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	1.00×10^0	1.00×10^0
$\log_2 \log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$\log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	2.56×10^2	1.02×10^3	1.05×10^6
$n\log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	4096	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	16	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{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 μsec	1 μsec	1 μsec	1 μsec	1 μsec
$\log_2 \log_2 n$	1 μsec	2 μsec	3 μsec	3.32 μsec	4.32 μsec
$\log_2 n$	2 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	4 μsec	16 μsec	256 μsec	1.02 msec	1.05 sec
$n\log_2 n$	8 μsec	64 μsec	2.05 msec	1.02 msec	21 sec
n^2	16 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wk
n^3	64 μsec	4.1 msec	16.8 sec	17.9 min	36,559 yr
2^n	16 μsec	65.5 msec	3.7×10^{63} yr	5.7×10^{294} yr	2.1×10^{315639} yr

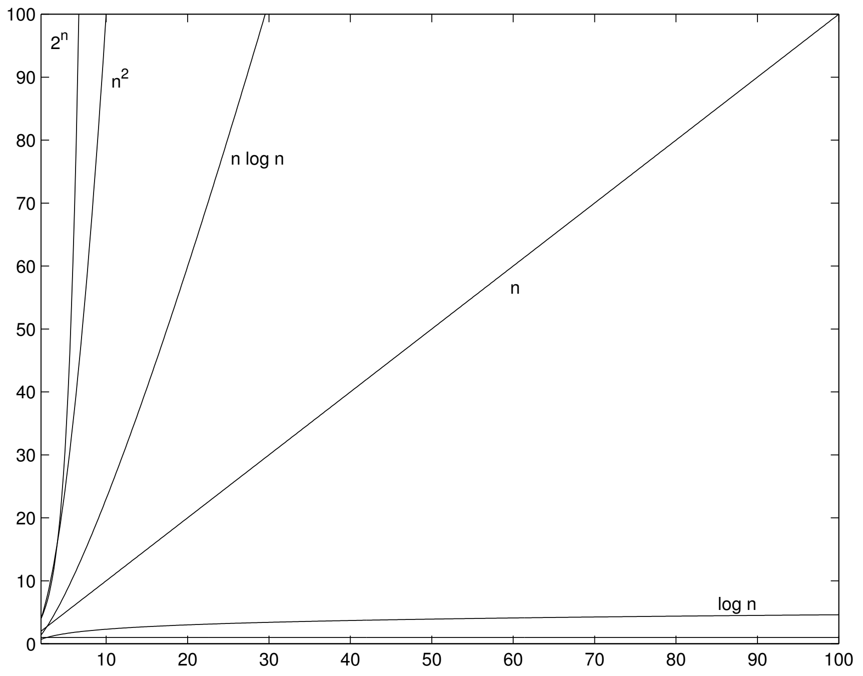
It is clear that, as the sizes of the problems get really big, there can be huge differences in the time it takes to run algorithms from different complexity classes. For algorithms with exponential complexity, $O(2^n)$, even modest sized problems have run times that are greater than the age of the universe (about 1.4×10^{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:

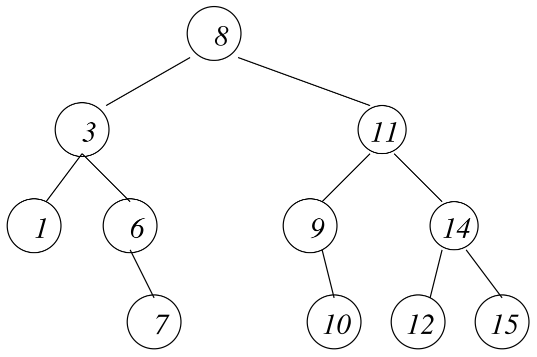


Figure 6.1: Example of a tree.

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`.

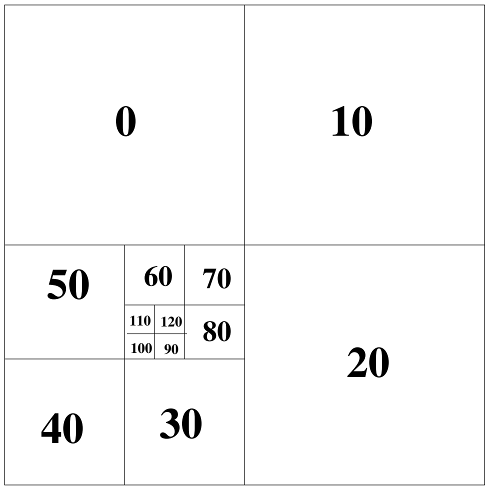
- `makeQT(luqt, ruqt, llqt, rlqt)`, which builds a quad-tree from four constituent quad-trees `luqt`, `llqt`, `ruqt`, `rlqt`.

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) )
}
```

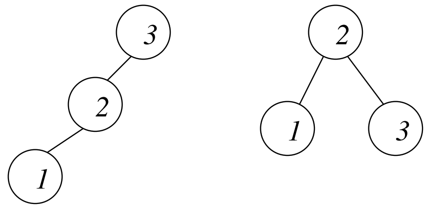
Here, each iteration of the while-loop restricts the search to either the left or right subtree as appropriate. The only way to leave the loop is to have found the required value, or to only have an empty tree remaining, so the procedure only needs to return whether or not the final tree is empty.

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