On \( k \)-Dimensional Balanced Binary Trees*

VIJAY K. VAISHNAVI

Department of Computer Information Systems, Georgia State University, Atlanta, Georgia 30303

Received May 30, 1991

An amortized analysis of the insertion and deletion algorithms of \( k \)-dimensional balanced binary trees (\( kBB \)-trees) is performed. It is shown that the total rebalancing time for a mixed sequence of \( m \) insertions and deletions in a \( kBB \)-tree of size \( n \) is \( O(k(m + n)) \). Based on \( 2BB \)-trees, a self-organizing tree, called a self-organizing balanced binary tree, is defined. It is shown that the average access time for an item stored in the tree is optimal to within a constant factor, while the worst-case access time is \( O(\log n) \). The amortized analysis of \( kBB \)-trees leads to the result that the total update time for a mixed sequence of \( m \) accesses, insertions, and (restricted) deletions in a self-organizing balanced binary tree initially storing \( n \) data items is \( O(m + n) \). © 1996 Academic Press, Inc.

1. INTRODUCTION

An ordered list of \( n \) items can be organized into a search tree [11]. A balanced binary tree [17] (same as a symmetric binary B-tree [1]) has some desirable properties and can be used as a search tree. It requires \( O(\log_2 n) \) time for access, insertion, and deletion. More significantly, it has been shown that it requires only a constant number of single rotations per insertion or deletion [17]. Also, it has been shown that a mixed sequence of insertions and deletions in an \( n \)-mode balanced binary tree takes \( O(m + n) \) total rebalancing time [9, 12, 18, 19]. The work in [24] defines and investigates multidimensional balanced binary trees, a multidimensional generalization of balanced binary trees. The current work provides amortized analysis results for this data structure and shows how it can be used to define a "nearly optimal self-organizing (search) tree" with some interesting properties.

A multidimensional balanced binary tree can, for example, be used as a "lexicographic search tree" for storing a set of \( n \) arbitrary strings of symbols such that a string of length \( k \) can be accessed, inserted, or deleted from the structure in \( O(\log_2 n + k) \) time using the same space (within a constant factor) as used by a trie to store the same set of strings (see [16] for an alternative approach). However, based on this property alone, a multidimensional balanced binary tree is similar to a number of other data structures available in the literature like lexicographic D-trees [13], lexicographic globally biased trees [3], multidimensional B-trees [6], and multidimensional AVL-trees [22]. There are, however, two properties of multidimensional balanced binary trees, described below, which makes the data structure distinct and worthy of further study.

First, the insertion and deletion algorithms for \( k \)-dimensional balanced binary trees (\( kBB \)-trees) [24] use almost the same restructuring operations as those used in the corresponding algorithms for balanced binary trees [17]. These operations are: promotion, single rotation, and double rotation for the insertion algorithm; demotion, single rotation, and modified double rotation for the deletion algorithm. Thus, the insertion and deletion algorithms for multidimensional balanced binary trees, expressed through case-analysis of the restructuring operations performed, would not be much different from the corresponding algorithms [17] for balanced binary trees. Multidimensional AVL-trees [22] have a similar property; interestingly, this data structure, like \( kBB \)-trees, follows the "neighbor-support paradigm" [10] for the multidimensional generalization of a balancing concept.

Second, a \( kBB \)-tree requires \( O(k) \) single rotations for insertion or deletion [24]. This generalizes the corresponding result for balanced binary trees [17], viz. a balanced binary tree requires \( O(1) \) single rotations per insertion or deletion. To the knowledge of the author, such a result is not available for any other known multidimensional tree structure. This result of \( kBB \)-trees is expected to increase the usefulness of the data structure. For example, this result makes it possible to make \( kBB \)-trees "partially persistent" [5, 15] in \( O(k) \) space per update (see the final section for some additional remarks). An obvious use for a partially persistent multidimensional balanced binary tree would be to organize in a space-efficient manner strings of symbols such that access can be provided in the present as well as in the past and update can be provided in the present in a time-efficient manner. Such a use of the data structure may, for...
example, be found in an application that involves finding
from a company personnel database whether a person with
a given name was employed in the company a given number
of years ago.

In this paper, we extend the work in [24] by showing that a
kBB-tree has additional interesting properties and uses.
We perform an amortized analysis of the insertion and
deletion algorithms for kBB-trees. We show that a mixed
sequence of m insertions and deletions in a kBB-tree storing
n vectors takes \( O(k(m + n)) \) total rebalancing time.
This result generalizes the corresponding result for balanced
binary trees and is, to the knowledge of the author, a new
result for multidimensional trees. The result has an obvious
implication when the data structure is used for storing a set
of strings. The other part of the work reported in this paper
analyzes, in this paper, a nearly optimal self-organizing tree,
with some interesting amortized analysis results.

Let \( T \) be binary search tree for \( n \) linearly ordered data
items, \( k_i \), \( 1 \leq i \leq n \), such that \( k_i < k_{i+1} \), \( 1 \leq i < n - 1 \). Let
the tree be searched only for the data items stored with frequency
\( w_i \), for data item \( k_i \) (also see Section 5). Let \( W = \Sigma w_i \).
It can be shown, using an entropy argument
[14], that the weighted path length of the tree is bound
from below by \( \Sigma w_i \log_2 \left( W/w_i \right) \). Thus, the best that can be
expected from such a tree is “logarithmic behavior,” i.e., the
access time for \( k_i \) is \( O(\log_2 (W/w_i)) \), in the worst case. This
ensures minimal average access time of the tree to within a
constant factor, i.e., nearly optimal behavior. The access
time for \( k_i \), as \( O(\log_2 (W/w_i)) \) has been called ideal [2].
A nearly optimal self-organizing (search) tree [2] is a search
tree that supports queries on data items in the tree whose
access probabilities are not known a priori, updating the tree
after each access in order to keep the access time ideal for
each data item.

As an important application of kBB-trees, we define and
analyze, in this paper, a nearly optimal self-organizing
tree, self-organizing balanced binary tree, based on 2BB-trees. In
order to show that the access time of each data item stored in
a self-organizing balanced binary tree is ideal, we derive a
new tight upper bound for the “rank” (and the height) of
a 2BB-tree. The amortized analysis of kBB-trees leads to the
result that the total rebalancing time for a mixed sequence
of \( m \) accesses, insertions, and (restricted) deletions in a self-
organizing balanced binary tree that initially stores \( n \) data
items, is \( O(m + n) \). This seems to be a new result for a nearly-optimal self-organizing search tree.

The rest of this paper is organized as follows. In Section 2,
we review the definition of kBB-trees and its update
algorithms [24]. In Section 3, we perform an amortized
analysis of the insertion and deletion algorithms for kBB-
trees. In Section 4, we define and analyze a self-organizing
balanced binary tree. In the final section, we discuss various
issues, draw conclusions, and review the results of this paper
in the light of the existing literature.

2. k-DIMENSIONAL BALANCED BINARY TREES

A k-dimensional binary tree, \( k \) a positive integer, is either
empty (represented by a k-dimensional external node) or is
a k-dimensional internal node with three subtrees (the left,
the middle, and the right subtree); the left and right subtrees
are k-dimensional binary trees and the middle subtree is a
nonempty \((k - 1)\)-dimensional binary tree unless \( k = 1 \).

The size of a k-dimensional binary tree \( T \) is the number of
one-dimensional internal nodes in \( T \).

Let \( P \) be the middle child of its parent, \( N \), in a k-dimen-
sional binary tree. \( Q \) is a left (resp., right) sibling of \( P \) if \( Q \)
is a left (resp., right) child of \( N \). \( Q \) is a left (resp., right) neighbor of \( P \) if \( Q \) is either a left (resp., right) sibling of \( P \)
or a right (resp., left) descendant \( 2 \) of the left (resp., right)
sibling of \( P \). If \( Q \) is a left (resp., right) neighbor of \( P \) then \( P \)
is a right (resp., left) neighbor of \( Q \).

A k-dimensional balanced binary tree (kBB-tree) [24],
\( k \) a positive integer, is a k-dimensional binary tree each of
whose nodes has an integer rank\(^2\) such that the ranks have
the following properties:

(a) Rank-value property. The rank of each external
node is zero. The rank of each internal node is greater than
zero.

(b) Rank-propagation property. The rank of a node is
less than or equal to that of its parent. The rank of the
middle child of a node is less than that of its parent. The
rank of a node \( N \) is equal to the rank of its child, \( M \), or one
plus the rank of \( M \), where \( M \) has the maximum rank among
the children of \( N \).

(c) Rank-gap\(^4\) property. If there is a rank-gap between
\( P \) and its left or right child \( Q \) then for at least one of the
neighbors, \( N \), of \( Q \), rank(\( N \)) \( \geq \) rank(\( P \)) \(- 1 \). If \( N \) is a sibling
of \( Q \) then the rank-gap is said to be supported directly by \( N \);
otherwise, the rank-gap is said to be supported indirectly
by \( N \).

(d) Rank-support property. If \( Q \) is the middle child
of its parent \( P \) and \( R \) is a sibling of \( Q \) such that
rank(\( Q \)) \(<\) rank(\( P \)) \(- 1 \) and rank(\( R \)) \( = \) rank(\( P \)), then for at
least one of the neighbors, \( N \), of \( Q \), rank(\( N \)) \( = \) rank(\( P \)) \(- 1 \).

(e) Rank-balance property. If a node \( P \) has a grand-
parent \( R \) then rank(\( P \)) \( < \) rank(\( R \)).

In a 1BB-tree, the middle subtree of each internal node is
empty and, hence, its rank is zero. Thus, in such a tree, no

---

1. A zero-dimensional binary tree is always empty.
2. \( R \) is a left (resp., right) descendant of \( S \) if either \( R \) is the left (resp., right)
child of \( S \) or \( R \) is the left (resp., right) child of a left (resp., right) descendant
of \( S \).
3. The rank of a node \( N \) is denoted by rank(\( N \)). The rank of a subtree is the
rank of its root.
4. There is a rank-gap between \( P \) and its left or right child \( Q \) if rank(\( P \)) \( > \)
rank(\( Q \)) \(+ 1 \). The width of the rank-gap is equal to rank(\( P \)) \(- \) rank(\( Q \)) \(- 1 \).
rank-gap can be supported. This leads to the following observation:

**Proposition 2.1.** A $1BB$-tree is the same as a balanced binary tree [17].

If a $k$-dimensional binary tree is a $kBB$-tree, except that it does not satisfy one or more properties of its ranks, then it is said to have the corresponding structure violation(s) and is obviously not a valid $kBB$-tree. Figure 2.1 illustrates rank-propagation violation, rank-gap violation, rank-support violation, and rank-balance violation at a node. Symmetric cases are omitted throughout the paper.

**Example 2.1.** The trees shown in Fig. 2.2 is a $3BB$-tree whose rank is 5 and whose size is 18. The vectors stored in the tree are: \{1, a, 1\}, \{1, b, 1\}, \{1, c, 2\}, \{1, d, 1\}, \{1, d, 2\}, \{1, d, 3\}, \{1, d, 4\}, \{1, d, 5\}, \{1, d, 6\}, \{1, d, 7\}, \{1, d, 8\}, \{1, d, 9\}, \{1, d, 10\}, \{2, a, 1\}, \{3, b, 2\}, \{4, a, 2\}, \{5, a, 1\}, and \{6, b, 2\}. The ranks of the nodes are shown beside the nodes. In the tree all empty subtrees (represented by external nodes) are omitted. The edge from a node to its middle child indicating a dimensional jump is shown as dotted. Observe that the rank gap between $A$ and its left child (an external node which is not shown) is supported by the two-dimensional node $B$. Again, observe that the rank of the middle child of $C$ is less than $3(\text{rank}(C) - 1)$ but the rank of $D$ is 3, thus maintaining the rank-support property.

The following observations help in an intuitive understanding of the definition of $kBB$-trees and its update

![FIG. 2.1. Illustration of different types of structure violations at $A$. The rank of a node is shown beside the node. A "*" beside a node indicates that there is a structure violation at the node. Ranks of certain subtrees are also shown: (a) Rank-propagation violation; (b) Rank-gap violation; (c) Rank-support violation; (d) Rank-balance violation.](image-url)
rebalancing algorithms. Balanced binary trees [17] (same as symmetric binary B-trees [1]) are binarized versions of 2–3–4–trees [1]. Similarly, kBB-trees are binarized versions of “k-dimensional 2–3–4–trees” which can be defined similarly as kBB-trees with the concept of height replacing that of the rank of a node. In such a definition, only properties corresponding to properties (c) and (d) are significant; the other properties follow from the concept of height of a node and that of a tree. Figure 2.3 shows the correspondence between nodes in a k-dimensional 2–3–4–tree and corresponding structures in its binarized version, a kBB-tree. This correspondence is exactly similar to that between a node of a 2–3–4–tree and the corresponding structure in its binarized version, a balanced binary tree.

Figure 2.4 shows the three-dimensional 2–3–4–tree such that Fig. 2.2 is its binarized version. With this correspondence between kBB-trees and k-dimensional 2–3–4–trees, one can better understand the insertion and deletion rebalancing in kBB-trees by relating them to that of k-dimensional 2–3–4–trees; as may be expected, the rebalancing operations used in the latter are similar to the rebalancing operations used in 2–3–4–trees, viz. split, merge, and shift.

3. AMORTIZED UPDATE TIME

The insertion and deletion algorithms [24] for kBB-trees are summarized in Fig. 3.1 through 3.5 and Fig. 3.6 through 3.10, respectively. Tables I and II summarize the key correctness arguments for the algorithms; Table I uses Lemma 3.1 in arguing the correctness of Case 1d. The following two results are useful in the amortized analysis of the insertion and deletion algorithms.

**Lemma 3.1.** In the insertion algorithm, if the rebalancing rule for Case 1d (Fig. 3.2(d)) is applicable then the previous

![Diagram](image)

**FIG. 2.2.** A 3BB-tree.

**FIG. 2.3.** Correspondence between k-dimensional 2–3–4–trees and kBB-trees.

**FIG. 2.4.** A three-dimensional 2–3–4–tree.

**FIG. 3.1.** Insertion causing the replacement of an external node A by a chain of i, 1 ≤ i ≤ k, internal nodes at A. The symbol “+” in parentheses beside i indicates that the rank of A has increased.

![Diagram](image)
FIG. 3.2. Case 1. Structure violation only at the parent, B, of the current node. (a) Case 1a. Rank-support violation at B; rank of the left/ right child of B is \( r \) or less; single rotation at B; terminating case. (b) Case 1b. Rank-support violation at B; both (left/right) children of B have rank \( r \); promotion (of rank) at B. (c) Case 1c. Rank-propagation violation; current node middle child of B; promotion at B. (d) Case 1d. Rank-propagation violation; current node left-right child of B; single rotation at B.

step in the execution of the algorithm must have been prompted by an increase in the rank of the middle child of A, current node.

Proof. Consider the subtree being restructured in this case (see Fig. 3.2(d)). The previous step cannot have been the insertion of the given vector (see Fig. 3.1.) because in such a case B must be at least \( r \)-dimensional and hence its rank must be \( r \) or more. Thus the previous step must have been a rebalancing step (application of a rebalancing rule) causing an increase by 1 in the rank of the rebalanced subtree. We shall now show that the previous rebalancing step must have been prompted by an increase in the rank of the middle child of A, the current node. Consider the subtree just before the performance of the previous step. At this point of rebalancing, let the left child of B be C with the rank of \( r - 1 \). (Note that if the previous step involved single or double rotation then C must be different from A.) In order to prove the result, let us assume to the contrary. Suppose the previous step was prompted by an increase in the rank of the left or right child of C, say D. Then the rank of D must have been either \( r - 1 \) or \( r \) to warrant any processing. In the former case, there must have been at least a rank-balance violation at B and a rebalancing rule for Case 2 or Case 3 must have been applied, resulting in either the termination of the algorithm or an increase in the rank of B to \( r \)—a contradiction. In the latter case, the rank of D must have increased to \( r \) from \( r - 1 \) because of a rebalancing step; the increase in the rank of D to \( r \) cannot be because of the insertion of the given vector (see Fig. 3.1.). In such a case, there must have been a rank-balance violation at B in the original tree (before the initiation of the insertion algorithm)—a contradiction. Thus the previous step must have been prompted by an increase in the rank of the middle child of C. Hence the result.

Lemma 3.2. In the deletion algorithm (see Fig. 3.6 through 3.10),

(a) if Case 1 (Fig. 3.7) is applicable because of a decrease in the rank of the left or right child of a node then the restructuring subtree is free from any structure violation;

(b) if Case 1 is applicable because of a decrease in the rank of the middle child in a node then the potential structure violation, if any, created within the restructured subtree is corrected in the next step (without introducing any new structure violation within the subtree);

(c) the potential rank-gap violation, if any, created in the restructured subtree in Case 2 gets corrected in the next two steps.
TABLE I

Insertion Algorithm and Key Correctness Arguments

<table>
<thead>
<tr>
<th>Case</th>
<th>Key correctness arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Structure violation only at the parent, $B$, of the current node $a$ (Fig. 3.2).</td>
<td>Subtree being restructured. No neighbor of the root of Subtree 4 with rank $r = 1$ to guarantee the rank-support property. Rank gap of at least one of the subtrees, 1 and 2, must be $r = 1$ (invariant property). Rank of Subtree 2 must be $r = 1$ to support the rank gap between $A$ and the root of Subtree 3. Restructured subtree. Rank gap between $A$ and $B$, if any, is directly supported.</td>
</tr>
<tr>
<td>1a (Fig. 3.2a). Rank-support violation at $B$; rank of the left/right child of $B$ is $r = 2$ or less; single rotation at $B$, terminating case.</td>
<td>Restructured subtree. If $B$ is the left or right child of its parent, then there cannot be a new structure violation at its parent; a rank-propagation violation would mean that there was a rank-balance violation at the grandparent of the current node in the subtree being restructured, which violates the premise of Case 1. On the other hand, if $B$ is the middle child of its parent then the only possible structure violation at its parent, is a rank-propagation violation when the rank of $P$ is $r + 1$. Thus, there is at most one type of structure violation at the parent of $B$—maintaining the invariant property.</td>
</tr>
<tr>
<td>1b (Fig. 3.2b). Rank-support violation at $B$; both (left/right) children of $B$ have rank $r$; promotion (of rank) at $B$.</td>
<td>Restructured subtree. If $B$ is the left or right child of its parent then there can be a rank-propagation violation only or rank-support violation only (but not both) at its parent. There is a rank-propagation violation at $P$, the parent of $B$, if the rank of $P$ is $r$; a rank-support violation too at $P$ would mean the existence of this structure violation at $P$ even before insertion and the initiation of restructuring. There can only be a rank-propagation violation at $P$ if $B$ is its middle child. Thus, the invariant property is maintained.</td>
</tr>
<tr>
<td>1c (Fig. 3.2c). Rank-propagation violation at $B$; current node $a$ middle child of $B$; promotion at $B$.</td>
<td>Restructured subtree. In view of Lemma 3.1, the rank of Subtree 1 as well as that of Subtree 3 must be $r = 2$ or less because otherwise there is a rank-balance violation in the original tree (before insertion and the initiation of restructuring). If the rank of Subtree 5 is $r = 1$ then the rank of at least one of the subtrees, 3, 4, and 5, and the left subtree of the root of Subtree 5, must be $r = 2$ because otherwise there is a rank-support violation at $B$ in the original tree (see Fig. 3.3a). Again, for the same reasons as above, if the rank of Subtree 5 is $r = 2$ or less then the rank of at least one of the subtrees, 3 and 4, and Subtree 5 must be $r = 2$ (see Fig. 3.3b).</td>
</tr>
<tr>
<td>1d (Fig. 3.2d). Rank-propagation violation at $B$; current node $a$ left/right child of $B$; single rotation at $B$.</td>
<td>Restructured subtree. Any rank-gap at $A$ is directly supported.</td>
</tr>
<tr>
<td>2. Rank-support violation at the parent of the current node $a$ as well as structure violation(s) at the grandparent of the current node $a$ (Fig. 3.4).</td>
<td>Subtree being restructured. There is a rank-support violation at $B$, the parent of the current node,$a$ and a rank-balance violation at $C$, the grandparent of the current node.$a$ This case is treated exactly the same way as Case 1a. Restructured subtree. There cannot be a rank-balance violation at $C$ because the rank of each child of $A$ is $r = 1$ or less.</td>
</tr>
<tr>
<td>2a (Figs. 3.4a and 3.4b). Only a rank-balance violation at the grandparent; single rotation at $B$, terminating case.</td>
<td>Subtree being restructured. The rank of Subtree 4 must be $r = 1$ in view of the invariant property and the fact that the ranks of subtrees, 3 and 5, are $r = 2$ or less to create rank support violations at $B$ and $C$. In Fig. 3.4d, the rank of Subtree 8 must be $r = 1$ in order to support the rank gap between $D$ and the root of Subtree 7. Restructured subtree. Any rank-gap at $A$ is directly supported.</td>
</tr>
<tr>
<td>2b (Figs. 3.4c and 3.4d). A rank-balance as well as a rank-support violation at the grandparent; either a double rotation or a double rotation followed by a single rotation at $C$; terminating case.</td>
<td>Subtree being restructured. There is a rank-support violation at $B$, the parent of the current node, and a rank-balance violation at $C$, the grandparent of the current node. This case is treated exactly the same way as Case 1a. Restructured subtree. There cannot be a rank-balance violation at $C$ because the rank of each child of $A$ is $r = 1$ or less.</td>
</tr>
<tr>
<td>3. No structure violation at the parent of the current node $a$ but a rank-balance violation and possibly a rank-support violation at its grandparent, $C$ (Fig. 3.5).</td>
<td>Subtree being restructured. There is a rank-support violation at $C$ and possibly (in Fig. 3.5b) a rank-support violation at $C$.</td>
</tr>
<tr>
<td>3a (Figs. 3.5a and 3.5b). Both (left/right) children of $C$ have rank $r$; promotion at $C$.</td>
<td>Fig. 3.5c—Subtree being restructured. The rank of Subtree 5 must be $r = 1$ or less and the rank of one of the subtrees, 5, 6, and 7, must be $r = 1$ because otherwise there must have been a structure violation in the original subtree. The rank of one of the subtrees, 3, 4, and 5, must be $r = 1$ in view of the premise of Case 3.</td>
</tr>
<tr>
<td>3b (Figs. 3.5c and 3.5d). Rank of a left/right child of $C$ is $r = 1$ or less; single rotation or double rotation at $C$, terminating case.</td>
<td></td>
</tr>
</tbody>
</table>
from the other children of B at A. Children of B are ranked at A (Fig. 3.7c). Ranks of both left and right children of B are \( r \pm 1 \) in view of the premise of the case. If there is a rank-support violation at C then the rank of Subtree 4 must be \( r - 1 \) in order to support the rank-gap between A and the root of Subtree 5. Thus, in such a case, there cannot be a rank-support violation at A in the restructured subtree, and any rank gap at A is directly supported. If, on the other hand, there is not a rank-support violation at C, then the rank of at least one of the subtrees, 5, 6, and 7, must be \( r - 1 \). Thus, in the restructured subtree, the rank of C is \( r \) and their is no rank-support violation at A because the rank of at least one of the subtrees, 3, 4, and 5, is \( r - 1 \) in view of the invariant property.²

² Current node. That node on the restructuring path whose rank has increased, causing structure violation(s) at its parent and/or grandparent; the subtree rooted at the current node does not have any structure violation.

² Invariant property. If the rank of a node (or the one replacing it), \( P \), on the restructuring path increases during the execution of the insertion algorithm \( P \) becoming the new current node), then (a) the rank of \( P \) is one plus the maximum of the ranks of the children of \( P \) and (b) there is at most one type of restructuring violation at the parent of \( P \).

### Table I—Continued

<table>
<thead>
<tr>
<th>Case</th>
<th>Key correctness arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 3.5d—Subtree being restructured. The rank of Subtree 1 must be ( r - 1 ) or less because otherwise there is a structure violation in the original tree. The rank of at least one of the subtrees, 1, 2, and 3 must be ( r - 1 ) in view of the premise of the case. If there is a rank-support violation at ( C ) then the rank of Subtree 4 must be ( r - 1 ) in order to support the rank-gap between ( A ) and the root of Subtree 5. Thus, in such a case, there cannot be a rank-support violation at ( A ), in the restructured subtree, and any rank gap at ( A ) is directly supported. If, on the other hand, there is not a rank-support violation at ( C ), then the rank of at least one of the subtrees, 5, 6, and 7, must be ( r - 1 ). Thus, in the restructured subtree, the rank of ( C ) is ( r ) and their is no rank-support violation at ( A ) because the rank of at least one of the subtrees, 3, 4, and 5, is ( r - 1 ) in view of the invariant property.²</td>
<td></td>
</tr>
</tbody>
</table>

### Table II

Deletion Algorithm and Key Correctness Arguments

<table>
<thead>
<tr>
<th>Case</th>
<th>Key arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A potential rank-gap violation at ( A ) (the possible indirect support is not checked); there may also be a rank-support violation at the parent of ( A ); rank of ( B ), a left/right child of ( A ), is ( r - 1 ) (Fig. 3.7).</td>
<td></td>
</tr>
<tr>
<td>1a (Fig. 3.7a). Ranks of all children of ( B ) are ( r - 2 ) or less; demotion (or rank) at ( A ); a rank-support violation may result at ( A ) which is treated according to Case 4b.</td>
<td></td>
</tr>
<tr>
<td>1b (Fig. 3.7b). Left/right child of ( B ) farthest from ( D ) has rank of ( r - 1 ) and the ranks of the other children of ( B ) are ( r - 2 ) or less; single rotation at ( A ); if, in the restructured subtree, ( s ) is ( r - 2 ) or less then a potential rank-gap violation at ( B ) as well as (the still existing) rank-support violation at the parent of ( B ), if any, are treated according to Case 1a.</td>
<td></td>
</tr>
<tr>
<td>1c (Fig. 3.7c). Ranks of both left/right children of ( B ) are ( 1 ) each; single rotation at ( A ); a rank-support violation may result at ( A ) which is treated according to Case 4b.</td>
<td></td>
</tr>
<tr>
<td>The demotion in the rank of ( A ) corrects the rank-support violation at the parent of ( A ), if any. If the rank of ( D ) is ( r - 2 ) or the rank of Subtree 1 is ( r - 2 ) then the potential rank-gap violation at ( A ) is corrected and no other structure violation is introduced in the resulting subtree. If the rank of the left neighbor of ( D ) has decreased to ( r - 2 ) then in the resulting subtree, the rank-gap at ( A ), if any, is supported. There may, however, be a rank-support violation at ( A ) in the resulting subtree if the rank of ( D ) is ( r - 3 ) or less and the ranks of subtrees 1 and 2 are ( r - 3 ) or less. In such a case, the rank of Subtree 3 must be ( r - 2 ) in order to support the rank-gap between ( B ) and the root of Subtree 2 in the subtree being restructured; the rebalancing rule for Case 4b is then clearly applicable.</td>
<td></td>
</tr>
<tr>
<td>If the rank of ( D ), that of Subtree 1, or that of Subtree 2 is ( r - 2 ), then the resulting subtree is free from any structure violation and there cannot be a rank-support violation at the parent of ( B ). Observe that if only the rank of Subtree 2 is ( r - 2 ) while the ranks of ( D ) and Subtree 1 are ( r - 3 ) or less, then there may still be a rank-gap at ( A ) in the restructured subtree; this rank-gap is, however, supported since the rank of the left neighbor of ( D ) must be ( r - 2 ). If the ranks of ( D ), Subtree 1, and Subtree 2 are ( r - 3 ) or less, then the rank-support violation, if any, at the parent of ( B ) still exists and there is also a potential rank-gap violation at ( B ) in the restructured subtree. In the latter case, observe the following: The rank of each of the children of the root of Subtree 4 must be ( r - 2 ) or less because otherwise there is a rank-balance violation at ( B ) in the subtree being restructured, which is not possible. Thus, the rebalancing rule for Case 1a must be applied next. After restructuring, there cannot be a rank-support violation at the parent of ( B ). If the rank of ( D ) or that of Subtree 1 is ( r - 2 ) then the resulting subtree is free from any structure violation. If, on the other hand, the rank-support violation at ( A ) was because of a decrease in the rank of the left neighbor of ( D ) to ( r - 2 ) then there is no rank-gap violation in the resulting subtree; there may, however, be a rank-support violation at ( A ) for which the rebalancing rule of Case 4b is applicable.</td>
<td></td>
</tr>
</tbody>
</table>
A potential rank-gap violation only or a rank-support violation at \( A \) can get created at the parent of \( A \) only if the ranks of \( A \) and \( B \) are \( r - 2 \) or less. In the subtree being restructured, the rank of Subtree 4 must be \( r - 1 \) in order to support the rank-gap at \( B \) and the root of Subtree 3. Again, the rank of Subtree 4 must be \( r - 1 \) or less because otherwise there would be a rank-balance violation at \( A \). In the restructured subtree, rank-gap(s) at \( B \) if any, are directly supported and there is clearly no structure violation at \( A \).

**Notes.** The restructuring is carried out because the rank of a node has decreased by 1 except at the very start when its rank may have decreased from \( i \) to 0 (Fig. 3.6a), in which case the rank of its sibling (middle child) cannot be less than \( i - 1 \). If this node, \( P \), is the left/right child of its parent then an action is taken at the parent or the grandparent of \( P \) according to the rebalancing rules in Table II. If, on the other hand, \( P \) is the middle child of its parent then a sequence of actions is initiated at one or both of its neighbors, say \( R \) (if there is a rank-gap at \( R \) which loses its support), before an action is considered at the parent of \( P \).

**Proof.** Figure 3.11 shows the cases in which potential structure violations may be created in the respective restructured subtrees and how they get corrected.

(a) Observe that if the rank of \( D \) in Fig. 3.7 is \( r - 2 \), then the resulting subtree in each case (Case 1a, Case 1b, Case 1c, Case 1d) is free from any structure violation. Hence the result.

(b) See Fig. 3.11. For Cases 1a and 1c, a structure violation introduced within the restructured subtree is treated according to Case 4b. Case 4b does not introduce any new structure violation within the subtree. For Cases 1b and 1d, on the other hand, a structure violation introduced within the subtree is treated according to Case 1a. Therefore, what we need to show is that if the application of the rebalancing rule for Case 1b or Case 1d results in a potential rank-gap violation (which is treated according to Case 1a), then the application of the rebalancing rule for Case 1a does not introduce a structure violation within the restructured subtree. Consider Case 1b (see Fig. 3.7(b)). The restructured subtree (after the application of single rotation) has a potential rank-gap violation at \( B \) only if the ranks of Subtree 2 as well as those of Subtree 1 and \( D \) are \( r - 3 \) or

---

**TABLE II—Continued**

<table>
<thead>
<tr>
<th>Case</th>
<th>Key arguments</th>
</tr>
</thead>
</table>
| 1d (Fig. 3.7d). Rank of left/right child of \( B \) nearest to \( D \) is \( r - 1 \) while the ranks of the other children of \( B \) are \( r - 2 \) or less; double rotation at \( A \); if \( s \) is \( r - 2 \) or less, then the resulting potential rank-gap violation at \( C \) as well as the (still remaining) rank-support violation, if any, at the parent of \( C \) is treated according to Case 1a.

2. A potential rank-gap violation only or a rank-support violation at \( A \); rank of \( B \), a left/right child of \( A \), is \( r \); single rotation at \( A \) (Fig. 3.8). (Note that in the subtree being restructured either the rank of \( D \) or that a neighbor of \( D \) has decreased to \( r - 2 \)).

3. Rank-propagation violation at \( A \); the rank of Subtree 1 or that of Subtree 2 has decreased to \( r - 2 \); demotion of \( A \) (Fig. 3.9).

4. Only a rank-support violation at \( A \) (Fig. 3.10). (No potential rank-gap violation.)

4a (Fig. 3.10a). Ranks of both left/right children of \( A \) are \( r \); modified double rotation (composed of two single rotations) at \( A \).

4b (Fig. 3.10b). Rank of a left/right child of \( A \) is \( r - 2 \) or less; single rotation at \( A \).

The rank-support violation at the parent of \( A \), if any, gets corrected and the restructured subtree is free from any structure violation.

There cannot be a rank-support violation at the parent of \( A \) in the subtree being restructured. A rank-support violation can get created at the parent of \( A \) only if the rank of the parent is \( r \) and either the rank of \( D \) decreases to \( r - 2 \) or the rank of the middle child of the parent decreases to \( r - 2 \). This is not, however, possible because it would imply the existence of a rank-balance violation in the original tree (before the deletion). The ranks of each of the subtrees, 2, 3, and 4, must be \( r - 1 \) or less and the rank of at least one of them must be \( r - 1 \); otherwise, there is a rank-propagation violation in the original tree. If the rank of Subtree 2 is \( r - 2 \) or less, then the rank of Subtree 3 must be \( r - 1 \) to support a rank-gap violation at \( B \) in the subtree being restructured. Thus, any rank-gap at \( B \) in the restructured subtree is directly supported and the restructured subtree is free from any structure violation.

In the subtree being restructured, the ranks of each of the subtrees, 2 and 6, must be \( r - 1 \) because otherwise there would be a rank-gap violation at \( B \) or \( C \).

In the subtree being restructured, the rank of Subtree 4 must be \( r - 1 \) in order to support the rank-gap between \( B \) and the root of Subtree 3. Again, the rank of Subtree 5 must be \( r - 1 \) or less because otherwise there would be a rank-balance violation at \( A \). In the restructured subtree, rank-gap(s) at \( B \) if any, are directly supported and there is clearly no structure violation at \( A \).
FIG. 3.4. Case 2. Rank-support violation at the parent of the current node as well as structure violation(s) at the grandparent of the current node. (a) (b) Case 2a. Only a rank-balance violation at the grandparent; single rotation at $B$; terminating case. (c) (d) Case 2b. A rank-balance as well as a rank-support violation at the grandparent; either double rotation or double rotation followed by single rotation at $C$; terminating case.

Amortized Analysis

We use the banker’s view of amortization [4, 9, 19] for analyzing the amortized rebalancing time for insertions and deletions in a $k$BB-tree. As an accounting device, we assume that in a $k$BB-tree, nonnegative credits are contained in certain internal nodes and in each edge between an internal node and its middle child. The credit invariant property is a generalization of the one used for the amortized analysis of balanced binary trees [18, 19], also known as “red-black trees” [8]. The credit invariant property is illustrated in Fig. 3.12, using the notion of colored nodes. An internal node of rank $r$ is black if the rank of its parent is $r + 1$ or
FIG. 3.5. Case 3. No structure violation at the parent (of the current node) but a rank-balance violation and possibly a rank-support violation at the grandparent, C. (a)–(b) Case 3a. Both (left/right) children of C have rank \( r \); promotion at C; (c)–(d) Case 3b. Rank of left/right child of C is \( r-1 \) or less; single rotation or double rotation at C; terminating case.

more; the root is black as a special case. If a node does not satisfy the above property then it is red. (This coloring notation is again a generalization of the one for balanced binary trees \([8, 19]\); we, however, limit its use to the amortization analysis.) In a \( k \)-BB-tree, each black node contains 0, 2, or 8 credits. A red node does not contain any credits. The credit invariant property (see Fig. 3.12) is as follows. A black node contains either 2 or 8 credits, if it has no red children or exactly two (left and right) red children, respectively. In all other cases, it contains zero credits. Every edge connecting an internal node \( P \) and its middle child \( Q \) (no matter what the colors of the two nodes are) contains credits equal to three times the “rank-gap” between the former and the latter (\( \text{rank}(P) - \text{rank}(Q) - 1 \)) or 9, whichever is smaller, but at least zero.

Amortized Rebalancing Time for Insertions

Let the rebalancing operations involved in Cases 3a, 1b, 1c, and 1d of the insertion algorithm (see Fig. 3.2 through 3.5) be called type-a, type-b, type-c, and type-d operations, respectively. Let the rebalancing operation for any other case be called a type-e operation.

An insertion rebalancing sequence is the sequence of rebalancing operations (of various types) involved in an insertion into a \( k \)-BB-tree. An \( x \) in such a sequence denotes the application of a type-\( x \) (rebalancing) operation.
Let an *a-string* denote a string composed of one or more *a*'s. Let an *a-substring*(*s*) denote an *a-string* that is a substring of a string *s* such that it is not immediately preceded or followed by an *a*.

**Lemma 3.3.** An insertion rebalancing sequence, *S*, is composed of *O*(k) occurrences each of *a*-substring(*s*), *b*, *c*, and *d*, and one occurrence of *e*; an *e* can only be the last symbol in *s*.

**Proof.** A type-e operation terminates the insertion algorithm. Thus a type-e operation can only be applied once as the last operation in *S*. A type-c operation is applied when the rank of the middle child of a node increases by 1 and there are only *k* − 1 such nodes in the restructuring path. A type-d operation must be immediately preceded by a type-c operation, in view of Lemma 3.1. A type-b operation cannot immediately follow a type-a operation or another type-b operation (see Figs. 3.2(b), 3.5(a), and 3.5(b)) because a rank-support violation cannot occur immediately after either of these operations. Thus the occurrence of *b*'s in *s* is limited to a maximum of *k* − 1. The number of *a*-substring(*s*)'s is *O*(k) in view of the limit on the number of occurrences of *b*, *c*, *d*, and *e*, and the definition of an *a*-substring(*s*). The result follows.

**Lemma 3.4.** If a type-a operation *x* is followed immediately by another type-a operation, in an insertion rebalancing sequence, then the execution of *x* releases at least three credits from the tree (without violating the credit invariant property).

**Proof.** Consider a type-a operation (see Fig. 3.13) *x* in an insertion rebalancing sequence, as constrained above. In the subtree being restructured, *C* cannot be red because otherwise there is a rank-balance violation in the original tree (even before an insertion into the tree). After the promotion of the rank of *C*, *B*, and *D* are clearly black; *C* must be red in order for its rank-increase to cause a type-a operation. The promotion operation may cause an increase in the credits stored in the edge between *C* and its middle child by at most 3. Also, since the right child of *B* cannot be red, *B* must store zero credits after the promotion. *D* must similarly store two credits. Thus, there is a decrease of at least 8 − 2 − 3 = 3 credits stored in the tree and the credit invariant property continues to be maintained. Hence the result.

**Theorem 3.5.** The rebalancing time for *m* insertions in a *k*BB-tree of size *n* is *O*(k(*m* + *n*)).

**Proof.** The initial tree with size *n* will have at most *kn* nodes and at most (*k* − 1)*n* nodes with middle children. Thus, in order for the initial tree to satisfy the credit invariant property, we must add *O*(kn) credits to the tree resulting in the *O*(kn) term in the bound. At the time of each insertion, *O*(k) credits will need to be added to the tree; all the insertions accounting for *O*(kn) credits. Observe that all but the last type-a operation in a group of consecutive type-a operations in an insertion rebalancing sequence is paid for through the release of credits in view of Lemma 3.4 and the fact that each rebalancing operation costs one credit. The result follows now in view of Lemma 3.3 and the fact that the execution of a rebalancing operation other than a type-a operation or a last *a*-operation in a sequence of *a*-operations can at most necessitate the addition of a constant number of credits to the tree (so as to maintain the credit invariant property).

**Corollary 3.5.1.** The amortized rebalancing time per update for *m* insertions into a *k*BB-tree of size *n* is *O*(k) if *m* = *Ω*(n).

**Corollary 3.5.2.** The amortized rebalancing time for insertions into an initially empty *k*BB-tree is *O*(k).

**Amortized Rebalancing Time for Deletions**

Let the rebalancing operation (demotion) involved in Case 1a of the deletion algorithm (see Fig. 3.7(a)) be called type-α operation if the operation is performed at a node because of a decrease in the rank of its left or right child (*D* in Fig. 3.7(a)); otherwise let the operation be called a type-α2 operation. Let the rebalancing operation involved in Case 3 (see Fig. 3.9) be called a type-β operation if it is performed at a node because of a decrease in the rank of
FIG. 3.7. Case 1. A potential rank-gap violation at $A$. There may also be a rank-support violation at the parent of $A$. Rank of $B$, a left-right child of $A$, is $r \leq 1$. (Either the rank of $D$ has decreased by 1 to $r - 2$ or the rank of a neighbor of $D$ has decreased to or is $r - 2$.)

(a) Case 1a. Ranks of all children of $B$ are $r \leq 2$ or less; Demotion (of rank) at $A$. A rank-support violation may result at $A$ which is treated according to Case 4b.

(b) Case 1b. Left/right child of $B$ farthest from $D$ has rank of $r \leq 1$ and the ranks of the other children of $B$ are $r \leq 2$ or less; single rotation at $A$. If, in the restructured subtree, $s \leq r - 2$, then a potential rank-gap violation at $B$, as well as (the still existing) rank-support violation at the parent of $B$, if any, are treated according to Case 1a.

(c) Case 1c. Ranks of both left-right children of $B$ are $r \leq 1$; single rotation at $A$. A rank-support violation may result at $A$ which is treated according to Case 4b. (d) Case 1d. Rank of left-right child of $B$ nearest to $D$ is $r \leq 1$ while the ranks of the other children of $B$ are $r \leq 2$ or less; double rotation at $A$. If $s \leq r - 2$, then the resulting potential rank-gap violation at $C$ as well the (still remaining) rank-support violation, if any, at the parent of $C$ is treated according to Case 1a.

its left child only or right child only; otherwise, let the operation be called a type-$b_2$ operation. Let the rebalancing operation involved in any other case of the deletion algorithm be called a type-$c$ operation.

A deletion rebalancing sequence is the sequence of rebalancing operations involved in a deletion in a $k$-dimensional balanced binary tree. An $x$ in such a sequence denotes the application of a type-$x$ operation.

Let an $(a_1, b_1)$-string denote a string composed of zero or more $a_1$’s and $b_1$’s, but at least one $a_1$ or $b_1$. Let an $(a_1, b_1)$-substring(s) denote an $(a_1, b_1)$-string that is a substring of $s$ such that it is not immediately preceded or followed by an $a_1$ or $b_1$.

Lemma 3.6. A deletion rebalancing sequence, $s$, is composed of $O(k)$ occurrences each of $(a_1, b_1)$-substring(s), $a_2, b_2$, and $c$.

Proof. Figure 3.14 illustrates the pattern in which rebalancing operations are applied because of a deletion in a $k$-BB-tree starting from the actual deletion of a $k$-dimensional vector (see table II for details of the deletion algorithm). Because of the deletion, the rank of the lowest
A (possibly empty) sequence of type-a\(^1\), and application of a type-a\(^2\) or type-b\(^2\) operation at the very rank of \(K\) nodes, are initiated starting at a node, \(Q\). This pattern continues until either the algorithm terminates or there is a decrease (by 1) in the rank of the highest node, \(P\), on Path A (\(P\) being the middle child of a node). The latter event leads to the application of rebalancing operations at path B nodes only or at path C nodes only, the application of such operations at Path B nodes followed by the application at path C nodes, or the application of no such rebalancing operations; followed by considering the application of a rebalancing operation (type-a\(_2\), type-b\(_2\), or type-c\(_2\)) at \(D\) (the middle child of a node). The rebalancing operations at, say Path B nodes, are initiated starting at a node, \(Q\), because a rank-gap at \(Q\) has lost its only (indirect) support with the decrease in the rank of \(P\). The rebalancing operations for Path B (or Path C) nodes thus start with a type-a\(_1\) operation followed possibly by a type-c operation or a type-c operation followed by at most two additional operations (see Fig. 3.7, Fig. 3.11, and Lemma 3.2). This is followed by a (possibly empty) sequence of type-a\(_1\), and/or type-b\(_1\) operations possibly ending in a type-c operation. If a rebalancing operation at \(D\) (which can only be a type-a\(_2\), type-b\(_2\), or type-c\(_2\) operation) results in a decrease in the rank of \(D\) then the same pattern of actions, as described above, repeats.

Observe that during rebalancing, the middle child of a node is encountered at most \(k-1\) times. The result follows now in view of the above discussion.

FIG. 3.8. Case 2. A potential rank-gap violation only or a potential rank-gap violation as well as a rank support violation at \(A\). Rank of \(R\), a left/right child of \(A\), is \(r\); single rotation at \(A\). If the rank of Subtree 2 is \(r-1\) then the potential rank-gap violation at \(A\) is treated according to Case 1.

FIG. 3.9. Case 3. Rank-propagation violation at \(A\). The rank of Subtree 1 or the rank of Subtree 2 has decreased to \(r-2\). Demotion at \(A\).

**Lemma 3.7.** An \((a_1, b_1)\)-substring \((s)\) can be expressed as \(y^*z\), where \(y = ((b_1 b_1 b_1) * a_1^+ V_1(b_1 b_1 b_1) * b_1 a_1^+ V_1(b_1 b_1 b_1) * b_1 b_1 a_1^+)\) and \(z = ((b_1 b_1 b_1) * V_1(b_1 b_1 b_1) * b_1 V_1(b_1 b_1 b_1) * b_1 b_1)\).

**Proof.** Obvious.

**Lemma 3.8.** If for a deletion rebalancing sequence \(s\), \(x\) is not the last symbol in an \((a_1, b_1)\)-substring \((s)\), then (a) if \(x = b_1\), then the application of the type-\(x\) operation does not necessitate the addition of any credits to the tree in order to maintain the credit invariant property; (b) if \(x = a_1\), then the application of the type-\(x\) operation releases enough credits from the tree to pay for itself and even that of up to three operations immediately preceding \(x\) in \(s\) (without violating the credit invariant property).

**Proof.** See Fig. 3.15. Consider the application of the type-\(x\) rebalancing operation. Since \(x\) is not the last symbol in the \((a_1, b_1)\)-substring \((s)\), we are justified in assuming that the root of the subtree being rebalanced is black; the node being red would mean the termination of the algorithm with the current operation:

(a) Follows from Fig. 3.15(a).

(b) The corresponding operation releases four credits from the tree which is enough to pay for itself and up to three type-\(b_1\) operations immediately preceding \(x\) in view of (a) above and the fact that the application of a rebalancing operation costs one credit.

**Lemma 3.9.** If for a deletion rebalancing sequence \(s\), three consecutive type \(b_1\) symbols occur in an \((a_1, b_1)\)-substring \((s)\), but do not terminate it, then the execution of the operations releases enough credits to pay for all the three operations (while maintaining the credit invariant property).

**Proof.** See Fig. 3.16. The root of the subtree being rebalanced is black because the \((a_1, b_1)\)-substring \((s)\) does not terminate with the last of the three type-\(b_1\) operations. In Fig. 3.16(a), the rank of Subtree 4 must be \(r-1\) in order to support the rank-gap between \(A\) and the root of Subtree 3; the rank of Subtree 6 must similarly be \(r\) so as to support the rank-gap between \(B\) and the root of Subtree 5. For similar reasons, the rank of Subtree 6 in Fig. 3.16(b) is \(r\) or \(r-1\), the rank of Subtree 6 in Fig. 3.16(c) is \(r-1\), and the rank of Subtree 2 is \(r\) in Fig. 3.16(d). It is clear from Fig. 3.16 that the execution of the operations releases at least three credits from the tree. Hence the result.

**Theorem 3.10.** The rebalancing time for \(m\) deletions from a \(kBB\)-tree of size \(n\) is \(O(k(m+n))\).

**Proof.** The presence of the \(O(kn)\) term in the bound is for the initial tree to satisfy the invariant property. Observe: \(a^*\) denotes zero or more occurrences of \(a\). \(a^+\) denotes one or more occurrences of \(a\). \(V\) stands for disjunction.
FIG. 3.10. Case 4. Only rank-support violation at A. (No potential rank-gap violation.) (a) Case 4a. Ranks of both left/right children of A are r; modified double rotation (composed of two single rotations) at A; This case can occur either because the rank of Subtree 3, Subtree 4, or Subtree 5 has decreased by 1 to r−2. (b) Case 4b. Rank of a left/right child of A is r−2 or less; single rotation at A; This case can occur either because the rank of Subtree 2 or 3 has decreased by 1 to r−2 or as a result of the rebalancing rule for Case 1a or 1c.

That in view of Lemma 3.8(b), Lemma 3.9, and Lemma 3.7, all but the last one operation or the last two operations in an (a₁, b₁)-substring(s), where s is a deletion rebalancing sequence, are paid for through the release of credits from the tree. The result follows now in view of Lemma 3.6.

**Theorem 3.11.** The rebalancing time for a mixed sequence of m insertions and deletions from a kBB-tree of size n is $O(k(m+n))$.

**Proof.** Follows from the following two facts. One, we have used the same credit invariant property for both insertions and deletions. Second, the proofs of Lemmas 3.3 and 3.4, as well as those of Lemmas 3.6 through 3.9, do not depend in any way on whether the kBB-tree being analyzed (for an insertion or deletion) is the result of performing an insertion or deletion.

**Corollary 3.11.1.** The amortized rebalancing time per update for a mixed sequence of m insertions and deletions for a kBB-tree of initial size n is $O(k)$ if $m = \Omega(n)$.

**Corollary 3.11.2.** The amortized rebalancing time for a mixed sequence of insertions and deletions in a kBB-tree that is initially empty, is $O(k)$.

4. SELF-ORGANIZING BALANCED BINARY TREES

Let a set of $n$ (linearly ordered) data items, $x_i$ with frequency of access $w_i$, $1 \leq i \leq n$, be organized as a two-dimensional balanced binary tree (2BB-tree) such that each two-dimensional internal node $N_i$, $1 \leq i \leq n$, stores $x_i$ and has as its middle-subtree any 1BB-tree, $T_i$ with $w_i$ (internal) nodes which lets $w_i$ play a role in the access and rebalancing of the tree. We can minimize the time needed for accessing $x_i$ by letting $T_i$ to be a 1BB-tree of maximum possible rank (and $w_i$ nodes), which is $\lfloor \log (w_i + 1) \rfloor$, according to Lemma 4.2. Actually, we can represent $T_i$ simply by its root which will store $w_i$ and will be of rank $\lfloor \log (w_i + 1) \rfloor$. This motivates the following definition of a “self-organizing balanced binary tree” (see [7, 10, 23]).

A one-dimensional self-organizing balanced binary tree is a 2BB-tree such that the middle (one-dimensional) subtree of each two-dimensional internal node $N_i$ storing a data item with frequency of access $w_i$, $w_i \geq 1$, consists of a single (internal) node that stores $w_i$ is of rank $\lfloor \log (w_i + 1) \rfloor$.

In the rest of this section, we first show that a self-organizing balanced binary tree is nearly optimal and then extend the amortized complexity results for a kBB-tree to such a tree.
FIG. 3.12. Invariant property of the credit scheme. Solid nodes are black; hollow nodes are red. The credits contained in a node or an edge are shown within parentheses. (a) Eight credits assigned to $P$; (b) Two credits assigned to $P$; (c) Zero credits assigned to $P$; (d) $\min(9, 3(r-s-1))$, but at least zero credits assigned to the edge connecting nodes, $P$ and $Q$. $r$ and $s$ are the ranks of $P$ and $Q$, respectively. The credits contained in the edge do not depend upon the colors of $P$ and $Q$.

**Access Time**

We now show through a sequence of results that the “access-time” for each data item stored in a self-organizing balanced binary tree is “ideal” (Theorem 4.9); see Section 1 for the meaning of the term, “ideal.”

A locally supported $k$BB-tree is a $k$BB-tree in which no rank-gap is indirectly supported. Observe that a 1BB-tree which is a balanced binary tree [17] is also a locally supported 1BB-tree. Let $n(r, k)$ denote the minimum possible size of a locally supported $k$BB-tree of rank $r, k \geq 1$.

**Lemma 4.1.** (a) $n(r, 1) = 2^r - 1$, $r \geq 1$; $n(r, 1) = 0$, $r < 1$.

(b) $n(r, 1) = 1 + 2n(r - 1, 1)$, $r \geq 1$.

**Proof.** (a) Follows from the fact that a locally supported 1BB-tree is the same as a balanced binary tree and such a tree of rank $r$ that stores the minimum number of one-dimensional vectors is a completely balanced binary tree of height $r$.\(^6\)

(b) Follows from (a) above.

**Lemma 4.2.** The rank of a 1BB-tree of size, $n$ is $\lceil \log_2(n+1) \rceil$ or less; moreover, the bound is achievable.

**Proof.** $n(r, 1) = 2^r - 1$, $r \geq 1$ (from Lemma 4.1.). Thus, $n \geq n(r, 1) = 2^r - 1$; $n(r, 1)$ being an achievable lower bound on $n$. Hence $\lceil \log_2(n+1) \rceil$ is an achievable upper bound on $r$.

**Lemma 4.3.** For $r \geq 2$, $n(r, 2) = n(r - 1, 1) = 1 + 2n(r - 1, 2)$.

**Proof.** By induction on $r$.

\(^6\) The height of an empty binary tree is zero. The height of a non-empty binary tree is one plus the height of its tallest subtree.
FIG. 3.13. Proof of Lemma 3.4. Promotion causes the release of at least \( 8 + p - 2 - (p + 3) = 3 \) credits from the tree.

**Basis.** \( r = 2 \).

\[
n(2, 2) = 1 = n(1, 1) = 1 + 2n(1, 2), \text{ because } \]

\[
n(1, 2) = 0 \quad \text{(see Fig. 4.1(a)).}
\]

**Induction hypothesis.** For \( 2 \leq r \leq s \), the result is true.

**Induction step.** To prove that the result is true for \( r = s + 1, s \geq 2 \).

**Proof of the Induction Step.** Observe that a locally supported 2BB-tree of rank \( s + 1 \) with minimal size is that three among the following two trees which has the minimum number of one-dimensional internal nodes (see Fig. 4.1(b)): (1) The root of the tree, \( P \), is of rank \( s + 1 \), the middle subtree of \( P \) is a locally supported 1BB-tree (i.e., a balanced binary tree) of rank \( s \) with minimal size, and the left as well as the right subtrees of the root are empty. (2) The root, \( P \), is of rank \( s + 1 \), the middle subtree of \( P \) is of rank 1 (a single internal node) and the left as well as the right subtrees of the root are empty. (The left or the right subtree of \( P \) cannot be of rank \( s + 1 \) because such a tree can only have a larger number of one-dimensional internal nodes. Similarly, if the rank of the middle subtree of \( P \) is less than \( s \) then it should be equal to 1 because such a subtree of higher rank does not reduce the number of one-dimensional internal nodes in the left or right subtree of \( P \), but increases the number of such internal nodes stored in the middle subtree of \( P \).)

Thus, \( n(s + 1, 2) = \min \{ n(s, 1), 1 + 2n(s, 2) \} \).

\[
n(s, 2) = n(s - 1, 1), \text{ by induction hypothesis.}
\]

\[
1 + 2n(s - 1, 1) = n(s, 1), \text{ by Lemma 4.1.}
\]

Thus, \( n(s, 1) = 1 + 2n(s, 2) \).

FIG. 3.14. Proof of Lemma 3.6. Path \( A \) contains left/right edges in any order. Paths \( B \) and \( E \) contain only right edges. Paths \( C \) and \( F \) contain only left edges.
Therefore, \( n(s + 1, 2) = n(s, 1) = 1 + 2ns(2) \).

Hence the result.

**Lemma 4.4.** The rank of a locally supported 2BB-tree of size \( n \) is \( \log_2(n + 1) + 1 \) or less.

**Proof.** \( n(r, 2) = n(r - 1, 1), r \geq 2 \), where \( r \) is the rank of the tree (by Lemma 4.3), \( = 2^{r-1} - 1 \) (by Lemma 4.1). Thus, \( n \geq n(r, 2) = 2^{r-1} - 1 \). Therefore, \( r \leq \log_2(n + 1) + 1 \).

**Lemma 4.5.** Corresponding to each 2BB-tree of rank \( r \) and size \( n \), there is a locally supported 2BB-tree of the same rank but of size \( 2n \) or less.

**Proof.** Given a 2BB-tree, \( T_r \), of rank \( r \) and size \( n \), we show how to transform it into a locally supported 2BB-tree, \( T_S \), of the same rank but of size \( 2n \) or less. A transformation procedure, \( P \), is applied to each two-dimensional node of \( T_r \) in order of the depth of the node, starting from the root of the tree.

**Procedure P.** The procedure (see Fig. 4.2) is applied to a node \( N \) in \( T_r \), only if the subtree rooted at \( N \) satisfies the NONS (no outside neighbor-support) property, which is stated as follows: There is no rank-gap on the two outermost paths of the subtree that is indirectly supported. (Observe that \( T_r \) trivially satisfies this property.)

The purpose of applying Procedure \( P \) to a node \( N \) is to guarantee that the left and right subtrees of \( N \) also satisfy the NONS property. The size of the resulting tree, as we will see below, increases by at most the size of \( M \), the middle subtree of \( N \) only if \( M \) is used actively; i.e., used to replace another subtree. After the application of Procedure \( P \) to each two-dimensional node of the tree, the result is completed, all subtrees of the resulting tree, \( T_S \), will be satisfying the NONS property. (Observe that all one-dimensional subtrees in a kBB-tree satisfy the NONS property.) \( T_S \) is clearly a locally supported 2BB-tree. As we will discuss below, at most all of the one-dimensional subtrees of \( T_S \) will have been used actively once to form \( T_S \). Thus, the size of \( T_S \) will be more than that of \( T_r \) by at most an amount equal to the sum of the sizes of all of its one-dimensional subtrees, i.e., \( n \). The following rules describe the transformation procedure \( P \), summarized in Fig. 4.2.

There are two rules in the procedure. According to the first rule (Fig. 4.2(a)), if the middle subtree of \( N \) has rank \( r = 1 \), then the left and right subtrees are replaced by external nodes. In this case, the rank of the resulting subtree clearly stays the same and the left and right subtrees of \( N \) trivially satisfy the NONS property. Observe that the application of this rule does not increase the size of the subtree; it may actually get reduced. The second rule (Fig. 4.2(b)) applies if the rank of the middle subtree of \( N \), \( s \), is less than \( r = 1 \). The ranks of \( A \) and \( C \) must be at least \( r = 1 \) each in this case because the subtree rooted at \( N \) satisfies the NONS property. In order to define the associated transformation, two nodes, \( B \) and \( D \), are first located on the rightmost path in the subtree rooted at \( A \) and the leftmost path in the subtree rooted at \( C \), respectively, with the following property: \( t_1 \), the rank of \( B \), is greater that but closest to \( s \); \( t_2 \), the rank of \( D \), is greater than by closest to \( s \). The rule causes the replacement of Subtrees 3, 5, 7, and 9 by empty trees (external nodes) and Subtree 6 by a one-node tree. The rule finally causes the replacement of Subtrees 4 and 8, by Subtrees \( I \) and \( J \), respectively. Subtree \( I \) is Subtree 6 if the rank of Subtree 6 is greater than that of Subtree 4; otherwise, it is Subtree 4. Similarly, Subtree \( J \) is the same as Subtree 6 if the rank of Subtree 6 is greater than that of Subtree 8; otherwise, it is Subtree 8.

Observe the following about the second rule. There cannot be a rank-gap between \( A \) and \( B \) that is indirectly supported. If \( t_1 = s + 1 \), then the replacement of Subtree 4 by Subtree \( I \) (of rank \( s \)) ensures that any rank-gap at \( B \), particularly with the replacement of Subtrees 3 and 5 by external nodes, is directly supported. Also, the rank \( f \) of \( B \) will continue to be \( t_1 \). If, on the other hand, \( t_1 > s + 1 \), then there is a rank-gap at \( B \) which cannot be indirectly supported and, hence, must be directly supported. Thus, the rank of Subtree 4 must be \( t_1 - 1 \) and so Subtree 4 will continue to directly support the rank-gaps at \( B \), if any, after the transformation and the rank of \( B \) will continue to be \( t_1 \). The transformation to the subtree rooted at \( C \) is symmetric and exactly the same argument applies. The replacement of Subtree 6 by one-node tree does not cause any rank-gap violation because with the other transformations caused by this rule, the rank-gaps on the rightmost path in the subtree rooted at \( A \) and the leftmost path in the subtree rooted at \( C \) are directly supported. The transformations may increase the size of the subtree rooted at \( N \) but only if Subtree 6 is used actively (to replace a subtree); the maximum increase in size is \( \text{size(Subtree 6)} + 1 = \text{size(Subtree 4)} - \text{size(Subtree 8)} \), which is at most \( \text{size(Subtree 6)} - 1 \).
FIG. 3.16. Proof of Lemma 3.9. Three consecutive type-$b_3$ operations release at least three credits from the tree.

Observe that the left and right subtrees of Nodes $B$ and $D$ in the transformed subtree, using the second rule, are empty. Thus, if the second rule is used and Subtree 6 is used actively, it cannot be used actively again in subsequent transformation(s).

Also note that after the application of the procedure is complete at $N$, the subtrees rooted at $A$ and $C$ will be satisfying the NONS property and, hence, will be ready for further transformation (by Procedure P).

**Lemma 4.6.** The rank of a 2BB-tree of size $n$ is $\log_2(n+1) + 2$, or less.

**Proof.** By Lemma 4.5, corresponding to a maximum-rank 2BB-tree of size $n$, there is a locally supported 2BB-tree of the same rank and of size $2n$ or less. Thus, an upper bound on the rank of a locally supported 2BB-tree of size $2n$ will serve as an upper bound for the 2BB-tree of size $n$. This upper bound, from Lemma 4.4, is $\log_2(2n+1) + 1 \leq \log_2(n+1) + 2$.

Hence the result.

**FIG. 4.1.** Proof of Lemma 4.3. Zero-dimensional subtrees are omitted.
Let $T$ be a self-organizing balanced binary tree. Let $W$ be the sum of the frequencies of access of all items stored in the tree. Let $R(i)$ be the difference between the ranks of the root of $T$ and that of a node $N$ storing an item $x_i$, with $w_i$ as its frequency of access. Let $A(i)$ be the access-time for $x_i$, the time needed for accessing $x_i$.

**Lemma 4.7.** $R(i) \leq \log_2(W + 1) - \log_2(w_i + 1) + 2$.

**Proof.** Follows from Lemma 4.6 and the definition of a self-organizing balanced binary tree.

**Lemma 4.8.** $A(i) = O(\log_2(W/w_i))$.

**Proof.** $A(i) = O(2(R(i)))$, because in the worst-case every two consecutive nodes on the search path will have the same rank,

$$= O(2\log_2(W + 1) - 2\log_2(w_i + 1) + 4)$$

$$= O(\log_2(W/w_i)).$$

**Theorem 4.9.** A self-organizing balanced binary tree has ideal access-time for all items.

**Proof.** Follows from Lemma 4.8.

**Amortized Complexity Results**

A self-organizing balanced binary tree is a particular form of a 2BB-tree. Thus, the results obtained for a kBB-tree in Section 3 are applicable to such a tree. Furthermore, the amortized complexity results proved for the rebalancing time involved in a sequence of insertions and deletions in a kBB-tree are equally true for the rebalancing time for a sequence of "rank-increase of a node by 1" and "rank-decrease of a node by 1" operations.

In a self-organizing balanced binary tree, each access of a data item is followed possibly by a rebalancing of the tree in order to keep the access time ideal for each data item. After each access of a data item, the rank of the node storing the data item can increase by at most 1. The frequency of access of a new data item inserted in a self-organizing balanced binary tree is initialized to 1. This results in replacing an external node (with rank zero) by an internal node with rank 1. We assume that the "deletion" of a data item is not performed unless its frequency of access is 1. Let us call such a deletion as restricted deletion. A restricted deletion replaces a node with rank 1 by an external node with rank zero.

The above observations lead to the following results:

**Theorem 4.2.** The total rebalancing time for a mixed sequence of $m$ accesses, insertions, and restricted deletions in a self-organizing balanced binary tree initially storing $n$ data items is $O(m + n)$.

**Corollary 4.2.1.** The amortized rebalancing time for a mixed sequence of $m$ accesses, insertions, and restricted deletions in a self-organizing balanced binary tree initially storing $n$ data items is $O(1) if m = \Omega(n)$ or if $n = 0$. 
5. CONCLUDING REMARKS

In this paper, we have analyzed the insertion and deletion algorithms for a $k$BB-tree [24] as well as defined and analyzed self-organizing balanced binary trees (which are based on 2BB-trees). We have shown that the amortized rebalancing time for a mixed sequence of $m$ insertions and deletions in a $k$BB-tree is $O(k(m + n))$, where $n$ is the initial size of the tree. It is already known [24] that the height of a $k$BB-tree is logarithmic ($O(\log n k)$) and the number of “rotations” involved in an insertion or a deletion in a $k$BB-tree is $O(k)$. We have used a new approach [25] for the analysis of the rank of a 2BB-tree and obtained for it a tighter upper bound; the upper bound is the same as that for the rank of a 1BB-tree to within a constant additive factor. This upper bound is instrumental in showing that the average access time for a self-organizing balanced binary tree is optimal to within a constant factor while the worst-case access time is logarithmic ($\log W/w_i$). The amortized complexity analysis of $k$BB-trees leads to the result that the amortized rebalancing time for a mixed sequence of $m$ accesses, insertions, and (restricted) deletions in a self-organizing balanced binary tree of initial size $n$, is $O(m + n)$.

The results reported here seem to be interesting and significant. The author is not aware of the availability of similar results for $k$-dimensional trees or self-organizing trees. Similar results are, however, available for one-dimensional trees—a mixed sequence of $m$ insertions and deletions in a balanced binary tree require a total rebalancing time of $O((m + n)\log n)$, where $n$ is the initial size of the tree [9, 12, 18, 19]. $k$BB-trees are a $k$-dimensional generalization of balanced binary trees and are binarized versions of “$k$-dimensional 2–3–4 trees” just as balanced binary trees (same as red–black trees [8] or symmetric binary B-trees [1]) are binarized versions of 2–3–4 trees. Our results for $k$BB-trees thus generalize similar results for balanced binary trees to $k$-dimensions.

Our results on self-organizing balanced binary trees assume that the tree is searched only for data items stored in the tree, i.e., the frequency of access for a data item not in the tree is zero. This restriction can, however, be removed by interpreting each key $k_{i}$ as the half-open interval $[k_{i}, k_{i+1})$, modifying the frequency of access of $k_{i}$ appropriately and distinguishing between $k_{i}$ and $k_{i+1}$ through one additional comparison (see [7, 13] for such an approach).

We have defined (one-dimensional) self-organizing balanced binary trees based on 2BB-trees. It is possible to similarly define $k$-dimensional self-organizing balanced binary trees based on $(k + 1)$BB-trees. However, to do this and to show that such a tree is nearly optimal in access time, a tight upper bound on the rank of a $k$BB-tree needs to be proved. Such a result can be proved similarly as the one for 2BB-trees.

Our amortized complexity results for $k$BB-trees and self-organizing balanced binary trees are for the rebalancing time only. Such results will be practically important if the search time is much faster than $O(\log n k)$ for $k$BB-trees and $O(\log W/w_i)$ for self-organizing balanced binary trees. This is possible if the search tree is augmented with “fingers.” Such trees (with fingers) need to be analyzed for amortized complexity (see [4, 9] for related work).

Our amortized complexity results on self-organizing balanced binary trees follow directly from those for $k$BB-trees. Thus, we do not allow deletion of a data item with an arbitrary frequency of access. It will be interesting to see whether the results can be generalized to a “weighted tree” (see [3, 23]) which supports operations like promotion (of the weight of a data item by an arbitrary amount), demotion (by an arbitrary amount), insertion of a data item with arbitrary weight), and deletion (of a data item with arbitrary weight), besides access.

Using the technique of “node copying” developed in [5, 15], a $k$BB-tree can be made partially persistent, i.e., all versions can be accessed but only the newest version can be modified, such that the time per query or update is $O(\log m + k)$, where $m$ is the total number of updates and the space needed is $O(k)$ per update. This result depends on the fact that an insertion or deletion in a $k$BB-tree involves $O(k)$ number of pointer changes (actually the amortized analysis result given in Corollary 3.11. would be enough).

We leave it for future work to investigate whether an idea similar to “lazy recoloring” [5, 20, 21] can be used to improve the rebalancing time for an insertion or deletion in a $k$BB-tree to $O(k)$. Such a result, if true, would be significant but mainly of theoretical interest as it is expected to be based on a rather complicated modification of the insertion and deletion algorithms [24] for $k$BB-trees. All the same, such a result, coupled with a method developed in [5] for making a search tree fully persistent would possibly lead to fully persistent $k$BB-trees (in which every version can be both accessed and modified) requiring $O(\log n + k)$ time per operation and $O(k)$ space per insertion or deletion.

ACKNOWLEDGMENTS

The author is indebted to the anonymous referees for providing many useful comments and insights and to Linda Davant for her patience in typing the initial version of the paper and drawing the figures.

REFERENCES


