CSE 250: Data Structures

Course Reference

Chapter 0 *Contents*

Chapter 1. *Introduction*

Data Structures classes often have a mixed reception. Students frequently say that it's hard to see how concepts from data structures get deployed into practice. They complain that the class doesn't teach them to solve specific problems. To some extent, that's true. This class is less about learning how to build the next AI, data management system, or website. Instead, this class is about giving you a set of simple tools that you'll be able to reach for, no matter what great things you end up doing. In short, this class will introduce you to the hammers, wrenches, and screwdrivers of computer science. We'll show you how to decide whether a phillips-head or a flathead screwdriver is better for your use case; when to use a socket wrench or a crescent wrench.

A bit less metaphorically, this class will teach you to think about different data management use cases (called abstract data types), and give you a set of data organization strategies (called data structures) suitable for each. We'll also introduce asymptotic complexity, a way to quickly summarize the performance characteristics of data structures (and a tool for thinking about algorithms). Finally, we'll nudge you to start thinking about code a bit more formally, less as a sequence of instructions, and more in terms of the goals those instructions are trying to achieve.

1.1. What should you get out of this Data Structures class?

This data structures class is fundamentally a math class, where the math will make you into a better programmer. On the subject of learning math, Terrence Tao[1] observes that, while learning to think rigorously is an important step in developing the discipline needed to avoid common logic errors, understanding the underlying intuition is critical too. Along these lines, our goal in this class is both to help you develop the mathematical rigor and discipline to reason about your code, as well as to develop an intuition for how data organization impacts your code's runtime.

1.1.1. An intuition for data structures

Throughout the book (and class), we'll try to be precise and formal when talking about course material. That said, we're less interested in you learning the precise formalism, and more interested in you developing an intuition about what the formalism represents. It's possible (even likely) that after leaving this class, you will never again consciously think about the fact that prepending to a linked list is $O(1)$. If so, that's fine. What we care about is …

- … that you develop an instinct for which data structure is right for a given situation.
- … that you get a little cringe in the back of your brain when you see a method with an $O(N)$ complexity.
- … that you get a little cringe in the back of your brain when you see a doubly nested loop, or another piece of $O(N^2)$ or worse code.

1.1.2. Practice with formal proofs and recursion

By the time you take this class, you should have already taken a discrete math class and gotten your first exposure to proofs and recursive thinking. This class is intended to develop those same skills further:

- We'll review an assortment of proofs regarding algorithm runtimes using specific data structures.
- We'll discuss specific strategies you can use while proving things.

[1] https://terrytao.wordpress.com/career-advice/theres-more-to-mathematics-than-rigour-and-proofs/

- We'll review recursion and discuss approaches to identifying problems that can be solved through recursion and how to use recursion as a problem solving technique.

Apart from preparing you for subsequent theory-oriented classes, our goal here is to give you some tools that you can use to think critically about code that you write, and that you need to debug. If, in five years, you completely forget how to prove that quick sort is Expected-$O(N \log N)$, it'll make us sad, but we'll understand. Instead, we hope that you'll walk away from the class with the instinct to write down invariants for code that you're trying to write or debug.

1.2. What this class is not.

In contrast to many data structures classes, which introduce C programming, memory management, and other related concepts, UB's 250 is intended as a concepts/theory-style class. We will use code. We will spend some time talking about the mechanics of how a computer runs that code.

We'll provide lots of example code in Java (or some cases Python), and we'll make extensive use of Java's class inheritance model. These examples are there to motivate concepts that you will learn throughout the class, or to make the concepts a bit more precise and concrete. However, we assume that you already know how to program in a major object-oriented language (like Java or Python). This is not a class to learn to program; We're here to teach you asymptotic analysis, data structures (ignoring language details where possible), and proof techniques.

1.3. A word on Lies and Trickery

"All models are wrong, but some are useful"

This is an introductory text. You should expect that many of the things we say are simplified for the purposes of presentation. Some things we say and write (e.g., all array accesses are constant-time) will be outright lies (at least for modern computers). However, we make these simplifications intentionally, because it's far easier to grok the simpler model of code and data organization, and because the simpler model is a reasonable approximation (up to a point).

We'll add footnotes that highlight some of the more blatant lies, and hint at some of the nuanced details. In a few cases (e.g., constant time array accesses), we'll also walk back the approximation a bit later in the book. That said, these footnotes are primarily present for the pedantic and the curious. You should still be able to understand the rest of the book even if you ignore every single footnote in the text.

Chapter 2. *Math Refresher*

Date
Text 1
Text 2
2024-06-08
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore.
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Table 1: Journal from 2024-06-08

Chapter 3. *Asymptotic Runtime Complexity*

Data Structures are the fundamentals of algorithms: How efficient an algorithm is depends on how the data is organized. Think about your workspace: If you know exactly where everything is, you can get to it much faster than if everything is piled up randomly. Data structures are the same: If we organize data in a way that meets the need of an algorithm, the algorithm will run much faster (remember the array vs linked list comparison from earlier)?

Since the point of knowing data structures is to make your algorithms faster, one of the things we'll need to talk about is how fast the data structure (and algorithm) is for specific tasks. Unfortunately, "how fast" is a bit of a nuanced comparison. I could time how long algorithms **A** and **B** take to run, but what makes a fair comparison depends on a lot of factors:

- How big is the data that the algorithm is running on?
  - **A** might be faster on small inputs, while **B** might be faster on big inputs.
- What computer is running the algorithm?
  - **A** might be much faster on one computer, **B** might be much faster on a network of computers.
  - **A** might be especially tailored to Intel x86 CPUs, while **B** might be tailored to the non-uniform memory latencies of AMD x86 CPUs.
- How optimized is the code?
  - Hand-coded optimizations can account for multiple orders of magnitude in algorithm performance.

In short, comparing two algorithms requires a lot of careful analysis and experimentation. This is important, but as computer scientists, it can also help to take a more abstract view. We would like to have a shorthand that we can use to quickly convey the 50,000-ft view of "how fast" the algorithm is going to be. That shorthand is asymptotic runtime complexity.

3.1. Why is Asymptotic Analysis important?

Try the following code in python:

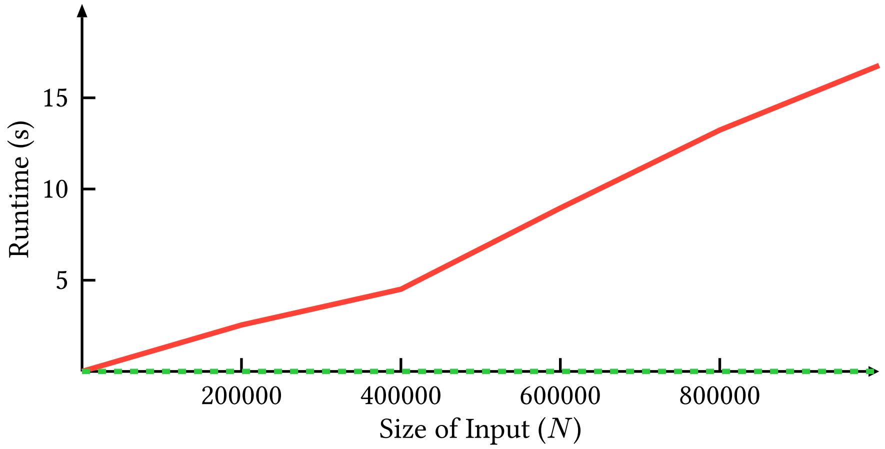
```python
from random import randrange
from datetime import datetime
N = 10000
TRIALS = 1000

#### BEGIN INITIALIZE data
data = []
for x in range(N):
    data += [x]
data = list(data)
#### END INITIALIZE data

contained = 0
start_time = datetime.now()
for x in range(TRIALS):
    if randrange(N) in data:
        contained += 1
end_time = datetime.now()

time = (end_time - start_time).total_seconds() / TRIALS

print(f"For N = {N}, that took {time} seconds per lookup")
```

This code creates a list of `N` elements, and then does `TRIALS` checks to see if a randomly selected value is somewhere in the list. This is a toy example, but see what happens as you increase the value of `N`. In most versions of python, you'll find that every time you multiply `N` by a factor of, for example 10, the total time taken per lookup grows by the same amount.

Now try something else. Modify the code so that the `data` variable is initialized as:

```
#### BEGIN INITIALIZE data
data = []
for x in range(N):
    data += [x]
data = set(data)
#### END INITIALIZE data
```

You'll find that now, as you increase `N`, the time taken **per lookup** grows at a much smaller rate. Depending on the implementation of python you're using, this will either grow as $\log N$ or only a tiny bit. The `set` data structure is much faster at checking whether an element is present than the `list` data structure. Here's the results from the experiment on my own computer, with `list` marked in solid red and `set` marked in dashed green.



Graph 1: Code with `list` and `set`

There's two important things to take away from this experiment.

First, the two lines are distinctly different: the runtime for `list` grows, not quite, but more or less as a straight line, while the runtime for set remains imperceptibly small. If you zoom in, you'll see that it's just about a horizontal line.

Second, this pattern shows up for everyone. It doesn't matter what OS you're using, your CPU, your python version, or any other circumstance of how you run the code. If you're using a CPU that's ten times faster than mine, your numbers will be ten times bigger, but if you plot your results for the same experiment, your graphs will have the same general shape[2].

The reason for this is simple: To find an element in a `list`, we need to check every element, one-by-one, until we find the element we're looking for. On the other hand, in a `set` (implemented as a hash

[2] This is not strictly true. For some implementations of python, you might see a *slight* increase in the runtime for `set` that will look like a logarithmic curve. The main point below, however, still holds.

table), there's only one possible place where a specific element might be found. In general, we only need to do a single check to test whether the element is present[3].

Put another way, in a `list`, if we have twice as many elements, it will take twice as long to check each and every one. Your computer might be ten times faster than mine, but it will still take your computer 2 times as long to find an item in a list of 2 million elements than in a list of 1 million elements. On the other hand, in a `set`, through the awesome power of hash tables, we only need to check a single element, regardless of whether the `set` contains 1 element, 100 elements, 1 million elements, or 1 trillion elements. No matter how big it gets, the cost to check whether an element is in the `set` will always be the same[4].

The idea that data organization creates a predictable relationship between the amount of data and the cost of accessing the data is at the heart of data structures. As a result, it's useful to have some terms that we can use to get across relationships like these without constantly having to resort to saying things like "If you double the number of elements in the list, the runtime of finding an element also doubles."

Asymptotic complexity, which we discuss in this chapter, is how define these terms precisely.

3.1.1. Some examples of asymptotic runtime complexity

Look at the documentation for data structures in your favorite language's standard library. You'll see things like:

- The cost of appending to a Linked List is $O(1)$
- The cost of finding an element in a Linked List is $O(N)$
- The cost of appending to an Array List is $O(N)$, but amortized $O(1)$
- The cost of inserting into a Tree Set is $O(\log N)$
- The cost of inserting into a Hash Map is Expected $O(1)$, but worst case $O(N)$
- The cost of retrieving an element from a Cuckoo Hash is always $O(1)$

These are all examples of asymptotic runtimes, and they are intended to give you a quick at-a-glance idea of how well the data structure handles specific operations. Knowing these properties of the data structures you work with can help you to pick data structures that match the needs of you algorithm, and avoid major performance pitfalls.

3.1.2. Asymptotic Analysis in General

Although our focus in this book is mainly on asymptotic **runtime** complexity, asymptotic analysis is a general tool that can be used to discuss all sorts of properties of code and algorithms. For example:

- How fast is an algorithm?
- How much space does an algorithm need?
- How much data does an algorithm read from disk?
- How much network bandwidth will an algorithm use?
- How many CPUs will a parallel algorithm need?

[3] Again, not entirely true. Under certain circumstances, hash tables can be as bad as lists. We'll talk about this in much more detail later in the book.

[4] This is a bit of a lie. With enough elements you'll run out of memory and your program will either crash or start using 'virtual' memory, which is much slower. Still, for the sizes of data that we'll be dealing with for most of this book, it's a reasonable approximation to assume that the cost won't change.

3.2. Runtime Growth Functions

Throughout most of the book, we'll use $T(N)$ to mean the runtime (T) of an algorithm run on an input of size N. You can think of this function as telling us "For an input of size N, this algorithm will take $T(N)$ seconds to run" This is a little bit of an inexact statement, since the actual number of seconds it takes depends on the type of computer, nuances of implementation, and more. As we'll see later, this imprecision won't actually matter, but for now, you can assume that we're talking about a specific implementation, on a specific computer (like e.g., your computer).

To make our lives easier, we're going to make a few assumptions about how $T(N)$ works:

1. For all $N \geq 0$, it must be the case that $T(N) \geq 0$
   - The algorithm can't take negative time to run.
2. For all $N \geq N'$, it must be the case that $T(N) \geq T(N')$
   - It shouldn't be the case that the algorithm runs faster on a bigger input[5].

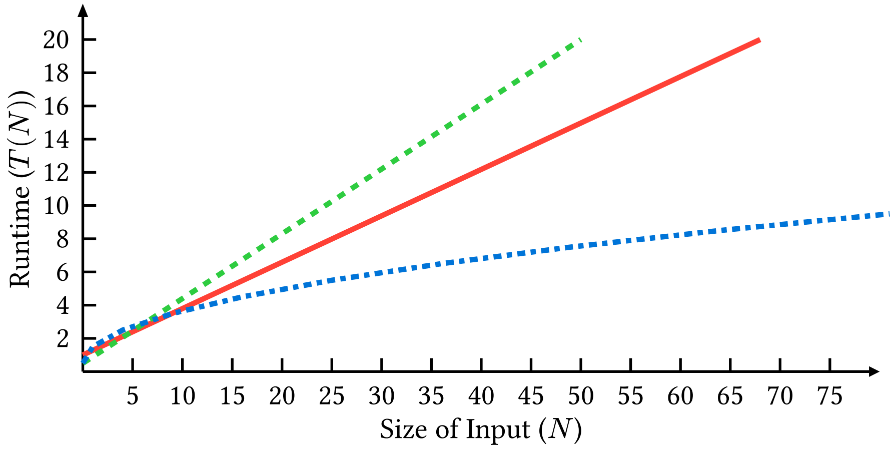
We call any function that follows these two rules a **growth function**, and since $T(N)$ is a runtime, we refer to it as a runtime growth function.

3.3. Complexity Classes

Although we want to define asymptotic complexity classes precisely, it can help to start with a more intuitive idea.

Remember the example above, where the two data structures behaved very differently: The runtime of `in` on a `list` grew linearly with the size of the `list`, while the runtime of `in` on a `set` was completely independent of the size of the `set`. We're going to group these behaviors into something that we're going to call **Complexity Classes**[6].

Let's look at a concrete example: Graph 2 shows three different runtime growth functions: Green (dashed), Red (solid), and Blue (dash-dotted). For an input of size $N = 2$, the green dashed function appears to run the fastest (the best), while the blue dash-dotted function is the slowest (worst). By the time we get to $N = 10$, the roles have reversed, and the blue dash-dotted function is the best.



Graph 2: Different types of growth functions

[5] In practice, this is not actually the case. We'll see a few examples of functions who's runtime can sometimes be faster on a bigger input. Still, for now, it's a useful simplification.

[6] To be pedantic, what we'll be describing is called "simple complexity classes", but throughout this book, we'll refer to them as just complexity classes.

So let's talk about these lines and what we can say about them. First, in this book, since we're taking the 50,000 ft view of algorithm performance, we're going to ignore what happens for "small" inputs. From this perspective, the blue dot-dashed line is the "best".

But why is it better? If we look closely, both the green dashed and the red solid line are straight lines. The blue dot-dashed line starts going up faster than both the other two lines, but bends downward. In short, the blue dot-dashed line draws a function of the form $a \log(N) + b$, while the other two lines draw functions of the form $a \cdot N + b$. For "big enough" values of N, any function of the form $a \log(N) + b$ will always be smaller than any function of the form $a \cdot N + b$. On the other hand, the value of any two functions of the form $a \cdot N + b$ will always "look" the same. No matter how far we zoom out, those functions will always be a constant factor different.

Our 50,000 foot view of the runtime of a function (in terms of N, the size of its input) will be to look at the "shape" of the curve as we plot it against N.

3.4. Formal Notation

Sometimes it's convenient to have a shorthand for writing down that a runtime belongs in a complexity class. We write:

$g(N) \in \Theta(f(n))$

… to mean that the mathematical function $g(N)$ belongs to the same **asymptotic complexity class** as $f(N)$. You may also see this written as an equality. For example

$T(N) = \Theta(N)$

… means that the runtime function $T(N)$ belongs to the **linear** complexity class. Continuing the example above, we would use our new shorthand to describe the two implementations of Python's `in` operator as:

- $T_{\text{set}} \in \Theta(\log N)$
- $T_{\text{list}} \in \Theta(N)$

Formalism: A little more formally, $\Theta(f(N))$ is the **set** of all mathematical functions $g(N)$ that belong to the same complexity class as $f(N)$. So, writing $g(N) \in \Theta(f(N))$ is saying that $g(N)$ is in ($\in$) the set of all mathematical functions in the same complexity class as $f(N)$[7].

Here are some of the more common complexity classes that we'll encounter throughout the book:

- **Constant**: $\Theta(1)$
- **Logarithmic**: $\Theta(\log N)$
- **Linear**: $\Theta(N)$
- **Loglinear**: $\Theta(N \log N)$
- **Quadratic**: $\Theta(N^2)$
- **Cubic**: $\Theta(N^3)$
- **Exponential** $\Theta(2^N)$

[7] We are sweeping something under the rug here: We haven't precisely defined what it means for two functions to be in the same complexity class yet. We'll get to that shortly, after we introduce the concept of complexity bounds.

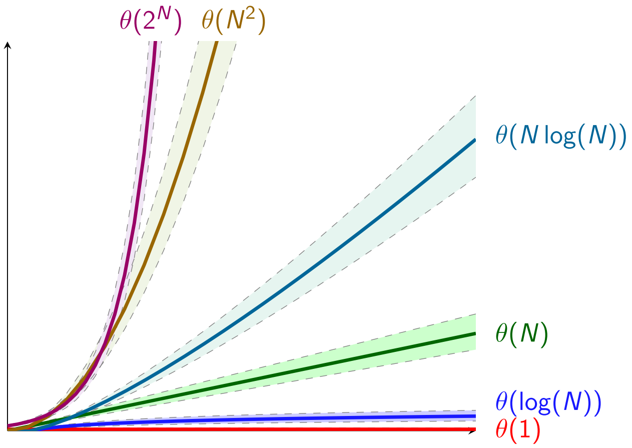
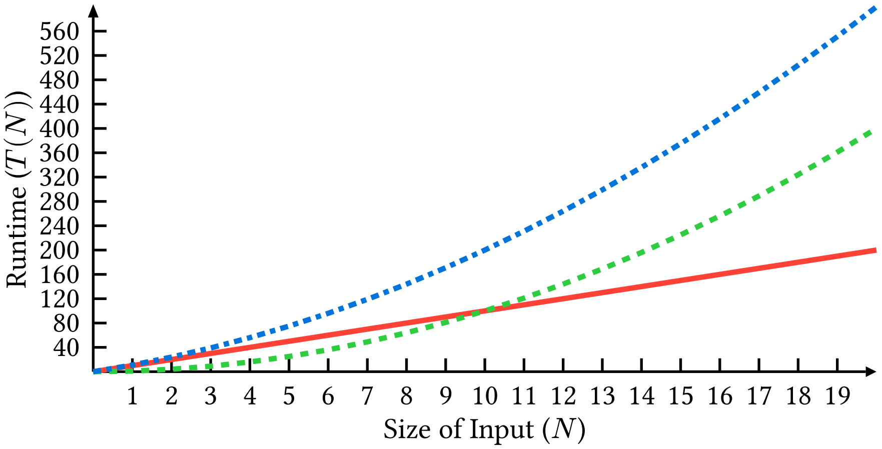


Figure 1: $\Theta(f(N))$ is the set of all mathematical functions including $f(N)$ and everything that has the same "shape", represented in the chart above as a highlighted region around each line.

This list of complexity classes is presented in a specific order. The later a class appears in the list above, the faster a function in the class grows. Any function that has a **linear** shape, will always be smaller (for big enough values of N) than a function with a **loglinear** shape.

3.4.1. Polynomials and Dominant Terms

What complexity class does $10N + N^2$ fall into? Let's plot it and see:



Graph 3: Comparing $10N$ (solid red), N^2 (dashed green), and $10N + N^2$ (dash-dotted blue)

Graph 3 compares these three functions. Observe that the dash-dotted blue line starts off very similar to the solid red line. However, as N grows, its shape soon starts resembling the dashed green N^2 more than the solid red $10N$.

Although we don't have the tools to prove it yet, take our word for it that this is a pattern. In any polynomial (a sum of mathematical expressions), for really big values of N, the complexity class of the "biggest" term starts to win out once we get to really big values of N.

In general, for any sum of mathematical functions:

$$g(N) = f_1(N) + f_2(N) + \ldots + f_k(N)$$

The complexity class of $g(N)$ is the greatest complexity class of any $f_i(N)$

For example:

- $10N + N^2 \in \Theta(N^2)$
- $2^N + 4N \in \Theta(2^N)$
- $1000 \cdot N\log(N) + 5N \in \Theta(N\log(N))$

We'll prove this formally at the end of the chapter in Section 3.13.6.

3.4.2. Θ in mathematical formulas

Sometimes we'll write $\Theta(g(N))$ in regular mathematical formulas. For example, we could write:

$$\Theta(N) + 5N$$

You should interpret this as meaning any function that has the form:

$$f(N) + 5N$$

… where $f(N) \in \Theta(N)$.

3.5. Code Complexity

Let's see a few examples of how we can figure out the runtime complexity class of a piece of code.

```
def userFullName(users: List[User], id: int) -> str:
    user = users[id]
    fullName = user.firstName + " " + user.lastName
    return fullName
```

The `userFullName` function takes a list of users, and retrieves the `id`th element of the list and generates a full name from the user's first and last names. For now, we'll assume that looking up any element of any array (`users[id]`), string concatenation (`user.firstName + " " + user.lastName`), assignment (`user = ...`, and `fullName`), and returns are all constant-time operations[8].

Under these assumptions, the first, second, and third lines can each be evaluated in constant time $\Theta(1)$. The total runtime of the function is the time required to run each line, one at a time, or:

$$T_{\text{userFullName}}(N) = \Theta(1) + \Theta(1) + \Theta(1)$$

Recall above, that $\Theta(1)$ in the middle of an arithmetic expression can be interpreted as $f(N)$ where $f(N) \in \Theta(1)$ (it is a constant). That is, $f(N) = c$. So, the above expression can be rewritten as[9]:

$$T_{\text{userFullName}}(N) = c_1 + c_2 + c_3$$

Adding three constant values together (even without knowing what they are, exactly) always gets us another constant value. So, we can say that $T_{\text{userFullName}}(N) \in \Theta(1)$.

```
def updateUsers(users: List[User]) -> None:
    x = 1
    for user in users:
```

[8] Array lookups being constant-time is a huge simplification, called the RAM model, that we'll roll back at the end of the book. Similarly, string concatenation is not quite $\Theta(1)$. It's usually $\Theta(N)$ where N is the size of the strings being concatenated. However, as long as we assume that strings are relatively small, we'll pretend (for now) that string concatenation is constant-time.

[9] There's another simplification here. Technically, $f(N)$ is always within a bounded factor of a constant c_1, and likewise for $g(N)$, but we'll clarify this when we get to complexity bounds below.

```
        user.id = x
        x += 1
```

The `updateUsers` function takes a list of users and assigns each user a unique id. For now, we'll assume that the assignment operations (`x = 1` and `user.id`), and the increment operation (`x += 1`) all take constant ($\Theta(1)$) time. So, we can model the total time taken by the function as:

$$T_{\text{updateUsers}}(N) = O(1) + \sum_{\text{user}} (O(1) + O(1))$$

Simplifying as above, we get

$$T_{\text{updateUsers}}(N) = c_1 + \sum_{\text{user}} (c_2 + c_3)$$

Recalling the rule for summation of a constant, using N as the total number of users, and then the rule for sums of terms, we get:

$$T_{\text{updateUsers}}(N) = c_1 + N \cdot (c_2 + c_3) = \Theta(N)$$

3.6. Complexity Bounds

Not every mathematical function fits neatly into a single complexity class. Let's go to our python code example above. The `in` operator tests to see whether a particular value is present in our data. If `data` is a `list`, then the implementation checks every position in the list, in order. Internally, Python implements the expression `target in data` with something like:

```
def __in__(data, target):
    N = len(data)
    for i in range(N):
        if data[i] == target:
            return True
    return False
```

In the best case, the value we're looking for happens to be at the first position `data[0]`, and the code returns after a single check. In the worst case, the value we're looking for is at the last position `data[N-1]` or is not in the list at all, and we have to check every one of the `N` positions. Put another way, the **best case** behavior of the function is constant (exactly one check), while the **worst case** behavior is linear (N checks). We can write the resulting runtime using a case distinction:

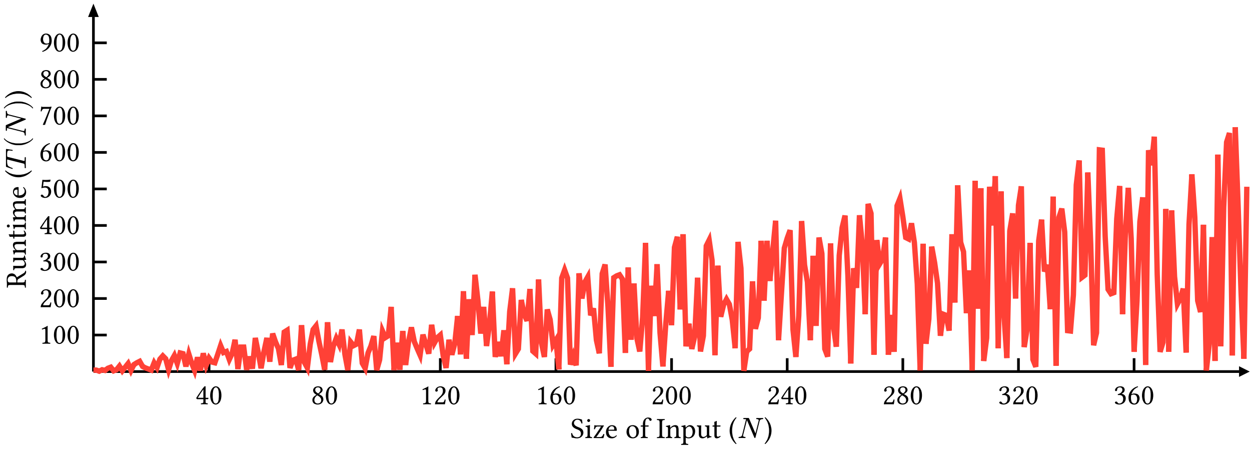
$$T_{\text{in}}(N) = \begin{cases} a \cdot 1 + b & \text{if } \texttt{data}[0] = \texttt{target} \\ a \cdot 2 + b & \text{if } \texttt{data}[1] = \texttt{target} \\ \dots & \\ a \cdot (N-1) + b & \text{if } \texttt{data}[\texttt{N-2}] = \texttt{target} \\ a \cdot N + b & \text{if } \texttt{data}[\texttt{N-1}] = \texttt{target} \end{cases}$$

We don't know the runtime exactly, as it is based on the computer and version of python we are using. However, we can model it, in general in terms of some upfront cost b (e.g., for computing `N = len(data)`), and some additional cost a for every time we go through the loop (e.g., for computing `data[i] == target`). Since we don't know where the target is, exactly,

Let's do a quick experiment. The code below is like our example above, but measures the time for one lookup at a time. Each point it prints out is the runtime of a single lookup as the `list` gets bigger and bigger.

```
from random import randrange
from datetime import datetime
```

```
N = 100000
TRIALS = 400
STEP = int(N/TRIALS)

data = list()

for i in range(TRIALS):
    # Increase the list size by STEP
    for j in range(STEP):
        data += [i * STEP + j]
    start = datetime.now()
    # Measure how long it takes to look up a random element
    if randrange(i * STEP + STEP) in data:
        pass
    end = datetime.now()
    # Print out the total time in microseconds
    microseconds = (end - start).total_seconds() * 1000000
    print(f"{i}, {microseconds}")
```

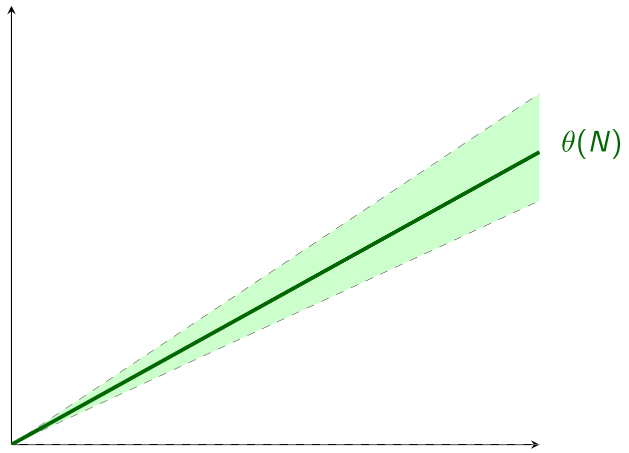


Graph 4: Scaling the `list` lookup.

Graph 4 shows the output of one run of the code above. You can see that it looks a lot like a triangle. The **worst case** (top of the triangle) looks a lot like the **linear** complexity class ($\Theta(N)$, or an angled line), but the **best case** (bottom of the triangle) looks a lot more like a flat line, or the **constant** complexity class ($\Theta(1)$, or a flat line). The runtime is *at least* constant, and *at most* linear: We can **bound** the runtime of the function between two complexity classes.

3.7. Big-O and Big-Ω

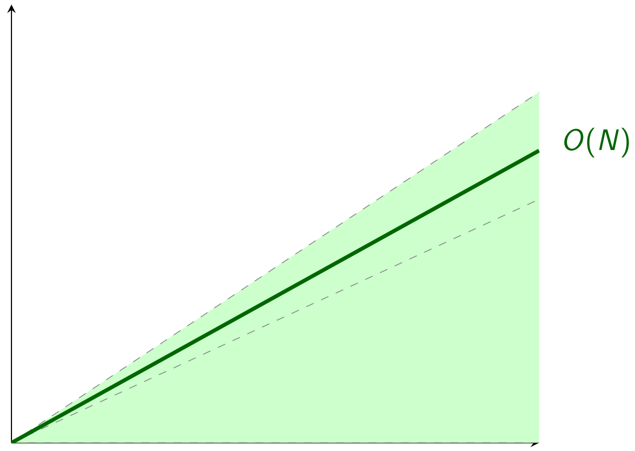
We capture this intuition of bounded runtime by introducing two new concepts: Worst-case (upper, or Big-O) and Best-case (lower, or Big-Ω) bounds. To see these in practice, let's take the linear complexity class as an example:



We write $O(N)$ to mean the set of all mathematical functions that are **no worse than** $\Theta(N)$. This includes:

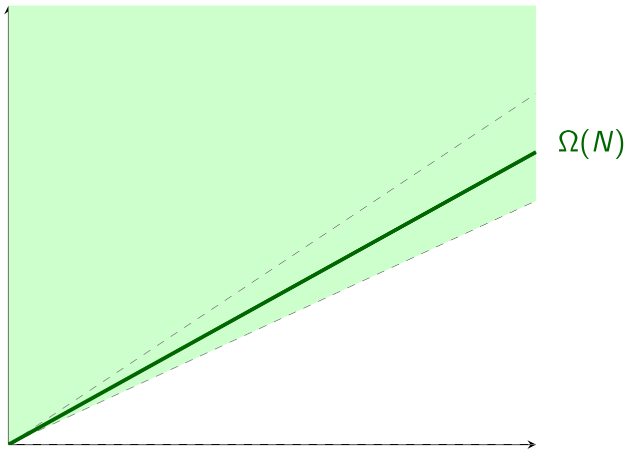
- all mathematical functions in $\Theta(N)$ itself
- all mathematical functions in lesser (slower-growing) complexity classes (e.g., $\Theta(1)$)
- any mathematical function that never grows faster than $O(N)$ (e.g., the runtime of each individual lookup in our Python example above)

The figure below illustrates $\Theta(N)$ (the dotted region) and all lesser complexity classes. Note the similarity to Graph 4.



Similarly, we write $\Omega(N)$ to mean the set of all mathematical functions that are **no better than** $\Theta(N)$. This includes:

- all functions in $\Theta(N)$ itself
- all greater (faster-growing) complexity classes (e.g., $\Theta(N^2)$), and anything in between.



To summarize, we write:

- $f(N) \in O(g(N))$ to say that $f(N)$ is in $\Theta(g(N))$ or a lesser complexity class.
- $f(N) \in \Omega(g(N))$ to say that $f(N)$ is in $\Theta(g(N))$ or a greater complexity class.

3.8. Formalizing Big-O

Before we formalize our bounds, let's first figure out what we want out of that formalism.

Let's start with the basic premise we outlined above: For a function $f(N)$ to be in the set $O(g(N))$, we want there to be some function in $\Theta(g(N))$ that is always bigger than $f(N)$.

The first problem we run into with this formalism is that we haven't really defined what exactly $\Theta(g(N))$ is yet, so we need to pin down something first. Let's start with the same assumption we made earlier: we can scale $g(N)$ by any constant value without changing its complexity class.

Formally: $\forall c : c \cdot g(N) \in \Theta(g(N))$

That is, for any constant c ($\forall$ means 'for all'), the product $c \cdot g(N)$ is in $\Theta(g(N))$ (remember that $\in$ means is in). This isn't meant to be all-inclusive: There are many more functions in $\Theta(g(N))$, but this gives us a pretty good range of functions that, at least intuitively, belong in $g(N)$'s complexity class.

Now we have a basis for formalizing Big-O: We can say that $f(N) \in O(g(N))$ if there is **some** multiple of $g(N)$ that is always bigger than $f(N)$. Formally:

$\exists c > 0, \forall N : f(N) \leq c \cdot g(N)$

That is, there exists ($\exists$ means there exists) some positive constant c, such that for each value of N, the value of $f(N)$ is smaller than the corresponding value of $c \cdot g(N)$.

Let's look at some examples:

Example: $f(N) = 2N$ vs $g(N) = N$

Can we find a c and show that for this c, for all values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $2N \leq c \cdot N$
- $2 \leq c$

We start with the basic inequality, substitute in the values of $f(N)$ and $g(N)$, and then divide both sides by N. So, the inequality is always true for any value of $c \geq 2$.

Example: $f(N) = 100N^2$ vs $g(N) = N^2$

Can we find a c and show that for this c, for all values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $100N^2 \leq c \cdot N^2$
- $100 \leq c$

We start with the basic inequality, substitute in the values of $f(N)$ and $g(N)$, and then divide both sides by N. So, the inequality is always true for any value of $c \geq 100$.

Example: $f(N) = N$ vs $g(N) = N^2$

Can we find a c and show that for this c, for all **integer** values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $N \leq c \cdot N^2$
- $1 \leq c \cdot N$

Uh-oh! For $N = 0$, there is no possible value of c that we can plug into that inequality to make it true ($0 \cdot c$ is never bigger than 1 for any c).

Attempt 2: So what went wrong? Well, we mainly care about how $f(N)$ behaves for really big values of N. In fact, for the example, for any $N \geq 1$ (and $c \geq 1$), the inequality is satisfied! It's just that pesky $N = 0$![10].

So, we need to add one more thing to the formalism: the idea that we only care about "big" values of N. Of course, that leaves the question of how big is "big"? Now, we could pick specific cutoff values, but any specific cutoff we picked would be entirely arbitrary. So, instead, we just make the cutoff definition part of the proof: When proving that $f(N) \in O(g(N))$, we just need to show that **some** cutoff exists, beyond which $f(N) \leq c \cdot g(N)$.

The formal definition of Big-O is:

$f(N) \in O(g(N)) \Leftrightarrow \exists c > 0, N_0 \geq 0 : \forall N \geq N_0 : f(N) \leq c \cdot g(N)$

In this equation, $\exists$ means "there exists" and $\forall$ means "for all". Teasing apart the above equation:

- $f(N) \in O(g(N))$, the thing we want to define, is equivalently defined as ($\Leftrightarrow$)…
- There exists some constant c strictly bigger than 0 ($\exists c > 0$).
- There exists some cutoff value for N ($\exists N_0 \geq 0$)…
- So that for any larger value of N ($N \geq N_0$)…
- $f(N)$ is smaller than $c \cdot g(N)$.

In other words, saying $f(N) \in O(g(N))$ is the same as saying that there is some constant c so that for sufficiently large N, $f(N) \leq c \cdot g(N)$.

[10] Recall that we're only allowing non-negative input sizes (i.e., $N \geq 0$), so negative values of N aren't a problem.

3.8.1. Proving a function has a specific Big-O bound

To show that a mathematical function is **in** $O(g(N))$, we need to find a c and an N_0 for which we can prove the Big-O inequality. A generally useful strategy is:

1. Write out the Big-O inequality
2. "plug in" the values of $f(N)$ and $g(N)$
3. "Solve for" c, putting it on one side of the inequality, with everything else on the other side.
4. Try a safe default of $N_0 = 1$.
5. Use the $A \leq B$ and $B \leq C$ imply $A \leq C$ trick (transitivity of inequality) to replace any term involving N with N_0
5. Use the resulting inequality to find a lower bound on c

Continuing the above example of $f(N) = N$ and $g(N) = N^2$, we want to show that there is a constant c so that for sufficiently large N.

$f(N) \leq c \cdot g(N)$

We start by "plugging in" values of f and g:

$N \leq c \cdot N^2$

We can solve for c by dividing both sides by N^2, getting:

$\frac{1}{N} \leq c$

From here, we need to find a constant c that is bigger than $\frac{1}{N}$ for all sufficiently large values of N. If we can find such a constant, then we can work backwards through the proof to show that $f(N) \leq c \cdot g(N)$ for that constant.

Remember that we defined "sufficiently large" values of N as all values of N greater than some constant N_0. Following the guidelines above, let's pick a value of $N_0 = 1$. Observe that the function $\frac{1}{N}$ shrinks as N grows. The greatest value of $\frac{1}{N}$ occurs when N is at its smallest value ($N_0 = 1$). In other words, we can say that:

$\frac{1}{N} \leq 1$ for all $N \geq 1$

This equation fits the pattern! So, since

$\frac{1}{N} \leq 1$

... and with $c = 1$, we have

$\frac{1}{N} \leq c$

Which in turn means that

$N \leq c \cdot N^2$

And so swapping in $f(N)$ and $g(N)$:

$f(N) \leq c \cdot g(N)$

3.8.2. Proving a function does not have a specific Big-O bound

To show that a mathematical function is **not in** $O(g(N))$, we need to prove that there can be **no** c or N_0 for which we can prove the Big-O inequality. A generally useful strategy is:

1. Write out the Big-O inequality
2. "plug in" the values of $f(N)$ and $g(N)$
3. "Solve for" c, putting it on one side of the inequality, with everything else on the other side.
4. Simplify the equation on the opposite side and show that it is strictly growing. Generally, this means that the right-hand-side is in a complexity class at least N.

Flipping the above example:

- $f(N) \leq c \cdot g(N)$
- $N^2 \leq c \cdot N$
- $N \leq c$

N is strictly growing: for bigger values of N, it gets bigger. There is no constant that can upper bound the mathematical function N.

3.9. Formalizing Big-Ω

Now that we've formalized Big-O (the upper bound), we can formalize Big-Ω (the lower bound) in exactly the same way:

$f(N) \in O(g(N)) \Leftrightarrow \exists c > 0, N_0 \geq 0 : \forall N \geq N_0 : f(N) \geq c \cdot g(N)$

The only difference is the direction of the inequality: To prove that a function exists in Big-Ω, we need to show that $f(N)$ is bigger than some constant multiple of $g(N)$.

3.10. Formalizing Big-Θ

Although we started with an intuition for Big-Θ, we haven't yet formalized it. To understand why, let's take a look at the following runtime:

$T(N) = 10N + \text{rand}(10)$

Here $\text{rand}(10)$ means a randomly generated number between 0 and 10 for each call. If the function were **just** $10N$, we'd be fine in using our intuitive definition of $\Theta(N)$ being all multiples of N. However, this function still "behaves like" $g(N) = N$… just with a little random noise added in. For big values of N (e.g., 10^{10}), the random noise is barely perceptible. Although we can't say that $T(N)$ is **equal to** some multiple $c \cdot N$, we can say that it is **close to** that multiple (in fact, it's always between $10N$ and $10N + 10$). In other words, we can bound it from both above and below!

Let's try proving this with the tricks we developed for Big-O and Big-Ω:

- $T(N) \leq c_{\text{upper}} \cdot N$
- $10N + \text{rand}(10) \leq c_{\text{upper}} \cdot N$ (plug in $T(N)$)
- $10 + \frac{\text{rand}(10)}{N} \leq c_{\text{upper}}$ (divide by N)

Looking at this formula, we can make a few quick observations. First, by definition $\text{rand}(10)$ is never bigger than 10. Second, if $N_0 = 1$, $\frac{1}{N}$ can never be bigger than 1. In other words, $\frac{\text{rand}(10)}{N}$ can not be bigger than 10. Let's prove that to ourselves.

Taking the default $N_0 = 1$ we get:

- $1 \leq N$ (plug in N_0)
- $10 \leq 10N$ (multiply by 10)
- $\text{rand}(10) \leq 10 \leq 10N$ (transitivity with $\text{rand}(10) \leq 10$)

- $\frac{\text{rand}(10)}{N} \leq 10$ (divide by N)
- $10 + \frac{\text{rand}(10)}{N} \leq 10 + 10$ (add 10)

So, if we pick $c_{\text{upper}} \geq 20$, we can show (again, by transitivity):

$10 + \frac{\text{rand}(10)}{N} \leq c_{\text{upper}}$

Which gets us $T(N) \leq c_{\text{upper}} \cdot N$ for all $N > N_0$.

Now let's try proving a lower (Big-Ω) bound:

- $T(N) \geq c_{\text{lower}} \cdot N$
- $10N + \text{rand}(10) \geq c_{\text{lower}} \cdot N$ (plug in $T(N)$)
- $10 + \frac{\text{rand}(10)}{N} \geq c_{\text{lower}}$ (divide by N)
- $10 \geq c_{\text{lower}}$ (By transitivity: $10 \geq 10 + \frac{\text{rand}(10)}{N}$)

This inequality holds for any $10 \geq c_{\text{lower}} > 0$ (recall that c has to be strictly bigger than zero).

So, we've shown that $T(N) \in O(N)$ **and** $T(N) \in \Omega(N)$. The new thing is that we've shown that the upper and lower bounds **are the same**. That is, we've shown that $T(N) \in O(g(N))$ and $T(N) \in \Omega(g(N))$ **for the same mathematical function** g. If we can prove that an upper and lower bound for some mathematical function $f(N)$ that is the same mathematical function $g(N)$, we say that $f(N)$ and $g(N)$ are in the same complexity class.

Formally, $f(N) \in \Theta(g(N))$ is defined as $f(N)$ being bounded from **both** above and below by $g(N)$. In other words, $f(N) \in \Theta(g(N))$ if and only if $f(N) \in O(g(N))$ **and** $f(N) \in \Omega(g(N))$.

3.11. Tight Bounds

In the example above, we said that $\text{rand}(10) \leq 10$. We could have just as easily said that $\text{rand}(10) \leq 100$. The latter inequality is just as true, but somehow less satisfying; yes, the random number will always be less than 100, but we can come up with a "tighter" bound (i.e., 10).

Similarly Big-O and Big-Ω are bounds. We can say that $N \in O(N^2)$ (i.e., N is no worse than N^2). On the other hand, this bound is just as unsatisfying as $\text{rand}(10) \leq 100$, we can do better.

If it is not possible to obtain a better Big-O or Big-Ω bound, we say that the bound is **tight**. For example:

- $10N^2 \in O(N^2)$ and $10N^2 \in \Omega(N^2)$ are tight bounds.
- $10N^2 \in O(2^N)$ is correct, but **not** a tight bound.
- $10N^2 \in \Omega(N)$ is correct, but **not** a tight bound.

Note that since we define Big-Θ as the intersection of Big-O and Big-Ω, all Big-Θ bounds are, by definition tight. As a result, we often call Big-Θ bounds "tight bounds".

Note: It **is** possible for a Big-O or Big-Ω bound to be tight, without having a Big-Θ bound. You'll see an example of this below in Section 3.13.2.

3.12. Which Variable?

We define asymptotic complexity bounds in terms of **some** variable, usually the size of the collection N. However, it's also possible to use other bounds. For example, consider the function, which computes factorial:

```
public int factorial(int idx)
{
  if(idx <= 0){ return 0; }
  int result = 1;
  for(i = 1; i <= idx; i++) { result *= i; }
  return result
}
```

The runtime of this loop depends on the input parameter `idx`, performing one math operation for each integer between 1 and `idx`. So, we could give the runtime as $\Theta(\text{idx})$.

When the choice of variable is implicit, it's customary to just use N, but this is not always the case. For example, when we talk about sequences and lists, the size of the sequence/list is most frequently the variable of interest. However, there might be other parameters of interest:

- If we're searching the list for a specific element, what position to we find the element at?
- If we're looking through a linked list for a specific element, what index are we looking for?
- If we have two or more lists (e.g., in an Edge List data structure), each list may have a different size.

In these cases, and others like them, it's important to be clear about which variable you're talking about.

3.12.1. Related Variables

When using multiple variables, we can often bound one variable in terms of another. Examples include:

- If we're looking through a linked list for the element at a specific index, the index must be somewhere in the range $[0, N)$, where N is the size of the list. As a result, we can can always replace $O(\text{index})$ with $O(N)$ and $\Omega(\text{index})$ with $\Omega(1)$, since `index` is bounded from above by a linear function of N and from below by a constant.

- The number of edges in a graph can not be more than the square of the number of vertices. As a result, we can always replace $O(\text{edges})$ with $O(\text{vertices}^2)$ and $\Omega(\text{edges})$ with $\Omega(1)$.

Note: Even though $O(\text{index})$ in the first example may be a tighter bound than $O(N)$, the $O(N)$ bound is still tight **in terms of** N: We can not obtain a tighter bound that is a function only of N.

3.13. Summary

We defined three ways of describing runtimes (or any other mathematical function):

- Big-O: The worst-case complexity:
  - $T(N) \in O(g(N))$ means that the runtime $T(N)$ scales **no worse than** the complexity class of $g(N)$
- Big-Ω: The best-case complexity
  - $T(N) \in \Omega(g(N))$ means that the runtime $T(N)$ scales **no better than** the complexity class of $g(N)$
- Big-Θ: The tight complexity
  - $T(N) \in \Theta(g(N))$ means that the runtime $T(N)$ scales **exactly as** the complexity class of $g(N)$

We'll introduce amortized and expected runtime bounds later on in the book; Since these bounds are given without qualifiers, and so are sometimes called the **Unqualified** runtimes.

3.13.1. Formal Definitions

For any two functions $f(N)$ and $g(N)$ we say that:

- $f(N) \in O(g(N))$ if and only if $\exists c > 0, N_0 \geq 0 : \forall N > N_0 : f(N) \leq c \cdot g(N)$
- $f(N) \in \Omega(g(N))$ if and only if $\exists c > 0, N_0 \geq 0 : \forall N > N_0 : f(N) \geq c \cdot g(N)$
- $f(N) \in \Theta(g(N))$ if and only if $f(N) \in O(g(N))$ and $f(N) \in \Omega(g(N))$

Note that a simple $\Theta(g(N))$ may not exist for a given $f(N)$, specifically when the tight Big-O and Big-Ω bounds are different.

3.13.2. Interpreting Code

In general[11], we will assume that most simple operations: basic arithmetic, array accesses, variable access, string operations, and most other things that aren't function calls will all be $\Theta(1)$.

Other operations are combined as follows…

Sequences of instructions

```
{
  op1;
  op2;
  op3;
}
...
```

Sum up the runtimes. $T(N) = T_{\text{op1}}(N) + T_{\text{op2}}(N) + T_{\text{op3}}(N) + \ldots$

Loops

```
for(i = min; i < max; i++)
{
  block;
}
```

Sum up the runtimes for each iteration. Make sure to consider the effect of the loop variable on the runtime of the inner block. $T(N) = \sum_{i=\min}^{\max} T_{\text{block}}(N, i)$

As a simple shorthand, if (i) the number of iterations is predictable (e.g., if the loop iterates N times) and (ii) the complexity of the loop body is independent of which iteration the loop is on (i.e., i does not appear in the loop body), you can just multiply the complexity of the loop by the number of iterations.

Conditionals

```
if(condition){
  block1;
} else {
  block2;
}
```

The total runtime is the cost of either `block1` or `block2`, depending on the outcome of `condition`. Make sure to add the cost of evaluating `condition`.

[11] All of these are lies. The cost of basic arithmetic is often $O(\log N)$, array access runtimes are affected by caching (we'll address that later in the book), and string operations are proportional to the length of the string. However, these are all useful simplifications for now.

$$T(N) = T_{\text{condition}(N)} + \begin{cases} T_{\text{block1}}(N) \text{ if condition is true} \\ T_{\text{block2}}(N) \text{ otherwise} \end{cases}$$

The use of a cases block is especially important here, since if $T_{\text{block1}}(N)$ and $T_{\text{block2}}(N)$ belong to different asymptotic complexity classes, the overall block of code belongs to multiple classes (and thus does not have a simple Θ bound).

3.13.3. Simple Complexity Classes

We will refer to the following specific complexity classes:

- **Constant**: $\Theta(1)$
- **Logarithmic**: $\Theta(\log N)$
- **Linear**: $\Theta(N)$
- **Loglinear**: $\Theta(N \log N)$
- **Quadratic**: $\Theta(N^2)$
- **Cubic**: $\Theta(N^3)$
- **Exponential** $\Theta(2^N)$

These complexity classes are listed in order.

3.13.4. Dominant Terms

In general, any function that is a sum of simpler functions will be dominated by one of its terms. That is, for a polynomial:

$$f(N) = f_1(N) + f_2(N) + \ldots + f_k(N)$$

The asymptotic complexity of $f(N)$ (i.e., its Big-O and Big-Ω bounds, and its Big-Θ bound, if it exists) will be the **greatest** complexity of any individual term $f_i(N)$[12].

Remember: If the dominant term in a polynomial belongs to a single simple complexity class, then the entire polynomial belongs to this complexity class, and the Big-O, Big-Ω, and Big-Θ bounds are all the same.

3.13.5. Multiclass Asymptotics

A mathematical function may belong to multiple simple classes, depending on an unpredictable input or the state of a data structure. Generally, multiclass functions arise in one of two situations. First, the branches of a conditional may have different complexity classes:

$$T(N) = \{ T_1(N) \text{ if a thing is true } T_2(N) \text{ otherwise}$$

If $T_1(N)$ and $T_2(N)$ belong to different complexity classes, then $T(N)$ as a whole belongs to **either** class. In this case, we can only bound the runtime $T(N)$. Specifically, if $\Theta(T_1(N)) > \Theta(T_2(N))$, then:

- $T(N) \in \Omega(T_2(N))$
- $T(N) \in O(T_1(N))$.

Second, the number of iterations of a loop may depend on an input that is bounded by multiple complexity classes. For example, if $\text{idx} \in [1, N]$ (`idx` is somewhere between 1 and N, inclusive), then the following code does not belong to a single complexity class:

```
for(i = 0; i < idx; i++){ do_a_thing(); }
```

[12] Note that this is only true when k is fixed. If the number of polynomial terms depends on N, we need to consider the full summation.

In this case, we can bound the runtime based on `idx`. Assuming `do_a_thing()` is $\Theta(1)$, then $T(N) \in \Theta(\text{idx})$. However, since we can't bound `idx` in terms of N, then we can only provide weaker bounds with respect to N:

- $T(N) \in \Omega(1)$
- $T(N) \in O(N)$

Remember that if we can not obtain identical, tight upper and lower bounds in terms of a given input variable, there is no simple Θ-bound in terms of that variable.

3.13.6. Proving Summation

Earlier, we claimed that the sum of a collection of functions belonged to the greatest complexity class of any of the component functions.

To make this statement more precise, let's start by defining the sum (g) and the component functions ($f_1 ... f_k$). Although we make this assumption in general, we'll be explicit here that we're going to assume that each f_i is a growth function.

$$g(N) = f_1(N) + f_2(N) + ... + f_k(N)$$

At least one of the functions has to belong to the greatest complexity class. Let's call this one f_{max}. In formal terms, this means that for all $1 \leq i \leq k$:

$$\forall i \in [1, k] : f_i(N) \in O(f_{\text{max}}(N)).$$

We want to prove that $g(N) = \Theta(f_{\text{max}}(N))$. Recall that, to prove this, we need to show (i) that $g(N) \in O(f_{\text{max}}(N))$, and (ii) that $g(N) \in \Omega(f_{\text{max}}(N))$

3.13.6.1. Upper Bound

Let's start with showing Big-O. We want to show that

$$g(N) \in O(f_{\text{max}})$$

Expanding out both $g(N)$ and the Big-O, this is equivalent to showing that there exists a c so that for any sufficiently large N:

$$f_1(N) + f_2(N) + ... + f_k(N) \leq c \cdot f_{\text{max}}(N)$$

Recall that, since f_{max} belongs to the greatest complexity class, we have that for all $1 \leq i \leq k$:

$$\forall i \in [1, k] : f_i(N) \in O(f_{\text{max}}(N)).$$

Or, expanding out the Big-O (for a sufficiently large N):

$$\forall i \in [1, k] : f_i(N) \leq c_i \cdot f_{\text{max}}(N).$$

If we add together both sides of this equation, we get

$$f_1(N) + f_2(N) + ... + f_k(N) \leq c_1 \cdot f_{\text{max}}(N) + c_2 \cdot f_{\text{max}}(N) + ... + c_k \cdot f_{\text{max}}(N)$$

Factoring the function out from the right-hand side:

$$f_1(N) + f_2(N) + ... + f_k(N) \leq (c_1 + c_2 + ... + c_k) \cdot f_{\text{max}}(N)$$

And since $c_1 + c_2 + ... + c_k$ is a constant, we can label that constant c:

$$f_1(N) + f_2(N) + ... + f_k(N) \leq c \cdot f_{\text{max}}(N)$$

Which is what we wanted to show in the first place, so as the math hippies say, QED.

3.13.6.2. Lower Bound

A bit less intuitively, we want to show that $g(N)$ grows at least as fast as $f_{\text{max}}(N)$, or:

$$g(N) \in \Omega(f_{\text{max}})$$

Again, we expand out both $g(N)$ and Big-Ω, and want to show that (for a sufficiently large N), there is a constant c for which we can show:

$$f_1(N) + f_2(N) + \ldots + f_k(N) \geq c \cdot f_{\text{max}}(N)$$

Remember that we're assuming that each f_i is a growth function. Going back to the definition of growth functions, this means that $f_i(N) \geq 0$ for **any** positive value of N. Since each f_i must be at least 0, replacing every f_i **except** f_{max} with 0 can only make their sum smaller:

$$f_1(N) + f_2(N) + \ldots + f_k(N) \geq 0 + \ldots + f_{\text{max}}(N) + \ldots + 0$$

Simplifying the right-hand side:

$$f_1(N) + f_2(N) + \ldots + f_k(N) \geq f_{\text{max}}(N)$$

Multiplying the right-hand side by 1

$$f_1(N) + f_2(N) + \ldots + f_k(N) \geq 1 \cdot f_{\text{max}}(N)$$

Which is the equation that we want to show, for $c = 1$.

Chapter 4. *The Sequence and List ADTs*

Now that we have the right language to talk about algorithm runtimes (asymptotic complexity), we can start talking about actual data structures that these algorithms can use. Since the choice of data structure can have a huge impact on the complexity of an algorithm, it's often useful to group specific data structures together by the roles that they can fulfill in an algorithm. We refer to such a grouping as an **abstract data type** or ADT.

4.1. What is an ADT?

An ADT is, informally, a contract that states what we can expect from a data structure. Take, for example, a Java `interface` like the following:

```
public interface Narf<T>
{
  public void foo(T poit, int zort);
  public T bar(int zort);
  public int baz();
}
```

This `interface` states that any class that implements `Narf` must provide `foo`, `bar`, and `baz` methods, with arguments and return values as listed above. This is not especially helpful, since it doesn't give us any idea of what these methods are supposed to do. Contrast `Narf` with the following interface:

```
public interface MutableSequence<T>
{
  public void set(int index, T element);
  public T get(int index);
  public int size();
}
```

`Sequence` is semantically identical to `Narf`, but far more helpful. Just by reading the names of these methods, you get some idea of what each method does, and how it interacts with the state represented by a `Sequence`. You can build a mental model of what a Sequence is from these names.

An Abstract Data Type is defined in three parts:
1. Formal rules for the type of data being stored (a data type).
2. One or more operations for accessing or modifying that state (an interface).
3. Any other rules for the state (constraints).

For most ADTs discussed in this book, the ADT models sort of collection of elements. The difference between them is how you're allowed to interact with the data.

A data structure is a **specific** strategy for organizing the data modeled by an ADT. We say that the data structure implements (or conforms to) an ADT if:
1. The data structure stores the same type of data as the ADT
2. Each of the operations required by the ADT can be implemented on the data structure
3. We can prove that the operation implementation is guaranteed to respect the rules for the ADT's state.

4.2. The Sequence ADT

A very common example of an ADT is a sequence. Examples include:

- The English alphabet (‘A’, ‘B’, …, ‘Z’)
- The Fibonacci Sequence (1, 1, 2, 3, 5, 8, …)
- An arbitrary collection of numbers (42, 19, 86, 23, 19)

What are some commonalities in these examples?

- Every sequence is a collection of elements of some type `T` (integer, character, etc…)
- The same element may appear multiple times.
- Every sequence has a size (that may be infinite). We often write N to represent this size.
- Every element (or occurrence of an element) is assigned to an index: $0 \leq \text{index} < N$.
- Every index in the range $0 \leq \text{index} < N$ has exactly one element assigned to it.

What kind of operations can we do on a sequence[13]?

```java
public interface Sequence<T>
{
  /** Retrieve the element at a specific index or throw an IndexOutOfBounds exception if
index < 0 or index >= size() */
  public T get(int index);
  /** Obtain the size of the sequence */
  public int size();
}
```

To recap, we have a mental model, and a series of rules:

- A sequence contains N elements.
- Every index i from $0 \leq i < N$ identifies exactly one element (no more, no less).

The `get` method returns the element identified by `index`, and the `size` method returns N.

4.2.1. Sequences by Rule

For some of the example sequences listed above, we can implement this interface directly. For example, for the English alphabet:

```java
public class Alphabet implements Sequence<Character>
{
  /** Retrieve the element at a specific index */
  public Character get(int index)
  {
    if(index == 0){ return 'A'; }
    if(index == 1){ return 'B'; }
    /* ... */
    if(index == 25){ return 'Z'; }
    throw IndexOutOfBoundsException("No character at index "+index)
  }
  /** Obtain the size of the sequence */
  public int size()
  { return 26; }
}
```

Is `Alphabet` a sequence? Yes! It implements the two operations of a sequence:

[13] Note: Although several of the ADTs and data structures we’ll present throughout this book correspond to interfaces and classes from the Java standard library, `Sequence` is **not** one of them. However, it serves as a useful simplification of the `List` interface from the Java standard library that we **will** encounter later on.

- `size`: There are $N = 26$ elements.
- `get`: Every element from $0 \leq i < 26$ is assigned exactly one value.

Similarly, we can implement the Fibonacci sequence according to the Fibonacci rule $\mathrm{Fib}(x) = \mathrm{Fib}(x - 1) + \mathrm{Fib}(x - 2)$:

```
public class Fibonacci implements Sequence<Int>
{
  /** Retrieve the element at a specific index */
  public Int get(int index)
  {
    if(index < 0){
      throw IndexOutOfBoundsException("Invalid index: "+index)
    }
    if(index == 0){ return 0; }
    if(index == 1){ return 1; }
    return get(index-1) + get(index-2);
  }
  /** Obtain the size of the sequence */
  public int size()
  { return INFINITY; }
}
```

Is `Fibonacci` a sequence? Yes! It implements the two operations of a sequence:

- `size`: There are $N = \infty$ elements.
- `get`: Every element from $0 \leq i < \infty$ is assigned exactly one value derived from the value of preceding elements of the sequence as $\mathrm{Fib}(i - 1) + \mathrm{Fib}(i - 2)$[14].

4.2.2. Arbitrary Sequences

Often, however, we want to represent an sequence of **arbitrary** elements. We won't have simple rule we can use to translate the index into a value. Instead, we need to store the elements somewhere if we want to be able to retrieve them. The easiest way to store a sequence, when we know the size of the sequence up front is called an **array**.

Let's take a brief diversion into the guts of a computer. Most computers store data in a component called RAM[15]. You can think of RAM as being a ginormous sequence of bits (0s and 1s). Folks who work on computers have, over the course of many years, come to agree on some common guidelines for how to represent other types of information in bit sequences. The details of most of these guidelines (e.g., Little- vs Big-endian, Floating-point encodings, Characters and strings, etc…) are usually covered in a computer organization or computer architecture class. However, there's a few details that are really helpful for us to review.

First, we group every 8 bits into a 'byte'. A byte can take on $2^8 = 256$ different possible values. It turns out that this is enough to represent most simple printable characters. Similarly, bytes can be grouped

[14] If you're paying attention, you might notice that we're waving our hands a bit with this proof. Specifically, how do we convince ourselves that there is **exactly** one value at index i, and not zero (can $\mathrm{Fib}(i)$ return `IndexOutOfBoundsException` for $i \geq 0$), or more than 1 (will $\mathrm{Fib}(i)$ always return the same value)? Proving either of these will require a technique called **induction** that we'll come back to later in the book.

[15] What you're about to read is called the RAM model of computing. It is a blatant lie: The way computers store and access data involves multiple levels of caches, disks, and networked storage. However, the RAM model is quite useful as a starting point. We'll walk the lie back when we introduce the External Memory model towards the end of the book.

together into 2, 4, or 8 bytes, allowing us to represent more complex data like emoji, larger integers, negative numbers, or floating point numbers. Strings are a special case that we'll come back to in a moment.

For example, the ASCII character encoding[16] forms the basis for most character representations we use today, and assigns each of the letters of the English alphabet (upper and lower case, and a few other symbols) to a particular sequence of 8 bits (one byte).

Most modern computers are 'byte-addressable', meaning that every byte is given its own address. The first byte is at address 0000, followed by address 0001, then address 0002, and so forth. If we want to represent a value that takes up multiple bytes (e.g., a 4-byte integer), we typically point at the address of the first byte of the value.

4.2.3. Arrays

Let's say we have a series of values we want to store. For example, let's say we wanted to store the sequence 'H', 'e', 'l', 'l', 'o'. The ASCII code for 'H' is hexadecimal 0x48, or binary 01001000. We could store that value at one specific address in RAM. The value takes up one byte of space, so we could put the second value in the sequence 'e' (hexadecimal 0x65, or binary 10101001) in the byte right after it. We then similarly store the remaining three elements of the sequence, each in the byte after the previous one.

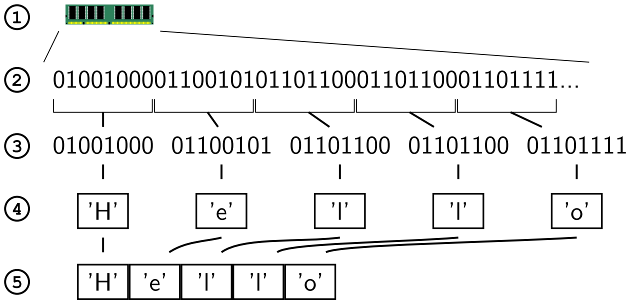


Figure 5: RAM (1) can be viewed as a sequence of bits (2). We can break these bits up into fixed size chunks (3). Each fixed size chunk can be used to identify some value, like for example a character (4). Thus a sequence of characters can be stored somewhere in RAM (5).

This arrangement, where we just concatenate elements side by side in memory is really useful, *if each element always uses up the same number of bytes*. Specifically, if we want to retrieve the ith element of the sequence, we need only two things:

1. The position of the 0th element in RAM (Let's call this S)
2. The number of bytes that each element takes up (Let's call this E).

The the ith element's E bytes will always start at byte $S + i \cdot E$ (with 0 being the first element).

4.2.3.1. Array Runtime Analysis

In order to retrieve the i'th element of an array, we need one multiplication, and one addition. This is true regardless of how big the array is. If the array has a thousand, or a million, or a trillion elements, we can still obtain the address of the ith element with one multiplication and one addition. Given the

[16] The American Standard Code for Information Interchange forms the basis for most modern character 'encodings'. More recent encodings (e.g. UTF-8, UTF-16) extend the ASCII code, and sometimes require more than one byte per printable character.

address, we can then obtain the value in a single access to memory. In other words, the cost of retrieving an element from an array is constant (i.e., $\Theta(1)$)[17].

4.2.3.2. Side Note: Memory Allocation

Most operating systems provide a way to allocate contiguous regions of memory (in C, this operation is called `malloc`, in Java it is called `new`). Eventually, when the data is no longer necessary (C's `free` operation, or based on Java's automatic 'garbage collector'), the allocated memory is returned to the general pool of 'free' memory. We will assume that both allocating and freeing memory are constant-time ($\Theta(1)$) operations[18]

It's important to note that once memory is allocated, it has a fixed size. It is not generally possible to simply resize a previously allocated chunk of memory, since that memory is located at a specific position in RAM, and it is possible that the space immediately following the array may already be in-use[19].

4.2.3.3. Arrays In Java

Java provides a convenient shorthand for creating and accessing arrays using square brackets. To instantiate an array, use `new`:

```
int[] myArray = new int[100];
```

Note that the type of an integer array is `int[]`[20], and the number of elements in the array must be specified upfront (`100` in the example above). To access an element of an array use `array[index]`.

```
myArray[10] = 29;
int myValue = myArray[10];
assert(myValue == 29);
```

Java arrays are **bounds-checked**. That is, Java stores the size of the array as part of the array itself. The array actually starts 4 bytes after the official address of the array; these bytes are used to store the size of the array. Whenever you access an array element, Java checks to make sure that the index is greater than or equal to zero and less than the size. If not, it throws an `IndexOutOfBounds` exception otherwise. As a convenient benefit, bounds checking means we can get the array size from Java.

```
int size = myArray.size
assert(size == 100)
```

Since the size is stored at a known location, we can always retrieve it in constant ($O(1)$) time.

[17] If you're paying close attention, you might note that we still need to retrieve E bytes, and so technically the cost of an array access is $\Theta(E)$. However, we're mainly interested in how runtime scales with the size of the collection, as E is fixed upfront, and usually very small. So, more precisely, **with respect to** N, the cost of retrieving an element from an array is $\Theta(1)$. The even more attentive reader might be aware of things like caches, which as previously mentioned, we're going to ignore for now.

[18] Again, this is a lie. Depending on the operating system's implementation (e.g., if it allocates pages lazily or not), allocating memory may require linear time in the size of the allocation, or non-constant (e.g., logarithmic or linear) time in the number of preceding allocations. Nevertheless, for the rest of the book, we'll treat it as being constant

[19] C provides a `realloc` operation that opportunistically *tries* to extend an allocated chunk of memory if space exists for it. However, if this is not possible, the entire allocation will be copied byte-for-byte to a new position in memory where space does exist. As we'll emphasize shortly, copying an array is a linear ($\Theta(N)$) operation.

[20] Java sort of allows you to store variable size objects in an array. For example, `String[]` is a valid array type. The trick to this is that Java allocates the actual string *somewhere else in RAM* called the heap. What it stores in the actual array is the address of the string (also called a pointer).

To prove to ourselves that the Array implements the Sequence ADT, we can implement its methods:

```
public class ArraySequence<T> implements Sequence<T>
{
  T[] data;

  public ArraySequence(T[] data){ this.data = data; }
  public T get(int index) { return data[index]; }
  public int size() { return data.size; }
}
```

The ArraySequence, initialized by an array, follows the rules for a Sequence:

- size: There are N =data.size elements in the array
- get: The ith element of the sequence is data[i].

4.2.4. Mutable Sequences

The Fibonacci sequence and the English alphabet are examples of **immutable** sequences. Immutable sequences are pre-defined and can not be changed; We can't arbitrarily decide that the 10th Fibonacci number should instead be 3. However, if the sequence is given explicitly (e.g., as an array), then it's physically possible to just modify the bytes in the array to a new value. To accommodate such edits, we can make our Sequence ADT a little more general by adding a way to modify its contents. We'll call the resulting ADT a MutableSequence[21].

```
public interface MutableSequence<T> extends Sequence<T>
{
  public void set(int index, T element)
}
```

The extends keyword in java can be used to indicate that an interface takes in all of the methods of the extended interface, and potentially adds more of its own. In this case MutableSequence has all of the methods of Sequence, plus its own set method.

A Mutable Sequence introduces one new rule:

- After we call set(i, value) with a valid index (i.e., $0 \leq i < N$), every following call to get(i) for the same index will return the value passed to set (until the next time index i is set).

4.2.4.1. Mutable Array

To prove to ourselves that the array implements the MutableSequence ADT, we can implement its methods:

```
public class MutableArraySequence<T> implements MutableSequence<T>
{
  T[] data;

  public ArraySequence(int size){ this.data = new T[size]; }
  public void set(int index, T element) { data[index] = value; }
  public T get(int index) { return data[index]; }
  public int size() { return data.size; }
}
```

[21] Note: Like Sequence, the MutableSequence is not a native part of Java. Its role is subsumed by List, which we'll discuss shortly.

As before, we can show that the array satisfies the rules on `get` and `size`, leaving the new rule:

- `set`: Calling set overwrites the array element at `data[index]` with value, which is the value returned by `get(index)`.

4.2.5. Array Summary

Observe that each of the three `MutableSequence` methods are implemented in a single, primitive operation. Thus:

- `get`: Retrieving an element of an Array (i.e., `array[index]`) is $\Theta(1)$
- `set`: Updating an element of an array (i.e., `array[index] = ...`) is $\Theta(1)$
- `size`: Retrieving the array's size (i.e., `array.size`) is $\Theta(1)$

Recall that `size` accesses a pre-computed value, stored with the array in Java.

4.3. The List ADT

Although we can change the individual elements of an array, once it's allocated, the size of the array is fixed. This is reflected in the `MutableSequence` ADT, which does not provide a way to change the sequence's size. Let's design our next ADT by considering how we might want to change an array's size:

- Inserting a new element at a specific position
- Removing an existing element at a specific position

It's also useful to treat inserting at/removing from the front and end of the sequence as special cases, since these are both particularly common operations.

We can summarize these operations in the `List` ADT[22]:

```
public interface List<T> extends MutableSequence<T>
{
  /** Append an element to the end of a list */
  public void add(T element);
  /** Insert an element at an arbitrary position */
  public void add(int index, T element);
  /** Remove an element at a specific index */
  public void remove(int index);

  // ... and more operations that are not relevant to us for now.
}
```

`List` brings with it a new set of rules:

- After calling `add(index, element)` with a valid $0 \leq \text{index} \leq N$ (note that N is an allowable index):
  - Every element previously at an index $i \geq \text{index}$ will be moved to index $i + 1$
  - The value `element` will be the new element at index index,
  - `size()` will increase by 1.
- After calling `remove(index)` with a valid $0 \leq \text{index} < N$:
  - Every element previously at an index $i > \text{index}$ will be moved to index $i - 1$
  - The value previously at index index will be removed

[22] See Java's List interface for the full list of operations supported on `Lists`. Most of these operations are so-called syntactic sugar on top of these basic operations, offering cleaner code, but no new functionality.

  - `size()` will decrease by 1.

Note that calling `add(element)` is the same as calling `add(element, size())`.

4.3.1. A simple Array as a List

We can still use `Arrays` to implement the `List` ADT. However, recall that it’s not (generally) possible to resize a chunk of memory once it’s been allocated. Since we can’t change the size of an `Array` once it’s allocated[23], we’ll need to allocate an entirely new array to store the updated list. Once we allocate the new array, we’ll need to copy over everything in our original array to the new array.

```java
public class SimpleArrayAsList<T> extends MutableArraySequence implements List<T>
{
  // data, get, set, size() inherited from MutableArraySequence

  public void add(T element){ add(size(), element); }
  public void add(int index, T element)
  {
    // Skipped: Check that index is in-bounds.
    T[] newData = new data[size() + 1];
    for(i = 0; i < newData.length; i++){
      if(i < index){ newData[i] = data[i]; }
      else if(i == index){ newData[i] = element; }
      else { newData[i] = data[i-1]; }
    }
    data = newData;
  }
  public void remove(int index)
  {
    // Skipped: Check that index is in-bounds.
    T[] newData = new data[size() - 1];
    for(i = 0; i < newData.length; i++){
      if(i < index){ newData[i] = data[i]; }
      else { newData[i] = data[i+1]; }
    }
  }
}
```

Does this satisfy our rules?

- `add`: `newData` is one larger, elements at positions `index` and above are shifted right by one position, and `element` is inserted at position `index`.
- `remove`: `newData` is one smaller, and elements at positions `index` and above are shifted left by one position.

Let’s look at the runtime of the `add` method:

- We’ll assume that memory allocation is constant-time $(\Theta(1))$[24].
- We already said that array changes and math operations are constant-time $(\Theta(1))$

[23] Again, the C language has a method called `realloc` that can **sometimes** change the size of an array… if you’re lucky and the allocator happens to have some free space right after the array. However, in this book we try to avoid relying purely on unconstrained luck.

[24] Lies! Lies and trickery! Memory allocation may require zeroing pages, multiple calls into the kernel, page faults, and a whole mess of other nonsense that scale with the size of the memory allocated. Still, especially for our purposes here, it’s usually safe to assume that the runtime of memory allocation is a bounded constant.

So, we can view the the `add` method as

```java
public void add(int index, T element)
{
  /* Theta(1) */
  for(i = 0; i < newData.size; i++)
  {
    if(/* Theta(1) */){ /* Theta(1) */ }
    else if(/* Theta(1) */){ /* Theta(1) */ }
    else { /* Theta(1) */ }
  }
}
```

Since every branch of the `if` inside the loop is the same, we can simplify the loop body to just $\Theta(1)$.

Recalling how we use Θ bounds in an arithmetic expression, we can rewrite the runtime and simplify it as:

- $T_{\text{add}(N)} = \Theta(1) + \sum_{i=0}^{\text{newData.size}} \Theta(1)$
- $= \Theta(1) + \sum_{i=0}^{N+1} \Theta(1)$ (`newData` is one bigger than the original N)
- $= \Theta(1) + (N + 2) \cdot \Theta(1)$ (Plugging in the formula for summation of a constant)
- $= \Theta(1 + N + 2)$ (Merging Θs)
- $= \Theta(N)$ (Dominant term)

If you analyze the runtime of the `remove` method similarly, you'll likewise get $\Theta(N)$.

4.4. Linked Lists

$\Theta(N)$ is not a particularly good runtime for simple "building block" operations like `add` and `remove`. Since these will be called in loops, the `Array` data structure is not ideal for situations where a `List` is required.

The main difficulty with the `Array` is that the entire list is stored in the same memory allocation. A single chunk of memory holds all elements in the list. So, instead of allocating one chunk for the entire list, we can go to the opposite extreme and give each element its own chunk.

Giving each element its own chunk of memory means that we can allocate (or free) space for new elements without having to copy existing elements around. However, it also means that the elements of the list are scattered throughout RAM. If we want to be able to find the ith element of the list (which we need to do to implement `get`), we need some way to keep track of where the elements are stored. One approach would be to keep a list of all N addresses somewhere, but this brings us back to our original problem: we need to be able to store a variable-size list of N elements.

Another approach is to use the chunks of memory as part of our lookup strategy: We can have each chunk of memory that we allocate store the address of (i.e., a pointer to) the **next** element in the list. That way, we only need to keep track of a single address: the position of the **first** element of the list (also called the list head). The resulting data structure is called a linked list. Figure 6 contrasts the linked list approach with an `Array`.

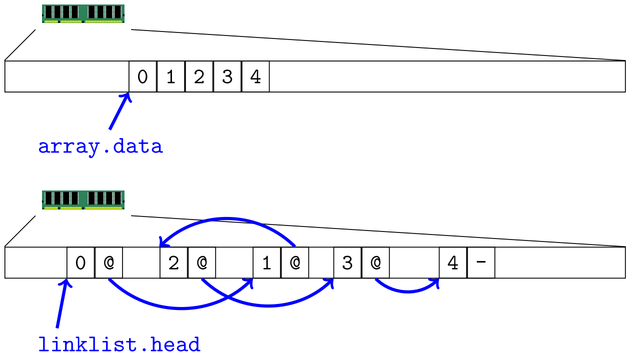


Figure 6: Instead of allocating a fixed unit of memory like an array, a linked list consists of chunks of memory scattered throughout RAM. Each element gets one chunk that has a pointer to the next (and/or previous) element in the sequence.

4.4.1. Side Note: How Java uses Memory

Java represents chunks of memory as classes, so we can implement a linked list by defining a new class that contains a value, and a reference to the next node on the list.

It's important here to take a step back and think about how Java represents chunks of memory. Specifically, primitive values (`int`, `long`, `float`, `double`, `byte`, and `char`) are stored **by value**. This means that if you have a variable with type `long`, that variable directly stores a number. If you assign the value to a new variable, the new variable stores a **copy** of the number.

```
long x = 12;
long y = x;
x = x + 5;
System.out.println(x); // prints 17
System.out.println(y); // prints 12
```

On the other hand, objects (anything that extends `Object`) and arrays are stored **by reference**. This means that what you're storing is the *address* of the actual object value (aka a pointer). When you assign the object to a new variable, what you're *really* doing is copying the address of the object or array; both variables will now point to the **same** object.

```
int[] x = new [1, 2, 3, 4, 5];
int[] y = x;
x[2] = 10;
System.out.println(y[1]); // prints 2
System.out.println(y[2]); // prints 10
System.out.println(y[3]); // prints 4
```

4.4.2. A Linked List as a List

We refer to the chunk of memory that we allocate for each element of a linked list as a `Node`. In java, we can allocate `Node` objects if we define them as a class, here storing the element represented by the `Node`, and a reference (pointer) to the next `Node`. We use java's `Optional` type to represent situations where a `Node` is not present[25]

[25] An `Optional` extends existing types to indicate that the value may be present or absent (empty). Although java already allows for object variables to take a `null` value, using `Optional` instead makes it easier for people writing code to know when it's necessary to explicitly check for a missing value (e.g., `value.isPresent()`). Since `Optional` was added to

A simple linked list, also known as a ‘singly’ linked list, is just a reference to a `head` node (which is empty if the list is empty). The last element of the list has an empty `next`.

```
public class SinglyLinkedList<T> implements List<T>
{
  class Node
  {
    T value;
    Optional<Node> next = Optional.empty();
    public this(T value) { this.value = value; }
  }
  /** The first element of the list, or None if empty */
  Optional<Node> head = Option.none();

  /* method implementations */
}
```

To help us convince ourselves that we’re actually creating a list, let’s state some rules for this structure we’re building:

1. If the list is empty then `head.isEmpty()`.
2. If the list is non-empty, then `head` refers to the 0th node.
3. The xth node stores the xth element of the list.
   - **Corrolary**: There are N nodes.
4. If the xth node is the last node (i.e., $x = N - 1$), then the node’s `next.isEmpty()`.
5. If the xth node is not the last node, then the node’s `next` points to the $x + 1$th node.

Rules about how a data structure should behave are often called **invariants**. When writing your own code and/or data structures, a great way to avoid bugs is to write down the invariants that you expect your code to follow, and then proving that your code follows the rules (“preserves the invariant”).

4.4.2.1. Linked List `get`

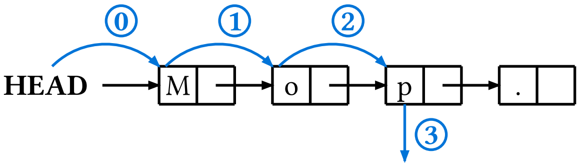


Figure 7: Retrieving the character at index 2 from a linked list representing the list ['M', 'o', 'p', '.']

To retrieve a specific element i of the list, we need to find the ith node. Since all we have to start is the 0th node (the `head`; step ⓪), we can start there. If that’s the node we’re looking for, great, we’re done; Otherwise, the only other place we can go is the 1st node (step ①). We repeat this process, moving from node to node until we get to the ith node (step ②), and return the corresponding value (step ③).

```
  public T get(int index)
  {
    if(index < 0){ throw new IndexOutOfBoundsException() }
    Option<Node> currentNode = head;
```

java, it’s been considered bad practice to use `null`. As a further side note, `Optional` can actually be viewed as a `MutableSequence`. Implementing this is left as an exercise for the reader.

```
    for(i = 0; i < index; i++){
      if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
      currentNode = currentNode.get().next;
    }
    if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
    return currentNode.get().value;
  }
```

Note that, if the rules we set for ourselves are being followed, we can prove to ourselves that this implementation is correct. Recall that we said `get(index)` is correct if it returns the indexth element of the list.

1. We start with `currentNode` assigned to `head`; By the rules we set for ourselves, this is the 0th node.
2. The ith node has a pointer to the $(i + 1)$th node, which we follow i times.
3. After i steps, `currentNode` has been reassigned to the $0 + \underbrace{1 + \ldots + 1}_{i \text{ times}} = 0 + \sum_i 1 = i$th node.
4. It follows from step 4 and the fact that the loop iterates `index` times that after the loop, `currentNode` is the indexth node.
5. It follows from step 5 and the rule that the ith node (if it exists) contains the ith element (if it exists), that the `value` in `currentNode` which `get` returns is the indexth node
6. Since a correct `get` is one that returns the indexth node, we can conclude that `get` is correct.

As usual, we can figure out the runtime of a snippet of code, starting by replacing every primitive operation with $\Theta(1)$

```
  public T get(int index)
  {
    if( /* Theta(1) */ ){ throw /* Theta(1) */ }
    /* Theta(1) */
    for(i = 0; i < index; i++){
      if( /* Theta(1) */ ){ throw /* Theta(1) */ }
      /* Theta(1) */
    }
    if( /* Theta(1) */ ){ throw /* Theta(1) */ }
    return /* Theta(1) */
  }
```

Before the `for` loop, we have only constant-time operations. Similarly, we have only constant-time operations inside the body of the loop. Both can be simplified to $\Theta(1)$ Let's start by figuring out the runtime of what's inside the for loop.

$\sum_{i=0}^{\text{index}-1} \Theta(1) = ((\text{index} - 1) - 0 + 1) \cdot \Theta(1) = \text{index} \cdot \Theta(1) = \Theta(\text{index})$

Since this is only the body of the for loop, we can add back the constant time operations before and after the loop to get the total runtime of `get`:

$T_{\text{get(index)}} = \Theta(1) + \Theta(\text{index}) + \Theta(1) = \Theta(\text{index}).$

The $\Theta(\text{index})$ runtime is a little unusual: With arrays, most costs were expressed in terms of the size of the array itself. That is, our asymptotic runtime complexity was given in terms of N, while here, it's given in terms of index. However, we know that $0 \leq \text{index} < N$, and we we can use that fact to create an analogous bound on the runtime of `get`[26].

Since $\text{index} < N$, `get` is at worst linear in the size of the array, and so we get the upper bound: $T_{\text{get}(N)} = O(N)$

Similarly, since $\text{index} \geq 0$, so `get` *could be* constant time (e.g., `get(0)` requires only constant time), and we get the lower bound: $T_{\text{get}(N)} = \Omega(1)$

4.4.2.2. Linked List set

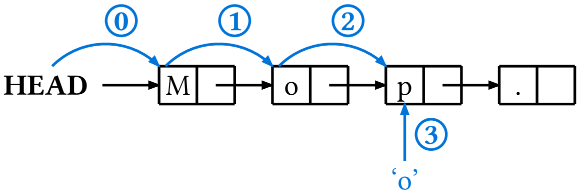


Figure 8: Updating the character at index 2 to 'o', on the linked list representing the list ['M', 'o', 'p', '.']. After the set, the linked list represents the list ['M', 'o', 'o', '.']

To set the ith element of a linked list, just like `get`, we need to first find the ith element. As a result, the code for `set` is almost exactly the same as `get`, except for the very last line.

```
public void set(int index, T element)
{
  if(index < 0){ throw new IndexOutOfBoundsException() }
  Option<Node> currentNode = head;
  for(i = 0; i < index; i++){
    if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
    currentNode = currentNode.get().next;
  }
  if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
  currentNode.get().value = element;
}
```

Using our Linked List invariants, we can similarly prove that our implementation of set follows the rules that we set out for `List`'s `set` operation. Remember that `set(index, element)` is supposed to change the linked list so that the next time we call `get(index)`, we get `element` instead of what was there before.

1. We start with `currentNode` assigned to `head`; By the rules we set for ourselves, this is the `0`th node.
2. The ith node has a pointer to the $(i + 1)$th node, which we follow i times.
3. After i steps, `currentNode` has been reassigned to the $0 + \underbrace{1 + ... + 1}_{i \text{ times}} = 0 + \sum_i 1 = i$th node.
4. It follows from step 4 and the fact that the loop iterates `index` times that after the loop, `currentNode` is the indexth node.
5. It follows from step 5 and the rule that the ith node (if it exists) contains the ith element (if it exists), that the `value` in `currentNode` which `set` changes is the indexth node

[26] The proof here is given only for *valid* inputs. For values of index less than zero, the function aborts immediately, and we can prove to ourselves (based on the rule that the last element in the list has an empty `next` node) that we will never go through the loop more than N times. Strictly speaking, the runtime should be, $\Theta(\min(N, \text{index}))$. However, this added complexity makes no immediate impact on our discussion, and so we omit it.

6. Since any subsequent calls to `get` will return the indexth node, we can conclude that `set` is correct.

Side Note: Observe how the proof for `set` is almost exactly the same as the proof for `get`.

As usual we can figure out the runtime by replacing every primitive operation with $\Theta(1)$

```
public void set(int index, T element)
{
  if( /* Theta(1) */ ){ throw /* Theta(1) */ }
  /* Theta(1) */
  for(i = 0; i < index; i++){
    if( /* Theta(1) */ ){ throw /* Theta(1) */ }
    /* Theta(1) */
  }
  if( /* Theta(1) */ ){ throw /* Theta(1) */ }
  /* Theta(1) */
}
```

Once we replace all primitive operations with $\Theta(1)$, `set` and `get` are exactly the same. Taking the same steps gets us to a runtime of $\Theta(\text{index})$, or $O(N)$.

4.4.2.3. Linked List `add`

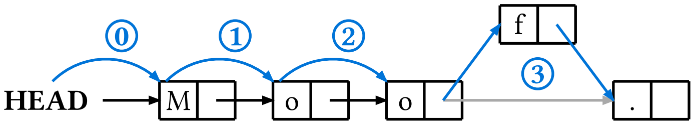


Figure 9: Inserting the character 'f' at index 3, on the linked list representing the list ['M', 'o', 'o', '.']. After the add, the linked list represents the list ['M', 'o', 'o', 'f', '.'] (the call of the noble dogcow).

Unlike the array, where we manually need to copy over every element of the array to a new position, in a linked list, the position of each node is given relative to the previous node. If we insert a new node by redirecting the `next` value of the previous node, we automatically shift every subsequent node by one position to the right.

```
public void add(int index, T element)
{
  if(index < 0){ throw new IndexOutOfBoundsException() }
  Node newNode = new Node(element);
  if(head.isEmpty){                   /* Case 1: Initially empty list */
    head = Optional.of(newNode);
  } else if(index == 0){              /* Case 2: Insertion at head */
    newNode.next = head;
    head = Optional.of(newNode);
  } else {                            /* Case 3: Insertion elsewhere */
    Option<Node> currentNode = head;
    for(i = 0; i < index-1; i++){
      if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
      currentNode = currentNode.get().next;
    }
    if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
    newNode.next = currentNode.get().next;
    currentNode.get().next = Optional.of(newNode);
    }
  }
```

Let's prove to ourselves that this code is correct, and start by recalling our rules for how `add` is supposed to behave:

1. The size of the List grows by 1.
2. The value `element` will be the new element at index index.
3. Every element previously at an index $i \geq$ index will be moved to index $i + 1$.

Let's set aside the size rule for the moment and focus on the other two rules. We have our original list (L_{old} of size N_{old}), and the list we get after calling `add` (L_{new} of size $N_{\text{new}} = N_{\text{old}} + 1$). Collectively, the latter two rules give us four cases for what will happen if we call L_{new}.`get`(i):

1. If $0 \leq i <$ `index`: L_{new}.`get`(i) = L_{old}.`get`(i)
2. If $i =$ `index`: L_{new}.`get`(i) = `element`
3. If $N_{\text{new}} > i >$ `index`: L_{new}.`get`(i) = L_{old}.`get`($i - 1$)
4. If $i \geq N_{\text{new}}$: L_{new}.`get`(i) throws an exception

We'll want to show that after calling `add`, any subsequent call to `get`(i) will return the correct value. We already proved that `get` is correct if the linked list follows the rules we set forth for ourselves, so we need to show that the new linked list is correct for the updated list. In other words, we want to show that:

1. All nodes at position $0 \leq i <$ `index` are left unchanged.
2. The newly created node (at position `index`) is pointed to by the node at position `index`-1, or by `head` if `index` $= 0$.
3. All nodes previously at positions `index` $\leq i < N_{\text{old}}$ are now at position $i + 1$.
4. The node at position $N_{\text{new}} - 1$ has an empty `next`.

The code has three cases that we'll look at individually.

Case 1: Initially empty list: Going down the list of things we need to prove:

1. There are no other nodes, so nothing changes.
2. The newly created node is at position 0, and so is correctly pointed to by `head` after the code runs.
3. There are no other nodes, so nothing needs to change.
4. Since $N_{\text{new}} = 1$, we need the node at position $N_{\text{new}} - 1 = 0$ to have an empty next. Since this is the node we just created, this is true.

Case 2: Insertion at head: Going down the list of things we need to prove:

1. Since `index` $= 0$, there can be no nodes at positions before `index`, so nothing changes.
2. The newly created node is at position 0, and so is correctly pointed to by `head` after the code runs.

For the remaining two rules, consider the fact that the (newly created) 0th node's `next` field now points at the node that used to be the `head`.

- By our rules for linked lists, this means that the old 0th element is now the 1st element (correct).
- The new 1st element is pointing at what used to be the 1st element, but is now the 2nd element (also correct).
- The new 2nd element is pointing at what used to be the 2nd element, but is now the 3rd element (also correct).
- The new 3rd element is pointing at what used to be the 3rd element, but is now the 4th element (also correct).

This pattern continues: The new ith element is pointing at what used to be the ith element, but is now the $i + 1$th element (still correct), up until we get to the $N_{\text{new}} - 1$th element, which used to be the $N_{\text{old}} - 1$th element, which still (correctly) has an empty next.

Case 3: Insertion anywhere else: Going down the list of things we need to prove:

1. The code finds the node at position index-1, skipping over all preceding nodes. These are unchanged.
2. The newly created node is pointed to by the node at position index-1, putting it at position index.

For the remaining two rules, we want to consider what happens to the node previously at position index, but also need to consider the possibility that there was no node at this index to begin with (if index $= N_{\text{old}}$). If the node exists, then as in **Case 2**, the node at position index moves to position index $+1$, likewise repositioning every following node, including the final one. If the node doesn't exist, then the newly created node's next field is assigned an empty value, (correctly) making it the last node in the list.

4.4.2.4. Aside: Invariants and Rule Preservation

It's worth taking a step back at this point and reviewing what we just did and why. We defined the expected state of the linked list (in terms of the behavior of get and our rules for a correct linked list) *after* the set, but we did so relatively to its state *before* the set. Specifically, we showed that, if the list was correct before we called set, it would still be correct (for the new list) after we called set, or in other words, we showed that set **preserves the invariants** we defined for the linked list.

Invariants are an extremely useful debugging technique for data structures (and other code). If you can precisely define one or more rules (invariants) for your data structure, you can check to see whether your code follows the rules: If you get a correct data structure as input to your function, is it always the case that you get a correct data structure as an output (keeping in mind that the function may change the definition of correctness).

4.4.2.5. Linked List add runtime

The runtime of add follows a pattern very similar to that of set:

```
public void add(int index, T element)
{
  if( /* Theta(1) */ ){ /* Theta(1) */ }
  /* Theta(1) */
  if( /* Theta(1) */ ){                /* Case 1: Initially empty list */
    /* Theta(1) */
  } else if( /* Theta(1) */ ){         /* Case 2: Insertion at head */
    /* Theta(1) */
  } else {                             /* Case 3: Insertion elsewhere */
    /* Theta(1) */
    for(i = 0; i < index-1; i++){
      /* Theta(1) */
    }
    if( /* Theta(1) */ ){ /* Theta(1) */ }
    /* Theta(1) */
  }
}
```

Based on the the outer if statement, we have three cases:

$$T_{\text{add}}(\text{index}) = \begin{cases} \Theta(1) \text{ if case } 1 \\ \Theta(1) \text{ if case } 2 \\ \Theta(1) + \sum_{i=0}^{\text{index}-1} (\Theta(1)) \text{ if case } 3 \end{cases}$$

Simplifying the third case, we get:

- $\Theta(1) + \sum_{i=0}^{\text{index}-1} (\Theta(1))$
- $\Theta(1) + (\text{index} - 1 + 0 + 1)(\Theta(1))$
- $\Theta(1) + \text{index} \cdot \Theta(1)$
- $\Theta(1) + \Theta(\text{index})$
- $\Theta(\text{index})$

So… $T_{\text{add}}(\text{index}) = \begin{cases} \Theta(1) \text{ if case } 1 \\ \Theta(1) \text{ if case } 2 \\ \Theta(\text{index}) \text{ if case } 3 \end{cases}$

Since $\text{index} = 0$ in cases 1 and 2, we can further simplify to

$$T_{\text{add}}(\text{index}) = \begin{cases} \Theta(\text{index}) \text{ if case } 1 \\ \Theta(\text{index}) \text{ if case } 2 \\ \Theta(\text{index}) \text{ if case } 3 \end{cases} = \Theta(\text{index})$$

As before, $\Theta(\text{index}) \in O(N)$ and $\Theta(\text{index}) \in \Omega(1)$

4.4.2.6. Linked List `remove`

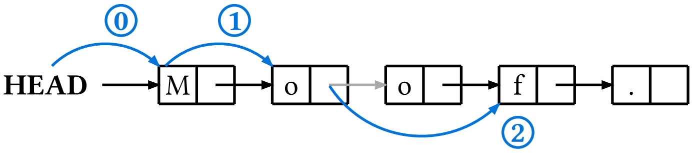


Figure 10: Removing the element at index 2 from the linked list representing the list ['M', 'o', 'o', 'f', '.']. After removal, the linked list represents the list ['M', 'o', 'f', '.']

As with `add`, we can benefit from the relative positioning of linked list nodes by simply moving a `next` reference to skip over the removed linked list node.

```
public void remove(int index)
{
  if(index < 0){ throw new IndexOutOfBoundsException() }
  if(head.isEmpty){                    /* Case 1: Initially empty list */
    throw new IndexOutOfBoundsException();
  } else if(index == 0){               /* Case 2: Deletion from head */
    head = head.get().next;
  } else {                             /* Case 3: Deletion elsewhere */
    Option<Node> currentNode = head;
    for(i = 0; i < index-1; i++){
      if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
      currentNode = currentNode.get().next;
    }
    if(currentNode.isNone){ throw new IndexOutOfBoundsException() }
    currentNode.next = currentNode.get().next;
  }
}
```

The proof of correctness and `remove`’s runtime of $\Theta(\text{index})$ are left as an exercise for the reader.

4.4.2.7. Linked List size (Take 1)

The size of the list is the number of elements. Thinking back to our rules, the last node is the node with an empty `next` pointer, so we need to figure out where this node is located

```
public int size()
{
  Option<Node> currentNode = head;
  int count = 0;
  while( ! currentNode.isEmpty() ){
    currentNode = currentNode.get().next
    count += 1
  }
  return count
}
```

Let's see if we can prove that this code is correct. This code is a bit harder to think about than `get` and `set`, since in both of those cases we had a nice handy `for` loop to keep track of how many 'steps' we take through the list. Here, the while loop just keeps going until `currentNode.isEmpty()`. When trying to tackle a tricky proof like this, it can often help to break down the proof into cases.

Let's start with the simplest case: What happens if the list is empty ($N = 0$)?

1. `count` starts off at 0.
2. If the list is empty, then by our rules for linked lists `head` is empty, so we start off with `currentNode` empty as well.
3. Since `currentNode` is empty from the start, we skip the `while` loop's body entirely.
4. `count` is never modified, and we correctly return 0.

Let's tackle the next simplest case next: What happens if the list only has 1 element ($N = 1$)

1. `count` starts off at 0.
2. `head` points to the 0th element of the list, so `currentNode` starts off pointing to the 0th node.
3. Since `currentNode` is not empty, we enter the body of the while loop.
4. We increment `count` to 1, and update `currentNode` from the 0th node to its `next` reference.
5. By our rules for linked lists, since the list has only 1 element, the 0th node's `next` reference is empty, so `currentNode` is now empty and we exit the `while` loop.
6. We correctly return `count`$= 1$,

Moving on, let's see what happens if the list has 2 elements ($N = 2$)

1. `count` starts off at 0.
2. `head` points to the 0th element of the list, so `currentNode` starts off pointing to the 0th node.
3. Since `currentNode` is not empty, we enter the body of the while loop, increment count, and update `currentNode` to its `next` reference.
4. We increment `count` to 1, and update `currentNode` from the 0th node to its `next` reference, the 1st node.
5. We increment `count` to 2, and update `currentNode` from the 1st node to its `next` reference.
6. By our rules for linked lists, since the list has only 2 elements, the 1st node's `next` reference is empty, so `currentNode` is now empty and we exit the `while` loop.
7. We correctly return `count`$= 2$,

Moving on, let's try a list with 3 elements ($N = 3$)

1. `count` starts off at 0.
2. `head` points to the 0th element of the list, so `currentNode` starts off pointing to the 0th node.
3. Since `currentNode` is not empty, we enter the body of the while loop, increment count, and update `currentNode` to its `next` reference.
4. We increment `count` to 1, and update `currentNode` from the 0th node to its `next` reference, the 1st node.
5. We increment `count` to 2, and update `currentNode` from the 1st node to its `next` reference, the 2nd node.
6. We increment `count` to 3, and update `currentNode` from the 2nd node to its `next` reference.
7. By our rules for linked lists, since the list has only 3 elements, the 2st node's `next` reference is empty, so `currentNode` is now empty and we exit the `while` loop.
8. We correctly return `count`$= 3$,

Although the proof is different for each case, you might notice a pattern forming. For a list of *any* size, we can come up with a similar proof by adding a bunch of lines into the middle, all from the template:

We increment `count` to i, and update `currentNode` from the $i - 1$th node to its `next` reference, the ith node.

In other words, it looks like the code ties the values of `count` and `currentNode` together. We can use this intuition to define a rule for ourselves: If `currentNode` is not empty, it always points to the `count` node of the list. This is true at the start of the loop, since `count`$= 0$ and by our rules for linked lists, `currentNode` points at the 0th node. Now, if we know that `currentNode` (if non-empty) points to the `count` node of the linked list when we **start** the loop body, we can show that it's still true at the end of the loop body:

1. If `currentNode` points to the ith node of the list, by our rules for linked lists, its `next` element points to the $i + 1$th node.
2. If `count` is i to start, then incrementing it sets it to $i + 1$
3. From lines 1 and 2, after the loop body, `count`$= i + 1$ and `currentNode` is the $i + 1$th node, and we're still following the rule.

Since the rule holds at the **start** of the loop, and since the loop body preserves the rule, we know that the rule is also followed at the **end** of the loop. We call a rule that is satisfied at the start of a loop and preserved by the loop body, a **loop invariant**.

The loop ends at the very first point where `currentNode` becomes empty. This happens in one of two cases:

- `head` is empty.
- The `next` element of `currentNode` is empty.

By our rules for linked lists, the first case happens only when the list is empty, and we've already shown that `size` is correct in this case. Similarly, by our rules for linked lists, the second case happens only when `currentNode` points to the last (i.e., $N - 1$th) node of the linked list. From that, and our loop invariant, it must be the case that `count` is $N - 1$. Since `count` gets incremented one last time on the 2nd line of the loop body, when the loop ends `count`$= N$, which is also correct.

4.4.2.8. Aside: Loop Invariants

Once again, let's take a quick moment to review what we just did, because it's very useful trick for debugging loops in your code[27]. We started by trying to prove that our code was correct for a couple of simple example cases. From those simple example cases, we found a pattern in the proof: a rule that our code seemed to obey throughout our proof. We then set up a **loop invariant** based on that pattern and showed that (i) it held at the start of the loop, and (ii) each line of the loop preserved the rule. Because we could show that the invariant held at the start of the loop, and each loop iteration preserved the invariant, we inferred that the invariant had to be true at the end of the loop as well. We used the fact that the invariant was true at the end of the loop to prove that `size` was correct.

When debugging code with loops, try tracing through the loop manually for a few iterations. You'll often start noticing patterns and relationships between elements of the code. Try to pin down **exactly** what that pattern is (like `currentNode` always points to the `count` node in our example), and write down the loop invariant. Then, try to prove to yourself that (i) the pattern holds at the start of the loop, (ii) the pattern is preserved by the loop body.

4.4.2.9. Linked List `size` (Take 1) runtime

As usual, we can replace all primitive operations with $\Theta(1)$

```
public int size()
{
  /* Theta(1) */
  while( /* Theta(1) */ ){
    /* Theta(1) */
  }
  return /* Theta(1) */
}
```

Once again, the `while` loop makes our job a little harder. However, just as before, we can use the loop invariant to help ourselves out:

1. count starts at 0.
2. Each loop iteration increments count by 1.
3. The loop ends when count $= N$.

Combining these facts, we can definitively state that the loop goes through N iterations. So, we get:

$$\Theta(1) + \underbrace{\Theta(1) + \ldots + \Theta(1)}_{N \text{ times}} + \Theta(1) = (N + 2) \cdot \Theta(1) = \Theta(N + 2) = \Theta(N)$$

4.4.2.10. Linked List `size` (Take 2)

It's not great when a function has a $\Theta(N)$ runtime, and it's really bad when it's something as fundamental as `size`. Many algorithms need to know the size of a collection, so it would be useful to have a way to cut the runtime to something more practical. In the case of `size`, there's one simple trick that lets us cut the runtime down to $\Theta(1)$: Precomputing the size. Let's add a `N` field to the linked list and modify the `size` method to just return it.

[27] Although it's a bit too early to use the term in the main body of the text, loop invariants are a specific example of a proof trick called "induction". We'll come back to induction as a more general technique next chapter, when we talk about recursion.

```
public class SinglyLinkedList<T> implements List<T>
{
  class Node
  {
    T value;
    Optional<Node> next = Optional.empty();
    public this(T value) { this.value = value; }
  }
  /** The first element of the list, or None if empty */
  Optional<Node> head = Option.none();

  /** The precomputed size of the list */
  private int N = 0;

  public int size() { return this.N; }

  /* method implementations */
}
```

The list starts off empty ($N = 0$), so initially this method is correct. However, we need to make sure that it stays correct as we change the structure. There are two operations, `add` and `remove`, that modify the size of a list, incrementing and decrementing the list's size. We need to modify these operations to properly maintain N:

```
  public void add(int index, T element)
  {
    /* ... as before ... */
    this.N += 1;
  }
  public void remove(int index)
  {
    /* ... as before ... */
    this.N -= 1;
  }
```

Let's review the trade-off we just made: Both changes add an additional $\Theta(1)$ runtime cost to `add` and `remove`. This doesn't change the overall code complexity of either ($\Theta(\text{index})$, or equivalently $O(N)$)[28]. In exchange, the asymptotic runtime complexity of `size` drops from $\Theta(N)$ to $\Theta(1)$. To summarize, the asymptotic runtime complexity of all the other operations stays the same, and `size`'s asymptotic runtime complexity drops massively. This is almost always a worthwhile trade.

4.5. The Iterator ADT

In the next section, we'll introduce a new type of abstract data type called the Iterator (sometimes called a 'Cursor'). Iterators are a little unique as far as this class goes, in that they represent something a bit different from a collection. They represent a loop over the elements of a collection. The ability to abstractly loop over the elements of a collection is really useful for keeping the abstraction layer around our other collection types, as we'll see shortly. We'll start with a motivating example.

[28] In practice, incrementing and decrementing an integer is a relatively inexpensive operation in most cases, so even outside of the magical land of asymptotics, this is usually a worthwhile trade. That said, there are *some* exceptions, primarily when dealing with concurrency (synchronized integers are expensive) or with random memory access (we'll talk about this later in the book).

4.5.1. Motivation: Summing up Integers

Have a look at the following code, which takes a `List` of `Integers` and computes their sum. Before you read on, try to work out the code's runtime for yourself.

```java
public int computeSum(List<Integer> list)
{
  int sum = 0;
  for(i = 0; i < list.size(); i++)
  {
    sum += list.get(i);
  }
  return sum;
}
```

Seriously, stop reading and figure out the code's runtime before you read on.

I mean it. Did you figure it out yet?

Ok… if you read on without figuring it out for yourself, you're cheating yourself of a learning experience. You've been warned thrice.

Using our normal technique of replacing primitive operations would lead you to a runtime of $\Theta(N)$, but this is actually not correct. We're allowed to replace **primitive** operations with $\Theta(1)$. However, this particular block of code relies on two non-primitive operations: the block of code invokes `list.size()` and `list.get()`. For these, instead of $\Theta(1)$, we need to "plug in" the runtime complexities that we computed before.

But what is the runtime complexity of these operations? To answer that question, we need to know which `List` data structure we're using. For arrays, which store the size explicitly, the `size` operation is $\Theta(1)$. For linked lists, our first attempt was a `size` operation that was $\Theta(N)$, but then we came up with an optimization that got us down to $\Theta(1)$. So:

$$T_{\text{size}}(N) = \begin{cases} \Theta(1) \text{ if using an array} \\ \Theta(1) \text{ if using a linked list} \end{cases} = \Theta(1)$$

Since all cases are the same, it doesn't matter which structure we're using, `size` is always $\Theta(1)$. However, the same isn't true for `get`. Recall that for the linked list, the cost of `get` depends on the index.

$$T_{\text{get}}(\text{index}) = \begin{cases} \Theta(1) \text{ if using an array} \\ \Theta(\text{index}) \text{ if using a linked list} \end{cases}$$

Since there are two possibilities here, we can't define a simple theta bound for `get`, but we can provide upper and lower bounds $O(\text{index})$ and $\Omega(1)$. Let's plug this all into our code and see what we get:

```java
public int computeSum(List<Integer> list)
{
  /* Theta(1) */
  for(i = 0; i < list.size(); i++)
  {
    /* Theta(1) + T_get(i) */
  }
  return /* Theta(1) */;
}
```

Since we don't have a (simple) tight bound for $T_{\text{get}}(i)$, we can switch to using both upper and lower bounds. Remember that wherever you see $\Theta(f(N))$, it means that $f(N)$ is **both** an upper and a lower bound, so we can always replace $\Theta(f(N))$ in an arithmetic expression, with $O(f(N))$ or $\Omega(f(N))$. Solving for the upper bound first, we get:

- $T_{\text{computeSum}}(N) = O(1) + \left(\sum_{i=0}^{N} O(1) + T_{\text{get}}(i)\right) + O(1)$
- $= O(1) + \left(\sum_{i=0}^{N} O(1) + O(i)\right) + O(1)$
- $= O(1) + \left(\sum_{i=0}^{N} O(i)\right) + O(1)$
- $= O(1) + \left(\sum_{i=0}^{N} i \cdot O(1)\right) + O(1)$
- $= O(1) + O(1) \cdot \left(\sum_{i=0}^{N} i\right) + O(1)$
- $= O(1) + O(1) \cdot \left(\frac{N \cdot (N+1)}{2}\right) + O(1)$
- $= O(1) + O(1) \cdot \left(\frac{N^2}{2} + \frac{N}{2}\right) + O(1)$
- $= O(1) + O\left(\frac{N^2}{2} + \frac{N}{2}\right) + O(1)$
- $= O(1) + O(N^2) + O(1)$
- $= O(N^2)$

And then for the lower bound:

- $T_{\text{computeSum}}(N) = \Omega(1) + \left(\sum_{i=0}^{N} \Omega(1) + T_{\text{get}}(i)\right) + \Omega(1)$
- $= \Omega(1) + \left(\sum_{i=0}^{N} \Omega(1) + \Omega(1)\right) + \Omega(1)$
- $= \Omega(1) + \left(\sum_{i=0}^{N} \Omega(1)\right) + \Omega(1)$
- $= \Omega(1) + \Omega(1) \cdot \left(\sum_{i=0}^{N} 1\right) + \Omega(1)$
- $= \Omega(1) + \Omega(1) \cdot (N - 0 + 1) + \Omega(1)$
- $= \Omega(1) + \Omega(N + 1) + \Omega(1)$
- $= \Omega(1) + \Omega(N) + \Omega(1)$
- $= \Omega(N)$

So `computeSum` will take **at least** linear time, but could take as much as quadratic time!

If we work our way backwards through the summation, the limiting factor seems to be T_{get}. If T_{get} were constant-time, for example if we were certain that `list` was an array, we could re-do the summations with $T_{\text{get}} = O(1)$. Try it yourself, and you'll see that the upper bound on the runtime works out to $O(N)$. This is **much** better. As we saw in the last chapter, for a sufficiently large list, quadratic to linear can be the difference between an algorithm that takes seconds, and one that takes hours.

So, how do we fix it? One strategy might be to observe that you can loop over the elements of a linked list **much** faster if you have some insight into the list's structure. Instead of using a loop variable `i`, you can use a pointer to the 'current' node:

```
public int computeSumOfLinkedList(LinkedList<Integer> list)
{
  int sum = 0;
  for(list.Node currentNode = list.head;
      currentNode.isPresent();
      currentNode = currentNode.get().next)
  {
    sum += currentNode.get().value;
  }
```

```
    return sum;
  }
```

In this version of the algorithm, every operation is primitive. Note that the algorithm looks almost identical to that of `size` (take 1). The proof that this this algorithm also runs $\Theta(N)$ is similar, and is left as an exercise for the reader. Most importantly, the $\Theta(N)$ we get from this algorithm is much better than the $O(N^2)$ we got from the "generic" version of the algorithm that we saw at the start of the section. However, this improved performance comes at a steep cost: The new algorithm is specialized to linked lists. If we need to change to a different data structure (e.g., the Buffered Array that we'll introduce shortly), then we have to rewrite the summation algorithm from scratch.

4.5.2. Abstracting Loops

The motivating problem goes beyond just computing sums: **Any** loop over the elements of a linked list using `get` is going to have an $O(N^2)$ runtime. We can play a similar trick, looping over the linked list nodes instead of the index, for any loop. However, doing so requires us to know that we're working with a linked list. So let's dig a little deeper.

To move from one index to the next, the generic linked list loop simply increments i to $i + 1$ (a constant-time operation). For the array, where the index is sufficient to find the value in constant time, the overall cost of stepping from one element of the list to the next is constant. However, for a linked list, going from an index to a value is a linear-time operation, since we need to find the node corresponding to the index. The 'trick' underlying the specialized linked list version of the loop, is that moving from a linked list **node** to the `next` node can be done in constant time. The structure of the loop is the same, but instead of using an index i, we use a node reference.

Looking at the commonalities between `computeSum` and `computeSumOfLinkedList`, the following code structure is common to both:

```
  public void computeSumV2(List<Integer> list)
  {
    int sum = 0;
    for( PositionType i = list.firstPosition();
         i.isNotAtEnd();
         i.moveToNext())
    {
      sum += i.getValue();
    }
  }
```

This code structure includes five placeholders:

1. The type of the position reference (`PositionType`).
2. A way to initialize the position reference to the first element (`firstPosition`).
3. A way to test whether there are more positions left to visit (`isNotAtEnd`)
4. A way to move to the next position (`moveToNext`)
5. A way to get the value at the current position (`value`)

	`computeSum`	`computeSumOfLinkedList`
`PositionType`	`int`	`list.Node`

	computeSum	computeSumOfLinkedList
firstPosition	0	list.head
isNotAtEnd	i < N	i.isPresent()
moveToNext	i += 1	i = i.get().next
getValue	list.get(i)	i.get().value

4.5.3. The Iterator ADT

The `Iterator` abstract data type, abstractly models the position of a loop in a list, providing the five operations named above. Iterators are implemented in each language slightly differently, but in Java, the main parts of the `Iterator` interface look like:

```
public interface Iterator<E>
{
  /**
   * Returns true if there are more values to retrieve
   * (The `isAtEnd` of our example above)
   */
  public boolean hasNext();
  /**
   * Retrieves the next element and advances the iterator
   * to the next position.
   * (Combines `moveToNext` and `getValue` from our example)
   */
  public E next();

  /* A few other methods that aren't relevant yet */
}
```

To initialize the iterator, the `List` interface defines an `iterator` method that returns an iterator pointing to the first position (`firstPosition` in our example). Plugging this into our template pattern, we get:

```
  public void computeSumWithIterators(List<Integer> list)
  {
    int sum = 0;
    for( Iterator<Integer> i = list.iterator();
         i.hasNext();
         /* next() automatically advances the iterator */)
    {
      sum += i.next();
    }
    return sum;
  }
```

4.5.3.1. Case Study: Array Iterator

To build an iterator, we can look at the table above and plug in the corresponding operations for `hasNext` and `next`. For the array, this means using the index.

```
public class SimpleArrayIterator<T> extends Iterator<T>
{
  SimpleArrayAsList<T> array;
  int i = 0;

  /** `firstPosition` */
  public this(SimpleArrayAsList<T> array) { this.array = array }
  /** `isNotAtEnd` */
  public boolean hasNext() { return i < array.size; }
  /** `getValue` and `moveToNext` */
  public T next() {
    T currentValue = array.get(i);
    i += 1;
    return currentValue;
  }
}
```

4.5.3.2. Case Study: Linked List Iterator

For the linked list, this means using the linked list node.

```
public class SinglyLinkedListIterator<T> extends Iterator<T>
{
  Optional<Node> currentNode;

  /** `firstPosition` */
  public this(SinglyLinkedList<T> list) { this.currentNode = list.head; }
  /** `isNotAtEnd` */
  public boolean hasNext() { return currentNode.isPresent(); }
  /** `getValue` and `moveToNext` */
  public T next() {
    T currentValue = currentNode.get().value;
    currentNode = currentNode.get().next;
    return currentValue;
  }
}
```

4.5.4. Summation Runtime with Iterators

Note that, each of these implementations use exclusively primitive operations. As a result, both iterators' initialization, `hasNext` and `next` are constant-time. Returning to the iterator-based implementation of `computeSum`, we can figure out the runtime:

```
  public void computeSumWithIterators(List<Integer> list)
  {
    /* Theta(1) */
    for( /* Theta(1) */;
         /* Theta(1) */;
         /* 0 */)
    {
```

```
    /* Theta(1) */
  }
  return /* Theta(1) */
}
```

This gives us a runtime of:

$$T_{\text{computeSumWithIterators}}(N) = \Theta(1) + \left(\sum_{\{i=0\}}^{\{N-1\}} \Theta(1)\right) + \Theta(1) = \Theta(N)$$

Using iterators, we get the same $\Theta(N)$ runtime for a loop over either list implementation, but we get the benefit **without having to specialize the loop implementation**.

The resulting code is far more flexible, since we can freely swap in **any** other data structure implementation, without impacting the code's correctness (or, in this case, performance).

4.5.4.1. Java Iterator shorthand

Looping over the elements of a collection with iterators is such a common pattern that Java adds a bit of syntax to make it easier to write. Note the `for` syntax below.

```
public void computeSumWithIteratorsV2(List<Integer> list)
{
  int sum = 0;
  for(Integer i : list)
  {
    sum += i.next();
  }
  return sum;
}
```

`computeSumWithIterators` and `computeSumWithIteratorsV2` do exactly the same thing.

```
for(T element : collection){ /* ... */ }
```

is simply a shorthand for:

```
for(Iterator<T> iter = collection.iterator(); iter.hasNext(); ){
  T element = iter.next();
  ...
}
```

4.5.5. List Access by Reference

Iterators have another benefit. They give us a highly efficient way to reference a **specific** element of a list.

Remember that for Arrays, we can efficiently identify an element by its index. For any operation that we want to perform on specific elements of the Array, we can use the list index to identify the element we want to affect. For example, let's talk about the following bit of code, which deletes every instance of an element in the list.

```
public void removeAll<T>(List<T> list, T element)
{
  for(i = 0; i < list.length(); i++){
    if(list.get(i).equals(element)){
      list.remove(i);
      i -= 1;
    }
  }
}
```

The code iterates through every element of the list, and then calls `remove` on the element being removed. In the implementation above, we use the index (`i`) to refer to a specific element. However, remember that that this is exactly the type of loop that led us to come up with iterators, since it performs quadratically if `list` is a `LinkedList`. Let's replace the code with an iterator[29].

```
public void removeAllWithIterator<T>(List<T> list, T element)
{
  int i = 0;
  for(Iterator<T> iter = list.iterator(); iter.hasNext();){
    if(iter.next().equals(element)){
      list.remove(i);
    } else {
      i++;
    }
  }
}
```

Note that we can't get rid of the list index (`i`), since we need a way to tell `remove` which element to remove. However, this has an unexpected consequence on performance: If `list` is a `LinkedList`, `remove` has a runtime complexity of $\Theta(i)$. If we do a runtime analysis of removeAllWithIterator, plugging in the cost of `remove` on a `LinkedList`[30], we start with:

```
public void removeAllWithIterator<T>(List<T> list, T element)
{
  /* O(1) */
  for(/* O(1) */){
    if( /* O(1) */){
      /* O(i) */
    } else {
      /* O(1) */
    }
  }
}
```

The `for` loop visits every element of the list. Although `i` isn't incremented for **every** element of the list, we can use the number of elements we've visited as a *upper bound* on `i`. That is: $i \in O(v)$ (where v is the number of visited nodes. Using this insight, we can compute the total runtime (with N as the `length` of `list`) as:

$$T_{\text{removeAllWithIterator}}(N) = O(1) + O(1) + \sum_{v:0}^{N} O(1) + \begin{cases} O(i) \\ O(1) \end{cases}$$

The cases block is upper-bounded by $O(i)$, and $O(i) + O(1) = O(i)$, so

$$T_{\text{removeAllWithIterator}}(N) = O(1) + O(1) + \sum_{v:0}^{N} O(i)$$

[29] If you try running `removeAllWithIterator` for real, you'll notice that it actually won't work. Because `remove` changes the list, it also 'invalidates' all iterators on the list. Keep reading on for a way to implement this algorithm without breaking the iterator.

[30] The analysis is only slightly different if you plug in the $O(N)$ cost of `remove` for an `Array`.

As we noted above, $i \in O(v)$, so:

$$T_{\text{removeAllWithIterator}}(N) = O(1) + O(1) + \sum_{v:0}^{N} O(v)$$

Applying our summation rules, we can simplify the summation:

$$T_{\text{removeAllWithIterator}}(N) = O(1) + O(1) + O(N^2)$$

And one more simplification step gets us:

$$T_{\text{removeAllWithIterator}}(N) = O(N^2)$$

$O(N^2)$ isn't great, so let's take a step back and ask ourselves why that's the runtime complexity. We need to do a loop over N elements just to find the elements that don't fit, so it'll be difficult to get the runtime under $O(N)$[31]. However, in every iteration of the loop, we have the $O(i)$ call to `remove`.

That $O(i)$ factor is there because it takes us `i` steps to find the linked list node for the `i`th element. However, remember that **once we have the ith linked list node, actually removing the element is $O(1)$.** To understand why that bit of information is useful, let's remember that the the `LinkedList` iterator works by storing a reference to the current linked list node. The variable `iter` already stores a reference to the `i`th linked list node! Meanwhile, here we are in this algorithm, repeatedly looking for an `i`th node that we already have.

To address this performance issue, `Java` iterators include a `remove` method that removes the element most recently returned by `next`. Our algorithm changes only slightly:

```
public void removeAllWithIteratorV2<T>(List<T> list, T element)
{
  for(Iterator<T> iter = list.iterator(); iter.hasNext();){
    if(iter.next().equals(element)){
      iter.remove();
    }
  }
}
```

The implementation of `remove` for an Array-based `Iterator` remains $O(N)$ (since the array still needs to shift elements at indices above i to the left). However, for a `LinkedList`-based iterator, the `remove` method drops to $O(1)$, since the iterator already has a *reference* to the linked list node.

Although `remove` is the only method that appears in an iterator, you can see that nearly every other operation on a `LinkedList` (`add`, `set`) is $O(i)$ for essentially the same reason: `List` uses the list position to identify elements, and it costs $O(i)$ to find the linked list node for the `i`th element. However, if we already have a reference to the `i`th element (technically the $i - 1$th element), each of these operations can be performed in $O(1)$.

Because of this unusual behavior, we usually give **two** separate runtimes for `LinkedList` operations: One runtime for the operation applied to a given index (on the `List` itself) and one runtime for the operation applied to a reference (e.g., on the `Iterator`).

1. `get(i)`
   - **By Index**: $O(i)$

[31] This is a tiny lie, but true for Arrays and Linked Lists. We'll get to efficiently finding elements in the sections on Trees and Hash Tables.

- **By Reference**: $O(1)$

2. `set(i, v)`
   - **By Index**: $O(i)$
   - **By Reference**: $O(1)$
3. `add(i, v)`
   - **By Index**: $O(i)$
   - **By Reference**: $O(1)$
4. `remove(i)`
   - **By Index**: $O(i)$
   - **By Reference**: $O(1)$

4.5.6. Doubly Linked Lists

In the simple `LinkedList` we've described so far, often called a **Singly** linked list, each node contains a pointer to the next node in the chain (or `null` if it's the last node). This is convenient for moving *forwards* through the list, but sometimes it's convenient to be able to go *backwards* through the list as well. For example, try implementing the `LinkedList` iterator's `remove` method with a singly linked list, and you'll find that you need to keep track of several of the preceding nodes.

In a **Doubly** linked list, each linked list node also keeps a pointer to the previous node, and the linked list itself usually keeps pointers to the first *and last* nodes in the list. Although the extra pointers require extra book-keeping, all of the extra work is constant time. As a result, a doubly linked list's operations have asymptotic runtime complexities that are at least as good as a singly linked list. In fact, the `add` (i.e., append) operation becomes $O(1)$, since we have a pointer to the last element in the list (and so can access it *by reference*).

Moving forward, **we will assume that every linked list is a Doubly Linked List**.

4.6. Array List, take 2 (Buffered Arrays)

There are many situations where it's useful to be able to append to a list. As we already discussed, Java implements append as a special case of `add` that omits the `index` argument. For example, if we have a `List` of log entries, new log entries will always be added to the end. Similarly, implementations of some ADTs like `Stack` and `Queue`, which we'll discuss later, involve putting new elements at the end of a `List`.

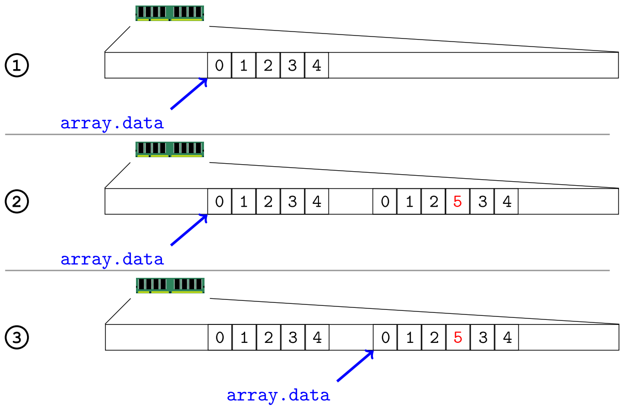


Figure 11: Adding an item to an array requires (1) allocating a new space for the array, (2) copying the contents of the old array to the new space, and (3) updating the data pointer to the new space.

As we just discussed, we can append to a doubly linked list in $O(1)$, but let's come back to the humble array, for which the `add` operation is shown in Figure 11. Thinking back, remember that the cost of `add` comes from two general tasks:

1. Copying every element of the array to a new array with one extra space: $O(N)$
2. Shifting every element of the array to the right by one spot: $O(N - i)$

In the case of append, we're inserting at the end; In other words $i = N$, and the cost of second step is $O(N - N) = 0$. Since we're inserting at the last element, nothing needs to be moved over to make space for the new element. However, we can't directly benefit from this observation, since we still need to copy every element into a new array. So let's think how we might be able to avoid that copy.

4.6.1. Buffered Arrays (Attempt 1: Fixed Increment)

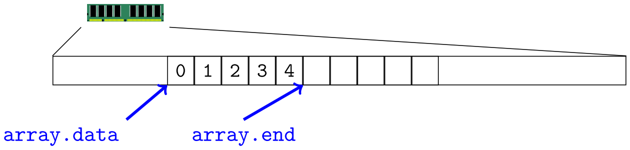


Figure 12: An `ArrayList` allocates extra space to efficiently support append operations. The example above shows an `ArrayList` with 5 elements in use, and 5 unused spaces. The structure keeps track of both how many elements in the array are in-use, and how big the allocated array is.

One idea might be to set aside a little bit of extra space in the array when we first allocate it. Let's say that when we're asked to create an N element array, we instead allocate space for $N + B$ elements (Figure 12). The extra B elements are held in reserve until `add` is called. Now, the next B calls to `add` are free, and even after that, we can still allocate another B elements. Here's a simplified version of this idea with the one argument `add` implemented:

```java
public class FixedIncrementArrayList<T> implements List<T>
{
  static int B = 10;
  T[] data = new T[0];
  int size = 0;
  /* Other methods omitted */

  public void add(T elem)
  {
    if(size >= data.length){
      T[] temp = new T[data.length + B];
      for(i = 0; i < data.length; i++){
        temp[i] = data[i];
      }
      data = temp;
    }
    data[size] = elem;
    size += 1;
  }
}
```

If we're out of space (`size >= data.length`), we allocate a new array (`temp`) that is B elements bigger, and copy every element of `data` into it. After we replace the old array with the new one, we have free space. Regardless of whether we had to allocate more space or not, we put the new `elem` into the first free slot, and shift `size` over by one step. It should be clear that, in the common case (which happens in $\frac{B-1}{B}$ calls to `add`), this data structure will be faster than `Array`. But let's dig deeper and do a proper analysis. Tagging every step in the algorithm with the runtime, we get:

```
  public void add(T elem)
  {
    if(/* O(1) */){
      /* O(1) */
      for(/* O(1), looping with i from 0 to N */){
        /* O(1) */
      }
      /* O(1) */
    }
    /* O(1) */
    /* O(1) */
  }
```

Summing this up, we get:

$$\left(\begin{cases} O(1)+\left(\sum_{i=0}^{N} O(1)\right)+O(1) \\ O(1) \end{cases}\right) + O(1) + O(1)$$

We can apply the summation rule for summation of a constant, and get rid of some of the redundant $O(1)$s

$$\left(\begin{cases} O(N) \\ O(1) \end{cases}\right) + O(1)$$

Remembering that Big-O is an *upper* bound, we take the worse case

$$O(N) + O(1) = O(N)$$

So, from a worst-case perspective, it seems as though we haven't actually improved anything. That makes sense: On **any** call to `add`, it's possible that we might trigger the worst case behavior, and have to pay the $O(N)$ cost. On the other hand, $B - 1$ out of every B calls only have to pay $O(1)$, so while any **one** call to `add` might behave badly, maybe this leads to better performance on average?

4.6.2. Amortized Analysis

A large part of the reason that asymptotic runtime complexity is useful is that it allows us to plan out our algorithms. Big-O, a.k.a. "worst-case" run-times are a safe bet for that planning process, since you know that it can't be any worse. However, there are some cases where strictly using Big-O ends up being too conservative, and you actually end up with an algorithm that has better performance than following the rules of worst-case analysis would lead you to believe. On the other hand, just giving up and saying "well, the analysis says X, but it's usually faster" isn't helpful when you're trying to be precise.

In short, we still want to be able to make *some* formal claim about the runtime of this new `add` function that will help us to understand its behavior, but Big-O isn't the right language to use for that. We need to define some new terms[32]. Intuitively, even if every B'th call to `add` is slow, most calls to `add` are fast. If we're trying to prove something, maybe we can show that these fast calls "average" out to a faster runtime if we call `add` repeatedly. For example, let's take the following loop, which just appends L elements to a list of some sort:

```java
public void addLots(List<Integer> list, int L)
{
  for(i = 0; i < L; i++)
  {
    list.add(i);
  }
}
```

Let's start off by seeing what happens if we assume that `list` starts off as an empty `SimpleArrayAsList`.

```java
public void addLots(List<Integer> list, int L)
{
  for(/* O(1), looping with i from 0 to L */)
  {
    /* O(N) = O(i) */;
  }
}
```

For a simple array, appending a new element is $O(N)$. If we start off with an empty array, then on the `i`th insertion, $N = i$, so the add costs $O(i)$. This gives us a total runtime of:

$$T_{\text{addLots1}}(L) = O(1) + \sum_{i=0}^{L} (O(1) + O(i))$$

Plugging in our standard summation formula and simplifying, we get:

$$T_{\text{addLots1}}(L) = O(1) + O(L^2) = O(L^2)$$

Now let's try the same thing with `FixedIncrementArrayList`. If we do it naively, we get *exactly* the same result, since `add` is $O(N) = O(i)$. However, this specific loop gives us a pattern we can benefit

[32] Amortized complexity, which we discuss here, is one of two relaxations of simple, or "unqualified" asymptotic complexity. We'll talk about Expected complexity in the next chapter, as we introduce Quick Sort.

from: We know that every B calls to `add` will be $O(N)$ and every other call will be $O(1)$. The summation looks like this (assuming L is divisible by B):

$$T_{\text{addLots2}}(L) = \underbrace{\left(\overbrace{O(1) + ... + O(1)}^{(B-1)\text{ times}} + O(B)\right) + ... + \left(\overbrace{O(1) + ... + O(1)}^{(B-1)\text{ times}} + O\left(\frac{L}{B} \cdot B\right)\right)}_{\frac{L}{B}\text{ times}}$$

Collapsing the $(B - 1)$ times, we get:

$$T_{\text{addLots2}}(L) = \underbrace{(O(B-1) + O(B)) + (O(B-1) + O(2 \cdot B)) + ... + \left(O(B-1) + O\left(\frac{L}{B} \cdot B\right)\right)}_{\frac{L}{B}\text{ times}}$$

We can rearrange these terms to get:

$$T_{\text{addLots2}}(L) = \left(\underbrace{O(B-1) + ... + O(B-1)}_{\frac{L}{B}\text{ times}}\right) + \left(\underbrace{O(1 \cdot B) + O(2 \cdot B) + O(3 \cdot B) + ... + O\left(\frac{L}{B} \cdot B\right)}_{\frac{L}{B}\text{ times}}\right)$$

And again, simplifying, we get:

$$T_{\text{addLots2}}(L) = O\left(\frac{L}{B} \cdot (B-1)\right) + \sum_{i=1}^{\frac{L}{B}} O(i \cdot B) = O(L) + O(B) \cdot \sum_{i=1}^{\frac{L}{B}} O(i)$$

Plugging in our summation formula and simplifying, we get

$$T_{\text{addLots2}}(L) = O(L) + O(B) \cdot O\left(\frac{L^2}{B^2}\right) = O(L) + O\left(\frac{L^2}{B}\right) = O\left(\frac{L^2}{B}\right)$$

This should make intuitive sense: Since we're only paying the cost of an expensive copy on every B insertions, the cost of L appends into an empty `FixedIncrementArrayList` is $\frac{1}{B}$th of the cost of the same appends into a simple array list. If we average the individual cost of each of the L calls, each individual call "behaves" like its runtime is $O\left(\frac{L}{B}\right)$.

Put another way, in the worst possible case, `FixedIncrementArrayList.add` is $O(N)$. However, if we're specifically analyzing the behavior of `add` when it is called in a loop, it's actually 100% safe to pretend like `add` is $O\left(\frac{N}{B}\right)$.

We want a way to talk about this sort of behavior, where an operation on a data structure has a better runtime when it's called in a loop: A way to distinguish the 'looping' runtime complexity from the normal runtime complexity. The technical term for this better runtime is the "**amortized** runtime complexity" of the algorithm.

Formally, if the amortized runtime complexity of an operation is $O(f(N))$, then the normal, or "unqualified" runtime complexity[33] of N calls to the operation are guaranteed to be $O(N \cdot f(N))$.

For the running example, we would say that:

- The **unqualified** runtime complexity of `add` for `FixedIncrementArrayList` is $O(N)$

[33] If the term "unqualified" seems a bit strange here, it's because the term "amortized" is a *qualifier* that modifies the meaning of runtime complexity. "un-qualified" doesn't mean that it's not good at anything, but just that we're talking about the runtime complexity without any qualifiers. Next chapter, we'll introduce one more type of qualifier: "expected."

- The **amortized** runtime complexity of `add` for `FixedIncrementArrayList` is $O\left(\frac{N}{B}\right)$

Note that, it's possible to describe amortized complexity bounds as being “tight”, if we know that we can't get a better bound. It's worth noting that the **tight amortized runtime complexity is never worse than the tight unqualified runtime complexity** of an algorithm, since invoking an algorithm in a loop can not make it slower. Formally, if the unqualified runtime is $O(f(N))$, then N invocations is $\sum_{i=0}^{N} O(f(N)) = O(N \cdot f(N))$, so by our definition above, the amortized runtime can not be worse than $O(f(N))$.

4.6.3. Buffered Arrays (Attempt 2: Exponential Increment)

(Alternative title: Try this one neat trick to get your asymptotic runtime bound down to $O(1)$)

The amortized runtime complexity of `add` for `FixedIncrementArrayList` is better than that of `SimpleArrayAsList`. However, in most cases B is a constant, and so asymptotically $O(N)$ and $O\left(\frac{N}{B}\right)$ are the same. Put another way, doubling N still doubles the runtime, even under amortized analysis. However, as it turns out, if we're a bit more clever about how we resize the array, we can actually push the asymptotic bound down to a lower complexity class: $O(1)$. The trick here is that, instead of adding a fixed size to the array (B) whenever we resize it, we always double the size of the array instead. This is the idea behind the `ArrayList` that is implemented in the Java standard library.

```
public class ArrayList<T> implements List<T>
{
  T[] data = new T[1];
  int size = 0;

  /* Other methods omitted */

  public void add(T elem)
  {
    if(size >= data.length){
      T[] temp = new T[data.length * 2]; // Double the array size
      for(i = 0; i < data.length; i++){
        temp[i] = data[i];
      }
      data = temp;
    }
    data[size] = elem;
    size += 1;
  }
}
```

Analyzing the amortized runtime of this new version of `add` is a little trickier than the previous two instances of Array-based lists, but we can use the same strategy that worked with `FixedIncrementArrayList`: (i) Plug `ArrayList` into `addLots`, (ii) Rearrange terms to put the fast and slow cases together, (iii) Simplify. Let's start by figuring out how ‘slow’ calls there are, where we need to copy.

- The first `add` call is $O(1)$ (array length 1).
- The 2nd `add` call requires a copy (array length 2).
- The 3rd `add` call requires a copy, and the next 1 call is $O(1)$ (array length 4).
- The 5th `add` call requires a copy, and the next 3 calls are $O(1)$ (array length 8).

- The 9th `add` call requires a copy, and the next 7 calls are $O(1)$ (array length 16).
- The 17th `add` call requires a copy, and the next 15 calls are $O(1)$ (array length 32).

The general pattern here is that the ith copy occurs on the $(2^i + 1)$th call to `add`. All the remaining calls are cheap. Put another way, after 2^x calls to `add`, we will have resized the array $x - 1$ times. So, if we want to call `add` N times, we can set up an equation to solve for x:

$N = 2^x$

Log is the inverse of exponent, so:

$\log_2 N = \log_2 2^x = x$

Plugging this back into our formula ($x - 1$), we find that we will have resized the array $\log_2(N) - 1 = O(\log_2(N))$ times. On the ith time we need to resize the array, the array will have 2^{i-1} elements in it. So, we can figure out the total runtime of the calls to add as:

$$T_{\text{addLots3}}(L) = \underbrace{O(1) + 2O(1) + 4O(1) + ... + (N-1)O(1)}_{\log_2(N)-1 \text{ times}} + \underbrace{O(1) + O(1) + ... + O(1))}_{N-(\log_2(N)-1) \text{ times}}$$

We can summarize this as:

$$T_{\text{addLots3}}(L) = \left(\sum_{i=0}^{\log_2(N)-1} 2^i O(1)\right) + \left(\sum_{i=0}^{N-(\log_2(N)-1)} O(1)\right)$$

Factoring out the $O(1)$ terms and expanding out the summations, we get:

$$T_{\text{addLots3}}(L) = O(1) \cdot \left(2^{\log_2(N)-1+1} - 1\right) + O(1) \cdot (N - \log_2(N) + 2)$$

Simplifying and noting that $2^{\log_2(N)} = N$, we get

$$T_{\text{addLots3}}(L) = (O(N) - O(1)) + (O(N) - O(\log_2(N)) + O(1))$$

And since the dominant term here is $O(N)$, the whole thing reduces to

$$T_{\text{addLots3}}(L) = O(N)$$

Remember that `addLots` calls `add` N times, so each individual call ‘behaves’ like it is $\frac{O(N)}{N} = O(1)$. In other words, the **amortized** runtime complexity of `add` is $O(1)$! It’s left as an exercise for you to prove that the unqualified runtime complexity of `add` is still $O(N)$, so this might feel a bit weird: N calls to an $O(N)$ operation should normally be $O(N^2)$, but in this specific case, it’s actually $O(N)$, a substantially lower complexity class. What is it about exponentially growing the array, doubling its size every time, that makes this possible?

The intuition here is that every time you double the size of the array from N to $2N$ you’re putting in $O(N)$ work to copy elements, but you can now do N insertions before you need to copy again. This is true, even as N grows. Each time, no matter how big N gets, $O(N)$ work gets you another N insertions. The $O(N)$ work “amortizes” over the next N insertions, and the ratio of cost to insertions stays the same. Contrast this with the Fixed-Increment Array List, where increasing the size of the array from N to $N + B$ took N work, but only provided space for another B insertions. N grows, while B stays the same, and the ratio of cost to insertions keeps growing.

Note: It can be tempting to think that, if a data structure promises an $O(1)$ amortized runtime for an operation, you can safely treat the operation as $O(1)$ in all cases. However, as exemplified in `ArrayList`, the cost of array resizing can actually get quite high. Even though the cost amortizes over multiple calls, there are many use cases where `ArrayList` is not appropriate. For example, let's say you're building a high-performance scientific data sensing application, which needs to be able to append sensor readings with sub-microsecond latencies. `ArrayList` would **not** be appropriate for such an application, because some sensor readings would have to block on the array list while it resizes, violating the quality of service guarantees.

4.7. Recap

To summarize, we have introduced two ADTs: `Sequence` and the more general `List`; as well as three main data structures: The `Array`, the `LinkedList`, and the `ArrayList`. Overall asymptotic runtime bounds for these structures, assuming a Doubly-Linked list with a pointer to the last element, are as follows:

	`Array`	`LinkedList` (by Idx)	`LinkedList` (by Ref)	`ArrayList`
`get(i)`	$O(1)$	$O(i)$	$O(1)$	$O(1)$
`set(i,v)`	$O(1)$	$O(i)$	$O(1)$	$O(1)$
`add(i,v)`	$O(N)$	$O(i)$	$O(1)$	$O(N - i)$ amortized, $O(N)$ unqualified
`add(v)`	$O(N)$	$O(1)$	$O(1)$	$O(1)$ amortized, $O(N)$ unqualified
`remove(i)`	$O(N)$	$O(i)$	$O(1)$	$O(N - i)$

Chapter 5. *Recursion, Divide and Conquer, Sorting*