Acta Informatica 21, 101-112 (1984)

© Springer-Verlag 1984

Maintaining Order in a Generalized Linked List

Athanasios K. Tsakalidis

Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes,
D-6600 Saarbrücken (Fed. Rep.)

Summary. We give a representation for linked lists which allows to efficiently *insert* and *delete* objects in the list and to quickly determine the *order* of two list elements. The basic data structure, called an *indexed* $BB[\alpha]$-tree, allows to do n insertions and deletions in $O(n\log n)$ steps and determine the order in constant time, assuming that the locations of the elements worked at are given. The improved algorithm does n insertions and deletions in $O(n)$ steps and determines the order in constant time. An application of this provides an algorithm which determines *the ancestor relationship* of two given nodes in a dynamic tree structure of bounded degree in time $O(1)$ and performs n arbitrary insertions and deletions at given positions in time $O(n)$ using *linear space.*

1. Introduction

We study the problem of performing a sequence of the following operations on a list L:

1. Insert(x, y): insert x immediately after y in the list L.
2. Delete(x): delete x from the list L.
3. Compare(x, y): return true iff x occurs before y in L.

These operations must be done on line; that is, each *compare* instruction must return a value before any further instructions are given.

We assume that the locations of the elements x, y are given. P. Dietz in [2] explores this problem and gives a data structure, called an *indexed* 2-3 tree which can perform n insertions in $O(n\log n)$ total running time and an individual comparison in $O(1)$ time.

Keeping the comparison in $O(1)$ time Dietz notes the following open problems:

1. A generalization of his algorithm which allows n insertions and deletions of arbitrary elements in $O(n\log n)$ total running time.

2. A modification of the algorithm which allows an individual insertion to be performed in $O(\log|L|)$ steps.

Dietz's data structure does not support insertions and deletions. In fact, a sequence of n insertions and deletions may have cost $\Theta(n^2)$.

We call a linked list *generalized*, if it can support efficiently insertions, deletions and comparisons.

A very simple algorithm would be to use the usual representation for linked lists. An insertion or deletion would take constant time, but comparisons could take up $O(|L|)$ time. If the list items are stored as leaves in a balanced tree scheme, the list items occuring in correct order at the leaves of the tree, insertions and deletions take $O(\log|L|)$ time. Comparisons would involve tracing back up through the tree until a common node is reached; this would also take $O(\log|L|)$ steps.

Using the weak B-tree as main structure in the last algorithm as it was defined and explored from S. Huddleston and K. Mehlhorn in [3], we can have $O(n)$ total running time for n arbitrary insertions and deletions and comparisons take $O(\log|L|)$ steps.

In this paper we give a solution for the first problem based on $BB[\alpha]$-trees, because these trees according to the results of N. Blum and K. Mehlhorn in [1] show a common behavior for both arbitrary insertions and deletions. To each item of the list *we assign proper numbers* which indicate the order of the items in the list: if a occurs before b in L then a's number is less than b's. Then we can do comparisons in constant time. For insertions and deletions we must renumber the respective leaves.

The main idea is that after renumbering we assign to adjacent elements such numbers that they can absorb enough operations without causing any renumbering of their adjacent elements.

Section 2 describes *indexed* $BB[\alpha]$-trees in detail and gives some results from [1] on these trees. In Sect. 3 we give the algorithm and show that it needs time $O(n\log n)$ for n arbitrary insertions and deletions and time $O(1)$ for a comparison. In Sect. 4 we improve the complexity for n arbitrary insertions and deletions to $O(n)$; the time for a comparison remains $O(1)$. The best previous result was from P. Dietz [2] who considered insertions only. His results were a total time of $O(n\log n)$, and time $O(1)$ for a comparison; an improvement of the total time to $O(n\log^* n)$ yielded a comparison time of $O(\log^* n)$.

Section 5 first gives an application of the algorithm to the area of CAD systems for VLSI design and Text-editing. Then an algorithm is exhibited which determines *the ancestor relationship* of two given nodes in a dynamic tree structure of bounded degree in time $O(1)$ and performs n arbitrary insertions and deletions in time $O(n)$.

2. The Main Data Structure

As main data structure we use a $BB[\alpha]$-tree. These trees were defined by J. Nievergelt and E. Reingold [7]. The Theorems 1 and 2 and some definitions of this section are taken from [1].

Definition 1. Let T be a binary tree. If T is a single leaf, then the root-balance $\rho(T)$ is 1/2, otherwise we define $\rho(T)=|T_l|/|T|$, where $|T_l|$ is the number of leaves in the left subtree of T and $|T|$ is the number of leaves in tree T.

Definition 2. A binary tree T is said *to be of bounded balance* α, or in the set $BB[\alpha]$, for $0 \leqq \alpha \leqq 1/2$, if and only if:

1) $\alpha \leqq \rho(T) \leqq 1-\alpha$;

2) T is a single leaf or both subtrees *are of bounded balance* α.

BB[α]-trees are balanced by rotations and double rotations. Figure 1 show these operations, where triangles represent subtrees and cycles nodes.

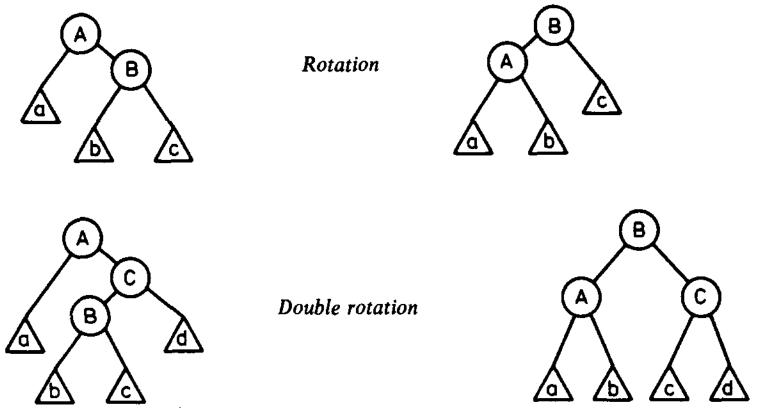


Fig. 1

Theorem 1. *There is a continuous, increasing function* c: $[0, 0.01] \to R$ *with* $c(0) = 0$, $c(0.01) = 0.0043$ *such that: for* $\alpha \in R$, $1/4 < \alpha \leqq 1 - \frac{1}{2}\sqrt{2} - c(\delta)$ *with* $0 \leqq \delta \leqq 0.01$, *and for T a binary tree with subtrees T_l and T_r such that:*

(1) *T_l and T_r are in* $BB[\alpha]$;

(2) $|T_l|/|T| < \alpha$ *and either*

(2.1) $|T_l|/(|T|-1) \geqq \alpha$ *(T is obtained by insertion of a leaf into the right subtree of T) or*

(2.2) $(|T_l|+1)/(|T|+1) \geqq \alpha$ *(T is obtained by a deletion of a leaf from the left subtree of T).*

(3) ρ_2 *is the root balance of* T_r;

we have:

(i) *if* $\rho_2 \leqq 1/(2-\alpha) + g(\alpha, \delta)$ *then a rotation rebalances the tree,*

(ii) *if* $\rho_2 > 1/(2-\alpha) + g(\alpha, \delta)$ *then a double rotation rebalances the tree, where* $g(\alpha, \delta) = \delta/([1+(1+\delta)(1-\alpha)](2-\alpha))$. □

In the next Theorem we use the following notations taken from [1, *p.* 316]. A *transaction* is either an insertion or a deletion. A *transaction goes through a node v* if v is on the path of search to the leaf to be inserted or deleted. A node takes part in a rebalancing operation, if it is one of the nodes explicitly shown in Fig. 1. A node causes a rebalancing operation if it is the root A of one of trees shown on the left side in Fig. 1.

Finally consider any sequence of transactions. We start with a tree T_0 and apply the first transaction to it. Then the tree is rebalanced by executing the balance changes and the necessary rotations or double rotations on the search path, resulting in tree T_1. The next transaction is applied to T_1, T_1 is rebalanced, ... Let $T_0, T_1, T_2, \ldots, T_m, \ldots$ be any such sequence of $BB[\alpha]$-trees.

Theorem 2. *Let* $0 \leqq \delta \leqq 0.01$, $1/4 < \alpha \leqq 1 - \frac{1}{2}\sqrt{2} - c(\delta)$ *and let* v *be a node. If*

(1) *v causes a rebalancing operation in T_m (after the transaction was applied to T_m) and*

(2) *either v took part in a rebalancing operation before or v was not a node of the initial tree T_0 and never took part in a rebalancing operation before and*

(3) *μ is the number of leaves in the subtree with root v in T_m, then at least $\lceil \delta \alpha \mu \rceil$ transactions went through v since v took part in a rebalancing operation for the last time or v was created.*

For the height $h(T)$ of T we know from [6, p. 99] *that:*

$$h(T) \leqq 1 + \frac{1}{\log(1/(1-\alpha))}(\log|T| - 1).$$

We give some definitions on a given binary tree T.

Definition 3.

1) symord(x) denotes the number assigned to x by the symmetric traversal of T (traverse the left tree, then the root and at last the right subtree).

2) A function p: *nodes U leaves* $\to \mathbb{N}_0$ labels T iff for all leaves x, y and any node v with $\text{symord}(x) < \text{symord}(v) < \text{symord}(y)$ the following holds: $p(x) \leqq p(v) < p(y)$.

3) $d(v)$ denotes the depth of a node v, where $d(\text{root}(T)) = 0$, T_x is the subtree with the node x as root, and $w(x)$ the weight of x defined as $w(x) = |T_x|$. $I(x)$ denotes the interval determined by the indices of the leaves of T_x, which are defined by the "p" function, having upper bound U_x and lower bound L_x.

Definition 4. We define c as $c = 1/\log(1/(1-\alpha))$ and $e > 2(1-\alpha)+1$. Then $h(T) \leqq c \log|T| + 1 - c$ and setting $\alpha \approx 1 - \frac{1}{2}\sqrt{2} = 0.2928$ we get $c = 2$.

Proposition 1. *The function p: nodes U leaves $\to \mathbb{N}_0$ defined as follows:*

$$p(x) = \begin{cases} p(v) & \text{if } x \text{ is the left son of } v \\ p(v) + \lfloor e 2^{c \log|T| - d(v)} \rfloor, & \text{if } x \text{ is the right son of } v. \end{cases}$$

and $p(\text{root}(T)) = 0$, makes T an indexed $BB[\alpha]$-tree.

Proof. We explore adjacent intervalls and we show that they don't overlap. Let x, y, v, w be nodes as in Fig. 2.

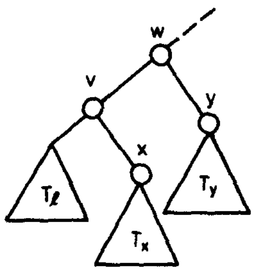


Fig. 2

Let $t=d(w)$, then we have

$$L_y = p(w) + \lfloor e 2^{c \log|T| - t} \rfloor.$$

Since $d(v)=t+1$ and $p(v)=p(w)$ we have:

$$p(x) = p(w) + \lfloor e 2^{c \log|T| - t - 1} \rfloor.$$

Let h_x be the height of T_x, then $h_x \leqq h(T) - t - 2$ and we have:

$$\begin{aligned} U_x &\leqq p(w) + \sum_{i=0}^{h_x} \lfloor e 2^{c \log|T| - t - 1 - i} \rfloor \\ &\leqq p(w) + \left\lfloor \sum_{i=1}^{h_x+1} e 2^{c \log|T| - t - i} \right\rfloor \\ &= p(w) + \lfloor e 2^{c \log|T| - t} (1 - 2^{-h_x - 1}) \rfloor < p(w) + \lfloor e 2^{c \log|T| - t} \rfloor = L_y, \end{aligned}$$

since $e 2^{c \log|T| - t} \cdot 2^{-h_x - 1} > 1$. The last inequality is valid, because

$$h_x \leqq h(T) - t - 2 \leqq c \log|T| + 1 - c - t - 2 \leqq c \log|T| - t - 1 + \log e$$

and thus $h_x + 1 \leqq c \log|T| - t + \log e$.

The same holds for the symmetric situation of Fig. 2. Hence $I(x) \cap I(y) = \emptyset$. □

3. The Algorithm and Its Complexity

We store the list elements at the leaves of an *indexed* $BB[\alpha]$-tree in the same list order. Elements can be inserted in the list and deleted from the list. We treat both insertion and deletion in a similar way. After inserting or deleting an item we trace the path backwards to the root, we change the balance of the nodes in this path, we carry out the necessary rotations or double rotations and at last we renumber the subtree whose root is the node rotated last.

Since the labeling function $p(x)$ depends on the number of leaves, we perform an operation RENUMBER(T) which renumbers the whole tree T with respect to $|T|$.

The operation RENUMBER(T) will be executed every time the current number of leaves becomes double or half as much as the number of leaves of T after the last execution of RENUMBER(T).

Now let us give the algorithm more formally and use a fixed value n_0 as an indicator for the execution of renumbering the whole tree T.

```
proc Insert$(x, y)$[Delete$(x)$]
        Let $T$ be the current tree, $v_0, v_1, \ldots, v_k$ the search path from the leaf
        worked at up to the root $v_k$ of the tree $T$. Let $b(v_i)$ be the balance of
        node $v_i$. After inserting [deleting] $x$ in $T$ we perform the following
        operations:
$a \leftarrow v_1$; $p(a) \leftarrow p(v_1)$;
for $i = 1$ to $k$ do
    change $b(v_i)$;
    if $b(v_i) \notin [\alpha, 1-\alpha]$ then [rotate or double rotate on $v_i$;
                                      $a \leftarrow v_i$; $p(a) \leftarrow p(v_i)$] fi
od;
        Let $|T_s|$ be the number of the leaves of $T$ by the last execution of
        RENUMBER$(T)$. We initialize $|T_s|$ by a fixed value $n_0$.
if $(|T| = 2|T_s|) \vee (|T| = |T_s|/2)$ then RENUMBER$(T)$
        else renumber$(a, p(a), |T_s|)$ fi
end;
proc RENUMBER$(T)$
    renumber(root$(T)$, 0, $|T|$)
end;
proc renumber$(v, i, k)$
    $p(v) \leftarrow i$;
        Let $s_l$, $s_r$ be the left and right son of $v$.
    renumber$(s_l, i, k)$;
    renumber$(s_r, i + \lfloor e 2^{c \log k - d(v)} \rfloor, k)$
end;
proc Compare$(x, y)$
    return$(p(x) < p(y))$
end;
```

Theorem 1 guarantees the correctness of insertions and deletions. On the correctness of comparisons we show the following Proposition.

Proposition 2. *Let x, y be nodes of a $BB[\alpha]$-tree T which does not stay in any ancestor relationship to each other, then during the execution of n arbitrary insertions and deletions by the above algorithm $I(x) \cap I(y) = \emptyset$.*

Proof. We explore the adjacent intervals from nodes x, y as they are given in Fig. 2.

According to Proposition 1 we have $I(x) \cap I(y) = \emptyset$ after RENUMBER(T). Let $|T| = m$ at this moment. We explore the possibility of the subtree T_x to label its elements until the next execution of RENUMBER(T), assuming that no rotation or double rotation on x or an ancestor of x occurs during that time. Note that any such rotation or double rotation would renumber all descendants of x. In particular, this implies that $d(x)$ has not changed and hence T_x can never include more than $(1-\alpha)^{t+2} 2m$, because every subtree T_v in a $BB[\alpha]$-tree T includes at most $(1-\alpha)^{d(v)} |T|$ leaves. We will show that all the elements of T_x can be numbered by integers included in the interval $I(x)$.

It suffices to show that: $(1-\alpha)^{t+2}2m<p(y)-p(x)$. We have $p(y)-p(x) =\lfloor e2^{c\log m-t}\rfloor-\lfloor e2^{c\log m-t-1}\rfloor$. Since $e2^{c\log m-t-1}-1<\lfloor e2^{c\log m-t}\rfloor-\lfloor e2^{c\log m-t-1}\rfloor$ it suffices to show that:

$$(1-\alpha)^{t+2}2m\leqq e\frac{m^c}{2^{t+1}}-1 \quad \text{for } 0\leqq t\leqq h(T)-2. \tag{1}$$

For $h(T)$ before the next execution of RENUMBER(T) we have

$$h(T)\leqq c\log(2m)+1-c=c\log m+1.$$

Setting $t=c\log m-i-2$ it suffices to show that:

$$(1-\alpha)^{c\log m-i}2m\leqq e\frac{m^c}{2^{c\log m-i-1}}-1, \tag{2}$$

for all integers i with $-1\leqq i\leqq c\log m-2$.

According to the definition of c we have $c\log(1-\alpha)=-1$. The last equality implies that $(1-\alpha)^{c\log m}=\frac{1}{m}$ and hence we can rewrite (2) as

$$\frac{2}{(1-\alpha)^i}\leqq e2^{i+1}-1. \tag{3}$$

Since $e>2(1-\alpha)+1$ for $1/4<\alpha\leqq 1-\frac{1}{2}\sqrt{2}-c(\delta)$ the inequality (3) is valid for $i\geqq -1$.

By deletions we have no problems. Rotations or double rotations renumber the elements in the local environment. □

Next we analyse the complexity of the algorithm. We tacitly assume that operations on the indices can be done in *unit time*. This is reasonable because each index is less than $2e|L|^c$.

Proposition 3. *The algorithm performs n arbitrary insertions and deletions in an initially empty linked list in $O(n\log n)$ time. The order of two given elements can be determined in $O(1)$ steps.*

Proof. The total running time consists of:

A: the total time needed for the balance changes and the rotations or double rotations.

B: the total time needed for the execution of renumber$(a,p(a),|T_s|)$.

C: the total time needed for the execution of RENUMBER(T).

Since every balance change, rotation or double rotation needs $O(1)$ time units, we have for n operations: $A=O(n\log n)$.

Let v be a node with weight $w(v)$. v causes an execution of renumber$(v,p(v),|T_s|)$, if either v was the last rebalanced node after an insertion or deletion, or v was the father of the inserted or deleted leaf in the case that none rotation is carried out. In the first case v causes $O(w(v))$ renumbering cost and in the second case $O(1)$, because at most $1/\alpha$ leaves can be renumbered by a deletion and at most 2 by an insertion. According to Theorem 2 v can cause these cost again, if $\Omega(w(v))$ operations are going through v. Thus for the first case we charge $O(1)$ renumbering cost for each operation going through v. Since a path consists of $O(\log n)$ nodes, we have $B = O(n \log n)$.

RENUMBER(T) causes cost $O(|T|)[O(|T|/2)]$ after at least $|T|$ insertions $[|T|/2$ deletions]. Thus we charge $O(1)$ renumbering cost to each operation. Hence we have $C = O(n)$.

Thus the total running time by n arbitrary insertions and deletions is $O(n \log n)$.

Comparisons are executed by comparing only the indices assigned to the elements and it needs $O(1)$ time units. □

4. Improvements to the Algorithm

In [9] an improved algorithm is given, which uses the same principle as in [2]. It reaches $O(n \log^* n)$ total running time for n arbitrary insertions and deletions. Here we use the idea of M. Overmars [8] to improve the total running time to $O(n)$; each comparison can still be performed in $O(1)$ time.

We divide the list L into *bags* of size $\lceil \log |L| \rceil$. Then we represent each bag by a leaf in an *indexed* $BB[\alpha]$-tree T keeping the same order in which the bag appears in the list L.

We use independent numbering in the $BB[\alpha]$-tree and in the bags. By the division phase, if the list includes n_0 elements, we assign to every element of the bag the following integers:

$$n_0, 2n_0, \ldots, \lceil \log n_0 \rceil n_0.$$

The size of every assigned number corresponds to the order of the element in the bag.

A procedure ORGANIZE(L) performs the division of the list L into the bags and the respective numbering in the $BB[\alpha]$-tree and the bags. ORGANIZE(L) will be executed every time the number of the elements included doubles or halves.

If a new element is inserted into a bag between x and y then it takes the number $\lceil (\text{num}(x) + \text{num}(y))/2 \rceil$, where num($x$) is the number assigned to x.

In the time between two executions of ORGANIZE(L) a bag will be kept if it includes at most $2 \log |L_s|$ elements, where $|L_s|$ is the size of the list L at the moment of the last execution of ORGANIZE(L). If a bag exceeds this size then it will be split into two bags which must be renumbered. The new bag will be inserted into the indexed $BB[\alpha]$-tree as a new leaf. Analogously, if all elements are deleted from the bag, then the respective leaf is deleted from the $BB[\alpha]$-tree.

With the above modifications, we perform the comparison of two elements x, y in two steps as follows:

First we check if the elements belong into the same bag. If so, we compare directly. If not, we compare the respective leaves in the *indexed* $BB[\alpha]$-tree.

If we equip each element in the bag with a direct pointer to the respective bag-leaf then it can be detected in unit time whether two elements belong to the same bag. The numbers of the indexed $BB[\alpha]$-tree guarantee that the second checking can be made in $O(1)$ time. Note that the numbers used by the improved algorithm are less than those used by the original one.

Now let us give the improved algorithm more formally.

```
proc INSERT($v$) [DELETE($v$)]
        The element $v$ must be inserted [deleted] in the bag $B$ of the list $L$
        between the elements $x$ and $y$;
    insert $v$ in $B$; [delete $v$ from $B$;]
    num($v$)$\leftarrow\lceil$(num($x$)+num($y$))/2$\rceil$;
    arrange the pointer of $v$ to its bag-leaf $B$;
        Let $|L_s|$ be the number of the elements of $L$ by the last execution of
        ORGANIZE($L$). We initialize $|L_s|$ by a fixed value $m$. Let $|T_s|$ be the
        respective number of the leaves of $T$. $|T_s|$ is initialized by $m/\log m$.
    if $|B|=2\lceil\log|L_s|\rceil$ then split $B$ in $B$ and $\bar{B}$ and renumber these;
        insert($\bar{B}, B$) fi; co $\bar{B}$ will be inserted in $T$ as a new bag-leaf after $B$;
    if $|B|=0$ then delete($B$) fi;
    if $(|L|=2|L_s|)\vee(|L|=|L_s|/2)$ then ORGANIZE($L$) fi
end;
proc ORGANIZE($L$)
    divide $L$ into $|L|/\log|L|$ bags and assign to the elements of each bag the
    numbers $\nu$, $2\nu$, ..., $\lceil\log\nu\rceil\nu$, with $\nu=|L|$; RENUMBER($T$);
    arrange each element-pointer to the respective bag-leaf;
end;
```

Theorem 3. *The improved algorithm performs n arbitrary insertions and deletions in an initially empty linked list L in $O(n)$ time; the order of two given list elements can be determined in $O(1)$ steps.*

Proof. The numbering on the bags assigns such numbers to the elements that after at least $\log|L_s|$ insertions two elements x, y can take the same indices. But at this moment the bag must be split and renumbered. Each insertion (except the last in a bag) and deletion in a bag can be performed in $O(1)$ time. The last insertion, which causes the splitting of a bag, causes $O(\log|L_s|)$ renumbering cost on it. Hence we charge to each insertion and deletion $O(1)$ cost for operations performed in the bag.

Every $\log|L_s|$ insertions or deletions can cause an insertion or deletion of a bag-leaf in the indexed $BB[\alpha]$-tree T. T includes $O(|L_s|/\log|L_s|)$ leaves during the division phase.

Analogously to the Proposition 3 we charge $O(1)$ renumbering cost to each transaction in T which goes through the node v.

After n operations a path in T consists of at most $O(\log n)$ nodes, and hence we charge $O(\log n)$ renumbering cost to each transaction in T. But n insertions and deletions in the list L can cause at most $O(n/\log n)$ transactions in the $BB[\alpha]$-tree T and these cause altogether $O(n)$ renumbering cost in T.

Analogously we show that the total rebalancing cost is $O(n)$. ORGANIZE(L) causes cost $O(|L|)$ and altogether we obtain for n arbitrary insertions and deletions $O(n)$ total running time of the improved algorithm, and time $O(1)$ for a comparison, since this will be performed in two steps. □

5. Applications

We first give an application of the algorithm to the area of CAD systems for VLSI design.

Th. Lengauer and K. Mehlhorn [4] described a VLSI design system which allows chip specification on a topological level (without specifying the distance between circuit components). The circuit is laid out on a grid which can change in size dynamically. Here the elements of the ordered set are the grid lines (in the x-resp. y-direction), and the order is given implicitly by the sequence in which they appear in the plane.

Inserting new layout components may require the insertion of new grid lines. If during the design process components are taken out of the circuit such that grid lines become empty, these grid lines are deleted.

As second application of the algorithm is the numbering of the rows in *text editing*. In such a case it is recommendable to view subsets of the text as blocks of some size and to use the algorithm for the numbering of the blocks. Then a number of a row is a tuple of numbers consisting of the number of the block in the whole text and the relative order of the row in the block.

For further applications we use the idea given in [2] to determine the *ancestor relationship* of two given nodes in a dynamic tree structure.

Our algorithm can in addition efficiently perform the deletion of a node from the tree structure.

We recursively define two traversal methods of an ordered tree as follows:

Preorder traversal:

(1) Visit the root r and print it.

(2) Do a preorder traversal of the subtrees rooted at r's sons, going from left to right.

Postorder traversal: Do (2) with postorder traversal, then (1).

We explore a tree structure where the order of the nodes according to the traversals above is known. Then we can determine the ancestor relationship of two nodes x and y as follows (see [2]):

Proposition 4. *A node x is an ancestor of y in a tree structure iff x occurs before y in the preorder traversal and after y in the postorder traversal.*

Proof. Let x be an ancestor of y, then x appears before y in the preorder traversal and after y in the postorder traversal. If x occurs before y in the preorder traversal then x is either an ancestor of y (case A) or x, y have at least a common ancestor which is different from x and y (case B).

Let v be the common ancestor with maximal depth and let $v_1, v_2, \ldots, v_t$ be its sons. Then x must belong to a subtree T_{v_i} and y to a subtree T_{v_j} with $i<j$.

But if in addition x occurs after y in the postorder traversal then $i>j$ in case B. Thus case B cannot apply, and case A is our desired result. □

In a tree structure we want to perform the following operations on the nodes:

1. Insert-Node(x, y): Insert a new node x as the rightmost son of y.
2. Delete-Node(x): Delete the node x from the tree. The sons of x will be direct sons of father(x).
3. Pre(x, y): Return true iff x occurs before y in a preorder traversal of the tree.
4. Post(x, y): Return true iff x occurs before y in a postorder traversal of the tree.
5. Ancestor(x, y): Return true iff x is an ancestor of y in the tree.

These operations will be performed online, i.e. each operation is completely executed before the next operation is handled.

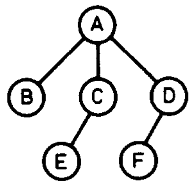
We use the idea of [2] and we have the following:

Theorem 4. *If the positions of the nodes worked at are given, then we can perform a sequence of n arbitrary operations Insert-Node(x, y) and Delete-Node(x) in an initially empty tree structure of bounded degree in time $O(n)$, and each Operation Pre(x, y), Post(x, y) and Ancestor(x, y) needs time $O(1)$.*

Proof. In addition to the usual implementation of the tree structure we use a linked list K, which combines the Informations about the preorder and postorder traversal. Each node A in the tree will be represented by two elements A_1 and A_2 in the list K.

The order of the element A_1 [A_2] corresponds to the time instant, at which the node A is visited for the first [last] time by a traversal of the tree (if the node B is visited once then B_1 and B_2 are adjacent in the list K).

We explain the structure of K in the following example:



List K:

$A_1 \leftarrow B_1 \leftarrow B_2 \leftarrow C_1 \leftarrow E_1 \leftarrow E_2 \leftarrow C_2 \leftarrow D_1 \leftarrow F_1 \leftarrow F_2 \leftarrow D_2 \leftarrow A_2$.

In our implementation we let every node A have a pointer to the elements A_1 and A_2. The list K will be implemented as in §4. Then it holds:

Pre(x, y) is *true* iff x_1 occurs before y_1 in K.

Post(x, y) is *true* iff x_2 occurs before y_2 in K.

In addition to the operations in the tree structure for the insertion or deletion of a node we also perform the following operations in the list K.

For Insert-Node(x, y) we insert the pair $x_1 \leftarrow x_2$ before y_2, and for Delete-Node(x) we only delete the elements x_1 and x_2 from the list K.

If the respective Positions of the nodes x, y are given, then the operation Insert-Node(x, y) for arbitrary tree structure as well as the operation Delete-Node(x) for the tree structures of bounded degree only can be performed in $O(1)$ time.

The total running time for the execution of n arbitrary operations Insert-Node(x, y) and Delete-Node(x) in the tree structure of bounded degree is $O(n)$ and in the list K $O(n)$, too, according to Theorem 3. The latter running time is independent from the degree of the tree structure.

According to Theorem 3, the operations Pre(x, y) and Post(x, y) can be performed in time $O(1)$; hence the operation Ancestor(x, y), too, because

Ancestor(x, y) = true iff Pre(x, y) = true and Post(x, y) = false
(see Proposition 4). $\square$

To our knowledge, the best previous algorithm for determining the ancestor relationship of two nodes was from D. Maier [5] with time $O((\log n/\log\log n)^{1/2})$ and from P. Dietz [2] with time $O(1)$ $[O(\log^* n)]$ and total running time $O(n \log n)$ $[O(n \log^* n)]$ for n insertions; however, the data structure of [5] does not use linear space and of [2] can not support efficient deletions.

Acknowledgment. The Author would like to thank Prof. K. Mehlhorn for many helpful suggestions in the presentation of this result.

References

1. Blum, N., Mehlhorn, K.: On the average number of rebalancing operations in weight-balanced trees. Theor. Comput. Sci. **11**, 303–320 (1980)
2. Dietz, P.: Maintaining order in a linked list. 14th annual ACM Symposium on Theory of Computing, pp. 122–127, 1982
3. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Informat. **17**, 157–184 (1982)
4. Lengauer, Th., Mehlhorn, K.: HILL-Hierarchical layout language, A CAD-system for VLSI-design. Technischer Bericht A 82/10, Universität des Saarlandes, 1982
5. Maier, D.: An efficient method for storing ancestor information in trees. SIAM J. Comput. **8**, 599–618 (1979)
6. Mehlhorn, K.: Effiziente Algorithmen. Stuttgart: Teubner 1977
7. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. SIAM J. Comput. **2**, 33–43 (1973)
8. Overmars, M.: A $O(1)$ average time update scheme for balanced search trees. Bull. EATCS, pp. 27–29 (1982)
9. Tsakalidis, A.: Maintaining order in a generalized linked list. 6th GI-Conference on Theoretical Computer Science, Dortmund LNCS, Vol. 145, pp. 343–352, 1983

Received May 2, 1983/November 29, 1983