Anyone who has written even a few lines of computer code comes to realize that programming largely revolves around **data.** Computer programs are all about receiving, manipulating, and returning data.

Data is a broad term that refers to all types of information, down to the most basic numbers and strings.

Data structures refer to how data is organized.

You're going to learn in this book that the **organization of data** doesn't just matter for organization's sake, but can significantly impact **how fast your code runs**.

## ARRAY

The **array** is one of the most basic data structures in computer science. We assume that you've worked with arrays before, so you're aware that an array is simply a list of data elements.

The index of an array is the number which identifies where a piece of data lives inside the array.

Most data structures are used in four basic ways, which we refer to as operations. They are:

**1. Read:** Reading refers to **looking something up from a particular spot within the data structure.** With an array, this would mean looking up a value at a particular index.

Reading from an array actually takes just one step. This is because the computer has the ability to jump to any particular index in the array and peer inside.

When the computer reads a value at a particular index of an array, it can jump straight to that index in one step because of the combination of the following facts:

**1.** A computer can jump to any memory address in one step. (Think of this as driving to 123 Main Street - you can drive there in one trip since you know exactly where it is.)
**2.** Recorded in each array is the memory address which it begins at. So the computer has this starting address readily.
**3.** Every array begins at index 0.

Reading from an array is, therefore, a very efficient operation, since it takes just one step. An operation with just one step is naturally the fastest type of operation. One of the reasons that the array is such a powerful data structure is that we can look up the value at any index with such speed.

**2. Search:** Searching refers to looking for a particular value within a data structure. With an array, this would mean looking to **see if a particular value exists within the array**, and if so, which index it's at.

Now, when you and I glance at the shopping list, our eyes immediately spot the "dates" , and we can even quickly count in our heads that it's at index 3. However, **a computer doesn't have eyes, and needs to make its way through the array step by step.**

Aha! We've found the elusive "dates" . We now know that the "dates" are at index 3. At this point, the computer does not need to move on to the next cell of the array, since we've already found what we're looking for.

In Why Algorithms Matter we'll learn about another way to search, but this basic search operation - in which the computer checks each cell one at a time - is known as linear search.

Another way of saying this is that for N cells in an array, linear search will take a maximum of N steps. In this context, N is just a variable that can be replaced by any number.

**3. Insert:** Insertion refers to **adding another value to our data structure.**

The efficiency of inserting a new piece of data inside an array depends on where inside the array you'd like to insert it.
Let's say we wanted to add "figs" to the very end of our shopping list. Such an insertion takes just one step.

As we've seen earlier, the computer knows which
memory address the array begins at. Now, the computer also knows how many elements the array currently contains, so it can calculate which memory address it needs to add the new element to, and can do so in one step.

Inserting a new piece of data at the beginning or the middle of an array, however, is a different story. In these cases, we need to shift many pieces of data to make room for what we're inserting, leading to additional steps.

So we can say that insertion in a worst case scenario can take up to N + 1 steps for an array containing N elements. This is because the worst case scenario is inserting a value into the beginning of the array in which there are N shifts (every data element of the array) and 1 insertion.

**4. Delete:** Deletion refers to removing a value from our data structure.

**When we measure how "fast" an operation takes, we do not refer to how fast the operation takes in terms of pure time, but instead in how many steps it takes.**

**We can never say with definitiveness that any operation takes, say, 5 seconds.** While the same operation may take 5 seconds on a particular computer, it may take longer on an older piece of hardware, or much faster on the super-computers of tomorrow. **Measuring the speed of an operation in terms of time is flaky, since it will always change depending on the hardware that it is run On.**

So we've just seen that when it comes to deletion, the actual deletion itself is really just one step, but we need to follow up with additional steps of shifting data to the left to close the gap caused by the deletion.

We can conclude, then, that for an array containing N elements, the maximum number of steps that deletion would take is N steps.

## THE SET

A set is a data structure that **does not allow duplicate values to be contained within it.**

**Insertion, however, is where arrays and sets diverge.** Let's first explore inserting a value at the end of a set, which was a best case scenario for an array. With an array, the computer can insert a value at its end in a single step.
**With a set, however, the computer first needs to determine that this value doesn't already exist in this set - because that's what sets do: They prevent duplicate data. So every insert first requires a search.**

Insertion into a set in a best case scenario will take N + 1 steps for N elements. This is because there are N steps of search to ensure that the value doesn't already exist within the set, and then 1 step for the actual insertion.

In a worst case scenario, where we're inserting a value at the beginning of a set, the computer needs to search N cells to ensure that the set doesn't already contain that value, and then another N steps to shift all the data to the right, and another final step to insert the new value. That's a total of 2N + 1 steps

**In this chapter, we're going to discover that even if we decide on a particular data structure, there is another major factor that can affect the efficiency of our code: The proper selection of which algorithm to use.**

When applied to computing, an algorithm refers to a process for going about a particular operation. In the previous chapter, we analyzed four major operations, including reading, searching, insertion, and deletion. In this chapter, we'll see that at times, it's possible to go about an operation in more than one way. That is to say, there are **multiple algorithms that can achieve a particular operation.**

**The ordered array is almost identical to the array we discussed in the previous chapter. The only difference is that ordered arrays require that the values are always kept - you guessed it - in order.** That is, every time a value is added, it gets placed in the proper cell so that the values of the array remain sorted. In a standard array, on the other hand, values can be added to the end of the array without taking the order of the values into consideration.

Let's see how linear search differs between a regular and ordered array.
Say that we have a regular array of [17, 3, 75, 202, 80] . If we were to search for the value 22 - which happens to be nonexistent in our example - we need to search each and every element because the 22 can potentially be anywhere in the array. The only time we can stop our search before we reach the array's end is if we happen to find the value we're looking for before we reach the end.
With an ordered array, however, we can stop a search early even if the value isn't contained within the array. Let's say we're searching for a 22 within an ordered array of [3, 17, 75, 80, 202] . We can stop the search as soon as we reach the 75, since it's impossible that the 22 is anywhere to the right of it.

We've been assuming until now that the only way to search for a value within an ordered array is **linear search.** The truth, however, is that linear search is only one possible algorithm - that is, it is one particular process for going about searching for a value. **It is the process of searching each and every cell until we find the desired value.** But it is not the only algorithm we can use to search for a value.
The big advantage of an ordered array over a regular array is that an ordered

array allows for an alternative searching algorithm. This algorithm is known as binary search, and it is a much, much faster algorithm than linear search.

The major advantage of an ordered array over a standard array is that we have the option of performing a binary search rather than a linear search. Binary search is impossible with a standard array because the values can be in any order.

**Binary Search Vs. Linear Search**

In order to help ease communication regarding time complexity, computer scientists have borrowed a concept from the world of mathematics to describe a concise and consistent language around the efficiency of data structures and algorithms. Known as **Big O Notation,** this formalized expression around these concepts allows us to easily categorize the efficiency of a given algorithm and convey it to others.

**O(N)** is the “Big O” way of saying that for N elements inside an array, the algorithm would take N steps to complete. It’s that simple.

Rather, it describes how many
steps an algorithm takes based on the number of data elements that the algorithm is acting upon. Another way of saying this is that Big O answers the following question: **How does the number of steps change as the data Increases?**

**O(1)** is the way to describe
any algorithm is that doesn’t change its number of steps even when the data Increases.

**Big O Notation generally refers to worst case scenario unless specified otherwise.**

In Big O, we describe binary search as having a time complexity of:
**O(log N)**

**Simply put, O(log N) is the Big O way of describing an algorithm that increases one step each time the data is doubled.**

Logarithms are the inverse of exponents.

Now, log 2 8 is the converse of the above. It means: How many times do you have to multiply 2 by itself to get a result of 8?

Another way of explaining log 2 8 is: If we kept dividing 8 by 2 until we ended up with 1, how many 2's would we have in our equation?
8 / 2 / 2 / 2 = 1

With Big O, you have the opportunity to compare your algorithm to general algorithms out there in the world, and you can say to yourself, **"Is this a fast or slow algorithm as far as algorithms generally go?"**

**Sorting algorithms** have been the subject of extensive research in computer science, and tens of such algorithms have been developed over the years. They all solve the following problem:

**Given an array of unsorted numbers, how can we sort them so that they end up in ascending order?**

Therefore, in Big O Notation, we'd say that Bubble Sort has an efficiency of O(N 2 ).
Said more officially: In an O(N pow 2 ) algorithm, for N data elements, there are roughly N 2 steps.
O(N 2 ) is considered to be a relatively inefficient algorithm, since as the data increases, the steps increase dramatically.

The **Bubble Sort algorithm** contains two kinds of steps:
**1.** Comparisons - in which two numbers are compared with one another to determine which is greater.
**2.** Swaps - two numbers are swapped with one another in order to sort them.

The steps of **Selection Sort** are as follows:
1) We check each cell of the array from left to right, to determine which value is least. As we move from cell to cell, we keep in a variable the lowest value we've encountered so far. (Really, we keep track of its index, but for the purposes of the diagrams below, we'll just focus on the actual value.) If we encounter a cell that contains a value that is even less than the one in our variable, we replace it so that the variable now points to the new index.

2) Once we've determined which index contains the lowest value, we swap that index with the value we began the passthrough with. This would be index 0 in the first passthrough, index 1 in the second passthrough, and so on and so forth

In our case, what should technically be O(N 2 / 2).

So Big O is an extremely useful tool, because if two algorithms fall under different classifications of Big O, you'll generally know which algorithm to use since with large amounts of data, one algorithm is guaranteed to be faster than the other at a certain point.

However, the main takeaway of this chapter is that when two algorithms fall under the same classification of Big O, it doesn't necessarily mean that both algorithms process at the same speed. After all, Bubble Sort is twice as slow as Selection Sort even though both are O(N 2 ). So while Big O is perfect for contrasting algorithms that fall under different classifications of Big O, **when two algorithms fall under the same classification, further analysis is required to determine which algorithm is faster.**

When evaluating the efficiency of an algorithm, we've focused primarily on how many steps the algorithm would take in a worst case scenario. The rationale behind this is simple: I**f you're prepared for the worst, things will turn out okay.**
However, we'll discover in this chapter that **the worst case scenario isn't the only situation worth considering.** Being able to consider all scenarios is an important skill that can help you choose the appropriate algorithm for every Situation.

Insertion Sort consists of the following steps:
1) In the first passthrough, we temporarily remove the value at index 1 (the second cell) and store it in a temporary variable. This will leave a gap at that index, since it contains no value:

In subsequent passthroughs, we remove the values at the subsequent indexes.
2) We then begin a shifting phase, where we take each value to the left of the gap, and compare it to the value in the temporary variable:

If the value to the left of the gap is greater than the temporary variable, we shift that value to the right:

As we shift values to the right, inherently, the gap moves leftwards. As soon as we encounter a value that is lower than the temporarily removed value, or we reach the left end of the array, this shifting phase is over.
3) We then insert the temporarily removed value into the current gap:
4) We repeat steps 1 through 3 until the array is fully sorted.

There are four types of steps that occur in Insertion Sort: Removals, comparisons, shifts, and insertions. To analyze the efficiency of Insertion Sort, we need to tally up each of these steps.

Shifts occur each time we move a value one cell to the right. When an array is sorted in reverse order, there will be as many shifts as there are comparisons since every comparison will force us to shift a value to the right.

Removing and inserting the temp_value from the array happen once per passthrough. Since there are always N - 1 passthroughs, we can conclude that there are N - 1 removals and N - 1 insertions.

Big O Notation only takes into account the highest order of N.

It emerges that in a worst case scenario, Insertion Sort has the same time complexity as Bubble Sort and Selection Sort. They're all O(N 2 ).

**The Average Case**

Indeed, in a worst case scenario, Selection Sort is faster than Insertion Sort. However, it's critical that we also take into account the average case scenario. Why? By defintion, the cases that occur most frequently are average scenarios. The worst and best case scenarios happen only rarely.

**Best and worst case scenarios happen relatively infrequently. In the real world, however, average scenarios are what occur most of the time. And this makes a lot of sense.** Think of a randomly sorted array. What are the odds that the values will occur in perfect ascending or descending order? It's much more likely that the values will be all over the place.

While in the worst case scenario, we compare and shift all the data, and in the best case scenario, we shift none of the data (and just make one comparison per passthrough), for the average scenario - we can say that in the aggregate, we probably compare and shift about half of the data.

In the worst case, there are 6 comparisons and 6 shifts, for a total of 12 steps. In the average case, there are 4 comparisons and 2 shifts, for a total of 6 steps. In the best case, there are 3 comparisons and 0 shifts.
We can now see that the performance of Insertion Sort varies greatly based on the scenario. In the worst case scenario, Insertion Sort takes N 2 steps. In an average scenario, it takes N 2 / 2 steps. And in the best case scenario, it takes about N steps.

**Most programming languages include a data structure called a hash table, and it has an amazing superpower: Fast reading.** Note that hash tables are called by different names in various programming languages. **Other names include hashes, maps, hash maps, dictionaries, and associative arrays.**

A hash table is a list of paired values. **The first item is known as the key, and the second item is known as the value.** In a hash table, the key and value have some significant association with one another.

**This process of taking characters and converting them to numbers is known as hashing. And the code that is used to convert those letters into particular numbers is called a hash function.**

There are many other hash functions besides this one. Another example of a hash function is to take each letter's corresponding number and return the sum of all the numbers. Another example of a hash function is to return the product of all the letters' corresponding numbers.

The truth is that a hash function needs to meet only one criterion to be valid: **A hash function must convert the same string to the same number every single time it's applied.** If the hash function can return inconsistent results for a given string, it's not valid.

Ultimately, a hash table's efficiency depends on three factors:
**1.** How much data we're storing in the hash table.
**2.** How many cells are available in the hash table.
**3.** Which hash function we're using.
It makes sense why the first two factors are important. If you have a lot of data to store in only a few cells, there will be many collisions and the hash table will lose its efficiency.

A good hash function, therefore, is one that distributes its data across all available cells.

And this is the balancing act that a hash table must perform. A good hash table strikes a balance of avoiding collisions while not consuming lots of mem-

ory.

To accomplish this, computer scientists have developed the following rule of thumb: **For every 7 data elements stored in a hash table, it should have 10 cells.**
So, if you are planning on storing 14 elements, you'd want to have 20 available cells, and so on and so forth.
This ratio of data to cells is called the **load factor.** Using this terminology, we'd say that the ideal load factor is 0.7 (7 elements / 10 cells).

**Until now, our discussion around data structures has focused primarily on how they affect the performance of various operations. However, having a variety of data structures in your programming arsenal allows you to create code that is simpler and easier to read.**
In this chapter, you're going to discover two new data structures: **Stacks and queues.** The truth is that these two structures are not entirely new. They're simply arrays with restrictions. Yet, these restrictions are exactly what make them so elegant.
More specifically, stacks and queues are elegant tools for handling temporary data. From operating system architecture to printing jobs to traversing data, stacks and queues serve as temporary containers that can be used to form beautiful algorithms.

**Think of temporary data like the food orders in a diner. What each customer ordered is important until the meal is made and delivered; then you throw the order slip away. You don't need to keep that information around. Temporary data is information that doesn't have any meaning after it's processed, so you don't need to keep it around. However, the order in which you process the data can be important - in the diner, you should ideally make each meal in the order in which it was requested. Stacks and queues allow you to handle data in order, and then get rid of it once you don't need it anymore.**

## Stacks

A stack stores data in the same way that arrays do - it's simply a list of elements. The one catch is that stacks have the following three constraints:

• Data can only be **inserted** at the **end of a stack.**
• Data can only be **read** from the **end of a stack.**
• Data can only be removed from the **end of a stack.**

**You can think of a stack as an actual stack of dishes: You can't look at the face of any dish other than the one at the top. Similarly, you can't add any**

**dish except to the top of the stack, nor can you remove any dish besides the one at the top.** (At least, you shouldn't.) In fact, most computer science literature refers to the end of the stack as its top, and the beginning of the stack as its bottom.

Stacks are ideal for processing any data that should be handled in reverse order to how it was received **(LIFO)**. **The "undo" function in a word processor,** or function calls in a networked application are examples of when you'd want to use a stack.

## Queues

A Queue also deals with temporary data elegantly, and is like a stack in that it is an array with restrictions. The difference lies in what order we want to process our data, and this depends on the particular application we're working on.
**You can think of a queue as a line of people at the movie theater.** The first one on the line is the first one to leave the line and enter the theater. With queues, the first item added to the queue is the first item to be removed. That's why computer scientists use the acronym **"FIFO"** when it comes to queues: First In, First Out.

Like stacks, queues are arrays with three restrictions. (It's just a different set of restrictions.)
• Data can only be inserted at the end of a queue. (This is identical behavior as the stack.)
• Data can only be read from the front of a queue. (This is the opposite of behavior of the stack.)
• Data can only be removed from the front of a queue. (This too, is the opposite behavior of the stack.)

While this example is simplified and abstracts away some of the nitty gritty details that a real live printing system may have to deal with, the fundamental use of a queue for such an application is very real and serves as the foundation for building such a system.

**Queues are also the perfect tool for handling asynchronous requests - they ensure that the requests are processed in the order in which they were received.** They are also commonly used to model real-world scenarios where events need to occur in a certain order - such as airplanes waiting for takeoff and patients waiting for their doctor.

**Recursion is official name for when a function calls itself.** While infinite function calls are generally useless - and even dangerous - recursion is a powerful tool that can be harnessed. And when we harness the power of recursion, we can solve particularly tricky problems, as you'll now see.

Before we continue stepping through the code, note how we're using recursion to achieve our goal. We're not using any loop constructs, but by simply having the countdown function call itself, we are able to count down from 10 and print each number to the console.
I**n almost any case in which you can use a loop, you can also use recursion.** Now, just because you can use recursion doesn't mean that you should use recursion. **Recursion is a tool that allows for writing elegant code.** In the above example, the recursive approach is not necessarily any more beautiful or efficient than using a classic for-loop. However, we will soon encounter examples in which recursion shines. In the meantime, let's continue exploring how recursion works.

In Recursion Land (a real place), this case in which the method will not recurse is known as the base case. So in our countdown() function, 0 is the base case.

The computer uses a stack to keep track of which functions it's in the middle of calling. This stack is known, appropriately enough, as the call stack.

While previous examples of the NASA countdown and calculating factorials can be solved with recursion, they can also be easily solved with classical loops. While recursion is interesting, it doesn't really provide an advantage when solving these problems.
However, **recursion is a natural fit in any situation where you find yourself having to repeat an algorithm within the same algorithm. In these cases, recursion can make for more readable code,** as you're about to see.

As you've seen in the filesystem example, recursion is often a great choice for an algorithm in which the algorithm itself doesn't know on the outset how many levels deep into something it needs to dig.
Now that you understand recursion, you've also unlocked a superpower. You're about to encounter some really efficient - yet advanced - algorithms, and many of them rely on the principles of recursion.

**Quicksort** is an extremely fast sorting algorithm that is particulary efficient for average scenarios. While in worst case scenarios (that is, inversely sorted arrays) it performs similarly to Insertion Sort and Selection Sort, it is much faster for average scenarios - which are what occur most of the time.

Partitioning
To partition an array is to take a random value from the array - which is then called the pivot - and make sure that every number that is less than the pivot ends up to the left of the pivot, and that every number that is greater than the pivot will be to the right of the pivot. We accomplish partitioning through a simple algorithm, that will be described in our example below.

When we're done with a partition, we are now assured that all values to the left of the pivot are less than the pivot, and all values to the right of the pivot are greater than it. And that means that the pivot itself is now in its correct place within the array, although the other values are not yet necessarily completely sorted.

**The Quicksort algorithm relies heavily on partitions**.
It works as follows:
1) Partition the array. The pivot is now in its proper place.
2) Treat the subarrays to the left and right of the pivot as their own arrays, and recursively repeat steps #1 and #2. That means that we'll partition each subarray, and end up with even smaller subarrays to the left and right of each subarray's pivot. We then partition those subarrays, and so on and so forth.
3) When we have a subarray that has 0 or 1 elements, that is our base case and we do nothing.

We can see that they have identical worst case scenarios, and that Insertion Sort is actually faster than Quicksort for a best case scenario. However, the reason why Quicksort is so much more superior than Insertion Sort is because of the average scenario - which, again, is what happens most of the time. For average cases, Insertion Sort takes a whopping O(N 2 ), while Quicksort is much faster at O(N log N).

## Quickselect

Let's say that you have an array in random order, and you don't need to sort it, but you do want to know the tenth-lowest value in the array, or the fifth-highest. This can be useful if we had a bunch of test grades and wanted to know what the 25th percentile was, or if we wanted to find the median grade. The obvious way to solve this would be to sort the entire array, and then jump to the appropriate cell.
Even were we to use a fast sorting algorithm like Quicksort, this algorithm takes at least O(N log N) for average cases, and while that isn't bad, we can do even better with a brilliant little algorithm known as Quickselect. Quicks-

elect relies on partitioning just like Quicksort, and **can be thought of as a hybrid of Quicksort and binary search.**
As we've seen earlier in this chapter, after a partition, the pivot value ends up in the appropriate spot in the array. Quickselect takes advantage of this in the following way.

We can then grab the value from the second cell and know with confidence that it's the second lowest value in the entire array. One of the beautiful things about Quickselect is that we can find the correct value without having to sort the entire array.
**With Quicksort, for every time we cut the array in half, we need to re-partition every single cell from the original array, giving us O(N log N). With Quickselect, on the other hand, for every time we cut the array in half, we only need to partition the one half that we care about - the half in which we know our value is to be found.**

For the next several chapters, we are going to explore a variety of d**ata structures that all build upon a single concept - the node.** Node-based data structures offer new ways to organize and access data that provide a number of major performance advantages.
In this chapter, we're going to explore the linked list, which is the simplest node-based data structure and the foundation for the future chapters. You're also going to discover that linked lists seem almost identical to arrays, but come with their own set of tradeoffs in efficiency that can give us a performance boost for certain situations.

## Linked Lists

A linked list is a data structure that represents a list of items, just like an array. In fact, in any application in which you're using an array, you could probably use a linked list instead. Under the hood, however, they are implemented differently, and can have different performance in varying situations.
As mentioned in our first chapter, **memory inside a computer exists as a giant set of cells in which bits of data are stored. When creating an array, your code finds a contiguous group of empty cells in memory and designates them to store data for your application.**

**Linked lists, on the other hand, do not consist of a bunch of memory cells in a row. Instead, it's a bunch of memory cells that are not next to each other, but can be spread across many different cells across the computer's memory. These cells that are not adjacent to each other are known as nodes.**

The big question is: If these nodes are not next to each other, how does the

computer know which nodes are part of the same linked list?
And this is the key to the linked list: **In addition to the data stored within the node, each node also stores the memory address of the next node in the linked List.**

**This extra piece of data - this pointer to the next node's memory address - is known as a link.**

**Each node consists of two memory cells.** The first cell holds the actual **data**, while the second cell serves as a **link** that indicates where in memory the next node begins. **The final node's link is contains null since the linked list ends there.**

For our code to work with a linked list, all it really needs to know is where in memory the first node begins. Since each node contains a link to the next node, if the application is given the first node of the linked list, it can piece together the rest of the list by following the link of the first node to the second node, and the link from the second node to the third node, and so on.
**One advantage of a linked list over an array is that the program doesn't need to find a bunch of empty memory cells in a row to store its data. Instead, the program can store the data across many cells that are not necessarily adjacent to each other.**

## Reading

We noted above that when reading a value from an array, the computer can jump to the appropriate cell in a single step, which is O(1). This is not the case, however, with a linked list.
**If your program wanted to read the value at index 2 of a linked list, the computer cannot look it up in one step, because it doesn't immediately know where to find it in the computer's memory.** After all, each node of a linked list can be anywhere in memory! Instead, all the program knows is the memory address of the first node of the linked list. To find index 2 (which is the third node), the program must begin looking up index 0 of the linked list, and then follow the link at index 0 to index 1. It must then again follow the link at index 1 to index 2, and finally inspect the value at index 2.

If we ask a computer to look up the value at a particular index, **the worst case scenario would be if we're looking up the last index in our list.** In such a case, the computer will take N steps to look up this index, since it needs to start at the first node of the list and follow each link until it reaches the final node. Since Big O Notation expresses the worst case scenario unless explicitly stated otherwise, we'd say that reading a linked list has an efficiency of O(N).

This is a significant disadvantage in comparison with arrays in which reads are O(1).

**Searching**
Arrays and linked lists have the same efficiency for search. Remember, searching is looking for a particular piece of data within the list and getting its index. With both arrays and linked lists, the program needs to start at the first cell and look through each and every cell until it finds the value it's searching for. In a worst case scenario - where the value we're searching for is in the final cell or not in the list altogether - it would take O(N) steps.

**Insertion**
Insertion is one operation in which linked lists can have a distinct advantage over arrays in certain situations. Recall that the worst case scenario for insertion into an array is when the program inserts data into index 0, because it then has to shift the rest of the data one cell to the right, which ends up yielding an efficiency of O(N). With linked lists, however, insertion at the beginning of the list takes just one step - which is O(1).

**In contrast with the array, therefore, a linked list provides the flexibility of inserting data to the front of the list without requiring the shifting of any other data.**

Therefore, inserting into a the middle of a linked list takes O(N), just as it does for an array.
Interestingly, our analysis shows that the best and worst case scenarios for arrays and linked lists are the opposite of one another. That is, inserting at the beginning is great for linked lists, but terrible for arrays. And inserting at the end is an array's best case scenario, but the worst case when it comes to a linked list

**Deletion**

**Deletion is very similar to insertion in terms of efficency. To delete a node from the beginning of a linked list, all we need to do is perform one step: We change the first_node of the linked list to now point to the second node.**

Contrast this with an array in which deleting the first element means shifting all remaining data one cell to the left, an efficiency of O(N).
When it comes to deleting the final node of a linked list, the actual deletion takes one step - we just take the second-to-last node and make its link null . However, it takes N steps to first get to the second-to-last node, since we need to start at the beginning of the list and follow the links until we reach it.

Search, insertion, and deletion seem to be a wash, and reading from an array is much faster than reading from a linked list. If so, why would one ever want to use a linked list?

## Linked Lists in Action

One case where linked lists shine are when we examine a single list and delete many elements from it. **Let's say, for example, that we were building an application that combs through lists of email addresses and removes any email address that has an invalid format.** Our algorithm inspects each and every email address one at a time, and uses a regular expression (a specific pattern for identifying certain types of data) to determine whether the email address is invalid. If it's invalid, we remove it from the list.

## Doubly Linked Lists

Another interesting application of a linked list is that it can be used as the underlying data structure behind a queue. We covered queues in Crafting Elegant Code with Stacks and Queues, and you'll recall that they are lists of items in which data can only be inserted at the end, and removed from the beginning. Back then, we used an array as the basis for the queue, explaining that the queue is simply an array with special constraints. However, we can also use a linked list as the basis for a queue, assuming that we enforce the same constraints of only inserting data at the end and removing data from the beginning. Does using a linked list instead of an array have any advantages? Let's analyze this.

Again, the queue inserts data at the end of the list. As we discussed earlier in this chapter, arrays will be superior when it comes to inserting data, since we'd be able to do so at an efficiency of O(1). Linked lists, on the other hand, would insert data at O(N). So when it comes to insertion, the array would make for a better choice than a linked list.

When it comes to deleting data from a queue, though, linked lists would be faster, since it would be O(1) compared to arrays which delete data from the beginning at O(N).

Based on this analysis, it would seem that it doesn't matter whether we use an array or a linked list, as we'd end up with one major operation that is O(1) and another that is O(N). For arrays, insertion is O(1) and deletion is O(N), and for linked lists, insertion is O(N) and deletion is O(1).

However, if we use a special variant of a linked list called the doubly linked list, we'd be able to insert and delete data from a queue at O(1).

**A doubly linked list is like a linked list, except that each node has two links - one that points to the next node, and one that points to the preceding node. In addition, the doubly linked list keeps track of both the first and last nodes.**

Because doubly linked lists have immediate access to both the front and end of the list, they can insert data on either side at O(1) as well as delete data on either side at O(1). Since doubly linked lists can insert data at the end in O(1) time and delete data from the front in O(1) time, they make the perfect underlying data structure for a queue.

## Binary Trees

We were introduced to node-based data structures in the previous chapter using linked lists. In a simple linked list, each node contains a link that connects this node to a single other node. **A tree is also a node-based data structure, but within a tree, each node can have links to multiple nodes.**

Trees come with their own unique nomenclature:

• The uppermost node (in our example, the “j”) is called the **root.** Yes, in our picture the root is at the top of the tree; deal with it.
• In our example, we’d say that the “j” is a **parent** to “m” and “b”, which are in turn **children** of “j”. The “m” is a parent of “q” and “z”, which are in turn children of “m”.
• Trees are said to have **levels.**

There are many different kinds of tree-based data structures, but in this chapter we’ll be focusing on particular tree known as a binary tree.

A binary tree is a tree that abides by the following **rules:**

• Each node has either **zero, one, or two children.**
• **If a node has two children, it must have one child that has a lesser value than the parent, and one child that has a greater value than the parent.**

Note that each node has one child with a lesser value than itself which is depicted using a left arrow, and one child with a greater value than itself which is depicted using a right arrow.

### Searching

More generally, we’d say that searching in a binary tree is O(log N). This is because each step we take eliminates half of the remaining possible values

in which our value can be stored. (We'll see soon, though, that this is only for perfectly balanced binary tree, which is a best case scenario.)

**Insertion**

Insertion always takes just one extra step beyond a search, which means that insertion takes log N + 1 steps, which is O(log N) as Big O ignores constants.

Because of this, if you ever wanted to convert an ordered array into a binary tree, you'd better first randomize the order of the data.
It emerges that in a worst case scenario, where a tree is completely imbalanced, search is O(N). In a best case scenario, where it is perfectly balanced, search is O(log N). In the typical scenario, in which data is inserted in random order, a tree will be pretty well balanced and search will take about O(log N).

**Deletion**
Deletion is the least straightforward operation within a binary tree, and requires some careful maneuvering.

Pulling all the steps together, the algorithm for deletion from a binary tree is:
• If the node being deleted has no children, simply delete it.
• If the node being deleted has one child, delete it and plug the child into the spot where the deleted node was.
• When deleting a node with two children, replace the deleted node with the successor node. The successor node is the child node whose value is the least of all values that are greater than the deleted node.
• If the successor node has a right child, after plugging the successor node into the spot of the deleted node, take the right child of the successor node and turn it into the left child of the parent of the successor node.

Like search and insertion, deleting from trees is also typically O(log N). This is because deletion requires a search plus a few extra steps to deal with any hanging children. Contrast this with deleting a value from an ordered array, which takes O(N) due to shifting elements to the left to close the gap of the deleted value.

**The process of visiting every node in a data structure is known as traversing the data Structure.**

Binary trees are a powerful node-based data structure that provides order maintenance, while also offering fast search, insertion, and deletion. They're more complex than their linked list cousins, but offer tremendous value.

It is worth mentioning that in addition to binary trees, there are many other types of tree-based data structures as well. **Heaps, B-trees, Red-Black Trees, and 2-3-4 Trees** as well as many other trees each have their own use for specialized scenarios.

## Graphs

A graph is a data structure than specializes in **relationships**, as it easily conveys how data is connected.

Each person is represented by a node, and each line indicates a friendship with another person. In graph jargon, each node is called a **vertex**, and each line is called an **edge**. Vertices that are connected by an edge are said to be adjacent to each other.

While the Facebook and Twitter examples are similar, the nature of the relationships in each example are different. Because relationships in Twitter are one-directional, we use arrows in our visual implementation, and such a graph is known as a directed graph. In Facebook, where the relationships are mutual and we use simple lines, the graph is called a non-directed graph.

Now, each vertex ends up being removed from the queue once. In Big O Notation this is called O(V). That is, for V vertices in the graph, there are V removals from the queue.
Why don't we simply call this O(N) - with N being the number of vertices? The answer is because in this (and many graph algorithms) we also have additional steps that process not just the vertices themselves, but also the edges, as we'll now explain.

So, for E edges, we check adjacent vertices 2E times. That is, for E edges in the graph, we check twice that number of adjacent vertices. However, since Big O ignores constants, we just write it as O(E).

## Space Constraints

Throughout this book, when analyzing the efficiency of various algorithms, we've focused exclusively on their time complexity - that is, how fast they run. There are situations, however, where we need to measure algorithm efficiency by another measure known as **space complexity**, which is how much memory an algorithm consumes.
**Space complexity becomes an important factor when memory is limited.** This can happen when programming for small hardware devices that only contain a relatively small amount of memory or when dealing with very large amounts of data, which can quickly fill even large memory containers.

In a perfect world, we'd always use algorithms that are both quick and consume a small amount of memory. However, there are times where we're faced with choosing betwen the speedy algorithm or the memory-efficient algorithm - and we need to look carefully at the situation to make the right choice.

Big O can similarly be used to describe how much space an algorithm takes up: For N elements of data, an algorithm consumes a relative number of additional data elements in memory,

Note that this graph is identical to the way we've depicted O(N) in the graphs from previous chapters, with the exception that the vertical axis now represents memory rather than speed.

Since this function does not consume any memory in addition to the original array, we'd describe the space complexity of this function as being O(1). Remember that by time complexity, O(1) represents that the speed of an algorithm is constant no matter how large the data. Similarly, by space complexity, O(1) means that the memory consumed by an algorithm is constant no matter how large the data.

It's important to reiterate that in this book, we judge the space complexity based on additional memory consumed - known as auxiliary space - meaning that we don't count the original data. Even in this second version, we do have an input of array , which contains N elements of memory. However, since this function does not consume any memory in addition to the original array, it is O(1).

**Like all technology decisions, when there are tradeoffs, we need to look at the big picture.**

I hope that you also take away from this book that topics like these - which seem so complex and esoteric - are just a combination of simpler, easier concepts that are within your grasp. Don't be intimidated by resources that make a concept seem difficult simply because they don't explain it well - you can always find a resource that explains it better.
The topic of data structures and algorithms is broad and deep, and we've only just scratched the surface. There is so much more to learn - but with the foundation that you now have, you'll be able to do so. Good luck!