Algorithmica (2000) 28: 255–267
DOI: 10.1007/s004530010039

Algorithmica
© 2000 Springer-Verlag New York Inc.

Simple Algorithms for the On-Line Multidimensional Dictionary and Related Problems

T. F. Gonzalez[1]

Abstract. The on-line multidimensional dictionary problem consists of executing on-line any sequence of the following operations: INSERT(p), DELETE(p), and MEMBERSHIP(p), where p is any (ordered) d-tuple (or string with d elements, or points in d-space where the dimensions have been ordered). We introduce a clean structure based on balanced binary search trees, which we call multidimensional balanced binary search trees, to represent the set of d-tuples. We present algorithms for each of the above operations that take $O(d + \log n)$ time, where n is the current number of d-tuples in the set, and each INSERT and DELETE operation requires no more than a constant number of rotations. Our structure requires dn words to represent the input, plus $O(n)$ pointers and data indicating the first component where pairs of d-tuples differ. This information, which can be easily updated, enables us to test for MEMBERSHIP efficiently. Other operations that can be performed efficiently in our multidimensional balanced binary search trees are: print in lexicographic order ($O(dn)$ time), find the (lexicographically) smallest or largest d-tuple ($O(\log n)$ time), and concatenation ($O(d + \log n)$ time). Finding the (lexicographically) kth smallest or largest d-tuple can also be implemented efficiently ($O(\log n)$ time), at the expense of adding an integer value at each node.

Key Words. Multidimensional dictionary, On-line algorithms, Data structures.

1. Introduction.

The on-line 1-dimensional dictionary, or simply the dictionary, problem consists of executing on-line any sequence of instructions of the form MEMBERSHIP(p), INSERT(p), and DELETE(p), where each p is a point (object) in 1-space. We assume that each point can be stored in a single word, and that it can be accessed in $O(1)$ time. It is well known that any of these three instructions can be carried out in $O(\log n)$ time, where n is the current number of points in the set, when the set is represented by AVL-trees, B-trees (of constant order), 2-3 trees, balanced binary search trees (i.e., symmetric B-trees, half-balanced trees, or red–black trees), or weight balanced trees. All of these trees are binary search trees, with the exception of B-trees which are m-way binary search trees. The balanced binary search trees require only $O(1)$ rotations for both the INSERT and DELETE operations [11], [13].

In this paper we consider the case when the data is multidimensional, i.e., ordered d-tuples, which we refer to simply as d-tuples, whose components are real values. It is assumed that each real value can be stored in one memory location. Multidimensional dictionaries have a multitude of uses when accessing multiattribute data by value. These applications include the management of geometrical objects and the solution of geometry search problems. For example, the efficient approximation algorithms in [3] use the abstract data type implemented in this paper to find suboptimal hyperrectangular covers

[1] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA. teo@cs.ucsb.edu.

Received June 13, 1997; revised September 3, 1998. Communicated by H. N. Gabow.
Online publication May 18, 2000.

for a set of multidimensional points. This covering problem has applications in image processing, and in locating emergency facilities so that all users are within a reasonable distance of one of the facilities [3].

The current set of d-tuples is denoted by P and each d-tuple $p \in P$ has coordinate values given by $(x_1(p), x_2(p), \ldots, x_d(p))$. We examine several data structures to represent a set of d-tuples P and develop algorithms to perform on-line any sequence of the multidimensional dictionary operations. We show that the three operations can be performed in $O(d + \log n)$ time, where n is the current number of d-tuples in the set and d is the number of dimensions. Furthermore, only a (small) constant number of rotations are required for each INSERT and DELETE operation. The space required by our algorithm is dn words to represent the d-tuples, plus $O(n)$ words for pointers and data. Other operations that can be performed efficiently in our multidimensional balanced binary search trees are: find the (lexicographically) smallest or largest d-tuple ($O(\log n)$ time), print in lexicographic order ($O(dn)$ time), and concatenation ($O(d + \log n)$ time). Finding the (lexicographically) k th smallest or largest d-tuple can also be implemented efficiently ($O(\log n)$ time), at the expense of adding an integer value at each node.

As noted in [10], it was a common belief two decades ago that “balanced tree schemes based on key comparisons (e.g., AVL-trees, B-trees, etc.) lose some of their usefulness in this more general context.” Because of this researchers combined TRIES with different balanced tree schemes to represent multikey sets (i.e., d-tuples). We now elaborate on this structure. A TRIE is used to represent strings (assume all have the same length) over some alphabet Σ by its tree of prefixes. There are several implementations of TRIES:

1. Each internal node in a TRIE is represented by a vector of m pointers, where m is the number of elements in Σ. A function, normally computable in constant time, transforms each element in Σ into an integer in $[0, m - 1]$ (see the structure in Figure 1(a), where $\Sigma = \{0, 1, 2, 3\}$).
2. Each internal node in the TRIE is represented by a linear list (see the structure in Figure 1(b)) [12].
3. Each internal node in the TRIE is represented by a binary search tree (see the structure in Figure 1(c)) [2].

The structures in Figure 1 represent the set $\{1133, 1232, 2131, 2132, 3113, 3121, 3213\}$. For large m or when we have vectors upon which only comparisons are possible (such as real values), method 1 is not suitable. In this case we can represent each node in the TRIE by a linear list of tuples each storing an element and a pointer (see Figure 1(b)), or a binary search tree replacing the list (see Figure 1(c)).

Bentley and Saxe [1] used the TRIE plus binary search tree representation just described, but with all trees and subtrees fully balanced. The full balancing is defined as follows. If one erases all the “middle” pointers (solid lines in Figure 1(c)), the structure is partitioned into a set of binary search trees. The root of each of these binary search trees and all their subtrees is such that the number of leaves (reachable through the left, middle, and right pointers) in their left and right subtrees differ by at most one. This balanced structure is very useful for static search problems like sorting or restricted searching [7], [8], but these fully balanced subtrees are very rigid structures which cannot be easily updated. Therefore, this structure is not appropriate for dynamic updates, and should be replaced by more flexible structures in dynamic environments.

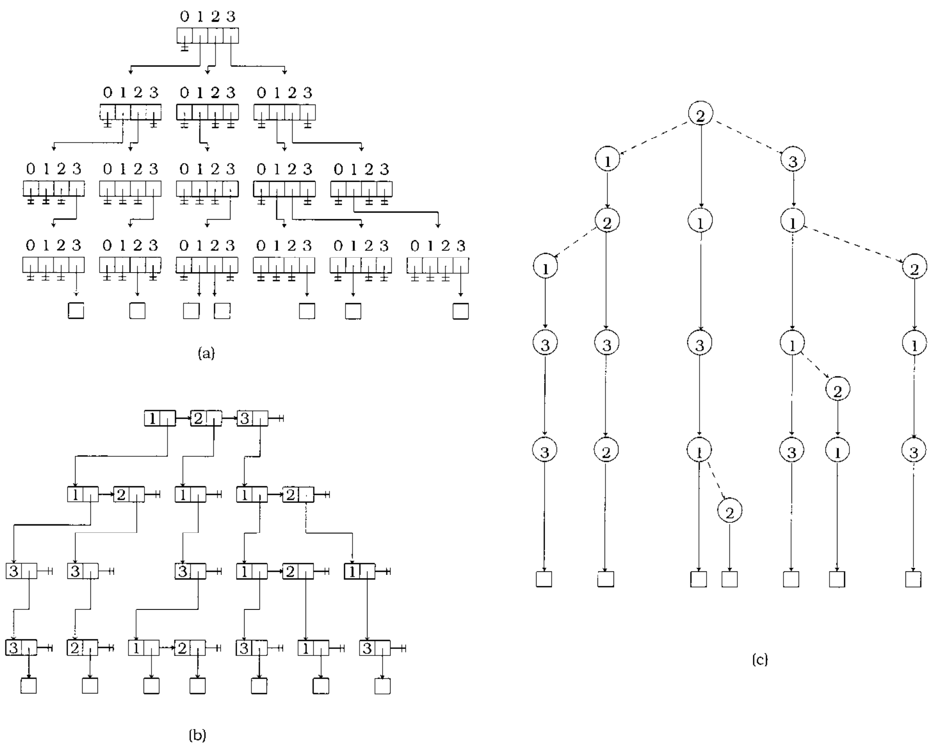


Fig. 1. (a) TRIE representation. (b) Linked list representation for TRIE nodes. (c) Binary search representation for TRIE nodes. (Dashed arcs are binary search tree pointers and other arcs are TRIE pointers.)

Balancing of the TRIE plus binary search tree structure can be performed in different ways by using techniques related to fixed order B-trees [5], weight balanced trees [10], AVL-trees [14], and balanced binary search trees [15]. In each of the resulting representations each of the three multidimensional dictionary operations can be implemented to take $O(d + \log n)$ time in the worst case (for some of these algorithms the time complexity bound is amortized, e.g., [10]). However, the number of rotations after each INSERT and DELETE operation is not bounded by a constant and the procedures are quite involved. An interesting open problem raised by Vaishnavi [15] is to find a way to implement multidimensional dictionaries within the above time complexity bounds that would only require a constant number of rotations for each INSERT and DELETE operation.

Before we explain our approach, we discuss two naive procedures to test for MEMBERSHIP. Suppose that we represent our set of d-tuples by a balanced binary search tree in which each node stores a d-tuple and the ordering of the d-tuples in the tree is lexicographic. MEMBERSHIP can be implemented in the obvious way, i.e., compare the d-tuple you are searching for with the d-tuple stored at the root of the current subtree and depending on the outcome it either terminates (having found it) or it proceeds to the left or right subtree of that node. Clearly, the time complexity for the above procedure is $O(d \log n)$ and there are problem instances for which it actually requires $\Theta(d \log n)$ time.

We now modify the above procedure and reduce its time complexity to $O(d + \log n)$. The new procedure, FAST-NAIVE, is similar, except that instead of comparing d-tuples starting always at position 1, we begin the comparison where we stopped at the end of the previous iteration. The d-tuple stored at a node is referred to by v and its components can be accessed via $x_1(v), x_2(v), \ldots, x_d(v)$. Let p and q be two d-tuples. For $1 \le i \le d$, we define *diff*(p, q, i) as the index of the first component starting at i where p and q differ or $d + 1$ (i.e., smallest integer j greater than or equal to i such that $x_k(p) = x_k(q)$, $i \le k < j$, and $x_j(p) \ne x_j(q)$, unless no such j exists, in which case j is $d + 1$). In what follows we say that j is the index of the first component starting at i where p and q differ when j is equal to *diff*(p, q, i). When i is 1 we say that j is the index of the first component where p and q differ. Procedure FAST-NAIVE is given below:

```
procedure FAST-NAIVE($q, r$);
    $t \leftarrow r$;
    $i \leftarrow 1$;
    while $t \ne null$ do
        $j \leftarrow diff(q, t, i)$;
        case
            :$j = d + 1$: return(true);
            :$x_j(q) < x_j(t \rightarrow v)$: $t$ is set to point to the left subtree of $t$;
            :$x_j(q) > x_j(t \rightarrow v)$: $t$ is set to point to the right subtree of $t$;
        endcase
        $i \leftarrow j$;
    endwhile
    return(false);
end of procedure FAST-NAIVE;
```

It is simple to show that the time complexity for procedure FAST-NAIVE is $O(d + \log n)$, since the total number of operations performed by the algorithm is proportional to the length of the path from the root to the current node t plus the value for j. We now apply procedure FAST-NAIVE to search for d-tuples in the tree given in Figure 2. When procedure FAST-NAIVE is invoked with $q = (1, 0, 0, 0, 0)$, it sets t to the root of the tree and j to 2. Procedure FAST-NAIVE then advances to the left child of t and j is set 3. Then t is advanced to the left subtree of t and j is set to 4. The value of t is then set to the left subtree of t, and since it is *null* the procedure returns the value of false, which is the correct answer. When searching for d-tuple (2, 3, 0, 8, 7) the procedure returns the value

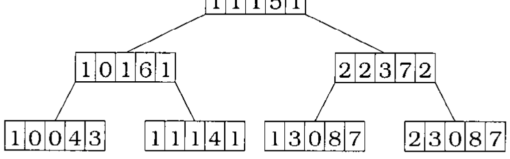


Fig. 2. Balanced binary search tree.

of true, which is the correct answer. However, the search for $(1, 1, 1, 4, 3)$ returns the value of true which is incorrect. One can eliminate this mistake by comparing q to $t \rightarrow v$ when procedure FAST-NAIVE claims success. Unfortunately, the new procedure is also incorrect. Searching for $(1, 3, 0, 8, 7)$ and $(1, 1, 1, 4, 1)$ will generate incorrect results. Note that procedure FAST-NAIVE performs the search for $(1, 3, 0, 8, 7)$ correctly, but the new procedure does not generate a correct solution. The main reason why the above procedure does not work correctly is that the prefix of the d-tuples stored at a node and at its children nodes can vary considerable. However, this variation is more predictable when comparing against the smallest or largest d-tuple in a subtree. This is a key idea exploited in [4].

Manber and Myers [9] also studied the static membership problem arising when given text (a static string) consisting of N symbols, and then a sequence of queries is given, each of which requests reporting all occurrences of a given string in the text. The way they solve their problem is to create a sorted list of all N suffixes of the text. Then the query is answered by performing a binary search to locate all the suffixes that begin with the string being searched. Since the list is sorted all of these suffixes appear in adjacent entries in the list. The advantage of this structure is that it requires only $O(N)$ space, and it can be constructed in $O(N)$ expected time [9]. The way they search for an element is a modified version of binary search. Initially L is the smallest string in the list and R is the largest string in the list. First they compute l, the index of the first component where q, the string they are searching for, and L differ. They also compute r, which is the index of the first component where R and q differ. Then the middle entry, M, in the remaining part of the list is located, and they compute the index of the first component where M and q differ. This cannot be done directly, otherwise the time complexity would not be the one they claim. They find this in a clever way by using l, r, as well as the index of the first component where M and L differ, and M and R differ. This last set of values have been precomputed, and would be expensive to recompute in a dynamic environment. However, in their application the text is static. This list plus binary search when viewed as a binary search tree corresponds to the fully balanced tree strategy in [1]. However, the additional information added allows testing for membership in $O(d + \log N)$ time, where d is the number of symbols in the string q.

Gonzalez [4] solved Vaishnavi's open problem mentioned above. To achieve the proposed time complexity bound he represents the set of points P in a balanced binary search tree in which additional information has been stored at each node. For every node v this information includes the index of the first component where v and the smallest d-tuple in the subtree rooted at v differ, the index of the first component where v and the largest d-tuple in the subtree rooted at v differ, as well as pointers to these d-tuples. This is similar to the information stored in the suffix lists by Manber and Myers [9]. The tree structure in [4] can be updated dynamically, whereas the preprocessed information in the list in [9] cannot be updated efficiently. However, testing for membership in the procedure in [9] is a little simpler. Gonzalez [4] developed for his structures $O(d + \log n)$ time procedures for INSERT, DELETE, and MEMBERSHIP that require only a constant number of rotations. His procedures are simpler than previous ones [5], [10], [14], [15], and almost identical to the ones for balanced binary search trees [13]. The main difference is in the way he searches for a d-tuple q. Each iteration in Gonzalez' algorithm [4] considers a subtree rooted at a node t, and the algorithm keeps the index of a component where q and the smallest d-tuple in the subtree t differ, or the index of a component where q and the largest d-tuple in the subtree t differ. If q is in the tree, it is in the subtree rooted at t. Then either the algorithm finds the d-tuple q at t, or it proceeds to the left or right subtrees of t maintaining the above invariant. Note that the invariant is: "the index of a component where P and the smallest d-tuple differ" rather than "the first index of a ...." The reason is that it is too expensive to find "the first index ..." in this structure with the information that is available. However, if q is in the tree it will be found efficiently, but if it is not in the tree then to avoid reporting that it is in the tree one must perform a simple verification step that takes $O(d)$ time. This is why Gonzalez [4] calls his search strategy principle "assume, verify and conquer" (AVC). The philosophy is to avoid multiple verifications he assumes that some prefixes of strings match. The outcome of our search depends on whether or not these assumptions were valid. This can be determined by performing one simple verification step that takes $O(d)$ time. The elimination of multiple verifications is important because in the worst case there are $\Omega(\log n)$ verifications, and each one could take $\Omega(d)$ time.

In this paper we modify slightly the structure in [4]. Our new structure is in general faster to update because every node has the index of the first component where the node and each of two of its ancestors (if any) differ, rather than the one between the node and the smallest and largest elements in its subtrees as in [4]. When inserting a node or deleting a leaf node, only a couple of entries need to be updated in the structure in this paper, whereas in the structure in [4] one may need to update $\log n$ nodes. However, when deleting a nonleaf node from the tree one has to do a little extra work. Testing for membership is simpler in our new structure. Our new membership procedure mimics the procedure in [9], and follows the update strategy in [4]. We show that INSERT, DELETE, and MEMBERSHIP can be implemented to take $O(d + \log n)$ time and only a constant number of rotations are needed for both INSERT and DELETE. Other operations which can be performed efficiently in our multidimensional balanced binary search trees are: find the (lexicographically) smallest or largest d-tuple ($O(\log n)$ time), print in lexicographic order ($O(dn)$ time), and concatenation ($O(d + \log n)$ time). Finding the (lexicographically) k th smallest or largest d-tuple can also be implemented efficiently ($O(\log n)$ time) by adding more information to each node in the tree. The asymptotic time complexity for the procedures in this paper is exactly the same as the one in [4], but the procedures in this paper are simpler. To distinguish this new type of balanced binary search trees from the classic ones and the ones in [4], we refer to our trees as *multidimensional balanced binary search trees*.

2. Data Structure and Algorithms.

In this section we define our multidimensional balanced binary search trees, and outline efficient procedures for insertion, deletion, and testing for membership. Our representation is based solely on balanced binary search trees, rather than based on TRIES and binary search trees as in previous representations. It is important to note that our trees are like the ones in [13], except for the fact that all the pointers to external nodes in [13] are replaced by *null* pointers in this paper. For example, an internal node with two external nodes as children in [13] is a leaf node in this paper. Readers that are not familiar with balanced binary search trees (or red–black trees) are referred to [13].

We now discuss our new structure and the procedures that operate on it. Each d-tuple is stored at a node in a balanced binary search tree in which the ordering is lexicographic. Each node t in the tree has the following information in addition to the information required to manipulate balanced binary search trees, i.e., the rank bit [13]:

v:	The d-tuple represented by the node. The point is represented by a d-tuple which can be accessed via $x_1(v), x_2(v), \ldots, x_d(v)$.
lchild:	Pointer to the root in the left subtree of t.
rchild:	Pointer to the root in the right subtree of t.
lptr:	Pointer to the node with largest d-tuple in r with value smaller than all the d-tuples in the subtree rooted at t, or null if no such d-tuple exists.
hptr:	Pointer to the node with smallest d-tuple in r with value larger than all the d-tuples in the subtree rooted at t, or null if no such d-tuple exists.
lj:	Index of first component where the d-tuple at t and the node pointed at by *lptr* differ, or one if *lptr* $=$ *null*.
hj:	Index of first component where the d-tuple at t and the node pointed at by *hptr* differ, or one if *hptr* $=$ *null*.

Our procedures perform two types of operations: operations required to manipulate balanced binary search trees (which we refer to as *standard* operations), and operations to manipulate and maintain our structure (which we refer to as *new* operations). The standard operations are well known [11], [13]; therefore, we only explain them briefly. The MEMBERSHIP procedure is similar to the one for searching in a binary search tree. The input to the search procedure is the d-tuple q. We start at the root and visit a set of tree nodes until we either reach a *null* pointer which indicates that q is not in the tree or we find a node with d-tuple q. In the former case we have identified the location where q could be inserted in order to maintain a binary search tree. For the INSERT operation, we first perform procedure MEMBERSHIP. If the d-tuple is in the tree the procedure terminates, since we do not need to insert the d-tuple. Otherwise, procedure MEMBERSHIP will give us the location where the d-tuple should be inserted. The d-tuple is inserted, some information stored at some nodes in the path from the root to the node inserted is updated and, if needed, we perform a constant number of rotations. The DELETE operation is a little more complex. First we need to perform operations similar to the ones in procedure MEMBERSHIP to find out whether the d-tuple is in the tree. If it is not in the tree the procedure terminates, otherwise we have a pointer to the node to be deleted. The following technique discussed in [6] (which is similar to the one used for AVL-trees) is used to reduce the deletion of an arbitrary node to the deletion of a leaf node. Assume that the node to be deleted has a nonempty right subtree, since the case when it has a nonempty left subtree and empty right subtree can be solved similarly. Find the d-tuple in the tree with the next larger value (node B in Figure 5). If such a node is a leaf, then the problem is reduced to deleting that leaf node by interchanging the values in these two nodes, otherwise three nodes have to interchange their values and again the problem is reduced to deleting a leaf node. Deletion of a leaf node is performed by deleting it, updating some information stored in the path from the position where the node was deleted to the root of the tree, and, if needed, performing a constant number of rotations.

To show that MEMBERSHIP, INSERT, and DELETE can be implemented in the proposed time bounds, we need to establish that the following (new) operations can be performed in $O(d + \log n)$ time:

A. Given q determine whether or not it is stored in the tree.
B. Update the structure after adding a node (just before rotation(s), if any).
C. Update the structure after performing a rotation.
D. Update the structure after deleting a leaf node (just before rotation(s), if any).
E. Transform the deletion problem to deletion of a leaf node.

First we discuss procedure MEMBERSHIP(q, r) to test whether or not the d-tuple q given by $(x_1(q), x_2(q), \ldots, x_d(q))$ is in the multidimensional binary search tree (or subtree) rooted at r. This procedure implements (A) and its operation can be summarized as follows. Let t point to any node in the multidimensional balanced binary search tree rooted at r. We define *prev*(t) to be the d-tuple in r with the largest value whose value is smaller than all the d-tuples stored in the subtree pointed at by t, unless no such tuple exists in which case its value is $(-\infty, -\infty, \ldots, -\infty)$, and define *next*$(t)$ to be the d-tuple in r with the smallest value whose value is larger than all the d-tuples stored in the subtree pointed at by t, unless no such tuple exists in which case its value is $(+\infty, +\infty, \ldots, +\infty)$. At each iteration we maintain the following invariants. Variable t points to the root of a subtree, initially pointing to the root of the tree. The variable d_{low} is the index of the first component where q and *prev*(t) differ, and variable d_{high} is the index of the first component where q and *next*(t) differ. The d-tuple being search for, q, is such that its value is (lexicographically) greater than *prev*(t) and (lexicographically) smaller than *next*(t)

The algorithm computes (indirectly) j' as the index of the first component where $t \rightarrow v$ and q differ. Consider the case when $d_{\text{low}} \geq d_{\text{high}}$ (the other case is similar). There are two cases: (1) if $d_{\text{low}} \neq t \rightarrow lj$, then j' is just $\min\{d_{\text{low}}, t \rightarrow lj\}$, (2) if $d_{\text{low}} = t \rightarrow lj$, then the j' is set to the index of the first component starting at position d_{low} where q and $t \rightarrow v$ differ. We will establish (Lemma 2.1) that j' ends up with the index of the first component where $t \rightarrow v$ and q differ. If it is the case that j' is equal to $d + 1$, then q is the d-tuple stored in t and we return the value of **true**. Otherwise by comparing the j'th element of q and t we decide whether to search in the left or right subtrees of t. In either case d_{high} or d_{low} is set appropriately so that the invariant holds at the next iteration. The actual code is given below:

```
**Procedure** MEMBERSHIP$(q, r)$;
/* Is $(x_1(q), x_2(q), \ldots, x_d(q))$ in the multidimensional balanced binary search
tree at $r$ */
$d_{\text{low}} = d_{\text{high}} = 1$;
$t \leftarrow r$;
**while** $t \neq null$ **do**
    **case**
        :$d_{\text{low}} \geq d_{\text{high}}$:
            **if** $d_{\text{low}} = t \rightarrow lj$ **then** $j' = diff(q, t \rightarrow v, d_{\text{low}})$ **else** $j' \leftarrow min\{t \rightarrow
            lj, d_{\text{low}}\}$;
        :$d_{\text{low}} < d_{\text{high}}$:
            **if** $d_{\text{high}} = t \rightarrow hj$ **then** $j' = diff(q, t \rightarrow v, d_{\text{high}})$ **else** $j' \leftarrow
            min\{t \rightarrow hj, d_{\text{high}}\}$;
    **endcase**
```

```
        if $j' = d + 1$ then return(true);
        case
           :$x_{j'}(q) < x_{j'}(t \to v)$:
                 $d_{\text{high}} \leftarrow j'$;
                 $t \leftarrow t \to lchild$;
           :$x_{j'}(q) > x_{j'}(t \to v)$:
              $d_{\text{low}} \leftarrow j'$;
              $t \leftarrow t \to rchild$;
        endcase
     endwhile
     return(false);
     end of Procedure MEMBERSHIP
```

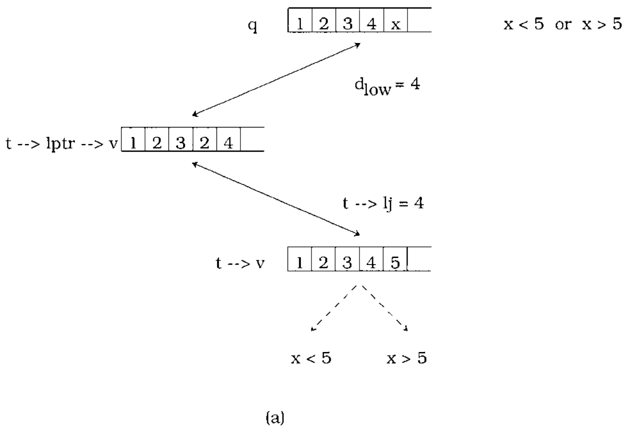
The following lemma establishes correctness for procedure MEMBERSHIP(q, r).

LEMMA 2.1. *Given a d-tuple q procedure* MEMBERSHIP(q, r) *determines whether or not q is in the multidimensional balanced binary search tree rooted at r in $O(d + \log n)$ time.*

PROOF. We claim that at each iteration we maintain the following invariants. Variable t points to the root of a subtree of r. The variable d_{low} is the index of the first component where q and *prev*(t) differ, and variable d_{high} is the index of the first component where q and *next*(t) differ. The d-tuple q is such that its value is (lexicographically) greater than *prev*(t) and (lexicographically) smaller than *next*(t). Initially t is set to r, and $d_{\text{low}} = d_{\text{high}} = 1$. Since *prev*$(t)$ is $(-\infty, -\infty, \ldots, -\infty,)$ and *next*(t) is $(\infty, \infty, \ldots, \infty,)$, and all the entries in the d-tuples are different than ∞ and $-\infty$, it then follows that the invariant holds just before the **while** loop is about to be executed for the first time.

We now show that if the invariant holds just before the **while** loop is about to be executed, then either the procedure terminates with the correct answer or the invariant holds at the beginning of the next iteration. If t is *null*, then clearly q is not in t and the procedure returns **false**. Now, we consider the case when t is not *null*. We claim that the algorithm computes (indirectly) j' as the index of the first component where $t \to v$ and q differ. We only prove this for the case when $d_{\text{low}} \geq d_{\text{high}}$, since the proof for the other case ($d_{\text{low}} < d_{\text{high}}$) is similar. If $d_{\text{low}} = t \to lj$ (Figure 3(a)), then the index of the first component where $t \to v$ and q differ is just the index of the first component starting at position d_{low} where $t \to v$ and q differ. Therefore, j' is computed correctly for this case. When $d_{\text{low}} > t \to lj$ (Figure 3(b)) then the index of the first component where $t \to v$ and q differ is just $t \to lj$ and the next value for t is $t \to lchild$. So, *next* for the new value of t is the old value of t and d_{high} will be set to j'. On the other hand, when $d_{\text{low}} < t \to lj$ (Figure 3(c)) then the index of the first component where $t \to v$ and q differ is just d_{low} and the next value for t is $t \to rchild$. So, *prev* for the new value of t is the old value of t and d_{low} will be set to j'. In either case the procedure computes j' correctly.

When $j' = d + 1$, we know that $t \to v = q$ and the procedure returns the value **true** which is the correct answer because, as we just established, j' is the index of the first component where $t \to v$ and q differ. So assume that $j' < d + 1$. The appropriate



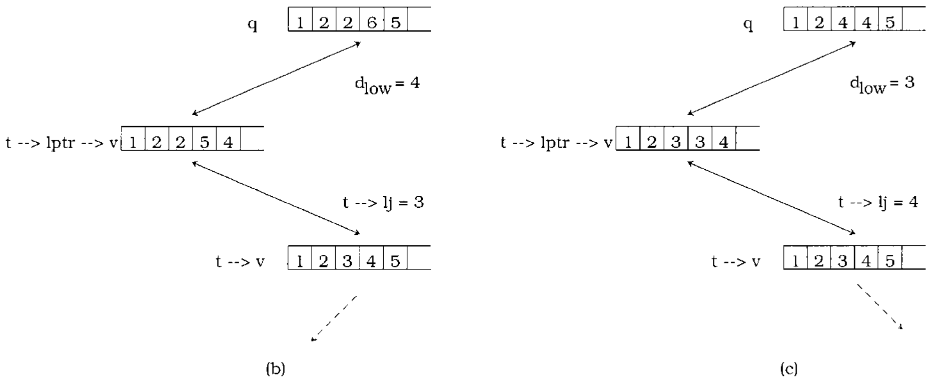


Fig. 3. Three cases depending on the relative values of d_{low} and $t \rightarrow lj$.

subtree where we should search for q is determined by the relative values of $x_{j'}(q)$ and $x_{j'}(t \rightarrow v)$. If $x_{j'}(q) < x_{j'}(t \rightarrow v)$, then we should search in the left subtree of t and d_{high} should be set to j' because *next* for the new value of t is the previous node pointed to by t. On the other hand if $x_{j'}(q) > x_{j'}(t \rightarrow v)$, then we should search in the right subtree of t and d_{low} should be set to j' because *prev* for the new value of t is the previous node pointed to by t. Clearly, the invariant holds just as we are about to execute the **while** statement again.

The number of operations at each level is not bounded by a constant; however, they are bounded by 1 plus the difference between the new and old value of $max\{d_{\text{low}}, d_{\text{high}}\}$. Since $max\{d_{\text{low}}, d_{\text{high}}\}$ does not decrease and it is at most $d + 1$ at the end of each operation, it follows that the total number of operations performed is of order d plus the height of the tree (which is $O(\log n)$). Thus the time complexity of procedure MEMBERSHIP(p, r) is $O(d + \log n)$. □

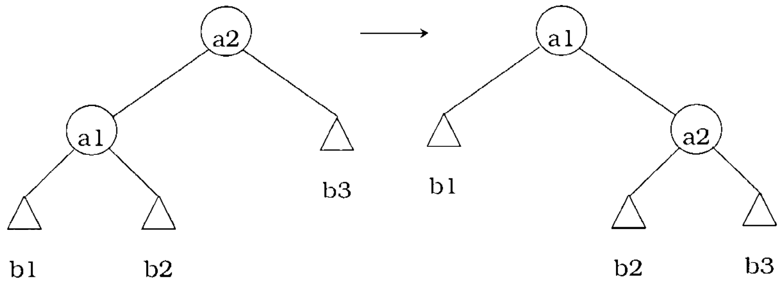


Fig. 4. Rotation.

We have identified an algorithm that implements (A) within the proposed time complexity bound. We now consider (B). If the tree is empty just before the insert operation, then the update of a single node is trivial. Suppose now that we add a d-tuple to a nonempty tree. Let q point to the node added. Clearly, the node that we add is a leaf node, therefore we must compute its values *lj*, *hj*, *lptr*, and *hptr*. The pointers *lptr* and *hptr* can be obtained from the parent of the new node q, and the *diff* values *lj*, and *hj* can be computed directly in $O(d)$ time.

LEMMA 2.2. *After inserting a node q in a multidimensional balanced binary search tree and just before rotation the structure can be updated as mentioned above in* $O(d + \log n)$ *time.*

We have identified an algorithm that implements (B) within the proposed time complexity. It is simple to see that a similar procedure can be used to implement (D). We now consider how to implement (C), i.e., rotations. This is the simplest part. A simple rotation is shown in Figure 4. We only consider single rotations, since the compound rotations in [13] can be obtained by applying several single rotations. The rotation is performed by moving the nodes rather than just the values. This reduces the number of updates that need to be performed. Clearly, the only nodes whose information needs to be updated are a_1, a_2, and the parent of a_2. Since there is a fixed number (3) of them the operations can be implemented to take $O(d)$ time. This result is summarized in Lemma 2.3, which we state without a proof.

LEMMA 2.3. *After a rotation in a multidimensional balanced binary search tree the structure can be updated as mentioned above in* $O(d)$ *time.*

We now consider operation (E) which is a little more elaborate to implement. Assume without loss of generality that the node to be deleted has a nonempty right subtree, since the case when it has a nonempty left subtree and empty right subtree can be solved similarly. It is well known [6] that the problem of deleting an arbitrary node from a balanced binary search tree can be reduced to deleting a leaf node by applying the transformation in Figure 5 (the original operation is to delete the node labeled A and it is transformed to deleting leaf node X). We now show how to update the resulting

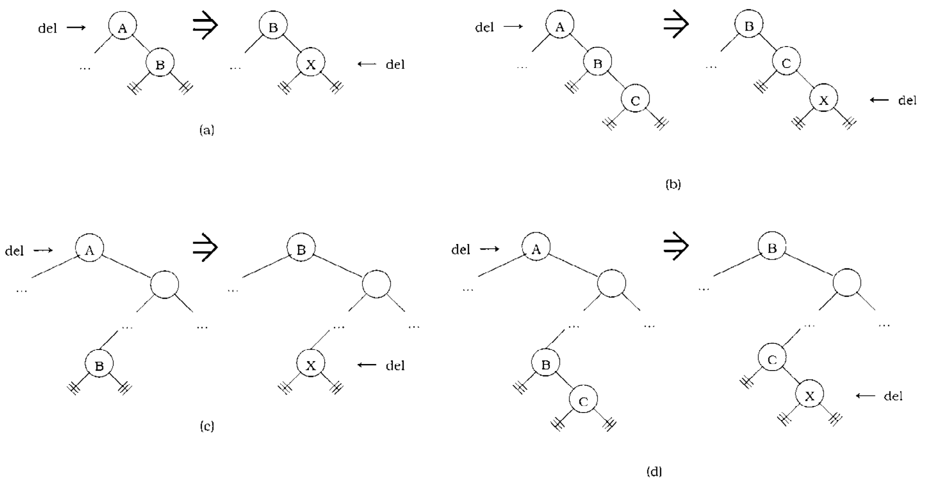


Fig. 5. Transforming deletion of an arbitrary node to deletion of a leaf node.

structure in $O(d + \log n)$ time. Since the node with the final value X will be deleted, we do not need to update it. For the new root (the one labeled B) we need to update the *lj* and *hj* values. Since we can use directly the old *lptr* and *hptr* values, the update can be done in $O(d)$ time. The *lj* (*hj*) value of all the nodes (if any) in the path that starts at the right (left) child of the new root (node labeled B) and continues through the left child (right child) pointers until the null pointer is reached needs to be updated. There are at most $O(\log n)$ such nodes. If we update them one by one without reusing partial results, the time complexity will not be the proposed one. However, the values stored at each of these nodes are decreasing (increasing) when traversing the path top down. Therefore, the *lj* (*hj*) values appear in increasing order. The correct values can be easily computed in $O(d + \log n)$ time by reusing previously computed *lj* (*hj*) values while traversing the path top down. Lemma 2.4, whose proof is omitted, summarizes our observations.

LEMMA 2.4. *Transforming the deletion problem to deleting a leaf node can be performed as mentioned above in $O(d + \log n)$ time.*

Our main result which is based on the above discussions and the lemmas is given below.

THEOREM 2.1. *Any on-line sequence of operations of the form* INSERT(p), DELETE(p), *and* MEMBERSHIP(p), *where p is any d-tuple, can be carried out by the above procedures on a multidimensional balanced binary search tree in $O(d + \log n)$ time, where n is the current number of points, and each* INSERT *and* DELETE *operation requires no more than a constant number of rotations.*

PROOF. By the above discussion, the lemmas, and the fact that only $O(1)$ rotations are needed for each INSERT and DELETE operation on balanced binary search trees [13]. □

3. Discussion. It is interesting to note that our technique cannot be adapted to AVL-trees, weight balanced trees, or B-trees of fixed order, because the number of rotations after a DELETE operation might be large $\Omega(\log n)$. Since each rotation could take $\Omega(d)$ time, the proposed time complexity bounds would not hold. The main reason why they work on balanced binary search trees is that only $O(1)$ rotations are needed after every INSERT and DELETE operation.

With respect to other operations, it is simple to see that the smallest or largest d-tuple can be easily found in $O(\log n)$ time, and that all the d-tuples can be printed in increasing or decreasing order in $O(dn)$ time. An $O(d + \log n)$ time algorithm to CONCATENATE two sets represented by our structure can be easily obtained by using standard procedures. However, the SPLIT operation cannot be implemented within this time complexity bound. The main reason is that there could be $\Omega(\log n)$ rotations. The k th smallest or largest d-tuple can be found in $O(\log n)$ time after adding to each node in the tree the number of nodes in its left subtree.

On average the TRIE plus binary search tree approach requires less space to represent the d-tuples than our structure. However, our procedures are simple, take only $O(d + \log n)$ time, and only a constant number of rotations are required after each INSERT and DELETE operation.

Acknowledgment. The author wishes to thank an anonymous referee for pointing out reference [9] to us.

References

[1] J. L. Bentley and J. B. Saxe, Algorithms on Vector Sets, *SIGACT News* (Fall 1979), pp. 36–39.
[2] H. A. Clampett, Randomized Binary Searching with the Tree Structures, *Comm. ACM*, 7(3) (1964), pp. 163–165.
[3] T. Gonzalez, Covering a Set of Points with Fixed Size Hypersquares and Related Problems, *Inform. Process. Lett.*, 40 (1991), 181–188.
[4] T. Gonzalez, The On-Line D-Dimensional Dictionary Problem, *Proceedings of the 3rd Symposium on Discrete Algorithms*, January 1992, pp. 376–385.
[5] R. H. Gueting and H. P. Kriegel, Multidimensional B-Tree: An Efficient Dynamic File Structure for Exact Match Queries, *Proceedings of the 10th GI Annual Conference*, Springer-Verlag, Berlin, 1980, pp. 375–388.
[6] L. J. Guibas and R. Sedgewick, A Dichromatic Framework for Balanced Trees, *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978, pp. 8–21.
[7] D. S. Hirschberg, On the Complexity of Searching a Set of Vectors, *SIAM J. Comput.* 9(1) (1980), 126–129.
[8] S. R. Kosaraju, On a Multidimensional Search Problem, *Proceedings of the ACM Symposium on the Theory of Computing*, 1979, pp. 67–73.
[9] U. Manber and G. Myers, Suffix Arrays: A New Method for On-Line String Searches, *SIAM J. Comput.*, 22(5) (1993), 935–948. Also in *Proceedings of the First ACM–SIAM Symposium on Discrete Algorithms*, Jan. 1990, pp. 319–327.
[10] K. Mehlhorn, Dynamic Binary Search, *SIAM J. Comput.*, 8(2) (1979), 175–198.
[11] H. J. Olivie, A New Class of Balanced Search Trees: Half-Balanced Binary Search Trees, Ph.D. Thesis, University of Antwerp, U.I.A., Wilrijk, Belgium, 1980.
[12] E. H. Sussenguth, Use of Tree Structures for Processing Files, *Comm. ACM*, 6(5) (1963), 272–279.
[13] R. E. Tarjan, Updating a Balanced Search Tree in $O(1)$ Rotations, *Inform. Process. Lett.*, 16 (1983), 253–257.
[14] V. Vaishnavi, Multidimensional Height-Balanced Trees, *IEEE Trans. Comput.*, 33(4) (1984), 334–343.
[15] V. Vaishnavi, Multidimensional Balanced Binary Trees, *IEEE Trans. Comput.*, 38(7) (1989), 968–985.