Tobias Nipkow, Jasmin Blanchette,
Manuel Eberl, Alejandro Gómez-Londoño
Peter Lammich, Christian Sternagel,
Simon Wimmer, Bohua Zhan

Functional Algorithms, Verified!

April 4, 2021

Preface

This book is an introduction to data structures and algorithms for functional languages, with a focus on proofs. It covers both functional correctness and running time analysis. It does so in a unified manner with inductive proofs about functional programs and their running time functions.

The unique feature of this book is that all the proofs have been machine-checked by the proof assistant Isabelle. That is, in addition to the text in the book, *which requires no knowledge of proof assistants*, there are the Isabelle definitions and proofs that can be accessed by following (in the PDF file) the links attached to section headings with a ↗ symbol. Our message:

> *Proof assistants are the right tools for truly trustworthy proofs about efficient algorithms.*

This book has been classroom-tested for a number of years in a course for graduate and advanced undergraduate students. At the same time it is a reference for programmers and researchers who are interested in the details of some algorithm or proof.

Isabelle ↗

Isabelle [63, 76, 57] is a proof assistant for the logic HOL (= Higher-Order Logic), which is why the system is often called Isabelle/HOL. HOL is a generalization of first-order logic: functions can be passed as parameters and returned as results, just as in functional programming, and they can be quantified over. Isabelle's version of HOL also supports a simple version of Haskell's type classes.

As emphasized above, the reader need not be familiar with Isabelle or HOL in order to read this book. However, to take full advantage of our proof assistant approach, readers are encouraged to learn how to write Isabelle definitions and proofs themselves — and to solve some of the exercises in this book. To this end we recommend the tutorial *Programming and Proving in Isabelle/HOL* [52], which is also Part I of the book *Concrete Semantics* [56].

Prerequisites

We expect the reader to be familiar with

- the basics of discrete mathematics: propositional and first-order logic, sets and relations, proof principles including induction;
- a typed functional programming language like Haskell [27], OCaml [60] or Standard ML [64];
- simple inductive proofs about functional programs.

Under Development

This book is meant to grow. New chapters are meant to be added over time. The list of authors is meant to grow — *you* could become one of them!

Colour

For the quick orientation of the reader, definitions are displayed in coloured boxes:

These boxes display functional programs.

These boxes display auxiliary definitions.

From a logical point of view there is no difference between the two kinds of definitions except that auxiliary definitions need not be executable.

Acknowledgements We are obviously indebted to the books by Cormen *et al.* [13] and Okasaki [62]. Mohammad Abdulaziz, Magnus Myreen and Larry Paulson commented on parts of the book. We are very grateful to all of them.

Contents

Part I Sorting and Selection

Part II Search Trees

Part III Priority Queues

Part IV Advanced Design and Analysis Techniques

Part V Appendix

1

Basics

In this chapter we describe the basic building blocks the book rests on.

Programs: The functional programming language we use is merely sketched because of its similarity with other well known functional languages.

Predefined types and notation: We introduce the basic predefined types and notations used in the book.

Inductive proofs: Although we do not explain proofs in general, we make an exception for certain inductive proofs.

Running time: We explain how we model running time by step counting functions.

1.1 Programs

The programs in this book are written in Isabelle's functional programming language which provides recursive algebraic data types (keyword: **datatype**), recursive function definitions and **let**, **if** and **case** expressions. The language is sufficiently close to a number of similar typed functional languages (SML [64], OCaml [60], Haskell [27]) to obviate the need for a detailed explanation. Moreover, Isabelle can generate SML, OCaml, Haskell and Scala code [24]. What distinguishes Isabelle's functional language from ordinary programming languages is that all functions in Isabelle must terminate. Termination must be proved. For all the functions in this book, termination is not difficult to see and Isabelle can prove it automatically. (If you want to go beyond, consult the function definition tutorial [43].)

Isabelle's functional language is pure logic. All language elements have precise definitions. However, this book is about algorithms, not programming language semantics. A functional programmer's intuition suffices for reading it. (If you want to know more about the logical basis see [7, 44, 26].)

A useful bit of notation: any infix operator can be turned into a function by enclosing it in parentheses, e.g. $(+)$.

1.2 Types

Type variables are denoted by $'a$, $'b$, etc. The function type arrow is $\Rightarrow$. Type constructor names follow their argument types, e.g. $'a\ \mathit{list}$. The notation $t :: \tau$ means that term t has type τ. The following types are predefined.

Booleans

Type *bool* comes with the constants *True* and *False* and the usual operations.

Numbers

There are three numeric types: the natural numbers *nat* $(0, 1, \ldots)$, the integers *int* and the real numbers *real*. They correspond to the mathematical sets $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ and not to any machine arithmetic. All three types come with the usual (overloaded) operations.

Sets

The type $'a\ \mathit{set}$ of sets (finite and infinite) over type $'a$ comes with the standard mathematical operations. The minus sign "$-$", unary or binary, can denote set complement or difference.

Lists

The type $'a\ \mathit{list}$ of lists whose elements are of type $'a$ is a recursive data type:

```
datatype 'a list = Nil | Cons 'a ('a list)
```

Constant *Nil* represents the empty list and $\mathit{Cons}\ x\ \mathit{xs}$ represents the list with first element x, the **head**, and rest list xs, the **tail**. The following syntactic sugar is sprinkled on top;

$$\begin{aligned} [] &\equiv \mathit{Nil} \\ x \mathbin{\#} \mathit{xs} &\equiv \mathit{Cons}\ x\ \mathit{xs} \\ [x_1, \ldots, x_n] &\equiv x_1 \mathbin{\#} \ldots \mathbin{\#} x_n \mathbin{\#} [] \end{aligned}$$

The $\equiv$ symbol means that the left-hand side is merely an abbreviation of the right-hand side.

A library of predefined functions on lists is shown in Appendix A. The length of a list xs is denoted by $|\mathit{xs}|$.

Type 'a option

The data type *'a option* is defined as follows:

datatype *'a option* = *None* | *Some 'a*

Pairs and Tuples

Pairs are written $(a,\ b)$. Functions *fst* and *snd* select the first and second component of a pair: $\mathit{fst}\ (a,\ b) = a$ and $\mathit{snd}\ (a,\ b) = b$. The type *unit* contains only a single element $()$, the empty tuple.

1.2.1 Pattern Matching

Functions are defined by equations and pattern matching, for example over lists. Natural numbers may also be used in pattern-matching definitions:

$$\mathit{fib}\ (n + 2) = \mathit{fib}\ (n + 1) + \mathit{fib}\ n$$

Occasionally we use an extension of pattern matching where patterns can be named. For example, the defining equation

$$f\ (x\ \#\ (y\ \#\ \mathit{zs}\ =:\ \mathit{ys})) = \mathit{ys}\ @\ \mathit{zs}$$

introduces a variables *ys* on the left that stands for $y\ \#\ \mathit{zs}$ and can be referred to on the right. Logically it is just an abbreviation of

$$f\ (x\ \#\ y\ \#\ \mathit{zs}) = (\textbf{let}\ \mathit{ys} = y\ \#\ \mathit{zs}\ \textbf{in}\ \mathit{ys}\ @\ \mathit{zs})$$

although it is suggestive of a more efficient interpretation. The general format is $\mathit{pattern}\ =:\ \mathit{variable}$.

1.2.2 Numeric Types and Coercions

The numeric types *nat*, *int* and *real* are all distinct. Converting between them requires explicit **coercion** functions, in particular the **inclusion** functions $\mathit{int} :: \mathit{nat} \Rightarrow \mathit{int}$ and $\mathit{real} :: \mathit{nat} \Rightarrow \mathit{real}$ that do not lose any information (in contrast to coercions in the other direction). We do not show inclusions unless they make a difference. For example, $(m + n) :: \mathit{real}$, where $m,\ n :: \mathit{nat}$, is mathematically unambiguous because $\mathit{real}\ (m + n) = \mathit{real}\ m + \mathit{real}\ n$. On the other hand, $(m - n) :: \mathit{real}$ is ambiguous because $\mathit{real}\ (m - n) \neq \mathit{real}\ m - \mathit{real}\ n$ because $(0{::}\mathit{nat}) - n = 0$. In some cases we can also drop coercions that are not inclusions, e.g. $\mathit{nat} :: \mathit{int} \Rightarrow \mathit{nat}$, which coerces negative integers to 0: if we know that $i \geq 0$ then we can drop the *nat* in $\mathit{nat}\ i$.

We prefer type *nat* over type *real* for ease of (Isabelle) proof. For example, for $m,\ n :: \mathit{nat}$ we prefer $m \leq 2^n$ over $\lg\ m \leq n$.

1.2.3 Multisets

Informally, a multiset is a set where elements can occur multiple times. Multisets come with the following operations:

$$\begin{aligned}
\{\!\{\}\!\} &:: {}'a\ \mathit{multiset} \\
(\in_{\#}) &:: {}'a \Rightarrow {}'a\ \mathit{multiset} \Rightarrow \mathit{bool} \\
\mathit{add\_mset} &:: {}'a \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \\
(+) &:: {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \\
\mathit{size} &:: {}'a\ \mathit{multiset} \Rightarrow \mathit{nat} \\
\mathit{mset} &:: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{multiset} \\
\mathit{set\_mset} &:: {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{set} \\
\mathit{image\_mset} &:: ({}'a \Rightarrow {}'b) \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'b\ \mathit{multiset} \\
\mathit{filter\_mset} &:: ({}'a \Rightarrow \mathit{bool}) \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'b\ \mathit{multiset} \\
\mathit{sum\_mset} &:: {}'a\ \mathit{multiset} \Rightarrow {}'a
\end{aligned}$$

Their meaning: $\{\!\{\}\!\}$ is the empty multiset; $(\in_{\#})$ is the element test; *add_mset* adds an element to a multiset; $(+)$ is the union of two multisets, where multiplicities of elements are added; *size* M, written $|M|$, is the number of elements in M, taking multiplicities into account; *mset* converts a list into a multiset by forgetting about the order of elements; *set_mset* converts a multiset into a set; *image_mset* applies a function to all elements of a multiset; *filter_mset* removes all elements from a multiset that do not satisfy the given predicate; *sum_mset* is the sum of the values of a multiset, the iteration of $(+)$ (taking multiplicity into account). Note that the instance $\mathit{sum\_mset} :: {}'a\ \mathit{multiset}\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset}$ takes the union of a multiset of multisets.

We use some additional suggestive syntax for some of these operations:

$$\begin{aligned}
\{\!\{x \in_{\#} M \mid P\ x\}\!\} &\equiv \mathit{filter\_mset}\ P\ M \\
\{\!\{f\ x \mid x \in_{\#} M\}\!\} &\equiv \mathit{image\_mset}\ f\ M \\
\textstyle\sum_{\#} M &\equiv \mathit{sum\_mset}\ M \\
\textstyle\sum_{x \in_{\#} M} f\ x &\equiv \mathit{sum\_mset}\ (\mathit{image\_mset}\ f\ M)
\end{aligned}$$

See Section C.3 in the appendix for an overview of such syntax.

1.3 Notation

We adopt the following notation:

- The type of functions ${}'a \Rightarrow {}'b$ comes with a predefined pointwise update operation with its own notation:
  $f(a := b) = (\lambda x.\ \textbf{if}\ x = a\ \textbf{then}\ b\ \textbf{else}\ f\ x)$

- Function lg is the binary logarithm.

We deviate from Isabelle's notation in favour of standard mathematics in a number of points:

- There is only one implication: $\Longrightarrow$ is printed as $\longrightarrow$ and $P \Longrightarrow Q \Longrightarrow R$ is printed as $P \wedge Q \longrightarrow R$.
- Multiplication is printed as $x \cdot y$.
- Exponentation is uniformly printed as x^y.
- We sweep under the carpet that type *nat* is defined as a recursive data type: **datatype** $nat = 0 \mid Suc\ nat$. In particular, constructor *Suc* is hidden: $Suc^k\ 0$ is printed as k and $Suc^k\ n$ (where n is not 0) is printed as $n + k$.
- Set comprehension syntax is the canonical $\{x \mid P\}$.

The reader who consults the Isabelle theories referred to in this book should be aware of these discrepancies.

1.4 Proofs

Proofs are the *raison d'être* of this book. Thus we present them in more detail than is customary in a book on algorithms. However, not all proofs:

- We omit proofs of simple properties of numbers, lists, sets and multisets, our predefined types. Obvious properties (e.g. $|xs\ @\ ys| = |xs| + |ys|$ or commutativity of $\cup$) are used implicitly without proof.
- With some exceptions, we only state properties if their proofs require induction, in which case we will say so, and we will always indicate which supporting properties were used.
- If a property is simply described as "inductive" or its proof is described by a phrase like "by an easy/automatic induction" it means that in the Isabelle proofs all cases of the induction were automatic, typically by simplification.

As a simple example of an easy induction consider the append function

$$(@) :: {}'a\ list \Rightarrow {}'a\ list \Rightarrow {}'a\ list$$

$$[]\ @\ ys = ys$$
$$(x\ \#\ xs)\ @\ ys = x\ \#\ xs\ @\ ys$$

and the proof of $(xs\ @\ ys)\ @\ zs = xs\ @\ ys\ @\ zs$ by structural induction on xs. (Note that $(@)$ associates to the right.) The base case is trivial by definition: $([]\ @\ ys)\ @\ zs = []\ @\ ys\ @\ zs$. The induction step is easy:

$$\begin{aligned}
&(x \mathbin{\#} xs \mathbin{@} ys) \mathbin{@} zs \\
&= x \mathbin{\#} (xs \mathbin{@} ys) \mathbin{@} zs && \text{by definition of } (@) \\
&= x \mathbin{\#} xs \mathbin{@} ys \mathbin{@} zs && \text{by IH}
\end{aligned}$$

Note that **IH** stands for *Induction Hypothesis*, in this case $(xs \mathbin{@} ys) \mathbin{@} zs = xs \mathbin{@} ys \mathbin{@} zs$.

1.4.1 Computation Induction

Because most of our proofs are about recursive functions, most of them are by induction, and we say so explicitly. If we do not state explicitly what form the induction takes, it is by an obvious structural induction. The alternative and more general induction schema is **computation induction** where the induction follows the terminating computation, but from the bottom up. For example, the terminating recursive definition for $gcd :: nat \Rightarrow nat \Rightarrow nat$

$$gcd\ m\ n = (\textbf{if}\ n = 0\ \textbf{then}\ m\ \textbf{else}\ gcd\ n\ (m\ mod\ n))$$

gives rise to the following induction schema:

If $(n \neq 0 \longrightarrow P\ n\ (m\ mod\ n)) \longrightarrow P\ m\ n$ (for all m and n),
then $P\ m\ n$ (for all m and n).

In general, let $f :: \tau \Rightarrow \tau'$ be a terminating function of, for simplicity, one argument. Proving $P(x :: \tau)$ by induction on the computation of f means proving

$$P\ r_1 \wedge \ldots \wedge P\ r_n \longrightarrow P\ e$$

for every defining equation

$$f\ e = \ldots\ f\ r_1\ \ldots\ f\ r_n\ \ldots$$

where $f\ r_1, \ldots, f\ r_n$ are all the recursive calls. For simplicity we have ignored the **if** and **case** contexts that a recursive call $f\ r_i$ occurs in and that should be preconditions of the assumption $P\ r_i$ as in the gcd example. If the defining equations for f overlap, the above proof obligations are stronger than necessary.

1.5 Running Time

Our approach to reasoning about the **running time** of a function f is very simple: we explicitly define a function T_f such that $T_f\ x$ models the time the computation of $f\ x$ takes. Unless stated otherwise, T_f counts the number of all function calls in the computation of f. It is not intended that T_f yields the exact running time but only that the running time of f is in $O(T_f)$.

Given a function $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ we define a **(running) time function** $T_f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow nat$ by translating every defining equation for f into a defining equation for T_f. The translation is defined by two functions: $\mathcal{E}$ translates defining equations for f to defining equations for T_f and $\mathcal{T}$ translates expressions that compute some value to expressions that computes the number of function calls. The unusual notation $\mathcal{E}[\![.]\!]$ and $\mathcal{T}[\![.]\!]$ emphasizes that they are not functions in the logic.

$$\begin{aligned}\mathcal{E}[\![f\ p_1\ \ldots\ p_n = e]\!] &= (T_f\ p_1\ \ldots\ p_n = \mathcal{T}[\![e]\!] + 1)\\ \mathcal{T}[\![g\ e_1\ \ldots\ e_k]\!] &= \mathcal{T}[\![e_1]\!] + \ldots + \mathcal{T}[\![e_k]\!] + T_g\ e_1\ \ldots\ e_k\end{aligned} \tag{1.1}$$

This is the general idea. It requires some remarks and clarifications:

- This definition of T_f is an abstraction of a call-by-value semantics. Thus it is also correct for lazy evaluation but may be a very loose upper bound.
- Definition (1.1) is incomplete: if g is a variable or constructor function (e.g. *Nil* or *Cons*), then there is no defining equation and thus no T_g. We simply define $T_g\ \ldots = 1$ if g is a variable, constructor function or predefined function on *bool* or numbers. These choices are discussed below.
- Conditionals and **case** expressions are treated specially:

$$\begin{aligned}&\mathcal{T}[\![\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]\\ &= \mathcal{T}[\![b]\!] + (\textbf{if}\ b\ \textbf{then}\ \mathcal{T}[\![e_1]\!]\ \textbf{else}\ \mathcal{T}[\![e_2]\!])\\ &\mathcal{T}[\![\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k]\!]\\ &= \mathcal{T}[\![e]\!] + (\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow \mathcal{T}[\![e_1]\!] \mid \ldots \mid p_k \Rightarrow \mathcal{T}[\![e_k]\!])\end{aligned}$$

- **let** expressions are also special:

$$\mathcal{T}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] = \mathcal{T}[\![e_1]\!] + (\lambda x.\ \mathcal{T}[\![e_2]\!])\ e_1$$

  Note that $\mathcal{T}[\![e_2]\!]$ must be evaluated completely before $(\lambda x.\ \mathcal{T}[\![e_2]\!])\ e_1$ is β-contracted to ensure that $\mathcal{T}[\![x]\!] = 1$ (see above) kicks in instead of generating multiple copies of $\mathcal{T}[\![e_1]\!]$.
- For simplicity we restrict ourselves to a first-order language above. Nevertheless we use a few basic higher-order functions like *map* in the book. Their running time functions are defined in Appendix B.1.

As an example consider the append function (@) defined above. The defining equations for $T_{append} :: {'a}\ list \Rightarrow {'a}\ list \Rightarrow nat$ are easily derived. The first equation translates like this:

$$\begin{aligned}&\mathcal{E}[\![[]\ @\ ys = ys]\!]\\ &= \ (T_{append}\ []\ ys = \mathcal{T}[\![ys]\!] + 1)\\ &= \ (T_{append}\ []\ ys = 2)\end{aligned}$$

The right-hand side of the second equation translates like this:

$$\begin{aligned}&\mathcal{T}[\![x\ \#\ xs\ @\ ys]\!]\\ &= \mathcal{T}[\![x]\!] + \mathcal{T}[\![xs\ @\ ys]\!] + T_{Cons}\ x\ (xs\ @\ ys)\\ &= 1 + (\mathcal{T}[\![xs]\!] + \mathcal{T}[\![ys]\!] + T_{append}\ xs\ ys) + 1\\ &= 1 + (1 + 1 + T_{append}\ xs\ ys) + 1\end{aligned}$$

Thus the two defining equations for T_{append} are

$$\begin{aligned}&T_{append}\ []\ ys = 2\\ &T_{append}\ (x\ \#\ xs)\ ys = T_{append}\ xs\ ys + 4\end{aligned}$$

The constants 2 and 4 in the above definition are somewhat arbitrary because accessing a variable and calling (#) take different amounts of time, even if we assume they are both constant-time operations. Because we are only interested in $O(T_f)$ it is justified to reduce all additive constants to 1. This is what we will call the **canonical running time function**. Thus the canonical T_{append} is this:

$$\begin{aligned}&T_{append}\ []\ ys = 1\\ &T_{append}\ (x\ \#\ xs)\ ys = T_{append}\ xs\ ys + 1\end{aligned}$$

Unless noted otherwise we always work with canonical running time functions. This may also involve looking inside **if**/**case** when reducing additive constants, e.g. reducing (**if** b **then** $2 + e$ **else** 3) $+ 2$ to **if** b **then** $1 + e$ **else** 1.

Occasionally we will apply one more simplification step: when defining some T_f, you may drop additive constants when they are added to some term $T_g \ldots$ where T_g has already been defined (i.e. $g \neq f$ and f and g are not mutually recursive). This is because by construction all T_g return positive results and adding a constant does not change $O(T_f)$.

In the main body of the book we initially show the definition of each T_f. Once the principles above have been exemplified sufficiently, the time functions are relegated to Appendix B.

1.5.1 Example: List Reversal

This section exemplifies not just the definition of time functions but also their analysis. The standard list reversal function *rev* is defined in Appendix A. This is the corresponding canonical time function:

$$T_{rev} :: {}'a\ list \Rightarrow nat$$

$$T_{rev}\ [] = 1$$
$$T_{rev}\ (x\ \#\ xs) = T_{rev}\ xs + T_{append}\ (rev\ xs)\ [x] + 1$$

The canonical T_{append} was shown before.

A simple induction shows $T_{append}\ xs\ ys = |xs| + 1$. The precise formula for T_{rev} is less immediately obvious (exercise!) but an upper bound is easy to guess and verify by induction:

$$T_{rev}\ xs \le (|xs| + 1)^2$$

We will frequently prove upper bounds only.

Of course one can also reverse a list in linear time:

$$itrev :: {}'a\ list \Rightarrow {}'a\ list \Rightarrow {}'a\ list$$

$$itrev\ []\ ys = ys$$
$$itrev\ (x\ \#\ xs)\ ys = itrev\ xs\ (x\ \#\ ys)$$

$$T_{itrev} :: {}'a\ list \Rightarrow {}'a\ list \Rightarrow nat$$

$$T_{itrev}\ []\ \_ = 1$$
$$T_{itrev}\ (x\ \#\ xs)\ ys = T_{itrev}\ xs\ (x\ \#\ ys) + 1$$

Function *itrev* has linear running time: $T_{itrev}\ xs\ ys = |xs| + 1$. A simple induction yields $itrev\ xs\ ys = rev\ xs\ @\ ys$. Thus *itrev* implements *rev*: $rev\ xs = itrev\ xs\ []$.

1.5.2 Discussion

Analysing the running time of any program, let alone a functional one, is tricky unless you have two formally specified entities: a machine model that includes a notion of running time and a translation from programs to machines. For imperative programs the standard model is the Random Access Machine (RAM) where each instruction takes one time unit. For functional programs a standard measure is the number of function calls. We adopt this measure and equate it with time although a more honest name would be computation steps.

A full proof that the execution time of our functional programs is in $O(T_f)$ on some actual soft- and hardware is a major undertaking: one would need to formalize the full stack of compiler, runtime system and hardware. We do not offer such a proof. Thus our formalization of “time” should be seen as conditional: given a stack that satisfies our basic assumptions in the definition of $\mathcal{E}$ and $\mathcal{T}$, our analyses are correct for that stack. We argue that these assumptions are not unreasonable (on a RAM) provided we accept that both the address space and numbers have a fixed size and cannot grow arbitrarily. Of course this means that actual program execution may abort if the resources are exhausted.

Our basic assumption is that function calls take constant time. This is reasonable because we just need to allocate (and later deallocate) a new stack frame of constant size because all parameters are references or numbers and thus of fixed size. We also assumed that variable access takes constant time. This is a standard RAM assumption. Assuming that constructor functions take constant time is reasonable because the memory manager could simply employ a single reference to the first free memory cell and increment that with each constructor function call. How to account for garbage collection is less clear. In the worst case we have to assume that garbage collection is switched off, which simply exhausts memory more quickly. Finally we assume that operations on *bool* and numbers take constant time. The former is obvious, the latter follows from our assumption that we have fixed-size numbers.

In the end, we are less interested in a specific model of time and more in the principle that time (and other resources) can be analyzed just as formally as functional correctness once the ground rules (e.g. $\mathcal{T}$) have been established.

1.5.3 Asymptotic Notation

The above approach to running time analysis is nicely concrete and avoids the more sophisticated machinery of asymptotic notation, $O(.)$ and friends. Thus we have intentionally lowered the entry barrier to the book for readers who want to follow the Isabelle formalization: we require no familiarity with Isabelle’s real analysis library and in particular with the existing formalization of and automation for asymptotic notation [17]. Of course this comes at a price: one has to come up with and reason about somewhat arbitrary constants in the analysis of individual functions. Moreover we rarely appeal to the “master theorem" (although Eberl [17] provides a generalized version) but prove solutions to recurrence equations correct by induction. Again, this is merely to reduce the required mathematical basis and to show that it can be done. In informal explanations, typically when considering inessential variations, we do use standard mathematical notation and write, for example, $O(n \lg n)$.

Part I

Sorting and Selection

2

Sorting

In this chapter we define and verify the following sorting functions: insertion sort, quicksort, and one top-down and two bottom-up merge sorts. We also analyze their running times, except for quicksort.

Sorting involves an ordering. In this book we assume such an ordering to be provided by comparison operators $\leq$ and $<$ defined on the underlying type.

Sortedness of lists is defined as follows:

```
sorted :: ('a::linorder) list ⇒ bool
sorted [] = True
sorted (x # ys) = ((∀y∈set ys. x ≤ y) ∧ sorted ys)
```

That is, every element is $\leq$ than all elements to the right of it: the list is sorted in increasing order. But what does *'a::linorder* mean?

The type variable *'a* is annotated with *linorder*, which means that *sorted* is only applicable if a binary predicate $(\leq) :: {'a} \Rightarrow {'a} \Rightarrow \mathit{bool}$ is defined and $(\leq)$ is a **linear order**, i.e. the following properties are satisfied:

reflexivity:	$x \leq x$
transitivity:	$x \leq y \wedge y \leq z \longrightarrow x \leq z$
antisymmetry:	$a \leq b \wedge b \leq a \longrightarrow a = b$
linearity/totality:	$x \leq y \vee y \leq x$

Moreover, the binary predicate $(<)$ must satisfy

$$x < y \longleftrightarrow x \leq y \wedge x \neq y.$$

Note that *linorder* is a specific predefined example of a **type class** [25]. We will not explain type classes any further because we do not require the general concept. In fact, we will mostly not even show the *linorder* annotation in types: you can assume that if you see $\leq$ or $<$ on a generic type *'a* in this book, *'a* is implicitly annotated with *linorder*. Note further that $(\leq)$ on the numeric types *nat*, *int* and *real* is a linear order.

2.1 Specification of Sorting Functions

A sorting function *sort* :: *'a list* ⇒ *'a list* (where, as usual, *'a*::*linorder*) must obviously satisfy the following property:

sorted (*sort xs*)

However, this is not enough — otherwise, *wrong_sort xs* = [] would be a correct sorting function. The set of elements in the output must be the same as in the input, and each element must occur the same number of times. This is most readily captured with the notion of a multiset (see Section 1.2.3). The second property that a sorting function *sort* must satisfy is

mset (*sort xs*) = *mset xs*

where function *mset* converts a list into its corresponding multiset.

2.2 Insertion Sort

Insertion sort is well-known for its intellectual simplicity and computational inefficiency. Its simplicity makes it an ideal starting point for this book. Below, it is implemented by the function *isort* with the help of the auxiliary function *insort* that inserts a single element into an already sorted list.

```
insort :: 'a ⇒ 'a list ⇒ 'a list
insort x [] = [x]
insort x (y # ys) = (if x ≤ y then x # y # ys else y # insort x ys)

isort :: 'a list ⇒ 'a list
isort [] = []
isort (x # xs) = insort x (isort xs)
```

2.2.1 Functional Correctness

We start by proving the preservation of the multiset of elements:

$$\mathit{mset}\ (\mathit{insort}\ x\ \mathit{xs}) = \{\!\{x\}\!\} + \mathit{mset}\ \mathit{xs} \tag{2.1}$$

$$\mathit{mset}\ (\mathit{isort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs} \tag{2.2}$$

Both properties are proved by induction; the proof of (2.2) requires (2.1).

Now we turn to sortedness. Because the definition of *sorted* involves *set*, it is frequently helpful to prove multiset preservation first (as we have done above) because that yields preservation of the set of elements. That is, from (2.1) we obtain:

$$\mathit{set}\ (\mathit{insort}\ x\ \mathit{xs}) = \{x\} \cup \mathit{set}\ \mathit{xs} \tag{2.3}$$

Two inductions prove

$$\mathit{sorted}\ (\mathit{insort}\ a\ \mathit{xs}) = \mathit{sorted}\ \mathit{xs} \tag{2.4}$$

$$\mathit{sorted}\ (\mathit{isort}\ \mathit{xs}) \tag{2.5}$$

where the proof of (2.4) uses (2.3) and the proof of (2.5) uses (2.4).

2.2.2 Running Time Analysis

These are the canonical running time functions (according to Section 1.5):

$$T_{\mathit{insort}} :: {'a} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{insort}}\ \_\ [] = 1$$

$$T_{\mathit{insort}}\ x\ (y\ \#\ \mathit{ys}) = (\textbf{if}\ x \le y\ \textbf{then}\ 0\ \textbf{else}\ T_{\mathit{insort}}\ x\ \mathit{ys}) + 1$$

$$T_{\mathit{isort}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{isort}}\ [] = 1$$

$$T_{\mathit{isort}}\ (x\ \#\ \mathit{xs}) = T_{\mathit{isort}}\ \mathit{xs} + T_{\mathit{insort}}\ x\ (\mathit{isort}\ \mathit{xs}) + 1$$

A dismal quadratic upper bound for the running time of insertion sort is proved readily:

Lemma 2.1. $T_{\mathit{isort}}\ \mathit{xs} \le (|\mathit{xs}| + 1)^2$

Proof. The following properties are proved by induction on xs:

$$T_{\mathit{insort}}\ x\ \mathit{xs} \le |\mathit{xs}| + 1 \tag{2.6}$$

$$|\mathit{insort}\ x\ \mathit{xs}| = |\mathit{xs}| + 1 \tag{2.7}$$

$$|\mathit{isort}\ \mathit{xs}| = |\mathit{xs}| \tag{2.8}$$

The proof of (2.8) needs (2.7). The proof of $T_{\mathit{isort}}\ \mathit{xs} \le (|\mathit{xs}| + 1)^2$ is also by induction on xs. The base case is trivial. The induction step is easy:

$$\begin{array}{ll} T_{isort}\ (x\ \#\ xs) = T_{isort}\ xs + T_{insort}\ x\ (isort\ xs) + 1 & \\ \le (|xs| + 1)^2 + T_{insort}\ x\ (isort\ xs) + 1 & \text{by IH} \\ \le (|xs| + 1)^2 + |xs| + 1 + 1 & \text{using (2.6) and (2.8)} \\ \le (|x\ \#\ xs| + 1)^2 & \square \end{array}$$

Exercise 2.2 asks you to show that *isort* actually has quadratic running time on all lists $[n,\ n{-}1,\ \ldots,\ 0]$.

2.2.3 Exercises

Exercise 2.1. Show that any sorting function has the same input/output behaviour as insertion sort:

$$(\forall xs.\ mset\ (f\ xs) = mset\ xs) \wedge (\forall xs.\ sorted\ (f\ xs)) \longrightarrow$$
$$f\ xs = isort\ xs$$

Exercise 2.2. Show that T_{isort} achieves its optimal value of $2 \cdot n + 1$ for sorted lists, and its worst-case value of $(n + 1) \cdot (n + 2)\ div\ 2$ for the list $rev\ [0..{<}n]$.

2.3 Quicksort

Quicksort is a divide-and-conquer algorithm that sorts a list as follows: pick a **pivot** element from the list; partition the remaining list into those elements that are smaller and those that are greater than the pivot (equal elements can go into either sublist); sort these sublists recursively and append the results. A particularly simple version of this approach, where the first element is chosen as the pivot, looks like this:

```
quicksort :: 'a list ⇒ 'a list
quicksort [] = []
quicksort (x # xs)
= quicksort (filter (λy. y < x) xs) @
  [x] @ quicksort (filter (λy. y ≥ x) xs)
```

2.3.1 Functional Correctness

Preservation of the multiset of elements

$$mset\ (quicksort\ xs) = mset\ xs \tag{2.9}$$

is proved by computation induction using these lemmas:

$$\mathit{mset}\ (\mathit{filter}\ P\ \mathit{xs}) = \mathit{filter\_mset}\ P\ (\mathit{mset}\ \mathit{xs})$$

$$(\forall x.\ P\ x = (\neg\ Q\ x)) \longrightarrow \mathit{filter\_mset}\ P\ M + \mathit{filter\_mset}\ Q\ M = M$$

A second computation induction proves sortedness

$$\mathit{sorted}\ (\mathit{quicksort}\ \mathit{xs})$$

using the lemmas

$$\mathit{sorted}\ (\mathit{xs}\ @\ \mathit{ys})$$
$$= (\mathit{sorted}\ \mathit{xs} \wedge \mathit{sorted}\ \mathit{ys} \wedge (\forall x{\in}\mathit{set}\ \mathit{xs}.\ \forall y{\in}\mathit{set}\ \mathit{ys}.\ x \le y))$$

$$\mathit{set}\ (\mathit{quicksort}\ \mathit{xs}) = \mathit{set}\ \mathit{xs}$$

where the latter one is an easy consequence of (2.9).

We do not analyze the running time of *quicksort*. It is well known that in the worst case it is quadratic (exercise!) but that the average-case running time (in a certain sense) is $O(n \lg n)$. If the pivot is chosen randomly instead of always choosing the first element, the *expected* running time is also $O(n \lg n)$. The necessary probabilistic analysis is beyond the scope of this text but can be found elsewhere [16, 18].

2.3.2 Exercises

Exercise 2.3. Function *quicksort* appends the lists returned from the recursive calls. This is expensive because the running time of (@) is linear in the length of its first argument. Define a function $\mathit{quicksort2} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$ that avoids (@) but accumulates the result in its second parameter via (#) only. Prove $\mathit{quicksort2}\ \mathit{xs}\ \mathit{ys} = \mathit{quicksort}\ \mathit{xs}\ @\ \mathit{ys}$.

Exercise 2.4. There is one obvious optimisation to the version of quicksort that we studied before: instead of partitioning the list into those elements that are smaller than the pivot and those that are at least as big as the pivot, we can use three-way partitioning:

```
partition3 :: 'a ⇒ 'a list ⇒ 'a list × 'a list × 'a list
partition3 x xs
= (filter (λy. y < x) xs, filter (λy. y = x) xs,
   filter (λy. x < y) xs)

quicksort3 :: 'a list ⇒ 'a list
quicksort3 [] = []
```

$$\begin{array}{l} \mathit{quicksort3}\ (x\ \#\ xs) \\ = (\textbf{let}\ (ls,\ es,\ gs) = \mathit{partition3}\ x\ xs \\ \quad \textbf{in}\ \mathit{quicksort3}\ ls\ @\ x\ \#\ es\ @\ \mathit{quicksort3}\ gs) \end{array}$$

Prove that this version of quicksort also produces the correct results.

Exercise 2.5. In this exercise, we will examine the worst-case behaviour of Quicksort, which is e.g. achieved if the input list is already sorted. Consider the time function for Quicksort:

$$\begin{array}{l} T_{\mathit{quicksort}} :: {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\ T_{\mathit{quicksort}}\ [] = 1 \\ T_{\mathit{quicksort}}\ (x\ \#\ xs) = T_{\mathit{quicksort}}\ (\mathit{filter}\ (\lambda y.\ y < x)\ xs) + \\ \qquad\qquad\qquad\qquad\quad T_{\mathit{quicksort}}\ (\mathit{filter}\ (\lambda y.\ y \ge x)\ xs) + \\ \qquad\qquad\qquad\qquad\quad 2 \cdot T_{\mathit{filter}}\ (\lambda\_.\ 1)\ xs + 1 \end{array}$$

1. Show that Quicksort takes quadratic time on sorted lists by proving

   $$\mathit{sorted}\ xs \longrightarrow T_{\mathit{quicksort}}\ xs = a \cdot |xs|^2 + b \cdot |xs| + c$$

   for suitable values a, b, c.
2. Show that this is the worst-case running time by proving

   $$T_{\mathit{quicksort}}\ xs \le a \cdot |xs|^2 + b \cdot |xs| + c$$

   for the values of a, b, c you determined in the previous step.

2.4 Top-Down Merge Sort

Merge sort is another prime example of a divide-and-conquer algorithm, and one whose running time is guaranteed to be $O(n \lg n)$. We will consider three variants and start with the simplest one: split the list into two halves, sort the halves separately and merge the results.

$$\begin{array}{l} \mathit{merge} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \\ \mathit{merge}\ []\ ys = ys \end{array}$$

```
merge xs [] = xs
merge (x # xs) (y # ys)
= (if x ≤ y then x # merge xs (y # ys)
   else y # merge (x # xs) ys)

msort :: 'a list ⇒ 'a list
msort xs
= (let n = |xs|
   in if n ≤ 1 then xs
      else merge (msort (take (n div 2) xs))
                 (msort (drop (n div 2) xs)))
```

2.4.1 Functional Correctness

We start off with multisets and sets of elements:

$$\mathit{mset}\ (\mathit{merge}\ \mathit{xs}\ \mathit{ys}) = \mathit{mset}\ \mathit{xs} + \mathit{mset}\ \mathit{ys} \tag{2.10}$$

$$\mathit{set}\ (\mathit{merge}\ \mathit{xs}\ \mathit{ys}) = \mathit{set}\ \mathit{xs} \cup \mathit{set}\ \mathit{ys} \tag{2.11}$$

Proposition (2.10) is proved by induction on the computation of *merge* and (2.11) is an easy consequence.

Lemma 2.2. $\mathit{mset}\ (\mathit{msort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$

Proof. The proof is by induction on the computation of *msort*. Let $n = |\mathit{xs}|$. The base case ($n \leq 1$) is trivial. Now assume $n > 1$ and let $\mathit{ys} = \mathit{take}\ (n\ \mathit{div}\ 2)\ \mathit{xs}$ and $\mathit{zs} = \mathit{drop}\ (n\ \mathit{div}\ 2)\ \mathit{xs}$.

$$\begin{aligned}
&\mathit{mset}\ (\mathit{msort}\ \mathit{xs}) = \mathit{mset}\ (\mathit{msort}\ \mathit{ys}) + \mathit{mset}\ (\mathit{msort}\ \mathit{zs}) && \text{by (2.10)}\\
&= \mathit{mset}\ \mathit{ys} + \mathit{mset}\ \mathit{zs} && \text{by IH}\\
&= \mathit{mset}\ (\mathit{ys}\ @\ \mathit{zs})\\
&= \mathit{mset}\ \mathit{xs} && \square
\end{aligned}$$

Now we turn to sortedness. An induction on the computation of *merge*, using (2.11), yields

$$\mathit{sorted}\ (\mathit{merge}\ \mathit{xs}\ \mathit{ys}) = (\mathit{sorted}\ \mathit{xs} \wedge \mathit{sorted}\ \mathit{ys}) \tag{2.12}$$

Lemma 2.3. $\mathit{sorted}\ (\mathit{msort}\ \mathit{xs})$

The proof is an easy induction on the computation of *msort*. The base case ($n \leq 1$) follows because every list of length ≤ 1 is sorted. The induction step follows with the help of (2.12).

2.4.2 Running Time Analysis

To simplify the analysis, and in line with the literature, we only count the number of comparisons:

```
C_merge :: 'a list ⇒ 'a list ⇒ nat
C_merge [] _ = 0
C_merge _ [] = 0
C_merge (x # xs) (y # ys)
= 1 + (if x ≤ y then C_merge xs (y # ys) else C_merge (x # xs) ys)

C_msort :: 'a list ⇒ nat
C_msort xs
= (let n = |xs|;
       ys = take (n div 2) xs;
       zs = drop (n div 2) xs
   in if n ≤ 1 then 0
      else C_msort ys + C_msort zs + C_merge (msort ys) (msort zs))
```

By computation inductions we obtain:

$$|merge\ xs\ ys| = |xs| + |ys| \tag{2.13}$$

$$|msort\ xs| = |xs| \tag{2.14}$$

$$C_{merge}\ xs\ ys \leq |xs| + |ys| \tag{2.15}$$

where the proof of (2.14) uses (2.13).

To simplify technicalities, we prove the $n \cdot \lg\ n$ bound on the number of comparisons in *msort* only for $n = 2^k$, in which case the bound becomes $k \cdot 2^k$.

Lemma 2.4. $|xs| = 2^k \longrightarrow C_{msort}\ xs \leq k \cdot 2^k$

Proof. The proof is by induction on k. The base case is trivial and we concentrate on the step. Let $n = |xs|$, $ys = take\ (n\ div\ 2)\ xs$ and $zs = drop\ (n\ div\ 2)\ xs$. The case $n \leq 1$ is trivial. Now assume $n > 1$.

$$\begin{aligned}
& C_{msort}\ xs \\
&= C_{msort}\ ys + C_{msort}\ zs + C_{merge}\ (msort\ ys)\ (msort\ zs) \\
&\leq C_{msort}\ ys + C_{msort}\ zs + |ys| + |zs| && \text{using (2.15) and (2.14)} \\
&\leq k \cdot 2^k + k \cdot 2^k + |ys| + |zs| && \text{by IH} \\
&= k \cdot 2^k + k \cdot 2^k + |xs| \\
&= (k + 1) \cdot 2^{k + 1} && \text{by assumption } |xs| = 2^{k + 1}
\end{aligned}$$

□

Exercise 2.6. The definition of *msort* is inefficient in that it calls *length*, *take* and *drop* for each list. Instead we can split the list into two halves by traversing it only once and putting its elements alternately on two piles, for example *halve* [2, 3, 4] ([0], [1]) = ([4, 2, 0], [3, 1]). Define *halve* and *msort2*

```
msort2 [] = []
msort2 [x] = [x]
msort2 xs
= (let (ys1, ys2) = halve xs ([], []) in merge (msort2 ys1) (msort2 ys2))
```

and prove *mset* (*msort2 xs*) = *mset xs* and *sorted* (*msort2 xs*). (Hint for Isabelle users: The definition of *msort2* is tricky because its termination relies on suitable properties of *halve*.)

2.5 Bottom-Up Merge Sort

Bottom-up merge sort starts by turning the input $[x_1, \ldots, x_n]$ into the list $[[x_1], \ldots, [x_n]]$. Then it passes over this list of lists repeatedly, merging pairs of adjacent lists on every pass until at most one list is left.

```
merge_adj :: 'a list list ⇒ 'a list list
merge_adj [] = []
merge_adj [xs] = [xs]
merge_adj (xs # ys # zss) = merge xs ys # merge_adj zss

merge_all :: 'a list list ⇒ 'a list
merge_all [] = []
merge_all [xs] = xs
merge_all xss = merge_all (merge_adj xss)

msort_bu :: 'a list ⇒ 'a list
msort_bu xs = merge_all (map (λx. [x]) xs)
```

Termination of *merge_all* relies on the fact that *merge_adj* halves the length of the list (rounding up). Computation induction proves

$$|\mathit{merge\_adj}\ \mathit{xs}| = (|\mathit{xs}| + 1)\ \mathit{div}\ 2 \tag{2.16}$$

2.5.1 Functional Correctness

We introduce the abbreviation *mset_mset* :: *'a list list* ⇒ *'a multiset*:

$$\mathit{mset\_mset}\ \mathit{xss} \equiv \sum\nolimits_{\#}\ (\mathit{image\_mset}\ \mathit{mset}\ (\mathit{mset}\ \mathit{xss}))$$

These are the key properties of the functions involved:

$$\mathit{mset\_mset}\ (\mathit{merge\_adj}\ \mathit{xss}) = \mathit{mset\_mset}\ \mathit{xss}$$

$$\mathit{mset}\ (\mathit{merge\_all}\ \mathit{xss}) = \mathit{mset\_mset}\ \mathit{xss} \tag{2.17}$$

$$\mathit{mset}\ (\mathit{msort\_bu}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$$

$$(\forall\, \mathit{xs}{\in}\mathit{set}\ \mathit{xss}.\ \mathit{sorted}\ \mathit{xs}) \longrightarrow (\forall\, \mathit{xs}{\in}\mathit{set}\ (\mathit{merge\_adj}\ \mathit{xss}).\ \mathit{sorted}\ \mathit{xs})$$

$$(\forall\, \mathit{xs}{\in}\mathit{set}\ \mathit{xss}.\ \mathit{sorted}\ \mathit{xs}) \longrightarrow \mathit{sorted}\ (\mathit{merge\_all}\ \mathit{xss}) \tag{2.18}$$

$$\mathit{sorted}\ (\mathit{msort\_bu}\ \mathit{xs})$$

The third and the last property prove functional correctness of *msort_bu*. The proof of each proposition may use the preceding proposition and the propositions (2.10) and (2.12). The propositions about *merge_adj* and *merge_all* are proved by computation inductions.

2.5.2 Running Time Analysis

Again, we count only comparisons:

$$C_{\mathit{merge\_adj}} :: {}'a\ \mathit{list}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&C_{\mathit{merge\_adj}}\ [] = 0\\
&C_{\mathit{merge\_adj}}\ [\_] = 0\\
&C_{\mathit{merge\_adj}}\ (\mathit{xs}\ \#\ \mathit{ys}\ \#\ \mathit{zss}) = C_{\mathit{merge}}\ \mathit{xs}\ \mathit{ys} + C_{\mathit{merge\_adj}}\ \mathit{zss}
\end{aligned}$$

$$C_{\mathit{merge\_all}} :: {}'a\ \mathit{list}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&C_{\mathit{merge\_all}}\ [] = 0\\
&C_{\mathit{merge\_all}}\ [\mathit{xs}] = 0\\
&C_{\mathit{merge\_all}}\ \mathit{xss} = C_{\mathit{merge\_adj}}\ \mathit{xss} + C_{\mathit{merge\_all}}\ (\mathit{merge\_adj}\ \mathit{xss})
\end{aligned}$$

$$C_{\mathit{msort\_bu}} :: {}'a\ \mathit{list} \Rightarrow \mathit{nat}$$

$$C_{\mathit{msort\_bu}}\ \mathit{xs} = C_{\mathit{merge\_all}}\ (\mathit{map}\ (\lambda x.\ [x])\ \mathit{xs})$$

By simple computation inductions we obtain:

$$\begin{aligned}
&\mathit{even}\ |\mathit{xss}| \wedge (\forall\, \mathit{xs}{\in}\mathit{set}\ \mathit{xss}.\ |\mathit{xs}| = m) \longrightarrow\\
&(\forall\, \mathit{xs}{\in}\mathit{set}\ (\mathit{merge\_adj}\ \mathit{xss}).\ |\mathit{xs}| = 2 \cdot m)
\end{aligned} \tag{2.19}$$

$$(\forall\, \mathit{xs}{\in}\mathit{set}\ \mathit{xss}.\ |\mathit{xs}| = m) \longrightarrow C_{\mathit{merge\_adj}}\ \mathit{xss} \le m \cdot |\mathit{xss}| \tag{2.20}$$

using (2.13) for (2.19) and (2.15) for (2.20).

Lemma 2.5. $(\forall\, xs \in set\ xss.\ |xs| = m) \wedge |xss| = 2^k \longrightarrow$
$C_{merge\_all}\ xss \le m \cdot k \cdot 2^k$

Proof by induction on the computation of *merge_all*. We concentrate on the nontrivial recursive case arising from the third equation. We assume $|xss| > 1$, $\forall\, xs \in set\ xss.\ |xs| = m$ and $|xss| = 2^k$. Clearly $k \ge 1$ and thus $even\ |xss|$. Thus (2.19) implies $\forall\, xs \in set\ (merge\_adj\ xss).\ |xs| = 2 \cdot m$. Also note

$$\begin{array}{ll}
|merge\_adj\ xss| & \\
= (|xss| + 1)\ div\ 2 & \text{using (2.16)} \\
= 2^{k - 1} & \text{using } |xss| = 2^k \text{ and } k \ge 1 \text{ by arithmetic}
\end{array}$$

Let $yss = merge\_adj\ xss$. We can now prove the lemma:

$$\begin{array}{ll}
C_{merge\_all}\ xss = C_{merge\_adj}\ xss + C_{merge\_all}\ yss & \\
\le m \cdot 2^k + C_{merge\_all}\ yss & \text{using } |xss| = 2^k \text{ and (2.20)} \\
\le m \cdot 2^k + 2 \cdot m \cdot (k - 1) \cdot 2^{k - 1} & \\
\quad \text{by IH using } \forall\, xs \in set\ yss.\ |xs| = 2 \cdot m \text{ and } |yss| = 2^{k - 1} & \\
= m \cdot k \cdot 2^k & \square
\end{array}$$

Setting $m = 1$ we obtain the same upper bound as for top-down merge sort in Lemma 2.4:

Corollary 2.6. $|xs| = 2^k \longrightarrow C_{msort\_bu}\ xs \le k \cdot 2^k$

2.6 Natural Merge Sort

A disadvantage of all the sorting functions we have seen so far (except insertion sort) is that even in the best case they do not improve upon the $n \cdot \lg\ n$ bound. For example, given the sorted input [1, 2, 3, 4, 5], *msort_bu* will, as a first step, create [[1], [2], [3], [4], [5]] and then merge this list of lists recursively.

A slight variation of bottom-up merge sort, sometimes referred to as "natural merge sort," first partitions the input into its constituent ascending and descending subsequences (collectively referred to as **runs**) and only then starts merging. In the above example we would get *merge_all* [[1, 2, 3, 4, 5]], which returns immediately with the result [1, 2, 3, 4, 5]. Assuming that obtaining runs is of linear complexity, this yields a best-case performance that is linear in the number of list elements.

Function *runs* computes the initial list of lists; it is defined mutually recursively with *asc* and *desc*, which gather ascending and descending runs in accumulating parameters:

```
runs :: 'a list ⇒ 'a list list
runs (a # b # xs)
= (if b < a then desc b [a] xs else asc b ((#) a) xs)
runs [x] = [[x]]
runs [] = []

asc :: 'a ⇒ ('a list ⇒ 'a list) ⇒ 'a list ⇒ 'a list list
asc a as (b # bs)
= (if ¬ b < a then asc b (as ∘ (#) a) bs else as [a] # runs (b # bs))
asc a as [] = [as [a]]

desc :: 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list list
desc a as (b # bs)
= (if b < a then desc b (a # as) bs else (a # as) # runs (b # bs))
desc a as [] = [a # as]
```

Function *desc* needs to reverse the descending run it collects. Therefore a natural choice for the type of its accumulator *as* is *list*, since recursively prepending elements (using (#)) ultimately yields a reversed list.

Function *asc* collects an ascending run and is slightly more complicated than *desc*. If we used lists, we could accumulate the elements similarly to *desc* but using *as* @ [*a*] instead of *a* # *as*. This would take quadratic time in the number of appended elements. Therefore the "standard" solution is to accumulate elements using (#) and to reverse the accumulator in linear time (as shown in Section 1.5.1) at the end. However, another interesting option (that yields better performance for some functional languages, like Haskell) is to use **difference lists**. This is the option we chose for *asc*.

In the functional programming world, difference lists are a well-known trick to append lists in constant time by representing lists as functions of type *'a list* ⇒ *'a list*. For difference lists, we have the following correspondences: empty list [] $\approx \lambda x.\ x$, singleton list $[x] \approx (\#)\ x$, and list append xs @ $ys \approx xs \circ ys$ (taking constant time). Moreover, transforming a difference list *xs* into a normal list is as easy as *xs* [] (taking linear time).

Note that, due to the mutually recursive definitions of *runs*, *asc*, and *desc*, whenever we prove a property of *runs*, we simultaneously have to prove suitable properties of *asc* and *desc* using mutual induction.

Natural merge sort is the composition of *merge_all* and *runs*:

$$nmsort :: {'a}\ list \Rightarrow {'a}\ list$$
$$nmsort\ xs = merge\_all\ (runs\ xs)$$

2.6.1 Functional Correctness

We have

$$\begin{aligned}&(\forall\, xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys) \longrightarrow\\ &mset\_mset\ (asc\ x\ f\ ys) = \{\!\!\{x\}\!\!\} + mset\ (f\ [\,])\ + mset\ ys\end{aligned} \tag{2.21}$$

$$mset\_mset\ (desc\ x\ xs\ ys) = \{\!\!\{x\}\!\!\} + mset\ xs + mset\ ys \tag{2.22}$$

$$mset\_mset\ (runs\ xs) = mset\ xs \tag{2.23}$$

$$mset\ (nmsort\ xs) = mset\ xs \tag{2.24}$$

where (2.23), (2.21), and (2.22) are proved simultaneously. The assumption of (2.21) on f ensures that f is a difference list. We use (2.23) together with (2.17) in order to show (2.24). Moreover, we have

$$\forall\, x {\in} set\ (runs\ xs).\ sorted\ x \tag{2.25}$$

$$sorted\ (nmsort\ xs) \tag{2.26}$$

where we use (2.25) together with (2.18) to obtain (2.26).

2.6.2 Running Time Analysis

Once more, we only count comparisons:

$$C_{runs} :: {'a}\ list \Rightarrow nat$$
$$\begin{aligned}&C_{runs}\ (a\ \#\ b\ \#\ xs) = 1 + (\textbf{if}\ b < a\ \textbf{then}\ C_{desc}\ b\ xs\ \textbf{else}\ C_{asc}\ b\ xs)\\ &C_{runs}\ [\,] = 0\\ &C_{runs}\ [\_] = 0\end{aligned}$$

$$C_{asc} :: {'a} \Rightarrow {'a}\ list \Rightarrow nat$$
$$\begin{aligned}&C_{asc}\ a\ (b\ \#\ bs) = 1 + (\textbf{if}\ \neg\ b < a\ \textbf{then}\ C_{asc}\ b\ bs\ \textbf{else}\ C_{runs}\ (b\ \#\ bs))\\ &C_{asc}\ \_\ [\,] = 0\end{aligned}$$

$$C_{desc} :: {'a} \Rightarrow {'a}\ list \Rightarrow nat$$
$$\begin{aligned}&C_{desc}\ a\ (b\ \#\ bs) = 1 + (\textbf{if}\ b < a\ \textbf{then}\ C_{desc}\ b\ bs\ \textbf{else}\ C_{runs}\ (b\ \#\ bs))\\ &C_{desc}\ \_\ [\,] = 0\end{aligned}$$

$$C_{nmsort} :: {}'a\ list \Rightarrow nat$$
$$C_{nmsort}\ xs = C_{runs}\ xs + C_{merge\_all}\ (runs\ xs)$$

Again note the mutually recursive definitions of C_{runs}, C_{asc}, and C_{desc}. Hence the remark on proofs about *runs* also applies to proofs about C_{runs}.

Before talking about C_{nmsort}, we need a variant of Lemma 2.5 that also works for lists whose lengths are not powers of two (since the result of *runs* will usually not satisfy this property).

To this end, we will need the following two results, which we prove by two simple computation inductions using (2.15) and (2.13):

$$C_{merge\_adj}\ xss \le |concat\ xss| \tag{2.27}$$

$$|concat\ (merge\_adj\ xss)| = |concat\ xss| \tag{2.28}$$

Lemma 2.7. $C_{merge\_all}\ xss \le |concat\ xss| \cdot \lceil \lg |xss| \rceil$

Proof by induction on the computation of C_{merge_all}. We concentrate on the nontrivial recursive case arising from the third equation. It follows that xss is of the form $xs\ \#\ ys\ \#\ zss$. Further note that for all $n :: nat$:

$$2 \le n \longrightarrow \lceil \lg n \rceil = \lceil \lg ((n - 1)\ div\ 2 + 1) \rceil + 1 \tag{2.29}$$

Now, let $m = |concat\ xss|$. Then we have

$$\begin{array}{ll}
C_{merge\_all}\ xss & \\
= C_{merge\_adj}\ xss + C_{merge\_all}\ (merge\_adj\ xss) & \\
\le m + C_{merge\_all}\ (merge\_adj\ xss) & \text{using (2.27)} \\
\le m + |concat\ (merge\_adj\ xss)| \cdot \lceil \lg |merge\_adj\ xss| \rceil & \text{by IH} \\
= m + m \cdot \lceil \lg |merge\_adj\ xss| \rceil & \text{by (2.28)} \\
= m + m \cdot \lceil \lg ((|xss| + 1)\ div\ 2) \rceil & \text{by (2.16)} \\
= m + m \cdot \lceil \lg ((|zss| + 1)\ div\ 2 + 1) \rceil & \\
= m \cdot (\lceil \lg ((|zss| + 1)\ div\ 2 + 1) \rceil + 1) & \\
= m \cdot \lceil \lg (|zss| + 2) \rceil & \text{by (2.29)} \\
= m \cdot \lceil \lg |xss| \rceil & \square
\end{array}$$

Two simple computation inductions, each performed simultaneously for the corresponding mutually recursive definitions, yield:

$$(\forall xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys) \longrightarrow |asc\ a\ f\ ys| \le 1 + |ys|$$

$$|desc\ a\ xs\ ys| \le 1 + |ys|$$

$$|runs\ xs| \le |xs| \tag{2.30}$$

$$C_{asc}\ a\ ys \le |ys|$$

$$C_{desc}\ a\ ys \le |ys|$$

$$C_{runs}\ xs \le |xs| - 1 \tag{2.31}$$

At this point we obtain an upper bound on the number of comparisons required by C_{nmsort}.

Lemma 2.8. $|xs| = n \longrightarrow C_{nmsort}\ xs \leq n + n \cdot \lceil \lg\ n \rceil$

Proof. Note that

$$C_{merge\_all}\ (runs\ xs) \leq n \cdot \lceil \lg\ n \rceil \qquad (\star)$$

as shown by the derivation:

$$\begin{aligned} & C_{merge\_all}\ (runs\ xs) \\ & \leq n \cdot \lceil \lg\ |runs\ xs| \rceil && \text{by Lemma 2.7 with } xss = runs\ xs \\ & \leq n \cdot \lceil \lg\ n \rceil && \text{by (2.30)} \end{aligned}$$

We conclude the proof by:

$$\begin{aligned} & C_{nmsort}\ xs = C_{runs}\ xs + C_{merge\_all}\ (runs\ xs) \\ & \leq n + n \cdot \lceil \lg\ n \rceil && \text{using (2.31) and } (\star) \end{aligned}$$ □

2.7 Stability

A sorting function is called **stable** if the order of equal elements is preserved. However, this only makes a difference if elements are not identified with their keys, as we have done so far. Let us assume instead that sorting is parameterized with a key function $f :: 'a \Rightarrow 'k$ that maps an element to its key and that the keys $'k$ are linearly ordered, not the elements. This is the specification of a sorting function *sort_key*:

$$mset\ (sort\_key\ f\ xs) = mset\ xs$$
$$sorted\ (map\ f\ (sort\_key\ f\ xs))$$

Assuming (for simplicity) we are sorting pairs of keys and some attached information, stability means that sorting $[(2,\ x),\ (1,\ z),\ (1,\ y)]$ yields $[(1,\ z),\ (1,\ y),\ (2,\ x)]$ and not $[(1,\ y),\ (1,\ z),\ (2,\ x)]$. That is, if we extract all elements with the same key *after* sorting *xs*, they should be in the same order as in *xs*:

$$filter\ (\lambda y.\ f\ y = k)\ (sort\_key\ f\ xs) = filter\ (\lambda y.\ f\ y = k)\ xs$$

We will now define insertion sort adapted to keys and verify its correctness and stability.

$$\mathit{insort\_key} :: ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b\ \mathit{list} \Rightarrow 'b\ \mathit{list}$$

$$\begin{aligned}
&\mathit{insort\_key}\ \_\ \ x\ [] = [x]\\
&\mathit{insort\_key}\ f\ x\ (y\ \#\ \mathit{ys})\\
&= (\textbf{if}\ f\ x \le f\ y\ \textbf{then}\ x\ \#\ y\ \#\ \mathit{ys}\ \textbf{else}\ y\ \#\ \mathit{insort\_key}\ f\ x\ \mathit{ys})
\end{aligned}$$

$$\mathit{isort\_key} :: ('a \Rightarrow 'k) \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$$

$$\begin{aligned}
&\mathit{isort\_key}\ \_\ \ [] = []\\
&\mathit{isort\_key}\ f\ (x\ \#\ \mathit{xs}) = \mathit{insort\_key}\ f\ x\ (\mathit{isort\_key}\ f\ \mathit{xs})
\end{aligned}$$

The proofs of the functional correctness properties

$$\begin{aligned}
&\mathit{mset}\ (\mathit{isort\_key}\ f\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}\\
&\mathit{sorted}\ (\mathit{map}\ f\ (\mathit{isort\_key}\ f\ \mathit{xs})) \qquad (2.32)
\end{aligned}$$

are completely analogous to their counterparts for plain *isort*.

The proof of stability uses three auxiliary properties:

$$(\forall x{\in}\mathit{set}\ \mathit{xs}.\ f\ a \le f\ x) \longrightarrow \mathit{insort\_key}\ f\ a\ \mathit{xs} = a\ \#\ \mathit{xs} \qquad (2.33)$$

$$\neg\ P\ x \longrightarrow \mathit{filter}\ P\ (\mathit{insort\_key}\ f\ x\ \mathit{xs}) = \mathit{filter}\ P\ \mathit{xs} \qquad (2.34)$$

$$\begin{aligned}
&\mathit{sorted}\ (\mathit{map}\ f\ \mathit{xs}) \wedge P\ x \longrightarrow\\
&\mathit{filter}\ P\ (\mathit{insort\_key}\ f\ x\ \mathit{xs}) = \mathit{insort\_key}\ f\ x\ (\mathit{filter}\ P\ \mathit{xs}) \qquad (2.35)
\end{aligned}$$

The first one is proved by a case analysis on *xs*. The other two are proved by induction on *xs*, using (2.33) in the proof of (2.35).

Lemma 2.9 (Stability of *isort_key*).
$\mathit{filter}\ (\lambda y.\ f\ y = k)\ (\mathit{isort\_key}\ f\ \mathit{xs}) = \mathit{filter}\ (\lambda y.\ f\ y = k)\ \mathit{xs}$

Proof. The proof is by induction on *xs*. The base case is trivial. In the induction step we consider the list $a\ \#\ \mathit{xs}$ and perform a case analysis. If $f\ a \ne k$ the claim follows by IH using (2.34). Now assume $f\ a = k$:

$$\begin{aligned}
&\mathit{filter}\ (\lambda y.\ f\ y = k)\ (\mathit{isort\_key}\ f\ (a\ \#\ \mathit{xs}))\\
&= \mathit{filter}\ (\lambda y.\ f\ y = k)\ (\mathit{insort\_key}\ f\ a\ (\mathit{isort\_key}\ f\ \mathit{xs}))\\
&= \mathit{insort\_key}\ f\ a\ (\mathit{filter}\ (\lambda y.\ f\ y = k)\ (\mathit{isort\_key}\ f\ \mathit{xs}))\\
&\qquad\qquad \text{using } f\ a = k\text{, (2.35), (2.32)}\\
&= \mathit{insort\_key}\ f\ a\ (\mathit{filter}\ (\lambda y.\ f\ y = k)\ \mathit{xs}) \qquad \text{by IH}\\
&= a\ \#\ \mathit{filter}\ (\lambda y.\ f\ y = k)\ \mathit{xs} \qquad \text{using } f\ a = k \text{ and (2.33)}\\
&= \mathit{filter}\ (\lambda y.\ f\ y = k)\ (a\ \#\ \mathit{xs}) \qquad \text{using } f\ a = k \quad \square
\end{aligned}$$

As exercises we recommend to adapt some of the other sorting algorithms above to sorting with keys and to prove their correctness and stability.

3

Selection

A topic that is somewhat related to that of sorting is *selection*: given a list xs of length n with some linear ordering defined on it and a natural number $k < n$, return the k-th smallest number in the list. If xs is sorted, this is exactly the k-th element of the list.

The defining properties of the selection operation are as follows:

$$\begin{aligned} k < |xs| \longrightarrow\ & |\{\!\{y \in_\# mset\ xs \mid y < select\ k\ xs\}\!\}| \le k \\ & \wedge\ |\{\!\{y \in_\# mset\ xs \mid y > select\ k\ xs\}\!\}| < |xs| - k \end{aligned} \tag{3.1}$$

These properties uniquely define the selection property, as shown by the following theorem:

Theorem 3.1 (Uniqueness of the selection operation).
If $k < |xs|$ *and*

$$\begin{array}{ll} |\{\!\{z \in_\# mset\ xs \mid z < x\}\!\}| \le k & |\{\!\{z \in_\# mset\ xs \mid z > x\}\!\}| < |xs| - k \\ |\{\!\{z \in_\# mset\ xs \mid z < y\}\!\}| \le k & |\{\!\{z \in_\# mset\ xs \mid z > y\}\!\}| < |xs| - k \end{array}$$

then $x = y$.

Proof. Suppose $x \neq y$ and then w.l.o.g. $x < y$. This implies:

$$\{\!\{z \in_\# mset\ xs \mid z \le x\}\!\} \subseteq_\# \{\!\{z \in_\# mset\ xs \mid z < y\}\!\} \tag{3.2}$$

From this we can prove the contradiction $|xs| < |xs|$:

$$\begin{aligned} |xs| &= |\{\!\{z \in_\# mset\ xs \mid z \le x\}\!\}| + |\{\!\{z \in_\# mset\ xs \mid x < z\}\!\}| \\ &\le |\{\!\{z \in_\# mset\ xs \mid z < y\}\!\}| + |\{\!\{z \in_\# mset\ xs \mid x < z\}\!\}| && \text{using (3.2)} \\ &< k + (|xs| - k) && \text{using (3.1)} \\ &= |xs| \end{aligned}$$

□

An equivalent, more concrete definition is the following:

$$select :: nat \Rightarrow ('a{::}linorder)\ list \Rightarrow 'a$$

$$select\ k\ xs = sort\ xs\ !\ k \tag{3.3}$$

Theorem 3.2. *Our definition of select satisfies the conditions* (3.1).

Proof. If ys is a sorted list, the following can be proved by a straightforward induction on ys:

$$\{\!\{x \in_{\#} \mathit{mset}\ ys \mid x < ys\ !\ k\}\!\} \subseteq_{\#} \mathit{mset}\ (\mathit{take}\ k\ ys)$$
$$\{\!\{x \in_{\#} \mathit{mset}\ ys \mid x > ys\ !\ k\}\!\} \subseteq_{\#} \mathit{mset}\ (\mathit{drop}\ (k + 1)\ ys)$$

Taking the size of these multisets on both sides, we obtain:

$$|\{\!\{x \in_{\#} \mathit{mset}\ ys \mid x < ys\ !\ k\}\!\}| \le k$$
$$|\{\!\{x \in_{\#} \mathit{mset}\ ys \mid x > ys\ !\ k\}\!\}| < |ys| - k$$

By setting $ys := \mathit{sort}\ ys$ here, we have

$$\begin{aligned} & k \ge |\{\!\{x \in_{\#} \mathit{mset}\ (\mathit{sort}\ xs) \mid x < \mathit{sort}\ xs\ !\ k\}\!\}| \\ & = |\{\!\{x \in_{\#} \mathit{mset}\ xs \mid x < \mathit{sort}\ xs\ !\ k\}\!\}| \quad \text{using } \mathit{mset}\ (\mathit{sort}\ xs) = \mathit{mset}\ xs \\ & = |\{\!\{x \in_{\#} \mathit{mset}\ xs \mid x < \mathit{select}\ k\ xs\}\!\}| \quad \text{using (3.3)} \end{aligned}$$

and analogously for the elements greater than *select k xs*. □

We will frequently need two more important lemmas about *sort* and *select*, namely that they are invariant under permutation of the input list:

$$\mathit{mset}\ xs = \mathit{mset}\ ys \longrightarrow \mathit{sort}\ xs = \mathit{sort}\ ys \tag{3.4}$$

$$\mathit{mset}\ xs = \mathit{mset}\ ys \longrightarrow \mathit{select}\ k\ xs = \mathit{select}\ k\ ys \tag{3.5}$$

Equation (3.4) follows directly from the uniqueness of the *sort* operation, and (3.5) then follows from (3.4) and our definition of *select*.

The definition using $\mathit{sort}\ xs\ !\ k$ already gives us a straightforward $O(n \lg n)$ algorithm for the selection operation: sort the list with one of our $O(n \lg n)$ sorting algorithms and then return the k-th element of the resulting sorted list. It is also fairly easy to come up with an algorithm that has running time $O(kn)$, i.e. that runs in linear-time in n for any fixed k (see Exercise 3.3).

In the remainder of this chapter, we will look at a selection algorithm due to Blum *et al.* [10] that achieves $O(n)$ running time *unconditionally* (i.e. for any $k < n$). Since a selection algorithm must, in general, inspect each element at least once (see Exercise 3.4), this running time is asymptotically optimal.

3.0.1 Exercises

Exercise 3.1. A simple special case of selection is *select* 0 *xs*, i.e. the minimum. Implement a linear-time function *select0* such that

$$xs \ne [] \longrightarrow \mathit{select0}\ xs = \mathit{select}\ 0\ xs$$

and prove this. This function should be tail-recursive and traverse the list exactly once. You need not prove the linear running time (it should be obvious).

Exercise 3.2. How can your *select0* algorithm be modified to obtain an analogous algorithm *select1* such that

$$1 < |xs| \longrightarrow \mathit{select1}\ xs = \mathit{select}\ 1\ xs$$

Do not try to prove the correctness yet; it gets somewhat tedious and you will be able to prove it more easily after the next exercise.

Exercise 3.3.

1. Based on the previous two exercises, implement and prove correct an algorithm *select_fixed* that fulfils
   $$k < |xs| \longrightarrow \mathit{select\_fixed}\ k\ xs = \mathit{select}\ k\ xs$$
   The algorithm must be tail-recursive with running time $O(kn)$ and traverse the list exactly once.
   Hint: one approach is to first define a function *take_sort* that computes *take* m (*sort* xs) in time $O(mn)$.
2. Prove your *select1* from the previous exercise correct by showing that it is equivalent to *select_fixed* 1.
3. Define a suitable time function for your *select_fixed*. Prove that this time function is $O(kn)$, i.e. that
   $$T_{\mathit{select\_fixed}}\ k\ xs \leq C_1 \cdot k \cdot |xs| + C_2 \cdot |xs| + C_3 \cdot k + C_4$$
   for all $k < |xs|$ for some constants C_1 to C_4.
   If you have trouble finding the concrete values for these constants, try proving the result with symbolic constants first and observe what conditions need to be fulfilled in order to make the induction step go through.

Exercise 3.4. Show that if xs is a list of integers with no repeated elements, an algorithm computing the result of *select* k xs must examine every single element, i.e. for any index $i < |xs|$, the i-th element can be replaced by some other number such that the result changes. Formally:

$$k < |xs| \land i < |xs| \land \mathit{distinct}\ xs \longrightarrow$$
$$(\exists\, z.\ \mathit{distinct}\ (xs[i := z]) \land \mathit{select}\ k\ (xs[i := z]) \neq \mathit{select}\ k\ xs)$$

Here, the notation $xs[i := z]$ denotes the list xs where the i-th element has been replaced with z (the first list element, as always, having index 0).

Hint: a lemma you might find useful is that $\lambda k.\ \mathit{select}\ k\ xs$ is injective if xs has no repeated elements.

3.1 A Divide-and-Conquer Approach

As a first step in our attempt to derive an efficient algorithm for selection, recall what we did with the function *partition*3 in the threeway quicksort algorithm in Exercise 2.4: we picked some pivot x from xs and partitioned the input list xs into the sublists ls, es, and gs that were smaller, equal, and greater than x, respectively.

If we do the same for *select* k xs, there are three possible cases:

- If $k < |ls|$, the element we are looking for is located in ls. To be more precise, it is the k-th smallest element of ls, i.e. *select* k ls.
- If $k < |ls| + |es|$, the element we are looking for is located in es and must therefore be x itself.
- Otherwise, the element we are looking for must be located in gs. More precisely, it is the k'-th smallest element of gs where $k' = k - |ls| - |es|$.

This gives us a straightforward recursive divide-and-conquer algorithm for selection. To prove this correct, we first prove the following lemma about *select*:

$$\begin{aligned}&k < |ys| + |zs| \wedge (\forall y{\in}set\ ys.\ \forall z{\in}set\ zs.\ y \le z) \longrightarrow\\ &select\ k\ (ys\ @\ zs)\\ &= (\textbf{if}\ k < |ys|\ \textbf{then}\ select\ k\ ys\ \textbf{else}\ select\ (k - |ys|)\ zs)\end{aligned} \tag{3.6}$$

Proof. The assumptions imply that *sort xs* @ *sort ys* is sorted, so that due to the uniqueness of the *sort* operation, we have:

$$sort\ (xs\ @\ ys) = sort\ xs\ @\ sort\ ys \tag{3.7}$$

Then:

$$\begin{aligned}&select\ k\ (xs\ @\ ys)\\ &= sort\ (xs\ @\ ys)\ !\ k && \text{using (3.3)}\\ &= (sort\ xs\ @\ sort\ ys)\ !\ k && \text{using (3.7)}\\ &= \textbf{if}\ k < |xs|\ \textbf{then}\ sort\ xs\ !\ k\ \textbf{else}\ sort\ ys\ !\ (k - |xs|)\\ &= \textbf{if}\ k < |xs|\ \textbf{then}\ select\ k\ xs\ \textbf{else}\ select\ (k - |xs|)\ ys && \text{using (3.3)}\end{aligned}$$

□

Now the recurrence outlined before is a direct consequence:

Theorem 3.3 (A recurrence for *select*).

$$\begin{aligned}&k < |xs| \longrightarrow\\ &select\ k\ xs\\ &= (\textbf{let}\ (ls,\ es,\ gs) = partition3\ x\ xs\\ &\quad \textbf{in if}\ k < |ls|\ \textbf{then}\ select\ k\ ls\\ &\qquad \textbf{else if}\ k < |ls| + |es|\ \textbf{then}\ x\ \textbf{else}\ select\ (k - |ls| - |es|)\ gs)\end{aligned}$$

Proof. We clearly have $\mathit{mset}\ \mathit{xs} = \mathit{mset}\ \mathit{ls} + \mathit{mset}\ \mathit{es} + \mathit{mset}\ \mathit{gs}$ and $|\mathit{xs}| = |\mathit{ls}| + |\mathit{es}| + |\mathit{gs}|$. Then:

$$\begin{aligned}
& \mathit{select}\ k\ \mathit{xs} \\
&= \mathit{select}\ k\ (\mathit{ls}\ @\ \mathit{es}\ @\ \mathit{gs}) && \text{using (3.5)} \\
&= \textbf{if}\ k < |\mathit{ls}|\ \textbf{then}\ \mathit{select}\ k\ \mathit{ls} \\
&\quad \textbf{else if}\ k - |\mathit{ls}| < |\mathit{es}|\ \textbf{then}\ \mathit{select}\ (k - |\mathit{ls}|)\ \mathit{es} && \text{using (3.6)} \\
&\quad \textbf{else}\ \mathit{select}\ (k - |\mathit{ls}| - |\mathit{es}|)\ \mathit{gs}
\end{aligned}$$

Clearly, $k - |\mathit{ls}| < |\mathit{es}| \longleftrightarrow k < |\mathit{ls}| + |\mathit{es}|$ and $\mathit{select}\ (k - |\mathit{ls}|)\ \mathit{es} = x$ since $\mathit{select}\ (k - |\mathit{ls}|)\ \mathit{es} \in \mathit{set}\ \mathit{es}$ and $\mathit{set}\ \mathit{es} = \{x\}$ by construction. □

Note that this holds for *any* pivot x. Indeed, x need not even be in the list itself. Therefore, the algorithm is (partially) correct no matter what pivot we choose. However, the number of recursive calls (and thereby the running time) depends strongly on the pivot choice:

- If we always choose a pivot that is smaller than any element in the list, the algorithm does not terminate at all.
- If we choose the smallest element in the list as a pivot every time, only one element is removed from the list in every recursion step so that we get n recursive calls in total. Since we do a linear amount of work in every step, this leads to a running time of $\Theta(n^2)$.
- If we choose pivots from the list at random, the worst-case running time is again $\Theta(n^2)$, but the expected running time can be shown to be $\Theta(n)$, similarly to the situation in Quicksort. Indeed, it can also be shown that it is very unlikely that the running time is significantly “worse than linear” [39, Section 2.5].
- If we choose a pivot that cuts the list in half every time, we get roughly $\lg n$ recursive steps and, by the Master Theorem, a running time of $\Theta(n)$ (assuming we can find such a pivot in linear time).

Clearly, the last case is the most desirable one. The element that cuts the list in half is called the *median* (a term widely used in statistics, albeit with a slightly different meaning). For lists of odd length, there is a unique element that achieves this, whereas for lists of even length there are two such elements (e.g. for the list [1,2,3,4], both 2 and 3 work). For our purposes, we always use the smaller one and define:

$$\mathit{median} :: {'a}\ \mathit{list} \Rightarrow {'a}$$

$$\mathit{median}\ \mathit{xs} = \mathit{select}\ ((|\mathit{xs}| - 1)\ \mathit{div}\ 2)\ \mathit{xs}$$

Unfortunately, computing the median of a list is no easier than selection (see Exercise 3.5), so it seems that, for now, this does not really help us.

Exercise 3.5. Show that *select k xs* can be reduced to computing the median of a list in linear time, i.e. give a linear-time function *reduce_select_median* such that

$$\begin{aligned}&xs \neq [] \wedge k < |xs| \longrightarrow\\&\mathit{reduce\_select\_median}\ k\ xs \neq [] \wedge\\&\mathit{median}\ (\mathit{reduce\_select\_median}\ k\ xs) = \mathit{select}\ k\ xs\end{aligned}$$

3.2 The Median of Medians

Since, as we have seen, computing a true median in every recursive step is too expensive, the next natural question is: is there something that is *almost* as good as a median but easier to compute?

This is indeed the case, and this is where the ingenuity of the algorithm is hidden: instead of computing the median of *all* the list elements, we use another list element M, which is computed as follows:

- chop the list into groups of 5 elements each (possibly with one smaller group at the end if n is not a multiple of 5)
- compute the median of each of the $\lceil\frac{n}{5}\rceil$ groups (which can be done in constant time for each group using e.g. insertion sort, since their sizes are bounded by 5)
- compute the median M of these $\lceil\frac{n}{5}\rceil$ elements (which can be done by a recursive call to the selection algorithm)

We call M the *median of medians*. M is not quite as good a pivot as the true median, but it is still fairly decent:

Theorem 3.4 (Pivoting bounds for the median of medians).
Let xs be a list and let $\prec$ *be either* $<$ *or* $>$. *Define*

$$M := \mathit{median}\ (\mathit{map}\ \mathit{median}\ (\mathit{chop}\ 5\ xs))\ .$$

where the chop function performs the chopping described earlier. Then:

$$|\{\!\!\{y \in_{\#} \mathit{mset}\ xs \mid y \prec M\}\!\!\}| \leq \lceil 0.7 \cdot n + 3\rceil$$

Proof. The result of $\mathit{chop}\ 5\ xs$ is a list of $\lceil n\ /\ 5\rceil$ chunks, each of size at most 5, i.e. $|\mathit{chop}\ 5\ xs| = \lceil n\ /\ 5\rceil$. Let us split these chunks into two groups according to whether their median is $\prec M$ or $\succeq M$:

$$\begin{aligned}Y_{\prec} &:= \{\!\!\{ys \in_{\#} \mathit{mset}\ (\mathit{chop}\ 5\ xs) \mid \mathit{median}\ ys \prec M\}\!\!\}\\Y_{\succeq} &:= \{\!\!\{ys \in_{\#} \mathit{mset}\ (\mathit{chop}\ 5\ xs) \mid \mathit{median}\ ys \succeq M\}\!\!\}\end{aligned}$$

We clearly have

$$mset\ xs = (\textstyle\sum_{ys \leftarrow chop\ 5\ xs} mset\ ys) \tag{3.8}$$

$$mset\ (chop\ 5\ xs) = Y_{\prec} + Y_{\succeq} \tag{3.9}$$

$$\lceil n\ /\ 5 \rceil = |Y_{\prec}| + |Y_{\succeq}| \tag{3.10}$$

and since M is the median of the medians of the groups, we also know that:

$$|Y_{\prec}| < \tfrac{1}{2} \cdot \lceil n\ /\ 5 \rceil \tag{3.11}$$

The core idea of the proof is that any group $ys \in_{\#} Y_{\succeq}$ can have at most 2 elements that are $\prec M$:

$$\begin{aligned}
&|\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| \\
&\leq |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec median\ ys \}\!\!\}| && \text{because } median\ ys \succeq M \\
&\leq |ys|\ div\ 2 && \text{using (3.1)} \\
&\leq 5\ div\ 2 = 2
\end{aligned}$$

And of course, since each group has size at most 5, any group in $ys \in_{\#} Y_{\prec}$ can contribute at most 5 elements. In summary, we have:

$$\begin{aligned}
&\forall\, ys {\in_{\#}} Y_{\prec}.\ |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| \leq 2 \\
&\forall\, ys {\in_{\#}} Y_{\succeq}.\ |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| \leq 5
\end{aligned} \tag{3.12}$$

With this, we can begin our estimation of the number of elements $\prec M$:

$$\begin{aligned}
&\{\!\!\{ y \in_{\#} mset\ xs \mid y \prec M \}\!\!\} \\
&= \{\!\!\{ y \in_{\#} (\textstyle\sum_{ys \leftarrow chop\ 5\ xs} mset\ ys) \mid y \prec M \}\!\!\} && \text{using(3.8)} \\
&= \textstyle\sum_{ys \leftarrow chop\ 5\ xs} \{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\} \\
&= \textstyle\sum_{ys \in_{\#} (Y_{\prec}\ +\ Y_{\succeq})} \{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\} && \text{using (3.9)}
\end{aligned}$$

Taking the size of both sides, we have

$$\begin{aligned}
&|\{\!\!\{ y \in_{\#} mset\ xs \mid y \prec M \}\!\!\}| \\
&\leq \textstyle\sum_{ys \in_{\#} (Y_{\prec}\ +\ Y_{\succeq})} |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| \\
&= \textstyle\sum_{ys \in_{\#} Y_{\prec}} |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| + \\
&\quad \textstyle\sum_{ys \in_{\#} Y_{\succeq}} |\{\!\!\{ y \in_{\#} mset\ ys \mid y \prec M \}\!\!\}| \\
&\leq (\textstyle\sum_{ys \in_{\#} Y_{\prec}} 5) + (\textstyle\sum_{ys \in_{\#} Y_{\succeq}} 2) && \text{using (3.12)} \\
&= 5 \cdot |Y_{\prec}| + 2 \cdot |Y_{\succeq}| \\
&= 2 \cdot \lceil n\ /\ 5 \rceil + 3 \cdot |Y_{\succeq}| && \text{using (3.10)} \\
&\leq 3.5 \cdot \lceil n\ /\ 5 \rceil && \text{using (3.11)} \\
&= 3.5 \cdot \lceil n\ /\ 5 \rceil \leq \lceil 0.7 \cdot n + 3 \rceil
\end{aligned}$$

The delicate arithmetic reasoning about rounding in the end can thankfully be done fully automatically by Isabelle's `linarith` tactic. □

3.3 Selection in Linear Time

We now have all the ingredients to write down our algorithm: the base cases (i.e. sufficiently short lists) can be handled using the naive approach of performing insertion sort and then returning the k-th element. For bigger lists, we perform the divide-and-conquer approach outlined in Theorem 3.3 using M as a pivot. We have two recursive calls: one on a list with exactly $\lceil 0.2 \cdot n \rceil$ elements to compute M, and one on a list with at most $\lceil 0.7 \cdot n + 3 \rceil$. We will still need to show later that this actually leads to a linear-time algorithm, but the fact that $0.7 + 0.2 < 1$ is at least encouraging.

The full algorithm looks like this:

```
chop :: nat ⇒ 'a list ⇒ 'a list list
chop 0 _ = []
chop _ [] = []
chop n xs = take n xs # chop n (drop n xs)

slow_select :: nat ⇒ 'a list ⇒ 'a
slow_select k xs = isort xs ! k

slow_median :: 'a list ⇒ 'a
slow_median xs = slow_select ((|xs| − 1) div 2) xs

mom_select :: nat ⇒ 'a list ⇒ 'a
mom_select k xs
= (if |xs| ≤ 20 then slow_select k xs
   else let M = mom_select ((⌈|xs| / 5⌉ − 1) div 2)
                  (map slow_median (chop 5 xs));
            (ls, es, gs) = partition3 M xs
        in if k < |ls| then mom_select k ls
           else if k < |ls| + |es| then M
           else mom_select (k − |ls| − |es|) gs)
```

The (partial) correctness statement is simple:

Theorem 3.5 (Partial Correctness of *mom_select*). *Let xs be a list and $k < |xs|$. Then if $mom\_select\ k\ xs$ terminates, we have*

$$mom\_select\ k\ xs = select\ k\ xs\ .$$

Proof. Straightforward computation induction using Theorem 3.3. □

Theorem 3.6 (Termination of *mom_select*). *Let xs be a list and $k < |xs|$. Then $mom\_select\ k\ xs$ terminates.*

Proof. We prove termination by using the length of the list xs as a termination measure. We perform a computation induction and need to show that this length decreases in every recursive call if $20 < |xs|$. This is easy to see:

- The list in the first recursive call has length $\lceil |xs|\ /\ 5 \rceil$, which is strictly less than $|xs|$ if $1 < |xs|$.
- The length of the list in the second recursive call is at most $|xs| - 1$: by induction hypothesis, the first recursive call terminates, so by Theorem 3.5 we know that $M = median\ (map\ median\ (chop\ 5\ xs))$ and thus:
  $$\begin{aligned} & M \in set\ (map\ median\ (chop\ 5\ xs)) \\ & = \{median\ ys \mid ys \in set\ (chop\ 5\ xs)\} \\ & \subseteq \textstyle\bigcup_{ys \in set\ (chop\ 5\ xs)} set\ ys = set\ xs \end{aligned}$$
  Hence, $M \in set\ xs$ but $M \notin set\ ls$ and $M \notin set\ gs$ by construction.

□

Exercise 3.6. The recursive definition of *mom_select* handles the cases $|xs| \leq 20$ through the naive algorithm using insertion sort. The constant 20 here seems somewhat arbitrary. Find the smallest constant n_0 for which the algorithm still works. Why do you think 20 was chosen?

3.4 Time Functions

It remains to show now that this indeed leads to a linear-time algorithm. The time function for our selection algorithm is as follows:

```
$T_{mom\_select}$ :: $nat \Rightarrow$ 'a $list \Rightarrow nat$
$T_{mom\_select}\ k\ xs$
= (if $|xs| \leq 20$ then $T_{slow\_select}\ k\ xs$
   else let $xss = chop\ 5\ xs$;
            $ms = map\ slow\_median\ xss$;
            $idx = ((|xs| + 4)\ div\ 5 - 1)\ div\ 2$;
            $x = mom\_select\ idx\ ms$;
            $(ls, es, gs) = partition3\ x\ xs$
        in $T_{mom\_select}\ idx\ ms + T_{chop}\ 5\ xs + T_{map}\ T_{slow\_median}\ xss$ +
           $T_{partition3}\ x\ xs + T_{length}\ ls + T_{length}\ es + 1$ +
           (if $k < |ls|$ then $T_{mom\_select}\ k\ ls$
            else if $k < |ls| + |es|$ then 0
            else $T_{mom\_select}\ (k - |ls| - |es|)\ gs$))
```

We can then prove

$$T_{mom\_select}\ k\ xs \le T'_{mom\_select}\ |xs|$$

where the upper bound T'_{mom_select} is defined as follows:

$$T'_{mom\_select} :: nat \Rightarrow nat$$

$$\begin{aligned}
&T'_{mom\_select}\ n\\
&= (\textbf{if}\ n \le 20\ \textbf{then}\ 463\\
&\quad\ \ \textbf{else}\ T'_{mom\_select}\ \lceil 0.2 \cdot n\rceil + T'_{mom\_select}\ \lceil 0.7 \cdot n + 3\rceil + 17 \cdot n + 50)
\end{aligned}$$

The time functions of the auxiliary functions used here can be found in Section B.2 in the appendix. The proof is a simple computation induction using Theorem 3.4 and the time bounds for the auxiliary functions from Chapter B in the appendix.

The next section will be dedicated to showing that $T'_{mom\_select} \in O(n)$.

Exercise 3.7. Show that the upper bound $\lceil 0.7 \cdot n + 3\rceil$ is fairly tight by giving an infinite family $(xs_i)_{i\in\mathbb{N}}$ of lists with increasing lengths for which more than 70 % of the elements are larger than the median of medians (with chopping size 5). In Isabelle terms: define a function $f :: nat \Rightarrow nat\ list$ such that $\forall n.\ |f\ n| < |f\ (n + 1)|$ and

$$\frac{|\{\!\{y \in_{\#} mset\ (f\ n) \mid y > mom\ (f\ n)\}\!\}|}{|f\ n|} > 0.7$$

where $mom\ xs = median\ (map\ median\ (chop\ 5\ xs))$.

3.5 Akra–Bazzi Light

The function T'_{mom_select} (let us write it as f for now) fulfils the recurrence

$$20 < n \longrightarrow f\ n = f\ \lceil 0.2 \cdot n\rceil + f\ \lceil 0.7 \cdot n + 3\rceil + 17 \cdot n + 50$$

Such divide-and-conquer recurrences are beyond the “normal” Master Theorem, but a generalisation, the Akra–Bazzi Theorem [3, 47, 17], does apply to them. Let us first abstract the situation a bit and consider the recurrence

$$20 < n \longrightarrow f\ n = f\ \lceil a \cdot n + b\rceil + f\ \lceil c \cdot n + d\rceil + C_1 \cdot n + C_2$$

where $0 < a,\ b < 1$ and $C_1,\ C_2 > 0$. The Akra–Bazzi Theorem then tells us that such a function is $O(n)$ if (and only if) $a + b < 1$. We will prove one direction of this (the one we need) now – Akra–Bazzi Light, so to say.

Instead of presenting the full theorem statement and its proof right away, let us take a more explorative approach. What we want to prove in the end is that there are real constants $C_3 > 0$ and C_4 such that $f\ n \leq C_3 \cdot n + C_4$ for all n. Suppose these constants are fixed already and now we want to prove that the inequality holds for them. For simplicity of the presentation, we assume $b,\ d \geq 0$, but note that these assumptions are unnecessary and the proof still works for negative b and d if we replace b and d with $max\ 0\ b$ and $max\ 0\ d$.

The obvious approach to show this is by induction on n, following the structure of the recurrence above. To do this, we use *complete induction* (i.e. the induction hypothesis holds for all $n' < n$)[1] and a case analysis on $n > n_0$ (where n_0 is some constant we will determine later).

The two cases we have to show in the induction are then:

Base case: $\forall\, n {\leq} n_0.\ f\ n \leq C_3 \cdot n + C_4$
Step: $\forall\, n {>} n_0.\ (\forall\, m {<} n.\ f\ m \leq C_3 \cdot m + C_4) \longrightarrow f\ n \leq C_3 \cdot n + C_4$

We can see that in order to even be able to apply the induction hypothesis in the induction step, we need $\lceil a \cdot n + b \rceil < n$. We can make the estimate[2]

$$\lceil a \cdot n + b \rceil \leq a \cdot n + b + 1 \overset{!}{<} n$$

and then solve for n, which gives us $n \overset{!}{>} \frac{b+1}{1-a}$. If we do the same for c and d as well, we get the conditions

$$n_0 \geq \frac{b+1}{1-a} \qquad \text{and} \qquad n_0 \geq \frac{d+1}{1-c} \tag{3.13}$$

However, it will later turn out that these are implied by the other conditions we will have accumulated anyway.

Now that we have ensured that the basic structure of our induction will work out, let us continue with the two cases.

The base cases ($n \leq n_0$) is fairly uninteresting: we can simply choose C_4 to be large enough to satisfy all of them.

In the recursive step, unfolding one step of the recurrence and applying the induction hypotheses leaves us with the following proof obligation

$$C_3 \cdot \lceil a \cdot n + b \rceil + C_4 + (C_3 \cdot \lceil c \cdot n + d \rceil + C_4) + C_1 \cdot n + C_2 \overset{!}{\leq} C_3 \cdot n + C_4\ ,$$

[1] In Isabelle, the corresponding rule is called `less_induct`.

[2] The notation $\overset{!}{<}$ stands for "must be less than". It emphasises that this inequality is not a consequence of what we have shown so far, but something that we still need to show, or in this case something that we need to ensure by adding suitable preconditions.

or, equivalently,

$$C_3 \cdot (\lceil a \cdot n + b \rceil + \lceil c \cdot n + d \rceil - n) + C_1 \cdot n + C_2 + C_4 \overset{!}{\le} 0 \ ,$$

We estimate the left-hand side like this:

$$\begin{aligned}
&C_3 \cdot (\lceil a \cdot n + b \rceil + \lceil c \cdot n + d \rceil - n) + C_1 \cdot n + C_2 + C_4 \\
&\le C_3 \cdot (a \cdot n + b + 1 + (c \cdot n + d + 1) - n) + C_1 \cdot n + C_2 + C_4 \\
&= C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n \quad (*) \\
&\le C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n_0 \quad (\dagger) \\
&\overset{!}{\le} 0
\end{aligned}$$

The step from $(*)$ to $(\dagger)$ uses the fact that $n > n_0$ and requires the factor $C_3 \cdot (1 - a - c) - C_1$ in front of the n to be positive, i.e. we need to add the assumption

$$C_3 > \frac{C_1}{1 - a - c} \ . \tag{3.14}$$

The term $(\dagger)$ (which we want to be ≤ 0) is now a constant. If we solve that inequality for C_3, we get the following two additional conditions:

$$n_0 > \frac{b + d + 2}{1 - a - c} \quad \text{and} \quad C_3 \ge \frac{C_1 \cdot n_0 + C_2 + C_4}{(1 - a - c) \cdot n_0 - b - d - 2} \tag{3.15}$$

The former of these directly implies our earlier conditions (3.13), so we can safely discard those now.

Now all we have to do is to find a combinations of n_0, C_3, and C_4 that satisfy (3.14) and (3.15). This is straightforward:

$$n_1 := max\ n_0\ \left(\left\lceil \frac{b + d + 2}{1 - a - c} \right\rceil + 1\right) \qquad C_4 := Max\ \{f\ n \mid n \le 20\}$$

$$C_3 := max\ \left(\frac{C_1}{1 - a - c}\right)\ \left(\frac{C_1 \cdot n_1 + C_2 + C_4}{(1 - a - c) \cdot n_1 - b - d - 2}\right)$$

And with that, the induction goes through and we get the following theorem:

Theorem 3.7 (Akra Bazzi Light).

$$\begin{aligned}
&a > 0 \wedge c > 0 \wedge a + c < 1 \wedge C_1 \ge 0 \wedge \\
&(\forall\, n{>}n_0.\ f\ n = f\ \lceil a \cdot n + b \rceil + f\ \lceil c \cdot n + d \rceil + C_1 \cdot n + C_2) \longrightarrow \\
&(\exists\, C_3\ \ C_4.\ \forall\, n.\ f\ n \le C_3 \cdot n + C_4)
\end{aligned}$$

Exercise 3.8.

1. Suppose that instead of groups of 5, we now chop into groups of size $l \ge 1$. Prove a corresponding generalisation of Theorem 3.4.
2. Examine (on paper only): how does this affect correctness and running time of our selection algorithm? Why do you think $l = 5$ was chosen?

Part II

Search Trees

4

Binary Trees

Binary trees are defined as a recursive data type:

```
datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)
```

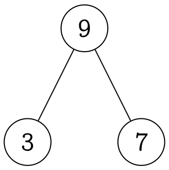
The following syntactic sugar is sprinkled on top:

$$\begin{array}{rcl} \langle\rangle & \equiv & \mathit{Leaf} \\ \langle l,\ x,\ r\rangle & \equiv & \mathit{Node}\ l\ x\ r \end{array}$$

The trees l and r are the left and right **children** of the node $\langle l,\ x,\ r\rangle$.

Because most of our tree will be binary trees, we drop the “binary” most of the time and have also called the type merely *tree*.

When displaying a tree in the usual graphical manner we show only the *Node*s. For example, $\langle\langle\langle\rangle,\ 3,\ \langle\rangle\rangle,\ 9,\ \langle\langle\rangle,\ 7,\ \langle\rangle\rangle\rangle$ is displayed like this:



The (label of the) **root** node is 9. The **depth** (or **level**) of some node (or leaf) in a tree is the distance from the root. We use these concepts only informally.

4.1 Basic Functions

Two canonical functions on data types are *set* and *map*:

$$\mathit{set\_tree} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{set}$$

$$\mathit{set\_tree}\ \langle\rangle = \{\}$$
$$\mathit{set\_tree}\ \langle l,\ x,\ r\rangle = \mathit{set\_tree}\ l \cup \{x\} \cup \mathit{set\_tree}\ r$$

$$\mathit{map\_tree} :: ({'a} \Rightarrow {'b}) \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'b}\ \mathit{tree}$$

$$\mathit{map\_tree}\ f\ \langle\rangle = \langle\rangle$$
$$\mathit{map\_tree}\ f\ \langle l,\ x,\ r\rangle = \langle \mathit{map\_tree}\ f\ l,\ f\ x,\ \mathit{map\_tree}\ f\ r\rangle$$

The *inorder*, *preorder* and *postorder* traversals (we omit the latter) list the elements in a tree in a particular order:

$$\mathit{inorder} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{list}$$

$$\mathit{inorder}\ \langle\rangle = []$$
$$\mathit{inorder}\ \langle l,\ x,\ r\rangle = \mathit{inorder}\ l\ @\ [x]\ @\ \mathit{inorder}\ r$$

$$\mathit{preorder} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{list}$$

$$\mathit{preorder}\ \langle\rangle = []$$
$$\mathit{preorder}\ \langle l,\ x,\ r\rangle = x\ \#\ \mathit{preorder}\ l\ @\ \mathit{preorder}\ r$$

These two size functions count the number of *Node*s and *Leaf*s in a tree:

$$\mathit{size} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$$

$$|\langle\rangle| = 0$$
$$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$$

$$\mathit{size}1 :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$$

$$|\langle\rangle|_1 = 1$$
$$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$$

The syntactic sugar $|t|$ for *size* t and $|t|_1$ for *size*1 t is only used in this text, not in the Isabelle theories.

Induction proves a convenient fact that explains the name *size*1:

$$|t|_1 = |t| + 1$$

The height and the minimal height of a tree are defined as follows:

```
height :: 'a tree ⇒ nat
h ⟨⟩ = 0
h ⟨l, _, r⟩ = max (h l) (h r) + 1

min_height :: 'a tree ⇒ nat
mh ⟨⟩ = 0
mh ⟨l, _, r⟩ = min (mh l) (mh r) + 1
```

You can think of them as the longest and shortest (cycle-free) path from the root to a leaf. The real names of these functions are *height* and *min_height*. The abbreviations *h* and *mh* are only used in this text, not in the Isabelle theories.

The obvious properties $h\ t \le |t|$ and $mh\ t \le h\ t$ and the following classical properties have easy inductive proofs:

$$2^{mh\ t} \le |t|_1 \qquad |t|_1 \le 2^{h\ t}$$

We will simply use these fundamental properties without referring to them by a name or number.

The set of subtrees of a tree is defined as follows:

```
subtrees :: 'a tree ⇒ 'a tree set
subtrees ⟨⟩ = {⟨⟩}
subtrees ⟨l, a, r⟩ = {⟨l, a, r⟩} ∪ subtrees l ∪ subtrees r
```

Note that every tree is a subtree of itself.

Exercise 4.1. Function *inorder* has quadratic complexity because the running time of (@) is linear in the length of its first argument. Define a function *inorder2* :: *'a tree* ⇒ *'a list* ⇒ *'a list* that avoids (@) but accumulates the result in its second parameter via (#) only. Its running time should be linear in the size of the tree. Prove *inorder2 t xs* = *inorder t* @ *xs*.

4.2 Complete Trees

A **complete tree** is one where all the leaves are on the same level. An example is shown in Figure 4.1. The predicate *complete* is defined recursively:

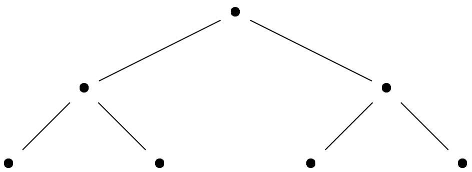


Fig. 4.1. A complete tree

```
complete :: 'a tree ⇒ bool
complete ⟨⟩ = True
complete ⟨l, _, r⟩ = (h l = h r ∧ complete l ∧ complete r)
```

This recursive definition is equivalent with the above definition that all leaves must have the same distance from the root. Formally:

Lemma 4.1. $\textit{complete}\ t \longleftrightarrow \textit{mh}\ t = h\ t$

Proof. The proof is by induction and case analyses on *min* and *max*. □

The following classic property of complete trees is easily proved by induction:

Lemma 4.2. $\textit{complete}\ t \longrightarrow |t|_1 = 2^{h\ t}$

It turns out below that this is in fact a defining property of complete trees.

For complete trees we have $(2{::}'b)^{mh\ t} \leq |t|_1 = 2^{h\ t}$. For incomplete trees the equalities the $=$ sign becomes a $<$ as the following two lemmas prove:

Lemma 4.3. $\neg\ \textit{complete}\ t \longrightarrow |t|_1 < 2^{h\ t}$

Proof. The proof is by induction. We focus in the induction step where $t = \langle l,\ x,\ r\rangle$. If t is incomplete, there are a number of cases and we prove $|t|_1 < 2^{h\ t}$ in each case. If $h\ l \neq h\ r$, consider the case $h\ l < h\ r$ (the case $h\ r < h\ l$ is symmetric). From $2^{h\ l} < 2^{h\ r}$, $|l|_1 \leq 2^{h\ l}$ and $|r|_1 \leq 2^{h\ r}$ the claim follows: $|t|_1 = |l|_1 + |r|_1 \leq 2^{h\ l} + 2^{h\ r} < 2 \cdot 2^{h\ r} = 2^{h\ t}$. If $h\ l = h\ r$, then either l or r must be incomplete. We consider the case $\neg\ \textit{complete}\ l$ (the case $\neg\ \textit{complete}\ r$ is symmetric). From the IH $|l|_1 < 2^{h\ l}$, $|r|_1 \leq 2^{h\ r}$ and $h\ l = h\ r$ the claim follows: $|t|_1 = |l|_1 + |r|_1 < 2^{h\ l} + 2^{h\ r} = 2 \cdot 2^{h\ r} = 2^{h\ t}$. □

Lemma 4.4. $\neg\ \textit{complete}\ t \longrightarrow 2^{mh\ t} < |t|_1$

The proof of this lemma is completely analogous to the previous proof except that one also needs to use Lemma 4.1.

From the contrapositive of Lemma 4.3 one obtains $|t|_1 = 2^{h\ t} \longrightarrow$ *complete t*, the converse of Lemma 4.2. Thus we arrive at:

Corollary 4.5. $\mathit{complete}\ t \longleftrightarrow |t|_1 = 2^{h\ t}$

The complete trees are precisely the ones where the height is exactly the logarithm of the number of leaves.

Exercise 4.2. Define a function *mcs* that computes a maximal complete subtree of some given tree. You are allowed only one traversal of the input but you may freely compute the height of trees and may even compare trees for equality. You are not allowed to use *complete* or *subtrees*.

Prove that *mcs* returns a complete subtree (which should be easy) and that it is maximal in height:

$$u \in \mathit{subtrees}\ t \wedge \mathit{complete}\ u \longrightarrow h\ u \leq h\ (\mathit{mcs}\ t)$$

Bonus: get rid of any tree equality tests in *mcs*.

4.3 Almost Complete Trees

An **almost complete tree** is one where the leaves may occur not just at the lowest level but also one level above:

$\mathit{acomplete} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$

$\mathit{acomplete}\ t = (h\ t - \mathit{mh}\ t \leq 1)$

An example of an almost complete tree is shown in Figure 4.2. You can think of an almost complete tree as a complete tree with (possibly) some additional nodes one level below the last full level.

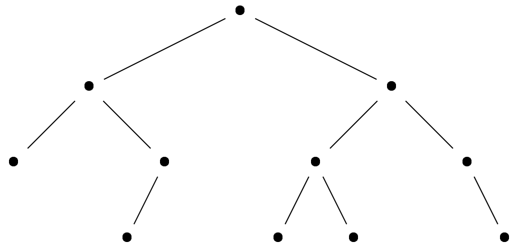


Fig. 4.2. Almost complete tree

Almost complete trees are important because among all the trees with the same number of nodes they have minimal height:

Lemma 4.6. $acomplete\ s \wedge |s| \leq |t| \longrightarrow h\ s \leq h\ t$

Proof. The proof is by cases. If *complete* s then, by Lemma 4.2, $2^{h\ s} = |s|_1 \leq |t|_1 \leq 2^{h\ t}$ and thus $h\ s \leq h\ t$. Now assume $\neg$ *complete* s. Then Lemma 4.4 yields $2^{mh\ s} < |s|_1 \leq |t|_1 \leq 2^{h\ t}$ and thus $mh\ s < h\ t$. Furthermore we have $h\ s - mh\ s \leq 1$ (from *acomplete* s), $h\ s \neq mh\ s$ (from Lemma 4.1) and $mh\ s \leq h\ s$, which together imply $mh\ s + 1 = h\ s$. With $mh\ s < h\ t$ this implies $h\ s \leq h\ t$. □

This is relevant for search trees because their height determines the worst case running time. Almost complete trees are optimal in that sense.

The following lemma yields an explicit formula for the height of an almost complete tree:

Lemma 4.7. $acomplete\ t \longrightarrow h\ t = \lceil \lg\ |t|_1 \rceil$

Proof. The proof is by cases. If t is complete, the claim follows from Lemma 4.2. Now assume t is incomplete. Then $h\ t = mh\ t + 1$ because *acomplete* t, $mh\ t \leq h\ t$ and $complete\ t \longleftrightarrow mh\ t = h\ t$ (Lemma 4.1). Together with $|t|_1 \leq 2^{h\ t}$ this yields $|t|_1 \leq 2^{mh\ t\ +\ 1}$ and thus $\lg\ |t|_1 \leq mh\ t + 1$. By Lemma 4.4 we obtain $mh\ t < \lg\ |t|_1$. These two bounds for $\lg\ |t|_1$ together imply the claimed $h\ t = \lceil \lg\ |t|_1 \rceil$. □

In the same manner we also obtain:

Lemma 4.8. $acomplete\ t \longrightarrow mh\ t = \lfloor \lg\ |t|_1 \rfloor$

4.3.1 Converting a List into an Almost Complete Tree

We will now see how to convert a list *xs* into an almost complete tree t such that *inorder* t = *xs*. If the list is sorted, the result is an almost complete binary search tree (see the next chapter). The basic idea is to cut the list in two halves, turn them into almost complete trees recursively and combine them. But cutting up the list in two halves explicitly would lead to an $n \cdot \lg\ n$ algorithm but we want a linear one. Therefore we use an additional *nat* parameter to tell us how much of the input list should be turned into a tree. The remaining list is returned with the tree:

```
bal :: nat ⇒ 'a list ⇒ 'a tree × 'a list
bal n xs
= (if n = 0 then (⟨⟩, xs)
   else let m = n div 2;
            (l, ys) = bal m xs;
```

```
        (r, zs) = bal (n − 1 − m) (tl ys)
    in (⟨l, hd ys, r⟩, zs))
```

The trick is not to chop *xs* in half but *n* because we assume that arithmetic is constant-time. Hence *bal* runs in linear time (see Exercise 4.4). Figure 4.3 shows the result of *bal* 10 [0..9].

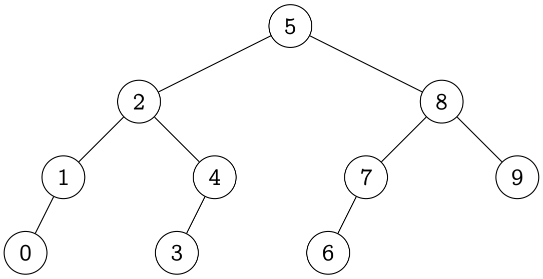


Fig. 4.3. Balancing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Balancing some prefix or all of a list or tree is easily derived:

```
bal_list :: nat ⇒ 'a list ⇒ 'a tree
bal_list n xs = fst (bal n xs)

balance_list :: 'a list ⇒ 'a tree
balance_list xs = bal_list |xs| xs

bal_tree :: nat ⇒ 'a tree ⇒ 'a tree
bal_tree n t = bal_list n (inorder t)

balance_tree :: 'a tree ⇒ 'a tree
balance_tree t = bal_tree |t| t
```

Correctness

The following lemma clearly expresses that *bal n xs* turns the prefix of length *n* of *xs* into a tree and returns the corresponding suffix of *xs*:

Lemma 4.9. $n \le |xs| \land bal\ n\ xs = (t,\ zs) \longrightarrow$
$xs = inorder\ t\ @\ zs \land |t| = n$

Proof by complete induction on n, assuming that the property holds for all values below n. If $n = 0$ the claim is trivial. Now assume $n \neq 0$ and let $m = n$ *div* 2 and $m' = n - 1 - m$ (and thus $m,\ m' < n$). From $bal\ n\ xs = (t,\ zs)$ we obtain l, r and ys such that $bal\ m\ xs = (l,\ ys)$, $bal\ m'\ (tl\ ys) = (r,\ zs)$ and $t = \langle l,\ hd\ ys,\ r\rangle$. Because $m < n \le |xs|$ the induction hypothesis implies $xs = inorder\ l\ @\ ys \land |l| = m$ (1). This in turn implies $m' \le |tl\ ys|$ and thus the induction hypothesis implies $tl\ ys = inorder\ r\ @\ zs \land |r| = m'$ (2). Properties (1) and (2) together with $t = \langle l,\ hd\ ys,\ r\rangle$ imply the claim $xs = inorder\ t\ @\ zs \land |t| = n$ because $ys \neq []$. □

The corresponding correctness properties of the derived functions are easy consequences:

$$\begin{aligned}
n \le |xs| \longrightarrow\ & inorder\ (bal\_list\ n\ xs) = take\ n\ xs\\
& inorder\ (balance\_list\ xs) = xs\\
n \le |t| \longrightarrow\ & inorder\ (bal\_tree\ n\ t) = take\ n\ (inorder\ t)\\
& inorder\ (balance\_tree\ t) = inorder\ t
\end{aligned}$$

To prove that *bal* returns an almost complete tree we determine its height and minimal height.

Lemma 4.10. $n \le |xs| \land bal\ n\ xs = (t,\ zs) \longrightarrow h\ t = \lceil \lg\ (n + 1)\rceil$

Proof. The proof structure is the same as for Lemma 4.9 and we reuse the variable names introduced there. In the induction step we obtain the simplified induction hypothesese $h\ l = \lceil \lg\ (m + 1)\rceil$ and $h\ r = \lceil \lg\ (m' + 1)\rceil$. This leads to

$$\begin{aligned}
& h\ t = max\ (h\ l)\ (h\ r) + 1\\
& = h\ l + 1 && \text{because } m' \le m\\
& = \lceil \lg\ (m + 1) + 1\rceil\\
& = \lceil \lg\ (n + 1)\rceil && \text{by (2.29)}
\end{aligned}$$ □

The following complementary lemma is proved in the same way:

Lemma 4.11. $n \le |xs| \land bal\ n\ xs = (t,\ zs) \longrightarrow mh\ t = \lfloor \lg\ (n + 1)\rfloor$

By definition of *acomplete* and because $\lceil x\rceil - \lfloor x\rfloor \le 1$ we obtain that *bal* (and consequently the functions that build on it) returns an almost complete tree:

Corollary 4.12. $n \le |xs| \land bal\ n\ xs = (t,\ ys) \longrightarrow acomplete\ t$

4.3.2 Exercises

Exercise 4.3. Find a formula B such that $\mathit{acomplete}\ \langle l,\ x,\ r\rangle = B$ where B may only contain the functions *acomplete*, *complete*, *height*, arithmetic and boolean operations, l and r, but in particular not *min_height* or *Node* $(= \langle \_,\ \_,\ \_\rangle)$. Prove $\mathit{acomplete}\ \langle l,\ x,\ r\rangle = B$.

Exercise 4.4. Prove that the running time of function *bal* is linear in its first argument. (Isabelle hint: you need to refer to *bal* as *Balance.bal*.)

4.4 Augmented Trees

A tree of type *'a tree* only stores elements of type *'a*. However, it is frequently necessary to store some additional information of type *'b* in each node too, usually for efficiency reasons. Typical examples are:

- The size or the height of the tree. Because recomputing them requires traversing the whole tree.
- Lookup tables where each key of type *'a* is associated with a value of type *'b*.

In this case we simply work with trees of type $('a \times 'b)$ *tree* and call them **augmented trees**. As a result we need to redefine a few functions that should ignore the additional information. For example, function *inorder*, when applied to an augmented tree, should return an *'a list*. Thus we redefine it in the obvious way:

$$\mathit{inorder} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'a\ \mathit{list}$$

$$\mathit{inorder}\ \langle\rangle = []$$
$$\mathit{inorder}\ \langle l,\ (a,\ \_),\ r\rangle = \mathit{inorder}\ l\ @\ a\ \#\ \mathit{inorder}\ r$$

Another example is $\mathit{set\_tree} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'a\ \mathit{set}$. In general, if a function f is originally defined on type *'a tree* but should ignore the *'b*-values in an $('a \times 'b)$ *tree* then we assume that there is a corresponding revised definition of f on augmented trees that focuses on the *'a*-values just like *inorder* above does. Of course functions that do not depend on the information in the nodes, e.g. size and height, stay unchanged.

Note that there are two alternative redefinitions of *inorder* (and similar functions): $\mathit{map}\ \mathit{fst} \circ \mathit{Tree.inorder}$ or $\mathit{Tree.inorder} \circ \mathit{map\_tree}\ \mathit{fst}$ where *Tree.inorder* is the original function.

4.4.1 Maintaining Augmented Trees

Maintaining the $'b$-values in a tree can be hidden inside a suitable smart version of *Node* that has only a constant time overhead. Take the example of augmentation by size:

$$\mathit{sz} :: ('a \times \mathit{nat})\ \mathit{tree} \Rightarrow \mathit{nat}$$
$$\mathit{sz}\ \langle\rangle = 0$$
$$\mathit{sz}\ \langle\_,\ (\_,\ n),\ \_\rangle = n$$

$$\mathit{node\_sz} :: ('a \times \mathit{nat})\ \mathit{tree} \Rightarrow 'a \Rightarrow ('a \times \mathit{nat})\ \mathit{tree} \Rightarrow ('a \times \mathit{nat})\ \mathit{tree}$$
$$\mathit{node\_sz}\ l\ a\ r = \langle l,\ (a,\ \mathit{sz}\ l + \mathit{sz}\ r + 1),\ r\rangle$$

A $('a \times \mathit{nat})\ \mathit{tree}$ satisfies $\mathit{invar\_sz}$ if the size annotation of every node is computed from its children as specified in $\mathit{node\_sz}$:

$$\mathit{invar\_sz} :: ('a \times \mathit{nat})\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{invar\_sz}\ \langle\rangle = \mathit{True}$$
$$\mathit{invar\_sz}\ \langle l,\ (\_,\ n),\ r\rangle$$
$$= (n = \mathit{sz}\ l + \mathit{sz}\ r + 1 \wedge \mathit{invar\_sz}\ l \wedge \mathit{invar\_sz}\ r)$$

This property is preserved by $\mathit{node\_sz}$ (i.e. $\mathit{invar\_sz}\ l \wedge \mathit{invar\_sz}\ r \longrightarrow \mathit{invar\_sz}\ (\mathit{node\_sz}\ l\ a\ r)$) and it guarantees that sz returns the size:

$$\mathit{invar\_sz}\ t \longrightarrow \mathit{sz}\ t = |t|$$

We can generalize this example easily. Assume we have a constant $\mathit{zero} :: 'b$ and a function $f :: 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ which we iterate over the tree:

$$F :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'b$$
$$F\ \langle\rangle = \mathit{zero}$$
$$F\ \langle l,\ (a,\ \_),\ r\rangle = f\ (F\ l)\ a\ (F\ r)$$

This generalizes the definition of size. Let $\mathit{node\_f}$ compute the $'b$-value from the $'b$-value of its children via f:

$$\mathit{b\_val} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'b$$
$$\mathit{b\_val}\ \langle\rangle = \mathit{zero}$$

```
b_val ⟨_ , (_ , b), _⟩ = b

node_f :: ('a × 'b) tree ⇒ 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
node_f l a r = ⟨l, (a, f (b_val l) a (b_val r)), r⟩
```

If all $'b$-values are computed as in $\mathit{node\_f}$

```
invar_f :: ('a × 'b) tree ⇒ bool
invar_f ⟨⟩ = True
invar_f ⟨l, (a, b), r⟩
= (b = f (b_val l) a (b_val r) ∧ invar_f l ∧ invar_f r)
```

then all $'b$-values equal F: $\mathit{invar\_f}\ t \longrightarrow \mathit{b\_val}\ t = F\ t$.

4.4.2 Exercises

Exercise 4.5. Augment trees by a pair of a boolean and something else where the boolean indicates whether the tree is complete or not. Define ch, $\mathit{node\_ch}$ and $\mathit{invar\_ch}$ as in Section 4.4.1 and prove the following properties:

$$\mathit{invar\_ch}\ t \longrightarrow \mathit{ch}\ t = (\mathit{complete}\ t,\ ?)$$
$$\mathit{invar\_ch}\ l \wedge \mathit{invar\_ch}\ r \longrightarrow \mathit{invar\_ch}\ (\mathit{node\_ch}\ l\ a\ r)$$

Exercise 4.6. Assume type $'a$ is of class $\mathit{linorder}$ and augment each Node with the maximum value in that tree. Following Section 4.4.1 (but mind the option type!) define $\mathit{mx} :: ('a \times 'b)\ \mathit{tree} \Rightarrow\ 'b\ \mathit{option}$, $\mathit{node\_mx}$ and $\mathit{invar\_mx}$ and prove

$$\mathit{invar\_mx}\ t \longrightarrow$$
$$\mathit{mx}\ t = (\textbf{if}\ t = \langle\rangle\ \textbf{then}\ \mathit{None}\ \textbf{else}\ \mathit{Some}\ (\mathit{Max}\ (\mathit{set\_tree}\ t)))$$

where Max is the predefined maximum operator on finite, non-empty sets.

5

Binary Search Trees

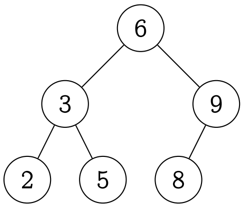
The purpose of this chapter is threefold: to introduce **binary search trees** (BSTs), to discuss their correctness proofs, and to provide a first example of an abstract data type, a notion discussed in more detail in the next chapter.

Search trees are a means for storing and accessing collections of elements efficiently. In particular they can support sets and maps. We concentrate on sets. We have already seen function *set_tree* that maps a tree to the set of its elements. This is an example of an **abstraction function** that maps concrete data structures to the abstract values that they represent.

BSTs require a linear ordering on the elements in the tree (as in Chapter Sorting). For each node, the elements in the left child are smaller than the root and the elements in the right child are bigger:

$$\mathit{bst} :: ('a{::}\mathit{linorder})\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{bst}\ \langle\rangle = \mathit{True}$$
$$\mathit{bst}\ \langle l,\ a,\ r\rangle$$
$$= ((\forall x{\in}\mathit{set\_tree}\ l.\ x < a) \wedge (\forall x{\in}\mathit{set\_tree}\ r.\ a < x) \wedge \mathit{bst}\ l \wedge \mathit{bst}\ r)$$

This is an example of a (coincidentally almost complete) BST:



It is obvious how to search for an element in a BST by comparing the element with the root and descending into one of the two children if you have not found it yet. In the worst case this takes time proportional to the height of the tree. In later chapters we discuss a number of methods for ensuring that the height of the tree is logarithmic in its size. For now we ignore all efficiency considerations and permit our BSTs to degenerate.

Exercise 5.1. The above recursive definition of *bst* is not a direct translation of the description "For each node" given in the text. For a more direct translation define a function

$$\mathit{nodes} :: {'a}\ \mathit{tree} \Rightarrow ({'a}\ \mathit{tree} \times {'a} \times {'a}\ \mathit{tree})\ \mathit{set}$$

that collects all the nodes as triples $(l,\ a,\ r)$. Now define *bst_nodes* as $\mathit{bst\_nodes}\ t = (\forall (l,a,r) \in \mathit{nodes}\ t.\ \ldots)$ and prove $\mathit{bst\_nodes}\ t = \mathit{bst}\ t$.

5.1 Interfaces

Trees are concrete data types that provide the building blocks for realizing abstract data types like sets. The abstract type has a fixed interface, i.e. set of operations, through which the values of the abstract type can be manipulated. The interface hides all implementation detail. In the Search Tree part of the book we focus on the abstract type of sets with the following interface:

$$\begin{aligned}
&\mathit{empty} :: {'s}\\
&\mathit{insert} :: {'a} \Rightarrow {'s} \Rightarrow {'s}\\
&\mathit{delete} :: {'a} \Rightarrow {'s} \Rightarrow {'s}\\
&\mathit{isin} :: {'s} \Rightarrow {'a} \Rightarrow \mathit{bool}
\end{aligned}$$

where $'s$ is the type of sets of elements of type $'a$. Most of our implementations of sets will be based on variants of BSTs and will require a linear order on $'a$, but the general interface does not require this. The correctness of an implementation of this interface will be proved by relating it back to HOL's type $'a$ *set* via an abstraction function, e.g. *set_tree*.

5.2 Implementing Sets via unbalanced BSTs

So far we have compared elements via $=$, $\leq$ and $<$. Now we switch to a comparator-based approach:

$$\textbf{datatype}\ \mathit{cmp\_val} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}$$

$$\begin{aligned}
&\mathit{cmp} :: ({'a}{::}\ \mathit{linorder}) \Rightarrow {'a} \Rightarrow \mathit{cmp\_val}\\
&\mathit{cmp}\ x\ y = (\textbf{if}\ x < y\ \textbf{then}\ \mathit{LT}\ \textbf{else if}\ x = y\ \textbf{then}\ \mathit{EQ}\ \textbf{else}\ \mathit{GT})
\end{aligned}$$

We will frequently phrase algorithms in terms of *cmp*, *LT*, *EQ* and *GT* instead of $<$, $=$ and $>$. This leads to more symmetric code. If some type comes with its own primitive *cmp* function this can yield a speed-up over the above generic *cmp* function.

Below you find an implementation of the set interface in terms of BSTs. Functions *isin* and *insert* are self-explanatory. Deletion is more interesting.

```
empty :: 'a tree
empty = ⟨⟩

isin :: 'a tree ⇒ 'a ⇒ bool
isin ⟨⟩ _ = False
isin ⟨l, a, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ isin r x)

insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
insert x ⟨l, a, r⟩ = (case cmp x a of
                       LT ⇒ ⟨insert x l, a, r⟩ |
                       EQ ⇒ ⟨l, a, r⟩ |
                       GT ⇒ ⟨l, a, insert x r⟩)

delete :: 'a ⇒ 'a tree ⇒ 'a tree
delete x ⟨⟩ = ⟨⟩
delete x ⟨l, a, r⟩
= (case cmp x a of
    LT ⇒ ⟨delete x l, a, r⟩ |
    EQ ⇒ if r = ⟨⟩ then l
          else let (a', r') = split_min r in ⟨l, a', r'⟩ |
    GT ⇒ ⟨l, a, delete x r⟩)

split_min :: 'a tree ⇒ 'a × 'a tree
split_min ⟨l, a, r⟩
= (if l = ⟨⟩ then (a, r) else let (x, l') = split_min l in (x, ⟨l', a, r⟩))
```

5.2.1 Deletion

Function *delete* deletes a from $\langle l,\ a,\ r\rangle$ (where $r \neq \langle\rangle$) by replacing a with a' and r by r' where

a' is the leftmost (least) element of r, also called the inorder successor of a, r' is the remainder of r after removing a'.

We call this **deletion by replacing**. Of course one can also obtain a' as the inorder predecessor of a in l.

An alternative is to delete a from $\langle l,\ a,\ r\rangle$ by "joining" l and r:

```
delete2 :: 'a ⇒ 'a tree ⇒ 'a tree
delete2 _ ⟨⟩ = ⟨⟩
delete2 x ⟨l, a, r⟩ = (case cmp x a of
                        LT ⇒ ⟨delete2 x l, a, r⟩ |
                        EQ ⇒ join l r |
                        GT ⇒ ⟨l, a, delete2 x r⟩)

join :: 'a tree ⇒ 'a tree ⇒ 'a tree
join t ⟨⟩ = t
join ⟨⟩ t = t
join ⟨$t_1$, a, $t_2$⟩ ⟨$t_3$, b, $t_4$⟩
= (case join $t_2$ $t_3$ of
    ⟨⟩ ⇒ ⟨$t_1$, a, ⟨⟨⟩, b, $t_4$⟩⟩ |
    ⟨$u_2$, x, $u_3$⟩ ⇒ ⟨⟨$t_1$, a, $u_2$⟩, x, ⟨$u_3$, b, $t_4$⟩⟩)
```

We call this **deletion by joining**. The characteristic property of *join* is *inorder* (*join* l r) = *inorder* l @ *inorder* r.

The definition of *join* may appear needlessly complicated. Why not this much simpler version:

```
join0 t ⟨⟩ = t
join0 ⟨⟩ t = t
join0 ⟨$t_1$, a, $t_2$⟩ ⟨$t_3$, b, $t_4$⟩ = ⟨$t_1$, a, ⟨join0 $t_2$ $t_3$, b, $t_4$⟩⟩
```

Because with this version of *join*, deletion may almost double the height of the tree, in contrast to *join* and also deletion by replacing, where the height cannot increase:

Exercise 5.2. First prove that *join* behaves well:

$$h\ (join\ l\ r) \le max\ (h\ l)\ (h\ r) + 1$$

Now show that *join*0 behaves badly: find an upper bound *ub* of h (*join*0 l r) such that *ub* is a function of $h\ l$ and $h\ r$. Prove h (*join*0 l r) $\le$ *ub* and prove that *ub* is a tight upper bound if l and r are complete trees.

We focus on *delete*, deletion by replacing, in the rest of the chapter.

5.3 Correctness

Why is the above implementation correct? Roughly speaking, because the implementations of *empty*, *insert*, *delete* and *isin* on type *'a tree* simulate the behaviour of $\{\}$, $\cup$, $-$ and $\in$ on type *'a set*. Taking the abstraction function into account we can formulate the simulation precisely:

$$\mathit{set\_tree}\ \mathit{empty} = \{\}$$
$$\mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \mathit{set\_tree}\ t \cup \{x\}$$
$$\mathit{set\_tree}\ (\mathit{delete}\ x\ t) = \mathit{set\_tree}\ t - \{x\}$$
$$\mathit{isin}\ t\ x = (x \in \mathit{set\_tree}\ t)$$

However, the implementation only works correctly on BSTs. Therefore we need to add the precondition *bst t* to all but the first property. But why are we permitted to assume this precondition? Only because *bst* is an **invariant** of this implementation: *bst* holds for *empty*, and both *insert* and *delete* preserve *bst*. Therefore every tree that can be manufactured through the interface is a BST. Of course this adds another set of proof obligations for correctness, **invariant preservation**:

$$\mathit{bst}\ \mathit{empty}$$
$$\mathit{bst}\ t \longrightarrow \mathit{bst}\ (\mathit{insert}\ x\ t)$$
$$\mathit{bst}\ t \longrightarrow \mathit{bst}\ (\mathit{delete}\ x\ t)$$

When looking at the abstract data type of sets from the user (or ‘client’) perspective, we would call the collection of all proof obligations for the correctness of an implementation the **specification** of the abstract type.

Exercise 5.3. Verify the implementation in Section 5.2 by showing all the proof obligations above, without the detour via sorted lists explained below.

5.4 Correctness Proofs

It turns out that direct proofs of the properties in the previous section are cumbersome — at least for *delete* and for proof assistants like Isabelle. Yet the correctness of the implementation is quite obvious to most (functional) programmers. Which is why most algorithm texts do not spend any time on functional correctness of search trees and concentrate on non-obvious structural properties that imply the logarithmic height of the trees — of course our simple BSTs do not guarantee the latter.

We will now present how the vague notion of “obvious” can be concretized and automated to such a degree that we do not need to discuss functional correctness of search tree implementations again in this book. This is because our approach is quite generic: it works not only for the BSTs in this chapter but also for the more efficient variants discussed in later chapters. The remainder of this section can be skipped if one is not interested in proof automation.

5.4.1 The Idea

The key idea [54] is to express *bst* and *set_tree* via *inorder*:

$$bst\ t = sorted\ (inorder\ t) \quad \text{and} \quad set\_tree\ t = set\ (inorder\ t)$$

where

```
sorted :: 'a list ⇒ bool
sorted [] = True
sorted [_] = True
sorted (x # y # zs) = (x < y ∧ sorted (y # zs))
```

Note that this is “sorted w.r.t. $(<)$” whereas in the chapter on sorting *sorted* was defined as “sorted w.r.t. $(\leq)$”.

Instead of showing directly that BSTs implement sets, we show that they implement an intermediate specification based on lists (and later that the list-based specification implies the set-based one). We can assume that the lists are *sorted* because they are abstractions of BSTs. Insertion and deletion on sorted lists can be defined as follows:

```
ins_list :: 'a ⇒ 'a list ⇒ 'a list
ins_list x [] = [x]
ins_list x (a # xs)
= (if x < a then x # a # xs
   else if x = a then a # xs else a # ins_list x xs)

del_list :: 'a ⇒ 'a list ⇒ 'a list
del_list _ [] = []
del_list x (a # xs) = (if x = a then xs else a # del_list x xs)
```

The abstraction function from trees to lists is function *inorder*. The specification in Figure 5.1 expresses that *empty*, *insert*, *delete* and *isin* implement [], *ins_list*, *del_list* and $\lambda xs\ x.\ x \in set\ xs$. One nice aspect of this specification is that it does not require us to prove invariant preservation explicitly: it follows from the fact (proved below) that *ins_list* and *del_list* preserve *sorted*.

$$\begin{aligned}
&\textit{inorder empty} = []\\
&\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{insert}\ x\ t) = \textit{ins\_list}\ x\ (\textit{inorder}\ t)\\
&\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{delete}\ x\ t) = \textit{del\_list}\ x\ (\textit{inorder}\ t)\\
&\textit{bst}\ t \longrightarrow \textit{isin}\ t\ x = (x \in \textit{set\_tree}\ t)
\end{aligned}$$

Fig. 5.1. List-based Specification of BSTs

5.4.2 BSTs Implement Sorted Lists — A Framework

We present a library of lemmas that automate the functional correctness proofs for the BSTs in this chapter and the more efficient variants in later chapters. This library is motivated by general considerations concerning the shape of formulas that arise during verification.

As a motivating example we examine how to prove

$$\textit{sorted}\ (\textit{inorder}\ t) \longrightarrow \textit{inorder}\ (\textit{insert}\ x\ t) = \textit{ins\_list}\ x\ (\textit{inorder}\ t)$$

The proof is by induction on t and we consider the case $t = \langle l,\ a,\ r\rangle$ such that $x < a$. Ideally the proof looks like this:

$$\begin{aligned}
&\textit{inorder}\ (\textit{insert}\ x\ t) = \textit{inorder}\ (\textit{insert}\ x\ l)\ @\ a\ \#\ \textit{inorder}\ r\\
&= \textit{ins\_list}\ x\ (\textit{inorder}\ l)\ @\ a\ \#\ \textit{inorder}\ r\\
&= \textit{ins\_list}\ x\ (\textit{inorder}\ l\ @\ a\ \#\ \textit{inorder}\ r) = \textit{ins\_list}\ x\ t
\end{aligned}$$

The first and last step are by definition, the second step by induction hypothesis, and the third step by lemmas in Figure 5.2: (5.1) rewrites the assumption $\textit{sorted}\ (\textit{inorder}\ t)$ to $\textit{sorted}\ (\textit{inorder}\ l\ @\ [a]) \wedge \textit{sorted}\ (a\ \#\ \textit{inorder}\ r)$, thus allowing (5.5) to rewrite the term $\textit{ins\_list}\ x\ (\textit{inorder}\ l\ @\ a\ \#\ \textit{inorder}\ r)$ to $\textit{ins\_list}\ x\ (\textit{inorder}\ l)\ @\ a\ \#\ \textit{inorder}\ r$.

The lemma library in Figure 5.2 helps to prove the properties in Figure 5.1. These proofs are by induction on t and lead to (possibly nested) tree constructor terms like $\langle\langle t_1,\ a_1,\ t_2\rangle,\ a_2,\ t_3\rangle$ where the t_i and a_i are variables. Evaluating *inorder* of such a tree leads to a list of the following form:

$$\textit{inorder}\ t_1\ @\ a_1\ \#\ \textit{inorder}\ t_2\ @\ a_2\ \#\ \ldots\ \#\ \textit{inorder}\ t_n$$

Now we discuss the lemmas in Figure 5.2 that simplify the application of *sorted*, *ins_list* and *del_list* to such terms.

Terms of the form $\textit{sorted}\ (\textit{xs}_1\ @\ a_1\ \#\ \textit{xs}_2\ @\ a_2\ \#\ \ldots\ \#\ \textit{xs}_n)$ are decomposed into the following *basic* formulas

$$\begin{array}{ll}
\textit{sorted}\ (\textit{xs}\ @\ [a]) & (\text{simulating } \forall x{\in}\textit{set xs}.\ x < a)\\
\textit{sorted}\ (a\ \#\ \textit{xs}) & (\text{simulating } \forall x{\in}\textit{set xs}.\ a < x)\\
a < b &
\end{array}$$

by the rewrite rules (5.1)–(5.2). Lemmas (5.3)–(5.4) enable deductions from basic formulas.

$$\mathit{sorted}\ (\mathit{xs}\ @\ y\ \#\ \mathit{ys}) = (\mathit{sorted}\ (\mathit{xs}\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ \mathit{ys})) \tag{5.1}$$

$$\begin{aligned}&\mathit{sorted}\ (x\ \#\ \mathit{xs}\ @\ y\ \#\ \mathit{ys})\\ &= (\mathit{sorted}\ (x\ \#\ \mathit{xs}) \wedge x < y \wedge \mathit{sorted}\ (\mathit{xs}\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ \mathit{ys}))\end{aligned} \tag{5.2}$$

$$\mathit{sorted}\ (x\ \#\ \mathit{xs}) \longrightarrow \mathit{sorted}\ \mathit{xs} \tag{5.3}$$

$$\mathit{sorted}\ (\mathit{xs}\ @\ [y]) \longrightarrow \mathit{sorted}\ \mathit{xs} \tag{5.4}$$

$$\begin{aligned}&\mathit{sorted}\ (\mathit{xs}\ @\ [a]) \Longrightarrow \mathit{ins\_list}\ x\ (\mathit{xs}\ @\ a\ \#\ \mathit{ys}) =\\ &\quad (\mathbf{if}\ x < a\ \mathbf{then}\ \mathit{ins\_list}\ x\ \mathit{xs}\ @\ a\ \#\ \mathit{ys}\ \mathbf{else}\ \mathit{xs}\ @\ \mathit{ins\_list}\ x\ (a\ \#\ \mathit{ys}))\end{aligned} \tag{5.5}$$

$$\begin{aligned}&\mathit{sorted}\ (\mathit{xs}\ @\ a\ \#\ \mathit{ys}) \Longrightarrow \mathit{del\_list}\ x\ (\mathit{xs}\ @\ a\ \#\ \mathit{ys}) =\\ &\quad (\mathbf{if}\ x < a\ \mathbf{then}\ \mathit{del\_list}\ x\ \mathit{xs}\ @\ a\ \#\ \mathit{ys}\ \mathbf{else}\ \mathit{xs}\ @\ \mathit{del\_list}\ x\ (a\ \#\ \mathit{ys}))\end{aligned} \tag{5.6}$$

$$\mathit{sorted}\ (x\ \#\ \mathit{xs}) = ((\forall\, y \in \mathit{set}\ \mathit{xs}.\ x < y) \wedge \mathit{sorted}\ \mathit{xs}) \tag{5.7}$$

$$\mathit{sorted}\ (\mathit{xs}\ @\ [x]) = (\mathit{sorted}\ \mathit{xs} \wedge (\forall\, y \in \mathit{set}\ \mathit{xs}.\ y < x)) \tag{5.8}$$

Fig. 5.2. Lemmas for *sorted*, *ins_list*, *del_list*

Terms of the form $\mathit{ins\_list}\ x\ (\mathit{xs}_1\ @\ a_1\ \#\ \mathit{xs}_2\ @\ a_2\ \#\ \ldots\ \#\ \mathit{xs}_n)$ are rewritten with equation (5.5) (and the defining equations for *ins_list*) to push *ins_list* inwards. Terms of the form $\mathit{del\_list}\ x\ (\mathit{xs}_1\ @\ a_1\ \#\ \mathit{xs}_2\ @\ a_2\ \#\ \ldots\ \#\ \mathit{xs}_n)$ are rewritten with equation (5.6) (and the defining equations for *del_list*) to push *del_list* inwards.

The *isin* property in Figure 5.1 can be proved with the help of (5.1), (5.7) and (5.8).

The lemmas in Figure 5.2 form the complete set of basic lemmas on which the automatic proofs of almost all search trees in the book rest; only splay trees (see Chapter 21) need additional lemmas.

5.4.3 Sorted Lists Implement Sets

It remains to be shown that the list-based specification (Figure 5.1) implies the set-based correctness properties in Section 5.3. Remember that $\mathit{bst}\ t = \mathit{sorted}\ (\mathit{inorder}\ t)$. The correctness properties

$$\begin{aligned}&\mathit{set\_tree}\ \mathit{empty} = \{\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \mathit{set\_tree}\ t \cup \{x\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{delete}\ x\ t) = \mathit{set\_tree}\ t - \{x\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{isin}\ t\ x = (x \in \mathit{set\_tree}\ t)\end{aligned}$$

are a consequence of $\mathit{set\_tree}\ t = \mathit{set}\ (\mathit{inorder}\ t)$ (a basic tree lemma), the properties in Figure 5.1 and the inductive correctness properties

$$\begin{aligned}&\mathit{set}\ (\mathit{ins\_list}\ x\ \mathit{xs}) = \mathit{set}\ \mathit{xs} \cup \{x\}\\ &\mathit{sorted}\ \mathit{xs} \longrightarrow \mathit{set}\ (\mathit{del\_list}\ x\ \mathit{xs}) = \mathit{set}\ \mathit{xs} - \{x\}\end{aligned}$$

Preservation of the invariant

$$\mathit{sorted}\ (\mathit{inorder}\ \mathit{empty})$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{insert}\ x\ t))$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{delete}\ x\ t))$$

are trivial or consequences of the properties in Figure 5.1 and the preservation of *sorted* by *ins_list* and *del_list*:

$$\mathit{sorted}\ \mathit{xs} \longrightarrow \mathit{sorted}\ (\mathit{ins\_list}\ x\ \mathit{xs})$$
$$\mathit{sorted}\ \mathit{xs} \longrightarrow \mathit{sorted}\ (\mathit{del\_list}\ x\ \mathit{xs})$$

This means in particular that preservation of $\mathit{sorted} \circ \mathit{inorder}$ is guaranteed for any implementation of *empty*, *insert* and *delete* that satisfies the list-based specification in Figure 5.1.

5.5 Interval Trees

In this section we study binary trees for representing a set of intervals, called **interval trees**. In addition to the usual insertion and deletion functions of standard BSTs, interval trees support a function for determining whether a given interval overlaps with some interval in the tree.

5.5.1 Augmented BSTs

The efficient implementation of the search for an overlapping interval relies on an additional piece of information in each node. Thus interval trees are another example of augmented trees as introduced in Section 4.4. We reuse the modified definitions of *set_tree* and *inorder* from that section. Moreover we use a slightly adjusted version of *isin*:

$$\mathit{isin} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'a \Rightarrow \mathit{bool}$$

$$\mathit{isin}\ \langle\rangle\ \_ = \mathit{False}$$
$$\mathit{isin}\ \langle l,\ (a,\ \_),\ r\rangle\ x$$
$$= (\textbf{case}\ \mathit{cmp}\ x\ a\ \textbf{of}\ \mathit{LT} \Rightarrow \mathit{isin}\ l\ x \mid \mathit{EQ} \Rightarrow \mathit{True} \mid \mathit{GT} \Rightarrow \mathit{isin}\ r\ x)$$

This works for any kind of augmented BST, not just interval trees.

5.5.2 Intervals

An interval *'a ivl* is simply a pair of lower and upper bound, accessed by functions *low* and *high*, respectively. Intuitively, an interval represents the closed set between *low* and *high*. The standard mathematical notation is $[l, h]$, the Isabelle notation is $\{l..h\}$. We restrict ourselves to non-empty intervals:

$$\mathit{low}\ p \le \mathit{high}\ p$$

Type $'a$ can be any linearly ordered type with a minimum element $\bot$ (for example, the natural numbers or the real numbers extended with $-\infty$). Intervals can be linearly ordered by first comparing *low*, then comparing *high*. The definitions are as follows:

$$(x < y) = (\mathit{low}\ x < \mathit{low}\ y \vee \mathit{low}\ x = \mathit{low}\ y \wedge \mathit{high}\ x < \mathit{high}\ y)$$
$$(x \le y) = (\mathit{low}\ x < \mathit{low}\ y \vee \mathit{low}\ x = \mathit{low}\ y \wedge \mathit{high}\ x \le \mathit{high}\ y)$$

Two intervals overlap if they have at least one point in common:

$$\mathit{overlap}\ x\ y = (\mathit{low}\ y \le \mathit{high}\ x \wedge \mathit{low}\ x \le \mathit{high}\ y)$$

The readers should convince themselves that *overlap* does what it is supposed to do: $\mathit{overlap}\ x\ y = (\{\mathit{low}\ x..\mathit{high}\ x\} \cap \{\mathit{low}\ y..\mathit{high}\ y\} \neq \{\})$

We also define the concept of an interval overlapping with some interval in a set:

$$\mathit{has\_overlap}\ S\ y = (\exists\, x{\in}S.\ \mathit{overlap}\ x\ y)$$

5.5.3 Interval Trees

An interval tree associates to each node a number *max_hi*, which records the maximum *high* value of all intervals in the subtrees. This value is updated during insert and delete operations, and will be crucial for enabling efficient determination of overlap with some interval in the tree.

type_synonym $'a\ \mathit{ivl\_tree} = ('a\ \mathit{ivl} \times\ 'a)\ \mathit{tree}$

$\mathit{max\_hi} :: {'a}\ \mathit{ivl\_tree} \Rightarrow {'a}$

$\mathit{max\_hi}\ \langle\rangle = \bot$
$\mathit{max\_hi}\ \langle\_,\ (\_,\ m),\ \_\rangle = m$

If the *max_hi* value of every node in a tree agrees with *max*3

```
inv_max_hi :: 'a ivl_tree ⇒ bool
inv_max_hi ⟨⟩ = True
inv_max_hi ⟨l, (a, m), r⟩
= (m = max3 a (max_hi l) (max_hi r) ∧ inv_max_hi l ∧
   inv_max_hi r)

max3 :: 'a ivl ⇒ 'a ⇒ 'a ⇒ 'a
max3 a m n = max (high a) (max m n)
```

it follows by induction that *max_hi* is the maximum value of *high* in the tree and comes from some node in the tree:

Lemma 5.1. $inv\_max\_hi\ t \wedge a \in set\_tree\ t \longrightarrow high\ a \leq max\_hi\ t$

Lemma 5.2. $inv\_max\_hi\ t \wedge t \neq \langle\rangle \longrightarrow$
$(\exists\, a{\in}set\_tree\ t.\ high\ a = max\_hi\ t)$

5.5.4 Implementing Sets of Intervals via Interval Trees

Interval trees can implement sets of intervals via unbalanced BSTs as in Section 5.2. Of course *empty* = ⟨⟩. Function *isin* was already defined in Section 5.5.1 Insertion and deletion are also very close to the versions in Section 5.2, but the value of *max_hi* must be computed (by *max3*) for each new node. We follow Section 4.4 and introduce a smart constructor *node* for that purpose and replace ⟨*l*, *a*, *r*⟩ by *node l a r* (on the right-hand side):

```
node :: 'a ivl_tree ⇒ 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
node l a r = ⟨l, (a, max3 a (max_hi l) (max_hi r)), r⟩

insert :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
insert x ⟨⟩ = ⟨⟨⟩, (x, high x), ⟨⟩⟩
insert x ⟨l, (a, m), r⟩ = (case cmp x a of
                             LT ⇒ node (insert x l) a r |
                             EQ ⇒ ⟨l, (a, m), r⟩ |
                             GT ⇒ node l a (insert x r))

split_min :: 'a ivl_tree ⇒ 'a ivl × 'a ivl_tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (x, l') = split_min l in (x, node l' a r))

delete :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ node (delete x l) a r |
    EQ ⇒ if r = ⟨⟩ then l else let (x, y) = split_min r in node l x y |
    GT ⇒ node l a (delete x r))
```

The correctness proofs for insertion and deletion cover two aspects. Functional correctness and preservation of the invariant *sorted* ∘ *inorder* (the BST property) are proved exactly as in Section 5.3 for ordinary BSTs. Preservation of the invariant *inv_max_hi* can be proved by a sequence of simple inductive properties. In the end the main correctness properties are

$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{insert}\ x\ t) = \mathit{ins\_list}\ x\ (\mathit{inorder}\ t)$$
$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{delete}\ x\ t) = \mathit{del\_list}\ x\ (\mathit{inorder}\ t)$$
$$\mathit{inv\_max\_hi}\ t \longrightarrow \mathit{inv\_max\_hi}\ (\mathit{insert}\ x\ t)$$
$$\mathit{inv\_max\_hi}\ t \longrightarrow \mathit{inv\_max\_hi}\ (\mathit{delete}\ x\ t)$$

Defining $\mathit{invar}\ t = (\mathit{inv\_max\_hi}\ t \wedge \mathit{sorted}\ (\mathit{inorder}\ t))$ we obtain the following top-level correctness corollaries:

$$\mathit{invar}\ s \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ s) = \mathit{set\_tree}\ s \cup \{x\}$$
$$\mathit{invar}\ s \longrightarrow \mathit{set\_tree}\ (\mathit{delete}\ x\ s) = \mathit{set\_tree}\ s - \{x\}$$
$$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ s)$$
$$\mathit{invar}\ s \longrightarrow \mathit{invar}\ (\mathit{delete}\ x\ s)$$

5.5.5 Searching for an Overlapping Interval

The added functionality of interval trees over ordinary BSTs is function *search* that searches for an overlapping rather than identical interval:

```
search :: 'a ivl_tree ⇒ 'a ivl ⇒ bool
search ⟨⟩ _ = False
search ⟨l, (a, _), r⟩ x
= (if overlap x a then True
    else if l ≠ ⟨⟩ ∧ low x ≤ max_hi l then search l x else search r x)
```

The following theorem expresses the correctness of *search* assuming the same invariants as before; *bst t* would work just as well as *sorted* (*inorder t*).

Theorem 5.3. $inv\_max\_hi\ t \wedge sorted\ (inorder\ t) \longrightarrow search\ t\ x = has\_overlap\ (set\_tree\ t)\ x$

Proof. The result is clear when t is $\langle\rangle$. Now suppose t is in the form $\langle l, (a, m), r\rangle$, where m is the value of *max_hi* at root. If a overlaps with x, search returns *True* as expected. Otherwise, there are two cases.

- If $l \neq \langle\rangle$ and $low\ x \leq max\_hi\ l$, the search goes to the left child. If there is an interval in the left child overlapping with x, then the search returns *True* as expected. Otherwise, we show there is also no interval in the right child overlapping with x. Since $l \neq \langle\rangle$, Lemma 5.2 yields a node p in the left child such that $high\ p = max\_hi\ l$. Since $low\ x \leq max\_hi\ l$, we have $low\ x \leq high\ p$. Since p does not overlap with x, we must have $high\ x < low\ p$. But then, for every interval rp in the right child, $low\ p \leq low\ rp$, so that $high\ x < low\ rp$, which implies that rp does not overlap with x.
- Now we consider the case where either $l = \langle\rangle$ or $max\_hi\ l < low\ x$. In this case, the search goes to the right. We show there is no interval in the left child that overlaps with x. This is clear if $l = \langle\rangle$. Otherwise, for each interval lp, we have $high\ lp \leq max\_hi\ l$ by Lemma 5.1, so that $high\ lp < low\ x$, which means lp does not overlap with x.

□

Exercise 5.4. Define a function that determines if a given point is in some interval in a given interval tree. Starting with

$$in\_ivl :: {'a} \Rightarrow {'a}\ ivl \Rightarrow bool$$

$$in\_ivl\ x\ iv = (low\ iv \leq x \wedge x \leq high\ iv)$$

write a recursive function

$$search1 :: {'a}\ ivl\_tree \Rightarrow {'a} \Rightarrow bool$$

(without using *search*) such that $search1\ x\ t$ is *True* iff there is some interval iv in t such that $in\_ivl\ x\ iv$. Prove

$$inv\_max\_hi\ t \wedge bst\ t \longrightarrow search1\ t\ x = (\exists iv \in set\_tree\ t.\ in\_ivl\ x\ iv)$$

5.5.6 Notes

While this section demonstrated how to augment an ordinary binary tree with intervals, any of the balanced binary trees (such as red-black tree) can be augmented in a similar manner. We leave this as exercises.

Interval trees have many applications in computational geometry. A basic example is as follows. Consider a set of rectangles whose sides are aligned to the x and y-axes. We wish to efficiently determine whether any pair of rectangles in the set intersect each other (i.e. sharing a point, including boundaries). This can be done using a "sweep line" algorithm as follows. For each rectangle $[x_l, x_h] \times [y_l, y_h]$, we create two events: insert interval $[x_l, x_h]$ at y-coordinate y_l and delete interval $[x_l, x_h]$ at y-coordinate y_h. Perform the events, starting from an empty interval tree, in ascending order of y-coordinates, with insertion events performed before deletion events. At each insertion, check whether the interval to be inserted overlaps with any of the existing intervals in the tree. If yes, we have found an intersection between two rectangles. If no overlap of intervals is detected throughout the process, then no pair of rectangles intersect. When using an interval tree based on a balanced binary tree, the time complexity of this procedure is $O(n \lg n)$, where n is the number of rectangles.

We refer to Cormen *et al.* [13, Section 14.3] for another exposition on interval trees and their applications. Interval trees, together with the application of finding rectangle intersection, have been formalized by Zhan [81].

6

Abstract Data Types

In the previous chapter we looked at a very specific example of an abstract data type, namely sets. In this chapter we consider abstract data types in general and in particular the model-oriented approach to the specification of abstract data types. This will lead to a generic format for such specifications.

6.1 Abstract Data Types

Abstract data types (ADTs) can be summarized by the following slogan:

$$\text{ADT} = \textit{interface} + \textit{specification}$$

where the interface lists the operations supported by the ADT and the specification describes the behaviour of these operations. For example, our set ADT has the following interface:

$\textit{empty} :: {'s}$
$\textit{insert} :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$\textit{delete} :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$\textit{isin} :: {'s} \Rightarrow {'a} \Rightarrow \textit{bool}$

The purpose of an ADT is to be able to write applications based on this ADT that will work with any implementation of the ADT. To this end one can prove properties of the application that are solely based on the specification of the ADT. That is, one can write generic algorithms and prove generic correctness theorems about them in the context of the ADT specification.

6.2 Model-Oriented Specifications

We follow the model-oriented style of specification [35]. In that style, an abstract type is specified by giving an abstract model for it. For simplicity we assume that each ADT describes one **type of interest** T. In the set interface T is $'s$. This type T must be specified by some existing HOL type A, the abstract model. In the case of sets this is straightforward: the model for sets is simply the HOL type $'a\ set$. The motto is that T should behave like A. In order to bridge the gap between the two types, the specification needs an

- **abstraction function** $\alpha :: T \Rightarrow A$

that maps concrete values to their abstract counterparts. Moreover, in general only some elements of T represent elements of A. For example, in the set implementation in the previous chapter not all trees but only BSTs represent sets. Thus the specification should also take into account an

- **invariant** $invar :: T \Rightarrow bool$

Note that the abstraction function and the invariant are not part of the interface, but they are essential for specification and verification purposes.

As an example, the ADT of sets is shown in Figure 6.1 with suggestive keywords and a fixed mnemonic naming schema for the labels in the specification. This is the template for ADTs that we follow throughout the book.

ADT *Set* =

interface
$empty :: 's$
$insert :: 'a \Rightarrow 's \Rightarrow 's$
$delete :: 'a \Rightarrow 's \Rightarrow 's$
$isin :: 's \Rightarrow 'a \Rightarrow bool$

abstraction $set :: 's \Rightarrow 'a\ set$
invariant $invar :: 's \Rightarrow bool$

specification
$invar\ empty$ (*empty-inv*)
$set\ empty = \{\}$ (*empty*)
$invar\ s \longrightarrow invar\ (insert\ x\ s)$ (*insert-inv*)
$invar\ s \longrightarrow set(insert\ x\ s) = set\ s \cup \{x\}$ (*insert*)
$invar\ s \longrightarrow invar\ (delete\ x\ s)$ (*delete-inv*)
$invar\ s \longrightarrow set\ (delete\ x\ s) = set\ s - \{x\}$ (*delete*)
$invar\ s \longrightarrow isin\ s\ x = (x \in set\ s)$ (*isin*)

Fig. 6.1. ADT *Set*

We have intentionally refrained from showing the Isabelle formalization using so-called **locales** and have opted for a more intuitive textual format that is not Isabelle-specific, in accordance with the general philosophy of this book.

The actual Isabelle text can of course be found in the source files, and locales are explained in a dedicated manual [5].

We conclude this section by explaining what the specification of an arbitrary ADT looks like. We assume that for each function f of the interface there is a corresponding function f_A in the abstract model, i.e. defined on A. For a uniform treatment we extend α and *invar* to arbitrary types by setting $\alpha\ x = x$ and $\mathit{invar}\ x = \mathit{True}$ for all types other than T. Each function f of the interface gives rise to two properties in the specification: preservation of the invariant and simulation of f_A. The precondition is shared:

$$
\begin{array}{ll}
\mathit{invar}\ x_1 \land \ldots \land \mathit{invar}\ x_n \longrightarrow & \\
\quad \mathit{invar}(f\ x_1\ \ldots\ x_n) & (f\text{-}\mathit{inv}) \\
\quad \alpha(f\ x_1\ \ldots\ x_n) = f_A\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n) & (f)
\end{array}
$$

To understand how the specification of ADT *Set* is the result of this uniform schema one has to take two things into account:

- Precisely which abstract operations on type *'a set* model the functions in the interface of the ADT *Set*? This correspondence is implicit in the specification: *empty* is modeled by $\{\}$, *insert* is modeled by $\lambda x\ s.\ s \cup \{x\}$, *delete* is modeled by $\lambda x\ s.\ s - \{x\}$ and *isin* is modeled by $\lambda s\ x.\ x \in s$.
- Because of the artificial extension of α and *invar* the above uniform format often collapses to something simpler where some α's and *invar*'s disappear.

6.3 Maps

An even more versatile data structure than sets are (efficient) maps from *'a* to *'b*. In fact, sets can be viewed as maps from *'a* to *bool*. Conversely, many data structures for sets also support maps, e.g. BSTs. Although, for simplicity, we focus on sets in this book, the ADT of maps should at least be introduced. It is shown in Figure 6.2. Type *'m* is the type of maps from *'a* to *'b*. The ADT *Map* is very similar to the ADT *Set* except that the abstraction function *lookup* is also part of the interface: it abstracts a map to a function of type $'a \Rightarrow\ 'b\ \mathit{option}$. This implies that the equations are between functions of that type. We use the function update notation (see Section 1.3) to explain *update* and *delete*: *update* is modeled by $\lambda m\ a\ b.\ m(a := b)$ and *delete* by $\lambda m\ a.\ m(a := \mathit{None})$.

ADT $\mathit{Map} =$

interface
$\mathit{empty} :: {'m}$
$\mathit{update} :: {'a} \Rightarrow {'b} \Rightarrow {'m} \Rightarrow {'m}$
$\mathit{delete} :: {'a} \Rightarrow {'m} \Rightarrow {'m}$
$\mathit{lookup} :: {'m} \Rightarrow {'a} \Rightarrow {'b}\ \mathit{option}$

abstraction *lookup*
invariant $\mathit{invar} :: {'m} \Rightarrow \mathit{bool}$

specification
$\mathit{invar}\ \mathit{empty}$ $\quad(\mathit{empty\text{-}inv})$
$\mathit{lookup}\ \mathit{empty} = (\lambda a.\ \mathit{None})$ $\quad(\mathit{empty})$
$\mathit{invar}\ m \longrightarrow \mathit{invar}\ (\mathit{update}\ a\ b\ m)$ $\quad(\mathit{update\text{-}inv})$
$\mathit{invar}\ m \longrightarrow \mathit{lookup}\ (\mathit{update}\ a\ b\ m) = (\mathit{lookup}\ m)(a := \mathit{Some}\ b)$ $\quad(\mathit{update})$
$\mathit{invar}\ m \longrightarrow \mathit{invar}\ (\mathit{delete}\ a\ m)$ $\quad(\mathit{delete\text{-}inv})$
$\mathit{invar}\ m \longrightarrow \mathit{lookup}\ (\mathit{delete}\ a\ m) = (\mathit{lookup}\ m)(a := \mathit{None})$ $\quad(\mathit{delete})$

Fig. 6.2. ADT *Map*

6.4 Implementing ADTs

An implementation of an ADT consists of definitions for all the functions in the interface. If you want to verify the correctness of the implementation, you also need to provide definitions for the abstraction function and the invariant. The latter two need not be executable unless they also occur in the interface and the implementation is meant to be executable. The abstraction function need not be surjective. For example, implementations of the ADT *Set* will normally only represent finite sets, e.g. by BSTs.

For Isabelle users: because ADTs are formalized as locales, an implementation of an ADT is an interpretation of the corresponding locale.

6.5 Exercises

Exercise 6.1. Modify the ADT *Set* specification by making *isin* the abstraction function (from ${'s}$ to ${'a} \Rightarrow \mathit{bool}$). Follow the example of the ADT *Map* specification.

Exercise 6.2. In the ADT *Map* specification, following the general schema, there should be a property labeled (*lookup*), but it is missing. The reason is that given the correct abstract model of *lookup*, the equation becomes trivial: $\mathit{invar}\ m \longrightarrow \mathit{lookup}\ m\ a = \mathit{lookup}\ m\ a$. Why is that, which function models *lookup*?

Exercise 6.3. Implement ADT *Map* via unbalanced BSTs (like *Set* in Chapter 5) using augmented trees. Verify the implementation by proving all the correctness properties in the specification of ADT *Map* directly, without any detour via sorted lists as in Section 5.4.

7

2-3 Trees

This is the first in a series of chapters examining **balanced search trees** where the height of the tree is logarithmic in its size and which can therefore be searched in logarithmic time.

The most popular first example of balanced search trees are red-black trees. We start with 2-3 trees, where nodes can have 2 or 3 children, because red-black trees simulate 2-3 trees. We introduce red-black trees in the next chapter. The type of 2-3 trees is similar to the binary trees but with an additional constructor *Node*3:

```
datatype 'a tree23 =
  Leaf |
  Node2 ('a tree23) 'a ('a tree23) |
  Node3 ('a tree23) 'a ('a tree23) 'a ('a tree23)
```

The familiar syntactic sugar is sprinkled on top:

$$\begin{aligned} \langle\rangle &\equiv \mathit{Leaf} \\ \langle l,\ a,\ r\rangle &\equiv \mathit{Node}2\ l\ a\ r \\ \langle l,\ a,\ m,\ b,\ r\rangle &\equiv \mathit{Node}3\ l\ a\ m\ b\ r \end{aligned}$$

The size, height and the completeness of a 2-3 tree are defined by adding an equation for *Node*3 to the corresponding definitions on binary trees:

$$|\langle l,\ \_,\ m,\ \_,\ r\rangle| = |l| + |m| + |r| + 1$$

```
h ⟨l, _, m, _, r⟩ = max (h l) (max (h m) (h r)) + 1

complete ⟨l, _, m, _, r⟩
= (h l = h m ∧ h m = h r ∧ complete l ∧ complete m ∧
   complete r)
```

A trivial induction yields $complete\ t \longrightarrow 2^{h\ t} \leq |t| + 1$: thus all operations on complete 2-3 trees have logarithmic complexity if they descend along a single branch and take constant time per node. This is the case and we will not discuss complexity in any more detail.

A nice property of 2-3 trees is that for every n there is a complete 2-3 tree of size n. As we will see below, completeness can be maintained under insertion and deletion in logarithmic time.

Exercise 7.1. Define a function $maxt :: nat \Rightarrow unit\ tree23$ that creates the tree with the largest number of nodes given the height of the tree. We use type *unit* because we are not interested in the elements in the tree. Prove $|maxt\ n| = (3^n - 1)\ div\ 2$ and that no tree of the given height can be larger: $|t| \leq (3^{h\ t} - 1)\ div\ 2$. Note that both subtraction and division on type *nat* can be tedious to work with. You may want to prove the two properties as corollaries of subtraction and division free properties. Alternatively, work with *real* instead of *nat* by replacing *div* by /.

7.1 Implementation of ADT *Set*

The implementation will maintain the usual ordering invariant and completeness. When we speak of a 2-3 tree we will implicitly assume these two invariants now.

Searching a 2-3 tree is like searching a binary tree (see Section 5.2) but with one more defining equation:

```
isin ⟨l, a, m, b, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True
   | GT ⇒ case cmp x b of LT ⇒ isin m x | EQ ⇒ True
               | GT ⇒ isin r x)
```

Insertion into a 2-3 tree must preserve the completeness invariant. Thus recursive calls must report back to the caller if the child has “overflown”, i.e. increased in height. Therefore insertion returns a result of type *'a upI*:

```
datatype 'a upI = TI ('a tree23) | OF ('a tree23) 'a ('a tree23)
```

This is the idea: If insertion into t returns

TI t' then t and t' should have the same height,
OF l x r then t and l and r should have the same height.

The insertion functions are shown in Figure 7.1. The actual work is performed by the recursive function *ins*. The element to be inserted is propagated down to a leaf, which causes an overflow of the leaf. If an overflow is returned from a recursive call it can be absorbed into a *Node*2 but in a *Node*3 it causes another overflow. At the root of the tree, function *treeI* converts values of type *'a upI* back into trees:

```
treeI :: 'a upI ⇒ 'a tree23
treeI (TI t) = t
treeI (OF l a r) = ⟨l, a, r⟩
```

Deletion is dual. Recursive calls must report back to the caller if the child has "underflown", i.e. decreased in height. Therefore deletion returns a result of type *'a upD*:

```
datatype 'a upD = TD ('a tree23) | UF ('a tree23)
```

This is the idea: If deletion from t returns

TD t' then t and t' should have the same height,
UF t' then t should be one level higher than t'.

The main deletion functions are shown in Figure 7.2. The actual work is performed by the recursive function *del*. If the element to be deleted is in a child, the result of a recursive call is reintegrated into the node via the auxiliary functions *nodeij* from Figure 7.3 that create a node with i children where child j is given as an *'a upD* value. If the element to be deleted is in the node itself, a replacement is obtained and deleted from a child via *split_min*. At the root of the tree, values of type *'a upD* are converted back into trees:

```
treeD :: 'a upD ⇒ 'a tree23
treeD (TD t) = t
treeD (UF t) = t
```

```
insert x t = treeI (ins x t)

ins :: 'a ⇒ 'a tree23 ⇒ 'a upI
ins x ⟨⟩ = OF ⟨⟩ x ⟨⟩
ins x ⟨l, a, r⟩
= (case cmp x a of
     LT ⇒ case ins x l of TI l' ⇒ TI ⟨l', a, r⟩
                | OF $l_1$ b $l_2$ ⇒ TI ⟨$l_1$, b, $l_2$, a, r⟩
   | EQ ⇒ TI ⟨l, a, r⟩
   | GT ⇒ case ins x r of TI r' ⇒ TI ⟨l, a, r'⟩
                | OF $r_1$ b $r_2$ ⇒ TI ⟨l, a, $r_1$, b, $r_2$⟩)
ins x ⟨l, a, m, b, r⟩
= (case cmp x a of
     LT ⇒ case ins x l of TI l' ⇒ TI ⟨l', a, m, b, r⟩
                | OF $l_1$ c $l_2$ ⇒ OF ⟨$l_1$, c, $l_2$⟩ a ⟨m, b, r⟩
   | EQ ⇒ TI ⟨l, a, m, b, r⟩
   | GT ⇒ case cmp x b of
              LT ⇒ case ins x m of TI m' ⇒ TI ⟨l, a, m', b, r⟩
                         | OF $m_1$ c $m_2$ ⇒ OF ⟨l, a, $m_1$⟩ c ⟨$m_2$, b, r⟩
            | EQ ⇒ TI ⟨l, a, m, b, r⟩
            | GT ⇒ case ins x r of TI r' ⇒ TI ⟨l, a, m, b, r'⟩
                         | OF $r_1$ c $r_2$ ⇒ OF ⟨l, a, m⟩ b ⟨$r_1$, c, $r_2$⟩)
```

Fig. 7.1. Insertion into 2-3 tree

7.2 Preservation of Completeness

As explained in Section 5.4, we do not go into the automatic functional correctness proofs but concentrate on invariant preservation. To express the relationship between the height of a tree before and after insertion we define a height function *hI* on *'a upI* as follows:

$$hI\ (TI\ t) = h\ t$$
$$hI\ (OF\ l\ a\ r) = h\ l$$

The intuition is that *hI* is the height of the tree *before* insertion. A routine induction proves

$$complete\ t \longrightarrow complete\ (treeI\ (ins\ a\ t)) \wedge hI\ (ins\ a\ t) = h\ t$$

which implies by definition that

$$complete\ t \longrightarrow complete\ (insert\ a\ t)$$

```
delete x t = treeD (del x t)

del :: 'a ⇒ 'a tree23 ⇒ 'a upD
del x ⟨⟩ = TD ⟨⟩
del x ⟨⟨⟩, a, ⟨⟩⟩ = (if x = a then UF ⟨⟩ else TD ⟨⟨⟩, a, ⟨⟩⟩)
del x ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩
= TD (if x = a then ⟨⟨⟩, b, ⟨⟩⟩
        else if x = b then ⟨⟨⟩, a, ⟨⟩⟩ else ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩)
del x ⟨l, a, r⟩
= (case cmp x a of LT ⇒ node21 (del x l) a r
   | EQ ⇒ let (a', r') = split_min r in node22 l a' r'
   | GT ⇒ node22 l a (del x r))
del x ⟨l, a, m, b, r⟩
= (case cmp x a of LT ⇒ node31 (del x l) a m b r
   | EQ ⇒ let (a', m') = split_min m in node32 l a' m' b r
   | GT ⇒ case cmp x b of LT ⇒ node32 l a (del x m) b r
                | EQ ⇒ let (b', r') = split_min r in node33 l a m b' r'
                | GT ⇒ node33 l a m b (del x r))

split_min :: 'a tree23 ⇒ 'a × 'a upD
split_min ⟨⟨⟩, a, ⟨⟩⟩ = (a, UF ⟨⟩)
split_min ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩ = (a, TD ⟨⟨⟩, b, ⟨⟩⟩)
split_min ⟨l, a, r⟩ = (let (x, l') = split_min l in (x, node21 l' a r))
split_min ⟨l, a, m, b, r⟩
= (let (x, l') = split_min l in (x, node31 l' a m b r))
```

Fig. 7.2. Deletion from 2-3 tree: main functions

The fact that deletion preserves completeness can be proved by a sequence of small lemmas. To express the relationship between the height of a tree before and after deletion we define a height function *hD* on *'a upD* as follows:

```
hD (TD t) = h t
hD (UF t) = h t + 1
```

The intuition is that *hD* is the height of the tree *before* deletion. We now list a sequence of properties that build on each other and culminate in completeness preservation of *delete*:

$$\mathit{complete}\ r \wedge \mathit{complete}\ (\mathit{treeD}\ l') \wedge h\ r = \mathit{hD}\ l' \longrightarrow$$
$$\mathit{complete}\ (\mathit{treeD}\ (\mathit{node21}\ l'\ a\ r))$$

$$0 < h\ r \longrightarrow \mathit{hD}\ (\mathit{node21}\ l'\ a\ r) = \mathit{max}\ (\mathit{hD}\ l')\ (h\ r) + 1$$

$$\begin{array}{l}
\mathit{node21} :: {'a}\ \mathit{upD} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree23} \Rightarrow {'a}\ \mathit{upD} \\
\mathit{node21}\ (\mathit{TD}\ t_1)\ a\ t_2 = \mathit{TD}\ \langle t_1,\ a,\ t_2\rangle \\
\mathit{node21}\ (\mathit{UF}\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle = \mathit{UF}\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node21}\ (\mathit{UF}\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle = \mathit{TD}\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node22} :: {'a}\ \mathit{tree23} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{upD} \Rightarrow {'a}\ \mathit{upD} \\
\mathit{node22}\ t_1\ a\ (\mathit{TD}\ t_2) = \mathit{TD}\ \langle t_1,\ a,\ t_2\rangle \\
\mathit{node22}\ \langle t_1,\ b,\ t_2\rangle\ a\ (\mathit{UF}\ t_3) = \mathit{UF}\ \langle t_1,\ b,\ t_2,\ a,\ t_3\rangle \\
\mathit{node22}\ \langle t_1,\ b,\ t_2,\ c,\ t_3\rangle\ a\ (\mathit{UF}\ t_4) = \mathit{TD}\ \langle\langle t_1,\ b,\ t_2\rangle,\ c,\ \langle t_3,\ a,\ t_4\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node31} :: {'a}\ \mathit{upD} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree23} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree23} \Rightarrow {'a}\ \mathit{upD} \\
\mathit{node31}\ (\mathit{TD}\ t_1)\ a\ t_2\ b\ t_3 = \mathit{TD}\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node31}\ (\mathit{UF}\ t_1)\ a\ \langle t_2,\ b,\ t_3\rangle\ c\ t_4 = \mathit{TD}\ \langle\langle t_1,\ a,\ t_2,\ b,\ t_3\rangle,\ c,\ t_4\rangle \\
\mathit{node31}\ (\mathit{UF}\ t_1)\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\ d\ t_5 \\
= \mathit{TD}\ \langle\langle t_1,\ a,\ t_2\rangle,\ b,\ \langle t_3,\ c,\ t_4\rangle,\ d,\ t_5\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node32} :: {'a}\ \mathit{tree23} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{upD} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree23} \Rightarrow {'a}\ \mathit{upD} \\
\mathit{node32}\ t_1\ a\ (\mathit{TD}\ t_2)\ b\ t_3 = \mathit{TD}\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle \\
\mathit{node32}\ t_1\ a\ (\mathit{UF}\ t_2)\ b\ \langle t_3,\ c,\ t_4\rangle = \mathit{TD}\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\rangle \\
\mathit{node32}\ t_1\ a\ (\mathit{UF}\ t_2)\ b\ \langle t_3,\ c,\ t_4,\ d,\ t_5\rangle \\
= \mathit{TD}\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3\rangle,\ c,\ \langle t_4,\ d,\ t_5\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{node33} :: {'a}\ \mathit{tree23} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{tree23} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{upD} \Rightarrow {'a}\ \mathit{upD} \\
\mathit{node33}\ l\ a\ m\ b\ (\mathit{TD}\ r) = \mathit{TD}\ \langle l,\ a,\ m,\ b,\ r\rangle \\
\mathit{node33}\ t_1\ a\ \langle t_2,\ b,\ t_3\rangle\ c\ (\mathit{UF}\ t_4) = \mathit{TD}\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\rangle \\
\mathit{node33}\ t_1\ a\ \langle t_2,\ b,\ t_3,\ c,\ t_4\rangle\ d\ (\mathit{UF}\ t_5) \\
= \mathit{TD}\ \langle t_1,\ a,\ \langle t_2,\ b,\ t_3\rangle,\ c,\ \langle t_4,\ d,\ t_5\rangle\rangle
\end{array}$$

Fig. 7.3. Deletion from 2-3 tree: auxiliary functions

$$\begin{array}{l}
\mathit{split\_min}\ t = (x,\ t') \wedge 0 < h\ t \wedge \mathit{complete}\ t \longrightarrow \mathit{hD}\ t' = h\ t \\
\mathit{split\_min}\ t = (x,\ t') \wedge \mathit{complete}\ t \wedge 0 < h\ t \longrightarrow \mathit{complete}\ (\mathit{treeD}\ t') \\
\mathit{complete}\ t \longrightarrow \mathit{hD}\ (\mathit{del}\ x\ t) = h\ t \\
\mathit{complete}\ t \longrightarrow \mathit{complete}\ (\mathit{treeD}\ (\mathit{del}\ x\ t)) \\
\mathit{complete}\ t \longrightarrow \mathit{complete}\ (\mathit{delete}\ x\ t)
\end{array}$$

For each property of *node*21 there are analogues properties for the other *nodeij* functions which we omit.

7.3 Converting a List into a 2-3 Tree

The naive method of converting a whole list of elements into a 2-3 tree is to insert them one by one starting from the empty tree. However, that takes time $\Theta(n \lg n)$. This holds for any kind of data structure where insertion takes time proportional to $\lg n$. In that case inserting n elements one by one takes time proportional to $\lg 1 + \cdots + \lg n = \lg(n!)$. Now $n! \leq n^n$ implies $\lg(n!) \leq n \lg n$. On the other hand, $n^n \leq (n \cdot 1) \cdot ((n-1) \cdot 2) \cdots (1 \cdot n) = (n!)^2$ implies $\frac{1}{2} n \lg n \leq \lg(n!)$. Thus $\lg(n!) \in \Theta(n \lg n)$. We have intentionally proved a Θ property because the O property is obvious but one might hope that $\lg 1 + \cdots + \lg n$ has a lower order of growth than $n \lg n$.

Luckily we can convert a whole list into a 2-3 tree in linear time. The bottom-up algorithm is particularly intuitive. It repeatedly passes over on an alternating list $t_1, e_1, t_2, e_2, \ldots, t_k$ of trees and elements, combining trees and elements into new trees. Given elements $a_1, \ldots, a_n$ we start with the alternating list $\langle\rangle, a_1, \langle\rangle, a_2, \ldots, a_n, \langle\rangle$. On every pass over this list, we replace adjacent triples t, a, t' by $\langle t,\ a,\ t' \rangle$, possibly creating a 3-node instead of a 2-node at the end of the list. Once a single tree is left over we terminate.

We define this type of alternating (and non-empty) list as a new data type:

```
datatype 'a tree23s = T ('a tree23) | TTs ('a tree23) 'a ('a tree23s)
```

The following examples demonstrate the encoding of alternating lists as terms of type *'a tree23s*:

Alternating list:	t_1	t_1, e_1, t_2	t_1, e_1, t_2, e_2, ts
Encoding:	$T\ t_1$	$TTs\ t_1\ e_1\ (T\ t_2)$	$TTs\ t_1\ e_1\ (TTs\ t_2\ e_2\ ts)$

We also need the following auxiliary functions:

```
len :: 'a tree23s ⇒ nat
len (T _) = 1
len (TTs _ _ ts) = len ts + 1

trees :: 'a tree23s ⇒ 'a tree23 set
trees (T t) = {t}
trees (TTs t _ ts) = {t} ∪ trees ts

inorder2 :: 'a tree23s ⇒ 'a list
inorder2 (T t) = inorder t
inorder2 (TTs t a ts) = inorder t @ a # inorder2 ts
```

Repeatedly passing over the alternating list until only a single tree remains is expressed by the following functions:

$$\mathit{join\_all} :: {}'a\ \mathit{tree23s} \Rightarrow {}'a\ \mathit{tree23}$$
$$\mathit{join\_all}\ (T\ t) = t$$
$$\mathit{join\_all}\ \mathit{ts} = \mathit{join\_all}\ (\mathit{join\_adj}\ \mathit{ts})$$

$$\mathit{join\_adj} :: {}'a\ \mathit{tree23s} \Rightarrow {}'a\ \mathit{tree23s}$$
$$\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (T\ t_2)) = T\ \langle t_1,\ a,\ t_2\rangle$$
$$\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (\mathit{TTs}\ t_2\ b\ (T\ t_3))) = T\ \langle t_1,\ a,\ t_2,\ b,\ t_3\rangle$$
$$\mathit{join\_adj}\ (\mathit{TTs}\ t_1\ a\ (\mathit{TTs}\ t_2\ b\ \mathit{ts})) = \mathit{TTs}\ \langle t_1,\ a,\ t_2\rangle\ b\ (\mathit{join\_adj}\ \mathit{ts})$$

Note that *join_adj* is not and does not need to be defined on single trees. We express this precondition with an abbreviation:

$$\mathit{not\_T}\ \mathit{ts} \equiv \forall\, t.\ \mathit{ts} \neq T\ t$$

Also note that *join_all* terminates only because *join_adj* shortens the list:

$$\mathit{not\_T}\ \mathit{ts} \longrightarrow \mathit{len}\ (\mathit{join\_adj}\ \mathit{ts}) < \mathit{len}\ \mathit{ts}$$

In fact, it reduces the length at least by a factor of 2:

$$\mathit{not\_T}\ \mathit{ts} \longrightarrow \mathit{len}\ (\mathit{join\_adj}\ \mathit{ts}) \leq \mathit{len}\ \mathit{ts}\ \mathit{div}\ 2 \tag{7.1}$$

The whole process starts with a list of alternating leaves and elements:

$$\mathit{tree23\_of\_list} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{tree23}$$
$$\mathit{tree23\_of\_list}\ \mathit{as} = \mathit{join\_all}\ (\mathit{leaves}\ \mathit{as})$$

$$\mathit{leaves} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{tree23s}$$
$$\mathit{leaves}\ [] = T\ \langle\rangle$$
$$\mathit{leaves}\ (a\ \#\ \mathit{as}) = \mathit{TTs}\ \langle\rangle\ a\ (\mathit{leaves}\ \mathit{as})$$

7.3.1 Functional Correctness

Functional correctness is easily established. The *inorder* and the completeness properties are proved independently by the following inductive lemmas:

$$\mathit{not\_T}\ \mathit{ts} \longrightarrow \mathit{inorder2}\ (\mathit{join\_adj}\ \mathit{ts}) = \mathit{inorder2}\ \mathit{ts}$$
$$\mathit{inorder}\ (\mathit{join\_all}\ \mathit{ts}) = \mathit{inorder2}\ \mathit{ts}$$
$$\mathit{inorder}\ (\mathit{tree23\_of\_list}\ \mathit{as}) = \mathit{as}$$

$$(\forall\, t{\in}\mathit{trees}\ \mathit{ts}.\ \mathit{complete}\ t \wedge h\ t = n) \wedge \mathit{not\_T}\ \mathit{ts} \longrightarrow$$
$$(\forall\, t{\in}\mathit{trees}\ (\mathit{join\_adj}\ \mathit{ts}).\ \mathit{complete}\ t \wedge h\ t = n + 1)$$
$$(\forall\, t{\in}\mathit{trees}\ \mathit{ts}.\ \mathit{complete}\ t \wedge h\ t = n) \longrightarrow \mathit{complete}\ (\mathit{join\_all}\ \mathit{ts})$$
$$t \in \mathit{trees}\ (\mathit{leaves}\ \mathit{as}) \longrightarrow \mathit{complete}\ t \wedge h\ t = 0$$
$$\mathit{complete}\ (\mathit{tree}23\mathit{\_of\_list}\ \mathit{as})$$

7.3.2 Running Time Analysis

Why does the conversion take linear time? Because the first pass over an alternating list of length n takes n steps, the next pass $n/2$ steps, the next pass $n/4$ steps, etc, and this sums up to $2n$. The canonical time functions for the formal proof are shown in Appendix B.3. The following upper bound is easily proved by induction on the computation of *join_adj*:

$$\mathit{not\_T}\ \mathit{ts} \longrightarrow T_{\mathit{join\_adj}}\ \mathit{ts} \le \mathit{len}\ \mathit{ts}\ \mathit{div}\ 2 \tag{7.2}$$

An upper bound $T_{\mathit{join\_all}}\ \mathit{ts} \le 2 \cdot \mathit{len}\ \mathit{ts}$ follows by induction on the computation of *join_adj*. We focus on the induction step:

$$\begin{aligned}
& T_{\mathit{join\_all}}\ \mathit{ts} \\
&= T_{\mathit{join\_adj}}\ \mathit{ts} + T_{\mathit{join\_all}}\ (\mathit{join\_adj}\ \mathit{ts}) + 1 \\
&\le \mathit{len}\ \mathit{ts}\ \mathit{div}\ 2 + 2 \cdot \mathit{len}\ (\mathit{join\_adj}\ \mathit{ts}) + 1 && \text{using (7.2) and IH} \\
&\le \mathit{len}\ \mathit{ts}\ \mathit{div}\ 2 + 2 \cdot (\mathit{len}\ \mathit{ts}\ \mathit{div}\ 2) + 1 && \text{by (7.1)} \\
&\le 2 \cdot \mathit{len}\ \mathit{ts} && \text{because } 1 \le \mathit{len}\ \mathit{ts}
\end{aligned}$$

Now it is routine to derive

$$T_{\mathit{tree23\_of\_list}}\ \mathit{as} \le 3 \cdot |\mathit{as}| + 4$$

Bibliographic Remarks

The invention of 2-3 trees is credited to Hopcroft in 1970 [13, p. 337]. Equational definitions were given by Hoffmann and O'Donnell [29] (only insertion) and Reade [66]. Our formalisation is based on the teaching material by Turbak [74] and the article by Hinze [28].

8

Red-Black Trees

Red-black trees are a popular implementation technique for BSTs: the code is simpler than for 2-3 trees but guarantees logarithmic height too. The nodes are colored either red or black. Abstractly, red-black trees encode 2-3-4 trees where nodes have between 2 and 4 children. Each 2-3-4 node is encoded by a group of between 2 and 4 colored binary nodes as follows:

$$\begin{aligned}
\langle\rangle &\approx \langle\rangle \\
\langle A,a,B\rangle &\approx \langle A,a,B\rangle \\
\langle A,a,B,b,C\rangle &\approx \langle\langle A,a,B\rangle,b,C\rangle \text{ or } \langle A,a,\langle B,b,C\rangle\rangle \\
\langle A,a,B,b,C,c,D\rangle &\approx \langle\langle A,a,B\rangle,b,\langle C,c,D\rangle\rangle
\end{aligned}$$

Color expresses grouping: a black node is the root of 2-3-4 node, a red node is part of a bigger 2-3-4 node. Thus a red-black tree needs to satisfy the following properties or invariants:

1. The root is black.
2. Every $\langle\rangle$ is considered black.
3. If a node is red, its children are black.
4. All paths from a node to a leaf have the same number of black nodes.

The final property expresses that the corresponding 2-3-4 tree is complete. The last two properties imply that the tree has logarithmic height (see below).

We implement red-black trees as binary trees augmented (see Section 4.4) with a colour tag:

```
datatype color = Red | Black

type_synonym 'a rbt = ('a × color) tree
```

Some new syntactic sugar is sprinkled on top:

$$R\ l\ a\ r \;\equiv\; \langle l,\ (a,\ \mathit{Red}),\ r\rangle$$
$$B\ l\ a\ r \;\equiv\; \langle l,\ (a,\ \mathit{Black}),\ r\rangle$$

The following functions get and set the color of a node:

$$\mathit{color} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{color}$$
$$\mathit{color}\ \langle\rangle = \mathit{Black}$$
$$\mathit{color}\ \langle\_,\ (\_,\ c),\ \_\rangle = c$$

$$\mathit{paint} :: \mathit{color} \Rightarrow {}'a\ \mathit{rbt} \Rightarrow {}'a\ \mathit{rbt}$$
$$\mathit{paint}\ \_\ \langle\rangle = \langle\rangle$$
$$\mathit{paint}\ c\ \langle l,\ (a,\ \_),\ r\rangle = \langle l,\ (a,\ c),\ r\rangle$$

The definition of *color* builds in the property that all leaves are black.

8.1 Invariants

The above informal description of the red-black tree invariants is formalized as the predicate *rbt* which (for reasons of modularity) is split into a color and a height invariant *invc* and *invh*:

$$\mathit{rbt} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{bool}$$
$$\mathit{rbt}\ t = (\mathit{invc}\ t \wedge \mathit{invh}\ t \wedge \mathit{color}\ t = \mathit{Black})$$

The color invariant expresses that red nodes must have black children:

$$\mathit{invc} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{bool}$$
$$\mathit{invc}\ \langle\rangle = \mathit{True}$$
$$\begin{aligned}&\mathit{invc}\ \langle l,\ (\_,\ c),\ r\rangle\\ &= ((c = \mathit{Red} \longrightarrow \mathit{color}\ l = \mathit{Black} \wedge \mathit{color}\ r = \mathit{Black}) \wedge\\ &\quad\ (\mathit{invc}\ l \wedge \mathit{invc}\ r))\end{aligned}$$

The height invariant expresses (via the **black height** *bh*) that all paths from the root to a leaf have the same number of black nodes:

$$\begin{array}{l}
\mathit{invh} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{bool}\\
\mathit{invh}\ \langle\rangle = \mathit{True}\\
\mathit{invh}\ \langle l,\ (\_,\ \_),\ r\rangle = (\mathit{bh}\ l = \mathit{bh}\ r \wedge \mathit{invh}\ l \wedge \mathit{invh}\ r)\\
\\
\mathit{bh} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{nat}\\
\mathit{bh}\ \langle\rangle = 0\\
\mathit{bh}\ \langle l,\ (\_,\ c),\ \_\rangle = (\textbf{if}\ c = \mathit{Black}\ \textbf{then}\ \mathit{bh}\ l + 1\ \textbf{else}\ \mathit{bh}\ l)
\end{array}$$

Note that although *bh* traverses only the left spine of the tree, the fact that *invh* traverses the complete tree ensures that all paths from the root to a leaf are considered. (See Exercise 8.2)

The split of the invariant into *invc* and *invh* improves modularity: frequently one can prove preservation of *invc* and *invh* separately, which facilitates proof search. For compactness we will mostly present the combined invariance properties.

8.1.1 Logarithmic Height

In a red-black tree, i.e. *rbt t*, every path from the root to a leaf has the same number of black nodes, and no such path has two red nodes in a row. Thus each leaf is at most twice as deep as any other leaf, and therefore $h\ t \leq 2 \cdot \lg\ |t|_1$. The detailed proof starts with the key inductive relationship between height and black height

$$\begin{array}{l}
\mathit{invc}\ t \wedge \mathit{invh}\ t \longrightarrow\\
h\ t \leq 2 \cdot \mathit{bh}\ t + (\textbf{if}\ \mathit{color}\ t = \mathit{Black}\ \textbf{then}\ 0\ \textbf{else}\ 1)
\end{array}$$

which has the easy corollary $\mathit{rbt}\ t \longrightarrow h\ t\ /\ 2 \leq \mathit{bh}\ t$. Together with the easy inductive property

$$\mathit{invc}\ t \wedge \mathit{invh}\ t \longrightarrow 2^{\mathit{bh}\ t} \leq |t|_1$$

this implies $2^{h\ t\ /\ 2} \leq 2^{\mathit{bh}\ t} \leq |t|_1$ and thus $h\ t \leq 2 \cdot \lg\ |t|_1$ if *rbt t*.

8.2 Implementation of ADT *Set*

We implement sets by red-black trees that are also BSTs. As usual, we only discuss the proofs of preservation of the *rbt* invariant.

Function *isin* is implemented as for all augmented BSTs (see Section 5.5.1).

```
insert x t = paint Black (ins x t)

ins :: 'a ⇒ 'a rbt ⇒ 'a rbt
ins x ⟨⟩ = R ⟨⟩ x ⟨⟩
ins x (B l a r) = (case cmp x a of
                     LT ⇒ baliL (ins x l) a r |
                     EQ ⇒ B l a r |
                     GT ⇒ baliR l a (ins x r))
ins x (R l a r) = (case cmp x a of
                     LT ⇒ R (ins x l) a r |
                     EQ ⇒ R l a r |
                     GT ⇒ R l a (ins x r))

baliL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliL (R (R t1 a t2) b t3) c t4 = R (B t1 a t2) b (B t3 c t4)
baliL (R t1 a (R t2 b t3)) c t4 = R (B t1 a t2) b (B t3 c t4)
baliL t1 a t2 = B t1 a t2

baliR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliR t1 a (R t2 b (R t3 c t4)) = R (B t1 a t2) b (B t3 c t4)
baliR t1 a (R (R t2 b t3) c t4) = R (B t1 a t2) b (B t3 c t4)
baliR t1 a t2 = B t1 a t2
```

Fig. 8.1. Insertion into red-black tree

8.2.1 Insertion

Insertion is shown in Figure 8.1. The workhorse is function *ins*. It descends to the leaf where the element is inserted and it adjusts the colors on the way back up. The adjustment is performed by *baliL*/*baliR*. They combine arguments $l\ a\ r$ into a tree. If there is a red-red conflict in l/r, they rebalance and replace it by red-black. Inserting into a red node needs no immediate balancing because that will happen at the black node above it:

$$\begin{aligned}
& \mathit{ins}\ 1\ (B\ (R\ \langle\rangle\ 0\ \langle\rangle)\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)) \\
&= \mathit{baliL}\ (\mathit{ins}\ 1\ (R\ \langle\rangle\ 0\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle) \\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (\mathit{ins}\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle) \\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (R\ \langle\rangle\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle) \\
&= R\ (B\ \langle\rangle\ 0\ \langle\rangle)\ 1\ (B\ \langle\rangle\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle))
\end{aligned}$$

Passing a red node up means an overflow occurred (as in 2-3 trees) that needs to be dealt with further up. At the latest at the very top where *insert* turns red into black.

Function *ins* preserves *invh* but not *invc*: it may return a tree with a red root node (see example above). However, once the root node is colored black, everything is fine again. Thus we introduce the weaker invariant *invc*2 as an abbreviation:

$$invc2\ t \equiv invc\ (paint\ Black\ t)$$

It is easy to prove that *baliL* and *baliR* preserve *invh* and upgrade from *invc*2 to *invc*:

$$invh\ l \wedge invh\ r \wedge invc2\ l \wedge invc\ r \wedge bh\ l = bh\ r \longrightarrow$$
$$invc\ (baliL\ l\ a\ r) \wedge invh\ (baliL\ l\ a\ r) \wedge bh\ (baliL\ l\ a\ r) = bh\ l + 1$$
$$invh\ l \wedge invh\ r \wedge invc\ l \wedge invc2\ r \wedge bh\ l = bh\ r \longrightarrow$$
$$invc\ (baliR\ l\ a\ r) \wedge invh\ (baliR\ l\ a\ r) \wedge bh\ (baliR\ l\ a\ r) = bh\ l + 1$$

Another easy induction yields

$$invc\ t \wedge invh\ t \longrightarrow$$
$$invc2\ (ins\ x\ t) \wedge (color\ t = Black \longrightarrow invc\ (ins\ x\ t)) \wedge$$
$$invh\ (ins\ x\ t) \wedge bh\ (ins\ x\ t) = bh\ t$$

The corollary $rbt\ t \longrightarrow rbt\ (insert\ x\ t)$ is immediate.

8.2.2 Deletion

Deletion from a red-black tree is shown in Figure 8.2. It follows the deletion-by-replacing approach (Section 5.2.1). The tricky bit is how to maintain the invariants. As before, intermediate trees may only satisfy the weaker invariant *invc*2. Functions *del* and *split_min* decrease the black height of a tree with a black root node and leave the black height unchanged otherwise. To see that this makes sense, consider deletion from a singleton black or red node. The case that the element to be removed is not in the black tree can be dealt with by coloring the root node red. These are the precise input/output relations:

Lemma 8.1. $split\_min\ t = (x,\ t') \wedge t \neq \langle\rangle \wedge invh\ t \wedge invc\ t \longrightarrow$
$invh\ t' \wedge (color\ t = Red \longrightarrow bh\ t' = bh\ t \wedge invc\ t') \wedge$
$(color\ t = Black \longrightarrow bh\ t' = bh\ t - 1 \wedge invc2\ t')$

Lemma 8.2. $invh\ t \wedge invc\ t \wedge t' = del\ x\ t \longrightarrow$
$invh\ t' \wedge (color\ t = Red \longrightarrow bh\ t' = bh\ t \wedge invc\ t') \wedge$
$(color\ t = Black \longrightarrow bh\ t' = bh\ t - 1 \wedge invc2\ t')$

It is easy to see that the *del*-Lemma implies correctness of *delete*:

Corollary 8.3. $rbt\ t \longrightarrow rbt\ (delete\ x\ t)$

```
delete x t = paint Black (del x t)

del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ let l' = del x l
            in if l ≠ ⟨⟩ ∧ color l = Black then baldL l' a r else R l' a r |
    EQ ⇒ if r = ⟨⟩ then l
            else let (a', r') = split_min r
                   in if color r = Black then baldR l a' r' else R l a' r' |
    GT ⇒ let r' = del x r
            in if r ≠ ⟨⟩ ∧ color r = Black then baldR l a r' else R l a r')

split_min :: 'a rbt ⇒ 'a × 'a rbt
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (x, l') = split_min l
          in (x, if color l = Black then baldL l' a r else R l' a r))

baldL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldL (R t1 a t2) b t3 = R (B t1 a t2) b t3
baldL t1 a (B t2 b t3) = baliR t1 a (R t2 b t3)
baldL t1 a (R (B t2 b t3) c t4) = R (B t1 a t2) b (baliR t3 c (paint Red t4))
baldL t1 a t2 = R t1 a t2

baldR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldR t1 a (R t2 b t3) = R t1 a (B t2 b t3)
baldR (B t1 a t2) b t3 = baliL (R t1 a t2) b t3
baldR (R t1 a (B t2 b t3)) c t4 = R (baliL (paint Red t1) a t2) b (B t3 c t4)
baldR t1 a t2 = R t1 a t2
```

Fig. 8.2. Deletion from red-black tree

The proofs of the two lemmas need the following precise characterizations of *baldL* and *baldR*, the counterparts of *baliL* and *baliR*:

Lemma 8.4. $\mathit{invh}\ l \wedge \mathit{invh}\ r \wedge \mathit{bh}\ l + 1 = \mathit{bh}\ r \wedge \mathit{invc2}\ l \wedge \mathit{invc}\ r \wedge$
$t' = \mathit{baldL}\ l\ a\ r \longrightarrow$
$\mathit{invh}\ t' \wedge \mathit{bh}\ t' = \mathit{bh}\ r \wedge \mathit{invc2}\ t' \wedge (\mathit{color}\ r = \mathit{Black} \longrightarrow \mathit{invc}\ t')$

Lemma 8.5. $\mathit{invh}\ l \wedge \mathit{invh}\ r \wedge \mathit{bh}\ l = \mathit{bh}\ r + 1 \wedge \mathit{invc}\ l \wedge \mathit{invc2}\ r \wedge$
$t' = \mathit{baldR}\ l\ a\ r \longrightarrow$
$\mathit{invh}\ t' \wedge \mathit{bh}\ t' = \mathit{bh}\ l \wedge \mathit{invc2}\ t' \wedge (\mathit{color}\ l = \mathit{Black} \longrightarrow \mathit{invc}\ t')$

The proofs of these lemmas are by case analyses over the defining equations using the characteristic properties of *baliL* and *baliR* given above.

Proof. Lemma 8.2 is proved by induction on the computation of $\mathit{del}\ x\ t$. We concentrate on the induction step where $t = \langle l,\ (a,\ c),\ r\rangle$, and thus $\mathit{invh}\ l$, $\mathit{invh}\ r$, $\mathit{bh}\ l = \mathit{bh}\ r$, $\mathit{invc}\ l$ and $\mathit{invc}\ r$ because $\mathit{invh}\ t$ and $\mathit{invc}\ t$. Let $t' = \mathit{del}\ x\ t$. We need to prove $\mathit{invh}\ t'$ and

(R) $c = \mathit{Red} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t \wedge \mathit{invc}\ t'$
(B) $c = \mathit{Black} \longrightarrow \mathit{bh}\ t' = \mathit{bh}\ t - 1 \wedge \mathit{invc2}\ t'$.

Consider the case $x < a$. Then $t' = \mathit{baldL}\ l'\ a\ r$ where $l' = \mathit{del}\ x\ l$. By IH we have $\mathit{invh}\ l'$, $\mathit{color}\ l = \mathit{Red} \longrightarrow \mathit{bh}\ l' = \mathit{bh}\ l \wedge \mathit{invc}\ l'$ and $\mathit{color}\ l = \mathit{Black} \longrightarrow \mathit{bh}\ l' = \mathit{bh}\ l - 1 \wedge \mathit{invc2}\ l'$. First we assume $l \neq \langle\rangle \wedge \mathit{color}\ l = \mathit{Black}$. Therefore $0 < \mathit{bh}\ l$ and hence $\mathit{bh}\ l' + 1 = \mathit{bh}\ r$ follows from $\mathit{bh}\ l' = \mathit{bh}\ l - 1$ and $\mathit{bh}\ l = \mathit{bh}\ r$. Thus all premises of Lemma 8.4 (where $l = l'$) hold and we obtain its conclusion: $\mathit{invh}\ t' \wedge \mathit{bh}\ t' = \mathit{bh}\ r \wedge \mathit{invc2}\ t' \wedge (\mathit{color}\ r = \mathit{Black} \longrightarrow \mathit{invc}\ t')$. It remains to prove (R) and (B). If $c = \mathit{Red}$ then $\mathit{bh}\ t' = \mathit{bh}\ r = \mathit{bh}\ l = \mathit{bh}\ t$ and $\mathit{invc}\ t$ implies $\mathit{color}\ r = \mathit{Black}$ and thus $\mathit{invc}\ t'$. If $c = \mathit{Black}$ then $\mathit{bh}\ t' = \mathit{bh}\ r = \mathit{bh}\ l = \mathit{bh}\ t - 1$ and $\mathit{invc2}\ t'$ follows from Lemma 8.4. Now we consider the case $\neg\ (l \neq \langle\rangle \wedge \mathit{color}\ l = \mathit{Black})$. First assume $l = \langle\rangle$. Hence $t = \langle\langle\rangle,\ (a,\ c),\ r\rangle$ and $t' = R\ \langle\rangle\ a\ r$. Thus $\mathit{invh}\ t'$ follows from $\mathit{invh}\ t$. It remains to prove (R) and (B). If $c = \mathit{Red}$ then $\mathit{bh}\ t' = 0 = \mathit{bh}\ t$ and $\mathit{invc}\ t$ implies $\mathit{color}\ r = \mathit{Black}$ and thus $\mathit{invc}\ t'$ because $\mathit{invc}\ r$. If $c = \mathit{Black}$ then $\mathit{bh}\ t' = 0 = 1 - 1 = \mathit{bh}\ t - 1$ and $\mathit{invc2}\ t'$ follows from $\mathit{invc}\ r$. Now assume $\mathit{color}\ l \neq \mathit{Black}$. This implies $c = \mathit{Black}$ because of $\mathit{invc}\ t$. Hence $t' = R\ l'\ a\ r$ and $\mathit{bh}\ l' = \mathit{bh}\ l \wedge \mathit{invc}\ l'$ (by IH). Therefore $\mathit{invh}\ t' = (\mathit{invh}\ l' \wedge \mathit{invh}\ r \wedge \mathit{bh}\ l' = \mathit{bh}\ r)$ — the first two propositions have been derived above and the last proposition is easy: $\mathit{bh}\ l' = \mathit{bh}\ l = \mathit{bh}\ r$. It remains to prove (B) because $c = \mathit{Black}$. Proposition (B) follows easily: $\mathit{bh}\ t' = \mathit{bh}\ l' = \mathit{bh}\ l = \mathit{bh}\ t - 1$ and $\mathit{invc2}\ t' = (\mathit{invc}\ l' \wedge \mathit{invc}\ r)$ where both conjuncts have been derived above.

The case $a < x$ is analogous and the case $x = a$ is similar but simpler. □

The proof of Lemma 8.1 is similar but simpler.

8.2.3 Deletion by Joining

The basic idea was exemplified in the context of ordinary BSTs in Section 5.2.1. The code for red-black trees is shown in Figure 8.3: compared to Figure 8.2, the *EQ* case of *del* has changed and *join* is new.

Invariant preservation is proved much like before except that instead of *split_min* we now have *join* to take care of. The characteristic lemma:

```
del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ if l ≠ ⟨⟩ ∧ color l = Black then baldL (del x l) a r
          else R (del x l) a r |
    EQ ⇒ join l r |
    GT ⇒ if r ≠ ⟨⟩ ∧ color r = Black then baldR l a (del x r)
          else R l a (del x r))

join :: 'a rbt ⇒ 'a rbt ⇒ 'a rbt
join ⟨⟩ t = t
join t ⟨⟩ = t
join (R $t_1$ a $t_2$) (R $t_3$ c $t_4$)
= (case join $t_2$ $t_3$ of
    R $u_2$ b $u_3$ ⇒ R (R $t_1$ a $u_2$) b (R $u_3$ c $t_4$) |
    $t_{23}$ ⇒ R $t_1$ a (R $t_{23}$ c $t_4$)
join (B $t_1$ a $t_2$) (B $t_3$ c $t_4$)
= (case join $t_2$ $t_3$ of
    R $u_2$ b $u_3$ ⇒ R (B $t_1$ a $u_2$) b (B $u_3$ c $t_4$) |
    $t_{23}$ ⇒ baldL $t_1$ a (R $t_{23}$ c $t_4$)
join $t_1$ (R $t_2$ a $t_3$) = R (join $t_1$ $t_2$) a $t_3$ |
join (R $t_1$ a $t_2$) $t_3$ = R $t_1$ a (join $t_2$ $t_3$)
```

Fig. 8.3. Deletion from red-black tree by combining children

Lemma 8.6. $invh\ l \wedge invh\ r \wedge bh\ l = bh\ r \wedge invc\ l \wedge invc\ r \wedge$
$t' = join\ l\ r \longrightarrow$
$invh\ t' \wedge bh\ t' = bh\ l \wedge invc2\ t' \wedge$
$(color\ l = Black \wedge color\ r = Black \longrightarrow invc\ t')$

8.3 Exercises

Exercise 8.1. Show that the logarithmic height of red-black trees is already guaranteed by the color and height invariants:

$$invc\ t \wedge invh\ t \longrightarrow h\ t \leq 2 \cdot \lg |t|_1 + 2$$

Exercise 8.2. We already discussed informally why the definition of *invh* captures “all paths from the root to a leaf have the same number of black nodes” although *bh* only traverses the left spine. This exercises formalizes that discussion. The following function computes the set of black heights (number of black nodes) of all paths:

$$\begin{aligned}
&\mathit{bhs} :: {}'a\ \mathit{rbt} \Rightarrow \mathit{nat\ set}\\
&\mathit{bhs}\ \langle\rangle = \{0\}\\
&\mathit{bhs}\ \langle l,\ (\_ ,\ c),\ r\rangle\\
&= (\textbf{let}\ H = \mathit{bhs}\ l \cup \mathit{bhs}\ r\ \textbf{in}\ \textbf{if}\ c = \mathit{Black}\ \textbf{then}\ \mathit{Suc}\ \text{‘}\ H\ \textbf{else}\ H)
\end{aligned}$$

where the infix operator (‘) is predefined as f ‘ $A = \{y \mid \exists\, x{\in}A.\ y = f\ x\}$. Prove $\mathit{invh}\ t \longleftrightarrow \mathit{bhs}\ t = \{\mathit{bh}\ t\}$. The direction $\mathit{invh}\ t \longrightarrow \mathit{bhs}\ t = \{\mathit{bh}\ t\}$ should be easy, the other direction should need some lemmas.

Exercise 8.3. Following Section 7.3, define a linear time function *rbt_of_list* :: *'a list* ⇒ *'a rbt* and prove both *inorder* (*rbt_of_list as*) = *as* and *rbt* (*rbt_of_list as*).

Bibliographic Remarks

Red-Black trees were invented by Bayer [6] who called them “symmetric binary B-trees”. The red-black color convention was introduced by Guibas and Sedgewick [23] who studied their properties in greater depth. The first functional version of red-black trees (without deletion) is due to Okasaki [62] and everybody follows his code. A functional version of deletion was first given by Kahrs [36][1] and Section 8.2.3 is based on it. Germane [22] presents a function for deletion by replacement that is quite different from the one in Section 8.2.2. Our starting point were Isabelle proofs by Reiter and Krauss (based on Kahrs). Other verifications of red-black trees are reported by Filliâtre and Letouzey [19] (using their own deletion function) and Appel [4] (based on Kahrs).

[1] The code for deletion is not in the article but can be retrieved from this URL: `http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html`

9

AVL Trees

The AVL tree [2] (named after its inventors Adelson-Velsky and Landis) is the granddaddy of efficient binary search trees. Its logarithmic height is maintained by rotating subtrees based on their height. For efficiency reasons the height of each subtree is stored in its root node. That is, the underlying data structure is a height-augmented tree (see Section 4.4):

```
type_synonym 'a tree_ht = ('a × nat) tree
```

Function *ht* extracts the height field and *node* is a smart constructor that sets the height field:

```
ht :: 'a tree_ht ⇒ nat
ht ⟨⟩ = 0
ht ⟨_, (_, n), _⟩ = n

node :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
node l a r = ⟨l, (a, max (ht l) (ht r) + 1), r⟩
```

An **AVL tree** is a tree that satisfies the AVL invariant: the height of the left and right child of any node differ by at most 1

```
avl :: 'a tree_ht ⇒ bool
avl ⟨⟩ = True
avl ⟨l, (_, n), r⟩
= (|int (h l) − int (h r)| ≤ 1 ∧
    n = max (h l) (h r) + 1 ∧ avl l ∧ avl r)
```

and the height field contains the correct value. The conversion function int :: $nat \Rightarrow int$ is required because on natural numbers $0 - n = 0$.

9.1 Logarithmic Height

AVL trees have logarithmic height. The key insight for the proof is that $M\ n$, the minimal number of leaves of an AVL tree of height n, satisfies the recurrence relation $M\ (n + 2) = M\ (n + 1) + M\ n$. Instead of formalizing this function M we prove directly that an AVL tree of height n has at least $\mathit{fib}\ (n + 2)$ leaves where fib is the Fibonacci function:

$$\mathit{fib} :: nat \Rightarrow nat$$

$$\begin{aligned}
&\mathit{fib}\ 0 = 0\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ (n + 2) = \mathit{fib}\ (n + 1) + \mathit{fib}\ n
\end{aligned}$$

Lemma 9.1. $avl\ t \longrightarrow \mathit{fib}\ (h\ t + 2) \le |t|_1$

Proof. The proof is by induction on t. We focus on the induction step $t = \langle l, (a, n), r\rangle$ and assume $avl\ t$. Thus the IHs reduce to $\mathit{fib}\ (h\ l + 2) \le |l|_1$ and $\mathit{fib}\ (h\ r + 2) \le |r|_1$. We prove $\mathit{fib}\ (max\ (h\ l)\ (h\ r) + 3) \le |l|_1 + |r|_1$, from which $avl\ t \longrightarrow \mathit{fib}\ (h\ t + 2) \le |t|_1$ follows directly. There are two cases. We focus on $h\ l \ge h\ r$, $h\ l < h\ r$ is dual.

$$\begin{aligned}
&\mathit{fib}\ (max\ (h\ l)\ (h\ r) + 3) = \mathit{fib}\ (h\ l + 3) && \\
&= \mathit{fib}\ (h\ l + 2) + \mathit{fib}\ (h\ l + 1) && \\
&\le |l|_1 + \mathit{fib}\ (h\ l + 1) && \text{by } \mathit{fib}\ (h\ l + 2) \le |l|_1\\
&\le |l|_1 + |r|_1 && \text{by } \mathit{fib}\ (h\ r + 2) \le |r|_1
\end{aligned}$$

The last step is justified because $h\ l + 1 \le h\ r + 2$ (which follows from $avl\ t$) and fib is monotone. □

Now we prove a well-known exponential lower bound for fib where $\varphi \equiv (1 + \sqrt{5})\ /\ 2$:

Lemma 9.2. $\varphi^n \le \mathit{fib}\ (n + 2)$

Proof. The proof is by induction on n by fib computation induction. The case $n = 0$ is trivial and the case $n = 1$ is easy. Now consider the induction step:

$$\begin{aligned} &\mathit{fib}\ (n + 2 + 2) = \mathit{fib}\ (n + 2 + 1) + \mathit{fib}\ (n + 2) \\ &\geq \varphi^{n + 1} + \varphi^{n} && \text{by IHs} \\ &= (\varphi + 1) \cdot \varphi^{n} \\ &= \varphi^{n + 2} && \text{because } \varphi + 1 = \varphi^{2} \quad \square \end{aligned}$$

Combining the two lemmas yields $\mathit{avl}\ t \longrightarrow \varphi^{h\ t} \leq |t|_1$ and thus

Corollary 9.3. $\mathit{avl}\ t \longrightarrow h\ t \leq 1\ /\ \lg\ \varphi \cdot \lg\ |t|_1$

That is, the height of an AVL tree is at most $1\ /\ \lg\ \varphi \approx 1.44$ times worse than the optimal $\lg\ |t|_1$.

9.2 Implementation of ADT *Set*

9.2.1 Insertion

Insertion follows the standard approach: insert the element as usual and reestablish the AVL invariant on the way back up.

```
insert :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
insert x ⟨⟩ = ⟨⟨⟩, (x, 1), ⟨⟩⟩
insert x ⟨l, (a, n), r⟩ = (case cmp x a of
                             LT ⇒ balL (insert x l) a r |
                             EQ ⇒ ⟨l, (a, n), r⟩ |
                             GT ⇒ balR l a (insert x r))
```

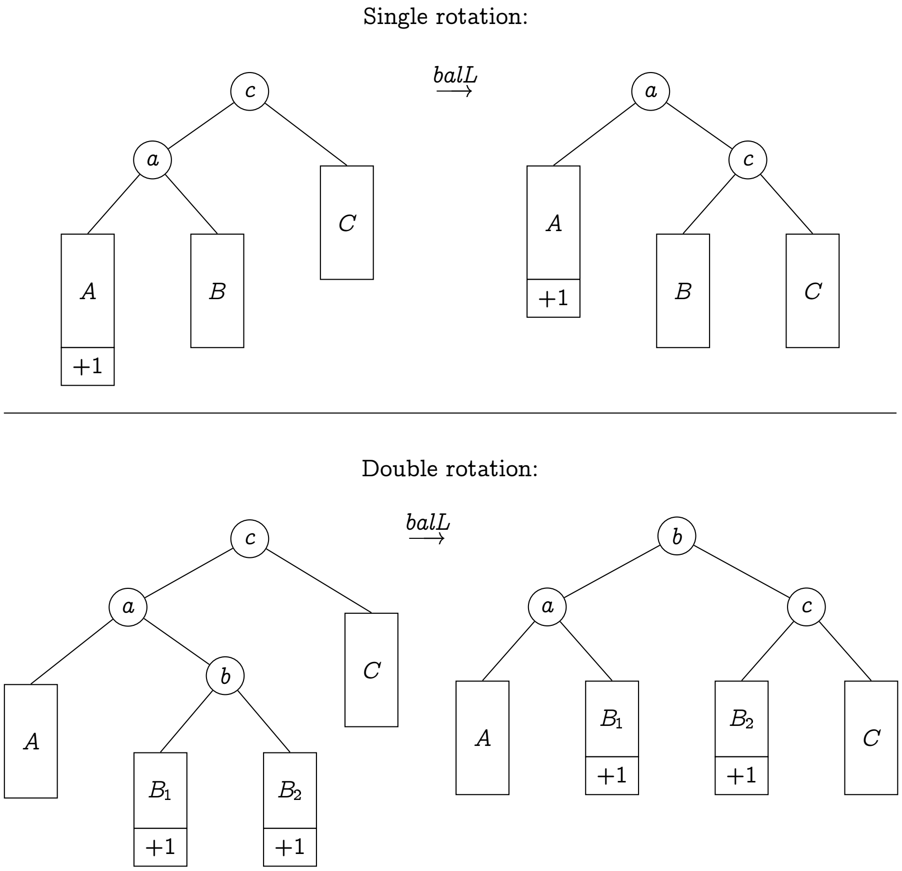
Functions *balL*/*balR* readjust the tree after an insertion into the left/right child. The AVL invariant has been lost if the difference in height has become 2 — it cannot become more because the height can only increase by 1. Consider the definition of *balL* in Figure 9.1 (*balR* in Figure 9.2 is dual). If the AVL invariant has not been lost, i.e. if $\mathit{ht}\ AB \neq \mathit{ht}\ C + 2$, then we can just return the AVL tree $\mathit{node}\ AB\ c\ C$. But if $\mathit{ht}\ AB = \mathit{ht}\ C + 2$, we need to "rotate" the subtrees suitably. Clearly AB must be of the form $\langle A,\ (a,\ \_),\ B\rangle$. There are two cases, which are illustrated in Figure 9.1. Rectangles denote trees. Rectangles of the same height denote trees of the same height. Rectangles with a +1 denote the additional level due to insertion of the new element.

If $\mathit{ht}\ B \leq \mathit{ht}\ A$ then *balL* performs what is known as a single rotation.

If $\mathit{ht}\ A < \mathit{ht}\ B$ then B must be of the form $\langle B_1,\ (b,\ \_),\ B_2\rangle$ (where either B_1 or B_2 has increased in height) and *balL* performs what is known as a double rotation.

It is easy to check that in both cases the tree on the right satisfies the AVL invariant.

```
balL :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balL AB c C
= (if ht AB = ht C + 2
    then case AB of
           ⟨A, (a, x), B⟩ ⇒
             if ht B ≤ ht A then node A a (node B c C)
             else case B of
                    ⟨B₁, (b, _), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
    else node AB c C)
```


Single rotation:

Double rotation:

Fig. 9.1. Function *balL*

```
balR :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balR A a BC
= (if ht BC = ht A + 2
   then case BC of
          ⟨B, (c, x), C⟩ ⇒
            if ht B ≤ ht C then node (node A a B) c C
            else case B of
                   ⟨B₁, (b, _), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
   else node A a BC)
```

Fig. 9.2. Function *balR*

Preservation of *avl* by *insert* cannot be proved in isolation but needs to be proved simultaneously with how *insert* changes the height (because *avl* depends on the height and *insert* requires *avl* for correct behaviour):

Theorem 9.4. $avl\ t \longrightarrow avl\ (insert\ x\ t) \wedge h\ (insert\ x\ t) \in \{h\ t,\ h\ t + 1\}$

The proof is by induction on t followed by a complete case analysis (which Isabelle automates).

9.2.2 Deletion

```
delete :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
   LT ⇒ balR (delete x l) a r |
   EQ ⇒ if l = ⟨⟩ then r else let (l', a') = split_max l in balR l' a' r |
   GT ⇒ balL l a (delete x r))

split_max :: 'a tree_ht ⇒ 'a tree_ht × 'a
split_max ⟨l, (a, _), r⟩
= (if r = ⟨⟩ then (l, a)
   else let (r', a') = split_max r in (balL l a r', a'))
```

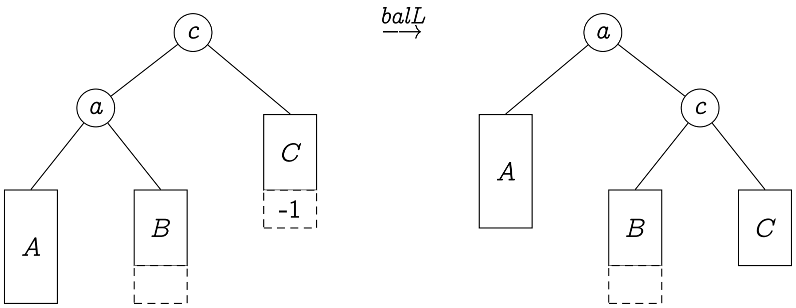
Fig. 9.3. Deletion from AVL tree

Figure 9.3 shows deletion-by-replacing (see 5.2.1). The recursive calls are dual to insertion: in terms of the difference in height, deletion of some element from one child is the same as insertion of some element into the other child. Thus functions *balR*/*balL* can again be employed to restore the invariant.

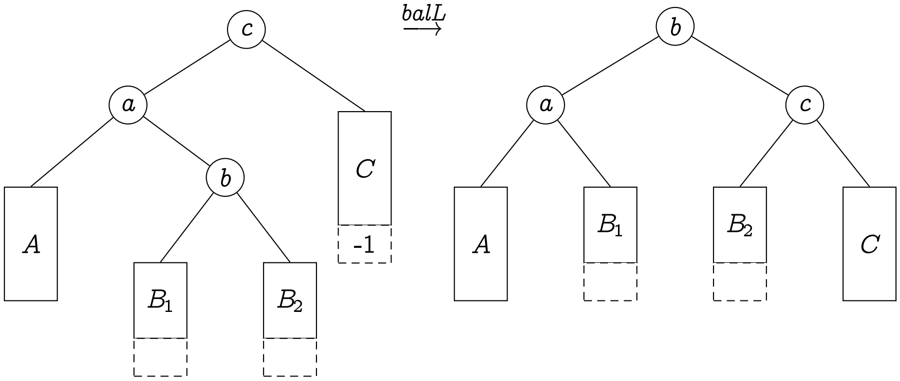
An element is deleted from a node by replacing it with the maximal element of the left child (the minimal element of the right child would work just as well). Function *split_max* performs that extraction and uses *balL* to restore the invariant after splitting an element off the right child.

The fact that *balR*/*balL* can be reused for deletion can be illustrated by drawing the corresponding rotation diagrams. We look at how the code for *balL* behaves when an element has been deleted from C. Dashed rectangles indicate a single additional level that may or may not be there. The label -1 indicates that the level has disappeared due to deletion.

Single rotation in *balL* after deletion in C:



Double rotation in *balL* after deletion in C:



At least one of B_1 and B_2 must have the same height as A.

Preservation of *avl* by *delete* can be proved in the same manner as for *insert* but we provide more of the details (partly because our Isabelle proof is less automatic). The following lemmas express that the auxiliary functions preserve *avl*:

$$avl\ l \wedge avl\ r \wedge h\ r - 1 \leq h\ l \wedge h\ l \leq h\ r + 2 \longrightarrow avl\ (balL\ l\ a\ r)$$

$$avl\ l \wedge avl\ r \wedge h\ l - 1 \le h\ r \wedge h\ r \le h\ l + 2 \longrightarrow avl\ (balR\ l\ a\ r)$$

$$\begin{aligned}&avl\ t \wedge t \neq \langle\rangle \longrightarrow\\ &avl\ (fst\ (split\_max\ t)) \wedge\\ &h\ t \in \{h\ (fst\ (split\_max\ t)),\ h\ (fst\ (split\_max\ t)) + 1\}\end{aligned}$$

The first two are proved by the obvious cases analyses, the last one also requires induction.

As for *insert*, preservation of *avl* by *delete* needs to be proved simultaneously with how *delete* changes the height:

Theorem 9.5. $avl\ t \wedge t' = delete\ x\ t \longrightarrow avl\ t' \wedge h\ t \in \{h\ t',\ h\ t' + 1\}$

Proof. The proof is by induction on t followed by the case analyses dictated by the code for *delete*. We sketch the induction step. Let $t = \langle l,\ (a,\ n),\ r\rangle$ and $t' = delete\ x\ t$ and assume the IHs and $avl\ t$. The claim $avl\ t'$ follows from the preservation of *avl* by *balL*, *balR* and *split_max* as shown above. The claim $h\ t \in \{h\ t',\ h\ t' + 1\}$ follows directly from the definitions of *balL* and *balR*. □

9.3 Exercises

Exercise 9.1. The logarithmic height of AVL trees can be proved directly. Prove

$$avl\ t \wedge h\ t = n \longrightarrow 2^{n\ div\ 2} \le |t|_1$$

by *fib* computation induction on n. This implies $avl\ t \longrightarrow h\ t \le 2 \cdot \lg\ |t|_1$.

Exercise 9.2. Fibonacci trees are defined in analogy to Fibonacci numbers:

$$fibt :: nat \Rightarrow unit\ tree$$

$$\begin{aligned}&fibt\ 0 = \langle\rangle\\ &fibt\ 1 = \langle\langle\rangle,\ (),\ \langle\rangle\rangle\\ &fibt\ (n + 2) = \langle fibt\ (n + 1),\ (),\ fibt\ n\rangle\end{aligned}$$

We are only interested in the shape of these trees. Therefore the nodes just contain dummy unit values (). Hence we need to define the AVL invariant again for trees without annotations:

$$avl0 :: 'a\ tree \Rightarrow bool$$

$$\begin{aligned}&avl0\ \langle\rangle = True\\ &avl0\ \langle l,\ \_,\ r\rangle = (|int\ (h\ l) - int\ (h\ r)| \le 1 \wedge avl0\ l \wedge avl0\ r)\end{aligned}$$

Prove the following properties of Fibonacci trees:

$$avl0\ (fibt\ n) \qquad |fibt\ n|_1 = fib\ (n + 2)$$

Conclude that the Fibonacci trees are minimal (w.r.t. their size) among all AVL trees of a given height:

$$\mathit{avl}\ t \longrightarrow |\mathit{fibt}\ (h\ t)|_1 \leq |t|_1$$

Exercise 9.3. Show that every almost complete tree is an AVL tree:

$$\mathit{acomplete}\ t \longrightarrow \mathit{avl0}\ t$$

As in the previous exercise we consider trees without height annotations.

Exercise 9.4. Generalize AVL trees to **height-balanced trees** where the condition

$$|\mathit{int}\ (h\ l) - \mathit{int}\ (h\ r)| \leq 1$$

in the invariant is replaced by

$$|\mathit{int}\ (h\ l) - \mathit{int}\ (h\ r)| \leq m$$

where $m \geq 1$ is some fixed integer. Modify the invariant and the insertion and deletion functions and prove that the latter fulfill the same correctness theorems as before. You do not need to prove the logarithmic height of height-balanced trees.

Exercise 9.5. Following Section 7.3, define a linear-time function *avl_of_list* :: *'a list* ⇒ *'a tree_ht* and prove both $\mathit{inorder}\ (\mathit{avl\_of\_list}\ \mathit{as}) = \mathit{as}$ and $\mathit{avl}\ (\mathit{avl\_of\_list}\ \mathit{as})$.

9.4 An Optimization

Instead of recording the height of the tree in each node, it suffices to record the **balance factor**, i.e. the difference in height of its two children. Rather than the three integers -1, 0 and 1 we utilize a new data type:

```
datatype bal = Lh | Bal | Rh

type_synonym 'a tree_bal = ('a × bal) tree
```

The names *Lh* and *Rh* stand for “left-heavy” and “right-heavy”. The AVL invariant for these trees reflect these names:

```
avl :: 'a tree_bal ⇒ bool
avl ⟨⟩ = True
avl ⟨l, (_, b), r⟩ = ((case b of
                          Lh ⇒ h l = h r + 1 |
                          Bal ⇒ h r = h l |
                          Rh ⇒ h r = h l + 1) ∧
                        avl l ∧ avl r)
```

The code for insertion (and deletion) is similar to the height-based version. The key difference is that the test if the AVL invariant as been lost cannot be based on the height anymore. We need to detect if the tree has increased in height upon insertion based on the balance factors. The key insight is that a height increase is coupled with a change from *Bal* to *Lh* or *Rh*. Except when we transition from ⟨⟩ to ⟨⟨⟩, (*a*, *Bal*), ⟨⟩⟩. This insight is encoded in the test *incr*:

```
is_bal :: 'a tree_bal ⇒ bool
is_bal ⟨_, (_, b), _⟩ = (b = Bal)

incr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
incr t t' = (t = ⟨⟩ ∨ is_bal t ∧ ¬ is_bal t')
```

The test for a height increase compares the trees before and after insertion. Therefore it has been pulled out of the balance functions into insertion:

```
insert :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
insert x ⟨⟩ = ⟨⟨⟩, (x, Bal), ⟨⟩⟩
insert x ⟨l, (a, b), r⟩
= (case cmp x a of
    LT ⇒ let l' = insert x l
           in if incr l l' then balL l' a b r else ⟨l', (a, b), r⟩ |
    EQ ⇒ ⟨l, (a, b), r⟩ |
    GT ⇒ let r' = insert x r
           in if incr r r' then balR l a b r' else ⟨l, (a, b), r'⟩)
```

The balance functions are shown in Figure 9.4. Function *rot*2 implements double rotations. Function *balL* is called if the left child *AB* has increased in height. If the tree was *Lh* then single or double rotations are necessary

```
balL :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balL AB c bc C
= (case bc of
    Lh ⇒ case AB of
           ⟨A, (a, Lh), B⟩ ⇒ ⟨A, (a, Bal), ⟨B, (c, Bal), C⟩⟩ |
           ⟨A, (a, Bal), B⟩ ⇒ ⟨A, (a, Rh), ⟨B, (c, Lh), C⟩⟩ |
           ⟨A, (a, Rh), B⟩ ⇒ rot2 A a B c C |
    Bal ⇒ ⟨AB, (c, Lh), C⟩ |
    Rh ⇒ ⟨AB, (c, Bal), C⟩)

balR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balR A a ba BC
= (case ba of
    Lh ⇒ ⟨A, (a, Bal), BC⟩ |
    Bal ⇒ ⟨A, (a, Rh), BC⟩ |
    Rh ⇒ case BC of
           ⟨B, (c, Lh), C⟩ ⇒ rot2 A a B c C |
           ⟨B, (c, Bal), C⟩ ⇒ ⟨⟨A, (a, Rh), B⟩, (c, Lh), C⟩ |
           ⟨B, (c, Rh), C⟩ ⇒ ⟨⟨A, (a, Bal), B⟩, (c, Bal), C⟩)

rot2 :: 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
rot2 A a B c C
= (case B of
    ⟨B1, (b, bb), B2⟩ ⇒
      let b1 = if bb = Rh then Lh else Bal;
          b2 = if bb = Lh then Rh else Bal
      in ⟨⟨A, (a, b1), B1⟩, (b, Bal), ⟨B2, (c, b2), C⟩⟩)
```

Fig. 9.4. Functions *balL* and *balR*

to restore balance. Otherwise we simply need to adjust the balance factors. Function *balR* is dual to *balL*.

For deletion we need to test if the height has decreased and *decr* implements this test:

```
decr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
decr t t' = (t ≠ ⟨⟩ ∧ (t' = ⟨⟩ ∨ ¬ is_bal t ∧ is_bal t'))
```

The functions *incr* and *decr* are almost dual except that *incr* implicitly assumes $t' \neq \langle\rangle$ because insertion is guaranteed to return a *Node*. Thus we could use *decr* instead of *incr* but not the other way around.

Deletion and *split_max* change in the same manner as insertion:

```
delete :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, ba), r⟩
= (case cmp x a of
    LT ⇒ let l' = delete x l
          in if decr l l' then balR l' a ba r else ⟨l', (a, ba), r⟩
  | EQ ⇒ if l = ⟨⟩ then r
          else let (l', a') = split_max l
               in if decr l l' then balR l' a' ba r
                  else ⟨l', (a', ba), r⟩
  | GT ⇒ let r' = delete x r
          in if decr r r' then balL l a ba r' else ⟨l, (a, ba), r'⟩)

split_max :: 'a tree_bal ⇒ 'a tree_bal × 'a
split_max ⟨l, (a, ba), r⟩
= (if r = ⟨⟩ then (l, a)
   else let (r', a') = split_max r;
            t' = if decr r r' then balL l a ba r' else ⟨l, (a, ba), r'⟩
        in (t', a'))
```

In the end we have the following correctness theorems:

Theorem 9.6. $avl\ t \wedge t' = insert\ x\ t \longrightarrow$
$avl\ t' \wedge h\ t' = h\ t + (\textbf{if}\ incr\ t\ t'\ \textbf{then}\ 1\ \textbf{else}\ 0)$

This theorem tells us not only that *avl* is preserved but also that *incr* indicates correctly if the height has increased or not.

Similarly for deletion and *decr*:

Theorem 9.7. $avl\ t \wedge t' = delete\ x\ t \longrightarrow$
$avl\ t' \wedge h\ t = h\ t' + (\textbf{if}\ decr\ t\ t'\ \textbf{then}\ 1\ \textbf{else}\ 0)$

The proofs of both theorems follow the standard pattern of induction followed by an exhaustive (automatic) cases analysis. The proof for *delete* requires an analogous lemma for *split_max*:

$$split\_max\ t = (t', a) \wedge avl\ t \wedge t \neq \langle\rangle \longrightarrow$$
$$avl\ t' \wedge h\ t = h\ t' + (\textbf{if}\ decr\ t\ t'\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

9.5 Exercises

Exercise 9.6. We map type *'a tree_bal* back to type *('a × nat) tree* called *'a tree_ht* in the beginning of the chapter:

$debal :: {}'a\ tree\_bal \Rightarrow ({}'a \times nat)\ tree$

$debal\ \langle\rangle = \langle\rangle$
$debal\ \langle l,\ (a,\ \_),\ r\rangle = \langle debal\ l,\ (a,\ max\ (h\ l)\ (h\ r) + 1),\ debal\ r\rangle$

Prove that the AVL property is preserved: $avl\ t \longrightarrow avl\_ht\ (debal\ t)$ where avl_ht is the avl predicate on type $'a\ tree\_ht$ from the beginning of the chapter.

Define a function $debal2$ of the same type that traverses the tree only once and in particular does not use function h. Prove $avl\ t \longrightarrow debal2\ t = debal\ t$.

Exercise 9.7. To realize the full space savings potential of balance factors we encode them directly into the node constructors and work with the following special tree type:

```
datatype 'a tree4 = Leaf
  | Lh ('a tree4) 'a ('a tree4)
  | Bal ('a tree4) 'a ('a tree4)
  | Rh ('a tree4) 'a ('a tree4)
```

On this type define the AVL invariant, insertion, deletion and all necessary auxiliary functions. Prove theorems 9.6 and 9.7. Hint: modify the theory underlying Section 9.4.

10

Beyond Insert and Delete: $\cup$, $\cap$ and $-$

So far we looked almost exclusively at insertion and deletion of single elements, with the exception of the conversion of whole lists of elements into search trees (see Section 7.3 and Exercises 8.3 and 9.5). This chapter is dedicated to operations that combine two sets (implemented by search trees) by union, intersection and difference. We denote set difference by $-$ rather than $\setminus$.

Let us focus on set union for a moment and assume that insertion into a set of size s takes time proportional to $\lg s$. Consider two sets A and B of size m and n where $m \le n$. The naive approach is to insert the elements from one set one by one into the other set. This takes time proportional to $\lg n + \cdots + \lg(n+m-1)$ or $\lg m + \cdots + \lg(m+n-1)$ depending on whether the smaller set is inserted into the larger one or the other way around. Of course the former sum is less than or equal to the latter sum. To estimate the growth of $\lg n + \cdots + \lg(n+m-1) = \lg(n \cdots (n+m-1))$ we can easily generalize the derivation of $\lg(n!) \in \Theta(n \lg n)$ in the initial paragraph of Section 7.3. The result is $\lg(n \cdots (n+m-1)) \in \Theta(m \lg n)$. That is, inserting m elements into an n element set one by one takes time $\Theta(m \lg n)$.

There is a second, possibly naive sounding algorithm for computing the union: flatten both trees to ordered lists (using function *inorder2* from Exercise 4.1), merge both lists and convert the resulting list back into a suitably balanced search tree. All three steps take linear time. The last step is the only slightly nontrivial one but has been dealt with before (see Section 7.3 and Exercises 8.3 and 9.5). This algorithm takes time $O(m+n)$ which is significantly better than $O(m \lg n)$ if $m \approx n$ but significantly worse if $m \ll n$.

This chapter is concerned with a third approach that has the following salient features:

- Union, intersection and difference take time $O(m \lg(\frac{n}{m} + 1))$
- It works for a whole class of balanced trees, including AVL, red-black and weight-balanced trees.

- It is based on a single function for joining two balanced trees to form a new balanced tree.

We call it the **join approach**. It is easily and efficiently parallelizable, a property we will not explore here.

The join approach is at least as fast as the one-by-one approach: from $m+n \leq mn$ it follows that $\frac{n}{m}+1 \leq n$ (if $m,n \geq 2$). The join approach is also at least as fast as the tree-to-list-to-tree approach because $m+n = m(\frac{n}{m}+1)$ (if $m \geq 1$).

10.1 Specification of Union, Intersection and Difference

Before explaining the join approach we extend the ADT *Set* by three new functions *union*, *inter* and *diff*. The specification in Figure 10.1 is self-explanatory.

ADT *Set2* = *Set* +

interface

$union :: {'s} \Rightarrow {'s} \Rightarrow {'s}$
$inter :: {'s} \Rightarrow {'s} \Rightarrow {'s}$
$diff :: {'s} \Rightarrow {'s} \Rightarrow {'s}$

specification

$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (union\ s_1\ s_2)$ (*union-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$ (*union*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (inter\ s_1\ s_2)$ (*inter-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (inter\ s_1\ s_2) = set\ s_1 \cap set\ s_2$ (*inter*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (diff\ s_1\ s_2)$ (*diff-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (diff\ s_1\ s_2) = set\ s_1 - set\ s_2$ (*diff*)

Fig. 10.1. ADT *Set2*

10.2 Just Join

Now we come to the heart of the matter, the definition of union, intersection and difference in terms of a single function *join*. We promised that the algorithms would be generic across a range of balanced trees. Thus we assume that we operate on augmented trees of type $('a \times 'b)$ *tree* where $'a$ is the type of the elements and $'b$ is the balancing information (which we can ignore here). This enables us to formulate the algorithms via pattern-matching. A more generic approach is the subject of Exercise 10.1.

The whole section is parameterized by the join function and an invariant:

$$\begin{aligned}&\mathit{join} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'a \Rightarrow ('a \times 'b)\ \mathit{tree} \Rightarrow ('a \times 'b)\ \mathit{tree}\\ &\mathit{inv} :: ('a \times 'b)\ \mathit{tree} \Rightarrow \mathit{bool}\end{aligned}$$

Function *inv* is meant to take care of the balancedness property only, not the BST property. Functions *join* and *inv* are specified with the help of the standard tree functions *set_tree* and *bst* in Figure 10.2. With respect to the set of elements, *join* must behave like union. But it need only return a BST if both trees are BSTs and the element a lies in between the elements of the two trees, i.e. if $\mathit{bst}\ \langle l, (a, \_), r\rangle$. The structural invariant *inv* must be preserved by formation and destruction of trees. Thus we can see *join* as a smart constructor that builds a balanced tree.

$$\mathit{set\_tree}\ (\mathit{join}\ l\ a\ r) = \mathit{set\_tree}\ l \cup \{a\} \cup \mathit{set\_tree}\ r \tag{10.1}$$
$$\mathit{bst}\ \langle l, (a, \_), r\rangle \longrightarrow \mathit{bst}\ (\mathit{join}\ l\ a\ r) \tag{10.2}$$
$$\mathit{inv}\ \langle\rangle$$
$$\mathit{inv}\ l \wedge \mathit{inv}\ r \longrightarrow \mathit{inv}\ (\mathit{join}\ l\ a\ r) \tag{10.3}$$
$$\mathit{inv}\ \langle l, (\_, \_), r\rangle \longrightarrow \mathit{inv}\ l \wedge \mathit{inv}\ r \tag{10.4}$$

Fig. 10.2. Specification of *join* and *inv*

To define union and friends we need a number of simple auxiliary functions shown in Figure 10.3. Function *split_min* decomposes a tree into its leftmost (minimal) element and the remaining tree; the remaining tree is reassembled via *join*, thus preserving *inv*. Function *join*2 is reduced to *join* with the help of *split_min*. Function *split* splits a BST w.r.t. a given element a into a triple (l, b, r) such that l contains the elements less than a, r contains the elements greater than a, and b is true iff a was in the input tree.

Although insertion and deletion could be defined by means of union and difference, we can define them directly from the auxiliary functions:

$$\mathit{insert} :: 'a \Rightarrow ('a \times 'b)\ \mathit{tree} \Rightarrow ('a \times 'b)\ \mathit{tree}$$
$$\mathit{insert}\ x\ t = (\mathbf{let}\ (l, b, r) = \mathit{split}\ t\ x\ \mathbf{in}\ \mathit{join}\ l\ x\ r)$$

$$\mathit{delete} :: 'a \Rightarrow ('a \times 'b)\ \mathit{tree} \Rightarrow ('a \times 'b)\ \mathit{tree}$$
$$\mathit{delete}\ x\ t = (\mathbf{let}\ (l, b, r) = \mathit{split}\ t\ x\ \mathbf{in}\ \mathit{join2}\ l\ r)$$

The efficiency can be improved a little by taking the returned b into account.

```
split_min :: ('a × 'b) tree ⇒ 'a × ('a × 'b) tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
   else let (m, l') = split_min l in (m, join l' a r))

join2 :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
join2 l r
= (if r = ⟨⟩ then l else let (m, r') = split_min r in join l m r')

split :: ('a × 'b) tree ⇒ 'a ⇒ ('a × 'b) tree × bool × ('a × 'b) tree
split ⟨⟩ _ = (⟨⟩, False, ⟨⟩)
split ⟨l, (a, _), r⟩ x
= (case cmp x a of
    LT ⇒ let (l₁, b, l₂) = split l x in (l₁, b, join l₂ a r) |
    EQ ⇒ (l, True, r) |
    GT ⇒ let (r₁, b, r₂) = split r x in (join l a r₁, b, r₂))
```

Fig. 10.3. Auxiliary functions

But we have bigger functions to fry: union, intersection and difference. They are shown in Figure 10.4. All three are divide-and-conquer algorithms that follow the same schema: both input trees are split at an element a (by construction or explicitly), the algorithm is applied recursively to the two trees of the elements below a and to the two trees of the elements above a, and the two results are suitably joined.

10.2.1 Correctness

We need to prove that *union*, *inter* and *diff* satisfy the specification in Figure 10.1 where *set* = *set_tree* and *invar t* = *inv t* ∧ *bst t*. That is, for each function we show its set-theoretic property and preservation of *inv* and *bst* using the assumptions in Figure 10.2. Most of the proofs in this section are obvious and automatic inductions and we do not discuss them.

First we need to prove suitable properties of the auxiliary functions *split_min*, *join2* and *split*:

$$\begin{array}{l} \mathit{split\_min}\ t = (m,\ t') \wedge t \neq \langle\rangle \longrightarrow \\ m \in \mathit{set\_tree}\ t \wedge \mathit{set\_tree}\ t = \{m\} \cup \mathit{set\_tree}\ t' \end{array}$$

$$\begin{array}{l} \mathit{split\_min}\ t = (m,\ t') \wedge \mathit{bst}\ t \wedge t \neq \langle\rangle \longrightarrow \\ \mathit{bst}\ t' \wedge (\forall x{\in}\mathit{set\_tree}\ t'.\ m < x) \end{array}$$

$$\mathit{split\_min}\ t = (m,\ t') \wedge \mathit{inv}\ t \wedge t \neq \langle\rangle \longrightarrow \mathit{inv}\ t'$$

```
union :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
union ⟨⟩ t = t
union t ⟨⟩ = t
union ⟨l₁, (a, _), r₁⟩ t₂
= (let (l₂, b₂, r₂) = split t₂ a
   in join (union l₁ l₂) a (union r₁ r₂))

inter :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
inter ⟨⟩ t = ⟨⟩
inter t ⟨⟩ = ⟨⟩
inter ⟨l₁, (a, _), r₁⟩ t₂
= (let (l₂, b₂, r₂) = split t₂ a;
       l' = inter l₁ l₂; r' = inter r₁ r₂
   in if b₂ then join l' a r' else join2 l' r')

diff :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
diff ⟨⟩ t = ⟨⟩
diff t ⟨⟩ = t
diff t₁ ⟨l₂, (a, _), r₂⟩
= (let (l₁, b₁, r₁) = split t₁ a
   in join2 (diff l₁ l₂) (diff r₁ r₂))
```

Fig. 10.4. Union, intersection and difference

$$\textit{set\_tree}\ (\textit{join2}\ l\ r) = \textit{set\_tree}\ l \cup \textit{set\_tree}\ r \tag{10.5}$$

$$\textit{bst}\ l \wedge \textit{bst}\ r \wedge (\forall x{\in}\textit{set\_tree}\ l.\ \forall y{\in}\textit{set\_tree}\ r.\ x < y) \longrightarrow$$
$$\textit{bst}\ (\textit{join2}\ l\ r)$$

$$\textit{inv}\ l \wedge \textit{inv}\ r \longrightarrow \textit{inv}\ (\textit{join2}\ l\ r)$$

$$\begin{aligned}&\textit{split}\ t\ x = (l,\ b,\ r) \wedge \textit{bst}\ t \longrightarrow\\ &\textit{set\_tree}\ l = \{a \in \textit{set\_tree}\ t \mid a < x\} \wedge\\ &\textit{set\_tree}\ r = \{a \in \textit{set\_tree}\ t \mid x < a\} \wedge\\ &b = (x \in \textit{set\_tree}\ t) \wedge \textit{bst}\ l \wedge \textit{bst}\ r\end{aligned} \tag{10.6}$$

$$\textit{split}\ t\ x = (l,\ b,\ r) \wedge \textit{inv}\ t \longrightarrow \textit{inv}\ l \wedge \textit{inv}\ r$$

The correctness properties of *insert* and *delete* are trivial consequences and are not shown. We move on to *union*. Its correctness properties are concretizations of the properties (*union*) and (*union-inv*) in Figure 10.1:

$$\textit{bst}\ t_2 \longrightarrow \textit{set\_tree}\ (\textit{union}\ t_1\ t_2) = \textit{set\_tree}\ t_1 \cup \textit{set\_tree}\ t_2$$

$$\textit{bst}\ t_1 \wedge \textit{bst}\ t_2 \longrightarrow \textit{bst}\ (\textit{union}\ t_1\ t_2)$$

$$\textit{inv}\ t_1 \wedge \textit{inv}\ t_2 \longrightarrow \textit{inv}\ (\textit{union}\ t_1\ t_2)$$

All three *union* properties are proved by computation induction. The first property follows easily from assumption (10.1) and (10.6). The assumption $bst\ t_2$ (but not $bst\ t_1$) is required because t_2 is split and (10.6) requires *bst*. Preservation of *bst* follows from assumption (10.2) with the help of the first *union* property and the preservation of *bst* by *split*. Preservation of *inv* follows from assumptions (10.3) and (10.4) with the help of the preservation of *inv* by *split*.

The correctness properties of *inter* look similar:

$$bst\ t_1 \wedge bst\ t_2 \longrightarrow set\_tree\ (inter\ t_1\ t_2) = set\_tree\ t_1 \cap set\_tree\ t_2$$

$$bst\ t_1 \wedge bst\ t_2 \longrightarrow bst\ (inter\ t_1\ t_2)$$

$$inv\ t_1 \wedge inv\ t_2 \longrightarrow inv\ (inter\ t_1\ t_2)$$

The proof of the preservation properties are automatic but the proof of the *set_tree* property is more involved than the corresponding proof for *union* and we take a closer look at the induction. We focus on the case $t_1 = \langle l_1, (a, \_), r_1\rangle$ and $t_2 \neq \langle\rangle$. Let $L_1 = set\_tree\ l_1$ and $R_1 = set\_tree\ r_1$. Let $(l_2, b, r_2) = split\ t_2\ a$, $L_2 = set\_tree\ l_2$, $R_2 = set\_tree\ r_2$ and $A = (\textbf{if}\ b\ \textbf{then}\ \{a\}\ \textbf{else}\ \{\})$. The separation properties

$$a \notin L_1 \cup R_1 \quad a \notin L_2 \cup R_2$$
$$L_2 \cap R_2 = \{\} \quad L_1 \cap R_2 = \{\} \quad L_2 \cap R_1 = \{\}$$

follow from $bst\ t_1$, $bst\ t_2$ and (10.6). Now for the main proof:

$$\begin{array}{ll} set\_tree\ t_1 \cap set\_tree\ t_2 & \\ = (L_1 \cup R_1 \cup \{a\}) \cap (L_2 \cup R_2 \cup A) & \text{by (10.6), } bst\ t_2 \\ = L_1 \cap L_2 \cup R_1 \cap R_2 \cup A & \text{by the separation properties} \\ = set\_tree\ (inter\ t_1\ t_2) & \text{by (10.1), (10.5), IHs, } bst\ t_1,\ bst\ t_2\text{, (10.6)} \end{array}$$

The correctness properties of *diff* follow the same pattern and their proofs are similar to the proofs of the *inter* properties. This concludes the generic join approach.

10.3 Joining Red-Black Trees

This section shows how to implement *join* efficiently on red-black trees. The basic idea is simple: descend along the spine of the higher of the two trees until reaching a subtree whose height is the same as the height of the lower tree. With suitable changes this works for other balanced trees as well [9]. The function definitions are shown in Figure 10.5. Function *join* calls *joinR* (descending along the right spine of l) if l is the higher tree, or calls *joinL* (descending along the left spine of r) if r is the higher tree, or returns $B\ l\ x\ r$

```
joinL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinL l x r
= (if bh r ≤ bh l then R l x r
   else case r of
          ⟨l', (x', Red), r'⟩ ⇒ R (joinL l x l') x' r' |
          ⟨l', (x', Black), r'⟩ ⇒ baliL (joinL l x l') x' r')

joinR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinR l x r
= (if bh l ≤ bh r then R l x r
   else case l of
          ⟨l', (x', Red), r'⟩ ⇒ R l' x' (joinR r' x r) |
          ⟨l', (x', Black), r'⟩ ⇒ baliR l' x' (joinR r' x r))

join :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
join l x r
= (if bh r < bh l then paint Black (joinR l x r)
   else if bh l < bh r then paint Black (joinL l x r) else B l x r)
```

Fig. 10.5. Function *join* on red-black trees

otherwise. The running time is linear in the black height (and thus logarithmic in the size) if we assume that the black height is stored in each node; our implementation of red-black trees would have to be augmented accordingly. Note that in *joinR* (and similarly in *joinL*) the comparison is not $bh\ l = bh\ r$ but $bh\ l \leq bh\ r$ to simplify the proofs.

10.3.1 Correctness

We need to prove that *join* on red-black trees (and a suitable *inv*) satisfies its specification in Figure 10.2. We start with properties of *joinL*; the properties of function *joinR* are completely symmetric. These are the automatically provable inductive properties:

$$\begin{aligned}
&invc\ l \wedge invc\ r \wedge invh\ l \wedge invh\ r \wedge bh\ l \leq bh\ r \longrightarrow\\
&invc2\ (joinL\ l\ x\ r) \wedge\\
&(bh\ l \neq bh\ r \wedge color\ r = Black \longrightarrow invc\ (joinL\ l\ x\ r)) \wedge\\
&invh\ (joinL\ l\ x\ r) \wedge bh\ (joinL\ l\ x\ r) = bh\ r
\end{aligned}$$

$$bh\ l \leq bh\ r \longrightarrow set\_tree\ (joinL\ l\ x\ r) = set\_tree\ l \cup \{x\} \cup set\_tree\ r$$

$$bst\ \langle l,\ (a,\ n),\ r\rangle \wedge bh\ l \leq bh\ r \longrightarrow bst\ (joinL\ l\ a\ r)$$

Because *joinL* employs *baliL* from the chapter on red-black trees, the proof of the first property makes use of the *baliL* property shown in Section 8.2.1.

We define the invariant *inv* required for the specification in Figure 10.2 as follows:

$$inv\ t = (invc\ t \wedge inv\ h\ t)$$

Although weaker than *rbt*, it still guarantees logarithmic height (see Exercise 8.1). Note that *rbt* itself does not work because it does not satisfy property (10.4). The properties of *join* and *inv* are now easy consequences of the *joinL* (and *joinR*) properties shown above.

10.4 Exercises

Exercise 10.1. Define an alternative version *diff*1 of *diff* where in the third equation pattern matching is on t_1 and t_2 is *split*. Prove $bst\ t_1 \wedge bst\ t_2 \longrightarrow set\_tree\ (diff1\ t_1\ t_2) = set\_tree\ t_1 - set\_tree\ t_2$.

Exercise 10.2. Following the general idea of the join function for red-black trees, define a join function for 2-3-trees. Start with two functions *joinL*, $joinR :: 'a\ tree23 \Rightarrow 'a \Rightarrow 'a\ tree23 \Rightarrow 'a\ upI$ and combine them into the overall join function:

$$join :: 'a\ tree23 \Rightarrow 'a \Rightarrow 'a\ tree23 \Rightarrow 'a\ tree23$$

Prove the following correctness properties:

$$complete\ l \wedge complete\ r \longrightarrow complete\ (join\ l\ x\ r)$$

$$complete\ l \wedge complete\ r \longrightarrow$$
$$inorder\ (join\ l\ x\ r) = inorder\ l\ @\ x\ \#\ inorder\ r$$

The corresponding (and needed) properties of *joinL* and *joinR* are slightly more involved.

Bibliographic Remarks

The join approach goes back to Adams [1]. Blelloch *et al.* [9] generalized the approach from weight-balanced trees to AVL trees, red-black trees and treaps. In particular they proved the $O(m \lg(\frac{n}{m} + 1))$ bound for the work (and an $O(\lg m \lg n)$ bound for the span).

11

Arrays via Braun Trees

Braun trees are a subclass of almost complete trees. In this chapter we explore their use as arrays and in Chapter 16 as priority queues.

11.1 Array

So far we have discussed sets (or maps) over some arbitrary linearly ordered type. Now we specialize that linearly ordered type to *nat* to model arrays. In principle we could model arrays as maps from a subset of natural numbers to the array elements. Because arrays are contiguous, it is more appropriate to model them as lists. The type *'a list* comes with two array-like operations (see Appendix A):

Indexing: $xs\ !\ n$ is the nth element of the list xs.
Updating: $xs[n := x]$ is xs with the nth element replaced by x.

By convention, indexing starts with $n = 0$. If $n \geq$ *length xs* then $xs\ !\ n$ and $xs[n := x]$ are underdefined: they are defined terms but we do not know what their value is.

Note that operationally, indexing and updating take time linear in the index, which may appear inappropriate for arrays. However, the type of lists is only an abstract model that specifies the desired functional behaviour of arrays but not their running time complexity.

The ADT of arrays is shown in Figure 11.1. Type *'ar* is the type of arrays, type *'a* the type of elements in the arrays. The abstraction function *list* abstracts arrays to lists. It would make perfect sense to include *list* in the interface as well. In fact, our implementation below comes with a (reasonably efficiently) executable definition of *list*.

The behaviour of *lookup*, *update*, *size* and *array* is specified in terms of their counterparts on lists and requires that the invariant is preserved.

ADT $\mathit{Array} =$

interface
$\mathit{lookup} :: {'}ar \Rightarrow nat \Rightarrow {'}a$
$\mathit{update} :: nat \Rightarrow {'}a \Rightarrow {'}ar \Rightarrow {'}ar$
$\mathit{len} :: {'}ar \Rightarrow nat$
$\mathit{array} :: {'}a\ list \Rightarrow {'}ar$

abstraction $\mathit{list} :: {'}ar \Rightarrow {'}a\ list$
invariant $\mathit{invar} :: {'}ar \Rightarrow bool$

specification
$invar\ ar \wedge n < len\ ar \longrightarrow lookup\ ar\ n = list\ ar\ !\ n$ (*lookup*)
$invar\ ar \wedge n < len\ ar \longrightarrow invar\ (update\ n\ x\ ar)$ (*update-inv*)
$invar\ ar \wedge n < len\ ar \longrightarrow list\ (update\ n\ x\ ar) = (list\ ar)[n := x]$ (*update*)
$invar\ ar \longrightarrow len\ ar = |list\ ar|$ (*len*)
$invar\ (array\ xs)$ (*array-inv*)
$list\ (array\ xs) = xs$ (*array*)

Fig. 11.1. ADT *Array*

What distinguishes the specifications of *lookup* and *update* from the standard schema (see Chapter 6) is that they carry a size precondition because the result of *lookup* and *update* is only specified if the index is less than the size of the array.

11.2 Braun Trees

One can implement arrays by any one of the many search trees presented in this book. Instead we take advantage of the fact that the keys are natural numbers and implement arrays by so-called **Braun trees** which are almost complete and thus have minimal height.

The basic idea is to index a node in a binary tree by the non-zero bit string that leads from the root to that node in the following fashion. Starting from the least significant bit and while we have not reached the leading 1 (which is ignored), we examine the bits one by one. If the current bit is 0, descend into the left child, otherwise into the right child. Instead of bit strings we use the natural numbers ≥ 1 that they represent. The Braun tree with nodes indexed by 1–15 is shown in Figure 11.2. The numbers are the indexes and not the elements stored in the nodes. For example, the index 14 is 0111 in binary (least significant bit first). If you follow the path left-right-right corresponding to 011 in Figure 11.2 you reach node 14.

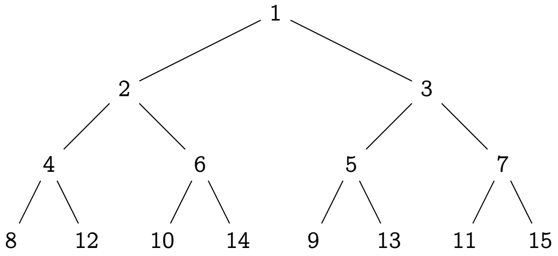
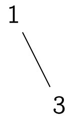


Fig. 11.2. Braun tree with nodes indexed by 1–15

A tree t is suitable for representing an array if the set of indexes of all its nodes is the interval $\{1..|t|\}$. The following tree is unsuitable because the node indexed by 2 is missing:



It turns out that the following invariant guarantees that a tree t contains exactly the nodes indexed by 1, ..., $|t|$:

```
braun :: 'a tree ⇒ bool
braun ⟨⟩ = True
braun ⟨l, _, r⟩ = ((|l| = |r| ∨ |l| = |r| + 1) ∧ braun l ∧ braun r)
```

The disjunction can alternatively be expresses as $|r| \leq |l| \leq |r| + 1$. We call a tree a **Braun tree** iff it satisfies predicate *braun*.

Although we do not need or prove this here, it is interesting to note that a tree that contains exactly the nodes indexed by 1, ..., $|t|$ is a Braun tree.

Let us now prove the earlier claim that Braun trees are almost complete. First, a lemma about the composition of almost complete trees:

Lemma 11.1.
$\mathit{acomplete}\ l \wedge \mathit{acomplete}\ r \wedge |l| = |r| + 1 \longrightarrow \mathit{acomplete}\ \langle l,\ x,\ r\rangle$

Proof. Using Lemmas 4.7 and 4.8 and the assumptions we obtain

$$h\ \langle l,\ x,\ r\rangle = \lceil \lg\ (|r|_1 + 1)\rceil + 1 \qquad (*)$$

$$mh\ \langle l,\ x,\ r\rangle = \lfloor \lg\ |r|_1\rfloor + 1 \qquad (**)$$

Because $1 \leq |r|_1$ there is an i such that $2^i \leq |r|_1 < 2^{i\ +\ 1}$ and thus $2^i < |r|_1 + 1 \leq 2^{i\ +\ 1}$. This implies $i = \lfloor \lg\ |r|_1\rfloor$ and $i + 1 = \lceil \lg\ (|r|_1 + 1)\rceil$. Together with $(*)$ and $(**)$ this implies $\mathit{acomplete}\ \langle l,\ x,\ r\rangle$. □

Now we can show that all Braun trees are almost complete:

Lemma 11.2. $\mathit{braun}\ t \longrightarrow \mathit{acomplete}\ t$

Thus we know that Braun trees have optimal height (Lemma 4.6) and can even quantify it (Lemma 4.7).

Proof. The proof is by induction. We focus on the induction step where $t = \langle l,\ x,\ r\rangle$. By assumption we have $\mathit{acomplete}\ l$ and $\mathit{acomplete}\ r$. Because of $\mathit{braun}\ t$ we can distinguish two cases. First assume $|l| = |r| + 1$. The claim $\mathit{acomplete}\ t$ follows immediately from the previous lemma. Now assume $|l| = |r|$. By definition, there are four cases to consider when proving $\mathit{acomplete}\ t$. By symmetry it suffices to consider only two of them. If $h\ l \le h\ r$ and $\mathit{mh}\ r < \mathit{mh}\ l$ then $\mathit{acomplete}\ t$ reduces to $\mathit{acomplete}\ r$, which is true by assumption. Now assume $h\ l \le h\ r$ and $\mathit{mh}\ l \le \mathit{mh}\ r$. Because $|l| = |r|$, the fact that the height of an almost complete tree is determined uniquely by its size (Lemma 4.7) implies $h\ l = h\ r$ and thus $\mathit{acomplete}\ t$ reduces to $\mathit{acomplete}\ l$, which is again true by assumption. □

Note that the proof does not rely on the fact that it is the left child that is potentially one bigger than the right one; it merely requires that the difference in size between two siblings is at most 1.

11.3 Arrays via Braun Trees

In this section we implement arrays by means of Braun trees and verify correctness and complexity. We start by defining array-like functions on Braun trees. After the above explanation of Braun trees the following lookup function will not come as a surprise:

```
lookup1 :: 'a tree ⇒ nat ⇒ 'a
lookup1 ⟨l, x, r⟩ n
= (if n = 1 then x else lookup1 (if even n then l else r) (n div 2))
```

The least significant bit is the parity of the index and we advance to the next bit by *div* 2. The function is called *lookup*1 rather than *lookup* to emphasize that it expects the index to be at least 1. This simplifies the implementation via Braun trees but is in contrast to the *Array* interface where by convention indexing starts with 0.

Function *update*1 descends in the very same manner but also performs an update when reaching 1:

```
update1 :: nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree
update1 _ x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
update1 n x ⟨l, a, r⟩
= (if n = 1 then ⟨l, x, r⟩
   else if even n then ⟨update1 (n div 2) x l, a, r⟩
        else ⟨l, a, update1 (n div 2) x r⟩)
```

The second equation updates existing entries in case $n = 1$. The first equation, however, creates a new entry and thus supports extending the tree. That is, *update*1 $(|t| + 1)$ x t extends the tree with a new node x at index $|t| + 1$. Function *adds* iterates this process (again expecting $|t| + 1$ as the index) and thus adds a whole list of elements:

```
adds :: 'a list ⇒ nat ⇒ 'a tree ⇒ 'a tree
adds [] _ t = t
adds (x # xs) n t = adds xs (n + 1) (update1 (n + 1) x t)
```

The implementation of the *Array* interface in Figure 11.3 is just a thin wrapper around the corresponding functions on Braun trees. An array is represented as a pair of a Braun tree and its size. Note that although *update*1 can extend the tree, the specification and implementation of the array *update* function does not support that: n is expected to be below the length of the array. Flexible arrays are specified and implemented in Section 11.4.

```
lookup (t, _) n     = lookup1 t (n + 1)
update n x (t, m) = (update1 (n + 1) x t, m)
len (t, m)           = m
array xs             = (adds xs 0 ⟨⟩, |xs|)
```

Fig. 11.3. Array implementation via Braun trees

11.3.1 Functional Correctness

The invariant on arrays is obvious:

$$invar\ (t,\ l) = (braun\ t \wedge l = |t|)$$

The abstraction function *list* could be defined in the following intuitive way, where $[m..{<}n]$ is the list of natural numbers from m up to but excluding n (see Appendix A):

$$list\ t = map\ (lookup1\ t)\ [1..{<}|t| + 1]$$

Instead we define *list* recursively and derive the above equation later on

$$list :: {'a}\ tree \Rightarrow {'a}\ list$$

$$list\ \langle\rangle = []$$
$$list\ \langle l,\ x,\ r\rangle = x\ \#\ splice\ (list\ l)\ (list\ r)$$

This definition is best explained by looking at Figure 11.2. The subtrees with root 2 and 3 will be mapped to the lists [2, 4, 6, 8, 10, 12, 14] and [1, 3, 5, 7, 9, 11, 13, 15]. The obvious way to combine these two lists into [1, 2, 3, ..., 15] is to splice them:

$$splice :: {'a}\ list \Rightarrow {'a}\ list \Rightarrow {'a}\ list$$

$$splice\ []\ ys = ys$$
$$splice\ (x\ \#\ xs)\ ys = x\ \#\ splice\ ys\ xs$$

Note that because of this reasonably efficient ($O(n \lg n)$, see Section 11.3.2) implementation of *list* we can also regard *list* as part of the interface of arrays.

Before we embark on the actual proofs we state a helpful arithmetic truth that is frequently used implicitly below:

$$\begin{array}{l} braun\ \langle l,\ x,\ r\rangle \wedge n \in \{1..|\langle l,\ x,\ r\rangle|\} \wedge 1 < n \longrightarrow \\ (odd\ n \longrightarrow n\ div\ 2 \in \{1..|r|\}) \wedge (even\ n \longrightarrow n\ div\ 2 \in \{1..|l|\}) \end{array}$$

where $\{m..n\} = \{k \mid m \le k \wedge k \le m\}$.

We will now verify that the implementation in Figure 11.3 of the *Array* interface in Figure 11.1 satisfies the given specification.

We start with property (*len*), the correctness of function *len*. Because of the invariant, (*len*) follows directly from

$$|list\ t| = |t|$$

which is proved by induction. We will also use this property implicitly in many proofs below.

The following proposition implies the correctness property (*lookup*):

$$\mathit{braun}\ t \wedge i < |t| \longrightarrow \mathit{list}\ t\ !\ i = \mathit{lookup}1\ t\ (i + 1) \tag{11.1}$$

The proof is by induction and uses the following proposition that is also proved by induction:

$$\begin{aligned}&n < |\mathit{xs}| + |\mathit{ys}| \wedge |\mathit{ys}| \le |\mathit{xs}| \wedge |\mathit{xs}| \le |\mathit{ys}| + 1 \longrightarrow\\ &\mathit{splice}\ \mathit{xs}\ \mathit{ys}\ !\ n = (\textsf{if}\ \mathit{even}\ n\ \textsf{then}\ \mathit{xs}\ \textsf{else}\ \mathit{ys})\ !\ (n\ \mathit{div}\ 2)\end{aligned}$$

As a corollary to (11.1) we obtain that function *list* can indeed be expressed via *lookup*1:

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ t = \mathit{map}\ (\mathit{lookup}1\ t)\ [1..{<}|t| + 1] \tag{11.2}$$

It follows by **list extensionality**:

$$\mathit{xs} = \mathit{ys} \longleftrightarrow |\mathit{xs}| = |\mathit{ys}| \wedge (\forall\, i{<}|\mathit{xs}|.\ \mathit{xs}\ !\ i = \mathit{ys}\ !\ i)$$

Let us now verify *update* as implemented via *update*1. The following two preservation properties (proved by induction) prove (*update-inv*):

$$\begin{aligned}&\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow |\mathit{update}1\ n\ x\ t| = |t|\\ &\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow \mathit{braun}\ (\mathit{update}1\ n\ x\ t)\end{aligned}$$

The following property relating *lookup*1 and *update*1 is again proved by induction:

$$\begin{aligned}&\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow\\ &\mathit{lookup}1\ (\mathit{update}1\ n\ x\ t)\ m = (\textsf{if}\ n = m\ \textsf{then}\ x\ \textsf{else}\ \mathit{lookup}1\ t\ m)\end{aligned}$$

The last three properties together with (11.2) and list extensionality prove the following proposition, which implies (*update*):

$$\mathit{braun}\ t \wedge n \in \{1..|t|\} \longrightarrow \mathit{list}\ (\mathit{update}1\ n\ x\ t) = (\mathit{list}\ t)[n - 1 := x]$$

Finally we turn to the constructor *array*. It is implemented in terms of *adds* and *update*1. Their correctness is captured by the following properties whose inductive proofs build on each other:

$$\mathit{braun}\ t \longrightarrow |\mathit{update}1\ (|t| + 1)\ x\ t| = |t| + 1 \tag{11.3}$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{update}1\ (|t| + 1)\ x\ t) \tag{11.4}$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{update}1\ (|t| + 1)\ x\ t) = \mathit{list}\ t\ @\ [x] \tag{11.5}$$

$$\mathit{braun}\ t \longrightarrow |\mathit{adds}\ \mathit{xs}\ |t|\ t| = |t| + |\mathit{xs}| \wedge \mathit{braun}\ (\mathit{adds}\ \mathit{xs}\ |t|\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{adds}\ \mathit{xs}\ |t|\ t) = \mathit{list}\ t\ @\ \mathit{xs}$$

The last two properties imply the remaining proof obligations (*array-inv*) and (*array*). The proof of (11.5) requires the following two properties of *splice* which are proved by simultaneous induction:

$$\begin{aligned}&|\mathit{ys}| \le |\mathit{xs}| \longrightarrow \mathit{splice}\ (\mathit{xs}\ @\ [x])\ \mathit{ys} = \mathit{splice}\ \mathit{xs}\ \mathit{ys}\ @\ [x]\\ &|\mathit{xs}| \le |\mathit{ys}| + 1 \longrightarrow \mathit{splice}\ \mathit{xs}\ (\mathit{ys}\ @\ [y]) = \mathit{splice}\ \mathit{xs}\ \mathit{ys}\ @\ [y]\end{aligned}$$

11.3.2 Running Time Analysis

The running time of *lookup* and *update* is obviously logarithmic because of the logarithmic height of Braun trees. We sketch why *list* and *array* both have running time $O(n \lg n)$. Linear time versions are presented in Section 11.5.

Function *list* is similar to bottom-up merge sort and *splice* is similar to *merge*. We focus on *splice* because it performs almost all the work. Consider calling *list* on a complete tree of height h. At each level k (starting with 0 for the root) of the tree, *splice* is called 2^k times with lists of size (almost) 2^{h-k-1}. The running time of *splice* with lists of the same length is proportional to the size of the lists. Thus the running time at each level is $O(2^k 2^{h-k-1}) = O(2^{h-1}) = O(2^h)$. Thus all the splices together require time $O(h\, 2^h)$. Because complete trees have size $n = 2^h$, the bound $O(n \lg n)$ follows.

Function *array* is implemented via *adds* and thus via repeated calls of *update*1. At the beginning of Section 7.3 we show that because *update*1 has logarithmic complexity, calling it n times on a growing tree starting with a leaf takes time $\Theta(n \lg n)$.

11.4 Flexible Arrays

Flexible arrays can be grown and shrunk at either end. Figure 11.4 shows the specification of all four operations. (For *tl* and *butlast* see Appendix A.) *Array_Flex* extends the basis specification *Array* in Figure 11.1.

ADT $\mathit{Array\_Flex} = \mathit{Array}\ +$

interface

$\mathit{add\_lo} :: {'a} \Rightarrow {'ar} \Rightarrow {'ar}$
$\mathit{del\_lo} :: {'ar} \Rightarrow {'ar}$
$\mathit{add\_hi} :: {'a} \Rightarrow {'ar} \Rightarrow {'ar}$
$\mathit{del\_hi} :: {'ar} \Rightarrow {'ar}$

specification

$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{invar}\ (\mathit{add\_lo}\ a\ \mathit{ar})$ $(\mathit{add\_lo\text{-}inv})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ \mathit{ar}) = a\ \#\ \mathit{list}\ \mathit{ar}$ $(\mathit{add\_lo})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{invar}\ (\mathit{del\_lo}\ \mathit{ar})$ $(\mathit{del\_lo\text{-}inv})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ \mathit{ar}) = \mathit{tl}\ (\mathit{list}\ \mathit{ar})$ $(\mathit{del\_lo})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{invar}\ (\mathit{add\_hi}\ a\ \mathit{ar})$ $(\mathit{add\_hi\text{-}inv})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{list}\ (\mathit{add\_hi}\ a\ \mathit{ar}) = \mathit{list}\ \mathit{ar}\ @\ [a]$ $(\mathit{add\_hi})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{invar}\ (\mathit{del\_hi}\ \mathit{ar})$ $(\mathit{del\_hi\text{-}inv})$
$\mathit{invar}\ \mathit{ar} \longrightarrow \mathit{list}\ (\mathit{del\_hi}\ \mathit{ar}) = \mathit{butlast}\ (\mathit{list}\ \mathit{ar})$ $(\mathit{del\_hi})$

Fig. 11.4. ADT *Array_Flex*

Below we first implement the *Array_Flex* functions on Braun trees. In a final step an implementation of *Array_Flex* on (tree,size) pairs is derived.

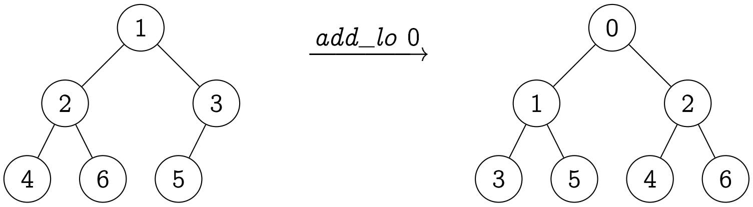
We have already seen that *update*1 adds an element at the high end. The inverse operation *del_hi* removes the high end, assuming that the given index is the size of the tree:

```
del_hi :: nat ⇒ 'a tree ⇒ 'a tree
del_hi _ ⟨⟩ = ⟨⟩
del_hi n ⟨l, x, r⟩
= (if n = 1 then ⟨⟩
   else if even n then ⟨del_hi (n div 2) l, x, r⟩
        else ⟨l, x, del_hi (n div 2) r⟩)
```

This was easy but extending an array at the low end seems hard because one has to shift the existing entries. However, Braun trees support a logarithmic implementation:

```
add_lo :: 'a ⇒ 'a tree ⇒ 'a tree
add_lo x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
add_lo x ⟨l, a, r⟩ = ⟨add_lo a r, x, l⟩
```

The intended functionality is *list* (*add_lo x t*) = *x* # *list t*. Function *add_lo* installs the new element *x* at the root of the tree. Because *add_lo* needs to shift the indices of the elements already in the tree, the left child (indices 2, 4, ...) becomes the new right child (indices 3, 5, ...). The old right child becomes the new left child with the old root *a* added in at index 2 and the remaining elements at indices 4, 6, ... In the following example, *add_lo* 0 transforms the left tree into the right one. The numbers in the nodes are the actual elements, not their indices.



Function *del_lo* simply reverses *add_lo* by removing the root and merging the children:

$$
\begin{array}{l}
\mathit{del\_lo} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{del\_lo}\ \langle\rangle = \langle\rangle \\
\mathit{del\_lo}\ \langle l,\ \_,\ r\rangle = \mathit{merge}\ l\ r
\end{array}
$$

$$
\begin{array}{l}
\mathit{merge} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{merge}\ \langle\rangle\ r = r \\
\mathit{merge}\ \langle l,\ a,\ r\rangle\ \mathit{rr} = \langle \mathit{rr},\ a,\ \mathit{merge}\ l\ r\rangle
\end{array}
$$

Figure 11.5 shows the obvious implementation of the functions in the *Array_Flex* specification from Figure 11.4 (on the left-hand side) with the help of the corresponding Braun tree operations (on the right-hand side). It is an extension of the basic array implementation from Figure 11.3. All *Array_Flex* functions have logarithmic time complexity because the corresponding Braun tree functions do because they descend along one branch of the tree.

$$
\begin{array}{ll}
\mathit{add\_lo}\ x\ (t,\ l) & = (\mathit{add\_lo}\ x\ t,\ l + 1) \\
\mathit{del\_lo}\ (t,\ l) & = (\mathit{del\_lo}\ t,\ l - 1) \\
\mathit{add\_hi}\ x\ (t,\ l) & = (\mathit{update1}\ (l + 1)\ x\ t,\ l + 1) \\
\mathit{del\_hi}\ (t,\ l) & = (\mathit{del\_hi}\ l\ t,\ l - 1)
\end{array}
$$

Fig. 11.5. Flexible array implementation via Braun trees

11.4.1 Functional Correctness

We now have to prove the properties in Figure 11.4. We have already dealt with *update*1 and thus *add_hi* above. Properties (*add_hi-inv*) and (*add_hi*) follow from (11.3), (11.4) and (11.5) stated earlier.

Correctness of *del_hi* on Braun trees is captured by the following two properties proved by induction:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_hi}\ |t|\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{del\_hi}\ |t|\ t) = \mathit{butlast}\ (\mathit{list}\ t) \tag{11.6}$$

They imply (*del_hi-inv*) and (*del_hi*). The proof of (11.6) requires the simple fact $\mathit{list}\ t = [] \longleftrightarrow t = \langle\rangle$ and the following property of *splice* which is proved by induction:

$$\begin{aligned}&\mathit{butlast}\ (\mathit{splice}\ \mathit{xs}\ \mathit{ys})\\&= (\textbf{if}\ |\mathit{ys}| < |\mathit{xs}|\ \textbf{then}\ \mathit{splice}\ (\mathit{butlast}\ \mathit{xs})\ \mathit{ys}\ \textbf{else}\ \mathit{splice}\ \mathit{xs}\ (\mathit{butlast}\ \mathit{ys}))\end{aligned}$$

Correctness of *add_lo* on Braun trees is captured by the following two properties proved by induction:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{add\_lo}\ x\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ t) = a\ \#\ \mathit{list}\ t$$

Properties (*add_lo-inv*) and (*add_lo*) follow directly from them.

Finally we turn to *del_lo*. Inductions (for *merge*) and case analyses (for *del_lo*) yield the following properties:

$$\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow \mathit{braun}\ (\mathit{merge}\ l\ r)$$

$$\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow \mathit{list}\ (\mathit{merge}\ l\ r) = \mathit{splice}\ (\mathit{list}\ l)\ (\mathit{list}\ r)$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_lo}\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ t) = \mathit{tl}\ (\mathit{list}\ t)$$

The last two properties imply (*del_lo-inv*) and (*del_lo*).

11.5 Bigger, Better, Faster, More!

In this section we meet efficient versions of some old and new functions on Braun trees. The implementation of the corresponding array operations is trivial and is not discussed.

11.5.1 Fast Size of Braun Trees

The size of a Braun tree can be computed without having to traverse the entire tree:

$$\mathit{size\_fast} :: {}'a\ \mathit{tree} \Rightarrow \mathit{nat}$$

$$\begin{aligned}&\mathit{size\_fast}\ \langle\rangle = 0\\&\mathit{size\_fast}\ \langle l,\ \_,\ r\rangle = (\textbf{let}\ n = \mathit{size\_fast}\ r\ \textbf{in}\ 1 + 2 \cdot n + \mathit{diff}\ l\ n)\end{aligned}$$

$$\mathit{diff} :: {}'a\ \mathit{tree} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$$

$$\begin{aligned}&\mathit{diff}\ \langle\rangle\ \_ = 0\\&\mathit{diff}\ \langle l,\ \_,\ r\rangle\ n\\&= (\textbf{if}\ n = 0\ \textbf{then}\ 1\\&\quad\ \textbf{else if}\ \mathit{even}\ n\ \textbf{then}\ \mathit{diff}\ r\ (n\ \mathit{div}\ 2 - 1)\ \textbf{else}\ \mathit{diff}\ l\ (n\ \mathit{div}\ 2))\end{aligned}$$

Function *size_fast* descends down the right spine, computes the size of a *Node* as if both children were the same size ($1 + 2 \cdot n$), but adds *diff l n* to compensate for bigger left children. Correctness of *size_fast*

Lemma 11.3. $\mathit{braun}\ t \longrightarrow \mathit{size\_fast}\ t = |t|$

follows from this property of *diff*:

$$\mathit{braun}\ t \wedge |t| \in \{n,\ n + 1\} \longrightarrow \mathit{diff}\ t\ n = |t| - n$$

The running time of *size_fast* is quadratic in the height of the tree (see Exercise 11.3).

11.5.2 Initializing a Braun Tree with a Fixed Value

Above we only considered the construction of a Braun tree from a list. Alternatively one may want to create a tree (array) where all elements are initialized to the same value. Of course one can call *update*1 *n* times, but one can also build the tree directly:

```
braun_of_naive x n
= (if n = 0 then ⟨⟩
   else let m = (n − 1) div 2
        in if odd n
           then ⟨braun_of_naive x m, x, braun_of_naive x m⟩
           else ⟨braun_of_naive x (m + 1), x,
                 braun_of_naive x m⟩)
```

This solution also has time complexity $O(n \lg n)$ but it can clearly be improved by sharing identical recursive calls. Function *braun2_of* shares as much as possible by producing trees of size n and $n + 1$ in parallel:

```
braun2_of :: 'a ⇒ nat ⇒ 'a tree × 'a tree
braun2_of x n
= (if n = 0 then (⟨⟩, ⟨⟨⟩, x, ⟨⟩⟩)
   else let (s, t) = braun2_of x ((n − 1) div 2)
        in if odd n then (⟨s, x, s⟩, ⟨t, x, s⟩) else (⟨t, x, s⟩, ⟨t, x, t⟩))

braun_of :: 'a ⇒ nat ⇒ 'a tree
braun_of x n = fst (braun2_of x n)
```

The running time is clearly logarithmic.

The correctness properties (see Appendix A for *replicate*)

$$\mathit{list}\ (\mathit{braun\_of}\ x\ n) = \mathit{replicate}\ n\ x$$

$$\mathit{braun}\ (\mathit{braun\_of}\ x\ n)$$

are corollaries of the more general statements

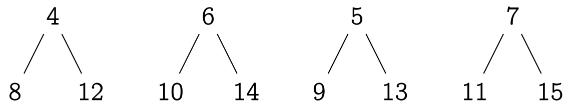
$$\mathit{braun2\_of}\ x\ n = (s,\ t) \longrightarrow$$
$$\mathit{list}\ s = \mathit{replicate}\ n\ x \wedge \mathit{list}\ t = \mathit{replicate}\ (n + 1)\ x$$

$$\mathit{braun2\_of}\ x\ n = (s,\ t) \longrightarrow$$
$$|s| = n \wedge |t| = n + 1 \wedge \mathit{braun}\ s \wedge \mathit{braun}\ t$$

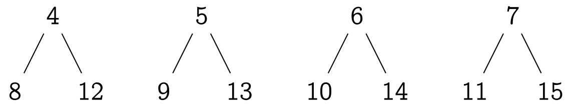
which can both be proved by induction.

11.5.3 Converting a List into a Braun Tree

We improve on function *adds* from Section 11.3 that has running time $\Theta(n \lg n)$ by developing a linear-time function. Given a list of elements $[1, 2, \ldots]$, we can subdivide it into sublists $[1]$, $[2, 3]$, $[4, \ldots, 7]$, $\ldots$ such that the kth sublist contains the elements of level k of the corresponding Braun tree. This is simply because on each level we have the entries whose index has $k{+}1$ bits. Thus we need to process the input list in chunks of size 2^k to produce the trees on level k. But we also need to get the order right. To understand how that works, consider the last two levels of the tree in Figure 11.2:



If we rearrange them in increasing order of the root labels



the following pattern emerges: the left subtrees are labeled $[8, \ldots, 11]$, the right subtrees $[12, \ldots, 15]$. Call t_i the tree with root label i. The correct order of subtrees, i.e. $t_4,\ t_6,\ t_5,\ t_7$, is restored when the three lists $[t_4,\ t_5]$, $[2,\ 3]$ (the labels above) and $[t_6,\ t_7]$ are combined into new trees by going through them simultaneously from left to right, yielding $[\langle t_4,\ 2,\ t_6\rangle,\ \langle t_5,\ 3,\ t_7\rangle]$, the level above.

Abstracting from this example we arrive at the following code. Loosely speaking, *brauns k xs* produces the Braun trees on level *k*.

```
brauns :: nat ⇒ 'a list ⇒ 'a tree list
brauns k xs
= (if xs = [] then []
    else let ys = take 2^k xs;
             zs = drop 2^k xs;
             ts = brauns (k + 1) zs
         in nodes ts ys (drop 2^k ts))
```

Function *brauns* chops off a chunk *ys* of size 2^k from the input list and recursively converts the remainder of the list into a list *ts* of (at most) 2^{k+1} trees. This list is (conceptually) split into *take* 2^k *ts* and *drop* 2^k *ts* which are combined with *ys* by function *nodes* that traverses its three argument lists simultaneously. As a local optimization, we pass all of *ts* rather than just *take* 2^k *ts* to *nodes*.

```
nodes :: 'a tree list ⇒ 'a list ⇒ 'a tree list ⇒ 'a tree list
nodes (l # ls) (x # xs) (r # rs) = ⟨l, x, r⟩ # nodes ls xs rs
nodes (l # ls) (x # xs) [] = ⟨l, x, ⟨⟩⟩ # nodes ls xs []
nodes [] (x # xs) (r # rs) = ⟨⟨⟩, x, r⟩ # nodes [] xs rs
nodes [] (x # xs) [] = ⟨⟨⟩, x, ⟨⟩⟩ # nodes [] xs []
nodes _ [] _ = []
```

Because the input list may not have exactly $2^n - 1$ elements, some of the chunks of elements and trees may be shorter than 2^k. To compensate for that, function *nodes* implicitly pads lists of trees at the end with leaves. This padding is the purpose of equations two to four.

The top-level function for turning a list into a tree simply extracts the first (and only) element from the list computed by *brauns* 0:

```
brauns1 :: 'a list ⇒ 'a tree
brauns1 xs = (if xs = [] then ⟨⟩ else brauns 0 xs ! 0)
```

Functional Correctness

The key correctness lemma below expresses a property of Braun trees: the subtrees on level k consist of all elements of the input list *xs* that are 2^k elements apart, starting from some offset. To state this concisely we define

```
take_nths :: nat ⇒ nat ⇒ 'a list ⇒ 'a list
take_nths _ _ [] = []
take_nths i k (x # xs)
= (if i = 0 then x # take_nths (2^k − 1) k xs
   else take_nths (i − 1) k xs)
```

The result of *take_nths i k xs* is every 2^k-th element in *drop i xs*.

A number of simple properties follow by easy inductions:

$$\mathit{take\_nths}\ i\ k\ (\mathit{drop}\ j\ \mathit{xs}) = \mathit{take\_nths}\ (i + j)\ k\ \mathit{xs} \tag{11.7}$$

$$\mathit{take\_nths}\ 0\ 0\ \mathit{xs} = \mathit{xs} \tag{11.8}$$

$$\mathit{splice}\ (\mathit{take\_nths}\ 0\ 1\ \mathit{xs})\ (\mathit{take\_nths}\ 1\ 1\ \mathit{xs}) = \mathit{xs} \tag{11.9}$$

$$\begin{aligned}&\mathit{take\_nths}\ i\ m\ (\mathit{take\_nths}\ j\ n\ \mathit{xs})\\ &= \mathit{take\_nths}\ (i \cdot 2^n + j)\ (m + n)\ \mathit{xs}\end{aligned} \tag{11.10}$$

$$\mathit{take\_nths}\ i\ k\ \mathit{xs} = [] \longleftrightarrow |\mathit{xs}| \le i \tag{11.11}$$

$$i < |\mathit{xs}| \longrightarrow \mathit{hd}\ (\mathit{take\_nths}\ i\ k\ \mathit{xs}) = \mathit{xs}\ !\ i \tag{11.12}$$

$$\begin{aligned}&|\mathit{xs}| = |\mathit{ys}| \lor |\mathit{xs}| = |\mathit{ys}| + 1 \longrightarrow\\ &\mathit{take\_nths}\ 0\ 1\ (\mathit{splice}\ \mathit{xs}\ \mathit{ys}) = \mathit{xs} \land\\ &\mathit{take\_nths}\ 1\ 1\ (\mathit{splice}\ \mathit{xs}\ \mathit{ys}) = \mathit{ys}\end{aligned} \tag{11.13}$$

$$\begin{aligned}&|\mathit{take\_nths}\ 0\ 1\ \mathit{xs}| = |\mathit{take\_nths}\ 1\ 1\ \mathit{xs}| \lor\\ &|\mathit{take\_nths}\ 0\ 1\ \mathit{xs}| = |\mathit{take\_nths}\ 1\ 1\ \mathit{xs}| + 1\end{aligned} \tag{11.14}$$

We also introduce a predicate relating a tree to a list:

```
braun_list :: 'a tree ⇒ 'a list ⇒ bool
braun_list ⟨⟩ xs = (xs = [])
braun_list ⟨l, x, r⟩ xs
= (xs ≠ [] ∧ x = hd xs ∧
   braun_list l (take_nths 1 1 xs) ∧
   braun_list r (take_nths 2 1 xs))
```

This definition may look a bit mysterious at first but it satisfies a simple specification: *braun_list t xs* ⟷ *braun t* ∧ *xs* = *list t* (see below). The idea of the above definition is that instead of relating ⟨*l*, *x*, *r*⟩ to *xs* via *splice* we invert the process and relate *l* and *r* to the even and odd numbered elements of *drop* 1 *xs*.

Lemma 11.4. *braun_list t xs* ⟷ *braun t* ∧ *xs* = *list t*

Proof. The proof is by induction on t. The base case is trivial. In the induction step the key properties are (11.14) to prove $braun\ t$ and (11.9) and (11.13) to prove $xs = list\ t$. □

The correctness proof of $brauns$ rests on a few simple inductive properties:

$$|nodes\ ls\ xs\ rs| = |xs| \tag{11.15}$$

$$\begin{aligned} & i < |xs| \longrightarrow \\ & nodes\ ls\ xs\ rs\ !\ i \\ & = \langle \textbf{if}\ i < |ls|\ \textbf{then}\ ls\ !\ i\ \textbf{else}\ \langle\rangle,\ xs\ !\ i, \\ & \quad \textbf{if}\ i < |rs|\ \textbf{then}\ rs\ !\ i\ \textbf{else}\ \langle\rangle\rangle \end{aligned} \tag{11.16}$$

$$|brauns\ k\ xs| = min\ |xs|\ 2^k \tag{11.17}$$

The main theorem expresses the following correctness property of the elements of $brauns\ k\ xs$: every tree $brauns\ k\ xs\ !\ i$ is a Braun tree and its list of elements is $take\_nths\ i\ k\ xs$:

Theorem 11.5. $i < min\ |xs|\ 2^k \longrightarrow$
$braun\_list\ (brauns\ k\ xs\ !\ i)\ (take\_nths\ i\ k\ xs)$

Proof. The proof is by induction on the length of xs. Assume $i < min\ |xs|\ 2^k$, which implies $xs \neq []$. Let $zs = drop\ 2^k\ xs$. Thus $|zs| < |xs|$ and therefore the IH applies to zs and yields the property

$$\begin{aligned} & \forall\, i\, j.\ j = i + 2^k \wedge i < min\ |zs|\ 2^{k+1} \longrightarrow \\ & \qquad braun\_list\ (ts\ !\ i)\ (take\_nths\ j\ (k+1)\ xs) \end{aligned} \tag{$*$}$$

where $ts = brauns\ (k+1)\ zs$. Let $ts' = drop\ 2^k\ ts$. Below we examine $nodes\ ts\ \_\ ts'\ !\ i$ with the help of (11.16). Thus there are four similar cases of which we only discuss one representative one: assume $i < |ts|$ and $i \geq |ts'|$.

$$\begin{aligned} & braun\_list\ (brauns\ k\ xs\ !\ i)\ (take\_nths\ i\ k\ xs) \\ \longleftrightarrow\ & braun\_list\ (nodes\ ts\ (take\ 2^k\ xs)\ ts'\ !\ i)\ (take\_nths\ i\ k\ xs) \\ \longleftrightarrow\ & braun\_list\ (ts\ !\ i)\ (take\_nths\ (2^k + i)\ (k+1)\ xs) \wedge \\ & \quad braun\_list\ \langle\rangle\ (take\_nths\ (2^{k+1} + i)\ (k+1)\ xs) \\ & \qquad\qquad \text{by (11.16), (11.10), (11.11), (11.12) and assumptions} \\ \longleftrightarrow\ & True \qquad\qquad \text{by } (*)\text{, (11.11), (11.17) and assumptions} \end{aligned}$$

□

Setting $i = k = 0$ in this theorem we obtain the correctness of $brauns1$ using Lemma 11.4 and (11.8):

Corollary 11.6. $braun\ (brauns1\ xs) \wedge list\ (brauns1\ xs) = xs$

Running Time Analysis

We focus on function *brauns*. In the step from *brauns* to T_{brauns} we simplify matters a little bit: we count only the expensive operations that traverse lists and ignore the other small additive constants. The time to evaluate *take n xs* and *drop n xs* is linear in $min\ n\ |xs|$ and we simply use $min\ n\ |xs|$. Evaluating *nodes ls xs rs* takes time linear in $|xs|$ and $|take\ n\ xs| = min\ n\ |xs|$. As a result we obtain the following definition of T_{brauns}:

```
T_brauns :: nat ⇒ 'a list ⇒ nat
T_brauns k xs
= (if xs = [] then 0
   else let ys = take 2^k xs; zs = drop 2^k xs; ts = brauns (k + 1) zs
        in 4 · min 2^k |xs| + T_brauns (k + 1) zs)
```

It is easy to prove that T_{brauns} is linear:

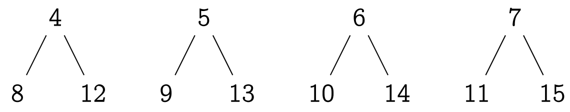
Lemma 11.7. $T_{brauns}\ k\ xs = 4 \cdot |xs|$

Proof. The proof is by induction on the length of *xs*. If $xs = []$ the claim is trivial. Now assume $xs \neq []$ and let $zs = drop\ 2^k\ xs$.

$$\begin{aligned}
&T_{brauns}\ k\ xs = T_{brauns}\ (k + 1)\ zs + 4 \cdot min\ 2^k\ |xs| \\
&= 4 \cdot |zs| + 4 \cdot min\ 2^k\ |xs| && \text{by IH} \\
&= 4 \cdot (|xs| - 2^k) + 4 \cdot min\ 2^k\ |xs| = 4 \cdot |xs| && \square
\end{aligned}$$

11.5.4 Converting a Braun Tree into a List

We improve on function *list* that has running time $O(n \lg n)$ by developing a linear-time version. Imagine that we want to invert the computation of *brauns*1 and thus of *brauns*. Thus it is natural to convert not merely a single tree but a list of trees. Looking once more at the reordered list of subtrees



the following strategy strongly suggests itself: first the roots, then the left children, then the right children. The recursive application of this strategy also takes care of the required reordering of the subtrees. Of course we have to ignore any leaves we encounter. This is the resulting function:

```
list_fast_rec :: 'a tree list ⇒ 'a list
list_fast_rec ts
= (let us = filter (λt. t ≠ ⟨⟩) ts
   in if us = [] then []
      else map value us @ list_fast_rec (map left us @ map right us))

value ⟨l, x, r⟩ = x
left ⟨l, x, r⟩ = l
right ⟨l, x, r⟩ = r
```

Termination of *list_fast_rec* is almost obvious because *left* and *right* remove the top node of a tree. Thus *size* seems the right measure. But if $ts = [\langle\rangle]$, the measure is 0 but it still leads to a recursive call (with argument []). This problem can be avoided with the measure function $\varphi = \mathit{sum\_list} \circ \mathit{map}\ f$ where $f = (\lambda t.\ 2 \cdot |t| + 1)$. Assume $ts \neq []$ and let $us = \mathit{filter}\ (\lambda t.\ t \neq \langle\rangle)\ ts$. We need to show that $\varphi\ (\mathit{map\ left\ us}\ @\ \mathit{map\ right\ us}) < \varphi\ ts$. Take some t in ts. If $t = \langle\rangle$, $f\ t = 1$ but t is no longer in us, i.e. the measure decreases by 1. If $t = \langle l, x, r\rangle$ then $f\ t = 2 \cdot |l| + 2 \cdot |r| + 3$ but $f\ (\mathit{left}\ t) + f\ (\mathit{right}\ t) = 2 \cdot |l| + 2 \cdot |r| + 2$ and thus the measure also decreases by 1. Because $ts \neq []$ this proves $\varphi\ (\mathit{map\ left\ us}\ @\ \mathit{map\ right\ us}) < \varphi\ ts$. We do not show the technical details.

Finally, the top level function to extract a list from a single tree:

```
list_fast :: 'a tree ⇒ 'a list
list_fast t = list_fast_rec [t]
```

From *list_fast* one can easily derive an efficient fold function on Braun trees that processes the elements in the tree in the order of their indexes.

Functional Correctness

We want to prove correctness of *list_fast*: $\mathit{list\_fast}\ t = \mathit{list}\ t$ if $\mathit{braun}\ t$. A direct proof of $\mathit{list\_fast\_rec}\ [t] = \mathit{list}\ t$ will fail and we need to generalize this statement to all lists of length 2^k. Reusing the infrastructure from the previous subsection this can be expressed as follows:

Theorem 11.8. $|ts| = 2^k \wedge (\forall\, i{<}2^k.\ \mathit{braun\_list}\ (ts\ !\ i)\ (\mathit{take\_nths}\ i\ k\ xs)) \longrightarrow \mathit{list\_fast\_rec}\ ts = xs$

Proof. The proof is by induction on the length of *xs*. Assume the two premises. There are two cases.

First assume $|xs| < 2^k$. Then

$$ts = map\ (\lambda x.\ \langle\langle\rangle,\ x,\ \langle\rangle\rangle)\ xs\ @\ replicate\ n\ \langle\rangle \qquad (*)$$

where $n = |ts| - |xs|$. This can be proved pointwise. Take some $i < 2^k$. If $i < |xs|$ then $take\_nths\ i\ k\ xs = take\ 1\ (drop\ i\ xs)$ (which can be proved by induction on xs). By definition of $braun\_list$ it follows that $t\ !\ i = \langle l,\ xs\ !\ i,\ r\rangle$ for some l and r such that $braun\_list\ l$ [] and $braun\_list\ l$ [] and thus $l = r = \langle\rangle$, i.e. $t\ !\ i = \langle\langle\rangle,\ xs\ !\ i,\ \langle\rangle\rangle$. If $\neg\ i < |xs|$ then $take\_nths\ i\ k\ xs =$ [] by (11.11) and thus $braun\_list\ (ts\ !\ i)$ [] by the second premise and thus $ts\ !\ i = \langle\rangle$ by definition of $braun\_list$. This concludes the proof of $(*)$. The desired $list\_fast\_rec\ ts = xs$ follows easily by definition of $list\_fast\_rec$.

Now assume $\neg\ |xs| < 2^k$. Then for all $i < 2^k$

$$\begin{array}{l}
ts\ !\ i \neq \langle\rangle \wedge value\ (ts\ !\ i) = xs\ !\ i\ \wedge \\
braun\_list\ (left\ (ts\ !\ i))\ (take\_nths\ (i + 2^k)\ (k + 1)\ xs)\ \wedge \\
braun\_list\ (right\ (ts\ !\ i))\ (take\_nths\ (i + 2 \cdot 2^k)\ (k + 1)\ xs)
\end{array}$$

follows from the second premise with the help of (11.10), (11.11) and (11.12). We obtain two consequences:

$$map\ value\ ts = take\ 2^k\ xs$$

$$list\_fast\_rec\ (map\ left\ ts\ @\ map\ right\ ts) = drop\ 2^k\ xs$$

The first consequence follows by pointwise reasoning, the second consequence with the help of the IH and (11.7). From these two consequences the desired conclusion $list\_fast\_rec\ ts = xs$ follows by definition of $list\_fast\_rec$. □

Running Time Analysis

We focus on $list\_fast\_rec$. In the step from $list\_fast\_rec$ to $T_{list\_fast\_rec}$ we simplify matters a little bit: we count only the expensive operations that traverse lists and ignore the other small additive constants. The time to evaluate $map\ value\ ts$, $map\ left\ ts$, $map\ right\ ts$, $filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$ and $ts\ @\ \_$ is linear in $|ts|$ and we simply use $|ts|$. As a result we obtain the following definition of $T_{list\_fast\_rec}$:

```
$T_{list\_fast\_rec} :: {'a}\ tree\ list \Rightarrow nat$
$T_{list\_fast\_rec}\ ts$
$= ($let $us = filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$
    in $|ts|\ +$
       $($if $us = []$ then $0$
        else $5 \cdot |us| + T_{list\_fast\_rec}\ (map\ left\ us\ @\ map\ right\ us)))$
```

The following inductive property is an abstraction of the core of the termination argument of *list_fast_rec* above.

$$(\forall\, t \in set\ ts.\ t \neq \langle\rangle) \longrightarrow$$
$$(\textstyle\sum_{t \leftarrow ts}\ k \cdot |t|) = (\textstyle\sum_{t \leftarrow map\ left\ ts\ @\ map\ right\ ts}\ k \cdot |t|) + k \cdot |ts| \qquad (11.18)$$

The suggestive notation $\sum x{\leftarrow}xs.\ f\ x$ abbreviates $sum\_list\ (map\ f\ xs)$.

Now we can state and prove a linear upper bound of $T_{list\_fast\_rec}$:

Theorem 11.9. $T_{list\_fast\_rec}\ ts \leq (\sum_{t \leftarrow ts} 7 \cdot |t| + 1)$

Proof. The proof is by induction on the size of *ts*, again using the measure function $\lambda t.\ 2 \cdot |t| + 1$ which decreases with recursive calls as we proved above. If $ts = []$ the claim is trivial. Now assume $ts \neq []$ and let $us = filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$ and $children = map\ left\ us\ @\ map\ right\ us$.

$$\begin{aligned}
&T_{list\_fast\_rec}\ ts = T_{list\_fast\_rec}\ children + 5 \cdot |us| + |ts| \\
&\leq (\textstyle\sum_{t \leftarrow children} 7 \cdot |t| + 1) + 5 \cdot |us| + |ts| && \text{by IH} \\
&= (\textstyle\sum_{t \leftarrow children} 7 \cdot |t|) + 7 \cdot |us| + |ts| \\
&= (\textstyle\sum_{t \leftarrow us} 7 \cdot |t|) + |ts| && \text{by (11.18)} \\
&\leq (\textstyle\sum_{t \leftarrow ts} 7 \cdot |t|) + |ts| = (\textstyle\sum_{t \leftarrow ts} 7 \cdot |t| + 1) && \square
\end{aligned}$$

11.6 Exercises

Exercise 11.1. Instead of first showing that Braun trees are almost complete, give a direct proof of $braun\ t \longrightarrow h\ t = \lceil \lg |t|_1 \rceil$ by first showing $braun\ t \longrightarrow 2^{h\ t} \leq 2 \cdot |t| + 1$ by induction.

Exercise 11.2. Show that function *bal* in Section 4.3.1 produces Braun trees: $n \leq |xs| \wedge bal\ n\ xs = (t, zs) \longrightarrow braun\ t$. (Isabelle hint: *bal* needs to be qualified as *Balance.bal*.)

Exercise 11.3. One can view Braun trees as tries (see Chapter 12) by indexing them not with a *nat* but a *bool list* where each bit tells us whether to go left or right (as explained at the start of Section 11.2). Function *nat_of* specifies the intended correspondence:

$$nat\_of :: bool\ list \Rightarrow nat$$

$$nat\_of\ [] = 1$$
$$nat\_of\ (b\ \#\ bs) = 2 \cdot nat\_of\ bs + (\textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

Define the counterparts of *lookup1* and *update1*

$$lookup\_trie :: 'a\ tree \Rightarrow bool\ list \Rightarrow 'a$$
$$update\_trie :: bool\ list \Rightarrow 'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$$

and prove their correctness:

$$\mathit{braun}\ t \land \mathit{nat\_of}\ \mathit{bs} \in \{1..|t|\} \longrightarrow$$
$$\mathit{lookup\_trie}\ t\ \mathit{bs} = \mathit{lookup1}\ t\ (\mathit{nat\_of}\ \mathit{bs})$$

$$\mathit{update\_trie}\ \mathit{bs}\ x\ t = \mathit{update1}\ (\mathit{nat\_of}\ \mathit{bs})\ x\ t$$

Exercise 11.4. Function *del_lo* is defined with the help of function *merge*. Define a recursive function $\mathit{del\_lo2} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$ without recourse to any auxiliary function and prove $\mathit{del\_lo2}\ t = \mathit{del\_lo}\ t$.

Exercise 11.5. Let *lh*, the “left height”, compute the length of the left spine of a tree. Prove that the left height of a Braun tree is equal to its height: $\mathit{braun}\ t \longrightarrow \mathit{lh}\ t = h\ t$

Exercise 11.6. Show that the running time of *size_fast* is quadratic in the height of the tree: Define the running time functions T_{diff} and $T_{\mathit{size\_fast}}$ (taking 0 time in the base cases) and prove $T_{\mathit{size\_fast}}\ t \le (h\ t)^2$.

Exercise 11.7. Prove correctness of function *braun_of_naive* defined in Section 11.5.2: $\mathit{list}\ (\mathit{braun\_of\_naive}\ x\ n) = \mathit{replicate}\ n\ x$.

Bibliographic Remarks

Braun trees were investigated by Braun and Rem [67] and later, in a functional setting, by Hoogerwoord [31] who coined the term “Braun tree”. Section 11.5 is partly based on work by Okasaki [61]. The whole chapter is based on work by Nipkow and Sewell [58].

12

Tries

A **trie** is a search tree where keys are strings, i.e. lists. A trie can be viewed as a tree-shaped finite automaton where the root is the start state. For example, the set of strings {a, an, can, car, cat} is encoded as the trie in Figure 12.1. The solid states are accepting, i.e. those nodes terminate the string leading to them.

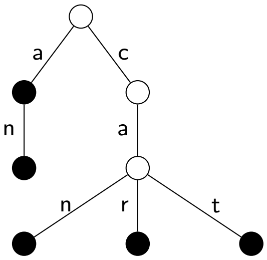


Fig. 12.1. A trie encoding {a, an, can, car, cat}

12.1 Abstract Tries via Functions

A nicely abstract model of tries is the following type:

```
datatype 'a trie = Nd bool ('a ⇒ 'a trie option)
```

In a node *Nd b f*, *b* indicates if it is an accepting node and *f* maps characters to sub-tries.

There is no invariant, i.e. the invariant is simply *True*: there are no ordering, balance or other requirements.

Below we use a variant of the function update notation $f(a := b)$:

$$f(a \mapsto b) \equiv f(a := \mathit{Some}\ b)$$

This is how the ADT *Set* is implemented by means of tries:

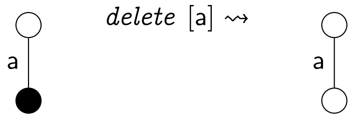
```
empty :: 'a trie
empty = Nd False (λ_. None)

isin :: 'a trie ⇒ 'a list ⇒ bool
isin (Nd b _) [] = b
isin (Nd _ m) (k # xs)
= (case m k of None ⇒ False | Some t ⇒ isin t xs)

insert :: 'a list ⇒ 'a trie ⇒ 'a trie
insert [] (Nd _ m) = Nd True m
insert (x # xs) (Nd b m)
= (let s = case m x of None ⇒ empty | Some t ⇒ t
   in Nd b (m(x ↦ insert xs s)))

delete :: 'a list ⇒ 'a trie ⇒ 'a trie
delete [] (Nd _ m) = Nd False m
delete (x # xs) (Nd b m)
= Nd b (case m x of None ⇒ m | Some t ⇒ m(x ↦ delete xs t))
```

The definitions are straightforward. But note that *delete* does not try to shrink the trie. For example:



Formally:

$$\begin{aligned}&\mathit{delete}\ [a]\ (\mathit{Nd}\ \mathit{False}\ [a \mapsto \mathit{Nd}\ \mathit{True}\ (\lambda\_.\ \mathit{None})])\\ &= \mathit{Nd}\ \mathit{False}\ [a \mapsto \mathit{Nd}\ \mathit{False}\ (\lambda\_.\ \mathit{None})]\end{aligned}$$

where $[x \mapsto t] \equiv (\lambda\_.\ \mathit{None})(x \mapsto t)$. The resulting trie is correct (it represents the empty set of strings) but could have been shrunk to *Nd False* ($\lambda\_$. *None*).

Functional Correctness

For the correctness proof we take a lazy approach and define the abstraction function in a trivial manner via *isin*:

$$set :: 'a\ trie \Rightarrow 'a\ list\ set$$
$$set\ t = \{xs \mid isin\ t\ xs\}$$

Correctness of *empty* and *isin* ($set\ empty = \{\}$ and $isin\ t\ xs = (xs \in set\ t)$) are trivial, correctness of insertion and deletion are easily proved by induction:

$$set\ (insert\ xs\ t) = set\ t \cup \{xs\}$$
$$set\ (delete\ xs\ t) = set\ t - \{xs\}$$

This simple model of tries leads to simple correctness proofs but is computationally inefficient because of the function space in $'a \Rightarrow 'a\ trie\ option$. In principle, any representation of this function space, for example by some search tree, works. However, such a generic theory is relatively complex. Hence we restrict ourselves to binary tries when exploring more efficient implementations of tries.

12.2 Binary Tries

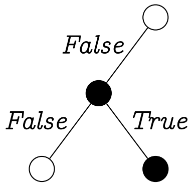
A **binary trie** is a trie over the alphabet *bool*. That is, binary tries represent sets of *bool list*. More concretely, every node has two children:

datatype $trie = Lf \mid Nd\ bool\ (trie \times trie)$

A binary trie, for example

$$Nd\ False\ (Nd\ True\ (Nd\ False\ (Lf,\ Lf),\ Nd\ True\ (Lf,\ Lf)),\ Lf)$$

can be visualized like this:



*Lf*s are not shown at all. The edge labels indicated that *False* refers to the left and *True* to the right child. This convention is encoded in the following auxiliary functions selecting from and modifying pairs:

$$\mathit{sel2} :: \mathit{bool} \Rightarrow {}'a \times {}'a \Rightarrow {}'a$$

$$\mathit{sel2}\ b\ (a_1,\ a_2) = (\textbf{if}\ b\ \textbf{then}\ a_2\ \textbf{else}\ a_1)$$

$$\mathit{mod2} :: ({}'a \Rightarrow {}'a) \Rightarrow \mathit{bool} \Rightarrow {}'a \times {}'a \Rightarrow {}'a \times {}'a$$

$$\mathit{mod2}\ f\ b\ (a_1,\ a_2) = (\textbf{if}\ b\ \textbf{then}\ (a_1,\ f\ a_2)\ \textbf{else}\ (f\ a_1,\ a_2))$$

The implementation of the *Set* interface is shown in Figure 12.2. Function *delete* shrinks a non-accepting *Nd* if both children have become empty.

$$\mathit{empty} :: \mathit{trie}$$

$$\mathit{empty} = \mathit{Lf}$$

$$\mathit{isin} :: \mathit{trie} \Rightarrow \mathit{bool\ list} \Rightarrow \mathit{bool}$$

$$\begin{array}{l}
\mathit{isin}\ \mathit{Lf}\ \mathit{ks} = \mathit{False} \\
\mathit{isin}\ (\mathit{Nd}\ b\ \mathit{lr})\ \mathit{ks} = (\textbf{case}\ \mathit{ks}\ \textbf{of}\ [\,] \Rightarrow b \mid k \# \mathit{ks}' \Rightarrow \mathit{isin}\ (\mathit{sel2}\ k\ \mathit{lr})\ \mathit{ks}')
\end{array}$$

$$\mathit{insert} :: \mathit{bool\ list} \Rightarrow \mathit{trie} \Rightarrow \mathit{trie}$$

$$\begin{array}{l}
\mathit{insert}\ [\,]\ \mathit{Lf} = \mathit{Nd}\ \mathit{True}\ (\mathit{Lf},\ \mathit{Lf}) \\
\mathit{insert}\ [\,]\ (\mathit{Nd}\ \_\ \mathit{lr}) = \mathit{Nd}\ \mathit{True}\ \mathit{lr} \\
\mathit{insert}\ (k \# \mathit{ks})\ \mathit{Lf} = \mathit{Nd}\ \mathit{False}\ (\mathit{mod2}\ (\mathit{insert}\ \mathit{ks})\ k\ (\mathit{Lf},\ \mathit{Lf})) \\
\mathit{insert}\ (k \# \mathit{ks})\ (\mathit{Nd}\ b\ \mathit{lr}) = \mathit{Nd}\ b\ (\mathit{mod2}\ (\mathit{insert}\ \mathit{ks})\ k\ \mathit{lr})
\end{array}$$

$$\mathit{delete} :: \mathit{bool\ list} \Rightarrow \mathit{trie} \Rightarrow \mathit{trie}$$

$$\begin{array}{l}
\mathit{delete}\ \_\ \mathit{Lf} = \mathit{Lf} \\
\mathit{delete}\ \mathit{ks}\ (\mathit{Nd}\ b\ \mathit{lr}) \\
= (\textbf{case}\ \mathit{ks}\ \textbf{of}\ [\,] \Rightarrow \mathit{node}\ \mathit{False}\ \mathit{lr} \\
\quad \mid k \# \mathit{ks}' \Rightarrow \mathit{node}\ b\ (\mathit{mod2}\ (\mathit{delete}\ \mathit{ks}')\ k\ \mathit{lr}))
\end{array}$$

$$\mathit{node}\ b\ \mathit{lr} = (\textbf{if}\ \neg\ b \wedge \mathit{lr} = (\mathit{Lf},\ \mathit{Lf})\ \textbf{then}\ \mathit{Lf}\ \textbf{else}\ \mathit{Nd}\ b\ \mathit{lr})$$

Fig. 12.2. Implementation of *Set* via binary tries

Functional Correctness

For the correctness proof we take the same lazy approach as above:

$$\mathit{set\_trie} :: \mathit{trie} \Rightarrow \mathit{bool\ list\ set}$$

$$\mathit{set\_trie}\ t = \{\mathit{xs} \mid \mathit{isin}\ t\ \mathit{xs}\}$$

We are also lazy in that we set the invariant to *True*. A more precise invariant would express that the tries are minimal, i.e. cannot be shrunk. See Exercise 12.2. It turns out that the correctness properties do not require this more precise invariant.

The two non-trivial correctness properties are

$$set\_trie\ (insert\ xs\ t) = set\_trie\ t \cup \{xs\} \tag{12.1}$$

$$set\_trie\ (delete\ xs\ t) = set\_trie\ t - \{xs\} \tag{12.2}$$

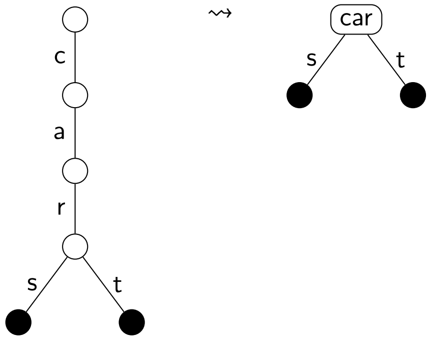
are simple consequences of the following inductive properties:

$$isin\ (insert\ xs\ t)\ ys = (xs = ys \lor isin\ t\ ys)$$

$$isin\ (delete\ xs\ t)\ ys = (xs \neq ys \land isin\ t\ ys)$$

12.3 Binary Patricia Tries

Tries can contain long branches without branching. These can be contracted by storing the branch directly in the start node. The result is called a **Patricia trie**. The following figure shows the contraction of a trie into a Patricia trie:



This is the data type of binary Patricia tries:

```
datatype trieP = LfP | NdP (bool list) bool (trieP × trieP)
```

The implementation of the *Set* interface via binary Patricia tries is shown in Figure 12.3.

Functional Correctness

This is an exercise in stepwise data refinement. We have already proved that *trie* implements *Set* via an abstraction function. Now we map *trieP* back

```
emptyP = LfP

isinP LfP ks = False
isinP (NdP ps b lr) ks
= (let n = |ps|
   in if ps = take n ks
      then case drop n ks of [] ⇒ b | k # x ⇒ isinP (sel2 k lr) x
      else False)

insertP ks LfP = NdP ks True (LfP, LfP)
insertP ks (NdP ps b lr)
= (case split ks ps of (_, [], []) ⇒ NdP ps True lr
   | (qs, [], p # ps') ⇒
       let t = NdP ps' b lr
       in NdP qs True (if p then (LfP, t) else (t, LfP))
   | (_, k # ks', []) ⇒ NdP ps b (mod2 (insertP ks') k lr)
   | (qs, k # ks', _ # ps') ⇒
       let tp = NdP ps' b lr; tk = NdP ks' True (LfP, LfP)
       in NdP qs False (if k then (tp, tk) else (tk, tp)))

deleteP ks LfP = LfP
deleteP ks (NdP ps b lr) =
(case split ks ps of
   (qs, ks', p # ps') ⇒ NdP ps b lr
 | (qs, k # ks', []) ⇒ nodeP ps b (mod2 (deleteP ks') k lr)
 | (qs, [], []) ⇒ nodeP ps False lr)

nodeP ps b lr = (if ¬ b ∧ lr = (LfP, LfP) then LfP else NdP ps b lr)
```

Fig. 12.3. Implementation of *Set* via binary Patricia tries

to *trie* via another abstraction function. Afterwards the overall correctness follows trivially by composing the two abstraction functions.

The abstraction function *abs_trieP* is defined via an auxiliary function that prefixes a trie with a bit list:

```
prefix_trie :: bool list ⇒ trie ⇒ trie
prefix_trie [] t = t
prefix_trie (k # ks) t
= (let t' = prefix_trie ks t
   in Nd False (if k then (Lf, t') else (t', Lf)))
```

$$\begin{array}{l} abs\_trieP :: trieP \Rightarrow trie \\ abs\_trieP\ LfP = Lf \\ abs\_trieP\ (NdP\ ps\ b\ (l,\ r)) \\ = prefix\_trie\ ps\ (Nd\ b\ (abs\_trieP\ l,\ abs\_trieP\ r)) \end{array}$$

Again we take a lazy approach and set the invariant on *trieP* to *True*.

Correctness of *emptyP* is trivial. Correctness of the remaining operations is proved by induction and requires a number of supporting inductive lemmas which we display before the corresponding correctness property.

Correctness of *isinP*:

$$isin\ (prefix\_trie\ ps\ t)\ ks = (ps = take\ |ps|\ ks \wedge isin\ t\ (drop\ |ps|\ ks))$$

$$isinP\ t\ ks = isin\ (abs\_trieP\ t)\ ks \tag{12.3}$$

Correctness of *insertP*:

$$prefix\_trie\ ks\ (Nd\ True\ (Lf,\ Lf)) = insert\ ks\ Lf$$

$$insert\ ps\ (prefix\_trie\ ps\ (Nd\ b\ lr)) = prefix\_trie\ ps\ (Nd\ True\ lr)$$

$$insert\ (ks\ @\ ks')\ (prefix\_trie\ ks\ t) = prefix\_trie\ ks\ (insert\ ks'\ t)$$

$$prefix\_trie\ (ps\ @\ qs)\ t = prefix\_trie\ ps\ (prefix\_trie\ qs\ t)$$

$$\begin{array}{l} split\ ks\ ps = (qs,\ ks',\ ps') \longrightarrow \\ ks = qs\ @\ ks' \wedge ps = qs\ @\ ps' \wedge \\ (ks' \neq [] \wedge ps' \neq [] \longrightarrow hd\ ks' \neq hd\ ps') \end{array}$$

$$abs\_trieP\ (insertP\ ks\ t) = insert\ ks\ (abs\_trieP\ t) \tag{12.4}$$

Correctness of *deleteP*:

$$\begin{array}{l} delete\ xs\ (prefix\_trie\ xs\ (Nd\ b\ (l,\ r))) \\ = (\textbf{if}\ (l,\ r) = (Lf,\ Lf)\ \textbf{then}\ Lf\ \textbf{else}\ prefix\_trie\ xs\ (Nd\ False\ (l,\ r))) \end{array}$$

$$\begin{array}{l} delete\ (xs\ @\ ys)\ (prefix\_trie\ xs\ t) \\ = (\textbf{if}\ delete\ ys\ t = Lf\ \textbf{then}\ Lf\ \textbf{else}\ prefix\_trie\ xs\ (delete\ ys\ t)) \end{array}$$

$$abs\_trieP\ (deleteP\ ks\ t) = delete\ ks\ (abs\_trieP\ t) \tag{12.5}$$

It is now trivial to obtain the correctness of the *trieP* implementation of sets. The abstraction function is simply the composition of the two abstraction abstraction functions: $set\_trieP = set\_trie \circ abs\_trieP$. The required correctness properties (ignoring *emptyP* and *isinP*)

$$set\_trieP\ (insertP\ xs\ t) = set\_trieP\ t \cup \{xs\}$$

$$set\_trieP\ (deleteP\ xs\ t) = set\_trieP\ t - \{xs\}$$

are trivial compositions of (12.1)/(12.2) and (12.4)/(12.5).

12.4 Exercises

Exercise 12.1. Rework the above theory of binary tries as follows. Eliminate the *bool* argument from constructor *Nd* by replacing *Nd* by two constructors representing *Nd True* and *Nd False*.

Exercise 12.2. Define the invariant *invar* that characterizes fully shrunk binary tries, i.e. tries where every non-*Lf* trie represents a non-empty set. Note that a trie represents the empty set if it does not contain any node *Nd True* _. Prove that *insert* and *delete* maintain the invariant.

Bibliographic Remarks

Tries were first sketched by De La Briandais [15] and described in more detail by Fredkin [20] who coined their name based on the word reTRIEval. However, “trie” is usually pronounced like “try” rather than “tree” to avoid confusion. Patricia tries are due to Morrison [50].

13

Huffman's Algorithm

Huffman's algorithm [32] is a simple and elegant procedure for constructing a binary tree with minimum weighted path length—a measure of cost that considers both the lengths of the paths from the root to the leaf nodes and the weights associated with the leaf nodes. The algorithm's main application is data compression: By equating leaf nodes with characters, and weights with character frequencies, we can use it to derive optimum binary codes. A *binary code* is a map from characters to non-empty sequences.

This chapter presents Huffman's algorithm and its optimality proof. The proof follows Knuth's informal argument [42]. This chapter's text is based on a published article [8]. An alternative formal proof, in Coq, is due to Théry [73].

13.1 Binary Codes

Suppose we want to encode strings over a finite source alphabet as sequences of bits. Fixed-length codes like ASCII are simple and fast, but they generally waste space. If we know the frequency w_a of each source symbol a, we can save space by using shorter code words for the most frequent symbols. We say that a variable-length code is *optimum* if it minimizes the sum $\sum_a w_a\,\delta_a$, where δ_a is the length of the binary code word for a.

As an example, consider the string '*abacabad*'. Encoding it with the code

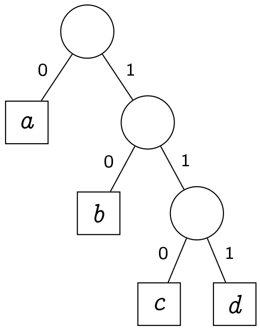
$$C_1 = \{a \mapsto 0,\ b \mapsto 10,\ c \mapsto 110,\ d \mapsto 111\}$$

gives the 14-bit code word 01001100100111. The code C_1 is optimum: No code that unambiguously encodes source symbols one at a time could do better than C_1 on the input '*abacabad*'. With a fixed-length code such as

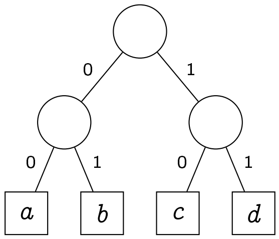
$$C_2 = \{a \mapsto 00,\ b \mapsto 01,\ c \mapsto 10,\ d \mapsto 11\}$$

we need at least 16 bits to encode the same string.

Binary codes can be represented by binary trees. For example, the trees



and



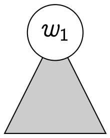
correspond to C_1 and C_2. The code word for a given symbol can be obtained as follows: Start at the root and descend toward the leaf node associated with the symbol one node at a time; emit a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The generated sequence of 0s and 1s is the code word.

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no code word is a prefix of another. Moreover, it is sufficient to consider only full binary trees (trees whose inner nodes all have two children), because any node with only one child can advantageously be eliminated by removing it and letting the child take its parent's place.

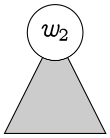
Each node in a code tree is assigned a *weight*. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. In diagrams, we often annotate the nodes with their weights.

13.2 The Algorithm

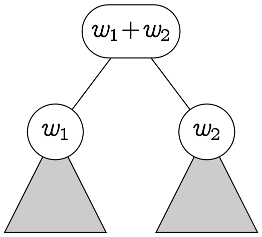
David Huffman [32] discovered a simple algorithm for constructing an optimum code tree for specified symbol frequencies: Create a forest consisting of only leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as initial weights for the nodes. Then pick the two trees



and



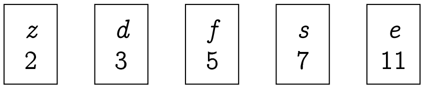
with the lowest weights and replace them with the tree



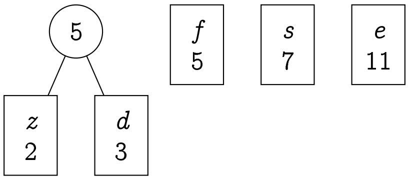
Repeat this process until only one tree is left.

As an illustration, executing the algorithm for the frequencies $f_d = 3$, $f_e = 11$, $f_f = 5$, $f_s = 7$, $f_z = 2$ gives rise to the following sequence of states:

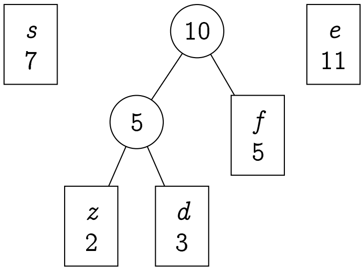
1.



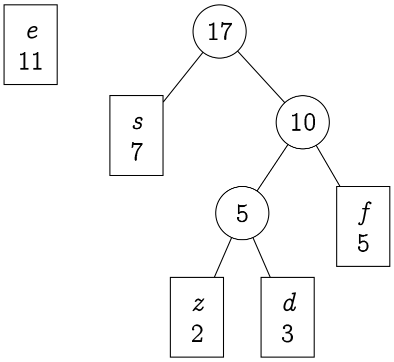
2.



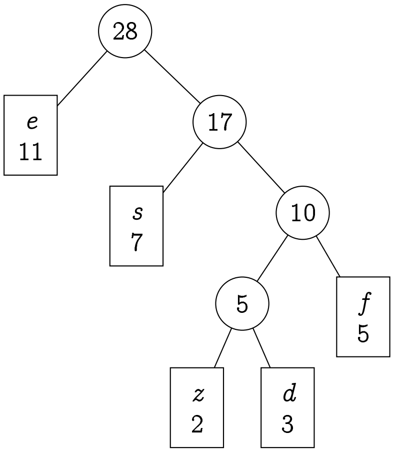
3.



4.



5.



The resulting tree is optimum for the given frequencies.

13.3 The Implementation

The functional implementation of the algorithm relies on the following type:

$$\textbf{datatype}\ 'a\ tree = Leaf\ nat\ 'a \mid Node\ nat\ ('a\ tree)\ ('a\ tree)$$

Leaf nodes are of the form *Leaf w a*, where a is a symbol and w is the frequency associated with a, and inner nodes are of the form *Node* $w\ t_1\ t_2$, where t_1 and t_2 are the left and right subtrees and w caches the sum of the weights of t_1 and t_2. The *cachedWeight* function extracts the weight stored in a node:

$$cachedWeight\ (Leaf\ w\ a) = w$$
$$cachedWeight\ (Node\ w\ t_1\ t_2) = w$$

The implementation builds on two additional auxiliary functions. The first one, *uniteTrees*, combines two trees by adding an inner node above them:

$$uniteTrees\ t_1\ t_2 = Node\ (cachedWeight\ t_1 + cachedWeight\ t_2)\ t_1\ t_2$$

The second function, *insortTree*, inserts a tree into a forest sorted by cached weight, preserving the sort order:

$$\begin{array}{l} \mathit{insortTree}\ u\ [\,] = [u] \\ \mathit{insortTree}\ u\ (t\ \#\ \mathit{ts}) \\ = (\textbf{if}\ \mathit{cachedWeight}\ u \le \mathit{cachedWeight}\ t\ \textbf{then}\ u\ \#\ t\ \#\ \mathit{ts} \\ \quad \textbf{else}\ t\ \#\ \mathit{insortTree}\ u\ \mathit{ts}) \end{array}$$

The main function that implements Huffman's algorithm follows:

$$\begin{array}{l} \mathit{huffman}\ [t] = t \\ \mathit{huffman}\ (t_1\ \#\ t_2\ \#\ \mathit{ts}) = \mathit{huffman}\ (\mathit{insortTree}\ (\mathit{uniteTrees}\ t_1\ t_2)\ \mathit{ts}) \end{array}$$

The function should initially be invoked with a non-empty list of leaf nodes sorted by weight. It repeatedly unites the first two trees of the forest it receives as argument until a single tree is left.

13.4 Basic Auxiliary Functions Needed for the Proof

This section introduces basic concepts such as alphabet, consistency, and optimality, which are needed to state the correctness and optimality of Huffman's algorithm. The next section introduces more specialized functions that arise in the proof.

The *alphabet* of a code tree is the set of symbols appearing in the tree's leaf nodes:

$$\begin{array}{l} \mathit{alphabet}\ (\mathit{Leaf}\ w\ a) = \{a\} \\ \mathit{alphabet}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{alphabet}\ t_1 \cup \mathit{alphabet}\ t_2 \end{array}$$

A tree is *consistent* if for each inner node the alphabets of the two subtrees are disjoint. Intuitively, this means that a symbol occurs in at most one leaf node. Consistency is a sufficient condition for δ_a (the length of the code word for a) to be uniquely defined. This well-formedness property appears as an assumption in many of the Isabelle lemmas. The definition follows:

$$\begin{array}{l} \mathit{consistent}\ (\mathit{Leaf}\ w\ a) = \mathit{True} \\ \mathit{consistent}\ (\mathit{Node}\ w\ t_1\ t_2) \\ = (\mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \{\} \wedge \mathit{consistent}\ t_1 \wedge \mathit{consistent}\ t_2) \end{array}$$

The *depth* of a symbol (which we wrote as δ_a above) is the length of the path from the root to that symbol, or equivalently the length of the code word for the symbol:

$$\begin{array}{l}
\mathit{depth}\ (\mathit{Leaf}\ w\ b)\ a = 0\\
\mathit{depth}\ (\mathit{Node}\ w\ t_1\ t_2)\ a\\
= (\textbf{if}\ a \in \mathit{alphabet}\ t_1\ \textbf{then}\ \mathit{depth}\ t_1\ a + 1\\
\quad \textbf{else if}\ a \in \mathit{alphabet}\ t_2\ \textbf{then}\ \mathit{depth}\ t_2\ a + 1\ \textbf{else}\ 0)
\end{array}$$

By convention, symbols that do not occur in the tree or that occur at the root of a one-node tree are given a depth of 0. If a symbol occurs in several leaf nodes (of an inconsistent tree), the depth is arbitrarily defined in terms of the leftmost node labeled with that symbol.

The *height* of a tree is the length of the longest path from the root to a leaf node, or equivalently the length of the longest code word:

$$\begin{array}{l}
\mathit{height}\ (\mathit{Leaf}\ w\ a) = 0\\
\mathit{height}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{max}\ (\mathit{height}\ t_1)\ (\mathit{height}\ t_2) + 1
\end{array}$$

The *frequency* of a symbol (which we wrote as w_a above) is the sum of the weights attached to the leaf nodes labeled with that symbol:

$$\begin{array}{l}
\mathit{freq}\ (\mathit{Leaf}\ w\ a)\ b = (\textbf{if}\ b = a\ \textbf{then}\ w\ \textbf{else}\ 0)\\
\mathit{freq}\ (\mathit{Node}\ w\ t_1\ t_2)\ b = \mathit{freq}\ t_1\ b + \mathit{freq}\ t_2\ b
\end{array}$$

For consistent trees, the sum comprises at most one nonzero term. The frequency is then the weight of the leaf node labeled with the symbol, or 0 if there is no such node.

Two trees are *comparable* if they have the same alphabet and symbol frequencies. This is an important concept, because it allows us to state not only that the tree constructed by Huffman's algorithm is optimal but also that it has the expected alphabet and frequencies.

The *weight* function returns the weight of a tree:

$$\begin{array}{l}
\mathit{weight}\ (\mathit{Leaf}\ w\ a) = w\\
\mathit{weight}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2
\end{array}$$

In the *Node* case, we ignore the weight cached in the node and instead compute the tree's weight recursively.

The *cost* (or *weighted path length*) of a consistent tree is the sum $\sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \cdot \mathit{depth}\ t\ a$ (which we wrote as $\sum_a w_a \delta_a$ above). It is defined recursively by

$$\begin{aligned}&\mathit{cost}\ (\mathit{Leaf}\ w\ a) = 0\\&\mathit{cost}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2\end{aligned}$$

A tree is optimum if and only if its cost is not greater than that of any comparable tree:

$$\begin{aligned}&\mathit{optimum}\ t\\&= (\forall u.\ \mathit{consistent}\ u \longrightarrow\\&\qquad\quad \mathit{alphabet}\ t = \mathit{alphabet}\ u \longrightarrow\\&\qquad\quad \mathit{freq}\ t = \mathit{freq}\ u \longrightarrow \mathit{cost}\ t \le \mathit{cost}\ u)\end{aligned}$$

Tree functions are readily generalized to forests; for example, the alphabet of a forest is defined as the union of the alphabets of its trees. The forest generalizations have a subscript '$_F$' attached to their name (e.g. $\mathit{alphabet}_F$).

13.5 Other Functions Needed for the Proof

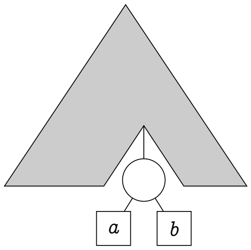
The optimality proof needs to interchange nodes in trees, to replace a two-leaf subtree with weights w_1 and w_2 by a single leaf of weight $w_1 + w_2$ and vice versa, and to refer to the two symbols with the lowest frequencies. These concepts are represented by four functions: *swapFourSyms*, *mergeSibling*, *splitLeaf*, and *minima*.

The four-way symbol interchange function *swapFourSyms* takes four symbols a, b, c, d with $a \ne b$ and $c \ne d$, and exchanges them so that a and b occupy c's and d's positions. A naive definition of this function would be $\mathit{swapSyms}\ (\mathit{swapSyms}\ t\ a\ c)\ b\ d$, where *swapSyms* exchanges two symbols. This naive definition fails in the face of aliasing: If $a = d$, but $b \ne c$, then $\mathit{swapFourSyms}\ a\ b\ c\ d$ would leave a in b's position.

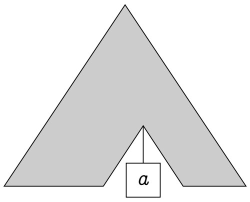
The following lemma about *swapSyms* captures the intuition that in order to minimize the cost, more frequent symbols should be encoded using fewer bits than less frequent ones:

Lemma 13.1. $\mathit{consistent}\ t \wedge a \in \mathit{alphabet}\ t \wedge b \in \mathit{alphabet}\ t \wedge \mathit{freq}\ t\ a \le \mathit{freq}\ t\ b \wedge \mathit{depth}\ t\ a \le \mathit{depth}\ t\ b \longrightarrow \mathit{cost}\ (\mathit{swapSyms}\ t\ a\ b) \le \mathit{cost}\ t$

Given a symbol a, the *mergeSibling* function transforms the tree



into



The frequency of a in the resulting tree is the sum of the original frequencies of a and b. The function is defined by the equations

```
mergeSibling (Leaf w_b b) a = Leaf w_b b
mergeSibling (Node w (Leaf w_b b) (Leaf w_c c)) a
= (if a = b ∨ a = c then Leaf (w_b + w_c) a
   else Node w (Leaf w_b b) (Leaf w_c c))
mergeSibling (Node w t_1 t_2) a
= Node w (mergeSibling t_1 a) (mergeSibling t_2 a)
```

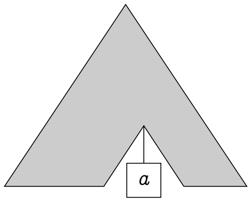
The defining equations are applied sequentially: The third equation is applicable only if the second one does not match.

The *sibling* function returns the label of the node that is the (left or right) sibling of the node labeled with the given symbol a in tree t. If a is not in t's alphabet or it occurs in a node with no sibling leaf, we simply return a. This gives us the nice property that if t is consistent, then $\mathit{sibling}\ t\ a \neq a$ if and only if a has a sibling. The definition, which is omitted here, distinguishes the same cases as *mergeSibling*.

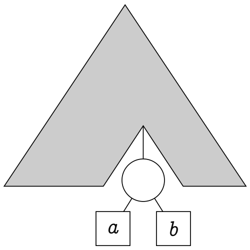
Using the *sibling* function, we can state that merging two sibling leaves with weights w_a and w_b decreases the cost by $w_a + w_b$:

Lemma 13.2. $\mathit{consistent}\ t \wedge \mathit{sibling}\ t\ a \neq a \longrightarrow \mathit{cost}\ (\mathit{mergeSibling}\ t\ a) + \mathit{freq}\ t\ a + \mathit{freq}\ t\ (\mathit{sibling}\ t\ a) = \mathit{cost}\ t$

The *splitLeaf* function undoes the merging performed by *mergeSibling*: Given two symbols a, b and two frequencies w_a, w_b, it transforms



into



In the resulting tree, a has frequency w_a and b has frequency w_b. We normally invoke *splitLeaf* with w_a and w_b such that $\mathit{freq}\ t\ a = w_a + w_b$. The definition follows:

$$
\begin{array}{l}
\mathit{splitLeaf}\ (\mathit{Leaf}\ w_c\ c)\ w_a\ a\ w_b\ b \\
= (\textbf{if}\ c = a\ \textbf{then}\ \mathit{Node}\ w_c\ (\mathit{Leaf}\ w_a\ a)\ (\mathit{Leaf}\ w_b\ b)\ \textbf{else}\ \mathit{Leaf}\ w_c\ c) \\
\mathit{splitLeaf}\ (\mathit{Node}\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b \\
= \mathit{Node}\ w\ (\mathit{splitLeaf}\ t_1\ w_a\ a\ w_b\ b)\ (\mathit{splitLeaf}\ t_2\ w_a\ a\ w_b\ b)
\end{array}
$$

Splitting a leaf with weight $w_a + w_b$ into two sibling leaves with weights w_a and w_b increases the cost by $w_a + w_b$:

Lemma 13.3. $\mathit{consistent}\ t \wedge a \in \mathit{alphabet}\ t \wedge \mathit{freq}\ t\ a = w_a + w_b \longrightarrow \mathit{cost}\ (\mathit{splitLeaf}\ t\ w_a\ a\ w_b\ b) = \mathit{cost}\ t + w_a + w_b$

Finally, the *minima* predicate expresses that two symbols a, b have the lowest frequencies in the tree t and that $\mathit{freq}\ t\ a \leq \mathit{freq}\ t\ b$:

$$
\begin{array}{l}
\mathit{minima}\ t\ a\ b \\
= (a \in \mathit{alphabet}\ t \wedge b \in \mathit{alphabet}\ t \wedge a \neq b \wedge \\
\quad (\forall\, c \in \mathit{alphabet}\ t. \\
\qquad c \neq a \longrightarrow c \neq b \longrightarrow \mathit{freq}\ t\ a \leq \mathit{freq}\ t\ c \wedge \mathit{freq}\ t\ b \leq \mathit{freq}\ t\ c))
\end{array}
$$

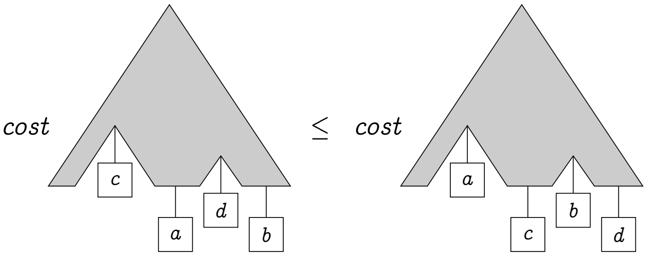
13.6 The Key Lemmas and Theorems

It is easy to prove that the tree returned by Huffman's algorithm preserves the alphabet, consistency, and symbol frequencies of the original forest:

$$
\begin{array}{l}
ts \neq [\,] \longrightarrow \mathit{alphabet}\ (\mathit{huffman}\ ts) = \mathit{alphabet}_F\ ts \\
\mathit{consistent}_F\ ts \wedge ts \neq [\,] \longrightarrow \mathit{consistent}\ (\mathit{huffman}\ ts) \\
ts \neq [\,] \longrightarrow \mathit{freq}\ (\mathit{huffman}\ ts)\ a = \mathit{freq}_F\ ts\ a
\end{array}
$$

The main difficulty is to prove the optimality of the tree constructed by Huffman's algorithm. We need three lemmas before we can present the optimality theorem.

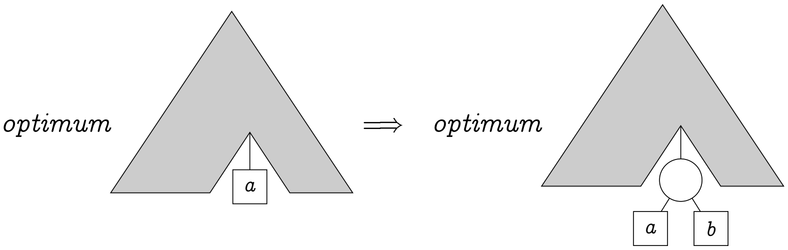
First, if a and b are minima, and c and d are at the very bottom of the tree, then exchanging a and b with c and d does not increase the tree's cost. Graphically, we have



Lemma 13.4. *consistent* $t \wedge$ *minima* $t\ a\ b \wedge c \in$ *alphabet* $t \wedge d \in$ *alphabet* $t \wedge$ *depth* $t\ c =$ *height* $t \wedge$ *depth* $t\ d =$ *height* $t \wedge c \neq d \longrightarrow$ *cost* (*swapFourSyms* $t\ a\ b\ c\ d) \leq$ *cost* t

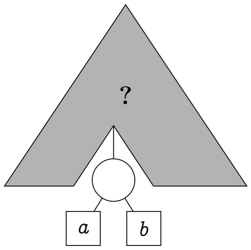
Proof. The proof is by case distinctions on $a = c$, $a = d$, $b = c$, and $b = d$. The cases are easy to prove by expanding the definition of *swapFourSyms* and applying Lemma 13.1. □

The tree *splitLeaf* $t\ w_a\ a\ w_b\ b$ is optimum if t is optimum, under a few assumptions, notably that a and b are minima of the new tree and that *freq* $t\ a = w_a + w_b$. Graphically:



Lemma 13.5. *consistent* $t \wedge$ *optimum* $t \wedge a \in$ *alphabet* $t \wedge b \notin$ *alphabet* $t \wedge$ *freq* $t\ a = w_a + w_b \wedge (\forall\, c \in$ *alphabet* $t.\ w_a \leq$ *freq* $t\ c \wedge w_b \leq$ *freq* $t\ c)$ $\longrightarrow$ *optimum* (*splitLeaf* $t\ w_a\ a\ w_b\ b$)

Proof. We assume that t's cost is less than or equal to that of any other comparable tree v and show that *splitLeaf* $t\ w_a\ a\ w_b\ b$ has a cost less than or equal to that of any other comparable tree u. For the nontrivial case where *height* $t > 0$, it is easy to prove that there must be two symbols c and d occurring in sibling nodes at the very bottom of u. From u we construct the tree *swapFourSyms* $u\ a\ b\ c\ d$ in which the minima a and b are siblings:



The question mark reminds us that we know nothing specific about u's structure. Merging a and b gives a tree comparable with t, which we can use to instantiate v:

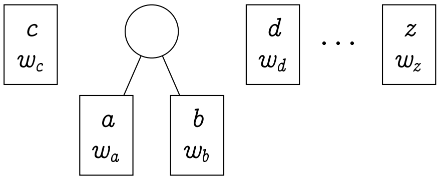
$$\begin{aligned}
& \mathit{cost}\ (\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b) = \mathit{cost}\ t + w_a + w_b && \text{by Lemma 13.3} \\
& \le \mathit{cost}\ (\mathit{mergeSibling}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d)\ a) + w_a + w_b && \\
& && \text{by optimality assumption} \\
& = \mathit{cost}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d) && \text{by Lemma 13.2} \\
& \le \mathit{cost}\ u && \text{by Lemma 13.4} \quad \square
\end{aligned}$$

An important property of Huffman's algorithm is that once it has combined two lowest-weight trees using *uniteTrees*, it does not visit these trees ever again. This suggests that splitting a leaf node before applying the algorithm should give the same result as applying the algorithm first and splitting the leaf node afterward.

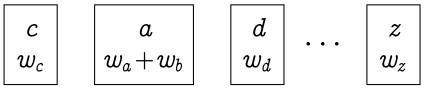
Lemma 13.6. $\mathit{consistent}_F\ \mathit{ts} \wedge \mathit{ts} \neq [] \wedge a \in \mathit{alphabet}_F\ \mathit{ts} \wedge \mathit{freq}_F\ \mathit{ts}\ a = w_a + w_b \longrightarrow \mathit{splitLeaf}\ (\mathit{huffman}\ \mathit{ts})\ w_a\ a\ w_b\ b = \mathit{huffman}\ (\mathit{splitLeaf}_F\ \mathit{ts}\ w_a\ a\ w_b\ b)$

The proof is by straightforward induction on the length of the forest *ts*.

As a consequence of this commutativity lemma, applying Huffman's algorithm on a forest of the form



gives the same result as applying the algorithm on the "flat" forest

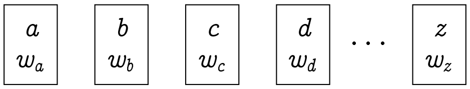


followed by splitting the leaf node a into two nodes a and b with frequencies w_a, w_b. The lemma effectively provides a way to flatten the forest at each step of the algorithm.

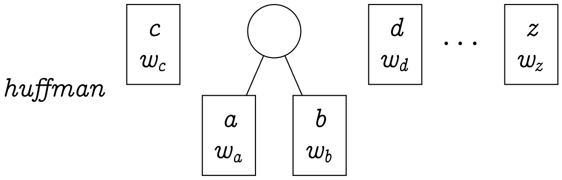
This leads us to our main result.

Theorem 13.7. $\mathit{consistent}_F\ \mathit{ts} \wedge \mathit{height}_F\ \mathit{ts} = 0 \wedge \mathit{sortedByWeight}\ \mathit{ts} \wedge \mathit{ts} \neq [] \longrightarrow \mathit{optimum}\ (\mathit{huffman}\ \mathit{ts})$

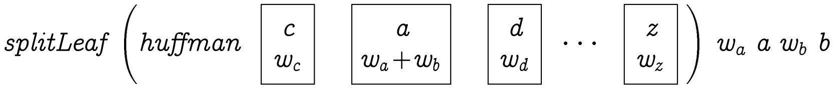
Proof. The proof is by induction on the length of *ts*. The assumptions ensure that *ts* is of the form



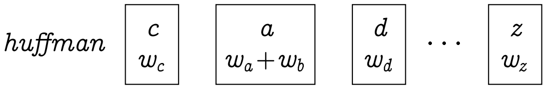
with $w_a \le w_b \le w_c \le w_d \le \cdots \le w_z$. If *ts* consists of a single node, the node has cost 0 and is therefore optimum. If *ts* has length 2 or more, the first step of the algorithm leaves us with a term such as



In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By Lemma 13.6, the above tree equals



To prove that this tree is optimum, it suffices by Lemma 13.5 to show that



is optimum, which follows from the induction hypothesis. □

In summary, we have established that the *huffman* program, which constitutes a functional implementation of Huffman's algorithm, constructs a binary tree that represents an optimal binary code for the specified alphabet and frequencies.

Part III

Priority Queues

14

Priority Queues

A **priority queue** of linearly ordered elements is like a multiset where one can insert arbitrary elements and remove minimal elements. Its specification as an ADT is show in Figure 14.1 where $\mathit{Min\_mset}\ m \equiv \mathit{Min}\ (\mathit{set\_mset}\ m)$ and Min yields the minimal element of a finite set of linearly ordered elements.

ADT $\mathit{Priority\_Queue} =$

interface
$\mathit{empty} :: {}'q$
$\mathit{insert} :: {}'a \Rightarrow {}'q \Rightarrow {}'q$
$\mathit{del\_min} :: {}'q \Rightarrow {}'q$
$\mathit{get\_min} :: {}'q \Rightarrow {}'a$

abstraction $\mathit{mset} :: {}'q \Rightarrow {}'a\ \mathit{multiset}$
invariant $\mathit{invar} :: {}'q \Rightarrow \mathit{bool}$

specification
$\mathit{invar}\ \mathit{empty}$ $(\mathit{empty\text{-}inv})$
$\mathit{mset}\ \mathit{empty} = \{\!\{\}\!\}$ (empty)
$\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ q)$ $(\mathit{insert\text{-}inv})$
$\mathit{invar}\ q \longrightarrow \mathit{mset}\ (\mathit{insert}\ x\ q) = \mathit{mset}\ q + \{\!\{x\}\!\}$ (insert)
$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\{\}\!\} \longrightarrow \mathit{invar}\ (\mathit{del\_min}\ q)$ $(\mathit{del\_min\text{-}inv})$
$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\{\}\!\} \longrightarrow$
$\mathit{mset}\ (\mathit{del\_min}\ q) = \mathit{mset}\ q - \{\!\{\mathit{get\_min}\ q\}\!\}$ $(\mathit{del\_min})$
$\mathit{invar}\ q \wedge \mathit{mset}\ q \neq \{\!\{\}\!\} \longrightarrow \mathit{get\_min}\ q = \mathit{Min\_mset}\ (\mathit{mset}\ q)$ $(\mathit{get\_min})$

Fig. 14.1. ADT *Priority_Queue*

Mergeable priority queues (see Figure 14.2) provide an additional function *merge* (sometimes: *meld* or *union*) with the obvious functionality.

ADT *Priority_Queue_Merge* = *Priority_Queue* +

interface
$\mathit{merge} :: {'q} \Rightarrow {'q} \Rightarrow {'q}$

specification
$\mathit{invar}\ q_1 \wedge \mathit{invar}\ q_2 \longrightarrow \mathit{invar}\ (\mathit{merge}\ q_1\ q_2)$
$\mathit{invar}\ q_1 \wedge \mathit{invar}\ q_2 \longrightarrow \mathit{mset}\ (\mathit{merge}\ q_1\ q_2) = \mathit{mset}\ q_1 + \mathit{mset}\ q_2$

Fig. 14.2. ADT *Priority_Queue_Merge*

Our priority queues are simplified. The more general version contains elements that are pairs of some item and its priority. In that case a priority queue can be viewed as a set, but because we have dropped the item we need to view a priority queue as a multiset. In imperative implementations, priority queues frequently also provide an operation *decrease_key*: given some direct reference to an element in the priority queue, decrease the element's priority. This is not completely straightforward in a functional language. Lammich and Nipkow [45] present an implementation, a Priority Search Tree.

Exercise 14.1. Give a list-based implementation of mergeable priority queues with constant-time *get_min* and *del_min*. Verify the correctness of your implementation w.r.t. *Priority_Queue_Merge*.

14.1 Heaps

A popular implementation technique for priority queues are **heaps**, i.e. trees where the minimal element in each subtree is at the root:

$$\mathit{heap} :: {'a}\ \mathit{tree} \Rightarrow \mathit{bool}$$

$$\begin{aligned}
&\mathit{heap}\ \langle\rangle = \mathit{True}\\
&\mathit{heap}\ \langle l,\ m,\ r\rangle\\
&= ((\forall x{\in}\mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r.\ m \le x) \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r)
\end{aligned}$$

Function *mset_tree* extracts the multiset of elements from a tree:

$$\mathit{mset\_tree} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{multiset}$$

$$\begin{aligned}
&\mathit{mset\_tree}\ \langle\rangle = \{\!\!\{\}\!\!\}\\
&\mathit{mset\_tree}\ \langle l,\ a,\ r\rangle = \{\!\!\{a\}\!\!\} + \mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r
\end{aligned}$$

When verifying a heap-based implementation of priority queues the invariant *invar* and the abstraction function *mset* in the ADT *Priority_Queue* are instantiated by *heap* and *mset_tree*. The correctness proofs need to talk about both multisets and (because of the *heap* invariant) sets of elements in a heap. We will only show the relevant multiset properties because the set properties follow easily via $\mathit{set\_mset}\ (\mathit{mset\_tree}\ t) = \mathit{set\_tree}\ t$.

Both *empty* and *get_min* have obvious implementations:

$$\mathit{empty} = \langle\rangle$$

$$\mathit{get\_min} = \mathit{value}$$

where $\mathit{value}\ \langle\_\,, a, \_\rangle = a$. If a heap-based implementation provides a *merge* function (e.g. skew heaps in Chapter 22), then *insert* and *del_min* can be defined like this:

$$\mathit{insert}\ x\ t = \mathit{merge}\ \langle\langle\rangle, x, \langle\rangle\rangle\ t$$

$$\begin{aligned}
&\mathit{del\_min}\ \langle\rangle = \langle\rangle\\
&\mathit{del\_min}\ \langle l, \_\,, r\rangle = \mathit{merge}\ l\ r
\end{aligned}$$

Note that the following tempting definition of *merge* is functionally correct but leads to very unbalanced heaps:

$$\begin{aligned}
&\mathit{merge}\ \langle\rangle\ t = t\\
&\mathit{merge}\ t\ \langle\rangle = t\\
&\mathit{merge}\ (\langle l_1, a_1, r_1\rangle =: t_1)\ (\langle l_2, a_2, r_2\rangle =: t_2)\\
&= (\textbf{if}\ a_1 \le a_2\ \textbf{then}\ \langle l_1, a_1, \mathit{merge}\ r_1\ t_2\rangle\ \textbf{else}\ \langle l_2, a_2, \mathit{merge}\ t_1\ r_2\rangle)
\end{aligned}$$

Many of the more advanced implementations of heaps focus on improving this merge function. We will see examples of this in the next chapter on leftist heaps, as well as in the chapters on skew heaps and pairing heaps.

Exercise 14.2. Show functional correctness of the above definition of *merge* (w.r.t. *Priority_Queue_Merge*) and prove functional correctness of the implementations of *insert* and *del_min* (w.r.t. *Priority_Queue*).

Bibliographic Remarks

The idea of the heap goes back to Williams [77] who also coined the name.

15

Leftist Heaps

Leftist heaps are heaps in the sense of Section 14.1 and implement mergeable priority queues. The key idea is to maintain the invariant that at each node the minimal height of the right child is $\leq$ that of the left child. We represent leftist heaps as augmented trees that store the minimal height in every node:

$$\textsf{\textbf{type\_synonym}}\ 'a\ \mathit{lheap} = ('a \times \mathit{nat})\ \mathit{tree}$$

$$\mathit{mht} :: 'a\ \mathit{lheap} \Rightarrow \mathit{nat}$$

$$\mathit{mht}\ \langle\rangle = 0$$
$$\mathit{mht}\ \langle\_,\ (\_,\ n),\ \_\rangle = n$$

There are two invariants: the standard *heap* invariant (on augmented trees)

$$\mathit{heap} :: ('a \times 'b)\ \mathit{tree} \Rightarrow \mathit{bool}$$
$$\mathit{heap}\ \langle\rangle = \mathit{True}$$
$$\mathit{heap}\ \langle l,\ (m,\ \_),\ r\rangle$$
$$= ((\forall x{\in}\mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r.\ m \leq x) \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r)$$

and the structural **leftist tree** invariant that requires that the minimal height of the right child is no bigger than that of the left child (and that the minimal height information in the node is correct):

$$\mathit{ltree} :: 'a\ \mathit{lheap} \Rightarrow \mathit{bool}$$
$$\mathit{ltree}\ \langle\rangle = \mathit{True}$$
$$\mathit{ltree}\ \langle l,\ (\_,\ n),\ r\rangle$$
$$= (\mathit{mh}\ r \leq \mathit{mh}\ l \wedge n = \mathit{mh}\ r + 1 \wedge \mathit{ltree}\ l \wedge \mathit{ltree}\ r)$$

Thus a tree is a leftist tree if for every subtree a shortest path from the root to a leaf is the right spine of the subtree. Remember $2^{mh\ t} \le |t|_1$. Thus the minimal height is upperbounded by the logarithm of the number of leaves. Because the expensive operations below descend along the right spine, this means that their running time is logarithmic in the number of leaves.

15.1 Implementation of ADT *Priority_Queue_Merge*

The key operation is *merge*:

```
merge :: 'a lheap ⇒ 'a lheap ⇒ 'a lheap
merge ⟨⟩ t = t
merge t ⟨⟩ = t
merge (⟨l₁, (a₁, n₁), r₁⟩ =: t₁) (⟨l₂, (a₂, n₂), r₂⟩ =: t₂)
= (if a₁ ≤ a₂ then node l₁ a₁ (merge r₁ t₂)
   else node l₂ a₂ (merge t₁ r₂))

node :: 'a lheap ⇒ 'a ⇒ 'a lheap ⇒ 'a lheap
node l a r
= (let rl = mht l; rr = mht r
   in if rr ≤ rl then ⟨l, (a, rr + 1), r⟩ else ⟨r, (a, rl + 1), l⟩)
```

Termination of *merge* can be proved either by the sum of the sizes of the two arguments (which goes down with every call) or by the lexicographic product of the two size measures: either the first argument becomes smaller or it stays unchanged and the second argument becomes smaller.

As shown in Section 14.1, once we have *merge*, the other operations are easily definable. We repeat their definitions simply because this chapter employs augmented rather than ordinary trees:

```
empty :: 'a lheap
empty = ⟨⟩

get_min :: 'a lheap ⇒ 'a
get_min ⟨_, (a, _), _⟩ = a

insert :: 'a ⇒ 'a lheap ⇒ 'a lheap
insert x t = merge ⟨⟨⟩, (x, 1), ⟨⟩⟩ t
```

$del\_min :: 'a\ lheap \Rightarrow 'a\ lheap$
$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle l,\ \_,\ r\rangle = merge\ l\ r$

15.2 Correctness

The above implementation is proved correct w.r.t. the ADT *Priority_Queue_Merge* where

$mset\_tree :: ('a \times 'b)\ tree \Rightarrow 'a\ multiset$
$mset\_tree\ \langle\rangle = \{\!\!\{\}\!\!\}$
$mset\_tree\ \langle l,\ (a,\ \_),\ r\rangle = \{\!\!\{a\}\!\!\} + mset\_tree\ l + mset\_tree\ r$

$invar\ t = (heap\ t \wedge ltree\ t)$

Correctness of *get_min* follows directly from the heap invariant:

$$heap\ t \wedge t \neq \langle\rangle \longrightarrow get\_min\ t = Min\ (set\_tree\ t)$$

From the following inductive lemmas about *merge*

$$mset\_tree\ (merge\ t_1\ t_2) = mset\_tree\ t_1 + mset\_tree\ t_2$$
$$ltree\ l \wedge ltree\ r \longrightarrow ltree\ (merge\ l\ r)$$
$$heap\ l \wedge heap\ r \longrightarrow heap\ (merge\ l\ r)$$

correctness of *insert* and *del_min* follow easily:

$$mset\_tree\ (insert\ x\ t) = mset\_tree\ t + \{\!\!\{x\}\!\!\}$$
$$mset\_tree\ (del\_min\ t) = mset\_tree\ t - \{\!\!\{get\_min\ t\}\!\!\}$$
$$ltree\ t \longrightarrow ltree\ (insert\ x\ t)$$
$$heap\ t \longrightarrow heap\ (insert\ x\ t)$$
$$ltree\ t \longrightarrow ltree\ (del\_min\ t)$$
$$heap\ t \longrightarrow heap\ (del\_min\ t)$$

Of course the above proof (ignoring the *ltree* part) works for any mergeable priority queue implemented as a heap.

15.3 Running Time Analysis

The running time functions are shown in Appendix B.4. By induction on the computation of merge we obtain

$$\mathit{ltree}\ l \wedge \mathit{ltree}\ r \longrightarrow T_{\mathit{merge}}\ l\ r \leq \mathit{mh}\ l + \mathit{mh}\ r + 1$$

With $2^{\mathit{mh}\ t} \leq |t|_1$ it follows that

$$\mathit{ltree}\ l \wedge \mathit{ltree}\ r \longrightarrow T_{\mathit{merge}}\ l\ r \leq \lg\ |l|_1 + \lg\ |r|_1 + 1 \tag{15.1}$$

which implies logarithmic bounds for insertion and deletion:

$$\mathit{ltree}\ t \longrightarrow T_{\mathit{insert}}\ x\ t \leq \lg\ |t|_1 + 3$$
$$\mathit{ltree}\ t \longrightarrow T_{\mathit{del\_min}}\ t \leq 2 \cdot \lg\ |t|_1 + 1$$

The derivation of the bound for insertion is trivial but the one for deletion uses a little lemma, assuming *ltree t*:

$$\begin{aligned} & T_{\mathit{del\_min}}\ t = T_{\mathit{merge}}\ l\ r + 1 \\ & \leq \lg\ |l|_1 + \lg\ |r|_1 + 2 && \text{using (15.1)} \\ & \leq 2 \cdot \lg\ |t|_1 + 1 && \text{because } \lg\ x + \lg\ y + 1 < 2 \cdot \lg\ (x + y) \\ & && \text{if } 0 < x \text{ and } 0 < y \end{aligned}$$

Bibliographic Remarks

Leftist heaps were invented by Crane [14]. Another version of leftist trees, based on weight rather than height, was introduced by Cho and Sahni [12, 68].

16 Priority Queues via Braun Trees

In Chapter 11 we introduced Braun trees and showed how to implement arrays. In the current chapter we show how to implement priority queues by means of Braun trees. Because Braun trees have logarithmic height this guarantees logarithmic running times for insertion and deletion. Remember that every node $\langle l, x, r\rangle$ in a Braun tree satisfies $|l| \in \{|r|, |r| + 1\}$ $(*)$.

16.1 Implementation of ADT *Priority_Queue*

We follow the heap approach in Section 14.1. Functions *empty*, *get_min*, *heap* and *mset_tree* are defined as in that section.

Insertion and deletion maintain the Braun tree property $(*)$ by inserting into the right (and possibly smaller) child, deleting from the left (and possibly larger) child, and swapping children to reestablish $(*)$.

Insertion is straightforward and clearly maintains both the heap and the Braun tree property:

```
insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert a ⟨⟩ = ⟨⟨⟩, a, ⟨⟩⟩
insert a ⟨l, x, r⟩
= (if a < x then ⟨insert x r, a, l⟩ else ⟨insert a r, x, l⟩)
```

To delete the minimal (i.e. root) element from a tree, extract the leftmost element from the tree and let it sift down to its correct position in the tree in the manner of heapsort:

```
del_min :: 'a tree ⇒ 'a tree
del_min ⟨⟩ = ⟨⟩
del_min ⟨⟨⟩, x, r⟩ = ⟨⟩
del_min ⟨l, x, r⟩ = (let (y, l') = del_left l in sift_down r y l')

del_left :: 'a tree ⇒ 'a × 'a tree
del_left ⟨⟨⟩, x, r⟩ = (x, r)
del_left ⟨l, x, r⟩ = (let (y, l') = del_left l in (y, ⟨r, x, l'⟩))

sift_down :: 'a tree ⇒ 'a ⇒ 'a tree ⇒ 'a tree
sift_down ⟨⟩ a _ = ⟨⟨⟩, a, ⟨⟩⟩
sift_down ⟨⟨⟩, x, _⟩ a ⟨⟩
= (if a ≤ x then ⟨⟨⟨⟩, x, ⟨⟩⟩, a, ⟨⟩⟩ else ⟨⟨⟨⟩, a, ⟨⟩⟩, x, ⟨⟩⟩)
sift_down (⟨l1, x1, r1⟩ =: t1) a (⟨l2, x2, r2⟩ =: t2)
= (if a ≤ x1 ∧ a ≤ x2 then ⟨t1, a, t2⟩
   else if x1 ≤ x2 then ⟨sift_down l1 a r1, x1, t2⟩
        else ⟨t1, x2, sift_down l2 a r2⟩)
```

In the first two equations for *sift_down*, the Braun tree property guarantees that the "_" arguments must be empty trees if the pattern matches.

16.2 Correctness

We outline the correctness proofs for *insert* and *del_min* by presenting the key lemmas. Correctness of *insert* is straightforward:

$$|\mathit{insert}\ x\ t| = |t| + 1$$
$$\mathit{mset\_tree}\ (\mathit{insert}\ x\ t) = \{\!\{x\}\!\} + \mathit{mset\_tree}\ t$$
$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{insert}\ x\ t)$$
$$\mathit{heap}\ t \longrightarrow \mathit{heap}\ (\mathit{insert}\ x\ t)$$

Correctness of *del_min* builds on analogous correctness lemmas for the auxiliary functions:

$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \longrightarrow \mathit{mset\_tree}\ t = \{\!\{x\}\!\} + \mathit{mset\_tree}\ t'$$
$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \land \mathit{heap}\ t \longrightarrow \mathit{heap}\ t'$$
$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \longrightarrow |t| = |t'| + 1$$
$$\mathit{del\_left}\ t = (x,\ t') \land t \neq \langle\rangle \land \mathit{braun}\ t \longrightarrow \mathit{braun}\ t'$$

$braun\ \langle l,\ a,\ r\rangle \longrightarrow |sift\_down\ l\ a\ r| = |l| + |r| + 1$

$braun\ \langle l,\ a,\ r\rangle \longrightarrow braun\ (sift\_down\ l\ a\ r)$

$braun\ \langle l,\ a,\ r\rangle \longrightarrow$
$mset\_tree\ (sift\_down\ l\ a\ r) = \{\!\{a\}\!\} + (mset\_tree\ l + mset\_tree\ r)$

$braun\ \langle l,\ a,\ r\rangle \wedge heap\ l \wedge heap\ r \longrightarrow heap\ (sift\_down\ l\ a\ r)$

$braun\ t \longrightarrow braun\ (del\_min\ t)$

$heap\ t \wedge braun\ t \longrightarrow heap\ (del\_min\ t)$

$braun\ t \longrightarrow |del\_min\ t| = |t| - 1$

$braun\ t \wedge t \neq \langle\rangle \longrightarrow$
$mset\_tree\ (del\_min\ t) = mset\_tree\ t - \{\!\{get\_min\ t\}\!\}$

Bibliographic Remarks

Our implementation of priority queues via Braun trees is due to Paulson [64] who credits it to Okasaki.

17

Binomial Heaps

Binomial heaps are another common implementation of mergeable priority queues, which supports efficient ($O(\log n)$) *insert*, *get_min*, *del_min*, and *merge* operations.

The basic building blocks of a binomial heap are *binomial trees*, which are defined recursively as follows. A binomial tree of rank r is a node with r children of ranks $r - 1$, ..., 0, in that order. Figure 17.1 shows an example binomial tree.

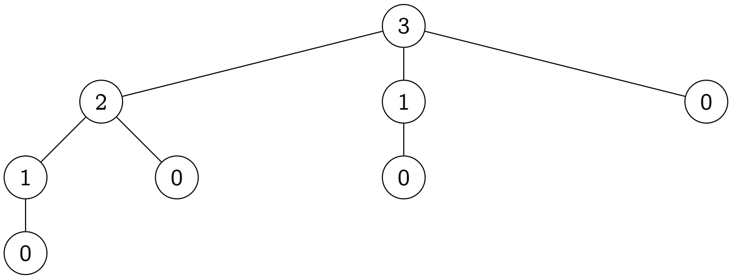


Fig. 17.1. A binomial tree of rank 3. The node labels depict the rank of each node. A node of rank r has child nodes of ranks $r - 1, \ldots, 0$.

To define binomial trees in Isabelle, we first define a more general datatype:

```
datatype 'a tree = Node nat 'a ('a tree list)
```

Apart from the list of children, a node stores a rank and an element. As usual, we use a shortcut notation for nodes:

```
⟨r, x, ts⟩ ≡ Node r x ts
```

This datatype contains all binomial trees, but also some non-binomial trees. To carve out the binomial trees, we define an invariant, which reflects the informal definition above:

```
invar_btree :: 'a tree ⇒ bool
invar_btree ⟨r, _, ts⟩
= ((∀t∈set ts. invar_btree t) ∧ map rank ts = rev [0..<r])
```

Additionally, we require the heap property, i.e., that the root element of each subtree is a minimal element in that subtree:

```
invar_otree :: 'a tree ⇒ bool
invar_otree ⟨_, x, ts⟩ = (∀t∈set ts. invar_otree t ∧ x ≤ root t)
```

Thus, a *binomial tree* is a tree that satisfies both the structural and the heap invariant. The two invariants are combined in a single predicate:

```
invar_tree :: 'a tree ⇒ bool
invar_tree t = (invar_btree t ∧ invar_otree t)
```

A *binomial heap* is a list of binomial trees with strictly ascending rank:

```
type_synonym 'a heap = 'a tree list
invar :: 'a heap ⇒ bool
invar ts
= ((∀t∈set ts. invar_tree t) ∧ sorted_wrt (<) (map rank ts))
```

Note that *sorted_wrt* states that a list is sorted wrt. the specified relation, here $(<)$. It is defined in Appendix A.

17.1 Size

The following functions return the multiset of elements in a binomial tree and in a binomial heap:

$$\mathit{mset\_tree} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{multiset}$$
$$\mathit{mset\_tree}\ \langle\_,\ a,\ \mathit{ts}\rangle = \{\!\{a\}\!\} + \textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset\_tree}\ (\mathit{mset}\ \mathit{ts}))$$
$$\mathit{mset\_heap} :: {'a}\ \mathit{heap} \Rightarrow {'a}\ \mathit{multiset}$$
$$\mathit{mset\_heap}\ \mathit{ts} = \textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset\_tree}\ (\mathit{mset}\ \mathit{ts}))$$

Most operations on binomial heaps are linear in the length of the heap. To show that the length is bounded by the number of heap elements, we first observe that the number of elements in a binomial tree is already determined by its rank. A binomial tree of rank r has 2^r nodes:

$$\mathit{invar\_btree}\ t \longrightarrow |\mathit{mset\_tree}\ t| = 2^{\mathit{rank}\ t}$$

This property is proved by induction on the tree structure. A tree of rank 0 has one element, and a tree of rank $r{+}1$ has subtrees of rank $0, 1, \ldots, r$. By the induction hypothesis, these have $2^0, 2^1, \ldots, 2^r$ elements, i.e., $2^{r+1} - 1$ elements together. Including the element at the root, there are 2^{r+1} elements.

The length of a binomial heap is bounded logarithmically in the number of its elements:

$$\mathit{invar}\ \mathit{ts} \longrightarrow |\mathit{ts}| \le \lg\ (|\mathit{mset\_heap}\ \mathit{ts}| + 1)$$

To prove this, recall that the heap *ts* is strictly sorted by rank. Thus, we can underestimate the ranks of the trees in ts by $0, 1, \ldots, |ts| - 1$. This means that they must have at least $2^0, 2^1, \ldots, 2^{|ts|-1}$ elements, i.e., at least $2^{|ts|} - 1$ elements together, which yields the desired bound.

17.2 Implementation of ADT *Priority_Queue*

17.2.1 Empty and Emptiness Test

Obviously, the empty list is a binomial heap with no elements, and a binomial heap is empty only if it is the empty list:

$$\mathit{invar}\ []$$
$$\mathit{mset\_heap}\ [] = \{\!\{\}\!\}$$
$$(\mathit{mset\_heap}\ \mathit{ts} = \{\!\{\}\!\}) = (\mathit{ts} = [])$$

17.2.2 Insertion

A crucial property of binomial trees is that we can link two binomial trees of rank r to form a binomial tree of rank $r + 1$, simply by prepending one tree as the first child of the other. To preserve the heap property, we add the tree with the bigger root element below the tree with the smaller root element. This *linking* of trees is illustrated in Figure 17.2. Formally:

$$\mathit{link} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}$$
$$\mathit{link}\ (\langle r,\ x_1,\ \mathit{ts}_1\rangle =: t_1)\ (\langle r',\ x_2,\ \mathit{ts}_2\rangle =: t_2)$$
$$= (\textbf{if}\ x_1 \le x_2\ \textbf{then}\ \langle r + 1,\ x_1,\ t_2\ \#\ \mathit{ts}_1\rangle\ \textbf{else}\ \langle r + 1,\ x_2,\ t_1\ \#\ \mathit{ts}_2\rangle)$$

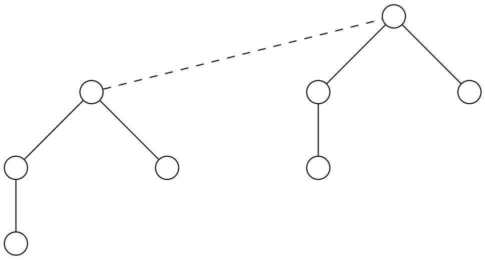


Fig. 17.2. Linking two binomial trees of rank 2 to form a binomial tree of rank 3, by linking the left tree as first child of the right tree, as indicated by the dashed line. We assume that the root element of the left tree is greater than or equal to the root element of the right tree, such that the heap property is preserved.

By case distinction, we can easily prove that *link* preserves the invariant and that the resulting tree contains the elements of both arguments.

$$\mathit{invar\_tree}\ t_1 \wedge \mathit{invar\_tree}\ t_2 \wedge \mathit{rank}\ t_1 = \mathit{rank}\ t_2 \longrightarrow$$
$$\mathit{invar\_tree}\ (\mathit{link}\ t_1\ t_2)$$
$$\mathit{mset\_tree}\ (\mathit{link}\ t_1\ t_2) = \mathit{mset\_tree}\ t_1 + \mathit{mset\_tree}\ t_2$$

The link operation forms the basis of inserting a tree into a heap: if the heap does not contain a tree with the same rank, we can simply insert the tree at the correct position in the heap. Otherwise, we merge the two trees and recursively insert the result. For our purposes, we can additionally assume that the rank of the tree to be inserted is smaller than or equal to the lowest rank in the heap, which saves us a case in the following definition:

$$\mathit{ins\_tree} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{heap} \Rightarrow {}'a\ \mathit{heap}$$
$$\mathit{ins\_tree}\ t\ [] = [t]$$
$$\mathit{ins\_tree}\ t_1\ (t_2\ \#\ \mathit{ts})$$
$$= (\textbf{if}\ \mathit{rank}\ t_1 < \mathit{rank}\ t_2\ \textbf{then}\ t_1\ \#\ t_2\ \#\ \mathit{ts}$$

```
  else ins_tree (link $t_1$ $t_2$) ts)
```

By straightforward induction, and using the respective properties for *link*, we can show that *ins_tree* preserves the invariant and yields a heap with the expected elements:

$$\mathit{invar\_tree}\ t \wedge \mathit{invar}\ \mathit{ts} \wedge (\forall t'\!\in\!\mathit{set}\ \mathit{ts}.\ \mathit{rank}\ t \le \mathit{rank}\ t') \longrightarrow \mathit{invar}\ (\mathit{ins\_tree}\ t\ \mathit{ts})$$

$$\mathit{mset\_heap}\ (\mathit{ins\_tree}\ t\ \mathit{ts}) = \mathit{mset\_tree}\ t + \mathit{mset\_heap}\ \mathit{ts}$$

A single element is inserted as a one-element (rank 0) tree:

```
insert :: 'a ⇒ 'a heap ⇒ 'a heap
insert x ts = ins_tree ⟨0, x, []⟩ ts
```

The above definition meets the specification for insert required by the *Priority_Queue* ADT:

$$\mathit{invar}\ t \longrightarrow \mathit{invar}\ (\mathit{insert}\ x\ t)$$

$$\mathit{mset\_heap}\ (\mathit{insert}\ x\ t) = \{\!\{x\}\!\} + \mathit{mset\_heap}\ t$$

17.2.3 Merging

Recall the merge algorithm used in top-down merge sort (Section 2.4). It merges two sorted lists by repeatedly taking the smaller list head. We use a similar idea for merging two heaps: if the rank of one list's head is strictly smaller, we choose it. If both ranks are equal, we link the two heads and insert the resulting tree into the merged remaining heaps. Thus, the resulting heap will be strictly ordered by rank. Formally:

```
merge :: 'a heap ⇒ 'a heap ⇒ 'a heap
merge $ts_1$ [] = $ts_1$
merge [] (v # va) = v # va
merge ($t_1$ # $ts_1$ =: $h_1$) ($t_2$ # $ts_2$ =: $h_2$)
= (if rank $t_1$ < rank $t_2$ then $t_1$ # merge $ts_1$ $h_2$
   else if rank $t_2$ < rank $t_1$ then $t_2$ # merge $h_1$ $ts_2$
        else ins_tree (link $t_1$ $t_2$) (merge $ts_1$ $ts_2$))
```

The *merge* function can be regarded as an algorithm for adding two sparse binary numbers. This intuition is explored in Exercise 17.2.

We show that the merge operation preserves the invariant and unites the elements:

$$\mathit{invar}\ \mathit{ts}_1 \wedge \mathit{invar}\ \mathit{ts}_2 \longrightarrow \mathit{invar}\ (\mathit{merge}\ \mathit{ts}_1\ \mathit{ts}_2)$$

$$\mathit{mset\_heap}\ (\mathit{merge}\ \mathit{ts}_1\ \mathit{ts}_2) = \mathit{mset\_heap}\ \mathit{ts}_1 + \mathit{mset\_heap}\ \mathit{ts}_2$$

The proof is straightforward, except for preservation of the binomial heap invariant. We first show that merging two heaps does not decrease the lowest rank in these heaps. This ensures that prepending the head with smaller rank to the merged remaining heaps results in a sorted heap. Moreover, when we link two heads of equal rank, this ensures that the linked tree's rank is smaller than or equal to the ranks in the merged remaining trees, as required by the *ins_tree* function. We phrase this property as preservation of lower rank bounds, i.e., a lower rank bound of both heaps is still a lower bound for the merged heap:

$$\begin{array}{l} t' \in \mathit{set}\ (\mathit{merge}\ \mathit{ts}_1\ \mathit{ts}_2) \wedge (\forall\, t_1{\in}\mathit{set}\ \mathit{ts}_1.\ \mathit{rank}\ t < \mathit{rank}\ t_1) \wedge \\ (\forall\, t_2{\in}\mathit{set}\ \mathit{ts}_2.\ \mathit{rank}\ t < \mathit{rank}\ t_2) \longrightarrow \\ \mathit{rank}\ t < \mathit{rank}\ t' \end{array}$$

The proof is by straightforward induction, relying on an analogous bounding lemma for *ins_tree*.

17.2.4 Finding a Minimal Element

For a binomial tree, the root node always contains a minimal element. Unfortunately, there is no such property for the whole heap—the minimal element may be at the root of any of the heap's trees. To get a minimal element from a non-empty heap, we look at all root nodes:

$$\mathit{get\_min} :: {}'a\ \mathit{heap} \Rightarrow {}'a$$

$$\begin{array}{l} \mathit{get\_min}\ [t] = \mathit{root}\ t \\ \mathit{get\_min}\ (t \ \#\ v\ \#\ \mathit{va}) = \mathit{min}\ (\mathit{root}\ t)\ (\mathit{get\_min}\ (v\ \#\ \mathit{va})) \end{array}$$

Correctness of this operation is proved by a simple induction:

$$\begin{array}{l} \mathit{mset\_heap}\ \mathit{ts} \neq \{\!\!\}\ \wedge \mathit{invar}\ \mathit{ts} \longrightarrow \\ \mathit{get\_min}\ \mathit{ts} = \mathit{Min\_mset}\ (\mathit{mset\_heap}\ \mathit{ts}) \end{array}$$

17.2.5 Deleting a Minimal Element

To delete a minimal element, we first need to find one and then remove it. Removing the root node of a tree with rank r leaves us with a list of its children, which are binomial trees of ranks $r - 1, \ldots, 0$. Reversing this list yields a valid binomial heap, which we merge with the remaining trees in the original heap:

```
del_min :: 'a heap ⇒ 'a heap
del_min ts
= (case get_min_rest ts of (⟨_, _, ts₁⟩, ts₂) ⇒ merge (rev ts₁) ts₂)
```

Here, the auxiliary function *get_min_rest* splits a heap into a tree with minimal root element, and the remaining trees.

```
get_min_rest :: 'a heap ⇒ 'a tree × 'a heap
get_min_rest [t] = (t, [])
get_min_rest (t # v # va)
= (let (t', ts') = get_min_rest (v # va)
   in if root t ≤ root t' then (t, v # va) else (t', t # ts'))
```

We prove that, for a non-empty heap, *del_min* preserves the invariant and deletes the minimal element:

$$ts \neq [] \wedge invar\ ts \longrightarrow invar\ (del\_min\ ts)$$

$$ts \neq [] \longrightarrow mset\_heap\ ts = mset\_heap\ (del\_min\ ts) + \{\!\{get\_min\ ts\}\!\}$$

The proof is straightforward. For invariant preservation, the key is to show that *get_min_rest* returns a binomial tree and a binomial heap:

$$\begin{array}{l} get\_min\_rest\ ts = (t',\ ts') \wedge ts \neq [] \wedge invar\ ts \longrightarrow invar\_tree\ t' \\ get\_min\_rest\ ts = (t',\ ts') \wedge ts \neq [] \wedge invar\ ts \longrightarrow invar\ ts' \end{array}$$

To show that we actually remove a minimal element, we show that *get_min_rest* selects the same tree as *get_min*:

$$ts \neq [] \wedge get\_min\_rest\ ts = (t',\ ts') \longrightarrow root\ t' = get\_min\ ts$$

17.3 Running Time Analysis

The running time functions are shown in Appendix B.5. Intuitively, the operations are linear in the length of the heap, which in turn is logarithmic in the number of elements (cf. Section 17.1).

The running time analysis for *insert* is straightforward. The running time is dominated by *ins_tree*. In the worst case, it iterates over the whole heap, taking constant time per iteration. By straightforward induction, we show

$$T_{ins\_tree}\ t\ ts \le |ts| + 1$$

and thus

$$invar\ ts \longrightarrow T_{insert}\ x\ ts \le \lg\ (|mset\_heap\ ts| + 1) + 2$$

The running time analysis for merge is more interesting. In each recursion, we need constant time to compare the ranks. However, if the ranks are equal, we link the trees and insert them into the merger of the remaining heaps. In the worst case, this costs linear time in the length of the merger. A naive analysis would estimate $|merge\ ts_1\ \ ts_2| \le |ts_1| + |ts_2|$, and thus yield a quadratic running time in the length of the heap.

However, we can do better: we observe that every link operation in *ins_tree* reduces the number of trees in the heap. Thus, over the whole merge, we can only have linearly many link operations in the combined size of both heaps.

To formalize this idea, we estimate the running time of *ins_tree* and *merge* together with the length of the result:

$$T_{ins\_tree}\ t\ ts + |ins\_tree\ t\ ts| = 2 + |ts|$$

$$|merge\ ts_1\ \ ts_2| + T_{merge}\ ts_1\ \ ts_2 \le 2 \cdot (|ts_1| + |ts_2|) + 1$$

Both estimates can be proved by straightforward induction, and from the second estimate we easily derive a bound for *merge*:

$$invar\ ts_1 \wedge invar\ ts_2 \longrightarrow$$
$$T_{merge}\ ts_1\ \ ts_2 \le 4 \cdot \lg\ (|mset\_heap\ ts_1| + |mset\_heap\ ts_2| + 1) + 1$$

From the bound for *merge*, we can easily derive a bound for *del_min*:

$$invar\ ts \wedge ts \ne [] \longrightarrow T_{del\_min}\ ts \le 6 \cdot \lg\ (|mset\_heap\ ts| + 1) + 3$$

The only notable point is that we use a linear time bound for reversing a list, as explained in Section 1.5.1:

$$T_{rev} :: 'a\ list \Rightarrow nat$$

$$T_{rev}\ xs = |xs| + 1$$

17.4 Exercises

Exercise 17.1. A node in a tree is on level n if it needs n edges from the root to reach it. Define a function $nol :: nat \Rightarrow 'a\ tree \Rightarrow nat$ such that $nol\ n\ t$ is the number of nodes on level n in tree t and show that a binomial tree of rank r has $\binom{r}{l}$ nodes on level l:

$$invar\_btree\ t \longrightarrow nol\ l\ t = rank\ t\ choose\ l$$

Hint: You might want to prove separately that

$$\sum_{i=0}^{i<r} \binom{i}{n} = \binom{r}{n+1}$$

Exercise 17.2. Sparse binary numbers represent a binary number by a list of the positions of set bits, sorted in ascending order. Thus, the list [1, 3, 4] represents the number 11010. In general, $[p_1, \ldots, p_n]$ represents $2^{p_1} + \cdots + 2^{p_n}$.

Implement sparse binary numbers in Isabelle, using the type *nat list*.

1. Define a function *invar* :: *nat list* ⇒ *bool* that checks for strictly ascending bit positions, a function *num_of* :: *nat list* ⇒ *nat* that converts a sparse binary number to a natural number, and a function *add* :: *nat list* ⇒ *nat list* ⇒ *nat list* to add sparse binary numbers.
2. Show that your *add* function preserves the invariant and actually performs addition as far as *num_of* is concerned.
3. Define a running time function for *add* and show that it is linear in the list size.

Hint: The bit positions in sparse binary numbers are analogous to binomial trees of a certain rank in a binomial heap. The *add* function is implemented similar to the *merge* function, using a *carry* function to insert a bit position into a number (similar to *ins_tree*). Correctness and running time can be proved similarly.

Bibliographic Remarks

Binomial queues were invented by Vuillemin [75]. Functional implementations were given by King [40] and Okasaki [62]. A functional implementation was verified by Meis *et al.* [49]; a Java implementation, by Müller [51].

Part IV

Advanced Design and Analysis Techniques

18

Dynamic Programming

You probably have seen this function before:

```
fib :: nat ⇒ nat
fib 0 = 0
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

It computes the well-known Fibonacci numbers. You may also have noticed that calculating *fib* 50 already causes quite some stress for your computer and there is no hope for *fib* 500 to ever return a result.

This is quite unfortunate considering that there is a very simple imperative program to compute these numbers efficiently:

```
int fib(n) {
  int a = 0;
  int b = 1;
  for (i in 1..n) {
    int temp = b;
    b = a + b;
    a = temp;
  }
  return a;
}
```

So we seem to be caught in a desperate situation here: either we use a clear and elegeant definition of *fib* or we get an efficient but convoluted implementation for *fib*. Admittedly, we could just prove that both formulations are the same function, and use whichever one is more suited for the task at hand. However, doing this for all recursive functions we would like to define is a tedious task. Instead, this chapter will sketch a recipe that allows to define such recursive functions in the natural way, while still getting an efficient implementation "for free".

18.1 Memoization

Let us consider the tree of recursive calls that are issued when computing *fib* 5 (c.f. Fig. 18.1). We can see that the subtree for *fib* 3 is recomputed two times, and that the subtree for *fib* 2 is even recomputed three times. How can we avoid these unnecessary computations? A common solution is *memoization*: we store previous computation results in some kind of memory and consult it to potentially recall a memoized result before issuing another recursive computation. Here is a memoizing version of *fib* that implements the memory as a mapping of type *nat* ⇒ *nat option*:

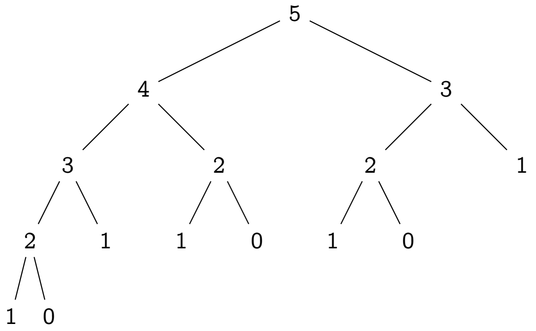


Fig. 18.1. Tree of the recursive call structure for *fib* 5

$$\begin{array}{l}
\mathit{fib}_1 :: \mathit{nat} \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat\ option}) \Rightarrow \mathit{nat} \times (\mathit{nat} \Rightarrow \mathit{nat\ option}) \\
\mathit{fib}_1\ 0\ m = (0,\ m(0 \mapsto 0)) \\
\mathit{fib}_1\ 1\ m = (1,\ m(1 \mapsto 1)) \\
\mathit{fib}_1\ (n + 2)\ m \\
= (\textbf{let}\ (a,\ m) = \textbf{case}\ m\ n\ \textbf{of}\ \mathit{None} \Rightarrow \mathit{fib}_1\ n\ m \mid \mathit{Some}\ a \Rightarrow (a,\ m); \\
\qquad\quad (b,\ m) = \\
\qquad\qquad \textbf{case}\ m\ (n + 1)\ \textbf{of}\ \mathit{None} \Rightarrow \mathit{fib}_1\ (n + 1)\ m
\end{array}$$

```
          | Some a ⇒ (a, m)
   in (a + b, m(n + 2 ↦ a + b)))
```

And indeed, we can ask Isabelle to compute fib_1 50 or even fib_1 500 and we get the result within a split second.

However, we are not yet happy with this code. Carrying the memory around means a lot of additional weight for the definition of fib_1, and proving that this function computes the same value as *fib* is not completely trivial (how would you approach this?). Let us streamline the definition first by pulling out the reading and writing of memory into a function *memo*:

```
memo ::
  'k ⇒ (('k ⇒ 'v option) ⇒ 'v × ('k ⇒ 'v option))
        ⇒ ('k ⇒ 'v option) ⇒ 'v × ('k ⇒ 'v option)
memo n f m ≡
case m n of None ⇒ let (a, m) = f m in (a, m(n ↦ a))
| Some a ⇒ (a, m)

fib_2 :: nat ⇒ (nat ⇒ nat option) ⇒ nat × (nat ⇒ nat option)
fib_2 0 = memo 0 (Pair 0)
fib_2 1 = memo 1 (Pair 1)
fib_2 (n + 2)
= memo (n + 2)
    (λm. let (a, m) = fib_2 n m; (b, y) = fib_2 (n + 1) m in (a + b, y))
```

This already looks a lot more like the original definition but it still has one problem: we have to thread the memory through the program explicitly. This can be become rather tedious for more complicated programs and diverges from the original shape of the program, rendering proofs harder.

18.1.1 Enter the Monad

Let us examine the type of fib_2 more closely. We can read it as the type of a function that, given a natural number, returns a *computation*. Given an initial memory, it computes a pair of a result and an updated memory. We can capture this notion of "stateful" computations in a data type:

```
datatype ('s, 'a) state = State ('s ⇒ 'a × 's)
```

A value of $('s,\ 'a)$ *state* represents a stateful computation that returns a result of type $'a$ and operates on states of type $'s$. The constant *run_state* forces the evaluation of a computation starting from some initial state: $\mathit{run\_state}\ (\mathit{State}\ a)\ s = a\ s$. The advantage of this definition may not seem immediate. It only starts to shine when we see how it allows us to *chain* stateful computations. To do so, we only need to define two constants: *return* to pack up a result in a computation, and *bind* to chain two computations after each other.

```
return :: 'a ⇒ ('s, 'a) state
return x ≡ State (λs. (x, s))

bind :: ('s, 'a) state ⇒ ('a ⇒ ('s, 'b) state) ⇒ ('s, 'b) state
bind a f ≡ State (λs. let (x, s) = run_state a s in run_state (f x) s)
```

We add a little syntax on top and write $\langle\!\langle x\rangle\!\rangle$ for $\mathit{return}\ x$, and $a \gg\!= f$ instead of $\mathit{bind}\ a\ f$. The “identity” computation $\langle\!\langle x\rangle\!\rangle$ simply leaves the given state unchanged and produces x as a result. The chained computation $a \gg\!= f$ starts with some state s, runs a on it to produce a pair of a result x and new state s', and then evaluates $f\ x$ to produce another computation that is run on s'.

We have now seen how to pass state around but we are not yet able to interact with it. For this purpose we define *get* and *set* to retrieve and update the current state, respectively:

```
get :: ('s, 's) state
get = State (λs. (s, s))

set :: 's ⇒ ('s, unit) state
set s' = State (λ_. ((), s'))
```

Let us reformulate fib_2 with the help of these concepts:

```
memo₁ ::
  'a ⇒ ('a ⇒ 'b option, 'b) state
        ⇒ ('a ⇒ 'b option, 'b) state
memo₁ n f ≡
get ≫=
(λm. case m n of None ⇒ f ≫= (λa. set (m(n ↦ a)) ≫= (λ_. ⟪a⟫))
```

$$\begin{array}{l}
\quad\quad \mid \mathit{Some}\ x \Rightarrow \langle\!\langle x \rangle\!\rangle) \\
\\
\mathit{fib}_3 :: \mathit{nat} \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat}\ \mathit{option},\ \mathit{nat})\ \mathit{state} \\
\mathit{fib}_3\ 0 = \langle\!\langle 0 \rangle\!\rangle \\
\mathit{fib}_3\ 1 = \langle\!\langle 1 \rangle\!\rangle \\
\mathit{fib}_3\ (n + 2) \\
= \mathit{memo}_1\ (n + 2)\ (\mathit{fib}_3\ n \gg\!= (\lambda a.\ \mathit{fib}_3\ (n + 1) \gg\!= (\lambda b.\ \langle\!\langle a + b \rangle\!\rangle)))
\end{array}$$

Can you see how we have managed to hide the whole handling of state behind the scenes? The only explicit interaction with the state is now happening inside of memo_1. This is sensible as this is the only place where we really want to recall a memoized result or to write a new value to memory.

While this is great, we still want to apply one more batch of polish: the syntactic structure of the last case of fib_3 still does not match *fib* exactly. To this end, we lift function application $f\ x$ to the state monad:

$$\begin{array}{l}
(.) :: ('s,\ 'a \Rightarrow ('s,\ 'b)\ \mathit{state})\ \mathit{state} \Rightarrow ('s,\ 'a)\ \mathit{state} \Rightarrow ('s,\ 'b)\ \mathit{state} \\
f_T\ .\ x_T \equiv f_m \gg\!= (\lambda f.\ x_m \gg\!= (\lambda x.\ f\ x))
\end{array}$$

We can now spell out our final memoizing version of *fib*:

$$\begin{array}{l}
\mathit{fib}_4 :: \mathit{nat} \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat}\ \mathit{option},\ \mathit{nat})\ \mathit{state} \\
\mathit{fib}_4\ 0 = \langle\!\langle 0 \rangle\!\rangle \\
\mathit{fib}_4\ 1 = \langle\!\langle 1 \rangle\!\rangle \\
\mathit{fib}_4\ (n + 2) \\
= \mathit{memo}_1\ (n + 2)\ (\langle\!\langle \lambda a.\ \langle\!\langle \lambda b.\ \langle\!\langle a + b \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\mathit{fib}_4\ n)\ .\ (\mathit{fib}_4\ (n + 1)))
\end{array}$$

You may wonder why we added that many additional computations in this last step. On the one hand, we have gained the advantage that we can now strictly follow the syntactic structure of *fib* to prove that fib_4 is correct. On the other hand, we can remove most of these additional computations in a final post-processing step.

18.1.2 Memoization and Dynamic Programming

Let us recap what we have seen above in this chapter. We noticed that the naive recursive formulation of the Fibonacci numbers leads to an highly inefficient implementation. We then showed how to work around this problem by using memoization to obtain a structurally similar but provably equivalent efficient implementation. After all this, you may wonder why this chapter is titled *Dynamic Programming* and not *Memoization*.

Dynamic programming relies on two main principles. First, to find an optimal solution for a problem by computing it from optimal solutions for "smaller" instances of the same problem, i.e., *recursion*. Second, to *memoize* these solutions for smaller problems in, e.g., a table. Thus we could be bold and state:

$$\text{dynamic programming} = \text{recursion} + \text{memoization}$$

For the remainder of this chapter, we will first outline how to prove that fib_4 is correct. Then, we will sketch how to apply our approach of memoization beyond *fib*. Afterwards, we will study some prototypical examples of dynamic programming problems and show how to apply above formula to them.

18.2 Correctness of Memoization

We now want to prove that fib_4 is correct. But what is it exactly that we want to prove? We surely want fib_4 to produce the same result as *fib* when run with an empty memory (where $\mathit{empty} = (\lambda\_.\ \mathit{None})$):

$$\mathit{fst}\ (\mathit{run\_state}\ (\mathit{fib}_4\ n)\ \mathit{empty}) = \mathit{fib}\ n$$

If we just wanted to hammer away at this prove, we would probably start with an induction on the computation of *fib* just to realize that the induction hypotheses are not strong enough to prove the recursion case, since they demand an empty memory. We can counter by applying generalization:

$$\mathit{fst}\ (\mathit{run\_state}\ (\mathit{fib}_4\ n)\ m) = \mathit{fib}\ n$$

However, this statement does not hold anymore for every memory m.

What do we need to demand from m? It should only memoize values that are *consistent* with *fib*:

$$\mathit{cmem}\ m \equiv \forall n \in \mathit{dom}\ m.\ m\ n = \mathit{Some}\ (\mathit{fib}\ n)$$

Using this, we can formulate a general notion of equivalence between a value v and its memoized version a, written $v \rhd a$: starting from a consistent memory m, a should produce another consistent memory m', and the result v.

$$\begin{array}{l}
(\rhd) :: {}'a \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat\ option},\ {}'a)\ \mathit{state} \Rightarrow \mathit{bool} \\
v \rhd s \equiv \\
\forall m.\ \mathit{cmem}\ m \longrightarrow \\
\quad\quad (\textbf{let}\ (v',\ m') = \mathit{run\_state}\ s\ m\ \textbf{in}\ v = v' \wedge \mathit{cmem}\ m')
\end{array}$$

Thus we want to prove

$$\textit{fib}\ n \triangleright \textit{fib}_4\ n$$

via computation induction on n as above. For the base cases we need to prove statements of the form $v \triangleright \langle v \rangle$, a property which follows trivially after unfolding the involved definitions. For the induction case, we can unfold $\textit{fib}_4$ $(n + 2)$, and get rid of $\textit{memo}_1$ by applying the following rule:

$$\textit{fib}\ n \triangleright a \longrightarrow \textit{fib}\ n \triangleright \textit{memo}_1\ n\ a$$

For the remainder of the proof, we now want to unfold $\textit{fib}$ $(n + 2)$ and then follow the syntactic structure of $\textit{fib}_4$ and $\textit{fib}$ in lockstep. To do so, we need to find a proof rule for function application. That is, what do we need in order to prove $f\ x \triangleright f_m\ .\ x_m$? For starters, $x \triangleright x_m$ seems reasonable to demand. But what about f and f_m? If f has type $'a \Rightarrow\ 'b$, then f_m is of type $('a \Rightarrow\ 'b\ \textit{mem})\ \textit{mem}$[1]. Intuitively, we want to state something along these lines:

> "f_m is a *memoized* function that when applied to a value x, yields a memoized value that is equivalent to $f\ x$".

This goes beyond what we can currently express with $(\triangleright)$ as $v \triangleright a$ merely states that "a is a memoized value equivalent to v". What we need is more liberty in our choice of equivalence. That is, we want to use statements $v \triangleright_R a$, with the meaning: "a is a memoized value that is related to v by R". The formal definition is analogous to $(\triangleright)$ (and $(\triangleright) = (\triangleright_{(=)})$):

$$
\begin{array}{l}
(\cdot\triangleright.\cdot) :: {}'a \Rightarrow ({}'a \Rightarrow {}'b \Rightarrow \textit{bool}) \Rightarrow {}'b\ \textit{mem} \Rightarrow \textit{bool} \\
v \triangleright_R s \equiv \\
\forall m.\ \textit{cmem}\ m \longrightarrow \\
\qquad (\textbf{let}\ (v', m') = \textit{run\_state}\ s\ m\ \textbf{in}\ R\ v\ v' \wedge \textit{cmem}\ m')
\end{array}
$$

However, we still do not have a means of expressing the second part of our sentence. To this end, we use the *function relator* $(\Rrightarrow)$:

$$
\begin{array}{l}
(\Rrightarrow) :: \\
\quad ({}'a \Rightarrow {}'c \Rightarrow \textit{bool}) \\
\quad \Rightarrow ({}'b \Rightarrow {}'d \Rightarrow \textit{bool}) \Rightarrow ({}'a \Rightarrow {}'b) \Rightarrow ({}'c \Rightarrow {}'d) \Rightarrow \textit{bool} \\
R \Rrightarrow S = (\lambda f\ g.\ \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y))
\end{array}
$$

[1] We use $'a\ \textit{mem}$ to abbreviate $(\textit{nat} \Rightarrow \textit{nat option},\ 'a)\ \textit{state}$.

Spelled out, we have $(R \Rrightarrow S)\ f\ g$ if for any values x and y that are related by R, the values $f\ x$ and $g\ y$ are related by S.

We can finally state a proof rule for application:

$$x \triangleright x_m \wedge f \triangleright_{(=) \Rrightarrow (\triangleright)} f_m \longrightarrow f\ x \triangleright f_m\ .\ x_m$$

In our concrete example, we apply it once to the goal

$$\mathit{fib}\ (n + 1) + \mathit{fib}\ n \triangleright \langle\!\langle \lambda a.\ \langle\!\langle \lambda b.\ \langle\!\langle a + b \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\mathit{fib}_4\ (n + 1))\ .\ (\mathit{fib}_4\ n),$$

solve the first premise with the induction hypotheses, and arrive at

$$(+)\ \mathit{fib}\ (\mathit{Suc}\ n) \triangleright_{(=) \Rrightarrow \triangleright_{(=)} \Rrightarrow (\triangleright)} \langle \lambda a.\ \langle \lambda b.\ \langle a + b \rangle \rangle \rangle\ .\ \mathit{fib}_4\ (\mathit{Suc}\ n).$$

Our current rule for application does not match this goal. Thus we need to generalize it. In addition, we need a new rule for *return*, and a rule for $(\Rrightarrow)$. To summarize, we need the following set of theorems about our consistency relation, applying them wherever they match syntactically to finish the proof:

$$R\ x\ y \longrightarrow x \triangleright_R \langle\!\langle y \rangle\!\rangle$$

$$x \triangleright_R x_m \wedge f \triangleright_{R \Rrightarrow \triangleright_S} f_m \longrightarrow f\ x \triangleright_S f_m\ .\ x_m$$

$$(\forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)) \longrightarrow (R \Rrightarrow S)\ f\ g$$

The theorem we aimed for initially

$$\mathit{fst}\ (\mathit{run\_state}\ (\mathit{fib}_4\ n)\ \mathit{empty}) = \mathit{fib}\ n$$

is now a trivial corollary of $\mathit{fib}\ n \triangleright \mathit{fib}_4\ n$. Note that by reading the equation from right to left, we have an easy way to make the memoization transparent to an end-user of *fib*.

18.3 Details of Memoization [*]

In this section, we will look at some further details of the memoization process and sketch how it can be applied beyond *fib*. First note that our approach of memoization hinges on two rather independent components: We transform the original program to use the state monad, to thread (an a priori arbitrary) state through the program. Only at the call sites of recursion, we then introduce the memoization functionality by issuing lookups and updates to the memory (as implemented by *memo*). We will name this first process *monadification*. For the second component, many different memory implementations can be

[*] If you are just interested in the dynamic programming algorithms of the following sections, this section can safely be skipped on the first read.

used, as long as we can define *memo* and prove the corresponding theorem. However, for now we want to turn our attention towards monadification.

To discuss some of the intricacies of monadification, let us first stick with *fib* for a bit longer and consider the following alternative definition:

$$\mathit{fib}\ n = (\textbf{if}\ n = 0\ \textbf{then}\ 0\ \textbf{else}\ 1 + \mathit{sum\_list}\ (\mathit{map}\ \mathit{fib}\ [0..{<}n - 1]))$$

We have not yet seen how to handle two ingredients of this program: constructs like *if-then-else* or case-combinators; and higher-order functions such as *map*.

It is quite clear how *if-then-else* can be lifted to the state monad:

$$\mathit{if}_m\ b_m\ x_m\ y_m \equiv b_m \ggg\!\!= (\lambda b.\ \textbf{if}\ b\ \textbf{then}\ x_m\ \textbf{else}\ y_m)$$

By following the structure of the terms, we can also deduce a proof rule for if_m:

$$\begin{array}{l} b \triangleright b_m \wedge x \triangleright_R x_m \wedge y \triangleright_R y_m \longrightarrow \\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) \triangleright_R \mathit{if}_m\ b_m\ x_m\ y_m \end{array}$$

However, suppose we want to apply this proof rule to our new equation for *fib*. We will certainly need the knowledge of whether $n = 0$ to make progress in the correctness proof. Thus we make our rule more precise:

$$\begin{array}{l} b \triangleright b_m \wedge (b \longrightarrow x \triangleright_R x_m) \wedge (\neg\ b \longrightarrow y \triangleright_R y_m) \longrightarrow \\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) \triangleright_R \mathit{if}_m\ b_m\ x_m\ y_m \end{array}$$

How can we lift *map* to the state monad level? Consider its defining equations:

$$\begin{array}{l} \mathit{map}\ f\ [] = [] \\ \mathit{map}\ f\ (x \mathbin{\#} \mathit{xs}) = f\ x \mathbin{\#} \mathit{map}\ f\ \mathit{xs} \end{array}$$

We can follow the pattern we used to monadify *fib* to monadify *map*:

$$\begin{array}{l} \mathit{map}_m{}'\ f\ [] = \langle\!\langle [] \rangle\!\rangle \\ \mathit{map}_m{}'\ f\ (x \mathbin{\#} \mathit{xs}) = \langle\!\langle \lambda a.\ \langle\!\langle \lambda b.\ \langle\!\langle a \mathbin{\#} b \rangle\!\rangle \rangle\!\rangle \rangle\!\rangle\ .\ (\langle\!\langle f \rangle\!\rangle\ .\ \langle\!\langle x \rangle\!\rangle)\ .\ (\mathit{map}_m{}'\ f\ \mathit{xs}) \end{array}$$

We have obtained a function $\mathit{map}_m{}'$ of type

$$('a \Rightarrow ('s,\ 'b)\ \mathit{state}) \Rightarrow\ 'a\ \mathit{list} \Rightarrow ('s,\ 'b\ \mathit{list})\ \mathit{state}$$

This is not yet compatible with our scheme of lifting function application to (.). We need a function of type

$$('s,\ ('a \Rightarrow ('s,\ 'b)\ \mathit{state}) \Rightarrow ('s,\ 'a\ \mathit{list} \Rightarrow ('s,\ 'b\ \mathit{list})\ \mathit{state})\ \mathit{state})\ \mathit{state}$$

because *map* has two arguments and we need one layer of the state monad for each of its arguments. Therefore we simply define

$$\mathit{map}_m \equiv \langle\!\langle \lambda f.\ \langle\!\langle \mathit{map}_m{}'\ f \rangle\!\rangle \rangle\!\rangle$$

To get through an induction proof for the new definition of *fib*, we also need the knowledge that *fib* is recursively applied only to smaller values than n when computing *fib* n. That is, we need to know which values f is applied to in *map* f *xs*. We can encode this knowledge in a proof rule for *map*:

$$\frac{\mathit{xs} = \mathit{ys} \qquad \bigwedge x.\ x \in \mathit{set}\ \mathit{ys} \longrightarrow f\ x \rhd_R f_m\ x}{\mathit{map}\ f\ \mathit{xs} \rhd_{\mathit{list\_all2}\ R}\ \mathit{map}_m\ .\ \langle\!\langle f_m \rangle\!\rangle\ .\ \langle\!\langle \mathit{ys} \rangle\!\rangle}$$

The relator *list_all2* lifts R to a pairwise relation on lists:

$$\mathit{list\_all2}\ R\ \mathit{xs}\ \mathit{ys} = (|\mathit{xs}| = |\mathit{ys}| \wedge (\forall i{<}|\mathit{xs}|.\ R\ (\mathit{xs}\ !\ i)\ (\mathit{ys}\ !\ i)))$$

To summarize, here is a fully memoized version of the alternative definition of *fib*:

$$\begin{array}{l}
\mathit{fib}_m{}'\ n \\
= \mathit{memo}_1\ n \\
\quad (\mathit{if}_m\ \langle\!\langle n = 0 \rangle\!\rangle\ \langle\!\langle 0 \rangle\!\rangle \\
\quad\quad (\langle\!\langle \lambda a.\ \langle\!\langle 1 + a \rangle\!\rangle \rangle\!\rangle\ . \\
\quad\quad\quad (\langle\!\langle \lambda a.\ \langle\!\langle \mathit{sum\_list}\ a \rangle\!\rangle \rangle\!\rangle\ .\ (\mathit{map}_m\ .\ \langle\!\langle \mathit{fib}_m{}' \rangle\!\rangle\ .\ \langle\!\langle [0..{<}n - 1] \rangle\!\rangle))))
\end{array}$$

The correctness proof for fib_m is analogous to the one for fib_4, once we have proved the new rules discussed above.

At the end of this section, we note that the techniques that were sketched above also extend to case-combinators and other higher-order functions. Most of the machinery for monadification and the corresponding correctness proofs can be automated in Isabelle. The reader shall be referred to [79] for more details. Finally note that none of the techniques we used so far are specific to *fib*. The only parts that have to be adopted are the definitions of *memo* and *cmem*. In Isabelle, this can be done by simply instantiating a locale.

18.4 The Bellman-Ford Algorithm

Calculating shortest paths in weighted graphs is a classic algorithmic task that we all encounter in everyday situations, such as planning the fastest route to drive from A to B. In this scenario we can view streets as edges in a graph and nodes as street crossings. Every edge is associated with a weight, e.g. the time to traverse a street. We are interested in the path from A to B with minimum weight, corresponding to the fastest route in the example. Note that in this example it is safe to assume that all edge weights are non-negative.

Some applications demand negative edge weights as well. Suppose, we transport ourselves a few years into the future, where we have an electric car that can recharge itself via solar cells while driving. If we aim for the most energy-efficient route from A to B, a very sunny route could then incur a negative edge weight.

The Bellman-Ford algorithm is a classic dynamic programming solution to the *single-destination shortest path problem* in graphs with negative edge weights. That is, we are given a directed graph with negative edge weights and some target vertex (known as *sink*), and we want to calculate the weight of the shortest (i.e. minimum weight) paths from every vertex to the sink. Figure 18.2 shows an example of such a graph.

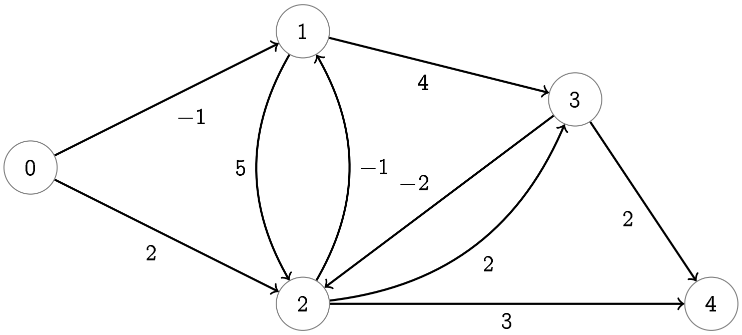


Fig. 18.2. Example of a weighted directed graph

Formally, we will take a simple view of graphs. We assume that we are given a number of nodes numbered $0, \ldots, n$, and some sink $t \in \{0..n\}$ (thus $n = t = 4$ in the example). Edge weights are given by a function $W :: int \Rightarrow int \Rightarrow int\ extended$. The type *int extended* extends the natural numbers with positive and negative infinity:

datatype *'a extended* = *Fin 'a* | ∞ | $-\infty$

We refrain from giving the explicit definition of addition and comparison on this domain, and rely on your intuition instead. A weight assignment $W\ i\ j = \infty$ means that there is no edge from i to j.

18.4.1 Deriving a Recursion

The main idea of the algorithm is to consider paths in order of increasing length *in the number of edges*. In the example, we can immediately read off the weights of the shortest paths to the sink that use only one edge: only nodes 2 and 3 are directly connected to the sink, with edge weights 3 and 2, respectively; for all others the weight is infinite. How can we now calculate the minimum weight paths (to the sink) with at most two edges? For node 3, the weight of the shortest path with at most two edges is: either the weight of the path with one edge; or the weight of the edge from node 3 to node 2 plus the weight of the path with one edge from node 2 to the sink. Because $-2 + 3 = 1 \leq 2$, we get a new minimum weight of 1 for node 3. Following the same scheme, we can iteratively calculate the minimum path weights given in table 18.1.

i/v	0	1	2	3	4
0	∞	∞	∞	∞	0
1	∞	∞	3	2	0
2	5	6	3	1	0
3	5	5	3	1	0
4	4	5	3	1	0

Table 18.1. The minimum weights of paths from vertices $v = 0 \ldots 4$ to t that use at most $i = 0 \ldots 4$ edges.

The analysis we just ran on the example already gives us a clear intuition on all we need to deduce a dynamic program: a recursion on sub-problems, in this case to compute the weight of shortest paths with at most $i + 1$ edges from the weights of shortest paths with at most i edges. To formalize this recursion, we first define the notion of a minimum weight path from some node v to t with at most i edges, denoted as $OPT\ i\ v$:

$$\begin{aligned}
&OPT\ i\ v\\
&= Min\ (\{weight\ (v\ \#\ xs\ @\ [t])\ |\ |xs| + 1 \leq i \wedge set\ xs \subseteq \{0..n\}\} \cup\\
&\qquad\quad \{\textbf{if}\ t = v\ \textbf{then}\ 0\ \textbf{else}\ \infty\})
\end{aligned}$$

```
weight :: (nat ⇒ nat ⇒ int extended) ⇒ nat list ⇒ int extended
weight _ [_] = 0
weight W (i # j # xs) = W i j + weight W (j # xs)
```

If $i = 0$, things are simple:

$$OPT\ 0\ v = (\textbf{if}\ t = v\ \textbf{then}\ 0\ \textbf{else}\ \infty)$$

A shortest path that constitutes $OPT\ (i + 1)\ v$ uses either at most i or exactly $i + 1$ edges. That is, $OPT\ (i + 1)\ v$ is either $OPT\ i\ v$, or the weight of the edge from v to any of its neighbours w plus $OPT\ i\ w$:

$$\begin{aligned}&OPT\ (i + 1)\ v\\ &= min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w \mid w \le n\})\end{aligned}$$

Proof. We prove this equality by proving two inequalities:

($lhs \le rhs$) For this direction, we essentially need to show that every path on the rhs is covered by the lhs, which is trivial.

($lhs \ge rhs$) We skip the cases where $OPT\ (i + 1)\ v$ is trivially 0 or ∞. Thus consider some xs such that $OPT\ (i + 1)\ v = weight\ (v\ \#\ xs\ @\ [t])$, $|xs| \le i$, and $set\ xs \subseteq \{0..n\}$. The cases where $|xs| < i$ or $i = 0$ are trivial. Otherwise, we have $OPT\ (i + 1)\ v = W\ v\ (hd\ xs) + weight\ (xs\ @\ [t])$ by definition of *weight*, and $OPT\ i\ (hd\ xs) \le weight\ (xs\ @\ [t])$ by definition of OPT. Therefore, we can show:

$$OPT\ (i + 1)\ v \ge OPT\ i\ (hd\ xs) + W\ v\ (hd\ xs) \ge rhs$$

□

We can turn these equations into a recursive program:

```
bf :: nat ⇒ nat ⇒ int extended
bf 0 v = (if t = v then 0 else ∞)
bf (i + 1) v
= min_list (bf i v # map (λi. W v i + bf i i) [0..<n + 1])
```

It is obvious that we can prove correctness of *bf* by induction:

$$bf\ i\ v = OPT\ i\ v$$

18.4.2 Negative Cycles

Have we solved the initial problem now? The answer is "not quite" because we have ignored one additional complication. Consider our example table 18.1

again. The table stops at path length five because no shorter paths with more edges exist. For this example, five corresponds to the number of nodes, which bounds the length of the longest *simple* path. However, is it the case that we will never find shorter non-simple paths in other graphs? The answer is “no”. If a graph contains a *negative reaching* cycle, i.e. a cycle with a negative sum of edge weights from which the sink is reachable, then we can use it arbitrarily often too find shorter and shorter paths.

Luckily, we can use the Bellman-Ford algorithm to detect this situation by examining the relationship of $OPT\ n$ and $OPT\ (n + 1)$. The following proposition summarizes the key insight:

> The graph contains a negative reaching cycle if and only if there exists a $v \le n$ such that $OPT\ (n + 1)\ v < OPT\ n\ v$

Proof. If there is no negative reaching cycle, then all shortest paths are either simple or contain superfluous cycles of weight 0. Thus, we have $OPT\ (n + 1)\ v = OPT\ n\ v$ for all $v \le n$.

Otherwise, there is a negative reaching cycle $ys = a\ \#\ xs\ @\ [a]$ with $weight\ ys < 0$. Working towards a contradiction, assume that $OPT\ n\ v \le OPT\ (n + 1)\ v$ for all $v \le n$. Using the recursion we proved above, this implies $OPT\ n\ v \le W\ v\ u + OPT\ n\ u$ for all u, $v \le n$. By applying this inequality to the nodes in $a\ \#\ xs$, we can prove the inequality

$$\begin{aligned} &sum\_list\ (map\ (OPT\ n)\ ys) \\ &\le sum\_list\ (map\ (OPT\ n)\ ys) + weight\ ys \end{aligned}$$

This implies $0 \le weight\ ys$, which yields the contradiction. □

This means we can use *bf* to detect the existence of negative reaching cycles by computing one more round, i.e. $bf\ (n + 1)\ v$ for all v. If nothing changes in this step, we know that there are no negative reaching cycles and that $bf\ n$ correctly represents the shortest path weights. Otherwise, there has to be a negative reaching cycle.

Finally, we can use memoization to obtain an efficient implementation that solves the single-destination shortest path problem. Applying our memoization technique from above, we first obtain a memoizing version bf_m of *bf*. We then define the following program:

```
bellman_ford ≡
iter_bf (n, n) ≫=
(λ_. map_m' (bf_m n) [0..<n + 1] ≫=
      (λxs. map_m' (bf_m (n + 1)) [0..<n + 1] ≫=
            (λys. ⟪if xs = ys then Some xs else None⟫)))
```

Here, *iter_bf* $(n,\ n)$ just computes the values from $bf_m\ 0\ 0$ to $bf_m\ n\ n$ in a row-by-row manner. Using the reasoning principles that were described above, we can then prove that *bellman_ford* indeed solves its intended task correctly (*shortest* v is the length of the shortest path from v to t):

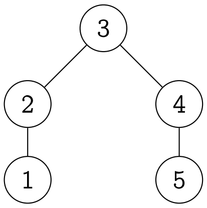
$$\begin{array}{l}(\forall\, i{\le}n.\ \forall\, j{\le}n.\ -\infty\ <\ W\ i\ j) \longrightarrow \\ \mathit{fst}\ (\mathit{run\_state}\ \mathit{bellman\_ford}\ \mathit{empty}) \\ = (\textbf{if}\ \mathit{has\_negative\_cycle}\ n\ W\ t\ \textbf{then}\ \mathit{None} \\ \quad \textbf{else}\ \mathit{Some}\ (\mathit{map}\ \mathit{shortest}\ [0..{<}n\ +\ 1]))\end{array}$$

18.5 Optimal Binary Search Trees

In this book, we have studied various tree data structures that guarantee logarithmic runtime bounds for operations such as lookups and updates into the tree. These bounds were usually worst-case and did not take into account any information about the actual series of queries that are to be issued to the data structure. In this section, instead, we want to focus on *binary search trees* that minimize the amount of work that needs to be done when the distribution of keys in a *sequence of lookup operations* is known in advance.

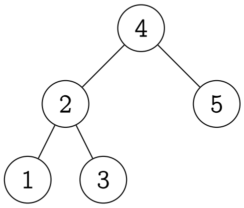
More formally, we want to study the following problem. We are given a range of integers $[i..j]$ and a function p :: *int* ⇒ *nat* that maps each key in the range to a percentage corresponding to the probability that this key is searched for [2]. Our goal is to find a binary search tree that minimizes the expected number of comparisons when presented with a sequence of lookup keys in the range $[i..j]$ that adhere to the distribution given by p.

As an example, consider the range [1..5] with probabilities [10, 30, 15, 25, 20]. This tree



incurs an expected value of 2.15 lookup operations. However, the minimal expected value is 2 and is achieved by this tree:

[2] In the rest of the discussion, p does not necessarily have to represent a probability distribution, i.e. the value of *sum* p $\{i..j\}$ does not have to be 100.



It is clear that our task is equivalent to minimizing the *weighted path length* (or *cost*) as we did for Huffman encodings above. Recall that the weighted path length is the sum of the frequencies of every node in the tree multiplied by its depth in the tree. It fulfills the following (recursive) equations:

$$\begin{aligned} &cost\ \langle\rangle = 0\\ &cost\ \langle l,\ k,\ r\rangle\\ &= sum\ p\ (set\_tree\ l) + cost\ l + p\ k + cost\ r + sum\ p\ (set\_tree\ r)\end{aligned}$$

The difference of our task compared to finding an optimal Huffman encoding is the constraint that the resulting tree needs to be *sorted*, making it hard to deploy a similar greedy solution. Instead, we want to come up with a dynamic programming solution and thus need to find a way to subdivide the problem.

18.5.1 Deriving a Recursion

The key insight into the problem is that subtrees of optimal binary search trees are also optimal. Consider the optimal tree from above again. The left and right subtrees of the root must be optimal, since if we could improve either one, we would also get a better tree for the complete range of keys. This motivates the following definition:

$$\begin{aligned} &wpl\ W\ i\ j\ \langle\rangle = 0\\ &wpl\ W\ i\ j\ \langle l,\ k,\ r\rangle\\ &= wpl\ W\ i\ (k - 1)\ l + wpl\ W\ (k + 1)\ j\ r + W\ i\ j\\ &W\ i\ j = sum\ p\ \{i..j\}\end{aligned}$$

It is easy to see that *wpl W i j* is just reformulation of *cost t*:

$$inorder\ t = [i..j] \longrightarrow wpl\ W\ i\ j\ t = cost\ t$$

We can actually forget about the original distribution *p* and just optimize *wpl W i j* for some fixed weight function $W :: int \Rightarrow int \Rightarrow nat$.

The binary search tree t that contains the keys $[i..j]$ and minimizes $wpl\ W\ i\ j\ t$ has some root k with $[i..j] = [i..k - 1]$ @ k # $[j + 1..k]$. Its left and right subtrees need to be optimal again, i.e. minimize $wpl\ W\ i\ (k - 1)$ and $wpl\ W\ (k + 1)\ j$. This yields the following recursive functions for computing the optimum weighted path length (*min_wpl*) and the binary search tree itself (*opt_bst*):

```
min_wpl :: int ⇒ int ⇒ nat
min_wpl i j
= (if j < i then 0
   else min_list
          (map (λk. min_wpl i (k − 1) + min_wpl (k + 1) j +
                    W i j)
            [i..j]))

opt_bst :: int × int ⇒ int tree
opt_bst (i, j)
= (if j < i then ⟨⟩
   else argmin (wpl W i j)
          (map (λk. ⟨opt_bst (i, k − 1), k, opt_bst (k + 1, j)⟩) [i..j]))
```

Here *argmin f xs* returns the rightmost $x \in set\ xs$ such that $f\ x$ is maximal among *xs*.

To prove that *min_wpl* and *opt_bst* are correct, we want to show two properties: the value of $min\_wpl\ i\ j$ should be a lower bound of $wpl\ W\ i\ j\ t$ for any search tree t for $[i..j]$, and $min\_wpl\ i\ j$ should correspond to the weight of an actual search tree, namely $opt\_bst\ (i, j)$. Formally, we prove the following properties:

$$inorder\ t = [i..j] \longrightarrow min\_wpl\ i\ j \le wpl\ W\ i\ j\ t$$

$$inorder\ (opt\_bst\ (i, j)) = [i..j]$$

$$wpl\ W\ i\ j\ (opt\_bst\ (i, j)) = min\_wpl\ i\ j$$

Proof. The first property is easily proved by computation induction on *wpl W*, and the second and third property are equally easily proved using computation induction on *min_wpl*. □

If we substitute the weights that are constructed from p (W), we can derive the following correctness theorems referring to the original problem:

$$inorder\ t = [i..j] \longrightarrow min\_wpl\ W\ i\ j \le cost\ t$$

$$cost\ (opt\_bst\ W\ (i, j)) = min\_wpl\ W\ i\ j$$

18.5.2 Memoization

We can apply the memoization techniques that were discussed above to efficiently compute *min_wpl* and *opt_bst*. The only remaining caveat is that W also needs to be computed efficiently from the distribution p. If we just use the defining equality $W\ i\ j = \mathit{sum}\ p\ \{i..j\}$, the computation of W is unnecessarily costly. Another way is to memoize W itself, using the following recursion:

$$W\ i\ j = (\textbf{if}\ i \le j\ \textbf{then}\ W\ i\ (j - 1) + p\ j\ \textbf{else}\ 0)$$

This yields a memoizing version $W_m{}'$ and a theorem that connects it to W:

$$W\ m\ x \triangleright W_m{}'\ p\ m\ x$$

We can now iterate $W_m{}'\ i\ n$ for $i = 0 \ldots n$ to pre-compute all relevant values of $W\ i\ j$:

$$W_c\ p\ n = \mathit{snd}\ (\mathit{run\_state}\ (\mathit{map}_m{}'\ (\lambda i.\ W_m{}'\ p\ i\ n)\ [0..n])\ \mathit{empty})$$

Using the correctness theorem for $\mathit{map}_m{}'$ from above, it can easily be shown that this yields a consistent memory:

$$\mathit{cmem}\ (\mathit{compute\_W}\ n)$$

Finally, we can show the following equation for computing W

$$W\ p\ i\ j = (\textbf{case}\ (W_c\ p\ n)\ (i,\ j)\ \textbf{of}\ \mathit{None} \Rightarrow W\ p\ i\ j \mid \mathit{Some}\ x \Rightarrow x)$$

and use it to pass the pre-computed values of W to *opt_bst*:

$$\begin{array}{l}
\mathit{opt\_bst}' :: (\mathit{int} \Rightarrow \mathit{nat}) \Rightarrow \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{int}\ \mathit{tree} \\
\mathit{opt\_bst}'\ p\ i\ j \equiv \\
\textbf{let}\ M = W_c\ p\ j; \\
\quad\ W = \lambda i\ j.\ \textbf{case}\ M\ (i,\ j)\ \textbf{of}\ \mathit{None} \Rightarrow W\ p\ i\ j \mid \mathit{Some}\ x \Rightarrow x \\
\textbf{in}\ \mathit{opt\_bst}\ W\ (i,\ j)
\end{array}$$

18.5.3 Optimizing the Recursion

While we have applied some trickery to obtain an efficient implementation of the simple dynamic programming algorithm expressed by *opt_bst*, we still have not arrived at the solution that is currently known to be most efficient. The most efficient known algorithm to compute optimal binary search trees due to Knuth [41] is a slight variation of *opt_bst* and relies on the following observation.

Let $R\ i\ j$ denote the maximal root of any optimal binary search for $[i..j]$:

$$\begin{aligned}&R\ i\ j\\ &= \mathit{argmin}\ (\lambda k.\ w\ i\ j + \mathit{min\_wpl}\ i\ (k - 1) + \mathit{min\_wpl}\ (k + 1)\ j)\ [i..j]\end{aligned}$$

It can be shown that $R\ i\ j$ is bounded by $R\ i\ (j - 1)$ and $R\ (i + 1)\ j$:

$$i < j \longrightarrow R\ i\ (j - 1) \le R\ i\ j \wedge R\ i\ j \le R\ (i + 1)\ j$$

The proof of this fact is rather involved and we refer the interested reader to the expositions of Yao [80] or Mehlhorn [48] for details. An Isabelle formalization by Nipkow and Somogyi is also available [59].

With this knowledge, we can make the following optimization to *opt_bst*:

```
opt_bst2 :: int ⇒ int ⇒ int tree
opt_bst2 i j
= (if j < i then ⟨⟩
   else if i = j then ⟨⟨⟩, i, ⟨⟩⟩
        else let left = root (opt_bst2 i (j - 1));
                 right = root (opt_bst2 (i + 1) j)
             in argmin (wpl i j)
                  (map (λk. ⟨opt_bst2 i (k - 1), k,
                              opt_bst2 (k + 1) j⟩)
                    [left..right]))
```

You may wonder whether this change really incurs an asymptotic runtime improvement. Indeed, it can be shown that it improves the algorithm's runtime by a factor of $O(n)$. For a fixed search tree size $d = i - j$, the total number of recursive computations is given by the following telescoping series:

$$\begin{aligned}&d \le n \longrightarrow\\ &(\textstyle\sum j = d..n.\ \mathbf{let}\ i = j - d\ \mathbf{in}\ R\ (i + 1)\ j - R\ i\ (j - 1) + 1)\\ &= R\ (n - d + 1)\ n - R\ 0\ (d - 1) + n - d + 1\end{aligned}$$

This quantity is bounded by $2 \cdot n$, which implies that the overall number of recursive computation calls is bounded by $O(n^2)$.

Bibliographic Remarks

The original $O(n^2)$ algorithm for Binary Search Trees is due to Knuth [41]. Yao later explained this optimization more elegantly in his framework of "quadrilateral inequalities" [80]. Nipkow and Somogyi follow Yao's approach in their Isabelle formalization [59], on which the last subsection of this chapter is based. The other parts of this chapter are based on a paper by Wimmer et al. [79] and its accompanying Isabelle formalization [78]. The formalization also contains further examples of dynamic programming algorithms, including solutions for the Knapsack and the minimum edit distance problems, and the CYK algorithm.

19

Amortized Analysis

Consider a k-bit binary counter and a sequence of increment (by one) operations on it where each one starts from the least significant bit and keeps flipping the 1s until a 0 is encountered (and flipped). Thus the worst-case running time of an increment is $O(k)$ and a sequence of n increments takes time $O(nk)$. However, this analysis is very coarse: in a sequence of increments there are many much faster ones (for half of them the least significant bit is 0!). It turns out that a sequence of n increments takes time $O(n)$. Thus the average running time of each increment is $O(1)$. Amortized analysis is the analysis of the running time of a sequence of operations on some data structure by upper-bounding the average running time of each operation.

As the example of the binary counter shows, the amortized running time for a single call of an operation can be much better than the worst-case time. Thus amortized analysis is unsuitable in a real-time context where worst-case bounds on every call of an operation are required.

Below we assume that there is a single data structure (in the example: the binary counter) that is an argument to each of the operations and may also be returned as a result.

Amortized analysis is valid only if the data structure is used in a single-threaded manner. The binary counter shows why: start from 0, increment the counter until all bits are 1, then increment that counter value again and again, without destroying it. Each of those increments takes time $O(k)$ and you can do that as often as you like, thus subverting the analysis. In an imperative language you can easily avoid this “abuse” by making the data structure stateful: every operation modifies the state of the data structure. This shows that amortized analysis has an imperative flavour. In a purely functional language, monads can be used to guarantee single-threadedness.

19.1 The Potential Method

The potential method is a particular technique for amortized analysis. The key idea is to define a potential function Φ from the data structure to non-negative numbers. The potential of the data structure is like a savings account that cheap calls can pay into (by increasing the potential) to compensate for later expensive calls (which decrease the potential). In a nutshell: the less "balanced" a data structure is, the higher its potential should be because it will be needed to pay for the impending restructuring.

The **amortized running time** is defined as the actual running time plus the difference in potential, i.e. the potential after the call minus the potential before it. If the potential increases, the amortized running time is higher than the actual running time and we pay the difference into our savings account. If the potential decreases, the amortized running time is lower than the actual running time and we take something out of our savings account to pay for the difference.

More formally, we are given some data structure with operations f, g, etc on it with corresponding time functions T_f, T_g etc. We are also given a potential function Φ. Then the amortized running time function A_f for f is defined as follows:

$$A_f\ s = T_f\ s + \Phi\ (f\ s) - \Phi\ s \tag{19.1}$$

where s is the data structure under consideration; f may also have additional parameters. Given a sequence of data structure states $s_0, \ldots, s_n$ where $s_{i+1} = f_i\ s_i$, it is not hard to see that

$$\sum_{i=0}^{n-1} A_{f_i}\ s_i = \sum_{i=0}^{n-1} T_{f_i}\ s_i + \Phi\ s_n - \Phi\ s_0$$

If we assume (for simplicity) that $\Phi\ s_0 = 0$, then it follows immediately that the amortized running time of the whole sequence is an upper bound of the actual running time (because Φ is non-negative). This observation becomes useful if we can bound $A_f\ s$ by some closed term $u_f\ s$. Typical examples for $u_f\ s$ are constants, logarithms etc. Then we can conclude that f has constant, logarithmic etc amortized complexity. Thus the only proof obligation is

$$A_f\ s \leq u_f\ s$$

possibly under the additional assumption *invar s* if the data structure comes with an invariant *invar*.

In the sequel we assume that s_0 is some fixed value, typically "empty", and that its potential is 0.

How do we analyze operations that combine two data structures, e.g. the union of two sets? Their amortized complexity can be defined in analogy to (19.1):

$$A_f\ s_1\ s_2 = T_f\ s_1\ s_2 + \Phi\ (f\ s_1\ s_2) - (\Phi\ s_1 + \Phi\ s_2)$$

So far we implicitly assumed that all operations return the data structure as a result, otherwise $\Phi\ (f\ s)$ does not make sense. How should we analyze so-called **observer functions** that do not modify the data structure but return a value of some other type? Amortized analysis does not make sense here because the same observer can be applied multiple times to the same data structure value without modifying it. Classical worst-case complexity is needed, unless the observer does modify the data structure as a side effect or by returning a new value. Then one can perform an amortized analysis that ignores the returned observer value (but not the time it takes to compute it).

19.2 Examples

19.2.1 Binary Counter

The binary counter is represented by a list of booleans where the head of the list is the least significant bit. The increment operation and its running time are easily defined:

$$\mathit{incr} :: \mathit{bool\ list} \Rightarrow \mathit{bool\ list}$$

$$\begin{aligned}
&\mathit{incr}\ [] = [\mathit{True}]\\
&\mathit{incr}\ (\mathit{False}\ \#\ \mathit{bs}) = \mathit{True}\ \#\ \mathit{bs}\\
&\mathit{incr}\ (\mathit{True}\ \#\ \mathit{bs}) = \mathit{False}\ \#\ \mathit{incr}\ \mathit{bs}
\end{aligned}$$

$$T_{\mathit{incr}} :: \mathit{bool\ list} \Rightarrow \mathit{real}$$

$$\begin{aligned}
&T_{\mathit{incr}}\ [] = 1\\
&T_{\mathit{incr}}\ (\mathit{False}\ \#\ \_) = 1\\
&T_{\mathit{incr}}\ (\mathit{True}\ \#\ \mathit{bs}) = T_{\mathit{incr}}\ \mathit{bs} + 1
\end{aligned}$$

The potential of a counter is the number of *True*'s because they increase T_{incr}:

$$\Phi :: \mathit{bool\ list} \Rightarrow \mathit{real}$$

$$\Phi\ \mathit{bs} = |\mathit{filter}\ (\lambda x.\ x)\ \mathit{bs}|$$

Clearly the potential is never negative.

The amortized complexity of *incr* is 2:

$$T_{\mathit{incr}}\ \mathit{bs} + \Phi\ (\mathit{incr}\ \mathit{bs}) - \Phi\ \mathit{bs} = 2$$

This can be proved automatically by induction on *bs*.

Bibliographic Remarks

Amortized analysis is due to Tarjan [72]. Introductions to it can be found in most algorithm textbooks. This chapter is based on earlier work by Nipkow [53, 55] which also formalizes the meta-theory of amortized analysis.

20

Queues

20.1 Queue Specification

A **queue** can be viewed as a glorified list with function *enq* for adding an element to the end of the list and function *first* for accessing and *deq* for removing the first element. This is the full ADT:

ADT $\mathit{Queue} =$

interface $\mathit{empty} :: {'q}$
$\mathit{enq} :: {'a} \Rightarrow {'q} \Rightarrow {'q}$
$\mathit{deq} :: {'q} \Rightarrow {'q}$
$\mathit{first} :: {'q} \Rightarrow {'a}$
$\mathit{is\_empty} :: {'q} \Rightarrow \mathit{bool}$

abstraction $\mathit{list} :: {'q} \Rightarrow {'a}\ \mathit{list}$

invariant $\mathit{invar} :: {'q} \Rightarrow \mathit{bool}$

specification $\mathit{list}\ \mathit{empty} = []$
$\mathit{invar}\ q \longrightarrow \mathit{list}\ (\mathit{enq}\ x\ q) = \mathit{list}\ q\ @\ [x]$
$\mathit{invar}\ q \longrightarrow \mathit{list}\ (\mathit{deq}\ q) = \mathit{tl}\ (\mathit{list}\ q)$
$\mathit{invar}\ q \wedge \mathit{list}\ q \neq [] \longrightarrow \mathit{first}\ q = \mathit{hd}\ (\mathit{list}\ q)$
$\mathit{invar}\ q \longrightarrow \mathit{is\_empty}\ q = (\mathit{list}\ q = [])$

$\mathit{invar}\ \mathit{empty}$
$\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{enq}\ x\ q)$
$\mathit{invar}\ q \longrightarrow \mathit{invar}\ (\mathit{deq}\ q)$

A trivial implementation is as a list, but then *enq* is linear in the length of the queue. To improve this we consider two more sophisticated implementations. First, a simple implementation where every operation has amortized constant complexity. Second, a tricky “real time” implementation where every operation has worst-case constant complexity.

20.2 Queues as Pairs of Lists

The queue is implemented as a pair of lists (*fs*, *rs*), the front and rear lists. Function *enq* adds elements to the head of the rear *rs* and *deq* removes elements from the head of the front *fs*. When *fs* becomes empty, it is replaced by *rev rs* (and *rs* is emptied) — the reversal ensures that now the oldest element is at the head. Hence *rs* is really the reversal of the rear of the queue but we just call it the rear. The abstraction function is obvious:

```
list :: 'a list × 'a list ⇒ 'a list
list (fs, rs) = fs @ rev rs
```

Clearly *enq* and *deq* are constant-time until the front becomes empty. Then we need to reverse the rear which takes linear time (if it is implemented by *itrev*, see Section 1.5.1). But we can pay for this linear cost up front by paying a constant amount for each call of *enq*. Thus we arrive at amortized constant time. See below for the formal treatment.

The implementation is shown in Figure 20.1. Of course *empty* = ([], []). Function *norm* performs the reversal of the rear once the front becomes empty. Why does not only *deq* but also *enq* call *norm*? Because otherwise *enq* x_n (...(*enq* x_1 *empty*)...) would result in ([], [x_n, ..., x_1]) and *first* would

```
norm :: 'a list × 'a list ⇒ 'a list × 'a list
norm (fs, rs) = (if fs = [] then (itrev rs [], []) else (fs, rs))

enq :: 'a ⇒ 'a list × 'a list ⇒ 'a list × 'a list
enq a (fs, rs) = norm (fs, a # rs)

deq :: 'a list × 'a list ⇒ 'a list × 'a list
deq (fs, rs) = (if fs = [] then (fs, rs) else norm (tl fs, rs))

first :: 'a list × 'a list ⇒ 'a
first (a # _, _) = a

is_empty :: 'a list × 'a list ⇒ bool
is_empty (fs, _) = (fs = [])
```

Fig. 20.1. Queue as a pair of lists

become an expensive operation because it would requires the reversal of the rear. Thus we need to avoid queues $([], rs)$ where $rs \neq []$. Thus *norm* guarantees the following invariant:

$$\begin{aligned}&\mathit{invar} :: \mathit{'a\ list} \times \mathit{'a\ list} \Rightarrow \mathit{bool}\\&\mathit{invar}\ (\mathit{fs},\ \mathit{rs}) = (\mathit{fs} = [] \longrightarrow \mathit{rs} = [])\end{aligned}$$

Functional correctness, i.e. proofs of the properties in the ADT *Queue*, are straightforward. Let us now turn to the amortized running time analysis. The time functions are shown in Appendix B.6.

For the amortized analysis we define the potential function

$$\begin{aligned}&\Phi :: \mathit{'a\ list} \times \mathit{'a\ list} \Rightarrow \mathit{nat}\\&\Phi\ (\_,\ \mathit{rs}) = |\mathit{rs}|\end{aligned}$$

because $|rs|$ is the amount we have accumulated by charging 1 for each *enq*. This is enough to pay for the eventual reversal. Now it is easy to prove that both *enq* and *deq* have amortized constant running time:

$$\begin{aligned}&T_{enq}\ a\ (\mathit{fs},\ \mathit{rs}) + \Phi\ (\mathit{enq}\ a\ (\mathit{fs},\ \mathit{rs})) - \Phi\ (\mathit{fs},\ \mathit{rs}) \le 4\\&T_{deq}\ (\mathit{fs},\ \mathit{rs}) + \Phi\ (\mathit{deq}\ (\mathit{fs},\ \mathit{rs})) - \Phi\ (\mathit{fs},\ \mathit{rs}) \le 3\end{aligned}$$

The two observer functions *first* and *is_empty* have constant running time.

20.3 A Real Time Implementation

This sections presents the **Hood-Melville queue**, a tricky implementation that improves upon the representation in the previous Section by preemptively performing reversals over a number of operations before they are required.

20.3.1 Stepped Reversal

Breaking down a reversal operation into multiple steps can be done using the following function:

$$\begin{aligned}&\mathit{rev\_step} :: \mathit{'a\ list} \times \mathit{'a\ list} \Rightarrow \mathit{'a\ list} \times \mathit{'a\ list}\\&\mathit{rev\_step}\ (x\ \#\ \mathit{xs},\ \mathit{ys}) = (\mathit{xs},\ x\ \#\ \mathit{ys})\\&\mathit{rev\_step}\ ([],\ \mathit{ys}) = ([],\ \mathit{ys})\end{aligned}$$

where x # xs is the list being reversed, and x # ys is the partial reversal result. Thus, to reverse a list of size 3 one should call *rev_step* 3 times:

$$\begin{array}{l} \mathit{rev\_step}\ ([1,\ 2,\ 3],\ []) = ([2,\ 3],\ [1]) \\ \mathit{rev\_step}\ (\mathit{rev\_step}\ ([1,\ 2,\ 3],\ [])) = ([3],\ [2,\ 1]) \\ \mathit{rev\_step}\ (\mathit{rev\_step}\ (\mathit{rev\_step}\ ([1,\ 2,\ 3],\ []))) = ([],\ [3,\ 2,\ 1]) \end{array}$$

Note that each call to *rev_step* takes constant time since its definition is non-recursive.

Using the notation f^n for the n-fold composition of function f we can state a simple inductive lemma:

Lemma 20.1. $\mathit{rev\_step}^{|xs|}\ (xs,\ ys) = ([],\ \mathit{rev}\ xs\ @\ ys)$

As a special case this implies $\mathit{rev\_step}^{|xs|}\ (xs,\ []) = ([],\ \mathit{rev}\ xs)$.

20.3.2 A Real Time Intuition

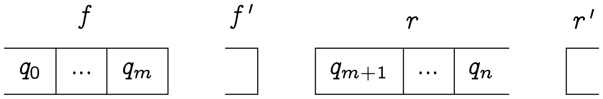
Hood-Melville queues are similar to those presented in Section 20.2 in that they use a pair of lists $(f,\ r)$ (front and rear — for succinctness we drop the s's now) to achieve constant running time *deq* and *enq*. However, they avoid a costly reversal operation once f becomes empty by preemptively computing a new front $\mathit{fr} = f\ @\ \mathit{rev}\ r$ one step at a time using *rev_step* as enqueueing and dequeueing operations occur. The process that generates fr consists of three phases:

1. Reverse r to form r', which is the tail end of fr
2. Reverse f to form f'
3. Reverse f' onto r' to form fr

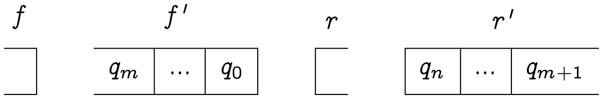
All three phases can be described in terms of *rev_step* as follows:

1. $r' = \mathit{snd}\ (\mathit{rev\_step}^{|r|}\ (r,\ []))$
2. $f' = \mathit{snd}\ (\mathit{rev\_step}^{|f|}\ (f,\ []))$
3. $\mathit{fr} = \mathit{snd}\ (\mathit{rev\_step}^{|f'|}\ (f',\ r'))$

Phases (1) and (2) are independent and can be performed at the same time, hence, when starting from this configuration



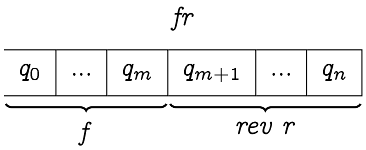
after $\mathit{max}\ |f|\ |r|$ steps of reversal the state would be the following:



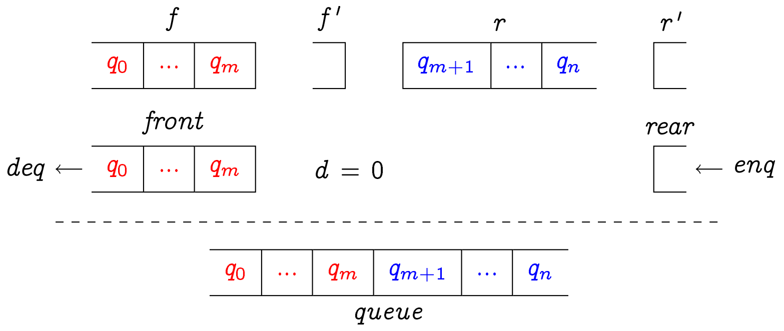
Phase (3) reverses f' onto r' fr to obtain the same result as a call to *list*:

$$\begin{aligned}
fr &= snd\ (rev\_step^{|f'|}\ (f',\ r')) && \text{by definition of } fr\\
&= rev\ f'\ @\ r' && \text{using Lemma 20.1}\\
&= rev\ f'\ @\ snd\ (rev\_step^{|r|}\ (r,\ [])) && \text{by definition of } r'\\
&= rev\ f'\ @\ rev\ r && \text{using Lemma 20.1}\\
&= rev\ (snd\ (rev\_step^{|f|}\ (f,\ []))) \ @\ rev\ r' && \text{by definition of } f'\\
&= rev\ (rev\ f)\ @\ rev\ r && \text{using Lemma 20.1}\\
&= f\ @\ rev\ r && \text{by } rev \text{ involution}
\end{aligned}$$

The resulting front list fr contains all elements previously in f and r:



A Hood-Melville queue spreads all reversal steps across queue altering operations requiring careful bookkeeping. To achieve this gradual reversal, additional lists *front* and *rear* are used for enqueuing and dequeuing, while internal operations rely only on f, f', r, and r'. At the start of the reversal process *rear* is copied into r and emptied; similarly, *front* is copied into f, but its contents are kept as they might need to be dequeued. Moreover, to avoid using elements from f or f' that may have been removed from *front*, a counter d records the number of dequeuing operations that have occurred since the reversal process started; this way, only $|f'| - d$ elements are appended into r to form fr. Once the reversal finishes fr become the new *front* and the internal lists are cleared. When the queue is not being reversed all operations are performed in a manner similar to previous implementations. The configuration of a queue at the beginning of the reversal process is as follows:



20.3.3 The Reversal Strategy

A crucial detail of this implementation is determining at which point the reversal process should occur. The strategy is to start once |*rear*| becomes larger than |*front*|, and ensure that all reversal steps are done before *front* runs out of elements or *rear* becomes larger than the new front (*fr*).

With this strategy, once |*rear*| = n+1 and |*front*| = n, the reversal processes starts. The first two phases take $n + 1$ steps (*max* |*front*| |*rear*|) to generate f' and r', and the third phase produces *fr* in n steps. A complete reversal takes $2 \cdot n + 1$ steps. Because the queue can only perform n *deq* operations before *front* is exhausted, $2 \cdot n + 1$ steps must be performed in at most n operations. This can be achieved by performing the first two steps in the operation that causes *rear* to become larger than *front* and two more steps in each subsequent operation. Therefore, $2 \cdot (n + 1)$ steps can occur before *front* is emptied, allowing the reversal process to finish in time.

Finally, since at most n *enq* or *deq* operations can occur during reversal, the largest possible *rear* has length n (only *enq* ops), while the smallest possible *fr* has length $n + 1$ (only *deq* ops). Thus, after the reversing process has finished the new front (*fr*) is always larger than *rear*.

20.3.4 Implementation

Queues are implemented using the following record type:

```
record 'a queue = lenf   :: nat
                  front  :: 'a list
                  status :: 'a status
                  rear   :: 'a list
                  lenr   :: nat
```

In a nutshell, a record is a product type with named fields and "built-in" construction, selection, and update operations. Values of *'a queue* are constructed using *make* :: *nat* ⇒ *'a list* ⇒ *'a status* ⇒ *'a list* ⇒ *nat* ⇒ *'a queue* were each argument corresponds to one of the fields of the record in canonical order. Additionally, given a queue *q* we can obtain the value in field *front* with *front q*, and update its content using *q*⦇*front* := []⦈. Multiple updates can be composed as *q*⦇*front* := [], *rear* := []⦈.

All values in the queue along with its internal state are stored in the various fields of *'a queue*. Fields *front* and *rear* contain the lists over which all queue operations are performed. The length of *front* and *rear* is recorded in *lenf* and *lenr* (respectively) to avoid calling *length* whose complexity is not constant. Finally, *status* tracks the current reversal phase of the queue in a *'a status* value.

```
datatype 'a status =
  Idle |
  Rev nat ('a list) ('a list) ('a list) ('a list) |
  App nat ('a list) ('a list) |
  Done
```

Each value of *'a status* represents either a phase of reversal or the queue's normal operation. Constructor *Idle* signals that no reversal is being performed. *Rev ok f f' r r'* corresponds to phases (1) and (2) where the lists *f*, *f'*, *r*, and *r'* are used for the reversal steps of the front and the rear. The *App ok f' r'* case corresponds to phase (3) where both lists are appended to form the new front (*fr*). In both *App* and *Rev*, the first argument *ok* (of type *nat*) keeps track of the number of elements in *f'* that have not been removed from the queue, effectively $ok = |f'| - d$, where *d* is the number of *deq* operations that have occurred so far. Lastly, *Done fr* marks the end of the reversal process and contains only the new front list *fr*.

In the implementation, all of the steps of reversal operations in the queue are performed by functions *exec* and *invalidate*; they ensure at each step that the front list being computed is kept consistent w.r.t. the contents and operations in the queue.

Function *exec* :: *'a status* ⇒ *'a status* performs the incremental reversal of the front list by altering the queue's *status* one step at a time in accordance with the reversal phases. Following the strategy described in Section 20.3.3, all queue operations call *exec* twice to be able to finish the reversal in time. On *Idle* queues *exec* has no effect. The implementation of *exec* is an extension of *rev_step* with specific considerations for each *status* value and is defined as follows:

```
exec :: 'a status ⇒ 'a status
exec (Rev ok (x # f) f' (y # r) r')
= Rev (ok + 1) f (x # f') r (y # r')
exec (Rev ok [] f' [y] r') = App ok f' (y # r')
exec (App 0 f' r') = Done r'
exec (App ok (x # f') r') = App (ok − 1) f' (x # r')
exec s = s
```

If the *status* is *Rev ok f f' r r'*, then *exec* performs two (or one if $f = []$) simultaneous reversal steps from *f* and *r* into *f'* and *r'*; moreover *ok* is incremented if a new element has been added to f'. Once f is exhausted and r is a singleton list, the remaining element is moved into r' and the *status* is updated to the next phase of reversal. In the *App ok* f' r' phase, *exec* moves elements from f' to r' until $ok = 0$, at which point r' becomes the new front by transitioning into *Done* r'. In all other cases *exec* behaves as the identity function. As is apparent from its implementation, a number of assumptions are required for *exec* to function properly and eventually produce *Done*. These assumption are discussed in Section 20.3.5.

If an element is removed from the queue during the reversal process, it also needs to be removed from the new front list (*fr*) being computed. Function *invalidate* is used to achieve this:

```
invalidate :: 'a status ⇒ 'a status
invalidate (Rev ok f f' r r') = Rev (ok − 1) f f' r r'
invalidate (App 0 f' (_ # r')) = Done r'
invalidate (App ok f' r') = App (ok − 1) f' r'
invalidate s = s
```

By decreasing the value of *ok*, the number of elements from f' that are moved into r' in phase (3) is reduced, since *exec* might produce *Done* early, once $ok = 0$, ignoring the remaining elements of f'. Furthermore, since f' is a reversal of the front list, elements left behind in its tail correspond directly to those being removed from the queue.

The rest of the implementation is shown below. Auxiliary function *exec*2, as its name suggests, applies *exec* twice and updates the queue accordingly if *Done* is returned.

```
exec2 :: 'a queue ⇒ 'a queue
exec2 q = (case exec (exec q) of
             Done fr ⇒ q⦇status = Idle, front = fr⦈ |
             newstatus ⇒ q⦇status = newstatus⦈)

check :: 'a queue ⇒ 'a queue
check q
= (if lenr q ≤ lenf q then exec2 q
   else let newstate = Rev 0 (front q) [] (rear q) []
        in exec2
            (q⦇lenf := lenf q + lenr q, status := newstate, rear := [],
                lenr := 0⦈))
```

```
empty :: 'a queue
empty = make 0 [] Idle [] 0

first :: 'a queue ⇒ 'a
first q = hd (front q)

enq :: 'a ⇒ 'a queue ⇒ 'a queue
enq x q = check (q⦇rear := x # rear q, lenr := lenr q + 1⦈)

deq :: 'a queue ⇒ 'a queue
deq q
= check
    (q⦇lenf := lenf q − 1, front := tl (front q),
         status := invalidate (status q)⦈)
```

The two main queue operations, *enq* and *deq*, alter *front* and *rear* as expected, with additional updates to *lenf* and *lenr* to keep track of the their length. To perform all "internal" operations, both functions call *check*. Additionally, *deq* uses *invalidate* to mark elements as removed.

Function *check* calls *exec*2 if *lenr* is not larger than *lenf*. Otherwise a reversal process is initiated: *rear* is emptied and *lenr* is set to 0; *lenf* is increased to the size of the whole queue since, conceptually, all element are now in the soon-to-be-computed front; the status *newstate* is initialized as described at the beginning of Section 20.3.2.

The time complexity of this implementation is clearly constant, since there are no recursive functions.

20.3.5 Functional Correctness

To show this implementation is an instance of the ADT *Queue*, we need a number of invariants to ensure the consistency of *'a queue* values are preserved by all operations.

Initially, as hinted by the definition of *exec*, values of type *'a status* should have certain properties to guarantee a *Done* result after a (finite) number of calls to *exec*. The predicate *inv_st* defines these properties as follows:

```
inv_st :: 'a status ⇒ bool
inv_st (Rev ok f f' r r') = (|f| + 1 = |r| ∧ |f'| = |r'| ∧ ok ≤ |f'|)
inv_st (App ok f′ r′) = (ok ≤ |f′| ∧ |f′| < |r′|)
inv_st Idle = True
inv_st (Done _) = True
```

These properties follow from our reversal strategy, initial configuration, and the definition of *exec* itself. First, *inv_st* ensures for pattern $Rev\ ok\ f\ f'\ r\ r'$ that arguments f and r follow the reversal strategy, and counter *ok* is only ever increased as elements are added to f'. Similarly for $App\ ok\ f'\ r'$, it must follow that r' remains larger than f', and $|f'|$ provides an upper bound for *ok*. Finally, all other patterns trivially fulfill the invariant.

Naturally, the consistency of *'a queue* values is defined as an extension of *inv_st* and predicates over all other fields in the queue. Operations *enq*, *deq* must preserve the relation of *lenr* and *lenf* with their corresponding lists, while *check* must start the reversal process at the right time and handle its result appropriately. The predicate *invar* defines the required properties:

```
invar q
= (lenf q = |front_list q| ∧ lenr q = |rear_list q| ∧
    lenr q ≤ lenf q ∧
    (case status q of
     Rev ok f f′ _ _ ⇒
       2 · lenr q ≤ |f′| ∧ ok ≠ 0 ∧
       2 · |f| + ok + 2 ≤ 2 · |front q|
     | App ok _ r ⇒ 2 · lenr q ≤ |r| ∧ ok + 1 ≤ 2 · |front q|
     | _ ⇒ True) ∧
    (∃ rest. front_list q = front q @ rest) ∧
    (∄ fr. status q = Done fr) ∧
    inv_st (status q))
```

For a queue q, value $rear\_list\ q$ is a list representing the rear of the queue in canonical order (first elements at the front), and it is defined as $(rev \circ rear)\ q$. Similarly, value $front\_list\ q$ represents the front of the queue; However, its definition is more intricate as it needs to account for the front list being generated by the reversal process. The function *rem_steps* calculates the number of steps required such that $\exists\, x.\ exec^{rem\_steps\ s}\ s = Done\ x$.

Using *invar* we can prove the properties of *enq* and *deq* required by the *Queue* ADT using this abstraction function:

```
list :: 'a queue ⇒ 'a list
list q = front_list q @ rear_list q
```

All other assumptions in *Queue* are straightforward.

Bibliographic Remarks

The representation of queues as pairs of lists is due to Burton [11]. The Hood-Melville queues are due to Hood and Melville [30]. The implementation is based on the presentation by Okasaki [62].

21

Splay Trees

Splay trees are fascinating self-organizing search trees. Self-organizing means that the tree structure is modified upon access (including *isin* queries) to improve the performance of subsequent operations. Concretely, every splay tree operation moves the element concerned to the root. Thus splay trees excel in applications where a small fraction of the entries are the targets of most of the operations. In general, splay trees perform as well as any static binary search tree.

Splay trees have two drawbacks. First, their performance guarantees (logarithmic running time of each operation) are only amortized. Self-organizing does not mean self-balancing: splay trees can become unbalanced, in contrast to, for example, red-black trees. Second, because *isin* modifies the tree, splay trees are less convenient to use in a purely functional language.

21.1 Implementation

The central operation on splay trees is the *splay* function shown in Figure 21.1. It rotates the given element x to the root of the tree if x is already in the tree. Otherwise the last element found before the search for x hits a leaf is rotated to the root.

Function *isin* has a trivial implementation in terms of *splay*:

```
isin :: 'a tree ⇒ 'a ⇒ bool
isin t x = (case splay x t of ⟨⟩ ⇒ False | ⟨_, a, _⟩ ⇒ x = a)
```

Except that *splay* creates a new tree that needs to be returned from a proper *isin* as well to achieve the amortized logarithmic complexity (see the discussion of observer functions at the end of Section 19.1). This is why splay trees

```
$splay\ x\ \langle AB,\ b,\ CD\rangle$
$= ($case $cmp\ x\ b$ of
    $LT \Rightarrow$ case $AB$ of
            $\langle\rangle \Rightarrow \langle AB,\ b,\ CD\rangle$ |
            $\langle A,\ a,\ B\rangle \Rightarrow$
              case $cmp\ x\ a$ of
              $LT \Rightarrow$ if $A = \langle\rangle$ then $\langle A,\ a,\ \langle B,\ b,\ CD\rangle\rangle$
                      else case $splay\ x\ A$ of
                           $\langle A_1,\ a',\ A_2\rangle \Rightarrow \langle A_1,\ a',\ \langle A_2,\ a,\ \langle B,\ b,\ CD\rangle\rangle\rangle$ |
              $EQ \Rightarrow \langle A,\ a,\ \langle B,\ b,\ CD\rangle\rangle$ |
              $GT \Rightarrow$ if $B = \langle\rangle$ then $\langle A,\ a,\ \langle B,\ b,\ CD\rangle\rangle$
                      else case $splay\ x\ B$ of
                           $\langle B_1,\ b',\ B_2\rangle \Rightarrow \langle\langle A,\ a,\ B_1\rangle,\ b',\ \langle B_2,\ b,\ CD\rangle\rangle$ |
    $EQ \Rightarrow \langle AB,\ b,\ CD\rangle$ |
    $GT \Rightarrow$ case $CD$ of
            $\langle\rangle \Rightarrow \langle AB,\ b,\ CD\rangle$ |
            $\langle C,\ c,\ D\rangle \Rightarrow$
              case $cmp\ x\ c$ of
              $LT \Rightarrow$ if $C = \langle\rangle$ then $\langle\langle AB,\ b,\ C\rangle,\ c,\ D\rangle$
                      else case $splay\ x\ C$ of
                           $\langle C_1,\ c',\ C_2\rangle \Rightarrow \langle\langle AB,\ b,\ C_1\rangle,\ c',\ \langle C_2,\ c,\ D\rangle\rangle$ |
              $EQ \Rightarrow \langle\langle AB,\ b,\ C\rangle,\ c,\ D\rangle$ |
              $GT \Rightarrow$ if $D = \langle\rangle$ then $\langle\langle AB,\ b,\ C\rangle,\ c,\ D\rangle$
                      else case $splay\ x\ D$ of
                           $\langle D_1,\ d,\ D_2\rangle \Rightarrow \langle\langle\langle AB,\ b,\ C\rangle,\ c,\ D_1\rangle,\ d,\ D_2\rangle)$
```

Fig. 21.1. Function *splay*

are inconvenient in functional languages. For the moment we ignore this aspect and stick with the above *isin* because it has the type required by the *Set* ADT.

The implementation of *insert* x t in Figure 21.2 is straightforward: let $\langle l,\ a,\ r\rangle = splay\ x\ t$; if $a = x$, return $\langle l,\ a,\ r\rangle$; otherwise make x the root of a suitable recombination of l, a and r.

The implementation of *delete* x t in Figure 21.3 starts similarly: let $\langle l,\ a,\ r\rangle = splay\ x\ t$; if $a \neq x$, return $\langle l,\ a,\ r\rangle$. Otherwise follow the deletion-by-replacing paradigm (Section 5.2.1): if $l \neq \langle\rangle$, splay the maximal element m in l to the root and replace x with it. Note that *splay_max* returns a tree that is just a glorified pair: if $t \neq \langle\rangle$ then $splay\_max\ t$ is of the form $\langle t',\ m,\ \langle\rangle\rangle$. The definition $splay\_max\ \langle\rangle = \langle\rangle$ is not really needed (*splay_max* is always called with non-$\langle\rangle$ argument) but some lemmas can be stated more slickly with this definition.

```
insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x t
= (if t = ⟨⟩ then ⟨⟨⟩, x, ⟨⟩⟩
   else case splay x t of
          ⟨l, a, r⟩ ⇒ case cmp x a of
                        LT ⇒ ⟨l, x, ⟨⟨⟩, a, r⟩⟩ |
                        EQ ⇒ ⟨l, a, r⟩ |
                        GT ⇒ ⟨⟨l, a, ⟨⟩⟩, x, r⟩)
```

Fig. 21.2. Function *insert*

```
delete :: 'a ⇒ 'a tree ⇒ 'a tree
delete x t
= (if t = ⟨⟩ then ⟨⟩
   else case splay x t of
          ⟨l, a, r⟩ ⇒
            if x ≠ a then ⟨l, a, r⟩
            else if l = ⟨⟩ then r
                 else case splay_max l of ⟨l', m, _⟩ ⇒ ⟨l', m, r⟩)

splay_max :: 'a tree ⇒ 'a tree
splay_max ⟨⟩ = ⟨⟩
splay_max ⟨A, a, ⟨⟩⟩ = ⟨A, a, ⟨⟩⟩
splay_max ⟨A, a, ⟨B, b, CD⟩⟩
= (if CD = ⟨⟩ then ⟨⟨A, a, B⟩, b, ⟨⟩⟩
   else case splay_max CD of ⟨C, c, D⟩ ⇒ ⟨⟨⟨A, a, B⟩, b, C⟩, c, D⟩)
```

Fig. 21.3. Functions *delete* and *splay_max*

21.2 Correctness

The *inorder* approach of Section 5.4 applies. Because the details are a bit different (everything is reduced to *splay*) we present the top-level structure.

The following easy inductive properties are used implicitly in a number of subsequent proofs:

$$\mathit{splay}\ a\ t = \langle\rangle \ \longleftrightarrow\ t = \langle\rangle$$
$$\mathit{splay\_max}\ t = \langle\rangle \ \longleftrightarrow\ t = \langle\rangle$$

Correctness of *isin*

$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{isin}\ t\ x = (x \in \mathit{set}\ (\mathit{inorder}\ t))$$

follows directly from this easy inductive property of *splay*:

$$splay\ x\ t = \langle l,\ a,\ r\rangle \wedge sorted\ (inorder\ t) \longrightarrow$$
$$(x \in set\ (inorder\ t)) = (x = a)$$

Correctness of *insert* and *delete*

$$sorted\ (inorder\ t) \longrightarrow inorder\ (insert\ x\ t) = ins\_list\ x\ (inorder\ t)$$
$$sorted\ (inorder\ t) \longrightarrow inorder\ (delete\ x\ t) = del\_list\ x\ (inorder\ t)$$

relies on the following characteristic inductive properties of *splay*:

$$inorder\ (splay\ x\ t) = inorder\ t \tag{21.1}$$
$$sorted\ (inorder\ t) \wedge splay\ x\ t = \langle l,\ a,\ r\rangle \longrightarrow$$
$$sorted\ (inorder\ l\ @\ x\ \#\ inorder\ r)$$

Correctness of *delete* also needs the inductive property

$$splay\_max\ t = \langle l,\ a,\ r\rangle \wedge sorted\ (inorder\ t) \longrightarrow$$
$$inorder\ l\ @\ [a] = inorder\ t \wedge r = \langle\rangle$$

Note that *inorder* $(splay\ x\ t) = inorder\ t$ is also necessary to justify the proper *isin* that returns the newly created tree as well.

Automation of the above proofs requires the lemmas in Figure 5.2 together with a few additional lemmas about *sorted*, *ins_list* and *del_list* that can be found in the Isabelle proofs.

Recall from Section 5.4 that correctness of *insert* and *delete* implies that they preserve $bst = sorted \circ inorder$. Similarly, (21.1) implies that *splay* preserves *bst*. Thus we may assume the invariant *bst* in the amortized analysis.

These two easy size lemmas are used implicitly below:

$$|splay\ a\ t| = |t| \qquad |splay\_max\ t| = |t|$$

21.3 Amortized Analysis

This section shows that *splay*, insertion and deletion all have amortized logarithmic complexity.

We define the potential Φ of a tree as the sum of the potentials φ of all nodes:

$$\Phi :: 'a\ tree \Rightarrow real$$
$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ a,\ r\rangle = \varphi\ \langle l,\ a,\ r\rangle + \Phi\ l + \Phi\ r$$

$$\varphi\ t \equiv \log_2\ |t|_1$$

The central result is the amortized complexity of *splay*. Function T_{splay} is shown in Appendix B.7. We follow (19.1) and define

$$A_{splay}\ a\ t\ =\ T_{splay}\ a\ t\ +\ \Phi\ (splay\ a\ t)\ -\ \Phi\ t$$

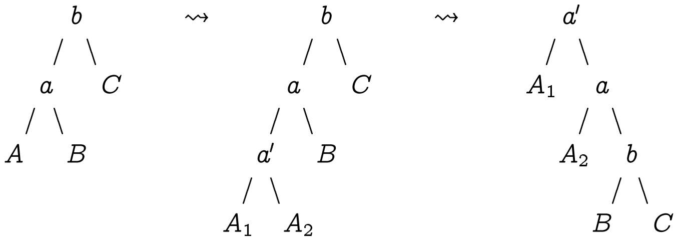
First we consider the case where the element is in the tree:

Theorem 21.1. $bst\ t \wedge \langle l,\ x,\ r\rangle \in subtrees\ t \longrightarrow$
$A_{splay}\ x\ t \leq 3 \cdot (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1$

Proof by induction on the computation of *splay*. The base cases involving $\langle\rangle$ are impossible. For example, consider the call $splay\ x\ t$ where $t = \langle\langle\rangle,\ b,\ C\rangle$ and $x < b$: from $\langle l,\ x,\ r\rangle \in subtrees\ t$ it follows that $x \in set\_tree\ t$ but because $bst\ t$ and $x < b$ this implies that $x \in set\_tree\ \langle\rangle$, a contradiction. There are three feasible base cases. The case $t = \langle\_\ ,\ x,\ \_\ \rangle$ is easy. We consider one of the two other symmetric cases. Let $t = \langle\langle A,\ x,\ B\rangle,\ b,\ C\rangle$ and $t' = splay\ x\ t = \langle A,\ x,\ \langle B,\ b,\ C\rangle\rangle$.

$$\begin{aligned}
&A_{splay}\ x\ t = \Phi\ t' - \Phi\ t + 1 && \text{by definition of } A_{splay} \text{ and } T_{splay}\\
&= \varphi\ t' + \varphi\ \langle B,\ b,\ C\rangle - \varphi\ t - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{by definition of } \Phi\\
&= \varphi\ \langle B,\ b,\ C\rangle - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{by definition of } \varphi\\
&\leq \varphi\ t - \varphi\ \langle A,\ x,\ B\rangle + 1 && \text{because } \varphi\ \langle B,\ b,\ C\rangle \leq \varphi\ t\\
&\leq 3 \cdot (\varphi\ t - \varphi\ \langle A,\ x,\ B\rangle) + 1 && \text{because } \varphi\ \langle A,\ x,\ B\rangle \leq \varphi\ t\\
&= 3 \cdot (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1 && \text{because } bst\ t \wedge \langle l,\ x,\ r\rangle \in subtrees\ t
\end{aligned}$$

There are four inductive cases. We consider two of them, the other two are symmetric variants. First the so-called zig-zig case:



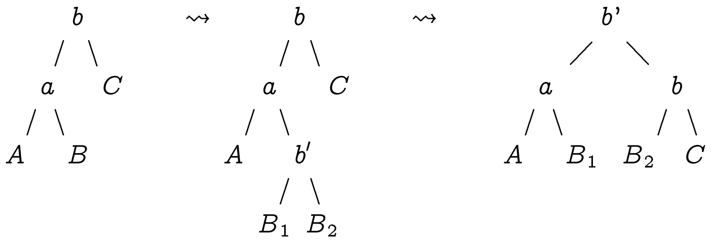
This is the case where $x < a < b$ and $A \neq \langle\rangle$. On the left we have the input and on the right the output of $splay\ x$. Because $A \neq \langle\rangle$, $splay\ x\ A = \langle A_1,\ a',\ A_2\rangle =: A'$ for some A_1, a' and A_2. The intermediate tree is obtained by replacing A by A'. This tree is shown for illustration purpose only; in the algorithm the right tree is constructed directly from the left one. Let $X = \langle l,\ x,\ r\rangle$. Clearly $X \in subtrees\ A$. We abbreviate compound trees like $\langle A,\ a,\ B\rangle$ by the names of their subtrees, in this case AB. First note that

$$\varphi\ A_1A_2BC = \varphi\ ABC \qquad (*)$$

because $|A'| = |\mathit{splay}\ x\ A| = |A|$. We can now prove the claim:

$$\begin{aligned}
& A_{\mathit{splay}}\ x\ ABC = T_{\mathit{splay}}\ x\ A + 1 + \Phi\ A_1 A_2 BC - \Phi\ ABC \\
& = T_{\mathit{splay}}\ x\ A + 1 + \Phi\ A_1 + \Phi\ A_2 + \varphi\ A_2 BC + \varphi\ BC - \Phi\ A - \varphi\ AB \\
& \hfill \text{by } (*) \text{ and definition of } \Phi \\
& = T_{\mathit{splay}}\ x\ A + \Phi\ A' - \varphi\ A' - \Phi\ A + \varphi\ A_2 BC + \varphi\ BC - \varphi\ AB + 1 \\
& = A_{\mathit{splay}}\ x\ A + \varphi\ A_2 BC + \varphi\ BC - \varphi\ AB - \varphi\ A' + 1 \\
& \leq 3 \cdot \varphi\ A + \varphi\ A_2 BC + \varphi\ BC - \varphi\ AB - \varphi\ A' - 3 \cdot \varphi\ X + 2 \\
& \hfill \text{by IH and } X \in \mathit{subtrees}\ A \\
& = 2 \cdot \varphi\ A + \varphi\ A_2 BC + \varphi\ BC - \varphi\ AB - 3 \cdot \varphi\ X + 2 \\
& \hfill \text{because } \varphi\ A = \varphi\ A' \\
& < \varphi\ A + \varphi\ A_2 BC + \varphi\ BC - 3 \cdot \varphi\ X + 2 \qquad \text{because } \varphi\ A < \varphi\ AB \\
& < \varphi\ A_2 BC + 2 \cdot \varphi\ ABC - 3 \cdot \varphi\ X + 1 \\
& \hfill \text{because } 1 + \lg\ x + \lg\ y < 2 \cdot \lg\ (x + y) \text{ if } x, y > 0 \\
& < 3 \cdot (\varphi\ ABC - \varphi\ X) + 1 \qquad \text{because } \varphi\ A_2 BC < \varphi\ ABC
\end{aligned}$$

Now we consider the so-called zig-zag case:



This is the case where $a < x < b$ and $B \neq \langle\rangle$. On the left we have the input and on the right the output of *splay x*. Because $B \neq \langle\rangle$, $\mathit{splay}\ x\ B = \langle B_1,\ b',\ B_2\rangle =: B'$ for some B_1, b' and B_2. The intermediate tree is obtained by replacing B by B'. Let $X = \langle l,\ x,\ r\rangle$. Clearly $X \in \mathit{subtrees}\ B$. The proof is very similar to the zig-zig case, the same naming conventions apply and we omit some details:

$$\begin{aligned}
& A_{\mathit{splay}}\ x\ ABC = T_{\mathit{splay}}\ x\ A + 1 + \Phi\ AB_1 B_2 C - \Phi\ ABC \\
& = A_{\mathit{splay}}\ x\ B + \varphi\ AB_1 + \varphi\ B_2 C - \varphi\ AB - \varphi\ B' + 1 \\
& \hfill \text{using } \varphi\ AB_1 B_2 C = \varphi\ ABC \\
& \leq 3 \cdot \varphi\ B + \varphi\ AB_1 + \varphi\ B_2 C - \varphi\ AB - \varphi\ B' - 3 \cdot \varphi\ X + 2 \\
& \hfill \text{by IH and } X \in \mathit{subtrees}\ B \\
& = 2 \cdot \varphi\ B + \varphi\ AB_1 + \varphi\ B_2 C - \varphi\ AB - 3 \cdot \varphi\ X + 2 \\
& \hfill \text{because } \varphi\ B = \varphi\ B' \\
& < \varphi\ B + \varphi\ AB_1 + \varphi\ B_2 C - 3 \cdot \varphi\ X + 2 \qquad \text{because } \varphi\ B < \varphi\ AB \\
& < \varphi\ B + 2 \cdot \varphi\ ABC - 3 \cdot \varphi\ X + 1 \\
& \hfill \text{because } 1 + \lg\ x + \lg\ y < 2 \cdot \lg\ (x + y) \text{ if } x, y > 0 \\
& < 3 \cdot (\varphi\ ABC - \varphi\ X) + 1 \qquad \text{because } \varphi\ B < \varphi\ ABC \qquad \square
\end{aligned}$$

Because $\varphi\ \langle l,\ x,\ r\rangle \geq 1$, the above theorem implies

Corollary 21.2. $\mathit{bst}\ t \wedge x \in \mathit{set\_tree}\ t \longrightarrow A_{\mathit{splay}}\ x\ t \leq 3 \cdot (\varphi\ t - 1) + 1$

If x is not in the tree we show that there is a y in the tree such that splaying with y would produce the same tree in the same time:

Lemma 21.3. $t \neq \langle\rangle \wedge \mathit{bst}\ t \longrightarrow$
$(\exists\, y{\in}\mathit{set\_tree}\ t.\ \mathit{splay}\ y\ t = \mathit{splay}\ x\ t \wedge T_{\mathit{splay}}\ y\ t = T_{\mathit{splay}}\ x\ t)$

Element y is the last element in the tree that the search for x encounters before it hits a leaf. Naturally, the proof is by induction on the computation of *splay*.

Combining this lemma with Corollary 21.2 yields the final unconditional amortized complexity of *splay* on BSTs:

Corollary 21.4. $\mathit{bst}\ t \longrightarrow A_{\mathit{splay}}\ x\ t \leq 3 \cdot \varphi\ t + 1$

The "$-\ 1$" has disappeared to accommodate the case $t = \langle\rangle$.

The amortized analysis of insertion is straightforward now. From the amortized complexity of *splay* it follows that

Lemma 21.5. $\mathit{bst}\ t \longrightarrow T_{\mathit{insert}}\ x\ t + \Phi\ (\mathit{insert}\ x\ t) - \Phi\ t \leq 4 \cdot \varphi\ t + 3$

We omit the proof which is largely an exercise in simple algebraic manipulations.

The amortized analysis of deletion is similar but a bit more complicated because of the additional function *splay_max* whose amortized running time is defined as usual:

$$A_{\mathit{splay\_max}}\ t = T_{\mathit{splay\_max}}\ t + \Phi\ (\mathit{splay\_max}\ t) - \Phi\ t$$

An inductive proof and then a simple case analysis yield

$$t \neq \langle\rangle \longrightarrow A_{\mathit{splay\_max}}\ t \leq 3 \cdot (\varphi\ t - 1) + 1$$

$$A_{\mathit{splay\_max}}\ t \leq 3 \cdot \varphi\ t + 1$$

Based on the canonical definitions of T_{delete} and $T_{\mathit{splay\_max}}$ the amortized logarithmic complexity of *delete* follows:

$$\mathit{bst}\ t \longrightarrow T_{\mathit{delete}}\ a\ t + \Phi\ (\mathit{delete}\ a\ t) - \Phi\ t \leq 6 \cdot \varphi\ t + 3$$

A running time analysis of *isin* is trivial because *isin* is just *splay* followed by a constant-time test.

21.4 Exercises

Exercise 21.1. Find a sequence of numbers $n_1,\ n_2,\ \ldots n_k$ such that the insertion of theses numbers one by one creates a tree of height k.

Bibliographic Remarks

Splay trees were invented and analyzed by Sleator and Tarjan [70] for which they received the 1999 ACM Paris Kanellakis Theory and Practice Award [38]. In addition to the amortized complexity as shown above they proved that splay trees perform as well as static BSTs (the Static Optimality Theorem) and conjectured that, roughly speaking, they even perform as well as any other BST-based algorithm. This Dynamic Optimality Conjecture is still open.

This chapter is based on earlier publications [69, 53, 54, 55].

22

Skew Heaps

22.1 Implementation of ADT *Priority_Queue_Merge*

Skew heaps are heaps in the sense of Section 14.1 and implement mergeable priority queues. Skew heaps can be viewed as a self-adjusting form of leftist heaps that attempts to maintain balance by unconditionally swapping all nodes in the merge path when merging two heaps:

$$\mathit{merge} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}$$

$$\begin{array}{l}
\mathit{merge}\ \langle\rangle\ t = t \\
\mathit{merge}\ t\ \langle\rangle = t \\
\mathit{merge}\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2) \\
= (\textbf{if}\ a_1 \le a_2\ \textbf{then}\ \langle \mathit{merge}\ t_2\ r_1,\ a_1,\ l_1\rangle\ \textbf{else}\ \langle \mathit{merge}\ t_1\ r_2,\ a_2,\ l_2\rangle)
\end{array}$$

The remaining operations are defined as in Section 14.1.

The following properties of *merge* have easy inductive proofs:

$$|\mathit{merge}\ t_1\ t_2| = |t_1| + |t_2|$$

$$\mathit{mset\_tree}\ (\mathit{merge}\ t_1\ t_2) = \mathit{mset\_tree}\ t_1 + \mathit{mset\_tree}\ t_2$$

$$\mathit{heap}\ t_1 \wedge \mathit{heap}\ t_2 \longrightarrow \mathit{heap}\ (\mathit{merge}\ t_1\ t_2)$$

Now it is straightforward to prove the correctness of the implementation w.r.t. the ADT *Priority_Queue_Merge*.

Skew heaps attempt to maintain balance, but this does not always work:

Exercise 22.1. Find a sequence of numbers $n_1,\ n_2,\ \ldots n_k$ such that the insertion of theses numbers one by one creates a tree of height k. Prove that this sequence will produce a tree of height k.

Nevertheless, insertion and deletion have amortized logarithmic complexity.

22.2 Amortized Analysis

The key is the definition of the potential. It counts the number of **right-heavy** (*rh*) nodes:

```
Φ :: 'a tree ⇒ int
Φ ⟨⟩ = 0
Φ ⟨l, _, r⟩ = Φ l + Φ r + rh l r

rh :: 'a tree ⇒ 'a tree ⇒ nat
rh l r = (if |l| < |r| then 1 else 0)
```

The rough intuition: because *merge* descends along the right spine, the more right-heavy nodes a tree contains, the longer *merge* takes.

Two auxiliary functions count the number of right-heavy nodes on the left spine (*lrh*) and left-heavy (= not right-heavy) nodes on the right spine (*rlh*):

```
lrh :: 'a tree ⇒ nat
lrh ⟨⟩ = 0
lrh ⟨l, _, r⟩ = rh l r + lrh l

rlh :: 'a tree ⇒ nat
rlh ⟨⟩ = 0
rlh ⟨l, _, r⟩ = 1 − rh l r + rlh r
```

The following properties have automatic inductive proofs

$$2^{lrh\ t} \le |t| + 1 \qquad 2^{rlh\ t} \le |t| + 1$$

and imply

$$lrh\ t \le \lg\ |t|_1 \qquad rlh\ t \le \lg\ |t|_1 \tag{22.1}$$

Now we are ready for the amortized analysis. All canonical time functions can be found in Appendix B.8. The key lemma is this:

Lemma 22.1. $T_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$
$\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$

Proof by induction on the computation of *merge*. We consider only the node-node case: let $t_1 = \langle l_1,\ a_1,\ r_1\rangle$ and $t_2 = \langle l_2,\ a_2,\ r_2\rangle$. W.l.o.g. assume $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$.

$$\begin{aligned}
&T_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
&= T_{merge}\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1 - \Phi\ t_1 - \Phi\ t_2 \\
&= T_{merge}\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1 - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2 \\
&\le lrh\ m + rlh\ t_2 + rlh\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1 && \text{by IH} \\
&= lrh\ m + rlh\ t_2 + rlh\ t_1 + rh\ m\ l_1 + 1 \\
&= lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 && \square
\end{aligned}$$

As a consequence we can prove the following logarithmic upper bound on the amortized complexity of *merge*:

$$\begin{aligned}
&T_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
&\le lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1 && \text{by Lemma 22.1} \\
&\le \lg\ |merge\ t_1\ t_2|_1 + \lg\ |t_1|_1 + \lg\ |t_2|_1 + 1 && \text{by (22.1)} \\
&\le \lg\ (|t_1|_1 + |t_2|_1 - 1) + \lg\ |t_1|_1 + \lg\ |t_2|_1 + 1 \\
& && \text{because } |merge\ t_1\ t_2| = |t_1| + |t_2| \\
&\le \lg\ (|t_1|_1 + |t_2|_1) + 2 \cdot \lg\ (|t_1|_1 + |t_2|_1) + 1 \\
& && \text{because } \lg\ x + \lg\ y \le 2 \cdot \lg\ (x + y) \text{ if } x,y > 0 \\
&= 3 \cdot \lg\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}$$

The amortized complexity of insertion and deletion follows easily from the complexity of *merge*:

$$\begin{aligned}
&T_{insert}\ a\ t + \Phi\ (insert\ a\ t) - \Phi\ t \le 3 \cdot \lg\ (|t|_1 + 2) + 2 \\
&T_{del\_min}\ t + \Phi\ (del\_min\ t) - \Phi\ t \le 3 \cdot \lg\ (|t|_1 + 2) + 2
\end{aligned}$$

Bibliographic Remarks

Skew heaps were invented by Sleator and Tarjan [71] as one of the first self-organizing data structures. Their presentation was imperative. Our presentation follows earlier work by Nipkow [53, 55] based on the functional account by Kaldewaij and Schoenmakers [37].

23

Pairing Heaps

The pairing heap is another form of a self-adjusting priority queue. Section 23.1 presents an intuitive version of pairing heaps based on lists. In the rest of the chapter we change to a slightly different presentation because it leads to a more succinct amortized analysis.

23.1 Implementation via Lists

A **pairing heap** is a heap in the sense that it is a tree with the minimal element at the root. Except that it is not a binary tree but a tree where each node has a list of children:

$$\textbf{datatype}\ 'a\ \mathit{heap} = \mathit{Empty} \mid \mathit{Hp}\ 'a\ ('a\ \mathit{heap}\ \mathit{list})$$

The abstraction function to multisets and the invariant follow the heap paradigm:

$$\mathit{mset\_heap} :: 'a\ \mathit{heap} \Rightarrow 'a\ \mathit{multiset}$$

$$\begin{array}{l}
\mathit{mset\_heap}\ \mathit{Empty} = \{\!\!\{\}\!\!\} \\
\mathit{mset\_heap}\ (\mathit{Hp}\ x\ \mathit{hs}) = \{\!\!\{x\}\!\!\} + \sum_{\#}\ (\mathit{mset}\ (\mathit{map}\ \mathit{mset\_heap}\ \mathit{hs}))
\end{array}$$

$$\mathit{pheap} :: 'a\ \mathit{heap} \Rightarrow \mathit{bool}$$

$$\begin{array}{l}
\mathit{pheap}\ \mathit{Empty} = \mathit{True} \\
\mathit{pheap}\ (\mathit{Hp}\ x\ \mathit{hs}) \\
= (\forall\, h{\in}\mathit{set}\ \mathit{hs}.\ (\forall\, y{\in_{\#}}\mathit{mset\_heap}\ h.\ x \le y) \wedge \mathit{pheap}\ h)
\end{array}$$

Note that *pheap* is not the full invariant: moreover, *Empty* does not occur inside a non-empty heap.

The implementations of *empty* and *get_min* are obvious and *insert* follows the standard heap paradigm:

```
empty = Empty

get_min :: 'a heap ⇒ 'a
get_min (Hp x _) = x

insert :: 'a ⇒ 'a heap ⇒ 'a heap
insert x h = merge (Hp x []) h
```

Function *merge* is not recursive (as in binary heaps) but simply adds one of the two heaps to the front of the top-level heaps of the other, depending on the root value:

```
merge :: 'a heap ⇒ 'a heap ⇒ 'a heap
merge h Empty = h
merge Empty h = h
merge (Hp x hsx =: hx) (Hp y hsy =: hy)
= (if x < y then Hp x (hy # hsx) else Hp y (hx # hsy))
```

Thus *merge* and *insert* have constant running time. All the work is offloaded on *del_min* which just calls *merge_pairs*:

```
del_min :: 'a heap ⇒ 'a heap
del_min Empty = Empty
del_min (Hp _ hs) = merge_pairs hs

merge_pairs :: 'a heap list ⇒ 'a heap
merge_pairs [] = Empty
merge_pairs [h] = h
merge_pairs (h₁ # h₂ # hs)
= merge (merge h₁ h₂) (merge_pairs hs)
```

Function *merge_pairs* is a compact way of expressing a two pass algorithm: on the first pass from left to right, it merges pairs of adjacent heaps (hence “pairing heap”) and on the second pass it merges the results in a cascade from right to left. A more readable formulation with the same running time is this one:

$$\mathit{del\_min} :: {}'a\ \mathit{heap} \Rightarrow {}'a\ \mathit{heap}$$

$$\mathit{del\_min}\ \mathit{Empty} = \mathit{Empty}$$
$$\mathit{del\_min}\ (\mathit{Hp}\ \_\ \mathit{hs}) = \mathit{pass}_2\ (\mathit{pass}_1\ \mathit{hs})$$

$$\mathit{pass}_1 :: {}'a\ \mathit{heap}\ \mathit{list} \Rightarrow {}'a\ \mathit{heap}\ \mathit{list}$$

$$\mathit{pass}_1\ (h_1\ \#\ h_2\ \#\ \mathit{hs}) = \mathit{merge}\ h_1\ h_2\ \#\ \mathit{pass}_1\ \mathit{hs}$$
$$\mathit{pass}_1\ \mathit{hs} = \mathit{hs}$$

$$\mathit{pass}_2 :: {}'a\ \mathit{heap}\ \mathit{list} \Rightarrow {}'a\ \mathit{heap}$$

$$\mathit{pass}_2\ [] = \mathit{Empty}$$
$$\mathit{pass}_2\ (h\ \#\ \mathit{hs}) = \mathit{merge}\ h\ (\mathit{pass}_2\ \mathit{hs})$$

The proof of $\mathit{pass}_2\ (\mathit{pass}_1\ \mathit{hs}) = \mathit{merge\_pairs}\ \mathit{hs}$ is an easy induction.

Clearly *del_min* can take linear time but it will turn out that the constant-time *insert* saves enough to guarantee amortized logarithmic complexity for both insertion and deletion.

We base the correctness proofs on the *merge_pairs* version of *del_min*. From the following lemmas (all proofs are routine inductions)

$$h \neq \mathit{Empty} \longrightarrow \mathit{get\_min}\ h \in_{\#} \mathit{mset\_heap}\ h$$
$$h \neq \mathit{Empty} \wedge \mathit{pheap}\ h \wedge x \in_{\#} \mathit{mset\_heap}\ h \longrightarrow \mathit{get\_min}\ h \leq x$$
$$\mathit{mset\_heap}\ (\mathit{merge}\ h_1\ h_2) = \mathit{mset\_heap}\ h_1 + \mathit{mset\_heap}\ h_2$$
$$\mathit{mset\_heap}\ (\mathit{merge\_pairs}\ \mathit{hs}) \\ = \textstyle\sum_{\#}\ (\mathit{image\_mset}\ \mathit{mset\_heap}\ (\mathit{mset}\ \mathit{hs}))$$
$$h \neq \mathit{Empty} \longrightarrow \\ \mathit{mset\_heap}\ (\mathit{del\_min}\ h) = \mathit{mset\_heap}\ h - \{\!\{\mathit{get\_min}\ h\}\!\}$$
$$\mathit{pheap}\ h_1 \wedge \mathit{pheap}\ h_2 \longrightarrow \mathit{pheap}\ (\mathit{merge}\ h_1\ h_2)$$
$$(\forall\, h {\in} \mathit{set}\ \mathit{hs}.\ \mathit{pheap}\ h) \longrightarrow \mathit{pheap}\ (\mathit{merge\_pairs}\ \mathit{hs})$$
$$\mathit{pheap}\ h \longrightarrow \mathit{pheap}\ (\mathit{del\_min}\ h)$$

the properties in the specifications *Priority_Queue*(*_Merge*) follow easily.

23.2 Amortized Analysis

The amortized analysis of pairing heaps is slightly simplified if we replace the above type of heaps by trees as follows: a heap $\mathit{Hp}\ x\ \mathit{hs}$ is expressed as the tree $\langle \mathit{hs},\ x,\ \langle\rangle\rangle$ and a list of heaps $[\mathit{Hp}\ x_1\ \mathit{hs}_1,\ \mathit{Hp}\ x_2\ \mathit{hs}_2,\ ...]$ is expressed

```
empty = ⟨⟩

get_min :: 'a tree ⇒ 'a
get_min ⟨_ , x, _⟩ = x

link :: 'a tree ⇒ 'a tree
link ⟨hsx, x, ⟨hsy, y, hs⟩⟩
= (if x < y then ⟨⟨hsy, y, hsx⟩, x, hs⟩ else ⟨⟨hsx, x, hsy⟩, y, hs⟩)
link hp = hp

pass_1 :: 'a tree ⇒ 'a tree
pass_1 ⟨hsx, x, ⟨hsy, y, hs⟩⟩ = link ⟨hsx, x, ⟨hsy, y, pass_1 hs⟩⟩
pass_1 hp = hp

pass_2 :: 'a tree ⇒ 'a tree
pass_2 ⟨hsx, x, hs⟩ = link ⟨hsx, x, pass_2 hs⟩
pass_2 ⟨⟩ = ⟨⟩

get_min :: 'a tree ⇒ 'a
get_min ⟨_ , x, _⟩ = x

merge :: 'a tree ⇒ 'a tree ⇒ 'a tree
merge ⟨⟩ hp = hp
merge hp ⟨⟩ = hp
merge ⟨hsx, x, ⟨⟩⟩ ⟨hsy, y, ⟨⟩⟩ = link ⟨hsx, x, ⟨hsy, y, ⟨⟩⟩⟩

insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x hp = merge ⟨⟨⟩, x, ⟨⟩⟩ hp
```

Fig. 23.1. Pairing heaps via trees

as the tree $\langle hs_1,\ x_1,\ \langle hs_2,\ x_2,\ ...\rangle...\rangle$. This simplifies the analysis because we now have to deal only with a single type, trees.

The code for the tree-representation of pairing heaps is shown in Figure 23.1. We work with the $pass_1/pass_2$ version of *del_min*. The correctness proof is very similar to what we saw in the previous section. We merely display the two invariants:

```
is_root :: 'a tree ⇒ bool
is_root hp = (case hp of ⟨⟩ ⇒ True | ⟨_, _, r⟩ ⇒ r = ⟨⟩)
```

$$
\begin{aligned}
&\mathit{pheap} :: \ 'a\ \mathit{tree} \Rightarrow \mathit{bool}\\
&\mathit{pheap}\ \langle\rangle = \mathit{True}\\
&\mathit{pheap}\ \langle l,\ x,\ r\rangle = ((\forall\, y{\in}\mathit{set\_tree}\ l.\ x \le y) \wedge \mathit{pheap}\ l \wedge \mathit{pheap}\ r)
\end{aligned}
$$

Now we turn to the amortized analysis. The potential of a tree is the sum of the logarithms of the sizes of the subtrees:

$$
\begin{aligned}
&\Phi :: \ 'a\ \mathit{tree} \Rightarrow \mathit{real}\\
&\Phi\ \langle\rangle = 0\\
&\Phi\ \langle l,\ x,\ r\rangle = \lg\ |\langle l,\ x,\ r\rangle| + \Phi\ l + \Phi\ r
\end{aligned}
$$

These easy inductive size-properties are frequently used implicitly below:

$$
\begin{aligned}
&|\mathit{link}\ \mathit{hp}| = |\mathit{hp}|\\
&|\mathit{pass}_1\ \mathit{hp}| = |\mathit{hp}|\\
&|\mathit{pass}_2\ \mathit{hp}| = |\mathit{hp}|\\
&\mathit{is\_root}\ h_1 \wedge \mathit{is\_root}\ h_2 \longrightarrow |\mathit{merge}\ h_1\ h_2| = |h_1| + |h_2|
\end{aligned}
$$

23.2.1 Potential Differences

We can now analyze the differences in potential caused by all the queue operations. In a separate step we will derive their amortized complexities.

For insertion, the following upper bound follows trivially from the definitions:

Lemma 23.1. $\mathit{is\_root}\ \mathit{hp} \longrightarrow \Phi\ (\mathit{insert}\ x\ \mathit{hp}) - \Phi\ \mathit{hp} \le \lg\ (|\mathit{hp}| + 1)$

For *merge* it needs a bit more work:

Lemma 23.2. $h_1 = \langle \mathit{hs}_1,\ x_1,\ \langle\rangle\rangle \wedge h_2 = \langle \mathit{hs}_2,\ x_2,\ \langle\rangle\rangle \longrightarrow$
$\Phi\ (\mathit{merge}\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2 \le \lg\ (|h_1| + |h_2|) + 1$

Proof. From

$$
\begin{aligned}
&\Phi\ (\mathit{merge}\ h_1\ h_2)\\
&= \Phi\ (\mathit{link}\ \langle \mathit{hs}_1,\ x_1,\ h_2\rangle)\\
&= \Phi\ \mathit{hs}_1 + \Phi\ \mathit{hs}_2 + \lg\ (|\mathit{hs}_1| + |\mathit{hs}_2| + 1) + \lg\ (|\mathit{hs}_1| + |\mathit{hs}_2| + 2)\\
&= \Phi\ \mathit{hs}_1 + \Phi\ \mathit{hs}_2 + \lg\ (|\mathit{hs}_1| + |\mathit{hs}_2| + 1) + \lg\ (|h_1| + |h_2|)
\end{aligned}
$$

it follows that

$$
\begin{aligned}
&\Phi\ (\mathit{merge}\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2\\
&= \lg\ (|\mathit{hs}_1| + |\mathit{hs}_2| + 1) + \lg\ (|h_1| + |h_2|)\\
&\quad - \lg\ (|\mathit{hs}_1| + 1) - \lg\ (|\mathit{hs}_2| + 1)
\end{aligned}
$$

$$\le \lg (|h_1| + |h_2|) + 1$$
because $\lg (1 + x + y) \le 1 + \lg (1 + x) + \lg (1 + y)$ if $x,y \ge 0$ □

Now we come to the core of the proof, the analysis of *del_min*. Its running time is linear in the number of nodes reachable by descending to the right (starting from the left child of the root). We denote this metric by *len*:

```
len :: 'a tree ⇒ nat
len ⟨⟩ = 0
len ⟨_, _, r⟩ = 1 + len r
```

Therefore we have to show that the potential change compensates for this linear work. Our main goal is this:

Theorem 23.3. $\Phi\ (\mathit{del\_min}\ \langle hs,\ x,\ \langle\rangle\rangle) - \Phi\ \langle hs,\ x,\ \langle\rangle\rangle$
$\le 2 \cdot \lg (|hs| + 1) - \mathit{len}\ hs + 2$

It will be proved in two steps: First we show that pass_1 frees enough potential to compensate for the work linear in *len hs* and increases the potential only by a logarithmic term. Then we show that the increase due to pass_2 is also only at most logarithmic. Combining these results one easily shows that the amortized running time of *del_min* is indeed logarithmic.

First we analyze the potential difference caused by pass_1:

Lemma 23.4. $\Phi\ (\mathit{pass}_1\ hs) - \Phi\ hs \le 2 \cdot \lg (|hs| + 1) - \mathit{len}\ hs + 2$

Proof by induction on the computation of pass_1. The base cases are trivial. We focus on the induction step. Let $t = \langle hs_1,\ x,\ \langle hs_2,\ y,\ hs\rangle\rangle$, $n_1 = |hs_1|$, $n_2 = |hs_2|$ and $m = |hs|$.

$$\begin{aligned}
&\Phi\ (\mathit{pass}_1\ t) - \Phi\ t\\
&= \lg (n_1 + n_2 + 1) - \lg (n_2 + m + 1) + \Phi\ (\mathit{pass}_1\ hs) - \Phi\ hs\\
&\le \lg (n_1 + n_2 + 1) - \lg (n_2 + m + 1)\\
&\quad + 2 \cdot \lg (m + 1) - \mathit{len}\ hs + 2 \qquad \text{by IH}\\
&\le 2 \cdot \lg (n_1 + n_2 + m + 1) - \lg (n_2 + m + 1) + \lg (m + 1)\\
&\quad - \mathit{len}\ hs \qquad \text{because } \lg x + \lg y + 2 \le 2 \cdot \lg (x + y) \text{ if } x,y > 0\\
&\le 2 \cdot \lg (n_1 + n_2 + m + 2) - \mathit{len}\ hs\\
&= 2 \cdot \lg |t| - \mathit{len}\ t + 2\\
&\le 2 \cdot \lg (|t| + 1) - \mathit{len}\ t + 2
\end{aligned}$$
□

Now we turn to pass_2:

Lemma 23.5. $hs \ne \langle\rangle \longrightarrow \Phi\ (\mathit{pass}_2\ hs) - \Phi\ hs \le \lg |hs|$

Proof by induction on *hs*. The base cases are trivial. The induction step (for $\langle hs_1,\ x,\ hs\rangle$) is trivial if $hs = \langle\rangle$. Assume $hs = \langle hs_2,\ y,\ r\rangle$. Now we need one more property of $pass_2$:

$$\exists\, hs_3\ z.\ pass_2\ \langle hs_2,\ y,\ r\rangle = \langle hs_3,\ z,\ \langle\rangle\rangle$$

The proof is a straightforward induction on *r*. This implies $|hs_3| + 1 = |hs|$ and thus

$$\begin{aligned} &\Phi\ (link\ \langle hs_1,\ x,\ pass_2\ hs\rangle) - \Phi\ hs_1 - \Phi\ (pass_2\ hs) \\ &= \lg\ (|hs_1| + |hs| + 1) + \lg\ (|hs_1| + |hs|) - \lg\ |hs| \qquad (*) \end{aligned}$$

Thus the overall claim follows:

$$\begin{aligned} &\Phi\ (pass_2\ \langle hs_1,\ x,\ hs\rangle) - \Phi\ \langle hs_1,\ x,\ hs\rangle \\ &= \Phi\ (link\ \langle hs_1,\ x,\ pass_2\ hs\rangle) - \Phi\ hs_1 - \Phi\ hs - \lg\ (|hs_1| + |hs| + 1) \\ &= \Phi\ (pass_2\ hs) - \Phi\ hs + \lg\ (|hs_1| + |hs|) - \lg\ |hs| && \text{by } (*) \\ &\le \lg\ (|hs_1| + |hs|) && \text{by IH} \\ &\le \lg\ |\langle hs_1,\ x,\ hs\rangle| && \square \end{aligned}$$

Corollary 23.6. $\Phi\ (pass_2\ hs) - \Phi\ hs \le \lg\ (|hs| + 1)$

Finally we can prove Theorem 23.3:

$$\begin{aligned} &\Phi\ (del\_min\ \langle hs,\ x,\ \langle\rangle\rangle) - \Phi\ \langle hs,\ x,\ \langle\rangle\rangle \\ &= \Phi\ (pass_2\ (pass_1\ hs)) - \lg\ (|hs| + 1) - \Phi\ hs \\ &\le \Phi\ (pass_1\ hs) - \Phi\ hs && \text{by Corollary 23.6} \\ &\le 2 \cdot \lg\ (|hs| + 1) - len\ hs + 2 && \text{by Lemma 23.4} \end{aligned}$$

23.2.2 Amortized Running Times

The canonical running time functions are displayed in Appendix B.9. It is now straightforward to derive these amortized running times:

$$is\_root\ h \longrightarrow T_{insert}\ a\ h + \Phi\ (insert\ a\ h) - \Phi\ h \le \lg\ (|h| + 1) + 1$$

$$\begin{aligned} &is\_root\ h_1 \wedge is\_root\ h_2 \longrightarrow \\ &T_{merge}\ h_1\ h_2 + \Phi\ (merge\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2 \\ &\le \lg\ (|h_1| + |h_2| + 1) + 2 \end{aligned}$$

They follow from the corresponding lemmas 23.1 and 23.2.

Combining this inductive upper bound for the running time of the two passes

$$T_{pass2}\ (pass_1\ hs_1) + T_{pass1}\ hs_1 \le len\ hs_1 + 2$$

with Theorem 23.3 yields the third and final amortized running time:

$$\begin{aligned} &is\_root\ h \longrightarrow \\ &T_{del\_min}\ h + \Phi\ (del\_min\ h) - \Phi\ h \le 2 \cdot \lg\ (|h| + 1) + 5 \end{aligned}$$

Thus we could prove that insertion, merging and deletion all have amortized logarithmic running times.

Bibliographic Remarks

Pairing heaps were invented by Fredman *et al.* [21] as a simpler but competitive alternative to Fibonacci heaps. The authors gave the amortized analysis presented above and conjectured that it can be improved. Later research confirmed this [33, 65, 34] but the final analysis is still open. An empirical study [46] showed that pairing heaps do indeed outperform Fibonacci heaps in practice. This chapter is based on an article by Nipkow and Brinkop [55].

Part V

Appendix

A

List Library

The following functions on lists are predefined:

$\mathit{length} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$|[]| = 0$
$|x\ \#\ \mathit{xs}| = |\mathit{xs}| + 1$

$(@) :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$[]\ @\ \mathit{ys} = \mathit{ys}$
$(x\ \#\ \mathit{xs})\ @\ \mathit{ys} = x\ \#\ \mathit{xs}\ @\ \mathit{ys}$

$\mathit{set} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{set}$

$\mathit{set}\ [] = \{\}$
$\mathit{set}\ (x\ \#\ \mathit{xs}) = \{x\} \cup \mathit{set}\ \mathit{xs}$

$\mathit{map} :: ({'a} \Rightarrow {'b}) \Rightarrow {'a}\ \mathit{list} \Rightarrow {'b}\ \mathit{list}$

$\mathit{map}\ f\ [] = []$
$\mathit{map}\ f\ (x\ \#\ \mathit{xs}) = f\ x\ \#\ \mathit{map}\ f\ \mathit{xs}$

$\mathit{filter} :: ({'a} \Rightarrow \mathit{bool}) \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$\mathit{filter}\ p\ [] = []$
$\mathit{filter}\ p\ (x\ \#\ \mathit{xs}) = (\textbf{if}\ p\ x\ \textbf{then}\ x\ \#\ \mathit{filter}\ p\ \mathit{xs}\ \textbf{else}\ \mathit{filter}\ p\ \mathit{xs})$

$\mathit{take} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

$\mathit{take}\ \_\ [] = []$
$\mathit{take}\ n\ (x\ \#\ \mathit{xs}) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow []\ |\ m + 1 \Rightarrow x\ \#\ \mathit{take}\ m\ \mathit{xs})$

```
drop :: nat ⇒ 'a list ⇒ 'a list
drop _ [] = []
drop n (x # xs) = (case n of 0 ⇒ x # xs | m + 1 ⇒ drop m xs)

hd :: 'a list ⇒ 'a
hd (x # xs) = x

tl :: 'a list ⇒ 'a list
tl [] = []
tl (x # xs) = xs

butlast :: 'a list ⇒ 'a list
butlast [] = []
butlast (x # xs) = (if xs = [] then [] else x # butlast xs)

rev :: 'a list ⇒ 'a list
rev [] = []
rev (x # xs) = rev xs @ [x]

(!) :: 'a list ⇒ nat ⇒ 'a
(x # xs) ! n = (case n of 0 ⇒ x | k + 1 ⇒ xs ! k)

list_update :: 'a list ⇒ nat ⇒ 'a ⇒ 'a list
[][_ := _] = []
(x # xs)[i := v] = (case i of 0 ⇒ v # xs | j + 1 ⇒ x # xs[j := v])

upt :: nat ⇒ nat ⇒ nat list
[_..<0] = []
[i..<j + 1] = (if i ≤ j then [i..<j] @ [j] else [])

replicate :: nat ⇒ 'a ⇒ 'a list
replicate 0 _ = []
replicate (n + 1) x = x # replicate n x

sum_list :: 'a list ⇒ 'a
sum_list [] = 0
sum_list (x # xs) = x + sum_list xs
```

```
min_list :: 'a list ⇒ 'a
min_list (x # xs)
= (case xs of [] ⇒ x | _ # _ ⇒ min x (min_list xs))

sorted_wrt :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool
sorted_wrt P [] = True
sorted_wrt P (x # ys) = ((∀ y∈set ys. P x y) ∧ sorted_wrt P ys)
```

B

Time Functions

B.1 Lists

$T_{\mathit{length}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{length}}\ [] = 1$
$T_{\mathit{length}}\ (\_ \ \#\ xs) = T_{\mathit{length}}\ xs + 1$

$T_{\mathit{map}} :: ({'a} \Rightarrow \mathit{nat}) \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{map}}\ \_\ [] = 1$
$T_{\mathit{map}}\ T_f\ (x\ \#\ xs) = T_f\ x + T_{\mathit{map}}\ T_f\ xs + 1$

$T_{\mathit{filter}} :: ({'a} \Rightarrow \mathit{nat}) \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{filter}}\ \_\ [] = 1$
$T_{\mathit{filter}}\ T_p\ (x\ \#\ xs) = T_p\ x + T_{\mathit{filter}}\ T_p\ xs + 1$

$T_{\mathit{take}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{take}}\ \_\ [] = 1$
$T_{\mathit{take}}\ n\ (\_\ \#\ xs) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow 1 \mid n' + 1 \Rightarrow T_{\mathit{take}}\ n'\ xs + 1)$

$T_{\mathit{drop}} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$

$T_{\mathit{drop}}\ \_\ [] = 1$
$T_{\mathit{drop}}\ n\ (\_\ \#\ xs) = (\textbf{case}\ n\ \textbf{of}\ 0 \Rightarrow 1 \mid n' + 1 \Rightarrow T_{\mathit{drop}}\ n'\ xs + 1)$

$T_{\mathit{length}}\ xs = |xs| + 1$
$T_{\mathit{map}}\ T_f\ xs = (\sum_{x \leftarrow xs} T_f\ x) + |xs| + 1$
$T_{\mathit{filter}}\ T_p\ xs = (\sum_{x \leftarrow xs} T_p\ x) + |xs| + 1$
$T_{\mathit{take}}\ n\ xs = \mathit{min}\ n\ |xs| + 1$
$T_{\mathit{drop}}\ n\ xs = \mathit{min}\ n\ |xs| + 1$

B.2 Selection

$$\begin{array}{l}
T_{\mathit{chop}} :: \mathit{nat} \Rightarrow {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{chop}}\ 0\ \_ = 1 \\
T_{\mathit{chop}}\ \_\ [] = 1 \\
T_{\mathit{chop}}\ n\ \mathit{xs} = T_{\mathit{take}}\ n\ \mathit{xs} + T_{\mathit{drop}}\ n\ \mathit{xs} + T_{\mathit{chop}}\ n\ (\mathit{drop}\ n\ \mathit{xs})
\end{array}$$

$$\begin{array}{l}
T_{\mathit{partition3}} :: {}'a \Rightarrow {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{partition3}}\ \_\ [] = 1 \\
T_{\mathit{partition3}}\ x\ (\_ \# \mathit{ys}) = T_{\mathit{partition3}}\ x\ \mathit{ys} + 1
\end{array}$$

$$\begin{array}{l}
T_{\mathit{slow\_select}} :: \mathit{nat} \Rightarrow {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{slow\_select}}\ k\ \mathit{xs} = T_{\mathit{isort}}\ \mathit{xs} + T_{\mathit{nth}}\ (\mathit{isort}\ \mathit{xs})\ k + 1
\end{array}$$

$$\begin{array}{l}
T_{\mathit{slow\_median}} :: {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{slow\_median}}\ \mathit{xs} = T_{\mathit{slow\_select}}\ ((|\mathit{xs}| - 1)\ \mathit{div}\ 2)\ \mathit{xs} + 1
\end{array}$$

$$\begin{array}{l}
T_{\mathit{chop}}\ d\ \mathit{xs} \le 5 \cdot |\mathit{xs}| + 1 \\
T_{\mathit{partition3}}\ x\ \mathit{xs} = |\mathit{xs}| + 1 \\
T_{\mathit{slow\_select}}\ k\ \mathit{xs} \le |\mathit{xs}|^2 + 3 \cdot |\mathit{xs}| + 3 \\
T_{\mathit{slow\_median}}\ \mathit{xs} \le |\mathit{xs}|^2 + 3 \cdot |\mathit{xs}| + 4
\end{array}$$

B.3 2-3 Trees

$$\begin{array}{l}
T_{\mathit{join\_adj}} :: {}'a\ \mathit{tree23s} \Rightarrow \mathit{nat} \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (T\ \_)) = 1 \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ (T\ \_))) = 1 \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ \mathit{ts})) = T_{\mathit{join\_adj}}\ \mathit{ts} + 1
\end{array}$$

$$\begin{array}{l}
T_{\mathit{join\_adj}} :: {}'a\ \mathit{tree23s} \Rightarrow \mathit{nat} \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (T\ \_)) = 1 \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ (T\ \_))) = 1 \\
T_{\mathit{join\_adj}}\ (\mathit{TTs}\ \_\ \_\ (\mathit{TTs}\ \_\ \_\ \mathit{ts})) = T_{\mathit{join\_adj}}\ \mathit{ts} + 1
\end{array}$$

$$\begin{array}{l}
T_{\mathit{tree23\_of\_list}} :: {}'a\ \mathit{list} \Rightarrow \mathit{nat} \\
T_{\mathit{tree23\_of\_list}}\ \mathit{as} = T_{\mathit{leaves}}\ \mathit{as} + T_{\mathit{join\_all}}\ (\mathit{leaves}\ \mathit{as}) + 1
\end{array}$$

B.4 Leftist Heaps

$$\begin{array}{l}
T_{merge} :: ('a \times nat)\ tree \Rightarrow ('a \times nat)\ tree \Rightarrow nat \\
T_{merge}\ \langle\rangle\ t = 1 \\
T_{merge}\ t\ \langle\rangle = 1 \\
T_{merge}\ (\langle l_1,\ (a_1,\ n_1),\ r_1\rangle =: t_1)\ (\langle l_2,\ (a_2,\ n_2),\ r_2\rangle =: t_2) \\
= (\textbf{if}\ a_1 \le a_2\ \textbf{then}\ T_{merge}\ r_1\ t_2\ \textbf{else}\ T_{merge}\ t_1\ r_2) + 1
\end{array}$$

$$\begin{array}{l}
T_{insert} :: 'a \Rightarrow ('a \times nat)\ tree \Rightarrow nat \\
T_{insert}\ x\ t = T_{merge}\ \langle\langle\rangle,\ (x,\ 1),\ \langle\rangle\rangle\ t + 1
\end{array}$$

$$\begin{array}{l}
T_{del\_min} :: ('a \times nat)\ tree \Rightarrow nat \\
T_{del\_min}\ \langle\rangle = 1 \\
T_{del\_min}\ \langle l,\ \_,\ r\rangle = T_{merge}\ l\ r + 1
\end{array}$$

B.5 Binomial Heaps

$$\begin{array}{l}
T_{link} :: 'a\ tree \Rightarrow 'a\ tree \Rightarrow nat \\
T_{link}\ \_\ \_ = 1
\end{array}$$

$$\begin{array}{l}
T_{ins\_tree} :: 'a\ tree \Rightarrow 'a\ tree\ list \Rightarrow nat \\
T_{ins\_tree}\ \_\ [] = 1 \\
T_{ins\_tree}\ t_1\ (t_2\ \#\ ts) \\
= (\textbf{if}\ rank\ t_1 < rank\ t_2\ \textbf{then}\ 1 \\
\quad\ \textbf{else}\ T_{link}\ t_1\ t_2 + T_{ins\_tree}\ (link\ t_1\ t_2)\ ts)
\end{array}$$

$$\begin{array}{l}
T_{insert} :: 'a \Rightarrow 'a\ tree\ list \Rightarrow nat \\
T_{insert}\ x\ ts = T_{ins\_tree}\ (Node\ 0\ x\ [])\ ts + 1
\end{array}$$

$$\begin{array}{l}
T_{merge} :: 'a\ tree\ list \Rightarrow 'a\ tree\ list \Rightarrow nat \\
T_{merge}\ \_\ [] = 1 \\
T_{merge}\ []\ (\_\ \#\ \_) = 1 \\
T_{merge}\ (t_1\ \#\ ts_1 =: h_1)\ (t_2\ \#\ ts_2 =: h_2) \\
= 1\ + \\
\quad (\textbf{if}\ rank\ t_1 < rank\ t_2\ \textbf{then}\ T_{merge}\ ts_1\ h_2
\end{array}$$

```
    $\textbf{else if } \mathit{rank}\ t_2 < \mathit{rank}\ t_1 \textbf{ then } T_{\mathit{merge}}\ h_1\ \mathit{ts}_2$
        $\textbf{else } T_{\mathit{ins\_tree}}\ (\mathit{link}\ t_1\ t_2)\ (\mathit{merge}\ \mathit{ts}_1\ \mathit{ts}_2) + T_{\mathit{merge}}\ \mathit{ts}_1\ \mathit{ts}_2)$
```

$$T_{\mathit{get\_min}} :: {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{get\_min}}\ [\_] = 1$$
$$T_{\mathit{get\_min}}\ (\_ \mathbin{\#} v \mathbin{\#} \mathit{va}) = 1 + T_{\mathit{get\_min}}\ (v \mathbin{\#} \mathit{va})$$

$$T_{\mathit{get\_min\_rest}} :: {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{get\_min\_rest}}\ [\_] = 1$$
$$T_{\mathit{get\_min\_rest}}\ (\_ \mathbin{\#} v \mathbin{\#} \mathit{va}) = 1 + T_{\mathit{get\_min\_rest}}\ (v \mathbin{\#} \mathit{va})$$

$$T_{\mathit{rev}} :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{rev}}\ \mathit{xs} = |\mathit{xs}| + 1$$

$$T_{\mathit{del\_min}} :: {'a}\ \mathit{tree}\ \mathit{list} \Rightarrow \mathit{nat}$$

```
$T_{\mathit{del\_min}}\ \mathit{ts}$
$= T_{\mathit{get\_min\_rest}}\ \mathit{ts}\ +$
  $(\textbf{case } \mathit{get\_min\_rest}\ \mathit{ts} \textbf{ of}$
   $(\mathit{Node}\ \_\ \_\ \mathit{ts}_1,\ \mathit{ts}_2) \Rightarrow T_{\mathit{rev}}\ \mathit{ts}_1 + T_{\mathit{merge}}\ (\mathit{rev}\ \mathit{ts}_1)\ \mathit{ts}_2) +$
  $1$
```

B.6 Queues

$$T_{\mathit{norm}} :: {'a}\ \mathit{list} \times {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{norm}}\ (\mathit{fs},\ \mathit{rs}) = (\textbf{if } \mathit{fs} = [] \textbf{ then } T_{\mathit{itrev}}\ \mathit{rs}\ [] \textbf{ else } 0) + 1$$

$$T_{\mathit{enq}} :: {'a} \Rightarrow {'a}\ \mathit{list} \times {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{enq}}\ a\ (\mathit{fs},\ \mathit{rs}) = T_{\mathit{norm}}\ (\mathit{fs},\ a \mathbin{\#} \mathit{rs}) + 1$$

$$T_{\mathit{deq}} :: {'a}\ \mathit{list} \times {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{deq}}\ (\mathit{fs},\ \mathit{rs}) = (\textbf{if } \mathit{fs} = [] \textbf{ then } 0 \textbf{ else } T_{\mathit{norm}}\ (\mathit{tl}\ \mathit{fs},\ \mathit{rs})) + 1$$

$$T_{\mathit{first}} :: {'a}\ \mathit{list} \times {'a}\ \mathit{list} \Rightarrow \mathit{nat}$$

$$T_{\mathit{first}}\ (\_ \mathbin{\#} \_,\ \_) = 1$$

```
$T_{is\_empty} :: 'a\ list \times 'a\ list \Rightarrow nat$
$T_{is\_empty}\ (\_,\ \_) = 1$
```

B.7 Splay Trees

```
$T_{splay} :: 'a \Rightarrow 'a\ tree \Rightarrow nat$
$T_{splay}\ \_\ \langle\rangle = 1$
$T_{splay}\ x\ \langle AB,\ b,\ CD\rangle$
= (case $cmp\ x\ b$ of
    $LT \Rightarrow$ case $AB$ of
             $\langle\rangle \Rightarrow 1$ |
             $\langle A,\ a,\ B\rangle \Rightarrow$ case $cmp\ x\ a$ of
                                $LT \Rightarrow$ if $A = \langle\rangle$ then $1$ else $T_{splay}\ x\ A + 1$ |
                                $EQ \Rightarrow 1$ |
                                $GT \Rightarrow$ if $B = \langle\rangle$ then $1$ else $T_{splay}\ x\ B + 1$ |
    $EQ \Rightarrow 1$ |
    $GT \Rightarrow$ case $CD$ of
             $\langle\rangle \Rightarrow 1$ |
             $\langle C,\ c,\ D\rangle \Rightarrow$ case $cmp\ x\ c$ of
                                $LT \Rightarrow$ if $C = \langle\rangle$ then $1$ else $T_{splay}\ x\ C + 1$ |
                                $EQ \Rightarrow 1$ |
                                $GT \Rightarrow$ if $D = \langle\rangle$ then $1$ else $T_{splay}\ x\ D + 1$)

$T_{splay\_max} :: 'a\ tree \Rightarrow nat$
$T_{splay\_max}\ \langle\rangle = 1$
$T_{splay\_max}\ \langle\_,\ \_,\ \langle\rangle\rangle = 1$
$T_{splay\_max}\ \langle\_,\ \_,\ \langle\_,\ \_,\ C\rangle\rangle$ = (if $C = \langle\rangle$ then $1$ else $T_{splay\_max}\ C + 1$)

$T_{insert} :: 'a \Rightarrow 'a\ tree \Rightarrow nat$
$T_{insert}\ x\ t = 1 +$ (if $t = \langle\rangle$ then $0$ else $T_{splay}\ x\ t$)

$T_{delete} :: 'a \Rightarrow 'a\ tree \Rightarrow nat$
$T_{delete}\ x\ t$
$= 1 +$
  (if $t = \langle\rangle$ then $0$
  else $T_{splay}\ x\ t\ +$
      $($case $splay\ x\ t$ of
       $\langle l,\ a,\ \_\rangle \Rightarrow$
         if $x \neq a$ then $0$ else if $l = \langle\rangle$ then $0$ else $T_{splay\_max}\ l))$
```

B.8 Skew Heaps

```
$T_{merge}$ :: $'a\ tree \Rightarrow\ 'a\ tree \Rightarrow nat$
$T_{merge}\ \langle\rangle\ \_\ = 1$
$T_{merge}\ \_\ \langle\rangle = 1$
$T_{merge}\ \langle l_1,\ a_1,\ r_1\rangle\ \langle l_2,\ a_2,\ r_2\rangle$
$=$ (if $a_1 \leq a_2$ then $T_{merge}\ \langle l_2,\ a_2,\ r_2\rangle\ r_1$ else $T_{merge}\ \langle l_1,\ a_1,\ r_1\rangle\ r_2) + 1$

$T_{insert}$ :: $'a \Rightarrow\ 'a\ tree \Rightarrow int$
$T_{insert}\ a\ t = T_{merge}\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ t + 1$

$T_{del\_min}$ :: $'a\ tree \Rightarrow int$
$T_{del\_min}\ t =$ (case $t$ of $\langle\rangle \Rightarrow 1 \mid \langle t_1,\ \_,\ t_2\rangle \Rightarrow T_{merge}\ t_1\ t_2 + 1)$
```

B.9 Pairing Heaps

```
$T_{insert}$ :: $'a \Rightarrow\ 'a\ tree \Rightarrow nat$
$T_{insert}\ \_\ \_\ = 1$

$T_{merge}$ :: $'a\ tree \Rightarrow\ 'a\ tree \Rightarrow nat$
$T_{merge}\ \_\ \_\ = 1$

$T_{del\_min}$ :: $'a\ tree \Rightarrow nat$
$T_{del\_min}\ \langle\rangle = 1$
$T_{del\_min}\ \langle hs,\ \_,\ \_\rangle = T_{pass2}\ (pass_1\ hs) + T_{pass1}\ hs + 1$

$T_{pass1}$ :: $'a\ tree \Rightarrow nat$
```

$$T_{pass1}\ \langle\_,\ \_,\ \langle\_,\ \_,\ hs'\rangle\rangle = T_{pass1}\ hs' + 1$$
$$T_{pass1}\ \langle\rangle = 1$$
$$T_{pass1}\ \langle\_,\ \_,\ \langle\rangle\rangle = 1$$

$$T_{pass2} :: {}'a\ tree \Rightarrow nat$$
$$T_{pass2}\ \langle\rangle = 1$$
$$T_{pass2}\ \langle\_,\ \_,\ hs\rangle = T_{pass2}\ hs + 1$$

C

Notation

C.1 Symbol Table

The following table gives an overview of all the special symbols used in this book and how to enter them into Isabelle. The second column shows the full internal name of the symbol; the third column shows additional ASCII abbreviations. Either of these can be used to input the character using the auto-completion popup.

	Code	ASCII abbrev.	Comment
λ	`\<lambda>`	`%`	function abstraction
≡	`\<equiv>`	`==`	meta equality
≠	`\<noteq>`	`~=`	
⋀	`\<And>`	`!!`	meta ∀-quantifier
∀	`\<forall>`	`!`	HOL ∀-quantifier
∃	`\<exists>`	`?`	
⟹	`\<Longrightarrow>`	`==>`	meta implication
⟶	`\<longrightarrow>`	`->`	HOL implication
⟷	`\<longleftrightarrow>`	`<->` or `<-->`	
⇒	`\<Rightarrow>`	`=>`	arrow in function types
←	`\<leftarrow>`	`<-`	list comprehension syntax
¬	`\<not>`	`~`	
∧	`\<and>`	`/\` or `&`	
∨	`\<or>`	`\/` or `|`	
∈	`\<in>`	`:`	
∉	`\<notin>`	`~:`	
∪	`\<union>`	`Un`	

	Code	ASCII abbrev.	Comment
∩	`\<inter>`	`Int`	
⋃	`\<Union>`	`Union` or `UN`	union/intersection of a set of sets
⋂	`\<Inter>`	`Inter` or `INT`	
⊆	`\<subseteq>`	`(=`	
⊂	`\<subset>`		
≤	`\<le>`	`<=`	
≥	`\<ge>`	`>=`	
∘	`\<circ>`		function composition
×	`\<times>`	`<*>`	cartesian prod., prod. type
|	`\<bar>`	`||`	absolute value
⌊	`\<lfloor>`	`[.`	`floor`
⌋	`\<rfloor>`	`.]`	
⌈	`\<lceil>`	`[.`	`ceiling`
⌉	`\<rceil>`	`.]`	
∑	`\<Sum>`	`SUM`	see Section C.3
∏	`\<Prod>`	`PROD`	

Note that the symbols “⦃” and “⦄” that is used for multiset notation in the book do not exist Isabelle; instead, the ASCII notation `{#` and `#}` are used (cf. Section C.3).

C.2 Subscripts and Superscripts

In addition to this, subscripts and superscripts with a single symbol can be rendered using two special symbols, `\<^sub>` and `\<^sup>`. The term x_0 for instance can be input as `x\<^sub>0`.

Longer subscripts and superscripts can be written using the symbols `\<^bsub>...\<^esub>` and `\<^bsup>...\<^esup>`, but this is only rendered in the somewhat visually displeasing form ⇘ . . . ⇙ and ⇗ . . . ⇖ by Isabelle/jEdit.

C.3 Syntactic Sugar

The following table lists relevant syntactic sugar that is used in the book or its supplementary material. In some cases, the book notation deviates slightly from the Isabelle notation for better readability.

The last column gives the formal meaning of the notation (i.e. what it expands to). In most cases, this is not important for the user to know, but it can occasionally be useful to find relevant lemmas, or to understand that e.g. if one encounters the term *sum f A*, this is just the η-contracted form of $\sum x{\in}A.\ f\ x$.

The variables in the table follow the following convention:

- x and y are of arbitrary type
- m and n are natural numbers
- P and Q are boolean values or predicates
- *xs* is a list
- A is a set
- M is a multiset

Book notation	Isabelle notation	Meaning	
Arithmetic (for numeric types)			
$x \cdot y$	$x * y$	*times x y*	
$x\ /\ y$ or $\frac{x}{y}$	$x\ /\ y$	*divide x y*	(for type *real*)
x div y	*x div y*	*divide x y*	(for type *nat* or *int*)
$\lvert x \rvert$	$\lvert x \rvert$	*abs x*	
$\lfloor x \rfloor$	$\lfloor x \rfloor$	*floor x*	
$\lceil x \rceil$	$\lceil x \rceil$	*ceiling x*	
x^n	x ^ n	*power x n*	
Lists			
$\lvert xs \rvert$		*length xs*	
[]	[]	*Nil*	
x # *xs*	*x* # *xs*	*Cons x xs*	
$[x,\ y]$	$[x,\ y]$	*x* # *y* # []	
$[m..{<}n]$	$[m..{<}n]$	*upt m n*	
xs ! *n*	*xs* ! *n*	*nth xs n*	
$xs[n := y]$	$xs[n := y]$	*list_update xs n y*	

Book notation	Isabelle notation	Meaning
	Sets	
$\{\}$	$\{\}$	empty
$\{x,\ y\}$	$\{x,\ y\}$	$\mathit{insert}\ x\ (\mathit{insert}\ y\ \{\})$
$x \in A$	$x \in A$	$\mathit{Set.member}\ x\ A$
$x \notin A$	$x \notin A$	$\neg(x \in A)$
$A \cup B$	$A \cup B$	$\mathit{union}\ A\ B$
$A \cap B$	$A \cap B$	$\mathit{inter}\ A\ B$
$A \subseteq B$	$A \subseteq B$	$\mathit{subset\_eq}\ A\ B$
$A \subset B$	$A \subset B$	$\mathit{subset}\ A\ B$
$f\ \text{‘}\ A$	$f\ \text{‘}\ A$	$\mathit{image}\ f\ A$
$f\ -\text{‘}\ A$	$f\ -\text{‘}\ A$	$\mathit{vimage}\ f\ A$
$\{x \mid P\ x\}$	$\{x.\ P\ x\}$	$\mathit{Collect}\ P$
$\{x \in A \mid P\ x\}$	$\{x{\in}A.\ P\ x\}$	$\{x.\ P\ x \wedge x \in A\}$
$\{f\ x\ y \mid P\ x\ y\}$	$\{f\ x\ y\ \mid x\ y.\ P\ x\ y\}$	$\{z.\ \exists\, x\ y.\ z = f\ x\ y \wedge P\ x\ y\}$
$\bigcup_{x \in A} f\ x$	$\bigcup x{\in}A.\ f\ x$	$\bigcup (f\ \text{‘}\ A)$
$\forall\, x{\in}A.\ P\ x$	$\forall\, x{\in}A.\ P\ x$	$\mathit{Ball}\ A\ P$
$\exists\, x{\in}A.\ P\ x$	$\exists\, x{\in}A.\ P\ x$	$\mathit{Bex}\ A\ P$
	Multisets	
$\lvert M \rvert$		$\mathit{size}\ M$
$\{\!\!\{\}\!\!\}$	$\{\#\}$	$\mathit{empty\_mset}$
$\{\!\!\{x\}\!\!\} + M$		$\mathit{add\_mset}\ x\ M$
$\{\!\!\{x,\ y\}\!\!\}$	$\{\#x,\ y\#\}$	$\mathit{add\_mset}\ x\ (\mathit{add\_mset}\ y\ \{\#\})$
$x \in_{\#} M$	$x \in\# M$	$x \in \mathit{set\_mset}\ M$
$x \notin_{\#} M$	$x \notin\# M$	$\neg(x \in\# M)$
$\{\!\!\{x \in_{\#} M \mid P\ x\}\!\!\}$	$\{\#\ x{\in}\#\ M.\ P\ x\ \#\}$	$\mathit{filter\_mset}\ P\ M$
$\{\!\!\{f\ x \mid x \in_{\#} M\}\!\!\}$	$\{\#\ f\ x.\ x \in\# M\ \#\}$	$\mathit{image\_mset}\ f\ M$
$\forall\, x{\in_{\#}}M.\ P\ x$	$\forall\, x{\in}\#M.\ P\ x$	$\forall\, x{\in}\mathit{set\_mset}\ M.\ P\ x$
$\exists\, x{\in_{\#}}M.\ P\ x$	$\exists\, x{\in}\#M.\ P\ x$	$\exists\, x{\in}\mathit{set\_mset}\ M.\ P\ x$
$M \subseteq_{\#} M'$	$M \subseteq\# M'$	$\mathit{subseteq\_mset}\ M\ M'$

Book notation	Isabelle notation	Meaning
	Sums	
$\sum A$	$\sum A$	$\mathit{sum}\ (\lambda x.\ x)\ A$
$\sum_{x \in A} f\ x$	$\sum x{\in}A.\ f\ x$	$\mathit{sum}\ f\ A$
$\sum_{\#} M$	$\sum_{\#} M$	$\mathit{sum\_mset}\ M$
$\sum_{x \in_{\#} M} f\ x$	$\sum x{\in}\#M.\ f\ x$	$\mathit{sum\_mset}\ (\mathit{image\_mset}\ f\ M)$
$\sum_{x \leftarrow xs} f\ x$	$\sum x{\leftarrow}xs.\ f\ x$	$\mathit{sum\_list}\ (\mathit{map}\ f\ xs)$
	(analogous for products)	
	Intervals (for ordered types)	
$\{x..\}$	$\{x..\}$	$\mathit{atLeast}\ x$
$\{..y\}$	$\{..y\}$	$\mathit{atMost}\ y$
$\{x..y\}$	$\{x..y\}$	$\mathit{atLeastAtMost}\ x\ y$
$\{x..{<}y\}$	$\{x..{<}y\}$	$\mathit{atLeastLessThan}\ x\ y$
$\{x{<}..y\}$	$\{x{<}..y\}$	$\mathit{greaterThanAtMost}\ x\ y$
$\{x{<}..{<}y\}$	$\{x{<}..{<}y\}$	$\mathit{greaterThanLessThan}\ x\ y$

References

1. S. Adams. Efficient sets - A balancing act. *J. Funct. Program.*, 3(4):553–561, 1993.
2. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962. Translated from Russian by M.J. Ricci.
3. M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.
4. A. Appel. Efficient verified red-black trees. Unpublished, 2011.
5. C. Ballarin. *Tutorial to Locales and Locale Interpretation*. https://isabelle.in.tum.de/doc/locales.pdf.
6. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
7. S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. Théry, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *LNCS*, pages 19–36. Springer, 1999.
8. J. C. Blanchette. Proof pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL. *J. Autom. Reasoning*, 43(1):1–18, 2009.
9. G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In C. Scheideler and S. Gilbert, editors, *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016*, pages 253–264. ACM, 2016.
10. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci*, 7(4):448–461, 1973.
11. F. W. Burton. An efficient functional implementation of FIFO queues. *Inf. Process. Lett.*, 14(5):205–206, 1982.
12. S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. *ACM J. Exp. Algorithmics*, 3:2, 1998.
13. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
14. C. A. Crane. *Linear Lists and Priority Queues as Balanced Binary Trees*. PhD thesis, Stanford University, 1972. STAN-CS-72-259.

15. R. De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298. ACM, 1959.
16. M. Eberl. The number of comparisons in quicksort. *Archive of Formal Proofs*, Mar. 2017. http://isa-afp.org/entries/Quick_Sort_Cost.html, Formal proof development.
17. M. Eberl. Proving divide and conquer complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017.
18. M. Eberl, M. W. Haslbeck, and T. Nipkow. Verified analysis of random binary tree structures. In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pages 196–214. Springer, 2018.
19. J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In *ESOP*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.
20. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.
21. M. L. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
22. K. Germane and M. Might. Deletion: The curse of the red-black tree. *J. Functional Programming*, 24(4):423–433, 2014.
23. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21, 1978.
24. F. Haftmann. *Code generation from Isabelle/HOL theories.* http://isabelle.in.tum.de/doc/codegen.pdf.
25. F. Haftmann. *Haskell-style type classes with Isabelle/Isar.* http://isabelle.in.tum.de/doc/classes.pdf.
26. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
27. Haskell website. https://www.haskell.org.
28. R. Hinze. On constructing 2-3 trees. *J. Funct. Program.*, 28:e19, 2018.
29. C. M. Hoffmann and M. J. O'Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1):83–112, 1982.
30. R. Hood and R. Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981.
31. R. R. Hoogerwoord. A logarithmic implementation of flexible arrays. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *LNCS*, pages 191–207. Springer, 1992.
32. D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1101, 1952.
33. J. Iacono. Improved upper bounds for pairing heaps. In M. M. Halldórsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *LNCS*, pages 32–45. Springer, 2000.
34. J. Iacono and M. V. Yagnatinsky. A linear potential function for pairing heaps. In T. H. Chan, M. Li, and L. Wang, editors, *Combinatorial Optimization and Applications — 10th International Conference, COCOA 2016*, volume 10043 of *LNCS*, pages 489–504. Springer, 2016.

35. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 2nd edition, 1990.
36. S. Kahrs. Red black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.
37. A. Kaldewaij and B. Schoenmakers. The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, 37:265–271, 1991.
38. ACM Paris Kanellakis Theory and Practice Award. https://awards.acm.org/kanellakis.
39. R. M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):11361150, Nov. 1994.
40. D. J. King. Functional binomial queues. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 141–150. Springer, 1994.
41. D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.
42. D. E. Knuth. *The Art of Computer Programming, vol. 1: Fundamental Algorithms*. Addison–Wesley, 3rd edition, 1997.
43. A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/doc/functions.pdf.
44. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning,IJCAR 2006*, volume 4130 of *LNCS*, pages 589–603. Springer, 2006.
45. P. Lammich and T. Nipkow. Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.
46. D. H. Larkin, S. Sen, and R. E. Tarjan. A back-to-basics empirical study of priority queues. In C. C. McGeoch and U. Meyer, editors, *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014*, pages 61–72. SIAM, 2014.
47. T. Leighton. Notes on better master theorems for divide-and-conquer recurrences (MIT lecture notes). 1996.
48. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.
49. R. Meis, F. Nielsen, and P. Lammich. Binomial heaps and skew binomial heaps. *Archive of Formal Proofs*, Oct. 2010. http://isa-afp.org/entries/Binomial-Heaps.html, Formal proof development.
50. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
51. P. Müller. The binomial heap verification challenge in Viper. In P. Müller and I. Schaefer, editors, *Principled Software Development*, pages 203–219. Springer, 2018.
52. T. Nipkow. *Programming and Proving in Isabelle/HOL*. http://isabelle.in.tum.de/doc/prog-prove.pdf.

53. T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 310–324. Springer, 2015.
54. T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.
55. T. Nipkow and H. Brinkop. Amortized complexity verified. *J. Autom. Reasoning*, 62(3):367–391, 2019.
56. T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. 298 pp. http://concrete-semantics.org.
57. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
58. T. Nipkow and T. Sewell. Proof pearl: Braun trees. In J. Blanchette and C. Hritcu, editors, *Certified Programs and Proofs, CPP 2020*, pages 18–31. ACM, 2020.
59. T. Nipkow and D. Somogyi. Optimal binary search trees. *Archive of Formal Proofs*, May 2018. https://isa-afp.org/entries/Optimal_BST.html, Formal proof development.
60. Ocaml website. https://ocaml.org.
61. C. Okasaki. Three algorithms on Braun trees. *J. Functional Programming*, 7(6):661–666, 1997.
62. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
63. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
64. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
65. S. Pettie. Towards a final analysis of pairing heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS*, pages 174–183. IEEE Computer Society, 2005.
66. C. Reade. Balanced trees with removals: An exercise in rewriting and proof. *Sci. Comput. Program.*, 18(2):181–204, 1992.
67. M. Rem and W. Braun. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology, 1983.
68. S. Sahni. Leftist trees. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.
69. B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.
70. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
71. D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.
72. R. E. Tarjan. Amortized complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.
73. L. Théry. Formalising Huffman's algorithm. Technical Report TRCS 034, Department of Informatics, University of L'Aquila, 2004.

74. F. Turbak. CS230 Handouts — Spring 2007, 2007. http://cs.wellesley.edu/~cs230/spring07/handouts.html.

75. J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978.

76. M. Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss2002020117092.

77. J. Williams. Algorithm 232 — Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

78. S. Wimmer, S. Hu, and T. Nipkow. Monadification, memoization and dynamic programming. *Archive of Formal Proofs*, May 2018. https://isa-afp.org/entries/Monad_Memo_DP.html, Formal proof development.

79. S. Wimmer, S. Hu, and T. Nipkow. Verified memoization and dynamic programming. In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *Lecture Notes in Computer Science*, pages 579–596. Springer, 2018.

80. F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *STOC*, pages 429–435. ACM, 1980.

81. B. Zhan. Efficient verification of imperative programs using auto2. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018*, volume 10805 of *LNCS*, pages 23–40. Springer, 2018.

Index