Design and Analysis of Purely Functional Programs

Christian Rinderknecht

Contents

Foreword

This book addresses *a priori* different audiences whose common interest is functional programming.

For undergraduate students, we offer a very progressive introduction to functional programming, with long developments about algorithms on stacks and some kinds of binary trees. We also study memory allocation through aliasing (dynamic data-sharing), the role of the control stack and the heap, automatic garbage collection (GC), the optimisation of tail calls and the total allocated memory. Program transformation into tail form, higher-order functions and continuation-passing style are advanced subjects presented in the context of the programming language Erlang. We give a technique for translating short functional programs to Java.

For postgraduate students, each functional program is associated with the mathematical analysis of its minimum and maximum cost (efficiency), but also its average and amortised cost. The peculiarity of our approach is that we use elementary concepts (elementary calculus, induction, discrete mathematics) and we systematically seek explicit bounds in order to draw asymptotic equivalences. Indeed, too often textbooks only introduce Bachmann notation $\mathcal{O}(\cdot)$ for the dominant term of the cost, which provides little information and may confuse beginners. Furthermore, we cover in detail proofs of properties like correctness, termination and equivalence.

For the professionals who do not know functional languages and who must learn how to program with the language XSLT, we propose an introduction which dovetails the part dedicated to undergraduate students. The reason of this unusual didactic choice lies on the observation that XSLT is rarely taught in college, therefore programmers who have not been exposed to functional programming face the two challenges of learning a new paradigm and use XML for programming: whereas the former puts forth recursion, the latter obscures it because of the inherent verbosity of XML. By learning first an abstract functional language, and then XML, we hope for a transfer of skills towards the design and implementation in XSLT without mediation.

This book has also been written with the hope of enticing the reader into theoretical computing, like programming language semantics, formal logic, lattice path counting and analytic combinatorics.

I thank Nachum Dershowitz, François Pottier, Sri Gopal Mohanty, Walter Böhm, Philippe Flajolet, Francisco Javier Barón López, Ham Jun-Wu and Kim Sung Ho for their technical help.

Most of this book was written while I was working at Konkuk University (Seoul, Republic of Korea) with the Department of Internet and Multimedia, from 2005 to 2012. Some parts were added while I was visiting the Department of Programming Languages and Compilers at the Eötvös Loránd University (Budapest, Hungary) — also known as ELTE —, from 2013 to 2014.

Please inform me of any error at `rinderknecht@free.fr`.

Budapest, Hungary.
9th May 2025.

Christian Rinderknecht

Part II

Arborescent Structures

Chapter 6

Catalan Trees

In part I, we dealt with linear structures, like stacks and queues, but, to really understand programs operating on such structures, we needed the concept of tree. This is why we introduced very early on abstract syntax trees, directed acyclic graphs (e.g., FIGURE 1.4 on page 7), comparison trees (e.g., FIGURE 2.41 on page 90), binary trees (e.g., FIGURE 2.43 on page 91), proof trees (e.g., FIGURE 3.2a on page 98), evaluation trees (e.g., FIGURE 3.6 on page 108) and merge trees (e.g., FIGURE 4.6 on page 126). Those trees were meta-objects, or concepts, used to understand the linear structure at hand.

In this chapter, we take a more abstract point of view and we consider the general class of *Catalan trees*, or general trees. We will study them as mathematical objects with the aim to transfer our results to the trees used as data structures to implement algorithms. In particular, we will be interested in measuring, counting them, and determining some average parameters relative to their shape, the reason being that knowing what a random tree looks like will tell us something about the cost of traversing it in different ways.

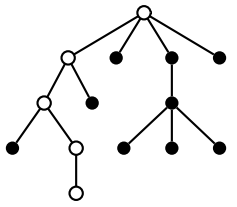


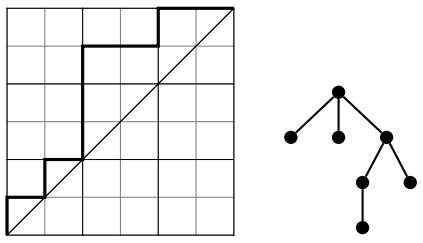
Figure 6.1: Catalan tree of height 4

Catalan trees are a special kind of graph, that is, an object made of *nodes* (also called *vertices*) connected by *edges*, without orientation (only the connection matters). What makes Catalan trees is the distinction of a node, called the *root*, and the absence of cycles, that is, closed paths made of nodes successively connected. Catalan trees are often called *ordered trees*, or *planted plane trees*, in graph theory, and *unranked trees*, *n-ary trees*, or *rose trees* in programming theory. An example is given in FIGURE 6.1. Note how the root is the topmost node and has four subtrees, whose

root are called *children*, given in order. The nodes drawn as white disks (∘) make up a maximal path starting from the root (the number of nodes along it is maximal). The ending node has no children; there are actually 8 such nodes in total, called the *leaves*. The number of edges connecting white disks is the *height* of the Catalan tree (there may be several maximal paths of same length), so the given example has height 4.

Programmers implement Catalan trees as a data structure, *e.g.*, using XML, in which case some information is stored in the nodes and its retrieval may – in the worst case – require the reaching of a leaf. The maximum cost of a search is thus proportional to the height of the tree and the determination of the average height becomes relevant when performing a series of random searches (Vitter and Flajolet, 1990). For this kind of analysis, we need first to find the number of Catalan trees with a given size. There are two common measures for the size: either we quantify the trees by their number of nodes or we count the edges. In fact, using one or the other is a matter of convenience or style: there are n edges if there are $n+1$ nodes, simply because each node, save the root, has one parent. It is often the case that formulas about Catalan trees are a bit simpler when using the number of edges, so this will be our measure of size in this chapter.

6.1 Enumeration



(a) Dyck path (b) Catalan tree

Figure 6.2: Bijection

In most textbooks (Sedgewick and Flajolet, 1996, § 5.1 & 5.2), it is shown how to find the number of Catalan trees with n edges by leveraging some extremely powerful mathematical method known as *generating functions* (Graham et al., 1994, chap. 7). Instead, here, for didactical purposes, we decided to use a more intuitive technique in enumerative combinatorics which consists in constructing a one-to-one correspondence between two finite sets, so the cardinal of one set is the cardinal of the other. In other words, we are going to relate bijectively, on the one hand, Catalan trees, and, on the other hand, other combinatorial objects which are relatively easy to count.

The objects most suitable for our purpose are *monotonic lattice paths* in a square grid (Mohanty, 1979, Humphreys, 2010). These paths are made up of steps oriented upwards (↑), called *rises*, and steps rightwards

($\rightarrow$), called *falls*, starting at the bottom left corner $(0, 0)$. *Dyck paths* of length $2n$ are paths ending at (n, n) which stay above the diagonal, or touch it. We want to show that *there exists a bijection between Dyck paths of length* $2n$ *and Catalan trees with* n *edges.*

To understand that bijection, we need first to present a particular kind of *traversal*, or *walk*, of Catalan trees. Let us imagine that a tree is a map where nodes represent towns and edges roads. A complete traversal of the tree consists then in starting our trip at the root and, following edges, to visit all the nodes. (It is allowed to visit several times the same nodes, since there are no cycles.) Of course, there are many ways to achieve this tour and the one we envisage here is called a *preorder traversal*. At every node, we take the leftmost unvisited edge and visit the subtree in preorder; when back at the node, we repeat the choice with the remaining unvisited children. For more clarity, we show in FIGURE 6.3 the *preorder numbering* of FIGURE 6.2b, where the order in which a node is visited first is shown instead of a black disk ($\bullet$).

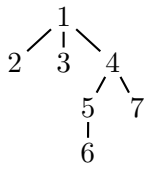


Figure 6.3

The first part of the bijection is an injection from Catalan trees with n edges to Dyck paths of length $2n$. By traversing the tree in preorder, we associate one rise to an edge on the way down, and a fall to the same edge on the way up. Obviously, there are $2n$ steps in the Dyck path. The surjection simply consists in reversing the process by reading the Dyck path step by step, rightwards, and build the corresponding tree. Now, we need to count the number of Dyck paths of length $2n$, which we know now is also the number of Catalan trees with n edges.

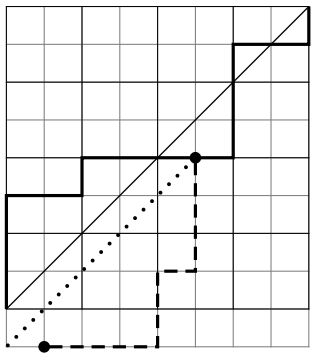


Figure 6.4: Reflection of a prefix w.r.t. $y = x - 1$

The total number of monotonic paths of length $2n$ is the number of choices of n rises amongst $2n$ steps, that is, $\binom{2n}{n}$. We need now to subtract the number of paths that start with a rise and cross the diagonal. Such a path is shown in FIGURE 6.4, drawn as a bold continuous line. The first point reached below the diagonal is used to plot a dotted line parallel to the diagonal. All the steps from that point back to $(0, 0)$ are then changed into their counterpart: a rise by a fall and vice-versa. The resulting segment is drawn as a dashed line. This operation is called a *reflection* (Renault, 2008). The crux of the matter is that we can reflect each monotonic path crossing the diagonal into a distinct path from $(1, -1)$ to (n, n). Furthermore, all reflected paths can be reflected when they reach the dotted line back into their original counterpart. In other

words, the reflection is bijective. (Another intuitive and visual approach to the same result has been published by Callan (1995).) Consequently, there are as many monotonic paths from $(0,0)$ to (n,n) that cross the diagonal as there are monotonic paths from $(1,-1)$ to (n,n). The latter are readily enumerated: $\binom{2n}{n-1}$. As a conclusion, the number of Dyck paths of length $2n$ is

$$\begin{aligned} C_n &= \binom{2n}{n} - \binom{2n}{n-1} = \binom{2n}{n} - \frac{(2n)!}{(n-1)!(n+1)!} \\ &= \binom{2n}{n} - \frac{n}{n+1} \cdot \frac{(2n)!}{n!n!} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}. \end{aligned}$$

The numbers C_n are called *Catalan numbers*. Using Stirling's formula, seen in equation (2.17) on page 83, we find that the number of Catalan trees with n edges is

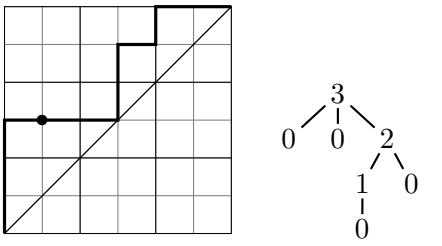
$$C_n = \frac{1}{n+1}\binom{2n}{n} \sim \frac{4^n}{n\sqrt{\pi n}}. \tag{6.1}$$

6.2 Average path length

The *path length* of a Catalan tree is the sum of the lengths of the paths from the root. We have seen this concept in the context of binary trees, where it was declined in two variants, *internal path length* (page 112) and *external path length* (page 91), depending on the end node being internal or external. In the case of Catalan trees, the pertinent distinction between nodes is to be a *leaf* (that is, a node without subtrees) or not, but some authors nevertheless speak of external path length when referring to the distances to the leaves, and of internal path length for the non-leaf nodes, hence we must bear in mind whether the context is the Catalan trees or the binary trees.

In order to study the average path length of Catalan trees, and some related parameters, we may follow Dershowitz and Zaks (1981) by finding first the average number of nodes of degree d at level l in a Catalan tree with n edges. The *degree of a node* is the number of its children and its *level* is its distance to the root counted in edges and the root is at level 0.

The first step of our method for finding the average path length consists in finding an alternative bijection between Catalan trees and Dyck paths. In FIGURE 6.2b, we see a Catalan tree equivalent to the Dyck path in FIGURE 6.2a, built from the preorder traversal of that tree. FIGURE 6.5b shows the same tree, where the contents of the nodes is their degree. The preorder traversal (of the degrees) is $[3,0,0,2,1,0,0]$. Since the last degree is always 0 (a leaf), we remove it and settle for



(a) Dyck path (b) Catalan tree

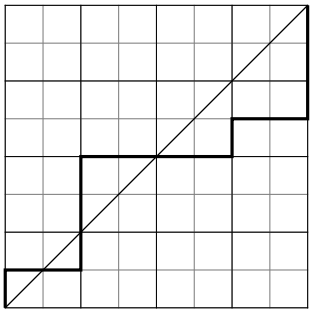
Figure 6.5: Degree-based bijection

$[3, 0, 0, 2, 1, 0]$. Another equivalent Dyck path is obtained by mapping the degrees of that list into as many occurrences of rises ($\uparrow$) and one fall ($\rightarrow$), so, for instance, 3 is mapped to $(\uparrow, \uparrow, \uparrow, \rightarrow)$ and 0 to $(\rightarrow)$. In the end, $[3, 0, 0, 2, 1, 0]$ is mapped into $[\uparrow, \uparrow, \uparrow, \rightarrow, \rightarrow, \rightarrow, \uparrow, \uparrow, \rightarrow, \uparrow, \rightarrow, \rightarrow]$, which corresponds to the Dyck path in FIGURE 6.5a. It is easy to convince ourself that we can reconstruct the tree from the Dyck path, so we indeed have a bijection.

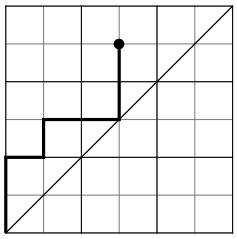
The reason for this new bijection is that we need to find the average number of Catalan trees whose root has a given degree. This number will help us in finding the average path length, following an idea of Ruskey (1983). From the bijection, it is clear that the number of trees whose root has degree $r = 3$ is the number of Dyck paths made of the segment from $(0, 0)$ to $(0, r)$, followed by one fall (see the dot at $(1, r)$ in FIGURE 6.5a), and then all monotonic paths above the diagonal until the upper right corner (n, n). Therefore, we need to determine the number of such paths.

We have seen in section 6.4 on page 202 the bijective reflection of paths and the counting principle of inclusion and exclusion. Let us add to our tools one more bijection which proves often useful: the *reversal*. It simply consists in reversing the order of the steps making up a path. Consider for example FIGURE 6.6a. Of course, the composition of two bijections being a bijection, the composition of a reversal and a reflection is bijective, hence the monotonic paths above the diagonal from $(1, r)$ to (n, n) are in bijection with the monotonic paths above the diagonal from $(0, 0)$ to $(n - r, n - 1)$. For example, FIGURE 6.6b shows the reversal and reflection of the Dyck path of FIGURE 6.5a after the point $(1, 3)$, distinguished by the black disk ($\bullet$).

Recalling that Catalan trees with n edges are in bijection with Dyck paths of length $2n$ (section 6.1 on page 194), we now know that the number of Catalan trees with n edges and whose root has degree r is the number of monotonic paths above the diagonal from the point $(0, 0)$ to $(n-r, n-1)$. We can find this number using the same technique we used



(a) Reversal of FIGURE 6.4



(b) Reversal and reflection of FIGURE 6.5a after $(1, 3)$

Figure 6.6: Reversals and reflections

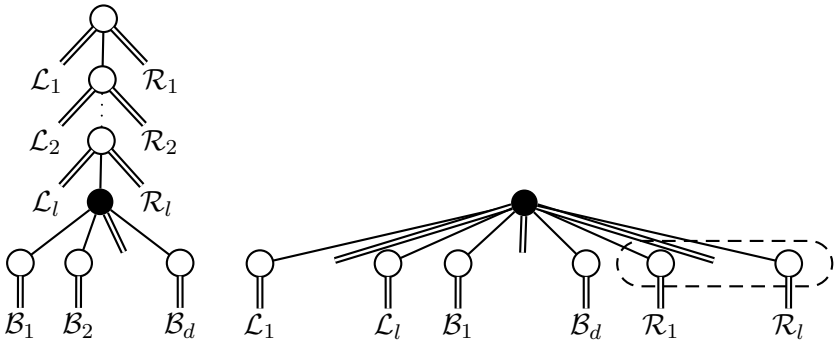
for the total number C_n of Dyck paths. The principle of inclusion and exclusion says that we should count the total number of paths with same extremities and retract the number of paths that cross the diagonal. The former is $\binom{2n-r-1}{n-1}$, which enumerates the ways to interleave $n-1$ rises ($\uparrow$) and $n-r$ falls ($\rightarrow$). The latter number is the same as the number of monotonic paths from $(1,-1)$ to $(n-r, n-1)$, as shown by reflecting the paths up to their first crossing, that is, $\binom{2n-r-1}{n}$; in other words, that is the number of interleavings of n rises with $n-r-1$ falls. Finally, imitating the derivation of equation (6.1), the number $\mathcal{R}_n(r)$ of trees with n edges and root of degree r is

$$\mathcal{R}_n(r) = \binom{2n-r-1}{n-1} - \binom{2n-r-1}{n} = \frac{r}{2n-r}\binom{2n-r}{n}.$$

Let $\mathcal{N}_n(l,d)$ be the number of nodes in the set of all Catalan trees with n edges, which are at level l and have degree d. This number is the next step in determining the average path length because Ruskey (1983) found a neat bijection to relate it to $\mathcal{R}_n(r)$ by the following equation:

$$\mathcal{N}_n(l,d) = \mathcal{R}_{n+l}(2l+d).$$

In FIGURE 6.7a is shown the general pattern of a Catalan tree with node ($\bullet$) of level d and degree d. The double edges denote a set of edges, so the $\mathcal{L}_i$, $\mathcal{R}_i$ and $\mathcal{B}_i$ actually represent forests. In FIGURE 6.7b we see a Catalan tree in bijection with the former, from which it is made by lifting the node of interest ($\bullet$) to become the root, the forests $\mathcal{L}_i$ with their respective parents are attached below it, then the $\mathcal{B}_i$, and, finally, the $\mathcal{R}_i$ for which new parents are needed (inside a dashed frame in the figure). Clearly, the new root is of degree $2l+d$ and there are $n+l$ edges. Importantly, the



(a) n edges, ($\bullet$) is of degree d and at level l

(b) $n + l$ edges, root of degree $2l + d$

Figure 6.7: Bijection

transformation can be inverted for any tree (it is injective and surjective), so it is indeed a bijection. We deduce

$$\mathcal{N}_n(l,d) = \frac{2l+d}{2n-d}\binom{2n-d}{n+l} = \binom{2n-d-1}{n+l-1} - \binom{2n-d-1}{n+l},$$

where the last step follows from expressing the binomial coefficient in terms of the factorial function. In particular, this entails that the total number of nodes at level l in all Catalan trees with n edges is

$$\sum_{d=0}^{n} \mathcal{N}_n(l,d) = \sum_{d=0}^{n} \binom{2n-d-1}{n+l-1} - \sum_{d=0}^{n} \binom{2n-d-1}{n+l}.$$

Let us consider the first sum:

$$\sum_{d=0}^{n} \binom{2n-d-1}{n+l-1} = \sum_{i=n-1}^{2n-1} \binom{i}{n+l-1} = \sum_{i=n+l-1}^{2n-1} \binom{i}{n+l-1}.$$

We can now make use of the identity (4.8) on page 124, which is equivalent to $\sum_{i=j}^{k} \binom{i}{j} = \binom{k+1}{j+1}$, so $j = n + l - 1$ and $k = 2n - 1$ yields

$$\sum_{d=0}^{n} \binom{2n-d-1}{n+l-1} = \binom{2n}{n+l}.$$

Furthermore, replacing l by $l+1$ gives $\sum_{d=0}^{n} \binom{2n-d-1}{n+l} = \binom{2n}{n+l+1}$, so the total number of nodes at level l in all Catalan trees with n edges is

$$\sum_{d=0}^{n} \mathcal{N}_n(l,d) = \binom{2n}{n+l} - \binom{2n}{n+l+1} = \frac{2l+1}{2n+1}\binom{2n+1}{n-l}. \qquad (6.2)$$

Let $\mathbb{E}[P_n]$ be the *average path length* of a Catalan tree with n edges. We have, by definition:

$$\mathbb{E}[P_n] := \frac{1}{C_n} \cdot \sum_{l=0}^{n} l \sum_{d=0}^{n} \mathcal{N}_n(l,d),$$

because there are C_n trees and the double summation is the sum of the path lengths of all the trees. If we average again by the number of nodes, *i.e.,* $n+1$, we obtain the average level of a node in a random Catalan tree and beware that some authors take this as the definition of the average path length. Alternatively, if we pick distinct Catalan trees with n edges at random and then pick random, distinct nodes in them, $\mathbb{E}[P_n]/(n+1)$ is the limit of the average cost of reaching the nodes in question from the root. Using equations (6.2) and (6.1) on page 196, we get

$$\begin{aligned}
\mathbb{E}[P_n] \cdot C_n &= \sum_{l=0}^{n} l \left[\binom{2n}{n+l} - \binom{2n}{n+l+1} \right] \\
&= \sum_{l=1}^{n} l \binom{2n}{n+l} - \sum_{l=0}^{n-1} l \binom{2n}{n+l+1} \\
&= \sum_{l=1}^{n} l \binom{2n}{n+l} - \sum_{l=1}^{n} (l-1) \binom{2n}{n+l} \\
&= \sum_{l=1}^{n} \binom{2n}{n+l} = \sum_{i=n+1}^{2n} \binom{2n}{i}.
\end{aligned}$$

The remaining summation is easy to crack because it is the sum of one half of an even row in Pascal's triangle. We see in FIGURE 4.5 on page 123 that the first half equals the second half, only the central element remaining (there is an odd number of entries in an even row). This is readily proven as follows:

$$\sum_{j=0}^{n-1} \binom{2n}{j} = \sum_{j=0}^{n-1} \binom{2n}{2n-j} = \sum_{i=n+1}^{2n} \binom{2n}{i}.$$

Therefore

$$\sum_{i=0}^{2n} \binom{2n}{i} = 2 \cdot \sum_{i=n+1}^{2n} \binom{2n}{i} + \binom{2n}{n},$$

and we can continue as follows:

$$\frac{\mathbb{E}[P_n]}{n+1} = \frac{1}{2} \binom{2n}{n}^{-1} \left[\sum_{i=0}^{2n} \binom{2n}{i} - \binom{2n}{n} \right] = \frac{1}{2} \left[\binom{2n}{n}^{-1} \sum_{i=0}^{2n} \binom{2n}{i} - 1 \right].$$

The remaining sum is perhaps the most famous combinatorial identity because it is a corollary of the venerable *binomial theorem*, which states that, for all real numbers x and y, and all positive integers n, we have the following equality:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

The truth of this statement can be seen by the following reckoning. Since, by definition, $(x+y)^n = \underbrace{(x+y)(x+y)\dots(x+y)}_{n \text{ times}}$, each term in the expansion of $(x+y)^n$ has the form $x^{n-k}y^k$, for some k ranging from 0 to n, included. The coefficient of $x^{n-k}y^k$ for a given k is simply the number of ways to choose k variables y from the n factors of $(x+y)^n$, the x variables coming from the remaining $n-k$ factors.

Setting $x = y = 1$ yields the identity $2^n = \sum_{k=0}^{n} \binom{n}{k}$, which finally unlocks our last step:

$$\mathbb{E}[P_n] = \frac{n+1}{2}\left[4^n \binom{2n}{n}^{-1} - 1\right]. \tag{6.3}$$

Recalling (6.1) on page 196, we obtain the asymptotic expansion:

$$\mathbb{E}[P_n] \sim \frac{1}{2} n\sqrt{\pi n}. \tag{6.4}$$

Note that this equivalence also holds if n denotes a number of nodes, instead of edges. The exact formula for the average path length of Catalan trees with n nodes is $\mathbb{E}[P_{n-1}]$ because there are then $n-1$ edges.

For some applications, it may be useful to know the external and internal path lengths, which are, respectively, the path lengths up to the leaves and to inner nodes (not to be confused with the external and internal path lengths of binary trees). Let $\mathbb{E}[E_n]$ be the former and $\mathbb{E}[I_n]$ the latter. We have

$$\begin{aligned}
\mathbb{E}[E_n] \cdot C_n &:= \sum_{l=0}^{n} l \cdot \mathcal{N}_n(l,0) = \sum_{l=0}^{n} l \left[\binom{2n-1}{n+l-1} - \binom{2n-1}{n+l}\right] \\
&= \sum_{l=0}^{n-1} (l+1)\binom{2n-1}{n+l} - \sum_{l=0}^{n-1} l\binom{2n-1}{n+l} = \sum_{l=0}^{n-1} \binom{2n-1}{n+l}, \\
\mathbb{E}[E_n] \cdot C_n &= \sum_{i=n}^{2n-1} \binom{2n-1}{i} = \frac{1}{2}\sum_{j=0}^{2n-1} \binom{2n-1}{j} = 4^{n-1},
\end{aligned}$$

where the penultimate step follows from the fact that an odd row in Pascal's triangle contains an even number of coefficients and the two halves have equal sums. We conclude:

$$\mathbb{E}[E_n] = (n+1)4^{n-1}\binom{2n}{n}^{-1} \sim \frac{1}{4}n\sqrt{\pi n}. \tag{6.5}$$

The derivation of $\mathbb{E}[I_n]$ is easy because

$$\mathbb{E}[P_n] = \mathbb{E}[E_n] + \mathbb{E}[I_n]. \tag{6.6}$$

From (6.3) and (6.5), we express $\mathbb{E}[P_n]$ in terms of $\mathbb{E}[E_n]$:

$$\mathbb{E}[P_n] = 2\mathbb{E}[E_n] - \frac{n+1}{2},$$

then, replacing it in (6.6), we finally draw

$$\mathbb{E}[I_n] = \mathbb{E}[E_n] - \frac{n+1}{2},$$

and

$$\mathbb{E}[I_n] = (n+1)4^{n-1}\binom{2n}{n}^{-1} - \frac{n+1}{2} \sim \frac{1}{4}n\sqrt{\pi n}. \tag{6.7}$$

Finally, formulas (6.3), (6.5) and (6.7) entail

$$\mathbb{E}[I_n] \sim \mathbb{E}[E_n] \sim \frac{1}{2}\mathbb{E}[P_n].$$

6.3 Average number of leaves

The degree-based bijection we introduced in FIGURE 6.5 on page 197 implies that there are $(n+1)/2$ leaves in average in a random Catalan tree with n edges. Indeed, a leaf is a corner in the ordinary lattice path, and it is *not* a corner in the degree-based lattice path, that is, an internal node (non-leaf), therefore, leaves and non-leaves are equinumerous and, since their total number is $n+1$, the average number of leaves is $(n+1)/2$.

For further reading, we recommend the papers by Dershowitz and Zaks (1980, 1981, 1990).

6.4 Average height

As mentioned earlier, the *height* of a tree is the number of edges on a maximal path from the root to a leaf, that is, a node without subtrees;

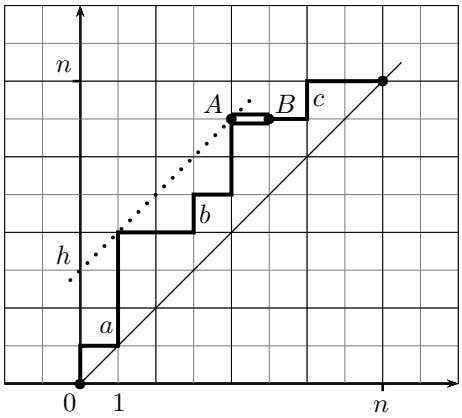


Figure 6.8: A Dyck path of length $2n$ and height h

for example, we can follow down and count the edges connecting the nodes ($\circ$) in FIGURE 6.1. A tree reduced to a single leaf has height 0.

We begin with the key observation that a Catalan tree with n edges and height h is in bijection with a Dyck path of length $2n$ and height h (see FIGURE 6.1 on page 193 and FIGURE 6.2). This simple fact allows us to reason on the height of the Dyck paths and transfer the result back to Catalan trees. Indeed, we have already seen the correspondence between lattice paths and Catalan trees, in which a rise reaching the lth diagonal corresponds to a node at level l in the tree, counting levels from root level 0. A simple bijection between paths will show that for every node on level l of a tree of height h and size n, there is a corresponding node on either level $h - l + 1$ or $h - l$ in another tree of same height and size (Dershowitz and Rinderknecht, 2015).

Consider the Dyck path in FIGURE 6.8, in bijection with a tree with $n = 8$ edges and height $h = 3$. Let us find the last (rightmost) point on the path where it reaches its full height (the dotted line of equation $y = x + h$), which we call the *apex* of the path (marked A in the figure). The immediately following fall leads to B and it is drawn with a double line. Let us rotate the segment from $(0, 0)$ to A, and the segment from B to (n, n) by 180°. The invariant fall (A, B) now connects the rotated segments. This way, what was the apex becomes the origin and vice-versa, making this a height-preserving bijection between paths. See FIGURE 6.9.

The point is that every rise to level l in FIGURE 6.8, representing a node on level l in the corresponding Catalan tree, ends up reaching level $h - l + 1$ or $h - l$ in FIGURE 6.9, depending on whether it was to the left (segment before A) or right (segment after B) of the apex. In the

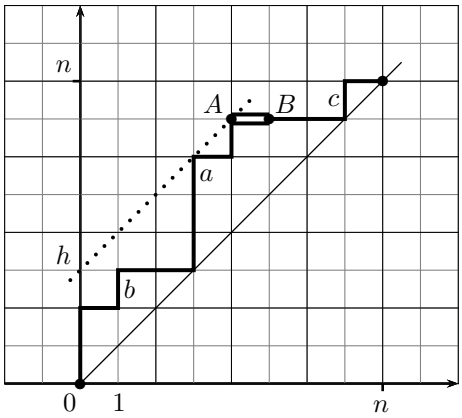


Figure 6.9: Dyck path in bijection with FIGURE 6.8

example in the figure, the rise a reaches level 1, and its counterpart after the transformation rises to level $3-1+1=3$; the rise b reached level 2 and still does so after because $3-2+1=2$; the rise c also reached level 2, but because it was to the right of the apex, it reaches now level $3-2=1$. It follows from this bijection that *the average height is within one of twice the average level of a node, that is, the average path length.* Equation (6.3) is equivalent to

$$2\frac{\mathbb{E}[P_n]}{n+1} = 4^n \binom{2n}{n}^{-1} - 1.$$

If $\mathbb{E}[H_n]$ is the average height of a Catalan tree with n edges, we then have, recalling (6.4),

$$\mathbb{E}[H_n] \sim 2\frac{\mathbb{E}[P_n]}{n+1} \sim \sqrt{\pi n}.$$

Chapter 7

Binary Trees

In this chapter, we focus on the binary tree as a data structure in itself, redefining it on the way and introducing related classical algorithms and measures.

FIGURE 7.1 displays a binary tree as an example. Nodes are of two kinds: internal (○ and ●) or external (□). The characteristic feature of a binary tree is that internal nodes are downwardly connected to two nodes, called *children*, whilst external nodes have no such links. The root is the topmost internal node, represented with a circle and a diameter. *Leaves* are internal nodes whose children are two external nodes; they are depicted as black discs.

Internal nodes are usually associated with some kind of information, whilst external nodes are not, like the one shown in FIGURE 7.3a on the following page: this is the default representation we use in the present book. Sometimes, in order to draw more attention to the internal nodes, the external nodes may be omitted, as in FIGURE 7.3b on the next page. Moreover, only

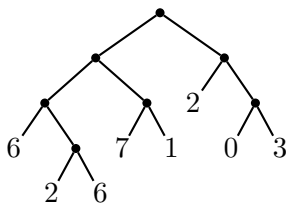


Figure 7.2: A leaf tree

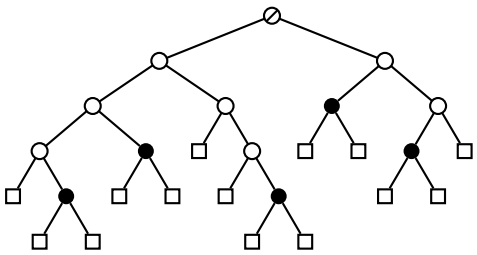
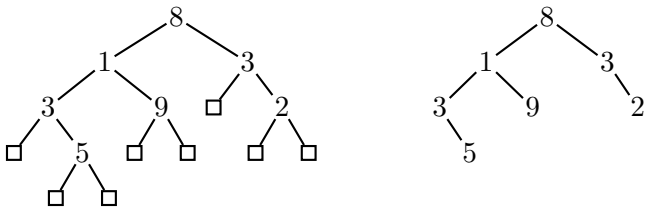


Figure 7.1: A binary tree



(a) Extended binary tree (b) Pruned binary tree

Figure 7.3: Two representations of a binary tree

the leaves might then carry information, in which case the tree is a *leaf tree*, as in FIGURE 7.2 on the preceding page. Another variant sees all nodes carrying some piece of data, as was the case with comparison trees (refer for example to FIGURE 2.41 on page 90, where external nodes contain permutations and internal nodes hold comparisons of keys).

As seen in the study of sorting algorithms optimal in average, on page 90, an *external path* is a path from the root to an external node and the *height* of a tree is the length of the maximum external paths. For example, the height of the binary tree in FIGURE 7.1 on the previous page is 5 and there are two external paths of maximum length. A path from the root to an internal node is an *internal path*. The *internal path length of a tree* is the sum of the lengths of all its internal paths. We already have met the *external path length of a tree*, on page 109, which is the sum of the lengths of all its external paths. (The length of a path is the number of its edges.)

Warning Some authors use a different nomenclature for the definition of the leaves and height. It is also common to stumble upon the concept of *depth* of a tree, coming from graph theory and which can be mistaken for its height, the former counting the number of nodes on a maximum path. Confusing depth and height leads to an off-by-one error in formulas.

Theorem 5 (Internal and external nodes)**.** *A binary tree with n internal nodes has $n + 1$ external nodes.*

Proof. Let e be the number of external nodes to be determined. We can count the edges in two complementary ways. Top-down, we see that each internal node has exactly two children, so $l = 2n$, where l is the number of edges. Bottom-up, we see that each node has exactly one parent, except the root, which as none. Therefore, $l = (n + e) - 1$. Identifying the two values of l yields $e = n + 1$. □

Data structure There are many ways to represent a binary tree as a data structure. First, we can remark that, just as a stack can be empty or not, there are two kinds of nodes, internal or external, and the empty tree can be identified to an external node. Therefore, we only need two data constructors, say ext/0 for external nodes, and int/3 for internal nodes. The latter applies to three arguments because two children are expected, as well as some information. The order of these arguments may differ. Because we may see an internal node to stand, horizontally, between its subtrees t_1 and t_2, we may prefer to write $\mathsf{int}(t_1, x, t_2)$. Alternatively, we may consider that an internal node lies, vertically, before its subtrees, in which case we might prefer to write $\mathsf{int}(x, t_1, t_2)$. The latter makes typing or handwriting small trees easier, in particular for testing purposes. For example, the binary tree of FIGURE 7.3a on the preceding page formally corresponds to $\mathsf{int}(8, t_1, \mathsf{int}(3, \mathsf{ext}(), \mathsf{ext}(2, \mathsf{ext}(), \mathsf{ext}())))$, where $t_1 = \mathsf{int}(1, \mathsf{ext}(3, \mathsf{ext}(), \mathsf{int}(5, \mathsf{ext}(), \mathsf{ext}())), \mathsf{ext}(9, \mathsf{ext}(), \mathsf{ext}()))$. We will use henceforth the convention $\mathsf{int}(x, t_1, t_2)$, sometimes called *prefix notation*.

The *size* of a binary tree is the number of its internal nodes. It is the most common measure used on trees when expressing costs of functions. As a warm-up exercise, let us write a program computing the size of a given binary tree:

$$\mathsf{size}(\mathsf{ext}()) \to 0; \quad \mathsf{size}(\mathsf{int}(x, t_1, t_2)) \to 1 + \mathsf{size}(t_1) + \mathsf{size}(t_2). \tag{7.1}$$

Notice the similarity with computing the length of a stack:

$$\mathsf{len}(\mathsf{nil}()) \to 0; \quad \mathsf{len}(\mathsf{cons}(x, s)) \to 1 + \mathsf{len}(s).$$

The difference lies in that two recursive calls are needed to visit all the nodes of a binary tree, instead of one for a stack. This bidimensional topology gives rise to many kinds of visits, called *walks* or *traversals*.

7.1 Traversals

In this section, we present the classic traversals of binary trees, which are distinguished by the order in which the data stored in the internal nodes is pushed onto a stack originally empty.

Preorder A *preorder* traversal of a non-empty binary tree consists in having in a stack the root (recursively, it is the current internal node), followed by the nodes in preorder of the left subtree and, finally, the nodes in preorder of the right subtree. (For the sake of brevity, we will

identify the contents of the nodes with the nodes themselves, when there is no ambiguity.) For example, the (contents of the) nodes in preorder of the tree in FIGURE 7.3a on page 206 are $[8, 1, 3, 5, 9, 3, 2]$. Because this method first visits the children of a node before its sibling (two internal nodes are siblings if they have the same parent), it is a *depth-first* traversal. A simple program is

$$\begin{aligned} \mathsf{pre}_0(\mathsf{ext}()) &\xrightarrow{\gamma} [\,]; \\ \mathsf{pre}_0(\mathsf{int}(x, t_1, t_2)) &\xrightarrow{\delta} [x \,|\, \mathsf{cat}(\mathsf{pre}_0(t_1), \mathsf{pre}_0(t_2))]. \end{aligned} \tag{7.2}$$

We used the concatenation on stacks provided by $\mathsf{cat}/2$, defined in (1.3) on page 7, to order the values of the subtrees. We know that the cost of $\mathsf{cat}/2$ is linear in the size of its first argument: $\mathcal{C}_p^{\mathsf{cat}} := \mathcal{C}[\![\mathsf{cat}(s,t)]\!] = p+1$, where p is the length of s. Let $\mathcal{C}_n^{\mathsf{pre}_0}$ be the cost of $\mathsf{pre}_0(t)$, where n is the number of internal nodes of t. From the definition of $\mathsf{pre}_0/1$, we deduce

$$\mathcal{C}_0^{\mathsf{pre}_0} = 1; \quad \mathcal{C}_{n+1}^{\mathsf{pre}_0} = 1 + \mathcal{C}_p^{\mathsf{pre}_0} + \mathcal{C}_{n-p}^{\mathsf{pre}_0} + \mathcal{C}_p^{\mathsf{cat}},$$

where p is the size of t_1. So $\mathcal{C}_{n+1}^{\mathsf{pre}_0} = \mathcal{C}_p^{\mathsf{pre}_0} + \mathcal{C}_{n-p}^{\mathsf{pre}_0} + p + 2$. This recurrence belongs to a class that is an instance of the *divide and conquer* principle because it springs from strategies which consists in splitting the input (here, of size $n+1$), recursively applying the relevant solving strategy to the smaller parts (here, of sizes p and $n-p$) and finally combining the solutions of the parts into a solution of the partition. The extra cost incurred by combining smaller solutions (here, $p+2$) is called the *toll* and the closed form and asymptotic behaviour of the solution to the recurrence crucially depends upon its kind.

In another context (see page 57), we, idiosyncratically, called this strategy *big-step design* because we wanted a convenient way to contrast it with another sort of modelling which we called *small-step design*. As a consequence, we already have seen instances of 'divide and conquer', for example, in the case of merge sort in chapter 4 on page 117, which often epitomises the concept itself.

The maximum cost $\mathcal{W}_k^{\mathsf{pre}_0}$ satisfies the extremal recurrence

$$\mathcal{W}_0^{\mathsf{pre}_0} = 1; \quad \mathcal{W}_{k+1}^{\mathsf{pre}_0} = 2 + \max_{0 \leqslant p \leqslant k} \{\mathcal{W}_p^{\mathsf{pre}_0} + \mathcal{W}_{k-p}^{\mathsf{pre}_0} + p\}. \tag{7.3}$$

Instead of attacking frontally these equations, we can guess a possible solution and check it. Here, we could try to consistently choose $p = k$, prompted by the idea that maximising the toll at each node of the tree will perhaps lead to a total maximum (*eager solving*). Thus, we envisage

$$\mathcal{W}_0^{\mathsf{pre}_0} = 1; \quad \mathcal{W}_{k+1}^{\mathsf{pre}_0} = \mathcal{W}_k^{\mathsf{pre}_0} + k + 3. \tag{7.4}$$

Summing both sides from $k = 0$ to $k = n - 1$ and simplifying yields

$$\mathcal{W}_n^{\mathsf{pre}_0} = \frac{1}{2}(n^2 + 5n + 2) \sim \frac{1}{2}n^2.$$

At this point, we check whether this closed form satisfies equation (7.3). We have $2(\mathcal{W}_p^{\mathsf{pre}_0} + \mathcal{W}_{n-p}^{\mathsf{pre}_0} + p + 2) = 2p^2 + 2(1-n)p + n^2 + 5n + 8$. This is the equation of a parabola whose minimum occurs at $p = (n-1)/2$ and maximum at $p = n$, over the interval $[0, n]$. The maximum, whose value is $n^2 + 7n + 8$, equals $2 \cdot \mathcal{W}_{n+1}^{\mathsf{pre}_0}$, so the closed form satisfies the extremal recurrence.

What does a binary tree maximising the cost of $\mathsf{pre}_0/1$ look like? The toll $k+3$ in (7.4) is a consequence of taking the maximum cost of $\mathsf{cat}/2$ at each node, which means that all the internal nodes being concatenated come from the left subtrees, the left subtree of the left subtree etc. so these nodes are concatenated again and again while going up (that is, returning from the recursive calls), leading to a quadratic cost. The shape of such a tree is shown in FIGURE 7.4a.

Dually, the minimum cost $\mathcal{B}_k^{\mathsf{pre}_0}$ satisfies the extremal recurrence

$$\mathcal{B}_0^{\mathsf{pre}_0} = 1; \quad \mathcal{B}_{k+1}^{\mathsf{pre}_0} = 2 + \min_{0 \leqslant p \leqslant k} \{\mathcal{B}_p^{\mathsf{pre}_0} + \mathcal{B}_{k-p}^{\mathsf{pre}_0} + p\}.$$

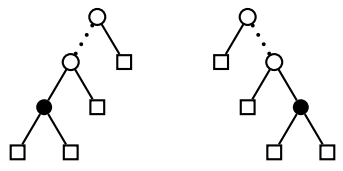
Along the same line as before, but on the opposite direction, we may try to minimise the toll by choosing $p = 0$, which means that all external nodes, but one, are left subtrees. Consequently, we have

$$\mathcal{B}_0^{\mathsf{pre}_0} = 1; \quad \mathcal{B}_{k+1}^{\mathsf{pre}_0} = \mathcal{B}_k^{\mathsf{pre}_0} + 3. \tag{7.5}$$

Summing both sides from $k = 0$ to $k = n - 1$ and simplifying yields

$$\mathcal{B}_n^{\mathsf{pre}_0} = 3n + 1 \sim 3n.$$

It is easy to check that this is indeed a solution to (7.5). The shape of the corresponding tree is shown in FIGURE 7.4b. Note that both extremal trees are isomorphic to a stack (that is, the abstract syntax tree of a stack)



(a) Maximum (b) Minimum

Figure 7.4: Extremal trees for $\mathcal{C}_n^{\mathsf{pre}_0}$

$$\mathsf{pre}_1(t) \rightarrow \mathsf{flat}(\mathsf{pre}_2(t)).$$

$$\mathsf{pre}_2(\mathsf{ext}()) \rightarrow [\,];$$
$$\mathsf{pre}_2(\mathsf{int}(x, t_1, t_2)) \rightarrow [x, \mathsf{pre}_2(t_1) \,|\, \mathsf{pre}_2(t_2)].$$

Figure 7.5: Preorder traversal using flat/1

and, as such, are instances of *degenerate trees*. Also, the maximum cost of $\mathsf{pre}_0/1$ is quadratic, which calls for some improvement.

Another big-step design we may come up with consists in not using cat/2 and calling instead flat/1, defined in FIGURE 2.12 on page 60, *once at the end*. FIGURE 7.5 shows that new version of the preorder traversal, named $\mathsf{pre}_1/1$. The cost of $\mathsf{pre}_2(t)$ is now reduced to $2n+1$ (see theorem 5 on page 206). On page 60, the cost of $\mathsf{flat}(s)$ was $1+n+\Omega+\Gamma+L$, where n is the length of $\mathsf{flat}(s)$, Ω is the number of empty stacks in s, Γ is the number of non-empty stacks and L is the sum of the lengths of the embedded stacks. The value of Ω is $n+1$ because this is the number of external nodes. The value of Γ is $n-1$, because each internal node yields a non-empty stack by the second rule of $\mathsf{pre}_2/1$ and the root is excluded because we only count embedded stacks. The value of L is $3(n-1)$ because those stacks have length 3 by the same rule. In the end, $\mathcal{C}[\![\mathsf{flat}(s)]\!] = 6n-2$, where $\mathsf{pre}_2(t) \twoheadrightarrow s$ and n is the size of t. Finally, we must account for the rule defining $\mathsf{pre}_1/1$ and assess afresh the cost incurred by the empty tree:

$$\mathcal{C}_0^{\mathsf{pre}_1} = 3; \quad \mathcal{C}_n^{\mathsf{pre}_1} = 1 + (2n+1) + (6n-2) = 8n, \text{ when } n > 0.$$

Despite a significant improvement in the cost and the lack of extreme cases, we should try a small-step design before giving up. The underlying principle in this kind of approach is to do as little as possible in each rule. Looking back at $\mathsf{pre}_0/1$, it is clear that the root is correctly placed, but, without resorting to $\mathsf{pre}_3(t_1)$ and $\mathsf{pre}_3(t_2)$ in the following canvas, what can be done further?

$$\mathsf{pre}_3(\mathsf{ext}()) \rightarrow [\,]; \quad \mathsf{pre}_3(\mathsf{int}(x, t_1, t_2)) \rightarrow [x \,|\, \boxed{}\,].$$

The way forth is to think in terms of forests, instead of single trees, because a forest is a stack of trees and, as such, can also be used to accumulate trees. This is a common technique when processing trees. See FIGURE 7.6 on the facing page. Empty trees in the forest are skipped in rule γ. In rule δ, the subtrees t_1 and t_2 are now simply pushed back onto the forest f, for later processing. This way, there is no need to compute

$$\mathsf{pre}_3(t) \xrightarrow{\alpha} \mathsf{pre}_4([t]).$$

$$\begin{aligned}
\mathsf{pre}_4([\,]) &\xrightarrow{\beta} [\,];\\
\mathsf{pre}_4([\mathsf{ext}() \,|\, f]) &\xrightarrow{\gamma} \mathsf{pre}_4(f);\\
\mathsf{pre}_4([\mathsf{int}(x, t_1, t_2) \,|\, f]) &\xrightarrow{\delta} [x \,|\, \mathsf{pre}_4([t_1, t_2 \,|\, f])].
\end{aligned}$$

Figure 7.6: Efficient preorder traversal with a forest

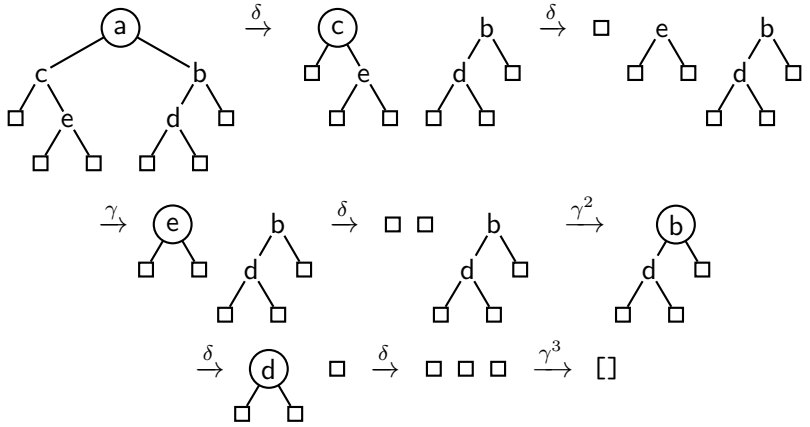


Figure 7.7: A preorder traversal with $\mathsf{pre}_4/1$

$\mathsf{pre}_4(t_1)$ or $\mathsf{pre}_4(t_2)$ immediately. This method is slightly different from using an accumulator which contains, at every moment, a partial result or a reversed partial result. Here, no parameter is added but, instead, a stack replaces the original parameter, and it does not contain partial results, but pieces of the original tree from which to pick internal nodes easily (the root of the first tree) in the expected order. An example is given in FIGURE 7.7, where the forest is the argument of $\mathsf{pre}_4/1$ and the circle nodes are the current value of x in rule δ of FIGURE 7.6.

The cost of $\mathsf{pre}_3(t)$, where the size of t is n, is simple:

- rule α is used once;
- rule β is used once;
- rule γ is used once for each external node, that is, $n+1$ times;
- rule δ is used once for each internal node, so n times, by definition.

In total, $\mathcal{C}_n^{\mathsf{pre}_3} = 2n + 3 \sim 2n$, which is a notable improvement. The keen reader may remark that we could further reduce the cost by not visiting the external nodes, as shown in FIGURE 7.8. We then have

$$\mathcal{C}_n^{\mathsf{pre}_5} = \mathcal{C}_n^{\mathsf{pre}_3} - (n+1) = n + 2.$$

$$
\begin{aligned}
\mathsf{pre}_5(t) &\to \mathsf{pre}_6([t]).\\
\\
\mathsf{pre}_6([\,]) &\to [\,];\\
\mathsf{pre}_6([\mathsf{ext}() \,|\, f]) &\to \mathsf{pre}_6(f);\\
\mathsf{pre}_6([\mathsf{int}(x, \mathsf{ext}(), \mathsf{ext}()) \,|\, f]) &\to [x \,|\, \mathsf{pre}_6(f)];\\
\mathsf{pre}_6([\mathsf{int}(x, \mathsf{ext}(), t_2) \,|\, f]) &\to [x \,|\, \mathsf{pre}_6([t_2 \,|\, f])];\\
\mathsf{pre}_6([\mathsf{int}(x, t_1, \mathsf{ext}()) \,|\, f]) &\to [x \,|\, \mathsf{pre}_6([t_1 \,|\, f])];\\
\mathsf{pre}_6([\mathsf{int}(x, t_1, t_2) \,|\, f]) &\to [x \,|\, \mathsf{pre}_6([t_1, t_2 \,|\, f])].
\end{aligned}
$$

Figure 7.8: Lengthy definition of a preorder traversal

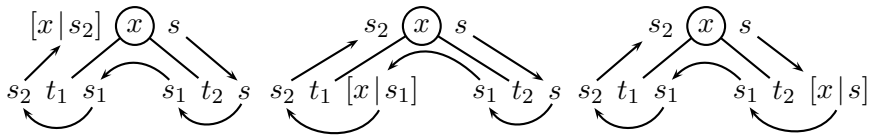
Despite the gain, the optimised program is significantly longer and the right-hand sides of the new rules are partial evaluations of rule δ. The measure of the input we use for calculating the costs does not include the abstract time needed to select the rule to apply but it is likely, though, that the more patterns, the higher this hidden penalty. In this book, we prefer to visit the external nodes unless there is a logical reason not to do so, if only for the sake of conciseness.

The total number of cons-nodes created by the rules α and δ is the total number of nodes, $2n + 1$, but if we want to know how many there can be at any time, we need to consider how the shape of the original tree influences the rules γ and δ. In the best case, t_1 in δ is $\mathsf{ext}()$ and will be eliminated next by rule γ without additional creation of nodes. In the worst case, t_1 maximises the number of internal nodes on its left branch. Therefore, these two configurations correspond to the extremal cases for the cost of $\mathsf{pre}_0/1$ in FIGURE 7.4 on page 209. In the worst case, all the $2n + 1$ nodes of the tree will be in the stack at one point, whilst, in the best case, only two will be. The question of the average stack size will be considered later in this text in relation with the average height.

The distinction between big-step design (or 'divide and conquer') and small-step design is not always a clear cut and is mainly intended to be a didactical means. In particular, we should not assume that there are only two possible kinds of design for every given task. To bring further clarity to the subject, let us use an heterogeneous approach to design another version, $\mathsf{pre}/1$, which computes efficiently the preorder traversal of a given binary tree. Looking back at $\mathsf{pre}_0/1$, we can identify the source of the inefficiency in the fact that, in the worst case,

$$\mathsf{pre}_0(t) \twoheadrightarrow [x_1 \,|\, \mathsf{cat}([x_2 \,|\, \mathsf{cat}(\ldots \mathsf{cat}([x_n \,|\, \mathsf{cat}([\,], [\,])], [\,]) \ldots)])]$$

where $t = \mathsf{int}(x_1, \mathsf{int}(x_2, \ldots, \mathsf{int}(x_n, \mathsf{ext}(), \mathsf{ext}()), \ldots, \mathsf{ext}()), \mathsf{ext}())$ is the tree in FIGURE 7.4a on page 209. We met this kind of partial rewrite



(a) Preorder with pre/1 (b) Inorder with in/1 (c) Postorder with post/1

Figure 7.9: Efficient classic traversals

in formula (2.4) on page 42 and (3.2) on page 94, and we found that it leads to a quadratic cost. Whilst the use of cat/2 in itself is not the issue, but rather the accumulation of calls to cat/2 as their first argument, let us nevertheless seek out a definition not relying on concatenation at all. This means that we want to build the preorder stack by using exclusively pushes. Therefore, we must add an auxiliary parameter, originally set to the empty stack, on which the contents of the nodes are pushed in the proper order: $\mathsf{pre}(t) \to \mathsf{pre}(t, [\,])$. Now, we should wonder what the interpretation of this accumulator is when considering the pattern $\mathsf{pre}(t, s)$.

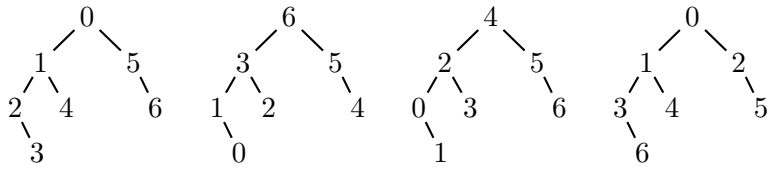
Let us have a look at the internal node $t = \mathsf{int}(x, t_1, t_2)$ in FIGURE 7.9a. The arrows evince the traversal in the tree and connect different stages of the preorder stack: a downwards arrow points to the argument of a recursive call on the corresponding child; an upward arrow points to the result of the call on the parent. For instance, the subtree t_2 corresponds to the recursive call $\mathsf{pre}(t_2, s)$ whose value is named s_1. Likewise, we have $\mathsf{pre}(t_1, s_1) \twoheadrightarrow s_2$, which is therefore equivalent to $\mathsf{pre}(t_1, \mathsf{pre}(t_2, s)) \twoheadrightarrow s_2$. Finally, the root is associated with the evaluation $\mathsf{pre}(t, s) \twoheadrightarrow [x \mid s_2]$, which is none other than the equivalence $\mathsf{pre}(t, s) \equiv [x \,|\, \mathsf{pre}(t_1, \mathsf{pre}(t_2, s))]$. The rule for external nodes is not shown and simply consists in letting the stack invariant. Finally we have the functional program in FIGURE 7.10.

$$\mathsf{pre}(t) \xrightarrow{\theta} \mathsf{pre}(t, [\,]).$$

$$\mathsf{pre}(\mathsf{ext}(), s) \xrightarrow{\iota} s;$$
$$\mathsf{pre}(\mathsf{int}(x, t_1, t_2), s) \xrightarrow{\kappa} [x \,|\, \mathsf{pre}(t_1, \mathsf{pre}(t_2, s))].$$

Figure 7.10: Cost and memory efficient preorder

We now understand that, given $\mathsf{pre}(t, s)$, the nodes in the stack s are the nodes that follow, in preorder, the nodes in the subtree t. The cost



(a) Preorder (b) Postorder (c) Inorder (d) Level order

Figure 7.11: Classic numberings of a pruned binary tree

is extremely simple:

$$\mathcal{C}_n^{\mathsf{pre}} = 1 + (n+1) + n = 2n + 2. \tag{7.6}$$

(Keep in mind that there are $n + 1$ external nodes when there are n internal nodes.)

This variant is to be preferred over $\mathsf{pre}_3/1$ because its memory usage is lower: rule δ in FIGURE 7.6 on page 211 pushes t_1 and t_2, thus allocates two cons-nodes per internal node, totalling $2n$ supplementary nodes. By contrast, $\mathsf{pre}/1$ creates none, but allocates n call-nodes pre (one per internal node), so the advantage stands, albeit moot. Note that $\mathsf{pre}_3/1$ is in tail form, but not $\mathsf{pre}/2$.

Preorder numbering FIGURE 7.11a shows a binary tree whose internal nodes have been replaced by their rank in preorder, with the smallest number, 0, at the root. In particular, the preorder traversal of that tree yields $[0, 1, 2, 3, 4, 5, 6]$. Producing such a tree from some initial tree is a *preorder numbering*. A complete example is shown in FIGURE 7.12, where the preorder numbers are exponents to the internal nodes. Note how these numbers increase along downwards paths.

Their generation can be tackled in two phases: first, we need to understand how to produce the right number for a given node; second, we need to use these numbers to build a tree. The scheme for the former is shown on internal nodes in FIGURE 7.13a on the next page. A num-

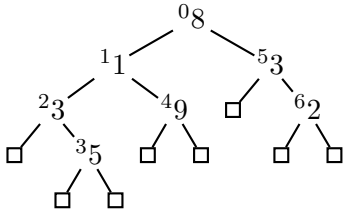
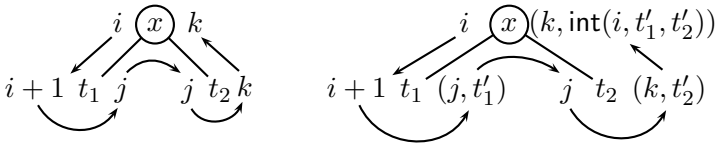


Figure 7.12: Preorder numbers as exponents



(a) Numbers only (b) Numbers and tree

Figure 7.13: Preorder numbering in two phases

ber on the left side of a node indexes it, for example node x is indexed with i: these numbers descend in the tree. A number on the right side of a node is the smallest number not used in the numbering of the subtrees attached to that node: these numbers ascend and may be used for the numbering of another subtree. For instance, j is the smallest integer not numbering the nodes of the subtree t_1. External nodes do not change their incoming number and are not depicted.

The second and final design phase consists in the construction of the tree made of these numbers and it is pictured in FIGURE 7.13b. Conceptually, it is the completion of the first phase in the sense that upwards arrows, which denote values of recursive calls on subtrees, now point at pairs whose first component is the number we found earlier and the second component is a numbered tree. As usual with recursive definitions, we assume that the recursive calls on substructures are correct (that is, they yield the expected values) and we infer the value of the call on the whole structure at hand, here $(k, \mathsf{int}(i, t'_1, t'_2))$. The function $\mathsf{npre/1}$ (*number in preorder*) in FIGURE 7.14 implements this algorithm, using a function $\mathsf{npre/2}$ such that $\mathsf{npre}(i, t) \twoheadrightarrow (j, t')$, where t' is the preorder numbering of t, with root i, and j is the smallest integer not found in t' (in other words, $j - i$ is the size of t and t'). This function is the one illustrated in FIGURE 7.13b. By the way, we should perhaps recall that inference systems, first seen on page 70, can be eliminated by the introduction of auxiliary functions (one for each premise). In this instance, we could equivalently write the program in FIGURE 7.15 on the following page.

$$\frac{\mathsf{npre}(0, t) \twoheadrightarrow (i, t')}{\mathsf{npre}(t) \twoheadrightarrow t'}\,. \qquad \mathsf{npre}(i, \mathsf{ext}()) \to (i, \mathsf{ext}());$$

$$\frac{\mathsf{npre}(i + 1, t_1) \twoheadrightarrow (j, t'_1) \qquad \mathsf{npre}(j, t_2) \twoheadrightarrow (k, t'_2)}{\mathsf{npre}(i, \mathsf{int}(x, t_1, t_2)) \twoheadrightarrow (k, \mathsf{int}(i, t'_1, t'_2))}\,.$$

Figure 7.14: Preorder numbering

$$\mathsf{npre}(t) \to \mathsf{snd}(\mathsf{npre}(0,t)).$$

$$\mathsf{snd}((x,y)) \to y.$$

$$\begin{aligned}\mathsf{npre}(i,\mathsf{ext}()) &\to (i,\mathsf{ext}());\\ \mathsf{npre}(i,\mathsf{int}(x,t_1,t_2)) &\to \mathsf{t}_1(\mathsf{npre}(i+1,t_1),i,t_2).\end{aligned}$$

$$\mathsf{t}_1((j,t_1'),i,t_2) \to \mathsf{t}_2(\mathsf{npre}(j,t_2),i,t_1').$$

$$\mathsf{t}_2((k,t_2'),i,t_1') \to (k,\mathsf{int}(i,t_1',t_2')).$$

Figure 7.15: Version of npre/1 without inference rules

Termination It is easy to prove the termination of pre/2 because the technique used for proving the termination of Ackermann's function on page 13 is pertinent in the current context as well. We define a lexicographic order on the calls to pre/2 (dependency pairs) as follows:

$$\mathsf{pre}(t,s) \succ \mathsf{pre}(t',s') :\Leftrightarrow t \succ_B t' \text{ or } (t = t' \text{ and } s \succ_S s'), \qquad (7.7)$$

where B is the set of all binary trees, S is the set of all stacks, $t \succ_B t'$ means that the tree t' is an immediate subtree of t, and $s \succ_S s'$ means that the stack s' is an immediate substack of s (page 12). From the definition in FIGURE 7.10 on page 213, we see that rule θ maintains termination if pre/2 terminates; rule ι terminates; finally, rule κ rewrites a call into smaller calls:

$$\begin{aligned}&\mathsf{pre}(\mathsf{int}(x,t_1,t_2),s) \succ \mathsf{pre}(t_2,s),\\ &\mathsf{pre}(\mathsf{int}(x,t_1,t_2),s) \succ \mathsf{pre}(t_1,u), \text{ for all } u,\end{aligned}$$

in particular if $\mathsf{pre}(t_2,s) \twoheadrightarrow u$. As a consequence, pre/1 terminates on all inputs. □

Equivalence To see how structural induction can be used to prove properties on binary trees, we will consider a simple statement we made earlier, formally expressed as $\mathsf{Pre}(t)\colon \mathsf{pre}_0(t) \equiv \mathsf{pre}(t)$. We need to prove

- the basis $\mathsf{Pre}(\mathsf{ext}())$;
- the inductive step $\forall t_1.\mathsf{Pre}(t_1) \Rightarrow \forall t_2.\mathsf{Pre}(t_2) \Rightarrow \forall x.\mathsf{Pre}(\mathsf{int}(x,t_1,t_2))$.

The basis is easy because

$$\mathsf{pre}_0(\mathsf{ext}()) \xrightarrow{\gamma} [\,] \xleftarrow{\iota} \mathsf{pre}(\mathsf{ext}(),[\,]) \xleftarrow{\theta} \mathsf{pre}(\mathsf{ext}()).$$

See definition of $\mathsf{pre}_0/1$ at equation (7.2) on page 208. It is useful here to recall the definition of $\mathsf{cat}/2$:

$$\mathsf{cat}([\,],t) \xrightarrow{\alpha} t; \qquad \mathsf{cat}([x\,|\,s],t) \xrightarrow{\beta} [x\,|\,\mathsf{cat}(s,t)].$$

In order to discover how to use the two induction hypotheses $\mathsf{Pre}(t_1)$ and $\mathsf{Pre}(t_2)$, let us start from one side of the equivalence we wish to establish, for example, the left-hand side, and rewrite it until we reach the other side or get stuck. Let $t := \mathsf{int}(x,t_1,t_2)$, then

$$\begin{array}{lll}
\mathsf{pre}_0(t) & = & \mathsf{pre}_0(\mathsf{int}(x,t_1,t_2)) \\
& \xrightarrow{\delta} & [x\,|\,\mathsf{cat}(\mathsf{pre}_0(t_1),\mathsf{pre}_0(t_2))] \\
& \equiv & [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1),\mathsf{pre}_0(t_2))] \qquad (\mathsf{Pre}(t_1)) \\
& \equiv & [x\,|\,\mathsf{cat}(\underline{\mathsf{pre}}(t_1),\mathsf{pre}(t_2))] \qquad (\mathsf{Pre}(t_2)) \\
& \xrightarrow{\theta} & [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\underline{\mathsf{pre}}(t_2))] \\
& \xrightarrow{\theta} & [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\mathsf{pre}(t_2,[\,]))].
\end{array}$$

At this point, we start rewriting the other side, until we get stuck as well:

$$\mathsf{pre}(t) = \mathsf{pre}(\mathsf{int}(x,t_1,t_2)) \xrightarrow{\theta} \mathsf{pre}(\mathsf{int}(x,t_1,t_2),[\,]) \xrightarrow{\kappa} [x\,|\,\mathsf{pre}(t_1,\mathsf{pre}(t_2,[\,]))].$$

Comparing the two stuck expressions suggests a subgoal to reach.

Let $\mathsf{CatPre}(t,s)\colon \mathsf{cat}(\mathsf{pre}(t,[\,]),s) \equiv \mathsf{pre}(t,s)$. When a predicate depends upon two parameters, we have different options to ponder: either we need lexicographic induction, or simple induction on one of the variables. It is best to use a lexicographic ordering on pairs and, if we realise afterwards that only one component was needed, we can rewrite the proof with a simple induction on that component. Let us then define

$$(t,s) \succ_{B\times S} (t',s') :\Leftrightarrow t \succ_B t' \text{ or } (t = t' \text{ and } s \succ_S s').$$

This is conceptually the same order as the one on the calls to $\mathsf{pre}/1$, in definition (7.7). If we find out later that immediate subterm relations are too restrictive, we would choose here general subterm relations, which means, in the case of binary trees, that a tree is a subtree of another. The minimum element for the lexicographic order we just defined is $(\mathsf{ext}(),[\,])$. The well-founded induction principle then requires that we establish

- the basis $\mathsf{CatPre}(\mathsf{ext}(),[\,])$;
- $\forall t,s.(\forall t',s'.(t,s) \succ_{B\times S} (t',s') \Rightarrow \mathsf{CatPre}(t',s')) \Rightarrow \mathsf{CatPre}(t,s)$.

The basis is easy:

$$\mathsf{cat}(\underline{\mathsf{pre}}(\mathsf{ext}(),[\,]),[\,]) \xrightarrow{\iota} \mathsf{cat}([\,],[\,]) \xrightarrow{\alpha} [\,] \xleftarrow{\iota} \mathsf{pre}(\mathsf{ext}(),[\,]).$$

We then assume $\forall t',s'.(t,s) \succ_{B\times S} (t',s') \Rightarrow \mathsf{CatPre}(t',s')$, which is the induction hypothesis, and proceed to rewrite the left-hand side after letting $t := \mathsf{int}(x,t_1,t_2)$. See FIGURE 7.16 on the next page, where

$$
\begin{array}{ll}
\mathsf{cat}(\mathsf{pre}(t,[\,]),s) & = \mathsf{cat}(\underline{\mathsf{pre}}(\mathsf{int}(x,t_1,t_2),[\,]),s) \\
& \xrightarrow{\kappa} \underline{\mathsf{cat}}([x\,|\,\mathsf{pre}(t_1,\mathsf{pre}(t_2,[\,]))],s) \\
& \xrightarrow{\beta} [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,\mathsf{pre}(t_2,[\,])),s)] \\
& \equiv_0 [x\,|\,\mathsf{cat}(\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\mathsf{pre}(t_2,[\,])),s)] \\
& \equiv_1 [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\mathsf{cat}(\mathsf{pre}(t_2,[\,]),s))] \\
& \equiv_2 [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\mathsf{pre}(t_2,s))] \\
& \equiv_3 [x\,|\,\mathsf{pre}(t_1,\mathsf{pre}(t_2,s))] \\
& \xleftarrow{\kappa} \mathsf{pre}(\mathsf{int}(x,t_1,t_2),s) \\
& = \mathsf{pre}(t,s). \qquad \square
\end{array}
$$

Figure 7.16: Proof of $\mathsf{CatPre}(t)$: $\mathsf{cat}(\mathsf{pre}(t,[\,]),s) \equiv \mathsf{pre}(t,s)$

$$
\begin{array}{lll}
\mathsf{pre}_0(t) & = \mathsf{pre}_0(\mathsf{int}(x,t_1,t_2)) & \\
& \xrightarrow{\delta} [x\,|\,\mathsf{cat}(\mathsf{pre}_0(t_1),\mathsf{pre}_0(t_2))] & \\
& \equiv [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1),\mathsf{pre}_0(t_2))] & (\mathsf{Pre}(t_1)) \\
& \equiv [x\,|\,\mathsf{cat}(\underline{\mathsf{pre}}(t_1),\mathsf{pre}(t_2))] & (\mathsf{Pre}(t_2)) \\
& \xrightarrow{\theta} [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\underline{\mathsf{pre}}(t_2))] & \\
& \xrightarrow{\theta} [x\,|\,\mathsf{cat}(\mathsf{pre}(t_1,[\,]),\mathsf{pre}(t_2,[\,]))] & \\
& \equiv [x\,|\,\mathsf{pre}(t_1,\mathsf{pre}(t_2,[\,]))] & (\mathsf{CatPre}(t_1,\mathsf{pre}(t_2,[\,]))) \\
& \xleftarrow{\kappa} \mathsf{pre}(\mathsf{int}(x,t_1,t_2),[\,]) & \\
& \xleftarrow{\theta} \mathsf{pre}(\mathsf{int}(x,t_1,t_2)) & \\
& = \mathsf{pre}(t). & \square
\end{array}
$$

Figure 7.17: Proof of $\mathsf{Pre}(t)$: $\mathsf{pre}_0(t) \equiv \mathsf{pre}(t)$

- $(\equiv_0)$ is the instance $\mathsf{CatPre}(t_1, \mathsf{pre}(t_2,[\,]))$ of the induction hypothesis because $(t,s) \succ_{B\times S} (t_1,s')$, for all stacks s', in particular when $\mathsf{pre}(t_2,[\,]) \twoheadrightarrow s'$;
- $(\equiv_1)$ is an application of the lemma on the associativity of stack concatenation (page 14), that is: $\mathsf{CatAssoc}(\mathsf{pre}(t_1,[\,]),\mathsf{pre}(t_2,[\,]),s)$;
- $(\equiv_2)$ is the instance $\mathsf{CatPre}(t_2,s)$ of the induction hypothesis because $(t,s) \succ_{B\times S} (t_2,s)$;
- $(\equiv_3)$ is the instance $\mathsf{CatPre}(t_1,\mathsf{pre}(t_2,s))$ of the induction hypothesis because $(t,s) \succ_{B\times S} (t_1,s')$, for all stacks s', in particular when $s' = \mathsf{pre}(t_2,s)$.

We can resume conclusively in FIGURE 7.17. $\square$

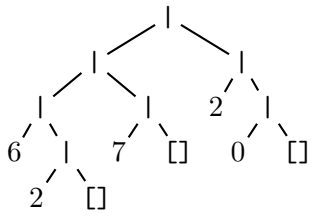


Figure 7.18: Embedded stacks as a leaf tree

Flattening revisited In section 2.4 on page 56, we defined two functions for flattening stacks (see FIGURE 2.8 on page 57 and FIGURE 2.12 on page 60). With the understanding of preorder traversals, it may occur to us that the flattening of a stack is equivalent to the preorder traversal of a binary leaf tree (see FIGURE 7.2 on page 205 for an example) which ignores empty stacks. The key is to see a stack, possibly containing other stacks, as a leaf tree, as shown for example in FIGURE 7.18, where the internal nodes (|) are cons-nodes.

The first step consists in defining inductively the set of the binary leaf trees as the smallest set L generated by the deductive (downwards) reading of the inference system

$$\mathsf{leaf}(x) \in L \qquad \frac{t_1 \in L \qquad t_2 \in L}{\mathsf{fork}(t_1, t_2) \in L}.$$

In other words, a leaf containing the piece of data x is noted $\mathsf{leaf}(x)$ and the other internal nodes are *forks*, written $\mathsf{fork}(t_1, t_2)$, with t_1 and t_2 being binary leaf trees themselves. The second step requires a modification of pre/1, defined in FIGURE 7.10 on page 213, so it processes binary leaf trees. The new function, lpre/1, is shown in FIGURE 7.19. The final step is the translation of lpre/1 and lpre/2 into flat_2/1 and flat_2/2, respectively, in FIGURE 7.20 on the following page. The key is to see that $\mathsf{fork}(t_1, t_2)$ becomes $[t_1 \,|\, t_2]$, and $\mathsf{leaf}(x)$, when x is not an empty stack, becomes x as the *last* pattern. The case $\mathsf{leaf}([\,])$ becomes $[\,]$ as the first pattern.

The cost of pre/1 was found to be $\mathcal{C}_n^{\mathsf{pre}} = 2n + 2$ in equation (7.6) on

$$\begin{array}{r} \mathsf{lpre}(t) \to \mathsf{lpre}(t, [\,]). \\ \\ \mathsf{lpre}(\mathsf{leaf}(x), s) \to [x \,|\, s]; \\ \mathsf{lpre}(\mathsf{fork}(t_1, t_2), s) \to \mathsf{lpre}(t_1, \mathsf{lpre}(t_2, s)). \end{array}$$

Figure 7.19: Preorder on binary leaf trees

$$\mathsf{flat}_2(t) \to \mathsf{flat}_2(t, [\,]).$$

$$\begin{aligned} \mathsf{flat}_2([\,], s) &\to s; \\ \mathsf{flat}_2([t_1 \,|\, t_2], s) &\to \mathsf{flat}_2(t_1, \mathsf{flat}_2(t_2, s)); \\ \mathsf{flat}_2(x, s) &\to [x \,|\, s]. \end{aligned}$$

Figure 7.20: Flattening like lpre/1

page 214, where n was the number of internal nodes. Here, n is the length of $\mathsf{flat}_2(t)$, namely, the number of non-stack leaves in the leaf tree. With this definition, the number of cons-nodes is $n + \Omega + \Gamma$, where Ω is the number of embedded empty stacks and Γ is the number of embedded non-empty stacks, so $S := 1 + \Omega + \Gamma$ is the total number of stacks. Consequently, $\mathcal{C}_n^{\mathsf{flat}_2} = 2(n + S)$. For example,

$$\mathsf{flat}_2([1, [[\,], [2, 3]], [[4]], 5]) \xrightarrow{22} [1, 2, 3, 4, 5],$$

because $n = 5$ and $S = 6$. (The latter is the number of opening square brackets.) When $S = 1$, that is, the stack is flat, then $\mathcal{C}_n^{\mathsf{pre}} = \mathcal{C}_n^{\mathsf{flat}_2}$, otherwise $\mathcal{C}_n^{\mathsf{pre}} < \mathcal{C}_n^{\mathsf{flat}_2}$.

Inorder The *inorder* (or *symmetric*) traversal of a non-empty binary tree consists in having in a stack the nodes of the left subtree in inorder, followed by the root and then the nodes of the right subtree in inorder. For example, the nodes in inorder of the tree in FIGURE 7.3a on page 206 are $[3, 5, 1, 9, 8, 3, 2]$. Clearly, it is a depth-first traversal, like a preorder, because children are visited before siblings. According to our findings about pre/1 in FIGURE 7.10 on page 213, we understand that we should structure our program to follow the strategy depicted in FIGURE 7.9b on page 213, where the only difference with FIGURE 7.9a is the moment when the root x is pushed on the accumulator: between the inorder traversals of the subtrees t_1 and t_2. The implicit rewrites in FIGURE 7.9b are

$$\begin{aligned} \mathsf{in}(t_2, s) &\twoheadrightarrow s_1, \\ \mathsf{in}(t_1, [x \,|\, s_1]) &\twoheadrightarrow s_2, \\ \mathsf{in}(t, s) &\twoheadrightarrow s_2, \end{aligned}$$

where $t = \mathsf{int}(x, t_1, t_2)$. Eliminating the intermediary variables s_1 and s_2 we obtain the equivalence

$$\mathsf{in}(t_1, [x \,|\, \mathsf{in}(t_2, s)]) \equiv \mathsf{in}(t, s).$$

$$\mathsf{in}(t) \xrightarrow{\xi} \mathsf{in}(t, [\,]).$$

$$\begin{aligned} \mathsf{in}(\mathsf{ext}(), s) &\xrightarrow{\pi} s; \\ \mathsf{in}(\mathsf{int}(x, t_1, t_2), s) &\xrightarrow{\rho} \mathsf{in}(t_1, [x \,|\, \mathsf{in}(t_2, s)]). \end{aligned}$$

Figure 7.21: Inorder traversal

$$\frac{\mathsf{nin}(0, t) \twoheadrightarrow (i, t')}{\mathsf{nin}(t) \twoheadrightarrow t'}. \qquad \mathsf{nin}(i, \mathsf{ext}()) \to (i, \mathsf{ext}());$$

$$\frac{\mathsf{nin}(i, t_1) \twoheadrightarrow (j, t_1') \qquad \mathsf{nin}(j+1, t_2) \twoheadrightarrow (k, t_2')}{\mathsf{nin}(i, \mathsf{int}(x, t_1, t_2)) \twoheadrightarrow (k, \mathsf{int}(j, t_1', t_2'))}.$$

Figure 7.22: Inorder numbering

The case of the external node is the same as for preorder. This reasoning yields the function defined in FIGURE 7.21, whose cost is the same as for preorder: $\mathcal{C}_n^{\mathsf{in}} = \mathcal{C}_n^{\mathsf{pre}} = 2n + 2$.

FIGURE 7.11c on page 214 gives the example of a binary tree which is the result of an *inorder numbering*. The inorder traversal of that tree yields $[0, 1, 2, 3, 4, 5, 6]$. Inorder numberings have an interesting property: given any internal node, all the nodes in its left subtree have smaller numbers, and all nodes in its right subtree have greater numbers. Let $\mathsf{nin}/1$ be a function computing the inorder numbering of a given tree in FIGURE 7.22, where j, at the root, is the smallest number greater than any number in t_1.

Flattening revisited The design of $\mathsf{flat}/1$ in FIGURE 2.12 on page 60 may suggest a new approach to inorder traversals. By composing right rotations as defined in FIGURES 2.11b to 2.11c on page 59 (the converse is, of course, a *left rotation*), the node to be visited first in inorder can be brought to be the root of a tree whose left subtree is empty. Recursively, the right subtree is then processed, in a top-down fashion. This algorithm is sound because *inorder traversals are invariant through rotations*, which is formally expressed as follows.

$$\mathsf{Rot}(x, y, t_1, t_2, t_3) \colon \mathsf{in}(\mathsf{int}(y, \mathsf{int}(x, t_1, t_2), t_3)) \equiv \mathsf{in}(\mathsf{int}(x, t_1, \mathsf{int}(y, t_2, t_3))).$$

In FIGURE 7.23 on the next page, we show how a binary tree becomes a

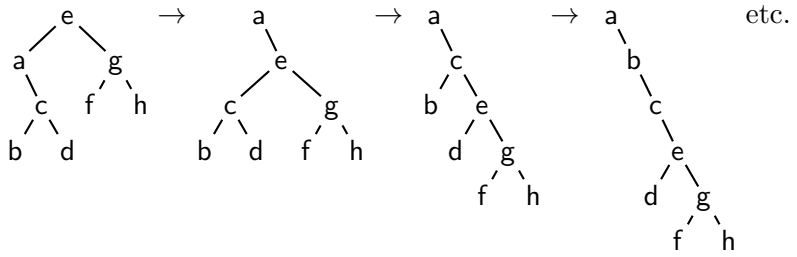


Figure 7.23: Composing right-rotations, top-down

$$\begin{array}{r} \mathsf{in}_1(\mathsf{ext}()) \xrightarrow{\alpha} [\,]; \\ \mathsf{in}_1(\mathsf{int}(y, \mathsf{int}(x, t_1, t_2), t_3)) \xrightarrow{\beta} \mathsf{in}_1(\mathsf{int}(x, t_1, \mathsf{int}(y, t_2, t_3))); \\ \mathsf{in}_1(\mathsf{int}(y, \mathsf{ext}(), t_3)) \xrightarrow{\gamma} [y \,|\, \mathsf{in}_1(t_3)]. \end{array}$$

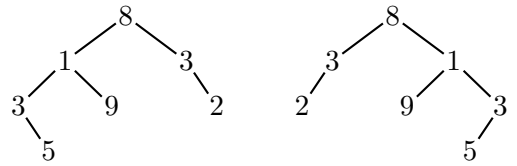
Figure 7.24: Inorder traversal by right rotations

right-leaning, degenerate tree, isomorphic to a stack, by repeatedly applying right rotations, top-down. Dually, we could compose left rotations and obtain a left-leaning, degenerate tree, whose inorder traversal is also equal to the inorder traversal of the original tree. The function $\mathsf{in}_1/1$ based on right rotations is given in FIGURE 7.24. Note how, in rule γ, we push the root y in the result as soon as we can, which would not be possible had we used left rotations instead, and thus we do *not* build the whole rotated tree, as in FIGURE 7.23.

The cost $\mathcal{C}_n^{\mathsf{in}_1}$ depends on the topology of the tree at hand. Firstly, let us note that rule α is used only once, on the rightmost external node. Secondly, if the tree to be traversed is already a right-leaning degenerate tree, rule β is not used and rule γ is used n times. Clearly, this is the best case and $\mathcal{B}_n^{\mathsf{in}_1} = n + 1$. Thirdly, we should remark that a right rotation brings exactly one more node (named x in $\mathsf{Rot}(x, y, t_1, t_2, t_3)$) into the *rightmost branch*, that is, the series of nodes starting with the root and reached using repeatedly right edges (for instance, in the initial tree in FIGURE 7.23, the rightmost branch is $[\mathsf{e}, \mathsf{g}, \mathsf{h}]$). Therefore, if we want to maximise the use of rule β, we must have an initial tree whose right subtree is empty, so the left subtree contains $n - 1$ nodes (the root belongs, by definition, to the rightmost branch): this yields

$$\mathcal{W}_n^{\mathsf{in}_1} = (n - 1) + (n + 1) = 2n.$$

Exercise Prove $\forall x, y, t_1, t_2, t_3.\mathsf{Rot}(x, y, t_1, t_2, t_3)$.



(a) t (b) $\mathsf{mir}(t)$

Figure 7.25: Mirroring a binary tree

Mirroring Let us define a function $\mathsf{mir/1}$ (*mirror*) such that $\mathsf{mir}(t)$ is the symmetric of the binary tree t with respect to an exterior vertical line. An example is given in FIGURE 7.25. It is easy to define this function:

$$\mathsf{mir}(\mathsf{ext}()) \xrightarrow{\sigma} \mathsf{ext}(); \quad \mathsf{mir}(\mathsf{int}(x, t_1, t_2)) \xrightarrow{\tau} \mathsf{int}(x, \mathsf{mir}(t_2), \mathsf{mir}(t_1)).$$

From the previous example, it is quite simple to postulate the property

$$\mathsf{InMir}(t)\colon \mathsf{in}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{in}(t)),$$

where $\mathsf{rev/1}$ reverses its stack argument (see definition (2.2) on page 40). This property is useful because the left-hand side of the equivalence is more costly than the right-hand side:

$$\mathcal{C}[\![\mathsf{in}(\mathsf{mir}(t))]\!] = \mathcal{C}_n^{\mathsf{mir}} + \mathcal{C}_n^{\mathsf{in}} = (2n+1) + (2n+2) = 4n+3,$$

to be compared with $\mathcal{C}[\![\mathsf{rev}(\mathsf{in}(t))]\!] = \mathcal{C}_n^{\mathsf{in}} + \mathcal{C}_n^{\mathsf{rev}} = (2n+2)+(n+2) = 3n+4$.

Structural induction on the immediate subtrees requires that we establish

- the basis $\mathsf{InMir}(\mathsf{ext}())$;
- the step $\forall t_1.\mathsf{InMir}(t_1) \Rightarrow \forall t_2.\mathsf{InMir}(t_2) \Rightarrow \forall x.\mathsf{InMir}(\mathsf{int}(x, t_1, t_2))$.

The former results from

$$\begin{array}{ll} \mathsf{in}(\underline{\mathsf{mir}}(\mathsf{ext}())) & \xrightarrow{\sigma} \underline{\mathsf{in}}(\mathsf{ext}()) \\ & \xrightarrow{\xi} \mathsf{in}(\mathsf{ext}(), [\,]) \\ & \xrightarrow{\pi} [\,] \\ & \xleftarrow{\zeta} \mathsf{rcat}([\,], [\,]) \\ & \xleftarrow{\epsilon} \mathsf{rev}([\,]) \\ & \xleftarrow{\pi} \mathsf{rev}(\underline{\mathsf{in}}(\mathsf{ext}(), [\,])) \\ & \xleftarrow{\xi} \mathsf{rev}(\underline{\mathsf{in}}(\mathsf{ext}())). \end{array}$$

Let us assume $\mathsf{InMir}(t_1)$ and $\mathsf{InMir}(t_2)$ and let $t := \mathsf{int}(x, t_1, t_2)$, for any x. We rewrite the left-hand side of the equivalence to be proved until we

reach the right-hand side or we get stuck:

$$\begin{aligned}
\mathsf{in}(\mathsf{mir}(t)) &= \mathsf{in}(\underline{\mathsf{mir}}(\mathsf{int}(x, t_1, t_2))) \\
&\xrightarrow{\tau} \underline{\mathsf{in}}(\mathsf{int}(x, \mathsf{mir}(t_2), \mathsf{mir}(t_1))) \\
&\xrightarrow{\xi} \underline{\mathsf{in}}(\mathsf{int}(x, \mathsf{mir}(t_2), \mathsf{mir}(t_1)), [\,]) \\
&\xrightarrow{\rho} \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \mathsf{in}(\mathsf{mir}(t_1), [\,])]) \\
&\xleftarrow{\zeta} \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \underline{\mathsf{in}}(\mathsf{mir}(t_1))]) \\
&\equiv \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \mathsf{rev}(\underline{\mathsf{in}}(t_1))]) \qquad (\mathsf{InMir}(t_1)) \\
&\xrightarrow{\xi} \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \underline{\mathsf{rev}}(\mathsf{in}(t_1, [\,]))]) \\
&\xrightarrow{\epsilon} \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \mathsf{rcat}(\mathsf{in}(t_1, [\,]), [\,])]).
\end{aligned}$$

We cannot use the induction hypothesis $\mathsf{InMir}(t_2)$ to get rid of $\mathsf{mir}(t_2)$. Close examination of the terms suggests to *weaken* the property and overload InMir with a new definition:

$$\mathsf{InMir}(t, s)\colon \mathsf{in}(\mathsf{mir}(t), s) \equiv \mathsf{rcat}(\mathsf{in}(t, [\,]), s).$$

We have $\mathsf{InMir}(t, [\,]) \Leftrightarrow \mathsf{InMir}(t)$. Now, we rewrite as follows:

$$\begin{aligned}
\mathsf{in}(\mathsf{mir}(t), s) &= \mathsf{in}(\underline{\mathsf{mir}}(\mathsf{int}(x, t_1, t_2)), s) \\
&\xrightarrow{\tau} \underline{\mathsf{in}}(\mathsf{int}(x, \mathsf{mir}(t_2), \mathsf{mir}(t_1)), s) \\
&\xrightarrow{\rho} \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \mathsf{in}(\mathsf{mir}(t_1), s)]) \\
&\equiv \mathsf{in}(\mathsf{mir}(t_2), [x \,|\, \mathsf{rcat}(\mathsf{in}(t_1, [\,]), s)]) \qquad (\mathsf{InMir}(t_1, s)) \\
&\equiv_0 \mathsf{rcat}(\mathsf{in}(t_2, [\,]), [x \,|\, \mathsf{rcat}(\mathsf{in}(t_1, [\,]), s)]) \\
&\xleftarrow{\eta} \underline{\mathsf{rcat}}([x \,|\, \mathsf{in}(t_2, [\,])], \mathsf{rcat}(\mathsf{in}(t_1, [\,]), s)),
\end{aligned}$$

where $(\equiv_0)$ is $\mathsf{InMir}(t_2, [x \,|\, \mathsf{rcat}(\mathsf{in}(t_1, [\,]), s)])$. The right-hand side now:

$$\mathsf{rcat}(\mathsf{in}(t, [\,]), s) = \mathsf{rcat}(\underline{\mathsf{in}}(\mathsf{int}(x, t_1, t_2), [\,]), s) \xrightarrow{\rho} \mathsf{rcat}(\mathsf{in}(t_1, [x \,|\, \mathsf{in}(t_2, [\,])]), s).$$

The two stuck expressions share the subterm $[x \mid \mathsf{in}(t_2, [\,])]$. The main difference is that the former expression contains two calls to rcat/2, instead of one in the latter. Can we find a way to have only one call in the former too? We need to find an expression equivalent to $\mathsf{rcat}(s, \mathsf{rcat}(t, u))$, whose shape is $\mathsf{rcat}(v, w)$, where v and w contain no call to rcat/2. Some examples quickly suggest

$$\mathsf{Rcat}(s, t, u)\colon \mathsf{rcat}(s, \mathsf{rcat}(t, u)) \equiv \mathsf{rcat}(\mathsf{cat}(t, s), u).$$

We actually do not need induction if we recall what we already proved:

- $\mathsf{CatRev}(s, t)\colon \mathsf{cat}(\mathsf{rev}_0(t), \mathsf{rev}_0(s)) \equiv \mathsf{rev}_0(\mathsf{cat}(s, t))$;
- $\mathsf{EqRev}(s)\colon \mathsf{rev}_0(s) \equiv \mathsf{rev}(s)$;
- $\mathsf{CatAssoc}(s, t, u)\colon \mathsf{cat}(s, \mathsf{cat}(t, u)) \equiv \mathsf{cat}(\mathsf{cat}(s, t), u)$;
- $\mathsf{RevCat}(s, t)\colon \mathsf{rcat}(s, t) \equiv \mathsf{cat}(\mathsf{rev}(s), t)$.

Then, we have the equivalences

$$\begin{aligned}
\mathsf{rcat}(s, \mathsf{rcat}(t,u)) &\equiv \mathsf{rcat}(s, \mathsf{cat}(\mathsf{rev}(t), u)) && (\mathsf{RevCat}(t,u))\\
&\equiv \mathsf{rcat}(s, \mathsf{cat}(\mathsf{rev}_0(t), u)) && (\mathsf{EqRev}(t))\\
&\equiv \mathsf{cat}(\mathsf{rev}(s), \mathsf{cat}(\mathsf{rev}_0(t), u)) && (\mathsf{RevCat}(s, \mathsf{cat}(\mathsf{rev}_0(t), u)))\\
&\equiv \mathsf{cat}(\mathsf{rev}_0(s), \mathsf{cat}(\mathsf{rev}_0(t), u)) && (\mathsf{EqRev}(s))\\
&\equiv \mathsf{cat}(\mathsf{cat}(\mathsf{rev}_0(s), \mathsf{rev}_0(t)), u) && (\mathsf{CatAssoc}(\mathsf{rev}(s), \mathsf{rev}(t), u))\\
&\equiv \mathsf{cat}(\mathsf{rev}_0(\mathsf{cat}(t,s)), u) && (\mathsf{CatRev}(s,t))\\
&\equiv \mathsf{cat}(\mathsf{rev}(\mathsf{cat}(t,s)), u) && (\mathsf{EqRev}(\mathsf{cat}(t,s)))\\
&\equiv \mathsf{rcat}(\mathsf{cat}(t,s), u) && (\mathsf{RevCat}(\mathsf{cat}(t,s), u))
\end{aligned}$$

Let us resume rewriting the first stuck expression:

$$\begin{aligned}
\mathsf{in}(\mathsf{mir}(t), s) &\equiv_0 \mathsf{rcat}([x \,|\, \mathsf{in}(t_2, [\,])], \mathsf{rcat}(\mathsf{in}(t_1, [\,]), s))\\
&\equiv_1 \mathsf{rcat}(\mathsf{cat}(\mathsf{in}(t_1, [\,]), [x \,|\, \mathsf{in}(t_2, [\,])]), s),
\end{aligned}$$

where $(\equiv_0)$ is a short-hand for the previous derivation and $(\equiv_1)$ is the instance $\mathsf{Rcat}([x \,|\, \mathsf{in}(t_2, [\,])], \mathsf{in}(t_1, [\,]), s)$. Another comparison of the stuck expressions reveals that we need to prove $\mathsf{cat}(\mathsf{in}(t, [\,]), s) \equiv \mathsf{in}(t, s)$. This equivalence is likely to be true, as it is similar to $\mathsf{CatPre}(t, s)$. Assuming this lemma, we conclude the proof. □

Exercises

1. Prove the missing lemma $\mathsf{CatIn}(t, s)$: $\mathsf{cat}(\mathsf{in}(t, [\,]), s) \equiv \mathsf{in}(t, s)$.

2. Define a function which builds a binary tree from its preorder and inorder traversals, assuming that its internal nodes are all distinct. Make sure its cost is linear in the number of internal nodes. Compare your solution with that of Mu and Bird (2003).

Postorder A *postorder* traversal of a non-empty binary tree consists in storing in a stack the nodes of the right subtree in postorder, followed by the nodes of the left subtree in postorder, and, in turn, by the root. For example, the nodes in postorder of the tree in FIGURE 7.3a on page 206 are $[5, 3, 9, 1, 2, 3, 8]$. Clearly, it is a depth-first traversal, like a preorder, but, unlike a preorder, it saves the root last in the resulting stack. This approach is summarised for internal nodes in FIGURE 7.9c on page 213. The difference with pre/1 and in/1 lies in the moment when the root is pushed in the accumulator. The function definition is given in FIGURE 7.26 on the next page. The meaning of the stack s in the call $\mathsf{post}(t, s)$ is the same as in $\mathsf{pre}(t, s)$, modulo ordering: s is made of the

$$\mathsf{post}(t) \xrightarrow{\lambda} \mathsf{post}(t, [\,]).$$

$$\begin{aligned} \mathsf{post}(\mathsf{ext}(), s) &\xrightarrow{\mu} s; \\ \mathsf{post}(\mathsf{int}(x, t_1, t_2), s) &\xrightarrow{\nu} \mathsf{post}(t_1, \mathsf{post}(t_2, [x \,|\, s])). \end{aligned}$$

Figure 7.26: Postorder traversal

$$\frac{\mathsf{npost}(0, t) \twoheadrightarrow (i, t')}{\mathsf{npost}(t) \to t'}. \qquad \mathsf{npost}(i, \mathsf{ext}()) \to (i, \mathsf{ext}());$$

$$\frac{\mathsf{npost}(i, t_1) \twoheadrightarrow (j, t_1') \qquad \mathsf{npost}(j, t_2) \twoheadrightarrow (k, t_2')}{\mathsf{npost}(i, \mathsf{int}(x, t_1, t_2)) \to (k+1, \mathsf{int}(k, t_1', t_2'))}.$$

Figure 7.27: Postorder numbering

contents, in postorder, of the nodes that follow, in postorder, the nodes of the subtree t. The cost is familiar as well: $\mathcal{C}_n^{\mathsf{post}} = \mathcal{C}_n^{\mathsf{in}} = \mathcal{C}_n^{\mathsf{pre}} = 2n + 2$.

An example of *postorder numbering* is shown in FIGURE 7.11b on page 214, so the postorder traversal of that tree yields $[0, 1, 2, 3, 4, 5, 6]$. Notice how the numbers increase along upwards paths. FIGURE 7.27 shows the program to number a binary tree in postorder. The root is numbered with the number coming up from the right subtree, following the pattern of a postorder traversal.

A proof Let $\mathsf{PreMir}(t)\colon \mathsf{pre}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{post}(t))$. Previous experience with proving $\mathsf{InMir}(t)$ leads us to weaken (generalise) the property in order to facilitate the proof: Let

$$\mathsf{PreMir}(t, s)\colon \mathsf{pre}(\mathsf{mir}(t), s) \equiv \mathsf{rcat}(\mathsf{post}(t, [\,]), s).$$

Clearly, $\mathsf{PreMir}(t, [\,]) \Leftrightarrow \mathsf{PreMir}(t)$. Let us then define

$$(t, s) \succ_{B\times S} (t', s') :\Leftrightarrow t \succ_B t' \text{ or } (t = t' \text{ and } s \succ_S s').$$

This is conceptually the same order as the one on the calls to $\mathsf{pre}/1$, in definition (7.7) on page 216. The minimum element for this lexicographic order is $(\mathsf{ext}(), [\,])$. Well-founded induction then requires that we prove

- the basis $\mathsf{PreMir}(\mathsf{ext}(), [\,])$;
- $\forall t, s.(\forall t', s'.(t, s) \succ_{B\times S} (t', s') \Rightarrow \mathsf{PreMir}(t', s')) \Rightarrow \mathsf{PreMir}(t, s)$.

$$
\begin{aligned}
\mathsf{pre}(\mathsf{mir}(t), s) &= \mathsf{pre}(\underline{\mathsf{mir}}(\mathsf{int}(x, t_1, t_2)), s) \\
&\xrightarrow{\tau} \underline{\mathsf{pre}}(\mathsf{int}(x, \mathsf{mir}(t_2), \mathsf{mir}(t_1)), s) \\
&\xrightarrow{\kappa} [x \,|\, \mathsf{pre}(\mathsf{mir}(t_2), \mathsf{pre}(\mathsf{mir}(t_1), s))] \\
&\equiv_0 [x \,|\, \mathsf{pre}(\mathsf{mir}(t_2), \mathsf{rcat}(\mathsf{post}(t_1, [\,]), s))] \\
&\equiv_1 [x \,|\, \mathsf{rcat}(\mathsf{post}(t_2, [\,]), \mathsf{rcat}(\mathsf{post}(t_1, [\,]), s))] \\
&\equiv_2 [x \,|\, \mathsf{rcat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,])), s)] \\
&\xleftarrow{\zeta} \underline{\mathsf{rcat}}([\,], [x \,|\, \mathsf{rcat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,])), s)]) \\
&\xleftarrow{\eta} \underline{\mathsf{rcat}}([x], \mathsf{rcat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,])), s)) \\
&\equiv_3 \mathsf{rcat}(\mathsf{cat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,])), [x]), s) \\
&\equiv_4 \mathsf{rcat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{cat}(\mathsf{post}(t_2, [\,]), [x])), s) \\
&\equiv_5 \mathsf{rcat}(\mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [x])), s) \\
&\equiv_6 \mathsf{rcat}(\mathsf{post}(t_1, \mathsf{post}(t_2, [x])), s) \\
&\xleftarrow{\nu} \mathsf{rcat}(\underline{\mathsf{post}}(\mathsf{int}(x, t_1, t_2), [\,]), s) \\
&= \mathsf{rcat}(\mathsf{post}(t, [\,]), s). \qquad \square
\end{aligned}
$$

Figure 7.28: Proof of $\mathsf{pre}(\mathsf{mir}(t), s) \equiv \mathsf{rcat}(\mathsf{post}(t, [\,]), s)$

The basis is proved as follows:

$$
\begin{aligned}
\mathsf{pre}(\underline{\mathsf{mir}}(\mathsf{ext}()), [\,]) &\xrightarrow{\sigma} \mathsf{pre}(\mathsf{ext}(), [\,]) \\
&\xrightarrow{\iota} [\,] \\
&\xleftarrow{\zeta} \mathsf{rcat}([\,], [\,]) \\
&\xleftarrow{\mu} \mathsf{rcat}(\underline{\mathsf{post}}(\mathsf{ext}(), [\,]), [\,]).
\end{aligned}
$$

Let $t := \mathsf{int}(x, t_1, t_2)$. In FIGURE 7.28, we have the rewrites of the left-hand side, where

- $(\equiv_0)$ is $\mathsf{PreMir}(t_1, s)$, an instance of the induction hypothesis;
- $(\equiv_1)$ is $\mathsf{PreMir}(t_2, \mathsf{rcat}(\mathsf{post}(t_1, [\,]), s))$, as inductive hypothesis;
- $(\equiv_2)$ is $\mathsf{Rcat}(\mathsf{post}(t_2, [\,]), \mathsf{post}(t_1, [\,]), s)$;
- $(\equiv_3)$ is $\mathsf{Rcat}([x], \mathsf{cat}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,])), s)$;
- $(\equiv_4)$ is $\mathsf{CatAssoc}(\mathsf{post}(t_1, [\,]), \mathsf{post}(t_2, [\,]), [x])$;
- $(\equiv_5)$ is $\mathsf{CatPost}(t_2, [x])$ if $\mathsf{CatPost}(t, s)\colon \mathsf{cat}(\mathsf{post}(t), s) \equiv \mathsf{post}(t, s)$;
- $(\equiv_6)$ is $\mathsf{CatPost}(t_1, \mathsf{post}(t_2, [x]))$.

Then $\mathsf{CatPost}(t, s) \Rightarrow \mathsf{PreMir}(t, s) \Rightarrow \mathsf{pre}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{post}(t))$. $\square$

Duality The dual theorem $\mathsf{PostMir}(t)\colon \mathsf{post}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{pre}(t))$ can be proved in at least two ways: either we design a new proof in the spirit of the proof of $\mathsf{PreMir}(t)$, or we take advantage of the fact that the theorem

is an equivalence and we produce equivalent but simpler theorems. Let us do the latter and start by considering $\mathsf{PreMir}(\mathsf{mir}(t))$ and proceed by finding equivalent expressions on both sides of the equivalence, until we reach $\mathsf{PostMir}(t)$:

$$\begin{aligned}
\mathsf{pre}(\mathsf{mir}(\mathsf{mir}(t))) &\equiv \mathsf{rev}(\mathsf{post}(\mathsf{mir}(t))) && (\mathsf{PreMir}(\mathsf{mir}(t))) \\
\mathsf{pre}(t) &\equiv \mathsf{rev}(\mathsf{post}(\mathsf{mir}(t))) && (\mathsf{InvMir}(t)) \\
\mathsf{rev}(\mathsf{pre}(t)) &\equiv \mathsf{rev}(\mathsf{rev}(\mathsf{post}(\mathsf{mir}(t)))) \\
\mathsf{rev}(\mathsf{pre}(t)) &\equiv \mathsf{post}(\mathsf{mir}(t)) && (\mathsf{InvRev}(\mathsf{post}(\mathsf{mir}(t)))),
\end{aligned}$$

where $\mathsf{InvMir}(t)\colon \mathsf{mir}(\mathsf{mir}(t)) \equiv t$ and $\mathsf{InvRev}(s) :\Leftrightarrow \mathsf{Inv}(s) \wedge \mathsf{EqRev}(s)$. □

Exercises

1. Prove the lemma $\mathsf{CatPost}(t, s)\colon \mathsf{cat}(\mathsf{post}(t, [\,]), s) \equiv \mathsf{post}(t, s)$.
2. Use $\mathsf{rev}_0/1$ instead of $\mathsf{rev}/1$ in $\mathsf{InMir}(t)$ and $\mathsf{PreMir}(t)$. Are the proofs easier?
3. Prove the missing lemma $\mathsf{InvMir}(t)\colon \mathsf{mir}(\mathsf{mir}(t)) \equiv t$.
4. Can you build a binary tree from its postorder and inorder traversals, assuming that its internal nodes are all distinct?

Level order The *level* l in a tree is a stack of nodes in preorder whose internal path lengths are l. In particular, the root is the only node at level 0. In the tree of FIGURE 7.3a on page 206, $[3, 9, 2]$ is level 2. To understand the preorder condition, we need to consider the preorder numbering of the tree, shown in FIGURE 7.12 on page 214 with preorder numbers as left exponents to the contents. This way, there is no more ambiguity when referring to nodes. For instance, $[3, 9, 2]$ was in fact ambiguous because there are two nodes whose associated data is 3. We meant that $[{}^{2}3, {}^{4}9, {}^{6}2]$ is the level 2 because these nodes all have internal path lengths 2 *and* have increasing preorders $(2, 4, 6)$.

A *level-order* traversal consists in making a stack with the nodes of all the levels by increasing path lengths. For instance, the level order of the tree in FIGURE 7.3a on page 206 is $[8, 1, 3, 3, 9, 2, 5]$. Because this method visits the sibling of a node before its children, it is said *breadth-first*.

In FIGURE 7.11d on page 214 is shown the *level-order numbering* of the tree in FIGURE 7.12 on page 214, more often called *breadth numbering*. (Mind the common misspellings 'bread numbering' and 'breath numbering.') Notice how the numbers increase along downwards path between nodes, as in a preorder numbering.

We may now realise that the notion of level in a tree is not straightforward. The reason is simple: the nodes in a level are not siblings, except

$$\mathsf{bf}_0(t) \to \mathsf{bf}_1([t]). \quad \mathsf{bf}_1([\,]) \to [\,]; \quad \frac{\mathsf{def}(f) \twoheadrightarrow (r, f')}{\mathsf{bf}_1(f) \twoheadrightarrow \mathsf{cat}(r, \mathsf{bf}_1(f'))}.$$

Figure 7.29: Level order $\mathsf{bf}_0/1$

in level 1, so, in general, we cannot expect to build a level of a tree $\mathsf{int}(x, t_1, t_2)$ by means of levels in t_1 and t_2 alone, that is, with a big-step design. As a consequence, a small-step approach is called for, standing in contrast, for example, with $\mathsf{size}/1$ in (7.1) on page 207.

Let $\mathsf{bf}_0/1$ (*breadth-first*) be the function such that $\mathsf{bf}_0(t)$ is the stack of nodes of t in level order. It is partially defined in FIGURE 7.29. If we imagine that we cut off the root of a binary tree, we obtain the immediate subtrees. If, in turn, we cut down these trees, we obtain more subtrees. This suggests that we should better work on general forests instead of trees, one or two at a time.

The cutting function is $\mathsf{def}/1$ (*deforest*), defined in FIGURE 7.30, such that $\mathsf{def}(f)$, where f represents a forest, evaluates in a pair (r, f'), where r are the roots in preorder of the trees in f, and f' is the immediate subforest of f. (Beware, the word deforestation is used by scholars of functional languages with a different meaning, but it will do for us, as we already encountered a function $\mathsf{cut}/2$.) Note how, in FIGURE 7.30, the inference rule augments the partial level r with the root x, and how t_2 is pushed before t_1 onto the rest of the immediate forest of f, to be processed later by $\mathsf{bf}_0/1$. Instead of building the stack of levels $[[8], [1, 3], [3, 9, 2], [5]]$, we actually flatten step by step simply by calling $\mathsf{cat}/2$ in rule μ. If we really want the levels, we would write $[r \mid \mathsf{bf}_1(f')]$ instead of $\mathsf{cat}(r, \mathsf{bf}_1(f'))$, which, by the way, reduces the cost.

The underlying concept here is that of the *traversal of a forest*. Except in inorder, all the traversals we have discussed naturally carry over to binary forests: the preorder traversal of a forest consists in the preorder traversal of the first tree in the forest, followed by the preorder traversal of the rest of the forest. Same logic for postorder and level order. This uniformity stems from the fact that all these traversals are performed

$$\mathsf{def}([\,]) \to ([\,], [\,]); \quad \mathsf{def}([\mathsf{ext}() \mid f]) \to \mathsf{def}(f); \quad \frac{\mathsf{def}(f) \twoheadrightarrow (r, f')}{\mathsf{def}([\mathsf{int}(x, t_1, t_2) \mid f]) \twoheadrightarrow ([x \mid r], [t_1, t_2 \mid f'])}.$$

Figure 7.30: Deforestation

rightwards, to wit, a left child is visited just before its sibling. The notion of height of a tree also extends naturally to a forest: the height of a forest is the maximum height of each individual tree. The reason why this is simple is because height is a purely vertical view of a tree, thus independent of the siblings' order.

To assess now the cost $\mathcal{C}_{n,h}^{\mathsf{bf}_0}$ of the call $\mathsf{bf}_0(t)$, where n is the size of the binary tree t and h is its height, it is convenient to work with *extended levels*. An extended level is a level where external nodes are included (they are not implicitly numbered in preorder because external nodes are indistinguishable). For example, the extended level 2 of the tree in FIGURE 7.12 on page 214 is $[^2 3, {}^4 9, \square, {}^6 2]$. If it is needed to draw a contrast with the other kind of level, we may call the latter *pruned levels*, which is consistent with our terminology in FIGURE 7.3 on page 206. Note that there is always one more extended level than pruned levels, made entirely of external nodes. (We avoided writing that these are the highest nodes, as the trouble with the term 'height' is that it really makes sense if the trees are laid out with the root at the bottom of the page. That is perhaps why some authors prefer the less confusing concept of *depth*, in use in graph theory. For a survey of the different manners of drawing trees, see Knuth (1997) in its section 2.3.) In other words, $l_h = 0$, where l_i is the number of internal nodes on the extended level i.

- Rule ι is used once;
- rule κ is used once;
- rules λ and μ are used once for each extended level of the original tree; these amount to $2(h+1)$ calls;
- the cost of $\mathsf{cat}(r, \mathsf{loc}_1(f'))$ is the length of the level r, plus one, thus the cumulative cost of concatenation is $\sum_{i=0}^{h} \mathcal{C}_{l_i}^{\mathsf{cat}} = n + h + 1$;
- rule ϵ is used once per extended level, that is, $h+1$ times;
- rule ζ is used once per external node, that is, $n+1$ times;
- rules η and θ are used once per internal node, that is $2n$ times.

Gathering these enumerations, we find that

$$\mathcal{C}_{n,h}^{\mathsf{bf}_0} = 4n + 4h + 7.$$

By definition, the minimum cost is $\mathcal{B}_n^{\mathsf{bf}_0} = \min_h \mathcal{C}_{n,h}^{\mathsf{bf}_0}$. The height is minimum when the binary tree is *perfect*, that is, when all levels are full (see FIGURE 2.43 on page 91). In this case, $l_i = 2^i$, for $0 \leqslant i \leqslant h-1$, and, by extension, there are 2^h external nodes. Theorem 5 on page 206 yields the equation $n+1 = 2^h$, so $h = \lg(n+1)$, and $\mathcal{B}_n^{\mathsf{bf}_0} = 4n + 4\lg(n+1) + 7 \sim 4n$.

The maximum cost is obtained by maximising the height, while keeping the size constant. This happens for degenerate trees, as the ones shown in FIGURE 7.4 on page 209 and FIGURE 7.31. Here, $h = n$ and the cost is $\mathcal{W}_n^{\mathsf{bf}_0} = 8n + 7 \sim 8n$.

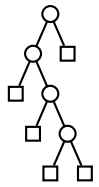


Figure 7.31

By contrast, we found programs that perform preorder, postorder and inorder traversals in $2n + 2$ function calls. It is possible to reduce the cost by using a different design, based on $\mathsf{pre}_3/1$ in FIGURE 7.6 on page 211. The difference is that, instead of using a stack to store subtrees to be traversed later, we use a queue, a linear data structure introduced in section 2.5. Consider in FIGURE 7.32 the algorithm at work on the same example found in FIGURE 7.7 on page 211. Keep in mind that trees are dequeued on the right side of the forest and enqueued on the left. (Some authors prefer the other way.) The root of the next tree to be dequeued is circled.

In order to compare with $\mathsf{pre}_4/1$ in FIGURE 7.6 on page 211, we write $x \prec q$ instead of $\mathsf{enq}(x, q)$, and $q \succ x$ instead of (q, x). The empty queue is noted $\ominus$. More importantly, we will allow these expressions in the patterns of $\mathsf{bf}_2/1$, which performs a level-order traversal in FIGURE 7.33. The difference in data structure (accumulator) has already been mentioned: $\mathsf{pre}_4/1$ uses a stack and $\mathsf{bf}_2/1$ a queue, but, as far as algorithms are concerned, they differ only in the relative order in which t_1 and t_2 are added to the accumulator.

In section 2.5, we saw how to implement a queue with two stacks as $\mathsf{q}(r, f)$: the rear stack r, where items are pushed (logically enqueued), and the front stack f, from whence items are popped (logically dequeued). Also, we defined enqueueing by $\mathsf{enq}/2$ in (2.10), on page 63, and dequeueing with $\mathsf{deq}/1$ in (2.11). This allows us to refine the definition of $\mathsf{bf}_2/1$ into $\mathsf{bf}_3/1$ in FIGURE 7.34 on the next page.

We can specialise the program further, so we save some memory by

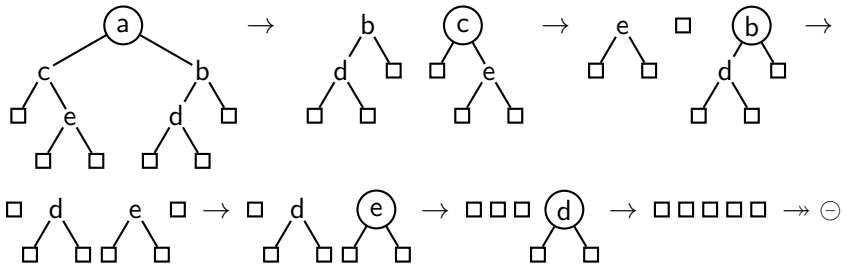


Figure 7.32: A level-order traversal with a queue

$$\begin{array}{r}\mathsf{bf}_1(t) \to \mathsf{bf}_2(t \prec \ominus).\\[2ex]\mathsf{bf}_2(\ominus) \to [\,];\\ \mathsf{bf}_2(q \succ \mathsf{ext}()) \to \mathsf{bf}_2(q);\\ \mathsf{bf}_2(q \succ \mathsf{int}(x, t_1, t_2)) \to [x \,|\, \mathsf{bf}_2(t_2 \prec t_1 \prec q)].\end{array}$$

Figure 7.33: Abstract level-order traversal with a queue

$$\mathsf{bf}_3(t) \to \mathsf{bf}_4(\mathsf{enq}(t, \mathsf{q}([\,],[\,]))). \qquad \mathsf{bf}_4(\mathsf{q}([\,],[\,])) \to [\,];$$
$$\frac{\mathsf{deq}(q) \twoheadrightarrow (q', \mathsf{ext}())}{\mathsf{bf}_4(q) \twoheadrightarrow \mathsf{bf}_4(q')}; \qquad \frac{\mathsf{deq}(q) \twoheadrightarrow (q', \mathsf{int}(x, t_1, t_2))}{\mathsf{bf}_4(q) \twoheadrightarrow [x \,|\, \mathsf{bf}_4(\mathsf{enq}(t_2, \mathsf{enq}(t_1, q')))]}.$$

Figure 7.34: Refinement of FIGURE 7.33

not using the constructor $\mathsf{q}/2$ and remembering that its first argument is the rear and the second is the front. Moreover, instead of calling $\mathsf{deq}/1$ and $\mathsf{enq}/2$, we can expand their definitions and merge them with the definition at hand. The result is shown in FIGURE 7.35. Recall that $\mathsf{rcat}/2$ (*reverse and concatenate*) is defined in equation (2.2) on page 40. Note that $\mathsf{bf}_4(\mathsf{enq}(t, \mathsf{q}([\,],[\,])))$ has been optimised in $\mathsf{bf}([\,],[t])$ in order to save a stack reversal. The definition of $\mathsf{bf}/1$ can be considered the most *concrete* of the refinements, the more *abstract* program being the original definition of $\mathsf{bf}_1/1$. The former is shorter than $\mathsf{bf}_0/1$, but the real gain is the cost. Let n be the size of the binary tree at hand and h its height. Rules are applied as follows:

- rule ν is used once;
- rule ξ is used once;
- rule π is used once per level, except the first one (the root), hence, in total h times;

$$\begin{array}{r}\mathsf{bf}(t) \xrightarrow{\nu} \mathsf{bf}([\,],[t]).\\[2ex]\mathsf{bf}([\,],[\,]) \xrightarrow{\xi} [\,];\\ \mathsf{bf}([t \,|\, r],[\,]) \xrightarrow{\pi} \mathsf{bf}([\,], \mathsf{rcat}(r,[t]));\\ \mathsf{bf}(r, [\mathsf{ext}() \,|\, f]) \xrightarrow{\rho} \mathsf{bf}(r, f);\\ \mathsf{bf}(r, [\mathsf{int}(x, t_1, t_2) \,|\, f]) \xrightarrow{\sigma} [x \,|\, \mathsf{bf}([t_2, t_1 \,|\, r], f)].\end{array}$$

Figure 7.35: Refinement of FIGURE 7.34

- all levels but the first (the root) are reversed by rev/1, accounting $\sum_{i=1}^{h} \mathcal{C}_{e_i}^{\mathsf{rev}} = \sum_{i=1}^{h}(e_i+2) = (n-1)+(n+1)+2h = 2n+2h$, where e_i is the number of nodes on the extended level i;
- rule ρ is used once per external node, that is, $n+1$;
- rule σ is used once per internal node, so n times.

Gathering all these enumerations yields the formula

$$\mathcal{C}_{n,h}^{\mathsf{bf}} = 4n + 3h + 3.$$

As with $\mathsf{bf}_0/1$, the minimum cost happens here when $h = \lg(n+1)$, so $\mathcal{B}_n^{\mathsf{bf}} = 4n + 3\lg(n+1) + 3 \sim 4n$. The maximum cost occurs when $h = n$, so $\mathcal{W}_n^{\mathsf{bf}} = 7n + 3 \sim 7n$. We can now compare $\mathsf{bf}_0/1$ and bf/1: $\mathcal{C}_{n,h}^{\mathsf{bf}} < \mathcal{C}_{n,h}^{\mathsf{bf}_0}$ and the difference in cost is most observable in their worst cases, which are both degenerate trees. Therefore bf/1 is preferable in any case.

Termination With the aim to prove the termination of bf/2, we reuse the lexicographic order on pairs of stacks, based on the immediate substack order ($\succ_S$) that proved the termination of mrg/2 on page 125:

$$(s,t) \succ_{S^2} (s',t') :\Leftrightarrow s \succ_S s' \text{ or } (s = s' \text{ and } t \succ_S t').$$

Unfortunately, $(\succ_{S^2})$ fails to monotonically order (with respect to the rewrite relation) the left-hand side and right-hand side of rule σ, because of $(r, [\mathsf{int}(x,t_1,t_2) \mid f]) \not\succ_{S^2} ([t_2, t_1 \mid r], f)$. Another approach consists in defining a well-founded order on the number of nodes in a pair of forests:

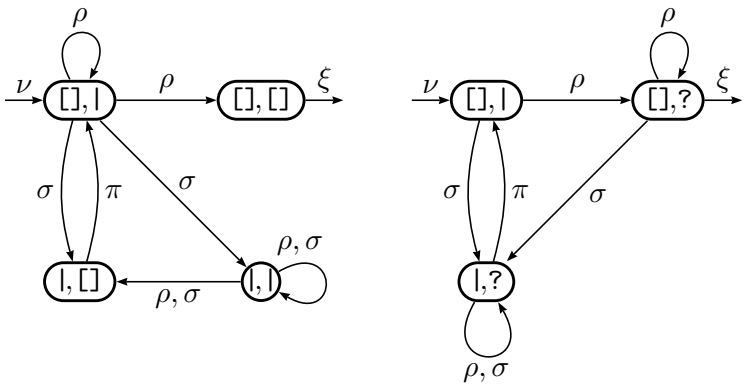
$$(r,f) \succ_{S^2} (r',f') :\Leftrightarrow \mathsf{dim}(r) + \mathsf{dim}(f) > \mathsf{dim}(r') + \mathsf{dim}(f'),$$

with

$$\mathsf{dim}([\,]) \to [\,]; \qquad \mathsf{dim}([t \mid f]) \to \mathsf{size}(t) + \mathsf{dim}(f).$$

where size/1 is defined in (7.1) on page 207. This is a kind of polynomial measure on dependency pairs, as exemplified with flat/1 on page 61. Here, $\mathcal{M}[\![\mathsf{bf}(s,t)]\!] := \mathsf{dim}(s) + \mathsf{dim}(t)$. Unfortunately, this order monotonically fails on rule π, because $(r, [\,]) \not\succ_{S^2} ([\,], \mathsf{rev}(r))$.

The conundrum can be lifted if we visualise the complete set of traces of bf/2 in a compact manner. If we assume that rev/1 is a constructor, the right-hand sides either contain no call or exactly one recursive call. The traces of calls to such definitions are nicely represented as finite automata. An example of a deterministic finite automaton (DFA) was given in FIGURE 5.14 on page 188. Here, a transition is a rewrite rule and a state corresponds to an abstraction of the input. In the case of bf/2, the input is a pair of stacks. Let us decide for the moment that



(a) Non-deterministic (NFA) (b) Deterministic (DFA)

Figure 7.36: Traces of bf/2 as finite automata

we will only take into account whether a stack is empty or not, yielding four states. Let '|' denote an arbitrary non-empty stack. Examining the definitions of bf/1 and bf/2 in FIGURE 7.35 on page 232, we see that

- rule ξ applies to the state $([\,], [\,])$ only;
- rule π applies to the state $(|, [\,])$, and leads to a state $([\,], |)$;
- rule ρ applies to the states $([\,], |)$ and $(|, |)$, and leads to any state;
- rule σ applies to the states $([\,], |)$ and $(|, |)$, and leads to the states $(|, [\,])$ and $(|, |)$.

In FIGURE 7.36a, we gather all this connectivity into a finite automaton. Note that, by definition, the initial state has an incoming edge ν without source and the final state has an outgoing edge ξ without destination. A trace is any sequence of transitions from the initial state $([\,], |)$ to the final state $([\,], [\,])$, for example, $\nu\rho^p\sigma^q\pi\rho\xi$, with $p \geqslant 0$ and $q \geqslant 2$. This automaton is called a *non-deterministic finite automaton* (NFA) because a state may have more than one outgoing transition with the same label (consider the initial state and the two transitions labelled σ, for example).

It is always possible to construct a *deterministic finite automaton* (DFA) equivalent to a given non-deterministic finite automaton (Perrin, 1990, Hopcroft et al., 2003). The outgoing transitions of each state of the former have a unique label. Equivalence means that the sets of traces of each automaton are the same. If '?' denotes a stack, empty or not, FIGURE 7.36b shows an equivalent DFA for the traces of bf/1 and bf/2.

As we observed earlier, using the well-founded order $(\succ_{S^2})$ based on the sizes of the stacks, all transitions $x \to y$ in the DFA satisfy $x \succ_{S^2} y$, except π, for which $x =_{S^2} y$ holds (the total number of nodes is invariant). We can nevertheless conclude that bf/2 terminates because the only way

$$\frac{\mathsf{bfn}_2(0, t \prec \ominus) \twoheadrightarrow \ominus \succ t'}{\mathsf{bfn}_1(t) \twoheadrightarrow t'}.$$

$$\mathsf{bfn}_2(i, \ominus) \to \ominus; \qquad \mathsf{bfn}_2(i, q \succ \mathsf{ext}()) \to \mathsf{ext}() \prec \mathsf{bfn}_2(i, q);$$

$$\frac{\mathsf{bfn}_2(i+1, t_2 \prec t_1 \prec q) \twoheadrightarrow q' \succ t_1' \succ t_2'}{\mathsf{bfn}_2(i, q \succ \mathsf{int}(x, t_1, t_2)) \twoheadrightarrow \mathsf{int}(i, t_1', t_2') \prec q'}.$$

Figure 7.37: Abstract breadth-first numbering

to have non-termination would be the existence of a π-*circuit*, that is, a series of successive transitions from a state to itself all labelled with π, along which the number of nodes remains identical. In fact, all traces have π followed by ρ or σ.

Yet another spin on this matter would be to prove, by examining all rules in isolation and all compositions of two rules, that

$$x \to y \Rightarrow x \succcurlyeq_{S^2} y \quad \text{and} \quad x \xrightarrow{2} y \Rightarrow x \succ_{S^2} y.$$

Consequently, if $n > 0$, then $x \xrightarrow{2n} y$ entails $x \succ_{S^2} y$, because $(\succ_{S^2})$ is transitive. Furthermore, if $x \xrightarrow{2n} y \to z$, then $x \succ_{S^2} y \succcurlyeq_{S^2} z$, hence $x \succ_{S^2} z$. In the end, $x \xrightarrow{n} y$ implies $x \succ_{S^2} y$, for all $n > 1$. □

Breadth-first numbering As mentioned earlier, FIGURE 7.11d on page 214 shows an example of breadth-first numbering. This problem has received notable attention (Jones and Gibbons, 1993, Gibbons and Jones, 1998, Okasaki, 2000) because functional programmers usually feel a bit challenged by this problem. A good approach consists in modifying the function $\mathsf{bf}_1/2$ in FIGURE 7.33 on page 232 so that it builds a tree instead of a stack. We do so by enqueueing immediate subtrees, so they are recursively numbered when they are dequeued, and by incrementing a counter, initialised with 0, every time a non-empty tree has been dequeued.

Consider $\mathsf{bfn}_1/1$ and $\mathsf{bfn}_2/2$ (*breadth-first numbering*) in FIGURE 7.37, and compare them with the definitions in FIGURE 7.33 on page 232. In particular, notice how, in contrast with $\mathsf{bf}_1/2$, external nodes are enqueued instead of being discarded, because they are later needed to make the numbered tree.

In FIGURE 7.38 on the next page is shown an example, where the numbers on the left represent the values of i (the first argument of $\mathsf{bf}_2/2$), the downward rewrites define the successive states of the working queue (the second argument of $\mathsf{bf}_2/2$), and the upward rewrites show the successive states of the resulting queue (right-hand side of $\mathsf{bf}_2/2$). Recall that

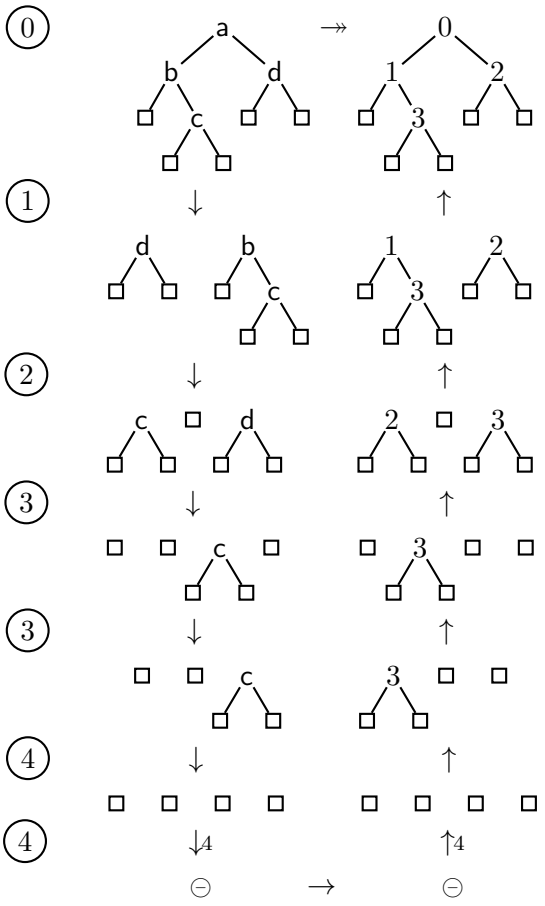


Figure 7.38: Example of breadth-first numbering

trees are enqueued on the left and dequeued on the right (other authors may use the opposite convention, as Okasaki (2000)) and pay attention to the fact that, in the vertical rewrites on the left, t_1 is enqueued first whilst, on the right, t'_2 is dequeued first, which appears when contrasting $t_2 \prec t_1 \prec q = t_2 \prec (t_1 \prec q)$ and $q' \succ t'_1 \succ t'_2 = (q' \succ t'_1) \succ t'_2$.

We can refine $\mathsf{bfn}_1/1$ and $\mathsf{bfn}_2/2$ by introducing explicitly the function calls for enqueueing and dequeueing, as shown in FIGURE 7.39 on the facing page, which could be contrasted with FIGURE 7.34 on page 232.

Exercises

1. How would you proceed to prove the correctness of $\mathsf{bfn}/1$?
2. Find the cost $\mathcal{C}_n^{\mathsf{bfn}}$ of $\mathsf{bfn}(t)$, where n is the size of t.

$$\frac{\mathsf{deq}(\mathsf{bfn}(0,\mathsf{q}([\,],[t]))) \twoheadrightarrow (\mathsf{q}([\,],[\,]),t')}{\mathsf{bfn}_3(t) \twoheadrightarrow t'}.$$

$$\mathsf{bfn}_4(i,\mathsf{q}([\,],[\,])) \to \mathsf{q}([\,],[\,]); \quad \frac{\mathsf{deq}(q) \twoheadrightarrow (q',\mathsf{ext}())}{\mathsf{bfn}_4(i,q) \twoheadrightarrow \mathsf{enq}(\mathsf{ext}(),\mathsf{bfn}_4(i,q'))};$$

$$\frac{\begin{array}{c}\mathsf{deq}(q) \twoheadrightarrow (q_1,\mathsf{int}(x,t_1,t_2)) \\ \mathsf{deq}(\mathsf{bfn}_4(i+1,\mathsf{enq}(t_2,\mathsf{enq}(t_1,q_1)))) \twoheadrightarrow (q_2,t_2') \qquad \mathsf{deq}(q_2) \twoheadrightarrow (q',t_1')\end{array}}{\mathsf{bfn}_4(i,q) \twoheadrightarrow \mathsf{enq}(\mathsf{int}(i,t_1',t_2'),q')}$$

Figure 7.39: Refinement of FIGURE 7.37 on page 235

$$\mathsf{per}(\mathsf{ext}()) \to \mathsf{true}(0); \quad \frac{\mathsf{per}(t_1) \twoheadrightarrow \mathsf{true}(h) \qquad \mathsf{per}(t_2) \twoheadrightarrow \mathsf{true}(h)}{\mathsf{per}(\mathsf{int}(x,t_1,t_2)) \to \mathsf{true}(h+1)};$$
$$\mathsf{per}(t) \to \mathsf{false}().$$

Figure 7.40: Abstract checking of perfection

7.2 Classic shapes

In this section, we briefly review some particular binary trees which are useful in assessing the extremal costs of many algorithms.

Perfection We already mentioned what a *perfect binary tree* is in the context of optimal sorting (see FIGURE 2.43 on page 91). One way to define such trees is to say that all their external nodes belong to the same level or, equivalently, the immediate subtrees of any node have same height. (The height of an external node is 0.) In FIGURE 7.40 is shown the definition of per/1 (*perfection*). If the tree t is perfect, we also know its height h: $\mathsf{per}(t) \twoheadrightarrow \mathsf{true}(h)$. Note that the rules are ordered, so the last one may only apply when the previous ones have not been matched. A refinement without inference rules is shown in FIGURE 7.41 on the following page, where $\mathsf{per}_0(t_1)$ is evaluated before $\mathsf{per}_0(t_2)$ since $\mathsf{t}(\mathsf{per}_0(t_1),\mathsf{per}_0(t_2))$ is inefficient if $\mathsf{per}_0(t_1) \twoheadrightarrow \mathsf{false}()$.

Completeness A binary tree is *complete* if the children of every internal node are either two external nodes or two internal nodes themselves. An example is provided in FIGURE 7.42 on the next page.

$$\begin{aligned}
\mathsf{per}_0(\mathsf{ext}()) &\to \mathsf{true}(0);\\
\mathsf{per}_0(\mathsf{int}(x,t_1,t_2)) &\to \mathsf{t}_1(\mathsf{per}_0(t_1),t_2).\\[1ex]
\mathsf{t}_1(\mathsf{false}(),t_2) &\to \mathsf{false}();\\
\mathsf{t}_1(h,t_2) &\to \mathsf{t}_2(h,\mathsf{per}_0(t_2)).\\[1ex]
\mathsf{t}_2(\mathsf{true}(h),\mathsf{true}(h)) &\to \mathsf{true}(h+1);\\
\mathsf{t}_2(h,x) &\to \mathsf{false}().
\end{aligned}$$

Figure 7.41: Refinement of FIGURE 7.40

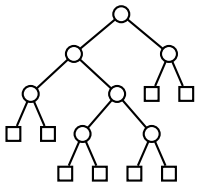


Figure 7.42

Recursively, a given tree is complete if, and only if, its immediate subtrees are complete. This is the same rule we used for perfection. In other words, perfection and completeness are propagated fron the bottom up. Therefore, we need to decide what to say about the external nodes, in particular, the empty tree. If we decide that the latter is complete, then $\mathsf{int}(x,\mathsf{ext}(),\mathsf{int}(y,\mathsf{ext}(),\mathsf{ext}()))$ would, incorrectly, be deemed complete. If not, leaves $\mathsf{int}(x,\mathsf{ext}(),\mathsf{ext}())$ would, incorrectly, be found to be incomplete. Thus, we can choose either option and handle the problematic case separately; for example, we may choose that external nodes are incomplete trees, but leaves are complete trees. The program is shown in FIGURE 7.43. The last rule applies either if $t = \mathsf{ext}()$ or $t = \mathsf{int}(x,t_1,t_2)$, with t_1 or t_2 incomplete.

$$\begin{aligned}
&\mathsf{comp}(\mathsf{int}(x,\mathsf{ext}(),\mathsf{ext}())) \to \mathsf{true}();\\
&\frac{\mathsf{comp}(t_1) \twoheadrightarrow \mathsf{true}() \qquad \mathsf{comp}(t_2) \twoheadrightarrow \mathsf{true}()}{\mathsf{comp}(\mathsf{int}(x,t_1,t_2)) \twoheadrightarrow \mathsf{true}()};\quad \mathsf{comp}(t) \to \mathsf{false}().
\end{aligned}$$

Figure 7.43: Checking completeness

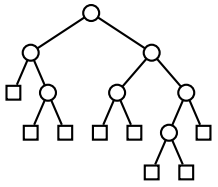


Figure 7.44

Balance The last interesting kind of binary trees is the *balanced trees*. There are two sorts of criteria to define balance: either height or size (Nievergelt and Reingold, 1972, Hirai and Yamamoto, 2011). In the latter case, siblings are roots of trees with similar sizes; in the former case, they have similar height. The usual criterion being height, we will use it in the

following. Depending on the algorithm, what 'similar' means may vary. For example, we may decide that two trees whose heights differ at most by 1 have similar heights. See FIGURE 7.44 on the preceding page for an instance. Let us start with a definition of the height of a binary tree and then modify it to obtain a function checking the balance:

$$\begin{aligned} \mathsf{height}(\mathsf{ext}()) &\to 0; \\ \mathsf{height}(\mathsf{int}(x, t_1, t_2)) &\to 1 + \max\{\mathsf{height}(t_1), \mathsf{height}(t_2)\}. \end{aligned}$$

The modification is shown in FIGURE 7.45, where the inference rule of $\mathsf{bal}_0/1$ is needed to check the condition $|h_1 - h_2| \leqslant 1$. Note that a perfect tree is balanced ($h_1 = h_2$).

$$\begin{array}{c} \mathsf{bal}_0(\mathsf{ext}()) \to \mathsf{true}(0); \\ \dfrac{\mathsf{bal}_0(t_1) \twoheadrightarrow \mathsf{true}(h_1) \quad \mathsf{bal}_0(t_2) \twoheadrightarrow \mathsf{true}(h_2) \quad |h_1 - h_2| \leqslant 1}{\mathsf{bal}_0(\mathsf{int}(x, t_1, t_2)) \twoheadrightarrow \mathsf{true}(1 + \max\{h_1, h_2\})}; \\ \mathsf{bal}_0(t) \to \mathsf{false}(). \end{array}$$

Figure 7.45: Checking balance

7.3 Tree encodings

In general, many binary trees yield the same preorder, postorder or inorder traversal, so it is not possible to rebuild the original tree from one traversal alone. The problem of uniquely representing a binary tree by a linear structure is called *tree encoding* (Mäkinen, 1991) and is related to the problem of generating all binary trees of a given size (Knuth, 2011, 7.2.1.6). One simple approach consists in extending a traversal with the external nodes; this way, enough information from the binary tree is retained in the encoding, allowing us to unambiguously go back to the original tree.

The encoding function $\mathsf{epost}/1$ (*extended postorder*) in FIGURE 7.46 on the next page, is a simple modification of $\mathsf{post}/1$ in FIGURE 7.26 on page 226. For example, the tree in FIGURE 7.11b on page 214 yields $[\Box, \Box, \Box, 0, 1, \Box, \Box, 2, 3, \Box, \Box, \Box, 4, 5, 6]$, where $\Box$ stands for $\mathsf{ext}()$. Since a binary tree with n internal nodes has $n+1$ external nodes (see theorem 5 on page 206), the cost is straightforward to find: $\mathcal{C}_n^{\mathsf{epost}} = 2n + 2$.

We already noticed that the postorder of the nodes increases along upwards paths, which corresponds to the order in which a tree is built:

$$\mathsf{epost}(t) \to \mathsf{epost}(t, [\,]).$$

$$\begin{aligned} \mathsf{epost}(\mathsf{ext}(), s) &\to [\mathsf{ext}() \,|\, s]; \\ \mathsf{epost}(\mathsf{int}(x, t_1, t_2), s) &\to \mathsf{epost}(t_1, \mathsf{epost}(t_2, [x \,|\, s])). \end{aligned}$$

Figure 7.46: Postorder encoding

from the external nodes up to the root. Therefore, all we have to do is to identify the growing subtrees by putting the unused numbers and subtrees in an auxiliary stack: when the contents of a root (anything different from $\mathsf{ext}()$) appears in the original stack, we can make an internal node with the two first subtrees in the auxiliary stack.

The definition of $\mathsf{post2b/1}$ (*extended postorder to binary tree*) is given in FIGURE 7.47. Variable f stands for *forest*, which is how stacks of trees are usually called in computing science. Note that a problem would arise if the original tree contains trees because, in that case, an external node contained in an internal node would confuse $\mathsf{post2b/1}$. The cost is easy to assess because a postorder encoding must have length $2n+1$, which is the total number of nodes of a binary with n internal nodes. Therefore, $\mathcal{C}_n^{\mathsf{post2b}} = 2n + 3$. The expected theorem is, of course,

$$\mathsf{post2b}(\mathsf{epost}(t)) \equiv t. \tag{7.8}$$

Considering preorder now, the encoding function $\mathsf{epre/1}$ (*extended preorder*) in FIGURE 7.48 on the facing page is a simple modification of $\mathsf{pre/1}$ in FIGURE 7.10 on page 213. The cost is as simple as for a postorder encoding: $\mathcal{C}_n^{\mathsf{epre}} = 2n + 2$.

Working out the inverse function, from preorder encodings back to binary trees, is a little trickier than for postorder traversals, because the preorder numbers increase downwards in the trees, which is the opposite direction in which trees grow (programmers have trees grow from the leaves to the root). One solution consists in recalling the relationship

$$\mathsf{post2b}(s) \to \mathsf{post2b}([\,], s).$$

$$\begin{aligned} \mathsf{post2b}([t], [\,]) &\to t; \\ \mathsf{post2b}(f, [\mathsf{ext}() \,|\, s]) &\to \mathsf{post2b}([\mathsf{ext}() \,|\, f], s); \\ \mathsf{post2b}([t_2, t_1 \,|\, f], [x \,|\, s]) &\to \mathsf{post2b}([\mathsf{int}(x, t_1, t_2) \,|\, f], s). \end{aligned}$$

Figure 7.47: Postorder decoding

$$\begin{array}{rl} \mathsf{epre}(t) & \to \mathsf{epre}(t, [\,]). \\ \\ \mathsf{epre}(\mathsf{ext}(), s) & \to [\mathsf{ext}() \,|\, s]; \\ \mathsf{epre}(\mathsf{int}(x, t_1, t_2), s) & \to [x \,|\, \mathsf{epre}(t_1, \mathsf{epre}(t_2, s))]. \end{array}$$

Figure 7.48: Preorder encoding

$$\begin{array}{rl} \mathsf{pre2b}(s) & \to \mathsf{pre2b}([\,], \mathsf{rev}(s)). \\ \\ \mathsf{pre2b}([t], [\,]) & \to t; \\ \mathsf{pre2b}(f, [\mathsf{ext}() \,|\, s]) & \to \mathsf{pre2b}([\mathsf{ext}() \,|\, f], s); \\ \mathsf{pre2b}([t_1, t_2 \,|\, f], [x \,|\, s]) & \to \mathsf{pre2b}([\mathsf{int}(x, t_1, t_2) \,|\, f], s). \end{array}$$

Figure 7.49: Preorder decoding

$\mathsf{PreMir}(t)$ between preorder and postorder we proved earlier on page 226:

$$\mathsf{pre}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{post}(t)).$$

We should extend the proof of this theorem so we have

$$\mathsf{epre}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{epost}(t)). \tag{7.9}$$

In section 2.2, we proved $\mathsf{Inv}(s)$ and $\mathsf{EqRev}(s)$, that is to say, the involution of $\mathsf{rev}/1$:

$$\mathsf{rev}(\mathsf{rev}(s)) \equiv t. \tag{7.10}$$

Property (7.10) and (7.9) yield

$$\mathsf{rev}(\mathsf{epre}(\mathsf{mir}(t))) \equiv \mathsf{rev}(\mathsf{rev}(\mathsf{epost}(t))) \equiv \mathsf{epost}(t).$$

Applying (7.8), we obtain

$$\mathsf{post2b}(\mathsf{rev}(\mathsf{epre}(\mathsf{mir}(t)))) \equiv \mathsf{post2b}(\mathsf{epost}(t)) \equiv t.$$

From exercise 3 on page 228, we have $\mathsf{mir}(\mathsf{mir}(t)) \equiv t$, therefore

$$\mathsf{post2b}(\mathsf{rev}(\mathsf{epre}(t))) \equiv \mathsf{mir}(t) \quad \text{hence} \quad \mathsf{mir}(\mathsf{post2b}(\mathsf{rev}(\mathsf{epre}(t)))) \equiv t.$$

Because we want the encoding followed by the decoding to be the identity, $\mathsf{pre2b}(\mathsf{epre}(t)) \equiv t$, we have $\mathsf{pre2b}(\mathsf{epre}(t)) \equiv \mathsf{mir}(\mathsf{post2b}(\mathsf{rev}(\mathsf{epre}(t))))$, that is, setting the stack $s := \mathsf{epre}(t)$,

$$\mathsf{pre2b}(s) \equiv \mathsf{mir}(\mathsf{post2b}(\mathsf{rev}(s))).$$

We obtain $\mathsf{pre2b}/1$ by modifying $\mathsf{post2b}/1$ in FIGURE 7.49. The differ-

ence between pre2b/2 and post2b/2 lies in their last pattern, that is, $\mathsf{post2b}([t_2, t_1 \mid f], [x \mid s])$ versus $\mathsf{pre2b}([t_1, t_2 \mid f], [x \mid s])$, which implements the fusion of mir/1 and post2b/1. Unfortunately, the cost of $\mathsf{pre2b}(t)$ is greater than the cost of $\mathsf{post2b}(t)$ because of the stack reversal $\mathsf{rev}(s)$ at the start:

$$\mathcal{C}_n^{\mathsf{pre2b}} = 2n + 3 + \mathcal{C}_n^{\mathsf{rev}} = 3n + 5.$$

The design of pre2b/1 is based on small steps with an accumulator. A more direct approach would extract the left subtree and then the right subtree from the rest of the encoding. In other words, the new version $\mathsf{pre2b}_1(s)$ would return a tree build from a prefix of the encoding s, paired with the rest of the encoding. The definition is displayed in FIGURE 7.50. Notice the absence of any adventitious concept, contrary to pre2b/1, which relies on the reversal of a stack and a theorem about mirror trees and postorders. To wit, $\mathsf{pre2b}_0/1$ is conceptually simpler, although its cost is greater than that of pre2b/1 because we count the number of function calls after the inference rules are translated into the core functional language (so two more calls matching (t_1, s_1) and (t_2, s_2) are implicit).

Tree encodings show that it is possible to compactly represent binary trees, as long as we do not care for the contents of the internal nodes. For instance, we mentioned that the tree in FIGURE 7.11b on page 214 yields the extended postorder traversal $[\square, \square, \square, 0, 1, \square, \square, 2, 3, \square, \square, \square, 4, 5, 6]$. If we only want to retain the shape of the tree, we could replace the contents of the internal nodes by 0 and the external nodes by 1, yielding the encoding $[1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0]$. A binary tree of size n can be uniquely represented by a binary number of $2n+1$ bits. In fact, we can discard the first bit because the first two bits are always 1, so $2n$ bits are actually enough. For an extended preorder traversal, we choose to map external nodes to 0 and internal nodes to 1, so, the tree in FIGURE 7.11a on page 214 yields $[0, 1, 2, \square, 3, \square, \square, 4, \square, \square, 5, \square, 6, \square, \square]$, which corresponds to the binary number $(111010010010100)_2$. We can also discard the rightmost bit, since the last two bits are always 0.

$$\frac{\mathsf{pre2b}_1(s) \twoheadrightarrow (t, [\,])}{\mathsf{pre2b}_0(s) \twoheadrightarrow t}. \qquad \mathsf{pre2b}_1([\mathsf{ext}() \mid s]) \to (\mathsf{ext}(), s);$$

$$\frac{\mathsf{pre2b}_1(s) \twoheadrightarrow (t_1, s_1) \qquad \mathsf{pre2b}_1(s_1) \twoheadrightarrow (t_2, s_2)}{\mathsf{pre2b}_1([x \mid s]) \twoheadrightarrow (\mathsf{int}(x, t_1, t_2), s_2)}.$$

Figure 7.50: Another preorder decoding

7.4 Random traversals

Some applications require a tree traversal to depend on the interaction with a user or another piece of software, that is, the tree is supplemented with the notion of a current node so the next node to be visited can be chosen amongst any of the children, the parent or even the siblings. This interactivity stands in contrast with preorder, inorder and postorder, where the visit order is predetermined and cannot be changed during the traversal.

Normally, the visit of a functional data structure starts always at the same location, for example, in the case of a stack, it is the top item and, in the case of a tree, the access point is the root. Sometimes, updating a data structure with an online algorithm (see page 9 and section 4.6 on page 165) requires to keep a direct access 'inside' the data structure, usually where the last update was performed, or nearby, in view of a better amortised (see page 9) or average cost (see 2-way insertion in section 3.2 on page 103).

Let us call the current node the *slider*, also called the *focus*. A *zipper* on a binary tree is made of a subtree, whose root is the slider, and a *path* from it up to the root. That path is the reification, in reverse order, of the recursive calls that led to the subtree (the *call stack*), together with the subtrees left unvisited on the way down. Put in more abstract terms, a zipper is made of a linear context (a rooted path) and a substructure (at the end of that path), whose handle is the focus. In one move, it is possible to visit the children in any order, the parent or the sibling. Consider FIGURE 7.51 where the slider is the node d. The substructure is

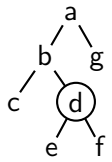


Figure 7.51

$$s := \mathsf{int}(\mathsf{d}(), \mathsf{int}(\mathsf{e}(), \mathsf{ext}(), \mathsf{ext}()), \mathsf{int}(\mathsf{f}(), \mathsf{ext}(), \mathsf{ext}())).$$

To define rooted paths for the zipper, we need three data constructors: one denoting the empty path, $\mathsf{top}()$, one denoting a turn to the left, $\mathsf{left}(x, t, p)$, where x is a node on the path, t is the right subtree of x, and p is the rest of the path up to the root, and one constructor denoting a turn to the right, $\mathsf{right}(x, t, p)$, where t is the left subtree of x.

Resuming our example above, the zipper is then the pair (p, s), with the path p being $\mathsf{right}(\mathsf{b}(), \mathsf{int}(\mathsf{c}(), \mathsf{ext}(), \mathsf{ext}()), p_1)$, meaning that the node b has an unvisited left child c (or, equivalently, we turned right when going down), and where $p_1 := \mathsf{left}(\mathsf{a}(), \mathsf{int}(\mathsf{g}(), \mathsf{ext}(), \mathsf{ext}()), \mathsf{top}())$, meaning that the node a has an unvisited right child g, and that it is the root of the whole tree, due to $\mathsf{top}()$. Note that since b is the first in the path up, it is the parent of the slider d.

$$\begin{aligned}
\mathsf{up}((\mathsf{left}(x, t_2, p), t_1)) &\to (p, \mathsf{int}(x, t_1, t_2));\\
\mathsf{up}((\mathsf{right}(x, t_1, p), t_2)) &\to (p, \mathsf{int}(x, t_1, t_2)).\\
\\
\mathsf{left}((p, \mathsf{int}(x, t_1, t_2))) &\to (\mathsf{left}(x, t_2, p), t_1).\\
\\
\mathsf{right}((p, \mathsf{int}(x, t_1, t_2))) &\to (\mathsf{right}(x, t_1, p), t_2).\\
\\
\mathsf{sibling}((\mathsf{left}(x, t_2, p), t_1)) &\to (\mathsf{right}(x, t_1, p), t_2);\\
\mathsf{sibling}((\mathsf{right}(x, t_1, p), t_2)) &\to (\mathsf{left}(x, t_2, p), t_1).
\end{aligned}$$

Figure 7.52: Basic steps in a binary tree

At the beginning, the original tree t is injected into a zipper $(\mathsf{top}(), t)$. Then, the operations we want for traversing a binary tree on demand are $\mathsf{up/1}$ (go to the parent), $\mathsf{left/1}$ (go to the left child), $\mathsf{right/1}$ (go to the right child) and $\mathsf{sibling/1}$ (go to the sibling). All take a zipper as input and all calls evaluate into a zipper. After any of these steps is performed, a new zipper is assembled as the value of the call. See FIGURE 7.52 for the program. Beyond random traversals of a binary tree, this technique, which is an instance of *Huet's zipper* (Huet, 1997, 2003), also allows local editing. This simply translates as the replacement of the current tree by another:

$$\mathsf{graft}(t', (p, t)) \to (p, t').$$

If we only want to change the slider, we would use

$$\mathsf{slider}(x', (p, \mathsf{int}(x, t_1, t_2))) \to (p, \mathsf{int}(x', t_1, t_2)).$$

If we want to go up to the root and extract the new tree:

$$\mathsf{zip}((\mathsf{top}(), t)) \to t; \quad \mathsf{zip}(z) \to \mathsf{zip}(\mathsf{up}(z)).$$

We do not need a zipper to perform a preorder, inorder or postorder traversal, because it is primarily designed to open down and close up paths from the root of a tree, in the manner of a zipper in a cloth. Nevertheless, if we retain one aspect of its design, namely, the accumulation of unvisited nodes and subtrees, we can define the classic traversals in tail form, that is, by means of a definition where the right-hand sides either are a value or a function call whose arguments are not function calls themselves. Such definitions are equivalent to loops in imperative languages and may be a target for some compilers (Appel, 1992).

We show a preorder traversal following this design in FIGURE 7.53 on the next page, where, in $\mathsf{pre}_8(s, f, t)$, the stack s is expected to collect

$$\begin{aligned}
\mathsf{pre}_7(t) &\to \mathsf{pre}_8([\,],[\,],t).\\
\\
\mathsf{pre}_8(s,[\,],\mathsf{ext}()) &\to s;\\
\mathsf{pre}_8(s,[\mathsf{int}(x,t_1,\mathsf{ext}())\,|\,f],\mathsf{ext}()) &\to \mathsf{pre}_8(s,[x\,|\,f],t_1);\\
\mathsf{pre}_8(s,[x\,|\,f],\mathsf{ext}()) &\to \mathsf{pre}_8([x\,|\,s],f,\mathsf{ext}());\\
\mathsf{pre}_8(s,f,\mathsf{int}(x,t_1,t_2)) &\to \mathsf{pre}_8(s,[\mathsf{int}(x,t_1,\mathsf{ext}())\,|\,f],t_2).
\end{aligned}$$

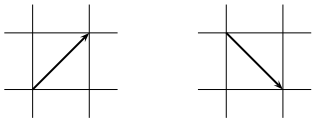
Figure 7.53: Preorder in tail form

the visited nodes in preorder, the stack f (forest) is the accumulator of unvisited parts of the original tree and t is the current subtree to be traversed. The cost is simple: $\mathcal{C}_n^{\mathsf{pre}_7} = 3n + 2$.

7.5 Enumeration

Many publications (Knuth, 1997, § 2.3.4.4) (Sedgewick and Flajolet, 1996, § 5.1) show how to find the number of binary trees of size n using an advanced mathematical tool called *generating functions* (Graham et al., 1994, chap. 7). Instead, for didactical purposes, we opt for a more intuitive technique in enumerative combinatorics which consists in constructing a one-to-one correspondence between two finite sets, so the cardinal of one is the cardinal of the other. In other words, we are going to relate bijectively, on the one hand, binary trees, and, on the other hand, other combinatorial objects which are relatively easy to count, for a given size.

We actually know the appropriate objects in the instance of *Dyck paths*, introduced in section 2.5 about queueing. A Dyck path is a broken line in a grid from the point $(0, 0)$ to $(2n, 0)$, made up of the two kinds of segments shown in FIGURE 7.54, such that it remains above the abscissa axis or reaches it. Consider again the example given in FIGURE 2.16 on page 65, without taking into account the individual costs associated



(a) Rise (b) Fall

Figure 7.54: Basic steps in a grid

to each step. Following the same convention as in chapter 5, we would say here that a *Dyck word* is a finite word over the alphabet made of the letters r (rise) and f (fall), such that all its prefixes contain more letters r than f, or an equal number. This condition is equivalent to the geometrical characterisation 'above the abscissa axis or reaches it.' For instance, rff is not a Dyck word because the prefix rff (actually, the whole word) contains more falls than rises, so the associated path ends below the axis. The Dyck word corresponding to the Dyck path in FIGURE 2.16 on page 65 is rrrfrfrrfffrff. Conceptually, there is no difference between a Dyck path and a Dyck word, we use the former when a geometrical framework is more intuitive and the latter when symbolic reasoning and programming are expected.

First, let us map injectively binary trees to Dyck words, in other words, we want to traverse any tree and produce a Dyck word which is not the mapping of another tree. Since, by definition, non-empty binary trees are made of an internal node connected to two binary subtrees, we may wonder how to split a Dyck word into three parts: one corresponding to the root of the tree and two corresponding to the immediate subtrees.

Since any Dyck word starts with a rise and ends with a fall, we may ask what is the word in-between. In general, it is not a Dyck word; for example, chopping off the ends of rfrrff yields frrf. Instead, we seek a decomposition of Dyck words into Dyck words. If the Dyck word has exactly one *return*, that is, one fall leading to the abscissa axis, then cutting out the first rise and that unique return (which must be the last fall) yields another Dyck word. For instance, $\mathsf{rrfrrfff} = \mathsf{r} \cdot \mathsf{rfrrff} \cdot \mathsf{f}$.

Such words are called *prime*, because any Dyck word can be uniquely decomposed as the concatenation of such words (whence the reference to prime factorisation in elementary number theory): for all non-empty Dyck words d, there exists $n > 0$ unique prime Dyck words p_i such that $d = p_1 \cdot p_2 \cdots p_n$. This naturally yields the *arch decomposition*, whose name stems from an architectural analogy: for all Dyck words d, there exists $n > 0$ Dyck words d_i and returns f_i such that

$$d = (\mathsf{r} \cdot d_1 \cdot \mathsf{f}_1) \cdots (\mathsf{r} \cdot d_n \cdot \mathsf{f}_n).$$

For more details, see Panayotopoulos and Sapounakis (1995), Lothaire (2005), Flajolet and Sedgewick (2009).

Unfortunately, this analysis is not suitable as it stands, because n may be greater than 2, precluding any analogy with binary trees. The solution is simple enough: let us keep the first prime factor $\mathsf{r} \cdot d_1 \cdot \mathsf{f}_1$ and *not* factorise the suffix, which is a Dyck word. To wit, for all non-empty Dyck words d, there exists one return f_1 and two Dyck subwords d_1 and d_2 (possibly

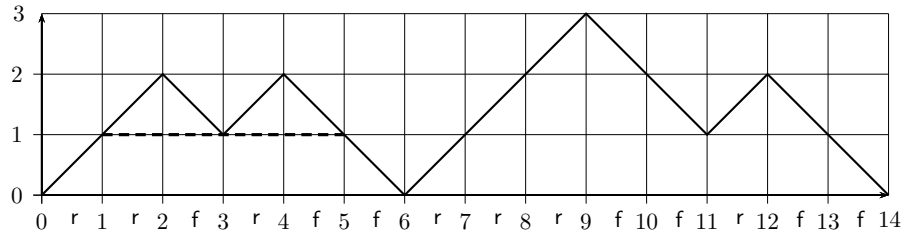
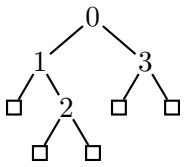
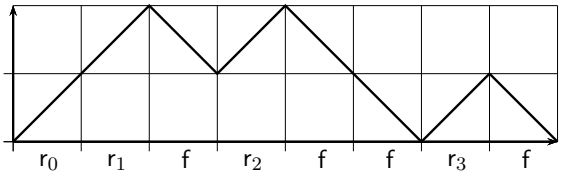


Figure 7.55: Quadratic decomposition of a Dyck path



(a) Binary tree



(b) Dyck path corresponding to preorder encoding $[0, 1, \square, 2, \square, \square, 3, \square, \square]$

Figure 7.56: Bijection between a binary tree and a Dyck path

empty) such that we have

$$d = (\mathsf{r} \cdot d_1 \cdot \mathsf{f}_1) \cdot d_2.$$

This is the *first return decomposition*, also known as *quadratic decomposition* – also possible is $d = d_1 \cdot (\mathsf{r} \cdot d_2 \cdot \mathsf{f}_1)$. For example, the Dyck word rrfrffrrrffrff, shown in FIGURE 7.55 admits the quadratic decomposition $\mathsf{r} \cdot \mathsf{rfrf} \cdot \mathsf{f} \cdot \mathsf{rrrffrff}$. This decomposition is unique because the prime factorisation is unique.

Given a tree $\mathsf{int}(x, t_1, t_2)$, the rise and fall explicitly distinguished in the quadratic decomposition are to be conceived as a pair which is the mapping of x, while d_1 is the mapping of t_1 and d_2 is the mapping of t_2. More precisely, the value of x is not relevant here, only the existence of an internal node, and a fork in a leaf tree would be mapped just as well. Formally, if $\delta(t)$ is the Dyck word mapped from the binary tree t, then we expect the following equations to hold:

$$\delta(\mathsf{ext}()) = \varepsilon; \quad \delta(\mathsf{int}(x, t_1, t_2)) = \mathsf{r}_x \cdot \delta(t_1) \cdot \mathsf{f} \cdot \delta(t_2). \tag{7.11}$$

Note that we attached the node contents x to the rise, so we do not lose information. For example, the tree in FIGURE 7.56a would formally be

$$t := \mathsf{int}(0, \mathsf{int}(1, \mathsf{ext}(), \mathsf{int}(2, \mathsf{ext}(), \mathsf{ext}())), \mathsf{int}(3, \mathsf{ext}(), \mathsf{ext}())),$$

and mapped into the Dyck path in FIGURE 7.56b as follows:

$$\begin{aligned}\delta(t) &= \mathsf{r}_0 \cdot \delta(\mathsf{int}(1, \mathsf{ext}(), \mathsf{int}(2, \mathsf{ext}(), \mathsf{ext}()))) \cdot \mathsf{f} \cdot \delta(\mathsf{int}(3, \mathsf{ext}(), \mathsf{ext}())) \\ &= \mathsf{r}_0 \cdot (\mathsf{r}_1 \cdot \delta(\mathsf{ext}()) \cdot \mathsf{f} \cdot \delta(\mathsf{int}(2, \mathsf{ext}(), \mathsf{ext}()))) \cdot \mathsf{f} \cdot \delta(\mathsf{int}(3, \mathsf{ext}(), \mathsf{ext}())) \\ &= \mathsf{r}_0\mathsf{r}_1\varepsilon \cdot \mathsf{f} \cdot (\mathsf{r}_2 \cdot \delta(\mathsf{ext}()) \cdot \mathsf{f} \cdot \delta(\mathsf{ext}())) \cdot \mathsf{f} \cdot (\mathsf{r}_3 \cdot \delta(\mathsf{ext}()) \cdot \mathsf{f} \cdot \delta(\mathsf{ext}())) \\ &= \mathsf{r}_0\mathsf{r}_1\mathsf{f} \cdot (\mathsf{r}_2 \cdot \varepsilon \cdot \mathsf{f} \cdot \varepsilon) \cdot \mathsf{f} \cdot (\mathsf{r}_3 \cdot \varepsilon \cdot \mathsf{f} \cdot \varepsilon) = \mathsf{r}_0 \cdot \mathsf{r}_1 \cdot \mathsf{f} \cdot \mathsf{r}_2 \cdot \mathsf{f} \cdot \mathsf{f} \cdot \mathsf{r}_3 \cdot \mathsf{f}.\end{aligned}$$

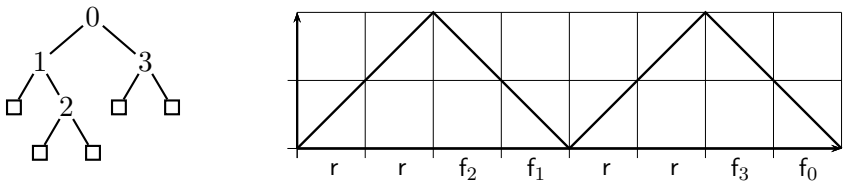
Notice that if we replace the rises by their associated contents (in subscript) and the falls by $\square$, we obtain $[0, 1, \square, 2, \square, \square, 3, \square]$, which is the preorder encoding of the tree without its last $\square$. We could then modify epre/2 in FIGURE 7.48 on page 241 to map a binary tree to a Dyck path, but we would have to remove the last item from the resulting stack, so it is more efficient to directly implement δ as function dpre/1 (*Dyck path as preorder*) in FIGURE 7.57. If the size of the binary tree is n, then $\mathcal{C}_n^{\mathsf{dpre}} = 2n + 2$ and the length of the Dyck path is $2n$.

This mapping is clearly reversible, as we already solved the problem of decoding a tree in preorder in FIGURES 7.49 and 7.50 on page 242, and we understand now that the reversed mapping is based on the quadratic ('first return') decomposition of the path.

If we are concerned about efficiency, though, we may recall that using a postorder encoding yields a more efficient decoding, as we saw in FIGURES 7.46 and 7.47 on page 240, therefore a faster reverse mapping. To create a Dyck path based on a postorder, we map external nodes to rises and internal nodes to falls (with associated contents), and then remove the first rise. See FIGURE 7.58b on the next page for the Dyck path obtained from the postorder traversal of the same previous tree. Of course, just as we did with the preorder mapping, we are not going to make the postorder encoding, but instead go directly from the binary tree to the Dyck path, as shown in FIGURE 7.59 on the facing page. Note that we push r() and f(x) in the same rule, so we do not have to remove the first rise at the end. (We use a similar optimisation with dpre/2 in FIGURE 7.57.) In structural terms, the inverse of this postorder mapping corresponds to a decomposition $d = d_1 \cdot (\mathsf{r} \cdot d_2 \cdot \mathsf{f}_1)$, which we mentioned earlier in passing as an alternative to the 'first return' decomposition.

$$\begin{aligned}\mathsf{dpre}(t) &\to \mathsf{dpre}(t, [\,]).\\ \\ \mathsf{dpre}(\mathsf{ext}(), s) &\to s;\\ \mathsf{dpre}(\mathsf{int}(x, t_1, t_2), s) &\to [\mathsf{r}(x) \,|\, \mathsf{dpre}(t_1, [\mathsf{f}() \,|\, \mathsf{dpre}(t_2, s)])].\end{aligned}$$

Figure 7.57: Mapping in preorder a binary tree to a Dyck path



(a) Binary tree

(b) Dyck path corresponding to postorder encoding $[\square, \square, \square, 2, 1, \square, \square, 3, 0]$

Figure 7.58: Bijection between a binary tree and a Dyck path

$$\mathsf{dpost}(t) \to \mathsf{dpost}(t, [\,]).$$

$$\mathsf{dpost}(\mathsf{ext}(), s) \to s;$$
$$\mathsf{dpost}(\mathsf{int}(x, t_1, t_2), s) \to \mathsf{dpost}(t_1, [\mathsf{r}() \,|\, \mathsf{dpost}(t_2, [\mathsf{f}(x) \,|\, s])]).$$

Figure 7.59: Mapping in postorder a binary tree to a Dyck path

The mapping from Dyck paths encoded in postorder to binary trees is a simple variation on post2b/1 and post2b/2 in FIGURE 7.47 on page 240: simply make the auxiliary stack be [ext()] at the beginning. The definition is named d2b/1 and shown in FIGURE 7.60. The cost is

$$\mathcal{C}_n^{\mathsf{d2b}} = 2n + 2.$$

Whether we choose a preorder or a postorder mapping, as a consequence of the established bijections, we know that there are as many binary trees of size n as Dyck paths of length $2n$. We already know, from chapter 6, and equation (6.1) on page 196, that there are C_n paths:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \sim \frac{4^n}{n\sqrt{\pi n}}.$$

Other encodings of binary trees can be found in (Knuth, 1997, 2.3.3) and (Sedgewick and Flajolet, 1996, 5.11).

$$\mathsf{d2b}(s) \to \mathsf{d2b}([\mathsf{ext}()], s).$$

$$\mathsf{d2b}([t], [\,]) \to t;$$
$$\mathsf{d2b}(f, [\mathsf{r}() \,|\, s]) \to \mathsf{d2b}([\mathsf{ext}() \,|\, f], s);$$
$$\mathsf{d2b}([t_2, t_1 \,|\, f], [\mathsf{f}(x) \,|\, s]) \to \mathsf{d2b}([\mathsf{int}(x, t_1, t_2) \,|\, f], s).$$

Figure 7.60: Mapping a Dyck path to a binary tree in postorder

Average path length Most of the usual average parameters of binary trees, like average internal path length, average height and width, are quite difficult to derive and require mathematical tools which are beyond the scope of this book.

The *internal path length* $I(t)$ of a binary tree t is the sum of the path lengths from the root to every internal node. We already saw the concept of external path length $E(t)$, that is, the sum of the path lengths from the root to every internal node, in section 2.8 on page 82 about optimal sorting, where we showed that the binary tree with minimum average external path length has all its external nodes on two successive levels. The relation between these two path lengths is quite simple because the binary structure yields an equation depending only on the size n:

$$E_n = I_n + 2n. \tag{7.12}$$

Indeed, let $\mathsf{int}(x, t_1, t_2)$ be a tree with n internal nodes. Then we have

$$I(\mathsf{ext}()) = 0, \quad I(\mathsf{int}(x, t_1, t_2)) = I(t_1) + I(t_2) + n - 1, \tag{7.13}$$

because each path in t_1 and t_2 is extended with one more edge back to the root x, and there are $n - 1$ such paths by definition. On the other hand,

$$E(\mathsf{ext}()) = 0, \quad E(\mathsf{int}(x, t_1, t_2)) = E(t_1) + E(t_2) + n + 1, \tag{7.14}$$

because the paths in t_1 and t_2 are extended by one more step to the root x and there are $n + 1$ such paths, from theorem 5 on page 206. Subtracting equation (7.13) from (7.14) yields

$$\begin{aligned} E(\mathsf{ext}()) - I(\mathsf{ext}()) &= 0, \\ E(\mathsf{int}(x, t_1, t_2)) - I(\mathsf{int}(x, t_1, t_2)) &= (E(t_1) - I(t_1)) + (E(t_2) - I(t_2)) + 2. \end{aligned}$$

In other words, each internal node adds 2 to the difference between the external and internal path lengths from it. Since the difference is 0 at the external nodes, we get equation (7.12) for the tree of size n. Unfortunately, almost anything else is quite hard to prove. For instance, the *average internal path length* $\mathbb{E}[I_n]$ has been shown to be

$$\mathbb{E}[I_n] = \frac{4^n}{C_n} - 3n - 1 \sim n\sqrt{\pi n}$$

by Knuth (1997), in exercise 5 of section 2.3.4.5, and Sedgewick and Flajolet (1996), in Theorem 5.3 of section 5.6. Using equation (7.12), we deduce $\mathbb{E}[E_n] = \mathbb{E}[I_n] + 2n$, implying that the cost for traversing a random binary tree of size n from the root to a random external node is $\mathbb{E}[E_n]/(n+1) \sim \sqrt{\pi n}$. Moreover, the value $\mathbb{E}[I_n]/n$ can be understood as the average level of a random internal node.

Average height The average height h_n of a binary tree of size n is even more difficult to obtain and was studied by Flajolet and Odlyzko (1981), Brown and Shubert (1984), Flajolet and Odlyzko (1984), Odlyzko (1984):

$$h_n \sim 2\sqrt{\pi n}.$$

In the case for Catalan trees, to wit, trees whose internal nodes may have any number of children, the analysis of the average height has been carried out by Dasarathy and Yang (1980), Dershowitz and Zaks (1981), Kemp (1984) in section 5.1.1, Dershowitz and Zaks (1990), Knuth et al. (2000) and Sedgewick and Flajolet (1996), in section 5.9.

Average width The *width* of a binary tree is the length of its largest extended level. It can be shown that the *average width* w_n of a binary tree of size n satisfies

$$w_n \sim \sqrt{\pi n} \sim \frac{1}{2} h_n.$$

In particular, this result implies that the average size of the stack needed to perform the preorder traversal with $\mathsf{pre}_4/2$ in FIGURE 7.6 on page 211 is twice the average size of the queue needed to perform a level-order traversal with $\mathsf{bf}/1$ in FIGURE 7.35 on page 232. This is not obvious, as the two stacks used to simulate the queue do not always hold a complete level.

For general trees, a bijective correspondence with binary trees and the transfer of some average parameters has been nicely presented by Dasarathy and Yang (1980).

Exercises

1. Prove $\mathsf{post2b}(\mathsf{epost}(t)) \equiv t$.
2. Prove $\mathsf{pre2b}(\mathsf{epre}(t)) \equiv t$.
3. Prove $\mathsf{epre}(\mathsf{mir}(t)) \equiv \mathsf{rev}(\mathsf{epost}(t))$.
4. Define the encoding of a tree based on its inorder traversal.

Chapter 8

Binary Search Trees

Searching for an internal node in a binary tree can be costly because, in the worst case, the whole tree must be traversed, for example, in preorder or level-order. To improve upon this, two situations are desirable: the binary tree should be as balanced as possible and the choice of visiting the left or right subtree should be taken only upon examining the contents in the root, called *key*.

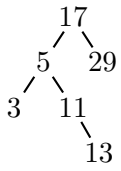


Figure 8.1

The simplest solution consists in satisfying the latter condition and later see how it fits the former. A *binary search tree* (Mahmoud, 1992) $\mathsf{bst}(x, t_1, t_2)$ is a binary tree such that the key x is greater than the keys in t_1 and smaller than the keys in t_2. (The external node $\mathsf{ext}()$ is a trivial search tree.) The comparison function depends on the nature of the keys, but has to be *total*, that is, any key can be compared to any other key. An example is given in FIGURE 8.1. An immediate consequence of the definition is that the inorder traversal of a binary search tree yields an increasingly sorted stack, for example, $[3, 5, 11, 13, 17, 29]$ from the tree in FIGURE 8.1.

This property enables checking simply that a binary tree is a search tree: perform an inorder traversal and then check the order of the resulting stack. The corresponding function, $\mathsf{bst}_0/1$, is legible in FIGURE 8.2 on the next page, where $\mathsf{in}_2/2$ is just a redefinition of $\mathsf{in}/2$ in FIGURE 7.21 on page 221. Thus, the cost of $\mathsf{in}_2(t)$, when t has size n, is $\mathcal{C}_n^{\mathsf{in}_2} = \mathcal{C}_n^{\mathsf{in}} = 2n+2$. The worst case for $\mathsf{ord}/1$ occurs when the stack is sorted increasingly, so the maximum cost is $\mathcal{W}_n^{\mathsf{ord}} = n$, if $n > 0$. The best case is manifest when the first key is greater than the second, so the minimum cost is $\mathcal{B}_n^{\mathsf{ord}} = 1$. Summing up: $\mathcal{B}_n^{\mathsf{bst}_0} = 1 + (2n + 2) + 1 = 2n + 4$ and $\mathcal{W}_n^{\mathsf{bst}_0} = 1 + (2n + 2) + n = 3n + 3$.

A better design consists in not constructing the inorder stack and only *keeping the smallest key so far*, assuming the traversal is from right

$$
\begin{aligned}
\mathsf{bst}_0(t) &\to \mathsf{ord}(\mathsf{in}_2(t, [\,])).\\
\\
\mathsf{in}_2(\mathsf{ext}(), s) &\to s;\\
\mathsf{in}_2(\mathsf{bst}(x, t_1, t_2), s) &\to \mathsf{in}_2(t_1, [x \,|\, \mathsf{in}_2(t_2, s)]).\\
\\
\mathsf{ord}([x, y \,|\, s]) &\to \mathsf{ord}([y \,|\, s]),\ \text{if } y \succ x;\\
\mathsf{ord}([x, y \,|\, s]) &\to \mathsf{false}();\\
\mathsf{ord}(s) &\to \mathsf{true}().
\end{aligned}
$$

Figure 8.2: Naïvely checking a binary search tree

$$
\begin{aligned}
\mathsf{bst}(t) &\to \mathsf{norm}(\mathsf{bst}_1(t, \mathsf{infty}())).\\
\\
\mathsf{bst}_1(\mathsf{ext}(), m) &\to m;\\
\mathsf{bst}_1(\mathsf{bst}(x, t_1, t_2), m) &\to \mathsf{cmp}(x, t_1, \mathsf{bst}_1(t_2, m)).\\
\\
\mathsf{cmp}(x, t_1, \mathsf{infty}()) &\to \mathsf{bst}_1(t_1, x);\\
\mathsf{cmp}(x, t_1, m) &\to \mathsf{bst}_1(t_1, x),\ \text{if } m \succ x;\\
\mathsf{cmp}(x, t_1, m) &\to \mathsf{false}().\\
\\
\mathsf{norm}(\mathsf{false}()) &\to \mathsf{false}();\\
\mathsf{norm}(m) &\to \mathsf{true}().
\end{aligned}
$$

Figure 8.3: Checking a binary search tree

to left, and compare it with the current key. But this is a problem at the beginning, as we have not visited any node yet. A common trick to deal with exceptional values is to use a *sentinel*, which is a dummy. Here, we would like to set the sentinel to $+\infty$, as any key would be smaller, in particular the largest key, which is *unknown*. (Would it be known, we could use it as a sentinel.) It is actually easy to model this infinite value in our functional language: let us simply use a constant data constructor infty/0 (*infinity*) and make sure that we handle its comparison separately from the others. Actually, infty() is compared only once, with the largest key, but we will not try to optimise this, lest the design is obscured.

The program is displayed in FIGURE 8.3. The parameter m stands for the *minimum* key so far. The sole purpose of norm/1 is to get rid of the smallest key m in the tree and instead terminate with true(), but, if the tree is not empty, we could as well end with true(m), or even false(x), if more information were deemed useful.

In the worst case, the original tree is a binary search tree, hence it has to be traversed in its entirety. If there are n internal nodes, the maximum cost is $\mathcal{W}_n^{\mathsf{bst}} = 1 + 2n + (n+1) + 1 = 3n + 3$ because each internal node triggers one call to $\mathsf{bst}_1/2$ and, in turn, one call to $\mathsf{cmp}/3$; also, all the $n+1$ external nodes are visited. Consequently, $\mathcal{W}_n^{\mathsf{bst}_0} = \mathcal{W}_n^{\mathsf{bst}}$, if $n > 0$, which is not an improvement. Nevertheless, here, we do not build a stack with all the keys, which is a definite gain in terms of memory allocation.

Memory is not the only advantage, though, as the minimum cost of $\mathsf{bst}/1$ is lower than for $\mathsf{bst}_0/1$. Indeed, the best case for both occurs when the tree is not a binary search tree, but this is discovered the sooner in $\mathsf{bst}/1$ at the second comparison, because the first one always succeeds by design ($+\infty \succ x$). Obviously, for the second comparison to occur as soon as possible, we need the first comparison to happen as soon as possible too. Two configurations work:

$$\mathsf{bst}(\mathsf{bst}(x, t_1, \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}())))\ \xrightarrow{8}\ \mathsf{false}(),$$
$$\mathsf{bst}(\mathsf{bst}(y, \mathsf{bst}(x, t_1, \mathsf{ext}()), \mathsf{ext}()))\ \xrightarrow{8}\ \mathsf{false}(),$$

where $x \succcurlyeq y$. (The second tree is the left rotation of the first. We have seen on page 221 that inorder traversals are invariant through rotations.) The minimum cost in both cases is $\mathcal{B}_n^{\mathsf{bst}} = 8$, to be contrasted with the linear cost $\mathcal{B}_n^{\mathsf{bst}_0} = 2n+4$ due to the inevitable complete inorder traversal.

8.1 Search

We now must find out whether searching for a key is faster than with an ordinary binary tree, which was our initial motivation. Given the search tree $\mathsf{bst}(x, t_1, t_2)$, if the key y we are searching for is such that $y \succ x$, then we search recursively for it in t_2; otherwise, if $x \succ y$, we look in t_1; finally, if $y = x$, we just found it at the root of the given tree. The definition of $\mathsf{mem}/2$ (*membership*) is shown in FIGURE 8.4. The crucial point is that we may not need to visit all the nodes. More precisely, if all nodes are visited, then the tree is degenerate, to wit, it is isomorphic

$$\begin{aligned}
\mathsf{mem}(y, \mathsf{ext}()) &\to \mathsf{false}();\\
\mathsf{mem}(x, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{true}();\\
\mathsf{mem}(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{mem}(y, t_1),\ \text{if } x \succ y;\\
\mathsf{mem}(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{mem}(y, t_2).
\end{aligned}$$

Figure 8.4: Searching in a binary search tree

to a stack, like the trees in FIGURE 7.4 on page 209 and FIGURE 7.31 on page 231. Clearly, the minimum cost of a successful search occurs when the key is at the root, so $\mathcal{B}^{\mathsf{mem}}_{n(+)} = 1$, and the minimum cost of an unsuccessful search happens when the root has an external node as a child and mem/2 visits it: $\mathcal{B}^{\mathsf{mem}}_{n(-)} = 2$. The maximum cost of a successful search occurs when the tree is degenerate and the key we are looking for is at the only leaf, so $\mathcal{W}^{\mathsf{mem}}_{n(+)} = n$, and the maximum cost of an unsuccessful search happens when visiting one of the children of the leaf of a degenerate tree: $\mathcal{W}^{\mathsf{mem}}_{n(-)} = n + 1$. Therefore,

$$\mathcal{B}^{\mathsf{mem}}_{n} = 1 \quad \text{and} \quad \mathcal{W}^{\mathsf{mem}}_{n} = n + 1.$$

These extremal costs are the same as for a linear search by ls/2:

$$\mathsf{ls}(x, [\,]) \to \mathsf{false}(); \quad \mathsf{ls}(x, [x \,|\, s]) \to \mathsf{true}(); \quad \mathsf{ls}(x, [y \,|\, s]) \to \mathsf{ls}(x, s).$$

The cost of a successful linear search is $\mathcal{C}^{\mathsf{ls}}_{n,k} = k$, if the sought key is at position k, where the first key is at position 1. Therefore, the average cost of a successful linear search, assuming that each distinct key is equally likely to be sought is

$$\mathcal{A}^{\mathsf{ls}}_{n} = \frac{1}{n} \sum_{k=1}^{n} \mathcal{C}^{\mathsf{ls}}_{n,k} = \frac{n+1}{2}.$$

This raises the question of the average cost of mem/2.

Average cost It is clear from the definition that a search path starts at the root and either ends at an internal node in case of success, or at an external node in case of failure; moreover, each node on these paths corresponds to one function call. Therefore, the average cost of mem/2 is directly related to the average internal and external path lengths. To clearly see how, let us consider a binary search tree of size n containing distinct keys. The total cost of searching all of these keys is $n + I_n$, where I_n is the internal path length (we add n to I_n because we count the nodes on the paths, not the edges, as one internal node is associated with one function call). In other words, a random key chosen amongst those in a given tree of size n is found by mem/2 with an average cost of $1 + I_n/n$. Dually, the total cost of reaching all the external nodes of a given binary search tree is $(n + 1) + E_n$, where E_n is the external path length (there are $n+1$ external nodes in a tree with n internal nodes; see theorem 5 on page 206). In other words, the average cost of a failed search by mem/2 is $1 + E_n/(n + 1)$.

At this point, we should realise that we are dealing with a double randomness, or, equivalently, an average of averages. Indeed, the previous discussion assumed the search tree was given, but the key was random. The general case is when both are chosen randomly, that is, when the previous results are averaged over all possible trees of the same size n. Let $\mathcal{A}^{\mathsf{mem}}_{n(+)}$ be the average cost of the successful search of a random key in a random tree of size n (any of the n keys being sought with same probability); moreover, let $\mathcal{A}^{\mathsf{mem}}_{n(-)}$ be the average cost of the unsuccessful search of a random key in a random tree (any of the $n+1$ intervals whose end points are the n keys being equally likely to be searched). Then

$$\mathcal{A}^{\mathsf{mem}}_{n(+)} = 1 + \frac{1}{n}\mathbb{E}[I_n] \quad \text{and} \quad \mathcal{A}^{\mathsf{mem}}_{n(-)} = 1 + \frac{1}{n+1}\mathbb{E}[E_n], \tag{8.1}$$

where $\mathbb{E}[I_n]$ and $\mathbb{E}[E_n]$ are, respectively, the average (or *expected*) internal path length and the average external path length. Reusing equation (7.12), page 250 ($E_n = I_n + 2n$), we deduce $\mathbb{E}[E_n] = \mathbb{E}[I_n] + 2n$ and we can now relate the average costs of searching by eliminating the average path lengths:

$$\mathcal{A}^{\mathsf{mem}}_{n(+)} = \left(1 + \frac{1}{n}\right)\mathcal{A}^{\mathsf{mem}}_{n(-)} - \frac{1}{n} - 2. \tag{8.2}$$

Importantly, this equation holds for all binary search trees, *independently of how they are built*. In the next section, we shall envisage two methods for making search trees and we will be able to determine $\mathcal{A}^{\mathsf{mem}}_{n(+)}$ and $\mathcal{A}^{\mathsf{mem}}_{n(-)}$ with the help of equation (8.2).

But before that, we could perhaps notice that in FIGURE 8.4 on page 255 we did not follow the order of the comparisons as we wrote it down. In the case of a successful search, the comparison $y = x$ holds exactly once, at the very end; therefore, checking it before the others, as we did in the second rule in FIGURE 8.4 on page 255, means that it fails for every key on the search path, except for the last. If we measure the cost as the number of function calls, we would not care, but, if we are interested in minimising the number of comparisons involved in a search, it is best to move that rule *after* the other inequality tests, as in FIGURE 8.5 on the next page. (We assume that an equality is checked as fast as an inequality.) With $\mathsf{mem}_0/2$, the number of comparisons for each search path is different because of the asymmetry between left and right: visiting t_1 yields one comparison ($x \succ y$), whilst t_2 begets two comparisons ($x \not\succ y$ and $y \succ x$). Moreover, we also moved the pattern for the external node after the rules with comparisons, because each search path contains exactly one external node at the end, so it is likely more efficient to check it last. By the way, all textbooks we are aware of suppose that

exactly one atomic comparison with three possible outcomes (*3-way comparison*) occurs, despite the programs they provide clearly employing the *2-way comparisons* ($=$) and ($\succ$). This widespread blind spot renders the theoretical analysis based on the number of comparisons less pertinent, because most high-level programming languages simply do not feature native 3-way comparisons.

Andersson's variant Andersson (1991) proposed a variant for searching which fully acknowledges the use of 2-way comparisons and reduces their number to a minimum, at the expense of more function calls. The design consists in threading a candidate key while descending in the tree and always ending a search at an external node: if the candidate then equals the sought key, the search is successful, otherwise it is not. Therefore, the cost in terms of function calls of an unsuccessful search is the same as with $\mathsf{mem}/2$ or $\mathsf{mem}_0/2$, and the ending external node is the same, but the cost for a successful search is higher. Nevertheless, the advantage is that *equality is not tested on the way down*, only when the external node is reached, so only one comparison per node is required. The program is shown in FIGURE 8.6 on the facing page. The candidate is the third argument to $\mathsf{mem}_2/2$ and its first instance is the root of the tree itself, as seen in the first rule of $\mathsf{mem}_1/2$. The only conceptual difference with $\mathsf{mem}_0/2$ is how a successful search is acknowledged: if, somewhere along the search path, $x = y$, then x becomes the candidate and it will be threaded down to an external node where $x = x$ is checked.

The worst case happens when the tree is degenerate and $\mathsf{mem}_1/2$ performs $n + 1$ 2-way comparisons, which we write as $\overline{\mathcal{W}}_n^{\mathsf{mem}_1} = n + 1$, after the notations we used in the analysis of merge sort, back in chapter 4 on page 117.

In the case of $\mathsf{mem}_0/2$, the recursive call to the right subtree incurs twice as much comparisons as in the left subtree, thus the worst case is a right-leaning degenerate tree, like in FIGURE 7.4b on page 209, and all internal nodes are visited: $\overline{\mathcal{W}}_n^{\mathsf{mem}_0} = 2n$.

In the case of $\mathsf{mem}/2$, the number of comparisons is symmetric be-

$$
\begin{aligned}
\mathsf{mem}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{mem}_0(y, t_1), \text{ if } x \succ y;\\
\mathsf{mem}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{mem}_0(y, t_2), \text{ if } y \succ x;\\
\mathsf{mem}_0(y, \mathsf{ext}()) &\to \mathsf{false}();\\
\mathsf{mem}_0(y, t) &\to \mathsf{true}().
\end{aligned}
$$

Figure 8.5: Searching with fewer 2-way comparisons

$$
\begin{aligned}
\mathsf{mem}_1(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{mem}_2(y, \mathsf{bst}(x, t_1, t_2), x);\\
\mathsf{mem}_1(y, \mathsf{ext}()) &\to \mathsf{false}().\\
\\
\mathsf{mem}_2(y, \mathsf{bst}(x, t_1, t_2), c) &\to \mathsf{mem}_2(y, t_1, c), \text{ if } x \succ y;\\
\mathsf{mem}_2(y, \mathsf{bst}(x, t_1, t_2), c) &\to \mathsf{mem}_2(y, t_2, x);\\
\mathsf{mem}_2(y, \mathsf{ext}(), y) &\to \mathsf{true}();\\
\mathsf{mem}_2(y, \mathsf{ext}(), c) &\to \mathsf{false}().
\end{aligned}
$$

Figure 8.6: Andersson's search (key candidate)

cause equality is tested first, so the worst case is a degenerate tree in which an unsuccessful search leads to the visit of all internal nodes and one external node: $\overline{\mathcal{W}}_n^{\mathsf{mem}} = 2n + 1$. Asymptotically, we have

$$\overline{\mathcal{W}}_n^{\mathsf{mem}} \sim \overline{\mathcal{W}}_n^{\mathsf{mem}_0} \sim 2 \cdot \overline{\mathcal{W}}_n^{\mathsf{mem}_1}.$$

In the case of Andersson's search, there is no difference between the cost, in terms of function calls, of a successful search and an unsuccessful one, so, for $n > 0$, we have

$$\mathcal{A}_n^{\mathsf{mem}_3} = 1 + \mathcal{A}_n^{\mathsf{mem}_2} \quad \text{and} \quad \mathcal{A}_n^{\mathsf{mem}_2} = \mathcal{A}_{n(-)}^{\mathsf{mem}}. \tag{8.3}$$

Choosing between $\mathsf{mem}_0/2$ and $\mathsf{mem}_1/2$ depends on the compiler or interpreter of the programming language chosen for the implementation. If a 2-way comparison is slower than an indirection (following a pointer, or, at the assembly level, jumping unconditionally), it is probably best to opt for Andersson's variant. But the final judgement requires a benchmark.

As a last note, we may simplify Andersson's program by getting rid of the initial emptiness test in $\mathsf{mem}_1/2$. What we need to do is simply have a candidate be the subtree whose root is the candidate in the original program. See FIGURE 8.7 where we have $\overline{\mathcal{W}}_n^{\mathsf{mem}_3} = \overline{\mathcal{W}}_n^{\mathsf{mem}_1} = n + 1$. This

$$
\begin{aligned}
\mathsf{mem}_3(y, t) &\to \mathsf{mem}_4(y, t, t).\\
\\
\mathsf{mem}_4(y, \mathsf{bst}(x, t_1, t_2), t) &\to \mathsf{mem}_4(y, t_1, t), \text{ if } x \succ y;\\
\mathsf{mem}_4(y, \mathsf{bst}(x, t_1, t_2), t) &\to \mathsf{mem}_4(y, t_2, \mathsf{bst}(x, t_1, t_2));\\
\mathsf{mem}_4(y, \mathsf{ext}(), \mathsf{bst}(y, t_1, t_2)) &\to \mathsf{true}();\\
\mathsf{mem}_4(y, \mathsf{ext}(), t) &\to \mathsf{false}().
\end{aligned}
$$

Figure 8.7: Andersson's search (tree candidate)

version may be preferred only if the programming language used for the implementation features *aliases* in patterns or, equivalently, if the compiler can detect that the term $\mathsf{bst}(x, t_1, t_2)$ can be shared instead of being duplicated in the second rule of $\mathsf{mem}_4/3$ (here, we assume that sharing is implicit and maximum within a rule). For additional information on Andersson's variant, read Spuler (1992).

8.2 Insertion

Leaf insertion Since all unsuccessful searches end at an external node, it is extremely tempting to start the insertion of a unique key by a (failing) search and then grow a leaf with the new key at the external node we reached. FIGURE 8.8 displays the program for $\mathsf{insl}/2$ (*insert a leaf*). Note that it allows duplicates in the binary search tree, which hinders the cost analysis (Burge, 1976, Archibald and Clément, 2006, Pasanen, 2010). FIGURE 8.9 shows a variant which maintains the unicity of the keys, based on the definition of $\mathsf{mem}_0/2$ in FIGURE 8.5 on page 258. Alternatively, we can reuse Andersson's lookup, as shown in FIGURE 8.10 on the next page.

$$\begin{aligned}
\mathsf{insl}(y, \mathsf{bst}(x, t_1, t_2)) &\xrightarrow{\tau} \mathsf{bst}(x, \mathsf{insl}(y, t_1), t_2), \text{ if } x \succ y;\\
\mathsf{insl}(y, \mathsf{bst}(x, t_1, t_2)) &\xrightarrow{\upsilon} \mathsf{bst}(x, t_1, \mathsf{insl}(y, t_2));\\
\mathsf{insl}(y, \mathsf{ext}()) &\xrightarrow{\phi} \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}()).
\end{aligned}$$

Figure 8.8: Leaf insertion with possible duplicates

$$\begin{aligned}
\mathsf{insl}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, \mathsf{insl}_0(y, t_1), t_2), \text{ if } x \succ y;\\
\mathsf{insl}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, t_1, \mathsf{insl}_0(y, t_2)), \text{ if } y \succ x;\\
\mathsf{insl}_0(y, \mathsf{ext}()) &\to \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}());\\
\mathsf{insl}_0(y, t) &\to t.
\end{aligned}$$

Figure 8.9: Leaf insertion without duplicates

Average cost In order to carry out the average case analysis of leaf insertion, we must assume that all inserted keys are distinct; equivalently, we consider all the search trees resulting from the insertion into originally empty trees of all the keys of each permutation of $(1, 2, \ldots, n)$. Because

$$\begin{aligned}
\mathsf{insl}_1(y,t) &\to \mathsf{insl}_2(y,t,t).\\
\\
\mathsf{insl}_2(y,\mathsf{bst}(x,t_1,t_2),t) &\to \mathsf{bst}(x,\mathsf{insl}_2(y,t_1,t),t_2),\ \text{if } x \succ y;\\
\mathsf{insl}_2(y,\mathsf{bst}(x,t_1,t_2),t) &\to \mathsf{bst}(x,t_1,\mathsf{insl}_2(y,t_2,\mathsf{bst}(x,t_1,t_2)));\\
\mathsf{insl}_2(y,\mathsf{ext}(),\mathsf{bst}(y,t_1,t_2)) &\to \mathsf{ext}();\\
\mathsf{insl}_2(y,\mathsf{ext}(),t) &\to \mathsf{bst}(y,\mathsf{ext}(),\mathsf{ext}()).
\end{aligned}$$

Figure 8.10: Andersson's insertion

the number of permutations is greater than the number of binary trees of same size, to wit, $n! > C_n$ if $n > 2$ (see equation (6.1) on page 196), we expect some tree shapes to correspond to many permutations. As we will see in the section about the average height, degenerate and wildly unbalanced trees are rare in average (Fill, 1996), making binary search trees a good random data structure as long as only leaf insertions are performed. Because we assume the unicity of the inserted keys, we shall only consider $\mathsf{insl}/2$ in the following. (Andersson's insertion is only worth using if duplicate keys are possible inputs that must be detected, leaving the search tree invariant.)

Let us define a function $\mathsf{mkl}/1$ (*make leaves*) in FIGURE 8.11 which builds a binary search tree by inserting as leaves all the keys in a given stack. Note that we could also define a function $\mathsf{mklR}/1$ (*make leaves in reverse order*) such that $\mathsf{mklR}(s) \equiv \mathsf{mkl}(\mathsf{rev}(s))$ in a compact manner:

$$\mathsf{mklR}([\,]) \to \mathsf{ext}(); \quad \mathsf{mklR}([x \,|\, s]) \to \mathsf{insl}(x, \mathsf{mklR}(s)). \tag{8.4}$$

The cost of $\mathsf{insl}(x,t)$ depends on x and the shape of t, but, because all shapes are obtained by $\mathsf{mkl}(s) \twoheadrightarrow t$ for a given length of s, and all external nodes of t are equally likely to grow a leaf containing x, the average cost $\mathcal{A}_k^{\mathsf{insl}}$ of $\mathsf{insl}(x,t)$ only depends on the size k of the trees:

$$\mathcal{A}_n^{\mathsf{mkl}} = 2 + \sum_{k=0}^{n-1} \mathcal{A}_k^{\mathsf{insl}}. \tag{8.5}$$

One salient feature of leaf insertion is that internal nodes do not move,

$$\begin{aligned}
&\mathsf{mkl}(s) \xrightarrow{\xi} \mathsf{mkl}(s,\mathsf{ext}()). \qquad \mathsf{mkl}([\,],t) \xrightarrow{\psi} t;\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{mkl}([x \,|\, s],t) \xrightarrow{\omega} \mathsf{mkl}(s,\mathsf{insl}(x,t)).
\end{aligned}$$

Figure 8.11: Making a binary search tree with leaf insertions

hence the internal path length of the nodes is invariant and the cost of searching all keys in a tree of size n is the cost of inserting them in the first place. We already noticed that the former cost is, in average, $n + \mathbb{E}[I_n]$; the latter cost is $\sum_{k=0}^{n-1} \mathcal{A}_k^{\mathsf{insl}}$. From equation (8.5) then comes

$$n + \mathbb{E}[I_n] = \mathcal{A}_n^{\mathsf{mkl}} - 2 \tag{8.6}$$

(The subtraction of 2 is to account for rules ξ and ψ, which perform no insertion.) The cost of a leaf insertion is that of an unsuccessful search:

$$\mathcal{A}_k^{\mathsf{insl}} = \mathcal{A}_{k(-)}^{\mathsf{mem}}. \tag{8.7}$$

Recalling equation (8.1) on page 257, equations (8.5), (8.6) and (8.7):

$$\mathcal{A}_{n(+)}^{\mathsf{mem}} = 1 + \frac{1}{n}\mathbb{E}[I_n] = \frac{1}{n}(\mathcal{A}_n^{\mathsf{mkl}} - 2) = \frac{1}{n}\sum_{k=0}^{n-1} \mathcal{A}_k^{\mathsf{insl}} = \frac{1}{n}\sum_{k=0}^{n-1} \mathcal{A}_{k(-)}^{\mathsf{mem}}.$$

Finally, using equation (8.2) on page 257, we deduce

$$\frac{1}{n}\sum_{k=0}^{n-1} \mathcal{A}_{k(-)}^{\mathsf{mem}} = \left(1 + \frac{1}{n}\right) \mathcal{A}_{n(-)}^{\mathsf{mem}} - \frac{1}{n} - 2.$$

Equivalently,

$$2n + 1 + \sum_{k=0}^{n-1} \mathcal{A}_{k(-)}^{\mathsf{mem}} = (n+1)\mathcal{A}_{n(-)}^{\mathsf{mem}}.$$

This recurrence is easy to solve if we subtract its instance when $n - 1$:

$$2 + \mathcal{A}_{n-1(-)}^{\mathsf{mem}} = (n+1)\mathcal{A}_{n(-)}^{\mathsf{mem}} - n\mathcal{A}_{n-1(-)}^{\mathsf{mem}}.$$

Noting that $\mathcal{A}_{0(-)}^{\mathsf{mem}} = 1$, the equation becomes

$$\mathcal{A}_{0(-)}^{\mathsf{mem}} = 1, \quad \mathcal{A}_{n(-)}^{\mathsf{mem}} = \mathcal{A}_{n-1(-)}^{\mathsf{mem}} + \frac{2}{n+1},$$

thus

$$\mathcal{A}_{n(-)}^{\mathsf{mem}} = 1 + 2\sum_{k=2}^{n+1} \frac{1}{k} = 2H_{n+1} - 1, \tag{8.8}$$

where $H_n := \sum_{k=1}^{n} 1/k$ is the nth harmonic number. Replacing $\mathcal{A}_{n(-)}^{\mathsf{mem}}$ back into equation (8.2) and using $H_{n+1} = H_n + 1/(n+1)$ yields

$$\mathcal{A}_{n(+)}^{\mathsf{mem}} = 2\left(1 + \frac{1}{n}\right) H_n - 3. \tag{8.9}$$

From inequations (3.11) on page 115 and equations (8.8) and (8.9):

$$\mathcal{A}_n^{\mathsf{insl}} \sim \mathcal{A}_{n(-)}^{\mathsf{mem}} \sim \mathcal{A}_{n(+)}^{\mathsf{mem}} \sim 2\ln n.$$

We obtain more information about the relative asymptotic behaviours of $\mathcal{A}_{n(-)}^{\mathsf{mem}}$ and $\mathcal{A}_{n(+)}^{\mathsf{mem}}$ by looking at their difference instead of their ratio:

$$\mathcal{A}_{n(-)}^{\mathsf{mem}} - \mathcal{A}_{n(+)}^{\mathsf{mem}} = \frac{2}{n}(n + 1 - H_{n+1}) \sim 2 \quad \text{and} \quad 1 \leqslant \mathcal{A}_{n(-)}^{\mathsf{mem}} - \mathcal{A}_{n(+)}^{\mathsf{mem}} < 2.$$

The average difference between an unsuccessful search and a successful one tends slowly to 2 for large values of n, which may not be intuitive. We can use this result to compare the average difference of the costs of a successful search with $\mathsf{mem}/2$ and $\mathsf{mem}_3/2$ (Andersson). Recalling equation (8.3) on page 259, we draw $1 + \mathcal{A}^{\mathsf{mem}}_{n(-)} = \mathcal{A}^{\mathsf{mem}_3}_{n(+)}$. The previous result now yields

$$\mathcal{A}^{\mathsf{mem}_3}_{n(+)} - \mathcal{A}^{\mathsf{mem}}_{n(+)} \sim 3.$$

Therefore, the extra cost of Andersson's variant in case of a successful search is asymptotically 3, in average.

Furthermore, replacing $\mathcal{A}^{\mathsf{mem}}_{n(-)}$ and $\mathcal{A}^{\mathsf{mem}}_{n(+)}$ into equations (8.1) leads to

$$\mathbb{E}[I_n] = 2(n+1)H_n - 4n \quad \text{and} \quad \mathbb{E}[E_n] = 2(n+1)H_n - 2n. \tag{8.10}$$

Thus $\mathbb{E}[I_n] \sim \mathbb{E}[E_n] \sim 2n \ln n$. Note how easier it is to find $\mathbb{E}[I_n]$ for binary search trees, compared to simple binary trees.

If we are interested in slightly more theoretical results, we may like to know the average number of comparisons involved in a search and an insertion. A glance back at FIGURE 8.4 on page 255 uncovers that two 2-way comparisons are done when going down and one 2-way comparison (equality) is checked when finding the key, otherwise none:

$$\overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} = 1 + \frac{2}{n}\mathbb{E}[I_n] \quad \text{and} \quad \overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} = \frac{2}{n+1}\mathbb{E}[E_n]. \tag{8.11}$$

Reusing equations (8.10), we conclude that

$$\overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} = 4\left(1 + \frac{1}{n}\right)H_n - 7 \quad \text{and} \quad \overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} = 4H_n + \frac{4}{n+1} - 4. \tag{8.12}$$

Clearly, we have $\overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} \sim \overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} \sim 4 \ln n$. Furthermore,

$$\overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} - \overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} = \frac{4}{n+1} - \frac{4}{n}H_n + 3 \sim 3 \quad \text{and} \quad 1 \leqslant \overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} - \overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} < 3.$$

The average costs for Andersson's search and insertions are easy to deduce as well, from equation (8.3) on page 259 and (8.8) on page 262: $\mathcal{A}^{\mathsf{mem}_3}_n = 2H_{n+1} \sim 2 \ln n$. A glimpse back at FIGURE 8.7 on page 259 brings to the fore that one 2-way comparison ($x \succ y$) is performed when descending in the tree and one more when stopping at an external node, whether the search is successful or not:

$$\overline{\mathcal{A}}^{\mathsf{mem}_3}_n = \frac{1}{n+1}\mathbb{E}[E_n] = 2H_n + \frac{2}{n+1} - 2 \sim 2 \ln n.$$

We can now finally compare the average number of comparisons between $\mathsf{mem}/2$ and $\mathsf{mem}_3/2$ (Andersson):

$$\overline{\mathcal{A}}^{\mathsf{mem}}_{n(+)} - \overline{\mathcal{A}}^{\mathsf{mem}_3}_{n} = 2\left(1 + \frac{2}{n}\right) H_n - \frac{2}{n+1} - 5 \sim 2\ln n,$$
$$\overline{\mathcal{A}}^{\mathsf{mem}}_{n(-)} - \overline{\mathcal{A}}^{\mathsf{mem}_3}_{n} = 2H_n + \frac{2}{n+1} - 2 \sim 2\ln n.$$

As far as leaf insertion itself is concerned, $\mathsf{insl}/2$ behaves as $\mathsf{mem}_3/2$, except that no comparison occurs at the external nodes. Also, from equations (8.7) and (8.8), we finish the average case analysis of $\mathsf{insl}/2$:

$$\overline{\mathcal{A}}^{\mathsf{insl}}_{n} = \overline{\mathcal{A}}^{\mathsf{mem}_3}_{n} - 1 = 2H_n + \frac{2}{n+1} - 3 \quad \text{and} \quad \mathcal{A}^{\mathsf{insl}}_{n} = 2H_{n+1} - 1.$$

Finally, from equation (8.6) and (8.10), we deduce

$$\mathcal{A}^{\mathsf{mkl}}_{n} = n + \mathbb{E}[I_n] + 2 = 2(n+1)H_n - n + 2 \sim 2n\ln n. \tag{8.13}$$

Amortised cost The worst case for leaf insertion occurs when the search tree is degenerate and the key to be inserted becomes the deepest leaf. If the tree has size n, then $n+1$ calls are performed, as seen in FIGURE 8.8 on page 260, so $\mathcal{W}^{\mathsf{insl}}_{n} = n+1$ and $\overline{\mathcal{W}}^{\mathsf{insl}}_{n} = n$. In the case of Andersson's insertion in FIGURE 8.10 on page 261, the worst case is identical but there is a supplementary call to set the candidate key, so $\mathcal{W}^{\mathsf{insl}_1}_{n} = n+2$. Moreover, the number of comparisons is symmetric and equals 1 per internal node, so $\overline{\mathcal{W}}^{\mathsf{insl}_1}_{n} = n$ and any degenerate tree is the worst configuration.

The best case for leaf insertion with $\mathsf{insl}/2$ and $\mathsf{insl}_1/2$ happens when the key has to be inserted as the left or right child of the root, to wit, the root is the minimum or maximum key in inorder, so $\mathcal{B}^{\mathsf{insl}}_{n} = 2$ and $\mathcal{B}^{\mathsf{insl}_1}_{n} = 3$. As far as comparisons are concerned: $\overline{\mathcal{B}}^{\mathsf{insl}}_{n} = 1$ and $\overline{\mathcal{B}}^{\mathsf{insl}_1}_{n} = 2$.

While turning our attention to the extremal costs of $\mathsf{mkl}/1$ and $\mathsf{mkr}/1$, we need to realise that we cannot simply sum minimum or maximum costs of $\mathsf{insl}/2$ because, as mentioned earlier, the call $\mathsf{insl}(x,t)$ depends on x and the shape of t. For instance, after three keys have been inserted into an empty tree, the root has no more empty children, so the best case we determined previously is not pertinent anymore.

Let $\overline{\mathcal{B}}^{\mathsf{mkl}}_{n}$ be the minimum number of comparisons needed to construct a binary search tree of size n using leaf insertions. If we want to minimise the cost at each insertion, then the path length for each new node must be as small as possible and this is achieved if the tree continuously grows as a perfect or almost perfect tree. The former is a tree whose external nodes

all belong to the same level, a configuration we have seen on page 237 (the tree fits tightly inside an isosceles triangle); the latter is a tree whose external nodes lie on two consecutive levels and we have seen this kind of tree in the paragraph devoted to comparison trees and the minimean of sorting on page 91.

Let us assume first that the tree is perfect, with size n and height h. The height is the length, counted in number of edges, of the longest path from the root to an external node. The total path length for a level k made only of internal nodes is $k2^k$. Therefore, summing all levels yields

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = \sum_{k=1}^{h-1} k2^k = (h-2)2^h + 2, \tag{8.14}$$

by reusing equation (4.52) on page 145. Moreover, summing the number of internal nodes by levels: $n = \sum_{k=0}^{h-1} 2^k = 2^h - 1$, hence $h = \lg(n+1)$, which we can replace in equation (8.14) to obtain

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = (n+1)\lg(n+1) - 2n.$$

We proved $1 + \lfloor \lg n \rfloor = \lceil \lg(n+1) \rceil$ when establishing the maximum number of comparisons of top-down merge sort in equation (4.37) on page 137, so we can proceed conclusively:

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = (n+1)\lfloor \lg n \rfloor - n + 1. \tag{8.15}$$

Let us assume now that the tree is almost perfect, with the penultimate level $h-1$ containing $q \neq 0$ internal nodes, so

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = \sum_{k=1}^{h-2} k2^k + (h-1)q = (h-3)2^{h-1} + 2 + (h-1)q. \tag{8.16}$$

Moreover, the total number n of internal nodes, when summed level by level, satisfies $n = \sum_{k=0}^{h-2} 2^k + q = 2^{h-1} - 1 + q$, hence $q = n - 2^{h-1} + 1$. By definition, we have $0 < q \leqslant 2^{h-1}$, hence $0 < n - 2^{h-1} + 1 \leqslant 2^{h-1}$, which yields $h - 1 < \lg(n+1) \leqslant h$, then $h = \lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$, whence $q = n - 2^{\lfloor \lg n \rfloor} + 1$. We can now substitute h and q by their newly found values in terms of n back into equation (8.16):

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor} + 2. \tag{8.17}$$

Comparing equations (8.15) and (8.17), we see that the number of comparisons is minimised when the tree is perfect, so $n = 2^p - 1$. The asymptotic approximation of $\overline{\mathcal{B}}_n^{\mathsf{mkl}}$ is not difficult to find, as long as we avoid the

pitfall $2^{\lfloor \lg n \rfloor} \sim n$. Indeed, consider the function $x(p) := 2^p - 1$ ranging over the positive integers. First, let us notice that, for all $p > 0$,

$$2^{p-1} \leqslant 2^p - 1 < 2^p \Rightarrow p - 1 \leqslant \lg(2^p - 1) < p \Rightarrow \lfloor \lg(2^p - 1) \rfloor = p - 1.$$

Therefore, $2^{\lfloor \lg(x(p)) \rfloor} = 2^{p-1} = (x(p) + 1)/2 \sim x(p)/2 \nsim x(p)$, which proves that $2^{\lfloor \lg(n) \rfloor} \nsim n$ when $n = 2^p - 1 \to \infty$. Instead, in the case of equation (8.15), let us use the standard inequalities $x - 1 < \lfloor x \rfloor \leqslant x$:

$$(n+1)\lg n - 2n < \overline{\mathcal{B}}_n^{\mathsf{mkl}} \leqslant (n+1)\lg n - n + 1.$$

In the case of equation (8.17), let us use the definition of the fractional part $\{x\} := x - \lfloor x \rfloor$. Obviously, $0 \leqslant \{x\} < 1$. Then

$$\overline{\mathcal{B}}_n^{\mathsf{mkl}} = (n+1)\lg n - n \cdot \theta(\{\lg n\}) + 2 - \{\lg n\},$$

where $\theta(x) := 1 + 2^{-x}$. Let us minimise and maximise the linear term: we have $\min_{0 \leqslant x < 1} \theta(x) = \theta(1) = 3/2$ and $\max_{0 \leqslant x < 1} \theta(x) = \theta(0) = 2$. Keeping in mind that $x = \{\lg n\}$, we have

$$(n+1)\lg n - 2n + 2 < \overline{\mathcal{B}}_n^{\mathsf{mkl}} < (n+1)\lg n - \frac{3}{2}n + 1.$$

In any case, it is now clearly established that $\overline{\mathcal{B}}_n^{\mathsf{mkl}} \sim n \lg n$.

Let $\overline{\mathcal{W}}_n^{\mathsf{mkl}}$ be the maximum number of comparisons to build a binary search tree of size n by leaf insertions. If we maximise each insertion, we need to grow a degenerate tree and insert at one external node of maximal path length:

$$\overline{\mathcal{W}}_n^{\mathsf{mkl}} = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \sim \frac{1}{2}n^2.$$

Root insertion If recently inserted keys are looked up, the cost is relatively high because these keys are leaves or close to a leaf. In this scenario, instead of inserting a key as a leaf, it is better to insert it as a root (Stephenson, 1980). The idea is to perform a leaf insertion and, on the way back to the root (that is to say, after the recursive calls are evaluated, one after the other), we perform rotations to bring the inserted node up to the root. More precisely, if the node was inserted in a left subtree, then a right rotation brings it one level up, otherwise a left rotation has the same effect. The composition of these rotations brings the leaf to the root. Right rotation, `rotr/1` (*rotate right*) and left rotation, `rotl/1` (*rotate left*), were discussed in section 7.1 on page 221 and are defined in FIGURE 8.12 on the facing page. Obviously, they commute

and are inverses of each other:

$$\mathsf{rotl}(\mathsf{rotr}(t)) \equiv \mathsf{rotr}(\mathsf{rotl}(t)) \equiv t.$$

Moreover, and less trivially, they preserve inorder traversals:

$$\mathsf{in}_3(\mathsf{rotl}(t)) \equiv \mathsf{in}_3(\mathsf{rotr}(t)) \equiv \mathsf{in}_3(t),$$

where $\mathsf{in}_3/1$ computes the inorder traversal of a tree: $\mathsf{in}_3(t) \to \mathsf{in}_2(t, [\,])$, with $\mathsf{in}_2/2$ being defined in FIGURE 8.2 on page 254. This theorem is inherently connected to $\mathsf{Rot}(x, y, t_1, t_2, t_3)$, on page 221, and it is easy to prove, without recourse to induction. First, we could remark that if $\mathsf{in}_3(t) \equiv \mathsf{in}_3(\mathsf{rotl}(t))$, then, replacing t by $\mathsf{rotr}(t)$ yields the equivalence

$$\mathsf{in}_3(\mathsf{rotr}(t)) \equiv \mathsf{in}_3(\mathsf{rotl}(\mathsf{rotr}(t))) \equiv \mathsf{in}_3(t),$$

so we only need to prove $\mathsf{in}_3(\mathsf{rotl}(t)) \equiv \mathsf{in}_3(t)$. Since the left-hand side is larger, we should try to rewrite it into the right-hand side. Because a left rotation requires the tree to have the shape $t = \mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3))$, we have the rewrites of FIGURE 8.13 on the next page. If we rotate subtrees, as we did, for example, in FIGURE 7.23 on page 222, the same theorem implies that the inorder traversal of the whole tree is invariant.

A corollary is that rotations keep invariant the property of being a binary search tree (FIGURE 8.3 on page 254):

$$\mathsf{bst}(\mathsf{rotl}(t)) \equiv \mathsf{bst}(\mathsf{rotr}(t)) \equiv \mathsf{bst}(t).$$

Indeed, assuming that $\mathsf{bst}/1$ is the specification of $\mathsf{bst}_0/1$ in FIGURE 8.2 on page 254, and that the latter is correct, that is, $\mathsf{bst}(t) \equiv \mathsf{bst}_0(t)$, it is quite easy to prove our theorem, with the help of the previous theorem $\mathsf{in}_3(\mathsf{rotl}(t)) \equiv \mathsf{in}_3(t)$, which is equivalent to $\mathsf{in}_2(\mathsf{rotl}(t), [\,]) \equiv \mathsf{in}_2(t, [\,])$, and noticing that it is sufficient to prove $\mathsf{bst}_0(\mathsf{rotl}(t)) \equiv \mathsf{bst}_0(t)$. We conclude:

$$\mathsf{bst}_0(\mathsf{rotl}(t)) \equiv \mathsf{ord}(\mathsf{in}_2(\mathsf{rotl}(t), [\,])) \equiv \mathsf{ord}(\mathsf{in}_2(t, [\,])) \leftarrow \mathsf{bst}_0(t).$$

Let us consider now an example of root insertion in FIGURE 8.14 on the next page, where the tree of FIGURE 8.1 on page 253 is augmented with 7. Remark that the transitive closure $(\twoheadrightarrow)$ captures the preliminary leaf insertion, $(\xrightarrow{\epsilon})$ is a right rotation and $(\xrightarrow{\zeta})$ is a left rotation. It is now a simple matter to modify the definition of $\mathsf{insl}/2$ so it becomes

$$\mathsf{rotr}(\mathsf{bst}(y, \mathsf{bst}(x, t_1, t_2), t_3)) \xrightarrow{\epsilon} \mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3)).$$
$$\mathsf{rotl}(\mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3))) \xrightarrow{\zeta} \mathsf{bst}(y, \mathsf{bst}(x, t_1, t_2), t_3).$$

Figure 8.12: Right (ϵ) and left (ζ) rotations

$$
\begin{aligned}
\underline{\mathsf{in}}_3(\mathsf{rotl}(t)) &\to \mathsf{in}_2(\mathsf{rotl}(t), [\,]) \\
&= \mathsf{in}_2(\underline{\mathsf{rotl}}(\mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3))), [\,]) \\
&\overset{\epsilon}{\to} \underline{\mathsf{in}}_2(\mathsf{bst}(y, \mathsf{bst}(x, t_1, t_2), t_3), [\,]) \\
&\to \mathsf{in}_2(\mathsf{bst}(x, t_1, t_2), [y \,|\, \mathsf{in}_2(t_3, [\,])]) \\
&\equiv \mathsf{in}_2(t_1, [x \,|\, \mathsf{in}_2(t_2, [y \,|\, \mathsf{in}_2(t_3, [\,])])]) \\
&\leftarrow \mathsf{in}_2(t_1, [x \,|\, \underline{\mathsf{in}}_2(\mathsf{bst}(y, t_2, t_3), [\,])]) \\
&\leftarrow \underline{\mathsf{in}}_2(\mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3)), [\,]) \\
&= \mathsf{in}_2(t, [\,]) \\
&\leftarrow \mathsf{in}_3(t). \qquad \square
\end{aligned}
$$

Figure 8.13: Proof of $\mathsf{in}_3(\mathsf{rotl}(t)) \equiv \mathsf{in}_3(t)$

root insertion as insr/2, in FIGURE 8.15 on the next page. Note that we can avoid creating the temporary internal nodes $\mathsf{bst}(x, \ldots, t_2)$ and $\mathsf{bst}(x, t_1, \ldots)$ by modifying rotl/1 and rotr/1 so that they take three arguments ($\mathsf{rotl}_0/3$ and $\mathsf{rotr}_0/3$), as shown along the new version $\mathsf{insr}_0/2$ in FIGURE 8.16.

Comparing leaf and root insertions A comparison between leaf and root insertions reveals interesting facts. For instance, because leaf insertion does not displace any node, making the same tree from two permutations of the keys bears the same cost, for example, $(1, 3, 2, 4)$ and $(1, 3, 4, 2)$. On the other hand, as noted by Geldenhuys and der Merwe (2009), making the same search tree using different root insertions may yield different costs, like $(1, 2, 4, 3)$ and $(1, 4, 2, 3)$. They also prove that all the trees of a given size can either be created by leaf or root insertions because we have

$$\mathsf{RootLeaf}(s)\colon \mathsf{mkr}(s) \equiv \mathsf{mkl}(\mathsf{rev}(s)), \tag{8.18}$$

where mkr/1 (*make roots*) is easily defined in FIGURE 8.17 on page 270. Notice that this is equivalent to claim that $\mathsf{mkr}(s) \equiv \mathsf{mklR}(s)$, where

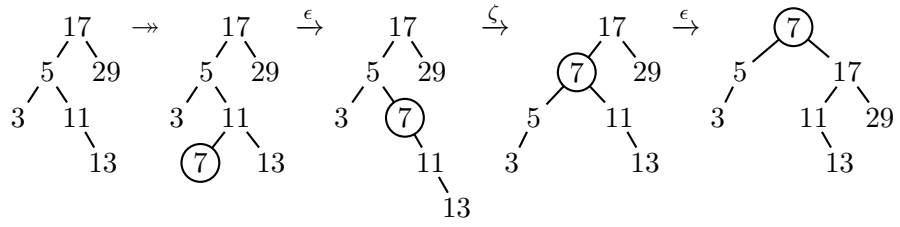


Figure 8.14: Root insertion of 7 into FIGURE 8.1 on page 253

$$\begin{array}{l}\mathsf{insr}(y, \mathsf{bst}(x, t_1, t_2)) \xrightarrow{\eta} \mathsf{rotr}(\mathsf{bst}(x, \mathsf{insr}(y, t_1), t_2)), \text{ if } x \succ y;\\ \mathsf{insr}(y, \mathsf{bst}(x, t_1, t_2)) \xrightarrow{\theta} \mathsf{rotl}(\mathsf{bst}(x, t_1, \mathsf{insr}(y, t_2)));\\ \mathsf{insr}(y, \mathsf{ext}()) \xrightarrow{\iota} \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}()).\end{array}$$

Figure 8.15: Root insertion with possible duplicates

mklR/1 is defined in equation (8.4) on page 261. It is worth proving RootLeaf(s) here because, contrary to Geldenhuys and der Merwe (2009), we want to use structural induction to exactly follow the syntax of the function definitions, instead of induction on sizes, an adventitious concept, and we want to avoid using ellipses when describing the data. Furthermore, our logical framework is not separated from our actual function definitions (the abstract program): the rewrites themselves, that is, the computational steps, give birth to a logical interpretation as classes of equivalent terms.

We start by remarking that RootLeaf(s) is equivalent to

$$\mathsf{RootLeaf}_0(s)\colon \mathsf{mkr}(s) \equiv \mathsf{mkl}(\mathsf{rev}_0(s)),$$

where $\mathsf{rev}_0/1$ is defined at the start of section 2.2 on page 38, where we prove EqRev(s): $\mathsf{rev}_0(s) \equiv \mathsf{rev}(s)$. It is often a good idea to use $\mathsf{rev}_0/1$ in inductive proofs because of rule δ defining $\mathsf{rev}_0([x \,|\, s])$ directly in terms of $\mathsf{rev}_0(s)$. Let us recall the relevant definitions:

$$\begin{array}{rlrl}\mathsf{cat}([\,], t) &\xrightarrow{\alpha} t; & \mathsf{rev}_0([\,]) &\xrightarrow{\gamma} [\,];\\ \mathsf{cat}([x \,|\, s], t) &\xrightarrow{\beta} [x \,|\, \mathsf{cat}(s, t)]. & \mathsf{rev}_0([x \,|\, s]) &\xrightarrow{\delta} \mathsf{cat}(\mathsf{rev}_0(s), [x]).\end{array}$$

Of course, $\mathsf{rev}_0/1$ is worthless as a program because of its quadratic cost, which cannot compete with the linear cost of rev/1, but, as far as theorem proving is concerned, it is a valuable specification and lemma EqRev(s) allows us to transfer any equivalence depending upon $\mathsf{rev}_0/1$ into an equivalence employing rev/1.

$$\begin{array}{l}\mathsf{insr}_0(y, \mathsf{bst}(x, t_1, t_2)) \to \mathsf{rotr}_0(x, \mathsf{insr}_0(y, t_1), t_2), \text{ if } x \succ y;\\ \mathsf{insr}_0(y, \mathsf{bst}(x, t_1, t_2)) \to \mathsf{rotl}_0(x, t_1, \mathsf{insr}_0(y, t_2));\\ \mathsf{insr}_0(y, \mathsf{ext}()) \to \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}()).\\ \\ \mathsf{rotr}_0(y, \mathsf{bst}(x, t_1, t_2), t_3) \to \mathsf{bst}(x, t_1, \mathsf{bst}(y, t_2, t_3)).\\ \mathsf{rotl}_0(x, t_1, \mathsf{bst}(y, t_2, t_3)) \to \mathsf{bst}(y, \mathsf{bst}(x, t_1, t_2), t_3).\end{array}$$

Figure 8.16: Root insertion with possible duplicates (bis)

$$\mathsf{mkr}(s) \xrightarrow{\kappa} \mathsf{mkr}(s, \mathsf{ext}()). \qquad \begin{array}{l} \mathsf{mkr}([\,], t) \xrightarrow{\lambda} t; \\ \mathsf{mkr}([x \,|\, s], t) \xrightarrow{\mu} \mathsf{mkr}(s, \mathsf{insr}(x, t)). \end{array}$$

Figure 8.17: Making a binary search tree with root insertions

Let us proceed by induction on the structure of the stack s. First, we need to prove directly (without induction) $\mathsf{RootLeaf}_0([\,])$. We have

$$\mathsf{mkr}([\,]) \xrightarrow{\kappa} \mathsf{mkr}([\,], \mathsf{ext}()) \xrightarrow{\lambda} \mathsf{ext}() \xleftarrow{\psi} \mathsf{mkl}([\,], \mathsf{ext}()) \xleftarrow{\xi} \mathsf{mkl}([\,]) \xleftarrow{\gamma} \mathsf{mkl}(\underline{\mathsf{rev}}_0([\,])).$$

Second, we set the inductive hypothesis to be $\mathsf{RootLeaf}_0(s)$ and we proceed to prove $\mathsf{RootLeaf}_0([x \,|\, s])$, for any x. Since the right-hand side is larger, we start rewriting it and whenever we feel astray, we rewrite the other side, aiming at their convergence. On the way, there will be steps, in the form of equivalences, which constitute lemmas (subgoals) that will need demonstration later.

$$\begin{array}{rll} \mathsf{mkl}(\underline{\mathsf{rev}}_0([x \,|\, s])) & \xrightarrow{\delta} & \mathsf{mkl}(\mathsf{cat}(\mathsf{rev}_0(s), [x])) \\ & \equiv & \mathsf{mkl}(\mathsf{cat}(\mathsf{rev}_0(s), [x]), \mathsf{ext}()) \\ & \equiv_0 & \mathsf{mkl}([x], \mathsf{mkl}(\mathsf{rev}_0(s), \mathsf{ext}())) \qquad \text{(Lemma)} \\ & \equiv & \mathsf{mkl}([\,], \mathsf{insl}(x, \mathsf{mkl}(\mathsf{rev}_0(s), \mathsf{ext}()))) \\ & \equiv & \mathsf{insl}(x, \mathsf{mkl}(\mathsf{rev}_0(s), \mathsf{ext}())) \\ & \equiv & \mathsf{insl}(x, \mathsf{mkl}(\mathsf{rev}_0(s))) \\ & \equiv & \mathsf{insl}(x, \underline{\mathsf{mkr}}(s)) \qquad (\mathsf{RootLeaf}_0(s)) \\ & \xrightarrow{\xi} & \mathsf{insl}(x, \mathsf{mkr}(s, \mathsf{ext}())) \\ & \equiv_1 & \mathsf{mkr}(s, \underline{\mathsf{insl}}(x, \mathsf{ext}())) \qquad \text{(Lemma)} \\ & \xrightarrow{\phi} & \mathsf{mkr}(s, \mathsf{bst}(x, \mathsf{ext}(), \mathsf{ext}())) \\ & \xleftarrow{\iota} & \mathsf{mkr}(s, \underline{\mathsf{insr}}(x, \mathsf{ext}())) \\ & \xleftarrow{\mu} & \underline{\mathsf{mkr}}([x \,|\, s], \mathsf{ext}()) \\ & \xleftarrow{\kappa} & \mathsf{mkr}([x \,|\, s]). \qquad \square \end{array}$$

Now, we have to prove the two lemmas that we identified with our proof sketch. The first one, in the instance of $(\equiv_0)$, looks like a corollary of $\mathsf{MklCat}(u, v, t)$: $\mathsf{mkl}(\mathsf{cat}(u, v), t) \equiv_0 \mathsf{mkl}(v, \mathsf{mkl}(u, t))$. The first action to be undertaken when facing a new proposition is to try to disprove it by some pertinent or tricky choice of variables. In this case, though, the truth of this lemma can be intuitively ascertained without effort, which gives us more confidence for working out a formal proof, instead of dispensing with one. It is enough to reason by induction on the structure of the stack u. First, we verify $\mathsf{MklCat}([\,], v, t)$:

$$\mathsf{mkl}(\underline{\mathsf{cat}}([\,], v), t) \xrightarrow{\alpha} \mathsf{mkl}(v, t) \xleftarrow{\psi} \underline{\mathsf{mkl}}(v, \mathsf{mkl}([\,], t)).$$

Second, we assume $\mathsf{MklCat}(u, v, t)$, for all v and t, which is thus the inductive hypothesis, and we prove $\mathsf{MklCat}([x \,|\, u], v, t)$:

$$\begin{array}{lll} \mathsf{mkl}(\underline{\mathsf{cat}}([x \,|\, u], v), t) & \xrightarrow{\beta} & \mathsf{mkl}([x \,|\, \mathsf{cat}(u, v)], t) \\ & \equiv & \mathsf{mkl}(\mathsf{cat}(u, v), \mathsf{insl}(x, t)) \\ & \equiv_0 & \mathsf{mkl}(v, \mathsf{mkl}(u, \mathsf{insl}(x, t))) \quad (\mathsf{MklCat}(u, v, \mathsf{inst}(x, t))) \\ & \xleftarrow{\omega} & \mathsf{mkl}(v, \underline{\mathsf{mkl}}([x \,|\, u], t)). \qquad \square \end{array}$$

Let us formally define the second lemma whose instance we identified as $(\equiv_1)$ in the proof of $\mathsf{RootLeaf}_0(s)$. Let

$$\mathsf{MkrInsr}(x, s, t) \colon \mathsf{insl}(x, \mathsf{mkr}(s, t)) \equiv_1 \mathsf{mkr}(s, \mathsf{insl}(x, t)).$$

This proposition, despite its pleasurable symbolic symmetry, is not trivial and may require some examples to be better grasped. It means that a leaf insertion can be performed before or after a series of root insertions, yielding in both cases the same tree. We approach the proof by induction on the structure of the stack s only. (The other parameters are unlikely to be inductively relevant because x is a key, so we can assume nothing about its internal structure, if any, and t is the second parameter of both $\mathsf{mkr}/2$ and $\mathsf{insl}/2$, so we do not know anything about its shape nor contents.) We start, as usual, with a verification (A verification, by definition, does not involve the use of any inductive argument.) of the basis $\mathsf{MkInsr}(x, [\,], t)$:

$$\mathsf{insl}(x, \underline{\mathsf{mkr}}([\,], t)) \xrightarrow{\lambda} \mathsf{insl}(x, t) \equiv \mathsf{mkr}([\,], \mathsf{insl}(x, t)).$$

We now assume $\mathsf{MkrInsr}(x, s, t)$ for all x and t, and we try to prove $\mathsf{MkrInsr}(x, [y \,|\, s], t)$, for all keys y, by rewriting both sides of the equivalence and aiming at the same term:

$$\begin{array}{lll} \mathsf{insl}(x, \underline{\mathsf{mkr}}([y \,|\, s], t)) & \xrightarrow{\mu} & \mathsf{insl}(x, \mathsf{mkr}(s, \mathsf{insr}(y, t))) \\ & \equiv_1 & \mathsf{mkr}(s, \mathsf{insl}(x, \mathsf{insr}(y, t))) \quad (\mathsf{MkrInsr}(x, s, \mathsf{insr}(y, t))) \\ & \equiv_2 & \mathsf{mkr}(s, \mathsf{insr}(y, \mathsf{insl}(x, t))) \qquad (\text{Lemma}) \\ & \equiv & \mathsf{mkr}([y \,|\, s], \mathsf{insl}(x, t)). \qquad \square \end{array}$$

Note that we have found that we need a lemma in the guise of its instance $(\equiv_2)$, which states that a root insertion commutes with a leaf insertion. This is not obvious and probably needs to be seen on some examples to be believed. The process of inductive demonstration itself has brought us to the important concept on which our initial proposition hinges. Let the lemma in question be formally defined as follows:

$$\mathsf{Ins}(x, y, t) \colon \mathsf{insl}(x, \mathsf{insr}(y, t)) \equiv_2 \mathsf{insr}(y, \mathsf{insl}(x, t)).$$

We will use induction on the structure of the tree t, because the other variables are keys, hence are atomic. The verification of $\mathsf{Ins}(x, y, \mathsf{ext}())$, the basis, happens to be rather lengthy, compared to earlier related proofs:

$$\mathsf{insl}(x, \underline{\mathsf{insr}}(y, \mathsf{ext}())) \xrightarrow{\iota} \mathsf{insl}(x, \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}())) \quad \otimes$$

The symbol $\otimes$ is a tag from which different rewrites are possible, depending on some condition, and we will need to resume from that mark. Here, two cases present themselves to us: either $x \succ y$ or $y \succ x$. We have

- If $x \succ y$, then

$$\begin{aligned}
\otimes &\xrightarrow{\upsilon} \mathsf{bst}(y, \mathsf{ext}(), \underline{\mathsf{insl}}(x, \mathsf{ext}())) && (x \succ y)\\
&\xrightarrow{\phi} \mathsf{bst}(y, \mathsf{ext}(), \mathsf{bst}(x, \mathsf{ext}(), \mathsf{ext}()))\\
&\xleftarrow{\epsilon} \underline{\mathsf{rotr}}(\mathsf{bst}(x, \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}()), \mathsf{ext}()))\\
&\xleftarrow{\iota} \mathsf{rotr}(\mathsf{bst}(x, \underline{\mathsf{insr}}(y, \mathsf{ext}()), \mathsf{ext}()))\\
&\xleftarrow{\eta} \underline{\mathsf{insr}}(y, \mathsf{bst}(x, \mathsf{ext}(), \mathsf{ext}())) && (x \succ y)\\
&\xleftarrow{\phi} \mathsf{insr}(y, \underline{\mathsf{insl}}(x, \mathsf{ext}())).
\end{aligned}$$

- If $y \succ x$, then

$$\begin{aligned}
\otimes &\xrightarrow{\tau} \mathsf{bst}(y, \underline{\mathsf{insl}}(x, \mathsf{ext}()), \mathsf{ext}()) && (y \succ x)\\
&\xrightarrow{\phi} \mathsf{bst}(y, \mathsf{bst}(x, \mathsf{ext}(), \mathsf{ext}()), \mathsf{ext}())\\
&\xleftarrow{\xi} \underline{\mathsf{rotl}}(\mathsf{bst}(x, \mathsf{ext}(), \mathsf{bst}(y, \mathsf{ext}(), \mathsf{ext}())))\\
&\xleftarrow{\iota} \mathsf{rotl}(\mathsf{bst}(x, \mathsf{ext}(), \underline{\mathsf{insr}}(y, \mathsf{ext}())))\\
&\xleftarrow{\theta} \underline{\mathsf{insr}}(y, \mathsf{bst}(x, \mathsf{ext}(), \mathsf{ext}())) && (y \succ x)\\
&\xleftarrow{\phi} \mathsf{insr}(y, \underline{\mathsf{insl}}(x, \mathsf{ext}())).
\end{aligned}$$

Now, let us assume $\mathsf{Ins}(x, y, t_1)$ and $\mathsf{Ins}(x, y, t_2)$ and proceed to prove $\mathsf{Ins}(x, y, t)$, with $t = \mathsf{bst}(a, t_1, t_2)$, for all keys a. We start arbitrarily with the right-hand side as follows:

$$\mathsf{insr}(y, \mathsf{insl}(x, t)) = \mathsf{insr}(y, \mathsf{insl}(x, \mathsf{bst}(a, t_1, t_2))) \quad \otimes$$

Two cases arise: either $a \succ x$ or $x \succ a$.

- If $a \succ x$, then $\otimes \xrightarrow{\tau} \mathsf{insr}(y, \mathsf{bst}(a, \mathsf{insl}(x, t_1), t_2))\ \otimes$. Two subcases reveal themselves: either $a \succ y$ or $y \succ a$.
  - If $a \succ y$, then

$$\begin{aligned}
\otimes &\equiv \mathsf{rotr}(\mathsf{bst}(a, \mathsf{insr}(y, \mathsf{insl}(x, t_1)), t_2)) && (a \succ y)\\
&\equiv_2 \mathsf{rotr}(\mathsf{bst}(a, \mathsf{insl}(x, \mathsf{insr}(y, t_1)), t_2)) && (\mathsf{Ins}(x, y, t_1))\\
&\equiv \mathsf{rotr}(\mathsf{insl}(x, \mathsf{bst}(a, \mathsf{insr}(y, t_1), t_2)))\\
&\equiv \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\mathsf{rotr}(\mathsf{bst}(a, \mathsf{insr}(y, t_1), t_2)))))\\
&\xleftarrow{\eta} \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\underline{\mathsf{insr}}(y, \mathsf{bst}(a, t_1, t_2)))))\\
&= \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\mathsf{insr}(y, t)))) && (t = \mathsf{bst}(a, t_1, t_2))\\
&\equiv_3 \mathsf{rotr}(\mathsf{rotl}(\mathsf{insl}(x, \mathsf{insr}(y, t)))) && (\text{Lemma})\\
&\equiv \mathsf{insl}(x, \mathsf{insr}(y, t)). && (\mathsf{rotr}(\mathsf{rotl}(z)) \equiv z)
\end{aligned}$$

What makes this case of the proof work is that $a \succ x$ and $a \succ y$ allow us to move the calls to the rotations down into the term so that they are composed on the subtree t_1, enabling the application of the inductive hypothesis $\mathsf{Ins}(x, y, t_1)$. Then we bring back up the commuted calls, using the fact that composing a left and right rotation, and vice-versa, is the identity. Note how, in the process, we found a new lemma we need to prove later in the instance of $(\equiv_3)$. The interpretation of this subgoal is that left rotation and leaf insertion commute, shedding more light on the matter.

– If $y \succ a$, then

$$
\begin{aligned}
\otimes &\xrightarrow{\theta} \mathsf{rotl}(\mathsf{bst}(a, \mathsf{insl}(x, t_1), \mathsf{insr}(y, t_2))) && (y \succ a)\\
&\equiv \mathsf{rotl}(\mathsf{insl}(x, \mathsf{bst}(a, t_1, \mathsf{insr}(y, t_2))))\\
&\equiv \mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(\mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{insr}(y, t_2))))))\\
&\xleftarrow{\theta} \mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(\underline{\mathsf{insr}}(y, \mathsf{bst}(a, t_1, t_2)))))\\
&= \mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(\mathsf{insr}(y, t)))) && (t = \mathsf{bst}(a, t_1, t_2))\\
&\equiv_4 \mathsf{rotl}(\mathsf{rotr}(\mathsf{insl}(x, \mathsf{insr}(y, t)))) && (\text{Lemma})\\
&\equiv \mathsf{insl}(x, \mathsf{insr}(y, t)). && (\mathsf{rotl}(\mathsf{rotr}(z)) \equiv z)
\end{aligned}
$$

Here, there was no need for the inductive hypothesis, because $a \succ x$ and $y \succ a$ imply $y \succ x$, hence the leaf and root insertions are not composed and apply to two different subtrees, t_1 and t_2. All we have to do then is to get them up in the same order we got them down (as in a queue). We discovered another subgoal that needs proving later, in the instance of $(\equiv_4)$, and which is the dual of $(\equiv_3)$ because it states that right rotation and leaf insertion commute. Together, they mean that rotations commute with leaf insertion.

- If $x \succ a$, then $\otimes \xrightarrow{\upsilon} \mathsf{insr}(y, \mathsf{bst}(a, t_1, \mathsf{insl}(y, t_2)))\ \otimes$. Two subcases become apparent: either $a \succ y$ or $y \succ a$.

  – If $a \succ y$, then

$$
\begin{aligned}
\otimes &\equiv \mathsf{rotr}(\mathsf{bst}(a, \mathsf{bst}(a, \mathsf{insr}(y, t_1), \mathsf{insl}(x, t_2)))) && (a \succ y)\\
&\equiv \mathsf{rotr}(\mathsf{insl}(x, \mathsf{bst}(a, \mathsf{insr}(y, t_1), t_2)))\\
&\equiv \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\mathsf{rotr}(\mathsf{bst}(a, \mathsf{insr}(y, t_1), t_2)))))\\
&\xleftarrow{\eta} \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\underline{\mathsf{insr}}(y, \mathsf{bst}(a, t_1, t_2)))))\\
&= \mathsf{rotr}(\mathsf{insl}(x, \mathsf{rotl}(\mathsf{insr}(y, t)))) && (t = \mathsf{bst}(a, t_1, t_2))\\
&\equiv_3 \mathsf{rotr}(\mathsf{rotl}(\mathsf{insl}(x, \mathsf{insr}(y, t))))\\
&\equiv \mathsf{insl}(x, \mathsf{insr}(y, t)). && (\mathsf{rotr}(\mathsf{rotl}(z)) \equiv z)
\end{aligned}
$$

This subcase is similar to the previous one in the sense that the insertions apply to different subtrees, thus there is no need

for the inductive hypothesis. The difference is that, here, $(\equiv_3)$ is required in stead of $(\equiv_4)$.

– If $y \succ a$, then

$$\begin{aligned}
\otimes \equiv\ & \mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{insr}(y, \mathsf{insl}(x, t_2)))) \\
\equiv_2\ & \mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{insl}(x, \mathsf{insr}(y, t_2)))) && (\mathsf{Ins}(x, y, t_2)) \\
\equiv\ & \mathsf{rotl}(\mathsf{insl}(x, \mathsf{bst}(a, t_1, \mathsf{insr}(y, t_2)))) \\
\equiv_3\ & \mathsf{insl}(x, \mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{insr}(y, t_2)))) \\
\overset{\theta}{\leftarrow}\ & \mathsf{insl}(x, \underline{\mathsf{insr}}(y, \mathsf{bst}(a, t_1, t_2))) \\
=\ & \mathsf{insl}(x, \mathsf{insr}(y, t)). && (t = \mathsf{bst}(a, t_1, t_2))
\end{aligned}$$

This is the last subcase. It is similar to the first one, because the insertions are composed, albeit on t_2 instead of t_1, therefore calling for the inductive hypothesis to be applied. Then, insertions are brought up in the same order they were moved down, *e.g.,* insl/2 was pushed down before insr/2 and is lifted up before insr/2. □

We now have to prove two remaining lemmas, dual of each other and meaning together that rotations commute with leaf insertions. Let us consider the first:

$$\mathsf{insl}(x, \mathsf{rotl}(t)) \equiv_3 \mathsf{rotl}(\mathsf{insl}(x, t)).$$

Implicitly, this proposition makes sense only if $t = \mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, t_3))$ is a binary search tree, which implies $b \succ a$. The proof is technical in nature, which means that it requires many cases and does not bring new insights, which the lack of induction underlies. We start as follows:

$$\begin{aligned}
\mathsf{insl}(x, \mathsf{rotl}(t)) =\ & \mathsf{insl}(x, \underline{\mathsf{rotl}}(\mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, t_3)))) \\
\overset{\zeta}{\rightarrow}\ & \mathsf{insl}(x, \mathsf{bst}(b, \mathsf{bst}(a, t_1, t_2), t_3)) \quad \otimes
\end{aligned}$$

Two cases arise: either $b \succ x$ or $x \succ b$.

- If $b \succ x$, then $\otimes \overset{\tau}{\rightarrow} \mathsf{bst}(b, \mathsf{insl}(x, \mathsf{bst}(a, t_1, t_2)), t_3)\ \otimes$. Two subcases surface: either $a \succ x$ or $x \succ a$.
  – If $a \succ x$, then
  $$\begin{aligned}
  \otimes \overset{\tau}{\rightarrow}\ & \mathsf{bst}(b, \mathsf{bst}(a, \mathsf{insl}(x, t_1), t_2), t_3) \\
  \equiv\ & \mathsf{rotl}(\mathsf{bst}(a, \mathsf{insl}(x, t_1), \mathsf{bst}(b, t_2, t_3))) \\
  \overset{\tau}{\leftarrow}\ & \mathsf{rotl}(\underline{\mathsf{insl}}(x, \mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, t_3)))) \\
  =\ & \mathsf{rotl}(\mathsf{insl}(x, t)).
  \end{aligned}$$
  – If $x \succ a$, then
  $$\begin{aligned}
  \otimes \overset{\upsilon}{\rightarrow}\ & \mathsf{bst}(b, \mathsf{bst}(a, t_1, \mathsf{insl}(x, t_2)), t_3) \\
  \equiv\ & \mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{bst}(b, \mathsf{insl}(x, t_2), t_3))) \\
  \overset{\tau}{\leftarrow}\ & \mathsf{rotl}(\mathsf{bst}(a, t_1, \underline{\mathsf{insl}}(x, \mathsf{bst}(b, t_2, t_3)))) \\
  \overset{\upsilon}{\leftarrow}\ & \mathsf{rotl}(\underline{\mathsf{insl}}(x, \mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, t_3)))) \\
  =\ & \mathsf{rotl}(\mathsf{insl}(x, t)).
  \end{aligned}$$

- If $x \succ b$, then the assumption $b \succ a$ implies $x \succ a$. We have

$$\begin{aligned}
\otimes &\xrightarrow{\upsilon} \mathsf{bst}(b, \mathsf{bst}(a, t_1, t_2), \mathsf{insl}(x, t_3)) \\
&\equiv \mathsf{rotl}(\mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, \mathsf{insl}(x, t_3)))) \\
&\xleftarrow{\upsilon} \mathsf{rotl}(\mathsf{bst}(a, t_1, \underline{\mathsf{insl}}(x, \mathsf{bst}(b, t_2, t_3)))) \qquad (x \succ b) \\
&\xleftarrow{\upsilon} \mathsf{rotl}(\underline{\mathsf{insl}}(x, \mathsf{bst}(a, t_1, \mathsf{bst}(b, t_2, t_3)))) \qquad (x \succ a) \\
&= \mathsf{rotl}(\mathsf{insl}(x, t)). \qquad \square
\end{aligned}$$

The last remaining lemma is $\mathsf{insl}(x, \mathsf{rotr}(t)) \equiv_4 \mathsf{rotr}(\mathsf{insl}(x, t))$. In fact, it is a simple algebraic matter to show that it is equivalent to $\mathsf{insl}(x, \mathsf{rotl}(t)) \equiv_3 \mathsf{rotl}(\mathsf{insl}(x, t))$. Indeed, we have the following equivalent equations:

$$\begin{aligned}
\mathsf{insl}(x, \mathsf{rotl}(t)) &\equiv_3 \mathsf{rotl}(\mathsf{insl}(x, t)) \\
\mathsf{insl}(x, \mathsf{rotl}(\mathsf{rotr}(t))) &\equiv \mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(t))) \\
\mathsf{insl}(x, t) &\equiv \mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(t))) \\
\mathsf{rotr}(\mathsf{insl}(x, t)) &\equiv \mathsf{rotr}(\mathsf{rotl}(\mathsf{insl}(x, \mathsf{rotr}(t)))) \\
\mathsf{rotr}(\mathsf{insl}(x, t)) &\equiv_4 \mathsf{insl}(x, \mathsf{rotr}(t)). \qquad \square
\end{aligned}$$

Average cost The average number of comparisons of root insertion is the same as with leaf insertion, because rotations do not involve any comparison:

$$\overline{\mathcal{A}}_n^{\mathsf{insr}} = \overline{\mathcal{A}}_n^{\mathsf{insr}_0} = \overline{\mathcal{A}}_n^{\mathsf{insl}} = 2H_n + \frac{2}{n+1} - 3 \sim 2 \ln n.$$

Rotations double the cost of a step down in the tree, though, and we have, recalling equations (8.11) and (8.12) on page 263,

$$\mathcal{A}_n^{\mathsf{insr}} = 1 + \frac{2}{n+1}\mathbb{E}[E_n] = 1 + \overline{\mathcal{A}}_{n(-)}^{\mathsf{mem}} = 4H_n + \frac{4}{n+1} - 3 \sim 4 \ln n.$$

As a consequence of theorem (8.18) on page 268, all permutations of a given size yield the same set of binary search trees under $\mathsf{mkl/1}$ and $\mathsf{mkr/1}$. Therefore, inserting another key will incur the same average number of comparisons by $\mathsf{insl/1}$ and $\mathsf{insr/1}$ since $\overline{\mathcal{A}}_n^{\mathsf{insr}} = \overline{\mathcal{A}}_n^{\mathsf{insr}_0} = \overline{\mathcal{A}}_n^{\mathsf{insl}}$. By induction on the size, we conclude that the average number of comparisons for $\mathsf{mkl/1}$ and $\mathsf{mkr/1}$ is the same:

$$\overline{\mathcal{A}}_n^{\mathsf{mkr}} = \overline{\mathcal{A}}_n^{\mathsf{mkl}} = \mathbb{E}[I_n] = 2(n+1)H_n - 4n.$$

Considering that the only difference between $\mathsf{insl/1}$ and $\mathsf{insr/1}$ is the additional cost of one rotation per edge down, we quickly realise, by recalling equations (8.13) and (8.10), that

$$\mathcal{A}_n^{\mathsf{mkr}} = \mathcal{A}_n^{\mathsf{mkl}} + \mathbb{E}[I_n] = n + 2 \cdot \mathbb{E}[I_n] + 2 = 4(n+1)H_n - 7n + 2.$$

Amortised cost Since the first phase of root insertion is a leaf insertion, the previous analyses of the extremal costs of insl/2 and $\mathsf{insl}_1/2$ apply as well to insr/2. Let us consider now the amortised costs of insr/2, namely, the extremal costs of mkr/1.

Let $\overline{\mathcal{B}}_n^{\mathsf{mkr}}$ the minimum number of comparisons to build a binary search tree of size n using root insertions. We saw that the best case with leaf insertion (insl/2) happens when the key is inserted as a child of the root. While this cannot lead to the best amortised cost (mkl/1), it yields the best amortised cost when using root insertions (mkr/1) because the newly inserted key becomes the root with exactly one rotation (a left rotation if it was the right child, and a right rotation if it was the left child of the root), leaving the spot empty again for another efficient insertion (insr/2). In the end, the search tree is degenerate, in fact, there are exactly two minimum-cost trees, whose shapes are those of FIGURE 7.4 on page 209. Interestingly, these trees correspond to maximum-cost trees built using leaf insertions. The first key is not compared, so we have $\overline{\mathcal{B}}_n^{\mathsf{mkr}} = n - 1 \sim \overline{\mathcal{B}}_n^{\mathsf{mkl}} / \lg n$.

Perhaps surprisingly, it turns out that finding the maximum number of comparisons $\overline{\mathcal{W}}_n^{\mathsf{mkr}}$ to make a search tree of size n with mkr/1, that is to say, the maximum amortised number of comparisons of insr/2, happens to be substantially more challenging than making out its average or minimum cost. Geldenhuys and der Merwe (2009) show that

$$\overline{\mathcal{W}}_n^{\mathsf{mkr}} = \frac{1}{4}n^2 + n - 2 - c,$$

where $c = 0$ for n even, and $c = 1/4$ for n odd. This implies in turn:

$$\overline{\mathcal{W}}_n^{\mathsf{mkr}} = \frac{1}{2}\overline{\mathcal{W}}_n^{\mathsf{mkl}} + \frac{5}{4}n - 2 - c \sim \frac{1}{2}\overline{\mathcal{W}}_n^{\mathsf{mkl}}.$$

Exercises

1. Prove $\mathsf{bst}_0(t) \equiv \mathsf{bst}(t)$. See definitions of $\mathsf{bst}_0/1$ and bst/1, respectively, in FIGURE 8.2 on page 254 and FIGURE 8.3 on page 254.
2. Prove $\mathsf{mem}(y, t) \equiv \mathsf{mem}_3(y, t)$, that is to say, the correctness of Andersson's search. See definitions of mem/2 and $\mathsf{mem}_3/2$, respectively, in FIGURE 8.4 on page 255 and FIGURE 8.7 on page 259.
3. Prove $\mathsf{insr}(x, t) \equiv \mathsf{bst}(x, t_1, t_2)$. In other words, root insertion is really doing what it says it does.
4. Prove $\mathsf{mklR}(s) \equiv \mathsf{mkl}(\mathsf{rev}(s))$.
5. Prove $\mathsf{bst}(t) \equiv \mathsf{true}() \Rightarrow \mathsf{mkl}(\mathsf{pre}(t)) \equiv t$. See definition of pre/1 in FIGURE 7.10 on page 213. Is the converse true as well?

8.3 Deletion

The removal of a key in a binary search tree is a bit tricky, in contrast with leaf insertion. Of course, 'removal' is a convenient figure of speech in the context of functional programming, where data structures are persistent, hence removal means that we have to rebuild a new search tree without the key in question. As with insertion, we could simply start with a search for the key: if absent, there is nothing else to be done, otherwise we replace the key with its immediate successor or predecessor in inorder, that is, the minimum of the right subtree or the maximum of the left subtree.

The definitions for these two phases are found in FIGURE 8.18. We have $\mathsf{min}(t_2) \twoheadrightarrow (m, t_2')$, where m is the minimum key of the tree t_2 and t_2' is the reconstruction of t_2 without m; in other words, the leftmost internal node of t_2 contains the key m and that node has been replaced by an external node. The call to $\mathsf{aux}_0/3$ simply substitutes the key x to be deleted by its immediate successor m. The purpose of the auxiliary function $\mathsf{aux}_1/3$ is to rebuild the tree in which the minimum has been removed. Note that the pattern of the third rule is not $\mathsf{del}(y, \mathsf{bst}(y, t_1, t_2))$, because we already know that $x = y$ and we want to avoid a useless equality test.

Of course, we could also have taken the maximum of the left subtree and this arbitrary asymmetry actually leads deletions followed by at least two insertions to trees which are less balanced, in average, than if they had been constructed directly only with insertions. This phenomenon is difficult to understand and examples are needed to see it at work

$$
\begin{aligned}
\mathsf{del}(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, \mathsf{del}(y, t_1), t_2), \text{ if } x \succ y;\\
\mathsf{del}(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, t_1, \mathsf{del}(y, t_2)), \text{ if } y \succ x;\\
\mathsf{del}(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{aux}_0(x, t_1, \mathsf{min}(t_2));\\
\mathsf{del}(y, \mathsf{ext}()) &\to \mathsf{ext}().\\
\\
\mathsf{min}(\mathsf{bst}(x, \mathsf{ext}(), t_2)) &\to (x, t_2);\\
\mathsf{min}(\mathsf{bst}(x, t_1, t_2)) &\to \mathsf{aux}_1(x, \mathsf{min}(t_1), t_2).\\
\\
\mathsf{aux}_1(x, (m, t_1'), t_2) &\to (m, \mathsf{bst}(x, t_1', t_2)).\\
\\
\mathsf{aux}_0(x, t_1, (m, t_2')) &\to \mathsf{bst}(m, t_1, t_2').
\end{aligned}
$$

Figure 8.18: Deletion of a key in a binary search tree

$$
\begin{aligned}
\mathsf{del}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, \mathsf{del}_0(y, t_1), t_2), \text{ if } x \succ y;\\
\mathsf{del}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{bst}(x, t_1, \mathsf{del}_0(y, t_2)), \text{ if } y \succ x;\\
\mathsf{del}_0(y, \mathsf{bst}(x, t_1, t_2)) &\to \mathsf{del}(x, t_1, t_2);\\
\mathsf{del}_0(y, \mathsf{del}(x, t_1, t_2)) &\to \mathsf{del}(x, \mathsf{del}_0(y, t_1), t_2), \text{ if } x \succ y;\\
\mathsf{del}_0(y, \mathsf{del}(x, t_1, t_2)) &\to \mathsf{del}(x, t_1, \mathsf{del}_0(y, t_2)), \text{ if } y \succ x;\\
\mathsf{del}_0(y, t) &\to t.
\end{aligned}
$$

Figure 8.19: Lazy deletion of a key in a binary search tree

(Eppinger, 1983, Culberson and Munro, 1989, Culberson and Evans, 1994, Knuth, 1998b, Heyer, 2009).

Another kind of asymmetry is that deletion is much more complicated to program than insertion. This fact has lead some researchers to propose a common framework for insertion and deletion (Andersson, 1991, Hinze, 2002). In particular, when Andersson's search with a tree candidate is modified into deletion, the program is quite short if the programming language is imperative.

Another approach to deletion consists in marking the targeted nodes as deleted without actually removing them. They are still needed for future comparisons but they are not to be considered part of the collection of keys implemented by the search tree. As such they are alike zombies, neither alive nor dead, or we could talk of lazy deletion. More seriously, this requires two kinds of internal nodes, bst/3 and del/3. This alternative design is shown in FIGURE 8.19. Note that the insertion of a key which happens to have been lazily deleted does not need to be performed at an external node: the constructor del/3 would simply be changed into bst/3, the mark of normal internal nodes.

Exercise Define the usual insertions on this new kind of search tree.

8.4 Average parameters

The average height h_n of a binary search tree of size n has been intensively studied (Devroye, 1986, 1987, Mahmoud, 1992, Knessl and Szpankowski, 2002), but the methods, mostly of analytic nature, are beyond the scope of this book. Reed (2003) proved that

$$h_n = \alpha \ln n - \frac{3\alpha}{2\alpha - 2} \ln \ln n + \mathcal{O}(1),$$

where α is the unique solution on $[2, +\infty[$ to the equation

$$\alpha \ln(2e/\alpha) = 1,$$

an approximation being $\alpha \simeq 4.31107$, and $\mathcal{O}(1)$ is an unknown function whose absolute value is bounded from above by an unknown constant. Particularly noteworthy is a rough logarithmic upper bound by Aslam (2001), expressed in a probabilistic model and republished by Cormen et al. (2009) in section 12.4.

Chauvin et al. (2001) studied the average width of binary search trees.

Bibliography

Arne Andersson. A note on searching in a binary search tree. *Software–Practice and experience*, 21(20):1125–1128, October 1991. (Short communication).

Arne Andersson. Balanced search trees made simple. In *Proceedings of the workshop on Algorithms and Data Structures*, volume 709/1993 of *Lecture Notes in Computer Science*, pages 60–71, 1993.

Arne Andersson, Christian Icking, Rolf Klein, and Thomas Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28 (2):165–178, 1990.

Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

Margaret Archibald and Julien Clément. Average depth in a binary search tree with repeated keys. In *Proceedings of the fourth colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, pages 209–320, September 2006.

Joe Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, July 2007.

Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010.

Thomas Arts and Jürgen Giesl. Termination of constructor systems. Technical report, Technische Hochschule Darmstadt, July 1996. ISSN 0924-3275.

Thomas Arts and Jürgen Giesl. Automatically proving termination where simplification orderings fail. In *Proceedings of the seventh International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science 1214, pages 261–272, Lille, France, 1997. Springer-Verlag.

Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 1-2(236):133–178, April 2000.

Thomas Arts and Jürgen Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, Aachen University of Technology, Department of Computer Science, September 2001. ISSN 0935-3232.

Javed A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Technical Report TR 2001-387, Deparment of Computer Science, Dartmouth College, 2001.

Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998.

Hendrik Pieter Barendregt. *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics), chapter Functional Programming and Lambda Calculus, pages 321–363. Elsevier Science, 1990.

Paul E. Black George Becker and Neil V. Murray. Formal verification of a merge-sort program with static semantics. In Kamal Karlapalem Amin Y. Noaman and Ken Barker, editors, *Proceedings of the ninth International Conference on Computing and Information*, pages 271–277, Winnipeg, Manitoba, Canada, June 1998.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. Springer, 2004.

Richard Bird. *Pearls of Functional Algorithm Design*, chapter The Knuth-Morris-Pratt algorithm, pages 127–135. Cambridge University Press, October 2010.

Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog now!*, volume 7 of *Texts in Computing.* College Publications, June 2006.

Stephen Bloch. Teaching linked lists and recursion without conditionals or null. *Journal of Computing Sciences in Colleges*, 18(5):96–108, May 2003. ISSN 1937-4771.

Peter B. Borwein. On the irrationality of certain series. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 112, pages 141–146, 1992.

Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, third edition, September 2000.

Gerald G. Brown and Bruno O. Shubert. On random binary trees. *Mathematics of Operations Research*, 9(1):43–65, February 1984.

Robert Creighton Buck. Mathematical induction and recursive definitions. *American Mathematical Monthly*, 70(2):128–135, February 1963.

William H. Burge. An analysis of binary search trees formed from sequences of nondistinct keys. *Journal of the ACM*, 23(2):451–454, July 1976.

F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

L. E. Bush. An asymptotic formula for the average sum of the digits of integers. *The American Mathematical Monthly*, 47(3):154–156, March 1940.

David Callan. Pair them up! A visual approach to the Chung-Feller theorem. *The College Mathematics Journal*, 26(3):196–198, May 1995.

Patrice Chalin and Perry James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the twenty-first European Conference on Object-Oriented Programming (ECOOP)*, pages 227–247, Berlin, Germany, 2007.

Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. College Publications, February 2004.

Brigitte Chauvin, Michael Drmota, and Jean Jabbour-Hattab. The profile of binary search trees. *The Annals of Applied Probability*, 11 (4):1042–1062, November 2001.

Wei-Mei Chen, Hsien-Kuei Hwang, and Gen-Huey Chen. The cost distribution of queue-mergesort, optimal mergesorts, and power-of-2 rules. *Journal of Algorithms*, 30(2):423–448, February 1999.

Richard C. Cobbe. *Much ado about nothing: Putting Java's null in its place*. PhD thesis, College of Computer and Information Science, Northeastern University, Boston, Massachusetts, USA, December 2008.

Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 20(2):79–86, January 1989.

Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11), November 1980.

Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, October 1998.

Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

Joseph Culberson and Patricia A. Evans. Asymmetry in binary search tree update algorithms. Technical Report TR 94-09, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, May 1994.

Joseph Culberson and J. Ian Munro. Explaining the behaviour of binary search trees under prolonged updates: A model and simulations. *The Computer Journal*, 32(1):68–75, 1989.

Olivier Danvy. On some functional aspects of control. In *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 445–449. Program Methodology Group, University of Göteborg and Chalmers University of Technology, Sweden, September 1988. Report 53.

Olivier Danvy. On listing list prefixes. *List Pointers*, 2(3/4):42–46, January 1989.

Olivier Danvy. Sur un exemple de Patrick Greussay. BRICS Report Series RS-04-41, Basic Research in Computer Science, University of Aarhus, Denmark, December 2004.

Olivier Danvy and Mayer Goldberg. There and back again. BRICS Report Series RS-01-39, Basic Research in Computer Science, University of Aarhus, Denmark, October 2001.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. BRICS Report Series RS-01-23, Basic Research in Computer Science, University of Aarhus, Denmark, June 2001.

B. Dasarathy and Cheng Yang. A transformation on ordered trees. *The Computer Journal*, 23(2):161–164, 1980.

Hubert Delange. Sur la fonction sommatoire de la fonction « somme des chiffres ». *L'Enseignement Mathématique*, XXI(1):31–47, 1975.

Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–115, February 1987. Corrigendum: Journal of Symbolic Computation (1987) **4**, 409–410.

Nachum Dershowitz. *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, chapter A taste of rewrite systems, pages 199–228. Springer, 1993.

Nachum Dershowitz. 33 examples of termination. In *Proceedings of the French Spring School of Theoretical Computer Science*, volume 909 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 1995.

Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics), chapter Rewrite Systems, pages 243–320. Elsevier Science, 1990.

Nachum Dershowitz and Christian Rinderknecht. **The Average Height of Catalan Trees by Counting Lattice Paths**. *Mathematics Magazine*, 88(3):187–195, June 2015. 18 pages (preprint, including supplement).

Nachum Dershowitz and Shmuel Zaks. Enumerations of ordered trees. *Discrete Mathematics*, 31(1):9–28, 1980.

Nachum Dershowitz and Shmuel Zaks. Applied tree enumerations. In *Proceedings of the Sixth Colloquium on Trees in Algebra and Programming*, volume 112 of *Lecture Notes in Computer Science*, pages 180–193, Berlin, Germany, 1981. Springer.

Nachum Dershowitz and Shmuel Zaks. The Cycle Lemma and some applications. *European Journal of Combinatorics*, 11(1):35–40, 1990.

Luc Devroye. A note on the height of binary search trees. *Journal of the Association for Computing Machinery*, 33(3):489–498, July 1986.

Luc Devroye. Branching processes in the analysis of the heights of trees. *Acta Informatica*, 24(3):277–298, 1987.

Edsger Wybe Dijkstra. Recursive Programming. *Numerische Mathematik*, 2(1):312–318, December 1960. Springer, ISSN 0029-599X.

Edsger Wybe Dijkstra. *A discipline of programming*. Series on Automatic Computation. Prentice Hall, October 1976.

Edsger Wybe Dijkstra. Why numbering should start at zero. University of Texas, Transcription EWD831, August 1982.

Kees Doets and Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing*. College Publications, May 2004.

Olivier Dubuisson. *ASN.1 Communication between heterogeneous systems*. Morgan Kaufmann, 2001.

Jefrey L. Eppinger. An empirical study of insertion and deletion in binary search trees. *Communications of the ACM*, 26(9):663–669, September 1983.

Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1992.

Yuguang Fang. A theorem on the k-adic representation of positive integers. *Proceedings of the American Mathematical Society*, 130(6): 1619–1622, June 2002.

Matthias Felleisen and Daniel P. Friedman. *A little Java, a few patterns*. The MIT Press, December 1997.

James Allen Fill. On the distribution of binary search trees under the random permutation model. *Random Structures and Algorithms*, 8 (1):1–25, January 1996.

Philippe Flajolet and Mordecai Golin. Mellin transforms and asymptotics: The Mergesort Recurrence. *Acta Informatica*, 31(7): 673–696, 1994.

Philippe Flajolet and Andrew M. Odlyzko. The average height of binary trees and other simple trees. Technical Report 56, Institut National de Recherche en Informatique et en Automatique (INRIA), February 1981.

Philippe Flajolet and Andrew M. Odlyzko. Limit distributions of coefficients of iterates of polynomials with applications to combinatorial enumerations. *Mathematical Proceedings of the Cambridge Philosophical Society*, 96:237–253, 1984.

Philippe Flajolet and Robert Sedgewick. Analytic combinatorics: Functional equations, rational and algebraic functions. Technical Report 4103, Institut National de Recherche en Informatique et en Automatique (INRIA), January 2001.

Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, January 2009.

Philippe Flajolet, Xavier Gourdon, and Philippe Dumas. Mellin Transforms and Asymptotics: Harmonic Sums. *Theoretical Computer Science*, 144:3–58, 1995.

Philippe Flajolet, Markus Nebel, and Helmut Prodinger. The scientific works of Rainer Kemp (1949–2004). *Theoretical Computer Science*, 355(3):371–381, April 2006.

Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31. American Mathematical Society, 1967.

Daniel P. Friedman and Mitchell Wand. *Essentials of Programing Languages*. Computer Science/Programming Languages Series. The MIT Press, third edition, 2008.

Jaco Geldenhuys and Brink Van der Merwe. Comparing leaf and root insertion. *South African Computer Journal*, 44:30–38, December 2009.

Thomas E. Gerasch. An insertion algorithm for a minimal internal path length binary search tree. *Communications of the ACM*, 31(5): 579–585, May 1988.

Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, USA, September 1998.

Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1):146–160, May 2001.

Jürgen Giesl. Automated termination proofs with measure functions. In *Proceedings of the Nineteenth Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pages 149–160. Springer-Verlag, 1995a.

Jürgen Giesl. Termination analysis for functional programs using term orderings. In *Proceedings of the second international Symposium on Static Analysis*, pages 154–171. Springer-Verlag, 1995b.

Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1), August 1997.

Jürgen Giesl, Christoph Walther, and Jürgen Brauburger. *Automated deduction: A basis for applications*, volume III (Applications) of *Applied Logic Series*, chapter Termination analysis for functional programs, pages 135–164. Kluwer Academic, Dordrecht, 1998.

Mayer Goldberg and Guy Wiener. Anonymity in Erlang. In *Erlang User Conference*, Stockholm, November 2009.

Mordecai J. Golin and Robert Sedgewick. Queue-mergesort. *Information Processing Letters*, 48(5):253–259, December 1993.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, third edition, 1994.

Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms*. Modern Birkhäuser Classics. Birkhäuser, Boston, USA, third edition, 2008.

Godfrey Harold Hardy. *Divergent series*. The Clarendon Press, Oxford, England, United Kingdom, 1949.

Michaela Heyer. Randomness preserving deletions on special binary search trees. *Electronic Notes in Theoretical Computer Science*, 225 (2):99–113, January 2009.

J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and Combinators*. Cambridge University Press, 2008.

Konrad Hinsen. The Promises of Functional Programming. *Computing in Science and Engineering*, 11(4):86–90, July/August 2009.

Ralf Hinze. A fresh look at binary search trees. *Journal of Functional Programming*, 12(6):601–607, November 2002. (Functional Pearl).

Yoichi Hirai and Kazuhiko Yamamoto. Balancing weight-balanced trees. *Journal of Functional Programming*, 21(3):287–307, 2011.

Charles A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

Tony Hoare. Null references: The billion dollar mistake. In *The Annual International Software Development Conference*, London, England, United Kingdom, August 2009.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education, 2nd edition, 2003.

Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5): 549–554, September 1997.

Gérard Huet. Linear Contexts, Sharing Functors: Techniques for Symbolic Computation. In *Thirty Five Years of Automating Mathematics*, volume 28 of *Applied Logic Series*, pages 49–69. Springer Netherlands, 2003.

John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

Katherine Humphreys. A history and a survey of lattice path enumeration. *Journal of Statistical Planning and Inference*, 140(8): 2237–2254, August 2010. Special issue on Lattice Path Combinatorics and Applications.

Hsien-Kuei Hwang. Asymptotic expansions of the mergesort recurrences. *Acta Informatica*, 35(11):911–919, November 1998.

Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of the conference on Object-Oriented Programming Systems, Languages and Applications*, pages 347–349, Portland, Oregon, USA, September 1986.

Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 71, Department of Computer Science,University of Auckland, New Zealand, May 1993.

Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference.* Wiley Publishings (Wrox), fourth edition, 2008.

Reiner Kemp. *Fundamentals of the average case analysis of particular algorithms*. Wiley-Teubner Series in Computer Science. John Wiley & Sons, B. G. Teubner, 1984.

Charles Knessl and Wojciech Szpankowski. The height of a binary search tree: The limiting distribution perspective. *Theoretical Computer Science*, 289(1):649–703, October 2002.

Donald E. Knuth. *Selected papers on Computer Science*, chapter Von Neumann's First Computer Program, pages 205–226. Number 59 in CSLI Lecture Notes. CSLI Publications, Stanford University, California, USA, 1996.

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1998a.

Donald E. Knuth. *Sorting and Searching*, chapter Binary Tree Searching, 6.2.2, pages 431–435. Addison-Wesley, 1998b.

Donald E. Knuth. *Selected Papers on the Analysis of Algorithms*, chapter Textbook Examples of Recursion, pages 391–414. Number 102 in CSLI Lecture Notes. CSLI Publications, Stanford University, California, USA, 2000.

Donald E. Knuth. *Selected Papers on Design of Algorithms*, chapter Fast pattern matching in strings, pages 99–135. Number 191 in CSLI Lecture Notes. CSLI Publications, Stanford University, California, USA, 2010.

Donald E. Knuth. *Combinatorial algorithms*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, 2011.

Donald E. Knuth, Nicolaas Govert de Bruijn, and S. O. Rice. *Graph Theory and Computing*, chapter The Average Height of Planted Plane Trees, pages 15–22. Academic Press, December 1972. Republished in Knuth et al. (2000).

Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2): 323–350, June 1977. Society for Industrial and Applied Mathematics.

Donald E. Knuth, Nicolaas Govert de Bruijn, and S. O. Rice. *Selected Papers on the Analysis of Algorithms*, chapter The Average Height of

Planted Plane Trees, pages 215–223. Number 102 in CSLI Lecture Notes. CSLI Publications, Stanford University, California, USA, 2000.

Đuro Kurepa. On the left factorial function !n. *Mathematica Balkanica*, 1:147–153, 1971.

John Larmouth. *ASN.1 Complete*. Morgan Kaufmann, November 1999.

Chung-Chih Li. An immediate approach to balancing nodes in binary search trees. *Journal of Computing Sciences in Colleges*, 21(4): 238–245, April 2006.

Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Unfolding the mystery of mergesort. In *Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*, pages 206–225. Springer, 1998.

M. Lothaire. *Applied Combinatorics on Words*, chapter Counting, Coding and Sampling with Words, pages 478–519. Number 105 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, United Kingdom, July 2005.

Hosam M. Mahmoud. *Evolution of random search trees*. Discrete Mathematics and Optimization. Wiley-Interscience, New York, USA, 1992.

Erkki Mäkinen. A survey on binary tree codings. *The Computer Journal*, 34(5), 1991.

Sal Mangano. *XSLT Cookbook*. O'Reilly, 2nd edition, 2006.

George Edward Martin. *Counting: The art of enumerative combinatorics*. Springer, 2001.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine (Part I). *Communications of the ACM*, 3 (4):184–195, April 1960.

John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28. North-Holland, 1962.

M. D. McIlroy. The number of 1's in the binary integers: Bounds and extremal properties. *SIAM Journal on Computing*, 3(4):255–261, December 1974. Society for Industrial and Applied Mathematics.

Kurt Mehlhorn and Athanasios Tsakalidis. *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter Data Structures, pages 301–341. Elsevier Science, 1990.

Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24(2):70–77, 1992.

Sri Gopal Mohanty. *Lattice path counting and applications*, volume 37 of *Probability and mathematical statistics*. Academic Press, New York, USA, January 1979.

Shin-Chen Mu and Richard Bird. Rebuilding a tree from its traversals: A case study of program inversion. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, LNCS 2895, pages 265–282, 2003.

Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, pages 563–582, Nashville, Tennesse, USA, October 2008.

Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly, October 2006.

Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. In *Proceedings of the fourth annual ACM symposium on Theory of Computing*, pages 137–142, Denver, Colorado, USA, May 1972.

Andrew M. Odlyzko. Some new methods and results in tree enumeration. *Congressus Numerantium*, 42:27–52, 1984.

Chris Okasaki. Simple and efficient purely functional queues and dequeues. *Journal of Functional Programming*, 5(4):583–592, October 1995.

Chris Okasaki. *Purely Functional Data Structures*, chapter Fundamentals of Amortization, pages 39–56. Cambridge University Press, 1998a. Section 5.2.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998b.

Chris Okasaki. Breadth-first numbering: Lessons from a small exercise in algorithm design. In *Proceedings of the fifth ACM SIGPLAN*

International Conference on Functional Programming, pages 131–136, Montréal, Canada, September 2000.

A. Panayotopoulos and A. Sapounakis. On binary trees and Dyck paths. *Mathématiques et Sciences Humaines, No. 131*, pages 39–51, 1995.

Wolfgang Panny and Helmut Prodinger. Bottom-up mergesort: A detailed analysis. *Algorithmica*, 14(4):340–354, October 1995.

Tomi A. Pasanen. Note: Random binary search tree with equal elements. *Theoretical Computer Science*, 411(43):3867–3872, October 2010.

Dominique Perrin. *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics), chapter Finite Automata, pages 3–57. Elsevier Science, 1990.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

Bruce Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, May 2003.

Mireille Régnier. Knuth-Morris-Pratt algorithm: An analysis. In *Proceedings of the conference on Mathematical Foundations for Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 431–444, Porubka, Poland, 1989.

Mireille Régnier. Average performance of Morris-Pratt-like algorithms. Technical Report 2164, Institut National de Recherche en Informatique et en Automatique (INRIA), January 1994. ISSN 0249-6399.

Marc Renault. Lost (and Found) in Translation: André's Actual Method and Its Application to the Generalized Ballot Problem. *American Mathematical Monthly*, 155(4):358–363, April 2008.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM annual conference*, volume 2, pages 717–740, 1972.

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

Christian Rinderknecht. **Une analyse syntaxique d'ASN.1:1990 en Caml Light**. Technical Report 171, INRIA, April 1995. English at `http://crinderknecht.free.fr/pub/TR171-eng.pdf`.

Christian Rinderknecht. A Didactic Analysis of Functional Queues. *Informatics in Education*, 10(1):65–72, April 2011.

Raphael M. Robinson. Primitive recursive functions. *Bulletin of the American Mathematical Society*, 53(10):925–942, 1947.

Raphael M. Robinson. Recursion and double recursion. *Bulletin of the American Mathematical Society*, 54(10):987–993, 1948.

Frank Ruskey. A simple proof of a formula of Dershowitz and Zaks. *Discrete Mathematics*, 43(1):117–118, 1983.

Jacques Sakarovitch. *Éléments de théorie des automates*. Les classiques de l'informatique. Vuibert Informatique, 2003.

Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley, 1996.

David B. Sher. Recursive objects: An object oriented presentation of recursion. *Mathematics and Computer Education*, Winter 2004.

Iekata Shiokawa. On a problem in additive number theory. *Mathematical Journal of Okayama University*, 16(2):167–176, June 1974.

David Spuler. The best algorithm for searching a binary search tree. Technical Report 92/3, Department of Computer Science, James Cook University of North Queensland, Australia, 1992.

Richard P. Stanley. *Enumerative Combinatorics*, volume 1 of *Cambridge Studies in Advanced Mathematics (No. 49)*. Cambridge University Press, July 1999a.

Richard P. Stanley. *Enumerative Combinatorics*, volume 2 of *Cambridge Studies in Advanced Mathematics (No. 62)*. Cambridge University Press, April 1999b.

C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer and Information Sciences (now International Journal of Parallel Programming)*, 9(1):15–29, 1980.

Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Advanced Programming Techniques. The MIT Press, second edition, 1994.

Kenneth B. Stolarsky. Power and exponential sums of digital sums related to binomial coefficient parity. *SIAM Journal of Applied Mathematics*, 32(4):717–730, June 1977.

J. R. Trollope. An explicit expression for binary digital sums. *Mathematics Magazine*, 41(1):21–25, January 1968.

Franklin Turbak and David Gifford. *Design Concepts in Programming Languages*. Computer Science/Programming Languages Series. The MIT Press, 2008.

Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges*, 14(4):86–101, May 1999.

Jeffrey Scott Vitter and Philippe Flajolet. *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter Average-Case Analysis of Algorithms and Data Structures, pages 431–524. Elsevier Science, 1990.

Patricia Walmsley. *Definitive XML Schema*. The Charles F. GoldFarb Definitive XML Series. Prentice-Hall PTR, 2002.

Tjark Weber and James Caldwell. Constructively characterizing fold and unfold. *Logic-based program synthesis and transformation*, 3018/2004:110–127, 2004. Lecture Notes in Computer Science.

Herbert S. Wilf. *Generatingfunctionology*. Academic Press, 1990.

Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, 1993.

Index