ELSEVIER

Science of Computer Programming 26 (1996) 149–165

Science of Computer Programming

Leaf trees

Anne Kaldewaij [a,*], Victor J. Dielissen [b]

[a] *Department of Physics, Astronomy, Mathematics and Computer Science, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands*

[b] *Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

Abstract

Leaf trees provide simple and efficient implementations of abstract datatypes that are related to so-called dynamic structures. For functions on such structures the notion of decomposability is defined. If operations defined on a dynamic structure can be expressed as decomposable functions, the leaf tree representing the structure is easily extended to accommodate efficient computations of function values. In a systematic, calculational way, decomposable functions are derived from the specifications of the required operations.

1. Introduction

Classical abstract data types, such as priority queues and dictionaries, are usually implemented by binary trees or variations thereof (cf. [8]). The elements of the set or bag represented by the tree are stored in the nodes of the tree. When the abstract data type is extended with new operations, it is often difficult to extend the corresponding tree in such a way that efficient implementations for these operations are obtained.

In this paper we use the leaf tree (as we call it): the elements of the set, bag, or list represented by the tree are stored in the leaves. It turns out that reasoning about leaf trees, and extending leaf trees in such a way that efficient implementations of required operations on leaf trees are obtained, is much easier than in the case of node-oriented trees.

An important aspect in the derivation of leaf tree structures and accompanying programs is the decomposability of functions. Informally, a function on trees is called decomposable if the function value of a tree is easily expressed in terms of the function values of its subtrees. Leaf-oriented trees, as well as divide-and-conquer techniques have been known for a long time, and also notions of "decomposability" have been exploited before (cf. [1]). They have, however, not been analysed in a systematic way,

* Corresponding author. E-mail: Kaldeway@fwi.uva.nl.

SSDI 0167-6423(95)00026-7

and the relation between decomposability and leaf trees has not been recognized explicitly. Moreover, the method described in this paper supports calculational derivations.

We provide three examples of the use of leaf trees. The first example is about functions on lists and reads as follows. Suppose that on a variable of type list the operations "insert an element at a given position into the list" and "remove an element at a given position from the list" are to be implemented. Furthermore, a function is required for the computation of the maximum sum over all non-empty segments (consecutive sublists) of the list. The fact that elements may be added or removed at any position in the list is often captured by the name "dynamic lists". Our solution to this problem shows how leaf trees can be used conveniently to implement dynamic lists in such a way that programs for insertions and deletions have a time complexity logarithmic in the size of the list and computation of the required function takes constant time.

The second example shows how dictionaries can be implemented by leaf trees. Elements of a totally ordered universe may be added to and removed from a set (the set is "dynamic"). Queries with respect to the set are

- The membership test.
- Given values p and q, report the elements of the set in the range $[p..q)$.

The solution yields $O(\log N)$ time complexity for insertions and deletions, where N is the size of the set. For the second query, a program is obtained that has time complexity $O(\log(N) +$ "the size of the answer").

In the final example, we show how flexible arrays can be implemented by leaf trees.

1.1. Overview

Definitions, notations, and the principal idea of decomposing functions are presented in Section 2. These are illustrated by small examples.

Section 3 discusses implementations of leaf trees and accompanying decomposable functions. It is assumed that the reader is familiar with implementations of classical data structures, such as binary search trees and lists. A simple implementation of leaf trees is explained, resulting in a data structure with $O(\log n)$ time complexity for insertion and deletion (where n is the number of leaves), and $O(1)$ time complexity for the computation of specific function values.

Section 4 emphasizes the method of decomposition. In a systematic way, decompositions of functions are derived from their specifications. The result is a collection of functions that, taken together, constitute a decomposable function.

In Section 5 we show how dictionary and priority queue operations can be implemented by means of operations on leaf trees. This section nicely demonstrates the advantages of leaf trees compared to node-oriented trees.

Section 6 shows how a flexible implementation of flexible arrays can be obtained in a systematic way. This results in trees for which the height is logarithmic in the size, although no rotations are needed.

2. Definitions and notations

Leaf trees are defined as follows. Let Ω be a set. The set $\mathcal{L}$ of *leaf trees* over Ω is the smallest set for which

(i) $\langle\,\rangle \in \mathcal{L}$ (the *empty* tree)
(ii) $a \in \Omega \;\Rightarrow\; \langle a\rangle \in \mathcal{L}$ (a *leaf*)
(iii) $l,r \in \mathcal{L}\backslash\{\langle\,\rangle\} \;\Rightarrow\; \langle l,r\rangle \in \mathcal{L}$ (a *node*)

Note that the empty tree does not occur as subtree of a non-empty tree. Leaf trees are full binary trees: nodes have two successors and leaves have zero successors.

For leaf tree t, we define the list of t, the bag of t, and the set of t by the so-called *abstraction functions* list, bag, and set, respectively, defined by

$$\begin{aligned}
&\text{list.}\langle\,\rangle &&= [\,] && \text{(the empty list)}\\
&\text{list.}\langle a\rangle &&= [a] && \text{(a singleton list)}\\
&\text{list.}\langle l,r\rangle &&= \text{list.}l +\!\!+ \text{list.}r && \text{(concatenation of lists)}
\end{aligned}$$

$$\begin{aligned}
&\text{bag.}\langle\,\rangle &&= \lfloor\,\rfloor && \text{(the empty bag)}\\
&\text{bag.}\langle a\rangle &&= \lfloor a\rfloor && \text{(a singleton bag)}\\
&\text{bag.}\langle l,r\rangle &&= \text{bag.}l + \text{bag.}r && \text{(bag summation)}
\end{aligned}$$

$$\begin{aligned}
&\text{set.}\langle\,\rangle &&= \emptyset && \text{(the empty set)}\\
&\text{set.}\langle a\rangle &&= \{a\} && \text{(a singleton set)}\\
&\text{set.}\langle l,r\rangle &&= \text{set.}l \cup \text{set.}r && \text{(union of sets)}
\end{aligned}$$

An important property of these abstraction functions is that trees $\langle\langle s,t\rangle,u\rangle$ and $\langle s,\langle t,u\rangle\rangle$ yield the same function value (due to the associativity of $+\!\!+$, $+$ en $\cup$). Phrased differently, rotations in a leaf tree do not affect what is represented. This property can be used to keep leaf trees height-balanced. We will return to this later.

A function $f: \mathcal{L} \to V$ for some set V, is called *decomposable* if

$$f.\langle l,r\rangle = f.l \oplus f.r \text{ for some operator } \oplus \text{ on } V.$$

Since empty trees do not occur as subtrees of a non-empty tree, $f.\langle\,\rangle$ does not play a rôle in this definition. Since bracketing (i.e., $l,r \to \langle l,r\rangle$) is *not* associative, we do not call $\oplus$ a homomorphism.

We are particularly interested in decomposable functions for which the associated operator can be efficiently implemented. When $\oplus$ is an O(1) operator on V, and $f.\langle a\rangle$ is O(1) computable, then f is called O(1)-decomposable.

Examples of O(1)-decomposable functions are the size and the height of a tree, and (for leaf trees over a totally ordered set) the maximum of the set of a non-empty tree. These are defined as follows:

$$\begin{aligned}
&\text{size.}\langle a\rangle &&= 1\\
&\text{size.}\langle l,r\rangle &&= \text{size.}l + \text{size.}r
\end{aligned}$$

$$\text{height}.\langle a\rangle = 0$$

$$\text{height}.\langle l, r\rangle = 1 + (\text{height}.l \uparrow \text{height}.r)$$

$$\max.\langle a\rangle = a$$

$$\max.\langle l, r\rangle = \max.l \uparrow \max.r$$

where $a \uparrow b$ denotes the maximum of a and b.

An example of a non-decomposable function is the balance of a tree: the difference in height between right subtree and left subtree, given by

$$\text{bal}.\langle a\rangle = 0$$

$$\text{bal}.\langle l, r\rangle = \text{height}.l - \text{height}.r$$

Function height, however, is decomposable, and, hence, so is the pair (bal, height), since

$$(\text{bal}, \text{height}).\langle a\rangle = (0, 1)$$

$$(\text{bal}, \text{height}).\langle l, r\rangle = (\text{bal}, \text{height}).l \oplus (\text{bal}, \text{height}).r$$

where

$$(x, h) \oplus (y, k) = (k - h, 1 + (h \uparrow k))$$

(This example is somewhat contrived, since x and y do not occur at the right-hand side of this equality. A more realistic example is given in Section 4.)

In general, we try starting with a specification of a function f, say, to derive a relation between $f.\langle s, t\rangle$, $f.s$ and $f.t$ (we try to find a "decomposition for f"). This may lead to the introduction of other functions (like height in the case of bal). For these functions decompositions are derived as well, until all functions have been decomposed. If this process terminates, the tuple consisting of all functions that were introduced in the derivations is a decomposable function. This process of successively decomposing functions is demonstrated by examples in later sections.

If a function f, defined on lists, satisfies

$$f.(s +\!\!+ t) = f.s \oplus f.t \quad \text{for some operator} \oplus,$$

then the composition $f \circ \text{list}$ is a decomposable function on leaf trees. In such a case, we call f decomposable as well.

3. Implementation aspects

This section has been added for two reasons. Firstly, it shows that leaf trees and extensions thereof can be easily and systematically implemented in terms of lower level constructs, such as records and pointers. Secondly, it shows that the time complexities of solutions derived for the problems presented in subsequent sections are

not theoretical. Indeed, the translation from the presented solutions to solutions in an imperative language is standard, and such a transformation does not introduce new difficulties.

A formal treatment of, for instance, implementations of height-balanced trees is beyond the scope of this paper. We assume that the reader is familiar with implementations of classical data structures, such as search trees and lists. We also assume that pointer implementations of such structures are known. In this section we informally explain a possible leaf tree implementation.

Let f be O(1)-decomposable, and let T be a leaf tree with n leaves. The definition of decomposability defines an O(n) divide-and-conquer algorithm for the computation of $f.T$. Insertions and deletions of leaves take O($\log n$) time when the tree is balanced. Compared to these operations, recomputing $f.T$ after every such modification of T is rather expensive.

To obtain a more efficient solution, the values of f are stored at each subtree of T, so when a local change is made to a tree, the values of f for the unchanged subtrees can be used to recompute f for the changed subtrees, including T. Hence, to each subtree (each node and each leaf) an attribute is added that has as value the function value of that subtree. Since f is O(1)-decomposable, recomputing $f.\langle l, r\rangle$ from $f.l$ and $f.r$ takes O(1) time.

As in the case of node-oriented binary trees, leaf trees can be kept balanced. (Since the height of a tree is a decomposable function, it can be added as attribute to all subtrees.)

Insertion or deletion of a leaf may lead to an unbalanced tree. The balance is restored by rotations, which may occur on the path from a leaf to the root of the tree only. New f-values for a node affected by a rotation can be computed in O(1) time. For instance, rotation $\langle s, \langle t, u\rangle\rangle \rightarrow \langle\langle s, t\rangle, u\rangle$ leads to the computations of $f.s \oplus f.t$ and $(f.s \oplus f.t) \oplus f.u$, using the f-values at s, t, and u.

As additional information, nodes of the tree may contain pointers to the predecessor, the left subtree and the right subtree. Leaves may be provided with pointers to the predecessor in the tree and to the predecessor and successor in the list defined by the tree. In this way, the leaves form a doubly-linked list.

The leaf tree can then be characterized by a descriptor, which contains a pointer to the root of the tree, and pointers to the first and the last element of the list represented by the tree, and possibly other global information (e.g. the number of leaves of the tree).

In this way, a simple and effective implementation of leaf trees is obtained. For each decomposable function, an attribute is added to all nodes and leaves of the tree. The programs for insertions and deletions of leaves are adapted to update the values of these attributes. The result is a data structure with O($\log n$) time complexity for insertion and deletion, and O(1) time complexity for queries about function values.

As shown in Section 6, a careful analysis of alternatives may lead to simpler structures for which, for instance, rotations are not needed.

4. Dynamic lists

A "dynamic list" is a data type consisting of a variable of type list and (at least) insertion and deletion as operations. An insertion has as arguments a position in the list and the value to be inserted at that position. (At an abstract level, the position is given by an index or by a prefix of the list. At a lower level, the position is given by a pointer to the position in the list where the value should be inserted.) A deletion has as argument a position in the list. We use a leaf tree to represent a dynamic list, as explained in the previous sections. As additional operation, we implement the query:

Report the maximum of all sums of non-empty segments of the list.

First, we introduce some definitions and notations to obtain a more precise specification of this query. The predicate "u is a non-empty segment of x" is denoted as $u \,\mathrm{seg}\, x$:

$$u \,\mathrm{seg}\, x \;\equiv\; u \neq [\,] \wedge (\exists v, w :: v +\!\!+ u +\!\!+ w = x)$$

We denote "u is a non-empty prefix of x" as $u \,\mathrm{prefix}\, x$ and "u is a non-empty postfix of x" as $u \,\mathrm{postfix}\, x$:

$$u \,\mathrm{prefix}\; x \;\equiv\; u \neq [\,] \wedge (\exists w :: u +\!\!+ w = x)$$
$$u \,\mathrm{postfix}\, x \;\equiv\; u \neq [\,] \wedge (\exists w :: w +\!\!+ u = x)$$

We are interested in the function f, defined on lists of integers by

$$f.x = (\,\max u :\; u \,\mathrm{seg}\, x :\; \mathrm{sum}.u)$$

where sum is defined by

$$\mathrm{sum}.[a] = a$$
$$\mathrm{sum}.(x +\!\!+ y) = \mathrm{sum}.x + \mathrm{sum}.y$$

To derive a decomposition of f, we consider $f.[a]$ first:

$$\begin{array}{ll} & f.[a] \\ = & \quad \{\,\text{definition of } f\,\} \\ & (\,\max u :\; u \,\mathrm{seg}\, [a] :\; \mathrm{sum}.u) \\ = & \quad \{\, u \,\mathrm{seg}\, [a] \;\equiv\; u = [a]\,\} \\ & \mathrm{sum}.[a] \\ = & \quad \{\,\text{definition sum}\,\} \\ & a \end{array}$$

Next, $f.(x +\!\!+ y)$ is considered, for $x \neq [\,]$ and $y \neq [\,]$:

$$\begin{array}{ll} & f.(x +\!\!+ y) \\ = & \quad \{\,\text{definition of } f\,\} \end{array}$$

$$
\begin{aligned}
& (\max u:\ u \operatorname{seg}(x +\!\!+ y):\ \text{sum}.u) \\
= & \quad \{\text{case analysis}\} \\
& (\max u:\ u \operatorname{seg} x:\ \text{sum}.u) \uparrow (\max u:\ u \operatorname{seg} y:\ \text{sum}.u) \uparrow \\
& (\max v, w:\ v \operatorname{postfix} x \wedge w \operatorname{prefix} y:\ \text{sum}.(v +\!\!+ w)) \\
= & \quad \{\text{definition of } f\} \\
& f.x \uparrow f.y \uparrow (\max v, w:\ v \operatorname{postfix} x \wedge w \operatorname{prefix} y:\ \text{sum}.(v +\!\!+ w)) \\
= & \quad \{\text{definition of sum}\} \\
& f.x \uparrow f.y \uparrow (\max v, w:\ v \operatorname{postfix} x \wedge w \operatorname{prefix} y:\ \text{sum}.v + \text{sum}.w) \\
= & \quad \{\text{distribution of addition over maximum}\} \\
& f.x \uparrow f.y \uparrow ((\max v:\ v \operatorname{postfix} x:\ \text{sum}.v) + (\max w:\ w \operatorname{prefix} y:\ \text{sum}.w)) \\
= & \quad \{\text{see below}\} \\
& f.x \uparrow f.y \uparrow (g.x + h.y)
\end{aligned}
$$

where g and h are defined by

$$g.x = (\max u:\ u \operatorname{postfix} x:\ \text{sum}.u)$$

$$h.x = (\max u:\ u \operatorname{prefix} x:\ \text{sum}.u)$$

For function g, we derive

$$
\begin{aligned}
& g.[a] \\
= & \quad \{\text{definition of } g\} \\
& (\max u:\ u \operatorname{postfix} [a]:\ \text{sum}.u) \\
= & \quad \{u \operatorname{postfix} [a] \equiv u = [a]\} \\
& \text{sum}.[a] \\
= & \quad \{\text{definition sum}\} \\
& a
\end{aligned}
$$

and, for $x \neq [\,]$ and $y \neq [\,]$,

$$
\begin{aligned}
& g.(x +\!\!+ y) \\
= & \quad \{\text{definition of } g\} \\
& (\max u:\ u \operatorname{postfix} (x +\!\!+ y):\ \text{sum}.u) \\
= & \quad \{\text{case analysis}\} \\
& (\max u:\ u \operatorname{postfix} y:\ \text{sum}.u) \uparrow (\max v:\ v \operatorname{postfix} x:\ \text{sum}.(v +\!\!+ y)) \\
= & \quad \{\text{definition of } g\} \\
& g.y \uparrow (\max v:\ v \operatorname{postfix} x:\ \text{sum}.(v +\!\!+ y))
\end{aligned}
$$

$=$ { definition of sum }

$g.y \uparrow (\max v:\ v \text{ postfix } x:\ \text{sum}.v + \text{sum}.y)$

$=$ { distribution of addition over maximum, definition of g }

$g.y \uparrow (g.x + \text{sum}.y)$

For reasons of symmetry, a similar result is obtained for function h. Summarizing the results, we have

$$f.[a] = a$$

$$f.(x +\!\!+ y) = f.x \uparrow f.y \uparrow (g.x + h.y)$$

$$g.[a] = a$$

$$g.(x +\!\!+ y) = g.y \uparrow (g.x + \text{sum}.y)$$

$$h.[a] = a$$

$$h.(x +\!\!+ y) = h.x \uparrow (\text{sum}.x + h.y)$$

$$\text{sum}.[a] = a$$

$$\text{sum}.(x +\!\!+ y) = \text{sum}.x + \text{sum}.y$$

Hence, the quadruple (f, g, h, sum) is an O(1)-decomposable function. Implementing the list as indicated in Section 3, yields an O(1) solution to the query. Insertions and deletions have a time complexity that is logarithmic in the size of the list.

Remark. Substitution of singleton list $[a]$ for y in the relations above, yields

$$f.[a] = a$$

$$f.(x +\!\!+ [a]) = f.x \uparrow a \uparrow (g.x + a)$$

$$g.[a] = a$$

$$g.(x +\!\!+ [a]) = a \uparrow (g.x + a)$$

This gives a linear time program for the computation of $f.x$ when the elements of x are inspected successively, which is optimal for the static version of this problem.

5. Dynamic sets

In this section, we show how classical operations on a set, such as insertion, deletion, membership test, computing the maximum and computing the minimum are easily implemented in terms of leaf trees.

A "dynamic set" is a data type consisting of a variable of type set with (at least) insertion, deletion and the membership test as operations. We assume that the elements come from a totally ordered set. Such a set can be implemented by what we call a *leaf search tree*. Leaf tree t is a leaf search tree when list.t is increasing.

A straightforward calculation yields as possible solution for the insertion of a value w in a leaf search tree:

$$\text{ins}.\langle\,\rangle.w = \langle w\rangle$$

$$\text{ins}.\langle a\rangle.w = \begin{cases} \langle\langle a\rangle, \langle w\rangle\rangle, & w > a \\ \langle\langle w\rangle, \langle a\rangle\rangle, & w < a \\ \langle a\rangle, & w = a \end{cases}$$

$$\text{ins}.\langle l, r\rangle.w = \begin{cases} \langle \text{ins}.l.w, r\rangle, & w \leq \max.l \\ \langle l, \text{ins}.r.w\rangle, & w > \max.l \end{cases}$$

Function max is decomposable. Note that for leaf search trees, it can be defined by

$$\begin{aligned} \max.\langle a\rangle &= a \\ \max.\langle l, r\rangle &= \max.r \end{aligned}$$

A program for deletion reads

$$\text{del}.\langle\,\rangle.w = \langle\,\rangle$$

$$\text{del}.\langle a\rangle.w = \begin{cases} \langle\,\rangle, & w = a \\ \langle a\rangle, & w \neq a \end{cases}$$

$$\text{del}.\langle l, r\rangle.w = \begin{cases} \langle \text{del}.l.w, r\rangle, & w \leq \max.l \\ \langle l, \text{del}.r.w\rangle, & w > \max.l \end{cases}$$

Since non-empty leaf trees do not have empty trees as subtrees (the above function may yield empty subtrees), we adopt the convention that $\langle t, \langle\,\rangle\rangle = t$ and $\langle\langle\,\rangle, t\rangle = t$, i.e., $\langle\,\rangle$ is the unit element of $s, t \rightarrow \langle s, t\rangle$. Note that deletion in node-oriented trees is much more complicated (see, for instance, the treatment of 'internal search trees' in [13]).

Because height is decomposable, ins and del can be modified in such a way that the trees remain height-balanced. This extension yields logarithmic time complexities for ins and del. Note that inspection of the maximum takes constant time and deletion of the maximum is then logarithmic. Hence, the priority queue operations "max" and "delmax" are easily added to the repertoire of operations. Of course, the same holds for "min" and "delmin".

The membership test is given by

$$\begin{aligned} \text{mem}.\langle\,\rangle.w &= \text{false} \\ \text{mem}.\langle a\rangle.w &= w = a \\ \text{mem}.\langle l, r\rangle.w &= \begin{cases} \text{mem}.l.w, & w \leq \max.l \\ \text{mem}.r.w, & w > \max.l \end{cases} \end{aligned}$$

To be able to compute efficiently all elements of the set within a given range $[p..q)$, it is convenient to have a function that yields, for a given value p, the position (in the

increasing list of values of the set) that divides the list into a part of elements smaller than p and a part of elements greater than p. For this purpose, we specify function access.$t.w$ by

$$\begin{aligned}&\text{access}.t.w = \langle a\rangle\\ &\equiv\\ &(\exists u,v:\ \text{list}.t = u +\!\!+ [a] +\!\!+ v:\ (\forall x:\ x \in u:\ x < w) \wedge (\forall x:\ x \in v:\ w < x))\end{aligned}$$

for $t \neq \langle\,\rangle$. (Since sets are involved, element $a \in \text{set}.t$ identifies a unique leaf of t that contains a.)

A program for access is given by

$$\begin{aligned}\text{access}.\langle\,\rangle.w &= \langle\,\rangle\\ \text{access}.\langle a\rangle.w &= \langle a\rangle\\ \text{access}.\langle l,r\rangle.w &= \begin{cases}\text{access}.l.w, & w \le \max.l\\ \text{access}.r.w, & w > \max.l\end{cases}\end{aligned}$$

Using the low level implementation described in Section 3, access.$t.p$ provides a pointer to an element in list.t. The predecessors of that element are smaller than p and the successors are larger than p. Traversing the list, starting at that element, yields the elements of set.t within the range $[p..q)$ in an efficient way. Since access.$t.p$ is logarithmic in the size n of t, such a query has time complexity O(log(n)+"the size of the answer").

We conclude that leaf search trees are suitable for implementing dictionaries, priority queues, and related abstract data types. For many (practical) applications, the leaf search tree with the operations described in this section, forms the starting point of a design. Additional operations require derivations of suitable decomposable functions.

6. Flexible arrays

6.1. Introduction

Flexible arrays, as described in [4], are an extension of static arrays. In addition to inspection and array element assignment, the size of the array can be changed. A flexible array can grow and shrink at both ends. Flexible array x with elements of type T is characterized by three components: the low bound $x.lb$, the high bound $x.hb$, and $x.v$ ("the array"), which is a function defined on the interval $[x.lb..x.hb)$ of the integers. Hence,

$$x.v:\ [x.lb..x.hb) \to T$$

Operations on x are inspections and updates. The value of $x.v$ in i, for $x.lb \le i < x.hb$, is, as usual, denoted as $x[i]$, and an assignment to this entry is denoted as $x[i] := E$, where E is an expression of type T.

More interesting are the operations hiext.w.x and loext.w.x (high extend and low extend), defined by

$$\begin{array}{ll} hiext.w.x: & x.hb := x.hb + 1; \quad x[x.hb - 1] := w \\ loext.w.x: & x.lb := x.lb - 1; \quad x[x.lb] := w \end{array}$$

Shrinking is possible by operations hirem and lorem (high remove and low remove), given by

$$\begin{array}{ll} hirem.x: & x.hb := x.hb - 1 \\ lorem.x: & x.lb := x.lb + 1 \end{array}$$

Flexible arrays are especially useful for representations of strings to which various operations like substring selection and substring replacement can be applied. The usefulness of being able to extend array bounds by small amounts had been emphasized by Lindsey in [10], who used the simple example of reading in an array of an unknown number of items.

Implementations of flexible arrays can, for instance, be found in [2, 3]. The implementation described in [2], which appears in a derivational style in [5], uses binary trees. The idea proposed in [2] is to store the array values in the internal nodes of a tree in such a way that sizes of left and right subtrees differ by at most one. This implementation, however, cannot be easily extended to accommodate additional operations defined on *segments* of the array, such as the maximal segment sum presented in Section 4.

In [3], (2,3)-trees are used (only hiext and hirem are implemented: the arrays are extendable at one end only). The values are stored in the leaves and the standard node splitting and node fusion algorithms for (a,b)-trees are used (cf. [8]). Although conceptually simple, implementations of these algorithms are rather complicated.

6.2. *One-sided flexible arrays*

In this section, we consider inspection, update, hiext and hirem only. In the next section it is shown how loext and lorem can be added to the repertoire of operations. As lower bound of the arrays, we choose 0, hence, array x is defined on $[0..x.hb)$. As representation of x, we use $x.hb$ and a leaf tree t, such that

$$\text{list}.t = x.v[0..x.hb)$$

Programs for the computation of the ith element of list.t and for an update thereof are easily derived. Since these are the easy parts, we present the solutions without derivation or proof.

A program for the computation of the ith element of x is given by

$$\begin{array}{lll} \text{elt}.i.\langle a\rangle & = a, & i = 0 \\ \text{elt}.i.\langle l, r\rangle & = \begin{cases} \text{elt}.i.l, & 0 \le i < \text{size}.l \\ \text{elt}.(i - \text{size}.l).r, & \text{size}.l \le i < x.hb \end{cases} & \end{array}$$

Similarly, a program corresponding to the assignment of w to the ith element is given by

$$\begin{aligned} \text{upd}.i.w.\langle a\rangle &= \langle w\rangle, && i = 0 \\ \text{upd}.i.w.\langle l,r\rangle &= \begin{cases} \langle \text{upd}.i.w.l, r\rangle, & 0 \le i < \text{size}.l \\ \langle l, \text{upd}.(i-\text{size}.l).w.r\rangle, & \text{size}.l \le i < x.hb \end{cases} \end{aligned}$$

The number of unfoldings that result when these functions are evaluated is bounded by the height of t, hence, their time complexity is O(height.t).

Function size is decomposable and can thus be stored in each node of the tree (its value is not affected by inspections or updates).

The more interesting operations are hiext and hirem, since these operations will affect the structure of the tree involved. We consider high extension first, and we denote the extension of tree t with a value w as hiext.$w.t$. Its specification is

$$\text{list}.(\text{hiext}.w.t) = \text{list}.t +\!\!+ [w]$$

Substitution of $\langle\,\rangle$ for t yields for the right-hand side

$$\begin{aligned} & \text{list}.\langle\,\rangle +\!\!+ [w] \\ = & \quad \{\text{definition of list}\} \\ & [\,] +\!\!+ [w] \\ = & \quad \{\text{definition of concatenation}\} \\ & [w] \\ = & \quad \{\text{definition of list}\} \\ & \text{list}.\langle w\rangle \end{aligned}$$

Hence, hiext.$w.\langle\,\rangle = \langle w\rangle$ is appropriate. A similar derivation for hiext.$w.\langle a\rangle$ yields as result

$$\text{hiext}.w.\langle a\rangle = \langle\langle a\rangle, \langle w\rangle\rangle$$

For $t = \langle l,r\rangle$, we have

$$\begin{aligned} & \text{list}.(\text{hiext}.w.\langle l,r\rangle) \\ = & \quad \{\text{specification of hiext}\} \\ & \text{list}.\langle l,r\rangle +\!\!+ [w] \\ = & \quad \{\text{definition of list}\} \\ & \text{list}.l +\!\!+ \text{list}.r +\!\!+ [w] \end{aligned}$$

This concatenation can be parenthesized in two ways:

$(\text{list}.l +\!\!+ \text{list}.r) +\!\!+ [w]$, which equals $\text{list}.\langle\langle l,r\rangle, \langle w\rangle\rangle$,

and

$\text{list}.l +\!\!+ (\text{list}.r +\!\!+ [w])$, which equals (by induction) $\text{list}.\langle l, \text{hiext}.w.r\rangle$

Hence, two solutions are at hand, viz.,

$$\text{hiext}.w.\langle l,r\rangle = \langle\langle l,r\rangle,\langle w\rangle\rangle \quad \text{and} \quad \text{hiext}.w.\langle l,r\rangle = \langle l, \text{hiext}.w.r\rangle$$

The first one is attractive, since it is an O(1) operation. It yields, however, an increase of the height of the tree to which it is applied. Since the time complexities of inspection and update are related to this height, we wish to keep heights as small as possible. The idea is to apply the first alternative only when adding an element to the tree will *inevitably* increase its height, whatever algorithm is used.

This is precisely the case when the tree is *perfect*, i.e., when all leaves are at the same depth (cf. [13]). It is formally defined by

$$\begin{aligned}\text{perfect}.\langle a\rangle &= \text{true}\\ \text{perfect}.\langle l,r\rangle &= \text{perfect}.l \wedge \text{perfect}.r \wedge \text{size}.l = \text{size}.r\end{aligned}$$

Function perfect is not (strictly) decomposable, however, pair (perfect, size) is a decomposable function.

Note that the height of a non-empty perfect leaf tree with n elements is $\log_2 n$.
As a complete solution, we propose

$$\begin{aligned}\text{hiext}.w.\langle\,\rangle &= \langle w\rangle\\ \text{hiext}.w.\langle a\rangle &= \langle\langle a\rangle,\langle w\rangle\rangle\\ \text{hiext}.w.\langle l,r\rangle &= \begin{cases}\langle\langle l,r\rangle,\langle w\rangle\rangle, & \text{perfect}.\langle l,r\rangle\\ \langle l, \text{hiext}.w.r\rangle, & \neg\text{perfect}.\langle l,r\rangle\end{cases}\end{aligned}$$

For hirem (being the left inverse of hiext) there is hardly a choice: the right-most element has to be removed. This yields

$$\begin{aligned}\text{hirem}.\langle a\rangle &= \langle\,\rangle\\ \text{hirem}.\langle l,r\rangle &= \langle l, \text{hirem}.r\rangle\end{aligned}$$

These programs yield as result leaf trees whose shapes are completely determined by their sizes. Fig. 1 shows the shapes of the trees corresponding to flexible arrays of sizes one through eight. The dots indicate the leaves. The class of trees obtained by successive applications of hiext has a number of nice properties that are easily proved by induction. One important property is being *left-perfect*: all left subtrees are perfect. Formally, this predicate is defined by

$$\begin{aligned}\text{left-perfect}.\langle a\rangle &= \text{true}\\ \text{left-perfect}.\langle l,r\rangle &= \text{perfect}.l \wedge \text{left-perfect}.r\end{aligned}$$

Note that a perfect tree is left-perfect as well. Trees obtained by these operations are also what we call *leftist* trees, i.e., for all subtrees the height of the left subtree is at least the height of the right subtree. Formally, this predicate is defined by

$$\begin{aligned}\text{leftist}.\langle\,\rangle &= \text{true}\\ \text{leftist}.\langle a\rangle &= \text{true}\\ \text{leftist}.\langle l,r\rangle &= \text{height}.l \geq \text{height}.r \wedge \text{leftist}.l \wedge \text{leftist}.r\end{aligned}$$

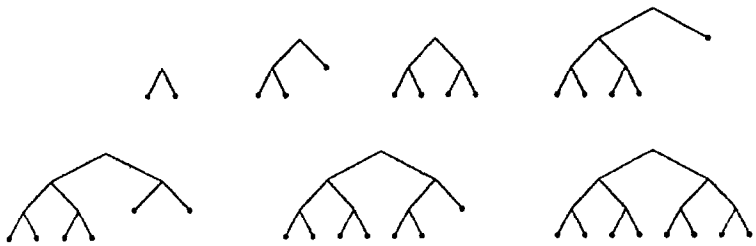

Fig. 1. Trees corresponding to arrays of sizes 1–8.

Since the shape of a leftist left-perfect tree is determined by its size, the class of trees generated by hiext and hirem operations, starting with the empty tree, is precisely the class of leftist left-perfect trees.

For a leftist tree $\langle l,r\rangle$, we have $h.\langle l,r\rangle = 1 + h.l$. Since the trees of this class are left-perfect as well, we have $h.l = \log_2(\text{size}.l)$, and, hence

$$h.\langle l,r\rangle = 1 + \log_2(\text{size}.l)$$

We conclude that this implementation of (one-sided) flexible arrays yields logarithmic time complexities for the operations required. As additional information in the nodes of the tree, perfect (a boolean) and size (an integer) are needed as attributes.

Without proof, we mention that for a leftist left-perfect tree

$$\begin{aligned} &\text{perfect}.\langle l,r\rangle &&\equiv \text{size}.l = \text{size}.r \\ &\neg\text{perfect}.\langle l,r\rangle &&\equiv \text{size}.l > \text{size}.r \end{aligned}$$

Hence, attribute perfect may be omitted, and hiext can be simplified to

$$\begin{aligned} &\text{hiext}.w.\langle\,\rangle &&= \langle w\rangle \\ &\text{hiext}.w.\langle a\rangle &&= \langle\langle a\rangle,\langle w\rangle\rangle \\ &\text{hiext}.w.\langle l,r\rangle &&= \begin{cases} \langle\langle l,r\rangle,\langle w\rangle\rangle, & \text{size}.l = \text{size}.r \\ \langle l,\text{hiext}.w.r\rangle, & \text{size}.l > \text{size}.r \end{cases} \end{aligned}$$

6.3. *Fully flexible arrays*

In the previous section, we showed how one-sided flexible arrays can be implemented. If only operations hiext and hirem are used, this gives rise to the class of leftist left-perfect leaf trees, which we will call ll-trees for short. Of course, its symmetric counterpart, in which only operations loext and lorem are used, yields the class of rightist right-perfect leaf trees, called rr-trees for short. The problem is how to combine these into a structure that allows all operations.

One idea is to represent flexible array x by a pair (s,t), such that s is an rr-tree, t is an ll-tree and

$$x[x.lb\,..\,x.hb) = \text{list}.s +\!\!+ \text{list}.t$$

This works well, but leads to case-analysis for hirem and lorem that complicates the resulting programs. Instead, we combine the programs obtained for the one-sided versions, using auxiliary function lo and hi that operate on rr-trees and ll-trees, respectively. This results in the following programs:

$$
\begin{aligned}
\text{hiext}.w.\langle\,\rangle &= \langle w\rangle\\
\text{hiext}.w.\langle a\rangle &= \langle\langle a\rangle,\langle w\rangle\rangle\\
\text{hiext}.w.\langle l,r\rangle &= \langle l,\text{hi}.w.r\rangle
\end{aligned}
$$

$$
\begin{aligned}
\text{loext}.w.\langle\,\rangle &= \langle w\rangle\\
\text{loext}.w.\langle a\rangle &= \langle\langle w\rangle,\langle a\rangle\rangle\\
\text{loext}.w.\langle l,r\rangle &= \langle \text{lo}.w.l,r\rangle
\end{aligned}
$$

$$
\begin{aligned}
\text{hi}.w.\langle a\rangle &= \langle\langle a\rangle,\langle w\rangle\rangle\\
\text{hi}.w.\langle l,r\rangle &= \begin{cases} \langle\langle l,r\rangle,\langle w\rangle\rangle, & \text{size}.l = \text{size}.r\\ \langle l,\text{hi}.w.r\rangle, & \text{size}.l > \text{size}.r \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\text{lo}.w.\langle a\rangle &= \langle\langle w\rangle,\langle a\rangle\rangle\\
\text{lo}.w.\langle l,r\rangle &= \begin{cases} \langle\langle w\rangle,\langle l,r\rangle\rangle, & \text{size}.r = \text{size}.l\\ \langle \text{lo}.w.l,r\rangle, & \text{size}.r > \text{size}.l \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\text{hirem}.\langle a\rangle &= \langle\,\rangle\\
\text{hirem}.\langle l,r\rangle &= \langle l,\text{hirem}.r\rangle
\end{aligned}
$$

$$
\begin{aligned}
\text{lorem}.\langle a\rangle &= \langle\,\rangle\\
\text{lorem}.\langle l,r\rangle &= \langle \text{lorem}.l,r\rangle
\end{aligned}
$$

It is easily shown (using induction) that for a tree $\langle l,r\rangle$ corresponding to a flexible array, its left subtree l is an rr-tree and its right subtree r is an ll-tree. As shown in the previous section the heights of these subtrees are logarithmic in their sizes. Hence, the height of tree $\langle l,r\rangle$ is logarithmic in its size. Fig. 2 shows the possible trees corresponding to an array of size 6.

Programs for inspection and assignment are similar to those in the previous section. In this general case, the index of an element is related to the bounds of the array and an element with index i in the array corresponds to the element with index $i - x.lb$ in the list of the tree representing the array.

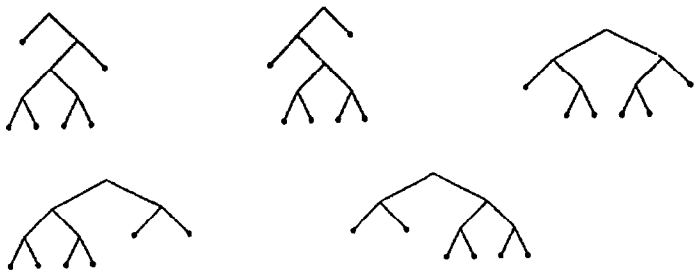

Fig. 2. The trees corresponding to arrays of size 6.

7. Concluding remarks

It is our experience, that leaf trees are much easier to manipulate than corresponding node-oriented trees. The fact that (for leaf search trees) programs for deletion and insertion are similar, as opposed to their counterparts in node-oriented trees, is one reason to prefer leaf trees. Another reason is that decomposability of functions combines so nicely with the use of leaf trees. The decomposition method allows a calculational derivation style, and the resulting programs (decompositions) are, compared to traditional descriptions of algorithms in this area, short and easy to read. Moreover, implementing these programs in imperative program notations like C or Pascal, is a straightforward activity. Starting with a (balanced) elementary leaf search tree, functions can be implemented one at a time by adding attributes to nodes and leaves, and adding statements to insert and delete. For each function these statements follow the same pattern.

Recent research indicates promising results for solutions to other long-known, not satisfactorily solved, problems. In particular, two-dimensional problems from the area of computational geometry, such as the contour of a set of rectangles, are interesting case studies. Other applications are leaf heaps and forests of leaf trees.

References

[1] J.L. Bentley, Decomposable searching problems, *Inform. Process. Lett.* **8** (1979) 244–251.
[2] W. Braun and M. Rem, A logarithmic implementation of flexible arrays, Memorandum MR83/4, University of Technology Eindhoven, 1983.
[3] D.J. Challab, Implementation of flexible arrays using balanced trees, *Comput. J.*, **34**(5) (1991) 386–396.
[4] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[5] R.R. Hoogerwoord, A logarithmic implementation of flexible arrays, in: R.S. Bird, C.C. Morgan and J.C.P. Woodcock, eds., *Mathematics of Program Construction: Proc. 2nd Internat. Conf.*, Oxford, UK, 1992; Lecture Notes in Computer Science, Vol. 669 (Springer, Berlin, 1993) 191–207.
[6] A. Kaldewaij, *Programming. The Derivation of Algorithms*, Prentice-Hall International Series in Computer Science, 1990, 67–69.
[7] A. Kaldewaij and V.J. Dielissen, Decomposable functions and leaf trees: a systematic approach, in: E.-R. Olderog, ed., *Proc. IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conf. on Programming Concepts, Methods and Calculi (PROCOMET'94)*, San Miniato, Italy, 1994 (Elsevier, Amsterdam, 1994) 3–17.
[8] D.E. Knuth, The Art of Computer Programming 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[9] H.R. Lewis and L. Denenberg, Data Structures and Their Algorithms (Harper Collins Publishers, 1991).
[10] C.H. Lindsey, ALGOL-68 with fewer tears, Algol Bulletin 28 (1968), Science Reference Library, London, UK.
[11] M.H. Overmars, Dynamization of order decomposable set problems, *J. Algorithms* **2** (1981) 245–260.
[12] P. Wadler, Linear types can change the world!, in: M. Broy and C.B. Jones, eds., *Programming Concepts and Methods* (North-Holland, Amsterdam, 1990) 561–581.
[13] D. Wood, *Data structures, Algorithms, and Performance* (Addison-Wesley, Reading, MA, 1993).