*David Liu*

# Data Structures and Analysis

Lecture Notes for CSC263 (Version 0.2)

Department of Computer Science
University of Toronto

*These notes are based heavily on past offerings of CSC263, and in particular the materials of François Pitt and Larry Zhang.*

*I also thank CSC263 students from Fall 2016 and Daniel Zingaro for pointing out numerous typos in older versions of the notes.*

# *Contents*

# 1 *Introduction and analysing running time*

Before we begin our study of different data structures and their applications, we need to discuss how we will approach this study. In general, we will follow a standard approach:

1. Motivate a new abstract data type or data structure with some examples and reflection of previous knowledge.
2. Introduce a data structure, discussing both its mechanisms for how it stores data and how it implements operations on this data.
3. Justify why the operations are correct.
4. Analyse the running time performance of these operations.

Given that we all have experience with primitive data structures such as arrays (or Python lists), one might wonder why we need to study data structures at all: can't we just do everything with arrays and pointers, already available in many languages?

Indeed, it is true that any data we want to store and any operation we want to perform can be done in a programming language using primitive constructs such as these. The reason we study data structures at all, spending time inventing and refining more complex ones, is largely because of the performance improvements we can hope to gain over their more primitive counterparts. Given the importance of this performance analysis, it is worth reviewing what you already know about how to analyse the running time of algorithms, and pointing out some common misconceptions and subtleties you may have missed along the way.

This is not to say arrays and pointers play no role. To the contrary, the study of data structures can be viewed as the study of how to organize and synthesize basic programming language components in new and sophisticated ways.

## *How do we measure running time?*

As we all know, the amount of time a program or single operation takes to run depends on a host of external factors — computing hardware, other running processes — over which the programmer has no control.

So, in our analysis, we focus on just one central measure of performance: the relationship between an algorithm's *input size* and the *number of basic operations* the algorithm performs. But because even what is meant by "basic operation" can

differ from machine to machine or programming language to programming language, we do not try to precisely quantify the exact number of such operations, but instead categorize *how the number grows* relative to the size of the algorithm's input.

This is our motivation for Big-Oh notation, which is used to bring to the foreground the type of long-term growth of a function, hiding all the numeric constants and smaller terms that do not affect this growth. For example, the functions $n + 1$, $3n - 10$, and $0.001n + \log n$ all have the same growth behaviour as $n$ gets large: they all grow roughly linearly with $n$. Even though these "lines" all have different slopes, we ignore these constants and simply say that these functions are $\mathcal{O}(n)$, "Big-Oh of $n$."

We will not give the formal definition of Big-Oh here. For that, please consult the CSC165 course notes.

We call this type of analysis *asymptotic analysis*, since it deals with the long-term behaviour of a function. It is important to remember two important facts about asymptotic analysis:

- The target of the analysis is always a relationship between the *size of the input* and *number of basic operations* performed.

  What we mean by "size of the input" depends on the context, and we'll always be very careful when defining input size throughout this course. What we mean by "basic operation" is any operation whose running time does not depend on the size of the algorithm's input.

This is deliberately a very liberal definition of "basic operation." We don't want you to get hung up on step counting, because that's completely hidden by Big-Oh expressions.

- The result of the analysis is a *qualitative rate of growth*, not an exact number or even an exact function. We will not say "by our asymptotic analysis, we find that this algorithm runs in 5 steps" or even "...in $10n + 3$ steps." Rather, expect to read (and write) statements like "we find that this algorithm runs in $\mathcal{O}(n)$ time."

## *Three different symbols*

In practice, programmers (and even theoreticians) tend to use the Big-Oh symbol $\mathcal{O}$ liberally, even when the precise definition of $\mathcal{O}$ is not exactly what we intended. However, in this course it will be important to be precise, and we will actually use three symbols to convey different pieces of information, so you will be expected to know which one means what. Here is a recap:

- **Big-Oh**. $f = \mathcal{O}(g)$ means that the function $f(x)$ grows *slower or at the same rate* as $g(x)$. So we can write $x^2 + x = \mathcal{O}(x^2)$, but it is also correct to write $x^2 + x = \mathcal{O}(x^{100})$ or even $x^2 + x = \mathcal{O}(2^x)$.

Or, "$g$ is an upper bound on the rate of growth of $f$."

- **Omega**. $f = \Omega(g)$ means that the function $f(x)$ grows *faster or at the same rate* as $g(x)$. So we can write $x^2 + x = \Omega(x^2)$, but it is also correct to write $x^2 + x = \Omega(x)$ or even $x^2 + x = \Omega(\log \log x)$.

Or, "$g$ is a lower bound on the rate of growth of $f$."

- **Theta**. $f = \Theta(g)$ means that the function $f(x)$ grows *at the same rate* as $g(x)$. So we can write $x^2 + x = \Theta(x^2)$, but not $x^2 + x = \Theta(2^x)$ or $x^2 + x = \Theta(x)$.

  Note: saying $f = \Theta(g)$ is equivalent to saying that $f = \mathcal{O}(g)$ and $f = \Omega(g)$, i.e., Theta is really an AND of Big-Oh and Omega.

Or, "$g$ has the same rate of growth as $f$."

Through unfortunate systemic abuse of notation, most of the time when a computer scientist says an algorithm runs in "Big-Oh of $f$ time," she really means "Theta of $f$ time." In other words, that the function $f$ is not just an upper bound on the rate of growth of the running time of the algorithm, but is in fact *the* rate of growth. The reason we get away with doing so is that in practice, the "obvious" upper bound is in fact *the* rate of growth, and so it is (accidentally) correct to mean Theta even if one has only thought about Big-Oh.

However, the devil is in the details: it is not the case in this course that the "obvious" upper bound will always be the actual rate of growth, and so we will say Big-Oh when we mean an upper bound, and treat Theta with the reverence it deserves. Let us think a bit more carefully about why we need this distinction.

## *Worst-case analysis*

The procedure of asymptotic analysis seems simple enough: look at a piece of code; count the number of basic operations performed in terms of the input size, taking into account loops, helper functions, and recursive calls; then convert that expression into the appropriate Theta form.

remember, a basic operation is any operation whose runtime doesn't depend on the input size

Given any exact mathematical function, it is always possible to determine its qualitative rate of growth, i.e., its corresponding Theta expression. For example, the function $f(x) = 300x^5 - 4x^3 + x + 10$ is $\Theta(x^5)$, and that is not hard to figure out.

So then *why do we need Big-Oh and Omega at all?* Why can't we always just go directly from the $f(x)$ expression to the Theta?

It's because we cannot always take a piece of code an come up with an exact expression for the number of basic operations performed. Even if we take the input size as a variable (e.g., $n$) and use it in our counting, we cannot always determine which basic operations will be performed. This is because input size alone is not the only determinant of an algorithm's running time: often the *value* of the input matters as well.

Consider, for example, a function that takes a list and returns whether this list contains the number 263. If this function loops through the list starting at the front, and stops immediately if it finds an occurrence of 263, then in fact the running time of this function depends not just on how long the list is, but whether and where it has any 263 entries.

Asymptotic notations alone cannot help solve this problem: they help us clarify *how* we are counting, but we have here a problem of *what* exactly we are counting.

This problem is why asymptotic analysis is typically specialized to **worst-case analysis**. Whereas asymptotic analysis studies the relationship between input size and running time, worst-case analysis studies only the relationship between input size of *maximum possible* running time. In other words, rather than answering the question "what is the running time of this algorithm for an input

size $n$?" we instead aim to answer the question "what is the *maximum possible* running time of this algorithm for an input size $n$?" The first question's answer might be a whole range of values; the second question's answer can only be a single number, and that's how we get a function involving $n$.

Some notation: we typically use the name $T(n)$ to represent the maximum possible running time as a function of $n$, the input size. The result of our analysis could be something like $T(n) = \Theta(n)$, meaning that the *worst-case* running time of our algorithm grows linearly with the input size.

### *Bounding the worst case*

But we still haven't answered the question: where do $\mathcal{O}$ and $\Omega$ come in? The answer is basically the same as before: even with our restricted focus on worst-case running time, it is not always possible to calculate an exact expression for this function. What is usually easy to do, however, is calculate an *upper bound* on the maximum number of operations. One such example is the likely familiar line of reasoning, "the loop will run at most $n$ times" when searching for 263 in a list of length $n$. Such analysis, which gives a pessimistic outlook on the most number of operations that could theoretically happen, results in an exact count — e.g., $n + 1$ — which is an *upper bound* on the maximum number of operations. From this analysis, we can conclude that $T(n)$, the worst-case running time, is $\mathcal{O}(n)$.

What we can't conclude is that $T(n) = \Omega(n)$. There is a subtle implication here of the English phrase "at most." When we say "you can have at most 10 chocolates," it is generally understood that you can indeed have exactly 10 chocolates; whatever number is associated with "at most" is achievable.

In our analysis, however, we have no way of knowing that the upper bound we obtain by being pessimistic in our operation counting is actually achievable. This is made more obvious if we explicitly mention the fact that we're studying the maximum possible number of operations: "the maximum running time is less than or equal to $n + 1$" surely says something different than "the maximum running time is equal to $n + 1$."

So how do we show that whatever upper bound we get on the maximum is actually achievable? In practice, we rarely try to show that the *exact* upper bound is achievable, since that doesn't actually matter. Instead, we try to show that an asymptotic lower bound — an Omega — is achievable. For example, we might want to show that the maximum running time is $\Omega(n)$, i.e., grows at least as quickly as $n$.

To show that the maximum running time grows *at least as quickly* as some function $f$, we need to find a family of inputs, one for each input size $n$, whose running time has a lower bound of $f(n)$. For example, for our problem of searching for 263 in a list, we could say that the family of inputs is "lists that contain only 0's." Running the search algorithm on such a list of length $n$ certainly requires checking each element, and so the algorithm takes *at least* $n$ basic operations. From this we can conclude that the *maximum* possible running time is $\Omega(n)$.

Remember, the constants don't matter. The family of inputs that contain only 1's for the first half, and only 263's for the second half, would also have given us the desired lower bound.

To summarize: to perform a complete worst-case analysis and get a tight, Theta bound on the worst-case running time, we need to do the following *two* things:

(i) Give a pessimistic *upper bound* on the number of basic operations that could occur for any input of a fixed size $n$. Obtain the corresponding *Big-Oh* expression (i.e., $T(n) = \mathcal{O}(f)$).

(ii) Give a *family of inputs* (one for each input size), and give a *lower bound* on the number of basic operations that occurs for this particular family of inputs. Obtain the corresponding *Omega* expression (i.e., $T(n) = \Omega(f)$).

Observe that (i) is proving something about *all* possible inputs, while (ii) is proving something about just *one family* of inputs.

If you have performed a careful analysis in (i) and chosen a good family in (ii), then you'll find that the Big-Oh and Omega expressions involve the same function $f$, and which point you can conclude that the worst-case running time is $T(n) = \Theta(f)$.

## *Average-case analysis*

So far in your career as computer scientists, you have been primarily concerned with worst-case algorithm analysis. However, in practice this type of analysis often ends up being misleading, with a variety of algorithms and data structures having a poor worst-case performance still performing well on the majority of possible inputs.

Some reflection makes this not too surprising: focusing on the maximum of a set of numbers (like running times) says very little about the "typical" number in that set, or, more precisely, the distribution of numbers within that set. In this section, we will learn a powerful new technique that enables us to analyse some notion of "typical" running time for an algorithm.

### *Warmup*

Consider the following algorithm, which operates on a non-empty array of integers:

```
1 def evens_are_bad(lst):
2     if every number in lst is even:
3         repeat lst.length times:
4             calculate and print the sum of lst
5         return 1
6     else:
7         return 0
```

Let $n$ represent the length of the input list `lst`. Suppose that `lst` contains only even numbers. Then the initial check on line 2 takes $\Omega(n)$ time, while the computation in the `if` branch takes $\Omega(n^2)$ time. This means that the worst-case

We leave it as an exercise to justify why the `if` branch takes $\Omega(n^2)$ time.

running time of this algorithm is $\Omega(n^2)$. It is not too hard to prove the matching upper bound, and so the worst-case running time is $\Theta(n^2)$.

However, the loop only executes when every number in `lst` is even; when just one number is odd, the running time is $\mathcal{O}(n)$, the maximum possible running time of executing the all-even check. Intuitively, it seems much more likely that **not** every number in `lst` is even, so we expect the more "typical case" for this algorithm is to have a running time bounded above by $\mathcal{O}(n)$, and only very rarely to have a running time of $\Theta(n^2)$.

Because executing the check might abort quickly if it finds an odd number early in the list, we used the pessimistic upper bound of $\mathcal{O}(n)$ rather than $\Theta(n)$.

Our goal now is to define precisely what we mean by the "typical case" for an algorithm's running time when considering a set of inputs. As is often the case when dealing with the distribution of a quantity (like running time) over a set of possible values, we will use our tools from probability theory to help achieve this goal.

We define the **average-case running time** of an algorithm to be the function $T_{avg}(n)$ which takes a number $n$ and returns the (weighted) average of the algorithm's running time for all inputs of size $n$.

For now, let's ignore the "weighted" part, and just think of $T_{avg}(n)$ as computing the average of a set of numbers. What can we say about the average-case for the function `evens_are_bad`? First, we fix some input size $n$. We want to compute the average of all running times over all input lists of length $n$.

At this point you might be thinking, "well, each number is even with probability one-half, so..." This is a nice thought, but a little premature – the first step when doing an average-case analysis is to **define the possible set of inputs**. For this example, we'll start with a particularly simple set of inputs: the lists whose elements are between 1 and 5, inclusive. The reason for choosing such a restricted set is to simplify the calculations we need to perform when computing an average.

As the calculation requires precise numbers, we will need to be precise about what "basic operations" we're counting. For this example, we'll count only the **number of times a list element is accessed**, either to check whether it is even, or when computing the sum of the list. So a "step" will be synonymous with "list access."

It is actually fairly realistic to focus solely on operations of a particular type in a runtime analysis. We typically choose the operation that happens the most frequently (as in this case), or the one which is the most expensive. Of course, the latter requires that we are intimately knowledgeable about the low-level details of our computing environment.

The preceding paragraphs are the work of setting up the context of our analysis: what inputs we're considering, and how we're measuring runtime. The final step is what we had initially talked about: **compute the average running time over inputs of length** $n$. This often requires some calculation, so let's get to it. To simplify our calculations even further, we'll assume that the all-evens check on line 2 always accesses all $n$ elements. In the loop, there are $n^2$ steps (each number is accessed $n$ times, once per time the sum is computed).

You'll explore the "return early" variant in an exercise.

There are really only two possibilities: the lists that have all even numbers will run in $n^2 + n$ steps, while all the other lists will run in just $n$ steps. How many of each type of list are there? For each position, there are two possible even numbers (2 and 4), so the number of lists of length $n$ with every element being even is $2^n$. That sounds like a lot, but consider that there are five possible values

In the language of counting: make $n$ independent decisions, with each decision having two choices.

per element, and hence $5^n$ possible inputs in all. So $2^n$ inputs have all even numbers and take $n^2 + n$ steps, while the remaining $5^n - 2^n$ inputs take $n$ steps.

| | Number | Steps |
|---|---|---|
| all even | $2^n$ | $n^2 + n$ |
| the rest | $5^n - 2^n$ | $n$ |

The average running time is:

$$\begin{aligned} T_{avg}(n) &= \frac{2^n(n^2+n) + (5^n - 2^n)n}{5^n} && (5^n \text{ inputs total}) \\ &= \frac{2^n n^2}{5^n} + n \\ &= \left(\frac{2}{5}\right)^n n^2 + n \\ &= \Theta(n) \end{aligned}$$

Remember that any exponential grows faster than any polynomial, so the first term goes to 0 as $n$ goes to infinity.

This analysis tells us that the average-case running time of this algorithm is $\Theta(n)$, as our intuition originally told us. Because we computed an exact expression for the average number of steps, we could convert this directly into a Theta expression.

As is the case with worst-case analysis, it won't always be so easy to compute an exact expression for the average, and in those cases the usual upper and lower bounding must be done.

*The probabilistic view*

The analysis we performed in the previous section was done through the lens of *counting* the different kinds of inputs and then computing the average of their running times. However, there is a more powerful technique: treating the algorithm's running time as a *random variable* $T$, defined over the set of possible inputs of size $n$. We can then redo the above analysis in this probabilistic context, performing these steps:

1. Define the set of possible inputs and a *probability distribution* over this set. In this case, our set is **all lists of length $n$ that contain only elements in the range 1-5**, and the probability distribution is the **uniform distribution**.

   Recall that the uniform distribution assigns equal probability to each element in the set.

2. Define how we are measuring runtime. (This is unchanged from the previous analysis.)
3. Define the random variable $T$ over this probability space to represent the running time of the algorithm.

   In this case, we have the nice definition

   $$T = \begin{cases} n^2 + n, & \text{input contains only even numbers} \\ n, & \text{otherwise} \end{cases}$$

   Recall that a random variable is a function whose domain is the set of inputs.

4. Compute the **expected value of** $T$, using the chosen probability distribution and formula for $T$. Recall that the expected value of a variable is determined by taking the sum of the possible values of $T$, each weighted by the probability of obtaining that value.

   $$\mathbb{E}[T] = \sum_t t \cdot \Pr[T = t]$$

In this case, there are only two possible values for $T$:

$$\begin{aligned}
\mathbb{E}[T] &= (n^2 + n) \cdot \Pr[\text{the list contains only even numbers}] \\
&\quad + n \cdot \Pr[\text{the list contains at least one odd number}] \\
&= (n^2 + n) \cdot \left(\frac{2}{5}\right)^n + n \cdot \left(1 - \left(\frac{2}{5}\right)^n\right) \\
&= n^2 \cdot \left(\frac{2}{5}\right)^n + n \\
&= \Theta(n)
\end{aligned}$$

Of course, the calculation ended up being the same, even if we approached it a little differently. The point to shifting to using probabilities is in unlocking the ability to change the *probability distribution*. Remember that our first step, defining the set of inputs and probability distribution, is a great responsibility of the ones performing the runtime analysis. How in the world can we choose a "reasonable" distribution? There is a tendency for us to choose distributions that are easy to analyse, but of course this is not necessarily a good criterion for evaluating an algorithm.

What does it mean for a distribution to be "reasonable," anyway?

By allowing ourselves to choose not only what inputs are allowed, but also give relative probabilities of those inputs, we can use the exact same technique to analyse an algorithm under several different possible scenarios. This is both the reason we use "weighted average" rather than simply "average" in our above definition of average-case, and also how we are able to calculate it.

---

### Exercise Break!

1.1 Prove that the `evens_are_bad` algorithm on page 11 has a worst-case running time of $\mathcal{O}(n^2)$, where $n$ is the length of the input list.

1.2 Consider this alternate input space for `evens_are_bad`: each element in the list is even with probability $\frac{99}{100}$, independent of all other elements. Prove that under this distribution, the average-case running time is still $\Theta(n)$.

1.3 Suppose we allow the "all-evens" check on line 2 of `evens_are_bad` to stop immediately after it finds an odd element. Perform a new average-case analysis for this mildly-optimized version of the algorithm using the same input distribution as in the notes.

*Hint*: part of the running time $T$ now can take on any value between 1 and $n$; first compute the probability of getting each of these values, and then use the expected value formula.

---

## *Quicksort is fast on average*

The previous example may have been a little underwhelming, since it was "obvious" that the worst-case was quite rare, and so not surprising that the average-case running time is asymptotically faster than the worst-case. However, this is

not a fact you should take for granted. Indeed, it is often the case that algorithms are asymptotically no better "on average" than they are in the worst-case. Sometimes, though, an algorithm can have a significantly better average case than worst case, but it is not nearly as obvious; we'll finish off this chapter by studying one particularly well-known example.

Keep in mind that because asymptotic notation hides the constants, saying two functions are different asymptotically is much more significant than saying that one function is "bigger" than another.

Recall the quicksort algorithm, which takes a list and sorts it by choosing one element to be the *pivot* (say, the first element); partitioning the remaining elements into two parts, those less than the pivot, and those greater than the pivot; recursively sorting each part; and then combining the results.

```
def quicksort(array):
    if array.length < 2:
        return
    else:
        pivot = array[0]
        smaller, bigger = partition(array[1:], pivot)
        quicksort(smaller)
        quicksort(bigger)
        array = smaller + [pivot] + bigger  # array concatenation

def partition(array, pivot):
    smaller = []
    bigger = []
    for item in array:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

This version of `quicksort` uses linear-size auxiliary storage; we chose this because it is a little simpler to write and analyse. The more standard "in-place" version of quicksort has the same running time behaviour, it just uses less space.

You have seen in previous courses that the choice of pivot is crucial, as it determines the size of each of the two partitions. In the best case, the pivot is the median, and the remaining elements are split into partitions of roughly equal size, leading to a running time of $\Theta(n \log n)$, where $n$ is the size of the list. However, if the pivot is always chosen to be the maximum element, then the algorithm must recurse on a partition that is only one element smaller than the original, leading to a running time of $\Theta(n^2)$.

So, given that there is a difference between the best- and worst-case running times of quicksort, the next natural question to ask is: What is the average-case running time? This is what we'll answer in this section.

First, let $n$ be the length of the list. Our set of inputs is all possible permutations of the numbers 1 to $n$ (inclusive). We'll assume any of these $n!$ permutations are equally likely; in other words, we'll use the uniform distribution over all possible permutations of $\{1, \ldots, n\}$. We will measure runtime by counting the *number of comparisons* between elements in the list in the `partition` function.

Our analysis will therefore assume the lists have no duplicates.

Let $T$ be the random variable counting the number of comparisons made. Before starting the computation of $\mathbb{E}[T]$, we define additional random variables: for each $i, j \in \{1, \ldots, n\}$ with $i < j$, let $X_{ij}$ be the indicator random variable defined as:

$$X_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are compared} \\ 0, & \text{otherwise} \end{cases}$$

Remember, this is all in the context of choosing a random permutation.

Because each pair is compared at most once, we obtain the total number of comparisons simply by adding these up:

$$T = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}.$$

The purpose of decomposing $T$ into a sum of simpler random variables is that we can now apply the *linearity of expectation* (see margin note).

If $X$, $Y$, and $Z$ are random variables and $X = Y + Z$, then $\mathbb{E}[X] = \mathbb{E}[Y] + \mathbb{E}[Z]$, *even if* $Y$ and $Z$ are dependent.

$$\begin{aligned} \mathbb{E}[T] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[i \text{ and } j \text{ are compared}] \end{aligned}$$

Recall that for an indicator random variable, its expected value is simply the probability that it takes on the value 1.

To make use of this "simpler" form, we need to investigate the probability that $i$ and $j$ are compared when we run quicksort on a random array.

**Proposition 1.1.** Let $1 \leq i < j \leq n$. The probability that $i$ and $j$ are compared when running quicksort on a random permutation of $\{1, \ldots, n\}$ is $2/(j - i + 1)$.

*Proof.* First, let us think about precisely when elements are compared with each other in quicksort. Quicksort works by selecting a pivot element, then comparing the pivot to every other item to create two partitions, and then recursing on each partition separately. So we can make the following observations:

(i) Every comparison must involve the "current" pivot.

(ii) The pivot is only compared to the items that have always been placed into the same partition as it by all previous pivots.

Or, once two items have been put into different partitions, they will never be compared.

So in order for $i$ and $j$ to be compared, one of them must be chosen as the pivot while the other is in the same partition. What could cause $i$ and $j$ to be in different partitions? This only happens if one of the numbers between $i$ and $j$ (exclusive) is chosen as a pivot before $i$ or $j$ is chosen. So then $i$ and $j$ are compared by quicksort if and only if one of them is selected to be pivot first out of the set of numbers $\{i, i + 1, \ldots, j\}$.

Because we've given an implementation of quicksort that always chooses the first item to be the pivot, the item that is chosen first as pivot must be the one that appears first in the random input permutation. Since we choose the permutation

Of course, the input list contains other numbers, but because our `partition` algorithm preserves relative order within a partition, if $a$ appears before $b$ in the original list, then $a$ will still

uniformly at random, each item in the set $\{i, \ldots, j\}$ is equally likely to appear first, and thus be chosen first as pivot.

Finally, because there are $j - i + 1$ numbers in this set, the probability that either $i$ or $j$ is chosen first is $2/(j - i + 1)$, and this is the probability that $i$ and $j$ are compared. ■

Now that we have this probability computed, we can return to our expected value calculation to complete our average-case analysis.

**Theorem 1.2** (Quicksort average-case runtime)**.** The average number of comparisons made by quicksort on a uniformly chosen random permutation of $\{1, \ldots, n\}$ is $\Theta(n \log n)$.

*Proof.* As before, let $T$ be the random variable representing the running time of quicksort. Our previous calculations together show that

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[i \text{ and } j \text{ are compared}] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\
&= \sum_{i=1}^{n} \sum_{j'=1}^{n-i} \frac{2}{j' + 1} && \text{(change of index } j' = j - i\text{)} \\
&= 2 \sum_{i=1}^{n} \sum_{j'=1}^{n-i} \frac{1}{j' + 1}
\end{aligned}$$

| $i$ | $j'$ values |
|---|---|
| 1 | $1, 2, 3, \ldots, n-2, n-1$ |
| 2 | $1, 2, 3, \ldots, n-2$ |
| $\vdots$ | $\vdots$ |
| $n-3$ | $1, 2, 3$ |
| $n-2$ | $1, 2$ |
| $n-1$ | $1$ |
| $n$ | |

Now, note that the individual terms of the inner summation don't depend on $i$; only the bound does. The first term, when $j' = 1$, occurs when $1 \leq i \leq n - 1$, or $n - 1$ times in total; the second ($j' = 2$) occurs when $i \leq n - 2$, and in general the $j' = k$ term appears $n - k$ times.

So we can simplify the counting to eliminate the summation over $i$:

$$\begin{aligned}
\mathbb{E}[T] &= 2 \sum_{j'=1}^{n-1} \frac{n - j'}{j' + 1} \\
&= 2 \sum_{j'=1}^{n-1} \left( \frac{n+1}{j'+1} - 1 \right) \\
&= 2(n+1) \sum_{j'=1}^{n-1} \frac{1}{j'+1} - 2(n-1)
\end{aligned}$$

We will use the fact from mathematics that the function $\sum_{i=1}^{n-1} \frac{1}{i+1}$ is $\Theta(\log n)$, and so we get that $\mathbb{E}[T] = \Theta(n \log n)$. ■

Actually, we even know the exact constant hidden in the $\Theta$: Look up the *Harmonic series* if you're interested in learning more!

## Exercise Break!

1.4 Review the insertion sort algorithm, which builds up a sorted list by repeatedly inserting new elements into a sorted sublist (usually at the front). We know that its worst-case running time is $\Theta(n^2)$, but its best case is $\Theta(n)$, even better than quicksort. So, does it beat quicksort on average?

Suppose we run insertion sort on a random permutation of the numbers $\{1, \ldots, n\}$, and consider counting the *number of swaps* as the running time. Let $T$ be the random variable representing the total number of swaps.

(a) For each $1 \leq i \leq n$, define the random variable $S_i$ to be the number of swaps made when $i$ is inserted into the sorted sublist.

Express $T$ in terms of $S_i$.

(b) For each $1 \leq i, j \leq n$, define the random *indicator* variables $X_{ij}$ that is 1 if $i$ and $j$ are swapped during insertion sort.

Express $S_i$ in terms of the $X_{ij}$.

(c) Prove that $\mathbb{E}[X_{ij}] = 1/2$.

(d) Show that the average-case running time of insertion sort is $\Theta(n^2)$.

---

# 2 *Priority Queues and Heaps*

In this chapter, we will study our first major data structure: the heap. As this is our first extended analysis of a new data structure, it is important to pay attention to the four components of this study outlined at the previous chapter:

1. Motivate a new abstract data type or data structure with some examples and reflection of previous knowledge.
2. Introduce a data structure, discussing both its mechanisms for how it stores data and how it implements operations on this data.
3. Justify why the operations are correct.
4. Analyse the running time performance of these operations.

## *Abstract data type vs. data structure*

The study of data structures involves two principal, connected pieces: a specification of what data we want to store and operations we want to support, and an implementation of this data type. We tend to blur the line between these two components, but the difference between them is fundamental, and we often speak of one independently of the other. So before we jump into our first major data structure, let us remind ourselves of the difference between these two.

**Definition 2.1** (abstract data type, data structure)**.** An **abstract data type (ADT)** is a theoretical model of an entity and the set of operations that can be performed on that entity.

The key term is *abstract*: an ADT is a definition that can be understood and communicated without any code at all.

A **data structure** is a value in a program which can be used to store and operate on data.

For example, contrast the difference between the *List ADT* and an *array* data structure.

### List ADT

- Length($L$): Return the number of items in $L$.
- Get($L, i$): Return the item stored at index $i$ in $L$.
- Store($L, i, x$): Store the item $x$ at index $i$ in $L$.

There may be other operations you think are fundamental to lists; we've given as bare-bones a definition as possible.

This definition of the List ADT is clearly abstract: it specifies what the possible operations are for this data type, but says nothing at all about *how* the data is stored, or *how* the operations are performed.

It may be tempting to think of ADTs as the definition of interfaces or abstract classes in a programming language – something that specifies a collection of methods that *must* be implemented – but keep in mind that it is *not* necessary to represent an ADT in code. A written description of the ADT, such as the one we gave above, is perfectly acceptable.

On the other hand, a data structure is tied fundamentally to code. It exists as an entity in a program; when we talk about data structures, we talk about how we write the code to implement them. We are aware of not just what these data structures do, but *how* they do them.

When we discuss arrays, for example, we can say that they implement the List ADT; i.e., they support the operations defined in the List ADT. However, we can say much more than this:

- Arrays store elements in consecutive locations in memory
- They perform GET and STORE in constant time with respect to the length of the array.
- How LENGTH is supported is itself an implementation detail specific to a particular language. In C, arrays must be wrapped in a struct to manually store their length; in Java, arrays have a special immutable attribute called `length`; in Python, native lists are implemented using arrays an a member to store length.

At least, the standard CPython implementation of Python.

The main implementation-level detail that we'll care about this in course is the running time of an operation. This is *not* a quantity that can be specified in the definition of an ADT, but is certainly something we can study if we know how an operation is implemented in a particular data structure.

## *The Priority Queue ADT*

The first abstract data type we will study is the **Priority Queue**, which is similar in spirit to the stacks and queues that you have previously studied. Like those data types, the priority queue supports adding and removing an item from a collection. Unlike those data types, the order in which items are removed does not depend on the order in which they are added, but rather depends on a *priority* which is specified when each item is added.

A classic example of priority queues in practice is a hospital waiting room: more severe injuries and illnesses are generally treated before minor ones, regardless of when the patients arrived at the hospital.

### Priority Queue ADT

- INSERT($PQ, x, priority$): Add $x$ to the priority queue $PQ$ with the given priority.
- FINDMAX($PQ$): Return the item in $PQ$ with the highest priority.
- EXTRACTMAX($PQ$): Remove and return the item from $PQ$ with the highest priority.

As we have already discussed, one of the biggest themes of this course is the distinction between the definition of an abstract data type, and the implementation of that data type using a particular data structure. To emphasize that these are separate, we will first give a naïve implementation of the Priority Queue ADT that is perfectly correct, but inefficient. Then, we will contrast this approach with one that uses the heap data structure.

One can view most hospital waiting rooms as a physical, buggy implementation of a priority queue.

### *A basic implementation*

Let us consider using an unsorted linked list to implement a priority queue. In this case, adding a new item to the priority queue can be done in constant time: simply add the item and corresponding priority to the front of the linked list. However, in order to find or remove the item with the lowest priority, we must search through the entire list, which is a linear-time operation.

Given a new ADT, it is often helpful to come up with a "naïve" or "brute force" implementation using familiar primitive data structures like arrays and linked lists. Such implementations are usually quick to come up with, and analysing the running time of the operations is also usually straight-forward. Doing this analysis gives us targets to beat: given that we can code up an implementation of priority queue which supports INSERT in $\Theta(1)$ time and FINDMAX and EXTRACTMAX in $\Theta(n)$ time, can we do better using a more complex data structure? The rest of this chapter is devoted to answering this question.

```
26 def Insert(PQ, x, priority):
27     n = Node(x, priority)
28     oldHead = PQ.head
29     n.next = old_head
30     PQ.head = n
31
32
33 def FindMax(PQ):
34     n = PQ.head
35     maxNode = None
36     while n is not None:
37         if maxNode is None or n.priority > maxNode.priority:
38             maxNode = n
39         n = n.next
40     return maxNode.item
41
42
43 def ExtractMax(PQ):
```

```
    n = PQ.head
    prev = None
    maxNode = None
    prevMaxNode = None
    while n is not None:
        if maxNode is None or n.priority > maxNode.priority:
            maxNode, prevMaxNode = n, prevNode
        prev, n = n, n.next

    if prevMaxNode is None:
        self.head = maxNode.next
    else:
        prevMaxNode.next = maxNode.next

    return maxNode.item
```

## *Heaps*

Recall that a *binary tree* is a tree in which every node has at most two children, which we distinguish by calling the left and right. You probably remember studying *binary search trees*, a particular application of a binary tree that can support fast insertion, deletion, and search.

We'll look at a more advanced form of binary search trees in the next chapter.

Unfortunately, this particular variant of binary trees does not exactly suit our purposes, since the item with the highest priority will generally be at or near the bottom of a BST. Instead, we will focus on a variation of binary trees that uses the following property:

**Definition 2.2** (heap property)**.** A tree satisfies the **heap property** if and only if for each node in the tree, the value of that node is greater than or equal to the value of all of its descendants.

Alternatively, for any pair of nodes $a, b$ in the tree, if $a$ is an ancestor of $b$, then the value of $a$ is greater than or equal to the value of $b$.

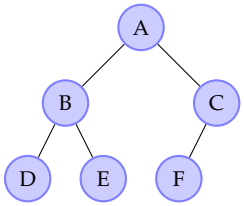
This property is actually less stringent than the BST property: given a node which satisfies the heap property, we cannot conclude anything about the relationships between its left and right subtrees. This means that it is actually much easier to get a compact, balanced binary tree that satisfies the heap property than one that satisfies the BST property. In fact, we can get away with enforcing as strong a balancing as possible for our data structure.

**Definition 2.3** (complete binary tree)**.** A binary tree is **complete** if and only if it satisfies the following two properties:

- All of its levels are full, except possibly the bottom one.

Every leaf must be in one of the two bottommost levels.

- All of the nodes in the bottom level are as far to the left as possible.




Complete trees are essentially the trees which are most "compact." A complete tree with $n$ nodes has $\lceil \log n \rceil$ height, which is the smallest possible height for a tree with this number of nodes.

Moreover, because we also specify that the nodes in the bottom layer must be as far left as possible, there is never any ambiguity about where the "empty" spots in a complete tree are. *There is only one complete tree shape for each number of nodes.*

Because of this, we do not need to use up space to store references between nodes, as we do in a standard binary tree implementation. Instead, we fix a conventional ordering of the nodes in the heap, and then simply write down the items in the heap according to that order. The order we use is called *level order*, because it orders the elements based on their depth in the tree: the first node is the root of the tree, then its children (at depth 1) in left-to-right order, then all nodes at depth 2 in left-to-right order, etc.

The array representation of the above complete binary tree.

| A | B | C | D | E | F |
|---|---|---|---|---|---|

With these two definitions in mind, we can now define a heap.

**Definition 2.4** (heap)**.** A **heap** is a binary tree that *satisfies the heap property* and is *complete*.

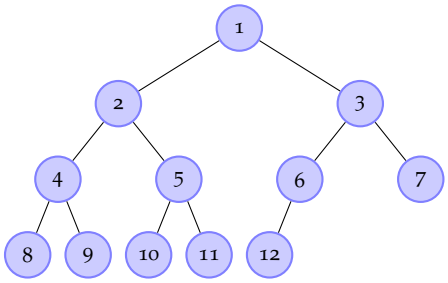
We implement a heap in a program as an array, where the items in the array correspond to the level order of the actual binary tree.

This is a nice example of how we use primitive data structures to build more complex ones. Indeed, a heap is nothing sophisticated from a technical standpoint; it is merely an array whose values are ordered in a particular way.

### *A note about array indices*

In addition to being a more compact representation of a complete binary tree, this array representation has a beautiful relationship between the indices of a node in the tree and those of its children.

We assume the items are stored starting at index 1 rather than 0, which leads to the following indices for the nodes in the tree.




We are showing the indices where each node would be stored in the array.

A pattern quickly emerges. For a node corresponding to index $i$, its left child is stored at index $2i$, and its right child is stored at index $2i + 1$. Going backwards, we can also deduce that the parent of index $i$ (when $i > 1$) is stored at index $\lfloor i/2 \rfloor$. This relationship between indices will prove very useful in the following section when we start performing more complex operations on heaps.

### *Heap implementation of a priority queue*

Now that we have defined the heap data structure, let us see how to use it to implement the three operations of a priority queue.

FindMax becomes very simple to both implement and analyse, because the root of the heap is always the item with the maximum priority, and in turn is always stored at the front of the array.

Remember that indexing starts at 1 rather than 0.

```
def FindMax(PQ):
    return PQ[1]
```

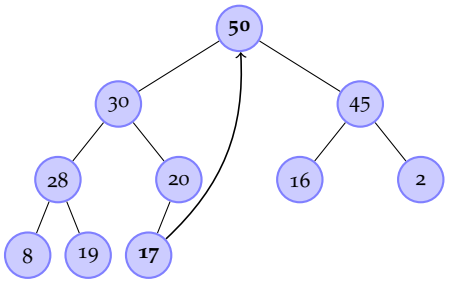
Remove is a little more challenging. Obviously we need to remove the root of the tree, but how do we decide what to replace it with? One key observation is that because the resulting heap must still be complete, we know how its structure must change: the very last (rightmost) leaf must no longer appear.

So our first step is to save the root of the tree, and then replace it with the last leaf. (Note that we can access the leaf in constant time because we know the size of the heap, and the last leaf is always at the end of the list of items.)

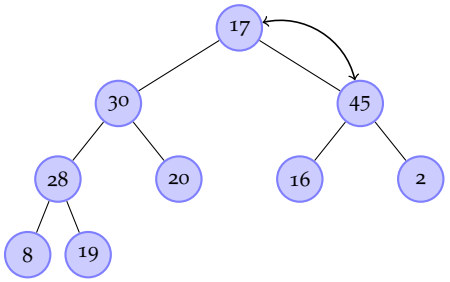
But the last leaf priority is smaller than many other priorities, and so if we leave the heap like this, the heap property will be violated. Our last step is to repeatedly swap the moved value with one of its children until the heap property is satisfied once more. On each swap, we choose the *larger* of the two children to ensure that the heap property is preserved.

```
29 def ExtractMax(PQ):
30     temp = PQ[1]
31     PQ[1] = PQ[PQ.size]  # Replace the root with the last leaf
32     PQ.size = PQ.size - 1
33
34     # Bubble down
35     i = 1
36     while i < PQ.size:
37         curr_p = PQ[i].priority
38         left_p = PQ[2*i].priority
39         right_p = PQ[2*i + 1].priority
40
41         # heap property is satisfied
42         if curr_p >= left_p and curr_p >= right_p:
43             break
44         # left child has higher priority
45         else if left_p >= right_p:
46             PQ[i], PQ[2*i] = PQ[2*i], PQ[i]
47             i = 2*i
48         # right child has higher priority
49         else:
50             PQ[i], PQ[2*i + 1] = PQ[2*i + 1], PQ[i]
51             i = 2*i + 1
52
53     return temp
```

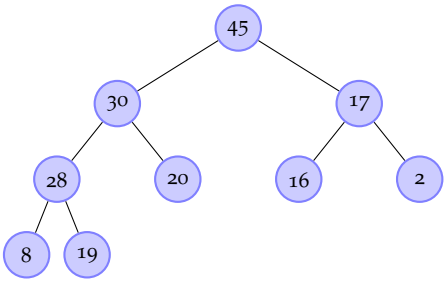
Rightmost leaf 17 is moved to the root.




17 is swapped with 45 (since 45 is greater than 30).




No more swaps occur; 17 is greater than both 16 and 2.




The repeated swapping is colloquially called the "bubble down" step, referring to how the last leaf starts at the top of the heap and makes its way back down in the loop.

What is the running time of this algorithm? All individual lines of code take constant time, meaning the runtime is determined by the number of loop iterations.

At each iteration, either the loop stops immediately, or $i$ increases by at least a factor of 2. This means that the total number of iterations is at most $\log n$, where $n$ is the number of items in the heap. The worst-case running time of REMOVE is therefore $\mathcal{O}(\log n)$.

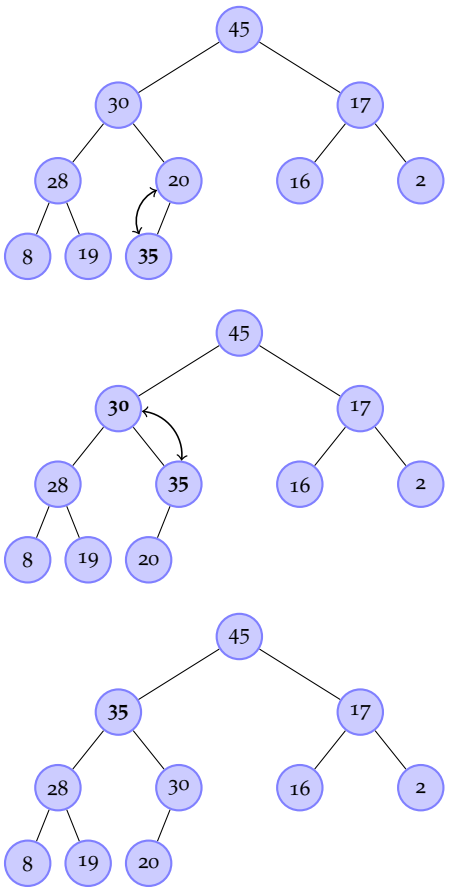
The implementation of INSERT is similar. We again use the fact that the number of items in the heap completely determines its structure: in this case, because a

new item is being added, there will be a new leaf immediately to the right of the current final leaf, and this corresponds to the next open position in the array after the last item.

So our algorithm simply puts the new item there, and then performs an inverse of the swapping from last time, comparing the new item with its parent, and swapping if it has a larger priority. The margin diagrams show the result of adding 35 to the given heap.

```
56 def Insert(PQ, x, priority):
57     PQ.size = PQ.size + 1
58     PQ[PQ.size].item = x
59     PQ[PQ.size].priority = priority
60 
61     i = PQ.size
62     while i > 1:
63         curr_p = PQ[i].priority
64         parent_p = PQ[i // 2].priority
65 
66         if curr_p <= parent_p:  # heap property satisfied, break
67             break
68         else:
69             PQ[i], PQ[i // 2] = PQ[i // 2], PQ[i]
70             i = i // 2
```

A "bubble up" instead of "bubble down"




35 is swapped twice with its parent (20, then 30), but does not get swapped with 45.

Again, this loop runs at most $\log n$ iterations, where $n$ is the number of items in the heap. The worst-case running time of this algorithm is therefore $\mathcal{O}(\log n)$.

### *Runtime summary*

Let us compare the worst-case running times for the three operations for the two different implementations we discussed in this chapter. In this table, $n$ refers to the number of elements in the priority queue.

| Operation | Linked list | Heap |
|---|---|---|
| Insert | $\Theta(1)$ | $\Theta(\log n)$ |
| FindMax | $\Theta(n)$ | $\Theta(1)$ |
| ExtractMax | $\Theta(n)$ | $\Theta(\log n)$ |

This table nicely illustrates the tradeoffs generally found in data structure design and implementation. We can see that heaps beat unsorted linked lists in two of the three priority queue operations, but are asymptotically slower than unsorted linked lists when adding a new element. Now, this particular case is not much of a choice: that the *slowest* operation for heaps runs in $\Theta(\log n)$ time in the worst case is substantially better than the corresponding $\Theta(n)$ time for unsorted linked lists, and in practice heaps are indeed widely used.

The reason for the speed of the heap operations is the two properties – the heap property and completeness – that are enforced by the heap data structures. These properties impose a structure on the data that allows us to more quickly extract the desired information. The cost of these properties is that they must be maintained whenever the data structure is mutated. It is not enough to take

a new item and add it to the end of the heap array; it must be "bubbled up" to its correct position to maintain the heap property, and this is what causes the `Insert` operation to take longer.

### *Heapsort and building heaps*

In our final section of this chapter, we will look at one interesting application of heaps to a fundamental task in computer science: sorting. Given a heap, we can extract a sorted list of the elements in the heap simply by repeatedly calling Remove and adding the items to a list.

Of course, this technically sorts by priority of the items. In general, given a list of values to sort, we would treat these values as priorities for the purpose of priority queue operations.

However, to turn this into a true sorting algorithm, we need a way of converting an input unsorted list into a heap. To do this, we interpret the list as the level order of a complete binary tree, same as with heaps. The difference is that this binary tree does not necessarily satisfy the heap property, and it is our job to fix it.

We can do this by performing the "bubble down" operation on each node in the tree, starting at the bottom node and working our way up.

```
9  def BuildHeap(items):
10     i = items.size
11     while i > 0:
12         BubbleDown(items, i)
13         i = i - 1
14
15 def BubbleDown(heap, i):
16     while i < heap.size:
17         curr_p = heap[i].priority
18         left_p = heap[2*i].priority
19         right_p = heap[2*i + 1].priority
20
21         # heap property is satisfied
22         if curr_p >= left_p and curr_p >= right_p:
23             break
24         # left child has higher priority
25         else if left_p >= right_p:
26             PQ[i], PQ[2*i] = PQ[2*i], PQ[i]
27             i = 2*i
28         # right child has higher priority
29         else:
30             PQ[i], PQ[2*i + 1] = PQ[2*i + 1], PQ[i]
31             i = 2*i + 1
```

What is the running time of this algorithm? Let $n$ be the length of `items`. Then the loop in BubbleDown iterates at most $\log n$ times; since BubbleDown is

called $n$ times, this means that the worst-case running time is $\mathcal{O}(n \log n)$.

However, this is a rather loose analysis: after all, the larger $i$ is, the fewer iterations the loop runs. And in fact, this is a perfect example of a situation where the "obvious" upper bound on the worst-case is actually not tight, as we shall soon see.

To be more precise, we require a better understanding of how long BUBBLEDOWN takes to run as a function of $i$, and not just the length of `items`. Let $T(n, i)$ be the maximum number of loop iterations of BUBBLEDOWN for input $i$ and a list of length $n$. Then the total number of iterations in all $n$ calls to BUBBLEDOWN from BUILDHEAP is $\sum_{i=1}^{n} T(n, i)$. So how do we compute $T(n, i)$? The maximum number of iterations is simply the height $h$ of node $i$ in the complete binary tree with $n$ nodes.

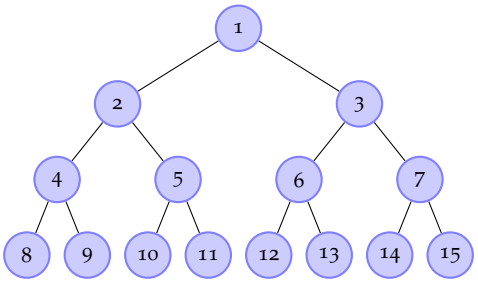
Note that $T$, a runtime function, has *two* inputs, reflecting our desire to incorporate both $i$ and $n$ in our analysis.

Each iteration goes down one level in the tree.

So we can partition the nodes based on their height:

$$\sum_{i=1}^{n} T(i, n) = \sum_{h=1}^{k} h \cdot \#\text{ nodes at height } h$$

The final question is, how many nodes are at height $h$ in the tree? To make the analysis simpler, suppose the tree has height $k$ and $n = 2^k - 1$ nodes; this causes the binary tree to have a full last level. Consider the complete binary tree shown at the right ($k = 4$). There are 8 nodes at height 1 (the leaves), 4 nodes at height 2, 2 nodes at height 3, and 1 node at height 4 (the root). In general, the number of nodes at height $h$ when the tree has height $k$ is $2^{k-h}$. Plugging this into the previous expression for the total number of iterations yields:




$$\begin{aligned}
\sum_{i=1}^{n} T(n, i) &= \sum_{h=1}^{k} h \cdot \#\text{ nodes at height } h \\
&= \sum_{h=1}^{k} h \cdot 2^{k-h} \\
&= 2^k \sum_{h=1}^{k} \frac{h}{2^h} && (2^k \text{ doesn't depend on } h) \\
&= (n+1) \sum_{h=1}^{k} \frac{h}{2^h} && (n = 2^k - 1) \\
&< (n+1) \sum_{h=1}^{\infty} \frac{h}{2^h}
\end{aligned}$$

It turns out quite remarkably that $\sum_{h=1}^{\infty} \frac{h}{2^h} = 2$, and so the total number of iterations is less than $2(n+1)$.

Bringing this back to our original problem, this means that the total cost of all the calls to BUBBLEDOWN is $\mathcal{O}(n)$, which leads to a total running time of BUILDHEAP of $\mathcal{O}(n)$, i.e., linear in the size of the list.

Note that the final running time depends only on the size of the list: there's no "$i$" input to BUILDHEAP, after all. So what we did was a more careful analysis of the helper function

*The Heapsort algorithm*

Now let us put our work together with the heapsort algorithm. Our first step is to take the list and convert it into a heap. Then, we repeatedly extract the maximum element from the heap, with a bit of a twist to keep this sort in-place: rather than return it and build a new list, we simply swap it with the current last leaf, making sure to decrement the heap size so that the max is never touched again for the remainder of the algorithm.

```
34 def Heapsort(items):
35     BuildHeap(items)
36
37     # Repeated build up a sorted list from the back of the list, in place.
38     # sorted\_index is the index immediately before the sorted part.
39     sorted_index = items.size
40     while sorted_index > 1:
41         swap items[sorted_index], items[1]
42         sorted_index = sorted_index - 1
43
44         # BubbleDown uses items.size, and so won't touch the sorted part.
45         items.size = sorted_index
46         BubbleDown(items, 1)
```

Let $n$ represent the number of elements in `items`. The loop maintains the invariant that all the elements in positions `sorted_index` + 1 to $n$, inclusive, are in sorted order, and are bigger than any other items remaining in the "heap" portion of the list, the items in positions 1 to `sorted_index`.

Unfortunately, it turns out that even though BuildHeap takes linear time, the repeated removals of the max element and subsequent BubbleDown operations in the loop in total take $\Omega(n \log n)$ time in the worst-case, and so heapsort also has a worst-case running time of $\Omega(n \log n)$.

This may be somewhat surprising, given that the repeated BubbleDown operations operate on smaller and smaller heaps, so it seems like the analysis should be similar to the analysis of BuildHeap. But of course the devil is in the details – we'll let you explore this in the exercises.

---

## Exercise Break!

2.1 Consider the following variation of BuildHeap, which starts bubbling down from the top rather than the bottom:

```
1 def BuildHeap(items):
2     i = 1
3     while i < items.size:
```

```
4           BubbleDown(items, i)
5           i = i + 1
```

(a) Give a good upper bound on the running time of this algorithm.

(b) Is this algorithm also correct? If so, justify why it is correct. Otherwise, give a counterexample: an input where this algorithm fails to produce a true heap.

2.2 Analyse the running time of the loop in HEAPSORT. In particular, show that its worst-case running time is $\Omega(n \log n)$, where $n$ is the number of items in the heap.

# 3 *Dictionaries, Round One: AVL Trees*

In this chapter and the next, we will look at two data structures that take very different approaches to implementing the same abstract data type: the dictionary. A **dictionary** is a collection of key-value pairs that supports the following operations:

**Dictionary ADT**

- SEARCH($D, key$): return the value corresponding to a given key in the dictionary.
- INSERT($D, key, value$): insert a new key-value pair into the dictionary.
- DELETE($D, key$): remove the key-value pair with the given key from the dictionary.

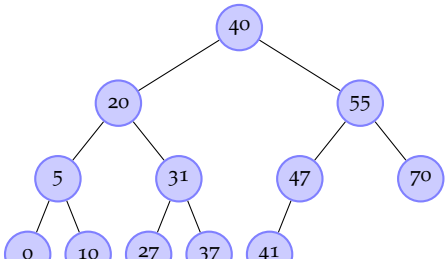
You have probably seen some basic uses of dictionaries in your prior programming experience; Python `dicts` and Java `Maps` are realizations of this ADT in these two languages. We use dictionaries as a simple and efficient tool in our applications for storing *associative* data with unique key identifiers, such as mapping student IDs to a list of courses each student is enrolled in. Dictionaries are also fundamental in the behind-the-scenes implementation of programming languages themselves, from supporting identifier lookup during programming compilation or execution, to implementing dynamic dispatch for method lookup during runtime.

One might wonder why we devote two chapters to data structures implementing dictionaries at all, given that we can implement this functionality using the various list data structures at our disposal. Of course, the answer is efficiency: it is not obvious how to use either a linked list or array to support all three of these operations in better than $\Theta(n)$ worst-case time. In these two chapters, we will examine some new data structures that do better both in the worst case and on average.

or even on average

## *Naïve Binary Search Trees*

Recall the definition of a **binary search tree (BST)**, which is a binary tree that satisfies the *binary search tree property*: for every node, its key is $\geq$ every key in its left subtree, and $\leq$ every key in its right subtree. An example of a binary search tree is shown in the figure on the right, with each key displayed.



We can use binary search trees to implement the Dictionary ADT, assuming the keys can be ordered. Here is one naïve implementation of the three functions SEARCH, INSERT, and DELETE for a binary search tree – you have seen something similar before, so we won't go into too much detail here.

The SEARCH algorithm is the simple recursive approach, using the BST property to decide which side of the BST to recurse on. The INSERT algorithm basically performs a search, stopping only when it reaches an empty spot in the tree, and inserts a new node. The DELETE algorithm searches for the given key, then replaces the node with either its predecessor (the maximum key in the left subtree) or its successor (the minimum key in the right subtree).

```
def Search(D, key):
    # Return the value in <D> corresponding to <key>, or None if key doesn't appear
    if D is empty:
        return None
    else if D.root.key == key:
        return D.root.value
    else if D.root.key > key:
        return Search(D.left, key)
    else:
        return Search(D.right, key)


def Insert(D, key, value):
    if D is empty:
        D.root.key = key
        D.root.value = value
    else if D.root.key >= key:
        Insert(D.left, key, value)
    else:
        Insert(D.right, key, value)


def Delete(D, key):
    if D is empty:
        pass  # do nothing
    else if D.root.key == key:
        D.root = ExtractMax(D.left) or ExtractMin(D.right)
    else if D.root.key > key:
        Delete(D.left, key)
    else:
        Delete(D.right, key)
```

We omit the implementations of ExtractMax and ExtractMin. These functions remove and return the key-value pair with the highest and lowest keys from a BST, respectively.

All three of these algorithms are recursive; in each one the cost of the *non-recursive* part is $\Theta(1)$ (simply some comparisons, attribute access/modification), and so each has running time proportional to the number of recursive calls

made. Since each recursive call is made on a tree of height one less than its parent call, in the worst-case the number of recursive calls is $h$, the height of the BST. This means that an upper bound on the worst-case running time of each of these algorithms is $\mathcal{O}(h)$.

We leave it as an exercise to show that this bound is in fact tight in all three cases.

However, this bound of $\mathcal{O}(h)$ does not tell the full story, as we measure the size of a dictionary by the number of key-value pairs it contains and not the height of some underlying tree. A binary tree of height $h$ can have anywhere from $h$ to $2^h - 1$ nodes, and so in the worst case, a tree of $n$ nodes can have height $n$. This leads to a worst-case running time of $\mathcal{O}(n)$ for all three of these algorithms (and again, you can show that this bound is tight).

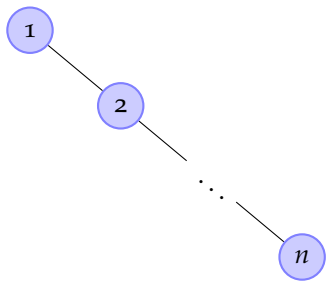
But given that the *best* case for the height of a tree of $n$ nodes is $\log n$, it seems as though a tree of $n$ nodes having height anywhere close to $n$ is quite extreme – perhaps we would be very "unlucky" to get such trees. As you'll show in the exercises, the deficiency is not in the BST property itself, but how we implement insertion and deletion. The simple algorithms we presented above make no effort to keep the height of the tree small when adding or removing values, leaving it quite possible to end up with a very linear-looking tree after repeatedly running these operations. So the question is: *can we implement BST insertion and deletion to not only insert/remove a key, but also keep the tree's height (relatively) small?*

---

## Exercise Break!

3.1 Prove that the worst-case running time of the naïve SEARCH, INSERT, and DELETE algorithms given in the previous section run in time $\Omega(h)$, where $h$ is the height of the tree.

3.2 Consider a BST with $n$ nodes and height $n$, structured as follows (keys shown):




Suppose we pick a random key between 1 and $n$, inclusive. Compute the expected number of *key comparisons* made by the search algorithm for this BST and chosen key.

3.3 Repeat the previous question, except now $n = 2^h - 1$ for some $h$, and the BST is complete (so it has height exactly $h$).

3.4 Suppose we start with an empty BST, and want to insert the keys 1 through $n$ into the BST.

(a) What is an order we could insert the keys so that the resulting tree has height $n$? (Note: there's more than one right answer.)

(b) Assume $n = 2^h - 1$ for some $h$. Describe an order we could insert the keys so that the resulting tree has height $h$.

(c) Given a random permutation of the keys 1 through $n$, what is the *probability* that if the keys are inserted in this order, the resulting tree has height $n$?

(d) (Harder) Assume $n = 2^h - 1$ for some $h$. Given a random permutation of the keys 1 through $n$, what is the *probability* that if the keys are inserted in this order, the resulting tree has height $h$?

---

## *AVL Trees*

Well, of course we can improve on the naïve SEARCH and DELETE – otherwise we wouldn't talk about binary trees in CS courses nearly as much as we do. Let's focus on insertion first for some intuition. The problem with the insertion algorithm above is that it always inserts a new key as a leaf of the BST, without changing the position of any other nodes. This renders the structure of the BST completely at the mercy of the order in which items are inserted, as you investigated in the previous set of exercises.

Note that for inserting a new key, there is only one leaf position it could go into which satisfies the BST property.

Suppose we took the following "just-in-time" approach. After each insertion, we compute the size and height of the BST. If its height is too large (e.g., $> \sqrt{n}$), then we do a complete restructuring of the tree to reduce the height to $\lceil \log n \rceil$. This has the nice property that it enforces some maximum limit on the height of the tree, with the downside that rebalancing an entire tree does not seem so efficient.

You can think of this approach as attempting to maintain an invariant on the data structure – the BST height is roughly $\log n$ – but only enforcing this invariant when it is extremely violated. Sometimes, this does in fact lead to efficient data structures, as we'll study in Chapter 8. However, it turns out that in the present case, being stricter with an invariant – enforcing it at *every* operation – leads to a faster implementation, and this is what we will focus on for this chapter.

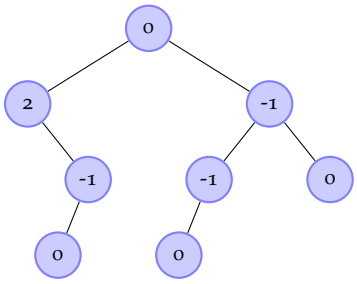
More concretely, we will modify the INSERT and DELETE algorithms so that they always perform a check for a particular "balanced" invariant. If this invariant is violated, they perform some minor *local* restructuring of the tree to restore the invariant. Our goal is to make both the check and restructuring as simple as possible, to not increase the asymptotic worst-case running times of $\mathcal{O}(h)$.

The implementation details for such an approach turn solely on the choice of invariant we want to preserve. This may sound strange: can't we just use the invariant "the height of the tree is $\leq \lceil \log n \rceil$"? It turns out that even though this invariant is the optimal in terms of possible height, it requires too much work to maintain every time we mutate the tree. Instead, several weaker invariants have been proposed and used in the decades that BSTs have been studied, and corresponding names coined for the different data structures. In this course, we will look at one of the simpler invariants, used in the data structure known as the **AVL tree**.

Data structures that do this work include red-black trees and 2-3-4 trees.

*The AVL tree invariant*

In a full binary tree ($2^h - 1$ nodes stored in a binary tree of height $h$), every node has the property that the height of its left subtree is equal to the height of its right subtree. Even when the binary tree is *complete*, the heights of the left and right subtrees of any node differ by at most 1. Our next definitions describe a slightly looser version of this property.



Each node is labelled by its balance factor.

**Definition 3.1** (balance factor)**.** The **balance factor** of a node in a binary tree is the height of its right subtree minus the height of its left subtree.

**Definition 3.2** (AVL invariant, AVL tree)**.** A node satisfies the **AVL invariant** if its balance factor is between -1 and 1. A binary tree is **AVL-balanced** if all of its nodes satisfy the AVL invariant.

An **AVL tree** is a binary *search* tree that is AVL-balanced.

The balance factor of a node lends itself very well to our style of recursive algorithms because it is a *local* property: it can be checked for a given node just by looking at the subtree rooted at that node, without knowing about the rest of the tree. Moreover, if we modify the implementation of the binary tree node so that each node maintains its height as an attribute, then we can check whether a node satisfies the AVL invariant in constant time!

There are two important questions that come out of this invariant:

- How do we preserve this invariant when inserting and deleting nodes?
- How does this invariant affect the height of an AVL tree?

These can be reframed as, "how much complexity does this invariant add?" and "what does this invariant buy us?"

For the second question, the intuition is that if each node's subtrees are almost the same height, then the whole tree is pretty close to being complete, and so should have small height. We'll make this more precise a bit later in this chapter. But first, we turn our attention to the more algorithmic challenge of modifying the naïve BST insertion and deletion algorithms to preserve the AVL invariant.

---

### Exercise Break!

3.5 Give an $\mathcal{O}(n \log n)$ algorithm for taking an arbitrary BST of size $n$ and modifying it so that its height becomes $\lceil \log n \rceil$.

3.6 Investigate the balance factors for nodes in a complete binary tree. How many nodes have a balance factor of 0? -1? 1?

---

*Rotations*

As we discussed earlier, our high-level approach is the following:

(i) Perform an insertion/deletion using the old BST algorithm.

(ii) If any nodes have the balance factor invariant violated, restore the AVL invariant.

How do we restore the AVL invariant? Before we get into the nitty-gritty details, we first make the following global observation: *inserting or deleting a node can only change the balance factors of its ancestors.* This is because inserting/deleting a node can only change the height of the subtrees that contain this node, and these subtrees are exactly the ones whose roots are ancestors of the node. For simplicity, we'll spend the remainder of this section focused on insertion; AVL deletion can be performed in almost exactly the same way.

Even better, the naïve algorithms already traverse exactly the nodes that are ancestors of the modified node. So it is extremely straightforward to check and restore the AVL invariant for these nodes; we can simply do so after the recursive INSERT, DELETE, EXTRACTMAX, or EXTRACTMIN call. So we go down the tree to search for the correct spot to insert the node, and then go back up the tree to restore the AVL invariant. Our code looks like the following (only INSERT is shown; DELETE is similar):

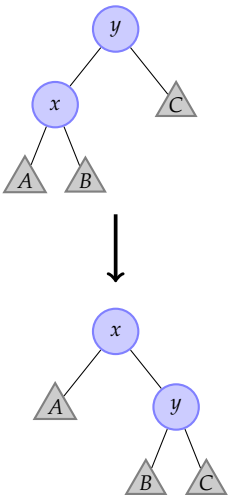
```
6  def Insert(D, key, value):
7      # Insert the key-value pair into <D>
8      # Same code as before, omitted
9      ...
10
11     D.balance_factor = D.right.height - D.left.height
12     if D.balance_factor < -1 or D.balance_factor > 1:
13       # Fix the imbalance for the root node.
14       fix_imbalance(D)
15
16     # Update the height attribute
17     D.height = max(D.left.height, D.right.height) + 1
```

Let us spell out what `fix_imbalance` must do. It gets as input a BST where the root's balance factor is less than $-1$ or greater than 1. However, because of the recursive calls to `Insert`, we can assume that *all the non-root nodes in D satisfy the AVL invariant*, which is a big help: all we need to do is fix the root.

Reminder here about the power of recursion: we can assume that the recursive INSERT calls worked properly on the subtree of `D`, and in particular made sure that the subtree containing the new node is an AVL tree.

One other observation is a big help. We can assume that at the beginning of every insertion, the tree is already an AVL tree – that it is balanced. Since inserting a node can cause a subtree's height to increase by at most 1, each node's balance factor can change by at most 1 as well. Thus if the root does not satisfy the AVL invariant after an insertion, its balance factor can *only* be -2 or 2.

These observations together severely limit the "bad cases" that we are responsible for fixing. In fact, these restrictions make it quite straightforward to define a small set of simple, constant-time procedures to restructure the tree to restore the balance factor in these cases. These procedures are called *rotations*.




### *Right rotation*

Rotations are best explained through pictures. In the top diagram, variables $x$ and $y$ are keys, while the triangles $A$, $B$, and $C$ represent arbitrary subtrees (that could consist of many nodes). We assume that all the nodes except $y$ satisfy the AVL invariant, and that the balance factor of $y$ is -2.

This means that $y$s left subtree must have height 2 greater than its right subtree, and so $A.height = C.height + 1$ or $B.height = C.height + 1$. We will first consider the case $A.height = C.height + 1$.

In this case, we can perform a **right rotation** to make $x$ the new root, moving around the three subtrees and $y$ as in the second diagram to preserve the BST property. It is worth checking carefully that this rotation does indeed restore the invariant.

**Lemma 3.1** (Correctness of right rotation)**.** Let $x$, $y$, $A$, $B$, and $C$ be defined as in the margin figure. Assume that this tree is a binary search tree, and that $x$ and every node in $A$, $B$, and $C$ satisfy the balance factor invariant. Also assume that $A.height = C.height + 1$, and $y$ has a balance factor of -2.

Then applying a right rotation to the tree results in an AVL tree, and in particular, $x$ and $y$ satisfy the AVL invariant.

*Proof.* First, observe that whether or not a node satisfies the balance factor invariant only depends on its descendants. Then, since the right rotation doesn't change the internal structure of $A$, $B$, and $C$, all the nodes in these subtrees still satisfy the AVL invariant after the rotation. So we only need to show that both $x$ and $y$ satisfy the invariant.

This is the power of having a local property like the balance factor invariant. Even though $A$, $B$, and $C$ move around, their contents don't change.

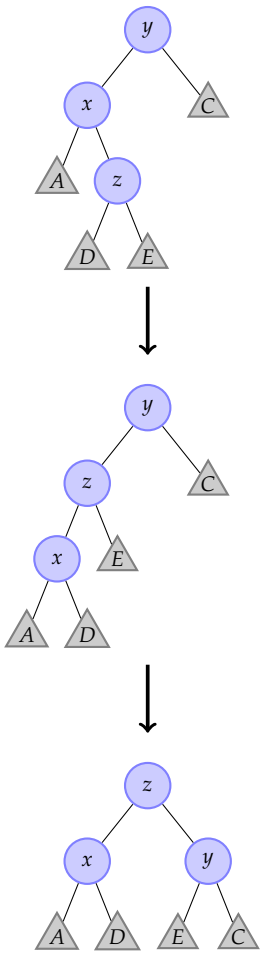
- Node $y$. The new balance factor of $y$ is $C.height - B.height$. Since $x$ originally satisfied the balance factor, we know that $B.height \geq A.height - 1 = C.height$. Moreover, since $y$ originally had a balance factor of $-2$, we know that $B.height \leq C.height + 1$. So $B.height = C.height$ or $B.height = C.height + 1$, and the balance factor is either -1 or 0.
- Node $x$. The new balance factor of $x$ is $(1 + \max(B.height, C.height)) - A.height$. Our assumption tells us that $A.height = C.height + 1$, and as we just observed, either $B.height = C.height$ or $B.height = C.height + 1$, so the balance factor of $x$ is 0 or 1.

■

### *Left-right rotation*

What about the case when $B.height = C.height + 1$ and $A.height = C.height$? Well, before we get ahead of ourselves, let's think about what would happen if we just applied the same right rotation as before.

The diagram looks exactly the same, and in fact the AVL invariant for $y$ still holds. The problem is now the relationship between $A.height$ and $B.height$: because $A.height = C.height = B.height - 1$, this rotation would leave $x$ with a balance factor of 2. Not good.




Since in this case $B$ is "too tall," we will break it down further and move its subtrees separately. Keep in mind that we're still assuming the AVL invariant is satisfied for every node except the root $y$.

By our assumption, $A.height = B.height - 1$, and so both $D$ and $E$'s heights are either $A.height$ or $A.height - 1$. Now we can first perform a **left rotation** rooted at $x$, which is symmetric to the right rotation we saw above, except the root's right child becomes the new root.

This brings us to the situation we had in the first case, where the left subtree of $z$ is at least as tall as the right subtree of $z$. So we do the same thing and perform a right rotation rooted at $y$. Note that we're treating the entire subtree rooted at $x$ as one component here.

Given that we now have two rotations to reason about instead of one, a formal proof of correctness is quite helpful.

**Lemma 3.2.** Let $A$, $C$, $D$, $E$, $x$, $y$, and $z$ be defined as in the diagram. Assume that $x$, $z$, and every node in $A, C, D, E$ satisfy the AVL invariant. Furthermore, assume the following restrictions on heights, which cause the balance factor of $y$ to be -2, and for the $D - z - E$ subtree ("$B$") to have height $C.height + 1$.

(i) $A.height = C.height$

(ii) $D.height \leq C.height$ and $E.height \leq C.height$, and one of them is equal to $C.height$.

Then after performing a left rotation at $x$, then a right rotation at $y$, the resulting tree is an AVL tree. This combined operation is sometimes called a *left-right double rotation*.

*Proof.* As before, because the subtrees $A$, $C$, $D$, and $E$ don't change their internal composition, all their nodes still satisfy the AVL invariant after the transformation. We need to check the balance factor for $x$, $y$, and $z$:

- Node $x$: by assumption (ii) and the fact that $z$ originally satisfied the AVL invariant, $D.height = C.height$ or $D.height = C.height - 1$. So then the balance factor of $x$ is $D.height - A.height$, which is either 0 or -1. (By (i), $A.height = C.height$.)
- Node $y$: as in the above argument, either $E.height = C.height$ or $E.height = C.height - 1$. Then the balance factor of $y$ is $C.height - E.height$, which is either 0 or 1.
- Node $z$: since $D.height \leq A.height$ and $E.height \leq C.height$, the balance factor of $z$ is equal to $(C.height + 1) - (A.height + 1)$. By assumption (i), $A.height = C.height$, so the balance factor of $z$ is 0.

■

We leave it as an exercise to think about the cases where the *right* subtree's height is 2 more than the left subtree's. The arguments are symmetric; the two relevant rotations are "left" and "right-left" rotations.

### *AVL tree implementation*

We now have a strong lemma telling us that we can rearrange the tree in constant time to restore the balanced property. This is great for our insertion and deletion algorithms, which only ever change one node. Here is our full AVL tree INSERT algorithm:

```
6  def Insert(D, key, value):
7      # Insert the key-value pair into <D>
8      # Same code as before, omitted
9      ...
10
11     D.balance_factor = D.right.height - D.left.height
12     if D.balance_factor < -1 or D.balance_factor > 1:
13       # Fix the imbalance for the root node.
14       fix_imbalance(D)
15
16     # Update the height attribute
17     D.height = max(D.left.height, D.right.height) + 1
18
19
20 def fix_imbalance(D):
21     # Check balance factor and perform rotations
22     if D.balance_factor == -2:
23         if D.left.left.height == D.right.height + 1:
24             right_rotate(D)
25         else:  # D.left.right.height == D.right.height + 1
26             left_rotate(D.left)
27             right_rotate(D)
28     elif D.balance_factor == 2:
29         # left as an exercise; symmetric to above case
30         ...
31
32
33 def right_rotate(D):
34     # Using some temporary variables to match up with the diagram
35     y = D.root
36     x = D.left.root
37     A = D.left.left
38     B = D.left.right
39     C = D.right
```

You'll notice that we've deliberately left some details to the exercises. In particular, we want you to think about the symmetric actions for detecting and fixing an imbalance.

```
    D.root = x
    D.left = A
    # Assume access to constructor AVLTree(root, left, right)
    D.right = AVLTree(y, B, C)


def left_rotate(D):
    # Left as an exercise
```

**Theorem 3.3** (AVL Tree Correctness)**.** The above AVL Tree insertion algorithm is correct. That is, it results in a binary tree that satisfies the binary search tree property and is balanced, with one extra node added, and with the `height` and `balance_factor` attributes set properly.

*Proof.* Since insertion starts with the naïve algorithm, we know that it correctly changes the contents of the tree. We need only to check that `fix_imbalance` results in a balanced BST.

Right before `fix_imbalance` is called in either function, we know that `D` still satisfies the BST property, and that its subtrees are AVL trees (since we assume that the recursive calls correctly restore this property). Moreover, because at most one node has changed in the BST, the balance factor of the root is between -2 and 2.

The correctness of `fix_imbalance` when the root's balance factor is -2 is directly implied by the previous two lemmas. The other case is symmetric, and was left as an exercise. Then we are done, since after `fix_imbalance` returns, the tree is an AVL tree – all its nodes satisfy the AVL invariant. ■

## *Analysis of AVL Algorithms*

In the preceding sections, we put in quite a bit of effort in ensuring that the AVL invariant is preserved for every node in the tree. Now, it is time to reap the benefits of this additional restriction on the structure of the tree in our running time analysis of the modified INSERT and DELETE algorithms.

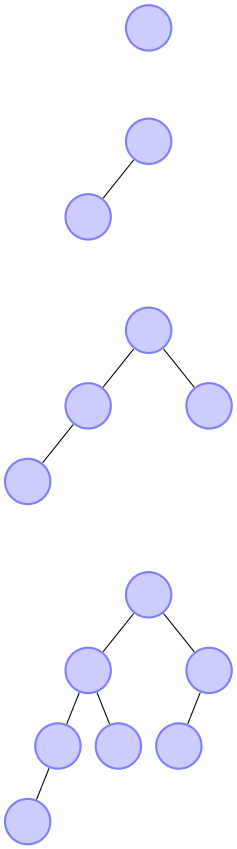
First, a simple lemma that should look familiar, as it is the basis of most recursive tree algorithm analyses.

**Lemma 3.4.** The worst-case running time of AVL tree insertion and deletion is $\mathcal{O}(h)$, the same as for the naïve insertion and deletion algorithms.

Don't forget that we've only proved a $\mathcal{O}(h)$ upper bound in these notes – you're responsible for proving a lower bound yourself!

*Proof.* We simply observe that the new implementation consists of the old one, plus the new updates of `height` and `balance_factor` attributes, and at most two rotations. This adds a constant-time overhead per recursive call, and as before, there are $\mathcal{O}(h)$ recursive calls made. So the total running time is still $\mathcal{O}(h)$. ■

So far, the analysis is exactly the same as for naïve BSTs. Here is where we diverge to success. As we observed at the beginning of this section, BSTs can be extremely imbalanced, and have height equal to the number of nodes in the tree. This is obviously not true for AVL trees, which cannot have all their nodes on (say) the left subtree. But AVL trees are not perfectly balanced, either, since each node's two subtree heights can differ by 1. So the question is, is restricting the balance factor for each node to the range $\{-1, 0, 1\}$ good enough to get a strong bound on the height of the AVL tree? The answer is a satisfying *yes*.



**Lemma 3.5** (AVL Tree Height)**.** An AVL tree with $n$ nodes has height at most $1.441 \log n$ (for large enough $n$).

*Proof.* It turns out to be easier to answer a question that is inverse to the statement in the theorem: what is $N(h)$, the *minimum* number of nodes in an AVL tree of height $h$? Our goal is to get a tight bound (or exact formula) for $N(h)$. Some playing around quickly yields the following small values: $N(0) = 0$, $N(1) = 1$, $N(2) = 2$, $N(3) = 4$, $N(4) = 7$.

Not much of a pattern so far, though drawing pictures is suggestive: an AVL tree of height $h$ with minimum size must have a subtree that is an AVL tree of height $h - 1$, and another subtree that is an AVL tree of height $h - 2$, and both of its subtrees must have minimum size. Here we're implicitly using the fact that being balanced, and hence an AVL tree, is a recursive property: the subtrees of an AVL tree are themselves AVL trees. This gives the following recurrence:

$$N(h) = \begin{cases} 0, & h = 0 \\ 1, & h = 1 \\ N(h-1) + N(h-2) + 1, & h \geq 2 \end{cases}$$

This looks almost, but not quite, the same as the Fibonacci sequence. A bit of trial-and-error or more sophisticated techniques for solving recurrences (beyond the scope of this course) reveal that $N(h) = f_{h+2} - 1$, where $f_i$ is the $i$-th Fibonacci number. Since we do have a closed form expression for the Fibonacci numbers in terms of the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$, we get the following bound on $N(h)$:

$$\begin{aligned} N(h) &> f_h \\ &> \frac{1}{\sqrt{5}}\varphi^h - 1 \end{aligned}$$

Recall now the definition of $N(h)$: the minimum number of nodes in an AVL tree of height $h$. So then if we have $n$ nodes in an AVL tree of height $h$, we know that

$$
\begin{aligned}
N(h) &\leq n \\
\frac{1}{\sqrt{5}}\varphi^h - 1 &\leq n \\
\frac{1}{\sqrt{5}}\varphi^h &\leq n+1 \\
\log_\varphi\left(\frac{1}{\sqrt{5}}\varphi^h\right) &\leq \log_\varphi(n+1) \\
\log_\varphi\left(\frac{1}{\sqrt{5}}\right) + h &\leq \log_\varphi(n+1) \\
h &\leq \log_\varphi(n+1) - \log_\varphi\left(\frac{1}{\sqrt{5}}\right) \\
h &\leq 1.4405\log(n+1) - \log_\varphi\left(\frac{1}{\sqrt{5}}\right) \\
h &\leq 1.4405\log(n+1) + 1.68
\end{aligned}
$$

■

Putting the previous two lemmas together, we can conclude that AVL insertion, deletion, and search all have *logarithmic* running time in the worst-case.

**Theorem 3.6** (AVL Tree Runtime)**.** AVL tree insertion, deletion, and search have worst-case running time $\Theta(\log n)$, where $n$ is the number of nodes in the tree.

---

## Exercise Break!

3.7 Show that you can use an AVL tree to implement a priority queue. What is an upper bound on the worst-case running time of the three priority queue operations in this case?

---

# 4 Dictionaries, Round Two: Hash Tables

We now look at a completely different approach for implementing dictionaries. To get warmed up, consider the following scenario: all of the key-value pairs we wish to store have keys that are numbers 0 through $K-1$. In this case, we can store these pairs inside an array of length $K$, and achieve **constant time** search, insertion, and deletion simply by storing each value at the array index corresponding to its key!

This technique is known as **direct addressing**, and works well when the number of possible keys (and hence total number of pairs) is small. However, array addressing is only constant time if the entire array has been allocated in memory. Thus as $K$ grows larger, even if the running time of the dictionary operations stays constant, the space required grows linearly in $K$. In this chapter, we will study a generalized version of direct addressing known as *hashing*, which allows us to store key-value pairs when the keys aren't necessarily numbers, and using much less space than the number of all possible keys.

This size constraint is infeasible in many real-world applications of dictionaries.

## *Hash functions*

Let $U$ represent the set of all possible keys we would like to store. In general, we do not restrict ourselves to just numeric keys, and so $U$ might be a set of strings, floating-point numbers, or other custom data types that we define based on our application. Like direct addressing, we wish to store these keys in an array, but here we'll let the size of the array be a separate variable $m$. We won't assume anything about the relationship between $m$ and $|U|$ right now, though you might already have some ideas about that.

**Definition 4.1** (hash function, hash table)**.** A **hash function** is a function $h : U \to \{0, 1, \ldots, m-1\}$. You should think of this as a function that takes a key and computes the array slot where the key is to be placed.

A **hash table** is a data structure containing an array of length $m$ and a hash function.

```
6 def Search(hash_table, key):
7     hash = h(key)
8     return hash_table[hash]
9
```

```
def Insert(hash_table, key, value):
    hash = h(key)
    hash_table[hash] = value

def Delete(hash_table, key):
    hash = h(key)
    hash_table[hash] = None
```

Of course, this is not the end of the story. Even though this implementation is certainly *efficient* – all three operations run in $\Theta(1)$ time – it actually is not guaranteed to be correct.

Remember that we have not made any assumptions about the relationship between $m$, the size of the hash table, and $|U|$, the total possible number of keys; nor have we said anything about what $h$ actually does. Our current implementation makes the assumption that there are no *collisions* in the hash function; i.e., that each key gets mapped to a unique array slot. Consider what happens when you try to use the above algorithm to insert two key-value pairs whose hashed keys are the same index $i$: the first value is stored at index $i$, but then the second value overwrites it.

Now, if $m \geq |U|$, i.e, there are at least as many array slots as possible keys, then there always exists a hash function $h$ which has no collisions. Such hash functions are called **perfect hash functions**, earning the superlative name because they enable the above naive hash table implementation to be correct. However, this assumption is often not realistic, as we typically expect the number of possible keys to be extremely large, and the array to not take up too much space. More concretely, if $m < |U|$, then at least one collision is guaranteed to occur, causing the above implementation to fail. The remainder of this chapter will be devoted to discussing two strategies for handling collisions.

Of course, even when we know that a perfect hash function exists for a given $U$ and $m$, it is sometimes non-trivial to find and compute it. Consider the case when $U$ is a set of 10 000 names – can you efficiently implement a perfect hash function without storing a sorted list of the names?

## *Closed addressing ("chaining")*

With **closed addressing**, each array element no longer stores a value associated with a particular key, but rather a pointer to a *linked list* of key-value pairs. Thus, collisions are sidestepped completely by simply inserting both the key and value into the linked list at the appropriate array index. Search and deletion are just slightly more complex, requiring a traversal of the linked list to find the correct key-value pair.

this approach is also called "chaining"

Why is it necessary to store both the key and the value?

```
def Search(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    if linked_list contains key:
        return corresponding value from linked_list
    else:
```

```
        return None

def Insert(hash_table, key, value):
    hash = h(key)
    linked_list = hash_table[hash]
    insert key, value at the head of linked_list

def Delete(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    delete key from linked_list
```

This implementation maintains the invariant that for every key $k$ stored in the table, it must appear in the linked list at position $h(k)$. From this observation, we can see that these algorithms are correct. In addition, these algorithms only consider the linked list at one particular array index, saving lots of time.

Or perhaps not. What if all of the keys get mapped to the same index? In this case, the worst case of search and deletion reduces to the worst-case search and deletion from a linked list, which is $\Theta(n)$, where $n$ is the number of pairs stored in the hash table. Can we simply pick a hash function that guarantees that this extreme case won't happen? Not if we allow the set of keys to be much larger than the size of the array: for any hash function $h$, there exists an index $i$ such that at least $|U|/m$ keys get mapped to $i$ by $h$.

This type of counting argument is extremely common in algorithm analysis. If you have a set of numbers whose average is $x$, then one of the numbers must have value $\geq x$.

So while the worst-case running time for INSERT is $\Theta(1)$,, the worst case performance for `Search` and `Delete` seems very bad: no better than simply using a linked list, and much worse than the AVL trees we saw in the previous section. However, it turns out that if we pick good hash functions, hash tables with closed addressing are much better than both of these data structures *on average*.

Insertions happen at the front of the linked list

### *Average-case analysis for closed addressing*

We will focus on analysing the SEARCH algorithm, whose running time of course depends on the lengths of the linked lists in the hash table. Remember that whenever we want to talk about average case, we need to define a distribution of inputs for the operation. We'll assume here that all possible keys are equally likely to be searched for; in other words, we'll use the uniform distribution on $U$, and assume $n$ random keys are already contained in the hash table.

DELETE is basically the same.

However, because the lengths of the linked lists depend on the number of collisions in our hash function, just knowing the key distribution isn't enough – for all we know, our hash function could map all keys to the same index! So we will make an additional assumption known as the **simple uniform hashing assumption**, which is that our hash function $h$ satisfies the property that for all $0 \leq i < m$, the probability that $h(k) = i$ is $\frac{1}{m}$, when $k$ is chosen uniformly at random from $U$. That is, choosing a random key doesn't favour one particu-

lar index over any other. We will measure running time here as the number of keys compared against the search input, plus the time for computing the hash function.

First, assume that the key $k$ being searched for is *not* in the hash table. In this case, the algorithm first computes $h(k)$, and then must traverse the entire linked list stored at index $h(k)$. We'll assume that the first step takes constant time, and we know that the second step takes time proportional to the length of the linked list at index $h(k)$. Let $T$ be the running time of SEARCH on input key $k$. Then $T = 1+$ the length of the linked list stored at $h(k)$, which is equal to the number of the $n$ keys that hash to that particular index. We let $L$ be a random variable representing the number of keys hashed to $h(k)$. So, $T = 1 + L$, and $\mathbb{E}[T] = 1 + \mathbb{E}[L]$. But, by the simple uniform hashing assumption, the probability that each of the $n$ randomly chosen keys already in the hash table have a hash value of $h(k)$ is $\frac{1}{m}$, and so $\mathbb{E}[L] = \frac{n}{m}$.

More on this later

The average number of keys checked by SEARCH when the key is not in the hash table is therefore $1 + \frac{n}{m}$. The average-case running time is then $\Theta(1 + \frac{n}{m})$. Remember what these variables mean: $n$ is the number of keys already stored in the hash table, and $m$ is the number of spots in the array. This running time tells us that if the number of keys in the hash table is any constant multiple of the size of the array, then doing a search when the key is not in the hash table is a constant-time operation on average.

This is the well-known expected value for a *hypergeometric distribution*.

What about when the search is successful, i.e., when we search for a key that is already in the hash table? Consider searching for $k_i$, the $i$-th key inserted into the hash table. How many keys are checked during this search? Since items are inserted at the *front* of the linked list, the number of keys searched is equal to the number of keys inserted after $k_i$ into the same index as $k_i$, plus one for examining $k_i$ itself. Since $n - i$ keys are inserted after $k_i$, by the simple uniform hashing assumption the expected number of keys inserted into $h(k_i)$ after $k_i$ is $\frac{n-i}{m}$. So then to compute the average running time, we add the cost of computing the hash function (just 1 step) plus the average of searching for each $k_i$:

$$\begin{aligned}
\mathbb{E}[T] &= 1 + \frac{1}{n}\sum_{i=1}^{n} \text{expected number of keys visited by search for } k_i \\
&= 1 + \frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) \\
&= 2 + \frac{1}{n}\sum_{i=1}^{n}\frac{n}{m} - \frac{1}{n}\sum_{i=1}^{n}\frac{i}{m} \\
&= 2 + \frac{n}{m} - \frac{1}{n}\cdot\frac{n(n+1)}{2m} \\
&= 2 + \frac{n}{m} - \frac{n+1}{2m} \\
&= 2 + \frac{n}{2m} - \frac{1}{2m} \\
&= \Theta\left(1 + \frac{n}{m}\right)
\end{aligned}$$

The average-case running time for DELETE is the same, since once a key has

been found in an array, it is only a constant-time step to remove that node from the linked list.

Because the ratio $\frac{n}{m}$ between the number of keys stored and the number of spots in the array comes up frequently in hashing analysis, we give it a special name.

**Definition 4.2** (load factor)**.** The **load factor** of a hash table is the ratio of the number of keys stored to the size of the table. We use the notation $\alpha = \frac{n}{m}$, where $n$ is the number of keys, and $m$ the number of slots.

So we can say that the average-case running time of SEARCH and DELETE are $\Theta(1+\alpha)$.

---

### Exercise Break!

4.1 Suppose that we use an AVL tree instead of a linked list to store the key-value pairs that are hashed to the same index.

(a) What is the *worst-case* running time of a search when the hash table has $n$ key-value pairs?

(b) What is the *average-case* running time of an unsuccessful search when the hash table has $n$ key-value pairs?

---

## *Open addressing*

The other strategy used to resolve collisions is to require each array element to contain only one key, but to allow keys to be mapped to alternate indices when their original spot is already occupied. This saves the space overhead required for storing extra references in the linked lists of closed addressing, but the cost is that the load factor $\alpha$ must always be less than 1, i.e., the number of keys stored in the hash table cannot exceed the length of the array.

In this type of hashing, we have a *parameterized* hash function $h$ that takes *two* arguments, a key and a positive integer. The "first" hash value for a key $k$ is $h(k,0)$, and if this spot is occupied, the next index chosen is $h(k,1)$, and then $h(k,2)$, etc.

$h : U \times \mathbb{N} \to \{0, \ldots, m-1\}$

```
5  def Insert(hash_table, key, value):
6      i = 0
7      while hash_table[h(key, i)] is not None:
8          i = i + 1
9
10     hash_table[h(key, i)].key = key
11     hash_table[h(key, i)].value = value
```

For simplicity, we have omitted a bounds check $i < m$. This would be necessary in practice to ensure that this code does not enter into an infinite loop.

Searching for an item requires examining not just one spot, but many spots. Essentially, when searching for a key $k$, we visit the same sequence of indices $h(k,0)$, $h(k,1)$, etc. until either we find the key, or reach a `None` value.

```
14 def Search(hash_table, key):
15     i = 0
16     while hash_table[h(key, i)] is not None and hash_table[h(key, i)].key != key:
17         i = i + 1
18 
19     if hash_table[h(key, i)].key == key:
20         return hash_table[h(key, i)].value
21     else:
22         return None
```

Deletion seems straightforward – simply search for the given key, then replace the item stored in the array with `None`, right? Not exactly. Suppose we insert a key $k$ into the hash table, but the spot $h(k,0)$ is occupied, and so this key is stored at index $h(k,1)$ instead. Then suppose we delete the pair that was stored at $h(k,0)$. Any subsequent search for $k$ will start at $h(k,0)$, find it empty, and so return `None`, not bothering to continue checking $h(k,1)$.

So instead, after we delete an item, we replace it with a special value `Deleted`, rather than simply `None`. This way, the SEARCH algorithm will not halt when it reaches an index that belonged to a deleted key.

We leave it as an exercise to modify the INSERT algorithm so that it stores keys in `Deleted` positions as well.

```
25 def Delete(hash_table, key):
26     i = 0
27     while hash_table[h(key, i)] is not None and hash_table[h(key, i)].key != key:
28         i = i + 1
29 
30     if hash_table[h(key, i)].key == key:
31         hash_table[h(key, i)] = Deleted
```

It remains to discuss how to implement different parameterized hash functions and to study the properties of the *probe sequences* $[h(k,0), h(k,1), \dots]$ generated by these different strategies. After all, since all three dictionary operations require traversing one of these probe sequences until reaching an empty spot, a natural question to ask is how the probe sequences for different keys overlap with each other, and how these overlaps affect the efficiency of open addressing.

### *Linear probing*

The simplest implementation of open addressing is to start at a given hash value, and then keep adding some fixed offset to the index until an empty spot is found.

**Definition 4.3** (Linear probing)**.** Given a hash function $hash : U \to \{0, \ldots, m-1\}$ and number $b \in \{1, \ldots, m-1\}$, we can define the parameterized hash function $h$ for **linear probing** as follows:

$$h(k, i) = hash(k) + bi \pmod{m}.$$

Recall that $\pmod{m}$ takes the remainder when divided by $m$.

The corresponding **linear probing sequence** for key $k$ is $hash(k, 0)$, $hash(k, 1)$, $hash(k, 2)$, ...

When $d = 1$, the probe sequence is simply $hash(k), hash(k) + 1, \ldots$

Unfortunately, this strategy suffers from a fundamental problem of *clustering*, in which contiguous blocks of occupied locations are created, causing further insertions of keys into any of these locations to take a long time. For example, suppose $d = 1$, and that three keys are inserted, all with the hash value 10. These will occupy indices 10, 11, and 12 in the hash table. But now any new key with hash value 11 will require 2 spots to be checked and rejected because they are full. The collision-handling for 10 has now ensured that hash values 11 and 12 will also have collisions, increasing the runtime for any operation that visits these indices. Moreover, the effect is cumulative: a collision with any index in a cluster causes an extra element to be added at the end of the cluster, growing it in size by one. You'll study the effects of clustering more precisely in an exercise.

### *Quadratic probing*

The main problem with linear probing is that the hash values in the middle of a cluster will follow the exact same search pattern as a hash value at the beginning of the cluster. As such, more and more keys are absorbed into this long search pattern as clusters grow. We can solve this problem using quadratic probing, which causes the offset between consecutive indices in the probe sequence to increase as the probe sequence is visited.

**Definition 4.4** (Quadratic probing)**.** Given a hash function $hash : U \to \{0, \ldots, m-1\}$ and numbers $b, c \in \{0, \ldots, m-1\}$, we can define the parameterized hash function $h$ for **quadratic probing** as follows:

$$h(k, i) = hash(k) + bi + ci^2 \pmod{m}.$$

Even if a key's hash value is in the middle of a cluster, it is able to "escape" the cluster much more quickly than using linear probing. However, a form of clustering still occurs: if many items have the same initial hash value, they still follow the exact same probe sequence. For example, if four keys are hashed to the same index, the fourth key inserted must examine (and pass) the three indices occupied by the first three keys.

### *Double hashing*

To resolve this problem – a collision in the original hash function causing identical probe sequences – we change how the offset is calculated so that it depends on the key being hashed in a particularly clever way.

**Definition 4.5** (Double hashing)**.** Given *two* hash functions $hash_1, hash_2 : U \to \{0, \ldots, m-1\}$, we can define the parameterized hash function for **double hashing** $h$ as follows:

$$h(k, i) = hash_1(k) + i \cdot hash_2(k) \pmod{m}.$$

While both linear and quadratic probing have at most $m$ different probe sequences (one for each distinct starting index), double hashing has at most $m^2$ different probe sequences, one for each pair of values $(hash_1(k), hash_2(k))$. Under such a scheme, it is far less likely for large clusters to form: not only would keys have to have the same initial hash value (using $hash_1$), but they would also have to have the same offset (using $hash_2$).

### *Running time of open addressing*

To round out this chapter, we'll briefly discuss some of the performance characteristics of open addressing. First, we observe that there are $m!$ possible probe sequences (i.e., the order in which hash table indices are visited). A typical simplifying assumption is that the probability of getting any one of these sequences is $1/m!$, which is not true for any of the probe sequences studied here – even double hashing has only $m^2$ possible sequences – but can be a useful approximation.

However, under this assumption, the average number of indices searched of all three dictionary operations is at most $\frac{1}{1-\alpha}$. So for example, if the table is 90% full (so $\alpha = 0.9$), the average number of indices searched would be just 10, which isn't bad at all!

---

## Exercise Break!

4.2 Suppose we use linear probing with $d = 1$, and our hash table has $n$ keys stored in the array indices 0 through $n - 1$.

(a) What is the maximum number of array indices checked when a new key (distinct from the ones already stored) is searched for in this hash table?

(b) What is the *probability* that this maximum number of array indices occurs? Use the simple uniform hashing assumption.

(c) What is the average running time of SEARCH for this array?

---

# 5 *Randomized Algorithms*

So far, when we have used probability theory to help analyse our algorithms we have always done so in the context of *average-case analysis*: define a probability distribution over the set of possible inputs of a fixed size, and then calculate the expected running time of the algorithm over this set of inputs.

This type of analysis is a way of refining our understanding of how quickly an algorithm runs, beyond simply the worst and best possible inputs, we look at the spread (or distribution) of running times across all different inputs. It allows us to say things like "Quicksort may have a worst-case running time of $\Theta(n^2)$, but on average its performance is $\Theta(n \log n)$, which is the same as mergesort."

However, average-case analysis has one important limitation: the importance of the asymptotic bound is directly correlated with the "plausibility" of the probability distribution used to perform the analysis. We can say, for example, that the average running time of quicksort is $\Theta(n \log n)$ when the input is a random, uniformly-chosen permutation of $n$ items; but if the input is instead randomly chosen among only the permutations which have almost all items sorted, the "average" running time might be $\Theta(n^2)$. We might assume that the keys inserted into a hash table are equally likely to be hashed to any of the $m$ array spots, but if they're all hashed to the same spot, searching for keys not already in the hash table will always take $\Theta(n)$ time.

This assumes naïve quicksort, which chooses the first element as the pivot.

In practice, an algorithm can have an *adversarial* relationship with the entity feeding it inputs. Not only is there no guarantee that the inputs will have a distribution close to the one we used in our analysis; a malicious user could conceivably hand-pick a series of inputs designed to realize the worst-case running time. That the algorithm has good average-case performance means nothing in the face of an onslaught of particularly bad inputs.

In this chapter, we will discuss one particularly powerful algorithm design technique which can (sometimes) be used to defend against this problem. Rather than assume the randomness exists external to the algorithm in the choice of inputs, this technique makes *no* assumptions about the input whatsoever, but instead moves the randomness to be inside the algorithm itself. The key idea is to allow our algorithms to make *random choices*, a departure from the deterministic algorithms — ones that must follow a fixed sequence of steps — that we have studied so far. Such algorithms, which make random choices as part of their natural execution, are known as **randomized algorithms**.

### *Randomized quicksort*

Let us return to quicksort from our first chapter, which we know has poor worst-case performance on pre-sorted lists. Because our implementation always chooses the first list element as the pivot, a pre-sorted list always results in maximally uneven partition sizes. We were able to "fix" this problem in our average-case analysis with the insight that in a random permutation, any of the $n$ input numbers is equally likely to be chosen as a pivot in any of the recursive calls to quicksort.

You'll recall our argument was a tad more subtle and relied on indicator variables, but this idea that the pivot was equally likely to be any number did play a central role.

So if we allow our algorithm to make random choices, we can turn any input into a "random" input simply by preprocessing it, and then applying the regular quicksort function:

```
def randomized_quicksort(A):
    randomly permute A
    quicksort(A)
```

Huh, this is so simple it almost feels like cheating. The only difference is the application of a random permutation to the input. The correctness of this algorithm is clear: because permuting a list doesn't change its contents, the exact same items will be sorted, so the output will be the same. What is more interesting is talking about the running time of this algorithm.

This isn't entirely true if the list contains duplicates, but there are ways of fixing this if we need to.

One important point to note is that for randomized algorithms, *the behaviour of the algorithm depends not only on its input, but also the random choices it makes*. Thus running a randomized algorithm multiple times on the same input can lead to different behaviours on each run. In the case of randomized quicksort, we know that the result — $A$ is sorted — is always the same. What can change is its running time; after all, it is possible to get as input a "good" input for the original quicksort (sorts in $\Theta(n \log n)$ time), but then after applying the random permutation the list becomes sorted, which is the worst input for quicksort.

In other words, randomly applying a permutation to the input list can cause the running time to increase, decrease, or stay roughly the same. So how do we draw conclusions about the running time of randomized quicksort if we can't even be certain how long it takes on a single input? Probability, of course!

More precisely, for a fixed input $A$ to randomized quicksort with length $n$, we can define the random variable $T_A$ to be the running time of the algorithm. The main difference between this random variable and the $T$ we used in the average-case analysis of Chapter 1 is that now we are considering the probability distribution that the algorithm uses to make its random choices, and *not* a probability distribution over inputs. In the case of randomized quicksort, we would say that the $T_A$ random variable has as its probability space the set of random permutations of $\{1, \ldots, n\}$ that the `randomized_quicksort` algorithm could pick from in its first step.

With this random variable defined, we can now talk about the *expected running*

*time* of randomized quicksort on the input $A$, $\mathbb{E}[T_A]$. First, while we won't prove it here, it is possible to randomly permute $A$ in $\mathcal{O}(n)$ time. Applying a random permutation on $A$ results in the standard quicksort running on a random permutation of $\{1, \ldots, n\}$ as its input, and so the analysis from Chapter 1 applies, and $E[T_A] = \Theta(n \log n)$.

See Section 5.3 of CLRS for details.

**Warning**: students often confuse this type of analysis with the average-case analysis we did earlier in the course. For average-case analysis, we have a completely deterministic algorithm and a probability distribution over a set of inputs, and are calculating the expected running time across this set of inputs. For the analysis in the present chapter, we have a randomized algorithm and a single input, and are calculating the expected running time across the possible random choices made by the algorithm itself.

You can think about average-case analysis as putting the randomness external to the algorithm, and randomized algorithm analysis as keeping the randomness internal to the algorithm.

The work we have done so far is just for one particular input. Let us extend this now to be able to say something about the overall running time of the algorithm. We define the **worst-case expected running time** of a randomized algorithm to be a function

$$ET(n) = \max\{\mathbb{E}[T_x] \mid x \text{ is an input of size } n\}.$$

For randomized quicksort, all lists of size $n$ have the same asymptotic expected running time, and so we conclude that randomized quicksort has a worst-case expected running time of $\Theta(n \log n)$.

You might hear the phrase "worst-case expected" and think this is a bit vague: it's not the "absolute" maximum running time, or even an "average" maximum. So what is this really telling us? The right way to think about worst-case expected running time is from the perspective of a malicious user who is trying to feed our algorithm bad inputs. With regular quicksort, it is possible to give the algorithm an input for which the running time is *guaranteed* to be $\Theta(n^2)$. However, this is not the case for randomized quicksort. Here, no matter what input the user picks, if the algorithm is run many times on this input, the average running time will be $\Theta(n \log n)$. Individual runs of randomized quicksort might take longer, but in the long run our adversary can't make our algorithm take more than $\Theta(n \log n)$ per execution.

## *Universal Hashing*

Now, let us turn to a well-known example of a randomized algorithm: universal hashing. Recall that the *simple uniform hashing assumption* from the previous chapter says that for any array index $0 \leq i < m$, the probability of choosing a key that hashes to $i$ is $\frac{1}{m}$. This assumption made the analysis tractable for us, and in particular ensured a $\Theta(1 + \alpha)$ average-case running time for the hash table operations. But as we observed, for *any* hash function we choose, a simple counting argument tells us that at least $|U|/m$ keys must be hashed to the same index. So if any subset of size $n$ of these keys is inserted, any search for a different key that hashes to this same index will take $\Theta(n)$ time.

To cause this worst-case behaviour in practice, a malicious user would study the

hash function used and determine such a set of colliding keys. You might be tempted, then, to keep your hash function a secret so that no one can figure out these collisions. But a reliance on such an approach only goes so far as how able you (and your colleagues) are in keeping this secret.

But let us take from this the insight that a malicious user needs to know which hash function is being used in order to purposefully generate large numbers of collisions. Rather than keep the hash function a secret, we simply use a set of hash functions, and allow our algorithm to pick randomly which one to actually use, without making its choice public. To "normal" users, the hash table behaves exactly as expected, regardless of which hash function is used. In other words, the interface of the hash table has not changed. But even if a malicious user knows the whole set of hash functions, she does not know which one is actually being used by the hash table, and in principle cannot determine how to generate collisions.

Of course, if we picked a poor set of hash functions, it might be possible that each one has a huge number of collisions, or that the same keys collide for all or most of the functions. So we certainly need some restriction on the hash functions that can go into this set; we make this idea precise in the following definition.

**Definition 5.1** (Universal hash family)**.** Let $U$ be a set of keys, and $m \in \mathbb{Z}^+$ be the size of the hash table. Let $\mathcal{H}$ be a set of hash functions, where each $h \in \mathcal{H}$ is a function $h : U \to \{0, \ldots, m-1\}$. We call $\mathcal{H}$ a **universal hash family** if and only if for all pairs of distinct keys $k_1, k_2 \in U$,

$$\Pr_{h \in \mathcal{H}}[h(k_1) = h(k_2)] \leq \frac{1}{m}.$$

Note that the probability space is now the choice of hash functions, and *not* choice of keys. This inequality must be true for *every* pair of keys!

Our algorithm is now quite simple. When a new hash table is created, it picks a hash function uniformly at random from a universal hash family, and uses this to do its hashing. This type of hashing is called **universal hashing**, and is a randomized implementation of hashing commonly used in practice.

### *Analysis of universal hashing*

Consider a universal hashing scheme that uses closed addressing (chaining). We will consider the *worst-case expected running time* of an unsuccessful search for a key $k$ in a hash table storing $n$ keys. Note that this analysis is related to, but distinct from, the analysis we did in the previous chapter. Both are probabilistic, of course, but their probability spaces are different (choosing keys vs. choosing hash functions), and the probability assumptions we are allowed to make are also different, albeit quite similar.

Let $h \in \mathcal{H}$ be the hash function used by the hash table (chosen uniformly at random). We want to consider how many of the $n$ keys already in the hash table were inserted into the index $h(k)$, since these keys are the ones that must be examined in a search for $k$.

For each inserted key $k_i$, let $X_i$ be the indicator random variable that is 1 if $h(k_i) = h(k)$, and 0 otherwise. We want to compute the expected sum of these

random variables, which we can do nicely using the techniques that we've seen before. Again, keep in mind that expected values and probabilities are over *choice of hash function $h$*, not over choice of keys.

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] &= \sum_{i=1}^{n} \mathbb{E}[X_i] \\
&= \sum_{i=1}^{n} \Pr[h(k_i) = h(k)] \\
&\leq \sum_{i=1}^{n} \frac{1}{m} && \text{(universal hash family)} \\
&= \frac{n}{m}
\end{aligned}$$

Note that we have made *no* assumptions about the keys in the hash table, nor about $k$ itself: the only property we used was the universal hash family property, which gave us the upper bound on the probability that the two hash values $h(k_i)$ and $h(k)$ were equal. Because this analysis is true for all possible inputs, we conclude that the worst-case expected running time of an unsuccessful search with universal hashing is $\mathcal{O}(1+\alpha)$, where the 1 comes from computing the hash value of $k$, as before. A similar analysis reaches the same asymptotic bound for a successful search.

Why didn't we immediately conclude $\Theta(1+\alpha)$?

### *Constructing a universal hash family*

Let us round off our two-chapter discussion of hashing by looking at one particular construction of a universal hash family.

We assume that $U = \{0, 1, \ldots, 2^w - 1\}$ is the set of natural numbers that can be stored in a machine word (so typically $w = 32$ or $w = 64$), and that $m = 2^M$ is a power of two. For every odd number $a$, $0 < a < 2^w$, and number (not necessarily odd) $b$, $0 \leq b < 2^{w-M}$, we define the following hash function:

$$h_{a,b}(k) = (ak + b \mod 2^w) \;//\; 2^{w-M}.$$

Note that these functions are very easy to implement; the first computation $ak + b \mod 2^w$ is simply unsigned integer arithmetic, and the second operation is a bit shift, taking the $M$ most significant bits.

While we won't go into the proof here, it turns out that the set of functions $\{h_{a,b}\}$, for the defined ranges on $a$ and $b$, is a universal hash family.

---

## Exercise Break!

5.1 Prove that the number of hash functions in a universal hash family must be $\geq m$.

5.2 Consider the hash function

$$h_{1,0}(k) = (k \mod 2^w) \ // \ 2^{w-M}.$$

Find a set of inputs of size $2^{w-M}$ that all have the same hash value.

This question underlines the fact that an individual hash function in a universal family still has a lot of collisions.

# 6 Graphs

In this chapter, we will study the graph abstract data type, which is one of the most versatile and fundamental in computer science. You have already studied trees, which can be used to model hierarchical data like members of a family or a classification of living things. Graphs are a generalization of trees that can represent arbitrary (binary) relationships between various objects. Some common examples of graphs in practice are modelling geographic and spatial relationships; activity between members of a social network; requirements and dependencies in a complex system like the development of a space shuttle. By the end of this chapter, you will be able to define the graph ADT and compare different implementations of this data type; and perform and analyse algorithms which traverse graphs.

There are generalizations of graphs that model relationships between more than two objects at a time, but we won't go into that here.

If you go on to take CSC373, you will build on this knowledge to write and analyse more complex graph algorithms.

## *Fundamental graph definitions*

As we hinted at in the introduction, a graph is an abstraction of the concept of a set of objects and the relationships between them. In the simplest case, we define a **vertex** (also called **node**) $v$ to be a single object, and an **edge** to be a tuple $e = (v_1, v_2)$.

This is deliberately abstract to be able to account for a variety of contexts. For example, in a social network, you can think of vertices as people, and edges as "friendships" or "connections."

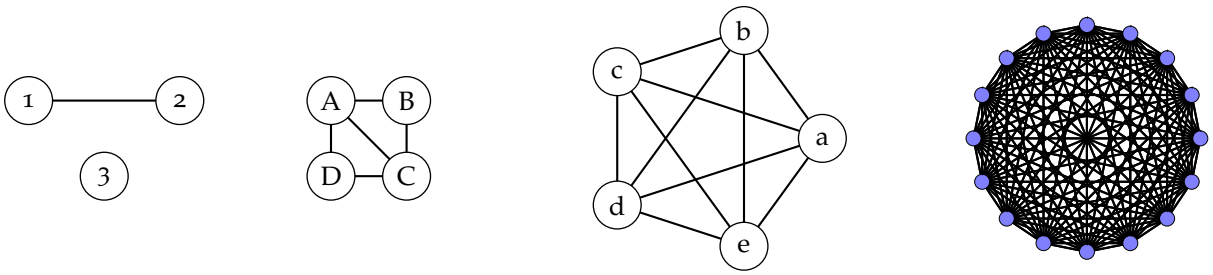
**Definition 6.1** (graph, undirected/directed)**.** A **graph** is a tuple of two sets $G = (V, E)$, where $V$ is a set of vertices, and $E$ is a set of edges on those vertices.

An **undirected graph** is a graph where order does not matter in the edge tuples: $(v_1, v_2)$ is equal to $(v_2, v_1)$. A **directed graph** is a graph where the tuple order does matter; $(v_1, v_2)$ and $(v_2, v_1)$ represent different edges. In this chapter, if neither type of graph is specified, our default will be an *undirected graph*.

We think of an edge in an undirected graph as being *between* two vertices, and an edge in a directed graph as being *from* a source vertex *to* a target vertex.

There are two obvious measures of a graph's size: the number of vertices and the number of edges. When analysing the running time of graph algorithms later in this chapter, it will be important to keep in mind how we are defining "size."

Before moving on, let's see some pictures to illustrate some simple graphs. In each diagram, the vertices are the circles, and the edges are the lines connecting pairs of circles. Vertices can be labelled or unlabelled, depending on the graph context.




*Graph theory* is a fascinating branch of mathematics that studies properties of graphs such as these, and ones which are far more complex.

It is clear that there can be a large variation in the characteristics of graphs: the number of edges, how many vertices are connected to each other, how many edges each vertex has, etc. For our purposes, we will need only a few more definitions to describe some simple graph properties.

**Definition 6.2** (adjacent, neighbour)**.** Two vertices are **adjacent** if there is an edge between them. In this case, we also say that the two vertices are **neighbours**.

**Definition 6.3** (path, length, distance)**.** A **path** between vertices $u$ and $w$ is a sequence of edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, where $u = v_0$ and $w = v_k$, and all the $v_i$ are distinct.This follows the English meaning of the term "path," where you start at vertex $u$, take an edge to get to a new vertex $v_1$, take another edge to get to another new vertex $v_2$, and keep following edges until arriving at $v_k$.

The **length** of the path is the number of edges in the path. For example, two adjacent vertices are connected by a path of length one, and the previous notation describes a path of length $k$. The **distance** between two vertices is the length of the shortest path between the two vertices. The distance between a vertex and itself is always 0.

Paths are fundamental to graphs because they allow us to take basic relationships (edges) and derive more complex ones. Suppose, for example, we are representing all the people of the world, with edges representing "A and B personally know each other." The famous example from social networks is the idea of *Six Degrees of Separation*, which says that on this graph, any two people have a path of length at most 6 between them. Given an arbitrary graph, one might wonder what the maximum path length is between any two vertices – or even whether there is a path at all!

**Definition 6.4** (connected)**.** We say that a graph is **connected** if for every pair of vertices in the graph, there is a path between them. Otherwise, we say the graph is **unconnected**.

## *Implementing graphs*

So far, we have discussed graphs as a mathematical abstraction. Our next step is to decide how to represent graphs in a program. To be a bit more concrete, we define a Graph ADT with two simple operations: obtaining a list of all vertices in the graph, and checking whether two given vertices are adjacent.

### Graph ADT

- VERTICES($G$): Return a list of all the vertices of $G$.
- CHECKADJACENT($G, u, v$): Return whether vertices $u$ and $v$ are adjacent in $G$.

In this section, we'll consider two implementations of this ADT: adjacency lists, which generalize our reference-based tree implementations, and an array-based implementation known as an adjacency matrix.

### *Adjacency lists*

The first approach is one that should be familiar, because it is essentially what we do to implement trees. We use a vertex data type that has a label attribute to identify the vertex, and a list of references to its neighbours. This latter attribute is known as an *adjacency list*, from which this implementation gets its name. Since the vertices keep track of the edge data, a second graph data type stores only a collection the vertices in the graph. To illustrate this approach, we show how one could implement a function that checks whether two vertices in a graph are adjacent. Note that this operation takes *labels*, and must use them to look up the corresponding vertices in the graph.

In our default *undirected* case, this means that two adjacent vertices each store a reference to the other in their respective adjacency lists.

A common implementation stores the vertices in an array, and labels each vertex with its array index. If we need to support arbitrary unique identifiers, any dictionary implementation (e.g., a hash table) can be used as well.

```
7  def CheckAdjacent(G, i, j):
8      # Access the vertices attribute and perform lookup by labels
9      u = G.vertices[i]
10     v = G.vertices[j]
11     for vertex in u.neighbours:
12         # Note: this assumes labels are unique
13         if vertex.label == v.label:
14             return True
15
16     return False
```

What is the running time of this algorithm? We assume that looking up the two vertices in `graph.vertices` takes constant time, so the running time is determined by the number of loop iterations (since the body of the loop always takes constant time). What is an upper bound on the number of iterations? Well, if there are $n$ vertices in the graph, then `u` can have at most $n - 1$ neighbours (if it has edges going to every other vertex). It might be tempting to say $\mathcal{O}(n)$ at this point. But remember that there are *two* measures of the size of the graph. If the graph has $m$ edges in total, then $m$ is also an upper bound on the number of neighbours that `u` can have. So the number of loop iterations is $\mathcal{O}(\min(m, n))$.

This is true if an array is used; we'll leave it as an exercise to think about other implementations of this attribute.

For a lower bound, suppose we consider a vertex $u$ that is adjacent to $\min(m, n - 2)$ vertices, and let $v$ be a vertex that is not adjacent to $u$. Then performing this algorithm will cause all $\min(m, n - 2)$ iterations of the loop to occur, resulting in a running time of $\Omega(\min(m, n))$.

### *Adjacency matrices*

The previous approach was vertex-focused: each vertex is represented as a separate entity in the program, with edges stored implicitly as the references in the adjacency lists. It is also possible to represent the edges directly in what we call an **adjacency matrix**. Suppose there are $n$ vertices in some graph, each with a label between 0 and $n-1$. We use an $n$-by-$n$ two-dimensional boolean array, where the $ij$-entry of the array is true if there is an edge between vertex $i$ and vertex $j$.

In the undirected case this matrix is symmetric across its diagonal, while in the directed case that's not guaranteed.

Using this approach, determining adjacency in a graph runs in constant time, since it is just an array lookup:

```
19 def CheckAdjacent(G, i, j):
20     return G.adjacency[i][j]
```

### *Implementation space usage*

The adjacency matrix approach seems conceptually simpler and easier to implement than the adjacency list approach. However, it does have a relatively high space cost, since it must use computer memory to store not only the edges, but also the "non-edges." This representation must store a full $|V|$-by-$|V|$ matrix, with a space cost of $\Theta(|V|^2)$.

In the adjacency list implementation, the graph itself stores an array of size $\Theta(|V|)$, and each vertex stores its label and references to its neighbours. We cannot count the number of references for an individual vertex because that can vary dramatically among vertices in an individual graph. But we do have a nice way of calculating the *total* number of references stored across all vertices.

**Definition 6.5** (degree)**.** The **degree** of a vertex is the number of its neighbours. For a vertex $v$, we will usually denote its degree by $d_v$.

**Lemma 6.1** (handshake lemma)**.** The sum of the degrees of all vertices is equal to twice the number of edges. Or,

$$\sum_{v \in V} d_v = 2|E|.$$

This lemma gets its name from the real-life example of meeting people at a party. Suppose a bunch of people are at a party always shake hands when they meet someone new. After greetings are done, if you ask each person how many new people they met and add them up, the sum is equal to twice the number of handshakes that occurred.

*Proof.* Each edge $e = (u, v)$ is counted twice in the degree sum: once for $u$, and once for $v$. ∎

So this means the total cost of storing the references in our adjacency list data structure is $\Theta(|E|)$, for a total space cost of $\Theta(|V| + |E|)$ (adding a $|V|$ term for storing the labels). Note that this is the cost for storing all the vertices, not just a single vertex. Adding up the cost for the graph data structure and all the vertices gives a total space cost of $\Theta(|V| + |E|)$.

At first glance, it may seem that $|V|^2$ and $|V| + |E|$ are incomparable. However, observe that the maximum number of possible edges is $\Theta(|V|^2)$, which arises in the case when every vertex is adjacent to every other vertex. So in asymptotic terms, $|V| + |E| = \mathcal{O}(|V|^2)$, but this bound is *not* tight: if there are very few (e.g., $\mathcal{O}(|V|)$) edges then adjacency lists will be much more space efficient than the adjacency matrix.

### *Graph traversals: breadth-first search*

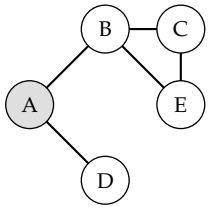
We are now ready to do our first real algorithm on graphs. Consider the central problem of *exploring* a graph by traversing its edges (rather than relying on direct access of references or array indices). For example, we might want to take a social network, and print out the names of everyone who is connected to a particular person. We would like to be able to start from a given vertex in a graph, and then visit its neighbours, the neighbours of its neighbours, etc. until we have visited every vertex that is connected to it. Moreover, we'd like to do so *efficiently*.

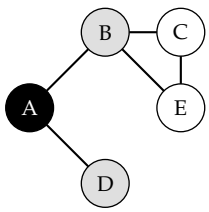
If we assume that the graph is connected, then such an exploration visits *every* vertex in the graph.

For the next two sections, we will study two approaches for performing such an exploration in a principled way. The first approach is called **breadth-first search (BFS)**. The basic idea is to take a starting vertex $s$, visit its neighbours, then visit the neighbours of its neighbours, then the neighbours of its neighbours of its neighbours, proceeding until all of the vertices have been explored. The trick is doing so without repeatedly visiting the same vertex. To formalize this in an algorithm, we need a way to keep track of which vertices have already been visited, and which vertex should be visited next.

To do this, we use a queue as an auxiliary container of pending vertices to be visited. A `enqueued` attribute is used to keep track of which vertices have been visited already. This algorithm enqueues the starting vertex, and then repeatedly dequeues a vertex and enqueues its neighbours that haven't yet been enqueued.

```
5  def BFS(graph, s):
6      queue = new empty queue
7      initialize all vertices in the graph to not enqueued
8
9      queue.enqueue(s)
10     s.enqueued = True
11     while queue is not empty:
12         v = queue.dequeue()
13         Visit(v)  # do something with v, like print out its label
14
15         for each neighbour u of v:
16             if not u.enqueued:
17                 queue.enqueue(u)
18                 u.enqueued = True
```




To illustrate this algorithm, suppose we run it on the graph to the right starting at vertex A. We'll also assume the neighbours are always accessed in alphabetical order. We mark in black when the vertices are visited (i.e., dequeued and passed to the VISIT function).

1. First, A is enqueued. We show this by marking it gray in the diagram.
2. **A** is dequeued, and its neighbours B and D are stored in the queue.
3. **B** is visited next, causing its neighbours C and E to be stored in the queue. Note that A is also B's neighbour, but it has already been enqueued, and so is not added again.
4. The next vertex to be dequeued is **D** — remember that we're storing vertices in a queue, and D was added before C and E. Since D doesn't have any unvisited neighbours, nothing is added to the queue.
5. The remaining vertices are removed from the queue in the order **C** then **E**.

So the order in which these vertices are visited is **A**, **B**, **D**, **C**, **E**.

### *Correctness of BFS*

Now that we have the algorithm in place, we need to do the two standard things: show that this algorithm is correct, and analyse its running time.

But what does it mean for breadth-first search to be correct? The introductory remarks in this section give some idea, but we need to make this a little more precise. Remember that we had two goals: to visit every vertex in the graph, and to visit all the neighbours of the starting vertex, *then* all of the neighbours of the neighbours, *then* all of the neighbours of the neighbours of the neighbours, etc. We can capture this latter idea by recalling our definition of *distance* between vertices. We can then rephrase the "breadth-first" part of the search to say that we want to visit all vertices at distance 1 from the starting vertex, then the vertices at distance 2, then distance 3, etc.

**Theorem 6.2** (Correctness of BFS)**.** Let $v$ be the starting vertex chosen by the BFS algorithm. Then for every vertex $w$ connected to $v$, the following statements hold:

(1) $w$ is visited.

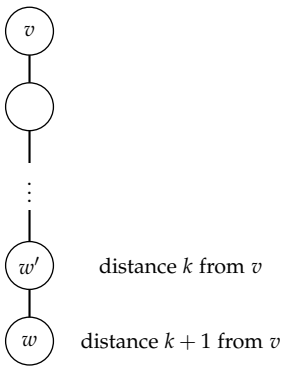
(2) Let $d$ be the distance between $v$ and $w$. Then $w$ is visited *after* every vertex at distance at most $d - 1$ from $v$.

By "visit" here we mean that the function VISIT is called on the vertex.

*Proof.* We will proceed by induction on the distance between a vertex and $v$. More formally, let $P(n)$ be the statement that (1) and (2) are true for all vertices with distance $n$ from $v$.

**Base case** ($P(0)$). The only vertex at distance 0 from $v$ is $v$ itself. Then (1) is true because $v$ is certainly visited — it is added to the queue at the beginning of the

algorithm. (2) is vacuously true because there aren't even two distinct vertices to consider.




**Inductive step**. Let $k \geq 0$, and assume $P(k)$ holds. That is, assume all vertices at distance $k$ from $v$ are visited, and visited after all vertices at distance at most $k-1$. If there are no vertices at distance $k+1$ from $v$, then $P(k+1)$ is vacuously true and we are done.

Otherwise, let $w$ be a vertex that is at distance $k+1$ from $v$. Then by the definition of distance, $w$ must have a neighbour $w'$ at distance $k$ from $v$ ($w'$ is the vertex immediately before $w$ on the path of length $k$ between $v$ and $w$). By the induction hypothesis, $w'$ is visited. When a vertex is visited, all of its neighbours are enqueued into the queue unless they have already been enqueued. So $w$ is enqueued. Because the loop does not terminate until the queue is empty, this ensures that $w$ at some point becomes the current vertex, and hence is visited. This proves that statement (1) holds for $w$.

What about (2)? We need to show that $w$ is visited after every vertex at distance at most $k$ from $v$. Let $u$ be a vertex at distance $d$ from $v$, where $d \leq k$. Note that $d = 0$ or $d \geq 1$. We'll only do the $d \geq 1$ case here, and leave the other case as an exercise.

By the induction hypothesis, we know that every vertex at distance $< d$ is visited before $u$. In particular, $u$ has a neighbour $u'$ at distance $d-1$ from $v$ that is visited before $u$. Also, $w$ must have a neighbour $w'$ at distance $k$ from $v$. Now, $u'$ is visited before $w'$, since $w'$ is at distance $k$ from $v$, and $u'$ is at distance $d-1 \leq k-1$ from $v$. Then the *neighbours* of $u'$, which include $u$, must be enqueued before the neighbours of $w'$, which include $w$. So $u$ is added to the queue before $w$, and hence is dequeued and visited before $w$. ■

This claim uses the induction hypothesis as well.

If you read through the proof carefully, it is only in the last sentence that we use the fact that we're storing the vertices in a queue and not some other collection data type!

### *Analysis of BFS*

Now let us analyse the running time of this algorithm. The creation of the new queue takes constant time, but the initialization of all vertices to "not enqueued" takes $\Theta(|V|)$ time.

The analysis of the loops is a little tricky because it is not immediately obvious how many times each loop runs. Let us start with the outer loop, which only terminates when `queue` is empty. We will do this a bit formally to illustrate the extremely powerful idea that determining the runtime of an algorithm often involves understanding the subtleties of what the algorithm really does.

**Proposition 6.3.** No vertex is added to the queue more than once.

*Proof.* Every time a vertex is added to the queue, it is always marked as 'enqueued.' Only vertices that are not enqueued are ever added to the queue — the starting vertex is initially unvisited, and there is an explicit check inside the inner loop before enqueuing new vertices. So every vertex is added to the queue at most once. ■

**Proposition 6.4.** The outer loop runs at most $|V|$ times.

*Proof.* This follows immediately from the previous proposition, and the fact that at each iteration, only one item is dequeued. ■

Unfortunately, having an upper bound on the number of loop iterations does not immediately give us an upper bound on the program runtime, even if we assume that VISIT runs in constant time. The running time of each loop iteration depends on the number of times the inner loop runs, which in turn depends on the current vertex.

To tackle the inner loop, you might notice that this is essentially the same as the space cost analysis for adjacency lists — the inner loop iterates once per neighbour of the current vertex. Because we know that `v` takes on the value of each vertex at most once, the total number of iterations of the inner loop across *all* vertices is bounded above by the total number of "neighbours" for each vertex. By the Handshake Lemma, this is exactly $2 \cdot |E|$.

Putting this together yields an upper bound on the worst-case running time of $\mathcal{O}(|V| + |E|)$. Is this bound tight? You may think it's "obvious," but keep in mind that actually the lower bound doesn't follow from the arguments we've made in this section. The initialization of the vertex `enqueued` attributes certainly takes $\Omega(|V|)$ time, but how do we get a better lower bound involving $|E|$ using the loops? After all, the first proposition says that each vertex is added *at most* once. It is conceivable that no vertices other than the start are added at all, meaning the loop would only run just once.

This is where our careful analysis of the *correctness* of the algorithm saves us. It tells us that every vertex that is connected to the starting vertex is added to the queue and visited. For a lower bound on the worst-case running time, we need a family of inputs (one for each size) that run in time $\Omega(|V| + |E|)$. Note that the size measure here includes *both* the number of vertices and the number of edges. So what kind of input should we pick? Well, consider a graph $G$ in which all edges are part of the same connected component of the graph, and then doing a breadth-first search starting at one of the vertices in this component.

By "connected component" we mean a group of vertices that are all connected to each other.

Then by Theorem 6.2, we know that all vertices in this component are visited, and the inner loop runs a total of $2 \cdot |E|$ times (since all edges are between vertices in this component). This, combined with the $\Omega(|V|)$ time for initialization, results in an $\Omega(|V| + |E|)$ running time for this family, and hence a lower bound on the worst-case running time of BFS.

---

## Exercise Break!

6.1 Build on our BFS algorithm to visit every vertex in a graph, even if the graph is not connected. *Hint*: you'll need to use a breadth-first search more than once.

6.2 Our analysis of BFS assumed that VISIT runs in constant time. Let's try to make this a bit more general: suppose that VISIT runs in time $\Theta(f(|V|))$, where $f : \mathbb{N} \to \mathbb{R}^+$ is some cost function. What is the running time of BFS in this case?

---

### *Graph traversals: depth-first search*

We have just finished studying breadth-first search, a graph traversal algorithm that prioritizes visiting vertices that are *close* to the starting vertex. A dual approach is called **depth-first search (DFS)**, which prioritizes visiting as many vertices as possible before backtracking to "choose a different neighbour." We give a recursive implementation of DFS below. The key idea is that a recursive `DFS_helper` call on a vertex $v$ fully explores the part of the graph reachable from $v$ without revisiting any previously-visited vertices.

Because recursive functions make great use of the function call stack, this implementation is a natural dual of our queue-based BFS algorithm.

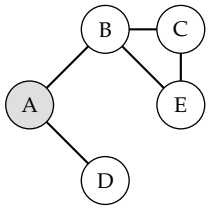
```
6  def DFS(graph, s):
7      initialize all vertices in the graph to not started or finished
8      DFS_helper(graph, s)
9
10 def DFS_helper(graph, v):
11     v.started = True
12     Visit(v)  # do something with v, like print out its label
13
14     for each neighbour u of v:
15         if not u.started:
16             DFS_helper(graph, u)
17
18     v.finished = True
```

The other major change that we make is that now we have *two* attributes, `started` and `finished`, tracking the status of each vertex throughout the search. You may notice that the `finished` attribute is not used by the algorithm itself; however, it does play a helpful role in *analysing* the behaviour of DFS. At any point during the run of DFS, each vertex has one of three possible states:

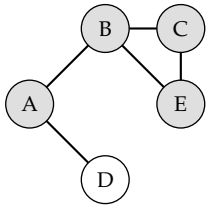
The `finished` attribute will also be helpful in one of the applications of depth-first search discussed later in this chapter.

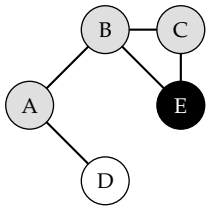
- Not started: has not been visited by the search.
- Started but not finished: has been visited by the search, but some vertices reachable from this vertex without revisiting other started vertices still need to be visited.
- Started and finished: it has been visited by the search, and all vertices reachable from this vertex without revisiting other started vertices have been visited.

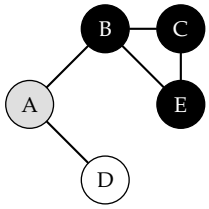
Let us trace through this algorithm on the same graph from the previous section, again starting at vertex A. In the diagram, white vertices have not been started, gray vertices have been started but not finished, and black vertices have been started and finished.




1. First, **A** is visited. It is marked as started, but is not yet finished.
2. Then **B** is visited (A's first neighbour). It is marked as started.
3. Then **C** is visited (B's first neighbour). It is marked as started. Note that A is *not* visited again, since it has already been marked as started.
4. Then **E** is visited (C's first neighbour). It is marked as started.
5. Since **E** has no neighbours that haven't been started, it is marked as finished, and the recursive call terminates.
6. Similarly, C and B are marked as finished, and their respective recursive calls terminate, *in that order*.
7. Control returns to the original search with start A. **D** is visited (A's second neighbour). It is marked as started.
8. Since D does not have any unvisited neighbours, it is marked as finished. And finally, A is marked as finished as well.



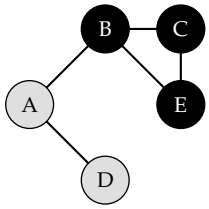




The order in which DFS visits the vertices is **A**, **B**, **C**, **E**, **D**.

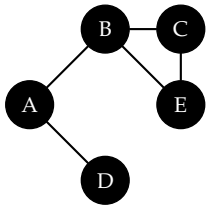
*A note about DFS correctness and runtime*

Using a similar analysis as breadth-first search, we are able to prove the following "weak" notion of correctness for depth-first search.




**Theorem 6.5** (Weak correctness of DFS)**.** Let $v$ be the starting vertex given as input to the DFS algorithm. Then for every vertex $w$ that is connected to $v$, the following statement holds:

(1) $w$ is visited.




It is harder to capture the intuitive notion of "depth-first" because there isn't an obvious notion of distance to measure. We can, however, formalize the properties of the attributes `started` and `finished` we asserted (without proof!) in the previous section.

**Theorem 6.6** (Properties of DFS)**.** Suppose `DFS_helper` is called on a vertex $v$, and let $w$ be any vertex in the graph that satisfies the following two properties at the beginning of the function call:

1. $w$ is not started.
2. There is a path between $v$ and $w$ consisting of only vertices that have *not* been started.

Then $w$ is *both started and finished* by the time `DFS_helper(v)` returns.

Finally, using a very similar analysis as BFS, we can obtain the following tight bound on the worst-case running time of DFS.

**Theorem 6.7** (DFS runtime)**.** The worst-case running time of depth-first search is $\Theta(|V| + |E|)$.

We will leave the proofs of these last two theorems as exercises.

---

### Exercise Break!

6.1 Prove Theorem 6.6.

6.2 Prove Theorem 6.7.

---

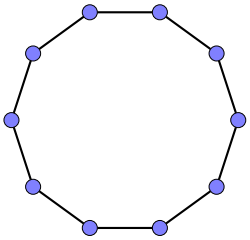
## *Applications of graph traversals*

In this section, we'll look at two applications of breadth-first and depth-first search that have wide-ranging uses in computer science: detecting cycles in a graph, and determining the shortest path between two vertices.

### *Detecting cycles (undirected graphs)*

**Definition 6.6** (cycle)**.** A **cycle** in a graph is a sequence of distinct edges $(v_0, v_1)$, $(v_1, v_2)$, $\ldots$, $(v_{n-1}, v_n)$, $(v_n, v_0)$ that start and end at the same vertex.

That is, a cycle is a path that starts and ends at the same vertex.



Cycles encode a natural "redundancy" in a graph: it is possible to remove any edge in a cycle without disconnecting its vertices (just take the long way around the cycle), and so it is often useful to know whether a graph has a cycle, or whether it does not. It turns out that determining whether a graph has a cycle or not can be done through a simple modification of either breadth-first or depth-first search, though we will only show the DFS version here.

Recall that depth-first search sets two attributes `started` and `finished` for each vertex, which are used to track the status of the vertex in the search. Intuitively, we can use these to detect cycles by checking whether we have reached an already-visited node when performing a search. Below, we give a modified version of DFS that reports when it detects a cycle, with two main changes to the implementation:

- Each vertex other than the starting vertex stores its *parent* vertex, which is the neighbour that caused the vertex to be visited
- The inner loop contains a check for `started` neighbours that are not the parent of the current vertex; as we'll show later, this is a valid check for a cycle.

The caveat for our (quite simple) implementation is that it only works when the graph is connected; we leave it as an exercise to decide how to modify it to always report a cycle, even when the graph is not connected.

```
5  def DetectCycle(graph):
6      initialize all vertices in the graph to not started or finished
7      s = pick starting vertex
8      DFS_helper(graph, s)
9
10 def DFS_helper(graph, v):
11     v.started = True
12
13     for each neighbour u of v:
14         if not u.started:
15             u.parent = v  # NEW: store parent attribute
16             DFS_helper(graph, u)
17         else:
18             # NEW: check for cycle and report
19             if u != v.parent:
20                 report that there is a cycle
21
22     v.finished = True
```

Before we prove the correctness of this algorithm, observe that it is an extremely minor modification of the standard DFS algorithm, and clearly runs in $\Theta(|V| + |E|)$ time. Indeed, one reason cycle detection is a classic application of DFS and BFS is that the code changes necessary to solve this problem add only a constant factor running time overhead to the basic search algorithms. You could say that the search algorithm does all the hard work, and cycle detection is thrown in for free.

Of course, one should at least be a little skeptical that minor modifications can be made to an algorithm to solve a different problem. Let us now prove that this algorithm is indeed correct. You may find it helpful to review Theorem 6.6 before continuing.

**Theorem 6.8** (Cycle Detection with DFS correctness)**.** The modified DFS algorithm reports that a cycle exists if and only if the input graph has a cycle.

*Proof.* Let us do the forward direction: assume that the algorithm reports a cycle. We want to prove that the input graph has a cycle.

Let $u$ and $v$ represent the values of the variables `u` and `v` when the cycle is reported (on line 16). We will claim that $u$ and $v$ are part of a cycle. Note that $u$ and $v$ are neighbours, so all we need to do is show that there is a path from $u$ to $v$ that doesn't use this edge.

We can "close" the path with the $(u, v)$ edge to form the cycle.

First, since $u$ is `started`, it must have been visited before $v$. Let $s$ be the starting vertex of the DFS. Because both $u$ and $v$ have been visited, they are both

connected to $s$.

Moreover, because $v$ is the current vertex, this means that $u$ cannot be `finished`, since all of the neighbours of $u$ must finish before $u$. So then $u$ must be on the path between $v$ and $s$. Moreover, because it is not the parent of $v$, the segment of this path between $u$ and $v$ contains at least one other vertex, and does not contain the edge $(u, v)$.

Now the backwards direction: assume the input graph has a cycle. We want to show that a cycle is reported by this algorithm. Let $u$ be the first vertex in the cycle that is visited by the DFS, and let $w$ be the next vertex in the cycle that is visited.

Though there may be other vertices visited by the search between $u$ and $w$, we will ignore those, looking only at the vertices on the cycle. When $w$ is visited, it and $u$ are the only vertices on the cycle that have been visited. In particular, this means that $u$ has at least one neighbour in the cycle that hasn't been visited. Let $v$ be a neighbour of $u$ that has not yet been visited. Then because there is a path between $w$ and $v$ consisting of vertices that are not started, $v$ must be visited before the `DFS_helper` call ends.

This is true even if $w$ and $u$ are neighbours, since $u$ has two neighbours on the cycle.

Now consider what happens when `DFS_helper` is called on $v$. We know that $u$ is a neighbour of $v$, that it is `started`, and that it is not the parent of $v$ (since the `DFS_helper` call on $w$ hasn't ended yet). Then the conditions are all met for the algorithm to report the cycle. ■

### *Detecting cycles (directed graphs)*

Before moving onto our next example, let us take a brief detour into the world of directed graphs. Recall that a directed graph is one where each edge has a direction: the tuple $(u, v)$ represents an edge from $u$ to $v$, and is treated as different from the edge $(v, u)$ from $v$ to $u$.

Cycle detection in directed graphs has several applications; here is just one example. Consider a directed graph where each vertex is a course offered by the University of Toronto, and an edge $(u, v)$ represents the relationship "$u$ is a prerequisite of $v$." Then a cycle of three directed edges with vertex sequence $v_1, v_2, v_3, v_1$ would represent the relationships "$v_1$ is a prerequisite of $v_2$, $v_2$ is a prerequisite of $v_3$, and $v_3$ is a prerequisite of $v_1$." But this is nonsensical: none of the three courses $v_1$, $v_2$, or $v_3$ could ever be taken. Detecting cycles in a prerequisite graph is a vital step to ensuring the correctness of this graph.

It turns out that essentially the same algorithm works in both the undirected and directed graph case, with one minor modification in the check for a cycle. Before we write the algorithm, we note the following changes in terminology in the directed case:

- A *neighbour* of $v$ is another vertex $u$ such that there is an edge *from $v$ to $u$*.
- A *directed cycle* can have length 2: if there are two vertices $u$ and $v$ such that there is an edge from $u$ to $v$ and one from $v$ to $u$, these two vertices form a

cycle.

Recall that this was a special case we had to check for with the "parent" attribute in the undirected case.

```
def DetectCycleDirected(graph):
    initialize all vertices in the graph to not started or finished
    s = pick starting vertex
    DFS_helper(graph, s)

def DFS_helper(graph, v):
    v.started = True

    for each neighbour u of v:
        if not u.started:
            DFS_helper(graph, u)
        else:
            # NEW: check for directed cycle and report
            if not u.finished:
                report that there is a cycle

    v.finished = True
```

We won't prove the correctness of this algorithm here, as the proof is quite similar to that of the undirected case. It truly is a testament to the flexibility of DFS that such minor modifications can have such interesting consequences.

### *Shortest Path*

Let us now turn our attention to a second application of these searches. Recall that BFS visits vertices in order of their distance from the starting vertex. From this information alone, however, we cannot determine precisely how far away a given vertex is from the start. However, if we adapt the "storing parent" idea to the breadth-first search algorithm, it turns out that we can not only get the distance between the starting vertex and an arbitrary vertex visited by the search, but also the *most direct route* as well.

More concretely, suppose we are given a graph and two vertices $v$ and $u$, and want to find a shortest path between them. Our algorithm is quite straightforward:

Note that there could be multiple paths of the same minimum length between $u$ and $v$, and we're only looking for one of them.

1. Perform a BFS starting at $v$, storing the parent of each vertex. Here, *parent* has the same meaning as DFS: the parent of a vertex $w$ is the vertex which, when visited by BFS, causes $w$ to be added to the queue.
2. Return the path formed by starting at $u$, and following the parent vertex references until $v$ is reached.

What can we conclude if $u$'s parent attribute is unset after the BFS?

Let us formally prove that this algorithm is correct.

**Theorem 6.9.** Let $v$ be the starting vertex of a BFS on a given graph, and $u$ be any vertex that is connected to $v$, and $d$ be the distance between $u$ and $v$.Let $u_0 = u$, $u_1$ be the parent of $u_0$, and in general for all $1 \leq i \leq d$, $u_i$ the parent of $u_{i-1}$. Then we claim that $u_d = v$, i.e., the path of vertices $u_0, u_1, \ldots, u_d$ of length $d$ does in fact reach $v$, making this path one of the shortest possible length.

*Proof.* We prove this by induction on $d$, the distance from $u$ to $v$.

**Base case**: $d = 0$. In this case, $u$ must equal $v$. Then, $u_0 = u = v$, and so the theorem holds in this case. Note that the "shortest path" consists of just the single vertex $v$, and has length 0.

**Inductive step**: suppose the statement holds for some distance $d \geq 0$; that is, for every vertex at distance $d$ from $v$, the path obtained by starting at that vertex and following the parent links $d$ times correctly ends at $v$. Let $u$ be a vertex at distance $d + 1$ from $v$. We want to prove that the same is true for $u$ (following the parent $d + 1$ times joins $u$ to $v$).

Because $u$ is connected to $v$, it must be visited by the BFS starting at $v$, and so has a parent. Let $u'$ be the parent of $u$. We claim that $u'$ must have distance $d$ from $v$.

First, $u'$ cannot be at distance $< d$ from $v$, because otherwise $u$ would be at distance $< d + 1$ from $v$. But also $u'$ cannot be at distance $\geq d + 1$ from $v$, because then it wouldn't be the parent of $u$. Why not? *u must* have a neighbour of distance $d$ from $v$, and by the BFS correctness theorem, this neighbour would be visited before $u'$ if $u'$ were at a greater distance than $d$ from $v$. But the parent of $u$ is the *first* of its neighbours to be visited by BFS, so $u'$ must be visited before any other of $u$'s neighbours.

So, $u'$ is at distance $d$ from $v$. By the induction hypothesis, the sequence of parents $u_0 = u'$, $u_1$, ..., $u_d$ is a path of length $d$ that ends at $v$. But then the sequence of parents starting at $u$ of length $d + 1$ is $u, u_0, u_1, \ldots, u_d$, which also ends at $v$. ■

So with this slight modification to breadth-first search, we are able to compute the shortest path between any two vertices in a graph. This has many applications in computer science, including computing efficient routes for transmitting physical goods and electronic data in a network and the analysis of dependency graphs of large-scale systems.

## *Weighted graphs*

The story of shortest paths does not end here. Often in real-world applications of graphs, not all edges — relationships — are made equal. For example, suppose we have a map of cities and roads between them. When trying to find the shortest path between two cities, we do not care about the *number* of roads that must be taken, but rather the *total length* across all the roads. But so far, our graphs can only represent the fact that two cities are connected by a road, and

not how long each road is. To solve this more general problem, we need to augment our representation of graphs to add such "metadata" to each edge in a graph.

**Definition 6.7** (weighted graph, edge weight)**.** A **weighted graph** is a graph $G = (V, E)$, where each edge now is a triple $(v_1, v_2, w)$, where $v_1, v_2 \in V$ are vertices as before, and $w \in \mathbb{R}$ is a real number called the **weight** of the edge. Given an edge $e$, we will use the notation $w_e$ to refer to its weight.

We will refer to our previous graphs with no weights as *unweighted graphs*, and that is our default (just like undirected graphs are our default).

Fun fact: you can think of an unweighted graphs as a weighted graph where each edge has weight 1.

The *weighted shortest path problem* for a weighted graph does not minimize the number of edges in a path, but instead minimizes the sum of the weights along the edges of a path. This is a more complex situation that cannot be solved by breadth-first search, and beyond the scope of this course. But fear not: you'll study algorithms for solving this problem in CSC373.

## *Minimum spanning trees*

Now that we have introduced weighted graphs, we will study one more fundamental weighted graph problem in this chapter. Suppose we have a connected weighted graph, and want to remove some edges from the graph but keep the graph connected. You can think of this as taking a set of roads connecting cities, and removing some the roads while still making sure that it is possible to get from any city to any other city. Essentially, we want to remove "redundant" edges, the ones whose removal does not disconnect the graph.

How do we know if such redundant edges exist? It turns out (though we will not prove it here) that if a connected graph has a cycle, then removing *any* edge from that cycle keeps the graph connected. Moreover, if the graph has no cycles, then it does not have any redundant edges. This property – not having any cycles – is important enough to warrant a definition.

We won't prove this here, but please take a course on graph theory if you interested in exploring more properties like this in greater detail.

**Definition 6.8** (tree)**.** A **tree** is a connected graph that has no cycles.

Warning: one difference between this definition of *tree* and the trees you are used to seeing as data structures is that the latter are rooted, meaning we give one vertex special status as the root of the tree. This is not the case for trees in general.

So you can view our goal as taking a connected graph, and identifying a subset of the edges to keep, which results in a tree on the vertices of the original graph.

It turns out that the *unweighted* case is not very interesting, because all trees obey a simple relationship between their number of edges and vertices: every tree with $n$ vertices has exactly $n - 1$ edges. So finding such a tree can be done quite easily by repeatedly finding cycles and choosing to keep all but one of the edges.

However, the weighted case is more challenging, and is the problem of study in this section.

**Definition 6.9** (minimum spanning tree)**.** The **Minimum Spanning Tree Problem** takes as input a connected weighted graph $G = (V, E)$, and outputs a set of edges $E'$ such that the graph $(V, E')$ is a tree, and which minimizes the sum

of the weights $\sum_{e \in E'} w_e$. We call the resulting tree a **minimum spanning tree (MST) of** $G$.

As with shortest paths, there might be more than one minimum spanning tree of $G$. We just want to find one.

*Prim's algorithm*

The first algorithm we will study for solving this problem is known as **Prim's algorithm**. The basic idea of this algorithm is quite straightforward: pick a starting vertex for the tree, and then repeatedly add new vertices to the tree, each time selecting an edge with minimum weight that connects the current tree to a new vertex.

```
def PrimMST(G):
    v = pick starting vertex from G.vertices
    TV = {v}  # MST vertices
    TE = {}   # MST edges

    while TV != G.vertices:
        extensions = { (u, v) in G.edges where exactly one of u, v are in TV }
        e = edge in extensions having minimum weight
        TV = TV + endpoints of e
        TE = TE + {e}

    return TE
```

This algorithm is quite remarkable for two reasons. The first is that it works regardless of which starting vertex is chosen; in other words, there is no such thing as the "best" starting vertex for this algorithm. The second is that it works without ever backtracking and discarding an already-selected edge to choose a different one. Let us spend some time studying the algorithm to prove that it is indeed correct.

This algorithm doesn't "make mistakes" when choosing edges.

**Theorem 6.10** (Correctness of Prim's algorithm)**.** Let $G = (V, E)$ be a connected, weighted graph. Then Prim's algorithm returns a set of edges $E'$ such that $(V, E')$ is a minimum spanning tree for $G$.

*Proof.* The proof of correctness of Prim's algorithm is an interesting one, because it relies on two loop invariants, one of which is straightforward, but the other of which is very neat and unexpected if you haven't seen anything like it before. Here are our two invariants (remember that these are true at the beginning and end of **every** iteration of the loop):

(1) The graph (TV, TE) is a tree.
(2) The tree (TV, TE) can be extended by only *adding* some edges and vertices to a minimum spanning tree of $G$.

Remember: trees are connected and have no cycles.

This is like saying that TE is a subset of a possible solution to this problem.

Before we prove that these invariants are correct, let's see why they imply the correctness of Prim's algorithm. When the loop terminates, both of these invariants hold, so the graph (TV, TE) is a tree and can be extended to an MST for the input graph. However, when the loop terminates, its condition is false, and so TV contains all the vertices in the graph. This means that (TV, TE) must *be* an MST for the input graph, since no more vertices/edges can be added to extend it into one.

We will omit the proof of the first invariant, which is rather straightforward. The second one is significantly more interesting, and harder to reason about. Fix a loop iteration. Let $(V_1, E_1)$ be the tree at the beginning of the loop body, and $(V_2, E_2)$ be the tree at the end of the loop body. We assume that $(V_1, E_1)$ can be

High level: every time a vertex is added, an edge connecting that vertex to the current tree is also added. Adding this edge doesn't create a cycle.

extended to an MST $(V, E')$ for the input, where $E_1 \subseteq E'$. Our goal is to show that $(V_2, E_2)$ can still be extended to an MST for the input (but *not necessarily the same one*).

This assumption is precisely the loop invariant being true at the beginning of the loop body.

What the loop body does is add one vertex $v$ to $V_1$ and one edge $e = (v, u, w_e)$ to $E_1$, where $u$ is already in $V_1$, and $w_e$ is the weight of this edge. So we have $V_2 = V_1 \cup \{v\}$ and $E_2 = E_1 \cup \{e\}$.

**Case 1**: $e \in E'$. In this case, $E_2 \subseteq E'$, so the tree $(V_2, E_2)$ can also be extended to the same MST as $(V_1, E_1)$, so the statement holds.

**Case 2**: $e \notin E'$. This is tougher because this means $E_2$ cannot simply be extended to $E'$; we need to find a different minimum spanning tree $(V, E'')$ that contains $E_2$.

Let us study the minimum spanning tree $(V, E')$ more carefully. Consider the partition of vertices $V_1, V \backslash V_1$, in this MST. Let $u$ and $v$ be the endpoints of $e$; then they must be connected in the MST, and so there is a path between them. Let $e' \in E'$ be an edge on this path that connects a vertex from $V_1$ to a vertex from $V \backslash V_1$. Since $e \notin E'$, this means that $e' \neq e$.

Now consider what happens in the loop body. Since $e'$ connects a vertex in $V_1$ to a vertex not in $V_1$, it is put into the set `extensions`. We also know that $e$ was selected as the minimum-weight edge from `extensions`, and so $w_{e'} \geq w_e$.

Note that their weights could be equal.

Now define the edge set $E''$ to be the same as $E'$, except with $e'$ removed and $e$ added. The resulting graph $(V, E'')$ is still connected, and doesn't contain any cycles – it is a tree. Moreover, the total weight of its edges is less than or equal to that of $E'$, since $w_{e'} \geq w_e$. Then the graph $(V, E'')$ is also a minimum spanning tree of the input graph, and $E \subseteq E''$. ■

In fact, because $E'$ forms a *minimum* spanning tree, the total weight of $E''$ can't be smaller than $E'$, so their weights are equal.

The idea of replacing $e'$ by $e$ in the initial MST to form a new MST is a very nice one. Intuitively, we argued that at the loop iteration where we "should" have chosen $e'$ to go into the MST, choosing $e$ instead was still just as good, and also leads to a correct solution. It is precisely this argument, "every choice leads to a correct solution", that allows Prim's algorithm to never backtrack, undoing a choice to make a different one. This has significant efficiency implications, as we'll study in the next section.

### *Analysis of Prim's algorithm*

What is the running time of Prim's algorithm? Let $n$ be the number of vertices in the graph, and $m$ be the number of edges. We know that the outer loop always runs exactly $n$ times. Each iteration adds one new vertex to the set `TV`; the loop terminates only when `TV` contains all $n$ vertices. But what happens inside the loop body?

This is actually where the choice of data structures is important. Note that we have two sets here, a set of vertices and a set of edges. We'll first use an array to represent these sets, so looking up a particular item or adding a new item takes constant time.

How do we compute `extensions`, the edges that have one endpoint in `TV` and one endpoint not in `TV`? We can loop through all the edges, each time checking its endpoints; this takes $\Theta(m)$ time. What about computing the minimum weight edge? This is linear in the size of `extensions`, and so in the worst case is $\Theta(m)$.

Using adjacency lists, this statement is certainly true, but requires a bit of thought. We leave this as an exercise.

This leads to a total running time of $\Theta(mn)$, which is not great (recall that BFS and DFS both took time linear in the two quantities, $\Theta(n + m)$). The key inefficiency here is that the inner computations of finding "extension" edges and finding the minimum weight edge both do a lot of repeated work in different iterations of the outer loop. After all, from one iteration to the next, the extensions only change slightly, since the set `TV` only changes by one vertex.

So rather than recompute `extensions` at every iteration, we maintain a heap of edges that extends the current tree (`TV`, `TE`), where the "priority" of an edge is simply its weight. This is quite nice because the operation we want to perform on `extensions`, find the edge with the minimum weight, is well-suited to what heaps can give us.

Suppose that we pre-sort the edges in nondecreasing order. Then, when each edge is checked, it must have the minimum weight of all the remaining edges. Since we only care about getting the remaining edge with the minimum weight, this is a perfect time for a heap:

```
def PrimMST(G):
    v = pick starting vertex from G.vertices
    TV = {v}  # MST vertices
    TE = {}   # MST edges
    extensions = new heap  # treat weights as priority
    for each edge on v:
        Insert(extensions, edge)

    while TV != G.vertices:
        e = ExtractMin(extensions)
        if both endpoints of e are in TV:
            continue  # go onto the next iteration

        u = endpoint of e not in TV

        TV = TV + {u}
        TE = TE + {e}
        for each edge on u:
            if other endpoint of edge is not in TV:
                Insert(extensions, edge)

    return (TV, TE)
```

We will do an informal running time analysis of this algorithm. First, we note

that each edge is added to the heap at most once, and so the total number of calls to `Insert` is $\mathcal{O}(|E|)$. This means that the cost of all the heap operations (both `Insert` and `ExtractMax`) take $\mathcal{O}(|E| \log |E|)$ in total.

Exercise: *why* is each edge added at most once?

At each iteration, the endpoints of the extracted edge must be checked for membership in `TV`. A naïve implementation of this would search every element of `TV`, for a worst-case running time of $\Theta(|V|)$. This can be improved by two methods: store an attribute for each vertex indicating whether it is currently in `TV` or not, or use an AVL tree or hash table to store the set to support asymptotically faster lookup. We will assume that we're using the first approach, which reduces the check to a constant-time operation.

Since all the other operations take constant time, the running time is $\mathcal{O}(|E| \log |E| + |V|) = \mathcal{O}(|E| \log |E|)$, since $|E| \geq |V| - 1$ for a connected graph. It turns out that with a bit more effort, we can get this down to $O(|V| \log |V|)$ — see CLRS, section 23.2 for details.

### *Kruskal's algorithm*

As a bridge between this chapter and our next major topic of discussion, we will look at another algorithm for solving the minimum spanning tree problem. You might wonder why we need another algorithm at all: didn't we just spend a lot of time developing an algorithm that seems to work perfectly well? But of course, there are plenty of reasons to study more than one algorithm for solving a given problem, much as there is for studying more than one implementation of a given ADT: generation of new ideas or insights into problem structure; improving running time efficiency; different possible directions of generalization. In this particular case, we have a pedagogical reason for introducing this algorithm here as well, as motivation for the final abstract data type we will study in this course.

The second algorithm for solving the minimum spanning tree problem is known as **Kruskal's algorithm**. It is quite similar to Prim's algorithm, in that it incrementally builds up an MST by selecting "good" edges one at a time. Rather than build up a single connected component of vertices, however, it simply sorts edges by weight, and always picks the smallest-weight edge that doesn't create a cycle with the edges already selected.

```
5  def KruskalMST(G):
6      TE = {}
7      sort G.edges in non-decreasing order of weights
8
9      for e in G.edges:  # starting with the smallest weight edge
10         if TE + {e} does not have a cycle:
11             TE = TE + {e}
12
13     return TE
```

We will omit the proof of correctness of Kruskal's algorithm, although it is quite remarkable that this algorithm works correctly given its simplicity.

The running time of this algorithm is more interesting. The sorting of edges takes $\Theta(|E| \log |E|)$ in the worst-case and the outer loop runs at most $|E|$ times. The really interesting part of the algorithm comes from checking whether adding a new edge `e` to `TE` creates a cycle or not. A natural approach is to use what we learned earlier in this chapter and perform either a depth-first or breadth-first search starting from one of the endpoints of the chosen edge, using only the edges in `TE + {e}`. This is rather inefficient, since the $\mathcal{O}(|V| + |E|)$ running time of the search is repeated $\mathcal{O}(|E|)$ times, for a total running time of $\mathcal{O}(|E| \log |E| + |E|(|V| + |E|)) = \mathcal{O}(|E| \cdot |V| + |E|^2)$.

Well, this is at least the obvious upper bound calculation. We leave it as an exercise to prove a tight lower bound on the worst-case running time.

However, there is a much smarter way of checking whether adding a given edge creates a cycle or not, by keeping track of the connected components of the vertices defined by the edges in `TE`. At the beginning of Kruskal's algorithm, we can view each vertex as being its own component, since `TE` is empty. When the first edge is added, its two endpoints are merged into the same component. As more and more edges are added, components merge together, until eventually there is only one component, the minimum spanning tree.

How does this help us check whether adding an edge creates a cycle or not? We look at the two endpoints of the new edge: if they are in the same connected component, then adding the edge creates a cycle; otherwise, adding an edge doesn't create a cycle, and causes the two components of its endpoints to be merged.

```
def KruskalMST(G):
    TE = {}
    sort G.edges in non-decreasing order of weights
    set each vertex to be its own connected component

    for e in G.edges:  # starting with the smallest weight edge
        if the endpoints of e are not in the same connected component:
            TE = TE + {e}
            merge the connected components containing the endpoints of e

    return TE
```

The running time of the algorithm now depends on how quickly we can maintain these components while performing these three operations:

- create a new component for each vertex
- determine whether two vertices are in the same component
- merge two components together

As we will see in the following two chapters, the answer is: "we can support all three of these operations very efficiently, if we consider the operations in bulk rather than one at a time." But before we formalize the abstract data type to support these operations and study some implementations of this data type, we need to really understand what it means to "consider the operations in bulk." This is a type of runtime analysis distinct from worst-case, average-case, and even worst-case expected running times, and is the topic of the next chapter.

# 7 *Amortized Analysis*

In this chapter, we will study another form of algorithm analysis that offers a new type of refinement of worst-case analysis. To set the stage for this technique, let us return to the context of hash tables from Chapter 4. Recall that in hashing with open addressing, keys are stored directly in the table itself, with collisions being resolved by iterating through a *probe sequence* of possible indices until an unoccupied spot is found. In our consideration of this technique, we had an underlying assumption that there would be space for every new key-value pair inserted (and that you'd find such a space with a sufficiently long probe sequence). This implied that the array used to store the pairs was bigger than the number of pairs stored. In general, however, we do not know a priori how many pairs are going to be stored in the hash table, and so it is impossible to predict how large an array to use.

One can get around this problem by storing the hash table with a data structure that uses dynamically-allocated memory like a linked list or AVL tree, but then we lose the constant-time addressing feature that we get from storing items in a contiguous block of memory. In this chapter, we will study a variant of arrays called **dynamic arrays**, which we use to implement a resizable collection of items that supports constant-time addressing and "efficient" insertion of new items. We put efficient in quotation marks here because our standard measures of efficiency either won't apply or won't be impressive. Instead, we will define and justify a new type of running time analysis known as *amortized analysis* under which dynamic arrays are excellent .

## *Dynamic arrays*

The idea of a dynamic array is to keep the fundamental array property of storing items in a contiguous memory block, but allow the array to expand when the existing array is full. One might think that this expansion is easy, as we can always increase the size of an array by giving it ownership of some memory blocks after its current endpoint. But of course that memory might be already allocated for a different use, and if it is then the existing array cannot simply be expanded.

This "different use" might even be external to our program altogether!

Instead, we will do the only thing we can, and allocate a separate and bigger block of memory for the array, copying over all existing elements, and then inserting the new element. We show below one basic implementation of this ap-

proach, where `A` has three attributes: `array`, a reference to the actual array itself; `allocated`, the total number of spots allocated for `array`; and `size`, the number of filled spots in `array`. For this implementation, we choose an "expansion factor" of two, meaning that each time the array is expanded, the memory block allocated doubles in size.

```
def insert(A, x):
    # Check whether current array is full
    if A.size == A.allocated:
        new_array = new array of length (A.allocated * 2)
        copy all elements of A.array into new_array
        A.array = new_array
        A.allocated = A.allocated * 2

    # insert the new element into the first empty spot
    A.array[A.size] = x
    A.size = A.size + 1
```

*Worst-case running time of dynamic arrays*

For the purposes of analysing the running time of this data structure, we will only count the *number of array accesses* made. This actually allows us to compute an exact number for the cost, which will be useful in the following sections.

While this does cause us to ignore the cost of the other operations, all of them are constant time, and so focusing on the number of array accesses won't affect our asymptotic analysis.

For now, suppose we have a dynamic array where all $n$ allocated spots are full. We note that the final two steps involve just one array access (inserting the new item `x`), while the code executed in the `if` block has exactly $2n$ accesses: accessing each of the $n$ items in the old array and copying each item to the new array.

So we have a worst-case running time of $2n + 1 = \mathcal{O}(n)$ array accesses. This upper bound does not seem very impressive: AVL trees were able to support insertion in worst-case $\Theta(\log n)$ time, for instance. So why should we bother with dynamic arrays at all? The intuition is that this linear worst-case insertion is not a truly representative measure of the efficiency of this algorithm, since it only occurs when the array is full. Furthermore, if we assume the array length starts at 1 and doubles each time the array expands, then the array lengths are always powers of 2. This means that $2n + 1$ array accesses can only ever occur when $n$ is a power of 2; the rest of the time, insertion accesses only a single element! A bit more formally, if $T(n)$ is the number of array accesses when inserting into a dynamic array with $n$ items, then

$$T(n) = \begin{cases} 2n + 1, & \text{if } n \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

So while it is true that $T(n) = \mathcal{O}(n)$, it is *not* true that $T(n) = \Theta(n)$.

Are average-case or worst-case expected analyses any better here? Well, remember that these forms of analysis are only relevant if we treat the input data as random, or if the algorithm itself has some randomness. This algorithm certainly doesn't have any randomness in it, and there isn't a realistic input distribution (over a finite set of inputs) we can use here, as the whole point of this application is to be able to handle insertion into arrays of *any* length. So what do we do instead?

## *Amortized analysis*

Before we give a formal definition of amortized analysis, let us expand on one important detail from our informal discussion in the previous section. We observed that dynamic array insertion was only very rarely slow (when the number of items is a power of two, meaning that the array is full), with any other insertion being extremely fast. But even more is true: the only way to reach the "bad" state of inserting into a full array is by *first* performing a series of (mostly fast) insertions.

Consider inserting an item into a full array of allocated length $2^{10} = 1024$. Since 1024 is a power of two, we know that this will take 2049 array accesses. However, to even reach such a state, we must have first taken an array of length 512, expanded it to an array of length 1024, and then performed 511 insertions to fill up the expanded array. And these 511 insertions would each have taken just one array access each!

Remember that the expansion is caused when a new item is inserted, so the expanded array starts with 513 elements in it.

So we can contextualize the "inefficient" insertion into the full 1024-length array by noting that it must have been preceded by 511 extremely efficient insertions, which doesn't sound so bad.

More formally, an **amortized analysis** of a data structure computes the maximum possible *average cost per operation in a sequence of operations*, starting from some initial base state. Unlike both worst- and average-case analyses, we are not concerned here with studying one individual operation, but rather the total cost of a sequence of operations performed on a data structure, and relating this to the number of operations performed. There are two different ways of thinking about the amortized cost of a set of operations:

You can ignore the "maximum possible" for now, since for dynamic arrays we only have one possible operation — insertion — and hence one possible sequence of operations.

- The *average cost* of an operation when multiple operations are performed in a row. Note that this is a different use of "average" than average-case analysis, which considers an operation as an isolated event, but allows for consideration of multiple *inputs*.
- The *total cost* of a sequence of $M$ operations as a function of $M$. Note that a sequence of $M$ constant-time operations must have a total running time of $\Theta(M)$, so expressions that look "linear" aren't directly referencing the worst-case running time of a single operation, but rather the rate of growth of the runtime of the entire sequence.

*Aggregate method*

The first method we'll use to perform an amortized analysis is called the **aggregate method**, so named because we simply compute the total cost of a sequence of operations, and then divide by the number of operations to find the average cost per operation.

Let's make this a bit more concrete with dynamic arrays. Suppose we start with an empty dynamic array (size 0, allocated space 1), and then perform a sequence of $M$ insertion operations on it. What is the total cost of this sequence of operations? We might naively think of the worst-case expression $2n + 1$ for the insertion on an array of size $n$, but as we discussed earlier, this vastly overestimates the cost for most of the insertions. Since each insertion increases the size of the array by 1, we have a simple expression of the total cost:

We're still counting only array accesses here.

$$\sum_{k=0}^{M-1} T(k).$$

Recall that $T(k)$ is 1 when $k$ isn't a power of two, and $2k + 1$ when it is. Define $T'(k) = T(k) - 1$, so that we can write

$$\sum_{k=0}^{M-1} T(k) = \sum_{k=0}^{M-1} (1 + T'(k)) = M + \sum_{k=0}^{M-1} T'(k).$$

Now, most of the $T'(k)$ terms are 0, and in fact we can simply count the terms that aren't 0 by explicitly using the powers of two from $2^0$ to $2^{\lfloor \log(M-1) \rfloor}$. So then

$$\begin{aligned}
\sum_{k=0}^{M-1} T(k) &= M + \sum_{k=0}^{M-1} T'(k) \\
&= M + \sum_{i=0}^{\lfloor \log(M-1) \rfloor} T'(2^i) && \text{(substituting } k = 2^i\text{)} \\
&= M + \sum_{i=0}^{\lfloor \log(M-1) \rfloor} 2 \cdot 2^i \\
&= M + 2 \cdot \left(2^{\lfloor \log(M-1) \rfloor + 1} - 1\right) && \left(\textstyle\sum_{i=0}^{d} 2^i = 2^{d+1} - 1\right) \\
&= M - 2 + 4 \cdot 2^{\lfloor \log(M-1) \rfloor}
\end{aligned}$$

The last term there looks a little intimidating, but note that the floor doesn't change the value of the exponent much, and that the power and the log roughly cancel each other out:

$$\begin{aligned}
\log(M-1)-1 \leq \lfloor \log(M-1) \rfloor &\leq \log(M-1) \\
2^{\log(M-1)-1} \leq 2^{\lfloor \log(M-1) \rfloor} &\leq 2^{\log(M-1)} \\
\frac{1}{2}(M-1) \leq 2^{\lfloor \log(M-1) \rfloor} &\leq M-1 \\
2(M-1)+(M-2) \leq \sum_{k=0}^{M-1} T(k) &\leq 4(M-1)+M-2 \\
3m-4 \leq \sum_{k=0}^{M-1} T(k) &\leq 5m-6
\end{aligned}$$

So the total cost is between roughly $3m$ and $5m$. This might sound linear, but remember that we are considering a sequence of $M$ operations, and so the key point is that the *average cost per operation* in a sequence is actually between 3 and 5, which is constant! So we conclude that the *amortized cost* of dynamic array insertion is $\Theta(1)$, formalizing our earlier intuition that the inefficient insertions are balanced out by the much more frequent fast insertions.

You may find it strange that the key variable is the number of operations, not the size of the data structure like usual. In amortized analysis, the size of the data structure is usually tied implicitly to the number of operations performed.

*Accounting (banker's) method*

Now let us perform the analysis again through a different technique known as the **accounting method**, also known as the **banker's method**. As before, our setup is to consider a sequence of $M$ insertion operations on our dynamic array. In this approach, we do not try to compute the total cost of these insertions directly, but instead associate with each operation a *charge*, which is simply a positive number, with the property that *the total value of the $M$ charges is greater than or equal to the total cost of the $M$ operations*.

You can think of cost vs. charge in terms of a recurring payment system. The *cost* is the amount of money you have to pay at a given time, and the *charge* is the amount you actually pay. At any point in time the total amount you've paid must be greater than or equal to the total amount you had to pay, but for an individual payment you can overpay (pay more than is owed) to reduce the amount you need to pay in the future.

In the simplest case, the charge is chosen to be the same for each operation, say some value $c$, in which case the total charge is $cM$. Let $T(M)$ be the total cost of the $M$ operations in the sequence. If we assume that $T(M) \leq cM$, then $T(M)/M \leq c$, and so the average cost per operation is at most $c$. In other words, the chosen charge $c$ is an upper bound on the amortized cost of the operation.

The motivation for this technique is that in some cases it is difficult to compute the total cost of a sequence of $M$ operations, but more straightforward to compute a charge value and argue that the charges are enough to cover the total cost of the operations. This style of argument tends to be quite *local* in nature: the charges are usually associated with particular elements or components of the data structure, and they cover the costs associated only with those elements. Let us make this more concrete by studying one charge assignment for dynamic array insertion.

Warning: this is just a rule of thumb when it comes to assigning charges, and not every accounting argument follows this style.

**Charging scheme for dynamic array insertion**. Each insertion gets a charge of **five**, subject to the following rules.

- The $i$-th insertion associates five "credits" with index $i$.
- One credit is immediately used (and removed) to pay for the insertion at index $i$.
- Let $2^k$ be the smallest power of two that is $\geq i$. When the $2^k$-th insertion occurs, the remaining four credits associated with index $i$ are used.

Think of a credit as representing one array access operation.

What we want to argue is that the "credits" here represent the difference between the charge and cost of the insertion sequence. The number of credits for an index never drops below zero: the first rule ensures that for each index $1 \leq i \leq M$, five credits are added; and the second and third rules together consume these five credits (note that each rule is only applied once per index $i$).

Credits are part of the analysis, *not* the actual data structure. There isn't anything like a `credits` attribute in the program!

So if we can prove that the total number of credits represents the difference between the charge and cost of the insertion sequence, then we can conclude that the total charge is always greater than or equal to the total cost. Because the charge is 5 per operation, this leads to an amortized cost of $\mathcal{O}(1)$, the same result as we obtained from the aggregate method.

As we hinted at above, we need to argue that each credit represents a single array access. So for each insertion, we need to argue that the above rules ensure that the number of credits removed is at least the number of array accesses that are performed for that insertion. Let's do it.

**Lemma 7.1** (Credit analysis for dynamic array insertion)**.** Let $1 \leq i \leq m$. Then the number of credits removed for insertion $i$ is equal to the number of array accesses at index $i$.

*Proof.* There are two cases: when $i$ is not a power of two, and when $i$ is a power of two.

**Case 1**: $i$ is not a power of two. Then there's only one array access, and this is paid for by removing one credit from index $i$.

**Case 2**: $i$ is a power of two. There are $2i + 1$ array accesses. There is one credit removed from index $i$, leaving $2i$ accesses to be accounted for.

There are four credits removed for each index $j$ such that $i$ is the smallest power of two greater than or equal to $j$. How many such $j$ are there? Let $i = 2^k$. The next smallest power of two is $2^{k-1} = i/2$. Then the possible values for $j$ are $\{i/2 + 1, i/2 + 2, \ldots, i\}$; there are $i/2$ such choices.

Note that this includes index $i$ itself; so if $i$ is a power of two, then index $i$ gets 5 credits, and then immediately all of them are used.

So then there are $i/2 \cdot 4 = 2i$ credits removed, accounting for the remaining array accesses. ■

This method often proves more flexible, but also more challenging, than the aggregate method. We must not only come up with a charging scheme for the operation, but also then analyse this scheme to ensure that the total charge

added is always greater than or equal to the total cost of the operations. As we saw, this is *not* the same as arguing that the charge is greater than the cost for an *individual* operation; some operations might have a far higher cost than charge, but have the excess cost be paid for by previous surplus charges. Creative charging schemes take advantage of this by keeping charges high enough to cover the cost of any future operation, while still being low enough to obtain a good bound on the amortized cost of each operation.

---

### Exercise Break!

7.1 Consider a modified implementation of dynamic array INSERT that increases the amount of allocated space by a multiplicative factor of $c > 1$. That is, the amount of allocated space after expansion is $c$ times the space before the expansion. Find the aggregate cost of $M$ operations using this sequence.

7.2 How would you modify the banker's method argument to analyse this modified array?

7.3 Consider a modified implementation of dynamic array INSERT that increases the amount of allocated space by an *additive* factor of $c \geq 1$. That is, the amount of allocated space is $c$ greater than the space before the expansion. Find the aggregate cost of $M$ operations using this sequence.

---

# 8 Disjoint Sets

Our discussion of Kruskal's algorithm at the end of Chapter 6 led to the desire for a new data type that essentially represents partitions of a set of objects, and supports membership tests (which partition has this element?) and merging partitions. We will formalize this in the **Disjoint Set ADT**. This ADT consists of a collection of sets $S_1, \ldots, S_k$ that are *disjoint* (every element appears in exactly one set), and where each set has a *fixed representative element* (a member of the set that serves as the set's identifier).

**Disjoint Set ADT**

- MakeSet($DS, v$): Take a single item $v$ and create a new set $\{v\}$. The new item is the representative element of the new set.
- Find($DS, x$): Find the unique set that contains $x$, and return the *representative element* of that set (which may or may not be $x$).
- Union($DS, x, y$): Take two items and merge the sets that contain these items. The new representative might be one of the two representatives of the original sets, one of $x$ or $y$, or something completely different.

The remainder of this chapter will be concerned with how we can efficiently implement this ADT.

## *Initial attempts*

One obvious approach is to store each set as a linked list, and to pick as the representative element the *last* element in the linked list. Let us try to implement the Disjoint Set ADT using linked lists: the data structure itself will contain an collection of linked lists, where each individual linked list represents a set. Each node in the linked list represents an element in a set. Given any element, it is possible to simply follow the references from that element to the end of its list. We assume that the inputs to Find and Union are references to the node of a linked list. This is an implementation-specific detail that is consistent across all of the implementations we'll look at in this chapter, but it is not necessarily intuitive, so please do keep it in mind.

```
5 def MakeSet(DS, v):
6     node = new Node with value v
```

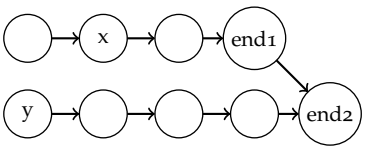
```
    DS.add(node)  # This registers the node as the head of a new linked list
    return node

def Find(DS, x):
    curr = x
    # Traverse the linked list starting at node x until reaching the last node.
    while curr.next is not null:
        curr = curr.next
    return curr

def Union(DS, x, y):
    # hmmm
```

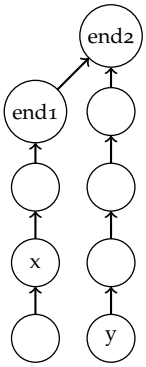
Unfortunately, the UNION operation is challenging to implement, because you cannot merge two linked lists together without knowing the head of at least one of them, and in this case `x` and `y` can certainly be nodes in the middle of the list.

While we can solve this problem using more sophisticated variants of linked lists like doubly-linked lists or circular linked lists, we will do something that is at once simpler and stranger: attach the end of one list to the end of the other.




```
def Union(DS, x, y):
    end1 = Find(DS, x)
    end2 = Find(DS, y)

    # end1 and end2 are the last nodes in their respective linked lists
    if end1 != end2:
        end1.next = end2
```




But wait, you say — if you do this, the result is no longer a linked list, since there are two different nodes pointing to `end2`. And indeed, the combined structure is no longer a linked list — it's a tree! Of course, the links are the reverse of the tree-based data structures we have seen so far: instead of a node having references to each of its children, now each node only has one reference, which is to its *parent*.

Below is a complete implementation of the Disjoint Set ADT using a collection of trees. You'll quickly find that this is basically the implementation we have already given, except with some renaming to better reflect the tree-ness of the implementation. Also, it will be helpful to keep in mind that the representative elements are now the *root* of each tree.

```
# Renaming to reflect the fact that we're using trees, not linked lists
def MakeSet(DS, v):
    node = new Node with value v
    DS.add(node)  # This registers the node as the root of a new tree
```

```
    return node

def Find(DS, x):
    curr = x
    while curr.parent is not null:
        curr = curr.parent
    return curr

def Union(DS, x, y):
    root1 = Find(DS, x)
    root2 = Find(DS, y)

    # root1 and root2 are the roots of their respective trees
    if root2 != root2
        root1.parent = root2
```

### *Runtime analysis*

Now, let us briefly analyse the running time of the operations for this tree-based disjoint set implementation. The MakeSet operation clearly takes $\Theta(1)$ time, while both Find and Union take time proportional to the distance from the inputs to the root of their respective trees. In the worst case, this is $\Theta(h)$, where $h$ is the *maximum* height of any tree in the collection.

Remember that the disjoint set collection can contain multiple trees, each with their own height. The running time of Find and Union depends on the trees that are involved.

Of course, as with general trees, it is possible for the heights to be proportional to $n$, the total number of items stored in all of the sets, leading to a worst-case running time of $\Theta(n)$.

## *Heuristics for tree-based disjoint sets*

It is quite discouraging that our initial tree-based implementation of disjoint sets has worst-case running time that is linear in the total number of items stored. After all, we could come up with a linear time implementation of disjoint sets that simply stores pairs `(item, set representative)` in a list, and loop over the list for Find and Union.

However, just as we improved our BST implementation of dictionaries by imposing more structure on the tree itself, we will now investigate two heuristics used to reduce the height of the trees for our disjoint set implementation.

### *Union-by-rank*

One idea we borrow from AVL trees is the idea of enforcing some local property of nodes to try to make the whole tree roughly balanced. While we do not have

to enforce the binary search tree property here, we are limited in two other ways:

- We do not want to have to traverse a full tree every time we perform a UNION.
- Nodes only have references to their parent, but not their children (child references turn out to be unnecessary).

Rather than have nodes keep track of their height or balance factor, we will store an attribute called *rank*, which has the following definition:

**Definition 8.1** (rank (disjoint sets))**.** The **rank** of a node in a disjoint set tree is defined recursively as follows:

- The rank of a leaf is 0.
- The rank of an internal node is one plus the maximum rank of its children.

It is no coincidence that this definition is closely related to the recursive definition of tree height you may have seen before. You may wonder, then, why we don't just use height. Indeed, we could for this section; but for the second heuristic we'll study, we want an attribute that never decreases, even when a tree's height decreases. And so for consistency, we use the same attribute for both heuristics.

With this notion of rank in hand, we no longer arbitrarily choose the merged tree root in a UNION; instead, we always use the root with the larger rank. The intuition is that if we merge a bigger tree and a smaller tree, we can make the smaller tree a subtree of the bigger tree without changing the bigger tree's height. We call this modification of the basic UNION algorithm the **union-by-rank** heuristic.

```
# Union-by-rank
def MakeSet(DS, v):
    node = new Node with value v
    DS.add(node)    # This registers the node as the root of a new tree
    node.rank = 0   # set the rank
    return node

def Union(DS, x, y):
    root1 = Find(DS, x)
    root2 = Find(DS, y)

    if root1 == root2:
        return

    # root1 and root2 are the roots of their respective trees
    # Choose the one with the larger rank as the new root.
    # Only need to increment a rank when the two root ranks are equal.
    if root1.rank > root2.rank:
        root2.parent = root1
```

```
33      else if root1.rank < root2.rank:
34          root1.parent = root2
35      else:
36          root1.parent = root2
37          root2.rank = root2.rank + 1
```

Observe that the rank of a node only ever increases by 1, and only if it is merged with a node of the same rank. In all other cases, UNION does not increase rank.

Let us make our previous intuition about attaching the "smaller" tree to the "bigger" one more precise in the following two lemmas.

**Lemma 8.1** (Rank for disjoint sets)**.** Let $T$ be a tree generated by a series of MAKESET and UNION operations using the union-by-rank heuristic. Let $r$ be the rank of the root of $T$, and $n$ be the number of nodes in $T$. Then $2^r \leq n$.

*Proof.* We will prove that the property $2^r \leq n$ is an *invariant* that is preserved by every MAKESET and UNION operation.

For MAKESET, the new tree that is created has one node, and its root has rank 0. The theorem holds in this case, since $2^0 = 1 \leq 1$.

For UNION, suppose we take the union of two trees $T_1$ and $T_2$, with sizes $n_1$ and $n_2$ and ranks $r_1$ and $r_2$, respectively. We assume that $2^{r_1} \leq n_1$ and $2^{r_2} \leq n_2$. Let $n = n_1 + n_2$ be the number of nodes in the union tree, and $r$ be the rank of the root of the union tree. We want to prove that $2^r \leq n$.

**Case 1**: $r_1 > r_2$. In this case, $r = r_1$, since the root of $T_1$ is chosen as the root of the union tree, and its rank doesn't change. So then $2^r = 2^{r_1} \leq n_1 < n$, so the property still holds. The same argument works for the case that $r_2 < r_1$ as well, by switching the 1's and 2's.

**Case 2**: $r_1 = r_2$. In this case, the root of $T_1$ is selected to be the new root, and its rank is increased by one: $r = r_1 + 1$. Then since $2^{r_1} \leq n_1$ and $2^{r_2} \leq n_2$, we get $2^{r_1} + 2^{r_2} \leq n_1 + n_2 = n$. Since $r_1 = r_2$, $2^{r_1} + 2^{r_2} = 2^{r_1+1} = 2^r$, and the property holds. ∎

We'll leave it as an exercise to prove the second lemma, which says that the rank really is a good approximation of the height of the tree.

**Lemma 8.2.** Let $T$ be a tree generated by a series of MAKESET and UNION operations using the union-by-rank heuristic, and let $r$ be the rank of its root. Then the height of $T$ is equal to $r + 1$.

With these two lemmas, we can prove the following much-improved bound on the worst-case running time of our disjoint set operations when we use the union-by-rank heuristic.

**Theorem 8.3** (Worst-case running time using union-by-rank)**.** The worst-case running time of FIND and UNION for the disjoint set implementation using union-by-rank is $\Theta(\log n)$, where $n$ is the number of items stored in the sets.

*Proof.* Note that the manipulations involving the new `rank` attribute in the implementation are all constant time, and so don't affect the asymptotic running time of these algorithms. So, by our previous argument, the worst-case running time of these two operations is still $\Theta(h)$, where $h$ is the maximum height of a tree in the disjoint set data structure. Now we show that $h = \mathcal{O}(\log n)$, where $n$ is the total number of items stored in the sets.

Let $T$ be a tree in the data structure with height $h$, $r$ be the rank of the root of this tree, and $n'$ be the size of this tree. By our previous two lemmas, we have the chain of inequalities

Obviously, $n' \leq n$.

$$h = r + 1 \leq \log n' + 1 \leq \log n + 1.$$

It follows immediately that $h = \mathcal{O}(\log n)$. We leave the lower bound as an exercise. ■

### Exercise Break!

8.1 Prove that the worst-case running time of a FIND or UNION operation when using union-by-rank is $\Omega(\log n)$, where $n$ is the number of items stored in the sets.

8.2 Suppose we have a disjoint set collection consisting of $n$ items, with each item in its own disjoint set.

Then, suppose we perform $k$ UNION operations using the union-by-rank heuristic, where $k < n$, followed by one FIND operation. Give a tight asymptotic bound on the worst-case running time of this FIND operation, in terms of $k$ (and possibly $n$).

*Path compression*

Our previous heuristic involved some optimization on the UNION operation to get a better bound on the height of the trees in the disjoint set collection. Such mutating operations are a natural place for optimization: given that we are modifying the structure of the sets, it is natural to think about how to best do this. In this section, we will investigate a different heuristic, which performs a mutation operation in a previously read-only operation, FIND. This draws inspiration from the common query optimization of *caching*, in which the result of a query are saved to make future repetitions of the same query faster. When we perform a FIND operation on a node $x$, we will explicitly store the resulting set representative by *restructuring* the tree so that performing a FIND on that same node in the future is guaranteed to run in constant time.

Think of "query" here as a generic term for any computation that returns a value, like FIND.

When a FIND operation takes a long time, it is because the input node is quite far from the root of the tree, and all of the node's ancestors must be traversed. The key insight is that unlike most other tree operations we have looked at in this course, *there is no reason to preserve the structure of the tree during a* FIND *operation*. Because each node in a tree only cares about its set representative, there is no reason a node shouldn't just point directly to it, other than the fact that set representatives change during UNIONS. So the **path compression** heuristic is the following: after the set representative is found, *all* of the nodes traversed have their `parent` attribute set to this set representative. In purely tree terms, the tree is restructured so that the node and all of its ancestors are made direct children of the root.

Hopefully it is clear why this is called "path compression." The (potentially long) path traversed from input node to root is split up, with each node on the path now "compressed" by becoming a child of the root.

```
# Path compression
def Find(DS, x):
    root = x
    # After this loop, root is the root of the tree containing x.
    while root.parent is not null:
        root = root.parent

    # Path compression: set all ancestors of x to be children of root.
    curr = x
    while curr.parent is not null:
        next_curr = curr.parent
        curr.parent = root
        curr = next_curr
```

*Running time analysis for path compression*

Despite all this fanfare about path compression, it is important to realize that this heuristic does not improve the speed of a FIND operation that has a long path — this operation still takes time proportional to the length of the path, and the worst-case running time is still $\Theta(n)$, where $n$ is the number of items in the sets. This is true of caching in general: the *first* time a query is run, there is no speedup due to previous computations, because there are no previous computations!

Keep in mind that we don't have the union-by-rank guarantee here, so the maximum height of a tree is the number of items in the disjoint sets.

What path compression does guarantee is that any *subsequent* FIND operations on any nodes along the compressed path will take constant time. Whenever you hear a phrase like "operation makes future operations faster," you should think about amortized analysis: even if the initial operation is slow, perhaps its running time cost can be amortized across multiple (faster) FIND operations. And it turns out that with the path compression heuristic, the *amortized running time* of FIND operations has a substantial improvement over the worst-case $\Theta(n)$ bound. We will only perform a special case of the full analysis here. You'll find that this is sufficient to get the general idea, while being able to ignore some extra parameters that come with the most generalized version.

One reminder before we go into the proof. This proof uses the notion of node rank from the previous section, with one important clarification: **node rank is only updated during a merge operation, and only if the node is the new root of the union tree**. This is quite important because path compression can cause tree height to decrease, so keep in mind that rank is not the same as height, and does not change for any node during FIND. There are four important properties of node rank to keep in mind (we leave proofs as exercises):

- The rank of a node is always between 0 and $n - 1$.
- Ranks are *strictly increasing* as you go up a tree; the rank of a node is always

less than the rank of its parent.

- The rank of a node only ever increases, and only during a UNION operation.
- Let $x$ be a node. The rank of the parent of $x$ only ever increases — not only when a UNION operation causes the parent node of $x$ to increase in rank, but *also* when a FIND operation causes $x$ to get a different parent node.

**Theorem 8.4** (Amortized analysis of path compression (special case))**.** Suppose we use a disjoint set implementation that uses the path compression heuristic. Let $n$ be a power of two, and consider a sequence of $n$ MAKESET operations, followed by any number of UNION operations mixed with $n$ FIND operations.

Then the total cost of all $n$ FIND operations is $\mathcal{O}(n \log n)$. Or in other words, the amortized cost of an individual FIND operation is $\mathcal{O}(\log n)$.

*Proof.* We will use the aggregate method to perform the amortized analysis here, but in a more advanced way: rather than computing the cost of each FIND operation separately and adding them up, we will attempt to add up all the costs at once, without knowing the value of any individual cost.

The key insight is that for any individual FIND operation, its running time is proportional to the number of *edges* traversed from the input node up to the root. In this proof, we will call the path traversed by a FIND operation a *find path*. So we define the collection of edges $\mathcal{F}$ in the $n$ find paths, allowing for duplicates in $\mathcal{F}$ because the same edge can belong to multiple find paths. To analyse this, we consider the collection of edges $\mathcal{F}$ that are traversed by any one of the $n$ Find operations. Then the total cost of the sequence of $n$ FIND operations is simply $|\mathcal{F}|$, and the amortized cost of an individual FIND operation is $|\mathcal{F}|/n$.

The second key idea is to partition $\mathcal{F}$ based on the differential rank of the endpoints of each edge. For each $1 \leq i \leq \log n$, define the set

$$\mathcal{F}_i = \{(x, y) \in \mathcal{F} \mid 2^{i-1} \leq rank(y) - rank(x) < 2^i\}.$$

Note that the final set, $\mathcal{F}_{\log n}$, contains an upper bound on the rank difference of $2^{\log n} = n$, so each edge belongs to exactly one of these sets. So $|\mathcal{F}| = \sum |\mathcal{F}_i|$, and our task is now to bound the sizes of each $\mathcal{F}_i$. What can we say about the size of an $\mathcal{F}_i$? It turns out that we can say quite a bit, if we do some careful counting.

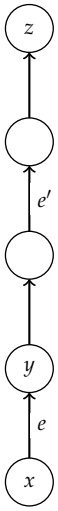
The notation $(x, y)$ indicates an edge traversed from node $x$ to its parent $y$. So $rank(y) \geq rank(x)$.

First, we divide each $\mathcal{F}_i$ into two subgroups based on the relative positions of the edges in their find paths. We say that an edge $e \in \mathcal{F}_i$ is a *last edge* if it is the edge in $\mathcal{F}_i$ closest to the set representative of its tree. Otherwise, we say that the edge $e$ is a *middle edge*. We let $\mathcal{L}_i$ and $\mathcal{M}_i$ be the set of last and middle edges from $\mathcal{F}_i$, respectively.

This is a little abstract, so let's make it more concrete. Suppose we have a find path that contains three edges $e_1, e_2, e_3$ from $\mathcal{F}_i$ (plus possibly many other edges). Suppose $e_3$ is closest to the root of the tree. Then $e_3$ is a last edge, and $e_1$ and $e_2$ are middle edges. In general, if multiple edges from $\mathcal{F}_i$ appear on the same find path, exactly one of them will be a last edge, and all the others will be middle edges.

Since each edge in $\mathcal{F}_i$ is either a last or middle edge, $|\mathcal{F}_i| = |\mathcal{L}_i| + |\mathcal{M}_i|$. Rather than computing the size of $\mathcal{F}_i$, we want to compute the sizes of $\mathcal{L}_i$ and $\mathcal{M}_i$. Since there are a total of $n$ FIND operations in the sequence, there are $n$ find paths. Each find path can have at most one last edge for each rank difference, so $|\mathcal{L}_i| \leq n$.

Don't worry, we aren't going to subdivide any further! Geez.




Now let us count the size of $|\mathcal{M}_i|$. We'll do this by looking at individual nodes. Consider a middle edge $e \in \mathcal{M}_i$, which goes from node $x$ to its parent $y$. At the time of the FIND operation that this edge is traversed, we know the following things:

- There is another edge $e' \in \mathcal{F}_i$ that is closer than $e$ to the root of the tree (since $e$ is a middle edge).
- $y$ is not the set representative of the tree (since it can't be the root).

Let $z$ be the set representative for this tree; so $s$ is distinct from $x$ and $y$. Since the nodes visited on the path from $y$ to $z$ have increasing rank, and $e'$ separates $y$ and $z$ along this path, we know that $rank(z) - rank(y) \geq 2^{i-1}$.

Checkpoint: this inequality only uses the fact that the edge $e' \in \mathcal{F}_i$ is between $w$ and $s$. We haven't yet used the fact that $e$ is also in $\mathcal{F}_i$.

Furthermore, due to path compression, after this FIND operation is complete, $z$ becomes the new parent of $x$. Here is the third big idea. The actual node change itself doesn't matter to our proof; what does matter is that from the point of view of $x$, its parent's rank has increased by at least $2^{i-1}$.

Remember, no node actually changed its rank. It's just that the parent of $x$ changed to one with a bigger rank.

Why is this such a big deal? We can use this to prove the following claim.

**Proposition 8.5.** Let $x$ be a node. There are at most two edges in $\mathcal{M}_i$ that start at $x$.

*Proof.* Suppose for a contradiction that there are three edges $(x, y_1)$, $(x, y_2)$, $(x, y_3) \in \mathcal{M}_i$. Also, assume that by the time a FIND operation visits the edge $(x, y_3)$, previous FIND operations have already visited $(x, y_1)$ and $(x, y_2)$. The previous observation tells us that the rank of the parent of $x$ must have increased by at least $2 \cdot 2^{i-1} = 2^i$, i.e., $rank(y_3) \geq rank(y_1) + 2^i$.

This is *finally* where we use the fact that these edges are in $\mathcal{M}_i$! This tells us that $rank(y_1) \geq rank(x) + 2^{i-1}$, and so we get $rank(y_3) \geq rank(x) + 2^i + 2^{i-1}$. But the fact that $(x, y_3) \in \mathcal{M}_i$ means that $rank(y_3) \leq rank(x) + 2^i$, a contradiction. ■

Since the above claim is true for any node $x$, this means that each of the $n$ nodes in the disjoint sets can be the starting point for at most two edges in $\mathcal{M}_i$. This implies that $|\mathcal{M}_i| \leq 2n$.

It is actually quite remarkable that the size of each $|\mathcal{M}_i|$ doesn't depend on $i$ itself.

Putting this all together yields a bound on the size of $\mathcal{F}$:

$$\begin{aligned}|\mathcal{F}| &= \sum_{i=1}^{\log n} |\mathcal{F}_i| \\ &= \sum_{i=1}^{\log n} |\mathcal{L}_i| + |\mathcal{M}_i| \\ &\leq \sum_{i=1}^{\log n} n + 2n \\ &= 3n \log n\end{aligned}$$

In other words, the total number of edges visited by the $n$ FIND operations is $\mathcal{O}(n \log n)$, and this is the aggregate cost of the $n$ FIND operations. ■

### *Combining the heuristics: a tale of incremental analyses*

We have seen that both union-by-rank and path compression can each be used to improve the efficiency of the FIND operation. We have performed the analysis on them separately, but what happens when we apply both heuristics at once? This is not merely an academic matter: we have seen that it is extremely straightforward to implement each one, and because union-by-rank involves a change to UNION and path compression involves a change to FIND, it is very easy to combine them in a single implementation.

The challenge here is performing the running time analysis on the combined heuristics. We can make some simple observations:

- The worst-case bound of $\Theta(\log n)$ for FIND from union-by-rank still applies, since it is possible to do a bunch of UNIONS, and then a single FIND on a leaf when the tree has height $\Theta(\log n)$.
- The amortized analysis we did for path compression still holds, since all the key properties of rank that we used still hold.

In other words, the two analyses that we performed for the two heuristics still apply when we combine them. The only lingering question is the amortized cost of a FIND operation, for which we only proved an *upper* bound in the previous section. Given that our analysis for path compression alone had to accommodate for a node of rank $n$, while union-by-rank ensured that a node's rank is at most $\log n$, there may be hope for a better amortized cost.

And indeed, there is. What follows is a tour of the major results in the history of this disjoint set implementation, going from its humble beginnings to the precise analysis we take for granted today. It is truly a remarkable story of an algorithm whose ease of implementation belied the full subtlety of its running time analysis.

The first known appearance of this disjoint set implementation, which combined both union-by-rank and path compression, was in a paper by Bernard A. Galler

and Michael J. Fischer in 1964. However, the authors of this paper did not perform a precise analysis of their algorithm, and in fact the question of how efficient this was, *really*, took almost two decades to resolve.

In 1972, Fischer proved that the amortized cost of each FIND operation is $\mathcal{O}(\log \log n)$, already a drastic improvement over the $\mathcal{O}(\log n)$ bound we proved in the previous section. The next year, John Hopcroft and Jeffrey Ullman published a further improvement, showing the amortized cost of FIND to be $\mathcal{O}(\log^* n)$, where $\log^* n$ is the *iterated logarithm* function, defined recursively as follows:

You should be able to modify our proof to get an amortized cost of $\mathcal{O}(\log \log n)$ for the rank bound that union-by-rank gives us.

$$\log^* n = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{otherwise} \end{cases}$$

In English, this function outputs the number of times the logarithm must be applied to a number to get a result $\leq 1$. To give some context about how slow-growing this function is, we show in a table the *largest* value of $n$ to get the first few natural numbers as output.

If $m$ is the largest number such that $\log^* m = k$, then $n = 2^m$ is the largest number such that $\log^* n = k + 1$.

| Largest $n$ | $\log^* n$ |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 16 | 3 |
| 65536 | 4 |
| $2^{65536}$ | 5 |

For context, the estimated number of atoms in the universe is roughly $2^{272}$.

While this running time is so close to constant that it seems like this should have been the end of it, investigation of this function did not stop at the Hopcroft-Ullman result. In 1975, just two years later, Robert Endre Tarjan published his paper "Efficiency of a Good But Not Linear Set Union Algorithm." His main result was that the amortized cost of each FIND operation is $\Theta(\alpha(n))$, where $\alpha(n)$ is the *inverse Ackermann function*, a function that grows even more slowly than the iterated logarithm.

For comparison to the iterated logarithm, $\alpha(2^{2^{65536}}) = 4$.

And even more impressive than getting a better upper bound was that Tarjan was able to show a matching lower bound, i.e., give a sequence of operations so that the amortized cost of the FIND operations when using both union-by-rank and path compression is $\Omega(\alpha(n))$. This showed that no one else would be able to come along and give a better upper bound for the amortized cost of these operations, and the question was settled — almost. At the end of the paper, Tarjan wrote the following (emphasis added):

> This is probably the first and maybe the only existing example of a simple algorithm with a very complicated running time. The lower bound given in Theorem 16 is general enough to apply to many variations of the algorithm, although it is an open problem whether there is a linear-time algorithm for the online set union problem. **On the basis of Theorem 16, I conjecture that there is no linear-time method, and that the algorithm considered here is optimal to within a constant factor.**

In other words, having completed his analysis of the disjoint set data structure using both union-by-rank and path compression, Tarjan proposed that no other heuristics would improve the asymptotic amortized cost of FIND, nor would any other completely different algorithm.

And in 1989 (!), Michael Fredman and Michael Saks proved that this was true: *any* implementation of disjoint sets would have an amortized cost for FIND of $\Omega(\alpha(n))$. This is quite a remarkable statement to prove, as it establishes some kind of universal truth over all possible implementations of disjoint sets — even ones that haven't been invented yet! Rather than analyse a single algorithm or single data structure, they defined a notion of *innate hardness* for the disjoint set ADT itself that would constrain any possible implementation. This idea, that one can talk about all possible implementations of an abstract data type, or all possible algorithms solving a problem, is truly one of the most fundamental — and most challenging — pillars of theoretical computer science.

---

## Exercise Break!

8.3 The idea of using an attribute to store the rank of each node is a powerful one, and can be extended to storing all sorts of metadata about each set.

Use an augmented version of the tree-based disjoint set data structure to support the following two operations, using only linear additional space:

- MAX($DS, x$): Return the maximum value in the set containing $x$
- MIN($DS, y$): Return the minimum value in the set containing $y$

What is the running time of these two operations?

8.4 Why is it harder to augment this data structure to store the median value of each set? More generally, can you characterize the *kinds* of queries that are "easy" to support using an augmented disjoint set data structure?

8.5 We have seen in Chapter 6 how to use depth-first search (or breadth-first search) to detect cycles in a graph.

Consider the following *dynamic* version of this problem: we are given $n$ vertices, and originally have no edges between them. Then edges are added, one at a time. Each time an edge is added, we want to report if that edge forms a cycle in the graph.

(a) Suppose that each time an edge is added, we perform a depth-first search starting at one of its endpoints. What is the amortized cost per edge addition in this case?

(b) Give an algorithm to support this cycle detection using disjoint sets, and analyse the amortized cost per edge addition in this case.

---