Data Structures

Andrew Rosen

Contents

Part I

Preliminaries

Chapter 1

Introduction

1.1 What is a Data Structures Course

Data Structures is all about defining the different ways we can organize data. This is not databases, which is concerned with defining the various attributes of a bunch of data; this is much more granular. We want to know how to store and retrieve a single item of data.

1.2 Why This Book?

This textbook is free.

It is both Java and Python, which is a bit insane. You have two valid choices:

- Understand that the concepts we are learning are way more important than the language and treat the other language as psuedocode (which isn't hard for Python)
- Be comfortable in multiple languages and embrace being a polyglot. Impress your friends, wow your rivals!

1.2.1 Where Does This Book Fit Into a Computer Science Curriculum

Education in Computer Science is based around three core topics: translating the steps of solving a problem into a language a computer can understand, organizing data for solving problems, and techniques that can be used to solve problems. These courses typically covered in a university's introductory course, data structures course, and algorithms course respectively, although different universities decide exactly what content fits in which course. Of course, there is are lot more concepts in computer science, from operating systems and low level programming, to networks and how computers talk to each other. However, all these concepts rely on the knowledge gained in the core courses of programming, data structures, and algorithms.

This textbook is all about Data Structures, the middle section between learning how to program and the more advanced problem solving concepts we learn

in Computer Science. Here, we focus on mastering the different ways to organize data, recognize the internal and performative differences between each structure, and learn to recognize the best (if there is one) for a given situation.

1.2.2 What Are My Base Assumptions about the Reader?

This textbook assumes that the student has taken a programming course that has covered the basics. Namely: data types such as ints, doubles, booleans, and strings; if statements, for and while loops; and object orient programming. This book is also suitable for the self taught programmer who has not learned much theoretical programming

1.3 To The Instructor

1.3.1 Professor Rosen's Extremely Opinionated Advice on How to Lecture

You'll note that this textbook lacks some of the features found in commercially available textbooks. The biggest of these is slides. For the most part, slides are too static to help students understand how to code.

I'll go a step further to be blunt: from intro to programming all the way thru data structures, slides are absolute trash; use them if and only if you have no time to prepare. In fact, even if you have no time to prepare, I would caution against using it.

I have been teaching Data Structures since Fall 2011. In order of preference, this is how I would tackle *this* class, which I fully recognize may not work for you or your teaching style.

Lecture with slides. Do this if slides are available (they are not for this book) and it is your first time teaching this class.

Lecture via live coding. Basically, your lecture unit for a data structure should look like this[1]:

- Introduce the ADT that you will be modeling. So for a [array]list, describe what it is, why we want it over an array, and the operations.
- Code a functional, pedagogical implementation live in class.
  - Functional means a student could ostensibly use it in an assignment. This means most of the work will focus on `add` and `remove` or their equivalents.
  - Pedagogical means that you should keep straightforward and not try to reproduce the entire built-in class. If you're working in Java, your implemented `MyArrayList` shouldn't try to implement the `List` interface. You should only focus on the primary ways a programmer interacts with the class in question. It also means you should emphasize that the built-in, real-world classes will have a number of optimizations that speed things up , but what you're covering is a close enough approximation.

[1] Conviently, the textbook is written for you to model this

- This might be a bit unnerving to have to reproduce a class in front of the class, but watching someone program these things from scratch works better than just reading snippets of text.
- Mistakes will be made, but students need to see mistakes are normal.

Do the above, but flip the lecture. You can see my example of this here:

1.3.2 Exercises

Does the lack of varied exercises make cheating on assignments easier as semesters go on? Yes, but that bridge was burned long ago. The cheating student can plagiarize from various websites or anonymously hire another to do their work for them. However, the student who cheats isn't exactly clever and certainly hasn't been exposed to much game theory. They will often cheat from the same source.

In addition, during the writing of this text, technologies such as GPTChat were released. This hasn't so much burned the bridge as dropped napalm on the entire surrounding forest. Newer technologies will then salt that earth. I recommend an open and honest dialogue with your students and at least 50% of their grade being the result of evaluations and assessments you do in class. This can range from proctored exams to flipping the classroom and giving students the chance to work on homework in class, where they are much more likely to turn to you or their peers for help.

My personal solution for assignments is to use require students to demo their homework to me or a TA to receive a grade. As part of this demo, they must answer questions about their solutions. Now, as you well know, a student being unable to answer question about their work or make on the fly adjustments **isn't necessarily** an indicator that a student has cheated. Personally, I find half of my students seem to lose approximate 50 IQ points upon being directly questioned. It is daunting to be the sole subject of attention to the person who is making the determination as to whether you pass or fail. But I digress. To summarize, being unable to answer questions about their code might be an indication of cheating, it might be an indication of nerves. In most cases, you can figure out if it's the latter case, in which case, just send them away until they can explain their code. Don't penalize them, but remind them that programming interviews require this kind of presentation of skills.

As far as student who use AI, AI generated code has a number of tells and those tells will change over time. However, the usual marker is using either too well documented code and data structures or syntax well above what you would expect from a student. Now, if the student is using Vim or Emacs or rocking Linux, they probably can explain exactly what they are doing. In fact, save yourself the trouble and just assign them a minimum of a B. However, most of the time a student won't be able to explain the thought process behind the solution or the way some syntax works. If pressed they will explain a vague "friend" helped them with the code. You can press further if you want and handle it however. Personally, it depends on where we are in the semester. In the first few weeks, I emphasize that they will be completely wrecked by the exams if they rely on this "friend" and they need to do the work themselves and tell them to come back when that is the case. At the end of the semester, I am much less inclined to have mercy.

1.3.3 The Order

1.3.4 Assignments

I will drop the sporadic assignment here or there, drawing from the same places you should draw from:

- Nifty Assignments
- Problem Solving with Algorithms and Data Structures using Java (Miller)

1.3.5 How to Use

1.4 To The Student

Why are we learning this? As Brad Miller and David Ranum put in their aforementioned book (which is creative commons and you should totally check out):

> To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. These models allow us to describe the data that our algorithms will manipulate in a much more consistent way with respect to the problem itself.
>
> Earlier, we referred to procedural abstraction as a process that hides the details of a particular function to allow the user or client to view it at a very high level. We now turn our attention to a similar idea, that of data abstraction. An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user' s view. This is called information hiding.
>
> Figure 2 shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.
>
> Figure 2: Abstract Data Type
>
> The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will allow us to define the complex data models for our problems without giving any indication as to the details of how

the model will actually be built. This provides an implementation-independent view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

1.4.1 How to use

1.5 License

This work is funded by Temple University's North Broad Press and is under Creative Commons - Attribution Non Commericial License

1.6 On Styles

On styles: Java convention is to use camel case for variable types (`myVariableName`), while python convention is to use underscores (`my_variable_name`). I will be using the Java style camel-casing for variables throughout the book for consistency and because it is my preference.

Chapter 2

Functions and How They Work

This will be an extremely short chapter, but an important one. We are already going to assume that you know what a function, a method, or procedure is and that you have written them before. After all, Data Structures is a point continuing your education in programming, not beginning it. That said it is possible that you missed some subtleties along the way.

That's understandable - programming is a very large topic and there's more than enough concepts that no one who graduates with a degree in computer science can be expected to be an expert in every area any more.

With this in mind, let me take the time to review some subtleties surrounding the vocabulary of functions.

2.1 Function vs Method

You'll often hear programmers use these two terms interchangeably to refer to what essentially amounts to a subprogram. But what is the difference? I like to explain it this way: functions are the verbs of the programming language. When we create a new function, we a creating a new verb in the programming language we are working in. Methods are a special type of function that are closely linked to objects; they are the actions or verbs you want your objects to perform.

Java blurs this a bit with `static` methods, but for the purposes of this text, when I write *method*, I am talking about Java's *instance methods*. *Function*, in the context of this book, is analogous to *static methods*. Similarly, if you are coming from Python, when I say function, I am talking about a boring top-level, unindented function such as the ones you've been writing since you first learned Python. Method would refer to the functions you create as part of your classes.

2.2 Argument vs Parameter

An argument is the actual value you pass in, the parameter is the variable that accepts it.

Listing (Java) 2.1: Java Parameter vs Argument Example

```java
public static void doubleThis(int num) {   // parameter
    return 2*num;
}

public static void main(String[] args){
    int x = 7;
    int y = doubleThis(x);  // argument
}
```

Listing (Python) 2.2: Python Parameter vs Argument Example

```python
def doubleThis(num): # parameter
    return 2*num

x = 7
y = doubleThis(x) # argument
```

In the above examples, `x` is an argument and `num` is a parameter.

2.2.1 Does anyone actually care?

I cared enough to look it up, but I also had to look it up to double check that I'm correct and I keep coming back to this page as a reference for myself. In a casual situation or talking with another programmer, everyone will be able to *grok* your meaning from the context, as you just did with the word `grok`. I would take care to get it correct for your assignments and exams, much like you would take care to avoid using "ain't" in a formal essay. One professor might be a stickler about it and one might not care.

2.3 Passing Arguments

The vast majority of programming languages are *pass by copy* with a huge honking asterisk.

- Pass by copy means that when something is input as the argument to a function (or method), the function gets a copy of the thing you are passing to it.
- The *huge honking asterisk* is that you are almost always passing a *reference* or *pointer* to an object, not the object itself. The reason for this is that if we had a super mega huge object, copying it would take up a super mega huge amount of time and memory.

2.3.1 How it Works in Java

In Java, we have two broad categories of data types: primitives and objects.

When you pass a primitive, such as an `int` or `double`, the value gets copied from where it is stored in memory and copied into the argument.

When you create an object, such as with `Scanner scan = new Scanner(System.in);`, the variable `scan` will hold not the Object that was created by the constructor, but the *memory location*, or *reference* of where to find it. Look at the code below where we are we create an array in `main` and then pass it to another method, `setIndexZeroToZero`:

Listing (Java) 2.3: Code to change index 0 to the value 0

```java
public static void setIndexZeroToZero(int[] array) {
    array[0] = 0;
}

public static void main(String[] args) {
    int[] arr = {5,5,6,6};
    System.out.println(Arrays.toString(arr));
    // prints [5, 5, 6, 6]

    setIndexZeroToZero(arr);
    System.out.println(Arrays.toString(arr));
    // prints [0, 5, 6, 6]
}
```

Because the memory location of the array `arr` was passed to `array`, the method `setIndexZeroToZero` was dealing with the same object.

Keep in mind that some objects are immutable, like any `String`. This means you can't actually change them. Operations that seem like they change them like replacing part of a string or converting things from upper case to lower case are all returning a newly generated string.

2.3.2 How it works in Python

Practically speaking , everything in Python works the same as in Java. Everything in Python is an object (including the integers, which are immutable.), so when things are passed or assigned into variables, the variable stores the memory location, or *reference*, to the object. Thus when you pass in a variable to a function, the function receives the memory location of the object; data is never duplicated.

Part II

Lists

Part III

Recursion

Chapter 3

Recursion

3.1 Introduction

3.1.1 Why?

Much in the same way we use Object Oriented Programming as a tool to organize our thoughts about how to design large programs, programmers can use recursion to craft elegant and efficient solutions. Once you get a hang of recursion, it's a really easy way create solutions. I often refer to it as a way to be lazy at programming, with my recursive problem solving typically going like this:

- I am at some amorphous spot in the puzzle or problem I am solving.
- This problem is too big to solve in one go.
- Let's just write code that solves only this specific part of the problem.
- Now that I have the solution to this portion, since I'm lazy, I'll just call a magic method that solves the rest of the problem starting at the point immediately after what I just solves.
- It turns out the magic method is what I just wrote.

Confused? That's fine. It often takes a few attempts to get a handle on recursion. It should start to make sense with some examples.

3.2 Recursive Mathematics

We'll start our discussion with some mathematical examples that you might already be familiar with.

3.2.1 Factorial

The factorial function is hopefully something you have seen before. The function, if not the name, has been know for thousands of years. Here it is in Sefer Yetzerah (4:12)[7] [3], the oldest book of Jewish Mysticism.

שבע כפולות כיצד צרפן. שתי אבנים בונות שני בתים. שלש בונות ששה בתים. ארבע בונות ארבעה ועשרים בתים. חמש בונות מאה ועשרים בתים. שש בונות שבע מאות ועשרים בתים. שבע בונות חמשת אלפים וארבעים בתים. מכאן ואילך צא וחשוב מה שאין הפה יכול לדבר ואין האוזן יכולה לשמוע.

Seven doubles - how are they combined? Two "stones" produce two houses; three form six; four form twenty-four; five form one hundred and twenty; six form seven hundred and twenty; seven form five thousand and forty; and beyond this their numbers increase so that the mouth can hardly utter them, nor the ear hear the number of them.

Mathematically, we use the ! symbol for factorial and define:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n$$

In other words, $n!$ is the product of all the numbers from 1 to n. Thus,

$$1! = 1$$
$$2! = 2$$
$$3! = 6$$
$$4! = 24$$
$$5! = 120$$
$$6! = 720$$
$$7! = 5040$$

0! defined as 1, as we are multiplying no numbers together and the multiplicative identity is 1. Less formally, if you do a running sum, you start at zero, but for a running product, you start with 1, since if you started your running product with zero, you'd get zero.

We can write an iterative implementation of this fairly easily.

Listing (Java) 3.1: Factorial - Iterative

```
public static long factorialIter(int n) {
    long total = 1;
    for(int i = 1; i <= n; i++) {
        total =  total * i;
    }
    return total;
}
```

Notice that I use `long` in Listing 3.1. The total gets very very big, very very fast. Or as Sefer Yetzerah put it: "their numbers increase so that the mouth can hardly utter them, nor the ear hear the number of them."

Now, let's play around with the equation a bit. It's fairly trivial to see in the calculations above that we can get the next value factorial value by multplying by the next integer, e.g. we can go from 2! to 3! by muliplying 2! by 3.

$$1! = 1 \cdot 0! = 1$$
$$2! = 2 \cdot 1! = 2$$
$$3! = 3 \cdot 2! = 6$$
$$4! = 4 \cdot 3! = 24$$
$$5! = 5 \cdot 4! = 120$$
$$6! = 6 \cdot 5! = 720$$
$$7! = 7 \cdot 6! = 5040$$

Going the other direction, we can say that some $n!$ can be figured out by calculating $(n-1)!$ and multiplying by n.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \ldots (n-1) \cdot n \\ &= n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1 \\ &= n \cdot (n-1)! \end{aligned} \tag{3.1}$$

We call this function, where a function is calculated by solving the same function on a (usually) smaller value, a **recursive** function. Let's implement it and take a look.

Listing (Java) 3.2: Factorial - Recursive

```java
public static long factorial(int n) {
    if(n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

Listing (Python) 3.3: Factorial - Recursive

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

This probably makes some sense because you were just looking at the math equation, but this might also seem like magic or weird or, worst of all, weird magic. In fact it's quite possible that you've accidentally created something resembling an infinite loop before by having a function or method call itself. So why does it work here?

A recursive function requires two parts in order to work: a base case and a recursive case. The base case is the foundation of our recursive problem. It is where we have a defined solution for some value. In the factorial, this is the line that checks if $n == 0$ in our code, or just defining $0! = 1$ in the mathematics. I look at the base case as the point where we can answer the question reflexively and without much thought.

The recursive case is where we solve our problem by solving a simpler subproblem. In our code, we So in our code, we look at solving `factorial(n)`, decide that's way too much work and decide to solve `factorial(n-1)` and multiply that by `n`. Solving `factorial(n-1)` presents us with the same challenge, so we call `factorial(n-2)` to multiply that against `(n-1)`. Solving `factorial(n-2)` presents us with the same challenge, so we call `factorial(n-3)` to multiply that against `(n-2)`…

This continues until we call `factorial(1)`, which calls `factorial(0)`, the base case, which finally gives us 1.

`facorial(1)` takes that 1 and returns `1 * 1`. Then `factorial(2)` takes the answer from `factorial(1)` and returns `2 * factorial(1)` Then `factorial(3)` takes the answer from `factorial(2)` and returns `3 * factorial(1)` And so on and so forth until `factorial(n)` takes the answer from `factorial(n-1)` and returns `n * factorial(n-1)`

We know this works because for any given non-negative integer[1] n each recursive call on `factorial` is on a smaller and smaller number, making progress to calculating `factorial(0)`. Once we hit `factorial(0)`, the answers start being calculated and trickling up this stack of function calls.

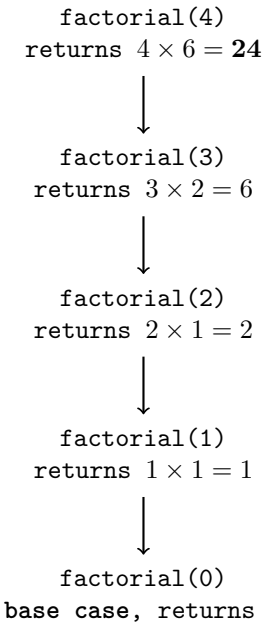


Figure 3.1: The call stack for `factorial(4)`. Each call must wait for the result of the call below it. Once `factorial(0)` returns 1, the results are multiplied back up the stack.

[1]Negative factorials are undefined and I'm ignoring that case in our code. My suggested solution is to either error or document turning something like $(-5)!$ into $-1 \cdot 5!$. It's wrong and will gravely upset the Math department, but might be the desired behavior for your program. But even more important, you should document what you do in weird cases like this!

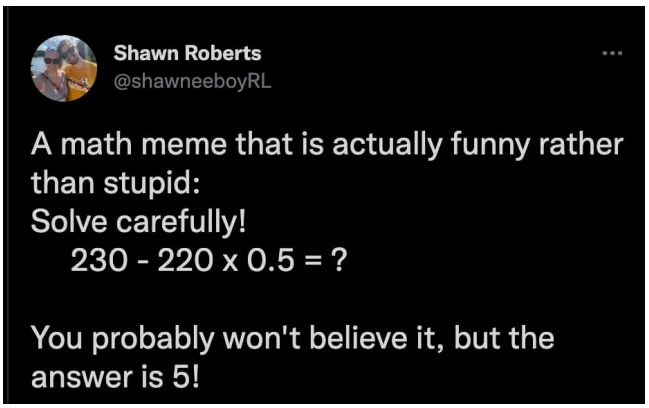


Figure 3.2: Hopefully you get it now.

3.2.2 Recursive Rules

As previously mentioned, all recursive functions:

- Must have one or more base cases where the solution is well defined.
- Must have one or more recursive cases, where the problem is defined by a smaller subproblem of the same type as the original.
- Must ensure the recursive cases make progress towards the defined base case.

You prove a recursive algorithm will solve the problem in question by showing all the above points are true. This is much the same as a proof by induction, just in the opposite direction.

Failure to follow the rules.

If your recursive case fails to make progress towards your base case, then you end up with a special type of infinite loop which is not actually infinite. Every time you make a method call, your computer needs to store where in the code it was and what conditions that were present. The specifics of how and why this is done are outside the scope of the textbook [2], but suffice to say, this information gets stored in a part of the computer memory designated as *the stack*. This stack is named such because it is a `Stack` just like what you have seen in Chapter **??**. Since this stack is living in your memory and your computer probably does not have infinite memory we can run the following program to see what happens when that stack "fills up."

Listing (Java) 3.4: Recursion with no end condition - Java

```
public static void bad(){
    bad();
}
```

[2] maybe

Listing (Python) 3.5: Recursion with no end condition - Python

```
def bad():
    bad()
```

You'll get something along the line of a `Stack Overflow` error or exception, which indicates that your stack in memory has gotten completely used up. This rarely happens in correctly created recursive programs.

3.2.3 Fibonacci

The Fibonacci sequence is the classic introduction to recursive formulas and recursion in programming. I opted for teaching the factorial sequence first due to the complications with runtime a naive implementation has. This might lead to the impression that *all* recursive functions have a terrible runtime. They do not.

History

The Fibonnaci sequence is named after Leonardo Bonacci, also known as Leonardo of Pisa, and also known as Fibonacci. He authored a book in 1202 called *Liber Abaci*, which introduced the western world to calculations using Hindu-Arabic numerals . It also enumerated the Fibonacci sequence, which is why it is named after him[1, 4]. Notice I said *western* world. It is hard to appreciate that humans could not share information in the same way we can today, as well as what can be lost due to damage or merely not writing it down. The Fibonacci sequence had been observed previously by Indian mathematicians such as Gopāla [5] Furthermore, it is completely possible someone had observed this sequence earlier, wrote it down in a text somewhere, and then the text being lost to fire or rot. Backups are important and can mean the difference between having something named after you or not.

Definition

The Fibonacci sequence (sequence A000045)[3] is defined as the sequence of numbers where each number in the sequence is the sum of the previous two numbers. The sequence starts with 0 and 1 and looks something like this:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987...$$

The sequence continues indefinitely. More formally, let F_n be the nth number of the Fibonacci sequence. We define the sequence with:

$$F_0 = 1, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Regardless, the Fibonacci sequence important. It shows up again and again in nature, in science, and in mathematics. The number of petals a flower has tends to be plucked from the Fibonacci sequence [6].

[3] Yes, humans are such nerds that we've created an online library for sequences - OEIS.

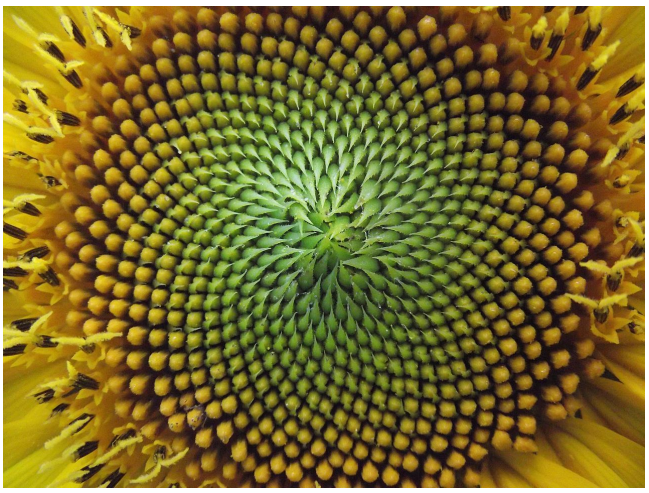


Figure 3.3: The sunflower's fibonacci spiral. Photo by Anna Benczur, CC by-SA 4.0.

Implementation

Implementing this as a recursive function is rather trivial!

Listing (Java) 3.6: Naive Java Implementation

```java
public static long fib(int n){
    if(n == 0 || n == 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Listing (Python) 3.7: Naive Python implementation

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

A Flaw appears in the plan

As it turns out, while this technically works...it's pretty terrible. In short, using recursion, I managed to accidentally[4] write an $O(2^n)$, or exponential time, algorithm. This is very bad. This means increasing n by one *doubles* the runtime of out algorithm! Go ahead and try it for yourself on your computer. You should start seeing some massive slowdowns when computing `fib(n)` somewhere around `n=45`. Notice that each time you increase `n` by one, the amount of time your computer spends working roughly doubles.

[4] All right, I did this totally on purpose.

This is because to solving the current `n` requires solving `fib(n-1)` and `fib(n-2)`. Furthermore, each recursive call is independent from each other; solving `fib(n-1)`

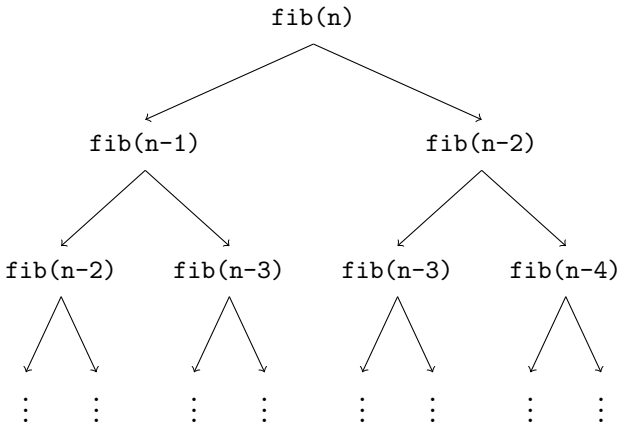


Figure 3.4: Recursive Function Calls for `fib(n)`. Notice that the call to `fib(n-1)` must independently compute `fib(n-2)`, thus duplicating a ton of work.

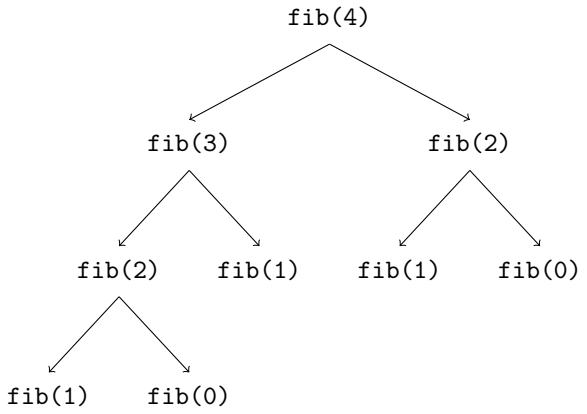


Figure 3.5: Computing `fib(4)`.

Don't let this terrible runtime scare you away from recursion! Recursion can make things quite efficient; this is merely an exception and presented here because Fibonacci is such a classic example we would be remiss to not include it.

Solutions

There's a lot of solutions to make this work. My personal favorite is **memoization**, which simply says "well if the issue is having to redo the work, let's instead store the results of each function call."

Listing (Java) 3.8: An Efficient Recursive Fibonacci Algorithm

```java
public static long fib(int n) {
    long[] lookup = new long[n];
    lookup[1] = 1;
    return fib(n , lookup);
}


private static long fib(int n, long[] F) {
    if(n <= 1) { //base case
        return F[n];
    }
    if(F[n-1] == 0) {
        F[n-1] = fib(n-1, F);
    }
    if(F[n-2] == 0) {
        F[n-2] = fib(n-2, F);
    }
    return F[n-1] + F[n-2];
}
```

So here we have a public method that the programmer will use to calculate the `n`th Fibonacci number and a private helper method to do the actual work. The array `F` is an array where we store any previously calculated Fibonacci numbers. The big change from our original solution is now we ask if the `n-1` Fibonacci number has been calculated before. If it has not, calculate it and store it in the array. We do the same for the `n-2` Fibonacci number. The reference to the array is shared between all recursive calls. After the check and the possible calculation is done, the function uses those numbers to calculate `fib(n)`.

Listing (Python) 3.9: An Efficient Recursive Fibonacci Algorithm in Python

```
def fib(n, F = []):
    if len(F) == 0:
        F = [0] * n
    if(n <= 1):
        return n
    if(F[n - 1] == 0):
        F[n - 1] = fib(n - 1,F)
    if(F[n - 2] == 0):
        F[n - 2] = fib(n - 2,F)
    return F[n - 1] + F[n - 2]
```

So here we have a function with a default variable `F` that is initially an empty list. If the program detects `F`'s empty, it is initialized to a list of zeroes. We do this to avoid writing a second function, like we did in the Java example. The list `F` is a list where we store any previously calculated Fibonacci numbers. The big change from our original solution is now we ask if the `n-1` Fibonacci number has been calculated before. If it has not, calculate it and store it in `F`. We do the same for the `n-2` Fibonacci number. The reference to `F` is shared between all recursive calls. After the check and the possible calculation is done, the function uses those numbers to calculate `fib(n)`.

3.3 More Examples

Some of the upcoming examples of the things we are about to see should not be actually used and serve only as examples, like our `printThis` function.

3.3.1 Printing Recursively

Listing (Java) 3.10: Recursive Printing: Java

```
public static void printThis(String s){
    if (s.length() == 0) {
        System.out.println();
    } else {
        System.out.print(s.charAt(0));
        printThis(s.substring(1));
    }
}
```

Listing (Python) 3.11: Recursive Printing: Python

```python
def printThis(s):
    if len(s) == 0:
        print()
    else:
        print(s[0], end='')
        printThis(s[1:])
```

3.4 Arrays with Recursion

3.4.1 Summation of an Array

The ay to think of this is in terms of a base case and a recursive case. The base case is size of one or zero. Either way, the answer is trivially easy to figure out. The recursive case is basically a way of saying adding a bunch of numbers is too hard. I'll just return adding the number at the first index of this section or subsection of the array to whatever the total the rest of the array is. I'll use a magic function to figure it out. it turns out the magic function is actually this one.

3.4.2 Recursive Linear Search

By this point, you know how to iteratively search a list for a specific item. We start at the first item/index and go thru the array one item at a time until we get to the last item or find the item we want. Let's take a look at the same algorithm, just implemented recursively.

Listing (Java) 3.12: Recursive Linear Search - Java

```java
// We return the index we found the item at
// -1 means item is not in the list
public static <E> int search(List<E> list, E target){
    return search(list, target, 0);
}

private static <E> int search(List<E> list, E target, int
↪   index) {
    if(index >= list.size()){
        return -1;
    }
    if(list.get(index).equals(target)){
        return index;
    }
    return search(list,target, index+1);
}
```

Listing (Python) 3.13: Recursive Linear Search - Python

```
def search(theList, target):
    return search(theList, target, 0)

def search(theList, target, index):
    if index >= len(theList):
        return False
    if theList[item] == target:
        return True
    return search(theList, target, index + 1)
```

Again, this is more of a case of pedagogical examples, rather than practical ones. We want to get some practice in before we get to the really interesting recursive problems.

Runtime

The above code has the exact same runtime as doing it iteratively – $O(n)$ in the case of an `ArrayList`. Remember, we don't want to use this for a `LinkedList` due to the $O(n)$ cost the `get` method incurs, which would yield an overall $O(n^2)$ runtime. Use the built-in iterator instead, i.e. use a for each loop.

3.4.3 Binary Search

Binary search is our reason for including Recursion at this location in the textbook. It will be an essential step in building Binary Search Trees.

Objective

Like our recursive linear search, our goal is to search for a particular item in an array or list. Once we find that item, we can either return true or the index we found it at, depending on our implementation. If we fail to find, we return either false or -1 or `null`; again this depends on our implementation.[5].

Assumptions

We will be using an array for Java and `List` for Python. This data structure will be sorted. This is a key assumption; if the array is not sorted, we cannot do a binary search.

Solution

Since our core assumption is that we are using a sorted collection, it makes sense that our algorithm exploits this. Think of a game that you might have played in school as a kid, the "I'm thinking Of a number from one to 100. I'll tell you if it's higher or lower." Now the linear strategy that we went over previously would be the equivalent of asking "Is it one? Oh, it's higher? Is it 2? It's higher? Is it 3? Oh, it's higher?" and so on and so forth Until we hit the number in

[5]Or force a win using *Gifts Ungiven*. Wait, wrong fail to find.

question. A more reasonable strategy would be to pick the number 50 because that number is in the middle of the entire range. Once we know whether the number is higher or lower we have effectively halved our range. This is because if the number is higher than 50 we know that the number cannot be between 1 and 50, inclusive. If it is lower than 50 we know the number cannot be between 50 and 100, inclusive. And if the number is 50 we just simply got lucky. The next step is to choose the number in the middle of our new range so we can do the same halving of our search space.

Let's take this strategy and apply it to an array of sorted numbers, seen in Figure 3.6.

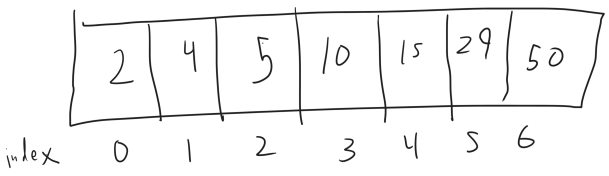


Figure 3.6:

In this example, we want to see query if this array contains the item 5. We start by asking figuring out what the middle index of the array is, since half the items in the array will be to the left and half to the right[6]. The array is 7 items total, so we start at index 3 and compare 5 to the value stored in there (Figure 3.7).

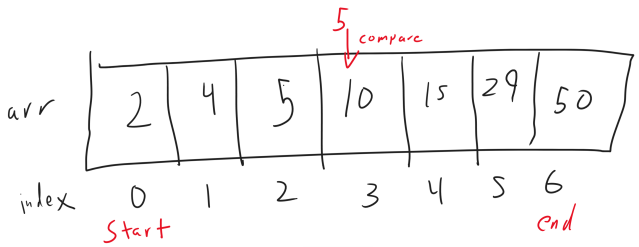


Figure 3.7: The labels `start` and `end` represent the start and end of our search space. This will make more sense as we progress, especially as we start coding.

Since $5 < 10$, we know that if 5 is in the array, it will be found to the left of index 3. We put the end of our search space one index to the left of the middle of our previous search space. Our new range to search is now index 0 thru index 2 (Figure 3.8). We compare 5 to the item in the middle of that range, which is the number 4 at index 1.

$5 > 4$, so if 5 is in the array, it is on the right side of our search space. Our search space contracts to a single item, index 2 (Figure 3.9).

The item at index 2 is the same item we've been looking for, so we have successfully found our item.

[6]Or half stored in lower indices and half stored in higher indices if you prefer.

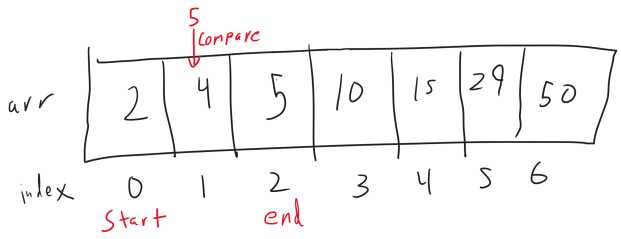


Figure 3.8: The labels `start` and `end` represent the start and end of our search space, which has now shrunk to less than half the original array.

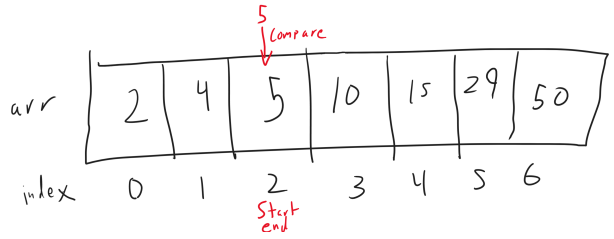


Figure 3.9: The labels `start` and `end` are now on the same item. This means a search space has a size of 1.

Code

Listing (Java) 3.14: Binary Search - Java

```
public static int binarySearch(int[] arr, int target) {
    return binarySearch(arr, target, 0, arr.length-1);
}

private static int binarySearch(int[] arr, int target, int
↪   start, int end) {
    if(start > end) {
        return -1;
    }
    int mid = (start + end) / 2;
    if(target == arr[mid] ) {
        return mid; // item found
    } else if( target < arr[mid]) {
        // search right side
        return binarySearch(arr, target, mid+1, end);
    } else {
        // search left side
        return binarySearch(arr, target, start, mid-1);
    }
}
```

Our outer, wrapper function exists to have a clean function to call. Our helper function does the actual work. If we fail to find our target, we will return -1, which is an invalid index.

Listing (Python) 3.15: Binary Search in Python

```
def binarySearch(arr, target, start = 0, end = 0):
    if len(arr) > 0 and end == 0:
        end = len(arr) - 1
    if start > end:
        return None
    mid = (start + end) // 2
    if arr[mid] == target:
        return mid
    elif target < arr[mid]:
        return binarySearch(arr, target, start, mid - 1)
    else:
        return binarySearch(arr, target, mid + 1, end)
```

Here, we have two base arguments for our initial call, with start and end to be used for a recursive call. The first if statement is to set `end` correctly for our topmost (initial) call, because I'm too lazy to write a second, private helper function. The first base case returns `None` to represent a failure to find.

In our above code, we create two base cases, which are quite simple if you think about it. The first is if our search space is size 0 or invalid. We obviously can't find `target` in the search space if the search spaces doesn't exist. This is the `if(start > end)` clause, since if the start of the search space is to the right of the end of the search space, we don't have a valid space to search anymore. In this scenario, we return a failure state of some sort[7].

The next step is to calculate `mid` which is the index in the middle of the search space. From there, we get our second base case: if the item at index `mid` is the item we are looking for, we're done. Otherwise, we check if `target < arr[mid]`. If this is true, then target must be on the left half of the array. To search that left half for `target`, we call `binarySearch(arr, target, start, mid - 1)`. Why are each of the arguments what they are?

- The first parameter of `binarySearch` is the array or list we are search, so the argument that we pass into our recursive call remains the same.
- The second parameter is the `target` item, which remains constant.
- The third parameter is the beginning index of our search space. When we search to the left side of the search space, we are searching between `start` and `mid`, not including `mid`. Thus the third argument won't change.
- We pass in `mid - 1` as the `end` of the new search space, since that is the right side of the new, smaller search.

Now, if `target < arr[mid]` is false, we search the right side, which means that the fourth argument is `end`, but we change the third argument to `mid + 1`.

[7] -1 in our Java example and `None` in our python example.

Runtime Analysis

Each call of `binarySearch` eliminates either exactly or almost exactly half of the search space if we don't find our `target`. This halving can be mathematically described by the operation $\log_2(n)$, where n is the number of items.[8] Thus, with an array of 256 items, this algorithm would take approximately 8 steps. Doubling the size of the array to 512 items increases the amount of work only by a single step. This yields $O(\log n)$ as the runtime.[9]

Compare that to our linear search, which starts at the beginning and goes through the array one item at a time. That takes $O(n)$ time. In this case, doubling the number of items means doubling the amount of work the algorithm has to do.

It bears emphasizing and repeating: $O(\log n)$ runtime is a major improvement over a linear runtime. Doubling the size of n does practically nothing to change the runtime of `binarySearch`, but would make a linear search take twice as long.

How to not be scared of logarithms

You may have learned that logarithms are the inverse operation to exponentiation. This is an utterly useless definition when programming.

A much more useful way of thinking about logarithms is "how many times can I recursively split something?" For example, $\log_b x$ asks "how many times can I recursively split my x items into b seperate piles?"

A more concrete example: $\log_2 16 = 4$, not because $2^4 = 16$, but because a pile of 16 items can be split in half into two piles of 8, each pile of 8 can be split in half into two piles of 4, the 4's can be split into 2's, the 2's into 1's — four splits total:

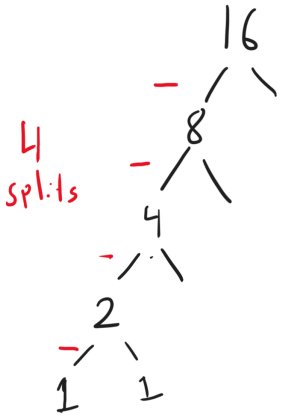


Figure 3.10:

In algorithm analysis, $\log n$ in the time complexity is used to indicate that the search space gets split in half. In the Binary Search algorithm above, we

[8] If the sudden appearance of logarithms risks scaring you off, just keep reading to the next subsubsection. I wrote that special for you.

[9] We drop the base of a logarithm when we use big O notation.

split the our search space in half each step of the way. We start out looking at the middle item and then decide to look at all the items below or all the items above. This reduces the number of items to search among from n to $\frac{n}{2}$. From there we perform the same choices and reduce that $\frac{n}{2}$ to $\frac{n}{4}$, then from $\frac{n}{4}$ to $\frac{n}{8}$ and so on.

Additional Implementation: Java with Lists

Listing (Java) 3.16: Binary Search - Java Lists

```
public static <E extends Comparable<E>> int
↪   binarySearch(List<E> list, E target) {
    return binarySearch(list, target, 0, list.size()-1);
}

public static <E extends Comparable<E>> int
↪   binarySearch(List<E> list, E target, int start, int
↪   end) {
    if(start > end) {
        return -1;
    }
    int mid = (start + end) / 2;
    if(target.compareTo(list.get(mid)) == 0) {
        return mid; // item found
    }
    if(target.compareTo(list.get(mid)) < 0) {
        // search right side
        return binarySearch(list, target, mid+1, end);
    } else {
        // search left side
        return binarySearch(list, target, start, mid-1);
    }
}
```

Performing binary search on a list looks something like this. Recall that `Comparable` is an interface that Java uses to let methods and classes know something can be put in order[a]. This necessarily means that they can be sorted. The generic `<E extends Comparable<E>>` means the the `List` of `E`'s is guaranteed to be made up of items that can be compared to other things of type `E` to see which comes first.

[a]Formally, this is a *total ordering* in fancy math lingo, which means any two items have an established order

3.5 Recursive Backtracking

Recursion really comes in handy when we are trying to solve complex puzzles. One of the most famous examples of this is using recursion to solve the eight

queens problem or Sudoku. Before we get into this, let's establish the rest of the chapter. I'll introduce the eight queen's puzzle. Then, I'll show you the generic recursive backtracking algorithm, explain it, and then show a partial solution to the eight queens puzzle. I'll also go over the Sudoku solver, but will leave that as a potential homework.

The entire reason for this aside is to speak to both the students and instructors who use this book. The eight queens problem is a recursive problem that has been used as a homework problem longer than I've been alive. This means there are a million and a half solutions floating around the internet.

Students: I ask you do not rob yourself of that learning experience and instead strive to follow the text I have here. Go to your teacher or TA or classmate if you get stuck for more than two hours.

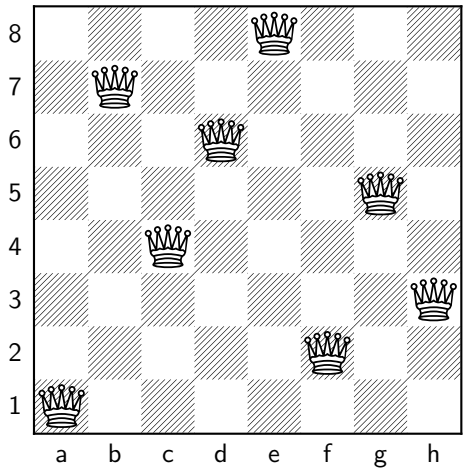
Teachers: Give these exercises, or something similar like the knight's tour, as a homework assignment. These are engaging and worth it. Yes, some students are going to completely ignore the plea of the last paragraph, but they were going to do that no matter what assignment you gave. Don't rob the engaged students of a great learning experience.

3.5.1 The Eight Queens Puzzle

The eight queens puzzle is an old chess puzzle. If you don't know how to play chess, you should change that, but knowing how to play is not necessary to solving the puzzle.

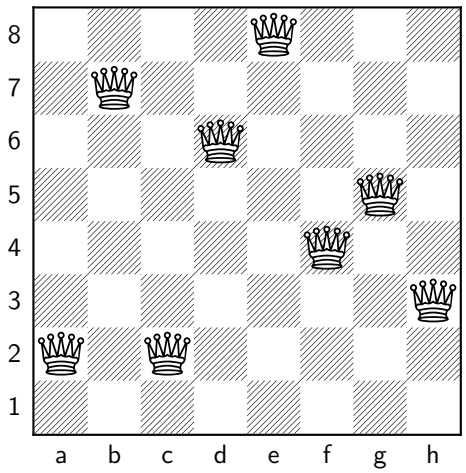
The goal of the puzzle is to place eight queens on a standard 8×8 chessboard. The queens should be placed in such a way that no queen can capture any other queen. In chess, a queen can capture any piece by traveling vertically, horizontally, or diagonally until it crashes into a piece. Or in more programming friendly terms, we need to place 8 queens such that none share a row, column, or diagonal.

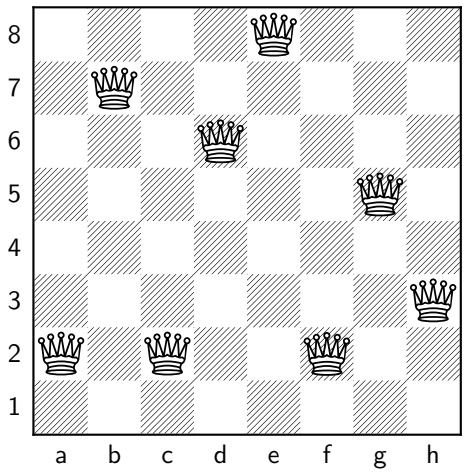
This is one example solution, but it is not the only solution.



Brute Force Solution

A brute force algorithm means we will be checking every single possible state to find a solution. In this case, a brute force solution for the Eight Queens Puzzle would every possible placement of eight queens on a chessboard, such as these two incorrect solutions:

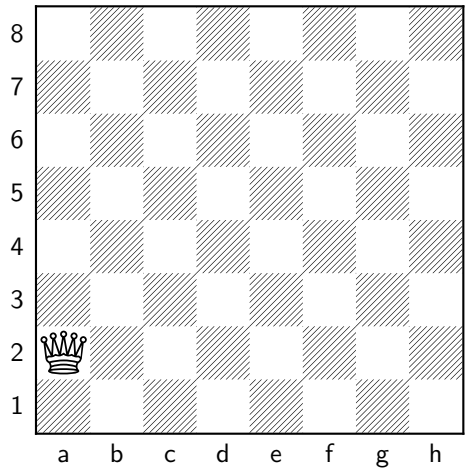


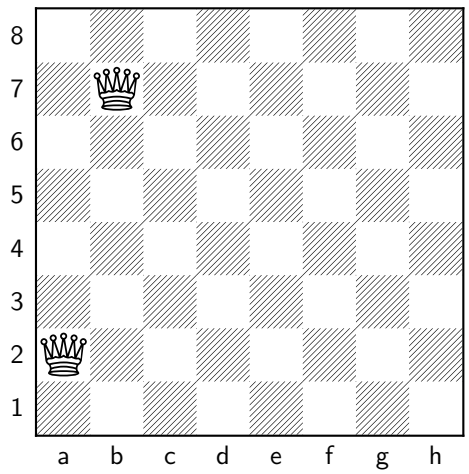


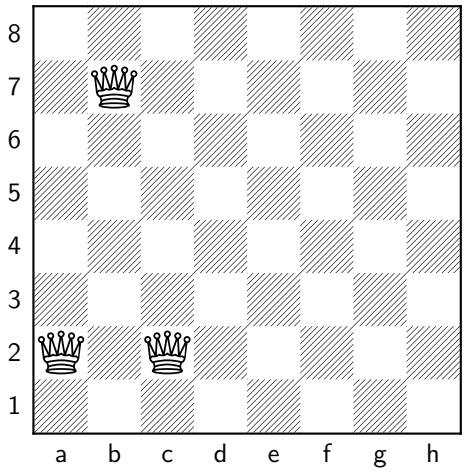
There are a total of $\binom{64}{8} = 4426165368$ possible ways to place 8 queens on a chessboard with 64 spaces. Our computer could sift thru all possible configurations until it finds a solution, and thus, performing a brute force solution will eventually work. It just won't work fast.

Our motive is to do better and apply some more logic to our searches. Take those two incorrect solutions, for example. The first has queens on spaces a2 and c2, which means that it is an incorrect solution. The second example also has queens on a2 and c2, but additionally it has a queen on f2. If we can go about finding a solution more methodically, there is no reason that we should ever check the second example. Once we establish that placing a queen on a2 and c2 doesn't work for a solution, we should never check any solution that

contains a2 and c2 ever again. If fact, let's try and go even further. Assume to generate a solution,we place queens right to left, going one column at a time, like so:







Once we hit this point, there is no need to place any more queens, since we know any of the remaining 5 queens we place down will be in a "dead" branch of the search space. This would allow us to "skip" checking the $\binom{40}{5} = 658008$ solutions that place the five other queens.

The Recursive Backtracking Algorithm

So how do kill a branch of our search space off?

We're going to generalize the puzzle for our next step. Let's introduce you to `solve`, our recursive backtracking algorithm.

```
boolean solve(board,  pos){
```

This solve method will take in two parameters: `board` and `pos`. Board is the puzzle we are trying to solve and all the work we've done on it so far. In eight queens, it will be our chess board and where we've paced the queens. For Sudoku, it would be the grid and all the numbers we have put down (more on that later). For a maze, such as in Chapter **??**, this would be the configuration of the maze, e.g. the walls and corridors and any marks we have made to avoid searching the same thing twice.

Now, `pos` is our *position* in the puzzle. Let's think of it as the current thing we are trying to solve. For the Eight Queens puzzle, our position the current column we're placing our queen on (more on that later). For Sudoku, we need to find what number goes in this current square. In the maze, we want to select the path leads to the end.

Finally, the `boolean` return value is used to signal whether solved the puzzle or failed.

So now that we are given the two parameters we need, we need to come up with a nice, concise algorithm that solves the entire puzzle. That's a bit of a tall order, but since we are in the Recursion chapter, we should probably think of a recursive solution[10]. That's good, because that will enable us to be *lazy*.

[10]Metagaming is a valuable skill in not just in games, but in school and life. Don't just study the material is for the exam, predict kind of questions your Professor likes to asked based on prior knowledge.

Our recursive solution requires the two elements all recursive solutions need - a base case and a recursive case. We'll also find we need a third case, but more on that in a bit.

```
boolean solve(board,  pos){
        // base case

        // recusive case
}
```

Let's start with the base case. The base case should be the simplest, laziest thing we can think of. For any puzzle, that would be the board already being solved. If our position indicates everything is done, we return true to signal that the maze is solve. This would be having no queens left to place, or no Sudoku squares to fill, or having found the exit.

```
boolean solve(board,  pos){
        // base case
        if(pos indicates puzzle is solved){
                return
        }
        // recusive case
}
```

Easy. Now onto the recursive case. The puzzle is not solved. So let's do the *next laziest* thing - solving our current position, the bare minimum. Now at our position, we have lots of possible choices.

We can think of it as which choice is needed to make to solve the *entire* puzzle from here. Where do we need to put the queen so it will be valid for all the other choices? What number is the correct number to put in this square? What turn in the maze will lead us to the exit? There is no way to know if we're not at the base case, but we're going to pretend that the choice we're making is the correct one for now.

What makes this better than brute forcing a solution is that not every possible choice we can make is valid. How do we figure that out which choice is valid? That depends on the puzzle, so we are going to be lazy and abstract checking if a choice is valid to a `valid` function.

This brings us to the first part of the recursive step. Look at each choice we can make, one at a time. As soon as we find a valid choice, we're going to pick it and assume it's correct.

```
boolean solve(board,  pos){
        // base case
        if(pos indicates puzzle is solved){
                return
        }
        // recusive case
        for each possible choice {
                if(valid(choice)){
                        select and mark that choice;
                        // part two goes here
```

```
                }
        }
}
```

This next part is the trickiest when we first look at it, but once we get the hang of it, writing `valid` will be the hardest part of doing this kind of algorithm.

Now that we've made our presumably correct choice, we are going to be lazy and call a magic `solve` function to solve the rest of the puzzle from the next position. That function will return true or false, depending on whether the magic `solve` function found a solution or not. If it returns true, we will return true to indicate to whatever function called us that the puzzle is solved with the found solution stored in *board*.

If `solve` returns false, that's going to indicate this choice will not lead to a solution, given the current state of the *board*. When that happens, we will need to undo whatever we did to mark our current choice and resume checking the other possible choices to find if one is valid.

```
boolean solve(board, pos) {
        // base case
        if(pos indicates puzzle is solved) {
                return
        }
        // recusive case
        for each possible choice {
                if(valid(choice)) {
                        select and mark that choice;
                        if(solve(board, nextPos) == true){  // recursive case
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid;
                }
        }
        // But what if there's no valid choice?
}
```

As it happens, the magic function to solve the rest of the puzzle is the one we are currently writing, so we're almost done.

Unlike our previous recursive problems which only had a base case and a recursive case, our solution involves a failure state of sorts. What if we look at all our choices and none of them work? If we hit this case, our program knows it won't be able to find a solution with the current configuration and uses `return false` to inform the function that called it that something in the current configuration needs to change.

This failure case is what makes this a recursive backtracking algorithm.

```
boolean solve(board, pos) {
        // base case
        if(pos indicates puzzle is solved) {
                return
        }
        // recusive case
```

```
        for each possible choice {
                if(valid(choice)) {
                        select and mark that choice;
                        if(solve(board, nextPos) == true){  // recursive case
                                return true;
                        }
                        unmark board at pos if needed, as choice was invalid;
                }
        }
        return false; // backtrack
}
```

As I previously mentioned, while this looks complicated at first, especially the `if(solve(board, nextPos) == true)` once we get used to it, the recursive part is fairly straightforward.

A **key feature** of this is that we always know where we are; we never have to search for the space we are trying to solve - it is the current position.

3.5.2 Recursively Solving the Eight Queens Problem

Now that we have generalized algorithm, let's apply it to the Eight Queens problem. We'll represent the chess board as a 2D array. There's many different valid type we could use, but I'm going to use an array of `String`, using `'Q'` and `'-'` to represent a space with a queen and an empty space respectively. Our position in the puzzle with be the `col` variable, representing the column we are currently working on. Finally, the choice for each column will be what `row` we place the queen on.

Listing (Java) 3.17: Outline of Solution - Java

```java
public static boolean solve(String[][] board, int col){
    if(col == 8) { // use board.length to generalize
        return true;
    }

    for(int row = 0; row < 8; row++) {
        if(valid(choice)){
            place "Q" at row,col
            if(solve(board, pos + 1) == true){
                return true;
            }
            replace "Q" with "-", as choice was invalid
        }
    }
    return false; // backtrack
}
```

The initial call to solve will pass in `0` for `col` to start at the first column. We use `col == 8` as the base case, as a call to `solve(board, 8)` would be trying to check a column that does not exist. The only way for that to happen is from a call on the 8th column (index 7), which means that that last column has found a working solution. If we want to generalize this solution to work on square boards other that a standard 8x8 chess board, we can do that by replacing all the 8's in the code with `board.length`.

Listing (Python) 3.18: Outline of Solution - Java

```python
def solve(board, col):
    if(col == 8): # use len(board) to generalize
        return True

    for row in range(8):
        if valid(choice):
            place "Q" at row,col
            if solve(board, pos + 1) == True:
                return True
            replace "Q" with "-", as choice was invalid
    return False # backtrack
```

The initial call to solve will pass in `0` for `col` to start at the first column. We use `col == 8` as the base case, as a call to `solve(board, 8)` would be trying to check a column that does not exist. The only way for that to happen is from a call on the 8th column (index 7), which means that that last column has found a working solution. If we want to generalize this solution to work on square boards other that a standard 8x8 chess board, we can do that by replacing all the 8's in the code with `len(board)`.

A Place Holder For Validity

Assume for a second that it is possible to not find a solution. Would our program continue forever? We ask this question to make sure the recursion is valid. Let's modify the `valid` function to do nothing but return `false` and examine the result. Whenever we fail to find a solution on a column, `solve` returns `false` to whatever function called it. Thus, if no solution is possible, we will eventually exhaust the search space and return `false` on the entire problem.

Performing the Recursion

Let's go ahead and fill in that code to demonstrate the recursion works. We can do this by setting `valid` function to do nothing but return `true`. Once we do that, there's basically only the most simple of changes to make. Test by calling `solve` by passing in an 8x8 array of `"-"` and `0`.

Listing (Java) 3.19: Solve - Java

```
public static boolean solve(String[][] board, int col){
    if(col == 8) { // use board.length to generalize
        return true;
    }

    for(int row = 0; row < 8; row++) {
        if(valid(board, row, col)){
            board[row][col] = "Q";
            if(solve(board, pos + 1) == true){
                return true;
            }
            board[row][col] = "-";
        }
    }
    return false; // backtrack
}
```

Listing (Python) 3.20: Solve - Python

```
def solve(board, col):
    if(col == 8): # use len(board) to generalize
        return True

    for row in range(8):
        if valid(board, row, col):
            board[row][col] = "Q"
            if solve(board, pos + 1) == True:
                return True
            board[row][col] = "-"
    return False # backtrack
```

The result should be eight queens in the first row and nowhere else, which makes sense; the `valid` function will return `true` no matter what, so the first square in the column is always valid.

Completing `valid` is left as an exercise for the user. Now you might be thinking since Queen moves in eight possible directions, you need to check for conflicts with placing a queen in eight directions too. Fortunately, you're wrong. Firstly, we are only ever placing a single Queen on a column. Since we are trying to figure out which row to place that Queen on for any particular column, we never have to check if the Queen has any pieces above or below it. Second, we are only ever moving from left to right. Whenever we place a Queen, the next column we choose is always to the right. If we don't find a valid queen, the column is cleared before returning `false`. This means we never have to check for Queens to the right of the space we want to place our Queen.

This leaves only three directions to check:

- Directly to the left.
- The upper left diagonal.
- The lower left diagonal.

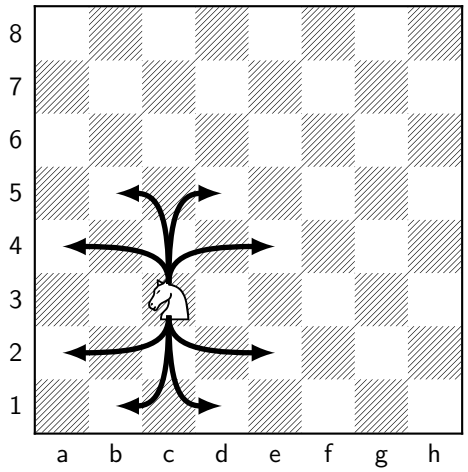
My advice is to work on the first case and test that. If it works, you'll get a line of Queens going from the top left corner to the bottom right.

3.5.3 Additional Problems left to the Reader

Knight's Tour

In the Knight's Tour, we place a Knight on the chess board and move him until he visits each square of the chess board exactly once.[11] A knight moves two squares horizontally or vertically and then one square in the axis it did not move it, creating a sort of "L" shaped (see below). A square counts as visited once the knight lands in it.

[11] Please do not sack Constantinople on your way to the answer.



You may start your knight anywhere you like. Your output should be either the chess board, but with each square marked by a number to designate the order in which the square was visited, or by listing the moves the knight makes. If you can figure out a better way to represent your answer, we are open to that too.

Sudoku

Sudoku (Japanese: 数独 [12] or ナンバープレイス) is a grid based number puzzle, as seen below.

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

The goal is to fill out the puzzle so there are no blank squares. Each square must take a single number from 1 to 9. No number may occur more than once

[12]. 数独 itself is a portmanteau of 数字は独身に限る, roughly "The numbers must occur only once."

in any row, column, or 3x3 box (indicated by the thicker lines). For the sake of simplicity, assume all puzzles are 9x9 and have a unique solution. Thus, the solution to the above puzzle would be:

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

While a human would use logic to find a solution, our `solve` function will be a bit more brutal, plugging in numbers one at a time until we find one that works, backtracking to the previous square when a dead end is reached. Psuedocode would look something like this to begin with:

```
solve(board, row, col) {
        return false; // backtrack
}
```

This indicates that out position in the problem that we are trying to solve is a specific row/column combination.

One thing that is **absolutely wrong**, but I see many students in Java do is:

```
public static boolean badSolve(board, row, col) {
        /*
        * maybe some code here, maybe not
        */

        for(int row = 0; row < 9; row++) {
                for(int col = 0; col < 9; col++) {
                        if(board[row][col] == 0){
                                // find a valid number for board[row][col]
                                // do the recursive stuff
                        }
                }
        }

        /*
```

```
        * maybe some code here, maybe not
        */
        return false; // backtrack
}
```

Students employing this strategy are trying to find the next blank spot fill it with the first number that works, then recursively call solve to find the next spot. The key issue here is that they are trying to find a blank spot. The algorithm should always know where it is in the problem, which is why `row` and `col` are given as arguments. Not doing so leads to issues of erasing original parts of the puzzle between what was originally given and what you placed.

A correct algorithm has two "base" cases and two recursive cases. For our base case, we would check if the `col` is out of bounds, meaning we are done with the current row. If so, go to the next `row`. If the `row` is out of bounds, we finished all rows and can return `true` to indicate a found solution.

Our recursive cases are fairly straightforward and depend on whether `board[row][col]` is empty or not. If it has a number already in it, that is a number that was given to us as part of the puzzle, so we recursively `return solve(board, row, col+1)`. If the square is empty, we look for a valid number to put in the square, then recursively call `solve(board, row, col+1)`.

Chapter 4

Trees

Our next major data structure is trees. Specifically, we will be looking at binary search trees.

Trees are an excellent data structure for storing things since they implement all the operations we care about for collections in logarhythmic time[1]

However, trees are not without limitations. Trees will only work with data that can be stored hierarchically or in an order.

4.1 Terminology

4.2 Youtube Dump

The other thing to know about this particular lecture is that I don't plan on doing any coding, well maybe a little bit, but I'm not actually my editor at any given point. This one is purely definitions and getting us getting our vocabulary. So fortunately, I'm not going to teach any like really weird words or anything that you need to know, like 'indicee' was a completely new word and a completely new context that you would have to figure out, or 'index,' right? And 'arrays,' those were weird words. Here, we're going to leverage knowledge that you already shouldn't have about trees. Okay, okay.

So, first and foremost, when it comes to trees, computer scientists really rock when it comes to naming things. We're really good at that. On the other hand, we make terrible gardeners. We're great at naming things because these things look like trees if you turn them upside down. All the trees we're dealing with, when I talk about trees, it's going to look like they're planted upside down, with the root at the top and all the leaves at the bottom, just imagine we're growing them in zero-G or something because we all like sci-fi or something like that. That's the only thing to say. But otherwise, they make sense as trees. You've actually dealt with trees like this before if you look at family trees and that kind of stuff.

[1]Specifically , Trees implement everything in average case log rhythmic time and worst case linear time , but if we do a bit of extra work and make it a self balancing binary tree (which will seem much later in this chapter) we can make this tree worst case log arhythmic for all operations

Today's lecture is on trees. We're going to learn how to use trees to represent hierarchical information, or hierarchical organization of information. We'll also be learning how to use recursion to process trees. Trees are actually fairly easy to program because you can do so recursively, and all the cases are basically the same. And 90% of tree problems can be solved by using one of two different algorithms we'll learn. And the remaining other ones, they can be solved by just throwing an additional data structure that you need around. So, if you need to do something weird, you can generally solve it by using a different data structure in addition to what you have here, such as a stack or a queue.

In this lecture, we're going to cover different ways of traversing a tree, and understand the differences between binary trees, binary search trees, and heaps. We're going to talk a bit about binary trees just because all binary search trees are binary trees. A binary search tree is a type of binary tree, so we're going to learn about binary trees indirectly. But the two primary data structures we're going to be learning this chapter are the binary search tree and the heap. And we're going to learn how to implement them. Binary search trees will use a linked data structure, so we'll use nodes, just like we use for a linked list, except the nodes are going to look slightly different. The variable names are going to change, but otherwise it's not going to get more complex than that. And then for heaps, we'll use arrays instead of all these data structures, but you could use either. We're also going to learn about basic compression when we talk about Huffman trees. So we're going to learn how to encode stuff and do basic file compression, making file sizes smaller by compressing the data and getting it in a lossless format.

All the data structures we've been dealing with so far, like lists, are linear. Every element has the item before and the items after it. So if you want to find something in a list, you have to go through the list, which takes time. Trees are nonlinear, meaning they're not stored in a list, they're not stored in a straight line. They're hierarchical, meaning that we treat something in the tree more importantly than something else. Tree nodes can have multiple successors, but only one predecessor. That doesn't really make sense now, but it will after you physically see what a tree looks like.

We deal with trees a lot. Class hierarchies, like the kind of hierarchies we've seen before with the object class and the collections class, and then all the different types of collections like lists and maps, and then under lists we have array lists and linked lists—that kind of hierarchy is modeled by using a tree. Your disk directory and subdirectories are also modeled by a tree. If you're using a UNIX machine right now, such as a Mac, or if you're running Linux (I'm running a UNIX box right here on Windows, the UNIX subsystem), the file system structure looks like a tree. If I go to the root, which is the topmost directory, what directories do I have inside? It's hard to see because it's blue on black. So let's see, properties, let's set the screen background to white. Sure, wonderful. Okay, pop-up background, pop-up text. Why in the world does it do that? I can't really do anything to change it. But I have directories. So much for being clever, right? Properties, screen background, let's set that back to black. So cd... then cd. In the binary directory, I have other stuff like this. Let's go into users, other stuff, games, include, libs. So let's go to cd Lib, or actually cd share. There's all sorts of stuff in here. We have a dictionary here, no, nothing in that. But basically, it's a tree-based, it's a hierarchical thing. You start at the top, and each folder has subfolders, and those subfolders have

their own subfolders. It's the same way it works in Windows. I've got my C Drive at the top, and then all my system files are in the Windows drive. Then I've got things like Program Files. Let's see what else I have. I've got Steam, which has all the games that I've installed on here. And in Steam apps, you have all the subfolders that hold the games, and then actually it's in common. So there are all the games I have. It's all basically folders inside folders inside folders.

In a file system, it's a tree-based, hierarchical thing. You start at the top, and each folder has subfolders, and those subfolders have their own subfolders. This is the same way it works in Windows. You have your C Drive at the top, and then your system files are in the Windows drive. Then you have Program Files, and other directories. For example, Steam contains all the games you've installed, and in Steam Apps, you have subfolders that hold the games, and in "common," you have all your games. So, it's all basically folders inside folders inside folders.

We can also represent a family tree, which consists of parents and their children. That's a tree in the name, so you understand how that looks.

Trees are recursive data structures because they can be defined recursively. While that definition isn't very helpful, it's more important that many methods to process trees are written recursively. The way a tree works is that every node in the tree is considered to be the root of its own subtree. This is why we are able to process them recursively, just like with a linked list, where every node was treated as the head of its own new linked list.

Why are we learning about trees and binary trees specifically? While there are trees with three or more children, "binary" means that each node has at most two successors. We deal with these because, when it comes to runtimes, binary trees are the jack-of-all-trades. They offer log(n) for add, log(n) for remove, log(n) for get, and log(n) for set. They are basically log(n) for everything if they're balanced. In Chapter 9 of the textbook (which is often skipped due to time constraints), they discuss self-balancing trees. These are trees that ensure they remain balanced. We won't cover them in that chapter, but in the real world, these self-balancing trees ensure that everything is always log(n) in both worst-case and average-case scenarios. However, with the trees we are learning, we aren't learning how to balance them because that involves learning how to do rotations. We're going to just deal with that. On average, it will be log(n) time. In the worst-case scenario, it could be O(n) time because you can accidentally grow a tree in such a way that it looks like a linked list. But that's very, very rare. On average though, log(n) time to do everything is very, very quick.

So I've been throwing out term successor, children, parent, root. So let's go ahead and get those. And what we're going to do is that we're going to leverage your, your knowledge of botany and family trees. And it doesn't need to be much botany at all to basically understand our stuff. So a tree is a collection of elements, our nodes. Okay, it's a set of elements or nodes, and every node is linked to its successors. So basically, these arrow, these links, there are links that point down. So here's dog, and it's connected to cat and wolf. So note at the top is called a root, is called the root of the tree. Okay, this is a very small tree with four nodes. Okay, now it doesn't necessarily make sense because it's like a dog is the parent of cat and wolf, cat and cat, it has a parent of canine. They'll make sense why it's in this order. Okay, so the node at the top of the tree is called its root. Okay, the links that go from from a node to its successors

are called, we can call them either links or we can call the brain, either way, everybody's going to know what you talked about, links or branches. Okay, and successor and predecessor, they, those are big, those are big warned, big words with lots of syllables. So why don't we just use something a bit easier? Uh, this is a dog as the parent of cat and wolf. Cat and wolf are the children of dog. So the successors of a node are called the children. Okay, and your predecessor, the node above you, is called your parent. Okay, every node in the tree has exactly one parent. Okay, basically, you have one boss, essentially. You have one parent, one person above you, except for the root node, who's an orphan, and he's all alone. Maybe it's Batman or something. So, so those guys, so everybody in the tree has exactly one parent, and you can have up to end. If it's a binary tree, you can have up to two children. But now since we've established this like pie of this child-parent relationship, we can actually leverage, we can leverage this like crazy. So for instance, cat and wolf, they're both children of dog. So we can call them what? Siblings. We can call them siblings. Notice that I have the same parents or siblings, right? Then we can also make this also much a much more generic relationship. Let's go ahead and use this fake laser pointer. Okay, so let's generalize this relationship. What would the relationship between dog and canine be? Yes, grandpa, grandpa. Let's go with grandparent, right? Grandparent and grandchild, right? Dog is the grandparent of canine. Canine is the grandchild of dog, right? So you can generalize this out. After grandparent, it becomes a bit silly, so you might just want to use ancestor and descendant at that point. Okay, at that point, use the ancestors descended rather than trying to fit count, be a number of greats. You have your great-great-great, right? And and also just try to avoid using like first cousin once removed, that stop. Um, because, well, I know what those term means, not everybody does. Um, so, so a node that has no children, it's called a leaf node, right? Because you're at if you're, here's the root, right? And so if you're at the opposite side of the roof, you're sorry, obviously side of the root, you're a leaf. Okay, so canine is a leaf, wolf is leaf because those two nodes don't have children. Cat isn't a leaf node because it has one kid. We can also call them external nodes, but that's boring and more syllables, right? So we just call them leaf, a leaf node, or a leaf or leaf nodes, or it leaves, right? Non-leaves can be cut, can be referred to as internal nodes if you really had to, but leaf is quicker to say. Okay, so again, the generalization of the is the ancestor descendant. So dog is the parent of cat. Cat if the parent of canine. Canine its sentence cat and dog is the ancestor of canine industry. Okay, so what makes trees trees though, is that they're recursive. Every node in this is the root of their own troop, of their own subtree. So dog is the root of his own tree made up of these four guys. Cat is the root of his own tree made up these guys. So cat is the root of this tree that starts a cat with the with the left branch of canine and the right branch of nothing. Canine is his own, is his, wolf is his own branch and it's just himself. And canine's his own brain, oh sorry, his own subtree, just himself. We also have two more terms. This one's a bit more abstract, which, but once, but basically it takes you about 10 seconds to remember what it is. The level of a node, it's basically the distance from its rib, from the root. So basically, how far down the tree are you? If you're at the top of the tree, you're level 1. If you're the second, if you're at the second row, you're level two, make sense, right? Basically, it's your distance from the roof plus Y. So the root is 0 from itself, so one. And we can, we'll see that we can

figure out what what your just what level you're at recursively. Okay, right? Um, the height is basically the length of the height of the tree, how big it is, or the thing that determines how big it is. And basically, it also determines a lot about the runtime will be is the longest path from the root to a weak node. And as we'll see, basically, the worst case scenario for an operation is that it's going to take height number of steps to do something in the tree if we're, if we're doing something like an add or remove. So the height of this tree is three, meaning because the longest path from root to a leaf is dog, cat, canine. It's not two because even though dog and wolf is a valid path, it's not the longest one. Another thing about these paths, it's not like one, two, three, four because these only go down. Let's go ahead and actually, hey, I've got a pen, I shouldn't use that, right?

These are arrows that go down. Typically, it's not, it's usually not bi-directional, right? These, you only have an attempt, you only point to your children, but your children, but you don't have a parent. Okay, so in a binary tree, each node has two subtrees essentially. So what makes binary trees, this is the formal definition of a binary tree. Each node has two subtrees. So a set of nodes is T's binary tree if either of the following is true. T is empty. So, so a set of nodes is a binary tree if it's a null set, right? Oh, and that's so this is a recursive definition. It's this is the base case over here. The recursive case is if it has two subtrees, left tree and right trees, such that left tree and right tree are both binary trees. So let's go ahead and put that on the board. Draw the example downward. So this is a binary tree. We'll see that this is not a binary search tree, but it is a binary tree right now. Okay, so this is a binary tree. If I were to do this, this is not a binary tree because one, because one of these guys has three, right? So first off, this, we're saying this entire set is a binary tree. We didn't split up into two. If we can take this and split up in, so we have the room and then the root has two trees, ones have left, the one to the right, and the one to left at the binary tree. So what about the ones are left? Well, this guy on the left, he has, he has three children, not to, right? He is a luxury, as left subtree, right subtree, in the center subtree. So he's not a binary tree, right? This center thing just doesn't work. Okay, so now he's a binary tree because he has two children, but he's not a binary search tree, which is what we're going to be dealing. So let's go ahead. So what we'll go back to these, these new, this other stuff later on. This stuff over here, like expression trees, we'll come back to that when we, when it acts, when we go to over tree traversals, same with Huffman trees. We'll come back to that. This is the different type of binary tree. But binary search trees are what we're going to be dealing with for the majority to chapter, which is that, um, and the reason we have this is the border in a binary search tree, all the elements to let on the left subtree are before all the elements on the right subtree. So this tree that we've been dealing with in the example, it's in alphabetical order, right? So cat and canine, soda. So cat and canine come before dog and wolf, and wolf comes after dog. Okay, so all the elements on the left come before the root, and all the elements on the right come after the root. And then this applies recursively for each subtree. In this subtree, all the elements to the right come before cat, and all these are all the elements left come before at all the elements the right come after cat. So a set of note, so this is the mathematical definition, a binary search tree. If T is right, well, this is the 702, you the binary search tree if either equality true, either if it's the empty set, or you have two subtrees, the left tree and the right tree,

such that they are both binary search trees. And all the stuff in the left side is bigger than all the root node is bigger than all the stuff on the left and smaller than all of this stuff on the right. So let's go ahead and deal that and do that with numbers because it's easier to check with numbers. Let me draw a binary search tree, or let me draw a binary search tree on the board, and then we'll go ahead and, you know, change it around to make sure that. So I'm going to start with basically just single-digit numbers because that's easy. Five, nine, seven over here, nine, six, three. So this is a binary search tree. So all these stuff on the left side of the tree is smaller than the root. All the stuff on the right side of the tree is bigger than the root. Not only that, it, this is true for any subtree we look at, right? Let's look at three. All the stuff to let them treat, those are all the stuff to the left of three is three is smaller than three. All the stuff to the right, it's bigger than three. Okay, so let's look at one. All the stuff that's to his left to his left and bigger than him. All the stuff to his right is bigger than him, which is, and or rather, all the sections left to right are binary search, right? And nothing use binary search tree. All right, technically, he's got a bunch of null pointers right over here, which in some solutely. So, and that matters. And visualizing that from x matters. And reviewing this stuff like red black trees, which are stuff that self balances again. So this is a binary search tree, and that's actually pretty cool because this is that it's sorted, believe it or not. This is sorted. Uh, so let's say I wanted to find six in the tree. Okay, say if I want to know if this tree holds six. Nah, just basically this is the DA contains method. So let's go ahead and see. So I'll start at the root and if, hey, does this tree contain six? And I was like, well, I'm five. I don't know what this tree contains six. Why should I keep track of all my kids? I've got a lot of them. It's a big family. But if you, but I know that basically six is bigger than me. So if you go to my right, so why don't you go ahead and ask my right subtree? So we don't have to right subtree because that will hold all the values that are bigger than pi, right? And as a consequence, we don't have to look at any of these values over here, right? Because those are all less. This is binary search all over again, right? So we buy, so by basically asking the question, how are we bigger or less than the root or equal to it, right? Just like is the number I'm thinking of bigger, smaller, or equal to? We can throw out half of the tree that we have to search. Now, basically, we split this tree in half, the search phase in half. If we do that with every time you ask question, it's logarithmic. So now I go to seven. Do you know where does this binary search tree contain six? And something's like, I don't know, I'm seven, so I'm not six. Why don't you go at, but if six is in this tree, he's going to be to the left because he's less than me. So you go to what the, okay, does this binary retreat surgery contain sex? He goes, yes, of course it does. Opposites. And so we return true to here, with your furniture to here, which will return true to use it. So three steps to buy to search seven items. Okay, um, and in fact, if the tree is properly balanced, a definition will learn once we get back from the break because that one's of the words that describe trees, which are full, perfect, complete, and balanced, which are like all these things that might describe some like on an optimal new weight, you know, an optimal lifestyle, right? Full about perfect, complete, balanced, like somebody who's basically, you know, posting all those positive messages on Facebook, right? Wondering if something is okay. These are, right, those are the words that we're going to use to describe trees. But we're going to be able to describe them in mathematical terms. So, so I guess they'll cause a lot

more irritation. So let's go ahead and do, so let's go ahead and and see if I was looking for one. I'd say, okay, five, one is less than five, one is less than three, one is equal to one to be bounded. Okay, so this is a binary search tree. Now, let's go ahead and make some changes. This still looks a binary search tree. Yeah, still binary search tree because all the items to left are bigger than the root and that also and that implies for every subtree. Okay, so now let's go ahead and see by root six, still binary search tree. What if I put six over here? No, because six is greater than five. It needs to be on this side of them. Okay, what if I were to put on say switch one over to here? Not a binary search tree. Well, because one is less than five on the right side of five, or beside the correct side of five, right, which is to the left, not the right. Three, oh sorry, one is to the right of three and it shouldn't be because one is smaller than three. Easy be on this side. Make sense to everybody there, right? It's got to be, it's got to be true all the way down. Got to be true completely recursively. Okay, so, um, let's see. So now let's go ahead and see it ask ourselves, is this a binary search tree? Five, seven, six, eight. This is a binary search tree. Yeah, everything to the right is big is all the elements on the light of everything gets bigger and all the items left or smaller moves. Let's go with this one. Go with five, ten, six, seven, with eight actually over here. So this is a binary search tree. Yes, it's a, it's a terrible one. It's this is definitely not what we call balanced, which again, is a term we have to define. But you can kind of see it's unbalanced, right? Five, ten, six, eight, seven. Okay, everything that's bigger than five, five is to the right. Everything that smaller than ten is two tens left. Everything that's bigger than six is to the left is to the left of a six in everything smaller than ages until is to let them of it, right? And we're looking at it in recursive manner, right? So this also brings us to an important consequence. When we're building trees, we always add like, we always add to a leaf essentially. When we're building a tree, we always build, we start from the root and then we build the leaves from the root and then we build out from there. So on, now this is not, now this is why I'm just saying that basically worst case scenario, it's going to take oh then to find something because this is just really a crooked looking link. Let's start here, right? If I wanted to find seven, I'd have to go, I have to go right, left, right, left. And basically, I'm not every time I'm making a choice, I'm just simply ruling out the thing I just surged. I'm not actually ruling out any tabs. I'm not having the search space at all. I'm doing a linear search here, which is so, um, this doesn't happen very often, but it can happen. But most of the time it doesn't, which is why we say the average time is a blog and to search for treating on average. But worst case scenarios like this are O of n, which are linear time. Um, now when we act again, once we actually use trees, if you're using a tree that's already been like a library that uses and treatments already built for you like the tree set, it's going to do worst case and average case and best case log of that. Some stuff might be O one because blog about. And why is this? Because will be something called self-balancing. And rebalancing the tree can be done in log n time. If you're basically when you add or remove something, you cause the tree to become unbalanced, it takes as so long as that was the up, you had one operation that calls to be unbalanced. Rebalancing it takes at log n time. So it's a very, there's a very effective manner on that. So today, it's just a very effective way to do that.

[Music]

Alright, let's see. Alright, so, so the cool thing about binary search trees, as

I mentioned, is that they never have to be sorted because they're always sorted. A consequence of this is that binary search trees, binary search trees always have to store sorted data. They can only be, can only use comparable or sort of Alita. You can only store comparable items in there, right? Because comparables can be, can you can put it, put them in order. And if you can't put them in order, you can't put them in a tree because going to the left and right has no meaning anymore, right? So you can put them in a binary tree, but not in a binary search tree. Okay, so as we add or remove items, the binary search tree will remain in order. Okay, this is better than like trying to keep an array or a linked list sorted, which will take cost, which will take linear time to do that. Okay, so again, worst case scenario, but on average, it's all of log n. Worst case scenario, it's O of n. Okay, I'm this was the algorithm who were using the pseudocode for. If the binary, let me remove my heads. So not that there were many letters to figure out. If the tree is empty, we return null, basically saying that the target is not found. Otherwise, if it matches the root data, we can return the data there. Otherwise, if it's less than the root, we look in our left subtree. Otherwise, we look in our right subtree. It's a very easy algorithm. This is one of two algorithms. Is what I call the search elephant. You other ones, the traverse algorithm. Those two algorithms together solve 90% of the, basically, our 90% of what you need to basically write all the algorithms you need. You'll treat the remaining 10, 10% can be dealt with either having a parent or variable or using using an additional data structure and being clever with that, such as a stack or queue. Um, it's very, it's just like link lists are very, are very popular. You use it as interview questions. Trees are very popular is interview questions. So let's go ahead and figure out those full, perfect, incomplete words. Those words that basically describe, scrap, there aren't really describing the wholesomeness of the tree's lifestyle. Who are we to judge that? But the, but rather, you know, just certain mathematical qualities they fulfill. So full is the easiest one to remember and probably and perfect is pretty much the next easiest one because it's visually easy to see. Full is easy to remember though. A full tree is a binary tree where all the nodes either have two children or no children. Okay, basically, uh, if you're a leaf node, you don't have any children by definition. So basically, we're saying here, all the non-leaves, they'll have both their children in slots filled, right? So here, in this example, this is a full tree, right? Because all the children, note, note zero, two, four, six, nine, eleven, and thirteen, they have no children. But seven, one, three, five, and ten and twelve, all of their children, all there, they both have, they all have two children. If I were to remove this two, right, this tree is no longer full because three has a right child, but not a left child. Make sense? See, can I erase all ink on slate? Cool. All right, that makes sense everybody. So full is a very easy one to see. You can just kind of look at it. In fact, you have to look at it to figure out to figure it out. And that's actually probably really good exam, a practice exam question, like write a method that detects whether that returns true if the tree is full and false if it's not, right? That's a pretty, that's a pretty nice one. I would do full, perfect, and complete binary trees. So a perfect binary tree, this one's also not so bad. It's the complete that gives people trouble, and it's understandable why because the definition is a bit obtuse. But once I start drawing, going to go through the examples, it's perfect is a full binary tree with a height of n. Honestly, it should be H to be honest, be a height of H with exactly two to the H minus one nodes because I like using n for the number of nodes. So it should

be, it should be a high eight, a pull by a binary tree of height H. So this one's of height three, and notice that this one's height three and it has seven nodes, which is two to the third power minus one. And there's only one possible way a binary search tree can do this if it's full and has a height, vivid hat. If it's full, has exactly this many nodes and has exactly this height, there's only one way it can look, which is this way. So let me go ahead and draw some perfect trees on the board, on the actual board. Okay, so this is a, this is a full tree. It also has complete, right? It's an empty tree. This one is a full, this one is full and perfect, right? Actually, and I'm going to ignore numbers right now. This one is full and it is perfect, right? All the nodes have exactly zero or two children, and its height is 1. And there's exactly one note of it to the one minus one. This is a perfect tree of my two, right? This is a perfect tree of height 3. The perfect tree of height 4. Basically, you know, it's each of the bump, each of the cup, each of the rows are completely filled up. That's kind of the best way to visually represent purgatory. Each of the rows are completely filled up, right? It's not too bad to understand. Okay.

So complete is the kind of a terrible one, but it's not actually too bad. It's this more annoying than anything else. So a complete binary tree is a perfect binary tree up to level, up to be, it's perfect all the way through the except for the bottom level, right? It's perfect up to this point. Okay, perfect complete trees are perfect up to this point, but then they have some extra leaf nodes on the bottom, and they're all towards the left, which is what makes this such a terrible descriptor. They're all over here to the left. So this is a complete treatment because this over here, let me go ahead and I'm going to start drawing them. This is a complete tree because this level is perfect, and the level below that is not perfect, but all the nodes over here are shifted over to the left. So this is a complete tree. This is a complete tree, complete and perfect, right? This is also complete, complete, but as soon as I removed this guy over here, this guy's left child, it's not complete anymore. Why? Because we have a gap here. Okay, and the answer, so imagine we were labeling all these on all these nodes zero, one, two, three, four, five, six, seven, eight, nine, ten. So long as I can keep doing this without a gap, right, it's a complete tree. But as soon as I go over here and I would write eleven here for this left child, as soon as I get over here and add, and have to skip the number, that is no longer complete. Does that make sense, right? There's a gap here. This is a, this isn't complete. Neither is this. This one is complete though. Okay, while there's no dots here, they're all shifted to the right, so this is not a complete. They all have to be towards the left. Okay, now you may be wondering why in the world do we have such a funky definition in the first place, right? Um, and the answer is, is that it's actually kind of irrelevant for binary search trees unless you're storing them and arrays. Because what you'll do is you'll store visit index zero, one, two, three, four, five, six, seven, eight, right? You can actually figure, you can use this notation over here to figure out where all of us, where they will be stored in an array. And there's actually a formula that will learn when we go through piece different gear out what slot and B, right? You need to be storing these things on. But we're going to learn how to do that, you know, and link them with nodes instead of parades. But primarily, we're going to use this in heaps, and heaps are always going to be complete, always. It's kind of their definition. So we'll revisit this when we go over heats. So full and perfect, those are important for trees. The complete is important for heat, which is specific type of tree.

Um, so I think it says is that we're not going to discuss general trees in this chapter, except we are discussing it right now. So I guess they're liars. But so the nodes of, but the point is, is that they're saying we're not figuring out how to solve these general trees, which is they're basically, yeah, you can make not just binary trees, but trinary trees and quaternary trees and whatever and airy trees you want. But we can pretty much, but we're going to show you that that doesn't matter because we can take this general tree. But the, you know, where and a tree basically, it's a tree so long as basically we have one parent. Okay, right? So here's basically like the, it's, you know, the English monarchy here, that's what we're doing here. And we're just looking, looking at basically that the Royal issue of the Royal of the Royal monarchy. Okay, so William the First, and then Henry the Tilde, Henry the Second, John, Henry Third, Edward, right? We can actually turn this into a binary tree so long as we basically care about only one parent and everyone. And you're like, no way, there's no way you do that. So here's the binary tree of that. And it's kind of just tilted. Okay, we kind of just tilted it vertically. And how did we change this? Well, if you sometimes you just have to be clever. In this case, what we did is that we said our left branch, so we're going to not use the term child over here because, you know, we're talking about people with actual children. So we're going to use left and right branch. So the left branch is the oldest actual child, right? And your right branch is your youngest sibling, right? So that's why we're using the branch term here, not the kid trip child because you want to make sure we're not. And that's also why we have branch, the branch terminology because that way we can, we can make sure we're not over, we can avoid overloading it too much. So the left branch is your oldest kid, and your right branch is your youngest sibling, right? So we'll use the first oldest kid is Robert. And bears all and to the right is all of William's kids, Robert, William the Second, Adela, and Henry, right? William and there's all his kids. So we just connected them all to each other. And if they have their own kids, those are on the left. So it's a pretty cool way to do that. So that's why I say, so you can pretty much turn these general trees into binary trees and get, you know, some log n time. But there's, uh, you know, there's some, no, I don't want to keep the said I did. So whatever. But you can do some really interesting things like learn about two, three trees, not two or three offense to three trees, right? You can do stuff where basically we have trees that not only have two children, but three children. So basically, you have two numbers in here. All the stuff that's less than a goes the left. All the stuff bigger than me is goes to the right. A is smaller than B. And if there's something in between, it goes in the center, right? So you can totally do that. And it's a two, three tree because it combines two nodes and three nodes. And then you can have two, three, four trees and all these other things like B trees, which is where it really gets exciting. Uh, you know, and that's kind of where it really gets advanced and really exciting. And, you know, even though it's 40 years old, we still teach them because they get used a lot for the for all as this basis for building a lot of other stuff. B plus tree, right? So the point here is that basically the stuff we're learning here is it is integral to understanding how basic file systems work. Integral. Okay, so all right.

So now we've got this tree up on the board. Let's go ahead and actually put an actual binary search tree on the board. The question is like, how do I like, you know, print it out or something, right? Like it's pretty easy to print out a

list. I just go through the list and print it out, right? Okay, now the obvious solution, now the obvious answer pens in your head. It's like wondering, stop beating around the bush, right? You're just going to cook them in order, right? And that's completely correct. One of the ways to print out a tree is, let's go with this one. Five, seven, six, eight, three, two, right? Easy to use single-digit numbers here, right? So say we wanted to print this out. There's a couple ways to do this. There's a couple ways to basically go through all the trees in the note, sorry, although all the nodes in the tree. Okay, because that's the question we're doing. Keith, we're asking here, when we want to print out all the nodes and all values in the tree, or we want to print the tree, you want four double ones in the tree, which means we want print out all the values in tree. This is called a tree traversal, which is the second algorithm. First one was search e, which is zigging and zagging your weight of the tree. The other question is, is that oftentimes you want an algorithm that hits everything in the tree, right? So on, so often we want to describe the nodes of a tree and the relationship. And we can basically walk through the tree. So there's three different types of tree traversal: in-order, pre-order, and post-order. Addition, what possibly, I guess, terrifyingly familiar because we were talking not too long ago. And we were talking about stacks. It out in order notation, preordered notation, post order notations. That about to make a repair. Biers, I fear some, yeah, that will definitely be making a reappearance today. So there's three types of traversal. We're going to start with in-order, even though it's technically between the first two, right? Those in the middle because it's going to print stuff in order. Okay.

In-order is one of those that are wonderfully named. If you do an in-order traversal, it's going to go through everything in the sorted order of the array. So let me just go, I'm going to go through the output, and then I'll just simply, actually, let me just. So if I do an in-order traversal of this tree, I will do two, three, four, five, six, seven, eight. Okay, so because that's going to produce in order. And the way that it works is that basically, well, uh, because it's a binary search tree on everything that's less than the roof is on two left. So I'm going to do an in-order traversal of the left subtree first. Okay, and recursively do that. Then once that's done, I'll print out myself, and then I'll do a in-order traversal of my right subtree. Okay, and because that's recursive, right? Let me go. So what I will do is says, I'll do an in-order traversal of the tree starting at three. And three will go, I'll do an in-order traversal of the tree starting at two. Okay, and she will go, I'll do an in-order traversal my left, then I'll do myself, then I'll do my right. My left and my right were empty, so that was easy. Then three goes, okay, my left side is done. I'll do my self, and I'll do my right side. My right side and so my right side goes, okay, let me do my left myself and then my right side, right? And that happens recursively. Then he reports that it's finished. And five says that it's right. That makes sense, right? So it just simply goes left, myself, and right. Now, if you take a look at all these algorithms that I still have up on the screen, notice that basically the left always comes before the right. And all of these traversal, the only thing that changes is whether or not you're visiting, we're handling the root node first in between our left and right branches or after, right? You can even see that in the algorithms down here. The only thing that changes the prefix, right, of the function we call. And and one of them, what line visit the root appears on line three, line four, line five. Okay, so what's the prefix? So what's the pre-order traversal going to look like? No, I'm going to go ahead and know that you have enough time to study this.

Go over here. Um, generally, but this is worth somewhere between six and eight points because not hard, but enough people, but but there's always just like one, there's like three people who do get it wrong. So it's worth keeping their it on the exam. So five, so here was what a pre-order traversal does is that we started the room. And pre-order says, I'm going to visit my cell first, then we'll do my left and not only my way. So does it myself first, then I'll visit my left. Okay, let goes, okay, let me visit myself, then visit my left and my right within two, right? Okay, then we go over here to the right side. So seven goes, let me visit myself, then I'll visit my left, right? And they do that same. So we've got the route here, then we've got our left and right trees. And then each of those guys, right? So there's kind of a structure that can be printed out here, right? Um, but basically, we've got myself and my left side, right? So there's kind of a structure that can be printed out back. The truthstream method that I already have written for my debugging my trees, what it does that an indent stuff in a nice manner. It will do something, it'll print it out in feet in order. Print out this treatise something like this. Five, and then it will indent it over here with three. And then two, four, sorry, then yeah, then to four for its children. And then it will have seven. It basically prints it out level by level where it's where the indent shows you what level that item is on. So it's not too bad.

Post-order is, well, let me, is I'll visit myself last. I'll do my left side, then my right side, then myself. So fought, so here I will go, let's do my left and my right. So then three will go, let me do my left and my right, and I'll do myself. Two, four, three, left, right, then myself. Um, so five ghosts, I do my left side, let's go might be my right side. This side goes, let me do my left, my right, and myself. Six, eight, seven. And now it does it myself, five. So we've got the left side, the right side of children, and the right children. Okay, and again, it's just, it's another way of just showing the order of the tree. So another question is like, okay, but are there any like actual applications of this? Okay, so I have to go through, let's see. I'm going to go all the way back to the, let's go all the way back, say here, too far, all the way forward. There we go. So the expression trees, I told you it come back this. Okay, so the idea here is that we can take some expression like X plus Y times A plus B over C. Okay, and we can encode that in a tree. We don't need to install operators industry. But basically, every node in the tree is an operator or an operand. Okay, right? It's a binary tree because up all the operators in you either have pretty much all the operators in in arithmetic our binary except for the positive sign in the negative sign, which just simply says, you know, make this positive, make this negative, which we could just easily encode. Um, so time, so X plus Y that, um, you know, that gets encoded here. Times A plus B divided by C. So the parentheses don't need to be stored in here because the tree structure dictates the order of operations, right? We're multiplying the result of adding of adding X and Y together, and we're adding and we're multiplying that against the result of dividing the sum of A and B divided by C. Okay, so the operators, so if you're at a higher tree level, it gets evaluated after like the lower you are in the tree, the higher precedences, right? Because we're going to need to solve you first. We're going to need to solve these first. And all and it makes sense that basically all the variables are going to be at the bottom of the tree because, um, they did, you know, if we had an operator there, that wouldn't make sense. Plus, if this was a plus, it would be plus what, right? Plus what left and what right? It wouldn't make sense there. So all the things have to be at the end. So

what does this have to do with infix notation, prefix notation? Well, if we do an in-order traversal of this tree, we would be, let's see, so do my left side. So this is original expression. Let's do my left side. So X plus Y times A plus B divided by C, right? So I always, so I was just doing my left divide. If I was simply doing my left side, then myself and my right side into doing that recursively. And what that prints out is X plus Y times A plus B times plus B divided by C. Okay, and why does it do that? Because that's the order that it was stored into of the expression that's in fixed notation, right there. Ah, but it's ambiguous because we need the parenthesis to figure out what we originally wanted, right? But if we remember, if we do a post order or pre order, a post order traversal will give us the post fix notation of this. It will give us so X, Y plus A plus B. So A, B plus C divided. And then multiply those results together, right? So we could just simply throw this into a stack and it's going to work, right? And it will be completely unambiguous. The operators follow the operands. Here, we do a pre-order, we get the Polish notation, which is prefix notation. So here we treat this kind of as a function. We were multiplying plus X, Y, right? So multiplies the function call. And it's arguably car the function call plus, and it's arguments are X and Y. And the other argument for multiplied was divided. And its arguments where plus S X, Y, sorry, plus A, B. And the other argument for divide was C. So again, we don't need any parentheses here to actually being able to parse this, which is really cool, which is why again, all those Facebook fights about that stuff is should be answered with you should be using postfix notation. And you wouldn't have these arguments. We are, we pretty much we blazed through that and we still got 20 minutes. So there is a bit of stuff to do that we did cover because the other students, you know, or slightly more inquisitive. Um, but let's go ahead and see. So first off, I mentioned that we're going to be using the node class here. So what is the node going to look like? You know, what does the node look like actually? So just to make you sure you understand the structure. So here's our node. So this is the internal node. So it's private. And I'll put me here, although it's going to be E extends comparable. I only have so much space, right? Because again, these things are going to be convertible. Okay, E, I'm going to call it data this time, right? Because we store data in the tree. You could call it item if you want, you know, private E private. And then it has a, it stores the memory location of another. Now, know D, but instead of calling Nick them exit pre, we're going to call them left or my left child. And we're also going to call it right for my right shots. Private, no, E right. All right. And now you've got basically just use and then we would just simply use the same constructor where we'd taken a date, we'd take in some data as an argument and store it in. Okay, other Y. So in other words, except for left and right, it's exactly the same as the note we used in in a linked list. Well, it makes it different is so what makes actually this different is course we call it left and right. And the way that the algorithm builds stuff is going to be a bit different. So this will obviously be stuff I'll have to review when we get back because you're probably not going to remember a lot of it. And that's understandable because a week is a long time. Yeah, so they have your left, right. It'll have a two string there. But basically, the way it's going to work is that your binary tree is going to have a root. And that's all it's going to keep track of. And that note in each node will point to a left child and the right child. And each of those will point to their own left and right child's, which might may or may not be known. Okay, and so what are we going

to have to figure out how to do this? Sorry, what methods do we really need? Well, they're, they have listed a bunch of methods like we'll get the left subtree. You get the right subtree of this. Honestly, that's not too necessary. But we really need our stuff like, um, adding, really, it's these things, adding something to the tree, removing something to the from the tree, and containing something in the tree. Sorry, asking if they contains them. The difference between a delete and remove is basically remove returns a boolean to say, hey, we successfully removed it. And delete return actually will return the value remove. Okay, so knows that this one returns null if you don't find anything. This one returns false if you didn't find anything to remove. Contains tells you if something's in the tree, whereas find actually returns the thing in the tree. You know, so one of the things that we will actually have to do is because Java in Java, you're limited to because you are actually limited to returning only one thing because of the way we write this, which makes it a lot easier. We're going to cheat a little. We're basically, we're going to store the values we want to return as as instance variables and then return the instance variables. So we have to do a little hackery where basically we have an outside value. We're going to store stuff in to give it back to the user because when we get into our recursive methods, our recursive methods are going to have to return on nodes to the user. I just just to make it so that it's nice and it's easy to read. But as a result, we have to just cheat a bit. But that's fine. It's not cheating if it works. But, um, we'll go over add and remove there. So let's see. Still 20 minutes left. But I didn't get into this into the evening lecture. And I don't want to get too far ahead. Let's see. Okay, so I'm probably going to actually stop the electric here because we're not like"

4.3 The Parts of a Tree

The first thing we need to do when introducing trees is define a vocabulary.

Much like the linked list, a tree is made of nodes. However, unlike a linked list , nodes in a tree are not arranged in a line, Instead, they are arranged in a heirachy.

Each node sit above multiple other nodes , with the nodes below it being referred to as their children or child nodes. The node connecting all these children is called the parent.

<A picture of one node, Represented by a circle with four arrows coming out below it. Each arrow points to yet another node. The Node with the arrows coming out of it is the parent, and the nodes below it are the children >

This relationship can be extended Ad infinitum as we can see with the picture below

<Picture with nodes labeled>

However anything above grandchild and grandparent just becomes tedious , so we tend to Generalize this relationship to ancestors and descendants. A key point here is to remember that while we are borrowing terms from the family tree , nodes will only have one parent . Each node can have multiple children, however .

We refer to the links connect each of the nodes as branches or links or edges. This tends to be a matter of personal preference.

Finally , we have one special node that sits above all the other nodes . This

note is the root and it is analogous to the head of a linked list . All of our operations will start at the root of the node[2].

Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[4]

4.3.1 Where the Recursion comes in

There is a reason we learned recursion before we introduce trees. Trees are the exemplar recursive data structure

Each tree has a root and That route has children . If we view each of those children as the root of their own subtree , this can make our algorithms for adding removing and searching extremely easy to write.

<picture Of tree, the recursive subtrees are dash circled.>

<Picture of the left subtree, with it's trees circled>

4.4 Binary Search Trees

A diagram of a binary search tree. It is made up of nodes, represented by circles, and edges (also called links or branches), represented by arrows.

In our diagrams, we will be using ◯ to represent node. A × will be to represent a `null` type, eg the explicit absence of a Node. Finally, △ represents a subtree. This could be any size, including empty.

4.5 Building a Binary Search Tree

4.5.1 The Code Outline

As explained in Section 3.4.3, when we use the `Comparable` class in Java to require that all objects stored in the tree has a **total ordering**, meaning every pair of objects we're storing has an ordering. In practice, this means that anything `Comparable` can be sorted.

Python, of course, doesn't need these restrictions.

```
public class BinaryTree<E extends Comparable<E>> {
}
```

Much like our Linked List, we don't need much in the way of instance variables. We'll create a `root` to keep track of the starting place for our tree and size to keep track of how many items we have stored.

Finally, we will also create our inner `Node` class for the Tree. It needs to hold the item and the locations of the left and right children. We'll also go ahead and add a The constructor and a method for printing out the item in the node (`toString` in Java and `__str__` in Python)

[2]Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[3]

[4]Or maybe it's some weird hydroponic zero-G kind of thing.

Listing (Java) 4.1: The Constructor and Inner Class

```
public class BinaryTree<E extends Comparable<E>> {
    private Node<E> root;
    private int size;

    public BinaryTree() {
        this.root = null;

    }

    private static class Node<E extends Comparable<E>> {
        private E item;
        private Node<E> left;  // left child
        private Node<E> right; // right child
        public Node(E item) {
            this.item = item;
        }
        public String toString() {
            return item.toString();
        }
    }


}
```

4.5.2 Contains

4.5.3 Add

All of our operations in out `BinaryTree` will be implemented recursively.

4.5.4 Delete

Part IV

Hashing

Part V

Relationships

Chapter 5

Graphs

In some ways, Graphs are the most important data structure. Graphs represent and model relationships, and humans are defined by relationships.

Graphs have two components: vertices (also called nodes) and edges. Graphs model the relationships between different vertices by connecting them with edges:

AN EXAMPLE GRAPH

The archtypical examples of graphs used to be maps and the distances between landmarks or looking for the shortest path.

With the advent of social media, we can talk about graphs with a few examples that might be easier to intuit.

5.1 Introduction and History

5.2 Qualities of a Graph

The physical layout of a graph doesn't actually matter[1]

5.2.1 Vertices

- Vertices must be unique.

[1]Some properies, such as whether a graph is *planar* or *bipartite* effectively care if a graph can be physically laid out in a certain way.

5.2.2 Edges

Undirected Edges

Directed Edges

Weighted Edges

5.3 Special Graphs and Graph Properties

5.3.1 Planar Graphs

Graphs that are planar can have their vertices and edges laid out in such a way that no two edges will cross.

5.3.2 Bipartite Graphs

5.3.3 Directed Acyclic Graphs

5.4 Building a Graph

5.4.1 Adjacency List

5.4.2 Adjacency Matrix

Matrix multiplication and GPU Abuse

5.5 Graph Libraries

Your programming language of choice may not

5.5.1 Java - JUNG

5.5.2 Python - networkx

There is only one realistic choice for using graphs in Python. The package networkx is extremely powerful, extremely versatile,and actively maintained.

5.6 Graphs, Humans, and Networks

5.6.1 The Small World

The Milgram Experiment

The Less-Known Milgram Experiment

5.6.2 Scale Free Graphs

5.7 Graphs in Art and Nature - Voronoi Tessellation

Figure 5.1: The wings of a dragonfly. Credit: Joi Ito (CC BY 2.0)

Chapter 6

Graph Algorithms

6.1 Searching and Traversing

6.1.1 Breadth First Search

6.1.2 Depth First Search

6.2 Shortest Path

6.2.1 Djikstra's Algorthim

Improving The Algorithm

Failure Cases

6.2.2 Bellman-Ford

6.3 Topological Sorting

6.3.1 Khan's Algorithm

6.4 Minimum Spanning Trees

6.4.1 Kruskal's Algorithm

6.4.2 Prim's Algorithm

End of book.

Bibliography

[1] Leonardo Bonacci. Liber abaci. 1202.

[2] Python Devs. listobject.c.

[3] Phineas Mordell. *The origin of letters and numerals: according to the Sefer Yetzirah*. P. Mordell, 1914.

[4] Laurence Sigler. *Fibonacci's Liber Abaci: A translation into modern English of Leonardo Pisano's book of calculation*. Springer.

[5] Parmanand Singh. The so-called fibonacci numbers in ancient and medieval india. *Historia Mathematica*, 12(3):229–244, 1985.

[6] Susie Turner. Flowers and the fibonacci sequence. `https://www.montananaturalist.org/blog-post/flowers-the-fibonacci-sequence/`, April 2020. Accessed: 2025-06-11.

[7] Unknown - Possibly Abraham. Sefer Yetzirah.