Acta Informatica 17, 157–184 (1982)

A New Data Structure for Representing Sorted Lists*

Scott Huddleston[1] and Kurt Mehlhorn[2]

[1] Scott Huddleston, Information and Computer Science, University of California, Irvine, Irvine, California 92717, USA

[2] Kurt Mehlhorn, FB 10 – Informatik, Universität des Saarlandes, 6600 Saarbrücken, West Germany

Summary. In this paper we explore the use of weak B-trees to represent sorted lists. In weak B-trees each node has at least a and at most b sons where $2a \leqq b$. We analyse the worst case cost of sequences of insertions and deletions in weak B-trees. This leads to a new data structure (level-linked weak B-trees) for representing sorted lists when the access pattern exhibits a (time-varying) locality of reference. Our structure is substantially simpler than the one proposed in [7], yet it has many of its properties. Our structure is as simple as the one proposed in [5], but our structure can treat arbitrary sequences of insertions and deletions whilst theirs can only treat non-interacting insertions and deletions. We also show that weak B-trees support concurrent operations in an efficient way.

1. Introduction

Balanced trees allow the execution of the three basic dictionary operations Search, Insert and Delete in logarithmic time.

In conventional applications searches always start at the root of the tree and then proceed towards the leaves. Hence, they take time $\Theta(\log n)$, where n is the current size of the file. Insertions and Deletions are always preceeded by a search. They consist of adding or pruning a leaf and subsequent rebalancing. Rebalancing is restricted to the path from the new or deleted leaf to the root and calls for local changes of the tree in *some* nodes of that path. Of course, "some" is always $O(\log n)$ and hence it "only" influences the constants in the insertion and deletion times. In a parallel environment, i.e., several processes are operating on the same tree, it also influences the degree of parallelism.

Recently, there has been a growing interest in less conventional applications of balanced trees. In these applications searches start at the leaves,

* A preliminary version of this paper was presented at the 5th workshop on graphtheoretic concepts in computer science, Bad Honnef, June 1980

0001-5903/82/0017/0157/$05.60

proceed towards the root then turn around and proceed downwards again. Thus, insertion and deletion costs are not naturally dominated by the cost of the searches. This observation has led to a number of new data structures [7, 12] as well as to a more detailed analysis of existing data structures [4, 5, 13].

In this paper we explore the use of weak B-trees for the representation of linear lists. In weak B-trees all leaves have the same depth and every interior node has at least a and at most b sons for some constants a, b with $b \geqslant 2a$. In ordinary B-trees [2] we have $b = 2a - 1$. We analyse the cost of sequences of insertions and deletions into weak B-trees and show that this cost is linear in the length of the sequence when the initial tree is empty, and sublinear when $b - 2a$ is sufficiently large. In the case of an arbitrary starting tree we derive a bound in terms of the positions of the insertions and deletions.

We also show that the number of insertions/deletions which require k rebalancing operations (i.e., the last k nodes of the path to the root are effected by local changes) decreases exponentially with k. We conclude that weak B-trees support a high degree of concurrency even in the presence of insertions and deletions.

In Sect. 4 we go on to show that weak B-trees are well suited to represent sorted linear lists. Level-linked weak B-trees allow very efficient searches by the use of fingers. A finger is a pointer to a leaf of the tree. Search times are logarithmic in the distance from the finger. Fingers can be established and moved in constant time, and insertions and deletions take constant time on the average (averaged over a sequence of insertions and deletions). Hence, the cost of the searches dominates the total cost.

In Sect. 5 we use level-linked weak B-trees in order to describe optimal realizations of many set operations; in particular the task of updating a master file against a file of updates can be performed optimally.

Our structure (almost) combines the advantages of the structures proposed in [7] and [5] and avoids their disadvantages. It is much simpler than the structure of [7] and yet has the same behavior for sequences of operations. More precisely, in [7] the cost of every single insertion and deletion is dominated by the preceeding search; in our case, this is only true for a sequence of operations. However, finger creations are much harder and the constants in the O-expressions for the running times are much larger in their case. Next we compare our structure with the one proposed in [5]. Our structure is as simple yet it does a lot more. We can treat arbitrary sequences of insertions and deletions whilst [5] can only manage sequences of non-interacting insertions and deletions.

2. Sequence of Operation Analysis of Weak B-trees

B-trees were introduced by [2]. In B-trees all leaves have the same depth and each internal node has at least a and at most $2a - 1$ sons where a is some constant, the order of the tree. In weak B-trees we allow for a wider range of

arities of nodes. In the sequel, node always stands for interior node, i.e., leaves are not called nodes.

Definition. $\rho(v)$ denotes the number of sons of node v. When T is a tree, $|T|$ denotes the number of leaves of T.

Definition. Let a and b integers with $a \geqq 2$ and $2a-1 \leqq b$.

A tree T is an (a,b)-tree if

a) all leaves of T have the same depth
b) all nodes v of T satisfy $\rho(v) \leqq b$
c) all nodes v except the root satisfy $\rho(v) \geqq a$
d) the root r of T satisfies $\rho(r) \geqq \min(2, |T|)$.

The class of order a B-trees is identical to the class of $(a, 2a-1)$-trees. For $b \geqq 2a$ we will refer to the class of (a,b)-trees as a class of *weak B-trees.* The *hysteresis* of (a, b)-trees is defined to be $\lceil b/2 \rceil - a$, following [10]. In our examples we will always use $(2,4)$-trees. $(2,4)$-trees have been considered in [1] as "symmetric binary B-trees", and in [6] as "$2-3-4$ trees".

Lemma 1. *Let T be an (a,b)-tree of height $h \geqq 1$ with $|T|$ leaves. Then*

$$2 \cdot a^{h-1} \leqq \max(2, |T|) \leqq b^h$$

Proof. Obvious.

We infer from Lemma 1 that the depth of (a,b)-trees is logarithmic in the number of leaves.

Insertion and deletion into (a,b)-trees is quite similar to the corresponding operations in B-trees. An insertion means the addition of a new leaf at a given position in the tree, a deletion means the pruning of an existing leaf at a given position in the tree. Note that we treat the searches for these positions seperately in what follows, i.e., for the moment we concentrate at the rebalancing aspect of (a,b)-trees.

Definition. (T,v) is a partially rebalanced (a,b)-tree, where v is a node of T and r is the root of T, if

a)	$a-1 \leqq \rho(v) \leqq b+1$	if $v \neq r$
	$\min(1, \|T\|) \leqq \rho(v) \leqq b+1$	if $v = r$
b)	$a \leqq \rho(w) \leqq b$	for all $w \neq v, r$
c)	$2 \leqq \rho(r) \leqq b$	if $r \neq v$.

Insertion. An insertion is accomplished by a sequence of node expansions and node splittings, terminating in a balanced (a,b)-tree. Let w be any leaf of T and suppose that a new leaf is to be inserted to the right (left) of w. Let v be the father of w.

1) Expand v, i.e., make the new leaf an additional son of v. The expansion of v increases $\rho(v)$ by 1. If $\rho(v)$ is still $\leqq b$ then rebalancing is complete. Otherwise v needs to be split. Since splitting may propagate we fomulate it as a loop.

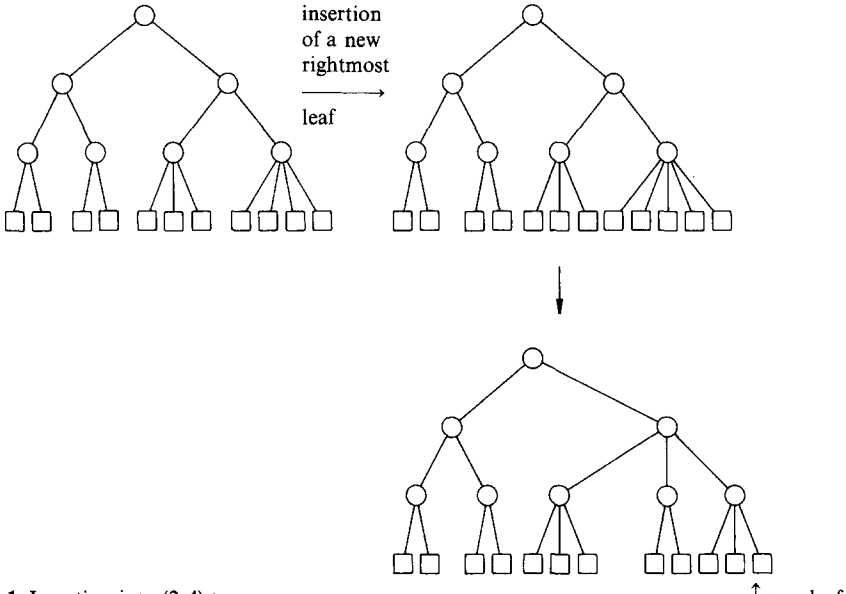


Fig. 1. Insertion into (2,4)-tree

2) **while** $\rho(v)=b+1$
do if v is the root of T
then let x be a new node and make v the only son of x
else let x be the father of v
fi;
let v' be a new node;
expand x; i.e., make v' an additional son of x immediately to the right of v;
split v, i.e., take the rightmost $\lceil (b+1)/2 \rceil$ sons away from v and make them sons of v';
$v \leftarrow x$;
od;

Fig. 1 shows the insertion of a leaf into a (2,4)-tree.

An insertion of a new leaf requires $r+1$ expansions and r node splittings for some integer $r \geqq 0$. It can be accomplished in time $\Theta(1+r)$.

Deletion. A deletion is accomplished by a sequence of node shrinkings and node fusings possibly followed by one node sharing. Deletion has two parameters, t and s, described afterwards.

Let w be any leaf of T (the leaf to be deleted) and let v be the father of w.

1) Shrink v by pruning w. This decreases $\rho(v)$ by 1. If $\rho(v)$ is still $\geqq a$ or the height of v is 1 then rebalancing is completed. (Note that we represent the empty tree by a single node of arity 0). Otherwise, v needs to be rebalanced by either fusing or sharing. Let y be any brother of v.

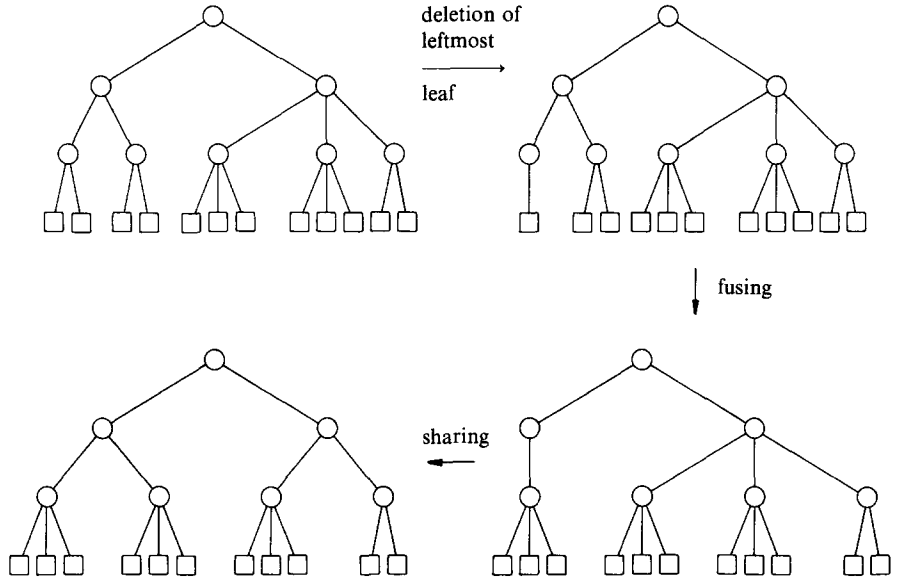


Fig. 2. Deletion from (2,4)-tree

while $(\rho(v)=a-1$ **and** $\rho(y)\leqq a+t)$
do let x be the father of v;
fuse v and y, i.e., make all sons of y sons of v and delete y;
co this will simultaneously shrink x, i.e., decrease its arity by one;
$v\leftarrow x$;
let y be a brother of x;
if x does not have a brother
then begin co x is the root of T;
if $\rho(x)=1$
then delete x
fi;
goto completed;
end
fi
od;
co at this point we either have $\rho(v)\geqq a$ and rebalancing is completed or $\rho(v)=a-1$ and $\rho(y)>a+t$ and rebalancing will be completed by sharing.
if $\rho(v)=a-1$
then take s sons away from y and make them additional sons of v.
fi
completed;

Fig. 2 shows the deletion of a leaf from a (2,4)-tree. Parameters s,t are $t=0$ and $s=1$.

A deletion of a leaf requires $r+1$ node shrinkings, r node fusings for some integer r and possibly one node sharing. It can be accomplished in time $\Theta(1+r)$.

The parameter t is a *sharing threshold*, which specifies when to fuse or share. When $\rho(v)=a-1$ and $\rho(y)=a+j$ during deletion, the algorithm performs a node fusing if $j \leqq t$, and a node sharing otherwise. The parameter s specifies how many sons to shift when sharing.

In (a,b)-trees, any values of the parameters t and s in $0 \leqq t \leqq b+1-2a$ and $1 \leqq s \leqq t+1$ give a correct rebalancing algorithm. We will consider two algorithms in this paper. Let p be the hysteresis of (a,b)-trees.

Algorithm 1 uses $s=\lceil (p+1)/2 \rceil$ and $t=p+s-1$. *Algorithm 2* uses $t=0$ and $s=1$. Algorithm 2 shares whenever possible and thus terminates rebalancing as soon as possible, Algorithm 1 moves the arities of balanced nodes as far away from the critical values a and b as possible and thus invests in the future. Algorithms that use a more general class of rebalancing strategies are considered in [8] and [9].

Next, we want to study the total cost of sequences of insertions and deletions into (a,b)-trees under the assumption that we start with an initially empty tree. We will show that the total cost is linear in the length of the sequence when $b \geqq 2a$. An even stronger result holds for Algorithm 1: the maximum number of rebalancing operations decreases in proportion to $b+1-2a$. This is particularly significant in some of the applications discussed later, where the cost (in disk accesses) of one rebalancing operation, though constant, is considerably larger than the cost accessing a finger or adding or pruning a leaf.

The proof follows a general paradigm for analyzing the cost of sequences of operations, the bank account paradigm. The paradigm defines a bank account for a tree, and associates operations on the tree with deposits and withdrawals in the account. A bound on rebalancing cost follows by relating the constraints on deposits and withdrawals to the initial and final balances of the account.

We will always use a special form of the bank account paradigm that we call a savings account.

Definition. A savings account is a real-valued function V, defined on sets of nodes in partially rebalanced trees, which satisfies properties P1–P3 below. We specify a savings account by giving two functions V_I and V_R on integers (arities of nodes), and define V as follows.

1) $V(x)=V_R(\rho(x))$	if x is the root of a tree,
2) $V(x)=V_I(\rho(x))$	if x is a node other than the root,
3) $V(S)=\sum_{x \in S} V(x)$	for any set S of nodes,
4) $V(T)=\sum_{x \text{ is a node of } T} V(x)$	for any tree T.

The properties satisfied by a savings account are:

P1) $V_I(j) \geqq V_R(j) \geqq 0$	for all j,
P2) $\lvert V_\alpha(j+1)-V_\alpha(j)\rvert \leqq 1$	for $\alpha \in \{I,R\}$ and all j,
P3) $V_R(0)=V_R(1)=0$.	

Note the following fact about any savings account.

Fact S1. a) $V(T) \geqq 0$ for any tree T.

b) If T is the empty tree then $V(T)=0$.

c) If T' is obtained from T by adding or pruning a leaf, then $V(T') \leqq V(T)+1$

Proof. Immediate from properties P1, P3, and P2 respectively.

Fact S2. Let (T, v) be a partially rebalanced tree, and T' be obtained from T by splitting or fusing v. Let x be the father of v before splitting or fusing, an x' be the father afterwards. Then

$$V(x') \leqq V(x)+1.$$

Proof. The result is immediate from property P2 when x and x' both exist. Otherwise a root was created or deleted, and the unordered set $\{V(x), V(x')\}$ is $\{V_R(2), V(\emptyset)\}$. Since $V(\emptyset)=V_R(1)=0$ (property P3), property P2 again gives the result.

Theorem 1. *Let $b \geqq 2a$. Consider an arbitrary sequence of k intermixed insertions and deletions into an initially empty (a, b)-tree. Let $B(k)$ be the total number of rebalancing operations (splittings, fusings, and sharings) during this sequence. Then*

a) $B(k) \leqq k/p$ *using Algorithm 1 when* $b > 2a$
where $p = \lceil b/2 \rceil - a$ *is the hysteresis of* (a, b)*-trees.*

b) $B(k) \leqq 3k/2$ *using Algorithm 2.*

Proof. a) Let $m = \lceil b/2 \rceil$. We define a savings account V for Algorithm 1 by

$$V_I(j) = m + |j-m| - (p+1)$$

on arities of nodes other than the root, and

$$V_R(j) = \max(0, j-(p+1))$$

on root node arity.

Fact 1. Let tree T' be obtained from T by a rebalancing operation using Algorithm 1. Then

$$V(T') \leqq V(T) - p.$$

Proof. We analyze each type of rebalancing operation.

Splitting. Let node x be split into nodes x' and x''. Let $c = b+1$. Then the decrease in V at the level of x is

$$\begin{aligned} V(x) - V(x') - V(x'') &= V_\alpha(c) - V_I(\lceil c/2 \rceil) - V_I(\lfloor c/2 \rfloor) (\alpha \in \{I, R\}) \\ &= c - \lceil c/2 \rceil - \lfloor c/2 \rfloor - (p+1) + 2(p+1) = p+1. \end{aligned}$$

By Fact S2 for savings accounts, V can increase at the father of x by at most 1. Thus

$$V(T) - V(T') \geqq V(x) - V(x') - V(x'') - 1 \geqq p$$

Sharing. Let $s=\lceil(p+1)/2\rceil$. Let node x borrow s sons from node x', and call the resulting nodes y and y'. Then $\rho(x')\geqq m+s$, $\rho(y')\geqq m$, and x's father does not change arity. Thus

$$\begin{aligned}V(T)-V(T')&=V(x)+V(x')-V(y)-V(y')\\&=(V_I(m-p-1)-V_I(m-p-1+s))+(V_I(\rho(x'))-V_I(\rho(x')-s))\\&=2s=2\lceil(p+1)/2\rceil\\&\geqq p+1>p.\end{aligned}$$

Fusing. Let node x be fused with node x' to form node y. Then $\rho(x')=m+j$, $-p\leqq j<\lceil(p+1)/2\rceil$. Also $V(y)=V_\alpha(\rho(y))\leqq V_I(\rho(y))$, where $\alpha\in\{I, R\}$. Let $p'=p+1$. Then the decrease in V at the level of x is

$$\begin{aligned}V(x)+V(x')-V(y)&\geqq V_I(m-p')+V_I(m+j)-V_I(2m-p'+j)\\&=(m+|p'|-p')+(m+|j|-p')-(m+|m-p'+j|-p')\\&=(|m-p'|+|j|-|m-p'+j|)+|p'|-p'-(-p')\\&\geqq p'=p+1.\end{aligned}$$

Thus by Fact S2 for savings accounts,

$$V(T)-V(T')\geqq V(x)+V(x')-V(y)-1\geqq p$$

This completes the proof of Fact 1.

We now summarize the savings account argument for Algorithm 1. We have just shown

$$V(T')\leqq V(T)-p$$

when T' is obtained from T by a rebalancing operation, and we know

$$V(T')\leqq V(T)+1$$

when T' is obtained by adding or pruning a leaf (Fact S1 for savings accounts). Let T_0 be the initial empty tree, and T_k be the tree after the k'th insertion or deletion. Then we have

$$0\leqq V(T_k)\leqq V(T_0)+k-pB(k)=k-pB(k)$$

hence

$$B(k)\leqq k/p$$

which proves part a).

b) For Algorithm 2, we define a savings account V by

$$\begin{aligned}V_R(j)&=\max(0, j-(b-1)),\\V_I(j)&=\max((a+1/3)-j, V_R(j)).\end{aligned}$$

Thus $V_I(j)=(4/3, 1/3, 0, \ldots, 0, 1, 2)$ when $j=(a-1, a, a+1, \ldots, b-1, b, b+1)$.

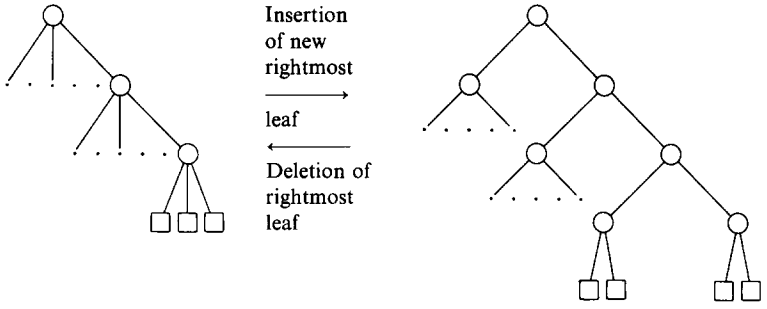


Fig. 3. In ordinary B-trees, $b=2a-1$, rebalancing can always run all the way to the root. The figure shows an example for $a=2$

Fact 2. Let tree T' be obtained from T by a rebalancing operation using Algorithm 2. Then

$$V(T') \leqq V(T) - 2/3.$$

Proof. Follow the proof of Fact 1 for splitting and fusing. We leave it to the reader to show that the decrease in V at the level of splitting or fusing is exactly 5/3, hence $V(T)-V(T')=2/3$. Note that analysis of fusing is simpler here, since fusing node x with x' can occur in only one case, when $\rho(x)=a-1$ and $\rho(x')=a$. Splitting always has just one case.

Now suppose tree T' is obtained from T by sharing, i.e., some node x borrows one son from node x', creating nodes y and y'. Since $\rho(x') \geqq a+1$, we have $a \leqq \rho(y') \leqq b-1$, and $V(x') \geqq 0$ before sharing, $V(y') \leqq 1/3$ after sharing. Thus

$$\begin{aligned} V(T)-V(T') &= V(x)+V(x')-V(y)-V(y') \\ &\geqq V_I(a-1)+0-V_I(a)-1/3 \\ &= 4/3-1/3-1/3=2/3. \end{aligned}$$

Now the savings account argument, using Fact 2, gives

$$0 \leqq V(T_k) \leqq k-(2/3)\,B(k)$$

hence

$$B(k) \leqq 3k/2.$$

This completes the proof of Theorem 1.

Remarks. 1) Note that there is a nonlinear lower bound on $B(k)$ for ordinary B-trees (i.e., $b=2a-1$). Figure 3 shows an example.

However, for ordinary B-trees a similar theorem holds for sequences of insertions only. Define

$$V_I(j)=V_R(j)=\max(0, j-\lceil b/2 \rceil)$$

and use the savings account argument. This fact is mentioned in [2].

2) For $(a, 2a)$-trees, the theorem depends on the fact that the sharing threshold t is 0 in Algorithm 2. [8] shows a nonlinear lower bound on $B(k)$ in any $(a, 2a)$-tree algorithm with $t=1$, even if it makes clever nondeterministic choices of which brother to use with underflow and which side to place the larger node after splitting. It is also shown that for $b>2a$, the natural node splitting strategy (as evenly as possible) guarantees a linear bound on $B(k)$ for any (a, b)-tree algorithm, independent of t.

Theorem 1 shows that on the average the cost of rebalancing is 0(1) if we start with an initially empty tree.

3. Refined Sequence of Operation Analysis

In this section we extend the results of the previous section in two respects. We first compute the distribution of the rebalancing operations on the different levels of the tree and then we extend our analysis to arbitrary initial trees. The second extension will lead to a new data structure in Sect. 4 and to efficient algorithms for several set manipulation problems in Sect. 5. The first extension is motivated by the use of trees in concurrent environments and multidimensional applications. [3] provides us with an elegant method for using trees in a parallel environment. In the analysis of their method Bayer and Schkolnik use the fact that the probability that k splits have to be done after an insertion is exponentially decreasing in k. Our results show that this assumption is justified even in the presence of insertions and deletions. Willard [14] and others use multidimensional trees in the following sense. Trees of dimension 0 are ordinary trees. A tree of dimension k is an ordinary tree T plus a $k-1$ dimensional tree associated with every node of T. Typically in these applications the cost of a rebalancing operation on a node of height h grows exponentially in h. It is therefore important to know that rebalancing operations high up in the tree occur very infrequently.

3.1. On the Distribution of Rebalancing Operations on the Levels of the Tree

We need some more notation. We say that a splitting (fusing, sharing) operation occurs at height h, if node v which is to be split (or fused with its brother y, or share sons with its brother y) has height h; the height of a leaf being 0. A splitting (fusing) operation at height h expands (shrinks) a node at height $h+1$. An insertion (deletion) of a leaf expands (shrinks) a node at height 1.

For any tree T and savings account V, we define the level h account $V^h(T)$ as the sum of $V(x)$ over level h nodes x of T.

A similar but weaker result than Theorem 2(*c*) is also shown in [10].

Theorem 2 (also with Ch. Backes). *Let $b \geqslant 2a$. Consider an arbitrary sequence of k intermixed insertions and deletions into an initially empty (a, b)-tree. Let $B^h(k)$ be the total number of rebalancing operations at level h during this sequence. Let $m = \lceil b/2 \rceil$. Let $p' = (b+1)/2 - a$ be the "fractional hysteresis" of (a, b)-trees, and $p = \lfloor p' \rfloor = m - a$ be the hysteresis. Then*

a) $B^h(k) \leqq k/(p+1)^h$ *using Algorithm 1,*
b) $B^h(k) \leqq k(3/5)^{h-1}$ *in (2,4)-trees,*
c) $B^h(k) \leqq k/c^{h-1}$ *using Algorithm 2 when* $b>4$,
where $c=\min(2p'+1, m-1) \geqq 2$.

Proof. We first examine how insertions and deletions propagate changes to higher levels of the tree.

Fact 1. Let $W^h(k)$ be the number of level h node expansions and shrinkings that occur during k insertions and deletions to an initially empty tree. Let $SF^h(k)$ be the number of level h node splittings and fusings during the same interval. Then

a) $W^1(k)=k$,
b) $W^{h+1}(k)=SF^h(k)$.

Proof. a) Immediate from the definition.

b) Node splitting or fusing at some level h expands or shrinks exactly one node in the tree, at level $h+1$; this might occur after creating or before deleting a 1-ary root. Adding or pruning a leaf does not affect $W^{h+1}(k)$ for $h \geqq 1$, and node sharing never expands or shrinks nodes (shifting sons is considered a different operation).

Now let accounts V be defined for Algorithms 1 and 2 as in Theorem 1. We next examine how V^h for Algorithm 1 is affected by all possible changes in the set of nodes or node arities at level h.

Fact 2. Using Algorithm 1, let tree T' be obtained from T by one of the following operations at level h. If the operation is

a) creating or deleting a root, then $V^h(T')=V^h(T)$,
b) expanding or shrinking a node, then $V^h(T') \leqq V^h(T)+1$,
c) a rebalancing operation, then $V^h(T') \leqq V^h(T)-(p+1)$.

Proof. a) The root must be 1-ary when it is created or deleted, and $V_R(1)=0$ for any savings account. Note that a) is always followed or preceded by b) as part of a node splittings or fusing operation at level $h-1 \geqq 1$.

b) Immediate from property P2 of savings accounts.

c) This has already been proved in Theorem 1. Note that, had we defined Algorithm 1 to shift $s'=\lceil p/2 \rceil$ sons (instead of $\lceil (p+1)/2 \rceil$), Theorem 1 would still hold, but not part a) of Theorem 2.

Fact 3. a) $B^h(k) \leqq W^h(k)/(p+1)$ using Algorithm 1,
b) $B^h(k) \leqq W^h(k)$ using Algorithm 2.

Proof. a) We use the savings account argument, restricted to nodes at level h. Let T_0 be the initial empty tree and T_k be the tree after the k'th insertion or deletion. Then by Fact 2 we have

$$0 \leqq V^h(T_k) \leqq V^h(T_0) + W^h(k) - (p+1)\,B^h(k) = W^h(k) - (p+1)\,B^h(k)$$

for all $h \geqq 1$, hence

$$B^h(k) \leqq W^h(k)/(p+1)$$

b) Each insertion or deletion makes at most one rebalancing operation at level h, and only after expanding or shrinking a node at level h. Hence

$$B^h(k) \leqq W^h(k).$$

Fact 4.

a) $SF^h(k) \leqq W^h(k)/(p+1)$ using Algorithm 1
b) $SF^h(k) \leqq (3/5)\, W^h(k)$ in (2,4)-trees
c) $SF^h(k) \leqq W^h(k)/c$ using Algorithm 2 when $b>4$, where c is as defined in the theorem.

Proof. a) Immediate from Fact 3 and $SF^h(k) \leqq B^h(k)$.

b) We saw in Theorem 1 (for Algorithm 2) that if tree T' is obtained from T by a node splitting or fusing at level h, then $V^h(T') \leqq V^h(T) - 5/3$. For tree T' obtained by node sharing at level h, we only need $V^h(T') \leqq V^h(T)$, although a stronger bounds holds. Now, the savings account argument applied to level h node splittings and fusings gives

$$0 \leqq V^h(T_k) \leqq V^h(T_0) + W^h(k) - (5/3)\, SF^h(k)$$

hence

$$SF^h(k) \leqq (3/5)\, W^h(k).$$

c) As in part b), we use the savings account argument applied to node splittings and fusings. But here the account defined in Theorem 1 for Algorithm 2 does not suffice. We define another savings account V as follows. Let $p'' = \lceil p' \rceil$.

Case 1. If $3p'' \leqq b-a$, define V by

$$V_R(j) = \max(0, j-(b-2p'')),$$
$$V_I(j) = \max((a+p'')-j, V_R(j)).$$

Case 2. If $3p'' > b-a$, define V by

$$V_I(j) = |j-m|+1,$$
$$V_R(j) = \max(0, j-m+1).$$

The savings account argument requires the following fact.

Fact 5. Let tree T' be obtained from T by rebalancing operation at level h, using Algorithm 2 with $b>4$. Then with V defined as above, if T' is obtained by

i) sharing, then $V^h(T') \leqq V^h(T)$,
ii) splitting or fusing, then $V^h(T') \leqq V^h(T) - c$.

Proof. i) Let node x borrow one son from x', creating nodes y and y'. Then in both cases, $V(x) - V(y) = V_I(a-1) - V_I(a) = 1$ and $V(x') - V(y') \geqq -1$. Thus,

$$V^h(T) - V^h(T') = (V(x) - V(y)) + (V(x') - V(y'))$$
$$\geqq 1 + (-1) = 0.$$

ii) For splitting, let node x be split into nodes x' and x''. For fusing, let node x be fused with x' to form node y. We have

$$V^h(T)-V^h(T')=V(x)-V(x')-V(x'') \quad \text{for splitting}$$

where $V(x)=V_\alpha(b+1)=V_I(b+1)$, $\alpha\in\{I,R\}$, and

$$V^h(T)-V^h(T')=V(x)+V(x')-V(y) \quad \text{for fusing,}$$

where $V(y)=V_\alpha(\rho(y))\leqq V_I(\rho(y))$, $\alpha\in\{I,R\}$.

Case 1. $3p''\leqq b-a$.

Case 1.1. b is odd. Then $p''=p'=p$.

Splitting. Note that $b+1=2m$ and $a+p''=m$. We have

$$\begin{aligned}V^h(T)-V^h(T')&=V_I(b+1)-2V_I(a+p'')\\&=(2p''+1)-0=2p'+1\geqq c.\end{aligned}$$

Fusing. Note that $2a-1=2m-1-2p=b-2p''$. We have

$$\begin{aligned}V^h(T)-V^h(T')&\geqq V_I(a-1)+V_I(a)-V_I(2a-1)\\&=(p''+1)+p''-0=2p''+1=2p'+1\geqq c.\end{aligned}$$

Case 1.2. b is even. Then $p''=p'+1/2=p+1$.

Splitting. Note that $b+1=2m+1$ and $a+p''=m+1$. We have

$$\begin{aligned}V^h(T)-V^h(T')&=V_I(2m+1)-V_I(m+1)-V_I(m)\\&=V_I(b+1)-V_I(a+p'')-V_I(a+p)=(2p''+1)-0-1\\&=2p''=2p'+1\geqq c.\end{aligned}$$

Fusing. Note that $2a-1=2m-2p-1=b-2p'=b-(2p''-1)$. We have

$$\begin{aligned}V^h(T)-V^h(T')&\geqq V_I(a-1)+V_I(a)-V_I(2a-1)\\&=(p''+1)+p''-1=2p''=2p'+1\geqq c.\end{aligned}$$

Case 2. $3p''>b-a$.

Splitting. Let $b'=b+1$, so $m=\lfloor b'/2\rfloor$. Then

$$\begin{aligned}V^h(T)-V^h(T')&=V_I(b')-V_I(\lfloor b'/2\rfloor)-V_I(\lceil b'/2\rceil)\\&=(b'-m+1)-(\lfloor b'/2\rfloor-m+1)-(\lceil b'/2\rceil-m+1)\\&=m-1\geqq c.\end{aligned}$$

Fusing.

$$\begin{aligned}V^h(T)-V^h(T')&\geqq V_I(a-1)+V_I(a)-V_I(2a-1)\\&=V_I(m-p-1)+V_I(m-p)-V_I(2m-(2p+1))\\&=(p+2)+(p+1)-(|m-(2p+1)|+1)\\&=2p+2-|(2p+1)-m|.\end{aligned}$$

This last quantity is $m+1$ when $2p+1 \geqq m$, and is $\geqq m-1$ if $2p+1 \geqq m-1$. Now $3p'' > b-a$ gives, when b is odd,

$$3p > (2m-1)-(m-p), \quad \text{hence} \quad 2p+1 > m$$

and when b is even,

$$3(p+1) > 2m-(m-p), \quad \text{hence} \quad 2p+1 > m-2 \quad \text{and} \quad 2p+1 \geqq m-1.$$

Thus we have $2p+1 \geqq m-1$ when $3p'' > b-a$, and

$$\begin{aligned} V^h(T) - V^h(T') &\geqq 2p+2-|(2p+1)-m| \\ &\geqq m-1 \geqq c. \end{aligned}$$

This completes the proof of Fact 5.

To complete the proof of Fact 4, part c), we use Fact 5 and the savings account argument restricted to splittings and fusings. This follows part b) of Fact 4 exactly; the details are left to the reader.

Now from Facts 1 and 4 and induction on h, we have

$W^h(k) \leqq k/(p+1)^{h-1}$	using Algorithm 1
$W^h(k) \leqq k(3/5)^{h-1}$	in $(2,4)$-trees
$W^h(k) \leqq k/c^{h-1}$	using Algorithm 2 when $b > 4$

Combining these results with Fact 3 gives the theorem.

Remarks. 1) Note that Theorem 2 also holds for Algorithm 1 when $p=0$. In $(a, 2a-1)$-trees we have $B^h(k) \leqq k$, which is consistent with $B(k) = 0(k \log k)$. In $(a, 2a)$-trees, Algorithm 1 = Algorithm 2, but parts b) and c) give a better bound than a).

2) Theorem 1 shows that Algorithm 1 is preferable to Algorithm 2 for reducing total rebalancing cost when $p \geqq 1$. Theorem 2 shows that Algorithm 2 is preferable for reducing propagation of rebalancing to higher levels of the tree.

3) $B^h(k)$ is the number of insertions and deletions which require rebalancing up to height h or higher. Theorem 2 shows that this number is exponentially decreasing with h.

We conclude from this last remark that the analysis in [3] can be used to show that (a,b)-trees for $b \geqq 2a$ behave well in a parallel environment in the presence of *insertions and deletions.*

3.2. Arbitrary Initial Tree

In this section we treat the case of an arbitrary initial tree.

Let T_0 be an arbitrary (a,b)-tree. Suppose now that we execute a sequence of k insertions and deletions on T_0, using either Algorithm 1 or Algorithm 2, and obtain tree T_k. Let $B(k)$ be the number of rebalancing operations that occur during this sequence. Then we can derive a bound on $B(k)$ using the savings account argument as follows. Let N be the set of nodes in T_0. Let account V be as defined in Theorem 1, w be the minimum amount by which

each rebalancing operation decreases V, and let $W = V_I(b) = \max_{a \leqq j \leqq b} V_I(j) \geqq \max_{0 \leqq j \leqq b} V_R(j)$. Then we have

$$0 \leqq V(T_k) \leqq V(T_0) + k - w\,B(k)$$

hence

$$B(k) \leqq k/w + V(T_0)/w \leqq k/w + (W/w)\,|N|.$$

However, the term $(W/w)\,|N|$ in this bound is much larger than necessary when $k \ll |N|$, since only a small subset of nodes in N can be affected by k insertions or deletions. It is also possible to significantly decrease the factor W/w for Algorithm 1 when $2a$ is close to b.

In Theorem 3 we give a bound on rebalancing cost starting from an arbitrary tree that answers these shortcomings. Let N_k^0 be the set of nodes in T_0 that are affected by rebalancing during k insertions or deletions, and let N_k be the similar set of nodes in T_k. Then

$$V(T_0) - V(T_k) = V(N_k^0) - V(N_k) \leqq (W/w)\,|N_k^0|.$$

We actually relate N_k^0 to the set A_k^0 of ancestors in T_0 of leaf positions at which the k insertions and deletions occur. It is intuitively obvious that only nodes in A_k^0 and their successors in subsequent trees (possibly after rebalancing) can overflow or underflow and initiate a rebalancing operation. However, N_k^0 can contain nodes not in A_k^0, which participate in fusing or sharing.

The proof uses the savings account argument, restricted to the set A_k^0 and its successors in subsequent trees. The fact that sometimes $N_k^0 \nsubseteqq A_k^0$ requires using different accounts V (with smaller w) than in Theorem 1, but also reduces W/w to 1 for Algorithm 1. We derive a bound on $|A_k^0|$ in Theorem 4.

Theorem 3. *Let $b \geqq 2a$. Consider any sequence of k intermixed insertions and deletions into an arbitrary initial tree T_0. Let $B(k)$ be the total number of rebalancing operations during this sequence. Let $p_1, \ldots, p_k$ be the set of leaf positions in T_0 at which leaves are eventually inserted or deleted in constructing T_k, ordered $p_1 \leqq p_2 \leqq \ldots \leqq p_k$, and let A be the number of ancestors in T_0 of these leaf positions. Then*

a) $B(k) \leqq 2k/p + A$ *using Algorithm 1 when $b > 2a$ where $p = \lceil b/2 \rceil - a$ is the hysteresis of (a, b)-trees.*

b) $B(k) \leqq 2k + 2A$ *in $(a, 2a)$-trees,*

c) $B(k) \leqq 2k + A$ *using Algorithm 2 when $b > 2a$.*

Proof (all parts). We first describe a process for marking nodes during the rebalancing process, and labelling the leaves of the trees constructed with positions of leaves in T_0. The process also (conceptually) keeps a copy of T_0, and marks certain nodes in T_0. For any tree T' in the sequence of partially rebalanced trees constructed during k insertions and deletions, let $M_{T'}$ denote the set of marked nodes in T', and $M_{T'}^0$ denote the set of nodes in T_0 that have been marked while constructing T'.

Initially, we have $T' = T_0$, $M_{T'} = M_{T'}^0 = \emptyset$, and all leaves in T' are labelled with their position in T_0, numbered 0 to $|T'| - 1$. Let $U_{T'}$ be the set of unmarked nodes in T', and $U_{T'}^0$ be the set of nodes in T_0 but not in $M_{T'}^0$.

The following fact is certainly true initially, and is easily seen to remain true after each step in insertion or deletion.

Fact 1. If node x is in $U_{T'}$, then the subtree rooted at x is unchanged from T_0 to T', i.e.,

a) x has not participated in any rebalancing operation,

b) no leaf has been added or pruned from any descendant of x.

The marking process now proceeds as follows, where tree T' is obtained from T by one step in insertion or deletion.

Case 1. A rebalancing operation: Mark all nodes in T' that participated in rebalancing, and leave $M^0_{T'} = M^0_T$.

Case 2. The j'th adding or pruning of a leaf.

a) The leaf position p_j in T_0 corresponding to the j'th insertion or deletion is determined as follows. If leaf l_j is pruned, let p_j be its label. If leaf l_j is added, let p_j be the label of an adjacent brother leaf in T' (or $p_j = 0$ if there is no brother), and label l_j with p_j. If there are two adjacent brothers of l_j with distinct fathers, choose p_j to be the label of the brother with the same father as l_j.

b) Mark all ancestors A_j in T' of leaf l_j and all ancestors A^0_j in T_0 of leaf p_j.

In Case 2(*b*), let UA_j and UA^0_j be the sets of newly marked nodes (unmarked ancestors) in T' and T_0. Thus

$$UA^0_j = U^0_T \cap A^0_j \quad \text{and} \quad UA_j = (U_T \text{ in } T') \cap A_j.$$

Also let UA^T_j be the set (UA_j in T).

Note that $M^0_{T_k}$ is the disjoint union of UA^0_j, $1 \leqq j \leqq k$, which is just the set of ancestors in T_0 of the leaves p_j defined in Case 2(*a*).

Next we examine the savings account argument restricted to the sets $M_{T'}$. We consider adding or pruning a leaf in Fact 2, and rebalancing operations in Fact 3.

Fact 2. Let tree T' be generated from T by the j'th adding or pruning of a leaf. Let V be any savings account. Then

a) $V(M_{T'}) \leqq V(M_T) + V(UA^T_j) + 1$,

b) $V(UA^T_j) \leqq V(UA^0_j)$.

Proof. a) The set ($M_{T'}$ in T) is just $M_T \cup UA^T_j$, and adding or pruning a leaf expands or shrinks one node in this set by 1.

b) Let UA' be the set of nodes (UA^T_j in T_0). Then $UA' \subseteq UA^0_j$ by Fact 1, and nodes in UA' have not changed in constructing T. Thus $V(UA^T_j) = V(UA') \leqq V(UA^0_j)$. Note that $UA^0_j - UA'$ consists only of nodes in T_0 that participate in fusing or sharing during the construction of T from T_0.

We use the following savings accounts for parts a) through c) of the Theorem.

a) For Algorithm 1 when $b > 2a$ (hence $p = \lceil b/2 \rceil - a > 0$),

$$V_R(j) = \max(0, j - (b - \lceil p/2 \rceil),$$
$$V_I(j) = \max((a + \lceil p/2 \rceil) - j, V_R(j)).$$

b) For $(a, 2a)$-trees,

$$V_R(j) = \max(0, j-(2a-1)),$$
$$V_I(j) = \max((a+1/2)-j, V_R(j)).$$

c) For Algorithm 2 when $b > 2a$,

$$V_R(j) = \max(0, j-(b-1/2)),$$
$$V_I(j) = \max((a+1/2)-j, V_R(j)).$$

Fact 3. In cases a) through c) of the theorem, let tree T' be obtained from T by a rebalancing operation. Then

$$V(M_{T'}) \leqq V(M_T) - w$$

where $w = \lceil p/2 \rceil$ for part a) and $w = 1/2$ for parts b) and c).

Proof. Let S and S' be the sets of nodes in T and T' that participate in rebalancing. Case 1 of the marking process guarantees that $S' \subseteq M_{T'}$. Case 2 guarantees that an unbalanced node and its father (if present) are in M_T. Thus for node splitting, $S \subseteq M_T$, and $V(M_T) - V(M_{T'}) = V(S) - V(S')$. We leave it to the reader to show the result for splitting, following Theorem 1.

For fusing and sharing, let node x either fuse with x' (forming node y) or borrow from x' (resulting in nodes y and y'). Then x' may or may not be in M_T. But, letting $M'' = M_T - \{x'\}$, we need only show

$$V(M_{T'}) \leqq V(M'') - w$$

since $V(M'') \leqq V(M_T)$.

Fusing. We have $V(x) = 1 + w$ in all cases. In cases b) and c) we have $V(y) = V_\alpha(2a-1) = 0$, $\alpha \in \{I, R\}$. In case a), let $m = \lceil b/2 \rceil$. Then

$$\rho(x') \leqq m + \lceil (p+1)/2 \rceil - 1 = m + \lfloor p/2 \rfloor,$$

hence

$$\rho(y) \leqq (m-p-1) + (m + \lfloor p/2 \rfloor) = (2m-1) - \lceil p/2 \rceil \leqq b - \lceil p/2 \rceil,$$

so $V(y) = 0$ also in case a). Thus

$$V(M'') - V(M_{T'}) \geqq V(x) - V(y) - 1 = w.$$

Sharing. a) We have $V(x) = 1 + \lceil p/2 \rceil$ before sharing, $V(y') = 0$ and $V(y) = V(x) - \lceil (p+1)/2 \rceil \leqq 1$ after sharing. Then

$$\begin{aligned} V(M'') - V(M_{T'}) &= V(x) - V(y) - V(y') \\ &\geqq (1 + \lceil p/2 \rceil) - 1 - 0 = \lceil p/2 \rceil. \end{aligned}$$

b, c) We have $V(x) = 3/2$, $V(y) = 1/2$, and $V(y') \leqq 1/2$, similar to Fact 2 in Theorem 1. Then

$$\begin{aligned} V(M'') - V(M_{T'}) &= V(x) - V(y) - V(y') \\ &\geqq 3/2 - 1/2 - 1/2 = 1/2. \end{aligned}$$

Now let w be as defined in Fact 3. Then by induction on the steps of insertion and deletion, Facts 2(a) and 3 give

$$V(M_{T_k}) \leqq V(\emptyset) + \sum_{1 \leqq j \leqq k} (V(U A_j^T) + 1) - wB(k).$$

Let $A = M_{T_k}^0$, the set of ancestors in T_0 of leaves p_j, $1 \leqq j \leqq k$. We have $V(U A_j^T) \leqq V(U A_j^0)$ by Fact 2(b), and $\sum_{1 \leqq j \leqq k} V(U A_j^0) = V(A)$. Thus

$$0 \leqq V(M_{T_k}) \leqq k + V(A) - wB(k)$$

hence

$$B(k) \leqq k/w + V(A)/w$$

by the savings account argument.

Now let $W = \max_{a \leqq i \leqq b} V_I(i) \geqq \max_{0 \leqq i \leqq b} V_R(i)$. Then $V(A) \leqq W|A|$.

This shows

$$B(k) \leqq k/w + (W/w)|A|$$

where

a) $W/w = \lceil p/2 \rceil / \lceil p/2 \rceil = 1$ for Algorithm 1 when $b > 2a$,
b) $W/w = V(2a)/w = 1/(1/2) = 2$ for $(a, 2a)$-trees,
c) $W/w = (1/2)/(1/2) = 1$ for Algorithm 2 when $b > 2a$.

This completes the proof of Theorem 3.

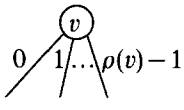
We now derive a bound on the number of ancestors of a set of leaves, given their positions. We use the name (a, ∞)-tree for any tree T where each interior node except the root has at least a sons, and the root has at least $\min(2, |T|)$ sons.

Theorem 4. *Let T be an (a, ∞)-tree with N leaves. Let $1 \leqq p_1 \leqq p_2 \leqq \ldots \leqq p_r \leqq N$. Let m be the total number of ancestors of the leaves in positions p_i, $1 \leqq i \leqq r$. Then*

$$m \leqq 3r + 2\left(\lfloor \log_a N \rfloor + \sum_{i=2}^{r} \lfloor \log_a (p_i - p_{i-1} + 1) \rfloor\right).$$

Proof. A proof of this fact for the case $a = 2$ can be found in [5] (Lemma 7). We give a self-contained proof here.

For every node v label the outgoing edges with $0, \ldots, \rho(v) - 1$ from left to right



Then a path from the root to a node corresponds to a word over alphabet $\{0, 1, 2, \ldots\}$ in a natural way.

Let A_i be the number of edges labelled 0 on the path from the root to leaf p_i, $1 \leqq i \leqq r$. Since an (a, ∞)-tree of height h has at least $2 \cdot a^{h-1}$ leaves, we conclude $0 \leqq A_i \leqq 1 + \lfloor \log_a N/2 \rfloor$. Furthermore, let l_i be the number of interior nodes on the path from leaf p_i to the root which are not on the path from leaf

p_{i-1} to the root. Then

$$m \leqq 1 + \lfloor \log_a N/2 \rfloor + \sum_{i=2}^{r} l_i.$$

Consider any $i \geqq 2$. Let v be the lowest common node on the paths from leaves p_{i-1} and p_i to the root. Then edge k_1 is taken out of v on the path to p_{i-1} and edge $k_2 > k_1$ is taken on the path to p_i.

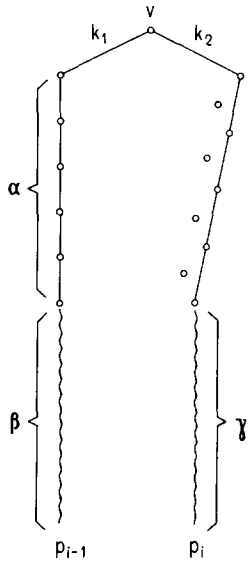
Note that the path from v to leaf p_{i-1} as well as to leaf p_i consists of $l_i + 1$ edges.

Claim. $A_i \geqq A_{i-1} + l_i - 2\lfloor \log_a(p_i - p_{i-1} + 1) \rfloor - 3$.

Proof. The paths from p_{i-1} and p_i to the root differ only below node v. Let s be minimal such that

a) the path from v to p_{i-1} has the form $k_1 \alpha\beta$ with $|\beta| = s$ and α contains no 0.

b) the path from v to p_i has the form $k_2 0^{|\alpha|} \gamma$ for some γ with $|\gamma| = s$. Note that either β starts with a 0 or γ starts with a non-zero and that $|\alpha| + |\beta| = l_i$.



Then

$$\begin{aligned} A_i = A_{i-1} &+ \textbf{if } k_1 = 0 \textbf{ then } -1 \\ &+ |\alpha| \\ &+ \text{number of zeroes in } \gamma \\ &- \text{number of zeroes in } \beta \\ \geqq A_{i-1} &- 1 + (l_i - s) + 0 - s. \end{aligned}$$

It remains to show that $s \leqq 1 + \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor$.

This is certainly the case if $s = 0$. Suppose now that $s > 0$.

We noted above that either β starts with a zero or γ starts with a non-zero. In the first case consider node w which is reached from v via $k_1 \alpha 1$, in the second case node w which is reached from v via $k_2 0^{|\alpha|} 0$. All leaf descendants of w lie properly between p_{i-1} and p_i. Furthermore, w has height $s-1$ and hence at least a^{s-1} leaf descendants. This proves

$$a^{s-1} \leqq p_i - p_{i-1} - 1$$

and hence

$$\begin{aligned} s &\leqslant 1 + \lfloor \log_a(p_i - p_{i-1} - 1) \rfloor \\ &\leqq 1 + \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor. \end{aligned}$$

Using our claim repeatedly, we obtain

$$\begin{aligned} A_r \geqq A_1 + \sum_{i=2}^{r} l_i - 2 \sum_{i=2}^{r} \log_a(p_i - p_{i-1} + 1) \\ - 3(r-1). \end{aligned}$$

Since $A_r \leqq 1 + \log_a \lfloor N/2 \rfloor$ and $A_1 \geqq 0$, this proves

$$\begin{aligned} \sum_{i=2}^{r} l_i \leqq 3r - 3 + 1 + \lfloor \log_a N/2 \rfloor \\ + 2 \sum_{i=2}^{r} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor \end{aligned}$$

and hence

$$m \leqq 3r - 1 + 2 \lfloor \log_a N/2 \rfloor + 2 \sum_{i=2}^{r} \lfloor \log_a(p_i - p_{i-1} + 1) \rfloor.$$

This completes the proof of Theorem 4.

4. A Representation for Linear Lists with Fingers

In this section we use (a, b)-trees in order to represent linear lists. This section follows [5] very closely.

Let L be a sorted sequence of n items drawn from some linearly ordered universe U. Let T be an (a, b)-tree with n leaves. We say that T represents L if

1) the elements of L are stored in the n leaves of T in increasing order from left to right.

2) in each interior node v of T, $\rho(v) - 1$ keys (= elements of U) are stored. If k is the i-th key in node v then the elements in all leaves in the $(i-1)$-th subtree (i-th subtree) of v are less than or equal (greater than) k.

Figure 4 gives an example.

A finger into list L is a pointer to an element of L. Fingers may be used to indicate areas of high activity in list L. (a, b)-trees as they stand do not support efficient search in the vicinity of fingers. This is due to the fact that neighboring leaves may be connected only by a very long path. Therefore, we introduce *level-linked* (a, b)-trees.

In level linked (a,b)-trees all tree edges are made traversible in both directions (i.e., there are also pointers from sons to fathers); in addition each node has pointers to the two neighboring nodes on the same level. Figure 5 gives an example.

A finger into a level-linked (a,b)-tree is a pointer to a leaf. Level-linked (a,b)-trees allow very fast searching in the vicinity of fingers.

Lemma 1. *Let p be a finger in a level-linked (a,b)-tree T. A search for a key k which is d keys away from p takes time $\Theta(1+\log d)$.*

Proof. We first check whether k is to the left or right of p, say k is to the right of p. Then we walk towards the root, say we reached node v. We check whether k is a descendant of v or v's right neighbor on the same level. If not, then we proceed to v's father. Otherwise we turn around and search for k in the ordinary way.

Suppose that we turn around at node w of height h. Let u be that son of w which is on the path to the finger p. Then all descendants of u's right neighbor lie between the finger p and key k. Hence, the distance d is at least 2^{h-1}. The time bound follows.

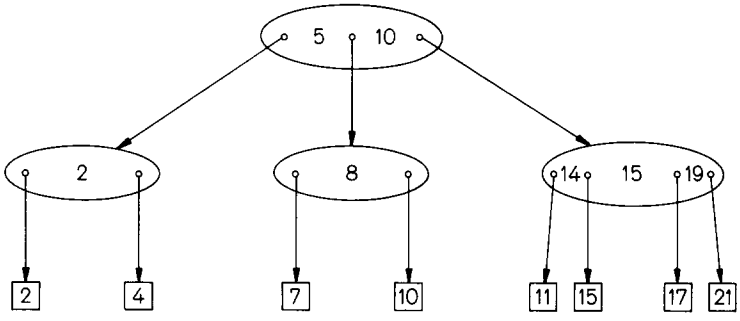


Fig. 4. A (2,4)-tree for list 2, 4, 7, 10, 11, 15, 17, 21. The universe is the set of natural numbers.

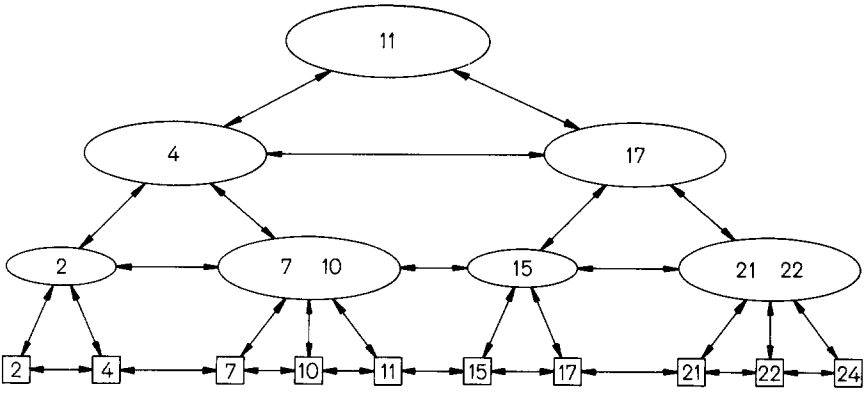


Fig. 5. A (2,4)-level linked tree for list 2, 4, 7, 10, 11, 15, 17, 21, 22, 24

Lemma 2. *A new leaf can be inserted in a given position of a level-linked (a,b)-tree in time $\Theta(1+s)$, where s is the number of splittings caused by the insertion.*

Proof. Let w be the leaf to be split, let cont(w) be the element stored in w, let x be the element to be inserted and let v be the father of w. We give v an additional leaf son immediately to the right of w, say w' and store $\min(x, \text{cont}(w))$ in w, $\max(x, \text{cont}(w))$ in w' and we make $\min(x, \text{cont}(w))$ the key between the pointers to w and w' in v. Next consider a split. If v is to be split it is easy to update the links in constant time. To maintain the key organization we place the left $\lfloor (b+1)/2 \rfloor - 1$ (right $\lceil (b+1)/2 \rceil - 1$) keys of v into the two new nodes produced by the split, and we move the remaining key into the father of v.

Lemma 3. *A leaf can be deleted from a level-linked (a,b)-tree in time $\Theta(1+f)$, where f is the number of node fusings caused by the deletion.*

Proof. Suppose that leaf w has to be deleted. This is achieved by deleting leaf w, the pointer to w in the father of w and one of the keys adjacent to the pointer (i.e., if w is the i-th son of v then we remove either the $(i-1)$-th key or the i-th key of v). The details of sharing or fusing are left to the reader.

Lemma 4. *Creation or removal of a finger in a level-linked (a,b)-tree takes time $\Theta(1)$.*

Proof. Obvious.

Now we apply our result of Sect. II and show that even though the search time in level linked (a,b)-trees can be greatly reduced by maintaining fingers, it still dominates the order of total execution time when $b \geqq 2a$. But note that some rebalancing operations are very expensive in level-linked trees (Remark 3, following). Thus, rebalancing cost can exceed total search cost by a large constant factor if all searches are for keys near fingers.

Theorem 5. *Let $b \geqq 2a$. Then any sequence of searches, finger creations, finger removals, insertions and deletions starting with an empty list takes time*

$$0\,(\textit{total cost of searches})$$

if a level-linked (a,b)-tree is used to represent the list.

Proof. Let n be the length of the sequence. Then the total cost for the searches is $\Omega(n)$ by Lemma 1. On the other hand the total cost for the finger creations and removals is $O(n)$ by Lemma 4 and the total cost of insertions and deletions is $O(n)$ by Lemma 2 and 3 and Theorem 1.

Theorem 6. *Let $b \geqq 2a$. Let L be a sorted list of n elements represented as a level-linked (a,b)-tree with one finger established. Then in any sequence of searches, finger creations, insertions, and deletions, the total cost of the sequence is*

$$O(\log n + \textit{total cost of searches}).$$

Proof. Let S be any sequence of k searches, finger creations, insertions and deletions containing exactly s insertions and d deletions. Let T_0 and T_k by the

(a, b)-trees which represent list L before and after S is performed, respectively. Assume that deleted elements (conceptually) remain in T_k as phantoms (cf. Fig. 6), which are invisible for purposes of searching, determining arities of nodes, or rebalancing. Thus, T_k has $n+s$ (real or phantom) leaves, and the positions of all operations in S correspond to leaves in T_k. Assign a label $l(p)$ in the range $0, \ldots, n-1$ to each leaf p in T_k, representing a leaf position in T_0, as in the proof of Theorem 3.

We now proceed precisely as in the proof of Theorem 4 in [5]. The details are left to the reader.

Remarks. 1) Theorem 6 is not true for ordinary B-trees, $b=2a-1$. In that case a related result was proven in [5]; they show that the theorem is true if the sequence of operations either does not contain insertions or does not contain deletions, or if insertions and deletions do not interact too much.

2) [7] describes a data structure which achieves a similar time bound in the worst case, i.e., for every single insertion and deletion the cost is bounded by the preceeding search; we claim this only for the average over a sequence. However, they have to pay a price for it: fingers creation are much harder and the constants in the bounds for the run times are much larger.

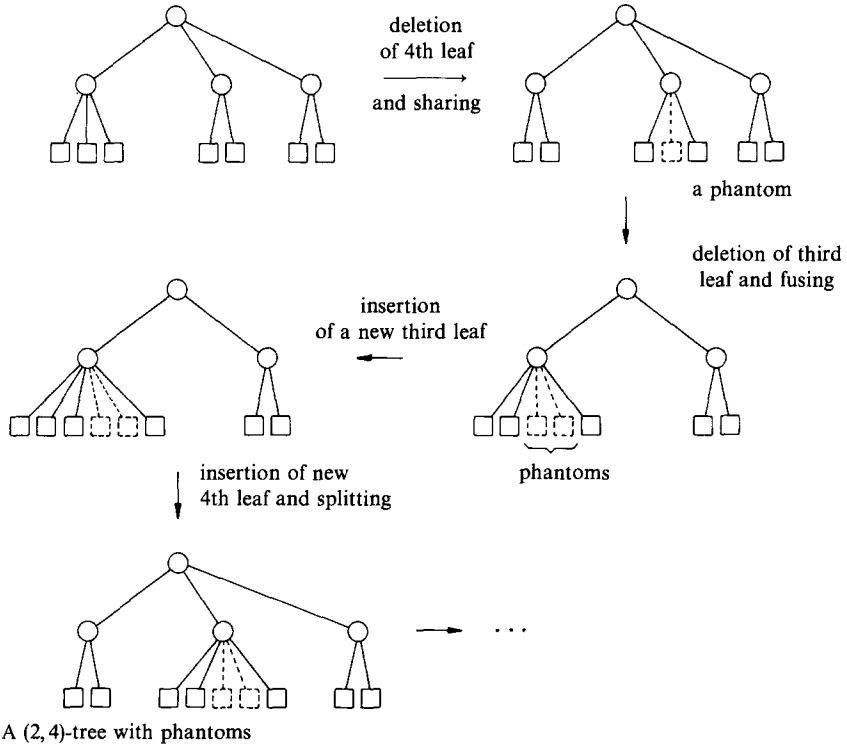


Fig. 6. A (2, 4)-tree with phantoms

3) It was remarked by one of the referees that in conventional applications (trees of high arity, stored on secondary storage) the constants in the 0-expressions in Lemmas 2 and 3 and hence in Theorem 6 are rather large. Note that splitting (or fusing) requires the change of about $b/2$ (a resp.) son-to-father pointers. In the applications mentioned above, this would require fetching about that number of nodes from secondary memory.

We will next describe a data structure, (a,b)-trees with fingers, which overcomes this difficulty at the cost of increased search times and yet supports most applications described in Sect. 5.

In an (a,b)-tree with fingers each node has the ability to store a pointer to its father. However, only nodes on a finger path, i.e., a path from one of the fingers to the root, make use of that ability and actually contain a pointer to their father. Also each node on a finger path knows which of his sons also are on a finger path. For all other nodes the son-to-father pointers contain trash. Also there are no side-links. We remark that in some applications, in particular, if there will be only one finger, it is preferable to store the finger paths separate from the tree, say in a linked list or pushdown store.

A search in an (a,b)-tree with fingers proceeds as follows. Say we start at finger p and search for key k. We first check whether k is to the right or left of p: say k is to the right of p. Then we walk to the root until we reach a node v such that k is a descendant of v. We turn around at v and search for k in the ordinary way. Also on the way down from v we set the son-to-father pointers of the descendants of v to their correct value. Once we reach the leaf level, we can establish a finger at k at the cost of $O(1)$, or insert/delete k at the cost $O(s)$, where s is the number of splittings/fusings caused by the insertion/deletion.

This shows that Lemmas 2 to 4 stay true, if level-linked (a,b)-tree is replaced by (a,b)-tree with fingers. Note however, that only the son-to-father links on the finger paths need to be maintained and hence the cost of a splitting/fusing will be generally lower in (a,b)-trees with fingers than in level-linked (a,b)-trees.

However, there is a price which we have to pay. Lemma 1 does not stay true. Rather the cost of a search is the height of node v defined above.

Lemma 1'. *Let p be a finger in an (a,b)-tree with fingers. Let k be a key which is d keys away from p and let h be the height of the lowest common ancestor of p and k. Then the cost of a search for k starting at p is*

$$\Theta(1+h)=\Omega(1+\log d).$$

Proof. By preceding discussion.

Lemma 1' tells us that a search in an (a,b)-tree with fingers is never cheaper than the corresponding search in a level-linked (a,b)-tree. Hence theorems 5 and 6 stay true if we replace level-linked (a,b)-trees by (a,b)-trees with fingers.

Theorem 6'. *Same as theorem 6 but level-linked (a,b)-tree replaced by (a,b)-tree with fingers.*

5. Applications of Level-linked (a, b)-Trees

In general, we advise to use level-linked (a,b)-trees ($b \geqq 2a$) whenever there are (maybe time-varying) areas of high activity. The finger will make the searches very fast, and finger creations, insertions and deletions take constant time on the average. This situation occurs quite frequently in the implementation of event lists.

More specifically, level-linked (a,b)-trees permit the optimal realization of many set operations.

Theorem 7. *Let A and B be sets represented as level-linked (a,b)-trees, $b \geqq 2a$.*

a) Insert(A,x), Delete(A,x), Search(A,x), Concatenate(A,B) *and* Split(A,x) *can be done in logarithmic time. (Here* Concatenate$(A,B)=A \cup B$ *if* $\max A < \min B$ *and undefined otherwise and* Split$(A,x)=(A_1,A_2)$ *where* $A_1=\{a;\ a \in A$ *and* $a \leqq x\}$ *and* $A_2=\{a;\ a \in A$ *and* $a>x\}$.

b) Let $n=\max(|A|,|B|)$ *and* $m=\min(|A|,|B|)$.

Then $A \cup B$, $A \oplus B$, $A \cap B$, $A \setminus B$ *can be constructed in time* $O(\log(\binom{n+m}{m}))$.

Proof. a) The algorithms are the same as for 2-3 trees. We refer the reader to [11] for details.

b) We first show how to perform $A \oplus B$.

Assume w.l.o.g. $|A| \geqq |B|$. The algorithm is as follows:

a) establish a finger at the first element of A

b) **while** B not exhausted
do
(b.1) take the next element, say x, of B and search for it in A starting at the finger
(b.2) insert or delete x from A, whatever is appropriate
(b.3) establish a finger at the position of x in A and destroy the old finger
od.

Let $p_1,\ldots,p_m$, $m=|B|$, be the positions of the elements of B in the set $A \cup B$, let $p_0=1$. Then the above program takes time ($|A|=n$)

$$O\left(m+\log(n+m)+\sum_{i=0}^{m-1}\log(p_{i+1}-p_i+1)\right)$$

by Theorem 6 and the observation that total search time is bounded by

$$\sum_{i=1}^{m-1}\log(p_{i+1}-p_i+1).$$

This expression is maximized for $p_{i+1}-p_i=(n+m)/m$ for all i, and then has value $O(\log(n+m)+m\log((n+m)/m))=O(m\log((n+m)/m))=O(\log(\binom{n+m}{m}))$.

In the case of $A \cup B$, we only do insertions in line (b.2). In the case of $A \cap B$, we collect the elements of $A \cap B$ in line (b.2) (there are at most m of them) and construct a level-linked (a,b)-tree for them afterwards.

Finally, we have to consider $A \backslash B$. If $|A| \geqq |B|$, then we use the program above. If $|A| < |B|$ then we scan through A linearly, search for the elements of A in B as described above (roles of A and B reversed) and delete the appropriate elements from A. Apparently, the same time bound holds.

Note that there are $\binom{n+m}{m}$ possibilities for B as a subset of $A \cup B$. Hence $\log\binom{n+m}{m}$ is also a lower bound on the complexity of union and symmetric difference.

Next we prove the corresponding theorem for (a, b)-trees with fingers.

Theorem 7'. *Same as Theorem 7, but level-linked (a, b)-tree replaced by (a, b)-tree with fingers.*

Proof. Part a) is obvious. For part b) we use the same algorithms as in the proof of Theorem 7. Note that the cost of establishing a finger at the first element of A is $O(\log n)$, since we have to traverse the left spine of the tree for A in order to establish the son-to-father pointers at the finger path. From Theorem 6' we infer that the cost of the program is bounded by

$$O(\log n + \text{total cost of the searches}).$$

It remains to derive a bound on the total cost of the searches. Whilst this task was trivial in the case of level-linked trees (using Lemma 1), it is non-trivial in the case of (a, b)-trees with fingers (using Lemma 1'). We will establish such a bound using Theorem 4. (Note that Theorem 4 gives a bound on the sum of the heights of the lowest common ancestors (l.c.a.) of positions p_{i-1} and p_i, $1 \leqq i \leqq r$.) Before we can apply Theorem 4 we need a lemma about the effect of insertions/deletions on the height of l.c.a.'s.

Lemma 5. *Let T be an (a, b)-tree, $b \geqq 2a - 1$. Let x and y be two leaves of T and let v be the l.c.a. of x and y. Let z be a leaf different from x and y*

a) let v' be the l.c.a. of x and y after the deletion of leaf z. Then $\text{height}(v') \leqq \text{height}(v) + 1$

If $\text{height}(v') = \text{height}(v) + 1$ *then z is a descendant of v' but not a descendant of v. If z is a descendant of v then* $\text{height}(v') \leqq \text{height}(v)$.

b) Let v' be the l.c.a. of x and y after splitting leaf z. Then $\text{height}(v') \geqq \text{height}(v)$.

Proof. a) A deletion of a leaf is followed by a sequence of fusings followed by at most one sharing. A fusing combines two nodes and can therefore never increase the height of the l.c.a. of x and y. Next consider the case of a sharing, say node u takes away some son from node w. Then z was a descendant of u. Also if either u and w are both descendants of v or if neither of them is, then v does not lose descendants and hence v is still an ancestor of x and y after the sharing. This shows $\text{height}(v') \leqq \text{height}(v)$ in this case. This leaves the case that exactly one of the nodes u and w is a descendant of v. Since u and w are brothers, this can only be the case if either u or w is equal to v and the other is a brother of v. If u is equal to v, then v stays an ancestor and hence $\text{height}(v') \leqq \text{height}(v)$.

If w is equal to v then either v stays ancestor of x and y or v's father v' becomes the l.c.a. of x and y. In the latter case z is a descendant of v' but not of v. This proves the lemma.

b) A split can never decrease the height of a l.c.a.

Next we will use Lemma 5 in order to show that the total cost of the searches is maximal if our algorithm has to compute $A \cup B$, i.e., interspersed deletions can only decrease search times. In the case of $A \cup B$, Theorem 4 gives us a bound on total search time.

Let $B = \{x_1, \ldots, x_m\}$, $x_1 < x_2 < \ldots < x_m$. Our algorithm for computing $A \oplus B$ $(A \cup B, \ldots)$ processes the x's in increasing order. When x_i is processed, we have processed $x_1, \ldots, x_{i-1}$ already, a finger p at the position of x_{i-1} is established and we search for x_i starting at that finger. Let $h_j(i)$ be the height of the lowest common ancestor of x_i and x_{i-1} (more precisely, of the leaves where the searches for x_{i-1} and x_i are going to end) after processing $x_1, \ldots, x_j$. If x_j is inserted into A by our algorithm then $h_j(i) \geqq h_{j-1}(i)$. If x_j is deleted from A by our algorithm, then $h_j(i) \leqq h_{j-1}(i) + 1$ by part a) of Lemma 5. Furthermore, if $h_j(i) = h_{j-1}(i) + 1$, then x_j was a descendant of the new l.c.a. of x_{i-1} and x_i and hence $x_{j+1}, \ldots, x_{i-2}$ are descendants of that new l.c.a. This shows that there is at most one key among $x_1, \ldots, x_{i-2}$ such that its deletion increases the height of the lowest common ancestor of x_{i-1} and x_i. This observation together with the fact that insertions never decrease the height of l.c.a.'s shows that the cost of the search for x_i in our algorithm, i.e., the height of the l.c.a. of the finger p and x_i, is bounded by $1+$ the height of the l.c.a. of x_{i-1} and x_i in the tree constructed for $A \cup B$ by our algorithm.

We are now in a position to use Theorem 4 and conclude from it that the total cost of the searches is

$$O\left(\log(n+m) + \sum_{i=0}^{m-1} \log(p_{i+1} - p_i + 1)\right)$$

where the p_i's are defined as in the proof of Theorem 7. Hence the total running time of our algorithm is

$$O\left(\log(n+m) + \sum_{i=0}^{m-1} \log(p_{i+1} - p_i + 1)\right) = O(\log(\tbinom{n+m}{m}))$$

by the argument in the proof of Theorem 7.

We conclude this section with the remark that the very frequent operation of updating a master file against a file of updates is subsumed in Theorems 7 and 7′ and hence level-linked (a, b)-trees, (a, b)-trees with fingers support optimal update.

6. Conclusion

We carried out a detailed sequence of operations analysis of (a, b)-trees in the case $b \geqq 2a$. Our analysis shows that weak B-trees are superior to ordinary B-trees $(b = 2a - 1)$ in two areas:

a) Concurrent usage of trees: in weak B-trees rebalancing operations are concentrated near the leaves even in the presence of insertions and deletions. Thus, a high degree of parallelism is guaranteed.

b) Finger searches: level-linked weak B-trees support finger searches. Although finger searches are usually more efficient (in particular if there is locality of reference) than ordinary searches, total search time still dominates the cost of sequences of operations. Level-linked weak B-trees allow the efficient realization of many set operations. In particular, they support optimal update of a master file against a file of updates.

Acknowledgements: We wish to thank colleagues for raising questions that stimulated this research. On October 29, 1979, Mehlhorn presented the results of [4] in Zürich. Ed McCreight was in the audience and asked, "Is a similar result true for 2-3 trees?" The counterexample shown in Fig. 3 was described. He then asked, "How about 10-50 trees?" Also in October 1979, Huddleston was discussing some extensions of [4] with George Lueker, and George raised similar questions. We also want to thank the referees for some very constructive remarks which led to the formulation of Theorem 7'.

References

1. Bayer, R.: Symmetric Binary B-trees: Data structures and maintenance algorithms. Acta Informat. **1**, 290–306 (1972)
2. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indizes. Acta Informat. **1**, 173–189 (1972)
3. Bayer, R., Schkolnik, M.: Concurrency of operations on B-trees. Acta Informat. **9**, 1–21 (1977)
4. Blum, N., Mehlhorn, K.: On the average number of rebalancing steps in weight-balanced trees. Theor. Comput. Sci. **11**, 303–320 (1980)
5. Brown, M.R., Tarjan, R.E.: Design and analysis of a data structure for representing sorted lists. SIAM J. Comput. **9**, 594–614 (1980)
6. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. Proc. 19th Annual Symposium on Foundations of Computer Science. Ann Arbor: IEEE Computer Socienty, pp. 8–21 (1978)
7. Guibas, L., McCreight, E., Plass, M., Roberts, J.: A new representation for linear lists. 9th ACM Symposium on Theory of Computing Boulder, pp. 49–60 (1977)
8. Huddleston, S.: Robust balancing in B-trees. PhD Dissertation, Computer Science Department, University of Washington, Seattle, 1981
9. Huddleston, S., Mehlhorn, K.: Robust balancing in B-trees. 5th GI-Conference on Theoretical Informatics 1981, Karlsruhe, LNCS **104**, 234–244 (1981)
10. Maier, D., Salveter, S.C.: Hysterical B-trees. Technical Report 79/007, Dept. of Computer Science, State University of New York at Stony Brook, November 1979
11. Mehlhorn, K.: Effiziente Algorithmen. Teubner-Verlag, Studienbücher Informatik 1977
12. Mehlhorn, K.: Sorting presorted files. 4th GI-Conference on Theoretical Computer Science 1979, Aachen, Lecture Notes in Computer Science Vol. 67, pp. 199–212. Berlin-Heidelberg-New York: Springer 1979
13. Mehlhorn, K.: Searching, sorting and information theory. MFCS 79, Lecture Notes in Computer Science Vol. 74, pp. 131–145. Berlin-Heidelberg-New York: Springer 1981
14. Willard, D.E.: The super-B-tree algorithm. Harvard Aiken Computation Laboratory Report TR-03-79

Received January 12, 1981