CHAPTER 1

Why Data Structures Matter

When people first learn to code, their focus is—and *should* be—on getting their code to run properly. Their code is measured using one simple metric: does the code actually work?

As software engineers gain more experience, though, they begin to learn about additional layers and nuances regarding the *quality* of their code. They learn that there can be two snippets of code that both accomplish the same task, but that one snippet is *better* than the other.

There are numerous measures of code quality. One important measure is code maintainability. Maintainability of code involves aspects such as the readability, organization, and modularity of one's code.

However, there's another aspect of high-quality code, and that is code *efficiency*. For example, you can have two code snippets that both achieve the same goal, but one *runs faster than the other*.

Take a look at these two functions, both of which print all the even numbers from 2 to 100:

```
def print_numbers_version_one():
  number = 2

  while number <= 100:
    # If number is even, print it:
    if number % 2 == 0:
      print(number)

    number += 1
```

```
def print_numbers_version_two():
  number = 2

  while number <= 100:
    print(number)

    # Increase number by 2, which, by definition,
    # is the next even number:
    number += 2
```

Which of these functions do you think runs faster?

If you said Version 2, you're right. This is because Version 1 ends up looping 100 times, while Version 2 only loops 50 times. The first version then, takes twice as many steps as the second version.

This book is about writing *efficient* code. Having the ability to write code that runs quickly is an important aspect of becoming a better software developer.

The first step in writing fast code is to understand what data structures are and how different data structures can affect the speed of our code. So, let's dive in.

Data Structures

Let's talk about data.

Data is a broad term that refers to all types of information, down to the most basic numbers and strings. In the simple but classic "Hello World!" program, the string "Hello World!" is a piece of data. In fact, even the most complex pieces of data usually break down into a bunch of numbers and strings.

Data structures refer to how data is *organized*. You're going to learn how the same data can be organized in a variety of ways.

Let's look at the following code:

```
x = "Hello! "
y = "How are you "
z = "today?"

print x + y + z
```

This simple program deals with three pieces of data, outputting three strings to make one coherent message. If we were to describe how the data is organized in this program, we'd say that we have three independent strings, each contained within a single variable.

However, this same data can also be stored in an array:

```
array = ["Hello! ", "How are you ", "today?"]

print array[0] + array[1] + array[2]
```

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.* Depending on how you choose to organize your data, your program may run faster or slower by orders of magnitude. And if you're building a program that needs to deal with lots of data, or a web app used by thousands of people simultaneously, the data structures you select may affect whether your software runs at all, or simply conks out because it can't handle the load.

When you have a solid grasp on data structures' performance implications on the software you are creating, you will have the keys to write fast and elegant code, and your expertise as a software engineer will be greatly enhanced.

In this chapter, we're going to begin our analysis of two data structures: arrays and sets. While the two data structures may seem almost identical, you're going to learn the tools to analyze the performance implications of each choice.

The Array: The Foundational Data Structure

The *array* is one of the most basic data structures in computer science. I assume you have worked with arrays before, so you are aware that an array is a list of data elements. The array is versatile, and can serve as a useful tool in many situations, but let's take a look at one quick example.

If you are looking at the source code for an application that allows users to create and use shopping lists for the grocery store, you might find code like this:

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

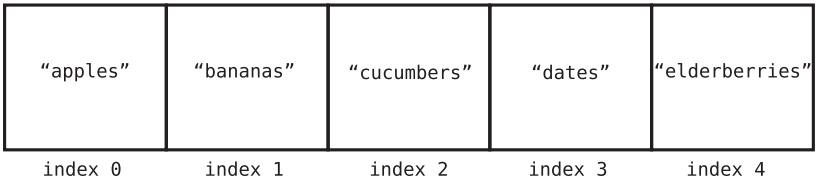
This array happens to contain five strings, each representing something that I might buy at the supermarket. (You've *got* to try elderberries.)

Arrays come with their own technical jargon.

The *size* of an array is how many data elements the array holds. Our grocery list array has a size of 5, since it contains five values.

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

In most programming languages, we begin counting the index at 0. So, for our example array, "apples" is at index 0, and "elderberries" is at index 4, like this:



Data Structure Operations

To understand the performance of any data structure—such as the array—we need to analyze the common ways our code might interact with that data structure.

Many data structures are used in four basic ways, which we refer to as *operations*. These operations are:

- *Read:* Reading refers to looking something up at a particular spot within the data structure. With an array, this means looking up a value at a particular index. For example, looking up which grocery item is located at index 2 would be *reading* from the array.
- *Search:* Searching refers to looking for a particular value within a data structure. With an array, this means looking to see if a particular value exists within the array, and if so, at which index. For example, looking up the index of "dates" in our grocery list would be *searching* the array.
- *Insert:* Insertion refers to adding a new value to our data structure. With an array, this means adding a new value to an additional slot within the array. If we were to add "figs" to our shopping list, we'd be *inserting* a new value into the array.
- *Delete:* Deletion refers to removing a value from our data structure. With an array, this means removing one of the values from the array. For example, if we removed "bananas" from our grocery list, this value would be *deleted* from the array.

In this chapter, we'll analyze how fast each of these operations are when applied to an array.

Measuring Speed

So, how do we measure the speed of an operation?

If you take away just one thing from this book, let it be this: 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've actually seen this earlier in the context of printing the even numbers from 2 to 100. The second version of that function was faster because it took half as many steps as the first version did.

Why do we measure code's speed in terms of steps?

We do this because we can never say definitively that any operation takes, say, five seconds. While a piece of code may take five seconds on a particular computer, that same piece of code may take longer on an older piece of hardware. For that matter, that same code might run much faster on the supercomputers of tomorrow. Measuring the speed of an operation in terms of time is undependable, since the time will always change depending on the hardware it is run on.

However, we *can* measure the speed of an operation in terms of how many computational *steps* it takes. If Operation A takes 5 steps, and Operation B takes 500 steps, we can assume that Operation A will always be faster than Operation B on *all* pieces of hardware. Measuring the number of steps is, therefore, the key to analyzing the speed of an operation.

Measuring the speed of an operation is also known as measuring its *time complexity.* Throughout this book, I'll use the terms *speed, time complexity, efficiency, performance,* and *runtime* interchangeably. They all refer to the number of steps a given operation takes.

Let's jump into the four operations of an array and determine how many steps each one takes.

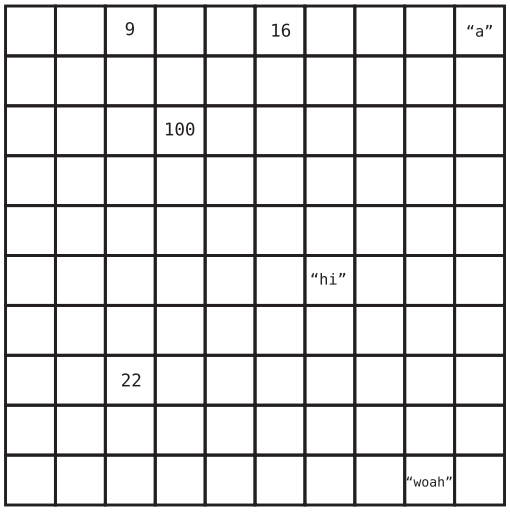
Reading

The first operation we'll look at is *reading,* which looks up what value is contained at a particular index inside the array.

A computer can read from an array in just one step. This is because the computer has the ability to jump to any particular index in the array and peer inside. In our example of `["apples", "bananas", "cucumbers", "dates", "elderberries"]`, if we looked up index 2, the computer would jump right to index 2 and report that it contains the value `"cucumbers"`.

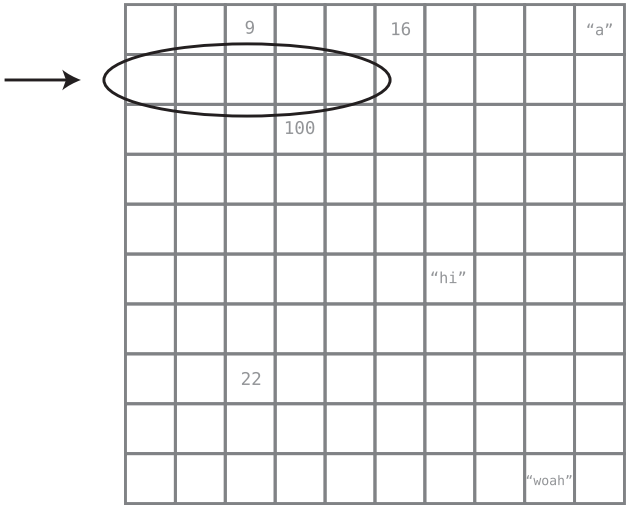
How is the computer able to look up an array's index in just one step? Let's see how.

A computer's memory can be viewed as a giant collection of cells. In the following diagram, you can see a grid of cells in which some are empty and some contain bits of data:



While this visual is a simplification of how computer memory works under the hood, it represents the essential idea.

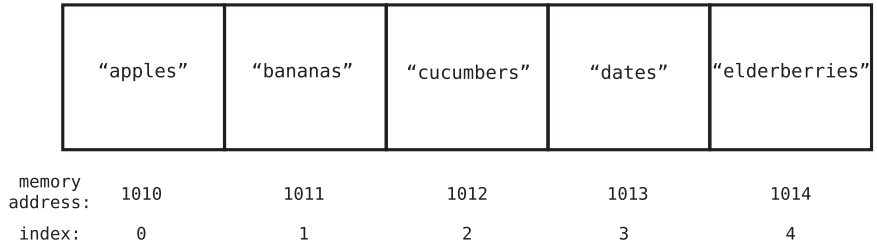
When a program declares an array, it allocates a contiguous set of empty cells for use in the program. So, if you were creating an array meant to hold five elements, your computer would find a group of five empty cells in a row and designate it to serve as your array:



Now, every cell in a computer's memory has a specific address. It's sort of like a street address (for example, 123 Main St.), except that it's represented with a number. Each cell's memory address is one number greater than the previous cell's address. Here's a visual that shows each cell's memory address:

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

In the next diagram, you can see our shopping list array with its indexes and memory addresses:



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

1. A computer can jump to any *memory address* in one step. For example, if you asked a computer to inspect whatever's at memory address 1063, it can access that without having to perform any search process. As an analogy, if I ask you to raise your right pinky finger, you wouldn't have to search all your fingers to find which one is your right pinky. You'd be able to identify it immediately.

2. Whenever a computer allocates an array, it also makes note at which memory address the array *begins*. So, if we asked the computer to find the first element of the array, it would be able to instantly jump to the appropriate memory address to find it.

Now, these facts explain how the computer can find the *first* value of an array in a single step. However, a computer can also find the value at *any* index by performing simple addition. If we asked the computer to find the value at index 3, the computer would simply take the memory address at index 0 and add 3. (Memory addresses are sequential, after all.)

Let's apply this to our grocery list array. Our example array begins at memory address 1010. So, if we told the computer to read the value at index 3, the computer would go through the following thought process:

1. The array begins with index 0, which is at memory address 1010.
2. Index 3 will be exactly three slots past index 0.
3. By logical extension, index 3 would be located at memory address 1013, since 1010 + 3 is 1013.

Once the computer knows that index 3 is at memory address 1013, it can jump right there and see that it contains the value `"dates"`.

Reading from an array is, therefore, an efficient operation, since the computer can read any index by jumping to any memory address in one step. Although I described the computer's thought process by breaking it down into three parts, we are currently focusing on the main step of the computer jumping to a memory address. (In later chapters, we'll explore how to know which steps are the ones worth focusing on.)

Naturally, an operation that takes just one step is the fastest type of operation. Besides being a foundational data structure, arrays are also a very powerful data structure because we can read from them with such speed.

Now, what if instead of asking the computer what value is contained at index 3, we flipped the question around and asked at what index `"dates"` can be found? That is the search operation, and we'll explore that next.

Searching

As I stated previously, *searching* an array means looking to see whether a particular value exists within an array and if so, at which index it's located.

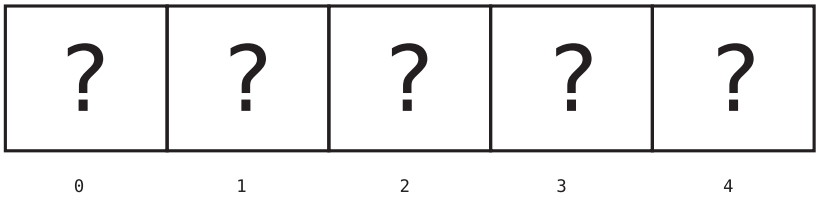
In a sense, it's the inverse of reading. Reading means providing the computer an *index* and asking it to return the value contained there. Searching, on the

other hand, means providing the computer a *value* and asking it to return the index of that value's location.

While these two operations sound similar, there's a world of difference between them when it comes to efficiency. Reading from an index is fast, since a computer can jump immediately to any index and discover the value contained there. Searching, though, is tedious, since the computer has no way to jump to a particular value.

This is an important fact about computers: a computer has immediate access to all of its memory addresses, but it has no idea offhand what *values* are contained at each memory address.

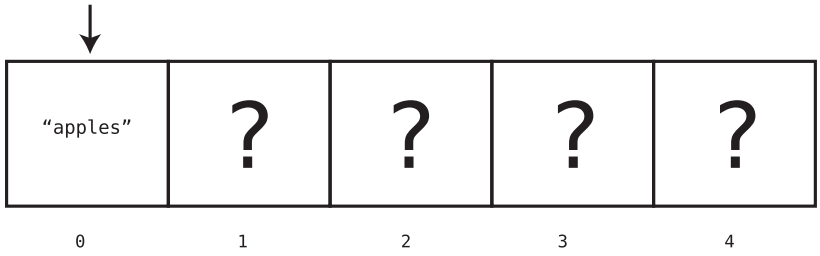
Let's take our earlier array of fruits and veggies, for example. The computer can't immediately see the actual contents of each cell. To the computer, the array looks something like this:



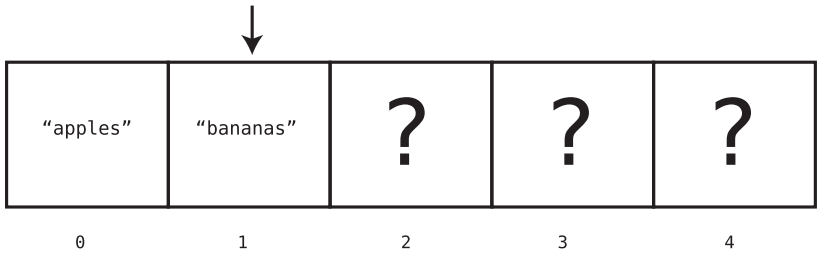
To search for a fruit within the array, the computer has no choice but to inspect each cell one at a time.

The following diagrams demonstrate the process the computer would use to search for `"dates"` within our array.

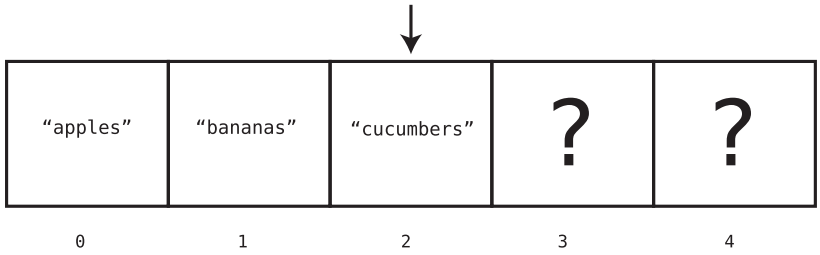
First, the computer checks index 0:



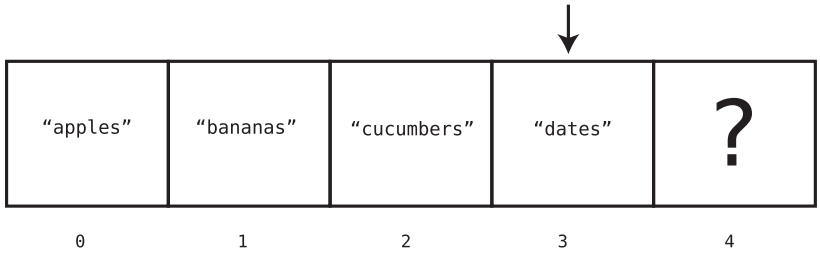
Since the value at index 0 is `"apples"`, and not the `"dates"` we're looking for, the computer moves on to the next index as shown in the diagram on page 10.



Since index 1 doesn't contain the "dates" we're looking for either, the computer moves on to index 2:



Once again, we're out of luck, so the computer moves to the next cell:



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

In this example, because the computer had to check four different cells until it found the value we were searching for, we'd say that this particular operation took a total of four steps.

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

Now, what is the *maximum* number of steps a computer would need to perform to conduct a linear search on an array?

If the value we're seeking happens to be in the final cell in the array (like "elderberries"), then the computer would end up searching through *every* cell of the

array until it finally finds the value it's looking for. Also, if the value we're looking for doesn't occur in the array at all, the computer likewise would have to search every cell so that it can be sure the value doesn't exist within the array.

So, it turns out that for an array of five cells, the maximum number of steps linear search would take is five. For an array of 500 cells, the maximum number of steps linear search would take is 500.

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

In any case, it's clear that searching is less efficient than reading, since searching can take many steps, while reading always takes just one step no matter the size of the array.

Next, we'll analyze the operation of insertion.

Insertion

The efficiency of inserting a new piece of data into an array depends on *where* within the array you're inserting it.

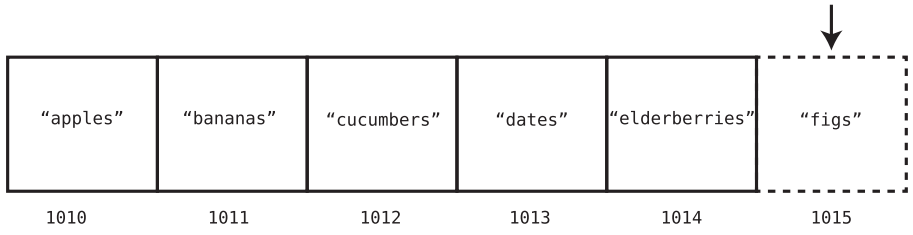
Let's say we want to add "figs" to the end of our shopping list. Such an insertion takes just one step.

This is true due to another fact about computers: when allocating an array, the computer always keeps track of the array's size.

When we couple this with the fact that the computer also knows at which memory address the array begins, computing the memory address of the last item of the array is a cinch: if the array begins at memory address 1010 and is of size 5, that means its final memory address is 1014. So, to insert an item beyond that would mean adding it to the *next* memory address, which is 1015.

Once the computer calculates which memory address to insert the new value into, it can do so in one step.

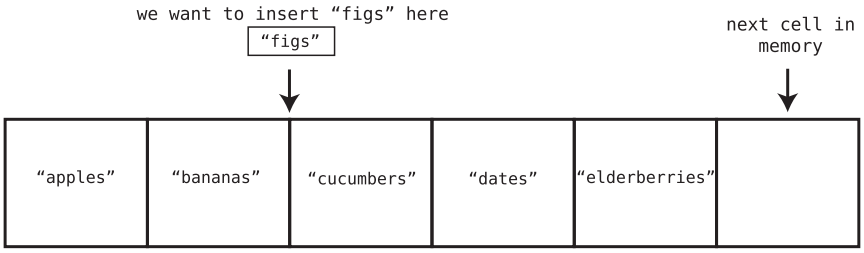
This is what inserting "figs" at the end of the array looks like:



But there's one hitch. Because the computer initially allocated only five cells in memory for the array, and now we're adding a sixth element, the computer may have to allocate additional cells toward this array. In many programming languages, this is done under the hood automatically, but each language handles this differently, so I won't get into the details of it.

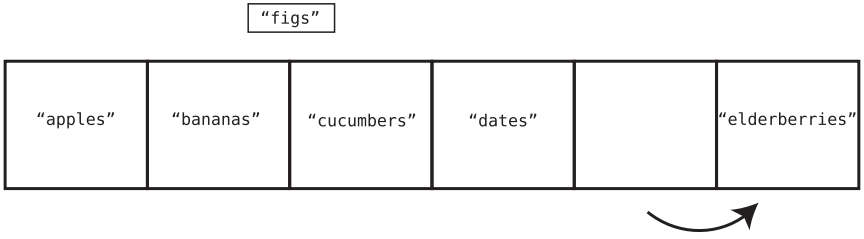
We've dealt with insertions at the end of an array, but inserting a new piece of data at the *beginning* or in the *middle* of an array is a different story. In these cases, we need to *shift* pieces of data to make room for what we're inserting, leading to additional steps.

For example, let's say we want to add "figs" to index 2 within the array. Take a look at the following diagram:

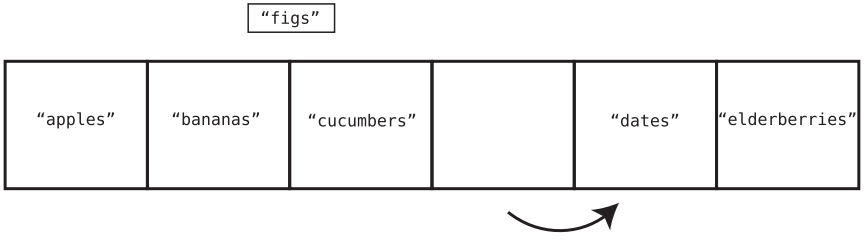


To do this, we need to move "cucumbers", "dates", and "elderberries" to the right to make room for "figs". This takes multiple steps, since we need to first move "elderberries" one cell to the right to make room to move "dates". We then need to move "dates" to make room for "cucumbers". Let's walk through this process.

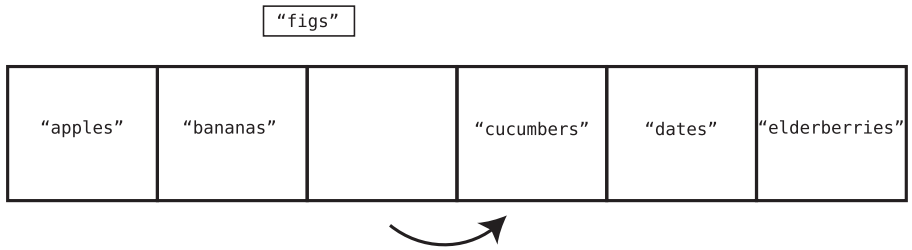
Step 1: We move "elderberries" to the right:



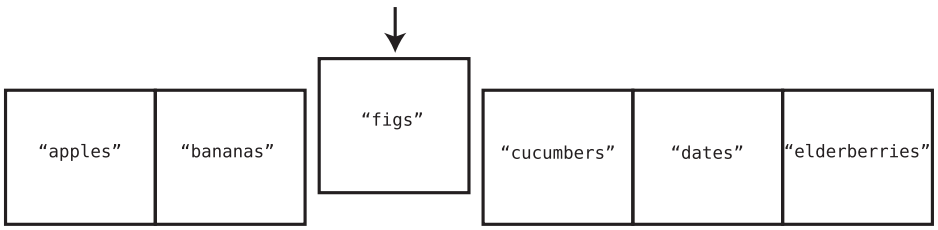
Step 2: We now move "dates" to the right:



Step 3: We now move "cucumbers" to the right:



Step 4: Finally, we can insert "figs" into index 2:



Notice that in the preceding example, insertion took four steps. Three of the steps involved shifting data to the right, while one step involved the actual insertion of the new value.

The worst-case scenario for insertion into an array—that is, the scenario in which insertion takes the most steps—is when we insert data at the *beginning* of the array. This is because when inserting at the beginning of the array, we have to move *all* the other values one cell to the right.

We can say that insertion in a worst-case scenario can take *N + 1 steps* for an array containing N elements. This is because we need to shift all N elements over, and then finally execute the actual insertion step.

Now that we've covered insertion, we're up to the array's final operation: deletion.

Deletion

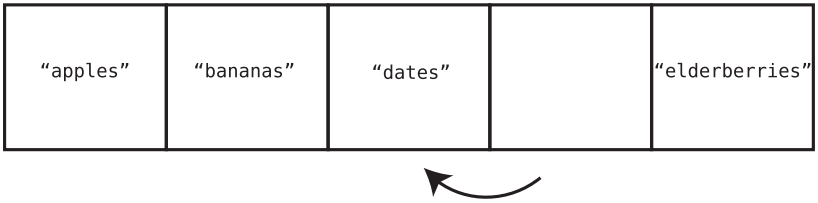
Deletion from an array is the process of eliminating the value at a particular index.

Let's return to our original example array and delete the value at index 2. In our example, this value is "cucumbers".

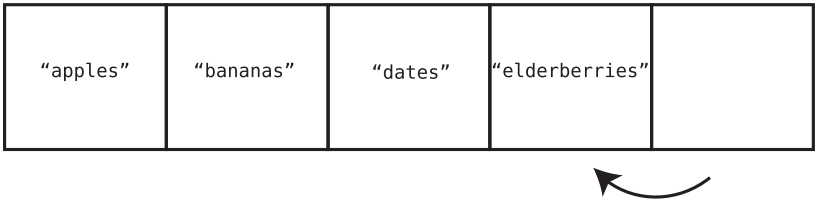
Step 1: We delete "cucumbers" from the array:

While the actual deletion of "cucumbers" technically took just one step, we now have a problem: we have an empty cell sitting smack in the middle of our array. An array is not effective when there are gaps in the middle of it, so to resolve this issue, we need to shift "dates" and "elderberries" to the left. This means our deletion process requires additional steps.

Step 2: We shift "dates" to the left:



Step 3: We shift "elderberries" to the left:



It turns out that for this deletion, the entire operation took three steps. The first step involved the actual deletion, and the other two steps involved data shifts to close the gap.

Like insertion, the worst-case scenario of deleting an element is deleting the very first element of the array. This is because index 0 would become empty, and we'd have to shift *all* the remaining elements to the left to fill the gap.

For an array of five elements, we'd spend one step deleting the first element, and four steps shifting the four remaining elements. For an array of 500 elements, we'd spend one step deleting the first element, and 499 steps shifting

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

Congratulations! We've analyzed the time complexity of our first data structure. Now that you've learned how to analyze a data structure's efficiency, you can now discover how different data structures have different efficiencies. This is crucial, because choosing the correct data structure for your code can have serious ramifications on your software's performance.

The next data structure—the *set*—seems so similar to the array at first glance. However, you'll see that the operations performed on arrays and sets have different efficiencies.

Sets: How a Single Rule Can Affect Efficiency

Let's explore another data structure: the *set*. A set is a data structure that does not allow duplicate values to be contained within it.

There are different types of sets, but for this discussion, I'll talk about an *array-based set*. This set is just like an array—it is a simple list of values. The only difference between this set and a classic array is that the set never allows duplicate values to be inserted into it.

For example, if you had the set `["a", "b", "c"]` and tried to add another `"b"`, the computer just wouldn't allow it, since a `"b"` already exists within the set.

Sets are useful when you need to ensure that you don't have duplicate data.

For instance, if you're creating an online phone book, you don't want the same phone number appearing twice. In fact, I'm currently suffering from this with my local phone book: my home phone number is not just listed for me, but also it is erroneously listed as the phone number for some family named Zirkind. (Yes, this is a true story.) Let me tell you—it's quite annoying to receive phone calls and voicemail from people looking for the Zirkinds. For that matter, I'm sure the Zirkinds are also wondering why no one ever calls them. And when I call the Zirkinds to let them know about the error, my wife picks up the phone because I've called my own number. (Okay, that last part never happened.) If only the program that produced the phone book had used a set…

In any case, an array-based set is an array with one additional constraint of barring duplicates. While not allowing duplicates is a useful feature, this simple constraint also causes the set to have a *different efficiency* for one of the four primary operations.

Let's analyze the reading, searching, insertion, and deletion operations in the context of an array-based set.

Reading from a set is exactly the same as reading from an array—it takes just one step for the computer to look up what's contained within a particular index. As I described earlier, this is because the computer can jump to any index within the set since it can easily calculate and jump to its memory address.

Searching a set also turns out to be no different than searching an array—it takes up to N steps to search for a value within a set. And deletion is also identical between a set and an array—it takes up to N steps to delete a value and move data to the left to close the gap.

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. We saw that with an array, the computer can insert a value at its end in a single step.

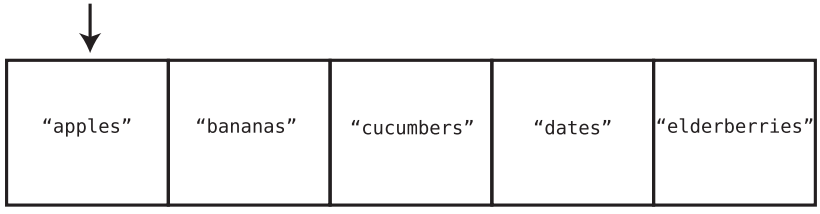
With a set, however, the computer first needs to determine that this value doesn't already exist in this set—because that's what sets do: they prevent duplicate data from being inserted into them.

Now, how will the computer ensure that the new data isn't already contained in the set? Remember, a computer doesn't know offhand what values are contained within the cells of an array or set. Because of this, the computer will first need to *search* the set to see whether the value we want to insert is already there. Only if the set does not yet contain our new value will the computer allow the insertion to take place.

So, every insertion into a set *first requires a search.*

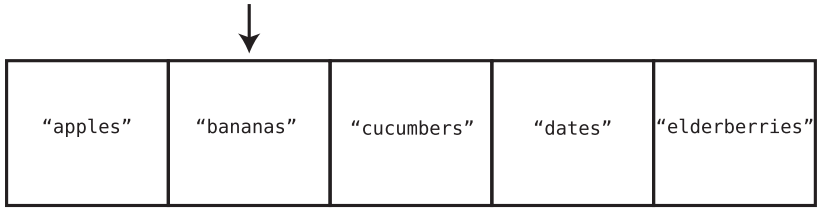
Let's see this in action with an example. Imagine our grocery list from earlier was stored as a set—which would be a decent choice since we don't want to buy the same thing twice, after all. If our current set is `["apples", "bananas", "cucumbers", "dates", "elderberries"]`, and we want to insert `"figs"` into the set, the computer must execute the following steps, beginning with a search for `"figs"`.

Step 1: Search index 0 for `"figs"`:

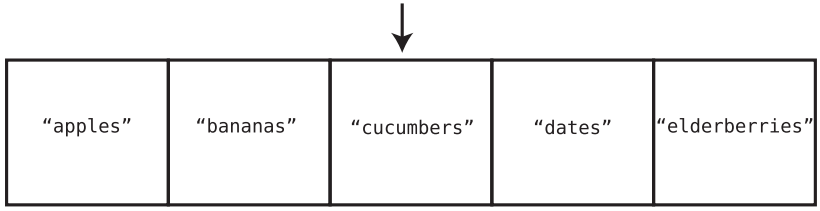


It's not there, but it might be somewhere else in the set. We need to make sure "figs" does not exist anywhere before we can insert it.

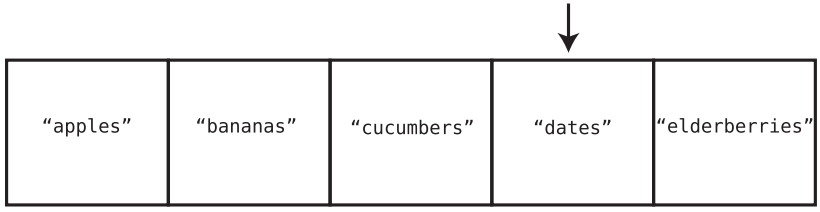
Step 2: Search index 1:



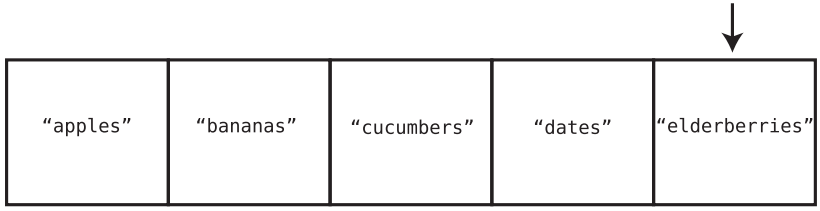
Step 3: Search index 2:



Step 4: Search index 3:

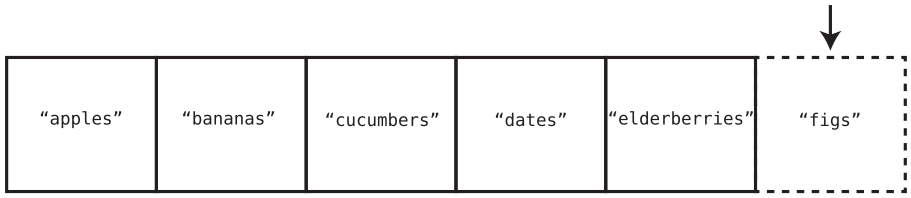


Step 5: Search index 4:



Now that we've searched the entire set, we know with certainty that it doesn't already contain "figs". At this point, it's safe to complete the insertion. And that brings us to our final step:

Step 6: Insert "figs" at the end of the set:



Inserting a value at the end of a set is the best-case scenario, but we still had to perform six steps for a set originally containing five elements. That is, we had to search all five elements before performing the final insertion step.

Said another way: insertion into the end of a set will take up to 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 one step for the actual insertion. Contrast this with the regular array, in which such an insertion takes a grand total of one step.

In the 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, 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. Contrast this to insertion into the beginning of a regular array, which only takes N + 1 steps.

Now, does this mean you should avoid sets just because insertion is slower for sets than regular arrays? Absolutely not. Sets are important when you need to ensure that there is no duplicate data. (Hopefully, one day my phone book will be fixed.) But when you don't have such a need, an array may be preferable, since insertions for arrays are more efficient than insertions for sets. You must analyze the needs of your own application and decide which data structure is a better fit.

Wrapping Up

Analyzing the number of steps an operation takes is the heart of understanding the performance of data structures. Choosing the right data structure for your program can spell the difference between bearing a heavy load versus collapsing under it. In this chapter, you've learned to use this analysis to weigh whether an array or a set might be the appropriate choice for a given application.

Now that you've begun to learn how to think about the time complexity of data structures, we can use the same analysis to compare competing algorithms (even within the *same* data structure) to ensure the ultimate speed and performance of our code. And that's exactly what the next chapter is about.

Exercises

The following exercises provide you with the opportunity to practice with arrays. The solutions to these exercises are found in the section, Chapter 1, on page 439.

1. For an array containing 100 elements, provide the number of steps the following operations would take:
   a. Reading
   b. Searching for a value not contained within the array
   c. Insertion at the beginning of the array
   d. Insertion at the end of the array
   e. Deletion at the beginning of the array
   f. Deletion at the end of the array
2. For an array-based set containing 100 elements, provide the number of steps the following operations would take:
   a. Reading
   b. Searching for a value not contained within the array
   c. Insertion of a new value at the beginning of the set
   d. Insertion of a new value at the end of the set
   e. Deletion at the beginning of the set
   f. Deletion at the end of the set
3. Normally the search operation in an array looks for the first instance of a given value. But sometimes we may want to look for *every* instance of a given value. For example, say we want to count how many times the value "apple" is found inside an array. How many steps would it take to find all the "apples"? Give your answer in terms of N.