AN IMPLEMENTATION OF A MULTIDIMENSIONAL DYNAMIC RANGE TREE BASED ON AN AVL TREE

by

Michael G. Lamoureux

TR95-100, November 1995

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca

An Implementation of a Multidimensional Dynamic Range Tree Based on an AVL Tree

Faculty of Computer Science Technical Report TR95-100

by

Michael G. Lamoureux

University of New Brunswick
Faculty of Computer Science
P.O. Box 4400
Fredericton, N.B., Canada
E3B 5A3

phone: (506) 453-4566 fax: (506) 453-3566
e-mail: y320@unb.ca (Michael G. Lamoureux)

ABSTRACT

In this paper we develop a dynamic multidimensional range tree data structure based on the AVL tree. The range tree was originally defined as a static data structure which is optimal ($O(\lg^k n + t)$ for t points in range), in a worst case analysis, for answering k-dimensional range queries on k-dimensional data when only $O(n\lg^{k-1}n)$ space is used. Theoretical modifications to the structure do exist such that the structure can be used in a dynamic environment and maintain both its low range search time and storage requirements (in fact, the structure is optimally balanced), but, to the extent of the author's knowledge, a detailed specification of the structure along with a corresponding implementation does not yet exist in the literature.

This paper defines the necessary transformations that are needed to transform the AVL tree into a 1-d dynamic range tree and the necessary modifications that are needed to extend this structure into a k-dimensional structure that can handle k-dimensional data. The resulting structure satisfies the specifications for a dynamic range tree and is optimally balanced. It is implemented in Pascal with complete code being provided in the appendices. This structure is important as it provides us with a baseline structure against which other structures, and their implementations, designed to handle multidimensional range queries on multidimensional data can be compared.

Keywords: range tree, AVL tree, range query, multidimensional data, dynamic data structures, optimally balanced data structures

Table of Contents

1 INTRODUCTION

The problem of k-d range search has long been of interest to computer science. Efficient access to large volumes of multi-attribute data is a fundamental requirement of most large-scale computer information systems. This is precisely where the k-d range search problem is applicable.

However, the problem is not as clear cut as it may seem. In order to have rapid execution of queries, the data must be organized such that range search is very fast but we must also insure that the storage requirements of the underlying structure are kept minimal in order to feasibly store the structure for use. Also, today's information systems require a dynamically updatable database so any index structure for the database must also allow dynamic updates, which must also be fast to ensure feasibility. These interdependent requirements imply that the design of a dynamic data structure for efficient k-d range search is difficult and often unintuitive.

Although range search is commonly provided by relational algebra operations on index structures based on the B-Tree [Come79] and permitted by a large number of other data structures, we find that, with respect to range search, many of these structures are either inefficient, difficult to implement, or unable to handle k-dimensional data efficiently without extensive modifications. Other structures that permit range search include binary search trees and their multidimensional equivalents (the k-d trees of [Bent75]), point quadtrees (see [Same90]), and priority search tree based data structures (see [McCr85] and [Edel81]). However, except for the RT-Tree of [Edel81] which, in fact, is based on a modification of the range tree of Bentley, none are as efficient, using a worst-case analysis, as the range tree in answering k-d range queries when the overall balance of the structure is taken into account (see [Lamo95b]).

The range tree was first introduced by Bentley ([Bent80]) as a static data structure which was designed to answer k-d range queries quickly. It had worst-case cost functions of $O(n\lg^{k-1}n)$ for storage, $O(n\lg^{k-1}n)$ for preprocessing, and $O(\lg^k n + t)$, to locate t points in range, for k-d range search. Willard and Lueker ([Will85], [Will85b], [Luek78], and [Luek82]) have since, independently and jointly, defined modifications on the structure to dynamize it, but their modifications were mostly theoretical and, as far as the author is aware, a detailed description of a dynamic range tree data structure has not yet been laid down nor has a k-dimensional implementation appeared in the literature.

The dynamic range tree permits update operations (insertions and deletions) in worst case time complexity of $O(\lg^k n)$ when an amortized analysis is used and the structure is optimal for the execution of k-dimensional range queries in a dynamic environment when the underlying structure is optimally balanced.

The purpose of this paper is to clearly define a 1-d dynamic range tree as a modified AVL tree and then extend the structure such that it is able to handle and efficiently process k-dimensional data according to the above cost functions. This structure is important as it provides us with a baseline structure against which future structures, and their implementations, designed to handle multidimensional range queries on multidimensional data, can be compared.

2 RANGE SEARCH

Using the definition of [Knut73], we define a *range query* as a query that asks for records (in a File F containing n records) whose attributes fall within a specific range of values (e.g. height > 6'2" or \$23,000 <= annual income <= \$65,000). We will call these limits L for low and H for high. Boolean combinations of range queries over different attributes are called *orthogonal range queries*. When the conjunction of range queries is required, we can view each separate attribute as one dimension of a k-dimensional space, and the orthogonal range query corresponds to asking for all records (points) that lie within a k-dimensional (hyper) rectangular box. A range search is performed to retrieve all records which satisfy the query. When specifying the limits of an orthogonal range query, we will use the limit vectors **L**, for lower limits, and **H**, for upper limits, and use the notation $\mathbf{L}_i$ to correspond to the ith lower limit (and, similarly, $\mathbf{H}_i$ corresponds to the ith upper limit).

We generally use five cost functions to portray and analyze the cost of range search on a specific data structure G that supports range search on F. The three basic cost functions (pertaining to those found in [Bent79]) are

$P(n,k)$ = preprocessing time required to build G,
$S(n,k)$ = storage space required by G,
$Q_R(n,k)$ = time required to perform a range search on G,

and, in addition, we must consider the time required to insert a point into or delete a point from G as we are permitting dynamic updates on our structure. The cost of these dynamic operations are represented as

$I(n,k)$ = time required to insert a new record into G, and
$D(n,k)$ = time required to delete a record from G.

Following the example of [Will85] we will use $U(n,k)$ to refer to the update cost function when the cost function for insertion, $I(n,k)$, is of the same order of complexity as the cost function for deletion, $D(n,k)$. This is often the case when the insertion and deletion operations are dependent upon the search operation. Also, we are often concerned with the time to locate a single record in our data structure G as a member query on some data structures designed for range queries is often of a lower order of complexity than range query. We thus use $Q_M(n,k)$ to denote the time required to answer a member query on G. (For example, in the k-d range tree of Bentley ([Bent80]) and in the k-d Range DSL of Lamoureux and Nickerson ([Lamo95a] and [Lamo95b]), one can perform a member query, $Q_M(n,k)$, in $O(\lg n)$ time. This is true since a record is indexed by the substructure in the first dimension if it appears in the structure.)

The convention in this paper, as in [Lamo95a] and [Lamo95b], is that, unless otherwise stated, we are performing a worst case Big-Oh analysis (which, by definition, is accurate to within a constant factor).

3 AVL TREES

In this section we give a brief overview of the AVL tree of [Avel62]. Our discussion is similar to that found in [Krus94] and we refer the reader to that reference for a more complete overview of this well known structure.

AVL trees achieve the goal that searches, insertions, and deletions in a search tree with n nodes can all be achieved in O(lg n) time, even in the worst case, while maintaining a minimum storage requirement of O(n). The AVL tree is a height balanced binary search tree structure whose height can never exceed 1.44 lg n, and thus, even in the worst case structure, the performance of AVL trees is comparable to that of the completely balanced binary search tree.

The height restriction is imposed by the definition of an AVL tree, which is as follows. An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

The node structure is essentially that of a binary search tree with an additional field for the *balance factor* which is set either *left high ('/'* or *-1), equal ('-'* or *0)*, or *right high ('\'* or *1)* to indicate that either the left subtree is higher than the right subtree, the two subtrees are of equal height, or that the right subtree is higher than the left subtree, respectively.

Insertion and deletion proceed analogously to insertion and deletion in binary search trees, with the major difference being that a *rotatio*n is used to restore the balance criterion (the two subtrees of a node cannot differ in height by more than 1) whenever an insertion or deletion causes the balance criterion to be violated.

There are four basic rotations that are used to restore the balance in AVL trees. They are called either single rotations (two) or double rotations (two), depending on the number of pointer variables that change. These are displayed in Figure 1. We note that the double rotations are actually a concatenation of two single rotations and that the rotations are applied to the first node (on the insertion/deletion path) encountered on the way back up to the root that violates the balance criterion.

We illustrate the basic rotations now as they are used in the modified AVL Tree which we use to implement our basic 1-d dynamic range tree structure. They are straightforward and we refer the reader to [Krus94] for a more detailed exposition of the dynamic update algorithms which includes working Pascal code. We define the nodes r, x, and w as the primary nodes of the rotation as the rotation is centered on these nodes and we designate the primary node r as the root node of the rotation. We also note that for a double rotation, one of the subtrees B and C must have a height of h.

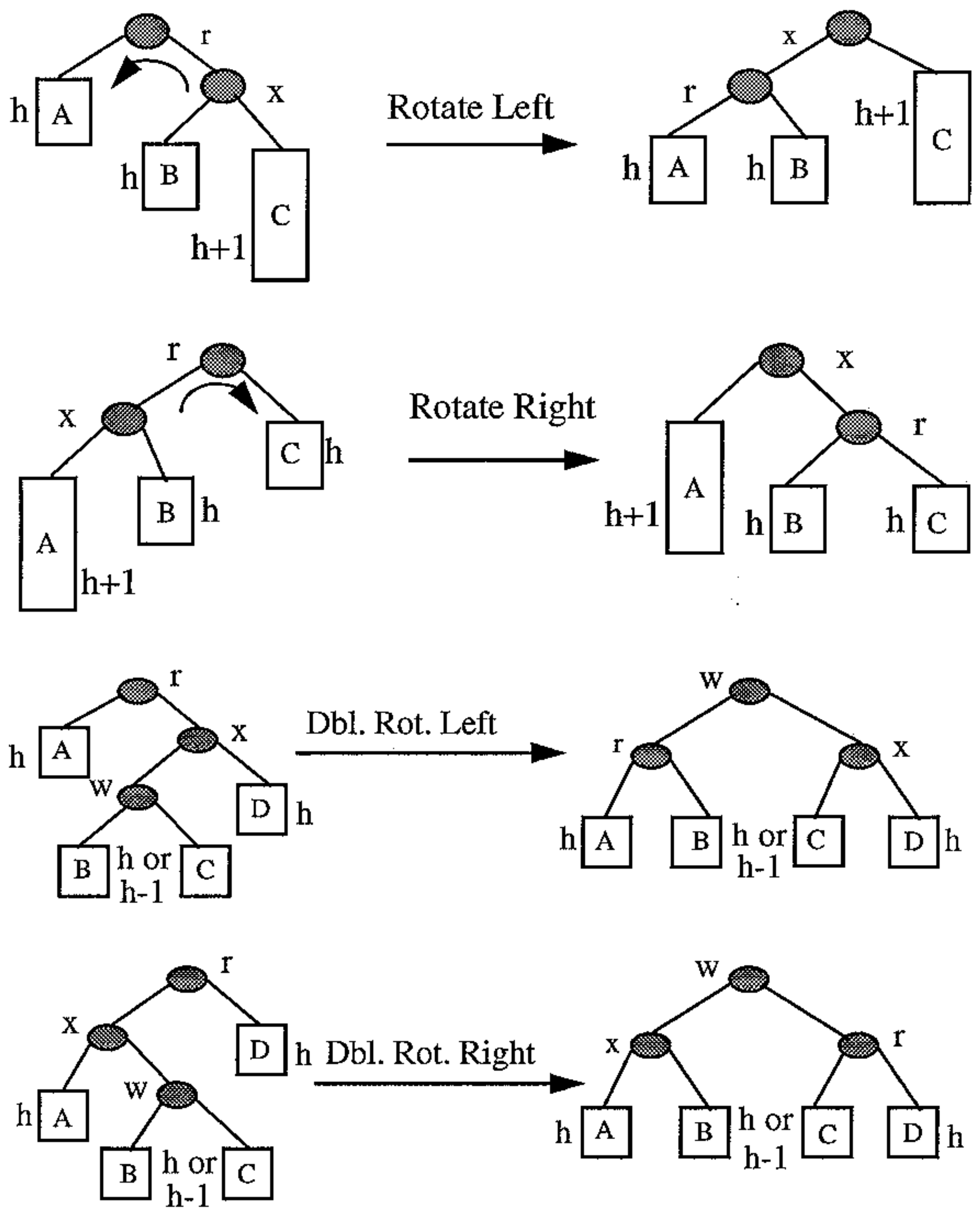


Figure 1. AVL tree rotations used to restore the balance criterion.

4 THE RANGE TREE

The range tree of Bentley ([Bent80]), like a binary search tree, stores datapoints and is designed to detect all points that lie in a given range (i.e. it was designed with range queries in mind). We will briefly review the range tree structure and refer the reader to [Same90] for a good overview.

A one dimensional range tree is a balanced binary search tree where the data points are stored in the leaf nodes and the leaf nodes are linked in sorted order by a linked list (i.e. the leaf nodes are threaded). A range search for [L:H] is performed by searching the tree for the node with the smallest key $\geq$ L and then following the links until reaching a leaf node with a key that is greater than or equal to H. For n points, we see that this procedure takes O(lg n + t) time and uses O(n) storage.

A two dimensional range tree is simply a range tree of range trees. We build a two-dimensional range tree as follows: We first sort all of the points along one of the attributes, say x, and then store them in a balanced one-dimensional range tree, say T. We then append to each non-leaf node, I, of the range tree T a range tree T_I of the points in the sub-tree rooted at I where these points are now sorted along the other attribute, say y.

A range search for ($[L_x:H_x],[L_y:H_y]$) is carried out as follows. It starts by searching the tree T for the smallest key that is $\geq L_x$, say L_x', and the largest key that is $\leq H_x$, say H_x'. It then finds the common ancestor of L_x' and H_x', Q, that is closest to them and assigns $\{PL_i\}$ and $\{PH_i\}$ to be the sequences of nodes, excluding Q, that form the paths in T from Q to L_x' and H_x', respectively.

Let LEFT(P) and RIGHT(P) denote the left and right children, respectively, of non-leaf node P and Range_Tree(P) be the one-dimensional range tree for y that is stored at non-leaf node P. Then, for each P that is an element of $\{PL_i\}$ such that LEFT(P) is also in $\{PL_i\}$, we perform a one-dimensional range search for $[L_y:H_y]$ in the one-dimensional range tree rooted at node RIGHT(P) {Range_Tree(RIGHT(P))}. For each P that is an element of $\{PH_i\}$ such that RIGHT(P) is also in $\{PH_i\}$, we perform a one-dimensional range search for $[L_y:H_y]$ in the one-dimensional range tree for y rooted at node LEFT(P) {Range_Tree(LEFT(P))}. We also check to see if Lx' and Hx' are in the given range.

It should be clear that the above search algorithm runs in $O(\lg^2 n + t)$ time and that the general search algorithm for k-dimensional range queries, which is extended in a manner that is analogous to the extension from the 1-d case to the 2-d case, runs in $O(\lg^k n + t)$ time.

The recursively defined k-dimensional range tree requires P(n,k) of $O(n\lg^{k-1}n)$ and S(n,k) of $O(n\lg^{k-1}n)$ in the static case, and $P(n,k) = O(n\lg^k n)$, $S(n,k) = O(n\lg^{k-1}n)$, and $U(n,k) = O(\lg^k n)$ in the dynamic case (see [Will85], [Will85b], [Luek78], and [Luek82]) where an amortized worst case analysis is used.

The range tree is an optimally balanced data structure (see [Lamo95b]) and optimal for the execution of range queries in the class of optimally balanced structures.

5 THE AVL TREE AS A RANGE TREE

There are two major differences between the AVL Tree and the range tree. The first difference is that the AVL tree stores key values corresponding to data records (data points) in all of its nodes, both internal and leaf, while the range tree only stores key values corresponding to data records in it's leaf nodes. The second major difference is that the leaf nodes of the range tree are threaded in sorted order whereas the AVL tree does not have any of its nodes threaded.

Thus, to use the AVL Tree as a range tree we must impose modifications that restrict key values corresponding to actual data records to the leaf nodes and modify the insertion and deletion algorithms so that the leaf nodes are threaded. Although these modifications may initially seem difficult to impose at first glance, they are very straightforward and very effective.

We accomplish the first modification by noting that we can maintain the existence of our key values only at the leaf level by inserting (deleting) two nodes instead of one at each insertion (deletion). The procedures one uses to accomplish this are essentially the normal AVL tree insertion and deletion procedures with the modifications outlined below. We call an AVL tree with this modification a range AVL tree.

One of the two nodes corresponds to the key value being (inserted / deleted), and thus is a leaf node, and the other node is an internal node that (takes the place of / is replaced by) an existing leaf node which must (drop down / move up) a level to allow for the (insertion / deletion) of a leaf node. This is illustrated in Figure 2.

In Figure 2(a) we are inserting a key value y corresponding to a new data record and, using the normal AVL tree insertion algorithm, we find that the last node on our insertion path is x. We replace node x with a new internal node, i, and drop x down into the left or right subtree of i, depending on whether y is greater than x or less than x, respectively. The node corresponding to the new key value y is then placed in the empty subtree and the remainder of the normal AVL tree insertion algorithm is used to ensure that the balance criterion is maintained.

In Figure 2(b) we are deleting a key value y corresponding to an existing node in the tree. Using the normal AVL tree deletion algorithm, modified such that a key value only matches the target key value if it is in a leaf node, we arrive at the target node at the deepest level of recursion. We make the modification that we don't remove the node when we return to the parent node (as we unfold the recursion) but wait until we return to the grandparent node to process the deletion. At this point the parent node is replaced by the sibling of the node we are removing and the remainder of the normal AVL tree deletion algorithm is used to ensure that the balance criterion is maintained.

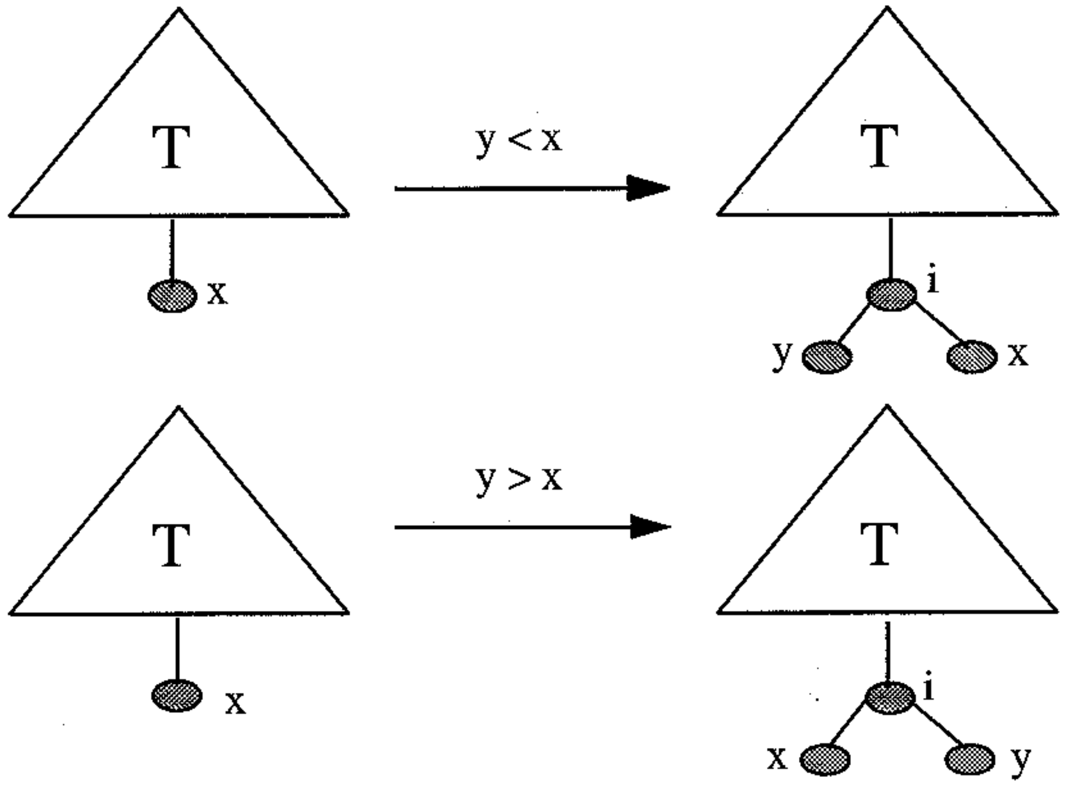


Figure 2(a). Insertion of key value y into a range AVL Tree.

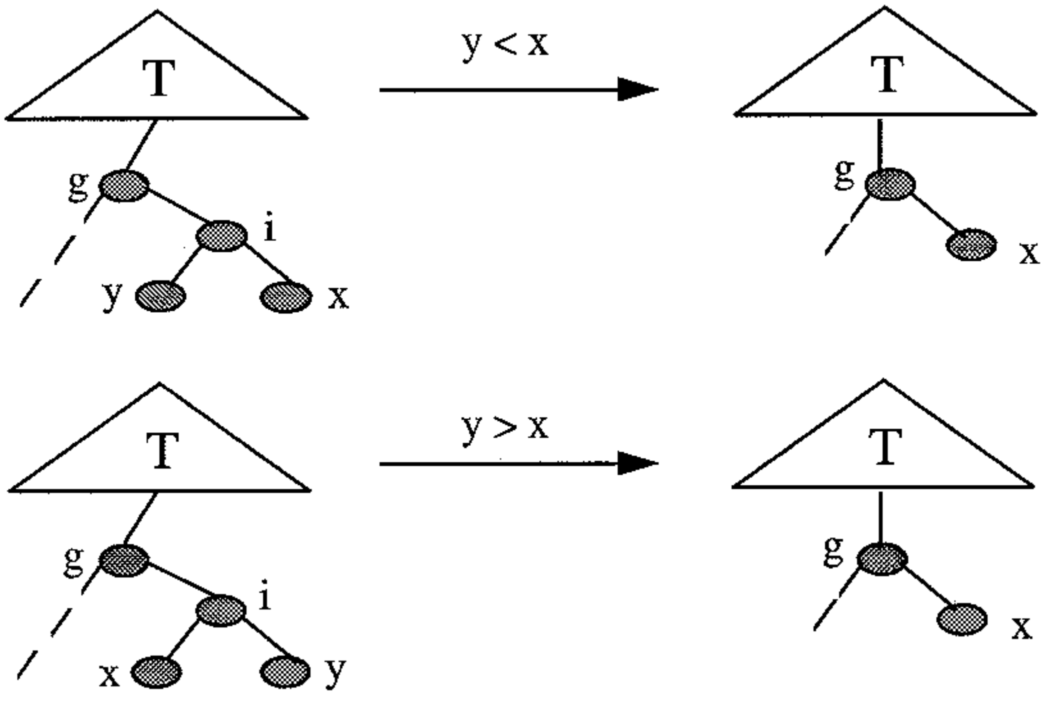


Figure 2(b). Deletion of key value y from a range AVL Tree.

We would like to note that the additional restriction that every node must have 0 or 2 children (maintained by inserting/deleting two nodes at a time) implies that we have to define two additional single rotations to handle the two new special cases that arise in the delete procedure. These are illustrated in Figure 3. They are essentially the same as the two standard single rotations with the only difference being that the primary node of the rotation which is a child of the root node has a balance factor of equal height ('-' or 0).

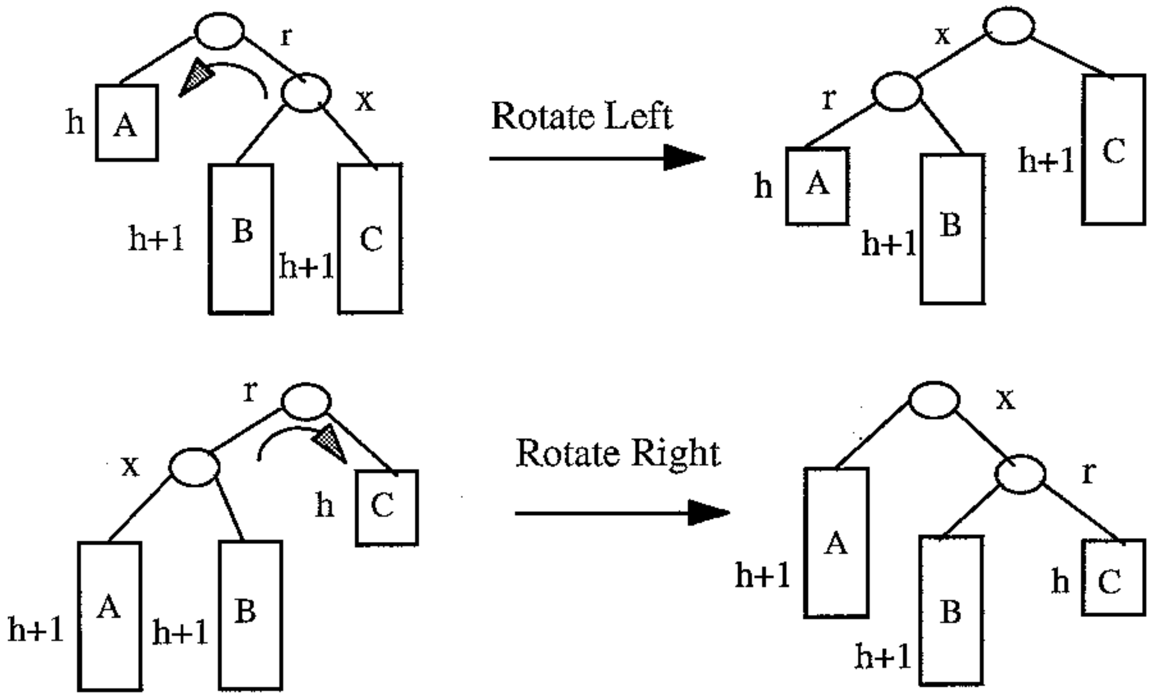


Figure 3. Additional rotations needed (in delete procedure) for range AVL tree.

The second modification we must make is that all the leaf level nodes appear in a doubly-linked linked list in sorted order, i.e. the leaf nodes are threaded in sorted order. This modification is fairly simple both to impose and maintain as a leaf node can never be a primary node of a rotation.

Leaf nodes can never be primary nodes of rotations because we are dealing with a restricted class of AVL trees. This class is partially illustrated in Figure 4. The fact that we insert two nodes every time we add a record to and delete two nodes every time we remove a record from our data set implies that we are working with a restricted class of AVL trees where every interior node must have both a left child and a right child. This implies that the root node of the rotation is at least the third interior node up from the leaf level and, since the primary nodes of an AVL tree rotation, as seen in Figure 1, can span at most three levels, the primary nodes, obviously, can not be leaf level nodes.

We thus correctly deduce that the threaded linked list at the leaf level only changes when a node is inserted or deleted and this implies that maintenance of a doubly linked list is not only simple but efficient as well. In order to insert a node we must first locate that node which is to be its sibling (which then becomes its predecessor or successor in the doubly linked list) and this allows us to quickly insert the node into the doubly linked list (in fact, we do so in constant time). Every node at the leaf level in the tree is in the doubly linked list so deleting a node is quite simple as well as the location of the node also determines its position in the doubly linked list. Thus we can maintain the doubly linked list in a time bounded by the normal AVL tree insertion and deletion times.

We refer the reader to Appendix 1 for the complete insertion and deletion algorithms and corresponding type definitions and Appendix 2 for the complete Pascal code. We point out that a data set of one record is a special case where we have a tree which consists solely of one leaf node and we also leave a detailed analysis to the reader pointing out that since we have an AVL tree structure, our worst case cost functions must be as follows:

$P(n) = O(n \lg n)$, $S(n) = O(n)$, $U(n) = O(\lg n)$, and $Q_R(n) = O(\lg n + t)$.

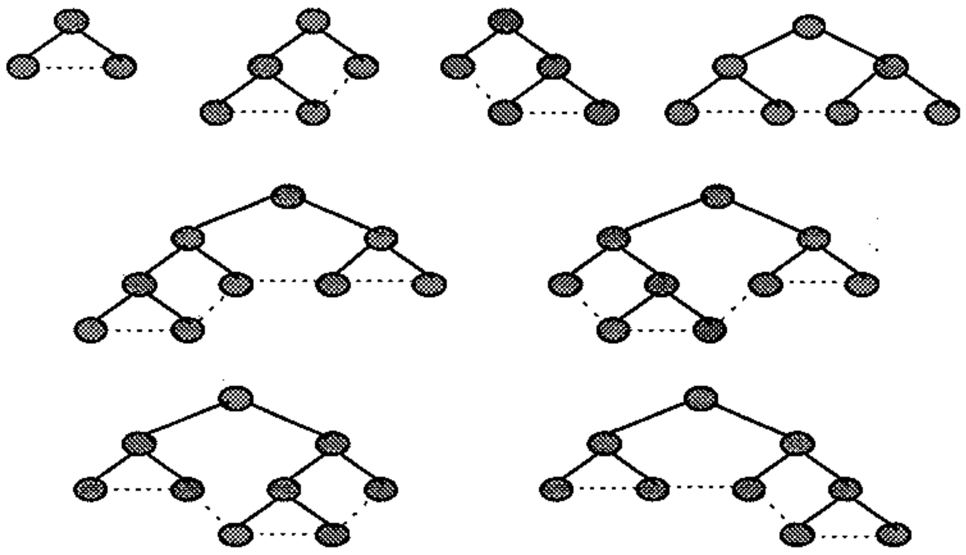


Figure 4. Some range AVL trees.

6 THE K-D RANGE TREE STRUCTURE

If we return to the definition of our k-d range tree, we find that at each interior node, I, of our tree, T^k, in dimension k we have a pointer to a range tree, T_I^{k+1}, in dimension k+1 which contains the points of the subtree rooted at I sorted along the k+1 coordinate value. Therefore, at each interior node of our range AVL tree in dimension k we must have a pointer to an range AVL tree in dimension (k+1) which contains copies of the points in the subtree rooted at I ordered on the next coordinate value.

If we ignore rotations for the moment, this condition is straightforward to maintain. As we progress down the tree, in dimension k, searching for the place where we perform the actual insertion or deletion, we insert the appropriate nodes into or delete the appropriate nodes from the range AVL tree structures in the next dimension associated with each node that we encounter along our search path. We add a second level of recursion to our existing insertion and deletion algorithms so that the algorithms recurse both through a 1-d range AVL tree structure and through the dimensions of the structure to give us our dynamic k-d range tree data structure.

Complications arise when we need a rotation to restore balance. The existing AVL tree rotations are not adequate when a tree structure becomes unbalanced in one dimension as each node has a pointer to a corresponding range AVL tree structure in the next dimension which would become invalid for primary nodes of the rotation. This complication is illustrated in Figure 5 where we use w, r, and x to label the primary nodes and (a), (b), and (c) for the associated range AVL tree structures in the next dimension.

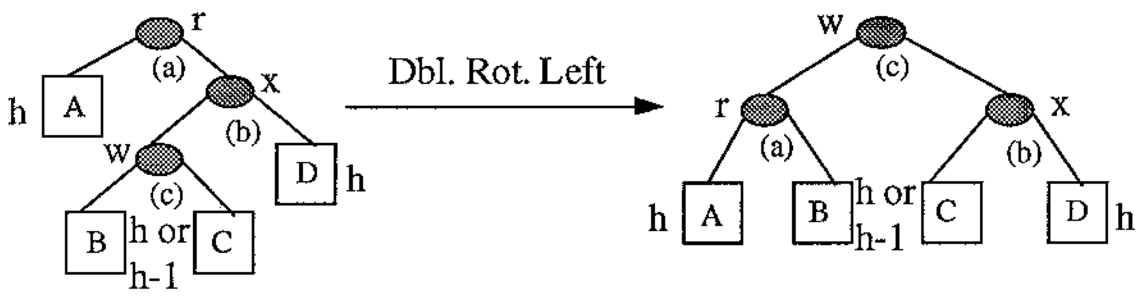


Figure 5. An illustration of the complications of normal AVL tree rotations in k-d space that result in an invalid range AVL tree.

Before the rotation: (a) contains the points in A, B, C, and D, (b) contains the points in B, C, and D, and (c) contains the points in B and C; and after the rotation: (a) contains the points in A and B, (b) contains the points in C and D, and (c) contains the points in A, B, C, and D.

This tells us that partial rebuilding is required. However, it is rather inefficient to rebuild all three range AVL tree substructures, especially when we only have to partially rebuild two. When we rotate the nodes, we exchange their pointers to the range AVL tree substructures in the next dimension as well. For instance, if we let node w get (a), node x get (c), and node r get (b), as illustrated in 6(c), then we only have to rebuild (b) and (c).

The reason that we let node x get (c) while node r gets (b) and not the other way around is that a double rotation is two single rotations and this is the result of a single rotation right on x followed by a single rotation left on r. Also, assigning (b) to node x and (c) to node r, as illustrated in 6(d), still results in (b) and (c) being invalid.

One might argue that assigning node (b) to x and node (c) to r is a better idea as we would then have only approximately 3/4 n update operations as compared to approximately 5/4 n update operations to validate (b) and (c), but we argue that this is irrelevant. Both assignments require $O(n\lg^{k-1}n)$ work to validate the substructures (b) and (c) and thus it is faster to rebuild the substructures using partial rebuilding which requires $O(n\lg^{k-2}n)$ time rather than choose one pointer assignment over another.

The rotations that we will use are illustrated in Figure 6 where w, r, and x remain the primary nodes and (a), (b), and (c) remain pointers to range AVL tree substructures in the next dimension. Note that, for the double rotations, one of B or C must be of height h. The reader is referred to Appendix 3 for the complete insertion and deletion algorithms and type definitions and Appendix 4 for the corresponding Pascal code.

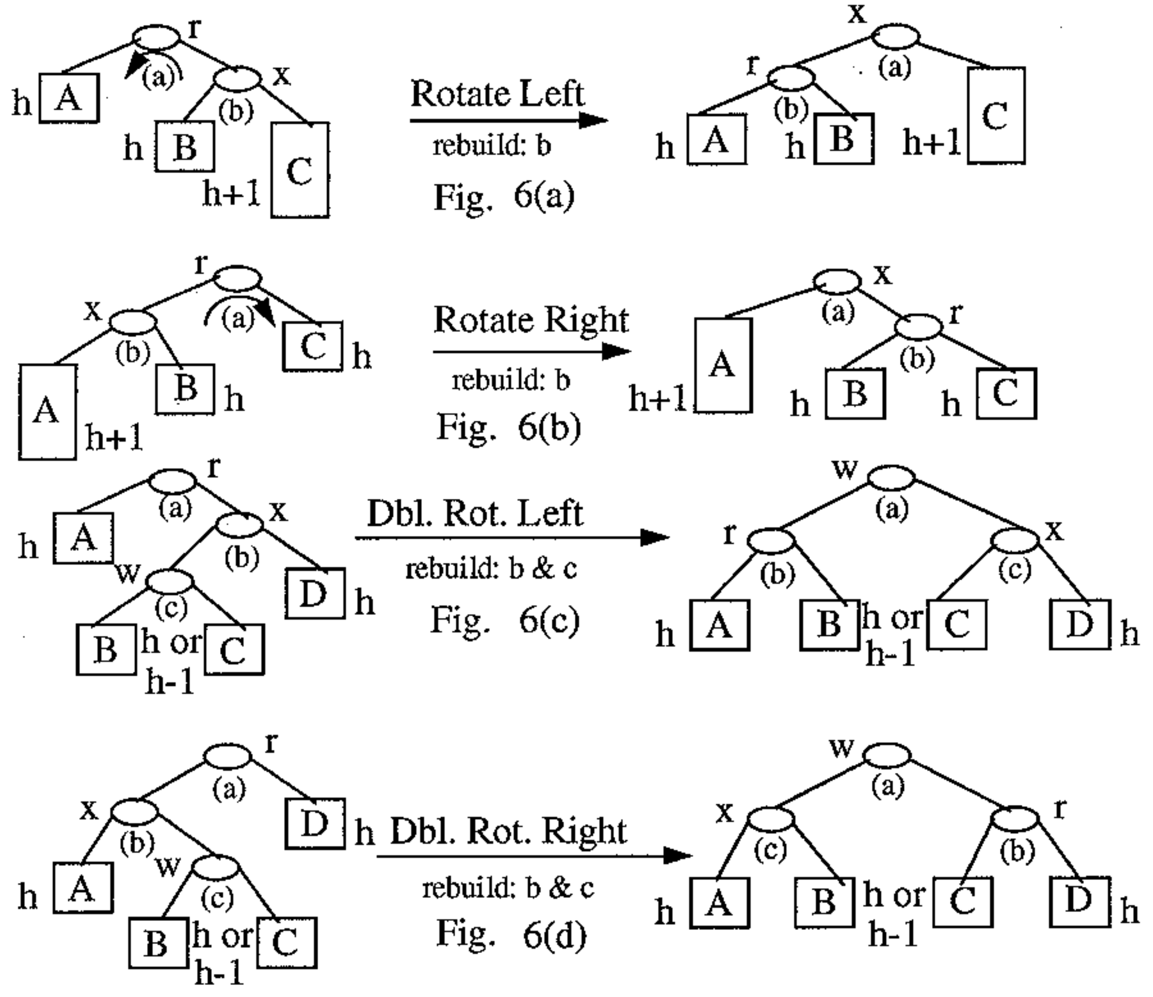


Figure 6. Rotations for a k-d dynamic range AVL tree.

7 ANALYSIS OF THE K-D RANGE TREE STRUCTURE

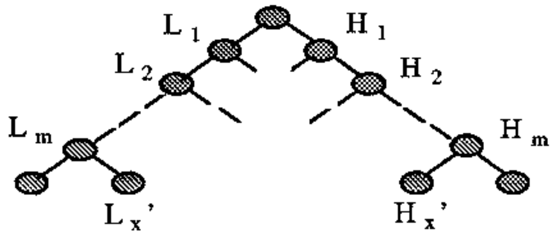
In this section we prove that the cost functions of our data structure are those of the k-d dynamic range tree and that our structure is optimally balanced (see [Lamo95b]).

Theorem 1: The time, $Q_R(n,k)$, required to perform a k-d range search on a k-d dynamic range AVL tree is $O(\lg^k n + t)$, for t the number of datapoints found in the specified range.

Proof:

The 1-d case is equivalent to performing a range search in both an AVL tree (if we use recursion) and a 1-d range tree (if we use the doubly linked list at the leaf level) and we know that both of these procedures require $O(\lg n + t)$ time.

The 2-d case is slightly more involved. The worst case is when the nearest common ancestor is the root and we have to search two range AVL tree structures in the next dimension at each level of our AVL tree except for the level that consists of the root alone, the last level of interior nodes on our way down to the nodes L_x' and H_x', and the leaf level. This can be visualized by observing the following diagram:



We see that for each of $L_1, L_2, ..., L_{m-1}$ and for each of $H_1, H_2, ..., H_{m-1}$ we must search a range AVL tree structure in the next dimension (given by Range_Tree(RIGHT(L_i)) and Range_Tree(LEFT(H_i))). We thus search at most $O(\lg n)$ range AVL tree structures in the next dimension (we search two per level on $O(\lg n)$ levels) at a search time bounded by $O(\lg n)$ per structure to give us the required search time of $O(\lg^2 n + t)$.

We now prove by induction that $Q_R(n,k) = O(\lg^k n + t)$. We have our base case of $Q_R(n,1) = O(\lg n)$ and we assume that $Q_R(n,k) = O(\lg^k n + t)$. We now show that $Q_R(n,k+1) = O(\lg^{k+1} n + t)$. In the worst case, we need to search one range AVL tree structure in the last dimension, k, corresponding to each of the interior nodes in our node sequences $\{PL_i\}$ and $\{PH_i\}$ except for the last two nodes, L_m and H_m, of the $O(\lg^{k-1} n)$ range AVL tree structures we are searching in dimension k. We search at most 2 range AVL tree structures per level of a range AVL tree structure and, as there are only $O(\lg n)$ levels in a range AVL tree structure, we compute the upper bound of $O(\lg^{k+1} n + t)$ as it only takes $O(\lg n + t)$ time to search a single range AVL tree structure and we are searching $O(\lg^k n)$ range AVL tree structures in dimension (k+1). ■

We will prove our worst case bound for insertion in two steps. Defining a simple insertion as an insertion which does not cause a rotation to be performed; we first prove that the time required for a simple insertion, $I_S(n,k)$, is bounded by $O(\lg^k n)$ and then prove that the time required for any necessary rebuilding over a sequence of n insertions, $R_{SI}(n,k)$, is of $O(n\lg^{k-1}n)$. This will give us our amortized insertion cost, $I(n,k)$, of $O(\lg^k n)$. We will then go on to prove our worst case bound for deletion, which is equivalent, analogously. Note that the definition of a simple deletion is analogous to the definition of a simple insertion and that our definitions are in agreement with [Lamo95a].

However, before we can determine our rebuilding costs, and therefore our insertion cost, along with our storage requirement and static preprocessing cost, we must state and prove the following lemma which gives us the number of nodes in dimension k.

Lemma 1: The number of nodes in dimension k is $O(n\lg^{k-1}n)$.

Proof:

This result is obvious for dimension one. If we have n datapoints it is clear that our range AVL tree has exactly 2n-1 nodes, one interior node for every leaf node except for the first leaf node inserted. This gives us O(n) nodes, as required.

The k-d case turns out to be slightly more difficult to calculate, but not much. We find that we have $2^x\ C_{k-2}^{x+k-2}$ structures of $n/2^x$ points in dimension k or

$$O(n\sum_{x=0}^{\lg n-1}(C_{k-2}^{x+k-2})) \tag{1}$$

nodes in dimension k as the number of nodes is bounded by twice the number of points.

We won't go into the specifics of the derivation or evaluation of equation (1) as a balanced range AVL tree structure is structurally equivalent to a worst case (1-3) k-d Range DSL structure and we use the proof found in [Lamo95a] to justify our result.

An asymptotic approximation on equation (1) evaluates to $O(n\lg^{k-1}n)$ and this gives us $O(n\lg^{k-1}n)$ nodes in dimension k, as we set out to prove. ■

Theorem 2: The time, $I_S(n,k)$, required for a simple insertion in a range AVL Tree is $O(\lg^k n)$.

Proof:

The proof depends on the observation that the time required to insert a point into a 1-d range AVL tree depends on the time required to find the location where the new point is to be inserted as the operations of creating the new nodes (one interior, one leaf) and changing the necessary pointers to accomplish the insertion take only constant time. This gives us $I_S(n,1) = O(\lg n)$ as expected.

The proof that $I_S(n,k) = O(\lg^k n)$ follows from the proof that $Q_R(n,k) = O(\lg^k n + t)$. In dimension k we must insert the new point into a total of $O(\lg^{k-1} n)$ range AVL tree structures associated with the range AVL tree structures we inserted the new point into in dimension (k-1). As it takes $O(\lg n)$ time to insert the point into a single range AVL tree structure, we see that it takes $O(\lg^k n)$ time to insert the point into a k-d range AVL tree. ■

Lemma 2: The time, $R_I(n,k)$, to do necessary rebuilding for a single insertion is $O(n\lg^{k-2} n)$.

Proof:

In the worst case insertion, a rotation occurs at the root node of the tree in dimension 1. We then find that we have to do rebuilding of two (k-1) d range AVL tree structures in the dimension 2 that consist of roughly n/2 nodes each. The worst case insertion is illustrated in Figure 7. Using the trick of presorting found in [Bent80] we can rebuild a k dimensional range tree structure of n nodes in $O(n\lg^{k-1} n)$ time as we are working with a well defined data set which we can assume to be static for the duration of the necessary rebuilding. This tells us that a single insertion can cause at most $O(n\lg^{k-2} n)$ rebuilding to take place (as we must rebuild two k-1 dimensional range tree structures which contain n points combined), as desired. ■

Lemma 3: The time, $R_{SI}(n,k)$, required to do any necessary rebuilding over a sequence of n insertions is given by $O(n\lg^{k-1} n)$.

The worst case insertion is guaranteed to occur once, and only once, in a sequence of O(n) insertions when we start with n datapoints in the range AVL tree (see [Fred81] and [Will85]). Likewise, the second worst case insertion (when we have to do rebuilding at depth 2) is guaranteed to occur twice, and only twice in a sequence of O(n) insertions. In general, the ith worst case insertion is guaranteed to occur 2^{i-1} times, and only 2^{i-1} times, in a sequence of O(n) insertions.

The work involved in rebuilding over a sequence of n insertions is given by

$$\sum_{i=1}^{\lg n-1} 2i\left(\frac{n}{2^i}\lg^{k-2}\frac{n}{2^i}\right) \qquad (2)$$

and the summation is bounded by $O(n\lg^{k-1} n)$ (see [Lamo95a]). This completes our proof that the time, $R_{SI}(n,k)$, to do any necessary rebuilding over a sequence of n insertions is given by $O(n\lg^{k-1} n)$. ■

Theorem 3: The time, I(n,k) required to perform a normal insertion into a range AVL tree is $O(\lg^k n)$.

Proof:

This proof follows directly from Theorem 2 and Lemma 3. A simple insertion is bounded by $O(\lg^k n)$ and the time to do any necessary rebuilding is bounded by $O(\lg^{k-1} n)$ where we use an amortized rebuilding cost. Thus, an insertion requires $O(\lg^k n)$ time. ■

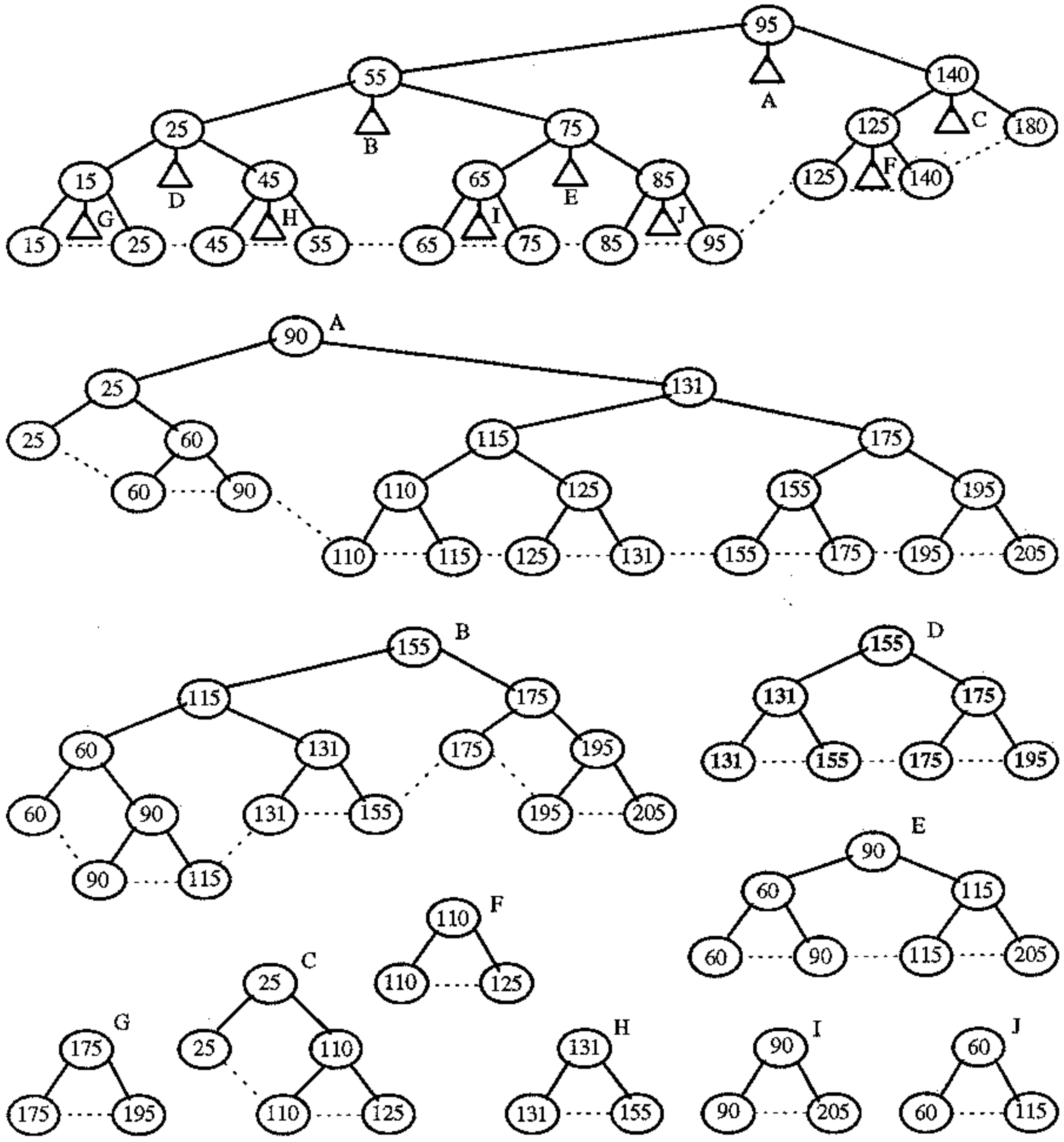


Figure 7(a). 2-d range AVL tree before the worst case insertion of (67,127).

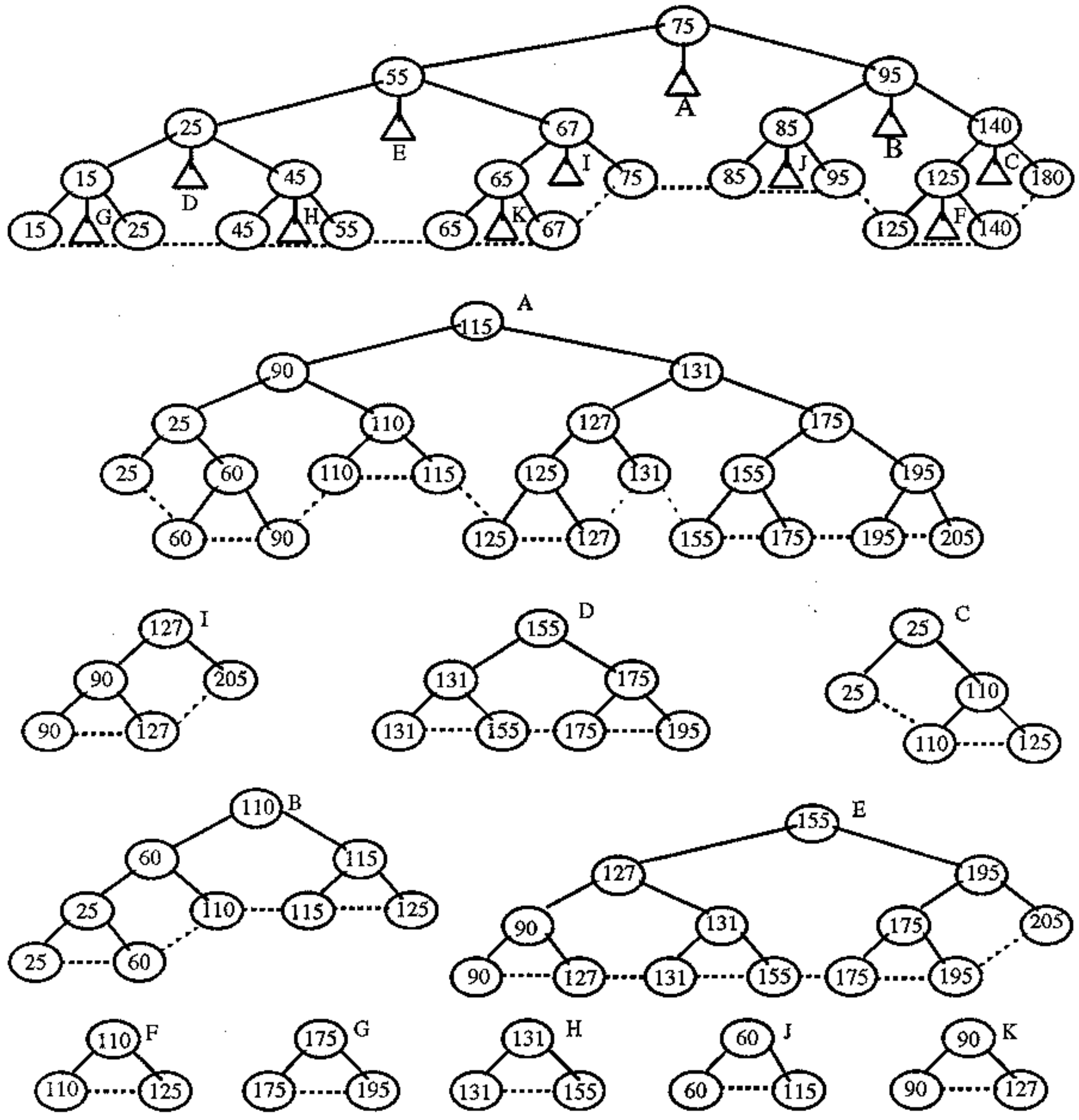


Figure 7(b). 2-d range AVL tree after the worst case insertion of (67,127).

Theorem 4: The time, $D_S(n,k)$, to perform a simple deletion of a point in a range AVL tree is $O(\lg^k n)$.

Proof:

The proof depends on the observation that the time required for deletion of a node in a 1-d range AVL tree depends on the time required to find the location of the point being deleted as the operations of removing the corresponding nodes (one internal and one leaf) and changing the necessary pointers require constant time. This gives us $D(n,1) = O(\lg n)$, as expected.

The proof that $D_S(n,k) = O(\lg^k n)$ follows from the proof that $Q_R(n,k) = O(\lg^k n + t)$. In dimension k we must remove the point from a total of $O(\lg^{k-1} n)$ range AVL tree structures. It takes $O(\lg n)$ time to remove the point from one range AVL tree structure and thus it takes $O(\lg^k n)$ time to remove the point from a k-d range AVL tree.■

Lemma 4: The time, $R_D(n,k)$, to do necessary rebuilding for a single deletion is $O(n\lg^{k-2} n)$.

In the worst case deletion, a rotation occurs at the root node of the tree in dimension 1. We then find that we have to do rebuilding of two (k-1)-d range AVL tree structures in the dimension (2) that consist of roughly n/2 nodes each. Using the trick of presorting found in [Bent80] we can rebuild a k dimensional range tree structure of n nodes in $O(n\lg^{k-1} n)$ time as we are working with a well defined data set which we can assume to be static for the duration of the necessary rebuilding. This tells us that a single deletion can cause at most $O(n\lg^{k-2} n)$ rebuilding to take place (as we must rebuild two k-1 dimensional range tree structures which contain n points combined), as desired. ■

Lemma 5: The time, $R_{SD}(n,k)$, required to do any necessary rebuilding over a sequence of n deletions is given by $O(n\lg^{k-1} n)$.

The worst case deletion is guaranteed to occur once, and only once, in a sequence of O(n) deletions when we start with 2n datapoints in the range AVL tree (see [Fred81] and [Will85]). Likewise, the second worst case deletion (when we have to do rebuilding at depth 2) is guaranteed to occur twice, and only twice in a sequence of O(n) deletions. In general, the ith worst case deletion is guaranteed to occur 2^{i-1} times in a sequence of O(n) deletions.

The work involved in rebuilding over a sequence of n deletions is given by

$$\sum_{i=1}^{\lg n-1} 2i\left(\frac{n}{2^i}\lg^{k-2}\frac{n}{2^i}\right) \qquad (2)$$

and we know that this summation is bounded by $O(n\lg^{k-1} n)$ from Lemma 3. This completes our proof that the time, $R_{SD}(n,k)$, to do any necessary rebuilding over a sequence of n deletions is given by $O(n\lg^{k-1} n)$. ■

Theorem 5: The time, $D(n,k)$ required to perform a normal deletion in a range AVL tree is $O(\lg^k n)$.

Proof:

This proof follows directly from Theorem 4 and Lemma 5. A simple deletion is bounded by $O(\lg^k n)$ and the time to do any necessary rebuilding is bounded by $O(\lg^{k-1} n)$ when we use an amortized rebuilding cost. Thus, a deletion requires $O(\lg^k n)$ time. ■

Theorem 6: The space, $S(n,k)$, required for storage of a k-d range AVL tree is $O(n\lg^{k-1} n)$.

Proof:

Each node in the k-d range AVL tree has three pointers and a constant number of data fields and thus requires $O(1)$ space so we need only determine the maximum number of nodes in the k-d range AVL tree to determine the space required. From Lemma 1, we know that dimension k requires $O(n\lg^{k-1} n)$ space. As we are performing a Big-Oh analysis, we see that this term is the dominant term in our expression for storage and we thus correctly determine that we require $O(n\lg^{k-1} n)$ space for storage, $S(n,k)$. ■

Theorem 7: The time, $P(n,k)$, to construct a k-d range AVL tree is $O(n\lg^k n)$.

We construct a dynamic k-d range AVL tree by insertion of one point at a time. We thus have to perform n insertions where each insertion is bounded by $O(\lg^k n)$ time when we used an amortized analysis. Thus, $P(n,k)$ is $O(n\lg^k n)$. ■

Theorem 8: The k-d range AVL tree is an optimally balanced data structure.

Proof

The balance cost, $P(n,k) * S(n,k) * I(n,k) * D(n,k) * Q_R(n,k) =$
$O(n \lg^k n) * O(n \lg^{k-1} n) * O(\lg^k n) * O(\lg^k n) * O(\lg^k n) =$
$O(\lg^k n * n \lg^{k-1} n * \lg^k n * \lg^k n * \lg^k n) = O(n^2 \lg^{5k-1} n) = O(n^2 \lg^k n)$
and this is known to be optimal from Lamo95b. ■

Theorem 9: Assuming an optimally balanced data structure, the k-d range AVL tree is optimal for k-d range search.

Proof:

From [Lamo95b] we know that an optimally balanced data structure is optimal for range search if it requires at most $O(\lg^k n + t)$ time for range search using a worst case analysis. Range search, $Q_R(n,k)$, in the k-d range AVL tree requires at most $O(\lg^k n + t)$ time in the worst case and therefore, since the structure is optimally balanced, the k-d range AVL tree is optimal for k-d range search. ■

8 CONCLUSIONS

This paper has provided us with an implementation of an optimally balanced data structure which is optimal for k-d range search on point data in a completely dynamic operating environment. It should provide a practical indexing mechanism to carry out range searches in a dynamic environment along with providing us with a baseline structure against which other structures, and their implementations, designed to handle multidimensional range queries on multidimensional data can be compared.

The techniques used to transform, or *map*, the AVL tree into a dynamic range tree provide insight into the similarities between different binary search tree structures and may prove useful in transforming and dynamizing other tree structures for more complicated tasks. The complete working Pascal code in the appendices provides us with a solid definition for our structure and illustrates a recursive technique which may be useful for defining and working with many other k-d data structures. The technique has already proved useful in the creation of the k-d Range DSL of [Lamo95a] and [Lamo95b].

9 REFERENCES

[Adel62] Adel'son-Vel'skii, G.M., and Landis, E.M., "An Algorithm for the Organization of Information", *Soviet Mathematics Doklady*, Vol. 3, 1962, pp. 1259 - 1262.

[Bent75] Bently, J.L., "Multidimensional Binary Search Trees Used For Associative Searching", *Communications of the ACM*, Vol. 18, No. 9, 1975, pp. 509 - 517.

[Bent79] Bently, J.L., and Friedman, Jerome H., "Data Structures for Range Searching", *ACM Computing Surverys*, Vol. 11, No. 4, December 1979, pp. 397 - 409.

[Bent80] Bently, J.L., "Multidimensional Divide-and-Conquer", *Communications of the ACM*, Vol. 23, No. 4, (April) 1980, pp. 214 - 229.

[Come79] Comer, D., "The Ubiquitous B-tree", *ACM Computing Surveys*, Vol. 11, No. 2, 1979, pp. 121 - 137.

[Edel81] Edelsbrunner, H., "A Note on Dynamic Range Searching", Bulletin of the EATCS, Number 15, October 1981, pp. 34 - 40.

[Fred81] Fredman, Michael L., "A Lower Bound on the Complexity of Orthogonal Range Queries", *J. ACM*, Vol. 28, No. 4, (October) 1981, pp. 696 - 705.

[Knut73] Knuth, D.E., *The Art of Computer Programming: Volume 3 Searching and Sorting*, Addison-Wesley, Reading, MA, 1973, pp. 550 - 555.

[Krus94] Kruse, Robert L., *Data Structures and Program Design*, Third Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Lamo95a] Lamoureux, Michael G., and Nickerson, Bradford G., "Deterministic Skip Lists For K-Dimensional Range Search", University of New Brunswick Technical Report TR95-098, Revision 1, November 1995.

[Lamo95b] Lamoureux, Michael G., and Nickerson, Bradford G., "A Deterministic Skip List For k-dimensional Range Search", [submitted for publication], October 1995.

[Luek78] Lueker, George S., "A Data Structure for Orthogonal Range Queries", Proceedings of the 19th Annual Symposium on Foundations of Computer Science, IEEE 78 CH1387-9 C, pp. 28 - 34.

[Luek82] Lueker, George S., and Willard Dan E., "A Data Structure for Dynamic Range Queries", *Information Processing Letters*, Vol. 15, No. 5, (December) 1982, pp. 209 - 213.

[McCr85] McCreight, Edward M., "Priority Search Trees", *SIAM J. Comput.*,Vol. 14, No.2, May 1985, pp 257 - 276.

[Same90] Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[Will85] Willard, Dan E. And Lueker, George S, "Adding Range Restriction Capability to Dynamic Data Structures", *J. ACM,* Vol. 32, No. 3, (July) 1985, pp. 597 - 617.

[Will85b] Willard, Dan E., "New Data Structures for Orthogonal Range Queries", *SIAM J. Comput.*, Vol. 14, No. 1, (February) 1985, pp. 232 -253.

APPENDIX 1

The 1-d dynamic range AVL tree

AVL trees are important as they achieve the goal that searches, insertions, and deletions in a tree with n nodes can all be achieved in time that is O(lg n), even in the worst case, with a minimal storage requirement of O(n). The height of an AVL tree can never exceed 1.44 lg n, and thus, even in the worst case, the behavior of an AVL tree is not far from that of a completely balanced binary search tree.

In a completely balanced binary search tree, the left and right subtrees of any node would have the same height. Although this goal is not always achievable in practice as it is difficult to maintain a uniformly balanced structure when we permit dynamic updates, we can always ensure that the heights of every left and right subtree never differ by more than 1 if we build our search tree carefully. We thus define an AVL tree as a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

The 1-d range tree is a height balanced binary search tree where the data values are stored only at the leaves which are threaded in sorted order by a doubly linked list. All internal nodes contain a key value that is used to facilitate the search. This key value can be set to be the maximum key value in the left subtree as this will maintain the search tree property when we use a ≤ in our search.

Based on the above definitions, we define our 1-d dynamic range AVL tree as an AVL tree where data values are stored only in the leaf nodes which are threaded in sorted order by a doubly linked list. To facilitate our insertion and deletion algorithms we make the further restriction that every internal node have 2 children and that it is assigned the maximum data value in its left subtree as its key value. This implies that our search tree property becomes: every node in the left subtree is less than or equal to the root which is less than every node in the right subtree, and that we must use ≤ when searching.

We define our node-structure as:

```
nodeptr = ^range_AVL_node;
range_AVL_node =
        record
          nt: nodetype;   {interior or leaf}
          key: keytype;   {usually integer}
          L, R: nodeptr;  {pointers to left and right children}
          bf: bf_type;    {balance factor, usually char or integer}
          p,n: nodeptr;   {threads leaf nodes, nil at interior nodes}
          da: nodeptr;    {direct ancestor; nil at interior nodes; connects a leaf node
                           to the interior node with its key value}
        end;
bf_type = (-1 or /, 0 or -, 1 or \);
```

In the following pages we describe the workings of the insertion, deletion, and range search procedures which are found in their entirety in Appendix 2.

Insertion of a node

We insert the new node into the range AVL tree by first using the usual binary search tree insertion algorithm which finds the location where the node is to be inserted by searching for the key of the node we are about to insert. The search terminates at the leaf level and once we reach a leaf node we have found the node that is to become the sibling of the node we are inserting.

We complete the insertion by making a copy of the leaf node which then becomes the right or left child of the node we end up at, depending on whether or not the new node has a key value that is less than or greater than the leaf node we end up at, which is converted into an interior node. After it is linked to the new interior node which is its parent, the new node is inserted into the sorted doubly linked list of leaf nodes using the standard linked list insertion procedure.

We also connect the newly inserted node to what we call its direct ancestor which is the interior node that "inherits" its key value. Note that, for a leaf that becomes a left child, its direct ancestor is simply its parent node at the time of insertion. If a new node is inserted as a right child, then it does not have a direct ancestor as a new node is only inserted as a right child when it has a key value that is larger than any key value currently in the tree. In this case, the leaf node that is its sibling, which used to contain the largest key value in the tree, is assigned the parent node as its direct ancestor while the new node acquires a nil link in its direct ancestor field.

The only special case that we must check for in our insertion procedure is that of a nil tree. As all other insertions take place at an existing leaf node, there are no other special cases. We also note that this data structure can support non-unique keys, and that they are always inserted as a left child, but that we usually use unique keys in practice.

Since an insertion always replaces a leaf node with an interior node which gets the old leaf node it replaced and the new node that is being inserted as its children, the height of the subtree that the new node is inserted into always increases. Thus, we call the procedures LeftBalance and RightBalance as appropriate, depending on whether the new node was inserted in the left subtree or right subtree, to update the affected balance factors and perform any necessary rotations needed to correct a height imbalance.

The procedures LeftBalance and RightBalance, like the procedures RotateLeft and RotateRight that they call, are identical to those used for the standard AVL tree and we refer the reader to [Krus94] for more detail and Appendix 2 for the Pascal code.

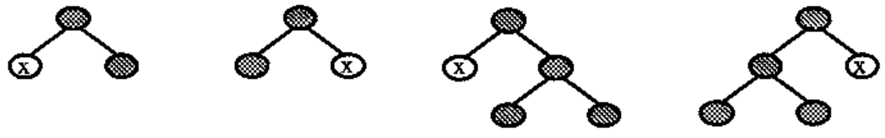
Deletion of a node

Deletion of a node from a range AVL tree requires the same basic ideas, including rotations, as deletion in a normal AVL tree but turns out to be much simpler as all the data values are stored only at the leaf level. This implies that we don't have the difficulties involved in normal AVL tree deletion where we have to recursively delete nodes from the tree when we are trying to remove a non-leaf node with two children.

To accomplish the deletion, we perform a recursive search for the node that we want to delete which is similar to the recursive search we perform in the normal AVL tree. The difference is that we must stop our search at the parent node as we must remove two nodes, an interior node (which is the parent node we terminate our search at) and the selected leaf node, to counterbalance the insertion procedure and maintain the search tree properties. (Every node must have two children and we must have 2n-1 nodes if we have n data points, i.e. n leaves and n-1 interior nodes.)

This implies that we have an additional two special cases to consider besides the four cases that we have to consider when we process a deletion. The special cases are that of an empty tree, from which no node can be deleted, and the tree with only one leaf node, which is replaced by the empty tree if the leaf node is the node we wish to delete.

The four cases we have to consider are as follows:



Case 1 Case 2 Case 3 Case 4

Case 1: The leaf node is a left child and its sibling is a leaf node.
In this case we simply replace the parent node with its right child and remove both the target node and the parent node from the tree. The direct ancestor is the parent node so we don't change a key value in any interior node and all we have to do to complete the deletion is remove the target node from the sorted linked list.

Case 2: The leaf node is a right child and its sibling is a leaf node.
Similar to case 1, but this time we replace the parent node with its left child and remove both the target node and the parent node from the tree. However, as the direct ancestor of the leaf node we are deleting is not the parent node, we must change the key value in the target node's direct ancestor to be the key value of its sibling. We also have to change the direct ancestor field of its sibling to point to the node that was the direct ancestor of the node we are deleting. Of course, if the node we are deleting happens to be that of the largest key value in the tree, then, since it has no direct ancestor, we don't have to change a key value in an interior node. However, we do have to change the direct ancestor field of its sibling to nil as its sibling becomes the node with the largest key value in the tree.

Case 3: The leaf node is a left child and its sibling is an interior node.
This case is almost identical to case one. Again we replace the parent node with its right child and remove both the node and the parent node from the tree. Again the direct ancestor is the parent node so all we have to do to complete the deletion is remove the target node from the sorted linked list.

Case 4: The leaf node is a right child and its sibling is an interior node.
This case is almost identical to case two. Again we replace the parent node with the its left child and remove both the target node and the parent node from the tree. As in case two, the direct ancestor, if it exists, is not the parent node and so we must change the key value in the direct ancestor to be the key value of the right child of the target node's sibling (which is an interior node). Also, we must change the direct ancestor field of the right child of the target node's sibling to point to the node that was the direct ancestor of the target node. Of course, as in case two, if the node we are deleting has the largest key value in the tree and, as such, has no direct ancestor then we don't have to change any key values and we simply replace the contents of the direct ancestor field of the right child of the target node's sibling to nil.

The BalanceL and BalanceR procedures which are called after every deletion are essentially the standard AVL tree BalanceL and BalanceR procedures which are used to change any affected balance factors and restore balance, if necessary, after a deletion. They are always called because a deletion, being complementary to an insertion, always decreases the height of a subtree.

Range Search in the 1-d range AVL tree

Range Search in the 1-d range AVL tree proceeds exactly like range search in the standard range tree. A range search for [L:H] is performed by searching the tree for the node with the smallest key $\geq$ L and then following the links until reaching a leaf node with a key that is greater than or equal to H. The procedure is quite succinct and can be found in its entirety in Appendix 2.

One advantage of the range AVL tree over the AVL tree is evident from the range search procedure. Unlike the AVL tree, the range AVL tree allows for an iterative range search procedure which allows for a slightly quicker range search as we lower the value of the constant hidden in the big-oh analysis when we replace a recursive procedure with an iterative one. Also, the above procedure is just as simple as the recursive one.

Summary of the 1-d range AVL tree

The 1-d dynamic range AVL tree is an alternative to the standard AVL tree and standard range tree which also allows for insertion, deletion, and member query in O(lg n) time complexity while requiring minimal storage space of O(n). It supports range searches in O(lg n + t) time, where t is the number of points found, and is as conceptually simple as the two structures that it combines. It also has the additional advantages that it is straightforward to extend it to handle k-d dimensional data and that it produces an optimally balanced structure which is optimal for range search in the class it belongs in. Generalized to k-d it provides an efficient alternative to B-trees as database indexes and the recursive algorithms it employs are generalizable to other data structures.

APPENDIX 2

The 1-d dynamic range AVL tree in Pascal

```
Program One_Dimensional_Range_AVL_Tree;

Type keytype = integer;  {usually integer, but not necessary}
     nodetype = (interior, leaf);
     bf_type = integer; {-1, 0, or 1} {LH, EQ, RH}
     nodeptr = ^range_AVL_node;
     range_AVL_node = record
                        nt: nodetype;  {type of node: interior or leaf}
                        key: keytype;  {usually integer}
                        L, R: nodeptr; {left and right children}
                        bf: bf_type;   {balance factor: int or char}
                        p,n: nodeptr;  {used to thread leaf nodes,
                                        nil at interior nodes}
                        da: nodeptr;   {direct ancestor: nil at interior
                                        nodes, points to interior node
                                        that inherits the leaf node's
                                        key value}
                     end;

Var T: nodeptr;          {Our 1-d dynamic range AVL tree}
    L, H: keytype;       {Search Range for Range Search}
    nn: nodeptr;         {new node to be inserted}
    taller: boolean;     {recursive control in Insert}
    reduced: boolean;    {recursive control in Delete}
    choice: char;        {for interactive testing purposes}
    key: keytype;        {a key value}

Procedure Copynode(T: nodeptr; Var nl: nodeptr);
{This procedure makes an exact duplicate of the leaf node, T, that is
about to be converted to an interior node by the Insert procedure.}

Begin  {Copynode}
  new(nl);                           {allocate the new node}

  nl^.nt:= T^.nt;                    {copy node type}
  nl^.da:= T^.da;                    {copy direct ancestor}
  nl^.key:= T^.key;                  {copy key value}
  nl^.L:= T^.L;  nl^.R:= T^.R;       {copy pointers to children}
  nl^.bf:= T^.bf;                    {copy balance factor}
  nl^.p:= T^.p;  nl^.n:= T^.n;       {copy pointers that maintain the thread}
End;  {Copynode}

Procedure RotateLeft(Var p: nodeptr);
{When the tree is unbalanced to the right, this procedure is called to
 perform a left rotation which is necessary to restore the balance.}

Var temp: nodeptr; {used as temp. var. for rotation}

Begin {RotateLeft}
  IF p = nil Then  {root node of rotation is nil, can't rotate}
    Begin {Error! Empty Subtree}
      writeln('Error in RotateLeft. Can''t rotate an empty subtree.');
      HALT;  {do not continue, critical error}
    End   {Error! Empty Subtree}
  Else {the root node of the rotation is not nil}
    IF p^.R = nil then  {a primary node of the rotation is nil}
      Begin {Error!  Empty subtree can't be root}
        writeln('Error in RotateLeft. Can''t make empty subtree root.');
        HALT; {do not continue, critical error}
      End   {Error!  Empty subtree can't be root}
    Else {primary nodes are not nil}
      IF ((p^.nt <> leaf) and (p^.R^.nt <> leaf)) Then
        Begin {valid rotation}
          {perform the rotation}
          temp:= p^.R;      {temp will be the root}
          p^.R:= temp^.L;   {left subtree of x --> right subtree of root}
          temp^.L:= p;      {root --> left subtree of x}
          p:= temp;         {root:= x}
        End  {valid rotation}
      ELSE {primary nodes are invalid as one or more is a leaf node}
        Begin {Error!  Can't rotate on a leaf node!}
          writeln('Invalid primary nodes in RotateLeft.');
          writeln('Critical Error! Terminating Program.');
          HALT;  {Critical Error!}
        End;  {Error!  Can't rotate on a leaf node!}
End;  {RotateLeft}

Procedure RotateRight(Var p: nodeptr);
{When the tree is unbalanced to the left, this procedure is called to
 perform a right rotation which is necessary to restore the balance.}

Var temp: nodeptr; {used as temp. var. for rotation}

Begin {RotateRight}
  IF p = nil Then
    Begin {Error! Empty Subtree}
      writeln('Error in RotateRight.  Can''t rotate an empty subtree.');
      HALT;  {do not continue, critical error}
    End   {Error! Empty Subtree}
  Else {root node of rotation is not nil}
    IF p^.L = nil Then  {primary node of rotation is nil}
      Begin {Error! Empty subtree can't become root}
        writeln('Error in RotateRight Can''t make empty subtree root.');
        HALT;  {do not continue, critical error}
      End   {Error! Empty subtree can't become root}
    Else {primary nodes of rotation are not nil}
      IF ((p^.nt <> leaf) and (p^.L^.nt <> leaf)) Then
        Begin {valid rotation}
          {perform the rotation}
          temp:= p^.L;      {temp will be the root}
          p^.L:= temp^.R;   {right subtree of x --> left subtree of root}
          temp^.R:= p;      {root --> right subtree of x}
          p:= temp;         {root:= x}
        End  {valid rotation}
      ELSE  {primary nodes are invalid as one or more is a leaf node}
        Begin {Error! Can't rotate on a leaf node!}
          writeln('Invalid primary nodes in RotateRight.');
          writeln('Critical Error! Terminating Program.');
          HALT;  {Critical Error!}
        End;  {Error! Can't rotate on a leaf node!}
End;  {RotateRight}
```

```
Procedure RightBalance(VAR T: nodeptr; Var taller: boolean);
{This procedure is called to correct an imbalance in the right subtree.}
{i.e. the tree is right high (BF = 2}

Var x,          {pointer to right subtree of T}
    w: nodeptr; {pointer to left subtree of x}

Begin {RightBalance}
  x:= T^.R;

  Case x^.bf of
     1: Begin {RH} {a single left rotation corrects this situation}
          T^.bf:=  0;     {after rotation, this node is balanced}
          x^.bf:=  0;     {after rotation, this node is balanced}
          RotateLeft(T);  {perform single left rotation}
          taller:= false; {subtree hasn't increased in height}
        End;  {RH} {a single left rotation corrects the situation}
     0: Begin {EH}  {By def'n of AVL trees, this case doesn't arise.}
          writeln('Error in RightBalance.  Condition of EH.');
          HALT; {do not continue, critical error}
        End;  {EH}  {By def'n of AVL trees, this case doesn't arise.}
    -1: Begin {LH} {we need a double rotation to correct the situation}
          w:= x^.L; {will become the root}
          Case w^.bf of   {w determines balance factors of x and T}
             0: Begin T^.bf:=  0;  x^.bf:=  0; End;
            -1: Begin T^.bf:=  0;  x^.bf:=  1; End;
             1: Begin T^.bf:= -1;  x^.bf:=  0; End;
          End; {Case w^.bf of}
          w^.bf:= 0;      {w is always balanced after rotation}
          RotateRight(x); {first rotate right on x, w replaces x}
          T^.R:= x;       {reconnect x to right child of T}
          RotateLeft(T);  {now rotate left on T}
          taller:= false; {the subtree has not increased in height}
        End;  {LH} {we need a double rotation to correct the situation}
  End; {Case x^.bf of}
End;  {RightBalance}

Procedure LeftBalance(Var T: nodeptr; Var taller: boolean);
{This procedure is called to correct an imbalance in the left subtree.}
{i.e. the tree is left high (BF = -2)}

Var x,          {pointer to left subtree of T}
    w: nodeptr; {pointer to right subtree of x}

Begin {LeftBalance}

  x:= T^.L;
  Case x^.bf of
    -1: Begin {LH} {a single right rotation corrects the situation}
          T^.bf:=  0;  {after rotation, this node is balanced}
          x^.bf:=  0;  {after rotation, this node is balanced}
          RotateRight(T);  {perform single right rotation}
          taller:= false;  {subtree hasn't increased in height}
        End;  {LH} {a single right rotation corrects the situation}
     0: Begin {EH} {By def'n of AVL trees, this case doesn't arise.}
          writeln('Error in LeftBalance.  Condtion of EH.');
          HALT;  {do not continue, critical error}
        End;  {EH} {By def'n of AVL trees, this case doesn't arise.}
     1: Begin {RH} {we need a double rotation to correct the situation}
          w:= x^.R; {will become the root}
          Case w^.bf of {w determines balance factors of x and T}
             0: Begin T^.bf:=  0;  x^.bf:=  0; End;
             1: Begin T^.bf:=  0;  x^.bf:= -1; End;
            -1: Begin T^.bf:=  1;  x^.bf:=  0; End;
          End;  {Case w^.bf of}
          w^.bf:= 0; {w is balanced after the rotation}
          RotateLeft(x);  {first rotate left on x, w replaces x}
          T^.L:= x;       {reconnect x to left child of T}
          RotateRight(T); {now rotate right on T}
          taller:= false;  {the subtree has not increased in height}
        End;  {RH} {we need a double rotation to correct the situation}
    End;  {Case x^.bf of}

End;  {LeftBalance}

Procedure Insert(Var T: nodeptr; nn: nodeptr; Var taller: boolean);
{This procedure inserts the node nn into the dynamic range AVL tree T.
 Taller is true if the height of tree increases and is false otherwise}

Var tallersubtree: boolean;  {has height of the subtree increased?}
       nl: nodeptr;          {we need a new leaf during insertion as the
                              last leaf node found in our search is
                              converted to an interior node}

Begin {Insert}
  tallersubtree:= false; {only true when height of subtree increases}

IF T = nil Then  {we are inserting the first node}
  T:= nn
ELSE {we are not inserting the first node}
  Begin {tree is not empty}
    IF T^.nt = leaf Then {insertion occurs only at leaf level}
      Begin {we have reached the leaf level, insert}
        taller:= true; {insertion increases height}
        Copynode(T, nl);  {before converting T to interior node, copy}
        T^.nt:= interior;    {T is now interior node}
        {whatever points to T must now point to nl}
        IF T^.p <> nil Then T^.p^.n:= nl;
        IF T^.n <> nil Then T^.n^.p:= nl;
        {Insert nn as left or right child of T as appropriate}
        IF nn^.key <= nl^.key Then {new node is left child}
          Begin  {Insert new node as left child}
           {Direct ancestor is new interior node T}
            nn^.da:= T;
            T^.da:= nil; {interior node has no direct ancestor}
            T^.L:= nn;  T^.R:= nl;  {connect leaves to interior node T}
            T^.p:= nil; T^.n:= nil; {T is interior, remove from list}
            T^.key:= nn^.key;        {T's key = max. key left subtree}
            {insert new leaf into linked list}
            IF nl^.p <> nil Then {new node not first node in list}
              Begin {make connection between new node and previous}
                nl^.p^.n:= nn;   nn^.p:= nl^.p;
              End   {make connection between new node and previous}
            ELSE  {new node is first node in list}
              nn^.p:= nil;
            {connect new node to sibling}
            nn^.n:= nl;
            nl^.p:= nn;
          End    {Insert new node as left child}
        ELSE {new leaf is right child, old leaf is left child}
```

```
          Begin  {Insert new node as right child}
            {New node is only right child when its key is larger than
             all keys currently in the tree.  In this instance, new node
             does not have a direct ancestor and its sibling gets the
             parent node as its direct ancestor}
            nn^.da:= nil; {new node has no direct ancestor}
            nl^.da:= T;   {sibling gets parent as direct ancestor}
            T^.da:= nil;  {interior node has no direct ancestor}
            T^.L:= nl;  T^.R:= nn;  {connect leaves to interior node T}
            T^.p:= nil; T^.n:= nil; {T is interior, remove from list}
            T^.key:= nl^.key;       {T's key = max. key left subtree}
            {insert new leaf into linked list}
            IF nl^.n <> nil Then {new node is not last in list}
              {Note that the next four lines should never fire!}
              Begin {make connection between new node and next}
                nl^.n^.p:= nn;
                nn^.n:= nl^.n;
              End   {make connection between new node and next}
            ELSE {new node is last node in list}
              nn^.n:= nil;
              {connect new node to sibling}
              nn^.p:= nl;
              nl^.n:= nn;
          End     {Insert new node as right child}
      End    {we have reached the leaf level, insert}
    ELSE {we are not yet at leaf level}
      Begin {trace down to leaf level}
        IF nn^.key <= T^.key Then
          Begin {Insert into left subtree}
            Insert(T^.L, nn, tallersubtree);
            IF tallersubtree Then
            Case T^.bf of
              -1: LeftBalance(T, taller);
               0: Begin  T^.bf:= -1;  taller:= true;  End;
               1: Begin  T^.bf:= 0;  taller:= false;  End;
            End {Case T^.bf}
          ELSE
            taller:= false;
          End    {Insert into left subtree}
        ELSE {nn^.key > T^.key}
          Begin {Insert into right subtree}
            Insert(T^.R, nn, tallersubtree);
            IF tallersubtree Then
              Case T^.bf of
                -1: Begin  T^.bf:= 0;  taller:= false;  End;
                 0: Begin  T^.bf:= 1;  taller:= true;   End;
                 1: RightBalance(T, taller);
              End {Case T^.bf}
            ELSE
              taller:= false;
          End    {Insert into right subtree}
      End;   {trace down to leaf level}
  End;   {tree is not empty}
End; {Insert}

Procedure BalanceL(Var T: nodeptr; Var reduced: boolean);
{We have just reduced the height in the left subtree by 1.  If the
balance criterion has been violated then we must restore it.}

Var x,w: nodeptr;
```

```
Begin {BalanceL}
  Case T^.bf of
    -1: T^.bf:= 0;  {two subtrees are equal height} {reduced stays true}
     0: Begin  T^.bf:= 1;  Reduced:= false;  end;
     1: Begin {must rotate T to the left}
          IF ((T^.R^.bf = 1) or (T^.R^.bf = 0)) Then
            Begin {perform single rotation}
              IF T^.R^.bf = 1 Then
                Begin {Perform a normal AVL tree single left rotation}
                  {Reduced stays true}
                  T^.bf:= 0;  T^.R^.bf:= 0;
                End    {Perform a normal AVL tree single left rotation}
              ELSE {T^.R^.bf = 0}
                Begin {Single left rotation but change balance factors}
                  Reduced:= false;
                  T^.bf:= 1;  T^.R^.bf:= -1;
                End;  {Single left rotation but change balance factors}
              RotateLeft(T);
            End    {perform single rotation}
          ELSE {T^.R^.bf = -1}
            Begin {perform double rotation}
              {Reduced stays true}
              x:= T^.R;
              {procedure RightBalance double rotation case}
              w:= x^.L;
             Case w^.bf of
                0: Begin  T^.bf:= 0;  x^.bf:= 0;  End;
               -1: Begin  T^.bf:= 0;  x^.bf:= 1;  End;
                1: Begin  T^.bf:= -1;  x^.bf:= 0;  End;
              End;  {Case}
              w^.bf:= 0;
              RotateRight(x);
              T^.R:= x;
              RotateLeft(T);
            End    {perform double rotation}
        End;  {must rotate T to the left}
  End;  {Case}
End;   {BalanceL}

Procedure BalanceR(Var T: nodeptr; Var reduced: boolean);
{We have just reduced the height in the right subtree by 1.  If the
balance criterion has been violated then we must restore it.}

Var x,w: nodeptr;

Begin {BalanceR}

  Case T^.bf of
     1: T^.bf:= 0;  {two subtrees equal height.}  {Reduced stays true}
     0: Begin  T^.bf:= -1;  Reduced:= false;  End;
    -1: Begin {must rotate T to the right}
          IF ((T^.L^.bf = -1) or (T^.L^.bf = 0)) Then
            Begin {perform single rotation}
              IF T^.L^.bf = -1 Then
                Begin {Perform a normal AVL tree single right rotation}
                  {Reduced stays true}
                  T^.bf:= 0;  T^.L^.bf:= 0;
                End {Perform a normal AVL tree rotation to the right}
              ELSE
```

```
                Begin {Single right rotation but change balance factors}
                  Reduced:= false;
                  T^.bf:= -1;  T^.L^.bf:= 1;
               End;  {Single right rotation but change balance factors}
              RotateRight(T);
            End    {perform single rotation}
          ELSE {T^.L^.bf = 1}
            Begin {perform double rotation}
              {Reduced stays true}
              x:= T^.L;
              {procedure LeftBalance double rotation case}
              w:= x^.R;
              Case w^.bf of
                0: Begin  T^.bf:= 0;  x^.bf:= 0;  End;
                1: Begin  T^.bf:= 0;  x^.bf:= -1;  End;
               -1: Begin  T^.bf:= 1;  x^.bf:= 0;  End;
              End; {Case}
              w^.bf:= 0;
              RotateLeft(x);
              T^.L:= x;
              RotateRight(T);
            End    {perform double rotation}
        End;  {must rotation T to the right}
  End; {Case}

End;   {BalanceR}

Procedure Delete(x: keytype; Var T: nodeptr; Var reduced: boolean);
{Delete node from T that contains key x.  If subtree height decreases,
 reduced:= true.}

Var Q: nodeptr;  {temporary pointer}

Begin {Delete}

  IF T <> nil Then
    IF T^.nt = leaf Then {one datapoint left in tree}
      IF T^.key = x Then
        T:= nil  {tree is now empty}
      ELSE
        writeln('key is not in tree')
    ELSE  {more than one datapoint left in tree}
      IF ((T^.L^.nt = leaf) and (T^.R^.nt = leaf)) Then
        {two datapoints in tree}
        IF T^.L^.key = x Then
          Begin {delete left child}
            {when deleting, we must change key value in direct ancestor}
            {however, in this case direct ancestor is parent node and it
             is replaced by right child so we do nothing}
            {we don't need to change direct ancestor in leaf that moves
             up as it doesn't change}
            T^.R^.p:= T^.L^.p; {remove T^.L from list}
            IF T^.R^.p <> nil Then
              T^.R^.p^.n:= T^.R;
            reduced:= true; {deletion always reduces height of subtree}
            Q:= T;    T:= T^.R;
            Dispose(Q^.L);  Dispose(Q);
          End    {delete left child}
        ELSE
          IF T^.R^.key = x Then
          Begin {delete right child}
            {when deleting, we change key value in direct ancestor}
            IF T^.R^.da <> nil Then
              Begin
                T^.R^.da^.key:= T^.L^.key;
                {we must change direct ancestor in leaf that moves up}
                T^.L^.da:= T^.R^.da;
              End
            ELSE {we are deleting last node;it has no direct ancestor}
              T^.L^.da:= nil;
            T^.L^.n:= T^.R^.n;  {remove T^.R from list}
            IF T^.L^.n <> nil Then
              T^.L^.n^.p:= T^.L;
            reduced:= true; {deletion reduces height of subtree}
            Q:= T;    T:= T^.L;
            Dispose(Q^.R);  Dispose(Q);
          End    {delete right child}
        ELSE
          writeln('key is not in tree')
    ELSE  {more than two datapoints in tree}
      IF x <= T^.key Then {key, if present, is in left subtree}
        IF T^.L^.nt = leaf Then {three keys in tree}
          IF T^.L^.key = x Then {process deletion}
            Begin {delete left child}
              {when deleting, change key value in direct ancestor}
              {again, direct ancestor is parent node which gets
               replaced by right child so again we do nothing}
              {we don't need to change direct ancestor as the node
               that moves up is an interior}
              T^.R^.L^.p:= T^.L^.p; {remove T from list}
              IF T^.R^.L^.p <> nil Then
                T^.R^.L^.p^.n:= T^.R^.L;
              reduced:= true; {deletion reduces height of subtree}
              Q:= T;  T:= T^.R;
              Dispose(Q^.L);  Dispose(Q);
            End    {delete left child}
          ELSE {key not in tree}
            writeln('key is not in tree')
        ELSE {more than three keys in left subtree, normal deletion}
          Begin {normal deletion in left subtree}
            Delete(x, T^.L, reduced);
            IF reduced Then BalanceL(T, reduced);
          End    {normal deletion in left subtree}
      ELSE {key, if present, is in right subtree}
        IF T^.R^.nt = leaf Then {three keys in tree}
          IF T^.R^.key = x Then {process deletion}
            Begin {delete right child}
              {when deleting, we change key value in direct ancestor}
              IF T^.R^.da <> nil Then
                Begin
                  T^.R^.da^.key:= T^.L^.R^.key;
                  {wechange direct ancestor in leaf that moves up}
                  T^.L^.R^.da:= T^.R^.da;
                End
              ELSE
                T^.L^.R^.da:= nil;
              T^.L^.R^.n:= T^.R^.n; {remove key from list}
              IF T^.L^.R^.n <> nil Then
                T^.L^.R^.n^.p:= T^.L^.R;
              Q:= T;  T:= T^.L;
```

```
                 reduced:= true;  {deletion reduces height of subtree}
                 Dispose(Q^.R);  Dispose(Q);
               End    {delete right child}
             ELSE {key is not in tree}
               writeln('key is not in tree')
           ELSE {more than three keys in right subtree, normal deletion}
             Begin {normal deletion in right subtree}
               Delete(x, T^.R, reduced);
               IF reduced Then BalanceR(T, reduced);
             End    {normal deletion in right subtree}
  ELSE {T = nil}
    writeln('Empty Tree.  Deletion is not possible.');

End;   {Delete}

Procedure RangeSearch(T: nodeptr; L,H: keytype);
{We are searching for all keys between L and H inclusive in the range
AVL tree T.}

Var Q: nodeptr;

  Function InRange(key, R: keytype): boolean;
  {key is automatically >= L}

  Begin  {InRange}
    IF key <= R then
      InRange:= true
    ELSE
      InRange:= false;
  End;   {InRange}

  Procedure Report(key: keytype);

  Begin  {Report}
    writeln(key);
  End;   {Report}

Begin {RangeSearch}
  Q:= T; {start at the root node}
  IF Q <> nil Then {we can only search a tree that is not empty}
    Begin {search not empty tree}
      {find smallest key in range}
      While Q^.nt <> leaf Do
        IF Q^.key < L Then
          Q:= Q^.R
        ELSE {Q^.key >= L}
          Q:= Q^.L;
      {We only go left when Q^.key is >= L. Therefore, at the leaf
       level, we either have Q^.key>=L or an empty query as Q^.key = max
       value in left subtree for an interior node.}
      IF not((Q^.key < L) or (Q^.key > H)) Then
        While ((Q <> nil) and InRange(Q^.key, H)) Do
          Begin
            Report(Q^.key);
            Q:= Q^.n;
          End
      Else {otherwise no keys are in range}
        writeln('No keys satisfy query.');
    End;  {search not empty tree}
End;   {RangeSearch}
```

```
Procedure Getnode(Var nn: nodeptr; key: keytype);

Begin {Getnode}
  new(nn); {allocate the new node}
  {set all data values as appropriate for a not yet inserted leaf}
  nn^.da:= nil;
  nn^.nt:= leaf;
  nn^.key:= key;
  nn^.L:= nil;   nn^.R:= nil;
  nn^.p:= nil;   nn^.n:= nil;
  nn^.bf:= 0;
End;  {Getnode}

Procedure DisplayT(Var dfile: text; T: nodeptr);

Begin {DisplayT}
  {Inorder Traversal is used to display tree}
  IF T <> nil Then
    Begin {T}
      DisplayT(dfile, T^.L);
        write(dfile, 'key: ', T^.key:3, ' bf: ', T^.bf:2);
        IF T^.L <> nil Then
          write(dfile, ' LC: ', T^.L^.key:3)
        ELSE
          write(dfile, ' LC: nil');
        IF T^.R <> nil Then
          write(dfile, ' RC: ', T^.R^.key:3)
        ELSE
          write(dfile, ' RC: nil');
        IF T^.p <> nil Then
          write(dfile, ' p: ', T^.p^.key:3)
        ELSE
          write(dfile, ' p: nil');
        IF T^.n <> nil Then
          write(dfile, ' n: ', T^.n^.key:3)
        ELSE
          write(dfile, ' n: nil');
        IF T^.da <> nil Then
          writeln(dfile, ' da: ', T^.da^.key:3)
        ELSE
          writeln(dfile);
      DisplayT(dfile, T^.R);
    End; {T}
End;  {DisplayT}

Procedure DisplayTree(T: nodeptr);

Const dtfilename = 'avlrt.dat';

Var dfile: text;
    Q: nodeptr;

Begin {DisplayTree}
  assign (dfile, dtfilename);
  rewrite(dfile);
  IF T <> nil Then
   Begin
    DisplayT(dfile, T);
    write(dfile, 'linked list: ');
    Q:= T;
    While Q^.nt <> leaf Do
      Q:= Q^.L;
    While Q <> nil Do
      Begin
        write(dfile, Q^.key:3);
        Q:= Q^.n;
      End;
    writeln(dfile);
    write(dfile, 'linked list: ');
    Q:= T;
    While Q^.nt <> leaf Do
      Q:= Q^.R;
    While Q <> nil Do
      Begin
        write(dfile, Q^.key:3);
        Q:= Q^.p;
      End;
    writeln(dfile);
   End
  ELSE
    writeln(dfile, 'Empty Tree.');

  close  (dfile);
End;  {DisplayTree}

Procedure OutputFile;

Const dtfilename = 'avlrt.dat'; {from DisplayTree}

Var dfile: text;
    aline: string[78];

Begin {Outputfile}
  assign(dfile, dtfilename);
  reset (dfile);
    While not(eof(dfile)) Do
      Begin
        readln(dfile, aline);
        writeln(aline);
      End;
  close (dfile);
End;  {Outputfile}

Procedure ShowMenu;

Begin {Menu}
  writeln('Please enter:');
  writeln('I: for Insertion    of a key');
  writeln('D: for Deletion     of a key');
  writeln('S: to  Display      the current tree');
  writeln('R: to  Range Search the current tree');
End; {Menu}
```

```
Procedure GetChoice(Var choice: char);

Begin {Getchoice}
  writeln('Please enter character of your choice');
  readln(choice);
End;  {Getchoice}

Procedure GetKey(Var key: keytype);

Begin {GetKey}
  writeln('Please enter your chosen key.');
  readln(key);
End;  {GetKey}

Procedure GetRange(Var L,H: keytype);

Begin {GetRange}
  writeln('Please enter range in the form L H:');
  readln(L,H);
End;  {GetRange}

Begin {Main}
{Initiailize tree}
  T:= nil;

{Test Driver}
  Repeat
    ShowMenu;
    GetChoice(choice);
    Case choice of
      'I','i': Begin
                 Getkey(key);
                 Getnode(nn, key);
                 taller:= false;
                 Insert(T, nn, taller);
               End;
      'S','s': Begin
                 DisplayTree(T);
                 OutputFile(T);
               End;
      'R','r': Begin
                 Getrange(L,H);
                 RangeSearch(T, L, H);
               End;
      'D','d': Begin
                 Getkey(key);
                 reduced:= false;
                 Delete(key, T, reduced);
               End;
    End; {Case}
  Until (choice IN ['Q','q']);
End.  {Main}
```

APPENDIX 3

The k-d dynamic range AVL tree

The k-d dynamic range AVL tree is a dynamic k-dimensional range tree data structure which is optimal ($O(\lg^k n + t)$ for t points in range), using a worst case analysis, for answering k-dimensional range queries on k-dimensional data when we assume that the underlying structure is optimally balanced, which the k-d dynamic range AVL tree is. The cost functions for the k-d dynamic range AVL tree are those of a k-d range tree, and, like the range tree, the k-d dynamic range AVL tree uses the technique of multidimensional divide and conquer [Bent80].

The structure is important as it provides us with a baseline structure against which other structures, and their implementations, designed to handle multidimensional range queries on multidimensional data can be compared. It is straightforward to implement and the complete code is provided in appendix 4. The k-d implementation can be achieved by modifying the 1-d implementation so that the appropriate procedures have a second level of recursion built in which traces through the dimensions.

The node structure is as follows and is essentially the same as the 1-d structure with the addition of a field that keeps a pointer to the structure in the next dimension and a field which, for a leaf node, stores the datapoint the key value is from.

```
range_AVL_node = record
                  nt: nodetype;   {interior or leaf}
                  key: keytype;   {usually integer}
                  {key is coordinate of a dp in current dimension}
                  dp: coordinate;{actual datapoint}
                  L, R: nodeptr; {left and right children}
                  bf: bf_type;    {balance factor: char or integer}
                  p,n: nodeptr;   {used to thread leaf nodes,
                                   nil at interior nodes}
                  da: nodeptr;    {direct ancestor: nil at interior
                                   nodes, points to interior node that
                                   inherits the leaf node's key value}
                  nd: nodeptr;    {pointer to nextdim substructure}
                      end;
```

In the following pages we outline the insertion, deletion, and range search procedures, focusing on the modifications that take the procedures from 1-d to k-d. We also describe the technique used for partial rebuilding and the modifications needed by any of the auxiliary procedures called by insertion, deletion, and range search. The complete code can be found in Appendix 4.

Insertion of a node

Insertion proceeds similar to that of the 1-d range AVL tree with the only modification being that for each interior node we visit on the way to the leaf level, we also insert the datapoint into the structure in the next dimension rooted at that node.

We find that we also have to modify the LeftBalance, RightBalance, RotateLeft, and RotateRight procedures. As described in section 6, when we execute a (single) rotation in a k-d range AVL tree, we must also exchange pointers to the tree structures in the next dimension. This implies that we have to rebuild one tree structure in the next dimension for a single rotation and two tree structures in the next dimension for a double rotation (as it is executed as two single rotations).

In the RotateLeft and RotateRight procedures we just have to add three lines of code to swap the nextdim pointers. In the LeftBalance and RightBalance procedures, we add a line of code that calls a procedure to rebuild any tree structure which has become unbalanced by a rotation.

Deletion of a node

Deletion proceeds similar to that of the 1-d range AVL tree with the only modification being that for each interior node we visit on the way to the leaf level, we also remove the datapoint from the structure in the next dimension at that node.

We find that we also have to modify the BalanceL and BalanceR procedures to call the necessary procedure to rebuild a tree structure in the next dimension after a rotation which invalidates the structure.

Range Search in the k-d range AVL tree

Range search proceeds similar to that of the k-d range tree. We progress down the tree in dimension k looking for the common ancestor node, Q_k. Once we find it, we iterate down the tree searching for the keys L_x' and H_x'. For each node, I, on the search path down to L_x' whose left child is also on the search path, we search the range AVL tree in the next dimension rooted at I's right child. For each node, J, on the search path down to H_x' whose right child is also on the search path, we search the range AVL tree in the next dimension rooted at J's left child. When we reach L_x' and H_x', we check to see if they are in range. This completes the range search algorithm.

Rebuilding a k-d range AVL tree

To rebuild a k-d range AVL tree structure, we use the trick of presorting as described in [Bent80] and rebuild the tree structure from the bottom up in a given dimension. We rebuild all of dimension k before rebuilding dimension k+1. This rebuilding can be done in $O(n\lg^{k-1}n)$ time and we describe it below.

Given a sorted list, we can rebuild the first dimension in O(n) time. To rebuild the structures in dimension 2, we rebuild the structures at the first non-leaf level first as these can be built in linear time. (They all contain two datapoints and two datapoints can be sorted with 1 comparison.) We then rebuild the next non-leaf level which, like successive levels, can be built in linear time as well. At a given node, we rebuild the structure in the next dimension by merging copies of the sorted lists found in the structures in the next dimension which are attached to the children nodes. As a merge of two sorted lists can be done in linear time, we can rebuild the structures in the next dimension attached to any level in linear time. This says we can rebuild a two-dimensional structure in O(n lg n) time as there are only O(lg n) levels in a range AVL tree.

The generalization of this technique to k-dimensions extends analogously to the extension of the technique from the first to the second dimension and it is straightforward to see that this procedure allows us to rebuild a k-d range AVL tree in $O(n\lg^{k-1}n)$ time.

The details of the rebuilding procedure are to be found in the complete Pascal code provided in Appendix 4.

Summary of the k-d range AVL tree

The k-d range AVL tree is an optimally balanced data structure which is optimal for k-d range search on point data in a completely dynamic operating environment. It should provide a practical indexing mechanism to carry out range searches in a dynamic environment along with providing us with a baseline structure against with other structures, and their implementations, designed to handle multidimensional range queries on multidimensional data can be compared.

The techniques used to transform, or *map*, the AVL tree into a dynamic range tree provide insight into the similarities between different binary search tree structures and may prove useful in transforming and dynamizing other tree structures for more complicated tasks. The transformation of the 1-d range AVL tree to the k-d range AVL tree illustrates a recursive technique which may be useful for defining and working with other k-d data structures. The technique has already proved useful in the creation of the k-d Range DSL of [Lamo95a] and [Lamo95b] and should be readily applicable to other structures.

APPENDIX 4

The k-d dynamic range AVL tree in Pascal

```
Program Dynamic_KD_Range_AVL_Tree;
{This program implements a dynamic range tree with an AVL tree.}
{we use DRT as short for k-d dynamic range tree}

{$M 65520, 65536, 655360}
{max stack, min heap, max heap}

Const dpointfilei = 'dpoints3.dat'; {contains datapoints for testing}
      dpointfiled = 'dpointsc.dat'; {contains datapoints for testing}

      outfile     = 'DRT.dat';      {datafile containing structure}

      maxdim      = 3;              {max num dimensions of structure}

Type keytype = integer;
     coordinate = array [1..maxdim] of keytype;
     nodetype = (interior, leaf);
     bf_type = integer; {-1, 0, or 1}
     nodeptr = ^range_AVL_node;
     range_AVL_node = record
                  nt: nodetype;  {interior or leaf}
                  key: keytype;  {usually integer}
                  dp: coordinate;{actual datapoint}
                  L, R: nodeptr; {left and right children}
                  bf: bf_type;   {balance factor: char or int}
                  p,n: nodeptr;  {used to thread leaf nodes,
                                  nil at interior nodes}
                  da: nodeptr;   {direct ancestor: nil at interior
                                  nodes, points to interior node that
                                  inherits the leaf node's key value}
                  nd: nodeptr;   {pointer to nextdim substructure}
                      end;

     listptr  = ^listnode; {ptr to a list node}
     listnode = record     {holds DRT nodes to be processed}
                  DRTnode: nodeptr;  {ptr to a DRT node}
                  next: listptr;     {next node in list}
                end;

     dfiletype = file of coordinate; {datafile type of datapoint file}

Const sv: coordinate = (0,0,0); {sentinel value stored in interior node}

Var T: nodeptr;           {Tree}
    numpoints: integer;   {how many datapoints for test run?}
    dimensions: integer;  {how many dimensions for test run?}

Procedure Copynode(T: nodeptr; Var nl: nodeptr; dim: integer;
                   flag: char);
{This procedure makes a copy of the leaf node, T, that is about to be
converted to an interior node by the Insert procedure with the exception
that the key is set to be the [dim]th coordinate of the datapoint,
T^.dp.  If the flag is set to 'y', all other fields are copied exactly,
otherwise, the previous (p) and next (n) pointers are set to nil.}
```

```
Begin {Copynode}
  new(nl);                       {allocate the new node}

  nl^.nt:= T^.nt;                {copy node type}
  nl^.da:= T^.da;                {copy direct ancestor}
  nl^.key:= T^.dp[dim];          {copy key value}
  nl^.dp:= T^.dp;                {copy datapoint}
  nl^.L:= T^.L;  nl^.R:= T^.R;   {copy pointers to children}
  nl^.bf:= T^.bf;                {copy balance factor}
  nl^.nd:= T^.nd;                {copy nextdim pointer}
  IF ((flag = 'y') or (flag = 'Y')) Then
    Begin
      nl^.p:= T^.p;  nl^.n:= T^.n;  {copy pointers to maintain thread}
    End
  ELSE
    Begin
      nl^.p:= nil;  nl^.n:= nil;    {set the links to nil}
    End;
End;  {Copynode}

Procedure Disposeof(Var T: nodeptr);
{When we have determined that a range tree in the next dimension must be
 rebuilt, we must first dispose of the invalid range tree and return all
 avalaible memory to the system}

Begin {Disposeof}
{We dispose using a modified postorder traversal where we remove a tree
 in the next dimension attached to a node before we remove the node}
  IF T <> nil Then
    Begin {dispose of range tree}
      Disposeof(T^.L);  {dispose of left child}
      Disposeof(T^.R);  {dispose of right child}
      IF T^.nd <> nil Then
        DisposeOF(T^.nd); {dispose of nextdim tree, if applicable}
      Dispose(T);       {dispose of the node}
    End;  {dispose of range tree}
End;  {Disposeof}

Procedure CreateNewTree(Var newtree:nodeptr; T:nodeptr;
                        dim:integer); Forward;
{This procedure creates a new tree in dimension dim when we have
determined needs rebuilding once the old tree has been disposed}

Procedure CreateTreesNextDim(Var T: nodeptr; dim: integer);
{We have created a new range tree in dimension dim.  We must now recurse
through dimensions and recreate range trees in dims (dim+1) to lastdim}

Begin {CreateTreesNextDim}
  IF T <> nil Then
    Begin {recurse through tree and create trees in next dim at all of
           the interior nodes}
      CreateTreesNextDim(T^.L, dim); {create tree in nextdim at L child}
      IF T^.nt = interior Then
        CreateNewTree(T^.nd, T, (dim+1));{create tree in nxtdim at node}
      CreateTreesNextDim(T^.R, dim); {create tree in nxtdim at R child}
    End;  {recurse through tree and create trees in next dim at all of
           the interior nodes}
End;  {CreateTreesNextDim}
```

```
Procedure CreateNewTree;
{Create a new range tree in dimension dim of the leaf nodes at the
subtree with root node T}

Var head, list: listptr; {somewhat misleading variable names here:
                          list is list of nodes that will create the DRT
                          head is used to traverse through the list
                          while list points to the start of the list}
    list2, list3,
    head2, head3: listptr;  {gets nodes from nextdim trees of children
                             (or leaves) which are merged in O(n) time
                             to rebuild range tree in minimal time}
    cur: nodeptr;         {cur is first node in linked list of nodes in
                           tree T that is to go in the new tree}
    nln: listptr;         {this is used to add a new node to the list}

  Procedure DisposeList(Var list: listptr);
  {After we have used a list to sort the nodes and removed the nodes
  from the list in building the tree structure, we dispose of the list}

  Var cur: listptr; {used to trace through the list}

  Begin {DisposeList}

    While list <> nil Do
      Begin {dispose of next element}
        cur:= list^.next;
        dispose(list);
        list:= cur;
      End;  {dispose of next element}

  End;  {DisposeList}

  Procedure StaticBuildDRT(Var newtree: nodeptr; list: listptr;
                           dim: integer);
  {called to build a DRT from a static data set}
  {when called, list contains AT LEAST two nodes}

  Var current: listptr; {traverses through list to build leaf level}
      prt:     nodeptr; {used to keep track of head node at leaf level}
      count:   integer; {number of nodes at leaf level}

    Procedure FinINode(Var nin: nodeptr; bfc: integer);
    {we have just created an interior node.  Initialize it's settings.}

    Begin {FinINode}

      nin^.nt:= interior; {interior node}
      nin^.dp:= sv;       {interior node}
      nin^.bf:= bfc;      {balance factor is passed in}
      nin^.p:= nil;       {interior nodes not linked across level}
      nin^.n:= nil;       {interior nodes not linked across level}
      nin^.nd:= nil;  {we build nextdim after this dim is finished}

    End;  {FinINode}
```

```
Procedure TopStruct(Var nt: nodeptr; count: integer;
                      Var prt: nodeptr);
{We rebuild our structure from the bottom up using a top-down
 recursion. That is sure to give us a balanced DRT as the recursion
 puts half of the datapoints on each side in each call}
Var lc, rc: nodeptr; {used to create L & R child internal nodes}
    bf: integer;     {balance factor of interior node}

  Function Powerof2(x: integer): boolean;

  Const base: integer = 2;
  Var prod: integer;

  Begin {Powerof2}
    prod:= 1;
    While (prod < x) Do
      prod:= prod * base;
    IF prod = x Then
      Powerof2:= true
    Else {prod > x}
      Powerof2:= false;
  End;  {Powerof2}

Begin {TopStruct}
  IF count >= 4 Then
    Begin {create two new interior nodes}
      new(lc);    new(rc);
      nt^.L:= lc; nt^.R:= rc;
      TopStruct(lc, round(count / 2) , prt);
      TopStruct(rc, trunc(count / 2) , prt);
      IF (Powerof2(trunc(count/2)) and ((round(count/2) -
          trunc(count/2)) = 1)) Then
        bf:= -1
      ELSE
        bf:=  0;
      FinINode(nt, bf);
    End   {create two new interior nodes}
  Else
    IF count = 3 Then
      Begin {create one new interior node}
        new(lc);          nt^.L:= lc;
        {attach last three nodes}
        nt^.L^.L:= prt;  prt:= prt^.n;
        nt^.L^.R:= prt;  prt:= prt^.n;
        nt^.R:=prt;      prt:= prt^.n;
        FinINode(nt^.L, 0);  FinINode(nt,   -1);
      End   {create one new interior node}
    Else
      IF count = 2 Then
        Begin {insert leaf nodes}
          nt^.L:= prt;  prt:= prt^.n;
          nt^.R:= prt;  prt:= prt^.n;
          FinINode(nt, 0);
        End  {insert leaf nodes}
      Else
        Begin {count does not = 1 or 0 when all is well}
          writeln('Error in TopStruct.  Count < 2!!!');
          Halt;
        End;  {count does not = 1 or 0 when all is well}
End;  {TopStruct}
```

```
Procedure TreeBuild(Var nt: nodeptr; prt: nodeptr; count: integer);
{Intializes the new tree and calls the procedures to build it}

Var p: nodeptr;

    Procedure FinStruct(T: nodeptr);

    Begin {FinStruct}
      IF T <> nil Then
        Begin
          IF T^.nt = interior Then
            Begin
              FinStruct(T^.L);
                T^.key:= p^.key;
                p^.da:= T;
              FinStruct(T^.R);
            End
          Else
            p:= T;
        End;

    End;  {FinStruct}

Begin {TreeBuild}
  New(nt);                     {we need to allocate the root node}
  nt^.L:= nil; nt^.R:= nil;    {initialize it not to point to anything}
  TopStruct(nt, count, prt);   {we build the structure}
  p:= nil;
  FinStruct(nt);               {we set keys of int. nodes and da's}
End;  {TreeBuild}

Begin {StaticBuildDRT}

  {we first build the leaf level}
  count:= 1;
  current:= list^.next; {remember, dummy header node}
  current^.DRTnode^.p:= nil; {the first node at leaf level in a DRT
                              has no predecessor}
  While current^.next <> nil Do
    Begin {continue to thread nodes}
      count:= count + 1;
      current^.DRTnode^.n:= current^.next^.DRTnode;
      current^.next^.DRTnode^.p:= current^.DRTnode;
      current:= current^.next;
    End;  {continue to thread nodes}

  current^.DRTnode^.n:= nil;  {last node at leaf level in a DRT
                               has no successor}

  prt:= list^.next^.DRTnode;  {first node at leaf level}
  DisposeList(list);  {nodes have been connected, no longer need list}

  {we build top down - stop when one node}
  TreeBuild(newtree, prt, count);

End;  {StaticBuildDRT}
```

```
Begin {CreateNewTree}
  {As we are going to create a perfectly balanced range tree, we first
   need to get the nodes to be inserted in order}
  {initialize list with dummy header node}
  new(list);                head:= list;
  list^.DRTnode:= nil;      list^.next:= nil;
  {remember that leaves are linked in sorted order}

  {get lists from nextdim trees of children or leaf nodes}
  new(list2);                  new(list3);
  list2^.DRTnode:= nil;        list3^.DRTnode:= nil;
  list2^.next:= nil;           list3^.next:= nil;

  IF T^.L^.nt = leaf Then
    Begin
      new(nln);      new(nln^.DRTnode);
      Copynode(T^.L, nln^.DRTnode, dim, 'n');
      nln^.next:= nil;
      list2^.next:= nln;
    End
  Else {T^.left = interior}
    Begin
      head2:= list2;
      cur:= T^.L^.nd;
      While cur^.nt <> leaf Do
        cur:= cur^.L;
      While cur <> nil Do
        Begin {get next node}
          new(nln);      new(nln^.DRTnode);
          Copynode(cur, nln^.DRTnode, dim, 'n');
          nln^.next:= nil;
          head2^.next:= nln;
          head2:= nln;
          cur:= cur^.n;
        End;  {get next node}
    End;
  IF T^.R^.nt = leaf Then
    Begin
      new(nln);
      new(nln^.DRTnode);
      Copynode(T^.R, nln^.DRTnode, dim, 'n');
      nln^.next:= nil;
      list3^.next:= nln;
    End
  Else {T^.right = interior}
    Begin
      head3:= list3;
      cur:= T^.R^.nd;
      While cur^.nt <> leaf Do
        cur:= cur^.L;
      While cur <> nil Do
        Begin {get next node}
          new(nln);      new(nln^.DRTnode);
          Copynode(cur, nln^.DRTnode, dim, 'n');
          nln^.next:= nil;
          head3^.next:= nln;
          head3:= nln;
          cur:= cur^.n;
        End;  {get next node}
    End;
  {we now merge list2 and list3 into list in linear time}

  head2:= list2^.next;  head3:= list3^.next;

  While ((head2 <> nil) and (head3 <> nil)) Do
      IF head2^.DRTnode^.key < head3^.DRTnode^.key Then
        Begin
          head^.next:= head2;
          head2:= head2^.next;
          head:= head^.next;
        End
      Else {head3^.DRTnode^.key < head2^.DRTnode^.key}
        Begin
          head^.next:= head3;
          head3:= head3^.next;
          head:= head^.next;
        End;

  IF head2 = nil Then
    head^.next:= head3
  Else {head3 = nil}
    head^.next:= head2;

  {at this point, all of the nodes, except the dummy headers, have been
   moved to list from list2 and list3 - dispose of dummy headers}

  list2^.next:= nil; dispose(list2);
  list3^.next:= nil; dispose(list3);

  {we now build the tree}
  StaticBuildDRT(newtree, list, dim);
    {this procedure disposes of list when it is finished}

  {if not in last dimension, must rebuild next dimension}
  IF dim <> dimensions Then
    CreateTreesNextDim(newtree, dim)
  Else
    newtree^.nd:= nil;

End;  {CreateNewTree}

Procedure Rebuild(Var T: nodeptr; root: nodeptr; dim: integer);
{Rebuild the range tree rooted at T in dimension dim of the leaf nodes
 in the subtree of root node root}

Begin {Rebuild}
  {Rotation Procedures should not call this on a leaf node}
  {They also shouldn't call this procedure when dim = dimensions!}
  IF (T <> nil) and (dim <= dimensions) Then
    Begin {rebuilding is possible - do it}
      Disposeof(T);  {dispose of the tree T}
      CreateNewTree(T, root, dim);  {rebuild the tree T}
    End  {rebuilding is possible - do it}
  Else
    Begin
      writeln('Error! Rebuild called with invalid conditions!');
      writeln('Terminating program.');
      HALT;
    End;
End;  {Rebuild}
```

```
Procedure RotateLeft(Var p: nodeptr);
{When the tree is unbalanced to the right, this procedure is called to
 perform a left rotation}

Var temp: nodeptr; {used as temp. var. for rotation}

Begin {RotateLeft}

  IF p = nil Then  {root node of rotation is nil, can't rotate}
    Begin {Error! Empty Subtree}
      writeln('Error in RotateLeft. Can''t rotate an empty subtree.');
      HALT;  {do not continue, critical error}
    End   {Error! Empty Subtree}
  Else {the root node of the rotation is not nil}
    IF p^.R = nil then  {a primary node of the rotation is nil}
      Begin {Error!  Empty subtree can't be root}
        writeln('Error in RotateLeft. Can''t make empty subtree root.');
        HALT; {do not continue, critical error}
      End   {Error!  Empty subtree can't be root}
    Else {primary nodes are not nil}
      IF ((p^.nt <> leaf) and (p^.R^.nt <> leaf)) Then
        Begin {valid rotation}
          {perform the rotation}
          temp:= p^.R;      {temp will be the root}
          p^.R:= temp^.L;   {left subtree of x --> right subtree of root}
          temp^.L:= p;      {root --> left subtree of x}
          p:= temp;         {root:= x}
          {rotate trees in next dimension as well}
          temp:= p^.nd;     {temp:= B}
          p^.nd:= p^.L^.nd;{root^.nd:= A}
          p^.L^.nd:= temp; {root^.left^.nd:= B}
        End  {valid rotation}
      ELSE {primary nodes are invalid as one or more is a leaf node}
        Begin {Error!  Can't rotate on a leaf node!}
          writeln('Invalid primary nodes in RotateLeft.');
          writeln('Critical Error! Terminating Program.');
          HALT;  {Critical Error!}
        End;  {Error!  Can't rotate on a leaf node!}

End;  {RotateLeft}

Procedure RotateRight(Var p: nodeptr);
{When the tree is unbalanced to the left, this procedure is called to
 perform a right rotation}

Var temp: nodeptr; {used as temp. var. for rotation}

Begin {RotateRight}

  IF p = nil Then
    Begin {Error! Empty Subtree}
      writeln('Error in RotateRight.  Can''t rotate an empty subtree.');
      HALT;  {do not continue, critical error}
    End   {Error! Empty Subtree}
  Else {root node of rotation is not nil}
    IF p^.L = nil Then  {primary node of rotation is nil}
      Begin {Error! Empty subtree can't become root}
        writeln('Error in R.Right. Can''t make empty subtree root.');
        HALT;  {do not continue, critical error}
      End   {Error! Empty subtree can't become root}
    Else {primary nodes of rotation are not nil}
      IF ((p^.nt <> leaf) and (p^.L^.nt <> leaf)) Then
        Begin {valid rotation}
          {perform the rotation}
          temp:= p^.L;      {temp will be the root}
          p^.L:= temp^.R;   {right subtree of x --> left subtree of root}
          temp^.R:= p;      {root --> right subtree of x}
          p:= temp;         {root:= x}
          {rotate trees in next dimension}
          temp:= p^.nd;     {temp:= B}
          p^.nd:= p^.R^.nd;{root^.nd:= A}
          p^.R^.nd:= temp; {root^.right^.nd:= B}
        End  {valid rotation}
      ELSE  {primary nodes are invalid as one or more is a leaf node}
        Begin {Error! Can't rotate on a leaf node!}
          writeln('Invalid primary nodes in RotateRight.');
          writeln('Critical Error! Terminating Program.');
          HALT;  {Critical Error!}
        End;  {Error! Can't rotate on a leaf node!}

End;  {RotateRight}

Procedure RightBalance(VAR T: nodeptr; dim: integer;
                        Var taller: boolean);
{This procedure is called to correct an imbalance in the right subtree.}
{i.e. the tree is right high (BF = 2)}

Var x,          {pointer to right subtree of T}
    w: nodeptr; {pointer to left subtree of x}

Begin {RightBalance}
  x:= T^.R;

  Case x^.bf of
     1: Begin {RH} {a single left rotation corrects this situation}
          T^.bf:=  0;     {after rotation, this node is balanced}
          x^.bf:=  0;     {after rotation, this node is balanced}
          RotateLeft(T);  {perform single left rotation}
          taller:= false; {subtree hasn't increased in height}
          IF dim <> dimensions Then
            Rebuild(T^.L^.nd, T^.L, (dim+1));
        End;  {RH} {a single left rotation corrects the situation}
     0: Begin {EH}  {By def'n of AVL trees, this case doesn't arise.}
          writeln('Error in RightBalance.  Condition of EH.');
          HALT; {do not continue, critical error}
        End;  {EH}  {By def'n of AVL trees, this case doesn't arise.}
    -1: Begin {LH} {we need a double rotation to correct the situation}
          w:= x^.L; {will become the root}
          Case w^.bf of   {w determines balance factors of x and T}
             0: Begin T^.bf:=  0;  x^.bf:=  0; End;
            -1: Begin T^.bf:=  0;  x^.bf:=  1; End;
             1: Begin T^.bf:= -1;  x^.bf:=  0; End;
          End; {Case w^.bf of}
          w^.bf:= 0;      {w is always balanced after rotation}
          RotateRight(x); {first rotate right on x, w replaces x}
          T^.R:= x;       {reconnect x to right child of T}
          RotateLeft(T);  {now rotate left on T}
          taller:= false; {the subtree has not increased in height}
          IF dim <> dimensions Then
            Begin
              Rebuild(T^.L^.nd, T^.L, (dim+1));
              Rebuild(T^.R^.nd, T^.R, (dim+1));
            End;
        End;  {LH} {we need a double rotation to correct the situation}
  End; {Case x^.bf of}
End;  {RightBalance}

Procedure LeftBalance(Var T: nodeptr; dim: integer;
                         Var taller: boolean);
{This procedure is called to correct an imbalance in the left subtree.}
{i.e. the tree is left high (BF = -2)}

Var x,          {pointer to left subtree of T}
    w: nodeptr; {pointer to right subtree of x}

Begin {LeftBalance}
  x:= T^.L;

  Case x^.bf of
    -1: Begin {LH} {a single right rotation corrects the situation}
          T^.bf:=  0;  {after rotation, this node is balanced}
          x^.bf:=  0;  {after rotation, this node is balanced}
          RotateRight(T);  {perform single right rotation}
          taller:= false;  {subtree hasn't increased in height}
          IF dim <> dimensions Then
            Rebuild(T^.R^.nd, T^.R, (dim+1));
        End;  {LH} {a single right rotation corrects the situation}
     0: Begin {EH} {By def'n of AVL trees, this case doesn't arise.}
          writeln('Error in LeftBalance.  Condtion of EH.');
          HALT;  {do not continue, critical error}
        End;  {EH} {By def'n of AVL trees, this case doesn't arise.}
     1: Begin {RH} {we need a double rotation to correct the situation}
          w:= x^.R; {will become the root}
          Case w^.bf of {w determines balance factors of x and T}
             0: Begin T^.bf:=  0;  x^.bf:=  0; End;
             1: Begin T^.bf:=  0;  x^.bf:= -1; End;
            -1: Begin T^.bf:=  1;  x^.bf:=  0; End;
          End;  {Case w^.bf of}
          w^.bf:= 0; {w is balanced after the rotation}
          RotateLeft(x);  {first rotate left on x, w replaces x}
          T^.L:= x;       {reconnect x to left child of T}
          RotateRight(T); {now rotate right on T}
          taller:= false;  {the subtree has not increased in height}
          IF dim <> dimensions Then
            Begin
              Rebuild(T^.L^.nd, T^.L, (dim+1));
              Rebuild(T^.R^.nd, T^.R, (dim+1));
            End;
        End;  {RH} {we need a double rotation to correct the situation}
    End;  {Case x^.bf of}
End;  {LeftBalance}
```

```
Procedure Insert(Var T: nodeptr; nn: nodeptr; Var taller: boolean;
                 dim: integer);
{This procedure inserts the node nn into the dynamic range AVL tree T.
 Taller is true if the height of tree increases and is false otherwise}

Var tallersubtree: boolean;  {has height of the subtree increased?}
       nl: nodeptr;          {we need a new leaf during insertion as the
                              last leaf node found in our search is
                              converted to an interior node}
     tmptaller: boolean; {init. False in recursive call to nextdim}
     newTree: nodeptr;   {when we need a new range tree in the nextdim}
     ndn: nodeptr;       {copy node for insertion in nextdim}

Begin {Insert}

  tallersubtree:= false; {only true when height of subtree increases}

IF T = nil Then  {we are inserting the first node}
  T:= nn
ELSE {we are not inserting the first node}
  Begin {tree is not empty}
    IF T^.nt = leaf Then {insertion occurs only at leaf level}
      Begin {we have reached the leaf level, insert}
        taller:= true; {insertion increases height}
        Copynode(T, nl,dim,'y');  {T becomes interior node so copy}
        T^.nt:= interior;    {T is now interior node}
        T^.dp:= sv;          {T is now interior node}
        {whatever points to T must now point to nl}
        IF T^.p <> nil Then T^.p^.n:= nl;
        IF T^.n <> nil Then T^.n^.p:= nl;
        {Insert nn as left or right child of T as appropriate}
        IF nn^.key <= nl^.key Then {new node is left child}
          Begin  {Insert new node as left child}
           {Direct ancestor is new interior node T}
            nn^.da:= T;
            T^.da:= nil; {interior node has no direct ancestor}
            T^.L:= nn;  T^.R:= nl;  {connect leaves to interior node T}
            T^.p:= nil; T^.n:= nil; {T is interior, remove from list}
            T^.key:= nn^.key;       {T's key = max. key left subtree}
            {insert new leaf into linked list}
            IF nl^.p <> nil Then {new node not first node in list}
              Begin {make connection between new node and previous}
                nl^.p^.n:= nn;   nn^.p:= nl^.p;
              End   {make connection between new node and previous}
            ELSE  {new node is first node in list}
              nn^.p:= nil;
            {connect new node to sibling}
            nn^.n:= nl;
            nl^.p:= nn;
          End     {Insert new node as left child}
        ELSE {new leaf is right child, old leaf is left child}
          Begin  {Insert new node as right child}
            {New node is only right child when its key is larger than
             all keys currently in the tree.  In this instance, new node
             does not have a direct ancestor and its sibling gets the
             parent node as its direct ancestor}
            nn^.da:= nil; {new node has no direct ancestor}
            nl^.da:= T;   {sibling gets parent as direct ancestor}
            T^.da:= nil;  {interior node has no direct ancestor}
            T^.L:= nl;  T^.R:= nn;  {connect leaves to interior node T}
            T^.p:= nil; T^.n:= nil; {T is interior, remove from list}
            T^.key:= nl^.key;       {T's key = max. key left subtree}
            {insert new leaf into linked list}
            IF nl^.n <> nil Then {new node is not last in list}
              {Note that the next four lines should never fire!}
              Begin {make connection between new node and next}
                nl^.n^.p:= nn;
                nn^.n:= nl^.n;
              End   {make connection between new node and next}
            ELSE {new node is last node in list}
              nn^.n:= nil;
              {connect new node to sibling}
              nn^.p:= nl;
              nl^.n:= nn;
          End;    {Insert new node as right child}
        IF dim <> dimensions Then
          Begin
            CreateNewTree(newtree, T, (dim+1));
            T^.nd:= newtree; {attach range tree in nextdim}
          End
        ELSE
          T^.nd:= nil;
      End    {we have reached the leaf level, insert}
    ELSE {we are not yet at leaf level}
      Begin {trace down to leaf level}
        IF nn^.key <= T^.key Then
          Begin {Insert into left subtree}
            Insert(T^.L, nn, tallersubtree, dim);
            IF tallersubtree Then
            Case T^.bf of
              -1: LeftBalance(T, dim, taller);
               0: Begin  T^.bf:= -1;  taller:= true;  End;
               1: Begin  T^.bf:= 0;  taller:= false;  End;
            End {Case T^.bf}
          ELSE
            taller:= false;
          End    {Insert into left subtree}
        ELSE {nn^.key > T^.key}
          Begin {Insert into right subtree}
            Insert(T^.R, nn, tallersubtree, dim);
            IF tallersubtree Then
              Case T^.bf of
                -1: Begin  T^.bf:= 0;  taller:= false;  End;
                 0: Begin  T^.bf:= 1;  taller:= true;   End;
                 1: RightBalance(T, dim, taller);
              End {Case T^.bf}
            ELSE
              taller:= false;
          End;    {Insert into right subtree}
          IF dim <> dimensions Then
            Begin
              CopyNode(nn, ndn, (dim+1), 'n');
              tmptaller:= false;
              Insert(T^.nd, ndn, tmptaller, (dim+1));
            End;
         End;    {trace down to leaf level}
  End;    {tree is not empty}
End; {Insert}

Procedure BalanceL(Var T: nodeptr; Var reduced: boolean; dim: integer);
{We have just reduced the height in the left subtree by 1.  If the
balance criterion has been violated then we must restore it.}
Var x,w: nodeptr;

Begin {BalanceL}
  Case T^.bf of
    -1: T^.bf:= 0;  {two subtrees are equal height} {reduced stays true}
     0: Begin  T^.bf:= 1;  Reduced:= false;  end;
     1: Begin {must rotate T to the left}
          IF ((T^.R^.bf = 1) or (T^.R^.bf = 0)) Then
            Begin {perform single rotation}
              IF T^.R^.bf = 1 Then
                Begin {Perform a normal AVL tree single left rotation}
                  {Reduced stays true}
                  T^.bf:= 0;  T^.R^.bf:= 0;
                End    {Perform a normal AVL tree single left rotation}
              ELSE {T^.R^.bf = 0}
                Begin {Single left rotation but change balance factors}
                  Reduced:= false;
                  T^.bf:= 1;  T^.R^.bf:= -1;
                End;  {Single left rotation but change balance factors}
              RotateLeft(T);
              IF dim <> dimensions Then
                Rebuild(T^.L^.nd, T^.L, (dim+1));
            End    {perform single rotation}
          ELSE {T^.R^.bf = -1}
            Begin {perform double rotation}
              {Reduced stays true}
              x:= T^.R;
              {procedure RightBalance double rotation case}
              w:= x^.L;
             Case w^.bf of
                0: Begin  T^.bf:= 0;  x^.bf:= 0;  End;
               -1: Begin  T^.bf:= 0;  x^.bf:= 1;  End;
                1: Begin  T^.bf:= -1;  x^.bf:= 0;  End;
              End;  {Case}
              w^.bf:= 0;
              RotateRight(x);
              T^.R:= x;
              RotateLeft(T);
              IF dim <> dimensions Then
                Begin
                  Rebuild(T^.L^.nd, T^.L, (dim+1));
                  Rebuild(T^.R^.nd, T^.R, (dim+1));
                End;
            End    {perform double rotation}
        End;  {must rotate T to the left}
  End;  {Case}
End;   {BalanceL}
```

```
Procedure BalanceR(Var T: nodeptr; Var reduced: boolean; dim: integer);
{We have just reduced the height in the right subtree by 1.  If the
balance criterion has been violated then we must restore it.}

Var x,w: nodeptr;

Begin {BalanceR}

  Case T^.bf of
     1: T^.bf:= 0;  {two subtrees equal height.}  {Reduced stays true}
     0: Begin  T^.bf:= -1;  Reduced:= false;  End;
    -1: Begin {must rotate T to the right}
          IF ((T^.L^.bf = -1) or (T^.L^.bf = 0)) Then
            Begin {perform single rotation}
              IF T^.L^.bf = -1 Then
                Begin {Perform a normal AVL tree single right rotation}
                  {Reduced stays true}
                  T^.bf:= 0;  T^.L^.bf:= 0;
                End {Perform a normal AVL tree rotation to the right}
              ELSE
                Begin {Single right rotation but change balance factors}
                  Reduced:= false;
                  T^.bf:= -1;  T^.L^.bf:= 1;
               End;  {Single right rotation but change balance factors}
              RotateRight(T);
              IF dim <> dimensions Then
                Rebuild(T^.R^.nd, T^.R, (dim+1));
            End    {perform single rotation}
          ELSE {T^.L^.bf = 1}
            Begin {perform double rotation}
              {Reduced stays true}
              x:= T^.L;
              {procedure LeftBalance double rotation case}
              w:= x^.R;
              Case w^.bf of
                0: Begin  T^.bf:= 0;  x^.bf:= 0;  End;
                1: Begin  T^.bf:= 0;  x^.bf:= -1;  End;
               -1: Begin  T^.bf:= 1;  x^.bf:= 0;  End;
              End; {Case}
              w^.bf:= 0;
              RotateLeft(x);
              T^.L:= x;
              RotateRight(T);
              IF dim <> dimensions Then
                Begin
                  Rebuild(T^.L^.nd, T^.L, (dim+1));
                  Rebuild(T^.R^.nd, T^.R, (dim+1));
                End;
            End    {perform double rotation}
        End;  {must rotation T to the right}
  End; {Case}

End;   {BalanceR}
```

```
Procedure Delete(dp: coordinate; Var T: nodeptr; Var reduced: boolean;
                  dim: integer);
{Delete node from T that contains key x.  If subtree height decreases,
 reduced:= true.}

Var Q: nodeptr;  {temporary pointer}
    x: integer;  {temp var}

Begin {Delete}
  x:= dp[dim];

  IF T <> nil Then
    IF T^.nt = leaf Then {one datapoint left in tree}
      IF T^.key = x Then
        T:= nil  {tree is now empty}
      ELSE
        writeln('key is not in tree')
    ELSE  {more than one datapoint left in tree}
      IF ((T^.L^.nt = leaf) and (T^.R^.nt = leaf)) Then
        {two datapoints in tree}
        IF T^.L^.key = x Then
          Begin {delete left child}
            {when deleting, we must change key value in direct ancestor}
            {however, in this case direct ancestor is parent node and it
             is replaced by right child so we do nothing}
            {we don't need to change direct ancestor in leaf that moves
             up as it doesn't change}
            T^.R^.p:= T^.L^.p; {remove T^.L from list}
            IF T^.R^.p <> nil Then
              T^.R^.p^.n:= T^.R;
            reduced:= true; {deletion always reduces height of subtree}
            Q:= T;    T:= T^.R;
            Disposeof(Q^.nd); Dispose(Q^.L);  Dispose(Q);
          End    {delete left child}
        ELSE
          IF T^.R^.key = x Then
            Begin {delete right child}
              {when deleting, we change key value in direct ancestor}
              IF T^.R^.da <> nil Then
                Begin
                  T^.R^.da^.key:= T^.L^.key;
                  {we must change direct ancestor in leaf that moves up}
                  T^.L^.da:= T^.R^.da;
                End
              ELSE {we are deleting last node;it has no direct ancestor}
                T^.L^.da:= nil;
              T^.L^.n:= T^.R^.n;  {remove T^.R from list}
              IF T^.L^.n <> nil Then
                T^.L^.n^.p:= T^.L;
              reduced:= true; {deletion reduces height of subtree}
              Q:= T;    T:= T^.L;
              Disposeof(Q^.nd); Dispose(Q^.R);  Dispose(Q);
            End    {delete right child}
          ELSE
            writeln('key is not in tree')
      ELSE  {more than two datapoints in tree}
        IF x <= T^.key Then {key, if present, is in left subtree}
          IF T^.L^.nt = leaf Then {three keys in tree}
            IF T^.L^.key = x Then {process deletion}
              Begin {delete left child}
                {when deleting, change key value in direct ancestor}
                {again, direct ancestor is parent node which gets
                 replaced by right child so again we do nothing}
                {we don't need to change direct ancestor as the node
                 that moves up is an interior}
                T^.R^.L^.p:= T^.L^.p; {remove T from list}
                IF T^.R^.L^.p <> nil Then
                  T^.R^.L^.p^.n:= T^.R^.L;
                reduced:= true; {deletion reduces height of subtree}
                Q:= T;  T:= T^.R;
                Dispose(Q^.L);  Q^.L:= nil; Q^.R:= nil; Disposeof(Q);
              End    {delete left child}
            ELSE {key not in tree}
              writeln('key is not in tree')
          ELSE {more than three keys in left subtree, normal deletion}
            Begin {normal deletion in left subtree}
              Delete(dp, T^.L, reduced, dim);
              IF reduced Then BalanceL(T, reduced, dim);
              IF dim <> dimensions Then
                Delete(dp, T^.nd, reduced, (dim+1));
            End    {normal deletion in left subtree}
        ELSE {key, if present, is in right subtree}
          IF T^.R^.nt = leaf Then {three keys in tree}
            IF T^.R^.key = x Then {process deletion}
              Begin {delete right child}
                {when deleting, we change key value in direct ancestor}
                IF T^.R^.da <> nil Then
                  Begin
                    T^.R^.da^.key:= T^.L^.R^.key;
                    {wechange direct ancestor in leaf that moves up}
                    T^.L^.R^.da:= T^.R^.da;
                  End
                ELSE
                  T^.L^.R^.da:= nil;
                T^.L^.R^.n:= T^.R^.n; {remove key from list}
                IF T^.L^.R^.n <> nil Then
                  T^.L^.R^.n^.p:= T^.L^.R;
                Q:= T;  T:= T^.L;
                reduced:= true;  {deletion reduces height of subtree}
                Dispose(Q^.R);  Q^.L:= nil; Q^.R:= nil; Disposeof(Q);
              End    {delete right child}
            ELSE {key is not in tree}
              writeln('key is not in tree')
          ELSE {more than three keys in right subtree, normal deletion}
            Begin {normal deletion in right subtree}
              Delete(dp, T^.R, reduced, dim);
              IF reduced Then BalanceR(T, reduced, dim);
              IF dim <> dimensions Then
                Delete(dp, T^.nd, reduced, (dim+1));
            End    {normal deletion in right subtree}
  ELSE {T = nil}
    writeln('Empty Tree.  Deletion is not possible.');

End;   {Delete}
```

```
Procedure RangeSearch(T: nodeptr; L,R: coordinate;
                         dim: integer); Forward;

Procedure ReportPoint(dp: coordinate);

Begin {ReportPoint}
  writeln(dp[1], dp[2], dp[3]);
End;  {ReportPoint}

Procedure CheckPoint(dp, L, R: coordinate; dim: integer);

Var i: integer;
    inrange: boolean;

Begin {CheckPoint}
  i:= dim;
  inrange:= true;
  While ((i <= dimensions) and (inrange)) Do
    IF not ((L[i] <= dp[i]) and (dp[i] <= R[i])) Then
      inrange:= false
    ELSE
      i:= i + 1;
  IF inrange Then
    ReportPoint(dp);
End;  {CheckPoint}

Procedure PerformRangeSearch(T: nodeptr; L,R: coordinate; dim: integer);

Begin {PRS}
  IF T^.nt = leaf Then
    Checkpoint(T^.dp, L, R, dim)
  ELSE
    IF dim <> dimensions Then
      RangeSearch(T^.nd, L,R, (dim+1));
End;  {PRS}

Procedure RangeSearch;
{We are searching for all keys between L and H inclusive in the range
AVL tree T.}

Var Q: nodeptr; LC, RC: nodeptr;

  Function NoOverlap(Q:nodeptr; L,R: coordinate; dim: integer): boolean;

  Begin {NoOverlap}
    IF Q <> nil Then
      IF ((R[dim] <= Q^.key) or (L[dim] > Q^.key)) Then
        NoOverlap:= true
      ELSE
        NoOverlap:= false
    ELSE
      NoOverlap:= false;
  End;  {NoOverlap}
```

```
Begin {RangeSearch}
  Q:= T;

  IF Q <> nil Then
    Begin {range tree not empty}
      While NoOverlap(Q,L,R,dim) Do
        IF R[dim] <= Q^.key Then
          Q:= Q^.L
        ELSE {L > Q^.key}
          Q:= Q^.R;

      IF Q <> nil Then
        Begin {normal range tree range search with common ancestor Q}
          LC:= Q^.L; {left child}
          RC:= Q^.R; {right child}

          {first deal with nodes in PLi}
          WHILE LC <> nil DO
           Begin {iterate to Lx'}
            IF LC^.nt <> leaf Then
              IF L[dim] <= LC^.L^.key Then
                Begin
                  PerformRangeSearch(LC^.R, L,R, dim);
                  LC:= LC^.L;
                End
              ELSE
                LC:= LC^.R
            ELSE
              Begin
                CHECKPOINT(LC^.dp, L,R, dim);
                LC:= nil;
              End;
           End;  {iterate to Lx'}

          {now deal with nodes in PHi}
          WHILE RC <> nil DO
           Begin {iterate to Hx'}
            IF RC^.nt <> leaf Then
              IF R[dim] > RC^.R^.key Then
                Begin
                  PerformRangeSearch(RC^.L, L,R, dim);
                  RC:= RC^.R;
                End
              ELSE
                RC:= RC^.L
            ELSE
              Begin
                CHECKPOINT(RC^.dp, L,R, dim);
                RC:= nil;
              End;
           End;  {iterate to Lx'}

        End   {normal range tree range search with common ancestor Q}
    End   {range tree not empty}
  ELSE
    writeln ('Empty Range Tree - Null Search Condition');
End;   {RangeSearch}
```

```
Procedure DisplayDRT(T: nodeptr);
{used for debugging purposes, output DRT structure to a datafile}
{at present, we are limited to three dimensions.  This could be modified
to output a k-dimensionsal tree}

Var ofile: text; {output data structure to this file}

  Procedure DispDRT(T: nodeptr; dim: integer); Forward;

  Function GNodetype(x: nodetype): char;
  {we output node type - this function get's the nodetype}

  Begin {Nodetype}

    IF x = leaf Then
      GNodetype:= ' ' {' ' = 'L'}
    Else
      GNodetype:= 'I'

  End;  {Nodetype}

  Procedure InorderTrav(T: nodeptr);
  {output the current tree via an inorder traversal}

  Begin {InorderTrav}

    IF T <> nil Then
      Begin {keep outputin'}
        InorderTrav(T^.L);
          write  (ofile, 'key: ', T^.key:5, GNodetype(T^.nt):3, '  ');
          IF ((T^.L <> nil) and (T^.R <> nil)) Then
           Begin
            writeln(ofile, '  T.L: ', T^.L^.key:4);
            writeln(ofile, '  T.R: ', T^.R^.key:3);
           End;
          Else
           writeln(ofile, '  T^.L: nil T^.R: nil');
        InorderTrav(T^.R);
      End;  {keep outputin'}

  End;  {InorderTrav}

  Procedure InorderNext(T: Nodeptr; dim: integer);
  {we must output trees in the nextdim attached to current tree -
   this procedure calls InorderTrav to output the tress 1 by 1}

  Begin {InorderNext}

     IF T <> nil Then
       Begin
         InorderNext(T^.L, dim);
           IF T^.nd <> nil Then
             Begin {output tree in nextdim}
               writeln(ofile, 'Next Structure attached to ', T^.key:3);
               InOrderTrav(T^.nd);
             End;  {output tree in nextdim}
         InorderNext(T^.R, dim);
       End;

  End;  {InorderNext}
  Procedure RecurseDim(T: nodeptr; dim: integer);
  {used to trace through trees in dimension two to get to and output
   trees in dimension three}

  Begin  {RecurseDim}
    IF T <> nil Then
      Begin {recurse through all trees in dimension two to get to
             trees in dimension three}
        RecurseDim(T^.L, dim);
        IF T^.nd <> nil Then
          InorderNext(T^.nd, dim);
        RecurseDim(T^.R, dim);
      End;  {recurse through all trees in dimension two to get to
             trees in dimension three}
  End;   {RecurseDim}

  Procedure DispDRT; {(T: nodeptr; dim: integer);}
  {does the recursion necessary to display the DRT structure}

  Begin {DispDRT}
    writeln(ofile, 'Dimension One: ');
    InorderTrav(T);  {output tree in dimension 1}
    IF T^.nd <> nil Then
      Begin {output dimension two structures}
        writeln(ofile, 'Dimension Two: ');
        InorderNext(T, dim); {output trees in dimension 2}
        writeln(ofile, 'Final Dimension Three: ');
        IF T^.nd^.nd <> nil Then
          RecurseDim(T, dim); {output trees in dimension 3}
      End;  {output dimension two structures}
  End;  {DispDRT}

Begin {DisplayDRT}
  {prepare the file for output}
  assign (ofile, outfile);
  rewrite(ofile);
  writeln(' New Tree: ');
  DispDRT(T, 1);
  close  (ofile);
  {we have now output the structure}
End;  {DisplayDRT}

Procedure Getnode(ndp: coordinate; Var nn: nodeptr; dim: integer);

Begin {Getnode}
  new(nn); {allocate the new node}

  {set all data values as appropriate for a not yet inserted leaf}
  nn^.da:= nil;
  nn^.nt:= leaf;
  nn^.key:= ndp[dim];
  nn^.dp:= ndp;
  nn^.L:= nil;   nn^.R:= nil;
  nn^.p:= nil;   nn^.n:= nil;
  nn^.bf:= 0;
  nn^.nd:= nil;

End;  {Getnode}
```

```
Procedure BuildDRT(Var T: nodeptr);

Var ndp: coordinate; {the datapoint being inserted}
    nn: nodeptr;     {the new node to be inserted}
    taller: boolean; {used by InsertDRT; height of the tree increased?}
    dim: integer;    {current dimension in which to insert new node}
    dfile: dfiletype;{file varaible pointing to datapoint file}
    i: integer;      {loop control variable}

Begin {BuildDRT}

  {prepare the file for input}
  assign (dfile, dpointfilei);
  reset  (dfile);

    For i:= 1 to numpoints Do
      Begin {insert next datapoint}
        dim:= 1;         {first insert datapoint in the first dimension}
        read(dfile, ndp);       {read in the datapoint}
        GetNode(ndp, nn, dim); {get/init. the node to be inserted}
        taller:= false; {must insert a node for the height to increase}
        {insert the new node into the DRT structure}
        Insert(T, nn, taller, dim);
        DisplayDRT(T);  {output the DRT to file for debugging purposes}
      End;  {insert next datapoint}

  close  (dfile);
  {we have tested our structure}

End;  {BuildDRT}

Procedure TestRS(T: nodeptr);

Const numsearches = 10;
      maxval      = 100;

Var dim: integer;
    i,j: integer;
    L,R: coordinate;
    tmp: integer;

Begin {TestRS}
  Randomize;
  For i:= 1 to numsearches Do
    Begin
      For j:= 1 to dimensions Do
        Begin
          L[j]:= Random(maxval);
          R[j]:= Random(maxval);
          IF R[j] < L[j] Then
            Begin
              tmp:=  R[j];
              R[j]:= L[j];
              L[j]:= tmp;
            End;
        End;
      dim:= 1;
      RangeSearch(T,L,R,dim);
    End;
End;  {TestRS}
```

```
Procedure DestroyDRT(Var T: nodeptr);

Var dp:       coordinate; {the datapoint being deleted}
    dim:      integer;    {current dimension to delete node from}
    dfile:    dfiletype;  {file variable pointing to datapoint}
    i: integer;           {loop control variable}
    reduced: boolean;     {recursive control in Delete}

Begin {DestroyDRT}

  {prepare file for input}
  assign (dfile, dpointfiled);
  reset  (dfile);

    For i:= 1 to numpoints Do
      begin {delete next datapoint}
        dim:= 1;  {we first delete datapoint from dimension 1}
        reduced:= false;
        read(dfile, dp); {read in the datapoint}
        Delete(dp, T, reduced, dim);  {delete next point from the DRT}
        DisplayDRT(T); {output the DRT to a file for debugging purposes}
      end;  {delete the next datapoint}

  close  (dfile);
  {we have tested our structure}
End;  {DestroyDRT}

Begin {Main}

  writeln('Please enter dimensions and number of datapoints.');
  readln (dimensions, numpoints);

  T:= nil;          {initialize tree as empty}
  BuildDrt(T);      {build a tree}
  TestRS(T);        {perform some range searches}
  DestroyDRT(T);    {destroy the tree}

End.  {Main}
```