MATHEMATICAL CENTRE TRACTS 108 (1979), 71-96

DYNAMIC DATA STRUCTURES

K. MEHLHORN

University of Saarland, Saarbrücken, W. Germany

The organization and manipulation of large sets of data is one of the central problems of computer science. In commercial computing centers about 1/3 of the total computing time is spent on searching and sorting. Sets of data are often dynamic in a twofold sense:

1) the set itself changes by inserting elements into it and deleting elements from it.
2) the access behavior of the users changes, i.e. the points of interest in the file change.

Let us look at an example: the set of books in a library. This set changes by the acquisition of new books (INSERT) and by discarding obsolete books (DELETE). Also the books are charged out with different probabilities (and these probabilities differ drastically from book to book). Furthermore, the access probabilities vary over time, as reading habits change. This fact could easily be accomodated for by counting the number of accesses to each book. It is also conceivable that access probabilities can change drastically sometimes. Consider a university library. Whenever a new term starts there will be a rush for the standard text books. Also, it should be possible to treat newly acquired books in different ways: the librarian might want to make a guess at the importance of a new book. In conventional libraries this is done by putting some new releases at a special shelf near the entrance door.

We propose the following definitions to model this situation.
Given is a subset $S = \{B_1,\dots,B_n\}$ of an ordered universe U. With every element $B_i \in S$ we associate a weight (= access frequency) $p_i \in \mathbb{N}$. The basic operations are ($d \in \mathbb{Z}$, $p \in \mathbb{N}$)

Member (X,S,d)	<u>if</u> $X \in S$ <u>then</u> return the information associated with X and change the weight of X by d <u>else</u> say "no"
Insert (X,S,p)	$S \leftarrow S \cup \{X\}$; the initial weight of X is p
Delete (X,S)	$S \leftarrow S - \{X\}$.

<u>REMARK</u>. We do not assume that d is specified when the operation MEMBER (X,S,d) is initiated. The case that d is a function of the old weight is also conceivable.

In this series of lectures we shall study data structures which support the three operations above efficiently. Since this is a formidable task, we proceed in five stages.

<u>STAGE 1</u>. The uniform problem: (= first kind of dynamics), i.e. all weights are equal to 1. In particular, $d = 0$ in the Member instruction and $p = 1$ in the Insert instruction. This is the classic *dictionary* problem. Many solutions to this problem are known (AVL-trees, 2-3 trees, HB-trees,...). We will treat weight-balanced trees (Nievergelt and Reingold).

<u>STAGE 2</u>. The nonuniform static case; the initial weights p_i are nonequal, no Insert and Delete instructions are allowed, and $d = 0$ in Member instructions. We will discuss how to construct *Binary Search Trees* and how to estimate their behavior.

<u>STAGE 3</u>. The nonuniform dynamic case I; this is as in stage 2, but we allow $d = \pm 1$ in Member instructions. In stage 3 the second kind of dynamics comes into play. However, access frequencies may only change slowly. We propose Dynamic Binary Search Trees (D-trees) as a solution to this problem. D-trees will be applied to digital search trees (TRIES).

<u>STAGE 4</u>. The uniform problem with the additional instructions Concatenate and Split

Concatenate (S_1,S_2,S_3)	$S_3 \leftarrow S_1 \cup S_2$; this operation is only applicable if $\max S_1 < \min S_2$
Split (S,a,S_1,S_2)	$S_1 \leftarrow \{X \in S;\ X \le a\}$ $S_2 \leftarrow \{X \in S;\ X > a\}$

The sets S_1, S_2 (resp. S) cease to exist after execution of Concatenate (resp. Split).

STAGE 5. The nonuniform dynamic case II. The full repertoire of Member, I sert and Delete operations is allowed. No restriction on d and p is place We will show how to extend D-trees in order to cope with the full problem

Finally we describe an application to sorting presorted files. Stage 1 and 3 will be discussed in some detail, only brief accounts are given for the others.

STAGE 1: Weight Balanced Trees

In a binary tree a node has either two sons or no sons at all. Nodes with no sons are called leaves and are drawn as rectangular boxes in subsequen figures. Non-leaf nodes (internal nodes) are drawn as circles and subtree are drawn as triangles.

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

For the remainder of the paper α is a fixed constant, $0 \le \alpha \le 1/2$.

DEFINITION. A binary tree T is said to be of bounded balance α, or in th set BB[α], if and only if

1. $\alpha \le \rho(T) \le 1-\alpha$
2. T is a single leaf or both subtrees are of bounded balance

There are two ways of storing an ordered set $S = \{B_1, \ldots, B_n\}$ in a binary tree T.

Leaf-oriented storage organization. The tree T has $|S|$ leaves. The leaves are labelled from left to right by the elements of S. An internal node v labelled by the largest element in the left subtree of v. Label B_i in internal node v corresponds to the query:

```
if X ≤ B_i      then goto root of left subtree
                else goto root of right subtree.
```

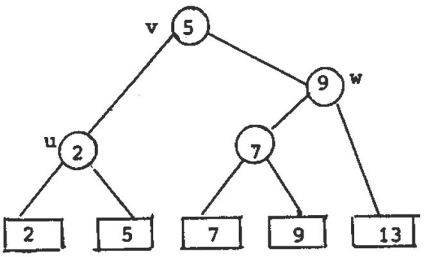
The leaf labelled B_i will be reached with all search arguments X such that $B_{i-1} < X \le B_i$, if $i < n$, and $B_{n-1} < X$ if $i = n$.

<u>Node-oriented storage organization</u>. The tree T has $|S|$ internal nodes. The internal nodes are labelled from left to right by the elements of S. Label B_i in internal node v corresponds to the query:

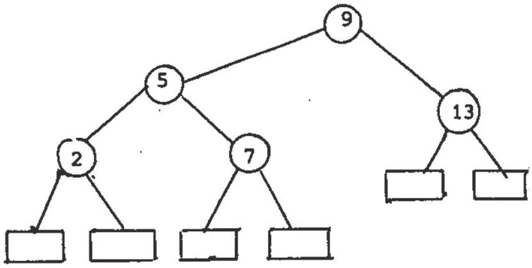
```
case  X ?  B_i  in
    < : goto root of left subtree
    = : found
    > : goto root of right subtree
```

Leaves correspond to unsuccessful searches in this case.

<u>EXAMPLE</u>. S = {2,5,7,9,13}



Leaf-oriented



Node-oriented

We search in a binary tree by comparing the search argument X with the query in the root and then taking the appropiate action (= go to left subtree,...). In our example the second leaf of the node-oriented tree will be reached with all X such that $2 < X < 5$. In the leaf-oriented tree nodes u,v,w have balance 1/2, 2/5 and 2/3 respectively.

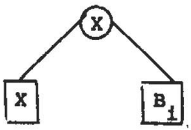
In the sequel we will always assume leaf-oriented storage organization except when explicitly stated otherwise.

<u>LEMMA 1</u>. *Let* $T \in BB[\alpha]$. *Then the depth of* T *(= length of longest path from root to leaf) is at most* $1 + (\log |T| - 1)/\log(1/(1-\alpha))$ *where* $|T|$ *is the number of leaves of* T.

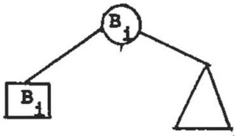
PROOF. Let $v_0, v_1, \ldots, v_d$ be a longest path from the root v_0 to a leaf v_d. Let w_i be the number of leaves in the subtree with root v_i. Then $w_{i+1} \le (1-\alpha)\ w_i$ by the definition of BB[α] tree and hence $2 \le w_{d-1} \le$ $\le (1-\alpha)^{d-1} \cdot w_0 = (1-\alpha)^{d-1}\ |T|$. Taking logarithms finishes the proof. □

EXAMPLE. $\alpha = 1-\sqrt{2}/2 = 0.2928$. Then $1/\log(1/(1-\alpha)) = 2$.

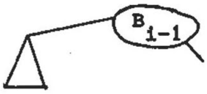
As an immediate consequence of Lemma 1 we have that MEMBER (X,S,0) instructions take time $O(\log |S|)$. Next we turn to INSERT (X,1) and DELETE (X) instructions. We first perform a search for X; this search will end in the leaf labelled B_i with $B_{i-1} < X \le B_i$. In the case of the INSERT instruction we are done if $X = B_i$. Otherwise we replace the leaf $\boxed{B_i}$ by the subtree



In the case of a DELETE instruction we are done if $X \ne B_i$. Otherwise $X = B_i$ and leaf $\boxed{B_i}$ is either the left of right son of its father. If it is the left son then we replace

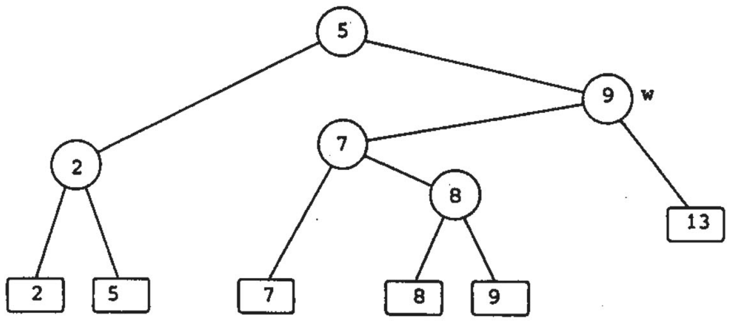


by △. If it is the right son then we replace

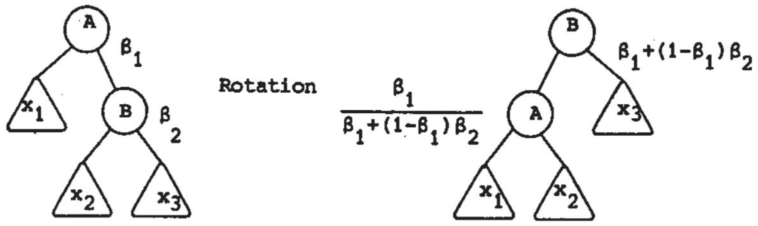


by △ and furthermore we replace the interior node labelled B_i (which necessarily lies on the path from the root to leaf $\boxed{B_i}$) by an interio node labelled B_{i-1}. One problem remains. The new tree might not be in class BB[α].

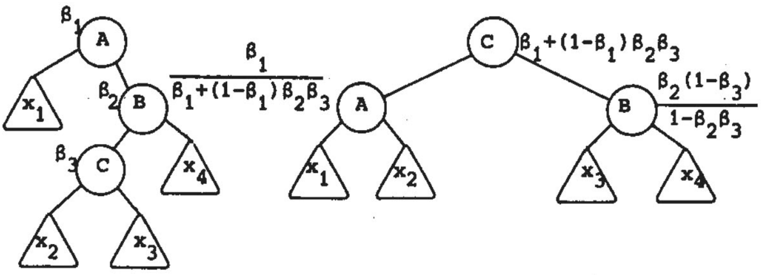
EXAMPLE. Suppose we want to insert 8 into set S. Also suppose $\alpha = 1-\sqrt{2}/2$ ≈ 0.29 (this choice of α becomes clear later on). We obtain



and node w is out of balance; $\rho(w) = 3/4 \notin [\alpha, 1-\alpha]$. In general some nodes on the path of search will be out of balance. Two operations (rotation and double-rotation) exist to restore balance. The figures show operations to the left about A. Symmetrical variants also exist.



A
β_1
x_1
B
β_2
x_2
x_3
Rotation
$\frac{\beta_1}{\beta_1+(1-\beta_1)\beta_2}$
B
$\beta_1+(1-\beta_1)\beta_2$
A
x_3
x_1
x_2



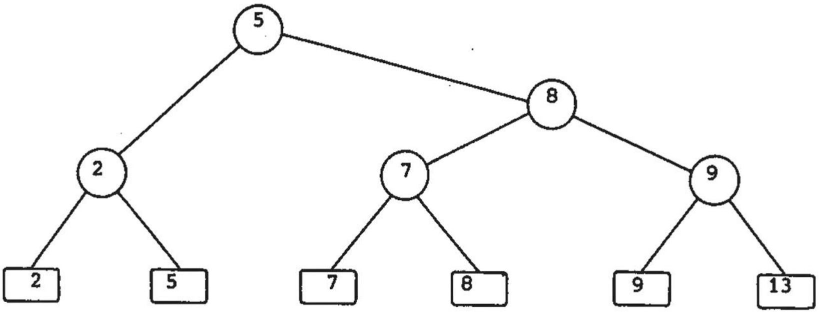
β_1
A
x_1
β_2
B
β_3
C
x_4
x_2
x_3
$\frac{\beta_1}{\beta_1+(1-\beta_1)\beta_2\beta_3}$
A
C
$\beta_1+(1-\beta_1)\beta_2\beta_3$
B
$\frac{\beta_2(1-\beta_3)}{1-\beta_2\beta_3}$
x_1
x_2
x_3
x_4

Let $\beta_1, \beta_2 (\beta_1, \beta_2, \beta_3)$ be the root-balances of nodes A,B(A,B,C) before the rotation (double-rotation). Then the balances of these nodes after the rotation (double-rotation) are as given in the figure above. Consider the case of a rotation. Let x_1, x_2, x_3 denote the number of leaves of the various subtrees shown. Then $\beta_1 = x_1/(x_1+x_2+x_3)$ and $\beta_2 = x_2/(x_2+x_3)$. The balance of node B after the rotation is

$$(x_1+x_2)/(x_1+x_2+x_3) = \beta_1 + \beta_2(x_2+x_3)/(x_1+x_2+x_3)$$
$$= \beta_1 + \beta_2(1-\beta_1).$$

The other nodes are treated analogously.

EXAMPLE CONTINUED. A double rotation about w yields



and this tree is in $BB[1-\sqrt{2}/2]$.

In general a tree is rebalanced by walking back from the (inserted or deleted) leaf to the root and performing rotations and double rotations as necessary. An exact statement can be found in

LEMMA 2. (Blum and Mehlhorn). *Let* $0 \le \delta \le 0.01$. *Then there exists a monotonically increasing function* c *with* $c(0) = 0$, $c(0.01) = 0.0045$ *such that: if* $1/4 < \alpha < 1-\sqrt{2}/2 - c(\delta)$ *then rotations and double rotations suffice to rebalance a* $BB[\alpha]$*-tree upon the insertion or deletion of a leaf.*
More precisely: walk back from the inserted or deleted leaf to the root. Say we reach node A and all subtrees below A are restored to be in $BB[\alpha]$.

CASE 1. $\rho(A) \in [\alpha,1-\alpha]$; proceed to the father of A.

CASE 2. $\rho(A) < \alpha$; let B the right son of A. If $\rho(B) < 1/(2-\alpha) + \delta/([1+(1+\delta)(1-\alpha)](2-\alpha))$ then a rotation else a double rotation rebalances the tree, i.e.

$$\rho'(A),\rho'(B),\rho'(C) \in [(1+\delta)\alpha,1-(1-\delta)\alpha].$$

Here ρ' denotes the balance after the rotation or double-rotation. Proceed to the father of A.

CASE 3. $\rho(A) > 1 - \alpha$; symmetric to case 2.

PROOF. The lengthy but simple proof can be found in Blum and Mehlhorn. □

Lemma 2 with $\delta = 0$ shows that rotations and double-rotations along the path of search suffice to rebalance a BB[α] tree after an insertion or deletion. We obtain:

THEOREM 1 (Nievergelt and Reingold, Blum and Mehlhorn). *BB[α]-trees ($2/11 < \alpha \le 1 - \sqrt{2}/2$) support the instructions* MEMBER, INSERT *and* DELETE *with processing time* $O(\log |S|)$ *per instruction.*

REMARK. Theorem 1 was first stated by Nievergelt and Reingold. The first complete proof is due to Blum and Mehlhorn.

Note that up to $O(\log |S|)$ rebalancing operations (= rotations, double rotations) may be required after a single insertion or deletion. Experiments show that on the average a constant number suffices. (Table 1 shows the findings of Baer and Schwab.) It has been a long-standing open problem whether this could actually be proven. The answer is theorem 2.

α	depth	average path length	rebalancing operations
$1 - \sqrt{2}/2$	12	9.26	426
0.25	14	9.46	206
0	22	12.14	0

TABLE I. (Baer and Schwab.) 1000 random insertions into an initially empty tree were performed for different values of α. The depth, average path length (see stage 2) of the resulting tree and total number of rebalancing operations are shown.

THEOREM 2 (Blum and Mehlhorn). *Let* $2/11 < \alpha < 1 - \sqrt{2}/2 - c(\delta)$, $0 \le \delta \le 0.01$ *and* c *be defined as in Lemma* 2. *Then there is a constant* d *such that:* d.m *rebalancing operation suffice to perform an arbitrary sequence of* m *insertions and deletions on an initially empty* BB[α]*-tree.*
($d \le \min\{k - 1 + 3(1-\alpha)^k/\delta\alpha^2;\ k \in \mathbb{N},\ k \ge 12\}$).

REMARK. For $\alpha = 1/4$, $\delta = 0.01$ we obtain $d \le 27$. There is certainly room for improvement.

The proof of theorem 2 relies upon lemma 2. The key observation is that the root-balances of nodes A,B,C after a rebalancing operation will be quite a distance (at least $\delta\alpha$) away from the critical values α and $1 - \alpha$. Hence a large number of searches (about $\delta\alpha n$ where n is the current number of leaves below such a node) can go through such a node without it being balanced again. Proper counting of rebalancing operations and of the number of insertions and deletions gives the desired result. The details can be found in Blum and Mehlhorn.

STAGE 2: BINARY SEARCH TREES.

In this section we treat the non-uniform static case; i.e. the initial weights p_i are non-equal, no Insert and Delete instructions are allowed and $d = 0$ in Member instruction. The weights p_i give rise to a probability distribution in a natural way: $\beta_i \leftarrow p_i/W$ where $W = \Sigma p_i$. In order to cope with leaf- and node-oriented storage organization we treat a slightly more general problem.

Given is a set $S = \{B_1, \dots, B_n\}$ and 2n+1 probabilities $\alpha_0, \beta_1, \alpha_1, \dots, \alpha_{n-1}, \beta_n, \alpha_n : \alpha_j \geq 0,\ \beta_i \geq 0$ and $\Sigma\beta_i + \Sigma\alpha_j = 1$. Here β_i is the probability of accessing B_i and α_j is the probability of accessing elements X with $B_j < X < B_{j+1}$. Let T be a node-oriented search tree for set S. (If we want to talk about leaf-oriented search trees then we should set $\beta_i = 0$ for all i and α_j = probability of access of B_j). Let b_i be the depth of node B_i in T and let a_j be the depth of leaf (B_j, B_{j+1}) in tree T. Then

$$P = \sum_{i=1}^{n} \beta_i (b_i + 1) + \sum_{j=0}^{n} \alpha_j\, a_j$$

is the weighted path length of tree T. It measures the average search time in tree T. Note that $b_i + 1$ are comparisons are required to find $X = B_i$ and a_j comparisons are required to find X with $B_j < X < B_{j+1}$.

<u>THEOREM 3</u> (Mehlhorn 77b). *There is a tree T such that*

$$b_i \leq \log 1/\beta_i$$

$$a_j \leq \log 1/\alpha_j + 2$$

$$P \leq H(\alpha_0, \beta_1, \dots, \beta_n, \alpha_n) + 2\Sigma\alpha_j$$

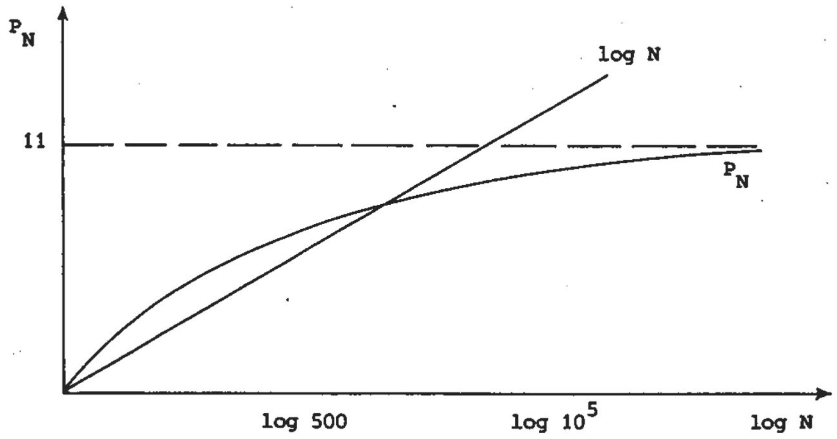
where $H = -\Sigma\beta_i \log \beta_i - \Sigma\alpha_j \log \alpha_j$ *is the entropy of the frequency distribution.*

THEOREM 4 (Bayer). *For every tree* T

$$H \le P + \Sigma\beta_i[\log e - 1 + \log (P/\Sigma\beta_i)].$$

A tree satisfying the requirements of theorem 3 can be found in only linear time (Fredman). From theorem 4 we infer that this tree is close to optimal. Theorems 3 and 4 are alphabetic versions of Shannon's noiseless coding theorem. Algorithms for the construction of optimal trees are due to Hu/Tucker and Garsia/Wachs in the leaf-oriented case (Mehlhorn/Tsagarakis show that the two algorithms are actually the same) and due to Knuth in the node-oriented case. They have time complexity $O(n \log n)$ and $O(n^2)$ respectively. Extensions to non-binary trees can be found in Itai and Altenkamp/Mehlhorn.

Theorem 4 is the most important one for what follows. It shows that no tree can have weighted path length much less than the entropy of the distribution of access probabilities and provides us with a yardstick for near optimality. We close this section with an example. Gotlieb/Walker took a text of 10^6 words and counted word frequencies. Then they constructed (nearly) optimal binary search trees for the N most common words, N = 10, 100, 1000, 10 000, 100 000. Let P_N be the weighted



path length of the tree constructed for the N most common words. The figure (due to Gotlieb and Walker) suggests that $P_N \to 11$ for $N \to \infty$. In view of theorems 3 and 4 this observation may be explained analytically. Let β_i be the probability of occurrence of the i-th most frequent word. Then (cf. Schwarz)

$$\beta_i \approx c/i^{1.12} \quad \text{where } c = 1/\sum_{i=1}^{\infty} {}^{1.12}$$

and (by a simple calculation)

$$H(\beta_1,\beta_2,\beta_3,\ldots) = -\sum_{i=1}^{\infty} \beta_i \log \beta_i \approx 10.2.$$

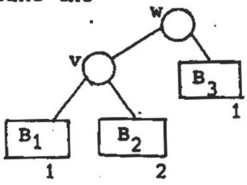
Since by theorems 3 and 4 entropy and weighted path length are closely related, this is an analytical explanation of the experiments.

STAGE 3. Dynamic Binary Search I (Mehlhorn 77c)

We are now ready to deal with the second kind of dynamics. We allow weight changes of size ±1. Let $S = \{B_1,\ldots,B_n\}$ and let $p_i \in \mathbb{N}$ be the weight of B_i. Strictly speaking, we should add a superscript t: p_i^t is then the weight of object B_i at time t.

Several solutions were proposed, notably Allan/Munro, Baer, Unterauer, Mehlhorn 77c. We describe the solution in Mehlhorn 77c, which is the only one with a proven worst case bound. The solutions proposed by Baer, Unterauer and Mehlhorn try to extend the BB[α] concept to weighted trees.

Suppose $\alpha = 1/5$ and $p_1 = 1$, $p_2 = 2$, $p_3 = 1$. Consider the following tree. We could define the



root-balances:

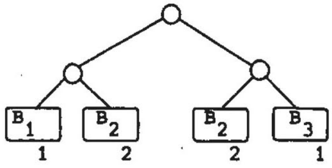
$\rho(v) = 1/(1+2) = 1/3$

$\rho(w) = (1+2)/(1+2+1) = 3/4$

Executing MEMBER $(B_2,1)$ twice increases p_2 to 4 and we have $\rho(w) = (1+4)/(1+4+1) = 5/6$. Neither a rotation or a double-rotation (not even applicable) will help us. Even worse, there is no BB[1/5] tree for weights 1,4,1. What should we do?

Baer and Unterauer suggest to give up on the strict BB[α] idea but retain the balancing operations rotation and double-rotation. Baer expresses the hope and Unterauer proves (under reasonable probability assumptions) that this will keep the tree nearly optimal on the average.

We follow a different approach. We stick to the strict BB[α] idea but give up the concept that an object $B_i \in S$ has to be represented by a single leaf of the tree. Why not represent object B_2 by 2 leaves of weight 2 each in our example? (See also van Leeuwen.) Then a double rotation helps and gives



a tree in BB[1/5]. However, we have a new problem now. There are several leaves labeled by B_2. The way out of this dilemma is the following: one of these leaves "really" represents object B_2 (the active 2-node below), the others are only around to make rebalancing always possible (the non-active 2-nodes below). In other words, the non-active 2 nodes only serve for bookkeeping purposes. We show below that it is not necessary to store them explicitly (compact dynamic trees). Of course, there is no a-priori bound on the number of times a certain object has to split. However, our knowledge of the non-weighted case (= ordinary BB[α]-trees) tells us that parts won't have to have weight less than one. It is therefore reasonable to assume that an object of weight p consists of p "atoms" of weight 1. We are now ready for the formal definition of D-trees

D-trees are an extension of BB[α]-trees. We imagine an object B_i of weight p_i to consist of p_i leaves of weight 1. A D-tree for set S is then a BB[α]-tree T with $W = p_1 + p_2 + \ldots + p_n$ leaves. The leftmost p_1 leaves are labelled by B_1, the next p_2 leaves are labelled by $B_2, \ldots$.

<u>DEFINITION.</u>

a) A leaf labelled by B_j is a j-leaf.

b) A node v of T is a j-node iff all leaves in the subtree with root v are j-leaves and v's father does not have this property.

c) A node v of T is the j-joint iff all j-leaves are descendants of v and neither of v's sons has this property.

d) Consider the j-joint v. p_j' j-leaves are to the left of v and p_j'' j-leaves are to the right of v. If $p_j' \geq p_j''$ then the j-node of minimal depth to the left of v is active, otherwise the j-node of minimal depth to the right of v is active.

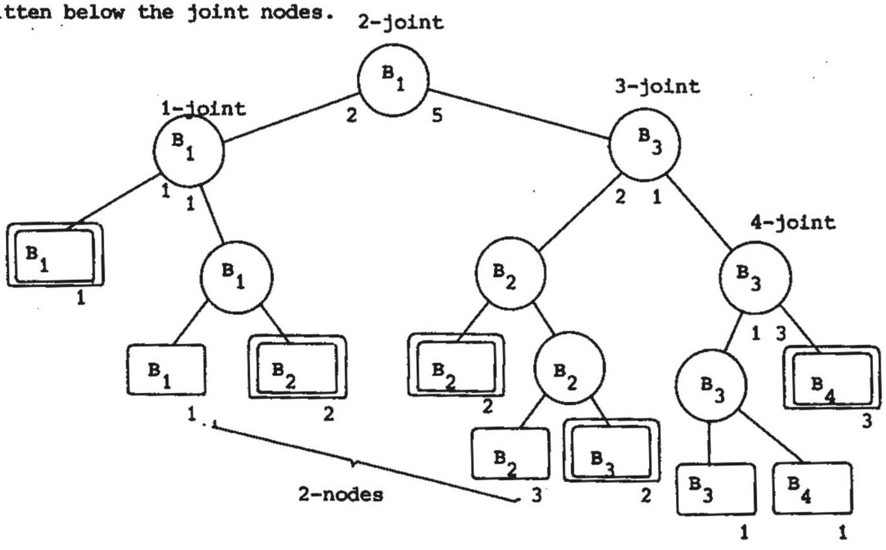
e) The thickness th(v) of a node v is the number of leaves in the subtree with root v.

Only parts of the underlying BB[α]-tree actually need to be stored, in particular all proper descendants of j-nodes can be pruned. Only their number is essential and is stored in the j-node. More precisely, a D-tree is obtained from the BB[α]-tree by

1) pruning all proper descendants of j-nodes
2) storing in each node.
   a) a query of the form "if $X \le B$ then go left else go right"
   b) the type of the node: joint node, j-node or neither of above
   c) its thickness
   d) in the case of the j-joint the number of j-leaves in its left and right subtree.

The queries are assigned in such a way as to direct a search for B_i to the active i-node. More precisely, let v be any interior node of the D-tree and let the active 1-node,..., j-node be to the left of v. Then the query "if $X \le B_j$ then go left else go right" is stored in v.

The next figure shows a D-tree for the distribution $(p_1,p_2,p_3,p_4) = (2,7,3,4)$ based on a tree in BB[1/4]. The j-nodes are indicated by squares active j-nodes by double lines, the thickness of j-nodes is written below them and the distribution of j-leaves with respect to the j-joints is written below the joint nodes.



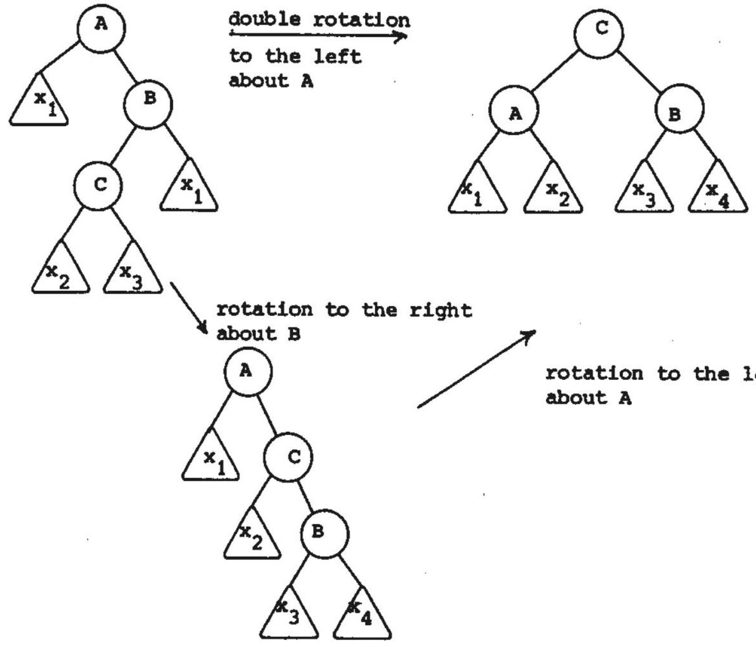
The following Lemma shows that D-trees are good search trees.

LEMMA 3. *Let* b_j *be the depth of the active* j*-node in tree* T*. Then* $b_j \le c_1 \log W/p_j + c_2$ *where* $c_1 = 1/\log(1/(1-\alpha))$, $c_2 = 1 + c_1$.

EXAMPLE. For $\alpha = 1 - \sqrt{2}/2$ we have $c_1 = 2$ and $c_3 = 3$. In the light of Theorem 4 we have that search time in D-trees is at most twice the search time in optimum trees and usually much better (cf. experimental data below).

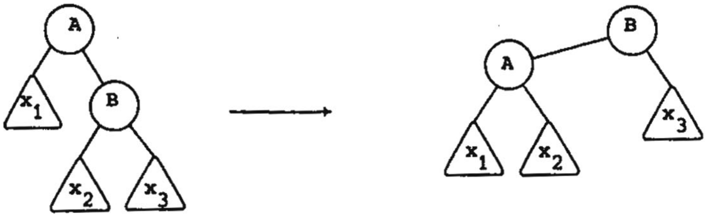
PROOF. Let v be the father of the active j-node. Then all j-leaves which are on the same side of the j-joint as the active j-node are descendants of v. Hence $th(v) \ge p_j/2$. The argument of Lemma 1 will finish the proof. □

Next we have to address the question of how to maintain D-trees. The answer is exactly as for BB[α]-trees, but be careful with the additional D-tree information. Suppose we execute a MEMBER $(B_j, \pm 1)$ instruction. The search will end in the active j-node. We have to update the thickness of all nodes on the path of search and the distribution of j-leaves with respect to the j-joint. The j-joint lies on the path of search and so this is easily done. Next we have to ascend the path of search from the active j-node to the root and perform rotations and double-rotations as required. Since a double-rotation is two rotations



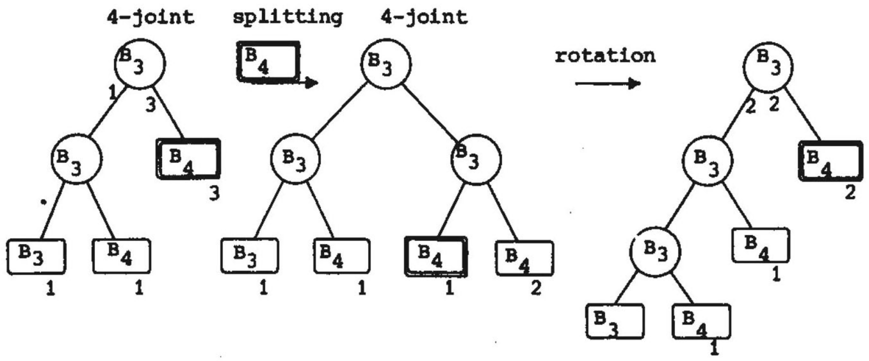
we only have to treat the case of a rotation. Let's call joint-nodes and j-nodes special nodes. If no special node is involved in the rotation then no additional actions are required. Suppose now, a special node is involved in the rotation.

CASE 1. A j-node is involved. Then we have the following picture



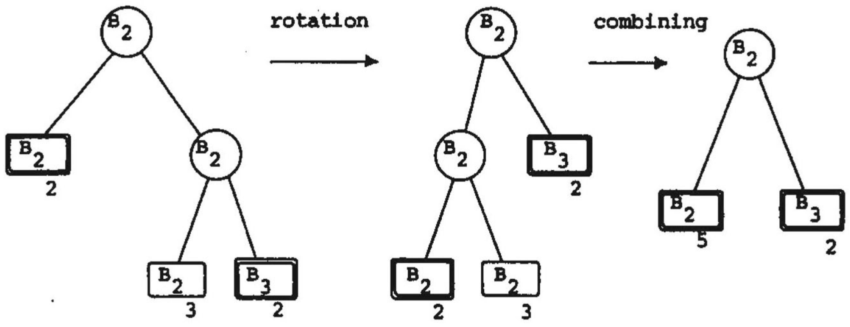
and node B is a j-node before the rotation, i.e. trees x_2 and x_3 do not exist explicitly. We create them by splitting B into two j-nodes of thickness $\lfloor th(B)/2 \rfloor$ and $\lceil th(B)/2 \rceil$ respectively. What query should we assign to B (Note that B is an interior node now)?. Suppose first that neither A nor B is the j-joint. Then A must be a left descendant of the j-joint. Otherwise x_1 can only contain j-leaves and hence A would be a j-node and hence B would not exist. So A must be a left descendant of the j-joint and hence the active j-node lies to the right of A. But then it also lies to the right of B (x_3 could be it) and thus we only have to copy the query from A into B. The discussion above also solves the case where B is the j-joint. Suppose next that A is the j-joint. Then the active j-node will be to the left of B after the split. Let Z be the nearest ancestor of A such that the left link was taken out of Z during theh search. Copy Z's query into B. Z can be found as follows: When the nodes on the path of search are stacked during the search, they are also entered into either one of two linear lists: the L-list or the R-list. The L-list contains all nodes which are left via their left links and the R-list contains all nodes which are left via their right links. Then Z is the first node on the L-list. This ends the discussion of B being a j-node.

EXAMPLE. Rotation to the left about the 4-joint.



The second possibility is that x_2 and x_3 are j-nodes and hence A is j-node after the rotation. In this case x_1 and x_2 are deleted after the rotation.

EXAMPLE. Rotation to the left about the father of the active 2-node



CASE 2. A joint node is involved, i.e. either A or B is a joint node or both. If B is a joint node then no additional actions are required. So let us consider the case that A is the j-joint. Let p_j', p_j'' be the distribution of j-leaves with respect to the j-joint A and let s be the thickness of the root of x_2. If $s \geq p_j''$ then x_3 contains no j-leaves and hence A will be the j-joint after the rotation. No action is required in this case.

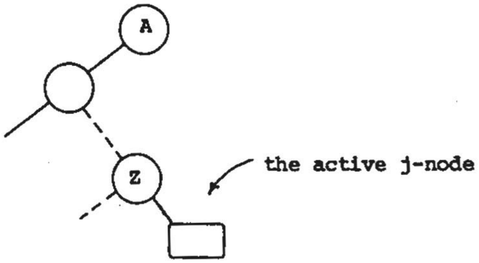
If $s < p_j''$ then B will be the j-joint after the rotation. The distribution of j-leaves with respect to B is $p_j' + s$, $p_j'' - s$.

CASE 2.1. $p_j' + s \leq p_j'' - s$. Then $p_j' \leq p_j''$ and the active j-node was to the right of A, in fact it was node x_2. Also the active j-node will be to the right of B after the rotation and it still is to the right of A. Hence we only have to copy A's query into B.

CASE 2.2. $p_j' + s > p_j'' - s$. Then the active j-node will be to the left of B after the rotation, and hence it will be node x_2.

CASE 2.2.1. $p_j' \le p_j''$. Then x_2 also was the active j-node before the rotation. No additional action is required in this case.

CASE 2.2.2. $p_j' > p_j''$. Then the active j-node was to the left of A and hence to the left of B before the rotation. In this case B's query remains unchanged, but A's query has to be changed. Suppose first that A's left son is a j-node. Then A ceases to exist after the rotation and we are done. Suppose next that A's left son is not a j-node. The next figure shows a microscopic view of tree x_1.



We only have to copy Z's query into A. Z can be found by a brute force search. Note that $th(z) \ge p_j' \ge p_j/2$. Note also that the thickness s of x_3 is less than $p_j'' \le p_j/2$. Since $s = th(x_3) \ge \alpha \cdot th(B)$ (the underlying tree is in BB[α]) and $th(B) \ge (1-\alpha) th(A)$ (a rotation to the left about A is performed) we have $s \ge \alpha(1-\alpha) th(\alpha)$ and hence $th(A) \le p_j/(2 \cdot \alpha(1-\alpha))$. The argument used in the proof of Lemma 1 shows that the depth of Z with respect to A is at most $\log(\alpha(1-\alpha))/\log(1-\alpha)$.

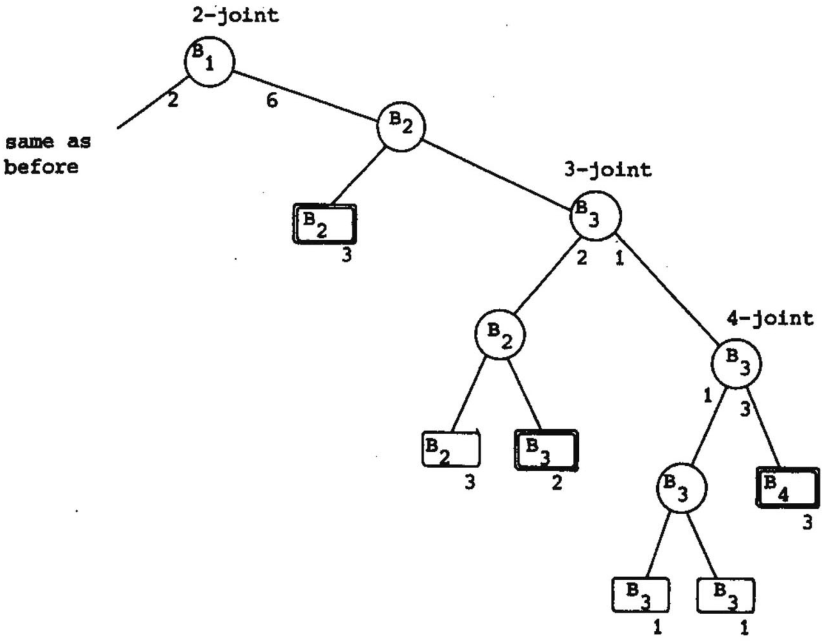
REMARKS. For $\alpha = 1 - \sqrt{2}/2$ we have $\log(\alpha \cdot (1-\alpha))/\log(1-\alpha) \approx 4.4$. Case 2.2.2 is not very likely to occur. In our simulations (several hundred thousand MEMBER (,+1) instructions) it never occurred.

We summarize the discussion in

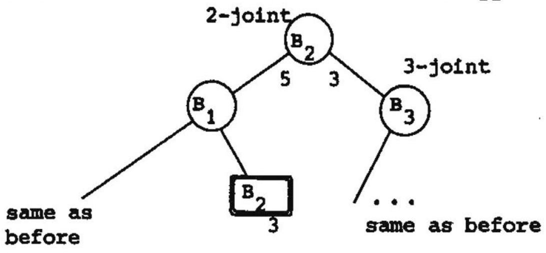
THEOREM 5 (Mehlhorn 77c). *Consider a D-tree based on a* BB[α]*-tree with* $2/11 \le \alpha \le 1 - \sqrt{2}/2$. *Let* p_i^t *be the weight of* B_i *at time* t, $1 \le i \le n$ *and let* $W^t = \sum_{i=1}^n p_i^t$. *A search for* B_i *at time* t *takes time* $c_1 \log W^t/p_i^t + c_2$.

Also a weight change by ± 1 *at time* t *takes time* $c_1 \log W^t/p_i^t + c_2$ *for some small constants* c_1 *and* c_2.

<u>EXAMPLE</u>. Suppose we want to execute a MEMBER $(B_2,+1)$ instruction. This would increase the thickness of the active j-node from 2 to 3 and move the balance parameter of the root (the active 2-joint) out of the range [1/4,3/4]. A double-rotation (to the left) about the root is required. It is simulated by a rotation to the right about the 3-joint followed by a rotation to the left about the 2-joint. The rotation about the 3-joint requires no special action since s = th(father of active 3-node) = 5 > 2 = number of 3-leaves to the left of 3-joint. We obtain



Next we have to rotate about the 2-joint. We have $p_2' = 2$, $p_2'' = 6$ and $s = 3$. and hence case 2.2.1 of the discussion above applies. We obtain



Having described the theory of D-trees to some extent the reader might be interested in experimental data. H. Reinshagen and A. del Fabro programmed D-trees and carried out the following experiments. They took an arbitrary BB[$1-\sqrt{2}/2$] tree with 200 leaves and $p_1 = \dots = p_{200} = 1$. Then they executed 30 000 MEMBER (,+1) instructions according to a fixed probability distribution (distribution I: $p_i = 100^i/(i!\ e^{100})$, distribution II: obtained by counting words starting with different 2 letter prefixes). The weighted path length of the actual D-tree and the total number of rotations and double rotations performed was recorded. The following table shows the

# of searches	$\frac{P_{actual}-P_{opt}}{P_{opt}}$ .100	# ROT	$\frac{P_{actual}-P_{opt}}{P_{opt}}$	# ROT
0	48.6	0	22.9	0
100	35.7	34	14.5	52
500	19.8	51	9.9	148
1000	11.9	58	7.6	207
5000	1.7	84	5.8	370
10000	1.7	90	5.7	420
20000	1.8	93	5.7	440
30000	1.7	96		

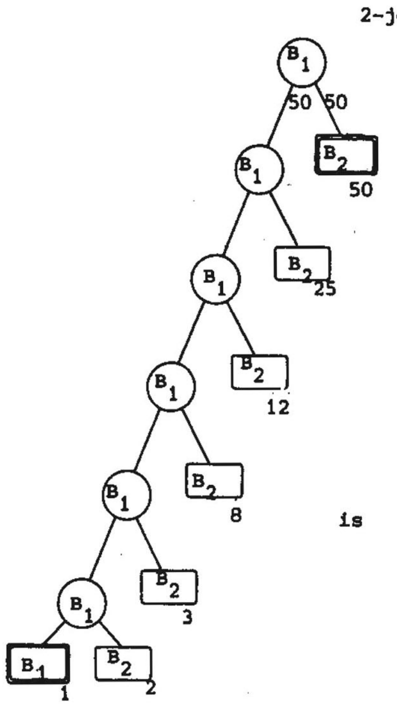
deviation (in percent) of the actual weighted path length from the weighted path length of the optimal tree for distribution I and II respectively. It also shows the number of rotations and double rotations required.

COMPACT D-TREES

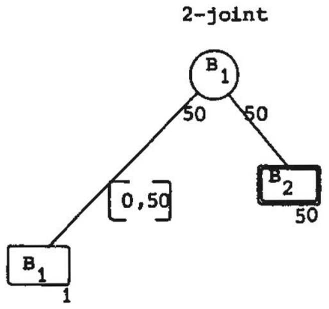
Non-active j-nodes only serve bookkeeping purposes; they permit a uniform treatment of rebalancing operations. In this section we indicate that they need not to be stored explicitly: compact D-trees. We introduce compact D-trees by way of example, the full theory can be found in [Mehlhorn 77 c].

In compact D-trees only those nodes are actually stored which are essential for the searches: the active j-nodes, the branch nodes (i.e. nodes having active descendants in both subtrees) and the joint-nodes. All other nodes are deleted, however their thickness is remembered.

Consider the following example; $p_1 = 1$, $p_2 = 100$. The compact version of the D-tree



is



The expression [0,50] on the right side of the edge from the 2-joint to the active 1-node denotes that right subtrees of that path containing a total number of 0 1-leaves and 50 2-leaves were deleted.

Compactification of (extended) D-trees to compact D-trees is a many-one mapping, i.e. in general many D-trees are represented by the same compact D-tree. The essential point is that one D-tree in the inverse image of a compact tree with respect to the compactification mapping is computed easily; in fact, reconstruction can be done locally.

Consider our example again. Say we want to expand the edge /[0,50] again. The query of the top node of the edge being B_1, we know that [0,50] represents 0 1-leaves and 50 2-leaves. The thickness of the bottom node is 1. Hence we might partition the 50 2-leaves into pieces of size 1,2,4,8,16, 19 and obtain a tree in BB[1/4]. For full details we refer the reader to [Mehlhorn 77c]. Some further compactifications are possible: it is possible to approximately reconstruct the edge labels during the search and to use height-balanced trees instead of weight-balanced trees [Del Fabro/Mehlhorn].

STAGE 4. The uniform problem with the additional instructions: Concatenate and Split. The traditional name for data structures supporting the instruction Member, Insert, Delete, Concatenate and Split is Concatenable Queue. It has long been known that height-balanced trees support the full repertoire of Concatenable Queue operations with $O(\log|S|)$ processing time per instruction. We show that weight-balanced trees also support the full repertoire with the same time bound.

LEMMA (Mehlhorn 78). $BB[\alpha]$*-trees support the full repertoire of concatenable queue instructions with* $O(\log|S|)$ *processing time per instruction.*

STAGE 5. The nonuniform dynamic case II. We finally treat the full problem: Member, Insert and Delete operations are allowed. No restriction on p and d is placed. Using the techniques developed in stage 4 we extend D-trees to cope with the full problem.

THEOREM 6 (Mehlhorn 78). *Let* $2/11 < \alpha \le 1-\sqrt{2}/2$ *and let* T *be a* D*-tree for set* $S = \{B_1,B_2,\dots,B_n\}$ *based on a* $BB[\alpha]$*-tree. Let* p_i *be the weight of* B_i *and let* $W = \Sigma p_i$.

a) *The operation Member* (B_i,S,d) *takes time* $O(\min[\log W/\min(p_i,p_i+d),n])$

b) *The operations Insert* (X,S,p) *and Delete* (B_i,S) *take time* $O(\min(\log W,n))$.

Part a) of theorem 6 says that the execution time of the instruction MEMBER (B_i,S,d) is at most proportional to the logarithm of the old access probability W/p_i or the new access probability $W(p_i+d)$. (Since a compact D-tree has depth at most n, execution time is also $O(n)$). In view of theorem 4 this is optimal up to a constant factor. Part b) is almost a corollary of part a) if one observes that either the old weight (INSERT) or the new weight (DELETE) is 0 in this case.

We are now at the end of a long journey. We finally arrived at a solution to the problem posed in the introduction. We close with a brief discussion of two applications.

AN APPLICATION TO TRIES

An alternative to searching based on key comparion is digital searching. Here a key is identified by successive identification of its component characters. One such method is the TRIE. A set of strings over some alphabet Σ is represented by its tree of prefixes. So every node of a trie corresponds to a word over $\Sigma = \{a_1,\ldots,a_p\}$.

Several implementations of tries were proposed.

1) Each node of the trie is represented by a vector of length $|\Sigma|$ (Fredkin). Identification of a character is done by indexing this vector. This method is very fast (one access per character) but it uses a large amount of storage.
2) Each node of the trie is represented by a linear list (Sussenguth). In a node w this list contains only those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is done by a linear search through the list. This method is slow (up to $|\Sigma|$ comparisons per character) but it saves storage space.
3) Each node of the trie is represented by a binary search tree (Clampton). In a node w of the trie this tree contains those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is by tree searching. This method is a compromise in speed and space requirement.

Let $S = \{B_1,\ldots,B_n\} \subseteq \Sigma^*$ be the set of keys and suppose that all keys are of equal length m (this is not essential but makes life easier), $|S| = n$. Clampton proposed to use a balanced binary search tree for each node.

EXAMPLE. $S = \{a_1^k a_j^{m-k};\ 0 \le k \le m,\ 1 \le j \le p\}$. Then $|S| = m\cdot p$ and a search for $X = a_1^{m-1} a_j$ takes time $O(m \log p) = O(|S|\cdot\log p/p)$.

We propose to use D-trees (or any other kind of nearly optimal search trees). More precisely, for $w \in \Sigma^*$ let

$$p_w = |\{B_i;\ w \text{ is prefix of } B_i\}|.$$

A node w of a TRIE is represented by a D-tree for the distribution $\{p_{wa};\ a \in \Sigma\}$. A key $B_i = a_{i1}a_{i2}\ldots a_{im}$ is identified by successively identifying the character a_{ik} in the tree corresponding to the node $a_{i1}\ldots a_{i(k-1)}$ of the tree. It takes time

$$O(c_1\cdot\log p_{a_{i1}\ldots a_{i(k-1)}}/p_{a_{i1}\ldots a_{ik}} + c_2)$$

to identify a_{ik} where c_1,c_2 only depend on the balance parameter (cf. lemma 3). Hence B_i can be identified in time

$$O(c_1 \log p_\varepsilon/p_{a_{i1}\ldots a_{im}} + c_2 m) = O(c_1 \log n + c_2 m).$$

Since log n comparisons are required in any scheme based on comparisons with binary outcome and every character of the input has to be inspected we have nearly optimal tries under implementation 3.

We use D-trees to implement the nodes of a trie because we want to deal with updates, i.e. insertions and deletions of names. Suppose we want to insert a new name B into the set S. This amounts to increase p_w by 1 for all prefixes of B. Retaining near optimality is no problem since we used D-trees to implement the nodes of a trie. Conversely, suppose we want to delete a name B from the set S. This amounts to decrease p_w by 1 for all prefixes of B. Again retaining near optimality causes no problems. We thus proved

<u>THEOREM 7</u>. *Let* S *be a set of keys of* m *characters each. If a trie is used to represent the set* S *and every node of the trie is implemented as a* D-*tree then searching for a key in* S, *inserting a new key into* S *and deleting a key from* S *can be done in time* $O(\log|S| + m)$ *and this is uptimal up to a constant factor.*

<u>EXAMPLE continued</u>. In the example above, TRIES + D-trees guarantee that searches never take more than $O(m + \log(m\cdot p))$ time units.

Note that TRIES + D-trees give execution times which are independent of $|\Sigma|$. In database applications objects are often m-tuples (e.g. cities given by geographical altitude and latitude). In these applications $|\Sigma| = \infty$ is conceivable.

AN APPLICATION TO SORTING

Consider the problem of sorting a sequence $x_n x_{n-1} \dots x_1$ by an insertion sort. Insertion sort proceeds by successively inserting x_i into its proper position. Let

$$f_i = |\{j;\ x_j < x_i \ \text{ and } \ j < i\}|.$$

When x_i is inserted into the sorted version of sequence $x_{i-1} \dots x_1$ then x_i has to be insorted after the f_i-th position of that sequence. If the sequence is presorted, i.e. $F = \Sigma f_i$ is small with respect to n^2, then the elements tend to be inserted near the front of the already sorted subsequence. Using the concepts developed in stage 3, Fredman has shown that it should be possible to sort the sequence with $O(n \log(F/n))$ comparisons. Later-on practical versions of Fredman's algorithm were developed by Guibas et al., Brown/Tarjan and Mehlhorn 79. The algorithm described by Mehlhorn is based on AVL-trees and has running time $26\ n \log F/n + 40n$ on the machine described in Mehlhorn 77a (similar to MIX). Comparing this with Quicksort's running time of $9n \log n$ on the same machine gives

$$26\ n \log F/n + 40n \le 9n \log n$$

iff

$$F \le 0.314\ n^{1.375}.$$

For presorted files the new method will be superior.

SUMMARY. New approaches to searching and sorting were discussed which exploit the fact that in some applications search requests or sequences to be sorted are non-random. More specifically a tree structure (D-trees) was presented which supports searching in, inserting into, deleting from and changing weights in a weighted set S in time optimal up to a constant factor. Also a sorting method which sorts presorted input sequences in time strictly less than n log n was presented.

REFERENCES

ALLAN, A. & I. MUNRO, *Self organizing binary search trees*, JACM, 25 (1978), pp. 526-535.

ALTENKAMP, D. & K. MEHLHORN, Codes, *unequal letter costs, unequal probabilities*, 5th ICALP, 1978, Springer Lecture Notes in Computer Science vol. 62, pp. 15-25, to appear JACM.

BAER, J.L., *Weight-balanced trees*, Proc. AFIPS, vol. 44 (1975), pp. 467-472.

BAER, J.L. & B. SCHWAB, *A comparison of tree balancing algorithms*, CACM 20 (1977), 322-330.

BAYER, P., *Improved bounds on the cost of optimal and balanced binary search trees*, techn. report, Dept. of Computer Science, MIT, 1975.

BLUM, N. & K. MEHLHORN, *On the average number of balancing operations in weight-balanced trees*, 4th GI Conference on Theoretical Computer Science, Aachen, 1979 (to appear).

BROWN, M.R. & R.E. TARJAN, *A representation for linear lists with movable fingers,* 10th ACM STOC, 1978, p. 19-29.

CLAMPTON, H.A., *Randomized binary searching with tree structures,* CACM 7, 3 (March 1964), 163-165.

DEL FABRO, A. & K. MEHLHORN, *Further compactification of D-trees,* in preparation.

FREDKIN, E., *Trie memory,* CACM 3, 9 (sept. 60), 490-499.

FREDMAN, M.L., *Two applications of a probabilistic search technique: Sorting X+Y and building balanced search trees,* Proc. 7th Annual ACM Symp. on Theory of Computing, Albuquerque, 1975, pp.240-244.

GARSIA, A.M. & M.L. WACHS, *A new algorithm for minimum cost binary trees,* SICOMP 4 (1977), 622-642.

GOTLIEB, C.C. & W.A. WALKER, *A top-down algorithm for contructing nearly optimal lexicographical trees,* in: R.C. Read (ed.), Graph Theory and Computing, Academic Press, London, 1972, pp.303-323.

GUIBAS, L.J., E.M. MCCREIGHT, M.F. PLASS, J.R. ROBERTS, *A new representation for linear lists,* 9th ACM STOC, 1977, 49-60.

HU, T.C. & A.C. TUCKER, *Optimal computer search trees and variable length alphabetic codes,* SIAM J. Applied Math. 21, 1971.

ITAI, A., *Optimal alphabetic trees,* SICOMP 5 (1976), 9-18.

KNUTH, D.E., *The art of computer programming,* vol. 3, *Sorting and searching,* Addison Wesley, 1973.

MEHLHORN, K., 77a, *Effiziente Algorithmen,* Teubner Studienbücher Informatik, Stuttgart 1977.

MEHLHORN, K., 77b, *Best possible bounds on the weighted path length of optimum binary search trees,* SICOMP 6 (1977) pp. 235-239.

MEHLHORN, K., 77c, *Dynamic binary search,* 4th Colloquium on Automata, Languages and Programming Turku, 1977, Springer Lecture Notes in Computer Science, vol. 52, pp. 323-336.

MEHLHORN, K., 78, *Arbitrary weight changes in dynamic trees,* Techn. Bericht, Universität des Saarlandes, May 1978.