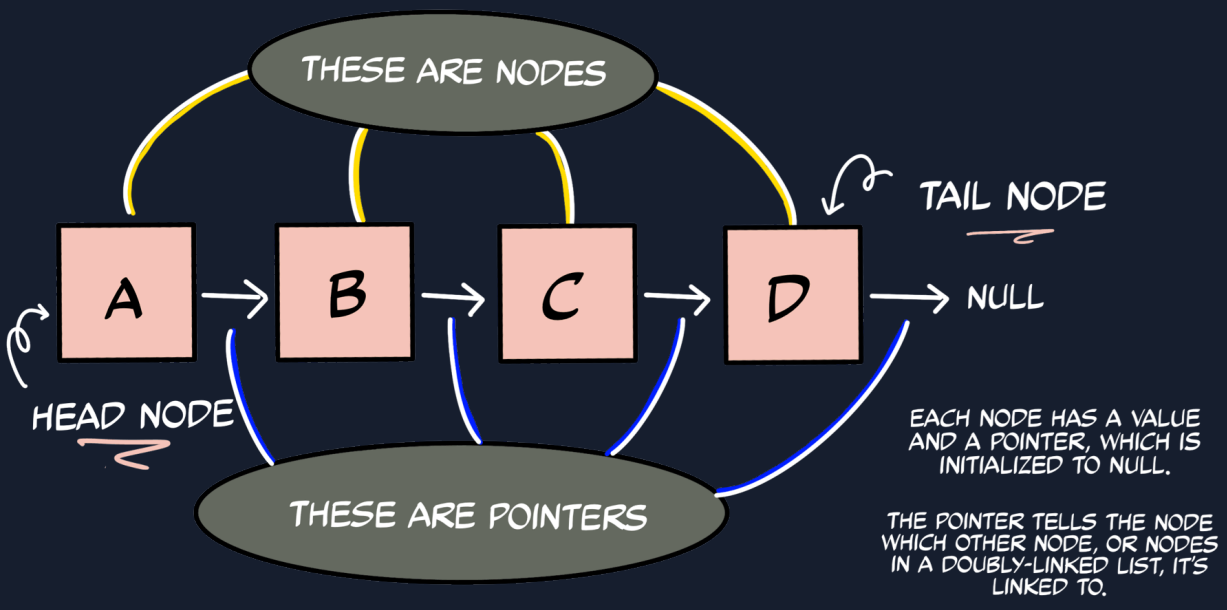
```
export default class Queue {
  constructor() {
    this.queue = []
  }
  get length() {
    return this.queue.length
  }
  enqueue(item) {
    this.queue.push(item)
  }
  dequeue() {
    return this.queue.shift()
  }
  peek() {
    return this.queue[0]
  }
  isEmpty() {
    return this.length === 0
  }
}
```

Linked Lists

Linked lists are a series of linked nodes where each node points to the next node in the list. Each node has a value and a pointer to the next node. There are also doubly-linked lists in which each node also points to the previous node in the list.

Linked lists use the “last-in-first-out” method (similar to a stack) where nodes are added to and deleted from the same end.

To search for a node in a linked list we have to start at the head node (the first node in the list) and iterate through each node until we find it or reach the end of the list. In a worst-case scenario this means that searching for an item has a runtime of O(n) where n is the number of items in the list.



Linked List Methods

Linked lists use two primary methods (`push` , `pop`) and several helper methods (`get index` , `delete` , `isEmpty`).

- `push(Node)` : Add an element to the linked list
- `pop()` : Remove an element from the linked list
- `get(index)` : Return an element from a given index (but don't remove it)
- `delete(index)` : Delete an item from a given index
- `isEmpty()` : Return a boolean indicating whether the list is empty

Singly-Linked Lists vs. Doubly-Linked Lists

Linked lists can be singly, or doubly-linked. In a singly-linked list, each node has one pointer which points to the next element in the list. In a doubly-linked list, each node has two pointers: one which points to the next element in the list and one which points to the previous element in the list.

Doubly-linked lists are great for removing nodes because they provide access to the previous and the next nodes. To remove a node from a singly-linked list, we have to iterate through the list, keeping track of the previous node (you can see this concept illustrated in just a few pages) so it’s a bit more complicated.

For our purposes today we’ll code a singly-linked list, but be aware that doubly-linked lists do exist!

Coding A Linked List In JavaScript

Let's first build our `Node` class. Nodes have a value and a pointer to the next node (for singly-linked lists, which is what we'll be building). When we create a new node we will pass the value to the constructor. We will also initialize the pointer to null (as we're adding this node to the end of the list).

```
class Node {
  constructor(value) {
    this.value = value
    this.next = null
  }
}
```

Now we can create our Linked List class. The constructor will keep track of three things:

- `head`: The head pointer that keeps track of the first node in the linked list
- `tail`: The tail pointer that keeps track of the last node in the linked list
- `length`: The number of nodes in the list

The head and tail pointers will be null until we add our first node.

```
class LinkedList {
  constructor() {
    this.head = null
    this.tail = null
    this.length = 0
  }
}
```

First let's create our isEmpty method which returns true if there are no nodes in the list.

```
isEmpty() {
  return this.length === 0;
}
```

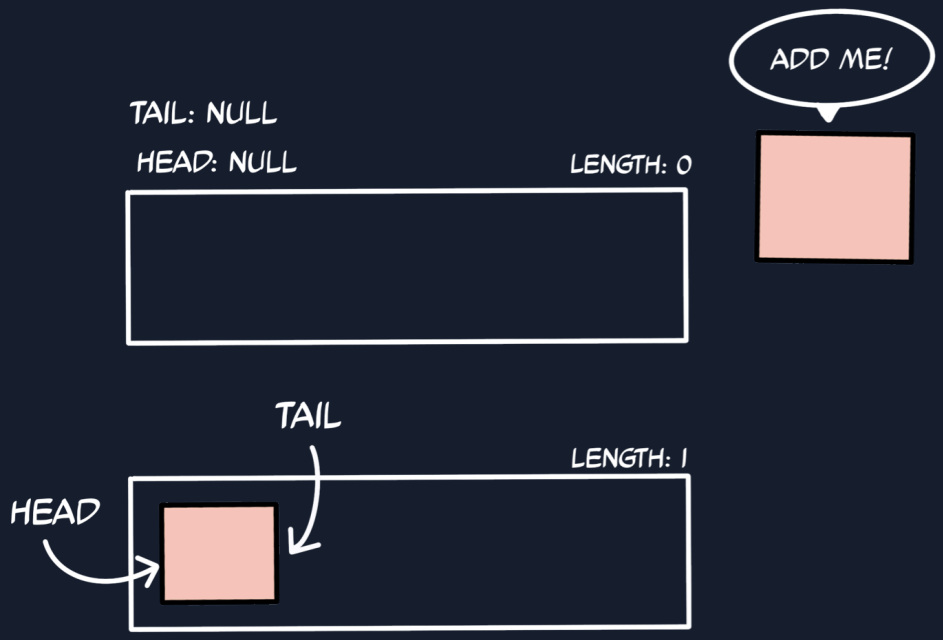
Adding Nodes To Linked Lists

Let’s create our push method which takes in a value and adds a node to the end of the linked list. Remember we have to update the tail pointer whenever we add a new node!

Let’s walk through two scenarios which change the way we add nodes.

The List Is Empty:

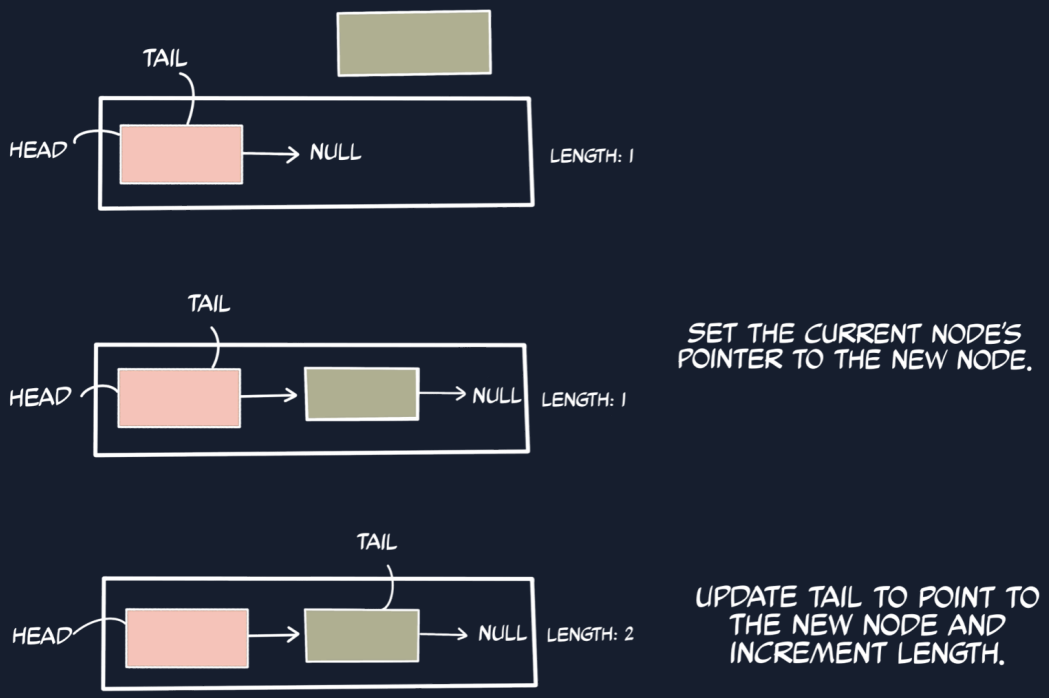
If the list is empty, we can set head and tail pointers to the newly added node, then increment the length.




```
push(value) {
  const newNode = new Node(value);
  if (this.isEmpty()) {
    this.head = newNode;
    this.tail = newNode;
  }
}
```

The List Has At Least One Node:

If the list is not empty we have to first set the current tail node's pointer to the new node. Then we can set `this.tail` to the new node and increment the list length.




```
push(value) {
  const newNode = new Node(value);
  if (this.isEmpty()) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    this.tail.next = newNode;
    this.tail = newNode;
  }
  this.length++;
}
```

Removing Nodes From Linked Lists

Now let's write our pop method which removes the node at the tail. We have a couple of scenarios to consider for removing a node.

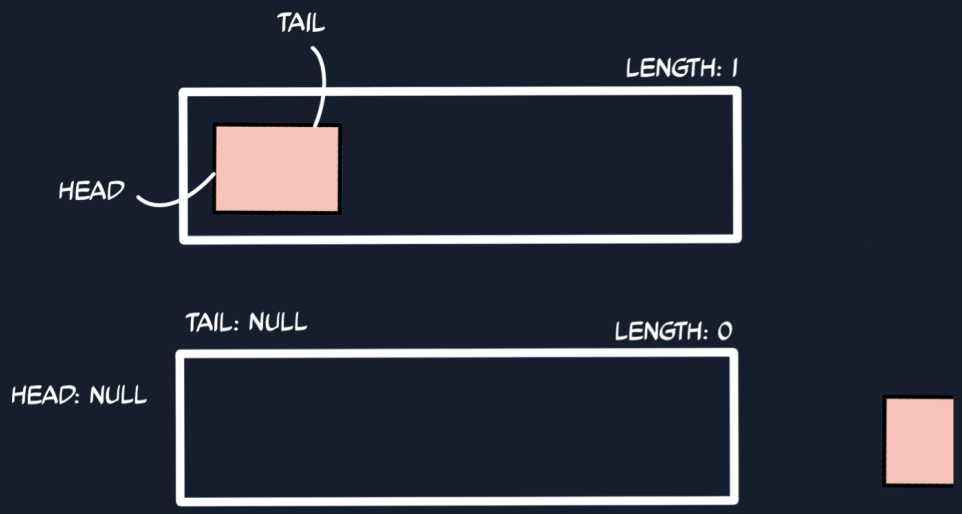
The List Is Empty:

If the list is empty let's return null.

```
pop() {
  if (this.isEmpty()) {
    return null
  }
}
```

There Is Only One Node In The List:

If there's only one node in the list we have to reset the head and tail pointers to null and decrement the length. How do we know if our list has one element? We can check the length property or check whether the head and tail pointers refer to the same element.




```
pop() {
  /* List is empty */
  if (this.isEmpty()) {
    return null;
  } else if (this.length === 1) {
    /* There is one node in the list */
    const nodeToRemove = this.head;
    this.head = null;
    this.tail = null;
    this.length--;
    return nodeToRemove;
  }
}
```

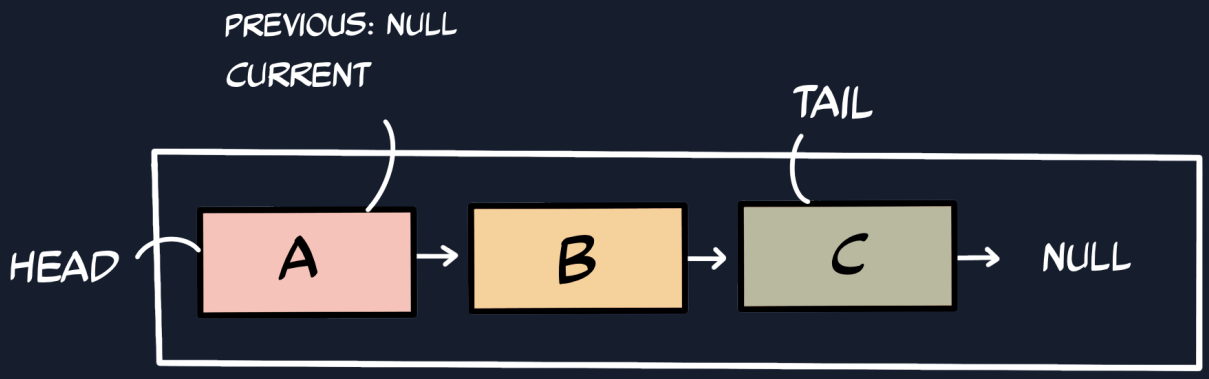
There Is More Than One Node In The List

If there are multiple nodes in the list we have to follow a few steps. Remember that we cannot simply access a specific node in a linked list and in a singly-linked list we only have access to the next item in the list. Thus to pop off the tail item in the list we first need to find the node that points to the tail node.

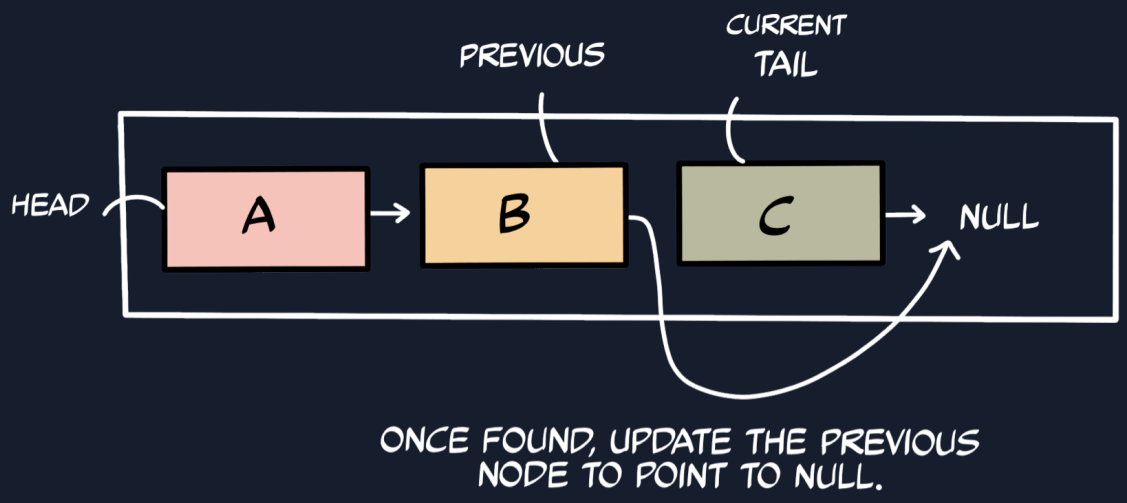
Once we find the second to last node we have to assign it to a variable so we can return it. We'll call this variable `nodeToRemove` and set it to the current tail node. Then we'll update tail pointer to point to the second to last node, and finally update the new tail node to point to null.

Here is a summary of these steps. You can see this process illustrated on the next few pages.

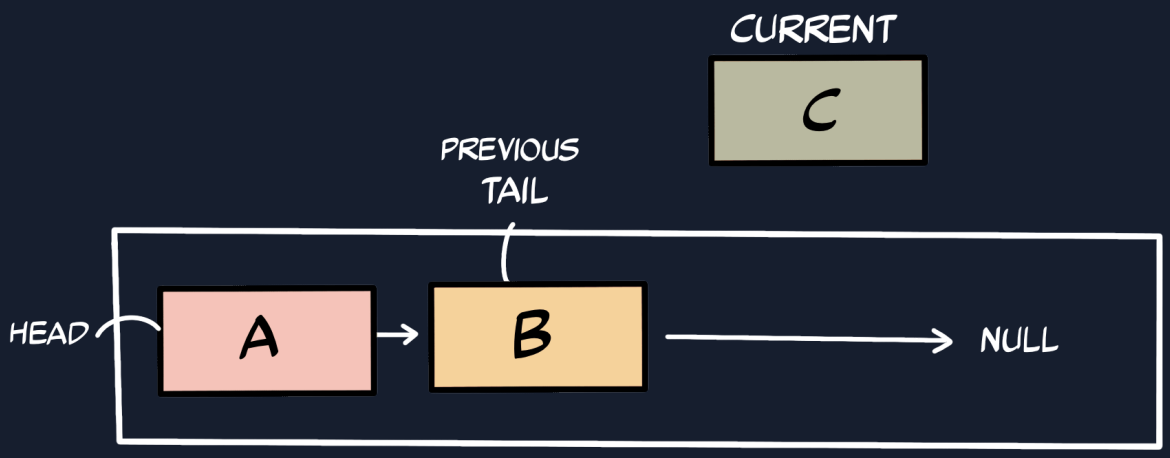
```
/*
  While there are nodes in the list
    If the next node in the list is the tail
      Update the tail pointer to point to the current node
      Set the new tail node to point to null
      Decrement the length of the list
      Return the previous tail element as removed
*/
```



START CURRENT NODE AT THE HEAD AND TRAVERSE UNTIL WE FIND THE SECOND TO LAST NODE.



ONCE FOUND, UPDATE THE PREVIOUS NODE TO POINT TO NULL.



UPDATE THE TAIL TO POINT TO THE PREVIOUS NODE, SPLICING OUT THE OLD TAIL.

```
pop() {
  /* List is empty */
  if (this.isEmpty()) {
    return null;
  } else if (this.length === 1) {
    /* There is one node in the list */
    const nodeToRemove = this.head;
    this.head = null;
    this.tail = null;
    this.length--;
    return nodeToRemove;
  } else {
    /* There are multiple nodes in the list */
    // Start our pointer at the head
    let currentNode = this.head;
    // We're removing the last node in the list
    const nodeToRemove = this.tail;
    // This will be our new tail
    let secondToLastNode;
    while (currentNode) {
      if (currentNode.next === this.tail) {
        secondToLastNode = currentNode;
        break;
      }
      currentNode = currentNode.next;
    }
    secondToLastNode.next = null;
    this.tail = secondToLastNode;
    this.length--;
    return nodeToRemove;
  }
}
```

Getting Nodes From Linked Lists

Let's now create a `get` method which takes an index and returns the node at that index. We need to check for three cases:

- The index requested is outside the bounds of the list
- The list is empty
- We're requesting the first or last element

The Index Isn't Valid:

If the index requested is outside the bounds of the list, or if the list is empty, let's return null.

```
get(index) {
  if (index < 0 || index > this.length || this.isEmpty()) {
    return null;
  }
}
```

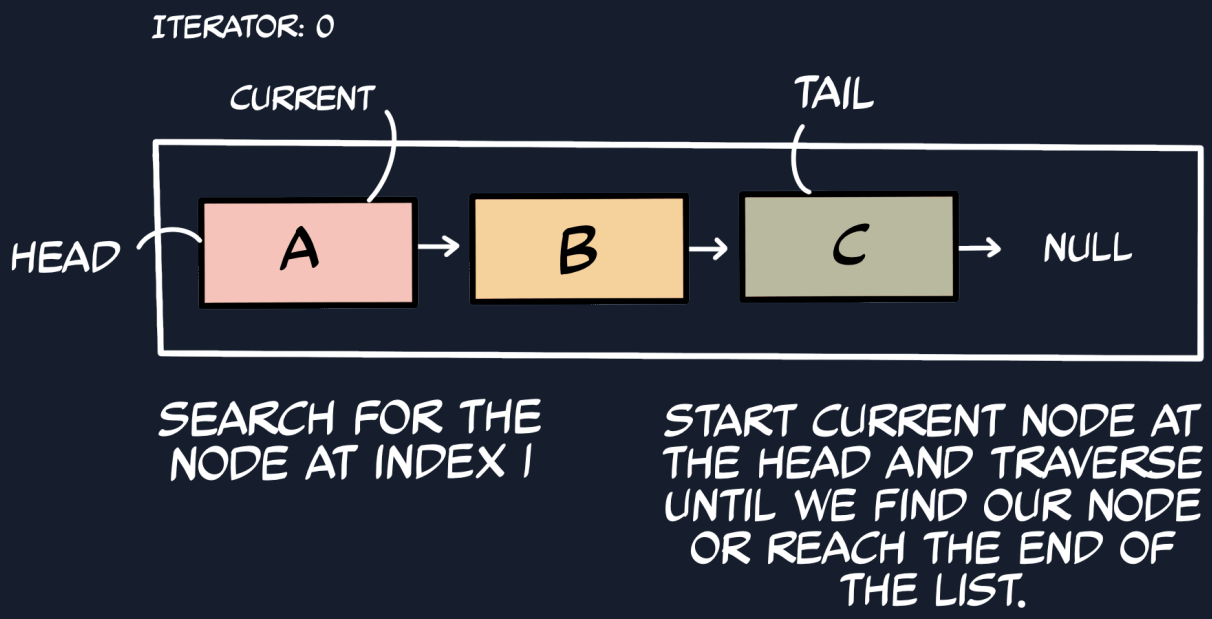
Requesting The First Or Last Index

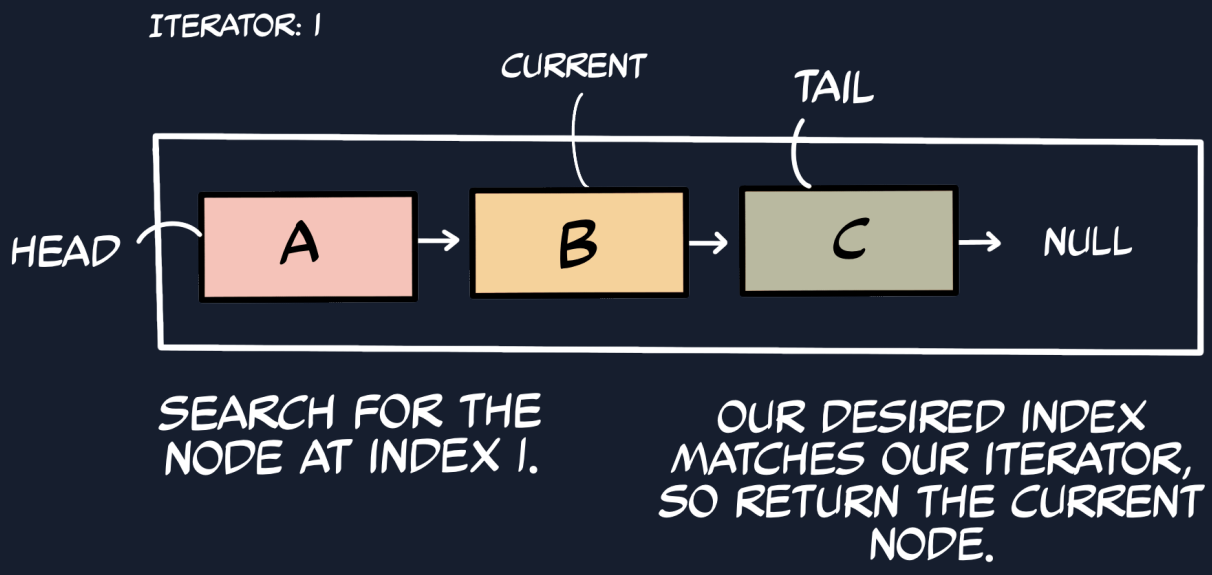
If we're requesting the first index, return the node at the head pointer. If we're requesting the last index (this.length - 1), return the tail node.

```
get(index) {
  if (index < 0 || index > this.length || this.isEmpty()) {
    return null;
  }
  /* We want the first node */
  if (index === 0) {
    return this.head;
  }
  /* We want the last node */
  if (index === this.length - 1) {
    return this.tail;
  }
}
```

Requesting A Valid Index

If the index is a valid index and isn't the first or last value in the array, we need to iterate through the list until we find the index. We can keep track of the current index with an iterator variable.





```
get(index) {
  if (index < 0 || index > this.length || this.isEmpty()) {
    return null;
  }
  /* We want the first node */
  if (index === 0) {
    return this.head;
  }
  /* We want the last node */
  if (index === this.length - 1) {
    return this.tail;
  }
  /* We want a node somewhere in the list */
  let currentNode = this.head;
  let iterator = 0;
  while (iterator < index) {
    iterator++;
    currentNode = currentNode.next;
  }
  return currentNode;
}
```

Graphs

A graph is a data structure composed of a collection of nodes and edges. Graphs are a non-linear data structure (as opposed to a linked list, stack, or queue). You may also hear nodes referred to as vertices.

Graphs are used to solve many real-world problems and can be used to represent networks.

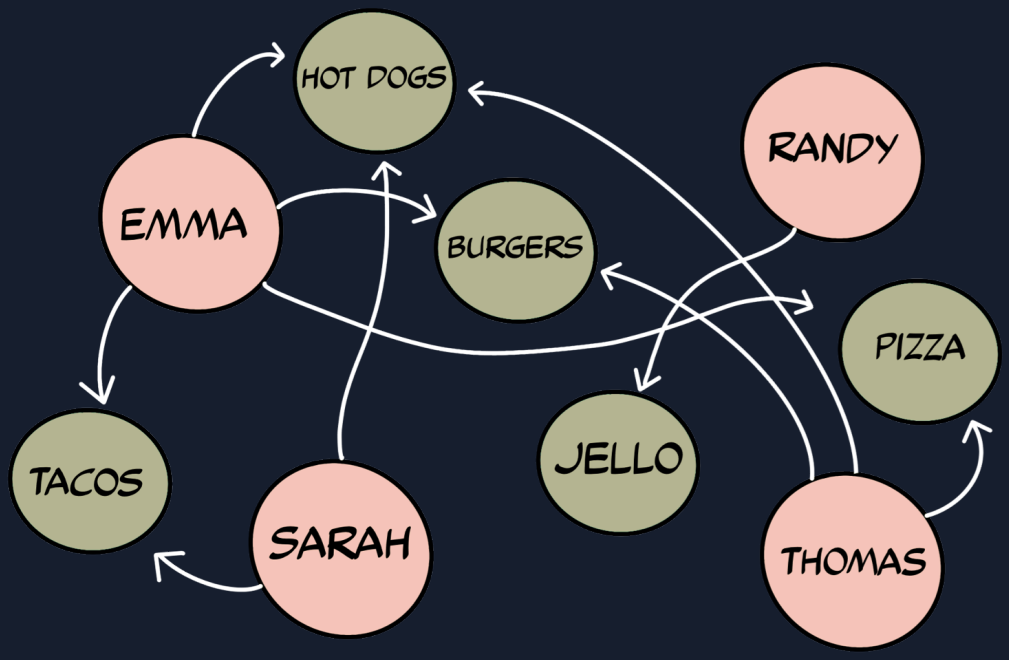
Let's say you're a school bus driver, for example, and you want to illustrate the different ways you can complete your bus route each school morning to maximize efficiency, you can use a graph to map it out (this is a version of the Traveling Salesman problem, also known as an NP-Hard problem, and would take a lot of time to solve).

There are two types of graphs:

- Directed graphs
- Undirected graphs

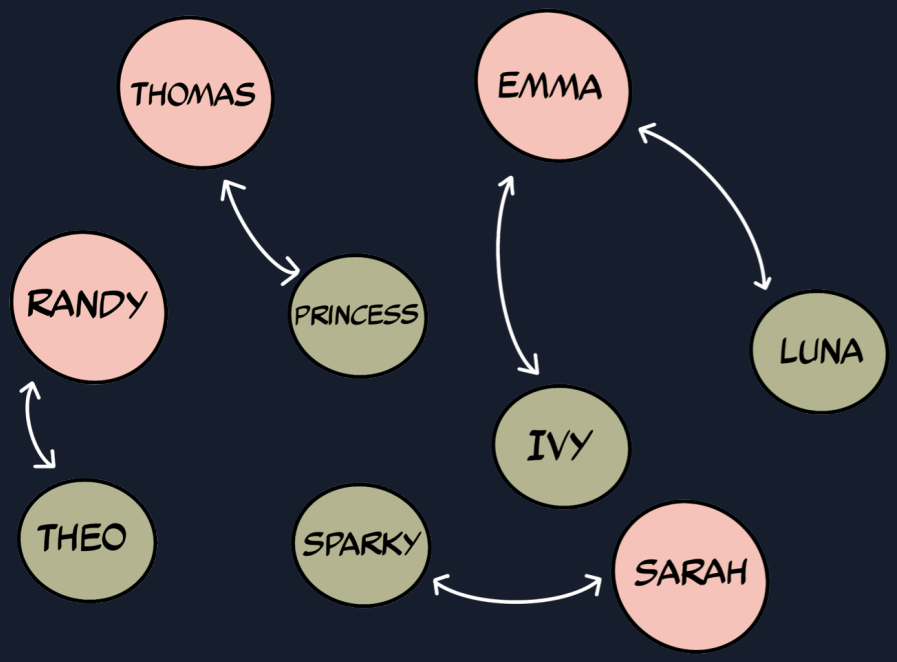
Directed Graphs

A directed graph contains edges which function similarly to a one-way street; they have a direction. For example you might have a graph where people can have favorite foods but foods don't have favorite people.



Undirected Graphs

An undirected graph, in contrast, contains edges which flow bidirectionally, like a two-way street. For example you might have a graph of pets where people have pets and pets have owners. The relationship goes both ways.



Coding A Graph In JavaScript

Let's first create our Node class. The constructor will take a value, which will be its identifier. The constructor will also contain a list of edges for that node. We can use an array to keep track of this.

```
class Node {
  constructor(value) {
    this.value = value
    this.edges = []
  }
}
```

Now let's create our Graph class. The Graph constructor will take one argument, a boolean value, which indicates whether this graph is directed or undirected. We'll set this value to be undirected by default.

The constructor will also contain a nodes array which keeps track of all nodes in our graph.

```
export default class Graph {
  constructor(directed = false) {
    this.undirected = undirected
    this.nodes = []
  }
}
```

Adding A Node

To add a node, we can create a new node with the provided value and push that to our Graphs nodes array.

```
addNode(value) {
  this.nodes.push(new Node(value));
}
```

Removing A Node

Removing a node is a bit tricker because we not only have to remove it from the Graph nodes array, we also have to iterate through each node in the nodes array and remove any edges containing that node.

This nested loop isn't very performant; can you think of a more performant way to refactor this?

```
removeNode(value) {
  this.nodes = this.nodes.filter(node => node.value !== value);
  this.nodes.forEach(node => {
    node.edges = node.edges.filter(edge => edge.value !== value);
  });
}
```

Getting A Node

To get a node, we just have to find it in the Graph nodes array and return it! We can use the array.find() method to accomplish this.

```
getNode(value) {
  return this.nodes.find(node => node.value === value);
}
```

Adding An Edge

To add an edge between two nodes we have to first get a handle on the nodes, as we're accepting the node value as the argument. This is where our getNode function comes in handy.

Next let's add an edge from node one to node two. If the graph is undirected, we also need to add an edge from node two to node one. Then we'll just return a nice little template string to indicate this process was successful.

```
addEdge(value1, value2) {
  const node1 = this.getNode(value1);
  const node2 = this.getNode(value2);
  node1.edges.push(node2);
  if (this.undirected) {
    node2.edges.push(node1);
  }
  return `An edge between ${node1.value} and ${node2.value} was added`;
}
```

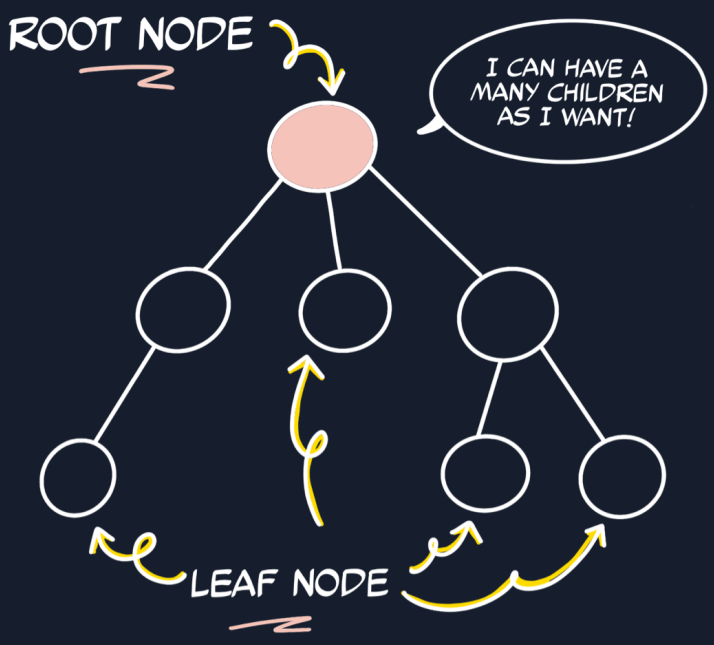
Trees

A tree is a data structure where a node can have zero or more children. Each node contains a value, and similar to graphs each node can have a connection between other nodes called an edge. A tree is a type of graph but not all graphs are trees.

The top-most node is called the root. The DOM, or document object model, is a tree data structure.

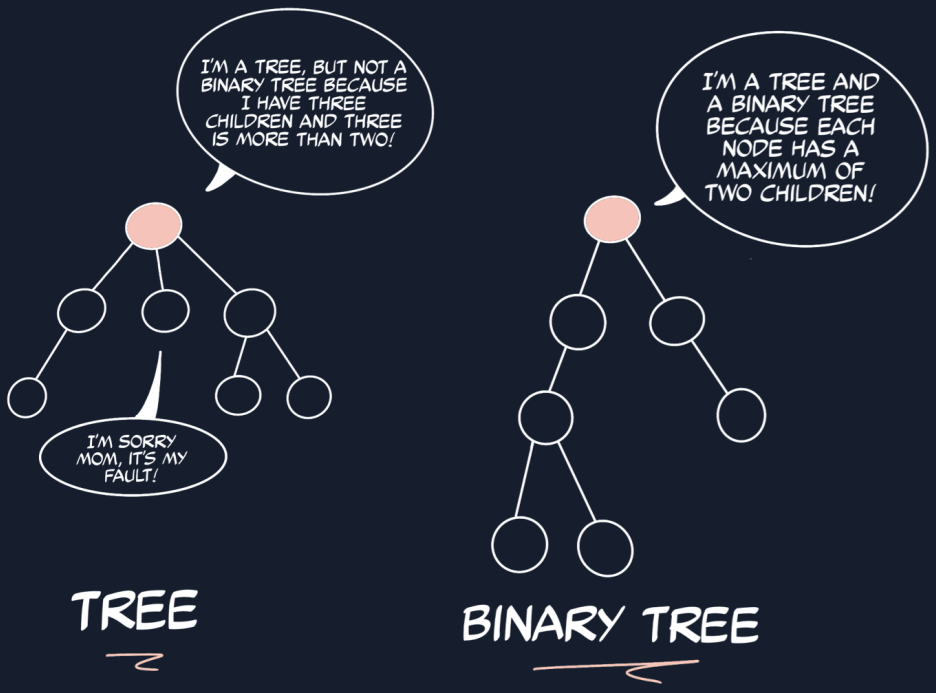
A node without children is called a leaf node.

The height of the tree is the distance between the farthest leaf node and the root node.



Binary Trees

Binary trees are a special type of tree in which each node can only have a maximum of two children: a left child and a right child.



Full Binary Trees

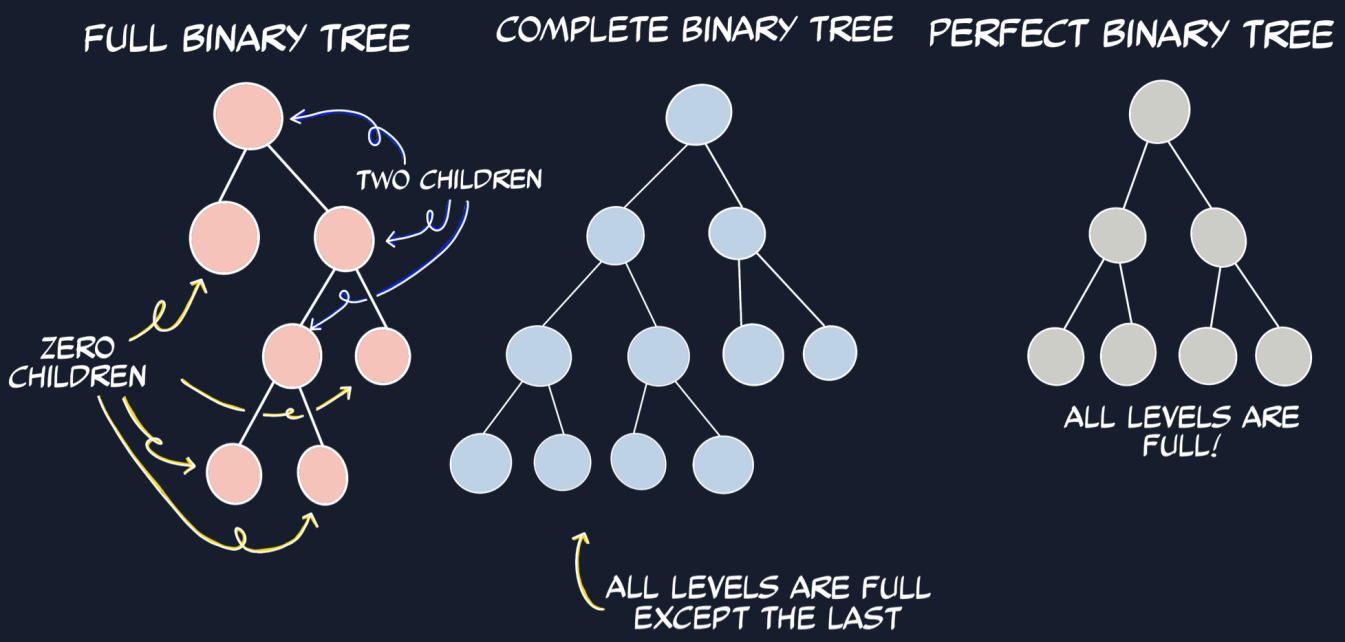
A full binary tree is a binary tree in which every node has exactly zero or two children (but never one).

Complete Binary Trees

A complete binary tree is a binary tree in which all levels except the last one are full with nodes.

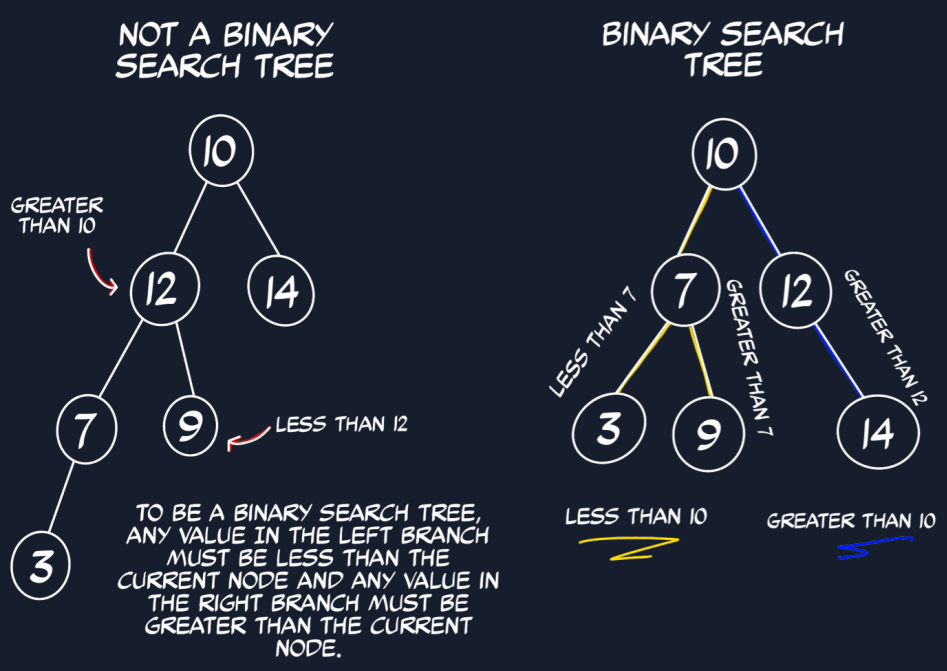
Perfect Binary Trees

A perfect binary tree is a binary tree in which all levels, including the last level, are full of nodes.



Binary Search Trees

A binary search tree is a special type of binary tree where the values of every node to the left are less than its value and the values of every node to the right are greater than its value.



Coding A Binary Search Tree In JavaScript

First let's build our node class. Each node can have at most two children, so we'll call these `leftChild` and `rightChild`. These will contain Node values.

```
export class Node {
  constructor(value) {
    this.value = value
    this.leftChild = null
    this.rightChild = null
  }
}
```

We'll also create a BinaryTree class. The constructor won't take in any arguments, but it will have a root node.

```
export default class BinaryTree {
  constructor() {
    this.root = null
  }
}
```

Adding Nodes

Let's create a function inside of our BinaryTree class which will add a node. There are two cases we need to account for:

- The tree is empty
- There are nodes in the tree

The Tree Is Empty:

First, if the tree is empty, set `this.root` equal to a new Node containing this value.

```
addChild(value) {
  if (this.root === null) {
    this.root = new Node(value);
    return;
  }
}
```

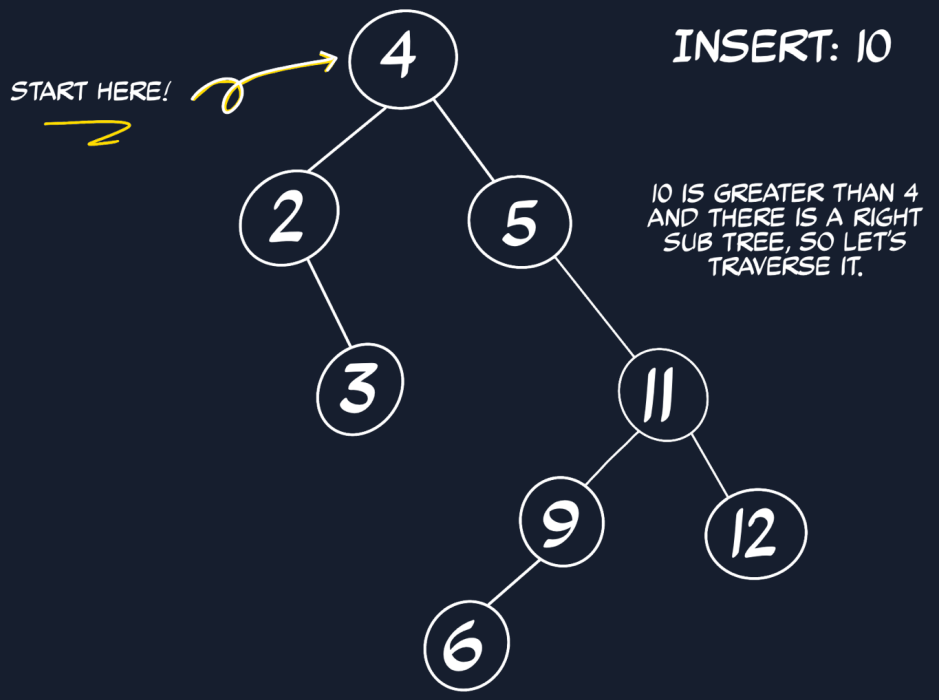
The Tree Has Nodes:

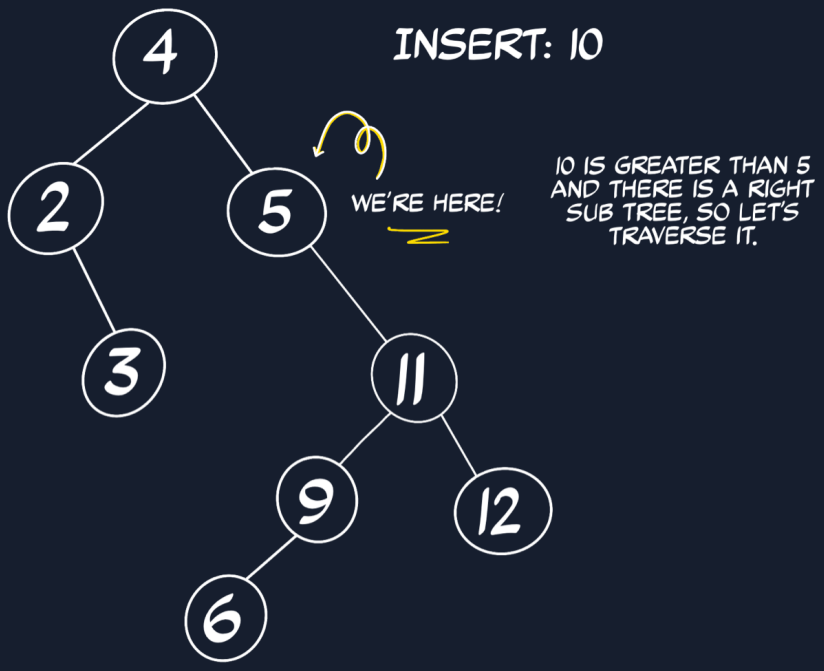
If there are nodes in the tree we must traverse it until we find an appropriate place to add it. We'll initialize a variable, `currentNode` to the root node (because we start at the root when traversing) and a boolean variable, `added`, which will be initialized to false and only updated to true when we have successfully added the node.

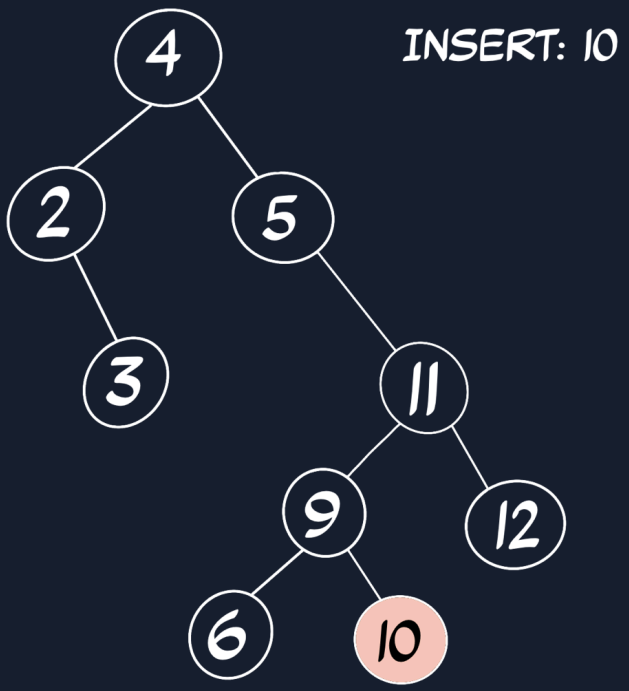
While we haven't added the node and we still have a node whose value we can check our value against, we want to compare the value of the node to be added with the current node's value. If the value we want to add equals the current node's value, return 'No duplicates can be added'.

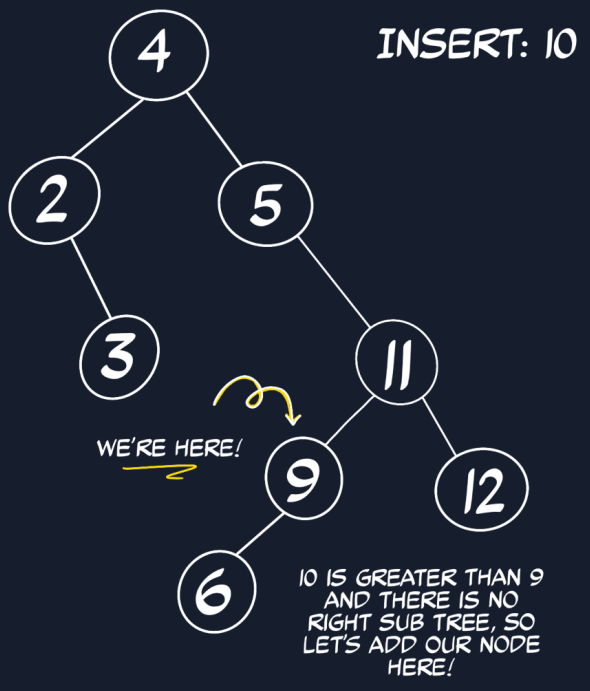
If the value is less than the current node's value, check whether there is a free spot in the current node's left child. If the value is greater than the current node's value, check whether there is a free spot in the current node's right child. If there is, great! We found a spot to add our node and we can update our added boolean to be true. If the spot isn't free we need to traverse down the left sub-tree until we find a spot that is free.

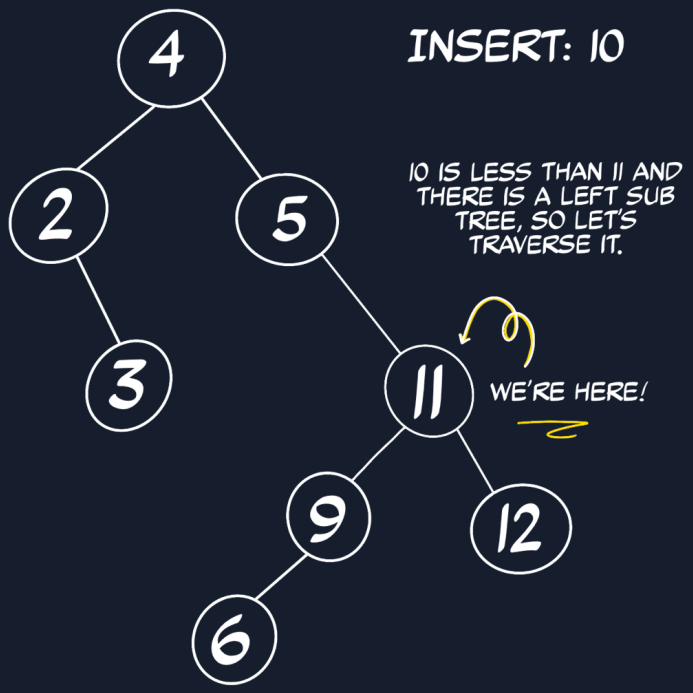
You can see this process illustrated on the next few pages.


START HERE!
4
INSERT: 10
2
5
10 IS GREATER THAN 4 AND THERE IS A RIGHT SUB TREE, SO LET'S TRAVERSE IT.
3
11
9
12
6


INSERT: 10
4
2
5
3
11
9
12
6
WE'RE HERE!
10 IS GREATER THAN 5 AND THERE IS A RIGHT SUB TREE, SO LET'S TRAVERSE IT.


INSERT: 10
4
2
5
3
11
9
12
6
10


INSERT: 10
4
2
5
3
11
WE'RE HERE!
9
12
6
10 IS GREATER THAN 9 AND THERE IS NO RIGHT SUB TREE, SO LET'S ADD OUR NODE HERE!


INSERT: 10
10 IS LESS THAN 11 AND THERE IS A LEFT SUB TREE, SO LET'S TRAVERSE IT.
WE'RE HERE!
4
2
5
3
11
9
12
6

```
addChild(value) {
  if (this.root === null) {
    this.root = new Node(value);
    return;
  } else {
    let currentNode = this.root;
    let added = false;
    while (!added && currentNode) {
      // Don't add duplicates
      if (value === currentNode.value) {
        return "Duplicates cannot be added";
      }
      if (value < currentNode.value) {
        // If the spot is free, add it
        if (currentNode.leftChild === null) {
          currentNode.leftChild = new Node(value);
          added = true;
        } else {
          // Otherwise find the next free spot
          currentNode = currentNode.leftChild;
        }
      } else if (value > currentNode.value) {
        if (currentNode.rightChild === null) {
          currentNode.rightChild = new Node(value);
          added = true;
        } else {
          currentNode = currentNode.rightChild;
        }
      }
    }
  }
}
```

Removing A Node

Removing a node is a bit trickier, so if you don't understand it at first, don't get frustrated. Refer to the illustrations for a visual representation of the theory I'm about to introduce.

When removing a node there are four cases to account for:

- The node isn't found
- The node is a leaf node (no children)
- The node has one child
- The node has two children

To remove a node we must keep track of the parent node. If the node is found, we'll redirect the parent's left or right child (depending upon which node we want to remove) to one of the children (if there are any children) which will splice out the node.

We'll also have a boolean, found, which will keep track of whether or not we've found the node we want to remove.

If the node isn't found, return "The node was not found".

```
removeChild(value) {
  let currentNode = this.root;
  let found = false;
  let nodeToRemove;
  let parentNode = null;
}
```

We must traverse the tree looking for the node we want to delete.

This is a similar process to adding a node (except we're not searching for a free spot, we're searching for an exact value) but the process of searching the left and right subtrees until the value is found remains the same.

Just remember that before traversing the left or right subtree, we have to set `parentNode` equal to `currentNode` so we have a handle on the parent when the deletion time comes.

```
removeChild(value) {
  let currentNode = this.root;
  let found = false;
  let nodeToRemove;
  let parentNode = null;
  // Find the node we want to remove
  while (!found) {
    if (currentNode === null || currentNode.value === null) {
      return "The node was not found";
    }
    if (value === currentNode.value) {
      nodeToRemove = currentNode;
      found = true;
    } else if (value < currentNode.value) {
      parentNode = currentNode;
      currentNode = currentNode.leftChild;
    } else {
      parentNode = currentNode;
      currentNode = currentNode.rightChild;
    }
  }
}
```

Once we've found the node we have to account for the last three cases (leaf node, one child, and two children).

The Node Is A Leaf Node:

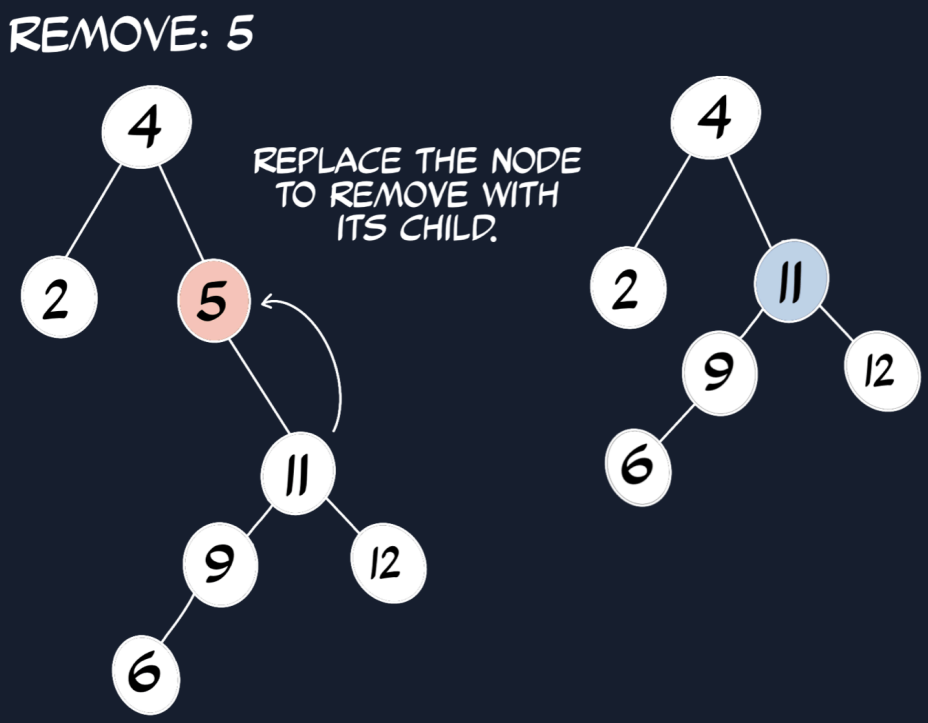
If the node we want to delete is a leaf node (it has no children) we can set parentNode's left or right child to null.

Let's create an additional helper variable for legibility called `nodeToRemoveIsParentsLeftChild` which returns true if the node we're removing is the left child and false if the node we're removing is the right child.

```
const nodeToRmoeveIsParentsLeftChild = parentNode.leftChild === nodeToRemove
// If nodeToRemove is a leaf node, remove it
if (nodeToRemove.leftChild === null && nodeToRemove.rightChild === null) {
  if (nodeToRemoveIsParentsLeftChild) {
    parentNode.leftChild = null
  } else {
    parentNode.rightChild = null
  }
}
```

The Node Has One Child:

If the node we're removing has one child, we can redirect the parent node's pointer to the child.



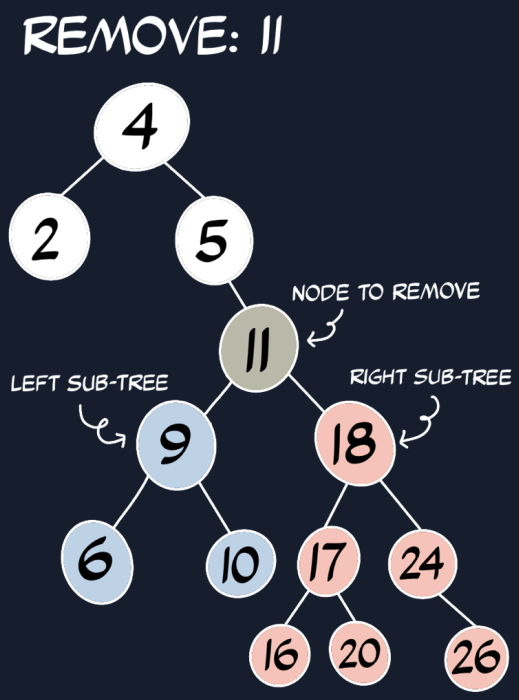
```
else if (nodeToRemove.leftChild !== null && nodeToRemove.rightChild === null) {
  // Only has a left child
  if (nodeToRemoveIsParentsLeftChild) {
    parentNode.leftChild = nodeToRemove.leftChild;
  } else {
    parentNode.rightChild = nodeToRemove.leftChild;
  }
} else if (nodeToRemove.rightChild !== null && nodeToRemove.leftChild === null) {
  // Only has a right child
  if (nodeToRemoveIsParentsLeftChild) {
    parentNode.leftChild = nodeToRemove.rightChild;
  } else {
    parentNode.rightChild = nodeToRemove.rightChild;
  }
}
```

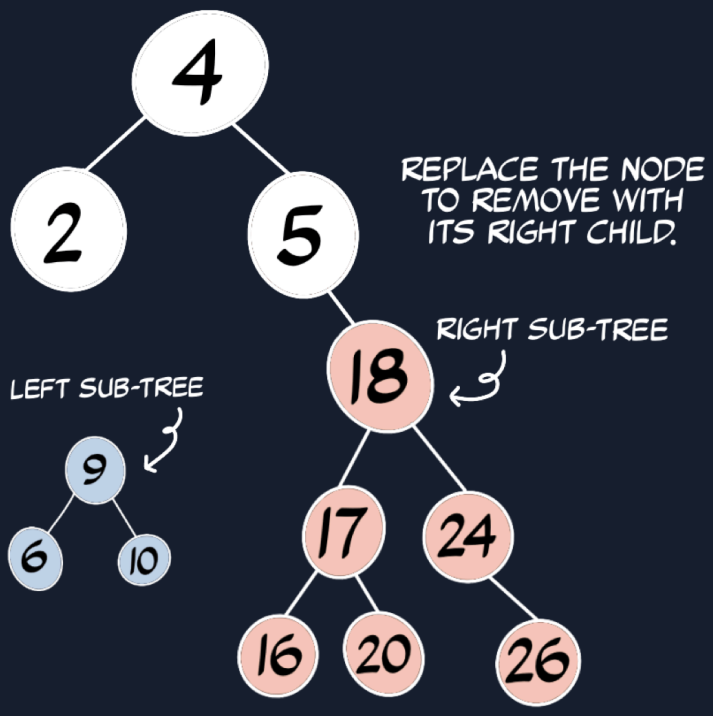
The Node Has Two Children:

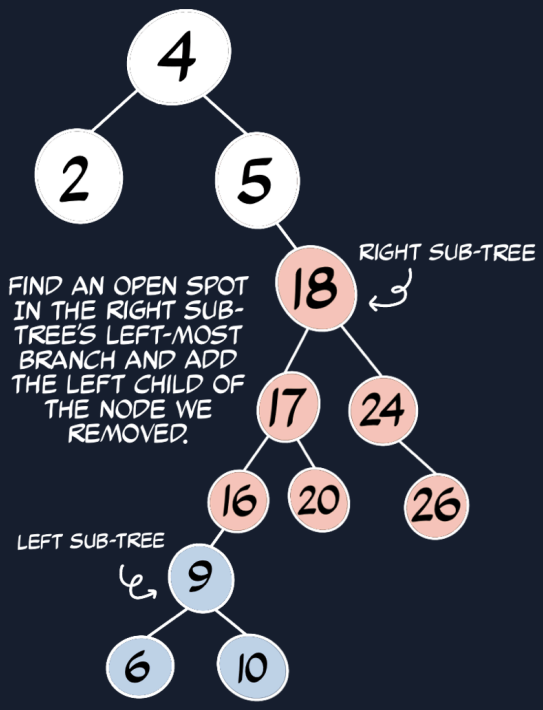
If the node we're deleting has two children, it gets a bit tricker. Here's the algorithm:

- Set the parentNode's pointer to the right subtree of the node we're deleting.
  - If the parent node's right child points to the node we're removing, set `parentNode.rightChild` to the `nodeToDelete.rightChild`.
  - If the parent node's left child points to the node we're removing, set `parentNode.leftChild` to the `nodeToDelete.rightChild`.
- Traverse the right subtree's left branches until we find a free spot.
- Once a free left subtree spot has been found, add the nodeToDelete's left subtree.

If that's confusing don't worry; it confused me too. The following pages contain this concept in illustrated form.


REMOVE: 11
4
2
5
NODE TO REMOVE
11
LEFT SUB-TREE
RIGHT SUB-TREE
9
18
6
10
17
24
16
20
26


4
2
5
REPLACE THE NODE TO REMOVE WITH ITS RIGHT CHILD.
RIGHT SUB-TREE
18
LEFT SUB-TREE
9
6
10
17
24
16
20
26

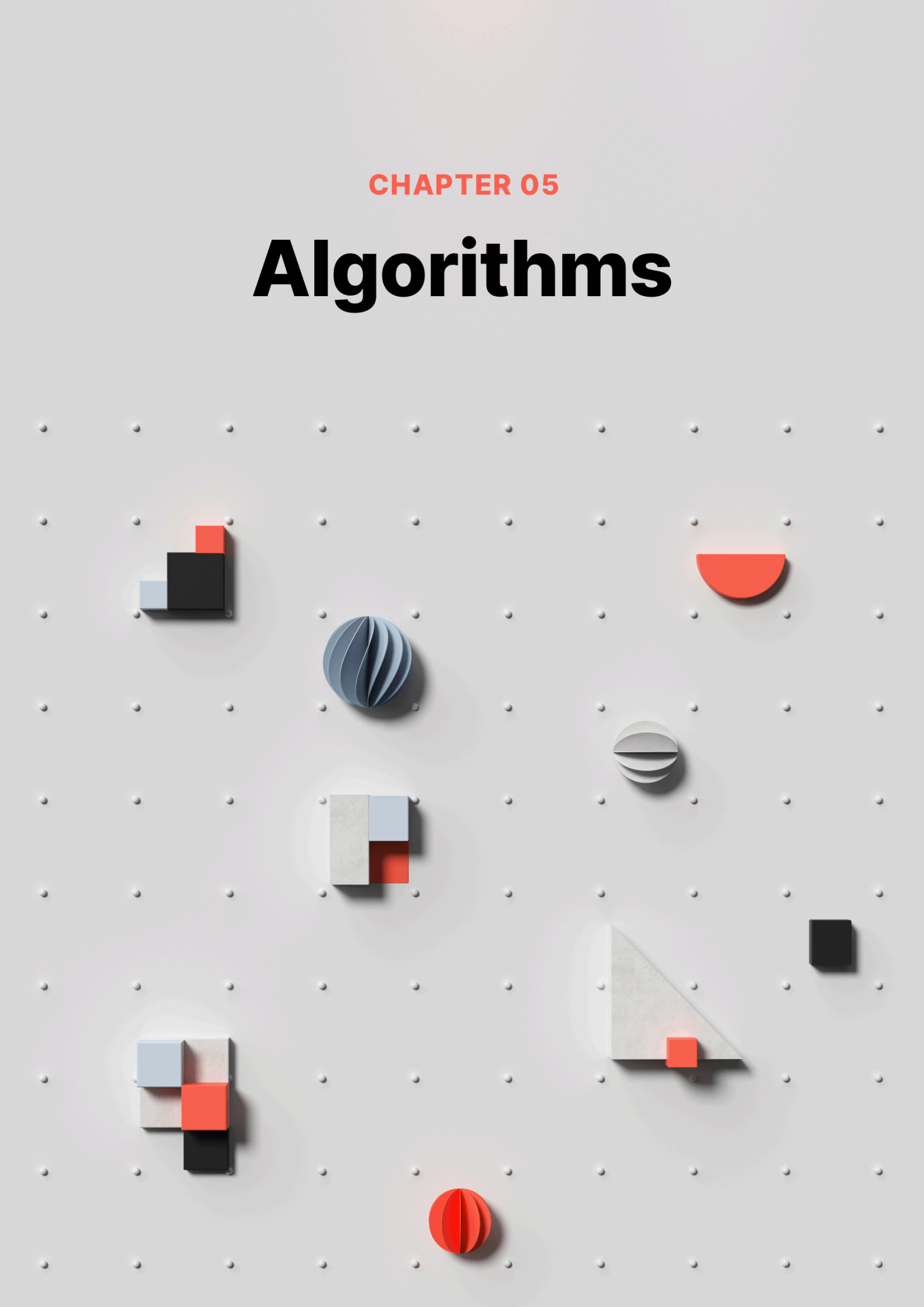

4
2
5
RIGHT SUB-TREE
18
FIND AN OPEN SPOT IN THE RIGHT SUB-TREE'S LEFT-MOST BRANCH AND ADD THE LEFT CHILD OF THE NODE WE REMOVED.
17
24
16
20
26
LEFT SUB-TREE
9
6
10

You can view the full code on my Code Sandbox.

```
  else {
    // Has two children
    const rightSubTree = nodeToRemove.rightChild;
    const leftSubTree = nodeToRemove.leftChild;
    // Set parent node's respective child to the right sub tree
    if (nodeToRemoveIsParentsLeftChild) {
      parentNode.leftChild = rightSubTree;
    } else {
      parentNode.rightChild = rightSubTree;
    }
    // Find the lowest free space on the left side of the
    // right sub tree and add the leftSubTree
    let currLeftNode = rightSubTree;
    let currLeftParent;
    let foundSpace = false;
    while (!foundSpace) {
      if (currLeftNode === null) {
        foundSpace = true;
      } else {
        currLeftParent = currLeftNode;
        currLeftNode = currLeftNode.leftChild;
      }
    }
    currLeftParent.leftChild = leftSubTree;
    return "The node was successfully deleted";
  }
}
```

CHAPTER 05

Algorithms



Understanding Algorithms

An algorithm is a set of instructions which are used to solve a problem. Some algorithms are more efficient than others in regards to the amount of data the algorithm is operating on. This worst-case scenario is known as Big-O notation.

Before we can understand Big-O notation we first have to understand the concept of an algorithm. An algorithm is a series of steps that are taken to solve a problem. We typically think of algorithms as they relate to computer science but we employ algorithms in our daily life without thinking about it.

Suppose you're telling someone how to make a pizza. You might say: "put sauce on the dough, top it with cheese and anything else you want on your pizza. Bake it in a pre-heated 400-degree oven for 12 minutes."

You might think these steps are straight-forward but in reality someone who has never made a pizza or had experience cooking might have a bit of trouble completing the task. Likewise, computers can't necessarily extract logic from our instructions so we have to be as explicit as possible. Re-writing our pizza-making instructions we might end up with something like this: "Turn on your oven by pressing the on button. Set the temperature to 400-degree Farenheit. Press the start button. Put some flour on the countertop. Open the package containing the dough and place on the floured countertop. Stretch the dough until it's a 15-inch circle..." You get the gist.

The first set of instructions were vague with a lot of room for interpretation. The second set of instructions were much more explicit, ensuring you end up with a final product that resembles the desired output.

Algorithmic Complexity With Big-O, Big-θ, Big-Ω

Every algorithm or set of instructions requires time and space to execute, so when we create algorithms to solve our technical problems we must examine the time and space complexity of the execution.

We typically only hear of Big-O notation when discussing algorithmic complexity and performance and there's a reason for this! But it's important to understand Big-θ (Big-Theta) and Big-Ω (Big-Omega) as well.

I stated previously that Big-O notation describes the worst-case performance or complexity of an algorithm. In other words, Big-O tells us the maximum amount of time or the maximum amount of storage this algorithm will need to execute.

We can chart an algorithm's worst-case runtime, or upper bound, on a two-dimensional axis. The x-axis represents the input size and the y-axis represents the amount of time or space this algorithm requires. In other words: As our data size increases, how does it impact the algorithm's performance or space requirement? Let's illustrate this with an example.

```
function findValue(array, value) {
  for (let i = 0; i <Page array.length; i++) {
    if (array[i] === value) {
      return true
    }
  }
  return false
}
```

In this function we're looking for a specific value. The first argument is an array of values to search and the second argument is the value we're searching for. We're using a for-

loop to iterate through each value in the array and check whether it equals the target value. If the value is in the array, return true, otherwise return false.

How long will this algorithm take to run? Well, what is our worst-case scenario? In the worst case, our array will be extremely large and the value we're looking for won't be there, which means we'll have to check every single item in the array and eventually return false. As the length of array increases, the time needed to execute this algorithm increases in direct correlation.

If we have an array with four elements, the maximum amount of times we'll iterate through our array looking for this element is four. Likewise if we have an array with 512 elements, the maximum amount of times we'll iterate through our array looking for this element is 512.

This type of algorithm is known as a linear-time algorithm and is represented as O(n) as the performance of the algorithm grows proportionally to the size of the input. Mapping over an array, filtering an array, cloning an array with the spread operator, and running a task for each item in an array all have O(n) runtime (assuming they don't contain another loop). As the number of data points in the array grows, so does the runtime.

Big-θ and Big-Ω are very similar to Big-O in that they describe bounds on an algorithm. While Big-O measures the worst-case scenario, Big-Ωnotation (Big-Omega) is used to describe the best-case space and time complexity of an algorithm. We call this best-case scenario an asymptotic lower-bound. Going back to findValue example, the best scenario would be if the value we're searching for is found in the first index of the array.

Big-θ (Big-Theta) describes the average case space and time complexity of an algorithm. In the average case of our findValue example we would search half of the array before finding our search value.

You shouldn't need to know Big-θ and Big-Ω for a technical interview, but they're good to be familiar with.

Calculating Complex Big-O Notation

Now that we have a basic understanding of Big-O notation we can begin to look at more expensive calculations. Let's say you're asked to sort an array of integers from lowest to highest value. The brute-force method would be to compare every array element with every other element. This algorithm is known as Bubble Sort and it's notorious for its horrendous performance.

```
function bubbleSortWithForLoops(arr) {
  for (var i = 0; i < arr.length; i++) {
    for (var j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        var temp = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = temp
      }
    }
    return arr
  }
}
```

In the previous code example we had one for-loop so our Big-O notation was pretty straightforward, but in this example we have two for-loops and they're nested one inside of another. What does this mean for our performance?

If you have nested loops, you must multiply their complexities together to find the asymptotic upper bound. Let's break this down a bit:

- Line 2: We have to run through each item in the array, the total number of which we'll call n. So this would be O(n)
- Line 4: We have to run through each item in the array, minus one (since the inner loop only runs to the penultimate item). This would be O(n-1)

- Lines 6, 8-10, 14: You might think this is run for every item in the array (which in the worst case it is) but we evaluate the runtime in isolation. In isolation this if-statement runs once. This is called constant-time and we write it as O(1). As the size of the data (n) increases, it doesn't change how many times this piece of code runs.

To get our overall Big-O value, we now have to evaluate these lines of code in context. The biggest caveat with understanding Big-O notation is that we drop the constants and the less-significant terms.

Big-O describes the long-term growth rate of functions as opposed to their absolute magnitudes. So O(3) is still constant time and will be written as O(1). O(2n) is still linear time and will be written as O(n). Here are a few other examples:

O(n + 3) ⇒ O(n) O(n^3 + n^2) ⇒ O(n3) O(524x + 240y) ⇒ O(x+y) x and y are different values so we can't drop or combine them

Given these parameters, line 4's runtime of O(n-1) will change to O(n) as we drop the less-significant term. Lines 6, 8-10, and 14 are less-significant constant-time terms so we can also drop them. We're left with the outer loop's O(n) and the inner loop's O(n). The question now becomes, what do we do with these n values? Do we multiply them or add them?

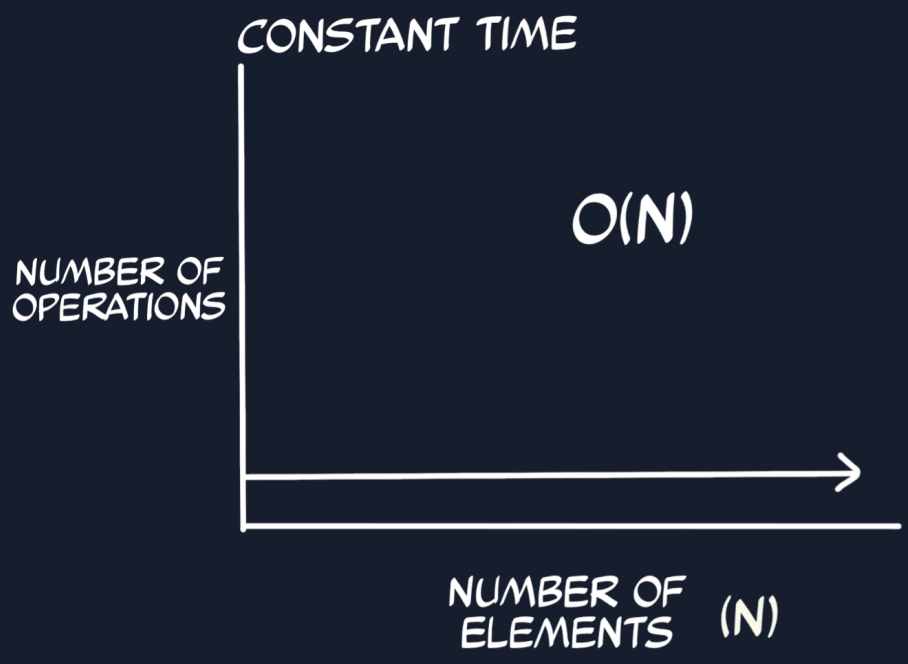
If the loops are nested, the Big-O values will be multiplied. If the loops are adjacent, the Big-O values will be added.

Given the rules above, the final Big-O runtime of our Bubble Sort algorithm is O(n * n) or O(n^2).

Constant Time

O(1), also known as constant time, describes an algorithm which will execute in the same amount of time regardless of the amount of data. Here we define a function that takes no arguments and returns the words “Hello World”. It doesn’t depend on any input and is therefore a constant-time algorithm with O(1).

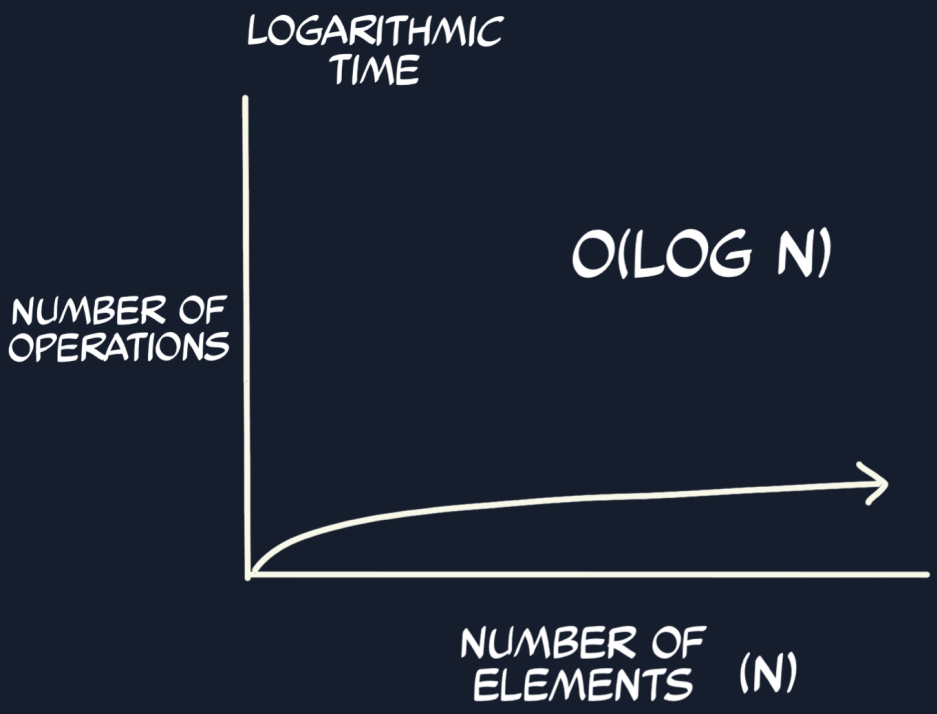
```
function sayHi() {
  return 'Hello world'
}
```



Logarithmic Time

O(log n) describes an algorithm which is logarithmic in nature (the size of the problem decreases by half each time it runs). Binary search, which we'll see later in this chapter, is logarithmic in nature: with each pass through the algorithm the amount of data is halved in size.

```
function binarySearch(list, item) {
  // if list is sorted in ascending order
  let low = 0
  let high = list.length
  while (low <= high) {
    let mid = Math.floor((low + high) / 2)
    let guess = list[mid]
    if (guess === item) {
      return true
    }
    if (guess < item) {
      low = mid + 1
    } else {
      high = mid - 1
    }
  }
  return false
}
```


LOGARITHMIC TIME
O(LOG N)
NUMBER OF OPERATIONS
NUMBER OF ELEMENTS (N)

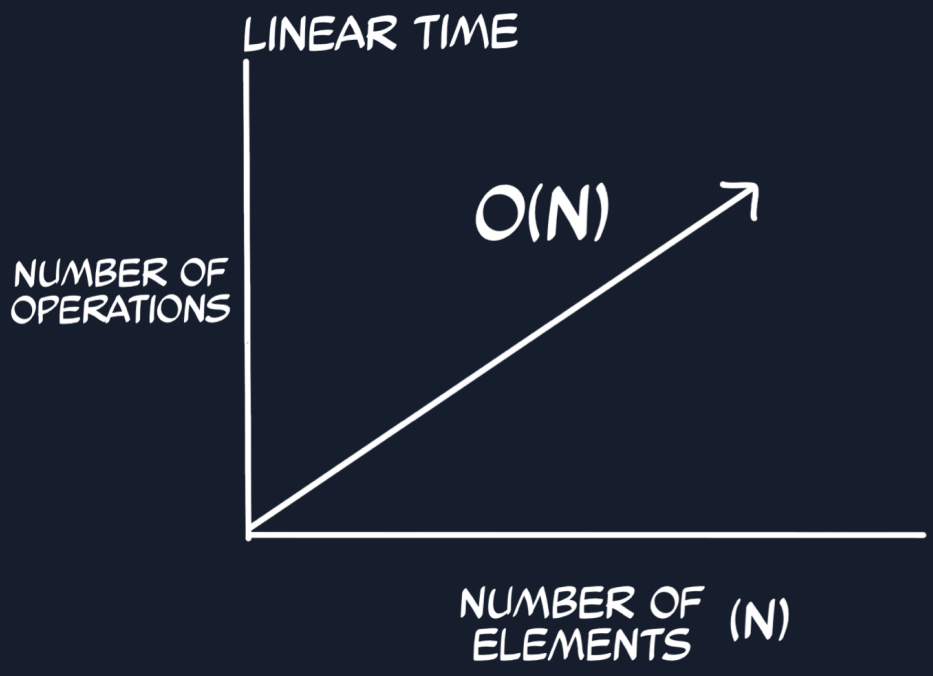
Linear Time

O(n) describes an algorithm whose performance grows linearly and proportionally to the size of the input. A for-loop has O(n) runtime.

The for-loop below runs once for each dog in the array. If n is the length of the dogsArray then this function has O(n), or linear time performance.

Mapping over an array, filtering an array, cloning an array with the spread operator, and running a task for each item in an array all have O(n) runtime, assuming they don't contain another loop. As the number of data points in the array grows, so does the runtime.

```
function printDogs(dogsArray) {
  for (let i = 0; i < dogsArray.length; i++) {
    console.log(dogsArray[i])
  }
}
```

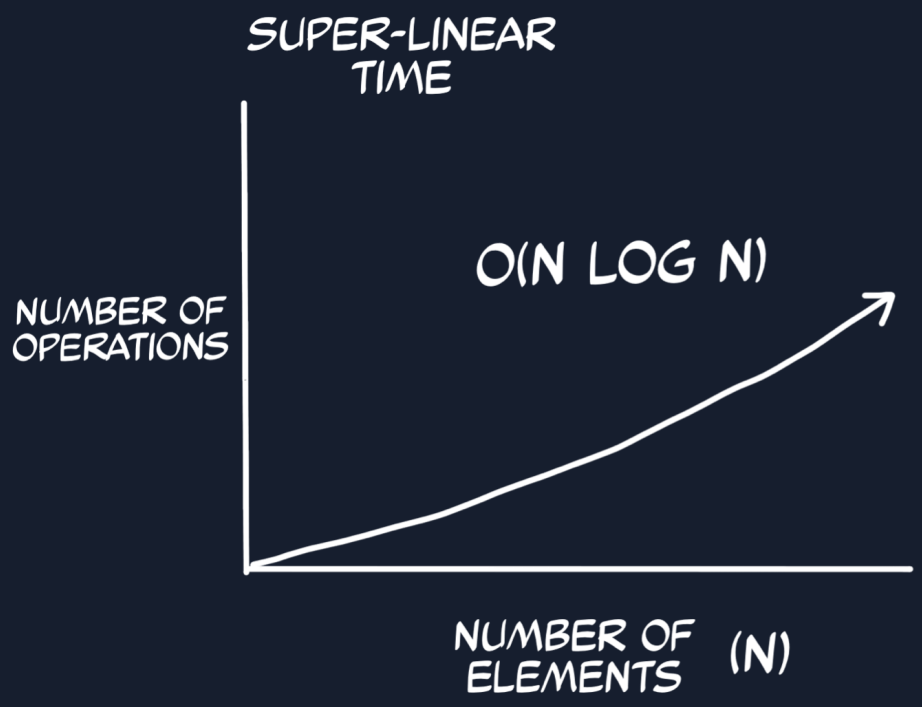


Super-Linear Time

A super-linear time algorithm, which runs at O(n log n), is less- performant than a linear time algorithm but more performant than an exponential algorithm.

As a result, algorithms like merge sort and heap sort are a better choice for sorting data than algorithms like bubble sort.

```
function mergeSort(arr) {
    if (arr.length < 2) {
        return arr;
    }
    const middle = Math.floor(arr.length / 2);
    const left = arr.slice(0, middle);
    const right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort( right));
}
function merge(left, right) {
    const sorted = [];
    while (left. length && right. length) {
        if (left [0] <= right[0]) {
            sorted.push(left.shift());
        } else {
            sorted.push(right.shift());
        }
    }
    let results = [...sorted, ...left, ...right];
    return results;
```

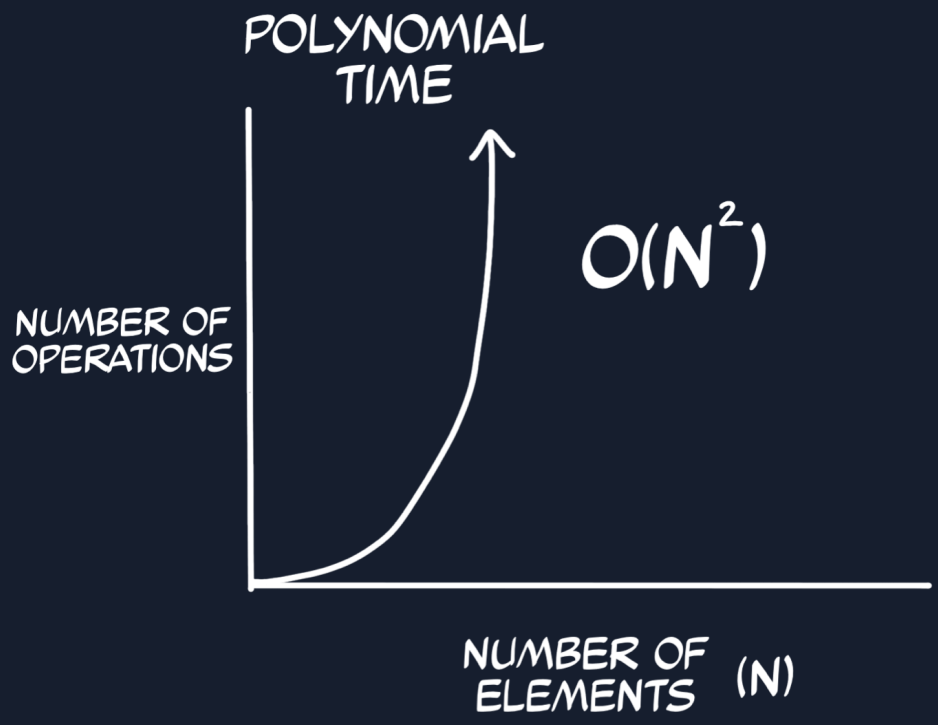

SUPER-LINEAR TIME
O(N LOG N)
NUMBER OF OPERATIONS
NUMBER OF ELEMENTS (N)

Polynomial Time

Polynomial time, or O(n^2) describes an algorithm where the performance is directly proportional to the square of the data size. This is commonly found in algorithms using two nested for-loops, like bubble sort.

The premise of bubble sort is to compare every item in an array with every other item and swap them if the item on the left is greater than the item on the right. This causes O(n^2) comparisons becaus we have to compare n elements against n other elements.

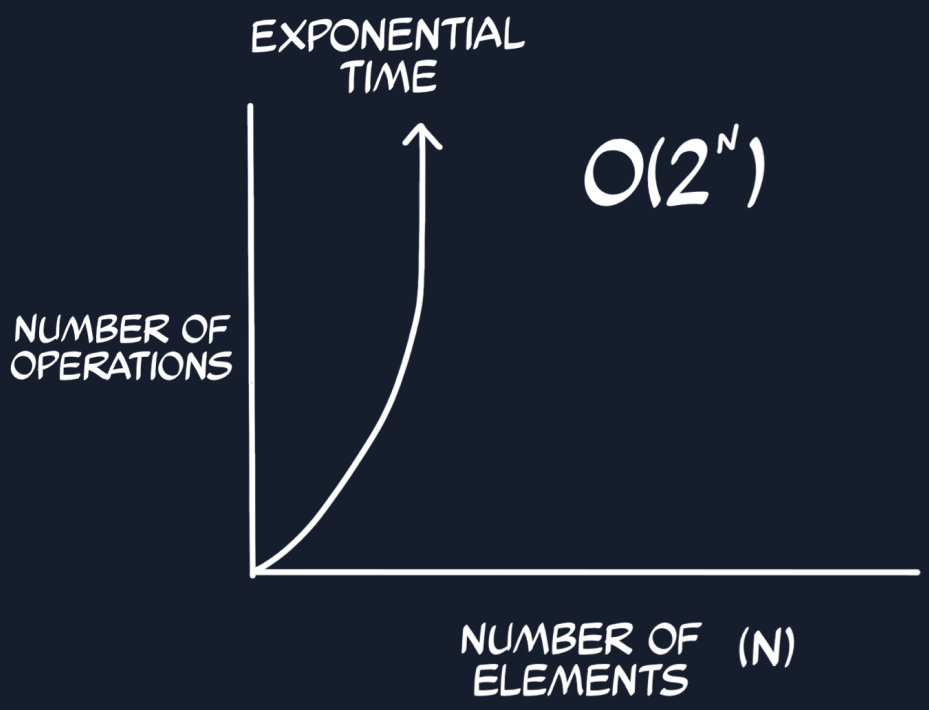
```
function bubbleSort(arr) {
    for (var i = 0; i < arr.length; i++) {
        // Notice that j < (length - i)
        forr (var j = 0; j < arr.length - i - 1; j++) {
            // Compare the adjacent positions
            if (arr[j] > arr[j + 1]) {
                // Swap the numbers
                var tmp = arr[j];
                arr[j] = arrr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
    return arr;
}
```


POLYNOMIAL TIME
O(N²)
NUMBER OF OPERATIONS
NUMBER OF ELEMENTS (N)

Exponential Time

Exponential time algorithms double with each new data point. The recursive Fibonacci algorithm (a recursive function is a function which calls itself), which finds the next number in a sequence by adding the previous two values, is an exponential algorithm.

```
function fibonacci(num) {
  if (num <= 1) return num
  return fibonacci(num - 1) + fibonacci(num - 2)
}
```



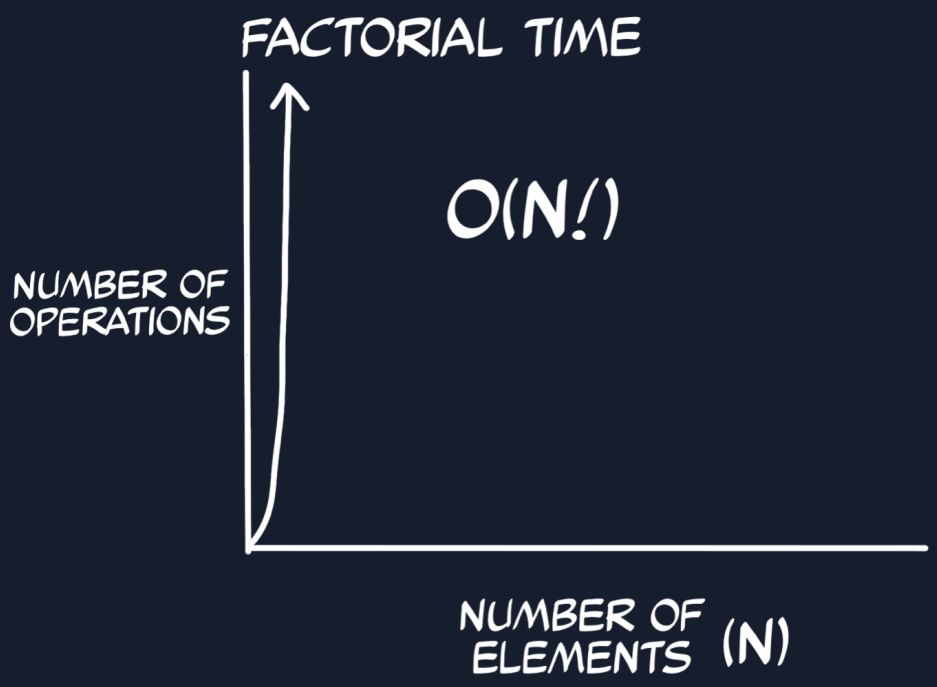
Factorial Time

Factorial time, or O(n!) is the worst-performing algorithm because it grows rapidly as the size of n increases.

The Traveling Salesman problem is a famous problem in computer science wherein the brute-force solution is O(n!).

Traveling Salesman asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?”

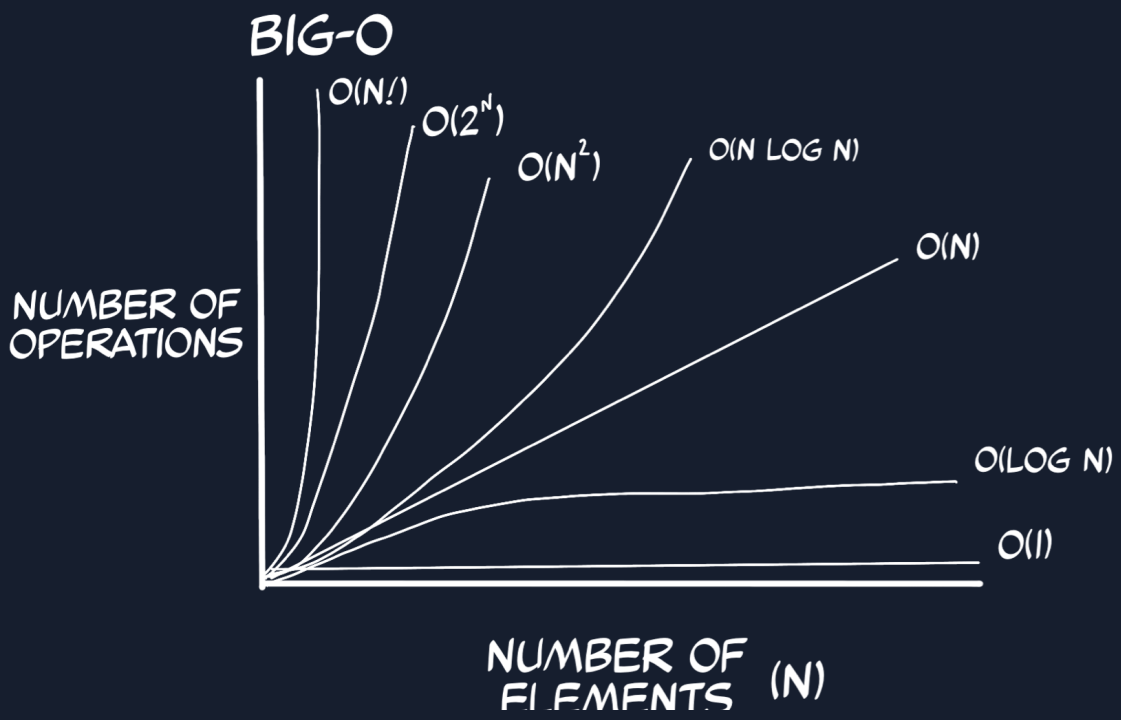
I won’t be writing code for the Traveling Salesman here, but you can read more on Wikipedia.



Graphing Big-O Notation

Here is a graph showing the aforementioned Big-O runtimes and their performance as the size of the input data increases.

Constant time is the most performant runtime and factorial time is the worst.



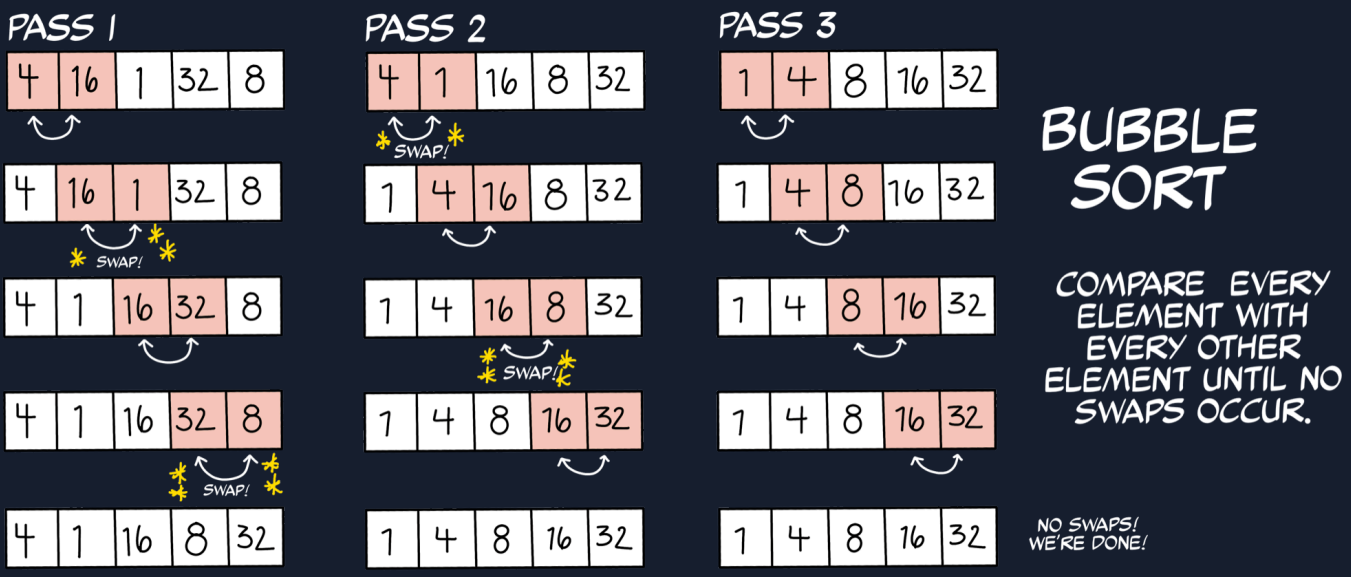
Bubble Sort

Bubble sort is a sorting algorithm where each value in an array is compared to the next and swapped if the value on the left is greater than the value on the right. Bubble sort is notorious for being non- performant, with a runtime of O(n^2).

You can use nested for-loops to code bubble sort, or a do while loop (which reads a bit cleaner and can be more performant).

The do-while loop solution will stop once a full pass is completed without swapping any items. This contrasts the nested for-loop solution which will test every item against every other regardless of if a swap has occurred.

Thus, with the do-while loop solution, even though the Big-O runtime remains O(n^2), as it’s the worst possible scenario, it’s not as likely to have a dataset which requires n^2 passes.


PASS 1
4 16 1 32 8
4 16 1 32 8
SWAP!
4 1 16 32 8
4 1 16 32 8
SWAP!
4 1 16 8 32
PASS 2
4 1 16 8 32
SWAP!
1 4 16 8 32
1 4 16 8 32
SWAP!
1 4 8 16 32
1 4 8 16 32
PASS 3
1 4 8 16 32
1 4 8 16 32
1 4 8 16 32
1 4 8 16 32
1 4 8 16 32
BUBBLE SORT
COMPARE EVERY ELEMENT WITH EVERY OTHER ELEMENT UNTIL NO SWAPS OCCUR.
NO SWAPS! WE'RE DONE!

Coding Bubble Sort In JavaScript

Bubble Sort With For-Loops:

```
function bubbleSortWithForLoops(arr) {
  for (var i = 0; i < arr.length; i++) {
    // Notice that j < (length - i)
    for (var j = 0; j < arr.length - i - 1; j++) {
      // Compare the adjacent positions
      if (arr[j] > arr[j + 1]) {
        // Swap the numbers
        // Temporary variable to hold the current number
        var tmp = arr[j]
        // Replace current number with adjacent number
        arr[j] = arr[j + 1]
        // Replace adjacent number with current number
        arr[j + 1] = tmp
      }
    }
  }
  return arr
}
```

Bubble Sort With A Do-While Loop:

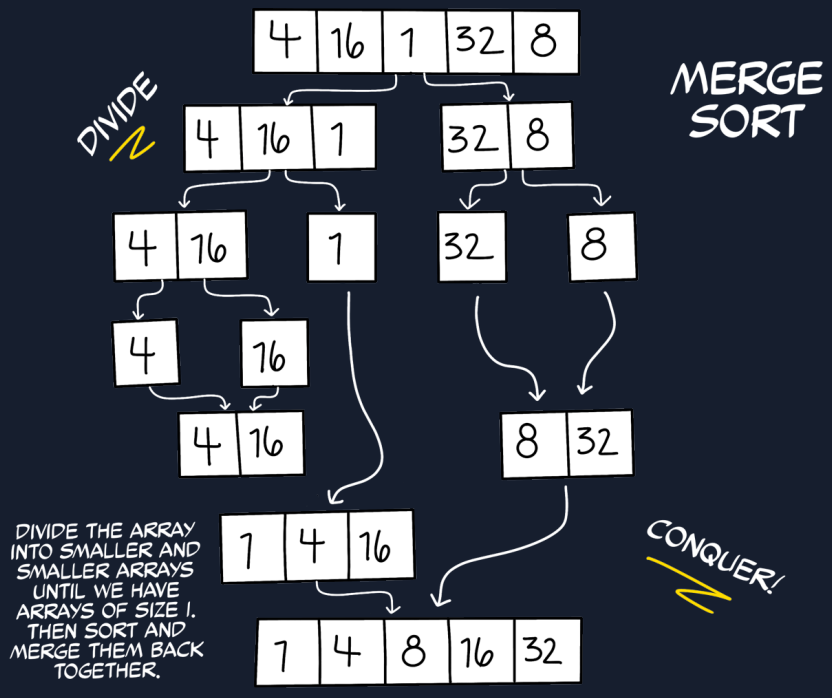
```
function bubbbleSort(arr) {
  let swapped = false
  do {
    swapped = false
    arr.forEach((item, index) => {
      if (item > arrr[index + 1]) {
        let temp = item
        arr[index] = arr[index + 1]
        arr[index + 1] = temp
        swapped = true
      }
    })
  } while (swapped)
  return arr
}
```

Merge Sort

Merge sort is a “divide and conquer” algorithm which means it divides its input array into two halves and recursively calls itself on each half, then merges the two sorted halves back together.

The time complexity of merge sort is O(n log n).

The mergeSort function is responsible for splitting the array into smaller sub-arrays while the merge function is the function which sorts the arrays.


4 16 1 32 8
MERGE SORT
DIVIDE
N
4 16 1
32 8
4 16
1
32
8
4
16
4 16
8 32
DIVIDE THE ARRAY INTO SMALLER AND SMALLER ARRAYS UNTIL WE HAVE ARRAYS OF SIZE 1. THEN SORT AND MERGE THEM BACK TOGETHER.
1 4 16
CONQUER!
N
1 4 8 16 32

Coding Merge Sort In JavaScript

```
function merrgeSort(arr) {
  if (arr.length < 2) {
    return arr
  }
  const middle = Math.floor(arr.length / 2)
  const left = arr.slice(0, middle)
  const right = arr.slice(middle)
  return merge(mergeSort(left), mergeSort(right))
}
function merge(left, right) {
  const sorted = []
  while (left.length && right.length) {
    if (left[0] <= right[0]) {
      sorrted.push(left.shift())
    } else {
      sorted.push(right.shift())
    }
  }
  let results = [...sorted, ...left, ...right]
  return results
}
```

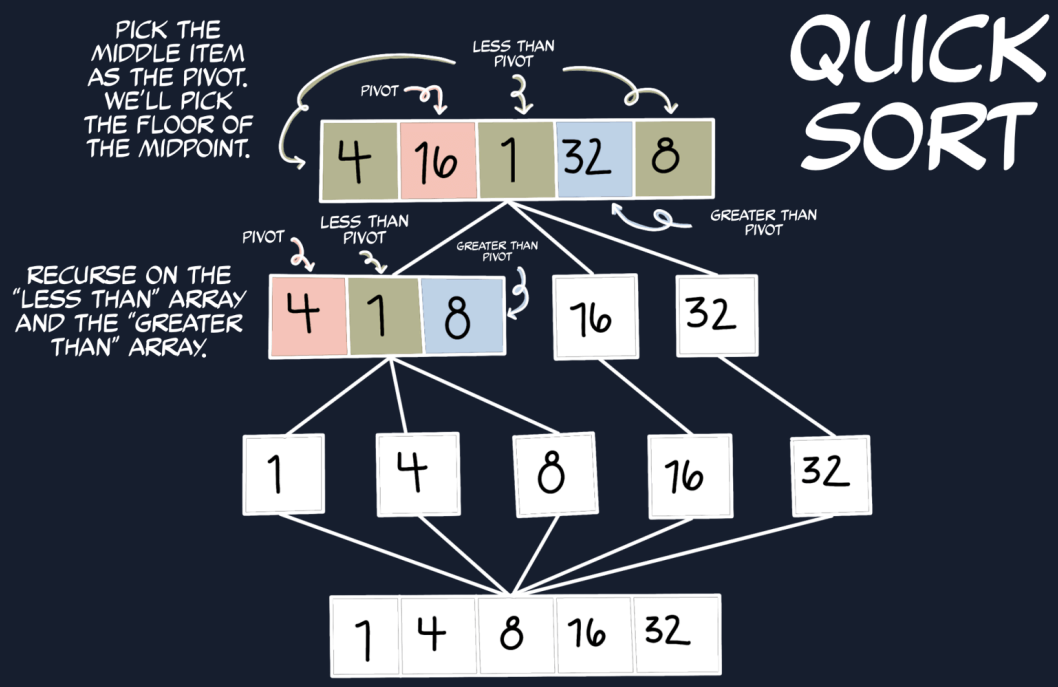
Quick Sort

Like merge sort, quick sort is also a “divide and conquer” algorithm. It picks an element as a pivot element and partitions the array around the pivot.

There are several methods of picking a pivot element:

- Always pick the first element
- Always pick the last element
- Pick a random element
- Pick the median element

There are some thoughts behind which pivot is the most performant, however for our purposes I will select the middle element. On average quick sort has a runtime of O(n log n).


QUICK SORT
PICK THE MIDDLE ITEM AS THE PIVOT. WE'LL PICK THE FLOOR OF THE MIDPOINT.
PIVOT
LESS THAN PIVOT
GREATER THAN PIVOT
4 16 1 32 8
RECURSE ON THE "LESS THAN" ARRAY AND THE "GREATER THAN" ARRAY.
4 1 8
16
32
1
4
8
16
32
1 4 8 16 32

Coding Quick Sort In JavaScript

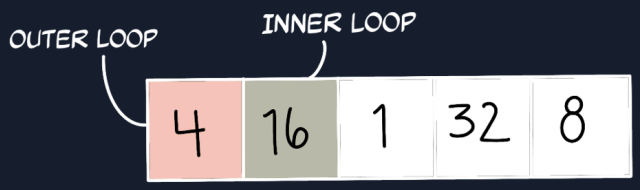
```
function quickSort(array) {
    if (array.length < 2) return array
    let pivotIndex = Math.floor(array.length / 2)
    let pivot = array[pivotIndex]
    let less = []
    let grerated = []
    for (let i in array) {
        if (if !== pivotIndex) {
            arrray[i] > pivot ? greater.push(array[i]) : less.push(array[i])
        }
    }
    return [...quickSort(less), pivot, ...quickSort(greater)]
}
```

Insertion Sort

Insertion sort picks an element in an array and inserts it in its correct position between 0 and the element preceding it.

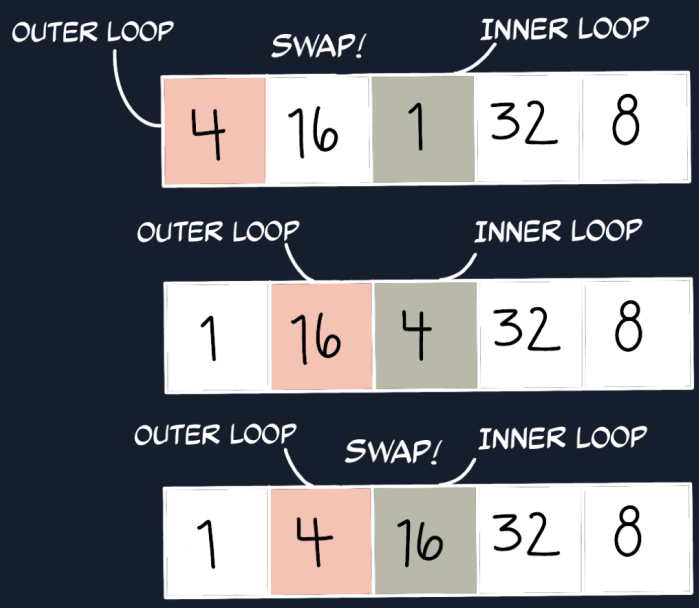
The worst case run-time for insertion sort is O(n^2), as we'll swap every element with every other element.

The following pages illustrate insertion sort.



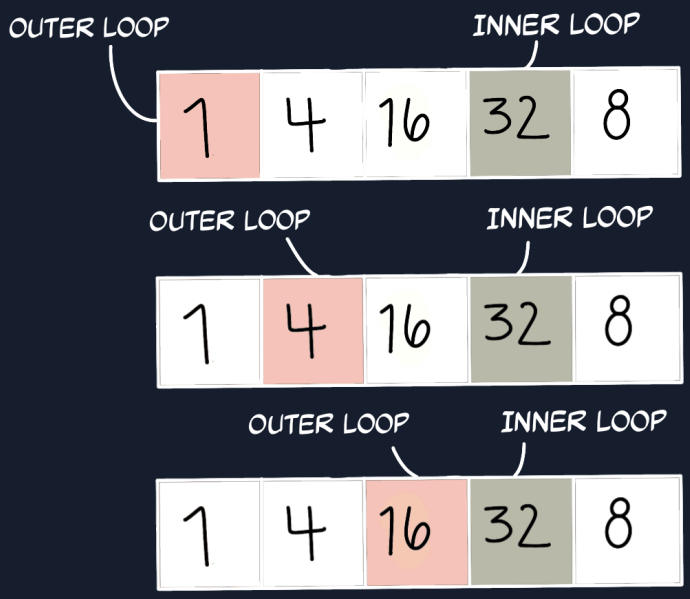
4 IS LESS THAN 16 SO WE DON'T NEED TO SWAP. THIS ITERATION IS COMPLETE.

INSERTION SORT



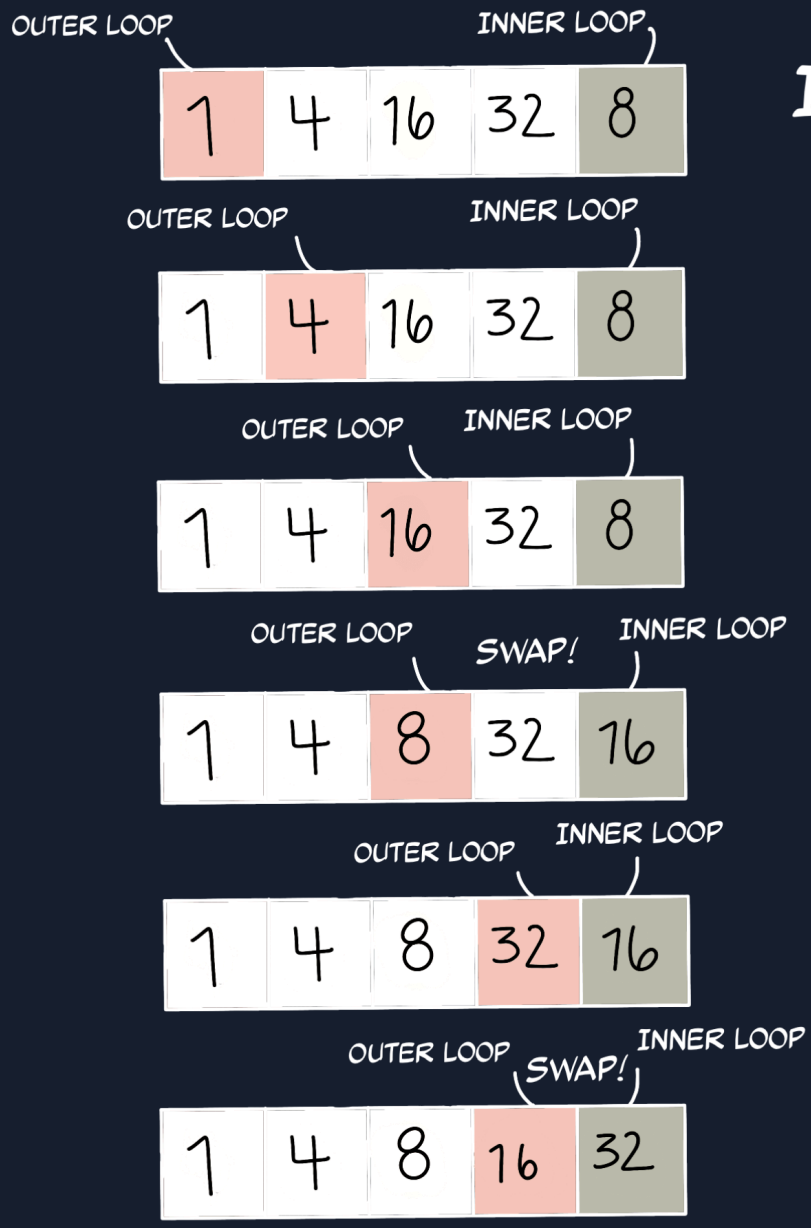
INSERTION SORT

WE CONTINUE TO INCREASE THE OUTER LOOP UNTIL IT REACHES THE INNER LOOP AND SWAP ELEMENTS IF THE ELEMENT AT THE OUTER LOOP IS GREATER THAN THE ELEMENT AT THE INNER LOOP.



INSERTION SORT

NO SWAPS THIS ROUND!


OUTER LOOP
INNER LOOP
1 4 16 32 8
INSERTION SORT
OUTER LOOP
INNER LOOP
1 4 16 32 8
OUTER LOOP
INNER LOOP
1 4 16 32 8
OUTER LOOP
SWAP!
INNER LOOP
1 4 8 32 16
OUTER LOOP
INNER LOOP
1 4 8 32 16
OUTER LOOP
SWAP!
INNER LOOP
1 4 8 16 32

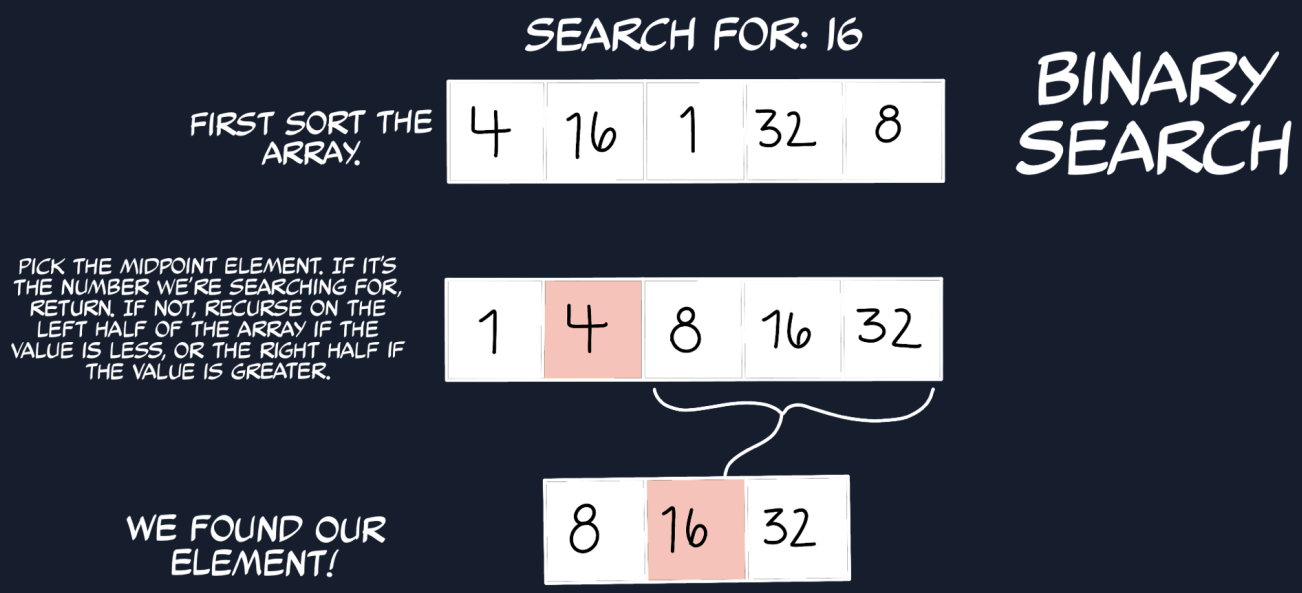
Coding Insertion Sort In JavaScript

```
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    for (let j = 0; j < i; j++) {
      if (arr[i] < arr[j]) {
        const [item] = arr.splice(i, 1)
        arr.splice(j, 0, item)
      }
    }
  }
  return arr
}
```

Binary Search

Binary search is an efficient algorithm for finding an element within a sorted list. It repeatedly divides the list in half until the element is found or there are no other elements left to search.

If your list is already sorted, binary search takes O(log n) as the size of the array is halved each pass through. But if you have to sort the array first, binary search will take O(nlogn) time to complete.



Coding Binary Search In JavaScript

```
function binarySearch(list, item) {
  // Sort the list in ascending order
  list.sort((a, b) => a - b)
  let low = 0
  let high = list.length
  while (low <= high) {
    let mid = Math.floor((low + high) / 2)
    let guess = list[mid]
    if (guess === item) {
      return true
    }
    if (guess < item) {
      low = mid + 1
    } else {
      high = mid - 1
    }
  }
  return false
}
```

Tree Traversals

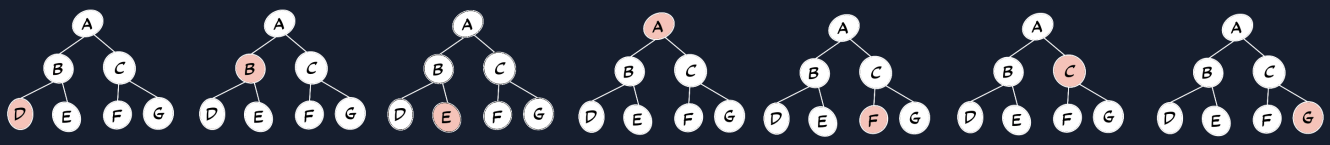
There are several ways to traverse a tree:

- In-order
- Pre-order
- Post-order

Refer to the binary tree section for the tree code.

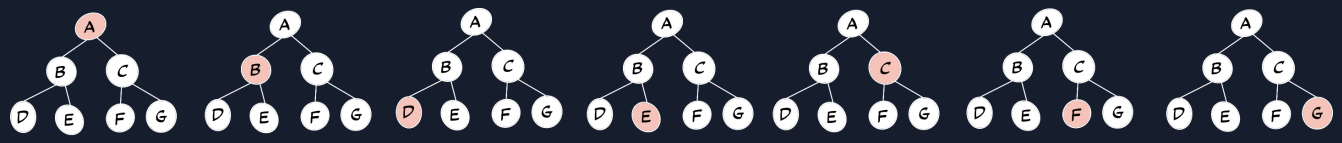
In-Order Traversal

With an in-order traversal, we visit the current node in the order it would naturally fall (left child, current node, right child).



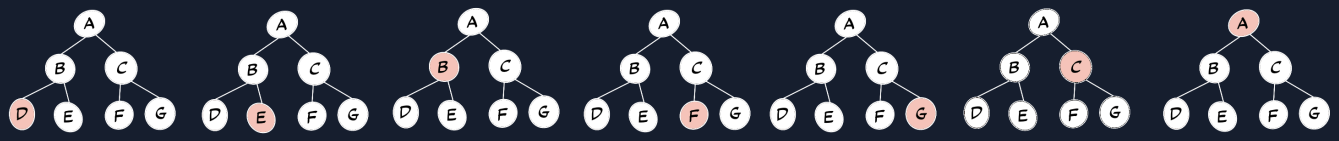
Pre-Order Traversal

With a pre-order traversal, we visit the current node before visiting the child nodes (current node, left child, right child).



Post-Order Traversal

With a post-order traversal, we visit the current node after visiting the child nodes (left child, right child, current node).



Coding Tree Traversals In JavaScript

```
const BINARY_TREE_TRAVERSALS = {
  IN_ORDER: (node, visitFunction) => {
    if (node !== null) {
      BINARY_TREE_TRAVERSALS.IN_ORDER(node.leftChild, visitFunction)
      visitFunction(node)
      BINARY_TREE_TRAVERSALS.IN_ORDER(node.rightChild, visitFunction)
    }
  },
  PRE_ORDER: (node, visitFunction) => {
    if (node !== null) {
      visitFunction(node)
      BINARY_TREE_TRAVERSALS.PRE_ORDER(node.leftChild, visitFunction)
      BINARY_TREE_TRAVERSALS.PRE_ORDER(node.rightChild, visitFunction)
    }
  },
  POST_ORDER: (node, visitFunction) => {
    if (node !== null) {
      BINARY_TREE_TRAVERSALS.POST_ORDER(node.leftChild, visitFunction)
      BINARY_TREE_TRAVERSALS.POST_ORDER(node.rightChild, visitFunction)
      visitFunction(node)
    }
  },
}
```

Tree Search

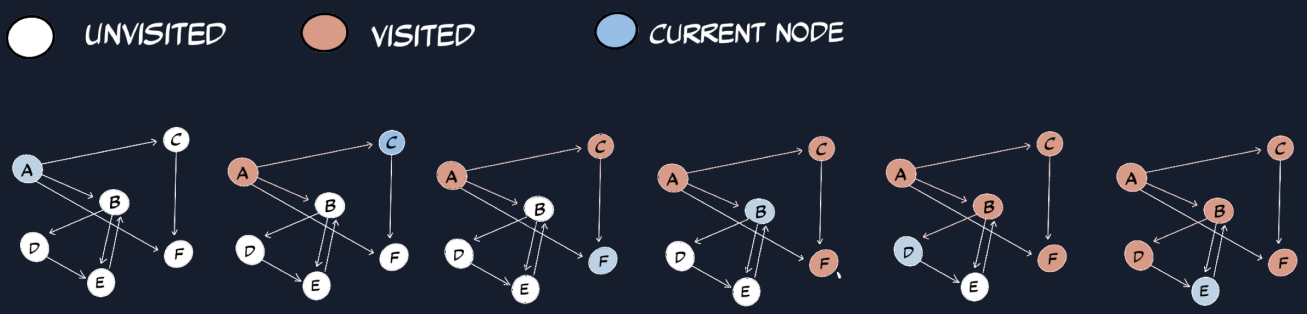
There are two algorithms for searching a tree for a node:

- Depth-first search
- Breadth-first search

Refer to the binary tree section for the tree code.

Depth-First Search (DFS)

In depth-first search we travel as far down each branch (starting with the left) as possible before moving on to the right branch.

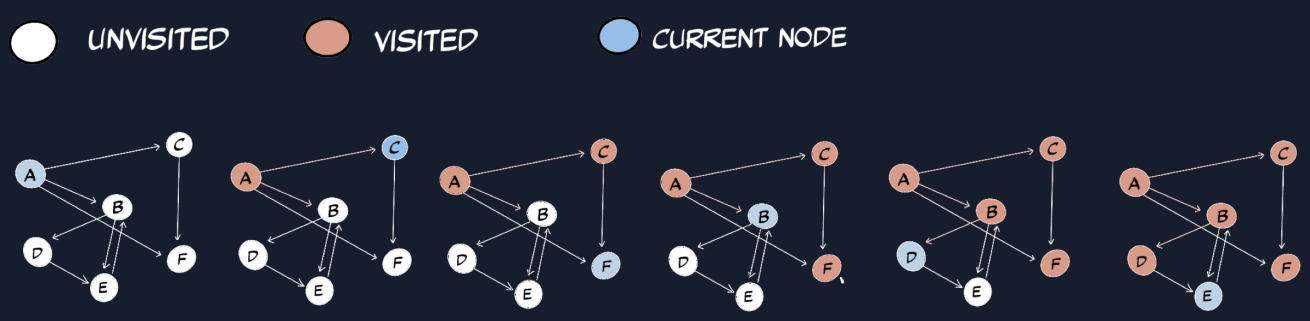


Coding Depth-First Search For A Graph In JavaScript

Let’s build depth-first search for a graph. Here’s what the process looks like.

We can use a hash array to track whether we’ve visited a node before.

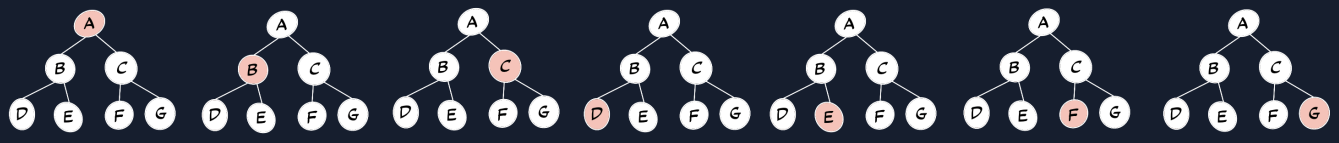
You can check out the full code on my Code Sandbox.



```
function DFS(graph, startValue) {
  let startNode = graph.getNode(startValue)
  let visitedNodesHash = graph.nodes.reduce((accumulator, currentNode) => {
    accumulator[currentNode.value] = false
    return accumulator
  }, {})
  /*
     {
        a: false,
        b: false,
        c: false,
        d: false,
        e: false,
        f: false,
     }
  */
  function exporeNode(node) {
    if (visitedNodesHash[node.value]) return
    console.log(`${node.value} => `)
    visitedNodesHash[node.value] = true
    node.edges.forEach((edge) => exporeNode(edge))
  }
  exporeNode(startNode)
}
```

Breadth-First Search (BFS)

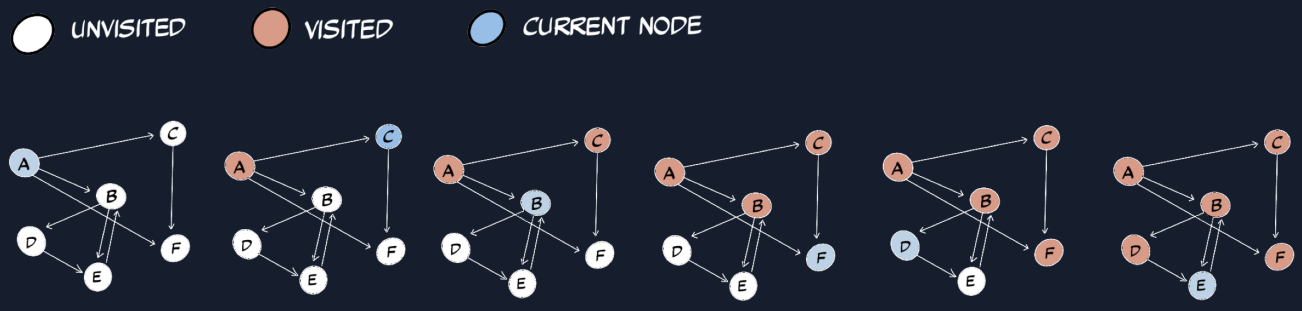
In breadth-first search we explore all nodes on the same tree level before moving to a deeper level.



Coding Breadth-First Search For A Graph In JavaScript

Let’s build breadth-first search for a graph. We can use a queue to keep track of the nodes we still have to visit.

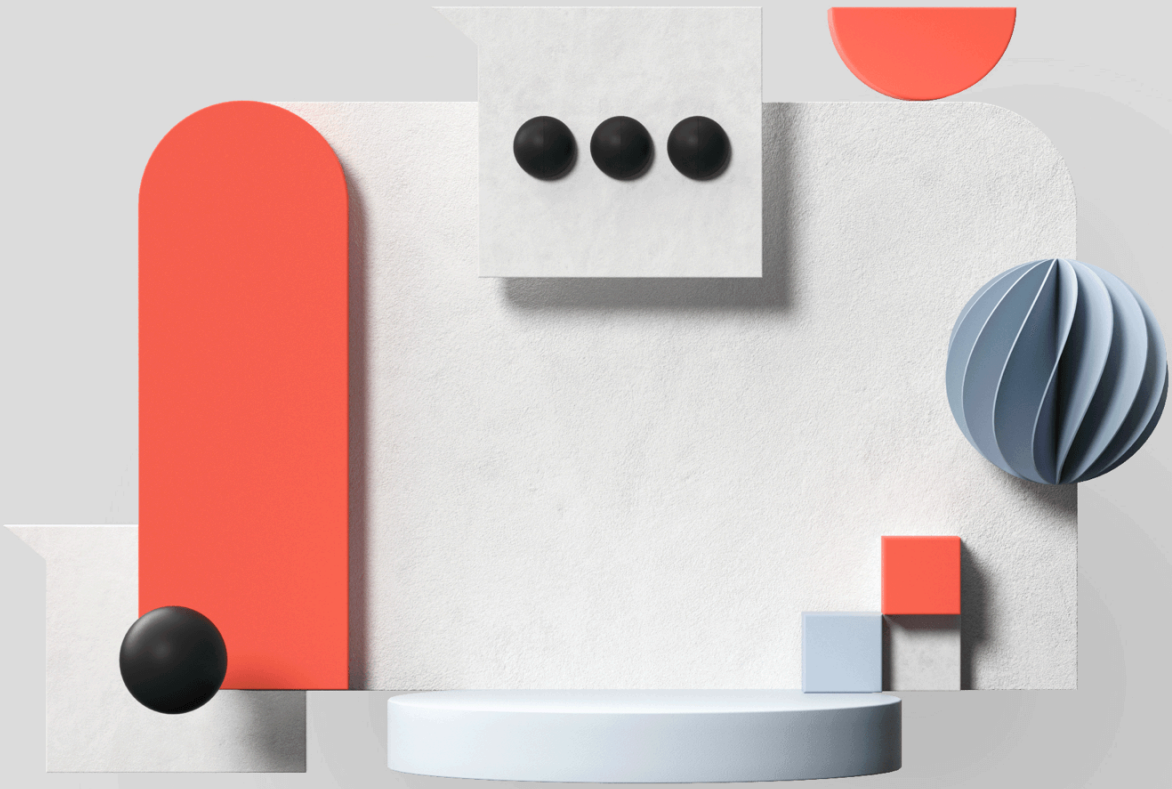
You can check out the full code on my Code Sandbox.



```
function BFS(graph, startValue) {
  let startNode = graph.getNode(startValue)
  let visitedNodesHash = graph.nodes.reduce((accumulator, currentNode) => {
    accumulator[currentNode.value] = false
    return accumulator
  }, {})
  /*
     {
        a: false,
        b: false,
        c: false,
        d: false,
        e: false,
        f: false,
     }
  */
  const queue = new Queue()
  queue.enqueue(startNode)
  while (!queue.isEmpty()) {
    let currentNode = queue.dequeue()
    if (!visitedNodesHash[currentNode.value]) {
      console.log(`${currentNode.value} => `)
      visitedNodesHash[currentNode.value] = true
    }
    currentNode.edges.forEach((node) => {
      if (!visitedNodesHash[node.value]) {
        queue.enqueue(node)
      }
    })
  }
}
```

CHAPTER 06

Front-End Interviews



Front-end interviews will cover all things web. Ranging from HTML and CSS to how the internet works, web security, and performance, front- end interviews were always the most daunting part of a technical interview for me.

I struggle a lot with asynchronous JavaScript so any interview question having to do with promises or infinite scrolling always threw me off.

Remember that everyone struggles with different topics but if you don't know the answer to a question, don't fudge it. Be honest. You can say something to the extent of "To be honest I'm not sure. If I had to guess I would say...". Your honesty indicates that you are self-aware and honest. Your interviewer will appreciate your honesty.

When working through front-end interview questions it's always a great idea to think about user experience (how the user interacts with your application), accessibility, and performance.

If you're tasked with building a UI, discuss how you would use semantic HTML elements, specifically landmark roles like `<nav>` and `<main>` to clearly denote prominent areas within your application as these assist screen readers.

If you're tasked with coding an infinite scroll for a photo sharing app,

you should discuss image performance (webP image format is a great format to learn) and lazy loading.

If you're tasked with building a login form, talk about cross-site scripting attacks and how you should sanitize the incoming data to prevent hackers from altering your database. Talk about client-side data validation and the visual design that would go along with that.

For example your login button could be disabled until the user has entered both a username and password, as this clearly indicates that there is missing information that must be corrected before a user can access their account.

The main takeaway for front-end interviews is to expand the question to areas outside of HTML, CSS, and JavaScript. Think about the end user. Think about performance. Think about accessibility. And think about security.

I won't be covering front-end interview topics in-depth as there are a multitude of resources for learning each of these and frankly they teach them better than I ever could.

I did, however, include detailed study plans at the end of this book which you can reference. These contain specific areas within web development, data structures and algorithms, and systems design, that I recommend studying. These are things that helped me in my interviews and I wager they'll help you too.

But be aware that this is based on my interviewing experiences. You may encounter other topics in your interviews.

HTML

HTML, or Hypertext Markup Language, gets a bad reputation in the frontend development ecosystem as many falsely perceive the language to be insignificant and inferior to more technical aspects of web development, like JavaScript. But HTML is ⅓ of the foundational technologies needed to ace your technical interviews and it shouldn't be taken for granted.

During my Spotify frontend interviews I was asked comprehensive web development questions that equally evaluated my HTML, CSS, and JavaScript skills. I was asked about semantic HTML and accessibility, and you likely will be too.

Here are some of the key concepts of HTML that you should focus on for your frontend interviews. This is not a comprehensive list but it does cover some of the most frequently asked areas of the markup language.

Semantic HTML

Semantic HTML are the set of HTMl elements that describe their meaning to both humans and screen readers. They include elements such as:

- `<header>`
- `<nav>`
- `<a>`
- `<button>`
- `<main>`
- `<aside>`
- `<footer>`
- `<ul>`
- `<ol>`

It's imperative to have knowledge of what differentiates a semantic element like `<nav>` from non-semantic elements like `<div>` and `<span>`.

Semantic HTML Challenge

Below is a code snippet containing non-semantic HTML code. See if you can transcribe this code to use semantic HTML elements.

```
<div>
  <div>
    <div>
      <span>Home</span>
      <span>About me</span>
      <span>My work</span>
      <span>Contact me</span>
    </div>
    <img src="my-logo.png" />
  </div>
  <div>
    <div>Welcome to my portfolio</div>
    <div>Take a look around and if you like what you see, contact me!</div>
    <div>Contact me</div>
  </div>
</div>
```

ARIA

ARIA, or Accessible Rich Internet Applications, are a set of HTML attributes and values that can be added to HTML elements to improve its semantic content and provide supplementary information to screen readers.

There are certain situations in which we must intentionally use a non-semantic HTML element, for example if we're building an accordion component. Here are a few of the most common ARIA attributes you would add to your HTML code to make it more semantic:

- `role`: Allows you to assign a semantic role to a non-semantic element. You can view a comprehensive list of roles on MDN. You do not need to add a role to a semantic element if the role and the element match (i.e. `<nav role="navigation">` is incorrect because the role and the element are the same)

```
<!-- Incorrect because it's redundant -->
<nav role="navigation">...</nav>
<!-- Correct because it provides a semantic role for an element which -->
<!-- doesn't have a native HTML element -->
<div role="dialog">...</div>
<!-- Correct because it changes the list item element into a tab role -->
<!-- to indicate this is part of a tab container -->
<ul>
  <li role="tab">...</li>
  <li role="tab">...</li>
  <li role="tab">...</li>
</ul>
```

- **`aria-visible`**: Removes an element and all of its children from the accessibility tree. This can be purely decorative content like icons or images, duplicated content, or collapsed content like an off-screen navigation menu.

```
<!-- Must be manually updated when the nav is visible -->
<nav aria-visible="false">...</nav>
<!-- Automatically updated when the nav is visible -->
<nav aria-visible="{isVisible}">...</nav>
```

- **`aria-labelledby`**: Contains the id of the element that labels the element the labelledby attribute is appended to.

```
<!-- Use the for attribute on the label to create
a semantic relationshp with the text input -->
<label for="username">Username</label>
<input type="text" id="username" />
<!-- Use the aria-labelledby attribute to create
a semantic relationshp with the text input -->
<h2 id="username">Username</h2>
<input type="text" aria-labelledby="username" />
```

- `aria-label` : Contains the text of the label since there is no visible text element that contains this text. Use only if aria-labelledby isn't possible.
- `aria-expanded` : Contains a boolean that indicates whether or not the container is expanded.

```
<!-- Custom accordion component -->
<div aria-expanded="{isExpanded}" aria-label="Read more about my work">...</div>
```

Understanding when to use ARIA in your HTML is extremely important as it shows interviewers that you care about accessibility. Here are a couple resources for learning more about web accessibility!

- MDN Web Accessibilty
- The Bootcamper's Guide To Web Accessibility by Lindsey Kopacz

Tab Index

Tabindex is an attribute that designates whether an element can be focused on with sequential keyboard navigation.

- A negative tabindex value means that the element cannot be reached via sequential keyboard navigation (although you may be able to focus on the element using JavaScript or the mouse).
- A positive tabindex value means the element is focusable in sequential keyboard navigation defined by the value number. For example an element with a tabindex=”5” will be focused on before an element with a tabindex=”6”. The default tabindex value is 0.

Tabindex is an attribute that designates whether an element can be focused on with sequential keyboard navigation.

- A negative tabindex value means that the element cannot be reached via sequential keyboard navigation (although you may be able to focus on the element using JavaScript or the mouse).
- A positive tabindex value means the element is focusable in sequential keyboard navigation defined by the value number. For example an element with a tabindex=”5” will be focused on before an element with a tabindex=”6”. The default tabindex value is 0.

You shouldn’t use tabindex with non-interactive content because this prevents screen readers and other assistive technology from navigating to and manipulating the component.

CSS

CSS, or cascading style sheets, is the language used to apply styling to our HTML content.

CSS can be extremely tricky, but hang in there; I promise it gets easier with time and experience.

Here are some specific areas of CSS that you should study up on to prepare for your front-end interviews.

Beginner & Intermediate CSS

- Specificity
- Box model
- Position
  - Relative
  - Fixed
  - Absolute
- Display
  - Block
  - Inline
  - Inline-block
  - Flex
  - Grid

- Media queries
- Hex vs. RGB vs. RGBA
- Pseudo-selectors
- Units
  - Pixels
  - Em
  - Rem

Advanced CSS

- Transitions
  - Transition timing functions
  - Bezier curves
- Combinators
- Pseudo-elements

JavaScript

JavaScript is the language we use to add behavior and interaction to our applications.

There is a ton to learn with JavaScript, but don't get discouraged; it takes time. I've included a comprehensive list of resources for learning JavaScript in chapter 8.

Here are some specific areas of JavaScript that you should study up on to prepare for your front-end interviews.

Foundational JavaScript

- Scope
  - Var
  - Let
  - Const
- Switch statements vs. if/else statements
- Loops & iteration
  - For-each
  - For-of
  - For-in

- Primitive vs. reference types
  - Strings
    - charAt()
    - slice()
    - includes()
    - match()
    - repeat()
    - replace()
    - startsWith()
    - toLowerCase()
    - toUpperCase()
    - substring()
    - split()
    - trim()
  - Numbers
    - isNaN()
  - Undefined
  - Null
  - Arrays
    - Spread operator (...)
    - find()
    - forEach()
    - every()
    - some()
    - filter()
    - includes()
    - indexOf()
    - join()
    - map()

    - pop()
    - push()
    - reduce()
    - reverse()
    - shift()
    - unshift()
    - slice()
    - splice()
- Objects
    - The prototype chain & inheritance
    - entries()
    - keys()

- Math
  - Modulus
  - floor()
  - ceil()
  - pow()
  - random()
  - min()
  - max()
  - sqrt()
- Event delegation / bubbling
  - Prevent default

Intermediate / Advanced JavaScript

- Data structures
  - Maps
  - Sets
  - Symbols
- Closures
  - setInterval()
  - setTimeout()
- Recursion
- DOM manipulation
- Regular expressions (regex)
- Immediately invoked function expressions (IIFE)
- Hoisting

- Functional programming
  - Higher-order functions
  - Immutability
  - Pure functions
  - First-class functions
- Debounce/throttle
- Asynchronous programming
  - Promises
  - Async/await
  - Callback functions

The Web

You will likely receive interview questions about how the web works during your front-end interviews so it's important to read up on concepts you may not have otherwise learned in-depth.

Here are some specific areas of the web that you should study up on to prepare for your front-end interviews.

- TCP/IP
- HTTP/HTTPS
- CORS (Cross-origin resource sharing)
- Security

UX / Visual

During your front-end interviews you may be asked to design a user interface or discuss user experience, but design isn't always something we learn during a computer science degree or at a bootcamp.

I've linked some resources in chapter 8 for learning about UX and visual design.

Here are some specific areas of design that you should study up on to prepare for your front-end interviews.

UX Design

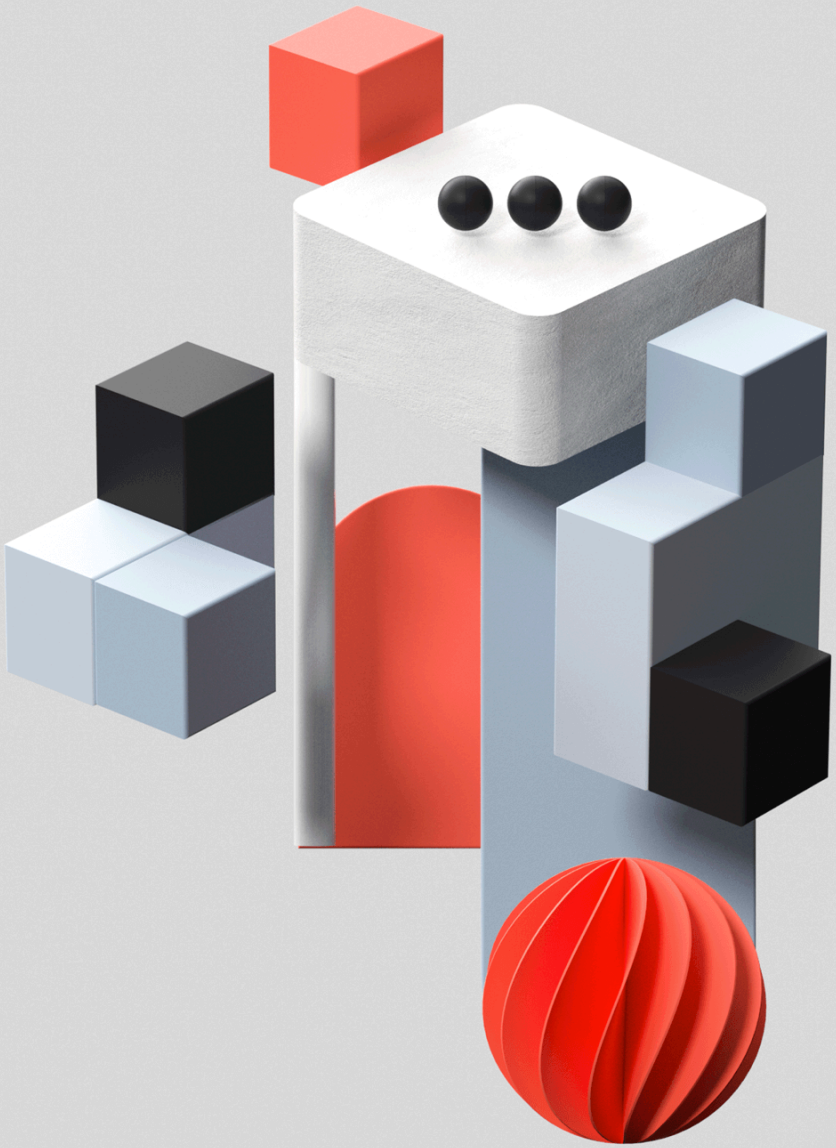
- Information architecture
- Heuristics
- User flows

Visual Design

- Typography
- Color
- Layout & spacing

CHAPTER 07

Systems Design Interviews



Systems Design

During your interview process you may encounter a systems design interview (I've had a couple of them myself).

These interviews can seem daunting as a front-end developer because they touch areas of computing that we wouldn't interact with on a day-to-day basis.

One of my favorite courses for learning systems design is Grokking The Systems Design Interview on educative.io. This is the most comprehensive tutorial I've found and will give you all the skills you need to succeed.

Here are some specific areas of systems design that you should study up on to prepare for your systems design interviews.

- Key characteristics of distributed systems
- SQL vs. NoSQL databases
- Load balancers
- Caching
- Data partitioning
- Replication / redundancy
- Database schemas

Foundations Of Systems Design

When working with enterprise applications we must consider the implications of our technical decisions on the larger system. It's a much smarter decision to sketch out your system architecture prior to investing in technologies and hardware.

A distributed system is a system with many components that are located on different machines. These machines communicate with one another to complete tasks such as adding a new user to the database and returning data from API requests. Despite being a distributed system with many moving parts, the user sees one cohesive process.

Characteristics Of Distributed Systems

If you do a Google search for “Key characteristics of distributed systems” you’ll get some variance in words. Some of the words and phrases you’ll encounter include “concurrency”, “fault tolerance”, and “scalability”. These words can seem intimidating if you don’t have experience with systems design so let’s break down a couple of these characteristics.

Scalability

For a system to be scalable it must be able to adapt to growing demand whether that demand is number of users or an increase in network requests. Most systems performance is inversely proportional to the system size meaning that performance worsens as the system grows. But it's important to note that systems can scale both horizontally and vertically. Let's discuss what that means.

Horizontal Scaling

Scaling a system horizontally means adding more servers to the pool of resources. Horizontal scaling might be easier than vertical scaling as you can purchase a new server and have it up and running quickly.

Cassandra and MongoDB are two database management systems that use horizontal scaling.

Vertical Scaling

Vertical scaling, in contrast, requires adding more power, such as CPU (Central Processing Unit) or RAM (Random Access Memory), to your existing servers. Vertical scaling is often much more difficult to achieve as you're limited to a single server, so scaling past that server's capability often requires downtime and has an upper-bound.

MySQL is one example of a vertically scaling database.

Reliability / Fault Tolerance

Reliability, or fault tolerance, is the probability that a system will fail during a given period of time. A system is reliable if it continues functioning when its hardware or software fails. Large commerce sites are one type of business which heavily prioritize reliability as a key characteristic of their system. In 2013 Forbes calculated the cost of Amazon's downtime and they concluded that for every minute that their system is down, Amazon loses approximately \$66,240.

If Amazon customers are adding items to their cart or finalizing transactions when Amazon's servers lose power, there will be many angry customers knocking at the doors of customer service. Thus, having a reliable system that continues to function during outages is a high priority for Amazon.

Load Balancing

Load balancing is one of the most vital aspects of systems design as it is the process that determines how your servers receive requests. Load balancing not only distributes traffic to servers but also monitors the status of the servers while distributing traffic. If a server suddenly becomes unavailable to accept new requests, the load balancer will prevent new requests from reaching that server.

Load balancers sit between the client and the server. They use various algorithms to determine how traffic should be distributed to different servers. By distributing traffic, load balancers prevent any individual server from being inundated with requests.

Load Balancing Algorithms

There are several algorithms we can use to distribute traffic from the client to the servers. Let's take a look at a few of these.

Least Connection Method

The least connection method distributes traffic to the server with the least active connections.

Least Response Time Method

The least response time method distributes traffic to the server with the least amount of active connections and the quickest average response time.

Lowest Bandwidth Method

The lowest bandwidth method distributes traffic to the server that is currently serving the least amount of traffic (measured in megabits per second, or Mbps).

Round Robin Method

The round robin method iterates through each server and sends each new request to the next server. When the last server in the cluster is reached, the load balancer starts over from the first server in the cluster.

Weighted Round Robin Method

The weighted round robin method is very similar to the round robin algorithm with one distinction: it can account for servers with different processing capabilities. Every server in the cluster is assigned a weight that indicates its processing capability and servers with higher weights receive new connections before servers with lower weights.

IP Hash Method

The IP hash method calculates the IP address of the requesting client and using this IP

hash redirects the request to a server.

Caching

Caching is the process of saving responses to be quickly served the next time that response is requested. Cache is limited and can be expensive but if managed properly can drastically improve the response time to your clients.

Cache Invalidation

Due to the fact that space in cache is sought-after, having an appropriate cache invalidation method is extremely important. It's imperative to have an up-to-date cache. So what happens when our data changes?

When our data is updated we must invalidate our cache. Let's take a look at three cache invalidation methods.

Write-Through Cache

With write-through cache, data is saved in cache as well as its corresponding database simultaneously which allows for quick retrieval. Additionally, having cache and database parity ensures consistency, making the system more secure in the event of a power failure or system error.

While write-through cache appears like a great method, and it is, each operation must be done twice (once in cache and once in the database) before it sends the response to the client so there is higher latency than other cache invalidation methods.

Write-Around Cache

With write-around cache, we write directly to the database and ignore cache completely. While this can prevent the cache from being inundated with write operations, the next time a read request is made for a piece of data it will cause a cache-miss and the response must be read from permanent storage.

Write-Back Cache

With write-back cache, we write solely to cache before sending the response to the client. This method provides the lowest latency (or wait time) however it can lead to data-loss in the event of system failure as we only write data from cache to permanent storage at particular intervals.

Cache Eviction Policies

How do we determine which data should be removed from cache to reduce latency? Let's take a look at a few cache eviction policies.

First-In-First-Out

With the first-in-first-out eviction policy, the first-accessed block is evicted regardless of how often it's accessed.

Last-In-First-Out

With the last-in-first-out eviction policy, the most recently-accessed block is evicted regardless of how often it's accessed.

Least Frequently Used

With the least frequently used eviction policy, the least frequently accessed block is evicted.

Most Frequently Used

With the most frequently used eviction policy, the most frequently accessed block is evicted.

Least Recently Used

With the least recently used eviction policy, the least recently accessed block is evicted.

Most Recently Used

With the most recently used eviction policy, the most recently accessed block is evicted.

Random Replacement

With the random replacement eviction policy, a random block is evicted.

Data Partitioning

Data partitioning is the method by which a large database is segmented and optimizing data partitioning can have a massive impact on the speed of your system.

Data Partitioning Methods

There are several methods for partitioning data. Let's take a look at a few of these.

Horizontal Partitioning

With horizontal data partitioning (also known as range-based partitioning and data sharding), different rows are placed into different tables.

For example let's say we are keeping track of a list of students in a school system. With horizontal partitioning we would place students having last names from A through C into one table, D through F in another table, and so on.

One problem with this approach is that perhaps there are hundreds more students with "F" last names than students with "Q" last names. This can lead to unbalanced servers.

Vertical Partitioning

With vertical data partitioning, data is separated by schema. For example if we head back to our school system example, vertical data partitioning would allow us to place all student information on one server, all class information on another server, and all teacher information on a third server. Simply put: related data is stored within the same table.

One issue with vertical partitioning is the inability to scale when the data becomes too large. If we have an influx of new students, we may have to split student data across multiple tables.

SQL vs. NoSQL Databases

There are two primary types of databases: SQL (relational) and NoSQL (non-relational).

SQL Databases

SQL databases, or relational databases, store data in rows, columns, and tables. A row includes information about one specific entity (i.e a student) and a column includes information about each different data points (i.e. name, age).

Examples of SQL databases include MySQL, Orace, and Postgres.

NoSQL Databases

NoSQL databases, or non-relational databases, store data in different ways. Let's take a look at a few of these.

Graph Databases

In a graph database, data is stored using the graph data structure (nodes with edges). One example of a graph database includes Neo4J.

Document Databases

In a document database, data is stored in documents which are grouped into collections where each document can have its own structure. Two examples of document databases are CouchDB and MongoDB.

Wide-Column Databases

In a wide-column database, data is stored in a column family, which is similar to a container for rows. One example of a wide-column database is Cassandra.

Key-Value Databases

In a key-value database, data is stored in an array of key-value pairs where the key is an attribute and it's linked to a value. Two examples of key-value databases are Dynamo and Redis.

Redundancy & Replication

Redundancy

Redundancy is the process of duplicating pieces of your system or functionality of the system in order to improve its reliability. If we only store data on one server, there is a possibility of losing that data if the server fails. Having a redundant system prevents a single point of failure.

Replication

Replication is the sharing of information to ensure a redundant system is consistent.

Tips For Systems Design Interviews

In my experience, the systems design interviews for front-end developers seem to be less data-focused (how many terabytes of storage will we need?) and more architecture-focused (what is a good data-partitioning method given the needs of our application?).

Below are some tips for having a successful systems design interview.

- Ask clarifying question
- List functional and non-functional requirements:
    - If designing Twitter, for example, a functional requirement would be that users should be able to follow other users and post new tweets. A non-functional requirement may be that the system must be highly available; our system might want to prioritize replication of data.
- If you're unsure where to start, ask!
- If you're unclear about something say "I am not sure how to proceed with this, I know it's not correct, but I will come back to this if I have time at the end and will move on to X."

Example Systems Design Interview Questions

Here are a few types of questions you might encounter in a systems design interview. These questions are extremely broad and can be overwhelming. I recommend starting with the functional and non-functional requirements and approaching the interview as more of a conversation than a quiz.

- How would you design Instagram?
- How would you design Twitter?
- How would you design a real-time chat application?

CHAPTER 08

Tips for Hiring Managers and Interviewers

Fifty percent of the interview process is experienced by hiring managers and interviewers so it's important that we discuss some tips for the corporate side of the process as well.

Have Diverse Interviewers

If your interviewers fit one demographic, you unequivocally need to expand your interviewer pool to include people from all ethnicities and backgrounds. Not making diversity and inclusion a priority in your hiring process will lead to unconscious bias, hiring of candidates who fit the current mold at the company, and hinder innovation.

Throughout my many technical and non-technical interviews I have been interviewed by only one woman. One. Out of hundreds of interviewers.

This lack of interviewer diversity is unacceptable, and I can't begin to imagine how much more frustrating this lack of diversity is to underrepresented minorities or intersectional candidates.

Have women in the room. Technical women. And not just white women; women of color and intersectional women too. Include LGBTQ+ interviewers.

Diversify the team members conducting your interviews and you'll be amazed how many amazing developers you'll bring on to your team.

Have Multiple Interviewers

One thing that set Spotify apart from other companies was having two interviewers conduct each interview.

Initially I was put off by this idea; I'd have twice as many people to impress. In reality, having two interviewers had two primary benefits I didn't anticipate.

First it felt more like a conversation and less like an exam. Having multiple people to discuss a question provided more viewpoints.

Second, having multiple interviewers reduces the likelihood of unconscious bias. Each interviewer submits their feedback in isolation which hopefully prevents a candidate from being discriminated against if one interviewer was having a bad day or hasn't recognized their bias.

Pay Candidates For Coding Projects (If Possible)

If you're asking candidates to complete hours worth of coding projects, please pay them for it (if possible). I interviewed for a company where there was a take home project with three pieces.

Each piece related to a different aspect of the job I would be tasked with each day and they would pay me for my time.

Many candidates aren't interviewing for fun; many times people are interviewing due to unforeseen circumstances (i.e. they were laid off) and do not have the luxury of completing a six hour take home coding project. Or perhaps they have families and simply don't have the time.

Do Not Make Candidates Solve Your Problems

You should never, under any circumstances, ask candidates to solve a problem your team is currently facing. This is unethical, unpaid labor. Do not do it.

Ask Questions Which Mirror Problems That Would Arise On The Job

It’s no secret that technical interviews often ask questions meant to “see how you think” which is problematic for many reasons.

Many candidates haven’t completed a degree in Computer Science and as a result haven’t had exposure to data structures and algorithms.

Many are coming from non-technical backgrounds, bootcamps, or are self-taught and as a result are at a disadvantage to computer science candidates when asked deeply computer-science heavy interview questions.

Instead give your candidates questions which would arise on their day-to-day job at the company. They’re more likely to succeed and you’re more likely to see if they’d be a good fit for the role.

Provide Question Options

I once had an interview which gave me three questions I could choose to answer. Do you know how rare and how exceptionally brilliant that idea is?

Many people, myself included, struggle with specific areas of development. I personally struggle with asynchronous JavaScript and every single time I'm asked to "define a promise" I freeze.

Giving candidates two or three alternative questions they can answer or coding projects they can tackle will give everyone a fair shot at receiving a job offer; candidates can focus on the skills they're confident in and not get tripped up over the fact that they forgot how to code merge sort from scratch.

Don't Play Games

If you like a candidate, don't play games. Tell them how great they did in the interviews and make them an offer.

Spotify continually gave me wonderful feedback and got back to me within one or two days. It was so refreshing to find a company who made their candidates a priority and made them feel desired instead of stringing them along with a "we'll get back to you within two weeks...maybe?"

If you want a candidate, make them an offer. No games.

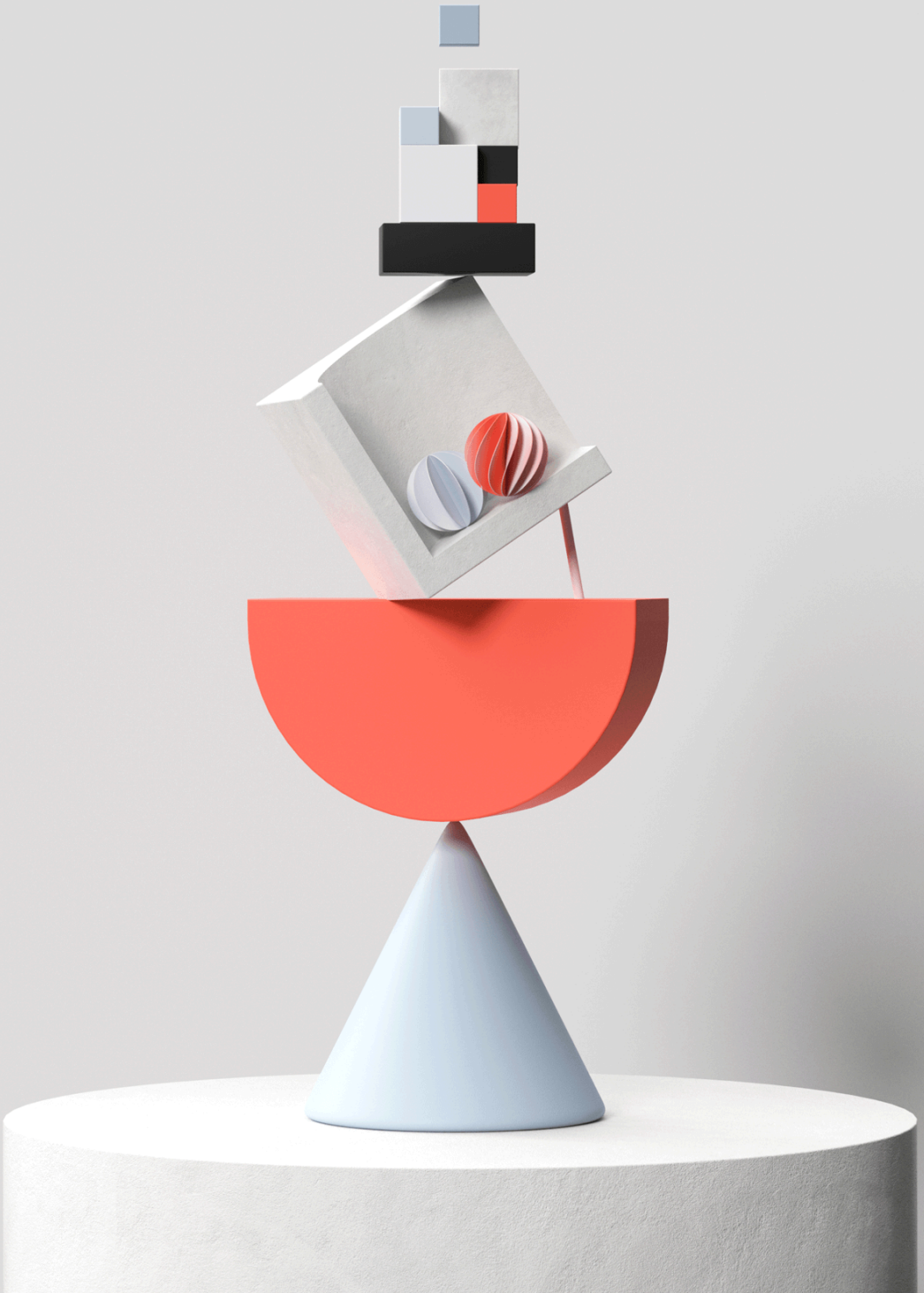
Don't Lowball Your Candidates

If you've gotten to the point where you're going to make a candidate an offer, do not lowball them. You're not trying to get a deal on a used bicycle. You're hiring a human being for their invaluable talent. A human being who has financial goals and potentially a family to support. Pay them what they're worth (if not more). Do not come in with a low offer to see if they'll negotiate.

Many candidates don’t negotiate (especially women) leaving them at a vast disadvantage when you lowball them in your offer. You’re not trying to get a deal. You’re hiring a talented human being and they deserve to be compensated for their skills.

CHAPTER 09

Resources



If you've gotten this far, congratulations! You're likely much better prepared than I was for my technical interviews!

Here are some of my favorite resources for learning web development, data structures and algorithms, and more.

Data Structures & Algorithms

Books & Blogs

- Cracking The Coding Interview by Gayle Laakmann McDowell

Courses

- Interviewing For Front-End Engineers by Jem Young on Frontend Masters
- Tree And Graph Data Structures by Bianca Gandolfo on Frontend Masters
- Introduction To Data Structures For Interviews by Bianca Gandolfo on Frontend Masters
- A Practical Guide To Algorithms With JavaScript by Bianca Gandolfo on Frontend Masters
- Four Semesters Of Computer Science In Five Hours by Brian Holt on Frontend Masters
- Four Semesters Of Computer Science In Five Hours, Part 2 by Brian Holt on Frontend Masters
- Data Structures & Algorithms In JavaScript by Bianca Gandolfo on Frontend Masters
- Data Structures & Algorithms In JavaScript by Kyle Shevlin on Egghead.io
- Algorithms In JavaScript by Tyler Clark on Egghead.io

Coding Practice

- Leetcode
- Hackerrank
- Project Euler
- Code Wars
- Top Coder
- Coderbyte
- Exercism

Web Development

Courses

- Complete Intro To Web Development V2 by Brian Holt on Frontend Masters
- Web Security by Mike North on Frontend Masters
- Free Code Camp

Accessibility

Books & Blogs

- Inclusive Components by Heydon Pickering on Smashing Magazine

Courses

- Accessibility In JavaScript Applications by Marcy Sutton on Frontend Masters
- Website Accessibility by Jon Kuperman on Frontend Masters

CSS

Courses

- Advanced CSS Layouts by Jen Kramer on Frontend Masters
- CSS In-Depth V2 by Estelle Weyl on Frontend Masters

Coding Practice

- Grid Garden
- Flexbox Froggy
- Flexbox Defense
- SS Diner

JavaScript

Books & Blogs

- Eloquent JavaScript by Marijn Haverbeke
- JavaScript Allongé by Reg Braithwaite
- You Don't Know JavaScript by Kyle Simpson
- Professional JavaScript For Web Developers by Nicholas C. Zakas

Courses

- JavaScript The Hard Parts by Will Sentance on Frontend Masters
- Getting Started With JavaScript, V2 by Kyle Simpson on Frontend Masters
- JavaScript: The Recent Parts by Kyle Simpson on Frontend Masters
- Deep JavaScript Foundations, V3 by Kyle Simpson on Frontend Masters
- JavaScript: The Hard Parts Of Object Oriented JavaScript by Will Sentance on Frontend Masters
- JavaScript Performance by Steve Kinney on Frontend Masters
- Advanced Asynchronous JavaScript by Jafar Husain on Frontend Masters
- Debugging & Fixing Common JavaScript Errors by Todd Gardner on Frontend Masters
- Advanced JavaScript Foundations by Tyler Clark on Egghead.io
- JavaScript Promises In-Depth by Marius Schulz
- Asynchronous JavaScript With Async/Await by Marius Schulz on Egghead.io

UX & Visual Design

Courses

- Information Architecture - on Udemy
- Design For Developers - by Sarah Drasner on Frontend Masters

Books & Blogs

- Ten Heuristics For User Interface Design

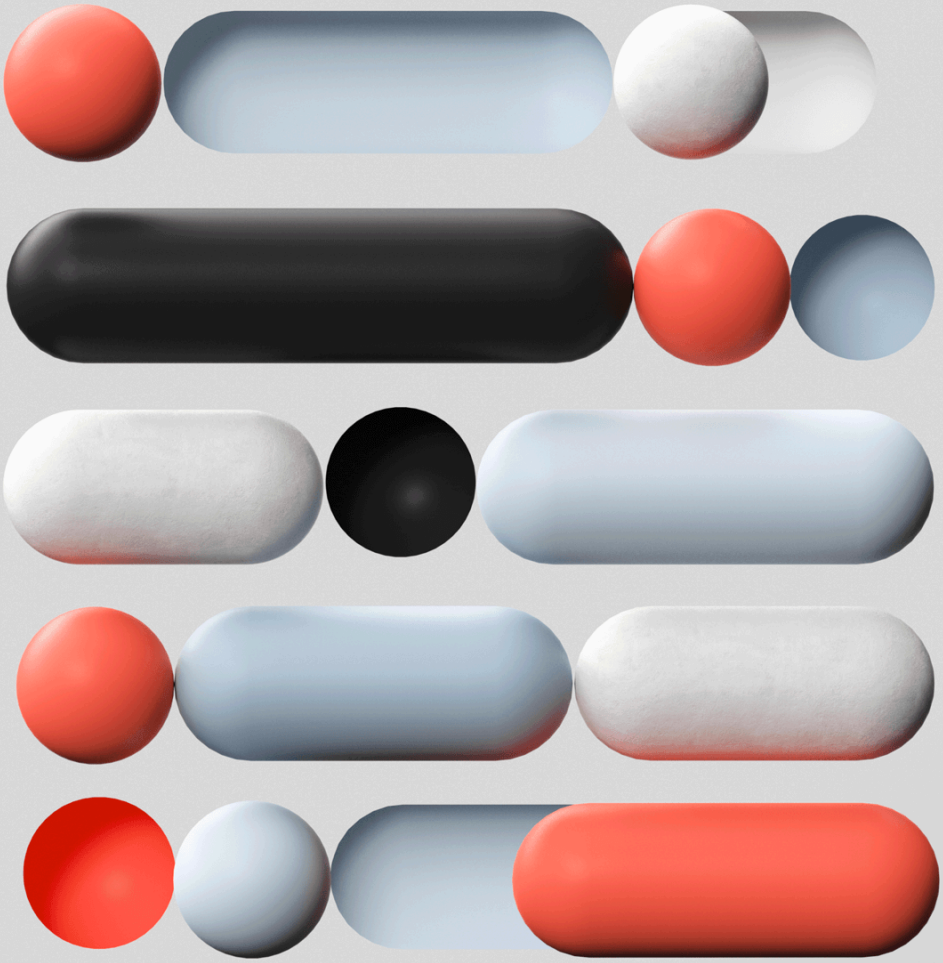
Job Searching

Websites

- Glassdoor
- LinkedIn
- XING
- Monster
- Indeed
- Hired

CHAPTER 10

Exercise Solutions



Find The Mammals Solution 1

```
/*
Brute-force method with a for-loop
0(n) where n = length of animals
*/
function findMammalsForLoop(animals) {
  const mammals = []
  for (let i = 0; i < animals.length; i++) {
    if (animals[i].mammal === true) {
      mammals.push(animals[i])
    }
  }
  return mammals
}
```

Find The Mammals Solution 2

```
/*
Using array.filter() method to test whether each
animal is a mammal
0(n) where n = length of animals but more readable
*/
function findMammalsArrayFilter(animals) {
  return animals.filter((animal) => animal.mammal === true)
}
```

Find The Mammals Solution 2 (Modified)

```
/*
Using array.filter() method to test whether
each animal is a mammal but de-structure mammal out of animal
0(n) where n = length of animals but more readable
*/
function findMammalsArrayDestructureFilter(animals) {
  return animals.filter(({mammal}) => mammal === true)
}
```

Display Sidebar - HTML

```
<aside class="sidebar">
  <nav>
    <button id="close-menu" aria-label="Close menu">+</button>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Work</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>
</aside>
```

Display Sidebar - CSS

```css
.sidebar {
  position: absolute;
  background-color: #000;
  height: 100%;
  left: -100%;
  transition: left 0.2s ease-in;
}
.sidebar--visible {
  left: 0;
}
.sidebar ul {
  list-style: none;
  margin: 0;
  padding: 72px;
  position: relative;
}
.sidebar li {
  margin: 12px 0;
}
.sidebar a {
  color: #000;
  text-decoration: none;
  font-size: 1.4rem;
  font-family: 'Dark mono', monospace;
  color: #fff;
}
```

Display Sidebar - CSS (continued)

```
.sidebar a:hover,
.sidebar a:focus {
  text-decoration: underline;
}
#open-menu {
  border: 2px solid #000;
  background: none;
  font-size: 1.4rem;
  font-family: 'Dank mono', monospace;
  padding: 6px 24px;
  cursor: pointer;
  position: fixed;
  right: 24px;
  top: 24px;
}
#close-menu {
  background: none;
  border: none;
  transform: rotate(45deg);
  font-size: 72px;
  font-family: 'Dank mono', monospace;
  color: #fff;
  cursor: pointer;
  position: absolute;
  right: 6px;
  top: -12px;
  z-index: 2;
}
```

Display Sidebar - JavaScript

```
const sidebar = document.querySelector('.sidebar')
const openMenu = document.querySelector('#open-menu')
const closeMenu = document.querySelector('#close-menu')
openMenu.addEventListener('click', toggleSidebarVisibility)
closeMenu.addEvenetListener('click', toggleSidebarVisibility)
function toggleSidebarVisibility() {
  sidebar.classList.toggle('sidebar--visible')
}
```

Find The Mammals (Version 2)

```
const animals = Array.from(document.querySelectorAlLl('.mammal-value'))
const mammals = animals.filter((animal) => animal.innerHTML === 'true')
const mammalsNode = document.querySelector('#only-mammals')
mammals.forEach((mammal) => {
  let grandparent = mammal.parentElement.parentElement
  mammalsNode.appendChild(grandparent)
})
```

The Social Network - HTML

```
<div id="friends"></div>
```

The Social Network - CSS

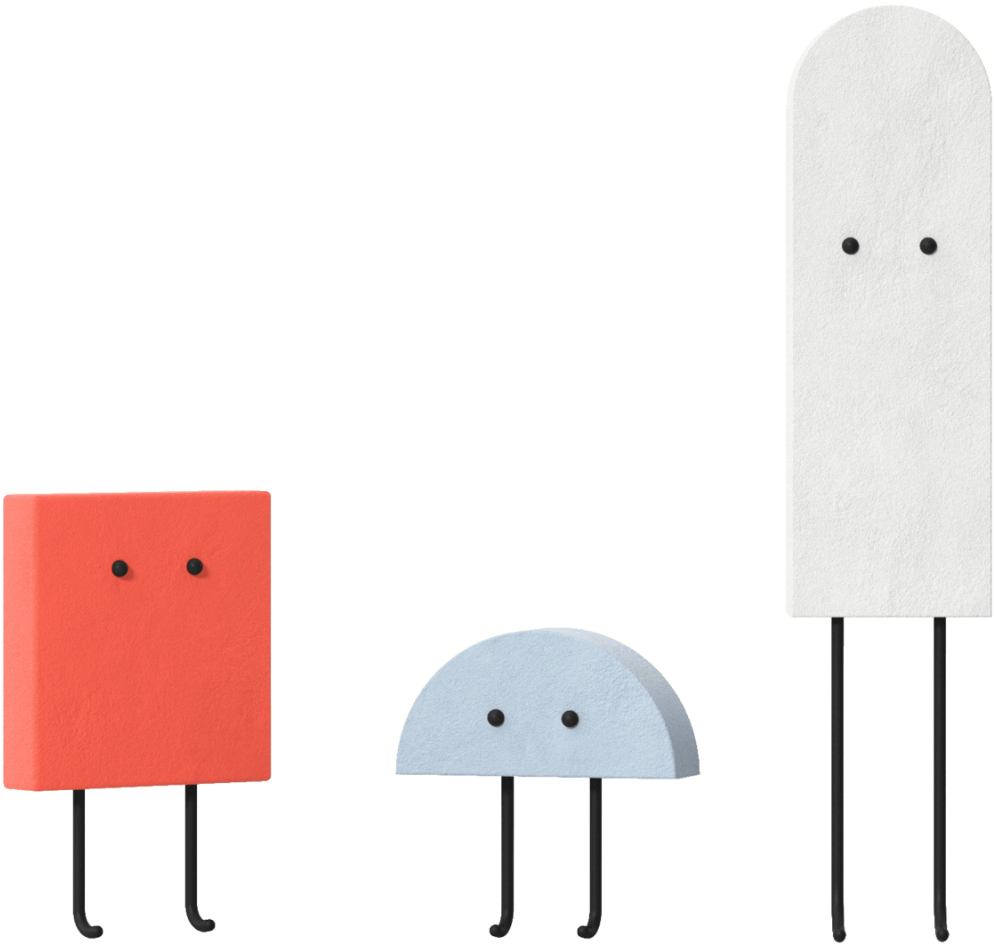
```
.person {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  border: 2px solid #000;
  padding: 24px;
  width: 300px;
  margin: 0 auto 80px;
}
.person > img {
  border-radius: 50%;
}
```

The Social Network - JavaScript

```
async function getFriends() {
  const response = await fetch('https://randomuser.me/api/?results=10')
  const friends = await response.json()
  const friendNodeWrapper = document.querySelector('#friends')
  friends.results.forEach((friend) => {
    let name = `${friend.name.first} ${friend.name.last}`
    let nameParagraph = document.createElement('p')
    nameParagraph.textContent = name
    let picture = friend.picture.large
    let pictureTag = document.createElement('img')
    pictureTag.src = picture
    pictureTag.alt = name
    let personWrapper = document.createElement('div')
    personWrapper.classList.add('person')
    personWrapper.appendChild(nameParagraph)
    personWrapper.appendChild(pictureTag)
    friendNodeWrapper.appendChild(personWrapper)
  })
}
getFriends()
```

CHAPTER 11

Thank You



Thank you for taking the time to read my book. Writing a book has been a life-long dream of mine, since I was a young girl. I remember scribbling down nonsense stories thinking I wanted to be an author someday. And even though this is a self-published book, my dream has come true.

I never wanted to get rich off this book; I simply wanted a way to give back to the community during a time when we needed it most.

Yet through your generosity you've helped me pay off my medical debt that I had hanging over my head for past five years.

You are smart. You deserve to be in the tech industry. And someday you will receive an offer from your dream job. And it will feel damn good.

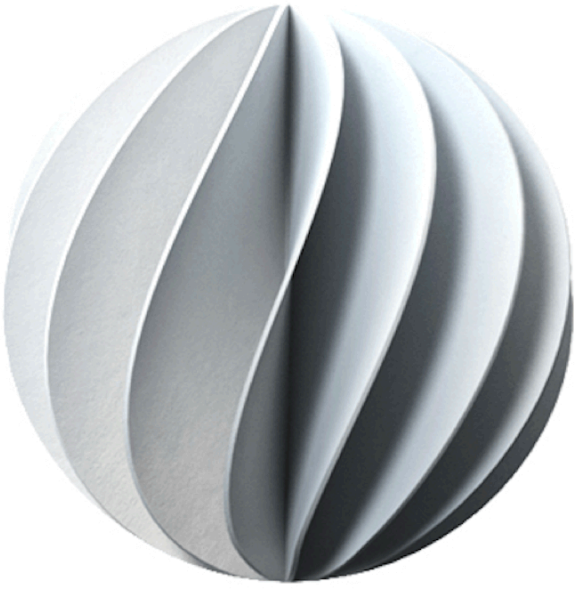
Thank you, from the bottom of my heart.

With love,

Emma

CHAPTER 12

Printable Assets



Study Guides

I've compiled three study guides for you to use if you're preparing for an interview. These guides relate to the topics listed in chapter four (data structures), chapter five (algorithms), chapter six (front-end interviews) and chapter seven (systems design interviews).

These study guides are how I would study for an interview if I had one, two, or four weeks to study. You may not need to study certain topics (like systems design or UX/visual design) so feel free to amend the plans as you see fit.

Additionally you may already be knowledgeable in the foundations of JavaScript, or another topic, so you may want to prioritize algorithms and data structures.

The main takeaway: these guides are how I personally would study but you may need a different plan, and that's okay! Do what works for you. Don't forget to take breaks and rest days. Your brain needs time to process all of the information it's digesting. Get enough sleep, drink enough water, and trust yourself.

You can do this.

One-Week Learning Plan

DAY 1	DAY 2	DAY 3	DAY 4	DAY 5	DAY 6	DAY 7
DATA STRUCTURES	ALGORITHMS	DATA STRUCTURES & ALGORITHMS REVIEW	HTML CSS ACCESSIBILITY FOUNDATIONAL JAVASCRIPT	INTERMEDIATE & ADVANCED JAVASCRIPT	JAVASCRIPT REVIEW	SYSTEMS DESIGN* UX & VISUAL DESIGN* PERFORMANCE SECURITY TESTING
CODING PRACTICE	CODING PRACTICE	CODING PRACTICE	CODING PRACTICE	CODING PRACTICE	CODING PRACTICE	CODING PRACTICE

* OPTIONAL DEPENDING UPON YOUR INTERVIEW SUBJECT MATTER

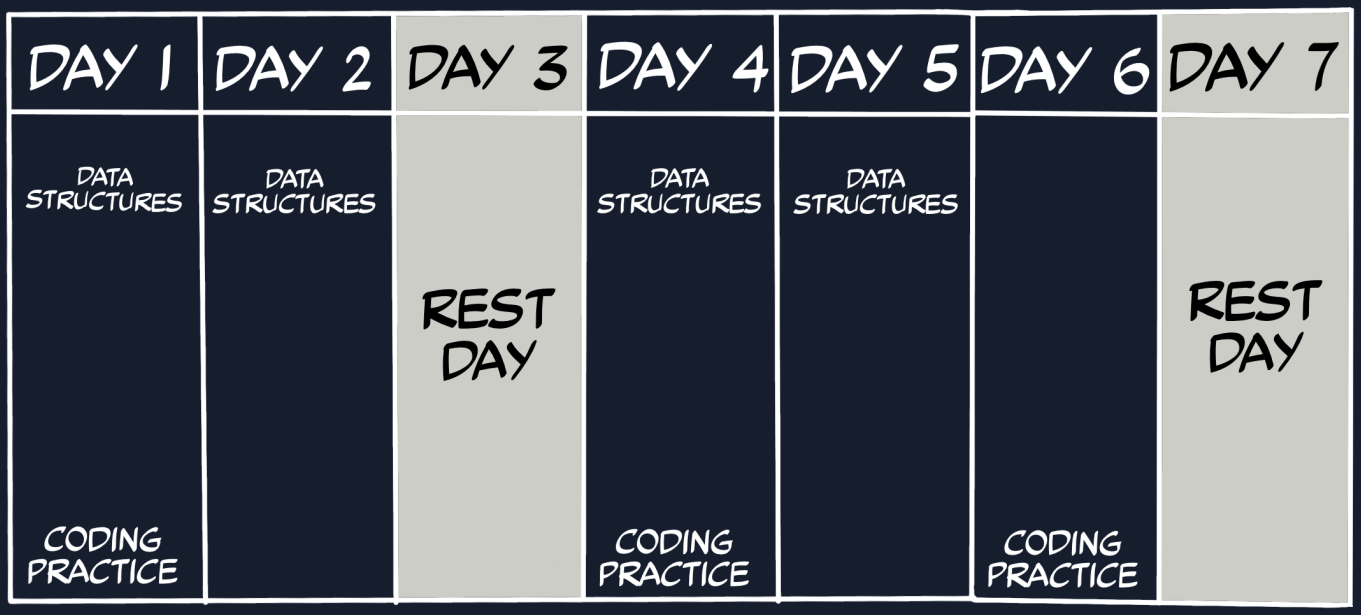
Two-Week Learning Plan

DAY 1	DAY 2	DAY 3	DAY 4	DAY 5	DAY 6	DAY 7
DATA STRUCTURES CODING PRACTICE	DATA STRUCTURES CODING PRACTICE	REST DAY	ALGORITHMS CODING PRACTICE	ALGORITHMS CODING PRACTICE	DATA STRUCTURES & ALGORITHMS REVIEW CODING PRACTICE	REST DAY

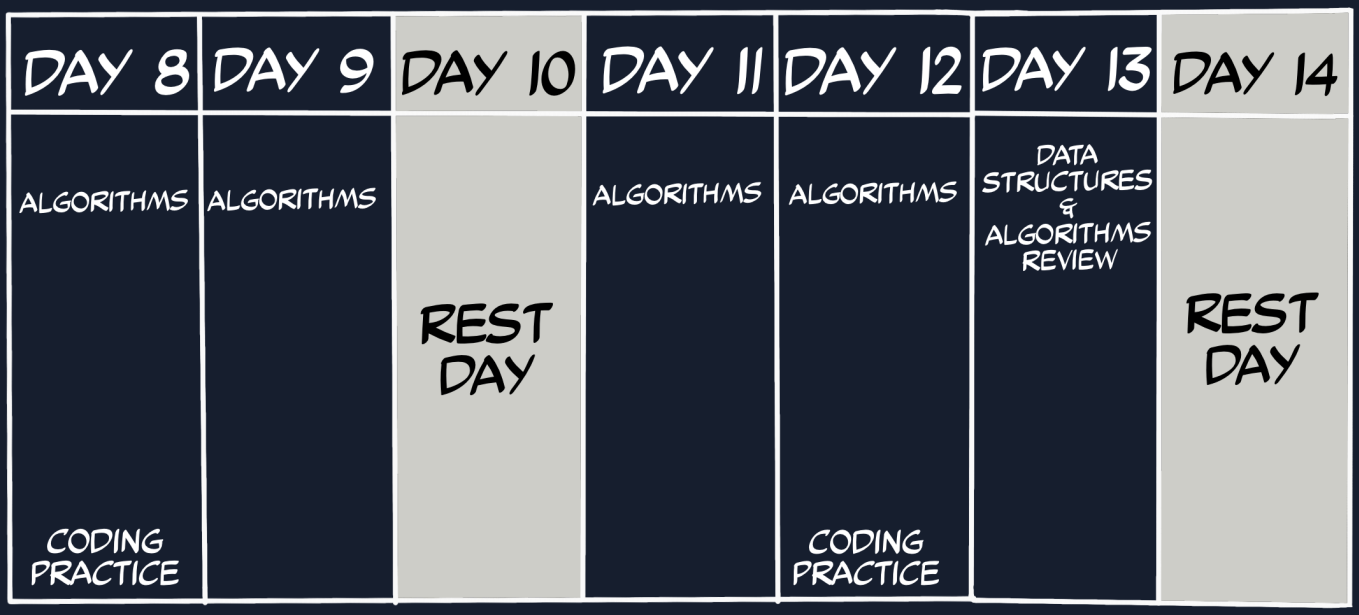
* OPTIONAL DEPENDING UPON YOUR INTERVIEW SUBJECT MATTER

DAY 8	DAY 9	DAY 10	DAY 11	DAY 12	DAY 13	DAY 14
HTML CSS ACCESSIBILITY CODING PRACTICE	FOUNDATIONAL JAVASCRIPT CODING PRACTICE	REST DAY	INTERMEDIATE & ADVANCED JAVASCRIPT CODING PRACTICE	INTERMEDIATE & ADVANCED JAVASCRIPT CODING PRACTICE	PERFORMANCE SECURITY TESTING CODING PRACTICE	SYSTEMS DESIGN* UX & VISUAL DESIGN*

Four-Week Learning Plan



* OPTIONAL DEPENDING UPON YOUR INTERVIEW SUBJECT MATTER



Four-Week Learning Plan

DAY 15	DAY 16	DAY 17	DAY 18	DAY 19	DAY 20	DAY 21
HTML FOUNDATIONAL CSS CODING PRACTICE	ADVANCED CSS	REST DAY	ACCESSIBILITY CODING PRACTICE	FOUNDATIONAL JAVASCRIPT	FOUNDATIONAL JAVASCRIPT	REST DAY

DAY 22	DAY 23	DAY 24	DAY 25	DAY 26	DAY 27	DAY 28
INTERMEDIATE & ADVANCED JAVASCRIPT CODING PRACTICE	INTERMEDIATE & ADVANCED JAVASCRIPT	REST DAY	JAVASCRIPT REVIEW	PERFORMANCE SECURITY TESTING CODING PRACTICE	SYSTEMS DESIGN* UX & VISUAL DESIGN CODING PRACTICE	REST DAY

Coding Project Checklist

Before submitting your coding project, take a quick run through this checklist!

It'll help ensure you present the best version of your work.

- [ ] Clarify requirements with the recruiter.
- [ ] List the functional requirements.
  - What features or functionality does your project have to have?
- [ ] Design a user flow or information architecture.
- [ ] Pick your tech stack.
- [ ] Sketch out project architecture.
  - Where will your components live? Your styles?
- [ ] Check the readability of function names.
- [ ] Document your app
  - Setup instructions
  - Project architecture
  - Included assets
  - Finding the live site
  - Areas of improvement
- [ ] Refactor non-performant code.
- [ ] Remove code comments.
- [ ] Check app accessibility.
- [ ] Add unit tests.*
- [ ] Check browser console for errors.
- [ ] Double check requirements.
- [ ] Submit and relax!