Data Structures and Amortized Complexity in a Functional Setting

Part II: Case Studies

Berry Schoenmakers

Doctoral Dissertation from
Eindhoven University of Technology[1]

September 1992

[1]Current address: CWI, Kruislaan 413, 1098 SJ Amsterdam, Netherlands. E-mail: berry@cwi.nl

Chapter 7

Tricky representations of sets and arrays

As remarked at the end of Chapter 3, there are amortized analyses in which a potential function that is bounded from below by a constant cannot be used. In this chapter we provide a simple example of such a case (Section 7.4). Furthermore, we show in this chapter how arrays can be combined with a functional programming style. An important aspect of this combination is that the algebra of arrays (as defined in Example 1.7) is destructive, hence that arrays must be restricted to linear usage. This restriction causes no problems, but the implemented algebras will be destructive too, so these algebras must be restricted to linear usage as well.

The material in this chapter is inspired by the problem stated in Exercise 2.12 from [1]:

> Develop a technique to initialize an entry of a matrix to zero the first time it is accessed, thereby eliminating the $O((\#V)^2)$ time to initialize an adjacency matrix. [*Hint:* Maintain a pointer in each initialized entry to a back pointer on a stack. Each time an entry is accessed, verify that the contents are not random by making sure the pointer in that entry points to the active region on the stack and that the back pointer points to the entry.]

Although the practical significance of this "trick" for avoiding initialization of large arrays may be small, it has been applied many times throughout the computing literature ([19] and [14] are just two arbitrary papers that refer to Exercise 2.12). Apparently, Mehlhorn recognized that this trick deserves more than the status of an exercise, for he presents a solution in his book on data structures and algorithms [23, pp. 289–290]. In this chapter we present two more solutions and compare them with Mehlhorn's solution.

At first we interpreted Exercise 2.12 as a problem of set representation. Realizing that the adjacency matrix in the exercise represents the *set* of arcs of a directed graph—whose vertices are numbered consecutively—we considered the problem of implementing an algebra with data type $\{[0..N)\}$, for a fixed natural number N. Sets of this type can be generated from the empty set by means of operation $\oplus$, a restricted version of insertion (as defined in Section 1.1.4). An element can be deleted from such a set by means of operation $\ominus$. To solve the exercise we implement the following algebra of "bounded sets":

$$\mathsf{BS} = (\,\{[0..N)\} \mid \{\,\}, \#, \in, \oplus, \ominus\,),$$

such that each operation is supported in $O(1)$ time. Operation $\#$ has been included instead of $(=\{\,\})$ because it can be implemented at no extra cost in the implementation of BS in Section 7.1.

On second thoughts, however, another way to solve the exercise is to implement the following algebra of arrays efficiently:

$$\mathsf{AK} = (\,[0..N) \to T \mid K, \mathit{lookup}, \mathit{update}\,).$$

The difference with the algebra of arrays defined in Example 1.7 is that an arbitrary (but fixed) function $K \in [0..N) \to T$ is provided instead of relation ?. Function K serves as initial array value: for example, $K.i$ = false is often an appropriate initial value when T = Bool. The problem is to support each operation of AK in constant time. In Section 7.2 we show how this can be done.

7.1 A tricky representation of sets

The problem is to implement BS by a concrete algebra with signature

$$\mathsf{C} = (\,C \mid \mathsf{empty}, \mathsf{size}, \mathsf{member}, \mathsf{insert}, \mathsf{delete}\,),$$

say, such that all operations of C have $O(1)$ time complexity. Although algebra BS is bijective—which enables us to use abstraction functions to describe its refinements—we will use a coupling $\simeq$, say, because this shortens the descriptions somewhat.

As stepping-stone towards a solution for C, we first present two well-known refinements of BS. We do not provide correctness proofs for the operations; this amounts to instantiation of Definition 2.12, where it should be noted that $\oplus$ and $\ominus$ are partial functions: $S \oplus i$ and $S \ominus i$ are defined only if $i \notin S$ and $i \in S$, respectively.

Bit-vector

A standard solution is to represent a subset of $[0..N)$ by an array of type B, where

$$B = [0..N) \to \text{Bool}.$$

In order to obtain an $O(1)$ program for operation size, we take $C = B \times [0..N]$, and define the coupling relation by

$$S \simeq \langle b, n \rangle \;\equiv\; S = \{i \mid b[i]\} \;\wedge\; \#S = n.$$

The operations are implemented as follows:

$$\begin{array}{lcl}
\mathsf{empty} & = & \langle g.?.0, 0 \rangle \\
 & & |[\ g.b.i = (g.b[i{:=}\mathrm{false}].(i{+}1)\ , i \neq N \ \ [] \ \ b\ , i = N)\ ]| \\
\mathsf{size}.\langle b, n \rangle & = & n \\
\mathsf{member}.i.\langle b, n \rangle & = & b[i] \\
\mathsf{insert}.\langle b, n \rangle.i & = & \langle b[i{:=}\mathrm{true}], n{+}1 \rangle \\
\mathsf{delete}.\langle b, n \rangle.i & = & \langle b[i{:=}\mathrm{false}], n{-}1 \rangle.
\end{array}$$

The only problem with this refinement is that operation empty takes $O(N)$ time.

Enumeration

Another well-known way to represent a set is by enumeration of its elements. For this purpose lists are adequate, but we use an array type instead to prepare for the next refinement. We introduce array type

$$E = [0..N) \to [0..N),$$

and we take $C = E \times [0..N]$. The coupling relation is defined by

$$S \simeq \langle e, n \rangle \;\equiv\; S = \{e[i] \mid 0 \leq i < n\} \;\wedge\; \#S = n,$$

and the operations are defined as

$$\begin{array}{lcl}
\mathsf{empty} & = & \langle ?, 0 \rangle \\
\mathsf{size}.\langle e, n \rangle & = & n \\
\mathsf{member}.i.\langle e, n \rangle & = & g.0 \neq n \\
 & & |[\ g.j \ = \ (j\ , i = e[j] \ \ [] \ \ g.(j{+}1)\ , i \neq e[j]) \quad , j \neq n \\
 & & \qquad\qquad [] \ \ n \qquad\qquad\qquad\qquad\qquad\qquad , j = n \\
 & &]| \\
\mathsf{insert}.\langle e, n \rangle.i & = & \langle e[n{:=}i], n{+}1 \rangle \\
\mathsf{delete}.\langle e, n \rangle.i & = & \langle e[g.0{:=}e[n{-}1]], n{-}1 \rangle \\
 & & |[\ g.j = (j\ , i = e[j] \ \ [] \ \ g.(j{+}1)\ , i \neq e[j])\ ]|.
\end{array}$$

This time, we see that empty takes $O(1)$ time, but member and delete take $O(N)$ time in the worst case.

Tricky representation

From the previous refinement, we now construct an efficient solution for C. We first concentrate on an $O(1)$ program for member. In the previous refinement we have that member.$i.\langle e, n \rangle$ is equivalent to $(\exists\, j : 0 \leq j < n : i = e[j])$. To determine the value of this predicate in constant time, we observe that segment $e[0..n)$ represents an injective function from $[0..n)$ to $[0..N)$. So, we can introduce an additional array f, say, of type E as "inverse" of $e[0..n)$ such that predicate $(\exists\, j : 0 \leq j < n : i = e[j])$ is equivalent to $0 \leq f[i] < n \ \wedge\ i = e[f[i]]$; this may be paraphrased as "$f[i]$ equals the location of i in $e[0..n)$ (if present)".

Thus, we take $C = E \times [0..N] \times E$, and define the coupling relation by

$$\begin{aligned} S \simeq \langle e, n, f\rangle \quad \equiv \quad & S = \{e[i] \mid 0 \leq i < n\} \ \wedge\ \#S = n \\ & \wedge\ (\forall\, i : 0 \leq i < n : f[e[i]] = i). \end{aligned}$$

In this definition the conjunct $\#S = n$ is actually redundant, but its presence reminds us that this coupling is an extension of the previous one. To prove that $i \in S$ is indeed equivalent to $0 \leq f[i] < n \ \wedge\ i = e[f[i]]$ when $S \simeq \langle e, n, f\rangle$, we observe:

$$\begin{array}{ll} & i \in S \\ \equiv & \{\ \text{definition of } S\ \} \\ & (\exists\, j : 0 \leq j < n : i = e[j]) \\ \equiv & \{\ \text{condition } (\forall\, i : 0 \leq i < n : f[e[i]] = i)\ \} \\ & (\exists\, j : 0 \leq j < n : i = e[j] \ \wedge\ f[e[j]] = j) \\ \equiv & \{\ f \text{ is a function}\ \} \\ & (\exists\, j : f[i] = j : 0 \leq j < n \ \wedge\ i = e[j]) \\ \equiv & \{\ \text{one-point rule}\ \} \\ & 0 \leq f[i] < n \ \wedge\ i = e[f[i]]. \end{array}$$

The operations of BS are now refined by the following $O(1)$ programs, which should be compared with those in the previous refinement:

$$\begin{array}{lcl} \mathsf{empty} & = & \langle ?, 0, ?\rangle \\ \mathsf{size}.\langle e, n, f\rangle & = & n \\ \mathsf{member}.i.\langle e, n, f\rangle & = & 0 \leq f[i] < n \ \wedge\ i = e[f[i]] \\ \mathsf{insert}.\langle e, n, f\rangle.i & = & \langle e[n{:=}i], n{+}1, f[i{:=}n]\rangle \\ \mathsf{delete}.\langle e, n, f\rangle.i & = & \langle e[f[i]{:=}e[n{-}1]], n{-}1, f[e[n{-}1]{:=}f[i]]\rangle. \end{array}$$

In the program for member we have exploited the fact that f is of type E: this means that $0 \leq f[i] < N$, hence $f[i]$ is in the domain of e.

The tricky thing about this implementation is that member always returns the desired value, despite the fact that evaluation of $\mathsf{member}.i.\langle e, n, f\rangle$ may give rise to inspection of "uninitialized" elements of f (and e).

7.2 A tricky representation of arrays

Starting from the tricky representation of BS, we now design a similar implementation of AK, for which we use as signature:

$$\mathsf{D} = (\,D \mid \mathsf{K}, \mathsf{lookup}, \mathsf{update}\,).$$

The idea is to represent an array of type $[0..N) \to T$ by a quadruple $\langle c, e, m, f\rangle$ in which c is the actual representation of the array and $\langle e, m, f\rangle$ represents the set of "initialized" elements of c, as in the tricky representation of sets in the previous section. To implement this idea we define data type D as

$$D = ([0..N) \to T) \times E \times [0..N] \times E,$$

and coupling $\simeq$ as

$$\begin{array}{lcl} a \simeq \langle c, e, m, f\rangle & \equiv & (\forall\, i : i \in S : a.i = c[i]) \ \wedge\ (\forall\, i : i \notin S : a.i = K.i) \\ & & \wedge\ \#S = m \ \wedge\ (\forall\, i : 0 \le i < m : f[e[i]] = i), \end{array}$$

where $S = \{e[i] \mid 0 \le i < m\}$. The following programs are now easily found (note that P is equivalent to $i \in S$):

$$\mathsf{K} = \langle ?, ?, 0, ?\rangle$$

$$\begin{array}{lcll} \mathsf{lookup}.\langle c, e, m, f\rangle.i & = & c[i] & ,P \\ & [] & K.i & ,\neg P \\ & & \multicolumn{2}{l}{|[\ P = 0 \le f[i] < m \ \wedge\ i = e[f[i]]\ ]|} \end{array}$$

$$\begin{array}{lcll} \mathsf{update}.\langle c, e, m, f\rangle.i.x & = & \langle c[i{:=}x], e, m, f\rangle & ,P \\ & [] & \langle c[i{:=}x], e[m{:=}i], m{+}1, f[i{:=}m]\rangle & ,\neg P \\ & & \multicolumn{2}{l}{|[\ P = 0 \le f[i] < m \ \wedge\ i = e[f[i]]\ ]|.} \end{array}$$

In order that the program for lookup be concrete, a concrete program for K should be available. Provided this program for K has $O(1)$ time complexity, all operations of D have $O(1)$ time complexity as well. For example, if $T = \mathrm{Nat}$, K could be defined as $K.i = i^2$. As for space utilization we remark that this implementation of arrays has an overhead of $O(N \log N)$ bits, and whether or not this overhead is negligible depends on T.

7.3 Mehlhorn's tricky representation of sets

On pages 289–290 in [23], we encountered a somewhat different solution than ours. On second reading of the hint in Exercise 2.12 we discovered that we did not follow it as closely as Mehlhorn did. Mehlhorn's solution is more of a mixture of the bit-vector refinement and enumeration. Indeed, his solution arises when we use algebra D to implement array b in the bit-vector refinement.

To describe his implementation of bounded sets (algebra BS), we take as concrete type $C = B \times [0..N] \times E \times [0..N] \times E$ and as coupling relation:

$$\begin{array}{lcl} S \simeq \langle b, n, e, m, f\rangle & \equiv & S = \{i \mid b[i] \ \wedge\ 0 \le f[i] < m \ \wedge\ i = e[f[i]]\} \\ & & \wedge\ \#S = n \ \wedge\ \#\{e[i] \mid 0 \le i < m\} = m. \end{array}$$

The operations are implemented as follows:

$$
\begin{array}{lcll}
\mathsf{empty} & = & \langle ?, 0, ?, 0, ? \rangle & \\
\mathsf{size}.\langle b, n, e, m, f \rangle & = & n & \\
\mathsf{member}.i.\langle b, n, e, m, f \rangle & = & b[i] \ \wedge \ 0 \le f[i] < m \ \wedge \ i = e[f[i]] & \\
\mathsf{insert}.\langle b, n, e, m, f \rangle.i & = & \langle b[i{:=}\mathrm{true}], n{+}1, e[m{:=}i], m{+}1, f[i{:=}m] \rangle & , \neg P \\
 & [] & \langle b[i{:=}\mathrm{true}], n{+}1, e, m, f \rangle & , P \\
 & & \quad |[\ P = 0 \le f[i] < m \ \wedge \ i = e[f[i]] \ ]| & \\
\mathsf{delete}.\langle b, n, e, m, f \rangle.i & = & \langle b[i{:=}\mathrm{false}], n{-}1, e, m, f \rangle . &
\end{array}
$$

Note that m is never decreased in these programs and that $m{=}N$ implies P. Hence, when m has reached its maximal value N, Mehlhorn's solution reduces to the bit-vector implementation. Unfortunately, this is of no use because refinement C is only required in case the total number of set operations is considerably less than $O(N)$ (see next section). Hence, in typical applications m will not reach value N.

7.4 An unboundedly small potential

Let us compare the bit-vector refinement with the tricky representations of sets by considering program $f \in [\,[0..N)\,] \to C$, which converts a list into (a representation of) a set:

$$
\begin{array}{lcll}
f.[\,] & = & \mathsf{empty} & \\
f.(i \vdash s) & = & \mathsf{insert}.c.i & , \neg \mathsf{member}.c.i \\
 & [] & c & , \mathsf{member}.c.i \\
 & & \quad |[\ c = f.s \ ]| . &
\end{array}
$$

Note that algebra C is used in a linear fashion, as required. Using a tricky representation of sets, computation of $f.s$ clearly takes $O(\#s)$ time. For the bit-vector refinement this computation takes $O(N + \#s)$ time, which is $O(\#s)$ when $\#s$ is $\Omega(N)$, and $O(N)$ otherwise. Phrased differently, when a sufficient number of set operations has been performed, the $O(N)$ cost of empty is negligible, but otherwise it is better to use a tricky representation of sets.

It is interesting to perform an amortized analysis of program f in case the bit-vector refinement is used. A realistic cost measure $\mathcal{T}$ for this implementation of bounded sets is defined by placing the dots as follows:

$$
\begin{array}{lcl}
\mathsf{empty} & = & \langle g.?.0, 0 \rangle \\
 & & |[\ g.b.i \doteq (g.b[i{:=}\mathrm{false}].(i{+}1) \ , i \ne N \ \ [] \ \ b \ , i = N) \ ]| \\
\mathsf{size}.\langle b, n \rangle & \doteq & n \\
\mathsf{member}.i.\langle b, n \rangle & \doteq & b[i] \\
\mathsf{insert}.\langle b, n \rangle.i & \doteq & \langle b[i{:=}\mathrm{true}], n{+}1 \rangle \\
\mathsf{delete}.\langle b, n \rangle.i & \doteq & \langle b[i{:=}\mathrm{false}], n{-}1 \rangle .
\end{array}
$$

Then $\mathcal{T}[\mathsf{empty}] = N + 1$. We introduce amortized costs for algebra C according to the general scheme of Section 5.5:

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{size}](c) & = & \mathcal{T}[\mathsf{size}](c) \\
\mathcal{A}[\mathsf{member}](i, c) & = & \mathcal{T}[\mathsf{member}](i, c) \\
\mathcal{A}[\mathsf{insert}](c, i) & = & \mathcal{T}[\mathsf{insert}](c, i) + \Phi.(\mathsf{insert}.c.i) - \Phi.c \\
\mathcal{A}[\mathsf{delete}](c, i) & = & \mathcal{T}[\mathsf{delete}](c, i) + \Phi.(\mathsf{delete}.c.i) - \Phi.c,
\end{array}$$

where $c \in B \times [0..N]$. Now, to obtain constant amortized costs for empty we have to choose Φ such that $N+1+\Phi.\mathsf{empty}$ is $O(1)$, and to obtain constant amortized costs for the remaining operations it suffices to take a constant function for Φ. Therefore, we take $\Phi.c = -N$, as a consequence of which the amortized costs of each operation of C is exactly one. To determine $\mathcal{T}[f]$, we now define

$$\mathcal{A}[f](s) = \mathcal{T}[f](s) + \Phi.(f.s).$$

Then $\mathcal{A}[f]([\,]) = 1$ and $\mathcal{A}[f](i \vdash s) \leq 2 + \mathcal{A}[f](s)$, and

$$\mathcal{T}[f](s) = \mathcal{A}[f](s) - \Phi.(f.s) \leq 1 + 2\#s + N.$$

This formally proves that the computation of $f.s$ costs $O(N + \#s)$.

So, in this analysis a negative potential function naturally arises. For fixed N this potential is however bounded from below by $-N$, which enables us to apply a trick to change this into a nonnegative potential: view empty as a transformation from $\{\perp\}$ to C, and define $\Phi.\perp = N$ and $\Phi.c = 0$ for $c \in C$. Furthermore, define

$$\mathcal{A}[\mathsf{empty}] = \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} - \Phi.\perp.$$

Then the potential is nonnegative and the amortized costs are all $O(1)$. (A drawback of this trick of defining $\Phi.\perp$ unequal to 0 is that the analysis of f must be adapted too: its amortized costs must be defined as $\mathcal{A}[f](s) = \mathcal{T}[f](s) + \Phi.(f.s) - \Phi.\perp$, which gives the same bound for $\mathcal{T}[f](s)$.)

By "promoting" N to a parameter of algebra BS, however, this trick does not work anymore. By this we mean that data type $\{[0..N)\}$ is changed into

$$\{(n, \{[0..n)\}) \mid n \in \mathrm{Nat}\}.$$

Then empty gets a natural number as parameter such that $\mathsf{empty}.n$ creates a representation for the empty set of type $\{[0..n)\}$. The appropriate potential is now $\Phi.(n, c) = -n$, and since n ranges over all natural numbers, this potential is not bounded from below. From this we conclude that the desire to achieve $O(1)$ amortized costs for this modified algebra, a negative potential is really needed.

7.5 Comparisons

Algebras BS and AK are strongly related in the sense that an efficient implementation for BS is easily designed given an efficient implementation for AK, and vice versa. Comparing the two tricky representations of sets, we see, on the one hand, that our programs for member and insert are simpler than Mehlhorn's programs, and, on the other hand, that his program for delete is simpler than ours. A possible drawback of Mehlhorn's solution is that it requires a boolean array.

In summary, the results for the respective refinements of BS are:

	bit-vector	enumeration	tricky representation	Mehlhorn's solution
$\{\,\}$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
$\#S$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$i \in S$	$O(1)$	$O(\#S)$	$O(1)$	$O(1)$
$S \oplus i$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$S \ominus i$	$O(1)$	$O(\#S)$	$O(1)$	$O(1)$
space (bits)	N	$N \log_2 N$	$2N \log_2 N$	$2N \log_2 N + N$

This table suggests that the second refinement is hardly useful. However, compared to the bit-vector refinement, enumeration has the advantage that the elements of set S can be enumerated in $O(\#S)$ time, while this takes $O(N)$ time in the bit-vector refinement, regardless of the size of the enumerated set S.

From the bottom line of this table, we conclude that the price to be paid for the $O(1)$ initialization of bounded sets is the increase in space utilization from $O(N)$ to $O(N \log N)$ bits. As explained in Section 7.4, the $O(N)$ initialization should be avoided only when the total number of applications of the other operations in the rest of the program is significantly smaller than $O(N)$, e.g. $O(\sqrt{N})$. An example of such a program is a simple recognition algorithm for so-called series-parallel graphs. The first part of this algorithm computes the adjacency matrix for the input graph, which normally takes $O((\#V)^2)$ time (cf. Exercise 2.12 at the beginning of this chapter). The number of accesses to this matrix in the second part of the algorithm is however linear in $\#V$. Therefore it is worth while to use the tricky representation for the arc set of the input graph, since it is given that the number of arcs is $O(\#V)$, thereby reducing the time complexity of the entire algorithm to $O(\#V)$.

In practice, however, the implicit constant in the $O(N)$ bound for the initialization of arrays is very small, and, consequently, a tricky representation of sets is not worth the trouble for small values of N.

Chapter 8

Maintaining the minimum of a list

In this chapter we show how inspection operation $\downarrow$ can be added to the list algebras from Chapter 4 in such a way that each operation still has constant—possibly amortized—time complexity. Using essentially the same method, we obtain a destructive solution for concatenable queues and purely-functional solutions for stacks, queues, and deques. An efficient implementation of concatenable deques is however beyond the scope of the method described in this chapter. We have found similar implementations of these list algebras in [10].

Throughout this chapter, T denotes a linearly ordered type. In Chapter 1, $\downarrow s$ has been defined for nonempty s only. Here we extend it with $\downarrow[\,] = \infty$, where ∞ is larger than any element of T. As announced in Example 2.26, we shall also discuss noninjective algebras which arise when inspections hd and lt are omitted from the above-mentioned list algebras. This give rise to simpler and more efficient implementations, as will be shown in Section 8.4.

8.1 Stacks

In this section we design an efficient purely-functional implementation for the algebra of *minstacks*:

$$\mathsf{MS} = (\,[T] \mid [\,], (=[\,]), \vdash, hd, tl, \downarrow\,).$$

Since the algebra of stacks is the basic list algebra for functional languages (cf. Chapter 4), this shows that inspection $\downarrow$ can be provided efficiently for any functional program.

Informally, the problem is to maintain the minimum of a list under the transformations $\vdash$ and tl. Since $\downarrow(a \vdash s) = a \min \downarrow s$, this can easily be done for $\vdash$. However, $\downarrow(tl.s)$ cannot be determined from $\downarrow s$ in each case: if $hd.s = \downarrow s$, there is no relation between the minima of $tl.s$ and s. A moment's reflection leads to the conclusion that besides the minimum of the stack also the minimum of the tail part after the minimum must be maintained. Repeating this observation leads to the following definitions.

As concrete data type we use a subset of type $[[T] \times T]$. Lists of this type represent lists in $[T]$ according to abstraction $[\![\cdot]\!]$:

$$\begin{array}{lcl} [\![\,[\,]\,]\!] & = & [\,] \\ [\![(s,a) \vdash x]\!] & = & s +\!\!+ [a] +\!\!+ [\![x]\!]. \end{array}$$

For a list $x = [(s_0, a_0), (s_1, a_1), \ldots]$ of this type, we call x the *outer list* and the s_i's *inner lists.* List x is a kind of "list partition" of $[\![x]\!]$.

To expedite the retrieval of the minimum value, these partitions are required to satisfy condition P defined by

$$\begin{array}{lcl} P.[\,] & \equiv & \text{true} \\ P.((s,a) \vdash x) & \equiv & a \leq \downarrow s \ \wedge \ a < \downarrow[\![x]\!] \ \wedge \ P.x. \end{array}$$

The concrete type thus consists of those lists in $[[T] \times T]$ satisfying P. The corresponding programs of $O(1)$ time complexity are:

$$\begin{array}{lcll} \mathsf{empty} & = & [\,] & \\ \mathsf{isempty}.x & = & x = [\,] & \\ \mathsf{cons}.b.x & = & ([\,], b) \vdash x & , b < \mathsf{min}.x \\ \mathsf{cons}.b.((s,a) \vdash x) & = & (b \vdash s, a) \vdash x & , b \geq a \\ \mathsf{hd}.((s,a) \vdash x) & = & a & , s = [\,] \\ & [\,] & hd.s & , s \neq [\,] \\ \mathsf{tl}.((s,a) \vdash x) & = & x & , s = [\,] \\ & [\,] & (tl.s, a) \vdash x & , s \neq [\,] \\ \mathsf{min}.[\,] & = & \infty & \\ \mathsf{min}.((s,a) \vdash x) & = & a. & \end{array}$$

Since the outer list as well as the inner lists are used as stacks, these programs are purely-functional. In the worst case (when all inner lists are empty), about $2\#x$ pointers are required for the representation of $[\![x]\!]$ (instead of $\#x$ for an ordinary stack, cf. Section 4.3).

Note that each list of type $[T]$ can be partioned in only one way, as we have the following representation function, which maps $[T]$ onto $[[T] \times T]$:

$$\begin{array}{lcll} (\![\,[\,]\,]\!) & = & [\,] & \\ (\![s +\!\!+ [a] +\!\!+ t]\!) & = & (s,a) \vdash (\![t]\!) & , a \leq \downarrow s \ \wedge \ a < \downarrow t\,. \end{array}$$

By writing $a \leq \downarrow s \ \wedge \ a < \downarrow t$ instead of $a < \downarrow s \ \wedge \ a \leq \downarrow t$, a is the rightmost occurrence of the minimum of $s +\!\!+ [a] +\!\!+ t$ instead of the leftmost occurrence; in this way, the length of $(\![s]\!)$ is minimized. Computation of $(\![s]\!)$ is easily done in linear time, using one application of empty and $\#s$ applications of cons.

Remark 8.1

The above partioning may be applied to the inner lists as well. Instead of $[[T] \times T]$ we then get the smallest solution of

$$X : X = [X \times T]$$

as concrete type. This tree type has already been introduced in Chapter 6, where it was called $\langle T \rangle_1$. The corresponding representation function now yields a binary tree:

$$\begin{array}{lcl} (\!|\,[\,]\,|\!) & = & [\,] \\ (\!| s +\!\!\!+ [a] +\!\!\!+ t |\!) & = & ((\!| s |\!), a) \vdash (\!| t |\!) \quad , a \leq \downarrow s \ \wedge \ a < \downarrow t \,. \end{array}$$

In terms of type $\langle T \rangle$, this a tree satisfying the following heap condition:

$$\begin{array}{lcl} H.\langle\,\rangle & \equiv & \text{true} \\ H.\langle t, a, u \rangle & \equiv & H.t \ \wedge \ a \leq \downarrow t \ \wedge \ a < \downarrow u \ \wedge \ H.u, \end{array}$$

and heap x represents list $\overline{x}$, the inorder traversal of x. These binary heaps are known as *Cartesian trees* [35] and have many applications (see e.g. [3, 9]).
□

8.2 Deques

In Section 5.6.1 we have implemented an algebra of deques using a pair of stacks. From the observations in the previous section we conclude that algebra DQ extended with operation $\downarrow$:

$$\mathsf{MDQ} = (\, [T] \mid [\,], (=[\,]), \vdash, hd, tl, \dashv, lt, ft, rev, \downarrow\,),$$

can thus be implemented by means of a pair of minstacks. The program for min is

$$\mathsf{min}.\langle s, t \rangle = \downarrow s \ \min \downarrow t \,,$$

and the programs for the other operations can simply be copied from Section 5.6.1. This yields an efficient purely-functional solution which uses at most two pointers per list element.

8.3 Concatenable queues

As far as we know there is no implementation for algebra MDQ extended with $+\!\!\!+$ such that all operations have $O(1)$ time complexity. An efficient implementation of *concatenable minqueues*, however, is possible:

$$\mathsf{CMQ} = (\, [T] \mid [\,], (=[\,]), [\cdot], +\!\!\!+, hd, tl, \downarrow\,).$$

As stepping-stone to an implementation of this algebra, we first give an alternative implementation of *minqueues*:

$$\mathsf{MQ} = (\, [T] \mid [\,], (=[\,]), \dashv, hd, tl, \downarrow\,).$$

This implementation is identical to the implementation of MS in Section 8.1, except that operation $\dashv$ is refined by

$$
\begin{array}{lll}
\mathsf{snoc}.x.b & = & f.x.([\,],b) \\
& & |[\ f.[\,].(t,b) \quad = \quad [(t,b)] \\
& & \quad f.(x \dashv (s,a)).(t,b) \quad = \quad x \dashv (s,a) \dashv (t,b) \qquad ,a < b \\
& & \qquad\qquad\qquad\qquad\quad\ \ \square \quad f.x.(s +\!\!\!+ [a] +\!\!\!+ t, b) \quad ,a \geq b \\
& &]|.
\end{array}
$$

In these programs the inner lists are used as concatenable queues. Since these lists are used in a linear fashion, the destructive implementation of concatenable queues described in Section 4.5 can be used. The outer list is used as a deque. Taking $\Phi.x = \#x$ as potential, the amortized costs of snoc (and the other operations) are all $O(1)$.

As implementation for $+\!\!\!+$ we have

$$
\begin{array}{llll}
\mathsf{cat}.x.y = g.x.y & & & \\
\quad |[\ g.[\,].y & = & y & \\
\quad\ \ g.x.[\,] & = & x & \\
\quad\ \ g.(x \dashv (s,a)).((t,b) \vdash y) & = & (x \dashv (s,a)) +\!\!\!+ ((t,b) \vdash y) & ,a < b \\
& \square & g.x.((s +\!\!\!+ [a] +\!\!\!+ t, b) \vdash y) & ,a \geq b \\
\quad]|. & & &
\end{array}
$$

With the same potential as above, this program also has $O(1)$ amortized cost. Note that the outer list is now used as a concatenable deque.

8.4 Noninjective algebras

There are applications of minqueues in which operation hd is not required. In those cases it suffices to implement the following algebra:

$$\mathsf{MQ}' = (\,[T] \mid [\,], (=[\,]), \dashv, tl, \downarrow\,).$$

As explained in Example 2.26, this algebra is not injective because lists $[1,0]$ and $[2,0]$, for example, cannot be distinguished. To describe an efficient implementation of MQ' we use a representation function $(\!|\cdot|\!)$ that maps $[T]$ onto concrete type $[\mathrm{Nat} \times T]$:

$$
\begin{array}{lll}
(\!|\,[\,]\,|\!) & = & [\,] \\
(\!| s +\!\!\!+ [a] +\!\!\!+ t |\!) & = & (\#s, a) \vdash (\!| t |\!) \quad , a \leq {\downarrow} s \ \wedge\ a < {\downarrow} t\,.
\end{array}
$$

Hence, instead of a list x of type $[[T] \times T]$, it now suffices to record the *length* of the inner lists.

Accordingly, the programs are obtained by replacing the inner lists in the programs for minqueues by their length:

$$\mathsf{empty} \quad = \quad [\,]$$

$$
\begin{array}{lcl}
\mathsf{isempty}.x & = & x = [\,] \\
\mathsf{snoc}.x.b & = & f.x.(0,b) \\
 & & \begin{array}{l}|[\ \begin{array}{lcll} f.[\,].(n,b) & = & [(n,b)] & \\ f.(x \dashv (m,a)).(n,b) & = & x \dashv (m,a) \dashv (n,b) & ,a < b \\ & [] & f.x.(m{+}1{+}n,b) & ,a \geq b \end{array} \\ ]| \end{array} \\
\mathsf{tl}.((m,a) \vdash x) & = & x \qquad\qquad\qquad ,m = 0 \\
 & [] & (m{-}1,a) \vdash x \qquad ,m \neq 0 \\
\mathsf{min}.[\,] & = & \infty \\
\mathsf{min}.((m,a) \vdash x) & = & a.
\end{array}
$$

As the outer list is used as a deque, it follows that there exists a purely-functional implementation of MQ'.

Example 8.2
As an application of algebra MQ', we consider an abstract version of the pagination algorithm used in the pagination strategy MS in [22]. The problem is to determine, for fixed positive integer K, a pair of elements of a given sequence that are no more than K positions apart and that have minimal sum. More formally, the problem is to compute

$$(\mathrm{Min}\, i,j : 0 \leq i < j < N \ \wedge\ j{-}K \leq i : X[i] + X[j])$$

for given integer array X of positive length N.

Our solution relies on the fact that for $1 \leq n < N$:

$$
\begin{array}{ll}
 & (\mathrm{Min}\, i,j : 0 \leq i < j < n{+}1 \ \wedge\ j{-}K \leq i : X[i] + X[j]) \\
= & \quad \{ \text{ split off } j = n \ \} \\
 & (\mathrm{Min}\, i,j : 0 \leq i < j < n \ \wedge\ j{-}K \leq i : X[i] + X[j]) \\
 & \min((\mathrm{Min}\, i : 0 \leq i < n \ \wedge\ n{-}K \leq i : X[i]) + X[n]).
\end{array}
$$

To compute the minimum expressed on the bottom line we use a minqueue q satisfying $[\![q]\!] = [\, i : 0 \leq i < n \ \wedge\ n{-}K \leq i : X[i]\,]$. This yields the following $O(N)$ program for the original problem:

$$
\begin{array}{l}
\quad m, q, n := \infty,\ \mathsf{snoc.empty}.X[0],\ 1 \\
;\ \mathbf{do}\ n \neq N \ \rightarrow \\
\qquad\quad m := m \min(\mathsf{min}.q + X[n]) \\
\qquad ;\ q := \mathsf{snoc}.q.X[n] \\
\qquad ;\ \mathbf{if}\ n \geq K \ \rightarrow q := \mathsf{tl}.q \\
\qquad\quad\ \ []\ n < K \ \rightarrow \mathrm{skip} \\
\qquad\quad\ \ \mathbf{fi} \\
\qquad ;\ n := n{+}1 \\
\quad \mathbf{od}\ \ .
\end{array}
$$

Clearly, operation hd is not used in this application, and therefore the above described implementation of minqueues suffices.
□

The algebra of minstacks without operation hd is also noninjective, and type $[\mathrm{Nat} \times T]$ can be used to implement this algebra as well. The following algebra of mindeques without operations hd and lt is however injective:

$$(\,[T] \mid [\,], (=[\,]), \vdash, tl, \dashv, ft, \downarrow).$$

If two lists of equal length differ in one position, both lists can be reduced by means of tl and ft to singleton lists containing these elements and operation $\downarrow$ will then return different results for these singleton lists. Implementation of these mindeques using a pair of minstacks therefore requires the full set of operations of algebra MS.

Chapter 9

Skew heaps

In this chapter we design and analyze functional programs for a number of priority queue operations. At the specification level (Section 9.1) priority queues are considered as bags, whereas they are represented by pointer-structures at the implementation level (Section 9.5). In the major part of this chapter, however, priority queues will be described in terms of trees so as to achieve a suitable separation of concerns (Sections 9.2 and 9.4).

The implementations of priority queues treated in this chapter are based upon the skew heaps of Sleator and Tarjan [30]. In terms of the appropriate tree types we are able to give concise programs for the various priority queue operations, and these programs form the starting-point for a formal performance analysis of skew heaps. Our systematic approach results in improvements for the programs as well as tighter bounds for the amortized costs thereof. The required potential functions will be derived as much as possible.

9.1 Priority queues

The problem under consideration is the design of efficient implementations of an abstract algebra of priority queues PQ to be defined below. The data type of PQ is $\Lbag \mathrm{Int} \Rbag$, and PQ should comprise all operations that pertain to bags in programs like $sort0$ presented below. This program puts a list of integers into ascending order by means of a number of bag operations ($sort0 \in [\mathrm{Int}] \rightarrow [\mathrm{Int}]$):

$$
\begin{array}{lll}
sort0 & = & g \circ f \\
 & & \begin{array}{llll}
 \lbrack\!\lbrack \; f.[\,] & = & \Lbag\,\Rbag & \\
 \quad f.(a \vdash s) & = & \Lbag a \Rbag \oplus f.s & \\
 \quad g.B & = & [\,] & , B = \Lbag\,\Rbag \\
 & \lbrack\!\rbrack & {\downarrow}B \vdash g.({\Downarrow}B) & , B \neq \Lbag\,\Rbag \\
 \rbrack\!\rbrack. & & &
 \end{array}
\end{array}
$$

As a possible definition for PQ we propose an algebra with six operations:

$$\mathsf{PQ} = (\,\Lbag \mathrm{Int} \Rbag \mid \Lbag\,\Rbag, \Lbag \cdot \Rbag, \oplus, (=\Lbag\,\Rbag), \downarrow, \Downarrow\,),$$

where the last two operations both have domain $\lbag \mathrm{Int} \rbag \backslash \{\lbag\,\rbag\}$. Since any bag can be constructed by means of the first three operations, and since any pair of distinct bags can be distinguished by means of the last three operations, algebra PQ is bijective. As explained in Section 2.5, any refinement of PQ may therefore be described by means of a (surjective) abstraction function. The problem is thus to design a concrete algebra with signature

$$\mathsf{SH} = (\, S \mid \mathsf{empty}, \mathsf{single}, \mathsf{union}, \mathsf{isempty}, \mathsf{min}, \mathsf{delmin}\,),$$

say, such that SH refines PQ under abstraction $[\![\cdot]\!]$.

In comparison with [30], we have included in PQ the operation $\lbag\cdot\rbag$ instead of the abstract counterpart of operation insert $\in \mathrm{Int} \times S \to S$, which satisfies

$$[\![\mathsf{insert}.a.p]\!] = \lbag a \rbag \oplus [\![p]\!].$$

Of course, there is no essential difference between our notion of priority queues and theirs, because, on the one hand operation insert can be implemented as

$$\mathsf{insert}.a.p = \mathsf{union}.(\mathsf{single}.a).p,$$

and on the other hand single can be defined as

$$\mathsf{single}.a = \mathsf{insert}.a.\mathsf{empty}.$$

We prefer to include single as a priority queue operation because it is simpler than insert. Note, however, that inclusion of insert instead of single as operation in PQ would make operation union superfluous for the refinement of $sort0$. This operation is nevertheless included because it is an important operation in other applications (e.g., computation of minimum-cost spanning trees); some authors emphasize its presence by using the term "*mergeable* priority queue".

With algebra SH at our disposal, we readily obtain the following concrete program as refinement of $sort0$:

$$\begin{array}{lll}
sort1 & = & g \circ f \\
 & & |[\; f.[\,] \;=\; \mathsf{empty} \\
 & & \quad f.(a \vdash s) \;=\; \mathsf{union}.(\mathsf{single}.a).(f.s) \\
 & & \quad g.p \;=\; [\,] \qquad\qquad\qquad\qquad , \;\mathsf{isempty}.p \\
 & & \qquad\quad [\!]\;\; \mathsf{min}.p \vdash g.(\mathsf{delmin}.p) \quad , \neg\mathsf{isempty}.p \\
 & &]|.
\end{array}$$

Of course, we are only interested in *efficient* refinements of PQ. Therefore, we end this section with a number of efficiency requirements.

To begin with, we aim at a representation for priority queues that takes linear space, i.e., $O(\#[\![p]\!])$ space to represent priority queue p. Next, we formulate some requirements for the time complexity of the operations, for which the amortized costs are defined according to the scheme of Section 5.5:

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{single}](a) & = & \mathcal{T}[\mathsf{single}](a) + \Phi.(\mathsf{single}.a) \\
\mathcal{A}[\mathsf{union}](p,q) & = & \mathcal{T}[\mathsf{union}](p,q) + \Phi.(\mathsf{union}.p.q) - \Phi.p - \Phi.q \\
\mathcal{A}[\mathsf{isempty}](p) & = & \mathcal{T}[\mathsf{isempty}](p) \\
\mathcal{A}[\mathsf{min}](p) & = & \mathcal{T}[\mathsf{min}](p) \\
\mathcal{A}[\mathsf{delmin}](p) & = & \mathcal{T}[\mathsf{delmin}](p) + \Phi.(\mathsf{delmin}.p) - \Phi.p.
\end{array}$$

We would like to have $O(1)$ bounds for the amortized costs of all priority queue operations. But, since we are able to sort with these operations (e.g., by means of program $sort1$), we have to allow that some operations have logarithmic bounds for their amortized costs. In any case, however, we want the computation of $sort1.s$ to take $O(\#s \log \#s)$ time. In addition we want the potential function to be nonnegative and small (e.g., $0 \le \Phi.p \le \#[\![p]\!]$).

9.2 Top-down skew heaps

It goes without saying that we try to keep the implementation of an algebra as simple as possible. We hope to achieve this by taking as our principal working hypothesis that—ideally—any design decision should be justly motivated by a phrase like "The simplest solution we can think of is now to ...".

Under this hypothesis, we shall derive a very simple implementation of priority queues. Despite its simplicity, however, it is not easy to show that the efficiency requirements are met. In order to show that the amortized costs of the priority queue operations are sufficiently low, we try to *derive*—as much as possible—a suitable potential function.

So, let us now see whether these intentions come to anything good. In order to get started, we first have to make a decision regarding data type S. In the previous section we have already concluded that abstraction $[\![\cdot]\!]$ must be surjective, hence S should satisfy:

(1) $(\forall B : B \in \Lbag\mathrm{Int}\Rbag : (\exists p : p \in S : B = [\![p]\!]))$.

In view of this requirement the simplest concrete type for S we can think of is [Int], or a subset thereof. Unfortunately, as the reader may verify, it is not at all clear how the efficiency requirements can be met in this way.

The next candidate for S is $\langle\mathrm{Int}\rangle$. We decide to let S be a subset of $\langle\mathrm{Int}\rangle$, that is, we require S to satisfy:

(2) $S \subseteq \langle\mathrm{Int}\rangle$.

This restriction on S enables us to define $[\![\cdot]\!]$ already at this point, namely as

$$\begin{array}{lcl}
[\![\langle\,\rangle]\!] & = & \Lbag\,\Rbag \\
[\![\langle t,a,u\rangle]\!] & = & [\![t]\!] \oplus \Lbag a\Rbag \oplus [\![u]\!].
\end{array}$$

This representation takes linear space.

9.2.1 Introduction of $\bowtie$

We now turn our attention to the operations of SH. "On the fly" we will derive additional restrictions on S and a suitable definition for S will be presented at the end of this section.

First of all, we deal with the operations empty, $\mathsf{isempty}$, and single, since their definitions are fixed by our choice for $[\![\cdot]\!]$. More precisely, $\langle\,\rangle$ is the only element of $\langle\mathrm{Int}\rangle$ whose $[\![\cdot]\!]$-value equals $\lbag\,\rbag$, so we have to define

$$\begin{array}{lcl} \mathsf{empty} & = & \langle\,\rangle \\ \mathsf{isempty}.z & = & z = \langle\,\rangle. \end{array}$$

Similarly, since $\langle a\rangle$ is the only element in $\langle\mathrm{Int}\rangle$ whose $[\![\cdot]\!]$-value equals $\lbag a\rbag$, we define

$$\mathsf{single}.a = \langle a\rangle.$$

These three programs all have constant time complexity.

Next, we consider operation min. For this operation we propose the simplest $O(1)$ program we can think of, viz.

$$\mathsf{min}.\langle t,a,u\rangle = a.$$

This program is correct if S satisfies the additional restriction

(3) $\quad \langle t,a,u\rangle \in S \;\Rightarrow\; a = {\downarrow}\langle t,a,u\rangle\,,$

which expresses that the root should contain the minimum value.

Of the remaining two operations, we first consider delmin. We derive, for $\langle t,a,u\rangle \in S$:

$$\begin{array}{cl} & [\![\mathsf{delmin}.\langle t,a,u\rangle]\!] \\ = & \{\ \text{specification of } \mathsf{delmin}\ \} \\ & {\Downarrow}[\![\langle t,a,u\rangle]\!] \\ = & \{\ \text{definition of } {\Downarrow}\ \} \\ & [\![\langle t,a,u\rangle]\!] \ominus \lbag{\downarrow}\langle t,a,u\rangle\rbag \\ = & \{\ \text{definition of } [\![\cdot]\!];\ \text{restriction (3) on } S\ \} \\ & ([\![t]\!] \oplus \lbag a\rbag \oplus [\![u]\!]) \ominus \lbag a\rbag \\ = & \{\ \text{bag calculus}\ \} \\ & [\![t]\!] \oplus [\![u]\!]. \end{array}$$

Since union is required to satisfy $[\![\mathsf{union}.x.y]\!] = [\![x]\!] \oplus [\![y]\!]$ as well, we can deal with delmin and union in a uniform way by introducing operator $\bowtie$ ("meld") satisfying

(4) $\quad \bowtie\ \in S \times S \to S$

(5) $\quad [\![x \bowtie y]\!] = [\![x]\!] \oplus [\![y]\!].$

The resulting programs are

$$\begin{array}{lcl} \mathsf{union}.x.y & = & x \bowtie y \\ \mathsf{delmin}.\langle t,a,u\rangle & = & t \bowtie u, \end{array}$$

provided that S satisfies

(6) $\quad \langle t,a,u\rangle \in S \;\Rightarrow\; t \in S \;\wedge\; u \in S.$

In order not to impose stronger conditions on the priority queue operations than necessary, we choose for S the *largest* set satisfying requirements (1), (2), (3), and (6). That is, we define

$$S = \{z \mid z \in \langle \mathrm{Int}\rangle \;\wedge\; H.z\},$$

where H is the *heap condition* for trees, defined by

$$\begin{array}{lcl} H.\langle\,\rangle & \equiv & \text{true} \\ H.\langle t,a,u\rangle & \equiv & H.t \;\wedge\; a \le {\downarrow}t \;\wedge\; a \le {\downarrow}u \;\wedge\; H.u. \end{array}$$

Notice that the above derivation hinges on the decision to impose restriction (3) on S; from this decision it follows in a calculational manner that S becomes the set of (binary) heaps.

From $H.\langle\,\rangle$ and $H.\langle a\rangle$ we infer that empty and single indeed satisfy their specifications, which leaves us with the problem of designing an efficient program for $\bowtie$.

9.2.2 Implementation of $\bowtie$

It is hardly surprising that a program for $\bowtie$ will be recursive. In the inductive derivation of candidates for $x \bowtie y$ below, we first focus on x, for which we distinguish the cases $x = \langle\,\rangle$ and $x \ne \langle\,\rangle$. Since the specification of $x \bowtie y$ is symmetric in x and y, the corresponding cases for y can then be settled by interchanging the role of x and y.

Case $x = \langle\,\rangle$. The obvious candidate for $\langle\,\rangle \bowtie y$ is y.

Case $x \ne \langle\,\rangle$. In this case we dissect x, say $x = \langle t,a,u\rangle$, but we refrain from dissecting y (just to keep it simple). Since part (4) of the specification of $\bowtie$ is too weak on its own, we have to consider part (5) first:

$$\begin{array}{ll} & [\![\langle t,a,u\rangle \bowtie y]\!] \\ = & \{\ (5);\ \text{definition of } [\![\cdot]\!];\ \oplus \text{ is associative}\ \} \\ & [\![t]\!] \oplus \lbag a \rbag \oplus [\![u]\!] \oplus [\![y]\!] \\ = & \{\ \oplus \text{ is symmetric (choosing 1 out of 3 options: } y,u \text{ or } t \text{ separate)}\ \} \\ & ([\![t]\!] \oplus [\![u]\!]) \oplus \lbag a \rbag \oplus [\![y]\!] \\ = & \{\ \text{ind. hypothesis (5) (choosing 1 out of 2 options: } t \bowtie u \text{ or } u \bowtie t)\ \} \\ & [\![t \bowtie u]\!] \oplus \lbag a \rbag \oplus [\![y]\!] \\ = & \{\ \text{definition of } [\![\cdot]\!] \text{ (choosing 1 out of 2 options: } \bowtie \text{ left or } \bowtie \text{ right)}\ \} \\ & [\![\langle t \bowtie u, a, y\rangle]\!]. \end{array}$$

Thus, for the time being, we take $\langle t \bowtie u, a, y\rangle$ as one out of twelve candidates for $x \bowtie y$. For this candidate, we now consider part (4) of the specification of $\bowtie$ (the heap condition). Assuming that $x \in S \wedge y \in S$, hence that $H.t \;\wedge\; a \leq {\downarrow}t \;\wedge\; a \leq {\downarrow}u \;\wedge\; H.u \;\wedge\; H.y$, we derive inductively:

$$
\begin{array}{ll}
 & H.\langle t \bowtie u, a, y\rangle \\
\equiv & \{ \text{ definition of } H \; \} \\
 & H.(t \bowtie u) \;\wedge\; a \leq {\downarrow}(t \bowtie u) \;\wedge\; a \leq {\downarrow}y \;\wedge\; H.y \\
\equiv & \{ \text{ ind. hypothesis (4), hence } H.t \wedge H.u \;\Rightarrow\; H.(t \bowtie u)\text{; assumption } \} \\
 & a \leq {\downarrow}(t \bowtie u) \;\wedge\; a \leq {\downarrow}y \\
\equiv & \{ \text{ ind. hypothesis (5), hence } [\![t \bowtie u]\!] = [\![t]\!] \oplus [\![u]\!]\text{; bag calculus } \} \\
 & a \leq {\downarrow}t \;\wedge\; a \leq {\downarrow}u \;\wedge\; a \leq {\downarrow}y \\
\equiv & \{ \text{ assumption; } a = {\downarrow}x \; \} \\
 & {\downarrow}x \;\leq {\downarrow}y\,.
\end{array}
$$

For the other candidates we obtain the same condition to ensure that (4) is satisfied in all cases.

So, we can solve the cases $x = \langle\,\rangle$ and $x \neq \langle\,\rangle \;\wedge\; {\downarrow}x \;\leq {\downarrow}y$. By interchanging x and y in the above derivation, we also obtain solutions for the cases $y = \langle\,\rangle$ and $y \neq \langle\,\rangle \;\wedge\; {\downarrow}y \;\leq\; {\downarrow}x$. Hence, we are done because the case analysis is now exhaustive. There is even some overlap in this case analysis and we can limit the number of cases to three by removing some of this overlap: instead of the two cases $x = \langle\,\rangle$ and $y = \langle\,\rangle$ we take case $x = \langle\,\rangle \;\wedge\; y = \langle\,\rangle$. (In Section 9.2.6 we will consider a slightly more complicated alternative.) In order to turn the above conditions into concrete guards, we introduce function m, satisfying $H.z \Rightarrow m.z{=}{\downarrow}z$, defined by

$$
\begin{array}{lcl}
m.\langle\,\rangle & = & \infty \\
m.\langle t, a, u\rangle & = & a.
\end{array}
$$

As a result, we obtain linearly recursive programs of the form

$$
\begin{array}{rclcll}
\langle\,\rangle & \bowtie & \langle\,\rangle & = & \langle\,\rangle & \\
\langle t, a, u\rangle & \bowtie & y & \doteq & \mathrm{B} & , a \leq m.y \\
x & \bowtie & \langle t, a, u\rangle & \doteq & \mathrm{B}' & , a \leq m.x,
\end{array}
$$

in which B is one of the twelve candidates for $x \bowtie y$ mentioned above and B′ is the corresponding candidate with y replaced by x. As for the possibilities for B, however, we can limit our attention to the following three candidates:

$$
\begin{array}{ll}
\text{(Y)} & \langle t \bowtie u, a, y\rangle \\
\text{(U)} & \langle t \bowtie y, a, u\rangle \\
\text{(T)} & \langle u \bowtie y, a, t\rangle,
\end{array}
$$

because, firstly, swapping the operands of $\bowtie$ is of no use, since our tentative program for $\bowtie$ is symmetric, and, secondly, swapping the subtrees only exchanges the role of left and right, which cannot induce essentially different results.

The cost measure defined by the dots in the above program scheme for $\bowtie$ will be called $\mathcal{N}$. Notice that $\mathcal{N}(E)$ is approximately equal to the number of integer comparisons performed in the evaluation of E (counting one comparison for each unfolding of the second or third alternative of $\bowtie$, also counting the comparisons with ∞).

The choice between (Y), (U), and (T) is motivated by the following observations. Consider program h (cf. program f in $sort1$, Section 9.1):

$$\begin{array}{lcl} h.[\,] & = & \langle\,\rangle \\ h.(a \vdash s) & = & \langle a\rangle \bowtie h.s. \end{array}$$

Then—as the reader may verify—$\mathcal{N}[h](s) \approx \frac{1}{4}(\#s)^2$, if we take (Y) for B and s decreasing. Similarly, we have that $\mathcal{N}[h](s) \approx \frac{1}{2}(\#s)^2$, if we take (U) for B and s decreasing. It is even the case that the last alternative of the program for $\bowtie$ is never applied in these computations of h. Hence, irrespective of our choice for B$'$ candidates (Y) and (U) drop out. Fortunately, candidate (T) cannot be eliminated in this way. The remaining program for $\bowtie$ reads:

$$\begin{array}{rcl} \langle\,\rangle \bowtie \langle\,\rangle & = & \langle\,\rangle \\ \langle t,a,u\rangle \bowtie y & \doteq & \langle u \bowtie y, a, t\rangle \quad ,a \le m.y \\ x \bowtie \langle t,a,u\rangle & \doteq & \langle u \bowtie x, a, t\rangle \quad ,a \le m.x. \end{array}$$

This program is almost the same as the program for top-down melding of skew heaps in [30], and Sleator and Tarjan have shown—informally—that their program is efficient in the amortized sense. The fact that t and u are "swapped" in the last two alternatives is crucial for the efficiency. In the next section we perform a formal amortized analysis of the above program for $\bowtie$. Note that $\mathcal{N}[\mathsf{delmin}](h.s)$ is approximately $\#(h.s)$ if s is increasing, hence some applications of delmin may take $\Omega(\#s)$ time during the evaluation of $sort1.s$; therefore an amortized analysis is really necessary.

9.2.3 Analysis of $\bowtie$

The amortized costs of $\bowtie$ are defined by

(7) $\quad \mathcal{A}[\bowtie](x,y) = \mathcal{N}[\bowtie](x,y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y.$

In order to facilitate computations with potential Φ, we introduce function φ and we choose the following recursive definition for Φ (cf. the recursive structure of $\langle\mathrm{Int}\rangle$).

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t + \varphi.t.u + \Phi.u. \end{array}$$

By writing $\varphi.t.u$ instead of $\varphi.t.a.u$, say, we make $\Phi.\langle t, a, u\rangle$ independent of the value of a. This shortens the calculations involving φ somewhat, and—as we shall see later on— this is a harmless design decision. (As a matter of fact, $\Phi.z$ will be independent of the values in z altogether because $\varphi.t.u$ will be independent of the values in t and u.)

Recall that we want Φ to be nonnegative and small. In order to fulfill these requirements we have chosen $\Phi.\langle\,\rangle = 0$. Moreover, we require φ to be nonnegative and small. Our main goal is to define φ in such a way that $\mathcal{A}[\bowtie](x, y)$ can be shown to be $O(\log(\sharp x + \sharp y))$. Here, $\sharp z$ denotes $1 + \# z$, hence it is positive and it makes sense to write things like $\log_2 \sharp z$ and $\sharp x/\sharp y$—also when empty trees are not excluded. In the sequel we will often use that $\sharp$ satisfies the following recurrence relation:

$$
\begin{array}{lcl}
\sharp\langle\,\rangle & = & 1 \\
\sharp\langle t, a, u\rangle & = & \sharp t + \sharp u .
\end{array}
$$

As a first step, we derive a recurrence relation for $\mathcal{A}[\bowtie]$.

Case $x = \langle\,\rangle \;\wedge\; y = \langle\,\rangle$. Then $\mathcal{N}[\bowtie](\langle\,\rangle, \langle\,\rangle) = 0$, and

$$
\begin{array}{cl}
 & \mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) \\
= & \{\ (7)\ \} \\
 & \mathcal{N}[\bowtie](\langle\,\rangle, \langle\,\rangle) + \Phi.(\langle\,\rangle \bowtie \langle\,\rangle) - \Phi.\langle\,\rangle - \Phi.\langle\,\rangle \\
= & \{\ \text{definition of } \bowtie;\ \Phi.\langle\,\rangle = 0\ \} \\
 & 0.
\end{array}
$$

Case $x = \langle t, a, u\rangle \;\wedge\; a \le m.y$. Then $\mathcal{N}[\bowtie](x, y) = 1 + \mathcal{N}[\bowtie](u, y)$, and $x \bowtie y = \langle u \bowtie y, a, t\rangle$, and

$$
\begin{array}{cl}
 & \mathcal{A}[\bowtie](x, y) \\
= & \{\ (7)\ \} \\
 & \mathcal{N}[\bowtie](x, y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y \\
= & \{\ \text{definition of } \bowtie;\ \text{definition of } \Phi\ \} \\
 & 1 + \mathcal{N}[\bowtie](u, y) + \Phi.(u \bowtie y) + \varphi.(u \bowtie y).t + \Phi.t - \Phi.t - \varphi.t.u - \Phi.u - \Phi.y \\
= & \{\ \text{definition of } \mathcal{A}\ \} \\
 & \mathcal{A}[\bowtie](u, y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u.
\end{array}
$$

By symmetry, we thus have:

$$
\begin{array}{lcll}
\mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) & = & 0 & \\
\mathcal{A}[\bowtie](\langle t, a, u\rangle, y) & = & \mathcal{A}[\bowtie](u, y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u & , a \le m.y \\
\mathcal{A}[\bowtie](x, \langle t, a, u\rangle) & = & \mathcal{A}[\bowtie](u, x) + \varphi.(u \bowtie x).t + 1 - \varphi.t.u & , a \le m.x.
\end{array}
\tag{8}
$$

Unfolding this recurrence relation several times and realizing that $\mathcal{A}[\bowtie](x, y)$ is defined in terms of x, y, and $x \bowtie y$ (cf. (7)), we aim at a result of the following form (symmetric in x and y):

(9) $\mathcal{A}[\bowtie](x,y) = \Gamma.(x \bowtie y) + \Delta.x + \Delta.y.$

Functions Γ and Δ are now derived inductively. Since $\mathcal{A}[\bowtie](\langle\,\rangle,\langle\,\rangle) = 0$, we take $\Gamma.\langle\,\rangle = 0$ and $\Delta.\langle\,\rangle = 0$. With case $x = \langle t,a,u\rangle \wedge a \leq m.y$ we deal as follows, using (9) as induction hypothesis:

$$
\begin{array}{ll}
 & \mathcal{A}[\bowtie](x,y) \\
= & \{\ (8)\ \} \\
 & \mathcal{A}[\bowtie](u,y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \{\ \text{induction hypothesis (9)}\ \} \\
 & \Gamma.(u \bowtie y) + \Delta.u + \Delta.y + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \{\ \text{definitions of } \Gamma \text{ and } \Delta\text{, see below}\ \} \\
 & \Gamma.\langle u \bowtie y, a, t\rangle + \Delta.\langle t,a,u\rangle + \Delta.y \\
= & \{\ \text{definition of } \bowtie\ \} \\
 & \Gamma.(x \bowtie y) + \Delta.x + \Delta.y.
\end{array}
$$

The remaining case follows by symmetry, hence (9) holds if we define

$$
\begin{array}{lcl}
\Gamma.\langle\,\rangle & = & 0 \\
\Gamma.\langle t,a,u\rangle & = & \Gamma.t + \varphi.t.u + \gamma \\
\Delta.\langle\,\rangle & = & 0 \\
\Delta.\langle t,a,u\rangle & = & \Delta.u - \varphi.t.u + \delta,
\end{array}
$$

with γ and δ constants to be chosen later on such that $\gamma + \delta = 1$.

A logarithmic bound for $\mathcal{A}[\bowtie](x,y)$ follows from (9) if we can define φ and choose γ and δ such that

(10) $\Gamma.z \leq \log_\alpha \sharp z \ \wedge\ \Delta.z \leq \log_\alpha \sharp z,$

for some constant α, $\alpha > 1$. To satisfy (10), we derive requirements on φ by induction on z.

Case $z = \langle\,\rangle$. Trivial, since $\sharp\langle\,\rangle = 1$.

Case $z = \langle t,a,u\rangle$. We derive for Γ and Δ, respectively:

$$
\begin{array}{ll}
 & \Gamma.\langle t,a,u\rangle \\
= & \{\ \text{definition of } \Gamma\ \} \\
 & \Gamma.t + \varphi.t.u + \gamma \\
\leq & \{\ \text{ind. hypothesis (10)}\ \} \\
 & \log_\alpha \sharp t + \varphi.t.u + \gamma \\
\leq & \{\ \text{upper bound } \varphi \text{ in (11)}\ \} \\
 & \log_\alpha \sharp\langle t,a,u\rangle
\end{array}
\qquad
\begin{array}{ll}
 & \Delta.\langle t,a,u\rangle \\
= & \{\ \text{definition of } \Delta\ \} \\
 & \Delta.u - \varphi.t.u + \delta \\
\leq & \{\ \text{ind. hypothesis (10)}\ \} \\
 & \log_\alpha \sharp u - \varphi.t.u + \delta \\
\leq & \{\ \text{lower bound } \varphi \text{ in (11)}\ \} \\
 & \log_\alpha \sharp\langle t,a,u\rangle.
\end{array}
$$

Hence, we require that φ satisfies for any t and u, using $\sharp\langle t,a,u\rangle = \sharp t + \sharp u$:

$$(11)\quad \log_\alpha \frac{\alpha^\delta \sharp u}{\sharp t + \sharp u} \le \varphi.t.u \le \log_\alpha \frac{\sharp t + \sharp u}{\alpha^\gamma \sharp t}.$$

The existence of such a φ is guaranteed if the lower bound never exceeds the upper bound. Therefore, we calculate (m and n correspond to $\sharp t$ and $\sharp u$, respectively; hence, m and n are positive integers):

$$
\begin{array}{cl}
 & (\forall\, m, n :: \log_\alpha \frac{\alpha^\delta n}{m+n} \le \log_\alpha \frac{m+n}{\alpha^\gamma m}) \\
\equiv & \{ \text{ monotonicity of } \log_\alpha\ (\alpha > 1) \ \} \\
 & (\forall\, m, n :: \frac{\alpha^\delta n}{m+n} \le \frac{m+n}{\alpha^\gamma m}) \\
\equiv & \{\ \gamma + \delta = 1\ \} \\
 & (\forall\, m, n :: \alpha m n \le (m+n)^2) \\
\equiv & \{\ \ \} \\
 & (\forall\, m, n :: \alpha \le 4 + \frac{(m-n)^2}{mn}) \\
\equiv & \{\ \ \} \\
 & \alpha \le 4.
\end{array}
$$

Hence, for any α, $1 < \alpha \le 4$, it follows from (9) and (10) that there exists a potential function for which

$$\mathcal{A}[\bowtie](x, y) \le \log_\alpha(\sharp x + \sharp y) + \log_\alpha \sharp x + \log_\alpha \sharp y.$$

Thus, we have shown that $\mathcal{A}[\bowtie](x, y)$ is $O(\log(\sharp x + \sharp y))$.

We are, however, not only interested in the asymptotic behaviour of $\mathcal{A}[\bowtie]$. We obtain a low upper bound for $\mathcal{A}[\bowtie](x, y)$ by choosing α as large as possible, viz. $\alpha = 4$. Then

$$\mathcal{A}[\bowtie](x, y) \le \tfrac{3}{2} \log_2(\sharp x + \sharp y).$$

This bound improves the bound of [30] by a factor 2. (In the next section we show that the bound can be reduced even further.)

So much for the amortized costs of $\bowtie$. Next we try to define φ such that it is nonnegative. Evidently, this requires that the upper bound for φ in (11) is nonnegative. We observe:

$$
\begin{array}{cl}
 & (\forall\, m, n :: 0 \le \log_4 \frac{m+n}{4^\gamma m}) \\
\equiv & \{ \text{ monotonicity of } \log_4 \ \} \\
 & (\forall\, m, n :: 1 \le \frac{m+n}{4^\gamma m}) \\
\equiv & \{\ \ \} \\
 & (\forall\, m, n :: 4^\gamma \le \frac{m+n}{m}) \\
\equiv & \{\ \ \} \\
 & 4^\gamma \le 1 \\
\equiv & \{ \text{ monotonicity of exponentiation } \} \\
 & \gamma \le 0.
\end{array}
$$

So, we have to take $\gamma \le 0$ to ensure that φ can be defined nonnegative. Furthermore, to obtain a small potential, we minimize the lower bound of φ in (11) by choosing δ as small as possible under the constraints $\gamma + \delta = 1$ and $\gamma \le 0$. This yields $\gamma = 0$ and $\delta = 1$. The smallest nonnegative choice for φ thus is:

$$\varphi.t.u = \log_4 \frac{4\sharp u}{\sharp t + \sharp u} \max 0$$

Then $0 \le \varphi.t.u \le 1$, and we have proved the following lemma.

Lemma 9.1
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t \;+\; \log_4 \frac{4\sharp u}{\sharp t+\sharp u} \max 0 \;+\; \Phi.u. \end{array}$$

Then $0 \le \Phi.z \le \#z$, and the amortized costs of top-down melding satisfy

$$\mathcal{A}[\bowtie](x,y) \le \tfrac{3}{2}\log_2(\sharp x + \sharp y).$$

□

9.2.4 Refined analysis of $\bowtie$

As pointed out to us by D.D. Sleator, it is possible to reduce the constant in the bound for $\mathcal{A}[\bowtie]$ from $\frac{3}{2}$ to approximately 1.44. To achieve this improvement, we refine the analysis of $\bowtie$ by introducing independent constants in the bounds for Γ and Δ. That is, instead of (10) we take

(12) $\Gamma.z \le \log_\alpha \sharp z \;\wedge\; \Delta.z \le \log_\beta \sharp z,$

with $\alpha > 1$ and $\beta > 1$. As may be gathered from the derivations in the previous section, these inequalities are satisfied if φ satisfies (cf. (11))

(13) $\log_\beta \dfrac{\beta^\delta \sharp u}{\sharp t + \sharp u} \le \varphi.t.u \le \log_\alpha \dfrac{\sharp t + \sharp u}{\alpha^\gamma \sharp t}.$

The existence of such a φ is now guaranteed if for any positive m and n (writing $\varepsilon = \log_\alpha \beta$):

$$\begin{array}{ll} & \log_\beta \frac{\beta^\delta n}{m+n} \le \log_\alpha \frac{m+n}{\alpha^\gamma m} \\ \equiv & \{\ \log_\alpha x = \log_\beta x^\varepsilon;\ \text{monotonicity of } \log_\beta\ (\beta > 1)\ \} \\ & \frac{\beta^\delta n}{m+n} \le (\frac{m+n}{\alpha^\gamma m})^\varepsilon \\ \equiv & \{\ \alpha^\varepsilon = \beta \text{ and } \gamma + \delta = 1\ \} \\ & \beta \le \frac{(m+n)^{\varepsilon+1}}{m^\varepsilon n} \\ \equiv & \{\ \ \} \\ & \beta \le \frac{(1+\frac{m}{n})^{\varepsilon+1}}{(\frac{m}{n})^\varepsilon}. \end{array}$$

So, proceeding in terms of β and ε ($\alpha = \beta^{1/\varepsilon}$), it suffices to choose β and ε such that $\beta > 1$, $\varepsilon > 0$, and

$$(14) \quad \beta \leq \frac{(\varepsilon{+}1)^{\varepsilon+1}}{\varepsilon^{\varepsilon}},$$

since function $\frac{(1+x)^{\varepsilon+1}}{x^{\varepsilon}}$ is minimal at $x = \varepsilon$.

It is easily seen that such β and ε exist, but we want to choose them such that we obtain a small bound for $\mathcal{A}[\bowtie]$. We have

$$\begin{array}{cl} & \mathcal{A}[\bowtie](x,y) \\ = & \{\ (9)\ \} \\ & \Gamma.(x \bowtie y) + \Delta.x + \Delta.y \\ \leq & \{\ (12)\ \} \\ & \log_\alpha \sharp(x \bowtie y) + \log_\beta \sharp x + \log_\beta \sharp y \\ \leq & \{\ \text{definition of } \sharp\ \} \\ & \log_\alpha(\sharp x + \sharp y) + 2\log_\beta(\sharp x + \sharp y) \\ = & \{\ \log_2 \alpha = (\log_2 \beta)/\varepsilon\ \} \\ & \frac{\varepsilon+2}{\log_2 \beta} \log_2(\sharp x + \sharp y), \end{array}$$

and therefore we want to minimize $\frac{\varepsilon+2}{\log_2 \beta}$ under constraint (14) over all $\beta > 1$ and $\varepsilon > 0$. For this purpose we may take equality in (14), since, for fixed ε, $\frac{\varepsilon+2}{\log_2 \beta}$ decreases as β increases. Using this equality gives

$$\frac{\varepsilon + 2}{(\varepsilon{+}1)\log_2(\varepsilon{+}1) - \varepsilon\log_2 \varepsilon}$$

as quantity to minimize over all positive ε. This gives $1/\log_2 \phi$ (≈ 1.44) as minimal value at $\varepsilon = \phi$ (≈ 1.618), where ϕ is the golden ratio. From the equality in (14) we get $\beta = \phi^{\phi+2}$ (≈ 5.70) and the definition of ε gives $\alpha = \phi^{2\phi-1}$ (≈ 2.93). In the same fashion as in the previous section, the following lemma now follows.

Lemma 9.2
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Phi.t \ + \ \log_\beta \frac{\beta\sharp u}{\sharp t+\sharp u} \max 0 \ + \ \Phi.u, \end{array}$$

with $\beta = \phi^{\phi+2}$. Then $0 \leq \Phi.z \leq \#z$, and the amortized costs of top-down melding satisfy

$$\mathcal{A}[\bowtie](x,y) \leq \log_\phi(\sharp x + \sharp y).$$

□

Since the above analysis does not leave much room for improvement, we conjecture that our bound for the amortized costs of $\bowtie$ is tight; that is, we conjecture $(\forall\, \eta : \eta < 1/\log_2 \frac{\sqrt{5}+1}{2} : (\exists\, x, y :: \mathcal{A}[\bowtie](x,y) > \eta \, \log_2(\sharp x + \sharp y)))$, for any potential Φ.

9.2.5 Results for priority queue operations

We have derived the following programs for the priority queue operations:

$$\begin{array}{lcl}
\mathsf{empty} & \doteq & \langle\,\rangle \\
\mathsf{isempty}.z & \doteq & z = \langle\,\rangle \\
\mathsf{single}.a & \doteq & \langle a\rangle \\
\mathsf{min}.\langle t,a,u\rangle & \doteq & a \\
\mathsf{union}.x.y & \doteq & x \bowtie y \\
\mathsf{delmin}.\langle t,a,u\rangle & \doteq & t \bowtie u.
\end{array}$$

The dots in these programs together with those in the program for $\bowtie$ define cost measure $\mathcal{T}$. As a result, we have that the amortized costs of empty, $\mathsf{isempty}$, single, and min are $O(1)$. Furthermore, the amortized costs of union are logarithmic because $\mathcal{A}[\mathsf{union}](x,y) = 1 + \mathcal{A}[\bowtie](x,y)$, and for delmin we have:

$$\begin{array}{ll}
& \mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle) \\
= & \{\ \text{definition of } \mathcal{A}\ \} \\
& \mathcal{T}[\mathsf{delmin}](\langle t,a,u\rangle) + \Phi.(\mathsf{delmin}.\langle t,a,u\rangle) - \Phi.\langle t,a,u\rangle \\
= & \{\ \text{definitions of } \mathsf{delmin},\ \mathcal{T}, \text{ and } \Phi\ \} \\
& 1 + \mathcal{N}[\bowtie](t,u) + \Phi.(t \bowtie u) - \Phi.t - \varphi.t.u - \Phi.u \\
= & \{\ \text{definition of } \mathcal{A}\ \} \\
& 1 + \mathcal{A}[\bowtie](t,u) - \varphi.t.u \\
\le & \{\ \varphi \text{ is nonnegative}\ \} \\
& 1 + \mathcal{A}[\bowtie](t,u).
\end{array}$$

Hence, $\mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle)$ is $O(\log \sharp\langle t,a,u\rangle)$. Note that such a conclusion can only be drawn if φ is bounded from below by a constant (e.g., if φ is nonnegative).

Thus—hiding the particular implementation of priority queue operations—we have as results for the amortized costs (cf. Lemma 9.2):

$$\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{union}](p,q) & \le & 1.44\ \log_2(2 + \#[\![p]\!] + \#[\![q]\!]) \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \le & 1.44\ \log_2(1 + \#[\![p]\!]) \text{ comparisons.}
\end{array}$$

Moreover, there exists a potential function satisfying $0 \le \Phi.p \le \#[\![p]\!]$.

9.2.6 Comparison with Sleator and Tarjan's results

On page 54 in [30], Sleator and Tarjan describe top-down melding as follows: "we meld by merging the right paths of the two trees and then swapping the left and right children of every node on the merge path *except the lowest*". Our program for $\bowtie$ is simpler because it does not have such a strange exception.

The potential function of Sleator and Tarjan for top-down melding is roughly given by

$$\varphi.t.u = \begin{cases} 0 & , \sharp u \le \sharp t \\ 1 & , \sharp t < \sharp u \end{cases} \qquad \left(= \left\lceil \log_2 \frac{2\sharp u}{\sharp t + \sharp u} \right\rceil \max 0\right).$$

This potential satisfies bounds (11) if we take $\alpha = 2$, $\gamma = 0$, and $\delta = 1$. This potential is simpler, but the corresponding bound for $\mathcal{A}[\bowtie]$ is more than twice as large:

$$\mathcal{A}[\bowtie](x, y) \le 3 \log_2(\sharp x + \sharp y).$$

Sleator and Tarjan also discussed a "lazy" version of top-down melding that terminates as soon as one of the heaps is empty. This corresponds to the following version for $\bowtie$, which also follows from our derivation in Section 9.2.2:

$$\begin{array}{rcll} \langle\,\rangle \bowtie y & = & y & \\ x \bowtie \langle\,\rangle & = & x & \\ \langle t, a, u\rangle \bowtie y & \doteq & \langle u \bowtie y, a, t\rangle & , y \ne \langle\,\rangle \ \wedge \ a \le m.y \\ x \bowtie \langle t, a, u\rangle & \doteq & \langle u \bowtie x, a, t\rangle & , x \ne \langle\,\rangle \ \wedge \ a \le m.x, \end{array}$$

where m is only used for nonempty trees.

For this implementation of $\bowtie$ one can derive that $\mathcal{A}[\bowtie](\langle\,\rangle, y) = 0 \le \Gamma.(\langle\,\rangle \bowtie y) + \Delta.\langle\,\rangle + \Delta.y$, provided that Γ and Δ are nonnegative. Since $\gamma = 0$, $\delta = 1$, and $0 \le \varphi.t.u \le 1$, functions Γ and Δ are indeed nonnegative. By induction it now follows that (cf. (9))

$$\mathcal{A}[\bowtie](x, y) \le \Gamma.(x \bowtie y) + \Delta.x + \Delta.y,$$

from which we infer that this alternative program for $\bowtie$ is at least as good the previous one. However, for a decreasing list s we have that $\mathcal{N}[h](s)$ is $\Omega(\#s \log \#s)$ for function h defined in Section 9.2.2, so the amortized cost of $\bowtie$ is still logarithmic. We conjecture that $1.44 \log_2(\sharp x + \sharp y)$ is also a tight bound for $\mathcal{A}[\bowtie](x, y)$ in this case .

9.3 Intermezzo on sorting

As illustration of the usage of skew heaps, we analyze to what extent sorting can be done efficiently using skew heaps. One way is to use top-down skew heaps in program *sort*1 (see Section 9.1); then computation of *sort*1.*s* takes $2.88\#s \log_2 \#s + O(\#s)$ comparisons in the worst-case. However, with a more sophisticated implementation of function f in *sort*1 we obtain a sorting program that uses at most $1.44\#s \log_2 \#s + O(\#s)$ comparisons. The idea is to see to it that in each application of union only priority queues of about equal size are united, so that computation of $f.s$ takes $O(\#s)$ time only.

$$
\begin{array}{lll}
PQsort & = & g \circ f \\
 & & \|[f = f_1 \circ f_0 \\
 & & \quad \|[\; f_0.[\,] \quad = \quad [\,] \\
 & & \quad\quad f_0.(a \vdash r) \quad = \quad f_0.r \dashv \mathsf{single}.a \\
 & & \quad\quad f_1.[\,] \quad = \quad \mathsf{empty} \\
 & & \quad\quad f_1.[p] \quad = \quad p \\
 & & \quad\quad f_1.(p \vdash q \vdash ps) \quad = \quad f_1.(ps \dashv \mathsf{union}.p.q) \\
 & & \quad]| \\
 & & \quad g.p \;\; = \;\; [\,] \qquad\qquad\qquad\qquad , \; \mathsf{isempty}.p \\
 & & \qquad\quad [] \;\; \mathsf{min}.p \vdash g.(\mathsf{delmin}.p) \;\; , \neg \mathsf{isempty}.p \\
 & &]|.
\end{array}
$$

Remark 9.3
Of course, $PQsort$ is not intended as a practical application of mergeable priority queues. If it is only for the above program, it is better to take the set of ascending lists of integers as definition for S in Section 9.2.1. Implementing $\mathsf{union}.p.q$ as the merge of lists p and q and implementing the other operations in the obvious way, the above program then reduces to *Mergesort* (see Example 4.4), which takes at most $\#s \log_2 \#s$ comparisons. (By the way, merging ascending lists may be considered as a special case of melding heaps. This is reflected by the following program for merging:

$$
\begin{array}{rcllll}
[\,] & \bowtie & [\,] & = & [\,] & \\
(a \vdash u) & \bowtie & y & = & a \vdash u \bowtie y & , a \le m.y \\
x & \bowtie & (a \vdash u) & = & a \vdash u \bowtie x & , a \le m.x,
\end{array}
$$

where $m.[\,] = \infty$ and $m.(a \vdash u) = a$. To compute $x \bowtie y$, however, this program takes $O(\#x + \#y)$ time instead of logarithmic time for top-down skew heaps.)
□

It is not trivial that $\mathcal{A}[f](s)$, which equals $\mathcal{T}[f](s) + \Phi.(f.s)$, is linear in $\#s$. To show this formally, we introduce amortized costs for the components of f:

$$
\begin{array}{lll}
\mathcal{A}[f_0](s) & = & \mathcal{T}[f_0](s) + \Phi^*.(f_0.s) \\
\mathcal{A}[f_1](ps) & = & \mathcal{T}[f_1](ps) + \Phi.(f_1.ps) - \Phi^*.ps,
\end{array}
$$

such that $\mathcal{A}[f](s) = \mathcal{A}[f_0](s) + \mathcal{A}[f_1](f_0.s)$. Here, potential Φ^* is defined for lists of priority queues by $\Phi^*.[\,] = 0$ and $\Phi^*.(p \vdash ps) = \Phi.p + \Phi^*.ps$.

Since it is easy to show that $\mathcal{A}[f_0](s)$ is linear in $\#s$, we focus our attention on $\mathcal{A}[f_1]$, for which we have the following recurrence relation:

$$
\begin{array}{lll}
\mathcal{A}[f_1]([\,]) & = & \mathcal{A}[\mathsf{empty}] \\
\mathcal{A}[f_1]([p]) & = & 0 \\
\mathcal{A}[f_1](p \vdash q \vdash ps) & = & \mathcal{A}[\mathsf{union}](p,q) + \mathcal{A}[f_1](ps \dashv \mathsf{union}.p.q).
\end{array}
$$

In order that $\mathcal{A}[f_1](s)$ is linear, it is important that in each application of union only priority queues of about equal sizes are united (more precisely, they differ at most a factor of two in size). This fact is exploited in the following lemma.

Lemma 9.4
For all natural k and l,

$$\mathcal{A}[f_1](ps) \leq \mathcal{A}[\mathsf{empty}] + 1.44\ 2^k\ (\Sigma\, i : 1 \leq i \leq k : (l+1+i)/2^i),$$

provided ps satisfies:

(i) $\#ps \leq 2^k$
(ii) $(\forall\, i : 0 \leq i < \#ps : \#[\![ps.i]\!] \leq 2^l)$.

Proof By induction on k.

Case $k = 0$. Then $\mathcal{A}[f_1](ps)$ is either $\mathcal{A}[\mathsf{empty}]$ or 0, hence at most $\mathcal{A}[\mathsf{empty}]$.

Case $k = m{+}1$. We consider $\lfloor \#ps/2 \rfloor$ successive unfoldings of f_1. The result of these unfoldings is called qs, and may be defined as follows. In case $\#ps$ is even, list qs satisfies $\#qs = \#ps/2$, and $qs.i = \mathsf{union}.(ps.(2i)).(ps.(2i{+}1))$ for $0{\leq}i{<}\#qs$. In case $\#ps$ is odd, $\#qs = (\#ps{+}1)/2$, $qs.0 = ps.(\#ps{-}1)$, and $qs.(i{+}1) = \mathsf{union}.(ps.(2i)).(ps.(2i{+}1))$ for $0{\leq}i{<}\#qs{-}1$.

Now we may apply the induction hypothesis to qs, since $\#qs \leq 2^m$ (cf. (i)) and $\#[\![qs.i]\!] \leq 2^{l+1}$ (cf. (ii)):

$$
\begin{array}{ll}
 & \mathcal{A}[f_1](ps) \\
= & \{\ \text{unfold recurrence relation for } \mathcal{A}[f_1]\ \lfloor \#ps/2 \rfloor \text{ times}\ \} \\
 & (\Sigma\, i : 0 \leq i < \lfloor \#ps/2 \rfloor : \mathcal{A}[\mathsf{union}](ps.(2i), ps.(2i{+}1))) + \mathcal{A}[f_1](qs) \\
\leq & \{\ \text{bound for } \mathcal{A}[\mathsf{union}], \text{ see Section 9.2.5}\ \} \\
 & (\Sigma\, i : 0 \leq i < \lfloor \#ps/2 \rfloor : 1.44 \log_2(2{+}\#[\![ps.(2i)]\!]{+}\#[\![ps.(2i{+}1]\!]))+\mathcal{A}[f_1](qs) \\
\leq & \{\ \text{(ii) and } l \geq 0;\ \lfloor \#ps/2 \rfloor \leq 2^m\ \} \\
 & 1.44\ 2^m(l{+}2) + \mathcal{A}[f_1](qs) \\
\leq & \{\ \text{induction hypothesis}\ \} \\
 & 1.44\ 2^m(l{+}2) + \mathcal{A}[\mathsf{empty}] + 1.44\ 2^m\ (\Sigma\, i : 1 \leq i \leq m : (l{+}2{+}i)/2^i) \\
= & \{\ \text{calculus}\ \} \\
 & \mathcal{A}[\mathsf{empty}] + 1.44\ 2^{m+1}\ (\Sigma\, i : 0 \leq i \leq m : (l{+}1{+}i{+}1)/2^{i+1}) \\
= & \{\ k = m{+}1\ \} \\
 & \mathcal{A}[\mathsf{empty}] + 1.44\ 2^k\ (\Sigma\, i : 1 \leq i \leq k : (l{+}1{+}i)/2^i).
\end{array}
$$

□

Instantiation of this lemma with $k = \lceil \log_2(\#s{+}1) \rceil$, $l = 0$, and $ps = f_0.s$ yields, using $\#(f_0.s) = \#s$:

$$\mathcal{A}[f_1](f_0.s) \leq \mathcal{A}[\mathsf{empty}] + 2.88(\#s{+}1)\ (\Sigma\, i : 1 \leq i \leq \lceil \log_2(\#s{+}1) \rceil : \frac{i{+}1}{2^i}).$$

This implies that $\mathcal{A}[f_1](f_0.s)$ is $O(\#s)$, since the summation has the same order of magnitude as $\int_0^\infty \frac{x+1}{2^x} dx$, which is $O(1)$. Hence, we have shown that $\mathcal{A}[f](s)$ is linear in s (and also that $\mathcal{T}[f](s)$ is linear).

9.4 Bottom-up skew heaps

Reconsidering the derivation of the top-down skew heaps we conclude that the decision to use binary trees as representations for bags is in fact the main design decision. Keeping matters as simple as possible, the introduction of the heap condition and the operator $\bowtie$ followed in a natural way. The implementations derived for $\bowtie$ both have logarithmic amortized costs and there seems to be no way to improve this. In [30], however, Sleator and Tarjan present the bottom-up skew heaps as alternative implementation of priority queues. For bottom-up skew heaps they achieve a logarithmic bound for the amortized costs of delmin and $O(1)$ bounds for the remaining operations; hence, the amortized costs of union are reduced to $O(1)$. How come?

Well, Sleator and Tarjan still use binary trees to represent bags, but—formally speaking—they take a different view of binary trees. In the previous section we have introduced set $\langle \mathrm{Int} \rangle$ in the common way, viz. as the smallest solution of

$$X: \quad X = \{\langle\ \rangle\} \ \cup\ X \times \mathrm{Int} \times X.$$

This "top-down" view of binary trees forces us to dissect a nonempty tree into its left subtree, its root, and its right subtree, and in this way we are almost inescapably led to the programs found for $\bowtie$. It appears that the way one introduces binary trees constitutes an important design decision, for we have to take a special view of binary trees to arrive at the bottom-up skew heaps.

In case of bottom-up skew heaps the following definition is appropriate. We define set $\langle \mathrm{Int} \rangle_1$ as the smallest solution of

$$X: \quad X = [X \times \mathrm{Int}].$$

We have already described this tree type in Section 6.2.2, from which we recall that elements of $\langle \mathrm{Int} \rangle_1$ are interpreted as binary trees in the following way: $[\,]$ stands for the empty tree and $\langle t, a \rangle \vdash u$ for a nonempty tree with left subtree t, root a, and right subtree u. In addition, we will use that nonempty elements of $\langle \mathrm{Int} \rangle_1$ can be dissected as $u \dashv \langle t, a \rangle$ (where a is the last element of the inorder traversal of this tree). For the programs in the sequel the following algebra is appropriate:

$$(\,\langle \mathrm{Int} \rangle_1 \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft\,).$$

As shown in Section 6.2.2, there exists a destructive implementation for this algebra that supports all operations in $O(1)$ time.

Since there is a clear correspondence between $\langle \mathrm{Int} \rangle$ and $\langle \mathrm{Int} \rangle_1$, we omit the definitions of functions like $[\![\cdot]\!]$, m, and H for the latter type. However, doing the same for $\#$ would make $\#z$ ambiguous (for $z \in \langle \mathrm{Int} \rangle_1$). We resolve this ambiguity by interpreting $\#z$ as the length of *list* z. To denote the size of the tree associated with z we write $\#[\![z]\!]$.

To obtain a definition of SH, we simply write the definitions for top-down skew heaps in terms of type $\langle \mathrm{Int} \rangle_1$. This gives for S:

$$S = \{z \mid z \in \langle \mathrm{Int} \rangle_1 \ \wedge \ H.z\},$$

and for the priority queue operations:

$$\begin{array}{lcl}
\mathsf{empty} & \doteq & [\,] \\
\mathsf{isempty}.z & \doteq & z = [\,] \\
\mathsf{single}.a & \doteq & [\langle [\,], a \rangle] \\
\mathsf{min}.(\langle t, a \rangle \vdash u) & \doteq & a \\
\mathsf{union}.x.y & \doteq & x \between y \\
\mathsf{delmin}.(\langle t, a \rangle \vdash u) & \doteq & t \between u,
\end{array}$$

where $\between$ ("bottom-up meld") is specified in the same way as $\bowtie$: $\between \in S \times S \to S$ and $[\![x \between y]\!] = [\![x]\!] \oplus [\![y]\!]$. We have introduced $\between$ to distinguish top-down and bottom-up implementations of melding in the sequel.

9.4.1 Implementation of $\between$

In this section we construct—in two steps—a bottom-up version of melding which is almost the same as the algorithm described in [30, p.62].

In the first step we turn the (top-down) program for $\bowtie$ into a tail-recursive (bottom-up) program. This transformation bears resemblance to the transformation prescribed by the following variation on well-known "tail-recursion theorems" from functional programming (see, e.g., [17, Theorem 6.2.0]).

Theorem 9.5
If function g is of the form

$$\begin{array}{lcl}
g.[\,] & = & X \\
g.(a \vdash s) & = & G.a.(g.s)
\end{array}$$

and function h is defined as

$$\begin{array}{lcl}
h.[\,].x & = & x \\
h.(s \dashv a).x & = & h.s.(G.a.x),
\end{array}$$

then

$$h.s.X = g.s$$

for all lists s.
□

In this theorem, function g is linearly recursive whereas function h is tail recursive.

In terms of type $\langle \mathrm{Int} \rangle_1$, the program for $\bowtie$ is:

$$
\begin{array}{rcllll}
[\,] & \bowtie & [\,] & = & [\,] & \\
(\langle t,a\rangle \vdash u) & \bowtie & y & \doteq & \langle u \bowtie y, a\rangle \vdash t & , a \le m.y \\
x & \bowtie & (\langle t,a\rangle \vdash u) & \doteq & \langle u \bowtie x, a\rangle \vdash t & , a \le m.x.
\end{array}
$$

As in Theorem 9.5, we introduce a function F, say, with one more parameter than $\bowtie$, and we use a recursion pattern based on $\dashv$ instead of $\vdash$ to define F. Then $x \bowtie y = F.x.y.[\,]$ (see after Lemma 9.6), where

$$
\begin{array}{lcll}
F.[\,].[\,].z & = & z & \\
F.(u \dashv \langle t,a\rangle).y.z & \doteq & F.u.y.(\langle z,a\rangle \vdash t) & , a \ge M.y \\
F.x.(u \dashv \langle t,a\rangle).z & \doteq & F.u.x.(\langle z,a\rangle \vdash t) & , a \ge M.x,
\end{array}
$$

with M defined by

$$
\begin{array}{lcl}
M.[\,] & = & -\infty \\
M.(u \dashv \langle t,a\rangle) & = & a.
\end{array}
$$

(Note that, in general, $M.z$ differs from the maximum of $[\![z]\!]$, for heap z.) The relation between F and $\bowtie$ is expressed in Lemma 9.6, in which we use the following characterization of H for $\langle \mathrm{Int}\rangle_1$-trees:

$$
(15)\quad
\begin{array}{lcl}
H.[\,] & \equiv & \text{true} \\
H.(u \dashv \langle t,a\rangle) & \equiv & H.u \ \wedge\ M.u \le a \ \wedge\ a \le m.t \ \wedge\ H.t.
\end{array}
$$

Lemma 9.6
For all *heaps* x, y, v, and w satisfying $M.x \max M.y \le m.v \min m.w$:

$$F.x.y.(v \bowtie w) = (x +\!\!+ v) \bowtie (y +\!\!+ w).$$

Proof Note that the premiss implies that $x +\!\!+ v$ and $y +\!\!+ w$ are heaps. The proof proceeds by induction on x and y, using the following exhaustive case analysis.

Case $x = [\,] \ \wedge\ y = [\,]$. By the definition of F, $F.[\,].[\,].(v \bowtie w) = v \bowtie w$.

Case $x = u \dashv \langle t,a\rangle \ \wedge\ a \ge M.y$. Then $M.x \max M.y = a$, and for all heaps v and w satisfying $a \le m.v \ \wedge\ a \le m.w$ we have:

$$
\begin{array}{cl}
 & F.(u \dashv \langle t,a\rangle).y.(v \bowtie w) \\
= & \{ \text{ definition of } F \ (a \ge M.y) \ \} \\
 & F.u.y.(\langle v \bowtie w, a\rangle \vdash t) \\
= & \{ \text{ definition of } \bowtie \ (a \le m.w) \ \} \\
 & F.u.y.((\langle t,a\rangle \vdash v) \bowtie w) \\
= & \{ \text{ induction hypothesis } (M.u \max M.y \le a \min m.w \text{, see below}) \ \} \\
 & (u +\!\!+ (\langle t,a\rangle \vdash v)) \bowtie (y +\!\!+ w) \\
= & \{ \text{ list calculus } \} \\
 & ((u \dashv \langle t,a\rangle) +\!\!+ v) \bowtie (y +\!\!+ w).
\end{array}
$$

Since x is a heap, we have $H.u \ \wedge \ M.u \leq a \ \wedge \ a \leq m.t \ \wedge \ H.t$ (cf. (15)). In conjunction with $a \leq m.v \ \wedge \ H.v$, this implies $H.(\langle t, a\rangle \vdash v)$.

Case $y = u \dashv \langle t, a\rangle \ \wedge \ a \geq M.x.$ This case follows by symmetry.
□

Instantiation of this lemma with $v = [\,]$ and $w = [\,]$ gives $F.x.y.[\,] = x \bowtie y$. Furthermore, we have $\mathcal{N}[F](x, y, [\,]) = \mathcal{N}[\bowtie](x, y)$, where $\mathcal{N}$ denotes the cost measure for the above programs. So, the introduction of F does not give an improvement. However, we are now ready for our second step in which we extend the base case for $F.x.y.z$ from $x = [\,] \ \wedge \ y = [\,]$ to $x = [\,] \ \vee \ y = [\,]$. As a result, we will obtain a program for ⧖ with $O(1)$ amortized costs, whereas the same extension applied to the program for $\bowtie$ gives no essential reduction (see Section 9.2.6).

We observe that $M.x \max M.y \leq m.z$ holds as precondition for all (recursive) applications $F.x.y.z$ generated during the evaluation of $F.x.y.[\,]$. For $x = [\,]$, this is equivalent to $M.y \leq m.z$, and consequently, $y +\!\!+ z$ is a heap. Since we assume that $+\!\!+$ takes $O(1)$ time for $\langle \mathrm{Int}\rangle_1$-trees, we simply take this for $F.[\,].y.z$. By symmetry, we take $F.x.[\,].z = x +\!\!+ z$, and we obtain as program for ⧖:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{x \mathbin{⧖} y = F.x.y.[\,]} \\
\quad |[\ F.[\,].y.z & = & y +\!\!+ z & \\
\quad\ \ F.(u \dashv \langle t, a\rangle).y.z & \doteq & F.u.y.(\langle z, a\rangle \vdash t) & , a \geq M.y \ \wedge \ y \neq [\,] \\
\quad\ \ F.x.[\,].z & = & x +\!\!+ z & \\
\quad\ \ F.x.(u \dashv \langle t, a\rangle).z & \doteq & F.u.x.(\langle z, a\rangle \vdash t) & , a \geq M.x \ \wedge \ x \neq [\,] \\
\quad]|. & & &
\end{array}
$$

In this program, M is used for nonempty trees only.

Compared to the program for $\bowtie$, the program for ⧖ is more intricate and so will be its analysis. In a first, exploratory analysis we will strive for an $O(1)$ bound for the amortized costs of ⧖. Unfortunately, this results in a linear bound for delmin, which is not what we are after. In Section 9.4.3 we will therefore combine the result of the next section with our approach to top-down skew heaps so as to obtain a logarithmic bound for delmin.

9.4.2 Bottom-up analysis of ⧖

As before, we use $\mathcal{N}$ to denote the cost measure defined by the dots in the above program. This measure exactly counts the number of comparisons. The amortized costs for ⧖ are defined as usual.

$$\mathcal{A}[⧖](x, y) = \mathcal{N}[⧖](x, y) + \Phi.(x \mathbin{⧖} y) - \Phi.x - \Phi.y.$$

However, the pattern of the definition of the amortized costs for F, given by

$$\mathcal{A}[F](x, y, z) = \mathcal{N}[F](x, y, z) + \Phi.(F.x.y.z) - \Phi.x - \Phi.y,$$

deviates from the general pattern used thus far. We have omitted "$-\Phi.z$", for otherwise it is impossible—as far as we know—to define Φ such that $\mathcal{A}[F]$ is $O(1)$. As a possible explanation, we remark that we use F in such a way that z is merely an accumulation parameter, whereas x and y are input parameters and $F.x.y.z$ is the result.

From the program for ⧖, it is immediate that $\mathcal{A}[⧖](x,y) = \mathcal{A}[F](x,y,[\,])$. Our goal is now to define Φ such that $\mathcal{A}[F]$ is $O(1)$. To achieve this, we change the above program for F a little bit because an analysis of this program along the same lines as below leads to a potential function for which $\mathcal{A}[F]$ is not $O(1)$. We shall therefore study the following equivalent program for F, which we have obtained by singling out the cases in which one of the first two arguments is a singleton list:

$$\begin{array}{lcll} F.[\,].y.[\,] & = & y & \\ F.[\langle t,a\rangle].y.z & \doteq & y \mathbin{+\!\!+} (\langle z,a\rangle \vdash t) & ,a \geq M.y \;\wedge\; y \neq [\,] \\ F.(u \dashv \langle t,a\rangle).y.z & \doteq & F.u.y.(\langle z,a\rangle \vdash t) & ,a \geq M.y \;\wedge\; y \neq [\,] \;\wedge\; u \neq [\,]. \end{array}$$

The symmetric counterparts of these alternatives are omitted.

We first consider the recursive case for F, since this puts the strongest requirements on Φ ($u \neq [\,]$):

$$\begin{array}{ll} & \mathcal{A}[F](u \dashv \langle t,a\rangle, y, z) \\ = & \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{N}[F](u \dashv \langle t,a\rangle, y, z) + \Phi.(F.(u \dashv \langle t,a\rangle).y.z) - \Phi.(u \dashv \langle t,a\rangle) - \Phi.y \\ = & \{ \text{ definitions of } F \text{ and } \mathcal{N} \ \} \\ & 1 + \mathcal{N}[F](u, y, \langle z,a\rangle \vdash t) + \Phi.(F.u.y.(\langle z,a\rangle \vdash t)) - \Phi.(u \dashv \langle t,a\rangle) - \Phi.y \\ = & \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{A}[F](u, y, \langle z,a\rangle \vdash t) + 1 + \Phi.u - \Phi.(u \dashv \langle t,a\rangle). \end{array}$$

In order to obtain an $O(1)$ bound for $\mathcal{A}[F]$ we define Φ such that $1 + \Phi.u - \Phi.(u \dashv \langle t,a\rangle) \leq 0$. To keep Φ as small as possible, we even take

(16) $\Phi.(u \dashv \langle t,a\rangle) = \Phi.u + 1$,

for $u \neq [\,]$.

Taking $\Phi.[\,] = 0$ gives $\mathcal{A}[F]([\,],y,[\,]) = 0$, and for the remaining nonrecursive case we derive ($y \neq [\,]$):

$$\begin{array}{ll} & \mathcal{A}[F]([\langle t,a\rangle], y, z) \\ = & \{ \text{ definition of } \mathcal{A} \ \} \\ & \mathcal{N}[F]([\langle t,a\rangle], y, z) + \Phi.(F.[\langle t,a\rangle].y.z) - \Phi.[\langle t,a\rangle] - \Phi.y \\ = & \{ \text{ definition of } F \text{ and } \mathcal{N} \ \} \\ & 1 + \Phi.(y \mathbin{+\!\!+} (\langle z,a\rangle \vdash t)) - \Phi.[\langle t,a\rangle] - \Phi.y \\ = & \{ \text{ (16) implies } \Phi.(y \mathbin{+\!\!+} (\langle z,a\rangle \vdash t)) = \Phi.y + 1 + \#t, \text{ for } y \neq [\,] \ \} \end{array}$$

$$
\begin{array}{cl}
 & 1+1+\#t-\Phi.[\langle t,a\rangle] \\
= & \{ \text{ take } \Phi.[\langle t,a\rangle] = \#t+1 \text{ (see definition } \Phi \text{ below) } \} \\
 & 1.
\end{array}
$$

Hence, $\mathcal{A}[F](x,y,z) \leq 1$, if we define Φ as follows:

$$
\begin{array}{lcl}
\Phi.[\,] & = & 0 \\
\Phi.(\langle t,a\rangle \vdash u) & = & \#t+1+\#u,
\end{array}
$$

and, consequently, $\mathcal{A}[\bowtie](x,y) \leq 1$. (We leave it to the reader to ascertain that a similar analysis of the program derived in Section 9.4.1 leads to potential $\Phi.z = \#z$, for which $\mathcal{A}[F]$ is not $O(1)$.)

Remark 9.7
The above derived potential Φ is reminiscent of the potential used by Sleator and Tarjan in their analysis of bottom-up skew heaps. In their terminology, $\Phi.x$ equals the sum of the lengths of the major path and the minor path of x. (The *major path* of x is the rightmost path of x, and the *minor path* of x is the rightmost path of x's left subtree)
□

With $\mathcal{T}$ as defined for top-down skew heaps, we now have $O(1)$ bounds for all priority queue operations, except that for delmin:

$$
\begin{array}{cl}
 & \mathcal{A}[\mathsf{delmin}](\langle t,a\rangle \vdash u) \\
= & \{ \text{ definition of } \mathcal{A} \ \} \\
 & \mathcal{T}[\mathsf{delmin}](\langle t,a\rangle \vdash u) + \Phi.(\mathsf{delmin}.(\langle t,a\rangle \vdash u)) - \Phi.(\langle t,a\rangle \vdash u) \\
= & \{ \text{ definitions of } \mathsf{delmin} \text{ and } \Phi \ \} \\
 & 1+\mathcal{N}[\bowtie](t,u) + \Phi.(t \bowtie u) - \#t - 1 - \#u \\
= & \{ \text{ definition of } \mathcal{A} \ \} \\
 & \mathcal{A}[\bowtie](t,u) + \Phi.t - \#t + \Phi.u - \#u \\
\leq & \{ \text{ bound for } \mathcal{A}[\bowtie] \ \} \\
 & 1+\Phi.t - \#t + \Phi.u - \#u.
\end{array}
$$

Unfortunately, $\Phi.z - \#z$ may be linear in the size of z, and therefore we can only conclude that $\mathcal{A}[\mathsf{delmin}](z)$ is linear in the size of z.

9.4.3 First top-down analysis of $\bowtie$

To obtain a logarithmic bound for the amortized costs of delmin we analyze yet another (equivalent) program for $\bowtie$. Inspired by the potential function of Sleator and Tarjan for bottom-up skew heaps, we take a potential function of the following form:

$$
\begin{array}{lcl}
\Phi.\langle\,\rangle & = & 0 \\
\Phi.\langle t,a,u\rangle & = & \Psi.\langle t,a,u\rangle + \Pi.t + \Pi.\langle t,a,u\rangle,
\end{array}
$$

with

$$
\begin{array}{lcl}
\Psi.\langle\,\rangle & = & 0 \\
\Psi.\langle t,a,u\rangle & = & \Psi.t + \psi.t.u + \Psi.u,
\end{array}
$$

and

$$
\begin{array}{lcl}
\Pi.\langle\,\rangle & = & 0 \\
\Pi.\langle t,a,u\rangle & = & \pi.t.u + \Pi.u.
\end{array}
$$

The form of Φ may be considered as a mixture of the form of the potential function for top-down skew heaps (Ψ) and the form of the potential function found in the previous section (Π, twice).

Remark 9.8 (see Remark 9.7)
In the above form for Φ, the nodes on the major path and those on the minor path are treated as special—as is also done in [30]. The term $\Pi.\langle t,a,u\rangle$ corresponds to the major path and $\Pi.t$ to the minor path.
□

Our analysis of top-down skew heaps proceeded smoothly because the recursion pattern in the program of $\bowtie$ matched the recursive definition of the potential so well. In view of the above definition of Φ we therefore decide to construct a top-down *simulation* (read "algorithmic refinement") of ⧖. This simulation is based on the observation that $x ⧖ y = t \mathbin{+\!\!+} (u \bowtie y)$ if $m.x \leq m.y$, where t and u satisfy $t \mathbin{+\!\!+} u = x$ and $M.t \leq m.y \leq m.u$. Phrased differently, $x ⧖ y$ can be computed in two stages: first compute t and then compute $u \bowtie y$. This is encoded in the following program:

$$
\begin{array}{rcll}
\langle\,\rangle ⧖ \langle\,\rangle & = & \langle\,\rangle & \\
\langle t,a,u\rangle ⧖ y & = & \langle t,a,u \lhd y\rangle & ,a \leq m.y \\
x ⧖ \langle t,a,u\rangle & = & \langle t,a,u \lhd x\rangle & ,a \leq m.x,
\end{array}
$$

where

$$
\begin{array}{rcll}
\langle\,\rangle \lhd \langle\,\rangle & = & \langle\,\rangle & \\
\langle t,a,u\rangle \lhd y & = & \langle t,a,u \lhd y\rangle & ,a \leq m.y \\
x \lhd \langle t,a,u\rangle & \doteq & \langle u \bowtie x,a,t\rangle & ,a \leq m.x,
\end{array}
$$

and where

$$
\begin{array}{rcll}
\langle\,\rangle \bowtie \langle\,\rangle & = & \langle\,\rangle & \\
\langle t,a,u\rangle \bowtie y & \doteq & \langle u \bowtie y,a,t\rangle & ,a \leq m.y \\
x \bowtie \langle t,a,u\rangle & \doteq & \langle u \bowtie x,a,t\rangle & ,a \leq m.x.
\end{array}
$$

Note that some of the above definitions are longer than necessary (e.g., the definition of ⧖). The uniformity of the above definitions, however, turns out to be the basis for a smooth amortized analysis. The cost measure $\mathcal{N}$ for the above program has been chosen such that it coincides with the cost measure for the bottom-up program for $\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}$.

To facilitate the calculations below, we introduce functions Θ and θ, defined by $\Theta.z = \Psi.z + \Pi.z$ and $\theta.t.u = \psi.t.u + \pi.t.u$. Then

$$\begin{array}{lcl} \Theta.\langle\,\rangle & = & 0 \\ \Theta.\langle t, a, u\rangle & = & \Psi.t + \theta.t.u + \Theta.u, \end{array}$$

and

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Theta.t + \theta.t.u + \Theta.u, \end{array}$$

as a consequence of which Π and π become superfluous. Recall that the amortized costs of $\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}$ are defined as

$$\mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}](x,y) = \mathcal{N}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}](x,y) + \Phi.(x \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} y) - \Phi.x - \Phi.y.$$

In what follows, we will derive suitable definitions for $\mathcal{A}[\triangleleft]$ and $\mathcal{A}[\bowtie]$.

Six requirements on θ and ψ

In order that Φ is nonnegative and small, we require that θ and ψ satisfy

$$\begin{array}{ll} \text{(i)} & 0 \le \theta.t.u \\ \text{(ii)} & 0 \le \psi.t.u, \end{array}$$

and also that θ and ψ are small. In the sequel we will accumulate further requirements on θ and ψ that ensure that $\mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}]$ is $O(1)$ and $\mathcal{A}[\mathsf{delmin}]$ is logarithmic.

For $\mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}]$ we have $\mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}](\langle\,\rangle,\langle\,\rangle) = 0$, and in case $x \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} y = \langle t, a, u \triangleleft y\rangle$, we derive:

$$\begin{array}{cl} & \mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}](x,y) \\ = & \{\ \text{definition of } \mathcal{A}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}]\ \} \\ & \mathcal{N}[\mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}](x,y) + \Phi.(x \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} y) - \Phi.x - \Phi.y \\ = & \{\ \text{definitions of } \mathcal{N} \text{ and } \Phi\ (x = \langle t,a,u\rangle)\ \} \\ & \mathcal{N}[\triangleleft](u,y) + \Theta.t + \theta.t.(u \triangleleft y) + \Theta.(u \triangleleft y) - \Theta.t - \theta.t.u - \Theta.u - \Phi.y \\ = & \{\ \text{definition of } \mathcal{A}[\triangleleft] \text{ (see below)}\ \} \\ & \mathcal{A}[\triangleleft](u,y) + \theta.t.(u \triangleleft y) - \theta.t.u. \end{array}$$

The appropriate definition for the amortized cost of $\triangleleft$ is thus given by

$$\mathcal{A}[\triangleleft](x,y) = \mathcal{N}[\triangleleft](x,y) + \Theta.(x \triangleleft y) - \Theta.x - \Phi.y.$$

Simply turning the handle gives $\mathcal{A}[\triangleleft](\langle\,\rangle,\langle\,\rangle) = 0$, and for the case $x \triangleleft y = \langle t, a, u \triangleleft y\rangle$:

$$\mathcal{A}[\triangleleft](x,y) = \mathcal{A}[\triangleleft](u,y) + \theta.t.(u \triangleleft y) - \theta.t.u.$$

In case $x \triangleleft y = \langle u \bowtie x, a, t\rangle$, we derive the definition of $\mathcal{A}[\bowtie]$ as follows:

$$\begin{array}{cl}
 & \mathcal{A}[\triangleleft](x,y) \\
= & \{ \text{ definition of } \mathcal{A}[\triangleleft] \text{ } \} \\
 & \mathcal{N}[\triangleleft](x,y) + \Theta.(x \triangleleft y) - \Theta.x - \Phi.y \\
= & \{ \text{ definitions of } \mathcal{N} \text{ and } \Theta \text{ } (y = \langle t,a,u\rangle) \text{ } \} \\
 & 1 + \mathcal{N}[\bowtie](u,x) + \Psi.(u \bowtie x) + \theta.(u \bowtie x).t + \Theta.t - \Theta.x - \Theta.t - \theta.t.u - \Theta.u \\
= & \{ \text{ definition of } \mathcal{A}[\bowtie] \text{ (see below) } \} \\
 & \mathcal{A}[\bowtie](u,x) + 1 + \theta.(u \bowtie x).t - \theta.t.u,
\end{array}$$

where the amortized cost of $\bowtie$ is defined by

$$\mathcal{A}[\bowtie](x,y) = \mathcal{N}[\bowtie](x,y) + \Psi.(x \bowtie y) - \Theta.x - \Theta.y.$$

Omitting the calculations for $\mathcal{A}[\bowtie]$, we thus obtain as recurrence relations:

$$\begin{array}{lcll}
\mathcal{A}[⧖](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[⧖](\langle t,a,u\rangle,y) & = & \mathcal{A}[\triangleleft](u,y) + \theta.t.(u \triangleleft y) - \theta.t.u & , a \le m.y \\
\mathcal{A}[⧖](x,\langle t,a,u\rangle) & = & \mathcal{A}[\triangleleft](u,x) + \theta.t.(u \triangleleft x) - \theta.t.u & , a \le m.x \\
 & & & \\
\mathcal{A}[\triangleleft](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[\triangleleft](\langle t,a,u\rangle,y) & = & \mathcal{A}[\triangleleft](u,y) + \theta.t.(u \triangleleft y) - \theta.t.u & , a \le m.y \\
\mathcal{A}[\triangleleft](x,\langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x) + 1 + \theta.(u \bowtie x).t - \theta.t.u & , a \le m.x \\
 & & & \\
\mathcal{A}[\bowtie](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[\bowtie](\langle t,a,u\rangle,y) & = & \mathcal{A}[\bowtie](u,y) + 1 + \psi.(u \bowtie y).t - \theta.t.u & , a \le m.y \\
\mathcal{A}[\bowtie](x,\langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x) + 1 + \psi.(u \bowtie x).t - \theta.t.u & , a \le m.x.
\end{array}$$

Solving $\mathcal{A}[⧖]$ from this set of recurrence relations leads to at most one unfolding of the last alternative of $\mathcal{A}[\triangleleft]$. In order that $\mathcal{A}[⧖]$ is $O(1)$, it is therefore sufficient to require that the term $1 + \theta.(u \bowtie x).t - \theta.t.u$ is bounded, while the other terms are required to be nonpositive. Thus, we require that, for some constant γ and for all t, u, x and y:

(iii) $\theta.t.(u \triangleleft y) \le \theta.t.u$

(iv) $\theta.(u \bowtie x).t \le \theta.t.u + \gamma$

(v) $1 + \psi.(u \bowtie y).t \le \theta.t.u.$

If we succeed in fulfilling these five requirements, we may conclude from the above set of recurrence relations that $\mathcal{A}[⧖](x,y) \le \gamma + 1$. These requirements are, of course, easily met when we choose $\psi.t.u = 0$ and $\theta.t.u = 1$; then Φ coincides with the potential function derived in Section 9.4.2. There is, however, an additional requirement to ensure a logarithmic bound for delmin:

$$
\begin{array}{ll}
 & \mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle) \\
= & \{ \text{ definition of } \mathcal{A} \text{ } \} \\
 & \mathcal{T}[\mathsf{delmin}](\langle t,a,u\rangle) + \Phi.(\mathsf{delmin}.\langle t,a,u\rangle) - \Phi.\langle t,a,u\rangle \\
= & \{ \text{ definitions of } \mathcal{T}, \mathsf{delmin}, \text{ and } \Phi \text{ } \} \\
 & 1 + \mathcal{N}[\,\bowtie\,](t,u) + \Phi.(t \bowtie u) - \Theta.t - \theta.t.u - \Theta.u \\
= & \{ \text{ definition of } \mathcal{A} \text{ } \} \\
 & 1 + \mathcal{A}[\,\bowtie\,](t,u) + \Phi.t + \Phi.u - \Theta.t - \theta.t.u - \Theta.u \\
\leq & \{ \text{ above bound for } \mathcal{A}[\,\bowtie\,];\ \theta \text{ nonnegative (requirement (i)) } \} \\
 & \gamma + 2 + \Phi.t - \Theta.t + \Phi.u - \Theta.u \\
\leq & \{ \ \Phi.z - \Theta.z \leq \log_\alpha \sharp z, \text{ see below } \} \\
 & \gamma + 2 + \log_\alpha \sharp t + \log_\alpha \sharp u \\
\leq & \{ \ 4mn \leq (m+n)^2 \ \} \\
 & \gamma + 2 + 2\log_\alpha(\sharp t + \sharp u) - \log_\alpha 4.
\end{array}
$$

We choose ψ and θ such that $\Phi.z - \Theta.z \leq \log_\alpha \sharp z$, for some constant α, $\alpha > 1$. Evidently, this is true for $z = \langle\ \rangle$. For $z = \langle t,a,u\rangle$, we have that $\Phi.z - \Theta.z = \Pi.t$, and therefore we want Π to satisfy

(17) $\Pi.z \leq \log_\alpha \sharp z$,

for all z. To obtain a requirement in terms of ψ and θ, we first translate this requirement on Π into a requirement on π. We apply induction on z.

Case $z = \langle\ \rangle$. Trivial, since $\sharp\langle\ \rangle = 1$.

Case $z = \langle t,a,u\rangle$.

$$
\begin{array}{ll}
 & \Pi.\langle t,a,u\rangle \\
= & \{ \text{ definition of } \Pi \text{ } \} \\
 & \pi.t.u + \Pi.u \\
= & \{ \text{ induction hypothesis (17) } \} \\
 & \pi.t.u + \log_\alpha \sharp u \\
\leq & \{ \text{ see upper bound for } \pi \text{ below } \} \\
 & \log_\alpha \sharp\langle t,a,u\rangle.
\end{array}
$$

Hence, we require that $\pi.t.u$ is bounded from above by $\log_\alpha \frac{\sharp t + \sharp u}{\sharp u}$, or, in terms of θ and ψ:

(vi) $\theta.t.u - \psi.t.u \leq \log_\alpha \frac{\sharp t + \sharp u}{\sharp u}$.

This concludes the derivation of six requirements on ψ and θ, which are sufficient to ensure that the amortized cost of delmin is logarithmic and that the amortized costs of the other priority queue operations are $O(1)$. In addition, we obtain a nonnegative and small potential Φ, provided ψ and θ are small.

Construction of θ and ψ

First of all, we remark that (i) is redundant, since it follows from (ii) and (v). Next, we do away with $\log_\alpha$ in requirement (vi) by introducing functions f and g and defining

$$\begin{array}{lcl} \theta.t.u & = & \log_\alpha f.\sharp t.\sharp u \\ \psi.t.u & = & \log_\alpha g.\sharp t.\sharp u. \end{array}$$

Apart from the removal of $\log_\alpha$, the introduction of f and g embodies the decision to let $\theta.t.u$ and $\psi.t.u$ depend on the sizes of t and u only—this simplifies the formulation of the requirements and for top-down skew heaps we also found that $\varphi.t.u$ depends on the sizes of t and u only. The corresponding requirements on f and g then read ((i) is redundant; constant β replaces α^γ):

$$\begin{array}{cl} \text{(i)} & 1 \leq f.k.l \\ \text{(ii)} & 1 \leq g.k.l \\ \text{(iii)} & f.k.m \leq f.k.l \\ \text{(iv)} & f.m.k \leq \beta\ f.k.l \\ \text{(v)} & \alpha\ g.m.k \leq f.k.l \\ \text{(vi)} & \dfrac{f.k.l}{g.k.l} \leq \dfrac{k+l}{l}, \end{array}$$

for all positive k, l, and m such that $l \leq m$ ($\alpha > 1$ and $\beta > 0$).

In our next step, we get rid of m in requirements (iv) and (v). Firstly, we may replace (iv) by (iv′), with

$$\text{(iv}'\text{)} \quad f.l.k \leq \beta\ f.k.l\ ,$$

because (iv′) is not weaker than (iv) on account of (iii):

$$\begin{array}{ll} & f.m.k \\ \leq & \{\ (\text{iv}')\ \} \\ & \beta\ f.k.m \\ \leq & \{\ (\text{iii})\ \} \\ & \beta\ f.k.l, \end{array}$$

for any k, l, and m with $l \leq m$. On similar grounds, (v) may be replaced by (v′), where

$$\text{(v}'\text{)} \quad \alpha\ g.l.k \leq f.k.l.$$

Now we observe that it is necessary that $\alpha \leq \beta$, since it is required that for all k and l:

$$
\begin{array}{ll}
 & \alpha \\
\leq & \{ \text{(v}'\text{)} \} \\
 & \dfrac{f.k.l}{g.l.k} \\
\leq & \{ \text{(iv}'\text{)} \} \\
 & \beta\,\dfrac{f.l.k}{g.l.k} \\
\leq & \{ \text{(vi)} \} \\
 & \beta\,\dfrac{l+k}{k}.
\end{array}
$$

Since we want α to be as large as possible and β as small as possible, we take $\alpha = \beta$.

As a further simplification we decide to eliminate g from the set of requirements. To this end we define

$$g.k.l = \frac{f.l.k}{\alpha},$$

i.e., we choose equality in (v$'$). This choice for g does not strengthen the requirements on f and α: requirement (vi) becomes $\alpha\,\frac{f.k.l}{f.l.k} \leq \frac{k+l}{l}$, but this already follows from (v$'$) and (vi). Taking $k = l$ in (vi) then gives $\alpha \leq 2$, so we take $\alpha = 2$ (as large as possible). As a result, we obtain the following set of requirements for f, where we have swapped k and l in (III) to let it resemble (IV):

$$
\begin{array}{ll}
\text{(I)} & 2 \leq f.k.l \leq 4 \\
\text{(II)} & f.k.m \leq f.k.l \\
\text{(III)} & f.k.l \leq 2\ f.l.k \\
\text{(IV)} & f.k.l \leq \dfrac{k+l}{2l} f.l.k,
\end{array}
$$

for all positive k, l, and m such that $l \leq m$. In (I) we have added an absolute upper bound for f to make explicit how small we want f to be; we have chosen 4 because $(\forall\, k, l :: 2 \leq f.k.l \leq \delta)$ implies $(\forall\, k, l :: 4l \leq \delta(k+l))$ on account of (IV), hence it is necessary that $\delta \geq 4$.

Requirements (I) through (III) are satisfied if we take $f.k.l = 2$, but then (IV) does not hold if $k < 3l$. To arrive at a solution for f, we distinguish the cases $k \leq l$ and $k \geq l$ in the definition of $f.k.l$. By simply taking $f.k.l = 2$ in case $k \leq l$, we see that (IV) reduces to $2 \leq \frac{k+l}{2l} f.l.k$. To satisfy this requirement we define f such that we have equality in (IV). The resulting f is then given by

$$f.k.l = 2\ \frac{k + k \max l}{k+l}.$$

The reader may now verify that this definition meets all requirements on f.

Remark 9.9
By first taking $f.k.l = 2$ in case $k \geq l$ and subsequently taking equality in (IV) as above, we find:

$$f.k.l = 4 \ \frac{k+l}{k \max l + l}.$$

The choice for f in the text is preferred because it results in a smaller potential.
□

Thus we find (since $g.k.l = f.l.k/2$):

$$\psi.t.u = \log_2 \frac{\sharp t \max \sharp u \ + \ \sharp u}{\sharp t \ + \ \sharp u} \qquad (= \log_2 \tfrac{2\sharp u}{\sharp t + \sharp u} \max 0),$$

and $\theta.t.u = 1 + \psi.u.t$. Since $0 \leq \psi.t.u \leq 1$, potential Φ satisfies $0 \leq \Phi.z \leq 2\#z$.

With this potential, each priority queue operation except delmin has $O(1)$ amortized costs. The amortized costs of the latter operation satisfy

$$\begin{array}{ll} & \mathcal{A}[\mathsf{delmin}](\langle t, a, u\rangle) \\ \leq & \{ \text{ see page 135 } \} \\ & \gamma + 2 + 2\log_\alpha(\sharp t + \sharp u) - \log_\alpha 4 \\ = & \{ \ \alpha = 2, \ \beta = 2, \text{ and } \gamma = \log_\alpha \beta = 1 \ \} \\ & 1 + 2\log_2 \sharp\langle t, a, u\rangle, \end{array}$$

which completes the proof of the following lemma.

Lemma 9.10
Let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Theta.t \ + \ \log_2 \frac{4\sharp t}{\sharp t + \sharp u} \max 1 \ + \ \Theta.u \end{array}$$

with

$$\begin{array}{lcl} \Theta.\langle\,\rangle & = & 0 \\ \Theta.\langle t, a, u\rangle & = & \Psi.t \ + \ \log_2 \frac{4\sharp t}{\sharp t + \sharp u} \max 1 \ + \ \Theta.u \\ \Psi.\langle\,\rangle & = & 0 \\ \Psi.\langle t, a, u\rangle & = & \Psi.t \ + \ \log_2 \frac{2\sharp u}{\sharp t + \sharp u} \max 0 \ + \ \Psi.u. \end{array}$$

Then $0 \leq \Phi.z \leq 2\#z$, and the amortized costs of union and delmin satisfy

$$\begin{array}{lcl} \mathcal{A}[\mathsf{union}](x, y) & \leq & 3 \\ \mathcal{A}[\mathsf{delmin}](z) & \leq & 1 + 2\,\log_2 \sharp z. \end{array}$$

□

Remark 9.11 (see Remark 9.7)
In Sleator and Tarjan's analysis of bottom-up skew heaps, a node with subtrees t and u contributes $\lceil\psi.t.u\rceil + 2\lceil\psi.u.t\rceil$ if it is on the major or minor path, and $\lceil\psi.t.u\rceil$ otherwise. Since $\lceil\psi.t.u\rceil + 2\lceil\psi.u.t\rceil \approx 1+\psi.u.t$, this may be compared to our potential that counts $1+\psi.u.t$ for nodes on the major and minor paths and $\psi.t.u$ for the remaining nodes.
□

9.4.4 Second top-down analysis of ⧖

Although the $O(1)$ bound for union is a nice result, it is somewhat disappointing that the bound for delmin.p is as high as $2\log_2(\#[\![p]\!]+1)$. These bounds imply that at most $2N\log_2 N$ comparisons are needed to sort N numbers by means of bottom-up skew heaps (since this requires N applications of delmin). As shown in Section 9.3, however, we have $1.44N\log_2 N$ as bound when top-down skew heaps are used, and we would like to have a similar bound for bottom-up skew heaps.

To achieve this, we allow logarithmic costs for ⧖. For $\alpha > 1$, we try as bound:

(18) $\mathcal{A}[\text{⧖}](x,y) \leq 1+\log_\alpha(\sharp x+\sharp y)$.

To take advantage of the work done to analyze top-down skew heaps, we take

$$\theta.t.u = 1,$$

and we derive several requirements for the remaining function ψ. This choice for θ considerably simplifies the recurrence relations for the amortized costs in the analysis of ⧖ in the previous section:

$$\begin{array}{lcll}
\mathcal{A}[\text{⧖}](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[\text{⧖}](\langle t,a,u\rangle,y) & = & \mathcal{A}[\lhd](u,y) & ,\ a\leq m.y \\
\mathcal{A}[\text{⧖}](x,\langle t,a,u\rangle) & = & \mathcal{A}[\lhd](u,x) & ,\ a\leq m.x \\
 & & & \\
\mathcal{A}[\lhd](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[\lhd](\langle t,a,u\rangle,y) & = & \mathcal{A}[\lhd](u,y) & ,\ a\leq m.y \\
\mathcal{A}[\lhd](x,\langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x)+1 & ,\ a\leq m.x \\
 & & & \\
\mathcal{A}[\bowtie](\langle\,\rangle,\langle\,\rangle) & = & 0 & \\
\mathcal{A}[\bowtie](\langle t,a,u\rangle,y) & = & \mathcal{A}[\bowtie](u,y)+\psi.(u\bowtie y).t & ,\ a\leq m.y \\
\mathcal{A}[\bowtie](x,\langle t,a,u\rangle) & = & \mathcal{A}[\bowtie](u,x)+\psi.(u\bowtie x).t & ,\ a\leq m.x.
\end{array}$$

To solve (18), we first observe—as in our analysis of top-down skew heaps (cf. (9))—that

$$\mathcal{A}[\bowtie](x,y) = \Gamma.(x \bowtie y),$$

where

$$\begin{array}{lcl} \Gamma.\langle\,\rangle & = & 0 \\ \Gamma.\langle t,a,u\rangle & = & \Gamma.t + \psi.t.u. \end{array}$$

Since $\mathcal{A}[\,\between\,](x,y) \le 1 + \mathcal{A}[\bowtie](x,y)$, we see that (18) is satisfied when

$$\Gamma.z \le \log_\alpha \sharp z,$$

which is in turn satisfied when

$$\text{(i)} \quad \psi.t.u \le \log_\alpha \tfrac{\sharp t + \sharp u}{\sharp t}.$$

To obtain logarithmic amortized costs for delmin, we want in addition—cf. (17)—that

(19) $\Pi.z \le \log_\beta \sharp z,$

for some $\beta > 1$, which leads to the following requirement on ψ:

$$\text{(ii)} \quad 1 - \psi.t.u \le \log_\beta \tfrac{\sharp t + \sharp u}{\sharp u}.$$

As shown in Section 9.2.4, the existence of a function ψ satisfying (i) and (ii) (cf. (13)) is guaranteed provided (cf. (14))

$$\beta \le \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^\varepsilon},$$

where $\varepsilon = \log_\alpha \beta$. Moreover, we have shown that it is possible to define ψ such that $0 \le \psi.t.u \le 1$. Since $\theta.t.u = 1$, we thus obtain a potential Φ satisfying $0 \le \Phi.z \le \#z$.

To obtain a small bound for delmin, we derive

$$\begin{array}{cl} & \mathcal{A}[\mathsf{delmin}](\langle t,a,u\rangle) \\ = & \{ \text{ see page 135 } \} \\ & 1 + \mathcal{A}[\,\between\,](t,u) + \Phi.t + \Phi.u - \Theta.t - \theta.t.u - \Theta.u \\ = & \{ \ \theta.t.u = 1 \ \} \\ & \mathcal{A}[\,\between\,](t,u) + \Phi.t - \Theta.t + \Phi.u - \Theta.u \\ \le & \{ \text{ (18); (19), using that } \Phi.z - \Theta.z = \Pi.(l.z) \text{ for nonempty } z \ \} \\ & 1 + \log_\alpha(\sharp t + \sharp u) + \log_\beta \sharp t + \log_\beta \sharp u \\ \le & \{ \ \log_\alpha x = \varepsilon \log_\beta x \ \} \\ & 1 + (\varepsilon + 2)\log_\beta(\sharp t + \sharp u). \end{array}$$

This proves the following lemma.

Lemma 9.12
Let $\varepsilon > 0$ and let potential Φ be defined by

$$\begin{array}{lcl} \Phi.\langle\ \rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Theta.t + 1 + \Theta.u \end{array}$$

with

$$\begin{array}{lcl} \Theta.\langle\ \rangle & = & 0 \\ \Theta.\langle t,a,u\rangle & = & \Psi.t + 1 + \Theta.u \\ \Psi.\langle\ \rangle & = & 0 \\ \Psi.\langle t,a,u\rangle & = & \Psi.t \ + \ \log_\beta \frac{\beta\sharp u}{\sharp t+\sharp u} \max 0 \ + \ \Psi.u. \end{array}$$

Then $0 \le \Phi.z \le \#z$, and the amortized costs of union and delmin satisfy

$$\begin{array}{lcl} \mathcal{A}[\mathsf{union}](x,y) & \le & \log_\alpha(\sharp x + \sharp y) \ + \ O(1) \\ \mathcal{A}[\mathsf{delmin}](z) & \le & (\varepsilon+2)\ \log_\beta \sharp z \ + \ O(1), \end{array}$$

where $\alpha = \beta^{1/\varepsilon}$ and $\beta = \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^\varepsilon}$.
□

For each ratio of the number of union's to the number of delmin's, a suitable choice for ε in Lemma 9.12 can be made. For example, if this ratio is 1, we minimize $\frac{2\varepsilon+2}{\log_2\beta}$ over all $\varepsilon > 0$. This yields 2 as minimal value at $\varepsilon = 1$. In this way, we get $2N\log_2 N$ as bound on the number of comparisons needed by program $sort1$ to sort N numbers. Note that this bound for $sort1$ also follows from Lemma 9.10.

To obtain a good bound for $PQsort$, we minimize the bound for delmin. As shown in Section 9.2.4, minimizing $\frac{\varepsilon+2}{\log_2\beta}$ yields

$$\mathcal{A}[\mathsf{delmin}](z) \le \log_\phi \sharp z \ + \ O(1)$$

for $\varepsilon = \phi$ (≈ 1.618) and $\beta = \phi^{\phi+2}$ (≈ 5.70). The corresponding bound for union is

$$\mathcal{A}[\mathsf{union}](x,y) \le \log_\alpha(\sharp x + \sharp y) \ + \ O(1),$$

with $\alpha = \phi^{2\phi-1}$ (≈ 2.93). Compared to the bounds for top-down skew heaps, this bound for union is lower whereas the bounds for delmin are equal. So, according to these bounds, bottom-up skew heaps are at least as good as top-down skew heaps.

9.4.5 Results for priority queue operations

With $\mathcal{T}$ as defined for top-down skew heaps, we have derived in Section 9.4.3 the following bounds for bottom-up skew heaps (cf. Lemma 9.10):

$$\begin{array}{lll}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & = & 1 \text{ comparison} \\
\mathcal{A}[\mathsf{union}](p,q) & \leq & 2 \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \leq & 2 \log_2(1 + \#[\![p]\!]) \text{ comparisons,}
\end{array}$$

for a potential function satisfying $0 \leq \Phi.p \leq 2\#[\![p]\!]$. These bounds for union and delmin improve the bounds obtained by Sleator and Tarjan by a factor 2.

The analysis in Section 9.4.4 yields as bounds (instantiate Lemma 9.12 with $\varepsilon = \phi$):

$$\begin{array}{lll}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1) \\
\mathcal{A}[\mathsf{single}](a) & = & 1 \text{ comparison} \\
\mathcal{A}[\mathsf{union}](p,q) & \leq & 0.64 \log_2(2 + \#[\![p]\!] + \#[\![q]\!]) \text{ comparisons} \\
\mathcal{A}[\mathsf{isempty}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{min}](p) & \text{is} & O(1) \\
\mathcal{A}[\mathsf{delmin}](p) & \leq & 1.44 \log_2(1 + \#[\![p]\!]) \text{ comparisons,}
\end{array}$$

for a potential function satisfying $0 \leq \Phi.p \leq \#[\![p]\!]$. These bounds for union and delmin cannot be compared with the bounds obtained by Sleator and Tarjan, since the bound for union is not $O(1)$ but the bound for delmin is better.

9.5 Pointer implementations

Top-down skew heaps

In the implementation of top-down skew heaps we have used algebra

$$(\,\langle\text{Int}\rangle \mid \langle\,\rangle, (=\langle\,\rangle), \langle\cdot,\cdot,\cdot\rangle, l, m, r\,).$$

A purely-functional implementation of this algebra, which supports all operations in $O(1)$ time, is described in Section 4.3. The only disadvantage of this implementation is that the recycling of cells is not trivial. This can be made trivial by disposing cells explicitly, like we did in the destructive implementation of stacks in Chapter 4. This yields a destructive implementation of top-down skew heaps.

A next step is to recycle cells in-line to obtain an *in-situ* version of top-down skew heaps. That is, we avoid a call to new by using the cell which is disposed by a nearby call to dispose; e.g., for the destructive implementation of stacks, $\mathsf{cons}.a.(\mathsf{tl}.p)$ is equivalent to $p^\wedge.a := a$; $\mathrm{return}(p)$. This gives the following destructive implementation of top-down skew heaps at pointer level:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{single}.a & = & |[\ h{:}B;\ \mathrm{new}(h);\ h^\wedge := \langle \mathrm{nil}, a, \mathrm{nil}\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{min}.p & = & p^\wedge.a \\
\mathsf{union}.p.q & = & p \bowtie q \\
\mathsf{delmin}.p & = & \mathrm{return}(p^\wedge.l \bowtie p^\wedge.r);\ \mathrm{dispose}(p),
\end{array}$$

where

$$\begin{array}{lcll}
p \bowtie q & = & q & , p = \mathrm{nil} \\
& [] & p & , q = \mathrm{nil} \\
& [] & (\ \mathrm{return}(p);\ p^\wedge.l, p^\wedge.r := p^\wedge.r \bowtie q, p^\wedge.l \quad , p^\wedge.a \le q^\wedge.a & \\
& & []\ \mathrm{return}(q);\ q^\wedge.l, q^\wedge.r := q^\wedge.r \bowtie p, q^\wedge.l \quad , q^\wedge.a \le p^\wedge.a & \\
& &) & , p \ne \mathrm{nil} \wedge q \ne \mathrm{nil}.
\end{array}$$

Strictly speaking, the program for $\bowtie$ is not in-situ yet because it is recursive. It is however a simple form of recursion, and therefore it is not difficult to transform it into an iterative program. We leave this to the interested reader (see [30] for more details).

Bottom-up skew heaps

The algebra used in the programs for bottom-up skew heaps on pages 127 and 129 is algebra T1 from Section 6.2.2:

$$\mathsf{T1} = (\ \langle \mathrm{Int}\rangle_1 \mid [\,], (=[\,]), [\cdot], +\!\!+, hd, tl, lt, ft\ ).$$

Operations $\vdash$ and $\dashv$ have been omitted because they are redundant in the presence of operations $[\cdot]$ and $+\!\!+$ (e.g., $\langle t, a\rangle \vdash u = [\langle t, a\rangle] +\!\!+ u$). It turns out that we must—as far as we know—content ourselves with a destructive refinement of T1 to achieve that all operations have $O(1)$ time complexity. As a consequence, the bottom-up skew heap implementation of PQ will also be destructive. (See Section 6.2.2.)

The programs are:

$$\begin{array}{lcl}
\mathsf{empty} & = & \mathrm{nil} \\
\mathsf{isempty}.p & = & p = \mathrm{nil} \\
\mathsf{single}.a & = & |[\ h{:}R;\ \mathrm{new}(h);\ h^\wedge := \langle h, \langle \mathrm{nil}, a\rangle, \mathrm{nil}\rangle;\ \mathrm{return}(h)\ ]| \\
\mathsf{min}.p & = & p^\wedge.a.1 \\
\mathsf{union}.p.q & = & p \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} q \\
\mathsf{delmin}.p & = & (\mathrm{skip}\ \ , p^\wedge.r = \mathrm{nil}\ \ []\ \ p^\wedge.r^\wedge.l := p^\wedge.l\ \ , p^\wedge.r \ne \mathrm{nil}) \\
& & ;\ \mathrm{return}(p^\wedge.a.0 \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} p^\wedge.r);\ \mathrm{dispose}(p).
\end{array}$$

We do not present a refinement of the program for ⧖ (page 129) at pointer level. A destructive implementation for ⧖ is readily obtained from the implementation of $\mathsf{T1}$ described in Section 6.2.2. For further—more interesting—refinements to an in-situ version of ⧖ we refer to [30]; the above described representation of $\langle \mathrm{Int} \rangle_1$ is called the *ring representation* by Sleator and Tarjan.

9.6 Concluding remarks

In a systematic and formal way we have designed and analyzed various implementations of priority queues. We have improved upon the results of Sleator and Tarjan by deriving somewhat simpler programs, and, more importantly, by deriving roughly twice as low upper bounds for the amortized costs of top-down melding and bottom-up melding. Moreover, we have shown that bottom-up skew heaps are at least as good as top-down skew heaps. The following table summarizes these new results.

	top-down skew heaps	bottom-up skew heaps	
		1st result	2nd result
$\mathcal{A}[\mathsf{union}](p,q)$	$1.44\log_2(m{+}n)$	$O(1)$	$0.64\log_2(m{+}n)$
$\mathcal{A}[\mathsf{delmin}](p)$	$1.44\log_2 m$	$2\log_2 m$	$1.44\log_2 m$

where $m = \#[\![p]\!] + 1$ and $n = \#[\![q]\!] + 1$. An interesting fact is that $1.44\log_2 m$ is an approximation of $\log_\phi m$ with $\phi = (1+\sqrt{5})/2$. So, instead of a base-2 logarithm we get a base-ϕ logarithm.

As for the top-down skew heaps, we are very satisfied with the derivation of the program for $\bowtie$ as well as with its analysis; we consider them to be highly calculational. Our program for $\bowtie$ precisely captures the essence of this operation, and we have shown that it may be refined either by a nondestructive or a destructive program at pointer level. The treatment of the bottom-up skew heaps is less calculational, but still rather systematic. We obtained the program for ⧖ by first transforming a version of $\bowtie$ in terms of type $\langle \mathrm{Int} \rangle_1$ into a tail-recursive one, and subsequently modifying this tail-recursive version a little bit. Applying well-known implementation techniques for lists, we finally constructed a destructive program for bottom-up melding at pointer level. In the analysis of bottom-up melding, however, we used a "top-down simulation" of the derived program for ⧖ in order to amortize the costs of the steps of bottom-up melding in a suitable way.

An interesting observation is that φ can be defined equal to its upper bound in (11) in the analysis of top-down skew heaps. This yields as potential (for $\alpha = 4$ and $\gamma = 0$):

$$
\begin{array}{lcl}
\Phi.\langle\,\rangle & = & 0 \\
\Phi.\langle t,a,u\rangle & = & \Phi.t + \log_4 \frac{\sharp t + \sharp u}{\sharp t} + \Phi.u.
\end{array}
$$

That is, $\Phi.z$ is approximately equal to the sum of the logarithms of the sizes of all *right* subtrees of z. This potential can thus be seen as a variation of the well-known potential for splay trees (see [29] and Chapter 11), which is the sum of the logarithms of the sizes of *all* subtrees.

Finally we would like to remark that deletion of an arbitrary value from a priority queue can be supported as well. To delete a from a skew heap, subtree $\langle t, a, u \rangle$, say, is replaced by $t \bowtie u$, resp. $t \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}} u$. This replacement not only affects the potential of the nodes in t and u, but also the potential of the nodes on the root path of a. In all cases, however, the amortized costs of deletion can be shown to be logarithmic (see also [30]).

Chapter 10

Fibonacci heaps

The following algebra of priority queues plays a central role in important graph algorithms such as Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm:

$$\mathsf{PQ}' = (\,\{\mathrm{Int}\}\ \mid\ \{\,\}, (=\{\,\}), \{\cdot\}, \uplus, \mathit{remin}, \mathit{deckey}\,).$$

Compared to algebra PQ of Chapter 9, algebra PQ' involves sets instead of bags. Note that union of sets is restricted to *disjoint* set union by means of operation $\uplus$. Another difference between these algebras is that operations $\downarrow$ and $\Downarrow$ have been combined into one operation *remin*: $\mathit{remin}.S = (\downarrow S\,, \Downarrow S\,)$ for nonempty S. In this way, the implementation of algebra PQ' is somewhat simplified, whereas its applicability is only slightly reduced. Moreover, it is not difficult to adapt the implementation presented in this chapter in such a way that $\downarrow$ is available as inspection operation.

The important difference with algebra PQ is, however, that algebra PQ' provides *deckey* ("decrease key") of type $\mathrm{Int}\times\mathrm{Nat}\times\{\mathrm{Int}\}\curvearrowright\{\mathrm{Int}\}$ as additional operation, which is defined by

$$\mathit{deckey}.a.k.S = S\backslash\{a\} \uplus \{a-k\}, \quad \text{provided } a \in S \text{ and } a-k \notin S\backslash\{a\}.$$

Using skew heaps, an implementation of *deckey* of logarithmic amortized complexity is easily constructed as a deletion followed by an insertion. The problem is that an implementation with $O(1)$ (amortized) time complexity is required to make the above-mentioned graph algorithms efficient.

In this chapter we present an implementation of PQ' based upon the *Fibonacci heaps* invented by Fredman and Tarjan [8]. As signature for this implementation we take

$$\mathsf{FH} = (\,F\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{union}, \mathsf{remin}, \mathsf{deckey}\,),$$

and we will use an abstraction function $[\![\cdot]\!] \in F \to \{\mathrm{Int}\}$ to couple this algebra with PQ'. To enhance the clarity of exposition we first construct an implementation in which operation deckey is ignored. The concrete algebra without operation deckey is called LBQ ("lazy binomial queues"). Our solution for LBQ is based upon *binomial queues*, a data structure invented by Vuillemin [34].

The aim is to achieve $O(\log \#[\![p]\!])$ amortized costs for remin.p and $O(1)$ amortized costs for the other operations. The amortized costs are defined according to the general scheme of Section 5.5:

$$
\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](p) & = & \mathcal{T}[\mathsf{isempty}](p) \\
\mathcal{A}[\mathsf{single}](a) & = & \mathcal{T}[\mathsf{single}](a) + \Phi.(\mathsf{single}.a) \\
\mathcal{A}[\mathsf{union}](p,q) & = & \mathcal{T}[\mathsf{union}](p,q) + \Phi.(\mathsf{union}.p.q) - \Phi.p - \Phi.q \\
\mathcal{A}[\mathsf{remin}](p) & = & \mathcal{T}[\mathsf{remin}](p) + \Phi.(\mathsf{remin}.p.1) - \Phi.p \\
\mathcal{A}[\mathsf{deckey}](a,k,p) & = & \mathcal{T}[\mathsf{deckey}](a,k,p) + \Phi.(\mathsf{deckey}.a.k.p) - \Phi.p.
\end{array}
$$

Deletion can then be implemented with logarithmic amortized cost as well, since $S \setminus \{a\} = \Downarrow(\mathit{deckey}.a.\infty.S)$—using the extreme value ∞ as a trick to achieve this.

10.1 Lazy binomial queues

The starting-point for the definition of LBQ is that data type F is a set of *forests* over Int. In contrast with skew heaps, the trees in these forests are not restricted to binary trees. Also the order of the trees in a forest is immaterial, so we have

(1) $\quad F \subseteq \{\langle \mathrm{Int} \rangle_6\}$,

where tree type $\langle \mathrm{Int} \rangle_6$ has been defined in Section 6.3.1 as the smallest solution of

$$X : \quad X = \mathrm{Int} \times \{X\}.$$

Recall from Chapter 6 that this means that an element $\langle a, t \rangle$ of this type denotes a nonempty tree with value $a \in \mathrm{Int}$ attached to the root and the trees in forest $t \in \{\langle \mathrm{Int} \rangle_6\}$ as subtrees of the root.

A forest represents a set in $\{\mathrm{Int}\}$ according to abstraction $[\![\cdot]\!]$ defined on $\{\langle \mathrm{Int} \rangle_6\}$ by

$$
\begin{array}{lcl}
[\![\{\,\}]\!] & = & \{\,\} \\
[\![\{\langle a,t \rangle\} \cup f]\!] & = & \{a\} \cup [\![t]\!] \cup [\![f]\!].
\end{array}
$$

To allow for efficient implementation of the operations, two restrictions are imposed on F. The first one is that

(2) $\quad (\forall f : f \in F : (\forall x : x \in f : x \text{ is a heap}))$,

where

$$\langle a, t \rangle \text{ is a heap} \;\equiv\; a < \downarrow[\![t]\!] \;\wedge\; (\forall x : x \in t : x \text{ is a heap}).$$

In connection with operation remin, a second restriction on F follows later on, but first we consider the other operations of LBQ for which programs with $O(1)$ cost are now easily obtained.

For the first three operations, there is no choice:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \{\,\} \\ \mathsf{isempty}.f & \doteq & f = \{\,\} \\ \mathsf{single}.a & \doteq & \{\langle a, \{\,\}\rangle\}, \end{array}$$

and for union we take the following simple program:

$$\mathsf{union}.f.g \doteq f \cup g.$$

The fact that—in contrast with the binomial queues in [34]—forests f and g are simply united in this program for union, is the reason why we speak of "*lazy* binomial queues". As a consequence, the number of trees in f may be as large as $\#[\![f]\!]$ for f in $\mathrm{rng}\,\mathsf{LBQ}$ because f may consist of $\#f$ singleton trees.

Remark 10.1
As union refines $\uplus$, it follows that $[\![f]\!]$ and $[\![g]\!]$ in the program for $\mathsf{union}.f.g$ are disjoint. Consequently, we have that for all f in the range of LBQ (and FH):

$$\#[\![f]\!] = (\Sigma\, x : x \in f : \#x),$$

which means that these forests do not contain duplicates. Moreover, this implies that f and g in the program for $\mathsf{union}.f.g$ are disjoint as well. In the sequel we will rely on these facts, for example when we use that $\#(f \cup g) = \#f + \#g$.
□

The computation of $\mathsf{remin}.f$ involves the computation of $\downarrow[\![f]\!]$ and the computation of a representation for $\Downarrow[\![f]\!]$. As the trees in f are heaps, a search through the roots of f suffices to determine $\downarrow[\![f]\!]$, which takes time proportional to $\#f$. Because of the lazy implementation of union, the worst-case complexity of remin is thus linear.

To obtain logarithmic amortized costs for remin, the idea is to define $\mathsf{remin}.f$ such that the number of trees in the forest returned by this operation is small compared to $\#[\![f]\!]$. In this way, future applications of remin will be cheaper. To implement this idea, we take a program of the following form:

$$\mathsf{remin}.f \doteq (a, t \cup g) \quad |[\; \{\langle a, t\rangle\} \cup g = \mathit{condense}.f \quad , a < \downarrow[\![g]\!] \;]|,$$

where $\mathit{condense}$ should satisfy $[\![\mathit{condense}.f]\!] = [\![f]\!]$. The sole purpose of function $\mathit{condense}$ is to reduce the number of trees in f. It will be implemented such that the actual cost of $\mathsf{remin}.f$ is proportional to $\#f$, and therefore we will define $\mathcal{T}$

such that $\mathcal{T}[\mathsf{remin}](f) = \#f$. With potential Φ defined by $\Phi.f = \#f$, we then have

$$\begin{array}{rcl} \mathcal{A}[\mathsf{remin}](f) & = & \mathcal{T}[\mathsf{remin}](f) + \Phi.(\mathsf{remin}.f.1) - \Phi.f \\ & = & \#f + \#(t \cup g) - \#f \\ & = & \#t + \#g. \end{array}$$

Note that with this potential, the amortized costs of the other operations are indeed $O(1)$; in particular, $\mathcal{A}[\mathsf{union}](f,g) = 1$ because $\#(f \cup g) = \#f + \#g$. To achieve that $\mathcal{A}[\mathsf{remin}]$ is logarithmic, we implement *condense* such that both $\#t$ and $\#g$ are $O(\log \#[\![f]\!])$.

Remark 10.2
An alternative program for remin is:

$$\mathsf{remin}.f = (a, condense.(t \cup g)) \quad |[\ \{\langle a,t\rangle\} \cup g = f \quad , a < \downarrow[\![g]\!]\ ]|.$$

That is, first the tree with minimum root is determined and its root is removed, and, subsequently, the remaining forest is "condensed". As actual costs for this program we take $1 + \#t + \#g$. With the same potential Φ, we now have

$$\begin{array}{rcl} \mathcal{A}[\mathsf{remin}](f) & = & \mathcal{T}[\mathsf{remin}](f) + \Phi.(\mathsf{remin}.f.1) - \Phi.f \\ & = & (1 + \#t + \#g) + \#(condense.(t \cup g)) - (1 + \#g) \\ & = & \#t + \#(condense.(t \cup g)). \end{array}$$

To guarantee that $\mathcal{A}[\mathsf{remin}]$ is logarithmic, *condense* should be designed such that both $\#t$ and $\#(condense.(t \cup g))$ are $O(\log \#[\![f]\!])$.

In this way, virtually the same requirements for *condense* and bounds for $\mathcal{A}[\mathsf{remin}]$ are obtained. The actual costs of the first program for remin are however smaller and therefore it is preferred; in other words, the first program is "lazier". □

Application of function *condense* will combine trees of the forest to which it is applied. For this purpose we use operator $\bowtie$ ("link"), which combines two trees into one under invariance of the heap order. For trees of type $\langle \mathrm{Int}\rangle_6$, this can be done by a simple nonrecursive program:

$$\begin{array}{rcll} \langle a,t\rangle \bowtie \langle b,u\rangle & \doteq & \langle a, t \cup \{\langle b,u\rangle\}\rangle & , a < b \\ & [\!] & \langle b, u \cup \{\langle a,t\rangle\}\rangle & , b < a. \end{array}$$

Since a forest f can be reduced to a singleton forest $\{\langle a,t\rangle\}$, say, by means of $\#f-1$ links, we could now define $condense.f = \{\langle a,t\rangle\}$. Then $g = \{\,\}$ in the program for $\mathsf{remin}.f$, hence $\#g$ is evidently logarithmic in $\#[\![f]\!]$. The problem is that $\#t$ may be linear in $\#[\![f]\!]$, when the trees of f are linked in an arbitrary order. Therefore, a more sophisticated method of linking is required.

To achieve that $\#t$ is logarithmic in $\#[\![f]\!]$, the solution of Vuillemin [34] is to constrain the applications of $\bowtie$ to trees of equal size. That is, $\#x = \#y$ is taken as a precondition for $x \bowtie y$. As a result, in a forest constructed by means of the operations of LBQ, any tree x satisfies

(3) $\#x = 2^{\dagger x}$,

where $\dagger x$ denotes the *rank of* x defined by $\dagger\langle a,t\rangle = \#t$, the number of subtrees of a. Indeed (3) holds for singleton trees, and with respect to linking we reason as follows. Let x and y satisfy (3) and assume that $\#x = \#y$. Then $\dagger x = \dagger y$ as well on account of (3), and we observe:

$$
\begin{array}{ll}
 & \#(x \bowtie y) \\
= & \{ \text{ definitions of } \bowtie \text{ and } \# \ \} \\
 & \#x + \#y \\
= & \{ \ x \text{ and } y \text{ satisfy (3) } \} \\
 & 2^{\dagger x} + 2^{\dagger y} \\
= & \{ \ \dagger(x \bowtie y) = \dagger x + 1 = \dagger y + 1 \ \} \\
 & 2^{\dagger(x \bowtie y)} .
\end{array}
$$

What is more, this property is preserved by $\bowtie$ for all subtrees of x and y as well. As a consequence, all trees in forests $f \in \operatorname{rng} \mathsf{LBQ}$ are *binomial*, where

$$\langle a,t\rangle \text{ is binomial } \equiv \ \#\langle a,t\rangle = 2^{\#t} \ \wedge \ (\forall x : x \in t : x \text{ is binomial}).$$

The third and last restriction on F now is

(4) $(\forall f : f \in F : (\forall x : x \in f : x \text{ is binomial}))$,

and type F is defined as the largest set satisfying (1), (2), and (4). This set may be paraphrased as the set of forests of binomial heaps (without duplicates).

Remark 10.3
Binomial trees enjoy numerous nice properties. For instance, the height of a binomial tree is equal to its rank, and the number of nodes at depth d in a binomial tree of rank r—hence of height r as well—equals $\binom{r}{d}$, for $0 \le d \le r$. The latter property explains the name of these trees and its straightforward proof relies on the identity $\binom{r+1}{d+1} = \binom{r}{d} + \binom{r}{d+1}$.
□

We can now complete the program for remin. Due to the definition of F, tree $\langle a,t\rangle$ is binomial, hence $\#t$, which is equal to $\log_2 \#\langle a,t\rangle$, is at most $\log_2 \#[\![f]\!]$. To bound the size of g, *condense* is programmed as follows:

$$
\begin{array}{lcll}
\mathit{condense}.f & = & f & ,S = \{\,\} \\
 & [\!] & \mathit{condense}.(f\backslash\{x,y\} \cup \{x \bowtie y\}) \ \ [\![\ (x,y) \in S \ ]\!] & ,S \neq \{\,\} \\
 & & [\![\ S = \{(x,y) \mid x,y \in f \ \wedge \ x \neq y \ \wedge \ \dagger x = \dagger y\} \ ]\!] . &
\end{array}
$$

The idea is that *condense* combines trees until all ranks are different, so that forest *condense*.f contains no trees of equal rank. Since the rank of any tree in *condense.f* is at most $\log_2 \#[\![f]\!]$, it then follows that $\#(condense.f)$ is at most $1 + \log_2 \#[\![f]\!]$, which implies $\#g \leq \log_2 \#[\![f]\!]$.

To implement this program such that *condense.f* takes $O(\#f)$ time, we use the method described in [8]. This method uses an array with domain $[0..\log_2 \#[\![f]\!]]$ and range $\mathrm{Bool} \times \langle \mathrm{Int} \rangle_6$. Initially, all boolean components are false, indicating that the tree components are "free". Then the trees of f are inserted one by one at the position indexed by their rank (which is at most $\log_2 \#[\![f]\!]$ for binomial trees). In case the tree component is free, the tree is inserted. Otherwise, this tree and the tree already occupying the position can be linked because they have the same rank, thereby emptying the position. The result is a tree whose rank is one larger, which must now be inserted at the next position, and so on.

When all trees have been inserted, forest *condense.f* can be extracted from the array, in the process determining the tree with minimum root. To avoid the $O(\log \#[\![f]\!])$ initialization of the array, the tricky representation of arrays (Section 7.2) can be used, which reduces the cost of initialization to $O(1)$. The actual cost of $\mathsf{remin}.f$ is then proportional to $\#f$—as assumed in the above. Another possibility is, however, to use $O(\log \#[\![f]\!])$ time for the initialization of the array. The actual cost is then $O(\#f \;\max\; \log \#[\![f]\!])$, hence the logarithmic bound for $\mathcal{A}[\mathsf{remin}]$ is not affected.

This completes the description of an implementation of $\mathsf{remin}.f$ whose amortized costs are bounded by $2 \log_2 \#[\![f]\!]$. Observing that cost measure $\mathcal{T}$ approximately counts the number of comparisons used by $\mathsf{remin}.f$, we see that the amortized number of comparisons used by $\mathsf{remin}.f$ is also at most $2 \log_2 \#[\![f]\!]$. (The actual number of comparisons made by $\mathsf{remin}.f$ equals $\#f{-}1$, namely $\#f{-}\#(condense.f)$ comparisons by links and $\#(condense.f){-}1$ comparisons to determine the minimum tree.) From this observation we conclude that $2N \log_2 N$ is an asymptotic bound on the number of comparisons required to sort N numbers by means of lazy binomial queues (since this requires N applications of remin). In Section 10.3 we shall see that addition of operation *deckey* slightly increases this bound.

10.2 Intermezzo on the precondition of linking

In the previous section $\bowtie$ is defined as

$$
\begin{array}{lcll}
\langle a,t \rangle \bowtie \langle b,u \rangle & \doteq & \langle a, t \cup \{\langle b,u \rangle\} \rangle & , a < b \\
 & [\,] & \langle b, u \cup \{\langle a,t \rangle\} \rangle & , b < a,
\end{array}
$$

and application of $\bowtie$ is restricted to trees of equal size (or rank) so that (3) holds. Instead of (3), however, a more general approach is to bound the ranks by

(5) $\quad \alpha^{\dagger x} \leq \#x,$

for some constant α larger than 1. Evidently, this condition is satisfied by singleton trees, and for $x \bowtie y$, with $x = \langle a, t \rangle$ and $y = \langle b, u \rangle$, we observe for the case $a < b$:

$$\begin{array}{ll} & \alpha^{\dagger(x \bowtie y)} \leq \#(x \bowtie y) \\ \equiv & \{ \text{ definition of } \bowtie ;\ \#(x \bowtie y) = \#x + \#y \ \} \\ & \alpha^{\dagger\langle a,\, t \,\cup\, \{\langle b, u \rangle\}\rangle} \leq \#x + \#y \\ \equiv & \{ \text{ definition of } \dagger \ \} \\ & \alpha^{1+\dagger x} \leq \#x + \#y. \end{array}$$

Under the assumption that x and y satisfy (5), there are several ways to simplify this condition. For instance, we can head for a condition in terms of $\#x$ and $\#y$:

$$\begin{array}{ll} & \alpha^{1+\dagger x} \leq \#x + \#y \\ \Leftarrow & \{ \text{ use } \alpha^{\dagger x} \leq \#x \text{ to eliminate } \dagger x \ \} \\ & \alpha\#x \leq \#x + \#y \\ \equiv & \{ \ \} \\ & (\alpha - 1)\#x \leq \#y. \end{array}$$

By symmetry, we obtain $(\alpha - 1)\#y \leq \#x$ as condition for the case $b < a$. As precondition for $x \bowtie y$ in terms of sizes we thus get

$$(6) \quad \alpha - 1 \leq \frac{\#x}{\#y} \leq \frac{1}{\alpha - 1}.$$

Another possibility is to look for a condition in terms of $\dagger x$ and $\dagger y$:

$$\begin{array}{ll} & \alpha^{1+\dagger x} \leq \#x + \#y \\ \Leftarrow & \{ \text{ use } \alpha^{\dagger x} \leq \#x \text{ to eliminate } \#x \text{ and similarly for } y \ \} \\ & \alpha^{1+\dagger x} \leq \alpha^{\dagger x} + \alpha^{\dagger y} \\ \equiv & \{ \ \} \\ & (\alpha - 1)\alpha^{\dagger x} \leq \alpha^{\dagger y}. \end{array}$$

Combined with the other case this yields as precondition for $x \bowtie y$ in terms of ranks:

$$(7) \quad \log_\alpha(\alpha - 1) \leq \dagger x - \dagger y \leq \log_\alpha \tfrac{1}{\alpha - 1},$$

using that $\log_\alpha$ is monotonic for $\alpha > 1$.

These preconditions make sense for $\alpha \leq 2$ only. Hence, at this point, we may decide to take $\alpha = 2$, since this gives the best bound for $\#t$ in the program for $\mathsf{remin}.f$, viz. $\#t \leq \log_2 \#\langle a, t \rangle$, using that $\#t = \dagger\langle a, t \rangle$. However, taking α smaller than 2 possibly reduces $\#(\mathit{condense}.f)$. For instance, condition (7) may be reformulated as

$$|\dagger x - \dagger y| \leq K,$$

with $K = -\log_\alpha(\alpha - 1)$. In the program for *condense*, trees can now be linked until all ranks are more than K apart. In this way, at most $(\log_\alpha \#[\![f]\!])/(K{+}1)$ trees remain in the end. As upper bound on $\#t + 1 + \#g$ we then have

$$\begin{array}{ll} & \log_\alpha \#[\![f]\!] + (\log_\alpha \#[\![f]\!])/(K{+}1) \\ = & \{\ K{+}1 = \log_\alpha \frac{\alpha}{\alpha-1}\ \} \\ & (\frac{1}{\log_2 \alpha} + \frac{1}{\log_2 \frac{\alpha}{\alpha-1}})\log_2 \#[\![f]\!]. \end{array}$$

However, minimizing this bound over all $\alpha > 1$ yields $2\log_2 \#[\![f]\!]$ as minimal bound at $\alpha = 2$, from which we conclude that $\alpha = 2$ is indeed an appropriate choice. It corresponds to precondition $\dagger x = \dagger y$ on account of (7). Note that (6) simplifies to $\#x = \#y$ if $\alpha = 2$.

10.3 Fibonacci heaps

Leaving the implementation of the other operations virtually the same, we will now define operation deckey. To allow for a concise definition, we use root-path view $\langle \mathrm{Int} \rangle_7$ which has been discussed in Section 6.3.2:

$$\langle \mathrm{Int} \rangle_7 = [\langle \mathrm{Int} \rangle_6].$$

We recall that a nonempty list of this type represents a tree in $\langle \mathrm{Int} \rangle_6$ according to the following abstraction:

$$\begin{array}{lcl} [\![[x]]\!] & = & x \\ [\![\langle a,t\rangle \vdash v]\!] & = & \langle a, t \cup \{[\![v]\!]\}\rangle \quad , v \neq [\,]. \end{array}$$

In terms of this type, a forest in which a occurs exactly once can be written in the form $\{v \dashv \langle a,t\rangle\} \uplus g$ in precisely one way.

Our goal is to define $\mathsf{deckey}.a.k.(\{v \dashv \langle a,t\rangle\} \uplus g)$ such that the amortized costs are $O(1)$. To this end, we first note that $\langle a{-}k, t\rangle$ is a heap because $\langle a,t\rangle$ is a heap and $a{-}k \leq a$. There is, however, no guarantee that $v \dashv \langle a{-}k, t\rangle$ is a heap as well (if v is nonempty). A first attempt is therefore to remove subtree $\langle a,t\rangle$ from tree $v \dashv \langle a,t\rangle$ and to add tree $\langle a{-}k,t\rangle$, yielding forest $\{v\} \cup \{\langle a{-}k,t\rangle\} \cup g$. But the problem with this solution is, of course, that tree v is in general not binomial, as a consequence of which the relation between v's rank and v's size is destroyed.

However, as observed in the previous section, a property like $\dagger x \leq \log_\alpha \#x$, with $\alpha > 1$, is maintained when a fixed upper bound K on $|\dagger x - \dagger y|$ is used as precondition for $x \bowtie y$. By viewing a tree that has lost some subtrees as a tree of smaller rank, the choice $\{v\} \cup \{\langle a{-}k,t\rangle\} \cup g$ is not so bad provided the number of subtrees lost is not too large.

A solution is therefore to bound for each node the number of subtrees it may lose by a fixed number K, say.[1] To implement this, forests over Int $\times$ $[0..K]$ are used instead of forests over Int. Hence, each node contains in addition a value of type $[0..K]$, and trees are of the form $\langle (a,m),t\rangle$, which we abbreviate to $\langle a,m,t\rangle$. Value m will be increased each time a node loses a subtree. As a consequence, all trees in the forests in rng FH will be *pseudo-binomial.* To be able to define this class of trees explicitly, we first present an alternative characterization of binomial trees:

$$\begin{array}{ll} & \langle a,m,t\rangle \text{ is binomial} \\ \equiv & \{x : x \in t : \dagger x\} = [0..\#t) \;\wedge\; (\forall\, x : x \in t : x \text{ is binomial}). \end{array}$$

That is, the ranks of the trees in t are all different and together they exactly cover the interval $[0..\#t)$. This follows from the fact that $2^k{-}1$ can be written as a sum of powers of 2 in only one way, viz. as $1{+}2{+}\ldots{+}2^{k-1}$. As stepping-stone towards the definition of pseudo-binomial trees, we reformulate this as follows: the increasing list of the ranks of the trees in t is equal to list $[0,1,\ldots,\#t{-}1]$.

To define the class of pseudo-binomial trees, we use $\ddagger x$ instead of $\dagger x$, which is called the *pseudo-rank of* x and is defined by $\ddagger\langle a,m,t\rangle = m{+}\#t$. The definition reads

$$\begin{array}{ll} & \langle a,m,t\rangle \text{ is pseudo-binomial} \\ \equiv & \Lbag x : x \in t : \ddagger x \Rbag \text{ is diagonal} \;\wedge\; (\forall\, x : x \in t : x \text{ is pseudo-binomial}), \end{array}$$

where a bag B is called *diagonal* (which is short for "upper-diagonal") when the ascending list of elements of B is pointwise at least list $[0,1,\ldots,\#B{-}1]$. The relevant properties of diagonal bags are:

(8) $\Lbag\,\Rbag$ is diagonal

(9) B is diagonal $\wedge\; k \geq \#B \;\Rightarrow\; B \oplus \Lbag k \Rbag$ is diagonal

(10) B is diagonal $\Rightarrow\; B \ominus \Lbag k \Rbag$ is diagonal.

The important fact is that for any pseudo-binomial tree x,

(11) $\alpha^{\dagger x} \leq \#x$,

where constant α, $\alpha > 1$, depends on K. Assuming that $\langle a,m,t\rangle$ is pseudo-binomial, we prove this by (well-founded) induction:

$$\begin{array}{ll} & \#\langle a,m,t\rangle \\ = & \{\text{ definition of } \# \;\} \\ & 1 + (\Sigma\, x : x \in t : \#x) \\ \geq & \{\text{ ind. hyp. (11), using that } x \text{ is pseudo-binomial } \} \\ & 1 + (\Sigma\, x : x \in t : \alpha^{\dagger x}) \end{array}$$

[1] This is a generalization of Fredman and Tarjan's method [8] which corresponds to $K = 1$.

$$
\begin{array}{ll}
\geq & \{\ \ddagger x \leq K + \dagger x \text{ and } \alpha > 1\ \} \\
 & 1 + (\Sigma\, x : x \in t : \alpha^{\ddagger x\,-\,K}) \\
\geq & \{\ \text{bag } \Lbag x : x \in t : \ddagger x \Rbag \text{ is diagonal and } \alpha > 1\ \} \\
 & 1 + (\Sigma\, i : 0 \leq i < \#t : \alpha^{i-K}) \\
= & \{\ \text{calculus}\ \} \\
 & 1 + \frac{\alpha^{\#t}-1}{\alpha^{K}(\alpha-1)} \\
\geq & \{\ \text{take } \alpha^{K}(\alpha-1) \leq 1\text{, see below}\ \} \\
 & \alpha^{\#t}.
\end{array}
$$

Taking α as large as possible, we see that we want α to be the *largest* real number satisfying

$$1 < \alpha \ \wedge\ \alpha^{K}(\alpha - 1) \leq 1,$$

or, equivalently, we want α to be the *unique* real number satisfying

$$1 < \alpha \ \wedge\ \alpha^{K+1} = \alpha^{K} + 1.$$

Since α decreases as K increases, $K = 0$ seems the best choice at this point, which corresponds to $\alpha = 2$. But, as we shall see shortly, K must be positive to ensure that the amortized costs of deckey are $O(1)$. The best value for α is therefore $\phi = (1 + \sqrt{5})/2$ for $K = 1$.

So, pseudo-binomial trees satisfy (11), which bounds the ranks of these trees. The problem is now to program the operations of FH such that the trees generated by these operations are pseudo-binomial. To achieve this for deckey, we proceed as follows. We take a definition of the form

$$\mathsf{deckey}.a.k.(\{v \dashv \langle a,m,t\rangle\} \uplus g) = cut.v \cup \{\langle a{-}k,m,t\rangle\} \cup g.$$

Since tree $v \dashv \langle a,m,t\rangle$ is pseudo-binomial, we have that $\langle a{-}k,m,t\rangle$ is pseudo-binomial as well. Tree v is, however, not necessarily pseudo-binomial. For this reason, function cut is introduced, which will be defined such that $cut.v$ transforms tree v into a *forest* of pseudo-binomial trees.

To see that v is not necessarily binomial, suppose $v = w \dashv \langle b,n,u\rangle$. Then $v \dashv \langle a,m,t\rangle$ and $w \dashv \langle b,n,u \uplus \{\langle a,m,t\rangle\}\rangle$ correspond to the same tree, hence both are pseudo-binomial. But $w \dashv \langle b,n,u\rangle$ is not necessarily pseudo-binomial, because $\ddagger\langle b,n,u\rangle = \ddagger\langle b,n,u \uplus \{\langle a,m,t\rangle\}\rangle - 1$ and diagonality of $B \oplus \Lbag k{-}1 \Rbag$ does not follow from diagonality of $B \oplus \Lbag k \Rbag$.

Notice, however, that $w \dashv \langle b,n{+}1,u\rangle$ is pseudo-binomial because $\ddagger\langle b,n{+}1,u\rangle$ equals $\ddagger\langle b,n,u \uplus \{\langle a,m,t\rangle\}\rangle$. Therefore, $cut.(w \dashv \langle b,n,u\rangle)$ is easy to define in case $n{+}1 \leq K$. To solve the remaining case $n = K$, subtree $\langle b,n,u\rangle$ is removed from $w \dashv \langle b,n,u\rangle$ and cut is applied recursively to w. Note that $\langle b,n,u\rangle$ is pseudo-binomial because $\langle b,n,u \uplus \{\langle a,m,t\rangle\}\rangle$ is pseudo-binomial due to property (10). The program for cut thus is:

$$
\begin{array}{lcll}
cut.[\,] & \doteq & \{\,\} & \\
cut.(w \dashv \langle b,n,u\rangle) & \doteq & \{w \dashv \langle b,n{+}1,u\rangle\} & ,n<K \\
 & [] & cut.w \cup \{\langle b,0,u\rangle\} & ,n=K.
\end{array}
$$

To achieve $O(1)$ amortized cost, the amortized cost of the recursive alternative must be at most 0. For this purpose, it is essential that n is replaced by 0 in case $n = K$, as will be shown below. That K should be positive is clear: otherwise case $n < K$ never applies.

Remark 10.4
As explained in Chapter 6, a list v of type $\langle T\rangle_7$ represents more than a tree: it represents the root path to the last element of v. This property is exploited in the above program for $\mathsf{deckey}.a.k.(\{v \dashv \langle a,m,t\rangle\} \uplus g)$ in which v denotes the root path to $\langle a,m,t\rangle$. Recursive applications of *cut* generated during evaluation of $cut.v$ all take place along this root path. These recursive applications are called "cascading cuts" by Fredman and Tarjan in [8].
□

To complete the analysis, potential Φ is extended as follows:

$$\Phi.f = \#f + 2\Psi.f, \text{ where } \Psi.f = (\#\, a,m : (a,m) \in f : m = K),$$

i.e., Ψ counts the number of nodes that have lost the maximal number of subtrees. The role of factor 2 in the definition of Φ will be clarified below. Note that $\Phi.(f \cup g) = \Phi.f + \Phi.g$ for disjoint f and g.

We calculate $\mathcal{A}[\mathsf{deckey}]$, writing f for $\{v \dashv \langle a,m,t\rangle\} \uplus g$:

$$
\begin{array}{ll}
 & \mathcal{A}[\mathsf{deckey}](a,k,f) \\
= & \{\text{ definition of } \mathcal{A}[\mathsf{deckey}] \ \} \\
 & \mathcal{T}[\mathsf{deckey}](a,k,f) + \Phi.(\mathsf{deckey}.a.k.f) - \Phi.f \\
= & \{\text{ definition of } \mathsf{deckey} \ \} \\
 & \mathcal{T}[cut](v) + \Phi.(cut.v \cup \{\langle a{-}k,m,t\rangle\} \cup g) - \Phi.(\{v \dashv \langle a,m,t\rangle\} \uplus g) \\
= & \{\text{ property of } \Phi \ \} \\
 & \mathcal{T}[cut](v) + \Phi.(cut.v) + \Phi.\{\langle a{-}k,m,t\rangle\} + \Phi.g - \Phi.\{v \dashv \langle a,m,t\rangle\} - \Phi.g \\
= & \{\text{ definition of } \Phi \text{ and } \Psi.\langle a{-}k,m,t\rangle = \Psi.\langle a,m,t\rangle \ \} \\
 & \mathcal{T}[cut](v) + \Phi.(cut.v) + 1 - \Phi.\{v\} \\
= & \{\text{ definition of } \mathcal{A}[cut], \text{ see below } \} \\
 & \mathcal{A}[cut](v) + 1.
\end{array}
$$

As amortized costs for *cut* we thus take:

$$\mathcal{A}[cut](v) = \mathcal{T}[cut](v) + \Phi.(cut.v) - \Phi.\{v\}.$$

Then $\mathcal{A}[cut]([\,]) = 0$, and $\mathcal{A}[cut](w \dashv \langle b,n,u\rangle) \le 3$ in case $n < K$, and for case $n = K$ we derive:

$$
\begin{array}{cl}
 & \mathcal{A}[cut](w \dashv \langle b,n,u\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[cut] \ \} \\
 & \mathcal{T}[cut](w \dashv \langle b,n,u\rangle) + \Phi.(cut.(w \dashv \langle b,n,u\rangle)) - \Phi.\{w \dashv \langle b,n,u\rangle\} \\
= & \{ \text{ definition of } cut \ \} \\
 & 1 + \mathcal{T}[cut](w) + \Phi.(cut.w \cup \{\langle b,0,u\rangle\}) - \Phi.\{w \dashv \langle b,n,u\rangle\} \\
= & \{ \text{ definition of } \Phi\text{, using } 0 < K \text{ and } n = K \ \} \\
 & 1 + \mathcal{T}[cut](w) + \Phi.(cut.w) + 1 + 0 + 2\Psi.u - \Phi.\{w\} - 2 - 2\Psi.u \\
= & \{ \text{ definition of } \mathcal{A}[cut] \ \} \\
 & \mathcal{A}[cut](w).
\end{array}
$$

This derivation explains why the factor 2 is needed in the definition of Φ. We conclude that $\mathcal{A}[cut](v) \leq 3$, hence that $\mathcal{A}[\mathsf{deckey}](a,k,f) \leq 4$.

To complete the definition of FH we must adapt the definition of $\bowtie$. A possible definition is

$$
\begin{array}{rcll}
\langle a,m,t\rangle \bowtie \langle b,n,u\rangle & \doteq & \langle a,m,t \cup \{\langle b,0,u\rangle\}\rangle & ,a < b \\
 & [] & \langle b,n,u \cup \{\langle a,0,t\rangle\}\rangle & ,b < a,
\end{array}
$$

under precondition $\#t = \#u$. Then $x \bowtie y$ is pseudo-binomial because of property (9) and the fact that $\ddagger x \geq \dagger x$ and $\ddagger y \geq \dagger y$. Note that n and m are replaced by 0 in cases $a < b$ and $b < a$, respectively; this is done to improve the performance a little bit. (An alternative is to use $\ddagger x = \ddagger y$ as precondition for $x \bowtie y$, but then this optimization is not allowed.)

In summary, the operations of FH are

$$
\begin{array}{lcl}
\mathsf{empty} & \doteq & \{\,\} \\
\mathsf{isempty}.f & \doteq & f = \{\,\} \\
\mathsf{single}.a & \doteq & \{\langle a,0,\{\,\}\rangle\} \\
\mathsf{union}.f.g & \doteq & f \cup g \\
\mathsf{remin}.f & \doteq & (a, t \cup g) \quad |[\ \{\langle a,m,t\rangle\} \cup g = condense.f \quad ,a < \downarrow[\![g]\!] \ ]|,
\end{array}
$$

in which

$$
\begin{array}{lcll}
condense.f & = & f & ,S = \{\,\} \\
 & [] & condense.(f \backslash \{x,y\} \cup \{x \bowtie y\}) \ \ |[\ (x,y) \in S \ ]| & ,S \neq \{\,\} \\
 & & |[\ S = \{(x,y) \mid x,y \in f \ \wedge \ x \neq y \ \wedge \ \dagger x = \dagger y\} \ ]|, &
\end{array}
$$

and

$$
\mathsf{deckey}.a.k.(\{v \dashv \langle a,m,t\rangle\} \cup g) = cut.v \cup \{\langle a{-}k,m,t\rangle\} \cup g,
$$

where

$$
\begin{array}{lcll}
cut.[\,] & \doteq & \{\,\} & \\
cut.(w \dashv \langle b,n,u\rangle) & \doteq & \{w \dashv \langle b,n{+}1,u\rangle\} & ,n < K \\
 & [] & cut.w \cup \{\langle b,0,u\rangle\} & ,n = K.
\end{array}
$$

The amortized cost of $\mathsf{remin}.f$ is bounded by $2\log_\phi \#[\![f]\!]$ if we take $K = 1$, and the amortized costs of the other operations are $O(1)$.

To sort N numbers by means of Fibonacci heaps, we see that the asymptotic bound on the number of comparisons becomes $2N\log_\phi N$, That is, at most $2.88N\log_2 N$ comparisons instead of $2N\log_2 N$ comparisons for lazy binomial queues.

10.4 Concluding remarks

We consider it a nice result that our formal description of Fibonacci heaps—which captures the essence of this data structure—fits on less than one page. The level of detail has been chosen such that this description provides a sound basis for a formal (amortized) efficiency analysis. In addition, the programs for FH can be refined further to programs operating on pointers and arrays, using the techniques described in Chapter 6. The key to this achievement is the use of type $\langle T\rangle_6$ along with its root-path view $\langle T\rangle_7$. In terms of the latter type, the programs for deckey and *cut* become remarkably concise. Moreover, the amortized costs of these programs can be determined in a calculational way.

The reason for the name "Fibonacci heaps" is that the Fibonacci sequence plays a role in the analysis by Fredman and Tarjan. In our analysis the Fibonacci sequence does not play a role anymore, but the quantity $\phi = (1+\sqrt{5})/2$, the "golden ratio", which is intimately connected with the Fibonacci sequence, still does. We remark that the number α given by

$$1 < \alpha \ \wedge\ \alpha^{K+1} = \alpha^K + 1,$$

is connected with a similar sequence H, which is defined by

$$\begin{array}{lcll} H.i & = & 1 & ,0 \le i \le K \\ H.(i{+}1) & = & H.i + H.(i{-}K) & ,i \ge K. \end{array}$$

A connection is, for instance, that for $i \ge 0$:

$$\alpha^{i-K} \le H.i \le \alpha^i.$$

To prove that each tree in a Fibonacci heap has a size at least exponential in the rank of its root, Fredman and Tarjan first prove the following lemma [8, Lemma 1]:

> Let x be any node in a Fibonacci heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the i-th child of x has a rank of at least $i{-}2$.

The formulation of this lemma as well as its proof are very operational (in the proof, they use phrases like "consider the time when y was linked to x" and "after the linking...".) We have avoided such operational reasoning altogether by introducing the notion of pseudo-binomial trees, which precisely captures the kind of trees that arise in Fibonacci heaps. For these trees we were able to show (in one go) that their size is exponential in their rank (cf. inequality (11)).

One may wonder why we have taken the trouble to treat the general case instead of confining ourselves to case $K{=}1$—as Fredman and Tarjan did in [8]. Well, a good reason is that our analysis shows that operation *cut* can be avoided in particular applications of the graph algorithms. Take, for instance, Dijkstra's shortest path algorithm which is defined for directed graphs, and assign to K the maximal number of incoming arcs of any vertex. Then the recursive alternative of *cut* never applies, hence *deckey* can be implemented much simpler—taking $cut.v = \{v\}$ for nonempty v. As bound on the amortized number of comparisons used by $\mathsf{remin}.f$ we then get $2\log_\alpha \#[\![f]\!]$, which is reasonably small for small values of K.

Chapter 11

Path reversal, splaying, and pairing

In this chapter we analyze the amortized efficiency of three similar operations on trees. In the next section, we first investigate path reversal in detail, for which a potential function is derived in a calculational way. Since splaying and pairing are very similar to path reversal, these operations can then be analyzed along the same lines.

11.1 Path reversal

Path reversal is an operation on trees that moves a node to the root in a specific way (see Figure 11.1). It has been introduced in [32] as an alternative path compression technique, called "reversal of type zero". Such path compression techniques are used in efficient solutions to the well-known disjoint set union problem, but here we shall only be concerned with the amortized complexity of path reversal.

In the next section we first present a concise program for path reversal. Using the same cost measure as in [12], we then analyze the amortized cost of this program. We also analyze a top-down simulation of path reversal to show how this simplifies the calculations. Subsequently, the cost of a series of path reversals is determined, which is also done in [12]. Our results exactly match the results of [12], the important difference being that we *derive* the required potential function to a large extent.

11.1.1 Program

Let T be a nonempty type. Path reversal operates on trees of type $\langle T\rangle_6$, which has been introduced in Section 6.3.1 as the smallest solution of

$$X: \quad X = T \times \{X\}.$$

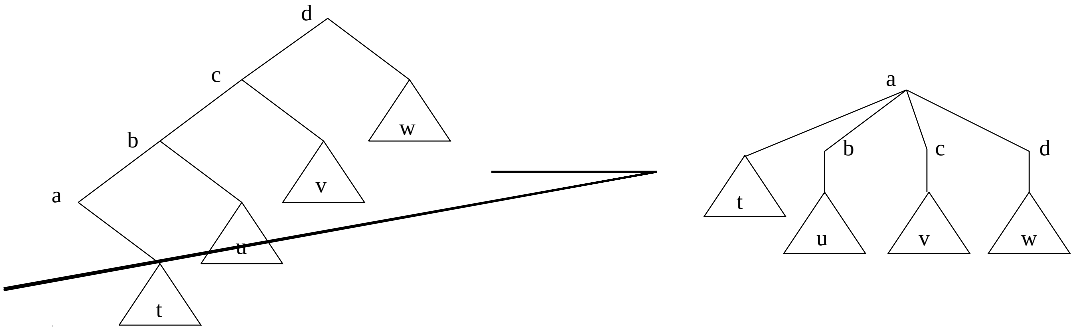

Figure 11.1: Path reversal at a (triangles denote *sets* of subtrees).

In terms of this tree type, path reversal is an operation of type $T \times \langle T\rangle_6 \curvearrowright \langle T\rangle_6$. We denote it by $\angle$, and $a \angle x$ is defined only if a occurs exactly once in x.

To allow for a simple description of path reversal, we use root-path view $\langle T\rangle_7$ from Section 6.3.2—which was also used in the definition of deckey in Section 10.3. In terms of this type, the program for path reversal reads

$$
\begin{array}{lll}
\text{(i)} & a \angle \langle a, t\rangle & = \langle a, t\rangle \\
\text{(ii)} & a \angle (y \dashv x \dashv \langle a, t\rangle) & \doteq a \angle (y \dashv \langle a, \{x\} \cup t\rangle),
\end{array}
$$

in which $t \in \{\langle T\rangle_6\}$, $x \in \langle T\rangle_6$, and $y \in \langle T\rangle_7$. The cost measure defined in this program, which we will call $\mathcal{T}$, coincides with the measure used by Ginat, Sleator, and Tarjan in [12], and therefore our results will be comparable to their results.

11.1.2 Bottom-up analysis

We analyze $(a\angle)$, for arbitrary a, for which the amortized cost is defined by

$$\mathcal{A}[a\angle](x) = \mathcal{T}[a\angle](x) + \Phi.(a\angle x) - \Phi.x.$$

Our goal is to define Φ such that

(1) $\mathcal{A}[a\angle](x) \le \log_\alpha \#x,$

and α is as large as possible—in any case larger than 1. Furthermore, we want Φ to be nonnegative and small.

The first step is to calculate a recurrence relation for $\mathcal{A}[a\angle]$. For case (i) we observe:

$$
\begin{array}{ll}
 & \mathcal{A}[a\angle](\langle a, t\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[\angle] \ \} \\
 & \mathcal{T}[a\angle](\langle a, t\rangle) + \Phi.(a\angle \langle a, t\rangle) - \Phi.\langle a, t\rangle
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } a\angle \ \} \\
& 0 + \Phi.\langle a, t\rangle - \Phi.\langle a, t\rangle \\
= & \{ \ \} \\
& 0,
\end{array}
$$

and for case (ii) we observe:

$$
\begin{array}{ll}
& \mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[\angle\,] \ \} \\
& \mathcal{T}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) + \Phi.(a\angle\,(y \dashv x \dashv \langle a, t\rangle)) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } a\angle \ \} \\
& 1 + \mathcal{T}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + \Phi.(a\angle\,(y \dashv \langle a, \{x\} \cup t\rangle)) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ definition of } \mathcal{A}[a\angle\,] \ \} \\
& \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle).
\end{array}
$$

So, the recurrence relation is:

$$
\begin{array}{lcl}
\mathcal{A}[a\angle\,](\langle a, t\rangle) & = & 0 \\
\mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) & = & \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) \\
& & +1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle).
\end{array}
$$

The term $1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle)$ may be interpreted as the amortized cost of unfolding $a\angle\,(y \dashv x \dashv \langle a, t\rangle)$.

The obvious way to proceed is now to establish (1) by induction on $\#x$, using the above recurrence relation. Unfortunately, this is bound to fail because the sizes of the arguments of $\mathcal{A}[a\angle\,]$ are the same on both sides in case (ii). There is however an important difference between $y \dashv x \dashv \langle a, t\rangle$ and $y \dashv \langle a, \{x\} \cup t\rangle$: the subtree rooted at a increases in size (and the depth of a decreases). For this reason, the following strengthening of (1) is an appropriate induction hypothesis:

$$
(2) \quad \mathcal{A}[a\angle\,](y \dashv \langle a, t\rangle) \leq \log_\alpha \frac{\#(y \dashv \langle a, t\rangle)}{\#\langle a, t\rangle}.
$$

We apply induction on y. It is obvious that (2) holds in case (i) ($y = [\,]$), and for case (ii) we derive:

$$
\begin{array}{ll}
& \mathcal{A}[a\angle\,](y \dashv x \dashv \langle a, t\rangle) \\
= & \{ \text{ above recurrence relation } \} \\
& \mathcal{A}[a\angle\,](y \dashv \langle a, \{x\} \cup t\rangle) + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
\leq & \{ \text{ induction hypothesis (2) } \} \\
& \log_\alpha \dfrac{\#(y \dashv \langle a, \{x\} \cup t\rangle)}{\#\langle a, \{x\} \cup t\rangle} + 1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \\
\leq & \{ \text{ see requirement on } \Phi \text{ below } \} \\
& \log_\alpha \dfrac{\#(y \dashv x \dashv \langle a, t\rangle)}{\#\langle a, t\rangle}.
\end{array}
$$

Hence, (2) holds provided Φ satisfies

$$1 + \Phi.(y \dashv \langle a, \{x\} \cup t\rangle) - \Phi.(y \dashv x \dashv \langle a, t\rangle) \le \log_\alpha \frac{\#\langle a, \{x\} \cup t\rangle}{\#\langle a, t\rangle}.$$

To remove $\log_\alpha$ from this requirement, we introduce function Ψ and define Φ of the form

$$\Phi.x = \beta \log_\alpha \Psi.x,$$

with $\beta > 0$. Constant β is introduced because there is no reason to assume that the base of the logarithm in the definition of Φ is equal to α. On account of the monotonicity of $\log_\alpha$ $(\alpha > 1)$, the above requirement on Φ thus becomes

$$\alpha \left(\frac{\Psi.(y \dashv \langle a, \{x\} \cup t\rangle)}{\Psi.(y \dashv x \dashv \langle a, t\rangle)}\right)^\beta \le \frac{\#\langle a, \{x\} \cup t\rangle}{\#\langle a, t\rangle}.$$

Defining $\sharp t = \#\langle a, t\rangle$, hence $\sharp t = 1 + (\Sigma\, x : x \in t : \#x)$, and writing x as $\langle b, u\rangle$, this may be reformulated as:

$$\alpha \left(\frac{\Psi.(y \dashv \langle a, \{\langle b, u\rangle\} \cup t\rangle)}{\Psi.(y \dashv \langle b, u \cup \{\langle a, t\rangle\}\rangle)}\right)^\beta \le 1 + \frac{\sharp u}{\sharp t}.$$

Since y does not occur in the right-hand side, we want to eliminate y from the left-hand side as well. To cancel out y in the fraction we introduce

$$\Delta.y.x = \Psi.(y \dashv x)/\Psi.x,$$

so that

$$\frac{\Psi.(y \dashv \langle a, \{\langle b, u\rangle\} \cup t\rangle)}{\Psi.(y \dashv \langle b, u \cup \{\langle a, t\rangle\}\rangle)} = \frac{\Delta.y.\langle a, \{\langle b, u\rangle\} \cup t\rangle}{\Delta.y.\langle b, u \cup \{\langle a, t\rangle\}\rangle} \; \frac{\Psi.\langle a, \{\langle b, u\rangle\} \cup t\rangle}{\Psi.\langle b, u \cup \{\langle a, t\rangle\}\rangle}.$$

The important observation is now that the left factor reduces to 1, if $\Delta.y.x$ depends on y and $\#x$ only. This proviso means formally that

(3) $\quad \#x = \#x' \;\Rightarrow\; \Delta.y.x = \Delta.y.x'.$

Since Δ is defined in terms of Ψ this imposes an extra requirement on Ψ to which we return in a moment, but first we consider the right factor.

The arguments of Ψ in the numerator and denominator of the right factor have a lot structure in common, viz. the structure of t and u. To cancel out these common parts, our next step is to take a definition of Ψ of the following general form:

$$\begin{array}{lcl} \Psi.\langle a, t\rangle & = & \psi.t \;\Psi^*.t \\ \Psi^*.t & = & (\Pi\, x : x \in t : \Psi.x). \end{array}$$

The right factor then becomes

$$\frac{\Psi.\langle a,\{\langle b,u\rangle\}\cup t\rangle}{\Psi.\langle b,u\cup\{\langle a,t\rangle\}\rangle}=\frac{\psi.(\{\langle b,u\rangle\}\cup t)}{\psi.(u\cup\{\langle a,t\rangle\})}\;\frac{\psi.u}{\Psi^*.u}\;\frac{\Psi^*.u}{\psi.t}\;\frac{\Psi^*.t}{\Psi^*.t},$$

which yields as requirement in terms of α, β, and ψ:

$$\alpha\left(\frac{\psi.(\{\langle b,u\rangle\}\cup t)}{\psi.(u\cup\{\langle a,t\rangle\})}\;\frac{\psi.u}{\psi.t}\right)^{\beta}\leq 1+\frac{\sharp u}{\sharp t}.$$

Since the right-hand side depends on $\sharp t$ and $\sharp u$ only, we decide to let $\psi.t$ depend on $\sharp t$ in the simplest possible way:

$$\psi.t=\sharp t.$$

With this choice for ψ, (3) indeed holds, as we leave to the reader to verify. (In Section 11.4, we will show that $\psi.t$ cannot be substantially smaller than $\sharp t$.) Since $\sharp(\{\langle b,u\rangle\}\cup t)=\sharp(u\cup\{\langle a,t\rangle\})$, the above requirement reduces to

$$\alpha\left(\frac{\sharp u}{\sharp t}\right)^{\beta}\leq 1+\frac{\sharp u}{\sharp t}.$$

As $\sharp t$ and $\sharp u$ range over all positive integers, α and β are thus required to satisfy: $\alpha>1$, $\beta>0$, and

$$(4)\qquad (\forall\, m,n :: \alpha\left(\frac{m}{n}\right)^{\beta}\leq 1+\frac{m}{n}).$$

To minimize the upper bound on $\mathcal{A}[a\angle\,]$, we maximize α under these constraints, and—with lower priority—we minimize β so as to keep Φ small. Assuming that $\alpha>1$ we observe for (4):

$$\begin{array}{ll}
 & (\forall\, m,n :: \alpha(\frac{m}{n})^{\beta}\leq 1+\frac{m}{n})\\
\Rightarrow & \{\ \text{take } n=1\ \}\\
 & (\forall\, m :: \alpha\leq \frac{m+1}{m^{\beta}})\\
\Rightarrow & \{\ 1<\alpha\ \}\\
 & (\forall\, m :: 1< \frac{m+1}{m^{\beta}})\\
\Rightarrow & \{\ \text{take } m\to\infty\ \}\\
 & \beta<1,
\end{array}$$

hence β must be smaller than 1. Therefore we assume $0<\beta<1$, and we derive:

$$\begin{array}{ll}
 & (\forall\, m,n :: \alpha(\frac{m}{n})^{\beta}\leq 1+\frac{m}{n})\\
\equiv & \{\ p=\frac{m}{n}\ \}\\
 & (\forall\, p : p>0 : \alpha\leq \frac{p+1}{p^{\beta}})\\
\equiv & \{\ \frac{x+1}{x^{\beta}} \text{ is minimal at } x=\frac{\beta}{1-\beta}\text{, which is positive because } 0<\beta<1\ \}\\
 & \alpha\leq \frac{1}{\beta^{\beta}(1-\beta)^{1-\beta}},
\end{array}$$

which results in the following constraints on α and β:

$$\begin{aligned} &1 < \alpha \leq \frac{1}{\beta^\beta(1-\beta)^{1-\beta}} \\ &0 < \beta < 1. \end{aligned}$$

To maximize α, we determine the maximum of its upper bound over all β between 0 and 1. This yields 2 as maximal value for α at $\beta = \frac{1}{2}$.

On account of (2), we thus have

$$\mathcal{A}[a\angle\,](y \dashv \langle a,t\rangle) \leq \log_2 \frac{\#(y \dashv \langle a,t\rangle)}{\#\langle a,t\rangle},$$

which implies

$$\mathcal{A}[a\angle\,](x) \leq \log_2 \#x.$$

Moreover, we have obtained as potential:

$$\Phi.\langle a,t\rangle = \tfrac{1}{2}\log_2 \#\langle a,t\rangle + (\Sigma\, x : x \in t : \Phi.x),$$

for which we have as bounds:

$$0 \leq \Phi.x \leq \tfrac{1}{2}\#x \log_2 \#x.$$

Note that $\Phi.x$ is $\Omega(\#x \log \#x)$ if each node of x has at most one subtree.

11.1.3 Top-down analysis

From the top-down analysis of bottom-up skew heaps in Section 9.4.3, we learn that the analysis of a bottom-up operation on trees may be simplified by considering a top-down "simulation". In this section we show that the calculations are simplified somewhat by analyzing the following program for path reversal:

$$\begin{array}{lllll} \text{(i)} & a\angle\,\langle a,t\rangle & = & \langle a,t\rangle & \\ \text{(ii)} & a\angle\,\langle b, u \cup \{x\}\rangle & \doteq & \langle a, \{\langle b,u\rangle\} \cup t\rangle \;\; [\![\; \langle a,t\rangle = a\angle\, x \;]\!] & , a \in x. \end{array}$$

This top-down program yields the same result as the bottom-up program, using the same number of unfoldings. The use of $\in$, however, makes this program not realistic.

To obtain a recurrence relation for $\mathcal{A}[a\angle\,]$, we take a potential of the form

$$\begin{aligned} \Phi.\langle a,t\rangle &= \varphi.t + \Phi^*.t \\ \Phi^*.t &= (\Sigma\, x : x \in t : \Phi.x), \end{aligned}$$

and derive for case (ii):

$$
\begin{array}{cl}
& \mathcal{A}[a\angle\,](\langle b, u \cup \{x\}\rangle) \\
= & \{\text{ definition of } \mathcal{A}[a\angle\,] \ \} \\
& \mathcal{T}[a\angle\,](\langle b, u \cup \{x\}\rangle) + \Phi.(a\angle\,\langle b, u \cup \{x\}\rangle) - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \{\text{ definition of } a\angle \ \} \\
& 1 + \mathcal{T}[a\angle\,](x) + \Phi.\langle a, \{\langle b, u\rangle\} \cup t\rangle - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \{\text{ definition of } \mathcal{A}[a\angle\,];\ a\angle\,x = \langle a, t\rangle \ \} \\
& 1 + \mathcal{A}[a\angle\,](x) + \Phi.x - \Phi.\langle a, t\rangle + \Phi.\langle a, \{\langle b, u\rangle\} \cup t\rangle - \Phi.\langle b, u \cup \{x\}\rangle \\
= & \{\text{ definition of } \Phi \ \} \\
& 1 + \mathcal{A}[a\angle\,](x) + \Phi.x - \varphi.t - \Phi^*.t + \varphi.(\{\langle b, u\rangle\} \cup t) + \varphi.u + \Phi^*.u + \Phi^*.t \\
& -\varphi.(u \cup \{x\}) - \Phi^*.u - \Phi.x \\
= & \{ \ \} \\
& \mathcal{A}[a\angle\,](x) + 1 - \varphi.t + \varphi.(\{\langle b, u\rangle\} \cup t) + \varphi.u - \varphi.(u \cup \{x\}).
\end{array}
$$

Observing that $\sharp(\{\langle b, u\rangle\} \cup t)$ is equal to $\sharp(u \cup \{x\})$ because $\#x = \#(a\angle\,x) = \sharp t$, the recurrence relation reduces to

$$
\begin{array}{lcl}
\mathcal{A}[a\angle\,](\langle a, t\rangle) & = & 0 \\
\mathcal{A}[a\angle\,](\langle b, u \cup \{x\}\rangle) & = & \mathcal{A}[a\angle\,](x) + 1 + \varphi.u - \varphi.t,
\end{array}
$$

provided $\varphi.t$ depends on $\sharp t$ only.

From this recurrence relation, it follows by induction that

$$\mathcal{A}[a\angle\,](x) \le \log_\alpha \#x,$$

with $\alpha > 1$, provided φ satisfies

$$1 + \varphi.u - \varphi.t \le \log_\alpha \frac{\#\langle b, u \cup \{x\}\rangle}{\#x}.$$

To remove $\log_\alpha$ from this requirement we define

$$\varphi.t = \beta \log_\alpha \sharp t,$$

with $\beta > 0$, so that $\varphi.t$ depends on $\sharp t$ in a simple way. This yields

$$\alpha \left(\frac{\sharp u}{\sharp t}\right)^\beta \le \frac{\sharp u + \sharp t}{\sharp t},$$

since $\#x = \sharp t$. The analysis can now be completed as in the previous section, which leads to the conclusion that

$$\mathcal{A}[a\angle\,](x) \le \log_2 \#x.$$

Remark 11.1
In the bottom-up analysis we used the following induction hypothesis:

$$\mathcal{A}[a\angle\,](y, \langle a, t\rangle) \le \log_2 \frac{\#(y, \langle a, t\rangle)}{\#\langle a, t\rangle}.$$

This stronger result can also be obtained in a top-down fashion by taking it as induction hypothesis.

□

11.1.4 Series of path reversals

We want to determine the cost of a series of path reversals performed on the same tree. Therefore, we study $\mathcal{T}[\angle^*](s,x)$, with operation $\angle^*$ performing a series of reversals:

$$\begin{array}{lll} [\,]\angle^* x & = & x \\ (s \dashv a)\angle^* x & = & s\angle^*(a\angle\, x). \end{array}$$

To amortize the cost of the applications of $\angle$, we introduce amortized costs for $\angle^*$:

$$\mathcal{A}[\angle^*](s,x) = \mathcal{T}[\angle^*](s,x) + \Phi.(s\angle^* x) - \Phi.x,$$

for which we have as recurrence relation:

$$\begin{array}{lll} \mathcal{A}[\angle^*]([\,],x) & = & 0 \\ \mathcal{A}[\angle^*](s \dashv a, x) & = & \mathcal{A}[\angle^*](s, a\angle\, x) + \mathcal{A}[\angle\,](a,x). \end{array}$$

Since $\mathcal{A}[a\angle\,](x) \le \log_2 \#x$, this yields by induction:

$$\mathcal{A}[\angle^*](s,x) \le \#s \log_2 \#x,$$

from which we conclude

$$\begin{array}{ll} & \mathcal{T}[\angle^*](s,x) \\ = & \{\ \text{definition of } \mathcal{A}[\angle^*]\ \} \\ & \mathcal{A}[\angle^*](s,x) + \Phi.x - \Phi.(s\angle^* x) \\ \le & \{\ \text{above bound for } \mathcal{A}[\angle^*]\text{; bounds for } \Phi\ \} \\ & \#s \log_2 \#x + \frac{1}{2}\#x \log_2 \#x. \end{array}$$

In [12], one is interested in the average cost per reversal when the number of reversals is significantly larger than the size of the tree. For this quantity we obtain the same bound as in [12]:

$$\frac{\mathcal{T}[\angle^*](s,x)}{\#s} \le \log_2 \#x + \frac{\#x \log_2 \#x}{2\#s}.$$

Note that the average cost of one reversal tends to the amortized cost of one reversal when the number of reversals ($= \#s$) is large. Phrased differently: the potential difference over a whole sequence of reversals is negligible when the sequence is sufficiently long.

11.2 Splaying

Splaying is the basic operation in a particular implementation of so-called *dictionaries*. A dictionary is an algebra involving subsets of an infinite, linearly ordered set (see e.g. [1, p.108]). As a simple example of a dictionary, we consider the following algebra:

$$\mathsf{D} = (\,\{\text{Int}\}\ \mid\ \{\,\}, (=\{\,\}), \in, \oplus, \ominus\,).$$

In this algebra, set Int is used as representative of an infinite, linearly ordered set. Another example of such a set is the lexicographically ordered set of identifiers of a programming language. (In that case one often speaks of "symbol tables" instead of dictionaries.)

If we replace Int in the definition of D by a finite set like $[0..N)$, the resulting algebra can be implemented very efficiently, as shown in Chapter 7. The fact that D involves sets of unbounded size forms a major obstacle that essentially excludes the use of (finite) arrays.

11.2.1 Splay trees

In [29], Sleator and Tarjan have developed an efficient implementation of dictionaries, called *splay trees*. Although the main subject here is the splaying operation, we will briefly describe splay trees in this section by presenting a refinement ST of algebra D with the following signature:

$$\mathsf{ST} = (\,BST\ \mid\ \mathsf{empty}, \mathsf{isempty}, \mathsf{member}, \mathsf{insert}, \mathsf{delete}\,).$$

Type BST is a subset of $\langle\text{Int}\rangle$, known as the set of *binary search trees*. It is defined by

$$BST = \{x \mid x \in \langle\text{Int}\rangle\ \wedge\ \overline{x} \text{ is increasing}\},$$

where $\overline{x}$ denotes the inorder traversal of binary tree x (see Section 1.1.4). Since an increasing list does not contain duplicates, each integer can occur at most once in a binary search tree. Consequently, a subtree may be identified by the value attached to its root, a fact that will be exploited in the next section. The abstraction function is defined by $[\![x]\!] = \{a \mid a \in x\}$.

In terms of operation $\angle$ ("splay") we now present programs for the operations of ST. The type of operation $\angle$ is $(\{-\infty\} \cup \text{Int} \cup \{\infty\}) \times BST \to BST$, and it is specified by

$$a\angle\langle\,\rangle = \langle\,\rangle,$$

and, for nonempty x:

$$(5)\qquad \overline{a\angle x} = \overline{x}\ \wedge\ (\exists\, t, b, u : a\angle x = \langle t, b, u\rangle : \overline{t} \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \overline{u} \text{ is increasing}).$$

So, while keeping the inorder traversal intact, splaying moves a value to the root which is either equal to a or "close to a". Note that for nonempty x, $\infty\angle x$ results in a tree whose root contains the maximum of x, and whose right subtree is empty. This property is exploited in the program for delete:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \langle\,\rangle \\ \mathsf{isempty}.x & \doteq & x = \langle\,\rangle \end{array}$$

$$
\begin{array}{lcl}
\mathsf{member}.a.x & \doteq & x := a\angle\, x \\
 & & ;\ (\ \text{return(false)} \qquad\qquad\qquad\qquad ,x = \langle\ \rangle \\
 & & \quad [\!]\ \text{return}(a{=}b)\ \ |[\ \langle t,b,u\rangle = x\ ]|\ \ \ ,x \neq \langle\ \rangle \\
 & & \quad) \\
\mathsf{insert}.x.a & \doteq & \langle a\rangle \qquad\qquad\qquad\qquad\qquad ,x = \langle\ \rangle \\
 & [\!] & (\ \langle t, a, \langle\langle\ \rangle, b, u\rangle\rangle \quad ,a < b \\
 & & [\!]\ \langle\langle t, b, \langle\ \rangle\rangle, a, u\rangle \quad ,b < a \\
 & &)\ \ |[\ \langle t,b,u\rangle = a\angle\, x\ ]| \quad ,x \neq \langle\ \rangle \\
\mathsf{delete}.x.a & \doteq & (\ u \qquad\qquad\qquad\qquad\qquad\qquad\qquad ,t = \langle\ \rangle \\
 & & [\!]\ \langle v,b,u\rangle\ \ |[\ \langle v,b,\langle\ \rangle\rangle = \infty\angle\, t\ ]|\ \ \ ,t \neq \langle\ \rangle \\
 & &)\ \ |[\ \langle t,a,u\rangle = a\angle\, x\ ]|.
\end{array}
$$

The program for member is an example of a program with benevolent side-effects. For the analysis we introduce program member′ (cf. Sections 4.7 and 5.5):

$$
\begin{array}{lcll}
\mathsf{member}'.a.x & = & (\text{false}, x) & ,x = \langle\ \rangle \\
 & [\!] & (a{=}b, \langle t,b,u\rangle)\ \ |[\ \langle t,b,u\rangle = a\angle\, x\ ]| & ,x \neq \langle\ \rangle.
\end{array}
$$

Remark 11.2
Starting from the definition of data refinement, the obvious implementation of $\in$ which exploits the fact that the concrete trees are binary search trees would be program f:

$$
\begin{array}{lcll}
f.a.\langle\ \rangle & = & \text{false} & \\
f.a.\langle t,b,u\rangle & = & f.a.t & ,a < b \\
 & [\!] & \text{true} & ,a = b \\
 & [\!] & f.a.u & ,a > b.
\end{array}
$$

But the performance of this program is poor: the evaluation of $f.a.x$ may take time proportional to $\#[\![x]\!]$ in the worst case, since trees in BST are not required to be balanced. To achieve logarithmic amortized costs for member, it is crucial that this operation transforms its tree-argument by means of operation $\angle$.
□

With member′ producing a tree, we can now associate a potential change with this "inspection" operation. The amortized costs are thus defined as follows:

$$
\begin{array}{lcl}
\mathcal{A}[\mathsf{empty}] & = & \mathcal{T}[\mathsf{empty}] + \Phi.\mathsf{empty} \\
\mathcal{A}[\mathsf{isempty}](x) & = & \mathcal{T}[\mathsf{isempty}](x) \\
\mathcal{A}[\mathsf{member}'](a,x) & = & \mathcal{T}[\mathsf{member}'](a,x) + \Phi.(\mathsf{member}'.a.x.1) - \Phi.x \\
\mathcal{A}[\mathsf{insert}](x,a) & = & \mathcal{T}[\mathsf{insert}](x,a) + \Phi.(\mathsf{insert}.x.a) - \Phi.x \\
\mathcal{A}[\mathsf{delete}](x,a) & = & \mathcal{T}[\mathsf{delete}](x,a) + \Phi.(\mathsf{delete}.x.a) - \Phi.x,
\end{array}
$$

where $\mathcal{T}$ is an appropriate cost measure and Φ a potential. The first two operations evidently have $O(1)$ amortized costs, and splaying is implemented such that the amortized cost of each of the other operations is logarithmic.

11.2.2 Definition of splaying

In this section we introduce a simplified version of splaying which is somewhat easier to analyze. The results obtained from the analysis of this version, which are presented in the next section, are, however, sufficiently general.

We present two programs for the following restricted version of splaying, which has type $\mathrm{Int}\times BST \curvearrowright BST$, and satisfies (cf. (5)):

$$(6) \quad a \in x \;\Rightarrow\; \overline{a\angle\, x} = \overline{x} \;\wedge\; (\exists\, t, u :: a\angle\, x = \langle t, a, u\rangle).$$

So, we assume that a occurs in x, and therefore $a\angle\, x$ has root a.

Operation $(a\angle\,)$ ("splaying at a") is performed by "rotating" a towards the root, and it is the particular way these rotations are chosen that leads to an efficient implementation of splaying. Traditionally, splaying is explained by means of pictures (see Figure 11.2, transformation (ii) is called a *rotation at* a), but to allow for a systematic analysis of splaying, we introduce a special representation of binary trees so that we can describe splaying concisely and in a linear way—as opposed to a two-dimensional pictorial description. The appropriate representation to describe splaying is root-path view $\langle\mathrm{Int}\rangle_3$ from Section 6.2.3:

$$\langle\mathrm{Int}\rangle_3 = [\langle\mathrm{Int}\rangle\times\mathrm{Int} \cup \mathrm{Int}\times\langle\mathrm{Int}\rangle] \times \langle\mathrm{Int}\rangle.$$

In terms of this type, we obtain the following program for $\angle$ (cf. Figure 11.2):

$$\begin{array}{lll}
\text{(i)} & a\angle\,\langle t, a, u\rangle & = \langle t, a, u\rangle \\
\text{(ii)} & a\angle\,\langle\langle t, a, u\rangle, b, v\rangle & = \langle t, a, \langle u, b, v\rangle\rangle \\
\text{(iii)} & a\angle\,(y, \langle\langle\langle t, a, u\rangle, b, v\rangle, c, w\rangle) & = a\angle\,(y, \langle t, a, \langle u, b, \langle v, c, w\rangle\rangle\rangle) \\
\text{(iv)} & a\angle\,(y, \langle\langle v, b, \langle t, a, u\rangle\rangle, c, w\rangle) & = a\angle\,(y, \langle\langle v, b, t\rangle, a, \langle u, c, w\rangle\rangle),
\end{array}$$

where the symmetrical counterparts of (ii), (iii), and (iv), which are obtained by interchanging the role of left and right subtrees, have been omitted. (In the sequel, "symmetry" refers to this kind of symmetry.) In this program we have ignored the fact that the position of a is actually determined by starting a search at the root (as in program f in Remark 11.2); this is safe because the cost of this search is proportional to the cost of moving a to the root.

It is easily seen that this program satisfies specification (6). Note that in cases (i), (ii), and (iv) the right-hand side is fully determined by the left-hand side provided one abstains from further decomposition of the components occurring in the left-hand sides. In case (iii), however, $a\angle\,(y, \langle t, a, \langle\langle u, b, v\rangle, c, w\rangle\rangle)$ is the only remaining alternative for $a\angle\,(y, \langle t, a, \langle u, b, \langle v, c, w\rangle\rangle\rangle)$. But inspection of the resulting program learns that it is equivalent to rotating a towards the root by means of single rotations only:

$$\begin{array}{ll}
a\angle\,\langle t, a, u\rangle & = \langle t, a, u\rangle \\
a\angle\,(y, \langle\langle t, a, u\rangle, b, v\rangle) & = a\angle\,(y, \langle t, a, \langle u, b, v\rangle\rangle) \\
a\angle\,(y, \langle v, b, \langle u, a, t\rangle\rangle) & = a\angle\,(y, \langle\langle v, b, u\rangle, a, t\rangle),
\end{array}$$

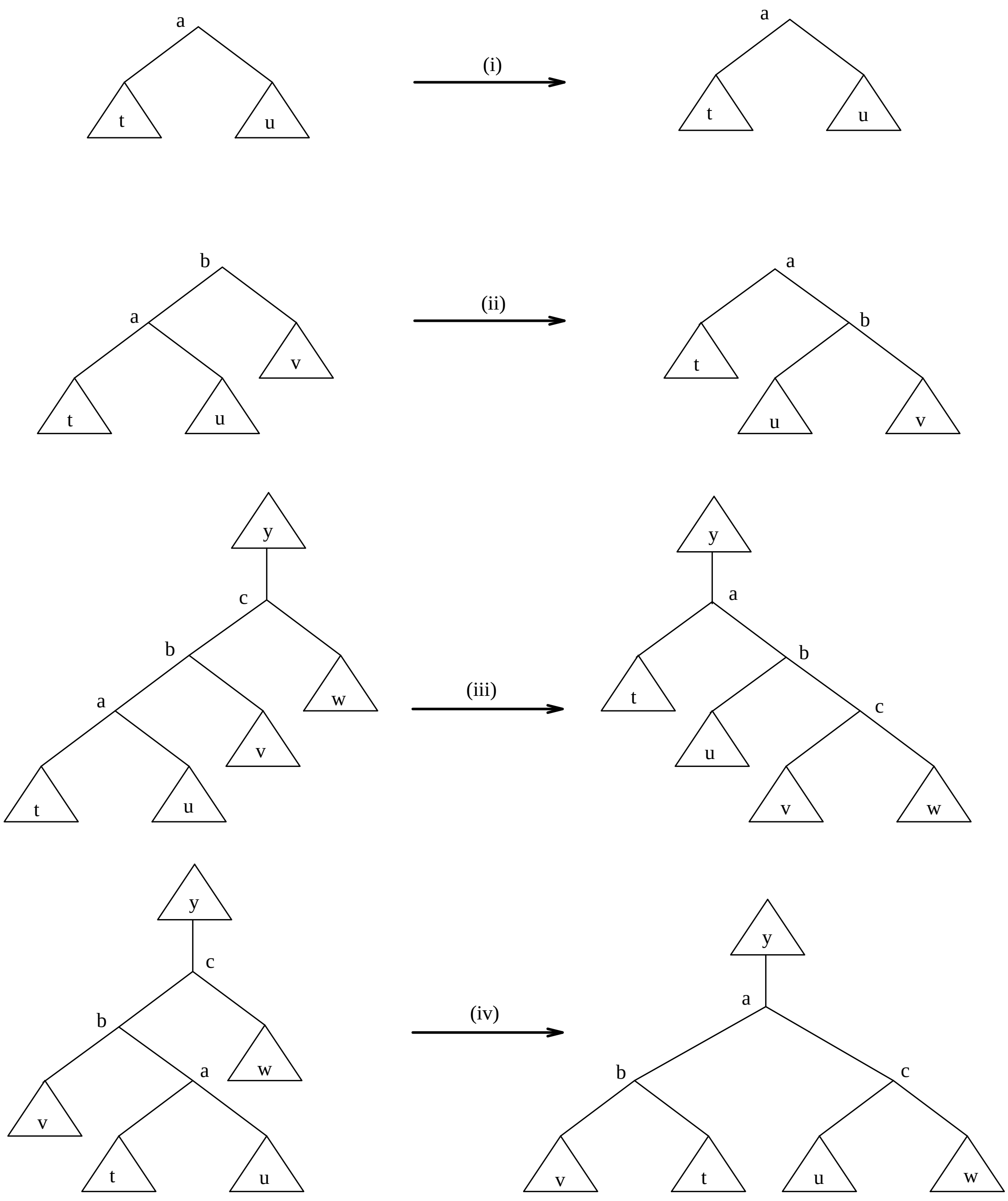

Figure 11.2: Splaying at a (cf. [29, Figure 3]).

and, consequently, the efficiency of this program is poor. (See also Remark 11.3.) The program merely distinguishes the cases "depth of a is 0" and "depth of a is greater than 0". To achieve the desired efficiency it is necessary to single out the case "depth of a is 1" and to act in the particular way described above in case "depth of a is greater than 1".

Instead of splaying in a bottom-up fashion as depicted in Figure 11.2, we may as well do it in a top-down fashion. In [29], Sleator and Tarjan present a rather complicated *iterative* method of top-down splaying. Doing this in a *recursive* way in terms of type $\langle\mathrm{Int}\rangle$, we obtain a much simpler description of top-down splaying:

$$\begin{array}{llcll}
\text{(i)} & a\angle\,\langle t,a,u\rangle & = & \langle t,a,u\rangle & \\
\text{(ii)} & a\angle\,\langle\langle t,a,u\rangle,b,v\rangle & = & \langle t,a,\langle u,b,v\rangle\rangle & ,a{<}b \\
\text{(iii)} & a\angle\,\langle\langle x,b,v\rangle,c,w\rangle & \ddot{=} & \langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle \;\;[\![\, \langle t,a,u\rangle = a\angle\, x \,]\!] & ,a{<}b{<}c \\
\text{(iv)} & a\angle\,\langle\langle v,b,x\rangle,c,w\rangle & \ddot{=} & \langle\langle v,b,t\rangle,a,\langle u,c,w\rangle\rangle \;\;[\![\, \langle t,a,u\rangle = a\angle\, x \,]\!] & ,b{<}a{<}c.
\end{array}$$

In this program the same transformations are performed as in Figure 11.2, but if the depth of a is odd, the result is a bit different. Since algebra T from Section 4.3 can be implemented in a nondestructive way, a purely-functional implementation of splay trees can be obtained from this program for $\angle$.

The cost measure defined by the dots in this program will be denoted by $\mathcal{T}$. Then $\mathcal{T}[a\angle\,](x)$ is approximately equal to the number of integer comparisons required for the evaluation of $a\angle\, x$ ($\mathcal{T}[a\angle\,](x)+2$ is a tight upper bound).

11.2.3 Analysis of top-down splaying

The analysis of top-down splaying follows the top-down analysis of path reversal very closely. To avoid too much duplication of that analysis, the derivation of the potential is therefore shortened a bit by drawing some conclusions from that analysis.

We try to derive a logarithmic bound for the amortized cost of $(a\angle\,)$, given by

$$\mathcal{A}[a\angle\,](x) = \mathcal{T}[a\angle\,](x) + \Phi.(a\angle\, x) - \Phi.x.$$

More precisely, we want to find a potential Φ such that

$$(7)\quad \mathcal{A}[a\angle\,](x) \le \log_\alpha \sharp x,$$

with $\alpha{>}1$ and as large as possible. Recall that $\sharp x$ denotes the size of x plus one, which may defined recursively by

$$\begin{array}{lcl}
\sharp\langle\,\rangle & = & 1 \\
\sharp\langle t,a,u\rangle & = & \sharp t + \sharp u.
\end{array}$$

In addition, we want Φ to be nonnegative and small. Since we have omitted the symmetrical counterparts of cases (ii)–(iv) we also require Φ to be symmetric in order to cover these cases.

Setting out for an inductive argument, we first derive a recurrence relation for $\mathcal{A}[a\angle\,]$. To this end, we take Φ of the form:

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t + \varphi.t.u + \Phi.u. \end{array}$$

Then we observe for case (iii):

$$\begin{array}{cl} & \mathcal{A}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle) \\ = & \{\ \text{definition of } \mathcal{A}[a\angle\,]\ \} \\ & \mathcal{T}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle) + \Phi.(a\angle\,\langle\langle x,b,v\rangle,c,w\rangle) - \Phi.\langle\langle x,b,v\rangle,c,w\rangle \\ = & \{\ \text{definition of } a\angle\ \} \\ & 2 + \mathcal{T}[a\angle\,](x) + \Phi.\langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle - \Phi.\langle\langle x,b,v\rangle,c,w\rangle \\ = & \{\ \text{definition of } \mathcal{A}[a\angle\,];\ a\angle\,x = \langle t,a,u\rangle\ \} \\ & 2 + \mathcal{A}[a\angle\,](x) + \Phi.x - \Phi.\langle t,a,u\rangle \\ & +\Phi.\langle t,a,\langle u,b,\langle v,c,w\rangle\rangle\rangle - \Phi.\langle\langle x,b,v\rangle,c,w\rangle \\ = & \{\ \text{definition of } \Phi\ \} \\ & 2 + \mathcal{A}[a\angle\,](x) + \Phi.x - \Phi.t - \varphi.t.u - \Phi.u \\ & +\Phi.t + \varphi.t.\langle u,b,\langle v,c,w\rangle\rangle + \Phi.u + \varphi.u.\langle v,c,w\rangle + \Phi.v + \varphi.v.w + \Phi.w \\ & -\Phi.x - \varphi.x.v - \Phi.v - \varphi.\langle x,b,v\rangle.w - \Phi.w \\ = & \{\ \ \} \\ & 2 + \mathcal{A}[a\angle\,](x) - \varphi.t.u + \varphi.t.\langle u,b,\langle v,c,w\rangle\rangle + \varphi.u.\langle v,c,w\rangle \\ & +\varphi.v.w - \varphi.x.v - \varphi.\langle x,b,v\rangle.w. \end{array}$$

As in the analysis of path reversal, the recurrence relation can be simplified by observing that $\sharp t + \sharp\langle u,b,\langle v,c,w\rangle\rangle = \sharp\langle x,b,v\rangle + \sharp w$ because $\sharp(a\angle\,x) = \sharp x$ and $a\angle\,x = \langle t,a,u\rangle$. Hence, provided $\varphi.t.u$ depends on $\sharp t + \sharp u$ only, the amortized cost of an unfolding of case (iii) simplifies to

$$2 + \varphi.u.\langle v,c,w\rangle + \varphi.v.w - \varphi.x.v - \varphi.t.u.$$

Omitting the calculations for the other cases, we thus have

$$\begin{array}{lcl} \mathcal{A}[a\angle\,](\langle t,a,u\rangle) & = & 0 \\ \mathcal{A}[a\angle\,](\langle\langle t,a,u\rangle,b,v\rangle) & = & \varphi.u.v - \varphi.t.u \\ \mathcal{A}[a\angle\,](\langle\langle x,b,v\rangle,c,w\rangle) & = & \mathcal{A}[a\angle\,](x)+2+\varphi.u.\langle v,c,w\rangle+\varphi.v.w-\varphi.x.v-\varphi.t.u \\ \mathcal{A}[a\angle\,](\langle\langle v,b,x\rangle,c,w\rangle) & = & \mathcal{A}[a\angle\,](x)+2+\varphi.v.t+\varphi.u.w-\varphi.v.x-\varphi.t.u. \end{array}$$

It is obvious that (7) holds in case (i). In order that (7) follows by induction in the other cases, the following requirements must be satisfied:

$$\begin{array}{lcl} \varphi.u.v - \varphi.t.u & \le & \log_\alpha(\sharp t + \sharp u + \sharp v) \\ 2 + \varphi.u.\langle v,c,w\rangle + \varphi.v.w - \varphi.x.v - \varphi.t.u & \le & \log_\alpha \frac{\sharp t+\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u} \\ 2 + \varphi.v.t + \varphi.u.w - \varphi.v.x - \varphi.t.u & \le & \log_\alpha \frac{\sharp t+\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u}. \end{array}$$

To remove $\log_\alpha$ from these requirements, we define

$$\varphi.t.u = \beta \log_\alpha(\sharp t + \sharp u),$$

with $\beta > 0$. The requirements then reduce to

$$\begin{array}{rcl}
\left(\frac{\sharp u+\sharp v}{\sharp t+\sharp u}\right)^\beta & \leq & \sharp t + \sharp u + \sharp v \\
\alpha^2 \left(\frac{\sharp u+\sharp v+\sharp w}{\sharp t+\sharp u+\sharp v}\frac{\sharp v+\sharp w}{\sharp t+\sharp u}\right)^\beta & \leq & 1 + \frac{\sharp v+\sharp w}{\sharp t+\sharp u} \\
\alpha^2 \left(\frac{\sharp v+\sharp t}{\sharp v+\sharp t+\sharp u}\frac{\sharp u+\sharp w}{\sharp t+\sharp u}\right)^\beta & \leq & 1 + \frac{\sharp v+\sharp w}{\sharp t+\sharp u}.
\end{array}$$

As constraints on α and β we thus have: $\alpha > 1$, $\beta > 0$, and

(8) $\quad (\forall\, k, l, m :: (\dfrac{l+m}{k+l})^\beta \leq k + l + m)$

(9) $\quad (\forall\, k, l, m, n :: \alpha^2(\dfrac{l+m+n}{k+l+m}\;\dfrac{m+n}{k+l})^\beta \leq 1 + \dfrac{m+n}{k+l})$

(10) $\quad (\forall\, k, l, m, n :: \alpha^2(\dfrac{k+m}{k+l+m}\;\dfrac{l+n}{k+l})^\beta \leq 1 + \dfrac{m+n}{k+l})$.

Under these constraints, we want to maximize α and, with lower priority, we want to minimize β so as to keep Φ small.

We distinguish two cases.

Case $\beta < \frac{1}{2}$. Lemma 11.5 at the end of this section states that for $0 < \beta < \frac{1}{2}$:

$$\begin{array}{lcl}
(8) & \equiv & \beta \leq 1 \\
(9) & \equiv & \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}} \\
(10) & \equiv & \alpha^2 \leq 4^\beta,
\end{array}$$

so, taking the conjunction of the requirements on α and β, we obtain:

$$1 < \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}} \min 4^\beta$$

$$0 < \beta < \frac{1}{2}.$$

To maximize α, we determine the maximum of its upper bound over all β satisfying $0 < \beta < \frac{1}{2}$. This yields $\sqrt[3]{4}$ as maximal value for α^2 for $\beta = \frac{1}{3}$ (see Figure 11.3).

Case $\beta \geq \frac{1}{2}$. Instantiation of (9) with "$k, l, m, n := 1, 1, 1, \infty$" yields that $\alpha^2 \leq \sqrt{\frac{3}{2}}$.

Since $\sqrt{\frac{3}{2}} < \sqrt[3]{4}$, we conclude from this case analysis that the optimal values for α and β are given by $\alpha = \sqrt[3]{2}$ and $\beta = \frac{1}{3}$.

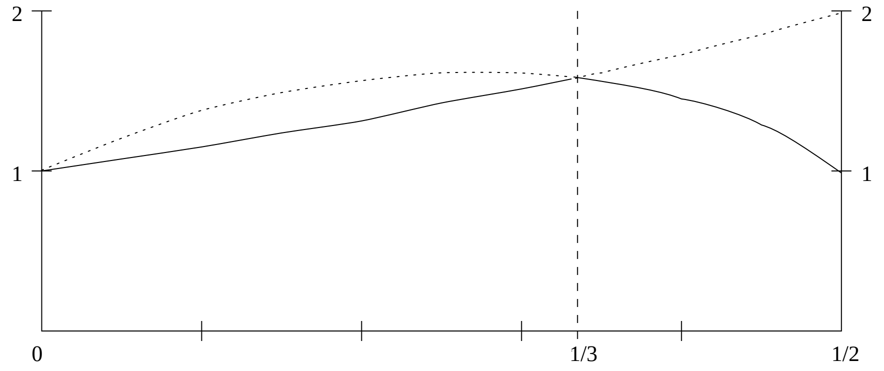

Figure 11.3: Maximum at $\beta = \frac{1}{3}$.

As result for the amortized costs of splaying we thus have (cf. (7)):

$$\mathcal{A}[a\angle\,](x) \leq 1 + 3\log_2 \sharp x.$$

Furthermore, we have derived the following potential:

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t,a,u\rangle & = & \Phi.t + \log_2 \sharp\langle t,a,u\rangle + \Phi.u. \end{array}$$

Note that $\Phi.x$ is $\Omega(\sharp x \log \sharp x)$ if each node of x has at most one nonempty subtree. It would be interesting to know whether there exists a linear potential for splaying; this question will be addressed in Section 11.4.

We end this section with two remarks and the promised lemma.

Remark 11.3
The remaining alternative for (iii) mentioned in Section 11.2.2,

$$a\angle\,\langle\langle x,b,v\rangle,c,w\rangle \doteqdot \langle t,a,\langle\langle u,b,v\rangle,c,w\rangle\rangle \quad [\![\ \langle t,a,u\rangle = a\angle\, x\ ]\!] \quad ,a{<}b{<}c,$$

drops out because it gives rise to constraint

$$(\forall\, k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\ \frac{l+m}{k+l})^\beta \leq 1 + \frac{m+n}{k+l}).$$

This implies that $\alpha \leq 1$ if we instantiate it with "$k,l,m,n := 1,\infty,1,1$", which conflicts with $\alpha > 1$.
□

Remark 11.4
The maximal value of $\frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}$ turns out to be $\phi = \frac{\sqrt{5}+1}{2}$, the golden ratio again (for $\beta = \frac{5-\sqrt{5}}{10}$ (≈ 0.276)). So, if it were only for case (iii), constant 3 in the bound for $\mathcal{A}[a\angle\,]$ could be replaced by $2/\log_2 \phi$ (≈ 2.88).
□

Lemma 11.5
For $0 < \beta < \frac{1}{2}$,

(a) $(\forall\, k, l, m :: \left(\frac{l+m}{k+l}\right)^{\beta} \leq k + l + m) \;\equiv\; \beta \leq 1$

(b) $(\forall\, k, l, m, n :: \alpha^2 \left(\frac{l+m+n}{k+l+m}\ \frac{m+n}{k+l}\right)^{\beta} \leq 1 + \frac{m+n}{k+l}) \;\equiv\; \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^{\beta}(1-2\beta)^{1-2\beta}}$

(c) $(\forall\, k, l, m, n :: \alpha^2 \left(\frac{k+m}{k+l+m}\ \frac{l+n}{k+l}\right)^{\beta} \leq 1 + \frac{m+n}{k+l}) \;\equiv\; \alpha^2 \leq 4^{\beta}$.

Proof The facts are proven by mutual implication.
Proof of (a), using that $\beta > 0$:

$$
\begin{array}{ll}
 & (\forall\, k, l, m :: (\frac{l+m}{k+l})^{\beta} \leq k + l + m) \\
\Leftarrow & \{\ \beta > 0\ \} \\
 & (\forall\, k, l, m :: (\frac{k+l+m}{k+l})^{\beta} \leq \frac{k+l+m}{k+l}) \\
\equiv & \{\ \} \\
 & \beta \leq 1;
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall\, k, l, m :: (\frac{l+m}{k+l})^{\beta} \leq k + l + m) \\
\Rightarrow & \{\ \text{take } k = 1 \text{ and } l = 1\ \} \\
 & (\forall\, m :: (\frac{m+1}{2})^{\beta} \leq m + 2) \\
\Rightarrow & \{\ \text{take } m \to \infty\ \} \\
 & \beta \leq 1.
\end{array}
$$

Proof of (b), using that $0 < \beta < \frac{1}{2}$:

$$
\begin{array}{ll}
 & (\forall\, k, l, m, n :: \alpha^2(\frac{l+m+n}{k+l+m}\ \frac{m+n}{k+l})^{\beta} \leq 1 + \frac{m+n}{k+l}) \\
\Leftarrow & \{\ \beta > 0\ \} \\
 & (\forall\, k, l, m, n :: \alpha^2(\frac{k+l+m+n}{k+l}\ \frac{m+n}{k+l})^{\beta} \leq 1 + \frac{m+n}{k+l}) \\
\equiv & \{\ p = \frac{m+n}{k+l}\ \} \\
 & (\forall\, p : p > 0 : \alpha^2((1+p)p)^{\beta} \leq 1 + p) \\
\equiv & \{\ \} \\
 & (\forall\, p : p > 0 : \alpha^2 \leq \frac{(1+p)^{1-\beta}}{p^{\beta}}) \\
\equiv & \{\ \text{minimize } \frac{(1+x)^{1-\beta}}{x^{\beta}}, \text{ using } 0 < \beta < \frac{1}{2} \text{ (see below)}\ \} \\
 & \alpha^2 \leq \frac{(1-\beta)^{1-\beta}}{\beta^{\beta}(1-2\beta)^{1-2\beta}};
\end{array}
$$

$$
\begin{array}{ll}
& (\forall k,l,m,n :: \alpha^2(\frac{l+m+n}{k+l+m}\ \frac{m+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Rightarrow & \{\ \text{take } k=m=1\ \} \\
& (\forall l,n :: \alpha^2(\frac{l+n+1}{l+2}\ \frac{n+1}{l+1})^\beta \le 1+\frac{n+1}{l+1}) \\
\equiv & \{\ \ \} \\
& (\forall l,n :: \alpha^2 \le (1+\frac{n+1}{l+1})(\frac{l+2}{l+n+1}\ \frac{l+1}{n+1})^\beta) \\
\Rightarrow & \{\ \text{take } \frac{n}{l} \to \frac{\beta}{1-2\beta} \text{ and } l\to\infty\ (\frac{\beta}{1-2\beta}>0, \text{ since } 0<\beta<\frac{1}{2})\ \} \\
& \alpha^2 \le (1+\frac{\beta}{1-2\beta})(\frac{1}{1+\frac{\beta}{1-2\beta}}\ \frac{1-2\beta}{\beta})^\beta \\
\equiv & \{\ 1+\frac{\beta}{1-2\beta}=\frac{1-\beta}{1-2\beta}\ \} \\
& \alpha^2 \le \frac{1-\beta}{1-2\beta}(\frac{1-2\beta}{1-\beta}\ \frac{1-2\beta}{\beta})^\beta \\
\equiv & \{\ \ \} \\
& \alpha^2 \le \frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}.
\end{array}
$$

To complete the proof of this part, we minimize $f(x)=\frac{(1+x)^{1-\beta}}{x^\beta}$ over all $x>0$. Then $f'(x)=0$ is equivalent to $(1-\beta)x=(1+x)\beta$, and f turns out to be minimal at $\frac{\beta}{1-2\beta}$, which is positive because $0<\beta<\frac{1}{2}$. So the minimum of f equals $\frac{(1+\frac{\beta}{1-2\beta})^{1-\beta}}{(\frac{\beta}{1-2\beta})^\beta}$, which in turn equals $\frac{(1-\beta)^{1-\beta}}{\beta^\beta(1-2\beta)^{1-2\beta}}$, since $1+\frac{\beta}{1-2\beta}=\frac{1-\beta}{1-2\beta}$.

Proof of (c), using that $0<\beta<\frac{1}{2}$:

$$
\begin{array}{ll}
& (\forall k,l,m,n :: \alpha^2(\frac{k+m}{k+l+m}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Leftarrow & \{\ \beta>0\ \} \\
& (\forall k,l,m,n :: \alpha^2(\frac{k+m}{k+l}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\equiv & \{\ p=\frac{k+m}{k+l} \text{ and } q=\frac{l+n}{k+l}\ \} \\
& (\forall p,q : p>0\ \wedge\ q>0\ \wedge\ p+q\ge 1 : \alpha^2(pq)^\beta \le p+q) \\
\equiv & \{\ \ \} \\
& (\forall p,q : p>0\ \wedge\ q>0\ \wedge\ p+q\ge 1 : \alpha^2 \le \frac{p+q}{(pq)^\beta}) \\
\equiv & \{\ \text{minimize } \frac{x+y}{(xy)^\beta}, \text{ using } 0<\beta<\frac{1}{2} \text{ (see below)}\ \} \\
& \alpha^2 \le 4^\beta;
\end{array}
$$

$$
\begin{array}{ll}
& (\forall k,l,m,n :: \alpha^2(\frac{k+m}{k+l+m}\ \frac{l+n}{k+l})^\beta \le 1+\frac{m+n}{k+l}) \\
\Rightarrow & \{\ \text{take } l=k \text{ and } m=n=1\ \} \\
& (\forall k :: \alpha^2(\frac{k+1}{2k+1}\ \frac{k+1}{2k})^\beta \le 1+\frac{2}{2k}) \\
\equiv & \{\ \ \} \\
& (\forall k :: \alpha^2 \le (1+\frac{1}{k})(\frac{2k+1}{k+1}\ \frac{2k}{k+1})^\beta) \\
\Rightarrow & \{\ \text{take } k\to\infty\ \} \\
& \alpha^2 \le 4^\beta.
\end{array}
$$

We determine the minimum of $\frac{x+y}{(xy)^\beta}$ over all positive x and y satisfying $x+y \geq 1$. To this end, we first observe that $\frac{x+y}{(xy)^\beta}$ takes its minimal value only if $x = y$ because $(xy)^\beta$ is maximized by taking x equal to y ($\beta > 0$), and this can be achieved without changing the value of $x + y$. We thus minimize $2x^{1-2\beta}$ over all $x \geq \frac{1}{2}$. Since this function is increasing in x ($\beta < \frac{1}{2}$), it attains its minimal value of 4^β at $x = \frac{1}{2}$.
□

11.2.4 Bounds for splay trees

Cost measure $\mathcal{T}$ has been chosen such that $\mathcal{T}[a\angle\,](x)$ is (as good as equal to) the number of comparisons needed to locate a in x (cf. program f in Remark 11.2). This measure corresponds to Sleator and Tarjan's measure in [29]. The amortized costs satisfy

$$\begin{array}{lll}
\mathcal{A}[\mathsf{empty}] & \text{is} & O(1)\\
\mathcal{A}[\mathsf{isempty}](x) & \text{is} & O(1)\\
\mathcal{A}[\mathsf{member}](a,x) & \leq & 3\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons}\\
\mathcal{A}[\mathsf{insert}](x,a) & \leq & 4\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons}\\
\mathcal{A}[\mathsf{delete}](x,a) & \leq & 6\log_2(1{+}\#[\![x]\!]) + O(1) \text{ comparisons.}
\end{array}$$

For potential Φ we have as bounds

$$0 \leq \Phi.x \leq (1{+}\#[\![x]\!])\log_2(1{+}\#[\![x]\!]).$$

11.3 Pairing

In [7], Fredman, Sedgewick, Sleator, and Tarjan propose *pairing heaps* as practical alternative for Fibonacci heaps. The central operation of this data structure, called *pairing*, transforms a list of heaps into a single heap by repeatedly linking two heaps in a specific order. In this section we consider one of the numerous variants of pairing, viz. the so-called *two-pass variant*.

11.3.1 Pairing heaps

Pairing heaps can be used to implement the following algebra of priority queues, which is about the same as algebra PQ' from Chapter 10:

$$\mathsf{PQ}'' = (\,\{\mathrm{Int}\}\ |\ \{\,\}, (=\{\,\}), \{\cdot\}, \uplus, \downarrow, \Downarrow, \mathit{deckey}\,).$$

As signature for the implementation of this algebra we use

$$\mathsf{PH} = (\,P\ |\ \mathsf{empty}, \mathsf{isempty}, \mathsf{single}, \mathsf{union}, \mathsf{min}, \mathsf{delmin}, \mathsf{deckey}\,).$$

Characteristic of pairing heaps is that P is a subset of tree type

$$\langle \mathrm{Int} \rangle_9 = \{\langle\, \rangle\} \cup \langle \mathrm{Int} \rangle_4,$$

where type $\langle \mathrm{Int} \rangle_4$ has been defined in Section 6.3.1 as the smallest solution of

$$X : \quad X = \mathrm{Int} \times [X].$$

Note that for a tree $\langle a, t \rangle$ in $\langle \mathrm{Int} \rangle_9$, the trees in t are nonempty—as they are elements of $\langle \mathrm{Int} \rangle_4$. Data type P is now, informally, defined as the set of trees of type $\langle \mathrm{Int} \rangle_4$ that are free of duplicates and satisfy the heap condition.

Again, the fundamental operation on heaps is linking, which is defined on trees of type $\langle \mathrm{Int} \rangle_9$ by

$$\begin{array}{rclll} \langle\, \rangle \bowtie y & \doteq & y & & \\ x \bowtie \langle\, \rangle & \doteq & x & & \\ \langle a, t \rangle \bowtie \langle b, u \rangle & \doteq & \langle a, \langle b, u \rangle \vdash t \rangle & , a < b \\ & [] & \langle b, \langle a, t \rangle \vdash u \rangle & , b < a. \end{array}$$

In terms of this operator, the operations of PH are defined as follows:

$$\begin{array}{lcl} \mathsf{empty} & \doteq & \langle\, \rangle \\ \mathsf{isempty}.x & \doteq & x = \langle\, \rangle \\ \mathsf{single}.a & \doteq & \langle a, [\,] \rangle \\ \mathsf{union}.x.y & \doteq & x \bowtie y \\ \mathsf{min}.\langle a, t \rangle & \doteq & a \\ \mathsf{delmin}.\langle a, t \rangle & \doteq & cp.t. \end{array}$$

Function cp transforms a list of heaps into a single heap by repeatedly linking two heaps. This can be done in many ways. The method analyzed in [7] is the two-pass variant: in the first pass, heaps are linked *pairwise* from left to right; in the second pass, the resulting heaps are combined from right to left. These two passes are concisely encoded in the following program for cp ("pair and combine", or "pairing" for short).

$$\begin{array}{lcl} cp.[\,] & = & \langle\, \rangle \\ cp.[x] & = & x \\ cp.(x \vdash y \vdash t) & = & (x \bowtie y) \bowtie cp.t. \end{array}$$

Note that the cost of pairing is charged to the applications of $\bowtie$.

To program deckey, we use the following root-path view of $\langle \mathrm{Int} \rangle_9$:

$$[\,[\langle \mathrm{Int} \rangle_9] \times \mathrm{Int} \times [\langle \mathrm{Int} \rangle_9]\,] \times \langle \mathrm{Int} \rangle_9,$$

which corresponds to root-path view $\langle T \rangle_8$ of tree type $\langle T \rangle_4$ from Section 6.3.2. In terms of this type, deckey is programmed as follows:

$$\mathsf{deckey}.a.k.(z, \langle a, t \rangle) \doteq z \bowtie \langle a{-}k, t \rangle,$$

where we have abbreviated $(z, \langle\,\rangle)$ to z, and $([\,], \langle a{-}k, t\rangle)$ to $\langle a{-}k, t\rangle$.

The cost measure defined by the dots in the above programs will be called $\mathcal{T}$. As for the complexity of these operations, it is still an open problem whether the amortized cost of deckey is $O(1)$. In the next section, we derive a logarithmic bound for the amortized cost of *cp*. The potential function obtained in this analysis yields a logarithmic bound for deckey.

11.3.2 Analysis of pairing

We use $\sharp t$ to denote $\#\langle a, t\rangle$, hence $\sharp t$ is positive for all lists t. The goal of the analysis of *cp* is to maximize α such that $\alpha > 1$ and

(11) $\mathcal{A}[cp](t) \le \log_\alpha \sharp t$,

where $\mathcal{A}[cp](t)$ denotes the amortized cost of pairing, defined by

$$\mathcal{A}[cp](t) = \mathcal{T}[cp](t) + \Phi.(cp.t) - \Phi.t.$$

Note that $cp.t$ is a tree of type $\langle \mathrm{Int}\rangle_9$, whereas t is a list of type $[\langle \mathrm{Int}\rangle_9]$. Potential Φ is therefore defined both on $\langle \mathrm{Int}\rangle_9$ and $[\langle \mathrm{Int}\rangle_9]$:

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle a, t\rangle & = & \varphi.t + \Phi^*.t \\ \Phi.t & = & \varphi.t + \Phi^*.t, \ \text{ with } \ \Phi^*.t = (\Sigma\, x : x \in t : \Phi.x). \end{array}$$

Hence, $\Phi.\langle a, t\rangle = \Phi.t$. To achieve that Φ is nonnegative, we require φ to be nonnegative. In terms of the amortized costs of linking, given by

$$\mathcal{A}[\bowtie](x, y) = \mathcal{T}[\bowtie](x, y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y,$$

the following recurrence relation can now be derived from the program for *cp*:

$$\begin{array}{lcl} \mathcal{A}[cp]([\,]) & = & -\varphi.[\,] \\ \mathcal{A}[cp]([x]) & = & -\varphi.[x] \\ \mathcal{A}[cp](x \vdash y \vdash t) & = & \mathcal{A}[\bowtie](x, y) + \mathcal{A}[\bowtie](x \bowtie y, cp.t) - \varphi.(x \vdash y \vdash t) + \varphi.t \\ & & +\mathcal{A}[cp](t), \end{array}$$

using that $\Phi.(x \vdash y \vdash t) = \varphi.(x \vdash y \vdash t) + \Phi.x + \Phi.y + \Phi.t - \varphi.t$.

Since φ will be nonnegative, (11) evidently holds if $t = [\,]$ or $t = [x]$. It follows by induction in the remaining case, provided

(12) $\mathcal{A}[\bowtie](x, y) + \mathcal{A}[\bowtie](x \bowtie y, cp.t) - \varphi.(x \vdash y \vdash t) + \varphi.t \le \log_\alpha \dfrac{\sharp(x \vdash y \vdash t)}{\sharp t}$.

To simplify this requirement on φ, we investigate $\mathcal{A}[\bowtie]$. For $x = \langle a, t\rangle$ and $y = \langle b, u\rangle$ we observe for the case $a < b$:

$$
\begin{array}{ll}
 & \mathcal{A}[\bowtie](x,y) \\
= & \{ \text{ definition of } \mathcal{A}[\bowtie] \ \} \\
 & \mathcal{T}[\bowtie](x,y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y \\
= & \{ \text{ definitions of } \bowtie \text{ and } \mathcal{T} \ \} \\
 & 1 + \Phi.\langle a, y \vdash t \rangle - \Phi.\langle a, t \rangle - \Phi.y \\
= & \{ \text{ definition of } \Phi \text{ and } \Phi^* \} \\
 & 1 + \varphi.(y \vdash t) + \Phi.y + \Phi^*.t - \varphi.t - \Phi^*.t - \Phi.y \\
= & \{ \ \} \\
 & 1 + \varphi.(y \vdash t) - \varphi.t.
\end{array}
$$

Introducing

(13) $\delta.(y \vdash t) = \varphi.(y \vdash t) - \varphi.t,$

we then have on account of symmetry

$$\mathcal{A}[\bowtie](x,y) = 1 + \begin{cases} \delta.(y \vdash t) & , a < b \\ \delta.(x \vdash u) & , b < a. \end{cases}$$

To avoid the case analysis in this result we let $\delta.t$ depend on $\sharp t$ only. Defining $\delta.\langle a, t \rangle = \delta.t$, we then have

(14) $\mathcal{A}[\bowtie](x,y) = 1 + \delta.(x \bowtie y),$

since both $\sharp(y \vdash t)$ and $\sharp(x \vdash u)$ are equal to $\#(x \bowtie y)$.

In terms of δ, requirement (12) becomes:

(15) $2 + \delta.(x \bowtie y) + \delta.((x \bowtie y) \bowtie cp.t) - \delta.(x \vdash y \vdash t) - \delta.(y \vdash t) \leq \log_\alpha \frac{\#x + \#y + \sharp t}{\sharp t}.$

As in the previous analyses in this chapter we take

$$\delta.t = \beta \log_\alpha \sharp t,$$

and to achieve that φ is nonnegative we also take $\varphi.[\,] = 0$, which defines φ on account of (13).

Using that $\#(cp.t) = \sharp t - 1$, requirement (15) then reduces to:

$$\alpha^2 \left(\frac{\#x + \#y + \sharp t - 1}{\#x + \#y + \sharp t} \ \frac{\#x + \#y}{\#y + \sharp t} \right)^\beta \leq 1 + \frac{\#x + \#y}{\sharp t}.$$

In summary, we thus have that (11) holds and that Φ is nonnegative provided α and β satisfy: $\alpha > 1$, $\beta > 0$, and

(16) $(\forall\, k, m, n :: \alpha^2 \left(\frac{m + n + k - 1}{m + n + k} \frac{m + n}{n + k} \right)^\beta \leq 1 + \frac{m + n}{k}).$

Since $\frac{m+n+k-1}{m+n+k}$ is about 1, this set of requirements is very similar to the one in the analysis of path reversal. To maximize α, we distinguish two cases.

Case $\beta < 1$. Since constraint (16) is about equal to constraint (4) in the analysis of path reversal, we omit the calculations that lead to the following constraints on α and β:

$$\begin{aligned}&1 < \alpha^2 \leq \frac{1}{\beta^\beta(1-\beta)^{1-\beta}}\\&0 < \beta < 1.\end{aligned}$$

The optimal values thus are $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$.

Case $\beta \geq 1$. Instantiating (16) with "$k, m, n := 1, \infty, 1$" yields that $\alpha^2 \leq 2$, hence $\alpha \leq \sqrt{2}$.

We conclude that the optimal values are $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$. As result for $\mathcal{A}[cp]$ we thus have (on account of (11)):

$$\mathcal{A}[cp](t) \leq 2\log_2 \sharp t,$$

for potential Φ defined by

$$\begin{array}{lcl}\Phi.\langle\,\rangle & = & 0\\ \Phi.\langle a,t\rangle & = & \Phi.t\\ \Phi.[\,] & = & 0\\ \Phi.(x \vdash t) & = & \Phi.x + \log_2 \sharp(x \vdash t) + \Phi.t.\end{array}$$

11.3.3 Bounds for pairing heaps

To analyze the respective operations we recall the following result of the previous section (cf. (14)):

(17) $\Phi.(x \bowtie y) - \Phi.x - \Phi.y = \log_2 \#(x \bowtie y)$.

This yields for the first six operations:

$$\begin{array}{lll}\mathcal{A}[\mathsf{empty}] & \text{is} & O(1)\\ \mathcal{A}[\mathsf{isempty}](x) & \text{is} & O(1)\\ \mathcal{A}[\mathsf{single}](a) & \text{is} & O(1)\\ \mathcal{A}[\mathsf{union}](x,y) & \leq & \log_2 \#[\![\mathsf{union}.x.y]\!] \text{ comparisons}\\ \mathcal{A}[\mathsf{min}](x) & \text{is} & O(1)\\ \mathcal{A}[\mathsf{delmin}](x) & \leq & 2\log_2 \#[\![x]\!] \text{ comparisons.}\end{array}$$

These bounds imply that sorting N numbers by means of pairing heaps requires asymptotically at most $2N\log_2 N$ comparisons, instead of $2.88N\log_2 N$ for Fibonacci heaps.

To analyze deckey we must be a little careful. We use the property that

$$\Phi.(z, \langle a, t\rangle) \geq \Phi.z + \Phi.\langle a, t\rangle.$$

Then we observe, writing $x = (z, \langle a, t\rangle)$:

$$\begin{array}{cl} & \mathcal{A}[\mathsf{deckey}](a, k, x) \\ = & \{ \text{ definition of } \mathcal{A}[\mathsf{deckey}] \ \} \\ & \mathcal{T}[\mathsf{deckey}](a, k, x) + \Phi.(\mathsf{deckey}.a.k.x) - \Phi.x \\ = & \{ \text{ definitions of } \mathcal{T} \text{ and } \mathsf{deckey} \ \} \\ & 1 + \Phi.(z \bowtie \langle a{-}k, t\rangle) - \Phi.(z, \langle a, t\rangle) \\ \leq & \{ \text{ above property } \} \\ & 1 + \Phi.(z \bowtie \langle a{-}k, t\rangle) - \Phi.z - \Phi.\langle a, t\rangle \\ = & \{ \ (17), \text{ using } \Phi.\langle a, t\rangle = \Phi.\langle a{-}k, t\rangle \ \} \\ & 1 + \log_2 \#x. \end{array}$$

Let us briefly compare pairing heaps with Fibonacci heaps. Between data type P and data type F defined in Section 10.3, we observe three major differences: (a) F consists of trees over Int $\times$ $[0..K]$ whereas trees in P are over Int, (b) trees in P can have any structure whereas trees in F are pseudo-binomial, and (c) the order of the subtrees is taken into account for trees in P, while it is irrelevant for trees in F (compare tree types $\langle T\rangle_4$ and $\langle T\rangle_6$).

Because of these differences, Fibonacci heaps are not as "self-adjusting" [7] as pairing heaps are. Combined with the fact that the implementation at pointer level is simpler and more space efficient, this makes pairing heaps interesting from a practical point of view. From a theoretical point of view, however, Fibonacci heaps are to be preferred because the amortized costs of deckey have been shown to be $O(1)$.

11.4 Why these "sum of logs" potentials?

For path reversal, splaying, and pairing, we have derived the following potential functions:

$$\Phi.\langle a, t\rangle = \tfrac{1}{2}\log_2 \#\langle a, t\rangle + (\Sigma\, x : x \in t : \Phi.x),$$

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle t, a, u\rangle & = & \Phi.t + \log_2 \sharp\langle t, a, u\rangle + \Phi.u, \end{array}$$

and

$$\begin{array}{lcl} \Phi.\langle\,\rangle & = & 0 \\ \Phi.\langle a, [\,]\rangle & = & 0 \\ \Phi.\langle a, x \vdash t\rangle & = & \Phi.x + \log_2 \#\langle a, x \vdash t\rangle + \Phi.\langle a, t\rangle. \end{array}$$

For these potential functions, the contribution of each node to the potential of the tree is proportional to the logarithm of the size of its subtrees. (In case of pairing, $\langle a, x \vdash t\rangle$ can be viewed as a node with left subtree x and right subtree $\langle a, t\rangle$.) The fact that these "sum of logarithms" potentials serve to analyze both splaying and pairing is easy to explain, since there is a clear connection between these two operations (see [7, p.121]). But the authors of [12] are at a loss to explain that such a potential can also be used to amortize the cost of path reversal, because they cannot discover a connection between splaying (and pairing) on the one hand, and path reversal on the other hand. To us, however, this fact is not a surprise anymore because we are able to analyze these three operations in the same systematic way.

In connection with the amortized complexity of deckey for pairing heaps, an important question is whether a smaller potential than "sum of logs" is possible. Since $\mathcal{A}[\mathsf{deckey}](a,k,x) \leq 1+\delta.x$, we would obtain a better bound if we were able to show that, for instance, $\delta.x = \log\log \sharp x$ is possible. Unfortunately, however, a smaller potential cannot be obtained in our analyses of path reversal, splaying, and pairing, as we will now show for path reversal.

In the bottom-up analysis of path reversal, we defined $\psi.t = \sharp t$, and this choice for ψ led to requirement (4) on α and β. To investigate whether a smaller choice for $\psi.t$ is possible, e.g. $\psi.t = \log \sharp t$, we introduce a function f on the positive integers, and take $\psi.t = \sqrt[\beta]{f.\sharp t}$. As requirement on f we then obtain formula (18) in the lemma below. In our analysis we have chosen $f.n = \sqrt{n}$ (for $\alpha = 2$ and $\beta = \frac{1}{2}$). Lemma 11.6 shows that f cannot be substantially smaller.

Lemma 11.6
Let f be a function defined on the positive integers such that $f.n \geq 1$. Suppose f satisfies

$$(18) \quad (\forall\, m,n :: \frac{f.m}{f.n} \leq \frac{m+n}{\alpha n}),$$

for some $\alpha > 1$. Then $f.n$ is $\Omega(n^{\epsilon})$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.
Proof For a sufficiently large integer c, we instantiate (18) with $m,n := cn, n$ and $m,n := n, cn$, respectively. This yields that, for all positive n:

$$\frac{f.(cn)}{f.n} \leq \frac{c+1}{\alpha} \quad \wedge \quad \frac{f.n}{f.(cn)} \leq \frac{1+c}{\alpha c},$$

or, equivalently,

$$\frac{\alpha c}{c+1} \leq \frac{f.(cn)}{f.n} \leq \frac{c+1}{\alpha}.$$

Since $\alpha \leq 2$ (instantiate (18) with $m = n$), the lower bound does not exceed the upper bound (using $4c \leq (c{+}1)^2$). We take $c > \frac{1}{\alpha-1}$ (hence $c > 1$), and define

$$\epsilon = \log_c \tfrac{\alpha c}{c+1}.$$

Then $0 < \epsilon < \frac{1}{2}$, and for all positive n:

$$c^\epsilon \leq \frac{f.(cn)}{f.n} \leq c^{1-\epsilon}.$$

From these inequalities we deduce, by induction on k, that

$$c^{k\epsilon} \leq \frac{f.(c^k n)}{f.n} \leq c^{k(1-\epsilon)},$$

for all $k \geq 0$, from which we conclude that $f.n$ is $\Omega(n^\epsilon)$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.
□

Along the same lines it can be shown that a potential which is substantially smaller than a "sum of logs" potential cannot be used in our analyses of splaying and pairing. For pairing the argument is as follows. The counterpart of (18), corresponding to requirement (16) on φ, is

$$(\forall\, k, m, n :: \frac{f.(m+n+k-1)}{f.(m+n+k)}\ \frac{f.(m+n)}{f.(n+k)} \leq \frac{m+n+k}{\alpha^2 k}).$$

Taking $n = 1$, this implies

$$(\forall\, k, m :: \frac{f.(m+k)}{f.(m+k+1)}\ \frac{f.(m+1)}{f.(k+1)} \leq \frac{m+k+1}{\alpha^2 k}).$$

Since $m + k \approx m + k + 1$, this formula is similar to (18); therefore, $f.n$ is again $\Omega(n^\epsilon)$ and $O(n^{1-\epsilon})$, with $0 < \epsilon < \frac{1}{2}$.

The moral thus is that a better bound for $\mathcal{A}[\mathsf{deckey}]$ can only be obtained by using more complex potential functions—if at all possible. It is, for instance, conceivable that a potential takes the depth of the nodes into account; for example, a suitable potential for a tree x might be $\Phi.0.x$, with

$$\begin{array}{lcl} \Phi.k.\langle\,\rangle & = & 0 \\ \Phi.k.\langle t, a, u\rangle & = & \Phi.(k{+}1).t + \varphi.k.t.u + \Phi.(k{+}1).u \end{array}$$

Whether such potentials exist is an open problem.