Multidimensional Balanced Binary Trees

Vijay K. Vaishnavi

Abstract—A new balanced multidimensional tree structure called a $k$-dimensional balanced binary tree, $k$ a positive integer, is presented and investigated. It is shown that the data structure can be used to manage a set of $n$ $k$-dimensional data items such that access, insertion, and deletion operations can be supported in $O(\log n + k)$ time. The data structure is a multidimensional generalization of a balanced binary tree and retains much of the simplicity and efficiency of the base structure. The data structure requires $O(k)$ single rotations for each insertion or deletion operation.

Index Terms—Algorithms, balanced trees, data structures, dynamic databases, information storage and retrieval, multidimensional balanced tree structures, multidimensional data structures, multidimensional trees, symmetric binary $B$-trees.

I. INTRODUCTION

The dictionary problem for one-dimensional data consists of finding an efficient implementation of access, insertion, and deletion operations in a list of length $n$. The standard balanced search trees provide a satisfactory solution to this problem, executing each operation in $O(\log n + k)$ time. A balanced binary tree (same as a symmetric binary $B$-tree) is, however, emerging as an optimally-balanced binary tree structure [1]. Olive [9] has shown that half-balanced binary trees, in addition to requiring $O(\log n)$ time for each operation, need only a constant number of single rotations per insertion or deletion. Tarjan [10] has provided similar algorithms for balanced binary trees and shown the equivalence of these trees with half-balanced binary trees.

Efficient organization and maintenance of $k$-dimensional data, $k$ a positive integer, is a basic problem encountered in many areas such as information retrieval, databases, robotics, and computational geometry. In such a problem, a set of $n$ $k$-dimensional (data) items has to be organized into an appropriate data structure such that access, insertion, and deletion operations can be supported efficiently. A balanced $k$-dimensional tree is an appropriately balanced $k$-dimensional tree that can support access, insertion, and deletion operations in $O(\log n + k)$ time per operation, which is optimal [3].

There has been considerable interest recently in developing and investigating balanced $k$-dimensional trees [2], [5], [6], [8], [13]. Most of the available work attempts at generalizing the "balancing concepts" used in balanced one-dimensional trees to higher dimensions. Multidimensional AVL-trees [13] generalize the concept of balancing in AVL-trees to higher dimensions. An important and interesting feature of these trees is that they retain a number of important properties of the base structure. The work reported in this paper demonstrates that such an "optimal" generalization is possible for other balancing concepts too.

The problem addressed in this paper is an "optimal" generalization of balanced binary trees [10] to higher dimensions. We show that the resulting tree structure, multidimensional balanced binary trees, require $O(k)$ number of single rotations per insertion or deletion. Also, these trees use almost the same tree restructuring operations as those used in balanced binary trees and have relatively simple insertion and deletion algorithms.

In Section II, we define $k$-dimensional balanced binary trees and related concepts as well as show that their height is $O(\log n + k)$. We also provide, in this section, a general introduction to the insertion and deletion algorithms which are described in Sections III and IV, respectively. In the final section, we review the contribution of the paper and discuss related issues.

II. PRELIMINARIES

A zero-dimensional (full) binary tree is empty (represented by a zero-dimensional external node with no children). A $k$-dimensional (full) binary tree, $k$ a positive integer, is a tree with the following properties. The tree is either empty (represented by a single $k$-dimensional external node with no children) or comprises of a $k$-dimensional internal node with three children (the left, the middle, and the right child); the left and right children are themselves roots of $k$-dimensional binary trees and the middle child is the root of a $(k - 1)$-dimensional binary tree (nonempty unless $k = 1$).

The following are some definitions related to $k$-dimensional binary trees ($k$ a positive integer) [14].

The height of a subtree is equal to zero if the subtree is empty (represented by an external node); otherwise, it is equal to one plus the height of the tallest subtree of its root. The height of a node is the height of the tree rooted at the node.

Let $T$ be a $k$-dimensional binary tree. If $T$ is empty, then $\text{size}(T) = 0$. Otherwise, we have the following. Let $T_L$, $T_M$, and $T_R$ be the left, middle, and right subtrees of the root of $T$. If $k > 1$ then $\text{size}(T) = \text{size}(T_L) + \text{size}(T_M) + \text{size}(T_R)$ else $\text{size}(T) = 1 + \text{size}(T_L) + \text{size}(T_R)$. Note that $\text{size}(T)$ can also be expressed as the number of one-dimensional internal nodes in $T$.

$P$ is a left (resp., right) descendent of $Q$ if $P$ is the left
Fig. 1. A 2BB-tree. Zero-dimensional external nodes are omitted. The rank of each external node is zero (not shown in the figure). The rank of each internal node is shown beside the node.

A k-dimensional balanced binary tree (kBB-tree), k a positive integer, is a k-dimensional binary tree each of whose nodes has a nonnegative integer rank, denoted by rank(v), such that the ranks have the following properties:

1) Rank-Value Property: Rank(v) = 0 if and only if v is an external node.
2) Rank-Propagation Property: If v has a parent, then
   a) rank(v) ≤ rank(p(v));
   b) if v is the middle child of its parent, then rank(v) < rank(p(v));
   c) if v has the maximum rank among the children of p(v), then rank(p(v)) ≤ rank(v) + 1.
3) Rank-Gap Property: If v is the left or right child of its parent and rank(p(v)) ≥ rank(v) + 2, then for at least one of the neighbors of v, n(v), rank(n(v)) ≥ rank(p(v)) − 1.
4) Rank-Support Property: If v is the middle child of its parent and rank(v) ≤ rank(p(v)) − 2, then for at least one of the neighbors of v, n(v), rank(n(v)) = rank(p(v)) − 1.
5) Rank-Balance Property: If v has a grandparent, then rank(v) < rank(p(p(v))).

Example 2.1: The tree shown in Fig. 1 is a 2BB-tree whose height is 6 and whose size is 14. A is the root of the tree. B, C, and D are the left, middle, and right children of A, respectively. C is a left neighbor of E and E is a right neighbor of C. G and H are right neighbors of F. Rank(E) = 0 and rank(D) = 3. Thus, rank(D) ≥ rank(E) + 2. However, in keeping with the rank-gap property of the tree, there is a neighbor C of E such that rank(C) ≥ rank(D) − 1 (rank(C) = 3). Again, rank(D) = 3 and rank(F) = 1. However, in keeping with the rank-support property of the tree, there is a neighbor G of F such that rank(G) = rank(D) − 1 = 2.

If the ranks of the nodes of a k-dimensional binary tree do not satisfy one or more of the properties listed above, then the tree is said to be a kBB-tree with the corresponding structure violation(s). Fig. 2 illustrates examples of certain types of structure violations at a node A. For example, Fig. 2(b) illustrates a rank-gap violation at A. The difference between the ranks of A and B is more than 1; the rank of A is r while the rank of each of the (two) neighbors of B is r − 2 or less.

Let T be a kBB-tree. Let P be the left or the right child of its parent Q in T (see Fig. 3). There is said to be a rank-gap at Q (between P and Q) if rank(Q) ≥ rank(P) + 2. The width of the rank-gap is said to be rank(Q) − rank(P) − 1. The rank-gap is said to be supported by M if M is a neighbor of P and rank(M) ≥ rank(Q) − 1. The rank-gap may be supported by one or both neighbors of P. If one of the supporting neighbors is the middle child of Q, then the rank-gap is said to be directly supported; otherwise, the rank-gap is said to be indirectly supported. Note that one of the properties of a kBB-tree is that each rank-gap in the tree is supported.

Example 2.2: In the 2BB-tree shown in Fig. 1, the rank-gap between G and H (of width 1) is directly supported by I while
the rank-gap between $D$ and $E$ (of width 2) is indirectly supported by $C$.

In a 1BB-tree, no rank-gap can be supported. It is thus easy to see that a 1BB-tree is the same as a balanced binary tree [11].

A $k$BB-tree can be used as a $k$-dimensional binary search tree; we will always mean it to be as such. A way of storing a set of $k$-dimensional vectors in a $k$-dimensional binary search tree is as follows. The root stores $x_k$, the $k$th component of a vector. The left subtree of the root is a $k$-dimensional binary search tree for those vectors whose $k$th components are less than $x_k$. The right subtree of the root is a $k$-dimensional binary search tree for those vectors whose $k$th components are greater than $x_k$. The middle subtree of the root is a $(k-1)$-dimensional binary search tree for the $(k-1)$-dimensional projection vectors obtained by omitting $x_k$, the identical $k$th component of vectors. Algorithms for searching, inserting, and deleting vectors follow directly from the way data are stored in the tree.

Observe that a $k$-dimensional vector can be converted into a one-dimensional vector by concatenating together all the components of a vector. Thus, in principle, a set of $k$-dimensional vectors can be organized into a one-dimensional binary search tree. However, even in such a balanced one-dimensional search tree, the search time (and hence insertion and deletion time) can be $O(k \log n)$. A balanced $k$-dimensional search tree like a $k$BB-tree can thus be used for storing $k$-dimensional vectors such that they can be searched, inserted, and deleted more efficiently.

**Height of $k$BB-Trees:** The following result is obvious from the rank-balance property of $k$BB-trees:

**Lemma 2.1:** The height of a $k$BB-tree is less than or equal to twice its rank (i.e., the rank of its root).

**Theorem 2.1:** The height of a $k$BB-tree of size $n$ is $O(\log_2 n + k)$.

**Proof:** We first show that we can transform a $k$BB-tree of rank $r$ and size $s$ into a $k$BB-tree of rank $r$ and size $s$ or less such that the resulting tree is also a $k$-dimensional height-
balanced tree [13], [14] of height $r$. The transformation procedure is illustrated in Fig. 4. The procedure is applied to the given tree and then to the tree resulting from an application of the procedure, etc., in a top-down manner starting from the root. In the transformed tree, the height of each node is the same as its rank.

The result follows from Lemma 2.1 and the fact that the height of a $k$-dimensional height-balanced tree of size $n$ is $O(\log_2 n) + k$ [13], [14].

In the next two sections, we describe the insertion and deletion algorithms, prove their correctness, and analyze them. These algorithms involve searching, performing insertion/deletion, and "rebalancing" in a "bottom-up" manner; we concentrate on the rebalancing part of the algorithm. The rebalancing actions involve the application of "restructuring operations"—"rotations,""double rotation,""modified double rotation," and/or updates of ranks (see [13]). All symmetric cases are omitted in the description of the algorithms. The description of the algorithms is interspersed with discussion having bearing on their correctness and time complexity.

III. INSERTION ALGORITHM

Fig. 5 illustrates the insertion of a $k$-dimensional vector in a $k$BB-tree. The root of the subtree that replaces an external node becomes the current node; the rank of the subtree increases from zero to $i$, $1 \leq i \leq k$. It is easy to see that the rank of the parent of the current node, if it exists, cannot be less than $i$. There may be structure violation(s) at the parent and/or the grandparent of the current node. These nodes (or the ones replacing them) are visited starting from the parent of the current node and action is taken according to the rebalancing rules given below. If, on the other hand, the parent of the current node does not exist or if there is no structure violation at the parent or the grandparent of the current node, then the algorithm is terminated.

The current node, in general, is the one on the "restructuring path" whose rank has increased, causing structure violation(s) at its parent and/or its grandparent; the subtree rooted at the current node does not have any structure violation. The increase in the rank of the current node is 1 except at the very start when it is $i$, $1 \leq i \leq k$ (see Fig. 5).

The invariant property of the algorithm is that if the rank of a node (or the one replacing it), $P$, on the restructuring path increases during the execution of the algorithm ($P$ becoming the new current node), then 1) the rank of $P$ is one plus the maximum of the ranks of the children of $P$, and 2) there is at most one (type of) structure violation at the parent of $P$.

Observe that at the very start (Fig. 5), the rank of the middle child of $P$ is $i - 1$. Also, $P$ cannot be the middle child of its parent (if it exists) and the rank of the latter must be $i$ or more. Thus, there can be at most one type of structure violation (a rank-support violation) at the parent of $P$. The invariant property is thus satisfied at the start of the rebalancing part of the algorithm.

A rebalancing step is said to be correct if the rebalanced subtree is free from any structure violation with the ranks of its nodes as indicated, and the invariant property is maintained.
Fig. 4. A transformation procedure for transforming a $k$BB-tree of rank $r$ into a $k$BB-tree of rank $r$ which is also a $k$-dimensional height-balanced tree [13] of height $r$. $1, 2, 3, \ldots$ are names of subtrees. In (a), $i$ is in $\{1, 3\}$, $j$ is in $\{2, 4, 6\}$, and $k$ is in $\{5, 7\}$, such that rank($i$) = max \{rank(1), rank(3)), rank($j$) = max \{rank(2), rank(4), rank(6)), and rank($k$) = max \{rank(5), rank(7)). Similarly in (b), $s$ is in $\{1, 3\}$ and $t$ is in $\{2, 4\}$, such that rank($s$) = max \{rank(1), rank(3)) and rank($t$) = max \{rank(2), rank(4)).

Fig. 5. Insertion results in the replacement of an $i$-dimensional external node, $1 \leq i \leq k$, by a chain of internal nodes. (A "-" beside the rank of a node indicates an increase in the rank of the subtree rooted at the node; such an increase in its rank can cause a structure violation at its parent and/or its grandparent as illustrated in Fig. 6.)

Fig. 6 illustrates the different types of structure violations that may be encountered during the execution of the insertion algorithm. Observe that an increase in the rank of the current node $A$, from $r - 1$ to $r$, cannot cause a structure violation at a node higher than the grandparent of $A$. Even though, in general, it is possible to have a rank-propagation as well as a rank-support violation at the parent of the current node, this type of situation is ruled out in view of the invariant property of the algorithm. Thus, the possible structure violations are

1) no structure violation at the grandparent of $A$ but a rank-support violation only [Fig. 6(a)] or a rank-propagation violation only [Fig. 6(b)] at the parent of $A$ (the latter cannot be the case at the very start of rebalancing)

2) a rank-support violation at the parent of $A$ together with either a rank-balance violation only at the grandparent of $A$ or a rank-balance as well as a rank-support violation at the grandparent of $A$ [Fig. 6(c)]

3) no structure violation at the parent of $A$ but a rank-balance violation at the grandparent of $A$ which may also be coupled with a rank-support violation [Fig. 6(d)].

Rebalancing Rules:

Case 1: There is a structure violation only at the parent of the current node (Fig. 7).

Subcase 1.1: The structure violation at the parent $B$ of the current node is a rank-support violation.

Subcase 1.1.1: The rank of the left or right child of $B$ is $r - 2$ or less [Fig. 7(a)].

Action: Perform an appropriate single rotation and terminate the algorithm.

Correctness: Unless otherwise stated, the following refers to the restructured subtree. The rank of $A$ as $r$ does not cause a rank-propagation violation because the rank of subtree 1 or 2 is $r - 1$ (in view of the invariant property) and the rank of $B$ is $r - 1$ or less (as explained next). The rank of $B$ is $r - 1$ or less because the rank of the subtrees, 3, 4, and 5, is $r - 2$ or less. The rank of the root of subtree 2 must be $r - 1$ in order
Fig. 6. Types of structure violations encountered in the insertion algorithm.

The rank of the current node $A$ has increased to $r$ from $r-1$ except possibly at the very start of rebalancing (see Fig. 5). The rank of each child of $A$ is $r-1$ or less; the rank of at least one of the children of $A$ is $r-1$.

to support the rank-gap between $A$ and the root of subtree 3 in the original subtree (the one being restructured) since $\text{rank}(4) \leq r - 2$; this very node supports the rank-gap between $A$ and $B$, if any. All other rank-gaps continue to be supported. Also, it is easy to see that there cannot be a rank-support violation at $A$ or $B$. The structure violation in the subtree is thus corrected and no new structure violation is created.

Subcase 1.1.2: Both left/right children of $B$ have rank $r$ [Fig. 7(b)].

Action: Promote the rank of the parent of the current node. Treat the parent as the new current node and look for a new structure violation.

Correctness: The structure violation is obviously corrected without introducing any new structure violation. If $B$ is the left or right child of its parent, then there cannot be a structure violation (in the restructured subtree) at the parent of $B$; a rank-propagation violation would mean that there was a structure violation (a rank-balance violation) at the grandparent of the current node in the original subtree, which violates the premise of Case 1. On the other hand, if $B$ is the middle child of its parent, then the only structure violation possible at the parent of $B$ is a rank-propagation violation. Thus, the invariant property is maintained.

Subcase 1.2: The structure violation is a rank-propagation violation.

Subcase 1.2.1: The current node is the middle child of its parent [Fig. 7(c)].

Action: Promote the rank of the parent of the current node. Treat the parent as the new current node and look for a new structure violation.

Correctness: In the restructured subtree, we have the following. Any rank-gap at $B$ is directly supported. The rank of $B$ (which has increased by 1 to $r + 1$) is one plus the maximum rank of its children. If $B$ is the middle child of its parent, then there can only be a rank-propagation violation at its parent. If $B$ is the left or right child of its parent, then there can be a rank-propagation violation or a rank-support violation at its parent. However, both of these structure violations
Subcase 1.2.2: The current node is the left or right child of its parent [Fig. 7(d)].

Action: Perform an appropriate single rotation. Treat the root of the restructured subtree as the new current node and look for a new structure violation.

Correctness: Let us first show that the previous step of rebalancing could have originated from the left or right subtree of the left child of \( B \) in the subtree being restructured.

...
(which may have been different from \( A \)). Suppose to the contrary. Then we have the following. In the previous step, the rank of \( C \), the left child of \( B \), must have been \( r-1 \) and that of the left or right subtree of \( C \) must have been either \( r-1 \) (increased from \( r-2 \), causing a rank-balance violation at \( B \)) or \( r \) (increased from \( r-1 \), causing a rank-propagation violation at the left child of \( B \)). In the former case, a rebalancing rule for Case 2 or Case 3 must have been applied, which contradicts the fact that the left child of \( B \) has become the current node. In the latter case, there must have been a rank-balance violation in the tree even before the insertion and subsequent rebalancing—a contradiction, since such a tree must be a valid \( k \)BB-tree.

It follows from above that the previous rebalancing step must have originated from the middle child of \( A \). The rank of subtree 3 cannot be \( r-1 \) or more because an assumption to the contrary implies a structure violation (a rank-balance or a rank-propagation violation) in the tree even before the increase in the rank of \( A \) as a result of an application of the rebalancing rule for Subcase 1.2.1—a contradiction. If the rank of subtree 5 is \( r-1 \), then the rank of at least one of the subtrees, 3, 4, and the left subtree of the root of subtree 5, must be \( r-2 \). This is because otherwise there is a rank-support violation at \( B \) (in addition to a rank-propagation violation) in the subtree being restructured, contradicting the invariant property. Again, for the same reason as above, if the rank of subtree 5 is \( r-2 \) or less, then the rank of at least one of the subtrees, 3, 4, and 5, must be \( r-2 \). It is now easy to see that the restructured subtree is free from any structure violation.

Observe that if there is a rank-propagation violation at the parent of \( A \) (in the restructured subtree), then there can be a rank-support violation as well at the parent of \( A \) (\( A \) being the left or right child of its parent) only if there was a rank-support violation at the grandparent of \( A \) (in the original subtree)—a contradiction, in view of the premise of Case 1. It thus follows that the invariant property is maintained by the rule.

**Case 2:** There is a structure violation at the parent as well as the grandparent of the current node. The only possible structure violation at the parent of the current node is a rank-support violation.

**Subcase 2.1:** The only structure violation at the grandparent of the current node is a rank-balance violation.

**Action:** Treat this case exactly as Subcase 1.1.1 [Fig. 7(a)].

**Correctness:** Observe that in the restructured subtree, the rank of each child of \( A \) is \( r-1 \) or less. Thus, the rank-balance violation at the grandparent of \( A \) (in the restructured subtree) is automatically taken care of.

**Subcase 2.2:** There is a rank-balance as well as a rank-support violation at the grandparent of the current node (Fig. 8).

**Action:** Perform an appropriate double rotation. Terminate the algorithm if the rank of a left or right child, \( D \), of the grandparent of the current node is \( r-2 \) or less [Fig. 8(a)]. Otherwise, terminate the algorithm after performing an appropriate single rotation at \( C \) [Fig. 8(b)].

**Correctness:** The rank of subtree 4 must be \( r-1 \) in view of the invariant property. Any rank-gap at \( B \) in the restructured subtree is directly supported. Observe that the rank of subtree 1 cannot be \( r \) or \( r-1 \). The restructured subtree in Fig. 8(a) is clearly free from any structure violation.

**Subcase 2.3:** The rank of subtree 8 must be \( r-1 \) because the major gap between \( D \) and the root of subtree 7, in the subtree being restructured, must be supported. It follows that the restructured subtree is free from any structure violation.

**Case 3:** There is structure violation at the grandparent \( C \) but not at the parent of the current node. The structure violation may be a rank-balance violation only or a rank-balance violation together with a rank-support violation.

**Subcase 3.1:** Both, left/right, children of \( C \) have rank \( r \) [Fig. 9(a)].

**Action:** Promote the rank of \( C \). Treat \( C \) as the new current node and look for a new structure violation.

**Correctness:** The restructured subtree is clearly free from any structure violation. It is easy to see that the invariant property is maintained.

**Subcase 3.2:** The rank of the left/right child of \( C \) is \( r-1 \) or less [Fig. 9(b) and (c)].

**Action:** Perform a single rotation or a double rotation as is appropriate [see Fig. 9(b) and (c)] and terminate the algorithm.

**Correctness:** Fig. 9(b): The increase in the rank of the current node (in the subtree being restructured) cannot cause a rank-support violation at its grandparent. Thus, the rank of at least one of the subtrees, 5, 6, and 7, is \( r-1 \). Therefore, the rank of \( C \) in the restructured subtree as \( r \) is correct. Again, the rank of at least one of the subtrees, 3, 4, and 5, is \( r-1 \), in view of the premise of Case 3. Thus, the restructured subtree does not have any structure violation.

**Subcase 3.2:** The rank of the left/right child of \( C \) is less than \( r \) [Fig. 9(c)]. If there is no rank-support violation at \( C \) (in the subtree being restructured), then the rank of \( C \) in the restructured subtree is \( r \). The rank of one of the subtrees, 1, 2, and 3, must be \( r-1 \) in view of the premise of Case 3. Thus, the rank of \( A \) in the restructured subtree as \( r \) is correct. If there is a rank-support violation too at the grandparent of the current node, then the rank of \( C \) in the restructured subtree may be \( r-1 \) or less. Any rank-gap at \( B \) in the restructured subtree is, however, directly supported. This is because, in such a case, there must have been a rank-gap at \( B \) (in the original subtree) which must have been supported by the root of subtree 4.

The correctness of the insertion algorithm follows from the correctness of the rebalancing rules. Thus, we have the following.

**Theorem 3.1:** The insertion algorithm is correct.

As far as the time complexity of the algorithm is concerned, we have the following.

**Theorem 3.2:** The number of single rotations that may be performed for inserting a \( k \)-dimensional vector in a \( k \)BB-tree (using the above insertion algorithm) is \( O(k) \).

**Proof:** Subcase 1.2.2 [see Fig. 7(d)] is the only case which involves the application of a single rotation (or double rotation) such that the restructuring increases the rank of the subtree and thus the algorithm is not terminated. Consider the subtree being restructured in Fig. 7(d). The last step must have originated from the middle subtree of the current node \( A \) (see the proof for the correctness of the rebalancing rule for Subcase 1.2.2). The result follows.
Corollary 3.2.1: The time complexity of the insertion algorithm is $O(\log n + k)$.

Proof: Follows in view of Theorem 2.1.

IV. DELETION ALGORITHM

In order to delete the given $k$-dimensional vector, a maximal chain of $i$ internal nodes, $1 \leq i \leq k$, has to be deleted such that at least all but the highest node in the chain have external nodes as their left and right children. Without loss of generality, we assume that at least one of the children of the highest node in the chain is an external node. Fig. 10 illustrates the actual deletion. A subtree of rank $r$, $1 \leq i \leq k$, is replaced by one of rank zero [Fig. 10(a)], a subtree of rank $r > i$ is replaced by one of rank $r - 1$ [Fig. 10(b)], or a subtree of rank $i$ is replaced by one of the same rank. In the last case as well as in case $s = r$ in the one preceding it, the algorithm is terminated because the deletion cannot cause any structure violation. Observe that the root of the replacing subtree cannot be the middle child of its parent.

If there is a new rank-gap as the result of performing the deletion which is not directly supported, then an appropriate rebalancing rule (described below) is applied; otherwise, the algorithm is terminated. Note that if an already existing rank-gap is widened, then there is no need for any action because the rank-gap must already be supported. Also, observe that even if the newly created rank-gap is not directly supported it may still be supported (i.e., indirectly supported). The algorithm does not, however, check for an indirect support, for the sake of efficiency. The application of a rebalancing rule may result in a decrease in the rank of a node $P$ by 1 (the decrease may be from the rank of the node replaced by $P$). The decrease in the rank of $P$ may cause a structure violation at its parent and/or its grandparent, or at one or both neighbors of $P$ (in case $P$ is the middle child of its parent). A newly
Fig. 9. Insertion restructuring when there is no structure violation at the parent of the current node but there is structure violation at the grandparent of the node. (a) Promotion. (b) Single rotation at C. (c) Double rotation at C.
created rank-gap that is not directly supported or a rank-gap that was indirectly supported by \( P \) and is no longer supported by it may possibly cause a rank-gap violation. For the purposes of this algorithm, such a rank-gap is assumed to be causing a rank-gap violation and is treated by the algorithm.

Fig. 11 illustrates the different types of structure violations that may be encountered during the execution of the algorithm, which are treated using the rebalancing rules. Fig. 11(a) shows a rank-gap violation at \( Q \) (actually a rank-gap that is not directly supported) corresponding to Fig. 10(a). Fig. 11(b) shows a rank-gap violation at \( Q \). Fig. 11(c) shows a rank-gap violation as well as a rank-support violation at \( Q \). Fig. 11(d) shows a rank-propagation violation. Fig. 11(e) shows a rank-gap violation at \( Q \) which may be coupled with a rank-support violation (at \( Q \)). Fig. 11(f) shows a rank-support violation at \( Q \) caused by the decrease in the rank of \( P \) (and no rank-gap violation). Fig. 11(g) and (h) shows rank-gap violations at one neighbor of \( P \) (actually there can be rank-gap violations at both neighbors of \( P \)). There may also be a rank-support violation at \( Q \) in Fig. 11(g) (only if the rank of \( Q \) is \( r \)). In Fig. 11(a)–(d), there may also be a rank-support violation at \( Q \) or only such a violation (and no rank-gap violation). Observe that the rank of the parent of \( P \) is \( i + 1 \) in Fig. 11(a) and \( r \) in Fig. 11(b) and (c). A larger value of the rank would imply that the rank-gap is not newly created because of the decrease in the rank of \( P \) and hence need not be treated. A smaller value of the rank would imply that there is no rank-gap or there is one but it is directly supported. Again, observe that the rank of the parent of \( P \) is \( r \) in Fig. 11(e) and the rank of the neighbor of \( P \), \( R \), is \( r \) in Fig. 11(g) and (h). This is in view of the fact that a
Fig. 11. Types of structure violations encountered and corrected in the deletion algorithm. A "**" beside a node indicates a (possible) structure violation at the node. Except in (a), the decrease in the rank of P is from \( r - 1 \) to \( r - 2 \). (A curvy line stands for a path comprised of zero or more tree edges.)

rank-gap which does not lose its support because of the decrease in the rank of P need not be treated.

If there are rank-gap violations at a left as well as a right neighbor of P (P being the middle child of its parent), then the following is done (see Fig. 12). The left and right children of the parent of P are designated as boundary nodes. The parent is designated as the junction node. One of the neighbors of P (with the rank-gap violation) is marked and restructuring is initiated at the other neighbor (with the rank-gap violation), using the rebalancing rules.

A rebalancing rule when applied at an \( i \)-dimensional node \( A, 1 \leq i \leq k \), can result in the following.

1) The rank of the (restructured) subtree does not change but there is still a structure violation in the subtree (which will get corrected in the next step). See Fig. 13(b), (c), and (d).

2) The rank of the (restructured) subtree stays the same and there is no structure violation in the subtree. See Fig. 13(b), (c), (d), (e), and (f).

3) The rank of the subtree decreases by 1 and there is no structure violation in the subtree. See Figs. 13(a) and 14.

4) The rank of the subtree decreases by 1 but there is still a structure violation in the subtree (which will get corrected in the next step). See Fig. 13(a).

After a rebalancing rule is applied, the next action taken depends on what the effect of the restructuring has been (i.e., which of the above is the case), whether \( A \) is a boundary node, and whether there is an \( i \)-dimensional marked node.

If 1) is the case, then an appropriate rebalancing rule is applied.

If 2) is the case, then the following is done. If there is no \( i \)-
Fig. 12. Rank-gap violations at neighbors of \( P \) (on both of its sides). The hatched nodes are the boundary nodes. (A curvy line stands for a path comprised of zero or more edges.)

dimensional marked node or there is one but the structure violation at the node has already been corrected, then the \( i \)-dimensional junction node is examined for a (possible) structure violation; if there is a structure violation, then it is treated using the rebalancing rules, else the algorithm is terminated. If, on the other hand, there is an \( i \)-dimensional marked node and the structure violation at the node has not been treated yet, then an appropriate rebalancing rule is applied at the node.

If 3) is the case, then the following is done. If \( A \) is a boundary node, then the same action is taken as in 2) above. If \( A \) is not a boundary node, then the parent of \( A \) is examined for (possible) structure violation(s). If there is a structure violation, then it is treated using the rebalancing rules; else, the same action is taken as in 2) above.

If 4) is the case, then the same action is taken as in 3) above after applying an appropriate rebalancing rule.

A rebalancing rule is said to be correct if, with the ranks as indicated, the restructured subtree is either free from structure violations or it has a (possible) structure violation which is, however, guaranteed to be corrected in at most the next step.

Rebalancing Rules: Let \( A \) be an \( i \)-dimensional node, \( 1 \leq i \leq k \), with rank \( r \), at which there is a (possible) structure violation. Let \( B \) and \( D \) be its left/right children. In each of the following cases, first the described action is taken and then a suitable action, as discussed above, is taken.

Case 1: There is at least a (potential) rank-gap violation at \( A \) (see Fig. 13). (The rank-gap violation is either because the rank of \( D \) (not shown in the figure) has decreased or because the rank-gap between \( A \) and \( D \) has lost its support.)

Subcase 1.1: The rank of \( B \) is \( r - 1 \). (Observe that there cannot be a rank-support violation at \( A \).)

Subcase 1.1.1: The ranks of all the children of \( B \) are \( r - 2 \) or less [Fig. 13(a)].

Action/Result: Demote the rank of \( A \). The result of the action is that the rank of the resulting subtree has decreased by
Fig. 13. Deletion restructuring when there is at least a rank-gap violation at A. D, the left child of A, is not shown. (a) Demotion (of rank). Any structure violation will get corrected in the next step. (b) Single rotation at A. If there still is a structure violation in the subtree (a rank-gap violation at B because s \( \leq r - 2 \)), it will get corrected in the next step. (c) Single rotation at A. If there still is a structure violation in the subtree (a rank-support violation at A), it will get corrected in the next step.
Fig. 13. (Continued). (d) Double rotation at A. If there still is a structure violation in the subtree (a rank-gap violation at C because s ⩽ r - 2), it will get corrected in the next step. (e) Single rotation at A.

1. If the structure violation at A was because a new rank-gap got created at A, then the resulting subtree is free from any structure violation. Otherwise, there may be a rank-support violation at A (and no other structure violation). In the latter case, apply an appropriate rebalancing rule.

Correctness: If the rank-gap violation was because of the decrease in the rank of the left/right child of A from r - 1 to r - 2 or to zero (in the latter case, see Fig. 11(a), the rank of subtree 2 must be r - 2), then it is obviously corrected for the rank-gap vanishes because of the demotion of A or is directly supported. Also, no other structure violation is introduced in the resulting subtree. If, on the other hand, the rank-gap violation was because the rank-gap at A had lost its support due to the decrease in the rank of its neighbor P from r - 1 to r - 2 or if the rank of P is r - 2 (applicable if the present case results from the action taken in Subcase 1.1.2 or Subcase 1.1.4), then in the resulting subtree the rank-gap at A, if any, is clearly supported by P. Observe that there cannot be a rank-support violation at A or rank-support violation at B in the resulting subtree because the rank of at least one of the subtrees, 3, 4, and 5, is r - 2. There may, however, be a rank-support violation at A in case the ranks of the subtrees, 1, 2, and 3, are r - 3 or less. The action to be taken to correct this structure violation will be according to Case 3.

Subcase 1.1.2: The 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 [Fig. 13(b)].

Action/Result: Perform an appropriate single rotation at A. If the structure violation at A (in the original subtree) was because a new rank-gap got created at A, then the resulting subtree (rooted at B) is free from any structure violation. Otherwise, there may still be a rank-gap violation at B (and no other structure violation). In either case, the rank of the subtree stays as r. If there still is a structure violation in the subtree, then apply an appropriate rebalancing rule.

Correctness: The rank of subtree 5 is r - 1. Thus, it is correct to have the rank of B in the restructured subtree as r. There is obviously no rank-support violation at B or at A (in the restructured subtree). If the rank of subtree 1 or 2 is r - 2 (which will be the case if the structure violation at A was because of a decrease in the rank of D), then the resulting subtree (rooted at B) is obviously free from any structure violation. Otherwise, if there is a rank-gap violation at B (because s ⩽ r - 2), then the structure violation will get corrected in the next step without introducing any new structure violation, in view of the following. The rank of each of the children of the root of subtree 5 must be r - 2 or less because otherwise there is a structure violation at B in the
fig. 14. Deletion restructuring when there is a rank-propagation violation at A. Demotion.

Subcase 1.1.1: The rank of each of the left/right children of B is \( r - 1 \) [Fig. 13(c)]

Action/Result: Perform an appropriate single rotation. The resulting subtree (rooted at B) will be free from any structure violation except that there may be a rank-support violation at A (Case 3). The latter will not be the case if the rank-gap violation at A was because of a decrease in the rank of D. The rank of the resulting subtree stays as \( r \). If there still is a structure violation in the subtree, then apply an appropriate rebalancing rule.

Correctness: In the restructured subtree, we have the following. There is no structure violation at B. The rank-gap violation at A obviously gets corrected. There may be a rank-support violation at A but only if the rank of either subtree 1 nor subtree 2 is \( r - 2 \). The action to be taken to correct the rank-support violation, if any, will be according to Case 3.

Subcase 1.1.2: The rank of the 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 [Fig. 13(d)].

Action/Result: Perform an appropriate double rotation at A. The resulting subtree (rooted at C) will be free from any structure violation except that there may be a rank-gap violation at C (Subcase 1.1.1). The latter will not be the case if the rank-gap violation at A was because of the creation of a new rank-gap at A. The rank of the resulting subtree stays as \( r \). If there still is a structure violation in the subtree, then apply an appropriate rebalancing rule.

Correctness: The rank of at least one of the subtrees, 5, 6, and 7, must be \( r - 2 \) because otherwise there is a rank-support violation at B in the subtree being restructured. Thus, it is correct to have the ranks of B and C as \( r - 1 \) and \( r \), respectively. There is obviously no rank-support violation at A, B, or C (in the restructured subtree). There may, however, be a rank-gap violation at C but only if the ranks of the subtrees, 1, 2, and 3, are \( r - 3 \) or less. The action to be taken to correct the rank-gap violation, if any, will be according to Subcase 1.1.1. The application of the rebalancing rule for Subcase 1.1.1 will not create any structure violation because in such a ease, the rank of subtree 4 must be \( r - 2 \) in order to support the rank-gap between C and the root of subtree 3 (in the subtree being restructured).

Subcase 1.2: The rank of B is \( r \) [Fig. 13(e)]. There is a rank-gap violation only or a rank-gap as well as a rank-support violation at A.

Action/Result: Perform an appropriate single rotation. The resulting subtree (rooted at B) is free from any structure violation.

Correctness: The rank of at least one of the subtrees, 3, 4, and 5, must be \( r - 1 \) because otherwise there is a rank-propagation violation at B. In view of this, there is no rank-propagation, rank-balancing, or rank-support violation at B (in the restructured subtree). There is obviously no structure violation at A. If the rank of subtree 3 is \( r - 2 \) or less, then the rank of subtree 4 must be \( r - 1 \) in order to support the rank-gap between B and the root of subtree 3 (in the subtree being restructured). Thus, any rank-gap between A and B in the restructured subtree is supported.

Case 2: There is a rank-propagation violation at A (Fig. 14).

Action/Result: Demote the rank of A to \( r - 1 \). The restructured subtree (rooted at A) is free from any structure violation while its rank has decreased by 1.

Correctness: Obvious.

Case 3: There is a rank-support violation and no rank-gap violation at A (Fig. 15).

Action/Result: Perform an appropriate "modified double rotation" at A if this case has not resulted from Case 1.1.1 or Case 1.1.3; otherwise, perform an appropriate single rotation. The restructured subtree is free from any structure violation and its rank stays the same (i.e., \( r \)).

Correctness: Fig. 15(a): Observe that in the subtree being restructured, the ranks of subtrees 2 and 6 must be \( r - 1 \) because otherwise there would be a rank-gap violation at B or C. It is easy to see the correctness of the indicated ranks of nodes in the restructured subtree. The fact that the restructured subtree is free from any structure violation can easily be verified.

Fig. 15(b): Observe that this situation may result because of a decrease in the rank of the root of subtree 3 (by 1) to \( r - 2 \) or because of an application of the rebalancing rule for Case 1.1.1 or Case 1.1.3. Note that the rank of subtree 4 must be \( r - 1 \) because otherwise there would be a rank-gap violation at B. Similarly the rank of subtree 5 must be \( r - 1 \) or less because otherwise there would be a rank-balancing violation at A. It is easy to see that the restructured subtree is free from any structure violation.

Theorem 4.1: The deletion algorithm is correct.

Proof: Observe that all the different types of structure violations that may be encountered during the execution of the algorithm (see Fig. 11) are taken into account in the rebalancing rules except that for Cases 1.1.1--1.1.4 and Case 2 there may also be a rank-support violation at the parent of A. It is easy to see that the action taken for these cases also corrects such a structure violation. The correctness of the algorithm follows from the correctness of the rebalancing rules.
Theorem 4.2: The number of single rotations that may be performed for deleting a $k$-dimensional vector from a $k$BB-tree is $O(k)$.

Proof: Observe from the correctness of the rebalancing rules that any time the rank of a node that is the left or right child of its parent decreases, resulting in a rank-gap violation, it gets corrected after a demotion (Subcase 1.1.1 or Case 2) or a single/double rotation (the rest of the cases). In the latter situation, note that the restructured subtree is free from any structure violation and its rank does not decrease. On the other hand, the decrease in the rank of a node that is the middle child of its parent can, in the worst case, result in a decrease in the rank of a subtree by 1 after one single/double rotation has been applied (an application of the rule for Subcase 1.1.1 followed by that for Case 3; an application of the rule for Subcase 1.1.2 followed by that for Subcase 1.1.1; an application of the rule for Subcase 1.1.4 followed by that for Subcase 1.1.1). □

Corollary 4.2.1: the time complexity of the deletion algorithm is $O(\log_2 n + k)$.

Proof: Follows in view of Theorem 2.1.

V. CONCLUDING REMARKS

In this paper, we have defined and investigated a new class of balanced $k$-dimensional trees called $k$-dimensional balanced binary trees. These trees generalize balanced binary trees to higher dimensions in such a way that the resulting tree structures retain important properties of the base structure such as the following. They require $O(k)$ number of single rotations for each insertion or deletion operation. Also, the insertion and deletion algorithms are relatively simple and use similar tree restructuring operations as those used in balanced binary trees.

$k$-dimensional balance binary trees have the distinction of being the only balanced $k$-dimensional tree structure known to the author that requires $O(k)$ tree restructuring operations per insertion or deletion operation. This property is expected to play a significant role in obtaining optimal implementation of localized search for $k$-dimensional data as well as in making $k$-dimensional search trees persistent (see [4]). A similar property of balanced binary trees has led to important applications of the data structure for one-dimensional data [4], [12].

The height analysis of $k$-dimensional balanced binary trees has been rather gross in this paper. A tight bound on the height of these trees has been left for future research (see [13] and [14]).

An important application of a balanced $(k + 1)$-dimensional tree structure is in defining an efficient dynamic
weighted $k$-dimensional tree structure by letting the lowest level subtrees be used as virtual subtrees representing the weights of items [7]. The investigation of $k$-dimensional balanced binary trees for such an application is under investigation (see [15]).

ACKNOWLEDGMENT

The author is indebted to the anonymous referees for their useful comments and to R. M. Vega for her excellent typing.

REFERENCES


Vijay K. Vaishnavi was born in Srinagar, Kashmir, India. He received the B.E. degree (with distinction) in electrical engineering from Jammu and Kashmir University in 1968, and the M. Tech and Ph.D. degrees in electrical engineering from the Indian Institute of Technology, Kanpur.

He is currently a Professor of Computer Information Systems at Georgia State University, Atlanta, where he has been since 1981. He has authored and refereed numerous papers in algorithms, data structures, and other related areas of computer science.

His papers have appeared in IEEE TRANSACTIONS ON COMPUTERS, SIAM Journal on Computing, Acta Informatica, Journal of Algorithms, and several other major international journals and conference proceedings. His current research interests include data structures, algorithms, and artificial intelligence.

Dr. Vaishnavi is a member of Association for Computing Machinery, SIGACT, SIGART, SIGMOD, the European Association for Theoretical Computer Science, and the IEEE Computer Society.