RAIRO. INFORMATIQUE THÉORIQUE

R. KEMP

The reduction of binary trees by means of an input-restricted deque

RAIRO. Informatique théorique, tome 17, n° 3 (1983), p. 249-284

<http://www.numdam.org/item?id=ITA_1983__17_3_249_0>

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 17, n° 3, 1983, p. 249 à 284)

THE REDUCTION OF BINARY TREES BY MEANS OF AN INPUT-RESTRICTED DEQUE (*)

by R. KEMP (1)

Communicated by J. BERSTEL

Abstract. — *In this paper we present a class of algorithms for the reduction of binary trees. Each algorithm D_k uses an input-restricted deque of length k and an auxiliary store. D_k is a possible generalization of the customary method for the reduction of a tree by means of a stack.*

We shall show that the number of binary trees with n leaves which can be reduced by algorithm D_k with at most i auxiliary cells is equal to the number of all trees with n leaves which can be reduced with a stack of maximum length $(k+1)(i+1)-1$. The corresponding sets are different. This fact implies that an algorithm D_k reduces all binary trees with n leaves if and only if D_k has at least $\lfloor n/(k+1) \rfloor$ auxiliary cells.

Finally, we shall present a detailed average case analysis of the space complexity of an algorithm D_k.

Résumé. — *Dans cet article nous présentons une classe d'algorithmes pour la réduction d'arbres binaires. Chaque algorithme D_k utilise une double queue avec restriction d'entrée de longueur k et une mémoire auxiliaire. D_k est une des généralisations possibles de la méthode habituelle de réduction d'un arbre à l'aide d'une pile.*

Nous montrons que le nombre d'arbres binaires à n feuilles qui peuvent être réduits par l'algorithme D_k avec au plus i cellules mémoire auxiliaires est égal au nombre de tous les arbres à n feuilles qui peuvent être réduits avec une pile de hauteur au plus $(k+1)(i+1)-1$. Les ensembles correspondants sont différents. Ce fait implique qu'un algorithme D_k réduit tous les arbres binaires à n feuilles si D_k a au moins $\lfloor n/(k+1) \rfloor$ cellules auxiliaires.

Finalement, nous présentons une analyse détaillée de la complexité en espace moyenne d'un algorithme D_k.

1. INTRODUCTION

The evaluation of expressions plays an important part in the compilers for programming languages. An expression may be represented by an *extended binary tree* ([8]; p. 399) provided that it consists of brackets, binary operators and operands; the operators correspond to the interior nodes and

(*) Received March 1981, revised January 1983.

(1) Johann Wolfgang Goethe-Universität, Fachbereich Informatik (20), 6000 Frankfurt a.M., R.F.A.

R.A.I.R.O. Informatique théorique/Theoretical Informatics, 0399-0540/1983/249/$ 5.00

the operands to the leaves. For example, the arithmetical expression $E=x_1/((x_2-x_3)\uparrow((x_4+x_5)*x_6))$ can be represented by the binary tree (*syntax tree*) drawn in Figure 1.

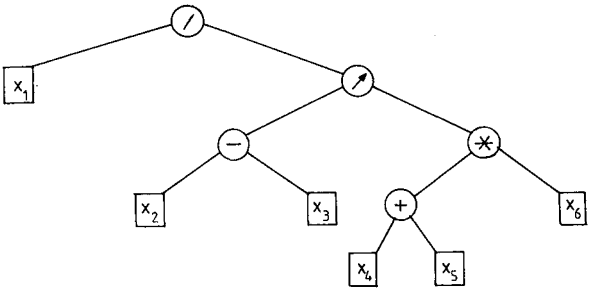


Figure 1

The evaluation of an expression is equivalent to the reduction of the corresponding tree according to its structure. These reductions are closely related to the process of code generation in compilers. For example, in order to evaluate E for given values of $x_1, x_2, x_3, x_4, x_5, x_6$, we have to use *registers* R_j, $j=1, 2, 3, \ldots$, and produce code such as:

Code 1		*Code* 2
$R_1 \leftarrow x_1$		$R_1 \leftarrow x_5$
$R_2 \leftarrow x_2$		$R_2 \leftarrow x_4$
$R_3 \leftarrow x_3$		$R_1 \leftarrow R_2+R_1$
$R_2 \leftarrow R_2-R_3$		$R_2 \leftarrow x_6$
$R_3 \leftarrow x_4$		$R_1 \leftarrow R_1*R_2$
$R_4 \leftarrow x_5$	or	$R_2 \leftarrow x_3$
$R_3 \leftarrow R_4+R_5$		$R_3 \leftarrow x_2$
$R_4 \leftarrow x_6$		$R_2 \leftarrow R_3-R_2$
$R_3 \leftarrow R_3*R_4$		$R_1 \leftarrow R_2\uparrow R_1$
$R_2 \leftarrow R_2\uparrow R_3$		$R_2 \leftarrow x_1$
$R_1 \leftarrow R_1/R_2$		$R_1 \leftarrow R_2/R_1$

which leaves the desired result in register R_1, using the registers R_2, R_3, R_4 (code 1) and R_2, R_3 (code 2) for storing intermediate values. Obviously, the second code is better than the previous one, since it uses one less register. There are well-known strategies for the computation of expressions formed with binary operators:

(*a*) " left-to-right" strategy based on a simple stack;

(*b*) "optimal" strategy (with respect to the number of registers).

Before discussing the algorithms induced by these strategies in more detail, we will briefly formalize the concept of the *reduction* of a binary tree T with

the set of interior nodes I, the set of leaves L and the root $r \in I$. An *instruction* Q is a string of symbols of one of the following three types:

(a) $A \leftarrow \omega$ with $\omega \in L$;

(b) $A \leftarrow A' A'' i$ with $i \in I$;

(c) $A \leftarrow A'$.

In these instructions, A, A' and A'' are *intermediate variables* (possibly the same). The *value* of an intermediate variable is always a string over $(I \cup L)^*$. We assume that the variables are numbered $A_1, A_2, \ldots$ in some order for identification. Instructions of type (a) replace the value of A by ω and instructions of type (b) replace the value of A by the concatenation of the values of A', A'' and node i. If A' or A'' is undefined, then A is also undefined. Instructions of type (c) replace the value of A by the value of A'.

A *program* Π is a sequence of instructions $\Pi = Q_1; Q_2; \ldots; Q_m$; Π is called a *k-program*, if there are k distinct variables appearing in Π. The *value of the variable A after instruction* Q_t, denoted by $v_t(A)$, is recursively defined by:

(1) $v_0(A)$ is undefined for all variables A.

(2) Let Q_t be of type (a). Then $v_t(A) = \omega$.

(3) Let Q_t be of type (b). Then $v_t(A) = v_{t-1}(A')\, v_{t-1}(A'')\, i$.

(4) Let Q_t be of type (c). Then $v_t(A) = v_{t-1}(A')$.

(5) If $v_t(A)$ is not defined by (3)-(4) but $v_{t-1}(A)$ has been defined, then $v_t(A) = v_{t-1}(A)$. Otherwise, $v_t(A)$ is undefined.

We say that a program Π *evaluates* a binary tree T if after the last instruction of Π, the variable A_1 has the value $PO(T)$, where $PO(T) \subseteqq (I \cup L)^*$ denotes the *postorder* of the nodes of T ([8]; p. 316). Furthermore, the tree T can be *reduced by a given algorithm ALG* if ALG with input $PO(T)$ computes a program which evaluates T. Obviously, if Π is a program evaluating T, then Π describes the reduction of T according to its structure. The relationship of the reduction of a binary tree T with the evaluation of an expression formed with binary operators should be evident: given an expression E, $PO(T)$ corresponds to the postfix notation of E, T is the syntax tree of E, the algorithm ALG describes the strategy for the computation of the value of E and the intermediate variables of the program Π produced by ALG correspond to the registers appearing in the code for E.

A customary method for the reduction of a tree T is the following algorithm S which uses a stack.

Algorithm S

INPUT: $PO(T) \subseteq (I \cup L)^*$ of a tree T.

OUTPUT: A program which evaluates T.

METHOD:

(1) A triple (j, γ, ρ) will be used to denote a *configuration* of the algorithm:

(*a*) $j \in \mathbb{N}$ represents the location of the input pointer. We assume that the first "input symbol" is the leftmost symbol in $PO(T)$.

(*b*) $\gamma \in (I \cup L)^*$ represents the stack list. The "top" is assumed to be at the right of γ.

(*c*) ρ is a sequence of instructions of type (*a*) and (*b*).

(2) If j is the location of the input pointer, than $c(j)$ is the "current" input symbol.

(3) The initial configuration of the algorithm is $C_0 = (1, \varepsilon, \varepsilon)$.

(4) There are two types of steps. These steps will be described in terms of their effect on the configurations of the algorithm. The heart of the algorithm is to compute successive configurations defined by a "goes to" relation $\perp$. The notation $(j, \gamma, \rho) \perp (j', \gamma', \rho')$ means that if the current configuration is (j, γ, ρ), then we are to go next into configuration (j', γ', ρ'). The two types of move are as follows:

4.1. Let $c(j) \in L$. Then:

$$(j, \gamma, \rho) \perp (j+1, \gamma\, c(j), \rho\, A_{l(\gamma)+1} \leftarrow c(j);)$$

4.2. Let $c(j) \in I$ and $\gamma = \gamma' ab$. Then:

$$(j, \gamma, \rho) \perp (j+1, \gamma'\, c(j), \rho\, A_{l(\gamma)-1} \leftarrow A_{l(\gamma)-1}\, A_{l(\gamma)}\, c(j);)$$

The execution of the algorithm S is as follows:

STEP 1: Starting in the initial configuration, compute successive configurations $C_0 \perp C_1 \perp C_2 \perp \ldots \perp C_i \perp \ldots$ until no further configuration can be computed.

STEP 2: If the last configuration is $(l(PO(T))+1, r, \Pi)$, emit Π and halt; Π is a program which evaluates T.

Obviously, if we number the stack list cells by $1, 2, 3, \ldots$ from the bottom, then the variable A_m corresponds to the m-th cell.

Example 1: Consider the tree $T = (I, L, a)$ with $I = \{a, c, d, e, h\}$ and $L = \{b, f, g, j, k, p\}$ given in Figure 2.

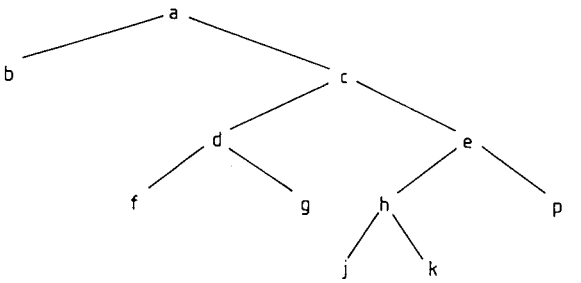


Figure 2

The postorder of the nodes of T is the string $PO(T) = bfgdjkhpeca$. The algorithm S computes the following 4-program which evaluates the tree T. For typographical reasons, we omit the third component in the configurations and give only the current instruction in each move.

Configuration	*Current instruction*	*Move*
$(1, \varepsilon)\perp(2, b)$	$A_1 \leftarrow b;$	4.1
$\perp(3, bf)$	$A_2 \leftarrow f;$	4.1
$\perp(4, bfg)$	$A_3 \leftarrow g;$	4.1
$\perp(5, bd)$	$A_2 \leftarrow A_2 A_3 d;$	4.2
$\perp(6, bdj)$	$A_3 \leftarrow j;$	4.1
$\perp(7, bdjk)$	$A_4 \leftarrow k;$	4.1
$\perp(8, bdh)$	$A_3 \leftarrow A_3 A_4 h;$	4.2
$\perp(9, bdhp)$	$A_4 \leftarrow p;$	4.1
$\perp(10, bde)$	$A_3 \leftarrow A_3 A_4 e;$	4.2
$\perp(11, bc)$	$A_2 \leftarrow A_2 A_3 c;$	4.2
$\perp(12, a)$	$A_1 \leftarrow A_1 A_2 a;$	4.2

Thus algorithm S reduces the tree T using a stack list of maximum length 4. Note that we obtain our code 1 if we replace each variable A_j by the register R_j, each $\alpha \in I \cup L$ by the corresponding operator or operand and each instruction $A_i \leftarrow A_r A_s \alpha$ by $R_i \leftarrow R_r \alpha R_s$. Obviously, the number of variables appearing in a program produced by algorithm S is equal to the maximum number of nodes in the stack during the execution of S. In the worst case, this number is equal to the number of leaves of the tree, in the best case, equal to two [4]. An application of a result given in [1, 4] shows that the average number of variables appearing in a program produced by S is given by $\sqrt{\pi n} - 0.5 + O(\ln(n)/\sqrt{n})$ provided that all trees with n leaves are equally likely.

Another type of algorithms for evaluating a given binary tree T is a procedure consisting in the main of the following two steps.

(A) Attach additional labels to the nodes of the tree.

(B) Convert the labelled tree into a program Π which evaluates T. The following algorithm is of this type and represents an optimal strategy (with respect to the number of variables appearing in Π) for the reduction of binary trees ([9, 10]).

For sake of clarity, we will not describe the algorithm in terms of operations on the postorder $PO(T)$.

Algorithm OP

INPUT: A binary tree T with the set of interior nodes I, the set of leaves L and the root $r \in I$.

OUTPUT: An optimal program Π which evaluates T.

METHOD:

(A) Attach additional labels to the nodes of T. The labels are integers which can be recursively computed by the labelling-function $f: I \cup L \to \mathbb{N}_0$ defined by:

$$f(x) := \textit{IF } x \in L \textit{ THEN } 1$$
$$\textit{ELSE } \mathrm{MIN}(\mathrm{MAX}(f(y), f(z)+1), \mathrm{MAX}(f(y)+1, f(z)));$$

where $y(z)$ is the root of the left (right) subtree of the node x.

(B) The computation of the program Π is as follows:

(B1) One starts from the root of the labelled tree. Scanning is performed from those nodes which have the larger integer label. If both have the same label, one begins on the right node.

(B2) One continues scanning until one has reached a leaf or a node with sons labelled zero. This node is "evaluated" and is substituted by the resulting variable name (zero is substituted for its label). Then one returns to the father of this node and continues scanning. The evaluation of a node x means, that the instruction "$A \leftarrow x$" is emitted if $x \in L$, and the instruction "$A_m \leftarrow A_r A_s\, x$;" with $m = \mathrm{MIN}(r, s)$ if $x \in I$ has the left son A_r and the right son A_s. In each step, the result appears as the value of the available variable with lowest index.

Example 2: Consider the binary tree T given in Example 1.

The corresponding labelled tree is drawn in Figure 3. Performing step (B) we have first to evaluate node $k \in L$. Thus the instruction "$A_1 \leftarrow k$;" is emitted, the label $f(k)$ is replaced by 0 and node k by the variable name A_1. We have to return to node h and to continue scanning. Performing again step (B1), we have now to evaluate node $j \in L$; the instruction "$A_2 \leftarrow j$;" is emitted, the label $f(j)$ is substituted by 0 and node j by A_2. Scanning is continued at node h.

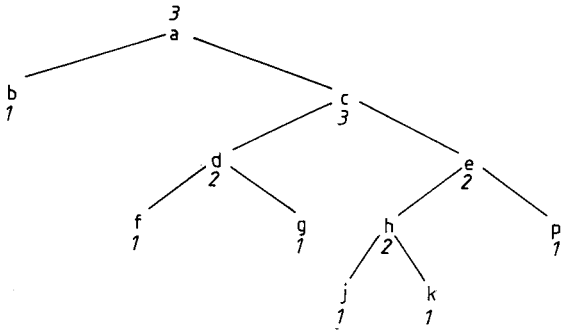


Figure 3

Since h has sons labelled zero, we have to evaluate node h itself. The instruction "$A_1 \leftarrow A_2 A_1 h$;" is emitted, the label $f(h)$ is substituted by 0 and node h by the variable name A_1. Returning to node e, we have next to evaluate node $p \in L$. Finally, after the evaluation of the root a, the following 3-program has been computed:

$$A_1 \leftarrow k;$$
$$A_2 \leftarrow j;$$
$$A_1 \leftarrow A_2 A_1 h;$$
$$A_2 \leftarrow p;$$
$$A_1 \leftarrow A_1 A_2 e;$$
$$A_2 \leftarrow g;$$
$$A_3 \leftarrow f;$$
$$A_2 \leftarrow A_3 A_2 d;$$
$$A_1 \leftarrow A_2 A_1 c;$$
$$A_2 \leftarrow b;$$
$$A_1 \leftarrow A_2 A_1 a;$$

This program is optimal that is, the tree cannot be evaluated by a k-program Π, where $k \leqq 2$. Note that we obtain our code 2 if we replace each variable A_j by the register R_j, each $\alpha \in I \cup L$ by the corresponding operator or operand and each instruction $A_i \leftarrow A_r A_s \alpha$ by $R_i \leftarrow R_r \alpha R_s$.

Using the results derived in [3, 7], the number of variables appearing in an optimal program for a tree T with n leaves is two in the best case, and $\lfloor \log_2(2n) \rfloor$ in the worst case; if all trees with n leaves are equally likely, the average number of variables in an optimal program is given by $\log_4(n) + F(n) + O(1)$, where $F(n)$ is an oscillating bounded function with

$F(n)=F(4n)$, $n\in\mathbb{N}$. Furthermore, the following relation holds: the number of binary trees with n leaves which can be evaluated by an optimal program with less than or equal to k variables is equal to the number of binary trees with n leaves which can be reduced by algorithm S producing a program with less than 2^k variables ([3, 7]).

In this paper we shall present an intermediate class of algorithms for the reduction of binary trees. Each algorithm is a possible generalization of algorithm S and uses only a restricted deque and some auxiliary cells. We shall derive several enumeration results concerning the number of trees which can be reduced by these algorithms D_k. In particular, with well chosen parameters, we shall prove that the worst case (average case) complexity is given by $O(n^{1/2})$ ($O(n^{1/4})$).

2. THE ALGORITHM D_k

Before presenting a formal description of algorithm D_k for the reduction of extended binary trees T, we will first give some basic definitions.

Let T be a binary tree with the set of interior nodes I, the set of leaves L and the root $r\in I$. If $i\in I\cup L$ and $\omega\in L$, the tree T_i^ω is the binary subtree with the set of interior nodes $I'\subseteq I$, the set of leaves $L'\subseteq L$ and the root i, where the leftmost leaf of T_i^ω is ω. Obviously, a given tree T and $i\in I\cup L$ defines an uniquely determined tree T_i^ω. Note that T_i^ω is surely the empty tree if there is no simple path from node i to node ω in T. Using this notation, the tree T_ω^ω is the one-node tree consisting of the leaf ω. Considering the tree of Figure 2, T_a^b is the tree itself, T_e^j is the subtree with the interior nodes e, h, the leaves j, k, p and the root e; T_e^f or T_e^k are empty trees.

Now we shall turn to the presentation of the algorithm D_k for the reduction of a binary tree. The algorithm uses an auxiliary store H, an input-restricted deque ([8]; p. 238) of length $k\in\mathbb{N}$, a counter containing the current position of the input pointer and an auxiliary cell Z. The deque list symbols are triples $(a,b,t)\in(I\cup L)\times(I\cup L)\times\{0,1\}$ representing the subtree T_b^a. If the reduction of the substree T_b^a has required an auxiliary cell in H, then $t=1$; otherwise $t=0$. A triple may be put onto the top of the deque and may be removed from the top or the bottom of the deque. The contents of an auxiliary cell in H is a tuple $(a,b)\in(I\cup L)\times(I\cup L)$ representing the subtree T_b^a; the contents of the auxiliary cell Z is always a node or the symbol \$. Henceforth, we say, that the algorithm D_k is in state x, if x is the contents of the cell Z. D_k will always be in a state $x\in I\cup L$, if (a,x,t) was the last triple which was removed from the bottom of the deque and was stored as the tuple (a,x) in an empty

auxiliary cell in H provided that this cell was not cleared or no other triple was removed from the deque after the deletion of (a, x, t). Otherwise, D_k is in state \$.

First, we shall give an informal description of algorithm D_k.

The algorithm reads the input $PO(T)$ from left to right and works as follows:

1. If the current input symbol is a leaf $\omega \in L$ and there is a tuple (a, b) in an auxiliary cell $h \in H$ with $a = \omega$, then we know that the tree T_b^a ought to be reduced in the following steps. Since this reduction was made in earlier steps, the input pointer moves to the right until the symbol following b, the triple $(a, b, 1)$ is put onto the top of the deque, if the deque list has a length less than k, and the auxiliary cell h is cleared. The state z is unchanged, if $z \neq b$; otherwise, D_k goes in state \$. If the deque list has a length equal to k, then the triple $(a, b, 1)$ cannot be inserted at the top of the deque. In this case, we have to consider the following two cases:

1.1. The input symbol c following b is an interior node. Then the triple (x, y, t) at the top of the deque represents the left subtree T_y^x and (a, b) the right subtree T_b^a of the tree T_c^x. In this case, the triple at the top of the deque is replaced by $(x, c, 1)$, the auxiliary cell h is cleared and the input pointer moves to the right. The state z is unchanged, if $z \neq b$; otherwise, D_k in state \$.

1.2. The input symbol c following b is a leaf. Let (x, y, t) be the triple at the bottom of the deque. If $t = 0$, then the triple $(x, y, 0)$ is removed from the deque, $(a, b, 1)$ is inserted at the top of the deque and the algorithm goes in state \$. If $t = 1$, the triple $(x, y, 1)$ is removed from the deque, the tuple (x, y) is stored in an empty auxiliary cell, the triple $(a, b, 1)$ is put onto the top of the deque and D_k goes in state y.

2. If the current input symbol is a leaf $\omega \in L$ and there is no tuple (a, b) in an auxiliary cell $h \in H$ with $a = \omega$, then we know that we have to reduce a subtree in the subsequent steps which was not reduced in earlier steps or was reduced and then forgotten because the corresponding triple was removed from the bottom of the deque. In this case, the triple $(\omega, \omega, 0)$ is inserted at the top of the deque, if the length of the deque list is less than k, the input pointer moves to the right and the state is unchanged. If the length of the deque list is equal to k and the triple at the bottom of the deque is (x, y, t), then (x, y, t) is removed from the deque, (x, y) is stored in an empty auxiliary cell, if $t = 1$, the triple $(\omega, \omega, 0)$ is put onto the top of the deque, the input pointer moves to the right and the algorithm goes in state y. If $t = 0$, the triple $(x, y, 0)$ will be forgotten, that is, the tuple (x, y) is not stored in an auxiliary cell and the algorithm goes in state \$.

3. If the current input symbol i is an interior node of T, then we have to consider the following two cases:

3.1. If the length of the deque list is $d \in [2 : k]$, then the string) at the top of the deque represents the left subtree T_a^x and the right subtree T_b^y of the tree T_i^x. At this stage, the input pointer moves to the right, the string at the top of the deque is replaced by $(x, i, t_1 \vee t_2)$ and the state is unchanged. Here, $t_1 \vee t_2$ is the disjunction of t_1 and t_2.

3.2. Let the length of the deque list equal to one. The triple (x, y, t) on the deque represents a subtree T_y^x. First, let the algorithm in state $z \in I \cup L$. We know then, that the last triple which was removed from the bottom of the deque and was stored as a tuple in an empty auxiliary cell h has the form (a, z, t'). This triple represents a tree T_z^a. The node z must be the left brother of y, because there is no other triple which was removed from the bottom of the deque after the deletion of (a, z, t'). Thus the input pointer moves to the right, the triple (x, y, t) is replaced by $(a, i, 1)$, the auxiliary cell h is cleared and the algorithm goes in state \$. Next, if the algorithm is in state \$, then we do not know the left brother of y. In this case, the input pointer goes back to the first position of $PO(T)$, the triple (x, y, t) is removed from the deque, (x, y) is stored in an empty auxiliary cell and the algorithm goes in state y.

To describe the algorithm D_k precisely, we shall use again a stylized notation similar to algorithm S given in section 1.

Algorithm D_k

INPUT: $PO(T) \subseteqq (I \cup L)^*$.

OUTPUT: A program which evaluates T.

METHOD:

(1) A 5-tuple $(z, j, \gamma, \underline{H}, \rho)$ will be used to denote a *configuration* of the algorithm:

(a) $z \in I \cup L \cup \{\$\}$, $\$ \notin I \cup L$, denotes the contents of a special denotes the contents of a special auxiliary cell; z is called the state of the algorithm.

(b) $j \in \mathbb{N}$ represents the location of the input pointer. We assume that the first "input symbol" is the leftmost symbol in $PO(T)$.

(c) $\gamma \in ((I \cup L) \times (I \cup L) \times \{0, 1\})^*$ represents the input-restricted deque list. The "bottom" (top) is assumed to be at the left (right) of γ. An item may be put onto the top and may be removed from the top or the bottom. The maximum length of the deque list is $k \in \mathbb{N}$. A deque list symbol (a, b, t) represents the tree T_b^a.

(*d*) $H \subseteq \{(a, b)_s \mid a, b \in I \cup L, s \in \mathbb{N}\}$ represents the contents of the nonempty cells in the auxiliary store. We assume that the cells are numbered 1, 2, 3, ... in some order for identification. The element $(a, b)_s \in \underline{H}$ indicates that the tuple (a, b) representing the tree T_b^a is stored in cell s. A tuple is always stored in the empty auxiliary cell with the lowest number.

(*e*) ρ is a sequence of instructions of type (*a*), (*b*) and (*c*). The statement " *DO* $A_{s-1} \leftarrow A_s \; s = \lambda, \eta$;" stands for a sequence of instructions of type (*c*), that is $A_{s-1} \leftarrow A_s$; for $\lambda \leqq s \leqq \eta$. If $\lambda > \eta$, this statement is to be interpreted as a dummy statement.

(2) If (a, b, t) is a deque list symbol, then $t = 1$ ($t = 0$) indicates that the reduction of the tree T_b^a has required at least one (no) auxiliary cell. If $t = 0$, a triple (a, b, t) deleted at the bottom of the deque will be "forgotten". If $t = 1$, the tuple (a, b) will be stored in an empty auxiliary cell and the algorithm goes in state b. The algorithm will always be in state $z \in I \cup L \cup \{\$\}$, if (x, z, t) was the last triple which was removed from the bottom of the deque and was stored as the tuple (x, z) in an empty auxiliary cell h provided that h was not cleared or no other triple was removed from the deque after the deletion of (x, z, t). Otherwise, the algorithm is in state \$.

(3) The contents of the first (second) component of an auxiliary cell with number s is denoted by $\mathrm{pr}_1(s)$ ($\mathrm{pr}_2(s)$); if j is the location of the input pointer, then $c(j)$ is the current input symbol. The notation "$\underline{H} \setminus \{s\}$" means, that the auxiliary cell s is cleared; similarly, "$\underline{H} \cup \{(x, y)_\eta\}$" indicates that the tuple (x, y) is to be stored in the auxiliary cell η. Successive operations of this kind are performed from left to right.

(4) The initial configuration of the algorithm is $C_0 = (\$, 1, \varepsilon, \emptyset, \varepsilon)$.

(5) There are ten types of steps. These steps will be described in terms of their effect on the configurations of the algorithm. The algorithm computes successive configurations defined by a "goes to" relation $\perp$. The notation $(z, j, \gamma, \underline{H}, \rho) \perp (z', j', \gamma', \underline{H}', \rho')$ means that if the current configuration is $(z, j, \gamma, \underline{H}, \rho)$, then we are to go next into the configuration $(z', j', \gamma', \underline{H}', \rho')$. The ten types of move are as follows:

5.1. Let $c(j) \in L$.

5.1.1. There is an auxiliary cell s with $\mathrm{pr}_1(s) = c(j)$. Let $c(\mu) = \mathrm{pr}_2(s)$.

5.1.1.1. If $l(\gamma) \leqq k - 1$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (\underline{z}, \mu + 1, \gamma\,(c(j), c(\mu), 1), \underline{H} \setminus \{s\}, \rho\, A_{l(\gamma)+1} \leftarrow M_s;)$$

(If $z = \mathrm{pr}_2(s)$, then $\underline{z} = \$$; otherwize, $z = \underline{z}$).

COMMENT: This move corresponds to 1 [with $l(\gamma) \leqq k-1$] in the informal description of the algorithm D_k.

5.1.1.2. If $l(\gamma)=k$ with $\gamma=\gamma'(x, y, t)$ and $c(\mu+1)\in I$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (\underline{z}, \mu+2, \gamma'(x, c(\mu+1), 1), \underline{H} \setminus \{s\}, \rho\, A_{l(\gamma)} \leftarrow A_{l(\gamma)} M_s\, c(\mu+1);)$$

(If $z=\mathrm{pr}_2(s)$, then $\underline{z}=\$$; otherwize, $z=\underline{z}$).

COMMENT: This move corresponds to 1.1 in the informal description of the algorithm D_k.

5.1.1.3. If $l(\gamma)=k$ with $\gamma=(x, y, 0)\gamma'$ and $c(\mu+1)\in L$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (\$, \mu+1, \gamma'(c(j), c(\mu), 1), \underline{H} \setminus \{s\}, \rho\, DO\, A_{\lambda-1} \leftarrow A_\lambda\, \lambda=2, k;\ A_k \leftarrow M_s;)$$

COMMENT: This move corresponds to 1.2 (with $t=0$) in the informal description of the algorithm D_k.

5.1.1.4. If $l(\gamma)=k$ with $\gamma=(x, y, 1)\gamma'$ and $c(\mu+1)\in L$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (y, \mu+1, \gamma'(c(j), c(\mu), 1), \underline{H} \setminus \{s\} \cup \{(x, y)_\eta\}, \rho\, M_\eta \leftarrow A_1;\ DO\, A_{\lambda-1} \leftarrow A_\lambda\, \lambda=2, k;\ A_k \leftarrow M_s;)$$

COMMENT: This move corresponds to 1.2 (with $t=1$) in the informal description of the algorithm D_k.

5.1.2. There is no auxiliary cell s with $\mathrm{pr}_1(s)=c(j)$.

5.1.2.1. If $l(\gamma)\leqq k-1$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (z, j+1, \gamma(c(j), c(j), 0), \underline{H}, \rho\, A_{l(\gamma)+1} \leftarrow c(j);)$$

COMMENT: This move corresponds to 2 [with $l(\gamma)\leqq k-1$] in the informal description of the algorithm D_k. Note also that this move corresponds to 4.1 in the definition of algorithm S given in section 1.

5.1.2.2. If $l(\gamma)=k$ with $\gamma=(x, y, 0)\gamma'$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (\$, j+1, \gamma'(c(j), c(j), 0), \underline{H}, \rho\, DO\, A_{\lambda-1} \leftarrow A_\lambda\, \lambda=2, k;\ A_k \leftarrow c(j);)$$

COMMENT: This move corresponds to 2 [with $l(\gamma)=k$ and $t=0$] in the informal description of the algorithm D_k.

5.1.2.3. If $l(\gamma)=k$ with $\gamma=(x, y, 1)\,\gamma'$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (y, j+1, \gamma'(c(j), c(j), 0), \underline{H} \cup \{(x, y)_\eta\}, \rho\, M_\eta \leftarrow A_1;$$
$$DO\ A_{\lambda-1} \leftarrow A_\lambda\ \lambda=2, k;\ A_k \leftarrow c(j);)$$

COMMENT: This move corresponds to 2 [with $l(\gamma)=k$ and $t=1$] in the informal description of the algorithm D_k.

5.2. Let $c(j)\in I$.

5.2.1. If $2\leqq l(\gamma)\leqq k$ with $\gamma=\gamma'(x, y, t_1)(a, b, t_2)$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (z, j+1, \gamma'(x, c(j), t_1 \vee t_2), \underline{H}, \rho\, A_{l(\gamma)-1} \leftarrow A_{l(\gamma)-1}\, A_{l(\gamma)}\, c(j);)$$

COMMENT: This move corresponds to 3.1 in the informal description of the algorithm D_k. Note also that this move corresponds to 4.2 in the definition of algorithm S given in section 1.

5.2.2. If $l(\gamma)=1$ with $\gamma=(x, y, t)$ and there is an auxiliary cell s with $\mathrm{pr}_2(s)=z$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (\$, j+1, (\mathrm{pr}_1(s), c(j), 1), \underline{H} \setminus \{s\}, \rho\, A_1 \leftarrow M_s\, A_1\, c(j);)$$

COMMENT: This move corresponds to 3.2 (with $z\in I \cup L$) in the informal description of the algorithm D_k.

5.2.3. If $l(\gamma)=1$ with $\gamma=(x, y, t)$ and $z=\$$, then:

$$(z, j, \gamma, \underline{H}, \rho) \perp (y, 1, \varepsilon, \underline{H} \cup \{(x, y)_\eta\}, \rho\, M_\eta \leftarrow A_1;)$$

COMMENT: This move corresponds to 3.2 (with $z=\$$) in the informal description of the algorithm D_k.

The execution of the algorithm D_k is as follows:

STEP 1: Starting in the initial configuration, compute successive next configurations $C_0 \perp C_1 \perp C_2 \perp \ldots \perp C_i \perp \ldots$ until no further configuration can be computed.

STEP 2: If the last computed configuration is:

$$(\$, l(PO(T))+1, (x, r, t), \emptyset, \Pi),$$

emit Π and halt; Π is a program which evaluates the three $T=T_r^x$.

Note that the algorithm D_k is deterministic for each $k\in\mathbb{N}$ that is, D_k has at most one choice of move in any configuration. Similar to algorithm S, the variable A_m corresponds to the m-th cell in the deque list, if we number the

deque list cells by 1, 2, 3, ... from the bottom, and the variable M_s corresponds to the auxiliary cell s.

Example 3: Consider the tree of example 1. We have $PO(T) = bfgdjkhpeca$. The algorithm D_k computes the following sequence of configurations. For typographical reasons, we omit the fifth component in the configurations and give only the current instruction in each move.

(*a*) *algorithm* D_1:

configuration	*current instruction*	*move*
(\$, 1, ε, ∅)⊥(\$, 2, (*b*, *b*, 0), ∅)	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 3, (*f*, *f*, 0), ∅)	$A_1 \leftarrow f$;	5.1.2.2
⊥(\$, 4, (*g*, *g*, 0), ∅)	$A_1 \leftarrow g$;	5.1.2.2
⊥(*g*, 1, ε, {(*g*, *g*)$_1$})	$M_1 \leftarrow A_1$;	5.2.3
⊥(*g*, 2, (*b*, *b*, 0), {(*g*, *g*)$_1$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 3, (*f*, *f*, 0), {(*g*, *g*)$_1$})	$A_1 \leftarrow f$;	5.1.2.2
⊥(\$, 5, (*f*, *d*, 1), ∅)	$A_1 \leftarrow A_1 M_1 d$;	5.1.1.2
⊥(*d*, 6, (*j*, *j*, 0), {(*f*, *d*)$_1$})	$M_1 \leftarrow A_1$; $A_1 \leftarrow j$;	5.1.2.3
⊥(\$, 7, (*k*, *k*, 0), {(*f*, *d*)$_1$})	$A_1 \leftarrow k$;	5.1.2.2
⊥(*k*, 1, ε, {(*f*, *d*)$_1$, (*k*, *k*)$_2$})	$M_2 \leftarrow A_1$;	5.2.3
⊥(*k*, 2, (*b*, *b*, 0), {(*f*, *d*)$_1$, (*k*, *k*)$_2$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 5, (*f*, *d*, 1), {(*k*, *k*)$_2$})	$A_1 \leftarrow M_1$;	5.1.1.3
⊥(*d*, 6, (*j*, *j*, 0), {(*f*, *d*)$_1$, (*k*, *k*)$_2$})	$M_1 \leftarrow A_1$; $A_1 \leftarrow j$;	5.1.2.3
⊥(*d*, 8, (*j*, *h*, 1), {(*f*, *d*)$_1$})	$A_1 \leftarrow A_1 M_2 h$;	5.1.1.2
⊥(*h*, 9, (*p*, *p*, 0), {(*f*, *d*)$_1$, (*j*, *h*)$_2$})	$M_2 \leftarrow A_1$; $A_1 \leftarrow p$;	5.1.2.3
⊥(\$, 10, (*j*, *e*, 1), {(*f*, *d*)$_1$})	$A_1 \leftarrow M_2 A_1 e$;	5.2.2
⊥(*e*, 1, ε, {(*f*, *d*)$_1$, (*j*, *e*)$_2$})	$M_2 \leftarrow A_1$;	5.2.3
⊥(*e*, 2, (*b*, *b*, 0), {(*f*, *d*)$_1$, (*j*, *e*)$_2$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 5, (*f*, *d*, 1), {(*j*, *e*)$_2$})	$A_1 \leftarrow M_1$;	5.1.1.3
⊥(\$, 11, (*f*, *c*, 1), ∅)	$A_1 \leftarrow A_1 M_2 c$;	5.1.1.2
⊥(*c*, 1, ε, {(*f*, *c*)$_1$})	$M_1 \leftarrow A_1$;	5.2.3
⊥(*c*, 2, (*b*, *b*, 0), {(*f*, *c*)$_1$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 12, (*b*, *a*, 1), ∅)	$A_1 \leftarrow A_1 M_1 a$;	5.1.1.2

(*b*) *algorithm* D_2:

configuration	*current instruction*	*move*
(\$, 1, ε, ∅)⊥(\$, 2, (*b*, *b*, 0), ∅)	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 3, (*b*, *b*, 0) (*f*, *f*, 0), ∅)	$A_2 \leftarrow f$;	5.1.2.1
⊥(\$, 4, (*f*, *f*, 0) (*g*, *g*, 0), ∅)	$A_1 \leftarrow A_2$; $A_2 \leftarrow g$;	5.1.2.2
⊥(\$, 5, (*f*, *d*, 0), ∅)	$A_1 \leftarrow A_1 A_2 d$;	5.2.1
⊥(\$, 6, (*f*, *d*, 0) (*j*, *j*, 0), ∅)	$A_2 \leftarrow j$;	5.1.2.1
⊥(\$, 7, (*j*, *j*, 0) (*k*, *k*, 0), ∅)	$A_1 \leftarrow A_2$; $A_2 \leftarrow k$;	5.1.2.2
⊥(\$, 8, (*j*, *h*, 0), ∅)	$A_1 \leftarrow A_1 A_2 h$;	5.2.1
⊥(\$, 9, (*j*, *h*, 0) (*p*, *p*, 0), ∅)	$A_2 \leftarrow p$;	5.1.2.1
⊥(\$, 10, (*j*, *e*, 0), ∅)	$A_1 \leftarrow A_1 A_2 e$;	5.2.1
⊥(*e*, 1, ε, {(*j*, *e*)$_1$})	$M_1 \leftarrow A_1$;	5.2.3
⊥(*e*, 2, (*b*, *b*, 0), {(*j*, *e*)$_1$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(*e*, 3, (*b*, *b*, 0) (*f*, *f*, 0), {(*j*, *e*)$_1$})	$A_2 \leftarrow f$;	5.1.2.1
⊥(\$, 4, (*f*, *f*, 0) (*g*, *g*, 0), {(*j*, *e*)$_1$})	$A_1 \leftarrow A_2$; $A_2 \leftarrow g$;	5.1.2.2
⊥(\$, 5, (*f*, *d*, 0), {(*j*, *e*)$_1$})	$A_1 \leftarrow A_1 A_2 d$;	5.2.1
⊥(\$, 10, (*f*, *d*, 0) (*j*, *e*, 1), ∅)	$A_2 \leftarrow M_1$;	5.1.1.1
⊥(\$, 11, (*f*, *c*, 1), ∅)	$A_1 \leftarrow A_1 A_2 c$;	5.2.1
⊥(*c*, 1, ε, {(*f*, *c*)$_1$})	$M_1 \leftarrow A_1$;	5.2.3
⊥(*c*, 2, (*b*, *b*, 0), {(*f*, *c*)$_1$})	$A_1 \leftarrow b$;	5.1.2.1
⊥(\$, 12, (*b*, *a*, 1), ∅)	$A_1 \leftarrow A_1 M_1 a$;	5.1.1.2

(*c*) *algorithm* D_3:

configuration	*current instruction*	*move*
(\$, 1, ε, ∅) ⊥ (\$, 2, (*b*, *b*, 0), ∅)	$A_1 \leftarrow b$;	5.1.2.1
⊥ (\$, 3, (*b*, *b*, 0) (*f*, *f*, 0), ∅)	$A_2 \leftarrow f$;	5.1.2.1
⊥ (\$, 4, (*b*, *b*, 0) (*f*, *f*, 0) (*g*, *g*, 0), ∅)	$A_3 \leftarrow g$;	5.1.2.1
⊥ (\$, 5, (*b*, *b*, 0) (*f*, *d*, 0), ∅)	$A_2 \leftarrow A_2 A_3 d$;	5.2.1
⊥ (\$, 6, (*b*, *b*, 0) (*f*, *d*, 0) (*j*, *j*, 0), ∅)	$A_3 \leftarrow j$;	5.1.2.1
⊥ (\$, 7, (*f*, *d*, 0) (*j*, *j*, 0) (*k*, *k*, 0), ∅)	$A_1 \leftarrow A_2$; $A_2 \leftarrow A_3$; $A_3 \leftarrow k$;	5.1.2.2
⊥ (\$, 8, (*f*, *d*, 0) (*j*, *h*, 0), ∅)	$A_2 \leftarrow A_2 A_3 h$;	5.2.1
⊥ (\$, 9, (*f*, *d*, 0) (*j*, *h*, 0) (*p*, *p*, 0), ∅)	$A_3 \leftarrow p$;	5.1.2.1
⊥ (\$, 10, (*f*, *d*, 0) (*j*, *e*, 0), ∅)	$A_2 \leftarrow A_2 A_3 e$;	5.2.1
⊥ (\$, 11, (*f*, *c*, 0), ∅)	$A_1 \leftarrow A_1 A_2 c$;	5.2.1
⊥ (*c*, 1, ε, {(*f*, *c*)$_1$})	$M_1 \leftarrow A_1$;	5.2.3
⊥ (*c*, 2, (*b*, *b*, 0), {(*f*, *c*)$_1$})	$A_1 \leftarrow b$;	5.1.2.1
⊥ (\$, 11, (*b*, *b*, 0) (*f*, *c*, 1), ∅)	$A_2 \leftarrow M_1$;	5.1.1.1
⊥ (\$, 12, (*b*, *a*, 1), ∅)	$A_1 \leftarrow A_1 A_2 a$;	5.2.1

Thus the reduction of the tree T by algorithm D_1 requires a deque list of length one and two auxiliary cells; therefore, D_1 emits a 3-program which evaluates T. Similarly, algorithm D_2 (D_3) requires a deque list of length two (three) and one auxiliary cell; D_2 emits a 3-program, D_3 emits a 4-program for the tree T. Obviously, algorithm D_k is a possible generalization of algorithm S given in section 1: D_k with no auxiliary cell reduces always a given tree T, if the algorithm S emits a program with $\lambda \leqq k$ variables, because in this case, algorithm D_k is identical to algorithm S; in other words, the deque can be interpreted as a stack. Thus D_k with $k \geqq 4$ reduces T with a deque list of length four and no auxiliary cell; the program for T produced by these algorithms is identical to the program computed by algorithm S in section 1. Note that the tree cannot be reduced by D_1 with one auxiliary cell, because D_1 stops in configuration (\$, 7, (*k*, *k*, 0), {(*f*, *d*)$_1$}). Generally, the algorithm D_k with i auxiliary cells reduces trees which cannot be reduced by D_{k+i} and no auxiliary cell.

Considerung the execution of D_k, it is easy to see that a tree $T = T_r^y$ is reduced if the left subtree $T_1 = T_x^y$ and the right subtree $T_2 = T_a^b$ has already been reduced. The correctness of algorithm D_k should be obvious; the formal proof consists in the main of an induction on the length of $PO(T)$, because in general $PO(T) = PO(T_1)\, PO(T_2)\, r$. Finally, we will note that the use of the auxiliary cell Z representing the state of the algorithm D_k is not really necessary. D_k works also correctly if we eliminate the first component of the configurations and all references to this component. But doing this, the following results become rather complicated.

3. A REDUCTION-STRATEGY INDUCED BY D_k

Henceforth, we shall say that an extended binary tree T is a (k, i)-*tree*, if T can be reduced by algorithm D_k requiring exactly i auxiliary cells. The set of all (k, i)-trees is denoted by $T(k, i)$. Making allowance for the above one-to-

one correspondence between the variables in a program computed by D_k and the deque list cells as well as the auxiliary cells, a tree T is in $T(k, i)$ if and only if D_k computes a program for T with $k+i$ variables. The following lemmata give a characterization of the set $T(k, i)$.

LEMMA 1: *Let T be an extended binary tree with left subtree T_1 and right subtree T_2. We have: $T \in T(k, 0)$ if and only if* (1) $T_1 \in T(k, 0)$ *and* $T_2 \in T(j, 0)$ *for some* $j \in [1 : k-1]$ *or* (2) $T_1 \in T(j, 0)$ *for some* $j \in [1 : k-1]$ *and* $T_2 \in T(k-1, 0)$.

Proof: Obviously, $PO(T) = PO(T_1)\, PO(T_2)\, r$. We have to consider the trees T requiring a deque list of length k and no auxiliary cell. There is exactly one triple on the deque after the reduction of T_1, namely $(x, a, 0)$, where $T_1 = T_a^x$. If T_1 has used a deque list of length k, then a deque list of length $j \leqq k-1$ is available for T_2. This proves (1). If T_1 has used a deque list of length $j \leqq k-1$, then T_2 has to use a deque list of length $k-1$ in order to get $T \in T(k, 0)$. This proves (2).

LEMMA 2: *Let T be an extended binary tree with left subtree T_1 and right subtree T_2. We have: $T \in T(k, 1)$ if and only if* (1) $T_1 \in T(k, 1)$ *and* $T_2 \in T(j, 0)$ *for some* $j \in [1 : k]$ *or* (2) $T_1 \in T(j, 0)$ *for some* $j \in [1 : k]$ *and* $T_2 \in T(k, 1)$ *or* (3) $T_1 \in T(j, 0)$ *for some* $j \in [1 : k]$ *and* $T_2 \in T(k, 0)$.

Proof: Again, $PO(T) = PO(T_1)\, PO(T_2)\, r$. We have to consider the trees T requiring a deque list k and one auxiliary cell during the execution of D_k with input $PO(T)$. There is exactly one triple on the deque after the reduction of T_1, namely (x, a, t), where $T_1 = T_a^x$; furthermore, the available auxiliary cell is empty.

First, we consider the case $t = 1$ that is, T_1 has used a deque list of length k and the auxiliary cell. If a deque list of length $j \leqq k-1$ is sufficient for T_2, then we have $T_2 \in T(j, 0)$ for $j \leqq k-1$ and $T \in T(k, 1)$. If T_2 requires a deque list of length k, then in any move, the triple $(x, a, 1)$ at the bottom of the deque is deleted, the tuple (x, a) is stored in the empty auxiliary cell and the algorithm goes in state a (corresponds to move 5.1.1.4 or 5.1.2.3). Now the auxiliary cell is not available for T_2 so that $T_2 \in T(j, 0)$ for some $j \leqq k$. Therefore, $T_2 \in T(j, 0)$ for $j \leqq k$ if $T_1 \in T(k, 1)$. This proves (1).

Next, let $t = 0$ that is, T_1 has used a deque list of length $j \leqq k$ and no auxiliary cell. T_2 must require a deque list of length k because in the contrary the tree T does not use the available auxiliary cell. Thus, during the reduction of $T_2 = T_b^y$, the triple $(x, a, 0)$ will be removed from the deque (corresponds to move 5.1.1.3 or 5.1.2.2) and will be forgotten. After the evaluation of T_2, the algorithm is in state \$, the triple (y, b, s) is on the deque and the auxiliary cell is empty. Now (y, b) is stored in the auxiliary cell and the input pointer

goes back to the first position in order to reduce again the tree T_1 (corresponds to move 5.2.3). Hence, in order to get $T \in T(k, 1)$, the tree T_2 must use a deque list of length k and eventually the auxiliary cell. Therefore, $T_2 \in T(k, 1)$ or $T_2 \in T(k, 0)$ if $T_1 \in T(j, 0)\ j \leqq k$. This proves (2) and (3).

LEMMA 3: *Let T be an extended binary tree with left subtree T_1 and right subtree T_2. We have for $i \geqq 2$: $T \in T(k, i)$ if and only if* (1) $T_1 \in T(k, i)$ *and* $T_2 \in T(j, 0) \cup T(k, m)$ *for some* $j \in [1 : k-1]$ *and some* $m \in [0 : i-1]$ *or* (2) $T_1 \in T(j, 0)$ *for some* $j \in [1 : k]$ *and* $T_2 \in T(k, i)$ *or* (3) $T_1 \in T(k, m)$ *for some* $m \in [1 : i-1]$ *and* $T_2 \in T(k, i-1)$.

Proof: Again, $PO(T) = PO(T_1)\, PO(T_2)\, r$. We have to consider the trees T requiring a deque list of length k and exactly i auxiliary cells during the execution of algorithm D_k with input $PO(T)$. There is exactly one triple on the deque after the reduction of T_1, namely (x, a, t), where $T_1 = T_a^x$; furthermore, the i auxiliary cells are empty.

First, we regard the case $T_1 \in T(k, i)$ that is, T_1 has used a deque list of length k and i auxiliary cells. We have $t = 1$. If a deque list of length $j \leqq k-1$ is sufficient for T_2, then $T_2 \in T(j, 0)$ for $j \leqq k-1$ and we have $T \in T(k, i)$. If T_2 requires a deque list of length k, then in any move, the triple $(x, a, 1)$ at the bottom of the deque is deleted, the tuple (x, a) is stored in the empty auxiliary cell 1 and the algorithm goes in state a (corresponds to move 5.1.1.4 or 5.1.2.3). Now $i-1$ auxiliary cells and a deque list of length k are available for T_2 that is, $T_2 \in T(k, m)$ for $m \in [0 : i-1]$. Therefore, $T_2 \in T(j, 0) \cup T(k, m)$ for $j \leqq k-1$ and $0 \leqq m \leqq i-1$, if $T_1 \in T(k, i)$. This proves (1).

Next, let $T_1 \in T(j, 0)$ with $j \leqq k$ that is, T_1 has used a deque list of length $j \leqq k$ and no auxiliary cell. We have $t = 0$. T_2 must require a deque list of length k because in the contrary the tree T does not use any auxiliary cell. Thus during the reduction of $T_2 = T_b^y$, the triple $(x, a, 0)$ will be removed from the deque (corresponds to move 5.1.1.3 or 5.1.2.2) and will be forgotten. After the reduction of T_2, the algorithm is in state \$, the triple (y, b, s) is on the deque and the auxiliary cells are empty. Now (y, b) is stored in the empty auxiliary cell 1 and the input pointer goes back to the first position in order to reduce again the tree T_1 (corresponds to move 5.2.3). Hence, in order to get $T \in T(k, i)$, the tree T_2 must use a deque list of length k and all auxiliary cells. Therefore, $T_2 \in T(k, i)$ if $T_1 \in T(j, 0)$ for $j \leqq k$. This proves (2).

Finally, let $T_1 \in T(k, m)$ with $m \in [1 : i-1]$ that is, T_1 uses a deque list of length k and m auxiliary cells. We have $t = 1$. T_2 must require a deque list of length k, because in the contrary the tree T would not use i auxiliary cells. Thus during the reduction of $T_2 = T_b^y$, the triple $(x, a, 1)$ will be removed from the deque and will be stored as the tuple (x, a) in the auxiliary cell 1

(corresponds to move 5.1.1.4 or 5.1.2.3). At this stage, $i-1$ auxiliary cells are available for T_2. After the evaluation of T_2, the algorithm is in state \$, the triple (y, b, s) is on the deque and $i-1$ auxiliary cells are empty. Now the input pointer goes back to the first position, the triple (y, b, s) is removed from the deque and the tuple (y, b) is stored in the empty auxiliary cell 2 (corresponds to move 5.2.3). Since the first input symbol x is equal to the first component of the contents of auxiliary cell 1, $(x, a, 1)$ is put onto the deque (corresponds to move 5.1.1.1) and is replaced by $(x, r, 1)$ in the next move by means of the contents in the auxiliary cell 2 (corresponds to move 5.1.1.2 if $k=1$ and to the move 5.1.1.1 followed by 5.2.1 if $k \geqq 2$). Thus T_2 must use $i-1$ auxiliary cells in order to get $T \in T(k, i)$. Therefore, we have $T_2 \in T(k, i-1)$ if $T_1 \in T(k, m)$ with $m \in [1 : i-1]$. This proves (3).

Now we are ready to consider the strategy for the reduction of a tree T induced by algorithm D_k. Let x be a node in T with the left son y and the right son z. We attach additional labels $(\alpha, \beta) \in \mathbb{N} \times \mathbb{N}_0$ to the nodes of T. If the node x is labelled by (α, β), then the reduction of the subtree with root x by algorithm D_k requires a deque list of length α and exactly β auxiliary cells. Note that $\alpha = k$ for $\beta > 0$. The preceding lemmata induce the following labelling rules:

Label of y	Label of z	x must be labelled by	Condition	Strategy	
				y	z
$(j, 0)$	$(p, 0)$	$(j, 0)$	$1 \leqq p < j \leqq k$	1	2
$(p-1, 0)$	$(j-1, 0)$	$(j, 0)$	$2 \leqq p \leqq j \leqq k$	1	2
$(k, 1)$	$(p, 0)$	$(k, 1)$	$1 \leqq p \leqq k$	1	2
$(p, 0)$	$(k, 1)$	$(k, 1)$	$1 \leqq p \leqq k$	2	1
$(p, 0)$	$(k, 0)$	$(k, 1)$	$1 \leqq p \leqq k$	2	1
(k, i)	$(p, 0)$	(k, i)	$1 \leqq p < k \wedge i \geqq 2$	1	2
(k, i)	(k, m)	(k, i)	$0 \leqq m < i \wedge i \geqq 2 \wedge k \geqq 1$	1	2
$(p, 0)$	(k, i)	(k, i)	$1 \leqq p \leqq k \wedge i \geqq 2$	2	1
(k, m)	$(k, i-1)$	(k, i)	$1 \leqq m < i \wedge i \geqq 2 \wedge k \geqq 1$	1	2

The last two columns indicate the subtree which is first reduced; in the cases of line 4,5 and 8, the right subtree is first reduced (in these cases, D_k reduces first the left subtree, forgets this reduction, reduces then the right subtree and then the left subtree again!); in the remaining cases, the left subtree is first reduced. Thus the right subtree is first reduced if and only if the left subtree has a label $(p, 0)$ with $1 \leqq p \leqq k$ and the right subtree has a label (k, i) with $i \geqq 0$. It is not hard to see that the above labelling rules can be described by the

following labelling-function $f_k : N \to \mathbb{N} \times \mathbb{N}_0$ with:

$$f_k(x) := \text{IF } x \in L \text{ THEN } (1,0)$$
$$\text{ELSE IF } f_k(y) \leqq (k,0) \wedge f_k(z) \leqq (k-1,0)$$
$$\text{THEN } (\text{MAX}(f_k^1(y), f_k^1(z)+1), 0)$$
$$\text{ELSE } (k, \text{MAX}(f_k^2(y), f_k^2(z)+1) - \delta_{0,f_k^2(y)}(1-\delta_{0,f_k^2(z)}));$$

where $f_k^1(a)$ ($f_k^2(a)$) is the first (second) component of $f_k(a)$ and $f_k(a) \leqq (\alpha, \beta)$ is an abbreviation for $f_k^1(a) \leqq \alpha$ and $f_k^2(a) \leqq \beta$; $\delta_{i,k}$ is the "Kronecker delta" notation.

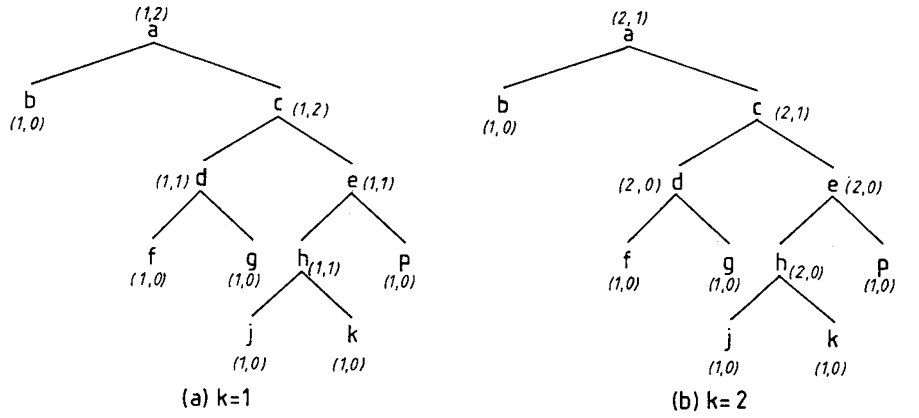
Now the conversion of the labelled tree T into a program Π for T is similar to the optimal procedure described in section 1:

1. One starts from the root node of the labelled tree. Scanning is performed from the right son if either its brother has a label $(0, 0)$ or if it has a label (k, i), $i \geqq 0$, and a left brother with label $(p, 0)$, $1 \leqq p \leqq k$. Otherwise, scanning is performed at the left son.

2. One continues scanning until one has reached a leaf or a node with sons labelled $(0, 0)$. This node is "evaluated" and is substituted by the resulting variable name A ($(0, 0)$ is substituted for its label). Then one returns to the next node above this.

If this node x has a label with a second component equal to zero, then one continues scanning; in the other case, A is substituted by the resulting variable M (the label is unchanged), the second component of the label of x is substituted by 0 and the instruction "$M \leftarrow A$" is emitted. Then one continues scanning at node x. In this context, the "evaluation of a node" is to be interpreted in a similar way as in step (B2) of the optimal procedure given in section 1.

Example 4: The tree of example 1 is labelled by f_k as follows:



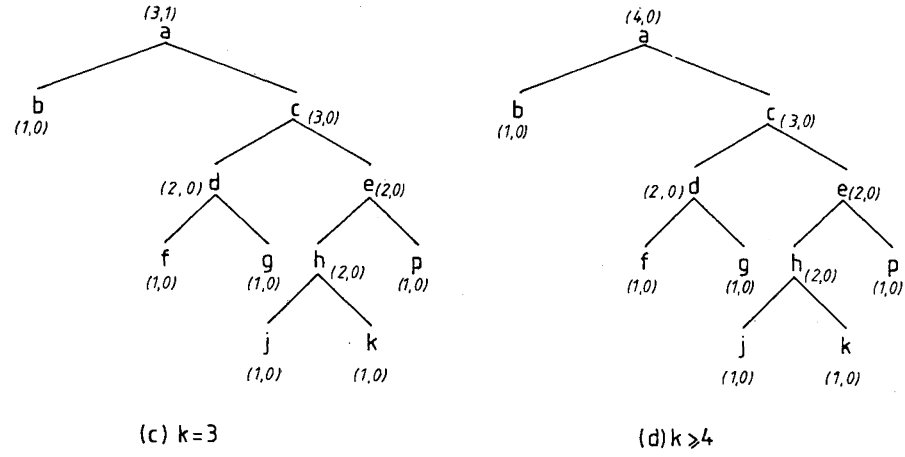


Figure 4

The above procedure leads to the following programs:

$k=1$	$k=2$	$k=3$	$k \geqq 4$
$A_1 \leftarrow g;$	$A_1 \leftarrow j;$	$A_1 \leftarrow f;$	$A_1 \leftarrow b;$
$M_1 \leftarrow A_1;$	$A_2 \leftarrow k;$	$A_2 \leftarrow g;$	$A_2 \leftarrow f;$
$A_1 \leftarrow f;$	$A_1 \leftarrow A_1 A_2 h;$	$A_1 \leftarrow A_1 A_2 d;$	$A_3 \leftarrow g;$
$A_1 \leftarrow A_1 M_1 d;$	$A_2 \leftarrow p;$	$A_2 \leftarrow j;$	$A_2 \leftarrow A_2 A_3 d;$
$M_1 \leftarrow A_1;$	$A_1 \leftarrow A_1 A_2 e;$	$A_3 \leftarrow k;$	$A_3 \leftarrow j;$
$A_1 \leftarrow k;$	$M_1 \leftarrow A_1;$	$A_2 \leftarrow A_2 A_3 h;$	$A_4 \leftarrow k;$
$M_2 \leftarrow A_1;$	$A_1 \leftarrow f;$	$A_3 \leftarrow p;$	$A_3 \leftarrow A_3 A_4 h;$
$A_1 \leftarrow j;$	$A_2 \leftarrow g;$	$A_2 \leftarrow A_2 A_3 e;$	$A_4 \leftarrow p;$
$A_1 \leftarrow A_1 M_2 h;$	$A_1 \leftarrow A_1 A_2 d;$	$A_1 \leftarrow A_1 A_2 c;$	$A_3 \leftarrow A_3 A_4 e;$
$M_2 \leftarrow A_1;$	$A_1 \leftarrow A_1 M_1 c;$	$M_1 \leftarrow A_1;$	$A_2 \leftarrow A_2 A_3 c;$
$A_1 \leftarrow p;$	$M_1 \leftarrow A_1;$	$A_1 \leftarrow b;$	$A_1 \leftarrow A_1 A_2 a;$
$A_1 \leftarrow M_2 A_1 e;$	$A_1 \leftarrow b;$	$A_1 \leftarrow A_1 M_1 a;$	
$A_1 \leftarrow M_1 A_1 c;$	$A_1 \leftarrow A_1 M_1 a;$		
$M_1 \leftarrow A_1;$			
$A_1 \leftarrow b;$			
$A_1 \leftarrow A_1 M_1 a;$			

Note that these programs simulate the reduction described by the programs computed by D_k in example 3; in the present programs, there are no "superfluous" instructions, because D_k in this version does not compute reductions which are forgotten in the course of the further computation. On the other hand, we have now to compute the additional labels of the nodes.

4. THE ENUMERATION OF (K, I)-TREES

In this section, we shall compute the cardinality $t(n, k, i)$ of the set $T(n, k, i)$ of all (k, i)-trees T with n leaves. Thus $t(n, k, i)$ is the number of the extended binary trees with n leaves which can be reduced by algorithm D_k using a deque

list of length k and exactly i auxiliary cells; there are exactly $k+i$ variables in programs computed.

Let:

$$H_n(k,i) = \sum_{1 \leqq j \leqq k} t(n,j,i)$$

and:

$$F_i(k,z) = \sum_{n \geqq 1} H_n(k,i)\, z^n,$$

be the generating function of the numbers $H_n(k,i)$. It is important to note that $H_n(k,i)$ is not the number of trees T with n leaves such that the algorithm D_k with input $PO(T)$ requires a deque list of length less than or equal to k and i auxiliary cells, because in general $T(n,j,i) \cap T(n,j',i) \neq \emptyset$ for $j \neq j'$; for $i=0$, we have in fact $T(n,j,0) \cap T(n,j',0) = \emptyset$ for $j \neq j'$.

Now let $\Delta_i(k,z)$ be defined by:

$$\Delta_i(k,z) = F_i(k,z) - F_i(k-1,z).$$

Thus:

$$\Delta_i(k,z) = \sum_{n \geqq 1} t(n,k,i)\, z^n.$$

We prove now the following:

LEMMA 4: *The function $F_0(k,z)$ satisfies the recurrence:*

$$F_0(1,z) = z,$$
$$F_0(k,z) = z/(1 - F_0(k-1,z)), \qquad k \geqq 1.$$

Proof: Obviously, $F_0(1,z) = z$. By lemma 1 we obtain for $j \geqq 2$:

$$\Delta_0(k,z) = \Delta_0(k,z)\, F_0(k-1,z) + \Delta_0(k-1,z)\, F_0(k-1,z)$$

which is equivalent to:

$$F_0(k,z) - F_0(k,z)\, F_0(k-1,z)$$
$$= F_0(k-1,z) - F_0(k-1,z)\, F_0(k-2,z).$$

Hence the sequence $F_0(k,z) - F_0(k,z)\, F_0(k-1,z)$ is a constant sequence. Since $F_0(1,z) = z$, this relation is equivalent to our proposition. This completes the proof.

An inspection of formula (2) in [1] shows, that $F_0(k, z)$ satisfies the same recurrence as the generating function $A_k(z)$ of the number of planted plane trees with n nodes and height less than or equal to k. This result is not very surprising, because $A_k(z)$ is also the generating function of the number of binary trees with n leaves using for postorder traversal a maximum size of the stack less than or equal to k ([4, 5]) and the algorithm D_k is identical to algorithm S given in section 1, if S emits a program with $\leqq k$ variables.

The following lemma 5 is a direct implication of lemma 4 and formula (6) in [1]:

LEMMA 5: *The generating function $F_0(k, z)$ is given by:*

$$F_0(k, z) = 2z \frac{(1+\sqrt{1-4z})^k - (1-\sqrt{1-4z})^k}{(1+\sqrt{1-4z})^{k+1} - (1-\sqrt{1-4z})^{k+1}}, \qquad k \geqq 1.$$

Before we compute the number of all (k, i)-trees with n leaves, we shall present two properties of the function $F_0(k, z)$. The correctness of these equations can be easily checked by means of lemma 5.

LEMMA 6: *The generating function $F_0(k, z)$ fulfills the following relations:*

(a) $$F_0(2k+1, z) - F_0(k, z) = \frac{F_0^2(k, z) - F_0(k, z)\, F_0(k-1, z)}{1 - 2F_0(k, z)}, \qquad k \geqq 1.$$

(b) $$F_0((i+1)(k+1)-1, z) - F_0(i(k+1)-1, z) = \frac{\left\{ \begin{array}{r} [F_0(i(k+1)-1, z) - F_0((i-1)(k+1)-1, z)] \\ \times [F_0(i(k+1)-1, z) - F_0(k, z)] \end{array} \right\}}{1 - F_0(k, z) - F_0(i(k+1)-1, z)}$$

for $k \geqq 1$ and $i \geqq 2$.

We prove now the following:

THEOREM 1: *The number $t(n, k, i)$ of all (k, i)-trees with n leaves is given by:*

$$t(n, k, i) = H_n((i+1)(k+1)-1, 0) - H_n(i(k+1)-1, 0)$$

for $i \geqq 1$ and $k \geqq 1$.

Proof: We have to prove:

$$\Delta_i(k, z) = F_0((i+1)(k+1)-1, z) - F_0(i(k+1)-1, z).$$

(a) First, let $i = 1$. We obtain by lemma 2:

$$\Delta_1(k, z) = \Delta_1(k, z)\, F_0(k, z) + F_0(k, z)\, \Delta_1(k, z) + F_0(k, z)\, \Delta_0(k, z)$$

which is equivalent to:

$$\Delta_1(k,z)=\frac{F_0^2(k,z)-F_0(k,z)\,F_0(k-1,z)}{1-2F_0(k,z)}.$$

Using lemma 6(*a*) we get further:

$$\Delta_1(k,z)=F_0(2k+1,z)-F_0(k,z)$$

(*b*) Now let $i\geqq 2$. We obtain by lemma 3:

$$\Delta_i(k,z)=\Delta_i(k,z)[F_0(k,z)+\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)]$$
$$+F_0(k,z)\,\Delta_i(k,z)+\Delta_{i-1}(k,z)\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)$$

which is equivalent to:

$$\Delta_i(k,z)=\frac{\Delta_{i-1}(k,z)\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)}{1-2F_0(k,z)-\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)}.$$

We prove now by induction on i:

$$\Delta_i(k,z)=F_0(i(k+1)+k,z)-F_0(i(k+1)-1,z).$$

$i=2$: We get:

$$\Delta_2(k,z)=\frac{\Delta_1^2(k,z)}{1-2F_0(k,z)-\Delta_1(k,z)}$$
$$=\frac{[F_0(2k+1,z)-F_0(k,z)]^2}{1-F_0(k,z)-F_0(2k+1,z)}\quad\text{[by part (}a\text{)]}$$
$$=F_0(3k+2,z)-F_0(2k+1,z)\quad\text{[by lemma 6(}b\text{) with } i=2\text{]}.$$

$i-1\curvearrowright i$: Let the above proposition be true for $\leqq i-1$. We obtain by part (*a*) and the induction hypothesis:

$$\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)=F_0(i(k+1)-1,z)-F_0(k,z)$$

and therefore:

$$\Delta_i(k,z)=\frac{\Delta_{i-1}(k,z)\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)}{1-2F_0(k,z)-\sum_{1\leqq r\leqq i-1}\Delta_r(k,z)}$$

$$=\frac{\left\{\begin{array}{r}[F_0(i(k+1)-1,z)-F_0((i-1)(k+1)-1,z)]\\ \times[F_0(i(k+1)-1,z)-F_0(k,z)]\end{array}\right\}}{1-F_0(k,z)-F_0(i(k+1)-1,z)}$$

$$=F_0((i+1)(k+1)-1,z)-F_0(i(k+1)-1,z)$$

by an application of Lemma 6(*b*). This completes the proof.

Using the definition of $H_n(k,i)$ and theorem 1, we obtain immediately the following:

COROLLARY 1: *The number of extended binary trees with n leaves which can be reduced by algorithm D_k using a deque list of length k and exactly $i\geqq 1$ auxiliary cells is equal to the number of all extended binary trees with n leaves which can be reduced by the algorithms D_m, $m\in[(k+1)i:(k+1)(i+1)-1]$ without any auxiliary cell; this number is also the cardinality of the set of all trees with n leaves which can be reduced by algorithm S using a stack size of length $m\in[(k+1)i:(k+1)(i+1)-1]$.*

It is not hard to see, that this fact is only a connection between the numbers of the trees. In general, the corresponding sets are different. For $i=0$ we have by definition:

$$t(n,k,0)=H_n(k,0)-H_n(k-1,0), \tag{1}$$

where $H_n(k,0)$ is the number of extended binary trees with n leaves which can be reduced by D_k without any auxiliary cell. Since $H_n(k,0)$ is also equal to the number of all extended binary trees with n leaves which can be reduced by algorithm S with a stack size less than or equal to k, we can use lemma 1 in [4] and obtain an explicit expression for these numbers. We get:

$$H_n(k,0)=t(n)$$

$$-\sum_{j\geqq 1}\left[\binom{2n-2}{n-j(k+1)}-2\binom{2n-2}{n-1-j(k+1)}+\binom{2n-2}{n-2-j(k+1)}\right], \tag{2}$$

where:

$$t(n)=\frac{1}{n}\binom{2n-2}{n-1}, \tag{3}$$

is the number of all extended binary trees with n leaves. Thus theorem 1 leads directly to an explicit expression for the number of all (k, i)-trees with n leaves. We obtain:

COROLLARY 2: *Let* $\psi_a(n, k, i)$, $a \in \mathbb{N}_0$, $n, k, i \in \mathbb{N}$, *be defined by:*

$$\psi_a(n, k, i) = \sum_{\lambda \geqq 1} \left[\binom{2n-2}{n-a-\lambda\, i(k+1)} - \binom{2n-2}{n-a-\lambda\,(i+1)(k+1)} \right].$$

An explicit expression for the number of all (k, i)*-trees with* n *leaves is given by:*

$$t(n, k, i) = \psi_0(n, k, i) - 2\psi_1(n, k, i) + \psi_2(n, k, i).$$

5. OPTIMAL CHOICE OF THE PARAMETERS k AND i

Generally, the algorithm D_k with i auxiliary cells does not reduce all extended binary trees with n leaves. In this section we shall derive a condition for n, k, i such that D_k with i auxiliary cells reduces all binary trees with n leaves.

The algorithm D_k with i auxiliary cells cannot only reduce trees $T \in T(k, i)$ but also trees which use a deque list of length $\leqq k$ or a deque list of length k and less than or equal to i auxiliary cells. Thus T can be reduced by D_k if $T \in \bigcup_{1 \leqq j \leqq k} T(j, 0) \cup \bigcup_{1 \leqq r \leqq i} T(k, r)$.

Now let RED (n, k, i) be the number of all trees T with n leaves which can be reduced by algorithm D_k with i auxiliary cells. Since for each k, the sets $T(n, 1, 0)$, $T(n, 2, 0), \ldots, T(n, k, 0)$, $T(n, k, 1)$, $T(n, k, 2), \ldots, T(n, k, i)$ are mutually disjoint, we get:

$$\mathrm{RED}(n, k, i) = \sum_{1 \leqq j \leqq k} t(n, j, 0) + \sum_{1 \leqq \lambda \leqq i} t(n, k, \lambda)$$

or by the definition of $H_n(k, i)$ and theorem 1:

$$\begin{aligned}\mathrm{RED}(n, k, i) &= H_n(k, 0) \\ &\quad + \sum_{1 \leqq \lambda \leqq i} [H_n((\lambda+1)(k+1)-1, 0) - H_n(\lambda(k+1)-1, 0)] \\ &= H_n((i+1)(k+1)-1, 0) \\ &= \sum_{1 \leqq j \leqq (i+1)(k+1)-1} t(n, j, 0).\end{aligned}$$

Thus we have the following:

THEOREM 2: *The number of extended binary trees with n leaves which can be reduced by algorithm D_k with less than or equal to i auxiliary cells is equal to the number of all extended binary trees with n leaves which can be reduced by the algorithms D_m, $m \in [1 : (k+1)(i+1)-1]$ without any auxiliary cell; this number is the cardinality of the set of all trees with n leaves which can be reduced by algorithm S using a maximum stack size of length $(k+1)(i+1)-1$.*

Using (2) we obtain an explicit expression for the numbers $\mathrm{RED}(n, k, i)$.

COROLLARY 3: *The number of all extended binary trees with n leaves which can be reduced by algorithm D_k with less than or equal to i auxiliary cells is given by:*

$$\mathrm{RED}(n, k, i) = t(n) - [\xi_0(n, k, i) - 2\xi_1(n, k, i) + \xi_2(n, k, i)],$$

where:

$$\xi_a(n, k, i) = \sum_{\lambda \geq 1} \binom{2n-2}{n-a-\lambda(k+1)(i+1)}$$

and $t(n)$ is given by the Catalan number (3).

If $\lambda \geq n$, $H_n(\lambda, 0)$ is the number of all extended binary trees with n leaves. Since $\mathrm{RED}(n, k, i) = H_n((i+1)(k+1)-1, 0)$, we obtain further the following:

THEOREM 3: *For each $k \geq 1$, algorithm D_k reduces all extended binary trees with n leaves if and only if D_k has at least $\lfloor n/(k+1) \rfloor$ auxiliary cells. The computed programs have at most $k + \lfloor n/(k+1) \rfloor$ variables.*

Since the function $f(k) = k + \lfloor n/(k+1) \rfloor$ has surely a minimum at $k = \lfloor \sqrt{n} \rfloor$, we get the following:

COROLLARY 4: *Among all algorithms D_k, $D_{\lfloor \sqrt{n} \rfloor}$ with $\lfloor n/\lfloor \sqrt{n} \rfloor + 1 \rfloor$ auxiliary cells reduces all extended binary trees with n leaves and requires a minimum number of deque list and auxiliary cells. The computed programs have at most $\lfloor \sqrt{n} \rfloor + \lfloor n/(\lfloor \sqrt{n} \rfloor + 1) \rfloor$ variables.*

6. THE AVERAGE CASE

In this section we shall derive some results concerning the average number of deque list and auxiliary cells required by algorithm D_k during the reduction of an extended binary tree with n leaves. Since $\mathrm{RED}(n, k, i) = H_n((i+1)(k+1)-1, 0)$, theorem 1 in [4] gives us information on the distribution of all trees with n leaves which can be reduced by D_k with i auxiliary cells. We obtain immediately the following:

THEOREM 4: *Assuming that all extended binary trees with n leaves are equally likely, the probability $V_n(k, i)$, that a tree can be reduced by algorithm D_k with i auxiliary cells is asymptotically given by:*

$$V_n(k,i) = 1 - \sum_{\lambda \geq 1} \left[4\lambda^2 \frac{[(k+1)(i+1)]^2}{n} - 2 \right]$$

$$\times \exp(-\lambda^2 [(k+1)(i+1)]^2/n) + O(\log^*(n)/\sqrt{n}),$$

where $O(n^*)$ is to be interpreted as $O(n^k)$ for some $k \in \mathbb{N}$

Using the numerical results given in [4], we see for instance that in the asymptotic case 99.58% of all trees can be reduced by algorithm D_k with i auxiliary cells provided that $(i+1)(k+1) = 3\sqrt{n}$. Thus 99.58% of all trees can be reduced by D_1 with approximately $1.5\sqrt{n} - 1$ auxiliary cells or by D_2 with approximately $\sqrt{n} - 1$ auxiliary cells etc.

Henceforth, we assume that the algorithm D_k has always $\lfloor n/(k+1) \rfloor$ auxiliary cells that is, D_k can reduce all extended binary trees with n leaves. We now turn to the average number of auxiliary cells used by D_k during the reduction of a tree with n leaves.

Considering all extended binary trees with n leaves equally likely, the quotient $p_k(n, i) = t(n, k, i)/t(n)$ is the probability that the reduction of a n-node tree by D_k uses exactly i auxiliary cells. Therefore, the average number of auxiliary cells required by D_k during the reduction of a tree with n leaves is given by the expected value $h_k(n)$, where:

$$h_k(n) = \sum_{1 \leq i \leq \lfloor n/(k+1) \rfloor} i p_k(n, i).$$

Using the definition of $p_k(n, i)$ and theorem 1, this expression can be easily transformed into:

$$h_k(n) = \lfloor n/(k+1) \rfloor - t(n)^{-1} \sum_{1 \leq i \leq \lfloor n/(k+1) \rfloor} H_n(i(k+1) - 1, 0). \tag{4}$$

Since $\lfloor n/(k+1) \rfloor (k+1) + k > n - 1$, we have:

$$H_n(\lfloor n/(k+1) \rfloor (k+1) + k, 0) = t(n), \tag{5}$$

where $t(n)$ is the number of all extended binary trees with n leaves given

by (3). Using the identity:

$$t(n)^{-1}\left[\binom{2n-2}{n-t}-2\binom{2n-2}{n-1-t}+\binom{2n-2}{n-2-t}\right]$$
$$=\binom{2n}{n}^{-1}\binom{2n}{n-t}\left[4\frac{t^2}{n}-2\right], \tag{6}$$

we obtain with (4) and (2):

$$h_k(n)=\binom{2n}{n}^{-1}\sum_{i\geqq 1}\sum_{j\geqq 1}\left[4\frac{[ji(k+1)]^2}{n}-2\right]\binom{2n}{n-ji(k+1)}. \tag{7}$$

We prove now the following:

THEOREM 5: *Assuming that all extended binary trees with n leaves are equally likely, the average number $h_k(n)$ of auxiliary cells used by algorithm D_k during the reduction of an extended binary tree with n leaves is asymptotically given by:*

$$h_k(n)=\begin{cases} O(\exp(-n^{2\beta})) & \text{if } k\geqq n^{0.5+\delta},\ \delta>0,\\ \sum_{R\geqq 1} d(R)[4R^2c^2-2]\exp(-R^2c^2)+O(n^{-0.5+\varepsilon}) & \\ & \text{if } k=c\sqrt{n},\ c>0,\\ \frac{\sqrt{\pi n}}{k+1}-\frac{1}{2}+O((k+1)^{-1}n^{-0.5+\varepsilon}) & \text{if } k\leqq n^{0.5-\delta},\ 0<\delta\leqq\frac{1}{2}\end{cases}$$

for all fixed $\varepsilon>0$, $\beta>0$ and $\delta>0$. The arithmetical function $d(n)$ is the number of all positive divisors of the natural number n.

Proof: The following approximation is given in [1]:

$$\binom{2n}{n}^{-1}\binom{2n}{n-p}=\begin{cases}\exp(-p^2/n)f(n,p) & \text{if } p<n^{0.5+\varepsilon}, \quad (8a)\\ O(\exp(-n^{2\varepsilon})) & \text{if } p\geqq n^{0.5+\varepsilon}, \quad (8b)\end{cases}$$

for all fixed $\varepsilon>0$. Here, $f(n,p)$ is defined by $f(n,p)=1+O(n^{-1+\varepsilon})$.

An inspection of (7) shows that an equivalent expression for $h_k(n)$ is given by:

$$h_k(n)=\binom{2n}{n}^{-1}\sum_{1\leqq R\leqq\lfloor n/(k+1)\rfloor} d(R)$$
$$\times\left[4\frac{[R(k+1)]^2}{n}-2\right]\binom{2n}{n-R(k+1)}. \tag{9}$$

Now let $k \geqq n^{0.5+\delta}$ with $\delta > 0$. We consider the sum given in (9). Choosing $\varepsilon > 0$ with $\varepsilon < \delta$, we obtain immediately $R(k+1) > n^{0.5+\varepsilon}$ for sufficient large n and therefore by the approximation (8 *b*):

$$h_k(n) = O(\exp(-n^{2\beta}))$$

for some $\beta > 0$.

Next, let $k = n^{0.5-\delta}$ with $0 \leqq \delta \leqq 1/2$. Using the approximation (8 *b*), the terms for $R \geqq n^{0.5+\varepsilon}/(k+1)$ in (9) are exponentially small and therefore negligible, being $O(n^{-m})$ for all $m > 0$. An application of (8 *a*) to the remaining terms leads to:

$$h_k(n) = \left[\sum_{R \geqq 1} d(R) \left[4 \frac{[R(k+1)]^2}{n} - 2 \right] \right.$$

$$\left. \times \exp(-[R(k+1)]^2/n) \right] [1 + O(n^{-1+\varepsilon})].$$

If $\delta = 0$ and $k = c\sqrt{n}$ with $c > 0$, we obtain:

$$h_k(n) = \sum_{R \geqq 1} d(R)[4R^2 c^2 - 2] \exp(-R^2 c^2) + O(n^{-0.5+\varepsilon}).$$

If $0 < \delta \leqq 1/2$ and $k \leqq n^{0.5-\delta}$, we have $n/(k+1)^2 \to \infty$ for $n \to \infty$. Therefore, we can make a similar computation as in [1]. Defining:

$$\Phi_a(n, k) = \sum_{R \geqq 1} d(R)\, R^a \exp(-R^2 (k+1)^2/n),$$

we can express $h_k(n)$ in terms of $\Phi_a(n, k)$ and obtain:

$$h_k(n) = \left[4 \frac{(k+1)^2}{n} \Phi_2(n, k) - 2\Phi_0(n, k) \right] [1 + O(n^{-1+\varepsilon})].$$

A comparison of $\Phi_a(n, k)$ with formula (26) in [1] shows that $\Phi_a(n, k) = g_a(n/(k+1)^2)$, where $g_a(n)$ is given by (29) in [1]. Using the explicit expressions for $g_a(n)$ given by (32) in [1], we obtain:

$$h_k(n) = \left[\frac{\sqrt{\pi n}}{k+1} - \frac{1}{2} + O(n^{-m}) \right] [1 + O(n^{-1+\varepsilon})]$$

for all $m > 0$. This completes the proof of theorem 5.

We now turn to the average number of auxiliary cells and deque list cells required by algorithm D_k during the reduction of an extended binary tree with n leaves. There are $t(n, j, 0)$, $1 \leqq j \leqq k$, trees using exactly j deque list cells and no auxiliary cells; there are $t(n, k, i)$, $1 \leqq i \leqq \lfloor n/(k+1) \rfloor$, trees using exactly k deque list cells and exactly i auxiliary cells. Thus considering all extended binary trees with n leaves equally likely, the average number of auxiliary cells and deque list cells required by D_k during the reduction of a tree with n leaves is given by the expected value:

$$e_k(n) = t(n)^{-1} \left[\sum_{1 \leqq j \leqq k} j t(n, j, 0) + \sum_{1 \leqq i \leqq \lfloor n/(k+1) \rfloor} (k+i)\, t(n, k, i) \right].$$

Using the above definition of $h_k(n)$ and theorem 1, this expression can be easily transformed into:

$$e_k(n) = h_k(n) + t(n)^{-1} \sum_{1 \leqq j \leqq k} j t(n, j, 0)$$

$$+ k t(n)^{-1} [H_n(\lfloor n/(k+1) \rfloor (k+1) + k, 0) - H_n(k, 0)].$$

Now an application of (1) and (5) leads directly to:

$$e_k(n) = h_k(n) + k - t(n)^{-1} \sum_{1 \leqq j \leqq k-1} H_n(j, 0). \tag{10}$$

Thus the problem of obtaining an asymptotic equivalent for $e_k(n)$ reduces to the analogous problem for:

$$q_k(n) = k - t(n)^{-1} \sum_{1 \leqq j \leqq k-1} H_n(j, 0).$$

Using (2) and (6), $g_k(n)$ can be easily transformed into:

$$q_k(n) = \sum_{1 - j - k} \sum_{\lambda \geqq 1} \binom{2n}{n}^{-1} \left[4 \frac{[j\lambda]^2}{n} - 2 \right] \binom{2n}{n - j\lambda}. \tag{11}$$

We now prove the following:

LEMMA 7: *The numbers* $q_k(n)$ *given by* (11) *have the following asymptotic behaviour:*

$$q_k(n) = \begin{cases} \sqrt{\pi n} - \frac{1}{2} + O(n^{-1+\varepsilon}) & \text{if } k \geqq n^{0.5+\delta},\ \delta > 0 \\ b\sqrt{n} + O(n^{-0.5+\varepsilon}) & \text{if } k = c\sqrt{n},\ c > 0, \\ k + O(kn^{-1+\varepsilon}), & \text{if } k \leqq n^{0.5-\delta},\ 0 < \delta \leqq \frac{1}{2}, \end{cases}$$

for all fixed $\varepsilon > 0$ *and* $\delta > 0$. *Here,* $c \sum_{R \geq 1} [4R^2 c^2 - 2] \exp(-R^2 c^2) < b \leqq c$.

Proof: An inspection of (11) shows that an equivalent expression for $q_k(n)$ is given by:

$$q_k(n) = \binom{2n}{n}^{-1} \sum_{R \geqq 1} d_k(R) \left[4\frac{R^2}{n} - 2 \right] \binom{2n}{n-R}, \tag{12}$$

where $d_k(R)$ is the number of the positive divisors less than or equal to k of the natural number R.

First, let $k \geqq n^{0.5+\delta}$ with $\delta > 0$. We consider the sum given in (12). Introducing the arithmetical function $d(R)$ of the number of all positive divisors of the natural number R, we find $d_k(R) = d(R)$ for $k \leqq R$. Hence:

$$q_k(n) = q(n) + q_k^{(1)}(n),$$

where:

$$q(n) = \binom{2n}{n}^{-1} \sum_{R \geqq 1} d(R) \left[4\frac{R^2}{n} - 2 \right] \binom{2n}{n-R}$$

and:

$$q_k^{(1)}(n) = \binom{2n}{n}^{-1} \sum_{R \geqq k+1} [d_k(R) - d(R)] \left[4\frac{R^2}{n} - 2 \right] \binom{2n}{n-R}.$$

Using the approximation (8 *a*), $q_k^{(1)}(n)$ is exponentially small and therefore negligible, being $O(n^{-m})$ for all $m > 0$. A comparison of $q(n)$ with (9) shows that $q(n) = h_0(n)$. Thus using theorem 5 (case $k \leqq n^{0.5-\delta}$, $0 < \delta \leqq 1/2$), we obtain:

$$q_k(n) = \sqrt{\pi n} - \frac{1}{2} + O(n^{-0.5+\varepsilon})$$

for all fixed $\varepsilon > 0$.

Next, let $k = n^{0.5-\delta}$ with $0 \leqq \delta \leqq 1/2$. Since $d_k(R)$ has the property:

$$d_k(R) = \begin{cases} d_{k-1}(R)+1 & \text{if } k \mid R, \\ d_{k-1}(R) & \text{otherwise,} \end{cases}$$

we obtain with (12) for $2 \leqq k \leqq n^{0.5-\delta}$:

$$q_k(n) = q_{k-1}(n) + \eta_k(n), \tag{13}$$

where:

$$\eta_k(n) = \binom{2n}{n}^{-1} \sum_{R \geqq 1} \left[4\frac{[R\,k]^2}{n} - 2 \right] \binom{2n}{n - R\,k} \tag{14}$$

An inspection of the approximation (8 *a*) shows that the terms for $R \geqq n^{0.5+\varepsilon}/k$ in $\eta_k(n)$ are exponentially small and therefore negligible, being $O(n^{-m})$ for all $m > 0$. An application of (8 *b*) to the remaining terms leads by (14) to:

$$\eta_k(n) = \left[\sum_{R \geqq 1} \left[4\frac{[R\,k]^2}{n} - 2 \right] \exp(-[R\,k]^2/n) \right] [1 + O(n^{-1+\varepsilon})]. \tag{15}$$

Now we regard the function:

$$\theta(z) = \sum_{j \geqq 1} \exp(-j^2\,\pi\,z) \tag{16}$$

which fulfills the "Theta-relation" [4]:

$$\theta(z) = z^{-1/2}\,\theta(z^{-1}) + \frac{1}{2} z^{-1/2} - \frac{1}{2}. \tag{17}$$

Using this equation, an elementary computation leads to:

$$-2\,\theta(z) - 4\,z\,\theta'(z) = 1 + 4\,z^{-1.5}\,\theta'(z^{-1}). \tag{18}$$

On the other hand, we obtain by (15):

$$\eta_k(n) = \left[-2\,\theta\left(\frac{k^2}{\pi\,n}\right) - 4\frac{k^2}{\pi\,n}\,\theta'\left(\frac{k^2}{\pi\,n}\right) \right] [1 + O(n^{-1+\varepsilon})]$$

and therefore,

$$\eta_k(n) = \left[1 + 4\left(\frac{\pi\,n}{k^2}\right)^{1.5} \theta'\left(\frac{\pi\,n}{k^2}\right) \right] [1 + O(n^{-1+\varepsilon})]$$

$$= [1 - \psi_k(n)]\,[1 + O(n^{-1+\varepsilon})], \tag{19}$$

where:

$$\psi_k(n) = 4\pi^2 n\sqrt{\pi n}/k^3 \sum_{R \geqq 1} R^2 \exp(-R^2 \pi^2 n/k^2). \tag{20}$$

It is not hard to see that the sum $\psi_k(n)$ is exponentially small for $k \leqq n^{0.5-\delta}$, $0 < \delta \leqq 1/2$, being $O(n^{-m})$ for all $m > 0$. In this case, we get $\eta_k(n) = 1 + O(n^{-1+\varepsilon})$ and therefore with (3) by partial summation:

$$q_k(n) = q_1(n) + (k-1) + O(kn^{-1+\varepsilon}).$$

Since $d_1(R) = 1$, we obtain further by (12) and (6):

$$q_1(n) = \binom{2n}{n}^{-1} \sum_{R \geqq 1} \left[4\frac{R^2}{n} - 2\right] \binom{2n}{n-R} = 1. \tag{21}$$

Hence

$$q_k(n) = k + O(kn^{-1+\varepsilon}) \qquad \text{for} \quad k \leqq n^{0.5-\delta}, \quad 0 < \delta \leqq \frac{1}{2}.$$

It remains the case $k = c\sqrt{n}$, $c > 0$. We can only give an approximation of $q_k(n)$ for this choice of k.

First, we shall prove that the function:

$$f(x) = 4\pi^2 \sqrt{\pi}\, x^{-3} \sum_{R \geqq 1} R^2 \exp(-R^2 \pi^2/x^2), \tag{22}$$

is strictly increasing for all $x \geqq 0$ with $0 \leqq f(x) < 1$. Computing the derivative of $f(x)$, we obtain:

$$f'(x) = 4\pi^2 \sqrt{\pi}\, x^{-6} \sum_{R \geqq 1} R^2 [2R^2 \pi^2 - 3x^2] \exp(-R^2 \pi^2/x^2)$$

Hence $f'(x) > 0$ for $0 < x < \sqrt{2}\pi/\sqrt{3} = 2.565...$ On the other hand, we have with (16) and (18):

$$\begin{aligned} f(x) &= -4\pi \sqrt{\pi}/x^3\, \theta'(\pi/x^2) \\ &= 1 + 2\theta(x^2/\pi) + 4x^2/\pi\theta'(x^2/\pi) = 1 - \sum_{R \geqq 1} [4R^2 x^2 - 2] \exp(-R^2 x^2). \end{aligned} \tag{23}$$

We now obtain:

$$f'(x) = -\sum_{R \geqq 1} 4R^2 x [3 - 2R^2 x^2] \exp(-R^2 x^2).$$

Thus $f'(x)>0$ for $x>\sqrt{3/2}=1.253...$ Altogether we have proved that $f(x)$ is strictly increasing for $x>0$. Furthermore, it is not hard to show that the series appearing in (22) and (23) are uniformly convergent. Hence by (22):

$$f(0)=\lim_{x\to 0} f(x)=4\pi^2\sqrt{\pi}\sum_{R\geqq 1} R^2 \lim_{x\to 0} x^{-3}\exp(-R^2\pi^2/x^2)=0$$

and by (23):

$$\lim_{x\to\infty} f(x)=1-\sum_{R\geqq 1}\lim_{x\to\infty}[4R^2x^2-2]\exp(-R^2x^2)=1.$$

Since $\psi_k(n)=f(k/\sqrt{n})$, we obtain with (13), (19) and (21) by partial summation:

$$q_{c\sqrt{n}}(n)=\left[c\sqrt{n}-\sum_{2\leqq\lambda\leqq c\sqrt{n}} f(\lambda/\sqrt{n})\right][1+O(n^{-1+\varepsilon})].$$

Since $k=c\sqrt{n}\geqq 1$, $0\leqq f(x)<1$ for $x\geqq 0$ and $f(x)$ is stricty increasing for all $x\geqq 0$ we obtain further:

$$c\sqrt{n}-\sum_{2\leqq\lambda\leqq c\sqrt{n}} f(\lambda/\sqrt{n})\leqq c\sqrt{n}-[c\sqrt{n}-1]f(2/\sqrt{n})\leqq c\sqrt{n}$$

and:

$$c\sqrt{n}-\sum_{2\leqq\lambda\leqq c\sqrt{n}} f(\lambda/\sqrt{n})>c\sqrt{n}-\sum_{1\leqq\lambda\leqq c\sqrt{n}} f(\lambda/\sqrt{n})\geqq c\sqrt{n}[1-f(c)].$$

Thus:

$$q_{c\sqrt{n}}(n)=b\sqrt{n}+O(n^{-0.5+\varepsilon})$$

with $c[1-f(c)]<b\leqq c$. This completes the proof of our lemma 7.

Using the results given in theorem 5 and lemma 7 we obtain immediately by (10) the following:

THEOREM 6: *Assuming that all extended binary trees with n leaves are equally likely, the average number $e_k(n)$ of auxiliary cells and deque list cells required by algorithm D_k during the reduction of a tree with n leaves is asymptotically*

given by:

$$e_k(n)=\begin{cases}\sqrt{\pi n}-\dfrac{1}{2}+O(n^{-1+\varepsilon}) & \text{if } k\geqq n^{0.5+\delta},\ \delta>0,\\ b\sqrt{n}+\displaystyle\sum_{R\geqq 1} d(R)[4R^2c^2-2]\exp(-R^2c^2)+O(n^{-0.5+\varepsilon}) & \\ \text{if } k=c\sqrt{n},\ c>0, & \\ k+\dfrac{\sqrt{\pi n}}{k+1}-\dfrac{1}{2}+O(n^{-0.5+\varepsilon}) & \text{if } k\leqq n^{0.5-\delta},\ 0<\delta\leqq\dfrac{1}{2}\end{cases}$$

for all fixed $\varepsilon>0$ *and* $\delta>0$. *Here,* $c\displaystyle\sum_{R\geqq 1}[4R^2c^2-2]\exp(-R^2c^2)<b\leqq c$.

Since the function $f(k)=k+(k+1)^{-1}\sqrt{\pi n}-0.5$ has a minimum at $k=(\pi n)^{1/4}-1$, we get the following:

COROLLARY 6: *Assume that all extended binary trees with n leaves are equally likely. Among all algorithms* $D_k, D_{k'}$ *with* $k'=(\pi n)^{1/4}-1$ *requires a minimum number of deque list and auxiliary cells, on the average. The programs computed by* $D_{k'}$ *have* $2(\pi n)^{1/4}-1.5$ *variables, on the average.*

Final remarks

We have presented a class of algorithms D_k which reduce a given extended binary tree. Each algorithm D_k uses an input-restricted deque of length k and an auxiliary store. D_k is a possible generalization of a customary method for the reduction of a tree by means of a stack. Although we have given a detailed analysis of the space complexity of D_k in the worst and average case, several questions are still waiting to be resolved.

One such problem is a detailed analysis of the time complexity of D_k in the worst and average case. Considering the algorithm D_k, it is not hard to see that in the worst case the input pointer is reset about n/k times to the first position. Hence the moves on the input tape give a contribution $O(n^2/k)$ to the time complexity of D_k in the worst case; but here, we have not considered the time which is necessary to insert or to delete a tuple in the auxiliary store.

Another open problem is implied by the results given in corollary 1 and theorem 2. The observation on the equality of the sizes of different tree classes challenges to look for an one-to-one correspondence, but the author does not know such an explicit transformation.

Finally, it remains the open problem to determine the exact asymptotic behaviour of the numbers $q_{c\sqrt{n}}(n)$ given in lemma 7; we have only proved that $q_{c\sqrt{n}}(n) = O(\sqrt{n})$.

ACKNOWLEDGEMENTS

The author likes to thank the referee for his delailed criticisms and the suggestions for improving the clarity of the presentation.

REFERENCES

1. N. G. DE BRUIJN, D. E. KNUTH and S. O. RICE, *The Average Height of Planted Plane Trees*, in Graph theory and computing, R. C. READ, Ed., New York-London: Academic Press, 1972, pp. 15-22.
2. P. FLAJOLET and A. ODLYZKO, *The Average Height of Binary Trees and Other Simple Trees*, J.C.S.S., Vol. 25 (2), 1982, pp. 171-213.
3. P. FLAJOLET, J. C. RAOULT and J. VUILLEMIN, *The Number of Registers Required for Evaluating Arithmetic Expressions*, Theoret. Comp. Sc., Vol. 9, 1979, pp. 99-125.
4. R. KEMP, *On the Average Stack Size of Regularly Distributed Binary Trees*, in Proc. of the 6-th ICALP Conf., H. MAURER, Ed., Berlin-Heidelberg-New York, Springer, 1979, pp. 340-355.
5. R. KEMP, *A Note on the Stack Size of Regularly Distributed Binary Trees*, BIT, Vol. 20, 1980, pp. 157-162.
6. R. KEMP, *The Average Height of R-Tuply Rooted Planted Plane Trees*, Computing, Vol. 25, 1980, pp. 209-232.
7. R. KEMP, *The Average Number of Registers needed to Evaluate a Binary Tree Optimally*, Acta Informatica, Vol. 11, 1979, pp. 363-372.
8. D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, Reading: Addison-Wesley, 1973.
9. I. NAKATA, *On Compiling Algorithms for Arithmetic Expressions*, Comm. A.C.M., Vol. 10, 1967, pp. 492-494.
10. R. SETHI and J. D. ULLMAN, *The Generation of Optimal Code for Arithmetic Expressions*, J.A.C.M., Vol. 17, 1970, pp. 715-728.