Data Structures and Algorithms

PETER LJUNGLÖF
ALEX GERDES
(EDITORS)

Art. No xxxxx
ISBN xxx–xxx–xx-x
Edition 1:1

Printed by The Printer, City 2012

Foreword

This book is a companion to the online book with the same name, a kind of "executive summary" of the main concepts and ideas within data structures and algorithms. The online book also contains interactive explanations, visualisations and exercises, and can be reached from the following URL:

https://chalmersgu-data-structure-courses.github.io/dsabook

Online book

Visit the site

The table of contents of the printed and online versions are the same, but sometimes there is additional information online which we chose not to include in this companion. In these cases there are QR codes in the margin that will lead you directly to that section online. For example, the QR code directly to the right will lead you to the main book website.

THE OPENDSA PROJECT

This book started out as a modernised and simplified version of a collection of teaching modules about data structures and algorithms from the open-source OpenDSA project, developed and maintained by Cliff Shaffer at Virginia Tech University. That project is still alive and thriving, and is much more than just a book – in their own words, it is…

> "…infrastructure and materials to support courses in a wide variety of Computer Science-related topics such as Data Structures and Algorithms (DSA), Formal Languages, Finite Automata, and Programming Languages. OpenDSA materials include many visualisations and interactive exercises. Our philosophy is that students learn best when they engage the material and then practice it until they have demonstrated their proficiency."

OpenDSA

Visit the project page

OpenDSA is well worth a visit, regardless if you like our book or not:

https://opendsa-server.cs.vt.edu/

ACKNOWLEDGEMENTS

Given that this is a live open source project, the texts in this book, as well as the visualisations and exercises in the online version, have been written and developed by a large number of people. However, most of the credit goes to Cliff Shaffer, who together with his team is developing the OpenDSA project and website.

CONTENTS

Introduction

How many cities with more than 100,000 people lie within 200 kilometres of Paris, France? How many people in Swedish towns with less than 50,000 people earn less than 50% of the average income of people in Sweden? How much more CO_2 will be emitted if I travel by plane from Gothenburg, Sweden, to Düsseldorf, Germany, compared to if I take the train? How can I see if a text contains plagiarism, i.e., if it copied some parts from another existing text? To answer questions like these, it is not enough to have the necessary information. We must organise that information in a way that allows us to find the answers in time to satisfy our needs.

Representing information is fundamental to computer science. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information – usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this book is about – helping you to understand how to structure information to support efficient processing.

Any course on data structures and algorithms will try to teach you about three things:

1. It will present a collection of commonly used data structures and algorithms. These form a programmer's basic "toolkit". For many problems, some data structure or algorithm in the toolkit will provide a good solution. We focus on data structures and algorithms that have proven over time to be most useful.
2. It will introduce the idea of tradeoffs, and reinforce the concept that there are costs and benefits associated with every data structure or algorithm. This is done by describing, for each data structure, the amount of space and time required for typical operations. For each algorithm, we examine the time required for key input types.
3. It will teach you how to measure the effectiveness of a data structure or algorithm. Only through such measurement can you determine which data

structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.

There are often many approaches to solving a problem. How do we choose between them? At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code, and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Ideally, the resulting program is true to both of these goals. We might say that such a program is "elegant." While the algorithms and program code examples presented here attempt to be elegant in this sense, it is not the purpose of this book to explicitly treat issues related to goal (1). These are primarily concerns for the discipline of Software Engineering. Rather, we mostly focus on issues relating to goal (2).

How do we measure efficiency? Our method for evaluating the efficiency of an algorithm or computer program is called asymptotic analysis. Asymptotic analysis also gives a way to define the inherent difficulty of a problem. Throughout the book we use asymptotic analysis techniques to estimate the time cost for every algorithm presented. This allows you to see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency.

1.1 Motivation

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won't today's efficiency problem be solved by tomorrow's hardware?

The short answer to this question is ***no***! It is not at all certain that a twice as fast computer can solve twice as large problems. On the contrary – most interesting problems are very difficult, in the sense that if we increase the computing power we can still only solve a very small increase in the size of the problem.

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Unfortunately, as tasks become more complex, they become less

like our everyday experience. So today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term "data structure" to mean an organisation or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. These ideas are explored further in Section 1.7 about abstract data types.

Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. The most obvious example is an unsorted array containing all of the data items. It is possible to perform all necessary operations on an unsorted array. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days. For example, searching for a given record in a hash table is much faster than searching for it in an unsorted array.

A solution is said to be efficient if it solves the problem within the required resource constraints. Examples of resource constraints include the total space available to store the data – possibly divided into separate main memory and disk space constraints – and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The cost of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

1.2 Selecting a data structure

It should go without saying that people write programs to solve problems. However, sometimes programmers forget this. So it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem. Only by first analysing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that

they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to "improve" a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyse your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalises a data-centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation of those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure.

1. Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically get by with simpler data structures to get an efficient implementation, while dynamic applications often require something more complicated.
2. Can data items be deleted? If so, this will probably make the implementation more complicated.
3. Are all data items processed in some well-defined order, or is searching for specific data items allowed? Efficient search generally requires more complex data structures.

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten. For nearly every data structure and algorithm presented in this book, you will see examples of where it is the best choice. Some of the examples might surprise you.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem's characteristics can you determine the best data structure for the task.

1.3 Practical examples

Here we list some real-world examples where data structures and algorithms are crucial.

Search engines The World Wide Web (WWW), often referred to as the Internet, has completely changed how we access and use information. With just a few clicks, we can explore an enormous number of websites filled with text, videos, and images on a wide variety of topics. The amount of content available is staggering, with millions of pages and an endless stream of data at our fingertips. But this vast amount of information also presents challenges. It can be overwhelming to sort through all the sources available online. Finding the exact address of a website on your own is nearly impossible.

This is where search engines come in. Search engines like Google, Bing, and DuckDuckGo help us find the information we need by organising the Internet and making it searchable. This process of organising information is called indexing.

Most of us use search engines every day without thinking about the incredible job they do. The image below shows the results of a search for "binary search":

It is already impressive to be shown a list of relevant web pages, but it is even more astonishing that, in this case, Google found 331 million results in just *0.23 seconds*.

Now imagine you are designing a search engine that must index billions of web pages. Each page has a unique address, called a Uniform Resource Locator (URL), and is associated with a set of keywords. When a user submits a search query, the engine must quickly find all the relevant pages. A simple list is not an efficient way to store and retrieve this information. With billions of pages, it is not practical to check each one in order. Even if looking at a single page takes only a millisecond, going through them all would take years. To handle this challenge, we need smart ways to store and search through the data.

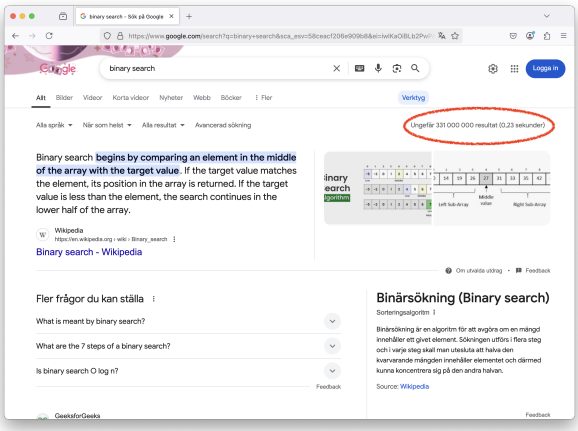


Figure 1.1 Searching the Internet with Google

Let's consider a simpler version of this problem. Suppose we use an array that connects keywords to lists of related web pages. A basic approach would be to go through the array one element at a time until we find the keyword. This works, but it becomes very slow as the array grows larger. To improve efficiency, we can *sort* the array of keywords alphabetically. Then, instead of scanning from the beginning, we start in the middle of the array and compare the search word with the keyword at that position:

- If they match, we return the corresponding list of web pages.
- If the search word is smaller, we continue the search in the lower half of the array.
- If it's larger, we search in the upper half.

Each step cuts the number of possibilities in half. So, how many times can we divide the array in half before we narrow it down to one element? This is a logarithmic process, meaning we divide the search space in half again and again. With an array of one billion keywords, we would need only about 39 steps to find the right one.

This model is simplified. In reality, search engines do much more than just match keywords. They also rank pages by relevance and take many details into account, such as whether letters are uppercase or lowercase, and whether the user is combining terms with "and" or "or".

Databases A company is developing a database system containing information about cities and towns in Europe. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a range query.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a fraction of a second is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the user might tolerate the entire operation to take a little longer, perhaps a couple of seconds. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The B-tree supports large databases, insertion and deletion of data records, and range queries. However, a simple linear index would be more appropriate if the database is created once, and then never changed, such as an atlas accessed from a website.

Section 1.3

More examples online

1.4 Mathematical preliminaries

This section presents the mathematical preliminaries assumed to be familiar to the reader. It serves as a review and reference, allowing you to revisit relevant sections when encountering unfamiliar notation or mathematical techniques in later chapters. If you're comfortable with these preliminaries, you can safely skip ahead to the next section.

Section 1.4

Read the rest online

1.5 Programming preliminaries

This section explains the pseudocode that we will use throughout the book. In addition, we introduce the programming preliminaries that we assume you are familiar with. If you're comfortable with these preliminaries, you can safely skip ahead to the next section.

Section 1.5

Read the rest online

1.6 Case study: Searching in a list

If you want to find the position in an unsorted array of n objects that stores a particular value, you cannot really do better than simply looking through the array from the beginning and moving toward the end until you find what you are looking for. This algorithm is called sequential search. If you do find it, we call this a successful search. If the value is not in the array, eventually you will reach the end. We will call this an unsuccessful search. Here is a simple implementation for sequential search.

```
// Return the position of an element in an array A.
// If the element is not found, return `null`.
function sequentialSearch(arr, val) -> Int:
    for i in 0 .. arr.size - 1:    // For each element in A,
        if arr[i] == val:          // if we found it
            return i               // return this position.
    return null                    // Otherwise, return null
```

It is natural to ask how long a program or algorithm will take to run. But we do not really care exactly how long a particular program will run on a particular computer. We just want some sort of estimate that will let us compare one approach to solving a problem with another. This is the basic idea of algorithm analysis. In the case of sequential search, it is easy to see that if the value is in position i of the array, then sequential search will look at i values to find it. If the value is not in the array at all, then we must look at all array values. This would be called the worst case for sequential search. Since the amount of work is proportional to the size of the array, we say that the worst case for sequential search has linear cost. For this reason, the sequential search algorithm is sometimes called linear search.

1.6.1 BINARY SEARCH

Sequential search is the best that we can do when trying to find a value in an unsorted array. But if the array is sorted in increasing order by value, then we can do much better. We use an algorithm called binary search.

Let's say we search for the value val in an array. Binary search begins by examining the value in the middle position of the array; call this position mid and the corresponding value val_{mid}. If $val_{mid} = val$, then processing can stop immediately. In this case we are lucky, if not, knowing the middle value provides useful information that can help guide the search process. In particular, if $val_{mid} > val$, then you know that the value val cannot appear in the array at any position greater than mid. Thus, you can eliminate future search in

the upper half of the array. Conversely, if $val_{mid} < val$, then you know that you can ignore all positions in the array less than mid. Either way, **half** of the positions are eliminated from further consideration. Binary search next looks at the middle position in that part of the array where value val may exist. The value at this position again allows us to eliminate half of the remaining positions from consideration. This process repeats until either the desired value is found, or there are no positions remaining in the array that might contain the value val.

Here is the method in pseudocode:

```
function binarySearch(arr, val) -> Int:
    low = 0
    high = arr.size - 1
    while low <= high:                // Stop when low and high meet.
        mid: Int = (low + high) / 2   // Integer division
        if arr[mid] < val:            // Check middle of subarray.
            low = mid + 1             // In upper half.
        else if arr[mid] > val:
            high = mid - 1            // In lower half.
        else:
            return mid                // Found it!
    return null                       // Search value not in array.
```

With the right math techniques, it is not too hard to show that the cost of binary search on an array of n values is at most $\log_2 n$. This is because we are repeatedly splitting the size of the subarray that we must look at in half. We stop (in the worst case) when we reach a subarray of size 1. And we can only cut the value of n in half $\log_2 n$ times before we reach 1.

Variations Binary search is designed to find the (single) occurrence of a value and return its position. A special value is returned if the value does not appear in the array. The algorithm can be modified to implement variations such as returning the position of the *first occurrence* of the value in the array if multiple occurrences are allowed. Another variation is returning the position of the greatest value less than the value we are looking for when it is not in the array.

1.7 Abstract data types

Earlier, we introduced the term data type, which refers to a type along with a collection of operations for manipulating values of that type. For instance, integers form a data type, and addition is an operation that can be performed on them. These are known as *concrete* data types, meaning they consist of actual values and a specific implementation. In contrast, an abstract data type (ADT)

does not specify concrete values or implementations. Instead, it defines a data type purely in terms of a set of operations and the expected behaviour of those operations, as determined by their inputs and outputs. An ADT does not dictate *how* the operations should be implemented, and multiple implementations are often possible. These implementation details are hidden from the user—a concept known as encapsulation. The set of operations offered by an abstract data type is known as its application programming interface (API).

Using an ADT, we can distinguish between the logical behaviour of a data type and its actual implementation in a concrete program. A classic example is the list abstract data type, which support the following set of operations:

```
interface List of T:
    add(i: Int, x: T)    // Adds x at position i; where 0 ≤ i ≤ size.
    get(i: Int) -> T     // Returns the element at position i; where 0 ≤ i < size.
    set(i: Int, x: T)    // Replaces the value at position i with x; where 0 ≤ i < size.
    remove(i: Int)       // Removes the element at position i; where 0 ≤ i < size.
```

A list can be implemented using either an array or a linked list. Users of a list do not need to know which implementation is used in order to make use of its functionality. The actual implementations of an ADT rely on specific *data structures* to realise the desired behaviour of the operations – for example, calculating the size of a list.

Although different implementations of an abstract data type offer the same set of operations, the choice of data structure can significantly impact the *efficiency* of those operations. Often, there are trade-offs involved: optimising one operation may come at the cost of another. For example, an array-based list allows fast access to elements at specific indices, while a linked-list implementation excels at inserting elements at the front. Furthermore, different applications may prioritise different operations. One program might frequently perform operation A, while another relies more heavily on operation B. In such cases, it is often not possible to implement all operations efficiently, so multiple implementations of the same ADT are needed. Additionally, one implementation may be more efficient for small datasets (thousands of elements), whereas another may scale better for large datasets (millions of elements). The most suitable data structure depends on the specific use case, and making informed and well-reasoned choices is one of the central goals of this book.

The concept of an ADT can help us to focus on key issues even in non-computing applications.

Example: Cars

When operating a car, the primary activities are steering, accelerating, and braking. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer", "accelerate", and "brake". Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden.

The concept of an ADT is one instance of an important principle that must be understood by any successful computer scientist: managing complexity through abstraction. A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a *metaphor*. A particular label might be related to other pieces of information or other labels. This collection can in turn be given a label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details.

Example: Computers, hard drives, and CPUs

We apply the label "hard drive" to a collection of hardware that manipulates data on a particular type of storage device, and we apply the label "CPU" to the hardware that controls execution of computer instructions. These and other labels are gathered together under the label "computer". Because even the smallest home computers today have millions of components, some form of abstraction is necessary to comprehend how a computer operates.

Consider how you might go about the process of designing a complex computer program that implements and manipulates an ADT. The ADT is implemented in one part of the program by a particular data structure. While designing those parts of the program that use the ADT, you can think in terms of operations on the data type without concern for the data structure's implementation. Without

this ability to simplify your thinking about a complex program, you would have no hope of understanding or implementing it.

Most of the abstract data types we introduce in this book are collections, that is, structures that store elements of an arbitrary type. The earlier example of a list illustrates this: a list is a collection that holds elements, which can be of any type.

We group these collection-based ADTs into two main categories:

- Linear collections
- Sets and maps

In addition to these, we also introduce graphs, along with their commonly used implementations.

The rest of this gives a high-level overview of the ADTs covered throughout the course. Figure 1.2 summarises these ADTs and highlights how they relate to one another. Each ADT will be discussed in more detail later in the book, including their operations and the data structures used to implement them.

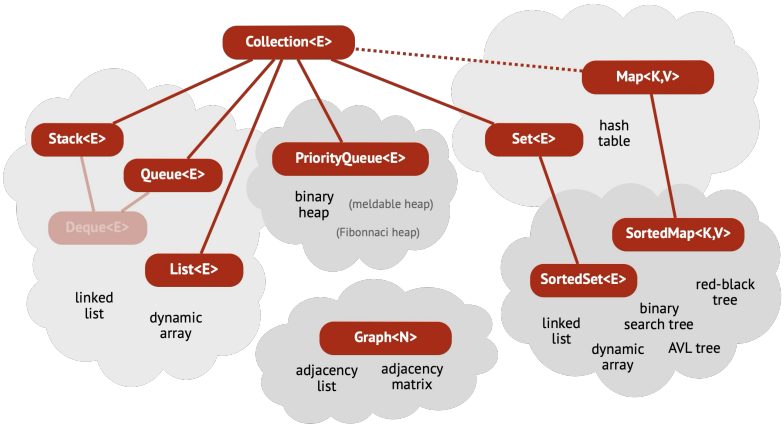


Figure 1.2 An overview of ADTs with their common implementations

1.7.1 LINEAR COLLECTIONS

Linear collections are a category of abstract data types in which the order of the elements matters. This means that each element has a specific position in the sequence, and operations on the ADT are sensitive to this order. Insertion, removal, and retrieval operations often depend on the position within the sequence.

The ordered sequence ADTs found in this book are:

- **Stacks**: sequences with a Last-In-First-Out (LIFO) ordering
- **Queues**: sequences with a First-In-First-Out (FIFO) ordering
- **Double-ended queues** (deques): allow insertion and removal at both ends
- **Priority queues**: return elements based on priority rather than insertion order
- **General lists**: support adding, removing, and accessing elements

Ordered sequences are used in many applications and algorithms where the order of operations or items is important. For example, maintaining a task list, simulating a line of customers, or a editor's undo/redo history.

1.7.2 SETS AND MAPS

Many programming tasks involve *retrieving specific information* from a large dataset. For example, given a collection of people, how do we efficiently find the person with a specific personnummer? Two key abstract data types are commonly used to solve such information retrieval problems:

- **Sets**: represent unordered collections of distinct items. They support operations to *add* and *remove* elements, and to *check* whether a particular element is present. Duplicate elements are not allowed.
- **Maps** (also called dictionaries): represent collections of *key-value pairs*, where each *key* maps to a corresponding *value*. Operations include *adding* or *removing* key-value pairs, *checking* if a key is present, and *retrieving* the value associated with a given key. Like sets, maps do not allow duplicate keys.

Most implementations of both sets and maps are designed to support fast insertion, deletion, and lookup operations, making them ideal for managing collections where quick access to data is important.

1.7.3 GRAPHS

Another well-known abstract data type is the *graph*. Graphs are used to model relationships between elements, where each element is called a *node* or *vertex*. A *relation* between two nodes is represented by an *edge*, which may carry additional information to describe the nature or strength of the relationship—such as distance, cost, or capacity.

Graphs appear in many real-world scenarios, often in surprising ways. A classic example is a map, where cities are represented as nodes and roads (with distances) as edges. Graph algorithms can then be used to solve problems such as finding the shortest route between two cities. Another, less obvious example

is the structure of Java programs: the dependencies between Java classes can be represented as a graph. This representation helps us determine the correct order to compile classes based on their dependencies.

Graphs are a fundamental concept in computer science, and we dedicate an entire chapter to them in this book. Chapter 13 explores how graphs can be represented and how we can traverse and manipulate them using various algorithms.

1.8 Review questions

Section 1.8

Answer quiz online

This final section contains some review questions about the contents of this chapter.

CHAPTER 2

Algorithm analysis, part 1: Introduction

How long will a program take when I run it on a dataset ten times as large? If a particular program is slow, is it badly implemented or is it solving a hard problem? What order of improvement can I expect if I switch to a better algorithm? Questions like these ask us to consider the difficulty of a problem and the efficiency of approaches to solving it.

This chapter introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. We focus on a methodology known as asymptotic algorithm analysis, or simply asymptotic analysis.

Asymptotic analysis estimates the resource consumption of an algorithm, called its complexity. Here, resource consumption can mean runtime, memory use, API calls, or any other measure. Instead of computing this resource consumption exactly, asymptotic analysis is only interested in its growth rate (also called order of growth). The growth rate is what determines the resource consumption for large inputs. Thankfully, growth rate expressions are relatively easy to compare. This allows us to decide which of two algorithms is better at solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program.

After reading this chapter, you should understand:

- The concept of complexity of an algorithm, the resource usage of an algorithm as a function of an input parameter. Different kinds of complexity such as worst-case and average-case.
- The concept of growth rate or order of growth of a (mathematical) function. How to compute and compare growth rates of functions. Notations such as big-O to describe upper and lower bounds of growth rates.
- The asymptotic complexity of an algorithm, which is the growth rate of its complexity. Sometimes, this is just called the growth rate of the algorithm.
- The difference between the asymptotic complexity of an algorithm (or pro-

gram) and that of a problem. The latter is the best asymptotic complexity over all algorithms that solve the problem.

The chapter concludes with a brief discussion of the practical difficulties encountered when empirically measuring the cost of a program, and some principles for code tuning to improve program efficiency.

2.1 Problems, algorithms, and programs

Programmers commonly deal with *problems*, *algorithms*, and computer *programs*. These are three distinct concepts.

2.1.1 PROBLEMS

As your intuition would suggest, a problem is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a "reasonable" amount of time.

Problems can be viewed as functions in the mathematical sense. A function is a matching between inputs (the domain) and outputs (the range). An input to a function might be a single value or a collection of information. The values making up an input are called the parameters of the function. A specific selection of values for the parameters is called an instance of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behaviour of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type `date` to a typical Linux command line prompt, you will get the current date. Naturally

the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program's full set of inputs. Even a "random number generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The limits to what functions can be implemented by programs is part of the domain of Computability.

2.1.2 ALGORITHMS

An algorithm is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function). This book covers many problems, and for several of these problems we will see more than one algorithm. For the important problem of sorting there are over a dozen commonly used algorithms!

The advantage of knowing several solutions to a problem is that solution **A** might be more efficient than solution **B** for specific variation of the problem, or for a specific class of inputs to the problem, while solution **B** might be more efficient than **A** for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). ther might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties:

1. It must be *correct*. In other words, it must implement the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.
2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood – and doable – by the person

or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.

3. There must be *no ambiguity* about which step is performed next. Typically, the next step follows directly from the algorithm's description. When the algorithm includes selection mechanisms, such as `if`-statements, these allow for alternative execution paths. However, the choice of which path to follow must always be unambiguous at the moment the decision is made.
4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the `while` and `for` loop constructs. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.
5. It must *terminate* for the intended input. In other words, it may not go into an infinite loop.

2.1.3 PROGRAMS

We often think of a computer program as an instance, or concrete representation, of an algorithm in some programming language. Algorithms are usually presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm, because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, people often use the terms "algorithm" and "program" interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual prob-

lems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

In this book we will usually present algorithms and not programs. We assume that you as the reader is competent enough to be able to translate our descriptions and pseudocode into working programs in your favourite language.

To summarise: A problem is a function or a mapping of inputs to outputs. An algorithm is a recipe for solving a problem whose steps are concrete and unambiguous. Algorithms must be correct, of finite length, and must terminate for all inputs. A program is an instantiation of an algorithm in a programming language.

2.2 Invariants, preconditions, and postconditions

When we formulate a problem, and describe a data structure or an algorithm, it is useful to describe them in terms of invariants, or pre- and postconditions.

These properties are very useful tools for analysing the correctness of an algorithm. But they are also pedagogical tools, they help our understanding how an data structure or algorithm works.

Invariants Simply formulated, an invariant is a condition (or a set of conditions) that must always hold for your data structure, or algorithm.

Invariants can be more or less detailed, but if we make them too detailed they might be a hinder for our understanding. Therefore we will keep them on a quite high level, in line with our decision to present algorithms in high-level pseudocode. We trust that you, the reader, is experienced enough to translate both pseudocode and invariants into more detailed descriptions.

Preconditions and postconditions Invariants can often be formulated as preconditions and postconditions on an algorithm: what do we assume of the inputs to the algorithm, and what can we promise about the output if the input is well-formed?

As an example, assume that we want to insert an element into a sorted list. An natural precondition is then that the input list is already sorted, and a postcondition would then be that the list is still sorted after the element is inserted. These specific pre- and postconditions are perhaps too obvious, so we usually don't spell them out – they are understood from the problem formulation, which says that we are working with a sorted list.

Keeping in line with our high-level approach, we often don't formulate specific pre- and postconditions, but instead write more abstract invariants.

Example: Binary search

In Section 1.6.1 we introduced binary search. What kind of invariants can be useful to understand the algorithm better? A precondition is of course that the array is sorted, but this is so obvious that we don't have to spell it out.

More interesting is to look into the algorithm itself. Binary search consists of a loop which updates two variables, `low` and `high`. So, can we say something about these variables?

Yes, we can say that after each iteration of the binary search loop, we know that the value we search for lies between the cells pointed to by `low` and `high` (if it's in the array at all). This means that we can formulate the following invariants:

- When searching for `v` in a sorted array `arr`, then `arr[low]` $\leq$ `v` $\leq$ `arr[high]` after each iteration of the main loop.
- The interval `high - low` strictly decreases in each iteration.

Knowing these invariants, it becomes much easier to convince ourselves that the algorithm is correct and always terminates.

2.3 Comparing algorithms

How do you compare two algorithms for solving some problem in terms of efficiency? We could implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was "better written" than the other, and therefore the relative qualities of the underlying algorithms are not truly represented by their implementations. This can easily occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favour one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyse the *time* required for an *algorithm* (or the instantiation of an algorithm in the form of a program), and the *space* required for a *data structure*.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's (or the network's) resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, if you want to compare two programs derived from two algorithms for solving the same problem, they should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient". In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

2.3.1 BASIC OPERATIONS AND INPUT SIZE

Of primary consideration when estimating an algorithm's performance is the number of basic operations required by the algorithm to process an input of a certain size. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analysed. Size is often the number of inputs processed. For example, when comparing sorting algorithms the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Because the most important factor affecting running time is normally the size of the input, for a given input size n we often express the time T to run the algorithm as a function of n, written as $T(n)$. We will always assume $T(n)$ is a non-negative value.

Example: Largest value in an array

Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far:

```
// Return position of largest value in integer array arr
function largest(arr):
    pos = 0                     // Position of pos element seen.
    for i in 1 .. arr.size-1:   // For each element,
        if arr[pos] < arr[i]:   // if arr[i] is larger,
            pos = i             // remember its position.
    return pos                  // Return pos position
```

Here, the size of the problem is `arr.size`, the number of integers stored in the array. The basic operation is to compare a value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the values or their positions in the array.

Let us call c the amount of time required to compare two integers in function `largest`. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable i because this must be done for each value in the array, or the time

for the actual assignment when a larger value is found, or the little bit of extra time taken to initialise `pos`. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run `largest` is therefore approximately cn, because we must make n comparisons, with each comparison costing c time. We say that function `largest` (and by extension, the largest-value sequential search algorithm for any typical implementation) has a running time expressed by the equation

$$T(n) = cn$$

Section 2.3.1

More examples online

More examples can be found in the online version of the book.

2.4 Growth rates

Imagine that you have a problem to solve, and you know of an algorithm whose running time is proportional to n^2 where n is a measure of the input size. Unfortunately, the resulting program takes ten times too long to run. If you replace your current computer with a new one that is ten times faster, will the n^2 algorithm become acceptable? If the problem size remains the same, then perhaps the faster computer will allow you to get your work done quickly enough even with an algorithm having a high growth rate. But a funny thing happens to most people who get a faster computer. They don't run the same problem faster. They run a bigger problem! Say that on your old computer you were content to sort 10,000 records because that could be done by the computer during your lunch break. On your new computer you might hope to sort 100,000 records in the same time. You won't be back from lunch any sooner, so you are better off solving a larger problem. And because the new machine is ten times faster, you would like to sort ten times as many records.

If your algorithm's growth rate is linear (i.e., if the equation that describes the running time on input size n is $T(n) = cn$ for some constant c), then 100,000 records on the new machine will be sorted in the same time as 10,000 records on the old machine. If the algorithm's growth rate is greater than cn, such as c_1n^2, then you will *not* be able to do a problem ten times the size in the same amount of time on a machine that is ten times faster.

2.4.1 GETTING A FASTER COMPUTER

How much larger problems a faster computer solve in the same amount of time? Say that the old machine could solve a problem of size n in an hour, and that

the new machine is ten times faster than the old. What is the largest problem that the new machine can solve in one hour? Table 2.1 shows just that for five running-time functions.

Table 2.1 The increase in problem size that can be run in the same time on a computer that is ten times faster

Growth rate	Max n (old)	Max n' (new)	Relation n and n'	n'/n
$10n$	$1,000$	$10,000$	$n' = n \cdot 10$	10
$20n$	500	$5,000$	$n' = n \cdot 10$	10
$5n \log n$	250	$1,842$	$n' \approx \frac{0.1386n}{W(0.1386n)}$	7.4
$2n^2$	70	223	$n' = n \cdot \sqrt{10}$	3.2
2^n	13	16	$n' = n + 3$	≈ 1

Explanations:

- The first column lists the right-hand sides for five growth rate equations.
- The second column shows the maximum value for n that can be run in 10,000 basic operations on the old machine.
- The third column shows the value for n', the new maximum size for the problem that can be run in the same time on the new machine that is ten times faster. Variable n' is the greatest size for the problem that can run in 100,000 basic operations.
- The fourth column shows how the size of n changed to become n' on the new machine.
- The fifth column shows the increase in problem size as the ratio of n' to n.

This table illustrates many important points. The first two equations are both linear; only the value of the constant factor has changed. In both cases, the machine that is ten times faster gives an increase in problem size by a factor of ten. In other words, while the value of the constant does affect the absolute size of the problem that can be solved in a fixed amount of time, it does not affect the *improvement* in problem size (as a proportion to the original size) gained by a faster computer. This relationship holds true regardless of the algorithm's growth rate: Constant factors never affect the relative improvement gained by a faster computer.

An algorithm with time equation $T(n) = 2n^2$ does not receive nearly as great an improvement from the faster machine as an algorithm with linear growth rate. Instead of an improvement by a factor of ten, the improvement is only

the square root of that: $\sqrt{10} \approx 3.16$. Thus, the algorithm with higher growth rate not only solves a smaller problem in a given time in the first place, it *also* receives less of a speedup from a faster computer. As computers get ever faster, the disparity in problem sizes becomes ever greater.

The algorithm with growth rate $T(n) = 5n \log n$ improves by a greater amount than the one with quadratic growth rate, but not by as great an amount as the algorithms with linear growth rates.

Note that something special happens in the case of the algorithm whose running time grows exponentially. If you look at its plot on a graph, the curve for the algorithm whose time is proportional to 2^n goes up very quickly as n grows. The increase in problem size on the machine ten times as fast is about $n + 3$ (to be precise, it is $n + \log_2 10$). The increase in problem size for an algorithm with exponential growth rate is by a constant addition, not by a multiplicative factor. Because the old value of n was 13, the new problem size is 16. If next year you buy another computer ten times faster yet, then the new computer (100 times faster than the original computer) will only run a problem of size 19. If you had a second program whose growth rate is 2^n and for which the original computer could run a problem of size 1000 in an hour, than a machine ten times faster can run a problem only of size 1003 in an hour! Thus, an exponential growth rate is radically different than the other growth rates shown in the table. The significance of this difference is an important topic in computational complexity theory.

Instead of buying a faster computer, consider what happens if you replace an algorithm whose running time is proportional to n^2 with a new algorithm whose running time is proportional to $n \log n$. In a graph relating growth rate functions to input size, a fixed amount of time would appear as a horizontal line. If the line for the amount of time available to solve your problem is above the point at which the curves for the two growth rates in question meet, then the algorithm whose running time grows less quickly is faster. An algorithm with running time $T(n) = n^2$ requires $1000 \times 1000 = 1,000,000$ time steps for an input of size $n = 1000$. An algorithm with running time $T(n) = n \log n$ requires $1000 \times 10 = 10,000$ time steps for an input of size $n = 1000$, which is an improvement of much more than a factor of ten when compared to the algorithm with running time $T(n) = n^2$. Because $n^2 > 10n \log n$ whenever $n > 58$, if the typical problem size is larger than 58 for this example, then you would be much better off changing algorithms instead of buying a computer ten times faster. Furthermore, when you do buy a faster computer, an algorithm with a slower growth rate provides a greater benefit in terms of larger problem size that can run in a certain time on the new computer.

2.4.2 COMPARING DIFFERENT GROWTH RATES

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. Figure 2.1 shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates that are representative of typical algorithms are shown.

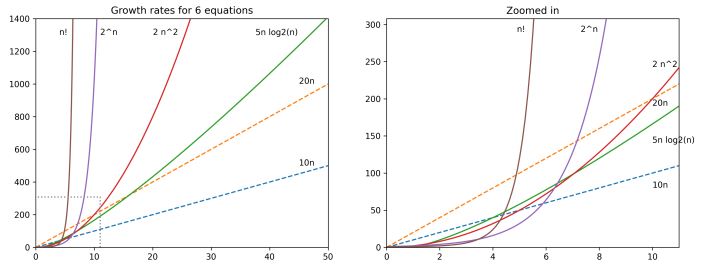


Figure 2.1 Illustration of the growth rates for six equations. The right view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

The two equations labeled $10n$ and $20n$ are graphed by straight lines. A growth rate of cn (for c any positive constant) is often referred to as a linear growth rate or running time. This means that as the value of n grows, the running time of the algorithm grows in the same proportion. Doubling the value of n roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a quadratic growth rate. In the figure, the line labeled $2n^2$ represents a quadratic growth rate. The line labeled 2^n represents an exponential growth rate. This name comes from the fact that n appears in the exponent. The line labeled $n!$ also grows exponentially.

As you can see from the figure, the difference between an algorithm whose running time has cost $T(n) = 10n$ and another with cost $T(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $T(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $T(n) = 2n^2$ is slower than the algorithm with cost $T(n) = 20n$. This graph also shows that the equation $T(n) = 5n \log n$ grows somewhat more quickly than both $T(n) = 10n$ and $T(n) = 20n$, but not nearly so quickly as the equation $T(n) = 2n^2$. For constants $a, b > 1$, n^a grows faster than either $\log^b n$ or $\log n^b$. Finally, algorithms

with cost $T(n) = 2^n$ or $T(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a, b \geq 1$, a^n grows faster than n^b.

We can get some further insight into relative growth rates for various algorithms from Table 2.2 below. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

Table 2.2 Costs for representative growth rates

n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	10^3	10^3	10^6
100	6.6	100	664	10^4	10^6	10^{30}	10^{158}
$1K = 1000$	10	1000	10^4	10^6	10^9	10^{300}	10^{2567}
$10K = 10^4$	13.3	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	10^{3000}	$10^{35,659}$
$100K = 10^5$	16.6	10^5	$1.6 \cdot 10^6$	10^{10}	10^{15}	$10^{30,000}$	⋯
$1M = 10^6$	20	10^6	$2 \cdot 10^7$	10^{12}	10^{18}	$10^{300,000}$	⋯
$1G = 10^9$	30	10^9	$3 \cdot 10^{10}$	10^{18}	10^{27}	⋯	⋯

2.5 Best, worst, and average cases

For some algorithms, the running time is always determined by the input size n. Consider the algorithm that finds the largest value in an array using sequential search algorithm. For any given size n there are inifinitely many possible inputs – all imaginable arrays of size n. However, no matter what array of size n that the algorithm looks at, its running time will always be the same. This is because it always looks at every element in the array exactly once.

For most algorithms however, different inputs of a given size require different amounts of time. For example, consider the problem of searching an array containing n integers to find the one with a particular value K. The sequential search algorithm begins at the first position in the array and looks at each value in turn until K is found, and then it stops. So there is a wide range of possible running times for the algorithm, which is different from the largest-value search algorithm above.

- If K is the first element, only one value is examined – this is the *best case*, because it is not possible for sequential search to look at less than one value.
- If K is the last element, the algorithm must examine n values – this is the

worst case, because sequential search never looks at more than each of the n values in the array.

If we run sequential search many times on many different arrays of size n, and many different values of K, we expect the algorithm on average to go halfway through the array before finding the value we seek. So, on average, the algorithm examines $(n+1)/2$ values – this is the *average case*.

However, note that the average case depends on an important assumption: that the searched value K is independent from how the values in the array is distributed! More about that below.

When analysing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterisation of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behaviour of the algorithm. However, there are rare instances where a best-case analysis is useful – in particular, when the best case has high probability of occurring. For example, if we know that the array we want to sort is *almost sorted*, we can take advantage of the best-case running time of insertion sort.

How about the worst case? The advantage to analysing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications – in particular when we wish to aggregate the cost of running the program many times on many different inputs – worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the *typical* behaviour of the algorithm on inputs of size n.

2.5.1 THE PROBLEM WITH AVERAGE CASE

Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with

value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does *not* necessarily examine half of the array values in the average case.

How the data is distributed has a significant effect on almost all data structures and algorithms, such as those based on hashing and binary search trees. Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance. Unusual data distributions can also be used to advantage, such as is done by self-organising lists and splay trees.

Relying on average-case analysis can be very dangerous for all applications where you don't have full control over your data. For example, all kinds of databases that are publicly available are a risk. Even if "bad" data are extremely unlikely to occur in your use cases, you can be certain that there are people out there who gladly will try to exploit any kind of weakness in your system. If there is just a tiny risk of a worst-case scenario, this opens up for *denial-of-service attacks* on your system.

In summary, for real-time applications and for applications that are openly available, we should always prefer a worst-case analysis of an algorithm. In other cases we usually desire an average-case analysis, but then we need to know enough about how the input is distributed. However, this is often difficult to calculate, so in most cases we must anyway resort to worst-case analysis.

2.6 Asymptotic analysis

Figure 2.1 gives two views of a graph illustrating the growth rates for six equations.

Despite the larger constant for the curve labeled $10n$ in the figure above, $2n^2$ crosses it at the relatively small value of $n = 5$. What if we double the value of the constant in front of the linear equation? As shown in the graph, $20n$ is surpassed by $2n^2$ once $n = 10$. The additional factor of two for the linear growth rate does not much matter. It only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants. For these reasons, we usually ignore the constants when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and

keeps us thinking about the most important aspect: the growth rate. This is called asymptotic algorithm analysis. To be precise, asymptotic analysis refers to the study of an algorithm as the input size "gets big" or reaches a limit (in the calculus sense). However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons.

In rare situations, it is not reasonable to ignore the constants. When comparing algorithms meant to run on small values of n, the constant can have a large effect. For example, if the problem requires you to sort many collections of exactly five records, then a sorting algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance. There are rare cases where the constants for two algorithms under comparison can differ by a factor of 1000 or more, making the one with lower growth rate impractical for typical problem sizes due to its large constant. Asymptotic analysis is a form of "back of the envelope" estimation for algorithm resource consumption. It provides a simplified model of the running time or other resource needs of an algorithm. This simplification usually helps you understand the behaviour of your algorithms. Just be aware of the limitations to asymptotic analysis in the rare situation where the constant is important.

2.6.1 ORDERS OF GROWTH

To be able to discuss orders of growth for algorithms we need to do some abstractions. The most important abstraction is to describe the runtime of an algorithm as a mathematical function from the input size to a number. We actually don't care how we encode the input size, as long as it is proportional to the actual size of the input. So we can, e.g., use the number of cells in an array as input, instead of trying to figure out the exact memory usage of the array. And in the same way, we don't care about the unit of measure for the result – it can be actual runtime, in seconds, minutes or hours, but it's more common to think about the number of "basic operations". Already here we have abstracted away lots of things that relate to hardware, which is vital because we want to analyse algorithms, not implementations. So we will say things like "the runtime of algorithm **A** is $f(n)$".

Now, the easiest way to view order of growth is not as an absolute propery of an algorithm, but instead as a relation between functions. When we say that an algorithm is quadratic, we actually mean that the mathematical function $f(n)$ that describes the abstract runtime of the algorithm, is related to the quadratic function $g(n) = n^2$ in some way.

So, how can we relate functions using orders of growth? We do this by saying that one function is a *bound* of another function. E.g., when we say that an

algorithm **A** is quadratic, we actually mean that the function n^2 is an upper bound of **A**. The following are the most common definitions of *upper*, *lower* and *tight* bounds:

Upper bound f is an upper bound of g **iff** f grows *at least as fast* as g, and we write this $f \in O(g)$

Lower bound f is a lower bound of g **iff** f grows *at most as fast* as g, and we write this $f \in \Omega(g)$

Tight bound f is a lower bound of g **iff** both functions grow *at the same rate*, and we write this $f \in \Theta(g)$

2.6.2 DEFINING ORDERS OF GROWTH

How do we define upper and lower bounds? First, if g is an upper bound of f, then this should mean something like $f(n) \leq g(n)$ *in the long run*. What we mean by this is that whenever n becomes sufficiently large, then $f(n)$ should not outgrow $g(n)$.

But this is not all there is to it. We have already mentioned that we want to abstract away from constant factors – if algorithm **A** is twice as fast as algorithm **B**, then they grow at the same rate, and we want our notation to capture that. So what we want to say is that $f(n) \leq k \cdot g(n)$, for some arbitrary constant k.

Now we can give formal definitions of upper, lower and tight bounds:

Upper bound $f \in O(g)$ **iff** there exist positive numbers k and n_0 such that $f(n) \leq k \cdot g(n)$ for all $n > n_0$

Lower bound $f \in \Omega(g)$ **iff** there exist positive numbers k and n_0 such that $f(n) \geq k \cdot g(n)$ for all $n > n_0$
or equivalently: $f \in \Omega(g)$ **iff** $g \in O(f)$

Tight bound $f \in \Theta(g)$ **iff** there exist positive numbers k_1, k_2 and n_0 such that $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ for all $n > n_0$
or equivalently: $f \in \Theta(g)$ **iff** $f \in O(g) \wedge f \in \Omega(g)$

These definitions assume that f and g are *monotonically increasing*, i.e., that they never decrease when the input increases. But since we will only be using them for comparing computational resources, they will always be monotonically increasing.

Example: Comparing two functions

Assume $f(n) = n^2$ and $g(n) = 1000n \log n$. How can we use the definitions above to prove that $f \in \Omega(g)$?

We have to find positive numbers k and n_0 so that $f(n) \geq k \cdot g(n)$. Since g has a constant factor of 1000, we can try with $k = 0.001$:

$$k \cdot g(n) = 0.001 \cdot 1000n \log n = n \log n$$

Now we readily see that $f(n) = n^2$ is larger than $k \cdot g(n) = n \log n$ for all $n \geq 1$, so we can set $n_0 = 1$.

Note that there are plenty of possible values to choose from, such as $k = 1$ and $n_0 = 13,789$. We can even use very large values such as $k = n_0 = 10^{99}$, what we are interested in is after all what happens when n grows infinitly large.

2.6.3 SHOULD WE USE O, Ω OR Θ?

If an algorithm has a lower bound of $\Omega(f)$, we know that it will never run asymptotically faster than f. But this is usually not very useful knowledge, because we are more interested in knowing how the algorithm works on on bad inputs. Therefore the upper bound $O(f)$ is by far the most common complexity measure that people use, and this is what we will be using in this book.

One could argue that $\Theta(f)$ would be an even better measure, because it gives more information about an algorithm. But it is much more difficult to reason about $\Theta(f)$, and therefore we will almost exclusively use the upper bound notation $O(f)$.

So, is the lower bound useless? – No, definitely not. The main use case for Ω is when we want to classify *problems*, not algorithms. One example is when proving that the lower bound for sorting is $\Omega(n \log n)$, which we do in Section 5.3.1. But classifying problems is out of scope for this book, so we will not use Ω much.

2.7 Algorithm analysis in practice

In this section we show how to analyse complexity in practice, by analysing several simple code fragments. As mentioned in the last section, we will only give the upper bound using the big-O notation.

2.7.1 SIMPLIFICATION RULES

Once you determine the running-time equation for an algorithm, it is often a simple matter to derive the big-O expressions from the equation. You do not need to resort to the formal definitions of asymptotic analysis. Instead, you can use the following rules to determine the simplest form.

(1) **Transitivity**: If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$

This first rule says that if some function $g(n)$ is an upper bound for your cost function, then any upper bound for $g(n)$ is also an upper bound for your cost function.

(2) **Constant factor**: If $f \in O(g)$, then $k \cdot f \in O(g)$, for any constant $k > 0$

- Special case: $O(k) = O(1)$ for all constants $k > 0$

The significance of this rule is that you can ignore any multiplicative constants in your equations when using big-O notation.

(3) **Addition**: If $f \in O(g)$ and $f' \in O(g')$, then $f + f' \in O(\max(g, g'))$

- Special case: if $f, f' \in O(g)$, then $f + f' \in O(g)$

This rule says that given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part.

(4) **Multiplication**: If $f \in O(g)$ and $f' \in O(g')$, then $f \cdot f' \in O(g \cdot g')$

The final rule is used to analyse simple loops in programs. If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function. The advantages and dangers of ignoring constants were discussed in the beginning of the previous section. Ignoring lower-order terms is reasonable when performing an asymptotic analysis. The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as n becomes larger. Thus, if $f(n) = 3n^4 + 5n^2$, then $f(n)$ is in $O(n^4)$. The n^2 term contributes relatively little to the total cost for large n.

From now on, we will use these simplifying rules when discussing the cost for a program or algorithm.

2.7.2 THE COMPLEXITY HIERARCHY

We can use the upper bound to define an ordering between complexity classes, where we write $O(f) \leq O(g)$ for $f \in O(g)$. Using this we can infer the following hierarchy of complexity classes:

$$O(1) < O(\log(\log n)) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < \cdots$$
$$\cdots < O(n^2) < O(n^2 \log n) < O(n^3) < O(n^k) < O(2^n) < O(10^n) < O(n!)$$

Note that we use strict equality here, and trust that you intuitively understand the difference between $\leq$ and $<$. One interesting consequence of asymptotic complexity is that the base of a logarithm becomes irrelevant:

$$O(\log_2(n)) = O(\ln(n)) = O(\log_{10}(n))$$

So we usually just write $O(\log n)$. The reason why the base is irrelevant is a direct consequence of the logarithm laws.

2.7.3 ANALYSING CODE FRAGMENTS

When we want to analyse the complexity of code fragments, the following three rules of thumb will get us very far:

- An **atomic operation** is always $O(1)$
- A **sequence**, $\langle p_1; p_2 \rangle$, has the complexity $O(\max(f_1, f_2))$, where $p_i \in O(f_i)$
- An **iteration**, $\langle \text{for } x \in A : p \rangle$, has the complexity $n \cdot O(f) = O(n \cdot f)$, where $n = |A|$ and $p \in O(f)$

Atomic operations Atomic operations are things like assigning a variable, looking up the value of an array index, or printing something. Not all atomic operations take the same time to execute, but we can assume that they are always constant.

If an operation takes constant time, $f(n) = k$, then $f \in O(1)$, so we can assume that all atomic operations are $O(1)$.

Sequence of operations A sequence of atomic operations, such as performing 10 assignments, is still constant (multiplying a constant with 10 is still constant). But what if we have a sequence of non-atomic operations?

E.g., suppose that we have the three operations $p_1 \in O(f_1)$, $p_2 \in O(f_2)$, and $p_3 \in O(f_3)$. The complexity of the sequence $\langle p_1; p_2; p_3 \rangle$ will then be sum of the parts, i.e.:

$$\langle p_1; p_2; p_3 \rangle \in O(f_1) + O(f_2) + O(f_3) = O(\max(f_1, f_2, f_3))$$

Loops and iterations What if we perform some operation $p \in O(f)$ for each element in an array A, what is the complexity? It depends on the size of the array: if we assume that the array has n elements, what is the complexity of the following loop?

$$\text{for } x \in A : p$$

The loop performs p once for every element in A, meaning that p will be executed n times. Therefore the complexity of a loop is:

$$\langle \text{for } x \in A : p \langle \in |A| \cdot O(f) = n \cdot O(f) = O(n \cdot f)$$

Note that p can be a complex operation, for example a loop itself. If p is a simple loop over A, with a constant-time operation in its body, then $p \in O(n)$. And then the outer loop $\langle \text{for } x \in A : p \rangle$ will be in $n \cdot O(n) = O(n^2)$.

2.7.4 EXAMPLES OF ALGORITHM ANALYSIS

Example: Simple for-loop

Consider a simple `for` loop.

```
sum = 0
for i in 0 .. n-1:
    sum = sum + n
```

The first line is $O(1)$. The `for` loop is repeated n times. The third line takes constant time so, by simplifying rule (4), the total cost for executing the two lines making up the `for` loop is $O(n)$. By rule (3), the cost of the entire code fragment is also $O(n)$.

Section 2.7.4

More examples online

More examples can be found in the online version of the book.

2.7.5 ADVANCED ALGORITHM ANALYSIS

The rules of thumb above do not always give the tightest possible complexity. In some (rare) cases the simple analysis might give a complexity of say $O(n \log n)$, but a more detailed analysis will give a tighter bound, such as $O(n)$. So, is there something wrong with the rules?

No, the rules are correct, and this is because the O notation gives an *upper bound*. Recall that every function $f \in O(n)$ is also in $O(n \log n)$, since $O(n) < O(n \log n)$.

Example: A nested loop with linear complexity

If we take the non-quadratic example above and just do a very small change, we get a completely different complexity.

```
sum2 = 0
k = 1
while k <= n:              // Do log n times.
    for j in 0 .. k-1:     // Do k times.
```

```
            sum2 = sum2 + 1
        k = k * 2
```

The only difference is that the inner loop runs k times, instead of n times. The rules of thumb gives us the same complexity as before, $O(n \log n)$, but a more careful analysis reveals a tighter bound.

The outer loop is executed $\log n + 1$ times, and the inner loop has cost k, which doubles each time. This can be expressed as the following summation, where n is assumed to be a power of two and again $k = 2^i$.

$$1 + 2 + 4 + \cdots + \log n = \sum_{i=0}^{\log n} 2^i$$

Now, as mentioned in Section 1.4, this summation has a closed form solution $2^{\log n+1} - 1 = 2n - 1$. So, the complexity of the code fragment above is actually linear, $O(n)$, and not linearithmic.

Note that this is an exception where the simple analysis rules do not give a tight enough bound. But in almost all cases the rules work fine, and when they don't it's usually only by a logarithmic factor. (And as we all know, logarithmic factors are not that big a deal when it comes to computational complexity.)

2.7.6 OTHER CONTROL STATEMENTS

What about other control statements? `While` loops are analysed in a manner similar to `for` loops. The cost of an `if` statement in the worst case is the greater of the costs for the `then` and `else` clauses. This is also true for the average case, assuming that the size of n does not affect the probability of executing one of the clauses (which is usually, but not necessarily, true). For `switch` statements, the worst-case cost is that of the most expensive branch. For subroutine calls, simply add the cost of executing the subroutine.

There are rare situations in which the probability for executing the various branches of an `if` or `switch` statement are functions of the input size. For example, for input of size n, the `then` clause of an `if` statement might be executed with probability $1/n$. An example would be an `if` statement that executes the `then` clause only for the smallest of n values. To perform an average-case analysis for such programs, we cannot simply count the cost of the `if` statement as being the cost of the more expensive branch. In such situations, the technique of amortised analysis can come to the rescue.

2.7.7 RECURSIVE FUNCTIONS

Determining the execution time of a recursive subroutine can be difficult. The running time for a recursive subroutine is typically best expressed by a recurrence relation. For example, the recursive factorial function calls itself with a value one less than its input value. The result of this recursive call is then multiplied by the input value, which takes constant time. Thus, the cost of the factorial function, if we wish to measure cost in terms of the number of multiplication operations, is one more than the number of multiplications made by the recursive call on the smaller input. Because the base case does no multiplications, its cost is constant. Thus, the running time for this function can be expressed as

$$\begin{aligned} T(n) &= T(n-1)+1, \text{ for } n>1 \\ T(1) &= 1 \end{aligned}$$

The closed-form solution for this recurrence relation is $O(n)$. Recurrence relations are discussed further in Section 7.4.

2.8 Case study: Analysing binary search

Now we will discuss the algorithmic complexity of the two search algorithms that we introduced in Section 1.6.

In Section 2.3.1, we deduced that the running time for linear search on an array if size n is $O(n)$. If we assume that the value is equally likely to appear in any location, this complexity is both the average case and the worst case. (The best case is constant, $O(1)$, and occurs when the searched value occurs first in the array. But as we already discussed we are rarely interested in the best case.)

But what is the complexity of the binary search algorithm?

2.8.1 COMPLEXITY OF BINARY SEARCH

To find the worst-case cost of binary search, we can model the running time as a recurrence and then find the closed-form solution. Each recursive call to `binarySearch` cuts the size of the array approximately in half, so we can model the worst-case cost as follows, assuming for simplicity that n is a power of two.

$$\begin{aligned} O(n) &= O(n/2)+1 \\ O(1) &= 1 \end{aligned}$$

If we expand the recurrence, we will find that we can do so only $\log n$ times before we reach the base case, and each expansion adds one to the cost.

- For a problem of size n, we have 1 unit of work plus the amount of work required for one subproblem of size $n/2$: $O(n) = 1 + O(n/2)$
- For a problem of size $n/2$, we have 1 unit of work plus the amount of work required for one subproblem of size $n/4$: $O(n) = 1 + (1 + O(n/4))$
- For a problem of size $n/4$, we have 1 unit of work plus the amount of work required for one subproblem of size $n/8$: $O(n) = 1 + (1 + (1 + O(n/8)))$
- …etc, until we reach a subproblem of size 1: $O(n) = \underbrace{1 + (1 + (1 + (\ldots)))}_{\log n \text{ levels}}$

Thus, the closed form solution of $O(n) = O(n/2) + 1$ can be modeled by the summation

$$\sum_{i=0}^{\log n} 1 \in O(\log n)$$

Comparing sequential search to binary search, we see that as n grows, the $O(n)$ running time for sequential search in the average and worst cases quickly becomes much greater than the $O(\log n)$ running time for binary search. Taken in isolation, binary search appears to be much more efficient than sequential search. This is despite the fact that the constant factor for binary search is greater than that for sequential search, because the calculation for the next search position in binary search is more expensive than just incrementing the current position, as sequential search does.

However, binary search comes with a precondition: the array must be sorted. And sorting an array is a time-consuming operation – in fact it is $O(n \log n)$ in the worst case, as we will see in chapter 4. So there's a tradeoff here – to be able to search the array efficiently we need to keep it sorted. This is not much of a problem if this is something we only have to do once, but it can be very costly if the array changes because we insert and delete elements. Only in the context of the complete problem to be solved can we know whether the advantage outweighs the disadvantage.

So, if we want to maintain a searchable collection which changes over time, a sorted array is not a good choice. But an unsorted array isn't either – instead we should use smarter data structures such as search trees (chapter 11) or hash tables (chapter 12).

2.9 Code tuning

Code tuning involves optimising specific parts of a program to improve performance, often by reducing runtime or memory usage. While it doesn't change

the algorithm's complexity, it can lead to significant speedups, sometimes by a factor of five to ten. Effective tuning focuses on the most time-consuming parts of the code and is best guided by empirical analysis and profiling tools. The full text can be read online.

Section 2.9

Read section online

2.10 Empirical analysis

Asymptotic algorithm analysis is a method for estimating how an algorithm's performance scales with input size. While it helps identify efficient algorithms, it has limitations such as inaccuracy for small inputs and difficulty modeling complex problems. The full text can be read online.

Section 2.10

Read section online

2.11 Review questions

This final section contains some review questions about the contents of this chapter.

Section 2.11

Answer quiz online

Sorting, part 1: Simple algorithms

We have seen that, when an array is sorted in ascending order, *binary search* can be used to find items in it efficiently. But what about when we have a collection of data that is not in any order? If we will often need to search for items in the data, it makes sense to *sort the data* first. In this chapter we will study algorithms for sorting arrays in ascending order.

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort customer records by zip code so that when we print an advertisement we can then mail them more cheaply. We might use sorting to help an algorithm to solve some other problem. For example, Kruskal's algorithm to find minimum spanning trees must sort the edges of a graph by their lengths before it can process them.

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. For example, a natural way to sort your cards in a bridge hand is to go from left to right, and place each card in turn in its correct position relative to the other cards that you have already sorted. This is the idea behind Insertion sort. Other sorting algorithms are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. For example, no normal person would use Quicksort to order a pile of bills by date, even though Quicksort is one of the standard sorting algorithms of choice for most software libraries. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special-purpose applications.

Sorting can be divided into two kinds depending on how we compare the items in the list:

Natural sorting The items have some kind of natural order. For example, sorting a list of words in alphabetical order, or sorting a list of numbers.

Key-based sorting Here, each item has a *key*, and we want to sort the items so that the keys come in order. For example, sorting a list of towns by population, or sorting a list of persons by their age.

Note that if we sort according to a *key*, it doesn't have to be explicitly stored in the object, but can instead be calculated on demand. E.g., if we want to sort a list of words case-insensitively, we can use a lower-case transformation when doing the comparisons. This is usually done by a comparator (in Java), or by a key function (in Python).

The following two chapters cover several standard algorithms appropriate for sorting a collection of records. In these chapters we concentrate on *natural sorting*, but all the algorithms work just as well for *key-based sorting* – and we trust that you are a mature enough programmer to be able to infer how to do this. The first chapter discusses three simple, but relatively slow, algorithms that require quadratic time in the size of the array. The following chapter then presents two algorithms with considerably better performance, with linearithmic worst-case or average-case running time.

3.1 Terminology and notation

Formally, the *sorting problem* is to arrange a list of elements $a_1, a_2, \ldots, a_n$ into any order s such that $a_{s_1} \leq a_{s_2} \leq \cdots \leq a_{s_n}$. In other words, the sorting problem is to arrange a set of elements so that they are in non-decreasing order.

Note that the definition above is for *natural sorting*. If we instead are interested in the more general problem of *key-based sorting*, the definition becomes slightly more complicated: The (key-based) *sorting problem* is to arrange the list into any order s such that $a_{s_1}, a_{s_2}, \ldots, a_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \cdots \leq k_{s_n}$.

3.1.1 COMPARING ALGORITHMS

When comparing two sorting algorithms, the simplest approach would be to program both and measure their running times. This is an example of *empirical comparison* (see Section 3.8). However, doing fair empirical comparisons can be tricky because the running time for many sorting algorithms depends on specifics of the input values. The number of records, the size of the keys and the records, the allowable range of the key values, and the amount by which

the input records are "out of order" can all greatly affect the relative running times for sorting algorithms.

When analysing sorting algorithms, it is traditional to measure the cost by *counting the number of comparisons* made between keys. This measure is usually closely related to the actual running time for the algorithm and has the advantage of being machine and data-type independent. However, in some cases records might be so large that their physical movement might take a significant fraction of the total running time. If so, it might be appropriate to measure the cost by counting *the number of swap operations* performed by the algorithm.

In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved. However, some special situations "change the rules" for comparing sorting algorithms. For example, an application with records or keys having widely varying length (such as sorting a sequence of variable length strings) cannot expect all comparisons to cost roughly the same. Not only do such situations require special measures for analysis, they also will usually benefit from a special-purpose sorting technique.

Other applications require that a small number of records be sorted, but that the sort be performed frequently. An example would be an application that repeatedly sorts groups of five numbers. In such cases, asymptotic analysis is usually of not much help. Instead it will be important to reduce the constant factors that are ignored by complexity analysis. Then we might very well find that the best algorithm can be one of the very simple "slow" algorithms, such as Insertion sort.

Finally, some situations require that a sorting algorithm use as little memory as possible. We will call attention to sorting algorithms that require significant extra memory beyond the input array.

3.1.2 TERMINOLOGY

Here are some important terminology which we can use to categorise different algorithms.

Stability As defined, the sorting problem allows input with two or more records that have the same key value. But it does not specify how these should be ordered, which means that there can be several possible solutions to the problem. Sometimes it is desirable to maintain the initial ordering between two elements that compare equal. A sorting algorithm is said to be *stable* if it does not change the relative ordering of records with identical key values.

Many, but not all, of the sorting algorithms presented in this book are stable, or can be made stable with minor changes.

Adaptivity An *adaptive* sorting algorithm can take advantage of the existing order in the input. In general this means that the algorithm runs faster if the list is almost sorted, compared to if the list is completely random. The classic example of an adaptive sorting algorithm is Insertion sort.

In-place An *in-place* sorting algorithm modifies the input array directly and does not build a new array for the sorted result. Usually one also requires that the algorithm does not allocate too much extra space while operating, where "not too much" can mean at most logarithmic extra memory in the size of the array. One sorting algorithm which is *not* in-place is Mergesort, while most other algorithms are.

3.2 Comparing values

If we want to sort some things, we have to be able to compare them, to decide which one is bigger. How do we compare two things? If all that we wanted to sort or search for was simple integer values, this would not be an interesting question. We can just use standard comparison operators like "<" or ">". Even if we wanted to sort strings, most programming languages give us built-in functions for comparing strings alphabetically. But we do not usually want to sort just integers or strings in a data structure. Often we want to sort records, where a record is made up of multiple values, such as a name, an address, and a phone number. In that case, how can we "compare" records to decide which one is "smaller"? We cannot just use "<" to compare the records! Nearly always in this situation, we are actually interested in sorting the records based on the values of one particular field used to represent the record, which itself is something simple like an integer or a string. This field is referred to as the key for the record.

Likewise, if we want to search for a given record in a database, how should we describe what we are looking for? A database record could simply be a number, or it could be quite complicated, such as a payroll record with many fields of varying types. We do not want to describe what we are looking for by detailing and matching the entire contents of the record. If we knew everything about the record already, we probably would not need to look for it. Instead, we typically define what record we want in terms of a key value. For example, if searching for payroll records, we might wish to search for the record that matches a particular ID number. In this example the ID number is the search key.

Finally, it is very often the case that we want to compare *virtual keys*, that is

keys that are not explicitly stored in the record, but calculated on demand. One simple example is if we want to sort a list of strings *case-insensitively*, ignoring if a letter is uppercase or lowercase. A more complex example is to sort a list if Unicode strings according to a certain language locale.

3.2.1 TWO MAIN APPROACHES TO COMPARING VALUES

When we compare two elements, there are *three* possible outcomes – the first element can be *smaller*, or *larger*, or *equal to*, the second element.

Most programming languages does not have any atomic datatype with three values, so different languages have implemented different solutions to how to compare values. There are two main approaches in how a programming language have solved the comparison problem:

The Python way In Python, each comparison operator ($<$, $>$, $=$, …) is implemented separately. This means that you can write $a < b$, $a = b$, $a > b$, etc., in a way that you are used to think about comparison. The disadvantage is that sometimes you have to perform *two* comparisons between two values: first you have to check if $a < b$, and then if $a = b$. Depending on how your elements are structured, this can lead to some duplicate work (although Python is quite good at optimising the code so that there will be no penalty).

The Java way In Java, there is one comparison operator, `compare` (or `compareTo`), which returns an integer k. If $k < 0$ then $a < b$, if $k > 0$ then $a > b$, and if $k = 0$ then $a = b$. The main advantage with this approach is that you do not risk duplicating work, but on the other hand the can code become slightly less readable.

In this book we will usually use the Python way when describing algorithms. Not because we think that is a better way of writing algorithms, but because the pseudocode becomes easier to read.

3.2.2 NATURAL VS KEY-BASED COMPARISON

As we already mentioned in the chapter introduction, we will usually just assume that you want to use the natural order when comparing objects. However, in real life it is much more common that you need to compare values by their keys.

Modern high-level languages (including Python and Java) have several different ways to accomplish key-based comparison, and we encourage you to find out how this is done in your favourite language. It can be done via special methods on objects, or by special classes, or by supplying a *key function*, or several other ways.

3.3 Overview of algorithms

In the sections following we will introduce three basic sorting algorithms. Here we will give very short descriptions of each of them, and later go through them in more detail.

For these short descriptions, assume that we want to sort N books in a bookshelf. It doesn't matter what we want to sort the books by – it could be by author, or title, or perhaps height. We only need to know how to compare two books, where we say that the "smaller" book should come before the "larger" in the final ordered bookshelf.

3.3.1 BUBBLE SORT OVERVIEW

The first algorithm, *bubble sort*, is also the shortest to describe:

- As long as there are two adjacent books out of order, swap them.

Note that this description is too unspecified to be a real algorithm – in particular, we have to know in which order we should look at adjacent books. This can be done in many different ways, but the commonest implementation is to go from left to right and compare each pair of adjacent books. After the first iteration we have moved the largest book to the far end of the bookshelf. The next iteration will move the second largest book to its right location (next to the largest book), and so on. Repeat this left-to-right pass until the list is sorted. It's perhaps not obvious, but in fact we only have to do this at most N times.

3.3.2 SELECTION SORT OVERVIEW

Now, assume that we instead empty the bookshelf and put all the books on the floor, so that we can have a good overview of them. This gives the idea for the *selection sort* algorithm:

- Repeat until there are no more books on the floor:
  - Select the smallest book on the floor and put it to the right of the books that are already in the shelf.

This description suggests that selection sort is not an in-place algorithm, because we first move all the books to a new place, meaning that we have to have enough extra floor space for all the books. But the algorithm can easily be made in-place, and we will describe that in a later section.

3.3.3 INSERTION SORT OVERVIEW

Finally, assume that we don't spread out all books on the floor, but instead put them in a single pile. Now we can proceed like as *insertion sort*:

- Repeat until there are no more books in the pile:
  - Take the topmost book in the pile and insert it in the right position among the books that are on the bookshelf.

Just as for selection sort, this description suggests that it is not in-place, but there is a simple in-place version, which we will introduce later.

3.3.4 SUMMARY

So, what is the complexity of these three algorithms? All of them has an outer loop that is repeated N times, so how long time does their inner loops take?

- *Bubble sort*: one left-to-right pass iterates through the whole shelf, so it's linear $O(N)$.
- *Selection sort*: finding the smallest book in an unsorted collection is linear $O(N)$.
- *Insertion sort*: inserting one book in the correct position in a sorted list is linear $O(N)$.

Therefore, the total complexity of each of the algorithms is quadratic, $O(N^2)$.

3.4 Bubble sort

Our first sorting algorithm is called Bubble sort. Bubble sort is often the first sorting algorithm that you learn, because it is relatively easy to understand and implement. However, it is rather slow, even compared to the other quadratic sorting algorithms that we will introduce in the next sections – Selection sort and Insertion sort. It's not even particularly intuitive – nobody is going to come naturally to bubble sort as a way to sort their bookshelf, their Bridge hand or their pile of bills, like they might with Insertion sort or Selection sort.

Bubble sort consists of a simple double `for` loop. The inner `for` loop moves through the array from left to right, comparing adjacent elements. If an element is greater than its right neighbour, then the two are swapped. Once the largest element is encountered, this process will cause it to "bubble" up to the right of the array (which is where Bubble sort gets its name). The second pass through the array repeats this process. However, because we know that the largest element

already reached the right of the array on the first pass, there is no need to compare the rightmost two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less element toward the end than in the preceding pass. Here is an implementation:

```
function bubbleSort(A):
    N = A.size
    for i in 0 .. N-2:
        // Bubble up the i'th element
        for j in 1 .. N-i-1:
            if A[j-1] > A[j]:
                swap(A, j-1, j)
```

3.4.1 BUBBLE SORT ANALYSIS

We have a nested for loop, where the inner loop depends on the loop variable of the outer loop.

- The outer loop is iterated $N-1$ times in total.
- In iteration i, the number of comparisons made by the inner loop is always $N-i-1$.

So the total number of iterations is

$$(n-1) + (n-2) + \cdots + 1 \quad = \quad \sum_{i=1}^{n-1} i$$

And this sum has the value $N(N-1)/2$, which means the runtime complexity is *quadratic*, $O(N^2)$. Note that this is regardless of how the initial array looks like, so bubble sort has the same best- and worst-case complexity.

The number of swaps required depends on how often an element is less than the one immediately preceding it in the array. In the worst case this will happen in every single comparison, leading to $O(N^2)$ number of swaps. If we assume that the initial array is random, we can expect that about half of the comparisons will lead to a swap. So the average case number of swaps is also quadratic, $O(N^2)$. However, if the initial array is already sorted we don't have to perform any swaps at all, so in the best case the number of swaps is constant $O(1)$. (But recall that the best case is not something we should rely upon.)

3.5 Selection sort

Let's say you have a large pile of books that you want to put in your bookshelf, in alphabetical order by author's surname. How would you go about? One natural

way to do handle this is to look through the pile until you find the first book (say, by an author named *Ajvide*), and put that first in the bookshelf. Then you look through the remaining pile until you find the second book (written by *Bengtsdotter*), and add that behind *Ajvide*. Then find the third book (by *Cassler*), and add behind *Bengtsdotter*. Proceed through the shrinking pile of books to select the next one in order until you are done. This is the inspiration for our next sorting algorithm, called Selection sort.

In the description above the books are not in the shelf from the start, which makes the algorithm not in-place. But it is easy to turn this into an in-place algorithm, where all books are in the shelf from the start. We just have to remember an invisible separator between the sorted books (on the left) and the still-unsorted books (on the right). Whenever we have found the next book to put in place, we *swap* it with the book that is in the way.

The i'th pass of Selection sort "selects" the i'th smallest element in the array, placing it at position i in the array. In other words, Selection sort first finds the smallest element in an unsorted list, then the next smallest, and so on. Its unique feature is that there are few swaps, much fewer than Bubble sort. To find the next-smallest element we have to search through the entire unsorted portion of the array, but only one swap is required to put the element into place.

Here is an implementation for Selection sort.

```
function selectionSort(A):
    N = A.size
    for i in 0 .. N-1:              // Select i'th smallest element
        minIndex = i                // Current smallest index
        for j in i+1 .. N-1:        // Find the smallest value
            if A[j] > A[minIndex]:  // Found something smaller
                minIndex = j        // Remember smaller index
        swap(A, i, minIndex)        // Put it into place
```

Any algorithm can be written in slightly different ways. For example, we could have written Selection sort to find the largest element and put it at the end of the array, then the next smallest, and so on. That version of Selection sort would behave very similar to our Bubble sort implementation, except that rather than repeatedly swapping adjacent values to get the next-largest element into place, it instead remembers the position of the element to be selected and does one swap at the end.

3.5.1 SELECTION SORT ANALYSIS

We have a nested for loop, where the inner loop depends on the loop variable of the outer loop.

- The outer loop is iterated N times in total.
- In iteration i, the number of comparisons made by the inner loop is always $N - i - 1$.

As you might notice, this is exactly the same as the number of comparisons bubble sort makes. So, selection sort makes $N(N-1)/2$ comparisons, which is quadratic, $O(N^2)$.

The advantage compared to bubble sort is that selection sort makes a lot fewer swaps. For each outer iteration it only makes one swap, so the total number of swaps will be $N - 1$ (we get the last element in place "for free"). So, selection sort makes a linear number of swaps, $O(N)$.

3.6 Insertion sort

Consider again the problem of sorting a pile of books. Another intuitive approach might be to pick up the two topmost books in the pile and put them in order in the bookshelf. Then you take another book from the pile and put it in the bookshelf, in the correct position with respect to the first two, and so on. As you take each book, you would add it in the bookshelf in the correct position to always keep the shelf sorted. This simple approach is the inspiration for our third sorting algorithm, called Insertion sort.

Just as for selection sort, the description above is not in-place. But just as for selection sort, it's relatively easy to turn it into an in-place algorithm, by remembering an invisible separator between the sorted books (on the left) and the still-unsorted books (on the right).

Insertion sort iterates through a list of elements. For each iteration, the current element is inserted in turn at the correct position within a sorted list composed of those elements already processed. Here is an implementation. The input is an array named A that stores N elements.

```
function insertionSort(A):
    N = A.size
    for i in 1 .. N-1:
        // Move the i'th element to its correct position.
        j = i
        while j > 0 and A[j] < A[j-1]:
```

```
swap(A, j, j-1)
j = j-1
```

3.6.1 INSERTION SORT ANALYSIS

Just as for the previous sorting algorithms, we have a nested for loop, where the inner loop depends on the loop variable of the outer loop.

- The outer loop is iterated $N-1$ times in total.
- The inner loop is harder to analyse since it depends on how many elements in positions $0, \ldots, i-1$ are smaller than the element in position i.
  - in the absolute worst case, we always have to move the element to the front of the list, so the number of comparisons will be $i-1$
  - in the best case, the element is already in place, and then we only need one comparison

Therefore, the worst case complexity of insertion sort is $\sum_0^N i$, which is quadratic, $O(N^2)$. In the best case – when the list is already sorted – the complexity is instead linear, $O(N)$, because we only have to do one comparison per iteration.

While the best case is significantly faster than the average and worst cases, the average and worst cases are usually more reliable indicators of the "typical" running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion sort might be a good idea if we know that the disordering is slight. And even when the input is not perfectly sorted, Insertion sort's cost goes up in proportion to the number of inversions. So a "nearly sorted" list will always be cheap to sort with Insertion sort. An example of an algorithm that take advantage of Insertion sort's near-best-case running time is Shellsort.

Counting comparisons or swaps yields similar results. Each time through the inner `for` loop yields both a comparison and a swap, except the last (i.e., the comparison that fails the inner `for` loop's test), which has no swap. Thus, the number of swaps for the entire sort operation is $N-1$ less than the number of comparisons. This is o in the best case, and $O(N^2)$ in the average and worst cases.

Later we will see algorithms whose growth rate is much better than $O(N^2)$. Thus for larger arrays, Insertion sort will not be so good a performer as other algorithms. So Insertion sort is not the best sorting algorithm to use in most situations. But there are special situations where it is ideal. We already know that Insertion sort works great when the input is sorted or nearly so. Another good time to use Insertion sort is when the array is very small, since Insertion sort is

so simple. The algorithms that have better asymptotic growth rates tend to be more complicated, which leads to larger constant factors in their running time. That means they typically need fewer comparisons for larger arrays, but they cost more per comparison. This observation might not seem that helpful, since even an algorithm with high cost per comparison will be fast on small input sizes. But there are times when we might need to do many, many sorts on very small arrays. You should spend some time right now trying to think of a situation where you will need to sort many small arrays. Actually, it happens a lot.

3.7 Summary analysis of basic sorting algorithms

How can we categorise our three sorting algorithms according to the terminology that we introduced in Section 3.1?

In-place All three algorithms are *in-place*, because we modify the array and only make use of a constant number of additional variables.

Stability Both bubble sort and insertion sort are *stable*. They only swap adjacent elements, and they will never swap two equal elements. Therefore the relative order of equal elements will be preserved.

However, insertion sort is not stable, and this is because it can swap over long distances. For example, if we insertion sort the array [2,2,1], then the first swap (putting 1 into the first position) will change the relative order between the first and the second 2.

Adaptivity Insertion sort is *adaptive*, because its best-case complexity is better than its worst-case. If the list is almost sorted, insertion sort is way much faster than if the list is completely unsorted.

Bubble sort and selection sort however, are not adaptive – they are always quadratic regardless of the input array.

Here is a summary table of the categorisations.

	Bubble	Selection	Insertion
In place?	yes	yes	yes
Stable?	yes	no	yes
Adaptive?	no	no	yes

Here is a summary table for the cost of Bubble sort, Selection sort, and Insertion sort, in terms of their required number of comparisons and swaps in the best and worst cases. The running time for each of these sorts is $O(n^2)$ in the worst case.

	Bubble	Selection	Insertion
Comparisons:			
Best case	$O(n^2)$	$O(n^2)$	$O(n)$
Worst case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Swaps:			
Best case	0	$O(n)$	0
Worst case	$O(n^2)$	$O(n)$	$O(n^2)$

3.7.1 INVERSIONS AND THE REASON FOR THE QUADRATIC BEHAVIOUR

The remaining sorting algorithms presented in the next chapter are significantly better than these three under typical conditions. But before continuing on,

These three sorting algorithms are all quadratic, but can we say something about *why* they are so slow? The crucial bottleneck is that only *adjacent* records are compared, and swapped (for Insertion and Bubble sort). To analyse this we first need to define the concept of *inversion*.

An *inversion* occurs when there are two elements in an array that come in the wrong order. Formally, if $A[i] > A[j]$ for array indices $i < j$, then there is an inversion between i and j (or between $A[i]$ and $A[j]$ if that is unambiguous). For example, in the array [A,B,X,C,D] there are inversions between indices 2 and 3 (elements X and C are out of order), and between indices 2 and 4 (elements X and D).

The number of inversions in an array is one measure of how sorted the array is (but not the only such measure). The most unsorted array according to this definition is reversely sorted, because then all pairs of indices are inversions. So, the maximum number of inversions is the number of pairs, which is $n(n-1)/2$.

Now, assume that we have an array, and we swap two adjacent elements (that are out of order, i.e., an inversion). This will reduce the number of inversions with at most 1, because all other inversions in the array will still be inversions. Therefore, any algorithm which can only swap *adjacent* elements has to perform at least as many swaps as there are inversions. And since there are $n(n-1)/2 = O(n^2)$ inversions in the worst case, any such algorithm will at least be quadratic, $O(n^2)$. This includes Insertion sort and Bubble sort.

But how about Selection sort? It does not swap adjacent elements, so is there perhaps a possibility that we can optimise it to be better than quadratic? – Unfortunately not, and this is because Selection sort compares only adjacent elements. Assume that we swap two non-adjacent elements, that have d elements in between themselves. If we are extremely lucky this single swap can reduce

the number of inversions by $2d$. However, our assumption was that we can only *compare adjacent* elements, and to be able to know which two indices to swap we have to compare all adjacent elements in between them. So we need to perform at least $d + 1$ comparisons to decide which indices to swap. Therefore, we need $d + 1 = O(d)$ comparisons to reduce the number of inversions by $2d = O(d)$. And since there are $O(n^2)$ inversions in the worst case, we need at least $O(n^2)$ comparisons.

Therefore, all sorting algorithms that can only compare adjacent elements (or swap adjacent elements) are doomed to be quadratic in the worst case, $O(n^2)$. This includes the algorithms we have presented so far, and numerous other.

In the next chapter we will present sorting algorithms that are significantly better than these three under typical conditions. How can they circumvent the quadratic behaviour? – They compare and swap non-adjacent elements (and they do it in a smart way).

Section 3.8

Read the rest online

3.8 Empirical analysis and code tuning

Since sorting is such an important application, it is natural for programmers to want to optimise their sorting code to run faster. Of course all quadratic sorts (Insertion sort, Bubble sort and Selection sort) are relatively slow. Each has (as the name "quadratic" suggests) $O(n^2)$ worst case running time. The best way to speed them up is to find a better sorting algorithm. Nonetheless, there have been many suggestions given over the years about how to speed up one or another of these particular algorithms. There are useful lessons to be learned about code tuning by seeing which of these ideas actually turn out to give better performance. It is also interesting to see the relative performance of the three algorithms, as well as how various programming languages compare.

Section 3.9

Answer quiz online

3.9 Review questions

This final section contains some review questions about the contents of this chapter.

CHAPTER 4

Sorting, part 2: Divide-and-conquer algorithms

In the previous chapter we presented three simple and relatively slow sorting algorithms, with quadratic runtime behaviour. Now we will introduce two algorithms with considerably better performance, with linearithmic worst-case or average-case running time: Mergesort and Quicksort. In addition we will briefly discuss some special-purpose sorting algorithms, such as Radix sort.

Both these algorithms make use of a basic strategy for algorithm design, which is called divide and conquer. The basic idea is to divide a big problem (how to sort a big array) into subproblems that can be solved independently (how to sort two smaller arrays). The main difference between the algorithms is how they do the dividing: Mergesort divides the array in two equal-sized halves, while Quicksort divides the array into the big values and the small values. Radix sort in turn divides the problem by working on one digit of the key at a time.

We will also make use of different algorithm analysis techniques. Quicksort illustrates that it is possible for an algorithm to have an average case whose growth rate is significantly smaller than its worst case. We can use code tuning to improve these algorithms, by falling back to a simpler algorithm (such as Insertion sort) when the subarray is small enough.

4.1 Recursion and divide-and-conquer

An algorithm (or a function in a computer program) is recursive if it invokes itself to do part of its work.

4.1.1 RECURSION

Recursion makes it possible to solve complex problems using programs that are concise, easily understood, and algorithmically efficient. Recursion is the process of solving a large problem by reducing it to one or more sub-problems which are identical in structure to the original problem and somewhat simpler to solve. Once the original subdivision has been made, the sub-problems divided into

new ones which are even less complex. Eventually, the sub-problems become so simple that they can be then solved without further subdivision. Ultimately, the complete solution is obtained by reassembling the solved components.

For a recursive approach to be successful, the recursive "call to itself" must be on a smaller problem than the one originally attempted. In general, a recursive algorithm must have two parts:

1. The base case, which handles a simple input that can be solved without resorting to a recursive call, and
2. The recursive part which contains one or more recursive calls to the algorithm. In every recursive call, the parameters must be in some sense "closer" to the base case than those of the original call.

Recursion has no counterpart in everyday, physical-world problem solving. The concept can be difficult to grasp because it requires you to think about problems in a new way. When first learning recursion, it is common for people to think a lot about the recursive process. We will spend some time in this section going over the details for how recursion works. But when writing recursive functions, it is best to stop thinking about how the recursion works beyond the recursive call. You should adopt the attitude that the sub-problems will take care of themselves, that when you call the function recursively it will return the right answer. You just worry about the base cases and how to recombine the sub-problems.

Newcomers who are unfamiliar with recursion often find it hard to accept that it is used primarily as a tool for simplifying the design and description of algorithms. A recursive algorithm does not always yield the most efficient computer program for solving the problem because recursion involves function calls, which are typically more expensive than other alternatives such as a while loop. However, the recursive approach usually provides an algorithm that is reasonably efficient. If necessary, the clear, recursive solution can later be modified to yield a faster implementation.

Imagine that someone in a movie theater asks you what row you're sitting in. You don't want to count, so you ask the person in front of you what row they are sitting in, knowing that they will tell you a number one less than your row number. The person in front could ask the person in front of them. This will keep happening until word reaches the front row and it is easy to respond: "I'm in row 1!" From there, the correct message (incremented by one each row) will eventually make it's way back to the person who asked.

Example: The Fibonacci sequence

Here is an example of a function that is naturally written using recursion. The Fibonacci sequence is the series of numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Any number in the sequence is found by adding up the two numbers before it, and the first two numbers in the sequence are both 1. Mathematically, the nth Fibonacci number is calculated recursively like this:

$$f(0) = f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

The first row is the *recursive case*, and the second row defines the two *base cases*. This mathematical definition is easily translated into pseudocode:

```
function fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

4.1.2 DIVIDE AND CONQUER

Solving a "big" problem recursively means to solve one or more smaller versions of the problem, and using those solutions of the smaller problems to solve the "big" problem.

Sometimes it is possible to divide a problem into *more than one* smaller sub-problems. This is the basis for *divide-and-conquer* algorithms, which can be found in several different areas, such as integer and matrix multiplication, computational geometry, and syntactical parsing. In this chapter we will how divide-and-conquer can be used to derive two efficient sorting algorithms, Mergesort and Quicksort.

Both Mergesort and Quicksort splits the array into two sub-arrays, which can be sorted independently of each other. The key point to making the algorithms efficient is that the sub-arrays must be (approximately) the same size, so that the problem sizes are (approximately) halved in each iteration.

4.2 Mergesort

A natural approach to problem solving is divide and conquer. To use divide and conquer when sorting, we might consider breaking the list to be sorted into

pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind Mergesort.

Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Unfortunately, even though it is based on a simple concept, it is relatively difficult to implement in practice. Here is a pseudocode sketch of Mergesort:

```
function mergeSort(A):
    if A.size <= 1:
        return
    L1 = half of A
    L2 = other half of A
    mergeSort(L1)
    mergeSort(L2)
    merge(L1, L2)
```

Merge The hardest step to understand about Mergesort is the merge function. The merge function starts by examining the first record of each sublist and picks the smaller value as the smallest record overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front records of the sublists and continually appending the smaller to the output list until no more input records remain.

Here is pseudocode for merging two lists:

```
function merge(L1, L2):
    answer = new empty list
    while L1 and L2 are nonempty:
        x = first element of L1
        y = first element of L2
        if x <= y:
            append x to answer
            remove x from L1
        else:
            append y to answer
            remove y from L2
    // Now only one of L1 and L2 is nonempty, so append its remaining elements
    append all elements of L1 or L2 to answer
    return answer
```

4.2.1 COMPLEXITY ANALYSIS

Consider repeatedly splitting an array of N elements, where $N = 2^k$ is a power of 2:

- The first level consists of the full array:
  - Splitting the full array into two halves requires N units of work.
- The second level consists of two halves, with $N/2$ elements each:
  - Splitting one of the halves into two requires $N/2$ units of work.
  - Splitting the other half into two requires $N/2$ units of work.
  - In total $2 \times N/2 = N$ units of work.
- The third level consists of four quarters, with $N/4$ elements each:
  - Splitting each of the quarters requires $N/4$ units of work.
  - In total $4 \times N/4 = N$ units of work.
- …
- …until level k, which consists of $N/2$ sub-arrays, each of which contains only 2 elements.
  - Splitting each of the pairs requires 2 units of work.
  - In total $N/2 \times 2 = N$ units of work.

After a sub-array has been split, the next level is called recursively which returns the sorted splits. Then we have to merge them before returning the sorted sub-array to the previous level:

- Level k consists of $N/2$ pairs of singleton arrays.
  - To merge a pair we have to look at each of the values, so it requires 2 units of work.
  - In total $N/2 \times 2 = N$ units of work.
- Level $k - 1$ consists of $N/4$ pairs of sorted 2-element arrays.
  - Merging two 2-elements arrays requires 4 units of work.
  - In total $N/4 \times 4 = N$ units.
- …
- Level two consists of 2 pairs of sorted $N/4$-element arrays.
  - Merging two $N/4$-element arrays requires $N/2$ units of work.
  - In total $2 \times N/2 = N$.
- And finally the first level constists of one pair of sorted $N/2$-element arrays.
  - Merging these two arrays requires N units of work.

In summary, each level spends $O(N)$ time, and there are $k = \log N$ levels. So the total running time of mergesort is $O(N \log N)$. Note that this cost is unaffected by the relative order of the values being sorted, thus this analysis holds for the best, average, and worst cases.

4.3 Implementing Mergesort

Implementing Mergesort presents some technical difficulties. The pseudocode in the last section is quite vague and we have to figure out how to make it work for arrays.

First, splitting an input array into two subarrays is easy. We don't even have to copy any elements, but we can use the same idea as for binary search: use `left` and `right` indices to refer to a subarray. To split this subarray into two halves, we just calculate the middle index, `mid = (left+right)/2`.

The main function for sorting a subarray can now be written like this:

```
// Sort the subarray A[left .. right]
function mergeSort(A, left, right):
    if left >= right:               // Base case: Subarray length is ≤ 1
        return
    mid = int((left + right) / 2)  // The midpoint is where the second half starts
    mergeSort(A, left, mid-1)       // Mergesort first half
    mergeSort(A, mid, right)        // Mergesort second half
    merge(A, left, mid, right)      // Merge the two sorted halves
```

The initial call would be `mergeSort(A,0,A.size-1)`, which sorts the whole array `A`.

The difficulty comes when we want to merge the two sorted subarrays. This cannot be done in-place (or rather, it is very very complicated to do it in-place). So we need an auxiliary array which we can merge the elements into.

```
// Merge the sorted subarrays A[left .. mid-1] and A[mid .. right]
function merge(A, left, mid, right):
    temp = new Array
    // Merge the two sublists into the auxiliary array
    i1 = left
    i2 = mid
    for i in left .. right:
        if i2 > right:              // Right sublist exhausted
            temp[i] = A[i1]
            i1 = i1 + 1
        else if i1 > mid:           // Left sublist exhausted
            temp[i] = A[i2]
```

```
            i2 = i2 + 1
        else if A[i1] <= A[i2]:   // Get smaller value
            temp[i] = A[i1]
            i1 = i1 + 1
        else:
            temp[i] = A[i2]
            i2 = i2 + 1
    // Copy the elements back from the auxiliary array
    for i in left .. right:
        A[i] = temp[i]
```

Notice that the merge function will create a new auxiliary array every time it is called. This is quite inefficient, because it takes some time to allocate memory for a new array, which can be destroyed directly when merge is finished. One simple optimisation is to create one single auxiliary array before the first call, and reuse this array in all invocations of merge. The only thing we would have to do is to add an extra argument to `mergeSort` and `merge`, for the reference to the auxiliary array. Then we can create a wrapper function that takes care of the initialisation, and makes the first recursive call:

```
function mergeSort(A):
    temp = new Array(A.size)
    mergeSort(A, temp, 0, A.size-1)
```

4.4 Quicksort

While Mergesort uses the most obvious form of divide and conquer (split the list in half then sort the halves), this is not the only way that we can break down the sorting problem. We saw that doing the merge step for Mergesort when using an array implementation is not so easy. So perhaps a different divide and conquer strategy might turn out to be more efficient?

Quicksort is aptly named because, when properly implemented, it is one of the fastest known general-purpose in-memory sorting algorithms in the average case. It does not require the extra array needed by Mergesort, so it is space efficient as well. Quicksort is widely used, and is typically the algorithm implemented in a library sort routine such as the UNIX `qsort` function. Interestingly, Quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications.

Quicksort first selects a value called the pivot. Assume that the input array contains k records with key values less than the pivot. The records are then rearranged in such a way that the k values less than the pivot are placed in the

first, or leftmost, k positions in the array, the pivot itself is placed at index k, and the values greater than or equal to the pivot are placed in the last, or rightmost, $N - k - 1$ positions. This is called a partition of the array. The values placed in a given partition need not (and typically will not) be sorted with respect to each other. All that is required is that all values end up in the correct partition. The pivot value itself is placed in position k. Quicksort then proceeds to sort the resulting subarrays now on either side of the pivot, one of size k and the other of size $N - k - 1$. How are these values sorted? Because Quicksort is such a good algorithm, using Quicksort on the subarrays would be appropriate.

Unlike some of the sorts that we have seen earlier in this chapter, Quicksort might not seem very "natural" in that it is not an approach that a person is likely to use to sort real objects. But it should not be too surprising that a really efficient sort for huge numbers of abstract objects on a computer would be rather different from our experiences with sorting a relatively few physical objects.

Here is an implementation for Quicksort. Parameters `left` and `right` define the left and right indices, respectively, for the subarray being sorted. The initial call to `quickSort` would be `quickSort(A,0,A.size-1)`, just as for mergesort.

```
function quickSort(A, left, right):
    if left >= right:                          // Base case: Subarray length is ≤ 1
        return
    pivot = findPivot(A, left, right)          // Pick a pivot index
    pivot = partition(A, left, right, pivot)   // Partition the subarray; update pivot with its new position
    quickSort(A, left, pivot-1)                // Sort left partition
    quickSort(A, pivot+1, right)               // Sort right partition
```

Function `partition` will move records to the appropriate partition and then return the final position of the pivot. This is the correct position of the pivot in the final, sorted array. By doing so, we guarantee that at least one value (the pivot) will not be processed in the recursive calls to `quickSort`. Even if a bad pivot is selected, yielding a completely empty partition to one side of the pivot, the larger partition will contain at most $N - 1$ records.

Selecting a pivot can be done in many ways. The simplest is to use the first key. However, if the input is sorted or reverse sorted, this will produce a poor partitioning with all values to one side of the pivot. One simple way to avoid this problem is to select the middle position in the array. Here is a simple `findPivot` function implementing this idea. Note that later in the chapter we will discuss better pivot selection strategies.

```
function findPivot(A, left, right) -> Int:
    // Not-so-good pivot selection: always choose the middle element.
    return int((left + right) / 2)
```

4.4.1 PARTITION

We now turn to partitioning. If we knew in advance how many keys are less than the pivot, we could simply copy records with key values less than the pivot to the low end of the array, and records with larger keys to the high end. Because we do not know in advance how many keys are less than the pivot, we use a clever algorithm that moves indices inwards from the ends of the subarray, swapping values as necessary until the two indices meet.

Since Quicksort is a recursive algorithm, we will not only partition the whole array, but also subarrays. Therefore `partition` needs the positions of the leftmost and rightmost elements in the subarray that we will partition.

```
function partition(A, left, right, pivot) -> Int:
    swap A[pivot] with A[left]
    while left ≤ right:
        move left rightwards until A[left] < A[pivot]
        move right leftwards until A[right] > A[pivot]
        swap A[left] with A[right]
    swap A[pivot] with A[right]
    return right
```

The partition function first puts the pivot at the leftmost position in the subarray. Then it moves the *left* and *right* pointers towards each other; until they both point to values which are unordered relative to the pivot value. Now the *left* and *right* values can be swapped, which puts them both in their correct partition. The loop continues until the *left* and *right* pointers have passed each other.

Finally, it puts the pivot at its correct position, by swapping with the *right* pointer. Why is *right* the correct pivot position, and not *left*? This is because we initially put the pivot first in the subarray, so the value that is swapped with the pivot must be smaller.

To analyse Quicksort, we first analyse the functions for finding the pivot and partitioning a subarray of length k. Clearly, `findPivot` takes constant time for any k.

The total cost of the partition operation is constrained by how far `left` and `right` can move inwards.

- The swap operation in the body of the main loop guarantees that `left` and `right` move at least one step each. Thus, the maximum number of times `swap` can be executed is $(k-1)/2$.
- In any case, since `left` and `right` always move towards each other, it will take a total of $k-1$ steps until they meet.
- Thus, the running time of the partition function is $O(k)$.

4.4.2 COMPLEXITY ANALYSIS

Quicksort's worst case will occur when the pivot does a poor job of breaking the array, that is, when there are no records in one partition, and $N-1$ records in the other.

- The pivot partitions the array into two parts: one of size 0 and the other of size $N-1$. This requires $N-1$ units of work.
- In the second level, the pivot breaks it into two parts: one of size 0 and the other of size $N-2$. This requires $N-2$ units of work.
- In the third level, the pivot breaks it into two parts: one of size 0 and the other of size $N-3$. This requires $N-3$ units of work.
- …
- In the last level, the pivot breaks a partition of size 2 into two parts: one of size 0 and the other of size 1. This requires a single unit of work.

Thus, the total amount of work is determined by the summation:

$$\sum_{i=1}^{N} i \quad = \quad \frac{1}{2}N(N-1) \ \in \ O(N^2)$$

Therefore, the worst case running time of Quicksort is $O(N^2)$.

This is terrible, no better than Insertion or Selection sort. When will this worst case occur? Only when each pivot yields a bad partitioning of the array. If the pivot values are selected at random, then this is extremely unlikely to happen. When selecting the middle position of the current subarray, it is still unlikely to happen. It does not take many good partitionings for Quicksort to work fairly well.

Best-case complexity Here is an explanation of the best-case running time of Quicksort:

- The pivot partitions the array into two halves of size $N/2$ each. This requires $O(N)$ amount of work.
- For each of the two partitions, the pivot breaks it into halves of size $N/4$ each. This requires $O(N)$ amount of work.
- For each of the four partitions, the pivot breaks it into halves of size $N/8$ each. This requires $O(N)$ amount of work.
- …
- In the last level, we reach N partitions each of size 1. This requires $O(N)$ amount of work.

Thus, at each level, all partition steps for that level do a total of $O(N)$ work. And if we always can find the perfect pivot, there will be only $\log N$ levels. So the best-case running time of Quicksort is $O(N \log N)$.

Average-case complexity Quicksort's average-case behaviour falls somewhere between the extremes of worst and best case. Average-case analysis considers the cost for all possible arrangements of input, summing the costs and dividing by the number of cases. We make one reasonable simplifying assumption: At each partition step, the pivot is equally likely to end in any position in the (sorted) array. In other words, the pivot is equally likely to break an array into partitions of sizes 0 and $N-1$, or 1 and $N-2$, and so on. Given this assumption, the average-case cost is computed from the following equation:

$$\begin{aligned} T(n) &= cn + \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-1-k)] \\ T(0) = T(1) &= c \end{aligned}$$

This is an unusual situation that the average case cost and the worst case cost have asymptotically different growth rates. Consider what "average case" actually means. We compute an average cost for inputs of size n by summing up for every possible input of size n the product of the running time cost of that input times the probability that that input will occur. To simplify things, we assumed that every permutation is equally likely to occur. Thus, finding the average means summing up the cost for every permutation and dividing by the number of permutations (which is $n!$). We know that some of these $n!$ inputs cost $O(n^2)$. But the sum of all the permutation costs has to be $(n!)(O(n \log n))$. Given the extremely high cost of the worst inputs, there must be very few of them. In fact, there cannot be a constant fraction of the inputs with cost $O(n^2)$. If even, say, 1% of the inputs have cost $O(n^2)$, this would lead to an average cost of $O(n^2)$. Thus, as n grows, the fraction of inputs with high cost must be going toward a limit of zero. We can conclude that Quicksort will run fast if we can avoid those very few bad input permutations. This is why picking a good pivot is so important.

4.5 Implementing quicksort

In this section we go into more details in how to implement Quicksort.

4.5.1 PARTITION

There were some important details that we didn't mention in the pseudocode for partition in the previous section.

- The pivot element should *not* be part of the actual partitioning, so after swapping the pivot into the leftmost position, we should move the left pointer one step.
- We also have to add a check that the left pointer doesn't continue moving rightwards when it has met the right pointer.
- And similarly for the right pointer.

Here is more detailed pseudocode that takes the details into account:

```
function partition(A, left, right, pivot) -> Int:
    swap(A, left, pivot)    // Put pivot at the leftmost index
    pivot = left
    left = left + 1         // Start partitioning from the element after the pivot

    while true:
        // Move the left pointer rightwards as far as possible,
        // as long as it hasn't passed the right pointer, *and* the value is smaller than the pivot.
        while left <= right and A[left] < A[pivot]:
            left = left + 1
        // Move the right pointer leftwards as far as possible,
        // as long as it hasn't passed the left pointer, *and* the value is greater than the pivot.
        while left <= right and A[right] > A[pivot]:
            right = right - 1
        // Break out of the loop if the pointers has passed each other.
        if left > right:
            break
        // Otherwise, swap the elements and move both pointers one step towards each other.
        swap(A, left, right)
        left = left + 1
        right = right - 1

    swap(A, pivot, right)   // Finally, move the pivot into place
    return right            // Return the position of the pivot
```

As we mentioned in the previous section, we swap the pivot with the value at the *right* pointer. This is because we started by putting the pivot first in the subarray.

Section 4.4.1

It is also possible to start by putting the pivot at the end of the subarray. In that case we have to swap the pivot value with the *left* pointer in the end. This is an equally good alternative partitioning algorithm.

Read the rest online

4.5.2 SELECTING THE PIVOT

Perhaps the most important choice in implementing Quicksort is how to choose the pivot. Choosing a bad pivot can result in all elements of the array ending up in the same partition, in which case Quicksort ends up taking quadratic time.

First or last element Choosing the *first* or the *last* element of the array is a bad strategy. If the input array is sorted, then the first element of the array will also be the smallest element. Hence all elements of the array will end up in the "greater than pivot" partition. Worse, the exact same thing will happen in all the recursive calls to Quicksort. Hence the partitioning will be as bad as possible, and Quicksort will end up taking quadratic time. You sometimes see implementations of Quicksort that use the first element as the pivot, but this is a bad idea!

Middle element Above, we picked the *middle* element of the array, to avoid this problem. This works well enough, but in practice, more sophisticated strategies are used.

Median-of-three The theoretically best choice of pivot is one that divides the array equally in two, i.e. the median element of the array. However, the median of an array is difficult to compute (unless you sort the array first!) Instead, many Quicksort implementations use a strategy called *median-of-three*. In median-of-three, we pick elements from three positions in the array: the *first* position, the *middle* position and the *last* position. Then we take the median of these three elements. For example, given the array [3, 1, 4, 1, 5, 9, 2], we pick out the elements 3 (first position), 1 (middle position) and 2 (last position). The median of 3, 1 and 2 is 2, so we pick 2 as the pivot.

Median-of-three is not guaranteed to pick a good pivot: there are cases where it partitions the input array badly. However, these bad cases do not seem to occur in practice. In practice, median-of-three picks good pivots, and it is also cheap to implement. It is used by most real-world Quicksort implementations.

Random pivot Another good approach is to pick a random element of the array as the pivot. This makes it somewhat unlikely to get a poor partitioning. What's more, if we do get a poor partitioning, it is likely that in the recursive call to `quickSort`, we will choose a different pivot and get a better partitioning. Unlike median-of-three, this approach is theoretically sound: there are no input arrays which make it work badly.

Another way to get the same effect is to pick e.g. the first element as the

pivot, but to *shuffle* the array before sorting, rearranging it into a random order. The array only needs to be shuffled once before Quicksort begins, not in every recursive call.

4.5.3 MORE PRACTICAL IMPROVEMENTS

There are some things we can do to improve the efficiency of Quicksort.

Backing off to Insertion sort A significant improvement can be gained by recognising that Quicksort is relatively slow when the array is small. This might not seem to be relevant if most of the time we sort large arrays, nor should it matter how long Quicksort takes in the rare instance when a small array is sorted because it will be fast anyway. But you should notice that Quicksort itself sorts many, many small arrays! This happens as a natural by-product of the divide and conquer approach.

A simple improvement is to replace Quicksort with a faster sort for small subarrays. This is a very common improvement, and usually one uses Insertion sort as the backoff algorithm. Now, at what size should we switch to Insertion sort? The answer can only be determined by empirical testing, but on modern machines the answer is probably somewhere between 10 and 100.

Section 4.5.3

Read the rest online

Note that this improvement can also be used for Mergesort!

::::::: #### Running Insertion sort in a single final pass

There is a variant of this optimisation: When Quicksort partitions are below a certain size, do nothing! The values within that partition will be out of order. However, we do know that all values in the array to the left of the partition are smaller than all values in the partition. All values in the array to the right of the partition are greater than all values in the partition. Thus, even if Quicksort only gets the values to "nearly" the right locations, the array will be close to sorted. This is an ideal situation in which to take advantage of the best-case performance of Insertion sort. The final step is a single call to Insertion sort to process the entire array, putting the records into final sorted order.

Reduce recursive calls The last speedup to be considered reduces the cost of making recursive calls. Quicksort is inherently recursive, because each Quicksort operation must sort two sublists. Thus, there is no simple way to turn Quicksort into an iterative algorithm. However, Quicksort can be implemented using a stack to imitate recursion, as the amount of information that must be stored is small. We need not store copies of a subarray, only the subarray bounds. Furthermore, the stack depth can be kept small if care is taken on the order in which Quicksort's

recursive calls are executed. We can also place the code for `findPivot` and `partition` inline to eliminate the remaining function calls. Note however that by not processing small sublists of size nine or less as suggested above, most of the function calls will already have been eliminated. Thus, eliminating the remaining function calls will yield only a modest speedup.

::::::

Section 4.6

Read the rest online

4.6 Empirical comparison of sorting algorithms

Which sorting algorithm is fastest? Asymptotic complexity analysis lets us distinguish between $O(n^2)$ and $O(n \log n)$ algorithms, but it does not help distinguish between algorithms with the same asymptotic complexity. Nor does asymptotic analysis say anything about which algorithm is best for sorting small lists. For answers to these questions, we can turn to empirical testing.

Section 4.7

Answer quiz online

4.7 Review questions

This final section contains some review questions about the contents of this chapter.

CHAPTER 5

Algorithm analysis, part 2: Theory

In chapter 2 we introduced the ideas behind algorithmic analysis, and explained the basics on an abstract level. Now that you understand everything about sorting algorithms, we can go into more depth with our analysis tools.

After reading this chapter you should be able to analyse the computational complexity of most algorithms, both for sorting and the data structures we will introduce later.

5.1 Upper bounds: the big-O notation

Several terms are used to describe the running-time equation for an algorithm. These terms – and their associated symbols – indicate precisely what aspect of the algorithm's behaviour is being described. One is the upper bound for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have.

Because the phrase "has an upper bound to its growth rate of $f(n)$" is long and often used when discussing algorithms, we adopt a special notation, called big-O notation. If the upper bound for an algorithm's growth rate (for, say, the worst case) is $f(n)$, then we would write that this algorithm is "in the set $O(f(n))$ in the worst case" (or just "in $O(f(n))$ in the worst case"). For example, if n^2 grows as fast as $T(n)$ (the running time of our algorithm) for the worst-case input, we would say the algorithm is "in $O(n^2)$ in the worst case".

We already defined the upper bound in Section 2.6.2, like this:

Upper bound $f \in O(g)$ **iff** there exist positive numbers k and n_0 such that $f(n) \leq k \cdot g(n)$ for all $n > n_0$

The constant n_0 is the smallest value of n for which the claim of an upper bound holds true. Usually n_0 is small, such as 1, but does not need to be. You must also be able to pick some constant k, but it is irrelevant what the value for k actually is. In other words, the definition says that for *all* inputs of the type in question (such as the worst case for all inputs of size n) that are large enough

(i.e., $n > n_0$), the algorithm *always* executes in less than or equal to $k \cdot g(n)$ steps for some constant k.

Note that the definition is somewhat simplified, it only works if f and g are *monotonically increasing*. This means that if $x \leq y$ then $f(x) \leq f(y)$, so the value can never decrease whenever x increases.

But this is not a real restriction for our purposes, because there are no algorithms that becomes faster when the input size grows. In the very best case, the runtime of an algorithm can be independent of the input size, but this is also monotonically increasing.

If we were to allow any non-monotonic functions, then the definition of upper bound would become slightly more complicated. You can look up the formal definition in mathematical textbooks, or in Wikipedia.

Example: Sequential search

Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $T(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Example: Quadratic algorithm

For a particular algorithm, $T(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then,

$$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$$

for all $n > 1$. So, $T(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $T(n)$ is in $O(n^2)$ by the definition.

Example: Accessing an array cell

Assigning the value from a given position of an array to a variable takes constant time regardless of the size of the array. Thus, $T(n) = c$ (for the best, worst, and average cases). We could say in this case that $T(n)$ is in

$O(c)$. However, it is traditional to say that an algorithm whose running time has a constant upper bound is in $O(1)$.

If someone asked you out of the blue "Who is the best?" your natural reaction should be to reply "Best at what?" In the same way, if you are asked "What is the growth rate of this algorithm", you would need to ask "When? Best case? Average case? Or worst case?" Some algorithms have the same behaviour no matter which input instance of a given size that they receive. An example is finding the maximum in an array of integers. But for many algorithms, it makes a big difference which particular input of a given size is involved, such as when searching an unsorted array for a particular value. So any statement about the upper bound of an algorithm must be in the context of some specific class of inputs of size n. We measure this upper bound nearly always on the best-case, average-case, or worst-case inputs. Thus, we cannot say, "this algorithm has an upper bound to its growth rate of n^2" because that is an incomplete statement. We must say something like, "this algorithm has an upper bound to its growth rate of n^2 *in the average case*".

Knowing that something is in $O(f(n))$ says only how bad things can be. Perhaps things are not nearly so bad. Because sequential search is in $O(n)$ in the worst case, it is also true to say that sequential search is in $O(n^2)$. But sequential search is practical for large n in a way that is not true for some other algorithms in $O(n^2)$. We always seek to define the running time of an algorithm with the tightest (lowest) possible upper bound. Thus, we prefer to say that sequential search is in $O(n)$. This also explains why the phrase "is in $O(f(n))$" or the notation "$\in O(f(n))$" is used instead of "is $O(f(n))$" or "$= O(f(n))$". There is no strict equality to the use of big-O notation. $O(n)$ is in $O(n^2)$, but $O(n^2)$ is not in $O(n)$.

5.1.1 SIMPLIFYING RULES

We introduced simplifying rules in Section 2.7.1, but repeat them here in compact form:

	Rule	Simplification	Alternatively
(1)	Transitivity	if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$	
(2)	Constant $k > 0$	if $f \in O(g)$, then $k \cdot f \in O(g)$	$k \cdot O(g) = O(g)$
(3)	Addition	if $f \in O(g)$ and $f' \in O(g')$, then $f + f' \in O(\max(g, g'))$	$O(g) + O(g') = O(\max(g,$

	Rule	Simplification	Alternatively
(4)	Multiplication	if $f \in O(g)$ and $f' \in O(g')$, then $f \cdot f' \in O(g \cdot g')$	$O(g) \cdot O(g') = O(g \cdot g')$

Using these rules we can easily determine the asymptotic growth rate for many algorithms.

- Rule (2) says that any constant-time operation is $O(1)$.
- Rule (3) says that if you have a sequence of statements, you only need to consider the most expensive statement: $O(\max(f_1, \ldots, f_k))$.
- Rule (4) says that if you have a loop that repeats a statement p a number of times, the total cost is the cost of p times the number of iterations: $O(n \cdot f)$.
- Rule (2) also says that if you repeat a statement p a *constant* number of times, you can treat it as you only execute p once.

5.1.2 BIG-*O* AND LOGARITHMS

One interesting consequence of asymptotic complexity is that the base of a logarithm becomes irrelevant:

$$O(\log_2(n)) = O(\ln(n)) = O(\log_1 0(n))$$

The reason for this is that according to the logarithm laws, $\log_b(n) = \log_a(n) \cdot \frac{1}{\log_a(b)}$. But $\frac{1}{\log_a(b)}$ is a constant which we can ignore, so $O(\log_b(n)) = O(\log_a(n))$. Therefore we can just ignore the base and write $O(\log n)$.

(Note that this *does not* hold for exponential growth – e.g., $2^n \in O(10^n)$, but $10^n \notin O(2^n)$.)

Another consequence of the logarithm laws is that it doesn't really matter if you take the logarithm from a linear, quadratic, cubic, or any power function:

$$O(\log n) = O(\log n^2) = O(\log n^3) = O(\log n^k)$$

The reason for this is that $\log n^k = k \cdot \log n$ according to the logarithm laws, so the exponent k becomes a multiplicative constant and can be ignored.

However, taking the power of a logarithm cannot be ignored, so $O(\log n)$ and $O(\log^2 n)$ are different complexity classes.

5.2 Lower bounds and tight bounds

The definition for big-*O* allows us to greatly overestimate the cost for an algorithm. But sometimes we know a tight bound – that is, a bound that truly reflects the cost of the algorithm or program with a constant factor. In that case, we

can express this more accurate state of our knowledge using the tight bound Θ instead of using big-O.

However, it is usually much more difficult to reason about the tight bound, for example, the simplifying rules for addition and multiplication do not hold for Θ. Therefore we will almost exclusively use the upper bound big-O notation.

In this section we assume that all functions are *monotonically increasing*, just as we did in the previous section. If we didn't assume this, the definitions would be slightly more complicated. But, as discussed in the previous section, the runtime of all algorithms never decreases, so it is a safe assumtion.

5.2.1 LOWER BOUNDS: THE Ω NOTATION

Big-O notation describes an upper bound. In other words, big-O states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n.

A similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-O, this is a measure of the algorithm's growth rate. And like big-O, it works for any resource (usually time), and for some particular class of inputs of size n.

The lower bound for an algorithm (or a problem, as we will discuss in Section 5.3) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega". The following definition for Ω is symmetric with the definition of big-O.

Lower bound $f \in \Omega(g)$ **iff** there exist positive numbers k and n_0 such that $f(n) \geq k \cdot g(n)$ for all $n > n_0$

Example: Quadratic algorithm

Assume $T(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

It is also true that the equation of the example above is in $\Omega(n)$. However, as with big-O notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n^2)$.

Recall the sequential search algorithm to find a value within an array of integers. In the worst case this algorithm is in $\Omega(n)$, because in the worst case we must examine *at least* n values.

Alternative definition for Ω An alternate (non-equivalent) definition for Ω is

Lower bound (alt.) $f \in \Omega(g)$ **iff** there exists a positive number k such that $f(n) \geq k \cdot g(n)$ for an infinite number of values for n.

This definition says that for an "interesting" number of cases, the algorithm takes at least $k \cdot g(n)$ time. Note that this definition is *not* symmetric with the definition of big-O. For g to be a lower bound, this definition *does not* require that $f(n) \geq k \cdot g(n)$ for all n greater than some constant. It only requires that this happen often enough, in particular that it happen for an infinite number of values for n. Motivation for this alternate definition can be found in the following example.

Assume a particular algorithm has the following behaviour:

$$T(n) = \begin{cases} n & \text{for all odd } n \\ n^2/100 & \text{for all even } n \end{cases}$$

From this definition, $n^2/100 \geq k \cdot n^2$ for all even n, for any $k < 0.01$. So, $T(n) \geq k \cdot n^2$ for an infinite number of values of n. Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

For this equation for $T(n)$, it is true that all inputs of size n take at least cn time. But an infinite number of inputs of size n take cn^2 time, so we would like to say that the algorithm is in $\Omega(n^2)$. Unfortunately, using our first definition will yield a lower bound of $\Omega(n)$ because it is not possible to pick constants k and n_0 such that $T(n) \geq k \cdot n^2$ for all $n > n_0$. The alternative definition does result in a lower bound of $\Omega(n^2)$ for this algorithm, which seems to fit common sense more closely. Fortunately, few real algorithms or computer programs display the pathological behaviour of this example. Our first definition for Ω generally yields the expected result.

As you can see from this discussion, asymptotic bounds notation is not a law of nature. It is merely a powerful modeling tool used to describe the behaviour of algorithms.

5.2.2 TIGHT BOUNDS: THE Θ NOTATION

The definitions for big-O and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular

class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ *and* it is in $\Omega(h(n))$. Note that we drop the word "in" for Θ notation, because there is a strict equality for two equations with the same Θ. In other words, if $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$.

Because the sequential search algorithm is both in $O(n)$ and in $\Omega(n)$ in the worst case, we say it is $\Theta(n)$ in the worst case.

Given an algebraic equation describing the time requirement for an algorithm, the upper and lower bounds always meet. That is because in some sense we have a perfect analysis for the algorithm, embodied by the running-time equation. For many algorithms (or their instantiations as programs), it is easy to come up with the equation that defines their runtime behaviour. The analysis for most commonly used algorithms is well understood and we can almost always give a Θ analysis for them. However, the class of NP-Complete problems all have no definitive Θ analysis, just some unsatisfying big-O and Ω analyses. Even some "simple" programs are hard to analyse. Nobody currently knows the true upper or lower bounds for the following code fragment.

```
while n > 1:
    if n is odd:
        n = 3 * n + 1
    else:
        n = n / 2
```

But even though Θ is a more accurate description of the behaviour of an algorithm, we have chosen to almost exclusively use the upper bound big-O notation. The reason for this because it is more difficult to reason about the tight bound than about big-O. For example, the simplifying rules for addition and multiplication do not hold for Θ. Another reason is that most other textbooks, research papers, and programmers will usually say that an algorithm is "order of" or "big-O" of some cost function, implicitly meaning that this is the tightest possible bound.

5.2.3 STRICT BOUNDS

The upper and lower bounds are not strict, meaning that a function is in its own class, $f \in O(f)$ and $f \in \Omega(f)$. We can also define strict versions of upper and lower bounds:

Strict upper bound (little-o) $f \in o(g)$ **iff** $f \in O(g)$ and $f \notin \Omega(g)$
Strice lower bound (ω) $f \in \omega(g)$ **iff** $f \in \Omega(g)$ and $f \notin O(g)$

5.2.4 ASYMPTOTIC NOTATION AND COMPARING COMPLEXITY CLASSES

We can summarise the different asymptotic notations (O, o, Ω, ω, and Θ) in the following table:

Name	Bound	Notation	Definition
Little-O	Strict upper bound	$f(n) \in o(g(n))$	$\lvert f(n)\rvert < k \cdot g(n)$
Big-O	Upper bound	$f(n) \in O(g(n))$	$\lvert f(n)\rvert \leq k \cdot g(n)$
Theta	Tight bound	$f(n) \in \Theta(g(n))$	$k_1 \cdot g(n) \leq \lvert f(n)\rvert \leq k_2 \cdot g(n)$
Big-Omega	Lower bound	$f(n) \in \Omega(g(n))$	$f(n) \geq k \cdot g(n)$
Little-Omega	Strict lower bound	$f(n) \in \omega(g(n))$	$f(n) > k \cdot g(n)$

All these different bounds correspond to comparison operators between complexity classes:

Comparison	Complexity class
$O(f) < O(g)$	$f \in o(g)$
$O(f) \leq O(g)$	$f \in O(g)$
$O(f) = O(g)$	$f \in \Theta(g)$
$O(f) \geq O(g)$	$f \in \Omega(g)$
$O(f) > O(g)$	$f \in \omega(g)$

Using these correspondences and the simplifying rules we can infer the following hierarchy of complexity classes:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^2 \log n) < O(n^3) < \cdots < O(n^k) < O(2^n) < \cdots$$

Zooming in on the very efficient (sub-linear) complexity classes we have:

$$O(1) < O(\log\log n) < O(\log n) = O(\log n^2) = O(\log n^3) < O(\log^2 n) < O(\log^3 n) < O(\sqrt[3]{n}) < O(\sqrt{n}) < O(n)$$

And if we instead look closer on the extreme other end of the scale:

$$\cdots < O(n^1000) < O(1.0001^n) < O(2^n) < O(10^n) < O(1000^n) < O(n!) < O(n^n) < \cdots$$

5.2.5 CLASSIFYING FUNCTIONS USING LIMITS

There are alternative definitions of the upper, lower and tight bounds. Instead of finding constants k and n_0, we can see how the quotient between the two functions behave in the limit.

Given functions f and g, we can take the limit of the quotient of the two as n grows towards infinity:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

We have the following possibilities:

Name	Notation	Limit, $\lim(f/g) \to k$
Little-O	$f \in o(g)$	$k = 0$
Big-O	$f \in O(g)$	$k < \infty$
Theta	$f \in \Theta(g)$	$0 < k < \infty$
Big-Omega	$f \in \Omega(g)$	$k > 0$
Little-Omega	$f \in \omega(g)$	$k = \infty$

Example: Comparing two functions

Assume $f(n) = n^2$ and $g(n) = 1000n \log n$. Is f in $O(g)$, $\Omega(g)$, or $\Theta(g)$? To answer this we can calculate the limit of the quotient $f(n)/g(n)$ when n grows:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{1000n \log n} = \frac{1}{1000} \cdot \lim_{n\to\infty} \frac{n}{\log n} = \infty$$

because n grows faster than $\log n$. Thus, $f \in \Omega(g)$ (or equivalently, $g \in O(f)$).

5.3 Analysing problems

You most often use the techniques of algorithm analysis to analyse an algorithm (or the instantiation of an algorithm as a program). But you can also use these same techniques to analyse the cost of a problem. The key question that we want to ask is: How hard is a problem?

Certainly we should expect that in some sense, the problem of sorting a list of records is harder than the problem of searching a list of records for a given key value. Certainly the algorithms that we know for sorting some records

seem to be more expensive than the algorithms that we know for searching those same records.

What we need are useful definitions for the upper bound and lower bound of a *problem*, instead of an algorithm.

One might start by thinking that the upper bound for a problem is how hard any algorithm can be for the problem. But we can make algorithms as bad as we want, so that is not useful. Instead, what is useful is to say that a problem is only as hard as what we *can* do. In other words, we should define the upper bound for a problem to be the *best* algorithm that we know for the problem.

But what does it then mean to give a lower bound for a problem? Lower bound refers to the minimum that any algorithm *must* cost. For example, when searching an unsorted list, we *must* look at every record. When sorting a list, we *must* look at every record (to even know if it is sorted).

So, how do upper and lower bounds relate to the key question – how hard is a problem? As we have argued, the upper bound relies on our knowledge of the currently best algorithm. But how can we be certain that this algorithm is as good as it can be? To know this we have to know about the lower bound of the problem, i.e., the lower bound tells us how hard a problem is.

As a rule of thumb we can say:

- when we analyse an algorithm, we are interested in the *upper bound*, big-O
- when we analyse a problem, we are instead interested in the *lower bound*, Ω

It is much easier to show that an algorithm (or program) is in $\Omega(f)$ than it is to show that a problem is in $\Omega(f)$. For a problem to be in $\Omega(f)$ means that *every* algorithm that solves the problem is in $\Omega(f)$, even algorithms that we have not thought of! In other words, *every* algorithm *must* have at least this cost. So, to prove a lower bound, we need an argument that is true, even for algorithms that we don't know.

So far all of our examples of algorithm analysis give "obvious" results, with big-O always matching Ω. To understand how big-O, Ω, and Θ notations are properly used to describe our understanding of a problem or an algorithm, it is best to consider an example where you do not already know a lot about the problem.

Let us look ahead to analysing the problem of sorting to see how this process works. What is the least possible cost for any sorting algorithm in the worst case? The algorithm must at least look at every element in the input, just to determine that the input is truly sorted. Thus, any sorting algorithm must take at least

$\Omega(n)$ time. For many problems, this observation that each of the n inputs must be looked at leads to an easy $\Omega(n)$ lower bound.

In the previous chapter about sorting, you learned about some sorting algorithms whose running time is in $O(n^2)$ – bubble sort, selection sort and insertion sort. But you also learned about the linearithmic sorting algorithms quicksort and mergesort with a running time in $O(n \log n)$. Thus, the problem of sorting can be said to have an upper bound in $O(n \log n)$. How do we close the gap between $\Omega(n)$ and $O(n \log n)$? Can there be even better sorting algorithms than mergesort and quicksort? If you can think of no algorithm whose worst-case growth rate is better than $O(n \log n)$, and if you have discovered no analysis technique to show that the least cost for the problem of sorting in the worst case is greater than $\Omega(n)$, then you cannot know for sure whether or not there is a better algorithm.

Should we search for a faster algorithm? Many have tried, without success. Fortunately (or perhaps unfortunately?), we can prove that *any* sorting algorithm must have running time in $\Omega(n \log n)$ in the worst case. So, the problem of sorting has a linearithmic lower bound, which is the same as the upper bounds for the best sorting algorithms. Thus, we can conclude that the problem of sorting is $\Theta(n \log n)$ in the worst case, because the upper and lower bounds have met.

Knowing the lower bound for a problem does not give you a good algorithm. But it does help you to know when to stop looking. If the lower bound for the problem matches the upper bound for the algorithm (within a constant factor), then we know that we can find an algorithm that is better only by a constant factor.

So, to summarise: The upper bound for a problem is the best that you *can* do, while the lower bound for a problem is the least work that you *must* do. If those two are the same, then we can say that we really understand our problem.

5.3.1 CASE STUDY: LOWER BOUNDS FOR SORTING

By now you have seen many analyses for algorithms. These analyses generally define the worst-case upper bounds. For many of the algorithms presented so far, analysis has been quite easy. This section considers a more difficult task: An analysis for the cost of a *problem* as opposed to an *algorithm*.

As we explained earlier, the lower bound defines the best possible cost for *any* algorithm that solves the problem, including algorithms not yet invented. Once we know the lower bound for the problem, we know that no future algorithm can possibly be (asymptotically) more efficient.

A simple estimate for a problem's lower bound can be obtained by measuring the size of the input that must be read and the output that must be written.

Certainly no algorithm can be more efficient than the problem's I/O time. From this we see that the sorting problem cannot be solved by *any* algorithm in less than $\Omega(n)$ time because it takes at least n steps to read and write the n values to be sorted. Alternatively, any sorting algorithm must at least look at every input value to recognise whether the input values are in sorted order. So, based on our current knowledge of sorting algorithms and the size of the input, we know that the *problem* of sorting is bounded by $\Omega(n)$ and $O(n \log n)$.

This section presents one of the most important and most useful proofs in computer science: No sorting algorithm based on key comparisons can possibly be faster than $\Omega(n \log n)$ in the worst case. This proof is important for three reasons. First, knowing that widely used sorting algorithms are asymptotically optimal is reassuring. In particular, it means that you need not bang your head against the wall searching for an $O(n)$ sorting algorithm. (Or at least not one that is in any way based on key comparisons. But it is hard to imagine how to sort without any comparisons.) Second, this proof is one of the few non-trivial lower-bounds proofs that we have for any problem; that is, this proof provides one of the relatively few instances where our lower bound is tighter than simply measuring the size of the input and output. As such, it provides a useful model for proving lower bounds on other problems. Finally, knowing a lower bound for sorting gives us a lower bound in turn for other problems whose solution could be made to work as the basis for a sorting algorithm. The process of deriving asymptotic bounds for one problem from the asymptotic bounds of another is called a reduction.

All of the sorting algorithms we have studied make decisions based on the direct comparison of two key values. For example, Insertion sort sequentially compares the value to be inserted into the sorted list until a comparison against the next value in the list fails.

The proof that any comparison sort requires $\Omega(n \log n)$ comparisons in the worst case is structured as follows:

1. First, comparison-based decisions can be modeled as the branches in a tree. This means that any sorting algorithm based on comparisons between records can be viewed as a binary tree whose nodes correspond to the comparisons, and whose branches correspond to the possible outcomes.
2. Next, the minimum number of leaves in the resulting tree is shown to be the factorial of n.
3. Finally, the minimum depth of a tree with $n!$ leaves is shown to be in $\Omega(n \log n)$.

Before presenting the proof of an $\Omega(n \log n)$ lower bound for sorting, we first must define the concept of a decision tree. A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions. Each (binary) decision is represented by a branch in the tree. For the purpose of modeling sorting algorithms, we count all comparisons of key values as decisions. If two keys are compared and the first is less than the second, then this is modeled as a left branch in the decision tree. In the case where the first value is greater than the second, the algorithm takes the right branch.

Any sorting algorithm requiring $\Omega(n \log n)$ comparisons in the worst case requires $\Omega(n \log n)$ running time in the worst case. Because any sorting algorithm requires $\Omega(n \log n)$ running time, the problem of sorting also requires $\Omega(n \log n)$ time. We already know of sorting algorithms with $O(n \log n)$ running time, so we can conclude that the problem of sorting requires $\Theta(n \log n)$ time. As a corollary, we know that no comparison-based sorting algorithm can improve on existing $\Theta(n \log n)$ time sorting algorithms by more than a constant factor.

5.4 Common misunderstandings

Asymptotic analysis is one of the most intellectually difficult topics that undergraduate computer science majors are confronted with. Most people find growth rates and asymptotic analysis confusing and so develop misconceptions about either the concepts or the terminology. It helps to know what the standard points of confusion are, in hopes of avoiding them.

One problem with differentiating the concepts of upper and lower bounds is that, for most algorithms that you will encounter, it is easy to recognise the true growth rate for that algorithm. Given complete knowledge about a cost function, the upper and lower bound for that cost function are always the same. Thus, the distinction between an upper and a lower bound is only worthwhile when you have incomplete knowledge about the thing being measured. We can use the Θ-notation to indicate that there is no meaningful difference between what we know about the growth rates of the upper and lower bound (which is usually the case for simple algorithms).

It is a common mistake to confuse the concepts of upper bound or lower bound on the one hand, and worst case or best case on the other. The best, worst, or average cases each **define a cost** for a specific input instance (or specific set of instances for the average case). In contrast, upper and lower bounds describe our understanding of the **growth rate** for that cost measure. So to define the

growth rate for an algorithm or problem, we need to determine what we are measuring (the best, worst, or average case) and also our description for what we know about the growth rate of that cost measure (big-O, Ω, or Θ).

The upper bound for an algorithm is not the same as the worst case for that algorithm for a given input of size n. What is being bounded is not the actual cost (which you can determine for a given value of n), but rather the **growth rate** for the cost. There cannot be a growth rate for a single point, such as a particular value of n. The growth **rate** applies to the **change** in cost as a **change** in input size occurs. Likewise, the lower bound is not the same as the best case for a given size n.

Another common misconception is thinking that the best case for an algorithm occurs when the input size is as small as possible, or that the worst case occurs when the input size is as large as possible. What is correct is that best- and worse-case instances exist for each possible size of input. That is, for all inputs of a given size, say n, one (or more) of the inputs of size n is the best and one (or more) of the inputs of size n is the worst. Often (but not always!), we can characterise the best input case for an arbitrary size, and we can characterise the worst input case for an arbitrary size. Ideally, we can determine the growth rate for the characterised best, worst, and average cases as the input size grows.

Example: Best case for sequential search

What is the growth rate of the best case for sequential search? For any array of size n, the best case occurs when the value we are looking for appears in the first position of the array. This is true regardless of the size of the array. Thus, the best case (for arbitrary size n) occurs when the desired value is in the first of n positions, and its cost is 1. It is *not* correct to say that the best case occurs when $n = 1$.

5.4.1 BEST-CASE UPPER BOUND, OR WORST-CASE LOWER BOUND?

Note that even though it is possible to analyse all possible combinations of (upper/lower/tight) bounds, and (best/worst/average) case, there are only a few combinations that are of interest.

When it comes to analysing algorithms, we are usually not at all interested in any best-case analysis. After all, knowing that an algorithm performs well in some very lucky cases doesn't say if it's a good algorithm – it is much more important to know how it performs on worst-case inputs, or sometimes in the average case.

In the same way, it is not very interesting to learn about the lower bound of

an algorithm. This tells us that the algorithm cannot run faster than the lower bound, but usually this lower bound is very fast anyway.

That leaves us with analysing the worst-case upper bound, or sometimes the average-case behaviour. So this is what we almost always do.

5.5 Review questions

This final section contains some review questions about the contents of this chapter.

Section 5.5

Answer quiz online

CHAPTER 6

Stacks, queues, and lists

If your program needs to store a few things – numbers, payroll records, or job descriptions for example – the simplest and most effective approach might be to put them in a list. Only when you have to organise and search through a large number of things do more sophisticated data structures like search trees become necessary. Many applications don't require any form of search, and they do not require that an ordering be placed on the objects being stored. Some applications require that actions be performed in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes two different representations lists-like structures, the linked list and the array-based list. We also show how these representations can be used to implement important list-like structures such as the stack and the queue. Along with presenting these fundamental data structures, the other goals of the chapter are to:

1. Give examples that show the separation of a logical representation in the form of an ADT from a physical implementation as a data structure.
2. Illustrate the use of asymptotic analysis in the context of simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analysing more sophisticated algorithms and data structures.

6.1 Collections

A *collection* is a general term for structures like lists and queues. It holds multiple elements and supports two main operations: checking the number of elements and iterating through them one at a time.

```
interface Collection of T:
    // We assume that we can iterate over the elements in the collection, using a for loop.
```

```
size: Int                // The number of elements in this collection.
isEmpty() -> Bool:       // Returns true if the collection is empty.
    return size == 0
```

Note that this very interface will not be implemented as it is, but instead we will use this as a base interface that we extend in different ways, e.g., for lists or sets or priority queues.

6.1.1 WHAT IS A SEQUENCE?

We all have an intuitive understanding of what we mean by a "list". We want to turn this intuitive understanding into a concrete data structure with implementations for its operations. The most important concept related to lists is that of position. In other words, we perceive that there is a first element in the list, a second element, and so on. So, define a list to be a finite, ordered sequence of data items known as elements. This is close to the mathematical concept of a sequence.

"Ordered" in this definition means that each element has a position in the list. So the term "ordered" in this context does ***not*** mean that the list elements are sorted by value. (Of course, we can always choose to sort the elements on the list if we want; it's just that keeping the elements sorted is not an inherent property of being a list.)

Each list element must have some data type. In the simple list implementations discussed in this chapter, all elements of the list are usually assumed to have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it. The operations defined as part of the list ADTs depend on the elemental data type. For example, the queue ADT can be used for queues of integers, queues of characters, queues of payroll records, even queues of queues.

A list is said to be empty when it contains no elements. The number of elements currently stored is called the length (or size) of the list. The beginning of the list is called the head, the end of the list is called the tail.

6.2 Stacks and queues

The stack is a list-like structure in which elements may be inserted or removed from only one end. This is an extremely simplistic ADT, but it is nevertheless used in many many algorithms. The restrictions make stacks very inflexible, but they also make stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and

remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than a generic list.

Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a "LIFO" list, which stands for "Last-In, First-Out." Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

Like the stack, the queue is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an enqueue operation) and removed from the front (called a dequeue operation). Queues operate like standing in line at a movie theater ticket counter. If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. In Britain, a line of people is called a "queue", and getting into line to wait for service is called "queuing up". Accountants have used queues since long before the existence of computers. They call a queue a "FIFO" list, which stands for "First-In, First-Out".

ADT for stacks The accessible element of the stack is called the *top* element. Elements are not said to be inserted, they are pushed onto the stack. When removed, an element is said to be popped from the stack. Here is our ADT for stacks:

```
interface Stack of T extends Collection:
    push(x: T)      // Pushes x on top of the stack.
    pop() -> T      // Pops the top of the stack and returns it.
    peek() -> T     // Returns the top element, without removing it.
```

There are two main approaches to implementing stacks: the linked stack and the array-based stack. They will be discussed in Sections 6.3, 6.5 and 6.7.

ADT for queues The accessible element of the queue is called the *front* element. Inserting is called enqueue and removing dequeue. Here is our ADT for queues:

```
interface Queue of T extends Collection:
    enqueue(x: T)     // Enqueues x at the end of the queue.
    dequeue() -> T    // Dequeues the frontmost element.
    peek() -> T       // Returns the frontmost element, without removing it.
```

There are two main queue implementations: the linked queue and the array-based queue. They are discussed in Sections 6.4, 6.6 and 6.7.

6.2.1 CASE STUDY: IMPLEMENTING RECURSION

Perhaps the most common computer application that uses [stacks] is not even visible to its users. This is the implementation of subroutine calls in most programming language runtime environments. A subroutine call is normally implemented by pushing necessary information about the subroutine (including the return address, parameters, and local variables) onto a stack. This information is called an activation record. Further subroutine calls add to the stack. Each return from a subroutine pops the top activation record off the stack.

Section 6.2.1

Read the rest online

6.3 Stacks implemented as linked lists

In this section we present one of the two traditional implementations for lists, usually called a linked list. The linked list uses dynamic memory allocation, that is, it allocates memory for new list elements as needed. The following diagram illustrates the linked list concept. There are three nodes that are "linked" together. Each node has two boxes. The box on the right holds a link to the next node in the list. Notice that the rightmost node does not have any link coming out of this box.

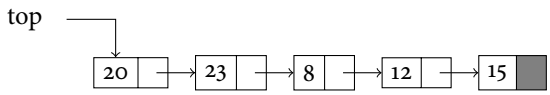
Linked list nodes Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate data type for list nodes. Objects of the data type `Node` contain a field `elem` to store the element value, and a field `next` to store a pointer to the next node on the list.

```
datatype Node of T:
    elem: T          // Value for this node
    next: Node of T  // Pointer to next node in list
```

The list built from such nodes is called a singly linked list, or a one-way list, because each list node has a single pointer to the next node on the list.

In this section and the next we describe how to use linked lists to implement stacks and queues, and in Section 6.9 and Section 6.10 we will discuss extensions such as double-ended queues and general lists.

Linked stacks The linked stack implementation is quite simple. Elements are inserted and removed only from the head of the list. Here is a visual representation for a linked stack.



Our data type for linked stacks contains two instance variables, one pointer to the head of the stack (called the `top`), and a variable storing the number of elements. (This second variable is in theory unnecessary, but it improves the efficiency of getting the stack size).

```
datatype LinkedStack implements Stack:
    top: Node = null    // Pointer to top of stack
    size: Int = 0       // Size of stack
```

To push a new element onto the stack, we first have to create a new node and set its value. Then we set its next pointer to the current top of the stack, and after that we can redirect the top to the new node. We also have to increase the size of the stack by one. The actions to create the node, set its value and pointer, and then redirect the stack top, can be done in one single line, like this:

```
datatype LinkedStack:
    ...
    push(elem):
        top = new Node(elem, top)
        size = size + 1
```

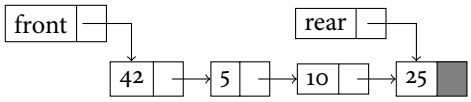
To pop the topmost element off the stack, we first have to remember a pointer to the current top node, because we want to return its value after we have updated the stack. Then we can redirect the stack top to the next node. After that we decrease the size and return the value of the removed node:

```
datatype LinkedStack:
    ...
    pop():
        removed = top
        top = removed.next
        removed.next = null   // For garbage collection
        size = size - 1
        return removed.elem
```

Note that we also set the `next` pointer of the old top to **null**. This is to help the garbage collection system actually remove the whole node when noone is using it anymore.

6.4 Queues implemented as linked lists

The linked queue implementation is an adaptation of the linked list. The only thing is that we have to add a pointer to the rear node in the queue, to be able to add new elements efficiently.



So the datatype for linked queues contains three instance variables, including the size of the queue:

```
datatype LinkedQueue implements Queue:
    front: Node = null    // Pointer to front node
    rear: Node = null     // Pointer to rear node
    size: Int = 0         // Size of queue
```

To enqueue a new element onto the stack, we first have to create a new node and set its value. But instead of inserting the node at the front of the queue, we add it to the rear. To do this we have to assign the next pointer of the current rear to the new node, and after that we can redirect the rear pointer. We also have to handle the special case when the queue is empty – then the new node will be both front and rear, so we have to assign the front variable too.

```
datatype LinkedQueue:
    ...
    enqueue(elem):
        newRear = new Node(elem, null)
        if size == 0:
            front = newRear
        else:
            rear.next = newRear
        rear = newRear
        size = size + 1
```

Dequeueing from a queue is actually exactly the same as popping from a stack, where we use the front of the queue instead of the stack top. There is only additional one thing we must assure – if the final queue becomes empty, we have to delete the rear pointer too, otherwise it will point to a non-existing element.

```
datatype LinkedQueue:
    ...
    dequeue():
```

```
removed = front
front = removed.next
removed.next = null   // For garbage collection
size = size - 1
if size == 0:
    rear = null
return removed.elem
```

6.4.1 CASE STUDY: SORTING A LINKED LIST USING MERGESORT

We introduced Mergesort in Section 4.2, and then we showed how to sort an array. But Mergesort can also be used to sort linked lists, because it does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list.

Section 6.4.1

Read the rest online

6.5 Stacks implemented using arrays

The array-based stack contains a pre-allocated internal array, and the size of the stack.

```
datatype ArrayStack implements Stack:
    internalArray = new Array(1000)  // Internal array containing the stack elements
    size = 0                         // Size of stack, also index of the next free slot
```

Note that here we use an internal *capacity* of 1000 for the internal array, but we could use any positive value.

The only important design decision to be made is which end of the array should represent the top of the stack. It might be tempting to let the top be the first element in the array, i.e., the element at position 0. However, this is inefficient, because then we have to shift all elements in the array one position to the left or to the right, whenever we want to push to or pop from the stack.

Much better is to have the top be the *last element*, i.e. the element at position $n-1$ (if n is the number of elements). Then we don't have to shift around a lot of element, but instead just move the pointer to the left or the right.

The *size* variable refers to the last uninhabited cell in the array. So, to push an element onto the stack, we assign `internalArray[size]` and then increase the size.

```
datatype ArrayStack:
    ...
    push(elem):
        internalArray[size] = elem
        size = size + 1
```

To pop an element from the stack we do the reverse of pushing: first we decrease the size, then we remember the result in a temporary variable. After that we can clear the old top cell in the array and return the result.

```
datatype ArrayStack:
    ...
    pop():
        size = size - 1
        result = internalArray[size]
        internalArray[size] = null   // For garbage collection
        return result
```

Example: Implementing two stacks using one array

If you need to use two stacks at the same time, you can take advantage of the one-way growth of the array-based stack by using a single array to store two stacks. One stack grows inward from each end, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

6.6 Queues implemented using arrays

The array-based queue is somewhat tricky to implement effectively. A simple conversion of the array-based stack implementation is not efficient.

- If we let the queue front be the same as the stack top, i.e. the last element in the array, then we can easily enqueue new elements by simply moving the front pointer. But then the rear will be the first array element, and if we want to dequeue it we have to shift all the remaining elements to the left in the array. This is time-consuming and dequeueing therefore becomes linear, $O(n)$.
- If we instead let the front be the first array element, dequeueing becomes easy. But then we will instead have to shift all elements to be able to enqueue, again giving complexity $O(n)$.

A more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the beginning of the array. This means

that we need two variables, for the positions of the front and the rear elements in the array.

- When we enqueue an element, we put it in the empty cell to the right of the rear, and increase the rear variable.
- When we dequeue an element, the result is in the cell in the front position, and then we increase the front variable.

6.6.1 CIRCULAR QUEUES

Both enqueueing and dequeueing will increase the front and rear positions – the variables will never decrease. This causes the queue to *drift* within the array, and at some point the rear will hit the end of the array. We might get into a situation where the queue itself only occupies a few array cells, but the rear is still at the very end.

This "drifting queue" problem can be solved by pretending that the array is *circular*, meaning that we can go from the last array cell directly to the first. This is easily implemented through use of the *modulus* operator (usually denoted by %). Instead of just using $i + 1$ for the next position, we have to use $(i + 1) \% n$ (if n is the size of the array).

There remains one more subtle problem to the array-based queue implementation. How can we recognise when the queue is empty or full?

If the array has size n, then it can store queues of size 0 to n – therefore it can store $n + 1$ different queue lengths. But both when the queue is empty (size 0) and when it is full (size n), the *front* variable is one larger than *rear*. So, if $front = rear + 1$, is the queue empty or full?

One obvious solution is to keep an explicit count of the number of elements in the queue, i.e., using a special *size* variable. Another solution is to make the array be of size $n + 1$, and only allow n elements to be stored. A third solution is to set `front` and `rear` to -1 when the queue becomes empty. Which of these solutions to adopt is purely a matter of the implementor's taste in such affairs. Our choice here is to keep an explicit count of the number of elements, in a variable *size*, because this will make the code more similar to our other implementations.

```
datatype ArrayQueue implements Queue:
    internalArray = new Array(1000)  // Internal array containing the queue elements
    size = 0                         // Size of queue
    front = 0                        // Index of front element
    rear = -1                        // Index of rear element
    nextPosition(i):
        return (i + 1) % internalArray.size
```

Enqueueing and dequeueing When enqueueing, we increase the *rear* pointer (modulo the size of the internal array to make it circular).

```
datatype ArrayQueue:
    ...
    enqueue(x):
        rear = nextPosition(rear)   // Circular increment
        internalArray[rear] = x
        size = size + 1
```

When dequeueing, we increase the *front* pointer (modulo the size of the internal array).

```
class ArrayQueue:
    ...
    dequeue():
        result = internalArray[front]
        internalArray[front] = null   // For garbage collection
        front = nextPosition(front)   // Circular increment
        size = size - 1
        return result
```

6.7 Dynamic arrays

The problem with static array-based stacks and queues is that they have limited capacity. We get an error if we try to add new elements when the internal array is full.

To solve this problem we have to make the internal array *dynamic* – meaning that we increase the size of the array when it becomes full. (We should also shrink the array to save space whenever it contains too few elements.)

6.7.1 RESIZING THE INTERNAL ARRAY

How can we modify our data type to allow for any number of elements? Remember that array are always *fixed-size*, so we cannot change the array size. Instead we have to create new a larger array whenever the capacity is exceeded, and copy over all elements to the new one.

```
datatype ArrayStack:   // or ArrayQueue
    ...
    resizeArray(newCapacity):
        oldArray = internalArray   // Remember the old array
        internalArray = new Array(newCapacity)
        for i in 0 .. size-1:
            internalArray[i] = oldArray[i]
```

So, how large should the new internal array be? For now, let's ***double the size of the internal array*** when we need to resize, which means that we add the following if-clause to the methods push or enqueue:

```
if size >= internalArray.size
    resizeArray(internalArray.size * 2)
```

That's the only difference from the push method from **ArrayStack** from earlier. So the new dynamic push will look like this:

```
datatype ArrayStack:
    ...
    push(x):
        if size >= internalArray.size:
            resizeArray(internalArray.size * 2)
        internalArray[size] = x
```

As we will explain below, we don't have to double the size, but we can multiply by 3 or 1.5 or 1.1. The important point is that we don't add a constant number, but increase the size by a factor.

6.7.2 HOW MUCH TO INCREASE THE ARRAY SIZE

In the code above we doubled the size of the internal array whenever we needed to resize it. But we could have done something else, like:

- Triple the size
- Grow the size by 10%
- Grow the size by 100 elements
- Grow the size by 1 element

But which is best, and why?

There is a tradeoff: if we grow the array by a lot, we might waste memory. For example, immediately after we double the size, half of the array's capacity is unused, so we use twice as much memory as needed. On the other hand, if we grow the array by a small amount, we need to resize it more often.

We will explore these tradeoffs by looking at the performance of the following small program under different resizing strategies:

```
stack = new ArrayStack()
for i in 1 .. n:
    stack.push(i)
```

The program builds a stack of size n by repeatedly calling push. In this case, we could have used a static array-based stack of capacity n. So we would like

the dynamic array-based stack to have comparable performance to the static array-based stack. This means that the program ought to take *linear* time, $O(n)$.

Growing by a constant amount What happens if we only grow the internal array by 1 element when we resize it?

```
if size >= internalArray.size
    resizeArray(internalArray.size + 1)
```

Every time we call `push`, the internal array will be resized. Resizing the array takes linear time, because if the internal array has size n, it has to copy n elements from the internal array to the new array. To put it another way, the loop body `internalArray[i]=oldArray[i]` will be executed n times.

Now suppose we run the program above to create a list of n elements. Adding up all the calls to `resizeArray` that happen, how many times does an array element get copied from the internal array to the new array (that is, how many times does the statement `internalArray[i]=oldArray[i]` get executed)? The array size is initially 1, so we get the following calls:

`resizeArray(2)`	copying	1	element
`resizeArray(3)`	copying	2	elements
...			
`resizeArray(n-1)`	copying	$n-2$	elements
`resizeArray(n)`	copying	$n-1$	elements

In total, there are $1 + 2 + \ldots + (n-1)$ element copy operations, which is equal to $n(n-1)/2 = (n^2 - n)/2$. This means that the program takes *quadratic* time, $O(n^2)$, not linear!

Suppose for example that $n = 1,000,000$. Using the formula above, the number of times an array element gets copied is $999,999 \times 1,000,000 \ / \ 2 = 499,999,500,000$. If copying one array element takes 1 ns, then the program spends nearly 10 minutes just resizing the array!

What happens if we instead grow the array by 100 elements every time? You can try the calculation yourself, for say $n = 1,000,000$. What happens is that `resizeArray` gets called 100 times less often – so there 100 times fewer elements copied. But the runtime is still quadratic. When $n = 1,000,000$, the total number of elements copied is about $5,000,000,000$ – still far too many.

In short, ***growing the array size by a constant amount is bad***, because a loop that repeatedly adds to the array will take quadratic time.

Growing by a constant factor One way to think about the problem is: as the array gets bigger, resizing it gets more expensive. So, to make up for that, when the array is bigger we need to grow it by more, so that we don't have to resize as often. One way to do this is to always double the array size when it gets full. This turns out to work well!

Suppose that we run the example program with $n = 1000$, i.e. we add 1000 elements to the list. As before, the internal array initially has a size of 1. What calls to `resizeArray` happen, and how many elements get copied each time?

`resizeArray(2)`	copying	1	element
`resizeArray(4)`	copying	2	elements
`resizeArray(4)`	copying	4	elements
`resizeArray(8)`	copying	8	elements
...			
`resizeArray(256)`	copying	128	elements
`resizeArray(512)`	copying	256	elements
`resizeArray(1024)`	copying	512	elements

You can see that the array gets resized a whole lot at the beginning – but as it gets bigger, it gets resized much less often. We can read off how many elements get copied: $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 = 1023$.

Since the array starts from size 1 and always doubles, the array size is always a power of two. So to calculate the total number of elements copied, instead of adding up all the terms by hand, we can use the following formula:

$$2^0 + 2^1 + 2^2 + \ldots + 2^n \;=\; 2^{n+1} - 1$$

Suppose that we now choose $n = 1,000,000$. How many elements get copied? In this case the final array size will be $2^{20} = 1,048,576$. The array size will eventually grow from 2^{18} to 2^{19} to 2^{20} elements, with the final call to `resizeArray` copying 2^{19} elements. Using the formula above, the total number of elements copied is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{19} \;=\; 2^{20} - 1 \;=\; 1,048,575$$

Compared to when we grew the array by a fixed size of 1 element, this is 500,000 times fewer! So this in fact seems to be nice and efficient.

Let us now generalise to an arbitrary n. The worst case is when the final call to `add` has to resize the array – that happens when n is one more than a power of two, $n - 1 = 2^k$. In that case, the final call to `resizeArray` grows the array from 2^k to 2^{k+1}, copying 2^k elements. The total number of elements copied is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^k \;=\; 2^{k+1} - 1 \;=\; 2 \cdot 2^k - 1 \;=\; 2(n-1) - 1 \;=\; 2n - 3$$

In fact, we have just proved the following result.

Theorem: Array-doubling

When using the array-doubling strategy, calling `add` n times starting from an empty dynamic array list causes fewer than $2n$ elements to be copied.

Section 6.7.2

Read the rest online

In short, the overhead of using a dynamic array list is at most *two array elements copied per element that we add*. But copying an array element is an extremely cheap operation, so dynamic array lists implemented using array doubling have almost no overhead, compared to static array lists. In particular, the complexity of our example program is *linear*, just as we wanted.

6.7.3 RESIZING AN ARRAY-BASED QUEUE

When we resize the internal array, we cannot keep the positions of the elements. If the queue is wrapped around (i.e., if `rear` < `front`) then we might end up in a large gap in the middle of the queue.

Instead we reset the `front` and `rear` pointers so that we copy the first queue element to position 0 of the new array, the second to position 1, etc. Apart from that, the implementation is similar to the one for lists and queues.

```
datatype ArrayQueue:
    ...
    resizeArray(newCapacity):
        oldArray = internalArray   // Remember the old array
        internalArray = new Array(newCapacity)
        for pos in 0 .. size-1:
            oldPos = (pos + front) % internalArray.size
            internalArray[pos] = oldArray[oldPos]
        front = 0
        rear = size - 1
```

6.7.4 SHRINKING THE INTERNAL ARRAY

Now we know how to make a dynamic array stack (as well as a queue) that has room for any number of elements.

But the problem is that if we first build a large stack (or queue) with 1000's of elements, and then remove most of them, we will still have a large internal array where almost all cells are unused. So, let's resize the array also when

removing elements! When the array contains too many unused cells, we shrink it to half the size.

Now, it's important that we *don't* shrink the array when it's half full. Why is that? Let's consider the following sequence of additions and deletions:

```
push("A"); pop(); push("B"); pop(); push("C"); pop(); push("D"); pop(); ...
```

If we are unlucky and the array is full just before the sequence starts, then the first push will have to resize the array. Then when we pop that element, the list becomes less than half-full, and we have to resize again. Then the next push will resize, and the next pop will also resize. And so on… This will lead to a linear-time resize every time we push/pop, and so the final complexity will be linear (per operation) and not constant time. Which is not what we want.

How can we alleviate this? The solution is to wait even longer until we shrink the internal array! E.g., we can shrink the array (i.e., halve it), when it is only 1/3 full. Note that the factors 1/3 and 1/2 are not important, as explained earlier. The only thing that matters is that the minimum load factor (1/3) is *smaller* than the shrinking factor (1/2).

In summary, to get a dynamically shrinking stack (or queue), we can add the following lines right before the end of the `pop` method (or the `dequeue` method):

```
if size <= internalArray.size * 1/3:
    resizeArray(internalArray.size * 1/2)
```

That's the only difference from previous `pop` method (and `dequeue` for queues).

So the dynamic `pop` method will look like this:

```
datatype ArrayStack:
    ...
    pop():
        size = size - 1
        result = internalArray[size]
        internalArray[size] = null  // For garbage collection
        if size <= internalArray.size * 1/3:
            resizeArray(internalArray.size * 1/2)
        return result
```

6.8 Comparison of linked lists vs dynamic arrays

Now that you have seen two substantially different implementations for stacks and queues, it is natural to ask which is better. In particular, if you must implement a stack or a queue for some task, which implementation should you choose?

Time complexity All the basic operations for the array-based and linked list implementations take constant time, so from a time efficiency perspective, neither has a significant advantage.

Array-based lists are usually slightly faster because they can make use of the internal memory cache that modern computers have, but it depends on many factors – the programming language, the operating system, the processor, etc.

One little disadvantage with array-based lists is that the operations are only *amortised* constant time. We will discuss amortised time more in Section 7.2 later. But what it means in practice is that push, pos, enqueue and dequeue are only guaranteed to be constant time *on average* if we run many operations. Now and then (very rarely) the internal array will be resized, and then the operation might take longer time than usual.

This means that if we are implementing an application that has hard real-time constraints, a linked list might be a slightly better choice.

Memory usage Given a collection of elements to store, they take up some amount of space whether they are simple integers or large objects with many fields. Any container data structure like a stack, a queue or a list then requires some additional space to organise the elements being stored. This additional space is called overhead.

- Array-based lists have the disadvantage that the *capacity* of the internal array is larger than the actual size of the list. When the array has recently been reallocated, a substantial amount of space might be tied up in a largely empty array. This empty space is the overhead required by the array-based list.
- Linked lists on the other hand have the advantage that they only need space for the objects actually on the list. However, each list node needs to allocate memory for the pointer to the next node, and all of these pointers combined is the overhead required by the array-based list.

The amount of space required by a linked list is directly proportional to the number of elements n. Assuming that each list node takes up K bytes of memory, the full list will use Kn bytes. The amount of space required by an array-based list is in the worst case three times as much as n times the size of an array cell. (This worst case will arise when we remove a lot of elements from the list, because we wait until it is 1/3 full until we shrink the array). So assuming that one array cell takes up C bytes, the full array-based list will use at least Cn bytes, and at most $3Cn$ bytes.

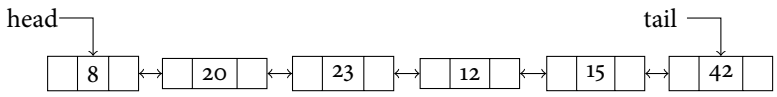
So, which one is the best? It depends on the size of the list nodes K, compared to the size of the array cells C. Array-based lists have the advantage that there

is no wasted space for an individual element. Linked lists require that an extra pointer for the `next` field be added to every list node. So the linked list has these `next` pointers as overhead. In many cases, K is 2–3 times as large as C, so they will be quite similar in size on average. But this depends on the programming language, the operating system, and perhaps other factors.

Note that these calculations exclude the memory used by the actual list elements, since the lists themselves only contain pointers to the elements! And in many cases, the objects themselves are much larger than the list nodes (or array cells).

6.9 Double-ended queues

The singly linked list (Section 6.3) allows for direct access from a list node only to the next node in the list. A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list. The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.



The most common reason to use a doubly linked list is because it gives an additional possibility to move both forwards and backwards in the list, and to efficiently add and remove elements from both ends.

Section 6.9

Read the rest online

6.10 General lists

We need some notation to show the contents of a list, so we will use the same angle bracket notation that is normally used to represent sequences. To be consistent with standard array indexing, the first position on the list is denoted as 0. Thus, if there are n elements in the list, they are given positions 0 through $n-1$ as $\langle\, a_0,\ a_1,\ \ldots,\ a_{n-1} \,\rangle$. The subscript indicates an element's position within the list. Using this notation, the empty list would appear as $\langle\, \rangle$.

What basic operations do we want our lists to support? Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove

elements from anywhere in the list. We should be able to gain access to any element's value, either to read it or to change it. Finally, we should be able to know the size of the list, and to iterate through the elements in the list – i.e., the list should be a Collection.

Section 6.10

Read the rest online

6.11 Priority queues

So far we have seen two ADTs that represent a collection of objects, and support adding and removing objects:

- Stacks, where the object removed is always the one *most recently* inserted (LIFO).
- Queues, where the object removed is always the one *first* inserted (FIFO).

There are many situations, both in real life and in computing applications, where we wish to choose the next "most important" from a collection of people, tasks, or objects. For example, doctors in a hospital emergency room often choose to see next the "most critical" patient rather than the one who arrived first. When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called jobs) ready to run. The next job selected is the one with the highest priority. Priority is indicated by a particular value associated with the job (and might change while the job remains in the wait list).

When a collection of objects is organised by importance or priority, we call this a priority queue. A priority queue supports the following operations:

- adding a new object to the priority queue
- removing the *smallest* object from the priority queue.

6.11.1 ADT FOR PRIORITY QUEUES

In chapter 9, we will see how to implement a priority queue so that both adding and removing the minimum take $O(\log n)$ time.

```
interface PriorityQueue of T extends Collection:
    add(x: T)          // Adds x to the priority queue.
    removeMin() -> T   // Removes and returns the minimum element.
    getMin() -> T      // Returns the minimum element, without removing it.
```

Note that this API assumes that the priority queue orders the elements in *ascending* order. There is also the possibility of ordering in descending order –

that kind of queue is called a *maximum priority queue*. If you have a minimum priority queue, it's straightforward to turn it into a maximum priority queue.

6.11.2 USE CASES FOR PRIORITY QUEUES

Now let's look at a couple of applications of priority queues.

Example: Sorting

We can use a priority queue to make an efficient sorting algorithm. To sort a list of items:

- First create an empty priority queue, and add all the items to it.
- Then repeatedly find and remove the smallest item. The items will come out in ascending order.

Here is an implementation of this algorithm in code:

```
function pqSort(A):
    pq = new PriorityQueue()
    for each item in A:
        pq.add(item)
    for i in 0 .. A.size-1:
        A[i] = pq.removeMin()
```

What is the time complexity of this algorithm? Well, for an input list of size n, the algorithm calls `add` n times and `removeMin` n times. In a binary heap, `add` and `removeMin` both take $O(\log n)$ time. Therefore, the total runtime is $O(n \log n)$ – as efficient as any of the sorting algorithms we have seen so far!

More examples can be found in the online version of the book.

Section 6.11.2

More examples online

6.11.3 IMPLEMENTING PRIORITY QUEUES USING SORTED LISTS

It is very easy to implement priority queues using sorted lists (either linked lists or dynamic arrays). Here are very basic ideas how to implement the operations:

```
datatype NaivePriorityQueue:
    list = new empty list
    add(x):
        insert x into list, keeping it sorted
    removeMin():
        remove the smallest element in list
```

If we decide to use a linked list, then we make sure it is always sorted with the smallest value first in the list. If we instead use a dynamic array, we have to keep it *reversely* sorted. The reason is the same as for stacks: it is efficient to remove elements from the *front* of a linked list, and from the *back* of a dynamic array. This means that `removeMin` will be a very efficient, constant time operation, just as `pop` for stacks.

However, inserting an element into a sorted list, keeping it sorted, is in the worst case linear, $O(n)$. Therefore, our sorting example in Section 6.11.2 becomes a quadratic implementation, $O(n^2)$, if we use this naive implementation of priority queues.

Later, in chapter 9, we will show a more efficient version of priority queues, based on *binary trees*.

6.12 Review questions

Section 6.12

Answer quiz online

This final section contains some review questions about the contents of this chapter.

CHAPTER 7

Algorithm analysis, part 3: Advanced theory

In this chapter, we move beyond basic runtime analysis to develop a richer understanding of algorithm performance. We consider not only execution time, but also memory usage. By examining these aspects, we gain deeper insights into the efficiency and practicality of advanced algorithms.

We start by examining space bounds and memory complexity, which are important when working with large datasets or constrained hardware. Next, we introduce amortised analysis, a method for evaluating the average performance of sequences of operations, where infrequent costly steps are balanced by many inexpensive ones. We then delve into recurrence relations, which are key to analysing the complexity of *recursive functions*, and demonstrate several techniques for solving them. Finally, we address algorithms influenced by multiple input parameters, learning how to express and analyse their complexity.

7.1 Space bounds

Besides time, space is the other computing resource that is commonly of concern to programmers. Just as computers have become much faster over the years, they have also received greater allotments of memory. Even so, the amount of available disk space or main memory can be significant constraints for algorithm designers.

The analysis techniques used to measure space requirements are similar to those used to measure time requirements. However, while time requirements are normally measured for an algorithm that manipulates a particular data structure, space requirements are normally determined for the data structure itself. The concepts of asymptotic analysis for growth rates on input size apply completely to measuring space requirements.

7.1.1 SPACE COMPLEXITY OF DATA STRUCTURES

Time complexity helps us to abstract away from hardware-specific details, constant factors and lower-order terms, so that we can focus on what has the most impact for large inputs. In the same way we want to abstract away from the actual memory usage in bytes, and instead focus on how the memory used by a data structure depends on the data size.

Example: Arrays and linked lists

What are the space requirements for an array of n integers? If each integer requires k bytes, then the array requires kn bytes, which is in $O(n)$.

What are the space requirements for a linked list of n integers? Each linked node requires k bytes for the integer, plus (say) k additional bytes for the pointer to the next node. Therefore the linked list requires $2kn$ bytes, which is also in $O(n)$.

A data structure's primary purpose is to store data in a way that allows efficient access to those data. To provide efficient access, it may be necessary to store additional information about where the data are within the data structure. For example, each node of a linked list must store a pointer to the next value on the list. All such information stored in addition to the actual data values is referred to as overhead. Ideally, overhead should be kept to a minimum while allowing maximum access. The need to maintain a balance between these opposing goals is what makes the study of data structures so interesting.

Section 7.1.1

Read the rest online

7.1.2 SPACE COMPLEXITY OF ALGORITHMS

We are not only interested in knowing how much memory a data structure will use, but also what the space complexity of an *algorithm* is. When we analyse space usage of algorithms, we are usually only interested in the *additional* space that the algorithm uses during execution.

Let's say that an algorithm is *in-place* if it only uses constant additional space, $O(1)$. For example, insertion sort is in-place, because it only allocates a constant number of variables to complete.

Example: Mergesort

How much additional space does mergesort use? Mergesort is a divide-and-conquer algorithm that calls itself recursively, halving the size of the problem in each call. The step that uses additional memory is the merging process, where we have to allocate a new array that we can fill with the merged values.

In the first level we have to allocate one array of size n. In the second level we allocate two arrays, each of size $n/2$. In the third level we allocate $2^2 = 4$ arrays, each of size $n/2^2 = n/4$. Continuing down we see that in level k we allocate 2^k arrays, each of size $n/2^k$.

As you can see, each level uses up an additional $O(n)$ space, because $2^k \cdot n/2^k = n$. And, since we already know that mergesort continues for $\log n$ levels, we get an additional space usage of $O(n \log n)$.

But it is possible to improve the space usage of mergesort. Instead of allocating new arrays at each level, we can create one single additional array of size n and then use only that auxilliary array. To make this work we have to change the implementation somewhat, and this can be done in several ways. The most common solution is called bottom-up mergesort, and it has an additional space usage of $O(n)$.

So, mergesort is not an in-place algorithm because it uses at least linear additional space.

Section 7.1.2

Read the rest online

7.1.3 SPACE/TIME TRADEOFF

One important aspect of algorithm design is referred to as the space/time tradeoff principle. The space/time tradeoff principle says that one can often achieve a reduction in time if one is willing to sacrifice space or vice versa. Many programs can be modified to reduce storage requirements by "packing" or encoding information. "Unpacking" or decoding the information requires additional time. Thus, the resulting program uses less space but runs slower. Conversely, many programs can be modified to pre-store results or reorganise information to allow faster running time at the expense of greater storage requirements. Typically, such changes in time and space are both by a constant factor.

Section 7.1.3

Read the rest online

7.2 Amortised analysis

This section presents the concept of amortised analysis, which is the analysis for a series of operations taken as a whole. In particular, amortised analysis allows us to deal with the situation where the worst-case cost for n operations is less than n times the worst-case cost of any one operation. Rather than focusing on the individual cost of each operation independently and summing them, amortised analysis looks at the cost of the entire series and "charges" each individual operation with a share of the total cost.

The standard example for amortised analysis is dynamic arrays which were introduced in Section 6.7. In that section we gave an informal argument why it is important to grow the array in the right way. If we do it by doubling the array size, we get *amortised* constant time for all basic operations, but if we do it in the wrong way we get linear time operations in the worst case.

Dynamic arrays are such an important example for amortised analysis that we will devote the whole of next section to them. But before that we will explain the concepts and give some other examples.

Example: Multipop on stacks

Assume that we want to add a new operation *multipop* on stacks, where *multipop*(k) will pop the k topmost elements off the stack. The operation is implemented in the straightforward by simply repeating a single *pop* operation k times.

What is the time complexity of this new operation? Since we're repeating the constant-time *pop* operation k times, we get a time complexity of $O(k)$. And the worst case of this is when k is as large as possible, i.e., the stack size n. So the worst-case complexity for *multipop* is linear in the stack size $O(n)$.

This is quite correct, the worst-case complexity of a single call to *multipop* is linear in n. But what is the complexity of executing a large number of stack operations in sequence?

Let's say that we start from an empty stack and execute a sequence of n *push* operations and n *multipop* operations. Using our analysis above, the whole sequence of $2n$ operations will have worst-case complexity $n \cdot O(1) + n \cdot O(n) = O(n + n^2) = O(n^2)$. Since we performed $2n$ operations, we get an average complexity per operation of $O(n^2)/2n = O(n)$. This analysis is unreasonably pessimistic. Clearly it is not really possible to pop n elements each time *multipop* is called. Analysis that focuses on single

operations cannot deal with this global limit, and so we turn to amortised analysis to model the entire series of operations.

We can reason like this instead: n elements are pushed to the stack, and each of these elements can only be popped once. The sum of costs for all calls to *multipop* can never be more than the total number of elements that has been pushed on the stack, which is n. This means that the total complexity of our n calls to *multipop* must be in $O(n)$. This is the same complexity as the n calls to *pop*, so our total complexity cannot be worse than $O(n) + O(n) = O(n)$.

Therefore the average worst-case complexity per operation must be $O(n)/n = O(1)$, i.e., constant.

In the *multipop* example we got two different complexities for the *multipop* operation: first we found that it is linear in the stack size, $O(n)$, but when averaging over a sequence of operations we found that it is constant, $O(1)$. So, which complexity is the right one?

Actually, both are correct. The worst-case complexity for a single call to *multipop* is linear in the worst case, but the *amortised* complexity is constant. This means that when executing *multipop* many many times, it will behave as it is constant time: some individual calls might take longer time, but this is balanced out by other calls that are fast.

In the example we used a very hand-waving, informal argument, but the underlying idea is the concept of a potential. In the potential method we let cheap operations "save up" some additional work that can be used by more expensive operations later. In the example, we let each *push* save an extra operation "for later", which is then used by *multipop*. In a way we can say that each *push* takes 2 time units instead of one, and this extra time unit is saved so that *multipop* can make use of it. These storage of "for later" operations is called the *potential*.

Section 7.2

Read the rest online

7.3 Case study: Analysing dynamic arrays

Dynamic arrays, as discussed in Section 6.7, are a flexible data structure that efficiently manages collections of elements whose size may change over time. Unlike fixed-size arrays, dynamic arrays automatically resize themselves to accommodate additional elements, making them especially useful when the total number of elements is not known in advance.

In this case study, we will analyse the time and memory complexity of dy-

namic arrays, with a particular focus on the operation of adding an element. This operation is central to understanding dynamic arrays because:

- Retrieving an element by index is identical to regular arrays and operates in constant time.
- Removing an element involves similar considerations as inserting, particularly regarding shifting elements, so we will not cover it separately here.

Complexity of adding an element to a dynamic array Let us first consider adding an element at the end of a dynamic array. In most cases, this operation is efficient, as it simply places the new element in the next available position in the internal array and takes constant time. However, when the array is full, it must resize itself to accommodate the new element. This resizing involves allocating a new array, typically twice the size of the original, copying all existing elements to this new array, and then inserting the new element. This step is more expensive and takes linear time in the number of elements. The amortised time complexity of appending arises from this pattern: although individual operations may occasionally be costly, the overall cost of performing many appends remains low, averaging out to constant time per operation.

To illustrate this, consider a dynamic array that starts with a capacity of 1. When you append the first element, it fits perfectly. The next time you append, the array is full, so it resizes to a capacity of 2, copying the existing element. The next append fills this new capacity, leading to another resize to 4, and so on. Each time the array resizes, it doubles its capacity and copies all existing elements to the new array. This means that the number of elements copied during a resize operation grows as follows:

- 1 element copied when resizing from 1 to 2,
- 2 elements copied when resizing from 2 to 4,
- 4 elements copied when resizing from 4 to 8,
- and so on.

The total number of elements copied during all these resizes can be summed up as follows:

$$1 + 2 + 4 + \ldots + n = 2n - 1$$

where n is the final size of the array. This means that the total cost of copying elements during all resizes is proportional to the final size of the array n. So, while a single append operation may take linear time due to resizing, the average time per append operation across many appends is *constant*. This is because the expensive operations (the resizes) are infrequent compared to the many

inexpensive appends that do not require resizing. Thus, the *amortised time complexity* for appending an element to a dynamic array is $O(1)$, meaning that on average, each append operation takes constant time.

The above analysis assumes that we add an element at the end of the dynamic array. If we consider adding an element at the beginning or in the middle of the array, the situation changes significantly. In these cases, the dynamic array must shift existing elements to make space for the new element. This shifting operation takes linear time in the number of elements, as each element must be moved one position to the right. Therefore, inserting an element at the beginning or in the middle of a dynamic array has a time complexity of $O(n)$, where n is the number of elements in the array. This is an important point to consider when choosing a data structure for a specific use case.

Memory complexity of dynamic arrays The memory complexity of dynamic arrays is also an important aspect to consider. When a dynamic array resizes, it typically allocates a new array that is *larger* than the current one. The memory used by a dynamic array is proportional to its capacity, not just the number of elements it currently holds. Thus, the memory complexity of a dynamic array is $O(n)$, where n is the current number of elements in the array. This can lead to situations where the dynamic array uses more memory than strictly necessary.

7.4 Recurrence relations

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A recurrence relation defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

$$\begin{aligned} n! &= n \cdot (n-1)! \text{ for } n > 1 \\ 1! = 0! &= 1 \end{aligned}$$

Another standard example of a recurrence is the Fibonacci sequence:

$$\begin{aligned} \mathrm{Fib}(n) &= \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \text{ for } n > 2 \\ \mathrm{Fib}(1) = \mathrm{Fib}(2) &= 1 \end{aligned}$$

From this definition, the first seven numbers of the Fibonacci sequence are

$$1, 1, 2, 3, 5, 8, \text{ and } 13.$$

Notice that this definition contains two parts: the general definition for $\text{Fib}(n)$ and the base cases for $\text{Fib}(1)$ and $\text{Fib}(2)$. Likewise, the definition for factorial contains a recursive part and base cases.

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by a recursive version of the factorial function for an input of size n will be zero when $n = 0$ or $n = 1$ (the base cases), and it will be one plus the cost of calling itself on a value of $n - 1$. This can be defined using the following recurrence:

$$\begin{aligned} T(n) &= T(n-1) + 1 \text{ for } n > 1 \\ T(0) = T(1) &= 0 \end{aligned}$$

As with summations, we typically wish to replace the recurrence relation with a closed-form solution. One approach is to expand the recurrence by replacing any occurrences of T on the right-hand side with its definition:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + (1 + T(n-2)) \\ &= 1 + (1 + (1 + T(n-3))) \\ &= 1 + (1 + (1 + (1 + (\cdots + 0)))) \end{aligned}$$

So, the closed form solution of $T(n) = T(n-1) + 1$ can be modeled by the summation $\sum_1^n 1 = n$.

A slightly more complicated recurrence is

$$\begin{aligned} T(n) &= T(n-1) + n \\ T(1) &= 1 \end{aligned}$$

Again, we will use expansion to help us find a closed form solution:

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1 + T(n-2)) \\ &= n + (n-1 + (n-2 + T(n-3))) \\ &= n + (n-1 + (n-2 + (n-3 + (\cdots + 1)))) \end{aligned}$$

So, the closed form solution of $T(n) = T(n-1) + n$ can be modeled by the summation $\sum_1^n i$. This is also a standard summation that we know, with the solution $T(n) = n(n+1)/2$.

A more complicated example is the standard Mergesort. This takes a list of size n, splits it in half, performs Mergesort on each half, and finally merges the two sublists in n steps. The cost for this can be modeled as

$$\begin{aligned} T(n) &= 2\,T(n/2) + n \\ T(1) &= 1 \end{aligned}$$

In other words, the cost of the algorithm on input of size n is two times the cost for input of size $n/2$ (due to the two recursive calls to Mergesort) plus n (the time to merge the sublists together again).

There are many approaches to solving recurrence relations, and we briefly consider three here. The first is an estimation technique: Guess the upper and lower bounds for the recurrence, use induction to prove the bounds, and tighten as required. The second approach is to expand the recurrence to convert it to a summation and then use summation techniques. The third approach is to take advantage of already proven theorems when the recurrence is of a suitable form. In particular, typical divide-and-conquer algorithms such as Mergesort yield recurrences of a form that fits a pattern for which we have a ready solution.

Section 7.4

Read the rest online

7.5 Multiple parameters

Sometimes the proper analysis for an algorithm requires multiple parameters to describe the cost. To illustrate the concept, consider an algorithm to compute the rank ordering for counts of all pixel values in a picture. Pictures are often represented by a two-dimensional array, and a pixel is one cell in the array. The value of a pixel is either the code value for the colour, or a value for the intensity of the picture at that pixel. Assume that each pixel can take any integer value in the range 0 to $C-1$. The problem is to find the number of pixels of each colour value and then sort the colour values with respect to the number of times each value appears in the picture. Assume that the picture is a rectangle with P pixels. A pseudocode algorithm to solve the problem follows.

```
// Initialise the counts:
for i in 0 .. C-1:
    count[i] = 0
// Increment the pixel value count for each of the pixels:
for i in 0 .. P-1:
    count[value(i)] = count[value(i)]+1
// Sort the pixel value counts:
sort(count)
```

In this example, `count` is an array of size `C` that stores the number of pixels for each colour value. Function `value(i)` returns the colour value for pixel i.

The time for the first `for` loop (which initialises `count`) is based on the number of colours, C. The time for the second loop (which determines the number of pixels with each colour) is $O(P)$. The time for the final line, the call to `sort`, depends on the cost of the sorting algorithm used. We will assume that the sorting algorithm has cost $O(P \log P)$ if P items are sorted, thus yielding $O(P \log P)$ as the total algorithm cost.

Is this a good representation for the cost of this algorithm? What is actually being sorted? It is not the pixels, but rather the colours. What if C is much smaller than P? Then the estimate of $O(P \log P)$ is pessimistic, because much fewer than P items are being sorted. Instead, we should use P as our analysis variable for steps that look at each pixel, and C as our analysis variable for steps that look at colours. Then we get $O(C)$ for the initialisation loop, $O(P)$ for the pixel count loop, and $O(C \log C)$ for the sorting operation. This yields a total cost of $O(P + C \log C)$.

Why can we not simply use the value of C for input size and say that the cost of the algorithm is $O(C \log C)$? Because, C is typically much less than P. For example, a picture might have 1000×1000 pixels and a range of 256 possible colours. So, P is one million, which is much larger than $C \log C$. But, if P is smaller, or C larger (even if it is still less than P), then $C \log C$ can become the larger quantity. Thus, neither variable should be ignored.

CHAPTER 8

Trees

Tree structures enable efficient access and efficient update to large collections of data. Binary trees in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include describing mathematical expressions and the syntactic elements of computer programs (using expression trees, see Section 8.3.1), prioritising jobs (using binary heaps, see Section 9.2), or organising the information needed to drive data compression algorithms (using Huffman coding, see Section 9.4).

This chapter covers terminology used for discussing binary trees (Section 8.1), tree traversals (Section 8.4), approaches to implementing tree nodes (Section 8.3), and various examples of binary trees. The chapter concludes by discussing non-binary trees, i.e., trees with more (or less) than two children (Section 8.6).

8.1 Binary trees

A binary tree is made up of a finite set of elements called nodes. This set either is empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are children of the root. There is an edge from a node to each of its children, and a node is said to be the parent of its children.

If $n_1, n_2, ..., n_k$ is a sequence of nodes in the tree such that n_i is the parent of $n_i + 1$ for $1 \leq i < k$, then this sequence is called a path from n_1 to n_k. The length of the path is $k - 1$. If there is a path from node R to node M, then R is an ancestor of M, and M is a descendant of R. Thus, all nodes in the tree are descendants of the root of the tree, while the root is the ancestor of all nodes. The depth of a node M in the tree is the length of the path from the root of the tree to M. The height of a tree is the depth of the deepest node in the tree. All nodes of depth d are at level d in the tree. The root is the only node at level 0,

and its depth is 0. A leaf node is any node that has two empty children. An internal node is any node that has at least one non-empty child.

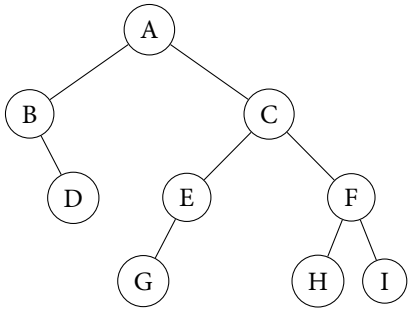


Figure 8.1 An example binary tree

Figure 8.1 above illustrates the various terms used to identify parts of a binary tree. Node *A* is the root, and nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 2 of the tree; node *A* is at level 0. The edges from *A* to *C* to *E* to *G* form a path of length 3. Nodes *D*, *G*, *H*, and *I* are leaves. Nodes *A*, *B*, *C*, *E*, and *F* are internal nodes. The depth of *I* is 3. The height of this tree is 3.

Figure 8.2 below illustrates an important point regarding the structure of binary trees. Because *all* binary tree nodes have two children (one or both of which might be empty), the two binary trees (a) and (b) in the figure are *not* the same.

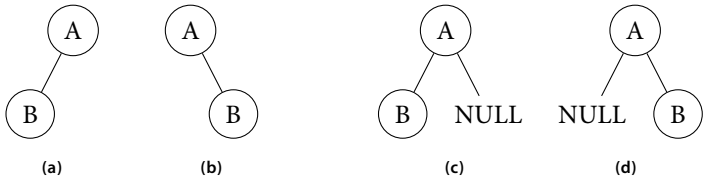


Figure 8.2 Two different binary trees: (a) the root has a non-empty left child; (b) the root has a non-empty right child; and (c) the same tree as (a), with the missing right child made explicit; (d) the same tree as (b), with the missing left child made explicit

Two restricted forms of binary tree are sufficiently important to warrant special names. Each node in a full binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf. A complete binary tree has a restricted

shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level d are completely full. The bottom level has its nodes filled in from the left side.

Figure 8.3 below illustrates the differences between full and complete binary trees. There is no particular relationship between these two tree shapes; that is, the tree (a) is full but not complete while the tree (b) is complete but not full. The binary heap (Section 9.2) is an example of a complete binary tree. The Huffman coding tree (Section 9.4) is an example of a full binary tree.

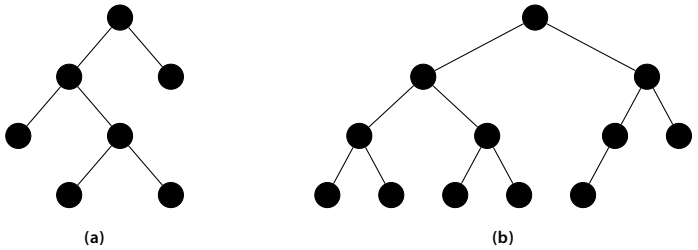


Figure 8.3 Examples of full and complete binary trees: (a) is full but not complete; (b) is complete but not full

Note: While these definitions for full and complete binary tree are the ones most commonly used, they are not universal. Because the common meaning of the words "full" and "complete" are quite similar, there is little that you can do to distinguish between them other than to memorise the definitions. Here is a memory aid that you might find useful: "Complete" is a wider word than "full", and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

8.1.1 BINARY TREES ARE RECURSIVE DATA STRUCTURES

A recursive data structure is a data structure that is partially composed of smaller or simpler instances of the same data structure. For example, linked lists and binary trees can be viewed as recursive data structures. A list is a recursive data structure because a list can be defined as either (1) an empty list or (2) a node followed by a list. A binary tree is typically defined as (1) an empty tree or (2) a node pointing to two binary trees, one its left child and the other one its right child.

One way to think about recursion is to see it as *delegation*: Suppose you want to compute the sum of the values stored in a binary tree. And since you

are a lazy person you don't want to do most of the work yourself, so you ask two friends to help you.

- The first friend will take the left subtree to sum it.
- The second friend will take the right subtree to sum it.
- The only thing you have to do is to sum the values that got from your friends.

You don't need to think about how your friends (the recursive calls) calculated their sums, as long as you accept that they are correct.

8.2 Case study: Full binary trees

Section 8.2

Read the rest online

This section discusses one particular kind of binary trees, where all nodes have either two children or no children, and how you can reason about them.

8.3 Implementing binary trees

In this section we examine one way to implement binary tree nodes. By definition, all binary tree nodes have two children, though one or both children can be empty. Binary tree nodes typically contain a value field, with the type of the field depending on the application. The most common node implementation includes a value field and pointers to the two children. Figure 8.4 is an illustration of how the tree from Figure 8.1 looks like, where the child pointers are shown explicitly.

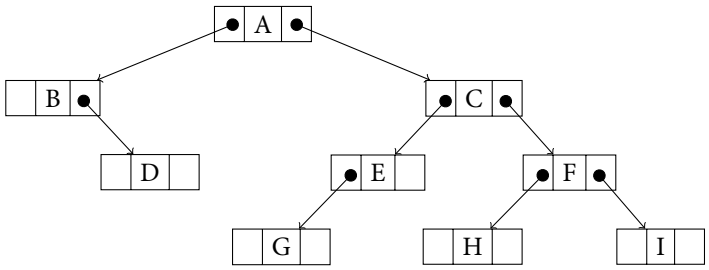


Figure 8.4 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value

Here is a simple implementation for binary tree nodes, which can store one single element in each node. Every `BinaryNode` object also has two pointers, one to its left child and another to its right child.

```
datatype BinaryNode of T:
    elem: T            // Element for this node.
```

```
    left: BinaryNode    // Pointer to left child.
    right: BinaryNode   // Pointer to right child.

    isLeaf() -> bool:
        // Return true if a leaf node, false otherwise.
        return this.left is null and this.right is null
```

We define a *leaf* to be a node with no children – i.e., where both childen pointers point to nothing.

Some programmers find it convenient to add a pointer to the node's parent, allowing easy upward movement in the tree. Using a parent pointer is somewhat analogous to adding a link to the previous node in a doubly linked list. In practice, the parent pointer is almost always unnecessary and adds to the space overhead for the tree implementation. It is not just a problem that parent pointers take space. More importantly, many uses of the parent pointer are driven by improper understanding of recursion and so indicate poor programming. If you are inclined toward using a parent pointer, consider if there is a more efficient implementation possible.

Binary trees Our final datatype for binary trees is in fact very similar to the linked lists that we introduced in Section 6.3 – we need a reference to the root node and the total size of the tree:

```
datatype BinaryTree:
    root: BinaryNode = null
    size: Int = 0
```

8.3.1 DIFFERENTIATING BETWEEN INTERNAL NODES AND LEAVES

An important decision in the design of a pointer-based node implementation is whether the same class definition will be used for leaves and internal nodes. Using the same class for both will simplify the implementation, but might be an inefficient use of space. Some applications require data values only for the leaves. Other applications require one type of value for the leaves and another for the internal nodes. Examples include the binary trie, the PR Quadtree, the Huffman coding tree, and the expression tree illustrated by the following figure. By definition, only internal nodes have non-empty children. If we use the same node implementation for both internal and leaf nodes, then both must store the child pointers. But it seems wasteful to store child pointers in the leaf nodes. Thus, there are many reasons why it can save space to have separate implementations for internal and leaf nodes.

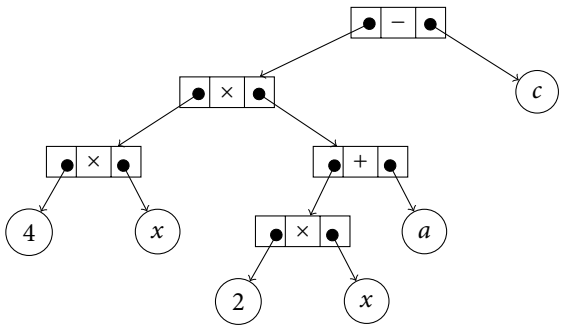


Figure 8.5 An example of an expression tree for $4x(2x + a) - c$

As an example of a tree that stores different information at the leaf and internal nodes, consider the expression tree illustrated by Figure 8.5. The expression tree represents an algebraic expression composed of binary operators such as addition, subtraction, multiplication, and division. Internal nodes store operators, while the leaves store operands. The tree of the figure represents the expression $4x(2x + a) - c$. The storage requirements for a leaf in an expression tree are quite different from those of an internal node. Internal nodes store one of a small set of operators, so internal nodes could store a small code identifying the operator such as a single byte for the operator's character symbol. In contrast, leaves store variable names or numbers, which is considerably larger in order to handle the wider range of possible values. At the same time, leaf nodes need not store child pointers.

Object-oriented languages allow us to differentiate leaf from internal nodes through the use of a class hierarchy. A base class provides a general definition for an object, and a subclass modifies the base class to add more detail. We will not discuss further how to implement different kind of tree nodes more in this book, but will just assume that all nodes are of the same class.

Section 8.3.2

Read the rest online

8.3.2 SPACE REQUIREMENTS

In this subsection we present techniques for calculating the amount of overhead required by a binary tree, based on its node implementation.

8.4 Traversing a binary tree

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a traversal. Any

traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.

8.4.1 PREORDER, POSTORDER AND INORDER

There are three main strategies for traversing a binary tree, depending on when we want to visit a node in relation to its children (and all their subtrees).

Preorder traversal Visit each node only *before* we visit its children (and their subtrees). For example, this is useful if we want to create a copy of a tree. First we create a copy of the current node, and then we can directly copy its subtrees into the new node.

Postorder traversal Visit each node only *after* we visit its children (and their subtrees). This is useful when we want to delete a tree to free storage space. Before we can delete the current node, we should delete all its children (and its children's children and so on).

Inorder traversal First visit the left child (including its entire subtree), then visit the node, and finally visit the right child (including its entire subtree). If the tree is a binary search tree, then we can use inorder traversal to list all values in increasing order.

Table 8.1 shows in which order the nodes in the example tree from Figure 8.1 are visited, depending on the traversal strategy.

Table 8.1 Visiting order for the example tree in Figure 8.1

Traversal	Visiting order	When is the root visited?
Preorder	**A, B, D, C, E, G, F, H, I**	A is the first visited node
Postorder	**D, B, G, E, H, I, F, C, A**	A is the very last node to visit
Inorder	**B, D, A, G, E, C, H, F, I**	after visiting the left subtree (B, D)

8.4.2 IMPLEMENTATION

A traversal routine is naturally written as a recursive function. The initial call to the traversal function passes in a pointer to the root node of the tree. The traversal function visits the node and its children (if any) in the desired order. Here is a very generic pseudocode for all kinds of traversal:

Section 8.4.2

Read the rest online

```
function traverse(node):
    if node is not null:        // Only continue if this is a tree
        visitPreorder(node)     // Visit root node (PREORDER traversal)
        traverse(node.left)     // Process all nodes in left subtree
        visitInorder(node)      // Visit root node (INORDER traversal)
        traverse(node.right)    // Process all nodes in right subtree
        visitPostorder(node)    // Visit root node (POSTORDER traversal)
```

8.5 Iteration, recursion, and information flow

Handling information flow in a recursive function can be a challenge. In any given function, we might need to be concerned with either or both of:

Section 8.5

Read the rest online

1. Passing down the correct information needed by the function to do its work,
2. Returning (passing up) information to the recursive function's caller.

Any given problems might need to do either or both. This section contains some examples and exercises.

8.6 General trees

Many organisations are hierarchical in nature, such as the military and most businesses. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, and so on. If we wanted to model this company with a data structure, it would be natural to think of the president in the root node of a tree, the vice presidents at level 1, and their subordinates at lower levels in the tree as we go down the organisational hierarchy.

Because the number of vice presidents is likely to be more than two, this company's organisation cannot easily be represented by a binary tree. We need instead to use a tree whose nodes have an arbitrary number of children. Unfortunately, when we permit trees to have nodes with an arbitrary number of children, they become much harder to implement than binary trees. We consider such trees in this chapter. To distinguish them from binary trees, we use the term general tree.

In this section we will examine general tree terminology and define a basic class for general trees.

8.6.1 DEFINITIONS AND TERMINOLOGY

A tree **T** is a finite set of one or more nodes such that there is one designated node R, called the root of **T**. If the set $(\mathbf{T} - \{R\})$ is not empty, these nodes

are partitioned into $n > 0$ disjoint sets $\mathbf{T}_0$, $\mathbf{T}_1$, ..., $\mathbf{T}_{n-1}$, each of which is a tree, and whose roots $R_1, R_2, ..., R_n$, respectively, are children of R. The subsets $\mathbf{T}_i (0 \leq i < n)$ are said to be subtrees of $\mathbf{T}$. These subtrees are ordered in that $\mathbf{T}_i$ is said to come before $\mathbf{T}_j$ if $i < j$. By convention, the subtrees are arranged from left to right with subtree $\mathbf{T}_0$ called the leftmost child of R. A node's out degree is the number of children for that node. A forest is a collection of one or more trees. Figure 8.6 presents further tree notation generalised from the notation for binary trees.

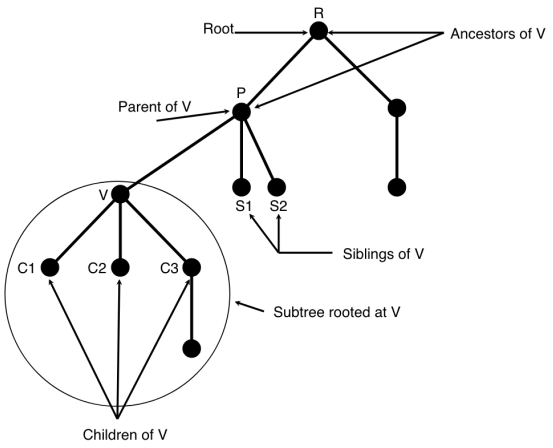


Figure 8.6 Notation for general trees. Node P is the parent of nodes V, $S1$, and $S2$. Thus, V, $S1$, and $S2$ are children of P. Nodes R and P are ancestors of V. Nodes V, $S1$, and $S2$ are called siblings. The oval surrounds the subtree having V as its root.

Each node in a tree has precisely one parent, except for the root, which has no parent. From this observation, it immediately follows that a tree with n nodes must have $n - 1$ edges because each node, aside from the root, has one edge connecting that node to its parent.

8.6.2 ADT FOR GENERAL TREES

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Any implementation must be able to initialise a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of binary tree nodes, this was done by providing instance variables for the left and right child pointers. Unfortunately, because we do not know in advance how

many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.

One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for node implementations in favour of an array-based approach, because these functions favour random access to a list of children.

An alternative is to provide access to a **List** of the children pointers. This list can be an array-based list or a linked list or even a dynamic function generating the children on demand. The only thing we will assume is that it follows the **List** ADT, as described in the section about [the List ADT].

```
// General tree nodes
datatype GTNode of T:
    elem: T                    // This is the value, just as for binary nodes
    children: List of GTNode   // All children of the node
```

8.6.3 TRAVERSING A GENERAL TREE

There are three traditional tree traversals for binary trees: preorder, postorder, and inorder (see Section 8.4). For general trees, preorder and postorder traversals are defined with meanings similar to their binary tree counterparts. Preorder traversal of a general tree first visits the root of the tree, then performs a preorder traversal of each subtree from left to right. A postorder traversal of a general tree performs a postorder traversal of the root's subtrees from left to right, then visits the root. Inorder traversal does not have a natural definition for the general tree, because there is no particular number of children for an internal node. An arbitrary definition – such as visit the leftmost subtree in inorder, then the root, then visit the remaining subtrees in inorder – can be invented. However, inorder traversals are generally not useful with general trees.

To perform a preorder traversal, it is necessary to visit each of the children for a given node (say *R*) from left to right. This is accomplished by starting at R's leftmost child (call it *T*). From *T*, we can move to *T*'s right sibling, and then to that node's right sibling, and so on.

Using the General Tree class shown above, here are implementations to process the nodes of a general tree in preorder and in postorder. The code is very simple, but this is because we defer all the complexity to the underlying **List** implementation of the children.

```
function preorder(node):
    process(node)
    for each child in node.children:
        preorder(child)

function postorder(node):
    for each child in node.children:
        postorder(child)
    process(node)
```

8.6.4 SEQUENTIAL TREE REPRESENTATIONS

Next we consider a fundamentally different approach to implementing trees. The goal is to store a series of node values with the minimum information needed to reconstruct the tree structure.

Section 8.6.4

Read the rest online

8.7 Review questions

This final section contains some review questions about the contents of this chapter.

Section 8.7

Answer quiz online

CHAPTER 9

Priority queues and heaps

In Section 6.11 we introduced priority queues and showed how to implement them using sorted lists. However, this is not a very good implementation, because inserting elements into the sorted list is linear, $O(n)$, in the worst case.

In this chapter we will see how to use binary trees to implement a much more efficient version of priority queues.

Recall that the API for a *minumum* priority queue only consists of the following methods:

```
interface PriorityQueue of T extends Collection:
    add(x: T)          // Adds x to the priority queue.
    removeMin() -> T   // Removes and returns the minimum element.
    getMin() -> T      // Returns the minimum element, without removing it.
```

Also recall that there is a mirrored variant, called a *maximum* priority queue, with the only difference being that it uses the methods `removeMax` and `getMax` instead of the previous ones.

In this chapter we will switch between talking about minimum and maximum priority queues. Note that it is always easy to convert between the two kinds of implementations: just replace any comparison (e.g., $<$ or $\leq$) with its counterpart (e.g., $>$ or $\geq$, respectively).

In general, we will try to use the term *priority* instead of minimum or maximum: in a minumum priority queue, the smallest element is the one with the highest priority, wheras in a maximum priority queue, it's the largest element.

9.1 Implementing priority queues using binary trees

A normal linear data structure, such as a linked list or dynamic array, cannot implement a priority queue efficiently. This is because either insertion or removal will take linear time, $O(n)$, in the worst case.

- If we use unordered lists, then searching for the element with the highest priority will be linear.

- If we instead use sorted lists, then inserting an element into that list will be linear.

In this we show how to use binary trees instead of lists, to implement priority queues.

9.1.1 THE HEAP PROPERTY

The main idea is that we always keep the highest priority element as the root of the tree. This means that we have constant access to it, so the method `getMin` will always be constant time.

In general, a heap is a tree which satisfies the *heap property*:

- Every tree node has at least as high priority as all its children

One immediate consequence of this property is that the root in a heap always contains the element with the highest priority.

Now, how do we implement these heaps so that they will always be efficient? There are actually several possible ways to do this, each having their own advantages and disadvantages. Some heap data structurues are *leftist heaps*, *skew heaps*, *Fibonacci heaps*, *2-3 heaps*, *binomial heaps*, and many more.

In this chapter we will focus on the most basic heap implementation, *binary heaps*. What makes this implementation special is that it stores the tree as an array, which makes it very space-efficient.

9.2 Binary heaps

This section presents the binary heap data structure. In addition to beaing a *heap*, meaning that it satisfies the heap property, it is also a complete binary tree.

Recall that complete binary trees have all levels except the bottom filled out completely, and the bottom level has all of its nodes filled in from left to right. Thus, a complete binary tree of n nodes has only one possible shape. You might think that a complete binary tree is such an unusual occurrence that there is no reason to develop a special implementation for it. However, it has two very nice properties:

- its height is *logarithmic* in the number of nodes
- it can be stored in an array, so it is very *space-efficient*

9.2.1 REPRESENTING COMPLETE BINARY TREES AS ARRAYS

From the full binary tree theorem, we know that a large fraction of the space in a typical binary tree node implementation is devoted to structural overhead, not to storing data. Here we present a simple, compact implementation for complete binary trees. First, note that a complete binary tree of n nodes has only one possible shape. This means that we can store the nodes directly in an array, without having to include the children pointers. Instead we can use simple calculations to find the array indices of the children or the parent of a given node.

We begin by assigning numbers to the node positions in the complete binary tree, level by level, from left to right as shown in Figure 9.1. Note that we start by assigning the root the number 0.

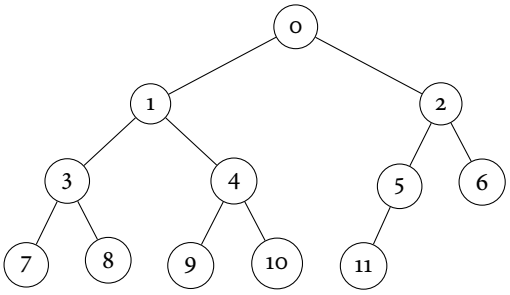


Figure 9.1 Complete binary tree node numbering

An array can store the data values of the tree efficiently, placing each value in the array position corresponding to that node's position within the tree. The following table lists the array indices for the children, parent, and siblings of each node in the figure.

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

Looking at the table, you should see a pattern regarding the positions of a node's relatives within the array. Simple formulas can be derived for calculating the array index for each relative of a node R from R's index. No explicit pointers are

necessary to reach a node's left or right child. This means there is no overhead to the array implementation if the array is selected to be of size n for a tree of n nodes.

The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to $n - 1$.

- Parent(r) = $\lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- Left child(r) = $2r + 1$ if $2r + 1 < n$.
- Right child(r) = $2r + 2$ if $2r + 2 < n$.
- Left sibling(r) = $r - 1$ if r is even and $r \neq 0$.
- Right sibling(r) = $r + 1$ if r is odd and $r + 1 < n$.

9.2.2 IMPLEMENTING BINARY HEAPS

Be sure not to confuse the logical representation of a heap with its physical implementation by means of the array-based complete binary tree. The two are not synonymous because the logical view of the heap is actually a tree structure, while the typical physical implementation uses an array.

Here is an implementation for *min*-heaps. It uses a dynamic array (see Section 6.7) that will resize automatically when the number of elements change.

```
datatype MinHeap implements PriorityQueue:
    H = new Array(10)   // 10 is the initial capacity.
    size = 0            // The initial heap contains 0 elements.
```

Note that, because we use an array to store the heap, we indicate the nodes by their logical position within the heap rather than by a pointer to the node. In practice, the logical heap position corresponds to the identically numbered physical position in the array.

Since it is a heap, we know that the first element always contains the element with the highest priority:

```
datatype MinHeap implements PriorityQueue:
    ...
    getMin():
        // Precondition: the heap must contain some elements.
        if size > 0:
            return H[0]
```

The datatype contains some private auxiliary methods that are used when adding and removing elements from the heap: `isLeaf` tells if a position represents a leaf

in the tree, while `getLeftChild`, `getRightChild` and `getParent` return the position for the left child, right child, and parent of the position passed, respectively.

```
datatype MinHeap:
    ...
    isLeaf(pos):
        return pos >= size / 2

    getLeftChild(pos):
        return 2 * pos + 1

    getRightChild(pos):
        return 2 * pos + 2

    getParent(pos):
        return int((pos - 1) / 2)
```

We also need an auxiliary method for swapping two elements in the heap.

```
datatype MinHeap:
    ...
    swap(i, j):
        H[i], H[j] = H[j], H[i]
```

Finally, since we use a dynamic array we have to be able to resize the internal array. This is explained in further detail in Section 6.7.

```
datatype MinHeap:
    ...
    resizeHeap(newCapacity):
        oldH = H
        H = new Array(newCapacity)
        for i in 0 .. size-1:
            H[i] = oldH[i]
```

9.2.3 INSERTING INTO A HEAP

Here is how to insert a new element V into the heap:

- First put the new value at the end of the array.
- Now move the value upward in the heap, comparing with its parent.
- If V has a higher priority than its parent, swap the two corresponding array cells.
- Continue until V does not have higher priority than its parent.

Here is an alternative explanation: If the heap takes up the first n positions of its array prior to the call to `add`, it must take up the first $n + 1$ positions after. To

accomplish this, `add` first places V at position n of the array. Of course, V is unlikely to be in the correct position. To move V to the right place, it is compared to its parent's value. If the value of V is less than or equal to the value of its parent, then it is in the correct place and the insert routine is finished. If the value of V is greater than that of its parent, then the two elements swap positions. From here, the process of comparing V to its (current) parent continues until V reaches its correct position.

Here is the pseudocode for insertion in our *min*-heap. Note that we use a helper method for "sifting" a value up the tree.

```
datatype MinHeap:
    ...
    add(elem):
        if size >= H.size:
            resizeHeap(H.size * 2)
        H[size] = elem       // Add the element at end of the heap.
        siftUp(size)         // Put it in its correct place.
        size = size + 1      // Increase the size of the heap.

    siftUp(pos):
        while pos > 0:       // Stop when we reach the root (if not earlier).
            parent = getParent(pos)
            if H[pos] >= H[parent]:
                return pos   // Stop if the parent is smaller or equal.
            swap(pos, parent)
            pos = parent     // Move up one level in the tree.
```

Important note: One common mistake is to start at the root and work yourself downwards through the heap. However, this approach does not work because the heap must maintain the shape of a complete binary tree.

Since the heap is a complete binary tree, its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $O(\log n)$. Intuitively, we can see that this must be true because each level that we add will slightly more than double the number of nodes in the tree (the i th level has 2^i nodes, and the sum of the first i levels is $2^{i+1} - 1$). Starting at 1, we can double only $\log n$ times to reach a value of n. To be precise, the height of a heap with n nodes is $\lceil \log n + 1 \rceil$.

Each call to `add` takes $O(\log n)$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree. Thus, to insert n values into the heap, if we insert them one at a time, will take $O(n \log n)$ time in the worst case.

9.2.4 REMOVING FROM THE HEAP

Here is how to remove the highest-priority element V from a binary heap:

- We know that the highest-priority element is at the tree root, i.e. array position 0.
- We also know that we need to reduce the array/heap size by 1 – so we can swap the first an last positions.
- Now the new root element does not satisfy the heap property.
- We move the new root downward in the heap – in each step comparing with the *highest-priority* child.
- Continue until V has a higher priority than both its children.

Because the heap is $\log n$ levels deep, the cost of deleting the maximum element is $O(\log n)$ in the average and worst cases.

Here is the pseudocode for removing the minimum value from our *min*-heap. Note that we use a helper method for "sifting" a value down the tree.

```
datatype MinHeap:
    ...
    removeMin():
        removed = H[0]       // Remember the current minimum value, to return in the end.
        swap(0, size-1)  // Swap the last array element into the first position...
        size = size - 1      // ...and remove the last element, by decreasing the size.
        if size > 0:
            siftDown(0)      // Put the new root in its correct place.
        return removed

    siftDown(pos):
        while not isLeaf(pos):           // Stop when we reach a leaf (if not earlier).
            child = getLeftChild(pos)
            right = getRightChild(pos)
            if right < size and H[right] < H[child]:
                child = right            // 'child' is now the index of the child with smaller value.
            if H[child] >= H[pos]:
                return pos               // Stop if the parent is smaller or equal.
            swap(pos, child)
            pos = child                  // Move down one level in the tree.
```

9.2.5 BINARY HEAPS AS PRIORITY QUEUES

The heap is a natural implementation for the priority queue discussed at the beginning of this chapter. Jobs can be added to the heap (using their priority value as the ordering key) when needed. Method `removeMin` can be called whenever a new job is to be executed.

Priority queues can be helpful for solving graph problems such as finding the shortest path and finding the minimum spanning tree. For a story about Priority Queues and dragons, see Computational Fairy Tales: Stacks, Queues, Priority Queues, and the Prince's Complaint Line

9.2.6 CHANGING THE PRIORITY OF ELEMENTS

For some applications, objects might get their priority modified. One solution to this is to remove the object and reinsert it. Another solution is to change the priority value of the object, and then update its position in the heap. In any of these cases the application needs to know the position of the object in the heap.

To be able to know the position of an arbitrary object in the heap, we need some auxiliary data structure with which we can find the position of an object. This auxiliary structure can be any kind of map, which we will discuss later in chapters X–Y.

Assuming that we know the position of the object, we either have to remove it from the heap, or modify its priority.

- To remove the object at an arbitrary position i, we first swap it with the last position. Then we sift the new value at position i, either up or down depending on the priorities of the parent and children.
- To modify the priority of the object at position i, we first modify it and then sift the value up or down depending on the priorities of the parent and children.

Section 9.2.7

9.2.7 BUILDING A HEAP

If all n values are available at the beginning of the building process, we can build the heap faster than just inserting the values into the heap one by one.

Read the rest online

9.3 Case study: Heapsort

We can use a heap to implement a very simple sorting algorithm:

1. Insert all elements from the unsorted array into a *min*-heap.
2. Remove each element in turn from the heap, putting it in its right place in the original array.

Section 9.3

Since the heap returns the smallest elements first, they will be inserted in sorted order into the new array. Here is pseudocode:

Read the rest online

```
function naiveHeapsort(A):
    heap = new MinHeap()
    for i in 0 .. A.size-1:
        heap.add(A[i])
    for i in 0 .. A.size-1:
        A[i] = heap.removeMin()
```

9.4 Case study: Huffman coding

One can often gain an improvement in space requirements in exchange for a penalty in running time. There are many situations where this is a desirable tradeoff. A typical example is storing files on disk. If the files are not actively used, the owner might wish to compress them to save space. Later, they can be uncompressed for use, which costs some time, but only once.

Section 9.4

Read the rest online

We often represent a set of items in a computer program by assigning a unique code to each item. For example, the standard Latin-1 scheme (the formal name is ISO 8859-1) assigns a unique eight-bit value to each character. It takes a certain minimum number of bits to provide enough unique codes so that we have a different one for each character. For example, it takes $\lceil \log\ 256 \rceil$ or eight bits to provide the 256 unique codes needed to represent the 256 symbols of the Latin-1 character set.

Latin-1 and most other encodings are *fixed-length* codes, where each value is stored in the same number of bits. If all characters were used equally often, then a fixed-length encoding is the most space efficient method. However, you are probably aware that not all characters are used equally often in many applications. For example, the various letters in an English language document have greatly different frequencies of use.

If some letters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other letters require longer codes, but this might be worthwhile if such letters appear rarely enough. This concept is at the heart of file compression techniques in common use today. In this section we present one such approach to assigning variable-length codes, called Huffman coding.

9.5 Review questions

This final section contains some review questions about the contents of this chapter.

Section 9.5

Answer quiz online

CHAPTER 10

Sets and maps

Many programming tasks involve *finding the right piece of information* in a large dataset. That is, we have a collection of items, and we want to quickly retrieve the items matching certain criteria. Here are some examples of information retrieval problems:

- *Spell-checking:* Given a set containing all valid English words, check if a given string is present in the set (i.e. is a valid word).
- *Cash register:* Given a database of all items for sale in a supermarket, find information about the item with a given *EAN code*.- *Search engine:* Given a collection of documents (e.g. web pages), find all documents containing a given word.
- *Between X and Y:* Given a list of all Swedish towns and their populations, are there any towns whose population is between 5,000 and 10,000? And if so, which are these towns?

These problems can all be addressed using two abstract data types (ADTs): the *set* and the *map*. Both provide efficient ways to manage a collection of elements, supporting operations to find, add, and remove elements.

In this section, we'll explore what sets and maps are, how they work, and how they can be applied to solve the four example problems introduced above.

You may have already used sets and maps in programming, because almost every programming language provides an implementation for them. For example, Java provides the HashSet and HashMap classes, and Python provides sets and dictionaries (another word for maps) as part of their standard libraries.

Sets and maps are useful in a huge variety of computer programs, and are perhaps the most useful of all data structures. But how can we design a class that implements a set or a map, in such a way that adding, removing and searching can be done efficiently? In this book we will see several different ways of implementing sets and maps.

- In Section 10.3, we will see how to implement a set or a map using a list. By sorting the items in the list, it is possible to look up information efficiently.

However, it turns out that adding and removing items is quite expensive. A list is a suitable way of storing a set or a map if its contents never changes.
- In chapter 11, we learn about *balanced binary search trees (BSTs)*, a data structure for sets and maps where adding, removing and searching are all efficient. BSTs also support the *sorted map* operations that we used i n our final example.
- In chapter 12, we learn about *hash tables*, another way to implement the set and map ADTs. In a hash table, `add`, `remove` and `contains` are even faster than in a BST, but hash tables are somewhat harder to use than BSTs, and do not support the *sorted map* operations.

Balanced BSTs and hash tables are the main ways that sets and maps are implemented in practice. Almost every programming language provides sets and maps as a built-in feature, based on one of these technologies. For example, Java's HashSet, HashMap, TreeSet and TreeMap, and Python's: sets and dictionaries. By the end of this book you will understand how all of these work.

10.1 Sets

A *set* represents a collection of items, where we can *add* and *remove* items, and *check* if a given item is present in the set. A set cannot contain duplicate items: if we try to add an item that is already present, nothing happens, and the set is left unchanged. We can specify a minimal interface for sets like this:

```
interface Set of T extends Collection:
    add(x: T)               // Adds x to the set.
    remove(x: T)            // Removes x from the set.
    contains(x: T) -> Bool  // Returns true if x is in the set.
```

Example: Spell-checking

We can use a set for the spell-checking example:

- Given a set containing all valid English words, check if a given string is present in the set (i.e. is a valid word).

To create the spell-checking dictionary, we start with an initially empty set, and then call `add` repeatedly to add each valid word to the set. Then to spell-check a given word, we just call `contains`.

```
datatype SpellChecker:
```

```
    validWords: Set of String

    constructor(listOfValidWords: Collection of String):
        // Convert the list of words into a set.
        validWords = new Set()
        for each word in listOfValidWords:
            validWords.add(word)

    isValidWord(word) -> Bool:
        return validWords.contains(word)
```

Here's how the `SpellChecker` can be used:

```
function main(wordsToCheck: Collection of String):
    // Create a new spell checker.
    checker = new SpellChecker(["cat", "dog"])

    // Now we can spell-check a word easily.
    for each word in wordsToCheck:
        if checker.isValidWord(word):
            print word "is valid"
        else:
            print word "is INVALID"
```

10.2 Maps, or dictionaries

A *map* (or dictionary) represents a set of *keys*, where each key has an associated *value*. We can *add* and *remove* keys, but when we add a key we must specify what *value* we want to associated with it. We can *check* if a given key is present in the map. We can also *look up* a key to find the associated value.

A map cannot contain duplicate *keys*, so each key is associated with exactly one value. If we call `put(k,v)`, but the key `k` is already present, then the value associated with `k` gets changed to `v`. On the other hand, a map *can* contain duplicate *values*: two keys can have the same value. Here is a possible minimal interface for maps:

```
interface Map of K to V extends Collection of K:
    put(key: K, value: V)         // Sets the value of the given key.
    get(key: K) -> V              // Returns the value associated with the given key, or `null` if the key is not there.
    remove(key: K)                // Removes the value associated with the given key.
    containsKey(key: K) -> Bool   // Returns true if the key has an associated value.
```

Note that maps depend on two different types, the keys `K` and the values `V`. These types can be the same or different, depending on the needs of your application.

Example: Cash register

The map is a perfect match for our supermarket example:

- *Cash register:* Given a database of all items for sale in a supermarket, find information about the item with a given *EAN code*.

Here, the key should be the EAN code that the barcode scanner recognises, and the value should be a record containing information about the item. If the EAN code is stored in a field `ean`, then to put an item `p` in the database we call `database.put(p.ean, p)`. To find the item with barcode `code` we call `database.get(code)`.

```
datatype Item
    ean: String
    name: String
    price: Number
    expires: Date

datatype CashRegister:
    database: Map of String to Item = new Map()

    // Put the item in the database.
    put(p: Item):
        database.put(p.ean, p)

    // Remove an item from the database.
    remove(p: Item):
        database.remove(p.ean)

    // Find the item with the given EAN code.
    find(ean: String) -> Item:
        return database.get(ean)
```

10.2.1 MULTIMAPS

Maps have the restriction that each key has only one value. However, sometimes we want to store a list of records, where some records might have the same key. Then we want something like a map, but where a key can have multiple values associated with it. This structure is called a *multimap*.

Unfortunately, most programming languages do not provide a multimap data structure. Instead, we can implement it ourselves. The idea is to use a map, whose value type is a *set* of the actual values that we are interested in.

Example: Search engine

A multimap is the perfect data structure for our search engine example:

- Given a collection of documents (e.g. web pages), find all web pages containing a given word.

To find all documents containing a given word, we will build a multimap, where the key is a word, and the values are all documents containing that word. Then, searching for a word will just mean looking it up in the multimap.

```
// We model a document as a list of words.
datatype Document:
    contents: Collection of String

datatype SearchEngine:
    database: Map of String to Set of Document = new Map()

    // Add a new document to the database.
    add(doc: Document):
        for each word in doc.contents:
            if not database.containsKey(word):
                // This is the first document containing this word.
                database.put(word, new Set())

            // Get the set of documents containing this word, and add the document.
            set = database.get(word)
            set.add(doc)

    // Find all documents containing a given word.
    find(word: String) -> Set of Document:
        if database.containsKey(word):
            return database.get(word)
        else:
            // If the word is not found, return an empty set.
            return new Set()
```

Note that we don't have to `put` the updated set back into the database (in the `add` method). This is because complex data structures are *mutable*, as explained in Section 1.5.

10.3 Case study: Implementing sets and maps using sorted lists

10.3.1 IMPLEMENTING SETS

We can implement either of these ADTs using an array. For a set, we can use an array of elements. We have one further choice: should the array be *sorted* or *unsorted*?

An unsorted array is usually not an appropriate choice, because the `contains` method must use *linear search*, which takes linear time. Thus we cannot really look up items in the set or map efficiently.

A sorted array is a lot better. The `contains` method can use *binary search*, which takes logarithmic time. Hence looking up items is efficient. Unfortunately, modifying the data structure is slow. If we want to `add` an item to a sorted array, we have to keep the array sorted – and that means we need to *insert* the new item at the right place in the array, using the insertion algorithm from Insertion Sort. This takes linear time in the worst case. Similarly, to `remove` an item without creating a hole in the array, we need to move all the items that come after one space backwards. This also takes linear time.

A sorted array is a suitable way to implement a set or a map that *never changes*, that is where we never need to add or remove items after the array is created. We start by sorting the array, using either quicksort or mergesort, and then we can use binary search to find items in it. Sorted arrays also support the *sorted set* and *sorted map* operations such as *range queries* – these can also be implemented using binary search.

Sorted arrays can also be useful in cases where we always add *many* items in one go. Given a sorted array A, and an unsorted list of items B, we can add the items in B to A as follows. First we sort B, then we merge A and B, using the merge algorithm from mergesort. Note that the merge step takes linear time, and sorting B takes a bit more than linear time, so this is a lot faster than adding all the items from B one by one (which would take quadratic time).

An array is not a good way to implement a set or a map, if we need both `add`, `remove` and `contains` to be efficient. Later we will learn about two data structures that are more suitable for implementing sets and maps: binary search trees and hash tables.

10.3.2 COMPLEXITY ANALYSIS

10.3.3 IMPLEMENTING MAPS

We can implement either of these ADTs using an array. For a set, we can use an array of elements. For a map, we have two choices:

- In languages which support *tuples* as a data type (such as Python), we can have an array of *key-value* pairs.
- Alternatively, we can use two arrays. One array, `keys`, holds the keys and the other array, `values`, holds the corresponding values. The two arrays must be "kept in sync" so that `values[i]` holds the value associated with key `keys[i]`.

It is not difficult to implement a **Map** using a list. The problem is that all the operations – searching for a key, updating the value for a key, and removing a key – will be linear in the number of entries, $O(n)$.

In later chapters we will see how to improve the efficiency, by using

- Balanced search trees (Section 11.2), which bring down the complexity of the operations to $O(\log n)$.
- Hash tables (chapter 12), which make the operations constant time, $O(1)$.

But some times it is enough to use a simple list-based implementation. And in fact, the separate chaining hash map (Section 12.3) requires an underlying simpler map implementation – and there a linked list works very fine!

Using a list of key-value pairs A very simple way of implementing a **Map** using a list, is to use key-value pairs.

```
datatype KVPair of K and V:
    key: K
    value: V
```

Now we can create a **Map** class that uses an underlying **List** of **KVPair**. So the only thing we need is really an internal variable referring to the underlying list.

```
datatype LinkedMap implements Map:
    internalList: LinkedList of KVPair = new LinkedList()
```

Finding the value for a certain key is easy. We just iterate through all entries and stop whenever we find a matching key.

```
datatype LinkedMap implements Map:
    ...
    get(key):
        for each entry in internalList:
            if key == entry.key:
```

```
            return entry.value
        return null
```

Setting a value for a given key means to search the list for a matching key, and then updating the value. If we cannot find the key, we add a new **KVPair** to the list.

```
datatype LinkedMap implements Map:
    ...
    put(key, value):
        for each entry in internalList:
            if key == entry.key:
                entry.value = value
                return
        // The key isn't present, so we add a new key-value pair to the list.
        // Because it's a linked list we add the pair to the front of the list.
        internalList.add(0, new KVPair(key, value))
```

In this example we're using a linked list, but we could equally well have used a dynamic array list. The only thing we have to think about is to add new pairs at the right location (beginning or end of the list) – because linked lists prefer adding at the front, while array lists rather add to the back of the list.

Other methods can be deferred to the underlying list.

```
class LinkedMap implements Map:
    ...
    size():
        return internalList.size
```

(Note that since the number of entries can vary, we need `size` to be a method and not a property.)

How to remove keys from the map There is one problem with this simple map implementation – how to remove keys from it. One possibility would be to first search for the index where the key is located, and then remove that index from the list.

But this would be slightly inefficient, because removing an element from a certain position takes $O(n)$ time in the worst case. So, first we find the position (which takes $O(n)$ time), and then we remove it (which takes another $O(n)$ time). This is double the work than it should be, which is unnecessary.

```
class LinkedMap implements Map:
    ...
    // This method is sub-optimal, because it makes two passes:
    // First a search to find the index, and then a loop delete that index.
    remove(key):
```

```
        i = 0
        for each entry in internalList:
            if key == entry.key:
                internalList.remove(i)
                return entry.value
            i = i + 1
        return null
```

If we allow ourselves to peek into the inner workings of the linked list, we can make a more efficient version. Note that to be able to remove a list node, we need a pointer to the *previous* node. This makes it possible for us to repoint the `next` pointer from the previous node to its new next node. Therefore we keep two variables, `prev` that points to the previous node, and `current` that points to the current.

```
class LinkedMap implements Map:
    ...
    remove(key):
        prev = null
        current = internalList.head
        while current is not null:
            if key == current.key:
                if prev is null:
                    // Special case: if prev is null,
                    // it's the list head that needs to be repointed.
                    internalList.head = current.next
                else:
                    prev.next = current.next
                current.next = null   // For garbage collection
                internalList.size = internalList.size - 1
                return current.value
            prev = current
            current = current.next
        return null
```

It is not good programming practice that one datatype (or class, or module) looks into the implementation details of another one. Therefore, a real library for linked lists should have more public methods to be able to implement an efficient version without having to look into its implementation.

For example, the **Iterator** interface in Java provides a "remove-the-current-node" method, so it is possible to implement optimise map removal just like above.

Using linked key-value nodes An alternative to using an underlying list of key-value pairs is to modify the implementation of linked lists just slightly. This gives us more control of the implementation, with the tradeoff that we have to reimplement some things.

Instead of using linked list nodes with just one value, we used key-value nodes.

```
datatype KVNode of K and V:
    key: K          // Key for this node
    value: V        // Value for this node
    next: KVNode    // Pointer to next node in list
```

The actual implementation of the datatype LinkedMap now becomes an exercise for the reader.

10.4 Sorted sets and maps

Consider the final example problem from the introduction to this chapter:

- *Between X and Y:* Given a list of all Swedish towns and their populations, are there any towns whose population is between 5,000 and 10,000? And if so, which are these towns?

These are two example of *range queries*: given a set, finding if there are any elements within a given range; or given a map, finding all items whose key lies in a given range. Some set and map implementations support answering range queries efficiently; we say that these data structures implement *sorted sets* and *maps*.

10.4.1 SORTED SETS

Example: Between X and Y

The first example range query is:

- Given a list of all Swedish towns and their populations, is there any town whose population is between 5,000 and 10,000?

One way to solve this problem would be to use a normal set of city populations. Then we could find the answer to our query by making a sequence of calls to `contains`:

- `contains(5000)` – is there a town with population 5,000?
- `contains(5001)` – is there a town with population 5,001?
- `contains(5002)` – is there a town with population 5,002?
- etc.

But this is not a sensible approach. We would need to make up to 5,000 calls to `contains`, and if we wanted to instead find if there are any cities in Europe having a population of between 1 and 2 million, we would need to up to 1,000,000 calls.

There is a better way. If the populations are stored in a sorted array, we can use the following algorithm:

- Find the position in the array of the *first* town with a population of *at least* 5,000. (This can be done efficiently using binary search.)
- Check if this population is at most 10,000.

So, a sorted array can be used as an efficient implementation of a sorted set. However, as we saw in the previous section, sorted arrays are not the best choice if you want to add or remove elements.

Some set implementations support answering range queries efficiently, such as sorted lists as we saw in the example above. But there are other implementations too – we say that these data structures implement *sorted sets*.

Apart from range queries, sorted sets support several other operations that take advantage of the natural order of the elements:

- Finding the *smallest* or *largest* element in the set.
- Finding the *closest* element to a given one. Given an element e (which may or may not be in the set), then:
  - The *successor* of e is the next element after e in the set, i.e. the smallest element e' such that $e < e'$.
  - The *predecessor* of e is the previous element before e in the set, i.e. the greatest element e' such that $e' < e$.

  A variant which is sometimes useful is *floor* and *ceiling*:

  - The *floor* of e is the greatest element e' such that $e' \leq e$. If e is in the set, then the floor of e is just e; otherwise it is the predecessor of e.
  - The *ceiling* of e is the least element k' such that $e \leq e'$. If e is in the map, then the ceiling of e is just e; otherwise it is the successor of e.

These operations are summarised in this interface for sorted sets. Note that it *extends* the `Set` interface, it has all the methods that normal sets also have.

```
interface SortedSet of T extends Set:
    first() -> T           // Returns the first (smallest) element.
    last() -> T            // Returns the last (largest) element.
    floor(x: T) -> T       // Returns the closest element ≤ x, or nothing if there is no such element.
    ceiling(x: T) -> T     // Returns the closest element ≤ x, or nothing if there is no such element.
    lower(x: T) -> T       // Returns the closest element < x, or nothing if there is no such element.
    higher(x: T) -> T      // Returns the closest element > x, or nothing if there is no such element.
    between(x1: T, x2: T) -> Collection of T
                           // Returns all elements x such that x1 ≤ x ≤ x2.
```

10.4.2 SORTED MAPS

Example: Between X and Y (again)

Now consider the second range query in the example above:

- Given a list of all Swedish towns and their populations, find the towns whose population is between 5,000 and 10,000.

One way to solve this problem would be to use a *multimap* (see Section 10.2.1). The key would be a population number, and the values would be all towns having that population. Then we could find the required towns by making a sequence of `get(5,000)`, `get(5,001)`, …., until `get(10,000)`.

But just as for the set range query, this is not feasible. Instead, we can store the towns an array which is sorted by population, and then use the following algorithm:

- Find the position in the array of the *first* town that has a population of *at least* 5,000.
- Find the position in the array of the *last* town that has a population of *at most* 10,000.
- Now return all towns between those two positions in the array.

The operation that is needed is: given a map, find all items whose key lies in a given range. Apart from range queries, sorted maps support similar operations as we introduced for sorted sets.

Here is a possible interface for sorted maps, which extends the normal map interface. Note the similarity to the interface for sorted sets.

```
interface SortedMap of K to V extends Map:
    firstKey() -> K              // Returns the first (smallest) key.
    lastKey() -> K               // Returns the last (largest) key.
    floorKey(key: K) -> K        // Returns the closest key ≤ k, or nothing if there is no key.
    ceilingKey(key: K) -> K      // Returns the closest key ≤ k, or nothing if there is no key.
    lowerKey(key: K) -> K        // Returns the closest key < k, or nothing if there is no such element.
    higherKey(key: K) -> K       // Returns the closest key > k, or nothing if there is no such element.
    keysBetween(key1: K, key2: K) -> Collection of K
                                 // Returns all keys k such that k1 ≤ k ≤ k2.
```

Example: Small Swedish towns

Here is how to use a sorted map ADT to find all Swedish towns having between 5,000 and 10,000 population. As there may be towns that have the same population, we need a *multimap*, where the key is the population and the value is a *set* of towns.

```
datatype City:
    name: String
    population: Int

// Use a sorted map where the value is a list of cities.
datatype CityPopulations:
    cities: SortedMap of Int to Set of City = new SortedMap()

    // Add a new city to the database.
    add(city: City):
        if not cities.containsKey(city.population):
            // This is the first city with this population.
                cities.put(city.population, new Set())

        // Get the set of documents containing this city.
        set = cities.get(city.population)
        set.add(doc)

    // Find all cities with a population between lower and upper
    findBetween(lower: Int, upper: Int) -> Set of City:
        result = new Set()
        // The range query returns a collection of keys, i.e. populations.
        for each population in cities.keysBetween(lower, upper):
            // cities.get(population) returns the list of cities with that population.
            for each city in cities.get(population):
                result.add(city)
        return result
```

CHAPTER 11

Search trees

This chapter covers *binary search trees (BSTs)*, a data structure that uses a binary tree to implement a set or a map.

However, as we will see, a problem with BSTs is that they might become *unbalanced*, which make them inefficient. If the tree is unbalanced, all basic operations (searching, adding and removing) become linear in the size of the tree, $O(n)$.

The solution to that problem is make the trees automatically rebalance. This can be done in many different ways, some examples are AVL trees (Section 11.3), Splay trees (Section 11.4) and Red-black trees, who all have their own way of making the basic operaitions logarithmic, $O(\log n)$, in the size of the tree. Another way of keeping the trees balanced is to use *non-binary* trees, such as 2-3 trees or more general B-trees, but we currently don't cover them in this book.

Not all data structures used for searching are trees – in the end of the chapter we discuss Skip lists, which is an interesting hybrid between a tree and a linked list (Section 6.3).

11.1 Binary search trees

A binary search tree (BST) is a binary tree that conforms to the following condition, known as the binary search tree property:

> All nodes stored in the left subtree of a node whose key value is K have key values less than or equal to K. All nodes stored in the right subtree of a node whose key value is K have key values greater than K.

Figure 11.1 below shows two BSTs for a collection of values. One consequence of the binary search tree property is that if the BST nodes are printed using an inorder traversal, then the resulting enumeration will be in sorted order from lowest to highest.

The only thing that differentiates a BST from a normal binary tree is the BST

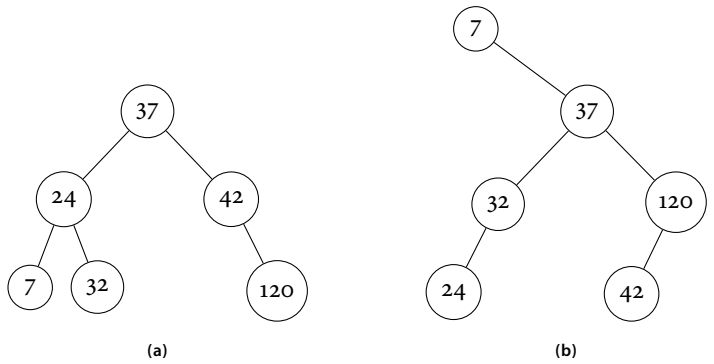


Figure 11.1 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 120, 32, 7. Tree (b) results if the same values are inserted in the order 7, 37, 42, 32, 120, 24.

property. This property is an *invariant*, as as explained in Section 2.2, and it is a condition that the BST *always* must satisfy.

Invariants are not stored in the datatype, so the actual declaration for a BST is exactly the same as a normal binary tree:

```
datatype BST:
    root: BinaryNode = null
    size: Int = 0
```

The BST property is therefore not explicit in the datatype declaration, but instead it is something that all operations must satisfy. Whenever we implement a new operation, we have to make sure that we never break the property.

Searching in a BST Because of the BST property, we don't have to search the whole tree if we want to find an element. Instead, we can start at the root node and compare its value with the value we are searching for. If the value is smaller than the root, we know that we can disregard everything in the right subtree. (And conversely, if the value is larger, we can disregard the left subtree.)

Now, assuming that the value was smaller than the root, we can go to the left child and compare again. If the value now is larger than the child, we can continue into the child's right subtree. We continue this until we have found the value in a node, or until we reach an empty child. If we reach an empty child we know that the value is not in the tree. Here is a possible implementation of a method that returns true if the element is in the tree:

```
datatype BST:
    ...
    contains(elem):
        node = root
        while node is not null:
            if elem == node.elem:
                return true
            else if elem < node.elem:
                node = node.left
            else if elem > node.elem:
                node = node.right
        return false
```

Inserting into a BST When we want to insert an element we first have to search for it, similar to what we did above. If it already exists we don't have to do anything, and if it isn't in the tree we create a new node and attach it to the right place. But how do we know where to add the new node? The variable `node` becomes null if the element isn't found, so we have nothing to attach the node to – instead we want to attach the new node to the *previous* tree node that we looked at. The solution is to add another temporary variable, `parent`, which points to the parent of `node` – i.e., its value in the previous iteration.

Now, after we have completed the search, and the element wasn't found, we can simply create a new node and attach it as a child to `parent`. Depending on if the element is smaller or larger than the parent element, we make it a left or right child. Alternatively, if the tree is empty then we have to attach the new node directly to the root.

```
datatype BST:
    ...
    add(elem):
        // Search for the parent of the new node.
        parent = null
        node = root
        while node is not null:
            parent = node
            if elem == node.elem
                return  // The element is already in the BST, so we do nothing.
            else if elem < node.elem:
                node = node.left
            else if elem > node.elem:
                node = node.right

        // Create a new node and attach it to the parent.
        node = new BinaryNode(elem)
```

```
if parent is null:
    root = node   // The tree is empty, so we update the root.
else if elem < parent.elem:
    parent.left = node
else if elem > parent.elem:
    parent.right = node
size = size + 1
```

We have to decide what to do when the node that we want to insert has a key value equal to the key of some node already in the tree. If during insert we find a node that duplicates the key value to be inserted, then we have two options. If the application does not allow nodes with equal keys, then this insertion should be treated as an error (or ignored). If duplicate keys are allowed, our convention will be to insert the duplicate in the left subtree.

The shape of a BST depends on the order in which elements are inserted. A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree. Figure 11.1 illustrates two BSTs for a collection of values. It is possible for the BST containing n nodes to be a chain of nodes with height n. This would happen if, for example, all elements were inserted in sorted order. In general, it is preferable for a BST to be as shallow as possible. This keeps the average cost of a BST operation low.

11.1.1 RECURSIVE SEARCH AND INSERT

Section 11.1.1

Read the rest online

Bot searching and insertion can be implemented as recursive function too, and it is instructive to see how to to that. And since we showed the *set* operations `contains` and `add` above, we will show how to implement recursive versions of the *map* operations `get` and `put`.

11.1.2 REMOVING FROM A BST

Removing a node from a BST is a bit trickier than inserting a node, but it is not complicated if all of the possible cases are considered individually. Before tackling the general node removal process, we need a useful companion method, `largestNode`, which returns a pointer to the node containing the maximum value in a subtree.

```
function largestNode(node):
    while node.right is not null:
        node = node.right
    return node
```

This time we will show a *recursive* implementation of `remove`. For this we need a recursive helper function which we initially call with the root of the tree.

```
datatype BSTMap:
    ...
    remove(key):
        root = removeHelper(root, key)
```

Now we are ready for the `removeHelper` method. Removing a node with given key value R from the BST requires that we first find R and then remove it from the tree. So, the first part of the remove operation is a search to find R. Once R is found, there are several possibilities. If R has no children, then R's parent has its pointer set to **null**. If R has one child, then R's parent has its pointer set to R's child. The problem comes if R has two children. One simple approach, though expensive, is to set R's parent to point to one of R's subtrees, and then reinsert the remaining subtree's nodes one at a time. A better alternative is to find a value in one of the subtrees that can replace the value in R.

Thus, the question becomes: Which value can substitute for the one being removed? It cannot be any arbitrary value, because we must preserve the BST property without making major changes to the structure of the tree. Which value is most like the one being removed? The answer is the least key value greater than the one being removed, or else the greatest key value less than (or equal to) the one being removed. If either of these values replace the one being removed, then the BST property is maintained.

When duplicate node values do not appear in the tree, it makes no difference whether the replacement is the greatest value from the left subtree or the least value from the right subtree. If duplicates are stored in the left subtree, then we must select the replacement from the *left* subtree. To see why, call the least value in the right subtree L. If multiple nodes in the right subtree have value L, selecting L as the replacement value for the root of the subtree will result in a tree with equal values to the right of the node now containing L. Selecting the greatest value from the left subtree does not have a similar problem, because it does not violate the Binary Search Tree Property if equal values appear in the left subtree.

Note: Alternatively, we can decide to store duplicate values in the right subtree instead of the left. Then we must replace a deleted node with the least value from its right subtree.

The code for removal is shown here. Note that the helper function returns the updated subtree, and we have to make sure to update the child pointers to this updated tree.

```
    removeHelper(node, key):
```

```
if node is null:
    return null
else if key < node.key:
    node.left = removeHelper(node.left, key)
    return node
else if key > node.key:
    node.right = removeHelper(node.right, key)
    return node
else if key == node.key:
    if node.left is null:
        size = size - 1
        return node.right
    else if node.right is null:
        size = size - 1
        return node.left
    else:
        predecessor = largestNode(node.left)
        node.key = predecessor.key
        node.value = predecessor.value
        node.left = removeHelper(node.left, predecessor.key)
        return node
```

Section 11.1.2

Read the rest online

11.1.3 ANALYSIS

The cost for `contains`, `get`, `add`, and `put` is the depth of the node found or inserted. The cost for `remove` is the depth of the node being removed, or in the case when this node has two children, the depth of the node with smallest value in its right subtree. Thus, in the worst case, the cost for any one of these operations is the depth of the deepest node in the tree. This is why it is desirable to keep BSTs balanced, that is, with least possible height. If a binary tree is balanced, then the height for a tree of n nodes is approximately $\log n$. However, if the tree is completely unbalanced, for example in the shape of a linked list, then the height for a tree with n nodes can be as great as n. Thus, a balanced BST will in the average case have operations costing $O(\log n)$, while a badly unbalanced BST can have operations in the worst case costing $O(n)$. Consider the situation where we construct a BST of n nodes by inserting records one at a time. If we are fortunate to have them arrive in an order that results in a balanced tree (a "random" order is likely to be good enough for this purpose), then each insertion will cost on average $O(\log n)$, for a total cost of $O(n \log n)$. However, if the records are inserted in order of increasing value, then the resulting tree will be a chain of height n. The cost of insertion in this case will be $\sum_{i=1}^{n} i \in O(n^2)$.

Traversing a BST costs $O(n)$ regardless of the shape of the tree. Each node is

visited exactly once, and each child pointer is followed exactly once. For example, if we want to print the nodes in ascending order we can perform an inorder traversal (Section 8.4), which then will take time linear in the size of the tree.

While the BST is simple to implement and efficient when the tree is balanced, the possibility of its being unbalanced is a serious liability. There are techniques for organising a BST to guarantee good performance. Two examples are the AVL tree (see Section 11.3) and the splay tree (see Section 11.4). There also exist other types of search trees that are guaranteed to remain balanced, such as the red-black tree or the 2-3 Tree.

11.1.4 GUIDED INFORMATION FLOW

When writing a recursive method to solve a problem that requires traversing a binary tree, we want to make sure that we are visiting the required nodes (no more and no less).

So far, we have seen several tree traversals that visited every node of the tree. We also saw the BST search, insert, and remove routines, that each go down a single path of the tree. Guided traversal refers to a problem that does not require visiting every node in the tree, though it typically requires looking at more than one path through the tree. This means that the recursive function is making some decision at each node that sometimes lets it avoid visiting one or both of its children. The decision is typically based on the value of the current node. Many problems that require information flow on binary search trees are "guided" in this way.

Example: Minimum value in a tree

An extreme example is finding the minimum value in a BST. A bad solution to this problem would visit every node of the tree. However, we can take advantage of the BST property to avoid visiting most nods in the tree. You know that the values greater than the root are always in the right subtree, and those values less than the root are in the left subtree. Thus, at each node we need only visit the left subtree until we reach a leaf node.

11.2 Self-balancing trees

The Binary Search Tree has a serious deficiency for practical use as a search structure. That is the fact that it can easily become unbalanced, so that some

nodes are deep in the tree. In fact, it is possible for a BST with n nodes to have a depth of n, making it no faster to search in the worst case than a linked list. If we could keep the tree balanced in some way, then search cost would only be $O(\log n)$, a huge improvement.

One solution to this problem is to adopt another search tree structure instead of using a BST at all. An example of such an alternative tree structure is the 2-3 tree or the B-tree. But another alternative would be to modify the BST access functions in some way to guarantee that the tree performs well. This is an appealing concept, and the concept works well for heaps, whose access functions maintain the heap in the shape of a complete binary tree. Unfortunately, the heap keeps its balanced shape at the cost of weaker restrictions on the relative values of a node and its children, making it a bad search structure. And requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update, as we see in the example in Figure 11.2.

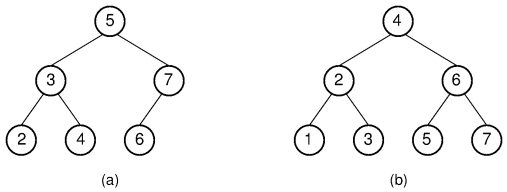


Figure 11.2 An attempt to re-balance a BST after insertion can be expensive. (a) A BST with six nodes in the shape of a complete binary tree. (b) A node with value 1 is inserted into the BST of (a). To maintain both the complete binary tree shape and the BST property, a major reorganisation of the tree is required.

If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure. The AVL tree (see Section 11.3) works in this way, using insertion and deletion routines altered from those of the BST to ensure that, for every node, the depths of the left and right subtrees differ by at most one. The red-black tree is also a binary tree, which uses a different balancing mechanism.

A different approach to improving the performance of the BST is to not require that the tree always be balanced, but rather to expend some effort toward making the BST more balanced every time it is accessed. One example of such a compromise is called the splay tree (see Section 11.4).

11.3 AVL trees

The AVL tree is a BST with the following additional property:

> For every node, the heights of its subtrees differ by at most 1.

As long as the tree maintains this property, if the tree contains n nodes, then it has a depth of at most $O(\log n)$. As a result, search for any node will cost $O(\log n)$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $O(\log n)$, even in the worst case.

The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property. Of course, to be practical, we must be able to implement the revised update routines in $O(\log n)$ time. To maintain the balance property, we are going to use what are called rotations.

11.3.1 ROTATIONS

Rotation is an operation that takes a node in the tree and moves it one level higher. Figure 11.3 illustrates rotation. Here, P and S are nodes, while A, B and C represent subtrees.

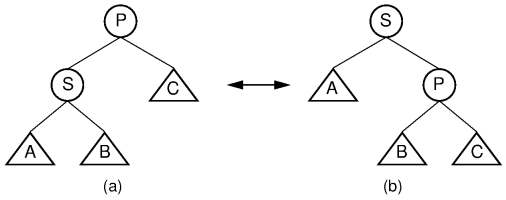


Figure 11.3 Rotation. In a rotation, node S is promoted to the root, rotating with node P. Because the value of S is less than the value of P, P must become S 's right child. The positions of subtrees A, B, and C are altered as appropriate to maintain the BST property, but the contents of these subtrees remains unchanged. (a) The original tree with P as the parent. (b) The tree after a rotation takes place. Going from (a) to (b) is called a *right rotation*. We can also go from (b) to (a) promoting node P to the root – this is called a *left rotation*. In general, a right rotation makes the tree more right-leaning, and a left rotation makes it more left-leaning.

In figure (a), node S is the left child of the root. A *right rotation* transforms it into the tree shown in figure (b), where node S has become the root. Note that, because the value of S is less than the value of P, P must become S's right child. Right rotation means transforming a tree from having the shape in (a) to having the shape in (b).

A *left rotation* is the opposite process: starting from the tree in (b), transforming it to the tree in (a), by lifting node P up. Notice that a right rotation

tends to make the tree more right-leaning, while a left rotation tends to make it more left-leaning.

11.3.2 AVL TREE INSERTION

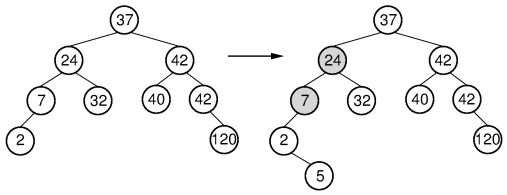


Figure 11.4 An insertion that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

Consider what happens when we insert a node with key value 5, as shown in Figure 11.4. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees. For the bottommost unbalanced node, call it S, there are 4 cases:

(1) The extra node is in the left child of the left child of S.
(2) The extra node is in the right child of the left child of S.
(3) The extra node is in the left child of the right child of S.
(4) The extra node is in the right child of the right child of S.

Cases 1 and 4 are symmetric, as are cases 2 and 3. Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

Our problem now is how to balance the tree in $O(\log n)$ time. It turns out that we can do this using a series of rotations. Cases 1 and 4 can be fixed using a single rotation, as shown in Figure 11.5. Cases 2 and 3 can be fixed using a double rotation, as shown in Figure 11.6.

AVL tree insertion and deletion are easiest to implement as recursive functions. They are very similar to the normal BST insertion and deletion, but as the recursion unwinds up the tree, we perform the appropriate rotation on any node that is found to be unbalanced.

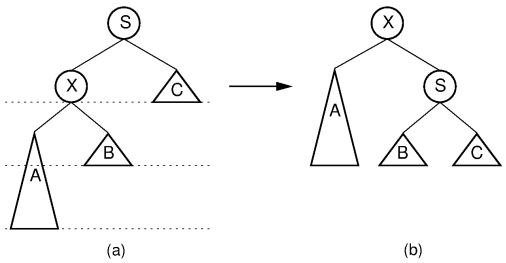


Figure 11.5 A single rotation in an AVL tree. This operation occurs when the excess node (in subtree A) is in the left child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

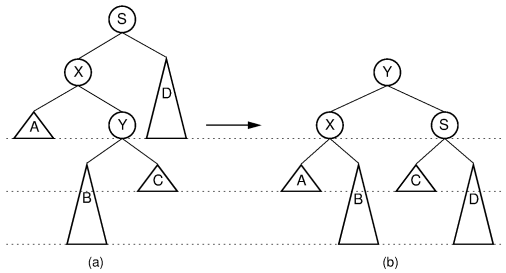


Figure 11.6 A double rotation in an AVL tree. This operation occurs when the excess node (in subtree B) is in the right child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the left child of the right child of S is handled in the same way.

Example: AVL insertion

In Figure 11.4 (b), the bottom-most unbalanced node has value 7. The excess node (with value 5) is in the right subtree of the left child of 7, so we have an example of Case 2. This requires a double rotation to fix. After the rotation, 5 becomes the left child of 24, 2 becomes the left child of 5, and 7 becomes the right child of 5.

Section 11.3.2

Read the rest online

11.4 Splay trees

Like the AVL tree, the splay tree is not actually a distinct data structure, but rather reimplements the BST insert, delete, and search methods to improve the performance of a BST. The goal of these revised methods is to provide guarantees on the time required by a series of operations, thereby avoiding the worst-case linear time behaviour of standard BST operations. No single operation in the splay tree is guaranteed to be efficient. Instead, the access rules of the splay tree guarantee that a series of m operations will take $O(m \log n)$ time for a tree of n nodes whenever $m \geq n$. Thus, a single insert or search operation could take $O(n)$ time. However, m such operations are guaranteed to require a total of $O(m \log n)$ time, for an average cost of $O(\log n)$ per access operation. This is a desirable performance guarantee for any search-tree structure.

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap. Ultimately, it is the cost of the series of operations that matters, not whether the tree is balanced. Maintaining balance is really done only for the sake of reaching this time efficiency goal.

The splay tree access functions operate in a manner reminiscent of the move-to-front rule for self-organising lists (see Section 7.2), and of the path compression technique for managing a series of Union/Find operations (see Section 11.5). These access functions tend to make the tree more balanced, but an individual access will not necessarily result in a more balanced tree.

Whenever a node S is accessed (e.g., when S is inserted, deleted, or is the goal of a search), the splay tree performs a process called splaying. Splaying moves S to the root of the BST. When S is being deleted, splaying moves the parent of S to the root. As in the AVL tree, a splay of node S consists of a series of rotations. A rotation moves S higher in the tree by adjusting its position with respect to its parent and grandparent. A side effect of the rotations is a tendency to balance the tree. There are three types of rotation.

11.4.1 SPLAYING

A single rotation is performed only if S is a child of the root node. The single rotation is illustrated by Figure 11.7. It basically switches S with its parent in a way that retains the BST property. While this figure is slightly different from Figure 11.5, in fact the splay tree single rotation is identical to the AVL tree single rotation.

Unlike the AVL tree, the splay tree requires two types of double rotation.

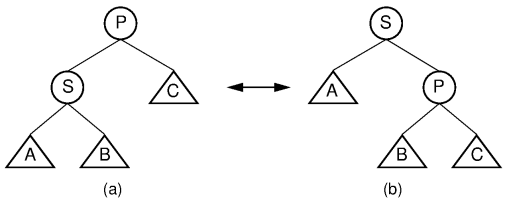


Figure 11.7 Splay tree single rotation. This rotation takes place only when the node being splayed is a child of the root. Here, node S is promoted to the root, rotating with node P. Because the value of S is less than the value of P, P must become S's right child. The positions of subtrees A, B, and C are altered as appropriate to maintain the BST property, but the contents of these subtrees remains unchanged. (a) The original tree with P as the parent. (b) The tree after a rotation takes place. Performing a single rotation a second time will return the tree to its original shape. Equivalently, if (b) is the initial configuration of the tree (i.e., S is at the root and P is its right child), then (a) shows the result of a single rotation to splay P to the root.

Double rotations involve S, its parent (call it P), and S's grandparent (call it G). The effect of a double rotation is to move S up two levels in the tree.

ZigZag The first double rotation is called a *zigzag rotation*. It takes place when either of the following two conditions are met:

(1) S is the left child of P, and P is the right child of G.
(2) S is the right child of P, and P is the left child of G.

In other words, a zigzag rotation is used when G, P, and S form a zigzag. The zigzag rotation is illustrated by Figure 11.8.

ZigZig The other double rotation is known as a zigzig rotation. It takes place when either of the following two conditions are met:

(1) S is the left child of P, which is in turn the left child of G.
(2) S is the right child of P, which is in turn the right child of G.

Thus, a zigzig rotation takes place in those situations where a zigzag rotation is not appropriate. The zigzig rotation is illustrated by Figure 11.9.

Note that zigzag rotations tend to make the tree more balanced, because they bring subtrees B and C up one level while moving subtree D down one level. The result is often a reduction of the tree's height by one. While Figure 11.8 appears somewhat different from Figure 11.6, in fact the zigzag rotation is identical to the AVL tree double rotation. Zigzig promotions and single rotations do not typically reduce the height of the tree; they merely bring the newly accessed record toward the root.

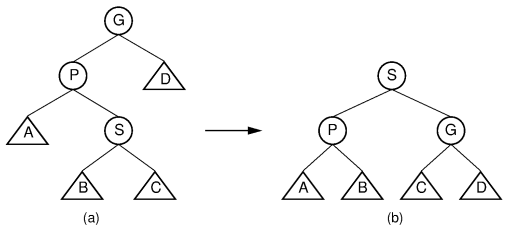


Figure 11.8 Splay tree zigzag rotation. (a) The original tree with S, P, and G in zigzag formation. (b) The tree after the rotation takes place. The positions of subtrees A, B, C, and D are altered as appropriate to maintain the BST property.

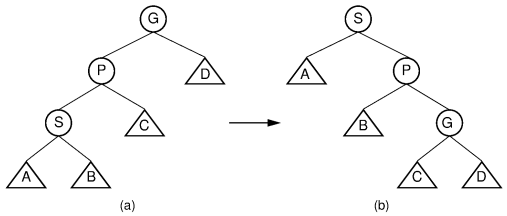


Figure 11.9 Splay tree zigzig rotation. (a) The original tree with S, P, and G in zigzig formation. (b) The tree after the rotation takes place. The positions of subtrees A, B, C, and D are altered as appropriate to maintain the BST property.

11.4.2 SEARCHING IN A SPLAY TREE

The splay tree's search operation is identical to searching in a BST. However, once the value has been found, it is splayed to the root. This means that when you search for a value *the tree structure is changed*. This is a big difference between splay trees and the search trees we have looked at previously.

Splaying a node involves a series of double rotations until the node reaches either the root or the child of the root. Then, if necessary, a single rotation makes it the root. This process tends to re-balance the tree. Regardless of balance, splaying will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost. Proof that the splay tree meets the guarantee of $O(m \log n)$ is beyond the scope of our study.

Section 11.4.2

Read the rest online

11.5 Disjoint sets and the Union/Find algorithm

General trees (Section 8.6) are trees whose internal nodes have no fixed number of children. Compared to general trees, binary trees are relatively easy to implement because each internal node of a binary tree can just store two pointers to

reach its (potential) children. In a general tree, we have to deal with the fact that a given node might have no children or few children or many children.

Even in a general tree, each node can have only one parent. If we didn't need to go from a node to its children, but instead only needed to go from a node to its parent, then implementing a node would be easy. A simple way to represent such a general tree would be to store for each node only a pointer to that node's parent. We will call this the parent pointer representation for general trees. Clearly this implementation is not general purpose, because it is inadequate for such important operations as finding the leftmost child or the right sibling for a node. Thus, it may seem to be a poor idea to implement a general tree in this way. However, the parent pointer implementation stores precisely the information required to answer the following, useful question: ***Given two nodes, are they in the same tree?***

Section 11.5

Read the rest online

11.6 Skip lists

This section presents a probabilistic search structure called the skip list. Like the BST, skip lists are designed to overcome a basic limitation of array-based and linked lists: Either search or update operations require linear time. The skip list is an example of a probabilistic data structure, because it makes some of its decisions at random.

Skip lists provide an alternative to the BST and related tree structures. The primary problem with the BST is that it may easily become unbalanced. The 2-3 Tree is guaranteed to remain balanced regardless of the order in which data values are inserted, but it is rather complicated to implement. The AVL tree and the splay tree are also guaranteed to provide good performance, but at the cost of added complexity as compared to the BST. The skip list is easier to implement than known balanced tree structures. The skip list is not guaranteed to provide good performance (where good performance is defined as $O(\log n)$ search, insertion, and deletion time), but it will provide good performance with extremely high probability (unlike the BST which has a good chance of performing poorly). As such it represents a good compromise between difficulty of implementation and performance.

Section 11.6

Read the rest online

11.7 Review questions

This final section contains some review questions about the contents of this chapter.

Section 11.7

Answer quiz online

CHAPTER 12

Hash tables

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is better than the $O(\log n)$ cost required to do a binary search on a sorted array of n records, or the $O(\log n)$ cost required to do an operation on a self-balancing binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a hash table, which we will call `HT` below. Hashing works by performing a computation on a search key k in a way that is intended to identify the position in `HT` that contains the record with key k. The function that does this calculation is called the hash function, and will be denoted by **h**. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a slot. The number of slots in hash table `HT` will be denoted by the variable M with slots numbered from 0 to $M - 1$.

The goal for a hashing system is to arrange things such that, for any key value k and some hash function **h**, $i = \mathbf{h}(k)$ is a slot in the table such that $0 \leq i < M$, and we have the key of the record stored at $\texttt{HT}[i]$ equal to k.

Since the records are not ordered by value, hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range. Nor can we easily find the record with the minimum or maximum key value, or visit the records in key order. Hashing is most appropriate for answering the question, "What record, if any, has key value k?" **For applications where all search is done by exact-match queries, hashing is a very good search method because it is extremely efficient when implemented correctly.** As we will see in this chapter, however, there are many approaches to hashing and it is easy to devise an inefficient implementation.

Hashing is suitable for both in-memory and disk-based searching and is one of the two most widely used methods for organising large databases stored on disk (the other is the B-tree).

In most applications, there are many more values in the key range than there are slots in the hash table. For a more realistic example, suppose the key can take any value in the range 0 to 65,535 (i.e., the key is a two-byte unsigned integer), and that we expect to store approximately 1000 records at any given time. It is impractical in this situation to use a hash table with 65,536 slots, because then the vast majority of the slots would be left empty. Instead, we must devise a hash function that allows us to store the records in a much smaller table. Because the key range is larger than the size of the table, at least some of the slots must be mapped to from multiple key values. Given a hash function $\mathbf{h}$ and two keys k_1 and k_2, if $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$ where β is a slot in the table, then we say that k_1 and k_2 have a collision at slot β under hash function $\mathbf{h}$.

Finding a record with key value k in a database organised by hashing follows a two-step procedure:

1. Compute the table location $\mathbf{h}(k)$.
2. Starting with slot $\mathbf{h}(k)$, locate the record containing key k using (if necessary) a collision resolution policy.

12.1 Hash functions

Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots. Collisions occur when two records hash to the same slot in the table. If we are careful – or lucky – when selecting a hash function, then the actual number of collisions will be few. Unfortunately, even under the best of circumstances, collisions are nearly unavoidable. To illustrate, consider a classroom full of students. What is the probability that some pair of students shares the same birthday (i.e., the same day of the year, not necessarily the same year)? If there are 23 students, then the odds are about even that two will share a birthday. This is despite the fact that there are 365 days in which students can have birthdays (ignoring leap years). On most days, no student in the class has a birthday. With more students, the probability of a shared birthday increases. The mapping of students to days based on their birthday is similar to assigning records to slots in a table (of size 365) using the birthday as a hash function. Note that this observation tells us nothing about *which* students share a birthday, or on *which* days of the year shared birthdays fall.

To be practical, a database organised by hashing must store records in a hash

table that is not so large that it wastes space. To balance time and space efficiency, this means that the hash table should be around half full (see Section 12.9). Because collisions are extremely likely to occur under these conditions (by chance, any record inserted into a table that is half full should have a collision half of the time), does this mean that we need not worry about how well a hash function does at avoiding collisions? Absolutely not. The difference between using a good hash function and a bad hash function makes a big difference in practice in the number of records that must be examined when searching or inserting to the table. Technically, any function that maps all possible key values to a slot in the hash table is a hash function. In the extreme case, even a function that maps all records to the same slot in the array is a hash function, but it does nothing to help us find records during a search operation.

We would like to pick a hash function that maps keys to slots in a way that makes each slot in the hash table have equal probablility of being filled for the actual set keys being used. Unfortunately, we normally have no control over the distribution of key values for the actual records in a given database or collection. So how well any particular hash function does depends on the actual distribution of the keys used within the allowable key range. In some cases, incoming data are well distributed across their key range. For example, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table. However, in many applications the incoming records are highly clustered or otherwise poorly distributed. When input records are not well distributed throughout the key range it can be difficult to devise a hash function that does a good job of distributing the records throughout the table, especially if the input distribution is not known in advance.

There are many reasons why data values might be poorly distributed.

1. Natural frequency distributions tend to follow a common pattern where a few of the entities occur frequently while most entities occur relatively rarely. For example, consider the populations of the 100 largest cities in the United States. If you plot these populations on a numberline, most of them will be clustered toward the low side, with a few outliers on the high side. This is an example of a Zipf distribution. Viewed the other way, the home town for a given person is far more likely to be a particular large city than a particular small town.
2. Collected data are likely to be skewed in some way. Field samples might be rounded to, say, the nearest 5 (i.e., all numbers end in 5 or 0).

3. If the input is a collection of common English words, the beginning letter will be poorly distributed.

Note that for items 2 and 3 on this list, either high- or low-order bits of the key are poorly distributed.

When designing hash functions, we are generally faced with one of two situations:

1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table, while avoiding obvious opportunities for clustering such as hash functions that are sensitive to the high- or low-order bits of the key value.
2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should *not* hash on the value of the first character because this is likely to be unevenly distributed.

In the rest of this section, and in Section 12.10, you will see several examples of hash functions that illustrate these points.

12.1.1 PROPERTIES OF A HASH FUNCTION

Any hash function **h** must respect the following properties:

- it must be *deterministic*: $\mathbf{h}(a)$ must always return the same value for a given a, regardless of when it is called
- it must *preserve equality*: if a is equal to b, then $\mathbf{h}(a)$ must be equal to $\mathbf{h}(b)$

Furthermore, a *good* hash function should also respect the following:

- *uniform distribution*: **h** should map the expected inputs as evenly as possible over the possible output values
- *efficiency*: the hash function should be fast to calculate

Example: The worst hash function

The requirements above tells us that the following is a valid hash function, because is both deterministic and preserves equality:

$$\mathbf{h}(x) \quad = \quad 42$$

However, it's a really bad hash function, because it gives the worst possible distribution.

This means that if you implement a hash table which uses this constant hash function, it will still work. It will answer all your questions correctly, and insertion and deletion will work just as expected. But it will be extremely slow – as slow as if you had used a simple list of values.

12.1.2 BASIC PRINCIPLES OF HASH FUNCTIONS

Here are some basic principles for creating good (or at least mostly-good-enough) hash functions.

Integers, characters and enumerations Since the value of a hash function should be an integer, it is very easy to just let value be its own hash code, if it's an integer.

$$\mathbf{h}(x) \quad = \quad x$$

This function is clearly deterministic and preserves equality, and it is very efficient too. But how about the distribution – is it uniform?

This depends on how the keys are distributed in the data, and this depends on the application we have in mind. As an example, assume that we only have even-numbered keys in our data – then only half of all possible hash codes will be used.

A more realistic example is if we want to search for people using their length. Person lengths are not uniformly distributed, but instead they have a normal distribution. This means that the hash codes will also be normal distributed, and there will be many collisions for lengths that are closer to the average length.

However, despite the possibly skewed distribution, it is still quite common to use a number as its own hash code. The reason for this is that it is so simple and efficient to do so.

Note that this simple strategy works for all kinds of atomic values, such as Unicode characters or enumeration types. All these are anyway encoded as integers internally by the compiler.

Floating point numbers There is an international standard for encoding floating point numbers, called IEEE 754, which almost all modern processors use. The details of the encoding are not important, what matters to us is that it encodes every floating point number in a fixed number of bytes.

One very easy and common hash function for floating point numbers is to simply use it as it is, but interpret the byte sequence as an integer. This means that the floating point value 42.0 will *not* have the hash code 42, but instead something completely different. However, this is not a problem in our case since the only thing we need to know is that it respects the requirements of a hash function. Just as above, it is clearly both efficient, deterministic and preserves equality, and the distribution is most definitely not worse than the integer distribution.

Strings But how can we handle more complex values, such as strings? A string is a sequence of characters, of arbitrary length, so how can we calculate a good hash code for a given string?

- How about taking the hash code of the first character in the string? No, that's not a good idea because you will get an awful distribution – all strings starting with "a" will get the same hash code, such as "abacus", "ape", and "aperture".
- So, perhaps we can take the sum of the first, the middle and the last characters? That's better but still not very good – common strings such as "state", "summarise" and "signature" will get the same hash code.

In both these cases the hash function will have a very skewed distribution, and the main reason is that they don't take every single character into account. So an important rule of thumb is to use every part of a complex object when calculating the hash code. But how should these hash codes be combined? One way is to just sum all of them:

```
function hashCode(string):
    code = 0
    for each char in string:
        code = code + hashCode(char)
    return code
```

This is much better than the previous suggestions, but still not very good. Each character in the string will influence the final hash code, but it does not take their internal order into account. Therefore all the following strings will get the same hash code: "alter", "later", and "alert".

Horner's method for strings, lists and other sequences So we want our hash function to take all character in a string into account, but also their position in the string. One common way is to treat the sequence of characters, $s = c_0 c_1 \dots c_n$ as a polynomial over some constant p, like this:

$$\mathbf{h}(c_0 c_1 c_2 \dots c_n) \quad = \quad c_0 + c_1 p + c_2 p^2 + c_3 p^3 + \cdots + c_n p^n$$

This kind of polynomial can be calculated efficiently using *Horner's method*, as a nested multiplication:

$$\mathbf{h}(c_0c_1c_2\ldots c_n) \quad = \quad c_0 + p\,(c_1 + p\,(c_2 + p\,(c_3 + \cdots + p(c_{n-1} + pc_n)\cdots)))$$

Which in turn is just a fancier way of writing a simple loop over all characters, and accumulating the hash code:

```
function hashCode(string):
    code = 0
    for each char in string:
        code = p*code + hashCode(char)
    return code
```

To get the best distribution, we should use a not-too-small prime number for p. (The standard hash function for strings in Java is exactly like this, where they use $p = 31$.)

Note that Horner's method can be used for all kinds of sequences, such as lists or tuples.

Complex values So how can we handle complex values, such as general datatypes or objects? E.g., how do we define a hash function for the following datatype for person names:

```
datatype Person:
    firstName: String
    middleName: String
    lastName: String
```

We can use the same strategy as we did for strings – simply treat the three elements as a sequence and apply Horner's method:

```
datatype Person:
    ...
    hashCode():
        code = 0
        for part in (firstName, middleName, lastName):
            code = p*code + part.hashCode()
        return code
```

However, sometimes we don't want to use all instance variables when calculating the hash code. Assume that we want to be able to search for just the last name, i.e., to find all persons having the same last name. Then we can build a hash table of persons, where the hash codes are only calculated from the last names. If we want to also be able to search for first names, we have to build *another* hash table – where the hash codes are calculated from the first names.

12.2 Converting objects to table indices

We want to be able to build hash tables for any kind of object – be it a string, a complex datatype of persons, or an x-ray image. But we (i.e., the ones implementing the hash table) do not know which kind of object the user (i.e., the one using the hash table) wants to put in their hash table. Therefore the easiest is to *delegate* the implementation of the hash function to the object itself. By this we mean that every datatype (strings, persons, x-ray images) should have a designated method that turns a specific object into an integer, just as for the Person datatype we showed in the end of the last section:

```
datatype Person:
    firstName: String
    middleName: String
    lastName: String

    hashCode() -> Integer:
        code = 0
        for part in (firstName, middleName, lastName):
            code = p*code + part.hashCode()
        return code
```

This is a very common strategy in many programming languages – e.g., in Java this designated method is called exactly `hashCode()`, and in Python it is called `__hash__()`.

12.2.1 COMPRESSING A HASH CODE

Internally, a hash table is an array of a fixed length, say M, and each of these M slots represent a bucket which contains some objects. The hash function should help us by telling in which bucket we should search for the given object.

Now, the problem is that the datatypes and objects have no idea how large the internal array is, i.e., which array index they should return. Instead, the only thing that the `hashCode()` method can do is to return an arbitrary integer. So, how can we solve this dilemma?

The solution is to implement a function that transforms an arbitrary hash code into an array index. This is called *compression* – we compress the arbitrary integer that is the hash code into an array index $0 \leq i < M$.

Therefore, computing the hash table index for a given object is a two-step process:

1. First calculate the hash code which is an arbitrary integer – this is calculated by the object itself.

2. Then compress the hash code into a table index $0 \leq i < M$ – this is calculated by an internal function in the hash table.

12.2.2 MODULAR COMPRESSION

The simplest way to compress a hash code $h \geq 0$ into a table index i, is to take h **modulo** the array size M:

$$i \quad = \quad h \,\%\, M$$

This is called *modular compression* and is the most common compression method.

Recall that it is common to use an integer as its own hash code. Now we see that this is not the full picture, because the integer will then be compressed into the internal array of the hash table. Assume that we have an integer hash table of sixteen slots – then the combined hash function will be the same as the modular compression:

```
function hash(n):
    return n % 16
```

Note that the values 0 to 15 can be represented with four bits (i.e., 0000 to 1111). The value returned by this hash function depends solely on the least significant four bits of the key. Because these bits are likely to be poorly distributed (as an example, a high percentage of the keys might be even numbers, which means that the low order bit is zero), the result will also be poorly distributed. This example shows that the size of the table M can have a big effect on the performance of a hash system.

One way to get a better distribution is to always let the size M of the internal array be a prime number.

12.2.3 NEGATIVE HASH CODES

However, in general integers are signed, so the method `hashCode()` might return a negative integer. If we take this modulo M, we might get a negative result. A negative index is not suitable as a table index, so first we have to make the hash code positive.

One way to do this is to mask off the sign bit. Most programming languages use integers in the range $-2^{31} \leq h < 2^{31}$. In these cases we can e.g. use `h & 0x7fffffff` to make the hash code positive.

```
function compress(h):
    h = h & 0x7fffffff
    return h % M
```

12.2.4 THE HASH CODE NEVER CHANGES

The generic hash codes should never change, because hashing must be predictable. Therefore we don't have to recalculate the hash code when we resize the internal table, it is only the table indices that have to be updated.

One implication of this is that it's only meaningful to calculate hash codes for *immutable objects*, i.e., objects that don't change (after they are initialised). Many programming languages make a difference between *tuples* and *lists*, where the latter are *mutable*. This means that we can add elements to and remove from lists, but we cannot do that with tuples. Once the tuple is initialised we cannot change it – and therefore we can use a tuple as an object in a hash table.

Python uses this fact to optimise their built-in hash tables by storing the hash codes together with the keys and values. This increases the size of the table slightly, but on the other hand it ensures that hash codes are not calculated more than once.

In Java, the optimisation is delegated to the object classes themselves. E.g., a Java string only calculates its hash code once and then stores it in an instance variable for immediate lookup.

12.2.5 HANDLING COLLISIONS

So far we have talked about how to calculate an array index for an arbitrary object. One thing we haven't discussed yet is what we should do if two two different objects get assigned the same slots in the table.

Assume that we have an internal array of size $M = 16$, and we use modular compression like above. Then every 16th integer will get the same array index, such as the integers 7, 23, 39, 55, etc.

So, one very important question is how to handle *collision*, i.e., when two objects want to occupy the same slot. There are two main approaches to this, two *collision resolution strategies*:

Separate chaining The internal array contains pointers to *collections* of objects – it could be linked lists, self-balancing trees, or some other suitable data structure.

Open addressing The internal array contains the objects themselves. Collisions are handled by *probing* the array – i.e., searching through the table to find an empty slot.

Both of these are common – e.g., the Java standard library implements a separate chaining hash table, while the Python sets and dictionaries are implemented using open addressing hash tables.

We will start by describing separate chaining hash tables in Section 12.3, and then continue with open addressing in Section 12.5.

12.3 Separate chaining

While the goal of a hash function is to minimise collisions, some collisions are unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. Collision resolution techniques can be broken into two classes: separate chaining (also called open hashing) and open addressing (also called closed hashing). (Yes, it is confusing when "open hashing" means the opposite of "open addressing", but unfortunately, that is the way it is.) The difference between the two has to do with whether collisions are stored outside the table (separate chaining), or whether collisions result in storing one of the records at another slot in the table (open addressing).

The simplest form of separate chaining defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. The following figure illustrates a hash table where each slot points to a linked list to hold the records associated with that slot. The hash function used is the simple mod function.

Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because we know to stop searching the list once we encounter a key that is greater than the one being searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. This value, N/M, is commonly called the load factor.

Assuming that the table has more slots than there are records to be stored, we can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $O(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Separate chaining is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing a separate chaining hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This

would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

There are similarities between separate chaining and Binsort. One way to view separate chaining is that each record is simply placed in a bin. While multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation. In a similar fashion, a simple Binsort reduces the number of records in each bin to a small number that can be sorted in some other way.

12.3.1 ALTERNATIVES TO A LINKED LIST

There is nothing that requires us to use a linked list as the underlying data structure, it could be a dynamic array or a balanced search tree too. (In fact, Java's hash tables use a combination of linked lists and balanced trees.)

Conceptually, a hash table can use any kind of collection data structure – the only thing that the actual array does is to partition the large collection into M disjoint collections. If the hash function is good and distributes the objects evenly among the bins, all operations will become M times faster (because the bins are M times smaller than the original large collection).

12.3.2 RESIZING IS IMPORTANT

Just as for dynamic arrays, it is important that we resize the internal table when it becomes too large (or too small). That is, we change the size M so that it is proportional to the number of table entries.

If M is always proportional to the number of entries, *and* if we have a good hash function, the number of elements in a bin will remain approximately constant. And then all operations will be expected constant time.

12.3.3 IMPLEMENTATION

Now we will give a very generic overview how to implement a separate chaining hash table. In the next section we will discuss implementation in more detail.

As already mentioned, a separate chaining hash table *partitions* its elements into an array of smaller collections, or *bins*.

```
datatype SeparateChainingHashTable:
    table: Array of Collections
```

Note that we don't have to know what type of collections the bins are, the only thing we have to know is that they support the same methods that we want

the hash table to support. To implement any kind of method, we first have to to decide on a bin and then *delegate* the method to that bin. We use the hash function to decide which bin to delegate to, and then simply call the method of that bin, with the same arguments as we got in the first place:

```
datatype SeparateChainingHashTable:
    ...
    method(elem, ...extra arguments...):
        bin = table[hashIndex(elem)]
        return bin.method(elem, ...extra arguments...)
```

As explained in the previous chapter, the hash index for an element consists of first getting the hash code and then compressing it to an array index.

```
datatype SeparateChainingHashTable:
    ...
    hashIndex(elem):
        hash = elem.hashCode()
        hash = hash & 0x7fffffff   // make the hash code non-negative
        return hash % table.size
```

There are of course some more details one has to take care of to get a working implementation. For example, we have not discussed how to calculate the size of the hash table, i.e., the total number of elements. We have also not discussed how to handle bins that are not initialised yet.

12.3.4 LOAD FACTOR AND RESIZING

The efficiency of a hash table depends on two factors:

1. how well the elements are distributed between the bins
2. the size of each bin

The first factor depends on the hash function, and for now we simply assume that it is good – meaning that it distributes the elements evenly among the bins. The second factor depends on the load factor – if there are N elements in total and we have M bins, then there are N/M elements per bin on average. Assuming that our underlying collections are simple linked lists, then all main operations (searching, adding, deleting) are linear in the load factor.

The key point to making hash tables efficient is to make sure that the load factor never becomes larger than a constant – the *maximum load factor*. If we have a constant load factor and a good hash function, then all operations on hash tables will be constant time, $O(1)$.

To be able to do this we have to *resize* the internal table when the number of elements grow. But we already know how to do this – use a dynamic array! So, whenever we implement a method that can change the number of elements, we have to check if we need to resize the table. Here is how we can do this:

```
datatype SeparateChainingHashTable:
    ...
    method(elem, ...extra arguments...):
        bin = table[hashIndex(elem)]
        result = bin.method(elem, ...extra arguments...)
        if load factor is too large:
            resizeTable(table.size * MULTIPLIER)
```

Recall from the dynamic arrays that we have to *multiply* by a factor (the `MULTIPLIER`) when resizing the table, not adding a constant.

When we resize the internal table, it is very important that we *do not keep* the old hash indices for the keys, because they will not be the same after resizing. Instead we save the current internal table to a temporary variable, and reinitialise the table to the new capacity. Then we iterate through all bins and entries in the old table, and simply insert them again into the new resized table.

```
datatype SeparateChainingHashTable:
    ...
    resizeTable(newCapacity):
        oldTable = table                // Remember the old table.
        table = new Array(newCapacity)  // Reset the internal table.
        size = 0                        // Reset the number of elements.
        for each bin in oldTable:
            add each elem in bin
```

12.4 Implementing sets and maps using separate chaining

In this section we go into more details in how to get a working implementation. First we show how to implement a *hash set*, an then we discuss how to extend this into a *hash map*.

12.4.1 IMPLEMENTING SETS

A separate chaining hash set consists of an internal array `table` of sets. We don't have to specify what kind of sets, as long as it supports the basic set methods. Usually it's perfectly fine to use a very simple linked list and not something

more fancy. To initialise the table, we first create the internal array of some initial minimum capacity:

```
datatype SeparateChainingHashSet implements Set:
    table: Array of Sets
    size: Int

    constructor():
        initialise(MIN_CAPACITY)
```

To simplify things we put the initialisation in a method of its own, because we will reuse it later when resizing the table. The initialisation not only creates the internal table, but also populates it with new empty sets.

```
datatype SeparateChainingHashSet:
    ...
    initialise(capacity):
        size = 0
        table = new Array(capacity)
        for i in 0 .. capacity-1:
            table[i] = new Set()
```

Note that we keep the total number of elements in a variable `size`. It is possible to calculate this value by summing the sizes of all bins, but this takes time so we remember it in a variable instead. This also means that we have to update the variable whenever the size of a bin changes.

Searching for an element To see if the set contains a given element, we look it up in the corresponding bin:

```
datatype SeparateChainingHashSet:
    ...
    contains(elem):
        bin = table[hashIndex(elem)]
        return bin.contains(elem)
```

Adding an element To add an element we add it to the bin where it should belong. If the size of the bin changed, we know that the key wasn't in the bin previously, and then we know that the number of elements have increased. We also have to check if the load factor becomes too large, and then we resize the internal table.

```
datatype SeparateChainingHashSet:
    ...
    add(elem):
```

```
        bin = table[hashIndex(elem)]
        oldBinSize = bin.size
        bin.add(elem)
        if bin.size > oldBinSize:
            size = size + 1
            if loadFactor() > MAX_LOAD_FACTOR:
                resizeTable(table.size * MULTIPLIER)
```

Removing an element To remove a value, we do the same: find the underlying bin and remove the value. If we actually removed the element (i.e., if it existed in the bin), then we decrease the total size. We also check if the table becomes too sparse, and then decrease the internal table by a factor.

```
datatype SeparateChainingHashSet:
    ...
    remove(elem):
        bin = table[hashIndex(elem)]
        oldBinSize = bin.size
        bin.remove(elem)
        if bin.size < oldBinSize:
            size = size - 1
            if loadFactor() < MIN_LOAD_FACTOR:
                resizeTable(table.size / MULTIPLIER)
```

Load factor and constants The load factor is simply the total number of elements divided by the number of bins, or N/M:

```
datatype SeparateChainingHashSet:
    ...
    loadFactor():
        return size / table.size
```

The constants for min and max load factors, and the resizing factor, are a bit arbitrary. With the following values, we ensure that the table on average contains between 0.5 and 2 entries per table index. Increasing these values leads to more better memory usage, but also more conflicts (i.e., longer search times). Also, we enlarge by 50%, or reduce by 33%, each time we resize the table. Increasing this value means that resizing will happen less often, but instead the memory usage will increase.

```
datatype SeparateChainingHashSet:
    ...
    MIN_CAPACITY = 8
    MIN_LOAD_FACTOR = 0.5
    MAX_LOAD_FACTOR = 2.0
    MULTIPLIER = 1.5
```

Resizing the internal table When we resize the internal table, it is very important that we *do not keep* the old hash indices for the keys, because they will not be the same after resizing. Instead we save the current internal table to a temporary variable, and reinitialise the table to the new capacity. Then we iterate through all bins and entries in the old table, and simply insert them again into the new resized table.

```
datatype SeparateChainingHashSet:
    ...
    resizeTable(newCapacity):
        if newCapacity >= MIN_CAPACITY:
            oldTable = table
            initialise(newCapacity)
            for each bin in oldTable:
                for each elem in bin:
                    add(elem)
```

12.4.2 IMPLEMENTING MAPS

It is straightforward to modify the implementation above to become a key-value map instead of a set.

```
datatype SeparateChainingHashMap implements Map:
    table: Array of Maps
    ...

    get(key):
        // Similar to contains for hash sets, but use Map.get instead:
        ...
        return bin.get(key)

    put(key, value):
        // Similar to add for hash sets, but use Map.put instead:
        ...
        bin.put(key, value)
        ...

    remove(key):
        // Similar to remove for hash sets, but use Map.remove instead:
        ...
        bin.remove(key)
        ...

    resizeTable(newCapacity):
        // Similar to resizeTable for hash sets, but use put instead:
```

```
    ...
        for each key, value in bin:
            put(key, value)
```

12.5 Open addressing

We now turn to the other commonly used form of hashing: open addressing (also called closed hashing).

Compared to separate chaining (Section 12.3), we now store all elements directly in the hash table. Each element E has a home position that is $\mathbf{h}(E)$, the slot computed by the hash function. If E is to be inserted and another record already occupies its home position, then E will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any element not found in its home position can be recovered by repeating the collision resolution process.

We will still denote the size of the internal arraw as M and the total number of elements as N, and the load factor is still calculated in the same way, as N/M. Note that since all elements are stored in the array, N can never be larger than M, so the load factor must always be smaller than 1.

Implementation overview The internal table in separate chaining consists of pointers to a linked list (or some other kind of set data structure). Open addressing, in contrast, stores the elements themselves directly in the table:

```
datatype OpenAddressingHashTable of Elem:
    table: Array of Elem
```

To implement a method on an open addressing hash table we first have to find the correct table index for the element in question, and then we can apply the method on the slot.

```
datatype OpenAddressingHashTable:
    ...
    method(elem, ...extra arguments...):
        i = hashAndProbe(elem)
        apply method to table[i]
```

Hashing and probing The goal of collision resolution is to find a free slot in the hash table when the "home position" for the record is already occupied. We can view any collision resolution method as generating a sequence of hash table slots that can potentially hold the record. The first slot in the sequence will be the

home position for the key. If the home position is occupied, then the collision resolution policy goes to the next slot in the sequence. If this is occupied as well, then another slot must be found, and so on. This sequence of slots is known as the probe sequence, and it is generated by some probe function that we will call **p**.

The simplest probe function is *linear probing*, which tries all consecutive positions starting from the home position. Once the bottom of the table is reached, the probe sequence wraps around to the beginning of the table (since the last step is to mod the result to the table size). Linear probing has the virtue that all slots in the table will be candidates for inserting a new record before the probe sequence returns to the home position.

In the following code we iterate through all possible *offsets* 0, 1, 2, ..., etc.

```
datatype OpenAddressingHashTable:
    ...
    hashAndProbe(elem):
        home = hashIndex(elem)
        for offset in 0 .. table.size-1:
            i = (home + offset) % table.size
            if table[i] == elem or table[i] is null:
                return i
        throw error "Hash table full"
```

Linear probing has some drawbacks, and there are several alternative strategies which we will discuss later in Section 12.8.

Finding and inserting Searching for an element in an open addressing hash table is straightforward: `hashAndProbe` returns a slot, and all we have to do is to check that it points to the correct element.

```
datatype OpenAddressingHashTable:
    ...
    contains(elem):
        i = hashAndProbe(elem)
        return table[i] == elem
```

To add an element we first find the correct slot, and then we set the value in that slot:

```
datatype OpenAddressingHashTable:
    ...
    add(elem):
        i = hashAndProbe(elem)
        table[i] = elem
```

Collisions and clusters When the table gets more and more populated, it starts building *clusters* – sequences of neighbouring non-empty slots in the table. The efficiency of the basic operations depend crucially on the size of the largest cluster. Assume we want to search for an element that's not in the table, but happens to have its home position as the very first slot in a large cluster. Then we have to inspect every slot in the cluster before we can be certain that the element is not there.

Note that a cluster can consist of elements that all have different hash codes and different home positions – as long as the neighbouring slots are occupied they all belong to the same cluster. Therefore it's not only important that the hash function gives different hash codes for different elements – ideally the hash codes should be spread apart as much as possible, even if the elements themselves are very close to each other.

It is possible to reduce some clustering by using different probing strategies than simple linear probing, see Section 12.8 for more details.

Deleting The difficult part when implementing deletion is how to handle clusters.

Assume for example that we have a cluster of five elements $[A, B, C, D, E]$, and we want to remove C. If we simply clear the slot we will get the two clusters $[A, B]$ and $[D, E]$. Let us also assume that the home positions of D is in the beginning of the original large cluster – i.e., A and D have the same home positions.

Now, what happens if we try to search for D in the table, after we have removed C? It will start searching in the first cluster and inspect slots A and B, but then it encounters an empty slot and stops. So it will report that D is not in the table, even though it hasn't been deleted.

Therefore we cannot just clear slots that we want to delete. The simplest solution is instead to *mark* the slot in some way. The most common way is to use a special value which is not used anywhere else, which will be the new value of the slot. This special `DELETED` value is sometimes called a tombstone.

```
datatype OpenAddressingHashTable:
    ...
    delete(elem):
        i = hashAndProbe(elem)
        table[i] = DELETED
```

Things are not exactly as simple as they look here, there are some details that are very important to get right, we will discuss these details later in Section 12.7.

12.6 Implementing sets and maps using open addressing

An open addressing hash set consists of an internal array `table` of elements. Apart from the special value `null`, denoting an empty slot, the table must also allow the special value `DELETED`, denoting a slot that has been deleted. To initialise the table, we first create the internal array of some initial minimum capacity:

```
datatype OpenAddressingHashSet implements Set:
    table = new Array(MIN_CAPACITY)
    size = 0
```

As already explained, all main methods (`contains`, `add` and `remove`) use the same probing function to get the same probe sequence. This is implemented by the method `hashAndProbe` that was shown in the previous section. In this way, a record not in its home position can always be recovered.

To search for an element we probe the table for a position and check if the slot contains the element we are looking for:

```
datatype OpenAddressingHashSet:
    ...
    contains(elem):
        i = hashAndProbe(elem)
        return table[i] == elem
```

When we want to add an element we first have to search for it. If the element isn't found, the probing returns the index to an empty slot, which we then can assign the element. Now we have to increase the size of the set, and check if we have exceeded the maximum allowed load factor. If the load factor is too large, then we resize the internal table – which was explained in the previous section.

```
datatype OpenAddressingHashSet:
    ...
    add(elem):
        i = hashAndProbe(elem)
        if table[i] is null:
            table[i] = elem
            size = size + 1
            if loadFactor() > MAX_LOAD_FACTOR:
                resizeTable(table.size * MULTIPLIER)
```

Deleting from an open addressing hash table was explained briefly in the last section and we will go into more details in the next section.

We use the same constants as for the separate chaining hash table, but the values are different. Most importantly, the max load factor must be smaller than 1, since there can only be one value per array slot.

```
datatype OpenAddressingHashSet:
    ...
    MIN_CAPACITY = 8
    MIN_LOAD_FACTOR = 0.3
    MAX_LOAD_FACTOR = 0.7
    MULTIPLIER = 1.5
```

12.6.1 RESIZING

When we resize the internal table, it is very important that we *do not keep* the old hash indices for the keys, because they will not be the same after resizing. This is exactly the same as for separate chaining. Instead we save the current internal table to a temporary variable, and reinitialise the table to the new capacity. Then we iterate through all entries in the old table, and simply insert them again into the new resized table.

```
datatype SeparateChainingHashSet:
    ...
    resizeTable(newCapacity):
        oldTable = table                 // Remember the old table.
        table = new Array(newCapacity)   // Reset the internal table.
        size = 0                         // Reset the number of elements.
        for each elem in oldTable:
            if elem is not null and not DELETED:
                add elem
```

Now we can get rid of all the deleted slots, simply by not adding them to the new table.

12.6.2 IMPLEMENTING MAPS

If we want to implement a map instead of a set, we have to store both a key and a value in the same slot. This can be done in two ways: either one array of key-value pairs, or two arrays of the same length – one for the keys and another for the values. Here we will use the second approach.

```
datatype OpenAddressingHashMap implements Map:
    keys = new Array(MIN_CAPACITY)
    values = new Array(MIN_CAPACITY)
    size = 0
```

The changes that has to be made are these, compared to how we implemented sets:

- when searching in the hash table, we have to use the `keys` array (which was the `table` array for the hash set)

- when getting and setting the value, we use the same index in the `values` array

Searching is straightforward:

```
datatype OpenAddressingHashSet:
    ...
    get(key):
        i = hashAndProbe(key)
        if keys[i] == key:
            return values[i]
```

Setting the value for a key becomes slightly longer, because we have to check if the key already exists or not. If the key doesn't exist we have to increase the size and check if we need to resize the tables.

```
datatype OpenAddressingHashSet:
    ...
    put(key, value):
        i = hashAndProbe(key)
        values[i] = value
        if keys[i] is null:
            keys[i] = key
            size = size + 1
            if loadFactor() > MAX_LOAD_FACTOR:
                resizeTable(keys.size * MULTIPLIER)
```

Reisizing is very similar to that for sets, we just have to remember to reset both internal tables (`keys` and `values`).

12.7 Deletion in open addressing

When deleting records from a hash table, there are two important considerations.

1. Deleting a record must not hinder later searches. In other words, the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. Thus, the delete process cannot simply clear the slot, because this will isolate records further down the probe sequence.
2. We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a tombstone. The tombstone indicates that a record once occupied the slot but does so no longer. If a tombstone is encountered

when searching along a probe sequence, the search procedure continues with the search. When a tombstone is encountered during insertion, that slot can be used to store the new record. However, to avoid inserting duplicate keys, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found, simply to verify that a duplicate is not in the table. However, the new record would actually be inserted into the slot of the first tombstone encountered.

The use of tombstones allows searches to work correctly and allows reuse of deleted slots. However, after a series of intermixed insertion and deletion operations, some slots will contain tombstones. This will tend to lengthen the average distance from a record's home position to the record itself, beyond where it could be if the tombstones did not exist. A typical database application will first load a collection of records into the hash table and then progress to a phase of intermixed insertions and deletions. After the table is loaded with the initial collection of records, the first few deletions will lengthen the average probe sequence distance for records (it will add tombstones). Over time, the average distance will reach an equilibrium point because insertions will tend to decrease the average distance by filling in tombstone slots. For example, after initially loading records into the database, the average path distance might be 1.2 (i.e., an average of 0.2 accesses per search beyond the home position will be required). After a series of insertions and deletions, this average distance might increase to 1.6 due to tombstones. This seems like a small increase, but it is three times longer on average beyond the home position than before deletions.

Two possible solutions to this problem are

1. Do a local reorganisation upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record (being careful not to remove any key from its probe sequence). This will not work for all collision resolution policies.
2. Periodically rehash the table by reinserting all records into a new hash table. Not only will this remove the tombstones, but it also provides an opportunity to place the most frequently accessed records into their home positions.

Note that since we are using a dynamic array when implementing hash tables, this can be viewed as a version of the second solution above (because all tombstones will be removed when the internal array is resized).

12.7.1 SIMPLE IMPLEMENTATION OF DELETION

If we are implementing a map (and not a set), then we actually don't need to use a special value `DELETED` to represent the tombstone. Instead, since we are using two internal arrays (one for the keys and one for the values), there are actually two possible ways of storing empty entries. We use this to encode both empty slots and tombstones:

- If the *keys* cell is empty (`keys[i] is null`), then the slot is unoccupied.
- If the *values* cell is empy (`values[i] is null`), then the slot is a tombstone.

So, when we remove an entry, we do not remove the key, but instead set the value to `null`. This will make the slot a tombstone.

```
datatype OpenAddressingHashMap:
    ...
    remove(key):
        i = hashAndProbe(key)
        if keys[i] is not null and values[i] is not null:
            // Make the slot a tombstone by setting the value to null.
            values[i] = null
            size = size - 1
            if loadFactor() < MIN_LOAD_FACTOR:
                resizeTable(keys.size * MULTIPLIER)
```

The current code has one problem: Adding new entries will never make use of the tombstones, but will only insert into completely empty slots. It is possible to fix this by implementing a sligthly different version of `hashAndProbe`, which will only be used by the `put` method. This is left as an exercise to the reader.

12.7.2 TWO LOAD FACTORS

There is however a bigger problem with our code above. When we remove keys we reduce the size of the hash table, but we haven't actually changed the number of occupied slot. Instead we turned a slot from having a value into a tombstone. So, the variable `size` – which is the number N of key/value pairs – does not say how many slots are actually occupied.

If we are very unlucky we might end up in a table that has a nice load factor, but where almost all slots are tombstones. But the table doesn't know this, so it will not try to resize and instead we will get a drop in performance.

To make deletion work properly together with resizing, we have to keep track of *two* instance variables – the number of occupied slots (`size`), and the number of tombstones:

```
datatype OpenAddressingHashMap implements Map:
    keys = new Array(MIN_CAPACITY)
    values = new Array(MIN_CAPACITY)
    size = 0
    tombstones = 0
```

When we have tombstones in our table, there are two possible ways of thinking about the load factor – depending on if we want to include the tombstones or not. And both variants are useful!

- When adding elements, we need to know if there are too few completely empty slots left, giving the load factor $N + D/M$ (where N is the number of occupied slots and D the number of tombstones).
- When deleting elements, we need to know if there are too few occupied slots, giving the load factor N/M.

All this combined result in slightly more complicated code for our methods. E.g., when deleting a key/value pair – or rather, turning the slot into a tombstone – we have to decrease the `size` variable, but we also have to increase the variable `tombstones`.

12.8 Different probing strategies

12.8.1 THE PROBLEM WITH LINEAR PROBING

While linear probing is probably the first idea that comes to mind when considering collision resolution policies, it is not the only one possible. Probe function **p** allows us many options for how to do collision resolution. In fact, linear probing is one of the worst collision resolution methods.

Again, the ideal behaviour for a collision resolution mechanism is that each empty slot in the table will have equal probability of receiving the next record inserted (assuming that every slot in the table has equal probability of being hashed to initially). This tendency of linear probing to cluster items together is known as primary clustering. Small clusters tend to merge into big clusters, making the problem worse.

The problem with primary clustering is that it leads to long probe sequences, which increases execution time. However, linear probing is still a very common probing method, because it is so simple and can be implemented efficiently.

12.8.2 LINEAR PROBING BY STEPS

How can we avoid primary clustering? One possible improvement might be to use linear probing, but to skip slots by some constant c other than 1. This would make the probe function $\mathbf{p}(K, i) = ci$, and so the i th slot in the probe sequence will be $(\mathbf{h}(K) + ic) \mod M$. In this way, records with adjacent home positions will not follow the same probe sequence.

One quality of a good probe sequence is that it will cycle through all slots in the hash table before returning to the home position. Clearly linear probing (which "skips" slots by one each time) does this. Unfortunately, not all values for c will make this happen. For example, if $c = 2$ and the table contains an even number of slots, then any key whose home position is in an even slot will have a probe sequence that cycles through only the even slots. Likewise, the probe sequence for a key whose home position is in an odd slot will cycle through the odd slots. Thus, this combination of table size and linear probing constant effectively divides the records into two sets stored in two disjoint sections of the hash table. So long as both sections of the table contain the same number of records, this is not really important. However, just from chance it is likely that one section will become fuller than the other, leading to more collisions and poorer performance for those records. The other section would have fewer records, and thus better performance. But the overall system performance will be degraded, as the additional cost to the side that is more full outweighs the improved performance of the less-full side.

Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (that is, c and M must share no factors). For a hash table of size $M = 10$, if c is any one of 1, 3, 7, or 9, then the probe sequence will visit all slots for any key. When $M = 11$, any value for c between 1 and 10 generates a probe sequence that visits all slots for every key.

12.8.3 PSEUDO-RANDOM PROBING

Consider the situation where $c = 2$ and we wish to insert a record with key k_1 such that $\mathbf{h}(k_1) = 3$. The probe sequence for k_1 is 3, 5, 7, 9, and so on. If another key k_2 has home position at slot 5, then its probe sequence will be 5, 7, 9, and so on. The probe sequences of k_1 and k_2 are linked together in a manner that contributes to clustering. In other words, linear probing with a value of $c > 1$ does not solve the problem of primary clustering. We would like to find a probe function that does not link keys together in this way. We would prefer that the

probe sequence for k_1 after the first step on the sequence should not be identical to the probe sequence of k_2. Instead, their probe sequences should diverge.

The ideal probe function would select the next position on the probe sequence at random from among the unvisited slots; that is, the probe sequence should be a random permutation of the hash table positions. Unfortunately, we cannot actually select the next position in the probe sequence at random, because we would not be able to duplicate this same probe sequence when searching for the key. However, we can do something similar called pseudo-random probing. In pseudo-random probing, the i th slot in the probe sequence is $(\mathbf{h}(K) + r_i) \mod M$ where r_i is the i th value in a random permutation of the numbers from 1 to $M-1$. All inserts and searches must use the same sequence of random numbers. The probe function would be $\mathbf{p}(K, i)$ = **Permutation**$[i]$ where **Permutation** is an array of length M that stores a value of 0 in position **Permutation**[0], and stores a random permutation of the values from 1 to $M-1$ in slots 1 to $M-1$.

12.8.4 QUADRATIC PROBING

Another probe function that eliminates primary clustering is called quadratic probing. Here the probe function is some quadratic function $\mathbf{p}(K, i) = c_1 i^2 + c_2 i + c_3$ for some choice of constants c_1, c_2, and c_3.

The simplest variation is $\mathbf{p}(K, i) = i^2$ (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$). Then the i th value in the probe sequence would be $(\mathbf{h}(K) + i^2) \mod M$.

There is one problem with quadratic probing: Its probe sequence typically will not visit all slots in the hash table.

For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, this means that the record cannot be inserted at all! A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $\mathbf{p}(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $\mathbf{p}(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

12.8.5 DOUBLE HASHING

Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the name given to the the situation when keys share substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value. This is because function $\mathbf{p}$ ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called secondary clustering.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, $\mathbf{h}_2$. Thus, the probe sequence would be of the form $\mathbf{p}(K, i) = i * \mathbf{h}_2(K)$. This method is called double hashing.

There are important restrictions on h_2. Most importantly, the value returned by h_2 must never be zero (or M) because that will immediately lead to an infinite loop as the probe sequence makes no progress. However, a good implementation of double hashing should also ensure that all of the probe sequence constants are relatively prime to the table size M. For example, if the hash table size were 100 and the step size for linear probing (as generated by function h_2) were 50, then there would be only one slot on the probe sequence. If instead the hash table size is 101 (a prime number), than any step size less than 101 will visit every slot in the table.

This can be achieved easily. One way is to select M to be a prime number, and have $\mathbf{h}_2$ return a value in the range $1 \leq \mathbf{h}_2(k) \leq M - 1$. We can do this by using this secondary hash function: $\mathbf{h}_2(k) = 1 + (k \mod (M - 1))$. An alternative is to set $M = 2^m$ for some value m and have $\mathbf{h}_2$ return an odd value between 1 and 2^m. We can get that result with this secondary hash function: $\mathbf{h}_2(k) = (((k/M) \mod (M/2)) * 2) + 1$.

Note: The secondary hash function $\mathbf{h}_2(k) = (((k/M) \mod (M/2)) * 2) + 1$ might seem rather mysterious, so let's break this down. This is being used in the context of two facts: (1) We want the function to return an odd value that is less than M the hash table size, and (2) we are using a hash table of size $M = 2^m$, which means that taking the mod of size M is using the bottom m bits of the key

value. OK, since $\mathbf{h}_2$ is multiplying something by 2 and adding 1, we guarentee that it is an odd number. Now, $((X \mod (M/2)) * 2) + 1$ must be in the range 1 and $M - 1$ (if you need to, play around with this on paper to convince yourself that this is true). This is exactly what we want. The last piece of the puzzle is the first part k/M. That is not strictly necessary. But remember that since the table size is $M = 2^m$, this is the same as shifting the key value right by m bits. In other words, we are not using the bottom m bits to decide on the second hash function value, which is especially a good thing if we used the bottom m bits to decide on the first hash function value! In other words, we really do not want the value of the step sized used by the linear probing to be fixed to the slot in the hash table that we chose. So we are using the next m bits of the key value instead. Note that this would only be a good idea if we have keys in a large enough key range, that is, we want plenty of use of those second m bits in the key range. This will be true if the max key value uses at least $2m$ bits, meaning that the max key value should be at least the square of the hash table size. This is not a problem for typical hashing applications.

12.9 Analysis of hash tables

How efficient is hashing? We can measure hashing performance in terms of the number of record accesses required when performing an operation. The primary operations of concern are insertion, deletion, and search. It is useful to distinguish between successful and unsuccessful searches. Before a record can be deleted, it must be found. Thus, the number of accesses required to delete a record is equivalent to the number required to successfully search for it. To insert a record, an empty slot along the record's probe sequence must be found. This is equivalent to an unsuccessful search for the record (recall that a successful search for the record during insertion should generate an error because two records with the same key are not allowed to be stored in the table).

12.9.1 ANALYSIS OF OPEN ADDRESSING

When the hash table is empty, the first record inserted will always find its home position free. Thus, it will require only one record access to find a free slot. If all records are stored in their home positions, then successful searches will also require only one record access. As the table begins to fill up, the probability that a record can be inserted into its home position decreases. If a record hashes to an occupied slot, then the collision resolution policy must locate another slot in which to store it. Finding records not stored in their home position also

requires additional record accesses as the record is searched for along its probe sequence. As the table fills up, more and more records are likely to be located ever further from their home positions.

From this discussion, we see that the expected cost of hashing is a function of how full the table is. Define the load factor for the table as $\alpha = N/M$, where N is the number of records currently in the table.

An estimate of the expected cost for an insertion (or an unsuccessful search) can be derived analytically as a function of α in the case where we assume that the probe sequence follows a random permutation of the slots in the hash table. Assuming that every slot in the table has equal probability of being the home slot for the next record, the probability of finding the home position occupied is α. The probability of finding both the home position occupied and the next slot on the probe sequence occupied is $(N(N-1))/(M(M-1))$. The probability of i collisions is $(N(N-1)...(N-i+1))/(M(M-1)...(M-i+1))$. If N and M are large, then this is approximately $(N/M)^i$. The expected number of probes is one plus the sum over $i \geq 1$ of the probability of i collisions, which is approximately

$$1 + \sum_{i=1}^{\infty} (N/M)^i \quad = \quad 1/(1-\alpha)$$

The cost for a successful search (or a deletion) has the same cost as originally inserting that record. However, the expected value for the insertion cost depends on the value of α not at the time of deletion, but rather at the time of the original insertion. We can derive an estimate of this cost (essentially an average over all the insertion costs) by integrating from 0 to the current value of α, yielding a result of $(1/\alpha)\log_e 1/(1-\alpha)$.

It is important to realise that these equations represent the expected cost for operations when using the unrealistic assumption that the probe sequence is based on a random permutation of the slots in the hash table. We thereby avoid all the expense that results from a less-than-perfect collision resolution policy. Thus, these costs are lower-bound estimates in the average case. The true average cost under linear probing is $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ for insertions or unsuccessful searches and $\frac{1}{2}(1 + 1/(1-\alpha))$ for deletions or successful searches.

Figure 12.1 shows how the expected number of record accesses grows as α grows. The horizontal axis is the value for α , the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for "random" probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two

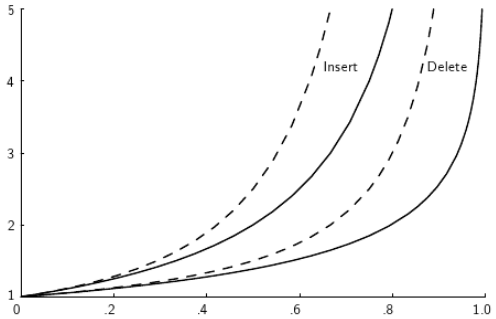


Figure 12.1 A plot showing the growth rate of the cost for insertion and deletion into a hash table as the load factor increases

leftmost lines show the cost for insertion (equivalently, unsuccessful search); the two rightmost lines show the cost for deletion (equivalently, successful search).

From the figure, you should see that the cost for hashing when the table is not too full is typically close to one record access. This is extraordinarily efficient, much better than binary search which requires $\log n$ record accesses. As α increases, so does the expected cost. For small values of α, the expected cost is low. It remains below two until the hash table is about half full. When the table is nearly empty, adding a new record to the table does not increase the cost of future search operations by much. However, the additional search cost caused by each additional insertion increases rapidly once the table becomes half full. Based on this analysis, the rule of thumb is to design a hashing system so that the hash table never gets above about half full, because beyond that point performance will degrade rapidly. This requires that the implementor have some idea of how many records are likely to be in the table at maximum loading, and select the table size accordingly. The goal should be to make the table small enough so that it does not waste a lot of space on the one hand, while making it big enough to keep performance good on the other.

12.10 Better hash functions

12.10.1 BINNING

Say we are given keys in the range 0 to 999, and have a hash table of size 10. In this case, a possible hash function might simply divide the key value by 100. Thus, all keys in the range 0 to 99 would hash to slot 0, keys 100 to 199 would

hash to slot 1, and so on. In other words, this hash function "bins" the first 100 keys to the first slot, the next 100 keys to the second slot, and so on.

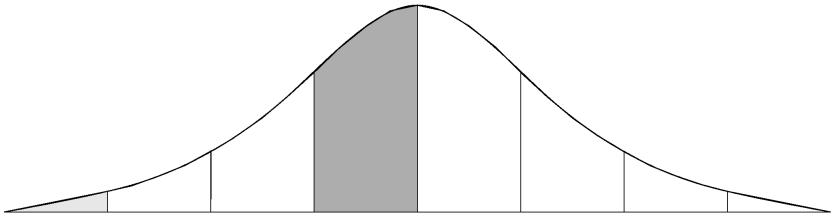
Binning in this way has the problem that it will cluster together keys if the distribution does not divide evenly on the high-order bits. In the above example, if more records have keys in the range 900-999 (first digit 9) than have keys in the range 100-199 (first digit 1), more records will hash to slot 9 than to slot 1. Likewise, if we pick too big a value for the key range and the actual key values are all relatively small, then most records will hash to slot 0. A similar, analogous problem arises if we were instead hashing strings based on the first letter in the string.

In general with binning we store the record with key value i at array position i/X for some value X (using integer division). A problem with Binning is that we have to know the key range so that we can figure out what value to use for X. Let's assume that the keys are all in the range 0 to 999. Then we want to divide key values by 100 so that the result is in the range 0 to 9. There is no particular limit on the key range that binning could handle, so long as we know the maximum possible value in advance so that we can figure out what to divide the key value by. Alternatively, we could also take the result of any binning computation and then mod by the table size to be safe. So if we have keys that are bigger than 999 when dividing by 100, we can still make sure that the result is in the range 0 to 9 with a mod by 10 step at the end.

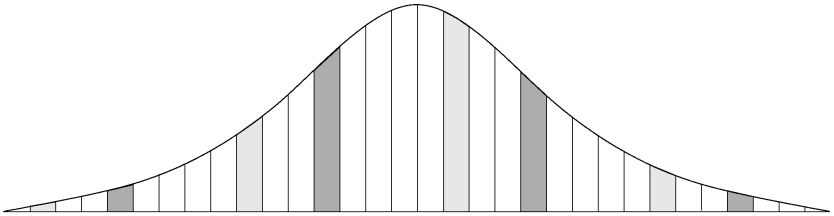
Binning looks at the opposite part of the key value from the mod function. The mod function, for a power of two, looks at the low-order bits, while binning looks at the high-order bits. Or if you want to think in base 10 instead of base 2, modding by 10 or 100 looks at the low-order digits, while binning into an array of size 10 or 100 looks at the high-order digits.

As another example, consider hashing a collection of keys whose values follow a normal distribution, as illustrated by Figure 12.2. Keys near the mean of the normal distribution are far more likely to occur than keys near the tails of the distribution. For a given slot, think of where the keys come from within the distribution. Binning would be taking thick slices out of the distribution and assign those slices to hash table slots. If we use a hash table of size 8, we would divide the key range into 8 equal-width slices and assign each slice to a slot in the table. Since a normal distribution is more likely to generate keys from the middle slice, the middle slot of the table is most likely to be used. In contrast, if we use the mod function, then we are assigning to any given slot in the table a series of thin slices in steps of 8. In the normal distribution, some of these slices

associated with any given slot are near the tails, and some are near the center. Thus, each table slot is equally likely (roughly) to get a key value.



(a) Binning



(b) Mod function

Figure 12.2 A comparison of binning vs. modulus as a hash function

12.10.2 THE MID-SQUARE METHOD

A good hash function to use with integer key values is the mid-square method. The mid-square method squares the key value, and then takes out the middle r bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most or all bits of the key value contribute to the result. For example, consider records whose keys are 4-digit numbers in base 10, as shown in Figure 12.3. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. Of course, if the key values all tend to be small numbers, then their squares will only affect the low-order digits of the hash value.

An example of the mid-square method. This image shows the traditional

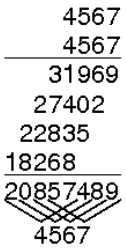


Figure 12.3 Mid-square method example

gradeschool long multiplication process. The value being squared is 4567. The result of squaring is 20857489. At the bottom, of the image, the value 4567 is show again, with each digit at the bottom of a "V". The associated "V" is showing the digits from the result that are being affected by each digit of the input. That is, "4" affects the output digits 2, 0, 8, 5, an 7. But it has no affect on the last 3 digits. The key point is that the middle two digits of the result (5 and 7) are affected by every digit of the input. :::

12.10.3 A SIMPLE HASH FUNCTION FOR STRINGS

Now we will examine some hash functions suitable for storing strings of characters. We start with a simple summation function.

```
function hashString(str):
    h = 0
    for each char in str:
        h = h + ord(char)
    return h
```

This function sums the ASCII values of the letters in a string. If the hash table size M is small compared to the resulting summations, then this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters in the string. This is an example of the folding method to designing a hash function.

As with many other hash functions, the final step is to apply the modulus operator to the result, using table size M to generate a value within the table range. If the sum is not sufficiently large, then the modulus operator will yield a poor distribution. For example, because the ASCII value for 'A' is 65 and 'Z' is 90, the sum will always be in the range 650 to 900 for a string of ten upper case letters. For a hash table of size 100 or less, a reasonable distribution results. For a hash table of size 1000, the distribution is terrible because only slots 650

to 900 can possibly be the home slot for some key value, and the values are not evenly distributed even within those slots.

Another problem is that the order of the characters in the string has no effect on the result. E.g., all permutations of the string "*ABCDEFG*" will result in the same hash value.

12.10.4 IMPROVED STRING FOLDING

If we instead multiply the hash with a prime number, before adding the next character, we get a much better distribution of the hash codes. This is Java's default hash code for strings, where the prime number is 31.

```
function hashStringImproved(str):
    h = 0
    for each char in str:
        h = 31 * h + ord(char)
    return h
```

Mathematically, the hash function is $s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-2} \cdot 31^1 + s_{n-1} \cdot 31^0$. This number grows quite fast when the string gets longer, but that's not a problem because Java will do an implicit modulo 2^{32} on each iteration.

For example, if the string "ABC" is passed to `hashStringImproved`, the resulting hash value will be $65 \cdot 31^2 + 66 \cdot 31 + 67 = 64,578$. If the table size is 101 then the modulus function will cause this key to hash to slot 39 in the table.

For any sufficiently long string, the sum will typically cause a 32-bit integer to overflow (thus losing some of the high-order bits) because the resulting values are so large. But this causes no problems when the goal is to compute a hash function.

12.11 Bucket hashing

One implementation for open addressing groups hash table slots into buckets. The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an overflow bucket of infinite capacity at the end of the table. All buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then

searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

12.11.1 AN ALTERNATIVE APPROACH TO BUCKET HASHING

A simple variation on bucket hashing is to hash a key value to some slot in the hash table as though bucketing were not being used. If the home position is full, then we search through the rest of the bucket to find an empty slot. If all slots in this bucket are full, then the record is assigned to the overflow bucket. The advantage of this approach is that initial collisions are reduced, because any slot can be a home position rather than just the first slot in the bucket.

Bucket methods are good for implementing hash tables stored on disk, because the bucket size can be set to the size of a disk block. Whenever search or insertion occurs, the entire bucket is read into memory. Because the entire bucket is then in memory, processing an insert or search operation requires only one disk access, unless the bucket is full. If the bucket is full, then the overflow bucket must be retrieved from disk as well. Naturally, overflow should be kept small to minimise unnecessary disk accesses.

12.12 Hash tables in practice

Congratulations! You have reached the end of the hashing chapter. In summary, a properly tuned hashing system will return records with an average cost of less than two record accesses. This makes it a very effective way known to store a database of records to support exact-match queries. Unfortunately, hashing is not effective when implementing *range queries*, or answering questions like "Which record in the collection has the smallest key value?"

In this section we will give some examples of problems with hashing, and how Java and Python implement hashing and hash tables internally.

12.12.1 ALGORITHMIC COMPLEXITY ATTACKS

As we wrote in the chapter introduction:

> "When properly implemented, these operations can be performed in

> constant time. (...) However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly."

This difficulty can be (and has been) exploited for malicious attacks on systems, so called algorithmic complexity attacks. We'll leave the word to Adam Jacobson and David Renardy to give an introduction to the problems (please read the whole text, it's an easy read):

> "An Algorithmic Complexity (AC) attack is a resource exhaustion attack that takes advantage of worst-case performance in server-side algorithms." (https://twosixtech.com/algorithmic-complexity-vulnerabilities-an-introduction)

As you can see, hash tables even have a category of itself ("Hashtable DoS Attacks"). They only mention separate chaining ("These hash tables utilised a linked list"), but the same problem arises for open addressing. (In fact, Python hash tables are open addressing – see below).

The problem is that it is extremely difficult to write good hash functions.

12.12.2 BREAKING HASH FUNCTIONS

Java's default algorithm for calculating a hash code from a string s looks like this (see the code for hashCode() in StringUTF16.java):

$$s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-2} \cdot 31^1 + s_{n-1} \cdot 31^0$$

This works well in practice, *if you assume that your data is normal*! But an attacker does not use normal data – instead they deliberately create data that will break the system. For example, suppose the attacker knows that you store user data in a Java HashMap with the username as key. Then they can create lots of artifical usernames to put in the database to make searching in the database slow. That's a hashtable DoS attack!

It's really easy to break Java's string hash function if you want to. It relies on the fact that the following two strings have the same hashcode:

```
"Aa".hashCode() == "BB".hashCode() == 2112
```

Therefore, all same-length repetitions of "Aa" and "BB" have the same hash code:

```
"AaAaAaAaAa", "AaAaAaAaBB", "AaAaAaBBAa", "AaAaAaBBBB", ...,
"BBBBBBAaAa", "BBBBBBAaBB", "BBBBBBBBAa", "BBBBBBBBBB"
```

There are 2^k possible strings with k repetitions of "Aa" resp. "BB", and all these strings have the same hash code! An attacker could insert all these strings into a

Java HashMap and then all of them would be in the same bucket and we would resort to linear search through a linked list of 2^k strings.

Note that this would not be improved by using an open addressing hash table, because then we would get a primary cluster with 2^k strings.

12.12.3 HOW ARE HASH TABLES IMPLEMENTED IN STANDARD LIBRARIES?

Java API Before version 1.2, Java had exactly the problem above, but in version 1.2 they changed the implementation. The current version of Java HashMap (and HashSet and Hashtable and similar) has the following characteristics:

- It's a separate chaining hash table.
- Small buckets (<8 elements) are linked lists, but larger buckets are red-black trees instead. This ensures $O(n \log n)$ worst time complexity, but it's linear if the data is well-behaved.
- The size of the hash table is a power of two ($n = 2^k$) – meaning that the resizing factor is 2.
- The maximum load factor for resizing is 0.75.

If you're interested you can read the comment on lines 125–232 in HashMap.java, where some implementation details are explained.

Python Python uses much more modern implementations than Java, of both hash functions and hash tables. Python hash functions uses a combination of the following techniques:

- Strings are hashed using SipHash (see PEP-456).
- Tuples are hashed using "a slightly simplified version of the xxHash non-cryptographic hash" (see lines 384–397 in tupleobject.c).
- In addition, Python adds randomisation: Whenever you start a new Python interpreter, it creates a random constant which is combined with the hashes. This means that every new instance will generate new hashes for the same strings, tuples, and other built-in objeccts.

Python dictionaries are implemented as hash tables. Since version 3.6 they are based on the following ideas (lecture video from PyCon 2017):

- Raymond Hettinger: Modern Python Dictionaries – A confluence of a dozen great ideas, PyCon 2017: https://www.youtube.com/watch?v=npw4s1QTmPg (the part about sharing several values in one table is only used for internal use in the Python interpreter)

- And a blog post explaining the new implementation: https://morepypy.blogspot.com/2015/01/faster-more-memory-efficient-and-more.html (it was first implemented in PyPy, but then they ported it to "standard" CPython too)

Here's a summary of the internal implementation:

- All hash tables have a power-of-two size ($n = 2^k$) – meaning that the resizing factor is 2.
- This means that the table index is the first (least significant) k bits of the hash value.
- It's an open adressing hash table.
- But it doesn't use linear probing, instead they probe with $+p$ in every step, where p depends on the higher bits in the hash value.
- It keeps the full hash value in the table (which improves the speed when resizing)
- To get better memory efficiency, it is split into (1) a size 2^k integer array with indices, and (2) a compact array with tuples of the form (hash,key,value). The indices in array(1) point to locations in array(2).
- This also has the effect that it can iterate over the insertion order of the elements.
- Deletion is done using tombstones.
- The maximum load factor for resizing is 0.66.
- See the source code here: https://github.com/python/cpython/blob/main/Objects/dictobject.c

12.13 Review questions

Section 12.13

Answer quiz online

This final section contains some review questions about the contents of this chapter.

Graphs

Graphs provide the ultimate in data structure flexibility. A graph consists of a set of nodes, and a set of edges where an edge connects two nodes. Trees and lists can be viewed as special cases of graphs.

Graphs are used to model both real-world systems and abstract problems, and are the data structure of choice in many applications. Here is a small sampling of the types of problems that graphs are routinely used for.

1. Modeling connectivity in computer and communications networks.
2. Representing an abstract map as a set of locations with distances between locations. This can be used to compute shortest routes between locations such as in a GPS routefinder.
3. Modeling flow capacities in transportation networks to find which links create the bottlenecks.
4. Finding a path from a starting condition to a goal condition. This is a common way to model problems in artificial intelligence applications and computerised game players.
5. Modeling computer algorithms, to show transitions from one program state to another.
6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
7. Modeling relationships such as family trees, business or military organisations, and scientific taxonomies.

The next section covers some basic graph terminology. The rest of the chapter will describe fundamental representations for graphs, provide a reference implementation, and cover core graph algorithms including traversal, topological sort, shortest paths algorithms, and algorithms to find the minimum spanning tree. Besides being useful and interesting in their own right, these algorithms illustrate the use of many other data structures presented throughout the course.

13.1 Definitions and properties

A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of vertices $\mathbf{V}$ and a set of edges $\mathbf{E}$, such that each edge in $\mathbf{E}$ is a connection between a pair of vertices in $\mathbf{V}$. The number of vertices is written $|\mathbf{V}|$, and the number of edges is written $|\mathbf{E}|$. $|\mathbf{E}|$ can range from zero to a maximum of $|\mathbf{V}|^2 - |\mathbf{V}|$.

Note: Some graph applications require that a given pair of vertices can have multiple or parallel edges connecting them, or that a vertex can have an edge to itself. However, the applications discussed here do not require either of these special cases. To simplify our graph API, we will assume that there are no duplicate edges.

A graph whose edges are not directed is called an undirected graph, as shown in part (a) of the following figure. A graph with edges directed from one vertex to another (as in (b)) is called a directed graph or digraph. A graph with labels associated with its vertices (as in (c)) is called a labeled graph. Associated with each edge may be a cost or weight. A graph whose edges have weights (as in (c)) is said to be a weighted graph. These are depicted in Figure 13.1.

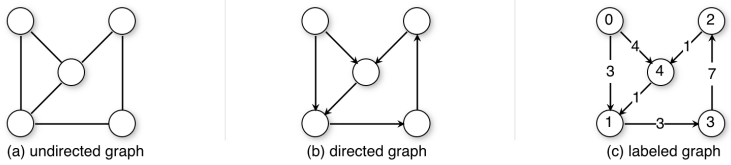


Figure 13.1 Some types of graphs

An edge connecting vertices a and b is written (a, b). Such an edge is said to be incident with vertices a and b. The two vertices are said to be adjacent. If the edge is directed from a to b, then we say that a is adjacent to b, and b is adjacent from a. The degree of a vertex is the number of edges it is incident with. For example, vertex e in Figure 13.2 has a degree of three.

In a directed graph, the out degree for a vertex is the number of neighbours adjacent from it (or the number of edges going out from it), while the in degree is the number of neighbours adjacent to it (or the number of edges coming in to it). In Figure 13.1 (c), the in degree of vertex 1 is two, and its out degree is one.

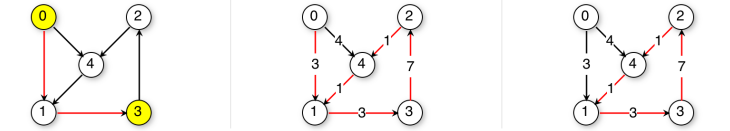
A sequence of vertices $v_1, v_2, ..., v_n$ forms a path of length $n - 1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$. A path is a simple path if all vertices on the path are distinct. The length of a path is the number of edges it contains. A cycle is a path of length three or more that connects some vertex v_1 to itself. A cycle

(a) Vertices *a* and *b* are *neighbors*

(b) The red edge is *incident* with vertices *a* and *b*

Figure 13.2 Neighbors and incidence

is a simple cycle if the path is simple, except for the first and last vertices being the same. These are depicted in Figure 13.3.



(a) A *simple* path from 0 to 3.

(b) Path 0, 1, 3, 2, 4, 1 is not simple

(c) *Simple cycle* 1, 3, 2, 4, 1

Figure 13.3 Different types of paths

An undirected graph is a connected graph if there is at least one path from any vertex to any other. The maximally connected subgraphs of an undirected graph are called connected components. For example, Figure 13.4 shows an undirected graph with three connected components.

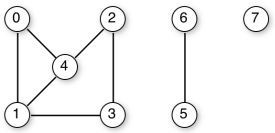
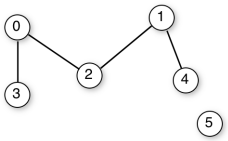


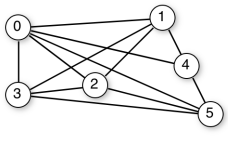
Figure 13.4 An undirected graph with three connected components

A graph with relatively few edges is called a sparse graph, while a graph with many edges is called a dense graph. A graph containing all possible edges is said to be a complete graph. A subgraph **S** is formed from graph **G** by selecting a subset $\mathbf{V}_s$ of **G**'s vertices and a subset $\mathbf{E}_s$ of **G** 's edges such that for every edge $e \in \mathbf{E}_s$, both vertices of e are in $\mathbf{V}_s$. Any subgraph of V where all vertices in the graph connect to all other vertices in the subgraph is called a clique. See Figure 13.5 for examples of these kinds of graphs.

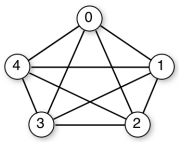
A graph without cycles is called an acyclic graph. Thus, a directed graph without cycles is called a directed acyclic graph or DAG. They are shown in Figure 13.6.



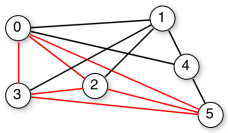
(a) A graph with relatively few edges is called a *sparse graph*.



(b) A graph with many edges is called a *dense graph*.

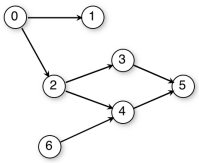


(c) A *complete graph* has edges connecting every pair of nodes.

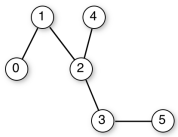


(d) A *clique* is a subset of V where all vertices in the subset have edges to all other vertices in the subset.

Figure 13.5 Sparse, dense and complete graphs



(a) Directed Acyclic Graph



(b) Acyclic Graph

Figure 13.6 Acyclic graph types

A free tree is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has $|\mathbf{V}| - 1$ edges.

13.2 Graph representations

There are two commonly used methods for representing graphs. The adjacency matrix for a graph is a $|\mathbf{V}| \times |\mathbf{V}|$ array. We typically label the vertices from v_0 through $v_{|\mathbf{V}|-1}$. Row i of the adjacency matrix contains entries for vertex v_i. Column j in row i is marked if there is an edge from v_i to v_j and is not marked otherwise. The space requirements for the adjacency matrix are $O(|\mathbf{V}|^2)$.

The second common representation for graphs is the adjacency list. The adjacency list is an array of linked lists. The array is $|\mathbf{V}|$ items long, with position i storing a pointer to the linked list of edges for vertex v_i. This linked list represents the edges by the vertices that are adjacent to vertex v_i.

Here is an example of the two representations on a directed graph. The entry for vertex 0 stores 1 and 4 because there are two edges in the graph leaving vertex

0, with one going to vertex 1 and one going to vertex 4. The list for vertex 2 stores an entry for vertex 4 because there is an edge from vertex 2 to vertex 4, but no entry for vertex 3 because this edge comes into vertex 2 rather than going out.

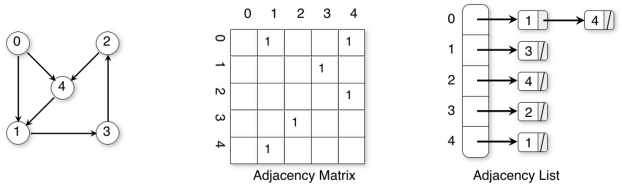


Figure 13.7 Representing a directed graph

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting vertices u and v is represented by two directed edges: one from u to v and one from v to u. Here is an example of the two representations on an undirected graph. We see that there are twice as many edge entries in both the adjacency matrix and the adjacency list. For example, for the undirected graph, the list for vertex 2 stores an entry for both vertex 3 and vertex 4.

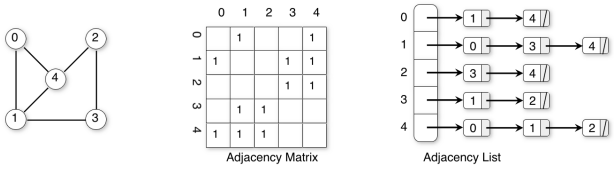


Figure 13.8 Representing an undirected graph

The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $O(|\mathbf{V}| + |\mathbf{E}|)$.

Sometimes we want to store weights or distances with each each edge, such as in Figure 13.1 (c). This is easy with the adjacency matrix, where we will just store values for the weights in the matrix. In Figure 13.7 and Figure 13.8 we store a value of "1" at each position just to show that the edge exists. That could have been done using a single bit, but since bit manipulation is typically complicated in most programming languages, an implementation might store a byte or an integer at each matrix position. For a weighted graph, we would need to store at

each position in the matrix enough space to represent the weight, which might typically be an integer.

The adjacency list needs to explicitly store a weight with each edge. In the adjacency list in Figure 13.9, each linked list node is shown storing two values. The first is the index for the neighbour at the end of the associated edge. The second is the value for the weight. As with the adjacency matrix, this value requires space to represent, typically an integer.

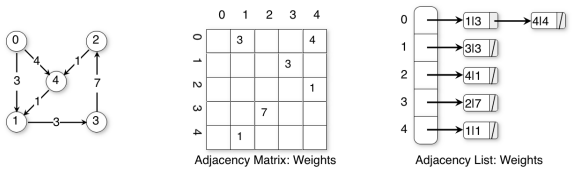


Figure 13.9 Representing a weighted graph

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

Example: Memory usage

Assume that a vertex index requires two bytes, a pointer requires four bytes, and an edge weight requires two bytes. Then, each link node in the adjacency list needs 2 + 2 + 4 = 8 bytes. The adjacency matrix for the directed graph above requires $2|\mathbf{V}^2| = 50$ bytes while the adjacency list requires $4|\mathbf{V}| + 8|\mathbf{E}| = 68$ bytes. For the undirected version of the graph above, the adjacency matrix requires the same space as before, while the adjacency list requires $4|\mathbf{V}| + 8|\mathbf{E}| = 116$ bytes (because there are now 12 edges represented instead of 6).

The adjacency matrix often requires a higher asymptotic cost for an algorithm than would result if the adjacency list were used. The reason is that it is common

for a graph algorithm to visit each neighbour of each vertex. Using the adjacency list, only the actual edges connecting a vertex to its neighbours are examined. However, the adjacency matrix must look at each of its $|\mathbf{V}|$ potential edges, yielding a total cost of $O(|\mathbf{V}^2|)$ time when the algorithm might otherwise require only $O(|\mathbf{V}| + |\mathbf{E}|)$ time. This is a considerable disadvantage when the graph is sparse, but not when the graph is closer to full.

13.3 Implementing graphs

We next turn to the problem of implementing a general-purpose graph class. There are two traditional approaches to representing graphs: The adjacency matrix and the adjacency list. In this section we will show actual implementations for each approach. We will begin with an interface defining an ADT for graphs that a given implementation must meet.

```
interface Graph of V:
    size: Int                          // The number of vertices in the graph.
    vertices() -> Collection of V      // Returns all vertices in the graph.
    outgoingEdges(v: V) -> Collection of Edge of V
                                       // Returns the edges that originates in vertex v.
```

Note that this API is quite generic, and perhaps not suited for all kinds of implementations. For example, the adjacency matrix implementation works best if the vertices are integers in the range $0 \ldots |\mathbf{V}| - 1$ where $|\mathbf{V}|$ is the number of vertices.

According to this interface, the size of the graph is the number of vertices, $|\mathbf{V}|$, and there is no method that returns the number of edges, $|\mathbf{E}|$. A practical implementation would have methods for both of these sizes, as well as methods for adding vertices and edges to the graph (and removing too).

Given an edge, we can use the attributes *start* and *end* to know the adjacent vertices, and *weight* to know its weight.

```
datatype Edge of V:
    start: V              // start vertex
    end: V                // end vertex
    weight: Float = 1.0   // weight, defaults to 1.0
```

Nearly every graph algorithm presented in this chapter will require visits to all neighbours of a given vertex. The `outgoingEdges` method returns a collection containing the edges that originate in the given vertex. To get the neighbours you can simply call `e.end` for each outgoing edge `e`. The following lines appear in many graph algorithms:

```
for each Edge e in G.outgoingEdges(v):
    w = e.end
    if w is not in visited:
        add w to visited
        ...do something with v, w, or e...
```

Here, `visited` is a set of vertices to keep track that we don't visit a vertex twice.

It is reasonably straightforward to implement our graph ADT using either the adjacency list or adjacency matrix. The sample implementations presented here do not address the issue of how the graph is actually created. The user of these implementations must add functionality for this purpose, perhaps reading the graph description from a file.

13.3.1 ADJACENCY MATRIX

Here is an implementation for the adjacency matrix. To simplify the implementation we assume that the vertices are integers $0 \ldots N-1$: then we can use the vertices as indices in the adjacency matrix.

```
datatype MatrixGraph implements Graph:
    // The edge matrix is an N x N matrix of weights.
    edgeMatrix: Array of (Array of Edges)
    size: Int

    constructor(vertexCount):
        edgeMatrix = new Array(size)
        for i = 0 .. size-1:
            edgeMatrix[i] = new Array(size)

    vertices():
        return the collection [0, 1, ..., size-1]

    outgoingEdges(v):
        outgoing = new List()
        for w in 0 .. size-1:
            weight = edgeMatrix[v][w]
            // We use the special weight o to indicate that there is no edge.
            if weight != 0:
                outgoing.append(new Edge(v, w, weight))
        return outgoing
```

The edge matrix is implemented as an integer array of size $n \times n$ for a graph of n vertices. Position (i, j) in the matrix stores the weight for edge (i, j) if it exists. A weight of zero for edge (i, j) is used to indicate that no edge connects vertices i and j.

This means that this simple implementation of an adjacency matrix does not work for all kinds of vertex types, but only for integer vertices. In addition, the vertices must be numbered $0 \ldots |\mathbf{V}| - 1$. The `vertices` method returns a collection of all vertices, which in this case is just the numbers $0 \ldots |\mathbf{V}| - 1$.

Given a vertex v, the `outgoingEdges` method scans through row `v` of the matrix to locate the positions of the various neighbours. It creates an edge for each neighbour and adds it to a list.

13.3.2 ADJACENCY LIST

Here is an implementation of the adjacency list representation for graphs. This implementation uses a generic type for vertices, so that you can use strings or anything else.

Its main data structure is a map from vertices to sets of edges. Exactly which kind of map or set we use can depend on our needs, but it can e.g. be any of the ones we have discussed earlier in the book.

One specific implementation that is particularly suited for an adjacency list separate chaining hash map, backed with a set implemented as a linked list. In that case, for each vertex we store a linked list of all the edges originating from that vertex. This makes the method `outgoingEdges` very efficient, because the only thing we have to do is to look up the given vertex in the internal map. To make the methods `vertexCount` and `vertices` efficient, we in addition store the vertices separately in the set `verticesSet`.

The implementations of the API methods are quite straightforward, as can be seen here:

```
class AdjacencyGraph implements Graph:
    edgesMap: Map from V to Edge = new Map()
    vertices: Set of V = new Set()
    size: Int = vertices.size

    vertices():
        return verticesSet

    outgoingEdges(v):
        return edgesMap.get(v)
```

13.4 Traversing graphs

Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph traversal and is

similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain might consist of a large collection of states, with connections between various pairs of states. Solving this sort of problem requires getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organised manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it might not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph might contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

Graph traversal algorithms can solve both of these problems by keeping track of the vertices that have been visited, in a set `visited`. At the beginning of the algorithm, this set is empty. When a vertex is first visited during the traversal, we add it to `visited`. If a vertex is encountered during traversal that already is in the set, we will not visit it a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking whether they are in the set `visited`. If not all vertices are in this set, we can continue the traversal from another unvisited vertex. Note that this process works regardless of whether the graph is directed or undirected. To ensure visiting all vertices, `graphTraverse` could be called as follows on a graph **G**:

```
function graphTraverse(G):
    visited = new Set()
    for each v in G.vertices():
        if v not in visited:
            doTraversal(G, v, visited)
```

The function `doTraversal` might be implemented by using one of the graph traversals described next.

13.4.1 DEPTH-FIRST SEARCH

Our first method for organised graph traversal is called depth-first search (DFS). Whenever a vertex v is visited during the search, DFS will recursively visit all of v 's unvisited neighbours. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a depth-first search tree. This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs.

The recursive DFS algorithm can be described as simply as this:

```
function visit(v):
    if v is unvisited:
        mark v as visited
        recursively visit all adjacent vertices
```

Here is a slightly more detailed implementation of the DFS algorithm.

```
function traverseDFS(G, v, visited):
    if v not in visited:
        visited.add(v)
        preVisit(G, v)
        for each edge in G.outgoingEdges(v):
            traverseDFS(G, edge.end, visited)
        postVisit(G, v)
```

This implementation contains calls to functions `preVisit` and `postVisit`. These functions specify what activity should take place during the search. Just as a preorder tree traversal requires action before the subtrees are visited, some graph traversals require that a vertex be processed before ones further along in the DFS. Alternatively, some applications require activity *after* the remaining vertices are processed; hence the call to function `postVisit`.

DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is $O(|\mathbf{V}| + |\mathbf{E}|)$.

13.4.2 BREADTH-FIRST SEARCH

Our second graph traversal algorithm is known as a breadth-first search (BFS). BFS examines all vertices connected to the start vertex before visiting vertices

further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

Here is an implementation for BFS. Note that it's not possible to call `postVisit` in BFS traversal, because you cannot know when the adjacent edges (the "children") have been traversed.

```
function traverseBFS(G, v, visited):
    agenda = new Queue()
    agenda.enqueue(v)
    while not agenda.isEmpty()
        v = agenda.dequeue()
        if v not in visited:
            visited.add(v)
            preVisit(G, v)
            for each edge in G.outgoingEdges(v):
                agenda.enqueue(edge.end)
            // postVisit is not possible in BFS search!
```

Fun fact: If you replace the queue with a stack (and the enqueing/dequeueing operations with push/pop), you will get depth-first search! This is because the recursive version of DFS implicitly uses the call stack to remember which vertices to visit.

13.5 Minimum spanning trees, MST

The minimum spanning tree (MST) problem takes as input a connected, undirected graph **G**, where each edge has a distance or weight measure attached. The MST is also called *minimal-cost spanning tree* (MCST).

The MST is the graph containing the vertices of **G** along with the subset of **G** 's edges that (1) has minimum total cost as measured by summing the values for all of the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

The MST contains no cycles. If a proposed MST did have a cycle, a cheaper MST could be had by removing any one of the edges in the cycle. Thus, the MST is a free tree with $|\mathbf{V}| - 1$ edges. The name "minimum-cost spanning tree" comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost. Figure 13.10 shows the MST for an example graph.

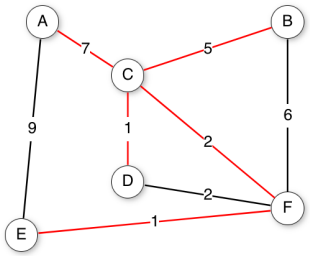


Figure 13.10 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

13.6 Prim's algorithm for finding the MST

The first of our two algorithms for finding MSTs is commonly referred to as Prim's algorithm. Prim's algorithm is very simple. Start with any vertex N in the graph, setting the MST to be N initially. Pick the least-cost edge connected to N. This edge connects N to another vertex; call this M. Add vertex M and edge (N, M) to the MST. Next, pick the least-cost edge coming from either N or M to any other vertex in the graph. Add this edge and the new vertex it reaches to the MST. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.

The following code shows an implementation for Prim's algorithm that searches the distance matrix for the next closest vertex.

```
function prim(graph, start):
    visited = new Set() of vertices
    parent = new Map() of vertices to vertices
    distances = new Map() of vertices to their distance from the MST
    // The distance from start to the MST is 0:
    distances.put(start, 0)

    repeat graph.size times:
        v = minVertex(graph, distances, visited)  // Find next-closest vertex
        visited.add(v)
        for each e in graph.outgoingEdges(v):
            if e.end not in distances or e.weight < distances.get(e.end):
                // If the edge makes the endpoint closer to the MST,
                // update the endpoint with the new distance, and add it to the MST
                distances.put(e.end, e.weight)
```

```
                parent.put(e.end, v)
    return parent
```

For each vertex v, when v is processed by Prim's algorithm, an edge going to v is added to the MST that we are building. The `parent` map stores the previously visited vertex that is closest to vertex v. This information lets us know which edge goes into the MST when vertex v is processed.

Finding the minimum vertex There are two reasonable solutions to the key issue of finding the unvisited vertex with minimum distance value during each pass through the main `for` loop. The first method is simply to scan through the list of $|\mathbf{V}|$ vertices searching for the minimum value, as follows:

```
function minVertex(graph, distances, visited):
    minV = null
    for each v in graph.vertices():
        if v not in visited:
            if minV is null or distances.get(v) < distances.get(minV):
                minV = v
    return minV
```

Because this scan is done $|\mathbf{V}|$ times, and because each edge requires a constant-time update to $\mathbf{D}$, the total cost for this approach is $O(|\mathbf{V}|^2 + |\mathbf{E}|) = O(|\mathbf{V}|^2)$, because $|\mathbf{E}|$ is in $O(|\mathbf{V}|^2)$.

13.6.1 PRIORITY QUEUE IMPLEMENTATION OF PRIM'S ALGORITHM

An alternative approach is to store unprocessed vertices in a minimum priority queue, such as a binary heap, ordered by their distance from the processed vertices. The next-closest vertex can be found in the heap in $O(\log|\mathbf{V}|)$ time. Every time we modify $\mathbf{D}(X)$, we could reorder X in the heap by deleting and reinserting it. This is an example of a priority queue with priority update. However, to implement true priority updating, we would need to store with each vertex its position within the heap so that we can remove its old distances whenever it is updated by processing new edges.

A simpler approach is to add the new (always smaller) distance value for a given vertex as a new record in the heap. The smallest value for a given vertex currently in the heap will be found first, and greater distance values found later will be ignored because the vertex will already be marked as **visited**. The only disadvantage to repeatedly inserting distance values in this way is that it will raise the number of elements in the heap from $O(|\mathbf{V}|)$ to $O(|\mathbf{E}|)$ in the worst case. But in practice this only adds a slight increase to the depth of the heap. The

time complexity is $O((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$, because for each edge that we process we must reorder the heap.

Here is the implementation for Dijkstra's algorithm using a priority queue.

```
function primPQ(graph, start):
    visited = new Set() of vertices
    parent = new Map() of vertices to vertices
    distances = new Map() of vertices to their distance from the MST
    agenda = new PriorityQueue() of vertices ordered by their distance from the MST
    // The distance from start to start is 0:
    distances.put(start, 0)
    agenda.add(start) with priority 0

    while not agenda.isEmpty():
        v = agenda.removeMin()
        if v not in visited:
            visited.add(v)
            for each e in graph.outgoingEdges():
                if e.end not in distances or dist < distances.get(e.end):
                    // If the edge makes the endpoint closer to the MST,
                    // update the endpoint with the new distance, add it to the MST and the agenda
                    distances.put(e.end, e.weight)
                    parent.put(e.end, v)
                    agenda.add(e.end) with priority e.weight
    return parent
```

13.6.2 CORRECTNESS OF PRIM'S ALGORITHM

Prim's algorithm is an example of a greedy algorithm. At each step in the `for` loop, we select the least-cost edge that connects some marked vertex to some unmarked vertex. The algorithm does not otherwise check that the MST really should include this least-cost edge. This leads to an important question: Does Prim's algorithm work correctly? Clearly it generates a spanning tree (because each pass through the `for` loop adds one edge and one unmarked vertex to the spanning tree until all vertices have been added), but does this tree have minimum cost?

Theorem: Prim's algorithm produces a minimum-cost spanning tree.

Proof: We will use a proof by contradiction. Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a graph for which Prim's algorithm does *not* generate an MST. Define an ordering on the vertices according to the order in which they were added by Prim's algorithm to the MST: $v_0, v_1, \ldots, v_{n-1}$. Let edge e_i connect (v_x, v_i) for some

$x < i$ and $i \leq 1$. Let e_j be the lowest numbered (first) edge added by Prim's algorithm such that the set of edges selected so far *cannot* be extended to form an MST for **G**. In other words, e_j is the first edge where Prim's algorithm "went wrong." Let **T** be the "true" MST. Call $v_p (p < j)$ the vertex connected by edge e_j, that is, $e_j = (v_p, v_j)$. Because **T** is a tree, there exists some path in **T** connecting v_p and v_j. There must be some edge e' in this path connecting vertices v_u and v_w, with $u < j$ and $w \geq j$. Because e_j is not part of **T**, adding edge e_j to **T** forms a cycle. Edge e' must be of lower cost than edge e_j, because Prim's algorithm did not generate an MST. This situation is illustrated in Figure 13.11. However, Prim's algorithm would have selected the least-cost edge available. It would have selected e', not e_j. Thus, it is a contradiction that Prim's algorithm would have selected the wrong edge, and thus, Prim's algorithm must be correct.

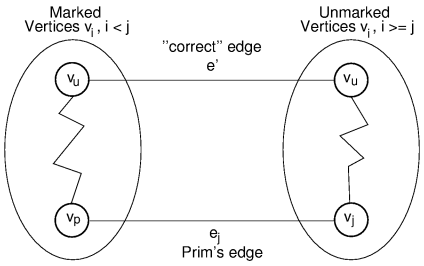


Figure 13.11 Proof of Prim's MST algorithm. The left oval contains that portion of the graph where Prim's MST and the "true" MST **T** agree. The right oval contains the rest of the graph. The two portions of the graph are connected by (at least) edges e_j (selected by Prim's algorithm to be in the MST) and e' (the "correct" edge to be placed in the MST). Note that the path from v_w to v_j cannot include any marked vertex $v_i, i \leq j$, because to do so would form a cycle.

13.7 Kruskal's algorithm for finding the MST

Our next MST algorithm is commonly referred to as Kruskal's algorithm. Kruskal's algorithm is also a simple, greedy algorithm. First partition the set of vertices into $|\mathbf{V}|$ disjoint sets, each consisting of one vertex. Then process the edges in order of weight. An edge is added to the MST, and two disjoint sets combined, if the edge connects two vertices in different disjoint sets. This process is repeated until only one disjoint set remains.

The edges can be processed in order of weight by putting them in an array and then sorting the array. Another possibility is to use a *minimum* priority queue, similar to what we did in Prim's algorithm in the previous section.

The only tricky part to this algorithm is determining if two vertices belong to the same equivalence class. Fortunately, the ideal algorithm is available for the purpose – the Union/Find algorithm, described in Section 11.5. Here is an implementation for Kruskal's algorithm. Note that since the MST will never have more than $|\mathbf{V}|-1$ edges, we can return as soon as the MST contains enough edges.

```
function kruskal(graph):
    edges = all edges in graph
    sort edges by their weight
    mst = new Set() of edges
    forest = new ParentPointerTree(graph.size)
    while not edges.isEmpty():
        e = edges.removeMin()
        if forest.find(e.start) != forest.find(e.end):
            mst.add(edge)      // If the vertices are not connected, add the edge to the MST
            if mst.size >= graph.size-1:
                return mst     // Return when the MST has |V|-1 edges
            forest.union(e.start, e.end)  // Connect the two vertices
```

Kruskal's algorithm is dominated by the time required to process the edges. The **Find** and **Union** functions are nearly constant in time if path compression and weighted union is used. Thus, the total cost of the algorithm is $O(|\mathbf{E}|\log|\mathbf{E}|)$ in the worst case, when nearly all edges must be processed before all the edges of the spanning tree are found and the algorithm can stop. More often the edges of the spanning tree are the shorter ones, and only about $|\mathbf{V}|$ edges must be processed. If so, the cost is often close to $O(|\mathbf{V}|\log|\mathbf{E}|)$ in the average case (provided we use a priority queue instead of sorting all edges in advance).

13.8 Shortest-paths problems

If you have an *unweighted* graph, the shortest path between two vertices is the smallest number of edges you have to pass to get from one of the vertices to the other.

If you agument the breadth-first search algorithm from Section 13.4.2 to remember which vertex a visited vertex came from, if will give you the shortest path between the start vertex and any other vertex. However, things become sligthly more complicated if the graph is weighted.

Shortest-paths on weighted graphs On a road map, a road connecting two towns is typically labeled with its distance. We can model a road network as a directed graph whose edges are labeled with real numbers. These numbers represent the distance (or other cost metric, such as travel time) between two vertices. These labels may be called weights, costs, or distances, depending on the application. Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices. This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices.

For example, in Figure 13.12, the cost of the path from A to B to D is 15. The cost of the edge directly from A to D is 20. The cost of the path from A to C to B to D is 10. Thus, the shortest path from A to D is 10 (rather than along the edge connecting A to D). We use the notation $\mathbf{d}(A, D) = 10$ to indicate that the shortest distance from A to D is 10. In the figure, there is no path from E to B, so we set $\mathbf{d}(E, B) = \infty$. We define $\mathbf{w}(A, D) = 20$ to be the weight of edge (A, D), that is, the weight of the direct connection from A to D. Because there is no edge from E to B, $\mathbf{w}(E, B) = \infty$. Note that $\mathbf{w}(D, A) = \infty$ because the graph is directed. We assume that all weights are positive.

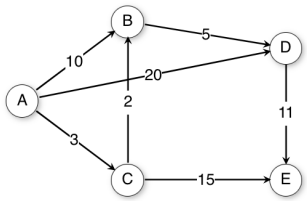


Figure 13.12 Example graph for shortest-path definitions

13.9 Dijkstra's shortest-path algorithm

We will now present an algorithm to solve the single-source shortest paths problem. Given vertex S in Graph $\mathbf{G}$, find a shortest path from S to every other vertex in $\mathbf{G}$. We might want only the shortest path between two vertices, S and T. However in the worst case, finding the shortest path from S to T requires us to find the shortest paths from S to every other vertex as well. So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices. The algorithm described here will only

compute the distance to every such vertex, rather than recording the actual path. Recording the path requires only simple modifications to the algorithm.

As mentioned in the previous section, the shortest path between two vertices can be found using a simple breadth-first search, for *unweighted* graphs (or whenever all edges have the same cost). However, when weights are added, BFS will not give the correct answer.

One approach to solving this problem when the edges have differing weights might be to process the vertices in a fixed order. Label the vertices v_0 to v_{n-1}, with $S = v_0$. When processing vertex v_1, we take the edge connecting v_0 and v_1. When processing v_2, we consider the shortest distance from v_0 to v_2 and compare that to the shortest distance from v_0 to v_1 to v_2. When processing vertex v_i, we consider the shortest path for vertices v_0 through v_{i-1} that have already been processed. Unfortunately, the true shortest path to v_i might go through vertex v_j for $j > i$. Such a path will not be considered by this algorithm. However, the problem would not occur if we process the vertices in order of distance from S. Assume that we have processed in order of distance from S to the first $i - 1$ vertices that are closest to S; call this set of vertices $\mathbf{S}$. We are now about to process the i th closest vertex; call it X.

A shortest path from S to X must have its next-to-last vertex in S. Thus,

$$\mathbf{d}(S, X) = \min_{U \in \mathbf{S}} (\mathbf{d}(S, U) + \mathbf{w}(U, X))$$

In other words, the shortest path from S to X is the minimum over all paths that go from S to U, then have an edge from U to X, where U is some vertex in $\mathbf{S}$.

This solution is usually referred to as Dijkstra's algorithm. It works by maintaining a distance estimate $\mathbf{D}(X)$ for all vertices X in $\mathbf{V}$. The elements of $\mathbf{D}$ are initialised to the value ∞ (positive infinity). Vertices are processed in order of distance from S. Whenever a vertex v is processed, $\mathbf{D}(X)$ is updated for every neighbour X of V.

Dijkstra's algorithm is quite similar to Prim's algorithm for finding the minimum spanning tree (Section 13.6). The primary difference is that we are seeking, not the next vertex which is closest to the MST, but rather the the next vertex which is closest to the start vertex. Thus the following lines in Prim's algorithm:

```
if e.weight < distances.get(e.end):
    distances.put(e.end, e.weight)
```

are replaced with the following lines in Dijkstra's algorithm:

```
dist = distances.get(v) + e.weight
if dist < distances.get(e.end):
    distances.put(e.end, dist)
```

Here is an implementation for Dijkstra's algorithm. At the end, `distances` will contain the shortest distance from the start to each reachable vertex.

```
function dijkstra(graph, start):
    visited = new Set() of vertices
    distances = new Map() of vertices to their distance from start
    distances.put(start, 0)   // The distance from start to start is 0

    repeat graph.size times:
        v = minVertex(graph, distances, visited)   // Find next-closest vertex
        visited.add(v)
        for each e in graph.outgoingEdges(v):
            w = e.end
            dist = distances.get(v) + e.weight
            if e.end not in distances or dist < distances.get(e.end):
                // If the new distance to the endpoint is shorter...
                distances.put(e.end, dist)   // ...update the endpoint with the new distance
    return distances
```

Finding the minimum vertex Just as for Prim's algorithm, there are two ways of finding the next-closest vertex. We can scan through all vertices searching for the minimum value (note that this code is exactly the same as for Prim's algorithm):

```
function minVertex(graph, distances, visited):
    minV = null
    for each v in graph.vertices():
        if v not in visited:
            if minV is null or distances.get(v) < distances.get(minV):
                minV = v
    return minV
```

And just as for Prim's algorithm, this will give us a complexity which is quadratic in the number of vertices, $O(|\mathbf{V}|^2)$.

13.9.1 USING A PRIORITY QUEUE

An alternative approach is to store unprocessed vertices in a minimum priority queue, such as a binary heap, ordered by their distance from the processed vertices. The next-closest vertex can be found in the heap in $O(\log|\mathbf{V}|)$ time. Every time we modify $\mathbf{D}(X)$, we could reorder X in the heap by deleting and reinserting it. This is an example of a priority queue with priority update. However, to implement true priority updating, we would need to store with each vertex

its position within the heap so that we can remove its old distances whenever it is updated by processing new edges.

A simpler approach is to add the new (always smaller) distance value for a given vertex as a new record in the heap. The smallest value for a given vertex currently in the heap will be found first, and greater distance values found later will be ignored because the vertex will already be marked as **visited**. The only disadvantage to repeatedly inserting distance values in this way is that it will raise the number of elements in the heap from $O(|\mathbf{V}|)$ to $O(|\mathbf{E}|)$ in the worst case. But in practice this only adds a slight increase to the depth of the heap. The time complexity is $O((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$, because for each edge that we process we must reorder the heap.

Here is the implementation for Dijkstra's algorithm using a priority queue.

```
function dijkstraPQ(graph, start):
    visited = new Set() of vertices
    distances = new Map() of vertices to their distance from start
    agenda = new PriorityQueue() of vertices ordered by their distance from start
    // The distance from start to start is 0:
    distances.put(start, 0)
    agenda.add(start) with priority 0

    while not agenda.isEmpty():
        v = agenda.removeMin()
        if v not in visited:
            visited.add(v)
            for each e in graph.outgoingEdges():
                dist = distances.get(v) + e.weight
                if w not in distances or dist < distances.get(e.end):
                    // If the new distance to the endpoint is shorter,
                    // update the endpoint with the new distance, and add it to the agenda
                    distances.put(e.end, dist)
                    agenda.add(e.end) with priority dist
    return distances
```

Which version is the best? Using `minVertex` to scan the vertex list for the minimum value is more efficient when the graph is dense, that is, when $|\mathbf{E}|$ approaches $|\mathbf{V}|^2$. Using a priority is more efficient when the graph is sparse because its cost is $O((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$. When the graph is dense, this cost can become as great as $O(|\mathbf{V}|^2 \log |\mathbf{E}|) = O(|\mathbf{V}|^2 \log |\mathbf{V}|)$.

In practice, most graphs are sparse so unless you know that the graph is dense you should use the priority queue version.

13.10 Specialised algorithms on weighted graphs

In this section we present two specialised algorithms on weighted directed graphs:

Floyd's algorithm This algorithm finds the shortest paths between *all pairs of vertices* in a graph.

Topological sort This algorithm sorts the vertices of a *directed acyclic graph* (DAG), so that no edges point "backwards". This is crucial to solve various scheduling problems.

Section 13.10

Read the rest online

13.11 Review questions

This final section contains some review questions about the contents of this chapter.

Section 13.11

Answer quiz online