The on-line d-dimensional dictionary problem

T.F. Gonzalez

RUU-CS-90-31
October 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

The on-line d-dimensional dictionary problem

T.F. Gonzalez

Technical Report RUU-CS-90-31
October 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

THE ON-LINE d-DIMENSIONAL DICTIONARY PROBLEM

by

Teofilo F. Gonzalez‡
Department of Computer Science
Utrecht University
the Netherlands

ABSTRACT: We study the on-line d-dimensional dictionary problem. This problem consists of executing on-line any sequence of operations of the form: INSERT(p), DELETE(p) and MEMBERSHIP(p), where p is any point in d-space. To represent the set of points we use a new generalization of balanced binary search trees, which we call d-dimensional balanced binary search trees. We show that any of the above operations can be implemented to take $O(d + \log n)$ time, where n is the current number of points in the set, and each INSERT and DELETE operation requires no more than a constant number of rotations. Our procedures are almost identical to the ones for balanced binary search trees. The main difference is in the way we search for an element. Our search strategy is based on the principle "assume, verify and conquer" (AVC). We apply this principle as follows. To avoid multiple verifications we shall assume that some prefixes of strings match. At the end of our search we must determine whether or not these assumptions were valid. This can be done 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 each taking $O(d)$ time.

KEYWORDS: d-dimensional dictionaries, balanced binary search trees, efficient algorithms, assume-verify-and-conquer.

‡ On Sabbatical leave from the Department of Computer Science, University of California, Santa Barbara.

September 29, 1990

I. INTRODUCTION.

The on-line 1-dimensional dictionary, or simply the dictionary, problem consists of executing any sequence of instructions of the form INSERT(p), DELETE(p) and MEMBERSHIP(p), where each p is a real number. It is well known that any of these three instructions can be implemented to take O($log\ n$) time, where n is the current number of elements in the set. The set can be 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 the B-trees which are m-way binary search trees. The balanced binary trees are the only ones that require only O(1) rotations for both the insert and the delete operation ([O], [T]).

In this paper we consider the dictionary problem when the set of points is defined in d-space. In this case we refer to the problem as the on-line d-dimensional dictionary problem. The problem consists of executing on-line any sequence of the following operations: INSERT(p), DELETE(p) and MEMBERSHIP(p), where p is any point in d-space (i.e., the universe is the set of points in d-space). The set of points is denoted by P and point $p \in P$ has coordinate values given by ($x_1(p)$, $x_2(p)$, ..., $x_d(p)$). We examine several data structures to represent a set of points P and develop algorithms to perform any sequence of on-line d-dimensional dictionary operations. We show that any of the three operations can be performed in O($d + log\ n$) time, where n is the current number of points in the set and d is the number of dimensions. Furthermore, only O(1) rotations are required for each INSERT and DELETE operation.

As noted by Mehlhorn [M], "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, TRIES have been combined with different balanced tree schemes to represent multikey sets (i.e., points in d-space). Let us now elaborate on this method. 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 length m, 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 $\{0, 1, ..., m-1\}$ (see structure in figure 1, where $\Sigma = \{0, 1, 2, 3\}$).

(2) (Sussenguth [S]) Each internal node is represented by a linear list (see structure in figure 2).

(3) (Clampett [C]) Each internal node in the TRIE is represented by a binary search tree (see structure in figure 3).

In general, implementation (1) uses the largest amount of space and implementation (2) uses the least amount of space. Also, implementation (1) is the fastest and (2) is the slowest. The performance of (3) is between that of (1) and (2).

For d-dimensional dictionaries defined over the set of integers $[0,m)$, the TRIE method is applied as follows. Under representation (1) each TRIE node is an m-element vector. The TRIE method treats a point in d-space as a string with d elements defined over the alphabet $\Sigma = \{0, 1, ..., m-1\}$ (see figure 1). For large m or when instead of integers we have real numbers this method 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 2), or a binary search tree replacing the list (see figure 3). Bentley and Saxe [BSa] used this technique together with the following balancing scheme. The root of each subtree is a node such that its "middle" subtree contains the terminal node of a median element in the set represented by the subtree. Such a structure is very useful for static search problems like sorting or restricted searching ([K] and [H]). Since fully balanced subtrees are very rigid structures which are not appropriate for dynamic updates, they should be replaced by more flexible structures in dynamic environments. For example, the balancing of these trees is performed by using techniques related to fixed order B-trees [GK], weight balanced trees [M], AVL trees [V1], and balanced binary search trees [V2]. For these representations each of the three operations in a d-dimensional dictionary can be implemented to take $O(d + \log n)$ time. However, the number of rotations after each INSERT and DELETE operation is not bounded by a constant.

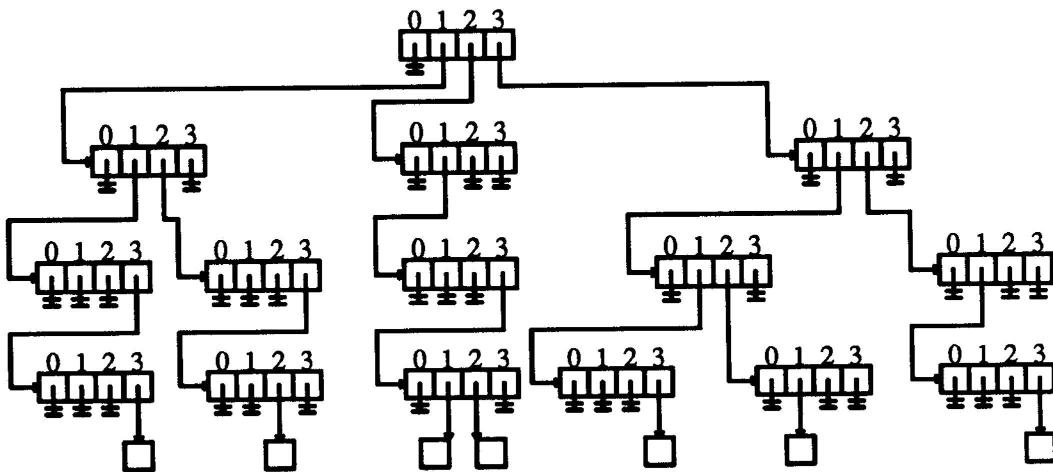


Figure 1: TRIE representation.

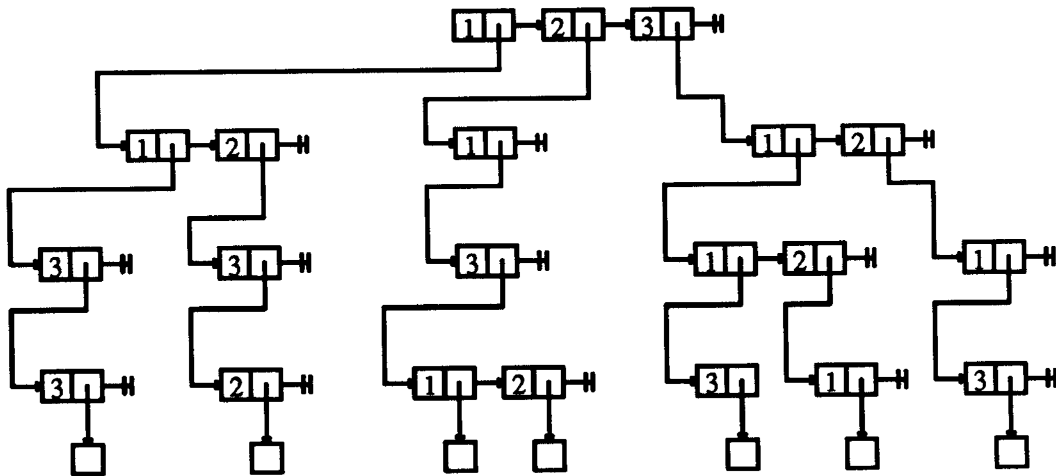


Figure 2: Linked list representation for TRIE nodes.

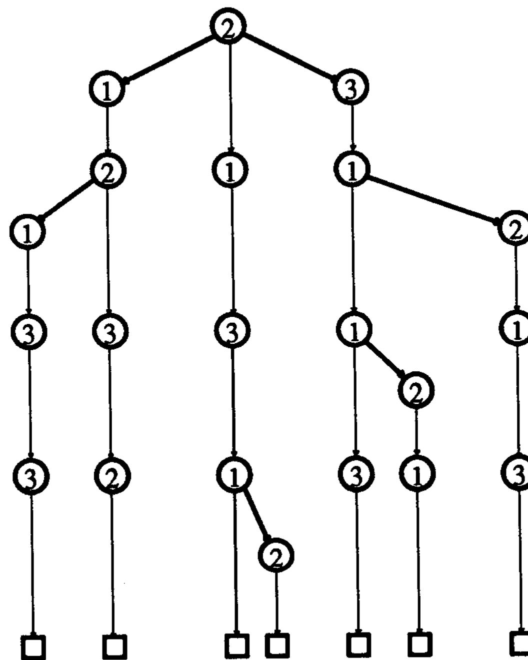


Figure 3: Binary search representation for TRIE nodes.
Solid arcs are binary search tree pointers and other arcs are TRIE pointers.

In this paper we investigate a representation which is based solely on binary search trees, rather than on a combination of TRIES and binary search trees. Let us briefly explain the naive approach, which does not achieve the proposed time complexity bound. In this method each point p_i is stored at a node of a binary search tree. The ordering in the binary search tree is lexicographic. Comparing two points can be accomplished by performing at most d operations. It is simple to show that if there are n elements currently being represented by an AVL, weight balanced, or balanced binary search trees, any of the three

dictionary operations can be carried out in at most $O(d \log n)$ time. Furthermore, there are problem instances such that the straight forward implementation of procedures INSERT, DELETE and MEMBERSHIP requires $\Omega(d \log n)$ time.

To achieve the proposed time complexity bound we represent the set of points P in a balanced binary search tree in which additional information has been stored at each node. To distinguish this new type of balanced binary search trees from the classic ones we shall refer to the former as *d-dimensional balanced binary search trees*. For this representation we present procedures for INSERT, DELETE and MEMBERSHIP which take $O(d + \log n)$ time and require only a constant number of rotations when executing an INSERT and DELETE operation.

II. THE ALGORITHMS.

In this section we outline our algorithms and the structure to implement d-dimensional directories. The representation is based solely on balanced binary search trees, rather than based on TRIES and binary search trees. To achieve the proposed time complexity bound we represent the set of points P in a d-dimensional balanced binary search tree in which additional information has been stored at each node. For this representation we present procedures for INSERT, DELETE and MEMBERSHIP which take $O(d + \log n)$ time. It is important to note that our trees are identical to the ones in [T], except for the fact that all the pointers to external nodes in [T] are replaced by null pointers in this paper. For example, an internal node with two external nodes as children in [T] is a leaf node in this paper. Before explaining our procedures, we define some terms as well as the additional information to be stored at each node.

Let p and q be two points (in d-space). For $1 \le i \le d$, we define $diff(p,q,i)$ as the 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 $d + 1$. Each node in the tree has the following information in addition to the information required to manipulate balanced binary search trees.

$t \rightarrow v$: point	The element represented by the node. The point is represented by a d-tuple which can be accessed via $x_1(t \rightarrow v), x_2(t \rightarrow v), \ldots, x_d(t \rightarrow v)$.
$t \rightarrow lchild$: pointer	Pointer to the root in the left subtree of t.
$t \rightarrow rchild$: pointer	Pointer to the root in the right subtree of t.
$t \rightarrow lptr$: pointer	Pointer to the node with smallest value of the subtree rooted t.
$t \rightarrow hptr$: pointer	Pointer to the node with largest value of the subtree rooted t.
$t \rightarrow jl$: integer	$diff(t \rightarrow v, t \rightarrow lptr \rightarrow v, 1)$.
$t \rightarrow jh$: integer	$diff(t \rightarrow v, t \rightarrow hptr \rightarrow v, 1)$.

Our procedures perform two types of operations: operations required to manipulate balanced binary search trees (which we refer to as *standard* operations) and operations for manipulating and maintaining our structure (which we refer to as *new* operations). The standard operations are well known, therefore we shall only explain them briefly. The MEMBERSHIP procedure is identical to the one for searching in a binary search tree. The input to the search procedure is a value p. We start at the root and visit a set of tree nodes until we either reach a pointer with value null which indicates that p is not in the tree, or we find the element. In the former case we have identified the location where p could be inserted in order to maintain a binary search tree, and in the later case we visit only those nodes which are predecessors of the node with value p. For the INSERT operation, we first perform procedure MEMBERSHIP. If the element is in the tree the procedure terminates, since we do not need to insert the element. Otherwise, procedure MEMBERSHIP will give us the location where the element should be inserted. The element is inserted, and if needed we perform a constant number of rotations. Also, the information stored at some nodes in the path from the root to the node inserted need to be updated. The delete operation is a little bit more complex. First we need to perform operations similar to the ones in procedure MEMBERSHIP to find out whether the element 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. We use the following technique discussed in [GS] (which is similar to the one used for AVL trees) to reduce the deletion of an arbitrary node to the deletion of a leaf node. If the node to be deleted is not a leaf node, then we either find the next element or the previous element in the tree which has at least one null pointer. 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, performing a constant number of rotations and then updating some information stored in the path from the root to the position where the node was deleted from the tree.

To show that all of the operations can be implemented in the proposed time bounds, we need to show that the following (new) operations can be performed $O(d + \log n)$ time.

(A) Given an element p determine whether or not it is stored in the tree and if it is in the tree then return a pointer to it.

(B) Given an element p which is not stored in the tree, find the place where it should be inserted in order to maintain a binary search tree.

(C) Update the structure after adding a node (just before rotation).

(D) Update the structure after performing a rotation.

(E) Update the structure after deleting a node (just before rotation).

(F) Transform the deletion problem to deleting a leaf node.

First we discuss procedure MEMBERSHIP(p,t) to test whether or not point p given by ($x_1(p)$, $x_2(p)$, ..., $x_d(p)$) is in the d-dimensional binary search tree (or subtree) rooted at t. This procedure implements (A) above, and as we shall see later on it can be easily modified to implement (B). At each iteration, t, points to the root of a subtree, and if $comp = low$ then j has a value between 1 and d such that $x_j(p) \neq x_j(t \rightarrow lptr \rightarrow v)$ or $j = d + 1$; otherwise $comp = high$ and j is an integer in $[1,d]$ such that $x_j(p) \neq x_j(t \rightarrow hptr \rightarrow v)$ or $j = d + 1$. As we shall see, j also satisfies some additional properties. We claim that at each step in our algorithm, one of the following three statements hold.

(i) If p is in the tree then $t \rightarrow lptr \rightarrow v \leq p \leq t \rightarrow hptr \rightarrow v$ and either $j = diff(p, t \rightarrow lptr \rightarrow v, 1)$ when $comp = low$, or $j = diff(p, t \rightarrow hptr \rightarrow v, 1)$ when $comp = high$.

(ii) If p is not in the tree and $t \rightarrow lptr \rightarrow v \leq p \leq t \rightarrow hptr \rightarrow v$, then either $j = diff(p, t \rightarrow lptr \rightarrow v, 1)$ when $comp = low$, or $j = diff(p, t \rightarrow hptr \rightarrow v, 1)$ when $comp = high$.

(iii) If p is not in the tree, then either $p < t \rightarrow lptr \rightarrow v$ or $p > t \rightarrow hptr \rightarrow v$.

Let us outline our strategy when searching for p at node t and $comp = low$. Assume that p is in the tree. Later on we explain how to modify our strategy to deal with the case when p is not in the tree. We claim that at each step in the algorithm (i) holds true. Initially t points to the root of the tree and j is set to $diff(p, t \rightarrow lptr \rightarrow v, 1)$. Therefore, (i) holds initially. We now show that if (i) holds during the kth iteration and our algorithm performs certain operations (that we specify), then (i) will hold at the $k+1$st iteration. If $j = d + 1$, then we know that p is equal to $t \rightarrow lptr \rightarrow v$ and we return; otherwise, $j < d + 1$. There are three cases.

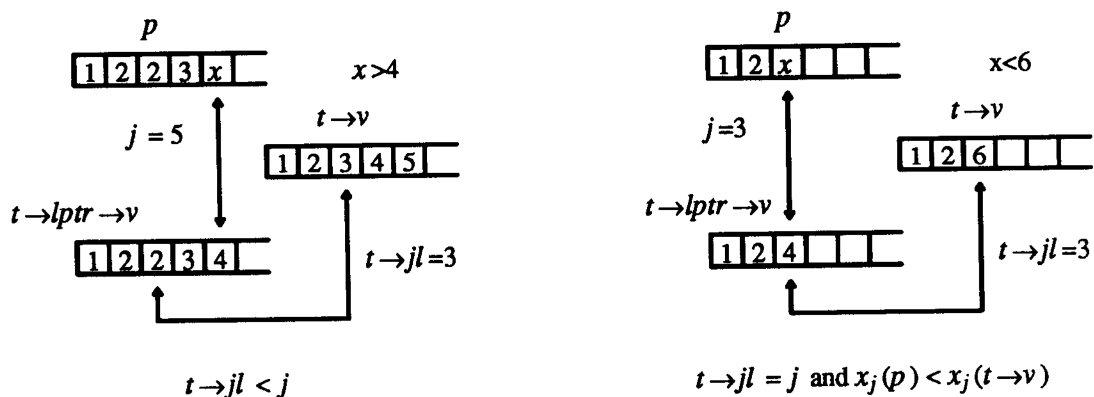


Figure 4: Case 1.

case 1: $t \rightarrow jl < j$ or ($t \rightarrow jl = j$ and $x_j(p) < x_j(t \rightarrow v)$) (see figure 4).
By assumption $t \rightarrow lptr \rightarrow v \leq p \leq t \rightarrow hptr \rightarrow v$ and $j = diff(p, t \rightarrow lptr \rightarrow v, 1)$. From the conditions of the case we know that $p < t \rightarrow v$, so p is not $t \rightarrow v$ nor it is in the right subtree of t. Since p is in the tree, it must be that $t \rightarrow lchild \neq null$. By definition $t \rightarrow lptr = t \rightarrow lchild \rightarrow lptr$ and since p is in the tree $p \leq t \rightarrow lchild \rightarrow hptr \rightarrow v$. Therefore, after setting t to $t \rightarrow lchild$, we know that (i) holds.

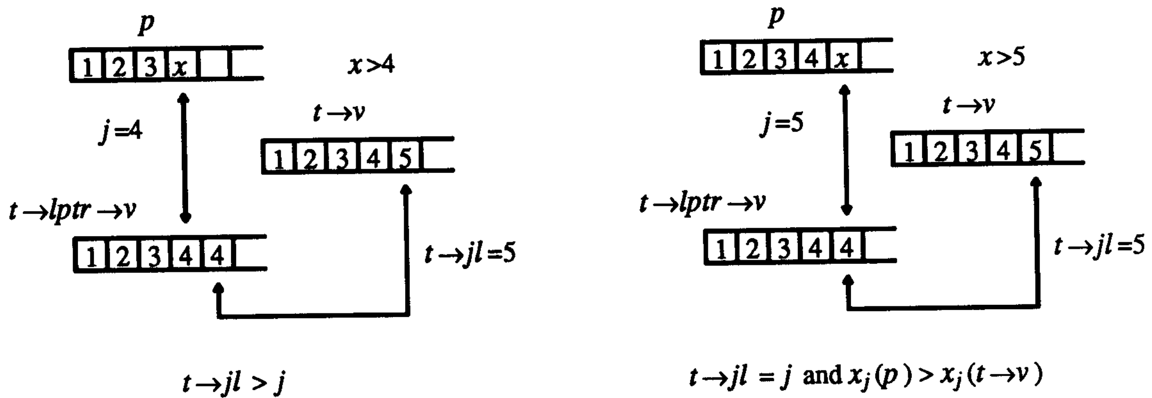


Figure 5: Case 2.

case 2: $t \to jl > j$ **or** $(t \to jl = j$ **and** $x_j(p) > x_j(t \to v))$ (see figure 5).
By assumption $t \to lptr \to v \le p \le t \to hptr \to v$ and $j = diff(p, t \to lptr \to v, 1)$. From the conditions of the case we know that $p > t \to v$, so p is not $t \to v$ nor it is in the left subtree of t. Since p is in the tree, it must be that $t \to rchild \neq null$. By definition $t \to hptr$ is equal to $t \to rchild \to hptr$ and since p is in the tree $p \ge t \to rchild \to lptr \to v$. Let $j' \leftarrow diff(p, t \to rchild \to lptr \to v, j)$. Since $j \le diff(p, t \to v, 1)$ and p is in the right subtree, it must be the case that $j \le diff(p, t \to rchild \to lptr \to v, 1)$. So, j' is equal to $diff(p, t \to rchild \to lptr \to v, 1)$. Therefore, after setting t to $t \to rchild$ and j to j', we know (i) holds.

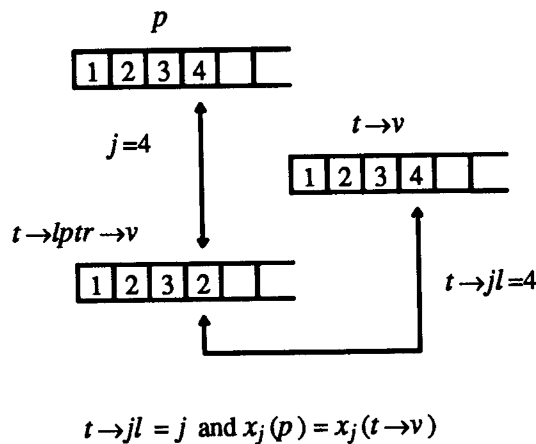


Figure 6: Case 3.

case 3: $t \to jl = j$ **and** $x_j(p) = x_j(t \to v)$ (see figure 6).
By assumption $t \to lptr \to v \le p \le t \to hptr \to v$ and $j = diff(p, t \to lptr \to v, 1)$. Let $j' \leftarrow diff(p, t \to v, j)$. Since $j = t \to jl$, we know that $j \le diff(p, t \to v, 1)$. Therefore, $j' = diff(p, t \to v, 1)$. If $j' = d + 1$, then $p = t \to v$. When $j' \le d$, there are two separate subcases.

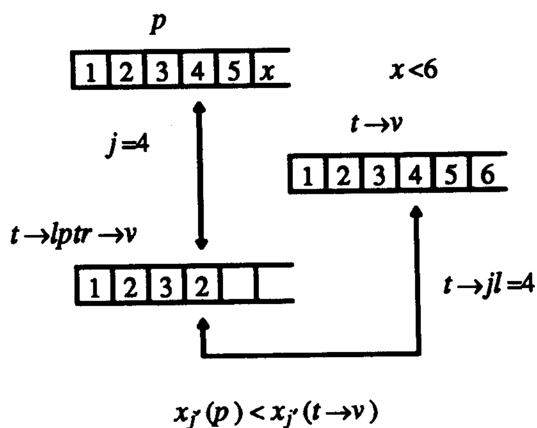
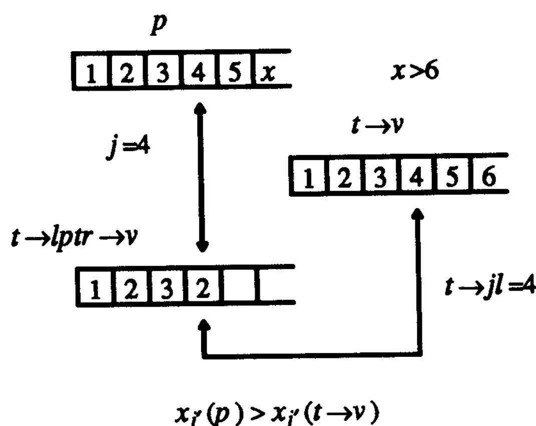


Figure 7: Subcase 3.1.

subcase 3.1: $x_{j'}(p) < x_{j'}(t \rightarrow v)$ (see figure 7).

Clearly, $p < t \rightarrow v$, so p is not $t \rightarrow v$ nor it is in the right subtree of t. Since p is in the tree it must be in the left subtree, therefore $t \rightarrow lchild \neq null$. By definition $t \rightarrow lptr$ is equal to $t \rightarrow lchild \rightarrow lptr$, and since p is in the tree it must be that $p \leq t \rightarrow lchild \rightarrow hptr \rightarrow v$. Let $j'' \leftarrow diff(p, t \rightarrow lchild \rightarrow hptr \rightarrow v, j')$. Since $j' = diff(p, t \rightarrow v, 1)$ and p is in the left subtree, it must be that $j' \leq diff(p, t \rightarrow lchild \rightarrow hptr \rightarrow v, 1)$. Therefore, j'' is equal to $diff(p, t \rightarrow lchild \rightarrow hptr \rightarrow v, 1)$. If $j'' = d + 1$, then p is equal to $t \rightarrow lchild \rightarrow hptr \rightarrow v$. Otherwise, since p is in the tree we know that $x_{j''}(p) < x_{j''}(t \rightarrow lchild \rightarrow hptr \rightarrow v)$. Therefore, after setting t to $t \rightarrow lchild$, j to j'' and *comp* to *high*, we know that (i) holds.



$x_{j'}(p) > x_{j'}(t \rightarrow v)$

Figure 8: Subcase 3.2.

subcase 3.2: $x_{j'}(p) > x_{j'}(t \rightarrow v)$ (see figure 8).
Clearly, $p > t \rightarrow v$, so p is not $t \rightarrow v$ nor it is in the left subtree of t. Since p is in the tree it must be in the right subtree, therefore $t \rightarrow rchild \neq null$. By definition $t \rightarrow hptr$ is equal to $t \rightarrow rchild \rightarrow hptr$, and since p is in the tree it must be that $p \geq t \rightarrow rchild \rightarrow lptr \rightarrow v$. Let $j'' \leftarrow diff(p, t \rightarrow rchild \rightarrow lptr \rightarrow v, j')$. Since j' $= diff(p, t \rightarrow v, 1)$ and p is in the right subtree, it must be that $j' \leq diff(p, t \rightarrow rchild \rightarrow lptr \rightarrow v, 1)$. Therefore, j'' is equal to $diff(p, t \rightarrow rchild \rightarrow lptr \rightarrow v, 1)$. If $j'' = d + 1$, then p is equal to $t \rightarrow rchild \rightarrow lptr \rightarrow v$. Otherwise, since p is in the tree we know that $x_{j''}(p) > x_{j''}(t \rightarrow rchild \rightarrow lptr \rightarrow v)$. Therefore, after setting t to $t \rightarrow rchild$ and j to j'', we know that (i) holds.

The case when $comp = high$ is similar. When p is not in the tree, the procedure is slightly different since additional statements need to be added. In this case the path followed during the search starts at the root and continues through all those nodes for which $t \rightarrow lptr \rightarrow v \leq p \leq t \rightarrow hptr \rightarrow v$. If at some point both of the direct descendants of a node do not satisfy the above condition, then either the search terminates with an answer **false**, or (iii) will hold from that point on. The specific details about the search strategy are spelled out in our procedure. It is important to note that once (iii) holds, j has no meaning. However, to guard against reporting that p is in the tree when it is not, we perform an additional test. This is why we call it AVC. Let us now outline procedure MEMBERSHIP(p, t) to test whether or not point p is in the subtree rooted at t.

```
MEMBERSHIP(p ,t );
/* Is (x1(p), x2(p), ..., xd(p)) in the d-dimensional balanced binary search tree rooted at t */
comp ← low;
j ← diff (t→lptr→v ,p ,1);
while t ≠ null do
 case
  :comp = low:
   if j = d+1 then if p = t→lptr→v then return(true) else return(false);
   case
    :t→jl < j or (t→jl = j and xj(p) < xj(t→v)):
        t ← t→lchild;
    :t→jl > j or (t→jl = j and xj(p) > xj(t→v)):
        if t→rchild = null then return(false);
        j' ← diff (p ,t→rchild→lptr→v ,j);
        if j' = d+1 then if p = t→rchild→lptr→v then return(true) else return(false);
        if xj'(p) < xj'(t→rchild→lptr→v) then return(false);
        t ← t→rchild; j ← j';
    :else: /* t→jl = j and xj(p) = xj(t→v) */
        j' ← diff (p ,t→v ,j);
        if j' = d + 1 then if p = t→v then return(true) else return(false);
        case
         :xj'(p) < xj'(t→v): /* try left subtree */
             j'' ← diff (p ,t→lchild→hptr→v ,j');
             if j'' = d + 1 then if p = t→lchild→hptr→v then return(true) else return(false);
             if xj''(p) > xj''(t→lchild→hptr→v) then return(false);
             comp ← high; t ← t→lchild; j ← j'';
         :xj'(p) > xj'(t→v): /* try right subtree */
             j'' ← diff (p ,t→rchild→lptr→v ,j');
             if j'' = d + 1 then if p = t→rchild→lptr→v then return(true) else return(false);
             if xj''(p) < xj''(t→rchild→lptr→v) then return(false);
             t ← t→rchild; j ← j'';
        endcase
   endcase
  :comp = high:
   /* This section of code is omitted since it is similar to the one for comp = low. */
 endcase
endwhile
return(false);
end of procedure MEMBERSHIP
```

Following arguments similar to the ones for the case when p is in the tree, one can easily prove the following lemma. It is trivial to modify the procedure so that when it returns the answer true it also returns a pointer to the place where the element is stored, for brevity we did not include such instructions.

Lemma 1: Given a point p procedure MEMBERSHIP(p, t) determines whether or not p is in the d-dimensional balanced binary search tree rooted at t in $O(d + \log n)$ time.

Proof: The proof follows arguments similar to those we used for the case when p is in the tree.

□

We have identified an algorithm that implements (A) within the proposed time complexity bound. Let us now consider how to implement (B), i.e., if p is not in the tree then find the position where it should be inserted. The node where procedure MEMBERSHIP terminates may not be the correct place where insertion should take place. However, while executing procedure MEMBERSHIP we can save the path traversed in the tree while searching for p. By the path traversed, we mean all the nodes to which t pointed to plus the next node that would be visited (i.e., the procedure returns false just before t is set to $t \rightarrow rchild$ or $t \rightarrow lchild$ and such a pointer was not null). Suppose that the path traversed is given in figure 9. The a_i are pointers to the nodes. Suppose that the node pointed at by a_{12} is the last node and it is not a leaf node. The small triangles are subtrees and the dot in it represents the place where an element smaller (larger) than all the elements in the subtree would be inserted. We label those positions $b_0, b_1, \ldots, b_{12}$. Later on we define the procedure for performing this labeling.

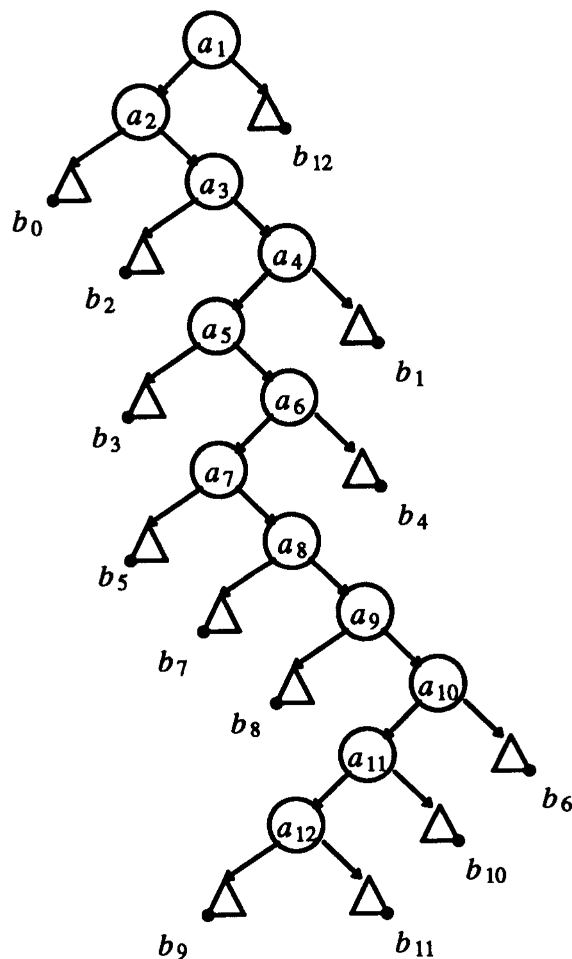


Figure 9: Tree nodes searched.

Since (ii) or (iii) hold at each step in the traversal, it must be that all the nodes for which (ii) holds are visited before all the nodes for which (iii) holds. Furthermore, there is at least one node for which (iii)

holds. To determine the the place where p must be inserted, we find the last node (if any) in the path for which (ii) holds. Consider figure 9. If (ii) does not hold for the node pointed at by a_1, then p should be inserted at location b_0 or b_{12}; if the last node for which (ii) holds is the one pointed at by a_1, then p should be inserted at b_1; and so forth. Note that it cannot be that the last node for which (ii) holds was the node pointed at by a_{12}. Using the above strategy it is simple to write a procedure that given the last node for which (ii) holds determines where the element should be inserted. Let us now show how to find such a node. Let $p_1, p_2, \ldots$ point to nodes with values $a_1 \rightarrow v$, $a_2 \rightarrow v$, ..., $a_{12} \rightarrow v$. There are O($log\ n$) elements in the list, since the length of the path is O($log\ n$). One can easily construct the list sorted in O($log\ n$) time by traversing the nodes in the path top-down. The sorted list of elements for the path given in figure 9 is:

$$a_2 \rightarrow v < a_3 \rightarrow v < a_5 \rightarrow v < a_7 \rightarrow v < a_8 \rightarrow v < a_9 \rightarrow v <$$
$$a_{12} \rightarrow v < a_{11} \rightarrow v < a_{10} \rightarrow v < a_6 \rightarrow v < a_4 \rightarrow v < a_1 \rightarrow v .$$

In this list we need to find the appropriate place for p. This can be accomplished as follows. Let k = 1. First let us eliminate all the elements which disagree with component k of p in the list. This can be accomplished by traversing the list top-down and bottom-up. The top-down (bottom-up) traversal advances if the element in the list has a value smaller (larger) than the one in p. If there remain no elements then the place where the search ends is the location where p belongs. Otherwise, we increase the value of k and repeat the above process. Eventually the appropriate position will be found. Clearly this process takes O(d + $log\ n$) time since there are O($log\ n$) elements. All of the above observations are summarized in lemma 2. For simplicity of exposition we separated this part of the algorithm from procedure MEMBERSHIP; however, it is simple to see how it can be incorporated into that procedure.

Lemma 2: Given a point p which is not in the tree t, an algorithm based on procedure MEMBERSHIP(p, t) and the above observations determines where p should be inserted in the d-dimensional balanced binary search tree rooted at t in O(d + $log\ n$) time.

Proof: The proof is based on lemma 1 and the arguments that appear before lemma 2.

□

We have identified an algorithm that implements (B) within the proposed time complexity bound. Let us now consider (C). If the tree is empty just before the insert operation, then the update of a single node is trivial. Suppose now that element p is added to a non-empty tree. Let q point to the node added. Now we must update the structure to reflect the new value at some nodes that are predecessors of node q. Let r be the predecessor to q closest to the root and such that the path from r to q contains at least one arc and consists only of *rchild* or *lchild* (but not both) tree arcs. Let us assume the path consists of only

rchild (*lchild*) tree arcs. Then, the *hptr* (*lptr*) of all of these nodes must now point to the new node. The value *jh* (*jl*) in the path must be updated. 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 increasing (decreasing). Therefore, the *jh* (*jl*) values are also increasing (decreasing). The correct values can be easily computed in $O(d + \log n)$ time by reusing previously computed *jh* (*jl*) values. Lemma 3 summarizes our observations.

Lemma 3: After inserting a point p in a d-dimensional balanced binary search tree and just before rotation the structure can be updated in $O(d + \log n)$ time.
Proof: By the above discussion.

□

We have identified an algorithm that implements (C) within the proposed time complexity. It is simple to see that a similar procedure can be used to implement (E). We now need to consider how to implement (D), i.e., rotations. This is the simplest part. Consider the rotation given by figure 10. Clearly the only nodes whose information needs to be updated are a_1 and a_2. Clearly, since there is a fixed number (2) of them the operations can be implemented to take $O(d)$ time. This result is summarized in lemma 4.

Lemma 4: After a rotation in a d-dimensional balanced binary search tree the structure can be updated in $O(d)$ time.
Proof: By the above discussion.

□

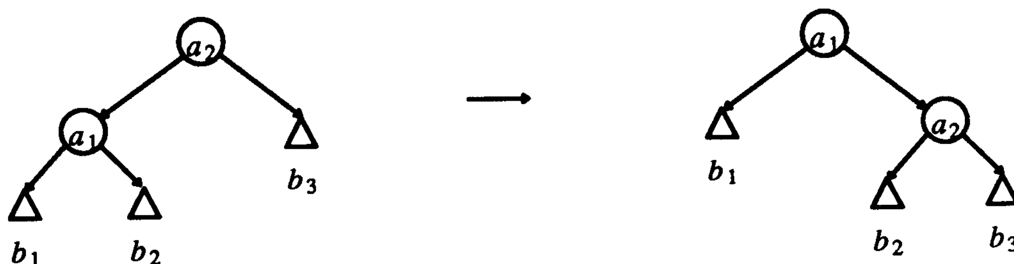


Figure 10: Rotation.

Using arguments similar to the ones in lemmas 3 and 4, one can easily show that (F) can be implemented to take $O(d + \log n)$ time. Our main result which is based on the above discussions and the lemmas is given below.

Theorem 1: Any on-line sequence of operations of the form INSERT(p), DELETE(p) and MEMBERSHIP(p), where p is any point in d-space can be carried out by the above procedures on a d-dimensional balanced binary search trees in $O(d + log\ n)$ time, where n is the 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 operations on balanced binary search trees [T].

□

III. DISCUSSION.

It is interesting that our technique cannot be adapted to AVL trees, weight balanced trees or B-trees of fixed order, because the number of rotations in those structures might be large ($\Omega(log\ n)$). Therefore, the claims on the proposed time complexity would not hold. The main reason why they hold on balanced binary search trees is that only O(1) rotations are needed.

For very large 1-dimensional data sets one normally uses B-trees of high order (degree) and the information is stored in external devices. The time required for insert and delete is not $O(log\ n)$, but the number of nodes visited is small. The objective in this case is to minimize the number of nodes visited since that is equal to the number of times one retrieves records from the external device which is the time consuming part of the algorithm. Our techniques can be adapted to handle the general case when the points are in d-space. The number of nodes visited will be small. The TRIE plus binary search tree approach using B-trees [GK] does not have this property since the number of nodes accessed could be as large as d. Another promising alternative for this case is to use the normal B-tree without any additional information stored at the nodes. Remember that the objective is just to visit the least number of nodes.

The TRIE plus binary search tree approach requires less space to represent the elements than ours. However, our procedures are simple and only a constant number of rotations are required after each INSERT and DELETE operations.

For simplicity we defined the procedures for MEMBERSHIP in multiple phases. It is simple to see that the multiple phases may be performed concurrently while traversing the tree from the root. It is important to note that procedures based on our techniques can be easily coded, for brevity we did not include the detailed procedures.

IV. REFERENCES.

[B] Bayer, R., "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms," *Acta Informatica*, 1, 1972, 290 - 306.

[BSa] Bentley, J. L., and Saxe, J. B., "Algorithms on Vector Sets," SIGACT News, Fall 1979, 36 - 39.

[C] Clampett, H. A., "Randomized Binary Searching With the Tree Structures," *Comm. ACM*, 7, No. 3, 1964, 163 - 165.

[GK] Gueting, R. H., and Kriegel, H. P., "Multidimensional B-tree: An Efficient Dynamic File Structure for Exact Match Queries," Proceedings 10th GI Annual Conference, Informatik Fachberichte, Springer-Verlag, 1980, pp. 375 - 388.

[GS] Guibas, L. J., and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, 8 - 21.

[H] Hirschberg, D. S., "On the Complexity of Searching a Set of Vectors," *SIAM J. on Computing* Vol. 9, No. 1, February 1980, 126 - 129.

[K] Kosaraju, S. R., "On a Multidimensional Search Problem," 1979 ACM Symposium on the Theory of Computing, 67 - 73.

[M] Mehlhorn, K, " Dynamic Binary Search," *SIAM J. Computing*, Vol. 8, No. 2, May 1979, 175 - 198.

[O] Olivie, H. J., "A New Class of Balanced Search Trees: Half-Balanced Binary Search Trees," Ph.D. Thesis, University of Antwerp, U.I.A., Wilrijk, Belgium, 1980.

[S] Sussenguth, E. H., "Use of Tree Structures for Processing Files," *Comm. ACM*, 6, No. 5, 1963, 272 - 279.

[T] Tarjan, R. E., "Updating a Balanced Search Tree in O(1) Rotations," *Information Processing Letters*, 16, 1983, 253 - 257.

[V1] Vaishnavi, V., "Multidimensional Height-Balanced Trees," *IEEE Transactions on Computers*, Vol. c-33, No. 4, April 1984, 334 - 343.

[V2] Vaishnavi, V., "Multidimensional Balanced Balanced Binary Trees," *IEEE Transactions on Computers*, Vol. 38, No. 7, April 1989, 968 - 985.