Chapter 44
The On-line d-Dimensional Dictionary Problem*

Teofilo F. Gonzalez†

Abstract

We present a new algorithm for the on-line d-dimensional dictionary problem which has many applications including the management of geometrical objects and geometrical searching. The dictionary 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. We introduce a clean structure based on balanced binary search trees, which we call d-dimensional balanced binary search trees, to represent the set of points. We present algorithms for each of the above operations that 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, and each could take $\Omega(d)$ time.

1 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 carried out in $O(log\ n)$ time, where n is the current number of elements 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 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 DELETE operations ([10], [12]).

The on-line d-dimensional dictionary problem has a multitude of uses when accessing multi-attribute 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 [4] use the abstract data type implemented in this paper to find suboptimal hyperrectangular covers for a set of multidimensional points. This covering problem has applications in the location of emergency facilities so that all users are within a reasonable distance of one of the facilities and it also has applications in image processing [4]. The current set of points is denoted by P and point $p \in P$ 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 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 a (small) constant number of rotations are required for each INSERT and DELETE operation.

As noted in [9], it was a common belief over a decade 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 have combined TRIES 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 $0, 1, \ldots, m-1$ (see structure in figure

*This research was carried out during the author's Sabbatical leave at the Department of Computer Science, Utrecht University, 3508 TB Utrecht, The Netherlands

†Author's current address: Department of Computer Science, University of California, Santa Barbara, CA, 93106 (teo@cs.ucsb.edu)

1, where $\Sigma = 0, 1, 2, 3$).

(2) (Sussenguth [11]) Each internal node is represented by a linear list (see structure in figure 2).

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

In general, implementation (1) requires the largest amount of space and implementation (2) uses the least amount of space. Also, efficient algorithms that operate on (1) are the fastest and the ones that operate on (2) are the slowest. The performance of (3) is between that of (1) and (2).

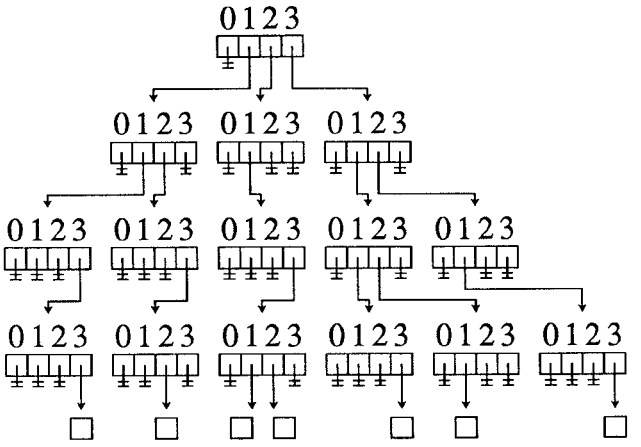

Figure 1: TRIE representation.

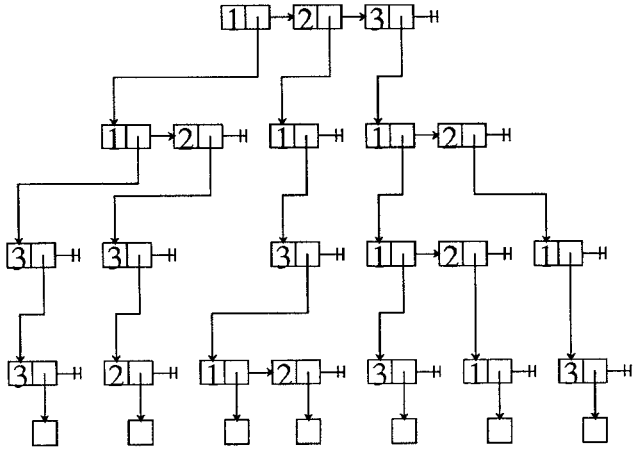

Figure 2: Linked list representation for TRIE nodes.

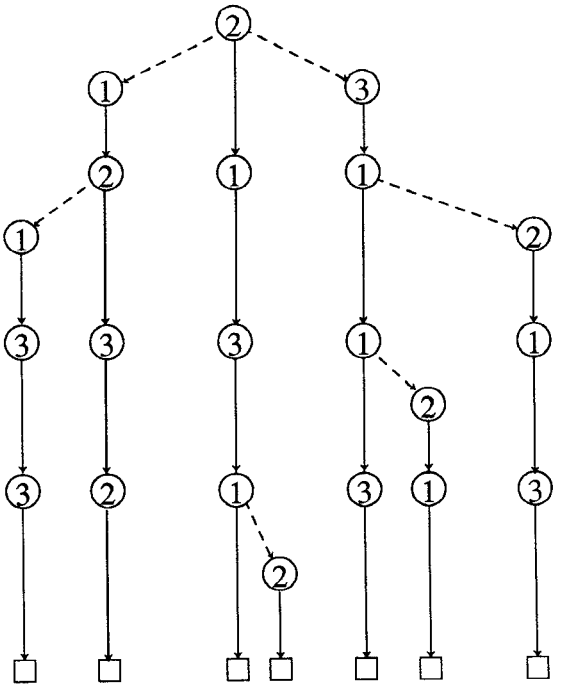

Figure 3: Binary search representation for TRIE nodes. Dashed arcs are binary search tree pointers and other arcs are TRIE pointers.

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 $\Sigma = 0, 1, \ldots, m-1$ (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 [2] used this technique together with the following fully 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 ([8] and [7]). Fully balanced subtrees are very rigid structures that cannot be easily updated. Therefore, are not appropriate for dynamic updates, and 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 [5], weight balanced trees [9], AVL trees [13], and balanced binary search trees [14]. 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.

We investigate a representation which is based solely on binary search trees, rather than on a combination of TRIES and binary search trees. 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 our trees 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. 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, which can be accomplished by performing one simple verification step.

2 The Algorithms.

In this section we outline our procedures and our structure to implement d-dimensional dictionaries. Our representation is based solely on balanced binary search trees, rather than based on TRIES and binary search trees as previous algorithms. It is important to note that our trees are of the same form as the ones in [12], except for the fact that all the pointers to external nodes in [12] are replaced by null pointers in this paper. For example, an internal node with two external nodes as children in [12] is a leaf node in this paper. Before explaining our procedures and our new data structure, let us present a couple of naive approaches for solving the d-dimensional dictionary problem.

Suppose that we represent our set of points by a balanced binary search tree in which each point is stored as a d-tuple at a node and the ordering of the d-tuples in the tree is lexicographic. Procedure NAIVE(p, r) implements MEMBERSHIP in the obvious way, i.e., compares p with the value stored at the root of a subtree and depending on the outcome it either terminates, or proceeds to the left or right subtree of that node. The value stored at a node is referred to by v and its components by $x_1(v), x_2(v), \ldots, x_d(v)$. Let p and q be two points (in d-space). For $1 \leq i \leq d$, 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 $x_k(p) = x_k(q)$, $i \leq k < j$, and $x_j(p) \neq x_j(q)$, unless no such j exists, in which case j is $d+1$). In what follows we say that j is the first diff starting at i between p and q when j is equal to $diff(p, q, i)$. When i is 1 we say that j is the first $diff$ between p and q. Procedure NAIVE is formally given below.

```
procedure NAIVE(p, r);
  t ← r;
  while t ≠ null do
    j ← diff(p, t, 1);
    case
      :j = d + 1: return(true);
      :x_j(p) < x_j(t → v): t is set to point to the
                            left subtree of t;
      :x_j(p) > x_j(t → v): t is set to point to the
                            right subtree of t;
    endcase
  endwhile
  return(false);
end of procedure NAIVE;
```

It is simple to show procedure NAIVE performs the MEMBERSHIP operation correctly; however, its time complexity is $O(d\ log\ n)$ and there are problem instances for which it requires $\Omega(d\ log\ n)$ time.

Let us now modify the above procedure and reduce its time complexity to $O(d + log\ n)$. The algorithm is similar, but the difference is that instead of comparing elements starting always at position 1, we start the comparison where we stopped during the previous iteration. The procedure is given below.

```
procedure FAST-NAIVE(p, r);
  t ← r;
  i ← 1;
  while t ≠ null do
    j ← diff(p, t, i);
    /* the case statement is identical to
       the one in procedure NAIVE */
    i ← j;
  endwhile
  return(false);
end of procedure FAST-NAIVE;
```

It is simple to show that the time complexity for procedure FAST-NAIVE is $O(d + log\ n)$. Let us now apply the procedure to search for points in the tree

given in figure 4. Suppose p = (1,0,0,0,0). Procedure FAST-NAIVE sets t to the root of the tree and j to 2. The procedure 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 the point (2,3,0,8,7) the procedure returns the value 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 p to $t \rightarrow v$ when procedure FAST-NAIVE claims success. Let us call this new procedure MOD-FAST-NAIVE. Since the number of additional operation is $O(d)$, the time complexity for FAST-NAIVE is $O(d + log\ n)$. Procedure MOD-FAST-NAIVE is based on the principle assume-verify-and-conquer (AVC). The idea is to avoid multiple verifications by assuming 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. Unfortunately, the procedure is 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 correctly for (1,3,0,8,7).

Let us now discuss our structure and the procedures that operate on it. Each node in the tree has the following information in addition to the information required to manipulate balanced binary search trees, i.e., the rank bit (see [12]).

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

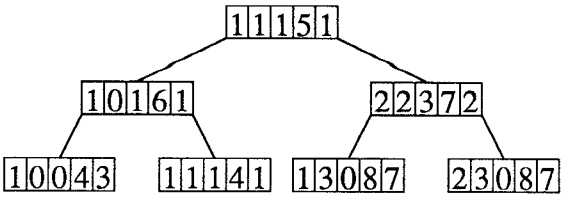

Figure 4: Sample balanced binary search tree.

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 ([10], [12]); 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 a node with element p. 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 ancestors (in the tree) 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, some information stored at some nodes in the path from the root to the node inserted is 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. 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. 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 position where the node was deleted to the root of 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 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 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, r) to test whether or not point p given by $(x_1(p), x_2(p), \ldots, x_d(p))$ is in the d-dimensional binary search tree (or subtree) rooted at r. This procedure implements (A) above, and as we shall see later on it can be easily modified to implement (B). Initially t is set to r and *comp* is set to *low*. At each iteration, t, points to the root of a subtree, and if $comp = low$ $(high)$ then j has a value between 1 and d such that the jth component of p and the smallest (largest) element in t differ or $j = d + 1$. As we shall see, j also satisfies some additional properties. We say that p is *in-bounds* at the subtree rooted at t if p is greater than or equal to the smallest element in subtree t, and less than or equal to the largest element in subtree t (i.e., $t \rightarrow lptr \rightarrow v \leq p \leq t \rightarrow hptr \rightarrow v$), otherwise p is *out-of-bounds* at the subtree rooted at t. We claim that at each step in our algorithm, one of the following three statements holds.

(i) If p is in the tree rooted at r then p is in-bounds and j is the first component where p and the smallest (largest) value in t differ when $comp = low$ $(high)$.

(ii) If p is not in the tree rooted at r and p is in-bounds at t, then j is the first component where p and the smallest (largest) value in t differ when $comp = low$ $(high)$.

(iii) If p is not in the tree rooted at r, then p is out-of-bounds at t.

Let us outline our strategy when searching for p at node t under the assumption 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 only consider the case when $comp = low$, since the other case is symmetric. We claim that at each step in the algorithm (i) holds true. Initially t points to the root of the tree and j is the first diff between p and the smallest value in t. This operation together with the assumption that p is in the tree implies that (i) holds initially. We now show that if (i) holds during the kth iteration and our algorithm performs certain operations (that we specify below), then either p is found in the tree and the algorithm terminates, or (i) holds at the $k+1$st iteration. If $j = d + 1$, then we know that p is equal to the smallest element in t and we return; otherwise, $j < d + 1$. There are three cases.

Case 1: $(t \rightarrow jl = j$ **and** $x_j(p) < x_j(t \rightarrow v))$ **or** $t \rightarrow jl < j$ (see figure 5).

By assumption p is in-bounds at t, and j is the first diff between p and the smallest value in t. 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. Therefore, the left subtree of t is not *null* and p is in-bounds at the left subtree of t, since we know that p is in the tree rooted at r. By definition the smallest value in t and the smallest value in the left subtree of t are identical. Therefore, after resetting t to $t \rightarrow lchild$, we know that (i) holds.

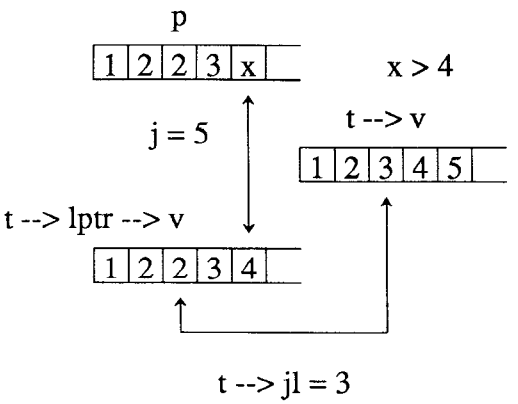

Figure 5: Case 1: $t \rightarrow jl < j$.

Case 2: $(t \rightarrow jl = j$ **and** $x_j(p) > x_j(t \rightarrow v))$ **or** $t \rightarrow jl > j$ (see figure 6).

By assumption p is in-bounds at t, and j is the first diff between p and the smallest value in t. 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 left subtree of t. Therefore, the right subtree of t is not *null* and p is in-bounds at the right subtree of t, since we know that p is in the tree rooted at r. Let j' be the first diff starting at j between p and the smallest element in the right subtree of t. Since j is less than or equal to the first diff between p and the value stored at t, and p is in the right subtree of t, it must be the case that j is less than or equal to the first diff between p and the smallest value in the right subtree of t. So, j' is equal to the first diff between p and the smallest value in the right subtree of t. If $j' = d + 1$, then p is equal to the smallest element in the right subtree of t. Otherwise, after setting t to $t \rightarrow rchild$ and j to j', we know (i) holds.

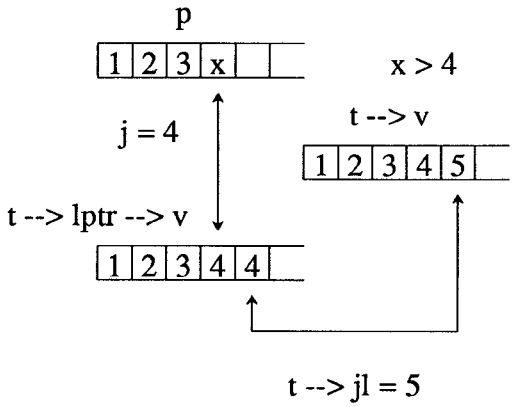

Figure 6: Case 2: $t \rightarrow jl > j$.

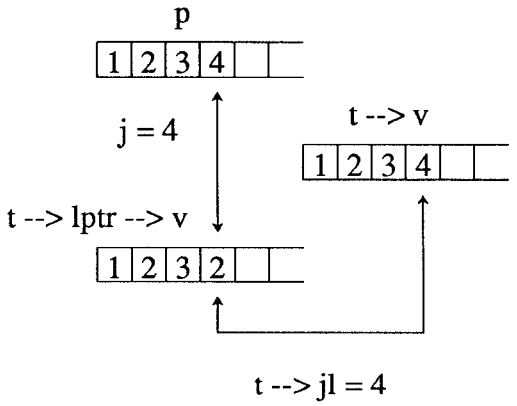

Figure 7: Case 3: $t \rightarrow jl = j$.

Case 3: $t \rightarrow jl = j$ **and** $x_j(p) = x_j(t \rightarrow v)$ (see figure 7).

By assumption p is in-bounds at t, and j is the first diff between p and the smallest value in t. Let j' equal the first diff starting at j between p and the value stored at t. Since $j = t \rightarrow jl$, we know that j is less than or equal to the first diff between p and the value stored at t. Therefore, j' is equal to the first diff between p and the value stored at t. If $j' = d + 1$, then p is equal to the value stored at t and we return. Otherwise, $j' \leq d$, and there are two separate subcases.

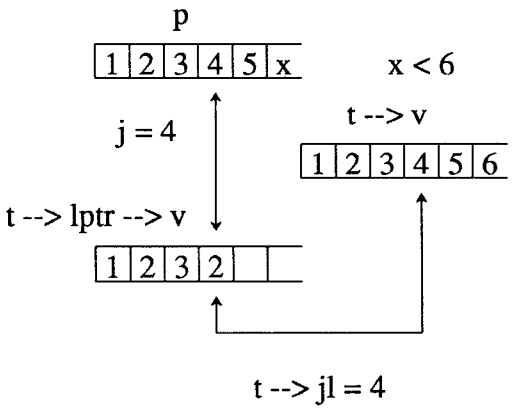

Figure 8: Subcase 3.1: $x'_j(p) < x'_j(t \rightarrow v)$.

Subcase 3.1: $x'_j(p) < x'_j(t \rightarrow v)$ (figure 8).

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. Therefore, the left subtree of t is not *null* and p is in-bounds at the left subtree of t, since we know that p is in the tree rooted at r. It is important to note that if at the next iteration we proceed with *comp* equal to *low*, then the time complexity bound for our procedure will not be the proposed one, because if this situation arises many times the total time required to compute the j's could be as large as $\Omega(d \log n)$. To eliminate this problem we switch *comp* to *high*. Let j'' equal to the first diff starting at j' between p and the largest value in the left subtree of t. Since j' is equal to the first diff between p and the value stored at t, and p is in-bounds at the left subtree of t, it must be that j' is less than or equal to the first diff between p and the largest value in the left subtree of t. Therefore, j'' is equal to the first diff between p and the largest value in the left child of t. If $j'' = d + 1$, then p is equal to the largest value in the left subtree of t and return. Otherwise, (i) holds after setting t to $t \rightarrow lchild$, j to j'' and *comp* to *high*.

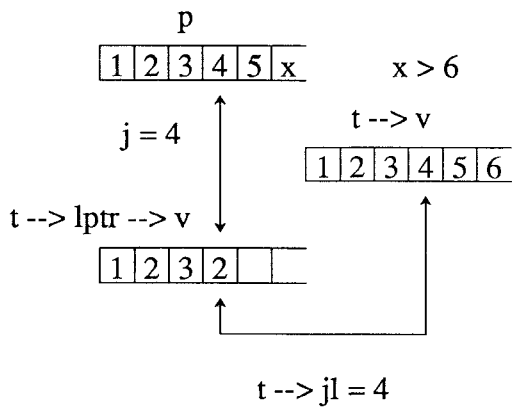

Figure 9: Subcase 3.2: $x'_j(p) > x'_j(t \rightarrow v)$.

Subcase 3.2: $x'_j(p) > x'_j(t \rightarrow v)$ (figure 9). 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 left subtree of t. Therefore, the right subtree of t is not *null* and p is in-bounds at the right subtree of t, since we know that p is in the tree rooted at r. Let j'' be the first diff starting at j' between p and the smallest element in the right subtree of t. Since j' is equal to the first diff between p and the value stored at t and p is in-bounds in the right subtree of t, it must be that j' is less than or equal to the first diff between p and the smallest element in the right subtree of t. Therefore, j'' is equal to the first diff between p and the smallest value stored in the right subtree of t. If $j'' = d+1$, then p is equal to the smallest element in the right subtree of t. Otherwise, (i) holds after setting t to $t \rightarrow rchild$ and j to j''.

We shall not discuss the case when $comp = high$ because it is similar. When p is not in the tree, the procedure is slightly different. In this case the path followed during the search starts at the root and continues through all those nodes for which p is in-bounds. If at some point both of the children of a node are out-of-bounds for p, then either the search terminates with an answer **false**, or we advance to one of the children of the t and (iii) will hold from that point on. The specific details about the search strategy are spelled out in procedure MEMBERSHIP given below. It is important to note that once (iii) holds, j has no "important" meaning. However, to guard against reporting that p is in the tree when it is not, we perform an additional test (verification step). The assumption that we make is that p is in the tree. When the assumption is wrong, it is caught by the verification step. This is why we call the technique assume-verify-and-conquer (AVC). Let us now formally define procedure MEMBERSHIP(p, r) to test whether or not point p is in the subtree rooted at r.

```
procedure MEMBERSHIP(p, r);
/* Is (x_1(p), x_2(p), ..., x_d(p)) in the d-dimensional
   balanced binary search tree rooted at r */
comp ← low; t ← r;
j ← diff(p, t → lptr → v, 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 and x_j(p) < x_j(t → v)) or
     t → jl < j: /* Case 1 */
       t ← t → lchild;
    :(t → jl = j and x_j(p) > x_j(t → v)) or
     t → jl > j: /* Case 2 */
       code-for-case-2( );
    :else: /* Case 3 */
      j' ← diff(p, t → v, j);
      if j' = d + 1 then
           if p = t → v then return(true)
                        else return(false);
      case
        :x'_j(p) < x'_j(t → v): /* Subcase 3.1 */
          code-for-subcase-3.1( );
        :x'_j(p) > x'_j(t → v): /* Subcase 3.2 */
          code-for-subcase-3.2( );
      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


procedure code-for-case-2( );
 t ← t → rchild;
 if t = null then return(false);
 j' ← diff(p, t → lptr → v, j);
 if j' = d + 1 then
    if p = t → lptr → v then return(true)
       else return(false);
 if x'_j(p) < x'_j(t → lptr → v) then return(false);
 j ← j';
end of procedure code-for-case-2
```

procedure code-for-subcase-3.1( );
  $t \leftarrow t \rightarrow lchild$;
  if $t = null$ **then return(false)**;
  $j'' \leftarrow diff(p, t \rightarrow hptr \rightarrow v, j')$;
  if $j'' = d + 1$ **then**
    if $p = t \rightarrow hptr \rightarrow v$ **then return(true)**
      else return(false);
  if $x_{j''}(p) > x_{j''}(t \rightarrow hptr \rightarrow v)$ **then return(false)**;
  $comp \leftarrow high$; $j \leftarrow j''$;
end of procedure code-for-subcase-3.1

procedure code-for-subcase-3.2( );
  $t \leftarrow t \rightarrow rchild$;
  if $t = null$ **then return(false)**;
  $j'' \leftarrow diff(p, t \rightarrow lptr \rightarrow v, j')$;
  if $j'' = d + 1$ **then**
    if $p = t \rightarrow lptr \rightarrow v$ **then return(true)**
      else return(false);
  if $x_{j''}(p) < x_{j''}(t \rightarrow lptr \rightarrow v)$ **then return(false)**;
  $j \leftarrow j''$;
end of procedure code-for-subcase-3.2

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. Following arguments similar to the ones for the case when p is in the tree, one can easily prove the following lemma.

LEMMA 2.1. *Given a point p procedure MEMBERSHIP(p, r) determines whether or not p is in the d-dimensional balanced binary search tree rooted at r 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 by figure 10. The a_is 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 triangles are subtrees and the dot in them represents the place where an element smaller or larger than all the elements in the subtree would be inserted. We label those locations $b_0, b_1, \ldots, b_{12}$. The procedure that performs this labeling is omitted, since it is straight forward.

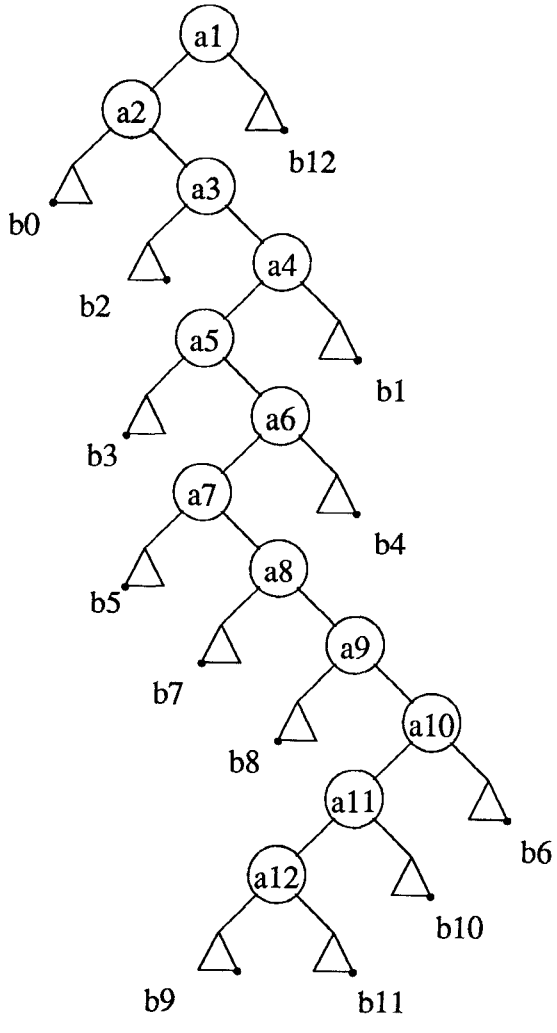

Figure 10: Tree nodes searched.

Since (ii) or (iii) hold at each step in the traversal and t is a binary search tree, it must be that all the nodes for which p is in-bounds are visited before all the nodes for which p is out-of-bounds. Furthermore, there is at least one node for which p is out-of-bounds and all the nodes in r for which p is in-bounds are in the path. To determine the place where p must be inserted, we find the last node (if any) in the path for which p

is in-bounds. Consider figure 10. If p is out-of-bounds at a_1, then p should be inserted at b_0 or b_{12}; if the last node for which p is in-bounds is a_1, then p should be inserted at b_1; and so forth except for a_{12}. Remember that it cannot be that the last node for which p is in-bounds is a_{12}. Using the above strategy it is simple to write a procedure that given the last node for which p is in-bounds it determines where p should be inserted. Let us now show how to find such a node. The list of nodes is $a_1 \rightarrow v, a_2 \rightarrow v, \ldots$. 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 10 is:

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

Suppose that we find the appropriate place for p in the above list. If p is smaller (larger) than all the elements in the list then p is out-of-bounds at a_1 and it should be inserted at b_0 (b_{12}). On the other hand if it appears before (after) a_{12} but just after (before) $a_i \rightarrow v$ then the last node for which p is in-bounds is a_i and it should be inserted at b_i. Let us now outline our procedure to find the appropriate place for p in the list. Initially k is set to 1. Now 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 component k of the element in the list has a value smaller (larger) than the one in p. If there remain no more elements in the list, then the place where the search ends is the location where p belongs. Otherwise, we increase the value of k and resume the above process where we left. Eventually the appropriate position for p 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 2.2. 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 2.2. *Given a point p which is not in the tree r, an algorithm based on procedure MEMBERSHIP(p, r) and the above observations determines where p should be inserted in the d-dimensional balanced binary search tree rooted at r in $O(d + log\ n)$ time.*

Proof. The proof is based on lemma 2.1 and the arguments that appear just before lemma 2.2.

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 s be the predecessor to q closest to the root and such that the path from s 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 increasing. The correct values can be easily computed in $O(d+log\ n)$ time by reusing previously computed jh (jl) values. Lemma 2.3, whose proof is omitted, summarizes our observations.

LEMMA 2.3. *After inserting a point p in a d-dimensional 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 (C) within the proposed time complexity. It is simple to see that a similar procedure can be used to implement (E). Let us now consider how to implement (D), i.e., rotations. This is the simplest part. A simple rotation is shown in figure 11. We only consider single rotations, since the compound rotations in [12] can be obtained by applying several single rotations. 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 2.4, without a proof.

LEMMA 2.4. *After a rotation in a d-dimensional balanced binary search tree the structure can be updated as mentioned above in $O(d)$ time.*

Using arguments similar to the ones in lemmas 2.3 and 2.4, 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 2.1. *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 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 operations on balanced binary search trees [12].

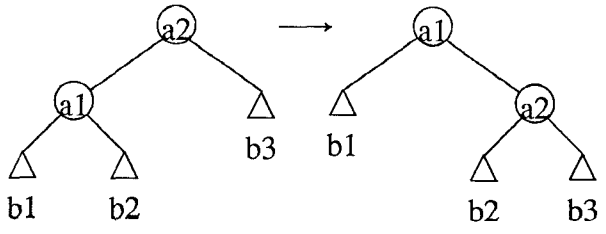

Figure 11: Rotation.

3 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 ($\Omega(log\ n)$). Since each rotation could take $\Omega(d)$ time, the proposed time complexity bounds would not hold. The main reason why they hold on balanced binary search trees is that only O(1) rotations are needed. An $O(d + log\ n)$ time algorithm to CONCATENATE two sets represented by our structure can be easily obtained. However, the SPLIT operation cannot be implemented within this time complexity bound. The main reason is that there could be $\Omega(log\ n)$ rotations.

For simplicity we defined the procedures for the MEMBERSHIP operation 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. 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.

4 Acknowledgements.

The author wishes to thank J. van Leeuwen and R. Tarjan for their comments and suggestions on preliminary versions of this paper. The author also wishes to thank J. van Leeuwen and M. Overmars for the bibliographical pointers they provided the author.

References

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

[2] J. L. Bentley, and J. B. Saxe, *Algorithms on Vector Sets*, SIGACT News, (Fall 1979), pp. 36–39.

[3] H. A. Clampett, *Randomized Binary Searching With the Tree Structures*, Comm. ACM, 7, No. 3, (1964), pp. 163–165.

[4] T. Gonzalez, *Covering a Set of Points with Fixed Size Hypersquares and Related Problems*, Proceedings of the 29th Annual Allerton Conference on Communications, Control and Computing, (October 1990), pp. 838–847, (to appear Information Processing Letters).

[5] R. H. Gueting, and H. P. Kriegel, *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.

[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. on Computing Vol. 9, No. 1, February (1980), pp. 126–129.

[8] S. R. Kosaraju, *On a Multidimensional Search Problem*, 1979 ACM Symposium on the Theory of Computing, pp. 67–73.

[9] K. Mehlhorn, *Dynamic Binary Search*, SIAM J. Computing, Vol. 8, No. 2, (May 1979), pp. 175–198.

[10] 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).

[11] E. H. Sussenguth, *Use of Tree Structures for Processing Files*, Comm. ACM, 6, No. 5, (1963), pp. 272–279.

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

[13] V. Vaishnavi, *Multidimensional Height-Balanced Trees*, IEEE Transactions on Computers, Vol. c-33, No. 4, (April 1984), pp. 334–343.

[14] V. Vaishnavi, *Multidimensional Balanced Balanced Binary Trees*, IEEE Transactions on Computers, Vol. 38, No. 7, (April 1989), pp. 968–985.