

Inorder Traversal of a Binary Heap and its Inversion in Optimal Time and Space

Berry Schoenmakers

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
wsinbs@win.tue.nl

Abstract

In this paper we derive a linear-time, constant-space algorithm to construct a binary heap whose inorder traversal equals a given sequence. We do so in two steps. First, we invert a program that computes the inorder traversal of a binary heap, using the proof rules for program inversion by W. Chen and J.T. Udding. This results in a linear-time solution in terms of binary trees. Subsequently, we data-refine this program to a constant-space solution in terms of linked structures.

1 Introduction

In [7] an elegant sorting algorithm is presented which exploits the presortedness of the input sequence. The first step of this variant of Heapsort comprises the conversion of the input sequence in a “mintree,” i.e., a binary heap whose inorder traversal equals the input sequence.¹ For this conversion, the authors of [7] provide a complicated, yet linear algorithm, consisting of no fewer than four repetitions. In this paper we show that the practical significance of the sorting algorithm can be increased considerably by deriving a conversion algorithm that consists of a single repetition only.

The derivation proceeds in two steps. In the first step we derive an algorithm in terms of binary trees. We do so by inverting a program that solves the “inverse problem,” i.e., it computes the inorder traversal of a binary heap in linear time. To guarantee the correctness of this inversion, we apply the proof rules given by W. Chen and J.T. Udding in [3]. These proof rules support stepwise program inversion.

Subsequently, in the second step, we refine this algorithm to a program that operates on linked structures instead of binary trees. Just as in [7], our object is to minimize space utilization. It turns out that we can implement the construction—in a simple way—such that only $O(1)$ additional space is required. This contrasts favourably with the complicated method the authors of [7] seem to have in mind to achieve this for their algorithm—as far as we can conclude from their hint in the footnote, where they remark that the construction can be done “without wasting space.” We also present a refinement using an array representation for binary trees, since this is advantageous when the input sequence is also represented by an array, as is often the case.

¹Actually, they use a “maxtree,” but we have taken the liberty to change this into “mintree” so as to comply with [2, pp. 55-67].

2 Problem Statement

The problem is specified in terms of lists of type $[Int]$ and binary trees of type $\langle Int \rangle$. Later, these types will be refined to pointer and array types.

For lists and trees we use the following notations. $[]$ stands for the empty list, $[a]$ stands for the singleton list with element a , and catenation is denoted by $++$. Furthermore, the head, tail, front, and last element of a nonempty list s are denoted by $hd.s$, $tl.s$, $ft.s$, and $lt.s$, respectively. Hence, s may be written both as $[hd.s] ++ tl.s$ and as $ft.s ++ [lt.s]$. The length of s is denoted by $\#s$. Similarly, $\langle \rangle$ stands for the empty tree, and $\langle t, a, u \rangle$ for a nonempty tree with left subtree t , root a , and right subtree u . The left subtree, root, and right subtree of nonempty tree t are denoted by $l.t$, $m.t$, and $r.t$, respectively.

In terms of these types, the problem is stated as follows. Let \bar{t} denote the *inorder traversal* of tree t . That is,

$$\begin{aligned} \overline{\langle \rangle} &= [] \\ \overline{\langle t, a, u \rangle} &= \bar{t} ++ [a] ++ \bar{u}. \end{aligned}$$

Furthermore, let $H.t$ denote that tree t satisfies the *heap condition*, defined by

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

in which \downarrow denotes the *minimum* of a tree (with $\downarrow \langle \rangle = \infty$). Then, given list s satisfying $s = S$, the problem is to design an $O(\#S)$ program with postcondition

$$H.t \wedge \bar{t} = S.$$

Here, S denotes a specification variable that may not be used in the program.

3 Proof Rules for Program Inversion

Assuming some familiarity with program inversion, we confine ourselves to a brief summary of the results of [3]. There, the inverse of a program is defined as follows.

Program T is said to be an inverse of program S under precondition P when $\{P \wedge Q\} S ; T \{Q\}$ for any predicate Q .

Obviously, **skip** is then its own inverse. For each of the other constructs of Dijkstra's guarded command language, Chen and Udding provide proof rules to support stepwise program inversion. Below, the rules for assignments and sequential compositions are simply copied from [3]. The rules for the other two constructs are instantiations of the more general rules presented in [3].

Proof rule for assignments

$$P \Rightarrow \text{def}(E0) \wedge \text{def}((E1)_{E0}^x) \wedge x = (E1)_{E0}^x$$

$$\{P \wedge Q\} x := E0 ; x := E1 \{Q\} \text{ for any } Q$$

Here, $\text{def}(E)$ means that expression E is well-defined.

Proof rule for sequential compositions

$$\begin{array}{l} \{P\} S0 \{R\} \\ \{P \wedge Q\} S0 ; T0 \{Q\} \text{ for any } Q \\ \{R \wedge Q\} S1 ; T1 \{Q\} \text{ for any } Q \end{array}$$

$$\{P \wedge Q\} S0 ; S1 ; T1 ; T0 \{Q\} \text{ for any } Q$$

Proof rule for selections

$$\begin{array}{l} \{P \wedge B0\} S0 \{C0 \wedge \neg C1\} \\ \{P \wedge B1\} S1 \{C1 \wedge \neg C0\} \\ P \Rightarrow B0 \vee B1 \\ \{P \wedge B0 \wedge Q\} S0 ; T0 \{Q\} \text{ for any } Q \\ \{P \wedge B1 \wedge Q\} S1 ; T1 \{Q\} \text{ for any } Q \end{array}$$

$$\{P \wedge Q\} \text{ if } B0 \rightarrow S0 \ \square \ B1 \rightarrow S1 \ \text{fi} ; \text{ if } C0 \rightarrow T0 \ \square \ C1 \rightarrow T1 \ \text{fi} \{Q\} \text{ for any } Q$$

Proof rule for repetitions

$$\begin{array}{l} \{P \wedge \neg C\} \text{ do } B \rightarrow S \ \text{od} \{\text{true}\} \\ \{P \wedge B\} S \{P \wedge C\} \\ \{P \wedge B \wedge Q\} S ; T \{Q\} \text{ for any } Q \end{array}$$

$$\{P \wedge \neg C \wedge Q\} \text{ do } B \rightarrow S \ \text{od} ; \text{ do } C \rightarrow T \ \text{od} \{Q\} \text{ for any } Q$$

Note that an inverse constructed according to these rules is such that it *exactly* retraces the steps of the program inverted. Also notice that such an inverse is *deterministic* by construction.

4 The Program to Be Inverted

As outlined in Section 1, we will first solve the “inverse problem:” given t , satisfying $H.t \wedge \bar{t} = S$, construct an $O(\#S)$ program with postcondition $s = S$. Since this problem has been solved in a neat way already many times (e.g., in [5, 6, 3]), we merely present a solution without derivation. In each of the programs in [5, 6, 3], a list of trees, which we name q , is used, and the loop invariant is something like

$$s \# \bar{t} \# \bar{q} = S, \tag{P0}$$

where $\bar{\cdot}$ denotes the inorder traversal of a *list of trees*:

$$\begin{array}{l} \overline{[]} = [] \\ \overline{[t] \# q} = \bar{t} \# \bar{q}. \end{array}$$

Starting from this invariant, the following program is easily calculated:

```

{  $\bar{t} = S$  }
   $s, q := [], []$ 
{ invariant:  $P0 \wedge P1$ ; bound:  $2\#S - (2\#s + \#q)$  }
; do  $t \neq \langle \rangle \vee q \neq [] \rightarrow$ 
    if  $t \neq \langle \rangle \rightarrow t, q := l.t, [\langle \rangle, m.t, r.t] \# q$ 
     $t = \langle \rangle \rightarrow s, t, q := s \# [m.(hd.q)], r.(hd.q), tl.q$ 
    fi
od
{  $s = S$  } ,

```

where P1 is required to prove the invariance of P0:

$$(\forall i : 0 \leq i < \#q : q.i \neq \langle \rangle \wedge l.(q.i) = \langle \rangle). \quad (\text{P1})$$

Having convinced ourselves of the correctness of this program, we can now ignore the above invariant and bound function, and annotate the program so as to facilitate its inversion. In finding the appropriate annotations, we are guided by the proof rules for program inversion. Of course, the precondition that t satisfies the heap condition must be exploited:

```

{  $H.t \wedge \bar{t} = S$  }
   $s, q := [], []$ 
{  $P \wedge \neg(s \neq [] \vee q \neq []) \wedge (H.t \wedge \bar{t} = S)$  }
; do  $t \neq \langle \rangle \vee q \neq [] \rightarrow$ 
    {  $P \wedge (t \neq \langle \rangle \vee q \neq [])$  }
    if  $t \neq \langle \rangle \rightarrow t, q := l.t, [\langle \rangle, m.t, r.t] \# q$            {  $A \wedge \neg B$  }
     $t = \langle \rangle \rightarrow s, t, q := s \# [m.(hd.q)], r.(hd.q), tl.q$    {  $B \wedge \neg A$  }
    fi
    {  $P \wedge (s \neq [] \vee q \neq [])$  }
od
{  $s = S \wedge t = \langle \rangle \wedge q = []$  } .

```

In these annotations, P denotes the *strongest* invariant for the above repetition that holds initially—hence, P implies both P0 and P1. Conditions A and B correspond to $C0$ and $C1$ in the proof rule for selections, and will be defined in the next section.

5 The Inversion

First of all, let us explain how the problem stated in Section 2 can be solved using an inverse of the repetition of the above program. To that end, let **DO** denote the repetition of this program, and let **OD** denote an inverse of **DO** under precondition $P \wedge \neg(s \neq [] \vee q \neq [])$. According to the definition of a program's inverse (Section 3), this means that

$$\{ P \wedge \neg(s \neq [] \vee q \neq []) \wedge Q \} \mathbf{DO} ; \mathbf{OD} \{ Q \}$$

for any predicate Q . The annotations in the above program show that $H.t \wedge \bar{t} = S$ is a precondition of **DO**, hence we may take this for Q . Moreover, we see that $s = S \wedge t = \langle \rangle \wedge q = []$ is a postcondition of **DO**. Since this postcondition is a one-point predicate, it is the strongest postcondition of **DO**, and we thus conclude that

$$\begin{aligned} & \{ P \wedge \neg(s \neq [] \vee q \neq []) \wedge (H.t \wedge \bar{t} = S) \} \\ & \quad \mathbf{DO} \\ & \{ s = S \wedge t = \langle \rangle \wedge q = [] \} \\ & \quad ; \mathbf{OD} \\ & \{ H.t \wedge \bar{t} = S \} . \end{aligned}$$

As a consequence, we can use **OD** to solve the problem as follows:

$$\begin{aligned} & \{ s = S \} \\ & \quad t, q := \langle \rangle, [] \\ & \{ s = S \wedge t = \langle \rangle \wedge q = [] \} \\ & \quad ; \mathbf{OD} \\ & \{ H.t \wedge \bar{t} = S \} . \end{aligned}$$

Thus, we have to invert **DO**. Inspection of the proof rule for repetitions reveals that this gives rise to inversion of the body of **DO**. So, let **IF** denote the body of **DO**. Then we have to determine an inverse of **IF** under precondition $P \wedge (t \neq \langle \rangle \vee q \neq [])$. To this end, we have to find conditions A and B such that the above annotation is valid (cf. the proof rule for selections, first two antecedents). Moreover, A and B have to be boolean expressions that may be used as guards in our inverse of **IF**.

Before we derive A and B, however, we first invert the assignments in **IF**, since this is also required by the proof rule for selections (last two antecedents). For the assignment

$$t, q := l.t, [\langle \rangle, m.t, r.t] \# q, \tag{a}$$

the following assignment is an inverse under precondition $t \neq \langle \rangle$:

$$t, q := \langle t, m.(hd.q), r.(hd.q) \rangle, tl.q.$$

This is easily verified by applying “substitution” (a) to $\langle t, m.(hd.q), r.(hd.q) \rangle, tl.q$, as prescribed by the proof rule for assignments.

The other assignment to be inverted is

$$s, t, q := s \# [m.(hd.q)], r.(hd.q), tl.q, \tag{b}$$

for which we find

$$s, t, q := ft.s, \langle \rangle, [[\langle \rangle, lt.s, t]] \# q$$

as an inverse under precondition

$$t = \langle \rangle \wedge q \neq [] \wedge hd.q \neq \langle \rangle \wedge l.(hd.q) = \langle \rangle. \tag{\star}$$

Since we only have to invert assignment (b) under precondition $P \wedge t = \langle \rangle \wedge q \neq []$, this suffices, because this precondition implies (\star) on account of invariant P1. (Recall that, by definition, P implies any invariant of **DO**.)

Our final problem is now to determine A and B. To this end we closely examine the effects of the assignments in **IF**. In order to get useful results, we use that—as may be expected— t is initially a heap, and that, consequently, P implies P2 as well, with

$$H.t \wedge (\forall i : 0 \leq i < \#q : H.(q.i)). \quad (\text{P2})$$

Examination of assignment (a) reveals that it has $m.t \geq m.(hd.q)$ as a postcondition; consequently, this condition is a candidate for A and its negation is a candidate for B. Unfortunately, however, $m.t < m.(hd.q)$ does not hold after assignment (b), since it turns out that P also implies

$$\mathbf{m}.t \geq \mathbf{m}hd.q \wedge (\forall i : 0 \leq i < \#q - 1 : m.(q.i) \geq m.(q.(i+1))), \quad (\text{P3})$$

with

$$\mathbf{m}.t = \begin{cases} m.t & , t \neq \langle \rangle \\ \infty & , t = \langle \rangle, \end{cases}$$

and

$$\mathbf{m}hd.q = \begin{cases} m.(hd.q) & , q \neq [] \\ -\infty & , q = []. \end{cases}$$

Here, functions \mathbf{m} and $\mathbf{m}hd$ are defined such that $\mathbf{m}.\langle \rangle \geq \mathbf{m}hd.q$ for all q and $\mathbf{m}.t \geq \mathbf{m}hd.[]$ for all t . Note that \mathbf{m} is not needed in the universal quantification in P3, since the trees in q are nonempty due to invariant P1.

Since the above examination of the first alternative does not give us a solution for A and B, we now examine the second alternative. We observe that after its assignment $lt.s \leq \mathbf{m}.t$, since $m.(hd.q) \leq \mathbf{m}.(r.(hd.q))$ is a precondition in this case (on account of P2). But again—unfortunately—we have a similar postcondition for the other alternative. More precisely, P implies

$$lt.s \leq \mathbf{m}.t, \quad (\text{P4})$$

with

$$lt.s = \begin{cases} lt.s & , s \neq [] \\ -\infty & , s = []. \end{cases}$$

Thus, also our examination of the second alternative does not give the desired result. However, the above examinations resulted in two additional invariants, P3 and P4, which we can exploit as follows. Firstly, we have $lt.s \geq \mathbf{m}hd.q$ after assignment (b), by the axiom of assignment and the invariance of P3. And, secondly, we have after assignment (a) that $lt.s \leq \mathbf{m}hd.q$, by the axiom of assignment and the invariance of P4. So, we are done if we can conclude that $lt.s \neq \mathbf{m}hd.q$ after one of the two guarded commands. For this purpose we introduce a *skewed* version of H , defined by

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

and we replace H by \widehat{H} in the annotations of the preceding programs. Consequently, we have that P also implies

$$\widehat{H}.t \wedge (\forall i : 0 \leq i < \#q : \widehat{H}.(q.i)), \text{ and} \quad (\text{P2a})$$

$$lt.s < m.t. \quad (\text{P4a})$$

Now it follows that $lt.s < mhd.q$ holds after assignment (a).

As a result, we obtain the following linear-time program for the original problem:

```

{ s = S }
  t, q := ⟨ ⟩, []
; do s ≠ [] ∨ q ≠ [] →
  if lt.s < mhd.q → t, q := ⟨t, m.(hd.q), r.(hd.q)⟩, tl.q
  [] lt.s ≥ mhd.q → s, t, q := ft.s, ⟨ ⟩, [⟨ ⟩, lt.s, t] ++ q
  fi
od
{  $\widehat{H}.t \wedge \bar{t} = S$ , hence  $H.t \wedge \bar{t} = S$  } .

```

Notice that the replacement of $H.t$ by $\widehat{H}.t$ is harmless in the sense that we are still able to solve the problem for any sequence S .

6 A Nondeterministic Solution

In the previous section we have arbitrarily chosen to replace H by \widehat{H} . Of course, we may also decide to replace H by the symmetrical counterpart of \widehat{H} . This leads to a solution with \leq instead of $<$ in the first alternative, and $>$ instead of \geq in the second alternative. Since this implies that P0–P4 is an invariant for both solutions, we infer that P0–P4 is invariant under both assignments in **OD**, if $lt.s = mhd.q$. Therefore we also have the following nondeterministic solution, in which we have made the types explicit to facilitate the data refinements in the next section.

```

proc C (in s:[Int] ; out t:⟨Int⟩)
  { s = S }
  || var q:⟨Int⟩ ;
  t, q := ⟨ ⟩, []
; do s ≠ [] ∨ q ≠ [] →
  if lt.s ≤ mhd.q → t, q := ⟨t, m.(hd.q), r.(hd.q)⟩, tl.q
  [] lt.s ≥ mhd.q → s, t, q := ft.s, ⟨ ⟩, [⟨ ⟩, lt.s, t] ++ q
  fi
od
||
{ H.t ∧  $\bar{t} = S$  }
corp.

```

The correctness of this nondeterministic procedure has to be established in the conventional way, because the proof rules of [3] can only be used to derive deterministic programs.

7 Two Data Refinements

We shall refine procedure C in two steps. In the first step, we do away with type $[\langle \text{Int} \rangle]$. Subsequently, we replace values of type $\langle \text{Int} \rangle$ by linked structures. Since the use of pointers in the resulting refinement is limited, we can also use arrays instead of pointers, as will be shown in Section 7.3.

7.1 Elimination of type $[\langle \text{Int} \rangle]$

The important observation is that the trees in list q all have empty left subtrees on account of invariant P1:

$$(\forall i : 0 \leq i < \#q : q.i \neq \langle \rangle \wedge l.(q.i) = \langle \rangle).$$

Such lists of trees may be represented by binary trees as follows: $[\]$ is represented by $\langle \rangle$, and a nonempty list $[\langle \rangle, a, t] \uparrow q$ is represented by tree $\langle u, a, t \rangle$, where u is the representation of q . Thus, under this representation, we can replace type $[\langle \text{Int} \rangle]$ by $\langle \text{Int} \rangle$, resulting in procedure $C1$:

```

proc C1 (in  $s : [\text{Int}]$  ; out  $t : \langle \text{Int} \rangle$ )
  || var  $q : \langle \text{Int} \rangle$  ;
     $t, q := \langle \rangle, \langle \rangle$ 
  ; do  $s \neq [\ ] \vee q \neq \langle \rangle \rightarrow$ 
    if  $lt.s \leq m.q \rightarrow t, q := \langle t, m.q, r.q \rangle, l.q$ 
    ||  $lt.s \geq m.q \rightarrow s, t, q := ft.s, \langle \rangle, \langle q, lt.s, t \rangle$ 
    fi
  od
  ||
corp,

```

$$\text{with } m.q = \begin{cases} m.q & , q \neq \langle \rangle \\ -\infty & , q = \langle \rangle. \end{cases}$$

Now, only type $\langle \text{Int} \rangle$ remains to be refined.

7.2 Implementation in terms of pointers

We adopt a Pascal-like notation for pointers and tuples (“records”). Binary trees are represented in the common way by values of type B , which is defined as

$$B = \uparrow \langle l : B, m : \text{Int}, r : B \rangle.$$

Hence, **nil** represents $\langle \rangle$ and for the assignment $t, q := \langle t, m.q, r.q \rangle, l.q$ we have as obvious refinement, in which q is now of type B :

$$[[\text{var } b : B; \text{new}(b); b \uparrow := \langle t, q \uparrow . m, q \uparrow . r \rangle; t, q, b := b, q \uparrow . l, q; \text{dispose}(b)]].$$

Assuming that disposed cells are recycled, we see that this block does not change the number of cells in use. However, we can avoid the calls to **new** and **dispose** altogether by recycling the cell, to which q points initially, in-line: the latter cell can be used instead of the cell returned by **new**(b). The required assignment is now a simple *rotation* of three pointers: $t, q, q \uparrow . l := q, q \uparrow . l, t$.

Without further comment, we thus obtain:

```

proc C2 (in s:[Int] ; out t:B)
  |[var q:B ;
    t, q := nil, nil
  ; do s ≠ [] ∨ q ≠ nil →
    if lt.s ≤ q↑.m → t, q, q↑.l := q, q↑.l, t
    [] lt.s ≥ q↑.m → |[var b:B; new(b) ; b↑ := ⟨q, lt.s, t⟩ ; s, t, q := ft.s, nil, b ]|
    fi
  od
  ]|
corp,

```

with $\mathbf{nil}\uparrow.m = -\infty$. Execution of this program for an input sequence of length N gives rise to N calls to **new**, from which we infer that exactly the number of cells required for the representation of the output tree is used. Hence, this program uses only constant extra space.

7.3 Implementation in terms of arrays

In case type [Int] is refined by an array type, it is advantageous to refine $\langle \text{Int} \rangle$ by an array type as well. To that end we introduce a global array a (again, in Pascal-like notation):

$$a : \mathbf{array} \text{ Nat of } \langle l:\text{Nat}, m:\text{Int}, r:\text{Nat} \rangle.$$

In this context, we can associate a binary tree with each natural number as follows: 0 represents $\langle \rangle$, and $n, n > 0$, represents a nonempty tree with root $a[n].m$, and with $a[n].l$ and $a[n].r$ representing its left and right subtree, respectively. (We assume that a is such that this definition defines a finite tree for all naturals—as we did, without mentioning, for values of type B.)

The desired refinement of $C1$ can now be obtained from $C2$ by replacing B by Nat, **nil** by 0, $q\uparrow$ by $a[q]$, and $b\uparrow$ by $a[b]$. Furthermore, we turn b into a global variable (initially $b = 0$), and replace **new**(b) by $b := b + 1$. As a consequence, procedure call $C3(s, t)$ establishes that t and segment $a[0..\#s]$ represent the tree corresponding to s , where

```

proc C3 (in s:[Int] ; out t:Nat)
  |[var q:Nat;
    t, q := 0, 0
  ; do s ≠ [] ∨ q ≠ 0 →
    if lt.s ≤ a[q].m → t, q, a[q].l := q, a[q].l, t
    [] lt.s ≥ a[q].m → b := b + 1 ; a[b] := ⟨q, lt.s, t⟩ ; s, t, q := ft.s, 0, b
    fi
  od
  ]|
corp,

```

with $a[0].m = -\infty$.

Now, note that a call to $C3$ establishes $(\forall i : 0 \leq i < \#s : s.i = a[\#s - i].m)$. Hence, in case type [Int] is refined by an array type, there is the opportunity to omit component m from the elements of a without loss of efficiency.

8 Concluding Remarks

The problem of constructing a heap from its inorder traversal had already been posed and solved by R.S. Bird [2, pp. 55–67]. Not being satisfied with Bird’s derivation, we first solved the problem—from scratch—using elementary techniques from functional programming. Having digested [3], however, in which an algorithm to construct a tree from its preorder and inorder traversal is derived by means of program inversion, it appeared to us that the same approach should be applicable to Bird’s problem. Surprisingly so, the technique of program inversion led rather straightforwardly to a nice conversion algorithm.

In fact, the only problem we encountered was to find suitable conditions A and B, and, in retrospect, it turns out that an investigation of the relations between *lt.s*, *m.t*, and *mhd.q* does the job. In order to arrive at a solution for A and B, however, we had to replace H by \widehat{H} . This has to do with the fact that t is in general not uniquely determined by $H.t \wedge \bar{t} = S$, but it is, for instance, by $\widehat{H}.t \wedge \bar{t} = S$.

We have confined the data refinement to the essential steps, viz. the elimination of type $[\langle \text{Int} \rangle]$ and the representation of type $\langle \text{Int} \rangle$ either by a pointer type or by an array type. We remark that it is crucial that the first step makes q and t of the same type, so that the **new** and **dispose** operations cancel out in the second step—leading to a constant-space solution.

The incentive to record the data refinement of the conversion algorithm has been its application in the sorting algorithm. As for the efficiency of procedures C2 and C3 we see that they are optimal with respect to time as well as space. We have found a similar result for the reconstruction of a binary tree from its preorder and inorder traversals in [1]. We remark that the latter result may also be achieved by refining the algorithm derived—by program inversion—in [3]. Compared to the way this result is achieved in [1], we observe that such an approach gives a much better separation of concerns; e.g., we do not have to discuss a tricky implementation of the “recursion stack,” a discussion in which algorithmic details and the representation of trees play a role at the same time.

Apart from the adaptive sorting algorithm [7], the conversion algorithm has many other applications. For example, the “largest rectangle under a histogram” can easily be computed once the histogram—which is just a list of natural numbers—has been converted into the corresponding heap (see [2]). Other applications of these heaps can be found in [4], which also contains a description of a linear-time conversion algorithm. In [4] the heaps are called “Cartesian trees” after Vuillemin, who introduced these structures in [8].

References

1. Andersson A., Carlsson S.: Construction of a tree from its traversals in optimal time and space. *Information Processing Letters* **34** (1990) 21–25
2. Bird R.S.: *Lectures on Constructive Functional Programming*. Technical monograph PRG **69**, Oxford University Computing Laboratory (1988)
3. Chen W., Udding J.T.: Program inversion: more than fun! *Science of Computer Programming* **15** (1990) 1–13
4. Gabow H.N., Bentley J.L., Tarjan R.E.: Scaling and related techniques for

- geometry problems. Proc. 16th Annual ACM Symposium on Theory of Computing (1984) 135–143
5. Gries D.: Inorder traversal of a binary tree. In: E.W. Dijkstra (ed.), *The Formal Development of Programs and Proofs*, Addison-Wesley, Amsterdam (1990)
 6. Gries D., v.d. Snepscheut J.L.A.: Inorder traversal of a binary tree and its inversion. In: E.W. Dijkstra (ed.), *The Formal Development of Programs and Proofs*, Addison-Wesley, Amsterdam (1990)
 7. Levcopoulos Ch., Petersson O.: Heapsort—adapted for presorted files. In: F. Dehne, J.-R. Sack, N. Santoro (eds.), *Algorithms and Data Structures*, LNCS **382** (1989) 499–509
 8. Vuillemin J.: A unifying look at data structures. *Communications of the ACM* **23** (1980) 229–239

Acknowledgement

One of the referees is acknowledged for pointing out references [8] and [4].