DYNAMIC BINARY SEARCH

Kurt Mehlhorn
Fachbereich 1o
Universität des Saarlandes
66 Saarbrücken
West - Germany

1. 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" [8].

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 [1,2,3,4,5,6,7,8,1o,11, 13,16]. 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

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

[2,12]. Furthermore these bounds are almost sharp for most nodes and leaves [5,17]. More precisely, for any d, $1 \leq d \leq \infty$ and $h > o$, let

$$L_h = \{j;\ a_j \leq (\log(1/\alpha_j) - h)/d\}$$

and

$$N_h = \{i; b_i \leq (\log 1/\beta_i - h + \log(1-2\cdot 2^{-d}))/d\}.$$

Then

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

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 in "logarithmic" behaviour.

In many applications the access frequencies are

a) not known in advance

b) changing over time

and therefore (nearly) optimal binary search 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 [8] : "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 occur most frequently gradually move towards the root, etc.".

With every node B_i and leaf(B_j, B_{j+1}) we associate its frequency count p_i and q_j respectively. Here p_i is the number of searches for $X=B_i$ performed so far and q_j is the number of searches for $X\in(B_j, B_{j+1})$ performed so far. We use $W=\Sigma p_i+\Sigma q_j$ for the total number of searches up to this point of time. Then $\beta_i=p_i/W(\alpha_j=q_j/W)$ is the relative access frequency of node B_i (of leaf (B_j, B_{j+1})) at this point of time. 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 $O(\log 1/\beta_i)$ $(O(\log 1/\alpha_j))$.
2. The time needed to update the tree structure is proportional to search time. This is archieved 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 [15]. In section III we introduce D-trees and show properties 1) and 2) above. In section IV we sketch some extensions : average search time, compact D-trees and insertions of new names.

II. Preliminaries : Weight Balanced Trees.

Nievergelt and Reingold introduced weight-balanced trees [15]. 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[α], for $0 \leq \alpha \leq 1/2$, if and only if

1. $\alpha \leq \rho(T) \leq 1-\alpha$
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[α] 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.

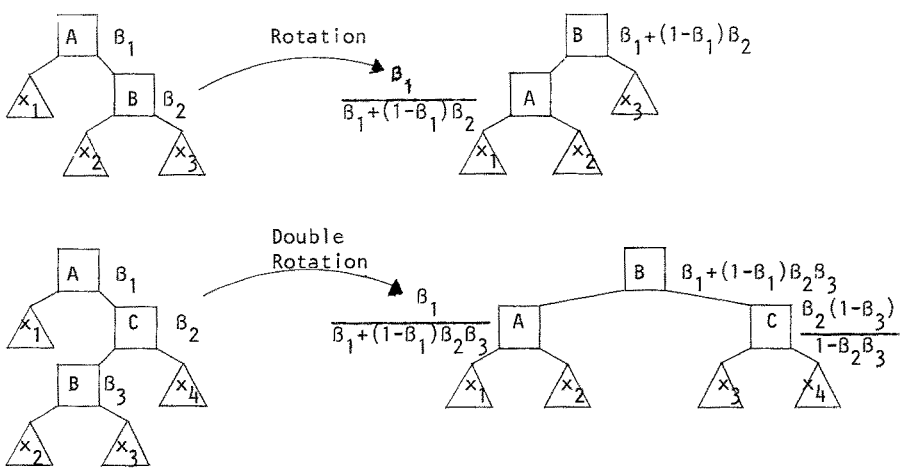


Fig. 1

Fact (Nievergelt and Reingold) : If $\alpha \leq 1-\sqrt{2}/2$ 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 insertion 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. 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 $X \in (B_j, B_{j+1})$, $0 \leq j \leq n$, performed up to now and let $W = \Sigma q_j$ be the total number of searches performed so far. From now on α is fixed, $0 < \alpha \leq 1-\sqrt{2}/2$.

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_0 leaves are labelled with $(,B_1)$, the next q_1 leaves are labelled with (B_1,B_2),... The idea of dublicating leaves appears implicitely in [3,13] and explicitely in [1o].

Definition :

a) A node v of T is a j-node, $0 \leq j \leq n$, iff 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 this 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.

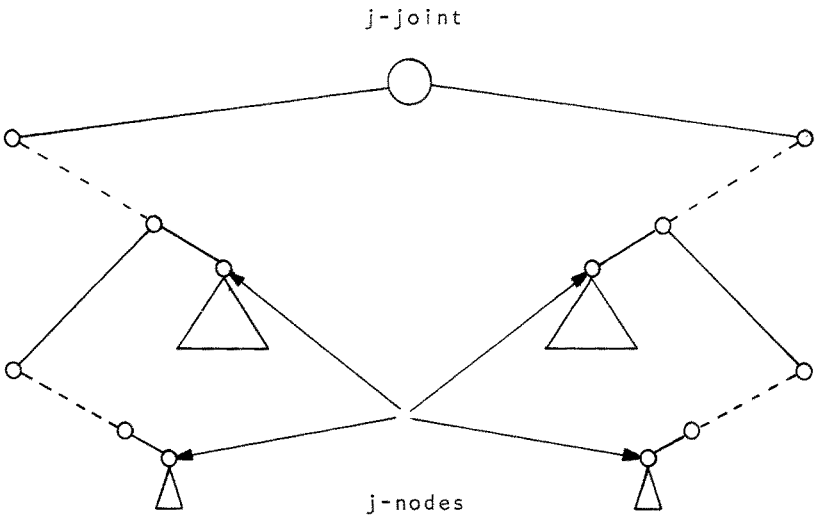


Fig. 2 : Dotted lines ... denote zero or more tree edges.

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 $q'_j \geq q''_j$ 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.

b) The thickness th(v) of a node v 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 α

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

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 the tree T. The queries are of the form

"if $X < B_j$ then go left else 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 "if $X < B_{j+1}$ then left else 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.

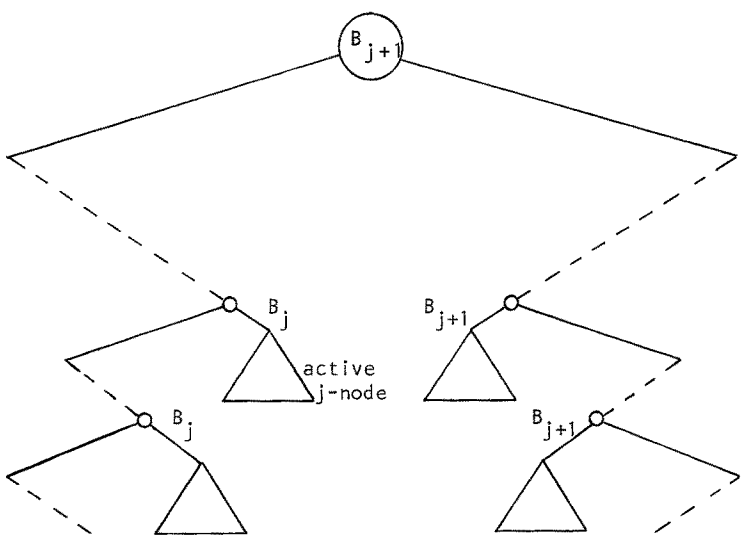


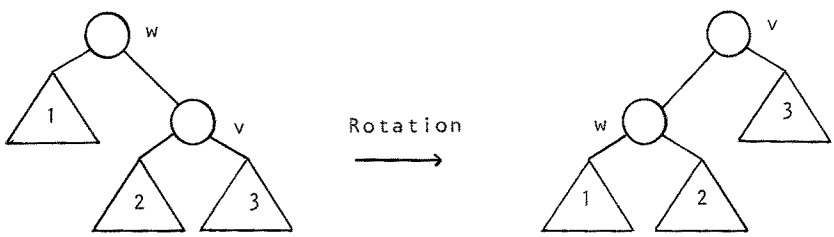
Fig. 3

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.

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



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 $^1/_3 \leq \rho(v) = \lfloor th(v)/2 \rfloor / th(v) \leq {}^1/_2$ because of $th(v) \geq 2$. A rotation will rebalance the tree.

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 is only necessary if the position (left, right) of the active j-node relative to some other node changes. This is only the case if a different node is active after the rotation as was before. But this can only happen if we rotate about the j-joint. Furthermore, in this case only the query assigned to the j-joint has to be changed. This is easy.

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 $c_1 \log 1/\alpha_j + c_2$ for some small constants c_1 and c_2. Here, α_j is the relative frequency of leaf (B_j, B_{j+1}) at the present time. 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 :

Let q_j be the numbers of searches for $X \in (B_j, B_{j+1})$, $0 \leq j \leq n$, performed up to time t_0 and let $W = \Sigma q_j$. Then at time t_0 a search for $X \in (B_j, B_{j+1})$ can be executed in time $O(\log W/q_j)$. The time needed to update the tree structure is proportional to the search time.

IV. Extension

In this section we sketch some extensions.

IV. 1. Average Search Time

Lemma 1 gives us a bound for the worst case search time in D-trees : $a_j \leq c_1 \cdot \log 1/\alpha_j + c_2$ with $c_1 = \log (1/(1-\alpha))$ and $c_2 = 1 + c_1$. From this one immediately derives a bound for the weighted path length (average search time) $P = \Sigma \alpha_j a_j$, namely

$$P \leq c_1 \cdot H(\alpha_0, \ldots, \alpha_n) + c_2$$

where $H = -\sum_{j=0}^{n} \alpha_j \log \alpha_j$ is the entropy of the distribution. A much better bound for P can be proved if we change the definition of D-tree slightly.

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 Theorem 1 are still true with this definition of active j-node. However, theorem 1 is much harder to prove. One has to keep all j-nodes and the j-joint in a doubly linked list. With each link one associates the distance of the j-node to the j-joint (if the j-node is of minimal depth) or to the father of the j-node above (otherwise). With this additional information it is again possible to change queries correctly after tree transformations. A complete proof will appear in the full paper. See also [12].

Theorem 2 :

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

$$P \leq (1/H(\alpha,1-\alpha)) \cdot [H(\alpha_0,\ldots,\alpha_n)+c]$$

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 $P\geq H(\alpha_0,\ldots,\alpha_n)$ [1,3,9,14] 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

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

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

$$\hat{P} \leq d \cdot H(\alpha_{0_1},\alpha_{0_2},\ldots,\alpha_{0m_0},\alpha_{11},\ldots,\alpha_{1m_1},\ldots)$$

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

$$H(\alpha_{0_1}, \alpha_{0_2} ,\ldots,\alpha_{0m_0},\alpha_{1_1},\ldots,\alpha_{1m_1},\ldots)$$
$$= H(\alpha_0,\ldots,\alpha_n) + \sum_{j=o}^{n} \alpha_j \cdot H(\frac{\alpha_{j_1}}{\alpha_j},\ldots,\frac{\alpha_{jm_j}}{\alpha_j}).$$

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

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

Consider nodes $v_1,\ldots,v_K$. Among these v_1 has maximal depth and v_K has minimal depth (compare Fig. 2). We have

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

for $\ell=2,\ldots,k$. Hence

$$H(\frac{\alpha_{j_1}}{\alpha'_j},\ldots, \frac{\alpha_{j_K}}{\alpha'_j}) \leq 1/\alpha$$

by repeated application of the grouping axiom. Thus

$$H\left(\frac{\alpha_{j_1}}{\alpha_j}, \ldots, \frac{\alpha_{j m_j}}{\alpha_j}\right) \leq 1 + 1/\alpha$$

and

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

IV. 2. 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 uses an immense amount of storage space. D-tree may have up to log q_0 + log q_1 + ... + log q_n j-nodes. Fig. 4 shows a D-tree for the distribution $q_0 = 2$, $q_1 = 4$, $q_2 = 20$ with $\alpha = 1/4$. The weight of the j-nodes, j = 0,1,2, is shown on the right lower corner of the j-nodes.

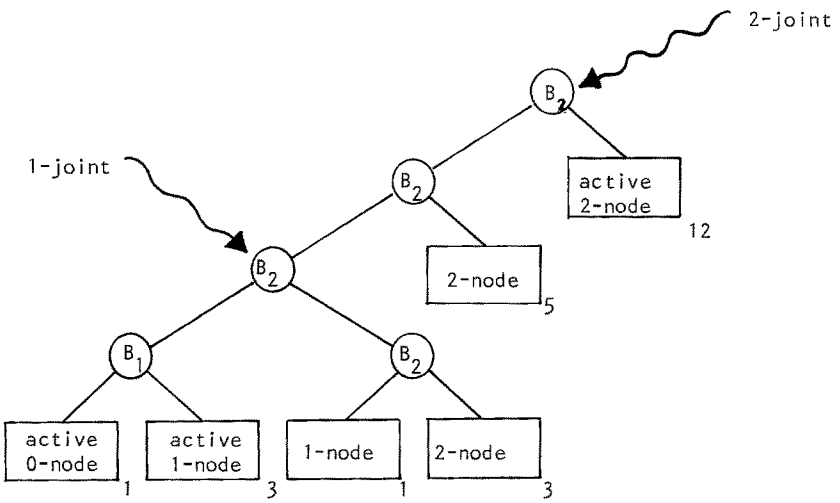


Fig. 4. A D-tree for $q_0 = 2$, $q_1 = 4$, $q_2 = 20$ with $\alpha = 1/4$.

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 behaviour as the basic structure of section III; in addition, it requires only O(n) storage cells and allows a linear time construction.

We obtain the compact tree from the tree of section III by node deletion and path compression. The skeleton of the tree is formed by the internal nodes which contain active nodes in both subtrees and by the active j-nodes

All other nodes are deleted. Applying this process to the tree of figure 4 yields :

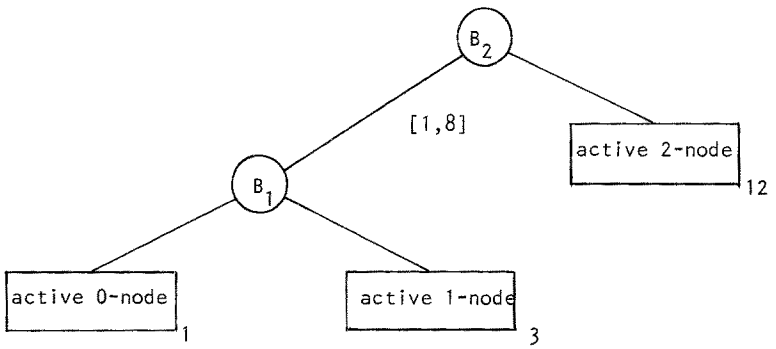


Fig. 5 : A compact D-tree for the distribution of fig. 4.

We have to remember the thickness of the active j-nodes and of the nodes deleted. We write the thickness of active j-nodes on the right lower corner of the corresponding boxes. We remember the deleted nodes by storing expressions of the form [number, number] along the compressed edges. For example, between the root B_2 of the tree and the node B_1 we deleted right subtrees representing a total of one 1-leaf and eight 2-leaves. The expression [1,8] on the right side of the edge from B_2 to B_1 is used to store this fact. Note that expressions of the form [n,m] suffice since the deleted right subtrees along a compressend path contain at most two kinds of leaves.

We interpret compact trees as concise descriptions of the trees of section III. The operations on the compact trees are performed as if they were performed on the full tree. The details are worked out in the full paper.

The compact tree for a set of n leaves consists of n leaves, n-1 internal nodes and 2(n-1) edges. It can be stored in space O(n).

Theorem 3 :

(Performance of Compact Tree). Let q_i be the number of searches for $X\in(B_i,B_{i+1})$ up to time t_0, $0\leq i\leq n$, and let $W=\Sigma q_i$. Then at time t_0 a search for $X\in(B_i,B_{i+1})$ can be carried out in time $O(\log W/q_i)$. Update time is proportional to search time. Furthermore, the compact tree needs storage space O(n).

Next we talk about initialization. Assume we are given a distribution $(q_0,\ldots,q_n)$ and a balancing factor $\alpha\leq 1-\sqrt{2}/2$. A procedure similar to the

one presented in [13] constructs the compact version of a tree of bounded balance α for the distribution in linear time.

Theorem 4 :

Given a distribution $(q_0,\ldots,q_n)$, a compact tree of bounded balance α can be constructed in time $O(n)$.

IV. 3. 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 want to drop that restriction now 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

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

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 IV.2) 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 will 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].

IV. 4. Insertions :

Suppose we want to insert a new name $B\notin\{B_1,\ldots,B_n\}$ in 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. It is easy to see that the necessary changes of the tree structure can be carried out in time proportional to the depth of the tree. The depth is bounded by O(log W) in the case of the simple D-trees of section III and by O(min (n,logW)) in the case of compact D-trees. This insertions requires time O(log W) (O(min(n log W)) resp.).

V. Conclusion :

D-trees were introduced as a method to deal with unknown and time changing frequencies. Search time in D-trees is always nearly optimal and update time is proportional to search time. A reasonable efficient way of inserting new names is also described.

The same problem was attacked in a different way by Allan and Munro [o]. The two approaches are compared in [14].

VI. References :

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

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

[2] Fredman, M.L. : Two Applications of a Probalistic Search Technique : Sorting X Y and Building Balanced Search Trees, Proc. 7th Annual ACM Symp. on Theory of Computing, 1975.

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

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

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

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

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

[8] Knuth, D.E. 71 : Optimum Binary Search Trees, Acta Informatica 1, 1971.

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

[1o] van Leeuwen, J.: On the construction of Wuffmann Trees, Proc. 3rd Coll. on Automata, Languages and Programming, 1976, Edinburgh, University Press, Ed. S. Michaelson.

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

[12] Mehlhorn, K.: Dynamic Binary Search, Techn. Bericht, FB 1o, Nr. 11, Universität des Saarlandes, 1976.

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

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

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

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