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

Functional Algorithms, Verified!

July 28, 2021
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. The structured nature of Isabelle proofs permits even novices to browse them and follow the high-level arguments.

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.

The main strength of Isabelle and other proof assistants is their trustworthiness: all definitions, lemma statements, and inferences are checked. Beyond trustworthiness, formal proofs can also clarify arguments, by exposing and
explaining difficult steps. Most Isabelle users will confirm that their pen-and-paper proofs have become more lucid, and more correct, after they subjected themselves to the discipline of formal proof.

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

1 Basics ............................................................... 7

## Part I Sorting and Selection

2 Sorting .............................................................. 19
3 Selection ............................................................ 35

## Part II Search Trees

4 Binary Trees.................................................... 49
5 Binary Search Trees ........................................... 61
6 Abstract Data Types ........................................... 75
7 2-3 Trees .......................................................... 81
8 Red-Black Trees ................................................ 91
9 AVL Trees .......................................................... 101
10 Beyond Insert and Delete: $\cup$, $\cap$ and $-$ .............. 113
11 Arrays via Braun Trees ........................................ 121
12 Tries ................................................................. 143
13 Huffman’s Algorithm ......................................... 151
## Part III Priority Queues

14 Priority Queues ........................................ 167
15 Leftist Heaps ........................................... 171
16 Priority Queues via Braun Trees ..................... 175
17 Binomial Heaps ......................................... 179

## Part IV Advanced Design and Analysis Techniques

18 Dynamic Programming ................................. 191
19 Amortized Analysis .................................... 213
20 Queues .................................................... 217
21 Splay Trees ............................................. 229
22 Skew Heaps ............................................. 237
23 Pairing Heaps .......................................... 241

## Part V Appendix

A List Library ............................................. 251
B Time Functions ......................................... 255
C Notation ................................................. 263
References ................................................. 269
Index ......................................................... 275
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 [44].)

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, 45, 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 list. The notation \( t : \tau \) means that term \( t \) has type \( \tau \). 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 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 list of lists whose elements are of type ‘a is a recursive data type:

\[
\text{datatype ‘a list} = \text{Nil} \mid \text{Cons ‘a (‘a list)}
\]

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

\[
\begin{align*}
[\] & \equiv \text{Nil} \\
x \# xs & \equiv \text{Cons x xs} \\
[x_1, \ldots, x_n] & \equiv x_1 \# \ldots \# x_n \# []
\end{align*}
\]

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 \(|xs|\).
Type `'a option`

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

```markdown
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: `fst (a, b) = a` and `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:

```
fib (n + 2) = fib (n + 1) + fib n
```

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

```
f (x # (y # zs #: ys)) = ys @ zs
```

introduces a variables `ys` on the left that stands for `y # zs` and can be referred to on the right. Logically it is just an abbreviation of

```
f (x # y # zs) = (let ys = y # zs in ys @ zs)
```

although it is suggestive of a more efficient interpretation. The general format is `pattern =: 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 `int :: nat ⇒ int` and `real :: nat ⇒ 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) :: real`, where `m`, `n :: nat`, is mathematically unambiguous because `real (m + n) = real m + real n`. On the other hand, `(m - n) :: real` is ambiguous because `real (m - n) ≠ real m - real n` because `(0 :: nat) - n = 0`. In some cases we can also drop coercions that are not inclusions, e.g. `nat :: int ⇒ nat`, which coerces negative integers to 0: if we know that `i ≥ 0` then we can drop the `nat` in `nat i`.

We prefer type `nat` over type `real` for ease of (Isabelle) proof. For example, for `m`, `n :: nat` we prefer `m ≤ 2^n` over `lg m ≤ n`. 
1.2.3 Multisets

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>a multiset</code></td>
</tr>
<tr>
<td><code>∈_a</code></td>
<td><code>a ⇒ 'a multiset ⇒ bool</code></td>
</tr>
<tr>
<td><code>add_mset</code></td>
<td><code>a ⇒ 'a multiset ⇒ 'a multiset</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>a multiset ⇒ 'a multiset ⇒ 'a multiset</code></td>
</tr>
<tr>
<td><code>size</code></td>
<td><code>a multiset ⇒ nat</code></td>
</tr>
<tr>
<td><code>mset</code></td>
<td><code>a list ⇒ 'a multiset</code></td>
</tr>
<tr>
<td><code>set_mset</code></td>
<td><code>a multiset ⇒ 'a set</code></td>
</tr>
<tr>
<td><code>image_mset</code></td>
<td><code>('a ⇒ 'b) ⇒ 'a multiset ⇒ 'b multiset</code></td>
</tr>
<tr>
<td><code>filter_mset</code></td>
<td><code>('a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset</code></td>
</tr>
<tr>
<td><code>sum_mset</code></td>
<td><code>a multiset ⇒ 'a</code></td>
</tr>
</tbody>
</table>

Their meaning: `{}` is the empty multiset; `∈_a` is the element test; `add_mset` adds an element to a multiset; `+` is the sum 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).

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

```
{ x ∈_a M | P x } ≡ filter_mset P M
{ f x | x ∈_a M } ≡ image_mset f M
∑_x M ≡ sum_mset M
∑_{x∈_a M} f x ≡ sum_mset (image_mset f M)
```

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 ⇒ 'b` comes with a predefined pointwise update operation with its own notation:
  
  `f(a := b) = (λx. if x = a then b 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: $\Rightarrow$ is printed as $\longrightarrow$ and $P \Rightarrow Q \Rightarrow R$ is printed as $P \land Q \longrightarrow R$.
• Multiplication is printed as $x \cdot y$.
• Exponentiation is uniformly printed as $x^y$.
• We sweep under the carpet that type $\textbf{nat}$ is defined as a recursive data type: \texttt{datatype nat = 0 | Suc nat}. In particular, constructor $\texttt{Suc}$ is hidden: $\texttt{Suc}^k 0$ is printed as $k$ and $\texttt{Suc}^k n$ (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

\[
(\emptyset) :: 'a list \Rightarrow 'a list \Rightarrow 'a list
\]

\[
[] @ ys = ys
\]

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

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

\[
(x @ xs @ ys) @ zs = x @ (xs @ ys) @ zs \quad \text{by definition of $(\emptyset)$}
\]

\[
= x @ xs @ ys @ zs \quad \text{by IH}
\]
Note that \textbf{IH} stands for \textit{Induction Hypothesis}, in this case \((xs @ ys) @ zs = xs @ ys @ zs\).

\subsection*{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 \texttt{gcd :: nat \Rightarrow nat \Rightarrow nat}

\[
gcd m n = (\text{if } n = 0 \text{ then } m \text{ else } gcd n (m \mod n))
\]
gives rise to the following induction schema:

If \((n \neq 0 \rightarrow P n (m \mod n)) \rightarrow P m n\) 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

\[
P r_1 \land \ldots \land P r_n \rightarrow P e
\]

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 \texttt{if} and \texttt{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 \texttt{gcd} example. If the defining equations for \(f\) overlap, the above proof obligations are stronger than necessary.

\subsection*{1.5 Running Time}

Our approach to reasoning about the \textbf{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 \textbf{(running) time function} \(T_f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \text{nat}\) 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 compute the number of function calls. The unusual notation $\mathcal{E}[\cdot]$ and $\mathcal{T}[\cdot]$ emphasizes that they are not functions in the logic.

\[
\begin{align*}
\mathcal{E}[f \; p_1 \ldots \; p_n = e] &= (T_f \; p_1 \ldots \; p_n = \mathcal{T}[e] + 1) \\
\mathcal{T}[g \; e_1 \ldots \; e_k] &= \mathcal{T}[e_1] + \ldots + \mathcal{T}[e_k] + T_g \; e_1 \ldots \; e_k \\
\end{align*}
\]  

(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 $T_g = 1$ 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:

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

- let expressions are also special:

\[
\mathcal{T}[\text{let } x = e_1 \text{ in } e_2] = \mathcal{T}[e_1] + (\lambda x. \mathcal{T}[e_2]) \; e_1
\]

Note that $\mathcal{T}[e_2]$ must be evaluated completely before $(\lambda x. \mathcal{T}[e_2]) \; e_1$ is $\beta$-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 $T_{\text{append}} :: \text{'a list} \Rightarrow \text{'a list} \Rightarrow \text{nat}$ are easily derived. The first equation translates like this:

\[
\begin{align*}
\mathcal{E}[\cdot @ ys = ys] &= (T_{\text{append}} \; [] \; ys = \mathcal{T}[ys] + 1) \\
&= (T_{\text{append}} \; [] \; ys = 2)
\end{align*}
\]

The right-hand side of the second equation translates like this:
Thus the two defining equations for $T_{\text{append}}$ are:

$T_{\text{append}} [] ys = 2$

$T_{\text{append}} (x \# xs) ys = T_{\text{append}} xs ys + 5$

The constants 2 and 5 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 $T_{\text{append}}$ is this:

$T_{\text{append}} [] ys = 1$

$T_{\text{append}} (x \# xs) ys = T_{\text{append}} xs ys + 1$

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 $\text{rev}$ is defined in Appendix A. This is the corresponding canonical time function:

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

$T_{\text{rev}} [] = 1$

$T_{\text{rev}} (x \# xs) = T_{\text{rev}} xs + T_{\text{append}} (\text{rev} \; xs) [x] + 1$
The canonical $T_{\text{append}}$ was shown before.

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

$$T_{\text{rev}} \, xs \leq (|xs| + 1)^2$$

We will frequently prove upper bounds only.

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

\begin{verbatim}
itrev :: 'a list ⇒ 'a list ⇒ 'a list
itrev [] ys = ys
itrev (x # xs) ys = itrev xs (x # ys)
\end{verbatim}

\begin{verbatim}
T_{itrev} :: 'a list ⇒ 'a list ⇒ nat
T_{itrev} [] _ = 1
T_{itrev} (x # xs) ys = T_{itrev} xs (x # ys) + 1
\end{verbatim}

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 $E$ and $T$, 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. \( T \)) 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
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 ≤ and < defined on the underlying type.

Sortedness of lists is defined as follows:

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

That is, every element is ≤ 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 (≤) :: 'a ⇒ 'a ⇒ bool is defined and (≤) is a linear order, i.e. the following properties are satisfied:

- **reflexivity:** \( x \leq x \)
- **transitivity:** \( x \leq y \land y \leq z \rightarrow x \leq z \)
- **antisymmetry:** \( a \leq b \land b \leq a \rightarrow a = b \)
- **linearity/totality:** \( x \leq y \lor y \leq x \)

Moreover, the binary predicate (<) must satisfy

\( x < y \iff x \leq y \land x \neq y \).

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 \( \text{sort} :: 'a \Rightarrow 'a \times 'a \Rightarrow 'a \times 'a \Rightarrow 'a \) must obviously satisfy the following property:

\[
\text{sorted} (\text{sort} \, xs)
\]

However, this is not enough — otherwise, \( \text{wrong}_\text{sort} \, xs = [] \) 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 \( \text{sort} \) must satisfy is

\[
\text{mset} (\text{sort} \, xs) = \text{mset} \, xs
\]

where function \( \text{mset} \) 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 \( \text{isort} \) with the help of the auxiliary function \( \text{insort} \) that inserts a single element into an already sorted list.

\[
\text{insort} :: 'a \Rightarrow 'a \times 'a \Rightarrow 'a \times 'a \Rightarrow 'a
\]

\[
\text{insort} \, x \, [] = [x]
\]

\[
\text{insort} \, x \, (y \# ys) = (\text{if} \, x \leq y \, \text{then} \, x \# y \# ys \, \text{else} \, y \# \text{insort} \, x \, ys)
\]

\[
\text{isort} :: 'a \times 'a \Rightarrow 'a \times 'a \Rightarrow 'a
\]

\[
\text{isort} \, [] = []
\]

\[
\text{isort} \, (x \# zs) = \text{insort} \, x \, (\text{isort} \, zs)
\]

#### 2.2.1 Functional Correctness

We start by proving the preservation of the multiset of elements:
2.2 Insertion Sort

\[
mset (\text{insort} \ x \ xs) = \{x\} + mset \ xs \tag{2.1}
\]
\[
mset (\text{isort} \ xs) = mset \ xs \tag{2.2}
\]

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

Now we turn to sortedness. Because the definition of \text{sorted} involves \text{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:

\[
\text{set} (\text{insort} \ x \ xs) = \{x\} \cup \text{set} \ xs \tag{2.3}
\]

Two inductions prove

\[
\text{sorted} (\text{insort} \ a \ xs) = \text{sorted} \ xs \tag{2.4}
\]
\[
\text{sorted} (\text{isort} \ xs) \tag{2.5}
\]

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):

\[
T_{\text{insort}} : \mathbb{N} \to \mathbb{N}
\]
\[
T_{\text{insort}} \ [] = 1
\]
\[
T_{\text{insort}} \ x \ (y \# ys) = (\text{if } x \leq y \text{ then } 0 \text{ else } T_{\text{insort}} \ xs + 1
\]

\[
T_{\text{isort}} : \mathbb{N} \to \mathbb{N}
\]
\[
T_{\text{isort}} \ [] = 1
\]
\[
T_{\text{isort}} \ x \ # \ xs = T_{\text{isort}} \ xs + T_{\text{insort}} \ x \ (\text{isort} \ xs) + 1
\]

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

**Lemma 2.1.** \(T_{\text{isort}} \ xs \leq (|xs| + 1)^2\)

**Proof.** The following properties are proved by induction on \(xs\):

\[
T_{\text{insort}} \ x \ xs \leq |xs| + 1 \tag{2.6}
\]
\[
|\text{insort} \ x \ xs| = |xs| + 1 \tag{2.7}
\]
\[
|\text{isort} \ xs| = |xs| \tag{2.8}
\]

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


2.2 Sorting

\[ T_{\text{isort}}(x \notin xs) = T_{\text{isort}}(xs) + T_{\text{insort}}(\text{isort}(xs)) + 1 \]

\[ \leq (|xs| + 1)^2 + T_{\text{insort}}(\text{isort}(xs)) + 1 \text{ \quad \text{by IH}} \]

\[ \leq (|xs| + 1)^2 + |xs| + 1 + 1 \text{ \quad \text{using (2.6) and (2.8)}} \]

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:

\[ (\forall xs. \text{mset}(f(xs)) = \text{mset}(xs)) \land (\forall xs. \text{sorted}(f(xs)) \rightarrow f(xs) = \text{isort}(xs)) \]

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

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:

\[
\text{quicksort} :: \text{'a list} \Rightarrow \text{'a list} \\
\text{quicksort} [] = [] \\
\text{quicksort} (x \notin xs) = \text{quicksort} (\text{filter} (\lambda y. y < x) xs) \rangle @ [x] @ \text{quicksort} (\text{filter} (\lambda y. y \geq x) xs)
\]

2.3.1 Functional Correctness

Preservation of the multiset of elements

\[ \text{mset}(\text{quicksort} xs) = \text{mset} xs \text{ \quad (2.9)} \]
2.3 Quicksort

is proved by computation induction using these lemmas:

\[ mset (\text{filter } P \ x) = \text{filter}_mset P (mset x) \]
\[ (\forall x. P x = (\neg Q x)) \rightarrow \text{filter}_mset P M + \text{filter}_mset Q M = M \]

A second computation induction proves sortedness

\[ \text{sorted} (\text{quicksort } x) \]

using the lemmas

\[ \text{sorted} (x \ # y) \]
\[ = (\text{sorted } x \land \text{sorted } y \land (\forall x \in \text{set } x. \ \forall y \in \text{set } y. \ x \leq y)) \]
\[ \text{set} (\text{quicksort } x) = \text{set } x \]

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 quicksort2 :: 'a list ⇒ 'a list ⇒ 'a list that avoids \((@)\) but accumulates the result in its second parameter via \((#)\) only. Prove quicksort2 \( x \ # \ y = \text{quicksort } x \ # \ y \).

**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:

\[
\text{partition}_3 :: 'a ⇒ 'a list ⇒ 'a list × 'a list × 'a list
\]
\[
\text{partition}_3 \ x \ xs
= (\text{filter } (\lambda y. \ y < x) \ xs, \text{filter } (\lambda y. \ y = x) \ xs,
\text{filter } (\lambda y. \ y > x) \ xs)
\]
\[
\text{quicksort}_3 :: 'a list ⇒ 'a list
\]
\[
\text{quicksort}_3 [] = []
\]
24  2 Sorting

\[
\text{quicksort3} \ (x \ # \ xs) \\
= (\text{let} \ (ls, es, gs) = \text{partition3} \ x \ xs \\
\quad \text{in} \ \text{quicksort3} \ ls \ @ \ x \ # \ es \ @ \ \text{quicksort3} \ gs)
\]

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:

\[
T_{\text{quicksort}} :: \ 'a \ list \ \Rightarrow \ \text{nat} \\
T_{\text{quicksort}} [] = 1 \\
T_{\text{quicksort}} (x \ # \ xs) = T_{\text{quicksort}} (\text{filter} \ (\lambda y. \ y < x) \ xs) + \\
T_{\text{quicksort}} (\text{filter} \ (\lambda y. \ y \geq x) \ xs) + \\
2 \cdot T_{\text{filter}} (\lambda _. \ 1) \ xs + 1
\]

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

\[
sorted \ xs \longrightarrow T_{\text{quicksort}} \ xs = a \cdot |xs|^2 + b \cdot |xs| + c
\]

for suitable values \(a, b, c\).

2. Show that this is the worst-case running time by proving

\[
T_{\text{quicksort}} \ xs \leq a \cdot |xs|^2 + b \cdot |xs| + c
\]

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.

\[
\text{merge} :: \ 'a \ list \ \Rightarrow \ 'a \ list \ \Rightarrow \ 'a \ list \\
\text{merge} [] \ ys = ys
\]
merge \textit{xs} \texttt{[]} = \text{xs}
merge (x \# \textit{xs}) (y \# \textit{ys})
= (\textbf{if} \ x \leq y \ \textbf{then} \ x \# \text{merge} \ \textit{xs} \ (y \# \textit{ys})
\hspace{1em} \textbf{else} \ y \# \text{merge} \ (x \# \textit{xs}) \ \textit{ys})

\textit{msort} :: \texttt{\textquotesingle a list} \Rightarrow \texttt{\textquotesingle a list}
\textit{msort} \ \textit{xs}
= (\textbf{let} \ n = |\textit{xs}|
\hspace{1em} \textbf{in} \ if \ n \leq 1 \ \textbf{then} \ \textit{xs}
\hspace{1em} \textbf{else} \text{merge} \ (\textit{msort} \ (\text{take} \ (n \ \text{div} \ 2) \ \textit{xs}))
\hspace{1em} (\textit{msort} \ (\text{drop} \ (n \ \text{div} \ 2) \ \textit{xs})))

### 2.4.1 Functional Correctness

We start off with multisets and sets of elements:

\begin{equation}
\textit{mset} \ (\text{merge} \ \textit{xs} \ \textit{ys}) = \textit{mset} \ \textit{xs} + \textit{mset} \ \textit{ys} \quad (2.10)
\end{equation}

\begin{equation}
\textit{set} \ (\text{merge} \ \textit{xs} \ \textit{ys}) = \textit{set} \ \textit{xs} \cup \textit{set} \ \textit{ys} \quad (2.11)
\end{equation}

Proposition (2.10) is proved by induction on the computation of \textit{merge} and (2.11) is an easy consequence.

**Lemma 2.2.** \textit{mset} \ (\textit{msort} \ \textit{xs}) = \textit{mset} \ \textit{xs}

**Proof.** The proof is by induction on the computation of \textit{msort}. Let \( n = |\textit{xs}| \).

The base case \((n \leq 1)\) is trivial. Now assume \( n > 1 \) and let \( \textit{ys} = \text{take} \ (n \ \text{div} \ 2) \ \textit{xs} \) and \( \textit{zs} = \text{drop} \ (n \ \text{div} \ 2) \ \textit{xs} \).

\[
\textit{mset} \ (\textit{msort} \ \textit{xs}) = \textit{mset} \ (\textit{msort} \ \textit{ys}) + \textit{mset} \ (\textit{msort} \ \textit{zs}) \quad \text{by (2.10)}
\]

\[
= \textit{mset} \ \textit{ys} + \textit{mset} \ \textit{zs} \quad \text{by IH}
\]

\[
= \textit{mset} \ (\textit{ys} \ @ \ \textit{zs})
\]

\[
= \textit{mset} \ \textit{xs} \quad \square
\]

Now we turn to sortedness. An induction on the computation of \textit{merge}, using (2.11), yields

\[
\textit{sorted} \ (\text{merge} \ \textit{xs} \ \textit{ys}) = (\textit{sorted} \ \textit{xs} \land \textit{sorted} \ \textit{ys}) \quad (2.12)
\]

**Lemma 2.3.** \textit{sorted} \ (\textit{msort} \ \textit{xs})

The proof is an easy induction on the computation of \textit{msort}. The base case \((n \leq 1)\) follows because every list of length \( \leq 1 \) 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:

\[
C_{\text{merge}} :: 'a list \Rightarrow 'a list \Rightarrow \text{nat}
\]

\[
C_{\text{merge}} [] = 0
\]

\[
C_{\text{merge}} [] = 0
\]

\[
C_{\text{merge}} (x \# xs) (y \# ys)
= 1 + \begin{cases} 
\text{if } x \leq y & \text{then } C_{\text{merge}} xs (y \# ys) \text{ else } C_{\text{merge}} (x \# xs) ys
\end{cases}
\]

\[
C_{\text{msort}} :: 'a list \Rightarrow \text{nat}
\]

\[
C_{\text{msort}} xs
= \begin{cases}
    0 & \text{if } n \leq 1 \\
    \text{let } n = |xs|; \\
    \quad ys = \text{take} \ (n \div 2) \ xs; \\
    \quad zs = \text{drop} \ (n \div 2) \ xs \\
    \quad \text{in if } n \leq 1 \text{ then } 0 \text{ else } C_{\text{msort}} ys + C_{\text{msort}} zs + C_{\text{merge}} (\text{msort} \ ys) (\text{msort} \ zs)
\end{cases}
\]

By computation inductions we obtain:

\[|\text{merge} \ xs \ ys| = |xs| + |ys|\] (2.13)

\[|\text{msort} \ xs| = |xs|\] (2.14)

\[C_{\text{merge}} \ xs \ ys \leq |xs| + |ys|\] (2.15)

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

To simplify technicalities, we prove the \(n \cdot \log n\) bound on the number of comparisons in \(\text{msort}\) only for \(n = 2^k\), in which case the bound becomes \(k \cdot 2^k\).

**Lemma 2.4.** \(|zs| = 2^k \implies C_{\text{msort}} \ zs \leq k \cdot 2^k\)

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

\[
C_{\text{msort}} \ zs
= C_{\text{msort}} \ ys + C_{\text{msort}} \ zs + C_{\text{merge}} (\text{msort} \ ys) (\text{msort} \ zs)
\leq C_{\text{msort}} \ ys + C_{\text{msort}} \ zs + |ys| + |zs| \quad \text{using (2.15) and (2.14)}
\leq k \cdot 2^k + k \cdot 2^k + |ys| + |zs| \quad \text{by IH}
= k \cdot 2^k + k \cdot 2^k + |zs|
= (k + 1) \cdot 2^k + 1 \quad \text{by assumption } |zs| = 2^k + 1 \quad \square
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 $msort2$

$$
msort2 \emptyset = \emptyset \\
msort2 [x] = [x] \\
msort2 xs = (\text{let } (ys1, ys2) = halve xs ([], []) \text{ in } merge (msort2 ys1) (msort2 ys2))
$$

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.

$$
merge\_adj :: 'a list list \Rightarrow 'a list list \\
merge\_adj \emptyset = \emptyset \\
merge\_adj [xs] = [xs] \\
merge\_adj (xs # ys # zss) = merge xs ys # merge\_adj zss
$$

$$
merge\_all :: 'a list list \Rightarrow 'a list \\
merge\_all \emptyset = \emptyset \\
merge\_all [xs] = xs \\
merge\_all xss = merge\_all (merge\_adj xss)
$$

$$
msort\_bu :: 'a list \Rightarrow 'a list \\
msort\_bu xs = merge\_all (map (\lambda x. [x]) xs)
$$

Termination of $merge\_all$ relies on the fact that $merge\_adj$ halves the length of the list (rounding up). Computation induction proves

$$
|merge\_adj xs| = (|xs| + 1) \text{ div } 2
$$

2.5.1 Functional Correctness

We introduce the abbreviation $mset\_mset :: 'a list list \Rightarrow 'a multiset$:
These are the key properties of the functions involved:

\[
\begin{align*}
\text{mset\_mset } xss & \equiv \sum_{a}(\text{image\_mset mset (mset } xss)) \\
\text{mset\_mset } (\text{merge\_adj } xss) & = \text{mset\_mset } xss \\
\text{mset } (\text{merge\_all } xss) & = \text{mset\_mset } xss \quad (2.17) \\
\text{mset } (\text{msort\_bu } xs) & = \text{mset } xs \\
\text{Ball (set } xss) \text{ sorted } & \rightarrow \text{Ball (set } (\text{merge\_adj } xss)) \text{ sorted} \\
\text{Ball (set } xss) \text{ sorted } & \rightarrow \text{sorted } (\text{merge\_all } xss) \\
\text{sorted } (\text{msort\_bu } xs) &
\end{align*}
\]

The third and the last property prove functional correctness of \texttt{msort\_bu}. The proof of each proposition may use the preceding proposition and the propositions (2.10) and (2.12). The propositions about \texttt{merge\_adj} and \texttt{merge\_all} are proved by computation inductions.

### 2.5.2 Running Time Analysis

Again, we count only comparisons:

\[
\begin{align*}
\text{C}_{\text{merge\_adj}} & \colon \text{a list list } \Rightarrow \text{nat} \\
\text{C}_{\text{merge\_adj}} [] & = 0 \\
\text{C}_{\text{merge\_adj}} [\_ &] = 0 \\
\text{C}_{\text{merge\_adj}} (xs \# ys \# zss) & = \text{C}_{\text{merge\_adj}} xs ys + \text{C}_{\text{merge\_adj}} zss \\
\text{C}_{\text{merge\_all}} & \colon \text{a list list } \Rightarrow \text{nat} \\
\text{C}_{\text{merge\_all}} [] & = 0 \\
\text{C}_{\text{merge\_all}} [zs] & = 0 \\
\text{C}_{\text{merge\_all}} xss = \text{C}_{\text{merge\_adj}} xss + \text{C}_{\text{merge\_all}} (\text{merge\_adj } xss) \\
\text{C}_{\text{msort\_bu}} & \colon \text{a list } \Rightarrow \text{nat} \\
\text{C}_{\text{msort\_bu}} xs & = \text{C}_{\text{merge\_all}} (\text{map } (\lambda x. [x]) xs)
\end{align*}
\]

By simple computation inductions we obtain:

\[
\begin{align*}
\text{even } |xss| \land (\forall xs \in \text{set } xss. |xs| = m) & \rightarrow \\
(\forall xs \in \text{set } (\text{merge\_adj } xss). |xs| = 2 \cdot m) & \quad (2.19) \\
(\forall xs \in \text{set } xss. |xs| = m) & \rightarrow \text{C}_{\text{merge\_adj}} xss \leq m \cdot |xss| \quad (2.20)
\end{align*}
\]

using (2.13) for (2.19) and (2.15) for (2.20).
Lemma 2.5. \((\forall \text{xs} \in \text{set} \text{ xss}. |\text{xs}| = m) \land |\text{xss}| = 2^k \rightarrow C_{\text{merge\_all} \text{ xss}} \leq m \cdot k \cdot 2^k\)

Proof: by induction on the computation of \(C_{\text{merge\_all} \text{ xss}}\). We concentrate on the nontrivial recursive case arising from the third equation. We assume \(|\text{xss}| > 1\), \(\forall \text{xs} \in \text{set} \text{ xss}. |\text{xs}| = m \land |\text{xss}| = 2^k\). Clearly \(k \geq 1\) and thus even \(|\text{xss}|\). Thus (2.19) implies \(\forall \text{xs} \in \text{set} (\text{merge\_adj} \text{ xss}). |\text{xs}| = 2 \cdot m\). Also note

\[
|\text{merge\_adj} \text{ xss}|
\]

\[
= (|\text{xss}| + 1) \div 2 & \text{using (2.16)}
\]

\[
= 2^k - 1 & \text{using } |\text{xss}| = 2^k \text{ and } k \geq 1 \text{ by arithmetic}
\]

Let \(\text{yss} = \text{merge\_adj} \text{ xss}\). We can now prove the lemma:

\[
C_{\text{merge\_all} \text{ xss}} = C_{\text{merge\_adj} \text{ xss}} + C_{\text{merge\_all} \text{ yss}}
\]

\[
\leq m \cdot 2^k + C_{\text{merge\_all} \text{ yss}} & \text{using } |\text{xss}| = 2^k \text{ and (2.20)}
\]

\[
\leq m \cdot 2^k + 2 \cdot m \cdot (k - 1) \cdot 2^k - 1 & \text{by IH using } \forall \text{xs} \in \text{set} \text{ yss}. |\text{xs}| = 2 \cdot m \land |\text{yss}| = 2^k - 1
\]

\[
= m \cdot k \cdot 2^k & \text{by IH using (2.19)}
\]

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

Corollary 2.6. \(|\text{xs}| = 2^k \rightarrow C_{\text{msort\_bu} \text{ xss}} \leq k \cdot 2^k\)

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 \([1, 2, 3, 4, 5]\), \(\text{msort\_bu}\) 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 \(\text{runs}\)) and only then begins merging. In the above example we would get \(\text{merge\_all} \ [[1, 2, 3, 4, 5]]\), which returns immediately with the result \([1, 2, 3, 4, 5]\). Assuming that obtaining \(\text{runs}\) is of linear complexity, this yields a best-case performance that is linear in the number of list elements.

Function \(\text{runs}\) computes the initial list of lists; it is defined mutually recursively with \(\text{asc}\) and \(\text{desc}\), which gather ascending and descending runs in accumulating parameters:
Function \textit{desc} needs to reverse the descending run it collects. Therefore a natural choice for the type of its accumulator \textit{as} is \textit{list}, since recursively prepending elements (using \((\#)\)) ultimately yields a reversed list.

Function \textit{asc} collects an ascending run and is slightly more complicated than \textit{desc}. If we used lists, we could accumulate the elements similarly to \textit{desc} but using \textit{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 \textit{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 \textit{‘a list \Rightarrow ‘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 \textit{runs}, \textit{asc}, and \textit{desc}, whenever we prove a property of \textit{runs}, we simultaneously have to prove suitable properties of \textit{asc} and \textit{desc} using mutual induction.

Natural merge sort is the composition of \textit{merge_all} and \textit{runs}:
2.6 Natural Merge Sort

\[ \text{nmsort} :: 'a \text{ list} \Rightarrow 'a \text{ list} \]
\[ \text{nmsort} \; xs = \text{merge} \; \text{all} \; (\text{runs} \; xs) \]

2.6.1 Functional Correctness

We have

\[
\forall \; xs \; ys. \; f \; (xs \; @ \; ys) = f \; xs \; @ \; ys \implies \\
\text{mset}_m \text{set} \; (\text{asc} \; x \; f \; ys) = \{x\} + \text{mset} \; (f \; []) + \text{mset} \; ys \quad (2.21) \\
\text{mset}_m \text{set} \; (\text{desc} \; x \; xs \; ys) = \{x\} + \text{mset} \; xs + \text{mset} \; ys \quad (2.22) \\
\text{mset}_m \text{set} \; (\text{runs} \; xs) = \text{mset} \; xs \quad (2.23) \\
\text{mset} \; (\text{nmsort} \; xs) = \text{mset} \; xs \quad (2.24)
\]

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

\[
\text{Ball} \; (\text{set} \; (\text{runs} \; xs)) \; \text{sorted} \quad (2.25) \\
\text{sorted} \; (\text{nmsort} \; xs) \quad (2.26)
\]

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:

\[
\text{C}_{\text{runs}} :: 'a \text{ list} \Rightarrow \text{nat} \]
\[
\text{C}_{\text{runs}} \; (a \; @ \; b \; @ \; xs) = 1 + (\text{if } b < a \text{ then } \text{C}_{\text{desc}} \; b \; xs \text{ else } \text{C}_{\text{asc}} \; b \; xs) \\
\text{C}_{\text{runs}} \; [] = 0 \\
\text{C}_{\text{runs}} \; [\_] = 0
\]

\[
\text{C}_{\text{asc}} :: 'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \]
\[
\text{C}_{\text{asc}} \; a \; (b \; @ \; bs) = 1 + (\text{if } \neg \; b < a \text{ then } \text{C}_{\text{asc}} \; b \; bs \text{ else } \text{C}_{\text{runs}} \; (b \; @ \; bs)) \\
\text{C}_{\text{asc}} \; [\_] = 0
\]

\[
\text{C}_{\text{desc}} :: 'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \]
\[
\text{C}_{\text{desc}} \; a \; (b \; @ \; bs) = 1 + (\text{if } b < a \text{ then } \text{C}_{\text{desc}} \; b \; bs \text{ else } \text{C}_{\text{runs}} \; (b \; @ \; bs)) \\
\text{C}_{\text{desc}} \; [\_] = 0
\]
Again note the mutually recursive definitions of \( \text{C} \text{runs} \), \( \text{C} \text{asc} \), and \( \text{C} \text{desc} \). Hence the remark on proofs about \( \text{runs} \) also applies to proofs about \( \text{C} \text{runs} \).

Before talking about \( \text{C} \text{nsort} \), we need a variant of Lemma 2.5 that also works for lists whose lengths are not powers of two (since the result of \( \text{runs} \) 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):

\[
\begin{align*}
\text{C} \text{merge}_\text{adj} \ xss & \leq |\text{concat} \ xss| \quad (2.27) \\
|\text{concat} (\text{merge}_\text{adj} \ xss)| & = |\text{concat} \ xss| \quad (2.28)
\end{align*}
\]

**Lemma 2.7.** \( \text{C} \text{merge}_\text{all} \ xss \leq |\text{concat} \ xss| \cdot |\lg | xss | | \)

**Proof** by induction on the computation of \( \text{C} \text{merge}_\text{all} \). We concentrate on the nontrivial recursive case arising from the third equation. It follows that \( xss \) is of the form \( xs \neq ys \neq xss \). Further note that for all \( n :: \text{nat} \):

\[
2 \leq n \rightarrow |\lg n| = |\lg ((n - 1) \div 2 + 1)| + 1 \quad (2.29)
\]

Now, let \( m = |\text{concat} \ xss| \). Then we have

\[
\begin{align*}
\text{C} \text{merge}_\text{all} \ xss & = \text{C} \text{merge}_\text{adj} \ xss + \text{C} \text{merge}_\text{all} (\text{merge}_\text{adj} \ xss) \\
& \leq m + \text{C} \text{merge}_\text{all} (\text{merge}_\text{adj} \ xss) & \text{using (2.27)} \\
& \leq m + |\text{concat} (\text{merge}_\text{adj} \ xss)| \cdot |\lg |\text{merge}_\text{adj} \ xss|| & \text{by IH} \\
& = m + m \cdot |\lg |\text{merge}_\text{adj} \ xss|| & \text{by (2.28)} \\
& = m + m \cdot |\lg ((|xss| + 1) \div 2)| & \text{by (2.16)} \\
& = m + m \cdot |\lg ((|xss| + 1) \div 2 + 1)| \\
& = m \cdot (|\lg ((|xss| + 1) \div 2 + 1)| + 1) & \text{by (2.29)} \\
& = m \cdot |\lg |xss|| & \square
\end{align*}
\]

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

\[
(\forall xs \ ys. \ f (xs \@ ys) = f xs \@ ys) \rightarrow \\
|\text{concat} (\text{asc} a f ys)| = 1 + |f || + |ys|, \\
|\text{concat} (\text{desc} a xs ys)| = 1 + |xs| + |ys|, \\
|\text{concat} (\text{runs} xs)| = |xs| \quad (2.30) \\
(\forall xs \ ys. \ f (xs \@ ys) = f xs \@ ys) \rightarrow |\text{asc} a f ys| \leq 1 + |ys|,
\]
2.7 Stability

\[ |\text{desc } a \text{ } xs \text{ } ys| \leq 1 + |ys|, |\text{runs } xs| \leq |xs| \]  
\[ C_{\text{asc}} a \text{ } ys \leq |ys|, C_{\text{desc}} a \text{ } ys \leq |ys|, C_{\text{runs}} xs \leq |xs| - 1 \]  

(2.31)

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

**Lemma 2.8.** \(|xs| = n \rightarrow C_{\text{nmsort}} xs \leq n + n \cdot \lfloor \lg n \rfloor\)

**Proof.** Note that

\[ C_{\text{merge\_all}} (\text{runs } xs) \leq n \cdot \lfloor \lg n \rfloor \] as shown by the derivation:

\[
C_{\text{merge\_all}} (\text{runs } xs) \\
\leq |\text{concat } (\text{runs } xs)| \cdot \lfloor \lg |\text{runs } xs| \rfloor \text{ by Lemma 2.7 with } xss = \text{runs } xs \\
\leq n \cdot \lfloor \lg |\text{runs } xs| \rfloor \text{ by (2.30)} \\
\leq n \cdot \lfloor \lg n \rfloor \text{ by (2.31)}
\]

We conclude the proof by:

\[ C_{\text{nmsort}} xs = C_{\text{runs}} xs + C_{\text{merge\_all}} (\text{runs } xs) \\
\leq n + n \cdot \lfloor \lg n \rfloor \text{ using (2.32) and (\star)} \quad \Box \]

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 \( f :: \mathcal{A} \Rightarrow \mathcal{k} \) that maps an element to its key and that the keys \( \mathcal{k} \) are linearly ordered, not the elements. This is the specification of a sorting function \( \text{sort\_key} \):

\[ \text{mset } (\text{sort\_key } f \text{ } xs) = \text{mset } xs \]

\[ \text{sorted } (\text{map } f \text{ } (\text{sort\_key } f \text{ } xs)) \]

Assuming (for simplicity) we are sorting pairs of keys and some attached information, stability means that sorting \([(2, x), (1, z), (1, y)]\) yields \([(1, k), (1, y), (2, x)]\) 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 \( xs \):

\[ \text{filter } (\lambda y. \text{ } f \text{ } y = k) \text{ } (\text{sort\_key } f \text{ } xs) = \text{filter } (\lambda y. \text{ } f \text{ } y = k) \text{ } xs \]

We will now define insertion sort adapted to keys and verify its correctness and stability.
The proofs of the functional correctness properties

\[
\text{mset} \ (\text{isort\_key} \ f \ xs) = \text{mset} \ xs \\
\text{sorted} \ (\text{map} \ f \ (\text{isort\_key} \ f \ xs))
\]  

are completely analogous to their counterparts for plain isort.

The proof of stability uses three auxiliary properties:

\[
(\forall x \in \text{set} \ xs. \ f \ a \leq f \ x) \rightarrow \text{isort\_key} \ f \ a \ xs = a \# xs \\
\neg P \ x \rightarrow \text{filter} \ P \ (\text{isort\_key} \ f \ xs) = \text{filter} \ P \ xs \\
\text{sorted} \ (\text{map} \ f \ xs) \land P \ x \rightarrow \\
\text{filter} \ P \ (\text{isort\_key} \ f \ x \ xs) = \text{isort\_key} \ f \ x \ (\text{filter} \ P \ xs)
\]

The first one is proved by a case analysis on \(xs\). The other two are proved by induction on \(xs\), using (2.34) in the proof of (2.36).

**Lemma 2.9 (Stability of isort\_key).**

\[
\text{filter} \ (\lambda y. \ f \ y = k) \ (\text{isort\_key} \ f \ xs) = \text{filter} \ (\lambda y. \ f \ y = k) \ xs
\]

**Proof.** The proof is by induction on \(xs\). The base case is trivial. In the induction step we consider the list \(a \# xs\) and perform a case analysis. If \(f \ a \neq k\) the claim follows by IH using (2.35). Now assume \(f \ a = k\):

\[
\text{filter} \ (\lambda y. \ f \ y = k) \ (\text{isort\_key} \ f \ (a \# xs)) \\
= \text{filter} \ (\lambda y. \ f \ y = k) \ (\text{isort\_key} \ f \ a \ (\text{isort\_key} \ f \ xs)) \\
= \text{isort\_key} \ f \ a \ (\text{filter} \ (\lambda y. \ f \ y = k) \ (\text{isort\_key} \ f \ xs))
\]

using \(f \ a = k\), (2.36), (2.33) by IH

\[
= \text{isort\_key} \ f \ a \ (\text{filter} \ (\lambda y. \ f \ y = k) \ xs) \\
= a \# \text{filter} \ (\lambda y. \ f \ y = k) \ xs \\
= \text{filter} \ (\lambda y. \ f \ y = k) \ (a \# xs)
\]

using \(f \ a = k\) and (2.34) using \(f \ a = k\)

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

$$k < |xs| \rightarrow |\{ y \inxs | y < \text{select } k \; xs \} | \leq k$$
$$\land |\{ y \inxs | y > \text{select } k \; xs \} | < |xs| - k$$

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

$$|\{ z \inxs | z < x \} | \leq k \quad |\{ z \inxs | z > x \} | < |xs| - k$$
$$|\{ z \inxs | z < y \} | \leq k \quad |\{ z \inxs | z > y \} | < |xs| - k$$

then $x = y$.

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

$$\{ z \inxs | z \leq x \} \subseteq \{ z \inxs | z < y \}$$

(3.2)

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

$$|xs| = |\{ z \inxs | z \leq x \} | + |\{ z \inxs | z > x \} |$$
$$\leq |\{ z \inxs | z < y \} | + |\{ z \inxs | z > x \} |$$
$$< k + (|xs| - k)$$
$$= |xs|$$

\[ \square \]
An equivalent, more concrete definition is the following:

\[
\text{select} :: \text{nat} \Rightarrow \text{`a list} \Rightarrow \text{`a} \\
\text{select } k \text{ xs} = \text{sort xs} ! k
\]  

Equation (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 \( ys \):

\[
\{ x \in_w \text{mset } ys \mid x < ys ! k \} \subseteq_w \text{mset } (\text{take } k \ ys) \{ x \in_w \text{mset } ys \mid x > ys ! k \} \\
\{ x \in_w \text{mset } ys \mid x > ys ! k \} \subseteq_w \text{mset } (\text{drop } (k + 1) \ ys)
\]

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

\[
|\{ x \in_w \text{mset } ys \mid x < ys ! k \}| \leq k |\{ x \in_w \text{mset } ys \mid x > ys ! k \}| < |ys| - k
\]

By setting \( ys := \text{sort } ys \) here, we have

\[
k \geq |\{ x \in_w \text{mset } (\text{sort } xs) \mid x < \text{sort } xs ! k \}|
\]

\[
= |\{ x \in_w \text{mset } xs \mid x < \text{sort } xs ! k \}| \quad \text{using } \text{mset } (\text{sort } xs) = \text{mset } xs
\]

\[
= |\{ x \in_w \text{mset } xs \mid x < \text{select } k \text{ xs} \}| \quad \text{using } (3.3)
\]

and analogously for the elements greater than \( \text{select } k \text{ xs} \).

We will frequently need two more important lemmas about \( \text{sort} \) and \( \text{select} \), namely that they are invariant under permutation of the input list:

\[
\text{mset } xs = \text{mset } ys \quad \rightarrow \quad \text{sort } xs = \text{sort } ys (3.4)
\]

\[
\text{mset } xs = \text{mset } ys \quad \rightarrow \quad \text{select } k \text{ xs} = \text{select } k \text{ ys} \quad (3.5)
\]

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

The definition using \( \text{sort } xs ! k \) 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 \( \text{select} \ 0 \ xs \), i.e. the minimum. Implement a linear-time function \( \text{select0} \) such that
\[
\text{select0} \ [\ ] = \text{select0} \ xs = \text{select} \ 0 \ xs
\]
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 \( \text{select0} \) algorithm be modified to obtain an analogous algorithm \( \text{select1} \) such that
\[
|xs| > 1 \rightarrow \text{select1} \ xs = \text{select} \ 1 \ xs
\]
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 \( \text{select\_fixed} \) that fulfils
\[
k < |xs| \rightarrow \text{select\_fixed} \ k \ xs = \text{select} \ k \ xs
\]
The algorithm must be tail-recursive with running time \( O(\k n) \) and traverse the list exactly once.
Hint: one approach is to first define a function \( \text{take\_sort} \) that computes \( \text{take} \ m \ (\text{sort} \ xs) \) in time \( O(mn) \).
2. Prove your \( \text{select1} \) from the previous exercise correct by showing that it is equivalent to \( \text{select\_fixed} \ 1 \).
3. Define a suitable time function for your \( \text{select\_fixed} \). Prove that this time function is \( O(\k n) \), i.e. that
\[
T_{\text{select\_fixed}} \ k \ xs \leq C_1 \cdot k \cdot |xs| + C_2 \cdot |xs| + C_3 \cdot k + C_4
\]
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 \( \text{select} \ k \ xs \) 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:
\[
k < |xs| \land i < |xs| \land \text{distinct} \ xs \rightarrow
(\exists z. \text{distinct} \ (xs[i := z]) \land \text{select} \ k \ (xs[i := z]) \neq \text{select} \ k \ xs)
\]
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 \( \lambda k. \text{select} \ k \ xs \) 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 \texttt{partition3} 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 \( select \):

\[
\begin{align*}
  k < |ys| + |zs| \land (\forall y \in \text{set } ys. \forall z \in \text{set } zs. \ y \leq z) \to \\
  select \ k \ (ys @ zs) \\
  = (\text{if } k < |ys| \text{ then } select \ k \ ys \text{ else } select \ (k - |ys|) \ zs) \quad (3.6)
\end{align*}
\]

\textit{Proof.} The assumptions imply that \( \text{sort } xs @ \text{sort } ys \) is sorted, so that due to the uniqueness of the \texttt{sort} operation, we have:

\[
\text{sort } (xs @ ys) = \text{sort } xs @ \text{sort } ys \quad (3.7)
\]

Then:

\[
\begin{align*}
  & select \ k \ (zs @ ys) \\
  &= \text{sort } (zs @ ys)!k \quad \text{using (3.3)} \\
  &= (\text{sort } xs @ \text{sort } ys)!k \quad \text{using (3.7)} \\
  &= \text{if } k < |zs| \text{ then } \text{sort } zs!k \text{ else } \text{sort } ys!k \ (k - |zs|) \\
  &= \text{if } k < |zs| \text{ then } select \ k \ xs \text{ else } select \ (k - |zs|) \ ys \quad \text{using (3.3)}
\end{align*}
\]

Now the recurrence outlined before is a direct consequence:

\textbf{Theorem 3.3 (A recurrence for \( select \)).}

\[
\begin{align*}
  & k < |zs| \to \\
  & select \ k \ xs \\
  &= (\text{let } (ls, es, gs) = \text{partition3 } x \ xs \\
  & \quad \text{in if } k < |ls| \text{ then } select \ k \ ls \\
  & \quad \text{else if } k < |ls| + |es| \text{ then } select \ (k - |ls| - |es|) \ gs)
\end{align*}
\]
Proof. We clearly have \(\text{mset } xs = \text{mset } ls + \text{mset } es + \text{mset } gs\) and \(|zs| = |ls| + |es| + |gs|\). Then:

\[
\begin{align*}
\text{select } k \quad &\quad = \text{select } k (ls \ominus es \ominus gs) \\
&\quad \text{using (3.5)} \\
&\quad = \text{if } k < |ls| \text{ then select } k ls \\
&\text{else if } k - |ls| < |es| \text{ then select } (k - |ls|) es \\
&\text{else select } (k - |ls| - |es|) gs
\end{align*}
\]

Clearly, \(k - |ls| < |es| \iff k < |ls| + |es|\) and \(\text{select } (k - |ls|) es = x\) since \(\text{select } (k - |ls|) es \in \text{set } es\) and \(\text{set } es = \{x\}\) 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” \cite[Section 2.5]{39}.
- If we choose a pivot that cuts the list in half every time, we get roughly \(\log n\) 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:

\[
\text{median :: 'a list } \Rightarrow \ 'a
\]

\[
\text{median } xs = \text{select } (|xs| - 1 \div 2) xs
\]

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

\[
x \neq \emptyset \land k < |x| \longrightarrow \\
reduce\_select\_median k x \neq \emptyset \land \\
median (reduce\_select\_median k x) = select k x
\]

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 \( x \) be a list and let \( < \) be either \( < \) or \( > \). Define

\[
M := \text{median} (\text{map median} (\text{chop 5} x))
\]

where the \text{chop} function performs the chopping described earlier. Then:

\[
|\{ y \in_x \text{mset} x \mid y < M \}| \leq [0.7 \cdot n + 3]
\]

Proof. The result of \( \text{chop 5} x \) is a list of \( \lceil \frac{n}{5} \rceil \) chunks, each of size at most 5, i.e. \( |\text{chop 5} x| = \lceil n / 5 \rceil \). Let us split these chunks into two groups according to whether their median is \( < M \) or \( \geq M \):

\[
Y_\prec := \{ y \in_x \text{mset} (\text{chop 5} x) \mid \text{median} y < M \}
\]

\[
Y_\succ := \{ y \in_x \text{mset} (\text{chop 5} x) \mid \text{median} y \geq M \}
\]

We clearly have
\[mset \, xs = \left( \sum_{ys \leftarrow \text{chop} \, 5 \, xs} mset \, ys \right)\]  
(3.8)

\[mset \left( \text{chop} \, 5 \, xs \right) = Y_\prec + Y_\succ\]  
(3.9)

\[\lfloor n / 5 \rfloor = |Y_\prec| + |Y_\succ|\]  
(3.10)

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

\[|Y_\prec| < \frac{1}{2} \cdot \lfloor n / 5 \rfloor\]  
(3.11)

The core idea of the proof is that any group \(ys \in Y_\succ\) can have at most 2 elements that are \(< M\):

\[
\begin{align*}
\{y \in_\prec mset \, ys \mid y < M\} 
\leq & \{y \in_\prec mset \, ys \mid y < \text{median} \, ys\} & \text{because median} \, ys \geq M \\
\leq & |ys| \div 2 & \text{using (3.1)} \\
\leq & 5 \div 2 = 2
\end{align*}
\]

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:

\[
\begin{align*}
\forall ys \in Y_\prec.\ |\{y \in_\prec mset \, ys \mid y < M\}| & \leq 2 \\
\forall ys \in Y_\succ.\ |\{y \in_\prec mset \, ys \mid y < M\}| & \leq 5
\end{align*}
\]
(3.12)

With this, we can begin our estimation of the number of elements \(< M\):

\[
\begin{align*}
\{y \in_\prec mset \, xs \mid y < M\} 
= & \{y \in_\prec (\sum_{ys \leftarrow \text{chop} \, 5 \, xs} mset \, ys) \mid y < M\} & \text{using (3.8)} \\
= & \sum_{ys \leftarrow \text{chop} \, 5 \, xs} \{y \in_\prec mset \, ys \mid y < M\} \\
= & \sum_{ys \in_\prec (Y_\prec + Y_\succ)} \{y \in_\prec mset \, ys \mid y < M\} & \text{using (3.9)}
\end{align*}
\]

Taking the size of both sides, we have

\[
\begin{align*}
\{y \in_\prec mset \, xs \mid y < M\} 
\leq & \sum_{ys \in_\prec (Y_\prec + Y_\succ)} \{y \in_\prec mset \, ys \mid y < M\} \\
= & \sum_{ys \in_\prec Y_\prec} \{y \in_\prec mset \, ys \mid y < M\} + \sum_{ys \in_\prec Y_\succ} \{y \in_\prec mset \, ys \mid y < M\} \\
\leq & (\sum_{ys \in_\prec Y_\prec} 5) + (\sum_{ys \in_\prec Y_\succ} 2) & \text{using (3.12)} \\
= & 5 \cdot |Y_\prec| + 2 \cdot |Y_\succ| \\
= & 2 \cdot \lfloor n / 5 \rfloor + 3 \cdot |Y_\succ| & \text{using (3.10)} \\
\leq & 3.5 \cdot \lfloor n / 5 \rfloor & \text{using (3.11)} \\
= & 3.5 \cdot \lfloor n / 5 \rfloor \leq [0.7 \cdot n + 3]
\end{align*}
\]

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 \( \lfloor 0.2 \cdot n \rfloor \) 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:

\[
\begin{align*}
\text{chop} :: \text{nat} & \Rightarrow \text{'a list} \Rightarrow \text{'a list list} \\
\text{chop} 0 \_ & = [] \\
\text{chop} \_ [] & = [] \\
\text{chop} n \_ xs & = \text{take} n xs \# \text{chop} n \left( \text{drop} n \_ xs \right)
\end{align*}
\]

\[
\begin{align*}
\text{slow_select} :: \text{nat} & \Rightarrow \text{'a list} \Rightarrow \text{'a} \\
\text{slow_select} k \_ xs & = \text{isort} xs ! k
\end{align*}
\]

\[
\begin{align*}
\text{slow_median} :: \text{'a list} & \Rightarrow \text{'a} \\
\text{slow_median} \_ xs & = \text{slow_select} \left( \left( |xs| - 1 \right) \div 2 \right) xs
\end{align*}
\]

\[
\begin{align*}
\text{mom_select} :: \text{nat} & \Rightarrow \text{'a list} \Rightarrow \text{'a} \\
\text{mom_select} k \_ xs & = \text{(if} |xs| \leq 20 \text{ then slow_select k xs } \\
& \quad \text{else let } M = \text{mom_select} \left( \left( |xs| \div 5 \right) - 1 \right) \div 2 \text{ ) } \\
& \quad \left( \text{map slow_median (chop 5 xs)} \right) \text{; } \\
& \quad (ls, es, gs) = \text{partition3 M xs } \\
& \quad \text{in if} k < |ls| \text{ then mom_select k ls } \\
& \quad \text{else if} k < |ls| + |es| \text{ then M } \\
& \quad \text{else mom_select (k - |ls| - |es|) gs})
\end{align*}
\]

The (partial) correctness statement is simple:

**Theorem 3.5 (Partial Correctness of mom_select).** Let \( xs \) be a list and \( k < |xs| \). Then if \( \text{mom_select k xs} \) terminates, we have
\[
\text{mom_select k xs} = \text{select k xs}.
\]

**Proof.** Straightforward computation induction using Theorem 3.3. \( \square \)
Theorem 3.6 (Termination of \textit{mom\_select}). Let \(xs\) be a list and \(k < |xs|\). Then \textit{mom\_select} \(k \) \(xs\) terminates.

\textit{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 \(|xs| / 5|\), 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 \(M = \text{median} \ (\text{map median} \ (\text{chop} \ 5 \ xs))\) and thus:
  \[M \in \text{set} \ (\text{map median} \ (\text{chop} \ 5 \ xs))\]
  \[\subseteq \bigcup_{ys \in \text{set} \ (\text{chop} \ 5 \ xs)} \text{set} \ ys = \text{set} \ xs\]
  Hence, \(M \in \text{set} \ xs\) but \(M \notin \text{set} \ ls\) and \(M \notin \text{set} \ gs\) by construction.

\[\Box\]

Exercise 3.6. The recursive definition of \textit{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:

\[
T_{\text{mom\_select}} :: \text{nat} \Rightarrow 'a \ list \Rightarrow \text{nat}
\]

\[
T_{\text{mom\_select}} \ k \ xs
= (\text{if } |xs| \leq 20 \text{ then } T_{\text{slow\_select}} \ k \ xs
  \text{ else let } xss = \text{chop} \ 5 \ xs;
  ms = \text{map slow\_median} \ xss;
  idx = ((|xs| + 4) \text{ div} 5 - 1) \text{ div} 2;
  x = \text{mom\_select} \ idx \ ms;
  (ls, es, gs) = \text{partition}3 \ x \ xs
  \text{ in } T_{\text{mom\_select}} \ idx \ ms + T_{\text{chop}} \ 5 \ xs + T_{\text{map}} \ T_{\text{slow\_median}} \ xss +
  T_{\text{partition}3} \ x \ xs + T_{\text{length}} \ ls + T_{\text{length}} \ es + 1 +
  (\text{if } k < |ls| \text{ then } T_{\text{mom\_select}} \ k \ ls
  \text{ else if } k < |ls| + |es| \text{ then } 0
  \text{ else } T_{\text{mom\_select}} \ (k - |ls| - |es|) \ gs))
\]
We can then prove

\[ T_{\text{mom\_select}} k \, x s \leq T'_{\text{mom\_select}} \mid x s \]

where the upper bound \( T'_{\text{mom\_select}} \) is defined as follows:

\[
T'_{\text{mom\_select}} :: \text{nat} \Rightarrow \text{nat} \\
T'_{\text{mom\_select}} \, n = \begin{cases} 
463 & \text{if } n \leq 20 \\
T'_{\text{mom\_select}} [0.2 \cdot n] + T'_{\text{mom\_select}} [0.7 \cdot n + 3] + 17 \cdot n + 50 & \text{else} 
\end{cases}
\]

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 \( T'_{\text{mom\_select}} \in O(n) \).

Exercise 3.7. Show that the upper bound \( [0.7 \cdot n + 3] \) is fairly tight by giving an infinite family \((x s_i)_{i \in \mathbb{N}}\) 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 \( f :: \text{nat} \Rightarrow \text{nat\ list} \) such that \( \forall n. \ |f \, n| < |f \, (n + 1)| \) and

\[
\left| \left\{ y \in \text{mset} \, (f \, n) \mid y > \text{mom} \, (f \, n) \right\} \right| \geq 0.7
\]

where \( \text{mom} \, x s = \text{median} \, (\text{map} \, \text{median} \, (\text{chop} \, 5 \, x s)) \).

3.5 Akra–Bazzi Light

The function \( T_{\text{mom\_select}} \) (let us write it as \( f \) for now) fulfils the recurrence

\[
n > 20 \longrightarrow f \, n = f \, [0.2 \cdot n] + f \, [0.7 \cdot n + 3] + 17 \cdot n + 50
\]

Such divide-and-conquer recurrences are beyond the “normal” Master Theorem, but a generalisation, the Akra–Bazzi Theorem [3, 48, 17], does apply to them. Let us first abstract the situation a bit and consider the recurrence

\[
n > 20 \longrightarrow f \, n = f \, [a \cdot n + b] + f \, [c \cdot n + d] + C_1 \cdot n + C_2
\]

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 $\max 0 \cdot b$ and $\max 0 \cdot d$.

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:** $\forall n > n_0. (\forall m < n. \ f(m) \leq C_3 \cdot m + C_4) \implies f(n) \leq C_3 \cdot n + C_4$

We can see that in order to even be able to apply the induction hypothesis in the induction step, we need $|a \cdot n + b| < n$. We can make the estimate\(^2\)

$$|a \cdot n + b| \leq a \cdot n + b + 1 < n$$

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

$$n_0 \geq \frac{b + 1}{1 - a} \quad \text{and} \quad n_0 \geq \frac{d + 1}{1 - c} \quad (3.13)$$

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

$$C_3 \cdot [a \cdot n + b] + C_4 + (C_3 \cdot [c \cdot n + d] + C_4) + C_1 \cdot n + C_2 \leq C_3 \cdot n + C_4,$$

\(^1\) In Isabelle, the corresponding rule is called `less_induct`.

\(^2\) The notation $<$ 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,

\[ C_3 \cdot ([a \cdot n + b] + [c \cdot n + d] - n) + C_1 \cdot n + C_2 + C_4 \leq 0, \]

We estimate the left-hand side like this:

\[
C_3 \cdot ([a \cdot n + b] + [c \cdot n + d] - n) + C_1 \cdot n + C_2 + C_4
\leq C_3 \cdot (a \cdot n + b + 1 + (c \cdot n + d + 1) - n) + C_1 \cdot n + C_2 + C_4
= C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n \quad (*)
\leq C_3 \cdot (b + d + 2) + C_2 + C_4 - (C_3 \cdot (1 - a - c) - C_1) \cdot n_0 \quad (\ddagger)
\]

The step from (\*) to (\ddagger) 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

\[ C_3 > \frac{C_1}{1 - a - c}. \quad (3.14) \]

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

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

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:

\[ n_1 := \max \ n_0 \left( \frac{b + d + 2}{1 - a - c} + 1 \right) \quad C_4 := \max \ \{ f \ n \mid n \leq 20 \} \]

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

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

**Theorem 3.7 (Akra Bazzi Light).**

\[ a > 0 \land c > 0 \land a + c < 1 \land C_1 \geq 0 \land \]

\[
(\forall n > n_0. \ f \ n = f \ [a \cdot n + b] + f \ [c \cdot n + d] + C_1 \cdot n + C_2) \rightarrow \]

\[ (\exists C_3 \ C_4. \ \forall n. \ f \ n \leq C_3 \cdot n + C_4) \]

**Exercise 3.8.**

1. Suppose that instead of groups of 5, we now chop into groups of size \( l \geq 1 \). 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
Binary Trees

Binary trees are defined as a recursive data type:

\[
\text{datatype } 'a \text{ tree } = \text{Leaf} \mid \text{Node ('a tree) 'a ('a tree)}
\]

The following syntactic sugar is sprinkled on top:

\[
\langle \rangle \equiv \text{Leaf} \\
\langle l, x, r \rangle \equiv \text{Node } l \ x \ r
\]

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 \(\text{tree}\).

When displaying a tree in the usual graphical manner we show only the \(\text{Nodes}\). For example, \(\langle\langle\langle\rangle, 3, \langle\rangle\rangle, 9, \langle\langle\rangle, 7, \langle\rangle\rangle\rangle\) is displayed like this:

```
       9
     /   \
   3     7
```

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 \(map\):
set_tree :: 'a tree ⇒ 'a set
set_tree {} = {}
set_tree (l, x, r) = set_tree l ∪ {x} ∪ set_tree r

map_tree :: ('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree
map_tree f {} = {}
map_tree f (l, x, r) = (map_tree f l, f x, map_tree f r)

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

inorder :: 'a tree ⇒ 'a list
inorder {} = []
inorder (l, x, r) = inorder l @ [x] @ inorder r

preorder :: 'a tree ⇒ 'a list
preorder {} = []
preorder (l, x, r) = x # preorder l @ preorder r

These two size functions count the number of Nodes and Leafs in a tree:

size :: 'a tree ⇒ nat
size {} = 0
size (l, _, r) = size l + size r + 1

size1 :: 'a tree ⇒ nat
size1 {} = 1
size1 (l, _, r) = size1 l + size1 r

The syntactic sugar |t| for size t and |t|₁ for size1 t is only used in this text, not in the Isabelle theories.

Induction proves a convenient fact that explains the name size1:

|t|₁ = |t| + 1

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

Let’s define the height and min_height functions:

- \( \text{height} :: \text{a tree} \Rightarrow \text{nat} \)
  \( \text{height} \langle \rangle = 0 \)
  \( \text{height} \langle l, _, r \rangle = \max (\text{height} l) (\text{height} r) + 1 \)

- \( \text{min_height} :: \text{a tree} \Rightarrow \text{nat} \)
  \( \text{min_height} \langle \rangle = 0 \)
  \( \text{min_height} \langle l, _, r \rangle = \min (\text{min_height} l) (\text{min_height} r) + 1 \)

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 \( \text{height} \) and \( \text{min_height} \). The abbreviations \( h \) and \( mh \) are only used in this text, not in the Isabelle theories.

The obvious properties \( h t \leq |t| \) and \( mh t \leq h t \) and the following classical properties have easy inductive proofs:

\[
2^{mh t} \leq |t|_1 \quad |t|_1 \leq 2^{h t}
\]

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:

- \( \text{subtrees} :: \text{a tree} \Rightarrow \text{a tree set} \)
  \( \text{subtrees} \langle \rangle = \{\langle \rangle\} \)
  \( \text{subtrees} \langle l, a, r \rangle = \{\langle l, a, r \rangle\} \cup \text{subtrees} l \cup \text{subtrees} r \)

Note that every tree is a subtree of itself.

**Exercise 4.1.** Function \( \text{inorder} \) has quadratic complexity because the running time of \( (\@) \) is linear in the length of its first argument. Define a function \( \text{inorder2} :: \text{a tree} \Rightarrow \text{a list} \Rightarrow \text{a list} \) 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 \( \text{inorder2} t xs = \text{inorder} t @ xs \).

### 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 \( \text{complete} \) is defined recursively:
Fig. 4.1. A complete tree

![Diagram of a complete tree](image)

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

**Lemma 4.1.** \( \text{complete } t \iff \text{mh } t = h_t \)

**Proof.** The proof is by induction and case analyses on \( \text{min} \) and \( \text{max} \). \( \square \)

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

**Lemma 4.2.** \( \text{complete } t \rightarrow |t|_1 = 2^h t \)

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

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

**Lemma 4.3.** \( \neg \text{complete } t \rightarrow |t|_1 < 2^h t \)

**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: \( |t|_1 = |l|_1 + |r|_1 < 2^h l + 2^h r < 2 \cdot 2^h r = 2^h t \). If \( h l = h r \), then either \( l \) or \( r \) must be incomplete. We consider the case \( \neg \text{complete } l \) (the case \( \neg \text{complete } r \) is symmetric). From the IH \( |l|_1 < 2^h l \), \( |r|_1 < 2^h r \) and \( h l = h r \) the claim follows: \( |t|_1 = |l|_1 + |r|_1 < 2^h l + 2^h r = 2 \cdot 2^h r = 2^h t \). \( \square \)

**Lemma 4.4.** \( \neg \text{complete } t \rightarrow 2^{\text{mh } t} < |t|_1 \)

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 \( |t|_1 = 2^h t \rightarrow \text{complete } t \), the converse of Lemma 4.2. Thus we arrive at:
Corollary 4.5. $complete\ t \iff |t|_1 = 2^{h\ t}$

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:

$$u \in\ subtrees\ t \land complete\ u \implies h\ u \leq h\ \left(mcs\ t\right)$$

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:

$$acomplete :: \forall t \Rightarrow bool$$

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

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.

![Almost Complete Tree](image)

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. \text{a}completely \ s \ \land |s| \leq |t| \implies h_s \leq h_t

Proof. The proof is by cases. If \text{complete} \ s \ then, by Lemma 4.2, \ 2^h s = |s|, 
\leq |t|, \leq 2^h t \ and \ thus \ h s \leq h t. \ Now \ assume \ \neg \text{complete} \ s. \ Then \ Lemma 4.4 
yields \ 2^{mh s} < |s|, \leq |t|, \leq 2^h t \ and \ thus \ mh s < h t. \ Furthermore \ we \ have \ h s = mh s \leq 1 \ (from \ \text{a}complete \ 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. \hfill \square

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. \text{a}complete \ t \implies h = \lfloor \log |t| \rfloor

Proof. The proof is by cases. If \text{t} is complete, the claim follows from 
Lemma 4.2. Now assume \text{t} is incomplete. Then \ h t = mh t + 1 \ because 
\text{a}complete \ t, \ mh t \leq h t \ and \ complete \ t \iff mh t = h t \ (Lemma \ 4.1). 
Together with \ |t| \leq 2^h t \ this \ yields \ |t|_1 \leq 2^{mh t + 1} \ and \ thus \ \log |t|_1 \leq mh t + 1. \ By \ Lemma \ 4.4 \ we \ obtain \ mh t < \log |t|_1. \ These \ two \ bounds \ for \ \log |t|_1 \ together \ imply \ the \ claimed \ h t = \lfloor \log |t|_1 \rfloor. \hfill \square

In the same manner we also obtain:

Lemma 4.8. \text{a}complete \ t \implies mh t = \lfloor \log |t|_1 \rfloor

4.3.1 Converting a List into an Almost Complete Tree

We will now see how to convert a list \text{xs} into an almost complete tree \text{t} such 
that \text{inorder} \ t = \text{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 \ \text{n} \cdot 
\log \text{n} \ algorithm \ but \ we \ want \ a \ linear \ one. \ Therefore \ we \ use \ an \ additional \ \text{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:

\begin{verbatim}
bal :: nat -> 'a list -> 'a tree x 'a list
bal n xs
  = (if n = 0 then (()), xs)
    else let m = n div 2;
         (l, ys) = bal m xs;
    in (l, ys);
\end{verbatim}
\[(r, zs) = \text{bal} \ (n - 1 - m) \ (\text{tl} \ ys) \]
\[\text{in} \ ((l, \text{hd} \ ys, \ r), \ zs)\]

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

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

\[
\begin{align*}
\text{bal\_list} & :: \text{nat} \Rightarrow \text{'a list} \Rightarrow \text{'a tree} \\
\text{bal\_list} \ n \ xs &= \text{fst} \ (\text{bal} \ n \ xs) \\
\text{balance\_list} & :: \text{'a list} \Rightarrow \text{'a tree} \\
\text{balance\_list} \ xs &= \text{bal\_list} \ |xs| \ xs \\
\text{bal\_tree} & :: \text{nat} \Rightarrow \text{'a tree} \Rightarrow \text{'a tree} \\
\text{bal\_tree} \ n \ t &= \text{bal\_list} \ n \ (\text{inorder} \ t) \\
\text{balance\_tree} & :: \text{'a tree} \Rightarrow \text{'a tree} \\
\text{balance\_tree} \ t &= \text{bal\_tree} \ |t| \ t
\end{align*}
\]

Correctness

The following lemma clearly expresses that \(\text{bal} \ n \ xs\) turns the prefix of length \(n\) of \(xs\) into a tree and returns the corresponding suffix of \(xs\):
Lemma 4.9. \( n \leq |xs| \land \text{bal } n \ xs = (t, zs) \rightarrow \)

\[ xs = \text{inorder } t \ @ \ zs \land |t| = n \]

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 \( m = n \ \text{div} \ 2 \) and \( m' = n - 1 - m \) (and thus \( m, m' < n \)). From \( \text{bal } n \ xs = (t, zs) \) we obtain \( l, r \) and \( ys \) such that \( \text{bal } m \ xs = (l, ys), \text{bal } m' \ (tl \ ys) = (r, zs) \) and \( t = \langle l, \ hd \ ys, r \rangle \). Because \( m < n \leq |xs| \) the induction hypothesis implies \( xs = \text{inorder } l \ @ \ ys \land |l| = m \ (1) \). This in turn implies \( m' \leq |tl \ ys| \) and thus the induction hypothesis implies \( tl \ ys = \text{inorder } r \ @ \ zs \land |r| = m' \ (2) \). Properties (1) and (2) together with \( t = \langle l, \ hd \ ys, r \rangle \) imply the claim \( xs = \text{inorder } t \ @ \ zs \land |t| = n \) because \( ys \neq [] \).

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

\[ n \leq |xs| \rightarrow \text{inorder } (\text{bal_list } n \ xs) = \text{take } n \ xs \]
\[ \text{inorder } (\text{balance_list } xs) = xs \]
\[ n \leq |t| \rightarrow \text{inorder } (\text{bal_tree } n \ t) = \text{take } n \ (\text{inorder } t) \]
\[ \text{inorder } (\text{balance_tree } t) = \text{inorder } t \]

To prove that \( \text{bal} \) returns an almost complete tree we determine its height and minimal height.

Lemma 4.10. \( n \leq |xs| \land \text{bal } n \ xs = (t, zs) \rightarrow h \ t = \lfloor \lg \ (n + 1) \rfloor \)

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 hypotheses \( h \ l = \lfloor \lg \ (m + 1) \rfloor \) and \( h \ r = \lfloor \lg \ (m' + 1) \rfloor \). This leads to

\[ h \ t = \max \ (h \ l) \ (h \ r) + 1 \]
\[ = h \ l + 1 \quad \text{because } m' \leq m \]
\[ = \lfloor \lg \ (m + 1) \rfloor + 1 \]
\[ = \lfloor \lg \ (n + 1) \rfloor \]

by (2.29) □

The following complementary lemma is proved in the same way:

Lemma 4.11. \( n \leq |xs| \land \text{bal } n \ xs = (t, zs) \rightarrow mh \ t = \lfloor \lg \ (n + 1) \rfloor \)

By definition of \( \text{acomplete} \) and because \( |x| - |x| \leq 1 \) we obtain that \( \text{bal} \) (and consequently the functions that build on it) returns an almost complete tree:

Corollary 4.12. \( n \leq |xs| \land \text{bal } n \ xs = (t, ys) \rightarrow \text{acomplete } t \)
4.3.2 Exercises

Exercise 4.3. Find a formula $B$ such that $\text{acomp}let\_\langle l, x, r \rangle = B$ where $B$ may only contain the functions $\text{acomp}let$, $\text{com}pl$te, height, arithmetic and boolean operations, $l$ and $r$, but in particular not $\text{min}\_\text{height}$ or $\text{Node}$ ($\Rightarrow \langle \_, \_, \_ \rangle$). Prove $\text{acomp}let\_\langle l, x, r \rangle = B$.

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

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 $\text{inorder}$, when applied to an augmented tree, should return an $'a$ list. Thus we redefine it in the obvious way:

\[
\text{inorder} :: ('a \times 'b) \text{tree} \Rightarrow 'a \text{list}
\]

\[
\text{inorder} \langle \rangle = []
\]

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

Another example is $\text{set\_tree} :: ('a \times 'b) \text{tree} \Rightarrow 'a \text{set}$. 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 $\text{inorder}$ 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 $\text{inorder}$ (and similar functions): $\text{map \_fst} \circ \text{Tree}$.inorder or $\text{Tree}$.inorder $\circ$ map_tree 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:

\[
\begin{align*}
  sz :: ('a \times \text{nat}) \text{tree} & \Rightarrow \text{nat} \\
  sz \langle \rangle & = 0 \\
  sz \langle _, (_, n), _ \rangle & = n \\

  node_{sz} :: ('a \times \text{nat}) \text{tree} & \Rightarrow ('a \times \text{nat}) \text{tree} \\
  node_{sz} l a r & = \langle l, (a, sz l + sz r + 1), r \rangle
\end{align*}
\]

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

\[
\begin{align*}
  \text{invar}_{sz} :: ('a \times \text{nat}) \text{tree} & \Rightarrow \text{bool} \\
  \text{invar}_{sz} \langle \rangle & = \text{True} \\
  \text{invar}_{sz} \langle l, (_, n), r \rangle & = (n = sz l + sz r + 1 \land \text{invar}_{sz} l \land \text{invar}_{sz} r)
\end{align*}
\]

This property is preserved by \text{node}_{sz} (i.e. \text{invar}_{sz} l \land \text{invar}_{sz} r \longrightarrow \text{invar}_{sz} (node_{sz} l a r)) and it guarantees that \text{sz} returns the size:

\[
\text{invar}_{sz} t \longrightarrow sz t = |t|
\]

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

\[
\begin{align*}
  F :: ('a \times 'b) \text{tree} & \Rightarrow 'b \\
  F \langle \rangle & = \text{zero} \\
  F \langle l, (a, _), r \rangle & = f (F l) a (F r)
\end{align*}
\]

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

\[
\begin{align*}
  b_{val} :: ('a \times 'b) \text{tree} & \Rightarrow 'b \\
  b_{val} \langle \rangle & = \text{zero}
\end{align*}
\]
b_val (_, (_, b), _) = b

node_f :: (\texttt{a} \times \texttt{b}) \text{ tree} \Rightarrow \texttt{a} \Rightarrow (\texttt{a} \times \texttt{b}) \text{ tree} \Rightarrow (\texttt{a} \times \texttt{b}) \text{ tree}

node_f l a r = \langle l, (a, f (b_val l) a (b_val r)), r \rangle

If all \texttt{b}-values are computed as in node_f

\begin{align*}
\text{invar}_f &:: (\texttt{a} \times \texttt{b}) \text{ tree} \Rightarrow \text{bool} \\
\text{invar}_f \langle \rangle &= \text{True} \\
\text{invar}_f \langle l, (a, b), r \rangle &= \langle b = f (b_val l) a (b_val r) \land \text{invar}_f l \land \text{invar}_f r \rangle
\end{align*}

then all \texttt{b}-values equal \texttt{F}: \text{invar}_f t \rightarrow 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 \texttt{ch}, node\_ch and invar\_ch as in Section 4.4.1 and prove the following properties:

\begin{align*}
invar\_ch t &\rightarrow ch t = (\text{complete } t, \?) \\
invar\_ch l \land invar\_ch r &\rightarrow invar\_ch (node\_ch l a r)
\end{align*}

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

\begin{align*}
invar\_mx t &\rightarrow mx t = (\text{if } t = \langle \rangle \text{ then } \textbf{None} \textbf{ else } \textbf{Some} (\text{Max (set}\_\text{tree} t)))
\end{align*}

where \texttt{Max} is the predefined maximum operator on finite, non-empty sets.
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 \texttt{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:

\[
\texttt{bst :: \langle \textit{a::linorder} \rangle \textit{tree} \Rightarrow bool}
\]
\[
\texttt{bst \langle \rangle = True}
\]
\[
\texttt{bst \langle l, a, r \rangle = ((\forall x \in \textit{set\_tree}\ l. \ x < a) \wedge (\forall x \in \textit{set\_tree}\ r. \ a < x) \wedge \texttt{bst}\ l \wedge \texttt{bst}\ r)}
\]

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

\[ \text{nodes} :: 'a \text{ tree} \Rightarrow (\forall (l,a,r) \in \text{nodes } t. \ldots) \] and prove \( \text{bst_nodes } t = \text{bst } t \).

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:

\[
\begin{align*}
\text{empty} :: 's \\
\text{insert} :: 'a \Rightarrow 's \Rightarrow 's \\
\text{delete} :: 'a \Rightarrow 's \Rightarrow 's \\
\text{isin} :: 's \Rightarrow 'a \Rightarrow \text{bool}
\end{align*}
\]

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 \( 'a \text{ set} \) via an abstraction function, e.g. \( \text{set_tree} \).

5.2 Implementing Sets via unbalanced BSTs

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

\[
\begin{align*}
\text{datatype} \quad \text{cmp_val} = \text{LT} | \text{EQ} | \text{GT} \\
\text{cmp} :: ('a::linorder) \Rightarrow 'a \Rightarrow \text{cmp_val} \\
\text{cmp } x \ y = (\text{if } x < y \text{ then } \text{LT} \text{ else if } x = y \text{ then } \text{EQ} \text{ else } \text{GT})
\end{align*}
\]
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.

```haskell
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 `(l, a, r)` (where `r ≠ ()`) 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 h l a r by “joining” l and r:

```plaintext
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 (t1, a, t2) (t3, b, t4) = (case join t2 t3 of
   () ⇒ (t1, a, (), b, t4) |
   (u2, x, u3) ⇒ ((t1, a, u2), x, (u3, b, t4)))
```

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:

```plaintext
join0 :: 'a tree ⇒ 'a tree ⇒ 'a tree
join0 t () = t
join0 () t = t
join0 (t1, a, t2) (t3, b, t4) = (t1, a, join0 t2 t3, b, t4)
```

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:

\[ h (join l r) \leq \max (h l) (h r) + 1 \]

Now show that join0 behaves badly: find an upper bound ub of \( h (join0 l r) \) 
such that ub is a function of h l and h r. Prove \( h (join0 l r) \leq ub \) 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 \texttt{empty}, \texttt{insert}, \texttt{delete} and \texttt{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:

\[
\begin{align*}
\text{set\_tree empty} &= \{\} \\
\text{set\_tree (insert x t)} &= \text{set\_tree t} \cup \{x\} \\
\text{set\_tree (delete x t)} &= \text{set\_tree t} - \{x\} \\
\text{isin t x} &= (x \in \text{set\_tree t})
\end{align*}
\]

However, the implementation only works correctly on BSTs. Therefore we need to add the precondition \texttt{bst t} to all but the first property. But why are we permitted to assume this precondition? Only because \texttt{bst} is an invariant of this implementation: \texttt{bst} holds for \texttt{empty}, and both \texttt{insert} and \texttt{delete} preserve \texttt{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:

\[
\begin{align*}
\text{bst empty} \\
\text{bst t} &\rightarrow \text{bst (insert x t)} \\
\text{bst t} &\rightarrow \text{bst (delete x t)}
\end{align*}
\]

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 \texttt{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 \( \text{bst} \) and \( \text{set_tree} \) via \( \text{inorder} \):

\[
\text{bst} \ t = \text{sorted} \ (\text{inorder} \ t) \quad \text{and} \quad \text{set_tree} \ t = \text{set} \ (\text{inorder} \ t)
\]

where

\[
\text{sorted} :: \ 'a \ \text{list} \ \Rightarrow \ \text{bool}
\]
\[
\text{sorted} \ [] = \ True
\]
\[
\text{sorted} \ [\_] = \ True
\]
\[
\text{sorted} \ (x \ # \ y \ # \ zs) = (x < y \land \text{sorted} \ (y \ # \ zs))
\]

Note that this is “sorted w.r.t. (\(<\))” whereas in the chapter on sorting \( \text{sorted} \) 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 \( \text{sorted} \) because they are abstractions of BSTs. Insertion and deletion on sorted lists can be defined as follows:

\[
\text{ins_list} :: \ 'a \Rightarrow \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list}
\]
\[
\text{ins_list} \ x \ [] = [x]
\]
\[
\text{ins_list} \ x \ (a \ # \ xs) = (\text{if} \ x < a \ \text{then} \ x \ # \ a \ # \ xs
\]
\[
\quad \text{else if} \ x = a \ \text{then} \ a \ # \ xs \ \text{else} \ a \ # \ \text{ins_list} \ x \ xs)
\]

\[
\text{del_list} :: \ 'a \Rightarrow \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list}
\]
\[
\text{del_list} \ _ \ [] = []
\]
\[
\text{del_list} \ x \ (a \ # \ xs) = (\text{if} \ x = a \ \text{then} \ xs \ \text{else} \ a \ # \ \text{del_list} \ x \ xs)
\]

The abstraction function from trees to lists is function \( \text{inorder} \). The specification in Figure 5.1 expresses that \( \text{empty}, \text{insert}, \text{delete} \) and \( \text{isin} \) implement \( [], \text{ins_list}, \text{del_list} \) and \( \lambda z x. \ x \in \text{set} \ x \). 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 \( \text{ins_list} \) and \( \text{del_list} \) preserve \( \text{sorted} \).
5.4 Correctness Proofs

\[
\begin{align*}
    \text{inorder empty} &= [] \\
    \text{sorted (inorder t)} &\rightarrow \text{inorder (insert x t) = ins_list x (inorder t)} \\
    \text{sorted (inorder t)} &\rightarrow \text{inorder (delete x t) = del_list x (inorder t)} \\
    \text{bst t} &\rightarrow \text{isin t x = (x \in \text{set_tree t})}
\end{align*}
\]

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

\[
\text{sorted (inorder t)} \rightarrow \text{inorder (insert x t) = ins_list x (inorder t)}
\]

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:

\[
\begin{align*}
    \text{inorder (insert x t)} &= \text{inorder (insert x l) @ a # inorder r} \\
    &= \text{ins_list x (inorder l) @ a # inorder r} \\
    &= \text{ins_list x (inorder l @ a # inorder r) = ins_list x t}
\end{align*}
\]

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 \( \text{sorted (inorder t)} \) to \( \text{sorted (inorder l @ [a]) \land sorted (a # inorder r)} \), thus allowing (5.5) to rewrite the term \( \text{ins_list x (inorder l @ a # inorder r)} \) to \( \text{ins_list x (inorder l @ a # inorder r)} \).

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 \( \text{inorder} \) of such a tree leads to a list of the following form:

\[
\text{inorder t}_1 @ a_1 # \text{inorder t}_2 @ a_2 # \ldots # \text{inorder t}_n
\]

Now we discuss the lemmas in Figure 5.2 that simplify the application of \( \text{sorted, ins_list} \) and \( \text{del_list} \) to such terms.

Terms of the form \( \text{sorted (xs1 @ a1 # xs2 @ a2 # \ldots # xs_n)} \) are decomposed into the following basic formulas

\[
\begin{align*}
    \text{sorted (xs @ [a])} &\quad (\text{simulating } \forall x \in \text{set xs. } x < a) \\
    \text{sorted (a # xs)} &\quad (\text{simulating } \forall x \in \text{set xs. } a < x) \\
    a < b
\end{align*}
\]

by the rewrite rules (5.1)–(5.2). Lemmas (5.3)–(5.4) enable deductions from basic formulas.
\textbf{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 \( \text{bst } t = \text{sorted } (\text{inorder } t) \). The correctness properties

\[
\text{set_tree empty } = \{\}
\]

\[
\text{set_tree } t \to \text{set_tree } (\text{insert } x t) = \text{set_tree } t \cup \{x\}
\]

\[
\text{set_tree } t \to \text{set_tree } (\text{delete } x t) = \text{set_tree } t - \{x\}
\]

\[
\text{set_tree } t = \text{set } (\text{inorder } t) \quad (\text{a basic tree lemma}), \text{ the properties in Figure 5.1 and the inductive correctness properties}
\]

\[
\text{set } (\text{ins_list } x xs) = \text{set } xs \cup \{x\}
\]

\[
\text{sorted } xs \to \text{set } (\text{del_list } x xs) = \text{set } xs - \{x\}
\]
Preservation of the invariant

\[
\begin{align*}
sorted \ (inorder \ empty) \\
sorted \ (inorder \ t) &\rightarrow \ sorted \ (inorder \ (insert \ x \ t)) \\
sorted \ (inorder \ t) &\rightarrow \ sorted \ (inorder \ (delete \ x \ t))
\end{align*}
\]

are trivial or consequences of the properties in Figure 5.1 and the preservation of \texttt{sorted} by \texttt{ins_list} and \texttt{del_list}:

\[
\begin{align*}
sorted \ xs &\rightarrow \ sorted \ (ins_list \ x \ xs) \\
sorted \ xs &\rightarrow \ sorted \ (del_list \ x \ xs)
\end{align*}
\]

This means in particular that preservation of \texttt{sorted} \ o \ \texttt{inorder} is guaranteed for any implementation of \texttt{empty}, \texttt{insert} and \texttt{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 \texttt{set_tree} and \texttt{inorder} from that section. Moreover we use a slightly adjusted version of \texttt{isin}:

\[
\begin{align*}
\text{isin} :: (\forall a \times b. \ \text{tree} \Rightarrow a \Rightarrow \text{bool}) \\
\text{isin} \ () &= \text{False} \\
\text{isin} \ (l, (a, _), r) \ x &= \begin{cases} 
\text{isin} \ l \ x \ | \ \text{EQ} \Rightarrow \text{True} \ | \ \text{GT} \Rightarrow \text{isin} \ r \ x
\end{cases}
\end{align*}
\]

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

5.5.2 Intervals

An interval \texttt{ivl} is simply a pair of lower and upper bound, accessed by functions \texttt{low} and \texttt{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:

\[
low \ p \leq \ high \ p
\]

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:

\[
(x < y) = (low x < low y \lor low x = low y \land high x < high y)
\]
\[
(x \leq y) = (low x < low y \lor low x = low y \land high x \leq high y)
\]

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

\[
\text{overlap} \ x \ y = (low y \leq high x \land low x \leq high y)
\]

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

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

\[
\text{has\_overlap} \ S \ y = (\exists \ x \in S. \ \text{overlap} \ x \ y)
\]

### 5.5.3 Interval Trees

An interval tree associates to each node a number \(\text{max\_hi}\), 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.

\[
\text{type\_synonym} \ 'a \ ivl\_tree = ('a \ ivl \times 'a) \ tree
\]
\[
\text{max\_hi} :: 'a \ ivl\_tree \Rightarrow 'a
\]
\[
\text{max\_hi} \ () = \bot
\]
\[
\text{max\_hi} \ (_, (\_, m), \_) = m
\]

If the \(\text{max\_hi}\) value of every node in a tree agrees with \(\text{max3}\)
it follows by induction that \( \text{max}_h \) is the maximum value of \text{high} in the tree and comes from some node in the tree:

**Lemma 5.1.** \( \text{inv}_\text{max}_h t \land a \in \text{set}_t \rightarrow \text{high} a \leq \text{max}_h t \)

**Lemma 5.2.** \( \text{inv}_\text{max}_h t \land t \neq \langle \rangle \rightarrow \\
(\exists a \in \text{set}_t. \ \text{high} a = \text{max}_h t) \)

### 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 \( \text{empty} = \langle \rangle \). Function \( \text{isin} \) 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 \( \text{max}_h \) must be computed (by \( \text{max3} \)) for each new node. We follow Section 4.4 and introduce a smart constructor \( \text{node} \) for that purpose and replace \( \langle l, a, r \rangle \) by \( \text{node} l a r \) (on the right-hand side):

\[
\begin{align*}
\text{inv}_\text{max}_h :: &\ 'a \text{ivl} \Rightarrow \text{bool} \\
\text{inv}_\text{max}_h \langle \rangle &= \text{True} \\
\text{inv}_\text{max}_h \langle l, (a, m), r \rangle &= (m = \text{max3} a (\text{max}_h l) (\text{max}_h r) \land \text{inv}_\text{max}_h l \land \\
&\ \text{inv}_\text{max}_h r) \\
\text{max3} :: &\ 'a \text{ivl} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \\
\text{max3} a m n &= \text{max} (\text{high} a) (\text{max} m n)
\end{align*}
\]
else let \((x, l') = split_min l\) in \((x, node l' a r)\)

\[
delete :: 'a ivl \Rightarrow 'a ivl_tree \Rightarrow 'a ivl_tree
\[
delete \_ \() \_ = \()
\]
delete \_ \((l, (a, \_), r)\)
= (case \(cmp \_ a \_\) of
  \(LT \Rightarrow node (delete \_ l) a r | \)
  \(EQ \Rightarrow if r = \() then l else let \((x, y) = split_min r\) in node l x y | \)
  \(GT \Rightarrow node l a (delete \_ r)\))

The correctness proofs for insertion and deletion cover two aspects. Functional correctness and preservation of the invariant \(sorted \circ inorder\) (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

\[
\text{sorted} (inorder t) \rightarrow inorder (insert x t) = \text{ins}\_\text{list} x (inorder t)
\]
\[
\text{sorted} (inorder t) \rightarrow inorder (delete x t) = \text{del}\_\text{list} x (inorder t)
\]
\[
\text{inv}\_\text{max}\_\text{hi} t \rightarrow \text{inv}\_\text{max}\_\text{hi} (insert x t)
\]
\[
\text{inv}\_\text{max}\_\text{hi} t \rightarrow \text{inv}\_\text{max}\_\text{hi} (delete x t)
\]

Defining \(\text{invar} t = (\text{inv}\_\text{max}\_\text{hi} t \land \text{sorted} (inorder t))\) we obtain the following top-level correctness corollaries:

\[
\text{invar} s \rightarrow \text{set}\_\text{tree} (insert x s) = \text{set}\_\text{tree} s \cup \{x\}
\]
\[
\text{invar} s \rightarrow \text{set}\_\text{tree} (delete x s) = \text{set}\_\text{tree} s - \{x\}
\]
\[
\text{invar} s \rightarrow \text{invar} (insert x s)
\]
\[
\text{invar} s \rightarrow \text{invar} (delete x s)
\]

The above insertion function allows overlapping intervals to be added into the tree and deletion supports only deletion of whole intervals. This is appropriate for the computational geometry application sketched below in Subsection 5.5.6. Other applications may require a different design.

### 5.5.5 Searching for an Overlapping Interval

The added functionality of interval trees over ordinary BSTs is function \(\text{search}\) that searches for an overlapping rather than identical interval:
Theorem 5.3. \( \text{inv\_max\_hi} \ t \land \text{sorted} \ (\text{inorder} \ t) \rightarrow \text{search} \ t \ x = \text{has\_overlap} \ (\text{set\_tree} \ t) \ x \)

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

- If \( l \neq \langle \rangle \) and \( \text{low} x \leq \text{max\_hi} \ l \), the search goes to the left child. If there is an interval in the left child overlapping with \( x \), then the search returns \( \text{True} \) 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 \( \text{high} p = \text{max\_hi} \ l \). Since \( \text{low} x \leq \text{max\_hi} \ l \), we have \( \text{low} x \leq \text{high} p \). Since \( p \) does not overlap with \( x \), we must have \( \text{high} x < \text{low} p \). But then, for every interval \( rp \) in the right child, \( \text{low} p \leq \text{low} \ rp \), so that \( \text{high} x < \text{low} \ rp \), which implies that \( \rp \) does not overlap with \( x \).

- Now we consider the case where either \( l = \langle \rangle \) or \( \text{max\_hi} \ l < \text{low} x \). 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 \( \text{high} lp \leq \text{max\_hi} \ l \) by Lemma 5.1, so that \( \text{high} lp < \text{low} x \), which means \( lp \) does not overlap with \( x \).

\( \square \)

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

\[
\text{in\_ivl} :: \text{'a} \Rightarrow \text{'a ivl} \Rightarrow \text{bool}
\]
\[
\text{in\_ivl} \ x iv = (\text{low} iv \leq x \land x \leq \text{high} iv)
\]

write a recursive function

\[
\text{search1} :: \text{'a ivl\_tree} \Rightarrow \text{'a} \Rightarrow \text{bool}
\]

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

\[
\text{inv\_max\_hi} \ t \land \text{bst} \ t \rightarrow \text{search1} \ t x = (\exists iv \in \text{set\_tree} \ t. \ \text{in\_ivl} \ x iv)
\]
5.5.6 Application

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 \(O(n \log n)\), 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].
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:

\[
\text{ADT} = \text{interface} + \text{specification}
\]

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:

- \(\text{empty} :: \emptyset\)
- \(\text{insert} :: a \Rightarrow s \Rightarrow s\)
- \(\text{delete} :: a \Rightarrow s \Rightarrow s\)
- \(\text{isin} :: s \Rightarrow a \Rightarrow \text{bool}\)

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** \( \text{invar} :: T \Rightarrow \text{bool} \)

Note that the abstraction function and the invariant are not part of the in-
terface, 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 speci-
fication. This is the template for ADTs that we follow throughout the book.

**ADT Set**

**interface**

\[
\begin{align*}
\text{empty} & :: 's \\
\text{insert} & :: 'a \Rightarrow 's \Rightarrow 's \\
\text{delete} & :: 'a \Rightarrow 's \Rightarrow 's \\
\text{isin} & :: 's \Rightarrow 'a \Rightarrow \text{bool}
\end{align*}
\]

**abstraction** \( \text{set} :: 's \Rightarrow 'a \text{ set} \)

**invariant** \( \text{invar} :: 's \Rightarrow \text{bool} \)

**specification**

\[
\begin{align*}
\text{invar empty} & \quad (\text{empty-inv}) \\
\text{set empty} & = \{\} \quad (\text{empty}) \\
\text{invar s} & \rightarrow \text{invar } (\text{insert x s}) \quad (\text{insert-inv}) \\
\text{invar s} & \rightarrow \text{set } (\text{insert x s}) = \text{set s } \cup \{x\} \quad (\text{insert}) \\
\text{invar s} & \rightarrow \text{invar } (\text{delete x s}) \quad (\text{delete-inv}) \\
\text{invar s} & \rightarrow \text{set } (\text{delete x s}) = \text{set s } \setminus \{x\} \quad (\text{delete}) \\
\text{invar s} & \rightarrow \text{isin } s \ x = (x \in \text{set s}) \quad (\text{isin})
\end{align*}
\]

**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 \( \alpha \) and \( \text{invar} \) to arbitrary types by setting \( \alpha x = x \) and \( \text{invar} x = \text{True} \) 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:

\[
\begin{align*}
invar x_1 \land \ldots \land invar x_n \rightarrow \\
\text{invar}(f x_1 \ldots x_n) \quad (f-\text{inv}) \\
\alpha(f x_1 \ldots x_n) = f_A (\alpha x_1) \ldots (\alpha x_n) \quad (f)
\end{align*}
\]

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: \( \text{empty} \) is modeled by \( \{\} \), \( \text{insert} \) is modeled by \( \lambda x. s \cup \{x\} \), \( \text{delete} \) is modeled by \( \lambda x. s - \{x\} \) and \( \text{isin} \) is modeled by \( \lambda s. x \in s \).
- Because of the artificial extension of \( \alpha \) and \text{invar} \ the above uniform format often collapses to something simpler where some \( \alpha \)'s and \text{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 \text{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 \( \text{lookup} \) is also part of the interface: it abstracts a map to a function of type \('a \Rightarrow \text{b option}. This implies that the equations are between functions of that type. We use the function update notation (see Section 1.3) to explain \( \text{update} \) and \( \text{delete}: \text{update} \) is modeled by \( \lambda m a b. m(a := b) \) and \( \text{delete} \) by \( \lambda m a. m(a := \text{None}) \).
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 ⇒ 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: `invar m → lookup m a = 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.
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 are best understood as an implementation of (a variant of) 2-3 trees. We introduce red-black trees in the next chapter. The type of 2-3 trees is similar to binary trees but with an additional constructor Node3:

```haskell
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:

```haskell
⟨⟩ ≡ Leaf
⟨l, a, r⟩ ≡ Node2 l a r
⟨l, a, m, b, r⟩ ≡ Node3 l a m b r
```

The size, height and the completeness of a 2-3 tree are defined by adding an equation for Node3 to the corresponding definitions on binary trees:

```
|⟨l, __, m, __, r⟩| = |l| + |m| + |r| + 1
```
\[ h(l, \_ , m, \_ , r) = \max(h(l))(\max(h(m))(h(r))) + 1 \]

\[ \text{complete}(l, \_ , m, \_ , r) \]
\[ = (h(l) = h(m) \land h(m) = h(r) \land \text{complete } l \land \text{complete } m \land \text{complete } r) \]

A trivial induction yields \( \text{complete } t \rightarrow 2^h t \leq |t| + 1 \) 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 \( \text{maxt} :: \text{nat} \Rightarrow \text{unit } \text{tree}23 \) that creates the tree with the largest number of nodes given the height of the tree. We use type \( \text{unit} \) because we are not interested in the elements in the tree. Prove \( \text{maxt } n = (3^n - 1) \div 2 \) 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 \( \text{nat} \) 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 \( \text{real} \) instead of \( \text{nat} \) by replacing \( \div \) by \( / \).

### 7.1 Implementation of ADT Set

The implementation will maintain the usual ordering invariant and additionally 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:

\[ \text{isin}(l, a, m, b, r) x \]
\[ = (\text{case } \text{cmp } x a \text{ of } \text{LT } \Rightarrow \text{isin } l x \mid \text{EQ } \Rightarrow \text{True} \]
\[ \mid \text{GT } \Rightarrow \text{case } \text{cmp } x b \text{ of } \text{LT } \Rightarrow \text{isin } m x \mid \text{EQ } \Rightarrow \text{True} \]
\[ \mid \text{GT } \Rightarrow \text{isin } r x) \]

Insertion into a 2-3 tree must preserve the completeness invariant. Thus recursive calls must report back to the caller if the child has “overflowed”, i.e. increased in height. Therefore insertion returns a result of type ‘\( \text{a upI} \):
7.1 Implementation of ADT Set

\[
\text{datatype}\ 'a\ upI = TI ('a\ tree23) \mid OF ('a\ tree23) 'a ('a\ tree23)
\]

This is the idea: If insertion into \( t \) returns

\( TI \ t' \) then \( t \) and \( t' \) should have the same height,
\( OF \ l \ a \ r \) 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 \( Node2 \) but in a \( Node3 \) it causes another overflow. At the root of the tree, function \( treeI \) converts values of type \( 'a\ upI \) back into trees:

\[
\begin{align*}
treeI :: \ 'a\ upI &\Rightarrow \ 'a\ tree23 \\
treeI (TI t) &= t \\
treeI (OF l a r) &= (l, a, r)
\end{align*}
\]

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 \( 'a\ upD \):

\[
\text{datatype}\ 'a\ upD = TD ('a\ tree23) \mid UF ('a\ tree23)
\]

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:

\[
\begin{align*}
treeD :: \ 'a\ upD &\Rightarrow \ 'a\ tree23 \\
treeD (TD t) &= t \\
treeD (UF t) &= t
\end{align*}
\]
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 $h_I$ on $\mathcal{a \uparrow}$ as follows:

\[
h_I (TI t) = h t
\]
\[
h_I (OF l a r) = h l
\]

The intuition is that $h_I$ is the height of the tree before insertion. A routine induction proves

\[
\text{complete } t \rightarrow \text{complete } (treeI (ins a t)) \land h_I (ins a t) = h t
\]

which implies by definition that

\[
\text{complete } t \rightarrow \text{complete } (insert a t)
\]
7.2 Preservation of Completeness

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 $\text{'a upD}$ as follows:

$$hD (\text{TD } t) = h t$$
$$hD (\text{UF } t) = h t + 1$$

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:

$$\text{complete } r \land \text{complete } (\text{treeD } l') \land h r = hD l' \quad \longrightarrow \quad \text{complete } (\text{treeD } (\text{node21 } l' a r))$$
$$0 < h r \quad \longrightarrow \quad hD (\text{node21 } l' a r) = max (hD l') (h r) + 1$$

$delete x t = \text{treeD } (\text{del } x t)$

$\text{del} : \ 'a \Rightarrow 'a \text{ tree23 } \Rightarrow 'a \text{ upD}$

$\text{del } x \langle \rangle = \text{TD } \langle \rangle$

$\text{del } x \langle a, b, \langle \rangle \rangle = (\text{if } x = a \text{ then } \text{UF } \langle \rangle \text{ else } \text{TD } \langle a, b, \langle \rangle \rangle)$

$\text{del } x \langle \langle \rangle, a, \langle \rangle \rangle = \text{TD } (\text{del } x \langle \langle \rangle, a, \langle \rangle \rangle)$

$$\begin{align*}
&= \text{TD } (\text{if } x = a \text{ then } \langle a, b, \langle \rangle \rangle \\
&\quad \text{else if } x = b \text{ then } \langle \langle \rangle, a, \langle b, \langle \rangle \rangle \rangle \\
&\quad \text{else } \langle \langle \rangle, a, b, \langle \rangle \rangle)
\end{align*}$$

$\text{del } x \langle l, a, r \rangle$

$$= (\text{case cmp } x \ a \ of \ LT \Rightarrow \text{node21 } (\text{del } x l) \ a \ r$$

$$| \ EQ \Rightarrow \text{let } (a', r') = \text{split_min } r \text{ in node22 } l \ a' \ r'$$

$$| \ GT \Rightarrow \text{node22 } l \ a \ (\text{del } x r))$$

$$\text{del } x \langle l, a, m, b, r \rangle$$

$$= (\text{case cmp } x \ a \ of \ LT \Rightarrow \text{node31 } (\text{del } x l) \ a \ m \ b \ r$$

$$| \ EQ \Rightarrow \text{let } (a', m') = \text{split_min } m \text{ in node32 } l \ a' \ m' \ b \ r$$

$$| \ GT \Rightarrow \text{case cmp } x \ b \ of \ LT \Rightarrow \text{node32 } l \ a \ (\text{del } x m) \ b \ r$$

$$| \ EQ \Rightarrow \text{let } (b', r') = \text{split_min } r \text{ in node33 } l \ a \ m \ b' \ r'$$

$$| \ GT \Rightarrow \text{node33 } l \ a \ m \ b \ (\text{del } x r))$$

$\text{split_min} : \ 'a \text{ tree23 } \Rightarrow 'a \times 'a \text{ upD}$

$\text{split_min } \langle \langle \rangle, a, \langle \rangle \rangle = (a, \text{UF } \langle \rangle)$

$\text{split_min } \langle \langle \rangle, a, \langle b, \langle \rangle \rangle \rangle = (a, \text{TD } \langle \langle \rangle, b, \langle \rangle \rangle)$

$\text{split_min } \langle l, a, r \rangle = (\text{let } (x, l') = \text{split_min } l \text{ in } (x, \text{node21 } l' \ a \ r))$

$\text{split_min } \langle l, a, m, b, r \rangle$

$$= (\text{let } (x, l') = \text{split_min } l \text{ in } (x, \text{node31 } l' \ a \ m \ b \ r))$$
Fig. 7.3. Deletion from 2-3 tree: auxiliary functions

\[
\text{split}_\text{min} \ t = (x, t') \land 0 < h \ t \land \text{complete} \ t \mapsto hD \ t' = h \ t
\]

\[
\text{split}_\text{min} \ t = (x, t') \land \text{complete} \ t \land 0 < h \ t \mapsto \text{complete} \ (t_\text{D} \ t')
\]

\[
\text{complete} \ t \mapsto hD \ (\text{del} \ x \ t) = h \ t
\]

\[
\text{complete} \ t \mapsto \text{complete} \ (t_\text{D} \ (\text{del} \ x \ t))
\]

\[
\text{complete} \ t \mapsto \text{complete} \ (\text{delete} \ x \ t)
\]

For each property of node21 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 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 data structure where insertion takes time proportional to $\lg n$. In that case inserting $n$ elements one by one takes time proportional to $\frac{1}{2}n \lg n \leq \lg (n!)$. Thus $\lg n! \geq \frac{1}{2}n \lg n \leq \lg (n!)$. Now $n! \geq (n-1)! \ldots (1) = (n!)^2$ implies $\frac{1}{2}n \lg n \leq \lg (n!)$. On the other hand, $n! \leq (n \cdot 1) \cdot ((n-1) \cdot 2) \ldots (1 \cdot n) = (n!)^2$ implies $\frac{1}{2}n \lg n \leq \lg (n!)$. Thus $\lg (n!) \geq \frac{1}{2}n \lg n \leq \lg (n!)$. We have intentionally proved a property because the $O$ property is obvious but one might hope that $\lg 1 + \ldots + \lg n$ 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 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 $\langle t_1, a_1 \rangle; \langle t_2, e_2 \rangle; \ldots; \langle t_k, a_n \rangle$. On every pass over this list, we replace adjacent triples $t, a, t'$ by $t, a, t$ 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: $t_1, e_1, t_2, e_2, \ldots, t_k$  
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 \Rightarrow nat
len (T _) = 1
len (TTs _ _ ts) = len ts + 1

trees :: 'a tree23s \Rightarrow 'a tree23 set
trees (T t) = \{t\}
trees (TTs t _ ts) = \{t\} \cup trees ts

inorder2 :: 'a tree23s \Rightarrow '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:

\[
\begin{align*}
\textit{join\_all} &:: \ 'a\tree23s \Rightarrow \ 'a\tree23 \\
\textit{join\_all} (\mathit{T} t) &= t \\
\textit{join\_all} ts &= \textit{join\_all} (\textit{join\_adj} ts)
\end{align*}
\]

\[
\begin{align*}
\textit{join\_adj} &:: \ 'a\tree23s \Rightarrow \ 'a\tree23s \\
\textit{join\_adj} (\mathit{TTs} t_1 \ a \ (\mathit{T} t_2)) &= \mathit{T} \ (<t_1, \ a, \ t_2>) \\
\textit{join\_adj} (\mathit{TTs} t_1 \ a \ (\mathit{TTs} t_2 \ b \ (\mathit{T} t_3))) &= \mathit{T} \ (<t_1, \ a, \ t_2, \ b, \ t_3>) \\
\textit{join\_adj} (\mathit{TTs} t_1 \ a \ (\mathit{TTs} t_2 \ b \ ts)) &= \mathit{TTs} \ (<t_1, \ a, \ t_2, \ b) \ (\textit{join\_adj} ts)
\end{align*}
\]

Note that \textit{join\_adj} is not and does not need to be defined on single trees. We express this precondition with an abbreviation:

\[
\textit{not\_T} ts \equiv \forall t. \ ts \neq \mathit{T} \ t
\]

Also note that \textit{join\_all} terminates only because \textit{join\_adj} shortens the list:

\[
\textit{not\_T} ts \rightarrow \text{len} \ (\textit{join\_adj} ts) < \text{len} ts
\]

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

\[
\textit{not\_T} ts \rightarrow \text{len} \ (\textit{join\_adj} ts) \leq \text{len} ts \div \mathit{2} \tag{7.1}
\]

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

\[
\begin{align*}
\textit{tree23\_of\_list} &:: \ 'a\list \Rightarrow \ 'a\tree23 \\
\textit{tree23\_of\_list} \ as &= \textit{join\_all} (\textit{leaves} \ as)
\end{align*}
\]

\[
\begin{align*}
\textit{leaves} &:: \ 'a\list \Rightarrow \ 'a\tree23s \\
\text{leaves} \ [[]] &= \mathit{T} \ () \\
\text{leaves} \ (a \ # \ as) &= \mathit{TT} () \ a \ (\text{leaves} \ as)
\end{align*}
\]

**7.3.1 Functional Correctness**

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

\[
\begin{align*}
\textit{not\_T} ts \rightarrow \text{inorder}2 \ (\textit{join\_adj} ts) &= \text{inorder}2 \ ts \\
\text{inorder} \ (\textit{join\_all} ts) &= \text{inorder}2 \ ts \\
\text{inorder} \ (\textit{tree23\_of\_list} \ as) &= as
\end{align*}
\]
(∀ t ∈ trees ts. complete t ∧ h t = n) ∧ not_ T ts →
(∀ t ∈ trees (join_adj ts). complete t ∧ h t = n + 1)
(∀ t ∈ trees ts. complete t ∧ h t = n) → complete (join_all ts)
t ∈ trees (leaves as) → complete t ∧ h t = 0
complete (tree23_of_list 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 \( \text{join}_\text{adj} \):

\[
\text{not}_\text{T ts} → T_{\text{join}_\text{adj} ts} ≤ \text{len ts div 2} \tag{7.2}
\]
An upper bound \( T_{\text{join}_\text{all} ts} ≤ 2 \cdot \text{len ts} \) follows by induction on the computa-
tion of \( \text{join}_\text{adj} \). We focus on the induction step:

\[
T_{\text{join}_\text{all} ts} = T_{\text{join}_\text{adj} ts} + T_{\text{join}_\text{all} (\text{join}_\text{adj} ts)} + 1
≤ \text{len ts div 2} + 2 \cdot \text{len (join}_\text{adj} ts) + 1 \quad \text{using (7.2) and IH}
≤ \text{len ts div 2} + 2 \cdot (\text{len ts div 2}) + 1 \quad \text{by (7.1)}
≤ 2 \cdot \text{len ts} \quad \text{because } 1 ≤ \text{len ts}
\]
Now it is routine to derive

\[
T_{\text{tree23_of_list as}} ≤ 3 \cdot |\text{as}| + 4
\]

Bibliographic Remarks

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

Red-black trees are a popular implementation technique for BSTs: they guarantee logarithmic height just like 2-3 trees but the code is arguably simpler. 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:

\[
\begin{align*}
&\langle \rangle \approx \langle \rangle \\
&\langle A, a, B \rangle \approx \langle A, a, B \rangle \\
&\langle A, a, B, b, C \rangle \approx \langle \langle A, a, B \rangle, b, C \rangle \text{ or } \langle A, a, \langle B, b, C \rangle \rangle \\
&\langle A, a, B, b, C, c, D \rangle \approx \langle \langle A, a, B \rangle, \langle b, C, c, D \rangle \rangle
\end{align*}
\]

Color expresses grouping: a black node is the root of a 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 color tag:

```haskell
datatype color = Red | Black

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

Some new syntactic sugar is sprinkled on top:
The following functions get and set the color of a node:

```
color :: 'a rbt ⇒ color
color ∅ = Black
color ∅, (__, c), ∅ = c

paint :: color ⇒ 'a rbt ⇒ 'a rbt
paint ∅ = ∅
paint c ∅, (a, __), r = ∅, (a, c), r
```

Note that the color of a leaf is by definition 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`:

```
rbt :: 'a rbt ⇒ bool
rbt t = (invc t ∧ invh t ∧ color t = Black)
```

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

```
invc :: 'a rbt ⇒ bool
invc ∅ = True
invc ∅, (__, c), ∅ = ((c = Red ⇒ color l = Black ∧ color r = Black) ∧
invc l ∧ invc r)
```

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:
8.2 Implementation of ADT Set

\[
\text{invh} :: 'a \text{ rbt} \Rightarrow \text{bool}
\]
\[
\text{invh} \; \epsilon = \text{True}
\]
\[
\text{invh} \; \langle l, (_ , _), \epsilon \rangle = (\text{bh} \; \epsilon = \text{bh} \; \epsilon \land \text{invh} \; \epsilon \land \text{invh} \; \epsilon)
\]

\[
\text{bh} :: 'a \text{ rbt} \Rightarrow \text{nat}
\]
\[
\text{bh} \; \epsilon = 0
\]
\[
\text{bh} \; \langle l, (_ , c), _ \rangle = (\text{if } c = \text{Black} \text{ then } \text{bh} \; \epsilon + 1 \text{ else } \text{bh} \; \epsilon)
\]

Note that although \text{bh} traverses only the left spine of the tree, the fact that \text{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 \text{invc} and \text{invh} improves modularity: frequently one can prove preservation of \text{invc} and \text{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. \text{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

\[
\text{invc} \; t \land \text{invh} \; t \longrightarrow \\
\text{h} \; t \leq 2 \cdot \text{bh} \; t + (\text{if } \text{color} \; t = \text{Black} \text{ then } 0 \text{ else } 1)
\]

which has the easy corollary \text{rbt} \; t \longrightarrow \text{h} \; t / 2 \leq \text{bh} \; t. Together with the easy inductive property

\[
\text{invc} \; t \land \text{invh} \; t \longrightarrow 2^{\text{bh} \; t} \leq |t|_1
\]

this implies \( 2^{\text{h} \; t} / 2 \leq 2^{\text{bh} \; t} \leq |t|_1 \) and thus \( \text{h} \; t \leq 2 \cdot \lg |t|_1 \) if \text{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 \text{rbt} invariant.

Function \text{isin} is implemented as for all augmented BSTs (see Section 5.5.1).
\[ \text{insert } x t = \text{paint Black} \ (\text{ins } x t) \]

\[
\begin{align*}
\text{ins} &:: 'a \Rightarrow 'a \ 	ext{rbt} \Rightarrow 'a \ 	ext{rbt} \\
\text{ins} \ x \ () &= R \ () \ x \ () \\
\text{ins} \ x \ (B \ l \ a \ r) &= \begin{cases} \\
LT \Rightarrow \text{baliL} \ ((\text{ins} \ x \ l) \ a \ r) \\
EQ \Rightarrow B \ l \ a \ r \\
GT \Rightarrow \text{baliR} \ l \ a \ ((\text{ins} \ x \ r)) \\
\end{cases} \\
\text{ins} \ x \ (R \ l \ a \ r) &= \begin{cases} \\
LT \Rightarrow R \ ((\text{ins} \ x \ l) \ a \ r) \\
EQ \Rightarrow R \ l \ a \ r \\
GT \Rightarrow R \ l \ a \ ((\text{ins} \ x \ r)) \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{baliL} &:: 'a \ 	ext{rbt} \Rightarrow 'a \Rightarrow 'a \ 	ext{rbt} \Rightarrow 'a \ 	ext{rbt} \\
\text{baliL} \ (R \ (R \ t_1 \ a \ t_2) \ b \ t_3) \ c \ t_4 &= R \ (B \ t_1 \ a \ t_2) \ b \ (B \ t_3 \ c \ t_4) \\
\text{baliL} \ (R \ t_1 \ a \ (R \ t_2 \ b \ t_3)) \ c \ t_4 &= R \ (B \ t_1 \ a \ t_2) \ b \ (B \ t_3 \ c \ t_4) \\
\text{baliL} \ t_1 \ a \ t_2 &= B \ t_1 \ a \ t_2
\end{align*}
\]

\[
\begin{align*}
\text{baliR} &:: 'a \ 	ext{rbt} \Rightarrow 'a \Rightarrow 'a \ 	ext{rbt} \Rightarrow 'a \ 	ext{rbt} \\
\text{baliR} \ t_1 \ a \ (R \ t_2 \ b \ (R \ t_3 \ c \ t_4)) &= R \ (B \ t_1 \ a \ t_2) \ b \ (B \ t_3 \ c \ t_4) \\
\text{baliR} \ t_1 \ a \ (R \ (R \ t_2 \ b \ t_3)) \ c \ t_4 &= R \ (B \ t_1 \ a \ t_2) \ b \ (B \ t_3 \ c \ t_4) \\
\text{baliR} \ t_1 \ a \ t_2 &= B \ t_1 \ a \ t_2
\end{align*}
\]

Fig. 8.1. Insertion into red-black tree

### 8.2.1 Insertion

Insertion is shown in Figure 8.1. The workhorse is function \( \text{ins} \). 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 \( \text{baliL}/\text{baliR} \). 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{align*}
\text{ins} \ 1 \ (B \ (R \ () \ 0 \ () \ 2 \ (R \ () \ 3 \ ())) \\
= \text{baliL} \ ((\text{ins} \ 1 \ (R \ () \ 0 \()))) \ 2 \ (R \ () \ 3 \ ())) \\
= \text{baliL} \ (R \ () \ 0 \ ((\text{ins} \ 1 \ ()))) \ 2 \ (R \ () \ 3 \ ())) \\
= \text{baliL} \ (R \ () \ 0 \ (R \ () \ 1 \ ())) \ 2 \ (R \ () \ 3 \ ())) \\
= R \ (B \ () \ 0 \ () \ 1 \ (B \ () \ 2 \ (R \ () \ 3 \ ())))
\end{align*}
\]

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 \( \text{insert} \) turns red into black.
8.2 Implementation of ADT Set

Function \( \text{ins} \) preserves \( \text{invh} \) but not \( \text{invc} \): 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 \( \text{invc2} \) as an abbreviation:

\[
\text{invc2} \ t \equiv \text{invc} (\text{paint Black } t)
\]

It is easy to prove that \( \text{baliL} \) and \( \text{baliR} \) preserve \( \text{invh} \) and upgrade from \( \text{invc2} \) to \( \text{invc} \):

\[
\begin{align*}
\text{invh} \ l & \land \text{invh} \ r \land \text{invc2} \ l \land \text{invc} \ r \land \text{bh} \ l = \text{bh} \ r \rightarrow \\
\text{invc} (\text{baliL } l \ a \ r) \land \text{invh} (\text{baliL } l \ a \ r) \land \text{bh} (\text{baliL } l \ a \ r) = \text{bh} \ l + 1 \\
\text{invh} \ l & \land \text{invc} \ r \land \text{invc} \ l \land \text{invc2} \ r \land \text{bh} \ l = \text{bh} \ r \rightarrow \\
\text{invc} (\text{baliR } l \ a \ r) \land \text{invh} (\text{baliR } l \ a \ r) \land \text{bh} (\text{baliR } l \ a \ r) = \text{bh} \ l + 1
\end{align*}
\]

Another easy induction yields

\[
\begin{align*}
\text{invc} \ t \land \text{invh} \ t \rightarrow \\
\text{invc2} (\text{ins x } t) \land (\text{color } t = \text{Black} \rightarrow \text{invc} (\text{ins x } t)) \land \\
\text{invh} (\text{ins x } t) \land \text{bh} (\text{ins x } t) = \text{bh} \ t
\end{align*}
\]

The corollary \( \text{rbt } t \rightarrow \text{rbt (insert x } t) \) 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 \( \text{invc2} \). Functions \( \text{del} \) and \( \text{split_min} \) 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.** \( \text{split_min } t = (x, t') \land t \neq () \land \text{invh } t \land \text{invc } t \rightarrow \\
\text{invh } t' \land (\text{color } t = \text{Red} \rightarrow \text{bh } t' = \text{bh } t \land \text{invc } t') \land \\
(\text{color } t = \text{Black} \rightarrow \text{bh } t' = \text{bh } t - 1 \land \text{invc2 } t')
\]

**Lemma 8.2.** \( \text{invh } t \land \text{invc } t \land t' = \text{del } x \ t \rightarrow \\
\text{invh } t' \land (\text{color } t = \text{Red} \rightarrow \text{bh } t' = \text{bh } t \land \text{invc } t') \land \\
(\text{color } t = \text{Black} \rightarrow \text{bh } t' = \text{bh } t - 1 \land \text{invc2 } t')
\]

It is easy to see that the \( \text{del}-\text{Lemma} \) implies correctness of \( \text{delete} \):

**Corollary 8.3.** \( \text{rbt } t \rightarrow \text{rbt (delete x } t) \)
delete x t = paint Black (del x t)

delete :: 'a => 'a rbt => 'a rbt
delete () = ()
delete (l, (a, _), r) = (case cmp x a of
  LT => let l' = delete 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' = delete 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))

The proofs of the two lemmas need the following precise characterizations of baldL and baldR, the counterparts of baliL and baliR:

**Lemma 8.4.** invh l ∧ invh r ∧ bh l + 1 = bh r ∧ invc2 l ∧ invc r ∧
t! = baldL l a r →
invh t' ∧ bh t' = bh r ∧ invc2 t' ∧ (color r = Black → invc t')

**Lemma 8.5.** invh l ∧ invh r ∧ bh l = bh r + 1 ∧ invc l ∧ invc2 r ∧
t! = baldR l a r →
invh t' ∧ bh t' = bh l ∧ invc2 t' ∧ (color l = Black → 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 \(del\ x\ t\). The base case is trivial. In the induction step \(t = \langle l, (a, c), r \rangle\). If \(x < a\) then we distinguish three subcases. If \(l = \langle \rangle\) the claim is trivial. Otherwise the claim follows from the IH: if \(color\ l = Red\) then the claim follows directly, if \(color\ l = Black\) then it follows with the help of Lemma 8.4 (with \(l = del\ x\ l\)). The case \(a < x\) is dual and the case \(x = a\) is similar (using Lemma 8.1). We do not show the details because they are tedious but routine.

The proof of Lemma 8.1 is similar but simpler.

### 8.2.3 Deletion by Joining

As an alternative to deletion by replacement we also consider deletion by joining (see 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 is proved by induction on the computation of \(join\):

**Lemma 8.6.** \(invh\ l \land invh\ r \land bh\ l = bh\ r \land invc\ l \land invc\ r \land\)

\(t' = join\ l\ r \rightarrow\)

\(invh\ t' \land bh\ t' = bh\ l \land invc2\ t' \land\)

(\(color\ l = Black \land color\ r = Black \rightarrow invc\ t')\)

### 8.3 Exercises

**Exercise 8.1.** Show that the logarithmic height of red-black trees is already guaranteed by the color and height invariants:

\(invc\ t \land invh\ t \rightarrow h\ t \leq 2 \cdot \lg |t| + 2\)

**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:

\(bhs \::\ ' a\ rbt \Rightarrow nat\ set\)

\(bhs\ \langle \rangle = \{0\}\)

\(bhs\ \langle l, (\_,\ c), r \rangle = (let\ H = bhs\ l \cup bhs\ r\ in\ if\ c = Black\ then\ Suc\ ' H\ else\ H)\)
where the infix operator (‘) is predefined as \( f ' A = \{ y \mid \exists x \in A. y = f x \} \).

Prove \( \mathrm{invh} \ t \leftarrow \mathrm{bhs} \ t = \{ \mathrm{bh} t \} \).

The \( \rightarrow \) direction should be easy, the other direction should need some lemmas.

**Exercise 8.3.** Following Section 7.3, define a linear time function \( \mathrm{rbt\_of\_list} :: 'a \ \text{list} \Rightarrow 'a \ \text{rbt} \) and prove both \( \text{inorder} (\mathrm{rbt\_of\_list} \ as) = \ as \) and \( \mathrm{rbt} (\mathrm{rbt\_of\_list} \ as) \).

### 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] 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 Fil-
liâtre 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
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 (ht l) − int (ht r)| ≤ 1 ∧
    n = max (ht l) (ht 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 \( \text{fib} (n + 2) \) leaves where \( \text{fib} \) is the Fibonacci function:

```plaintext
fib :: nat \Rightarrow nat
fib 0 = 0
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

**Lemma 9.1.** \( \textit{avl t} \longrightarrow \text{fib} (h t + 2) \leq |t|_1 \)

**Proof.** The proof is by induction on \( t \). We focus on the induction step \( t = \langle l, (a, n), r \rangle \) and assume \( \textit{avl t} \). Thus the IHs reduce to \( \text{fib} (h l + 2) \leq |l|_1 \) and \( \text{fib} (h r + 2) \leq |r|_1 \). We prove \( \text{fib} (\text{max} (h l) (h r) + 3) \leq |l|_1 + |r|_1 \), from which \( \textit{avl t} \longrightarrow \text{fib} (h t + 2) \leq |t|_1 \) follows directly. There are two cases. We focus on \( h l \geq h r, h l < h r \) is dual.

\[
\begin{align*}
\text{fib} (\text{max} (h l) (h r) + 3) &= \text{fib} (h l + 3) \\
&= \text{fib} (h l + 2) + \text{fib} (h l + 1) \\
&\leq |l|_1 + \text{fib} (h l + 1) \\
&\leq |l|_1 + |r|_1
\end{align*}
\]

The last step is justified because \( h l + 1 \leq h r + 2 \) (which follows from \( \textit{avl t} \)) and \( \text{fib} \) is monotone. \( \square \)

Now we prove a well-known exponential lower bound for \( \text{fib} \) where \( \varphi \equiv \left( 1 + \sqrt{5} \right) / 2 \):

**Lemma 9.2.** \( \varphi^n \leq \text{fib} (n + 2) \)

**Proof.** The proof is by induction on \( n \) by \( \text{fib} \) computation induction. The case \( n = 0 \) is trivial and the case \( n = 1 \) is easy. Now consider the induction step:
fib(n + 2 + 2) = fib(n + 2 + 1) + fib(n + 2) 
≥ \phi^n + 1 + \phi^n 
= (\phi + 1) \cdot \phi^n 
= \phi^n + 2 

by IHs

because \phi + 1 = \phi^2 \Box

Combining the two lemmas yields $avl t \rightarrow \phi^h t \leq |t|_1$ and thus

**Corollary 9.3.** $avl t \rightarrow h t \leq 1 / \lg \phi \cdot \lg |t|_1$

That is, the height of an AVL tree is at most $1 / \lg \phi \approx 1.44$ 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 \Rightarrow 'a tree ht \Rightarrow 'a tree ht
insert x () = ((), (x, 1), ())
insert x (l, (a, n), r) = (case cmp x a of
    LT \Rightarrow balL (insert x l) a r |
    EQ \Rightarrow l (a, n), r |
    GT \Rightarrow 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 $ht AB \neq ht C + 2$, then we can just return the AVL tree node $AB \ c \ C$. But if $ht AB = ht C + 2$, we need to “rotate” the subtrees suitably. Clearly $AB$ must be of the form $(A, (a, \_, ), B)$. 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 $ht B \leq ht A$ then $balL$ performs what is known as a single rotation.

If $ht A < ht B$ then $B$ must be of the form $(B_1, (b, \_, ), B_2)$ (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:**

```
        c
       / \
      a   c
     /     \
    A     B
```

```
balL
```

```
        c
       / \
      a   c
     /     \
    A     B
```

**Double rotation:**

```
        c
       / \
      a   c
     /     \
    A   b
```

```
balL
```

```
        c
       / \
      a   c
     /     \
    A   b
```

---

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 → avl (insert x t) ∧ h (insert x t) ∈ {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 \( \text{balR} / \text{balL} \) 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 \( \text{split\_max} \) performs that extraction and uses \( \text{balL} \) to restore the invariant after splitting an element off the right child.

The fact that \( \text{balR} / \text{balL} \) can be reused for deletion can be illustrated by drawing the corresponding rotation diagrams. We look at how the code for \( \text{balL} \) 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 \( \text{balL} \) after deletion in \( C \):

Double rotation in \( \text{balL} \) after deletion in \( C \):

At least one of \( B_1 \) and \( B_2 \) must have the same height as \( A \).

Preservation of \( \text{avl} \) by \( \text{delete} \) can be proved in the same manner as for \( \text{insert} \) but we provide more of the details (partly because our Isabelle proof is less automatic). The following lemmas express that the auxiliary functions preserve \( \text{avl} \):

\[
\text{avl} \ l \land \ \text{avl} \ r \land \ h \ r - 1 \leq h \ l \land h \ l \leq h \ r + 2 \rightarrow \ \text{avl} \ (\text{balL} \ l \ a \ r)
\]
9.3 Exercises

The first two are proved by the obvious cases analyses, the last one also requires induction.

As for insert, preservation of \( \text{avl} \) by delete needs to be proved simultaneously with how delete changes the height:

**Theorem 9.5.** \( \text{avl} t \land t' = \text{delete} x t \rightarrow \text{avl} t' \land h t \in \{ h t', h t' + 1 \} \)

**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 \( t' = \text{delete} x t \) and assume the IHs and \( \text{avl} t \). The claim \( \text{avl} t' \) follows from the preservation of \( \text{avl} \) by \( \text{balL}, \text{balR} \) and \( \text{split_max} \) as shown above. The claim \( h t \in \{ h t', h t' + 1 \} \) follows directly from the definitions of \( \text{balL} \) and \( \text{balR} \).

\[ \square \]

9.3 Exercises

**Exercise 9.1.** The logarithmic height of AVL trees can be proved directly. Prove

\[ \text{avl} t \land h t = n \rightarrow 2^n \div 2 \leq |t|_1 \]

by \( \text{fib} \) computation induction on \( n \). This implies \( \text{avl} t \rightarrow h t \leq 2 \cdot \lg |t|_1 \).

**Exercise 9.2.** Fibonacci trees are defined in analogy to Fibonacci numbers:

\[ \text{fibt} :: \text{nat} \Rightarrow \text{unit tree} \]

\[ \text{fibt} 0 = \langle \rangle \]

\[ \text{fibt} 1 = \langle \langle \rangle, \langle \rangle, \langle \rangle \rangle \]

\[ \text{fibt} (n + 2) = \langle \text{fibt} (n + 1), \langle \rangle, \text{fibt} n \rangle \]

We are only interested in the shape of these trees. Therefore the nodes just contain dummy unit values \( \langle \rangle \). Hence we need to define the AVL invariant again for trees without annotations:

\[ \text{avl0} :: \text{a tree} \Rightarrow \text{bool} \]

\[ \text{avl0} \langle \rangle = \text{True} \]

\[ \text{avl0} \langle l, \_ , r \rangle = (|\text{int} (h l) - \text{int} (h r)| \leq 1 \land \text{avl0} l \land \text{avl0} r) \]

Prove the following properties of Fibonacci trees:

\[ \text{avl0} (\text{fibt} n) \quad |\text{fibt} n|_1 = \text{fib} (n + 2) \]
Conclude that the Fibonacci trees are minimal (w.r.t. their size) among all AVL trees of a given height:

$$avl \ t \rightarrow |f_{ibt} (h \ t)|_1 \leq |t|_1$$

**Exercise 9.3.** Show that every almost complete tree is an AVL tree:

$$acomplete \ t \rightarrow AVL \ t$$

As in the previous exercise we consider trees without height annotations.

**Exercise 9.4.** Generalize AVL trees to height-balanced trees where the condition

$$|int (h \ l) - int (h \ r)| \leq 1$$

in the invariant is replaced by

$$|int (h \ l) - int (h \ r)| \leq m$$

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 $$avl\_of\_list :: 'a \ list \Rightarrow 'a \ tree\_ht$$ and prove both $$inorder (avl\_of\_list as) = as$$ and $$avl (avl\_of\_list as)$$.

### 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:

```plaintext
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:
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 \( \langle \rangle \) to \( \langle \rangle, (a, Bal), \langle \rangle \). This insight is encoded in the test \( \text{incr} \):

\[
\text{is\_bal} :: \text{\textquote{a tree\_bal} \Rightarrow bool}
\]
\[
\text{is\_bal} \langle \rangle, (\_ \_ b), \_ \_ = (b = Bal)
\]
\[
\text{incr} :: \text{\textquote{a tree\_bal} \Rightarrow \textquote{b tree\_bal} \Rightarrow bool}
\]
\[
\text{incr} t t' = (t = \langle \rangle \lor \text{is\_bal} t \land \neg \text{is\_bal} t')
\]

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:

\[
\text{insert} :: \text{\textquote{a \rightarrow \textquote{a tree\_bal} \Rightarrow \textquote{a tree\_bal}}}
\]
\[
\text{insert} x \langle \rangle = \langle \rangle, (x, Bal), \langle \rangle
\]
\[
\text{insert} x \langle l, (a, b), r \rangle = (\text{case} \ cmp x a \ of
\]
\[
LT \Rightarrow \text{let} \ l' = \text{insert} x \ l
\]
\[
\text{in if incr l l' then balL l' a b r else} \ l', (a, b), r \ |
\]
\[
EQ \Rightarrow \ l, (a, b), r \ |
\]
\[
GT \Rightarrow \text{let} \ r' = \text{insert} x \ r
\]
\[
\text{in if incr r r' then balR l a b r' else} \ l, (a, b), r')
\]

The balance functions are shown in Figure 9.4. Function \text{rot2} implements double rotations. Function \text{balL} is called if the left child \( AB \) has increased in height. If the tree was \( Lh \) then single or double rotations are necessary.
\[
\text{balL} :: \text{‘a tree_bal} \Rightarrow \text{‘a } \Rightarrow \text{ bal } \Rightarrow \text{‘a tree_bal} \Rightarrow \text{‘a tree_bal}
\]
\[
\text{balL AB c bc C} = (\text{case bc of}
\begin{align*}
Lh & \Rightarrow \text{ case AB of} \\
& \quad \{ (A, (a, Lh), B) \Rightarrow \{ A, (a, Bal), \{ B, (c, Bal), C \} \} | \\
& \quad \{ A, (a, Bal), B \Rightarrow \{ A, (a, Rh), \{ B, (c, Lh), C \} \} | \\
& \quad \{ A, (a, Rh), B \Rightarrow \text{ rot2 } A a B c C \} \\
Bal & \Rightarrow \{ AB, (c, Lh), C \} \\
Rh & \Rightarrow \{ AB, (c, Bal), C \}
\end{align*}
\]
\]
\[
\text{balR} :: \text{‘a tree_bal} \Rightarrow \text{‘a } \Rightarrow \text{ bal } \Rightarrow \text{‘a tree_bal} \Rightarrow \text{‘a tree_bal}
\]
\[
\text{balR A a ba BC} = (\text{case ba of}
\begin{align*}
Lh & \Rightarrow \{ A, (a, Bal), BC \} | \\
Bal & \Rightarrow \{ A, (a, Rh), BC \} | \\
Rh & \Rightarrow \text{ case BC of}
\begin{align*}
& \quad \{ B, (c, Lh), C \Rightarrow \text{ rot2 } A a B c C \} |\\
& \quad \{ B, (c, Bal), C \Rightarrow \{ (A, (a, Rh), B), (c, Lh), C \} | \\
& \quad \{ B, (c, Rh), C \Rightarrow \{ (A, (a, Bal), B), (c, Bal), C \}
\end{align*}
\]
\]
\[
\text{rot2} :: \text{‘a tree_bal} \Rightarrow \text{‘a } \Rightarrow \text{‘a tree_bal} \Rightarrow \text{‘a tree_bal} \Rightarrow \text{‘a tree_bal}
\]
\[
\text{rot2 } A a B c C = (\text{case B of}
\begin{align*}
& \quad \{ B_1, (b, bb), B_2 \} \\
& \quad \text{let } b_1 = \text{ if } bb = Rh \text{ then } Lh \text{ else } Bal; \\
& \quad b_2 = \text{ if } bb = Lh \text{ then } Rh \text{ else } Bal \\
& \quad \text{in } \{ (A, (a, b_1), B_1), (b, Bal), (B_2, (c, b_2), C) \}
\end{align*}
\]
\]

**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 \text{decr} implements this test:

\[
\text{decr :: ‘a tree_bal} \Rightarrow \text{‘b tree_bal} \Rightarrow \text{ bool}
\]
\[
\text{decr } t t’ = (t \neq \langle \rangle) \land (t’ = \langle \rangle) \lor \neg \text{ is_bal } t \land \text{ is_bal } t’
\]

The functions \text{incr} and \text{decr} are almost dual except that \text{incr} implicitly assumes \(t’ \neq \langle \rangle\) because insertion is guaranteed to return a Node. Thus we could use \text{decr} instead of \text{incr} but not the other way around.

Deletion and \text{split_max} change in the same manner as insertion:
\[
\text{delete} :: 'a \Rightarrow 'a\text{ tree_bal} \Rightarrow 'a\text{ tree_bal}
\]
\[
\text{delete}_{\text{\_}}(\text{\_}) = (\text{\_})
\]
\[
delete x (l, (a, ba), r)
= \begin{cases}
\text{case cmp } x \text{ a of} \\
\quad \text{LT } \Rightarrow \text{ let } l' = \text{ delete } x l \\
\quad \quad \text{in if } \text{ decr } l l' \text{ then } \text{ balR } l'\ a\ ba\ r\ \text{ else } (l', (a, ba), r) \\
\quad \text{EQ } \Rightarrow \text{ if } l = (\text{\_}) \text{ then } r \\
\quad \quad \text{else let } (l', a') = \text{ split_max } l \\
\quad \quad \quad \text{in if } \text{ decr } l l' \text{ then } \text{ balR } l'\ a'\ ba\ r \\
\quad \quad \quad \quad \text{else } (l', (a', ba), r) \\
\quad \text{GT } \Rightarrow \text{ let } r' = \text{ delete } x r \\
\quad \quad \text{in if } \text{ decr } r r' \text{ then } \text{ balL } l\ a\ ba\ r'\ \text{ else } (l, (a, ba), r'))
\end{cases}
\]

\[
\text{split_max} :: 'a\text{ tree_bal} \Rightarrow 'a\text{ tree_bal} \times 'a
\]
\[
\text{split_max}(l, (a, ba), r)
= \begin{cases}
\text{if } r = (\text{\_}) \text{ then } (l, a) \\
\text{else let } (r', a') = \text{ split_max } r; \\
\text{ in } (t', a')
\end{cases}
\]

In the end we have the following correctness theorems:

**Theorem 9.6.** \(\text{avl } t \land t' = \text{ insert } x t \longrightarrow \)

\(\text{avl } t' \land h t' = h t + (\text{if } \text{ incr } t t' \text{ then } 1 \text{ else } 0)\)

This theorem tells us not only that \(\text{avl}\) is preserved but also that \(\text{incr}\) indicates correctly if the height has increased or not.

Similarly for deletion and \(\text{decr}\):

**Theorem 9.7.** \(\text{avl } t \land t' = \text{ delete } x t \longrightarrow \)

\(\text{avl } t' \land h t' = h t + (\text{if } \text{ decr } t t' \text{ then } 1 \text{ else } 0)\)

The proofs of both theorems follow the standard pattern of induction followed by an exhaustive (automatic) cases analysis. The proof for \(\text{delete}\) requires an analogous lemma for \(\text{split_max}\):

\[
\text{split_max } t = (t', a) \land \text{avl } t \land t \neq (\text{\_}) \longrightarrow \)

\(\text{avl } t' \land h t = h t' + (\text{if } \text{ decr } t t' \text{ then } 1 \text{ else } 0)\)

### 9.5 Exercises

**Exercise 9.6.** We map type \('a\text{ tree_bal}\) back to type \(('a \times \text{nat})\text{ tree}\) called \('a\text{ tree_ht}\) in the beginning of the chapter:
Prove that the AVL property is preserved: \( \text{avl } t \Rightarrow \text{avl}_{\text{ht}}(\text{debal } t) \) where \( \text{avl}_{\text{ht}} \) is the \( \text{avl} \) predicate on type \( 'a \text{ tree}_{\text{ht}} \) from the beginning of the chapter.

Define a function \( \text{debal2} \) of the same type that traverses the tree only once and in particular does not use function \( h \). Prove \( \text{avl } t \Rightarrow \text{debal2 } t = \text{debal } t \).

**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:

```plaintext
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.
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 $\backslash$.

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 $m \leq n$. 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.
10 Beyond Insert and Delete: ∪, ∩ and −

- 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 ⇒ 's ⇒ 's
inter :: 's ⇒ 's ⇒ 's
diff :: 's ⇒ 's ⇒ 's

**specification**

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{invar} (\text{union } s_1 \ s_2) \) \quad \text{(union-inv)}

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{set} (\text{union } s_1 \ s_2) = \text{set } s_1 \cup \text{set } s_2 \) \quad \text{(union)}

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{invar} (\text{inter } s_1 \ s_2) \) \quad \text{(inter-inv)}

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{set} (\text{inter } s_1 \ s_2) = \text{set } s_1 \cap \text{set } s_2 \) \quad \text{(inter)}

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{invar} (\text{diff } s_1 \ s_2) \) \quad \text{(diff-inv)}

\( \text{invar } s_1 \land \text{invar } s_2 \rightarrow \text{set} (\text{diff } s_1 \ s_2) = \text{set } s_1 - \text{set } s_2 \) \quad \text{(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 × 'b) tree where 'a is the type of the elements and 'b is the balancing information (which we can ignore
10.2 Just Join

...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:

\[
\begin{align*}
\text{join} &:: (\mathcal{B} \times \mathcal{B}) \text{ tree} \Rightarrow \mathcal{B} \Rightarrow (\mathcal{B} \times \mathcal{B}) \text{ tree} \\
\text{inv} &:: (\mathcal{B} \times \mathcal{B}) \text{ tree} \Rightarrow \text{bool}
\end{align*}
\]

Function \text{inv} is meant to take care of the balancedness property only, not the BST property. Functions \text{join} and \text{inv} are specified with the help of the standard tree functions \text{set_tree} and \text{bst} in Figure 10.2. With respect to the

\[
\begin{align*}
\text{set_tree} (\text{join} \ l \ a \ r) &= \text{set_tree} \ l \cup \{a\} \cup \text{set_tree} \ r \quad (10.1) \\
\text{bst} \ (l, (a, \_), r) &\rightarrow \text{bst} \ (\text{join} \ l \ a \ r) \quad (10.2) \\
\text{inv} () &\rightarrow \text{inv} (\text{join} \ l \ a \ r) \quad (10.3) \\
\text{inv} (l, (\_, \_), r) &\rightarrow \text{inv} \ l \land \text{inv} \ r \quad (10.4)
\end{align*}
\]

Fig. 10.2. Specification of \text{join} and \text{inv}

...set of elements, \text{join} must behave like union. But it need only return a BST if both trees are BSTs and the element \text{a} lies in between the elements of the two trees, i.e. if \text{bst} (l, (a, \_), r). The structural invariant \text{inv} must be preserved by formation and destruction of trees. Thus we can see \text{join} as a smart constructor that builds a balanced tree.

To define union and friends we need a number of simple auxiliary functions shown in Figure 10.3. Function \text{split_min} decomposes a tree into its leftmost (minimal) element and the remaining tree; the remaining tree is reassembled via \text{join}, thus preserving \text{inv}. Function \text{join2} is reduced to \text{join} with the help of \text{split_min}. Function \text{split} splits a BST w.r.t. a given element \text{a} into a triple (\text{l}, \text{b}, \text{r}) such that \text{l} contains the elements less than \text{a}, \text{r} contains the elements greater than \text{a}, and \text{b} is true iff \text{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:

\[
\begin{align*}
\text{insert} &:: 'a \Rightarrow (\mathcal{B} \times \mathcal{B}) \text{ tree} \Rightarrow (\mathcal{B} \times \mathcal{B}) \text{ tree} \\
\text{insert} \ x \ t &= (\text{let} \ (l, b, r) = \text{split} \ t \ x \ \text{in} \ \text{join} \ l \ x \ r) \\
\text{delete} &:: 'a \Rightarrow (\mathcal{B} \times \mathcal{B}) \text{ tree} \Rightarrow (\mathcal{B} \times \mathcal{B}) \text{ tree} \\
\text{delete} \ x \ t &= (\text{let} \ (l, b, r) = \text{split} \ t \ x \ \text{in} \ \text{join2} \ l \ r)
\end{align*}
\]

The efficiency can be improved a little by taking the returned \text{b} into account.
\begin{verbatim}
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 (l_1, b, l_2) = split l x in (l_1, b, join l_2 a r) |
  EQ ⇒ (l, True, r) |
  GT ⇒ let (r_1, b, r_2) = split r x in (join l a r_1, b, r_2))
\end{verbatim}

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:

\(\text{split}_\text{min} \ t = (m, t') \land t ≠ () \quad \rightarrow\)
\(m \in set_tree \ t \land set_tree \ t = \{m\} \cup set_tree \ t'\)

\(\text{split}_\text{min} \ t = (m, t') \land bst \ t \land t ≠ () \quad \rightarrow\)
\(bst t' \land (∀ x \in set_tree \ t'. \ m < x)\)

\(\text{split}_\text{min} \ t = (m, t') \land inv \ t \land t ≠ () \quad \rightarrow inv t'\)
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))

ter :: ('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))

set_tree (join2 l r) = set_tree l ∪ set_tree r
bst l ∧ bst r ∧ (∀x ∈ set_tree l. ∀y ∈ set_tree r. x < y) −→
bst (join2 l r)
inv l ∧ inv r −→ inv (join2 l r)

split t x = (l, b, r) ∧ bst t −→
set_tree l = {a ∈ set_tree t | a < x} ∧
set_tree r = {a ∈ set_tree t | x < a} ∧
b = (x ∈ set_tree t) ∧ bst l ∧ bst r
split t x = (l, b, r) ∧ inv t −→ inv l ∧ 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:

bst t₂ −→ set_tree (union t₁ t₂) = set_tree t₁ ∪ set_tree t₂
bst t₁ ∧ bst t₂ −→ bst (union t₁ t₂)
inv t₁ ∧ inv t₂ −→ inv (union t₁ t₂)
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:

\[
\begin{align*}
\text{bst } t_1 & \wedge \text{bst } t_2 \rightarrow \text{set_tree } (\text{inter } t_1 \ t_2) = \text{set_tree } t_1 \cap \text{set_tree } t_2 \\
\text{bst } t_1 & \wedge \text{bst } t_2 \rightarrow \text{bst } (\text{inter } t_1 \ t_2) \\
\text{inv } t_1 & \wedge \text{inv } t_2 \rightarrow \text{inv } (\text{inter } t_1 \ t_2)
\end{align*}
\]

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 \( L_1 = \text{set_tree } l_1 \) and \( R_1 = \text{set_tree } r_1 \). Let \( (l_2, b, r_2) = \text{split } t_2 a, L_2 = \text{set_tree } l_2, R_2 = \text{set_tree } r_2 \) and \( A = (\text{if } b \text{ then } \{a\} \text{ else } \{\}) \). The separation properties

\[
\begin{align*}
a \notin L_1 & \cup R_1 \quad a \notin L_2 & \cup R_2 \\
L_2 & \cap R_2 = \{\} \quad L_1 & \cap R_2 = \{\} \quad L_2 & \cap R_1 = \{\}
\end{align*}
\]

follow from bst \( t_1 \), bst \( t_2 \) and (10.6). Now for the main proof:

\[
\begin{align*}
\text{set_tree } t_1 & \cap \text{set_tree } t_2 \\
= (L_1 \cup R_1 \cup \{a\}) & \cap (L_2 \cup R_2 \cup A) & \text{by (10.6), bst } t_2 \\
= L_1 & \cap L_2 \cup R_1 \cap R_2 \cup A & \text{by the separation properties} \\
= \text{set_tree } (\text{inter } t_1 \ t_2) & \text{by (10.1), (10.5), IHs, bst } t_1, \ bst t_2, (10.6)
\end{align*}
\]

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 \( B \ l \ x \ r \).
10.3 Joining Red-Black Trees

\[ \text{joinL} :: \text{rbt} \to \text{rbt} \to \text{rbt} \]
\[ \text{joinL} \ x \ r = (\text{if } \text{bh} \ r \leq \text{bh} \ l \ \text{then} \ \text{R} \ x \ r) \]
\[ \text{else case } r \ of \]
\[ (l', (x', \text{Red}), r') \to \text{R} (\text{joinL} \ l \ l') \ x' \ r' | \]
\[ (l', (x', \text{Black}), r') \to \text{baliL} (\text{joinL} \ l \ l') \ x' \ r' \]

\[ \text{joinR} :: \text{rbt} \to \text{rbt} \to \text{rbt} \]
\[ \text{joinR} \ x \ r = (\text{if } \text{bh} \ l \leq \text{bh} \ r \ \text{then} \ \text{R} \ l \ r) \]
\[ \text{else case } l \ of \]
\[ (l', (x', \text{Red}), r') \to \text{R} \ l' \ x' (\text{joinR} \ r' \ x \ r) | \]
\[ (l', (x', \text{Black}), r') \to \text{baliR} \ l' \ x' (\text{joinR} \ r' \ x \ r) \]

\[ \text{join} :: \text{rbt} \to \text{rbt} \to \text{rbt} \]
\[ \text{join} \ l \ r = (\text{if } \text{bh} \ r < \text{bh} \ l \ \text{then} \ \text{paint Black} (\text{joinR} \ l \ x \ r) \]
\[ \text{else if } \text{bh} \ l < \text{bh} \ r \ \text{then} \ \text{paint Black} (\text{joinL} \ l \ x \ r) \ \text{else} \ \text{B} \ l \ x \ r) \]

Fig. 10.5. Function \text{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 \text{joinR} (and similarly in \text{joinL}) the comparison is not \text{bh} \ l = \text{bh} \ r
but \text{bh} \ l \leq \text{bh} \ r to simplify the proofs.

10.3.1 Correctness

We need to prove that \text{join} on red-black trees (and a suitable \text{inv}) satisfies its
specification in Figure 10.2. We start with properties of \text{joinL}; the properties
of function \text{joinR} are completely symmetric. These are the automatically
provable inductive properties:

\[ \text{invc} \ l \land \text{invc} \ r \land \text{invh} \ l \land \text{invh} \ r \land \text{bh} \ l \leq \text{bh} \ r \to \]
\[ (\text{bh} \ l \neq \text{bh} \ r \land \text{color} \ r = \text{Black} \to \text{invc} (\text{joinL} \ l \ x \ r)) \land \]
\[ (\text{invh} (\text{joinL} \ l \ x \ r) \land \text{bh} (\text{joinL} \ l \ x \ r) = \text{bh} \ r \]
\[ \text{bh} \ l \leq \text{bh} \ r \to \text{set_tree} (\text{joinL} \ l \ x \ r) = \text{set_tree} \ l \cup \{x\} \cup \text{set_tree} \ r \]
\[ \text{bst} (l, (a, \ n), \ r) \land \text{bh} \ l \leq \text{bh} \ r \to \text{bst} (\text{joinL} \ l \ a \ r) \]

Because \text{joinL} employs \text{baliL} from the chapter on red-black trees, the proof
of the first property makes use of the \text{baliL} property shown in Section 8.2.1.
120 10 Beyond Insert and Delete: ∪, ∩ and −

We define the invariant inv required for the specification in Figure 10.2 as follows:

\[ \text{inv } t = (\text{invc } t \land \text{inv } h t) \]

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 \( \text{diffL} \) of \( \text{diff} \) where in the third equation pattern matching is on \( t_1 \) and \( t_2 \) is split. Prove \( \text{bst } t_1 \land \text{bst } t_2 \rightarrow \text{set_tree } (\text{diffL } t_1 t_2) = \text{set_tree } t_1 - \text{set_tree } t_2 \).

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 \( \text{joinL, joinR} :: 'a tree23 \Rightarrow 'a \Rightarrow 'a tree23 \Rightarrow 'a upI \) and combine them into the overall join function:

\[ \text{join} :: 'a tree23 \Rightarrow 'a \Rightarrow 'a tree23 \Rightarrow 'a tree23 \]

Prove the following correctness properties:

\[ \text{complete } l \land \text{complete } r \rightarrow \text{complete } (\text{join } l x r) \]
\[ \text{complete } l \land \text{complete } r \rightarrow \text{inorder } (\text{join } l x r) = \text{inorder } l \oplus x \neq \text{inorder } r \]

The corresponding (and needed) properties of \( \text{joinL} \) and \( \text{joinR} \) 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 \( O(m \log(\frac{n}{m}) + 1) \) bound for the work (and an \( O(\log m \log n) \) 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:** \(xs \text{ ! } n\) is the \(n\)th element of the list \(xs\).

**Updating:** \(xs|n := x\) is \(xs\) with the \(n\)th element replaced by \(x\).

By convention, indexing starts with \(n = 0\). If \(n \geq \text{length } xs\) then \(xs \text{ ! } n\) 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 Array =

interface
lookup :: 'ar ⇒ nat ⇒ 'a
update :: nat ⇒ 'a ⇒ 'ar ⇒ 'ar
len :: 'ar ⇒ nat
array :: 'a list ⇒ 'ar
abstraction list :: 'ar ⇒ 'a list
invariant invar :: 'ar ⇒ bool

specification
invar ar ∧ n < len ar ⇒ lookup ar n = list ar ! n
(lookup)
invar ar ∧ n < len ar ⇒ invar (update n x ar)
(update-inv)
invar ar ∧ n < len ar ⇒ list (update n x ar) = (list ar)[n := x]
(update)
invar ar ⇒ 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 \( \geq 1 \) 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.
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:

\[ \begin{array}{c}
1 \\
\downarrow \\
3
\end{array} \]

It turns out that the following invariant guarantees that a tree \( t \) contains exactly the nodes indexed by \( 1, ..., |t| \):

\[
\text{braun} :: \forall t. \text{a tree} \Rightarrow \text{bool}
\]

\[
\text{braun} \langle \rangle = \text{True}
\]

\[
\text{braun} \langle l, _., r \rangle = (|l| = |r| \lor |l| = |r| + 1) \land \text{braun} l \land \text{braun} r
\]

The disjunction can alternatively be expresses as \( |r| \leq |l| \leq |r| + 1 \). We call a tree a **Braun tree** iff it satisfies predicate \( \text{braun} \).

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.**

\( a\text{complete} \land a\text{complete} \land |l| = |r| + 1 \longrightarrow a\text{complete} \langle l, x, r \rangle \)

**Proof.** Using Lemmas 4.7 and 4.8 and the assumptions we obtain

\[
\begin{align*}
h \langle l, x, r \rangle &= \lceil \log (|r|_1 + 1) \rceil + 1 \\
mh \langle l, x, r \rangle &= \lceil \log |r|_1 \rceil + 1
\end{align*}
\]

Because \( 1 \leq |r|_1 \), there is an \( i \) such that \( 2^i \leq |r|_1 < 2^i + 1 \) and thus \( 2^i < |r|_1 + 1 \leq 2^{i+1} + 1 \). This implies \( i = \lceil \log |r|_1 \rceil \) and \( i + 1 = \lceil \log (|r|_1 + 1) \rceil \). Together with \((*)\) and \((**)\) this implies \( a\text{complete} \langle l, x, r \rangle \). \( \square \)
Now we can show that all Braun trees are almost complete:

**Lemma 11.2.** braun $t \rightarrow$ 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 acomplete $l$ and acomplete $r$. Because of braun $t$ we can distinguish two cases. First assume $|l| = |r| + 1$. The claim acomplete $t$ follows immediately from the previous lemma. Now assume $|l| = |r|$. By definition, there are four cases to consider when proving acomplete $t$. By symmetry it suffices to consider only two of them. If $h l \leq h r$ and $mh r < mh l$ then acomplete $t$ reduces to acomplete $r$, which is true by assumption. Now assume $h l \leq h r$ and $mh l \leq mh r$. 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 acomplete $t$ reduces to 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:

```plaintext
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 $\text{div} \ 2$. The function is called `lookup1` 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 `update1` descends in the very same manner but also performs an update when reaching 1:
11.3 Arrays via Braun Trees

update₁ :: nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree
update₁ _ x () = ⟨⟩, x, ⟨⟩
update₁ n x ⟨i, a, r⟩ = (if n = 1 then ⟨i, 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, \( \text{update₁} \ (|t| + 1) \times t \) extends the tree with a new node \( x \) at index \( |t| + 1 \).

Function \( \text{adds} \) iterates this process (again expecting \( |t| + 1 \) as the index) and thus adds a whole list of elements:

\[
\begin{align*}
\text{adds} :: 'a list ⇒ nat ⇒ 'a tree ⇒ 'a tree \\
\text{adds} [\ ] _ t = t \\
\text{adds} (x \# xs) n t = \text{adds} xs (n + 1) \ (\text{update₁} \ (n + 1) \times t)
\end{align*}
\]

Fig. 11.3. Array implementation via Braun trees

11.3.1 Functional Correctness

The invariant on arrays is obvious:
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):

\[
list\ t = \text{map}\ (\text{lookup}\ t)\ [1..<|t| + 1]
\]

Instead we define list recursively and derive the above equation later on.

\[
\text{list} :: \ 'a\ \text{tree} \Rightarrow \ 'a\ \text{list}
\]

\[
\text{list}\ \langle\ \rangle = \ [\]
\]

\[
\text{list}\ \langle\ l,\ x,\ r\rangle = x\ #\ \text{splice}\ (\text{list}\ l)\ (\text{list}\ r)
\]

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:

\[
\text{splice} :: \ 'a\ \text{list} \Rightarrow \ 'a\ \text{list} \Rightarrow \ 'a\ \text{list}
\]

\[
\text{splice}\ [\]\ ys = ys
\]

\[
\text{splice}\ (x\ #\ xs)\ ys = x\ #\ \text{splice}\ ys\ xs
\]

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:

\[
\text{braun}\ \langle\ l,\ x,\ r\rangle\ \wedge\ n\ \in\ \{1..|\langle\ l,\ x,\ r\rangle|\}\ \wedge\ 1 < n\ \rightarrow
\]

\[
(\text{odd}\ n\ \rightarrow\ n\ \text{div}\ 2\ \in\ \{1..|r|\})\ \wedge\ (\text{even}\ n\ \rightarrow\ n\ \text{div}\ 2\ \in\ \{1..|l|\})
\]

where \(\{m..n\} = \{k\mid m \leq k \wedge k \leq m\}\). 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 \(\text{len}\), the correctness of function len. Because of the invariant, \(\text{len}\) follows directly from

\[
|\text{list}\ t| = |t|
\]

which is proved by induction. We will also use this property implicitly in many proofs below.

The following proposition implies the correctness property \(\text{lookup}\):
11.3 Arrays via Braun Trees

\[ \text{braun} \ t \land i < |t| \longrightarrow \text{list} \ t ! i = \text{lookup} \ 1 \ t \ (i + 1) \] (11.1)

The proof is by induction and uses the following proposition that is also proved by induction:

\[ n < |xs| + |ys| \land |ys| \leq |xs| \land |xs| \leq |ys| + 1 \longrightarrow \text{splice} \ xs \ ys \ ! n = (\text{if even} \ n \ \text{then} \ xs \ \text{else} \ ys) ! (n \ \text{div} \ 2) \]

As a corollary to (11.1) we obtain that function \text{list} can indeed be expressed via \text{lookup}:

\[ \text{braun} \ t \longrightarrow \text{list} \ t = \text{map} \ (\text{lookup} \ 1 \ t) \ [1..<|t| + 1] \] (11.2)

It follows by \text{list extensionality}:

\[ xs = ys \iff |xs| = |ys| \land (\forall i < |xs|. \ xs ! i = ys ! i) \]

Let us now verify \text{update} as implemented via \text{update1}. The following two preservation properties (proved by induction) prove (\text{update-inv}):

\[ \text{braun} \ t \land n \in \{1..|t|\} \longrightarrow |\text{update} \ 1 \ n \ x \ t| = |t| \]
\[ \text{braun} \ t \land n \in \{1..|t|\} \longrightarrow \text{braun} \ (\text{update} \ 1 \ n \ x \ t) \]

The following property relating \text{lookup}1 and \text{update1} is again proved by induction:

\[ \text{lookup} \ 1 \ (\text{update} \ 1 \ n \ x \ t) \ m = (\text{if} \ n = m \ \text{then} \ x \ \text{else} \ \text{lookup} \ 1 \ t \ m) \]

The last three properties together with (11.2) and list extensionality prove the following proposition, which implies (\text{update}):

\[ \text{braun} \ t \land n \in \{1..|t|\} \longrightarrow \text{list} \ (\text{update} \ 1 \ n \ x \ t) = (\text{list} \ t)[n - 1 := x] \]

Finally we turn to the constructor \text{array}. It is implemented in terms of \text{adds} and \text{update1}. Their correctness is captured by the following properties whose inductive proofs build on each other:

\[ \text{braun} \ t \longrightarrow |\text{update} \ 1 \ (|t| + 1) \ x \ t| = |t| + 1 \] (11.3)
\[ \text{braun} \ t \longrightarrow \text{braun} \ (\text{update} \ 1 \ (|t| + 1) \ x \ t) \] (11.4)
\[ \text{braun} \ t \longrightarrow \text{list} \ (\text{update} \ 1 \ (|t| + 1) \ x \ t) = \text{list} \ t @ [x] \] (11.5)
\[ \text{braun} \ t \longrightarrow |\text{adds} \ xs \ |t| \ t| = |t| + |xs| \land \text{braun} \ (\text{adds} \ xs \ |t| \ t) \]
\[ \text{braun} \ t \longrightarrow \text{list} \ (\text{adds} \ xs \ |t| \ t) = \text{list} \ t @ xs \]

The last two properties imply the remaining proof obligations (\text{array-inv}) and (\text{array}). The proof of (11.5) requires the following two properties of \text{splice}

\[ |ys| \leq |xs| \longrightarrow \text{splice} \ (xs @ [x]) \ ys = \text{splice} \ xs \ ys @ [x] \]
\[ |xs| \leq |ys| + 1 \longrightarrow \text{splice} \ xs \ (ys @ [y]) = \text{splice} \ xs \ ys @ [y] \]
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 \( O(n \log n) \). 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 \( O(n \log n) \) follows.

Function array is implemented via adds and thus via repeated calls of update1. At the beginning of Section 7.3 we show that because update1 has logarithmic complexity, calling it \( n \) times on a growing tree starting with a leaf takes time \( \Theta(n \log n) \).

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.

\[
\textbf{ADT} \quad \text{Array_Flex} = \text{Array} +
\]

\begin{align*}
\text{interface} & \\
\text{add_lo :: 'a \Rightarrow 'ar \Rightarrow 'ar} & \\
\text{del_lo :: 'ar \Rightarrow 'ar} & \\
\text{add_hi :: 'a \Rightarrow 'ar \Rightarrow 'ar} & \\
\text{del_hi :: 'ar \Rightarrow 'ar} & \\
\end{align*}

\begin{align*}
\text{specification} & \\
\text{invar ar} & \rightarrow \text{invar (add_lo a ar)} & (\text{add_lo-inv}) \\
\text{invar ar} & \rightarrow \text{list (add_lo a ar) = a \# list ar} & (\text{add_lo}) \\
\text{invar ar} & \rightarrow \text{invar (del_lo ar)} & (\text{del_lo-inv}) \\
\text{invar ar} & \rightarrow \text{list (del_lo ar) = tl (list ar)} & (\text{del_lo}) \\
\text{invar ar} & \rightarrow \text{invar (add_hi a ar)} & (\text{add_hi-inv}) \\
\text{invar ar} & \rightarrow \text{list (add_hi a ar) = list ar \@ [a]} & (\text{add_hi}) \\
\text{invar ar} & \rightarrow \text{invar (del_hi ar)} & (\text{del_hi-inv}) \\
\text{invar ar} & \rightarrow \text{list (del_hi ar) = butlast (list ar)} & (\text{del_hi}) \\
\end{align*}

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 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:

\[
del\_hi :: \text{nat} \Rightarrow \text{a tree} \Rightarrow \text{a tree}
\]

\[
del\_hi \_ \langle \rangle = \langle \rangle
\]

\[
del\_hi n \langle l, x, r \rangle = (\text{if } n = 1 \text{ then } \langle \rangle \text{ else if } \text{even } n \text{ then } (del\_hi (n \div 2) l, x, r) \text{ else } \langle l, x, del\_hi (n \div 2) r \rangle)
\]

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:

\[
add\_lo :: \text{a} \Rightarrow \text{a tree} \Rightarrow \text{a tree}
\]

\[
add\_lo x \langle \rangle = \langle \langle \rangle, x, \langle \rangle \rangle
\]

\[
add\_lo x \langle l, a, r \rangle = \langle add\_lo a r, x, l \rangle
\]

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.

\[
\begin{array}{c}
1 \\
2 \\
4 \\
\end{array}
\leftarrow add\_lo 0,
\begin{array}{c}
0 \\
1 \\
3 \\
\end{array}
\begin{array}{c}
2 \\
5 \\
6 \\
\end{array}
\]

Function del_lo simply reverses add_lo by removing the root and merging the children:
Arrays via Braun Trees

\[
\text{del}_{\text{lo}} :: \text{'tree} \Rightarrow \text{'tree} \\
\text{del}_{\text{lo}} \langle \rangle = \langle \rangle \\
\text{del}_{\text{lo}} \langle \ell, \_, \, r \rangle = \text{merge} \, l \, r
\]

\[
\text{merge} :: \text{'tree} \Rightarrow \text{'tree} \Rightarrow \text{'tree} \\
\text{merge} \langle \rangle \, r = r \\
\text{merge} \langle \ell, \, a, \, r \rangle \, rr = \langle rr, \, a, \text{merge} \, l \, r \rangle
\]

Figure 11.5 shows the obvious implementation of the functions in the \textit{Array}._\text{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 \textit{Array}._\text{Flex} functions have logarithmic time complexity because the corresponding Braun tree functions do because they descend along one branch of the tree.

\[
\text{add}_{\text{lo}} \, x \, (t, \, l) = (\text{add}_{\text{lo}} \, x \, t, \, l + 1) \\
\text{del}_{\text{lo}} \, (t, \, l) = (\text{del}_{\text{lo}} \, t, \, l - 1) \\
\text{add}_{\text{hi}} \, x \, (t, \, l) = (\text{update} \, (l + 1) \, x \, t, \, l + 1) \\
\text{del}_{\text{hi}} \, (t, \, l) = (\text{del}_{\text{hi}} \, l \, t, \, l - 1)
\]

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 \text{update} and thus \text{add}_{\text{hi}} above. Properties (\text{add}_{\text{hi-inv}}) and (\text{add}_{\text{hi}}) follow from (11.3), (11.4) and (11.5) stated earlier.

Correctness of \text{del}_{\text{hi}} on Braun trees is captured by the following two properties proved by induction:

\[
\text{braun} \, t \rightarrow \text{braun} \, (\text{del}_{\text{hi}} \mid t) \, t
\]
\[
\text{braun} \, t \rightarrow \text{list} \, (\text{del}_{\text{hi}} \mid t) \, t = \text{butlast} \, (\text{list} \, t)
\] (11.6)

They imply (\text{del}_{\text{hi-inv}}) and (\text{del}_{\text{hi}}). The proof of (11.6) requires the simple fact \text{list} \, t = \langle \rangle \leftrightarrow t = \langle \rangle and the following property of \text{splice} which is proved by induction:
\[ \text{butlast} \ (\text{splice} \ xs \ ys) = (\text{if} \ |ys| < |xs| \ \text{then} \ \text{splice} \ (\text{butlast} \ xs) \ ys \ \text{else} \ \text{splice} \ xs \ (\text{butlast} \ ys)) \]

Correctness of \( \text{add}_lo \) on Braun trees is captured by the following two properties proved by induction:

\[ \text{braun} \ t \rightarrow \text{braun} \ (\text{add}_lo \ x \ t) \]
\[ \text{braun} \ t \rightarrow \text{list} \ (\text{add}_lo \ a \ t) = a \ \# \ \text{list} \ t \]

Properties \( \text{(add}_lo\text{-inv}) \) and \( \text{(add}_lo) \) follow directly from them.

Finally we turn to \( \text{del}_lo \). Inductions (for \( \text{merge} \)) and case analyses (for \( \text{del}_lo \)) yield the following properties:

\[ \text{braun} \ (l, x, r) \rightarrow \text{braun} \ (\text{merge} \ l \ r) \]
\[ \text{braun} \ (l, x, r) \rightarrow \text{list} \ (\text{merge} \ l \ r) = \text{splice} \ (\text{list} \ l) \ (\text{list} \ r) \]
\[ \text{braun} \ t \rightarrow \text{braun} \ (\text{del}_lo \ t) \]
\[ \text{braun} \ t \rightarrow \text{list} \ (\text{del}_lo \ t) = \text{tl} \ (\text{list} \ t) \]

The last two properties imply \( \text{(del}_lo\text{-inv}) \) and \( \text{(del}_lo) \).

### 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:

\[ \text{size}_fast :: \mathbf{'} \text{a} \ \text{tree} \Rightarrow \mathbf{nat} \]
\[ \text{size}_fast \ () = 0 \]
\[ \text{size}_fast \ (l, \_, \_, r) = (\text{let} \ n = \text{size}_fast \ r \ \text{in} \ 1 + 2 \cdot n + \text{diff} \ l \ n) \]

\[ \text{diff} :: \mathbf{'} \text{a} \ \text{tree} \Rightarrow \mathbf{nat} \Rightarrow \mathbf{nat} \]
\[ \text{diff} \ () \_ = 0 \]
\[ \text{diff} \ (l, \_, \_, r) \ n \]
\[ = (\text{if} \ n = 0 \ \text{then} \ 1 \ \text{else} \ if \ \text{even} \ n \ \text{then} \ \text{diff} \ r \ (n \ \text{div} \ 2 - 1) \ \text{else} \ \text{diff} \ l \ (n \ \text{div} \ 2)) \]
Function \textit{size\_fast} descends down the right spine, computes the size of a \textit{Node} as if both children were the same size ($1 + 2 \cdot n$), but adds $\text{diff}\_l\ n$ to compensate for bigger left children. Correctness of \textit{size\_fast}

\textbf{Lemma 11.3.} $\text{braun\ } t \rightarrow \text{size\_fast}\ t = |t|$

follows from this property of $\text{diff}$:

$$\text{braun\ } t \land |t| \in \{n, n + 1\} \rightarrow \text{diff}\ t\ n = |t| - n$$

The running time of \textit{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 \textit{update1} $n$ times, but one can also build the tree directly:

\begin{verbatim}
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))
\end{verbatim}

This solution also has time complexity $O(n \lg n)$ but it can clearly be improved by sharing identical recursive calls. Function \textit{braun2\_of} shares as much as possible by producing trees of size $n$ and $n + 1$ in parallel:

\begin{verbatim}
braun2\_of :: 'a \Rightarrow \text{nat} \Rightarrow 'a\ tree \times '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)))
\end{verbatim}

\textbf{braun\_of :: 'a \Rightarrow \text{nat} \Rightarrow 'a\ tree}

\textbf{braun\_of\ x\ n = fst\ (braun2\_of\ x\ n)}

The running time is clearly logarithmic.

The correctness properties (see Appendix A for \textit{replicate})
list (braun_of x n) = replicate n x
braun (braun_of x n)

are corollaries of the more general statements

braun2_of x n = (s, t) →
list s = replicate n x ∧ list t = replicate (n + 1) x
braun2_of x n = (s, t) →
|s| = n ∧ |t| = n + 1 ∧ braun s ∧ 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 $O(n \lg n)$ by developing a linear-time function. Given a list of elements [1, 2, ...,], we can subdivide it into sublists [1], [2, 3], [4, ..., 7], ... such that the $k$th 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:

\[
\begin{array}{cccc}
4 & 6 & 5 & 7 \\
8 & 12 & 10 & 14 & 9 & 13 & 11 & 15 \\
\end{array}
\]

If we rearrange them in increasing order of the root labels

\[
\begin{array}{cccc}
4 & 5 & 6 & 7 \\
8 & 12 & 9 & 13 & 10 & 14 & 11 & 15 \\
\end{array}
\]

the following pattern emerges: the left sublists are labeled [8, ..., 11], the right sublists [12, ..., 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 $[(t_4, 2, t_6), (t_5, 3, t_7)]$, 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.
11 Arrays via Braun Trees

**brauns :: nat ⇒ 'a list ⇒ 'a tree list**

\[
\text{brauns } k \ xs = \begin{cases} \text{if } xs = [] \text{ then } [] & \\
\text{else let } ys = \text{take } 2^k \ xs; & \\
zs = \text{drop } 2^k \ xs; & \\
ts = \text{brauns } (k + 1) \ zs & \\
in \text{nodes } ts \ ys (\text{drop } 2^k \ ts) & 
\end{cases}
\]

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**

\[
\text{nodes } (l \# \ _\_ \ xs) (x \# \ xs) (r \# \ rs) = \langle l, x, r \rangle \# \text{nodes } ls \ xs \ rs\\n\text{nodes } (l \# \ _\_ \ xs) (x \# \ xs) [] = \langle l, x, [] \rangle \# \text{nodes } ls \ xs []\\n\text{nodes } [\_ \# \ _\_ \ xs] (r \# \ rs) = \langle [], x, r \rangle \# \text{nodes } [\_ \# \ _\_ \ xs] rs\\n\text{nodes } [\_ \# \ _\_ \ xs] [] = \langle [], x, [] \rangle \# \text{nodes } [\_ \# \ _\_ \ xs] []\\n\text{nodes } _\_ \ [\_ \# \ _\_ ] = []
\]

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**

\[
\text{brauns1 } xs = \begin{cases} \text{if } xs = [] \text{ then } () & \\
\text{else brauns } 0 \ xs ! 0 & 
\end{cases}
\]

**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
The result of \( \text{take\_nths} \ i \ k \ xs \) is every \( 2^k \)-th element in \( \text{drop} \ i \ xs \).

A number of simple properties follow by easy inductions:

\[
\begin{align*}
\text{take\_nths} \ i \ k \ (\text{drop} \ j \ xs) &= \text{take\_nths} \ (i + j) \ k \ xs \\
\text{take\_nths} \ 0 \ 0 \ xs &= xs \\
\text{splice} \ (\text{take\_nths} \ 0 \ 1 \ xs) \ (\text{take\_nths} \ 1 \ 1 \ xs) &= xs \\
\text{take\_nths} \ i \ m \ (\text{take\_nths} \ j \ n \ xs) &= \text{take\_nths} \ (i \cdot 2^n + j) \ (m + n) \ xs \\
\text{take\_nths} \ i \ k \ xs &= [] \iff |xs| \leq i \\
i < |xs| &\rightarrow \text{hd} \ (\text{take\_nths} \ i \ k \ xs) = xs \text{ ! } i \\
|xs| = |ys| \lor |xs| = |ys| + 1 &\rightarrow \\
\text{take\_nths} \ 0 \ 1 \ (\text{splice} \ xs \ ys) &= xs \land \\
\text{take\_nths} \ 1 \ 1 \ (\text{splice} \ xs \ ys) &= ys \\
|\text{take\_nths} \ 0 \ 1 \ xs| &= |\text{take\_nths} \ 1 \ 1 \ xs| \lor \\
|\text{take\_nths} \ 0 \ 1 \ xs| &= |\text{take\_nths} \ 1 \ 1 \ xs| + 1
\end{align*}
\]

We also introduce a predicate relating a tree to a list:

\[
\begin{align*}
\text{braun\_list} :: \text{'}a \text{ tree} \Rightarrow \text{'}a \text{ list} \Rightarrow \text{bool} \\
\text{braun\_list} \ () \ xs &= (xs = []) \\
\text{braun\_list} \ (l, x, r) \ xs &= (xs \neq [] \land x = \text{hd} \ xs \land \\
&\text{braun\_list} \ l \ (\text{take\_nths} \ 1 \ 1 \ xs) \land \\
&\text{braun\_list} \ r \ (\text{take\_nths} \ 2 \ 1 \ xs))
\end{align*}
\]

This definition may look a bit mysterious at first but it satisfies a simple specification: \( \text{braun\_list} \ t \ xs \iff \text{braun} \ t \land xs = \text{list} \ t \) (see below). The idea of the above definition is that instead of relating \( \langle l, x, r \rangle \) to \( xs \) via \( \text{splice} \) we invert the process and relate \( l \) and \( r \) to the even and odd numbered elements of \( \text{drop} \ 1 \ xs \).

**Lemma 11.4.** \( \text{braun\_list} \ t \ xs \iff \text{braun} \ t \land xs = \text{list} \ t \)
Theorem 11.5. \( i < \min |xs| 2^k \rightarrow \)

\[ \text{braun_list (brauns k xs ! i) (take_nths i k xs)} \]

Proof. The proof is by induction on the length of \( xs \). Assume \( i < \min |xs| 2^k \), which implies \( xs \neq [] \). Let \( zs = \text{drop} 2^k xs \). Thus \( |zs| < |xs| \) and therefore the IH applies to \( zs \) and yields the property

\[ \forall i j. j = i + 2^k \land i < \min |zs| 2^k + 1 \rightarrow \text{braun_list (ts ! i) (take_nths j (k + 1) xs)} \]

(*)

where \( ts = \text{brauns} (k + 1) zs \). Let \( ts' = \text{drop} 2^k ts \). Below we examine \( \text{nodes ts _ ts' ! i} \) with the help of (11.16). Thus there are four similar cases of which we only discuss one representative one: assume \( i < |ts| \) and \( i \geq |ts'| \).

\[ \text{braun_list (brauns k xs ! i) (take_nths i k xs)} \leftrightarrow \text{braun_list (nodes ts (take 2^k xs) ts'! i) (take_nths i k xs)} \]

\[ \leftrightarrow \text{braun_list (ts ! i) (take_nths (2^k + i) (k + 1) zs) \land braun_list () (take_nths (2^k + 1 + i) (k + 1) xs)} \]

by (11.16), (11.10), (11.11), (11.12) and assumptions

\[ \leftrightarrow \text{True} \]

by (*), (11.11), (11.17) and assumptions

\[ \Box \]

Setting \( i = k = 0 \) in this theorem we obtain the correctness of \( \text{braun1} \) using Lemma 11.4 and (11.8):

Corollary 11.6. \( \text{braun (braun1 xs) \land list (braun1 xs) = xs} \)
Running Time Analysis

We focus on function \textit{brauns}. In the step from \textit{brauns} to \textit{T\textsubscript{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 \textit{take} \textit{n} \textit{xs} and \textit{drop} \textit{n} \textit{zs} is linear in \textit{min} \textit{n} \textit{|xs|} and we simply use \textit{min} \textit{n} \textit{|xs|}. Evaluating \textit{nodes} \textit{ls} \textit{xs} \textit{rs} takes time linear in \textit{|zs|} and \textit{|take} \textit{n} \textit{xs|} = \textit{min} \textit{n} \textit{|xs|}. As a result we obtain the following definition of \textit{T\textsubscript{brauns}}:

\[
T_{\text{brauns}} :: \text{nat} \Rightarrow \text{'a list} \Rightarrow \text{nat}
\]

\[
T_{\text{brauns}} k xs = (\text{if} \; xs = [] \; \text{then} \; 0 \; \text{else let} \; ys = \text{take} \; 2^k \; xs; \; zs = \text{drop} \; 2^k \; xs; \; ts = \text{brauns} \; (k + 1) \; zs \; \text{in} \; 4 \cdot \text{min} \; 2^k \; |xs| + T_{\text{brauns}} \; (k + 1) \; zs)
\]

It is easy to prove that \textit{T\textsubscript{brauns}} is linear:

\textbf{Lemma 11.7.} \textit{T\textsubscript{brauns}} \textit{k} \textit{xs} = 4 \cdot \textit{|xs|}

\textit{Proof.} The proof is by induction on the length of \textit{xs}. If \textit{xs} = [] the claim is trivial. Now assume \textit{xs} \neq [] and let \textit{zs} = \textit{drop} \; 2^k \; \textit{xs}.

\[
T_{\text{brauns}} k xs = T_{\text{brauns}} \; (k + 1) \; zs + 4 \cdot \text{min} \; 2^k \; |xs| \\
= 4 \cdot |zs| + 4 \cdot \text{min} \; 2^k \; |zs| \\
= 4 \cdot (|zs| - 2^k) + 4 \cdot \text{min} \; 2^k \; |zs| = 4 \cdot |zs| \\
\]

11.5.4 Converting a Braun Tree into a List

We improve on function \textit{list} that has running time \textit{O(n lg n)} by developing a linear-time version. Imagine that we want to invert the computation of \textit{brauns} and thus of \textit{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

\[
\begin{array}{cccccc}
4 & & 5 & & 6 & \text{7} \\
& 8 & & 9 & & 10 & & 14 & & 11 & & 15 \\
\end{array}
\]

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:
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 = []`, the measure is 0 but it still leads to a recursive call (with argument []). This problem can be avoided with the measure function \( \varphi = \text{sum_list} \circ \text{map f} \) where \( f = (\lambda t. 2 \cdot |t| + 1) \). Assume `ts \neq []` and let `us = filter (\lambda t. t \neq \langle \rangle) ts`. We need to show that \( \varphi (\text{map left us} \circ \text{map right us}) < \varphi ts \). 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 (\text{left t}) + f (\text{right t}) = 2 \cdot |l| + 2 \cdot |r| + 2` and thus the measure also decreases by 1. Because `ts \neq []` this proves \( \varphi (\text{map left us} \circ \text{map right us}) < \varphi ts \). 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`: `list_fast t = list t` if `braun t`. A direct proof of `list_fast_rec [t] = 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.** \( |ts| = 2^k \land (\forall i < 2^k. \text{braun_list} (ts ! i) (\text{take_nths i} k x)) \) -> `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
\[
    ts = \text{map}\ (\lambda x. \langle \langle, x, \langle \rangle \rangle)\ xs \otimes \text{replicate}\ n\ \langle \rangle
\]
where $n = |ts| - |xs|$. This can be proved pointwise. Take some $i < 2^k$. If $i < |xs|$ then \(\text{take}_n\ths i k xs = \text{take}\ 1\ \text{(drop}\ i\ xs)\) (which can be proved by induction on $xs$). By definition of \text{braun_list} it follows that $t! i = \langle l, xs! i, r \rangle$ for some $l$ and $r$ such that \text{braun_list} $l[]$ and \text{braun_list} $l[]$ and thus $l = r = \langle \rangle$, i.e. $t! i = \langle \langle, xs! i, \langle \rangle \rangle$. If $i < |xs|$ then $\text{take}_n\ths i k xs = []$ by (11.11) and thus \text{braun_list} $\langle ts! i \rangle[]$ by the second premise and thus $ts! i = \langle \rangle$ by definition of \text{braun_list}. This concludes the proof of $(*)$. The desired \text{list_fast_rec} $ts = xs$ follows easily by definition of \text{list_fast_rec}.

Now assume $-|xs| < 2^k$. Then for all $i < 2^k$
\[
    ts! i \neq \langle \rangle \land \text{value}\ (ts! i) = xs! i \land
\]
\[
    \text{braun_list}\ (\text{left}\ (ts! i))\ (\text{take}_n\ths (i + 2^k)\ (k + 1)\ xs) \land
\]
\[
    \text{braun_list}\ (\text{right}\ (ts! i))\ (\text{take}_n\ths (i + 2 \cdot 2^k)\ (k + 1)\ xs)
\]
follows from the second premise with the help of (11.10), (11.11) and (11.12).

We obtain two consequences:
\[
    \text{map}\ \text{value}\ ts = \text{take}\ 2^k\ xs
\]
\[
    \text{list_fast_rec}\ (\text{map}\ \text{left}\ ts @\ \text{map}\ \text{right}\ ts) = \text{drop}\ 2^k\ xs
\]
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 \text{list_fast_rec} $ts = xs$ follows by definition of \text{list_fast_rec}. \[ \square \]

Running Time Analysis

We focus on \text{list_fast_rec}. In the step from \text{list_fast_rec} to $T_{\text{list_fast_rec}}$ 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 $\text{map}\ \text{value}\ ts,\ \text{map}\ \text{left}\ ts,\ \text{map}\ \text{right}\ ts,\ \text{filter}\ (\lambda t. t \neq \langle \rangle)\ ts$ and $ts @ _-$ is linear in $|ts|$ and we simply use $|ts|$. As a result we obtain the following definition of $T_{\text{list_fast_rec}}$:

\[
    T_{\text{list_fast_rec}} :: \ \text{a tree list} \Rightarrow \text{nat}
\]
\[
    T_{\text{list_fast_rec}}\ ts
    = (\text{let}\ us = \text{filter}\ (\lambda t. t \neq \langle \rangle)\ ts
    \text{ in } |ts| +
    \begin{cases}
    0, & \text{if } us = [] \text{ then } 0 \\
    5 \cdot |us| + T_{\text{list_fast_rec}}\ (\text{map}\ \text{left}\ us @\ \text{map}\ \text{right}\ us))
    \end{cases}
    )
\]
The following inductive property is an abstraction of the core of the termination argument of \( \text{list\_fast\_rec} \) above.

\[
(\forall t \in \text{set } ts \cdot t \neq \langle \rangle ) \rightarrow \\
(\sum_{t \in ts} k \cdot |t|) = (\sum_{t \in \text{map left } ts @ \text{ map right } ts} k \cdot |t|) + k \cdot |ts| \tag{11.18}
\]

The suggestive notation \( \sum x \mapsto xs \cdot f x \) abbreviates \( \text{sum\_list} (\text{map } f xs) \).

Now we can state and prove a linear upper bound of \( T_{\text{list\_fast\_rec}} \):

**Theorem 11.9.** \( T_{\text{list\_fast\_rec}} ts \leq (\sum_{t \in ts} 7 \cdot |t| + 1) \)

**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 \( ts = \langle \rangle \) the claim is trivial. Now assume \( ts \neq \langle \rangle \) and let \( us = \text{filter} (\lambda t. t \neq \langle \rangle) ts \) and \( children = \text{map left } us @ \text{ map right } us \).

\[
T_{\text{list\_fast\_rec}} ts = T_{\text{list\_fast\_rec}} children + 5 \cdot |us| + |ts| \leq (\sum_{t \in children} 7 \cdot |t| + 1) + 5 \cdot |us| + |ts| \tag{by IH}
\]

\[
= (\sum_{t \in us} 7 \cdot |t| + |ts|) + 7 \cdot |us| + |ts| \tag{by (11.18)}
\]

\[
\leq (\sum_{t \in ts} 7 \cdot |t| + |ts|) = (\sum_{t \in ts} 7 \cdot |t| + 1)
\]

\( \square \)

### 11.6 Exercises

**Exercise 11.1.** Instead of first showing that Braun trees are almost complete, give a direct proof of \( \text{braun } t \rightarrow h t = |\lg |t|| \) by first showing \( \text{braun } t \rightarrow 2^h t \leq 2 \cdot |t| + 1 \) by induction.

**Exercise 11.2.** Show that function \( \text{bal} \) in Section 4.3.1 produces Braun trees:

\( n \leq |zs| \land \text{bal } n zs = (t, zs) \rightarrow \text{braun } t \) (Isabelle hint: \( \text{bal} \) needs to be qualified as \( \text{Balance. bal} \)).

**Exercise 11.3.** One can view Braun trees as tries (see Chapter 12) by indexing them not with a \( \text{nat} \) but a \( \text{bool list} \) where each bit tells us whether to go left or right (as explained at the start of Section 11.2). Function \( \text{nat\_of} \) specifies the intended correspondence:

\[
\text{nat\_of} :: \text{bool list} \Rightarrow \text{nat}
\]

\[
\text{nat\_of } \langle \rangle = 1
\]

\[
\text{nat\_of } (b \# bs) = 2 \cdot \text{nat\_of } bs + (\text{if } b \text{ then } 1 \text{ else } 0)
\]

Define the counterparts of \( \text{lookup\_trie} \) and \( \text{update\_trie} \):

\[
\text{lookup\_trie} :: 'a \text{ tree } \Rightarrow \text{bool list } \Rightarrow 'a
\]

\[
\text{update\_trie} :: \text{bool list } \Rightarrow 'a \Rightarrow 'a \text{ tree } \Rightarrow 'a \text{ tree}
\]
and prove their correctness:

\[
\begin{align*}
\text{braun } t \land \text{nat}_o f \text{ bs } & \in \{1..|t|\} \rightarrow \\
\text{lookup_trie } t \text{ bs } & = \text{lookup1 } t (\text{nat}_o f \text{ bs}) \\
\text{update_trie } \text{ bs } x t & = \text{update1 } (\text{nat}_o f \text{ bs}) x t
\end{align*}
\]

Exercise 11.4. Function `del_lo` is defined with the help of function `merge`. Define a recursive function `del_lo2 :: 'a tree ⇒ 'a tree` without recourse to any auxiliary function and prove `del_lo2 t = 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: `braun t → 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_{size_fast}` (taking 0 time in the base cases) and prove `T_{size_fast} t \leq (h t)^2`.

Exercise 11.7. Prove correctness of function `braun_of_naive` defined in Section 11.5.2: `list (braun_of_naive x n) = 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].
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\}](image)

### 12.1 Abstract Tries via Functions

A nicely abstract model of tries is the following type:

```plaintext
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 \textit{True}: there are no ordering, balance or other requirements.

Below we use a variant of the function update notation $f(a := b)$:

\[
f(a := b) \equiv f(a := \text{Some } b)
\]

This is how the ADT \textit{Set} is implemented by means of tries:

\[
\begin{align*}
\text{empty} & \:: \ 'a \text{ trie} \\
\text{empty} & \equiv \text{Nd False } (\lambda_\_ \ None) \\
\text{isin} & \:: \ 'a \text{ trie} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \\
\text{isin} \ (\text{Nd } b \ _\_) \ [] & = b \\
\text{isin} \ (\text{Nd } _\_ m) \ (k \ # \ xs) & = (\text{case } m\ k\ of\ \text{None} \Rightarrow \text{False} \mid \text{Some } t \Rightarrow \text{isin } t\ xs) \\
\text{insert} & \:: \ 'a \text{ list} \Rightarrow 'a \text{ trie} \Rightarrow 'a \text{ trie} \\
\text{insert} \ [] \ (\text{Nd } m) & = \text{Nd True } m \\
\text{insert} \ (x \ # \ xs) \ (\text{Nd } b\ m) & = (\text{let } s = \text{case } m\ x\ of\ \text{None} \Rightarrow \text{empty} \mid \text{Some } t \Rightarrow t \\
& \quad \text{in } \text{Nd } b\ (m(x \mapsto \text{insert } xs\ s))) \\
\text{delete} & \:: \ 'a \text{ list} \Rightarrow 'a \text{ trie} \Rightarrow 'a \text{ trie} \\
\text{delete} \ [] \ (\text{Nd } m) & = \text{Nd False } m \\
\text{delete} \ (x \ # \ xs) \ (\text{Nd } b\ m) & = \text{Nd } b\ (\text{case } m\ x\ of\ \text{None} \Rightarrow m \mid \text{Some } t \Rightarrow m(x \mapsto \text{delete } xs\ t))
\end{align*}
\]

The definitions are straightforward. But note that \textit{delete} does not try to shrink the trie. For example:

Formally:

\[
delete\ [a]\ (\text{Nd False } [a \mapsto \text{Nd True } (\lambda_\_ \ None)]) = \text{Nd False } [a \mapsto \text{Nd False } (\lambda_\_ \ None)]
\]

where \( [x \mapsto t] \equiv (\lambda_\_ \ None)(x \mapsto t) \). The resulting trie is correct (it represents the empty set of strings) but could have been shrunk to \text{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 ⇒ 'a list set
set t = {xs | isin t xs}
```

Correctness of `empty` and `isin (set empty = {}) and isin t xs = (xs ∈ set t)` are trivial, correctness of insertion and deletion are easily proved by induction:

```
set (insert xs t) = set t ∪ {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 ⇒ '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 | Nd bool (trie × trie)
```

A binary trie, for example

\[
Nd \ False \ (Nd \ True \ (Nd \ False \ (Lf, \ Lf), \ Nd \ True \ (Lf, \ Lf)), \ Lf)
\]

can be visualized like this:

```
False
   False
   \ True
```

`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:
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.

```
set2 :: bool \rightarrow 'a \times 'a \Rightarrow 'a
set2 b (a1, a2) = (if b then a2 else a1)

mod2 :: ('a \Rightarrow 'a) \Rightarrow bool \Rightarrow 'a \times 'a \Rightarrow 'a 
mod2 f b (a1, a2) = (if b then (a1, f a2) else (f a1, a2))
```

Fig. 12.2. Implementation of Set via binary tries

## Functional Correctness

For the correctness proof we take the same lazy approach as above:

```
set_trie :: trie \Rightarrow bool list set
set_trie t = \{xs | isin t 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

\[
\begin{align*}
\text{set\_trie } (\text{insert } xs \ t) &= \text{set\_trie } t \cup \{xs\} \\
\text{set\_trie } (\text{delete } xs \ t) &= \text{set\_trie } t - \{xs\}
\end{align*}
\]

(12.1) (12.2)

are simple consequences of the following inductive properties:

\[
\begin{align*}
\text{isin } (\text{insert } xs \ t) \ ys &= (xs = ys \lor \text{isin } t \ ys) \\
\text{isin } (\text{delete } xs \ t) \ ys &= (xs \neq ys \land \text{isin } t \ ys)
\end{align*}
\]

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:

\[
\text{datatype } \text{trieP} = \text{LfP} | \text{NdP} (\text{bool list}) \text{ bool } (\text{trieP} \times \text{trieP})
\]

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 ⇒ isnP (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)))
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:

\[
\text{isin} \ (\text{prefix\_trie} \ ps \ t) \ ks = (ps = \text{take} |ps| \ ks \land \text{isin} \ tin \ (\text{drop} \ |ps| \ ks))
\]
\[
isinP \ t \ ks = \text{isin} \ (\text{abs\_trieP} \ t) \ ks
\] (12.3)

Correctness of insertP:

\[
\text{prefix\_trie} \ ks \ (Nd \ True \ (Lf, Lf)) = \text{insert} \ ks \ Lf
\]
\[
\text{insert} \ ps \ (\text{prefix\_trie} \ ps \ (Nd \ b \ lr)) = \text{prefix\_trie} \ ps \ (Nd \ True \ lr)
\]
\[
\text{insert} \ (ks @ ks') \ (\text{prefix\_trie} \ ks \ t) = \text{prefix\_trie} \ ks \ (\text{insert} \ ks' \ t)
\]
\[
\text{prefix\_trie} \ (ps @ qs) \ t = \text{prefix\_trie} \ ps \ (\text{prefix\_trie} \ qs \ t)
\]
\[
\text{split} \ ks \ ps = (qs, ks', ps') \rightarrow
\]
\[
ks = qs @ ks' \land ps = qs @ ps' \land
\]
\[
(ks' \neq [] \land ps' \neq [] \rightarrow \text{hd} \ ks' \neq \text{hd} \ ps')
\]
\[
\text{abs\_trieP} \ (\text{insertP} \ ks \ t) = \text{insert} \ ks \ (\text{abs\_trieP} \ t)
\] (12.4)

Correctness of deleteP:

\[
\text{delete} \ xs \ (\text{prefix\_trie} \ xs \ (Nd \ b \ (l, r))) =
\]
\[
= (\text{if} \ (l, r) = (Lf, Lf) \ \text{then} \ Lf \ \text{else} \ \text{prefix\_trie} \ xs \ (Nd \ False \ (l, r)))
\]
\[
\text{delete} \ (xs @ ys) \ (\text{prefix\_trie} \ xs \ t) =
\]
\[
= (\text{if} \ \text{delete} \ ys \ t = Lf \ \text{then} \ Lf \ \text{else} \ \text{prefix\_trie} \ xs \ (\text{delete} \ ys \ t))
\]
\[
\text{abs\_trieP} \ (\text{deleteP} \ ks \ t) = \text{delete} \ ks \ (\text{abs\_trieP} \ t)
\] (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_trieP o abs_trieP. The required correctness properties (ignoring emptyP and isinP)

\[
\text{set\_trieP} \ (\text{insertP} \ xs \ t) = \text{set\_trieP} \ t \cup \{xs\}
\]
\[
\text{set\_trieP} \ (\text{deleteP} \ xs \ t) = \text{set\_trieP} \ t \setminus \{xs\}
\]

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 \texttt{Nd} by replacing \texttt{Nd} by two constructors representing \texttt{Nd True} and \texttt{Nd False}.

Exercise 12.2. Define the invariant \texttt{invar} that characterizes fully shrunk binary tries, i.e. tries where every non-\texttt{Lf} trie represents a non-empty set. Note that a trie represents the empty set if it does not contain any node \texttt{Nd True \_}. Prove that \texttt{insert} and \texttt{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 of bits.

This chapter presents Huffman’s algorithm and its optimality proof. In a slight departure from the rest of this book, the emphasis is more on graphical intuitions and less on rigorous logical arguments.

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 $\delta_a$ 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

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d}
\end{array}
\end{align*}
\]

and

\[
\begin{align*}
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d}
\end{array}
\end{align*}
\]

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

Huffman’s algorithm is a very simple procedure for constructing an optimum code tree for specified symbol frequencies: First, create a list of leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as node weights. The nodes must be sorted in increasing order of weight. Second, pick the two trees

\[
\begin{align*}
\begin{array}{c}
\text{w}_1 \\
\text{w}_2
\end{array}
\end{align*}
\]

with the lowest weights and insert the tree.
into the list so as to keep it ordered. Finally, repeat the process until only one tree is left in the list.

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. $\begin{array}{cccccc}
    z & 2 \\
    d & 3 \\
    f & 5 \\
    s & 7 \\
    e & 11
\end{array}$

2. $\begin{array}{cccc}
    5 \\
    z & 2 \\
    d & 3
\end{array}$

3. $\begin{array}{cccc}
    s & 7 \\
    10 \\
    f & 5 \\
    z & 2 \\
    d & 3
\end{array}$

4. $\begin{array}{cccc}
    e & 11 \\
    17 \\
    s & 7 \\
    10 \\
    5 \\
    f & 5 \\
    z & 2 \\
    d & 3
\end{array}$
The resulting tree is optimum for the given frequencies.

13.3 The Implementation

The functional implementation of the algorithm relies on the following type:

```haskell
datatype 'a tree = Leaf nat 'a | 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 t₁ t₂`, where `t₁` and `t₂` are the left and right subtrees and `w` caches the sum of the weights of `t₁` and `t₂`. The `cachedWeight` function extracts the weight stored in a node:

```haskell
cachedWeight :: 'a tree ⇒ nat
cachedWeight (Leaf w _) = w
cachedWeight (Node w _ _) = w
```

The implementation builds on two additional auxiliary functions. The first one, `uniteTrees`, combines two trees by adding an inner node above them:

```haskell
uniteTrees :: 'a tree ⇒ 'a tree ⇒ 'a tree
uniteTrees t₁ t₂ = Node (cachedWeight t₁ + cachedWeight t₂) t₁ t₂
```
The second function, \texttt{insortTree}, inserts a tree into a list sorted by cached weight, preserving the sort order:

\begin{verbatim}
insortTree :: 'a tree \Rightarrow 'a tree list \Rightarrow 'a tree list
insortTree u [] = [u]
insortTree u (t \# ts) = (if cachedWeight u \leq cachedWeight t then u \# t \# ts
       else t \# insortTree u ts)
\end{verbatim}

The main function that implements Huffman's algorithm follows:

\begin{verbatim}
huffman :: 'a tree list \Rightarrow 'a tree
huffman [t] = t
huffman (t_1 \# t_2 \# ts) = huffman (insortTree (uniteTrees t_1 t_2) ts)
\end{verbatim}

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 list 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 \textit{alphabet} of a code tree is the set of symbols appearing in the tree's leaf nodes:

\begin{verbatim}
alphabet :: 'a tree \Rightarrow 'a set
alphabet (Leaf _ a) = \{a\}
alphabet (Node _ t_1 t_2) = alphabet t_1 \cup alphabet t_2
\end{verbatim}

A tree is \textit{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 $\delta_a$ (the length of the code word for \textit{a}) to be uniquely defined. This well-formedness property appears as an assumption in many of the lemmas. The definition follows:
consistent :: 'a tree ⇒ bool
consistent (Leaf _) = True
consistent (Node _ t1 t2) = (alphabet t1 ∩ alphabet t2 = {} ∧ consistent t1 ∧ consistent t2)

The depth of a symbol (which we wrote as δa above) is the length of the path from the root to that symbol, or equivalently the length of the code word for the symbol:

depth :: 'a tree ⇒ 'a ⇒ nat
depth (Leaf _) a = 0
depth (Node _ t1 t2) a
= (if a ∈ alphabet t1 then depth t1 a + 1
   else if a ∈ alphabet t2 then depth t2 a + 1 else 0)

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:

height :: 'a tree ⇒ nat
height (Leaf _) = 0
height (Node _ t1 t2) = max (height t1) (height t2) + 1

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:

freq :: 'a tree ⇒ 'a ⇒ nat
freq (Leaf w a) b = (if b = a then w else 0)
freq (Node _ t1 t2) b = freq t1 b + freq t2 b

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:

```plaintext
weight :: 'a tree ⇒ nat
weight (Leaf w _) = w
weight (Node _ t1 t2) = weight t1 + weight t2
```

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

\[ \sum_{a \in \text{alphabet } t} \text{freq } t \ a \cdot \text{depth } t \ a \]

which we wrote as \( \sum_a w_a \delta_a \) above. It is defined recursively by

```plaintext
cost :: 'a tree ⇒ nat
cost (Leaf _ _) = 0
cost (Node _ t1 t2) = weight t1 + cost t1 + weight t2 + cost t2
```

A tree is optimum if and only if its cost is not greater than that of any comparable tree:

```plaintext
optimum :: 'a tree ⇒ bool
optimum t
= (\forall u. \text{consistent } u →
  \text{alphabet } t = \text{alphabet } u →
  \text{freq } t = \text{freq } u → \text{cost } t \leq \text{cost } u)
```

Tree functions are readily generalized to lists of trees, or 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. 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: \textit{swapSyms}, \textit{swapFourSyms}, \textit{mergeSibling}, \textit{splitLeaf} and \textit{minima}.

The interchange function \textit{swapSyms} takes a three \texttt{t} and two symbols \texttt{a}, \texttt{b}, and exchanges them:

\[
\text{swapSyms} :: \texttt{a \Rightarrow t \Rightarrow a} \Rightarrow \texttt{t} \Rightarrow \texttt{a}
\]

\[
\text{swapSyms} \ t \ a \ b = \text{swapLeaves} \ t \ (\text{freq} \ t \ a) \ a \ (\text{freq} \ t \ b) \ b
\]

The following lemma captures the intuition that to minimize the cost, more frequent symbols should be encoded using fewer bits than less frequent ones:

\textbf{Lemma 13.1.} consistent \( t \wedge a \in \text{alphabet} \wedge b \in \text{alphabet} \wedge \text{freq} \ t \ a \leq \text{freq} \ t \ b \wedge \text{depth} \ t \ a \leq \text{depth} \ t \ b \rightarrow \text{cost} \ (\text{swapSyms} \ t \ a \ b) \leq \text{cost} \ t \)

The four-way symbol interchange function \textit{swapFourSyms} takes four symbols \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d} with \( a \neq b \) and \( c \neq d \), and exchanges them so that \texttt{a} and \texttt{b} occupy \texttt{c}'s and \texttt{d}'s positions. A naive definition of this function would be \textit{swapSyms} (\textit{swapSyms} \texttt{t a c}) \texttt{b d}. This naive definition fails in the face of aliasing: If \( a = d \), but \( b \neq c \), then \textit{swapFourSyms} \texttt{a b c d} would leave \texttt{a} in \texttt{b}'s position. Instead, we use this definition:

\[
\text{swapFourSyms} :: \texttt{a \Rightarrow t \Rightarrow a \Rightarrow \texttt{t} \Rightarrow a} \Rightarrow \texttt{t} \Rightarrow \texttt{a}
\]

\[
\text{swapFourSyms} \ t \ a \ b \ c \ d
= \begin{cases} 
\text{swapSyms} \ t \ b \ c & \text{if } a = d \\
\text{swapSyms} \ t \ a \ d & \text{else if } b = c \\
\text{swapSyms} \ (\text{swapSyms} \ t \ a \ c) \ b \ d & \text{else}
\end{cases}
\]

Given a symbol \texttt{a}, the \textit{mergeSibling} function transforms the tree

![Diagram](image)

The frequency of \texttt{a} in the resulting tree is the sum of the original frequencies of \texttt{a} and \texttt{b}. The function is defined by the equations
mergeSibling :: 'a tree ⇒ 'a ⇒ 'a tree
mergeSibling (Leaf wb b) _ = Leaf wb b
mergeSibling (Node w (Leaf wb b) (Leaf wc c)) a
= (if a = b ∨ a = c then Leaf (wb + wc) a
  else Node w (Leaf wb b) (Leaf wc c))
mergeSibling (Node w (Node v va vb) t2) a
= Node w (mergeSibling (Node v va vb) a) (mergeSibling t2 a)
mergeSibling (Node w t1 (Node v va vb)) a
= Node w (mergeSibling t1 a) (mergeSibling (Node v va vb) a)

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 \(\text{sibling } t \ a \neq a\) if and only if \(a\) has a sibling. The definition, which is omitted here, distinguishes the same cases as \(\text{mergeSibling}\).

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.** consistent \(t\) ∧ sibling \(t \ a \neq a\) →

\[
\text{cost (mergeSibling } t \ a\) + \text{freq } t \ a + \text{freq } t \ (\text{sibling } t \ a) = \text{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

\[
\text{into}
\]

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 \(\text{freq } t \ a = w_a + w_b\). The definition follows:

splitLeaf :: 'a tree ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a ⇒ 'a tree
splitLeaf (Leaf wc c) wa a wb b
= (if c = a then Node wc (Leaf wa a) (Leaf wb b) else Leaf wc c)
splitLeaf (Node w t1 t2) wa a wb b

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 \(\text{sibling } t \ a \neq a\) if and only if \(a\) has a sibling. The definition, which is omitted here, distinguishes the same cases as \(\text{mergeSibling}\).

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.** consistent \(t\) ∧ sibling \(t \ a \neq a\) →

\[
\text{cost (mergeSibling } t \ a\) + \text{freq } t \ a + \text{freq } t \ (\text{sibling } t \ a) = \text{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

\[
\text{into}
\]

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 \(\text{freq } t \ a = w_a + w_b\). The definition follows:

splitLeaf :: 'a tree ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a ⇒ 'a tree
splitLeaf (Leaf wc c) wa a wb b
= (if c = a then Node wc (Leaf wa a) (Leaf wb b) else Leaf wc c)
splitLeaf (Node w t1 t2) wa a wb b
= Node w (splitLeaf t₁ wₐ a wₙ b) (splitLeaf t₂ wₐ a wₙ b)

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.** \( \text{consistent } t \land a \in \text{alphabet } t \land \text{freq } t a = w_a + w_b \rightarrow \text{cost } (\text{splitLeaf } t w_a a w_b b) = \text{cost } t + w_a + w_b \)

Finally, the \text{minima} predicate expresses that two symbols \( a, b \) have the lowest frequencies in the tree \( t \) and that \( \text{freq } t a \leq \text{freq } t b \):

\[
\text{minima} :: \text{'}a \text{ tree} \Rightarrow \text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}
\]

\[
\text{minima } t \ a \ b = (a \in \text{alphabet } t \land b \in \text{alphabet } t \land a \neq b \land \\
(\forall c \in \text{alphabet } t. \\
\quad c \neq a \rightarrow c \neq b \rightarrow \text{freq } t a \leq \text{freq } t c \land \text{freq } t b \leq \text{freq } t c))
\]

### 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:

\[
\text{ts} \neq [] \rightarrow \text{alphabet } (\text{huffman } ts) = \text{alphabet} \ F \ \text{ts} \\
\text{consistent} \ F \ \text{ts} \land \text{ts} \neq [] \rightarrow \text{consistent } (\text{huffman } ts) \\
\text{ts} \neq [] \rightarrow \text{freq } (\text{huffman } ts) a = \text{freq} \ F \ \text{ts} a
\]

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

\[
\text{cost } c \ d \ a \ b \leq \text{cost } a \ b \ c \ d
\]
Lemma 13.4. \textit{consistent} \( t \land \text{minima} \ a \ b \land \)
\[ c \in \text{alphabet} \ t \land d \in \text{alphabet} \ t \land \]
\[ \text{depth} \ t \ c = \text{height} \ t \land \text{depth} \ t \ d = \text{height} \ t \land c \neq d \quad \rightarrow \quad \]
\[ \text{cost} \ (\text{swapFourSyms} \ t \ a \ b \ c \ d) \leq \text{cost} \ t \]

\textit{Proof.} The proof is by case analysis on \( a = c \), \( a = d \), \( b = c \), and \( b = d \). The cases are easy to prove by expanding the definition of \text{swapFourSyms} and applying Lemma 13.1. \qed

The tree \text{splitLeaf} \( t \ a \ w_a \ b \ w_b \) is optimum if \( t \) is optimum, under a few assumptions, notably that \( \text{freq} \ t \ a = w_a + w_b \). Graphically:

![optimum tree diagram]

Lemma 13.5. \textit{consistent} \( t \land \text{optimum} \ t \land \)
\[ a \in \text{alphabet} \ t \land b \notin \text{alphabet} \ t \land \text{freq} \ t \ a = w_a + w_b \land \]
\[ (\forall c \in \text{alphabet} \ t. \ w_a \leq \text{freq} \ t \ c \land w_b \leq \text{freq} \ t \ c) \quad \rightarrow \quad \]
\[ \text{optimum} \ (\text{splitLeaf} \ t \ w_a \ a \ w_b \ b) \]

\textit{Proof.} We assume that \( t \)'s cost is less than or equal to that of any other comparable tree \( v \) and show that \text{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 \( \text{height} \ t > 0 \), 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 \text{swapFourSyms} \( u \ a \ b \ c \ d \) in which the minima \( a \) and \( b \) are siblings:

![swap four symbols diagram]

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 \( v \).
cost (splitLeaf t a w_a b w_b) = cost t + w_a + w_b by Lemma 13.3
\leq cost (mergeSibling (swapFourSyms u a b c d) a) + w_a + w_b
\hspace{2cm} \text{by optimality assumption}
= cost (swapFourSyms u a b c d) \hspace{2cm} \text{by Lemma 13.2}
\leq cost u \hspace{2cm} \text{by Lemma 13.4} \quad \Box

Once it has combined two lowest-weight trees using uniteTrees, Huffman’s algorithm 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. consistent_F ts \wedge ts \neq [] \wedge a \in \text{alphabet}_F ts \wedge
freq_F ts a = w_a + w_b \longrightarrow
splitLeaf (huffman ts) w_a a w_b b
= huffman (splitLeaf_F 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

```
c  \quad \text{weight } w_c
\quad \text{weight } w_a
\quad \text{weight } w_b
\quad \text{weight } w_d
\quad \text{weight } w_z
```
gives the same result as applying the algorithm on the “flat” forest

```
c  \quad \text{weight } w_c
\quad \text{weight } w_a + w_b
\quad \text{weight } w_d
\quad \text{weight } w_z
```
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. consistent_F ts \wedge \text{height}_F ts = 0 \wedge \text{sortedByWeight} ts \wedge
\quad ts \neq [] \longrightarrow
\quad \text{optimum} (huffman ts)

Proof. The proof is by induction on the length of ts. The assumptions ensure that ts is of the form

```
a  \quad \text{weight } w_a
b  \quad \text{weight } w_b
\quad \text{weight } w_c
\quad \text{weight } w_d
\quad \text{weight } w_z
```

with \( w_a \leq w_b \leq w_c \leq w_d \leq \cdots \leq w_z \). 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

\[
\begin{array}{c}
\text{huffman} \\
\quad c \quad w_c \\
\quad a \quad w_a \\
\quad b \quad w_b \\
\end{array} \quad \begin{array}{c}
\quad d \quad w_d \\
\quad \ldots \\
\quad z \quad w_z \\
\end{array}
\]

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

\[
\text{splitLeaf} \left( \text{huffman} \begin{array}{c}
\quad c \quad w_c \\
\quad a \quad w_a + w_b \\
\quad d \quad w_d \\
\quad \ldots \\
\quad z \quad w_z \\
\end{array} \right) w_a \ a \ w_b \ b
\]

To prove that this tree is optimum, it suffices by Lemma 13.5 to show that

\[
\text{huffman} \begin{array}{c}
\quad c \quad w_c \\
\quad a \quad w_a + w_b \\
\quad d \quad w_d \\
\quad \ldots \\
\quad z \quad w_z \\
\end{array}
\]

is optimum, which follows from the induction hypothesis. \( \square \)

In summary, we have established that the \texttt{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.

13.7 Chapter Notes

The sorted list of trees constitutes a simple priority queue (Part III). The time complexity of Huffman’s algorithm is quadratic in the size \( n \) of this queue. By using a binary search to implement \texttt{insortTree}, we can obtain an \( O(n \lg n) \) imperative implementation. An \( O(n) \) implementation is possible by maintaining two queues, one containing the unprocessed leaf nodes and the other containing the combined trees [43].

Huffman’s algorithm was invented by David Huffman [32]. The proof above was inspired by Knuth’s informal argument [43]. This chapter’s text is based on a published article [8]. An alternative formal proof, developed using Coq, is due to Théry [73].

Knuth [42] presented an alternative, more abstract view of Huffman’s algorithm as a “Huffman algebra.” Could his approach help simplify our proof? The most tedious steps above concerned splitting nodes, merging siblings, and
swapping symbols. These steps would still be necessary, as the algebraic approach seems restricted to abstracting over the arithmetic reasoning, which is not very difficult in the first place. On the other hand, with Knuth’s approach, perhaps the proof would gain in elegance.
Part III

Priority Queues
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 Min_mset m \equiv \text{Min} (\text{set}_m) and Min yields the minimal element of a finite set of linearly ordered elements.

**Fig. 14.1. ADT Priority\_Queue**

**ADT Priority\_Queue** =

**interface**
- empty :: 'a => 'q
- insert :: 'a \Rightarrow 'q \Rightarrow 'q
- del_min :: 'q \Rightarrow 'q
- get_min :: 'q \Rightarrow 'a

**abstraction** mset :: 'q \Rightarrow 'a multiset

**invariant** invar :: 'q \Rightarrow bool

**specification**
- invar empty (empty-inv)
- mset empty = {} (empty)
- invar q \rightarrow invar (insert x q) (insert-inv)
- invar q \rightarrow mset (insert x q) = mset q + x (insert)
- invar q \land mset q \neq {} \rightarrow invar (del_min q) (del_min-inv)
- invar q \land mset q \neq {} \rightarrow mset (del_min q) = mset q - get_min q (del_min)
- invar q \land mset q \neq {} \rightarrow get_min q = \text{Min}_\text{mset} (mset q) (get_min)

Mergeable priority queues (see Figure 14.2) provide an additional function merge (sometimes: meld or union) with the obvious functionality.
14 Priority Queues

**ADT** `Priority_Queue_Merge = Priority_Queue +`  

interface  
`merge : q ⇒ q ⇒ q`

**specification**  
`invar q1 ∧ invar q2 → invar (merge q1 q2)`  
`invar q1 ∧ invar q2 → mset (merge q1 q2) = mset q1 + mset q2`

*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 [46] 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:

```
heap :: 'a tree ⇒ bool
heap () = True
heap (l, m, r) = ((∀ x∈set_tree l ∪ set_tree r. m ≤ x) ∧ heap l ∧ heap r)
```

Function `mset_tree` extracts the multiset of elements from a tree:

```
mset_tree :: 'a tree ⇒ 'a multiset
mset_tree () = {}  
mset_tree (l, a, r) = {a} + mset_tree l + mset_tree r
```
When verifying a heap-based implementation of priority queues the invariant \textit{invar} and the abstraction function \textit{mset} in the ADT \textit{Priority_Queue} are instantiated by \textit{heap} and \textit{mset\_tree}. The correctness proofs need to talk about both multisets and (because of the \textit{heap} invariant) sets of elements in a heap. We will only show the relevant multiset properties because the set properties follow easily via \textit{set\_mset} (\textit{mset\_tree} \textit{t}) = \textit{set\_tree} \textit{t}.

Both \textit{empty} and \textit{get\_min} have obvious implementations:

\begin{verbatim}
empty = ()
get_min = value
\end{verbatim}

where \textit{value} \textit{(\_, a, \_ \_ \_)} = \textit{a}. If a heap-based implementation provides a \textit{merge} function (e.g. skew heaps in Chapter 22), then \textit{insert} and \textit{del\_min} can be defined like this:

\begin{verbatim}
insert x t = merge (\{\}, x, (\}) t

del_min (\{\}) = ()

del_min (l, \_, r) = merge l r
\end{verbatim}

Note that the following tempting definition of \textit{merge} is functionally correct but leads to very unbalanced heaps:

\begin{verbatim}
merge (\{\}) t = t
merge t (\{\}) = t
merge (\{l_1, a_1, r_1\} =: t_1) (\{l_2, a_2, r_2\} =: t_2) =
(if a_1 \leq a_2 then \{l_1, a_1, merge r_1 t_2\} else \{l_2, a_2, merge t_1 r_2\})
\end{verbatim}

Many of the more advanced implementations of heaps focus on improving this \textit{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 \textit{merge} (w.r.t. \textit{Priority_Queue\_Merge}) and prove functional correctness of the implementations of \textit{insert} and \textit{del\_min} (w.r.t. \textit{Priority\_Queue}).

Bibliographic Remarks

The idea of the heap goes back to Williams [77] who also coined the name.
**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:

\[
\text{type_synonym } 'a \text{lheap} = ('a \times \text{nat}) \text{tree}
\]

\[
mht :: 'a \text{lheap} \Rightarrow \text{nat}
mht \langle \rangle = 0
mht \langle _, (_, n), _ \rangle = n
\]

There are two invariants: the standard *heap* invariant (on augmented trees)

\[
\text{heap} :: ('a \times 'b) \text{tree} \Rightarrow \text{bool}
\text{heap} \langle \rangle = \text{True}
\text{heap} \langle l, (m, _), r \rangle
= ((\forall x \in \text{set_tree} \ l \cup \text{set_tree} \ r. \ m \leq x) \land \text{heap} \ l \land \text{heap} \ r)
\]

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):

\[
\text{ltree} :: 'a \text{lheap} \Rightarrow \text{bool}
\text{ltree} \langle \rangle = \text{True}
\text{ltree} \langle l, (_, n), r \rangle
= (m h r \leq m h l \land n = m h r + 1 \land \text{ltree} \ l \land \text{ltree} \ r)
\]
Thus a tree is a leftist tree if for every subtree the right spine is a shortest path from the root to a leaf. Pictorially:

Now remember $2^{mh} t \leq |t|_1$, i.e. $mh t \leq \lg |t|_1$. Because the expensive operations on leftist heaps descend along the right spine, this means that their running time is logarithmic in the size of the heap.

Exercise 15.1. An alternative definition of leftist tree is via the length of the right spine of the tree:

$rank :: 'a tree => nat$

$rank () = 0$

$rank (l, r) = rank r + 1$

Prove that $l t e e r t \mapsto rank t = mh t$.

15.1 Implementation of ADT \texttt{Priority\_Queue\_Merge}

The key operation is \texttt{merge}:

\begin{verbatim}
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 mhl = mht l; mhr = mht r
        in if mhr <= mhl then (l, (a, mhr + 1), r)
            else (r, (a, mhl + 1), l))
\end{verbatim}

Termination of \texttt{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 ⇒ 'a lheap
del_min ⟨⟩ = ⟨⟩
del_min ⟨l, _, r⟩ = merge l r
```

15.2 Correctness

The above implementation is proved correct w.r.t. the ADT `Priority_Queue_Merge` where

```
mset_tree :: ('a × 'b) tree ⇒ 'a multiset
mset_tree ⟨⟩ = {⟨⟩}
mset_tree ⟨l, (a, _), r⟩ = {⟨a⟩} + mset_tree l + mset_tree r

invar t = (heap t ∧ ltree t)
```

Correctness of `get_min` follows directly from the heap invariant:

\[ \text{heap } t \land t \neq ⟨⟩ \rightarrow \text{get_min } t = \text{Min} (\text{set_tree } t) \]

From the following inductive lemmas about `merge`

\[ \text{mset_tree } (\text{merge } t_1 \ t_2) = \text{mset_tree } t_1 + \text{mset_tree } t_2 \]

\[ \text{ltree } l \land \text{ltree } r \rightarrow \text{ltree } (\text{merge } l \ r) \]

\[ \text{heap } l \land \text{heap } r \rightarrow \text{heap } (\text{merge } l \ r) \]

correctness of `insert` and `del_min` follow easily:
\[ mset\_tree (\text{insert} \ x \ t) = mset\_tree \ t + \{x\} \]
\[ mset\_tree (\text{del}\_\text{min} \ t) = mset\_tree \ t - \{\text{get}\_\text{min} \ t\} \]
\[ \text{ltree} \ t \rightarrow \text{ltree} (\text{insert} \ x \ t) \]
\[ \text{heap} \ t \rightarrow \text{heap} (\text{insert} \ x \ t) \]
\[ \text{ltree} \ t \rightarrow \text{ltree} (\text{del}\_\text{min} \ t) \]
\[ \text{heap} \ t \rightarrow \text{heap} (\text{del}\_\text{min} \ t) \]

Of course the above proof (ignoring the \text{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

\[ \text{ltree} \ l \land \text{ltree} \ r \rightarrow T_{\text{merge}} \ l \ r \leq mh \ l + mh \ r + 1 \]

With \(2^{mh} t \leq |t|_1\) it follows that

\[ \text{ltree} \ l \land \text{ltree} \ r \rightarrow T_{\text{merge}} \ l \ r \leq \lg |l|_1 + \lg |r|_1 + 1 \quad (15.1) \]

which implies logarithmic bounds for insertion and deletion:

\[ \text{ltree} \ t \rightarrow T_{\text{insert}} \ x \ t \leq \lg |t|_1 + 3 \]
\[ \text{ltree} \ t \rightarrow T_{\text{del}\_\text{min}} \ t \leq 2 \cdot \lg |t|_1 + 1 \]

The derivation of the bound for insertion is trivial but the one for deletion uses a little lemma, assuming \text{ltree} \ t:

\[ T_{\text{del}\_\text{min}} \ t = T_{\text{merge}} \ l \ r + 1 \]
\[ \leq \lg |l|_1 + \lg |r|_1 + 2 \quad \text{using (15.1)} \]
\[ \leq 2 \cdot \lg |t|_1 + 1 \quad \text{because} \ \lg x + \lg y + 1 < 2 \cdot \lg (x + y) \]
\[ \text{if } 0 < x \text{ and } 0 < y \]

### 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].
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 \((l, x, r)\) in a Braun tree satisfies \(|l| \in \{|r|, |r| + 1\}\) (\(*\)).

### 16.1 Implementation of ADT *PriorityQueue*

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:

\[
\begin{align*}
\text{insert} &:: 'a \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree} \\
\text{insert } a \\ &= ((), a, ()) \\
\text{insert } a \ (l, x, r) \\ &= (\text{if } a < x \ \text{then } (\text{insert } x r, a, l) \ \text{else } (\text{insert } a r, x, l))
\end{align*}
\]

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:
\( \text{del}_\text{min} :: 'a \text{ tree} \Rightarrow 'a \text{ tree} \)
\( \text{del}_\text{min} () = () \)
\( \text{del}_\text{min} (l, x, r) = (\text{let } (y, l') = \text{del}_\text{left} l \text{ in } \text{sift}_\text{down} r y l') \)

\( \text{del}_\text{left} :: 'a \times 'a \text{ tree} \)
\( \text{del}_\text{left} () , x , r = (x, r) \)
\( \text{del}_\text{left} (l, x, r) = (\text{let } (y, l') = \text{del}_\text{left} l \text{ in } (y, (r, x, l'))) \)

\( \text{sift}_\text{down} :: 'a \text{ tree} \Rightarrow 'a \Rightarrow 'a \text{ tree} \Rightarrow 'a \text{ tree} \)
\( \text{sift}_\text{down} () a _ = ((), a, ()) \)
\( \text{sift}_\text{down} (l, x, _ ) a () = (\text{if } a \leq x \text{ then } ((l, x), a, ()) \text{ else } ((l, a), x, ())) \)
\( \text{sift}_\text{down} (l_1, x_1, r_1) =: t_1 \text{ a } (l_2, x_2, r_2) =: t_2 \)
\( = (\text{if } a \leq x_1 \wedge a \leq x_2 \text{ then } (t_1, a, t_2) \text{ else } (x_1 \leq x_2 \text{ then } (\text{sift}_\text{down} l_1 a r_1, x_1, t_2) \text{ else } (t_1, x_2, \text{sift}_\text{down} l_2 a r_2)) \)

In the first two equations for \( \text{sift}_\text{down} \), 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 \( \text{insert} \) and \( \text{del}_\text{min} \) by presenting the key lemmas. Correctness of \( \text{insert} \) is straightforward:

\[ |\text{insert} \ x \ t| = |t| + 1 \]
\[ \text{mset}_\text{tree} (\text{insert} \ x \ t) = \{x\} + \text{mset}_\text{tree} t \]
\[ \text{braun} t \longrightarrow \text{braun} (\text{insert} \ x \ t) \]
\[ \text{heap} t \longrightarrow \text{heap} (\text{insert} \ x \ t) \]

Correctness of \( \text{del}_\text{min} \) builds on analogous correctness lemmas for the auxiliary functions:

\[ \text{del}_\text{left} t = (x, t') \wedge t \neq () \longrightarrow \text{mset}_\text{tree} t = \{x\} + \text{mset}_\text{tree} t' \]
\[ \text{del}_\text{left} t = (x, t') \wedge t \neq () \wedge \text{heap} t \longrightarrow \text{heap} t' \]
\[ \text{del}_\text{left} t = (x, t') \wedge t \neq () \longrightarrow |t| = |t'| + 1 \]
\[ \text{del}_\text{left} t = (x, t') \wedge t \neq () \wedge \text{braun} t \longrightarrow \text{braun} t' \]
\[ \text{braun}\ (l, a, r) \rightarrow |\text{sift\_down}\ l\ a\ r| = |l| + |r| + 1 \]
\[ \text{braun}\ (l, a, r) \rightarrow \text{braun}\ (\text{sift\_down}\ l\ a\ r) \]
\[ \text{braun}\ (l, a, r) \rightarrow \text{mset\_tree}\ (\text{sift\_down}\ l\ a\ r) = \{a\} + (\text{mset\_tree}\ l + \text{mset\_tree}\ r) \]
\[ \text{braun}\ (l, a, r) \land \text{heap}\ l \land \text{heap}\ r \rightarrow \text{heap}\ (\text{sift\_down}\ l\ a\ r) \]
\[ \text{braun}\ t \rightarrow \text{braun}\ (\text{del\_min}\ t) \]
\[ \text{heap}\ t \land \text{braun}\ t \rightarrow \text{heap}\ (\text{del\_min}\ t) \]
\[ \text{braun}\ t \rightarrow |\text{del\_min}\ t| = |t| - 1 \]
\[ \text{braun}\ t \land t \neq \langle \rangle \rightarrow \text{mset\_tree}\ (\text{del\_min}\ t) = \text{mset\_tree}\ t - \{\text{get\_min}\ t\} \]

**Bibliographic Remarks**

Our implementation of priority queues via Braun trees is due to Paulson [64] who credits it to Okasaki.
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 \(r - 1, \ldots, 0\), 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 \(r - 1, \ldots, 0\).](image)

To define binomial trees in Isabelle, we first define a more general datatype:

```isabelle
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:
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:

\[
\langle r, x, ts \rangle \equiv \text{Node } r x ts
\]

Additionally, we require the heap property, i.e., that the root element of each subtree is a minimal element in that subtree:

\[
\text{invar} \quad :: \quad \forall\ t \in \text{set } ts. \ \text{invar}_{-\text{btree}} t \land \text{map rank } ts = \text{rev } [0..r]
\]

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:

\[
\text{invar}_{-\text{tree}} \quad :: \quad \forall\ t \in \text{set } ts. \ \text{invar}_{-\text{btree}} t \land \text{invar}_{-\text{otree}} t
\]

A binomial heap is a list of binomial trees

\[
\text{type} \quad \equiv \quad \forall\ t \in \text{set } ts. \ \text{invar}_{-\text{btree}} t \land \text{sorted}_{-\text{wrt}} (<) \ (\text{map rank } ts))
\]

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:

\[ mset\_tree :: \text{a tree} \Rightarrow \text{a multiset} \]
\[ mset\_tree (\_, a, ts) = \{ a \} + \sum_{i} (image\_mset mset\_tree (mset ts)) \]
\[ mset\_heap :: \text{a heap} \Rightarrow \text{a multiset} \]
\[ mset\_heap ts = \sum_{i} (image\_mset mset\_tree (mset ts)) \]

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:

\[ \text{invar} \_\text{btree} \ t \rightarrow |mset\_tree \ t| = 2^{\text{rank} \ t} \]

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,...,\( r \). 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:

\[ \text{invar} \ ts \rightarrow |ts| \leq \lg (|mset\_heap \ ts| + 1) \]

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,...,\( |ts| - 1 \). This means that they must have at least \( 2^0, 2^1, \ldots, 2^{\lfloor \log |ts| \rfloor} - 1 \) elements, i.e., at least \( 2^{\lfloor \log |ts| \rfloor} - 1 \) 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:

\[ \text{invar} [] \]
\[ mset\_heap [] = \{ \} \]
\[ (mset\_heap \ ts = \{ \}) = (ts = []) \]
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:

\[
\text{link} :: \text{'a tree} \Rightarrow \text{'a tree} \Rightarrow \text{'a tree}
\]

\[
\text{link} ((r, x_1, ts_1) =: t_1) ((r', x_2, ts_2) =: t_2) =
(\text{if } x_1 \leq x_2 \text{ then } (r + 1, x_1, t_2 \# ts_1) \text{ else } (r + 1, x_2, t_1 \# ts_2))
\]

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 \( \text{link} \) preserves the invariant and that the resulting tree contains the elements of both arguments.

\[
\text{invar_tree } t_1 \land \text{invar_tree } t_2 \land \text{rank } t_1 = \text{rank } t_2 \quad \Rightarrow \\
\text{invar_tree } (\text{link } t_1 t_2)
\]

\[
\text{mset_tree } (\text{link } t_1 t_2) = \text{mset_tree } t_1 + \text{mset_tree } t_2
\]

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:
17.2 Implementation of ADT Priority_Queue

\[ \text{ins\_tree :: 'a tree \Rightarrow 'a heap \Rightarrow 'a heap} \]

\[ \text{ins\_tree t [] = [t]} \]

\[ \text{ins\_tree t1 (t2 \# ts) = (if rank t1 < rank t2 then t1 \# t2 \# ts else ins\_tree (link t1 t2) ts)} \]

Invariant preservation and functional correctness of \text{ins\_tree} is easily proved by induction using the respective properties for \text{link}:

\[ \text{invar\_tree t \land invar ts \land (V t' \in set ts. rank t \leq rank t')} \rightarrow \text{invar (ins\_tree t ts)} \]

\[ \text{mset\_heap (ins\_tree t ts) = mset\_tree t + mset\_heap ts} \]

A single element is inserted as a one-element (rank 0) tree:

\[ \text{insert :: 'a \Rightarrow 'a heap \Rightarrow 'a heap} \]

\[ \text{insert x ts = ins\_tree (0, x, []) ts} \]

The above definition meets the specification for insert required by the Priority_Queue ADT:

\[ \text{invar t \rightarrow invar (insert x t)} \]

\[ \text{mset\_heap (insert x t) = \{x\} + mset\_heap t} \]

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:

\[ \text{merge :: 'a heap \Rightarrow 'a heap \Rightarrow 'a heap} \]

\[ \text{merge ts1 [] = ts1} \]

\[ \text{merge [] (v \# va) = v \# va} \]

\[ \text{merge (t1 \# ts1 =: h1) (t2 \# ts2 =: h2) = (if rank t1 < rank t2 then t1 \# \text{merge ts1 h2 else if rank t2 < rank t1 then t2 \# \text{merge} h1 ts2 else ins\_tree (link t1 t2) (merge ts1 ts2))} \]
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:

\[
\text{invar } ts_1 \land \text{invar } ts_2 \rightarrow \text{invar (merge } ts_1 \text{ ts}_2) \\
\text{mset_heap (merge } ts_1 \text{ ts}_2) = \text{mset_heap } ts_1 + \text{mset_heap } ts_2
\]

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 \text{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:

\[
t' \in \text{set (merge } ts_1 \text{ ts}_2) \land (\forall t_1 \in \text{set } ts_1. \text{ rank } t < \text{rank } t_1) \land (\forall t_2 \in \text{set } ts_2. \text{ rank } t < \text{rank } t_2) \rightarrow \text{rank } t < \text{rank } t'
\]

The proof is by straightforward induction, relying on an analogous bounding lemma for \text{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:

\[
\begin{align*}
\text{get_min} & : \text{ 'a heap } \Rightarrow \text{ 'a} \\
\text{get_min} [t] & = \text{root } t \\
\text{get_min} (t \# v \# va) & = \text{min (root } t) (\text{get_min} (v \# va))
\end{align*}
\]

Correctness of this operation is proved by a simple induction:

\[
\text{mset_heap } ts \neq \{\} \land \text{invar } ts \rightarrow \text{get_min } ts = \text{Min_mset } (\text{mset_heap } ts)
\]

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:

$$
\text{del}_\text{min} :: \text{'a heap} \Rightarrow \text{'a heap}
\text{del}_\text{min} \ ts
= (\text{case get}_\text{min}_\text{rest} \ ts \ of \ (\_, \ _, \ ts_1), \ ts_2) \Rightarrow \text{merge} \ (\text{rev} \ ts_1) \ ts_2)
$$

Here, the auxiliary function $\text{get}_\text{min}_\text{rest}$ splits a heap into a tree with minimal root element, and the remaining trees.

$$
\text{get}_\text{min}_\text{rest} :: \text{'a heap} \Rightarrow \text{'a tree} \times \text{'a heap}
\text{get}_\text{min}_\text{rest} \ [\_] = ([], [])
\text{get}_\text{min}_\text{rest} \ (t \# v \# va) = (let \ (t_0; ts_0) = \text{get}_\text{min}_\text{rest} \ (v \# va)
\text{in if root t} \leq \text{root t'} \ then \ (t, v \# va) \ else \ (t', t \# ts'))
$$

We prove that, for a non-empty heap, $\text{del}_\text{min}$ preserves the invariant and deletes the minimal element:

$$
\begin{align*}
\text{ts} \neq [] & \land \text{invar ts} \rightarrow \text{invar} \ (\text{del}_\text{min} \ ts) \\
\text{ts} \neq [] & \rightarrow \text{mset}_\text{heap} \ ts = \text{mset}_\text{heap} \ (\text{del}_\text{min} \ ts) + \{\text{get}_\text{min} \ ts\}
\end{align*}
$$

The proof is straightforward. For invariant preservation, the key is to show that $\text{get}_\text{min}_\text{rest}$ returns a binomial tree and a binomial heap:

$$
\begin{align*}
\text{get}_\text{min}_\text{rest} \ ts = (t', ts') & \land \text{ts} \neq [] \land \text{invar ts} \rightarrow \text{invar}_\text{tree} \ t' \\
\text{get}_\text{min}_\text{rest} \ ts = (t', ts') & \land \text{ts} \neq [] \land \text{invar ts} \rightarrow \text{invar} \ ts'
\end{align*}
$$

To show that we actually remove a minimal element, we show that $\text{get}_\text{min}_\text{rest}$ selects the same tree as $\text{get}_\text{min}$:

$$
\begin{align*}
\text{ts} \neq [] & \land \text{get}_\text{min}_\text{rest} \ ts = (t', ts') \rightarrow \text{root} \ t' = \text{get}_\text{min} \ ts
\end{align*}
$$

### 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 $\text{insert}$ is straightforward. The running time is dominated by $\text{ins}_\text{tree}$. In the worst case, it iterates over the whole heap, taking constant time per iteration. By straightforward induction, we show
and thus
\[
T_{\text{ins\_tree}} x ts = 2 + |ts| + \lg (|mset\_heap ts|) + 1 + 2.
\]

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 \( |\text{merge ts}_1 ts_2| \leq |ts_1| + |ts_2| \), 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 \( \text{ins\_tree} \) 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 \( \text{ins\_tree} \) and \( \text{merge} \) together with the length of the result:
\[
T_{\text{ins\_tree}} t ts + |\text{ins\_tree} t ts| = 2 + |ts| + |\text{merge ts}_1 ts_2| + T_{\text{merge}} ts_1 ts_2 \leq 2 \cdot (|ts_1| + |ts_2|) + 1
\]

Both estimates can be proved by straightforward induction, and from the second estimate we easily derive a bound for \( \text{merge} \):
\[
invar ts_1 \land invar ts_2 \rightarrow
T_{\text{merge}} ts_1 ts_2 \leq 4 \cdot \lg (|mset\_heap ts_1| + mset\_heap ts_2) + 1 + 1
\]

From the bound for \( \text{merge} \), we can easily derive a bound for \( \text{del\_min} \):
\[
invar ts \land ts \neq [] \rightarrow T_{\text{del\_min}} ts \leq 6 \cdot \lg (|mset\_heap ts| + 1) + 3
\]

The only notable point is that we use a linear time bound for reversing a list, as explained in Section 1.5.1:
\[
T_{\text{rev}} :: 'a list \Rightarrow nat
T_{\text{rev}} zs = |zs| + 1
\]

### 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 \( \text{nod} :: nat \Rightarrow 'a \text{ tree} \Rightarrow nat \) such that \( \text{nod} n t \) 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 \( l \):
\[
invar_{\text{btree}} t \rightarrow \text{nod l t} = \text{rank t choose l}
\]
Hint: You might want to prove separately that

\[
\sum_{i=0}^{r} \binom{i}{n} = \binom{r}{n+1}
\]

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 \(\text{nat list}\).

1. Define a function \(\text{invar} :: \text{nat list} \Rightarrow \text{bool}\) that checks for strictly ascending bit positions, a function \(\text{num_of} :: \text{nat list} \Rightarrow \text{nat}\) that converts a sparse binary number to a natural number, and a function \(\text{add} :: \text{nat list} \Rightarrow \text{nat list} \Rightarrow \text{nat list}\) to add sparse binary numbers.

2. Show that your \(\text{add}\) function preserves the invariant and actually performs addition as far as \(\text{num_of}\) is concerned.

3. Define a running time function for \(\text{add}\) 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 \(\text{add}\) function is implemented similar to the \(\text{merge}\) function, using a \(\text{carry}\) function to insert a bit position into a number (similar to \(\text{ins_tree}\)). 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
You probably have seen this function before:

\[
\begin{align*}
\text{fib} :: \text{nat} & \Rightarrow \text{nat} \\
\text{fib} \ 0 & = 0 \\
\text{fib} \ 1 & = 1 \\
\text{fib} \ (n + 2) & = \text{fib} \ (n + 1) + \text{fib} \ n
\end{align*}
\]

It computes the well-known Fibonacci numbers. You may also have noticed that calculating \( \text{fib} \ 50 \) already causes quite some stress for your computer and there is no hope for \( \text{fib} \ 500 \) to ever return a result.

This is quite unfortunate considering that there is a very simple imperative program to compute these numbers efficiently:

```c
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 an adverse situation here: either we use a clear and elegant definition of \( \text{fib} \) or we get an efficient but convoluted implementation for \( \text{fib} \). Admittedly, we could just prove that both formulations are the
same function, and use whichever one is more suited for the task at hand. For \( \text{fib} \), of course, it is trivial to define a functional analogue of the imperative program and to prove its correctness. However, doing this for all recursive functions we would like to define is tedious. 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”.

In the following, the Fibonacci function will serve as a simple example on which we can illustrate the idea. Next, we will show how to prove the correctness of the efficient implementation in an efficient way. Subsequently, we will discuss further details of the how approach and how it can be applied beyond \( \text{fib} \). The chapter closes with the study of two famous (and archetypical) dynamic programming algorithms: the Bellman-Ford algorithm for finding shortest paths in weighted graphs and an algorithm due to Knuth for computing optimal binary search trees.

### 18.1 Memoization

Let us consider the tree of recursive calls that are issued when computing \( \text{fib} \) 5 (Fig. 18.1). We can see that the subtree for \( \text{fib} \) 3 is computed twice, and that the subtree for \( \text{fib} \) 2 is even computed three times. How can we avoid these repeated 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 simple memoizing version of \( \text{fib} \) that implements the memory as a mapping of type \( \text{nat} \rightarrow \text{nat} \) (the notation \( m(k \mapsto v) \) is a shorthand for \( m(k := \text{Some } v) \), see also Section 12.1):

![Fig. 18.1. Tree of the recursive call structure for \( \text{fib} \) 5](image)
\( \text{fib}_1 :: \text{nat} \Rightarrow (\text{nat} \to \text{nat}) \Rightarrow \text{nat} \times (\text{nat} \to \text{nat}) \)

\[
\begin{align*}
\text{fib}_1 \ 0 \ m &= (0, m(0 \mapsto 0)) \\
\text{fib}_1 \ 1 \ m &= (1, m(1 \mapsto 1)) \\
\text{fib}_1 \ (n + 2) \ m &= (\text{let} \ (i, m) = \text{case} \ m \ n \ \text{of} \ \text{None} \Rightarrow \text{fib}_1 \ n \ m \ | \ \text{Some} \ i \Rightarrow (i, m); \\
& \hspace{1em} (j, m) = \\
& \hspace{2em} \text{case} \ m \ (n + 1) \ \text{of} \ \text{None} \Rightarrow \text{fib}_1 \ (n + 1) \ m \\
& \hspace{3em} | \ \text{Some} \ j \Rightarrow (j, m) \\
& \hspace{1em} \text{in} \ (i + j, m(n + 2 \mapsto i + j))
\end{align*}
\]

And indeed, we can ask Isabelle to compute (via the \textit{value} command) \( \text{fib}_1 \ 50 \) or even \( \text{fib}_1 \ 500 \) 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 \( \text{fib}_1 \), and proving that this function computes the same value as \( \text{fib} \) 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 \( \text{memo} \) (for a type \( k \) of keys and a type \( v \) of values):

\[ \text{memo} ::
\]

\( k \Rightarrow (('k \to 'v) \Rightarrow 'v \times ('k \to 'v)) \Rightarrow ('k \to 'v) \Rightarrow 'v \times ('k \to 'v) \)

\[
\begin{align*}
\text{memo} \ k \ f \ m &= (\text{case} \ m \ k \ \text{of} \ \text{None} \Rightarrow \text{let} \ (v, m) = f \ m \ \text{in} \ (v, m(k \mapsto v)) \\
& \hspace{1em} | \ \text{Some} \ v \Rightarrow (v, m))
\end{align*}
\]

\( \text{fib}_2 :: \text{nat} \Rightarrow (\text{nat} \to \text{nat}) \Rightarrow \text{nat} \times (\text{nat} \to \text{nat}) \)

\[
\begin{align*}
\text{fib}_2 \ 0 &= \text{memo} \ 0 \ (\lambda m. \ (0, m)) \\
\text{fib}_2 \ 1 &= \text{memo} \ 1 \ (\lambda m. \ (1, m)) \\
\text{fib}_2 \ (n + 2) &= \text{memo} \ (n + 2) \\
& \hspace{1em} (\lambda m. \ \text{let} \ (i, m) = \text{fib}_2 \ n \ m; \\
& \hspace{2em} (j, m) = \text{fib}_2 \ (n + 1) \ m \\
& \hspace{3em} \text{in} \ (i + j, m)) \\
\text{fib}_2 \ (n + 2) &= \text{memo} \ (n + 2) \\
& \hspace{1em} (\lambda m. \ \text{let} \ (i, m) = \text{fib}_2 \ n \ m; \ (j, y) = \text{fib}_2 \ (n + 1) \ m \ \text{in} \ (i + j, y))
\end{align*}
\]
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 become rather tedious for more complicated programs and diverges from the original shape of the program, complicating the proofs.

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:

```haskell
datatype ('s, 'a) state = State ('s ⇒ 'a × 's)
```

A value of type ('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:

```haskell
run_state :: ('s, 'a) state ⇒ 's ⇒ 'a × 's
run_state (State f) s = f s
```

The advantage of this definition may not seem immediate. Its value only starts to show 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.

```haskell
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 \(\langle x\rangle\) for `return x`, and \(a \Rightarrow f\) instead of `bind a f`. The “identity” computation \(\langle x\rangle\) simply leaves the given state unchanged and produces \(x\) as a result. The chained computation \(a \Rightarrow f\) starts with some state \(s\), runs \(a\) on it to produce a pair of a result \(x\) and a 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:

\[
\begin{align*}
  \text{get} &:: \ ('s, 's) \text{ state} \\
  \text{get} & = \text{State} (\lambda s. (s, s)) \\
  \text{set} &:: \ 's \Rightarrow ('s, \text{unit}) \text{ state} \\
  \text{set} s' & = \text{State} (\lambda _. ((), s'))
\end{align*}
\]

Let us reformulate `fib` with the help of these concepts:

\[
\begin{align*}
  \text{memo}_1 &:: \ 'k \Rightarrow ('k \rightarrow 'v, 'v) \text{ state} \Rightarrow ('k \rightarrow 'v, 'v) \text{ state} \\
  \text{memo}_1 \ 'k \ a \\
  &= \text{get} \gg= \quad (\lambda m. \text{case} \ m \ 'k \ of \\
  &\quad \text{None} \Rightarrow a \gg= (\lambda v. \text{set} (m \mapsto v)) \gg= (\lambda _. \langle v \rangle)) \\
  &\quad \text{Some} \ x \Rightarrow \langle x \rangle)
\end{align*}
\]

\[
\begin{align*}
  \text{fib}_3 &:: \text{nat} \Rightarrow (\text{nat} \rightarrow \text{nat}, \text{nat}) \text{ state} \\
  \text{fib}_3 \ 0 & = \langle 0 \rangle \\
  \text{fib}_3 \ 1 & = \langle 1 \rangle \\
  \text{fib}_3 \ (n + 2) \\
  &= \text{memo}_1 \ (n + 2) \ (\text{fib}_3 \ n \gg= (\lambda i. \text{fib}_3 \ (n + 1) \gg= (\lambda j. \langle i + j \rangle))))
\end{align*}
\]

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 polish the definition further: 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:

\[
\begin{align*}
  (.) &:: \ ('s, 'a \Rightarrow ('s, 'b) \text{ state}) \text{ state} \Rightarrow ('s, 'a) \text{ state} \Rightarrow ('s, 'b) \text{ state} \\
  f_m \cdot z_m & = (f_m \gg= (\lambda f. \cdot z_m \gg= (\lambda x. f x)))
\end{align*}
\]

We can now spell out our final memoizing version of `fib` where `(.)` replaces ordinary function applications in the original definition:
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 closely follow the syntactic structure of \textit{fib} to prove that \textit{fib}_4 is correct (notwithstanding that \textit{memo}_1 will need a special treatment, of course). 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 so far in this chapter. We noticed that the naive recursive formulation of the Fibonacci numbers leads to a highly inefficient implementation. We then showed how to work around this problem by using memoization to obtain a structurally similar but efficient implementation. After all this, you may wonder why this chapter is titled \textbf{Dynamic Programming} and not \textbf{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. \textit{recursion}. Second, to \textit{memoize} these solutions for smaller problems in, e.g. a table. Thus we could be bold and state:

\begin{center}
\text{dynamic programming} = \text{recursion} + \text{memoization}
\end{center}

A common objection to this equation would be that memoization should be distinguished from \textit{tabulation}. In this view, the former only computes “necessary” solutions for smaller sub-problems, while the latter just “blindly” builds solutions for sub-problems of increasing size, many of which might be unnecessary. The benefit of tabulation could be increased performance, for instance due to improved caching performance. We believe that this distinction is largely irrelevant to our approach. First, in this book we focus on asymptotically efficient solutions, not constant-factor optimizations. Second, in many dynamic programming algorithms memoization would actually compute solutions for the same set of sub-problems as tabulation does. No matter which of the two approaches is used in the implementation, the hard part is
usually to come up with a recursive solution that can efficiently make use of sub-problems in the first place.

There are problems, however, where clever tabulation instead of naive memoization is necessary to achieve an asymptotically optimal solution in terms of memory consumption. One instance of this is the Bellman-Ford algorithm presented in Section 18.4. On this example, we will show that our approach is also akin to tabulation. It can easily be introduced as a final "post-processing" step.

Some readers may have noticed that our optimized implementations of fib are not really optimal as they use a map for memoization. Indeed it is possible to swap in other memory implementations as long as they provide a lookup and an update method. One can even make use of imperative data structures like arrays. As this is not the focus of this book, the interested reader is referred to the literature that is provided at the end of this chapter. Here, we will just assume that the maps used for memoization are implemented as red-black trees (and Isabelle's code generator can be instructed to do so).

For the remainder of this chapter, we will first outline how to prove that fib4 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 the above formula to them.

18.2 Correctness of Memoization

We now want to prove that fib4 is correct. But what is it exactly that we want to prove? We surely want fib4 to produce the same result as fib when run with an empty memory (where empty \( \equiv \lambda x. \text{None} \)):

\[
\text{fst } (\text{run_state } (\text{fib4 } n) \text{ empty}) = \text{fib } n \tag{18.1}
\]

If we were to make a naive attempt 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 attempt generalization as a remedy:

\[
\text{fst } (\text{run_state } (\text{fib4 } n) m) = \text{fib } n
\]

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:
**type_synonym** 'a mem = (nat -> nat, 'a) state

cmem :: (nat -> nat) => bool
cmem m = (\forall n \in dom m. m n = Some (fib n))

dom :: ('k -> 'v) => 'k set
dom m = \{ a \mid m a \neq None\}

Note that, from now, we use the type 'a mem to denote memoized values of type 'a that have been “wrapped up” in our memoizing state monad. Using cmem, we can formulate a general notion of equivalence between a value v and its memoized version a, written v \triangleright a: starting from a consistent memory m, a should produce another consistent memory m', and the result v.

\[
(v) :: 'a \Rightarrow 'a mem \Rightarrow bool
\]
\[
v \triangleright a = (\forall m. cmem m \rightarrow (let (v', m') = run_state a m in v = v' \land cmem m'))
\]

Thus we want to prove

\[
\text{fib } n \triangleright \text{fib}_4 \ n \quad (18.2)
\]

via computation induction on n as above. For the base cases we need to prove statements of the form v \triangleright (v), a property which follows trivially after unfolding the involved definitions. For the induction case, we can unfold \text{fib}_4 (n + 2), and get rid of \text{memo}_1 by applying the following rule (which we instantiate with a = \text{fib}_4 n):

\[
\text{fib } n \triangleright a \rightarrow \text{fib } n \triangleright \text{memo}_1 \ n \ a \quad (18.3)
\]

For the remainder of the proof, we now want to unfold \text{fib} (n + 2) and then follow the syntactic structure of \text{fib}_4 and \text{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 mem) mem. 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 (\(\triangleright\)) = (\(\triangleright_{\text{=}1}\))):

\[
(\triangleright) :: \ a \Rightarrow (\lambda a \Rightarrow (\lambda b \Rightarrow \text{bool}) \Rightarrow \text{bool}) \Rightarrow \text{bool}
\]

\[
v \triangleright_R s
= (\forall m. \text{cmem } m \rightarrow \text{let } (\lambda' m' = \text{run_state } s m \text{ in } R v' \triangleright \text{cmem } m'))
\]

However, we still do not have a means of expressing the second part of our sentence. To this end, we use the function relator (\(\Rightarrow\)):

\[
(\Rightarrow) :: \ a \Rightarrow (\lambda c \Rightarrow \text{bool})
\Rightarrow (\lambda b \Rightarrow (\lambda d \Rightarrow \text{bool}) \Rightarrow (\lambda a \Rightarrow (\lambda b \Rightarrow (\lambda c \Rightarrow (\lambda d \Rightarrow \text{bool}))) \Rightarrow \text{bool})
\]

\[
R \Rightarrow S = (\lambda f. \forall x y. \ R x y \rightarrow S (f x) (g y))
\]

Spelled out, we have \((R \Rightarrow S) f g\) 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:

\[
x \triangleright x_m \land f \triangleright (\text{=}1) \Rightarrow (\triangleright) f_m \rightarrow f x \triangleright f_m . x_m
\]

In our concrete example, we apply it once to the goal

\[
fib (n + 1) + fib n \triangleright \text{} (\lambda a. \text{fib a}) \text{ . (fib a) . (fib a)}
\]

solve the first premise with the induction hypotheses, and arrive at

\[
(+) \ (\text{fib a}) \triangleright (\text{=}1) \Rightarrow (\triangleright) (\lambda a. \text{fib a}) \text{ . (fib a)}
\]

Our current rule for application (18.4) does not match this goal. Thus we need to generalize it. In addition, we need a new rule for return, and a rule for (\(\Rightarrow\)).

To summarize, we need the following set of theorems about our consistency relation, applying them wherever they match syntactically to finish the proof of (18.2):

\[
R x y \rightarrow x \triangleright_R y
\]

\[
x \triangleright_R x_m \land f \triangleright_R \Rightarrow \triangleright_S f_m \rightarrow f x \triangleright f_m . x_m
\]

\[
(\forall x y. \ R x y \rightarrow S (f x) (g y)) \rightarrow (R \Rightarrow S) f g
\]

The theorem we aimed for initially
is now a trivial corollary of \( \text{fib } n \vdash \text{fib}_4 \ n \). 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 \( \text{fib} \).

### 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 \( \text{fib} \). 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 \( a \ pri\text{ori} \) 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 \( \text{memo}_1 \)). We will name this first process monadification. For the second component, many different memory implementations can be used, as long as we can define \( \text{memo}_1 \) and prove its characteristic theorem (18.3). For details on this, the reader is referred to the literature. Here, we want to turn our attention towards monadification.

To discuss some of the intricacies of monadification, let us first stick with \( \text{fib} \) for a bit longer and consider the following alternative definition (which is mathematically equivalent but not the same program):

\[
\text{fib } n = (\text{if } n = 0 \text{ then } 0 \text{ else } 1 + \text{sum_list (map fib [0..<n - 1])})
\]

We have not yet seen how to handle two ingredients of this program: constructs like \( \text{if}-\text{then}-\text{else} \) or case-combinators; and higher-order functions such as \( \text{map} \).

It is quite clear how \( \text{if}-\text{then}-\text{else} \) can be lifted to the state monad:

\[
\text{if}_m :: \text{bool mem } \Rightarrow 'a \text{ mem } \Rightarrow 'a \text{ mem } \Rightarrow 'a \text{ mem } \\
\text{if}_m \ b_m \ x_m \ y_m = b_m \Rightarrow (\lambda b. \text{if } b \text{ then } x_m \text{ else } y_m)
\]

By following the structure of the terms, we can also deduce a proof rule for \( \text{if}_m \):

\[
b \vdash b_m \land x \vdash_R x_m \land y \vdash_R y_m \rightarrow \\
(\text{if } b \text{ then } x \text{ else } y) \vdash_R \text{if}_m \ b_m \ x_m \ y_m
\]

* If you are just interested in the dynamic programming algorithms of the following sections, this section can safely be skipped on first reading.
However, suppose we want to apply this proof rule to our new equation for \( \text{fib} \). We will certainly need the knowledge of whether \( n = 0 \) to make progress in the correctness proof. Thus we make our rule more precise:

\[
\begin{align*}
\text{if } b \text{ then } x \text{ else } y & \triangleright_R m \overset{b_m}{\triangleright} & (\neg b \rightarrow y \triangleright_R y_m)
\end{align*}
\]

How can we lift \( \text{map} \) to the state monad level? Consider its defining equations:

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x \# xs) &= f \; x \# \text{map } f \; xs
\end{align*}
\]

We can follow the pattern we used to monadify \( \text{fib} \) to monadify \( \text{map} \):

\[
\begin{align*}
\text{map}_m' \; f \; [] &= \{[]\} \\
\text{map}_m' \; f \; (x \# xs) &= \{\lambda a. \{\lambda b. \{a \# b\}\}\} \cdot (\{f\} \cdot \{x\}) \cdot (\text{map}_m' \; f \; xs)
\end{align*}
\]

We have obtained a function \( \text{map}_m' \) of type

\[
('a \Rightarrow 'b \text{ mem}) \Rightarrow ('a \text{ list}) \Rightarrow ('b \text{ list mem})
\]

This is not yet compatible with our scheme of lifting function application to \((\cdot)\). We need a function of type

\[
(('a \Rightarrow 'b \text{ mem}) \Rightarrow ('a \text{ list}) \Rightarrow ('b \text{ list mem}) \text{ mem}) \text{ mem}
\]

because \( \text{map} \) has two arguments and we need one layer of the state monad for each of its arguments. Therefore we simply define

\[
\text{map}_m = \{\lambda f. \{\text{map}_m' \; f\}\}
\]

For inductive proofs about the new definition of \( \text{fib} \), we also need the knowledge that \( \text{fib} \) is recursively applied only to smaller values than \( n \) when computing \( \text{fib} \; n \). That is, we need to know which values \( f \) is applied to in \( \text{map } f \; xs \). We can encode this knowledge in a proof rule for \( \text{map} \):

\[
\begin{align*}
xs &= ys \land (\forall x. \; x \in \text{set } ys \rightarrow f \; x \triangleright_R f_m \; x) \rightarrow \\
\text{map } f \; xs &\triangleright_R \text{list}\_\text{all2} \; R \; \text{map}_m \cdot \{f\} \cdot \{ys\}
\end{align*}
\]

The relator \( \text{list}\_\text{all2} \) lifts \( R \) to a pairwise relation on lists:

\[
\text{list}\_\text{all2} \; R \; xs \; ys = (|xs| = |ys| \land (\forall i < |xs|. \; R \; (xs ! i) \; (ys ! i)))
\]
To summarize, here is a fully memoized version of the alternative definition of \(\text{fib}\):

\[
\text{fib}_m :: \text{nat} \Rightarrow \text{nat} \text{ mem} \\
\text{fib}_m = \text{memo}_1 \ n \\
(\text{if} \ m \ (\{n = 0\} \ \{0\}) \\
\ (\\lambda a. \ \{1 + a\}) \\
\ (\\lambda a. \ \{\text{sum_list} \ a\}) \ . \ (\\{\text{map}_m \ . \ \{\text{fib}_m'\} \ . \ \{0..<n - 1\}\})))))
\]

The correctness proof for \(\text{fib}_m\) is analogous to the one for \(\text{fib}_4\), 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 [79]. Finally note that none of the techniques we used so far are specific to \(\text{fib}\). The only parts that have to be adopted are the definitions of \(\text{memo}_1\) and \(\text{cmem}\). In Isabelle, this can be done by simply instantiating a locale.

This concludes the discussion of the fundamentals of our approach towards verified dynamic programming. We now turn to the study of two typical examples of dynamic programming algorithms: the Bellman-Ford algorithm and an algorithm for computing optimal binary search trees.

### 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.

Formally, we will take a simple view of graphs. We assume that we are given a number of nodes numbered 0, ..., n, and some sink \( t \in \{0..n\} \) (thus \( n = t \) = 4 in the example). Edge weights are given by a function \( W : \text{int} \Rightarrow \text{int} \Rightarrow \text{int extended} \). The type \( \text{int extended} \) extends the natural numbers with positive and negative infinity:

\[
\text{datatype } \ 'a \text{ extended } = \text{Fin } 'a \mid \infty \mid -\infty
\]

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 \( j \). The purpose of \(-\infty\) will become clear later.

18.4.1 Deriving a Recursive Solution

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 ≤ 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.

<table>
<thead>
<tr>
<th>i/v</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $OPT \ i \ v$:

$OPT :: nat \Rightarrow nat \Rightarrow int \ extended$

$OPT \ i \ v$

$= Min \ (\{weight \ (v \ # \ xs \ @ \ [t]) \ | \ xs\ |
\leq \ i \wedge \ set \ xs \subseteq \ \{0..n\}\cup
\{if \ t = v \ then \ 0 \ else \ \infty\})$

$weight :: (nat \Rightarrow nat \Rightarrow int \ extended) \Rightarrow nat \ list \Rightarrow int \ extended$

$weight \ _ \ [\_] = 0$

$weight \ W \ (v \ # \ w \ # \ xs) = W \ v \ w + weight \ W \ (w \ # \ xs)$

If $i = 0$, things are simple:

$OPT \ 0 \ v = \ (if \ t = v \ then \ 0 \ else \ \infty)$

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 $OPT \ i \ w$:

$OPT \ (i + 1) \ v$

$= min \ (OPT \ i \ v) \ (Min \ \{W \ v \ w + OPT \ i \ w \ | \ w \leq n\})$

Proof. We prove this equality by proving two inequalities:
(\text{lhs} \leq \text{rhs}) For this direction, we essentially need to show that every path on the rhs is covered by the lhs, which is trivial.

(\text{lhs} \geq \text{rhs}) We skip the cases where \(OPT (i + 1) v\) is trivially 0 or \(\infty\) (i.e. where it is given by the singleton set in the definition of \(OPT\)). Thus consider some \(xs\) such that \(OPT (i + 1) v = \text{weight} (v \# xs @ [t])\), \(|xs| \leq i\), and \(\text{set} xs \subseteq \{0..n\}\). The cases where \(|xs| < i\) or \(i = 0\) are trivial. Otherwise, we have \(OPT (i + 1) v = W v (hd xs) + \text{weight} (xs @ [t])\) by definition of \(\text{weight}\), and \(OPT i (hd xs) \leq \text{weight} (xs @ [t])\) by definition of \(OPT\). Therefore, we can show:

\[OPT (i + 1) v \geq W v (hd xs) + OPT i (hd xs) \geq \text{rhs}\]

We can turn these equations into a recursive program:

\[
bf :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{int} \ \text{extended}
\]

\[
bf 0 v = (\text{if } t = v \text{ then } 0 \text{ else } \infty)
\]

\[
bf (i + 1) v = \text{min_list} (bf i v \# \text{map} (\lambda w. W v w + bf i w) [0..<n + 1])
\]

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 \(v \leq n\) 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 $v \leq n$.

Otherwise, there is a negative reaching cycle $ys = a \ # \ xs \ @ \ [a]$ with weight $ys < 0$. Working towards a contradiction, assume that $OPT \ n \ v \leq OPT \ (n + 1) \ v$ for all $v \leq n$. Using the recursion we proved above, this implies $OPT \ n \ v \leq W \ v \ u + OPT \ n \ u$ for all $u, v \leq n$. By applying this inequality to the nodes in $a \ # \ xs$, we can prove the inequality

$$\sum_{i=0}^{n+1} (OPT \ n \ i) \leq \sum_{i=0}^{n+1} (OPT \ n \ i) + weight \ ys$$

This implies $0 \leq weight \ ys$, 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:

```markdown
bellman_ford ::
((nat × nat, int extended) mapping, int extended list option) state
bellman_ford = iter_bf (n, n) >=
(λ_. mapm' (bf_m n) [0..<n + 1] >=
(λxs. mapm' (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 (for $fib$), 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 $t$):

$$(\forall i \leq n. \ \forall j \leq n. \ -\infty < W \ i \ j) \rightarrow
fst \ (run_state bellman_ford empty) = (if \ contains_negative_reaching_cycle \ then \ None
else \ Some \ (map \ shortest \ [0..<n + 1]))$$

Here, $shortest$ is defined analogously to $OPT$ but for paths of unbounded length.
18.5 Optimal Binary Search Trees

In this book, we have studied various tree data structures that guarantee logarithmic running time 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 list \([i..j]\) of integers ranging from \(i\) to \(j\) and a function \(p : \text{int} \Rightarrow \text{nat}\) that maps each key in the range to a frequency with which this key is searched for. Our goal is to find a binary search tree that minimizes the expected number of comparisons when presented with a sequence of lookup operations for 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

```
               3
             /   \
            2     4
           /     /  \
          1     5
```

incurs an expected value of 2.15 comparison operations. However, the minimal expected value is 2 and is achieved by this tree:

```
               4
             /   \
            2     5
           /     /  \
          1     3
```

Our task is equivalent to minimizing the weighted path length (or cost) as we did for Huffman encodings (Chapter 13). 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:

\[
\text{cost } \langle \rangle = 0 \\
\text{cost } \langle l, k, r \rangle
\]
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 Recursive Solution

The key insight into the problem is that subtrees of optimal binary search trees are also optimal. 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:

\[
\text{wp}l\ W\ i\ j = 0 \\
\text{wp}l\ W\ i\ j\ (l, k, r) = \text{wp}l\ W\ i\ (k - 1)\ l + \text{wp}l\ W\ (k + 1)\ j\ r + \text{wp}l\ W\ i\ j \cdot \sum_{k = i}^{p\ k} W\ i\ j = \text{wp}l\ W\ i\ j
\]

It is easy to see that \text{wp}l\ W\ i\ j\ is just a reformulation of \text{cost} t:

\[
inorder\ t = [i..j] \rightarrow \text{wp}l\ W\ i\ j\ t = \text{cost}\ t
\]

We can actually forget about the original frequencies \(p\) and just optimize \text{wp}l\ W\ i\ j\ for some fixed weight function \(W : \text{int} \Rightarrow \text{int} \Rightarrow \text{nat}\).

The binary search tree \(t\) that contains the keys \([i..j]\) and minimizes \text{wp}l\ 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 minimal again, i.e. minimize \text{wp}l\ W\ i\ (k - 1)\ l\ and \text{wp}l\ W\ (k + 1)\ j\ . This yields the following recursive functions for computing the minimal weighted path length (\text{min}\ _\text{wp}l) and a corresponding binary search tree (\text{opt}\ _\text{bst}):
Here \( \text{argmin} f xs \) returns the rightmost \( x \in \text{set} \, xs \) such that \( f \, x \) is minimal among \( xs \) (i.e. \( f \, x \leq f \, y \) for all \( y \in \text{set} \, xs \)).

To prove that \( \min \, \text{wpl} \) and \( \text{opt_bst} \) are correct, we want to show two properties: \( \min \, \text{wpl} \, i \, j \) should be a lower bound of \( \text{wpl} \, W \, i \, j \, t \) for any search tree \( t \) for \([i..j]\), and \( \min \, \text{wpl} \, i \, j \) should correspond to the weight of an actual search tree, namely \( \text{opt_bst} \, i \, j \). Formally, we prove the following properties:

\[
\begin{align*}
\text{inorder} \, t &= [i..j] \rightarrow \min \, \text{wpl} \, i \, j \leq \text{wpl} \, W \, i \, j \, t \\
\text{inorder} \, (\text{opt_bst} \, i \, j) &= [i..j] \\
\text{wpl} \, W \, i \, j \, (\text{opt_bst} \, i \, j) &= \min \, \text{wpl} \, i \, j
\end{align*}
\]

The first property is easily proved by computation induction on \( \text{wpl} \, W \), and the second and third property are equally easily proved using computation induction on \( \min \, \text{wpl} \).

If \( W \) is constructed from \( p \) as above, we can derive the following correctness theorems referring to the original problem:

\[
\begin{align*}
\text{inorder} \, t &= [i..j] \rightarrow \min \, \text{wpl} \, W \, i \, j \leq \text{cost} \, t \\
\text{cost} \, (\text{opt_bst} \, W \, i \, j) &= \min \, \text{wpl} \, W \, i \, j
\end{align*}
\]

### 18.5.2 Memoization

We can apply the memoization techniques that were discussed above to efficiently compute \( \min \, \text{wpl} \) and \( \text{opt_bst} \). The only remaining caveat is that \( W \) also needs to be computed efficiently from the distribution \( p \). If we just use the defining equality \( W \, i \, j = (\sum_k^{j-i} p \, k) \), the computation of \( W \) is unnecessarily costly. Another way is to memoize \( W \) itself, using the following recursion:

\[
W \, p \, i \, j = (\text{if } i \leq j \text{ then } W \, p \, i \, (j-1) + p \, j \text{ else } 0)
\]

This yields a memoizing version \( W_m \)' and a theorem that connects it to \( W \):
We can now iterate $W_{m'} p i n$ for $i = 0 \ldots n$ to pre-compute all relevant values of $W p i j$:

$$W_c p n = \text{snd} (\text{run_state} (\text{map}_m' (\lambda i. W_{m'} p i n) [0..n]) \text{empty})$$

Using the correctness theorem for $\text{map}_m'$ from above, it can easily be shown that this yields a consistent memory:

$$\text{cmem} (W_c p n)$$

We can show the following equation for computing $W$

$$W p i j = (\text{case} (W_c p n) (i, j) \text{ of } \text{None} \Rightarrow W p i j \mid \text{Some } x \Rightarrow x)$$

Note that the $\text{None}$ branch will only be triggered when indices outside of $0 \ldots n$ are accessed. Finally, we can use $W_c$ to pass the pre-computed values of $W$ to $\text{opt_bst}$:

```
\text{opt_bst}' :: (\text{int} \Rightarrow \text{nat}) \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{tree}
\text{opt_bst}' p i j \equiv
\begin{array}{l}
\text{let } M = W_c p j; \\
W = \lambda i. \text{case } M (i, j) \text{ of } \text{None} \Rightarrow W p i j \mid \text{Some } x \Rightarrow x \\
\text{in } \text{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 $\text{opt_bst}$, 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 $\text{opt_bst}$ and relies on the following observation.

Let $R i j$ denote the maximal root of any optimal binary search for $[i..j]$:

$$R i j = \text{argmin} (\lambda k. w i j + \text{min}_wpl i (k - 1) + \text{min}_wpl (k + 1) j) [i..j]$$

It can be shown that $R i j$ is bounded by $R i (j - 1)$ and $R (i + 1) j$:

$$i < j \rightarrow R i (j - 1) \leq R i j \land R i j \leq R (i + 1) j$$
The proof of this fact is rather involved and the details can be found in the references provided at the end of this section.

With this knowledge, we can make the following optimization to \( \text{opt\_bst} \):

\[
\text{opt\_bst} \colon \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \text{ tree}
\]

\[
\text{opt\_bst} i j = \begin{cases} 
\text{if } j < i & \text{then } \{ \} \\
\text{else if } i = j & \text{then } \{ \}, i, \{ \} \\
\text{else let } \text{left} = \text{root} (\text{opt\_bst} i (j - 1)); \\
\text{right} = \text{root} (\text{opt\_bst} (i + 1) j) \\
\text{in } \text{argmin} (wpl i j) \\
\map (\lambda k. (\text{opt\_bst} i (k - 1), k, \\
\text{opt\_bst} (k + 1) j)) \\
[\text{left..right}])
\end{cases}
\]

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:

\[
d \leq n \implies \left( \sum_{j = d}^{n} j \cdot n \text{ let } i = j - d \text{ in } R \left( i + 1 \right) j - R \left( i \right) (j - 1) + 1 \right) = R \left( n - d + 1 \right) n - R 0 (d - 1) + n - d + 1
\]

This quantity is bounded by \( 2 \cdot n \), which implies that the overall number of recursive 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.
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.
The potential method is a particular technique for amortized analysis. The
key idea is to define a potential function $\Phi$ 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 $\Phi$. Then the amortized running time function $A_f$ for $f$ is
defined as follows:

$$A_f s = T_f s + \Phi (f s) - \Phi s$$

(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

$$\sum_{i=0}^{n-1} A_f s_i = \sum_{i=0}^{n-1} T_f s_i + \Phi s_n - \Phi s_0$$

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 $\Phi$ 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 $\text{invar} \ s$ if the data structure comes
with an invariant $\text{invar}$.

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:

\[
\begin{align*}
\text{incr} &:: \text{bool list} \Rightarrow \text{bool list} \\
\text{incr} \ [] & = [\text{True}] \\
\text{incr} \ (\text{False} \ # \ bs) & = \text{True} \ # \ bs \\
\text{incr} \ (\text{True} \ # \ bs) & = \text{False} \ # \ \text{incr} \ bs \\
T_{\text{incr}} &:: \text{bool list} \Rightarrow \text{real} \\
T_{\text{incr}} \ [] & = 1 \\
T_{\text{incr}} \ (\text{False} \ # \ _) & = 1 \\
T_{\text{incr}} \ (\text{True} \ # \ bs) & = T_{\text{incr}} \ bs + 1
\end{align*}
\]

The potential of a counter is the number of True's because they increase \( T_{\text{incr}} \):

\[
\Phi :: \text{bool list} \Rightarrow \text{real} \\
\Phi \ bs = |\text{filter} \ (\lambda x. \ x) \ bs|
\]

Clearly the potential is never negative.

The amortized complexity of incr is 2:

\[
T_{\text{incr}} \ bs + \Phi \ (\text{incr} \ bs) - \Phi \ bs = 2
\]

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.
20.1 Queue Specification

A queue can be viewed as a glorified list with function \texttt{enq} for adding an element to the end of the list and function \texttt{first} for accessing and \texttt{deq} for removing the first element. This is the full ADT:

\textbf{ADT} Queue =

\textbf{interface} empty :: 'q
\hspace{1em} enq :: 'a \Rightarrow 'q \Rightarrow 'q
\hspace{1em} deq :: 'q \Rightarrow 'q
\hspace{1em} first :: 'q \Rightarrow 'a
\hspace{1em} is_empty :: 'q \Rightarrow bool

\textbf{abstraction} list :: 'q \Rightarrow 'a list

\textbf{invariant} invar :: 'q \Rightarrow bool

\textbf{specification} list empty = []
\hspace{1em} invar q \Rightarrow list (enq x q) = list q @ [x]
\hspace{1em} invar q \Rightarrow list (deq q) = tl (list q)
\hspace{1em} invar q \land list q \neq [] \Rightarrow first q = hd (list q)
\hspace{1em} invar q \Rightarrow is_empty q = (list q = [])
\hspace{1em} invar empty
\hspace{1em} invar q \Rightarrow invar (enq x q)
\hspace{1em} invar q \Rightarrow invar (deq q)

A trivial implementation is as a list, but then \texttt{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:

\[
\begin{align*}
\text{list} & : \text{`a list} \times \text{`a list} \Rightarrow \text{`a list} \\
\text{list} (fs, rs) & = fs @ rev rs
\end{align*}
\]

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 \(enq\) \(xn\) \(...(enq x1 empty)...\) would result in \(([], [x_n, ..., x_1])\) and \(first\) would

\[
\begin{align*}
\text{norm} & : \text{`a list} \times \text{`a list} \Rightarrow \text{`a list} \times \text{`a list} \\
\text{norm} (fs, rs) & = (\text{if fs = [] then } (itrev rs [], []) \text{ else } (fs, rs)) \\
\text{enq} & : \text{`a} \Rightarrow \text{`a list} \times \text{`a list} \Rightarrow \text{`a list} \times \text{`a list} \\
\text{enq} a (fs, rs) & = \text{norm} (fs, a # rs) \\
\text{deq} & : \text{`a list} \times \text{`a list} \Rightarrow \text{`a list} \times \text{`a list} \\
\text{deq} (fs, rs) & = (\text{if fs = [] then } (fs, rs) \text{ else } \text{norm} (tl fs, rs)) \\
\text{first} & : \text{`a list} \times \text{`a list} \Rightarrow \text{`a} \\
\text{first} (a # _, _) & = a \\
\text{is_empty} & : \text{`a list} \times \text{`a list} \Rightarrow \text{bool} \\
\text{is_empty} (fs, _) & = (fs = [])
\end{align*}
\]

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 (\([\text{[]}, rs\) where \(rs \neq \text{[]}\). Thus \(\text{norm}\) guarantees the following invariant:

\[
\begin{align*}
\text{invar} &:: 'a\ \text{list} \times 'a\ \text{list} \Rightarrow \text{bool} \\
\text{invar} (fs, rs) & = (fs = \text{[]} \rightarrow rs = \text{[]})
\end{align*}
\]

Functional correctness, i.e. proofs of the properties in the ADT \(\text{Queue}\), 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

\[
\Phi :: 'a\ \text{list} \times 'a\ \text{list} \Rightarrow \text{nat} \\
\Phi (_, rs) = |rs|
\]

because \(|rs|\) is the amount we have accumulated by charging 1 for each \(\text{enq}\). This is enough to pay for the eventual reversal. Now it is easy to prove that both \(\text{enq}\) and \(\text{deq}\) have amortized constant running time:

\[
\begin{align*}
T_{\text{enq}} a (fs, rs) + \Phi (\text{enq} a (fs, rs)) &- \Phi (fs, rs) \leq 4 \\
T_{\text{deq}} (fs, rs) + \Phi (\text{deq} (fs, rs)) &- \Phi (fs, rs) \leq 3
\end{align*}
\]

The two observer functions \(\text{first}\) and \(\text{is_empty}\) 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:

\[
\begin{align*}
\text{rev\_step} &:: 'a\ \text{list} \times 'a\ \text{list} \Rightarrow 'a\ \text{list} \times 'a\ \text{list} \\
\text{rev\_step} (x \# xs, ys) & = (xs, x \# ys) \\
\text{rev\_step} (\text{[]}, ys) & = (\text{[]}, ys)
\end{align*}
\]
where \( x \neq xs \) is the list being reversed, and \( x \neq ys \) is the partial reversal result. Thus, to reverse a list of size 3 one should call \texttt{rev\_step} 3 times:

\[
\text{rev\_step} ([1, 2, 3], []) = ([2, 3], [1])
\]
\[
\text{rev\_step} (\text{rev\_step} ([1, 2, 3], [])) = ([3], [2, 1])
\]
\[
\text{rev\_step} (\text{rev\_step} (\text{rev\_step} ([1, 2, 3], []))) = ([], [3, 2, 1])
\]

Note that each call to \texttt{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.** \( \text{rev\_step}^{|xs|} (xs, ys) = ([], \text{rev} \, xs @ ys) \)

As a special case this implies \( \text{rev\_step}^{|xs|} (xs, []) = ([], \text{rev} \, xs) \).

### 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 \texttt{deq} and \texttt{enq}. However, they avoid a costly reversal operation once \( f \) becomes empty by preemptively computing a new front \( fr = f @ \text{rev} \, r \) one step at a time using \texttt{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 \texttt{rev\_step} as follows:

\[
1. \quad r' = \text{snd} \, (\text{rev\_step}^{|r|} (r, []))
\]
\[
2. \quad f' = \text{snd} \, (\text{rev\_step}^{|f|} (f, []))
\]
\[
3. \quad fr = \text{snd} \, (\text{rev\_step}^{|f'|} (f', r'))
\]

Phases (1) and (2) are independent and can be performed at the same time, hence, when starting from this configuration

\[
\begin{array}{c|c|c}
\hline
f & f' & r \\
\hline
q_0 & \cdots & q_m \\
\hline
r & \cdots & q_n \\
\hline
\end{array}
\]

after \( \max |f|, |r| \) steps of reversal the state would be the following:

\[
\begin{array}{c|c|c}
\hline
f & f' & r \\
\hline
\cdots & q_m & \cdots \\
\hline
r & \cdots & q_n \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\hline
\ & \ & \ \\
\hline
q_0 & \cdots & q_m \\
\hline
\ & \ & \ \\
\hline
q_m & \cdots & q_0 \\
\hline
\ & \ & \ \\
\hline
q_n & \cdots & q_{m+1} \\
\hline
\end{array}
\]
Phase (3) reverses \( f' \) onto \( r' \) to obtain the same result as a call to `list`:

\[
fr = snd (rev_step f' \| f') (f', r') \\
= rev f' @ r' \\
= rev f' @ snd (rev_step f' \| (r, [])) \\
= rev f' @ rev r \\
= rev (snd (rev_step f' \| (f, []))) @ rev r' \\
= rev (rev f) @ rev r \\
= f @ rev r
\]

by definition of \( fr \)

by definition of \( r' \)

by definition of \( f' \)

by definition of \( f \)

by \( rev \) involution

The resulting front list \( fr \) contains all elements previously in \( f \) and \( r \):

\[
\begin{array}{c}
\mathcal{q}_0 \\
\vdots \\
\mathcal{q}_m \\
\mathcal{q}_m+1 \\
\vdots \\
\mathcal{q}_n
\end{array}
\]

\[
\begin{array}{c}
f \\
rev \quad r
\end{array}
\]

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:

\[
\begin{array}{cccc}
f & f' & r & r' \\
\mathcal{q}_0 & \ldots & \mathcal{q}_m & \mathcal{q}_m+1 & \ldots & \mathcal{q}_n \\
\mathcal{q}_0 & \ldots & \mathcal{q}_m \\
\mathcal{q}_0 & \ldots & \mathcal{q}_m \\
d = 0
\end{array}
\]

\[
\begin{array}{c}
deq \\
\mathcal{q}_0 & \ldots & \mathcal{q}_m \\
\mathcal{q}_0 & \ldots & \mathcal{q}_m
\end{array}
\]

\[
\begin{array}{c}
\mathcal{q}_0 & \ldots & \mathcal{q}_m & \mathcal{q}_m+1 & \ldots & \mathcal{q}_n
\end{array}
\]

\[
\begin{array}{c}
d = 0
\end{array}
\]
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 \(|\text{rear}|\) becomes larger than \(|\text{front}|\), and ensure that all reversal steps are done before \(\text{front}\) runs out of elements or \(\text{rear}\) becomes larger than the new front \((fr)\).

With this strategy, once \(|\text{rear}| = n+1\) and \(|\text{front}| = n\), the reversal processes starts. The first two phases take \(n + 1\) steps \((\max |\text{front}| |\text{rear}|)\) 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 \(\text{front}\) 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 \(\text{rear}\) to become larger than \(\text{front}\) and two more steps in each subsequent operation. Therefore, \(2 \cdot (n + 1)\) steps can occur before \(\text{front}\) 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 \(\text{rear}\) 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 \(\text{rear}\).

20.3.4 Implementation

Queues are implemented using the following record type:

\[
\text{record } \text{'a queue} = \text{lenf} :: \text{nat} \\
\text{front} :: \text{'a list} \\
\text{status} :: \text{'a status} \\
\text{rear} :: \text{'a list} \\
\text{lenr} :: \text{nat}
\]

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 \(\text{make} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ status} \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \text{ queue}\) 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 \(\text{front}\) with \(q\text{front}\), and update its content using \(q\{\text{front} := []\}\). Multiple updates can be composed as \(q\{\text{front} := [], \text{rear} := []\}\).

All values in the queue along with its internal state are stored in the various fields of \('a queue\). Fields \(\text{front}\) and \(\text{rear}\) contain the lists over which all queue operations are performed. The length of \(\text{front}\) and \(\text{rear}\) is recorded in \(\text{lenf}\) and \(\text{lenr}\) (respectively) to avoid calling \text{length}\) whose complexity is
not constant. Finally, \textit{status} tracks the current reversal phase of the queue in a 'a \textit{status} value.

\begin{verbatim}
    datatype 'a status =
        Idle |
        Rev nat ('a list) ('a list) ('a list) ('a list) |
        App nat ('a list) ('a list) |
        Done
\end{verbatim}

Each value of 'a \textit{status} represents either a phase of reversal or the queue's normal operation. Constructor \textit{Idle} signals that no reversal is being performed. \textit{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 \textit{App ok} \(f\ 0\ r\ 0\) case corresponds to phase (3) where both lists are appended to form the new front \((fr)\). In both \textit{App} and \textit{Rev}, the first argument \textit{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 \textit{deq} operations that have occurred so far. Lastly, \textit{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 \textit{exec} and \textit{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 \textit{exec} :: 'a \textit{status} \Rightarrow 'a \textit{status} performs the incremental reversal of the front list by altering the queue's \textit{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 \textit{exec} twice to be able to finish the reversal in time. On \textit{Idle} queues \textit{exec} has no effect. The implementation of \textit{exec} is an extension of \textit{rev_step} with specific considerations for each \textit{status} value and is defined as follows:

\begin{verbatim}
    exec :: 'a status \Rightarrow '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
\end{verbatim}

If the \textit{status} is \textit{Rev ok} \(f\ f'\ r\ r'\), then \textit{exec} performs two (or one if \(f = []\)) simultaneous reversal steps from \(f\) and \(r\) into \(f'\) and \(r'\); moreover \textit{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:

\[
\text{invalidate :: 'a status ⇒ 'a status}
\]

\[
\text{invalidate (Rev ok f f' r r')} = \text{Rev} (ok - 1) f f' r r'
\]

\[
\text{invalidate (App 0 f' (\_ \# r'))} = \text{Done} r'
\]

\[
\text{invalidate (App ok f' r')} = \text{App} (ok - 1) f' r'
\]

\[
\text{invalidate} s = s
\]

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 exec2, as its name suggests, applies exec twice and updates the queue accordingly if Done is returned.

\[
\text{exec2 :: 'a queue ⇒ 'a queue}
\]

\[
\text{exec2 q} = (\text{case exec (exec q)} of
\]

\[
\text{Done fr ⇒ q(status = Idle, front = fr)} |\]

\[
\text{newstatus ⇒ q(status = newstatus[])}\]

\[
\text{check :: 'a queue ⇒ 'a queue}
\]

\[
\text{check q}
\]

\[
= (\text{if lenr q ≤ lenf q then exec2 q}
\]

\[
\text{else let newstate = Rev 0 (front q) [] (rear q) []}\]

\[
\text{in exec2}
\]

\[
(q[lenf := lenf q + lenr q, status := newstate, rear := [],
\]

\[
\text{lenr := 0]})
\]
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 their length. To perform all “internal” operations, both functions call check. Additionally, deq uses invalidate to mark elements as removed.

Function check calls exec2 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 specific properties to guarantee a Done result after a (finite) number of calls to exec. The predicate inv_st defines these properties as follows:

\[
\text{inv\_st} :: \text{'a status} \Rightarrow \text{bool} \\
\text{inv\_st} (\text{Rev ok f f'} r r') = (|f| + 1 = |r| \land |f'| = |r'| \land ok \leq |f'|)
\]
\[ inv\_st\ (\text{App ok } f'\ r') = (ok \leq |f'| \land |f'| < |r'|) \]
\[ inv\_st\ \text{Idle} = \text{True} \]
\[ inv\_st\ (\text{Done } _) = \text{True} \]

First, \( inv\_st \) ensures for pattern, \( \text{Rev ok } f'\ r' \) that arguments \( f \) and \( r \) follow the reversal strategy, and counter \( ok \) is only ever increased as elements are added to \( f' \). Similarly, for \( \text{App ok } f'\ r' \), it must follow that \( r' \) remains larger than \( f' \), and \( |f'| \) provides an upper bound for \( ok \). All other patterns trivially fulfill the invariant.

The consistency of 'a queue values is then defined as an extension of \( inv\_st \) and considers all the other fields in the queue. The predicate \( \text{invar} \) defines the required properties:

\[
\text{invar } q = (\text{lenf } q = |\text{front\_list } q| \land \text{lenr } q = |\text{rear\_list } q| \land
\]
\[
\text{lenr } q \leq \text{lenf } q \land
\]
\[
\begin{cases}
  \text{Rev ok } f'\ r'\ _\ = \Rightarrow \\
  2 \cdot \text{lenr } q \leq |f'| \land \text{ok } \neq 0 \land \\
  2 \cdot |f'| + \text{ok } + 2 \leq 2 \cdot |\text{front } q| \\
  \text{App ok } _\ r\ _\ = \Rightarrow 2 \cdot \text{lenr } q \leq |r| \land \text{ok } + 1 \leq 2 \cdot |\text{front } q| \\
  _\ = \Rightarrow \text{True} \land \\
  (\exists \text{rest}. \text{front\_list } q = \text{front } q \circ \text{rest}) \land \\
  (\exists \text{fr}. \text{status } q = \text{Done } fr) \land \\
  \text{inv\_st } (\text{status } q) 
\end{cases}
\]

Predicate \( \text{lenr } q = |\text{rear\_list } q| \) ensures \( \text{lenr} \) is equal to the length of the queue's rear, where function \( \text{rear\_list } q \), defined as \((\text{rev } \circ \text{rear}) q\), produces the rear list in canonical order. Likewise, \( \text{lenf } q = |\text{front\_list } q| \) matches \( \text{lenf} \) to the queue's front. However, function \( \text{front\_list} \) warrants special attention as it must compute the list representing the front of the queue even during a reversal. \( \text{front\_list} \) is defined as follows:

\[
\text{front\_list} :: 'a\ queue \Rightarrow 'a\ list
\]
\[
\text{front\_list } q = (\begin{cases}
  \text{case } \text{status } q \text{ of } \text{Idle } \Rightarrow \text{front } q \\
  \text{Rev ok } f'\ r' \Rightarrow \text{rev } (\text{take } ok f') \circ \text{rev } r \circ r' \\
  \text{App ok } f'\ x \Rightarrow \text{rev } (\text{take } ok f') \circ x | \text{Done } f \Rightarrow f
\end{cases})
\]
For case $App ok f' r'$, the front list corresponds to the final result of the stepped reversal (20.1) but only elements in $f'$ that are still in the queue, denoted by $take ok f'$, are considered. Analogously for $Rev ok f f' r r'$, both stepped reversal results are appended and only relevant elements in $f'$ are used, however, rear lists $r$ and $r'$ are reversed again to achieve canonical order.

Continuing with $invar$, inequality $lenr q \leq lenf q$ is the main invariant in our reversal strategy, and by the previous two equalities must holds even as internal operations occur. Furthermore, predicate $\exists \text{ rest. } front\_list q = front q \@ \text{ rest}$ ensures $front q$ is contained within $front\_list q$, thus preventing any mismatch between the internal state and the queue’s front. Given that $exec2$ is the only function that manipulates a queue’s $status$, it holds that $\exists fr. \ status q = Done fr$ since any internal $Done$ result is replaced by $Idle$.

The case distinction on $status q$ places size bounds on internal lists $front$ and $rear$ ensuring the front does not run out of elements and the rear never grows beyond $lenr q \leq lenf q$. In order to clarify some of formulations used in this part of $invar$, consider the following correspondences, which hold once the reversal process starts:

- $lenr q$ corresponds to the number of $enq$ operations performed so far, and $2 \cdot lenr q$ denotes the $exec$ applications in those operations.
- $|front q|$ corresponds to the number of $deq$ operations that can be performed before $front q$ is exhausted. Therefore, $2 \cdot |front q|$ is the minimum number of $exec$ applications the queue will be able to do at any given point.
- On $Rev ok f f' r r'$ status, $|f'|$ corresponds to the number of applications of $exec$ performed so far and the internal length of front being constructed. Expression $|r|$ is the analogous for a $App ok f r$.
- From a well formed $App ok f r$ it takes $ok + 1$ applications of $exec$ to reach $Done$. Since, the base case of $App$ is obtained after $ok$ applications, and the transition into $Done$ takes an extra step.
- From a well formed $Rev ok f f' r r'$ it takes $2 \cdot |f'| + ok + 2$ applications of $exec$ to reach $Done$. Since, the base case of $Rev$ is obtained after $|f'|$ applications (incrementing $ok$ by the same amount), the transitioning into $App$ takes one step, and $ok + |f'|$ extra steps are need to reach $Done$ from $App$.

In the $Rev ok f f' r r'$ case, $2 \cdot lenr q \leq |f'|$ ensures $f'$ grows larger with every $enq$ operation and the internal list is at least twice the length of the queue’s rear. Additionally, the value of $ok$ cannot be 0 as this either marks the beginning of a reversal which calls $exec2$ immediately, or signals that elements in $front q$ have run out. Finally, to guarantee the reversal process can finish before the $front q$ is exhausted the number of $exec$ applications
before reaching \textit{Done} must be less than the minimum number of applications possible, denoted by $2 \cdot |f| + ok + 2 \leq |\text{front } q|$. 

Case \textit{App ok f r} has similar invariants, with equation $2 \cdot \text{lenr } q \leq |r|$ bounding the growth of $r$ as it was previously done with $f'$. Moreover, $ok + 1 \leq 2 \cdot |\text{front } q|$ ensures $\text{fron } q$ is not exhausted before the reversal is completed.

Using $\text{invar}$ we can prove the properties of \textit{enq} and \textit{deq} required by the \textit{Queue} ADT using this abstraction following abstraction function:

\begin{verbatim}
list :: 'a queue => 'a list
list q = front_list q @ rear_list q
\end{verbatim}

Proofs follow mostly by case distinction on the \textit{status} field. Invariants $\text{invar}$ and $\text{inv_st}$ are used to filter invalid \\textquote{a queue} values, and to simplify terms based on their relationship with other terms (e.g. $f = [] \land |f| + 1 = |r| \rightarrow (\exists x. r = [x])$). All other properties in \textit{Queue} are straightforward.

\section*{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].
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`:

```plaintext
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
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 `h l a r = splay x t; if a = x, return `h l a r`; 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 `h l a r = splay x t; if a ≠ x, return `h l a r`. Otherwise follow the deletion-by-replacing paradigm (Section 5.2.1): if `l ≠ ∅`, 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 ≠ ∅` then `splay_max t` is of the form `h t0, m, ∅`. The definition `splay_max ∅ = ∅` is not really needed (`splay_max` is always called with non-(`∅`) argument) but some lemmas can be stated more slickly with this definition.

---

**Fig. 21.1. Function `splay`**

\[
\text{splay } x \langle AB, b, CD \rangle = (\text{case } \text{cmp } x b \text{ of} \\
\quad LT \Rightarrow \text{case } AB \text{ of} \\
\quad \quad \langle \rangle \Rightarrow \langle AB, b, CD \rangle \\
\quad \quad \langle A, a, B \rangle \Rightarrow \\
\quad \quad \quad \text{case } \text{cmp } x a \text{ of} \\
\quad \quad \quad \quad LT \Rightarrow \text{if } A = \langle \rangle \text{ then } \langle A, a, \langle B, b, CD \rangle \rangle \\
\quad \quad \quad \quad \quad \text{else case } \text{splay } x A \text{ of} \\
\quad \quad \quad \quad \quad \quad \langle A_1', a', A_2 \rangle \Rightarrow \langle A_1', a', \langle A_2, a, \langle B, b, CD \rangle \rangle \rangle \\
\quad \quad \quad \quad \quad \quad \text{EQ } \Rightarrow \langle A, a, \langle B, b, CD \rangle \rangle \\
\quad \quad \quad \quad \quad \quad \text{GT } \Rightarrow \text{if } B = \langle \rangle \text{ then } \langle A, a, \langle B, b, CD \rangle \rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{else case } \text{splay } x B \text{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \langle B_1', b', B_2 \rangle \Rightarrow \langle \langle A, a, B_1 \rangle, b', \langle B_2, b, CD \rangle \rangle \\
\quad \quad \quad \quad \quad \quad \text{EQ } \Rightarrow \langle AB, b, CD \rangle \\
\quad \quad \quad \quad \quad \quad \text{GT } \Rightarrow \text{case } CD \text{ of} \\
\quad \quad \quad \quad \quad \quad \quad \langle \rangle \Rightarrow \langle AB, b, CD \rangle \\
\quad \quad \quad \quad \quad \quad \quad \langle C, c, D \rangle \Rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{case } \text{cmp } x c \text{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad LT \Rightarrow \text{if } C = \langle \rangle \text{ then } \langle AB, b, C_1 \rangle, c, D \rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else case } \text{splay } x C \text{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \langle C_1, c', C_2 \rangle \Rightarrow \langle AB, b, C_1 \rangle, c', \langle C_2, c, D \rangle \rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{EQ } \Rightarrow \langle AB, b, C_1 \rangle, c, D \rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{GT } \Rightarrow \text{if } D = \langle \rangle \text{ then } \langle AB, b, C_1 \rangle, c, D \rangle \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else case } \text{splay } x D \text{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \langle D_1, d, D_2 \rangle \Rightarrow \langle AB, b, C_1 \rangle, c, D_1 \rangle, d, D_2 \rangle)
\]
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:

\[
\begin{align*}
\text{splay } a \ t &= \langle \rangle \iff t = \langle \rangle \\
\text{splay\_max } t &= \langle \rangle \iff t = \langle \rangle \\
\text{Correctness of } \text{isin} \\
\text{sorted } (\text{inorder } t) &\iff \text{isin } t \ x = (x \in \text{set } (\text{inorder } t))
\end{align*}
\]
follows directly from this easy inductive property of *splay*:

\[
splay \ x \ t = \langle l, a, r \rangle \land \text{sorted (inorder } t) \rightarrow (x \in \text{set (inorder } t)) = (x = a)
\]

Correctness of *insert* and *delete*

\[
sorted \ (\text{inorder } t) \rightarrow \text{inorder (insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)
\]

\[
sorted \ (\text{inorder } t) \rightarrow \text{inorder (delete } x \ t) = \text{del_list } x \ (\text{inorder } t)
\]

relies on the following characteristic inductive properties of *splay*:

\[
inorder \ (splay \ x \ t) = \text{inorder } t
\]

\[
sorted \ (\text{inorder } t) \land \text{splay } x \ t = \langle l, a, r \rangle \rightarrow \text{sorted (inorder } l \ @ \ x \ # \ \text{inorder } r)
\]

Correctness of *delete* also needs the inductive property

\[
splay\_\text{max } t = \langle l, a, r \rangle \land \text{sorted (inorder } t) \rightarrow \text{inorder } l \ @ [a] = \text{inorder } t \land r = \langle \rangle
\]

Note that \(\text{inorder (splay } x \ t) = \text{inorder } t\) 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 \(\text{bst} = \text{sorted } \circ \text{inorder}\). Similarly, (21.1) implies that *splay* preserves \(\text{bst}\). Thus we may assume the invariant \(\text{bst}\) in the amortized analysis.

These two easy size lemmas are used implicitly below:

\[
|\text{splay } a \ t| = |t| \quad |\text{splay\_max } t| = |t|
\]

### 21.3 Amortized Analysis

This section shows that *splay*, insertion and deletion all have amortized logarithmic complexity.

We define the potential \(\Phi\) of a tree as the sum of the potentials \(\varphi\) of all nodes:

\[
\Phi :: 'a \text{ tree } \Rightarrow \text{real}
\]

\[
\Phi \ \langle \rangle = 0
\]

\[
\Phi \ \langle l, a, r \rangle = \varphi \ \langle l, a, r \rangle + \Phi \ l + \Phi \ r
\]

\[
\varphi \ t \equiv \log_2 |t|_1
\]
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.** $bst \ t \land \langle l, x, r \rangle \in \text{subtrees} \ t \rightarrow A_{splay \ x \ t} \leq 3 \cdot (\varphi \ t - \varphi \langle l, x, r \rangle) + 1$

**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 $t = \langle \lambda, b, c \rangle$ and $x < b$: from $\langle l, x, r \rangle \in \text{subtrees} \ t$ it follows that $x \in \text{set_tree} \ t$ but because $bst \ t$ and $x < b$ this implies that $x \in \text{set_tree} \langle \rangle$, a contradiction.

There are three feasible base cases. The case $t = \langle \_, b, \_ \rangle$ is easy. We consider one of the two other symmetric cases. Let $t = \langle A, x, B \rangle$, $b, C \rangle$ and $t' = splay \ x \ t = \langle A, x, B, b, C \rangle$.

$$A_{splay \ x \ t} = \Phi t' - \Phi t + 1$$

by definition of $A_{splay}$ and $T_{splay}$

$$= \varphi t' + \varphi \langle B, b, C \rangle - \varphi t - \varphi \langle A, x, B \rangle + 1$$

by definition of $\Phi$

$$\leq \varphi t - \varphi \langle A, x, B \rangle + 1$$

because $\varphi \langle B, b, C \rangle \leq \varphi t$

$$\leq 3 \cdot (\varphi t - \varphi \langle A, x, B \rangle) + 1$$

because $\varphi \langle A, x, B \rangle \leq \varphi t$

$$= 3 \cdot (\varphi t' - \varphi \langle l, x, r \rangle) + 1$$

because $bst \ t \land \langle l, x, r \rangle \in \text{subtrees} \ t$

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 $X \in \text{subtrees} \ A$. We abbreviate compound trees like $\langle A, a, B \rangle$ by the names of their subtrees, in this case $AB$. First note that

$$\varphi \ A_1 A_2 BC = \varphi \ ABC$$

(*)
because \(|A'| = |splay x A| = |A|\). We can now prove the claim:

\[
A_{splay} x ABC = T_{splay} x A + 1 + \phi A_1 A_2 BC - \phi ABC
\]

\[
= T_{splay} x A + 1 + \phi A_1 + \phi A_2 + \varphi A_2 BC + \varphi BC - \phi A - \varphi AB
\]

by the definition of \(\varphi\) and \(\phi\).

\[
= T_{splay} x A + \phi A' - \phi A' - \phi A + \varphi A_2 BC + \varphi BC - \varphi AB + 1
\]

\[
= A_{splay} x A + \varphi A_2 BC + \varphi BC - \varphi AB - \varphi A' + 1
\]

\[
\leq 3 \cdot \varphi A + \varphi A_2 BC + \varphi BC - \varphi AB - \varphi A' - 3 \cdot \varphi X + 2
\]

by IH and \(X \in \text{subtrees } A\),

\[
= 2 \cdot \varphi A + \varphi A_2 BC + \varphi BC - \varphi AB - 3 \cdot \varphi X + 2
\]

because \(\varphi A = \varphi A'\)

\[
< \varphi A + \varphi A_2 BC + \varphi BC - 3 \cdot \varphi X + 2
\]

because \(\varphi A_2 < \varphi AB\)

\[
< \varphi A_2 BC + 2 \cdot \varphi ABC - 3 \cdot \varphi X + 1
\]

because \(1 + \log x + \log y < 2 \cdot \log (x + y)\) if \(x, y > 0\)

\[
< 3 \cdot (\varphi ABC - \varphi X) + 1
\]

because \(\varphi A_2 BC < \varphi ABC\).

Now we consider the so-called zig-zag case:

\[
\begin{array}{ccc}
  b & \rightarrow & b' \\
  / \backslash & \rightarrow & / \backslash \\
  a & C & a & C & a & b \\
  / \backslash & & / \backslash & & / \backslash & \\
  A & B & A & b' & A & B_1 & B_2 & C \\
  / \backslash & & & & & & & \\
  B_1 & B_2 & & & & & & \\
\end{array}
\]

This is the case where \(a < x < b\) and \(B \neq \emptyset\). On the left we have the input and on the right the output of \(splay x\). Because \(B \neq \emptyset\), \(splay x B = \langle B_1, b', B_2 \rangle =: B'\) 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 \(X \in \text{subtrees } B\). The proof is very similar to the zig-zag case, the same naming conventions apply and we omit some details:

\[
A_{splay} x ABC = T_{splay} x A + 1 + \phi AB_1 B_2 C - \phi ABC
\]

\[
= A_{splay} x B + \phi AB_1 + \phi B_2 C - \phi AB - \phi B' + 1
\]

using \(\phi AB_1 B_2 C = \varphi ABC\)

\[
\leq 3 \cdot \varphi B + \varphi AB_1 + \varphi B_2 C - \varphi AB - \varphi B' - 3 \cdot \varphi X + 2
\]

by IH and \(X \in \text{subtrees } B\)

\[
= 2 \cdot \varphi B + \varphi AB_1 + \varphi B_2 C - \varphi AB - 3 \cdot \varphi X + 2
\]

because \(\varphi B = \varphi B'\)

\[
< \varphi B + \varphi AB_1 + \varphi B_2 C - 3 \cdot \varphi X + 2
\]

because \(\varphi B < \varphi AB\)

\[
< \varphi B + 2 \cdot \varphi ABC - 3 \cdot \varphi X + 1
\]

because \(1 + \log x + \log y < 2 \cdot \log (x + y)\) if \(x, y > 0\)

\[
< 3 \cdot (\varphi ABC - \varphi X) + 1
\]

because \(\varphi B < \varphi ABC\) \(\square\)
Because $\varphi(l, x, r) \geq 1$, the above theorem implies

Corollary 21.2. $bst t \land x \in set\_tree t \rightarrow A_{splay} x t \leq 3 \cdot (\varphi t - 1) + 1$

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. $t \neq \langle \rangle \land bst t \rightarrow$

$(\exists y \in set\_tree t. splay y t = splay x t \land T_{splay} y t = T_{splay} x t)$

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. $bst t \rightarrow A_{splay} x t \leq 3 \cdot \varphi t + 1$

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. $bst t \rightarrow T_{insert} x t + \Phi (insert x t) - \Phi t \leq 4 \cdot \varphi t + 3$

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:

$$A_{splay\_max} t = T_{splay\_max} t + \Phi (splay\_max t) - \Phi t$$

An inductive proof and then a simple case analysis yield

$$t \neq \langle \rangle \rightarrow A_{splay\_max} t \leq 3 \cdot (\varphi t - 1) + 1$$

$$A_{splay\_max} t \leq 3 \cdot \varphi t + 1$$

Based on the canonical definitions of $T_{delete}$ and $T_{splay\_max}$ the amortized logarithmic complexity of $delete$ follows:

$$bst t \rightarrow T_{delete} a t + \Phi (delete a t) - \Phi t \leq 6 \cdot \varphi t + 3$$

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 these numbers one by one creates a tree of height $k$. 
Bibliographic Remarks

Splay trees were invented and analyzed by Sleator and Tarjan \cite{70} for which they received the 1999 ACM Paris Kanellakis Theory and Practice Award \cite{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 \cite{69, 53, 54, 55}. 
Skew Heaps

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.

22.1 Implementation of ADT Priority_Queue_Merge

The central operation is \texttt{merge}:

\begin{verbatim}
merge :: 'a tree \Rightarrow 'a tree \Rightarrow 'a tree
merge () t = t
merge t () = t
\textcolor{red}{merge (\\langle l_1, a_1, r_1 \rangle =: t_1) (\\langle l_2, a_2, r_2 \rangle =: t_2) =
\begin{cases}
\text{if } a_1 \leq a_2 & \text{then } \langle \text{merge } t_2 \ r_1, a_1, l_1 \rangle \ \\
\text{else } & \langle \text{merge } t_1 \ r_2, a_2, l_2 \rangle
\end{cases}}
\end{verbatim}

The remaining operations (\{\}, insert, get\_min and del\_min) are defined as in Section 14.1.

The following properties of \texttt{merge} have easy inductive proofs:

\begin{align*}
|merge t_1 \ t_2| &= |t_1| + |t_2| \\
\text{mset\_tree} (merge t_1 \ t_2) &= mset\_tree t_1 + mset\_tree t_2 \\
heap \ t_1 \wedge \heap \ t_2 \rightarrow & \heap (merge t_1 \ t_2)
\end{align*}

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 these 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:

$$\Phi : \text{'a tree} \Rightarrow \text{int}$$
$$\Phi (\langle \rangle) = 0$$
$$\Phi (\langle l, _, r \rangle) = \Phi (l) + \Phi (r) + rh (l, r)$$

$$rh : \text{'a tree} \Rightarrow \text{'a tree} \Rightarrow \text{nat}$$
$$rh (l, r) = (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)$$

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 ($rlh$):

$$lrh : \text{'a tree} \Rightarrow \text{nat}$$
$$lrh (\langle \rangle) = 0$$
$$lrh (\langle l, _, r \rangle) = rh (l, r) + lhr (l)$$

$$rlh : \text{'a tree} \Rightarrow \text{nat}$$
$$rlh (\langle \rangle) = 0$$
$$rlh (\langle l, _, r \rangle) = 1 - rh (l, r) + rlh (r)$$

The following properties have automatic inductive proofs:

$$2^{lrh} t \leq |t| + 1 \quad 2^{rlh} t \leq |t| + 1$$

They imply

$$lrh t \leq \lg |t|_1 \quad rlh t \leq \lg |t|_1$$

Now we are ready for the amortized analysis. All canonical time functions can be found in Appendix B.8. The key lemma is an upper bound of the amortized complexity of merge in terms of $lrh$ and $rlh$: 
Lemma 22.1. $T_{\text{merge}}(t_1, t_2) + \Phi (merge t_1, t_2) - \Phi t_1 - \Phi t_2 \leq lrh(merge t_1, t_2) + rlh t_1 + rlh t_2 + 1$

Proof by induction on the computation of $merge$. We consider only the node-node case: let $t_1 = (l_1, a_1, r_1)$ and $t_2 = (l_2, a_2, r_2)$. W.l.o.g. assume $a_1 \leq a_2$. Let $m = merge t_2 r_1$.

$$
T_{\text{merge}}(t_1, t_2) + \Phi (merge t_1, t_2) - \Phi t_1 - \Phi t_2
= T_{\text{merge}}(t_2, r_1) + 1 + \Phi m + \Phi l_1 + rh m l_1 - \Phi t_1 - \Phi t_2
= T_{\text{merge}}(t_2, r_1) + 1 + \Phi m + rh m l_1 - \Phi r_1 - rh r_1 r_1 - \Phi t_2
\leq lrh m + rlh t_2 + rlh r_1 + rh m l_1 + 2 - rh l_1 r_1 \quad \text{by IH}
= lrh m + rlh t_2 + rlh t_1 + rh m l_1 + 1
= lrh(merge t_1, t_2) + rlh t_1 + rlh t_2 + 1 \quad \square
$$

As a consequence we can prove the following logarithmic upper bound on the amortized complexity of $merge$:

$$
T_{\text{merge}}(t_1, t_2) + \Phi (merge t_1, t_2) - \Phi t_1 - \Phi t_2
\leq lrh(merge t_1, t_2) + rlh t_1 + rlh t_2 + 1 \quad \text{by Lemma 22.1}
\leq lg |merge t_1, t_2| + lg |t_1| + lg |t_2| + 1
\leq lg (|t_1| + |t_2| - 1) + lg |t_1| + lg |t_2| + 1
\quad \text{because } |merge t_1, t_2| = |t_1| + |t_2|
\leq lg (|t_1| + |t_2|) + 2 \cdot lg (|t_1| + |t_2|) + 1
\quad \text{because } lg x + lg y \leq 2 \cdot lg (x + y) \text{ if } x, y > 0
= 3 \cdot lg (|t_1| + |t_2|) + 1
$$

The amortized complexity of insertion and deletion follows easily from the complexity of $merge$:

$$
T_{\text{insert}}(a, t) + \Phi (insert a, t) - \Phi t \leq 3 \cdot lg (|t_1| + 2) + 2
T_{\text{del_min}}(t) + \Phi (del_min t) - \Phi t \leq 3 \cdot lg (|t_1| + 2) + 2
$$

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 that 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:

```
datatype 'a heap = Empty | Hp 'a ('a heap list)
```

The abstraction function to multisets and the invariant follow the heap paradigm:

```
mset_heap :: 'a heap ⇒ 'a multiset
mset_heap Empty = {}
mset_heap (Hp x hs) = {x} + ∑ₙ mset (map mset_heap hs)

pheap :: 'a heap ⇒ bool
pheap Empty = True
pheap (Hp x hs) = (∀ h ∈ set hs. (∀ y ∈ mset_heap h. x ≤ y) ∧ pheap h)
```

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:

```haskell
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:

```haskell
merge :: 'a heap ⇒ 'a heap ⇒ 'a heap
merge h Empty = h
merge Empty h = h
merge (Hp x hxs) (Hp y hys) = (if x < y then Hp x (hy # hxs) else Hp y (hx # hys))
```

Thus merge and insert have constant running time. All the work is offloaded on del_min which just calls merge_pairs:

```haskell
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. By reformulating the definition in terms of these two passes, we obtain a more readable formulation with the same running time:
23.2 Amortized Analysis

\[ \text{del\_min} :: 'a heap \Rightarrow 'a heap \]
\[ \text{del\_min} \; \text{Empty} = \text{Empty} \]
\[ \text{del\_min} \; (\text{Hp} \; x \; hs) = \text{pass}_2 \; (\text{pass}_1 \; hs) \]

\[ \text{pass}_1 :: 'a \text{ heap list} \Rightarrow 'a \text{ heap list} \]
\[ \text{pass}_1 \; (h_1 \; # \; h_2 \; # \; hs) = \text{merge} \; h_1 \; h_2 \; # \; \text{pass}_1 \; hs \]
\[ \text{pass}_1 \; hs = hs \]

\[ \text{pass}_2 :: 'a \text{ heap list} \Rightarrow 'a \text{ heap} \]
\[ \text{pass}_2 \; [] = \text{Empty} \]
\[ \text{pass}_2 \; (h \; # \; hs) = \text{merge} \; h \; (\text{pass}_2 \; hs) \]

The proof of \( \text{pass}_2 \; (\text{pass}_1 \; hs) = \text{merge\_pairs} \; hs \) is an easy induction. Clearly \text{del\_min} can take linear time but it will turn out that the constant-time \text{insert} saves enough to guarantee amortized logarithmic complexity for both insertion and deletion.

We base the correctness proofs on the \text{merge\_pairs} version of \text{del\_min}. From the following lemmas (all proofs are routine inductions)

\[ h \neq \text{Empty} \longrightarrow \text{get\_min} \; h \in \omega \; \text{mset\_heap} \; h \]
\[ h \neq \text{Empty} \land pheap \; h \land x \in \omega \; \text{mset\_heap} \; h \longrightarrow \text{get\_min} \; h \leq x \]
\[ \text{mset\_heap} \; (\text{merge} \; h_1 \; h_2) = \text{mset\_heap} \; h_1 + \text{mset\_heap} \; h_2 \]
\[ \text{mset\_heap} \; (\text{merge\_pairs} \; hs) \]
\[ = \sum_{x} (\text{image\_mset} \; \text{mset\_heap} \; (\text{mset} \; hs)) \]
\[ h \neq \text{Empty} \longrightarrow \text{mset\_heap} \; (\text{del\_min} \; h) = \text{mset\_heap} \; h - \{ \text{get\_min} \; h \} \]
\[ pheap \; h_1 \land pheap \; h_2 \longrightarrow pheap \; (\text{merge} \; h_1 \; h_2) \]
\[ (\forall h \in \text{set} \; hs. \; pheap \; h) \longrightarrow pheap \; (\text{merge\_pairs} \; hs) \]
\[ pheap \; h \longrightarrow pheap \; (\text{del\_min} \; h) \]

the properties in the specifications \text{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 \( \text{Hp} \; x \; hs \) is expressed as the tree \( \langle hs, x, \langle \rangle \rangle \) and a list of heaps \([ \text{Hp} \; x_1 \; hs_1, \; \text{Hp} \; x_2 \; hs_2, \ldots] \) is expressed
as the tree \( \langle h_1, x_1, \langle h_2, x_2, \ldots \rangle \ldots \rangle \). 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 \( \text{pass}_1/\text{pass}_2 \) version of \( \text{del}\_\text{min} \). The correctness proof is very similar to what we saw in the previous section. We merely display the two invariants:

\[
\text{is}\_\text{root} :: \text{a tree} \Rightarrow \text{bool}
\]

\[
\text{is}\_\text{root} \ hp = (\text{case } hp \ of \ \emptyset \Rightarrow \text{True} \ | \ \_ \Rightarrow \_ \Rightarrow \_ = \emptyset)
\]
23.2 Amortized Analysis

\begin{align*}
pheap & : \text{ 'a tree } \Rightarrow \text{ bool} \\
pheap () &= \text{ True} \\
pheap (l, x, r) &= ((\forall y \in \text{ set_tree } l \cdot x \leq y) \land pheap l \land pheap r)
\end{align*}

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{align*}
\Phi & : \text{ 'a tree } \Rightarrow \text{ real} \\
\Phi () &= 0 \\
\Phi (l, x, r) &= \lg |(l, x, r)| + \Phi l + \Phi r
\end{align*}

These easy inductive size properties are frequently used implicitly below:

\begin{align*}
|\text{link } hp| &= |hp| \\
|\text{pass}_1 hp| &= |hp| \\
|\text{pass}_2 hp| &= |hp| \\
is\_\text{root } h_1 \land is\_\text{root } h_2 &\rightarrow |\text{merge } h_1 \ h_2| = |h_1| + |h_2|
\end{align*}

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.** \(is\_\text{root } hp\) \(\rightarrow\) \(\Phi (\text{insert } x \ hp) - \Phi hp \leq \lg (|hp| + 1)\)

For \text{merge} it needs a bit more work:

**Lemma 23.2.** \(h_1 = \langle h_{s_1}, x_1, \langle \rangle \rangle\) \(\land\) \(h_2 = \langle h_{s_2}, x_2, \langle \rangle \rangle\) \(\rightarrow\)

\(\Phi (\text{merge } h_1 \ h_2) - \Phi h_1 - \Phi h_2 \leq \lg (|h_1| + |h_2|) + 1\)

**Proof.** From

\begin{align*}
\Phi (\text{merge } h_1 \ h_2) &= \Phi (\text{link } \langle h_{s_1}, x_1, h_2 \rangle) \\
&= \Phi h_{s_1} + \Phi h_{s_2} + \lg (|h_{s_1}| + |h_{s_2}| + 1) + \lg (|h_{s_1}| + |h_{s_2}| + 2) \\
&= \Phi h_{s_1} + \Phi h_{s_2} + \lg (|h_{s_1}| + |h_{s_2}| + 1) + \lg (|h_1| + |h_2|)
\end{align*}

it follows that

\begin{align*}
\Phi (\text{merge } h_1 \ h_2) - \Phi h_1 - \Phi h_2 &= \lg (|h_{s_1}| + |h_{s_2}| + 1) + \lg (|h_1| + |h_2|) \\
&- \lg (|h_{s_1}| + 1) - \lg (|h_{s_2}| + 1)
\end{align*}
\[ \leq \lg (|h_1| + |h_2|) + 1 \]

because \( \lg (1 + x + y) \leq 1 + \lg (1 + x) + \lg (1 + y) \) if \( x, y \geq 0 \)

Now we come to the core of the proof, the analysis of \( \text{del_min} \). 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 \( \text{len} \):

\[
\text{len} :: \text{a tree} \Rightarrow \text{nat}
\]

\[
\text{len} () = 0
\]

\[
\text{len} (_, _, r) = 1 + \text{len} r
\]

Therefore we have to show that the potential change compensates for this linear work. Our main goal is this:

**Theorem 23.3.** \( \Phi (\text{del_min} \langle h_s, x, () \rangle) - \Phi \langle h_s, x, () \rangle \)

\[ \leq 2 \cdot \lg (|h_s| + 1) - \text{len} h_s + 2 \]

It will be proved in two steps: First we show that \( \text{pass}_1 \) frees enough potential to compensate for the work linear in \( \text{len} h_s \) and increases the potential only by a logarithmic term. Then we show that the increase due to \( \text{pass}_2 \) is also only at most logarithmic. Combining these results one easily shows that the amortized running time of \( \text{del_min} \) is indeed logarithmic.

First we analyze the potential difference caused by \( \text{pass}_1 \):

**Lemma 23.4.** \( \Phi (\text{pass}_1 h_s) - \Phi h_s \leq 2 \cdot \lg (|h_s| + 1) - \text{len} h_s + 2 \)

**Proof** by induction on the computation of \( \text{pass}_1 \). The base cases are trivial. We focus on the induction step. Let \( t = \langle h_s, x, \langle h_s, y, h_s \rangle \rangle, n_1 = |h_s|, n_2 = |h_s|, m = |h_s| \).

\[
\Phi (\text{pass}_1 t) - \Phi t
\]

\[ = \lg (n_1 + n_2 + 1) - \lg (n_2 + m + 1) + \Phi (\text{pass}_1 h_s) - \Phi h_s \]

\[ \leq \lg (n_1 + n_2 + 1) - \lg (n_2 + m + 1) + 2 \cdot \lg (m + 1) \quad \text{by IH} \]

\[ \leq 2 \cdot \lg (n_1 + n_2 + m + 1) - \lg (n_2 + m + 1) + \lg (m + 1) - \text{len} h_s \]

\[ \leq 2 \cdot \lg (n_1 + n_2 + m + 2) - \text{len} h_s \]

\[ = 2 \cdot \lg |t| - \text{len} t + 2 \]

\[ \leq 2 \cdot \lg (|t| + 1) - \text{len} t + 2 \quad \square \]

Now we turn to \( \text{pass}_2 \):

**Lemma 23.5.** \( h_s \neq () \rightarrow \Phi (\text{pass}_2 h_s) - \Phi h_s \leq \lg |h_s| \)
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 $pass_2$:

$$\exists hs_3 \; z. \; pass_2 \langle hs_2, y, r \rangle = \langle hs_3, \langle \rangle \rangle$$

The proof is a straightforward induction on $r$. This implies $|hs_3| + 1 = |hs|$ and thus

$$\Phi (\text{link} \langle hs_1, x, pass_2 \langle hs \rangle \rangle) - \Phi hs_1 - \Phi (pass_2 \langle hs \rangle) = \lg (|hs_1| + |hs| + 1) + \lg (|hs_1| + |hs|) - \lg |hs| \quad (*)$$

Thus the overall claim follows:

$$\Phi \langle pass_2 \langle hs_1, x, hs \rangle \rangle - \Phi \langle hs_1, x, hs \rangle = \Phi \langle \text{link} \langle hs_1, x, pass_2 \langle hs \rangle \rangle \rangle - \Phi hs_1 - \Phi hs - \lg (|hs_1| + |hs| + 1)
= \Phi \langle pass_2 \langle hs \rangle \rangle - \Phi hs + \lg (|hs_1| + |hs|) - \lg |hs| \quad \text{by (*)}
\leq \lg (|hs_1| + |hs|) \quad \text{by IH}
\leq \lg (|hs_1, x, hs|)$$

**Corollary 23.6.** $\Phi \langle pass_2 \langle hs \rangle \rangle - \Phi hs \leq \lg (|hs| + 1)$

Finally we can prove Theorem 23.3:

$$\Phi \langle \text{del_min} \langle hs, x, \langle \rangle \rangle \rangle - \Phi \langle hs, x, \langle \rangle \rangle = \Phi \langle pass_2 \langle \text{pass_1} \langle hs \rangle \rangle \rangle - \lg (|hs| + 1) - \Phi hs
\leq \Phi \langle \text{pass_1} \langle hs \rangle \rangle - \Phi hs \quad \text{by Corollary 23.6}
\leq 2 \cdot \lg (|hs| + 1) - \text{len} \; hs + 2 \quad \text{by Lemma 23.4}$$

### 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:

$$\text{is_root} \; h \longrightarrow T_{\text{insert}} a \; h + \Phi (\text{insert a} \; h) - \Phi h \leq \lg (|h| + 1) + 1$$
$$\text{is_root} \; h_1 \land \text{is_root} \; h_2 \longrightarrow T_{\text{merge}} \; h_1 \; h_2 + \Phi (\text{merge} \; h_1 \; h_2) - \Phi h_1 - \Phi h_2
\leq \lg (|h_1| + |h_2| + 1) + 2$$

They follow from the corresponding Lemmas 23.1 and 23.2.

Combining this inductive upper bound for the running time of the two passes

$$T_{\text{pass_2}} (\text{pass_1} \; hs_1) + T_{\text{pass_2}} \; hs_1 \leq \text{len} \; hs_1 + 2$$

with Theorem 23.3 yields the third and final amortized running time:

$$\text{is_root} \; h \longrightarrow T_{\text{del_min}} \; h + \Phi (\text{del_min} \; h) - \Phi h \leq 2 \cdot \lg (|h| + 1) + 5$$
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 [47] 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
The following functions on lists are predefined:

\[ \text{length :: } 'a \text{ list } \Rightarrow \text{ nat} \]
\[ |[]| = 0 \]
\[ |x # xs| = |xs| + 1 \]

\[ \text{( @ ) :: } 'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list} \]
\[ [] @ ys = ys \]
\[ (x # xs) @ ys = x # xs @ ys \]

\[ \text{set :: } 'a \text{ list } \Rightarrow 'a \text{ set} \]
\[ \text{set } [] = \{\} \]
\[ \text{set } (x # xs) = \{x\} \cup \text{set } xs \]

\[ \text{map :: } ('a \Rightarrow 'b) \Rightarrow 'a \text{ list } \Rightarrow 'b \text{ list} \]
\[ \text{map } f [] = [] \]
\[ \text{map } f (x # xs) = f x # \text{map } f xs \]

\[ \text{filter :: } ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list} \]
\[ \text{filter } p [] = [] \]
\[ \text{filter } p (x # xs) = (\text{if } p x \text{ then } x # \text{filter } p xs \text{ else } \text{filter } p xs) \]

\[ \text{concat :: } 'a \text{ list list } \Rightarrow 'a \text{ list} \]
\[ \text{concat } [] = [] \]
\[ \text{concat } (x # xs) = x @ \text{concat } xs \]
take :: nat ⇒ 'a list ⇒ 'a list

take _ [] = []
take n (x # xs) = (case n of 0 ⇒ [] | m + 1 ⇒ x # take m 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
Time Functions

B.1 Lists

\[
\begin{align*}
T_{\text{length}} &\colon \text{`a list } \Rightarrow \text{ nat} \\
T_{\text{length}} [] & = 1 \\
T_{\text{length}} (_\#xs) & = T_{\text{length}} xs + 1 \\
T_{\text{map}} &\colon (\text{`a } \Rightarrow \text{ nat}) \Rightarrow \text{`a list } \Rightarrow \text{ nat} \\
T_{\text{map}} [] & = 1 \\
T_{\text{map}} T_f (x \#xs) & = T_f x + T_{\text{map}} T_f xs + 1 \\
T_{\text{filter}} &\colon (\text{`a } \Rightarrow \text{ nat}) \Rightarrow \text{`a list } \Rightarrow \text{ nat} \\
T_{\text{filter}} [] & = 1 \\
T_{\text{filter}} T_p (x \#xs) & = T_p x + T_{\text{filter}} T_p xs + 1 \\
T_{\text{take}} &\colon \text{ nat } \Rightarrow \text{`a list } \Rightarrow \text{ nat} \\
T_{\text{take}} [] & = 1 \\
T_{\text{take}} n (_\#xs) & = \text{(case n of } 0 \Rightarrow 1 | n' + 1 \Rightarrow T_{\text{take}} n' xs + 1) \\
T_{\text{drop}} &\colon \text{ nat } \Rightarrow \text{`a list } \Rightarrow \text{ nat} \\
T_{\text{drop}} [] & = 1 \\
T_{\text{drop}} n (_\#xs) & = \text{(case n of } 0 \Rightarrow 1 | n' + 1 \Rightarrow T_{\text{drop}} n' xs + 1) \\
\end{align*}
\]
B.2 Selection

\( T_{\text{chop}} :: \text{nat} \Rightarrow \text{'a list} \Rightarrow \text{nat} \)

\( T_{\text{chop}} 0 = 1 \)

\( T_{\text{chop}} \_ \[] = 1 \)

\( T_{\text{chop}} n \; \text{xs} = T_{\text{take}} n \; \text{xs} + T_{\text{drop}} n \; \text{xs} + T_{\text{chop}} n (\text{drop} \; n \; \text{xs}) + 1 \)

\( T_{\text{partition}} 3 :: \text{'a} \Rightarrow \text{'a list} \Rightarrow \text{nat} \)

\( T_{\text{partition}} 3 \_ \[] = 1 \)

\( T_{\text{partition}} 3 x (\_ \; \# \; \text{ys}) = T_{\text{partition}} 3 \; x \; \text{ys} + 1 \)

\( T_{\text{slow_select}} :: \text{nat} \Rightarrow \text{'a list} \Rightarrow \text{nat} \)

\( T_{\text{slow_select}} k \; \text{xs} = T_{\text{isort}} \; \text{xs} + T_{\text{nth}} (\text{isort} \; \text{xs}) \; k + 1 \)

\( T_{\text{slow_median}} :: \text{'a list} \Rightarrow \text{nat} \)

\( T_{\text{slow_median}} \; \text{xs} = T_{\text{slow_select}} ((|\text{xs}| - 1) \; \text{div} \; 2) \; \text{xs} + 1 \)

\( T_{\text{chop}} \; d \; \text{xs} \leq 5 \cdot |\text{xs}| + 1 \)

\( T_{\text{partition}} 3 \; x \; \text{xs} = |\text{xs}| + 1 \)

\( T_{\text{slow_select}} k \; \text{xs} \leq |\text{xs}|^2 + 3 \cdot |\text{xs}| + 3 \)

\( T_{\text{slow_median}} \; \text{xs} \leq |\text{xs}|^2 + 3 \cdot |\text{xs}| + 4 \)

B.3 2-3 Trees

\( T_{\text{join_adj}} :: \text{'a tree23s} \Rightarrow \text{nat} \)

\( T_{\text{join_adj}} (\text{TTs} \_ \_ (T \_ \_)) = 1 \)

\( T_{\text{join_adj}} (\text{TTs} \_ \_ (\text{TTs} \_ \_ (T \_ \_))) = 1 \)

\( T_{\text{join_adj}} (\text{TTs} \_ \_ (\text{TTs} \_ \_ \text{ts})) = T_{\text{join_adj}} \; \text{ts} + 1 \)

\( T_{\text{join_all}} :: \text{'a tree23s} \Rightarrow \text{nat} \)

\( T_{\text{join_all}} (T \_ \_) = 1 \)

\( T_{\text{join_all}} \; \text{ts} = T_{\text{join_adj}} \; \text{ts} + T_{\text{join_all}} (\text{join_adj} \; \text{ts}) + 1 \)

\( T_{\text{tree23_of_list}} :: \text{'a list} \Rightarrow \text{nat} \)

\( T_{\text{tree23_of_list}} \; \text{as} = T_{\text{leaves}} \; \text{as} + T_{\text{join_all}} (\text{leaves} \; \text{as}) + 1 \)
B.4 Leftist Heaps

\[
T_{\text{merge}} :: (\forall a \rightarrow \text{nat}) \text{ tree} \Rightarrow (\forall a \rightarrow \text{nat}) \text{ tree} \Rightarrow \text{nat}
\]

\[
T_{\text{merge}} \langle \rangle \ t = 1
\]

\[
T_{\text{merge}} \ t \langle \rangle = 1
\]

\[
T_{\text{merge}} \ ((t_1, (a_1, n_1), r_1) =: t_1) ((t_2, (a_2, n_2), r_2) =: t_2) = (\text{if } a_1 \leq a_2 \text{ then } T_{\text{merge}} r_1 t_2 \text{ else } T_{\text{merge}} t_1 r_2) + 1
\]

\[
T_{\text{insert}} :: \forall a \rightarrow (\forall a \rightarrow \text{nat}) \text{ tree} \Rightarrow \text{nat}
\]

\[
T_{\text{insert}} \ x \ t = T_{\text{merge}} \langle \langle \rangle, (x, 1), \langle \rangle \rangle \ t + 1
\]

\[
T_{\text{del_min}} :: (\forall a \rightarrow \text{nat}) \text{ tree} \Rightarrow \text{nat}
\]

\[
T_{\text{del_min}} \langle \rangle = 1
\]

\[
T_{\text{del_min}} \langle l, \_, r \rangle = T_{\text{merge}} l \ r + 1
\]

B.5 Binomial Heaps

\[
T_{\text{link}} :: \forall a \rightarrow \text{tree} \Rightarrow \forall a \rightarrow \text{tree} \Rightarrow \text{nat}
\]

\[
T_{\text{link}} \ _ \ _ = 1
\]

\[
T_{\text{ins_tree}} :: \forall a \rightarrow \text{tree} \Rightarrow \forall a \rightarrow \text{tree list} \Rightarrow \text{nat}
\]

\[
T_{\text{ins_tree}} \ _ \ [] = 1
\]

\[
T_{\text{ins_tree}} \ t_1 \ (t_2 \neq \# \ ts) = (\text{if } \text{rank} \ t_1 < \text{rank} \ t_2 \text{ then } 1 \text{ else } T_{\text{link}} t_1 t_2 + T_{\text{ins_tree}} \ (\text{link} \ t_1 t_2) \ ts)
\]

\[
T_{\text{insert}} :: \forall a \rightarrow \forall a \rightarrow \text{tree list} \Rightarrow \text{nat}
\]

\[
T_{\text{insert}} \ x \ ts = T_{\text{ins_tree}} \ (\text{Node} \ 0 \ x \ []) \ ts + 1
\]

\[
T_{\text{merge}} :: \forall a \rightarrow \text{tree list} \Rightarrow \forall a \rightarrow \text{tree list} \Rightarrow \text{nat}
\]

\[
T_{\text{merge}} \ [] \ (\# \ _) = 1
\]

\[
T_{\text{merge}} \ (t_1 \ neq \ ts_1 \ =: \ h_1) \ (t_2 \ neq \ ts_2 \ =: \ h_2) = 1 + (\text{if } \text{rank} \ t_1 < \text{rank} \ t_2 \text{ then } T_{\text{merge}} \ ts_1 \ h_2)
\]
else if rank $t_2 < rank t_1$ then $T_{merge} h_1 t s_2$
else $T_{ins\_tree} (link t_1 t_2) (merge t s_1 t s_2) + T_{merge} t s_1 t s_2$

$T_{get\_min} :: 'a tree list ⇒ nat$
$T_{get\_min} [\_] = 1$
$T_{get\_min} (_ # v # va) = 1 + T_{get\_min} (v # va)$

$T_{get\_min\_rest} :: 'a tree list ⇒ nat$
$T_{get\_min\_rest} [\_] = 1$
$T_{get\_min\_rest} (_ # v # va) = 1 + T_{get\_min\_rest} (v # va)$

$T_{rev} :: 'a list ⇒ nat$
$T_{rev} xs = |xs| + 1$

$T_{del\_min} :: 'a tree list ⇒ nat$
$T_{del\_min} ts$
$= T_{get\_min\_rest} ts +$
  (case $get\_min\_rest$ ts of
    (Node _ _ ts1, ts2) ⇒ $T_{rev}$ ts1 + $T_{merge}$ (rev ts1) ts2) +
  1

B.6 Queues

$T_{norm} :: 'a list × 'a list ⇒ nat$
$T_{norm} (fs, rs) = (if fs = [] then $T_{itre}$ rs [] else 0) + 1$

$T_{enq} :: 'a ⇒ 'a list × 'a list ⇒ nat$
$T_{enq} a (fs, rs) = T_{norm} (fs, a # rs) + 1$

$T_{deq} :: 'a list × 'a list ⇒ nat$
$T_{deq} (fs, rs) = (if fs = [] then 0 else $T_{norm}$ (tl fs, rs)) + 1$

$T_{first} :: 'a list × 'a list ⇒ nat$
$T_{first} (_ # _ # _, _) = 1$
B.7 Splay Trees

\[ T_{\text{is\_empty}} :: \text{'a list} \times \text{'a list} \Rightarrow \text{nat} \]
\[ T_{\text{is\_empty}} (_{-}, _) = 1 \]

\[ T_{\text{is\_empty}} :: \text{'a list} \times \text{'a list} \Rightarrow \text{nat} \]
\[ T_{\text{is\_empty}} (_{-}, _) = 1 \]

B.7 Splay Trees

\[ T_{\text{splay}} :: \text{'a} \Rightarrow \text{'a tree} \Rightarrow \text{nat} \]
\[ T_{\text{splay}} \_ \langle \rangle = 1 \]
\[ T_{\text{splay}} \_ \langle AB, b, CD \rangle = (\text{case } \text{cmp} x b \text{ of} \]
\[ LT \Rightarrow \text{case } AB \text{ of} \]
\[ \langle \rangle \Rightarrow 1 | \]
\[ \langle A, a, B \rangle \Rightarrow \text{case } \text{cmp} x a \text{ of} \]
\[ LT \Rightarrow \text{if } A = \langle \rangle \text{ then } 1 \text{ else } T_{\text{splay}} x A + 1 | \]
\[ EQ \Rightarrow 1 | \]
\[ GT \Rightarrow \text{if } B = \langle \rangle \text{ then } 1 \text{ else } T_{\text{splay}} x B + 1 | \]
\[ EQ \Rightarrow 1 | \]
\[ GT \Rightarrow \text{case } CD \text{ of} \]
\[ \langle \rangle \Rightarrow 1 | \]
\[ \langle C, c, D \rangle \Rightarrow \text{case } \text{cmp} x c \text{ of} \]
\[ LT \Rightarrow \text{if } C = \langle \rangle \text{ then } 1 \text{ else } T_{\text{splay}} x C + 1 | \]
\[ EQ \Rightarrow 1 | \]
\[ GT \Rightarrow \text{if } D = \langle \rangle \text{ then } 1 \text{ else } T_{\text{splay}} x D + 1 \]

\[ T_{\text{splay\_max}} :: \text{'a tree} \Rightarrow \text{nat} \]
\[ T_{\text{splay\_max}} \langle \rangle = 1 \]
\[ T_{\text{splay\_max}} _{-} _{-} _{-} = 1 \]
\[ T_{\text{splay\_max}} _{-} _{-} _{-} _{-} = (\text{if } C = \langle \rangle \text{ then } 1 \text{ else } T_{\text{is\_empty}} C + 1) \]

\[ T_{\text{insert}} :: \text{'a} \Rightarrow \text{'a tree} \Rightarrow \text{nat} \]
\[ T_{\text{insert}} x t = 1 + (\text{if } t = \langle \rangle \text{ then } 0 \text{ else } T_{\text{splay}} x t) \]

\[ T_{\text{delete}} :: \text{'a} \Rightarrow \text{'a tree} \Rightarrow \text{nat} \]
\[ T_{\text{delete}} x t \]
\[ = 1 + \]
\[ (\text{if } t = \langle \rangle \text{ then } 0 \text{ else } T_{\text{is\_empty}} t) \]
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

$T_{merge} :: 'a \text{ tree } \Rightarrow 'a \text{ tree } \Rightarrow \text{ nat}$
$T_{merge} \langle \rangle _ _ = 1$
$T_{merge} _ \langle \rangle = 1$
$T_{merge} \langle l_1, a_1, r_1 \rangle \langle l_2, a_2, r_2 \rangle$
   = (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

$T_{insert} :: 'a \Rightarrow 'a \text{ tree } \Rightarrow \text{ int}$
$T_{insert} a t = T_{merge} \langle \langle \rangle, a, \langle \rangle \rangle t + 1$

$T_{del\_min} :: 'a \text{ tree } \Rightarrow \text{ int}$
$T_{del\_min} t = (\text{case } t \text{ of } \langle \rangle \Rightarrow 1 | \langle t_1, _, t_2 \rangle \Rightarrow T_{merge} t_1 t_2 + 1)$

B.9 Pairing Heaps

$T_{insert} :: 'a \Rightarrow 'a \text{ tree } \Rightarrow \text{ nat}$
$T_{insert} _ _ = 1$

$T_{merge} :: 'a \text{ tree } \Rightarrow 'a \text{ tree } \Rightarrow \text{ nat}$
$T_{merge} _ _ = 1$

$T_{del\_min} :: 'a \text{ tree } \Rightarrow \text{ nat}$
$T_{del\_min} \langle \rangle = 1$
$T_{del\_min} \langle hs, _, \_ \rangle = T_{pass2} (pass_1 hs) + T_{pass1} hs + 1$

$T_{pass1} :: 'a \text{ tree } \Rightarrow \text{ nat}$
\[ T_{\text{pass}} \langle \_, \_, \langle \_, \_, hs' \rangle \rangle = T_{\text{pass}} hs' + 1 \]
\[ T_{\text{pass}} \langle \rangle = 1 \]
\[ T_{\text{pass}} \langle \_, \_, \rangle \rangle = 1 \]

\[ T_{\text{pass}} :: \ 'a \ \text{tree} \Rightarrow \text{nat} \]
\[ T_{\text{pass}} \langle \rangle = 1 \]
\[ T_{\text{pass}} \langle \_, \_, hs \rangle = T_{\text{pass}} hs + 1 \]
## 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>ASCII abbrev.</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lambda</td>
<td>\langle \text{lambda} \rangle</td>
<td>%</td>
</tr>
<tr>
<td>\equiv</td>
<td>\langle \text{equiv} \rangle</td>
<td>==</td>
</tr>
<tr>
<td>\noteq</td>
<td>\langle \text{noteq} \rangle</td>
<td>-=</td>
</tr>
<tr>
<td>\land</td>
<td>\langle \text{And} \rangle</td>
<td>!!</td>
</tr>
<tr>
<td>\forall</td>
<td>\langle \text{forall} \rangle</td>
<td>!</td>
</tr>
<tr>
<td>\exists</td>
<td>\langle \text{exists} \rangle</td>
<td>?</td>
</tr>
<tr>
<td>\implies</td>
<td>\langle \text{Longrightarrow} \rangle</td>
<td>==&gt;</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>\langle \text{Longrightarrow} \rangle</td>
<td>-&gt;</td>
</tr>
<tr>
<td>\iff</td>
<td>\langle \text{Longlefttrightarrow} \rangle</td>
<td>&lt;-&gt; or --&gt;</td>
</tr>
<tr>
<td>\Rightarrow</td>
<td>\langle \text{Rightarrow} \rangle</td>
<td>=&gt;</td>
</tr>
<tr>
<td>\leftarrow</td>
<td>\langle \text{Leftarrow} \rangle</td>
<td>&lt;-</td>
</tr>
<tr>
<td>\not</td>
<td>\langle \text{not} \rangle</td>
<td>-</td>
</tr>
<tr>
<td>\land</td>
<td>\langle \text{and} \rangle</td>
<td>&amp;</td>
</tr>
<tr>
<td>\lor</td>
<td>\langle \text{or} \rangle</td>
<td>\lor</td>
</tr>
<tr>
<td>\in</td>
<td>\langle \text{in} \rangle</td>
<td>:</td>
</tr>
<tr>
<td>\notin</td>
<td>\langle \text{notin} \rangle</td>
<td>-:</td>
</tr>
<tr>
<td>\cup</td>
<td>\langle \text{union} \rangle</td>
<td>\cup</td>
</tr>
</tbody>
</table>
### C.2 Subscripts and Superscripts

In addition to this, subscripts and superscripts with a single symbol can be rendered using two special symbols, `<sup>` and `<sub>`. The term $x_0$ for instance can be input as $x_0$. 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 $x_\ldots$ and $\ldots x$ by Isabelle/jEdit.

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.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 $\sum x \in A. \ f x$, this is just the $\eta$-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

<table>
<thead>
<tr>
<th>Book notation</th>
<th>Isabelle notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot y$</td>
<td>$x \cdot y$</td>
<td>times $x \ y$</td>
</tr>
<tr>
<td>$x / y$ or $x \div y$</td>
<td>$x / y$</td>
<td>divide $x \ y$ (for type real)</td>
</tr>
<tr>
<td>$x \div y$</td>
<td>$x \div y$</td>
<td>divide $x \ y$ (for type nat or int)</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
<td>$</td>
</tr>
<tr>
<td>$[x]$</td>
<td>$[x]$</td>
<td>floor $x$</td>
</tr>
<tr>
<td>$[x]$</td>
<td>$[x]$</td>
<td>ceiling $x$</td>
</tr>
<tr>
<td>$x^n$</td>
<td>$x^n$</td>
<td>power $x \ n$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lists</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$[xs]$</td>
<td>$[xs]$</td>
<td>length $xs$</td>
</tr>
<tr>
<td>$\ []$</td>
<td>$\ []$</td>
<td>Nil</td>
</tr>
<tr>
<td>$x # xs$</td>
<td>$x # xs$</td>
<td>Cons $x \ xs$</td>
</tr>
<tr>
<td>$[x, y]$</td>
<td>$[x, y]$</td>
<td>$x # y # \ []$</td>
</tr>
<tr>
<td>$[m..&lt;n]$</td>
<td>$[m..&lt;n]$</td>
<td>upt $m \ n$</td>
</tr>
<tr>
<td>$xs ! n$</td>
<td>$xs ! n$</td>
<td>nth $xs \ n$</td>
</tr>
<tr>
<td>$xs[n := y]$</td>
<td>$xs[n := y]$</td>
<td>list_update $xs \ n \ y$</td>
</tr>
<tr>
<td>Book notation</td>
<td>Isabelle notation</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>---------</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
<td>empty</td>
</tr>
<tr>
<td>{x, y}</td>
<td>{x, y}</td>
<td>insert x (insert y {})</td>
</tr>
<tr>
<td>x ∈ A</td>
<td>x ∈ A</td>
<td>Set.member x A</td>
</tr>
<tr>
<td>x ∉ A</td>
<td>x ∉ A</td>
<td>~(x ∈ A)</td>
</tr>
<tr>
<td>A ∪ B</td>
<td>A ∪ B</td>
<td>union A B</td>
</tr>
<tr>
<td>A ∩ B</td>
<td>A ∩ B</td>
<td>inter A B</td>
</tr>
<tr>
<td>A ⊆ B</td>
<td>A ⊆ B</td>
<td>subset_eq A B</td>
</tr>
<tr>
<td>A ⊂ B</td>
<td>A ⊂ B</td>
<td>subset A B</td>
</tr>
<tr>
<td>f ' A</td>
<td>f ' A</td>
<td>image f A</td>
</tr>
<tr>
<td>f − ' A</td>
<td>f − ' A</td>
<td>vimage f A</td>
</tr>
<tr>
<td>{x</td>
<td>P x}</td>
<td>{x. P x}</td>
</tr>
<tr>
<td>{x ∈ A</td>
<td>P x}</td>
<td>{x ∈ A. P x}</td>
</tr>
<tr>
<td>{f x y</td>
<td>P x y}</td>
<td>{f x y</td>
</tr>
<tr>
<td>\bigcup_{x∈A} f x</td>
<td>\bigcup x∈A. f x</td>
<td>\bigcup(f ' A)</td>
</tr>
<tr>
<td>∀ x∈A. P x</td>
<td>∀ x∈A. P x</td>
<td>Ball A P</td>
</tr>
<tr>
<td>∃ x∈A. P x</td>
<td>∃ x∈A. P x</td>
<td>Bex A P</td>
</tr>
</tbody>
</table>

**Sets**

**Multisets**

<table>
<thead>
<tr>
<th>M</th>
<th>size M</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{#}</td>
</tr>
<tr>
<td>{x} + M</td>
<td>{#}</td>
</tr>
<tr>
<td>{x, y}</td>
<td>{#x, y#}</td>
</tr>
<tr>
<td>x ∈# M</td>
<td>x ∈# M</td>
</tr>
<tr>
<td>x ∉# M</td>
<td>x ∉# M</td>
</tr>
<tr>
<td>{x ∈# M</td>
<td>P x}</td>
</tr>
<tr>
<td>{f x</td>
<td>x ∈# M}</td>
</tr>
<tr>
<td>∀ x∈#M. P x</td>
<td>∀ x∈#M. P x</td>
</tr>
<tr>
<td>∃ x∈#M. P x</td>
<td>∃ x∈#M. P x</td>
</tr>
<tr>
<td>M ⊆# M'</td>
<td>M ⊆# M'</td>
</tr>
</tbody>
</table>
### C.3 Syntactic Sugar

<table>
<thead>
<tr>
<th>Book notation</th>
<th>Isabelle notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum A$</td>
<td>$\sum A$</td>
<td>$\text{sum} (\lambda x. x) A$</td>
</tr>
<tr>
<td>$\sum_{x \in A} f(x)$</td>
<td>$\sum x \in A. f x$</td>
<td>$\text{sum} f A$</td>
</tr>
<tr>
<td>$\sum_{k=i..j} f k$</td>
<td>$\sum k = i..j. f k$</td>
<td>$\text{sum} f {i..j}$</td>
</tr>
<tr>
<td>$\sum M$</td>
<td>$\sum # M$</td>
<td>$\text{sum}_\text{mset} M$</td>
</tr>
<tr>
<td>$\sum_{x \in M} f(x)$</td>
<td>$\sum x \in # M. f x$</td>
<td>$\text{sum}<em>\text{mset} (\text{image}</em>\text{mset} f M)$</td>
</tr>
<tr>
<td>$\sum_{x \leftarrow xs} f x$</td>
<td>$\sum x \leftarrow xs. f x$</td>
<td>$\text{sum}_\text{list} (\text{map} f xs)$</td>
</tr>
</tbody>
</table>

(Analogous for products)

### Intervals (for ordered types)

<table>
<thead>
<tr>
<th>Interval notation</th>
<th>Isabelle notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>${x..}$</td>
<td>${x..}$</td>
<td>$\text{atLeast} x$</td>
</tr>
<tr>
<td>${..y}$</td>
<td>${..y}$</td>
<td>$\text{atMost} y$</td>
</tr>
<tr>
<td>${x..y}$</td>
<td>${x..y}$</td>
<td>$\text{atLeastAtMost} x y$</td>
</tr>
<tr>
<td>${x..&lt;y}$</td>
<td>${x..&lt;y}$</td>
<td>$\text{atLeastLessThan} x y$</td>
</tr>
<tr>
<td>${x&lt;y..}$</td>
<td>${x&lt;y..}$</td>
<td>$\text{greaterThanAtMost} x y$</td>
</tr>
<tr>
<td>${x&lt;y..&lt;}$</td>
<td>${x&lt;y..&lt;}$</td>
<td>$\text{greaterThanLessThan} x y$</td>
</tr>
</tbody>
</table>
References


60. Ocaml website. [https://ocaml.org](https://ocaml.org).


Index

<table>
<thead>
<tr>
<th>ANSI C keywords</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>50, 251</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>251</td>
</tr>
<tr>
<td></td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions and concepts</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction function</td>
<td>61, 76</td>
</tr>
<tr>
<td>almost complete tree</td>
<td>53</td>
</tr>
<tr>
<td>amortized running time</td>
<td>53</td>
</tr>
<tr>
<td>array</td>
<td>121</td>
</tr>
<tr>
<td>augmented tree</td>
<td>57</td>
</tr>
<tr>
<td>AVL tree</td>
<td>101</td>
</tr>
<tr>
<td>balance factor</td>
<td>108</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>81</td>
</tr>
<tr>
<td>Bellman-Ford algorithm</td>
<td>202</td>
</tr>
<tr>
<td>binary search trees</td>
<td>61</td>
</tr>
<tr>
<td>binary trie</td>
<td>145</td>
</tr>
<tr>
<td>binomial heap</td>
<td>179</td>
</tr>
<tr>
<td>black height</td>
<td>92</td>
</tr>
<tr>
<td>Braun tree</td>
<td>122, 123</td>
</tr>
<tr>
<td>butlast</td>
<td>252</td>
</tr>
<tr>
<td>child</td>
<td>49</td>
</tr>
<tr>
<td>cmp</td>
<td>62</td>
</tr>
<tr>
<td>coercion</td>
<td>9</td>
</tr>
<tr>
<td>complete</td>
<td>52</td>
</tr>
<tr>
<td>complete tree</td>
<td>51</td>
</tr>
<tr>
<td>computation induction</td>
<td>12</td>
</tr>
<tr>
<td>concat</td>
<td>251</td>
</tr>
<tr>
<td>deletion by joining</td>
<td>64</td>
</tr>
<tr>
<td>deletion by replacing</td>
<td>64</td>
</tr>
<tr>
<td>depth</td>
<td>49</td>
</tr>
<tr>
<td>difference lists</td>
<td>30</td>
</tr>
<tr>
<td>drop</td>
<td>252</td>
</tr>
<tr>
<td>dynamic programming</td>
<td>191</td>
</tr>
<tr>
<td>Fibonacci tree</td>
<td>107</td>
</tr>
<tr>
<td>filter</td>
<td>251</td>
</tr>
<tr>
<td>filter_mset</td>
<td>10</td>
</tr>
<tr>
<td>h</td>
<td>51</td>
</tr>
<tr>
<td>hd</td>
<td>252</td>
</tr>
<tr>
<td>head</td>
<td>8</td>
</tr>
<tr>
<td>heap</td>
<td>168</td>
</tr>
<tr>
<td>height</td>
<td>51</td>
</tr>
<tr>
<td>height-balanced tree</td>
<td>108</td>
</tr>
<tr>
<td>Huffman's algorithm</td>
<td>151</td>
</tr>
<tr>
<td>IH</td>
<td>12</td>
</tr>
<tr>
<td>image_mset</td>
<td>10</td>
</tr>
<tr>
<td>inclusion</td>
<td>9</td>
</tr>
<tr>
<td>inorder</td>
<td>50</td>
</tr>
</tbody>
</table>
insertion sort, 20
int, 9
interval tree, 69
invariant, 65, 76
  preservation, 65
join approach, 114
leftist heap, 171
leftist tree, 171
length, 251
level, 49
linear order, 19
linorder, 19
list, 8
list extensionality, 127
locales, 76
Map, 78
map, 251
map_tree, 50
median, 40
merge sort, 24
mh, 51
min_height, 51
min_list, 253
mset, 10
multiset, 10
multiset, 10

nat, 9

observer functions, 215
optimal binary search tree, 207
pairing heap, 241
Patricia trie, 147
pivot, 22
preorder, 50
Priority_Queue, 167
priority queue, 167
  mergeable, 167
Priority_Queue_Merge, 168
Queue, 217
queue, 217
  Hood-Melville, 219
quicksort, 22
real, 9
red-black tree, 91
replicate, 252
rev, 252
right-heavy, 238
root, 49
running time, 12
running time function, 12
  canonical, 14
runs, 29
selection, 35
Set, 76
set, 251
set_mset, 10
set_tree, 50
skew heap, 237
sorted, 19, 66
sorted_wrt, 253
sorting, 19
specification, 65
splay tree, 229
stability (sorting), 33
stable, 33
sum_list, 253

T, 12
tail, 8
take, 252
time function, see running time function
tl, 252
tree, 49
trie, 143
2-3 trees, 81
type class, 19
type of interest, 76
type variable, 8

unit, 9