UNIVERSITY OF CAMBRIDGE

Department of Computer Science and Technology

# Algorithms

**Professor Frank Stajano**

**Computer Science Tripos — Part IA**

**Academic year 2018–2019**

**Lent term 2019**

http://www.cl.cam.ac.uk/teaching/1819/Algorithms/

frank.stajano--algs@cst.cam.ac.uk

**Revised 2019 edition**

Revision 22 of 2019-01-03 15:55:17 +0000 (Thu, 03 Jan 2019).

# Contents

# Preliminaries

## Course content and textbooks

This course is about some of the coolest stuff a programmer can do.

Most real-world programming is conceptually pretty simple. The undeniable difficulties come primarily from size: enormous systems with millions of lines of code and complex APIs that won't all comfortably fit in a single brain. But each piece usually does something pretty bland, such as moving data from one place to another and slightly massaging it along the way.

Here, it's different. We look at pretty advanced hacks—those ten-line chunks of code that make you want to take your hat off and bow.

The only way to understand this material in a deep, non-superficial way is to program and debug it yourself, and then run your programs step by step on your own examples, visualizing intermediate results along the way. You might think you are fluent in $n$ programming languages but you aren't really a programmer until you've written and debugged some hairy pointer-based code such as that required to cut and splice the circular doubly-linked lists used in Fibonacci trees. (Once you do, you'll know why.)

However this course isn't about programming: it's about designing and analysing algorithms and data structures—the ones that great programmers then write up as tight code and put in libraries for other programmers to reuse. It's about finding smart ways of solving difficult problems, and about evaluating different solutions to see which one results in the best performance.

In order to gain a more than superficial understanding of the course material you will also need a full-length textbook, for which this handout is *not* a substitute. The one I recommend is a classic, adopted at many of the top universities around the world, and which at some point was *the* most cited book in computer science:

> [CLRS3] Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Third edition.* MIT press, 2009. ISBN 978-0-

262-53305-8.

A heavyweight book at about 1300 pages, it covers a little more material and at slightly greater depth than most others. It includes careful mathematical treatment of the algorithms it discusses and is a natural candidate for a reference shelf. Despite its bulk and precision this book is written in a fairly friendly style. I know some of you have already read most of it before even coming to Cambridge. You can't properly call yourself a computer scientist if CLRS3 is not on your bookshelf. It's the default text for this course: by all means feel free to refer to other books too, but chapter references in the chapter headings of these notes are to this textbook.

Other textbooks on algorithms that may well be worth consulting as optional additional references include at least Sedgewick; Kleinberg and Tardos; and of course the legendary multi-volume Knuth. Full bibliographic details are in the syllabus and on the course web page. However none of these other textbooks covers *all* the topics in the syllabus, so you're still better off getting yourself a copy of CLRS3 (which by the way, in spite of its excellent quality, is also the cheapest of the bunch).

Remember that your undergraduate education costs in the region of at least 10 k£ per year and is the foundation of your future professional career. You've been admitted to Cambridge, so you're already pretty good. Once you've made this investment, go through with it properly. Attempting to save 35 £ by learning the fundamentals of computer science without a good textbook is really a false economy and will put you at a disadvantage compared to your more dedicated contemporaries who'll become superstars and get Firsts. I don't get a percentage from book sales—I say this purely in your interest.

Note also that a growing number of mostly accurate descriptions of algorithms and data structures can be found on Wikipedia. Once you master the material in this course, and especially after you've earned some experience by writing and debugging your own implementation, consider improving any such descriptions that you find lacking or unclear.

## What is in these notes

After you finish university and go into the real world, you won't always be given the slides, so it's a good idea to acquire the useful life skill of creating your own notes when you listen to a presentation. Some of my students do so with a beautiful fountain pen; others have taught themselves to write LaTeX in real time; yet others may be content with just scribbling in the margin—which I have made especially big to allow

that.

Whatever you do, I encourage you to recreate the content of the lectures yourself and to study the wonderful textbook. The more you redo by yourself, the more you'll actually learn. This handout is not in the form of slides but it does cover the same topics as the lectures, and in the same order, while sometimes providing more detailed and precise commentaries than one could convey in a bullet point format.

These notes contain short exercises, highlighted by boxes, that you would do well to solve as you go along to prove that you are not just reading on autopilot. They tend to be easy (most are meant to take not more than five minutes each) and are therefore insufficient to help you really *own* the material covered here. For that, program the algorithms yourself[1] and solve problems found in your textbook or assigned by your supervisor. There is a copious supply of past exam questions at `http://www.cl.cam.ac.uk/teaching/exams/pastpapers/` under Algorithms, Algorithms I, Algorithms II and Data Structures and Algorithms, and in the example sheets available from the course webpage I have curated a subset of questions that are still relevant for this year's syllabus.

## Acknowledgements and history

I produced my first version of these notes in 2005, for a 16-lecture course[2] entitled "Data Structures and Algorithms", building on the excellent notes for that course originally written by Arthur Norman and then enhanced by Roger Needham (my academic grandfather—the PhD supervisor of my PhD supervisor) and Martin Richards. I hereby express my gratitude to my illustrious predecessors. In later years the course evolved into two (Algorithms I and II, to first and second year students respectively) and the aggregate number of my lectures gradually expanded from 4+12 to 0+27, until the 2013–2014 reorganization that brought all this material back into the first year, for a 24-lecture course of which the second half is now covered by another lecturer (first Thomas Sauerwald, until 2016, and then Damon Wischik). A number of interesting topics have been dropped during the various reorganizations, from string searching to Van Emde Boas trees, and maybe one day I'll collect them all into an extended

[1] The more programs you write to recreate what I show you in the lectures, the more you will really own this material.

[2] 16 lectures including 4 on what effectively was "remedial discrete mathematics" for Diploma students, thus in fact only 12 lectures of data structures and algorithms proper.

version. But, for now, this version of the handout only has topics that are in this year's syllabus.

Although I don't know where they are, from experience I am pretty sure that these notes still contain a few bugs, as all non-trivial documents do. Consult the course web page for the errata corrige. I am grateful to Kay Henning Brodersen, Sam Staton, Simon Spacey, Rasmus King, Chloë Brown, Robert Harle, Larry Paulson, Daniel Bates, Tom Sparrow, Marton Farkas, Wing Yung Chan, Tom Taylor, Trong Nhan Dao, Oliver Allbless, Aneesh Shukla, Christian Richardt, Long Nguyen, Michael Williamson, Myra VanInwegen, Manfredas Zabarauskas, Ben Thorner, Simon Iremonger, Heidi Howard, Tom Sparrow, Simon Blessenohl, Nick Chambers, Nicholas Ngorok, Miklós András Danka, Hauke Neitzel, Alex Bate, Darren Foong, Jannis Bulian, Gábor Szarka, Suraj Patel, Diandian Wang, Simone Teufel and particularly Alastair Beresford and Jan Polášek for sending me corrections to previous editions. Whether you are a student or a supervisor, if you find any more corrections and email them to me, I'll credit you in any future revisions (unless you prefer anonymity). The responsibility for any remaining mistakes remains of course mine.

# Chapter 1

# What's the point of all this?

**Textbook**

Study chapter 1 in CLRS3.

## 1.1 What is an algorithm?

An **algorithm** is a systematic recipe for solving a problem. By "systematic" we mean that the problem being solved will have to be specified quite precisely and that, before any algorithm can be considered complete, it will have to be provided with a proof that it works and an analysis of its performance. In a great many cases, all of the ingenuity and complication in algorithms is aimed at making them fast (or at reducing the amount of memory that they use) so a justification that the intended performance will be attained is very important.

In this course you will learn, among other things, a variety of "prior art" algorithms and data structures to address recurring computer science problems; but what would be especially valuable to you is acquiring the skill to invent new algorithms and data structures to solve difficult problems you weren't taught about. The best way to make progress towards that goal is to participate in this course actively, rather than to follow it passively. To help you with that, here are three difficult problems for which you should try to come up with suitable algorithms and data structures. The rest of the course will eventually teach you good ways to solve these problems but you will have a much greater understanding of the answers and of their applicability if you attempt (and perhaps fail) to solve these problems on your own before you are taught the canonical solution.

## 1.2 DNA sequences

In bioinformatics, a recurring activity is to find out how "similar" two given DNA sequences are. For the purposes of this simplified problem definition, assume that a DNA sequence is a string of arbitrary length over the alphabet {`A`, `C`, `G`, `T`} and that the degree of similarity between two such sequences is measured by the length of their longest common subsequence, as defined next. A subsequence $T$ of a sequence $S$ is any string obtained by dropping zero or more characters from $S$; for example, if $S =$ `AGTGTACCCAT`, then the following are valid subsequences: `AGGTAAT` (=`AG`$\not$`T``GTA`$\not$`C`$\not$`C`$\not$`C``AT`), `TACAT` (=$\not$`A`$\not$`G`$\not$`T`$\not$`G``TA`$\not$`C`$\not$`C`$\not$`A`$\not$`T`), `GGT` (=$\not$`A`$\not$`G`$\not$`T`$\not$`G``T`$\not$`A`$\not$`C`$\not$`C`$\not$`C`$\not$`A`$\not$`T`); but the following are not: `AAG`, `TCG`. You must find an algorithm that, given two sequences $X$ and $Y$ of arbitrary but finite lenghts, returns a sequence $Z$ of maximal length that is a subsequence of both $X$ and $Y$[1].

You might wish to try your candidate algorithm on the following two sequences: $X =$ `CCGTCAGTCGCG`, $Y =$ `TGTTTCGGAATGCAA`. What is the longest subsequence you obtain? Are there any others of that length? Are you sure that there exists no longer common subsequence (in other words: can you prove your algorithm is correct)? Is your algorithm simple enough that you can run it with pencil and paper in a reasonable time on an input of this size? How long do you estimate your algorithm would take to complete, on your computer, if the sequences were about 30 characters each? Or 100? Or a million?

## 1.3 Bestseller chart

Imagine an online store with millions of items in its catalogue. For each item, the store keeps track of how many instances it sold. Every day the new sales figures come in and a web page is compiled with a list of the top 100 best sellers. How would you generate such a list? How long would it take to run this computation? How long would it take if, hypothetically, the store had *trillions* of different items for sale instead of merely millions? Of course you could re-sort the whole catalogue each time and take the top 100 items, but can you do better? And is it cheaper to maintain the chart up to date after each sale or to recompute it from scratch once a day? *(You note here that we are not merely concerned with finding an algorithm, but also with how to estimate the relative performance of different alternatives, before actually running them.)*

[1] There may be several, all of maximal length.

## 1.4 Database indexing

Imagine a very large database of transactions (e.g. microbilling for a telecomms operator or bids history for an online auction site), with several indices over different keys, so that you can sequentially walk through the database records in order of account number but also, alternatively, by transaction date or by value or by surname. Each index has one entry per record (containing the key and the disk address of the record) but there are so many records that even the indices (never mind the records) are too large to fit in RAM[2] and must themselves be stored as files on disk. What is an efficient way of retrieving a particular record given its key, if we consider scanning the whole index linearly as too slow? Can we arrange the data in some other way that would speed up this operation? And, once you have thought of a specific solution: how would you keep your new indexing data structure up to date when adding or deleting records to the original database?

## 1.5 Questions to ask

I recommend you spend some time attacking the three problems above, as seriously as if they were exam questions, before going any further with the course. You may not fully succeed yet, perhaps depending on what you studied in high school, but you must give each of them your best shot. Then, after each new lecture, ask yourself whether what you learnt that day gives any insight towards a better solution. The first and most obvious question (and the one often requiring the greatest creativity) is of course:

- What strategy to use? What is the algorithm? What is the data structure?

But there are several other questions that are important too.

- Is the algorithm correct? How can we prove that it is?
- How long does it take to run? How long would it take to run on a much larger input? Besides, since computers get faster and cheaper all the time, how long would it take to run on a different type of

[2] This is becoming less and less common, given the enormous size of today's memories, but trust computer people to keep inventing new ways of filling them up! Alternatively, change the context and imagine that the whole thing must run inside your watch.

computer, or on the computer I will be able to buy in a year, or in three years, or in ten? Can you roughly estimate what input sizes it would not be able to process even if run on the computing cluster of a large corporation? Or on that of a three-letter agency?

- If there are several possible algorithms, all correct, how can we compare them and decide which is best? If we rank them by speed on a certain computer and a certain input, will this ranking carry over to other computers and other inputs? And what other ranking criteria should we consider, if any, other than speed?

Your overall goal for this course is to learn general methods for answering all of these questions, regardless of the specific problem.

# Chapter 2

# Sorting

**Chapter contents**

Review of complexity and O-notation. Trivial sorting algorithms of quadratic complexity. Review of merge sort and quicksort, understanding their memory behaviour on statically allocated arrays. Minimum cost of sorting. Heapsort. Stability. Other sorting methods including sorting in linear time. Median and order statistics.
Expected coverage: about 4 lectures.
Study 1, 2, 3, 6, 7, 8, 9 in CLRS3.

Our look at algorithms starts with sorting, which is a big topic: any course on algorithms, including *Foundations of Computer Science* that precedes this one, is bound to discuss a number of sorting methods. Volume 3 of Knuth (almost 800 pages) is entirely dedicated to sorting (covering over two dozen algorithms) and the closely related subject of searching, so don't think this is a small or simple topic! However much is said in this lecture course, there is a great deal more that is known.

Some lectures in this chapter will cover algorithms (such as insertion sort, merge sort and quicksort) to which you have been exposed before from a functional language (ML) perspective. While these notes attempt to be self-contained, the lecturer may go more quickly through the material you have already seen. During this second pass you should pay special attention to issues of memory allocation and array usage which were not evident in the functional programming presentation.

## 2.1 Insertion sort

**Textbook**

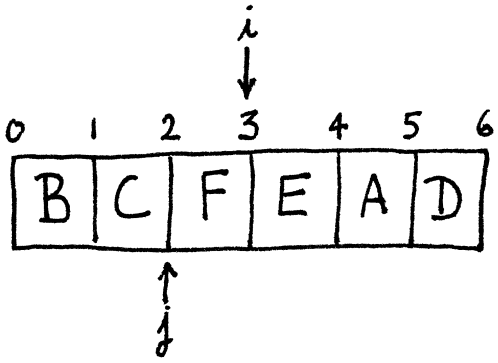
Study chapter 2 in CLRS3.

Let us approach the problem of sorting a sequence of items by modelling what humans spontaneously do when arranging in their hand the cards they were dealt in a card game: you keep the cards in your hand in order and you insert each new card in its place as it comes.

We shall look at data types in greater detail later on in the course but let's assume you already have a practical understanding of the "array" concept: a sequence of adjacent "cells" in memory, indexed by an integer. If we implement the hand as an array `a[]` of adequate size, we might put the first card we receive in cell `a[0]`, the next in cell `a[1]` and so on. Note that one thing we cannot actually do with the array, even though it is natural with lists or when handling physical cards, is to insert a new card between `a[0]` and `a[1]`: if we need to do that, we must first shift right all the cells after `a[0]`, to create an unused space in `a[1]`, and then write the new card there.

Let's assume we have been dealt a hand of $n$ cards[1], now loaded in the array as `a[0]`, `a[1]`, `a[2]`, ..., `a[`$n$`-1]`, and that we want to sort it. We pretend that all the cards are still face down on the table and that we are picking them up one by one in order. Before picking up each card, we first ensure that all the preceding cards in our hand have been sorted.

[1] Each card is represented by a capital letter in the diagram so as to avoid confusion between card numbers and index numbers. Letters have the obvious order implied by their position in the alphabet and thus A < B < C < D..., which is of course also true of their ASCII or Unicode code.




A little aside prompted by the diagram. A common source of bugs in computing is the off-by-one error. Try implementing insertsort on your own and you might accidentally trip over an off-by-one error yourself, before producing a debugged version. You are hereby encouraged to acquire a few good hacker habits that will reduce the chance of your succumbing to that particular pitfall. One is to number items from zero rather than from one—then most "offset + displacement" calculations will just work. Another is to adopt the convention used in this diagram when referring to arrays, strings, bitmaps and other structures with lots of linearly numbered cells: the index always refers to the position *between* two cells and gives its name to the cell to its right. (The Python programming language, for example, among its many virtues, uses this convention consistently.) Then the difference between two indices gives you the number of cells between them. So for example the subarray `a[2:5]`, containing the elements from `a[2]` included to `a[5]` excluded, has $5 - 2 = 3$ elements, namely `a[2] = "F"`, `a[3] = "E"` and `a[4] = "A"`.

When we pick up card `a[`$i$`]`, since the first $i$ items of the array have been sorted, the next is inserted in place by letting it sink towards the left down to its rightful place: it is compared against the item at position $j$ (with $j$ starting at $i - 1$, the rightmost of the already-sorted elements) and, if smaller than it, a swap moves it down. If the new element does move down, then so does the $j$ pointer; and then the new element is again compared against the one in position $j$ and swapped if necessary, until it gets to its place. We can write down this algorithm in pseudocode as follows:

```
0  def insertSort(a):
1      """BEHAVIOUR: Run the insertsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for i from 1 included to len(a) excluded:
10         # ASSERT: the first i positions are already sorted.
11
12         # Insert a[i] where it belongs within a[0:i].
13         j = i - 1
14         while j >= 0 and a[j] > a[j + 1]:
15             swap(a[j], a[j + 1])
16             j = j - 1
```

Pseudocode is an informal notation that is pretty similar to real source code but which omits any irrelevant details. For example we write `swap(x,y)` instead of the sequence of three assignments that would normally be required in many languages. The exact syntax is not terribly important: what matters more is clarity, brevity and conveying the essential ideas and features of the algorithm. It should be trivial to convert a piece of well-written pseudocode into the programming language of your choice.

**Exercise 1**
Assume that each `swap(x, y)` means three assignments (namely `tmp = x; x = y; y = tmp`). Improve the insertsort algorithm pseudocode shown in the handout to reduce the number of assignments performed in the inner loop.

## 2.2 Is the algorithm correct?

How can we convince ourselves (and our customers) that the algorithm is correct? In general this is far from easy. An essential first step is to specify the objectives as clearly as possible: to paraphrase Virgil Gligor, who once said something similar about attacker modelling, without a specification the algorithm can never be correct or incorrect—only surprising!

In the pseudocode above, we provided a (slightly informal) specification in the documentation string for the routine (lines 1–7). The *precondition* (line 4) is a request, specifying what the routine expects to receive from its caller; while the *postcondition* (lines 6–7) is a promise, specifying what the routine will do for its caller (provided that the precondition is satisfied on call). The pre- and post-condition together form a kind of "contract", using the terminology of Bertrand Meyer, between the routine and its caller[2]. This is a good way to provide a specification.

There is no universal method for proving the correctness of an algorithm; however, a strategy that has very broad applicability is to reduce a large problem to a suitable sequence of smaller subproblems to which you can apply mathematical induction[3]. Are we able to do so in this case?

To reason about the correctness of an algorithm, a very useful technique is to place key *assertions* at appropriate points in the program. An assertion is a statement that, whenever that point in the program is reached, a certain property will always be true. Assertions provide "stepping stones" for your correctness proof; they also help the human reader understand what is going on and, by the same token, help the programmer debug the implementation[4]. Coming up with good invariants is not always easy but is a great help for developing a convincing proof (or indeed for discovering bugs in your algorithm while it isn't correct yet). It is especially helpful to find a good, meaningful invariant at the beginning of each significant loop. In the algorithm above we have an invariant on line 10, at the beginning of the main loop: the $i^{th}$ time we enter the loop, it says, the previous passes of the loop will have sorted the leftmost $i$ cells of the array. How? We don't care, but our job now is to prove the inductive step: assuming the assertion is true when we

[2]If the caller fails to uphold her part of the bargain and invokes the routine while the precondition is not true, the routine cannot be blamed if it doesn't return the correct result. On the other hand, if the caller ensures that the precondition is true before calling the routine, the routine will be considered faulty unless it returns the correct result. The "contract" is as if the routine said to the caller: "provided you satisfy the precondition, I shall satisfy the postcondition".

[3]Mathematical induction in a nutshell: "How do I solve the case with $k$ elements? I don't know, but assuming someone smarter than me solved the case with $k-1$ elements, I could tell you how to solve it for $k$ elements starting from *that*"; then, if you also independently solve a starting point, e.g. the case of $k = 0$, you've essentially completed the job.

[4]Many modern programming languages allow you to write assertions as program statements (as opposed to comments); then the expression being asserted is evaluated at runtime and, if it is not true, an exception is raised; this alerts you as early as possible that something isn't working as expected, as opposed to allowing the program to continue running while in a state inconsistent with your beliefs and assumptions about it.

enter the loop, we must prove that *one* further pass down the loop will make the assertion true when we reenter. Having done that, and having verified that the assertion holds for the trivial case of the first iteration ($i = 1$; it obviously does, since the first *one* positions cannot possibly be out of order), then all that remains is to check that we achieve the desired result (whole array is sorted) at the end of the last iteration.

Check the recommended textbook for further details and a much more detailed walkthrough, but this is the jist of a powerful and widely applicable method for proving the correctness of your algorithm.

**Exercise 2**
Provide a useful invariant for the inner loop of insertion sort, in the form of an assertion to be inserted between the "while" line and the "swap" line.

## 2.3 Computational complexity

**Textbook**

Study chapter 3 in CLRS3.

### 2.3.1 Abstract modelling and growth rates

How can we estimate the time that the algorithm will take to run if we don't know how big (or how jumbled up) the input is? It is almost always necessary to make a few simplifying assumptions before performing cost estimation. For algorithms, the ones most commonly used are:

1. We only worry about the worst possible amount of time that some activity could take.

2. Rather than measuring absolute computing times, we only look at *rates of growth* and we ignore constant multipliers. If the problem size is $n$, then $100000f(n)$ and $0.000001f(n)$ will both be considered equivalent to just $f(n)$.

3. Any finite number of exceptions to a cost estimate are unimportant so long as the estimate is valid for all large enough values of $n$.

4. We do not restrict ourselves to just reasonable values of $n$ or apply any other reality checks. Cost estimation will be carried through as an abstract mathematical activity.

Despite the severity of all these limitations, cost estimation for algorithms has proved very useful: almost always, the indications it gives relate closely to the practical behaviour people observe when they write and run programs.

The notations big-O, $\Theta$ and $\Omega$, discussed next, are used as short-hand for some of the above cautions.

### 2.3.2 Big-O, $\Theta$ and $\Omega$ notations

A function $f(n)$ is said to be $O(g(n))$ if there exist constants $k$ and $N$, all $> 0$, such that $0 \leq f(n) \leq k \cdot g(n)$ whenever $n > N$. In other words, $g(n)$ provides an upper bound that, for sufficiently large values of $n$, $f(n)$ will never exceed[5], except for what can be compensated by a constant factor. In informal terms: $f(n) \in O(g(n))$ means that $f(n)$ grows *at most* like $g(n)$, but no faster.

A function $f(n)$ is said to be $\Theta(g(n))$ if there are constants $k_1$, $k_2$ and $N$, all $> 0$, such that $0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ whenever $n > N$. In other words, for sufficiently large values of $n$, the functions $f()$ and $g()$ agree within a constant factor. This constraint is much stronger than the one implied by Big-O. In informal terms: $f(n) \in \Theta(g(n))$ means that $f(n)$ grows *exactly* at the same rate as $g(n)$.

Some authors also use $\Omega()$ as the dual of $O()$ to provide a lower bound. In informal terms: $f(n) \in \Omega(g(n))$ means that $f(n)$ grows *at least* like $g(n)$.

Some authors also use lowercase versions of $O()$ and $\Omega()$ to make a subtle point. The "big" versions describe asymptotic bounds that might or might not be tight; informally, $O()$ is like $\leq$ and $\Omega()$ is like $\geq$. The "small" versions, instead, describe asymptotic bounds that are definitely not tight: informally, $o()$ is like $<$ and $\omega()$ is like $>$.

Here is a very informal[6] summary table:

[5] We add the "greater than zero" constraint to avoid confusing cases of a $f(n)$ with a high growth rate dominated by a $g(n)$ with a low growth rate because of sign issues, e.g. $f(n) = -n^3$ which is $< g(n) = n$ for any $n > 0$.

[6] For sufficiently large $n$, within a constant factor and blah blah blah.

| | If... | then $f(n)$ grows ... $g(n)$. | $f(n)$ ... $g(n)$ |
|---|---|---|---|
| small-o | $f(n) \in o(g(n))$ | strictly more slowly than | $<$ |
| big-o | $f(n) \in O(g(n))$ | at most as quickly as | $\leq$ |
| big-theta | $f(n) \in \Theta(g(n))$ | exactly like | $=$ |
| big-omega | $f(n) \in \Omega(g(n))$ | at least as quickly | $\geq$ |
| small-omega | $f(n) \in \omega(g(n))$ | strictly more quickly than | $>$ |

Note that none of these notations says anything about $f(n)$ being a computing time estimate, even though that will be the most common use in this lecture course.

Note also that it is common to say that $f(n) = O(g(n))$, with "$=$" instead of "$\in$". This is formally incorrect[7] but it's a broadly accepted custom, so we shall sloppily adopt it too from time to time.

Various important computer procedures have costs that grow as $O(n \log(n))$ and a gut-feeling understanding of logarithms will be useful to follow this course. Formalities apart, the most fundamental thing to understand about logarithms is that $\log_b(n)$ is the number of digits of $n$ when you write $n$ down in base $b$. If this isn't intuitive, then any fancy algebra you may be able to perform on logarithms will be practically useless.

In the proofs, the logarithms will often come out as ones to base 2—which, following Knuth, we shall indicate as "lg": for example, $\lg(1024) = \log_2(1024) = 10$. But observe that $\log_2(n) = \Theta(\log_{10}(n))$ (indeed a stronger statement could be made—the ratio between them is a constant); so, with Big-O or $\Theta$ or $\Omega$ notation, there is no real need to worry about the base of logarithms—all versions are equally valid.

The following exercise contains a few examples that may help explain, even if (heavens forbid) you don't actually do the exercise.

[7] For example it violates the transitivity of equality: we may have $f_1(n) = O(\lg n)$ and $f_2(n) = O(\lg n)$ even though $f_1(n) \neq f_2(n)$.

**Exercise 3**

$$
\begin{aligned}
|sin(n)| &= O(1) \\
|sin(n)| &\neq \Theta(1) \\
200 + sin(n) &= \Theta(1) \\
123456n + 654321 &= \Theta(n) \\
2n - 7 &= O(17n^2) \\
\lg(n) &= O(n) \\
\lg(n) &\neq \Theta(n) \\
n^{100} &= O(2^n) \\
1 + 100/n &= \Theta(1)
\end{aligned}
$$

For each of the above "=" lines, identify the constants $k, k_1, k_2, N$ as appropriate. For each of the "$\neq$" lines, show they can't possibly exist.

Please note the distinction between the value of a function and the amount of time it may take to compute it: for example $n!$ can be computed in $O(n)$ arithmetic operations, but has value bigger than $O(n^k)$ for any fixed $k$.

### 2.3.3 Models of memory

Through most of this course there will be a tacit assumption that the computers used to run algorithms will always have enough memory, and that this memory can be arranged in a single address space so that one can have unambiguous memory addresses or pointers. Put another way, we pretend you can set up a single array of integers that is as large as you will ever need.

There are of course practical ways in which this idealization may fall down. Some archaic hardware designs may impose quite small limits on the size of any one array, and even current machines tend to have but finite amounts of memory, and thus upper bounds on the size of data structure that can be handled.

A more subtle issue is that a truly unlimited memory will need integers (or pointers) of unlimited size to address it. If integer arithmetic on a computer works in a 32-bit representation (as is still common for embedded systems) then the largest integer value that can be represented is certainly less than $2^{32}$ and so one can not sensibly talk about arrays

with more elements than that. This limit represents only 4 gigabytes of main memory, which used to be considered enormous but these days is the amount installed by default in the kind of basic computer you can pick up in a supermarket next to your groceries. The solution is obviously that the width of integer subscripts used for address calculation has to increase with the logarithm of the size of a memory large enough to accommodate the problem. So, to solve a hypothetical problem that needed an array of size $10^{100}$, all subscript arithmetic would have to be done using 100 decimal digit precision.

It is normal in the analysis of algorithms to ignore these problems and assume that any element `a[i]` of an array can be accessed in unit time, however large the array is. The associated assumption is that integer arithmetic operations needed to compute array subscripts can also all be done at unit cost. This makes good practical sense since the assumption holds pretty well true for all problems—or at least for most of those you are actually likely to *want* to tackle on a computer[8].

Strictly speaking, though, on-chip caches in modern processors make the last paragraph incorrect. In the good old days, all memory references used to take unit time. Now, since processors have become faster at a much higher rate than memory[9], CPUs use super fast (and expensive and comparatively small) cache stores that can typically serve up a memory value in one or two CPU clock ticks; however, when a cache miss occurs, it often takes tens or even hundreds of ticks. Locality of reference is thus becoming an issue, although one which most textbooks on algorithms still largely ignore for the sake of simplicity of analysis.

### 2.3.4 Models of arithmetic

The normal model for computer arithmetic used here will be that each arithmetic operation[10] takes unit time, irrespective of the values of the numbers being combined and regardless of whether fixed or floating point numbers are involved. The nice way that $\Theta$ notation can swallow up constant factors in timing estimates generally justifies this. Again there is a theoretical problem that can safely be ignored in almost all cases: in the specification of an algorithm (or of an Abstract Data Type) there may be some integers, and in the idealized case this will imply that the procedures described apply to arbitrarily large integers, including ones with values that will be many orders of magnitude larger than native

[8] With the notable exception of cryptography.

[9] This phenomenon is referred to as "the memory gap".

[10] Not merely the ones on array subscripts mentioned in the previous section.

computer arithmetic will support directly[11]. In the fairly rare cases where this might arise, cost analysis will need to make explicit provision for the extra work involved in doing multiple-precision arithmetic, and then timing estimates will generally depend not only on the number of values involved in a problem but on the number of digits (or bits) needed to specify each value.

### 2.3.5 Worst, average and amortized costs

Usually the simplest way of analyzing an algorithm is to find the worst-case performance. It may help to imagine that somebody else is proposing the algorithm, and you have been challenged to find the very nastiest data that can be fed to it to make it perform really badly. In doing so you are quite entitled to invent data that looks very unusual or odd, provided it comes within the stated range of applicability of the algorithm. For many algorithms the "worst case" is approached often enough that this form of analysis is useful for realists as well as pessimists.

Average case analysis ought by rights to be of more interest to most people, even though worst case costs may be really important to the designers of systems that have real-time constraints, especially if there are safety implications in failure. But, before useful average cost analysis can be performed, one needs a model for the probabilities of all possible inputs. If in some particular application the distribution of inputs is significantly skewed, then analysis based on uniform probabilities might not be valid. For worst case analysis it is only necessary to study one limiting case; for average analysis the time taken for every case of an algorithm must be accounted for—and this, usually, makes the mathematics a lot harder.

Amortized analysis (covered in the second half of this course) is applicable in cases where a data structure supports a number of operations and these will be performed in sequence. Quite often the cost of any particular operation will depend on the history of what has been done before; and, sometimes, a plausible overall design makes most operations cheap at the cost of occasional expensive internal reorganization of the data. Amortized analysis treats the cost of this re-organization as the joint responsibility of all the operations previously performed on the data structure and provides a firm basis for determining if it was worthwhile. It is typically more technically demanding than just single-operation worst-case analysis.

[11] Or indeed, in theory, larger than the whole main memory can even hold! After all, the entire RAM of your computer might be seen as just one very long binary integer.

A good example of where amortized analysis is helpful is garbage collection, where it allows the cost of a single large expensive storage reorganization to be attributed to each of the elementary allocation transactions that made it necessary. Note that (even more than is the case for average cost analysis) amortized analysis is not appropriate for use where real-time constraints apply.

## 2.4 How much does insertion sort cost?

Having understood the general framework of asymptotic worst-case analysis and the simplifications of the models we are going to adopt, what can we say about the cost of running the insertion sort algorithm we previously recalled? If we indicate as $n$ the size of the input array to be sorted, and as $f(n)$ the very precise (but very difficult to accurately represent in closed form) function giving the time taken by our algorithm to compute an answer on the worst possible input of size $n$, on a specific computer, then our task is not to find an expression for $f(n)$ but merely to identify a much simpler function $g(n)$ that works as an upper bound, i.e. a $g(n)$ such that $f(n) = O(g(n))$. Of course a loose upper bound is not as useful as a tight one: if $f(n) = O(n^2)$, then $f(n)$ is also $O(n^5)$, but the latter doesn't tell us as much.

Once we have a reasonably tight upper bound, the fact that the big-O notation eats away constant factors allows us to ignore the differences between the various computers on which we might run the program.

If we go back to the pseudocode listing of insertsort found on page 16, we see that the outer loop of line 9 is executed exactly $n - 1$ times (regardless of the values of the elements in the input array), while the inner loop of line 14 is executed a number of times that depends on the number of swaps to be performed: if the new card we pick up is greater than any of the previously received ones, then we just leave it at the rightmost end and the inner loop is never executed; while if it is smaller than any of the previous ones it must travel all the way through, forcing as many executions as the number of cards received until then, namely $i$. So, in the worst case, during the $i^{th}$ invocation of the outer loop, the inner loop will be performed $i$ times. In total, therefore, for the whole algorithm, the inner loop (whose body consists of a constant number of elementary instructions) is executed a number of times that won't exceed the $n^{th}$ triangular number, $\frac{n(n+1)}{2}$. In big-O notation we ignore constants and lower-order terms, so we can simply write $O(n^2)$.

Note that it is possible to implement the algorithm slightly more efficiently at the price of complicating the code a little bit, as suggested in another exercise on page 16.

**Exercise 4**
What is the asymptotic complexity of the variant of insertsort that does fewer swaps?

## 2.5 Minimum cost of sorting

We just established that insertion sort has a worst-case asymptotic cost dominated by the square of the size of the input array to be sorted (we say in short: "insertion sort has quadratic cost"). Is there any possibility of achieving better asymptotic performance with some other algorithm?

If I have $n$ items in an array, and I need to rearrange them in ascending order, whatever the algorithm there are two elementary operations that I can plausibly expect to use repeatedly in the process. The first (comparison) takes two items and compares them to see which should come first[12]. The second (exchange) swaps the contents of two nominated array locations.

In extreme cases either comparisons or exchanges[13] may be hugely expensive, leading to the need to design methods that optimize one regardless of other costs. It is useful to have a limit on how good a sorting method could possibly be, measured in terms of these two operations.

**Assertion 1 (lower bound on exchanges).** If there are $n$ items in an array, then $\Theta(n)$ exchanges always suffice to put the items in order. In the worst case, $\Theta(n)$ exchanges are actually needed.

**Proof.** Identify the smallest item present: if it is not already in the right place, one exchange moves it to the start of the array. A second exchange moves the next smallest item to place, and so on. After at worst $n-1$ exchanges, the items are all in order. The bound is $n-1$ rather than $n$ because at the very last stage the biggest item has to be in its right place without need for a swap—but that level of detail is unimportant to $\Theta$ notation.

[12] Indeed, to start with, this course will concentrate on sorting algorithms where the *only* information about where items should end up will be that deduced by making pairwise comparisons.

[13] Often, if exchanges are costly, it can be useful to sort a vector of pointers to objects rather than a vector of the objects themselves—exchanges in the pointer array will be cheap.

**Exercise 5**
The proof of Assertion 1 (lower bound on exchanges) convinces us that $\Theta(n)$ exchanges are always *sufficient*. But why isn't that argument good enough to prove that they are also *necessary*?

Conversely, consider the case where the original arrangement of the data is such that the item that will need to end up at position $i$ is stored at position $i + 1$ (with the natural wrap-around at the end of the array). Since every item is in the wrong position, you must perform enough exchanges to touch each position in the array and that certainly means at least $n/2$ exchanges, which is good enough to establish the $\Theta(n)$ growth rate. Tighter analysis would show that more than $n/2$ exchanges are in fact needed in the worst case.

**Assertion 2 (lower bound on comparisons).** Sorting by pairwise comparison, assuming that all possible arrangements of the data might actually occur as input, necessarily costs at least $\Omega(n \lg n)$ comparisons.

**Proof.** As you saw in *Foundations of Computer Science*, there are $n!$ permutations of $n$ items and, in sorting, we in effect identify one of these. To discriminate between that many cases we need at least $\lceil \log_2(n!) \rceil$ binary tests. Stirling's formula tells us that $n!$ is roughly $n^n$, and hence that $\lg(n!)$ is about $n \lg n$.

Note that this analysis is applicable to any sorting method whose only knowledge about the input comes from performing pairwise comparisons between individual items[14]; that it provides a lower bound on costs but does not guarantee that it can be attained; and that it is talking about worst case costs. Concerning the last point, the analysis can be carried over to average costs when all possible input orders are equally probable.

For those who can't remember Stirling's name or his formula, the following argument is sufficient to prove that $\lg(n!) = \Theta(n \lg n)$.

$$\lg(n!) = \lg(n(n-1)(n-2)\ldots 2\cdot 1) = \lg n + \lg(n-1) + \lg(n-2) + \ldots + \lg(2) + \lg(1)$$

All $n$ terms on the right are less than or equal to $\lg n$ and so

$$\lg(n!) \leq n \lg n.$$

Therefore $\lg(n!)$ is bounded by $n \lg n$. Conversely, since the lg function is monotonic, the first $n/2$ terms, from $\lg n$ to $\lg(n/2)$, are all greater than or equal to $\lg(n/2) = \lg n - \lg 2 = (\lg n) - 1$, so

[14] Hence the existence of sorting methods faster than $n \lg n$ when we know more, a priori, about the items to be sorted—as we shall see in section 2.14.

$$\lg(n!) \quad \geq \quad \frac{n}{2}(\lg n - 1) + \lg(n/2) + \ldots + \lg(1) \quad \geq \quad \frac{n}{2}(\lg n - 1),$$

proving that, when $n$ is large enough, $n \lg n$ is bounded by $k \lg(n!)$ (for $k = 3$, say). Thus $\lg(n!) = \Theta(n \lg n)$.

## 2.6 Selection sort

In the previous section we proved that an array of $n$ items may be sorted by performing no more than $n - 1$ exchanges. This provides the basis for one of the simplest sorting algorithms known: selection sort. At each step it finds the smallest item in the remaining part of the array and swaps it to its correct position. This has, as a sub-algorithm, the problem of identifying the smallest item in an array. The sub-problem is easily solved by scanning linearly through the (sub)array, comparing each successive item with the smallest one found so far. If there are $m$ items to scan, then finding the minimum clearly costs $m-1$ comparisons. The whole selection-sort process does this on a sequence of sub-arrays of size $n, n-1, \ldots, 1$. Calculating the total number of comparisons involved requires summing an arithmetic progression, again yielding a triangular number and a total cost of $\Theta(n^2)$. This very simple method has the advantage (in terms of how easy it is to analyse) that the number of comparisons performed does not depend at all on the initial organization of the data, unlike what happened with insert-sort.

```
0  def selectSort(a):
1      """BEHAVIOUR: Run the selectsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for k from 0 included to len(a) excluded:
10         # ASSERT: the array positions before a[k] are already sorted
11
12         # Find the smallest element in a[k:END] and swap it into a[k]
13         iMin = k
14         for j from iMin + 1 included to len(a) excluded:
15             if a[j] < a[iMin]:
16                 iMin = j
17         swap(a[k], a[iMin])
```

We show this and the other quadratic sorting algorithms in this section not as models to adopt but as examples of the kind of wheel one is likely to reinvent before having studied better ways of doing it. Use them to learn to compare the trade-offs and analyze the performance on simple algorithms where understanding what's happening is not the most difficult issue, as well as to appreciate that coming up with asymptotically better algorithms requires a lot more thought than that.

**Exercise 6**
When looking for the minimum of $m$ items, every time one of the $m-1$ comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the Selection sort algorithm, maximizes the number of times that the above-mentioned comparison fails.

## 2.7 Binary insertion sort

Now suppose that data movement is cheap (e.g. we use pointers, as per footnote 13 on page 25), but comparisons are expensive (e.g. it's string comparison rather than integer comparison). Suppose that, part way through the sorting process, the first $k$ items in our array are neatly in ascending order, and now it is time to consider item $k+1$. A binary search in the initial part of the array can identify where the new item should go, and this search can be done in $\lceil \lg(k) \rceil$ comparisons. Then we can drop the item in place using at most $k$ exchanges. The complete sorting process performs this process for $k$ from 1 to $n$, and hence the total number of comparisons performed will be

$$\lceil \lg(1) \rceil + \lceil \lg(2) \rceil + \ldots + \lceil \lg(n-1) \rceil$$

which is bounded by

$$\lg(1) + 1 + \lg(2) + 1 + \ldots + lg(n-1) + 1$$

and thus by $\lg((n-1)!) + n = O(\lg(n!)) = O(n \lg n)$. This effectively attains the lower bound for general sorting that we set up earlier, in terms of the number of comparisons. But remember that the algorithm has high (quadratic) data movement costs. Even if a swap were a million times cheaper than a comparison (say), so long as both elementary operations can be bounded by a constant cost then the overall asymptotic cost of this algorithm will be $O(n^2)$.

```
0  def binaryInsertSort(a):
1      """BEHAVIOUR: Run the binary insertion sort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for k from 1 included to len(a) excluded:
10         # ASSERT: the array positions before a[k] are already sorted
11
12         # Use binary partitioning of a[0:k] to figure out where to insert
13         # element a[k] within the sorted region;
14
15         ### details left to the reader ###
16
17         # ASSERT: the place of a[k] is i, i.e. between a[i-1] and a[i]
18
19         # Put a[k] in position i. Unless it was already there, this
20         # means right-shifting by one every other item in a[i:k].
21         if i != k:
22             tmp = a[k]
23             for j from k - 1 included down to i - 1 excluded:
24                 a[j + 1] = a[j]
25             a[i] = tmp
```

**Exercise 7**
Code up the details of the binary partitioning portion of the binary insertion sort algorithm.

## 2.8 Bubble sort

Another simple sorting method, similar to Insertion sort and very easy to implement, is known as Bubble sort. It consists of repeated passes through the array during which adjacent elements are compared and, if out of order, swapped. The algorithm terminates as soon as a full pass requires no swaps.

```
0  def bubbleSort(a):
1      """BEHAVIOUR: Run the bubble sort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      repeat:
10         # Go through all the elements once, swapping any that are out of order
11         didSomeSwapsInThisPass = False
12         for k from 0 included to len(a) - 1 excluded:
13             if a[k] > a[k + 1]:
14                 swap(a[k], a[k + 1])
15                 didSomeSwapsInThisPass = True
16     until didSomeSwapsInThisPass == False
```

Bubble sort is so called because, during successive passes, "light" (i.e. low-valued) elements bubble up towards the "top" (i.e. the cell with the lowest index, or the left end) of the array. Like Insertion sort, this algorithm has quadratic costs in the worst case but it terminates in linear time on input that was already sorted. This is clearly an advantage over Selection sort.

**Exercise 8**
Prove that Bubble sort will never have to perform more than $n$ passes of the outer loop.

## 2.9 Mergesort

Given a pair of sub-arrays each of length $n/2$ that have already been sorted, merging their elements into a single sorted array is easy to do in around $n$ steps: just keep taking the lowest element from the sub-array that has it. In a previous course (*Foundations of Computer Science*) you have already seen the sorting algorithm based on this idea: split the input array into two halves and sort them recursively, stopping when the chunks so small that they are already sorted, and then merge the two sorted halves into one sorted array.

```
0  def mergeSort(a):
1      """*** DISCLAIMER: this is purposefully NOT a model of good code
2     (indeed it may hide subtle bugs---can you see them?) but it is
3     a useful starting point for our discussion. ***
4
5      BEHAVIOUR: Run the merge sort algorithm on the integer array a,
6      returning a sorted version of the array as the result. (Note that
7      the array is NOT sorted in place.)
8
9      PRECONDITION: array a contains len(a) integer values.
10
11     POSTCONDITION: a new array is returned that contains the same
12     integer values originally in a, but sorted in ascending order."""
13
14     if len(a) < 2:
15         # ASSERT: a is already sorted, so return it as is
16         return a
17
18     # Split array a into two smaller arrays a1 and a2
19     # and sort these recursively
20     h = int(len(a) / 2)
21     a1 = mergeSort(a[0:h])
22     a2 = mergeSort(a[h:END])
23
24     # Form a new array a3 by merging a1 and a2
25     a3 = new empty array of size len(a)
26     i1 = 0 # index into a1
27     i2 = 0 # index into a2
28     i3 = 0 # index into a3
29     while i1 < len(a1) or i2 < len(a2):
30         # ASSERT: i3 < len(a3)
31         a3[i3] = smallest(a1, i1, a2, i2) # updates i1 or i2 too
32         i3 = i3 + 1
33     # ASSERT: i3 == len(a3)
34     return a3
```

Compared to the other sorting algorithms seen so far, this one hides several subtleties, many to do with memory management issues, which may have escaped you when you studied it in ML:

- Merging two sorted sub-arrays (lines 24–32) is most naturally done by leaving the two input arrays alone and forming the result into a temporary buffer (line 25) as large as the combination of the two inputs. This means that, unlike the other algorithms seen so far, we cannot sort an array in place: we need additional space.

- The recursive calls of the procedure on the sub-arrays (lines 21–22) are easy to write in pseudocode and in several modern high level languages but they may involve additional acrobatics (wrapper functions etc) in languages where the size of the arrays handled by a procedure must be known in advance. The best programmers among you will learn a lot (and maybe find hidden bugs in the pseudocode above) by implementing mergesort in a programming language such as C, without automatic memory management.
- Merging the two sub-arrays is conceptually easy (just consume the "emerging" item from each deck of cards) but coding it up naïvely will fail on boundary cases, as the following exercise highlights.

**Exercise 9**
Can you spot any problems with the suggestion of replacing the somewhat mysterious line `a3[i3] = smallest(a1, i1, a2, i2)` with the more explicit and obvious `a3[i3] = min(a1[i1], a2[i2])`? What would be your preferred way of solving such problems? If you prefer to leave that line as it is, how would you implement the procedure `smallest` it calls? What are the trade-offs between your chosen method and any alternatives?

**Exercise 10**
In one line we return the same array we received from the caller, while in another we return a new array created within the merge-sort subroutine. This asymmetry is suspicious. Discuss potential problems.

How do we evaluate the running time of this recursive algorithm? The invocations that don't recurse have constant cost but for the others we must write a so-called *recurrence relation*. If we call $f(n)$ the cost of invoking mergesort on an input array of size $n$, then we have

$$f(n) = 2f(n/2) + kn,$$

where the first term is the cost of the two recursive calls (lines 21–22) on inputs of size $n/2$ and the second term is the overall cost of the merging

phase (lines 24–32), which is linear because a constant-cost sequence of operations is performed for each of the $n$ elements that is extracted from the sub-arrays `a1` and `a2` and placed into the result array `a3`.

To solve the recurrence, i.e. to find an expression for $f(n)$ that doesn't have $f$ on the right-hand side, let's "guess" that exponentials are going to help (since we split the input in two each time, doubling the number of arrays at each step) and let's rewrite the formula[15] with the substitution $n = 2^m$.

$$\begin{aligned}
f(n) &= f(2^m) \\
&= 2f(2^m/2) + k2^m \\
&= 2f(2^{m-1}) + k2^m \\
&= 2(2f(2^{m-2}) + k2^{m-1}) + k2^m \\
&= 2^2 f(2^{m-2}) + k2^m + k2^m \\
&= 2^2 f(2^{m-2}) + 2 \cdot k2^m \\
&= 2^2(2f(2^{m-3}) + k2^{m-2}) + 2 \cdot k2^m \\
&= 2^3 f(2^{m-3}) + k2^m + 2 \cdot k2^m \\
&= 2^3 f(2^{m-3}) + 3 \cdot k2^m \\
&= \dots \\
&= 2^m f(2^{m-m}) + m \cdot k2^m \\
&= f(1) \cdot 2^m + k \cdot m2^m \\
&= k_0 \cdot 2^m + k \cdot m2^m \\
&= k_0 n + kn \lg n.
\end{aligned}$$

We just proved that $f(n) = k_0 n + kn \lg n$ or, in other words, that $f(n) = O(n \lg n)$. Thus Mergesort is the first sorting algorithm we discuss in this course whose running time is better than quadratic. Much more than that, in fact: mergesort guarantees the *optimal* cost of $O(n \lg n)$, is relatively simple and has low time overheads. Its main disadvantage is that it requires extra space to hold the partially merged results. The implementation is trivial if one has another empty $n$-cell array available; but experienced programmers can get away with just $n/2$. Theoretical

[15] This is just an ad-hoc method for solving this particular recurrence, which may not work in all cases—though it's a powerful and versatile trick that we'll exploit several other times in this course. There is a whole theory on how to solve recurrences in chapter 4 of CLRS3.

computer scientists have been known to get away with just *constant* space overhead[16].

**Exercise 11**
Never mind the theoretical computer scientists, but how do you mergesort in $n/2$ space?

An alternative is to run the algorithm bottom-up, doing away with the recursion. Group elements two by two and sort (by merging) each pair. Then group the sorted pairs two by two, forming (by merging) sorted quadruples. Then group those two by two, merging them into sorted groups of 8, and so on until the last pass in which you merge two large sorted groups. Unfortunately, even though it eliminates the recursion, this variant still requires $O(n)$ additional temporary storage, because to merge two groups of $k$ elements each into a $2k$ sorted group you still need an auxiliary area of $k$ cells (move the first half into the auxiliary area, then repeatedly take the smallest element from either the second half or the auxiliary area and put it in place).

**Exercise 12**
Justify that the merging procedure just described will not overwrite any of the elements in the second half.

**Exercise 13**
Write pseudocode for the bottom-up mergesort.

[16] Cfr. Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola. "Practical in-place mergesort". *Nordic Journal of Computing* 3:27-–40, 1996. Note that real programmers and theoretical computer scientists tend to assign different semantics to the word "practical".

## 2.10 Quicksort

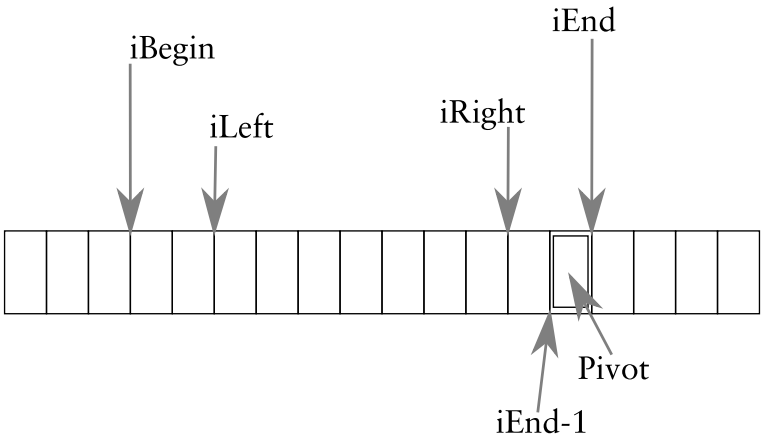
**Textbook**

Study chapter 7 in CLRS3.

Quicksort is the most elaborate of the sorting algorithms you have already seen, from a functional programming viewpoint, in the *Foundations* course. The main thing for you to understand and appreciate in this second pass is how cleverly it manages to sort the array in place, splitting it into a "lower" and a "higher" part without requiring additional storage. You should also have a closer look at what happens in presence of duplicates.

The algorithm is relatively easy to explain and, when properly implemented and applied to non-malicious input data, the method can fully live up to its name. However Quicksort is somewhat temperamental. It is remarkably easy to write a program based on the Quicksort idea that is wrong in various subtle cases (e.g. if all the items in the input list are identical) and, although in almost all cases Quicksort turns in a time proportional to $n \lg n$ (with a quite small constant of proportionality), for worst case input data it can be as slow as $n^2$. There are also several small variants. It is strongly recommended that you study the description of Quicksort in your favourite textbook and that you look carefully at the way in which code can be written to avoid degenerate cases leading to accesses off the end of arrays etc.

The core idea of Quicksort, as you will recall from the *Foundations* course, is to select some value from the input and use that as a "pivot" to split the other values into two classes: those smaller and those larger than the pivot. What happens when applying this idea to an array rather than a list? A selection procedure partitions the values so that the lower portion of the array holds values not exceeding that of the pivot, while the upper part holds only larger values. This selection can be performed in place by scanning in from the two ends of the array, exchanging values as necessary. Then the pivot is placed where it belongs, so that the array contains a first region (still unsorted) with the low values, then the pivot, then a third region (still unsorted) with the high values. For an $n$ element array it takes about $n$ comparisons and data exchanges to partition the array. Quicksort is then called recursively to deal with the low and high parts of the data, and the result is obviously that the entire array ends up perfectly sorted.



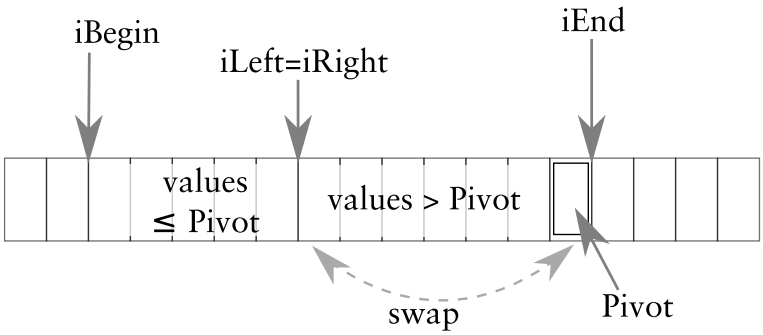

Let's have a closer look at implementing Quicksort. Remember we scan the array (or sub-array) from both ends to partition it into three regions. Assume you must sort the sub-array `a[iBegin:iEnd]`, which contains the cells from `a[iBegin]` (included) to `a[iEnd]` (excluded)[17]. We use two auxiliary indices `iLeft` and `iRight`. We arbitrarily pick the last element in the range as the pivot: `Pivot = A[iEnd - 1]`. Then `iLeft` starts at `iBegin` and moves right, while `iRight` starts at `iEnd - 1` and moves left. All along, we maintain the following invariants:

- `iLeft` $\leq$ `iRight`
- `a[iBegin:iLeft]` only has elements $\leq$ `Pivot`
- `a[iRight:iEnd - 1]` only has elements $>$ `Pivot`

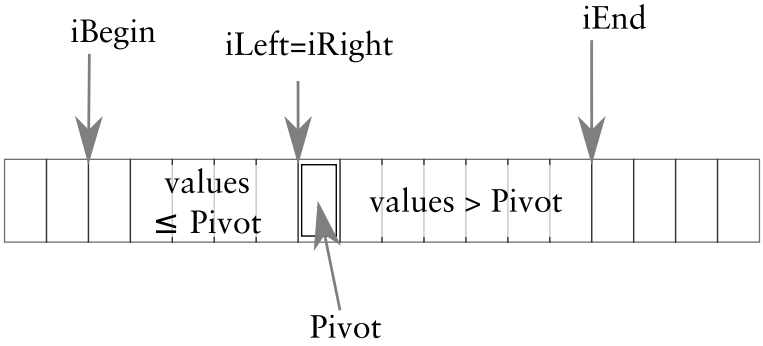
So long as `iLeft` and `iRight` have not met, we move `iLeft` as far right as possible and `iRight` as far left as possible without violating the invariants. Once they stop, if they haven't met, it means that `A[iLeft]` $>$ `Pivot` (otherwise we could move `iLeft` further right) and that `A[iRight - 1]` $\leq$ `Pivot` (thanks to the symmetrical argument[18]). So we swap these two elements pointed to by `iLeft` and `iRight - 1`. Then we repeat the process, again pushing `iLeft` and `iRight` as far towards each other as possible, swapping array elements when the indices stop and continuing until they touch.

[17] See the boxed aside on page 15.

[18] Observe that, in order to consider `iRight` the symmetrical mirror-image of `iLeft`, we must consider `iRight` to be pointing, conceptually, at the cell to its *left*, hence the `- 1`.




At that point, when `iLeft` = `iRight`, we put the pivot in its rightful place between the two regions we created, by swapping `A[iRight]` and `[iEnd - 1]`.




We then recursively run Quicksort on the two smaller sub-arrays `a[iBegin:iLeft]` and `a[iRight + 1:iEnd]`.

Now let's look at performance. Consider first the ideal case, where each selection manages to split the array into two equal parts. Then the total cost of Quicksort satisfies $f(n) = 2f(n/2) + kn$, and hence grows as $O(n \lg n)$ as we proved in section 2.9. But, in the worst case, the array might be split very unevenly—perhaps at each step only a couple of items, or even none, would end up less than the selected pivot. In that case the recursion (now $f(n) = f(n-1) + kn$) will go around $n$ deep, and therefore the total worst-case costs will grow to be proportional to $n^2$.

One way of estimating the average cost of Quicksort is to suppose that the pivot could equally probably have been any one of the items in the data. It is even reasonable to use a random number generator to select an arbitrary item for use as a pivot to ensure this.

**Exercise 14**
Can picking the pivot at random *really* make any difference to the expected performance? How will it affect the average case? The worst case? Discuss.

Then it is easy to set up a recurrence formula that will be satisfied by the average cost:

$$f(n) = kn + \frac{1}{n}\sum_{i=1}^{n}\Big(f(i-1) + f(n-i)\Big)$$

where the $kn$ term is the cost of partitioning, whereas the summation adds up, with equal weight, the expected costs corresponding to all the (equally probable) ways in which the partitioning might happen. After some amount of playing with this equation, it can be established that the average cost for Quicksort is $\Theta(n \lg n)$.

Quicksort provides a sharp illustration of what can be a problem when selecting an algorithm to incorporate in an application. Although its average performance (for random data) is good, it does have a quite unsatisfactory (albeit uncommon) worst case. It should therefore not be used in applications where the worst-case costs could have safety implications. The decision about whether to use Quicksort for good average speed or a slightly slower but guaranteed $O(n \lg n)$ method can be a delicate one.

There are a great number of small variants on the Quicksort scheme and the best way to understand them for an aspiring computer scientist is to code them up, sprinkle them with diagnostic print statements and run them on examples. There are good reasons for using the median[19] of the mid point and two others as the pivot at each stage, and for using recursion only on partitions larger than a preset threshold. When the region is small enough, Insertion sort may be used instead of recursing down. A less intuitive but probably more economical arrangement is for Quicksort just to *return* (without sorting) when the region is smaller than a threshold; then one runs Insertion sort over the messy array produced by the truncated Quicksort.

**Exercise 15**
Justify why running Insertion sort over the messy array produced by the truncated Quicksort might not be as stupid as it may sound at first. How should the threshold be chosen?

[19] Cfr. section 2.11.

## 2.11 Median and order statistics using Quicksort

**Textbook**

Study chapter 9 in CLRS3.

The **median** of a collection of values is the one such that as many items are smaller than that value as are larger. In practice, when we look for algorithms to find a median, it is wise to expand to more general **order statistics**: we shall therefore look for the item that ranks at some parametric position $k$ in the data. If we have $n$ items, the median corresponds to taking the special case $k = n/2$, while $k = 1$ and $k = n$ correspond to looking for minimum and maximum values.

One obvious way of solving this problem is to sort that data: then the item with rank $k$ is trivial to read off. But that costs $O(n \lg n)$ for the sorting.

Two variants on Quicksort are available that solve the problem. One has linear cost in the average case but, like Quicksort itself, has a quadratic worst-case cost. The other is more elaborate to code and has a much higher constant of proportionality, but guarantees linear cost. In cases where guaranteed worst-case performance is essential the second method might in theory be useful; in practice, however, it is so complicated and slow that it is seldom implemented[20].

**Exercise 16**
What is the smallest number of pairwise comparisons you need to perform to find the smallest of $n$ items?

**Exercise 17**
*(More challenging.)* And to find the *second* smallest?

[20] In this course we no longer describe the overly elaborate (though, to some, perversely fascinating) guaranteed-linear-cost method. It's explained in your textbook if you're curious: see CLRS3, 9.3.

The simpler scheme selects a pivot and partitions as for Quicksort, at linear cost. Now suppose that the partition splits the array into two parts, the first having size $p$, and imagine that we are looking for the item with rank $k$ in the whole array. If $k < p$ then we just continue be looking for the rank-$k$ item in the lower partition. Otherwise we look for the item with rank $k - p$ in the upper one. The cost recurrence for this method (assuming, unreasonably optimistically, that each selection stage divides out values neatly into two even sets) is $f(n) = f(n/2)+kn$, whose solution exhibits linear growth as we shall now prove. Setting $n = 2^m$ as we previously did (and for the same reason), we obtain

$$\begin{aligned}
f(n) &= f(2^m) \\
&= f(2^m/2) + k2^m \\
&= \underbrace{f(2^{m-1})} + k2^m \\
&= \underbrace{f(2^{m-2}) + k2^{m-1}} + k2^m \\
&= f(2^{m-3}) + k2^{m-2} + k2^{m-1} + k2^m \\
&= \ldots \\
&= f(2^{m-m}) + k2^{m-(m-1)} + \ldots + k2^{m-2} + k2^{m-1} + k2^m \\
&= f(2^0) + k(2^1 + 2^2 + 2^3 + \ldots 2^m) \\
&= f(1) + 2k(2^m - 1) \\
&= k_0 + k_1 2^m \\
&= k_0 + k_1 n
\end{aligned}$$

which is indeed $O(n)$, QED.

As with Quicksort itself, and using essentially the same arguments, it can be shown that this best-case linear cost also applies to the average case; but, equally, that the worst-case, though rare, has quadratic cost.

## 2.12 Heapsort

**Textbook**

Study chapter 6 in CLRS3.

Despite the good average behaviour of Quicksort, there are circumstances where one might want a sorting method that is guaranteed to

run in time $O(n \lg n)$ whatever the input[21], even if such a guarantee may cost some increase in the constant of proportionality.

Heapsort is such a method, and is described here not only because it is a reasonable sorting scheme, but because the data structure it uses (called a **heap**, a term that is also used, but with a totally different meaning, in the context of free-storage management) has other useful applications.

Consider an array that has values stored in all its cells, but with the constraint (known as "the heap property") that the value at position $k$ is greater than (or equal to) those at positions[22] $2k + 1$ and $2k + 2$. The data in such an array is referred to as a heap. The heap is isomorphic to a binary tree in which each node has a value at least as large as those of its children—which, as one can easily prove, means it is also the largest value of all the nodes in the subtree of which it is root. The root of the heap (and of the equivalent tree) is the item at location 0 and, by what we just said, it is the largest value in the heap.

The data structure we just described, which we'll use in heapsort, is also known as a max-heap. You may also encounter the dual arrangement, appropriately known as min-heap, where the value in each node is at least as *small* as the values in its children; there, the root of the heap is the *smallest* element (see section 4.8).

Note that any binary tree that represents a binary heap must have a particular "shape", known as **almost full binary tree**: every level of the tree, except possibly the last, must be full, i.e. it must have the maximum possible number of nodes; and the last level must either be full or have empty spaces only at its right end. This constraint on the shape comes from the isomorphism with the array representation: a binary tree with any other shape would map back to an array with "holes" in it.

**Exercise 18**
What are the minimum and maximum number of elements in a heap of height $h$?

The Heapsort algorithm consists of two phases. The first phase takes an array full of unsorted data and rearranges it in place so that the data forms a heap. Amazingly, this can be done in linear time, as we shall prove shortly. The second phase takes the top (leftmost) item from the

[21] Mergesort does; but, as we said, it can't sort the array in place.

[22] Supposing that those two locations are still within the bounds of the array, and assuming that indices start at 0.

heap (which, as we saw, was the largest value present) and swaps it to the last position in the array, which is where that value needs to be in the final sorted output. It then has to rearrange the remaining data to be a heap with one fewer element. Repeating this step will leave the full set of data in order in the array. Each heap reconstruction step has a cost bounded by the logarithm of the amount of data left, and thus the total cost of Heapsort ends up being bounded by $O(n \lg n)$, which is optimal.

The auxiliary function `heapify` (lines 22–36) takes a (sub)array that is almost a heap and turns it into a heap. The assumption is that the two (possibly empty) subtrees of the root are already proper heaps, but that the root itself may violate the max-heap property, i.e. it might be smaller than one or both of its children. The `heapify` function works by swapping the root with its largest child, thereby fixing the heap property for that position. What about the two subtrees? The one not affected by the swap was already a heap to start with, and after the swap the root of the subtree is certainly $\leq$ than its parent, so all is fine there. For the other subtree, all that's left to do is ensure that the old root, in its new position further down, doesn't violate the heap property there. This can be done recursively. The original root therefore sinks down step by step to the position it should rightfully occupy, in no more calls than there are levels in the tree. Since the tree is "almost full", its depth is the logarithm of the number of its nodes, so `heapify` is $O(\lg n)$.

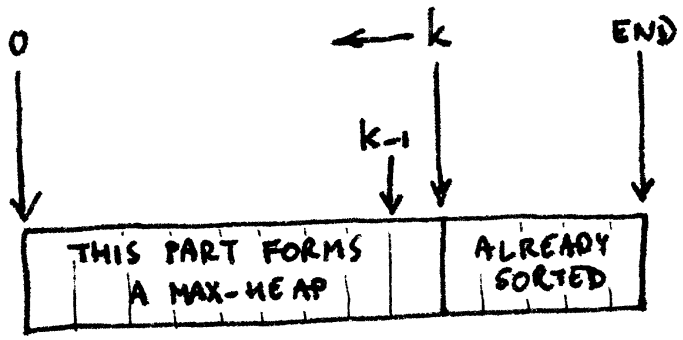
```
0  def heapSort(a):
1      """BEHAVIOUR: Run the heapsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      # First, turn the whole array into a heap
10     for k from floor(END/2) excluded down to 0 included: # nodes with children
11         heapify(a, END, k)
12
13     # Second, repeatedly extract the max, building the sorted array R-to-L
14     for k from END included down to 1 excluded:
15         # ASSERT: a[0:k] is a max-heap
16         # ASSERT: a[k:END] is sorted in ascending order
17         # ASSERT: every value in a[0:k] is <= than every value in a[k:END]
18         swap(a[0], a[k - 1])
19         heapify(a, k - 1, 0)
20
```

```
21
22 def heapify(a, iEnd, iRoot):
23     """BEHAVIOUR: Within array a[0:iEnd], consider the subtree rooted
24     at a[iRoot] and make it into a max-heap if it isn't one already.
25
26     PRECONDITIONS: 0 <= iRoot < iEnd <= END. The children of
27     a[iRoot], if any, are already roots of max-heaps.
28
29     POSTCONDITION: a[iRoot] is root of a max-heap."""
30
31     if a[iRoot] satisfies the max-heap property:
32         return
33     else:
34         let j point to the largest among the existing children of a[iRoot]
35         swap(a[iRoot], a[j])
36         heapify(a, iEnd, j)
```

The first phase of the main `heapSort` function (lines 9–11) starts from the bottom of the tree (rightmost end of the array) and walks up towards the root, considering each node as the root of a potential sub-heap and rearranging it to be a heap. In fact, nodes with no children can't possibly violate the heap property and therefore are automatically heaps; so we don't even need to process them—that's why we start from the midpoint `floor(END/2)` rather than from the end. By proceeding right-to-left we guarantee that any children of the node we are currently examining are already roots of properly formed heaps, thereby matching the precondition of `heapify`, which we may therefore use. It is then trivial to put an $O(n \lg n)$ bound on this phase—although, as we shall see, it is not tight.

In the second phase (lines 13–19), the array is split into two distinct parts: `a[0:k]` is a heap, while `a[k:END]` is the "tail" portion of the sorted array. The rightmost part starts empty and grows by one element at each pass until it occupies the whole array. During each pass of the loop in lines 13–19 we extract the maximum element from the root of the heap, `a[0]`, reform the heap and then place the extracted maximum in the empty space left by the last element, `a[k]`, which conveniently is just where it should go[23]. To retransform `a[0:k - 1]` into a heap after placing `a[k - 1]` in position `a[0]` we may call `heapify`, since the two subtrees of the root `a[0]` must still be heaps given that all that changed was the root and we started from a proper heap. For this second phase, too, it is trivial to establish an $O(n \lg n)$ bound.

[23] Because all the items in the right part are $\geq$ than the ones still in the heap, since each of them was the maximum at the time of extraction.




Now, what was that story about the first phase actually taking *less* than $O(n \lg n)$? Well, it's true that all heaps are at most $O(\lg n)$ tall, but many of them are much shorter because most of the nodes of the tree are found in the lower levels[24], where they can only be roots of short trees. So let's redo the budget more accurately.

| level | num nodes in level | height of tree | max cost of heapify |
|---|---|---|---|
| 0 | 1 | $h$ | $kh$ |
| 1 | 2 | $h-1$ | $k(h-1)$ |
| 2 | 4 | $h-2$ | $k(h-2)$ |
| ... | | | |
| $j$ | $2^j$ | $h-j$ | $k(h-j)$ |
| ... | | | |
| $h$ | $2^h$ | 0 | 0 |

By "max cost of heapify" we indicate a bound on the cost of performing the heapify operation on any node on the given level. The total cost for a given level cannot exceed that amount, i.e. $k(h-j)$, times the number of nodes in that level, $2^j$. The cost for the whole tree, as a function of the number of levels, is simply the sum of the costs of all levels:

$$\begin{aligned} C(h) &= \sum_{j=0}^{h} 2^j \cdot k(h-j) \\ &= k\frac{2^h}{2^h} \sum_{j=0}^{h} 2^j (h-j) \\ &= k2^h \sum_{j=0}^{h} 2^{j-h} (h-j) \end{aligned}$$

[24] Indeed, in a full binary tree, each level contains one more node than the *whole* tree above it.

$$
\begin{aligned}
& \ldots \text{let } l = h - j \ldots \\
= \quad & k2^h \sum_{l=0}^{h} l2^{-l} \\
= \quad & k2^h \sum_{l=0}^{h} l \left(\frac{1}{2}\right)^l
\end{aligned}
$$

The interesting thing is that this last summation, even though it is a monotonically growing function of $h$, is in fact bounded by a constant, because the corresponding series converges to a finite value if the absolute value of the base of the exponent (here $\frac{1}{2}$) is less than 1:

$$|x| < 1 \quad \Rightarrow \quad \sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}.$$

This means that the cost $C(h)$ grows like $O(2^h)$ and, if we instead express this in terms of the number of nodes in the tree, $C(n) = O(n)$ and not $O(n \lg n)$, QED.

Heapsort therefore offers at least two significant advantages over other sorting algorithms: it offers an asymptotically optimal worst-case complexity of $O(n \lg n)$ *and* it sorts the array in place. Despite this, on non-pathological data it is still usually beaten by the amazing Quicksort.

## 2.13 Stability of sorting methods

Data to be sorted often consists of records made of key and payload; the key is what the ordering is based upon, while the payload is some additional data that is just carried around in the rearranging process. In some applications one can have keys that should be considered equal, and then a simple specification of sorting might not indicate the order in which the corresponding records should end up in the output list. "Stable" sorting demands that, in such cases, the order of items in the input be preserved in the output. Some otherwise desirable sorting algorithms are not stable, and this can weigh against them.

If stability is required, despite not being offered by the chosen sorting algorithm, the following technique may be used. Extend the records with an extra field that stores their original position, and extend the ordering predicate used while sorting to use comparisons on this field to break ties. Then, any arbitrary sorting method will rearrange the data in a stable way, although this clearly increases space and time overheads a little (but by no more than a linear amount).

**Exercise 19**
For each of the sorting algorithms seen in this course, establish whether it is stable or not.

## 2.14 Faster sorting

**Textbook**

Study chapter 8 in CLRS3.

If the condition that sorting must be based on pair-wise comparisons is dropped it may sometimes be possible to do better than $O(n \lg n)$ in terms of asymptotic costs. In this section we consider three algorithms that, under appropriate assumptions, sort in linear time. Two particular cases are common enough to be of at least occasional importance: sorting integer keys from a fixed range (*counting sort*) and sorting real keys uniformly distributed over a fixed range (*bucket sort*). Another interesting algorithm is *radix sort*, used to sort integer numerals of fixed length[25], which was originally used to sort punched cards mechanically.

### 2.14.1 Counting sort

Assume that the keys to be sorted are integers that live in a known range, and that the range is fixed regardless of the number of values to be processed. If the number of input items grows beyond the cardinality of the range, there will necessarily be duplicates in the input. If no data is involved at all beyond the integers, one can set up an array whose size is determined by the range of integers that can appear (not by the amount of data to be sorted) and initialize it to all 0s. Then, for each item in the input data, $w$ say, the value at position $w$ in the array is incremented. At the end, the array contains information about how many instances of each value were present in the input, and it is easy to create a sorted output list with the correct values in it. The costs are obviously linear.

If additional satellite data beyond the keys is present (as will usually happen) then, once the counts have been collected, a second scan through the input data can use the counts to indicate the exact position, in the

[25] Or, more generally, any keys (including character strings) that can be mapped to fixed-length numerals in an arbitrary base.

output array, to which each data item should be moved. This does not compromise the overall linear cost.

During the second pass, the fact of scanning the items in the order in which they appear in the input array ensures that items with the same key maintain their relative order in the output. Thus counting sort is not only fast but also stable. It doesn't, however, sort in place.

**Exercise 20**
Give detailed pseudocode for the counting sort algorithm (particularly the second phase), ensuring that the overall cost stays linear. Do you need to perform any kind of precomputation of auxiliary values?

### 2.14.2 Bucket sort

Assume the input data is guaranteed to be uniformly distributed over some known range (for instance it might be real numbers in the range 0.0 to 1.0). Then a numeric calculation on the key can predict with reasonable accuracy where a value must be placed in the output. If the output array is treated somewhat like a hash table (cfr. section 4.7), and this prediction is used to insert items in it, then, apart from some local clustering effects, that data has been sorted.

To sort $n$ keys uniformly distributed between 0.0 and 1.0, create an array of $n$ linked lists and insert each key $k$ to the list at position $\lfloor k \cdot n \rfloor$. This phase has linear cost. (We expect each list to be one key long on average, though some may be slightly longer and some may be empty.) In the next phase, for each entry in the array, sort the corresponding list with insertsort if it is longer than one element, then output it.

Insertsort, as we know, has a quadratic worst-case running time. How does this affect the running time of bucket sort? If we could assume that the lists are never longer than a constant $k$, it would be trivial to show that the second pass too has linear costs in the worst case. However we can't, so we need to resort to a rather more elaborate argument. But it is possible to prove that, under the assumption that the input values are uniformly distributed, the *average-case* (but not worst-case) overall running time of bucket sort is linear in $n$.

### 2.14.3 Radix sort

Historically, radix sort was first described in the context of sorting integers encoded on punched cards, where each column of a card represented a digit by having a hole punched in the corresponding row. A mechanical device could be set to examine a particular column and distribute the cards of a deck into bins, one per digit, according to the digit in that column. Radix sort used this "primitive" to sort the whole deck.

The obvious way to proceed is perhaps to sort on the most significant digit, then recursively for each bin on the next significant digit, then on the next, all the way down to the least significant digit. But this would require a great deal of temporary "desk space" to hold the partial mini-decks still to be processed without mixing them up.

Radix sort instead proceeds, counter-intuitively, from the least significant digit upwards. First, the deck is sorted into $b = 10$ bins based on the least significant digit. Then the contents of the bins are collected together, in order, to reform a full deck, and this is then sorted according to the next digit up. But the per-digit sorting method used is chosen to be stable, so that cards that have the same second digit still maintain their relative order, which was induced by the first (least significant) digit. The procedure is repeated going upwards towards the most significant digits. Before starting pass $i$, the digits in positions 0 to $i-1$ have already been sorted[26]. During pass $i$, the deck is sorted on the digit in position $i$, but all the cards with the same $i$ digit remain in the relative order determined by the even less significant digits to their right. The result is that, once the deck has been sorted on the most significant digit, it is fully sorted. The number of passes is equal to the number of digits ($d$) in the numerals being sorted and the cost of each pass can be made linear by using counting sort.

**Exercise 21**
Why couldn't we simply use counting sort in the first place, since the keys are integers in a known range?

Note that counting sort does not sort in place; therefore, if that is the stable sort used by radix sort, neither does radix sort. This, as well as the constants hidden by the big-O notation, must be taken into account when deciding whether radix sort is advantageous, in a particular application, compared to an in-place algorithm like quicksort.

[26] Position 0 being of course the least significant digit, i.e. the rightmost column.

# Chapter 3

# Algorithm design

**Chapter contents**

Strategies for algorithm design: dynamic programming, divide and conquer, greedy algorithms and other useful paradigms.
Expected coverage: about 3 lectures.
Study 4, 15, 16 in CLRS3.

There exists no general recipe for designing an algorithm that will solve a given problem—never mind designing an optimally efficient one. There are, however, a number of generally useful strategies, some of which we have seen in action in the sorting algorithms discussed so far. Each design paradigm works well in at least some cases; with the flair you acquire from experience you may be able to choose an appropriate one for your problem.

In this chapter we shall first study a powerful technique called dynamic programming. After that we shall name and describe several other paradigms, some of which you will recognize from algorithms we already discussed, while others you will encounter later in the course. None of these are guaranteed to succeed in all cases but they are all instructive ways of approaching algorithm design.

## 3.1 Dynamic programming

**Textbook**

Study chapter 15 in CLRS3.

Sometimes it makes sense to work up towards the solution to a problem by building up a table of solutions to smaller versions of the problem. For historical reasons, this process is known as "dynamic programming", but the use of the term "programming" in this context comes from operations research rather than computing and has nothing to do with our usual semantics of writing instructions for a machine: it originally meant something like "finding a plan of action".

Dynamic programming is related to the strategy of "divide and conquer" (section 3.3.3, q.v., but already seen in mergesort and quicksort) in that it breaks up the original problem recursively into smaller problems that are easier to solve. But the essential difference is that here the subproblems may overlap. Applying the divide and conquer approach in this setting would inefficiently solve the same subproblems again and again along different branches of the recursion tree. Dynamic programming, instead, is based on computing the solution to each subproblem only once: either by remembering the intermediate solutions and reusing them as appropriate instead of recomputing them; or, alternatively, by deriving out the intermediate solutions in an appropriate bottom-up order that makes it unnecessary to recompute old ones again and again.

This method has applications in various tasks related to combinatorial search. It is difficult to describe it both accurately and understandably without having demonstrated it in action so let's use examples.

An instructive preliminary exercise is the computation of Fibonacci numbers:

**Exercise 22**
Leaving aside for brevity Fibonacci's original 1202 problem on the sexual activities of a pair of rabbits, the Fibonacci sequence may be more abstractly defined as follows:

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{for } n \geq 2 \end{cases}$$

(This yields $1, 1, 2, 3, 5, 8, 13, 21, \ldots$)
In a couple of lines in your favourite programming language, write a *recursive* program to compute $F_n$ given $n$, using the definition above. And now, finally, the question: how many function calls will your recursive program perform to compute $F_{10}$, $F_{20}$ and $F_{30}$? First, guess; then instrument your program to tell you the actual answer.

Where's the dynamic programming in this? We'll come back to that, but note the contrast between the exponential number of recursive calls, where smaller Fibonacci numbers are recomputed again and again, and the trivial and much more efficient iterative solution of computing the Fibonacci numbers bottom-up.

A more significant example is the problem of finding the best order in which to perform matrix chain multiplication. If $A$ is a $p \times q$ matrix and $B$ is a $q \times r$ matrix, the product $C = AB$ is a $p \times r$ matrix whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{q-1} a_{i,k} \cdot b_{k,j}.$$

The product matrix can thus be computed using $p \cdot q \cdot r$ scalar multiplications ($q$ multiplications[1] for each of the $p \cdot r$ elements). If we wish to compute the product $A_1A_2A_3A_4A_5A_6$ of 6 matrices, we have a wide choice of the order in which we do the matrix multiplications.

**Exercise 23**
Prove (an example is sufficient) that the order in which the matrix multiplications are performed may dramatically affect the total number of scalar multiplications—despite the fact that, since matrix multiplication is associative, the final matrix stays the same.

[1] As well as $q - 1$ sums.

Suppose the dimensions of $A_1$, $A_2$, $A_3$, $A_4$, $A_5$ and $A_6$ are respectively $30 \times 35$, $35 \times 15$, $15 \times 5$, $5 \times 10$, $10 \times 20$ and $20 \times 25$. Clearly, adjacent dimensions must be equal or matrix multiplication would be impossible; therefore this set of dimensions can be specified by the vector $(p_0, p_1, \ldots, p_6) = (30, 35, 15, 5, 10, 20, 25)$. The problem is to find an order of multiplications that minimizes the number of scalar multiplications needed.

Observe that, with this notation, the dimensions of matrix $A_i$ are $p_{i-1} \times p_i$. Suppose that $i, j, k$ are integer indices with $i \leq k \leq j$, that $A_{i..j}$ stands for the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ and that $m(i, j)$ is the minimum number of scalar multiplications to compute the product $A_{i..j}$. Then $m$ can be defined as a recursive function:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{k \in [i,j)} \{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i \neq j \end{cases}$$

With a sequence $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ of $j - i + 1$ matrices, $j - i$ matrix multiplications are required. The last such matrix multiplication to be performed will, for some $k$ between $i$ and $j$, combine a left-hand cumulative matrix $A_{i..k}$ (with $i < k$) and a right-hand one $A_{k+1..j}$ (with $k+1 < j$). The cost of that matrix multiplication will be $p_{i-1} \cdot p_k \cdot p_j$. The expression above for the function $m(i, j)$ is obtained by noting that one of the possible values of $k$ must be the one yielding the minimum total cost and that, for that value, the two contributions from the cumulative matrices must also each be of minimum cost.

For the above numerical problem the answer is $m(1, 6) = 15125$ scalar multiplications. A naïve recursive implementation of this function will compute $m(i, j)$ in time exponential in the number of matrices to be multiplied[2], but the value can be obtained more efficiently by computing and remembering the values of $m(i, j)$ in a systematic order so that, whenever $m(i, k)$ or $m(k+1, j)$ is required, the values are already known.

An alternative approach, as hinted at above, is to modify the simple-minded recursive definition of the $m()$ function so that it checks whether $m(i, j)$ has already been computed. If so, it immediately returns with the previously computed result, otherwise it computes and saves the result in a table before returning. This technique is known as **memoization** (this not a typo for "memorization"—it comes from "jotting down a memo" rather than from "memorizing"). For the previous example, the naïve implementation computes $m(1, 6) = 15125$ in 243 invocations of $m()$,

[2]Do not confuse the act of computing $m$ (minimum number of multiplications, and related choice of which matrices to multiply in which order) and the act of computing the matrix product itself. The former is the computation of a *strategy* for performing the latter cheaply. But *either* of these acts can be computationally expensive.

while a memoized version yields the same result in only 71 invocations (try it).

Going back to general principles, dynamic programming tends to be useful against problems with the following features:

1. There exist many choices, each with its own "score" which must be minimized or maximized (optimization problem). *In the example above, each parenthesization of the matrix expression is a choice; its score is the number of scalar multiplications it involves.*

2. The number of choices is exponential in the size of the problem, so brute force is generally not applicable. *The number of choices here is the number of possible binary trees that have the given sequence of matrices (in order) as leaves.*

3. The structure of the optimal solution is such that it is composed of optimal solutions to smaller problems. *The optimal solution ultimately consists of multiplying together two cumulative matrices; these two matrices must themselves be optimal solutions for the corresponding subproblems because, if they weren't, we could substitute the optimal sub-solutions and get a better overall result.*

4. There is overlap: in general, the optimal solution to a sub-problem is required to solve several higher-level problems, not just one. *The optimal sub-solution $m(2,4)$ is needed in order to compute $m(2,5)$ but also to compute $m(1,4)$.*

Because of the fourth property of the problem, a straightforward recursive divide-and-conquer approach will end up recomputing the common sub-solutions many times (just like the wasteful recursive Fibonacci), unless it is turned into dynamic programming through memoization.

The non-recursive, bottom-up approach to dynamic programming consists instead of building up the optimal solutions incrementally, starting from the smallest ones and going up gradually.

To solve a problem with dynamic programming one must define a suitable sub-problem structure so as to be able to write a kind of recurrence relation that describes the optimal solution to a sub-problem in terms of optimal solutions to smaller sub-problems.

## 3.2 Greedy algorithms

**Textbook**

Study chapter 16 in CLRS3.

Many algorithms involve some sort of optimization. The idea of "greed" is to start by performing whatever operation contributes as much as any single step can towards the final goal. The next step will then be the best step that can be taken from the new position and so on. The procedures for finding minimal spanning sub-trees, described later in the course, are examples of how greed can sometimes lead to good results. Other times, though, greed can get you stuck in a local maximum—so it's always a prudent idea to develop a correctness proof before blindly using a greedy algorithm.

Most problems that can be solved by a greedy algorithm can also be solved by dynamic programming: both strategies exploit the optimal structure of the sub-problems. However the greedy strategy, when it can be applied, is a much cheaper way of reaching the overall optimal solution. Let's start with an example.

A university has a sports hall that can be used for a variety of activities, but only one at a time. The various sports societies of the University propose bookings for the hall, in the form of (start, finish) time segments, and the management wishes to maximize the number of proposals it can satisfy. More formally, we are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ activities, each of the form $a_i = (s_i, f_i)$, sorted for convenience in order of finishing time ($f_1 \leq f_2 \leq \ldots \leq f_n$), and we wish to find a maximum-size subset of $S$ in which no two activities overlap.

The number of subsets of $S$ is $2^n$, since each of the $n$ elements of $S$ can either be or not be in the subset. Examining each subset (to establish whether its activities are compatible and then selecting a maximum-cardinality subset among those that pass this filter) has exponential cost and is therefore infeasible other than for very small $n$.

Using the dynamic programming approach, we note that assuming activity $a_i$ to be a member of the optimal subset identifies two further subsets: excluding the activities that overlap with $a_i$, we have a first set $S_{iL}$ with activities that complete before $a_i$ starts, and a second set $S_{iR}$ with activities that start after $a_i$ finishes. If the optimal solution does include $a_i$, then it must consist of $a_i$ together with the optimal solution for $S_{iL}$ and the optimal solution for $S_{iR}$. (It's easy to show that, if it didn't, a "more optimal" solution could be built by including them instead. This

is the usual "cut and paste" argument used in dynamic programming.) If, conversely, the optimal solution doesn't include $a_i$, then by the same argument the union of $\{a_i\}$, the optimal solution for $S_{iL}$ and the optimal solution for $S_{iR}$ is still the best solution we can produce that includes $a_i$.

We don't know whether any particular $a_i$ will be included in the optimal solution, but we know that at least one of them will, so we try all possible values of $i$ and pick the one yielding the subset of greatest cardinality. If we indicate with $opt(X)$ the cardinality of the maximum subset of non-overlapping activities in set $X$, then we can define the function recursively as:

$$opt(\{\}) = 0$$

$$opt(S) = \max_{0 \le i \le n} (opt(S_{iL}) + 1 + opt(S_{iR}))$$

with $S_{iL}$ and $S_{iR}$ defined as above. We can then either use recursion[3] and memoize, or use iteration and compute the subsets bottom-up, thus in either case bringing the costs down from exponential to polynomial in $n$.

This is a great improvement, but the greedy strategy takes a more daring gamble: instead of keeping all options open, and deciding on the maximum only after having tried all possibilities and having unwound all the levels of the memoized recursion, the greedy strategy says "hey, I bet I can tell you one of these activities that is in the subset"; then, assuming the guess is correct, the optimal solution must consist of *that* activity plus the optimal solution to the remaining ones that don't overlap with it; and so forth. Very similar in structure to dynamic programming but much more direct because we immediately commit to one choice at every

[3]The notation I used above is simplified for clarity of explanation, but incomplete. To recurse down into subproblems we need a more complicated notation such as the one used in the CLRS3 textbook. The simplified notation used above is OK at the top level of the problem but is insufficiently powerful to express that, lower down, I want (say) the activities that complete before $a_{i'}$ starts *but* within those that start after the higher-level $a_i$ finishes. The full notation in CLRS3 defines $S_{ij}$ as the set of activities that start after the end of $a_i$ and finish before the start of $a_j$, thus

$$S_{ij} = \{a_k \in S \quad : \quad f_i \le s_k \le f_k \le s_j\}.$$

And then you'd need some extra notational acrobatics (not quite specified even in the book) to express the top-level starting set $S$, which can't be generated from the above definition as $S_{ij}$ by assigning in-range values to $i$ and $j$. The cardinality of the best possible overall solution containing activity $a_i$ will still be $opt(S_{iL}) + 1 + opt(S_{iR})$ but we will now also be able to speak of the best possible solution containing activity $a_k$ *for the subset of activities that start after the end of* $a_i$ *and finish before the start of* $a_j$. This additional expressive power is required to recurse into the subproblems.

stage, discarding all the others a priori. (Of course this relies on being able to make the correct choice!)

In this example the greedy strategy is to pick the activity that finishes first, based on the intuition that it's the one that leaves most of the rest of the timeline free for the allocation of other activities. Now, to be able to use the greedy strategy safely on this problem, we must prove that this choice is indeed optimal; in other words, that the $a_i \in S$ with the smallest $f_i$ is included in an optimal solution for $S$. (That would be $a_1$, by the way, since we conveniently said that activities were numbered in order of increasing finishing time.)

Proof by contradiction. Assume there exists an optimal solution $O \subset S$ that does not include $a_1$. Let $a_x$ be the activity in $O$ with the earliest finishing time. Since $a_1$ had the smallest finishing time in all $S$, $f_1 \leq f_x$. There are two cases: either $f_1 \leq s_x$ or $f_1 > s_x$. In the first case, $s_1 < f_1 \leq s_x < f_x$, we have that $a_1$ and $a_x$ are disjoint and therefore compatible, so we could build a better solution than $O$ by adding $a_1$ to it; so this case cannot happen. We are left with the second case, in which there is overlap because $a_1$ finishes after $a_x$ starts (but before $a_x$ finishes, by hypothesis that $a_1$ is first to finish in $S$). Since $a_x$ is first to finish in $O$ and no two activities in $O$ overlap, then no activity in $O$ occurs before $a_x$, thus no activity in $O$ overlaps with $a_1$. Thus we could build another equally good optimal solution by substituting $a_1$ for $a_x$ in $O$. Therefore there will always exist an optimal solution that includes $a_1$, QED. This tells us that the greedy strategy works well for this problem.

The general pattern for greedy algorithms is:

1. Cast the problem as one where we make a (greedy) choice and are then left with *just one* smaller problem to solve.

2. Prove that the greedy choice is always part of an optimal solution.

3. Prove that there's optimal substructure, i.e. that the greedy choice plus an optimal solution of the subproblem yields an optimal solution for the overall problem.

Because both greedy algorithms and dynamic programming exploit optimal substructure, one may get confused as to which of the two techniques to apply: using dynamic programming where a greedy algorithm suffices is wasteful, whereas using a greedy algorithm where dynamic programming is required will give the wrong answer. Here is an example of the latter situation.

The knapsack problem, of which a bewildering array of variations have been studied, can be described as follows (this will be the "0-1 knapsack" flavour). A thief has a knapsack of finite carrying capacity and, having

broken into a shop, must choose which items to take with him, up to a maximum weight $W$. Each item $i$ has a weight $w_i$, which is an integer, and a value $v_i$. The goal of the thief is to select a subset of the available items that maximizes the total value while keeping the total weight below the carrying capacity $W$.

This problem has optimal substructure because, with the usual cut-and-paste argument, it is easy to show that, having picked one item $i$ that belongs to the optimal subset, the overall optimal solution is obtained by adding to that item the optimal solution to the 0-1 knapsack problem with the remaining items and with a knapsack of capacity $W - w_i$.

The greedy strategy might be to pick the item of greatest value. But we can easily prove with a counterexample that this won't lead to the optimal solution. Consider a carrying capacity of $W = 18$ kg, an item 1 of weight $w_1 = 10$ kg and value $v_1 = 101$ £ and two items 2 and 3 each of weight 9 kg and value 100 £. Following the stated strategy, the thief would take item 1 and would not have space for anything else, for a total value of 101 £, whereas the optimal solution is to take items 2 and 3 and go away with 200 £.

A perhaps smarter greedy strategy might be to pick the item with the highest £/kg ratio. This would have worked in the previous example but here too we can show it doesn't in general.

**Exercise 24**
Provide a small counterexample that proves that the greedy strategy of choosing the item with the highest £/kg ratio is not guaranteed to yield the optimal solution.

It can be shown, however, that the greedy strategy of choosing the item with the highest £/kg ratio is optimal for the fractional (as opposed to 0-1) knapsack problem, in which the thief can take an arbitrary amount, between 0 and 100%, of each available item[4].

## 3.3 Overview of other strategies

This course is too short to deal with every possible strategy at the same level of detail as we did for dynamic programming and greedy algorithms. The rest of this chapter is therefore just an overview, meant essentially as hints for possible ways to solve a new problem. In a professional situation,

[4]Works with gold dust but not with plasma TVs.

reach for your textbook to find worked examples (and, sometimes, theory) for most of these approaches.

### 3.3.1 Recognize a variant on a known problem

This obviously makes sense! But there can be real inventiveness in seeing how a known solution to one problem can be used to solve the essentially tricky part of another. The Graham Scan method for finding a convex hull (described in the second half of the course), which uses as a sub-algorithm a particularly efficient way of comparing the relative positions of two vectors, is an illustration of this.

### 3.3.2 Reduce to a simpler problem

Reducing a problem to a smaller one tends to go hand in hand with inductive proofs of the correctness of an algorithm. Almost all the examples of recursive functions you have ever seen are illustrations of this approach. In terms of planning an algorithm, it amounts to the insight that it is not necessary to invent a scheme that solves a whole problem all in one step—just a scheme that is guaranteed to make non-trivial progress.

Quicksort (section 2.10), in which you sort an array by splitting it into two smaller arrays and sorting these on their own, is an example of this technique.

This method is closely related to the one described in the next section.

### 3.3.3 Divide and conquer

**Textbook**

Study chapter 4 in CLRS3.

This is one of the most important ways in which algorithms have been developed. It suggests that a problem can sometimes be solved in three steps:

1. **Divide:** If the particular instance of the problem that is presented is very small, then solve it by brute force. Otherwise divide the problem into two (rarely more) parts. To keep the recursion balanced it is usually best to make the sub-problems have similar size.

2. **Conquer:** Use recursion to solve the smaller problems.

3. **Combine:** Create a solution to the final problem by using information from the solution of the smaller problems.

This approach is similar to the one described in the previous section but distinct in so far as we use an explicit recombination step.

In the most common and useful cases, both the dividing and combining stages will have linear cost in terms of the problem size—certainly we expect them to be much easier tasks to perform than the original problem seemed to be. Mergesort (section 2.9) provides a classical example of this approach.

### 3.3.4 Backtracking

If the algorithm you require involves a search, it may be that backtracking is what is needed. This splits the conceptual design of the search procedure into two parts: the first just ploughs ahead and investigates what it thinks is the most sensible path to explore, while the second backtracks when needed. The first part will occasionally reach a dead end and this is where the backtracking part comes in: having kept extra information about the choices made by the first part, it unwinds all calculations back to the most recent choice point and then resumes the search down another path. The Prolog language makes an institution of this way of designing code. The method is of great use in many graph-related problems.

### 3.3.5 The MM method

This approach is perhaps a little frivolous, but effective all the same. It is related to the well known scheme of giving a million monkeys a million typewriters for a million years (the MM Method) and waiting for a Shakespeare play to be written. What you do is give your problem to a group of students (no disrespect intended or implied) and wait a few months. It is quite likely they will come up with a solution that any individual is unlikely to find. Ross Anderson once did this by setting a Tripos exam question on the problem and then writing up the edited results, with credit to the candidates, as an academic paper[5].

Sometimes a variant of this approach is automated: by systematically trying ever increasing sequences of machine instructions, one may eventually find one that has the desired behaviour. This method was once applied to the following C function:

[5]Ross Anderson, "How to cheat at the lottery (or, Massively parallel requirements engineering)", *Proc. Annual Computer Security Applications Conference, Phoenix, AZ*, 1999.

```
int sign(int x) {
    if (x < 0) return -1;
    if (x > 0) return  1;
    return 0;
}
```

The resulting code for the i386 architecture was 3 instructions excluding the return, and for the m68000 it was 4 instructions.

In software testing, this method is the foundation for *fuzzing*: throw lots of random data at the program and see if it crashes or violates its internal assertions.

### 3.3.6 Look for wasted work in a simple method

It can be productive to start by designing a simple algorithm to solve a problem, and then analyze it to the extent that the critically costly parts of it can be identified. It may then be clear that, even if the algorithm is not optimal, it is good enough for your needs; or it may be possible to invent techniques that explicitly attack its weaknesses. You may view under this light the various elaborate ways of ensuring that binary search trees are kept well balanced (sections 4.5 and 4.6).

### 3.3.7 Seek a formal mathematical lower bound

The process of establishing a proof that some task must take at least a certain amount of time can sometimes lead to insight into how an algorithm attaining that bound might be constructed—we did something similar with Selectsort (section 2.6). A properly proved lower bound can also prevent wasted time seeking improvement where none is possible.

# Chapter 4

# Data structures

**Chapter contents**

Primitive data structures. Abstract data types. Pointers, stacks, queues, lists, trees. Hash tables. Binary search trees. Red-black trees. B-trees. Priority queues and heaps.
Expected coverage: about 5 lectures.
Study 6, 10, 11, 12, 13, 18 in CLRS3.

Typical programming languages such as C or Java provide primitive data types such as integers, reals and boolean values. They allow these to be organized into arrays[1] which generally have a statically determined size. It is also common to provide for record data types, where an instance of the type contains a number of components, or possibly pointers to other data. C, in particular, allows the user to work with a fairly low-level idea[2] of a pointer to a piece of data. In this course a "data structure" will be implemented in terms of these language-level constructs, but will always be thought of in association with a collection of operations that can be performed with it and a number of consistency conditions which must always hold. One example of this would be the structure "sorted vector" which might be thought of as just a normal array of numbers but subject to the extra constraint that the numbers must be in ascending

[1] Which we have already used informally from the start.

[2] A C pointer is essentially a memory address, and it's "low level" in the sense that you can increment it to look at what's at the next address, even though nothing guarantees that it's another item of the appropriate type, or even that it's a memory address that you are allowed to read. A pointer in a higher level language might instead only give you access to the item being pointed to, without facilities for accessing adjacent portions of memory.

order. Having such a data structure may make some operations (for instance finding the largest, smallest and median numbers present) easier, but setting up and preserving the constraint (in that case ensuring that the numbers are sorted) may involve work.

Frequently, the construction of an algorithm involves the design of data structures that provide natural and efficient support for the most important steps used in the algorithm, and these data structures then call for further code design for the implementation of other necessary but less frequently performed operations.

## 4.1 Implementing data structures

This section introduces some fundamental data types and their machine representation. Variants of all of these will be used repeatedly as the basis for more elaborate structures.

**Textbook**

Study chapter 10 in CLRS3.

### 4.1.1 Machine data types, arrays, records and pointers

It first makes sense to agree that boolean values, characters, integers and real numbers will exist in any useful computer environment. At the level of abstraction used in this course it will generally be assumed that integer arithmetic never overflows, that floating point arithmetic can be done as fast as integer work and that rounding errors do not exist. There are enough hard problems to worry about without having to face up to the exact limitations on arithmetic that real hardware tends to impose!

The so-called "procedural" programming languages provide for arrays of these primitive types: an array is an ordered collection of a predefined number of elements of the same type, where an integer index can be used to select a particular element of the array, with the access taking unit time. For the moment it is only necessary to consider one-dimensional arrays.

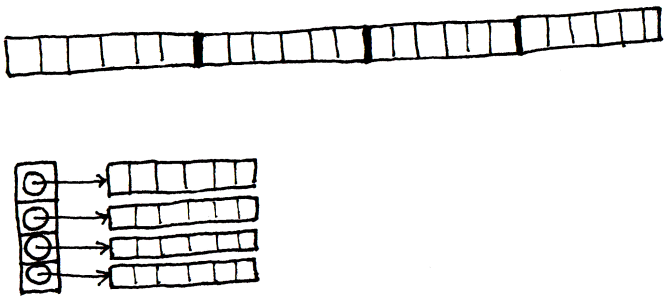
It will also be supposed that one can declare record data types: a record is a collection of a predefined number of elements of potentially different types and each element, normally known as a *field* of the record, is usually referred to not by a positional index but by a field name. Where

records are dynamically allocated (as is frequently the case), we assume that some mechanism is provided for allocating new instances of records and (where appropriate) getting rid of unwanted ones.

The introduction of dynamically allocated records naturally introduces the use of pointers. While a statically allocated object already has a name within the program, a dynamically allocated object doesn't: the programmer needs some kind of handle to refer to the dynamically allocated object and to distinguish it from all other dynamically allocated objects of the same type. A pointer provides such a handle. At the machine level, it is simply the memory address of the object. In a typed system, knowing the type of object that the pointer refers to also says, among other things, how large an area of memory is logically being pointed at; whereas, at the level of machine code, all the pointer says is where that area starts.

This course will not concern itself much about type security (despite the importance of that discipline in keeping whole programs self-consistent), provided that the proof of an algorithm guarantees that all operations performed on data are proper.

It is common to use some form of boxes-and-arrows drawing to represent dynamically allocated objects and the pointers between them. For lack of a universally accepted name, we shall refer to such graphical representations as **records-and-pointers diagrams**. Roughly speaking, a basic machine data type is drawn as a rectangle; a record is drawn as a rectangle containing other shapes (its fields); and a pointer is represented as a circle, out of which emanates an arrow to the dynamically allocated record being pointed to, or an X (or sometimes a "ground connection") to represent a null pointer. For example, you might represent a two-dimensional (2D) array as a 1D-array of 1D-arrays, or as a 1D-array of pointers to 1D-arrays (see section 4.1.2 next) and once you represent them as a records and pointers diagram then the differences become obvious: which of the two is more space-efficient? Which of the two is fastest in accessing a given element? Which of the two makes it easier to swap two rows? Which of the two makes it easier to swap two columns? Which of the two could most efficiently represent the lines of a piece of text? And so forth.



It is worth remembering that the records-and-pointers diagram is a pretty low level representation, but not quite as low level as the actual machine view of the world. At the lowest level, every box on the diagram is allocated somewhere in memory, but without any guarantee that items that appear next to each other in the diagram are anywhere near each other, or even in the same order, in the memory layout. At the level of abstraction of this course we shall generally ignore such details.

### 4.1.2 Vectors and matrices

Some authors use the terms "vector" and "array" interchangeably. Others make the subtle distinction that an array is simply a raw low-level type provided natively by the programming language, while a vector is an abstract data type (section 4.2) with methods and properties. We lean towards the second interpretation but won't be very pedantic on this.

A vector supports two basic operations: the first operation (read) takes an integer index and returns a value. The second operation (write) takes an index and a new value and updates the vector. When a vector is created, its size will be given and only index values inside that pre-specified range will be valid. Furthermore it will only be legal to read a value after it has been set—i.e. a freshly created vector will not have any automatically defined initial contents. Even something this simple can have several different possible implementations.

At this stage in the course we will just think about implementing vectors as blocks of memory where the index value is added to the base address of the vector to get the address of the cell wanted. Note that vectors of arbitrary objects can be handled by multiplying the index value by the size of the objects to get the physical offset of an item in memory with respect to the base, *i.e.* the address of the 0-th element.

As we said in passing in the previous section, there are two simple ways of representing two-dimensional (and indeed arbitrary multi-dimensional) matrices. The first takes the view that an $n \times m$ matrix can be implemented as an array with $n$ items, where each item is an array

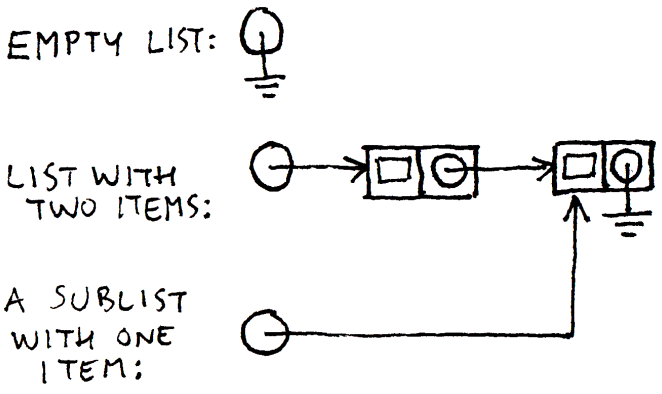
of length $m$. The other representation starts with an array of length $n$ which has as its elements the *addresses* of other arrays of length $m$. One of these representations needs a multiplication (by $m$) for every access, the other an additional memory access. Although there will only be a constant factor between these costs, at this low level it may (just about) matter; but which works better may also depend on the exact nature of the hardware involved[3].

**Exercise 25**
Draw the memory layout of these two representations for a 3×5 matrix, pointing out where element (1,2) would be in each case.

There is scope for wondering about whether a matrix should be stored by rows or by columns (for large matrices and particular applications this may have a big effect on the behaviour of virtual memory systems), and how special cases such as boolean matrices, symmetric matrices and sparse matrices should be represented.

### 4.1.3 Simple lists and doubly-linked lists

A simple and natural implementation of lists is in terms of a record structure.




[3]Note also that the multiplication by $m$ may be performed very quickly with just a shift if $m$ is a power of 2. In cases where speed is of paramount importance, aligning the array rows on such boundaries (which implies increasing the row size to the next power of 2) may be beneficial even at the cost of some wasted memory.

In the records-and-pointers diagrams above, we clearly see that the list is like a little train with zero or more wagons (carriages), each of which holds one list value (the payload of the wagon) and a pointer to rest of the list (i.e. to the next wagon[4], if there is one). In C one might write

```
0 struct ListWagon {
1     int payload; /* We just do lists of integers here */
2     struct ListWagon *next; /* Pointer to the next wagon, if any */
3 };
```

where all lists are represented as pointers to `ListWagon` items. In C it would be very natural to use the special `NULL` pointer to stand for an empty list. We have not shown code to allocate and access lists here.

In other languages, including Java, the analogous declaration would hide the pointers:

```
0 class ListWagon {
1     int payload;
2     ListWagon next; /* The next wagon looks nested, but isn't really. */
3 };
```

There is a subtlety here: if pointers are hidden, how do you represent lists (as opposed to wagons)? Can we still maintain a clear distinction between lists and list wagons? And what is the sensible way of representing an empty list?

**Exercise 26**
Show how to declare a variable of type list in the C case and then in the Java case. Show how to represent the empty list in the Java case. Check that this value (empty list) can be assigned to the variable you declared earlier.

**Exercise 27**
As a programmer, do you notice any uncomfortable issues with your Java definition of a list? *(Requires some thought and O-O flair.)*

[4]You might rightfully observe that it would be more proper to say "to a train with one fewer wagon": anything pointed to by a pointer-to-ListWagon is a train. Congratulations—you are thinking like a proper computer scientist, and you seem to have got the hang of recursion. Read on and do the exercises.

A different but actually isomorphic view will store lists in an array. The items in the array will be similar to the C `ListWagon` record structure above, but the `next` field will just contain an integer. An empty list will be represented by the value zero, while any non-zero integer will be treated as the index into the array where the record with the two components of a non-empty list can be found. Note that there is no need for parts of a list to live in the array in any especially neat order—several lists can be interleaved in the array without that being visible to users of the abstract data type[5]. In fact the array in this case is roughly equivalent to the whole memory in the case of the C `ListWagon`.

If it can be arranged that the data used to represent the `payload` and `next` components of a non-empty list be the same size (for instance both might be held as 32-bit values) then the array might be just an array of storage units of that size. Now, if a list somehow gets allocated in this array so that successive items in it are in consecutive array locations, it seems that about half the storage space is being wasted with the `next` pointers. There have been implementations of lists that try to avoid that by storing a non-empty list as a payload element plus a boolean flag (which takes one bit[6]) with that flag indicating if the next item stored in the array is a pointer to the rest of the list (as usual) or is in fact itself the rest of the list (corresponding to the list elements having been laid out neatly in consecutive storage units).

**Exercise 28**
Draw a picture of the compact representation of a list described in the notes.

The variations on representing lists are described here both because lists are important and widely-used data structures, and because it is instructive to see how even a simple-looking structure may have a number of different implementations with different space/time/convenience trade-offs.

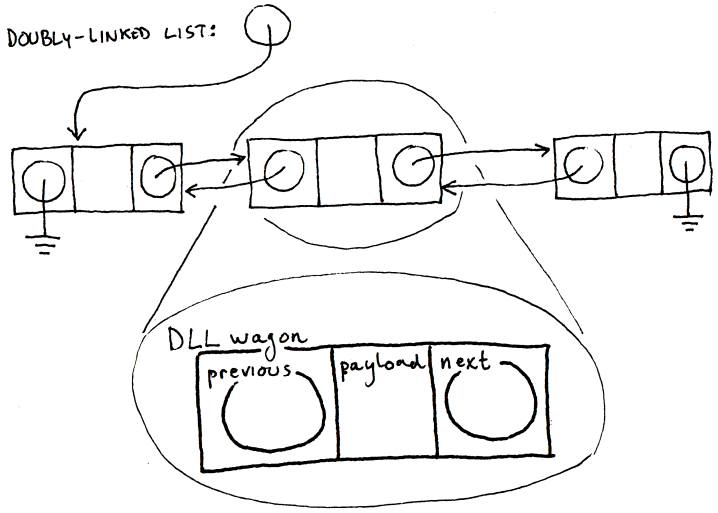
The links in lists make it easy to splice items out from the middle of lists or add new ones. Lists provide one natural implementation of stacks (see section 4.2.1), and are the data structure of choice in many places where flexible representation of variable amounts of data is wanted.

A feature of lists is that, from one item, you can progress along the list in one direction very easily; but, once you have taken the `next` of a list,

[5] Cfr. section 4.2.

[6] And thereby slightly reduces the space available for the payload.

there is no way of returning (unless you independently remember where the original head of your list was). To make it possible to traverse a list in both directions one might define a new type called Doubly Linked List (DLL) in which each wagon had both a `next` and a `previous` pointer.




The following equation

```
w.next.previous == w
```

would hold for every wagon `w` except the last, while the following equation

```
w.previous.next == w
```

would hold for every wagon `w` except the first[7]. Manufacturing a DLL (and updating the pointers in it) is slightly more delicate than working with ordinary uni-directional lists. It is normally necessary to go through an intermediate internal stage where the conditions of being a true DLL are violated in the process of filling in both forward and backwards pointers.

[7]Yet another variant would be the circular doubly linked list, in which these equations would hold for all elements without exceptions. You shall use circular doubly linked lists when you implement Fibonacci heaps in the second half of the course.

### 4.1.4 Graphs

We won't give a formal definition of a graph until the second half of the course, but you probably already have a reasonable mental image of a bunch of nodes (vertices) with arrows (edges) between them. If a graph has $n$ vertices then it can be represented by an $n \times n$ "adjacency matrix", which is a boolean matrix with entry $g_{ij}$ true if and only if the graph contains an edge running from vertex $i$ to vertex $j$. If the edges carry data (for instance the graph might represent an electrical network and we might need to represent the impedance of the component on each edge) then the cells of the matrix might hold, say, complex numbers (or whatever else were appropriate for the application) instead of booleans, with some special value reserved to mean "no link".

An alternative representation would represent each vertex by an integer, and have a vector such that element $i$ in the vector holds the head of a list (an "adjacency list") of all the vertices connected directly to edges radiating from vertex $i$.

The two representations clearly contain the same information, but they do not make it equally easily available. For a graph with only a few edges attached to each vertex, the list-based version may be more compact, and it certainly makes it easy to find a vertex's neighbours, while the matrix form gives instant responses to queries about whether a random pair of vertices are joined, and can be more compact (especially when there are very many edges, and if the bit-array is stored in packed form to make full use of machine words). We shall have much more to say about graphs later in the course.

## 4.2 Abstract data types

When designing data structures and algorithms it is desirable to avoid making decisions based on the accident of how you first sketch out a piece of code. All design should be motivated by the explicit needs of the application. The idea of an Abstract Data Type (ADT) is to support this[8]. The specification of an ADT is a list of the operations that may be performed on it, together with the identities (invariants) that they satisfy. This specification does *not* show how to implement anything in terms of any simpler data types. The user of an ADT is expected to view this specification as the complete description of how the data type and its associated functions will behave—no other way of interrogating

[8]The idea is generally considered good for program maintainability as well, but that is not the primary concern of this particular course.

or modifying data is available, and the response to any circumstances not covered explicitly in the specification is deemed undefined.

In Java, the idea of the Abstract Data Type can be expressed with the `interface` language construct, which looks like a class with all the so-called "signatures" of the class methods (i.e. the types of the input and output parameters) but without any implementations. You can't instantiate objects from an interface; you must first derive a genuine class from the interface and then instantiate objects from the class. And you can derive several classes from the same interface, each of which implements the interface in a different way—that's the whole point. In the rest of this chapter I shall describe ADTs with pseudocode resembling Java interfaces. One thing that is missing from the Java `interface` construct, however, is a formal way to specify the invariants that the ADT satisfies; on the other hand, at the level of this course we won't even attempt to provide a formal definition of the semantics of the data type through invariants, so I shall informally just resort to comments in the pseudocode.

Using the practice we already introduced when describing sorting algorithms, each method will be tagged with a possibly empty *precondition*[9] (something that must be true before you invoke the method, otherwise it won't work) then with an imperative description (labelled *behaviour*) of what the method must do and finally with a possibly empty *postcondition* (something that the method promises will be true after its execution, provided that the precondition was true when you invoked it).

Examples given later in this course should illustrate that making an ADT out of even quite simple operations can sometimes free one from enough preconceptions to allow the invention of amazingly varied collections of implementations.

### 4.2.1 The Stack abstract data type

Let us now introduce the Abstract Data Type for a Stack: the standard mental image is that of a pile of plates in your college buttery. The distinguishing feature of this structure is that the only easily accessible item is the one on top of the stack. For this reason this data structure is also sometimes indicated as LIFO, which stands for "Last in, first out".

[9] Some programming languages allow you to enter such invariants in the code not just as comments but as *assertions*—a brilliant feature that programmers should learn to use more frequently.

```
0  ADT Stack {
1    boolean isEmpty();
2    // BEHAVIOUR: return true iff the structure is empty.
3
4    void push(item x);
5    // BEHAVIOUR: add element <x> to the top of the stack.
6    // POSTCONDITION: isEmpty() == false.
7    // POSTCONDITION: top() == x
8
9    item pop();
10   // PRECONDITION: isEmpty() == false.
11   // BEHAVIOUR: return the element on top of the stack.
12   // As a side effect, remove it from the stack.
13
14
15   item top();
16   // PRECONDITION: isEmpty() == false.
17   // BEHAVIOUR: Return the element on top of the stack (without removing it).
18 }
```

In the ADT spirit of specifying the semantics of the data structure using invariants, we might also add that, for each stack `s` and for each item `x`, after the following two-operation sequence

```
0  s.push(x)
1  s.pop()
```

the return value of the second statement is `x` and the stack `s` "is the same as before"; but there are technical problems in expressing this correctly and unambiguously using the above notation, so we won't try. The idea here is that the definition of an ADT should collect all the essential details and assumptions about how a structure must behave (although the expectations about common patterns of use and performance requirements are generally kept separate). It is then possible to look for different ways of implementing the ADT in terms of lower level data structures.

Observe that, in the Stack type defined above, there is no description of what happens if a user tries to compute `top()` when `isEmpty()` is `true`, i.e. when the precondition of the method is violated. The outcome is therefore undefined, and an implementation would be entitled to do *anything* in such a case—maybe some garbage value would get returned without any mention of the problem, maybe an error would get reported or perhaps the computer would crash its operating system and delete all your files. If an ADT wants exceptional cases to be detected and reported, it must specify this just as clearly as it specifies all other behaviour[10]

[10]For example, it would be a great idea for the ADT to specify that all its methods

The stack ADT given above does not make allowance for the `push` operation to fail—although, on any real computer with finite memory, it must be possible to do enough successive pushes to exhaust some resource. This limitation of a practical realization of an ADT is not deemed a failure to implement the ADT properly: at the level of abstraction of this introductory algorithms course, we do not really admit to the existence of resource limits!

There can be various different implementations of the Stack data type, but two are especially simple and commonly used. The first represents the stack as a combination of an array and an index (pointing to the "top of stack", or TOS). The `push` operation writes a value into the array and increments the index[11], while `pop` does the converse. The second representation of stacks is as linked lists, where pushing an item just adds an extra cell to the front of a list, and popping removes it. In both cases the `push` and `pop` operations work by modifying stacks in place, so (unlike what might happen in a functional language such as ML) after invoking either of them the original stack is no longer available.

Stacks are useful in the most diverse situations. The page description language PostScript is actually, as you may know, a programming language organized around a stack (and the same is true of Forth, which may have been an inspiration). Essentially in such languages the program is a string of tokens that include operands and operators. During program execution, any operands are pushed on the stack; operators, instead, pop from the stack the operands they require, do their business on them and finally push the result back on the stack. For example, the program

```
3 12 add 4 mul 2 sub
```

computes $(3 + 12) \times 4 - 2$ and leaves the result, 58, on the stack. This way of writing expressions is called Reverse Polish Notation and one of its attractions is that it makes parentheses unnecessary (at the cost of having to reorder the expression and making it somewhat less legible).

**Exercise 29**
Invent (or should I say "rediscover"?) a linear-time algorithm to convert an infix expression such as
`(3+12)*4 - 2`
into a postfix one without parentheses such as
`3 12 + 4 * 2 -`.
By the way, would the reverse exercise have been easier or harder?

could raise a "Precondition violation" exception when appropriate.

[11] Note that stacks growing in the reverse direction are also plausible and indeed frequent. (Why is that?)

### 4.2.2 The List abstract data type

We spoke of linked lists as a basic low level building block in section 4.1.3, but here we speak of the ADT, which we define by specifying the operations that it must support. Note how you should be able to implement the List ADT with any of the implementations described in 4.1.3 (wagons and pointers, arrays and so forth) although sometimes the optimizations will get in the way (e.g. packed arrays). The List version defined here will allow for the possibility of re-directing links in the list. A really full and proper definition of the ADT would need to say something rather careful about when parts of lists are really the same (so that altering one alters the other) and when they are similar in structure and values but distinct[12]. Such issues will be ducked for now but must be clarified before writing programs, or they risk becoming the source of spectacular bugs.

```
0  ADT List {
1    boolean isEmpty();
2    // BEHAVIOUR: Return true iff the structure is empty.
3
4    item head(); // NB: Lisp people might call this ‘‘car’’.
5    // PRECONDITION: isEmpty() == false
6    // BEHAVIOUR: return the first element of the list (without removing it).
7
8    void prepend(item x); // NB: Lisp people might call this ‘‘cons’’.
9    // BEHAVIOUR: add element <x> to the beginning of the list.
10   // POSTCONDITION: isEmpty() == false
11   // POSTCONDITION: head() == x
12
13   List tail(); // NB: Lisp people might call this ‘‘cdr’’.
14   // PRECONDITION: isEmpty() == false
15   // BEHAVIOUR: return the list of all the elements except the first (without
16   // removing it).
17
18   void setTail(List newTail);
19   // PRECONDITION: isEmpty() == false
20   // BEHAVIOUR: replace the tail of this list with <newTail>.
21 }
```

You may note that the List type is very similar to the Stack type mentioned earlier. In some applications it might be useful to have a variant on the List data type that supported a `setHead()` operation to update list contents (as well as chaining) in place, or an `isEqualTo()`

[12]For example, does the `tail()` method return a copy of the rest of the list or a pointer to it? And similarly for `setTail()`.

test. Applications of lists that do not need `setTail()` may be able to use different implementations of lists.

### 4.2.3 The Queue and Deque abstract data types

In the Stack ADT, the item removed by the `pop()` operation was the most recent one added by `push()`. A Queue[13] is in most respects similar to a stack, but the rules are changed so that the item accessed by `top()` and removed by `pop()` will be the oldest one inserted by `push()` (we shall rename these operations to avoid confusion). Even if finding a neat way of expressing this in a mathematical description of the Queue ADT may be a challenge, the idea is simple. The above description suggests that stacks and queues will have very similar interfaces. It is sometimes possible to take an algorithm that uses a stack and obtain an interesting variant by using a queue instead; and vice-versa.

```
0  ADT Queue {
1    boolean isEmpty();
2    // BEHAVIOUR: return true iff the structure is empty.
3
4    void put(item x);
5    // BEHAVIOUR: insert element <x> at the end of the queue.
6    // POSTCONDITION: isEmpty() == false
7
8    item get();
9    // PRECONDITION: isEmpty() == false
10   // BEHAVIOUR: return the first element of the queue, removing it
11   // from the queue.
12
13   item first();
14   // PRECONDITION: isEmpty() == false
15   // BEHAVIOUR: return the first element of the queue, without removing it.
16
17
18 }
```

A variant is the perversely-spelt Deque (double-ended queue), which is accessible from both ends both for insertions and extractions and therefore allows four operations:

```
0  ADT Deque {
1    boolean isEmpty();
2
3    void putFront(item x);
```

[13] Sometimes referred to as a FIFO, which stands for "First In, First Out".

```
4   void putRear(item x);
5   // POSTCONDITION for both: isEmpty() == false
6
7   item getFront();
8   item getRear();
9   // PRECONDITION for both: isEmpty() == false
10 }
```

The Stack and Queue may be seen as subcases of the Deque in which only one `put` and one `get` (as opposed to two of each) are enabled.

### 4.2.4 The Dictionary abstract data type

We are concerned with the kind of data structure that associates keys (e.g. a word, or the name of a person) with values (e.g. the word's definition, or the person's postal address) and allows you to look up the relevant value if you supply the key. Note that, within the dictionary, the mapping between keys and values is a function[14]: you cannot have different values associated with the same key. For generality we assume that keys are of type `Key` and that values are of type `Value`.

```
0  ADT Dictionary {
1   void set(Key k, Value v);
2   // BEHAVIOUR: store the given (<k>, <v>) pair in the dictionary.
3   // If a pair with the same <k> had already been stored, the old
4   // value is overwritten and lost.
5   // POSTCONDITION: get(k) == v
6
7   Value get(Key k);
8   // PRECONDITION: a pair with the sought key <k> is in the dictionary.
9   // BEHAVIOUR: return the value associated with the supplied <k>,
10  // without removing it from the dictionary.
11
12  void delete(Key k);
13  // PRECONDITION: a pair with the given key <k> has already been inserted.
14  // BEHAVIOUR: remove from the dictionary the key-value pair indexed by
15  // the given <k>.
16 }
```

Observe that this simple version of a dictionary does not provide a way of asking if some key is in use, and it does not mention anything about the number of items that can be stored in a dictionary. Practical implementations may concern themselves with both these issues.

[14] As opposed to a generic relation. In other words, only one arrow goes out of each source element.

Dictionaries are also variously known as Maps, Tables, Associative arrays or Symbol tables.

Probably the most important special case of a dictionary is when the keys are known to be drawn from the set of integers in the range $0..n$ for some modest $n$. In that case the dictionary can be modelled directly by a simple vector, and both `set()` and `get()` operations have unit cost. If the key values come from some other integer range (say $a..b$) then subtracting $a$ from key values gives a suitable index for use with a vector. This simple and efficient strategy is known as **direct addressing**. If the number of keys that are actually used is much smaller than the number $b-a$ of items in the range that the keys lie in, direct addressing becomes extremely inefficient (or even infeasible) in space, even though its time performance remains optimal, namely $O(1)$ worst-case.

**Exercise 30**
How would you deal efficiently with the case in which the keys are English words? *(There are several possible schemes of various complexity that would all make acceptable answers provided you justified your solution.)*

For sparse dictionaries one could try holding the data in a list, where each item in the list could be a record storing a key-value pair. The `get()` function can just scan along the list, searching for the key that is wanted; if the desired key is not found, the behaviour of the function is undefined[15]. But now there are several options for the `set()` function. The first natural one just sticks a new key-value pair at the front of the list, allowing `get()` to be coded so as to retrieve the first value that it finds. The second one would scan the list and, if a key was already present, it would update the associated value in place. If the required key was not present it would have to be added.

**Exercise 31**
Should the new key-value pair added by `set()` be added at the start or the end of the list? Or elsewhere?

Since in this second case duplicate keys are avoided, the order in which

[15] In a practical implementation we would do well to define what happens: raising a "key not found" exception would be a sensible option. A less clean but commonly used alternative might be to return a special value meaning "not found".

items in the list are kept will not affect the correctness of the data type, and so it would be legal (if not always useful) to make arbitrary permutations of the list each time it was touched.

If one assumes that the keys passed to `get()` are randomly selected and uniformly distributed over the complete set of keys used, the linked list representation calls for a scan down (an average of) half the length of the list. For the version that always adds a new key-value pair at the head of the list, this cost increases without limit as values are changed. The other version keeps that cost down but has to scan the list when performing `set()` operations as well as `get()`s.

To try to get rid of some of the overhead of the linked list representation, keep the idea of storing a dictionary as a bunch of key-value pairs but now put these in an array rather than a linked list. Now suppose that the keys used are ones that support an ordering, and sort the array on that basis. Of course there now arise questions about how to do the sorting and what happens when a new key is mentioned for the first time—but here we concentrate on the data retrieval part of the process. Instead of a linear search as was needed with lists, we can now probe the middle element of the array and, by comparing the key there with the one we are seeking, can isolate the information we need in one or the other half of the array. If the comparison has unit cost, the time needed for a complete look-up in an array with $n$ cells will satisfy

$$f(n) = f(n/2) + k$$

and the solution to this recurrence shows us that the complete search can be done in $\Theta(\lg n)$.

**Exercise 32**
Solve the $f(n) = f(n/2) + k$ recurrence, again with the trick of setting $n = 2^m$.

Another representation of a dictionary that also provides $O(\lg n)$ costs is obtained by building a binary tree, where the tree structure relates very directly to the sequence of comparisons that could be done during binary search in an array. If a tree of $n$ items can be built up with the median key from the whole data set in its root, and each branch is similarly well balanced, the greatest depth of the tree will be around $\lg n$.
Having a linked representation makes it fairly easy to adjust the structure of a tree when new items need to be added, but details of that will be left until section 4.3. Note that, in such a tree, all items in the left sub-tree

come before the root in sorting order, and all those in the right sub-tree come after.

### 4.2.5 The Set abstract data type

There are very many places in the design of larger algorithms where it is necessary to have ways of keeping sets of objects. In different cases, different operations will be important, and finding ways in which various subsets of the possible operations can be best optimized leads to the discussion of a large range of sometimes quite elaborate representations and procedures. We shall cover some of the more important (and more interesting) options in this course.

Until we are more specific about the allowed operations, there isn't much difference between a Set and the Dictionary seen in the previous section: we are still storing key-value pairs, and keys are unique. But now we *are* going to be more specific, and define extra operations for the Set by adding to the basic ones introduced for the Dictionary. We may think of many plausible extensions that are quite independent of each other. Since this is a course on data structures and not on object-oriented programming, in the pseudocode I shall gloss over the finer points of multiple inheritance, mixin classes and diamond diagrams, but I hope the spirit is clear: the idea is that you could form a Set variant that suits your needs by adding almost any combination of the following methods to the Dictionary ADT seen earlier.

Simple variants could just add elementary utilities:

```
0  boolean isEmpty()
1  // BEHAVIOUR: return true iff the structure is empty.
2
3  boolean hasKey(Key x);
4  // BEHAVIOUR: return true iff the set contains a pair keyed by <x>.
5
6  Key chooseAny();
7  // PRECONDITION: isEmpty() == false
8  // BEHAVIOUR: Return the key of an arbitrary item from the set.
```

For a more sophisticated variant, let us introduce the assumption that there exists a total order on the set of keys—something that the Dictionary did not require[16]. We may then meaningfully introduce methods

[16] One might argue that, in mathematics, one of the defining characteristics of a set is that its elements are not ordered. But this does not prevent from making a set out of elements on which an order is defined, nor from asking what is the smallest element in the set, and so forth.

that return the smallest or largest key of the set, or the next largest or next smallest with respect to a given one.

```
0   Key min();
1   // PRECONDITION: isEmpty() == false
2   // BEHAVIOUR: Return the smallest key in the set.
3
4   Key max();
5   // PRECONDITION: isEmpty() == false
6   // BEHAVIOUR: Return the largest key in the set.
7
8   Key predecessor(Key k);
9   // PRECONDITION: hasKey(k) == true
10  // PRECONDITION: min() != k
11  // BEHAVIOUR: Return the largest key in the set that is smaller than <k>.
12
13  Key succcessor(Key k);
14  // PRECONDITION: hasKey(k) == true
15  // PRECONDITION: max() != k
16  // BEHAVIOUR: Return the smallest key in the set that is larger than <k>.
```

Another interesting and sometimes very useful feature is the ability to form a set as the union of two sets. Note how a proper ADT definition would have to be much more careful about specifying whether the original sets are preserved or destroyed by the operation, as well as detailing what to do if the two sets contain pairs with the same key but different values.

```
0   Set unionWith(Set s);
1   // BEHAVIOUR: Change this set to become the set obtained by
2   // forming the union of this set and <s>.
```

The remaining sections in this chapter will describe implementations of specific variations on the Dictionary/Set theme, each with its own distinctive features and trade-offs in terms of supported operations and efficiency in space and time.

## 4.3 Binary search trees

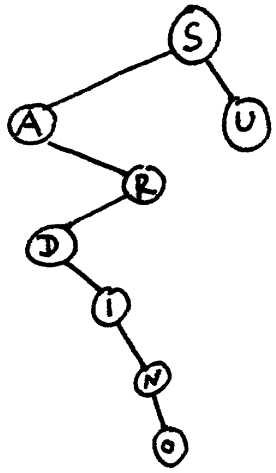
**Textbook**

Study chapter 12 in CLRS3.

A binary search tree requires a key space with a total order and implements the Set variant that also supports the computation of `min()`,

`max()`, `predecessor()` and `successor()`.

Each node of the binary tree will contain an item (consisting of a key-value pair[17]) and pointers to two sub-trees, the left one for all items with keys smaller than the stored one and the right one for all the items with larger keys.




Searching such a tree is simple: just compare the sought key with that of the visited node and, until you find a match, recurse down in the left or right subtree as appropriate. The maximum and minimum values in the tree can be found in the leaf nodes discovered by following all left or right pointers (respectively) from the root.

**Exercise 33**
*(Clever challenge, straight from CLRS3—exercise 12.2-4.)* Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key $k$ in a binary search tree ends up in a leaf. Consider three sets: $A$, the keys to the left of the search path; $B$, the keys on the search path; and $C$, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

To find the successor[18] $s$ of a node $x$ whose key is $k_x$, look in $x$'s

[17] If the value takes up more than a minimal amount of space, it is actually stored elsewhere as "satellite data" and only a pointer is stored together with the key.

[18] The case of the predecessor is analogous, except that left and right must be reversed.

right subtree: if that subtree exists, the successor node $s$ must be in it—otherwise any node in that subtree would have a key between $k_x$ and $k_s$, which contradicts the hypothesis that $s$ is the successor. If the right subtree does not exist, the successor may be higher up in the tree. Go up to the parent, then grand-parent, then great-grandparent and so on until the link goes up-and-right rather than up-and-left, and you will find the successor. If you reach the root before having gone up-and-right, then the node has no successor: its key is the highest in the tree.

**Exercise 34**
Why, in BSTs, does this up-and-right business find the successor? Can you sketch a proof?

**Exercise 35**
*(Important.)* Prove that, in a binary search tree, if node $n$ has two children, then its successor has no left child.

To insert in a tree, one searches to find where the item ought to be and then inserts there. Deleting a leaf node is easy. To delete a non-leaf is harder, and there will be various options available. A non-leaf with only one child can be simply replaced by its child. For a non-leaf with two children, an option is to replace it with its successor (or predecessor—either will work). Then the item for deletion can't have two children (cfr. exercise above) and can thus be deleted in one of the ways already seen; meanwhile, the newly moved up object satisfies the order requirements that keep the tree structure valid.

**Exercise 36**
Prove that this deletion procedure, when applied to a valid binary search tree, always returns a valid binary search tree.

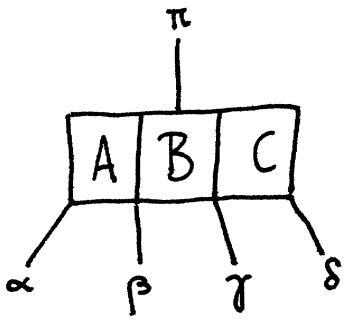
Most of the useful operations on binary search trees (`get()`, `min()`, `max()`, `successor()` and so on) have cost proportional to the depth of the tree. If trees are created by inserting items in random order, they usually end up pretty well balanced, and this cost will be $O(\lg n)$. But the

tree pictured as an example at the beginning of this section, while valid, is instead very unbalanced: it has a depth almost equal to its number of nodes. An easy way to obtain a worst-case BST is to build it by inserting items in ascending order: then the tree degenerates into a linear list of height $n$. It would be nice to be able to re-organize things to prevent that from happening. In fact there are several methods that work, and the trade-offs between them relate to the amount of space and time that will be consumed by the mechanism that keeps things balanced. The next section describes two (related) sensible compromises.

## 4.4 2-3-4 trees

We were not able to guarantee logarithmic performance for Binary Search Trees because we could not be sure they would always be balanced. So let's now discuss a more ingenious alternative, 2-3-4 trees[19]. Practically nobody uses 2-3-4 trees nowadays, but understanding their principles is instructive because it will help us make sense of both Red-Black Trees (which are balanced BSTs based on seemingly arcane rules and axioms) and the much larger B-trees that let you index more data than would physically fit in your RAM. We'll be studying all of these data structures next.

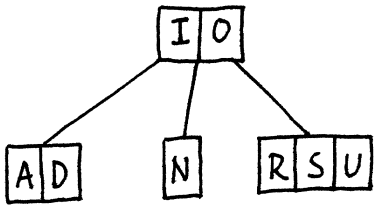
Binary trees had one key and two outgoing pointers in each node. 2-3-4 trees generalize this structure to allow nodes to contain more keys and pointers. Specifically, besides binary nodes (2-nodes) with 2 children and 1 key, they also allow 3-nodes, with three outgoing pointers and therefore 2 keys, and 4-nodes, with 4 outgoing pointers and therefore 3 keys. As with regular binary trees, the pointers are all to subtrees which only contain key values separated by the keys in the parent node: in the following picture, the branch labelled $\beta$ leads to a subtree containing keys $k$ such that "A" $\leq k \leq$ "B".




[19]Note that 2-3-4 trees are one of the very few topics in this course's syllabus that are not discussed in the CLRS3 textbook—so pay special attention.

Searching a 2-3-4 tree is almost as easy as searching a binary tree. Any concern about extra work within each node should be balanced by the realization that, with a larger branching factor, 2-3-4 trees will generally be shallower than pure binary trees.

Inserting into a 2-3-4 node also ends up being fairly easy, and—even better—it turns out that a simple insertion process automatically leads to trees that are always balanced. To insert, search down through the tree looking for where the new item must be added. If an item with the same key is found, just overwrite it[20]—the structure of the tree does not change at all in this trivial case. If you can't find the key, continue *until the bottom level* (never add a new item into a node in the middle of the tree, even if there's space) to reach the place where the new key should be added. If that place is a 2-node or a 3-node, then the item can be stuck in without further ado, upgrading that node to a 3-node or 4-node respectively. If the insertion was going to be into a 4-node, something has to be done to make space for the newcomer. The operation needed is to decompose the 4-node into a pair of 2-nodes before attempting the insertion—this then means that the parent of the original 4-node will gain an extra key (the middle key of the former 4-node) and an extra child, which in turn will be a pain if the parent was itself a 4-node. To ensure that there will always be room for insertion, we apply some foresight: while searching down the tree to find where to make an insertion, if we ever come across a 4-node we split it immediately; thus, by the time we go down and look at its offspring and have our final insertion to perform, we can be certain that there are no 4-nodes in the path from the root to where we are. If the root node gets to be a 4-node, it can be split into three 2-nodes, and this is the only circumstance in which the height of the tree increases.




The key to understanding why 2-3-4 trees remain balanced is the recognition that splitting a 4-node (other than the root) does not alter the length of any path from the root to a leaf of a tree. Splitting the root increases the length of *all* paths by 1. Thus, at all times, all paths through the tree from root to a leaf have the same length. The tree has

[20]Meaning: replace the old value (or pointer to satellite data) with the new.

a branching factor of at least 2 at each level, and so, in a tree with $n$ items, all items will be at worst $\lg n$ down from the root.

Once again, deletions require some more intellectual effort and attention to detail and we don't have time to cover all of that here. The key point is that, to preserve the 2-3-4 structure, deletions may only happen in the bottom layer[21]; therefore, if you must delete an item that resides elsewhere, you must first move things around a bit. Our experience with BSTs suggests swapping the item to be deleted with its successor or predecessor (at least one of which must be in the bottom layer) and then deleting it there. That's a good idea, but it's not the whole story: what if the bottom node you're deleting from was already a 2-node? What would it become? Stay tuned until we talk of B-tree deletions in section 4.6.2, where we shall describe in detail an even more general case.

## 4.5 Red-black trees

**Textbook**

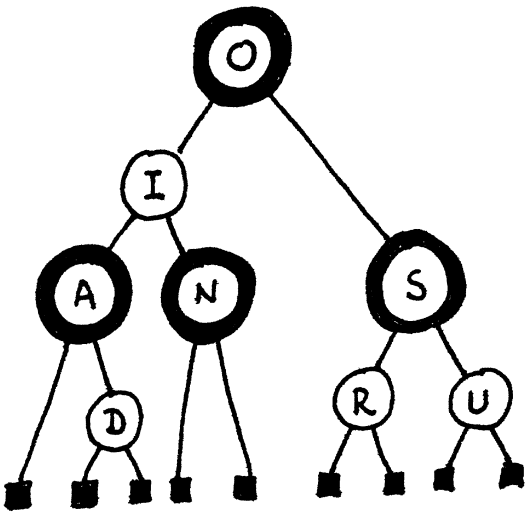
Study chapter 13 in CLRS3.

### 4.5.1 Definition of red-black trees

Red-black trees are special binary search trees that are guaranteed always to be reasonably balanced: no path from the root to a leaf is more than twice as long as any other. They are defined by mystical rules that, in your textbook, seem to come out of the blue. But now, with 2-3-4 trees under our belt, we have the intellectual key to unravel the mystery. Let's have a look.

(Imagine that the nodes with a thin border are red. If we could afford the luxury of colour printing, they would be. Feel free to colour them in.)

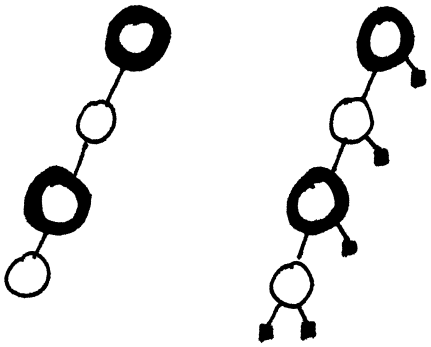
[21] Otherwise, if you deleted a key from a node in a non-bottom layer, such as "O" in the picture above, what would happen to the subtrees hanging off that key to its left ("N") and right ("RSU")?




Formally, a red-black tree can be defined as a binary search tree that satisfies the following five invariants.

1. Every node is either red or black.
2. The root is black.
3. All leaves are black and never contain key-value pairs[22].
4. If a node is red, both its children are black.
5. For each node, all paths from that node to descendent leaves contain the same number of black nodes.

The most important property is clearly the last: the others are only relevant as a framework that allows us to enforce the last property.



[22]Since leaves carry no information, they are sometimes omitted from the drawings; but they are necessary for the consistency of the remaining invariants.

The above formulation sometimes confuses people, leading alert readers wondering whether a linear list of alternating black and red nodes wouldn't match the properties and yet be maximally unbalanced. Or do the rules say anywhere that each node of the binary tree is forced to have two children? The answer is somewhat subtle. No, the rules don't require each node to have two "full" children, meaning children with keys and values (if they did, it would be impossible to have a tree of finite size); however each binary node will have two *pointers* to its children and, if either or both of these children is missing, the corresponding pointer(s) will be null. The subtlety is that we consider all these null pointers to be empty black leaves. You could imagine memory position 0, where null pointers point, to be holding the archetypal black leave; and, since leaves carry no information, we don't need to store distinct ones—an indicator that "there's a black leaf there" is sufficient, and that's precisely what the null pointer does. In practice, people often don't even *draw* the empty black leaves when drawing the Red-Black tree, since their position is obvious and they carry no information. But what matters is that those hidden leaves are there, so the tree above, shaped like a linear list, is not a valid Red-Black tree because the paths from the root to the little black leaves hanging off the side of each node in the chain violate the fifth invariant.

From invariant 4 you see that no path from root to leaf may contain two consecutive red nodes. Therefore, since each path starts with a black root (invariant 2) and ends with a black leaf (invariant 3), the number of red nodes in the path is at most equal to that of the black non-leaf nodes. Since, by invariant 5, the number of black non-leaf nodes in each path from the root to any leaf, say $b$, is the same across the tree, all such paths have a node count between $b$ and $2b$.

**Exercise 37**
What are the smallest and largest possible number of nodes of a red-black tree of height $h$, where the height is the length in edges of the longest path from root to leaf?

It is easy to prove from this that the maximum depth of an $n$-node red-black tree is $O(\lg n)$, which is therefore the time cost of `get()`, `min()`, `max()`, `successor()` and so on. Methods that only inspect the tree, without changing it, are identical to those for the Binary Search Tree.

### 4.5.2 Understanding red-black trees

How did this structure and these invariants come about? With a little thought you will now recognize Red-Black trees as really nothing more than a trick to transport the clever ideas of 2-3-4 trees into the more uniform and convenient realm of pure binary trees. The 2-3-4 trees are easier to understand but less convenient to code, because of all the complication associated with having three different types of nodes and several keys (and therefore comparison points) for each node. The binary red-black trees have complementary advantages and disadvantages.

The idea is that we can represent any 2-3-4 tree as a red-black tree: a black node stands for the "core" (the root, actually) of a regular 2-3-4 node, while red nodes are used as a way of providing extra pointers. Just as 2-3-4 trees have the same number ($k$, say) of nodes from root to each leaf, red-black trees always have $k$ black nodes on any path, and can have from 0 to $k$ red nodes as well. Thus the depth of the new red-black tree is at worst twice that of a 2-3-4 tree. Insertions and node splitting in red-black trees will just have to follow rules equivalent to those that were set up for 2-3-4 trees.

The key to understanding red-black trees is to map out explicitly the isomorphism between them and 2-3-4 trees. So, to set you off in the right direction...

**Exercise 41**
For each of the three possible types of 2-3-4 nodes, draw an isomorphic "node cluster" made of 1, 2 or 3 red-black nodes. The node clusters you produce must:

- Have the same number of keys, incoming links and outgoing links as the corresponding 2-3-4 nodes. as the corresponding 2-3-4 nodes.
- Respect all the red-black rules when composed with other node clusters.

But what about `set()`, i.e. inserting a new item? Finding the correct place in the tree involves an algorithm very similar to that of `get()`, but as we discussed for 2-3-4 trees we can't just insert in the middle of the tree because we also need to preserve the red-black properties, and this can be tricky. There are complicated recipes to follow, based on left and

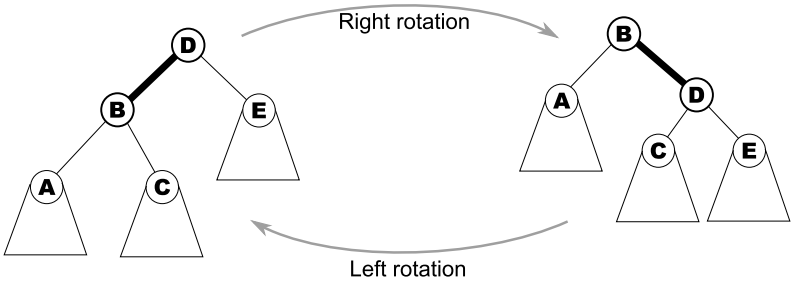
right rotations, for restoring the RBT invariants after an insertion, and we'll look at them next.

First, though, rotations.

### 4.5.3 Rotations

A rotation is a local transformation of a BST (any BST, not necessarily a Red-Black one) that changes the shape of the BST but preserves the BST properties. It is a useful operation for someone who wishes to rebalance a BST. When we say a "local" transformation, we mean that a rotation involves rewriting a small and finite number of pointers. This operation has a constant cost that does not depend on the size of the BST or on the position in the tree of the nodes on which we perform the rotation.

Two types of rotation are possible: a left and a right rotation. Sometimes people speak of a rotation of a node, but really what is being rotated is an edge connecting a pair of nodes. In the diagram below, when moving from the situation on the left to that on the right we perform a *right rotation*, and what is being rotated to the right is the BD edge.




Some authors may speak of a right rotation of the D node (the "parent" among the two nodes), but calling it a right rotation of the BD edge is probably much clearer.

**Exercise 38**
With reference to the rotation diagram in the handout, and to the stupid way of referring to rotations that we don't like, what would a *left* rotation of the D node be instead? (Hint: it would *not* be the one marked as "Left rotation" in the diagram.)

Note how, in the right rotation shown, node D changes from being the parent of B to being its right child. Also, while A remains B's left

child and E remains D's right child, C changes from being B's right child to becoming D's left child. Clearly it could not stay on as B's right child since now B has acquired D as its new right child! And conversely D has lost its left child (formerly B), so there is a convenient empty space there.

The end result is another tree, with a slightly different shape, but still definitely a BST, as you can easily verify. A, and all the other nodes in the subtree rooted at A, was $<$ B, as B's left child, and continues to be so, since it is still B's left child. Similarly, E and all the nodes of E's subtree were $>$ D and still are, as D's right child. As for C and its subtree: it used to be $>$ B, as B's right child, but also $<$ D, by virtue of being within B's subtree with B a left child of D. After the rotation, C and its subtree are still $>$ B, by virtue of having moved into D's subtree, and are $<$ D because they are D's left child. So, with a slight abuse of notation, we can write that A $<$ B $<$ C $<$ D $<$ E holds before and after the rotation.

Note also how, in the right rotation, "B goes up and D goes down". Depending on the heights of the A, C and E subtrees, this may cause the heights of B and D to change. This is why a rotation (or, more likely, a carefully chosen sequence of rotations) may be useful in rebalancing a tree.

One thing that may not be immediately obvious from the diagram is that the rotation does not necessarily have to be performed at the root of the tree: any edge of the BST can be rotated. The top node of the edge being rotated (D in the example above) may well be the child of another parent node, say for example the right child of some higher node P. At the end of the rotation, the new right child of P will be B, the node that has replaced D as the root of the subtree involved in the rotation (or, in other words, the node at the other end of the DB edge being rotated).

### 4.5.4 Implementing red-black trees

If we want to implement the main methods of a red-black-tree class, namely `set()`, `get()` and `delete()`, we must ensure that every invocation of every method preserves the five RBT invariants.

As we already noted, methods that don't change the tree, such as `get()`, are identical to those of a BST (which the RBT also is).

Let's look at insertion next: the `set()` method does change the tree, and indeed in the case of the BST might cause an inbalance (think of inserting the values of an ascending sequence), which would clearly violate the RBT invariants. So we expect the RBT version to be different if it

is going to yield a balanced[23] tree.

In how many ways could an insertion violate the invariants? The new node $n$ is always appended to a parent node $p$, except if the tree was empty (in which case $n$ becomes the root and the only node, and won't violate any invariants so long as we paint it black). If parent $p$ was black, by appending a red $n$ we satisfy all invariants automatically and insertion is concluded. However, if $p$ was red, we are in trouble whatever we do: if we append a black $n$ we have introduced a path with a different black-height from $p$ (compared to the path going to the other child of $p$), violating invariant 5; whereas, if we append a red $n$, we have introduced two consecutive reds, violating invariant 4. Changing the black-height of one path and not the others sounds like a lot of trouble, so let's instead choose always to insert a red $n$ into a red $p$ and fix things up. Sometimes this causes trouble further up the tree, which by then is temporarily not a valid RBT, so we have to be quite careful about our preconditions and invariants if we want to produce a correct algorithm. There are several cases to consider.

The red parent node $p$ in turn has a parent (it must do, because the root can't be red thanks to invariant 2, so the red $p$ can't be the root), which we'll call $g$ for "grandparent", and this $g$ may well have another child $u$, the sibling of $p$ and therefore the "uncle" of $n$. What are the possible relative positions and colours of these nodes? Clearly $g$ is always at the top of this subtree, and has $u$ (if it exists) and $p$ as its children, and $p$ may be the left or the right child of $g$. We have already established that $p$ is red (because if it's black then no invariants are violated by appending a red child to it, so we don't need to do any further fix-up work), which means that $g$ must be black (else the original tree would violate invariant 4, as red node $g$ would have a red child $p$, and two consecutive reds are not allowed). We might be tempted to say that $u$, if it exists, must be red too (else the original tree would violate invariant 5 about all paths from $g$ to descendent leaves having the same number of black nodes), but unfortunately this is not necessarily true.

**Exercise 39**
During RBT insertion, if $p$ is red and $g$ is black, how could $u$ ever possibly be black? How could $p$ and $u$ ever be of different colours? Would that not be an immediate violation of invariant 5?

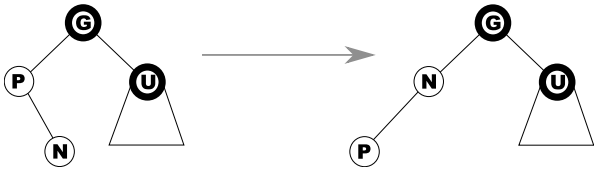
So, starting with the premise that $n$ and $p$ are red and that $g$ is

[23] Not literally: only in the RBT sense that all paths from a node to its descendent leaves have lengths within a factor of 2 of each other.

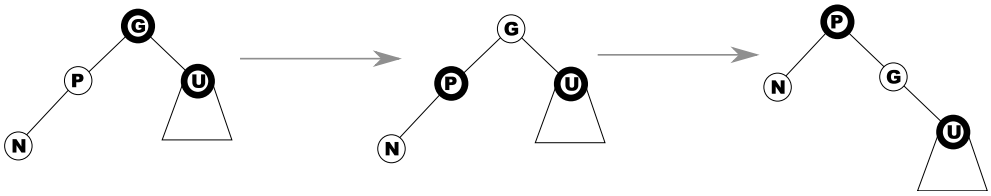
black, we distinguish three cases. Case 1 is when $u$ is red. If $u$ is black or doesn't exist, we have one of the other two cases. If the path $g \to p \to n$ makes a zig-zag (left-right or right-left), it's case 2; if instead it's straight (left-left or right-right), it's case 3. Case 2 can be reduced to case 3 with a rotation.

**Exercise 40**
Draw the three cases by yourself and recreate, without reading, the correct procedure to fix each of them. Then apply it to figure 13.4.(a) of CLRS3, without looking at the rest of the figure.


When in **case 1** ($n, p, u$ red and $g$ black, as above), simply flip the colour of $p$, $u$ and $g$. As a result, the subtree rooted at $g$ no longer violates invariant 4 (if a node is red, its children must be black). If $g$ has a black parent we have finished. If not, the only possible RBT violation is either that $g$ is the root (has no parent) and is red (violating invariant 2) or that $g$ and its parent are both red. In the former case it suffices to recolour $g$ black, because no other part of the tree is affected. In the latter case, the problem of two consecutive red nodes is moved two levels up the tree (rename $g$ as the new $n$, $g$'s red parent as $p$ and so forth) and must be dealt with again as one of the three cases.




When in **case 2** ($n, p$ red, $g, u$ black, $g \to p \to n$ is a zig-zag, as above), perform a rotation on the $p - n$ edge to transform the zig-zag into a straight $g \to n \to p$. Note how this has made $n$ the parent of $p$, so the names are no longer appropriate. If you swap the names of $n$ and $p$, however, you are now in case 3.




When in **case 3** ($n, p$ red, $g, u$ black, $g \to p \to n$ is straight, as above), swap the colours of $p$ and $g$ and rotate the $p - g$ edge to make $p$ become the root of the subtree instead of $g$. You end up with $p$ as a black root-of-subtree with two red children ($n$ and $g$), and $g$ has a black child. Since $p$ is black, it can be child of any node (or no node) without violating invariants 2 or 4. You no longer have any local RBT violations anywhere and you may stop.

This summary does cover all possible cases and gives a fair idea of what happens but proving that everything is correct is not trivial (particularly proving that the global invariant 5 is maintained). The crucial part of the correctness proof is establishing and maintaining the loop invariant, which unfortunately will be less clean than a naïve "the tree obeys the five RBT invariants" because previous iterations of the loop (of case 1 in particular) may have violated some of them, and we may not have fixed them back yet. Deletion is even more involved and we won't have time to look at the detailed rules for the red-black version; but we'll deal with deletion for B-trees, which are a more general version of 2-3-4 trees, which in turn are another form of red-black trees...

When you come to writing and debugging actual code (always a recommended, instructive and enjoyable activity in this course) you will probably feel that there seem to be uncomfortably many pointers to keep track of and cases to deal with, and that it is tedious having to cope with both each case and its mirror image. But, with a clear head, it is still fundamentally OK.

## 4.6 B-trees

**Textbook**

Study chapter 18 in CLRS3.

The trees described so far (BSTs, red-black trees and 2-3-4 trees) are meant to be instantiated in dynamically allocated main memory. With data structures kept on disc, instead, it is sensible to make the unit of

data fairly large—perhaps some size related to the natural storage unit that your physical disc uses (a sector, cluster or track). Minimizing the total number of separate disc accesses will be more important than getting the ultimately best packing density. There are of course limits, and use of over-the-top data blocks will use up too much fast main memory and cause too much unwanted data to be transferred between disc and main memory along with each necessary bit.

B-trees are a good general-purpose disc data structure. We start by generalizing the idea of a sorted binary tree to a tree with a very high branching factor. The expected implementation is that each node will be a disc block containing an alternation of keys[24] and pointers to subtrees. This will tend to define the maximum branching factor that can be supported in terms of the natural disc block size and the amount of memory needed for each key. When new items are added to a B-tree it will often be possible to add the item within an existing block without overflow. Any block that becomes full can be split into two, and the single reference to it from its parent block expands to the two references to the new half-empty blocks. For B-trees of reasonable branching factor, any reasonable amount of data can be kept in a quite shallow tree: although the theoretical cost of access grows with the logarithm of the number of data items stored, in practical terms it is constant.

Each node of a B-tree has a lower and an upper bound on the number of keys it may contain[25]. When the number of keys exceeds the upper bound, the node must be split; conversely, when the number of keys goes below the lower bound, the node must be merged with another one—and either of these operations might potentially trigger other rearrangements. The tree as a whole is characterized by an integer parameter $t \geq 2$ called the **minimum degree** of the B-tree: each node must have between $t$ and $2t$ pointers[26], and therefore between $t-1$ and $2t-1$ keys. There is a variant known as B*-tree ("b star tree") in which non-root internal nodes must be at least 2/3 full, rather than at least 1/2 full as in the regular B-tree. The formal rules can be stated as follows.

1. There are internal nodes (with keys and payloads and children) and leaf nodes (without keys or payloads or children).

[24] More precisely key-value pairs, as usual, since the reason for looking up a key is ultimately to retrieve the value or satellite data associated with it. In practice the "payload" shown below each key in the picture will often be a *pointer* to the actual payload, unless values are of small and constant size.

[25] Except that no lower bound is imposed on the root, otherwise it would be impossible to represent B-trees that were nearly empty.

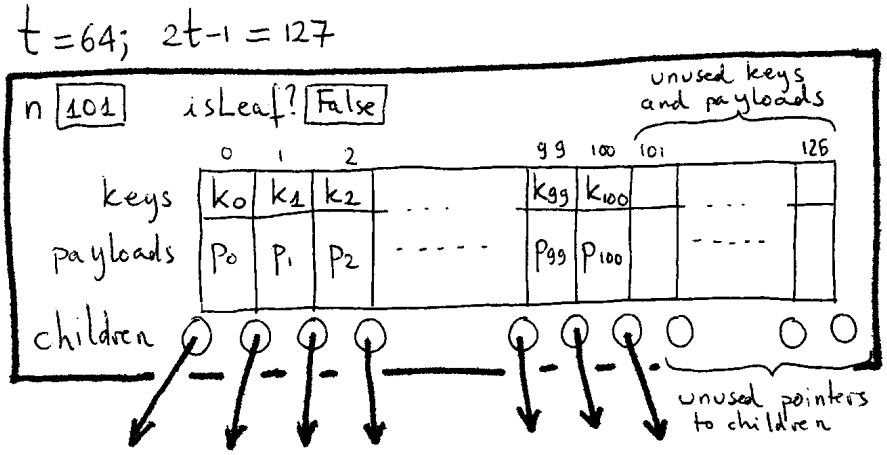
[26] See footnote 25 again.

2. For each key in a node, the node also holds the associated payload[27].

3. All leaf nodes are at the same distance from the root.

4. All internal nodes have at most $2t$ children; all internal nodes except the root have at least $t$ children.

5. A node has $c$ children iff it has $c - 1$ keys.




The example above assumes a B-tree with $t = 64$, meaning each internal node will have between 64 and 128 pointers to children and thus between 63 and 127 keys (numbered from 0, so reaching up to 126 at most). The particular internal node pictured holds $n = 101$ keys (numbered from 0 to 100) and thus points to 102 children.

The algorithms for adding new data into a B-tree ensure that the tree remain balanced. This means that the cost of accessing data in such a tree can be guaranteed to remain low even in the worst case. The ideas behind keeping B-trees balanced are a generalization of those used for 2-3-4-trees[28] but note that the implementation details may be significantly different, firstly because the B-tree will have such a large branching factor and secondly because all operations will need to be performed with a view to the fact that the most costly step is reading a disc block. In contrast, 2-3-4-trees are typically used as in-memory data structures so you count memory accesses rather than disc accesses when evaluating and optimizing an implementation.

[27]Or, more likely, a pointer to it, unless the payload has a small fixed size comparable to that of the pointer.

[28]Structurally, you can view 2-3-4 trees as a subcase of B-trees with $t = 2$.

### 4.6.1 Inserting

To insert a new key (and payload) into a B-tree, look for the key in the B-tree in the usual way. If found, update the payload in place. If not found, you'll be by then in the right place at the bottom level of the tree (the one where nodes have keyless leaves as children); on the way down, whenever you find a full node, split it in two on the median key and migrate the median key and resulting two children to the parent node (which by inductive hypothesis won't be full). If the root is full when you start, split it into three nodes (yielding a new root with only one key and adding one level to the tree). Once you get to the appropriate bottom level node, which won't be full or you would have split it on your way there, insert there.

**Exercise 42**
*(The following is not hard but it will take somewhat more than five minutes.)* Using a soft pencil, a large piece of paper and an eraser, draw a B-tree with $t = 2$, initially empty, and insert into it the following values in order:

$$63, 16, 51, 77, 61, 43, 57, 12, 44, 72, 45, 34, 20, 7, 93, 29.$$

How many times did you insert into a node that still had room? How many node splits did you perform? What is the depth of the final tree? What is the ratio of free space to total space in the final tree?

### 4.6.2 Deleting

Deleting is a more elaborate affair because it involves numerous subcases.

You can't delete a key from anywhere other than a bottom node (i.e. one whose children are keyless leaves), otherwise you upset its left and right children that lose their separator. In addition, you can't delete a key from a node that already has the minimum number of keys. So the general algorithm consists of creating the right conditions and then deleting (or, alternatively, deleting and then readjusting).

To move a key to a bottom node for the purpose of deleting it, swap it with its successor (which must be in a bottom node). The tree will have a temporary inversion, but that will disappear as soon as the unwanted key is deleted.

**Exercise 43**
Prove that, if a key is not in a bottom node, its successor, if it exists, must be.

To refill a node that has too few keys, use an appropriate combination of the following three operations, which rearrange a local part of a B-tree in constant time preserving all the B-tree properties.

**Merge** The first operation *merges* two adjacent sibling nodes and the key that separates them from the parent node. The parent node loses one key.

**Split** The reverse operation *splits* a node into three: a left sibling, a separating key and a right sibling. The separating key is sent up to the parent.

**Redistribute** The last operation *redistributes* the keys among two adjacent sibling nodes. It may be thought of as a merge followed by a split in a different place[29], and this different place will typically be the centre of the large merged node.

Each of these operations is only allowed if the new nodes thus formed respect their min and max capacity constraints.

Here is then the pseudocode algorithm to delete a key $k$ from the B-tree.

```
0  def delete(k):
1      """B-tree method for deleting key k.
2      PRECONDITION: k is in this B-tree.
3      POSTCONDITION: k is no longer in this B-tree."""
4
5      if k is in a bottom node B:
6          if B can lose a key without becoming too small:
7              delete k from B locally
8          else:
9              refill B (see below)
10             delete k from B locally
11     else:
12         swap k with its successor
13         # ASSERT: now k is in a bottom node
14         delete k from the bottom node with a recursive invocation
```

[29] We say "thought of" because such a merge might be disallowed as a stand-alone B-tree operation—the resulting node might end up having more than the allowed number of keys.

The following pseudocode, invoked as a subroutine by the previous fragment, refills a node $B$ that currently has the minimum allowed number of keys.

```
0  def refill(B):
1      """B-tree method for refilling node B.
2      PRECONDITION: B is an internal node of this B-tree, with t-1 keys.
3      POSTCONDITION: B now has more than t-1 keys."""
4
5      if either the left or right sibling of B can afford to lose any keys:
6          redistribute keys between B and that sibling
7      else:
8          # ASSERT: B and its siblings all have the min number of keys, t-1
9          merge B with either of its siblings
10         # ...this may require recursively refilling the parent of B,
11         # because it will lose a key during the merge.
```

## 4.7 Hash tables

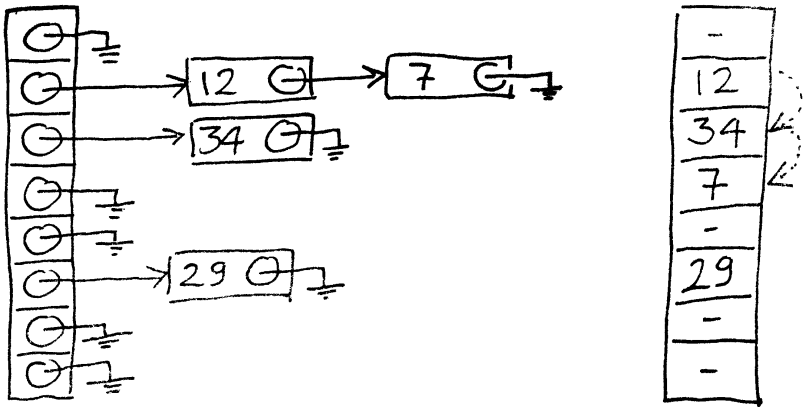
**Textbook**

Study chapter 11 in CLRS3.

A hash table implements the general case of the Dictionary ADT, where keys may not have a total order defined on them. In fact, even when the keys used *do* have an order relationship associated with them, it may be worth looking for a way of building a dictionary without using this order. Binary search makes locating items in a dictionary easier by imposing a coherent and ordered structure; hashing, instead, places its bet the other way, on chaos.

A hash function $h(k)$ maps a key of arbitrary length onto an integer in the range 0 to $m-1$ for some size $m$ and, for a good hash function, this mapping will appear to have hardly any pattern. Now, if we have an array of size $m$, we can try to store a key-value pair with key $k$ at location $h(k)$ in the array. Unfortunately, sometimes two distinct keys $k_1$ and $k_2$ will map to the same location $h(k_1) = h(k_2)$; this is known as a *collision*. There are two main strategies for handling it, called *chaining* and *open addressing* and illustrated below[30].

[30] See footnote 33.




**Chaining.** We can arrange that the locations in the array hold little linear lists[31] that collect all the keys that hash to that particular value. A good hash function will distribute keys fairly evenly over the array, so with luck this will lead to lists with average length $\lceil n/m \rceil$ if $n$ keys are in use[32].

**Exercise 44**
*(Trivial)* Make a hash table with 8 slots and insert into it the following values:

$$15, 23, 12, 20, 19, 8, 7, 17, 10, 11.$$

Use the hash function

$$h(k) = (k \bmod 10) \bmod 8$$

and, of course, resolve collisions by chaining.

**Open addressing.** The second way of using hashing is to use the hash value $h(n)$ as just a first preference for where to store the given key in the array. On adding a new key, if that location is empty then well and good—it can be used; otherwise, a succession of other probes are made of the hash table according to some rule until either the key is found to be present or an empty slot for it is located. The simplest (but not the best) method of collision resolution is to try successive array locations on from the place of

[31] These lists are dynamically allocated and external to the array.
[32] Note that $n$ might be $\gg m$.

the first probe, wrapping round at the end of the array[33]. Note that, with the open addressing strategy, where all keys are kept in the array, the array may become full, and that its performance decreases significantly when it is nearly full; implementations will typically double the size of the array once occupancy goes above a certain threshold.

**Exercise 45**
*Non-trivial* Imagine redoing the exercise above but resolving collisions by open addressing. When you go back to the table to retrieve a certain element, if you land on a non-empty location, how can you tell whether you arrived at the location for the desired key or on one occupied by the overspill from another one? *(Hint: describe precisely the low level structure of each entry in the table.)*

**Exercise 46**
How can you handle deletions from an open addressing table? What are the problems of the obvious naïve approach?

The worst-case cost of using a hash table can be dreadful. For instance, given some particular hash function, a malicious adversary could select keys so that they all hashed to the same value. The average case, however, is pretty good: so long as the number of items stored is sufficiently smaller than the size of the hash table, then both adding and retrieving data should have constant cost. When the hash table is mostly empty this result can be derived trivially. The analysis of expected costs for hash tables that have a realistic load is of course more complex but, so long as the hash table isn't too close to being full, we still get constant-time performance for the average case.

[33]In the picture above, keys 12 and 7 both hash to array index 1. In the chaining version, they are both stored in a linked list at array position 1. In the open addressing version, with linear probing, key 12 was stored at position 1 but, on attempting to store key 7, it was found that $h(7) = 1$ was an already occupied position. So, position 2 was tried; but that too was occupied, by 34 whose hash was 2. So the next free position was used, which turned out to be 3, even though $h(7) \neq 3$.

### 4.7.1 A short note on terminology

Confusingly, some people (not us) use the terms "open hashing" to refer to "chaining" and "closed hashing" to refer to "open addressing"—be ready for it. But we shall instead consistently use the terminology in CLRS3, as indicated above.

### 4.7.2 Probing sequences for open addressing

Many strategies exist for determining the sequence of slots to visit until a free one is found. We may describe the probe sequence as a function of the key $k$ and of the attempt number $j$ (in the range from 0 to $m-1$). Using a Java-like pseudocode to show argument types:

```
0 int probe(Key k, int j);
1 // BEHAVIOUR: return the array index to be probed at attempt <j>
2 // for key <k>.
```

So as not to waste slots, the curried function obtained by fixing the key to any constant value must be a permutation of $0, \ldots, m-1$ (the range of indices of the array). In other words, we wish to avoid probe sequences that, for some keys, fail to explore some slots.

**Linear probing** This easy probing function just returns $h(k) + j \bmod m$. In other words, at every new attempt, try the next cell in sequence. It is always a permutation. Linear probing is simple to understand and implement but it leads to *primary clustering*: many failed attempts hit the same slot *and* spill over to the same follow-up slots. The result is longer and longer runs of occupied slots, increasing search time.

**Quadratic probing** With quadratic probing you return $h(k) + cj + dj^2 \bmod m$ for some constants $c$ and $d$. This works much better than linear probing, provided that $c$ and $d$ are chosen appropriately: when two distinct probing sequences hit the same slot, in subsequent probes they then hit different slots. However it still leads to *secondary clustering* because any two keys that hash to the same value will yield the same probing sequence.

**Double hashing** With double hashing the probing sequence is $h_1(k) + j \cdot h_2(k) \bmod m$, using two different hash functions $h_1$ and $h_2$. As a consequence, even keys that hash to the same value (under $h_1$) are in fact assigned different probing sequences. It is the best of the three methods in terms of spreading the probes across all slots, but of course each access costs an extra hash function computation.

How should the probing sequence be used? The `get()`, `set()` and `delete()` methods all need first to determine the array slot to read from, write to or delete respectively. To do that, they keep calling the probing function until they reach the correct slot or until they have tried $m$ times unsuccessfully.

In the case of `get()`, the correct slot is the first one in the sequence that contains the sought key; and if an empty slot is found along the way, then the key is not in the dictionary even if fewer than $m$ probes have been attempted.

In the case of `set()`, if a slot with the sought key is found, that's the one to use, and we are overwriting a previous value for the same key; otherwise, the first empty slot in the probing sequence should be used, and we'll be setting a value for this key for the first time. If $m$ probes are all unsuccessful, then the array is full and insertion cannot take place without first resizing it.

In the case of deletion, there are complications. Assuming that we found the slot with the key to be deleted, if we naïvely marked it as empty we would be potentially interrupting some chains and making it impossible for `get()` to reach any stored keys whose probing sequence had that slot in a previous position. What we need to do is mark the slot as *deleted* (a special value distinct from empty) and change the implementations of `get()` and `set()` to treat the deleted value appropriately (treating it like non-empty for `get()` but like empty for `set()`).

Unfortunately, even with this fix, one moment's thought shows that a long sequence of `set()` and `delete()` operations will eventually result in the array running out of empty slots even though many may be marked deleted. As a result, the `get()` operation will become slower and slower (because chains of deleted slots need to be traversed until the sought value is found or an empty cell is reached) until, in the limit, all unsuccessful searches will cost $m$ probes. This is of course unacceptable and therefore, when a dictionary must support deletions, it should either be implemented with chaining or, in case of open addressing, it should be rehashed well before it runs out of empty cells.

Rehashing a dictionary consists of creating a new array (usually twice as large, to amortize the cost of the operation; clearly with a new hash function, since the index range is different), inserting every key-value pair of the old array into the new one and deleting the old array. Deleted slots are not copied and are therefore implicitly transformed into empty ones. Even without deletions, the resizing and rehashing procedure is also necessary to preserve performance when the load factor (array cells in use divided by array cells available) becomes too high, simply as a consequence of too many insertions for the size of the array. In the

HashMap data structure supplied with the Java library, for example, by default a rehash is triggered when the load factor reaches 75%.

## 4.8 Priority queues and heaps

**Textbook**

Study chapter 6 in CLRS3.

If we concentrate on the operations `set()`, `min()` and `delete()`, subject to the extra condition that the only item we ever delete will be the one just identified as the minimum one in our set, then the data structure we have is known as a **priority queue**. As the name says, this data structure is useful to keep track of a dynamic set of "clients" (e.g. operating system processes, or graph edges), each keyed with its priority, and have the highest-priority one always be promoted to the front of the queue, regardless of when it joined. Another operation that this structure must support is the "promotion" of an item to a more favourable position in the queue, which is equivalent to rewriting the key of the item.

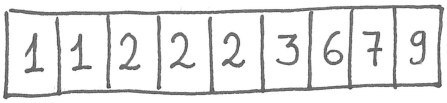
A priority queue is thus a data structure that holds a dynamic set of items and offers convenient access to the item with highest priority[34], as well as facilities for extracting that item, inserting new items and promoting a given item to a priority higher than its current one.

```
0  ADT PriorityQueue {
1    void insert(Item x);
2    // BEHAVIOUR: add item <x> to the queue.
3
4    Item first(); // equivalent to min()
5    // BEHAVIOUR: return the item with the smallest key (without
6    // removing it from the queue).
7
8    Item extractMin(); // equivalent to delete(), with restriction
9    // BEHAVIOUR: return the item with the smallest key and remove it
10   // from the queue.
11
12   void decreaseKey(Item x, Key new);
13   // PRECONDITION: new < x.key
14   // PRECONDITION: Item <x> is already in the queue.
```

[34] "Highest priority" by convention means "earliest in the sorting order" and therefore "numerically smallest" in case of integers. Priority 1 is higher than priority 3.

```
15    // POSTCONDITION: x.key == new
16    // BEHAVIOUR: change the key of the designated item to the designated
17    // value, thereby increasing the item's priority (while of course
18    // preserving the invariants of the data structure).
19
20    void delete(Item x);
21    // PRECONDITION: item <x> is already in the queue.
22    // BEHAVIOUR: remove item <x> from the queue.
23    // IMPLEMENTATION: make <x> the new minimum by calling decreaseKey with
24    // a value (conceptually: minus infinity) smaller than any in the queue;
25    // then extract the minimum and discard it.
26  }
27
28  ADT Item {
29    // A total order is defined on the keys.
30    Key k;
31    Payload p;
32  }
```

As for implementation, you could simply use a sorted array, but you'd have to keep the array sorted at every operation, for example with one pass of bubble-sort, which gives linear time costs for any operations that change the priority queue.




**Exercise 47**
Why do we claim that keeping the sorted-array priority queue sorted using bubble sort has linear costs? Wasn't bubble sort quadratic?

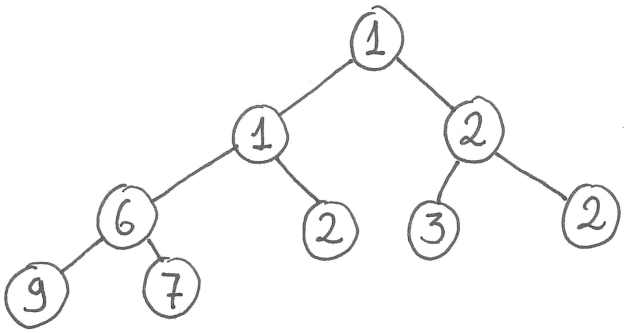
| Operation | Cost with sorted array |
|---|---|
| creation of empty queue | $O(1)$ |
| `first()` | $O(1)$ |
| `insert()` | $O(n)$ |
| `extractMin()` | $O(n)$ |
| `decreaseKey()` | $O(n)$ |
| `delete()` | $O(n)$ |

But we can do better than this.

### 4.8.1 Binary heaps

A good representation for a priority queue is the binary heap, which is the data structure implicitly used in the heapsort algorithm[35] of section 2.12. It is a clever yet comparatively simple construction that allows you to read out the highest priority item in constant time cost (if you don't remove it from the queue) and lets you achieve $O(\lg n)$ costs for all other priority queue operations.

A min-heap is a binary tree that satisfies two additional invariants: it is "almost full" (i.e. all its levels except perhaps the lowest have the maximum number of nodes, and the lowest level is filled left-to-right) and it obeys the "heap property" whereby each node has a key less than or equal to those of its children.




As a consequence of the heap property, the root of the tree is the smallest element. Therefore, to read out the highest priority item, just look at the root (constant cost). To insert an item, add it at the end of the heap and let it bubble up (following parent pointers) to a position where it no longer violates the heap property (max number of steps: proportional to the height of the tree). To extract the root, read it out, then replace it with the element at the end of the heap, letting the latter sink down until it no longer violates the heap property (again the max number of steps is proportional to the height of the tree). To reposition an item after decreasing its key, let it bubble up towards the root (again in no more steps than the height of the tree, within a constant factor).

Since the tree is balanced (by construction, because it is always "almost full"), its height never exceeds $O(\lg n)$, which is therefore the asymptotic complexity bound on all the priority queue operations that alter the tree.

[35] Except that heapsort uses a max-heap and here we use a min-heap.

| Operation | Cost with binary min-heap |
|---|---|
| creation of empty heap | $O(1)$ |
| `first()` | $O(1)$ |
| `insert()` | $O(\lg n)$ |
| `extractMin()` | $O(\lg n)$ |
| `decreaseKey()` | $O(\lg n)$ |
| `delete()` | $O(\lg n)$ |

These logarithmic costs represent good value and the binary heap, which is simple to code and compact to store[36], is therefore a good choice, in many cases, for implementing a priority queue.

### 4.8.2 Binomial heaps

**Exercise 48**
Before reading ahead: what is the most efficient algorithm you can think of to merge two binary heaps? What is its complexity?

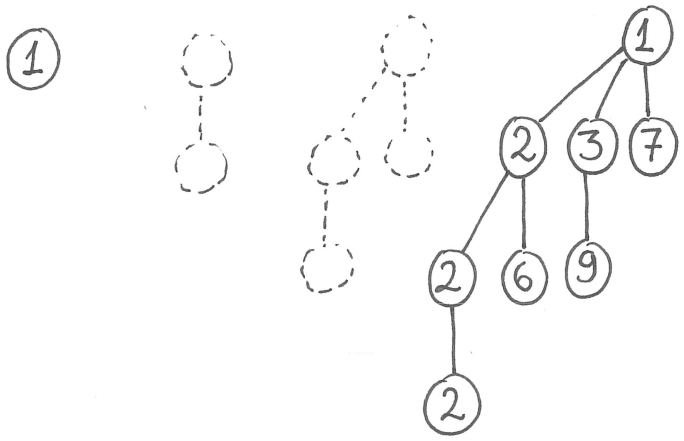
For some applications you might need to merge two priority queues (each with at most $n$ elements) into a larger one. With a binary heap, a trivial solution is to extract each of the elements from the smaller queue and insert them into the other, at a total cost bounded by $O(n \lg n)$. A smarter solution is to concatenate the two underlying arrays and then heapify the result in $O(n)$ time.

A more complex implementation of the priority queue is the binomial heap, whose main additional advantage is that it allows you to merge two priority queues, still at a time cost not exceeding $O(\lg n)$.

```
0 ADT BinomialHeap extends PriorityQueue {
1   void merge(BinomialHeap h);
2   // BEHAVIOUR: combine the current heap with the supplied heap h. In
3   // the process, make the supplied heap h empty and incorporate all its
4   // elements into the current heap.
5 }
```

A binomial heap is a forest of binomial trees, with special properties detailed below.

[36] The array representation does not even require any extra space for pointers.




A **binomial tree** (not heap) **of order 0** is a single node, containing one `Item`. It has height 0.

A **binomial tree of order** $k$ is a tree obtained by combining two binomial trees of order $k-1$, by appending one of the trees to the root of the other as the (new) leftmost child[37]. By induction, it contains $2^k$ nodes (since the number of nodes doubles at each new order). By induction, the number of child subtrees of the root of the tree (known as the *degree* of the tree) is $k$, the same as the tree's order (since at each new order the tree gains one more child). By induction, the height of the tree is also $k$, again same as the tree's order (since at each new order the tree grows taller by one level, because the new child is as tall as the previous order's tree and is shifted down by one).

A **binomial heap** is a collection of binomial trees (at most one for each tree order), sorted by increasing size, each obeying the "heap property" by which each node has a key less than or equal to those of its children. If the heap contains $n$ nodes, it contains $O(\lg n)$ binomial trees and the largest of those trees[38] has degree $O(\lg n)$.

**Exercise 49**
Draw a binomial tree of order 4.

[37] Note that a binomial tree is not a binary tree: each node can have an arbitrary number of children. Indeed, by "unrolling" the recursive definition above, you can derive an equivalent one that says that a tree of order $k$ consists of a root node with $k$ children that are, respectively, binomial trees of all the orders from $k-1$ down to 0.

[38] And hence *a fortiori* also each of the trees in the heap.

**Exercise 50**
Give proofs of each of the stated properties of binomial trees (trivial) and heaps (harder until you read the next paragraph—try before doing so).

The following property is neat: since the number of nodes and even the *shape* of the binomial tree of order $k$ is completely determined a priori, and since each binomial heap has at most one binomial tree for any given order, then, given the number $n$ of nodes of a binomial heap, one can immediately deduce the orders of the binomial trees contained in the heap just by looking at the "1" digits in the binary representation of $n$. For example, if a binomial heap has 13 nodes (binary $1101 = 2^3+2^2+2^0$), then the heap must contain a binomial tree of order 3, one of order 2 and one of order 0—just so as to be able to hold precisely 13 nodes.

**Exercise 51**
Prove that the sequence of trees in a binomial heap exactly matches the bits of the binary representation of the number of elements in the heap.

The operations that the binomial heap data structure provides are implemented as follows.

`first()` To find the element with the smallest key in the whole binomial heap, scan the roots of all the binomial trees in the heap, at cost $O(\lg n)$ since there are that many trees.

`extractMin()` To extract the element with the smallest key, which is necessarily a root, first find it, as above, at cost $O(\lg n)$; then cut it out from its tree. Its children now form a forest of binomial trees of smaller orders, already sorted by decreasing size. Reverse this list of trees[39] and you have another binomial heap. Merge this heap with what remains of the original one. Since the merge operation itself (*q.v.*) costs $O(\lg n)$, this is also the total cost of extracting the minimum.

[39] An operation linear in the number of child trees of the root that was just cut off. Since the degree of a binomial tree of order $k$ is $k$, and the number of nodes in the tree is $2^k$, the number of child trees of the cut-off root is bounded by $O(\lg n)$.

`merge()` To merge two binomial heaps, examine their trees by increasing tree order and combine them following a procedure similar to the one used during binary addition with carry with a chain of full adders.

> "binary addition" procedure. Start from order 0 and go up. At each position, say that for tree order $j$, consider up to three inputs: the tree of order $j$ of the first heap, if any; the tree of order $j$ of the second heap, if any; and the "carry" from order $j-1$, if any. Produce two outputs: one tree of order $j$ (or none) as the result for order $j$, and one tree of order $j+1$ (or none) as the carry from order $j$ to order $j+1$. All these inputs and outputs are either empty or they are binomial trees. If all inputs are empty, all outputs are too. If exactly one of the three inputs is non-empty, that tree becomes the result for order $j$, and the carry is empty. If exactly two inputs are non-empty, combine them to form a tree of order $j+1$ by appending the tree with the larger root to the other; this becomes the carry, and the result for order $j$ is empty. If three inputs are non-empty, two of them are combined as above to become the carry towards order $j+1$ and the third becomes the result for order $j$.

The number of trees in each of the two binomial heaps to be merged is bounded by $O(\lg n)$ (where by $n$ we indicate the total number of nodes in both heaps together) and the number of elementary operations to be performed for each tree order is bounded by a constant. Therefore, the total cost of the merge operation is $O(\lg n)$.

`insert()` To insert a new element, consider it as a binomial heap with only one tree with only one node and merge it as above, at cost $O(\lg n)$.

`decreaseKey()` To decrease the key of an item, proceed as in the case of a normal binary heap within the binomial tree to which the item belongs, at cost no greater than $O(\lg n)$ which bounds the height of that tree.

| Operation | Cost with binomial heap |
|---|---|
| creation of empty heap | $O(1)$ |
| `first()` | $O(\lg n)$ |
| `insert()` | $O(\lg n)$ |
| `extractMin()` | $O(\lg n)$ |
| `decreaseKey()` | $O(\lg n)$ |
| `delete()` | $O(\lg n)$ |
| `merge()` | $O(\lg n)$ |

Although the programming complexity is greater than for the binary heap, these logarithmic costs represent good value and therefore implementing a priority queue with a binomial heap is a good choice for applications where an efficient merge operation is required. If however there is no need for efficient merging, then the binary heap is less complex and somewhat faster.

Having said that, when the cost of an algorithm is *dominated* by specific priority queue operations, and where very large data sets are involved, as will be the case for some of the graph algorithms from the second half of the course when applied to a country's road network, or to the Web, then the search for even more efficient implementations is justified. We shall describe even more efficient (albeit much more complicated) priority queue implementations in the next chapter. If your analysis shows that the performance of your algorithm is limited by that of your priority queue, you may be able to improve asymptotic complexity by switching to a Fibonacci heap or a van Emde Boas tree. Since achieving the best possible computing times may occasionally rely on the performance of data structures as elaborate as these, it is important at least to know that they exist and where full details are documented. But be aware that the constant hidden by the big-O notation can be quite large.

## Finally, have your say

Those of you who tell me what they liked about this course motivate me to keep on improving it. Please leave your feedback, on paper and/or online.

**End of lecture course**

Thank you, and best wishes for the rest of your Tripos.