Dynamic Binary Search*

Kurt Mehlhorn

Fachbereich 10 - Angewandte
Mathematik und Informatik -
der Universität des Saarlandes
D-6600 Saarbrücken

*A preliminary version of this paper was presented at the 4th Colloquium on Automata, Languages and Programming, Turku, 1977, Springer Lecture Notes, Vol. 52, 323-336

Abstract

We consider search trees under time-varying access probabilities. Let $S = \{B_1,\ldots,B_n\}$ and let p_i^t be the number of accesses to object B_i up to time t, $W^t = \sum p_i^t$. We introduce D-trees with the following properties.

1) A search for $X = B_i$ at time t takes time $O(\log W^t/p_i^t)$. This is nearly optimal.

2.) Update time after a search is at most proportional to search time, i.e. the overhead for administration is small.

I. Introduction

"One of the popular methods for retrieving information by its 'name' is to store the names in a binary tree. We are given n names $B_1, B_2, \ldots, B_n$ and 2n+1 frequencies $\beta_1, \ldots, \beta_n, \alpha_o, \ldots, \alpha_n$ with $\Sigma\beta_i + \Sigma\alpha_j = 1$. Here β_i is the frequency of encountering name B_i and α_j is the frequency of encountering a name which lies between B_j and B_{j+1}, α_o and α_n have obvious interpretations" [K 71].

A binary search tree T is a tree with n interior nodes (nodes having two sons), which we denote by circles, and n+1 leaves, which we denote by squares. The interior nodes are labelled by the B_i in increasing order from left to right and the leaves are labelled by the intervals (B_j, B_{j+1}) in increasing order from left to right. Let b_i be the distance of interior node B_i from the root and let a_i be the distance of leaf (B_j, B_{j+1}) from the root. To retrieve a name X, b_i+1 comparisons are needed if $X=B_i$ and a_i comparisons are required if $B_j<X<B_{j+1}$. Therefore we define the weighted path length of tree T as :

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

A large number of papers was written on the subject of constructing optimal or nearly optimal binary search trees [BAY, FM, GM, GW, GMS, HO, HT, K 71, L, M 75, M 77a, R].
We quote two results:

It is possible to construct a tree T, even in time linear in the number of nodes, such that

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

and

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

[FM, M 77a, M 77b]. Furthermore these bounds are almost sharp for most nodes and leaves [GMS, M 77b]. More precisely, for any $d \in \mathbb{R}$

and $h > o$ let

$$L_h = \{j;\ a_j \le (\log(1/\alpha_j)-h)/\log(2+2^{-d})\}$$

and

$$N_h = \{j;\ b_i+1 \le (\log 1/\beta_i-h-d)/\log(2+2^{-d})\}$$

Then $$\sum_{j\in L_c} \alpha_j + \sum_{i\in N_c} \beta_i \le 2^{-h}$$

i.e. only a small percentage of the nodes can be considerably higher in the tree than stated in the upper bound. These results show that the best we can expect from binary search trees is "logarithmic" behaviour.

In many applications the access frequencies are

a) not known in advance
b) changing over time

and therefore (nearly) optimal binary trees are not readily applicable. In this paper we introduce D-trees (dynamic-trees) in an attempt to resolve this difficulty and thus answer a challenge of Knuth [K 71]: "A harder problem, but perhaps solvable, is to devise an algorithm which keeps its frequency counts empirically, maintaining the tree in optimum form depending on the past history of the searches. Names occuring most frequently gradually move towards the root, etc.".

We suggest the following model. With every node B_i and leaf (B_j,B_{j+1}) we associate its frequency count p_i^t and q_j^t respectively. Here p_i^t is the number of searches for $X=B_i$ performed up to time t and q_j^t is the number of searches performed for $X\in(B_j,B_{j+1})$ performed up to time t. We use $W^t = \sum p_i^t+\sum q_j^t$ for the total number of accesses up to time t. Then $\beta_i^t = p_i^t/W^t$ $(\alpha_j^t=q_j^t/W^t)$ is the relative access frequency of node B_i (of leaf (B_j,B_{j+1})) at time t. A search for $X \in B_i$ $(X \in (B_j,B_{j+1})$ at time t increases p_i^t (q_j^t) by one. We drop the upper index t when it is clear from the context. Our tree structure exhibits the following behaviour:

1. The tree is always nearly optimal, i.e. a search for $X=B_i$ $(X\in(B_j,B_{j+1}))$ can be carried out in time $O(\log 1/\beta_i^t)$ $(O(\log 1/\alpha_j^t))$.

2. The time needed to update the tree structure is at most proportional to search time. This is achieved by restricting updating to the path from the root to the node (leaf) searched for.

3. New names can be inserted in time O(min(n, log W)).

In section II we review some facts about weight-balanced trees [NR].
In section II we introduce D-trees and show properties 1) and 2) above.
In section IV we give an alternate definition and work out some of its properties. Section V is dedicated to compact D-trees, in section VI we give some extensions. Finally, we apply D-trees to TRIES.

II. Preliminaries: Weight Balanced Trees

Nievergelt and Reingold introduced weight balanced trees (cf. [NR] and [M 77b]). We review some of their definitions and adapt them for our purposes. In a binary tree every node has either two sons or no son at all. Nodes with no sons are called leaves.

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 the number of leaves in the left subtree of T and $|T|$ is the number of leaves in tree T.

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

The depth of a tree T of bounded balance α is $O(\log|T|)$. We add a leaf to a tree T by replacing a leaf by a tree consisting of one node and two leaves. "If upon the addition of a leaf to a tree in $BB[\alpha]$ the tree becomes unbalanced relative to α, that is, some subtree of T has root-balance outside the range $[\alpha,1-\alpha]$ then that subtree can be rebalanced by a rotation or a double rotation. In Fig. 1 we have used squares to represent nodes, and triangles to represent subtrees; the root-balance is given beside each node". Symmetrical variants of the operations exist.

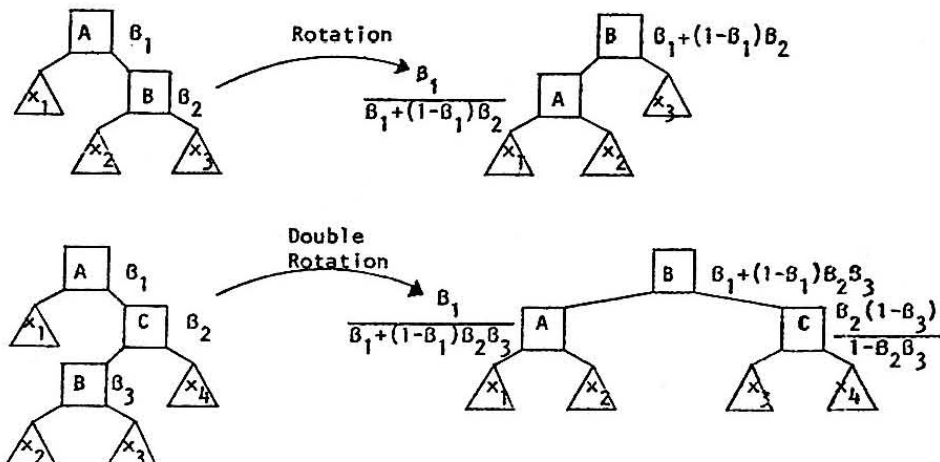


Fig. 1

Fact (Nievergelt and Reingold) : If $\alpha \le 1-\sqrt{2}/2$ and the insertion of a leaf in a tree in BB[α] causes a subtree T of that tree to have root-balance less than α, T can be rebalanced by performing one of the two transformations shown above. More precisely let β_2 denote the balance of the right subtree of T after the insertions has been done. If $\beta_2 < (1-2\alpha)/(1-\alpha)$ then a rotation will rebalance T, otherwise a double rotation will rebalance T.

The search time in weight-balanced trees is proportional to the logarithm of the number of leaves. Updating the structure upon insertion (or deletion) of a leaf can be done in time proportional to the search time. It takes constant time on the average [BM]. In the next section we adapt weight-balanced trees to binary search trees.

III. D-Trees: The basic scheme

In this chapter we restrict the discussion to the case that only searches for the leaves of a binary search tree are performed. Let q_j be the number of searches for some $X \in (B_j, B_{j+1})$, $0 \le j \le n$, performed up to now and let $W = \Sigma q_j$ be the total number of searches performed so far. We assume $q_o = q_n = 1$ to avoid some technical difficulties. This can always be achieved by adding two extra names.
From now on α is fixed, $0 < \alpha \le 1-\sqrt{2}/2$.

Let T be a tree in BB[α] with W leaves. The leaves of T are labelled from left to right according to the following rule. The first q_o leaves are labelled with $(,B_1)$, the next q_1 leaves are labelled with (B_1,B_2),... . The idea of duplicating leaves appears implicitely in [M 77a] and explicitely in [L].

Definition:

a) A node v of T is a j-node, $0 \le j \le n$, if all leaves in the subtree with root v are labelled with (B_j, B_{j+1}) and v's father does not have this property.

b) A node v of T is the j-joint, if all leaves labelled with (B_j, B_{j+1}) are descendants of v and neither of v's sons has the property.

In general, the j-joint is not a j-node. If it is, then there is just one j-node. Fig. 2 shows the relative position of j-nodes and the j-joint.

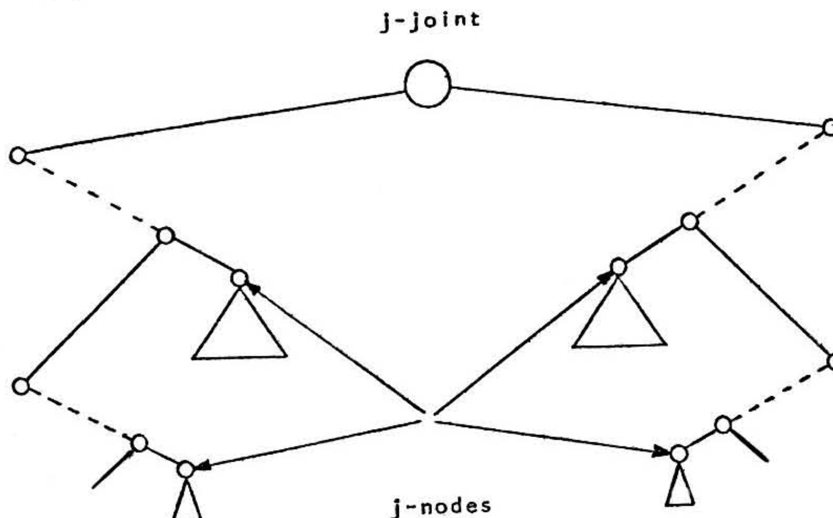


Fig. 2 : Dotted lines ... denote zero or more tree edges.

Definition:

a) Consider the j-joint v. q'_j of the leaves labelled with (B_j,B_{j+1}) are left of v and q''_j are right of v. If $q'_j \geq q''_i$ 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.

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

Lemma 1:

Let a_j be the depth of the active j-node in tree T. Then $a_j \leq c_1 \log 1/\alpha_j+c_2$, where $c_1=1/\log(1/(1-\alpha))$, $c_2=1+c_1$ and $\alpha_j = q_j/W$.

Proof:

Let v be the active j-node, a_j the depth of v and let w be the father of v. We show th(w) $\geq q_j/2$. If v is the j-joint then th(v) $= q_j$ and we are done. Otherwise v is a left or right descendant of the j-joint. Suppose v is a left descendant. Then $\geq q_j/2$ of the leaves labelled with (B_j,B_{j+1}) are in the left subtree of the j-joint. All of them are descendants of w. Hence th(w) $\geq q_j/2$. w has depth a_j-1. Since the tree T is of bounded balance

$$\text{th}(w) \leq (1-\alpha)^{a_j-1} \cdot W$$

and hence

$$\alpha_j \leq 2 \cdot (1-\alpha)^{a_j-1}$$

Taking logarithms yields the result. □

Example:

Let $\alpha = 1-\sqrt{2}/2$. Then $c_1=2$, $c_3=3$ and hence $a_j \leq 2 \log 1/\alpha_j+3$.

No analogue to lemma 1 exists if one takes height-balanced trees instead of weight-balanced trees as the underlying tree structure.

Next we have to assign queries to the nodes of tree T.
The queries are of the form

"<u>if</u> $X < B_j$ <u>then</u> go left <u>else</u> go right".

We assign queries in such a way as to direct a search for $X \in (B_j, B_{j+1})$ to the active j-node. Then lemma 1 assures us that search time is logarithmic and thus nearly optimal.

Let v be any node of T. Let j be maximal with: the active j-node is left of v. Then we assign the query "<u>if</u> $X < B_{j+1}$ <u>then</u> left <u>else</u> right" to v.

This rule assigns queries to all nodes of v. It is apparent that a search for $X \in (B_j, B_{j+1})$ is directed to the active j-node. Fig. 3 is Fig. 2 redrawn; this time the queries are shown.

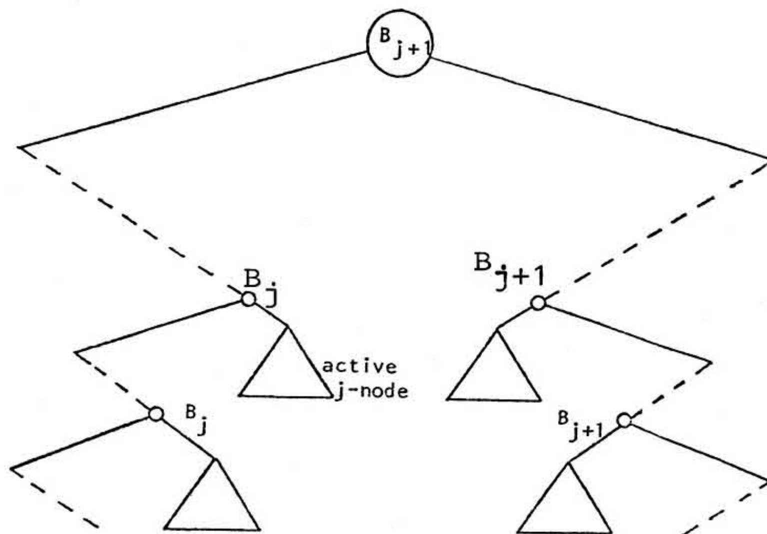


Fig. 3

Before we describe searching in and updating of our tree structure we have to say more about the information stored in the nodes.

1. All proper descendants of j-nodes are pruned.
2. In each remaining node of the underlying tree of bounded balance α we store
   a) the type of the node: joint node or j-node or neither of above
   b) its thickness

c) in the case of a joint node the number of j-leaves in its left and right subtree.

d) in the case of a j-node a pointer to the element B_j of a linear list containing the names $B_1,\dots,B_n$ in that order. An array will do if the insertion of new names is not required (cf. section VI).

Suppose now that we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. As we descend the thickness of every node encountered during the descent is increased by one. Then we ascend and rebalance the trees as in the case of trees of bounded balance α. Three new problems arise. Assume that we reach node w from its right son v. Then we searched below v. The thickness of w and v were both increased by 1. If the root balance $\rho(w) = (th(w)-th(v))/th(w)$ is less than α then we have to rebalance the tree. We treat the case of a rotation and leave the case of a double rotation for the reader

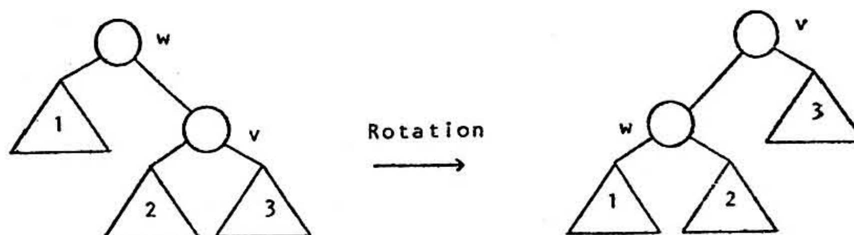


Figure 4:

Problem 1:

v is a j-node for some j. Then the trees 2 and 3 do not exist explicitely. We recreate them by splitting v into 2 j-nodes of thickness $\lfloor th(v)/2 \rfloor$ and $\lceil th(v)/2 \rceil$ respectively. Then $1/3 \le \rho(v) = \lfloor th(v)/2 \rfloor / th(v) \le 1/2$ because of $th(v) \ge 2$. A rotation will rebalance the tree.Note further that the j-node v is to the left of the j-joint (cf. figure 2). Hence the query assigned to v after the rotation should be "if $X < B_j$ ... ". The name B_j can be found using the pointer into the linear list containing all names.

Problem 2:

w is a j-node after the rotation. Then we have to combine the two trees 1 and 2 into a single node.

Problem 3:

Queries have to be changed. This can only happen if the active nodes change. This can only be the case if the active node is split (problem 1) or combined (problem 2) or if the distribution of the leaves labelled with (B_j,B_{j+1}) with respect to the j-joint changes. The first two cases were treated already. The distribution of the leaves labelled with (B_j,B_{j+1}) with respect to the j-joint can only change if the j-joint is one of the nodes involved in the transformation, i.e. it either node w or v in the figure above. If v is the j-joint then the distributions does not change. If w is the j-joint then we have to distinguish two cases.

Case 1: The tree $\triangle 3$ contains no j-node. Then all j-nodes to the right of w (the j-joint) are elements of the subtree $\triangle 2$. This can be checked by comparing the thickness of the root of $\triangle 2$ with the number stored in the j-joint w. In this case nothing has to be changed.

Case 2: The tree $\triangle 3$ contains a j-node. Then $\triangle 2$ is a j-node and its thickness is strictly smaller than the number stored in w. Then w ceases to be the j-joint, v becomes the j-joint. The query assigned to w after the rotation is "if $X < B_j$ then... ". The distribution of the j-leaves with respect to the new j-joint v can be computed from the thickness of the j-node $\triangle 2$ and the distribution stored in the old j-joint w. The distribution is stored in v and the appropiate query is assigned to v. Again the names B_j and B_{j+1} can be found using the pointer stored in the j-node $\triangle 2$.

Problem 4: Nodes change their type.

Case 1: A joint node can only change its type if it is involved in the rotation. Hence this problem was treated already in problem 3.

Case 2: A j-node can only change its type if it is involved in the rotation. Hence this problem reduces to problems 1 and 2.

We summarize the discussion. We use trees of bounded balance in order to implement search trees. The depth of the active j-node is always less than $c_1 \log 1/\alpha_j^t + c_2$ for some small constants c_1 and c_2. Here, α_j^t is the relative frequency of leaf (B_j, B_{j+1}) at time t. Updating is restricted to the path from the root to the active j-node. In each node of the path a constant amount of work is necessary. Thus searching and updating the structure can be performed in time $O(\log 1/\alpha_j)$.

Theorem 1:

Consider a D-Tree based on a BB[α] -tree with $0 < \alpha \le 1-\sqrt{2}/2$.

a) Let q_j^t be the number of searches for $X \in (B_j, B_{j+1})$, $0 \le j < n$, performed up to time t and let $W^t = \Sigma q_j^t$. Then at time t a search for $X \in (B_j, B_{j+1})$ can be executed in time $O(c_1 \log W^t/q_j^t + c_2)$ where $c_1 = 1/\log(1/1-\alpha))$ and $c_2 = 1+c_1$. The time needed to update the tree structure is proportional to the search time.

b) Let $\alpha_j^t = q_j^t/W^t$, $H = H(\alpha_o^t, \alpha_1, \dots, \alpha_n^t) = -\Sigma\alpha_j^t \log \alpha_j^t$, let P_{opt} be the weighted path length of an optimal search tree for the distribution and let P be the weighted path length of the D-tree at time t . Then

$$P \le c_1 H + c_2$$

$$\le c_1 P_{opt} + c_2$$

where c_1, c_2 are defined as in a)

c) The number of nodes in a D-tree is $O(n(1+\log W))$

Proof:

a) is immediate from lemma 1 and the preceeding discussion.

b) Let a_j^t be the depth of the active j-node at time t. Then $a_j^t \le c_1 \log 1/\alpha_j^t + c_2$ by lemma 1. Hence

$$P = \Sigma a_j^t \alpha_j^t \le \Sigma \alpha_j^t (-c_1 \log 1/\alpha_j^t) + c_2$$

$$\le c_1 \cdot H + c_2$$

$$\le c_1 \cdot P_{opt} + c_2$$

since $H \le P_{opt}$ by Gilbert and Moore [GM, K 73, M 77b].

c) Suppose there are k j-nodes $v_1,\dots,v_k$ of thickness $q^{(1)},\dots,q^{(k)}$ respectively. k_1 of these j-nodes are to the left of the j-joint. Among these v_1 has maximal depth and v_{k_1} has minimal depth. Let w be the father of v_{k_1}. Then

$$th(v_{k_1}) \;/\; th(w) \ge \alpha$$

and hence

$$th(w) \le q^{(k_1)}/\alpha \le q_j/\alpha \quad .$$

Furthermore $1 \le q^{(1)} = th(v_1)$. The depth of node v_1 in the tree with root w is bounded above by $c_1 \log (th(w)/th(v_1)) + c_2$. (Lemma 1). v_1 has depth $\ge k_1$.
Thus

$$\begin{aligned} k_1 &\le c_1 \log th(w) + c_2 \\ &\le c_1 \log q_j/\alpha + c_2. \end{aligned}$$

From this we conclude that the number of j-nodes is $\le d_1 \log q_j + d_2$ for suitable constants d_1 and d_2. Hence the total number of j-nodes, $0 \le j \le n$, is

$$\le d_1 \cdot \sum_{j=0}^{n} \log q_j + d_2(n+1)$$

$$\le d_1 \cdot n \cdot \log W + d_2(n+1)$$

Since the j-nodes are the leaves of the D-tree the total number of nodes is $O(n(1 + \log W))$. □

<u>Example:</u> Figure 5 shows a D-tree for the distribution $(q_0,q_1,q_2,q_3,q_4) = (2,3,1,8,2)$ based on a tree in BB[1/4]. The j-nodes are indicated by square boxes, the active j-nodes are underlined, the thickness of the j-nodes is written on top of them and finally the distribution of the j-leaves with respect to the j-joints is written on top of the joint nodes.

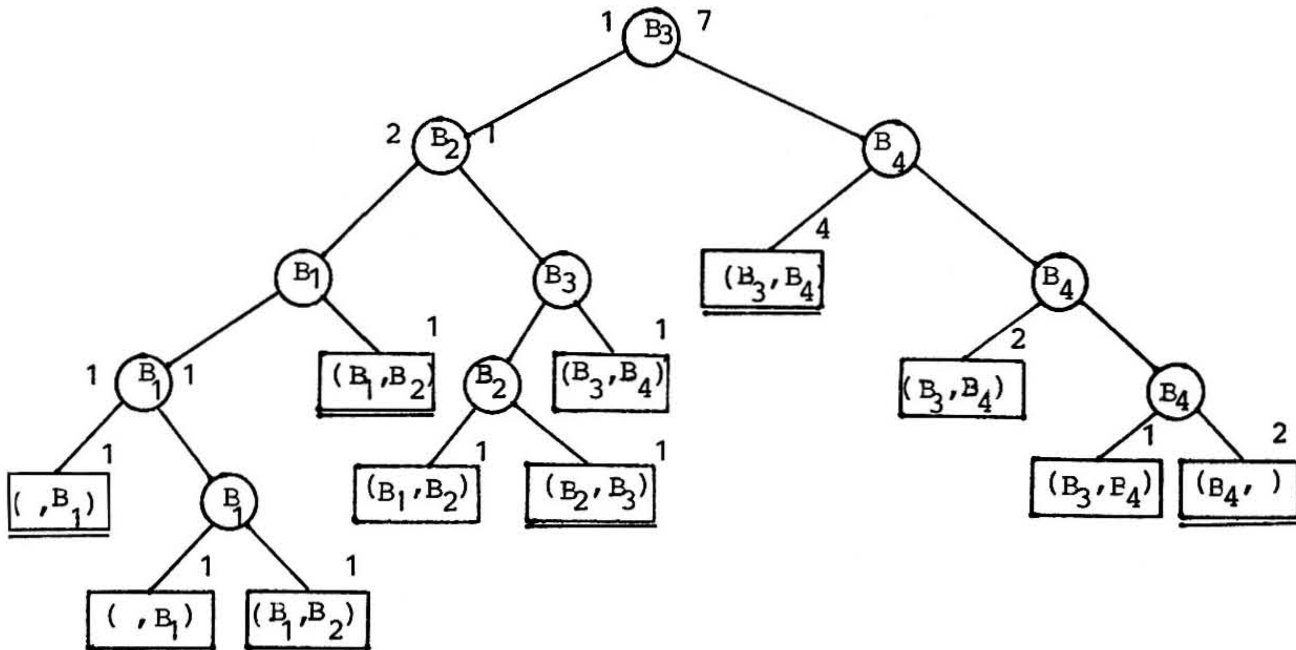


Figure 5

Suppose we search for $X < B_1$. The search is directed towards the active 0-node and destroys the balance in the node between the 0-joint and the 1-joint. A rotation about that node rebalances the tree. One gets

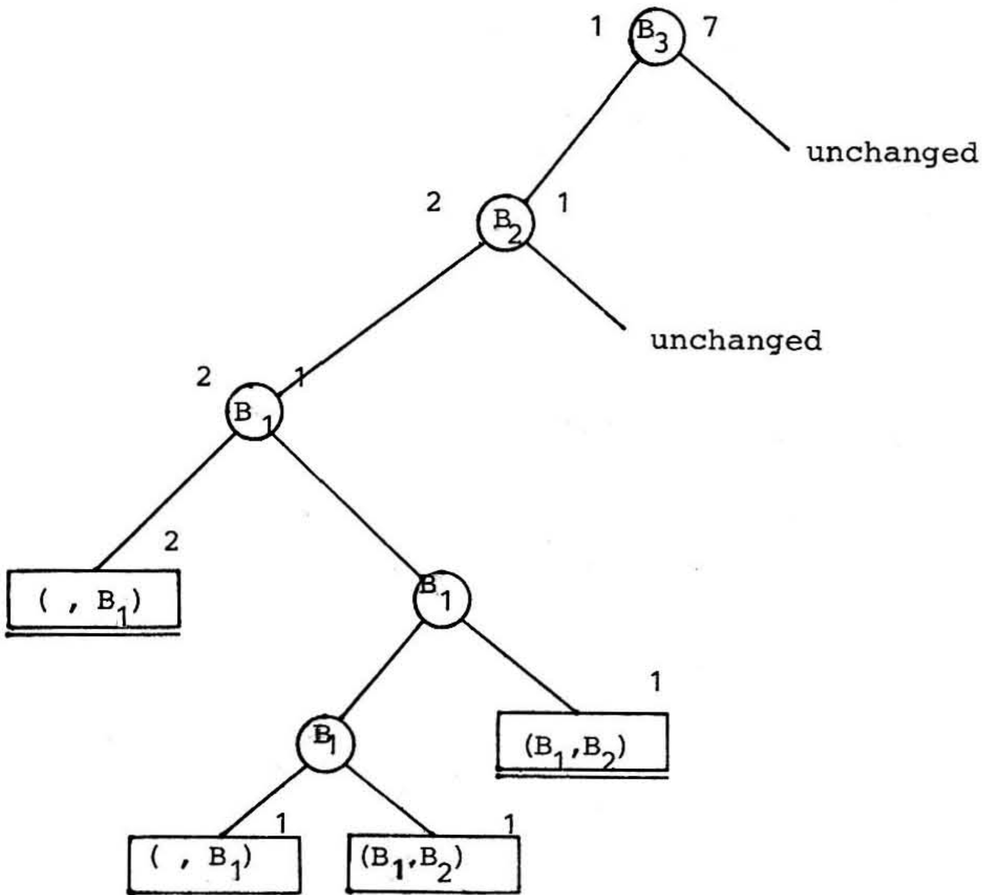


Figure 6:

IV. A Modified Definition

Recall that the active j-node is the minimal depth j-node in that subtree of the j-joint which contains at least one half of the j-leaves. Hence the active j-node is not necessarily a j-node of minimal depth. An example is shown in figure 7.

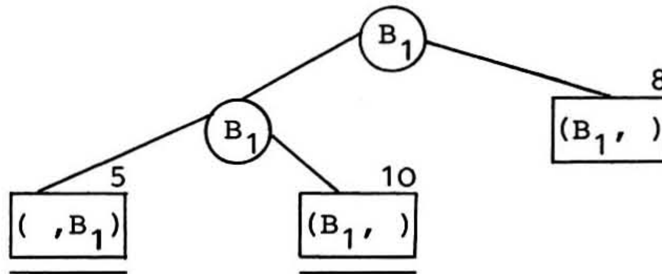


Figure 7: A D-tree for $q_o = 5$, $q_1 = 18$.

Of course, search time could be improved by ensuring that the active j-node is always a j-node of minimal depth. This leads to the following alternate definition of active j-node.

Definition: (alternate definition of active j-node). Exactly one of the j-nodes is active. The active j-node is a j-node of minimal depth.

Lemma 1 and hence Thm. 1b) is obviously true for the alternate definition of active j-node. However, the bound for the weighted path length can be improved considerably.

Theorem 2:

(Average Search Time in D-trees with the alternate definition of active j-node).

$$P \leq (1/H(\alpha,1-\alpha)) \cdot H(\alpha_o,\ldots,\alpha_n)+c$$

where $c = 1 + 1/\alpha$.

Example: Let $\alpha = 1 - \sqrt{2}/2$. Then $1/H(\alpha,1-\alpha) = 1.09$ and $c = 3 + \sqrt{2} \approx 4.41$. Because of $P_* \geq H(\alpha_0,\ldots,\alpha_n)$ [GM,K73,M77b] always, average search time is at most 9% above the optimum.

Proof: Suppose there are m_j j-nodes $v_1,\ldots,v_{m_j}$ with thickness $q_{j_1},\ldots,q_{jm_j}$ and depth $a_{j_1},\ldots,a_{jm_j}$ respectively. Then

$$q_j = q_{j_1} +\ldots+ q_{jm_j}$$

and

$$a_j = \min(a_{j_1},\ldots,a_{jm_j}).$$

Thus

$$P \leq \hat{P} = \sum_{j=0}^{n} \sum_{i=1}^{m_j} (q_{j_i}/W) \cdot a_{j_i}.$$

An easy induction argument on the height of the D-tree shows

$$\hat{P} \leq d \cdot H(\alpha_{01},\alpha_{02},\ldots,\alpha_{0m_0},\alpha_{11},\ldots,\alpha_{1m_1},\ldots)$$

where $d = 1/H(\alpha,1-\alpha)$ and $\alpha_{j_i} = q_{j_i}/W$. By the grouping axiom

$$H(\alpha_{01},\alpha_{02},\ldots,\alpha_{0m_0},\alpha_{11},\ldots,\alpha_{1m_1},\ldots)$$
$$= H(\alpha_0,\ldots,\alpha_n) + \sum_{j=0}^{n} \alpha_j \cdot H\left(\frac{\alpha_{j_1}}{\alpha_j},\ldots,\frac{\alpha_{jm_j}}{\alpha_j}\right).$$

Choose k such that $v_1,\ldots,v_K$ are left of the j-joint and $v_{K+1},\ldots,v_{m_j}$ are right of the j-joint. Let $\alpha'_j = \alpha_{j_1}+\ldots+\alpha_{j_K}$ and $\alpha''_j = \alpha_j - \alpha'_j$. Again, by the grouping axiom

$$H\left(\frac{\alpha_{j_1}}{\alpha_j},\ldots,\frac{\alpha_{jm_j}}{\alpha_j}\right) \leq H\left(\frac{\alpha'_j}{\alpha_j}, \frac{\alpha''_j}{\alpha_j}\right) + \frac{\alpha'_j}{\alpha_j} H\left(\frac{\alpha_{j_1}}{\alpha'_j},\ldots,\frac{\alpha_{j_K}}{\alpha'_j}\right) +$$
$$\frac{\alpha''_j}{\alpha_j} \cdot H\left(\frac{\alpha_{j_{K+1}}}{\alpha''_j},\ldots,\frac{\alpha_{jm_j}}{\alpha_j}\right).$$

Consider nodes $v_1,\ldots,v_K$. Among these v_1 has maximal depth and v_k has minimal depth (cf. figure 2). For $2 \leq \ell \leq k$ let w_ℓ be the

father of v_ℓ and let z_ℓ be the other son of w_ℓ. Then

$$th(w_\ell) = th(v_1)+\ldots+th(v_\ell) + x$$
$$th(z_\ell) = th(v_1)+\ldots+th(v_{\ell-1}) + x$$

for some number x. Furthermore $th(z_\ell) \le (1-\alpha)\,th(w_\ell)$ since the underlying tree is in $BB[\alpha]$. Hence

$$th(v_1)+\ldots+th(v_{\ell-1}) \le (1-\alpha)(th(v_1)+\ldots+ th(v_\ell))$$

for $2 \le \ell \le k$. By repeated application of the grouping axiom

$$\alpha_j' \, H\left(\frac{\alpha_{j_1}}{\alpha_j'}, \ldots, \frac{\alpha_{j_k}}{\alpha_j'}\right) = \sum_{\ell=2}^{k} (\alpha_{j_1}+\ldots+\alpha_{j_\ell}) \cdot H\left(\frac{\alpha_{j_\ell}}{\alpha_{j_1}+\ldots+\alpha_{j_\ell}}, \frac{\alpha_{j_1}+\ldots+\alpha_{j_{\ell-1}}}{\alpha_{j_1}+\ldots+\alpha_{j_\ell}}\right)$$

$$\le \sum_{\ell=2}^{k} (\alpha_{j_1}+\ldots+\alpha_{j_\ell})$$

$$\le \sum_{\ell=0}^{k-2} (1-\alpha)^{\ell} \cdot (\alpha_{j_1}+\ldots+\alpha_{j_k})$$

$$\le (1/\alpha) \cdot (\alpha_{j_1}+\ldots+\alpha_{j_k})$$

Hence

$$H\left(\frac{\alpha_{j_1}}{\alpha_j}, \ldots, \frac{\alpha_{jm_j}}{\alpha_j}\right) \le 1 + 1/\alpha$$

and

$$P \le (1/H(\alpha,1-\alpha)) \cdot [H(\alpha_0,\ldots,\alpha_n) + 1 + 1/\alpha]. \qquad \square$$

The implementation of D-trees with the alternate definition of active nodes is considerably more difficult than the one suggested in section III. This comes from the following fact: With the old definition of active j-node a different j-node can become active after a transformation only if the distribution of the j-leaves with respect to the j-joint changes.

This can only be the case if the j-joint node is involved in the rotation or double rotation. However, this is no longer true for the modified definition of active j-node. A different j-node may become active if the distance of some j-node to the j-joint changes. This can happen even if the j-joint is not involved in the tree transformation but rather the transformation takes place far below the joint node. This forces us to include some additional information in a D-tree:

For every $j (0 \leq j \leq n)$: The j-nodes and the j-joint are kept in a doubly linked list in symmetric order. With each link of the list we associate the distance of the j-node (if the j-node is of minimal depth) or the distance to the father of the j-node above. Figure 8 shows the D-tree of figure 5 with the additional data structure.

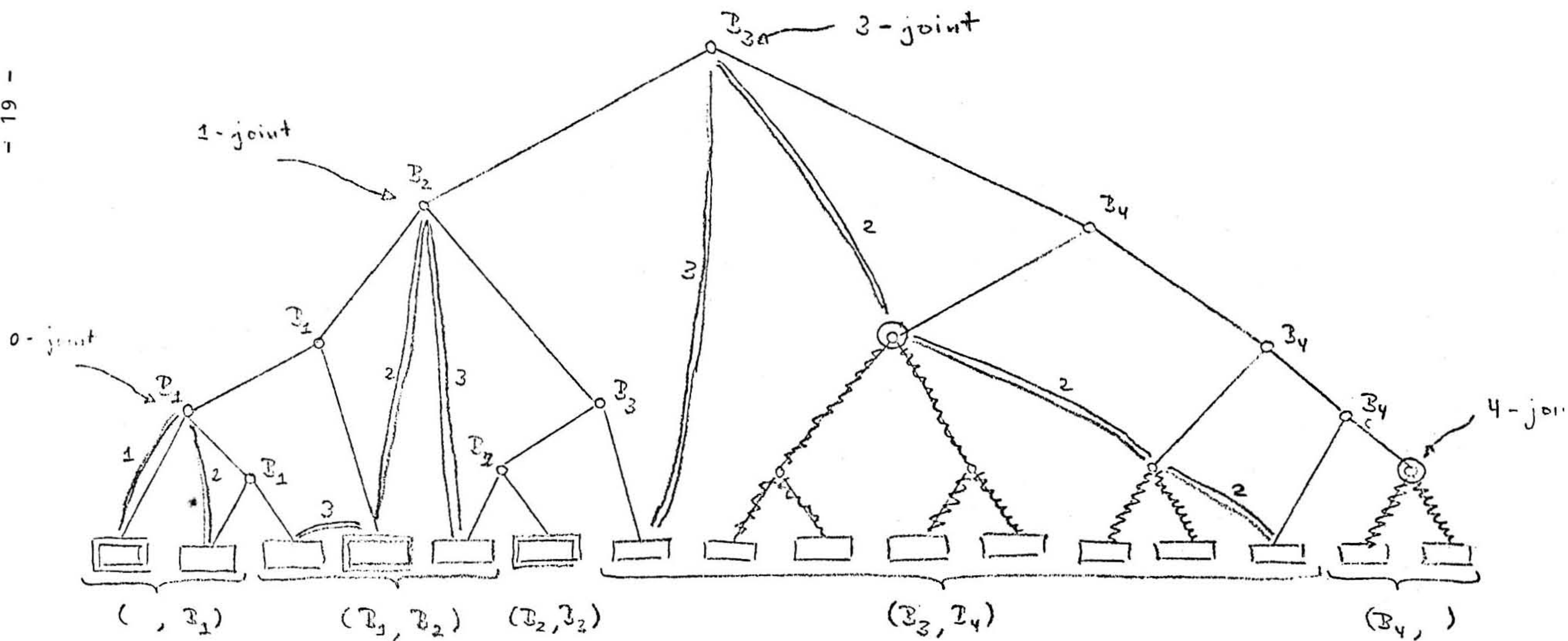
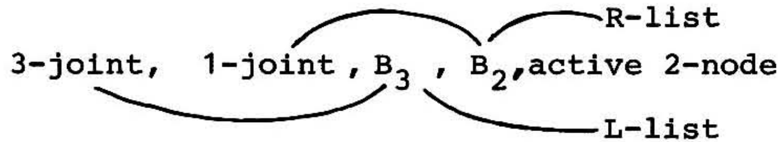


Figure 8: The D-tree of figure 5 redrawn for the modified definition of active node.

It remains to describe the search and update process. Assume we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. The nodes passed during the descent are stacked. In addition, the nodes are entered into two linked lists, the R-list and the L-list. A node is stored on the R-list if it is left via its right link; the L-list contains all nodes which are left via their left links.

<u>Example:</u> Assume that we search for $X \in (B_2, B_3)$ in the tree of Fig. 8. Then the stack contains



with the R- and L-lists as shown.

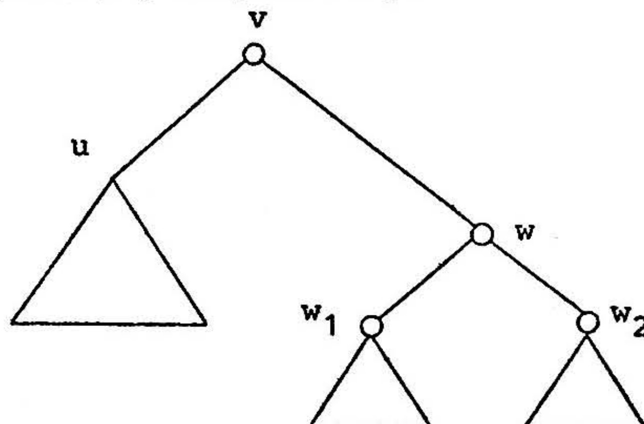
Furthermore the thickness of every node (including the active j-node) encountered during the descent is increased by one.

Then we ascend (using the stack). Suppose we reach node v from node w. We may assume w.l.o.g. that w is v's right son.

<u>Case 1:</u> $th(w) \le (1-\alpha)th(v)$. Then everything is fine. We delete v from the stack and the R- and L-list (wherever it is on) and ascend one more level.

<u>Case 2:</u> $th(w) > (1-\alpha)th(v)$. We distinguish two cases.

<u>Case 2.1 :</u> w is a j-node for some j. Then w is the j-node with $X \in (B_j, B_{j+1})$. We split w into 2 j-nodes w_1, w_2 of thickness $\lfloor th(w)/2 \rfloor$ and $\lfloor th(w)/2 \rfloor$ respectively.



<u>Figure 9:</u>

Then $\alpha \leq \rho(W) \leq 1/2 \leq (1-2\alpha)/(1-\alpha)$. Remember that $\alpha \leq 1-\sqrt{2}/2$ and $th(w) \geq 2$. By fact 1 a rotation will rebalance the tree. We obtain

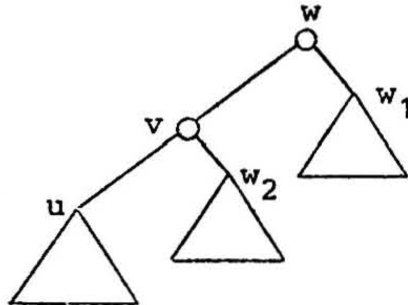


Figure 10:

If either v or w was the j-joint before the rotation when w is the j-joint afterwards. In either case we insert w_1, w_2 (and may be w) into the proper place of the doubly linked list of j-nodes. The rotation demotes u by one level. This might cause a node below u to become inactive. Let (B_j, B_{j+1})

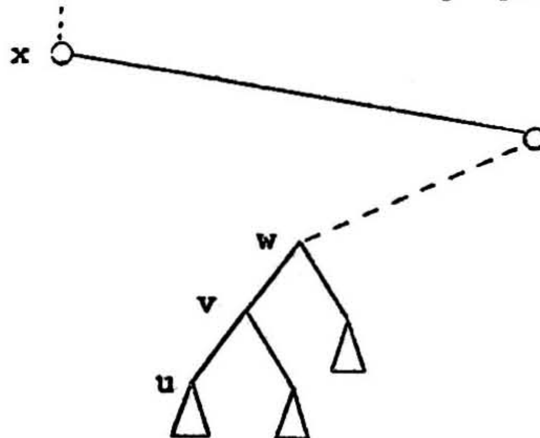


figure 11:

be the leftmost leaf below u and let x be the first node on the R-list. Three cases may occur.

a) x is the j-joint. Then we increase the number associated with the link from x to the j-node below u by 1. Then we compare the number with the number associated with the link to the left and change the query assigned to x if necessary. Note that this will direct future accesses to $X \in (B_j, B_{j+1})$ to the newly active j-node.

b) the other subtree of x is an j-node. Then we increase the distance from this j-node to the j-node below u by 1.

c) neither of above. Then the active j-node is also below u and we have nothing to do. This finishes case 2.1 . We delete v from the stack and the R- and L-list (wherever it is on) and ascend.

Note that w is not a j-node for any j after the rotation is performed. This shows that case 2.1 occurs only in the first step of the ascent.

Case 2.2: w is not a j-node. We have the following picture.

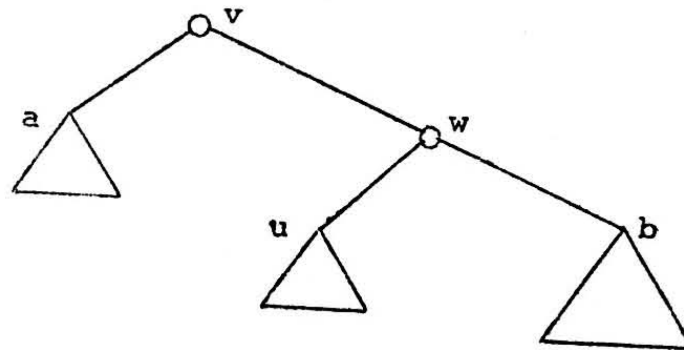


figure 12:

If $\rho(w) \leq (1-2\alpha)/(1-\alpha)$ then we perform a rotation, otherwise a double rotation. Since a double rotation is just two rotations we only have to treat the case of a rotation. A rotation about v demotes subtree a one level and promotes subtree b one level. Hence we have to go through the routine described above for the left- and rightmost leaves of subtrees a and b. We also have to distinguish whether u is a j-node for some j or not. The details are left to the reader.

We summarize the discussion. Modifications of the tree structure are limited to the path of search. In each node of that path a constant amount of work is required. Hence update time is proportional to search time. Theorem 1 is true even for the modified definition of active nodes.

V. Compact D-Trees

The tree structure of section III achieves one main goal: search time is logarithmically bounded and update time is proportional to search time. However, our solution might use $O(n \cdot \log W)$ storage cells. In addition, the depth of a node, though being bounded by $O(\log w^t/q_j^t)$, can be arbitrarily large compared to n. Consider the case that $q_j^t = 1$ for all t and w^t becomes large.

However, most of the nodes are only present to make rebalancing and bookkeeping easy to explain. In this section we propose a compact version of the tree structure. It exhibits the same search time and update behavior as the basic structure of section III; in addition, it requires only $O(n)$ storage cells and permits a linear time construction. Also the depth of the active j-node is bounded by $O(\min(n, \log w^t/q_j^t))$.

We obtain a compact tree from a D-tree in the sense of section III which we call an extended tree from now on by node deletion and path compression. The compact tree is formed by the query nodes which contain active nodes in both subtrees, the joint nodes and the active nodes. All other nodes are deleted. Applying this process to the tree of figure 5 yields:

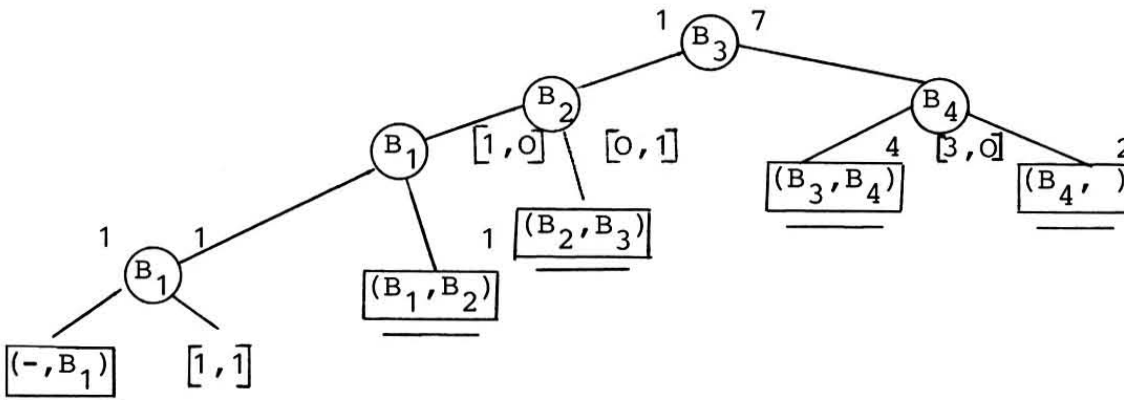


figure 13: The compact version of the tree in figure 5.

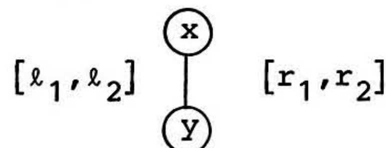
We remember the deleted nodes by storing expressions of the form [number, number] along the compressed edges. For example between the node B_4 and the active 4-node we deleted two

left subtrees representing a total number of 3 leaves labelled (B_3, B_4) and no leaves labelled $(B_4, \ )$. We denote this by the expression [3,0] on the left side of the edge joining B_4 and the active 4-node. The right subtree of the 1-joint was deleted completely. It contained one 0-leaf and one 1-leaf. This is denoted by the expression [1,1].

More formally, the edge labels are assigned as follows. Consider any node x of the extended tree which is not deleted and any edge emanating from it.

Case 1: x has no right (left) descendant which is not deleted. Then the right (left) subtree of x contains no active node and hence at most two kinds of leaves. The right (left) subtree is replaced by the expression $[n_1, n_2]$ where n_1 (n_2) denotes the number of the first (second) kind of leaves. If the query assigned to x is "if B_j ..." and the edge is right emanating then n_1 denotes the number of (B_{j-1}, B_j)-leaves and n_2 denotes the number of (B_j, B_{j+1})-leaves. An analogous statement holds if the edge is left emanating.

Case 2: x has a right (left) descendant which is not deleted. Let y be such a descendant of minimal depth. Then all proper descendants of x which are not descendants of y were deleted. (Otherwise y is not minimal depth). Hence the path from x to y is compacted to a single edge.



The left (right) subtrees along the path from x to y contain no active node and hence at most two kinds of leaves. Their respective numbers are stored in the expressions $[\ell_1, \ell_2]$, $[r_1, r_2]$. The above comment about the meaning of n_1, n_2 holds analogously.

Note also that more than one extended tree may be represented by the same compact tree. E.g. the compact tree above also represents the tree whose right subtree looks as follows:

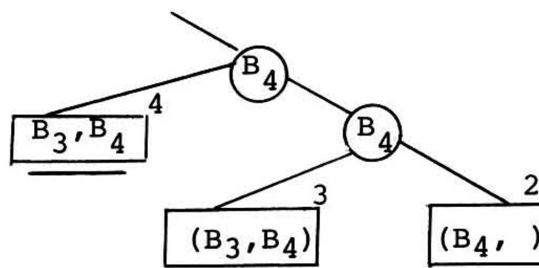
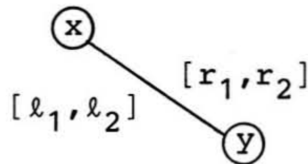


Figure 15

So every compact D-tree represents a whole class of extended D-trees. We will operate on a compact D-tree as if we were operating on one of the extended D-trees represented by that compact D-tree. Hence we need a method to (locally) construct an extended tree from a compact tree. The method is based on the following facts about the edge labels in compact trees.

Lemma 1: Let T be an extended D-tree and let T^c be the compact D-tree constructed from it.

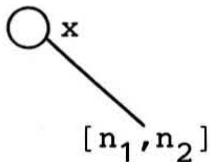
1) Consider any edge



in T^c. Let $\ell = \ell_1 + \ell_2$ and $r = r_1 + r_2$ and $\gamma = \alpha/(1-\alpha)$ where α is the balancing parameter. Then either

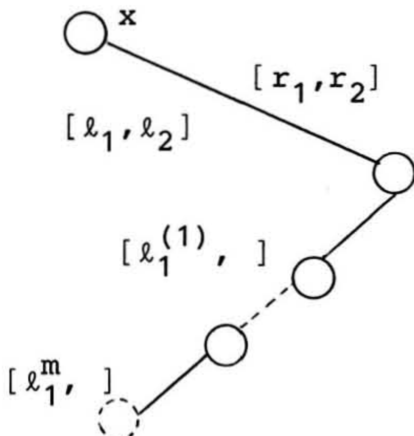
a) $\ell = r = 0$ or
b) $\ell \geq \beta t$ and ($r \geq \gamma(\ell+t)$ or $r=0$) or
c) $r \geq \beta t$ and ($\ell \geq \gamma(r+t)$ or $\ell=0$).

2) Consider any node x in the compact tree and an edge right emanating from it. Then either



and x is a joint node, say the j-joint, and $n_1 \leq q_j/2$

or



Then y has at least one active descendant. Let the active (j+1)-node be the leftmost active descendant of y and let the active (k-1)-node be the rightmost active descendant of y. Let t = th(y)

2.1) If $\ell_2 \neq 0$ then $j+1 = k-1$, y is the active $(j+1)$-node, $r_2=r_1=0$, $\ell_2+1 \leq t/\gamma$ and either $\ell_1+\ell_2 \leq t/\gamma$ or $(t+\ell_2+1) \leq (\ell_1-1)/\gamma$.

2.2) If $r_1 \neq 0$ then $j+1 = k-1$, y is the active $(j+1)$-node, $\ell_1=\ell_2=0$, $r_1+1 \leq t/\gamma$ and either $r_1+r_2 \leq t/\gamma$ or $(t+r_1+1) \leq (r_2-1)/\gamma$

2.3) If $\ell_1 \neq 0$ then x is a descendant of the j-joint, if $\ell_2 \neq 0$ then x is a descendant of the $(j+1)$-joint.

2.4) If $r_2 \neq 0$ then x is a descendant of the $(k-1)$-joint, if $r_1 \neq 0$ then x is a descendant of the $(k-1)$-joint.

2.5) If x is the j-joint then let $[\ell_1^{(1)},], \ldots, [\ell_1^{(m)},]$ be the edge labels on the left frontier of the subtree with root y. Then $0 < \ell_1 + \sum_{p=1}^{m} \ell_i^{(p)} \leq q_j/2$.

Proof: 1) The path from x to y in the extended D-tree has length k, $k \geq 1$. If $k = 1$ then y is a son of x, no subtrees were deleted and hence $\ell = r = 0$. Suppose $k > 1$. Let $w \neq y$ be the son of x on the path from x to y. Then $th(w) = \ell + r + t$. Let c_1 (c_2) be the thickness of that subtree of w which contains y (does not contain y). Then either $c_2 \leq r$ or $c_2 \leq \ell$ and $c_1 + c_2 = th(w)$. Furthermore,

$$c_2 \geq \alpha \cdot th(w)$$

$$c_1 \leq (1-\alpha)\ th(w)$$

Hence $c_2 \geq \gamma c_1$ and therefore either $\ell \geq \gamma(t+r)$ or $r \geq \gamma(\ell+t)$. If y is in the left (right) subtree of w then the same argument applied to a node v (if it exists) on the path from x to y such that y is a right (left) descendant of v finishes the proof.

2) If x has an edge right emanting from it but no right descendant then x does not have active nodes in both of its subtrees. Hence x must be a joint node by the definition of compact D-tree. Say node x is the j-joint. Then n_1 is the number of j-leaves to the right of x and hence $n_1 \leq q_j/2$ since the active j-node is a left descendant of x.

Suppose x has a right descendant, say y. Since every node in a compact D-tree has at least one active descendant (not necessarily proper), so does y. Let the active j+1 be the leftmost active descendant of y and let the active k-1 node be the rightmost active descendant of y. Suppose $\ell_2 \neq 0$. ℓ_2 is the number of (j+1)-leaves which are members of the left subtrees of the path from x to y in the extended tree. Hence the (j+1)-joint w is a proper ancestor of y. w is either to the left of y or to the right of y in tree T. If w is to the left of y then $\ell_1 = 0$ and the left subtree of the path from x to y in the extended tree are (j+1)-nodes. But then the active (j+1)-node is not of minimal depth. Contradiction.

Suppose that the (j+1)-joint is to the right of y. Then k-1 = j+1 and there is only one active descendant of y. Hence y is either the (j+1)-joint or the active (j+1)-node. In the first case we have a contradiction since w was supposed to be the (j+1)-joint. In the latter case $r_2 = 0$ and hence $r_1 = 0$. In the extended tree the path from x to y has $k \geq 1$ left subtrees. All of them have to contain j-leaves because otherwise y would not be a (j+1)-node. The thickness of the left subtree of the father of y must be $\leq t/\gamma$ where $t = th(y)$ since the underlying tree is in $BB[\alpha]$. This subtree contains ℓ_2 (j+1)-leaves and at least 1 j-leaf. Hence $\ell_2 + 1 \leq t/\gamma$.
If k = 1 then even $\ell_1 + \ell_2 \leq t/\gamma$. If k > 1 let $s = \ell_1' + \ell_2 > \ell_2$ be the thickness of the left subtree of the father of y and let ℓ_1'' be the thickness of the left subtree of the grandfather of y. Then $s \geq \ell_2 + 1$ and $\ell_1 \geq \ell_1' + \ell_1'' \geq 1 + \gamma(t+s) \geq 1 + \gamma(t+\ell_2+1)$. This proves 2.1 . A similar argument proves 2.2 .

Suppose $\ell_1 \neq 0$. Hence x has j-leaves below it in the extended tree and is a descendant of the j-joint. This shows 2.3. The same argument proves 2.4 .

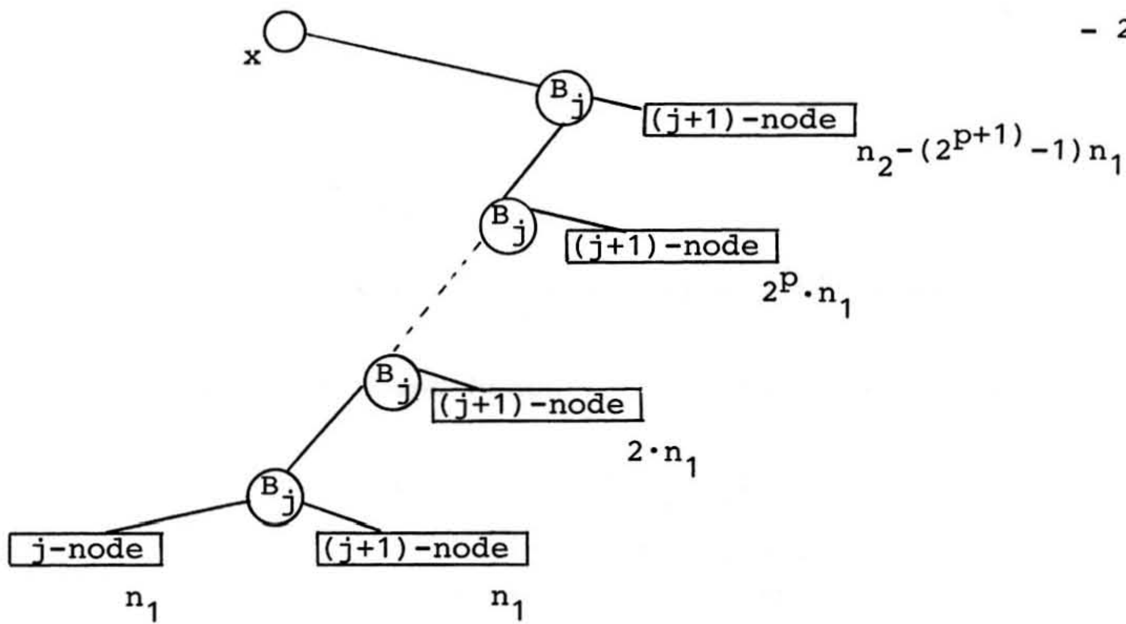
Finally 2.5 follows from the definition of active j-node and the fact that the active j-node is a proper left descendant of node x. □

Note as a consequence of lemma 1 that at most two of the 4 numbers stored in the labels of an edge can be $\neq 0$. This reduces the space requirement of compact trees somewhat. The crucial observation is that the conditions of lemma 1 are strong enough to characterize compact D-tree. Let T^c be a compact D-tree. In particular, the edge labels of T^c satisfy lemma 1. We want to construct an extended D-tree T such that compacting T gives T^c. The construction will only use the properties of the edge labels stated in lemma 1.

Let x be any node in T^c and let B_j be the query assigned to node x. Consider any edge right emanating from x. (Left emanating edges are treated similarly). If the edge is dangling i.e. x has no right descendant,then let $[n_1,n_2]$ be the edge label. We have to connect the dangling edge to a subtree with n_1 j-leaves and n_2(j+1)-leaves. If either $n_1 = 0$ or $n_2 = 0$ then we construct a j or (j+1)-node of the appropiate thickness. Otherwise, suppose $n_1 \leq n_2$ (the symmetric case is treated analogously). Then there exists a $p \geq -1$ with

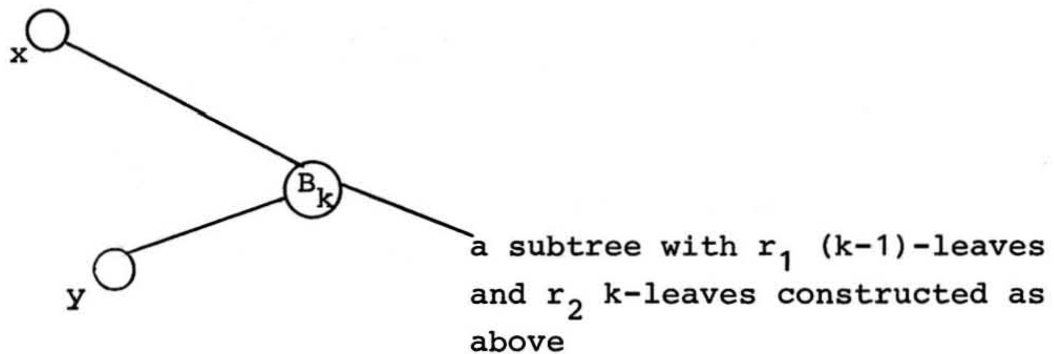
$$3 \cdot 2^p \cdot n_1 \leq n_1 + n_2 \leq 3 \cdot 2^{p+1} \cdot n_1$$

In this case we construct



It is easy to see that this tree is weight-balanced.

It remains to consider the case that x has a right son y in the compact tree. Let $[\ell_1, \ell_2]$, $[r_1, r_2]$ be the labels of the edge from x to y. If $\ell_1 + \ell_2 = r_1 + r_2 = 0$ then there is nothing to do. Assume otherwise. We treat the case $\ell_1 + \ell_2 = 0 \neq r_1 + r_2$; the other cases being similar. Let $r = r_1 + r_2$. Then $r \geq \gamma t$ where $t = th(y)$ by lemma 1. In the extended tree there is a path from x to y such that the right subtrees along that path contain r_1 (k-1)-leaves and r_2 k-leaves for some k. We show later how to determine k. If $r \leq t/\gamma$ then we construct



Note that the newly constructed node satisfies the balance criterion. Suppose $r > t/\gamma$. Let $s = \max(r_1+1, t)$. Choose

p, $p \geq -1$, such that

$$2^{(p+1)}(s+t) \leq (1-\alpha)(t+r) \leq 2^{(p+2)}(s+t).$$

If $s = r_1 + 1 > t$ and hence $r_1 \neq 0$ then p exists since

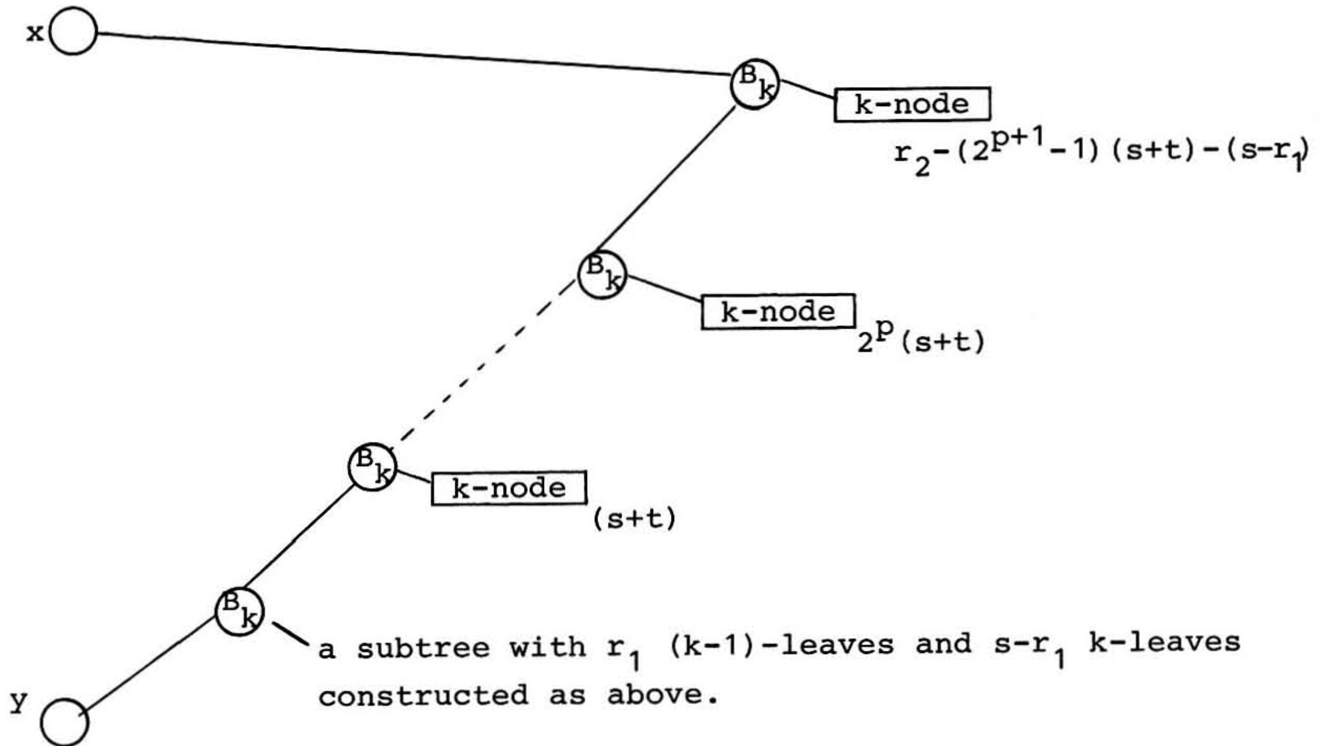
$$\begin{aligned}(1-\alpha)(t+r) &= (1-\alpha)(t+r_1+r_2)\\ &\geq (1-\alpha)(t+r_1+\gamma(t+r_1+1)+1)\\ &= t+r_1+1=2^{-1+1}(s+t)\end{aligned}$$

If $s = t$ then p exists since

$$\begin{aligned}(1-\alpha)(t+r) &\geq (1-\alpha)(t+t/\gamma)\\ &= t(1-\alpha)\cdot(1+1/\gamma)\\ &= t/\gamma > 2\cdot t\end{aligned}$$

since $\gamma = \alpha/(1-\alpha) \leq 0.43$ and hence $1/\gamma \geq 2.3$.

We construct the following tree

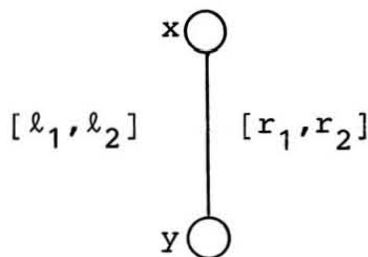


All newly constructed nodes satisfy the balance criterion: The father of y satisfies it since $r_1+1 \leq t/\gamma$, the son of x satisfies it since its thickness is t+r, the thickness of its left subtree is $2^{p+1} \cdot (t+s)$ and the choice of p (remember that $\alpha \leq 1 - \sqrt{2}/2$ and hence $(1-\alpha)/2 \geq \alpha$). The other nodes trivially satisfy the balance criterion. This ends the description of the expansion process.

It remains to be seen that the extended tree obtained in this way is actually a D-tree. The newly constructed nodes satisfy the balance criterion, the other nodes satisfied the balance criterion as nodes of the compact tree and hence satisfy it as nodes of the extended tree.

It is also easy to see that the joint nodes of the compact tree are still joint nodes of the extended tree. Let x be the j-joint in the compact tree. It follows from 2.3 and 2.4 of lemma 1 that the j-joint in the extended tree cannot be below x. Hence x is the j-joint in the extended tree.

Finally, consider the active j-node z in the compact tree. We want to show that z is the active j-node in the extended tree obtained by the expansion process. Certainly z is on that side of the j-joint which contains $\geq q_j/2$ of the j-leaves (by 2.5). Hence if z were not active in the extended tree then there must be a j-node of smaller depth on the same side of the j-joint. Since this j-node does not exist in the compact tree, it was constructed during the expansion process. Hence there must be an edge x



in the compact tree such that the active j-node is the leftmost active descendant of y and $\ell_2 \neq 0$ or it is the rightmost active descendant of y and $r_1 \neq 0$. In either case y is the active j-node

(2.1 or 2.2 of lemma 1) and either $r_1+r_2 = 0$ or $\ell_1 + \ell_2 = 0$. We treated the case $\ell_1 + \ell_2 = 0$, $r_1 \neq 0$ of the expansion process in detail above. It does not introduce any j-nodes of smaller depth than y.

Lemma 2: Let T^c be a compact D-tree. The expansion process applied to T^c yields an extended D-tree T.

Proof: By the discussion above.

Another property of the expansion process is usefull in the sequel. Expansions can be done locally, i.e. knowing the edge label and the thickness of the end point permits proper expansion of an edge. Furthermore, it is possible to expand an edge partially. Say, only to generate father and grandfather of the end point y and son and grandson of the starting point x.

We are now able to describe the searching process in compact D-trees. Assume we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. The nodes passed during the descent are stacked, their thickness is increased by one and they are entered into an R-list and L-list as described in section IV. Then we ascend. Suppose w.l.o.g. we reach node x from its right son y. Two things can happen which require an action. Either the edge label of the edge from x to y does not satisfy lemma 1 anymore or node x has gone out of balance or both.

Case 1: The label of the edge from x to y does not satisfy lemma 1 any longer. Then either condition 1 or 2.1 or 2.2 is violated. Violations of conditions 2.1 and 2.2 are treated analogously. So suppose 2.2 is violated. Then y is the active (j+1)-node for some j+1. Let t be the thickness of y before the search. Then $(t+r_1+1) \leq (r_2-1)/\gamma < (t+r_1+2)$. Furthermore $r_1 + 1 \leq t/\gamma$. Hence $r = r_1+r_2 \leq c \cdot t$ for some constant c dependent only of α. Hence the expansion of the edge from x to y yields a path of bounded length, the bound only depending on α. So we can afford to expand the right subtree of x complete and operate on it as we did in the case of extended D-trees. This shows how to

handle violations of 2.1 or 2.2 . Suppose now that condition 1 is violated. Then either $0 < \ell < \gamma(t+1)$ or $0 < r < \gamma(t+1)$ or ($r < \gamma(r+t+1)$ and $\ell < \gamma(r+t+1)$ and $\ell \neq 0 \neq r$). Here t denotes the weight of the right subtree of x before the search. In the first two cases we have to expand the edge from x to y only near y (generation of y's father suffices) and in the third case we can afford to expand the edge completely. Note also that using the L- and R-lists it is possible to determine what kind of leaves have to be constructed. In either case we reduced the operations on compact trees to the corresponding operations on expanded trees.

Case 2: Node x has gone out of balance. A rotation or double rotation will rebalance the tree. Performing the transformation might require to partially expand the edges emanating from x. This causes no problems.

We summarize the discussion in

Theorem 3: Consider a compact D-tree based on a BB[α]-tree with $0 < \alpha \leq 1-\sqrt{2}/2$.

a) Let q_j^t be the number searches for $X \in (B_j, B_{j+1})$, $0 \leq j < n$, performed up to time t and let $W^t = \Sigma q_j^t$. Then at time t a search for $X \in (B_j, B_{j+1})$ can be executed in time $O(c_1 \log W^t/q_j^t + c_2)$ where $c_1 = 1/\log(1/1-\alpha))$ and $c_2 = 1+c_1$. The time needed to update the tree structure is proportional to the search time.

b) A compact D-tree has O(n) nodes and edges and thus requires storage space O(n).

c) Given a distribution $(q_0, q_1, \ldots, q_n)$ it is possible to construct a compact D-tree for it in linear time O(n).

Proof: a) was proved by the discussion above.

b) A compact D-tree has $\leq$ 2n interior nodes and hence $\leq$ 4n edges. In each node and edge only a fixed constant number of fields are required.

c) In [M77a] (cf. also [M77b]) an algorithm for constructing nearly optimal binary search trees was introduced. It essentially constructs a compact-tree except for the labels of the edges. The labeling process is easily incorporated in that procedure. The details are left to the reader.

VI. Extensions

VI.1 General Search Trees

So far, we considered only searches for elements not in the name set, i.e. $X \notin \{B_1,\ldots,B_n\}$. We drop that restriction and return to the model described in the introduction. Let p_i (q_j) the number of searches conducted for $X=B_i$ (($X \in (B_j,B_{j+1})$ up to now and let

$$W = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_j$$

be the total number of searches conducted so far. Then $\beta_i = p_i/W$ ($\alpha_j=q_j/W$) is the relative access frequency of $X = B_i$ ($X \in (B_j,B_{j+1})$) at this point of time.

Define new frequencies q_j', $0 \le j \le n$, by $q_0' = q_0$ and $q_j' = q_j + p_j$ for $1 \le j \le n$, i.e. we change the open intervals (B_j,B_{j+1}) into the half-open intervals $[B_j,B_{j+1})$, and construct the tree structure of section III (the compact tree of section V) for the new set of frequencies. A search for a name X is carried out as above. With search argument $X \in [B_j,B_{j+1})$ we wil reach the j-active node. (Note that we assigned queries of the form "<u>if</u> $X < B_i$ <u>then</u> left <u>else</u> right"). In the j-active node we will distinguish between $X = B_j$ and $X \in (B_j,B_{j+1})$ by one more comparison. A search for $X \in [B_j,B_{j+1})$ will take time $O(\log W/q_j') = O(\log W/p_j)$ $(= O(\log W/q_j))$, i.e. we still have logarithmic behaviour.

The trick used here to due to D.E. Knuth [Vol. 3, section 6.2.2, exercise 36].

VI.2 Insertions

Suppose we want to insert a new name $B \notin \{B_1,\ldots,B_n\}$ into the name set, say $B \in (B_j,B_{j+1})$. A search for B will end in the active j-node representing the half open interval $[B_j,B_{j+1})$. We have to split the interval $[B_j,B_{j+1})$ and the associated frequency $p_j + q_j$ into two intervals $[B_j,B)$ and $[B,B_{j+1})$ with frequencies $p_j + q_j'$ and $1 + q_j''$ respectively (1 is the frequency of name B and $q_j = q_j' + q_j''$). The splitting of q_j into q_j' and q_j'' may be prescribed arbitrarily.

Using the distribution of j-leaves with respect to the j-joint (stored in the j-joint) and the thickness of the various j-nodes we identify that j-node v such that the j-nodes to the left (right) of v contain $\le q_j'$ ($\le 1+q_j''$) j-leaves. We split this j-node into two nodes of type $[B_j,B)$ and $[B,B_{j+1})$. Then we perform the required changes to the tree structure. The j-joint gets new distribution numbers, the active j-node may change and we need to create a new $[B,B_{j+1})$-joint (the father of node v). It is easy to see that these changes can be carried out in time proportional to the depth of the newly created nodes of type $[B_j,B)$ and $[B,B_{j+1})$. This depth is bounded by $O(\log \max (w/q_j', w/q_j''))$ in the case of extended D-trees and by $O(\min (n, \max (w/q_j', w/q_j''))$ in the case of compact trees.

VII. Experimental Results

Compact D-trees were implemented by H. Reinshagen and A. Del Fabro. They kindly provided some of the results of their experiments.

A node of a compact D-tree is represented by six components, the query, the pointers to the two sons, the thickness, and the labels of the incoming edge. The distribution of j-leaves with respect to the j-joint is stored in the son pointers of the active j-node. The type of a node is stored in the sign bits. Since a compact D-tree has between 2n and 4n nodes and leaves the storage requirements are between 12n and 24n storage cells. Some additional savings are possible. Lemma 1 of section V states that the label of an edge has only one non-zero component except in two special cases. This observation can be used to reduce the space requirements somewhat. A more promising approach is to delete all interior nodes which do not have active nodes in both subtrees. The updating is more complicated in this case but space requirement goes down to 12n.

Experiments of the following form were carried out. Starting with an arbitrary tree searches were performed according to a fixed probability distribution. After some number of searches the weighted path length P_D of the D-tree was computed and compared with the weighted path length P_{opt} of an optimal tree for the fixed underlying probability distribution. The optimal tree was constructed by means of the Hu & Tucker algorithm. Two probability distributions were used.

Distribution 1: n = 200; $p_i = e^{-100} \frac{100^i}{i!}$, $1 \le i \le 200$, Poissondistribution

Distribution 2: n = 200; the second distribution was obtained by counting the number of words in a German dictionary starting with different two letter combinations.

Table 1 shows the statistics for the first 5ooo searches in the case α = o.25. The table shows the deviation (in percent) from the weighted path length of the optimum tree $(P_D-P_{opt})/P_{opt}$ and the total number of rotations and double rotation (#R+DR)

	Distribution 1		Distribution 2	
# of searches	$(P_D-P_{opt})/P_{opt}$	#R+DR	$(P_D-P_{opt})/P_{opt}$	#R+DR
o	48.6	o	31.5	o
1oo	36.3	13	14.4	52
2oo	33.4	18	12.4	71
5oo	2o.9	22	8.1	1o6
1ooo	15.3	25	5.9	145
2ooo	8.8	33	5.4	187
3ooo	6.3	34	5.2	2o7
4ooo	5.2	34	5.2	229
5ooo	4.9	35	4.9	244

Table 1: α = 0.25, different distributions

Table 2 shows the statistics for the same initial tree, but different values of α. The second distribution was used.

	α = o.1		α = o.2		α = o.25		α = 1-√2/2	
#of searches	$(P_D-P_{opt})/P_{opt}$	#R+DR	$(P_D-P_{opt})/P_{opt}$	#R+DR	$(P_D-P_{opt})/P_{opt}$	#R+DR	$(P_D-P_{opt})/P_{opt}$	# R+DR
o	22.9	o	22.9	o	22.9	o	22.9	o
1oo	23.5	o	22.7	7	18.3	22	14.5	52
2oo			2o.6	2o	16.o	38	12.6	79
5oo	23.9	9	17.o	47	14.5	74	9.9	148
1ooo	23.3	18	15.1	8o	11.7	115	7.6	2o7
2ooo	19.9	37	14.6	1o9	1o.5	166	6.7	273
3ooo	19.6	43	14.4	135	1o.5	2o8	6.2	315
4ooo	2o.9	48	13.9	15o	1o.1	232	6.o	345
5ooo	19.4	54	13.8	165	1o.1	248	5.8	37o

<u>Table 2:</u> Same initial tree, 2nd distribution, different values of α .

The examples show that the weighted path length of the compact D-tree approaches the weighted path length of the optimum tree quite rapidly. They also show that the overhead for maintaining the D-tree is not very large. The maximal number of rotations and double rotations in the first 5ooo searches were 37o. It is interesting to observe here that Blum and Mehlhorn [BM] have shown that at most $c \cdot n$ rotations and double rotations suffice to perform n insertions and deletions on an initially empty $BB[\alpha]$ tree. c is a constant independent of n. This is in sharp contrast to the self-organizing binary search trees proposed by Allan and Munro: A rotation about every node on the path of search is required there. In our examples the weighted path length of the optimum tree was about 6.8 . Hence about $5ooo \cdot 6.8 = 34ooo$ rotations would be required for the first 5ooo searches. This shows that with respect to efficieny self-organizing binary search trees are not competitive with D-trees. However, they use less space.

VIII. 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 charcters. One such method is the TRIE. (ef. [K 73]). 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 Σ.

Several implementations of tries were proposed.

1) Each node of the trie is represented by a vector of length $|\Sigma|$. 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 mostly 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.

There is no a priori reason why the identification of characters has to proceed from left to right; any order will do. Comer and Sethi show that it is NP-complete to find the ordering which minimizes average search time under implementation 1.

However, with respect to implementation 3 and nearly optimal average search time all orderings will do. Let $S = \{B_1,\ldots,B_n\}$ be the set of keys and suppose all keys are of equal length m. For a string $w \in \Sigma^*$ let

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

We represent a node q of a trie by a D-tree (or any other kind of nearly optimal search tree) for the distribution $\{p_{wa};\ a \in \Sigma\}$. A key $B_i = a_{i1}a_{i2},\dots,a_{im}$ is identified by successively identifying the character a_{ik} in the tree corresponding to the node $a_{i1},\dots,a_{i(k-1)}$ of the tree. It takes time $O(c_1 \cdot \log p_{a_{i1},\dots,a_{i(k-1)}}/p_{a_{i1},\dots,a_{ik}} + c_2)$ to identify a_{ik} where c_1, c_2 only depend on the balance parameter (cf. lemma 1). Hence B_i can be identified in time $O(c_1 \log p_\varepsilon/p_{a_{i1},\dots,a_{im}} + c_2 \cdot m) = O(c_1 \log n + c_2 m)$. 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. This problem is discussed in greater detail in Fredman and Güttler, Mehlhorn, Schneider.

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 alle prefixes of w. 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 it is no problem to retain near optimality since we use D-trees to implement the nodes of a trie. (Although we assumed in section III-VI that a search increases the frequency of a node by one we used in the proofs only that it changes the frequency by at most one. A slight complication arises in the case of compact D-trees since the active node can switch sides with respect to the joint node; the reader should have no difficulties to remedy that problem). This leads to the following

<u>Theorem</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 $O(\log|S| + m)$.

In database applications keys frequently are m-tuples and comparisons between keys is no longer an elementary operation. Balanced tree schemes based on key comparisons (AVL-trees, B-trees,...) loose some of their usefulness in this context. In this case TRIES combined with D-trees may prove a real alternative.

We restricted our discussion to keys of uniform length and equal probability. The results are readily extended to the general case [M 77c].

IX. Conclusion

We introduced D-trees as an extension of weight-balanced trees. D-trees permit near optimal access under time-varying access probabilities. More precisely, let p be the number of accesses to object B and let W be the total number of accesses up to time t_0. Then at time t_0 an access to oject B can be performed with $c_1 \cdot \log(W/p) + c_2$ comparisons between keys for some small constants c_1, c_2. Furthermore, updating the tree structure is limited to the path of search and takes time $O(c_1 \log W/p + c_2)$. On the average only a constant amount of work is required (NR,BM)

Two different versions of D-trees are introduced, one favoring access time , the other favoring update time. Compact D-trees were introduced to cut down on the space requirement of the basic scheme. Finally, an application to TRIES is given. Searching in a set S of multi-attribute keys (length m), inserting into and deleting from it can be done in time $O(c_1 \log|S| + c_2 \cdot m)$.

Similar problems were considered by Allan & Munro , Baer and Unterauer. Unterauer and Baer also describe extensions of weight-balanced trees. Unterauer proves bounds on the search time (similar to ours), however update time may be $\Omega(n^2)$ in the worst case. It is O(search time) in the average case. Baer only gives empirical results. Allan & M nro describe an extension of self-optimizing linear list schemes to trees. They derive a bound on the asymptotic average search time; updating is limited to the path of search; however, a rotation about every node of the path of search is necessary.

X. References

[AM] Allan, Munro: Self-Organizing Binary Search, Proc. 17 th Symposium on Foundation of Computer Science, 1976.

[BAE] Baer, J.L.: Weight-Balanced Trees, Proc. AFIPS 1975 NCC, Vol. 44, 467-472.

[BAY] Bayer, P.: Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees, MIT, 1975.

[CL] Clampett, H.A.: Randomized binary searching with the tree structures, CACM 7, 3 (March 1964), 163-165

[CS] Comer, M., Sethi, R.: Complexity of Trie Index Construction, 17th IEEE Symposium on Foundations of Computer Science, 1976, 197-207.

[FK] Fredkin, E.: Trie Memory, CACM 3, 9 (sept. 60), 490-499

[FM] 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 and Computing, 1975.

[GM] Gilbert, E.N. and Moore, E.F.: Variable length binary encodings, Bell Systems Techn. J. 38 (1959).

[GW] Gotlieb, CC. and Walker, W.A.: A Top-Down Algorithm for Constructing Nearly Optimal Lexicographical Trees, Graph Theory and Computing, Academic Press, 1972.

[GMS] Güttler, R., Mehlhorn, K., Schneider, W., Wernet, N.: Binary Search Trees : Average and Worst Case Behaviour, GI-Jahrestagung 1976, Informatik Fachberichte Nr. 5, Springer-Verlag

[HO] Hotz, G.: Schranken für die mittlere Suchzeit bei ausgewogenen Verteilungen, Theoretical Computer Science, 1976.

[HT] Hu, T.C. and Tucker, A.C.: Optimal Computer Search Trees and Variable Length Alphabetic Codes, SIAM J. Applied Math., 21, 1971.

[K71] Knuth, D.E., 71: Optimum Binary Search Trees, Acta Informatica I, 1971.

[K73] Knuth, D.E., 73: The Art of Computer Programming, Vol. III, Addison-Wesley, 1973.

[L] van Leeuwen, J.: On the construction of Huffmann Trees, Proc. 3rd Coll. on Automata Languages and Programming, 1976, Edinburgh, University Press, Ed. S. Michaelson.

[M75] Mehlhorn, K.: Nearly Optimal Binary Search Trees, Acta Informatica, 5, 1975.

[M77a] Mehlhorn, K.: Best Possible Bounds on the Weighted Path Length of Optimum Binary Search Trees, SIAM J. of Computing, 2, 1977, 235-239.

[M77b] Mehlhorn, K.: Effiziente Algorithmen, Teubner-Verlag, 1977.

[M77c] Mehlhorn, K.: Some Remarks on Digital Searching, 3ieme Colloque de Lille, Febr. 1978

[NR] Nievergelt, Reingold : Binary Search Trees of Bounded Balance, SIAM J. of Computing, Vol. 2, Nr. 1, March, 1973.

[R] Rissanen, J.: Bounds for weighted balanced trees, IBM J. of Res. and Development, March 1973

[S] Sussenguth, E.H.: Use of tree structures for processing files, CACM 6,5 (May 63), 272-279.

[U] Unterauer, K.: Optimierung gewichteter Binärbäume zur Organisation geordneter dynamischer Dateien, Doktorarbeit, TU München, 1977.

[BM] Blum, N., Mehlhorn, K.: On the average behavior of weight-balanced trees, Technischer Bericht, FB 10 der Universität des Saarlandes, 1978