In: *Information Processing Letters* 37 (1991) 265–271.

The Derivation of a Tighter Bound for Top-Down Skew Heaps

Anne Kaldewaij and Berry Schoenmakers
Dept of Math & CS, TU Eindhoven, The Netherlands

Abstract

In this paper we present and analyze functional programs for a number of priority queue operations. These programs are based upon the top-down skew heaps—a truly elegant data structure—designed by D.D. Sleator and R.E. Tarjan. We show how their potential technique can be used to determine the time complexity of functional programs. This functional approach enables us to *derive* a potential function leading to tighter bounds for the amortized costs of the priority queue operations. From the improved bounds it follows, for instance, that Skewsort, a simple sorting program using these operations, requires only about $1.44N\log_2 N$ comparisons to sort N numbers (in the worst case).

1 Amortized complexity in a functional setting

By means of a simple example we explain how the potential technique of Sleator and Tarjan [7] can be used to determine the time complexity of functional programs. In this example lists of zeros and ones are used as binary representations of natural numbers. We denote the empty list by $[\,]$, and the list with head b and tail s is denoted by $[b] +\!\!+ s$. We abbreviate $[b] +\!\!+ [\,]$ to $[b]$. Binary list s represents natural number $[\![s]\!]$, where $[\![\cdot]\!]$ is the abstraction function defined by $[\![[\,]]\!] = 0$ and $[\![[b] +\!\!+ s]\!] = b + 2[\![s]\!]$. The successor function on the natural numbers is in this representation implemented by program suc:

$$\begin{array}{lcl} \mathsf{suc}.[\,] & = & [1] \\ \mathsf{suc}.([0] +\!\!+ s) & = & [1] +\!\!+ s \\ \mathsf{suc}.([1] +\!\!+ s) & = & [0] +\!\!+ \mathsf{suc}.s, \end{array}$$

where the dot "." denotes function application. Note that $[\![\mathsf{suc}.s]\!] = [\![s]\!] + 1$. As a suitable cost measure for the evaluation of expressions involving function suc, we define $\mathcal{T}(E)$ as the number of unfoldings of the definition of suc needed to evaluate expression E. For example, $\mathcal{T}(\mathsf{suc}.([0] +\!\!+ s))$ equals $1 + \mathcal{T}([1] +\!\!+ s)$, which in turn equals $1 + \mathcal{T}(s)$.

Now, suppose that we want to determine a tight bound for $\mathcal{T}(\mathsf{suc}^n.[\,])$ as function of n. To be able to decompose the cost for n successive applications of suc, we introduce the following "derivative" of $\mathcal{T}$:

$$\mathcal{T}[\mathsf{suc}](E) = \mathcal{T}(\mathsf{suc}.E) - \mathcal{T}(E).$$

Note that $\mathcal{T}[\mathsf{suc}](E)$ depends on the *value* of E only; for example, $\mathcal{T}[\mathsf{suc}](\mathsf{suc}.[\,]) = \mathcal{T}[\mathsf{suc}]([1])$. Since $\mathcal{T}([\,]) = 0$, we then have

$$\mathcal{T}(\mathsf{suc}^n.[\,]) = (\Sigma\, i : 0 \le i < n : \mathcal{T}[\mathsf{suc}](\mathsf{suc}^i.[\,])).$$

We use this equation to estimate $\mathcal{T}(\mathsf{suc}^n.[\,])$ as follows. For $\mathcal{T}[\mathsf{suc}]$ we have the following recurrence relation:

$$\begin{array}{lcl} \mathcal{T}[\mathsf{suc}]([\,]) & = & 1 \\ \mathcal{T}[\mathsf{suc}]([0] +\!\!+ s) & = & 1 \\ \mathcal{T}[\mathsf{suc}]([1] +\!\!+ s) & = & 1 + \mathcal{T}[\mathsf{suc}](s). \end{array}$$

Hence $\mathcal{T}[\mathsf{suc}](s)$ is at most $1 + \#s$ where $\#s$ denotes the length of list s, and, since $\mathsf{suc}^i.[\,]$ has length at most $\log_2(i{+}1)$, the above sum is bounded by $(\Sigma\, i : 0 \leq i < n : 1 + \log_2(i{+}1))$. This yields an $O(n \log n)$ bound for $\mathcal{T}(\mathsf{suc}^n.[\,])$.

However, $\mathcal{T}(\mathsf{suc}^n.[\,])$ is $O(n)$, which is shown in the following amortized analysis. The amortized costs of function suc are defined as

$$\mathcal{A}[\mathsf{suc}](E) = \mathcal{T}[\mathsf{suc}](E) + \Phi.(\mathsf{suc}.E) - \Phi.E,$$

where Φ, the *potential function*, is a real-valued function defined on binary lists. Then the following important equality holds:

$$\mathcal{T}(\mathsf{suc}^n.[\,]) = (\Sigma\, i : 0 \leq i < n : \mathcal{A}[\mathsf{suc}](\mathsf{suc}^i.[\,])) + \Phi.[\,] - \Phi.(\mathsf{suc}^n.[\,]). \tag{1}$$

From this equality we obtain the desired bound for $\mathcal{T}(\mathsf{suc}^n.[\,])$ by choosing a suitable definition for Φ. With $\Phi.[\,] = 0$ and $\Phi.([b] +\!\!+ s) = b + \Phi.s$, we obtain as recurrence relation for $\mathcal{A}[\mathsf{suc}]$:

$$\begin{array}{lcll}
\mathcal{A}[\mathsf{suc}]([\,]) & = & 1 + 1 - 0 & = \quad 2 \\
\mathcal{A}[\mathsf{suc}]([0] +\!\!+ s) & = & 1 + 1 + \Phi.s - 0 - \Phi.s & = \quad 2 \\
\mathcal{A}[\mathsf{suc}]([1] +\!\!+ s) & = & 1 + \mathcal{T}[\mathsf{suc}](s) + 0 + \Phi.(\mathsf{suc}.s) - 1 - \Phi.s & = \quad \mathcal{A}[\mathsf{suc}](s),
\end{array}$$

from which we deduce, by induction, that $\mathcal{A}[\mathsf{suc}](s) = 2$, for all s. This yields $2n + \Phi.[\,] - \Phi.(\mathsf{suc}^n.[\,])$ as expression for $\mathcal{T}(\mathsf{suc}^n.[\,])$, which is at most $2n$, since $\Phi.[\,] = 0$ and Φ is nonnegative.

Summarizing the above amortized analysis, we see that we first choose a (realistic) cost measure $\mathcal{T}$. In terms of $\mathcal{T}$ we then define the cost measure for a single application of suc. Next, we introduce a potential function Φ and we define the amortized costs for suc in terms of its cost measure $\mathcal{T}[\mathsf{suc}]$ and Φ. The key step is, finally, to define Φ such that $(\max s : \#s = n : \mathcal{A}[\mathsf{suc}](s))$ is considerably smaller than $(\max s : \#s = n : \mathcal{T}[\mathsf{suc}](s))$, for large n, and, moreover, to define Φ such that it is nonnegative and small, so that relation (1) gives us a good estimate of $\mathcal{T}(\mathsf{suc}^n.[\,])$.

The remainder of this paper is organised as follows. In the next section we specify six priority queue operations and implement them as functional programs. We introduce a realistic cost measure for these programs and we define amortized costs for each of the six operations. Subsequently, in Section 3, we *derive* a suitable potential function that gives small amortized costs. In Section 4 the priority queue operations are applied in a simple sorting program. We conclude the paper with some remarks on related results.

The approach sketched in this paper is more thoroughly treated in [4].

2 Priority queues

We consider the set $(\!|\mathrm{Int}|\!)$ of finite bags (multisets) of integers for which we use the following notations: $(\!|\,|\!)$ denotes the empty bag and $(\!|a|\!)$ denotes the singleton bag with element a. Bag addition and bag subtraction are denoted by $\oplus$ and $\ominus$ respectively. For a nonempty bag B, $\downarrow\!B$ stands for the minimum of B.

A (mergeable) priority queue is a type P with the following operations:

$$\begin{array}{lll} \mathsf{empty} & : P & [\![\mathsf{empty}]\!] = (\!|\,|\!) \\ \mathsf{single} & : \mathrm{Int} \to P & [\![\mathsf{single}.a]\!] = (\!|a|\!) \\ \mathsf{union} & : P \times P \to P & [\![\mathsf{union}.p.q]\!] = [\![p]\!] \oplus [\![q]\!] \\ \mathsf{delmin} & : P \to P & [\![\mathsf{delmin}.p]\!] = [\![p]\!] \ominus (\!|{\downarrow}[\![p]\!]|\!) \quad , \text{for } [\![p]\!] \neq (\!|\,|\!) \\ \mathsf{isempty} & : P \to \mathrm{Bool} & \mathsf{isempty}.p \equiv [\![p]\!] = (\!|\,|\!) \\ \mathsf{min} & : P \to \mathrm{Int} & \mathsf{min}.p = {\downarrow}[\![p]\!] \quad , \text{for } [\![p]\!] \neq (\!|\,|\!), \end{array}$$

in which $[\![\cdot]\!] : P \to (\!|\mathrm{Int}|\!)$ denotes the abstraction function. In this paper we choose for P a subset of the set $\langle \mathrm{Int} \rangle$ of binary trees of integers which is the smallest set satisfying

$$\begin{array}{l} \langle\,\rangle \in \langle \mathrm{Int} \rangle \\ t \in \langle \mathrm{Int} \rangle \ \wedge\ a \in \mathrm{Int} \ \wedge\ u \in \langle \mathrm{Int} \rangle \ \Rightarrow\ \langle t, a, u \rangle \in \langle \mathrm{Int} \rangle. \end{array}$$

In words, $\langle\,\rangle$ is the empty tree, and $\langle t, a, u \rangle$ is a nonempty tree with left subtree t, root a, and right subtree u. Abstraction function $[\![\cdot]\!]$ is defined in the obvious way as

$$\begin{array}{rcl} [\![\langle\,\rangle]\!] & = & (\!|\,|\!) \\ [\![\langle t, a, u \rangle]\!] & = & [\![t]\!] \oplus (\!|a|\!) \oplus [\![u]\!] \quad . \end{array}$$

For an efficient implementation of min and delmin we restrict P to the binary trees that satisfy heap condition H:

$$\begin{array}{rcl} H.\langle\,\rangle & \equiv & true \\ H.\langle t, a, u \rangle & \equiv & H.t \ \wedge\ (a = {\downarrow}[\![\langle t, a, u \rangle]\!]) \ \wedge\ H.u. \end{array}$$

The implementations of empty, $\mathsf{isempty}$, single, and min are now straigthforward:

$$\begin{array}{lcl} \mathsf{empty} & = & \langle\,\rangle \\ \mathsf{isempty}.z & = & z = \langle\,\rangle \\ \mathsf{single}.a & = & \langle \langle\,\rangle, a, \langle\,\rangle \rangle \\ \mathsf{min}.\langle t, a, u \rangle & = & a. \end{array}$$

Since $[\![\mathsf{delmin}.\langle t, a, u \rangle]\!] = [\![t]\!] \oplus [\![u]\!]$ and $[\![\mathsf{union}.t.u]\!] = [\![t]\!] \oplus [\![u]\!]$ as well, we can deal with delmin and union in a uniform way by introducing operator $\bowtie$ ("meld") on trees satisfying

$$\begin{array}{l} H.x \wedge H.y \ \Rightarrow\ H.(x \bowtie y) \\ [\![x \bowtie y]\!] = [\![x]\!] \oplus [\![y]\!]. \end{array}$$

The programs for union and delmin are then given by

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

Apart from the strange exception for "the lowest node on the merge path" (see p.54 in [6]), the following program for $\bowtie$ is in essence the same as the one that is operationally described by Sleator and Tarjan:

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

where $m.\langle\,\rangle = \infty$ and $m.\langle t, a, u\rangle = a$. A systematic derivation of this program can be found in [4]; the fact that t and u are "swapped" in the last two alternatives is crucial for its efficiency.

As cost measure for the above implementation of priority queues we define $\mathcal{T}(E)$ as the number of unfoldings of the definitions of the six priority queue operations plus the number of unfoldings of the definition of $\bowtie$ needed to evaluate E. The amortized costs are defined as

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

in which Φ is the potential function. It is not difficult to give a definition for Φ such that the amortized costs of empty, isempty, single, and min are $O(1)$. In the next section we derive a potential function which in addition yields logarithmic bounds for $\mathcal{A}[\mathsf{union}]$ and $\mathcal{A}[\mathsf{delmin}]$.

3 Derivation of Φ

For $\mathcal{T}[\bowtie]$ we have the following recurrence relation:

$$\begin{array}{lcll}
\mathcal{T}[\bowtie](\langle\,\rangle, \langle\,\rangle) & = & 1 & \\
\mathcal{T}[\bowtie](\langle t, a, u\rangle, y) & = & 1 + \mathcal{T}[\bowtie](u, y) & , a \leq m.y \\
\mathcal{T}[\bowtie](x, \langle t, a, u\rangle) & = & 1 + \mathcal{T}[\bowtie](u, x) & , a < m.x,
\end{array}$$

from which we conlude that $\mathcal{T}[\bowtie](x, y)$ may be linear in the size of x and y (e.g., when all left subtrees of x and y are empty). As in our simple example, we therefore perform an amortized analysis. As amortized costs for $\bowtie$ we take

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

Our main goal is to define Φ such that $\mathcal{A}[\bowtie](x, y)$ can be shown to be logarithmic in the size of its arguments, i.e., $O(\log(\#x + \#y))$, where $\#z$ equals the number of nodes in z plus one, and is defined recursively by $\#\langle\,\rangle = 1$ and $\#\langle t, a, u\rangle = \#t + \#u$. Furthermore, we want Φ to be nonnegative and small.

As in the example in Section 1 we first derive a recurrence relation for $\mathcal{A}[\bowtie]$. To facilitate the computations, we introduce function φ and choose—in accordance with the recursive structure of the program for $\bowtie$—the following definition for Φ:

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

Note that a does not occur in the right-hand side of the definition of $\Phi.\langle t, a, u\rangle$; this reflects our decision to let $\Phi.z$ depend on the structure of z only.

It is immediate that $\mathcal{A}[\bowtie](\langle\,\rangle, \langle\,\rangle) = 1$, and for case $x = \langle t, a, u\rangle \;\wedge\; a \leq m.y$ we derive, using $x \bowtie y = \langle u \bowtie y, a, t\rangle$:

$$
\begin{array}{ll}
& \mathcal{A}[\bowtie](x,y) \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
& \mathcal{T}[\bowtie](x,y) + \Phi.(x \bowtie y) - \Phi.x - \Phi.y \\
= & \quad \{ \text{ definitions of } \mathcal{T} \text{ and } \Phi \ \} \\
& 1 + \mathcal{T}[\bowtie](u,y) + \Phi.(u \bowtie y) + \varphi.(u \bowtie y).t + \Phi.t - \Phi.t - \varphi.t.u - \Phi.u - \Phi.y \\
= & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
& \mathcal{A}[\bowtie](u,y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u.
\end{array}
$$

So we have the following recurrence relation:

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

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

$$
\mathcal{A}[\bowtie](x,y) = 1 + \Gamma.(x \bowtie y) + \Delta.x + \Delta.y. \tag{2}
$$

Functions Γ and Δ are determined inductively as follows. Evidently, we take $\Gamma.\langle\,\rangle = 0$ and $\Delta.\langle\,\rangle = 0$. Using (2) as induction hypothesis, we obtain the definitions for Γ and Δ for nonempty trees by examining case $x = \langle t,a,u\rangle \ \wedge \ a \leq m.y$:

$$
\begin{array}{ll}
& \mathcal{A}[\bowtie](x,y) \\
= & \quad \{ \text{ above recurrence relation for } \mathcal{A} \ \} \\
& \mathcal{A}[\bowtie](u,y) + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \quad \{ \text{ induction hypothesis (2) } \} \\
& \Gamma.(u \bowtie y) + \Delta.u + \Delta.y + \varphi.(u \bowtie y).t + 1 - \varphi.t.u \\
= & \quad \{ \text{ definitions of } \Gamma \text{ and } \Delta \text{ (see below) } \} \\
& \Gamma.\langle u \bowtie y, a, t\rangle + \Delta.\langle t,a,u\rangle + \Delta.y \\
= & \quad \{ \text{ definition of } \bowtie \ \} \\
& \Gamma.(x \bowtie y) + \Delta.x + \Delta.y.
\end{array}
$$

The remaining case follows by symmetry, hence (2) holds, if we define

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

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

From (2) we can now obtain a logarithmic bound for $\mathcal{A}[\bowtie](x,y)$ by deriving a suitable definition for φ. To this end we introduce two constants α and β, say, and try to define φ such that $\Gamma.z \leq \log_\alpha \#z$ and $\Delta.z \leq \log_\beta \#z$, with $\alpha > 1$ and $\beta > 1$. Then it follows from (2) that

$$
\mathcal{A}[\bowtie](x,y) \leq 1 + (\tfrac{1}{\log_2 \alpha} + \tfrac{2}{\log_2 \beta})\log_2(\#x + \#y). \tag{3}
$$

We have $\Gamma.\langle\,\rangle \leq \log_\alpha \#\langle\,\rangle$ and $\Delta.\langle\,\rangle \leq \log_\alpha \#\langle\,\rangle$ and we derive inductively the requirements for φ:

$$\begin{array}{ll}
& \Gamma.\langle t,a,u\rangle \\
= & \quad \{ \text{ definition of } \Gamma \ \} \\
& \Gamma.t + \varphi.t.u + \gamma \\
\leq & \quad \{ \text{ induction hypothesis } \} \\
& \log_\alpha \#t + \varphi.t.u + \gamma \\
\leq & \quad \{ \text{ see upper bound for } \varphi \text{ below } \} \\
& \log_\alpha \#\langle t,a,u\rangle
\end{array}
\qquad
\begin{array}{ll}
& \Delta.\langle t,a,u\rangle \\
= & \quad \{ \text{ definition of } \Delta \ \} \\
& \Delta.u - \varphi.t.u + \delta \\
\leq & \quad \{ \text{ induction hypothesis } \} \\
& \log_\beta \#u - \varphi.t.u + \delta \\
\leq & \quad \{ \text{ see lower bound for } \varphi \text{ below } \} \\
& \log_\beta \#\langle t,a,u\rangle.
\end{array}$$

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

$$\log_\beta \frac{\beta^\delta \#u}{\#t + \#u} \leq \varphi.t.u \leq \log_\alpha \frac{\#t + \#u}{\alpha^\gamma \#t}.$$

The existence of such a φ is guaranteed if the lower bound never exceeds the upper bound. For any positive m and n, we have with $\varepsilon = \log_\beta \alpha$:

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

and therefore it suffices to choose α and β such that

$$\alpha \leq \frac{(\varepsilon+1)^{\varepsilon+1}}{\varepsilon^\varepsilon}, \tag{4}$$

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

It is easily seen that such α and β exist, but we want to choose them such that the constant in (3) is small. In terms of α and ε, (3) reads:

$$\mathcal{A}[\bowtie](x,y) \leq 1 + (\tfrac{2\varepsilon+1}{\log_2 \alpha})\log_2(\#x + \#y),$$

and therefore we want to minimize $\frac{2\varepsilon+1}{\log_2 \alpha}$ under condition (4) over all $\alpha > 1$ and $\varepsilon > 0$. For this purpose we may take equality in (4), since, for fixed ε, $\frac{2\varepsilon+1}{\log_2 \alpha}$ decreases for increasing α. Using this equality then gives

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

as quantity to minimize for positive ε. This gives 1.4404 as minimal value at $\varepsilon = 0.61803$. From the equality in (4) we then get $\alpha = 2.9330$, and the definition of ε gives $\beta = 5.7033$.

So much for the amortized costs of $\bowtie$. We now try to define φ such that it is nonnegative and small. For this purpose we take $\gamma = 0$ and $\delta = 1$, which yields as bounds for φ:

$$1 + \log_\beta \frac{\#u}{\#t + \#u} \leq \varphi.t.u \leq \log_\alpha \frac{\#t + \#u}{\#t}.$$

For this choice for γ and δ, we have that the upper bound is at least 0 and the lower bound is at most 1. This enables us to define φ such that $0 \leq \varphi.t.u \leq 1$ as follows. In case the lower bound is at most 0, we just take $\varphi.t.u = 0$. Similarly, in case the upper bound is at least 1, we take $\varphi.t.u = 1$. For the remaining case any function between the bounds will do. Arbitrarily choosing the upper bound for φ in the middle case, we thus define:

$$\varphi.t.u = \begin{cases} 0 & ,(\beta{-}1)\#u \leq \#t \\ \log_\alpha(\frac{\#t+\#u}{\#t}) & ,\#t < (\beta{-}1)\#u \ \wedge \ \#u < (\alpha{-}1)\#t \\ 1 & ,(\alpha{-}1)\#t \leq \#u, \end{cases}$$

with $\alpha = 2.93$ and $\beta = 5.70$. As a result, potential Φ satisfies $0 \leq \Phi.z \leq \#z$. This completes the analysis of $\bowtie$.

By taking $\alpha = \beta = 2$ in the above analysis, we obtain for φ:

$$\varphi.t.u = \begin{cases} 0 & ,\#u \leq \#t \\ 1 & ,\#t < \#u. \end{cases}$$

The corresponding Φ is then roughly the potential function of Sleator and Tarjan for top-down melding, and the constant in (3) becomes 3, so we have reduced the constant from 3 to 1.44.

We leave it to the reader to verify that the amortized costs of empty, isempty, single, and min are $O(1)$, and that $\mathcal{A}[\mathsf{union}](x, y)$ is $O(\log(\#x + \#y))$ (approximately $1.44 \log_2(\#x + \#y)$ comparisons in the worst case). For delmin we derive:

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

Hence, $\mathcal{A}[\mathsf{delmin}](z)$ is $O(\log \#z)$ (approximately $1.44 \log_2 \#z$ comparisons in the worst case). Note that this conclusion can only be drawn if φ is bounded from below by a constant (e.g., if φ is nonnegative).

4 Skewsort

As a simple application of priority queues, we present the following sorting program, where $s +\!\!+ [b]$ denotes the list with front s and last element b:

$$
\begin{array}{lll}
\mathit{Skewsort} & = & h \circ g \circ f \\
 & & |[\ f.[\,] \quad = \quad [\,] \\
 & & \quad f.([a] +\!\!+ s) \quad = \quad [\mathsf{single}.a] +\!\!+ f.s \\
 & & \quad g.[\,] \quad = \quad \mathsf{empty} \\
 & & \quad g.[p] \quad = \quad p \\
 & & \quad g.([p] +\!\!+ [q] +\!\!+ ps) \quad = \quad g.(ps +\!\!+ [\mathsf{union}.p.q]) \\
 & & \quad h.p \quad = \quad [\,] \qquad\qquad\qquad\qquad , \quad \mathsf{isempty}.p \\
 & & \qquad\quad [] \quad [\mathsf{min}.p] +\!\!+ h.(\mathsf{delmin}.p) \quad , \neg \mathsf{isempty}.p \\
 & &]|.
\end{array}
$$

In each application of union only priority queues that differ by at most about a factor two in size are united. As shown in [4], this ensures that the amortized cost of $g \circ f$ is linear, assuming that each list operation takes $O(1)$ time. As a consequence, Skewsort takes about $1.44N\log_2 N$ comparisons to sort N numbers, in the worst case. (Of course, Skewsort is not a useful application of mergeable priority queues. If it were only for the above program it is better to take the set of ascending lists of integers as definition for P in Section 2. Taking $\mathsf{union}.p.q$ equal to the merge of lists p and q and implementing the other operations in the obvious way, the above program then reduces to Mergesort, which needs at most $N\log_2 N$ comparisons.)

In the above it is assumed that each list operation takes $O(1)$ time only. The list passed by function f to function g is however modified at both ends, or, more precisely, it is used as a queue. To achieve the $O(1)$ time bounds for these list operations we can, for instance, use the queue implementation described by Gries ([3], pp. 250–251). This is a purely functional solution. In this case, however, it is also possible to use a so-called *destructive* implementation of lists, viz. singly-linked lists with an additional pointer to the last element. So, the assumption that all list operations in Skewsort are $O(1)$ is justified.

5 Concluding remarks

Although we did not say it in so many words in Section 3, we hope that the reader has noticed that the derivation of the potential function depends on only a few choices and that the rest follows by calculation. In [4] this fact is discussed more extensively, and there it is also shown that the programs presented in Section 2 follow from only a few design decisions.

We have shown how a potential function can be derived from efficiency requirements that are imposed on the operations. The functional approach enables us to discuss the program and its efficiency without operational arguments and without using notions such as paths and pointers. In [4] we have investigated the bottom-up skew heaps in the same fashion, and in a systematic way we have improved the bounds for the amortized costs of [6] by a factor 2: we have proved that the amortized number of comparisons for $\mathsf{delmin}.p$ is bounded by $2(1 + \log_2 \#[\![p]\!])$ and that the amortized costs of the remaining operations are all $O(1)$. From the improved bound for delmin it follows, for instance, that the above sorting program with

$g \circ f$ replaced by gf, where

$$\begin{array}{lcl} gf.[\,] & = & \mathsf{empty} \\ gf.([a] +\!\!+ s) & = & \mathsf{union}.(\mathsf{single}.a).(gf.s), \end{array}$$

takes about $2N\log_2 N$ comparisons in the worst case if we use bottom-up skew heaps.

An interesting observation is that by taking for $\varphi.t.u$ its upper bound $\log_4 \frac{\#t+\#u}{\#t}$, a potential $\Phi.z$ is obtained, which is roughly the sum of the logarithms of the sizes of all *right* subtrees of z. This potential may be considered as a variation of the well-known potential for splay trees [5], which is the sum of the logarithms of the sizes of *all* subtrees. So, also for skew heaps there exists a potential that is related to the "sum of logarithms" potential, which also serves to analyze pairing heaps [1] and path reversal [2]. In all these cases we are able to reason rather systematically that the sum of logarithms is an appropriate potential function. Since the sum of logarithms potential may be as large as $n \log n$ for n-node trees, whereas our potential function for skew heaps is at most n, we raise the question whether there exist *linear* potentials for these data structures.

Acknowledgements

D.D. Sleator is gratefully acknowledged for showing us how to reduce the constant from $\frac{3}{2}$ to 1.44. In a previous analysis we identified constants α and β. In that case $\varepsilon = 1$ and condition (4) reduces to $\alpha \leq 4$. Taking $\alpha = 4$ then gives $\frac{2\varepsilon+1}{\log_2 \alpha} = \frac{3}{2}$.

We also thank the members of the Eindhoven Algorithm Club, especially Lex Bijlsma and Wim Nuij, and the referees for their useful suggestions that improved the presentation of this paper.

References

1. Fredman M.L., Sedgewick R., Sleator D.D., Tarjan R.E., *The pairing heap: a new form of self-adjusting heap.* Algorithmica 1 (1986) 111–129.
2. Ginat D., Sleator D.D., Tarjan R.E., *A tight amortized bound for path reversal.* Information Processing Letters 31 (1989) 3–5.
3. Gries D., *The science of computer programming.* Springer-Verlag (1981).
4. Schoenmakers, L.A.M., *Algebra refinement and amortized complexity in a functional setting.* Computing Science Notes, Eindhoven University of Technology (1990).
5. Sleator D.D., Tarjan R.E., *Self-adjusting binary search trees.* Journal of the ACM 32 (1985) 652–686.
6. Sleator D.D., Tarjan R.E., *Self-adjusting heaps.* SIAM Journal on Computing 15 (1986) 52–69.
7. Tarjan R.E., *Amortized computational complexity.* SIAM Journal on Algebraic and Discrete Methods 6 (1985) 306–318.