Lecture Notes for

# Data Structures and Algorithms

Revised each year by John Bullinaria

School of Computer Science
University of Birmingham
Birmingham, UK

Version of 27 March 2019

These notes are currently revised each year by John Bullinaria. They include sections based on notes originally written by Martín Escardó and revised by Manfred Kerber. All are members of the School of Computer Science, University of Birmingham, UK.

# Contents


**1 Introduction** **5**
- 1.1 Algorithms as opposed to programs . . . 5
- 1.2 Fundamental questions about algorithms . . . 6
- 1.3 Data structures, abstract data types, design patterns . . . 7
- 1.4 Textbooks and web-resources . . . 7
- 1.5 Overview . . . 8

**2 Arrays, Iteration, Invariants** **9**
- 2.1 Arrays . . . 9
- 2.2 Loops and Iteration . . . 10
- 2.3 Invariants . . . 10

**3 Lists, Recursion, Stacks, Queues** **12**
- 3.1 Linked Lists . . . 12
- 3.2 Recursion . . . 15
- 3.3 Stacks . . . 16
- 3.4 Queues . . . 17
- 3.5 Doubly Linked Lists . . . 18
- 3.6 Advantage of Abstract Data Types . . . 20

**4 Searching** **21**
- 4.1 Requirements for searching . . . 21
- 4.2 Specification of the search problem . . . 22
- 4.3 A simple algorithm: Linear Search . . . 22
- 4.4 A more efficient algorithm: Binary Search . . . 23

**5 Efficiency and Complexity** **25**
- 5.1 Time versus space complexity . . . 25
- 5.2 Worst versus average complexity . . . 25
- 5.3 Concrete measures for performance . . . 26
- 5.4 Big-O notation for complexity class . . . 26
- 5.5 Formal definition of complexity classes . . . 29

**6 Trees** **31**
- 6.1 General specification of trees . . . 31
- 6.2 Quad-trees . . . 32
- 6.3 Binary trees . . . 33

6.4 Primitive operations on binary trees . . . 34
6.5 The height of a binary tree . . . 36
6.6 The size of a binary tree . . . 37
6.7 Implementation of trees . . . 37
6.8 Recursive algorithms . . . 38

**7 Binary Search Trees** **40**
7.1 Searching with arrays or lists . . . 40
7.2 Search keys . . . 40
7.3 Binary search trees . . . 41
7.4 Building binary search trees . . . 41
7.5 Searching a binary search tree . . . 42
7.6 Time complexity of insertion and search . . . 43
7.7 Deleting nodes from a binary search tree . . . 44
7.8 Checking whether a binary tree is a binary search tree . . . 46
7.9 Sorting using binary search trees . . . 47
7.10 Balancing binary search trees . . . 48
7.11 Self-balancing AVL trees . . . 48
7.12 B-trees . . . 49

**8 Priority Queues and Heap Trees** **51**
8.1 Trees stored in arrays . . . 51
8.2 Priority queues and binary heap trees . . . 52
8.3 Basic operations on binary heap trees . . . 53
8.4 Inserting a new heap tree node . . . 54
8.5 Deleting a heap tree node . . . 55
8.6 Building a new heap tree from scratch . . . 56
8.7 Merging binary heap trees . . . 58
8.8 Binomial heaps . . . 59
8.9 Fibonacci heaps . . . 61
8.10 Comparison of heap time complexities . . . 62

**9 Sorting** **63**
9.1 The problem of sorting . . . 63
9.2 Common sorting strategies . . . 64
9.3 How many comparisons must it take? . . . 64
9.4 Bubble Sort . . . 66
9.5 Insertion Sort . . . 67
9.6 Selection Sort . . . 69
9.7 Comparison of $O(n^2)$ sorting algorithms . . . 70
9.8 Sorting algorithm stability . . . 71
9.9 Treesort . . . 71
9.10 Heapsort . . . 72
9.11 Divide and conquer algorithms . . . 74
9.12 Quicksort . . . 75
9.13 Mergesort . . . 79
9.14 Summary of comparison-based sorting algorithms . . . 81

9.15 Non-comparison-based sorts . . . 81
9.16 Bin, Bucket, Radix Sorts . . . 83

**10 Hash Tables** **85**
10.1 Storing data . . . 85
10.2 The Table abstract data type . . . 85
10.3 Implementations of the table data structure . . . 87
10.4 Hash Tables . . . 87
10.5 Collision likelihoods and load factors for hash tables . . . 88
10.6 A simple Hash Table in operation . . . 89
10.7 Strategies for dealing with collisions . . . 90
10.8 Linear Probing . . . 92
10.9 Double Hashing . . . 94
10.10 Choosing good hash functions . . . 96
10.11 Complexity of hash tables . . . 96

**11 Graphs** **98**
11.1 Graph terminology . . . 99
11.2 Implementing graphs . . . 100
11.3 Relations between graphs . . . 102
11.4 Planarity . . . 103
11.5 Traversals – systematically visiting all vertices . . . 104
11.6 Shortest paths – Dijkstra's algorithm . . . 105
11.7 Shortest paths – Floyd's algorithm . . . 111
11.8 Minimal spanning trees . . . 113
11.9 Travelling Salesmen and Vehicle Routing . . . 117

**12 Epilogue** **118**

**A Some Useful Formulae** **119**
A.1 Binomial formulae . . . 119
A.2 Powers and roots . . . 119
A.3 Logarithms . . . 119
A.4 Sums . . . 120
A.5 Fibonacci numbers . . . 121

# Chapter 1

# Introduction

These lecture notes cover the key ideas involved in designing *algorithms*. We shall see how they depend on the design of suitable *data structures*, and how some structures and algorithms are more *efficient* than others for the same task. We will concentrate on a few basic tasks, such as storing, sorting and searching data, that underlie much of computer science, but the techniques discussed will be applicable much more generally.

We will start by studying some key data structures, such as arrays, lists, queues, stacks and trees, and then move on to explore their use in a range of different searching and sorting algorithms. This leads on to the consideration of approaches for more efficient storage of data in hash tables. Finally, we will look at graph based representations and cover the kinds of algorithms needed to work efficiently with them. Throughout, we will investigate the computational efficiency of the algorithms we develop, and gain intuitions about the pros and cons of the various potential approaches for each task.

We will not restrict ourselves to implementing the various data structures and algorithms in particular computer programming languages (e.g., *Java*, *C*, *OCaml*), but specify them in simple *pseudocode* that can easily be implemented in any appropriate language.

## 1.1 Algorithms as opposed to programs

An *algorithm* for a particular task can be defined as "a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time". As such, an *algorithm* must be precise enough to be understood by *human beings*. However, in order to be *executed* by a *computer*, we will generally need a *program* that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind, programs usually need to contain more details than algorithms. Here we shall ignore most of those programming details and concentrate on the design of algorithms rather than programs.

The task of *implementing* the discussed algorithms as computer programs is important, of course, but these notes will concentrate on the theoretical aspects and leave the practical programming aspects to be studied elsewhere. Having said that, we will often find it useful to write down segments of actual programs in order to clarify and test certain theoretical aspects of algorithms and their data structures. It is also worth bearing in mind the distinction between different programming paradigms: *Imperative Programming* describes computation in terms of instructions that change the program/data state, whereas *Declarative Programming*

specifies what the program should accomplish without describing how to do it. These notes will primarily be concerned with developing algorithms that map easily onto the imperative programming approach.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called *pseudocode*, which comes in a variety of forms. Often these notes will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the overlap of *C* and *Java*, with the advantage that they can easily be inserted into runnable programs.

## 1.2 Fundamental questions about algorithms

Given an algorithm to solve a particular problem, we are naturally led to ask:

1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical terms normally used for these three aspects are:

1. Specification.
2. Verification.
3. Performance analysis.

The details of these three aspects will usually be rather problem dependent.

The *specification* should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.

For simple problems, it is often easy to see that a particular algorithm will always work, i.e. that it satisfies its specification. However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all. In this case, we need to spend some effort *verifying* whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect. However, since the number of different potential inputs for most algorithms is infinite in theory, and huge in practice, more than just testing on particular cases is needed to be sure that the algorithm satisfies its specification. We need *correctness proofs*. Although we will discuss proofs in these notes, and useful relevant ideas like *invariants*, we will usually only do so in a rather informal manner (though, of course, we will attempt to be rigorous). The reason is that we want to concentrate on the data structures and algorithms. Formal verification techniques are complex and will normally be left till after the basic ideas of these notes have been studied.

Finally, the *efficiency* or *performance* of an algorithm relates to the *resources* required by it, such as how quickly it will run, or how much computer memory it will use. This will

usually depend on the problem instance size, the choice of data representation, and the details of the algorithm. Indeed, this is what normally drives the development of new data structures and algorithms. We shall study the general ideas concerning efficiency in Chapter 5, and then apply them throughout the remainder of these notes.

## 1.3 Data structures, abstract data types, design patterns

For many problems, the ability to formulate an efficient algorithm depends on being able to organize the data in an appropriate manner. The term *data structure* is used to denote a particular way of organizing data for particular types of operation. These notes will look at numerous data structures ranging from familiar arrays and lists to more complex structures such as trees, heaps and graphs, and we will see how their choice affects the efficiency of the algorithms based upon them.

Often we want to talk about data structures without having to worry about all the implementational details associated with particular programming languages, or how the data is stored in computer memory. We can do this by formulating abstract mathematical models of particular classes of data structures or data types which have common features. These are called *abstract data types*, and are defined only by the operations that may be performed on them. Typically, we specify how they are built out of more *primitive data types* (e.g., integers or strings), how to extract that data from them, and some basic checks to control the flow of processing in algorithms. The idea that the implementational details are hidden from the user and protected from outside access is known as *encapsulation*. We shall see many examples of abstract data types throughout these notes.

At an even higher level of abstraction are *design patterns* which describe the design of algorithms, rather the design of data structures. These embody and generalize important design concepts that appear repeatedly in many problem contexts. They provide a general structure for algorithms, leaving the details to be added as required for particular problems. These can speed up the development of algorithms by providing familiar proven algorithm structures that can be applied straightforwardly to new problems. We shall see a number of familiar design patterns throughout these notes.

## 1.4 Textbooks and web-resources

To fully understand data structures and algorithms you will almost certainly need to complement the introductory material in these notes with textbooks or other sources of information. The lectures associated with these notes are designed to help you understand them and fill in some of the gaps they contain, but that is unlikely to be enough because often you will need to see more than one explanation of something before it can be fully understood.

There is no single best textbook that will suit everyone. The subject of these notes is a classical topic, so there is no need to use a textbook published recently. Books published 10 or 20 years ago are still good, and new good books continue to be published every year. The reason is that these notes cover important fundamental material that is taught in all university degrees in computer science. These days there is also a lot of very useful information to be found on the internet, including complete freely-downloadable books. It is a good idea to go to your library and browse the shelves of books on data structures and algorithms. If you like any of them, download, borrow or buy a copy for yourself, but make sure that most of the

topics in the above contents list are covered. Wikipedia is generally a good source of fairly reliable information on all the relevant topics, but you hopefully shouldn't need reminding that not everything you read on the internet is necessarily true. It is also worth pointing out that there are often many different equally-good ways to solve the same task, different equally-sensible names used for the same thing, and different equally-valid conventions used by different people, so don't expect all the sources of information you find to be an exact match with each other or with what you find in these notes.

## 1.5 Overview

These notes will cover the principal fundamental data structures and algorithms used in computer science, and bring together a broad range of topics covered elsewhere into a coherent framework. Data structures will be formulated to represent various types of information in such a way that it can be conveniently and efficiently manipulated by the algorithms we develop. Throughout, the recurring practical issues of algorithm specification, verification and performance analysis will be discussed.

We shall begin by looking at some widely used basic data structures (namely arrays, linked lists, stacks and queues), and the advantages and disadvantages of the associated abstract data types. Then we consider the ubiquitous problem of searching, and how that leads on to the general ideas of computational efficiency and complexity. That will leave us with the necessary tools to study three particularly important data structures: trees (in particular, binary search trees and heap trees), hash tables, and graphs. We shall learn how to develop and analyse increasingly efficient algorithms for manipulating and performing useful operations on those structures, and look in detail at developing efficient processes for data storing, sorting, searching and analysis. The idea is that once the basic ideas and examples covered in these notes are understood, dealing with more complex problems in the future should be straightforward.

# Chapter 2

# Arrays, Iteration, Invariants

Data is ultimately *stored* in computers as patterns of bits, though these days most programming languages deal with higher level objects, such as characters, integers, and floating point numbers. Generally, we need to build algorithms that manipulate collections of such objects, so we need procedures for storing and sequentially processing them.

## 2.1 Arrays

In computer science, the obvious way to store an ordered collection of items is as an *array*. Array items are typically stored in a sequence of computer memory locations, but to discuss them, we need a convenient way to write them down on paper. We can just write the items in order, separated by commas and enclosed by square brackets. Thus,

$$[1, 4, 17, 3, 90, 79, 4, 6, 81]$$

is an example of an array of integers. If we call this array $a$, we can write it as:

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

This array $a$ has 9 items, and hence we say that its *size* is 9. In everyday life, we usually start counting from 1. When we work with arrays in computer science, however, we more often (though not always) start from 0. Thus, for our array $a$, its positions are $0, 1, 2, \ldots, 7, 8$. The element in the $8^{\text{th}}$ position is 81, and we use the notation $a[8]$ to denote this element. More generally, for any integer $i$ denoting a position, we write $a[i]$ to denote the element in the $i^{\text{th}}$ position. This position $i$ is called an *index* (and the plural is *indices*). Then, in the above example, $a[0] = 1$, $a[1] = 4$, $a[2] = 17$, and so on.

It is worth noting at this point that the symbol = is quite *overloaded*. In mathematics, it stands for equality. In most modern programming languages, = denotes assignment, while equality is expressed by ==. We will typically use = in its mathematical meaning, unless it is written as part of code or pseudocode.

We say that the individual items $a[i]$ in the array $a$ are *accessed* using their index $i$, and one can move sequentially through the array by incrementing or decrementing that index, or jump straight to a particular item given its index value. Algorithms that process data stored as arrays will typically need to visit systematically all the items in the array, and apply appropriate operations on them.

## 2.2 Loops and Iteration

The standard approach in most programming languages for repeating a process a certain number of times, such as moving sequentially through an array to perform the same operations on each item, involves a *loop*. In *pseudocode*, this would typically take the general form

```
For i = 1,...,N,
   do something
```

and in programming languages like *C* and *Java* this would be written as the *for-loop*

```
for( i = 0 ; i < N ; i++ ) {
   // do something
}
```

in which a *counter* $i$ keep tracks of doing "the something" $N$ times. For example, we could compute the sum of all 20 items in an array $a$ using

```
for( i = 0, sum = 0 ; i < 20 ; i++ ) {
   sum += a[i];
}
```

We say that there is *iteration* over the index $i$. The general *for-loop* structure is

```
for( INITIALIZATION ; CONDITION ; UPDATE ) {
   REPEATED PROCESS
}
```

in which any of the four parts are optional. One way to write this out explicitly is

```
INITIALIZATION
if ( not CONDITION ) go to LOOP FINISHED
LOOP START
   REPEATED PROCESS
   UPDATE
   if ( CONDITION ) go to LOOP START
LOOP FINISHED
```

In these notes, we will regularly make use of this basic loop structure when operating on data stored in arrays, but it is important to remember that different programming languages use different syntax, and there are numerous variations that check the condition to terminate the repetition at different points.

## 2.3 Invariants

An *invariant*, as the name suggests, is a condition that does not change during execution of a given program or algorithm. It may be a simple inequality, such as "$i < 20$", or something more abstract, such as "the items in the array are sorted". Invariants are important for data structures and algorithms because they enable *correctness proofs* and *verification*.

In particular, a *loop-invariant* is a condition that is true at the beginning and end of every iteration of the given loop. Consider the standard simple example of a procedure that finds the minimum of $n$ numbers stored in an array $a$:

```
minimum(int n, float a[n]) {
   float min = a[0];
   // min equals the minimum item in a[0],...,a[0]
   for(int i = 1 ; i != n ; i++) {
      // min equals the minimum item in a[0],...,a[i-1]
      if (a[i] < min) min = a[i];
   }
   // min equals the minimum item in a[0],...,a[i-1], and i==n
   return min;
}
```

At the beginning of each iteration, and end of any iterations before, the invariant "*min* equals the minimum item in $a[0], ..., a[i-1]$" is true – it starts off true, and the repeated process and update clearly maintain its truth. Hence, when the loop terminates with "$i == n$", we know that "*min* equals the minimum item in $a[0], ..., a[n-1]$" and hence we can be sure that *min* can be returned as the required minimum value. This is a kind of *proof by induction*: the invariant is true at the start of the loop, and is preserved by each iteration of the loop, therefore it must be true at the end of the loop.

As we noted earlier, formal proofs of correctness are beyond the scope of these notes, but identifying suitable loop invariants and their implications for algorithm correctness as we go along will certainly be a useful exercise. We will also see how invariants (sometimes called *inductive assertions*) can be used to formulate similar correctness proofs concerning properties of data structures that are defined inductively.

# Chapter 3

# Lists, Recursion, Stacks, Queues

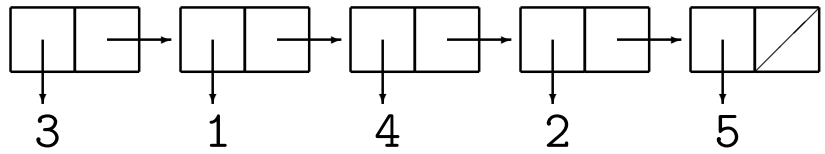
We have seen how arrays are a convenient way to *store* collections of items, and how loops and iteration allow us to sequentially process those items. However, arrays are not always the most efficient way to store collections of items. In this section, we shall see that lists may be a better way to store collections of items, and how recursion may be used to process them. As we explore the details of storing collections as lists, the advantages and disadvantages of doing so for different situations will become apparent.

## 3.1 Linked Lists

A list can involve virtually anything, for example, a list of integers $[3, 2, 4, 2, 5]$, a shopping list $[\texttt{apples}, \texttt{butter}, \texttt{bread}, \texttt{cheese}]$, or a list of web pages each containing a picture and a link to the next web page. When considering *lists*, we can speak about-them on different levels - on a very abstract level (on which we can define what we mean by a list), on a level on which we can depict lists and communicate as humans about them, on a level on which computers can communicate, or on a machine level in which they can be implemented.

### Graphical Representation

Non-empty *lists* can be represented by *two-cells*, in each of which the first cell contains a pointer to a list element and the second cell contains a *pointer* to either the empty list or another two-cell. We can depict a pointer to the empty list by a diagonal bar or cross through the cell. For instance, the list $[3, 1, 4, 2, 5]$ can be represented as:




### Abstract Data Type "List"

On an abstract level , a list can be *constructed* by the two *constructors*:

- `EmptyList`, which gives you the *empty list*, and

- `MakeList(element, list)`, which puts an *element* at the top of an existing *list*.

Using those, our last example list can be constructed as

$$\texttt{MakeList}(3, \texttt{MakeList}(1, \texttt{MakeList}(4, \texttt{MakeList}(2, \texttt{MakeList}(5, \texttt{EmptyList}))))).$$

and it is clearly possible to *construct* any list in this way.

This *inductive* approach to data structure creation is very powerful, and we shall use it many times throughout these notes. It starts with the "*base case*", the `EmptyList`, and then builds up increasingly complex lists by repeatedly applying the "*induction step*", the `MakeList(element, list)` operator.

It is obviously also important to be able to get back the elements of a list, and we no longer have an item index to use like we have with an array. The way to proceed is to note that a list is always constructed from the first element and the rest of the list. So, conversely, from a non-empty list it must always be possible to get the first element and the rest. This can be done using the two *selectors*, also called *accessor methods*:

- `first(list)`, and
- `rest(list)`.

The selectors will only work for non-empty lists (and give an error or exception on the empty list), so we need a *condition* which tells us whether a given list is empty:

- `isEmpty(list)`

This will need to be used to check every list before passing it to a selector.

We call everything a list that can be constructed by the constructors `EmptyList` and `MakeList`, so that with the selectors `first` and `rest` and the condition `isEmpty`, the following relationships are automatically satisfied (i.e. true):

- `isEmpty(EmptyList)`
- `not isEmpty(MakeList(x, l))` (for any `x` and `l`)
- $\texttt{first}(\texttt{MakeList}(\texttt{x}, \texttt{l})) = \texttt{x}$
- $\texttt{rest}(\texttt{MakeList}(\texttt{x}, \texttt{l})) = \texttt{l}$

In addition to constructing and getting back the components of lists, one may also wish to *destructively* change lists. This would be done by so-called *mutators* which change either the first element or the rest of a non-empty list:

- `replaceFirst(x, l)`
- `replaceRest(r, l)`

For instance, with $\texttt{l} = [3, 1, 4, 2, 5]$, applying $\texttt{replaceFirst}(9, \texttt{l})$ changes `l` to $[9, 1, 4, 2, 5]$. and then applying $\texttt{replaceRest}([6, 2, 3, 4], \texttt{l})$ changes it to $[9, 6, 2, 3, 4]$.

We shall see that the concepts of *constructors*, *selectors* and *conditions* are common to virtually all abstract data types. Throughout these notes, we will be formulating our data representations and algorithms in terms of appropriate definitions of them.

### XML Representation

In order to communicate data structures between different computers and possibly different programming languages, *XML* (eXtensible Markup Language) has become a quasi-standard. The above list could be represented in XML as:

```
<ol>
  <li>3</li>
  <li>1</li>
  <li>4</li>
  <li>2</li>
  <li>5</li>
</ol>
```

However, there are usually many different ways to represent the same object in XML. For instance, a cell-oriented representation of the above list would be:

```
<cell>
  <first>3</first>
  <rest>
    <cell>
      <first>1</first>
      <rest>
        <cell>
          <first>4</first>
          <rest>
            <cell>
              <first>2</first>
              <rest>
                <first>5</first>
                <rest>EmptyList</rest>
              </rest>
            </cell>
          </rest>
        </cell>
      </rest>
    </cell>
  </rest>
</cell>
```

While this looks complicated for a simple list, it is not, it is just a bit lengthy. XML is flexible enough to represent and communicate very complicated structures in a uniform way.

### Implementation of Lists

There are many different *implementations* possible for lists, and which one is best will depend on the primitives offered by the programming language being used.

The programming language *Lisp* and its derivates, for instance, take lists as the most important primitive data structure. In some other languages, it is more natural to implement

lists as arrays. However, that can be problematic because lists are conceptually not limited in size, which means array based implementation with fixed-sized arrays can only approximate the general concept. For many applications, this is not a problem because a maximal number of list members can be determined a priori (e.g., the maximum number of students taking one particular module is limited by the total number of students in the University). More general purpose implementations follow a pointer based approach, which is close to the diagrammatic representation given above. We will not go into the details of all the possible implementations of lists here, but such information is readily available in the standard textbooks.

## 3.2 Recursion

We previously saw how iteration based on for-loops was a natural way to process collections of items stored in arrays. When items are stored as linked-lists, there is no index for each item, and *recursion* provides the natural way to process them. The idea is to formulate procedures which involve at least one step that invokes (or calls) the procedure itself. We will now look at how to implement two important *derived procedures* on lists, `last` and `append`, which illustrate how recursion works.

To find the last element of a list `l` we can simply keep removing the first remaining item till there are no more left. This *algorithm* can be written in *pseudocode* as:

```
last(l) {
   if ( isEmpty(l) )
      error('Error: empty list in last')
   elseif ( isEmpty(rest(l)) )
      return first(l)
   else
      return last(rest(l))
}
```

The running time of this depends on the length of the list, and is proportional to that length, since `last` is called as often as there are elements in the list. We say that the procedure has *linear time complexity*, that is, if the length of the list is increased by some factor, the execution time is increased by the same factor. Compared to the *constant time complexity* which access to the last element of an array has, this is quite bad. It does not mean, however, that lists are inferior to arrays in general, it just means that lists are not the ideal data structure when a program has to access the last element of a long list very often.

Another useful procedure allows us to *append* one list `l2` to another list `l1`. Again, this needs to be done one item at a time, and that can be accomplished by repeatedly taking the first remaining item of `l1` and adding it to the front of the remainder appended to `l2`:

```
append(l1,l2) {
   if ( isEmpty(l1) )
      return l2
   else
      return MakeList(first(l1),append(rest(l1),l2))
}
```

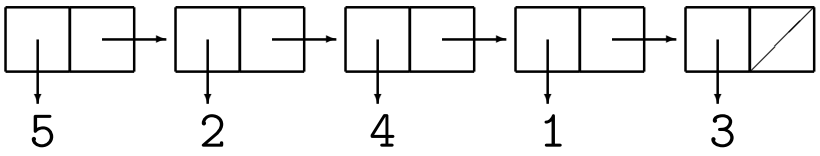
The time complexity of this procedure is proportional to the length of the first list, `l1`, since we have to call `append` as often as there are elements in `l1`.

## 3.3 Stacks

*Stacks* are, on an abstract level, equivalent to linked lists. They are the ideal data structure to model a *First-In-Last-Out (FILO)*, or *Last-In-First-Out (LIFO)*, strategy in search.

### Graphical Representation

Their relation to linked lists means that their graphical representation can be the same, but one has to be careful about the order of the items. For instance, the stack created by inserting the numbers $[3, 1, 4, 2, 5]$ in that order would be represented as:




### Abstract Data Type "Stack"

Despite their relation to linked lists, their different use means the *primitive operators* for stacks are usually given different names. The two *constructors* are:

- `EmptyStack`, the empty stack, and
- `push(element, stack)`, which takes an element and pushes it on top of an existing stack,

and the two *selectors* are:

- `top(stack)`, which gives back the top most element of a stack, and
- `pop(stack)`, which gives back the stack without the top most element.

The selectors will work only for non-empty stacks, hence we need a *condition* which tells whether a stack is empty:

- `isEmpty(stack)`

We have equivalent automatically-true relationships to those we had for the lists:

- `isEmpty(EmptyStack)`
- `not isEmpty(push(x, s))` (for any `x` and `s`)
- `top(push(x, s)) = x`
- `pop(push(x, s)) = s`

In summary, we have the direct correspondences:

| | constructors | | selectors | | condition |
|---|---|---|---|---|---|
| List | `EmptyList` | `MakeList` | `first` | `rest` | `isEmpty` |
| Stack | `EmptyStack` | `push` | `top` | `pop` | `isEmpty` |

So, stacks and linked lists are the same thing, apart from the different names that are used for their constructors and selectors.

### Implementation of Stacks

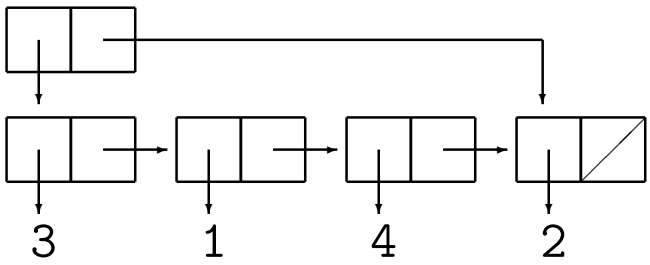
There are two different ways we can think about implementing stacks. So far we have implied a *functional* approach. That is, `push` does not change the original stack, but creates a new stack out of the original stack and a new element. That is, there are at least two stacks around, the original one and the newly created one. This functional view is quite convenient. If we apply `top` to a particular stack, we will always get the same element. However, from a practical point of view, we may not want to create lots of new stacks in a program, because of the obvious memory management implications. Instead it might be better to think of a single stack which is destructively changed, so that after applying `push` the original stack no longer exits, but has been changed into a new stack with an extra element. This is conceptually more difficult, since now applying `top` to a given stack may give different answers, depending on how the state of the system has changed. However, as long as we keep this difference in mind, ignoring such implementational details should not cause any problems.

## 3.4 Queues

A *queue* is a data structure used to model a *First-In-First-Out (FIFO)* strategy. Conceptually, we add to the end of a queue and take away elements from its front.

### Graphical Representation

A queue can be graphically represented in a similar way to a list or stack, but with an additional two-cell in which the first element points to the front of the list of all the elements in the queue, and the second element points to the last element of the list. For instance, if we insert the elements $[3, 1, 4, 2]$ into an initially empty queue, we get:



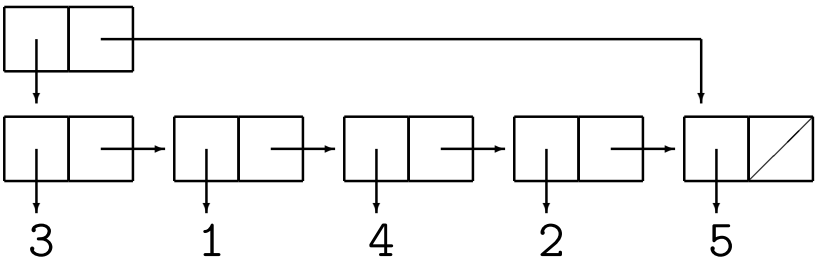

This arrangement means that taking the first element of the queue, or adding an element to the back of the queue, can both be done efficiently. In particular, they can both be done with constant effort, i.e. independently of the queue length.

### Abstract Data Type "Queue"

On an abstract level, a queue can be *constructed* by the two *constructors*:

- `EmptyQueue`, the empty queue, and
- `push(element, queue)`, which takes an element and a queue and returns a queue in which the element is added to the original queue at the end.

For instance, by applying `push(5, q)` where `q` is the queue above, we get




The two *selectors* are the same as for stacks:

- `top`(`queue`), which gives the top element of a queue, that is, `3` in the example, and
- `pop`(`queue`), which gives the queue without the top element.

And, as with stacks, the selectors only work for non-empty queues, so we again need a *condition* which returns whether a queue is empty:

- `isEmpty`(`queue`)

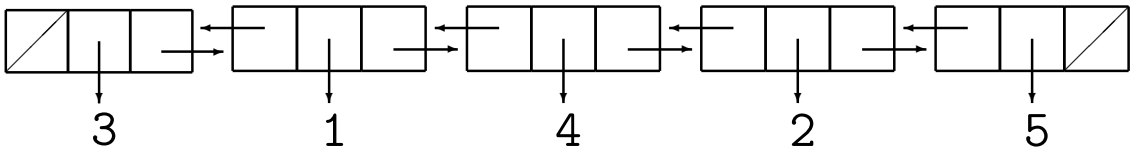
In later chapters we shall see practical examples of how queues and stacks operate with different effect.

## 3.5 Doubly Linked Lists

A *doubly linked list* might be useful when working with something like a list of web pages, which has each page containing a picture, a link to the previous page, and a link to the next page. For a simple list of numbers, a linked list and a doubly linked list may look the same, e.g., $[3, 1, 4, 2, 5]$. However, the doubly linked list also has an easy way to get the previous element, as well as to the next element.

### Graphical Representation

Non-empty doubly linked lists can be represented by *three-cells*, where the first cell contains a pointer to another three-cell or to the empty list, the second cell contains a pointer to the list element and the third cell contains a pointer to another three-cell or the empty list. Again, we depict the empty list by a diagonal bar or cross through the appropriate cell. For instance, $[3, 1, 4, 2, 5]$ would be represented as doubly linked list as:




### Abstract Data Type "Doubly Linked List"

On an abstract level , a doubly linked list can be *constructed* by the three *constructors*:

- `EmptyList`, the empty list, and

- `MakeListLeft(element, list)`, which takes an element and a doubly linked list and returns a new doubly linked list with the element added to the left of the original doubly linked list.
- `MakeListRight(element, list)`, which takes an element and a doubly linked list and returns a new doubly linked list with the element added to the right of the original doubly linked list.

It is clear that it may possible to construct a given doubly linked list in more that one way. For example, the doubly linked list represented above can be constructed by either of:

```
MakeListLeft(3, MakeListLeft(1, MakeListLeft(4, MakeListLeft(2,
                                                MakeListLeft(5, EmptyList)))))
```

```
MakeListLeft(3, MakeListLeft(1, MakeListRight(5, MakeListRight(2,
                                                    MakeListLeft(4, EmptyList)))))
```

In the case of doubly linked lists, we have four *selectors*:

- `firstLeft(list)`,
- `restLeft(list)`,
- `firstRight(list)`, and
- `restRight(list)`.

Then, since the selectors only work for non-empty lists, we also need a *condition* which returns whether a list is empty:

- `isEmpty(list)`

This leads to automatically-true relationships such as:

- `isEmpty(EmptyList)`
- `not isEmpty(MakeListLeft(x, l))` (for any `x` and `l`)
- `not isEmpty(MakeListRight(x, l))` (for any `x` and `l`)
- `firstLeft(MakeListLeft(x, l)) = x`
- `restLeft(MakeListLeft(x, l)) = l`
- `firstRight(MakeListRight(x, l)) = x`
- `restRight(MakeListRight(x, l)) = l`

### Circular Doubly Linked List

As a simple extension of the standard doubly linked list, one can define a *circular doubly linked list* in which the left-most element points to the right-most element, and vice versa. This is useful when we might need to move efficiently through a whole list of items, but might not be starting from one of two particular end points.

## 3.6 Advantage of Abstract Data Types

It is clear that the implementation of the abstract linked-list data type has the disadvantage that certain useful procedures may not be directly accessible. For instance, the standard *abstract data type* of a list does not offer an efficient procedure `last(l)` to give the last element in the list, whereas it would be trivial to find the last element of an array of a known number of elements. One could modify the linked-list data type by maintaining a pointer to the last item, as we did for the queue data type, but we still wouldn't have an easy way to access intermediate items. While `last(l)` and `getItem(i,l)` procedures can easily be implemented using the primitive constructors, selectors, and conditions, they are likely to be less efficient than making use of certain aspects of the underlying implementation.

That disadvantage leads to an obvious question: Why should we want to use abstract data types when they often lead to less efficient algorithms? Aho, Hopcroft and Ullman (1983) provide a clear answer in their book:

> *"At first, it may seem tedious writing procedures to govern all accesses to the underlying structures. However, if we discipline ourselves to writing programs in terms of the operations for manipulating abstract data types rather than making use of particular implementations details, then we can modify programs more readily by reimplementing the operations rather than searching all programs for places where we have made accesses to the underlying data structures. This flexibility can be particularly important in large software efforts, and the reader should not judge the concept by the necessarily tiny examples found in this book."*

This advantage will become clearer when we study more complex abstract data types and algorithms in later chapters.

# Chapter 4

# Searching

An important and recurring problem in computing is that of *locating information*. More succinctly, this problem is known as *searching*. This is a good topic to use for a preliminary exploration of the various issues involved in algorithm design.

## 4.1 Requirements for searching

Clearly, the information to be searched has to first be *represented* (or *encoded*) somehow. This is where *data structures* come in. Of course, in a computer, everything is ultimately represented as sequences of binary digits (bits), but this is too low level for most purposes. We need to develop and study useful data structures that are closer to the way humans think, or at least more structured than mere sequences of bits. This is because it is humans who have to develop and maintain the software systems – computers merely run them.

After we have chosen a suitable representation, the represented information has to be processed somehow. This is what leads to the need for *algorithms*. In this case, the process of interest is that of searching. In order to simplify matters, let us assume that we want to search a collection of integer numbers (though we could equally well deal with strings of characters, or any other data type of interest). To begin with, let us consider:

1. The most obvious and simple representation.
2. Two potential algorithms for processing with that representation.

As we have already noted, *arrays* are one of the simplest possible ways of representing collections of numbers (or strings, or whatever), so we shall use that to store the information to be searched. Later we shall look at more complex data structures that may make storing and searching more efficient.

Suppose, for example, that the set of integers we wish to search is {1,4,17,3,90,79,4,6,81}. We can write them in an array $a$ as

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

If we ask where 17 is in this array, the answer is 2, the index of that element. If we ask where 91 is, the answer is *nowhere*. It is useful to be able to represent *nowhere* by a number that is not used as a possible index. Since we start our index counting from 0, any negative number would do. We shall follow the convention of using the number $-1$ to represent *nowhere*. Other (perhaps better) conventions are possible, but we will stick to this here.

## 4.2 Specification of the search problem

We can now formulate a *specification* of our search problem using that data structure:

> *Given an array $a$ and integer $x$, find an integer $i$ such that*
>
> 1. *if there is no $j$ such that $a[j]$ is $x$, then $i$ is $-1$,*
> 2. *otherwise, $i$ is any $j$ for which $a[j]$ is $x$.*

The first clause says that if $x$ does not occur in the array $a$ then $i$ should be $-1$, and the second says that if it does occur then $i$ should be a position where it occurs. If there is more than one position where $x$ occurs, then this specification allows you to return any of them – for example, this would be the case if $a$ were $[17, 13, 17]$ and $x$ were 17. Thus, the specification is ambiguous. Hence different algorithms with different behaviours can satisfy the same specification – for example, one algorithm may return the smallest position at which $x$ occurs, and another may return the largest. There is nothing wrong with ambiguous specifications. In fact, in practice, they occur quite often.

## 4.3 A simple algorithm: Linear Search

We can conveniently express the simplest possible *algorithm* in a form of *pseudocode* which reads like English, but resembles a computer program without some of the precision or detail that a computer usually requires:

```
// This assumes we are given an array a of size n and a key x.
For i = 0,1,...,n-1,
    if a[i] is equal to x,
        then we have a suitable i and can terminate returning i.
If we reach this point,
    then x is not in a and hence we must terminate returning -1.
```

Some aspects, such as the ellipsis "...", are potentially ambiguous, but we, as human beings, know exactly what is meant, so we do not need to worry about them. In a programming language such as *C* or *Java*, one would write something that is more precise like:

```
for ( i = 0 ; i < n ; i++ ) {
    if ( a[i] == x ) return i;
}
return -1;
```

In the case of *Java*, this would be within a method of a class, and more details are needed, such as the parameter $a$ for the method and a declaration of the auxiliary variable $i$. In the case of *C*, this would be within a function, and similar missing details are needed. In either, there would need to be additional code to output the result in a suitable format.

In this case, it is easy to see that the algorithm satisfies the specification (assuming $n$ is the correct size of the array) – we just have to observe that, because we start counting from zero, the last position of the array is its size minus one. If we forget this, and let $i$ run from 0 to $n$ instead, we get an incorrect algorithm. The practical effect of this mistake is that the execution of this algorithm gives rise to an error when the item to be located in the array is

actually not there, because a non-existing location is attempted to be accessed. Depending on the particular language, operating system and machine you are using, the actual effect of this error will be different. For example, in *C* running under Unix, you may get execution aborted followed by the message "segmentation fault", or you may be given the wrong answer as the output. In *Java*, you will always get an *error message*.

## 4.4 A more efficient algorithm: Binary Search

One always needs to consider whether it is possible to improve upon the performance of a particular algorithm, such as the one we have just created. In the worst case, searching an array of size $n$ takes $n$ steps. On average, it will take $n/2$ steps. For large collections of data, such as all web-pages on the internet, this will be unacceptable in practice. Thus, we should try to organize the collection in such a way that a more efficient algorithm is possible. As we shall see later, there are many possibilities, and the more we demand in terms of efficiency, the more complicated the data structures representing the collections tend to become. Here we shall consider one of the simplest – we still represent the collections by arrays, but now we enumerate the elements in ascending order. The problem of obtaining an ordered list from any given list is known as *sorting* and will be studied in detail in a later chapter.

Thus, instead of working with the previous array $[1, 4, 17, 3, 90, 79, 4, 6, 81]$, we would work with $[1, 3, 4, 4, 6, 17, 79, 81, 90]$, which has the same items but listed in ascending order. Then we can use an improved *algorithm*, which in English-like pseudocode form is:

```
// This assumes we are given a sorted array a of size n and a key x.
// Use integers left and right (initially set to 0 and n-1) and mid.
While left is less than right,
    set mid to the integer part of (left+right)/2, and
    if x is greater than a[mid],
        then       set left to mid+1,
        otherwise  set right to mid.
If a[left] is equal to x,
    then       terminate returning left,
    otherwise  terminate returning -1.
```

and would correspond to a segment of *C* or *Java* code like:

```
/* DATA */
int a = [1,3,4,4,6,17,79,81,90];
int n = 9;
int x = 79;
/* PROGRAM */
int left = 0, right = n-1, mid;
while ( left < right ) {
    mid = ( left + right ) / 2;
    if ( x > a[mid] ) left = mid+1;
    else right = mid;
}
if ( a[left] == x ) return left;
else return -1;
```

This algorithm works by repeatedly splitting the array into two segments, one going from $left$ to $mid$, and the other going from $mid + 1$ to $right$, where $mid$ is the position half way from $left$ to $right$, and where, initially, $left$ and $right$ are the leftmost and rightmost positions of the array. Because the array is sorted, it is easy to see which of each pair of segments the searched-for item $x$ is in, and the search can then be restricted to that segment. Moreover, because the size of the sub-array going from locations $left$ to $right$ is halved at each iteration of the while-loop, we only need $\log_2 n$ steps in either the average or worst case. To see that this runtime behaviour is a big improvement, in practice, over the earlier linear-search algorithm, notice that $\log_2 1000000$ is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in the worst case of the linear-search algorithm.

With the binary search algorithm, it is not so obvious that we have taken proper care of the boundary condition in the while loop. Also, strictly speaking, this algorithm is not correct because it does not work for the empty array (that has size zero), but that can easily be fixed. Apart from that, is it correct? Try to convince yourself that it is, and then try to explain your argument-for-correctness to a colleague. Having done that, try to *write down* some convincing arguments, maybe one that involves a *loop invariant* and one that doesn't. Most algorithm developers stop at the first stage, but experience shows that it is only when we attempt to write down seemingly convincing arguments that we actually find all the subtle mistakes. Moreover, it is not unusual to end up with a better/clearer algorithm after it has been modified to make its correctness easier to argue.

It is worth considering whether linked-list versions of our two algorithms would work, or offer any advantages. It is fairly clear that we could perform a linear search through a linked list in essentially the same way as with an array, with the relevant pointer returned rather than an index. Converting the binary search to linked list form is problematic, because there is no efficient way to split a linked list into two segments. It seems that our array-based approach is the best we can do with the data structures we have studied so far. However, we shall see later how more complex data structures (trees) can be used to formulate efficient recursive search algorithms.

Notice that we have not yet taken into account how much effort will be required to sort the array so that the binary search algorithm can work on it. Until we know that, we cannot be sure that using the binary search algorithm really is more efficient overall than using the linear search algorithm on the original unsorted array. That may also depend on further details, such as how many times we need to performa a search on the set of $n$ items – just once, or as many as $n$ times. We shall return to these issues later. First we need to consider in more detail how to compare algorithm efficiency in a reliable manner.

# Chapter 5

# Efficiency and Complexity

We have already noted that, when developing algorithms, it is important to consider how *efficient* they are, so we can make informed choices about which are best to use in particular circumstances. So, before moving on to study increasingly complex data structures and algorithms, we first look in more detail at how to measure and describe their efficiency.

## 5.1 Time versus space complexity

When creating software for serious applications, there is usually a need to judge how quickly an algorithm or program can complete the given tasks. For example, if you are programming a flight booking system, it will not be considered acceptable if the travel agent and customer have to wait for half an hour for a transaction to complete. It certainly has to be ensured that the waiting time is reasonable for the size of the problem, and normally faster execution is better. We talk about the *time complexity* of the algorithm as an indicator of how the execution time depends on the *size* of the data structure.

Another important efficiency consideration is how much memory a given program will require for a particular task, though with modern computers this tends to be less of an issue than it used to be. Here we talk about the *space complexity* as how the memory requirement depends on the size of the data structure.

For a given task, there are often algorithms which trade time for space, and vice versa. For example, we will see that, as a data storage device, *hash tables* have a very good time complexity at the expense of using more memory than is needed by other algorithms. It is usually up to the algorithm/program designer to decide how best to balance the *trade-off* for the application they are designing.

## 5.2 Worst versus average complexity

Another thing that has to be decided when making efficiency considerations is whether it is the *average case* performance of an algorithm/program that is important, or whether it is more important to guarantee that even in the *worst case* the performance obeys certain rules. For many applications, the average case is more important, because saving time overall is usually more important than guaranteeing good behaviour in the worst case. However, for time-critical problems, such as keeping track of aeroplanes in certain sectors of air space, it may be totally unacceptable for the software to take too long if the worst case arises.

Again, algorithms/programs often trade-off efficiency of the average case against efficiency of the worst case. For example, the most efficient algorithm on average might have a particularly bad worst case efficiency. We will see particular examples of this when we consider efficient algorithms for sorting and searching.

## 5.3 Concrete measures for performance

These days, we are mostly interested in *time complexity*. For this, we first have to decide how to measure it. Something one might try to do is to just implement the algorithm and run it, and see how long it takes to run, but that approach has a number of problems. For one, if it is a big application and there are several potential algorithms, they would all have to be programmed first before they can be compared. So a considerable amount of time would be wasted on writing programs which will not get used in the final product. Also, the machine on which the program is run, or even the compiler used, might influence the running time. You would also have to make sure that the *data* with which you tested your program is typical for the application it is created for. Again, particularly with big applications, this is not really feasible. This empirical method has another disadvantage: it will not tell you anything useful about the next time you are considering a similar problem.

Therefore complexity is usually best measured in a different way. First, in order to not be bound to a particular programming language or machine architecture, it is better to measure the efficiency of the *algorithm* rather than that of its *implementation*. For this to be possible, however, the algorithm has to be described in a way which very much *looks* like the program to be implemented, which is why algorithms are usually best expressed in a form of *pseudocode* that comes close to the implementation language.

What we need to do to determine the time complexity of an algorithm is count the number of times each operation will occur, which will usually depend on the *size* of the problem. The size of a problem is typically expressed as an integer, and that is typically the number of items that are manipulated. For example, when describing a search algorithm, it is the number of items amongst which we are searching, and when describing a sorting algorithm, it is the number of items to be sorted. So the *complexity* of an algorithm will be given by a function which maps the number of items to the (usually approximate) number of time steps the algorithm will take when performed on that many items.

In the early days of computers, the various operations were each counted in proportion to their particular 'time cost', and added up, with multiplication of integers typically considered much more expensive than their addition. In today's world, where computers have become much faster, and often have dedicated floating-point hardware, the differences in time costs have become less important. However, we still we need to be careful when deciding to consider all operations as being equally costly – applying some function, for example, can take much longer than simply adding two numbers, and swaps generally take many times longer than comparisons. Just counting the most costly operations is often a good strategy.

## 5.4 Big-O notation for complexity class

Very often, we are not interested in the actual function $C(n)$ that describes the time complexity of an algorithm in terms of the problem size $n$, but just its *complexity class*. This ignores any constant overheads and small constant factors, and just tells us about the principal growth

of the complexity function with problem size, and hence something about the performance of the algorithm on large numbers of items.

If an algorithm is such that we may consider all steps equally costly, then usually the complexity class of the algorithm is simply determined by the number of loops and how often the content of those loops are being executed. The reason for this is that adding a constant number of instructions which does not change with the size of the problem has no significant effect on the overall complexity for large problems.

There is a standard notation, called the *Big-O notation*, for expressing the fact that constant factors and other insignificant details are being ignored. For example, we saw that the procedure `last(l)` on a list `l` had time complexity that depended linearly on the size $n$ of the list, so we would say that the time complexity of that algorithm is $O(n)$. Similarly, linear search is $O(n)$. For binary search, however, the time complexity is $O(\log_2 n)$.

Before we define complexity classes in a more formal manner, it is worth trying to gain some intuition about what they actually mean. For this purpose, it is useful to choose one function as a representative of each of the classes we wish to consider. Recall that we are considering functions which map natural numbers (the size of the problem) to the set of non-negative real numbers $\mathbb{R}^+$, so the classes will correspond to common mathematical functions such as powers and logarithms. We shall consider later to what degree a representative can be considered 'typical' for its class.

The most common complexity classes (in increasing order) are the following:

- $O(1)$, pronounced 'Oh of one', or *constant* complexity;
- $O(\log_2 \log_2 n)$, 'Oh of log log en';
- $O(\log_2 n)$, 'Oh of log en', or *logarithmic* complexity;
- $O(n)$, 'Oh of en', or *linear* complexity;
- $O(n\log_2 n)$, 'Oh of en log en';
- $O(n^2)$, 'Oh of en squared', or *quadratic* complexity;
- $O(n^3)$, 'Oh of en cubed', or *cubic* complexity;
- $O(2^n)$, 'Oh of two to the en', or *exponential* complexity.

As a representative, we choose the function which gives the class its name – e.g. for $O(n)$ we choose the function $f(n) = n$, for $O(\log_2 n)$ we choose $f(n) = \log_2 n$, and so on. So assume we have algorithms with these functions describing their complexity. The following table lists how many operations it will take them to deal with a problem of a given size:

| $f(n)$ | $n = 4$ | $n = 16$ | $n = 256$ | $n = 1024$ | $n = 1048576$ |
|---|---|---|---|---|---|
| $1$ | 1 | 1 | 1 | $1.00 \times 10^0$ | $1.00 \times 10^0$ |
| $\log_2 \log_2 n$ | 1 | 2 | 3 | $3.32 \times 10^0$ | $4.32 \times 10^0$ |
| $\log_2 n$ | 2 | 4 | 8 | $1.00 \times 10^1$ | $2.00 \times 10^1$ |
| $n$ | 4 | 16 | $2.56 \times 10^2$ | $1.02 \times 10^3$ | $1.05 \times 10^6$ |
| $n\log_2 n$ | 8 | 64 | $2.05 \times 10^3$ | $1.02 \times 10^4$ | $2.10 \times 10^7$ |
| $n^2$ | 16 | 256 | $6.55 \times 10^4$ | $1.05 \times 10^6$ | $1.10 \times 10^{12}$ |
| $n^3$ | 64 | 4096 | $1.68 \times 10^7$ | $1.07 \times 10^9$ | $1.15 \times 10^{18}$ |
| $2^n$ | 16 | 65536 | $1.16 \times 10^{77}$ | $1.80 \times 10^{308}$ | $6.74 \times 10^{315652}$ |

Some of these numbers are so large that it is rather difficult to imagine just how long a time span they describe. Hence the following table gives time spans rather than instruction counts, based on the assumption that we have a computer which can operate at a speed of 1 MIP, where one MIP = a million instructions per second:

| $f(n)$ | $n = 4$ | $n = 16$ | $n = 256$ | $n = 1024$ | $n = 1048576$ |
|---|---|---|---|---|---|
| 1 | 1 $\mu$sec | 1 $\mu$sec | 1 $\mu$sec | 1 $\mu$sec | 1 $\mu$sec |
| $\log_2 \log_2 n$ | 1 $\mu$sec | 2 $\mu$sec | 3 $\mu$sec | 3.32 $\mu$sec | 4.32 $\mu$sec |
| $\log_2 n$ | 2 $\mu$sec | 4 $\mu$sec | 8 $\mu$sec | 10 $\mu$sec | 20 $\mu$sec |
| $n$ | 4 $\mu$sec | 16 $\mu$sec | 256 $\mu$sec | 1.02 msec | 1.05 sec |
| $n\log_2 n$ | 8 $\mu$sec | 64 $\mu$sec | 2.05 msec | 1.02 msec | 21 sec |
| $n^2$ | 16 $\mu$sec | 256 $\mu$sec | 65.5 msec | 1.05 sec | 1.8 wk |
| $n^3$ | 64 $\mu$sec | 4.1 msec | 16.8 sec | 17.9 min | 36,559 yr |
| $2^n$ | 16 $\mu$sec | 65.5 msec | $3.7 \times 10^{63}$ yr | $5.7 \times 10^{294}$ yr | $2.1 \times 10^{315639}$ yr |

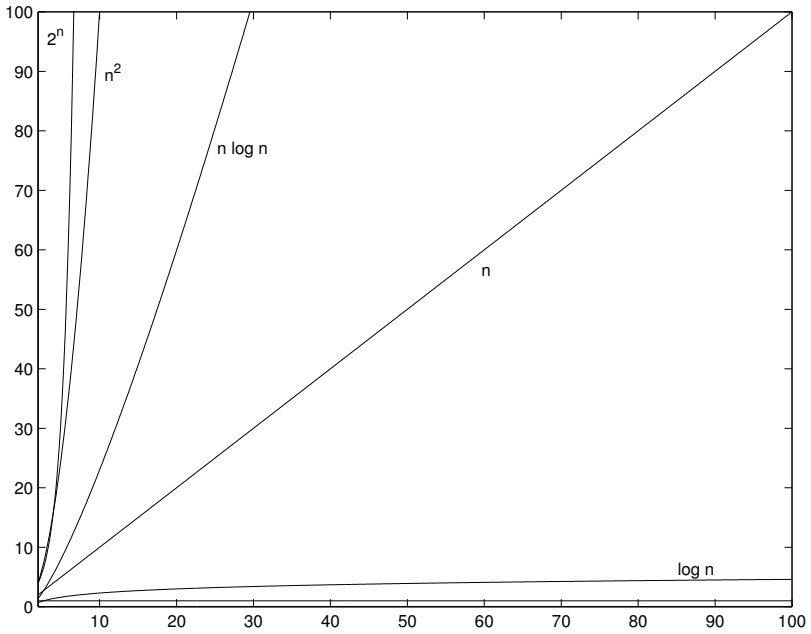
It is clear that, as the sizes of the problems get really big, there can be huge differences in the time it takes to run algorithms from different complexity classes. For algorithms with exponential complexity, $O(2^n)$, even modest sized problems have run times that are greater than the age of the universe (about $1.4 \times 10^{10}$ yr), and current computers rarely run uninterrupted for more than a few years. This is why complexity classes are so important – they tell us how feasible it is likely to be to run a program with a particular large number of data items. Typically, people do not worry much about complexity for sizes below 10, or maybe 20, but the above numbers make it clear why it is worth thinking about complexity classes where bigger applications are concerned.

Another useful way of thinking about growth classes involves considering how the compute time will vary if the problem size doubles. The following table shows what happens for the various complexity classes:

| $f(n)$ | If the size of the problem doubles then $f(n)$ will be | |
|---|---|---|
| 1 | the same, | $f(2n) = f(n)$ |
| $\log_2 \log_2 n$ | almost the same, | $\log_2 (\log_2 (2n)) = \log_2 (\log_2 (n) + 1)$ |
| $\log_2 n$ | more by $1 = \log_2 2$, | $f(2n) = f(n) + 1$ |
| $n$ | twice as big as before, | $f(2n) = 2f(n)$ |
| $n\log_2 n$ | a bit more than twice as big as before, | $2n\log_2 (2n) = 2(n\log_2 n) + 2n$ |
| $n^2$ | four times as big as before, | $f(2n) = 4f(n)$ |
| $n^3$ | eight times as big as before, | $f(2n) = 8f(n)$ |
| $2^n$ | the square of what it was before, | $f(2n) = (f(n))^2$ |

This kind of information can be very useful in practice. We can test our program on a problem that is a half or quarter or one eighth of the full size, and have a good idea of how long we will have to wait for the full size problem to finish. Moreover, that estimate won't be affected by any constant factors ignored in computing the growth class, or the speed of the particular computer it is run on.

The following graph plots some of the complexity class functions from the table. Note that although these functions are only defined on natural numbers, they are drawn as though they were defined for all real numbers, because that makes it easier to take in the information presented.




It is clear from these plots why the non-principal growth terms can be safely ignored when computing algorithm complexity.

## 5.5 Formal definition of complexity classes

We have noted that complexity classes are concerned with *growth*, and the tables and graph above have provided an idea of what different behaviours mean when it comes to growth. There we have chosen a representative for each of the complexity classes considered, but we have not said anything about just how 'representative' such an element is. Let us now consider a more formal definition of a 'big O' class:

**Definition.** A function $g$ belongs to the *complexity class* $O(f)$ if there is a number $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that for all $n \geq n_0$, we have that $g(n) \leq c * f(n)$. We say that the function $g$ is 'eventually smaller' than the function $c * f$.

It is not totally obvious what this implies. First, we do not need to know exactly *when* $g$ becomes smaller than $c * f$. We are only interested in the *existence* of $n_0$ such that, from then on, $g$ is smaller than $c * f$. Second, we wish to consider the efficiency of an algorithm independently of the speed of the computer that is going to execute it. This is why $f$ is multiplied by a constant $c$. The idea is that when we measure the time of the steps of a particular algorithm, we are not sure how long each of them takes. By definition, $g \in O(f)$ means that eventually (namely beyond the point $n_0$), the growth of $g$ will be at most as much as the growth of $c * f$. This definition also makes it clear that *constant factors* do not change the growth class (or $O$-class) of a function. Hence $C(n) = n^2$ is in the same growth class as $C(n) = 1/1000000 * n^2$ or $C(n) = 1000000 * n^2$. So we can write $O(n^2) = O(1000000 * n^2) = O(1/1000000 * n^2)$. Typically, however, we choose the *simplest* representative, as we did in the tables above. In this case it is $O(n^2)$.

The various classes we mentioned above are related as follows:

$$O(1) \subseteq O(\log_2 \log_2 n) \subseteq O(\log_2 (n)) \subseteq O(n) \subseteq O(n\log_2 n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

We only consider the principal growth class, so when adding functions from different growth classes, their sum will always be in the larger growth class. This allows us to simplify terms. For example, the growth class of $C(n) = 500000\log_2 n + 4n^2 + 0.3n + 100$ can be determined as follows. The summand with the largest growth class is $4n^2$ (we say that this is the 'principal sub-term' or 'dominating sub-term' of the function), and we are allowed to drop constant factors, so this function is in the class $O(n^2)$.

When we say that an algorithm 'belongs to' some class $O(f)$, we mean that it is *at most* as fast growing as $f$. We have seen that 'linear searching' (where one searches in a collection of data items which is unsorted) has linear complexity, i.e. it is in growth class $O(n)$. This holds for the *average case* as well as the *worst case*. The operations needed are comparisons of the item we are searching for with all the items appearing in the data collection. In the worst case, we have to check all $n$ entries until we find the right one, which means we make $n$ comparisons. On average, however, we will only have to check $n/2$ entries until we hit the correct one, leaving us with $n/2$ operations. Both those functions, $C(n) = n$ and $C(n) = n/2$ belong to the same complexity class, namely $O(n)$. However, it would be equally correct to say that the algorithm belongs to $O(n^2)$, since that class contains all of $O(n)$. But this would be *less informative*, and we would not say that an algorithm has quadratic complexity if we know that, in fact, it is linear. Sometimes it is difficult to be sure what the exact complexity is (as is the case with the famous NP = P problem), in which case one might say that an algorithm is 'at most', say, quadratic.

The issue of efficiency and complexity class, and their computation, will be a recurring feature throughout the chapters to come. We shall see that concentrating only on the complexity class, rather than finding exact complexity functions, can render the whole process of considering efficiency much easier. In most cases, we can determine the time complexity by a simple counting of the loops and tree heights. However, we will also see at least one case where that results in an overestimate, and a more exact computation is required.

# Chapter 6

# Trees

In computer science, a *tree* is a very general and powerful data structure that resembles a real tree. It consists of an ordered set of linked *nodes* in a connected *graph*, in which each node has at most one *parent* node, and zero or more *children* nodes with a specific order.

## 6.1 General specification of trees

Generally, we can specify a *tree* as consisting of *nodes* (also called *vertices* or *points*) and *edges* (also called *lines*, or, in order to stress the directedness, *arcs*) with a tree-like structure. It is usually easiest to represent trees pictorially, so we shall frequently do that. A simple example is given in Figure 6.1:

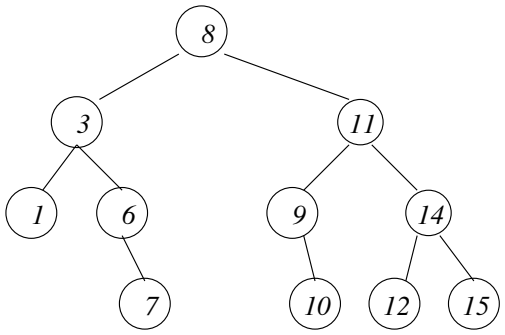



Figure 6.1: Example of a tree.

More formally, a *tree* can be defined as either the empty tree, or a node with a list of successor trees. Nodes are usually, though not always, *labelled* with a data item (such as a *number* or *search key*). We will refer to the label of a node as its *value*. In our examples, we will generally use nodes labelled by integers, but one could just as easily choose something else, e.g. strings of characters.

In order to talk rigorously about trees, it is convenient to have some terminology: There always has to be a unique 'top level' node known as the *root*. In Figure 6.1, this is the node labelled with 8. It is important to note that, in computer science, trees are normally displayed upside-down, with the root forming the top level. Then, given a node, every node on the next level 'down', that is connected to the given node via a branch, is a *child* of that node. In

Figure 6.1, the children of node 8 are nodes 3 and 11. Conversely, the node (there is at most one) connected to the given node (via an edge) on the level above, is its *parent*. For instance, node 11 is the parent of node 9 (and of node 14 as well). Nodes that have the same parent are known as *siblings* – siblings are, by definition, always on the same level.

If a node is the child of a child of ... of a another node then we say that the first node is a *descendent* of the second node. Conversely, the second node is an *ancestor* of the first node. Nodes which do not have any children are known as *leaves* (e.g., the nodes labelled with 1, 7, 10, 12, and 15 in Figure 6.1).

A *path* is a sequence of connected edges from one node to another. Trees have the property that for every node there is a unique path connecting it with the root. In fact, that is another possible definition of a tree. The *depth* or *level* of a node is given by the length of this path. Hence the root has level 0, its children have level 1, and so on. The maximal length of a path in a tree is also called the *height* of the tree. A path of maximal length always goes from the root to a leaf. The *size* of a tree is given by the number of nodes it contains. We shall normally assume that every tree is finite, though generally that need not be the case. The tree in Figure 6.1 has height 3 and size 11. A tree consisting of just of one node has height 0 and size 1. The empty tree obviously has size 0 and is defined (conveniently, though somewhat artificially) to have height $-1$.

Like most data structures, we need a set of *primitive operators* (constructors, selectors and conditions) to *build* and manipulate the trees. The details of those depend on the type and purpose of the tree. We will now look at some particularly useful types of tree.

## 6.2 Quad-trees

A *quadtree* is a particular type of tree in which each leaf-node is labelled by a value and each non-leaf node has exactly four children. It is used most often to partition a two dimensional space (e.g., a pixelated image) by recursively dividing it into four quadrants.

Formally, a quadtree can be defined to be either a single node with a number or value (e.g., in the range 0 to 255), or a node without a value but with four quadtree children: `lu`, `ll`, `ru`, and `rl`. It can thus be defined "*inductively*" by the following rules:

**Definition.** A *quad tree* is either

(Rule 1) a root node with a value, or

(Rule 2) a root node without a value and four quad tree children: `lu`, `ll`, `ru`, and `rl`.

in which Rule 1 is the "*base case*" and Rule 2 is the "*induction step*".

We say that a quadtree is *primitive* if it consists of a single node/number, and that can be tested by the corresponding *condition*:

- `isValue(qt)`, which returns true if quad-tree `qt` is a single node.

To *build* a quad-tree we have two *constructors*:

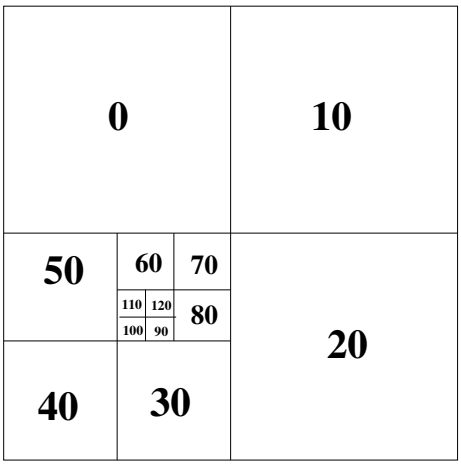
- `baseQT(value)`, which returns a single node quad-tree with label `value`.
- `makeQT(luqt, ruqt, llqt, rlqt)`, which builds a quad-tree from four constituent quad-trees `luqt`, `llqt`, `ruqt`, `rlqt`.

Then to extract components from a quad-tree we have four *selectors*:

- `lu(qt)`, which returns the left-upper quad-tree.
- `ru(qt)`, which returns the right-upper quad-tree.
- `ll(qt)`, which returns the left-lower quad-tree.
- `rl(qt)`, which returns the right-lower quad-tree.

which can be applied whenever `isValue(qt)` is false. For cases when `isValue(qt)` is true, we could define an operator `value(qt)` that returns the value, but conventionally we simply say that `qt` itself is the required value.

Quad-trees of this type are most commonly used to store grey-value pictures (with 0 representing black and 255 white). A simple example would be:




We can then create algorithms using the operators to perform useful manipulations of the representation. For example, we could rotate a picture `qt` by 180° using:

```
rotate(qt) {
   if ( isValue(qt) )
      return qt
   else return makeQT( rotate(rl(qt)), rotate(ll(qt)),
                       rotate(ru(qt)), rotate(lu(qt)) )
}
```

or we could compute average values by recursively averaging the constituent sub-trees.

There exist numerous variations of this general idea, such coloured quadtrees which store value-triples that represent colours rather than grey-scale, and *edge quad-trees* which store lines and allow curves to be represented with arbitrary precision.

## 6.3 Binary trees

Binary trees are the most common type of tree used in computer science. A *binary tree* is a tree in which every node has at most two children, and can be defined "inductively" by the following rules:

**Definition.** A *binary tree* is either

(Rule 1) the empty tree `EmptyTree`, or

(Rule 2) it consists of a node and two binary trees, the *left subtree* and *right subtree*.

Again, Rule 1 is the "*base case*" and Rule 2 is the "*induction step*". This definition may appear circular, but actually it is not, because the subtrees are always simpler than the original one, and we eventually end up with an empty tree.

You can imagine that the (infinite) collection of (finite) trees is created in a sequence of days. Day 0 is when you "get off the ground" by applying Rule 1 to get the empty tree. On later days, you are allowed to use any trees that you have created on earlier days to construct new trees using Rule 2. Thus, for example, on day 1 you can create exactly trees that have a root with a value, but no children (i.e. both the left and right subtrees are the empty tree, created at day 0). On day 2 you can use a new node with value, with the empty tree and/or the one-node tree, to create more trees. Thus, binary trees are the objects created by the above two rules in a finite number of steps. The height of a tree, defined above, is the number of days it takes to create it using the above two rules, where we assume that only one rule is used per day, as we have just discussed. (Exercise: work out the sequence of steps needed to create the tree in Figure 6.1 and hence prove that it is in fact a binary tree.)

## 6.4 Primitive operations on binary trees

The *primitive operators* for binary trees are fairly obvious. We have two *constructors* which are used to *build* trees:

- `EmptyTree`, which returns an empty tree,
- `MakeTree(v, l, r)`, which builds a binary tree from a root node with label `v` and two constituent binary trees `l` and `r`,

a *condition* to test whether a tree is empty:

- `isEmpty(t)`, which returns true if tree `t` is the `EmptyTree`,

and three *selectors* to break a non-empty tree into its constituent parts:

- `root(t)`, which returns the value of the root node of binary tree `t`,
- `left(t)`, which returns the left sub-tree of binary tree `t`,
- `right(t)`, which returns the right sub-tree of binary tree `t`.

These operators can be used to create all the algorithms we might need for manipulating binary trees.

For convenience though, it is often a good idea to define *derived operators* that allow us to write simpler, more readable algorithms. For example, we can define a derived constructor:

- `Leaf(v)` = `MakeTree(v, EmptyTree, EmptyTree)`

that creates a tree consisting of a single node with label `v`, which is the root and the unique leaf of the tree at the same time. Then the tree in Figure 6.1 can be constructed as:

```
t = MakeTree(8, MakeTree(3,Leaf(1),MakeTree(6,EmptyTree,Leaf(7))),
  MakeTree(11,MakeTree(9,EmptyTree,Leaf(10)),MakeTree(14,Leaf(12),Leaf(15))))
```

which is much simpler than the construction using the primitive operators:

```
t = MakeTree(8, MakeTree(3,MakeTree(1,EmptyTree,EmptyTree),
  MakeTree(6,EmptyTree,MakeTree(7,EmptyTree,EmptyTree))),
    MakeTree(11,MakeTree(9,EmptyTree,MakeTree(10,EmptyTree,EmptyTree)),
      MakeTree(14,MakeTree(12,EmptyTree,EmptyTree),
        MakeTree(15,EmptyTree,EmptyTree))))
```

Note that the selectors can only operate on non-empty trees. For example, for the tree `t` defined above we have

```
root(left(left(t)) = 1,
```

but the expression

```
root(left(left(left(t))))
```

does not make sense because

```
left(left(left(t))) = EmptyTree
```

and the empty tree does not have a root. In a language such as *Java*, this would typically raise an exception. In a language such as *C*, this would cause an unpredictable behaviour, but if you are lucky, a core dump will be produced and the program will be aborted with no further harm. When writing algorithms, we need to check the selector arguments using `isEmpty(t)` before allowing their use.

The following equations should be obvious from the primitive operator definitions:

```
root(MakeTree(v,l,r)) = v
left(MakeTree(v,l,r)) = l
right(MakeTree(v,l,r)) = r
isEmpty(EmptyTree) = true
isEmpty(MakeTree(v,l,r)) = false
```

The following makes sense only under the assumption that `t` is a non-empty tree:

```
MakeTree(root(t),left(t),right(t)) = t
```

It just says that if we break apart a non-empty tree and use the pieces to build a new tree, then we get an identical tree back.

It is worth emphasizing that the above specifications of quad-trees and binary trees are further examples of *abstract data types*: Data types for which we exhibit the constructors and destructors and describe their behaviour (using equations such as defined above for lists, stacks, queues, quad-trees and binary trees), but for which we explicitly hide the implementational details. The concrete data type used in an implementation is called a *data structure*. For example, the usual data structures used to implement the list and tree data types are records and pointers – but other implementations are possible.

The important advantage of abstract data types is that we can develop algorithms without having to worry about the details of the representation of the data or the implementation. Of course, everything will ultimately be represented as sequences of bits in a computer, but we clearly do not generally want to have to think in such low level terms.

## 6.5 The height of a binary tree

Binary trees don't have a simple relation between their size $n$ and height $h$. The maximum height of a binary tree with $n$ nodes is $(n-1)$, which happens when all non-leaf nodes have precisely one child, forming something that looks like a chain. On the other hand, suppose we have $n$ nodes and want to build from them a binary tree with minimal height. We can achieve this by 'filling' each successive level in turn, starting from the root. It does not matter where we place the nodes on the last (bottom) level of the tree, as long as we don't start adding to the next level before the previous level is full. Terminology varies, but we shall say that such trees are *perfectly balanced* or *height balanced*, and we shall see later why they are optimal for many of our purposes. Basically, if done appropriately, many important tree-based operations (such as searching) take as many steps as the height of the tree, so minimizing the height minimizes the time needed to perform those operations.

We can easily determine the maximum number of nodes that can fit into a binary tree of a given height $h$. Calling this size function $s(h)$, we obtain:

| $h$ | $s(h)$ |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 7 |
| 3 | 15 |

In fact, it seems fairly obvious that $s(h) = 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$. This hypothesis can be proved by induction using the definition of a binary tree as follows:

(a) The base case applies to the empty tree that has height $h = -1$, which is consistent with $s(-1) = 2^{-1+1} - 1 = 2^0 - 1 = 1 - 1 = 0$ nodes being stored.

(b) Then for the induction step, a tree of height $h+1$ has a root node plus two subtrees of height $h$. By the induction hypothesis, each subtree can store $s(h) = 2^{h+1} - 1$ nodes, so the total number of nodes that can fit in a height $h+1$ tree is $1 + 2 \times (2^{h+1} - 1) = 1 + 2^{h+2} - 2 = 2^{(h+1)+1} - 1 = s(h+1)$. It follows that if $s(h)$ is correct for the empty tree, which it was shown to be in the base case above, then it is correct for all $h$.

An obvious potential problem with any *proof by induction* like this, however, is the need to identify an induction hypothesis to start with, and that is not always easy.

Another way to proceed here would be to simply sum the series $s(h) = 1 + 2 + 4 + \cdots + 2^h$ algebraically to get the answer. Sometimes, however, the relevant series is too complicated to sum easily. An alternative is to try to identify two different expressions for $s(h+1)$ as a function of $s(h)$, and solve them for $s(h)$. Here, since level $h$ of a tree clearly has $2^h$ nodes, we can explicitly add in the $2^{h+1}$ nodes of the last level of the height $h+1$ tree to give

$$s(h+1) = s(h) + 2^{h+1}$$

Also, since a height $h+1$ tree is made up of a root node plus two trees of height $h$

$$s(h+1) = 1 + 2s(h)$$

Then subtracting the second equation from the first gives

$$s(h) = 2^{h+1} - 1$$

which is the required answer. From this we can get an expression for $h$

$$h = \log_2 (s + 1) - 1 \approx \log_2 s$$

in which the approximation is valid for large $s$.

Hence a perfectly balanced tree consisting of $n$ nodes has height approximately $\log_2 n$. This is good, because $\log_2 n$ is very small, even for relatively large $n$:

| $n$ | $\log_2 n$ |
|---:|---:|
| 2 | 1 |
| 32 | 5 |
| 1,024 | 10 |
| 1,048,576 | 20 |

We shall see later how we can use binary trees to hold data in such a way that any search has at most as many steps as the height of the tree. Therefore, for perfectly balanced trees we can reduce the search time considerably as the table demonstrates. However, it is not always easy to create perfectly balanced trees, as we shall also see later.

## 6.6 The size of a binary tree

Usually a binary tree will not be perfectly balanced, so we will need an algorithm to determine its *size*, i.e. the number of nodes it contains.

This is easy if we use *recursion*. The terminating case is very simple: the empty tree has size 0. Otherwise, any binary tree will always be assembled from a root node, a left sub-tree `l`, and a right sub-tree `r`, and its size will be the sum of the sizes of its components, i.e. 1 for the root, plus the size of `l`, plus the size of `r`. We have already defined the primitive operator `isEmpty(t)` to check whether a binary tree `t` is empty, and the selectors `left(t)` and `right(t)` which return the left and right sub-trees of binary tree `t`. Thus we can easily define the procedure `size(t)`, which takes a binary tree `t` and returns its size, as follows:

```
size(t) {
   if ( isEmpty(t) )
      return 0
   else return (1 + size(left(t)) + size(right(t)))
}
```

This recursively processes the whole tree, and we know it will terminate because the trees being processed get smaller with each call, and will eventually reach an empty tree which returns a simple value.

## 6.7 Implementation of trees

The natural way to *implement* trees is in terms of *records* and *pointers*, in a similar way to how linked lists were represented as two-cells consisting of a pointer to a list element and a pointer to the next two-cell. Obviously, the details will depend on how many children each node can have, but trees can generally be represented as data structures consisting of a pointer to the root-node content (if any) and pointers to the children sub-trees. The inductive definition

of trees then allows recursive algorithms on trees to operate efficiently by simply passing the pointer to the relevant root-node, rather than having to pass complete copies of whole trees. How data structures and pointers are implemented in different programming languages will vary, of course, but the general idea is the same.

A binary tree can be implemented as a data record for each node consisting simply of the node value and two pointers to the children nodes. Then `MakeTree` simply creates a new data record of that form, and `root`, `left` and `right` simply read out the relevant contents of the record. The absence of a child node can be simply represented by a Null Pointer.

## 6.8 Recursive algorithms

Some people have difficulties with *recursion*. A source of confusion is that it appears that "the algorithm calls itself" and it might therefore get confused about what it is operating on. This way of putting things, although suggestive, can be misleading. The algorithm itself is a passive entity, which actually cannot do anything at all, let alone call itself. What happens is that a *processor* (which can be a machine or a person) *executes* the algorithm. So what goes on when a processor executes a recursive algorithm such as the `size(t)` algorithm above? An easy way of understanding this is to imagine that whenever a recursive call is encountered, new processors are given the task with a copy of the same algorithm.

For example, suppose that John (the first processor in this task) wants to compute the size of a given tree `t` using the above recursive algorithm. Then, according to the above algorithm, John first checks whether it is empty. If it is, he simply returns zero and finishes his computation. If it isn't empty, then his tree `t` must have left and right subtrees `l` and `r` (which may, or may not, be empty) and he can extract them using the selectors `left(t)` and `right(t)`. He can then ask two of his students, say Steve and Mary, to execute the same algorithm, but for the trees `l` and `r`. When they finish, say returning results $m$ and $n$ respectively, he computes and returns $1+m+n$, because his tree has a root node in addition to the left and right sub-trees. If Steve and Mary aren't given empty trees, they will themselves have to delegate executions of the same algorithm, with their sub-trees, to other people. Thus, the algorithm is not calling itself. What happens, is that there are many people running their own copies of the same algorithm on different trees.

In this example, in order to make things understandable, we assumed that each person executes a single copy of the algorithm. However, the same processor, with some difficulty, can impersonate several processors, in such a way that it achieves the same result as the execution involving many processors. This is achieved via the use of a *stack* that keeps track of the various positions of the same algorithm that are currently being executed – but this knowledge is not needed for our purposes.

Note that there is nothing to stop us keeping count of the recursions by passing integers along with any data structures being operated on, for example:

```
function(int n, tree t) {
   // terminating condition and return
      .
   // procedure details
      .
   return function(n-1, t2)
}
```

so we can do something `n` times, or look for the `n`th item, etc. The classic example is the recursive factorial function:

```
factorial(int n) {
   if ( n == 0 ) return 1
   return n*factorial(n-1)
}
```

Another example, with two termination or base-case conditions, is a direct implementation of the recursive definition of *Fibonacci numbers* (see Appendix A.5):

```
F(int n) {
   if ( n == 0 ) return 0
   if ( n == 1 ) return 1
   return F(n-1) + F(n-2)
}
```

though this is an extremely inefficient algorithm for computing these numbers. Exercise: Show that the time complexity of this algorithm is $O(2^n)$, and that there exists a straightforward iterative algorithm that has only $O(n)$ time complexity. Is it possible to create an $O(n)$ recursive algorithm to compute these numbers?

In most cases, however, we won't need to worry about counters, because the relevant data structure has a natural end point condition, such as `isEmpty(x)`, that will bring the recursion to an end.

# Chapter 7

# Binary Search Trees

We now look at Binary Search Trees, which are a particular type of binary tree that provide an efficient way of *storing* data that allows particular items to be found as quickly as possible. Then we consider further elaborations of these trees, namely AVL trees and B-trees, which operate more efficiently at the expense of requiring more sophisticated algorithms.

## 7.1 Searching with arrays or lists

As we have already seen in Chapter 4, many computer science applications involve *searching* for a particular item in a collection of data. If the data is stored as an unsorted array or list, then to find the item in question, one obviously has to check each entry in turn until the correct one is found, or the collection is exhausted. On average, if there are $n$ items, this will take $n/2$ checks, and in the worst case, all $n$ items will have to be checked. If the collection is large, such as all items accessible via the internet, that will take too much time. We also saw that if the items are sorted before storing in an array, one can perform binary search which only requires $\log_2 n$ checks in the average and worst cases. However, that involves an overhead of sorting the array in the first place, or maintaining a sorted array if items are inserted or deleted over time. The idea here is that, with the help of binary trees, we can speed up the storing and search process without needing to maintain a sorted array.

## 7.2 Search keys

If the items to be searched are labelled by comparable *keys*, one can order them and store them in such a way that they are *sorted* already. Being 'sorted' may mean different things for different keys, and which key to choose is an important design decision.

In our examples, the search keys will, for simplicity, usually be integer numbers (such as student ID numbers), but other choices occur in practice. For example, the comparable keys could be words. In that case, comparability usually refers to the *alphabetical* order. If `w` and `t` are words, we write `w` $<$ `t` to mean that `w` precedes `t` in the alphabetical order. If `w` = *bed* and `t` = *sky* then the relation `w` $<$ `t` holds, but this is not the case if `w` = *bed* and `t` = *abacus*. A classic example of a collection to be searched is a dictionary. Each entry of the dictionary is a pair consisting of a word and a definition. The definition is a sequence of words and punctuation symbols. The search key, in this example, is the word (to which a definition is attached in the dictionary entry). Thus, *abstractly*, a dictionary is a sequence of

entries, where an entry is a pair consisting of a word and its definition. This is what matters from the point of view of the search algorithms we are going to consider. In what follows, we shall concentrate on the search keys, but should always bear in mind that there is usually a more substantial data entry associated with it.

Notice the use of the word "abstract" here. What we mean is that we abstract or remove any details that are irrelevant from the point of view of the algorithms. For example, a dictionary usually comes in the form of a book, which is a sequence of pages – but for us, the distribution of dictionary entries into pages is an accidental feature of the dictionary. All that matters for us is that the dictionary is a sequence of entries. So "abstraction" means "getting rid of irrelevant details". For our purposes, only the search key is important, so we will ignore the fact that the entries of the collection will typically be more complex objects (as in the example of a dictionary or a phone book).

Note that we should always employ the data structure to hold the items which performs best for the typical application. There is no easy answer as to what the best choice is – the particular circumstances have to be inspected, and a decision has to be made based on that. However, for many applications, the kind of *binary trees* we studied in the last chapter are particularly useful here.

## 7.3 Binary search trees

The solution to our search problem is to store the collection of data to be searched using a binary tree in such a way that searching for a particular item takes minimal effort. The underlying idea is simple: At each tree node, we want the value of that node to either tell us that we have found the required item, or tell us which of its two subtrees we should search for it in. For the moment, we shall assume that all the items in the data collection are distinct, with different search keys, so each possible node value occurs at most once, but we shall see later that it is easy to relax this assumption. Hence we define:

**Definition.** A *binary search tree* is a binary tree that is either empty or satisfies the following conditions:

- All values occurring in the left subtree are smaller than that of the root.
- All values occurring in the right subtree are larger than that of the root.
- The left and right subtrees are themselves binary search trees.

So this is just a particular type of binary tree, with node values that are the search keys. This means we can *inherit* many of the operators and algorithms we defined for general binary trees. In particular, the primitive operators `MakeTree(v,l,r)`, `root(t)`, `left(t)`, `right(t)` and `isEmpty(t)` are the same – we just have to maintain the additional node value ordering.

## 7.4 Building binary search trees

When building a binary search tree, one naturally starts with the root and then adds further new nodes as needed. So, to insert a new value $v$, the following cases arise:

- If the given tree is empty, then simply assign the new value $v$ to the root, and leave the left and right subtrees empty.

- If the given tree is non-empty, then insert a node with value $v$ as follows:
  - If $v$ is smaller than the value of the root: insert $v$ into the left sub-tree.
  - If $v$ is larger than the value of the root: insert $v$ into the right sub-tree.
  - If $v$ is equal to the value of the root: report a violated assumption.

Thus, using the primitive binary tree operators, we have the procedure:

```
insert(v,bst) {
   if ( isEmpty(bst) )
      return MakeTree(v, EmptyTree, EmptyTree)
   elseif ( v < root(bst) )
      return MakeTree(root(bst), insert(v,left(bst)), right(bst))
   elseif ( v > root(bst) )
      return MakeTree(root(bst), left(bst), insert(v,right(bst)))
   else error('Error: violated assumption in procedure insert.')
}
```

which inserts a node with value `v` into an existing binary search tree `bst`. Note that the node added is always a leaf. The resulting tree is once again a binary search tree. This can be proved rigorously via an inductive argument.

Note that this procedure creates a new tree out of a given tree `bst` and new value `v`, with the new value inserted at the right position. The original tree `bst` is not modified, it is merely inspected. However, when the tree represents a large database, it would clearly be more efficient to modify the given tree, rather than to construct a whole new tree. That can easily be done by using *pointers*, similar to the way we set up linked lists. For the moment, though, we shall not concern ourselves with such implementational details.

## 7.5 Searching a binary search tree

Searching a binary search tree is not dissimilar to the process performed when inserting a new item. We simply have to compare the item being looked for with the root, and then keep 'pushing' the comparison down into the left or right subtree depending on the result of each root comparison, until a match is found or a leaf is reached.

Algorithms can be expressed in many ways. Here is a concise description in words of the search algorithm that we have just outlined:

> In order to search for a value `v` in a binary search tree `t`, proceed as follows. If `t` is empty, then `v` does not occur in `t`, and hence we stop with `false`. Otherwise, if `v` is equal to the root of `t`, then `v` does occur in `t`, and hence we stop returning `true`. If, on the other hand, `v` is smaller than the root, then, by definition of a binary search tree, it is enough to search the left sub-tree of `t`. Hence replace `t` by its left sub-tree and carry on in the same way. Similarly, if `v` is bigger than the root, replace `t` by its right sub-tree and carry on in the same way.

Notice that such a description of an algorithm embodies both the steps that need to be carried out *and* the reason why this gives a correct solution to the problem. This way of describing algorithms is very common when we do not intend to run them on a computer.

When we do want to run them, we need to provide a more precise specification, and would normally write the algorithm in pseudocode, such as the following recursive procedure:

```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

Each recursion restricts the search to either the left or right subtree as appropriate, reducing the search tree height by one, so the algorithm is guaranteed to terminate eventually.

In this case, the recursion can easily be transformed into a while-loop:

```
isIn(value v, tree t) {
   while ( (not isEmpty(t)) and (v != root(t)) )
      if (v < root(t) )
         t = left(t)
      else
         t = right(t)
   return ( not isEmpty(t) )
}
```

Here, each iteration of the while-loop restricts the search to either the left or right subtree as appropriate. The only way to leave the loop is to have found the required value, or to only have an empty tree remaining, so the procedure only needs to return whether or not the final tree is empty.

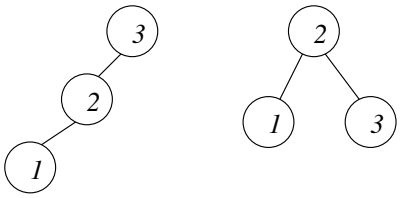
In practice, we often want to have more than a simple `true`/`false` returned. For example, if we are searching for a student ID, we usually want a pointer to the full record for that student, not just a confirmation that they exist. In that case, we could store a record pointer associated with the search key (ID) at each tree node, and return the record pointer or a null pointer, rather than a simple `true` or `false`, when an item is found or not found. Clearly, the basic tree structures we have been discussing can be elaborated in many different ways like this to form whatever data-structure is most appropriate for the problem at hand, but, as noted above, we can abstract out such details for current purposes.

## 7.6 Time complexity of insertion and search

As always, it is important to understand the time complexity of our algorithms. Both item insertion and search in a binary search tree will take at most as many comparisons as the height of the tree plus one. At worst, this will be the number of nodes in the tree. But how many comparisons are required on average? To answer this question, we need to know the average height of a binary search tree. This can be calculated by taking all possible binary search trees of a given size $n$ and measuring each of their heights, which is by no means an

easy task. The trouble is that there are many ways of building the same binary search tree by successive insertions.

As we have seen above, perfectly balanced trees achieve minimal height for a given number of nodes, and it turns out that the more balanced a tree, the more ways there are of building it. This is demonstrated in the figure below:




The only way of getting the tree on the left hand side is by inserting 3, 2, 1 into the empty tree in that order. The tree on the right, however, can be reached in two ways: Inserting in the order 2, 1, 3 or in the order 2, 3, 1. Ideally, of course, one would only use well-balanced trees to keep the height minimal, but they do not have to be perfectly balanced to perform better than binary search trees without restrictions.

Carrying out exact tree height calculations is not straightforward, so we will not do that here. However, if we assume that all the possible orders in which a set of $n$ nodes might be inserted into a binary search tree are equally likely, then the average height of a binary search tree turns out to be $O(\log_2 n)$. It follows that the average number of comparisons needed to search a binary search tree is $O(\log_2 n)$, which is the same complexity we found for binary search of a sorted array. However, inserting a new node into a binary search tree also depends on the tree height and requires $O(\log_2 n)$ steps, which is better than the $O(n)$ complexity of inserting an item into the appropriate point of a sorted array.

Interestingly, the average height of a binary search tree is quite a bit better than the average height of a general binary tree consisting of the same $n$ nodes that have not been built into a binary search tree. The average height of a general binary tree is actually $O(\sqrt{n})$. The reason for that is that there is a relatively large proportion of high binary trees that are not valid binary search trees.

## 7.7 Deleting nodes from a binary search tree

Suppose, for some reason, an item needs to be removed or deleted from a binary search tree. It would obviously be rather inefficient if we had to rebuild the remaining search tree again from scratch. For $n$ items that would require $n$ steps of $O(\log_2 n)$ complexity, and hence have overall time complexity of $O(n\log_2 n)$. By comparison, deleting an item from a sorted array would only have time complexity $O(n)$, and we certainly want to do better than that. Instead, we need an algorithm that produces an updated binary search tree more efficiently. This is more complicated than one might assume at first sight, but it turns out that the following algorithm works as desired:

- If the node in question is a leaf, just remove it.
- If only one of the node's subtrees is non-empty, 'move up' the remaining subtree.
- If the node has two non-empty sub-trees, find the 'left-most' node occurring in the right sub-tree (this is the smallest item in the right subtree). Use this node to overwrite the

one that is to be deleted. Replace the left-most node by its right subtree, if this exists; otherwise just delete it.

The last part works because the left-most node in the right sub-tree is guaranteed to be bigger than all nodes in the left sub-tree, smaller than all the other nodes in the right sub-tree, and have no left sub-tree itself. For instance, if we delete the node with value `11` from the tree in Figure 6.1, we get the tree displayed in Figure 7.1.

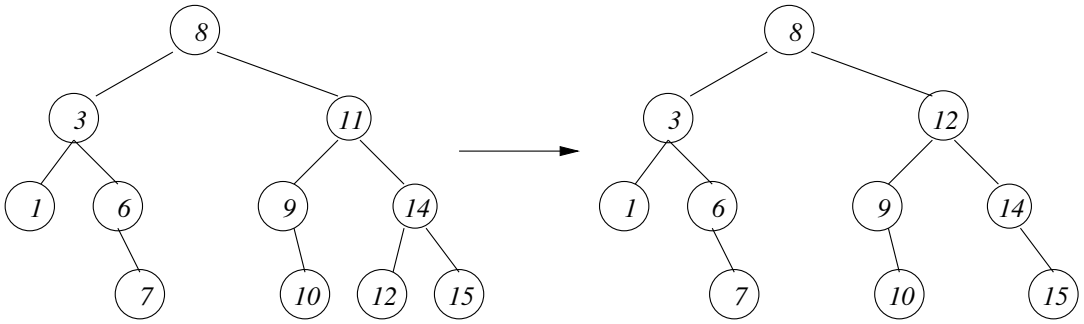



Figure 7.1: Example of node deletion in a binary search tree.

In practice, we need to turn the above algorithm (specified in words) into a more detailed algorithm specified using the primitive binary tree operators:

```
delete(value v, tree t) {
  if ( isEmpty(t) )
     error('Error: given item is not in given tree')
  else
     if ( v < root(t) )    // delete from left sub-tree
        return MakeTree(root(t), delete(v,left(t)), right(t));
     else if ( v > root(t) )     // delete from right sub-tree
        return MakeTree(root(t), left(t), delete(v,right(t)));
     else    // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
           return right(t)
        elseif ( isEmpty(right(t)) )
           return left(t)
        else    // difficult case with both subtrees non-empty
           return MakeTree(smallestNode(right(t)), left(t),
                                              removeSmallestNode(right(t))
}
```

If the empty tree condition is met, it means the search item is not in the tree, and an appropriate error message should be returned.

The `delete` procedure uses two sub-algorithms to find and remove the smallest item of a given sub-tree. Since the relevant sub-trees will always be non-empty, these sub-algorithms can be written with that *precondition*. However, it is always the responsibility of the programmer to ensure that any preconditions are met whenever a given procedure is used, so it is important to say explicitly what the preconditions are. It is often safest to start each procedure with a

check to determine whether the preconditions are satisfied, with an appropriate *error message* produced when they are not, but that may have a significant time cost if the procedure is called many times. First, to find the smallest node, we have:

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}
```

which uses the fact that, by the definition of a binary search tree, the smallest node of `t` is the left-most node. It recursively looks in the left sub-tree till it reaches an empty tree, at which point it can return the root. The second sub-algorithm uses the same idea:

```
removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

except that the remaining tree is returned rather than the smallest node.

These procedures are further examples of recursive algorithms. In each case, the recursion is guaranteed to terminate, because every recursive call involves a smaller tree, which means that we will eventually find what we are looking for or reach an empty tree.

It is clear from the algorithm that the deletion of a node requires the same number of steps as searching for a node, or inserting a new node, i.e. the average height of the binary search tree, or $O(\log_2 n)$ where $n$ is the total number of nodes on the tree.

## 7.8 Checking whether a binary tree is a binary search tree

Building and using binary search trees as discussed above is usually enough. However, another thing we sometimes need to do is *check* whether or not a given binary tree is a binary search tree, so we need an algorithm to do that. We know that an empty tree is a (trivial) binary search tree, and also that all nodes in the left sub-tree must be smaller than the root and themselves form a binary search tree, and all nodes in the right sub-tree must be greater than the root and themselves form a binary search tree. Thus the obvious algorithm is:

```
isbst(tree t) {
   if ( isEmpty(t) )
      return true
   else
      return ( allsmaller(left(t),root(t)) and isbst(left(t))
                  and allbigger(right(t),root(t)) and isbst(right(t)) )
}
```

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and allsmaller(left(t),v)
                                        and allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) > v) and allbigger(left(t),v)
                                        and allbigger(right(t),v) )
}
```

However, the simplest or most obvious algorithm is not always the most efficient. Exercise: identify what is inefficient about this algorithm, and formulate a more efficient algorithm.

## 7.9 Sorting using binary search trees

*Sorting* is the process of putting a collection of items in order. We shall formulate and discuss many sorting algorithms later, but we are already able to present one of them.

The node values stored in a binary search tree can be printed in ascending order by recursively printing each left sub-tree, root, and right sub-tree in the right order as follows:

```
printInOrder(tree t) {
   if ( not isEmpty(t) ) {
      printInOrder(left(t))
      print(root(t))
      printInOrder(right(t))
   }
}
```

Then, if the collection of items to be sorted is given as an array `a` of known size `n`, they can be printed in sorted order by the algorithm:

```
sort(array a of size n) {
   t = EmptyTree
   for i = 0,1,...,n-1
      t = insert(a[i],t)
   printInOrder(t)
}
```

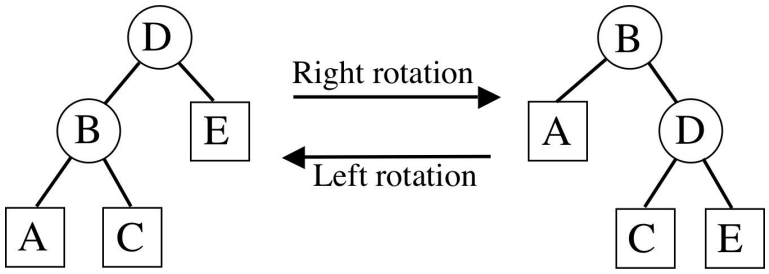
which starts with an empty tree, inserts all the items into it using `insert`(`v`, `t`) to give a binary search tree, and then prints them in order using `printInOrder`(`t`). Exercise: modify this algorithm so that instead of printing the sorted values, they are put back into the original array in ascending order.

## 7.10 Balancing binary search trees

If the items are added to a binary search tree in random order, the tree tends to be fairly well balanced with height not much more than $\log_2 n$. However, there are many situations where the added items are not in random order, such as when adding new student IDs. In the extreme case of the new items being added in ascending order, the tree will be one long branch off to the right, with height $n \gg \log_2 n$.

If all the items to be inserted into a binary search tree are already sorted, it is straightforward to build a perfectly balanced binary tree from them. One simply has to recursively build a binary tree with the middle (i.e., median) item as the root, the left subtree made up of the smaller items, and the right subtree made up of the larger items. This idea can be used to *rebalance* any existing binary search tree, because the existing tree can easily be output into a sorted array as discussed in Section 7.9. Exercise: Write an algorithm that rebalances a binary search tree in this way, and work out its time complexity.

Another way to avoid unbalanced binary search trees is to *rebalance* them from time to time using *tree rotations*. Such tree rotations are best understood as follows: Any binary search tree containing at least two nodes can clearly be drawn in one of the two forms:




where B and D are the required two nodes to be rotated, and A, C and E are binary search sub-trees (any of which may be empty). The two forms are related by left and right tree rotations which clearly preserve the binary search tree property. In this case, any nodes in sub-tree A would be shifted up the tree by a right rotation, and any nodes in sub-tree E would be shifted up the tree by a left rotation. For example, if the left form had A consisting of two nodes, and C and E consisting of one node, the height of the tree would be reduced by one and become *perfectly balanced* by a right tree rotation.

Typically, such tree rotations would need to be applied to many different sub-trees of a full tree to make it perfectly balanced. For example, if the left form had C consisting of two nodes, and A and E consisting of one node, the tree would be balanced by first performing a left rotation of the A-B-C sub-tree, followed by a right rotation of the whole tree. In practice, finding suitable sequences of appropriate tree rotations to rebalance an arbitrary binary search tree is not straightforward, but it is possible to formulate systematic balancing algorithms that are more efficient than outputting the whole tree and rebuilding it.

## 7.11 Self-balancing AVL trees

Self-balancing binary search trees avoid the problem of unbalanced trees by automatically *rebalancing* the tree throughout the insertion process to keep the height close to $\log_2 n$ at each stage. Obviously, there will be a cost involved in such rebalancing, and there will be a

trade-off between the time involved in rebalancing and the time saved by the reduced height of the tree, but generally it is worthwhile.

The earliest type of self-balancing binary search tree was the *AVL tree* (named after its inventors G.M. Adelson-Velskii and E.M. Landis). These maintain the difference in heights of the two sub-trees of all nodes to be at most one. This requires the tree to be periodically rebalanced by performing one or more *tree rotations* as discussed above, but the complexity of insertion, deletion and search remain at $O(\log_2 n)$.

The general idea is to keep track of the *balance factor* for each node, which is the height of the left sub-tree minus the height of the right sub-tree. By definition, all the nodes in an AVL-tree will have a balance factor in the integer range $[-1, 1]$. However, insertion or deletion of a node could leave that in the wider range $[-2, 2]$ requiring a tree-rotation to bring it back into AVL form. Exercise: Find some suitable algorithms for performing efficient AVL tree rotations. Compare them with other self-balancing approaches such as *red-black trees*.

## 7.12 B-trees

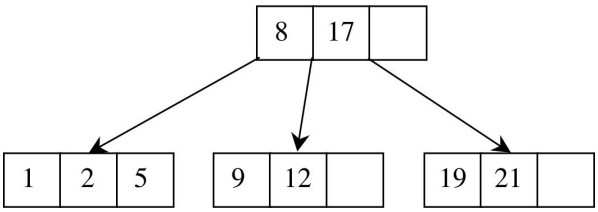
A *B-tree* is a generalization of a self-balancing binary search tree in which each node can hold more than one search key and have more than two children. The structure is designed to allow more efficient self-balancing, and offers particular advantages when the node data needs to be kept in external storage such as disk drives. The standard (Knuth) definition is:

**Definition.** A *B-tree* of order $m$ is a tree which satisfies the following conditions:

- Every node has at most $m$ children.
- Every non-leaf node (except the root node) has at least $m/2$ children.
- The root node, if it is not a leaf node, has at least two children.
- A non-leaf node with $c$ children contains $c-1$ search keys which act as separation values to divide its sub-trees.
- All leaf nodes appear in the same level, and carry information.

There appears to be no definitive answer to the question of what the "B" in "B-Tree" stands for. It is certainly not "Binary", but it could equally well be "balanced", "broad" or "bushy", or even "Boeing" because they were invented by people at Boeing Research Labs.

The standard representation of simple order 4 example with 9 search keys would be:




The search keys held in each node are ordered (e.g., 1, 2, 5 in the example), and the non-leaf node's search keys (i.e., the items 8 and 17 in the example) act as separation values to divide