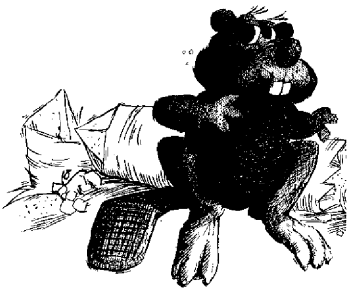
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT



On 1-Pass Top-Down Update Algorithms for Stratified Search Trees

Thomas Ottmann
Michael Schrapp
Derick Wood

Data Structuring Group
CS-83-11

May, 1983

ON 1-PASS TOP-DOWN UPDATE ALGORITHMS FOR STRATIFIED SEARCH TREES [1]

Thomas Ottmann[2]

Michael Schrapp[2]

Derick Wood[3]

ABSTRACT

In taking the work of van Leeuwen and Overmars as a basis, we present a uniform theory for the 1-pass top-down manipulation of stratified search trees. This theory allows us to consider many different classes of balanced trees and their update schemes as special cases of a new 'super'-class of balanced trees: *the stratified search trees*. However we show that weight balanced trees do not fit into this framework.

We also consider various methods of representing keys in a stratified search tree, that is routing schemes, and present a single O (log n) 1-pass top-down algorithm for insertion *and* deletion of keys in this class of trees. This algorithm can be used for many more routing schemes than those explicitly mentioned here.

1. INTRODUCTION

Traditional solutions to the dictionary problem, that is how to insert, delete or search for a given key in a set of ordered keys, use balanced search trees, see [AHU], [K], [BM], for example. The maintenance algorithms for these data structures are usually of the following kind: starting at the root perform a first pass to find the position among the leaves where a new key should be inserted or deleted and in a second pass retrace the search path towards the root and rebalance the tree if necessary. We call such algorithms *top-down bottom-up* algorithms. However, there are other algorithms for

[1] Work carried out partially under NATO Grant No. RG 155.81 and the work of the third author was partially supported by Natural Sciences and Engineering Research Council of Canada Grant No. A-5692.

[2] Institute für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6380, D-7500 Karlsruhe, W. Germany.

[3] Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

these data structures which perform the rebalancing transformations on the way down the tree. These algorithms are said to be *top-down*. An examination of the extensive literature on balanced search trees shows that many top-down algorithms have already been designed, for example [GS], [Z], [NR], [KW2], [OS1] and [OS2]. However the problems arising from this purely top-down approach are much more involved than they appear at first glance. Some of the algorithms, for example [Z], perform a first pass from the root to a leaf to find a *deepest safe node* and then a second pass from this node to a leaf. These algorithms should not really be considered as strictly *top-down*. In fact two passes are still required as with the top-down bottom-up algorithms; we say these algorithms are *2-pass top-down*. We are really interested in purely top-down algorithms, that is *1-pass top-down algorithms*.

In [NR] such an algorithm is given for weight-balanced search trees. Unfortunately this algorithm does not work correctly if redundant updates, have to be considered. An insertion is redundant, if the key to be inserted is already present, and a deletion is redundant, if the key to be deleted is not present. In this case a second top-down pass has to be carried out. Clearly, this again is in conflict with the purely 1-pass top-down idea. In [KW1] the crucial importance of a good choice of a so called *routing scheme* in this context is explained in detail. It suffices to point out that some of the 1-pass top-down algorithms, for example the one in [GS], neither work well for arbitrary routing schemes nor for a given routing scheme when only insertions or only deletions are considered. In contrast to these approaches we consider 1-pass top-down algorithms for insertion, deletion and search (including redundant insertion, and deletion) in balanced search trees with respect to different routing schemes. In [OS2] such an algorithm is given for the class of $\alpha-\beta$ trees, which are very similar to B-trees.

In [LO] a unifying theory for balanced search trees is developed. Using their framework many different classes of balanced trees can be handled similarly. The maintenance algorithms for these stratified trees, however, are top-down bottom-up, for which different routing schemes do not cause any problems.

In this paper we give a slightly modified definition of the stratified trees of [LO] and show how these trees can be updated in a 1-pass top-down manner with different routing schemes. It should further be pointed out that 1-pass top-down algorithms can give code which is simple, efficient and elegant since only one loop is needed. These algorithms have also inherent advantages in a concurrent environment as each updater need lock only a bounded portion of the tree at any time.

The remainder of this paper is organized as follows: In Section 2 the definition of stratified trees is given and the relevance of routing schemes is discussed. Sections 3 and 4, which form the major part of the paper, describe a 1-pass top-down update algorithm for stratified trees with different routing schemes. Finally in Section 5 related results are discussed and modifications as well as extensions are proposed, in particular it is proved that no subclass of the BB(α) trees forms a stratified class of trees.

2. STRATIFICATION AND ROUTING SCHEMES

Let some class X of (balanced) trees be given, then following [LO] we introduce the notion of a stratified class of trees by way of four definitions.

Definition 2.1: X is $\alpha-$*proper*, for some given non-negative integer α, if and only if for each integer $t \geq \alpha$ there is a tree in X with t leaves.

This definition requires that each set of keys of size $t \geq \alpha$ can be represented in the leaves of an X-tree with exactly t leaves. For most known classes of balanced trees α equals 1.

The next two definitions identify specific subtrees in X, which will form the required subclass, and also are good in the sense that they allow 1-pass top-down restructuring of some bounded portion of the tree along the search path. This idea will be pursued in the next section. Let Z be a set of trees, and let l_z and h_z be the least and greatest, respectively, number of leaves that members of Z can have.

Definition 2.2: Z is a $\beta-$*variety* if and only if the following conditions are satisfied

(i) all trees in Z have height β.

(ii) $1 < l_z < h_z - 1$, [†]

(iii) for each t with $l_z \leq t \leq h_z$ there is a tree in Z with exactly t leaves.

Notation 2.3: Let $T_1, \ldots, T_t$ be trees and let T be a tree with leaves $x_1, \ldots, x_t$ from left to right. We shall denote by $T[T_1, \ldots, T_t]$ the tree obtained by replacing each x_i in T by T_i $(1 \leq i \leq t)$, see Figure 2.1.

Definition 2.4: Z is a *regular* β-*variety* for X if and only if the following conditions are satisfied:

(i) Z is a β-variety

(ii) for all $t \geq \alpha$, for all T in X with t leaves and for all $T_1, \ldots, T_t$ in Z, $T[T_1, \ldots, T_t]$ is in X.

Definition 2.4 ensures that given an X-tree we obtain another X-tree by

[†] In [LO] this condition is $1 < l_z < h_z$.

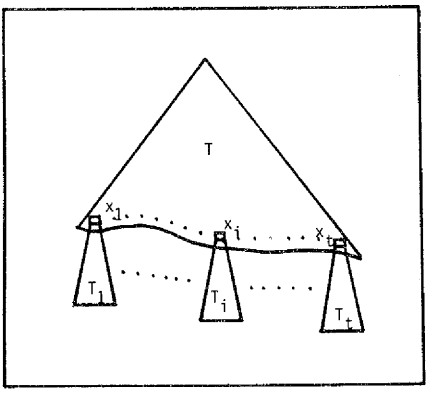


Figure 2.1

appending Z-trees to each of the leaves of the given X-tree. This holds for X-trees with at least α leaves since smaller X-trees may behave irregularly. Condition (iii) of Definition 2.2 makes it possible to obtain an X-tree with l leaves, for all l in the range $[l_z \cdot t, h_z \cdot t]$ by starting with an X-tree with t leaves, see Figure 2.2. In particular it is possible to replace one of the appended Z-trees by any other Z-tree in order to obtain a new X-tree.

We now define those subclasses of X that can be maintained in a 1-pass top-down manner by the methods of Section 3. Let X be an α-proper class of trees and Z be a regular β-variety for X ($Z \neq \emptyset$). Let

$$k = \max \left\{ \alpha\, l_z\,,\, \left\lfloor \frac{l_z - 1}{h_z - l_z} \right\rfloor \cdot l_z \right\} \text{ † } .$$

Let γ be the smallest integer such that for each t with $\alpha \leq t \leq k$ there is a T in X of height $\leq \gamma$ with exactly t leaves.

Definition 2.5: The class $S(X,Z)$ of Z-stratified trees (in X) is the smallest class of trees satisfying the following properties:

(i) Each T in X of height $\leq \gamma$ having t leaves, $\alpha \leq t \leq k$, is Z-stratified.

(ii) If T is Z-stratified and has t leaves and $T_1, \ldots, T_t$ are in Z, then $T[T_1, \ldots, T_t]$ is Z-stratified.

† In [LO] this value is $\max \left\{ \alpha\, l_z\,,\, \left\lfloor \frac{l_z - 1}{h_z - l_z} \right\rfloor \cdot l_z \right\} - 1$

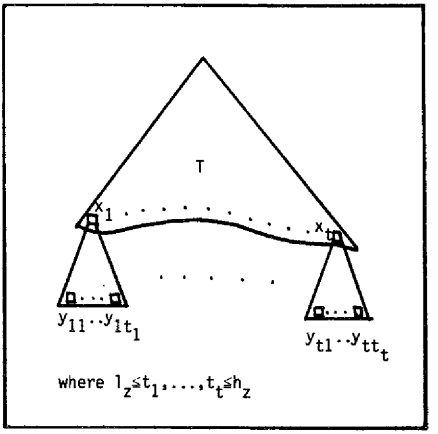


Figure 2.2

Thus for each t, $\alpha \le t \le k$ there is a tree T in the class $S(X,Z)$ with t leaves, and other trees in $S(X,Z)$ are obtained by appending layers or strata of Z-trees to some tree in $S(X,Z)$. Hence each T in $S(X,Z)$ can be decomposed as shown in Figure 2.3.

We further note that the constant k must be at least $\alpha\, l_z - 1$, since αl_z is the least value for which another stratum can be obtained from the *top* giving a new top with α leaves and a layer consisting of trees having l_z leaves. We shall explain in Section 3 why we choose k to be $\ge \alpha\, l_z$ (and not $\ge \alpha\, l_z - 1$). The value of $\left\lceil \frac{l_z - 1}{h_z - l_z} \right\rceil \cdot l_z$ is irrelevant in this context, however it is necessary for showing that the class $S(X,Z)$ of X is α-proper, which we leave for the reader to demonstrate (cf. [LO]).

In [KW1] the problem of updating routing information when rebalancing search trees has been discussed in detail. We only want to summarize and discuss briefly the results of [KW1]. In order to allow searching in a tree, routing information (also called *routers* and *separating keys*) must be provided in a uniform way. In leaf search trees these routers are associated with the internal nodes. (Note that node search trees are not really different from leaf search trees in this context.) We consider as routing information any method of associating routers with the internal nodes, provided that starting at the root node with a query value we end up at the appropriate leaf

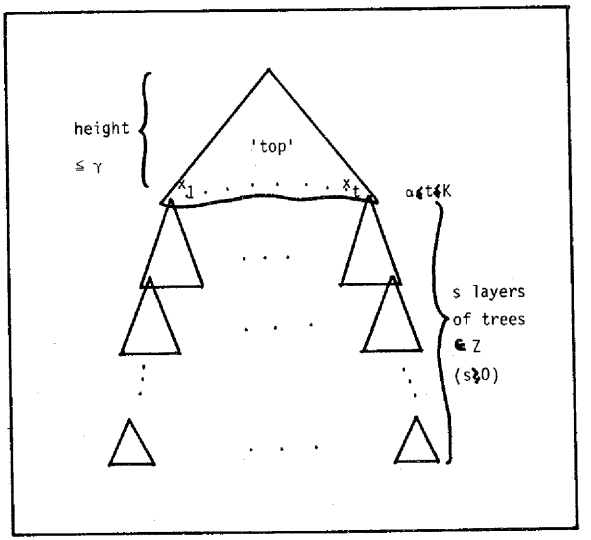


Figure 2.3

when following the routing information. Because of the generality of this idea it is not possible to formulate update algorithms for trees without referring to the specific routing scheme involved.

A method commonly used in the literature is to assign as a router the largest key in the subtree immediately to the left of the router (routers are used to separate subtrees). This method, however, is *bad* with respect to deletion. According to [KW1] bad or dirty means that restructuring an internal node may affect the routers of nodes above it, and *good* or clean means that this is not the case.

In order to facilitate the discussion in the next two sections, we restrict ourselves to one of the following routing schemes: If $p_0 r_1 p_1 r_2 \cdots r_{m-1} p_{m-1}$ represent the m roots of subtrees of an m-ary node and the $m-1$ routers and $T(p)$ denotes the subtree with root p, then for all i, $1 \le i \le m-1$:

(a) $(\le, <)$-scheme:
all keys in $T(p_{i-1}) \le r_i <$ all keys in $T(p_i)$

(b) $(<, \le)$-scheme:
all keys in $T(p_{i-1}) < r_i \le$ all keys in $T(p_i)$

(c) $(<, <)$-scheme:
all keys in $T(p_{i-1}) < r_i <$ all keys in $T(p_i)$

(d) (max, <)-scheme:
r_i = largest key in $T(p_{i-1})$ and $r_i <$ all keys in $T(p_i)$
(e) (<, min)-scheme:
r_i = smallest key in $T(p_i)$ and all keys in $T(p_{i-1}) < r_i$.

Observe that every router does indeed serve as a separator for the keys in its left and right subtree. Figures 2.4 and 2.5 give examples of binary trees with a good and an insertion-bad routing scheme, respectively.

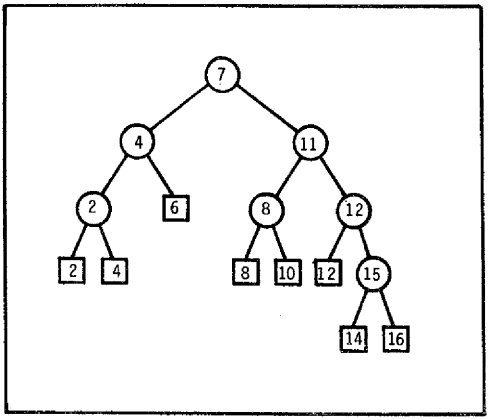


Figure 2.4

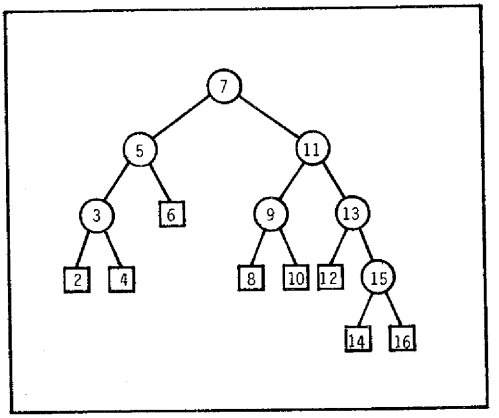


Figure 2.5

3. A 1-PASS TOP-DOWN UPDATE ALGORITHM FOR STRATIFIED TREES WITH GOOD ROUTING SCHEMES

In this section we consider only good routing schemes for this enables us to concentrate on the main idea of 1-pass top-down update algorithms. In order to achieve 1-pass top-down update algorithms we must ensure that once the frontier of the given tree has been reached, the insertion or deletion of a leaf yields a new stratified tree without affecting any internal nodes higher up in the tree. We do, however, allow restructurings within a bounded region or *window* of the leaf being considered. This implies that the Z-tree containing the leaf in question has neither the minimum nor the maximum number of leaves, that is $l_z <$ # leaves of Z-tree $< h_z$. Because of condition (ii) in Definition 2.2 such a Z-tree always exists. In order to perform the update operation we may either leave the Z-tree unchanged in the case of a redundant operation or replace it by a Z-tree with one more leaf or one fewer leaf giving a new stratified tree. So it only remains to form a Z-tree of the desired size in the proper position, that is where the addition or removal of a key has to be carried out. As we don't know the structure of the tree at the leaf level when starting at the root, we must take into account both possibilities, namely a dense or sparse tree.

We do this by building a Z-tree of the desired size higher up in the tree (near to the root) and moving it down the search path.

The following algorithm is just a formalization of this idea. Let T be the given stratified tree. Let T be decomposed into a root portion T_0 and a number of strata. Let T_i be the tree on stratum i on the search path. Note that the height of a stratum is the height of the Z-trees in the stratum, that is β, while the height of the root portion is known when constructing the tree from the empty tree. (Every time a new stratum is formed, the height of the root portion is updated. In other words the stratification of the tree is known globally.) Let $t(T)$ be the number of leaves of T. Let x be the key to be inserted or deleted, see Figure 3.1. Then *Update*(x, T) begins at the root of T (and of T_0).

Update(x,T):

```
begin
    if t(T) ≤ k then
    (I)  { Construct a new Z-stratified tree T' with t(T)+1 or t(T)−1
           leaves containing the updated set of keys}
    else
    begin
        i := 1;
        repeat
            if t(T_i) = l_z or t(T_i)=h_z then
    (II)            { Rebuild T_{i-1} and all the trees of stratum i appended
                      to T_{i-1} such that  a) T_i is on the search path, and
                                            b) l_z < t(T_i) < h_z } ;
            if i=1 then { Update height of T_0 } ;
            i:=i+1
        until { The bottom stratum has been reached with l_z<t(T_{i-1})<h_z } ;
    (III)  { Insert or delete x in T_{i-1} };
    end
end { Update }
```

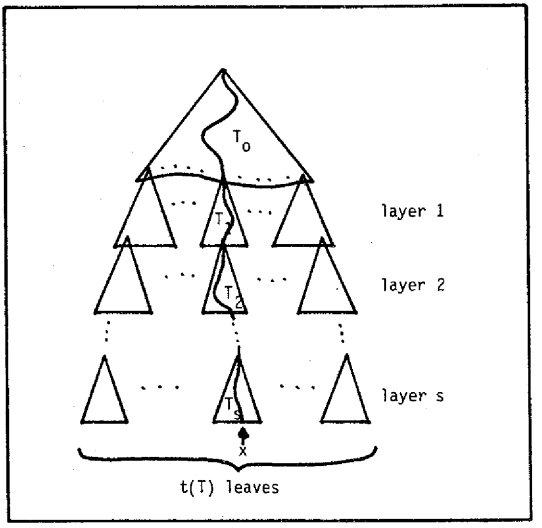


Figure 3.1

Theorem 3.1: *Let $S(X,Z)$ be a class of Z-stratified trees with a good routing scheme. Then for any tree T in $S(X,Z)$ and any key x Update(x, T) is again a tree in $S(X, Z)$.*

Proof: We consider the three possible restructurings separately.

Restructuring I:

It is clear that this restructuring is correct because $S(X, Z)$ is α-proper and we never perform an update operation yielding a tree with less than α leaves.

Restructuring II:

Let $T_i + sons$ denote the tree consisting of T_i and the Z-trees appended to the leaves of T_i .

(A) $i = 1$ {$T_0 + sons$ has to be reformed }

In this case:

$$k + 1 \leq t(T_0 + sons) \leq k \cdot h_z \quad .$$

We show that $t(T_0 + sons)$ is large enough, but not too large, to form the desired tree.

As the new root portion must have at least α leaves and all but one son must have at least l_z leaves while the remaining one must have at least l_z+1 leaves, we must have at least:

$$(\alpha-1) \cdot l_z + (l_z+1) = \alpha \cdot l_z+1$$

leaves, which is $\leq k+1$ by the choice of k . So there are enough leaves available for the desired tree. (We need the modified value of k here).

It might be necessary to form one additional stratum or layer between T_0 and the first layer. We show that in this case $t(T_0 + sons)$ is small enough to enable the desired structure to be formed.

The new root portion and the first layer have at most $k \cdot h_z$ leaves. All but one of the Z-trees in the next (second) layer has at most h_z leaves and the remaining one has at most h_z-1 leaves. Thus we have at most:

$$(k \cdot h_z-1) \cdot h_z + (h_z-1) = k \cdot h_z \cdot h_z-1$$

leaves, which is $\geq k \cdot h_z$. So we don't need to form a further layer.

It remains to show that there is no gap between no additional layer and one additional layer, so that we can form the desired structure for the whole range of $t(T_0+sons)$. We have:

$$(k-1) \cdot h_z+(h_z-1) \quad \geq \quad (\alpha l_z-1) \cdot l_z+(l_z+1)$$

Maximum # of leaves with no additional layer, that is root portion plus one layer		Minimum # of leaves with one additional layer, that is root portion plus two layers

which is equivalent to:

$$k \cdot h_z-1 \geq \alpha l_z \cdot l_z+1 \text{ , and since } k \geq \alpha\, l_z,$$

the latter inequality follows from:

$$\alpha l_z h_z-1 \geq \alpha l_z \cdot l_z+1 \quad \text{or}$$

$$\alpha l_z(h_z-l_z) \geq 2 \ .$$

Now because $\alpha \geq 1$, $l_z > 1$, $h_z > l_z+1$ this inequality holds.

(B) $i > 1$ $\{T_{i-1}+sons$ has to be reformed $\}$.

By the invariant condition $t(T_{i-1})$ must be in the range $[l_z+1,\ h_z-1]$, so we have:

$$(l_z+1) \cdot l_z \leq t(T_{i-1}+sons) \leq (h_z-1) \cdot h_z$$

Again we must show that this is enough to form the desired structure. After the restructuring the Z-tree replacing T_{i-1} might have l_z leaves and all but one of the Z-trees appended to its leaves might have l_z leaves, and the remaining one must have at least l_z+1 leaves yielding at least:

$$(l_z-1) \cdot l_z + (l_z+1) = l_z \cdot l_z+1$$

leaves, which is less than the number of leaves of $T_{i-1}+sons$. We further show that $t(T_{i-1}+sons)$ is small enough to form the desired structure. We have at most:

$$(h_z-1) \cdot h_z + (h_z-1) = h_z \cdot h_z-1 \geq (h_z-1) \cdot h_z \quad \text{leaves.}$$

Restructuring III:

When T_{i-1} is part of the bottom layer:

$$l_z+1 \leq t(T_{i-1}) \leq h_z-1$$

an insertion yields:

$$l_z+2 \leq t(T_{i-1}) \leq h_z$$

and a deletion yields:

$$l_z \leq t(T_{i-1}) \leq h_z - 2$$

giving a stratified tree once more. □

The algorithm clearly can be carried out in time $O(\log n)$ since a single pass from the root of the tree to a leaf is performed and for each node on the search path only a constant number of other nodes are visited.

It seems worthwhile to say something about the routing scheme. The algorithm does not take into account any routing information. This is valid because of the definition of good routing schemes, that is restructuring at an internal node does not affect routers above it. With this idea in mind it is clear that we may always leave correct routing information behind us when moving the window down the search path. This changes when considering bad routing schemes as will be shown in the next section.

4. A 1-PASS TOP-DOWN UPDATE ALGORITHM FOR STRATIFIED TREES WITH BAD ROUTING SCHEMES

With the routing schemes under consideration it might happen that a router, at an internal node, has to be replaced by a new value depending on the immediate successor or predecessor of the old router and the routing scheme chosen. Figure 4.1 gives an example of a binary tree with the $(<, <)$-scheme where the router with value 7 has to be replaced by a new one in the open interval (7, 10) when the key with value 7 is inserted.

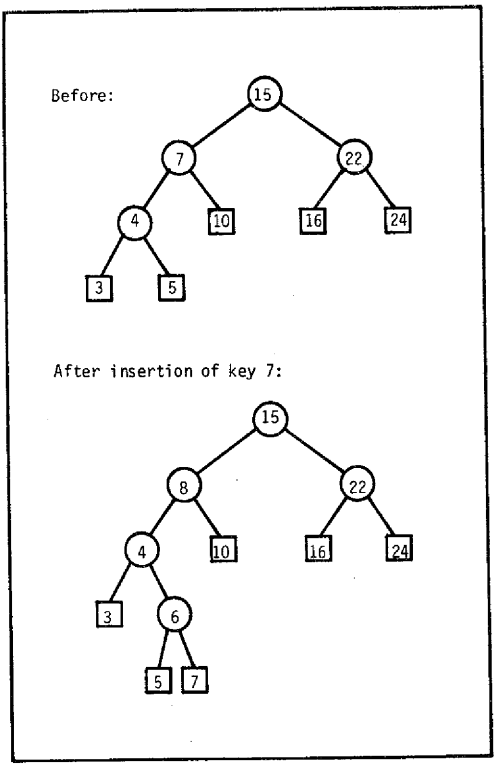


Figure 4.1

All routing schemes considered here have the property that an insertion

or deletion affects at most one router higher up in the tree and this router can be uniquely identified when running down the search path. This fact can be used to modify the algorithm of Section 3 such that an update operation will never affect routers higher up in the tree.

The only change in the modified algorithm is in the restructuring part: as soon as we have identified the router which might be affected by the update operation (it need not be affected if the operation is redundant) we restructure the tree such that the router will always remain within the window as we move down the search path. Observe that routers are not changed in value when moving downwards: we just reassemble the leaves of Z-trees on two layers in a different way. The following theorem shows that all restructurings can be carried out such that one router can be moved down the search path together with the structure we need to carry out the update operation.

Theorem 4.1: *Let $S(X, Z)$ be a class of Z-stratified trees with a bad routing scheme. Then for any tree T in $S(X, Z)$ and any key x, Update(x, T) is again a tree in $S(X, Z)$.*

Proof: The proof is similar to that of Theorem 3.1. But instead of just redistributing the leaves over the sons of T_{i-1}, we must ensure that every router occurring in T_{i-1} or in one of the Z-trees appended to it can be moved into a Z-tree appended to the tree T_{i-1}^{new} replacing T_{i-1} with this Z-tree having the properties of being on the search path and satisfying $l_z <$ number of leaves $< h_z$.

In order to show this we use the fact that a router is in the tree T_i^{new} if and only if it has a (not necessarily immediate) successor and predecessor key appearing at the leaves of T_i^{new}. For, assume that a router occurring in T_i^{new} has neither a successor nor a predecessor in T_i^{new}, then this would imply that the router separates T_i^{new} from some other tree $T_j^{new} \neq T_i^{new}$ and, thus, cannot occur in T_i^{new}.

We consider the possible restructurings separately.

Restructuring I:

It is clear that this restructuring can be carried out correctly beause $S(X, Z)$ is α-proper and we do not perform update operations yielding trees with less than α leaves. No router has to be moved downwards because the tree is small enough.

Restructuring II:

(A) $i = 1$ {T_0 + *sons* has to be reformed}

We leave this case to the reader since it is similar to both the corresponding case in the proof of Theorem 3.1 and the following case.

(B) $i > 1$ { T_{i-1} + *sons* has to be reformed }

By the invariant condition $t(T_{i-1})$ must be in the range $[l_z+1,\ h_z-1]$, so we have:

$$(l_z+1) \cdot l_z \leq t(T_{i-1} + sons) \leq (h_z-1) \cdot h_z$$

Let us assume that the leaves of T_{i-1} + *sons* are indexed consecutively from left to right. We show that for every pair of adjacent leaves $(j\ ,\ j+1)$, $1 \leq j < t(T_{i-1} + sons)$, the following holds:

We can construct new Z-trees such that:

(1) T_i^{new} contains the leaves j and $j+1$ with $l_z+1 \leq t(T_i^{new}) \leq h_z-1$,

(2) to the left of T_i^{new} there are q trees $T_1^{new}, \ldots, T_q^{new}$ ($q \geq 0$) with

$$l_z \leq t(T_n^{new}) \leq h_z\ ,\quad \min(1,q) \leq n \leq q\ ,$$

(3) to the right of T_i^{new} there are r trees $\bar{T}_1^{new}, \ldots, \bar{T}_r^{new}$ ($r \geq 0$) with

$$l_z \leq t(\bar{T}_m^{new}) \leq h_z\ ,\quad \min(1,r) \leq m \leq r\ ,\quad \text{and}$$

(4) $l_z \leq q + 1 + r \leq h_z$.

Figure 4.2 illustrates these conditions. The proof is by induction on the indices of the pair of leaves.

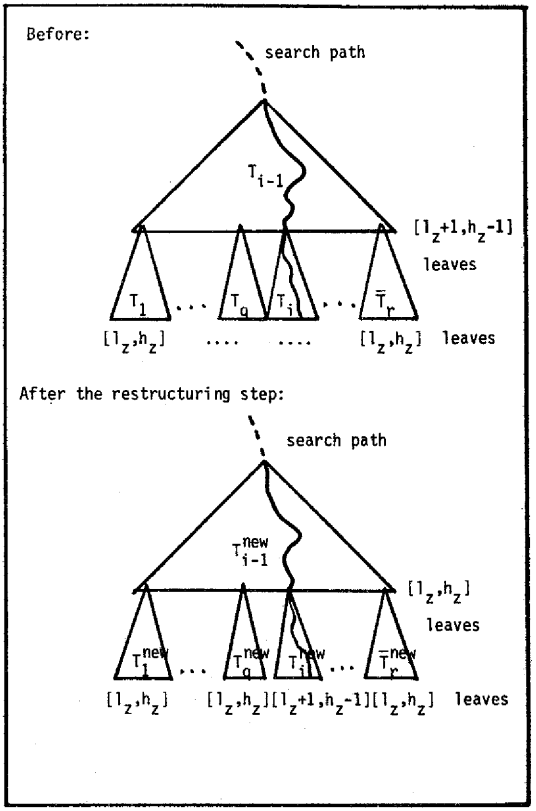


Figure 4.2

Basis: $j = 1$ and $j+1 = 2$.

Already in the proof of Theorem 3.1 we did not make any assumptions about the placement of the desired Z-tree, having neither the minimal nor the maximal number of leaves, amongst the Z-trees appended to T^{new}_{i-1}. Hence, we can form this Z-tree in the leftmost position. Now the hypothesis holds with:

$$q = 0\,; \quad (j,j+1) \quad \text{are leaves of } T^{new}_i;$$

$$l_z+1 \le t(T_l^{new}) \le h_z-1 ;$$
$$l_z \le t(\bar{T}_m^{new}) \le h_z , \quad \min(1,r) \le m \le r ;$$
$$l_z \le (1+r) = t(T_{l-1}^{new}) \le h_z .$$

Induction Hypothesis:

For all pairs $(j,j+1)$ of leaves, $1 \le j \le k$, for some $k \ge 1$, the following holds:

We can construct new Z-trees such that:

(i) $(j, j+1)$ are leaves of T_l^{new} and $l_z+1 \le t(T_l^{new}) \le h_z-1$,
(ii) $q \ge 0$, $l_z \le t(T_n^{new}) \le h_z$, $\min(1,q) \le n \le q$,
(iii) $r \ge 0$, $l_z \le t(\bar{T}_m^{new}) \le h_z$, $\min(1,r) \le m \le r$,
(iv) $l_z \le (q+1+r) = t(T_{l-1}^{new}) \le h_z$.

Induction Step:

Consider the case $j = k+1$, that is the pair of leaves $(k+1, k+2)$. In order to show that the desired new trees can be formed we distinguish the following two cases.

(a) Using the induction hypothesis for the leaves $(k, k+1)$ yields a new tree T_l^{new} containing not only the leaves $(k, k+1)$ but also the leaf $k+2$. In this case (i)-(iv) are trivially fulfilled.

(b) The tree T_l^{new} obtained by using the induction hypothesis for the leaves $(k, k+1)$ does not contain the leaf $k+2$. In this case the $(k+2)$nd leaf clearly must occur in $\bar{T}_1^{new}$ which is the right neighbor of T_l^{new}. We show that a different restructuring is possible yielding new trees $T_1^{new'} \cdots T_{q+1+r}^{new'}$ and $T_{l-1}^{new'}$ such that $T_l^{new'}$ now contains leaves $(k+1, k+2)$. (The superscript *new* always indicates trees obtained by the induction hypothesis; the superscript *new'* denotes the finally formed trees.) We distinguish the following cases.

1. $t(T_l^{new}) < h_z - 1$

1.1 $\sum_{m=1}^{r} t(\bar{T}_m^{new}) > r \cdot l_z$

Shift leaf $k+2$ to T_l^{new} and redistribute the remaining leaves to the right of T_l^{new} over the r trees, yielding trees $T_l^{new'}$ and $\bar{T}_m^{new'}$.

1.2 $\sum_{m=1}^{r} t(\bar{T}_m^{new}) = r \cdot l_z$.

Form new trees such that $t(T_{q+1}^{new'}) = l_z$ (with l_z leaves taken from T_l^{new})

$$\begin{aligned} t(T_l^{new'}) &= t(T_l^{new}) - t(T_{q+1}^{new'}) + t(\bar{T}_1^{new}) \\ &= t(T_l^{new}) - l_z + l_z = t(T_l^{new}). \end{aligned}$$

Since $k+1$ was at least the (l_z+1)-th leaf in T_l^{new} it is in $T_l^{new'}$ as is leaf $k+2$.

2. $t(T_l^{new}) = h_z - 1$.

2.1 $\sum_{m=1}^{r} t(\bar{T}_m^{new}) = r \cdot l_z$.

Form new trees such that $t(T_{q+1}^{new'}) = t(T_l^{new}) - 1 = h_z - 2 \geq l_z$ and $t(T_l^{new'}) = l_z + 1$ (leaf $k+1$ plus l_z leaves from $\bar{T}_1^{new}$.)

2.2 $\sum_{m=1}^{r} t(\bar{T}_m^{new}) > r \cdot l_z$.

2.2.1 $\sum_{n=1}^{q} t(T_n^{new}) < q \cdot h_z$.

Shift the leftmost leaf of T_l^{new} to the left and leaf $k+2$ from $\bar{T}_1^{new}$ to T_l^{new} and redistribute the leaves left and right of T_l^{new} over the q, respectively r, trees.

2.2.2 $\sum_{n=1}^{q} t(T_n^{new}) = q \cdot h_z$.

2.2.2.1 $\sum_{m=1}^{r} t(\bar{T}_m^{new}) < r \cdot h_z - 1$.

Form new trees such that $t(T_{q+1}^{new'}) = t(T_l^{new}) - 1 = h_z - 2$. Build $T_l^{new'}$ such that $l_z+1 \le t(T_l^{new'}) \le h_z-1$ (leaves $k+1$ and $k+2$ are in $T_l^{new'}$.) After this the number of leaves to the right of $T_l^{new'}$ is:

$$r \cdot l_z-(t(T_l^{new'})-1) < \sum_{m=1}^{r-1} t(\bar{T}_m^{new'}) < r \cdot h_z-1 - (t(T_l^{new'})-1).$$

On the one hand if $t(T_l^{new'}) = l_z+1$ then

$$\sum_{m=1}^{r-1} t(\bar{T}_m^{new'}) > (r-1) \cdot l_z ,$$

that is we have enough leaves left. On the other hand if $t(T_l^{new'}) = h_z-1$ then

$$\sum_{m=1}^{r-1} t(\bar{T}_m^{new'}) < (r-1) \cdot h_z+1 \quad \text{or}$$

$$\sum_{m=1}^{r-1} t(\bar{T}_m^{new'}) \le (r-1) \cdot h_z ,$$

that is we don't have too many leaves to form the desired trees.

2.2.2.2 $r \cdot h_z-1 \le \sum_{m=1}^{r} t(\bar{T}_m^{new}) \le r \cdot h_z$.

By the assumptions of this case we know that $t(T_{i-1}+sons) \le (h_z-1) \cdot h_z$, $t(T_l^{new}) = h_z-1$,

$$\sum_{n=1}^{q} t(T_n^{new}) = q \cdot h_z , \quad \text{and}$$

$$\sum_{m=1}^{r} t(\bar{T}_m^{new}) \in \{ rh_z-1 , rh_z \} .$$

We show that in this case $q+1+r < h_z$. For assume the contrary, that is $q+1+r = h_z$ then

$$q \cdot h_z + (h_z-1)+r \cdot h_z-1 \le t(T_{i-1}^{new}+sons) \le q \cdot h_z+(h_z-1)+r \cdot h_z .$$

or

$$(q+1+r) \cdot h_z - 2 \leq t(T_{l-1}^{new}+sons) \leq (q+1+r) \cdot h_z - 1$$

which is, by assumption, equivalent to

$$h_z \cdot h_z - 2 \leq t(T_{l-1}^{new}+sons) \leq h_z \cdot h_z - 1 \ .$$

Clearly, $t(T_{l-1}^{new}+sons) = t(T_{l-1}+sons)$ thus $t(T_{l-1}+sons) \leq (h_z-1) \cdot h_z$ leads to a contradiction because $h_z > 2$. This shows that T_{l-1}^{new} cannot have the maximal number of leaves. Hence, we can reform it to obtain one more leaf to the left of the root of $T_l^{new'}$. Therefore we can shift the leftmost leaf of T_l^{new} to the left and redistribute the leaves over the trees to the left of T_l^{new} and shift leaf $k+2$ from $\bar{T}_1^{new}$ to T_l^{new} to give $T_l^{new'}$. To the right of $T_l^{new'}$ we then have:

$$\sum_{m=1}^{r} t(\bar{T}_m^{new}) \geq r \cdot h_z - 2 \geq r \cdot l_z \ .$$

Restructuring (III)

Same as in the proof of Theorem 3.1. □

Theorem 4.1 states that there is a 1-pass top-down update algorithm for stratified trees with the bad routing schemes of Section 2. The following section gives an overview of related results, modifications to the algorithm and further considerations.

5. RELATED RESULTS

Many known classes of balanced trees are either stratified or contain stratified subclasses, for example AVL-trees [AVL], B-trees [BM], $\alpha-\beta$ trees [OS2], and 1-2 brother trees [OW], cf. [LO]. It is thus clear from Theorems 3.1 and 4.1 that for all of these classes there exists a 1-pass top-down update algorithm. Nevertheless it should be noted that the given algorithm is far from being optimal for any single class. In fact, the 1-pass top-down update algorithm given in [OS2] for the class of $\alpha-\beta$ trees bears evidence of this as it involves at most 4 nodes at a restructuring step instead of a node and all of its sons. However the results given here hold for virtually all classes of height balanced trees, but what about the weight-balanced trees or $BB(\alpha)$-trees of [NR]?

The following theorem shows that $BB(\alpha)$-trees are not stratified, so it remains an open problem whether or not there are 1-pass top-down update algorithms for this class. There probably isn't, but as with most negative results, it seems difficult to prove.

Theorem 5.1: *For every* α, there is no Z, *such that* Z *is a regular* β-*variety for the class of* $BB(\alpha)$-*trees.*

Corollary 5.2: *There is no stratified subclass of the class of* $BB(\alpha)$-trees.

Proof of Theorem 5.1: Let α be such that $0 < \alpha < 1/2$. For each subtree T' of T in $BB(\alpha)$ the following inequality must hold:

$$\alpha \leq \frac{l+1}{(r+1)+(l+1)} \leq 1-\alpha$$

where l and r are the number of internal nodes in the left and right subtree of T', respectively. It follows that:

$$\frac{\alpha}{1-\alpha}(l+1) \leq (r+1) \leq \frac{(1-\alpha)}{\alpha}(l+1) \quad . \tag{5.1}$$

We also need the following extension of the notation introduced in Section 2:

$$T[T_1, \ldots, T_t]^1 = T[T_1, \ldots, T_t]$$
$$T[T_1, \ldots, T_t]^s = T[T_1, \ldots, T_t]^{s-1}[T_1, \ldots, T_{t_1}]$$

So $T[T_1, \ldots, T_t]^s$ is the tree obtained by appending s layers to the leaves of

tree T.

In order to prove the theorem we assume that there exists a regular β-variety Z for the class of $BB(\alpha)$-trees. From our definition of β-varieties, c.f. Section 2, we infer that $1 < l_z < h_z - 1$. For the proof of Theorem 5.1 the weaker condition $1 < l_z < h_z$ suffices. Hence our proof also holds for β-varieties as introduced in [LO].

Thus, because Z is a regular β-variety, it follows that:

(a) $1 < l_z < h_z$ and

(b) for all t, $l_z \le t \le h_z$ implies there exists T in Z with t leaves. (5.2)

We now show that for any tree T in $BB(\alpha)$ with t leaves we can find an s and obtain a tree $T[T_1, \ldots, T_s]'$ with all T_i from Z that is not in the class $BB(\alpha)$. It follows directly that Z is not a regular β-variety for $BB(\alpha)$.

Let T in $BB(\alpha)$ have t leaves. We know that there exists at least one node v in T with two leaves as its sons. Because of (5.2) there is a tree T^1 in Z and a tree T^2 in Z with $i \ge l_z$ and $i+1 \le h_z$ leaves, respectively.

We build $T[T_1, \ldots, T_s]'$ such that the tree appended to the left leaf of node v and all trees of layers subsequently appended to that tree are equal to T^1 and the tree appended to the right leaf of node v and all trees of layers subsequently appended to that tree are equal to T^2. (See Figure 5.1.)

We now choose s to be equal to $\left\lceil \log_{(i+1)/i}\left(\frac{1-\alpha}{\alpha}\right) + 1 \right\rceil$. In the left subtree of node v we have $a = i^{\left\lceil \log_{(i+1)/i}\left(\frac{1-\alpha}{\alpha}\right)+1 \right\rceil}$ leaves. In the right subtree of node v we then have $b = (i+1)^{\left\lceil \log_{(i+1)/i}\left(\frac{1-\alpha}{\alpha}\right)+1 \right\rceil}$ leaves. Inequality (5.1) implies:

$$\frac{b}{a} \le \frac{1-\alpha}{\alpha} .$$

Thus

$$\left(\frac{i+1}{i}\right)^{\left\lceil \log_{(i+1)/i}\left(\frac{1-\alpha}{\alpha}\right)+1 \right\rceil} \le \frac{1-\alpha}{\alpha}$$

yielding

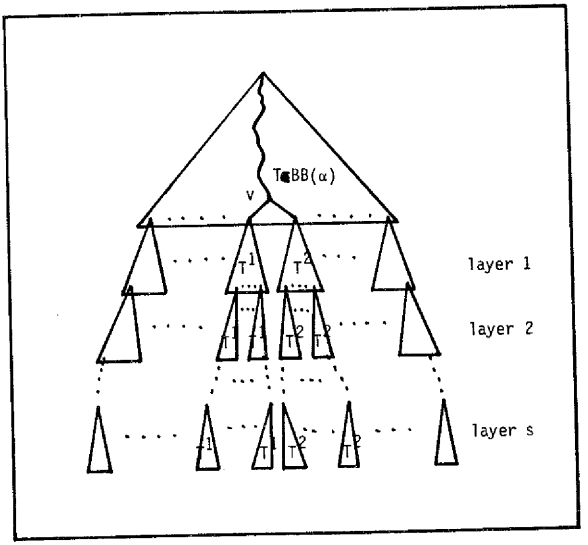


Figure 5.1

$$\frac{1-\alpha}{\alpha} + 1 \leq \frac{1-\alpha}{\alpha}$$

giving a contradiction.

So after appending s layers to tree T, node v is out of balance. □

Furthermore in addition to the algorithm given for stratified trees, 1-pass top-down algorithms can also be developed for unbalanced trees with different routing schemes. The downward movement of a router can be achieved by rotations. Another feature of our algorithm is that it deals with insertions *and* deletions. The algorithm is oblivious to the kind of update required until the leaf level is reached. This is the reason for changing the condition for Z to $1 < l_z < h_z - 1$, cf. Section 2.

If we need only an insertion algorithm we can drop this restriction and have $1 < l_z < h_z$ as in [LO], and similarly if only a deletion algorithm is required. The algorithm changes slightly in the sense that we allow a tree to have l_z leaves for insertion and h_z leaves for deletion.

In Section 4 we have considered only bad routing schemes chosen from the five given in Section 2. In these cases the values of routers depend on at

most two keys. In particular for the (<, <)-scheme these values depend on the leftmost key in the subtree immediately to the right of a router and the rightmost key in the subtree immediately to its left. Other routing schemes with the value of the router depending on more than the above mentioned keys can also be considered. As long as the router's value depends on at most k_1 keys immediately to its left and at most k_2 keys immediately to its right, where k_1 and k_2 are constants, our algorithm can be used with the following rider. The window at the bottom of the tree must be large enough to contain *all* the routers affected by the deletion or insertion of a key. Note that the height of the window is $O(\log_2 (2(k_1 + k_2)-1)) = O(1)$. In this case we always have to move at most one router downwards. Only if the value of a router depends on keys which are not within a constant range to the left or right of the router, might it be necessary to move more than one router down the search path; also the size of the window almost certainly increases in this case. It seems to us that even with such a routing scheme a 1-pass top-down update algorithm could be achieved albeit being of little practical importance.

ACKNOWLEDGEMENT:

The authors wish to thank P. Widmayer for many helpful discussions and suggestions.

6. REFERENCES

[AHU] Aho, A.V., Hopcroft, J.E. and Ullmann, J.D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1974.

[AVL] Adel'son-Vel'skii, G.M. and Landis, E.M.: An Information Organization Algorithm, *Doklady Akad. Nauk SSSR 146*, (1962), 263-266, transl. *Soviet Math. Dokl. 3*, (1962), 1259-1262.

[BM] Bayer, R. and McCreight, E.M.: Organisation and Maintenance of Large Ordered Indexes, *Acta Informatica 1*, (1972), 173-189.

[GS] Guibas, L.J. and Sedgewick, R.: A Dichromatic Framework for Balanced Trees, *Proceedings 19th Annual IEEE Symposium on Foundations of Computer Science*, Ann Arbor, October 16-18 (1978), 8-21.

[K] Knuth, D.E.: *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. 1973.

[KW1] Kwong, Y.S. and Wood, D.: On B-Trees: Routing Schemes and Concurrency, *Proceedings of the 1980 ACM/SIGMOD International Conference on Management of Data* (1980), 207-213.

[KW2] Kwong, Y.S. and Wood, D.: Some Programming Concepts for Concurrent Deletion in B-Trees, *Proceedings Twentieth Annual Allerton Conference on Communication, Control, and Computing* (1982), 472-480.

[LO] van Leeuwen, J. and Overmars, M.H.: Stratified Balanced Search Trees, *Acta Informatica 18*, (1982), 345-359.

[NR] Nievergelt, J. and Reingold, E.M.: Binary Search Trees of Bounded Balance, *SIAM Journal on Computing 2*, (1973), 33-43.

[OS1] Ottmann, Th. and Schrapp, M.: A Purely Top-Down Insertion Algorithm for 1-2 Brother Trees, University of Karlsruhe, Technical Report No. 94 (1980).

[OS2] Ottmann, Th. and Schrapp, M.: 1-Pass Top-Down Update Schemes for Balanced Search Trees, *Proceedings 7th Conference on Graphtheoretic Concepts in Computer Science WG81*, J. Mühlbacher (ed.), Carl Hansen Verlag, Vienna (1982), 279-292.

[OW] Ottmann, Th. and Wood, D.: 1-2 Brother Trees or AVL Trees Revisited, *The Computer Journal 23*, (1980), 248-255.

[Z] Zaki, A.S.: Top-Down Deletion Algorithm for Minimum-Order B-Trees, University of Washington, Technical Report. (ca. 1978)