

RAIRO

INFORMATIQUE THÉORIQUE

H. J. OLIVIÉ

A new class of balanced search trees : half-balanced binary search trees

RAIRO – Informatique théorique, tome 16, n° 1 (1982), p. 51-71.

http://www.numdam.org/item?id=ITA_1982__16_1_51_0

© AFCET, 1982, tous droits réservés.

L'accès aux archives de la revue « RAIRO – Informatique théorique » implique l'accord avec les conditions générales d'utilisation (<http://www.numdam.org/legal.php>). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

*Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques*

<http://www.numdam.org/>

A NEW CLASS OF BALANCED SEARCH TREES: HALF-BALANCED BINARY SEARCH TREES (*)

by H. J. OLIVIÉ (¹)

Communicated by M. SINTZOFF

Abstract. — *A new class of balanced binary search trees is introduced: the half-balanced binary trees. When used as a node search tree, a half-balanced binary tree containing n keys has a height at most $2 \lg(n+2) - 2$. Algorithms are given for INSERT and DELETE instructions, having time complexity $O(\lg n)$. A remarkable result is that at most one local restructuring must be performed during the deletion or the insertion of a key, whereas up to $\lg n$ local restructurings may be necessary during deletion in the other known types of balanced binary search trees.*

Résumé. — *Une nouvelle classe d'arbres binaires de recherche est présentée à l'attention du lecteur : l'arbre binaire semi-équilibré. La hauteur de l'arbre semi-équilibré avec n sommets est limitée au maximum à $2 \lg(n+2) - 2$. On utilise des algorithmes à complexité de temps $O(\lg n)$ pour traduire les instructions INSERER et EXTRAIRE, avec comme résultat remarquable qu'une seule restructuration locale suffit pour l'insertion ou l'extraction d'une clé, alors que les restructurations locales peuvent être de l'ordre $\lg n$ pour les autres types connus d'arbres binaires de recherche quasi équilibrés.*

1. INTRODUCTION

Binary search trees can be used to organize a dynamic file, i. e. a file which changes in time through insertions and deletions. In order to keep the search time short, the trees must be close to being balanced. If a tree becomes too much unbalanced, a restructuring must be performed. This may not occur too frequently and the restructuring must be easy.

Different classes of so called "balanced trees" have been proposed for such a dynamic file where SEARCH, INSERT and DELETE instructions are possible. Such a file is also known as a 'dictionary' [2]. There are three basic classes of balanced binary node search trees:

- (1) the height-balanced trees or AVL-trees [1, 7];

(*) Received June 1980.

(¹) Interstedelijke Industriële Hogeschool Antwerpen-Mechelen, Antwerpen, Belgium.

- (2) the symmetric binary B-trees or SBB-trees [3];
- (3) the binary search trees of bounded balance or $BB[\alpha]$ -trees [8].

From the latter we will only consider the case where $\alpha = 1 - \sqrt{2}/2$.

For these classes, the maximal heights are given as follows for a tree with n inner nodes (storing n keys):

$$\text{AVL: } h \leq 1.44 \lg(n+2) - 0.328;$$

$$\text{SBB: } h \leq 2 \lg(n+2) - 2;$$

$$\text{BB}(1 - \sqrt{2}/2): h \leq 2 \lg(n+1) - 1.$$

The algorithms for searching, inserting and deleting an element are all $O(h)$, so $O(\lg n)$.

Due to an insertion or a deletion of an element in a balanced search tree of a certain class, a rebalancing or restructuring operation may be necessary in order to obtain a balanced search tree of the same class. The number of rebalancing operations due to an insertion or a deletion is constant on the average as has been shown experimentally for AVL-trees [5] and SBB-trees [14], and analytically for $BB(1 - \sqrt{2}/2)$ -trees [8, 4]. The maximal number of rebalancing operations due to a deletion is $O(\lg n)$ for the three classes, and for an insertion it is also $O(\lg n)$ for SBB-trees and $BB(1 - \sqrt{2}/2)$ -trees, but only 1 for AVL-trees [12, 3]. For AVL-trees and $BB(1 - \sqrt{2}/2)$ -trees the rebalancing operations are the well-known single and double rotations [7].

In the present paper, a new class of balanced binary node search trees is introduced. Instead of making the difference of two heights as in AVL-trees, we take the quotient of two heights and require that quotient to be between certain limits. The new trees are called half-balanced binary trees, or HBB-trees.

The algorithms for insertion and deletion in an HBB-tree are also $O(\lg n)$. The height is bounded by $2 \lg(n+1) - 2$, which is comparable to the height of SBB-trees and $BB(1 - \sqrt{2}/2)$ -trees. An important property is that the insertion of an element needs at most two single rotations, equivalent to a double rotation, and that the deletion of an element needs at most three single rotations.

This means that HBB-trees are superior to both SBB-trees and $BB(1 - \sqrt{2}/2)$ -trees for insertion and deletion, and even superior to AVL-trees for deletion, when the worst case is considered.

This paper contains a further three sections and an appendix. In section 2 the definitions and the basic properties of HBB-trees are given. Section 3 contains the proofs of the procedures and the main results. Section 4 gives an indication of the further work that is done on HBB-trees, and the complete insert and delete procedures are given in the appendix.

2. DEFINITIONS AND BASIC PROPERTIES

We use *extended binary trees* [6, p. 399], where each inner node has exactly two sons, and we will call these trees simply *binary trees*.

Let T be a binary tree, and let v be a node of T . The distance of v from the root of T , i. e. the number of edges in the path from the root to v , is called the *depth* of v . All nodes with equal depth d are said to be on *level* d . The *height* of v is the number of edges in the longest path from v to a leaf. The *height of a tree* T is the height of its root.

If the elements of an ordered set S are assigned in inorder to the inner nodes of a binary tree T , then T is called a *binary node search tree* over S . The elements of S are often called keys. The number of keys must be equal to the number of inner nodes. In the sequel we will call node search trees simply *search trees*.

In the figures, inner nodes are represented by circles and leaves by squares.

If v is a node of a binary tree T , then the subtree of T with root v is denoted by T_v .

DEFINITIONS: 1. A binary tree T is called a *half-balanced binary tree*, *HBB-tree* for short, if for each node v of T the height h_v of v is not greater than twice the length s_v of a shortest path from v to a leaf: $h_v \leq 2s_v$.

2. A node v of a binary tree T is called a *half-balanced node* if the subtree T_v of T with root v is half-balanced.

To each node v of an HBB-tree two integers h_v and s_v are associated. In an implementation of an HBB-tree, these integers are stored together with the other information of node v .

Examples:

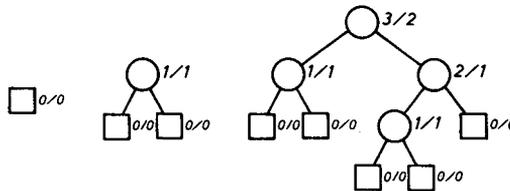


Figure 1. — Some half-balanced binary trees; the values h_v and s_v are indicated as h_v/s_v outside the symbol representing node v .

To determine the bounds of the height of an HBB-tree with a given number n of inner nodes, we introduce *minimal HBB-trees*.

DEFINITION 3: An HBB-tree with height h is called a *minimal HBB-tree* if it has the least possible number of nodes of all the HBB-trees with height h . The class of all minimal HBB-trees of height h is denoted by $\tau_{\min}(h)$.

LEMMA 1: Let $N(h)$ denote the number of inner nodes of a minimal HBB-tree with height h , then:

$$N(h) = \begin{cases} 2^{h/2+1} - 2 & \text{if } h \text{ is even;} \\ 3 \times 2^{(h-1)/2} - 2 & \text{if } h \text{ is odd.} \end{cases}$$

Proof: By induction on h .

(1) Base.

It is clear that $N(0)=0$ and $N(1)=1$.

(2) Induction step.

We suppose the hypothesis holds for $0, 1, 2, \dots, h-1$; and $h \geq 2$.

An element of $\tau_{\min}(h)$ has the structure shown in figure 2, up to some symmetry:

all the levels from level 0 (= level of the root) to level $h/2$ must be completely full, so the shorter subtree of r is the complete binary tree of height $\lceil h/2 \rceil - 1$, denoted by $T_c(\lceil h/2 \rceil - 1)$. The longer subtree must have the following properties:

- (i) it has height $h-1$;
- (ii) it is half-balanced;
- (iii) it contains the least possible number of inner nodes;

so it is a minimal HBB-tree of height $h-1$.

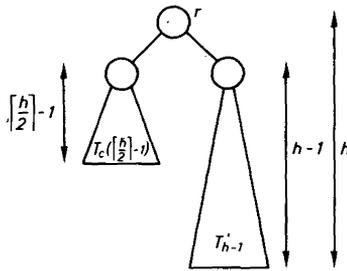


Figure 2. — The structure of a minimal HBB-tree of height h ; $T_c(\lceil h/2 \rceil - 1)$ denotes the complete binary tree of height $\lceil h/2 \rceil - 1$, and $T_{h-1} \in \tau_{\min}(h-1)$.

If h is odd, then $T_c(\lceil h/2 \rceil - 1)$ has height $(h-1)/2$, and $2^{(h-1)/2} - 1$ inner nodes.

Then:

$$N(h) = N(h-1) + 2^{h/2-1} - 1 + 1 = 2^{h/2+1} - 2.$$

If h is even, then $T_c(\lceil h/2 \rceil - 1)$ has height $(h-1)/2$, and $2^{(h-1)/2} - 1$ inner nodes.

Then:

$$N(h) = N(h-1) + 2^{(h-1)/2} - 1 + 1 = 3 \times 2^{(h-1)/2} - 2. \quad \square$$

THEOREM 1: *The height h of an HBB-tree with n inner nodes is bounded as follows: $h \leq 2 \lg(n+2) - 2$.*

Proof: Lemma 1 yields: for even h :

$$n \geq 2^{h/2+1} - 2 \Rightarrow h \leq 2 \lg(n+2) - 2$$

for odd h :

$$n \geq 3 \times 2^{(h-1)/2} - 2 \Rightarrow h \leq 2 \lg(n+2) - 2 \lg 3 + 1.$$

In both cases the hypothesis is satisfied. \square

3. HALF-BALANCED BINARY TREES AS DICTIONARIES

A dictionary [2] is a data structure for an ordered set S that can process the instructions MEMBER, INSERT and DELETE. In this section it is shown that half-balanced binary trees can be used to implement dictionaries and that there are algorithms for each of these instructions with $O(\lg n)$ time complexity, where n is the number of elements of set S .

3.1. The MEMBER instruction

The purpose of a MEMBER instruction is to determine whether a given key x is an element of set S . If T is an HBB-search tree over S , then the well-known search procedure for any binary search tree can be used. It is well known that this procedure has time complexity $O(h)$, where h is the height of the binary tree T .

3.2. The INSERT instruction

The purpose of an INSERT instruction is to insert a key x into a set S . Let again T be an HBB-search tree over S . The procedure HBB-INSERT is analogous to the insert-procedure for height-balanced trees:

- (1) Follow a search path until it is verified that key x is not yet in the tree (we must end up in a leaf, say u).
- (2) Create a new inner node v with key x and two leaves below; let $h_v = s_v = 1$ and replace leaf u by node v .
- (3) Retreat along the search path: update the h and s values and check the half-balance in each node on that path; if in a node w the half-balance is lost, restore the half-balance by a restructuring of subtree T_w .

The restructuring of a subtree consists of a "single rotation" or two consecutive single rotations, also known as a "double rotation". These rotations

are well known as they are also used in height-balanced trees [7, p. 454, 13, p. 217] and in binary trees of bounded balance [8].

The procedure as explained above contains some redundant checking: if during the retreating along the search path we arrive in a node where the h and s values do not change, we need not visit that node's ancestors any more.

It is possible to prove the following results. The proofs are omitted here because the more difficult proofs for analogous theorems are given for the deletion procedure.

THEOREM 2: *The HBB-INSERT procedure applied to an HBB-search tree with n keys yields an HBB-search tree and has-time complexity $O(\lg n)$.*

THEOREM 3: *When the HBB-INSERT procedure is applied on an HBB-search tree, then at most one restructuring must be performed, involving at most two single rotations.*

3.3. The DELETE instruction

The purpose of a DELETE instruction is to delete a key x from a set S . Let again T be an HBB-search tree over S . The procedure HBB-DELETE works as follows:

(1) Follow a search path until a node containing x is found; let v be the node with key x . If no such node is found (i.e. if we end up in a leaf), then the procedure stops.

(2) If v has two inner nodes as sons, then search for the node w of T containing the next higher key (this node is called the inorder successor of v). Copy the key of w in v and from now on consider w as the node to be deleted. (Notice that the left son of w is a leaf, namely the leaf which separates nodes v and w in the inorder sequence of the nodes of T .)

(3) Delete node v (or w) either by removing it if it has no inner sons, or by replacing it by its only inner son.

(4) Retreat along the search path: update the h and s values and check the half-balance in each node on that path; if in a node u the half-balance is lost, restore the half-balance by a restructuring of subtree T_u . As soon as a node is encountered where the h and s values do not change, the retreating may be stopped.

When a key x is successfully deleted from an HBB-tree, three phases can be distinguished:

- (1) There is an HBB-search tree containing key x ;
- (2) A node containing key x has been removed from the tree; the tree obtained

then might have lost the half-balance property in some of its nodes;

(3) If necessary the tree is restructured and an HBB-tree is obtained.

NOTATIONS: 1. \mathcal{HBB} denotes the class of HBB-trees.

2. φv denotes the father of node v .

3. Let T be a binary tree, T_1 the left subtree of the root and T_2 the right subtree of the root, then we write:

$$T = \langle T_1, T_2 \rangle.$$

By T_{11} we denote the left subtree of the root of T_1 , and by T_{12} the right subtree of the root of T_1 , and more generally:

$$T_\alpha = \langle T_{\alpha 1}, T_{\alpha 2} \rangle \quad \text{for } \alpha \in \{1, 2\}^*.$$

h_α denotes the height of T_α , s_α denotes the s -value of the root of T_α .

DEFINITION 4: Let T be a tree in phase (2), obtained from an HBB-tree by removing a node v . Then the closest ancestor y of φv (eventually φv itself) which is not half-balanced is called a *critical node* of T . The (sub) tree T_y with root y is called a *critical (sub) tree*, abbreviated as *CR-tree*.

We try to find a method to restructure a critical (sub) tree. Therefore we investigate the properties of a critical tree first.

LEMMA 2: Let v be a node of a binary tree with sons u and w .

Then v is half-balanced iff:

$$\left\{ \begin{array}{l} u \text{ and } w \text{ are both half-balanced, and} \\ \frac{h_u + 1}{s_u + 1} \leq 2 \quad \text{and} \quad \frac{h_w + 1}{s_w + 1} \leq 2. \end{array} \right.$$

Proof: From the definition 2 we know that v is half-balanced iff u and w are both half-balanced, and:

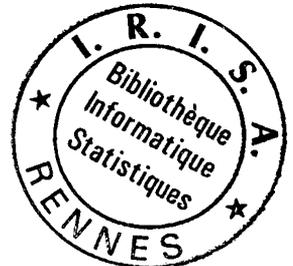
$$\frac{h_v}{s_v} \leq 2. \tag{1}$$

Since:

$$h_v = \max(h_u, h_w) + 1 \quad \text{and} \quad s_v = \min(s_u, s_w) + 1,$$

$$(1) \Leftrightarrow \frac{h_u + 1}{s_u + 1} \leq 2 \quad \text{and} \quad \frac{h_w + 1}{s_w + 1} \leq 2$$

$$\text{and} \quad \frac{h_w + 1}{s_u + 1} \leq 2 \quad \text{and} \quad \frac{h_w + 1}{s_w + 1} \leq 2.$$



When u and w are both half-balanced the first and the last inequality are satisfied, so the hypothesis follows. \square

LEMMA 3: If $T = \langle T_1, T_2 \rangle$ is a critical tree obtained by a deletion of a node from T_2 , then the following properties hold in T :

- (P 1) $h_1 \leq 2s_2 + 3,$
- (P 2) $h_1 > 2s_2 + 1 \geq h_2 + 1,$
- (P 3) $\max(h_{11}, h_{12}) > h_2 + 1.$

Proof: Before the removal of a node from T_2 , the tree is half-balanced; after that removal the tree T is critical. This is only possible if the value s_2 has decreased by one.

So:

$$\frac{h_1 + 1}{s_2 + 2} \leq 2 \tag{1}$$

and:

$$\frac{h_1 + 1}{s_2 + 1} > 2 \tag{2}$$

- (1) $\Leftrightarrow h_1 + 1 \leq 2s_2 + 4 \Leftrightarrow h_1 \leq 2s_2 + 3$ (P 1).
- (2) $\Leftrightarrow h_1 + 1 > 2s_2 + 2 \Leftrightarrow h_1 > 2s_2 + 1,$

and because $2s_2 \geq h_2$ as $T_2 \in \mathcal{HBB}$, (P 2) follows.

Since $h_1 = \max(h_{11}, h_{12})$, and $h_1 > h_2 + 1$ by (P 2), (P 3) follows. \square

The transformations used to restructure a CR-tree are known as “single right rotation” and “single left rotation” (SRR and SLR). The single right rotation on T can be described as follows:

$$\text{SRR}(\langle \langle T_{11}, T_{12} \rangle, T_2 \rangle) = \langle T_{11}, \langle T_{12}, T_2 \rangle \rangle$$

and it is illustrated in figure 3. The single left rotation is the mirror image of the single right rotation.

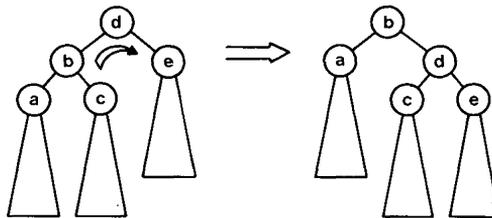


Figure 3. – A single right rotation.

The following lemma gives us the condition when a single right rotation will restructure a (critical) tree.

LEMMA 4: If $T = \langle T_1, T_2 \rangle$ with $T_1 = \langle T_{11}, T_{12} \rangle$ and $T_1, T_2 \in \mathcal{HBB}$ then $\langle T_{11}, \langle T_{12}, T_2 \rangle \rangle$ iff:

$$\left\{ \begin{array}{l} h_{12} \leq 2 s_2 + 1 \\ \text{and:} \\ \max(h_{12}, h_2) \leq 2 s_{11}. \end{array} \right. \quad \begin{array}{l} \text{(C 1)} \\ \\ \text{(C 2)} \end{array}$$

Proof: $\langle T_{11}, \langle T_{12}, T_2 \rangle \rangle$ can only be half-balanced if $\langle T_{12}, T_2 \rangle$ is half-balanced.

$$(a) \quad \langle T_{12}, T_2 \rangle \in \mathcal{HBB} \text{ iff } \left\{ \begin{array}{l} \frac{h_{12} + 1}{s_2 + 1} \leq 2 \\ \text{and:} \\ \frac{h_2 + 1}{s_{12} + 1} \leq 2 \end{array} \right. \quad \begin{array}{l} \text{(1)} \\ \\ \text{(2)} \end{array}$$

(by lemma 2).

$$(1) \Leftrightarrow h_{12} + 1 \leq 2 s_2 + 2 \Leftrightarrow h_{12} \leq 2 s_2 + 1, \quad \text{(C 1)}$$

(2) is true because:

$$\frac{h_2 + 1}{s_{12} + 1} \underset{(P3)}{<} \frac{\max(h_{11}, h_{12}) + 1}{s_{12} + 1} \leq 2,$$

since $T_{12}, T_1 \in \mathcal{HBB}$.

(b) Provided that $\langle T_{12}, T_2 \rangle \in \mathcal{HBB}$, then $\langle T_{11}, \langle T_{12}, T_2 \rangle \rangle \in \mathcal{HBB}$ iff:

$$\left\{ \begin{array}{l} \frac{h_{11} + 1}{s_{12} + 2} \leq 2 \\ \text{and:} \\ \frac{h_{11} + 1}{s_2 + 2} \leq 2 \\ \text{and:} \\ \frac{h_{12} + 2}{s_{11} + 1} \leq 2 \\ \text{and:} \\ \frac{h_2 + 2}{s_{11} + 1} \leq 2 \end{array} \right. \quad \begin{array}{l} \text{(3)} \\ \\ \text{(4)} \\ \\ \text{(5)} \\ \\ \text{(6)} \end{array}$$

(by lemma 2).

(3) is true because $T_1 \in \mathcal{HBB}$.

(4) $\Leftrightarrow h_{11} \leq 2s_2 + 3$, which is true because:

$$h_{11} \leq h_1 - 1 \quad \text{and} \quad h_1 \leq 2s_2 + 3. \quad (\text{P } 1)$$

$$\left. \begin{array}{l} (5) \Leftrightarrow h_{12} \leq 2s_{11} \\ (6) \Leftrightarrow h_2 \leq 2s_{11} \end{array} \right\} \Leftrightarrow \max(h_{12}, h_2) \leq 2s_{11}. \quad \square \quad (\text{C } 2)$$

The restructuring procedure for a critical tree T , obtained by a deletion of a node in the right subtree, is given in the flow-chart of figure 4.

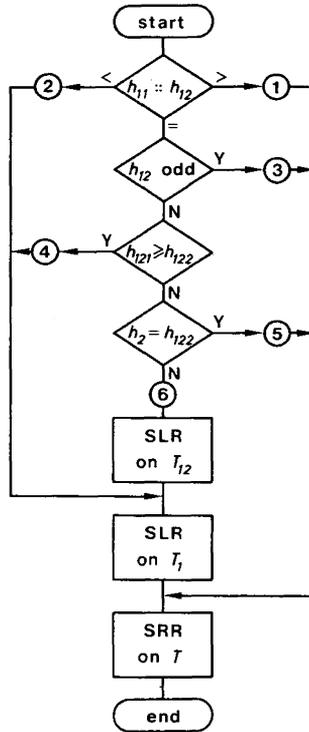


Figure 4. — Restructuring of critical tree T .

The proof for the restructuring procedure is long and tedious. We will not give the complete proof, it can be found in [9] and [10]. However we will give an indication how this restructuring procedure can be found.

Lemma 4 gives the condition when a SRR restructures the critical tree T . This condition is also met if $T_1, T_2 \in \mathcal{HBB}$ and:

- (a) $h_{12} \leq h_{11}$,
- (b) $h_2 \leq h_{11}$,
- (c) $h_{12} \leq 2s_2 + 1$,

because $h_{11} \leq 2s_{11}$ as $T_{11} \in \mathcal{HBB}$.

We want to find other conditions which are simpler to specify.

1. Let us firstly try $h_{12} \leq h_{11}$ then (b) is true because:

$$h_{11} = \max(h_{12}, h_{11}) = h_1 - 1 \quad \text{and} \quad h_2 > h_1 - 1 \quad \text{by (P2).}$$

(c) is not always true:

$$h_{12} \leq h_1 - 1 \stackrel{(P1)}{\leq} 2s_2 + 3 - 1 = 2s_2 + 2.$$

So we have to refine the assumption.

1.1. If $h_{12} < h_{11}$ then:

$$h_{12} \leq h_1 - 2 \stackrel{(P1)}{\leq} 2s_2 + 3 - 2 = 2s_2 + 1$$

and (c) is true.

This corresponds to branch 1 of the flow-chart.

1.2. If $h_{12} = h_{11}$ and h_{12} is odd then:

$$h_{12} = h_1 - 1 \stackrel{(P1)}{\leq} 2s_2 + 2$$

yields:

$$h_{12} \leq 2s_2 + 1 \quad \text{since } h_{12} \text{ is odd.}$$

This corresponds to branch 3 of the flow-chart.

1.3. If $h_{12} = h_{11}$ and h_{12} is even:

$$(c) \quad h_{12} \leq 2s_2 + 1$$

will be satisfied if:

$$h_{12} \leq h_2 + 1$$

or:

$$\max(h_{121}, h_{122}) \leq h_2 + 1,$$

in particular if:

$$h_{121} < h_{122} = h_2.$$

This corresponds to branch 5 of the flow-chart.

2. A single right rotation will not always be sufficient to restructure T . This is illustrated in figure 5.

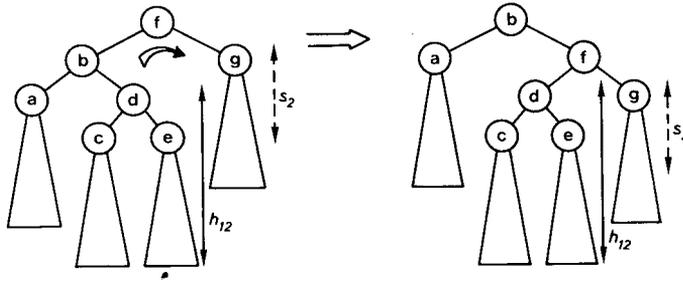


Figure 5. — After a single right rotation node f might be critical.

If:

$$\frac{h_{12} + 2}{s_2 + 2} = 2$$

(the tree is HBB before the removing of a node) then:

$$\frac{h_{12} + 2}{s_2 + 1} > 2$$

(the tree T is critical after the removing of a node) and:

$$\frac{h_{12} + 1}{s_2 + 1} > 2$$

(the tree is not HBB after a SRR).

2. 1. We can firstly try to obtain a tree where $h_{12} < h_{11}$ by a single left rotation in subtree T_1 , as illustrated in figure 6.

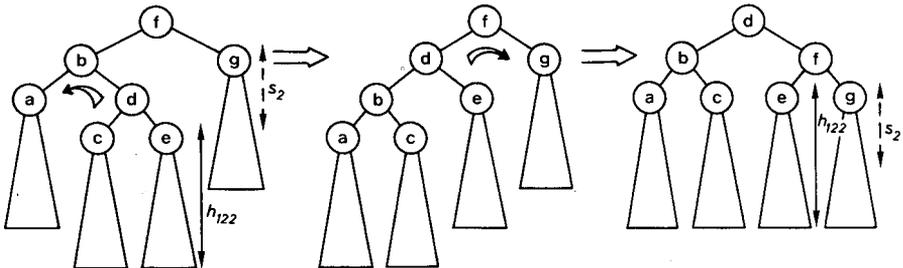


Figure 6. — A double right rotation on T , consisting of a single left rotation on T_1 , followed by a single right rotation on T .

Then if:

$$\frac{h_{122} + 3}{s_2 + 2} = 2,$$

we will have after the removal of a node and the two single rotations:

$$\frac{h_{122} + 1}{s_2 + 1} = 2,$$

so the subtree with root f is an HBB-tree. The restructuring consisting of these two consecutive single rotations is called a *double right rotation*.

2.2. However, there is still a possibility that node b is not half-balanced after this double rotation. In that case three consecutive single rotations have to be done. This restructuring is called a *triple right rotation* and it is illustrated in figure 7.

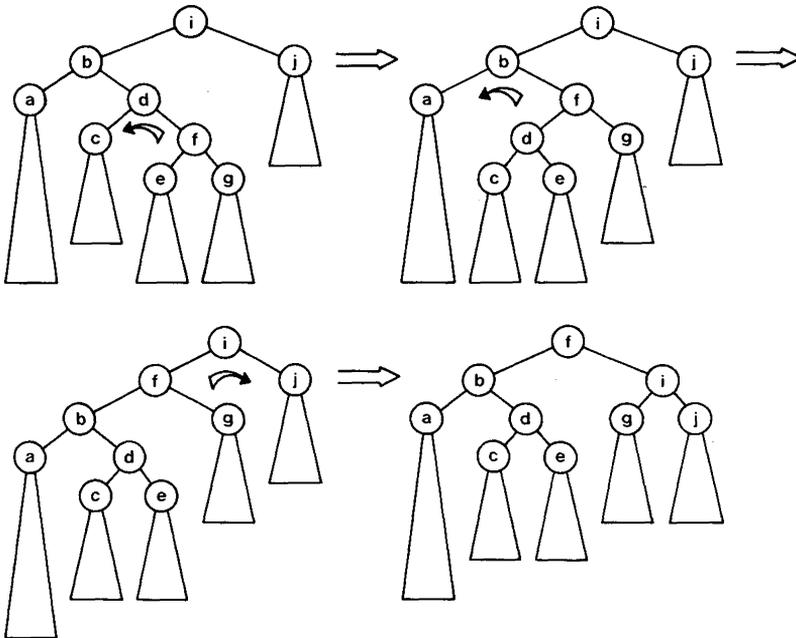


Figure 7. — A triple right rotation on T .

THEOREM 4: *The HBB-DELETE procedure applied to delete a key from an HBB search tree with n keys yields an HBB search tree, and has time complexity $O(\lg n)$.*

Proof: In [9] and [10] it is proved that the HBB-DELETE procedure applied on an HBB search tree yields an HBB-tree. The latter is again a search tree because a single rotation maintains the inorder of the keys.

During the execution of HBB-DELETE we start at the root and follow a path down, at most to a leaf; after the removing of a node we climb up again towards the root, updating h and s factors, and eventually we perform a restructuring. During this DELETE-procedure we visit at most $2h$ nodes, and all actions taken in the nodes need only constant time, so the time complexity of HBB-DELETE is $O(h)$, which means $O(\lg n)$ when the tree contains n keys, according to theorem 1. \square

THEOREM 5: *When the HBB-DELETE procedure is applied to delete an element from an HBB-search tree, then at most three single rotations must be performed.*

Proof: Let T be a critical subtree with root r , with height h , and with shortest pathlength from the root to a leaf s . We know that:

$$2s < h \leq 2(s+1).$$

After the restructuring, T is replaced by a subtree T' with root r' , height h' , and shortest pathlength from the root to a leaf s' . Then:

$$h' \leq h,$$

$$s' = s + 1,$$

and:

$$h' \leq 2s'.$$

If before the restructuring there is a node w above the critical node r such that:

$$2s_w < h_w \leq 2(s_w + 1),$$

with:

$$s_w = s + c,$$

where c is a positive integer, then the shortest path from w to a leaf contains r .

After the restructuring, r is replaced by r' , and if h'_w, s'_w denote the new h -, s -values of w , then:

$$s'_w = s' + c = s + c + 1 = s_w + 1,$$

$$h'_w \leq h_w.$$

So w is half-balanced then.

This means that after the restructuring of a critical subtree, the entire tree is half-balanced, and we knew already that a restructuring needs at most three single rotations. \square

4. CONCLUSIONS AND INDICATIONS OF FURTHER WORK

The new class of half-balanced binary trees was introduced and procedures to search, to insert and to delete a key in an HBB-node search tree with n keys were given. These algorithms have $O(\lg n)$ time complexity. The maximal height is about $2 \lg n$, which is the same as for SBB-trees [3], $BB(1 - \sqrt{2}/2)$ -trees [8] and son-trees [11].

During insertion in an HBB-tree at most one local restructuring, involving at most two single rotations, must be performed. The insertion algorithm for height-balanced trees has the same property. During deletion of an element from an HBB-tree also at most one local restructuring, involving this time at most three single rotations, must be performed. This is a remarkable result as no deletion algorithm of any already known class of balanced trees has that property.

HBB-trees have been investigated further in [10]: their correspondences with SBB-trees, son-trees, AVL-trees and $BB(\alpha)$ -trees, and their behaviour under random insertions, both analytically and empirically.

ACKNOWLEDGMENT

The author is grateful to J. Paredaens for reading different drafts of this paper and suggesting substantial improvements, and to Danielle Simons and Marc Asselberg for testing the procedures. He also thanks the referees who helped to improve the presentation of some proofs.

APPENDIX

In the procedures for HBB-trees a node is represented by a record with five fields, defined as follows:

```

type ref = ↑ node;
node = record key      : integer;
              left, right : ref;
              h, s       : integer
end

```

The empty tree is represented by a node "leaf", with:

```
leaf↑.key   = 0
leaf↑.left  = nil
leaf↑.right = nil
leaf↑.h     = 0
leaf↑.s     = 0
```

All the pointers to external nodes also point to that node.

1. The insert-procedure

The procedure HBBINSERT has three parameters:

x of type integer, the key to be inserted.

p of type ref, the pointer to the actual node.

k of type boolean, $k = true$ if a key has been inserted into subtree T_p ; $k = false$ if no key has been inserted into subtree T_p .

For the first call of HBBINSER, p points to the root of the tree T , x is the key to be inserted, and k is *false*.

```
procedure HBBINSERT (x : integer; var p : ref;
                    var k : integer);
  { insertion of key x into HBB-tree with root p }
procedure UPDATE (var p : ref);
  { updating of h and s values of node p }
begin
  p↑.h := max(p↑.left↑.h, p↑.right↑.h) + 1;
  p↑.s := min(p↑.left↑.s, p↑.right↑.s) + 1
end { update };
procedure RIGHTROT (var p : ref);
  { right rotation on node p }
  var p2 : ref;
begin
  p2 := p↑.left;
  p↑.left := p2↑.right;
  p2↑.right := p;
  UPDATE (p);
  UPDATE (p2);
  p := p2
end { rightrot };
procedure LEFTROT (var p : ref);
  { left rotation on node p : analogous to RIGHTROT }
end { leftrot };
procedure RIGHTRES (var p : ref);
  { restructuring of a tree with root p which is critical due to an insertion into its right subtree }
  var p1 : ref;
begin
  p1 := p↑.right;
  if p1↑.right↑.h > p1↑.left↑.h
  then LEFTROT (p)
```

```

else begin
    RIGHTROT (p1);
    p↑.right := p1;
    LEFTROT (p)
end
end { righ tres };
procedure LEFTRES (var p : ref);
{ restructuring of a tree with root p which is critical due to an insertion into its left subtree:
analogous to RIGHTRES }
end { leftres };
begin { insert }
    if p = leaf
        then begin { key x is not in tree, create a new
            node with key x }
            new (p);
            k := true;
            with p↑ do begin
                key := x;
                left := leaf;
                right := leaf;
                h := 1;
                s := 1
            end
        end
    else if x < p↑.key
        then begin { insert key x in left subtree of p }
            HBBINSERT (x, p↑.left, k);
            if k and (p↑.s ≠ min (p↑.left↑.s, p↑.right↑.s) + 1
                or p↑.h ≠ max (p↑.left↑.h, p↑.right↑.h) + 1)
                then begin
                    UPDATE (p);
                    if p↑.h > 2 * p↑.s
                        then LEFTRES (p)
                end;
        end
    else if x > p↑.key
        then begin { insert key x in right subtree of p }
            HBBINSERT (x, p↑.right, k);
            if k and (p↑.s ≠ min (p↑.left↑.s, p↑.right↑.s) + 1
                or p↑.h ≠ max (p↑.left↑.h, p↑.right↑.h) + 1)
                then begin
                    UPDATE (p);
                    if p↑.h > 2 * p↑.s
                        then RIGHTRES (p)
                end;
        end
    else { x = p↑.key, so key x is already in tree }
        k := false
    end { HBBinsert };

```

2. The delete-procedure

The procedure HBBDELETE has three parameters :

- x of type integer, the key to be deleted;
- p of type ref, the pointer to the actual node;

k of type boolean, $k = true$ if a key has been deleted from subtree T_p ; $k = false$ if no key has been deleted from subtree T_p .

For the first call of HBBDELETE, p points to the root of the tree T , x is the key to be deleted, and k is *false*.

```

procedure HBBDELETE ( $x$  : integer; var  $p$  : ref;
                    var  $k$  : integer);
  { deletion of key  $x$  from HBB-tree with root  $p$  }
  var  $q$  : ref;
procedure UPDATE
procedure RIGHTROT }      see HBBINSERT
procedure LEFTROT  }
procedure RESDELRIGHT (var  $p$  : ref);
  { restructuring of a tree with root  $p$  which is critical due to a deletion from its right subtree }
  var  $p1, p2$  : ref;
       $h1$  : integer;
begin
   $p1 := p \uparrow .left$ ;
  if  $p1 \uparrow .left \uparrow .h > p1 \uparrow .right \uparrow .h$ 
  then { case 1 }
    RIGHTROT ( $p$ )
  else if  $p1 \uparrow .left \uparrow .h < p1 \uparrow .right \uparrow .h$ 
  then begin
    LEFTROT ( $p1$ );
     $p \uparrow .left := p1$ ;
    RIGHTROT ( $p$ )
  end
  else begin {  $p1 \uparrow .left \uparrow .h = p1 \uparrow .right \uparrow .h$  }
     $p2 := p1 \uparrow .right$ ;
    if odd( $p2 \uparrow .h$ )
    then { case 3 }
      RIGHTROT ( $p$ )
    else if  $p2 \uparrow .left \uparrow .h > p2 \uparrow .right \uparrow .h$ 
    then begin { case 4 }
      LEFTROT ( $p1$ );
       $p \uparrow .left := p1$ ;
      RIGHTROT ( $p$ )
    end
    else if  $p \uparrow .right \uparrow .h = p2 \uparrow .right \uparrow .h$ 
    then { case 5 }
      RIGHTROT ( $p$ )
    else begin { case 6 }
      LEFTROT ( $p2$ );
       $p1 \uparrow .right := p2$ ;
      LEFTROT ( $p1$ );
       $p \uparrow .left := p1$ ;
      RIGHTROT ( $p$ )
    end
  end
end { resdelright }
procedure RESDELLEFT (var  $p$  : ref);
  { restructuring of a tree with root  $p$  which is critical due to a deletion from its left subtree: analogous
  to RESDELRIGHT }
  end { resdelleft };
procedure DEL (var  $r, p$  : ref; var  $k$  : boolean);
  { search for inorder successor of node  $p$  }

```

```

begin
  if  $r \uparrow . \text{left} \neq \text{leaf}$ 
  then begin
    DEL ( $r \uparrow . \text{left}$ ,  $p$ ,  $k$ );
    if  $k$  and ( $r \uparrow . s \neq \min(r \uparrow . \text{left} \uparrow . s, r \uparrow . \text{right} \uparrow . s) + 1$ 
      or  $r \uparrow . h \neq \max(r \uparrow . \text{left} \uparrow . h, r \uparrow . \text{right} \uparrow . h) + 1$ )
    then begin
      UPDATE ( $r$ );
      if  $r \uparrow . h > 2 * r \uparrow . s$ 
      then RESEDELLEFT ( $r$ )
    end.
  end
  else begin
     $p \uparrow . \text{key} := r \uparrow . \text{key}$ ;
     $q := r$ ;
     $r := r \uparrow . \text{right}$ ;
     $k := \text{true}$ 
  end
end { del };
begin { delete }
  if  $p = \text{leaf}$ 
  then { key  $x$  is not in tree }
     $k := \text{false}$ 
  else if  $x < p \uparrow . \text{key}$ 
  then
    begin { delete key  $x$  from left subtree of  $p$  }
      HBBDELETE ( $x$ ,  $p \uparrow . \text{left}$ ,  $k$ );
      if  $k$  and ( $p \uparrow . s \neq \min(p \uparrow . \text{left} \uparrow . s, p \uparrow . \text{right} \uparrow . s) + 1$ 
        or  $p \uparrow . h \neq \max(p \uparrow . \text{left} \uparrow . h, p \uparrow . \text{right} \uparrow . h) + 1$ )
      then
        begin
          UPDATE ( $p$ );
          if  $p \uparrow . h > 2 * p \uparrow . s$ 
          then RESEDELLEFT ( $p$ )
        end
      end
    end
  else if  $x > p \uparrow . \text{key}$ 
  then
    begin { delete key  $x$  from right subtree of  $p$  }
      HBBDELETE ( $x$ ,  $p \uparrow . \text{right}$ ,  $k$ );
      if  $k$  and ( $p \uparrow . s \neq \min(p \uparrow . \text{left} \uparrow . s, p \uparrow . \text{right} \uparrow . s) + 1$ 
        or  $p \uparrow . h \neq \max(p \uparrow . \text{left} \uparrow . h, p \uparrow . \text{right} \uparrow . h) + 1$ )
      then
        begin
          UPDATE ( $p$ );
          if  $p \uparrow . h > 2 * p \uparrow . s$ 
          then RESEDELRIGHT ( $p$ )
        end
      end
    end
  else
    begin { key  $x$  is in tree, remove it }
       $q := p$ ;
      if  $q \uparrow . \text{right} = \text{leaf}$ 
      then
        begin
           $p := q \uparrow . \text{left}$ ;
           $k := \text{true}$ 
        end
      else if  $q \uparrow . \text{left} = \text{leaf}$ 
      then

```

```

begin
  p := q ↑ .right;
  k := true
end
else
begin
  DEL(p ↑ .right, p, k);
  if p ↑ .s ≠ min(p ↑ .left ↑ .s, p ↑ .right ↑ .s) + 1
    or p ↑ .h ≠ max(p ↑ .left ↑ .h, p ↑ .right ↑ .h) + 1
  then
    begin
      UPDATE(p);
      if p ↑ .h > 2 * p ↑ .s
        then RESDELRIGHT(p)
      end
    end
  end;
dispose(q)
end { delete }

```

REFERENCES

1. G. M. ADELSON-VELSKII and E. M. LANDIS, *An Algorithm for the Organization of Information*, Dokl. Akad. Nauk S.S.S.R., Vol. 146, 1962, pp. 263-266 (Russian). English translation in Soviet Math. Dokl., Vol. 3, 1962, pp. 1259-1263.
2. A. V. AHO, J. E. HOPCROFT and J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
3. R. BAYER, *Symmetric Binary B-trees Data Structure and Maintenance Algorithms*, Acta Informatica, Vol. 1, 1972, pp. 290-306.
4. N. BLUM and K. MEHLHORN, *Mittlere Anzahl von Rebalancierungoperationen in Gewichtsbalancierten Bäumen*, 4th GI Conference on Theoretical Computer Science, Aachen 1979, Lecture Notes in Computer Science, Vol. 67, pp. 67-78, Springer, Berlin, Heidelberg, New York.
5. P. L. KARLTON, S. H. FULLER, R. E. SCROGGS and E. B. KAEHLER, *Performance of Height-Balanced Trees*, Com. A.C.M. 19, Vol. 1, 1976, pp. 23-28.
6. D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968, 1973.
7. D. E. KNUTH, *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.
8. J. NIEVERGELT and E. M. Reingold, *Binary Search Trees of Bounded Balance*, S.I.A.M. J. Comput., Vol. 2, 1973, pp. 33-43.
9. H. J. OLIVIÉ, *A New Class of Balanced Search Trees: Half-Balanced Binary Search Trees*, Technical Report 80-02, IHAM, Paardenmarkt 94, B-2000 Antwerp, Belgium, 1980.
10. H. J. OLIVIÉ, *A Study of Balanced Binary Trees and Balanced One-Two Trees*, Ph. D. Thesis, Dept. of Mathematics, U.I.A., University of Antwerp, Belgium, 1980.

11. Th. OTTMANN and H.-W. SIX, *Eine Neue Klasse von Ausgeglichenen Binärbäumen*, *Angewandte Informatik*, Vol. 9, 1976, pp. 395-400.
12. E. M. REINGOLD, J. NIEVERGELT and N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N. J., 1977.
13. N. WIRTH, *Algorithm + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
14. N. ZIVIANI and F. TOMPA, *A Look at Symmetric Binary B-Trees*, Report CS-80-51, Dept. of Computer Science, University of Waterloo, Canada, 1980.