A DYNAMIC DATA STRUCTURE FOR MULTI-DIMENSIONAL RANGE SEARCHING

by

Michael G. Lamoureau

TR96-105, March 1996

This is an unaltered version of the author's
MCS Thesis

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
www: http://www.cs.unb.ca

A DYNAMIC DATA STRUCTURE FOR MULTI-DIMENSIONAL RANGE SEARCHING

by

Michael G. Lamoureux

B.Sc., University of Prince Edward Island, Canada, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science in the
Faculty of Computer Science

This thesis is accepted

.....
Dean of Graduate Studies
THE UNIVERSITY OF NEW BRUNSWICK

March 1996

ABSTRACT

This thesis addresses the following question: Is it possible to have a k-dimensional data structure which provides for efficient k-d range search and dynamic updates on a set of n points in the worst case while maintaining reasonable storage and preprocessing requirements? Define a data structure to be optimally balanced when the product of its worst case preprocessing, storage, insertion, deletion, and range search cost functions is minimal and dynamically balanced when the product of its insert, delete and range search cost functions is minimal. The optimal balance cost is $\Omega(n^2 \lg^{4k} n)$ and the dynamic balance cost is $\Theta(\lg^{3k} n)$. The optimal worst case range search time is $\Theta(\lg^k n + t)$ (where we report t points in range) for such structures. Structures *optimal* for range search in the class of dynamically balanced structures are illustrated and found to be within $O(\lg^{k-1} n)$ of optimal.

A new k-d structure labeled the k-d Range Deterministic Skip List (DSL) is defined and analyzed along with a new variation of the dynamic range tree labeled the k-d Range AVL tree. Both structures are dynamically balanced and optimal for worst case range search in the model. Experimentally, a mere 20 milliseconds was required to report all 500 datapoints in range for the largest 4-d structure (of 336 MB) built.

Both structures perform well. They possess similar update times but we find that the k-d Range DSL is approximately twice as fast as the k-d Range AVL tree for insertions while the k-d Range AVL tree is approximately fifteen times faster for deletions.

ACKNOWLEDGMENTS

A kind thank-you is extended to my supervisor, Dr. Bradford G. Nickerson, for his suggestion of this specific topic and continual guidance throughout the research and preparation of this thesis. His comments and advice were valuable and insightful and helped greatly in the overall shaping of this work. A thank you is also extended to the readers, Dr. Joseph Horton, Dr. Colin Ware, and Dr. Barry Monson, whose valuable comments improved the overall presentation of the work.

Thanks are also extended to Kirby Ward and Mike MacDonald in the Faculty of Computer Science and Doug Swift, Robert Murray, and Anthony Fitzgerald of Computing Services for the continued technical advice and technical support that was necessary to complete the experimental portions of this research.

And finally, a thank you is extended to the Faculty of Computer Science whose continual funding made this work possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

The problem of k-dimensional range search, where k is the number of dimensions, 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. The areas of computer graphics, database management systems, computational geometry, pattern recognition, statistics, design automation, and geodesy and geomatics (geographic information systems) are just a few examples of places where one may find large-scale computer information systems. This is precisely where the problem of k-d range search 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 one must also ensure that the storage requirements of the underlying structure are kept minimal in order to feasibly use the structure. Also, today's applications require a dynamically updatable database and thus 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.

Upper bounds must be placed on the worst case search time, update operations, and storage requirement in order to have a structure that is feasible for use. The storage requirement must be $O(n^2)$, where n is the number of data points in the structure, or there

may not be enough storage space for the structure when the data set is large. The k-d range search time must be $O(kn)$, otherwise the structure provides no improvement over an unstructured brute force search. The same holds true for the time required by dynamic updates; otherwise the structure would have to be rebuilt with each insertion and deletion and such a structure should not be considered dynamic.

Nonetheless, implementable structures do exist which provide for efficient k-d range search in a dynamic environment. These include dynamizations of the k-d range tree of Bentley [Bent80a] and the k-d Range Deterministic Skip List (k-d Range DSL) of Lamoureux and Nickerson [Lamo95a]. However, until now, their efficiency has been in question. Although efficient, they do not meet the lower bounds for range search provided by the decision tree model. However, they are close to optimal if one defines a class of optimally balanced data structures based on the more reasonable dynamic model of Fredman [Fred81] which uses commutative semi-groups in ordered key spaces. In this model, we find that they are optimal for k-d range search.

Informally, *optimal balance* is defined as the product of the worst case storage, preprocessing, insert, delete, and range search cost functions. The class of *optimally balanced data structures* are those structures that have an optimal balance cost given by the minimal product. *Dynamic balance* is defined in terms of the product of the worst case insert, delete and range search cost functions and the class of *dynamically balanced structures* are those structures with a dynamic balance cost given by the minimal product.

The objectives of this thesis are stated as follows:

- review and analyze data structures that permit k-dimensional range search;
- determine time-space trade-offs for k-d range search in a dynamic environment;
- determine the cost functions that define an optimally balanced and a dynamically balanced data structure;
- illustrate implementable data structures that are dynamically balanced;
- experimentally analyze dynamically balanced structures optimal for k-d range search;

In the pages that follow, each of the objectives is addressed and an attempt is made to summarize and unite much of the relevant work in the discipline. The balance costs for optimally balanced and dynamically balanced data structures are determined and a dynamization of the k-d range tree and the k-d Range DSL are shown to be optimal for range search in the class of dynamically balanced data structures. Also, the k-d range tree and k-d Range DSL are found to be within $O(\lg^{k-1} n)$ of being optimally balanced. An experimental analysis of the k-d Range AVL Tree and the k-d Range DSL was performed and the results are illustrated and compared in Chapter 7.

2 DEFINITIONS

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 some specific range of values (e.g. height > 6'2' or \$23,000 ≤ annual income ≤ \$65,000). We 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 k range queries on different attributes 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 locate all records which satisfy a range query. Note that this definition does not imply that all the records in range have to be retrieved. Some authors define a range search to be a search which determines the number of records in the given range while others consider it to be a function which simply locates the records in range and a separate function is used for reporting. Unless noted otherwise, the common definition of range search which dictates the retrieval of all records in range is assumed. This implies that many of our Ω, O, and Θ analyses of the range search cost functions will have an extra term of t which captures the time needed to report the t records in range.

When specifying the limits of an orthogonal range query, we use the limit vectors **L**, for lower limits, and **H**, for upper limits, and use the notation $\mathbf{L}_i$ and $\mathbf{H}_i$ to correspond to the ith lower and upper limits.

We use five cost functions to portray and analyze the cost of a range search on a specific data structure X that supports range search on F. The three basic cost functions

P(n,k) = preprocessing time required to build X,

S(n,k) = storage space required by X,

Q(n,k) = time required to perform a range search on X

are as defined in [Bent79a]. In addition, we also consider the time required to insert a point into or delete a point from X as we are permitting dynamic updates on our structure. The cost of these dynamic operations are represented by

I(n,k) = time required to insert a new record into X, and

D(n,k) = time required to delete a record from X.

Following the example of [Will85a], we use U(n,k) to refer to the dynamic 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), in the worst case. This is often the case when the insertion and deletion times are dependent upon the search time. Unless specified otherwise, in this thesis, **all cost functions** given **are** for the **worst case**.

Sometimes we wish to refer only to the time to locate a single point as some k-d structures permit a member query to be answered in a time which is less than the time required to answer a range query. In this case we use $Q_M(n,k)$ to refer to the time required to answer a member query. For example, structures such as the k-d range tree and k-d Range DSL permit member queries to be answered in $\Theta(\lg n)$ time as compared to the $\Theta(\lg^k n + t)$ (for t datapoints in range) time required to answer a range query.

Some of the structures that we examine support partial match queries on s coordinates. The notation $Q_s(n,k)$ denotes a partial match query on s of the k coordinates. We define a partial match query to be an s-dimensional orthogonal range query where search intervals, possibly semi-infinite, are specified on s of the k dimensions. The remaining k-s dimensions are "free" or "unbound"; any value is permissible providing that the coordinate values in the specified dimensions are in the given ranges.

We define the optimal and dynamic balance cost in terms of an **amortized worst case analysis** which is a worst case analysis averaged over a sequence of *n* dynamic operations. An amortized analysis not only gives us lower bounds on the cost functions, but is deemed to be more realistic as the worst case for many of the data structures we illustrate occurs very rarely; the number of possible occurrences in a sequence of *n* dynamic operations is finite and bounded.

As many of the structures we examine are trees or tree-like structures, unless explicitly stated otherwise, we denote the leaf level as level 0 and the root node is denoted to be at level h in a structure of depth h. This implies that the leaf level is at depth h and that the root node is at depth 0.

We use the definitions of structural equivalence and functional equivalence as found in [Lamo96d].

Structural equivalence means that any structure or sub-structure of one data structure exists in a *similar* or isomorphic form in the other data structure. By *similar* form we imply that any operation that can be performed on one structure can be performed on the other structure in an analogous fashion. One implication of this definition is that if datum A logically exists next to datum B in structure G_1, then datum A logically exists next to datum B in structure G_2.

Functional equivalence means that the worst-case preprocessing, storage, insert, delete, and range search cost functions are of the same Θ complexity.

3 DATA STRUCTURES FOR RANGE SEARCHING

This thesis focuses only on data structures for orthogonal range queries as defined by Knuth ([Knut73]). Data structures and algorithms that support other types of range queries can be found in the excellent survey paper of [Mato94] which illustrates many problems which are of great importance in the field of computational geometry.

3.1 1-dimensional Data Structures

We first illustrate 1-dimensional data structures that permit range search, as many of the existing k-dimensional structures are based on multi-dimensional equivalents of efficient 1-dimensional data structures.

The simplest 1-dimensional data structure for range search is the unsorted list which permits a brute force range search of $\Theta(n)$ time complexity but which may be sorted to allow for a range search of complexity $\Theta(\lg n + t)$ which is executed by performing two binary searches, to locate the first and last points in range, and listing all points in between.

Using the idea of binary search, one may build a binary search tree to permit a worst-case range search of $\Theta(n)$ time complexity and then impose restrictions to maintain a height balance, such as those which define the AVL tree of [Adel62], to allow for a worst-case range search of $\Theta(\lg n + t)$ time complexity.

The AVL tree is a height balanced binary search tree which satisfies the invariant that the height difference between the left and right subtrees of a given node is never greater than one (1). As AVL trees are part of the dichromatic framework of Guibas and Sedgewick [Guib78], we may also use red-black trees for range search and obtain the same time bound on range search complexity.

One may also use a multi-way height balanced search tree known as the B-tree, of which a good overview may be found in [Come79], which allows for a range search of $\Theta(\lg_m n + t)$ (where m is the order of the B-tree) time complexity, or a deterministic skip list (DSL), as introduced by [Munr92], which allows for the same time complexity.

Although not the focus of this thesis, it is interesting to note that the B-tree and DSL are functionally and structurally *equivalent* structures [Lamo96d] and that one can be interchanged for the other in theory and practice. Those interested in a multi-way tree structure that provides an alternative to the B-Tree are referred to [Culi81].

We can combine the result of [Lamo96d] with the work of [Guib78] to conclude that many of the height balanced data structures which are efficient for range search are in fact in the same class of data structures and this allows us more flexibility in choosing an efficient data structure for a given task. The primary difference between the structures lies in the trade-off between recursion and iteration in the algorithms that define and dynamically maintain the structures.

3.2 k-dimensional Data Structures

Although there are many data structures which facilitate rapid average query time in k-dimensions, we find that there are relatively few which exhibit good worst case behavior. In comparison to the 1-d case, there are simple, inefficient structures that one may use to execute a k-d range search. These include the unsorted list, which requires $\Theta(kn)$ time to perform a range search, inverted tables, which are composed of k sorted lists on each of the coordinates and which require $\Theta(kn)$ time to perform a range search in the worst case (but often require less time in the average case), and cells, which divide the space up into a number of "boxes" or "blocks" which are searched separately if they are partially (or totally) within the desired search range. Cells are no better than inverted tables for range search in the worst case, but are often better in the average case. The reader is referred to [Bent79a] for a more detailed overview of these simple structures.

Another simple, inefficient approach is that of the multidimensional B-tree of [Guti80]. Designed specifically for exact match (member) queries, it is no better than inverted tables for range search in the worst case but may exhibit good average case performance for uniformly distributed random data sets.

A more sophisticated approach is that of the k-d tree of [Bent75] which is the multidimensional equivalent of the binary search tree. Although no better than inverted tables in the worst case scenario, the worst case is extremely rare and often taken to be the average case range search time of $O(n^{1-1/k} + t)$ which is guaranteed to be the worst case

search time if the given tree structure is height balanced (and therefore of minimal depth). The advantages of the structure are its low storage requirement, being essentially that of a binary search tree ($\Theta(kn)$), and the fact that dynamic operations are almost as simple as those of the regular 1-d binary search tree. In the average case, which is the worst case for a height balanced structure, $P(n,k) = O(n \lg n)$ and $U(n,k) = O(\lg n)$. The drawback of the structure is that height balance **cannot** be dynamically maintained ([Same90]).

An approach that is similar in structure, complexity, and functionality to that of the k-d tree of [Bent75] is that of the K-D-B-Tree of [Robi81] which combines the k-d tree of Bentley with the B-tree (as defined in [Come79]). Although a detailed analysis, to the author's knowledge, does not exist in the literature, experimental results indicate that its efficiency parallels that of the k-d tree.

A good overview of data structures for range searching can be found in [Bent79a], [Bent78], and [Maur78]. The point quadtree of [Same90] may also be used for multi-dimensional range search and, under the strong assumption of relatively evenly spaced data, has a worst case query time of $O(kn^{1-1/k})$ which is comparable to that of the k-d tree.

We examine k-d structures that allow for a more efficient (guaranteed) worst-case search time in the sections that follow.

3.2.1 The Range Tree

The range tree of Bentley [Bent80a] is a modified height-balanced binary search tree which is designed to detect all points that lie in a given range. We briefly review the range tree data structure and refer the reader to [Same90] for a good overview.

The 1-dimensional range tree (see Figure 1 (a)) is a height balanced binary search tree where the data points are stored in the leaf nodes which are linked in sorted order by a doubly linked list (the leaf nodes are threaded). A range search for [L:H] is performed by searching the tree for the node with the smallest key ≥ 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 $\Theta(\lg n + t)$ time and uses $\Theta(n)$ storage.

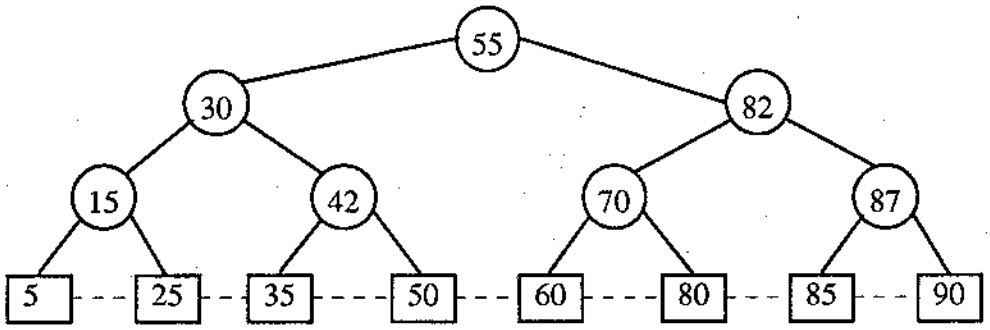

Figure 1. (a) A 1-d range tree from [Same90].

A 2-dimensional range tree (see Figure 1 (b)) is simply a range tree of range trees. We build a 2-dimensional range tree as follows: We first sort all of the points along one of the attributes, x, and then store them in a balanced 1-dimensional range tree, 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, y. In Figure 1 (b), the darkened links connect the range tree T_I (which has a *primed* node as root) in dimension 2 to the (*un-primed*) non-leaf node it is rooted at in dimension 1.

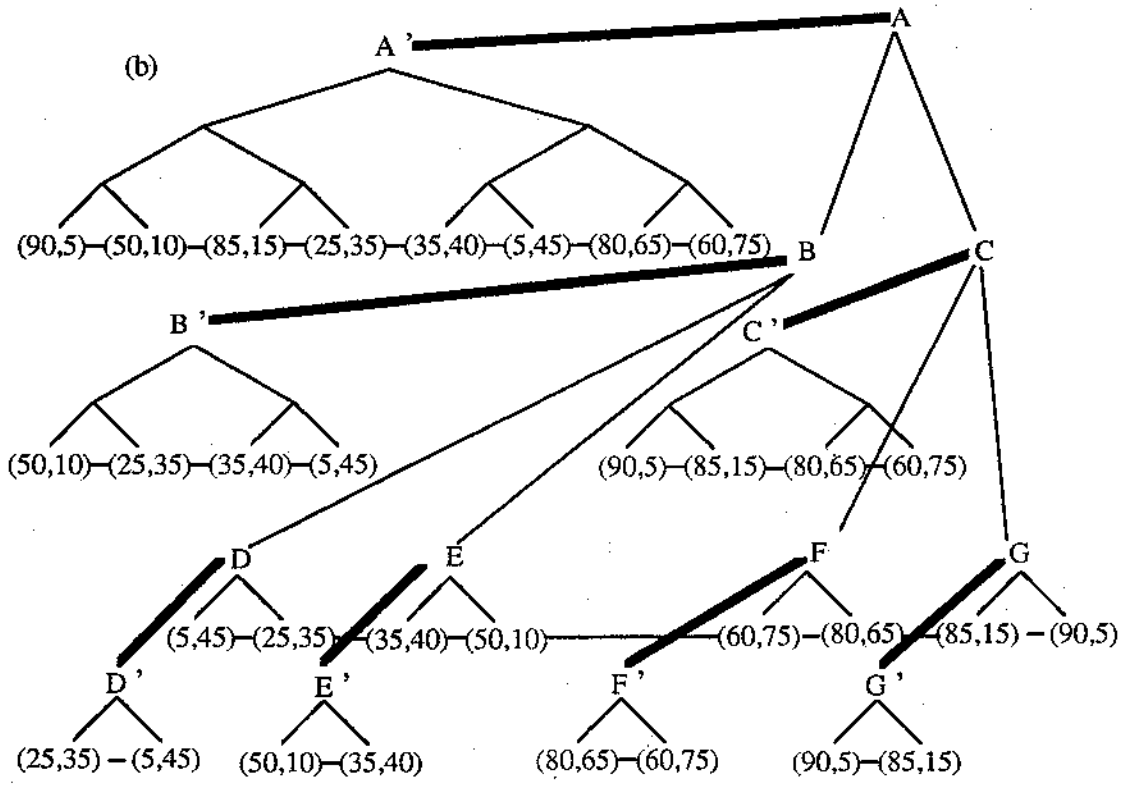

Figure 1. (b) A 2-d range tree from [Same90].

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. Then, for each P that is an element of $\{PL_i\}$ such that LEFT(P) is also in $\{PL_i\}$, we perform a 1-dimensional range search for $[L_y:H_y]$ in the 1-dimensional range tree rooted at node RIGHT(P). For each P that is an element of $\{PH_i\}$ such that RIGHT(P) is also in $\{PH_i\}$, we perform a 1-dimensional range search for $[L_y:H_y]$ in the 1-dimensional range tree rooted at node LEFT(P). We also check to see if L_x' and H_x' are located in the given range.

The above search algorithm can be seen to run in $\Theta(\lg^2 n + t)$ time and 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 $\Theta(\lg^k n + t)$ time.

The recursively defined k-dimensional range tree has $P(n,k) = \Theta(n \lg^{k-1} n)$ and $S(n,k) = \Theta(n \lg^{k-1} n)$ in the static case and has $P(n,k) = \Theta(n \lg^k n)$, $S(n,k) = \Theta(n \lg^{k-1} n)$, and $U(n,k) = \Theta(\lg^k n)$ in the dynamic case where an amortized worst case analysis is used. Dynamizations of the k-d range tree can be found in [Luek78], [Luek82], [Will85a], [Will85b], and, more recently, in [Lamo95b].

The range tree is important for a number of reasons. Not only is it a prime example of the paradigm of multi-dimensional divide and conquer, introduced by Bentley [Bent80a], but it is also a prime example of the effective use of a class of transformations which can be used to add range restriction to an existing data structure for a

decomposable searching problem such as range search [Bent79b]. Thus, we can build it bottom up starting with a 1-d range tree structure on the first coordinate or build it top down starting with a k-d data set and applying the paradigm of multi-dimensional divide and conquer until we reach a 1-d range tree structure.

The paradigm of multi-dimensional divide and conquer states that we can solve a problem of N points in k-space by first recursively solving two problems each of N/2 points in k-space and then recursively solving one problem of N points in (k-1) space. Thus, a structure which solves a problem using the paradigm stores two structures of N/2 points in k-space and one structure of N points in (k-1) space.

A given interior node in the range tree is the root of a subtree of n' nodes. It has two children which are the root nodes of range trees that each have approximately n'/2 nodes and it has a pointer to a (k-1)-dimensional tree of n' nodes. It is clear that the range tree is an example of the paradigm of multidimensional divide and conquer. Another example of the paradigm in use can be found in [Bent79c].

The theory of decomposable searching problems can be viewed as the *dual* of the paradigm of multidimensional divide and conquer. We can add a second range variable to a 1-d range tree structure (on one range variable) by building a range tree structure on the new variable and attaching to it 1-d range tree structures on the first range variable.

We find that the range tree is a dynamically balanced data structure and that it is optimal for the execution of range queries in the class of dynamically balanced data structures. Also, the static range tree can be derived from non-overlapping k-ranges, an array-based data structure that we explore shortly. This leads us to hypothesize that our minimal optimal balance cost holds for array based data structures in the random access model of computation as well and not just the pointer model that we normally assume and use in the definition and implementation of dynamic data structures.

3.2.2 The Priority Search Tree

The priority search tree of McCreight [McCr85] is designed for solving semi-infinite range queries of the form $([L_x:H_x],[L_y:\infty])$ in optimal time. It is a variant of a range tree in x and a heap (priority queue) in y. It makes the assumptions that no two data points have the same x coordinate and the reader is again referred to [Same90] for the details on the construction of the priority search tree. An example of a priority search tree is found in Figure 2.

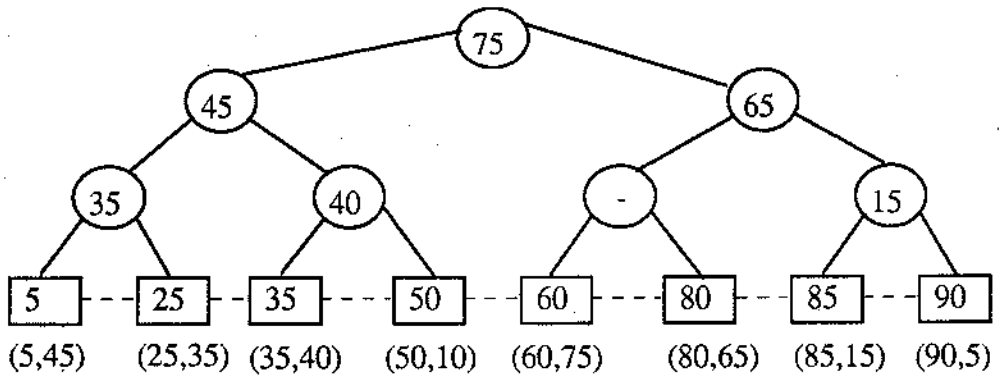

Figure 2. A priority search tree from [Same90].

The priority search tree requires minimal storage space, having $S(n,2) = \Theta(n)$ and is optimal for semi-infinite range search in two dimensions, i.e. $Q(n,1.5) = \Theta(\lg n + t)$. Dynamic updates are optimal as well, requiring only $\Theta(\lg n)$ time per update. However, a normal 2-d range query, $Q(n,2)$, requires $\Theta(n)$ time in the worst case.

The importance of the structure is that it is the basis and inspiration for the Range Priority (RT) tree of Edelsbrunner [Edel81] and the 2-d Search Skip List of Nickerson [Nick94]. The Range Priority (RT) tree is important as it shaves off a logarithmic factor from the k-d range query time of the range tree. The 2-d Search Skip List is important as it is the first well defined 2-d skip list data structure and the inspiration for the k-d Range DSL [Lamo95a], another dynamically balanced data structure.

3.2.3 The Range Priority Tree

The range priority tree, or the RT-tree, of Edelsbrunner [Edel81] is a k-d data structure designed specifically for k-d range search and it is similar to the range tree of Bentley. The structure is optimal for range search in two dimensions, $Q(n,2) = \Theta(\lg n + t)$, and takes advantage of the priority search tree of McCreight [McCr85] to achieve its optimality. An example of the range priority tree can be found in Figure 3.

An inverse priority search tree is a priority search tree, S, such that with each non-leaf node, I, of S we associate the point in the subtree rooted at I with the minimum value for its y coordinate that has not already been stored at a shallower depth in the tree.

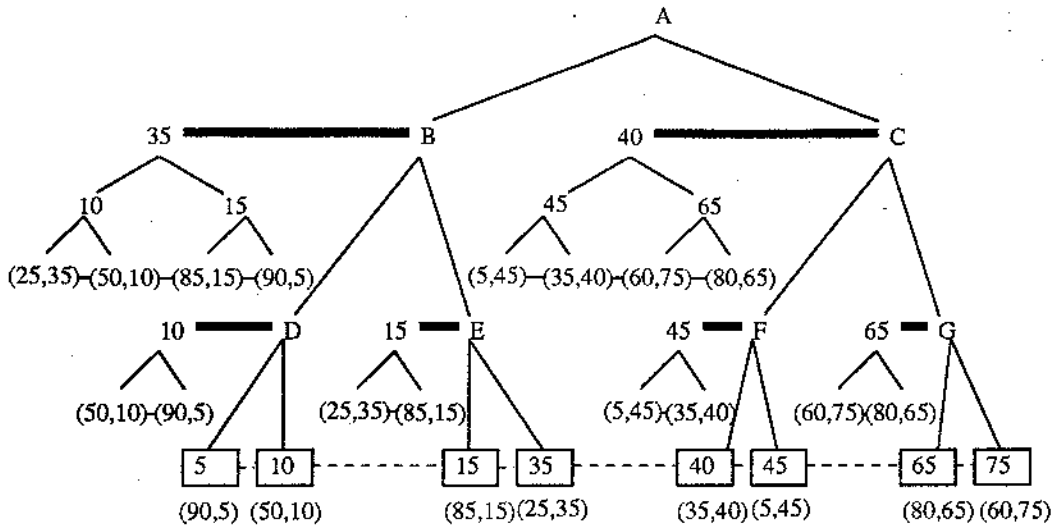

Figure 3. A 2-d range priority tree from [Same90].

A range priority tree is a balanced binary search tree, T, where all data points are stored in the leaf nodes and are sorted by their y coordinate values. With each non-leaf node I of T which is a right child of its parent we store an inverse priority search tree of the points in the subtree rooted at I ordered on their x coordinate values. With each non-leaf node I of T which is a left child of its parent, we store a priority search tree of the points rooted in the subtree of I ordered on their x coordinate values. This scheme allows us to shave off a logarithmic factor from the k-d range search time of the range tree for a $Q(n,k)$ of $\Theta(\lg^{k-1} n + t)$ for $k \geq 2$.

3.2.4 k-d Skip Lists

Nickerson [Nick94] has defined a number of k-d skip list data structures for k-d range search which are built from 1-d deterministic skip lists and similar both in structure and range query time to inverted tables. As they are not very efficient for k-d range search

in the worst case, we do no more than mention them and instead focus on Nickerson's 2-d search skip list as it provides an efficient alternative to the priority search tree and is the inspiration for the k-d Range DSL.

Remembering that a deterministic skip list is a tree-like structure which has a node structure that contains (at least) a down pointer, a right pointer, and a data-value, we use the following terms when describing the structure and the associated algorithms. The *down subtree* of a node consists of all nodes that can be reached by first traversing the down pointer of the current node such that those nodes are not reachable by traversing the down pointer of the current node's right sibling. The *immediate down subtree* of a node is composed of all nodes at depth (i+1) in the down subtree of the current node which is assumed to be located at depth i. The *direct descendent* in the immediate down subtree is that node which is the first node in a node's immediate down subtree. Finally, the *gap size* is defined to be the number of nodes in the *immediate down subtree* where we exclude the direct descendent.

For example, in Figure 4, the down subtree of the node (140, 37, -30, 41) at level 1 consists of the nodes (14, 41, 41, 41), (31, -30, -30, -30), (104, 2, 2, 2), and (140, 37, 37, 37) at level 0. The immediate down subtree of the node (M, ∅, -34, 52) at level 2 consists of the nodes (0, 52, 39, 52), (140, 37, -30, 41), and (M, ∅, -34, -34) at level 1. The direct descendent of (0, 52, 39, 52) at level 1 is (-123, 48, 48, 48) at level 0, and the gap size of (M, ∅, -34, -34) is 1.

3.2.4.1 2-d Search Skip List

The 2-d search skip list of Nickerson [Nick94] uses the idea inherent in the priority search tree and is based on the linked-list version of the 1-3 deterministic skip list (DSL) of [Munr92]. It is always balanced (in the sense of B-trees) as the leaves are located at the same depth c lg n, $1/2 \leq c \leq 1$. The properties of the structure are

1. Each node contains four values. These are

   i) (K_1, K_2) = keys of the node (the two top fields in Figure 4)

   ii) mink2 = minimum K_2 value for all nodes in the down subtree

   iii) maxk2 = maximum K_2 value for all nodes in the down subtree

2. Each node has two pointers; a down pointer and a right pointer.

3. Special nodes head, tail, and bottom indicate the start and end of list conditions.

4. All data points appear at the leaf level and some may appear at higher levels.

5. Every gap size in the skip list is of size 1, 2, or 3.

6. The skip list is ordered on the K_1 key values.

The cost functions are the same as those for the priority search tree and the structure is a prime example of the fact that deterministic skip lists provide efficient alternatives to height balanced search trees as illustrated in [Lamo95c] and [Lamo96d]. An example of a 2-d Search Skip List can be found in Figure 4.

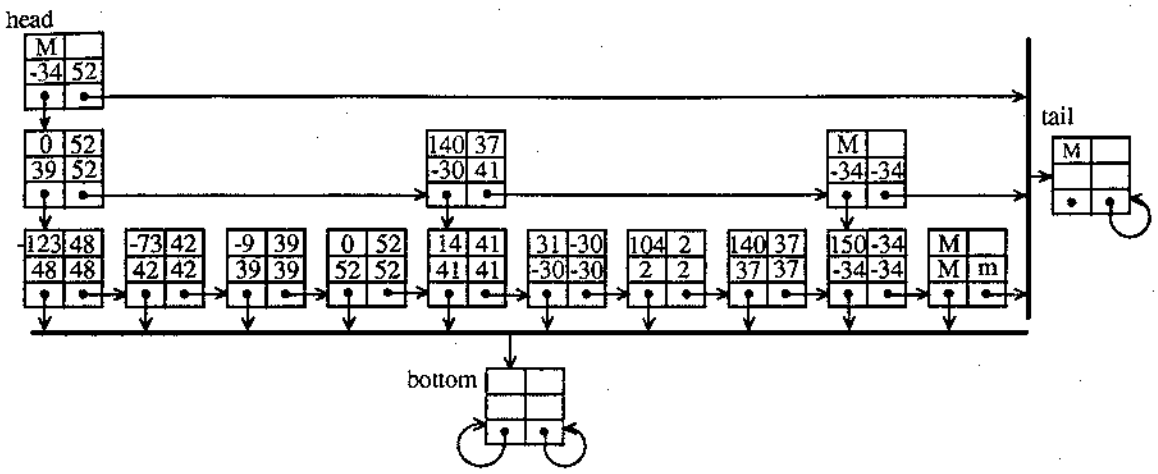

Figure 4. A 2-d search skip list from [Nick94].

3.2.5 k-Ranges

The k-ranges of Bentley and Maurer ([Bent80b] and [Bent78]) are important for a number of reasons. They allow for optimal worst-case range search under a decision tree analysis. They are one of the few array based data structures and the only one that the author knows of that was specifically designed for k-dimensional range search on k-dimensional data. They are primarily a theoretical construct and, to the author's knowledge, have not been implemented due to the unrealistically high storage requirements and difficulties inherent in attempting an implementation.

We note that k-ranges assume that the data has been normalized to the integer range 1 ... n and that all keys are unique. As the normalization time is of $\Theta(kn \lg n)$, it does not increase the overall complexity of the preprocessing, $P(n,k)$, needed to build the structure and, as such, is acceptable. Also, a range query can be normalized in $\Theta(k \lg n)$ time and is thus within the time needed for a range search, $Q(n,k)$.

Bentley and Maurer have defined three kinds of k-ranges which fall into the categories of overlapping and non-overlapping. Overlapping k-ranges are further divided into single level and multi-level, the first of which is optimal for k-d range search under a decision tree analysis and the second of which is almost optimal, trading query time for a decreased storage and preprocessing requirement. Non-overlapping k-ranges are defined as multi-level structures. We describe and provide a brief analysis of the three types of k-ranges. We note that all of the k-ranges are based on a generalization of Bentley's multi-dimensional divide and conquer paradigm [Bent80a] as this knowledge helps one to understand the cost-functions and underlying analysis.

There are a number of reasons that k-ranges are not used in practice, the most important one being that they are not dynamic data structures. As they are based on arrays, they are a static data structure and do not permit dynamic updates. To insert or delete a point from the structure essentially involves rebuilding the entire structure and this gives us an unacceptable update cost function which is equivalent to the preprocessing cost, unacceptable in any situation that requires dynamic updates.

3.2.5.1 Overlapping k-Ranges

A k-range is a data structure defined inductively for k = 1,2, ... m. We first exposit the 1-dimensional structure and then show how it can be extended to higher dimensions. Let G be some subset of our dataset F. To store G as a 1-range, we store G in a linear

array M of n elements. Each element in our array M consists of a set of points M_i and a pointer p_i ($1 \leq i \leq n$), where M_i is the set of all points of G with first coordinate equal to i, and where p_i points to the next non-empty set M_j with $i < j$ and j minimal. An example of a 1-range can be found in Figure 5(a).

To extend k-ranges to the case where k = 2, we store the 2-range for F by storing each of the sets $F_{i,j}^{(2)}$ for $1 \leq i \leq j \leq n$ as 1-ranges $R_{i,j}$ and setting up a two dimensional array P of pointers, each element $P_{i,j}$ pointing to $R_{i,j}$. Thus, to carry out a range search $([L_1:H_1],[L_2:H_2])$, we just have to search the 1-range $F_{L2,H2}^{(2)}$ for [L1:H1].

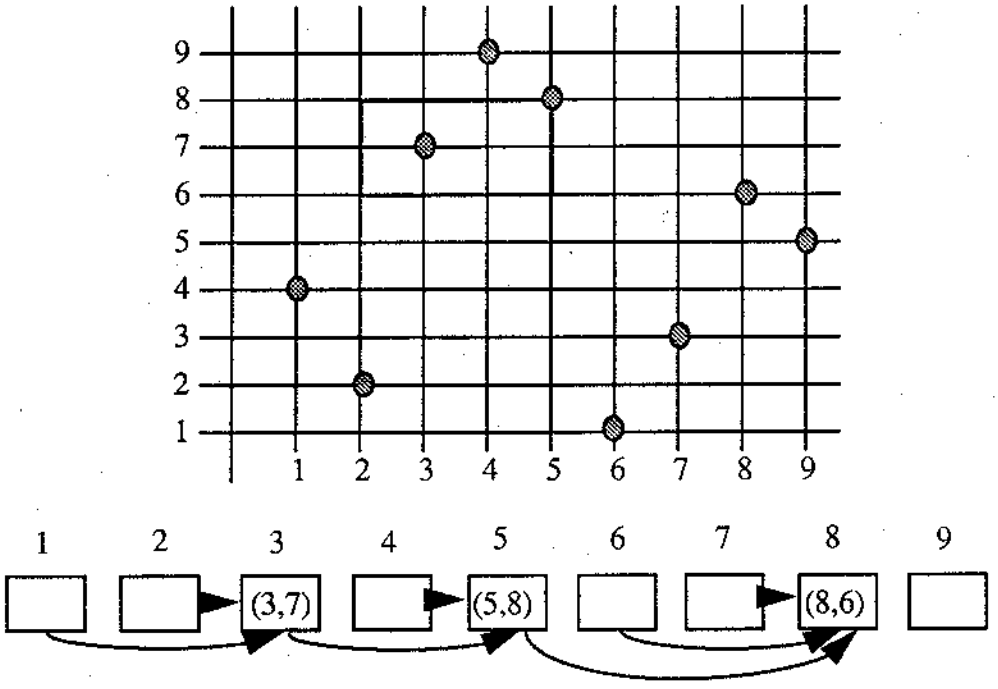

Figure 5. (a) Example of a 1-range ($F_{6,8}^{(2)}$) in a 2-range from [Bent80b].

Extension of k-ranges to the case where k = m should now be obvious. Store first all $F_{i,j}^{(k)}$ for $1 \le i \le j \le n$ as (k-1)-ranges $R_{i,j}$. Then, construct a 2-dimensional array P of pointers where each element $P_{i,j}$ points to $R_{i,j}$. To carry out a range search $([L_1:H_1],[L_2:H_2], ..., [L_k:H_k])$ in F, it suffices to carry out a range search $([L_1:H_1],[L_2:H_2], ..., [L_{k-1}:H_{k-1}])$ in $F_{Lk,Hk}^{(k)}$. Since this is stored as a (k-1) range, the process continues until it remains to search for $[L_1:H_1]$ in a 1-range.

Although these overlapping k-ranges have an optimal query time of $Q(n,k) = \Theta(k \lg n + t)$, they have a downside in the fact that storage and preprocessing are exponential in the number of dimensions, namely $S(n,k) = P(n,k) = \Theta(n^{2k-1})$.

3.2.5.2 Multi-Level k-Ranges

Bentley and Maurer [Bent80b] have devised a method to decrease the storage and preprocessing requirements substantially at the expense of an increased range query time. By defining the structure to be multi-level, they are able to reduce the storage and preprocessing costs to $\Theta(\ell^{k-1} n^{1+2/\ell})$ where ℓ is the number of levels in the structure. The query time is increased proportionally, becoming $\Theta(\ell^{k-1} \lg n + t)$.

The analysis above is more exact than that performed by Bentley and Maurer who ignore the factor of ℓ^{k-1} as ℓ, once chosen, is a constant. It is true that if small values of ℓ are chosen then ℓ^{k-1} is essentially constant and thus irrelevant, but, if ℓ is even moderately

large as compared to n, say $\ell = \lg n$, then we find that the analysis of Bentley and Maurer is insufficient. An example of a two-level 2-range can be found in Figure 5(b).

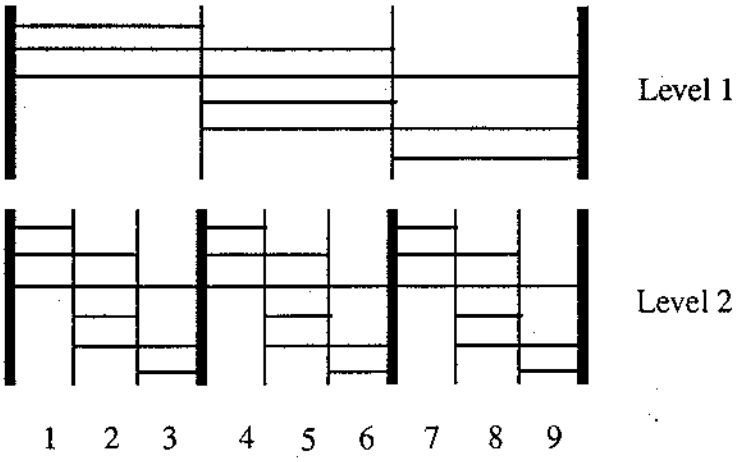

Figure 5. (b) Example of a two-level 2-range from [Bent80b].

We illustrate the technique to modify k-ranges into ℓ-level k-ranges by first focusing on 2-ranges, as the extension from ℓ-level 2-ranges to ℓ-level k-ranges is rather analogous to the extension from one-level 2-ranges to one-level k-ranges. The reader is referred to [Bent80b] for further information.

We now focus on a two-level 2-range. On the first level we consider one "block" which contains $n^{1/2}$ "units" (we may assume n is a perfect square) which represent $n^{1/2}$ points each. On the first level of the 2-level 2-range (see Figure 5(b)) we then store all $C_2^{1+n^{1/2}} = \Theta(n)$ consecutive intervals of units; that is, we store $\Theta(n)$ 1-ranges. The second level of our covering consists of $n^{1/2}$ blocks each containing $n^{1/2}$ units (which are

individual points). Within each block we store all possible intervals of units (points) as 1-ranges. The structure is depicted in Figure 5(b) for the case of $n = 9$. The bold vertical lines represent block boundaries and the regular vertical lines represent unit boundaries. Each horizontal line represents a 1-range structure.

A range query in a two-level 2-range is answered by choosing a covering of the particular y-range from the 2-level structure. This can always be accomplished by selecting at most one sequence of units from level one and two sequences of units from level two, giving us our range query time of $\Theta(\ell \lg n + t)$, as desired.

Although these structures are extremely interesting from a theoretical viewpoint, we still find them horribly inefficient for implementation. It is not just the fact that k-ranges are essentially static data structures and as such **do not** permit dynamic updates, but also the fact that an overlapping structure, by nature, contains too much repeated information in the index structure to be useful in terms of the storage requirement.

3.2.5.3 Non-Overlapping k-Ranges

The third type of k-ranges is probably the most efficient from a practical viewpoint. Although the cost of a range query is substantially higher than that for overlapping k-ranges, with the proper choice of the number of levels, ℓ, it is comparable to that of the k-d range tree which has been deemed to be quite acceptable. In fact, if we assume that our elements are from a commutative semi-group, then the k-d range tree is

optimal for range search (as set union is the commutative and associative addition operation), see [Fred81]. A two-level non-overlapping k-range is shown in Figure 5(c).

As in Figure 5(b), the bold vertical lines represent block boundaries and the regular vertical lines represent unit boundaries. However, unlike the overlapping k-ranges, each unit on level ℓ corresponds to one, and only-one, k-range structure.

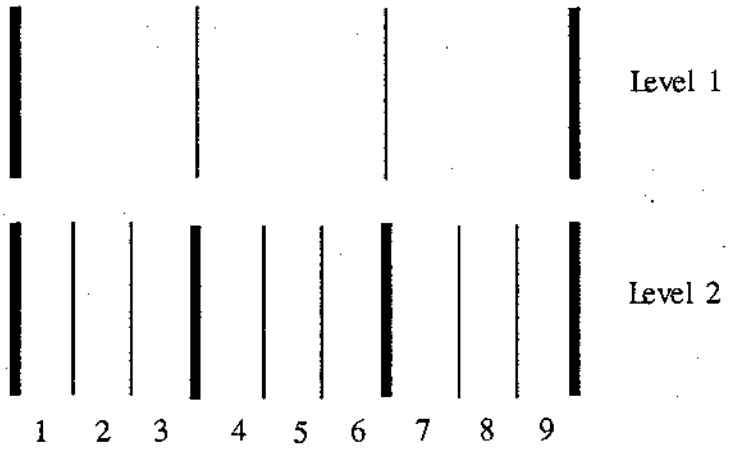

Figure 5. (c) A non-overlapping two-level 2-range.

As with the overlapping two-level 2-range, a range query in a non-overlapping two-level 2-range is answered by choosing a covering of the particular y-range from the two-level structure. The covering can always be chosen by choosing at most $O(n^{1/\ell})$ k-ranges at at most ℓ levels. This gives us a search time of $\Theta(\ell\, n^{1/\ell} \lg n + t)$ as it takes most $\lg n$ time to search a given k-range.

Range search time for an arbitrary ℓ-level k-range is $Q(n,k) = \Theta(\ell^{k-1}\, n^{1/\ell} \lg n + t)$. In contrast to the overlapping k-ranges, storage and preprocessing costs are substantially reduced, with $S(n,k) = \Theta(\ell^{k-1}\, n)$ and $P(n,k) = \Theta(\ell^{k-1}\, n \lg n)$. If we let ℓ vary with n, we achieve a tree-like data structure, and $\ell = \lg n$ gives the k-d range tree of Bentley. To see this, we substitute $\ell = \lg n$ into $S(n,k)$ to give $\Theta(n \lg^{k-1} n)$, $\ell = \lg n$ into $P(n,k)$ to give us $\Theta(n \lg^{k} n)$, and $\ell = \lg n$ into $Q(n,k)$ to give $\Theta(\lg^{k-1} n\ \ n^{1/\lg n} \lg n + t)$ which is equivalent to $\Theta(\lg^{k} n + t)$.

Non-overlapping k-ranges allow us to show that our minimal optimal balance cost holds in the array based memory model as well (and not just in the pointer based memory model). We set $\ell = \lg n$ in non-overlapping k-ranges to obtain a minimal balance cost function for the structure. Since non-overlapping k-ranges have a lower balance cost function than overlapping k-ranges and any other structural variations on the array model, we can use this minimal balance cost function to show that our optimal balance cost function holds in the array based memory model as well. Further discussion of k-ranges and optimal balance can be found in Chapter 6.

4 DYNAMIZING THE RANGE TREE

The range tree of Bentley [Bent80a] was originally defined as a static data structure which is optimal (assuming our records are from a commutative semi-group), in a worst case analysis, for answering k-dimensional range queries when only $\Theta(n \lg^{k-1} n)$ space is used. Willard and Lueker have devised theoretical modifications to the structure, which are summarized in [Will85a], that permit dynamic operations in $\Theta(\lg^k n + t)$ time when an amortized worst case analysis is used, but these dynamic modifications are difficult to implement, especially as their structures are based on the class of BBα trees, which are obscure at best. Only the simplest of their structures has been implemented and, to the author's knowledge, a detailed specification of the structure along with a complete and well-defined implementation of a dynamic k-d range tree structure did not exist in the literature until the author devised and elucidated the k-d Range AVL tree in [Lamo95b].

Although of the same order of complexity as the other k-d dynamic range tree data structures, the k-d Range AVL tree has advantages that the other structures do not. It is based on a simple modification of the well known AVL tree data structure [Adel62] and is thus straight-forward to understand and implement. The rotations needed to maintain balance are straight-forward modifications of the AVL tree rotations and easily specified (see [Lamo95b]). This structure provides us with a baseline structure against which other structures, and their implementations, designed to handle multi-dimensional range queries on multi-dimensional data, can be compared.

4.1 The 1-d Range AVL 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 its 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.

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 that we use are essentially the normal AVL tree insertion and deletion procedures with the modifications outlined below. We call an AVL tree with these modifications 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 6.

In Figure 6(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 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 6(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 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 the recursion unfolds) 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.

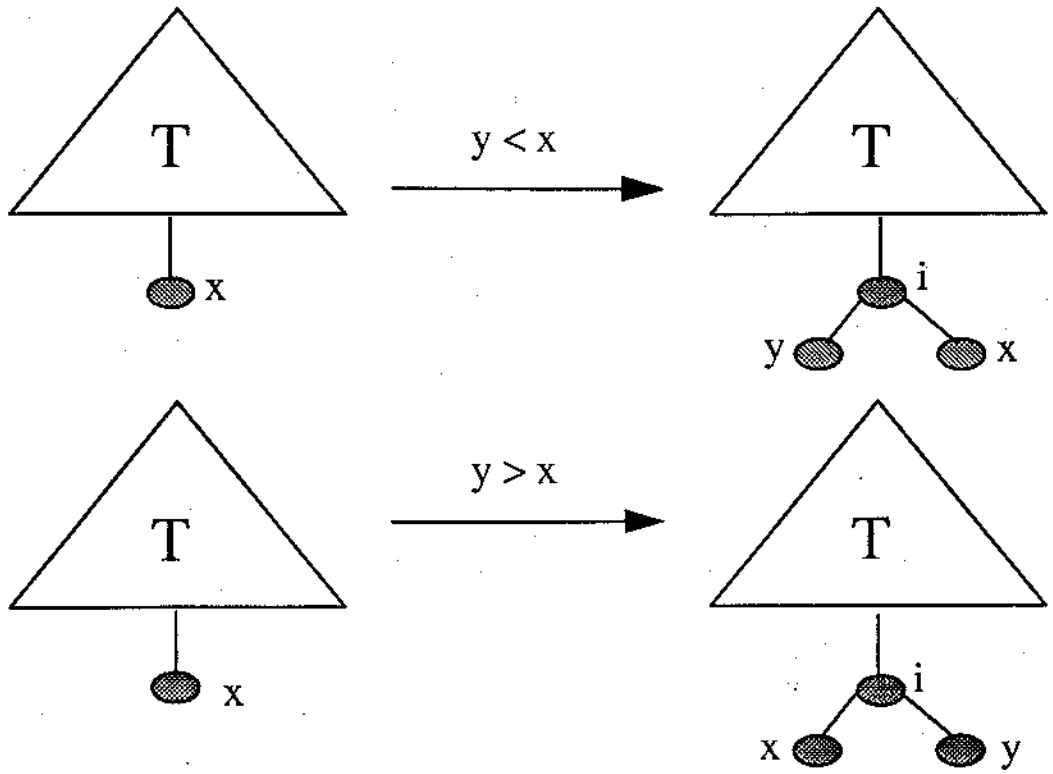

Figure 6. (a) Insertion of a key y into a Range AVL tree from [Lamo95b].

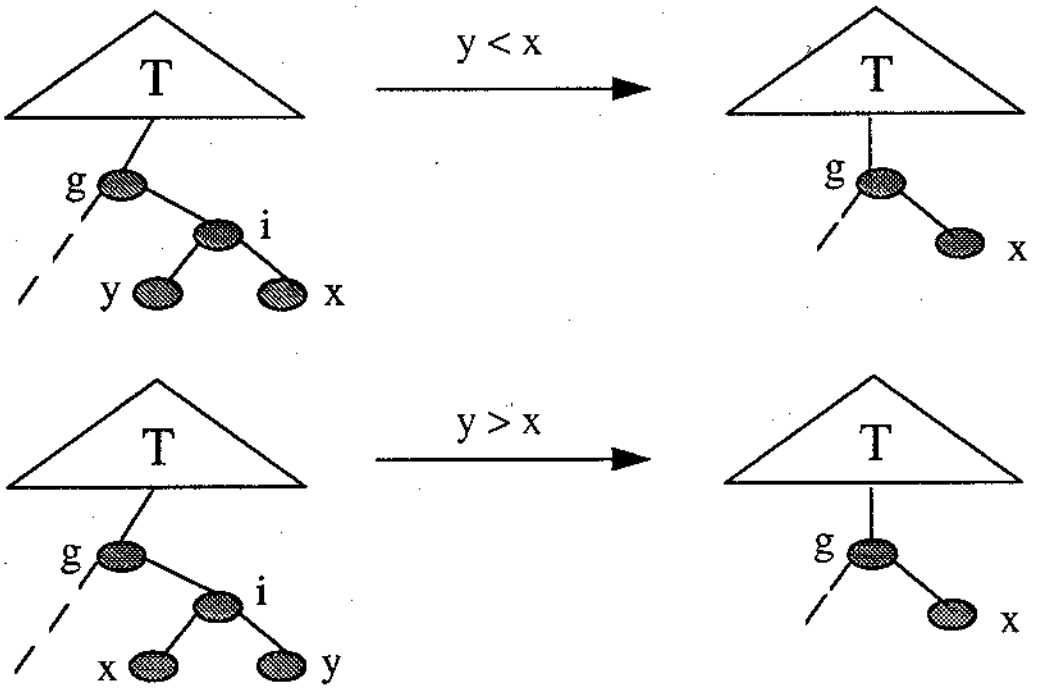

Figure 6. (b) Deletion of a key y from a Range AVL tree from [Lamo95b].

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 special cases that arise in the delete procedure. These are illustrated in Figure 7. They are essentially the two standard single rotations with the only difference being that a primary node of the rotation which is the child of the root node has a balance factor indicating equal height ('-' or 0).

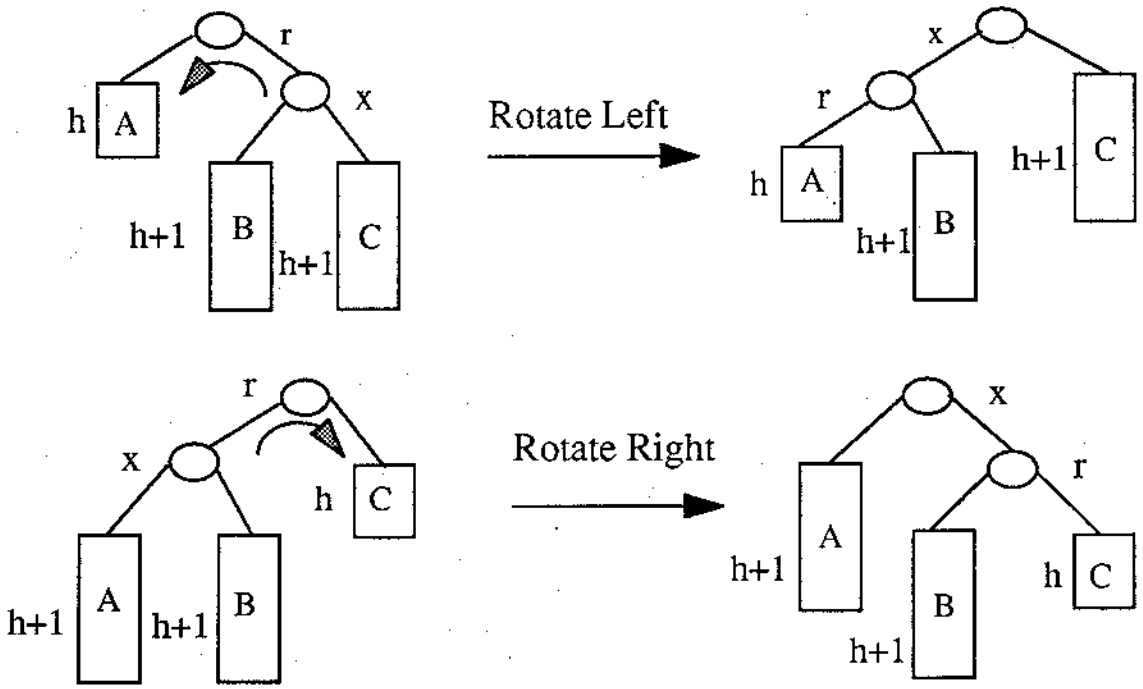

Figure 7. Additional rotations needed for the Range AVL tree from [Lamo95b].

The second modification that we must make is that all of the leaf level nodes appear in a doubly-linked list in sorted order, i.e. the leaf nodes are threaded. This modification is simple both to impose and maintain as a leaf node can never be a primary node of a rotation as the restriction that each node must have 0 or 2 children implies that

we are dealing with a restricted class of AVL trees where the root node of the rotation is at least the third interior node from the leaf level.

This allows us to deduce that the threaded linked list at the leaf level can only be changed by the insertion or deletion of a node and this implies that the maintenance of the doubly-linked list is simple. A straightforward modification to the insert and delete procedures suffices to maintain the threads.

We refer the reader to [Lamo95b] for the complete insertion and deletion algorithms and the corresponding Pascal source code. We also note that the worst-case cost functions are asymptotically those of the AVL tree.

4.2 The k-d Range AVL Tree

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 values. Therefore, at each interior node of our Range AVL tree in dimension k, we must have a pointer to a 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 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 in a dimension and through the dimensions of the structure to give us our dynamic k-d range tree data structure.

Complications arise when we need to use 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 the primary nodes of the rotation. To correct this problem, we rotate the pointers to the (k+1)-dimensional trees as well and use partial rebuilding to rebuild the (k+1)-dimensional trees that remain invalid after the rotation. When we rotate the pointers, we never have to do more rebuilding than that required to rebuild the (k+1)-dimensional trees that are attached to the child nodes after the rotation, and, for single rotations, we need only do half of that (i.e., only one tree structure has to be rebuilt). The rotations used are illustrated in Figure 8 and the reader is referred to [Lamo95b] for the complete algorithms and corresponding Pascal code.

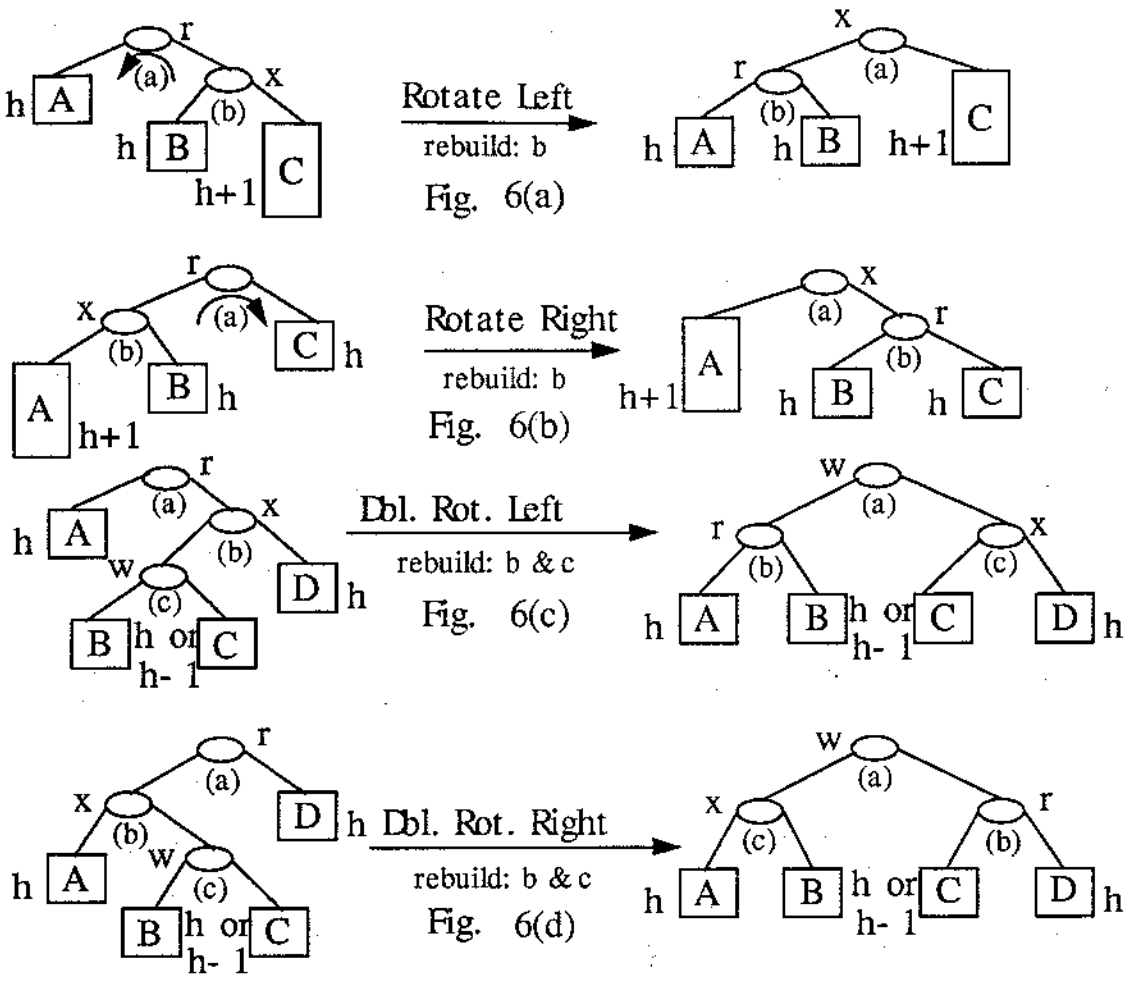

Figure 8. Rotations for the k-d Range AVL tree from [Lamo95b].

The techniques used to transform the AVL tree into a dynamic range tree provide insight into the similarities between different tree structures and may prove useful in transforming and dynamizing other (binary search) tree structures for more complicated tasks. The algorithms illustrate a recursive technique that may be useful for defining and working with other k-dimensional data structures. It has already been used in the definition and construction of the k-d Range DSL which we explore in the next chapter.

4.3 Analysis of The k-d Range AVL Tree

A complete analysis of the k-d Range AVL tree is lengthy and we are content to summarize the important theorems and results and refer the reader to [Lamo95b] for lengthy proofs and non-essential details. A 2-d Range AVL tree is illustrated in Figure 9.

Theorem 1: The time, Q(n,k), to perform a range search on a k-d Range AVL tree is $\Theta(\lg^k n + t)$, where t is the number of data points found in the specified range.

Proof:

The search algorithm used is identical to that used for Bentley's range tree [Bent80a] and this is known to require $\Theta(\lg^k n + t)$ time to answer a range query. The reader is referred to [Same90] and [Lamo95b] for additional details. ■

Corollary 1: A Range AVL tree partial match query, $Q_s(n,k)$, requires $\Theta(\lg^s n + t)$ time.

Proof:

By construction of the k-d Range AVL tree structure, each interior node in dimension i (i < k) has a pointer to a Range AVL tree structure in dimension i+1 that contains all the points in the subtree, for which it is the root node, ordered on the i+1 coordinate. As such, attached to the root node of every Range AVL tree structure in dimension i (i < k) is a Range AVL tree ordered on the i+1 coordinate values.

If dimension i is one of the s dimensions that we search, then the search algorithm proceeds normally in the Range AVL tree structures in that dimension. However, if dimension i is not one of the s dimensions to be searched, then we merely proceed from

the root node of the Range AVL tree structure in dimension i to the root node of the Range AVL tree structure in dimension i+1. In this way we search structures in only s of the k dimensions of the structure and it is straightforward to verify that this requires only $\Theta(\lg^s n)$ time by construction of the k-d Range AVL tree. ■

Fact: The time, $Q_M(n,k)$, to perform a member query in a k-d Range AVL tree is $\Theta(\lg n)$.

By definition of the structure, if a data point is in the structure then it is indexed by the first dimension. Since all coordinates within a dimension are unique, by definition, one can verify that a member query can be completed in $\Theta(\lg n)$ time. ■

For reference, the following are the datapoints in the dataset on which the k-d range tree of Figure 9 was built: (65, 90), (125, 110), (180, 25), (55, 155), (140, 125), (25, 195), (95, 60), (45, 131), (15, 175), (75, 205), and (85, 115).

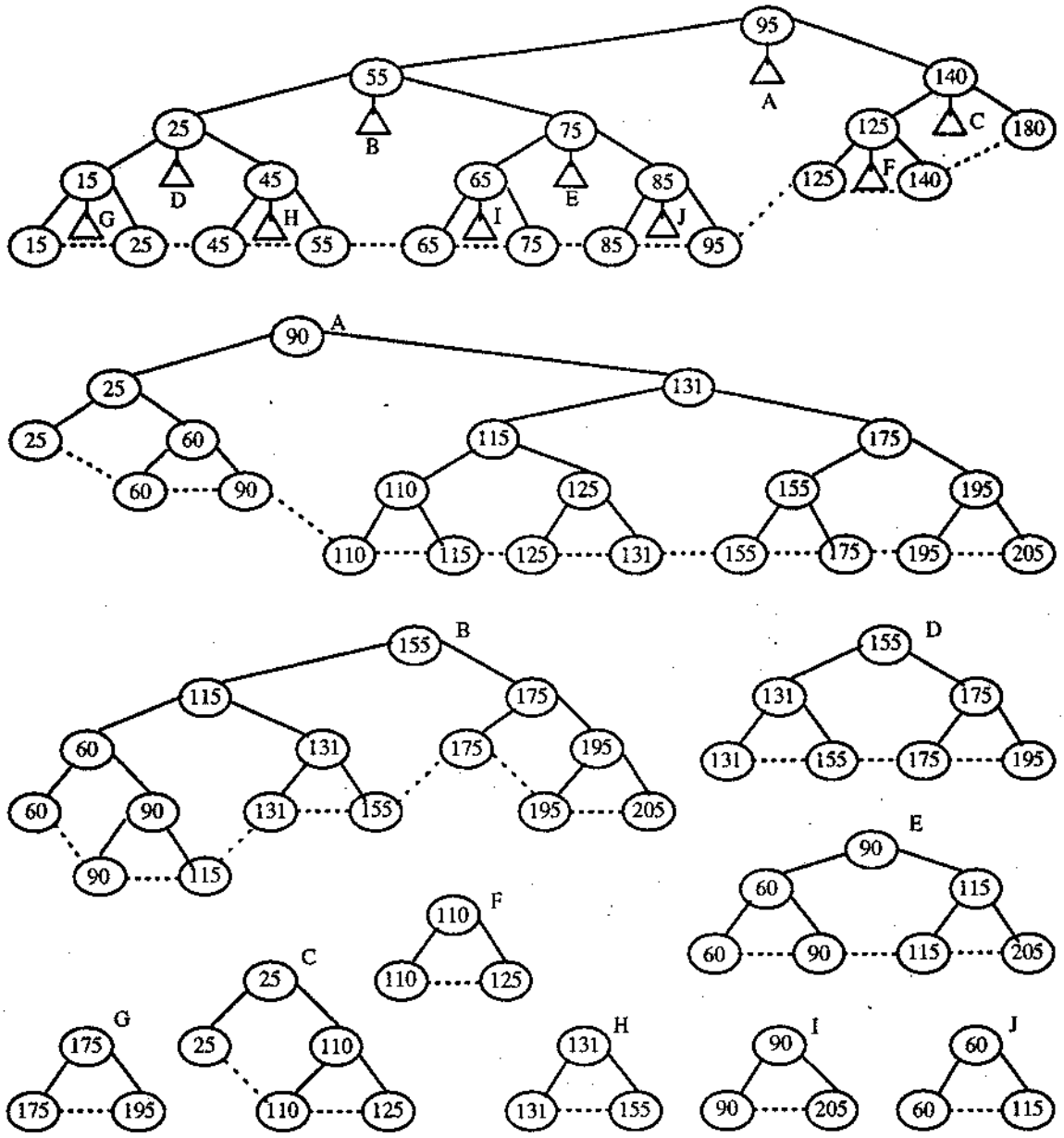

Figure 9. A 2-d Range AVL Tree from [Lamo95b].

The following lemma is used in determination of the storage cost and the worst case rebuilding cost, which are necessary for one to determine the update cost functions and preprocessing cost.

Lemma 1: The number of nodes in dimension k in a k-d Range AVL tree is $\Theta(n \lg^{k-1} n)$.

Proof:

The result is obvious for dimension one. It turns out that the calculation required for the k-d case is quite involved, so we refer the reader to [Lamo95b] for a complete calculation. In summary, we have $2^x \, C_{k-2}^{x+k-2}$ structures of $n/2^x$ points in dimension k or

$$\Theta(n \sum_{x=0}^{\lg n - 1} \left(C_{k-2}^{x+k-2} \right)) \tag{4.1}$$

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

An asymptotic approximation on equation (4.1) evaluates to $\Theta(n \lg^{k-1} n)$ and this gives us $\Theta(n \lg^{k-1} n)$ nodes in dimension k, as we set out to prove. ■

We now prove our amortized worst case bound for insertion in two steps. We define a simple insertion as an insertion which does not cause a rotation to be performed and then prove that the time required for a simple insertion, denoted $I_S(n,k)$, is bounded by $\Theta(\lg^k n)$. After this, we prove that the time required for any necessary rebuilding over a sequence of n insertions, $R_{SI}(n,k)$, is $\Theta(n \lg^{k-1} n)$. We are then able to prove that our amortized worst-case insertion cost function, $I(n,k)$, is $\Theta(\lg^k n)$. We can then go on to prove our worst case bound for deletion analogously. Note that we define a simple deletion in a manner which is analogous to the definition of a simple insertion.

Theorem 2: A simple insertion, $I_S(n,k)$, in a Range AVL tree requires $\Theta(\lg^k n)$ time.

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 and changing the necessary pointers to accomplish the insertion require only constant time. This gives us $I_S(n,1) = \Theta(\lg n)$, as expected.

The proof that $I_S(n,k) = \Theta(\lg^k n)$ follows from the proof that $Q(n,k) = \Theta(\lg^k n + t)$. In dimension k we must insert the new point into a total of $\Theta(\lg^{k-1} n)$ Range AVL tree structures associated with the Range AVL tree structures we inserted the new point into in dimension (k-1). As it takes $\Theta(\lg n)$ time to insert the point into a single Range AVL tree structure, it takes $\Theta(\lg^k n)$ 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 in a k-d Range AVL tree is $\Theta(n \lg^{k-2} n)$.

Proof:

In the worst case insertion, a rotation occurs at the root node of the tree in dimension 1. We find that we have to do rebuilding of two (k-1)-d Range AVL tree structures in dimension 2 that consist of roughly $n/2$ nodes each. We can use the trick of presorting found in [Bent80a] and rebuild a k-dimensional range tree structure of n nodes in $\Theta(n \lg^{k-1} n)$ time as the data set can be assumed static for the duration of the rebuilding

(see [Lamo95b]). This implies that a single insertion causes at most $\Theta(n \lg^{k-2} n)$ rebuilding to take place, as was desired. ■

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

Proof:

The worst case insertion is guaranteed to occur once, and only once, in a sequence of $\Theta(n)$ insertions when we start with n data points in the Range AVL tree (see [Fred81] and [Will85a]). Likewise, the second worst case insertion is guaranteed to occur twice, and only twice, in a sequence of $\Theta(n)$ 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 $\Theta(n)$ insertions.

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

$$\sum_{i=1}^{\lg n - 1} 2^{i} \left(\left(\frac{n}{2^i} \right) \lg^{k-2} \left(\frac{n}{2^i} \right) \right) \qquad (4.2)$$

and this summation is bounded by $\Theta(n \lg^{k-1} n)$ (see [Lamo95b]). 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 $\Theta(n \lg^{k-1} n)$. ■

Theorem 3: The amortized time, $I(n,k)$, required to perform a normal insertion into a k-d Range AVL tree is $\Theta(\lg^k n)$.

Proof:

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

Theorem 4: A simple deletion, $D_S(n,k)$, in a Range AVL tree requires $\Theta(\lg^k n)$ time.

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 and changing the necessary pointers require constant time. This gives us $D(n,1) = \Theta(\lg n)$, as expected.

The proof that $D_S(n,k) = \Theta(\lg^k n)$ follows from the proof that $Q(n,k) = \Theta(\lg^k n + t)$. In dimension k we must remove the point from a total of $\Theta(\lg^{k-1} n)$ Range AVL tree structures. It takes $\Theta(\lg n)$ time to remove the point from one Range AVL tree structure and thus it takes $\Theta(\lg^k n)$ 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 in a Range AVL tree is $\Theta(n \lg^{k-2} n)$.

Proof:

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 dimension 2 that consist of roughly $n/2$ nodes each. We can use the trick of presorting found in [Bent80a] and rebuild a k-dimensional range tree structure of n nodes in $\Theta(n \lg^{k-1} n)$ time as the data set can be assumed static for the duration of the rebuilding. Thus, a single deletion can cause at most $\Theta(n \lg^{k-2} n)$ rebuilding to take place. ■

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

Proof:

The worst case deletion is guaranteed to occur once, and only once, in a sequence of $\Theta(n)$ deletions when we start with $2n$ data points in a Range AVL tree (see [Fred81] and [Will85a]). Likewise, the second worst case deletion is guaranteed to occur twice, and only twice, in a sequence of $\Theta(n)$ deletions. In general, the ith worst case deletion is guaranteed to occur 2^{i-1} times, and only 2^{i-1} times, in a sequence of $\Theta(n)$ deletions.

The work involved in rebuilding over a sequence of n deletions is given by equation (4.2) and we know that the summation is bounded by $\Theta(n \lg^{k-1} n)$ 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 $\Theta(n \lg^{k-1} n)$. ■

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

Proof:

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

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

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 $\Theta(n \lg^{k-1} n)$ space. As we are performing an asymptotic analysis, we see that this term is the dominant term in our expression for storage and we thus correctly determine that we require $\Theta(n \lg^{k-1} n)$ space for storage. ■

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

Proof:

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 $\Theta(\lg^k n)$ time when we use an amortized analysis. Thus, $P(n,k)$ is $\Theta(n \lg^k n)$. ■

5 THE k-D RANGE DSL

The k-d Range DSL is a new data structure for multi-dimensional point data based on an extension of the 1-3 Deterministic Skip List (DSL) data structure of [Munr92] *projected* into k-dimensions. The data structure, labeled the k-d Range Deterministic Skip List, supports fast insertions, deletions, and range search. It is dynamically balanced and optimal for k-d range search in the class of dynamically balanced data structures.

It uses a generalization of the paradigm of multidimensional divide and conquer of [Bent80a], which is defined as follows: To store a structure representing n points in k-space, store m structures representing (approximately) n/m points in k-space and one structure of n points in (k-1)-space. It is always balanced in the sense of B-trees as the leaves are all at the same depth h where h is bounded by $\lceil \lg n / 2 \rceil \leq h \leq \lceil \lg n \rceil$.

The structure is generalizable such that it is based on an order m DSL (which has at least $\lceil m/2 \rceil$ nodes but no more than m nodes in an immediate down subtree) but we use a 1-3 DSL for simplicity and clarity in exposition and implementation.

5.1 Definition of a k-d Range DSL

The properties of the structure are:

1. Each node contains a data point which is either

   i) an actual data point if the node is a leaf node $(K_1, K_2, \ldots, K_k)$

ii) a pre-defined sentinel value (s.v.) if the node is not a leaf node

2. Each node contains two values. These are

i) minki (m) = minimum K_i value in the down subtree of the current node

ii) maxki ($\mathcal{M}$) = maximum K_i value in the down subtree of the current node

3. Each node contains three pointers; a down pointer, a right pointer, and a nextdim pointer. The down and right pointers are analogous to the down and right pointers in the 1-d DSL and the nextdim pointer points to a Range DSL structure of the points in the down subtree ordered on the K_{i+1} coordinate values.

4. Four special nodes called head, tail, bottom, and lastdim are used to indicate start and end of structure conditions. The nodes head, tail, and bottom are identical to the head, tail, and bottom nodes in the 1-3 DSL and the node lastdim lets us know that we have reached the last dimension of the structure or the leaf level.

5. All data points appear at the leaf level and only at the leaf level.

6. The skip list is ordered by the i coordinate values where i represents the dimension that is currently being built or searched (i starts at 1 and proceeds to i = k).

7. Every gap size in the skip list is of size 1, 2, or 3.

The basic 1-d structure is somewhat similar to a 2-3-4 tree [Papa93] and the resulting structure is similar to a 2-3-4 tree of 2-3-4 trees. Figure 10 illustrates the composition of a single node and Figure 11 illustrates a 2-d Range DSL.

The node structure is seen to be strikingly similar to the node structure of the 2-d Search Skip List. In fact, the only differences are 1) the minki and maxki are the minimum and maximum coordinates in the current (and not the next) dimension and 2) the existence of an extra field for the nextdim pointer.

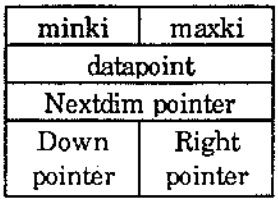

Figure 10. The node structure for a k-d Range DSL from [Lamo95a].

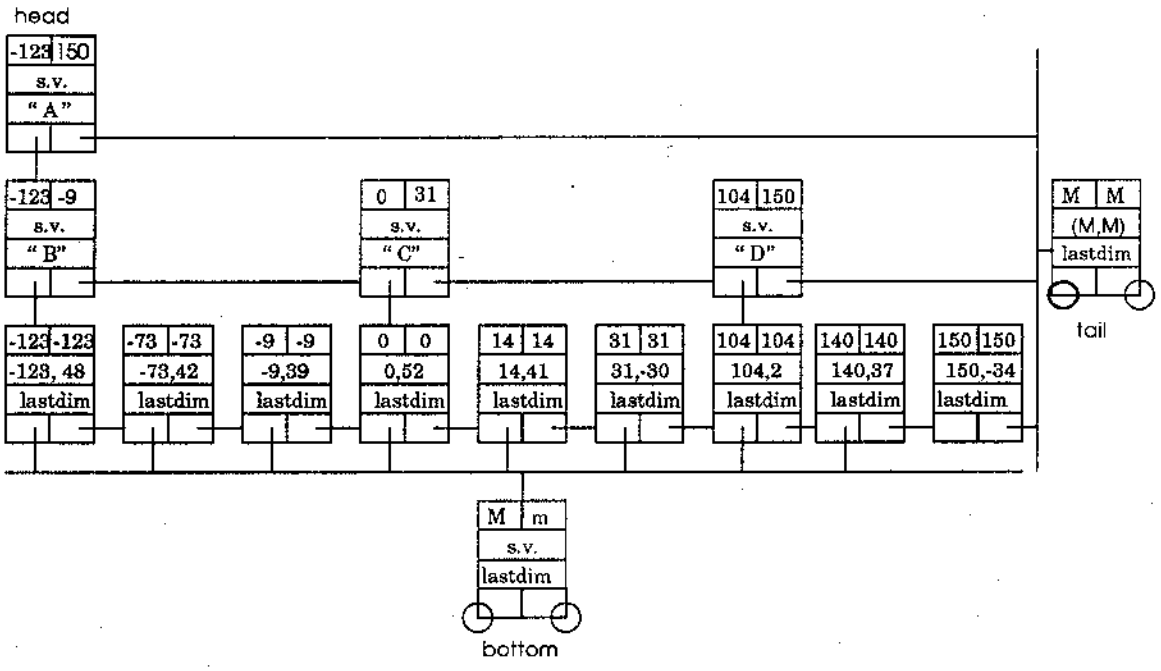

Figure 11. (a) The Range DSL structure in dimension 1 (ordered on K_1) for a 2-d Range DSL from [Lamo95a].

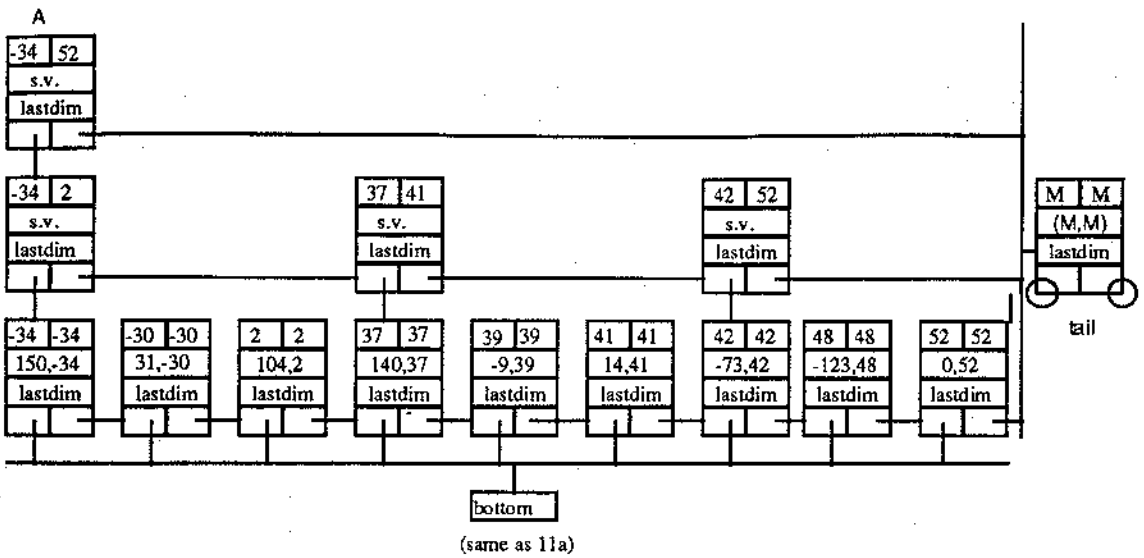

Figure 11. (b) The Range DSL structure "A" in dimension 2 (ordered on K_2) for a 2-d Range DSL from [Lamo95a].

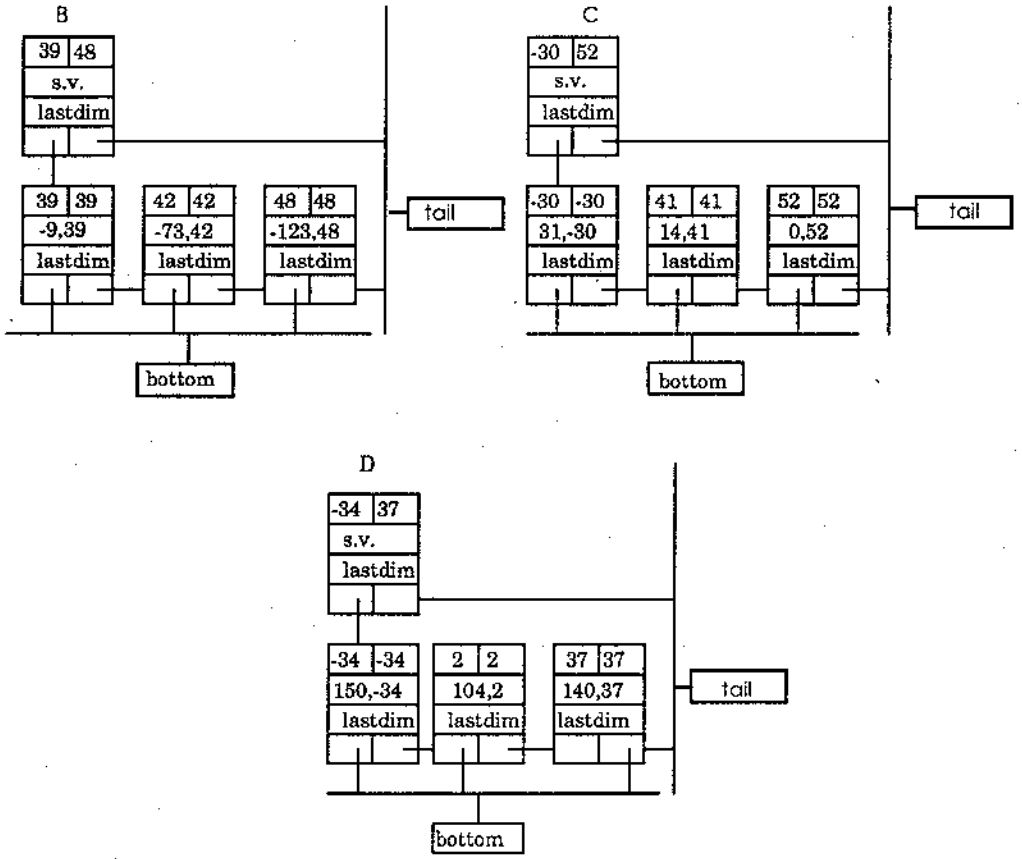

Figure 11. (c) The Range DSL structures "B", "C", & "D" (ordered on K_2) for a 2-d Range DSL from [Lamo95a].

We would like to note that a slight distinction is made between the use of the terms *k-d Range DSL*, *Range DSL structure*, and *Range DSL substructure*. Unless the context clearly specifies otherwise, we use *k-d Range DSL* to refer to the entire data structure, *Range DSL structure* to refer to a substructure in one dimension, and the term *Range DSL substructure* implies that we are also considering all *Range DSL structures* in the next and subsequent dimensions that are attached to the *Range DSL structure* that we are currently considering. Thus, Figure 11 illustrates a *k-d Range DSL*, Figure 11(a) illustrates a *Range DSL structure*, and the *Range DSL substructure* for Figure 11(a) includes the Range DSL structures in Figures 11(b) and 11(c).

The main advantage of the k-d Range DSL lies in the fact that it is much simpler to conceptualize and implement than a dynamic range tree structure. It has the additional advantage that the algorithms are highly iterative in nature. This implies that some implementations could outperform some implementations of a range tree as recursion involves extra overhead, and therefore extra time. Although the analysis that we perform is exact only to a constant factor, it should be noted that some of the hidden constants are lower than those of the k-d Range AVL tree. This is largely due to the highly iterative nature of the structure as compared to the highly recursive nature of the k-d Range AVL tree. Also, as we usually won't have a worst-case structure, the depth of the structure is usually less than that of a corresponding range tree structure and thus range searches may be faster than they would be in a range tree.

The k-d Range DSL has the additional advantages that it is extendible, well suited for range search in a parallel processing environment, and it permits multi-dimensional semi-infinite range queries in an equivalent time. It contains the 2-d Search Skip List structure of [Nick94] as a substructure and serves as a multi-dimensional equivalent to the structure. At any interior node in dimension i ($i < k$), we can access the minki and maxki values for the (i+1)st coordinate by checking the minki and maxki values of the root node of the attached substructure; and the 2-d Search Skip List structure is therefore a substructure of any two dimensions.

Future work on the structure would include the incorporation of the Interval Skip List of [Hans92] to provide a multi-dimensional alternative to the R-tree (see [Same90]). We feel that this would be a valuable index structure as [Hans94] have already made use of the Interval Skip List to accomplish selection predicate indexing for active databases.

5.2 Building a k-d Range DSL

Constructing a k-d Range DSL requires inserting each point into the k-d Range DSL sequentially, starting with an empty k-d Range DSL. Algorithm 1 of [Lamo95a] can be used for this. As the k-d Range DSL is constructed by inserting one point at a time, we discuss the time and space requirements for building the structure in the section on insertion. Note that minkey and maxkey are special constants which delineate the range that all data points must fall in.

5.3 Searching a k-d Range DSL

Algorithm 1 may be used to perform a k-d range search on a k-d Range DSL. Note that it is in some ways more involved than the search algorithm for the range tree as we don't always know at which node in the immediate down subtree we are to continue our search and must allow for this situation. This procedure may also be used for a member query which is the special case where our search interval is a data point.

The algorithm assumes that the Range DSL structure we are currently searching is rooted at *current* and that the query ranges are given in the global variables **L** and **H**. Note that *current* is set to the head node of the k-d Range DSL on the first call, that *prev* points to the previous node in our search, that *next* is set to false on the first invocation of the procedure, and that *dim* is the dimension we are currently searching (i in section 5.1).

A proof of the validity of the k-d range search algorithm can be found in [Lamo95a]. The proof is constructed by considering all of the possible cases where the search region overlaps the range of our data set (cases 2 through 5 in Figure 12) and showing that the search algorithm handles them correctly. There are 13 individual overlap cases to consider and it is straightforward to show that they are all accounted for, handled correctly, and checked in the right order by the range search algorithm.

```
Procedure Search_RDSL(current: nodeptr; prev: nodeptr; next: boolean; dim: integer);
Var snode: nodeptr;  {the node after the last node in the immediate down sub-tree}
Begin  {Search_RDSL}
{01}   IF current = prev Then  snode:= current^.right
{02}   ELSE snode:= prev^.right^.down;
{03}    IF next Then
{04}    Begin While L[dim]>current^.𝒎 Do current:= current^.right; next:= false; End;
{05}   With current^ Do
{06}    IF NotLeafLevel(current) Then
{07}     IF not ((L[dim] > 𝒎) or (H[dim] < 𝒎)) Then
{08}       IF ((L[dim] <= 𝒎) and (𝒎 <= H[dim])) Then
             Begin
{09} 2)       IF (current^.nextdim <> lastdim) Then
{10}            Search_RDSL(current^.nextdim, current^.nextdim, next, (dim+1))
{11}          Else
{12}            ReportAllPoints(current) {All points in sub-DSL in range: report them}
{13}             IF current^.right <> snode Then
{14}               Search_RDSL(current^.right, prev, next, dim);
             End
{15}       Else
{16}         IF (H[dim] <= 𝒎) Then
{17}           IF (L[dim] <= 𝒎) Then
{18} 3)           Search_RDSL(current^.down, current, next, dim)
{19}           Else
{20} 4)           Search_RDSL(current^.down, current, next:= true, dim)
{21}         Else  {H[dim] > 𝒎 and 𝒎 <= L[dim] <= 𝒎}
              Begin
{22} 5)         Search_RDSL(current^.down, current, next:= true, dim);
{23}            IF current^.right <> snode Then
{24}                  Search_RDSL(current^.right, prev, next, dim);
              End
{25}      Else
{26} 1)     Return  {No points are in range in the down subtree of the current node}
{27}    Else
        Begin  {we are at leaf level}
{28}     While ((current <> snode) and (H[dim] >= current^.𝒎)) Do
         Begin
{29}      IF InRange(current) Then Report(current^.datapoint); current:= current^.right;
         End,
        End;  {we are at leaf level}
End;  {Search_RDSL}
```

Algorithm 1: k-d Range Search in a k-d Range DSL.

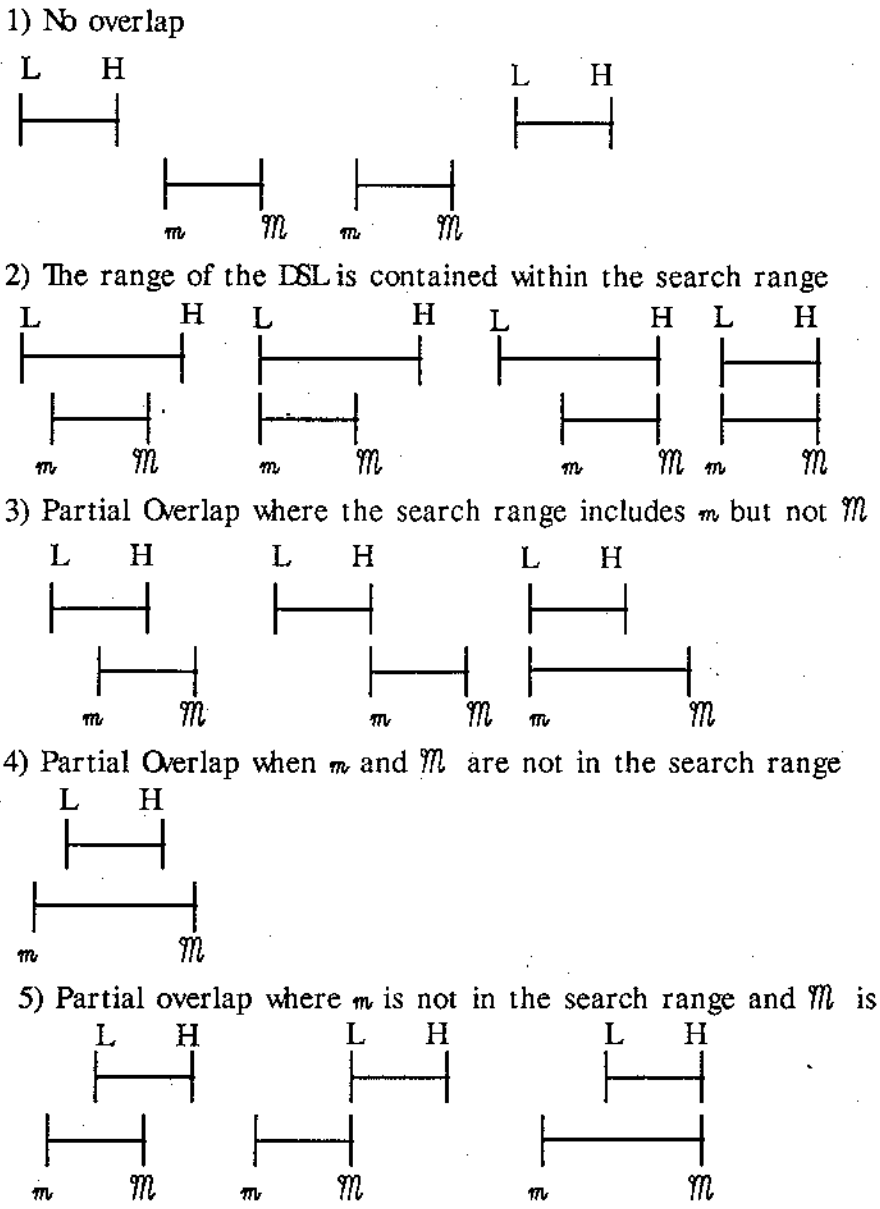

Figure 12. Overlap cases for the k-d Range DSL search algorithm from [Lamo95a].

Theorem 8: The time, $Q(n,k)$, required to perform a k-d range search on a k-d Range DSL is $\Theta(\lg^k n + t)$.

Proof:

The 1-d case is equivalent to searching a standard 1-3 DSL and we know that this takes $\Theta(\lg n + t)$ time from [Munr92].

The 2-d case is slightly more involved. The worst case is when we have a worst case DSL structure (see Figure 13) in the first dimension (all gaps are of size 1 and the height of the DSL is $\lceil \lg n \rceil$). In this instance, we have 2^{ℓ} nodes per level ℓ (where the head node is taken to be level 0) and our resulting Range DSL structure is very similar to the range tree. We see that the worst case range search implies that we search a Range DSL structure in the next dimension for one node at every level but the head node and leaf level. For instance, in Figure 13, we search nodes **e** and **f** in the next dimension and then conclude our search by checking the node **g** which contains a single data point. We thus search a total of $(\lg n - 1)$ Range DSL structures in the next dimension, each at a search time bounded by $\Theta(\lg n)$, for an overall search time of $\Theta(\lg^2 n + t)$ in dimension 2.

An inductive proof shows that $Q(n,k) = \Theta(\lg^k n + t)$ for a worst case k-d Range DSL structure and the reader is referred to [Lamo95a] for the details. ■

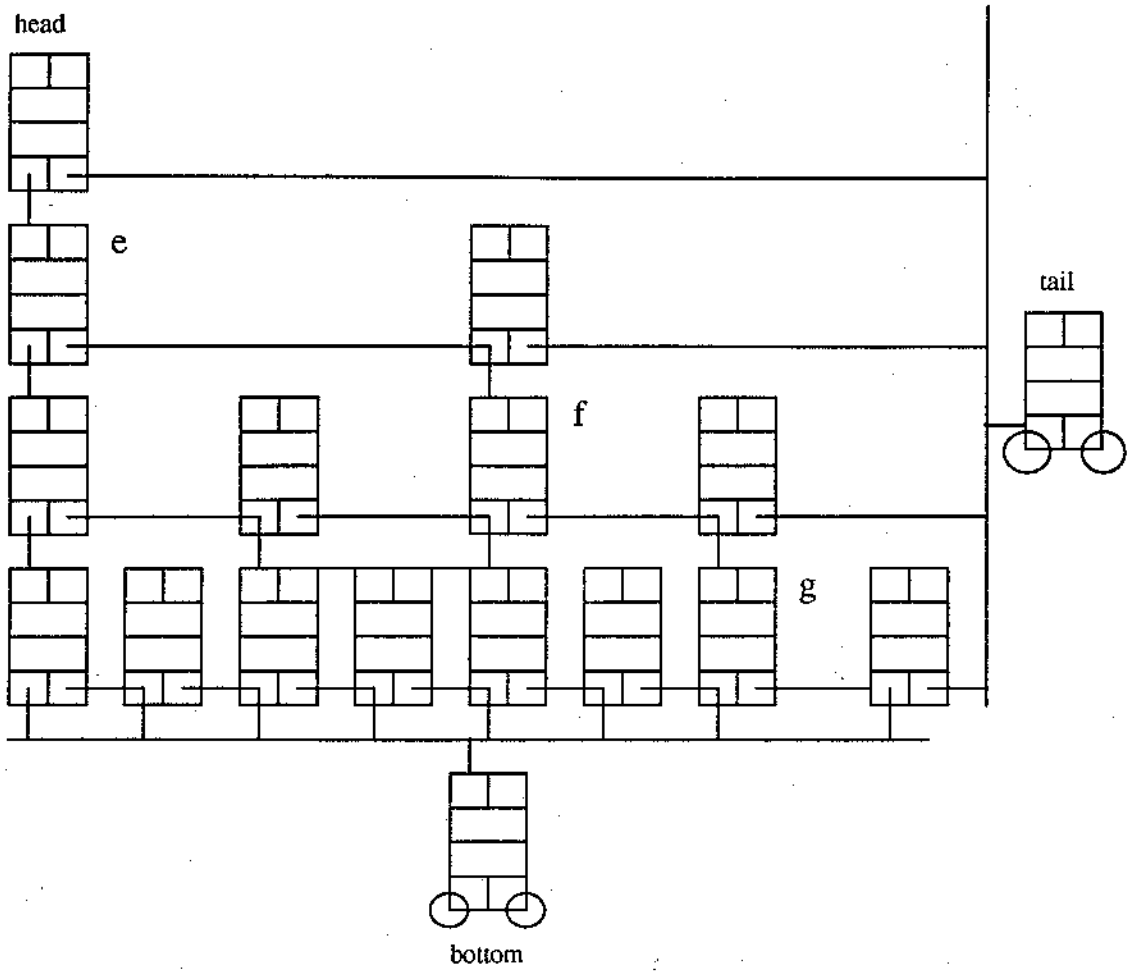

Figure 13. A worst case Range DSL structure from [Lamo95a].

Corollary 2: A partial match query, $Q_s(n,k)$, in a k-d Range DSL requires $\Theta(\lg^s n + t)$ time.

Proof:

By construction of the k-d Range DSL structure, each node in dimension i ($i < k$) has a pointer to a Range DSL structure in dimension i+1 that contains all the points in the subtree, for which it is the root node, ordered on the i+1 coordinate. As such, attached to the root node of every Range DSL structure in dimension i ($i < k$) is a Range DSL ordered on the i+1 coordinate.

If dimension i is one of the s dimensions that we search, than the search algorithm proceeds normally in the Range DSL structures in the dimension. However, if dimension i is not one of the s dimensions to be searched, then we merely proceed from the root node of the Range DSL structures in dimension i to the root node of the Range DSL structures in dimension i+1. In this way we search structures in only s of the k dimensions of the structure and it is straightforward to verify that this requires only $\Theta(\lg^s n)$ time by construction of the k-d Range DSL. ■

Fact: The time, $Q_M(n,k)$, to perform a member query in a k-d Range DSL is $\Theta(\lg n)$.

By definition of the structure, if a data point is in the structure then it is indexed by the first dimension. Since all coordinates in a dimension are unique, by assumption, one can verify that a member query can be completed in $\Theta(\lg n)$ time. ■

5.4 Insertion in a k-d Range DSL

Insertion proceeds similar to that of the 1-3 DSL and the 2-d Search Skip List of [Nick94] with the only difference being that we must also insert the node into the appropriate Range DSL substructures in the next dimension. Note that the algorithm implicitly relies on the existence of the global variables head, tail, bottom, and lastdim as previously defined along with the concepts of down subtrees and gap size. In addition, we assume that all of the K_i coordinates are unique and that the data point isn't already in the k-d Range DSL.

The (top down) insertion algorithm, in summary, is as follows:
(D is the node being inserted and h is the height of the skip list in the current dimension)

1. Start at the head node of the Range DSL.
2. If the gap we are about to drop into is of size 1 or 2, determine where to drop and alter the m or M value of the dropping node as appropriate.
3. If the gap is of size 3, we first raise the middle element of the gap by allocating a new node C to create two gaps of size 1. We then drop as appropriate and alter the m and M values of the dropping node and the node C as necessary. If raising an element implies the existence of two elements at depth 0, we must raise the height of the skip list by creating a new header node.
4. If we are not in the last dimension, we must insert the new node D into the Range DSL structure rooted at the nextdim field of the last node visited at each level except for the leaf level.
5. When we reach the bottom level, we insert a new element of height 1. This new element has a m and M value = K_i.

This algorithm allows only gaps of sizes 1 and 2 on the path being traced down to the leaf level and the resulting skip list with a newly inserted element is indeed a valid 1-3 skip list. In addition, since the m and M values are always determined with respect to the new point's K_i coordinate value, the m and M values for a node are always correct.

We find that a complete analysis of the k-d Range DSL is rather lengthy, so we are content to summarize the important theorems and results and refer the reader to [Lamo95a] for lengthy proofs, calculations, and non-essential details in our analysis.

The following lemma is used in determination of the storage cost and the worst case rebuilding cost which are necessary for one to determine the update cost functions and preprocessing cost.

Lemma 6: The number of nodes in dimension k in a k-d Range DSL is $\Theta(n \lg^{k-1} n)$.

Proof:

This is easy to see for dimension 1. Assuming a worst case DSL (see Figure 13), we have the maximum ($\lg n + 1$) levels which gives us our maximum of $2n - 1$ nodes which gives us our result of $\Theta(n)$ nodes in the 1-d Range DSL, as desired.

The k-d case turns out to be much more involved to calculate, but upon careful investigation, where we assume that all of the Range DSL structures are worst case DSL structures, we find that we have $2^x \, C_{k-2}^{x+k-2}$ structures of $n/2^x$ points in dimension k. This expression is equivalent to equation (4.1), as the number of nodes is bounded by twice the number of points, and an asymptotic approximation on equation (4.1) gives us our result. As the calculation is involved and lengthy, the reader is referred to [Lamo95a] for a detailed calculation. The summation in equation (4.1) can be approximated by $\Theta(\lg^{k-1} n)$ as the largest term in the summation is of $\Theta(\lg^{k-2} n)$ and there are $\Theta(\lg n)$ terms in the summation. This approximation thus gives us $\Theta(n \lg^{k-1} n)$ nodes in dimension k. ■

We now prove our amortized worst-case bound for insertion in two steps. We define a simple insertion as an insertion which does not cause rebuilding to be performed and then prove that the time required for a simple insertion, denoted $I_S(n,k)$, is bounded by $\Theta(\lg^k n)$. After this, we prove that the time required for any necessary rebuilding over a sequence of n insertions, $R_{SI}(n,k)$, is of $\Theta(n \lg^{k-1} n)$. We are then able to prove that our amortized worst-case insertion cost function, $I(n,k)$, is of $\Theta(\lg^k n)$. We can then go on to

prove our worst case bound for deletion analogously. Note that we define a simple deletion analogous to the definition of a simple insertion.

Theorem 9: A simple insertion, $I_S(n,k)$, in a k-d Range DSL requires $\Theta(\lg^k n)$ time.

Proof:

The proof depends on the observation that the time required for insertion of a node into a 1-d Range DSL depends on the time required to find the location where the new point is to be inserted as the operations of creating a new node and inserting a node require constant time (as only a constant number of values and pointers need to be changed). This tells us that insertion takes $\Theta(\lg n)$ time in the 1-d case.

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

Lemma 7: The time, $R_I(n,k)$, to do necessary rebuilding for a single insertion in a k-d Range DSL is given by $\Theta(n \lg^{k-1} n)$.

Proof:

In the worst case insertion a split propagates all the way up to the head node in dimension 1 and we find that we have to do rebuilding of two Range DSL substructures in the next dimension corresponding to each level of the Range DSL structure in dimension

1, except for the head node, which we propagate up after creating a new level consisting of two nodes to go between it and the level it used to connect to in order to avoid doing unnecessary work.

The largest Range DSL substructures that we have to rebuild are at depth 1 and they contain roughly $n/2$ points each. Rebuilding of a Range DSL substructure attached to depth 1 takes $\Theta(n \lg^{k-2} n)$, as we can use the trick of presorting found in [Bent80a] to rebuild a k-d Range DSL structure in $\Theta(n \lg^{k-1} n)$ time. This time bounds the time needed to rebuild the Range DSL substructures in the next dimension at all other levels and, since we have $\Theta(\lg n)$ levels, we can thus bound the rebuilding time for the worst case insertion at $\Theta(n \lg^{k-1} n)$, as desired. ■

Lemma 8: The time, $R_{SI}(n,k)$, to perform all necessary rebuilding over a sequence of n insertions in a k-d Range DSL is bounded by $\Theta(n \lg^{k-1} n)$.

Proof:

The worst case insertion is guaranteed to occur once, and only once, in a sequence of $\Theta(n)$ insertions when we start with n data points in the k-d Range DSL (see [Fred81] and [Will85a]). Likewise, the second worst case insertion is guaranteed to occur twice, and only twice, in a sequence of $\Theta(n)$ insertions. The pattern continues until we hit the base case for rebuilding which is when we rebuild at depth (h-1) and this occurs at most $n/2$ times in a sequence of $\Theta(n)$ insertions.

Examining the pattern, we see that the work involved in rebuilding over a sequence of n insertions is closely approximated by equation (4.2). The summation is

bounded by $\Theta(n \lg^{k-1} n)$ ([Lamo95a]) and 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 $\Theta(n \lg^{k-1} n)$. ■

Theorem 10: The amortized time, $I(n,k)$, to perform a general insertion in a k-d Range DSL is $\Theta(\lg^k n)$.

Proof:

The proof follows directly from Theorem 9 and Lemma 8. A simple insertion is bounded by $\Theta(\lg^k n)$ and the amortized rebuilding cost for a general insertion is $\Theta(\lg^{k-1} n)$. Thus, an insertion can be seen to take $\Theta(\lg^k n)$ time. ■

Theorem 11: The space, $S(n,k)$, required for storage of a k-d Range DSL is $\Theta(n \lg^{k-1} n)$.

Proof:

Each node in the k-d Range DSL has three pointers and three data fields and thus occupies O(1) space so we need only determine the maximum number of nodes in the k-d Range DSL to determine the space required. From Lemma 6, we know that dimension k requires $\Theta(n \lg^{k-1} n)$ space. As we are performing an asymptotic analysis, we see that this term is the dominant term in our expression for storage and we thus conclude that we require $\Theta(n \lg^{k-1} n)$ space for storage, $S(n,k)$. ■

Theorem 12: The time, P(n,k), required to dynamically build a Range DSL is $\Theta(n \lg^k n)$.

Proof:

As a k-d Range DSL is constructed dynamically by inserting one point at a time into an initially empty k-d Range DSL, we compute the construction time of $\Theta(n \lg^k n)$ as each insertion is bounded by $\Theta(\lg^k n)$ and there are n of them. ■

5.5 Deletion in a k-d Range DSL

Deletion proceeds similar to that of the 1-3 DSL and the 2-d Search Skip List of [Nick94], with the only difference being that we must also delete the node from the appropriate Range DSL substructures in the next dimension. Again, note that the algorithm implicitly relies on the global variables head, tail, bottom, and lastdim as previously defined and the concepts of immediate down subtree and gap size. In addition, we assume that all of the K_i coordinates are unique and that the data point isn't already located in the k-d Range DSL.

The (top down) deletion algorithm, in summary, is as follows:
(D is the node being deleted and h is the height of the skip list)

1. Start at the head node of the Range DSL.
2. If the gap we are about to drop into is of size 2 or 3 then drop and if the m or M value equals that of the node about to be deleted, record the node in a path list P.
   If the gap we are about to drop into is of size 1 then
   IF G is not the last gap in the current level, then
      IF the following gap G' is of size 1 then
         merge G and G' (lower the separating element)
      ELSE
         remove an element from G' and insert it into G
   ELSE
      IF the preceding gap G' is of size 1 then
         merge G and G' (lower the separating element)

ELSE

remove an element from G' and insert it into G.

3. Continue until we reach the bottom level, removing the node from the associated Range DSL substructure in the next dimension for the last node visited on each level on the way down, where we remove the element of height 1. If the node to be deleted is not of height 1, we swap with its predecessor (successor).
4. Follow the pointers in the list P to reestablish the m and M values as necessary.

This algorithm allows only gaps of sizes 2 or 3 on the path being traced down to the leaf level and therefore the resulting skip list with a newly deleted element is indeed a valid 1-3 skip list. In addition, the m and M values of the nodes on the deletion path are adjusted as necessary with respect to the deleted node's m and M values so the m and M values for a node are always correct.

Theorem 13: The time, $D_S(n,k)$, to perform a simple deletion of a point in a k-d Range DSL is $\Theta(\lg^k n)$.

Proof:

The proof depends on the observation that the time required for deletion of a node in a 1-d Range DSL depends on the time required to find the node being deleted as the operations of removing a node and borrowing a node require constant time (as only a constant number of pointers and values need to be changed). Thus, the deletion of a point from a 1-d Range DSL takes $\Theta(\lg n)$ time.

The proof that $D_S(n,k) = \Theta(\lg^k n)$ follows from the proof that $Q(n,k) = \Theta(\lg^k n + t)$. In dimension k we must delete the point from a total of $\Theta(\lg^{k-1} n)$ Range DSL structures associated with the Range DSL structures that we wish to delete the point from in

dimension k-1. As it takes $\Theta(\lg n)$ time to delete the point from a single Range DSL structure, we see that it takes $\Theta(\lg^k n)$ time to delete the point from a k-d Range DSL. ■

Lemma 9: The time, $R_D(n,k)$, to do any necessary rebuilding for a single deletion in a k-d Range DSL is $\Theta(n \lg^{k-1} n)$.

Proof:

In the worst case deletion, a merge propagates all the way up to depth 1 in dimension 1 and we find that we have to do rebuilding of a Range DSL substructure in the next dimension corresponding to each level of the Range DSL structure in dimension 1.

The largest Range DSL substructure that we have to rebuild is at depth 1 and it contains roughly $n/2$ points. Rebuilding of a Range DSL substructure attached to depth 1 takes $\Theta(n \lg^{k-2} n)$ time as we can use the trick of presorting found in [Bent80a] to rebuild a k-d Range DSL structure in $\Theta(n \lg^{k-1} n)$ time. This time bounds the time needed to rebuild the Range DSL substructures in the next dimension at all other levels and, since we have $\Theta(\lg n)$ levels, we are thus able to bound the rebuilding time for the worst case deletion at $\Theta(n \lg^{k-1} n)$, as desired. ■

Lemma 10: The time, $R_{SD}(n,k)$, for all necessary rebuilding over a sequence of n deletions in a k-d Range DSL is $\Theta(n \lg^{k-1} n)$.

Proof:

The worst case deletion is guaranteed to occur once, and only once, in a sequence of $\Theta(n)$ deletions when we start with n data points in the k-d Range DSL (see [Fred81]

and [Will85a]). Likewise, the second worst case deletion is guaranteed to occur twice, and only twice, in a sequence of $\Theta(n)$ deletions. This pattern continues until we hit the base case for rebuilding which is when we rebuild at depth (h-1) and this occurs at most $n/2$ times in a sequence of $\Theta(n)$ deletions.

Looking at the pattern, we see that the work involved in rebuilding over a sequence of n deletions is given by the summation of equation (4.2) as we have to rebuild 2^i Range DSL structures of $n/2^i$ data points over the course of $\Theta(n)$ deletions. The largest term is of $\Theta(n \lg^{k-2} n)$ and so we can bound our expression at $\Theta(n \lg^{k-1} n)$, as desired. ■

Theorem 14: The amortized time, D(n,k), to perform a general deletion in a k-d Range DSL is $\Theta(\lg^k n)$.

Proof:

The proof follows directly from Theorem 13 and Lemma 10. A simple deletion is bounded by $\Theta(\lg^k n)$ and the amortized rebuilding cost for a general deletion is $\Theta(\lg^{k-1} n)$. Thus, a deletion can be seen to require $\Theta(\lg^k n)$ time. ■

6 LOWER BOUNDS

6.1 Optimal Balance

We claim that the k-d Range DSL and the k-d range tree maintain dynamic balance and that they are within a factor of $O(\lg^{k-1} n)$ of being optimally balanced. We illustrate this fact by defining the minimal optimal balance and dynamic balance cost functions. We remind the reader that we assume our records are from a commutative semigroup as we use Fredman's results [Fred81] regarding lower bounds on the worst-case cost functions.

The optimal balance cost is defined as the minimum product of the worst-case storage, preprocessing, insert, delete and range search cost functions. The dynamic balance cost function, a factor in the optimal balance cost function, is defined to be the minimum product of the worst-case range search, insert, and delete cost functions.

Before illustrating the minimal optimal balance cost and dynamic balance cost, we first show that the minimal preprocessing cost function, $P_O(n,k)$, and minimal update cost functions, $I_O(n,k)$ and $D_O(n,k)$, are optimal. From [Fred81] we know that the work involved in a single insertion or deletion is dependent not only on the number of data points in the data structure, but also on the sequence of previous insertions and deletions. From [Fred81], we know that any sequence of n update operations in k-d must take $\Omega(n \lg^k n)$ time in the worst case. This tells us that $P_O(n,k)$ must be of $\Theta(n \lg^k n)$ and that, using an amortized worst case analysis, $I_O(n,k)$ and $D_O(n,k)$ must be of $\Theta(\lg^k n)$.

[Fred81] tells us that the minimum k-d range search time, $Q_O(n,k)$, is $\Omega(\lg^k n)$, where we use the basic definition of range search which specifies only the location, and not the reporting, of the points in range. We also know that the minimal storage requirement, $S_O(n,k)$, is always $\Omega(n)$ as each datapoint must be stored at least once.

Theorem 15: The minimal optimal balance cost function for a data structure is given by $B_{MO}(n,k) = \Omega(n^2 \lg^{4k} n)$.

Proof:

The minimal optimal balance cost is given by:

$P_O(n,k) * S_O(n,k) * I_O(n,k) * D_O(n,k) * Q_O(n,k) =$

$\Theta(n \lg^k n) * \Omega(n) * \Theta(\lg^k n) * \Theta(\lg^k n) * \Theta(\lg^k n) = \Omega(n^2 \lg^{4k} n)$ ■

We define the optimal balance cost as the product of the worst-case relevant cost functions since a product captures interdependencies between functions that summations cannot. The inspiration behind the definition comes from [Bent80b] in which Bentley asked: what bounds exist on the product of $P(n,k)$, $S(n,k)$, and $Q(n,k)$? By restating the question in the dynamic framework and using Fredman's model, we are able to determine a class of data structures that are in many ways optimal for use in a dynamic environment. The class of structures that we are referring to are the class of dynamically balanced structures which are given by the following theorem and optimal for k-d range search.

Theorem 16: The dynamic balance cost of a structure is given by $\mathcal{D}_O(n,k) = \Theta(\lg^{3k} n)$.

Proof:

The dynamic balance cost is given by:

$$I_O(n,k) * D_O(n,k) * Q_O(n,k) = \Theta(\lg^k n) * \Theta(\lg^k n) * \Theta(\lg^k n) = \Theta(\lg^{3k} n)$$ ■

Thus, when we say that a structure is dynamically balanced, we are really saying that the three dynamic cost functions of range search, insertion, and deletion are of optimal complexity (within the model that we use).

We use Fredman's [Fred81] commutative semigroups as the analysis they permit is more realistic than that obtainable from decision trees, providing us with a combinatorial setting which yields stronger bounds than those which can be derived from decision trees. It makes the more realistic assumption that the data points are from an ordered key space where each data point is a member of an arbitrary commutative semigroup. The cost is accounted for in terms of arithmetics over the semigroups and Knuth's [Knut73] definition of an orthogonal range query is equivalent to Fredman's formulation where we choose our commutative semigroup to have set union as its addition operation. We note that Fredman assumes that the cost of a single semigroup addition is 1 time unit. This implies that the lower bounds obtained are actually stronger than those achievable in practice.

Theorem 17: The k-d range tree and k-d Range DSL are dynamically balanced.

Proof:

The dynamic balance cost of the structures is given by

$I_O(n,k)*D_O(n,k)*Q_O(n,k) = \Theta(\lg^k n) * \Theta(\lg^k n) * \Theta(\lg^k n) = \Theta(\lg^{3k} n) = \mathcal{D}_O(n,k).$ ■

Theorem 18: Assuming an optimally balanced data structure, $\Theta(\lg^k n + t)$ time complexity is optimal for a k-d range search which reports the t points in range.

Proof:

The data structures in the class of optimally balanced data structures are dynamic and based on Fredman's commutative semigroups [Fred81]. This tells us that $\Omega(\lg^k n)$ is the lower bound on the worst case time complexity of a single range search which locates the points in range. As structures exist where $O(\lg^k n)$ is the worst case search time, $\Theta(\lg^k n)$ is optimal for a range search which locates the points in range. Thus, $\Theta(\lg^k n + t)$ is optimal for a k-d range search which reports the t points in range. ■

The proof that Fredman uses to show that $\Omega(n \lg^k n)$ is the lower bound on a sequence of n dynamic operations, which implies that $\Omega(\lg^k n)$ is the lower bound on a single dynamic operation, is difficult and involved. As such, we do not summarize it in this thesis. In its place, we present an illustrative argument as to why $\Theta(\lg^k n)$ must be the lower bound for k-d range search in an optimally balanced structure.

Any optimally balanced data structure must require preprocessing of $\Theta(n \lg^k n)$, an update time of $\Theta(\lg^k n)$ (using an amortized analysis), and storage of $\Omega(n)$ as these costs are minimal for any such structure. This gives us a minimal balance cost of $\Omega(xn^2\lg^{3k} n)$, where x stands for range query time. This tells us that our range query cost function must be $O(\lg^k n)$ if we are to maintain a minimal balance. From [Fred81] we know that $\Omega(\lg^k n)$ is minimal in the *model* we use and thus $\Theta(\lg^k n)$ must be minimal for a range query on a structure that is optimally (and dynamically) balanced.

Theorem 19: The k-d range tree and the k-d Range DSL are optimal for k-d range search in the class of dynamically balanced data structures.

Proof:

Both the k-d range tree and the k-d Range DSL have a k-d range query cost function of $\Theta(\lg^k n + t)$, where t is the reporting time, and are thus optimal for k-d range search in the class of dynamically balanced data structures. ■

Theorem 20: Our minimal optimal balance cost, $B_{MO}(n,k)$, of $\Omega(n^2\lg^{4k} n)$ holds in the random access model of computation as well.

Proof:

Bentley's k-ranges [Bent80b] are representative of array-based data structures that exist under the random access model of computation and not under the pointer model of computation that is normally assumed. Multi-level non-overlapping k-ranges are the most efficient k-range structures (see [Bent80b] or Sec. 3.2.5) and, therefore, we use them to show that our minimal optimal balance cost holds in the array model of computation.

We know that the cost functions for $P_O(n,k)$, $S_O(n,k)$, and $Q_O(n,k)$ are given by $P_O(n,k) = \Theta(\ell^{k-1} n \lg n)$, $S_O(n,k) = \Theta(\ell^{k-1} n)$, and $Q_O(n,k) = \Theta(\ell^{k-1} n^{1/\ell} \lg n)$. If we can show that $P_O(n,k) * S_O(n,k) * Q_O(n,k)$ is $\Omega(n^2 \lg^{2k} n)$, then, since $I_O(n,k)$ and $D_O(n,k)$ are always $\Theta(\lg^k n)$ in a dynamization of any structure, by [Fred81], our theorem will be proved. The product $P_O(n,k) * S_O(n,k) * Q_O(n,k)$, equal to $\Theta(\ell^{2k-3} n^2 n^{1/\ell} \lg^2 n)$, is minimized when $\ell = \lg n$ (giving us a range tree structure). Letting $\ell = \lg n$ gives us $P_O(n,k) * S_O(n,k) * Q_O(n,k) = \Theta(n^2 \lg^{3k-1} n)$ and this is $\Omega(n^2 \lg^{2k} n)$. ■

6.2 Minimal Update Cost

In sections 4.3, 5.4, and 5.5 we have seen that the worst case dynamic updates require $\Theta(n \lg^{k-1} n)$ time to complete. This is because, as pointed out by [Luek78], rebalancing a node may require that the associated structure be completely rebuilt.

An obvious and important question for one to ask is the following: Are there any modifications to the structure or associated algorithms for k-dimensions that one could use to improve the worst-case update time?

We could try the technique of clustering as described in [Bent79b] to reduce the update time. The idea is to decompose our structure into n/c clusters of c elements and store each cluster as the data structure of our choice. This reduces the update time to

$U(n,k) = \Theta(c\ \lg^{k-1}c)$ while increasing our query time to $Q(n,k) = \Theta(\frac{n}{c}\ \lg^k c + t)$. This tells us that we must have $c \geq n^{1/2}$ as $U(n,k)$ must be greater or equal to $Q(n,k)$ as the update time is dependent upon the query time. This implies that $U(n,k)$ is $\Omega(n^{1/2}\lg^{k-1}n^{1/2})$ and this is not a considerable improvement for large n.

In [Bent80c], Bentley and Saxe in their paper on decomposable searching problems have also noted that we can never go below the $n^{1/2}$ limit when balancing query and update times. By balancing, we imply that decreasing one cost function below $n^{1/2}$ increases the other above $n^{1/2}$ as the product of the cost factors, n/c and c, must equal n. An idea and methodology which is similar to that of clustering appears in [VanL80].

We might also try amortized rebuilding where we postpone excessive rebuilding until necessary to distribute the $\Theta(n\ \lg^{k-1}n)$ rebuilding caused by a sequence of $\Theta(n)$ updates over the next $\Theta(n)$ updates in an attempt to keep the rebuilding needed for a single update minimal. This is the approach taken by Willard and Lueker in [Will85a]. A structure is only completely rebuilt when necessary and when a structure needs to be rebuilt, it is marked as invalid and placed into a priority queue. Substructures attached to lower levels have a greater chance of becoming invalid and are given higher priority.

After an update has been performed and any structures that have become invalid marked as such and added to the priority queue, a structure(s) is (are) removed from the queue and (partially) rebuilt until approximately $c\ \lg^k n$ work has been performed for some

predetermined constant c. Willard and Lueker's structures maintain the invariant that if a substructure attached to a node is invalid, then the substructures attached to the child nodes must be valid. Thus, priority is assigned to the structures needing to be rebuilt such that the substructure of a node should be completely rebuilt before it is again needed in a rotation which would invalidate another substructure (and the invariant) in the rotation.

In theory, according to [Will85a], the complex method of Willard and Lueker can be used to insure that no single update requires more than $O(\lg^k n)$ time to complete, but in practice one cannot use the method for reasons that are clarified in Theorem 21. It is an open question whether a good implementation of the amortization algorithm would provide us with a better upper bound on the worst-case update function.

Theorem 21: The worst case cost for a single update, $U(n,k)$, is $\Omega(n)$ for $k > 1$.

Proof:

The amortization method, which is the most efficient method, fails for large n since one must rebuild an entire range tree structure in one dimension once it is started. In the worst case we have to rebuild a k-1 dimensional substructure containing $\Theta(n)$ points. Rebuilding of a structure of $\Theta(n)$ points requires $\Theta(n)$ overhead which is greater than $\Theta(\lg^k n)$ for even moderate n (e.g. $n > (k+c)^{(k+c)}$ for any $c \geq 2$).

The reason we must rebuild an entire structure in one dimension is that we would otherwise have to maintain the $\Theta(n)$ overhead between updates to maintain the state of

rebuilding for a partially rebuilt structure. An overhead of $\Theta(n)$ implies an $\Omega(n)$ rebuilding cost and thus, in the worst case, a single update is of $\Omega(n)$ time complexity. ■

This theorem invalidates the claim of Willard and Lueker (which was restated in [Edel81]), who state that a single update can be completed in $O(\lg^k n)$ time [Will85a], as we have shown the minimal rebuilding time in the worst case to be at least of $\Omega(n)$ as a range tree structure in one dimension must be completely rebuilt once it is started.

However, the worst case rarely happens, and a modification of Willard and Lueker's method for the k-d Range AVL tree and k-d Range DSL, which postpones unnecessary rebuilding, may give update algorithms that are noticeably faster in implementation than the analysis would lead one to expect.

Willard and Lueker state in their paper that the constant factor c is quite large and that the bookkeeping required for the approach is extensive and expensive. As can be seen from our analysis, the method could be inefficient for all but the largest data sets and our analysis agrees with their claim. We can thus conclude that in all but the most exceptional of circumstances, the worst case update does indeed require $\Theta(n \lg^{k-1} n)$ time to complete and that our analysis of the k-d Range AVL tree and k-d Range DSL is valid.

7 EXPERIMENTAL RESULTS

7.1 Experimental Setup

The k-d Range AVL tree and the k-d Range DSL were experimentally analyzed using the methodical approach described below, in an attempt to facilitate an accurate comparison of their actual run times. We are interested in the average insertion time, the average deletion time, and the average search time as the worst-case is not only rare for large structures but difficult to specify. In this section we outline the general approach that is used in the experimental analysis of the k-d Range AVL tree in section 7.2 and the k-d Range DSL in section 7.3. Section 7.4 compares their performance.

Both structures were tested on randomly generated data sets which were uniformly distributed throughout a k-dimensional space. The structures were analyzed based on their construction times, destruction times, and range search times. It is the construction and destruction times that allow us to approximate the "hidden" constants of our big-oh analysis of the insertion and deletion algorithms that we performed in Chapters 4 and 5. Our analysis of the update times is based on these constants as they allow us to make a direct comparison of the structures.

The construction time is defined as the time to build a structure by inserting n points sequentially into an initially empty structure. The destruction time is defined as the time to dispose of a structure by deleting one data point at a time until we are left with an

empty structure. The range search time is simply the time to execute a k-d range search for a given k-d orthogonal range query.

The range search time is recorded for 8 different query windows that cover 0, 5, 10, 25, 50, 75, 90, and 100 % of the search space, respectively. The query windows are restricted such that they are completely contained within the search space (inclusively) and contain the required percentage of data points. Algorithm 2 was used to generate the random query windows.

```
Procedure GenerateQueryWindow(L, H, qws, n, k, dp);
{L and H store the lower left and upper right corners of the query window}
{qws is the fraction of points that must be contained within the query window}
{n is the number of points in the structure and k is the number of dimensions}
{dp is an array which holds the n data points}

{random(n) returns a random integer between 1 and n, inclusive}

Var r: integer; {stores a random dimension between 1 and k}
    i: integer;  {a loop control variable}

Begin
 r:= random(k);
 L[r]:= random (n - qws*n); {randomly determine left endpoint of the interval for dim r}
 H[r]:= L[r] + (qws*n) - 1;   {the left endpoint uniquely determines the right endpoint}

 For i:= 1 to k Do        {now we must determine the intervals for the other dimensions}
   IF i <> r Then         {don't re-calculate the interval for dimension r}
     L[i]:= min{dp[j,i]: L[j,r] ≤ dp[j,r] ≤ H[j,r], 1 ≤ j ≤ n}
     H[i]:= max{dp[j,i]: L[j,r] ≤ dp[j,r] ≤ H[j,r], 1≤ j ≤ n}
   EndIF
 EndFor

End;
```

Algorithm 2. Procedure for generating a random query window for a k-d range search.

To allow for an accurate analysis, each structure is constructed, searched for the 8 different query windows, and destroyed five times. Also, for each structure that is built, each range search is performed 1000 times in order to bring the search time into the high millisecond range so that accurate timings may be obtained as a single range query is extremely fast, requiring only a few milliseconds for the largest structures we build.

Three test runs were performed on 1, 2, 3, and 4 dimensional data for data sets that ranged in size from 100 to 1700 data points, as illustrated in Table 1. A single test run consisted of five passes over 25 data structures whose sizes are given in Table 1. Also, the data sets were regenerated with each test run. A single pass consisted of constructing, searching, and destroying a single data structure.

Table 1. The sizes of the randomly generated data sets that were used in initial testing.

dim	Number of data points in the randomly generated datasets								
1	100	300	500	700	900	1100	1300	1500	1700
2	100	300	500	700	900	1100	1300	1500	
3	100	300	500	700	900				
4	100	300	500						

After the data obtained from the initial three test runs was analyzed and interpreted, we performed three further test runs, which consisted of three passes over 12 data structures whose sizes are given in Table 2, to yield further results which we could use to determine the accuracy of our initial test runs and interpretations. These further randomly generated, uniformly distributed data sets also contained 1, 2, 3, and 4 dimensional data and ranged in size from 1000 to 3500 data points.

Table 2. The sizes of the randomly generated data sets that were used in further testing.

dim	Number of data points in the randomly generated datasets		
1	2500	3000	3500
2	2000	2500	3000
3	1500	2000	2500
4	1000	1500	2000

The timings were taken using SUN SPARCompiler Pascal's built in function *sysclock*, accurate to at least 1/60 of a second, which keeps track of the amount of system time used and returns an integer result in *milliseconds*. All of our construction and destruction times are given in milliseconds and all of our search times are in microseconds. Due to the size of $n \lg^k n$ as compared to the actual run times, the constants for the insertion and deletion times of the data structures, calculated according to the following equation, were normalized by a factor of 10,000:

$$c = \frac{\text{actual run time}}{1 \text{ millisecond}} \times \frac{10{,}000}{n(lg^k\, n)} = \frac{\dfrac{\text{actual run time}}{1 \text{ millisecond}}}{\dfrac{n(\lg^k n)}{10{,}000}} \qquad (7.1)$$

The factor used in the computation of c, $n \lg^k n$ / 10,000, can be found in Tables 3 and 4.

Table 3. Values of $n \lg^k n$ /10,000 for the values of n and k in Table 1.

dim	100	300	500	700	900	1100	1300	1500	1700
1	0.07	0.25	0.45	0.66	0.88	1.11	1.34	1.58	1.82
2	0.44	2.03	4.02	6.25	8.67	11.23	13.91	16.70	
3	2.93	16.72	36.04	59.10	85.07				
4	19.48	137.55	323.09						

Table 4. Values of $n \lg^k n$ /10,000 for the values of n and k in Table 2.

dim	#pts as per Table 2	#pts as per Table 2	#pts as per Table 2
1	2.82	3.47	4.12
2	24.05	31.85	40.03
3	176.17	263.72	359.55
4	986.38	1858.76	2891.94

Approximating the constant for the range search cost function, which is c $\lg^k n$ + t, is extremely difficult for a number of reasons. The range search function is highly dependent on the time needed to report the t data points in range and the time needed to open and close the output file, both of which are extremely difficult to determine accurately. The time to report a point is in the microsecond range and the time to open a file for writing and close it requires only a few milliseconds. Also, $\lg^k n$ is very small, especially compared to the values of $n \lg^k n$, and is in the microsecond range for many of the searches we perform. As we cannot accurately time the reporting time t, the time to open and close a data file (as these make calls to the operating system which, in practice, take variable time), or the time to perform the actual k-d range search, we do not attempt to approximate the constant for the range search function and simply analyze the average run times (which are highly dependent on k and t).

The method we use for analyzing the search times is to determine whether or not the search times increase appropriately with an increase in the size of the query window. For example, when the query window increases from 5% to 10% coverage, the search time should approximately double if the search is performed on the same structure under the assumption that t dominates the search time (which our results seem to indicate).

Also, we plot the search times relative to k*t. If t dominates the search time, then a graph which plots $\frac{\text{search time}}{k*t}$, for each range search performed in dimension k with t points in range, should be approximately constant (as k*t should grow linearly relative to the search time when t dominates the actual search time). This provides us with an easy visual check of whether or not the search algorithm is behaving as expected.

Our methodology is to average the construction, search, and destruction times for each structure built (where one structure corresponds to one data file) and determine our constants and approximate run times from these averages. Since we have built a number of structures for each *n* and k, we can be reasonably confident in our analysis. As building, searching, and destroying 483 data structures implies that we have many tables of *raw* data for each of the k-d Range AVL Tree and the k-d Range DSL structures (which summarize construction times, destruction times, and search times), we only summarize the results in the sections that follow and refer the reader to the appendices for additional data and results.

The testing was performed on *sol*, a Sun SPARC 1000 workstation, under SUN OS 5.3. *Sol* has 640 MB of RAM and 820 MB of hard disk space available for use as virtual memory. *Sol* is located in the Computing Services Department of the University of New Brunswick and is a two-processor multi-user system which runs at 50 MHZ. *Sol* is

heavily used throughout the campus and testing took place between Thursday January 4 and Wednesday January 16, 1996 when usage was usually below normal.

The pseudo-code for the test runs can be found in Algorithm 3. It contains pseudo-code for the main driver used in testing both the k-d Range AVL tree and the k-d Range DSL along with pseudo-code for the driver procedures to accomplish the construction, searching, and destruction of the data structures. The complete test driver code for the k-d Range AVL Tree can be found in Appendix U and the complete test driver code for the k-d Range DSL can be found in Appendix V.

```
Driver Test_k-d_Data_Structure;

Var  i, j, k, l: integer;          {loop control variables}
     numruns: integer;             {how many runs are we performing? - we perform 3 runs}
     numpasses: integer;           {how many times we build, search, and destroy a structure?}
                                   {5 passes for initial testing and 3 passes for further testing}
     numdim: integer;              {number of dimensions - we test for 1 to 4 dimensions}
     numqw: integer;               {number of query windows - we use 8 for all of our test runs}
     numsearches: integer;         {number of searches we perform per query window}
                                   {we perform 1000 searches for every query window}
     nd: array of integer;         {holds number of data sets per dimension}
                                   {given by Tables 1 and 2}
     basenumpts: integer;          {number of data points is always a multiple of this number}
                                   {we use basenumpts = 100 for all of our testing}
     ndpts: array of array of integer; {indicates number of data points in a data set}
                                   {given by Tables 1 and 2}
   datapoints: array/list of coordinates; {the data points for our structure}
                                          {uniformly distributed and randomly generated}

Begin  {Main Driver}

for i:= 1 to numruns do
  for j:= 1 to numdim do
    for k:= 1 to nd[j] do
      for l:= 1 to numpasses do
          begin  {main testing loop}
            read in datapoints from a datafile;

            BUILD the structure and output the construction time
            SEARCH the structure for the given query windows and save the results
            DESTROY the structure and output the destruction time

          end;  {main testing loop}

End.  {Main Driver}
```

Algorithm 3. (a) Main driver routine for testing the k-dimensional data structures.

```
Sub-driver BUILD

  Var getstarttime, getendtime, m: integer;

    Begin
      getstarttime:= sysclock;
      for m:= 1 to ndpts[j,k]*basenumpts do
        insert datapoints[m];
      getendtime:= sysclock;
      output(getendtime - getstarttime);
    End;

  Sub-driver DESTROY

  Var getstarttime, getendtime, m: integer;

    Begin
      getstarttime:= sysclock;
      for m:= 1 to ndpts[j,k]*basenumpts do
        delete datapoints[m];
      getendtime:= sysclock;
      output(getendtime - getstarttime);
    End;

  Sub-driver SEARCH

  Var getstarttime, getendtime, m, n: integer;

    Begin
      open output file for writing;
      for m:= 1 to numqw do
        begin
          getstarttime:= sysclock;
          for n:= 1 to numsearches do
            search the data structure for the given query window and output
            the points found in range to the output file;
          getendtime:= sysclock;
          output(getendtime - getstarttime);
        end;
      close the output file;
    End;
```

Algorithm 3. (b) Sub-driver routines for testing the k-dimensional data structures.

7.2 An Analysis of the k-d Range AVL Tree

In this section we experimentally analyze the k-d Range AVL Tree of chapter 4. The construction and destruction times for all five passes of all three runs for randomly generated datasets whose sizes are given in Table 1 in section 7.1 can be found in Appendix A and their averages can be found in Appendix B. Tables 7 and 8 below summarize the construction and destruction times for the initial test runs.

The constants, as computed by equation 7.1, corresponding to the construction and destruction times of Appendix A can be found in Appendix C and the constants corresponding to the averages of Appendix B and their standard deviations can be found in Appendix D and are summarized below, by dimension, in Tables 5 and 6.

Table 5
Average insertion constants for initial testing of the range AVL tree

dimension	run 1	run 2	run 3	all runs
2	36.47	45.53	26.35	36.12
3	47.00	31.04	34.44	37.49
4	24.64	34.46	36.25	31.79

Table 6
Average deletion constants for initial testing of the range AVL tree

dimensions	run 1	run 2	run 3	all runs
2	4.69	12.30	2.80	6.59
3	1.33	0.82	1.23	1.13
4	1.33	0.24	0.37	0.65

Table 7
Average construction times (milliseconds) for initial testing of the range AVL tree

dim	Construct time averages for data set sizes given in Table 1								
1	2	2	7	9	4	10	7	15	18
2	44	92	87	70	114	96	181	237	
3	23	305	595	980	1523				
4	16	1874	4968						

Table 8
Average destruction times (milliseconds) for initial testing of the range AVL tree

dim	Destruct time averages for data set sizes given in Table 1								
1	1	2	7	14	3	12	11	13	19
2	0	8	17	18	30	23	49	42	
3	1	10	14	25	57				
4	14	28	122						

Notice that Tables 5 and 6 only contain data values for dimensions 2, 3, and 4. This is done because the construction and destruction times for dimension 1 are very low and thus inaccurate. One notices that many of the values are 0, 16/17, 33/34, and 50 and these values correspond to clock ticks of 1/60 of a second, illustrating the limitations of the timing procedure. Also, the first column (which corresponds to a data set size of 100 data points) is seen to be generally inaccurate as well and is thus also ignored in our calculations. However, the other timings can be seen to be quite accurate and the remainder of our analysis is thus carried out on these values.

From tables 5 and 6, it would appear that the constant for the insertion procedure is moderately high at about 35 and that the constant for the deletion procedure is rather low at about 2. It was expected that the insertion procedure would be slower than the deletion procedure in practice as an insertion always involves the creation of a

substructure in the next dimension while deletion always causes a substructure in the next dimension to be destroyed and building a new structure involves more work than deallocating the memory for a discarded structure. However, a difference greater than a factor of 2 or 3 certainly was not expected and is thus taken to be a significant result.

If we look at tables 9(a) and 9(b) in Appendix D which summarize the standard deviations of the constants, it appears that we can assume our approximations to be rather good as the standard deviations are fairly low, especially for the higher dimensions.

We now use the data from our second set of test runs to verify our analysis. We expect the constants to be slightly higher as larger structures require more overhead and more recursion which could considerably slow down the construction and destruction times, especially when the constant paging and swapping of the process that is bound to happen in a multi-user system is taken into account.

The construction and destruction time averages for the second set of test runs on the randomly generated datasets whose sizes are given in Table 2 can be found in Appendix M. The associated constants and standard deviations can be found in Appendix N and a summary appears below in Tables 9 and 10. Tables 11 and 12 provide the corresponding construction and destruction times.

Table 9
Average insertion constants for further testing of the range AVL tree

dimension	run 1	run 2	run 3	all runs
2	116.75	121.25	120.32	119.45
3	42.76	41.70	47.08	43.84
4	16.69	15.69	16.41	16.26

Table 10
Average deletion constants for further testing of the range AVL tree

dimensions	run 1	run 2	run 3	all runs
2	2.15	1.98	2.71	2.28
3	0.49	0.69	0.44	0.54
4	0.22	0.08	0.12	0.14

Table 11
Average construction times (milliseconds) for further testing of the range AVL tree

dim	Construct time averages for data set sizes given in Table 2.		
1	3741	2850	3098
2	4361	2991	3326
3	11168	9493	11557
4	26800	23258	26324

Table 12
Average destruction times (milliseconds) for further testing of the range AVL tree

dim	Destruct time averages for data set sizes given in Table 2.		
1	52	74	82
2	56	87	72
3	137	106	161
4	183	215	348

If we ignore dimension 2, which is reasonable given the exceedingly high standard deviation (see Tables 9(a) and 9(b) in Appendix N), then we see that the results are consistent with the results which we would expect to get and we can therefore assume that our analysis is fairly accurate. We ignore dimension 2 as the high standard deviations most likely indicate that there is a lot of noise in our data.

We now go on to analyze the search times. The average search times (i.e. the average of 1000 searches) for the initial runs are given in Table 13 and the corresponding multipliers that correspond to the increases in search time for an increase in query window size are given in Table F4 (Appendix F). Remember that, unlike the construction and destruction times, the search times are in microseconds.

The multipliers of Table F4 represent the increase in query window size from 5 - 10%, 10 - 25%, 25 - 50%, 50 - 75%, 75 - 90%, and 90 - 100% and should be roughly 2.0, 2.5, 2.0, 1.5, 1.2, and 1.1 if t dominates the search time as predicted. The xxxx's in Table F4 indicate that accurate timings were not obtained, indicating that the multipliers could not be determined.

Examination of the multipliers of Table F4 leads us to conclude that the range search procedure is correct and very efficient, behaving as predicted in our analysis. The multipliers are close to their estimated values and the search times clearly depend on both k and t. Figure 14, which plots the search times relative to k*t (i.e. it plots $\frac{\text{search time}}{k * t}$ for each range search on each data set in each dimension), exemplifies this fact as the plots are all very close to straight lines, as expected.

Table 13
Average search times (microseconds) for initial testing of the k-d Range AVL Tree

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Actual Search Times in microseconds							
1	100	4	11	42	77	163	288	357	386
	300	4	48	113	280	551	912	1110	1297
	500	2	98	183	490	1005	1506	1854	2026
	700	6	146	307	694	1383	2018	2651	2842
	900	3	173	326	847	1843	2818	3390	3888
	1100	4	223	452	1091	2178	3372	3998	4589
	1300	4	230	501	1267	2612	4056	4767	5520
	1500	9	318	622	1558	2930	4532	5580	6313
	1700	7	329	668	1674	3553	5331	6337	7108
2	100	4	29	71	149	308	478	596	674
	300	3	95	206	561	989	1481	1776	2051
	500	9	146	311	862	1676	2523	3126	3597
	700	4	206	472	1158	2421	3636	4371	4976
	900	4	273	621	1573	3085	4615	5608	6401
	1100	7	354	700	1840	3806	5519	7002	7777
	1300	9	351	676	1857	3673	5498	6781	5764
	1500	4	361	671	1629	3273	4911	5561	4705
3	100	9	58	89	254	507	780	910	1025
	300	11	146	304	780	1542	2453	2843	3414
	500	13	247	552	1399	2670	4063	4954	5638
	700	9	348	690	1783	3581	5348	6500	6791
	900	5	438	952	2225	4496	6743	8121	7276
4	100	8	70	148	367	703	1081	1273	1444
	300	4	214	542	1106	2307	3410	4231	4629
	500	20	369	692	1701	3395	5324	6382	6816

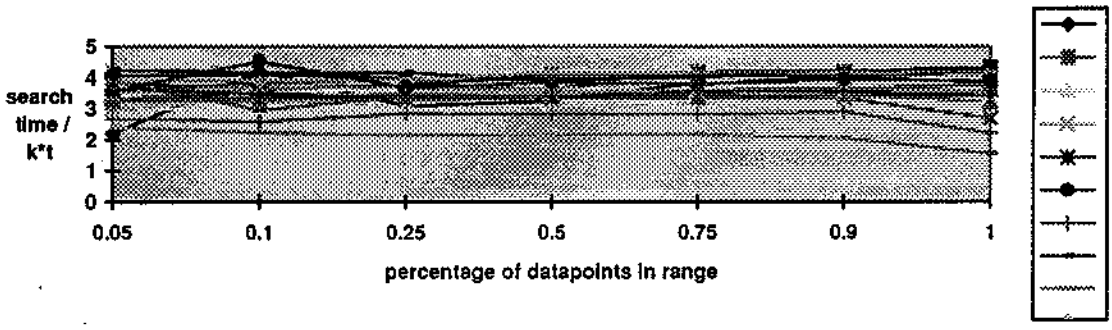

Figure 14. A plot of the search times relative to k*t for the k-d Range AVL Tree.

Tables 14 and P4 (Appendix P), below, provide the average search times obtained from further testing along with the corresponding multipliers. Examination of the multipliers in Table P4 and Figure 15 which plots the search times relative to k*t leads us to conclude that our initial results are correct and accurate.

TABLE 14
Average search times (microseconds) of further testing of the k-d Range AVL Tree

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average search times in microseconds							
1	2500	7	554	1072	2754	5385	8309	9991	11339
	3000	9	580	1265	3431	6513	9793	11991	13298
	3500	2	730	1389	3865	7544	11620	13793	15209
2	2000	9	709	1489	3531	7171	11041	13206	15476
	2500	4	880	1896	4533	8961	13863	16930	18667
	3000	5	1115	2252	5267	10750	16811	20343	22685
3	1500	9	844	1630	4261	8274	12491	15198	16878
	2000	7	1104	2267	5600	10642	16613	20446	22550
	2500	11	1048	2158	5400	10843	16098	19567	14942
4	1000	9	628	1163	2843	5678	8165	9948	11002
	1500	7	1209	2257	5363	10804	15689	18758	21324
	2000	9	1297	2526	5811	11376	17407	20428	12245

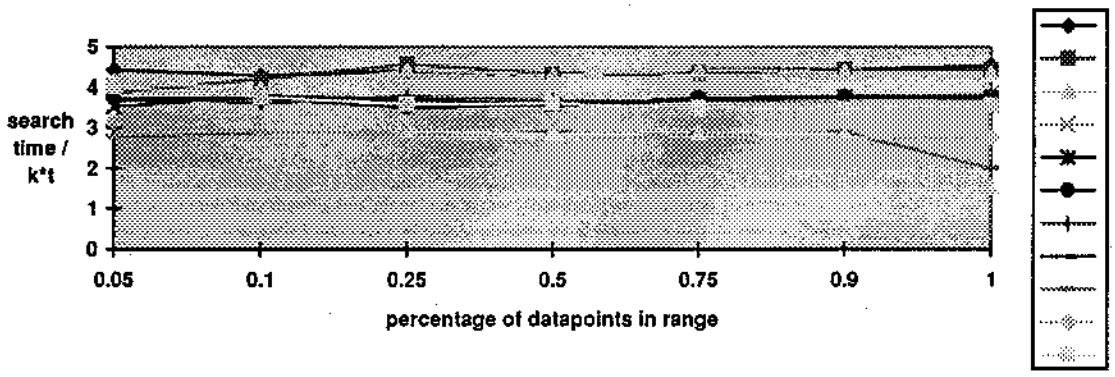

Figure 15. A plot of the search times relative to k*t for the k-d Range AVL Tree.

7.3 An Analysis of the k-d Range DSL

In this section we experimentally analyze the k-d Range DSL of chapter 5. The construction and destruction times for all five passes of all three runs for randomly generated datasets whose sizes are given in Table 1 in section 7.1 can be found in Appendix G and their averages can be found in Appendix H. Tables 17 and 18 below summarize the construction and destruction times for the initial test runs.

The constants, as computed by equation 7.1, corresponding to the construction and destruction times of Appendix G can be found in Appendix I and the constants corresponding to the average construction and destruction times of Appendix H and their standard deviations can be found in Appendix J and are summarized in tables 15 and 16.

Table 15
Average insertion constants for initial testing of the k-d Range DSL

dimensions	run 1	run 2	run 3	all runs
2	23.31	23.38	23.75	23.48
3	13.90	20.58	18.53	17.67
4	8.92	7.56	8.82	8.44

Table 16
Average deletion constants for initial testing of the k-d Range DSL

dimensions	run 1	run 2	run 3	all runs
2	44.25	45.04	55.81	48.37
3	43.77	44.83	38.64	42.41
4	19.42	17.54	19.90	18.96

Table 17
Average construction times (milliseconds) for initial testing of the Range DSL

dim	Construct time averages for data set sizes given in Table 1								
1	2	0	4	10	5	5	8	20	22
2	4	38	82	148	192	294	358	457	
3	96	454	510	924	1166				
4	163	1288	2425						

Table 18
Average destruction times (milliseconds) for initial testing of the Range DSL

dim	Destruct time averages for data set sizes given in Table 1								
1	0	4	2	13	13	11	17	17	14
2	6	34	131	196	326	794	1102	1171	
3	89	595	1546	2547	4087				
4	155	2513	6345						

Notice that the above tables only contain data values for dimensions 2, 3, and 4. This is done because the building and destruction times for dimension 1 are very low and thus inaccurate, again due to the limitations of the timing procedure.

From tables 15 and 16, it would appear that the constant for the insertion procedure is reasonable at around 17 and that the constant for deletion is moderately high at around 37. Although the procedures should require the same amount of time on average as they both require the same amount of rebuilding in the worst case, it was expected that the deletion procedure might be slightly slower than the insertion procedure as it involves extra overhead and requires backtracking along the deletion path when a maximal or minimal value in a down subtree is removed. However, a difference of a factor of 2 is taken to be a moderately significant result, especially when the deletion

procedure is slower than the insertion procedure, the opposite of what we might expect considering the results obtained from our analysis of the k-d Range AVL tree.

If we examine tables 9(a) and 9(b) in Appendix J which summarize the standard deviations of the constants, it appears that we can be quite confident in our approximations of the constants as the standard deviations are fairly low.

We now use the data from our second set of test runs to verify our analysis. As with the k-d Range AVL tree, we expect the constants to be slightly higher as larger structures require more overhead which could considerably slow down the construction and destruction times, especially when the constant paging and swapping of the process that is bound to happen in a multi-user system is taken into account.

The construction and destruction time averages for the second set of test runs on the randomly generated datasets whose sizes are given in Table 2 can be found in Appendix Q. The associated constants and standard deviations can be found in Appendix R and a summary appears below in Tables 19 and 20. Tables 21 and 22 provide the corresponding construction and destruction times.

Table 19
Average insertion constants for further testing of the k-d Range DSL

dimensions	run 1	run 2	run 3	all runs
2	15.58	16.35	17.32	16.41
3	40.59	40.12	44.42	41.71
4	6.15	6.20	6.12	6.18

Table 20
Average deletion time constants for further testing of the k-d Range DSL

dimensions	run 1	run 2	run 3	all runs
2	55.19	52.50	48.58	52.09
3	339.20	247.89	349.69	312.26
4	92.66	92.28	91.09	92.01

Table 21
Construction time averages (milliseconds) for further testing of the k-d Range DSL

dim	Average construction time for data set sizes given in Table 2.		
1	22	46	39
2	383	504	700
3	3459	10620	23450
4	5085	13471	17763

Table 22
Destruction time averages (milliseconds) for further testing of the k-d Range DSL

dim	Average destruction time for data set sizes given in Table 2.		
1	31	35	30
2	844	1593	2848
3	20894	80883	187493
4	40053	196555	375012

If we ignore dimension 3 when looking at our destruction constant, which is reasonable given the exceedingly high standard deviation (see Tables 9(a) and 9(b) in Appendix R) and the high probability that all it is measuring is noise, then we see that the results are consistent with the results which we would expect to get and we can therefore assume that our analysis is fairly accurate. The high standard deviations are most likely due to other processes which required higher time slices relative to when dimensions 2 and 4 were being tested. The other processes were possibly regularly run system administration processes which occurred during some of the test runs.

Three further test runs on dimension 3 gave average insertion constants of 13.67, 33.27, 59.82, and 35.58 and average deletion constants of 8.26, 22.55, 33.20, and 21.34 which were much closer to predicted values. This confirmed our hypothesis that the previous timings were mostly measuring noise as the standard deviations of 3.67, 9.32, 12.15, and 8.38 for the insertion constants and 2.50, 5.57, 3.88, and 3.98 for the deletion constants were much lower.

We now go on to analyze the search times. The average search times for the initial runs are given in Table 23 and the corresponding multipliers that correspond to the increases in search time for an increase in query window size are given in Table L4 (Appendix L). Again we note that the search times are given in microseconds.

The multipliers of Table L4 represent the increase in query window size from 5 - 10%, 10 - 25%, 25 - 50%, 50 - 75%, 75 - 90%, and 90 - 100% and should be roughly 2.0, 2.5, 2.0, 1.5, 1.2, and 1.1. Again, the xxxx's in Table L4 indicate that accurate timings were not obtained, indicating that the multipliers could not be determined.

Examination of the multipliers of Table L4 leads us to conclude that the range search procedure is correct and very efficient, behaving as predicted in our analysis. The multipliers are close to their estimated values and the search times depend on k and t. Figure 16, which plots the search times relative to k*t, exemplifies this fact as the plots are all very close to straight lines, as expected.

Table 23
Average search times (microseconds) for initial testing of the k-d Range DSL

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average Search Times in microseconds							
1	100	10	30	59	131	272	438	551	572
	300	3	88	177	431	865	1353	1656	1791
	500	4	156	315	725	1497	2251	2736	3113
	700	10	222	442	1041	2066	3246	3995	4371
	900	3	311	572	1402	2844	4399	5269	5580
	1100	8	315	671	1684	3388	4975	6278	6740
	1300	4	404	809	2183	4279	6418	7674	8708
	1500	6	410	924	2216	4481	6537	7833	8258
	1700	7	497	944	2429	5020	7227	8858	8229
2	100	11	62	111	218	509	769	957	1053
	300	7	175	367	818	1644	2556	3078	3440
	500	17	318	579	1468	2813	4350	5076	5722
	700	6	384	727	1819	3589	5266	6353	6770
	900	17	440	898	2118	4302	6348	7527	7449
	1100	12	590	1138	2881	5567	8419	9233	9049
	1300	13	595	1180	2860	5703	8420	9114	9246
	1500	9	584	1211	2880	5806	8549	7730	6808
3	100	21	72	143	386	718	1173	1444	1522
	300	17	271	526	1257	2605	3898	4680	5080
	500	11	391	792	1878	3836	5568	6604	7170
	700	17	513	985	2462	4893	7424	8660	8887
	900	16	608	1350	3156	6187	9487	9394	9157
4	100	23	96	180	487	998	1405	1745	1855
	300	13	319	697	1511	3237	4796	5560	6472
	500	20	553	1048	2562	5418	8016	9820	10630

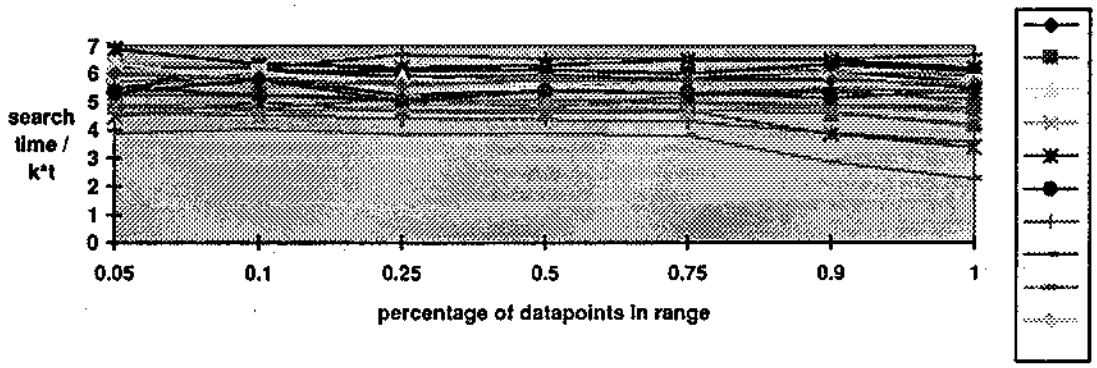

Figure 16. A plot of the search times relative to k*t for the k-d Range DSL.

Tables 24 and T4 (Appendix T), below, provide the average search times obtained from further testing along with the corresponding multipliers. Examination of the multipliers in Table T4 and Figure 17 which plots the search times relative to k*t leads us to conclude that our initial results are correct and accurate.

Table 24
Average search times (microseconds) for further testing of the k-d Range DSL

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	#pts	Average search times							
1	2500	4	363	801	2076	4083	6093	7176	8085
	3000	5	483	919	2376	5013	7484	8509	9739
	3500	9	595	1041	2807	5554	8658	10017	11300
2	2000	11	495	986	2548	4124	7489	9154	9130
	2500	11	561	1162	2684	5252	7702	9532	8218
	3000	11	615	1507	3624	7229	11115	10844	8357
3	1500	6	459	874	2178	4224	6300	7954	7152
	2000	15	746	1694	3723	7376	10927	11517	8128
	2500	11	830	1522	3717	7319	10900	10615	10754
4	1000	15	361	904	2080	4493	6615	7778	8286
	1500	15	619	1406	3546	6717	9841	10511	7820
	2000	20	1965	3779	7863	17219	26045	12211	11857

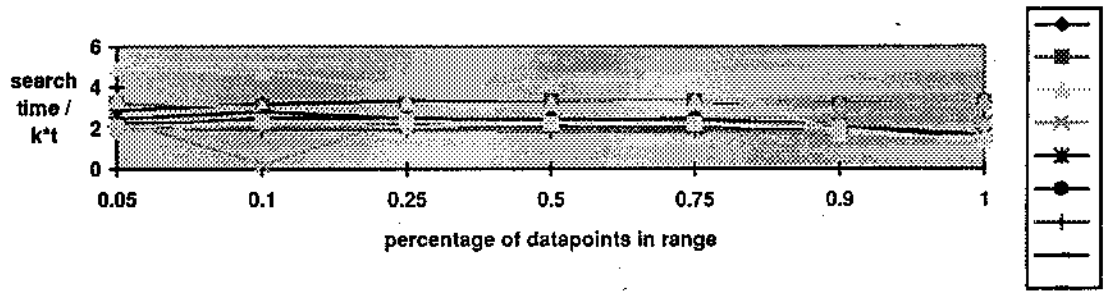

Figure 17. A plot of the search times relative to k*t for the k-d Range DSL.

7.4 Comparison of the k-d Structures

In this section we perform a brief comparison of the two structures. We expect a close correspondence between the k-d Range DSL and the k-d Range AVL tree as the two structures are in the same class (section 6.1) and known to be equivalent [Lamo96d].

Sections 7.2 and 7.3 have provided us with approximations for the constants in the insertion and deletion procedures and timings for range search which indicate that both structures are extremely fast and have close to the same range search time.

Table 25
A comparison of the constants for the k-d Range AVL Tree and the k-d Range DSL

	Insertion constant (approx.)	Deletion constant (approx.)
k-d Range AVL Tree	35	2
k-d Range DSL	17	37

The constants for the insert and delete procedures are given above in Table 25. The constants indicate that the two structures are similar in balance, the k-d Range DSL being approximately 2 times faster for insertion and the k-d Range AVL Tree being approximately 15 times faster for deletion, and that they should be relatively interchangeable in practice, depending on the application.

Table 26 lists the time to locate and report 0, 5, 10, 25, 50, 75, 90, and 100 points, in microseconds, in each of the four dimensions for the k-d Range AVL Tree and the k-d Range DSL. Examining the table we conclude that it takes about 1.5 milliseconds to

locate and report 100 points in a four dimensional Range AVL Tree structure and about 2.0 milliseconds to locate and report the same amount of points in a 4-d Range DSL. Based on the data in Table 26, we conclude that the k-d Range AVL Tree is about 1.5 times faster, on average, for range search in practice. Although this result may initially seem counterintuitive as we know that the k-d Range DSL is usually of a shallow depth than the k-d Range AVL tree, we can explain it as follows. When we move to a subtree in the Range AVL Tree we don't have to determine the node at which we must continue our search as we must do in the Range DSL. It is the fact that we must often check multiple nodes in the subtree before we are able to determine the node that we move to at the next level that that causes the search procedure to often be slower in practice.

Table 26 was calculated using the data of Tables 13 and 23. All of the search times in each dimension were normalized so that they represented the time to report 100 datapoints and the averages of these times were taken across each dimension.

Table 26(a)
Time to locate the given number of points (in microseconds) in the k-d Range AVL Tree

dim	0	5	10	25	50	75	90	100
1	1	18	40	96	195	303	370	416
2	1	29	62	158	314	470	574	620
3	3	51	101	259	513	785	936	1014
4	4	72	156	359	717	1094	1320	1450

Table 26(b)
Time to locate the given number of points (in microseconds) in the k-d Range DSL

dim	0	5	10	25	50	75	90	100
1	2	30	61	148	300	455	557	595
2	3	53	103	244	493	741	850	891
3	7	76	153	377	748	1140	1321	1387
4	10	104	207	501	1054	1536	1854	2046

We now go on to perform a brief analysis of the storage space required, S(n,k). Both structures have the same analytical storage cost, $S(n,k) = \Theta(n \lg^{k-1} n)$, and approximately the same number of nodes. As a range AVL tree structure always contains $2n - 1$ nodes for n datapoints, the storage is roughly $2n \lg^{k-1} n$. On the other hand, a 1-3 Range DSL structure always has between $4/3n$ and $2n - 1$ nodes. Thus, on the average, it usually has about $1.5n \lg^{k-1} n$ nodes.

Our implementation of the node structure for the k-d Range AVL Tree requires 9 integers plus 1 integer for each dimension of the structure. Our implementation of the node structure for the k-d Range DSL requires 5 integers plus 1 integer for each dimension of the structure. On UNIX, each integer requires 4 bytes of storage. Table 27 gives the expected storage requirement, in kilobytes, for the k-d Range AVL Tree and the k-d Range DSL for the data set sizes of Table 1. We assume a storage requirement of $1.5\, n \lg^{k-1} n$ for the k-d Range DSL.

Examining the tables, we find that the k-d Range AVL tree usually requires about twice the amount of storage that the k-d Range DSL requires, but that both are obviously

of the same storage complexity. They both require in the hundreds of megabytes for the largest structures that we build and test.

Table 27(a)
Storage required, in KB, for the Range AVL tree structures with sizes defined by Table 1

dim	Storage requirements, in KB, for the data structures with sizes given in Table 1								
1	53	197	359	529	707	889	1076	1266	1459
2	388	1788	3537	5502	7628	9881	12241	14694	
3	2815	16047	34594	56732	81663				
4	20263	143056	336013						

Table 27(b)
Storage required, in KB, for the Range DSL structures whose sizes are given in Table 1

dim	Storage requirements, in KB, for the data structures with sizes given in Table 1								
1	24	89	161	238	318	400	484	570	657
2	185	853	1688	2626	3641	4716	5842	7013	
3	1408	8024	17297	28366	40831				
4	10521	74279	174468						

One question that immediately comes to mind is the following: how many datapoints can we insert into our structure for a given number of dimensions? It is obvious that as the number of dimensions increase that the size of a workable data set decreases as the $\lg^k n$ factor becomes more prominent with larger k. For sol, it was found that, when usage was low, it was possible to build a 1-d structure of 10,000,000 datapoints, a 2-d structure of 100,000 datapoints, a 3-d structure of 32,000 datapoints, and a 4-d structure of 4,000 datapoints. The corresponding structures required approximately 800 MB, 880 MB, 692 MB, and 720 MB for the k-d Range AVL Tree and approximately 360 MB, 420 MB, 345 MB, and 375 MB for the k-d Range DSL.

8. CONCLUSIONS

8.1 Summary

We conclude that it is possible to have a k-dimensional data structure efficient for range search and dynamic updates and which maintains a reasonable storage requirement and preprocessing cost and offer the k-d Range DSL and k-d Range AVL Tree as examples. Both structures are dynamically balanced ($\Theta(\lg^{3k} n)$ balance cost function) under our model and within a factor of $O(\lg^{k-1} n)$ of being optimally balanced. They are optimal for range search ($\Theta(\lg^2 n + t)$) in the class of dynamically balanced structures.

Dynamic implementations of the structures are feasible and our implementations are extremely fast for k-d range search, requiring only 1.5 to 2.0 milliseconds to locate and report 100 datapoints in a four-dimensional structure. Also, the constants in the update procedures are reasonable, with the k-d Range AVL Tree being faster for deletions and the k-d Range DSL being faster for insertions. They have a similar storage requirement, but the k-d Range AVL Tree requires approximately twice as much storage in practice.

As memory and hard drive space continues to increase on today's computers, it is reasonable to assume that the k-d Range AVL Tree and the k-d Range DSL provide efficient, feasible index structures for k-d range search on k-dimensional structures. Also, the facts that they allow for optimal member queries and efficient partial match queries make them very attractive as index structures.

8.2 Future Work

In this thesis we have made a number of assumptions about our data set and it's size, including that of $k << n$ and $k < \lg n$. The natural question to ask now is: what happens to our analysis if $\lg n < k < n$? If $k > n$, is it feasible to work with the dual problem, and, if so, how does one define the dual?

Other questions concern that of the underlying computing environment. What increases can be obtained in a parallel processing environment? What happens if we have one processor per dimension? What if we have one processor per datapoint?

In the section on Lower Bounds (Ch. 6.2), we mentioned that clustering could moderately improve the worst case update time. How efficient would an implementation of our structure be, in practice, if clustering was used? Would the average update time decrease? Would the average search time increase?

We also mentioned that using an amortization scheme might reduce the worst-case update time as well. How efficient would an implementation of a good amortization scheme be? By how much would the search time increase, if at all? How good could an amortization scheme be in theory? What is the lower bound on the worst case update time when an amortization scheme is used? Is amortization as effective as clustering in practice? If it is more effective, by how much?

One of the big open questions is: what is the lower bound on a single update in our dynamic model? We know, from Theorem 21, that it is $\Omega(n)$ and that it is $O(n^{1/2} \lg^k n^{1/2})$ (from [Bent79b]), but it's true value in a Θ analysis still remains a mystery.

We also want to know if there are any modifications we can make to the update algorithms so that the average update time is decreased. We have noticed that the average deletion time for the k-d Range DSL, in particular, is quite high. Would a bottom-up deletion scheme be more effective on average?

Another big question is whether $\Omega(n)$ storage really is minimal in our model? It might be the case that having a range search and update cost function of $\Theta(\lg^k n)$ requires a structure of larger storage requirement, possibly even a storage function of $\Theta(n\lg^{k-1} n)$ if a large number of subsets of our set of datapoints must be indexed in sorted order for each coordinate value. If this were true, our optimal balance cost function would be $\Theta(n\lg^{5k-1} n)$ and our structures would be optimally balanced.

Finally, in Chapter 7 we mentioned that it might be possible to incorporate the Interval Skip List data structure into the k-d Range DSL data structure and thereby provide a multi-dimensional alternative to the R-Tree data structure. Could this be done? If so, how efficient would our structure be using a worst case analysis?

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. 509 - 517.

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

[Bent78] Bentley, J.L., Friedman, J.H., and Maurer, H.A., "Two Papers on Range Searching", CMU-CS-78-136 (Technical Report of the Departments of Computer Science and Mathematics, Carnegie-Mellon University), August 1978.

[Bent79a] Bentley, J.L., & Friedman, Jerome H., "Data Structures for Range Searching", *Computing Surveys*, Vol. 11, No. 4, December 1979, pp. 397 - 409.

[Bent79b] Bentley, J. L., "Decomposable Searching Problems", *Information Processing Letters*, Volume 8, Number 5, June 1979, pp. 244 - 251.

[Bent79c] Bentley, J.L., and Saxe, J.B., "Algorithms on Vector Sets", *SIGACT NEWS*, Fall 1979, pp. 36 - 39.

[Bent80a] Bentley, J.L., "Multidimensional Divide-and-Conquer", *Communications of the ACM*, Volume 23, Number 4 (April 1980), pp. 214 - 229.

[Bent80b] Bentley, J.L., and Maurer, H.A., "Efficient Worst-Case Data Structures for Range Searching", *Acta Informatica*, Vol. 13, No. 2, pp. 155 - 168 (1980).

[Bent80c] Bentley, J.L., and Saxe, J.B., "Decomposable Searching Problems I. Static-to-Dynamic Transformations", *Journal of Algorithms*, Vol. 1, pp. 301 - 358 (1980).

[Come79] Comer, Douglas, "The Ubiquitous B-Tree", *Computing Surveys*, Vol. 11, No. 2, June 1979.

[Culi81] Culik, K., Ottmann, Th., and Wood, D., "Dense Multiway Trees", *ACM Transactions on Database Systems*, Vol. 6, No. 3, September 1981, pp. 486 - 512.

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

[Fred81a] Fredman, Michael L., "A Lower Bound on the Complexity of Orthogonal Range Queries", *Journal of the ACM*, Vol. 28, No. 4, October 1981, pp. 696 - 705.

[Guti80] Guting, H. and Kriegel, H.P., "Multidimensional B-Tree: An efficient Dynamic File Structure for Exact Match Queries", Proceedings of the 10th GI Annual Conference Informatik Fachberichte, Springer Verlag, 1980, pp. 375-388.

[Guib78] Guibas, Leo J. & Sedgewick, Robert, "A Dichromatic Framework for Balanced Trees", Proceedings of the 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, Oct 16-18, 1978, pp. 8 - 21.

[Hans92] Hanson, Eric N. & Johnson, Theodore, "The Interval Skip List: A Data Structure for Finding All Intervals That Overlap a Point", Computer and Information Sciences Department of the University of Florida Technical Report, UF - CIS - 92 - 016 (June 16, 1992).

[Hans94] Hanson, Eric N. & Johnson, Theodore, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists", Computer and Information Sciences Department of the University of Florida Technical Report, TR94-017 (April 15, 1994).

[Knut73] Knuth, D.E., *The Art of Computer Programming: Volume 3 Searching and Sorting*, Addison-Wesley, Reading, M.A., 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., "An Implementation of a Multidimensional Dynamic Range Tree Based On An AVL Tree", University of New Brunswick Technical Report, TR95-100, November 1995.

[Lamo95c] Lamoureux, Michael G., and Nickerson, Bradford G., "Deterministic Skip List Data Structures: Efficient Alternatives to Balanced Search Trees", Proceedings of the 19th Annual Mathematics and Computing Science Days (APICS'95), Sydney, Cape Breton (Nova Scotia, Canada), Oct. 20-21, 1995.

[Lamo96d] Lamoureux, Michael G., and Nickerson, Bradford G., "On the equivalence of B-trees and deterministic skip lists", University of New Brunswick Technical Report, TR96-102, January 1996.

[Luek78] Lueker, George S., "A Data Structure for Orthogonal Range Queries", Proceedings of the 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, Oct 16-18, 1978, 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.

[Mato94] Matousek, Jiri, "Geometric Range Searching", *ACM Computing Surveys*, Vol. 26, No. 4, December 1994, pp. 421 - 461.

[Maur78] Maurer, H., and Ottmann, Th., "Manipulating sets of points - a survey", Graphs, Data Structures, Algorithms Proceedings of the Workshop WG 78 on Graphtheoretic Concepts in Computer Science, pp. 9 - 29.

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

[Munr92] Munro, J. Ian; Papadakis, Thomas; and Sedgewick, R., "Deterministic Skip Lists", Proceedings of the Third Annual Symposium on Discrete Algorithms, Orlando, Florida, January 27-29, 1992.

[Nick94] Nickerson, Bradford G., "Skip List Data Structure for Multidimensional Data", Computer Science Technical Report Series, CS TR-3262 UMIACS CS-TR-94-52, April 1994.

[Papa93] Papadakis, T., "Skip Lists and Probabilistic Analysis of Algorithms", Ph.D. Thesis, University of Waterloo, 1993.
(anon. ftp site "cs-archive.uwaterloo.ca" in file "cs-archive/cs-93-28")

[Robi81] Robinson, John T., "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes", ACM-SIGMOD 1981, International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981, pp.10 - 18.

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

[VanL80] Van Leeuwen, Jan, and Wood, Derick, "Dynamization of Decomposable Searching Problems", *Information Processing Letters*, Volume 10, Number 2 (March 1980), pp. 51 - 56.

[Will85a] Willard, Dan. E., and Lueker, George S., "Adding Range Restriction Capability to Dynamic Data Structures", *Journal of the 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 A

RAW CONSTRUCTION AND DESTRUCTION TIMES FOR INITIAL TESTING OF THE K-D RANGE AVL TREE

Table A1(a)

Raw construct times for pass 1 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	17	17	0	0	0	33	0	50	17
2	0	83	100	116	167	266	200	284	
3	0	683	883	1617	2083				
4	0	2100	4467						

Table A1(b)

Raw destruct times for pass 1 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	17	0	17	0	0	17	33	17
2	0	0	0	34	33	17	50	50	
3	0	17	16	34	67				
4	0	67	50						

Table A2(a)

Raw construct times for pass 2 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	0	17	16	33	17
2	0	34	83	67	167	50	267	300	
3	17	400	500	1200	1900				
4	17	1333	2950						

Table A2(b)

Raw destruct times for pass 2 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	0	34
2	0	0	0	17	0	33	100	100	
3	0	0	0	50	33				
4	0	0	400						

Table A3(a)

Raw construct times for pass 3 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	17	0	50	0	0	16
2	0	67	100	116	167	133	166	166	
3	17	416	750	934	2134				
4	0	1783	3317						

Table A3(b)
Raw destruct times for pass 3 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	17	17	0	0	33	0	0
2	0	0	50	17	50	0	50	50	
3	0	0	17	33	34				
4	34	50	267						

Table A4(a)
Raw construct times for pass 4 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	17	0	0	0	0	16
2	17	17	50	83	284	166	234	217	
3	0	384	650	1150	1616				
4	0	1800	3366						

Table A4(b)
Raw destruct times for pass 4 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	17	16	0	17	0	0	17
2	0	0	16	0	0	0	33	16	
3	0	0	33	33	116				
4	17	116	250						

Table A5(a)
Raw construct times for pass 5 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	16	16	0	0	0
2	0	33	134	50	100	67	200	400	
3	0	267	717	1050	1617				
4	84	1350	2950						

Table A5(b)
Raw destruct times for pass 5 of run 1 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	16	0	17	0	0	0	17	0
2	0	0	0	16	0	16	17	33	
3	0	17	17	34	166				
4	66	17	367						

Table A6(a)
Raw construct times for pass 1 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	50	50	16	0	34	67	66
2	400	634	266	83	166	17	200	250	
3	100	350	734	1150	1650				
4	0	2650	5916						

Table A6(b)
Raw destruct times for pass 1 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	33	17	17	17	16	50
2	0	33	34	17	16	34	67	50	
3	17	0	0	50	17				
4	0	0	17						

Table A7(a)
Raw construct times for pass 2 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	0	50
2	117	100	100	34	66	50	133	200	
3	50	217	466	766	1183				
4	0	1750	4650						

Table A7(b)
Raw destruct times for pass 2 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	17	17	50	17	33	0
2	0	33	17	0	34	67	67	17	
3	0	33	0	34	17				
4	0	0	83						

Table A8(a)
Raw construct times for pass 3 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	17	0	0	17	0	17	0
2	17	50	100	67	83	17	167	250	
3	50	150	417	750	1083				
4	0	1633	5100						

Table A8(b)
Raw destruct times for pass 3 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	16	0	0	33	34	50	33
2	0	34	50	50	17	50	67	67	
3	0	17	0	0	17				
4	0	0	33						

Table A9(a)
Raw construct times for pass 4 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	0	17
2	50	116	50	33	33	17	183	134	
3	67	200	567	834	1250				
4	16	1734	5867						

Table A9(b)
Raw destruct times for pass 4 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	16	0	0	33	0	0	0	0	17
2	0	17	0	50	133	17	50	100	
3	0	0	0	17	33				
4	17	17	83						

Table A10(a)
Raw construct times for pass 5 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	16	17
2	33	67	17	34	67	50	200	283	
3	33	267	350	650	1200				
4	117	1900	6400						

Table A10(b)
Raw destruct times for pass 5 of run 2 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	50	50	0	17	16	33	17
2	0	0	34	67	67	50	134	34	
3	0	17	0	50	33				
4	17	34	16						

Table A11(a)
Raw construct times for pass 1 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	33	17	0	0	50	16	16
2	0	67	217	100	84	150	200	267	
3	0	483	933	1150	1517				
4	0	2883	6850						

Table A11(b)
Raw destruct times for pass 1 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	16	0	16	0	0	16
2	0	0	0	0	0	0	34	0	
3	0	0	50	17	16				
4	17	33	50						

Table A12(a)
Raw construct times for pass 2 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	16	0	0	0	0	17	0
2	0	0	0	33	33	100	150	216	
3	0	167	534	850	1383				
4	17	1717	4967						

Table A12(b)
Raw destruct times for pass 2 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	0	16	0	0	0	17
2	0	0	17	17	33	0	16	16	
3	0	34	17	0	16				
4	17	33	50						

Table A13(a)
Raw construct times for pass 3 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	17	0	0	17	17	0	0	17
2	0	0	16	117	117	133	134	267	
3	17	217	550	883	1350				
4	0	1833	5566						

Table A13(b)
Raw destruct times for pass 3 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	0	33
2	0	0	0	0	33	17	16	34	
3	0	0	17	0	133				
4	16	17	67						

Table A14(a)
Raw construct times for pass 4 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	0	0	0	0	0	0	0	17	17
2	17	50	16	50	66	100	150	150	
3	0	133	466	933	1217				
4	0	1700	5966						

Table A14(b)
Raw destruct times for pass 4 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	17	0	0	17	16	0	0
2	0	0	17	0	17	50	0	34	
3	0	17	0	17	100				
4	17	17	17						

Table A15(a)
Raw construct times for pass 5 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw construct times for data set sizes given in Table 1.								
1	17	0	0	33	17	0	17	0	0
2	17	67	67	67	117	134	133	183	
3	0	250	416	783	1667				
4	0	1950	6200						

Table A15(b)
Raw destruct times for pass 5 of run 3 of initial testing of the k-d Range AVL Tree

dim	Raw destruct times for data set sizes given in Table 1.								
1	0	0	0	0	0	16	17	17	17
2	0	0	16	0	17	0	33	34	
3	0	0	50	16	67				
4	0	17	83						

APPENDIX B

AVERAGE CONSTRUCTION AND DESTRUCTION TIMES FOR EACH INITIAL TEST RUN OF THE K-D RANGE AVL TREE

Table B1(a)
Construct time averages for run 1 of initial testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 1								
1	3	3	0	6	3	23	3	16	13
2	3	46	93	86	177	136	213	273	
3	6	430	700	1190	1870				
4	20	1673	3410						

Table B1(b)
Destruct time averages for run 1 of initial testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 1								
1	0	6	6	13	0	3	10	10	17
2	0	0	13	16	16	13	50	49	
3	0	6	16	36	83				
4	23	50	266						

Table B2(a)
Construct time averages for run 2 of initial testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 1								
1	0	0	13	10	3	3	6	20	30
2	123	193	106	50	83	30	176	223	
3	60	236	506	830	1273				
4	26	1933	5586						

Table B2(b)
Destruct time averages for run 2 of initial testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 1								
1	3	0	13	26	6	23	16	26	23
2	0	23	27	36	53	43	77	53	
3	3	13	0	30	23				
4	6	10	46						

Table B3(a)
Construct time averages for run 3 of initial testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 1								
1	3	3	9	10	6	3	13	10	10
2	6	36	63	73	83	123	153	216	
3	3	250	579	919	1426				
4	3	2016	5909						

Table B3(b)
Destruct time averages for run 3 of initial testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 1								
1	0	0	3	3	3	9	6	3	16
2	0	0	10	3	20	13	19	23	
3	0	10	26	10	66				
4	13	23	53						

Table B4(a)
Construct time averages for all passes of all runs of initial testing of the Range AVL Tree

dim	Construct time averages for data set sizes given in Table 1								
1	2	2	7	9	4	10	7	15	18
2	44	92	87	70	114	96	181	237	
3	23	305	595	980	1523				
4	16	1874	4968						

Table B4(b)
Destruct time averages for all passes of all runs of initial testing of the Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 1								
1	1	2	7	14	3	12	11	13	19
2	0	8	17	18	30	23	49	42	
3	1	10	14	25	57				
4	14	28	122						

APPENDIX C

RAW INSERTION AND DELETION CONSTANTS FOR INITIAL TESTING OF THE K-D RANGE AVL TREE

Table C1(a)

Insertion constants for pass 1 of run 1 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	255.88	111.20	0.00	0.00	0.00	59.60	0.00	64.81	19.25
2	0.00	71.03	49.23	38.81	41.55	52.05	31.99	38.17	
3	0.00	76.46	52.82	62.59	57.80				
4	0.00	30.76	32.47						

Table C1(b)

Deletion constants for pass 1 of run 1 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	111.20	0.00	49.17	0.00	0.00	25.70	42.77	19.25
2	0.00	0.00	0.00	11.38	8.21	3.33	8.00	6.72	
3	0.00	1.90	0.96	1.32	1.86				
4	0.00	0.98	0.36						

Table C2(a)

Insertion constants for pass 2 of run 1 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	30.70	24.18	42.77	19.25
2	0.00	29.10	40.86	22.42	41.55	9.78	42.70	40.32	
3	5.80	44.78	29.91	46.45	52.73				
4	0.87	19.52	21.45						

Table C2(b)

Deletion constants for pass 2 of run 1 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	38.49
2	0.00	0.00	0.00	5.69	0.00	6.46	15.99	13.44	
3	0.00	0.00	0.00	1.94	0.92				
4	0.00	0.00	2.91						

Table C3(a)

Insertion constants for pass 3 of run 1 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	49.17	0.00	90.30	0.00	0.00	18.12
2	0.00	57.33	49.23	38.81	41.55	26.03	26.55	22.31	
3	5.80	46.57	44.87	36.15	59.22				
4	0.00	26.11	24.11						

Table C3(b)

Deletion constants for pass 3 of run 1 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	68.86	49.17	0.00	0.00	49.88	0.00	0.00
2	0.00	0.00	24.61	5.69	12.44	0.00	8.00	6.72	
3	0.00	0.00	1.02	1.28	0.94				
4	1.75	0.73	1.94						

Table C4(a)

Insertion constants for pass 4 of run 1 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	49.17	0.00	0.00	0.00	0.00	18.12
2	38.51	14.55	24.61	27.77	70.66	32.48	37.42	29.17	
3	0.00	42.99	38.88	44.52	44.84				
4	0.00	26.36	24.47						

Table C4(b)

Deletion constants for pass 4 of run 1 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	68.86	46.28	0.00	30.70	0.00	0.00	19.25
2	0.00	0.00	7.88	0.00	0.00	0.00	5.28	2.15	
3	0.00	0.00	1.97	1.28	3.22				
4	0.87	1.70	1.82						

Table C5(a)

Insertion constants for pass 5 of run 1 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	35.69	28.89	0.00	0.00	0.00
2	0.00	28.24	65.96	16.73	24.88	13.11	31.99	53.76	
3	0.00	29.89	42.89	40.64	44.87				
4	4.31	19.77	21.45						

Table C5(b)

Deletion constants for pass 5 of run 1 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	104.66	0.00	49.17	0.00	0.00	0.00	22.03	19.25
2	0.00	0.00	0.00	5.35	0.00	3.13	2.72	4.44	
3	0.00	1.90	1.02	1.32	4.61				
4	3.39	0.25	2.67						

Table C6(a)

Insertion constants for pass 1 of run 2 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	202.54	144.61	35.69	0.00	51.39	86.84	74.72
2	906.19	542.54	130.94	27.77	41.30	3.33	31.99	33.60	
3	34.10	39.18	43.91	44.52	45.79				
4	0.00	38.81	43.01						

Table C6(b)

Deletion constants for pass 1 of run 2 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	95.44	37.92	30.70	25.70	20.74	56.61
2	0.00	28.24	16.74	5.69	3.98	6.65	10.72	6.72	
3	5.80	0.00	0.00	1.94	0.47				
4	0.00	0.00	0.12						

Table C7(a)

Insertion constants for pass 2 of run 2 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	56.61
2	265.06	85.57	49.23	11.38	16.42	9.78	21.27	26.88	
3	17.05	24.29	27.88	29.65	32.83				
4	0.00	25.63	33.81						

Table C7(b)

Deletion constants for pass 2 of run 2 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	49.17	37.92	90.30	25.70	42.77	0.00
2	0.00	28.24	8.37	0.00	8.46	13.11	10.72	2.28	
3	0.00	3.69	0.00	1.32	0.47				
4	0.00	0.00	0.60						

Table C8(a)

Insertion constants for pass 3 of run 2 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	68.86	0.00	0.00	30.70	0.00	22.03	0.00
2	38.51	42.79	49.23	22.42	20.65	3.33	26.71	33.60	
3	17.05	16.79	24.95	29.03	30.05				
4	0.00	23.92	37.08						

Table C8(b)

Deletion constants for pass 3 of run 2 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	64.81	0.00	0.00	59.60	51.39	64.81	37.36
2	0.00	29.10	24.61	16.73	4.23	9.78	10.72	9.00	
3	0.00	1.90	0.00	0.00	0.47				
4	0.00	0.00	0.24						

Table C9(a)

Insertion constants for pass 4 of run 2 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	19.25
2	113.27	99.27	24.61	11.04	8.21	3.33	29.27	18.01	
3	22.85	22.39	33.92	32.28	34.69				
4	0.82	25.40	42.65						

Table C9(b)

Deletion constants for pass 4 of run 2 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	240.82	0.00	0.00	95.44	0.00	0.00	0.00	0.00	19.25
2	0.00	14.55	0.00	16.73	33.09	3.33	8.00	13.44	
3	0.00	0.00	0.00	0.66	0.92				
4	0.87	0.25	0.60						

Table C10(a)

Insertion constants for pass 5 of run 2 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	20.74	19.25
2	74.76	57.33	8.37	11.38	16.67	9.78	31.99	38.04	
3	11.25	29.89	20.94	25.16	33.30				
4	6.00	27.83	46.53						

Table C10(b)

Deletion constants for pass 5 of run 2 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	202.54	144.61	0.00	30.70	24.18	42.77	19.25
2	0.00	0.00	16.74	22.42	16.67	9.78	21.43	4.57	
3	0.00	1.90	0.00	1.94	0.92				
4	0.87	0.50	0.12						

Table C11(a)

Insertion constants for pass 1 of run 3 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	133.68	49.17	0.00	0.00	75.58	20.74	18.12
2	0.00	57.33	106.82	33.46	20.90	29.35	31.99	35.89	
3	0.00	54.07	55.81	44.52	42.10				
4	0.00	42.22	49.80						

Table C11(b)

Deletion constants for pass 1 of run 3 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	46.28	0.00	28.89	0.00	0.00	18.12
2	0.00	0.00	0.00	0.00	0.00	0.00	5.44	0.00	
3	0.00	0.00	2.99	0.66	0.44				
4	0.87	0.48	0.36						

Table C12(a)

Insertion constants for pass 2 of run 3 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	64.81	0.00	0.00	0.00	0.00	22.03	0.00
2	0.00	0.00	0.00	11.04	8.21	19.57	23.99	29.03	
3	0.00	18.70	31.95	32.90	38.38				
4	0.87	25.15	36.11						

Table C12(b)

Deletion constants for pass 2 of run 3 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	35.69	0.00	0.00	0.00	19.25
2	0.00	0.00	8.37	5.69	8.21	0.00	2.56	2.15	
3	0.00	3.81	1.02	0.00	0.44				
4	0.87	0.48	0.36						

Table C13(a)

Insertion constants for pass 3 of run 3 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	111.20	0.00	0.00	37.92	30.70	0.00	0.00	19.25
2	0.00	0.00	7.88	39.15	29.11	26.03	21.43	35.89	
3	5.80	24.29	32.90	34.18	37.46				
4	0.00	26.85	40.46						

Table C13(b)

Deletion constants for pass 3 of run 3 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	37.36
2	0.00	0.00	0.00	0.00	8.21	3.33	2.56	4.57	
3	0.00	0.00	1.02	0.00	3.69				
4	0.82	0.25	0.49						

Table C14(a)

Insertion constants for pass 4 of run 3 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	22.03	19.25
2	38.51	42.79	7.88	16.73	16.42	19.57	23.99	20.16	
3	0.00	14.89	27.88	36.12	33.77				
4	0.00	24.90	43.37						

Table C14(b)

Deletion constants for pass 4 of run 3 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	68.86	0.00	0.00	30.70	24.18	0.00	0.00
2	0.00	0.00	8.37	0.00	4.23	9.78	0.00	4.57	
3	0.00	1.90	0.00	0.66	2.78				
4	0.87	0.25	0.12						

Table C15(a)

Insertion constants for pass 5 of run 3 of initial testing of the k-d Range AVL Tree

dim	Insertion constants for the data set sizes given by Table 1.								
1	255.88	0.00	0.00	95.44	37.92	0.00	25.70	0.00	0.00
2	38.51	57.33	32.98	22.42	29.11	26.22	21.27	24.60	
3	0.00	27.99	24.89	30.31	46.26				
4	0.00	28.56	45.07						

Table C15(b)

Deletion constants for pass 5 of run 3 of initial testing of the k-d Range AVL Tree

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	28.89	25.70	22.03	19.25
2	0.00	0.00	7.88	0.00	4.23	0.00	5.28	4.57	
3	0.00	0.00	2.99	0.62	1.86				
4	0.00	0.25	0.60						

APPENDIX D

INSERTION AND DELETION CONSTANTS

FOR EACH INITIAL TEST RUN

OF THE K-D RANGE AVL TREE

Table D1(a)
Constants for insertion averages of run 1 of initial testing of the Range AVL Tree

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	45.15	19.62	0.00	17.35	6.69	41.54	4.53	20.74	14.72
2	6.80	39.36	45.78	28.78	44.04	26.61	34.06	36.69	
3	2.05	48.14	41.88	46.06	51.89				
4	1.03	24.50	24.79						

Table D1(b)
Constants for deletion averages of run 1 of initial testing of the Range AVL Tree

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	39.25	24.30	37.60	0.00	5.42	15.12	12.96	19.25
2	0.00	0.00	6.40	5.35	3.98	2.54	8.00	6.59	
3	0.00	0.67	0.96	1.39	2.30				
4	1.18	0.73	1.93						

Table D2(a)
Standard deviations of constants for insertion averages of run 1 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D1(a).								
1	114.63	49.82	0.00	27.06	15.97	34.30	10.82	30.49	8.38
2	17.25	23.29	15.03	9.85	16.54	16.94	6.14	11.94	
3	3.19	17.14	8.40	10.04	6.86				
4	1.87	4.81	4.52						

Table D2(b)
Standard deviations of constants for deletion averages of run 1 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D1(b).								
1	0.00	59.32	37.89	21.74	0.00	13.75	22.39	19.20	13.61
2	0.00	0.00	10.68	4.04	5.85	2.70	4.98	4.22	
3	0.00	1.05	0.70	0.29	1.59				
4	1.42	0.66	1.00						

Table D3(a)
Constants for insertion averages of run 2 of initial testing of the Range AVL Tree

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	0.00	0.00	52.66	28.92	6.69	5.42	9.07	25.92	33.97
2	278.65	165.16	52.18	16.73	20.65	5.87	28.15	29.97	
3	20.46	26.42	30.27	32.13	35.33				
4	1.33	28.31	40.61						

Table D3(b)
Constants for deletion averages of run 2 of initial testing of the Range AVL Tree

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	45.15	0.00	52.66	75.20	13.38	41.54	24.18	33.70	26.04
2	0.00	19.68	13.29	12.05	13.19	8.41	12.31	7.12	
3	1.02	1.46	0.00	1.16	0.64				
4	0.31	0.15	0.33						

Table D4(a)
Standard deviations for constants of insertion averages of run 2 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D3(a).								
1	0.00	0.00	88.10	64.67	15.97	13.75	23.02	35.70	30.66
2	360.76	211.95	47.17	7.81	12.40	3.53	4.48	7.81	
3	8.66	8.49	8.95	7.38	6.08				
4	2.62	6.03	5.09						

Table D4(b)
Standard deviations for constants of deletion averages of run 2 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D3(b).								
1	107.75	0.00	87.94	54.70	20.87	34.15	18.23	24.68	21.41
2	0.00	12.74	9.39	9.18	12.20	3.70	5.23	4.29	
3	2.60	1.55	0.00	0.84	0.25				
4	0.48	0.22	0.25						

Table D5(a)
Constants of insertion averages of run 3 of initial testing of the Range AVL Tree

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	45.15	19.62	36.46	28.92	13.38	5.42	19.65	12.96	11.32
2	13.59	30.81	31.01	24.43	20.65	24.07	24.47	29.03	
3	1.02	27.99	34.64	35.57	39.57				
4	0.15	29.53	42.96						

Table D5(b)
Constants of deletion averages of run 3 of initial testing of the Range AVL Tree

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	0.00	12.15	8.68	6.69	16.25	9.07	3.89	18.12
2	0.00	0.00	4.92	1.00	4.98	2.54	3.04	3.09	
3	0.00	1.12	1.56	0.39	1.83				
4	0.67	0.34	0.39						

Table D6(a)
Standard deviations for constants of insertion averages of run 3 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D5(a).								
1	114.63	49.82	59.67	42.85	20.87	13.75	32.88	11.84	10.35
2	21.19	29.36	44.11	11.63	8.89	4.38	4.37	6.94	
3	2.60	15.43	12.24	5.41	4.76				
4	0.39	7.24	5.11						

Table D6(b)
Standard deviations for constants of deletion averages of run 3 of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D5(b).								
1	0.00	0.00	30.85	20.71	15.97	16.25	13.71	9.87	13.24
2	0.00	0.00	4.50	2.55	3.42	4.25	2.26	2.06	
3	0.00	1.70	1.33	0.35	1.43				
4	0.38	0.13	0.18						

Table D7(a)
Constants for insertion averages of all passes of all runs of initial testing of the Range AVL Tree

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	30.10	13.08	29.71	25.07	8.92	17.46	11.08	19.87	20.00
2	99.68	78.44	42.99	23.31	28.45	18.85	28.89	31.90	
3	7.84	34.18	35.59	37.92	42.26				
4	0.84	27.45	36.12						

Table D7(b)
Constants for deletion averages of all passes of all runs of initial testing of the Range AVL Tree

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	15.05	13.08	29.71	40.49	6.69	21.07	16.12	16.85	21.13
2	0.00	6.56	8.20	6.13	7.38	4.50	7.78	5.60	
3	0.34	1.08	0.84	0.98	1.59				
4	0.72	0.41	0.88						

Table D8(a)
Standard deviations for constants for insertion averages of all passes of all runs of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D7(a).								
1	76.42	33.21	49.26	44.86	17.60	20.60	22.24	26.01	16.46
2	133.07	88.20	35.44	9.76	12.61	8.28	5.00	8.90	
3	4.82	13.69	9.86	7.61	5.90				
4	1.63	6.03	4.91						

Table D8(b)
Standard deviations for constants for deletion averages of all passes of all runs of initial testing of the Range AVL Tree

dim	Standard deviations of constants in Table D2(a).								
1	35.92	19.77	52.23	32.38	12.28	21.38	18.11	17.92	16.09
2	0.00	4.25	8.19	5.26	7.16	3.55	4.16	3.52	
3	0.87	1.43	0.68	0.49	1.09				
4	0.76	0.34	0.48						

Table D9(a)
Standard deviations of the constants in Table 5

dimensions	run 1	run 2	run 3	all runs
2	14.20	42.16	15.67	24.02
3	10.61	7.73	9.46	9.27
4	4.67	5.56	6.18	5.47

Table D9(b)
Standard deviations of the constants in Table 6

dimensions	run 1	run 2	run 3	all runs
2	4.64	8.10	2.72	5.16
3	0.91	0.66	1.20	0.92
4	0.83	0.24	0.16	0.41

APPENDIX E

AVERAGE SEARCH TIMES FOR EACH INITIAL TEST RUN OF THE K-D RANGE AVL TREE

Table E1
Average search times for run 1 of initial testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Actual Search Times							
1	100	7	10	43	83	163	270	363	380
	300	3	70	124	350	570	963	1103	1350
	500	3	96	184	563	1073	1610	1877	2097
	700	0	174	307	727	1527	2070	2719	3020
	900	3	173	367	820	1907	2910	3493	3977
	1100	3	227	533	1250	2320	3707	4510	4970
	1300	0	203	540	1437	2827	4333	5227	5996
	1500	7	327	673	1663	3106	4844	6193	6657
	1700	7	373	744	1737	3576	5660	6907	7250
2	100	10	30	73	154	323	523	640	717
	300	3	103	213	643	1087	1543	1840	2196
	500	0	143	376	990	1830	2697	3333	3940
	700	10	250	510	1233	2657	3870	4600	5270
	900	0	273	730	1670	3190	4657	6217	6900
	1100	0	353	724	1893	4010	5727	7297	8260
	1300	13	477	890	2373	4713	6953	8660	9537
	1500	0	443	763	1883	3927	5977	7207	7603
3	100	17	50	83	250	433	733	787	950
	300	16	130	353	817	1524	2743	3017	3503
	500	20	240	587	1483	2727	4113	4980	5807
	700	10	383	717	1974	4033	6173	7430	8260
	900	10	533	1140	2637	5247	7933	9600	10353
4	100	3	67	174	413	747	1073	1367	1567
	300	7	223	543	1180	2403	3660	4250	4963
	500	23	474	803	1960	3793	6223	7647	8000

Table E2
Average search times for run 2 of initial testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Actual Search Times							
1	100	3	13	53	46	133	310	383	430
	300	7	43	130	286	567	937	1157	1483
	500	3	110	183	527	1054	1540	1993	2043
	700	10	147	317	683	1333	2073	2753	2913
	900	7	187	287	947	1820	3003	3550	4077
	1100	10	220	457	1027	2027	3177	3786	4317
	1300	3	243	473	1256	2657	3760	4436	5207
	1500	10	293	540	1477	2770	4373	5370	5940
	1700	10	313	643	1670	3593	5100	5903	7040
2	100	3	33	74	150	320	447	583	617
	300	0	113	184	510	906	1433	1747	2043
	500	3	150	313	807	1580	2333	3056	3390
	700	3	227	460	1087	2244	3440	4297	5100
	900	6	253	567	1540	2984	4573	5537	6263
	1100	13	327	677	1720	3603	5360	6653	7643
	1300	3	280	597	1583	3037	4707	5660	5257
	1500	0	330	617	1480	2790	4190	5114	3466
3	100	7	63	70	270	547	844	940	1020
	300	10	160	280	803	1647	2390	2907	3397
	500	3	290	547	1393	2660	4087	5040	5487
	700	17	384	890	2033	4024	5833	7143	7860
	900	3	453	1160	2537	5226	7964	9680	8650
4	100	7	56	117	373	670	1127	1206	1393
	300	0	240	546	1104	2406	3490	4436	4707
	500	23	384	768	1750	3618	5809	7075	7690

Table E3

Average search times for run 3 of initial testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Actual Search Times							
1	100	3	10	30	100	193	284	323	347
	300	3	30	87	204	517	837	1070	1057
	500	0	87	183	380	887	1367	1693	1937
	700	7	117	297	673	1290	1910	2480	2594
	900	0	160	323	773	1803	2540	3127	3610
	1100	0	224	366	997	2187	3233	3697	4480
	1300	10	244	490	1107	2353	4074	4636	5357
	1500	10	333	653	1533	2913	4380	5176	6344
	1700	3	300	617	1617	3490	5233	6200	7033
2	100	0	23	67	143	280	463	563	690
	300	7	70	220	530	973	1467	1740	1913
	500	23	143	243	790	1617	2540	2990	3460
	700	0	140	447	1153	2363	3597	4217	4557
	900	7	293	567	1510	3080	4617	5070	6040
	1100	7	383	700	1906	3803	5470	7056	7427
	1300	10	297	540	1613	3270	4834	6023	2500
	1500	13	310	633	1523	3104	4567	4363	3047
3	100	3	60	114	243	540	763	1003	1107
	300	7	147	280	720	1457	2227	2607	3343
	500	16	210	523	1320	2623	3990	4843	5620
	700	0	277	463	1344	2687	4037	4927	4253
	900	3	327	557	1500	3013	4333	5083	2824
4	100	13	87	154	313	693	1043	1247	1373
	300	7	180	536	1033	2110	3080	4007	4217
	500	13	250	507	1393	2773	3940	4424	4756

Table E4
Average search times for all runs of initial testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Actual Search Times							
1	100	4	11	42	77	163	288	357	386
	300	4	48	113	280	551	912	1110	1297
	500	2	98	183	490	1005	1506	1854	2026
	700	6	146	307	694	1383	2018	2651	2842
	900	3	173	326	847	1843	2818	3390	3888
	1100	4	223	452	1091	2178	3372	3998	4589
	1300	4	230	501	1267	2612	4056	4767	5520
	1500	9	318	622	1558	2930	4532	5580	6313
	1700	7	329	668	1674	3553	5331	6337	7108
2	100	4	29	71	149	308	478	596	674
	300	3	95	206	561	989	1481	1776	2051
	500	9	146	311	862	1676	2523	3126	3597
	700	4	206	472	1158	2421	3636	4371	4976
	900	4	273	621	1573	3085	4615	5608	6401
	1100	7	354	700	1840	3806	5519	7002	7777
	1300	9	351	676	1857	3673	5498	6781	5764
	1500	4	361	671	1629	3273	4911	5561	4705
3	100	9	58	89	254	507	780	910	1025
	300	11	146	304	780	1542	2453	2843	3414
	500	13	247	552	1399	2670	4063	4954	5638
	700	9	348	690	1783	3581	5348	6500	6791
	900	5	438	952	2225	4496	6743	8121	7276
4	100	8	70	148	367	703	1081	1273	1444
	300	4	214	542	1106	2307	3410	4231	4629
	500	20	369	692	1701	3395	5324	6382	6816

APPENDIX F

MULTIPLIERS REPRESENTING SEARCH TIME INCREASES IN THE INITIAL TEST RUNS OF THE K-D RANGE AVL TREE

Table F1

Multipliers corresponding to search time increases for increase in query window size for run 1 of initial testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	xxxx	2.08	1.86	1.38	1.07
	300	xxxx	2.96	1.77	1.75	1.15	1.25
	500	2.54	3.18	1.94	1.51	1.17	1.14
	700	1.96	2.56	2.13	1.36	1.32	1.12
	900	2.17	2.25	2.33	1.58	1.21	1.14
	1100	2.40	2.37	1.87	1.60	1.22	1.10
	1300	2.71	2.74	2.00	1.54	1.21	1.15
	1500	2.13	2.64	1.89	1.56	1.28	1.08
	1700	2.01	2.35	2.06	1.59	1.22	1.05
2	100	xxxx	3.05	2.65	1.83	1.27	1.14
	300	2.16	3.24	1.76	1.44	1.21	1.20
	500	2.77	2.65	1.85	1.48	1.24	1.18
	700	2.19	2.42	2.22	1.47	1.19	1.15
	900	2.73	2.30	1.93	1.46	1.33	1.11
	1100	2.15	2.64	2.13	1.43	1.28	1.14
	1300	1.87	2.71	2.01	1.47	1.25	1.10
	1500	1.82	2.49	2.08	1.52	1.18	1.07
3	100	1.78	2.95	1.84	1.72	1.10	1.23
	300	2.85	2.35	1.92	1.80	1.10	1.16
	500	2.53	2.56	1.86	1.51	1.22	1.17
	700	1.93	2.77	2.07	1.54	1.21	1.11
	900	2.17	2.32	2.00	1.51	1.21	1.08
4	100	2.62	2.37	2.04	1.45	1.31	1.16
	300	2.53	2.20	2.04	1.54	1.16	1.17
	500	1.71	2.46	1.96	1.64	1.23	1.05

Table F2

Multipliers corresponding to search time increases for increase in query window size for run 2 of initial testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	xxxx	4.35	2.60	1.24	1.16
	300	xxxx	4.21	2.03	1.69	1.24	1.29
	500	2.03	2.99	2.04	1.48	1.31	1.04
	700	2.19	2.18	1.99	1.56	1.34	1.07
	900	1.73	3.40	1.94	1.67	1.19	1.15
	1100	2.19	2.26	2.00	1.57	1.20	1.14
	1300	2.04	2.68	2.13	1.43	1.19	1.17
	1500	1.91	2.78	1.88	1.58	1.23	1.11
	1700	2.15	2.67	2.18	1.42	1.17	1.19
2	100	xxxx	2.59	2.26	1.46	1.31	1.06
	300	1.76	3.03	1.80	1.59	1.23	1.17
	500	2.39	2.75	1.97	1.48	1.31	1.11
	700	2.13	2.39	2.09	1.54	1.25	1.19
	900	2.36	2.79	1.95	1.54	1.21	1.13
	1100	2.14	2.57	2.13	1.49	1.24	1.15
	1300	2.94	3.03	1.86	1.52	1.22	0.96
	1500	2.06	2.42	2.03	1.43	1.37	0.81
3	100	1.39	4.89	2.02	1.56	1.14	1.10
	300	1.88	3.03	2.11	1.46	1.23	1.17
	500	2.00	2.61	1.93	1.54	1.24	1.09
	700	2.32	2.31	1.99	1.45	1.23	1.10
	900	2.67	2.19	2.06	1.53	1.22	0.89
4	100	2.49	3.44	1.90	1.76	1.08	1.19
	300	2.41	2.05	2.24	1.46	1.27	1.06
	500	2.16	2.29	2.07	1.61	1.22	1.09

Table F3
Multipliers corresponding to search time increases for increase in query window size for run 3 of initial testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	xxxx	2.18	1.50	1.21	1.17
	300	xxxx	2.62	3.32	1.64	1.29	1.00
	500	2.68	2.24	2.55	1.55	1.27	1.15
	700	2.83	2.31	1.97	1.49	1.33	1.05
	900	2.21	2.50	2.36	1.42	1.23	1.16
	1100	1.77	2.84	2.22	1.48	1.15	1.21
	1300	2.16	2.32	2.14	1.74	1.14	1.16
	1500	2.02	2.39	1.92	1.51	1.18	1.23
	1700	2.23	2.70	2.16	1.50	1.19	1.14
2	100	xxxx	2.64	1.96	1.77	1.30	1.23
	300	4.22	2.71	1.83	1.52	1.20	1.10
	500	1.98	3.47	2.08	1.57	1.18	1.16
	700	3.26	2.65	2.08	1.53	1.18	1.08
	900	1.96	2.74	2.06	1.52	1.11	1.19
	1100	1.89	2.72	2.00	1.44	1.29	1.05
	1300	2.93	3.04	2.00	1.53	1.22	0.64
	1500	2.32	2.39	2.08	1.43	1.08	0.89
3	100	3.13	2.19	2.43	1.42	1.35	1.12
	300	2.13	2.64	2.06	1.53	1.17	1.29
	500	2.79	2.59	2.00	1.52	1.22	1.18
	700	1.84	3.01	1.92	1.52	1.17	0.99
	900	1.96	2.56	1.96	1.44	1.24	0.76
4	100	2.00	2.42	2.24	1.53	1.24	1.11
	300	3.38	1.94	2.16	1.47	1.32	1.06
	500	2.53	2.55	2.21	1.41	1.14	1.05

Table F4
Multipliers corresponding to search time increases for increase in query window size for all runs of initial testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	xxxx	2.87	1.99	1.28	1.14
	300	xxxx	3.26	2.37	1.69	1.23	1.18
	500	2.42	2.80	2.18	1.51	1.25	1.11
	700	2.33	2.35	2.03	1.47	1.33	1.08
	900	2.04	2.72	2.21	1.55	1.21	1.15
	1100	2.12	2.49	2.03	1.55	1.19	1.15
	1300	2.30	2.58	2.09	1.57	1.18	1.16
	1500	2.02	2.60	1.90	1.55	1.23	1.14
	1700	2.13	2.57	2.13	1.50	1.19	1.13
2	100	xxxx	2.76	2.29	1.69	1.29	1.14
	300	2.71	2.99	1.80	1.51	1.21	1.16
	500	2.38	2.96	1.97	1.51	1.24	1.15
	700	2.53	2.49	2.13	1.51	1.21	1.14
	900	2.35	2.61	1.98	1.51	1.22	1.15
	1100	2.06	2.65	2.09	1.46	1.27	1.11
	1300	2.58	2.93	1.96	1.51	1.23	0.90
	1500	2.07	2.43	2.07	1.46	1.21	0.92
3	100	2.10	3.34	2.10	1.57	1.19	1.15
	300	2.29	2.67	2.03	1.60	1.17	1.21
	500	2.44	2.59	1.93	1.52	1.22	1.15
	700	2.03	2.70	1.99	1.50	1.20	1.07
	900	2.26	2.36	2.01	1.49	1.22	0.91
4	100	2.37	2.74	2.06	1.58	1.21	1.15
	300	2.77	2.06	2.15	1.49	1.25	1.10
	500	2.13	2.43	2.08	1.56	1.20	1.06

APPENDIX G

RAW CONSTRUCTION AND DESTRUCTION TIMES

FOR INITIAL TESTING OF THE K-D RANGE DSL

Table G1(a)
Raw construct times for pass 1 of run 1 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	17	17	16	17	33	0
2	16	17	83	250	234	316	450	383	
3	50	334	550	783	867				
4	183	1416	2400						

Table G1(b)
Raw destruct times for pass 1 of run 1 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	33	17	0	0	16	34
2	0	34	234	166	366	900	1067	333	
3	83	317	1567	2466	5700				
4	116	1833	7517						

Table G2(a)
Raw construct times for pass 2 of run 1 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	17	0	0	0	34	16
2	0	33	150	150	234	217	350	366	
3	133	200	367	884	1000				
4	84	1484	2317						

Table G2(b)
Raw destruct times for pass 2 of run 1 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	16	0	0	17	17	0
2	0	67	200	184	383	883	834	384	
3	50	366	1833	2467	5183				
4	150	2350	7150						

Table G3(a)
Raw construct times for pass 3 of run 1 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	17	0	0	0	33	33
2	0	50	100	117	284	267	283	183	
3	100	200	500	784	933				2
4	100	1234	2566						

Table G3(b)
Raw destruct times for pass 3 of run 1 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	17	16	16	17	17	33	50
2	0	34	234	150	400	1034	1050	266	
3	67	334	1550	1983	5467				
4	117	2333	7200						

Table G4(a)
Raw construct times for pass 4 of run 1 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	0	0	16	16	33	16
2	0	33	66	116	200	334	267	300	
3	0	166	550	1000	967				
4	250	1317	2267						

Table G4(b)
Raw destruct times for pass 4 of run 1 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	33	17	16	16	17	0
2	0	33	200	166	434	900	717	484	
3	50	484	1616	2700	5884				
4	267	2466	7784						

Table G5(a)
Raw construct times for pass 5 of run 1 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	16	33	0	0	0	17
2	0	66	83	217	233	367	216	134	
3	83	383	534	850	1267				
4	184	1717	2433						

Table G5(b)
Raw destruct times for pass 5 of run 1 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	0	0	17	34	17	0
2	0	67	66	133	384	917	783	467	
3	33	450	1750	2667	5000				
4	150	2133	6983						

Table G6(a)
Raw construct times for pass 1 of run 2 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	0	0	0	17	0	33
2	0	50	117	100	100	450	434	483	
3	50	633	584	783	1316				
4	83	1133	2583						

Table G6(b)
Raw destruct times for pass 1 of run 2 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	0	16	0	50	16	0
2	0	0	83	216	283	433	1100	1250	
3	117	683	1883	2700	3900				
4	133	2834	5767						

Table G7(a)
Raw construct times for pass 2 of run 2 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	16	17	0	0	0	17	33
2	0	67	66	100	217	200	317	633	
3	133	617	434	1066	1267				
4	150	1384	2200						

Table G7(b)
Raw destruct times for pass 2 of run 2 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	16	50	0	17	33	33
2	17	17	133	200	384	517	1033	1350	
3	66	717	1900	2500	3517				
4	116	2517	5250						

Table G8(a)
Raw construct times for pass 3 of run 2 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	16	0	0	17	0	0	33	16	17
2	0	16	83	150	150	233	383	583	
3	83	634	384	1133	1250				
4	134	933	2050						

Table G8(b)
Raw destruct times for pass 3 of run 2 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	16	16	16	17	33	17
2	0	33	183	234	300	584	1050	1333	
3	66	800	1666	2467	3516				
4	150	2450	5266						

Table G9(a)
Raw construct times for pass 4 of run 2 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	0	0	33	0	16	50
2	0	0	67	150	150	350	466	600	
3	184	700	517	950	1183				
4	183	1083	2383						

Table G9(b)
Raw destruct times for pass 4 of run 2 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	17	0	0	0	0	16	0	34
2	17	0	133	266	350	400	1217	1284	
3	66	700	1667	2850	3133				
4	117	2334	5433						

Table G10(a)
Raw construct times for pass 5 of run 2 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	16	0	0	0	0	0	17	0	16
2	0	34	67	100	200	283	350	583	
3	67	583	566	833	1167				
4	150	950	2384						

Table G10(b)
Raw destruct times for pass 5 of run 2 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	16	0	17	17	0	17	17	34
2	33	33	116	283	316	516	933	1400	
3	83	684	1700	2816	4000				
4	134	2166	6050						

Table G11(a)
Raw construct times for pass 1 of run 3 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	16	34	17	0	16	16	17
2	0	50	67	167	217	300	350	517	
3	117	466	700	934	1283				
4	150	1133	2450						

Table G11(b)
Raw destruct times for pass 1 of run 3 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	0	0	17	0	0	0
2	0	17	100	183	317	833	1333	1816	
3	116	933	1150	2417	3184				
4	183	2500	5617						

Table G12(a)
Raw construct times for pass 2 of run 3 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	17	0	17	0	16	0	17
2	0	33	67	200	150	317	367	533	
3	150	567	267	984	1183				
4	217	1234	2666						

Table G12(b)
Raw destruct times for pass 2 of run 3 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	50	33	33	0	0	0
2	17	33	117	183	267	983	1284	1833	
3	150	900	1233	2433	3084				
4	150	2683	6067						

Table G13(a)
Raw construct times for pass 3 of run 3 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	16	0	0	0	16	17
2	0	50	83	133	216	267	450	550	
3	50	533	533	1066	1250				
4	200	1317	2400						

Table G13(b)
Raw destruct times for pass 3 of run 3 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	17	0	0	0	0	33	50	0
2	0	67	83	233	316	1183	1566	1750	
3	117	950	1183	2650	3400				
4	200	2766	6433						

Table G14(a)
Raw construct times for pass 4 of run 3 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	0	0	0	17	0	50	33
2	17	50	50	117	184	283	384	500	
3	133	600	466	883	1367				
4	133	1400	2700						

Table G14(b)
Raw destruct times for pass 4 of run 3 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	17	17	0	16	17	17	0	0
2	16	50	66	200	184	900	1317	1833	
3	134	300	1267	2583	3267				
4	233	2817	6633	g					

Table G15(a)
Raw construct times for pass 5 of run 3 of initial testing of the k-d Range DSL

dim	Raw construct times for data set sizes given in Table 1								
1	0	0	17	0	0	0	0	50	17
2	34	33	84	150	117	233	300	517	
3	116	200	700	933	1200				
4	250	1600	2584						

Table G15(b)
Raw destruct times for pass 5 of run 3 of initial testing of the k-d Range DSL

dim	Raw destruct times for data set sizes given in Table 1								
1	0	0	0	0	0	33	16	17	17
2	0	33	33	150	216	934	1250	1784	
3	150	316	1233	2516	3083				
4	117	3517	6033						

APPENDIX H

AVERAGE CONSTRUCTION AND DESTRUCTION TIMES FOR EACH INITIAL TEST RUN OF THE K-D RANGE DSL

Table H1(a)
Construct time averages for run 1 of initial testing of the k-d Range DSL

dim	Construct time averages for data set sizes given in Table 1								
1	0	0	0	13	10	6	6	26	16
2	3	39	96	170	237	300	313	273	
3	73	256	500	860	1006				
4	160	1433	2396						

Table H1(b)
Destruct time averages for run 1 of initial testing of the k-d Range DSL

dim	Destruct time averages for data set sizes given in Table 1								
1	0	0	3	19	10	10	16	20	16
2	0	47	186	159	393	926	890	386	
3	56	390	1663	2456	5446				
4	160	2223	7326						

Table H2(a)
Construct time averages for run 2 of initial testing of the k-d Range DSL

dim	Construct time averages for data set sizes given in Table 1								
1	6	0	3	6	0	6	13	9	29
2	0	33	80	120	163	303	390	576	
3	103	633	497	953	1236				
4	140	1096	2320						

Table H2(b)
Destruct time averages for run 2 of initial testing of the k-d Range DSL

dim	Destruct time averages for data set sizes given in Table 1								
1	0	6	0	9	19	3	23	19	23
2	13	16	129	239	326	490	1066	1323	
3	79	716	1763	2666	3613				
4	130	2460	5553						

Table H3(a)
Construct time averages for run 3 of initial testing of the k-d Range DSL

dim	Construct time averages for data set sizes given in Table 1								
1	0	0	10	10	6	3	6	26	20
2	10	43	70	153	176	280	370	523	
3	113	473	533	960	1256				
4	190	1336	2560						

Table H3(b)
Destruct time averages for run 3 of initial testing of the k-d Range DSL

dim	Destruct time averages for data set sizes given in Table 1								
1	0	6	3	10	9	20	13	13	3
2	6	40	79	189	260	966	1350	1803	
3	133	679	1213	2519	3203				
4	176	2856	6156						

Table H4(a)
Construct time averages for all runs of initial testing of the k-d Range DSL

dim	Construct time averages for data set sizes given in Table 1								
1	2	0	4	10	5	5	8	20	22
2	4	38	82	148	192	294	358	457	
3	96	454	510	924	1166				
4	163	1288	2425						

Table H4(b)
Destruct time averages for all runs of initial testing of the k-d Range DSL

dim	Destruct time averages for data set sizes given in Table 1								
1	0	4	2	13	13	11	17	17	14
2	6	34	131	196	326	794	1102	1171	
3	89	595	1546	2547	4087				
4	155	2513	6345						

APPENDIX I

RAW INSERTION AND DELETION CONSTANTS FOR INITIAL TESTING OF THE K-D RANGE DSL

Table I1(a)

Constants for Insertions of pass 1 of run 1 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	25.70	19.25	14.40	12.64	20.85	0.00
2	36.25	8.37	20.65	39.98	27.00	28.14	32.35	22.94	
3	17.05	19.98	15.26	13.25	10.19				
4	9.39	10.29	7.43						

Table I1(b)

Constants for Deletions of pass 1 of run 1 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	49.88	19.25	0.00	0.00	10.11	18.64
2	0.00	16.74	58.22	26.55	42.22	80.15	76.70	19.94	
3	28.30	18.96	43.48	41.73	67.01				
4	5.95	13.33	23.27						

Table I2(a)

Constants for Insertions of pass 2 of run 1 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	25.70	0.00	0.00	0.00	21.48	8.77
2	0.00	16.24	37.32	23.99	27.00	19.33	25.16	21.92	
3	45.35	11.96	10.18	14.96	11.76				
4	4.31	10.79	7.17						

Table I2(b)

Constants for Deletions of pass 2 of run 1 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	24.18	0.00	0.00	12.64	10.74	0.00
2	0.00	32.98	49.76	29.43	44.19	78.64	59.95	23.00	
3	17.05	21.90	50.87	41.75	60.93				
4	7.70	17.08	22.13						

Table I3(a)

Constants for Insertions of pass 3 of run 1 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	25.70	0.00	0.00	0.00	20.85	18.09
2	0.00	24.61	24.88	18.71	32.76	23.78	20.34	10.96	
3	34.10	11.96	13.88	13.27	10.97				
4	5.13	8.97	7.94						

Table I3(b)
Constants for Deletions of pass 3 of run 1 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	37.92	24.18	18.12	15.30	12.64	20.85	27.41
2	0.00	16.74	58.22	23.99	46.15	92.09	75.48	15.93	
3	22.85	19.98	43.01	33.56	64.27				
4	6.00	16.96	22.28						

Table I4(a)
Constants for Insertions of pass 4 of run 1 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	14.40	11.90	20.85	8.77
2	0.00	16.24	16.42	18.55	23.07	29.75	19.19	17.97	
3	0.00	9.93	15.26	16.92	11.37				
4	12.83	9.57	7.02						

Table I4(b)
Constants for Deletions of pass 4 of run 1 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	49.88	19.25	14.40	11.90	10.74	0.00
2	0.00	16.24	49.76	26.55	50.07	80.15	51.54	28.99	
3	17.05	28.95	44.84	45.69	69.17				
4	13.70	17.93	24.09						

Table I5(a)
Constants for Insertions of pass 5 of run 1 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	24.18	37.36	0.00	0.00	0.00	9.32
2	0.00	32.49	20.65	34.70	26.88	32.68	15.53	8.03	
3	28.30	22.91	14.82	14.38	14.89				
4	9.44	12.48	7.53						

Table I5(b)
Constants for Deletions of pass 5 of run 1 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	15.30	25.28	10.74	0.00
2	0.00	32.98	16.42	21.27	44.30	81.67	56.29	27.97	
3	11.25	26.92	48.56	45.13	58.78				
4	7.70	15.51	21.61						

Table I6(a)
Constants for Insertions of pass 1 of run 2 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	0.00	12.64	0.00	18.09
2	0.00	24.61	29.11	15.99	11.54	40.08	31.20	28.93	
3	17.05	37.87	16.21	13.25	15.47				
4	4.26	8.24	7.99						

Table I6(b)
Constants for Deletions of pass 1 of run 2 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	18.12	0.00	37.18	10.11	0.00
2	0.00	0.00	20.65	34.54	32.65	38.56	79.08	74.86	
3	39.90	40.86	52.25	45.69	45.85				
4	6.83	20.60	17.85						

Table I7(a)
Constants for Insertions of pass 2 of run 2 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	35.69	25.70	0.00	0.00	0.00	10.74	18.09
2	0.00	32.98	16.42	15.99	25.03	17.81	22.79	37.91	
3	45.35	36.91	12.04	18.04	14.89				
4	7.70	10.06	6.81						

Table I7(b)
Constants for Deletions of pass 2 of run 2 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	24.18	56.61	0.00	12.64	20.85	18.09
2	38.51	8.37	33.09	31.99	44.30	46.04	74.26	80.85	
3	22.51	42.89	52.73	42.30	41.34				
4	5.95	18.30	16.25						

Table I8(a)
Constants for Insertions of pass 3 of run 2 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	240.82	0.00	0.00	25.70	0.00	0.00	24.54	10.11	9.32
2	0.00	7.88	20.65	23.99	17.31	20.75	27.53	34.91	
3	28.30	37.93	10.66	19.17	14.69				
4	6.88	6.78	6.34						

Table I8(b)

Constants for Deletions of pass 3 of run 2 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	24.18	18.12	14.40	12.64	20.85	9.32
2	0.00	16.24	45.53	37.42	34.61	52.01	75.48	79.83	
3	22.51	47.86	46.23	41.75	41.33				
4	7.70	17.81	16.30						

Table I9(a)

Constants for Insertions of pass 4 of run 2 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	29.69	0.00	10.11	27.41
2	0.00	0.00	16.67	23.99	17.31	31.17	33.50	35.93	
3	62.74	41.88	14.35	16.08	13.91				
4	9.39	7.87	7.38						

Table I9(b)

Constants for Deletions of pass 4 of run 2 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	68.86	0.00	0.00	0.00	0.00	11.90	0.00	18.64
2	38.51	0.00	33.09	42.54	40.38	35.62	87.49	76.90	
3	22.51	41.88	46.26	48.23	36.83				
4	6.00	16.97	16.82						

Table I10(a)

Constants for Insertions of pass 5 of run 2 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	240.82	0.00	0.00	0.00	0.00	0.00	12.64	0.00	8.77
2	0.00	16.74	16.67	15.99	23.07	25.20	25.16	34.91	
3	22.85	34.88	15.71	14.10	13.72				
4	7.70	6.91	7.38						

Table I10(b)

Constants for Deletions of pass 5 of run 2 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	64.81	0.00	25.70	19.25	0.00	12.64	10.74	18.64
2	74.76	16.24	28.86	45.26	36.46	45.95	67.07	83.84	
3	28.30	40.92	47.18	47.65	47.02				
4	6.88	15.75	18.73						

Table I11(a)
Constants for Insertions of pass 1 of run 3 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	35.69	51.39	19.25	0.00	11.90	10.11	9.32
2	0.00	24.61	16.67	26.71	25.03	26.72	25.16	30.96	
3	39.90	27.88	19.43	15.80	15.08				
4	7.70	8.24	7.58						

Table I11(b)
Constants for Deletions of pass 1 of run 3 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	15.30	0.00	0.00	0.00
2	0.00	8.37	24.88	29.27	36.57	74.19	95.83	108.76	
3	39.55	55.81	31.91	40.90	37.43				
4	9.39	18.17	17.39						

Table I12(a)
Constants for Insertions of pass 2 of run 3 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	37.92	0.00	19.25	0.00	11.90	0.00	9.32
2	0.00	16.24	16.67	31.99	17.31	28.23	26.38	31.92	
3	51.15	33.92	7.41	16.65	13.91				
4	11.14	8.97	8.25						

Table I12(b)
Constants for Deletions of pass 2 of run 3 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	75.58	37.36	29.69	0.00	0.00	0.00
2	38.51	16.24	29.11	29.27	30.80	87.55	92.30	109.78	
3	51.15	53.84	34.22	41.17	36.25				
4	7.70	19.51	18.78						

Table I13(a)
Constants for Insertions of pass 3 of run 3 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	24.18	0.00	0.00	0.00	10.11	9.32
2	0.00	24.61	20.65	21.27	24.92	23.78	32.35	32.94	
3	17.05	31.89	14.79	18.04	14.69				
4	10.26	9.57	7.43						

Table I13(b)
Constants for Deletions of pass 3 of run 3 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	68.86	0.00	0.00	0.00	0.00	24.54	31.59	0.00
2	0.00	32.98	20.65	37.26	36.46	105.36	112.58	104.80	
3	39.90	56.83	32.83	44.84	39.97				
4	10.26	20.11	19.91						

Table I14(a)
Constants for Insertions of pass 4 of run 3 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	15.30	0.00	31.59	18.09
2	38.51	24.61	12.44	18.71	21.23	25.20	27.60	29.94	
3	45.35	35.89	12.93	14.94	16.07				
4	6.83	10.18	8.36						

Table I14(b)
Constants for Deletions of pass 4 of run 3 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	68.86	37.92	0.00	18.12	15.30	12.64	0.00	0.00
2	36.25	24.61	16.42	31.99	21.23	80.15	94.68	109.78	
3	45.69	17.95	35.16	43.71	38.41				
4	11.96	20.48	20.53						

Table I15(a)
Constants for Insertions of pass 5 of run 3 of initial testing of the k-d Range DSL

dim	Insertion constants for the data set sizes given by Table 1.								
1	0.00	0.00	37.92	0.00	0.00	0.00	0.00	31.59	9.32
2	77.03	16.24	20.90	23.99	13.50	20.75	21.57	30.96	
3	39.55	11.96	19.43	15.79	14.11				
4	12.83	11.63	8.00						

Table I15(b)
Constants for Deletions of pass 5 of run 3 of initial testing of the k-d Range DSL

dim	Deletion constants for the data set sizes given by Table 1.								
1	0.00	0.00	0.00	0.00	0.00	29.69	11.90	10.74	9.32
2	0.00	16.24	8.21	23.99	24.92	83.18	89.86	106.84	
3	51.15	18.90	34.22	42.57	36.24				
4	6.00	25.57	18.67						

APPENDIX J

INSERTION AND DELETION CONSTANTS

FOR EACH INITIAL TEST RUN

OF THE K-D RANGE DSL

Table J1(1)
Constants for insertion averages of run 1 of initial testing of the k-d Range DSL

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	0.00	0.00	0.00	19.65	11.32	5.40	4.46	16.43	8.77
2	6.80	19.20	23.88	27.19	27.34	26.72	22.50	16.35	
3	24.89	15.31	13.88	14.55	11.83				
4	8.21	10.42	7.42						

Table J1(b)
Constants for deletion averages of run 1 of initial testing of the k-d Range DSL

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	0.00	6.69	28.72	11.32	9.00	11.90	12.64	8.77
2	0.00	23.14	46.28	25.43	45.34	82.47	63.98	23.12	
3	19.10	23.33	46.15	41.56	64.02				
4	8.21	16.16	22.67						

Table J2(a)
Standard deviation for constants for insertion averages of run 1 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J1(a).								
1	0.00	0.00	0.00	11.36	16.77	7.90	6.74	9.41	6.40
2	16.22	9.23	8.03	9.71	3.47	5.24	6.49	6.62	
3	17.29	5.72	2.14	1.51	1.80				
4	3.50	1.34	0.36						

Table J2(b)
Standard deviation for constants for deletion averages of run 1 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J1(b).								
1	0.00	0.00	16.99	20.99	10.35	8.22	8.97	4.60	13.00
2	0.00	8.99	17.33	3.08	2.97	5.45	11.45	5.47	
3	6.49	4.38	3.42	4.84	4.26				
4	3.19	1.81	0.99						

Table J3(a)
Constants for insertion averages of run 2 of initial testing of the k-d Range DSL

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	90.31	0.00	6.69	9.07	0.00	5.40	9.67	5.69	15.90
2	0.00	16.24	19.90	19.19	18.80	26.99	28.04	34.50	
3	35.12	37.87	13.79	16.13	14.53				
4	7.19	7.97	7.18						

Table J3(b)
Constants for deletion averages of run 2 of initial testing of the k-d Range DSL

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	24.30	0.00	13.60	21.51	2.70	17.10	12.01	12.61
2	29.45	7.88	32.10	38.22	37.61	43.64	76.63	79.23	
3	26.94	42.83	48.92	45.11	42.47				
4	6.67	17.88	17.19						

Table J4(a)
Standard deviation for constants for insertion averages of run 2 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J3(a).								
1	132.07	0.00	15.97	14.14	0.00	13.29	10.32	5.69	7.68
2	0.00	13.08	5.44	4.38	5.34	8.88	4.36	3.36	
3	18.65	2.55	2.38	2.52	0.72				
4	1.87	1.32	0.63						

Table J4(b)
Standard deviation for constants for deletion averages of run 2 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J3(b).								
1	0.00	36.74	0.00	13.60	20.75	6.44	11.07	8.74	8.26
2	31.43	8.13	9.00	5.50	4.67	6.54	7.45	3.49	
3	7.56	2.90	3.28	2.99	4.08				
4	0.72	1.80	1.07						

Table J5(a)
Constants for insertion averages of run 3 of initial testing of the k-d Range DSL

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	0.00	0.00	22.31	15.12	6.79	2.70	4.46	16.43	10.96
2	22.65	21.17	17.42	24.47	20.30	24.94	26.60	31.32	
3	38.53	28.30	14.79	16.24	14.77				
4	9.75	9.71	7.92						

Table J5(b)
Constants for deletion averages of run 3 of initial testing of the k-d Range DSL

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	24.30	6.69	15.12	10.19	18.00	9.67	8.21	1.64
2	13.59	19.69	19.66	30.23	30.00	86.03	97.05	107.98	
3	45.35	40.62	33.66	42.63	37.65				
4	9.03	20.76	19.05						

Table J6(a)
Standard deviation for constants for insertion averages of run 3 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J5(a).								
1	0.00	0.00	20.38	22.82	10.59	6.85	6.53	14.23	3.92
2	34.45	4.59	3.48	5.13	4.99	2.87	3.92	1.13	
3	12.94	9.61	5.02	1.17	0.86				
4	2.47	1.29	0.41						

Table J6(b)
Standard deviation for constants for deletion averages of run 3 of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J5(b).								
1	0.00	37.89	16.99	33.80	16.68	12.37	10.27	13.74	4.18
2	20.55	9.39	8.05	4.83	6.86	11.82	8.98	2.15	
3	5.72	20.33	1.29	1.67	1.58				
4	2.30	2.82	1.21						

Table J7(a)
Constants for insertion averages of all runs of initial testing of the k-d Range DSL

dim	Constants for insertion averages for data set sizes given by Table 1.								
1	30.10	0.00	9.67	14.61	6.04	4.50	6.20	12.85	11.88
2	9.82	18.87	20.40	23.62	22.15	26.21	25.71	27.39	
3	32.85	27.16	14.15	15.64	13.71				
4	8.38	9.37	7.51						

Table J7(b)
Constants for deletion averages of all runs of initial testing of the k-d Range DSL

dim	Constant for deletion averages for data set sizes given by Table 1.								
1	0.00	16.20	4.46	19.15	14.34	9.90	12.89	10.95	7.67
2	14.35	16.90	32.68	31.29	37.65	70.71	79.22	70.11	
3	30.46	35.59	42.91	43.10	48.05				
4	7.97	18.27	19.64						

Table J8(a)
Standard deviation for constants for insertion averages of all runs of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J7(a).								
1	44.02	0.00	12.12	16.11	9.12	9.35	7.86	9.78	6.00
2	16.89	8.97	5.65	6.41	4.60	5.66	4.92	3.70	
3	16.29	5.96	3.18	1.73	1.13				
4	2.61	1.32	0.47						

Table J8(b)
Standard deviation for constants for deletion averages of all runs of initial testing of the k-d Range DSL

dim	Standard deviations of constants in Table J7(b).								
1	0.00	24.88	11.33	22.80	15.93	9.01	10.10	9.03	8.48
2	17.33	8.84	11.46	4.47	4.83	7.94	9.29	3.70	
3	6.59	9.20	2.66	3.17	3.31				
4	2.07	2.14	1.09						

Table J9(a)
Standard deviations for the values of Table 15

dimensions	run 1	run 2	run 3	all runs
2	190.96	195.41	205.33	197.23
3	64.60	62.90	60.87	62.79
4	26.45	24.64	22.69	24.59

Table J9(b)
Standard deviations for the values of Table 16

dimensions	run 1	run 2	run 3	all runs
2	2.28	1.81	2.14	2.07
3	0.60	0.88	0.49	0.66
4	0.33	0.08	0.14	0.18

APPENDIX K

AVERAGE SEARCH TIMES FOR EACH INITIAL TEST RUN OF THE K-D RANGE DSL

Table K1
Average search times for run 1 of initial testing of the k-d Range DSL

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average Search Times							
1	100	7	23	80	177	290	446	550	564
	300	0	63	170	466	934	1383	1620	1680
	500	10	167	343	750	1533	2340	2953	3273
	700	13	233	414	1060	2127	3263	4163	4494
	900	0	317	557	1410	2883	4507	5477	5367
	1100	7	297	603	1587	2983	4690	5550	5890
	1300	0	403	910	2217	4350	6673	7757	8723
	1500	0	307	740	1600	3397	4740	5920	5087
	1700	10	356	760	1880	3960	6000	7263	3280
2	100	13	74	116	203	513	727	867	940
	300	7	173	363	866	1670	2557	3100	3563
	500	17	353	587	1543	2767	4390	5107	5963
	700	7	257	547	1400	2800	4387	5167	4717
	900	13	267	563	1450	2733	4007	4970	2587
	1100	10	500	887	2293	4573	6793	5746	3217
	1300	13	480	956	2297	3983	6370	4680	4020
	1500	10	583	1313	3167	6517	9607	4207	4877
3	100	17	67	147	380	693	1193	1517	1503
	300	20	250	543	1344	2826	4357	5210	5370
	500	10	317	653	1567	3017	4293	5040	4603
	700	7	386	644	1563	3263	4877	5120	2930
	900	13	483	1153	2893	5313	8307	4204	3633
4	100	26	100	177	434	917	1460	1750	1920
	300	10	310	700	1473	3267	4947	5540	6600
	500	17	537	1083	2647	5463	8080	9973	10770

Table K2
Average search times for run 2 of initial testing of the k-d Range DSL

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average Search Times							
1	100	10	40	50	130	257	463	516	627
	300	7	123	170	427	877	1366	1717	1940
	500	3	170	317	690	1537	2197	2700	3100
	700	3	200	476	1074	2093	3347	4023	4467
	900	0	330	590	1387	2910	4403	5460	5827
	1100	3	294	700	1723	3737	5364	6813	7550
	1300	7	470	776	2084	4290	6527	7717	8933
	1500	0	427	950	2473	4866	7324	8617	9533
	1700	3	523	977	2780	5520	7810	9717	10700
2	100	10	50	126	227	463	777	1047	1054
	300	6	180	343	780	1620	2517	3057	3337
	500	17	297	590	1380	2860	4334	4920	5467
	700	10	453	743	1934	3920	5620	6877	7770
	900	17	510	1037	2397	4997	7747	8700	9680
	1100	17	583	1223	3183	5893	9140	10597	11797
	1300	3	527	1033	2563	5090	7667	9087	8703
	1500	7	453	927	2357	4563	6927	8503	5913
3	100	20	60	150	427	820	1310	1590	1647
	300	10	287	497	1257	2477	3800	4367	4957
	500	17	444	820	2010	4240	6313	7467	8817
	700	24	580	1140	2980	5747	8610	10883	12303
	900	14	493	1236	2677	5620	8457	9923	8473
4	100	17	93	173	470	946	1317	1683	1760
	300	13	337	690	1550	3267	4947	5540	6600
	500	17	537	1083	2647	5463	8080	9973	10770

Table K3
Average search times for run 3 of initial testing of the k-d Range DSL

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average Search Times							
1	100	13	27	47	87	270	404	587	527
	300	3	77	190	400	783	1310	1630	1753
	500	0	130	286	734	1420	2217	2554	2966
	700	13	233	437	990	1977	3127	3797	4153
	900	10	287	570	1410	2740	4287	4870	5547
	1100	13	353	710	1743	3443	4870	6470	6780
	1300	6	340	740	2250	4197	6053	7550	8467
	1500	17	497	1083	2573	5180	7547	8963	10153
	1700	7	610	1097	2627	5580	7870	9593	10707
2	100	10	63	90	223	550	803	957	1167
	300	7	170	393	807	1643	2593	3077	3420
	500	17	303	560	1480	2813	4327	5200	5737
	700	0	443	890	2123	4047	5790	7017	7823
	900	20	543	1094	2507	5177	7290	8910	10080
	1100	10	687	1303	3167	6233	9323	11357	12133
	1300	23	777	1550	3720	8037	11223	13577	15013
	1500	10	717	1393	3117	6337	9113	10480	9634
3	100	27	90	133	350	640	1017	1227	1416
	300	20	277	536	1170	2513	3537	4463	4913
	500	7	413	903	2057	4250	6097	7307	8090
	700	20	573	1170	2843	5670	8787	9977	11427
	900	20	847	1660	3897	7627	11697	14057	15363
4	100	27	93	190	557	1130	1440	1803	1883
	300	17	310	700	1510	3177	4493	5600	6217
	500	27	587	977	2394	5327	7887	9513	10350

Table K4
Average search times for all runs of initial testing of the k-d Range DSL

% of pts		0%	5%	10%	25%	50%	75%	90%	100%
dim	# pts	Average Search Times							
1	100	10	30	59	131	272	438	551	572
	300	3	88	177	431	865	1353	1656	1791
	500	4	156	315	725	1497	2251	2736	3113
	700	10	222	442	1041	2066	3246	3995	4371
	900	3	311	572	1402	2844	4399	5269	5580
	1100	8	315	671	1684	3388	4975	6278	6740
	1300	4	404	809	2183	4279	6418	7674	8708
	1500	6	410	924	2216	4481	6537	7833	8258
	1700	7	497	944	2429	5020	7227	8858	8229
2	100	11	62	111	218	509	769	957	1053
	300	7	175	367	818	1644	2556	3078	3440
	500	17	318	579	1468	2813	4350	5076	5722
	700	6	384	727	1819	3589	5266	6353	6770
	900	17	440	898	2118	4302	6348	7527	7449
	1100	12	590	1138	2881	5567	8419	9233	9049
	1300	13	595	1180	2860	5703	8420	9114	9246
	1500	9	584	1211	2880	5806	8549	7730	6808
3	100	21	72	143	386	718	1173	1444	1522
	300	17	271	526	1257	2605	3898	4680	5080
	500	11	391	792	1878	3836	5568	6604	7170
	700	17	513	985	2462	4893	7424	8660	8887
	900	16	608	1350	3156	6187	9487	9394	9157
4	100	23	96	180	487	998	1405	1745	1855
	300	13	319	697	1511	3237	4796	5560	6472
	500	20	553	1048	2562	5418	8016	9820	10630

APPENDIX L

MULTIPLIERS REPRESENTING SEARCH TIME INCREASES IN THE INITIAL TEST RUNS OF THE K-D RANGE DSL

Table L1
Multipliers corresponding to search time increases for increase in query window size for run 1 of initial testing

Expected:		2.0	2.5	2.0	1.5	1.2	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	2.45	1.71	1.60	1.31	1.06
	300	xxxx	2.95	2.06	1.51	1.18	1.04
	500	2.07	2.26	2.11	1.54	1.27	1.12
	700	1.79	2.65	2.03	1.55	1.28	1.09
	900	1.75	2.66	2.08	1.57	1.22	0.97
	1100	2.01	2.68	1.91	1.56	1.21	1.04
	1300	2.36	2.46	1.96	1.54	1.17	1.13
	1500	3.37	2.07	2.14	1.33	1.29	0.95
	1700	5.02	2.71	2.09	1.53	1.25	0.63
2	100	xxxx	1.98	2.44	1.71	1.17	1.24
	300	2.37	2.60	1.96	1.56	1.22	1.16
	500	1.69	2.70	1.80	1.60	1.17	1.18
	700	2.08	2.49	1.93	1.55	1.21	0.95
	900	2.00	2.45	1.94	1.54	1.21	0.79
	1100	1.70	2.66	2.06	1.48	0.92	0.72
	1300	1.98	2.31	1.75	1.61	0.99	0.88
	1500	2.74	2.24	2.19	1.53	0.60	1.16
3	100	xxxx	2.93	1.69	2.07	1.26	0.98
	300	2.37	2.57	2.11	1.55	1.20	1.03
	500	2.25	2.23	2.18	1.43	1.18	0.99
	700	2.01	2.55	2.04	1.48	1.16	0.75
	900	2.59	2.47	1.79	1.67	0.71	0.89
4	100	2.06	2.51	2.30	1.55	1.24	1.07
	300	2.33	2.14	2.24	1.53	1.12	1.19
	500	2.04	2.47	2.07	1.48	1.24	1.08

Table L2
Multipliers corresponding to search time increases for increase in query window size for run 2 of initial testing

Expected:		2.0	2.5	2.0	1.5	1.2	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	2.90	2.79	1.80	1.13	1.24
	300	xxxx	2.63	2.13	1.60	1.29	1.15
	500	2.16	2.25	2.26	1.46	1.23	1.15
	700	2.39	2.40	1.98	1.61	1.21	1.11
	900	1.87	2.40	2.16	1.52	1.24	1.07
	1100	2.41	2.49	2.17	1.44	1.28	1.11
	1300	1.74	2.69	2.06	1.52	1.19	1.16
	1500	2.29	2.63	1.98	1.51	1.18	1.11
	1700	1.87	2.88	1.99	1.42	1.25	1.10
2	100	xxxx	1.88	2.09	1.72	1.37	1.01
	300	2.04	2.49	2.10	1.56	1.22	1.11
	500	2.08	2.36	2.09	1.53	1.14	1.12
	700	1.73	2.60	2.04	1.44	1.22	1.14
	900	2.07	2.32	2.10	1.55	1.12	1.11
	1100	2.13	2.62	1.85	1.55	1.16	1.11
	1300	2.36	2.53	2.00	1.52	1.18	1.01
	1500	2.33	2.53	1.88	1.56	1.24	0.88
3	100	xxxx	3.25	1.95	1.64	1.22	1.05
	300	1.93	2.59	2.02	1.55	1.15	1.14
	500	1.94	2.48	2.14	1.49	1.19	1.18
	700	1.99	2.62	1.93	1.50	1.26	1.13
	900	2.44	2.09	2.13	1.49	1.22	0.90
4	100	1.83	3.04	2.21	1.44	1.26	1.05
	300	2.10	2.26	2.13	1.53	1.12	1.19
	500	2.04	2.47	2.07	1.48	1.24	1.08

Table L3
Multipliers corresponding to search time increases for increase in query window size for run 3 of initial testing

Expected:		2.0	2.5	2.0	1.5	1.2	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	2.84	3.90	1.53	1.47	0.92
	300	xxxx	2.12	2.04	1.76	1.28	1.10
	500	2.30	2.59	1.95	1.59	1.15	1.18
	700	1.94	2.30	2.08	1.59	1.23	1.10
	900	2.07	2.57	1.95	1.57	1.14	1.14
	1100	2.11	2.50	2.01	1.42	1.33	1.05
	1300	2.30	3.15	1.88	1.44	1.25	1.12
	1500	2.28	2.38	2.02	1.46	1.19	1.14
	1700	1.81	2.44	2.13	1.41	1.22	1.12
2	100	xxxx	3.93	2.55	1.49	1.22	1.23
	300	2.60	2.11	2.09	1.59	1.19	1.11
	500	1.87	2.65	1.94	1.55	1.21	1.11
	700	2.04	2.38	1.95	1.43	1.21	1.12
	900	2.07	2.31	2.07	1.42	1.22	1.13
	1100	1.93	2.45	1.99	1.50	1.22	1.07
	1300	2.00	2.40	2.17	1.40	1.21	1.11
	1500	1.79	2.39	2.04	1.42	1.15	1.00
3	100	xxxx	13.57	2.23	1.54	1.24	1.17
	300	2.04	2.27	2.20	1.42	1.27	1.10
	500	2.19	2.33	2.08	1.44	1.20	1.11
	700	2.06	2.44	2.00	1.55	1.14	1.15
	900	1.97	2.35	1.97	1.54	1.20	1.09
4	100	2.11	3.42	2.06	1.28	1.26	1.06
	300	2.39	2.16	2.13	1.42	1.25	1.11
	500	1.72	2.51	2.24	1.48	1.21	1.09

Table L4
Multipliers corresponding to search time increases for increase in query window size for all runs of initial testing

Expected:		2.0	2.5	2.0	1.5	1.2	1.11
dim	# pts			Actual Multipliers			
1	100	xxxx	2.73	2.80	1.64	1.30	1.07
	300	xxxx	2.57	2.08	1.62	1.25	1.10
	500	2.18	2.37	2.11	1.53	1.22	1.15
	700	2.04	2.45	2.03	1.58	1.24	1.10
	900	1.90	2.54	2.06	1.55	1.20	1.06
	1100	2.18	2.56	2.03	1.47	1.27	1.07
	1300	2.13	2.76	1.97	1.50	1.20	1.14
	1500	2.65	2.36	2.05	1.43	1.22	1.07
	1700	2.90	2.68	2.07	1.45	1.24	0.95
2	100	xxxx	2.59	2.36	1.64	1.25	1.16
	300	2.34	2.40	2.05	1.57	1.21	1.13
	500	1.88	2.57	1.94	1.56	1.17	1.13
	700	1.95	2.49	1.98	1.47	1.22	1.07
	900	2.05	2.36	2.03	1.50	1.19	1.01
	1100	1.92	2.58	1.97	1.51	1.10	0.97
	1300	2.12	2.41	1.97	1.51	1.13	1.00
	1500	2.29	2.39	2.04	1.50	1.00	1.01
3	100	xxxx	6.59	1.96	1.75	1.24	1.07
	300	2.11	2.48	2.11	1.50	1.21	1.09
	500	2.13	2.35	2.13	1.45	1.19	1.09
	700	2.02	2.54	1.99	1.51	1.19	1.01
	900	2.33	2.30	1.96	1.57	1.04	0.96
4	100	2.00	2.99	2.19	1.42	1.25	1.06
	300	2.27	2.19	2.17	1.49	1.17	1.17
	500	1.93	2.48	2.12	1.48	1.23	1.08

APPENDIX M

AVERAGE CONSTRUCTION AND DESTRUCTION TIMES FOR EACH FURTHER TEST RUN OF THE K-D RANGE AVL TREE

Table M1(a)
Averages for construct times of run 1 of further testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 2.		
1	2756	2311	2505
2	4372	2895	3105
3	10878	8911	11767
4	27234	24200	27272

Table M1(b)
Averages for destruct times of run 1 of further testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 2.		
1	50	61	72
2	67	78	50
3	111	117	150
4	311	400	394

Table M2(a)
Averages for construct times of run 2 of further testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 2.		
1	2611	1989	2372
2	4484	3044	3272
3	10361	9267	11205
4	26233	22123	24783

Table M2(b)
Averages for destruct times of run 2 of further testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 2.		
1	39	78	83
2	39	67	89
3	167	139	217
4	122	111	156

Table M3(a)
Averages for construct times of run 3 of further testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 2.		
1	5856	4250	4417
2	4228	3034	3600
3	12267	10300	11700
4	26933	23450	26917

Table M3(b)
Averages for destruct times of run 3 of further testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 2.		
1	67	84	89
2	61	117	78
3	133	61	117
4	117	133	494

Table M4(a)
Averages of construct times for all runs of further testing of the k-d Range AVL Tree

dim	Construct time averages for data set sizes given in Table 2.		
1	3741	2850	3098
2	4361	2991	3326
3	11168	9493	11557
4	26800	23258	26324

Table M4(b)
Averages of destruct times for all runs of further testing of the k-d Range AVL Tree

dim	Destruct time averages for data set sizes given in Table 2.		
1	52	74	82
2	56	87	72
3	137	106	161
4	183	215	348

APPENDIX N

INSERTION AND DELETION CONSTANTS

FOR EACH FURTHER TEST RUN

OF THE K-D RANGE AVL TREE

Table N1(a)
Constants for insertion averages of run 1 of further testing of the Range AVL Tree

dim	Constants for insertion averages of data set sizes of Table 2		
1	976.52	666.91	608.00
2	181.79	90.88	77.58
3	61.75	33.79	32.73
4	27.61	13.02	9.43

Table N1(b)
Constants for deletion averages of run 1 of further testing of the Range AVL Tree

dim	Constants for deletion averages of data set sizes of Table 2		
1	17.72	17.60	17.55
2	2.77	2.44	1.24
3	0.63	0.44	0.42
4	0.32	0.22	0.14

Table N2(a)
Standard deviations of constants for insertion averages of run 1 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N1(a).		
1	1584.59	1084.99	983.51
2	294.13	149.73	129.01
3	90.58	51.77	51.47
4	43.14	20.89	15.30

Table N2(b)
Standard deviations of constants for deletion averages of run 1 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N1(b).		
1	20.56	19.39	20.80
2	2.40	3.33	1.11
3	0.86	0.55	0.37
4	0.46	0.33	0.20

Table N3(a)
Constants for insertion averages of run 2 of further testing of the Range AVL Tree

dim	Constants for insertion averages of data set sizes of Table 2		
1	925.25	573.89	575.64
2	186.43	95.57	81.76
3	58.81	35.14	31.16
4	26.60	11.90	8.57

Table N3(b)
Constants for deletion averages of run 2 of further testing of the Range AVL Tree

dim	Constants for deletion averages of data set sizes of Table 2		
1	13.82	22.41	20.22
2	1.62	2.10	2.22
3	0.95	0.53	0.60
4	0.12	0.06	0.05

Table N4(a)
Standard deviations of constants for insertion averages of run 2 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N3(a).		
1	1390.98	883.59	927.94
2	300.26	154.37	131.59
3	85.00	54.63	49.04
4	41.58	18.75	13.59

Table N4(b)
Standard deviations of constants for deletion averages of run 2 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N3(b).		
1	12.34	15.34	24.60
2	1.03	2.10	2.29
3	1.32	0.59	0.72
4	0.11	0.07	0.05

Table N5(a)
Constants for insertion averages of run 3 of further testing of the Range AVL Tree

dim	Constants for insertion averages of data set sizes of Table 2		
1	2075.18	1226.38	1071.93
2	175.79	95.24	89.93
3	69.63	39.06	32.54
4	27.30	12.62	9.31

Table N5(b)
Constants for deletion averages of run 3 of further testing of the Range AVL Tree

dim	Constants for deletion averages of data set sizes of Table 2		
1	23.62	24.14	21.60
2	2.54	3.66	1.94
3	0.76	0.23	0.33
4	0.12	0.07	0.17

Table N6(a)
Standard deviations of constants for insertion averages of run 3 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N5(a).		
1	3578.66	2120.15	1832.08
2	300.88	162.22	152.89
3	91.76	49.11	41.75
4	36.79	17.89	13.38

Table N6(b)
Standard deviations of constants for deletion averages of run 3 of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N5(b).		
1	20.46	25.52	11.77
2	1.03	3.44	1.96
3	0.82	0.35	0.30
4	0.12	0.06	0.23

Table N7(a)
Constants for insertion averages of all runs of further testing of the Range AVL Tree

dim	Constants for insertion averages of data set sizes of Table 2		
1	1325.65	822.39	751.86
2	181.34	93.90	83.09
3	63.39	35.99	32.14
4	27.17	12.51	9.10

Table N7(b)
Constants for deletion averages of all runs of further testing of the Range AVL Tree

dim	Constants for deletion averages of data set sizes of Table 2		
1	18.39	21.39	19.79
2	2.31	2.73	1.80
3	0.78	0.40	0.45
4	0.19	0.12	0.12

Table N8(a)
Standard deviations of constants for insertion averages of all runs of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N7(a).		
1	2184.74	1362.91	1247.84
2	298.42	155.44	137.83
3	89.11	51.83	47.42
4	40.50	19.18	14.09

Table N8(b)
Standard deviations of constants for deletion averages of all runs of further testing of the Range AVL Tree

dim	Standard deviations of constants in Table N7(b).		
1	17.79	20.08	19.06
2	1.49	2.95	1.78
3	1.00	0.50	0.47
4	0.23	0.15	0.16

Table N9(a)
Standard deviations for the values of Table 9

dimensions	run 1	run 2	run 3	all runs
2	6.97	6.41	3.73	5.70
3	2.80	2.04	4.17	3.00
4	0.85	0.98	0.85	0.90

Table N9(b)
Standard deviations for the values of Table 10

dimensions	run 1	run 2	run 3	all runs
2	7.82	6.40	7.44	7.22
3	4.23	3.31	6.22	4.59
4	1.40	1.44	2.02	1.62

APPENDIX O

AVERAGE SEARCH TIMES FOR EACH FURTHER TEST RUN OF THE K-D RANGE AVL TREE

TABLE O1

Average search times for run 1 of further testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	6	478	1144	2784	5239	8177	9856	11450
	3000	6	556	1217	3283	6239	9561	11800	13067
	3500	0	778	1350	3894	7372	11478	13450	14806
2	2000	22	689	1500	3511	7506	10650	13144	15128
	2500	0	878	1683	4367	8616	13967	16883	18939
	3000	11	995	2278	5311	10350	16483	19056	22011
3	1500	6	800	1461	4067	7928	12322	14178	15956
	2000	11	1044	2222	5556	10250	15834	19250	21522
	2500	0	1044	2067	4933	10456	14622	18200	15700
4	1000	11	467	1184	2617	5078	7272	8928	10066
	1500	6	1250	2334	4861	10394	15033	18378	20556
	2000	6	1422	2300	5756	11672	18061	20933	12978

TABLE O2

Average search times for run 2 of further testing of the k-d Range AVL Tree

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	11	655	1083	2689	5439	8394	10428	11333
	3000	6	600	1484	3561	6805	9917	12511	13789
	3500	0	789	1372	3922	7656	11366	14039	15955
2	2000	6	833	1517	3477	6973	11327	13573	15922
	2500	6	839	1922	4511	9194	13856	16678	18117
	3000	0	1094	2245	5033	10700	17105	20700	22750
3	1500	11	1000	1600	4072	8222	12138	15078	16811
	2000	6	1016	2100	5283	10294	16422	20133	22233
	2500	22	978	2089	5361	10650	16206	19517	13427
4	1000	6	561	1122	2544	5306	7889	9523	10283
	1500	11	1239	2150	5695	10845	15494	18467	21139
	2000	5	1123	2656	5544	10600	16345	19555	10872

TABLE O3

Average search times for run 3 of further testing of the k-d Range AVL Tree

dim	dpts	Average search times							
1	2500	6	528	989	2789	5478	8355	9689	11233
	3000	17	583	1094	3450	6494	9900	11661	13039
	3500	6	622	1444	3778	7606	12017	13889	14867
2	2000	0	606	1450	3606	7033	11145	12900	15378
	2500	6	922	2083	4722	9072	13767	17228	18944
	3000	5	1255	2234	5456	11200	16844	21272	23295
3	1500	11	733	1828	4644	8672	13011	16339	17867
	2000	5	1250	2478	5961	11383	17583	21956	23894
	2500	11	1122	2317	5905	11422	17467	20983	15700
4	1000	11	856	1183	3367	6650	9333	11395	12655
	1500	6	1139	2289	5533	11172	16539	19428	22277
	2000	16	1345	2622	6133	11855	17817	20795	12883

TABLE O4

Average search times for all runs of further testing of the k-d Range AVL Tree

dim	dpts	Average search times							
1	2500	7	554	1072	2754	5385	8309	9991	11339
	3000	9	580	1265	3431	6513	9793	11991	13298
	3500	2	730	1389	3865	7544	11620	13793	15209
2	2000	9	709	1489	3531	7171	11041	13206	15476
	2500	4	880	1896	4533	8961	13863	16930	18667
	3000	5	1115	2252	5267	10750	16811	20343	22685
3	1500	9	844	1630	4261	8274	12491	15198	16878
	2000	7	1104	2267	5600	10642	16613	20446	22550
	2500	11	1048	2158	5400	10843	16098	19567	14942
4	1000	9	628	1163	2843	5678	8165	9948	11002
	1500	7	1209	2257	5363	10804	15689	18758	21324
	2000	9	1297	2526	5811	11376	17407	20428	12245

APPENDIX P

MULTIPLIERS CORRESPONDING TO SEARCH TIME INCREASES FOR INCREASES IN QUERY WINDOW SIZE FOR FURTHER TESTING OF THE RANGE AVL TREE

Table P1

Multipliers corresponding to search time increases for increases in query window size for run 1 in further testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.95	2.44	1.89	1.56	1.21	1.16
	3000	2.20	2.72	1.91	1.54	1.24	1.11
	3500	1.74	2.90	1.90	1.56	1.17	1.10
2	2000	2.18	2.37	2.14	1.42	1.23	1.15
	2500	1.95	2.62	1.98	1.63	1.21	1.12
	3000	2.29	2.33	1.95	1.59	1.16	1.16
3	1500	1.84	2.86	1.95	1.56	1.15	1.13
	2000	2.14	2.51	1.85	1.55	1.22	1.12
	2500	2.04	2.48	2.16	1.35	1.26	0.97
4	1000	2.45	2.14	1.98	1.47	1.23	1.16
	1500	1.87	2.09	2.15	1.45	1.22	1.12
	2000	1.54	2.50	1.97	1.58	1.13	0.75

Table P2

Multipliers corresponding to search time increases for increases in query window size for run 2 in further testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	1.63	2.61	2.03	1.54	1.24	1.10
	3000	2.63	2.48	1.92	1.46	1.26	1.11
	3500	1.74	2.89	1.96	1.49	1.24	1.14
2	2000	1.83	2.33	2.01	1.63	1.20	1.17
	2500	2.33	2.35	2.04	1.51	1.21	1.09
	3000	2.09	2.25	2.14	1.60	1.21	1.10
3	1500	1.62	2.56	2.03	1.48	1.24	1.12
	2000	2.11	2.52	1.95	1.60	1.23	1.11
	2500	2.25	2.57	1.92	1.53	1.19	0.78
4	1000	2.29	2.19	2.06	1.51	1.16	1.11
	1500	1.79	2.66	1.91	1.43	1.19	1.15
	2000	2.24	2.12	1.91	1.51	1.19	0.69

Table P3

Multipliers corresponding to search time increases for increases in query window size for run 3 in further testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.00	2.87	1.97	1.53	1.16	1.16
	3000	1.87	3.20	1.89	1.53	1.18	1.12
	3500	2.36	2.64	2.02	1.58	1.16	1.07
2	2000	2.45	2.52	1.95	1.58	1.16	1.19
	2500	2.33	2.27	1.95	1.52	1.25	1.10
	3000	1.82	2.44	2.06	1.50	1.26	1.09
3	1500	2.54	2.54	1.87	1.50	1.26	1.09
	2000	2.00	2.44	1.91	1.55	1.25	1.09
	2500	2.11	2.44	1.98	1.55	1.19	0.83
4	1000	1.43	2.87	2.00	1.36	1.27	1.10
	1500	2.08	2.42	2.02	1.48	1.18	1.15
	2000	2.32	2.31	1.89	1.51	1.16	0.73

Table P4

Multipliers corresponding to search time increases for increases in query window size for all runs in further testing of the k-d Range AVL Tree

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.20	2.64	1.96	1.54	1.20	1.14
	3000	2.23	2.80	1.91	1.51	1.23	1.11
	3500	1.95	2.81	1.96	1.54	1.19	1.10
2	2000	2.15	2.40	2.03	1.54	1.20	1.17
	2500	2.20	2.41	1.99	1.55	1.22	1.10
	3000	2.07	2.34	2.05	1.57	1.21	1.12
3	1500	2.00	2.66	1.95	1.51	1.22	1.11
	2000	2.08	2.49	1.90	1.56	1.23	1.10
	2500	2.13	2.50	2.02	1.48	1.22	0.86
4	1000	2.06	2.40	2.01	1.45	1.22	1.12
	1500	1.91	2.39	2.02	1.46	1.20	1.14
	2000	2.03	2.31	1.92	1.53	1.16	0.73

APPENDIX Q

AVERAGE CONSTRUCTION AND DESTRUCTION TIMES FOR EACH FURTHER TEST RUN OF THE K-D RANGE DSL

Table Q1(a)
Construction time averages for run 1 of further testing of the k-d Range DSL

dim	Average construction time for data set sizes given in Table 2.		
1	33	28	55
2	300	539	694
3	3367	9761	23600
4	5156	12884	18195

Table Q1(b)
Destruction time averages for run 1 of further testing of the k-d Range DSL

dim	Average destruction time for data set sizes given in Table 2.		
1	22	28	28
2	1028	1556	2961
3	16911	69900	246845
4	39297	197139	382009

Table Q2(a)
Construction time averages for run 2 of further testing of the k-d Range DSL

dim	Average construction time for data set sizes given in Table 2.		
1	11	56	28
2	411	500	650
3	3339	10434	22239
4	5267	13313	17634

Table Q2(b)
Destruction time averages for run 2 of further testing of the k-d Range DSL

dim	Average destruction time for data set sizes given in Table 2.		
1	44	44	39
2	672	1672	3084
3	19816	71389	129611
4	41152	200396	368111

Table Q3(a)
Construction time averages for run 3 of further testing of the k-d Range DSL

dim	Average construction time for data set sizes given in Table 2.		
1	22	56	34
2	439	472	756
3	3672	11667	24511
4	4833	14216	17460

Table Q3(b)
Destruction time averages for run 3 of further testing of the k-d Range DSL

dim	Average destruction time for data set sizes given in Table 2.		
1	27	33	22
2	833	1550	2500
3	25955	101361	186022
4	39710	192130	374917

Table Q4(a)
Construction time averages for all runs of further testing of the k-d Range DSL

dim	Average construction time for data set sizes given in Table 2.		
1	22	46	39
2	383	504	700
3	3459	10620	23450
4	5085	13471	17763

Table Q4(b)
Destruction time averages for all runs of further testing of the k-d Range DSL

dim	Average destruction time for data set sizes given in Table 2.		
1	31	35	30
2	844	1593	2848
3	20894	80883	187493
4	40053	196555	375012

APPENDIX R

INSERTION AND DELETION CONSTANTS

FOR EACH FURTHER TEST RUN

OF THE K-D RANGE DSL

Table R1(a)
Constants for average insertions of run 1 of further testing of the k-d Range DSL

dim	Average insertion constants for data set sizes of Table 2.		
1	11.81	8.08	13.43
2	12.47	16.91	17.35
3	19.11	37.01	65.64
4	5.23	6.93	6.29

Table R1(b)
Constants for average deletions of run 1 of further testing of the k-d Range DSL

dim	Average deletion constants for data set sizes of Table 2.		
1	7.80	7.98	6.80
2	42.74	48.84	73.98
3	95.99	265.05	686.54
4	39.84	106.06	132.09

Table R2(a)
Standard deviations of constants for average insertions of run 1 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of Table R1(a).		
1	0.20	2.75	4.70
2	1.19	0.78	1.27
3	4.50	10.58	12.16
4	0.27	1.40	0.37

Table R2(b)
Standard deviations of constants for average deletions of run 1 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of table R1(b).		
1	9.05	7.34	2.32
2	5.02	4.46	1.34
3	18.40	72.18	117.70
4	3.76	7.39	7.31

Table R3(a)
Constants for average insertions of run 2 of further testing of the k-d Range DSL

dim	Average insertion constants for data set sizes of Table 2.		
1	4.02	16.06	6.71
2	17.10	15.70	16.24
3	18.95	39.56	61.85
4	5.34	7.16	6.10

Table R3(b)
Constants for average deletions of run 2 of further testing of the k-d Range DSL

dim	Average deletion constants for data set sizes of Table 2.		
1	15.59	12.79	9.46
2	27.94	52.50	77.04
3	112.48	270.70	360.48
4	41.72	107.81	127.29

Table R4(a)
Standard deviations of constants for average insertions of run 2 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of Table R3(a).		
1	3.48	10.21	6.17
2	3.15	1.04	3.41
3	4.12	10.85	7.06
4	0.15	1.22	0.19

Table R4(b)
Standard deviations of constants for average deletions of run 2 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of table R3(b).		
1	9.05	9.94	6.04
2	6.28	8.34	4.41
3	19.92	65.74	41.97
4	3.59	2.78	2.83

Table R5(a)
Constants for average insertions of run 3 of further testing of the k-d Range DSL

dim	Average insertion constants for data set sizes of Table 2.		
1	7.91	16.06	8.17
2	18.25	14.83	18.88
3	20.84	44.24	68.17
4	4.90	7.65	6.04

Table R5(b)
Constants for average deletions of run 3 of further testing of the k-d Range DSL

dim	Average deletion constants for data set sizes of Table 2.		
1	9.69	9.62	5.34
2	34.64	48.66	62.45
3	147.33	384.35	517.38
4	40.26	103.36	129.64

Table R6(a)
Standard deviations of constants for average insertions of run 3 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of Table R5(a).		
1	9.01	2.83	8.13
2	6.28	2.89	2.85
3	6.51	12.23	9.63
4	1.17	1.02	0.17

Table R6(b)
Standard deviations of constants for average deletions of run 3 of further testing of the k-d Range DSL

dim	Standard deviations of the data values of table R5(b).		
1	12.20	4.76	4.62
2	3.67	7.19	5.42
3	31.79	76.17	66.17
4	3.10	9.35	7.04

Table R7(a)
Constants for average insertions of all runs of further testing of the k-d Range DSL

dim	Average insertion constants for data set sizes of Table 2.		
1	7.91	13.40	9.44
2	15.94	15.81	17.49
3	19.64	40.27	65.22
4	5.16	7.25	6.14

Table R7(b)
Constants for average deletions of all runs of further testing of the k-d Range DSL

dim	Average deletion constants for data set sizes of Table 2.		
1	11.02	10.13	7.20
2	35.11	50.00	71.16
3	118.60	306.70	521.47
4	40.61	105.74	129.68

Table R8(a)
Standard deviations of constants for average insertions of all runs of further testing of the k-d Range DSL

dim	Standard deviations of the data values of Table R7(a).		
1	4.23	5.27	6.33
2	3.54	1.57	2.51
3	5.04	11.22	9.62
4	0.53	1.22	0.24

Table R8(b)
Standard deviations of constants for average deletions of all runs of further testing of the k-d Range DSL

dim	Standard deviations of the data values of table R7(b).		
1	10.10	7.34	4.33
2	4.99	6.66	3.72
3	23.37	71.37	75.28
4	3.48	6.51	5.72

Table R9(a)
Standard deviations for the values of Table 19

dimensions	run 1	run 2	run 3	all runs
2	1.08	2.53	4.01	2.54
3	9.08	7.34	9.46	8.63
4	0.68	0.52	0.79	0.66

Table R9(b)
Standard deviations for the values of Table 20

dimensions	run 1	run 2	run 3	all runs
2	3.61	6.34	5.43	5.12
3	69.43	42.54	58.04	56.67
4	6.15	3.07	6.50	5.24

APPENDIX S

AVERAGE SEARCH TIMES FOR EACH FURTHER TEST RUN OF THE K-D RANGE DSL

TABLE S1
Average search times for run 1 of further testing of the k-d Range DSL

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	6	700	1641	4339	8450	12533	14511	16989
	3000	6	933	1939	4844	10189	15544	17645	20684
	3500	5	1161	2094	5772	11417	18061	20939	23439
2	2000	11	867	1855	4811	6961	14305	17434	17006
	2500	6	972	1839	4267	8566	12711	15739	10816
	3000	6	972	2600	6483	13128	20433	17577	8683
3	1500	6	700	1278	3473	6383	9511	12444	9411
	2000	23	1422	3111	6750	14044	20389	19786	8839
	2500	11	1250	2089	5606	11372	17355	12328	11250
4	1000	22	444	1622	3595	8300	12100	14061	14428
	1500	11	961	2322	6206	11600	17850	17055	7306
	2000	17	1589	2648	6614	12640	18833	13489	14039

TABLE S2
Average search times for run 2 of further testing of the k-d Range DSL

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	0	189	322	745	1539	2428	2856	2917
	3000	11	228	361	1011	2100	3128	3516	3878
	3500	16	323	433	1167	2289	3417	4017	4400
2	2000	11	255	522	1317	2294	3661	4555	4639
	2500	17	278	550	1644	3228	4489	5578	6372
	3000	22	411	822	1833	3889	5611	6528	7172
3	1500	0	339	616	1495	2939	4239	5234	5533
	2000	11	339	950	2011	3489	5515	6866	7161
	2500	11	622	1378	2550	5178	7733	9367	10122
4	1000	11	272	444	1161	2317	3556	4439	4811
	1500	17	517	950	2066	3945	5383	6628	7678
	2000	17	1767	3483	8322	15400	22578	10194	9767

TABLE S3

Average search times for run 3 of further testing of the k-d Range DSL

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	6	200	439	1144	2261	3317	4161	4350
	3000	0	289	455	1272	2750	3778	4366	4655
	3500	6	300	595	1483	2956	4497	5095	6061
2	2000	11	361	581	1517	3117	4500	5472	5744
	2500	11	433	1098	2142	3961	5906	7278	7466
	3000	5	461	1100	2556	4671	7300	8428	9217
3	1500	11	339	728	1566	3350	5150	6185	6511
	2000	11	478	1022	2409	4594	6878	7900	8385
	2500	11	617	1100	2995	5405	7611	10150	10889
4	1000	11	367	644	1483	2861	4189	4833	5620
	1500	16	378	944	2366	4606	6289	7850	8478
	2000	28	2539	5205	8651	23617	36725	12950	11767

TABLE S4

Average search times for all runs of further testing of the k-d Range DSL

% of points		0%	5%	10%	25%	50%	75%	90%	100%
dim	dpts	Average search times							
1	2500	4	363	801	2076	4083	6093	7176	8085
	3000	5	483	919	2376	5013	7484	8509	9739
	3500	9	595	1041	2807	5554	8658	10017	11300
2	2000	11	495	986	2548	4124	7489	9154	9130
	2500	11	561	1162	2684	5252	7702	9532	8218
	3000	11	615	1507	3624	7229	11115	10844	8357
3	1500	6	459	874	2178	4224	6300	7954	7152
	2000	15	746	1694	3723	7376	10927	11517	8128
	2500	11	830	1522	3717	7319	10900	10615	10754
4	1000	15	361	904	2080	4493	6615	7778	8286
	1500	15	619	1406	3546	6717	9841	10511	7820
	2000	20	1965	3779	7863	17219	26045	12211	11857

APPENDIX T

MULTIPLIERS CORRESPONDING TO SEARCH TIME INCREASES FOR INCREASES IN QUERY WINDOW SIZE FOR FURTHER TESTING OF THE K-D RANGE DSL

Table T1

Multipliers corresponding to search time increases for increases in query window size for run 1 in further testing of the k-d Range DSL

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.36	2.65	1.95	1.49	1.16	1.17
	3000	2.08	2.50	2.11	1.53	1.14	1.17
	3500	1.81	2.76	1.98	1.58	1.16	1.12
2	2000	2.44	2.54	1.57	2.29	1.20	1.02
	2500	1.83	2.29	2.05	1.49	1.24	0.88
	3000	3.12	2.46	2.03	1.54	0.97	0.62
3	1500	1.79	2.83	1.80	1.43	1.29	0.93
	2000	2.11	2.26	2.05	1.43	1.03	0.60
	2500	1.62	3.01	2.06	1.48	1.02	0.94
4	1000	7.41	2.18	2.42	1.48	1.16	1.01
	1500	2.89	2.81	1.88	1.56	1.06	0.56
	2000	1.65	2.54	1.84	1.49	0.94	1.03

Table T2

Multipliers corresponding to search time increases for increases in query window size for run 2 in further testing of the k-d Range DSL

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	1.96	2.37	2.09	1.58	1.18	1.03
	3000	1.86	2.81	2.10	1.51	1.12	1.11
	3500	1.44	2.92	1.97	1.50	1.18	1.10
2	2000	2.05	2.54	1.75	1.62	1.25	1.02
	2500	2.03	2.99	1.96	1.40	1.24	1.15
	3000	2.00	2.26	2.14	1.45	1.16	1.10
3	1500	1.90	2.63	1.98	1.45	1.26	1.06
	2000	3.00	2.59	1.74	1.58	1.24	1.05
	2500	2.22	1.86	2.06	1.50	1.22	1.08
4	1000	1.63	2.73	2.03	1.54	1.26	1.08
	1500	1.81	2.25	1.92	1.38	1.23	1.17
	2000	2.02	2.45	1.85	1.48	0.65	0.96

Table T3

Multipliers corresponding to search time increases for increases in query window size for run 3 in further testing of the k-d Range DSL

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.57	2.61	1.99	1.47	1.26	1.05
	3000	1.59	2.85	2.16	1.39	1.16	1.07
	3500	2.16	2.60	2.00	1.52	1.14	1.19
2	2000	1.67	2.71	2.06	1.45	1.22	1.05
	2500	2.60	2.10	1.90	1.49	1.23	1.03
	3000	2.48	2.32	1.83	1.56	1.15	1.10
3	1500	2.18	2.21	2.16	1.55	1.20	1.06
	2000	2.16	2.37	1.91	1.50	1.16	1.06
	2500	1.80	2.72	1.83	1.40	1.38	1.07
4	1000	1.97	2.31	1.94	1.46	1.15	1.17
	1500	2.52	2.55	1.98	1.37	1.25	1.08
	2000	2.06	1.59	7.81	1.56	0.35	0.91

Table T4

Multipliers corresponding to search time increases for increases in query window size for all runs in further testing of the k-d Range DSL

Expected:		2.0	2.5	2.0	1.5	1.20	1.11
dim	points			Multipliers			
1	2500	2.30	2.54	2.01	1.51	1.20	1.08
	3000	1.84	2.72	2.12	1.48	1.14	1.12
	3500	1.80	2.76	1.98	1.53	1.16	1.14
2	2000	2.05	2.60	1.79	1.78	1.22	1.03
	2500	2.15	2.46	1.97	1.46	1.24	1.02
	3000	2.54	2.35	2.00	1.52	1.10	0.94
3	1500	1.95	2.56	1.98	1.48	1.25	1.02
	2000	2.42	2.41	1.90	1.50	1.14	0.91
	2500	1.88	2.53	1.99	1.46	1.21	1.03
4	1000	3.67	2.41	2.13	1.50	1.19	1.09
	1500	2.41	2.54	1.93	1.44	1.18	0.94
	2000	1.91	2.19	3.83	1.51	0.65	0.97

APPENDIX U

COMPLETE SOURCE CODE FOR THE TEST DRIVER ROUTINES FOR THE K-D RANGE AVL TREE

```
program dynamic_kd_range_avl_tree;
{this program contains the test drivers to test the k-d range AVL tree}

{datapts are read in from a datafile which follows a naming convention:
(also, the query windows are precomputed for a data set and stored in a
 file as well)

 [xxxx][d][r][np].dat

 where [xxxx] is either 'dpts' for a data set
                    or 'rqpr' for a file containing range query windows
       [d]    a single digit denoting how many dimensions to the data
       [r]    a single digit; the run number in which structure is built
       [np]   two digits: signifies the number of datapoints in the file
              # dpts in file = np * basenumpts
              (if np < 10 then it contains a leading 0)                  }

const numsearches = 100; {used to test rangesearch}
      numper      =   8;  {how many query windows?}

      dptnam    = 'dpts'; {contains datapoints for testing}
      dptext    = '.dat';  {contains datapoints for testing}

      rqfnam    = 'rqpr'; {contains range query windows}
      rqfext    = '.dat';  {contains range query windows}

      maxdim    =      4; {max num dimensions of our structure}
      maxdset   =      9; {test for max this many datasets per dimension}

      rsearch2file = 'rqres.dat'; {results of range queries}

      basenumpts  = 100; {all datafiles have a # of points
                          that is a multiple of this number}

type  coordinate = array [1..maxdim] of integer;
      dpfsize    = array [1..maxdim, 1..maxdset] of integer;

var t:  tree;              {pointer to our tree}
    numpoints: integer;    {how many datapoints for test run?}
    dimensions: integer;   {how many dimensions for test run?}
    sv: coordinate;        {non-leaf nodes contain this value}

    testpar: char;       {initial (i) or further (f) testing?}
    ni: coordinate;      {the number of data sets per dimension}
    ns: dpfsize;         {the sizes of the data sets}

    passes: integer;     {the number of passes over a single structure}
    sdim, edim: integer; {the starting and ending dimension of our test}
    srun, erun: integer; {the starting and ending run of our test}
    sset: integer;       {the starting multiplier of our test}

    k,i,j,r: integer;    {loop control variables}
                         {r: runs  k: dimensions  i:data set  j: passes}
    xfile: text;         {construct / destruct times file variable}
    rfile: text;         {search times file variable}

    rqfile: text;        {temporary datafile for searches file variable}

    resfile,
    rsearchfile: string; {names of output files}
```

```
function achr(k: integer): char;
{used to convert a digit to a character}

begin {achr}
  case k of
    0: achr:= '0';
    1: achr:= '1';
    2: achr:= '2';
    3: achr:= '3';
    4: achr:= '4';
    5: achr:= '5';
    6: achr:= '6';
    7: achr:= '7';
    8: achr:= '8';
    9: achr:= '9';
  end;
end;  {achr}

function bchr(k, c: integer): char;
{used to convert a number between 0 and 99 to two characters}

begin
  if c = 1 then
    bchr:= achr(k div 10)
  else
    bchr:= achr(k mod 10);
end;

procedure builddrt(var t: nodeptr; numdim, ptndx, run: integer);

var ndp: coordinate; {the datapoint being inserted}
    nn: nodeptr;     {the new node to be inserted}
    taller: boolean; {used by insertdrt, has height of tree increased?}
    dim: integer;    {current dimension in which to insert new node}
    dfile: text;     {file varaible of file containing datapoints}
    i,j: integer;    {loop control variables}
    st, et: integer; {get starting and ending times, report difference}

begin {builddrt}
  {prepare the file for input}
  open(dfile,
dptnam+achr(numdim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+dptext,'old');
  reset  (dfile);

  st:= sysclock;
    for i:= 1 to numpoints do
      begin {insert next datapoint}
        dim:= 1; {we first insert the datapoint in the first dimension}
        for j:= 1 to numdim do
          read(dfile, ndp[j]);      {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);
      end;  {insert next datapoint}
  et:= sysclock;

  close  (dfile);    {we have tested our structure}
  writeln(xfile, (et-st));
end;  {builddrt}
```

```
procedure testrs(t: nodeptr; numdim, ptndx, run: integer);

var dim:   integer;
    i,j,p: integer;
    l,r:   coordinate;
    dfile: text;
    st, et: integer;

begin {testrs}
  open(dfile,
rqfnam+achr(numdim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+rqfext,'old');
  reset  (dfile);

  rewrite(rqfile);

  for p:= 1 to numper do
    begin {for p}
      write(p:2, '  '); {track progression of testing}

      {read in lower left and upper right corners of query window}
      for j:= 1 to dimensions do
        read (dfile, l[j], r[j]);
      readln (dfile);

      {we time for numsearches}
      st:= sysclock;
        for i:= 1 to numsearches do
          begin {for i}
            dim:= 1;
            rangesearch(t,l,r,dim);
          end;  {for i}
      et:= sysclock;
      writeln(rfile,  (et - st));
    end;  {for p}

  close  (rqfile);
  close  (dfile);

end;  {testrs}
```

```
procedure destroydrt(var t: nodeptr; numdim, ptndx, run: integer);

var dp:       coordinate; {the datapoint being deleted}
    dim:      integer;    {current dimension to delete node from}
    dfile:    text;       {file variable of file contianing points}
    i,j: integer;         {loop control variable}
    reduced: boolean;     {recursive control in delete}
    st, et: integer;      {get start and end times and output difference}

begin {destroydrt}
  {prepare file for input}
  open(dfile,
dptnam+achr(numdim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+dptext,'old');
  reset  (dfile);

  st:= sysclock;
    for i:= 1 to numpoints do
      begin {delete next datapoint}
        dim:= 1;  {we first delete datapoint from dimension 1}
        reduced:= false;
        for j:= 1 to numdim do
          read(dfile, dp[j]);          {read in the datapoint}
       delete(dp, t, reduced, dim);  {delete next point from the drt}
      end;  {delete the next datapoint}
  et:= sysclock;

  close  (dfile);  {we have tested our structure}
  writeln(xfile, (et - st));
end;  {destroydrt}

begin {main}
  {determine if we are to perform initial or further testing}
  writeln ('initial (i) or further (f) testing?');
  readln  (testpar);
  {get parameters which govern test - may perform partial or full test}
  writeln ('how many passes per run?');
  readln  (passes);
  writeln ('starting and ending runs?');
  readln  (srun, erun);
  writeln ('starting and ending dimensions?');
  readln  (sdim, edim);
  writeln ('starting multiplier?');
  readln  (sset);
 {sentinel value stored in interior node}
  sv[1]:= 0;  sv[2]:= 0;  sv[3]:= 0;  sv[4]:= 0;
  if testpar = i then
    begin
      ni[1]:= 9;  ni[2]:= 8;  ni[3]:= 5;  ni[4]:= 3;
      ns[1,1]:= 1; ns[1,2]:= 3; ns[1,3]:= 5; ns[1,4]:= 7; ns[1,5]:= 9;
      ns[1,6]:= 11;  ns[1,7]:= 13;  ns[1,8]:= 15;  ns[1,9]:= 17;
      ns[2,1]:= 1; ns[2,2]:= 3; ns[2,3]:= 5; ns[2,4]:= 7; ns[2,5]:= 9;
      ns[2,6]:= 11;  ns[2,7]:= 13;  ns[2,8]:= 15;
      ns[3,1]:= 1; ns[3,2]:= 3; ns[3,3]:= 5; ns[3,4]:= 7; ns[3,5]:= 9;
      ns[4,1]:=  1;  ns[4,2]:=  3;  ns[4,3]:=  5;
    end
  else
    begin
      ni[1]:= 3;  ni[2]:= 3;  ni[3]:= 3;  ni[4]:= 3;
      ns[1,1]:= 25; ns[1,2]:= 30; ns[1,3]:= 35;
      ns[2,1]:= 20; ns[2,2]:= 25; ns[2,3]:= 30;
      ns[3,1]:= 15; ns[3,2]:= 20; ns[3,3]:= 25;
      ns[4,1]:= 10; ns[4,2]:= 15; ns[4,3]:= 20;
    end
```

```
    writeln('enter name of datafile to hold construct/destruct times.');
    readln (resfile);
    writeln('please enter name of datafile to hold search times.');
    readln (rsearchfile);

    {open datafile to hold construct/destruct times}
    open (xfile, resfile, 'new');
    rewrite (xfile);
    {open datafile to hold search times}
    open (rfile, rsearchfile, 'new');
    rewrite (rfile);

    {temporary datafile used by rangesearch procedures to report results}
    open (rqfile, rsearch2file, 'new');

    {main test loop}
    for r:= srun to erun do
      for k:= sdim to edim do
        for i:= sset to ni[k] do
          for j:= 1 to passes do
            begin {for j}
              numpoints:= ns[k,i]*basenumpts;  {# datapoints in dataset}
              dimensions:= k;    {how many dimensions will our tree have?}

              {output to track progression of testing}
              writeln ('ready. set. gone.  run: ', r:2, 'dim: ', k:2,
                       ' points: ', numpoints);

              t:= nil;                          {initialize tree as empty}

              write ('building      ');
                builddrt(t,k,(ns[k,i], r);    {build a tree}
              write ('searching     ');
                testrs(t,k,ns[k,i], r);       {perform some range searches}
              write ('annihilating  ');
                destroydrt(t,k,ns[k,i], r);   {destroy the tree}
            end;  {for j}

    {close result datafiles}
    close (rfile);
    close (xfile);
  end.  {main}
```

APPENDIX V

COMPLETE SOURCE CODE FOR THE TEST DRIVER ROUTINES FOR THE K-D RANGE DSL

```
program rangedsl;
{this program contains the test drivers to test the k-d Range DSL}

{datapts are read in from a datafile that follows a naming convention:
(also, the query windows are precomputed for a data set and stored in a
 file as well)

 [xxxx][d][r][np].dat

 where [xxxx] is either 'dpts' for a data set
                     or 'rqpr' for a file containing range query windows
       [d]    a single digit denoting how many dimensions to the data
       [r]    a single digit; run number in which the structure is built
       [np]   two digits; signify the number of datapoints in the file
              # dpts in file = np * basenumpts
              (if np < 10 then it contains a leading 0)}

const maxdim = 4;    {maximum number of dimensions}
      maxdsets = 9;  {max data sets in a dimension}

      dptnam  = 'dpts';  {contains datapoints for testing}
      dptext  = '.dat';  {contains datapoints for testing}

      rqfnam  = 'rqpr';  {contains range query windows}
      rqfext  = '.dat';  {contains range query windows}

      basenumpts:= 100; {all data sets are multiples of this many pts}

      mx      = maxkey;  {abbreviation for maxkey}
      mn      = minkey;  {abbreviation for minkey}

     numper     =    8;  {this many query windows for range search}
     numsearch =    1000; {average over this many searches}

type  crdptr      = ^coordinate;
      coordinate = array [1..maxdim] of integer; {stores a datapoint}

      dsetsizes = array [1..maxdim, 1..maxdsets] of integer;

var sv, amax, amin: crdptr; {sentinel value for non-leaf nodes, amin
                             for bottom node, amax for tail node}
    head, tail,
    bottom, lastdim: nodeptr; {sentinel nodes of the rdsl}

    testpar: char;      {initial (i) or further (f) testing}
    passes: integer;    {number of passes}
    ndss: integer;      {starting run}
    numdsets: integer;  {ending run}
    ntds: integer;      {starting dim}
    ntestdim: integer;  {ending dim}
    sm: integer;        {starting multiplier}
    ni: coordinate;     {stores number of datasets in a dimension}
    ns: dsetsizes;      {stores multipliers that give *pts in data sets}

    numpoints: integer;  {the number of data points of our structure}
    dimensions: integer; {the number of dimensions of the data}

    bldname: string;  {name of file storing building/destruction times}
    srchname: string; {name of file that stores search times}
    bldfile:  text;  {file variable for construct/destruct times file}
    srchfile: text;  {file variable for search times file}

    i,j,runs,p: integer; {loop control variables}
                         {runs: runs  i: dim  j: dsets  p: passes}
```

```
function achr(k: integer): char;
{converts a digit to a character}

begin {achr}
  case k of
    0: achr:= '0';
    1: achr:= '1';
    2: achr:= '2';
    3: achr:= '3';
    4: achr:= '4';
    5: achr:= '5';
    6: achr:= '6';
    7: achr:= '7';
    8: achr:= '8';
    9: achr:= '9';
  end;
end;  {achr}

function bchr(k,c: integer): char;
{converts a two digit number to two characters}

begin {bchr}
  if c = 1 then     bchr:= achr(k div 10)
  else              bchr:= achr(k mod 10);
end;  {bchr}

procedure buildrds1(var head, tail, bottom, lastdim: nodeptr;
                    ndim, ptndx, run: integer);
{this is a driver procedure to test the insertion procedure. It builds a
rds1 structure by inserting one datapoint at a time}
var dpoint: crdptr;      {the datapoint we are about to insert}
    current: nodeptr;    {the new node we are about to insert}
    i,j: integer;        {loop control variables}
    dfile: text;         {the input datafile}
    et, st: integer;     {for timings}

begin {buildrds1}
  createemptyrds1(head, tail, bottom, lastdim);
  open(dfile,
dptnam+achr(ndim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+dptext, 'old');
  reset (dfile);

  st:= sysclock;
  for i:= 1 to numpoints do
    begin {get and insert next datapoint}
      new(dpoint);
      for j:= 1 to ndim do
        read(dfile, dpoint^[j]);
      readln(dfile);
      for j:= (ndim+1) to maxdim do
        dpoint^[j]:= 0;
      new(current);
        current^.datapoint:= dpoint;
        current^.minki:= current^.datapoint^[1];
        current^.maxki:= current^.datapoint^[1];
        current^.nextdim:= lastdim;
        current^.down:= bottom;
      insert_rds1(head, current, 1);
    end; {get and insert next datapoint}
  et:= sysclock;

  writeln (bldfile, et-st);
  close (dfile);
end;  {buildrds1}
```

```
procedure search(head: nodeptr; ndim, ptndx, run: integer);
{this procedure contains the driver code to test the search procedure,
kdrsearch, which is located in unit rdslsrch.}

var next: boolean;
    i,j: integer;     {loop control}
    st, et: integer; {for timings}
    lrfile: text;
    count: integer;

  procedure getlar(ndim: integer);
  {this procedure initializes l and r for our search}

  var j: integer;

  begin {getl&r}
    for j:= 1 to ndim do
      begin
        read(lrfile, l[j]);
        read(lrfile, r[j]);
      end;
    readln(lrfile);
    for j:= (ndim+1) to maxdim do
      begin
        l[j]:= 0;
        r[j]:= 0;
      end;
  end;  {getl&r}

begin {search}

  open(lrfile,
rqfnam+achr(ndim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+rqfext,'old');
  reset  (lrfile);
  rewrite (sfile);

  for i:= 1 to numper do
    begin
      getlar(ndim);

      {track progressin of range search}
      write(i:2, ' ');

      st:= sysclock;
        for j:= 1 to numsearch do
          begin
            next:= false;
            kdrsearch(head, head, next, 1, 1);
            {1 if we want to report points in sfile}
         end;

      et:= sysclock;
      writeln(srchfile, (et-st));
    end;

  close (sfile);
  close  (lrfile);

end;  {search}
```

```
procedure destroyrdsl(var head: nodeptr; ndim, ptndx, run: integer);
{this is a driver procedure for testing the delete procedure.  we
dispose of the rdsl built up by the driver procedure buildrdsl in the
unit rdslitst by deleting one datapoint at a time until we are left with
an empty rdsl.}

var dpoint: coordinate;
    i,j: integer;      {loop control variable}
    dfile: text;       {the input datafile}
    et, st: integer;   {for timings}

begin {destroyrdsl}

  open  (dfile,
dptnam+achr(ndim)+achr(run)+bchr(ptndx,1)+bchr(ptndx,2)+dptext,'old');
  reset (dfile);

  st:= sysclock;
  for i:= 1 to numpoints do
    begin {obtain and delete the next datapoint}
      for j:= 1 to ndim do
        read(dfile, dpoint[j]);
      readln(dfile);
      for j:= (ndim+1) to maxdim do
        dpoint[j]:= 0;
      delete_rdsl(head, dpoint, 1);
   end;  {obtain and delete the next datapoint}
  et:= sysclock;

  writeln (bldfile, et-st);
  close  (dfile);

end;  {destroyrdsl}

begin {main}

  {determine if we are to perform initial or further testing}
  writeln('initial (i) or further (f) testing?');
  readln (testpar):

  {get parameters that govern tests - support full and partial testing}
  writeln('how many passes per run?');
  readln (passes);
  writeln('please enter start and end runs');
  readln (ndss, numdsets);
  writeln('please enter start and end dimension');
  readln (ntds, ntestdim);
  writeln('please enter start multiplier');
  readln(sm);

  {initialize values needed by structure}
  new(amax);
  amax^[1]:= mx; amax^[2]:= mx; amax^[3]:= mx; amax^[4]:= mx;
  new(amin);
  amin^[1]:= mn; amin^[2]:= mn; amin^[3]:= mn; amin^[4]:= mn;
  new(sv);
  sv^[1]:= 0; sv^[2]:= 0; sv^[3]:= 0; sv^[4]:= 0;
```

```
    if testpar = i then
      begin
        ni[1]:= 9;  ni[2]:= 8;  ni[3]:= 5;  ni[4]:= 3;
        ns[1,1]:= 1; ns[1,2]:= 3; ns[1,3]:= 5; ns[1,4]:= 7; ns[1,5]:= 9;
        ns[1,6]:= 11;  ns[1,7]:= 13;  ns[1,8]:= 15;  ns[1,9]:= 17;
        ns[2,1]:= 1; ns[2,2]:= 3; ns[2,3]:= 5; ns[2,4]:= 7; ns[2,5]:= 9;
        ns[2,6]:= 11;  ns[2,7]:= 13;  ns[2,8]:= 15;
        ns[3,1]:= 1; ns[3,2]:= 3; ns[3,3]:= 5; ns[3,4]:= 7; ns[3,5]:= 9;
        ns[4,1]:=  1;  ns[4,2]:=  3;  ns[4,3]:=  5;
      end
    else
      begin
        ni[1]:= 3;  ni[2]:= 3;  ni[3]:= 3;  ni[4]:= 3;
        ns[1,1]:= 25; ns[1,2]:= 30; ns[1,3]:= 35;
        ns[2,1]:= 20; ns[2,2]:= 25; ns[2,3]:= 30;
        ns[3,1]:= 15; ns[3,2]:= 20; ns[3,3]:= 25;
        ns[4,1]:= 10; ns[4,2]:= 15; ns[4,3]:= 20;
      end

    writeln('enter name of datafile to hold construct/destruct times.');
    readln (bldname);
    writeln('please enter name of datafile to hold search times.');
    readln (srchname);

    {open datafile to hold construct/destruct times}
    open   (bldfile, bldname, 'new');
    rewrite(bldfile);
    {open datafile to hold search times}
    open (srchfile, srchname, 'new');
    rewrite(srchfile);

    {main test loop}
    for runs:= ndss to numdsets do
      for i:= ntds to ntestdim do
        for j:= sm to ni[i] do
         for p:= 1 to passes do {it's a multipass tester!}
          begin
            numpoints:= ns[i,j]*basenumpts; {# datapts in our data set}
            dimensions:= i;                 {num dimensions of tree}

            {output to track progression of testing}
            writeln ('ready. set. gone. dim: ', i:2, ' points: ',
                     numpoints);
            write('building ... ');
            buildrdsl(head,tail,bottom,lastdim, i,ns[i,j],runs);
                     {build the rdsl structure}
            writeln('built');
            write('searching ... ');
            search(head, i,ns[i,j],runs);  {search the structure}
            writeln('searched');
            write('annihilating ... ');
            destroyrdsl(head, i,ns[i,j],runs);
                     {dispose of the rdsl structure}
            writeln('it''s history!');
          end;

    {close our result datafiles}
    close  (srchfile);
    close  (bldfile);
  end.  {main}
```

VITA

Candidate's Full Name: Michael Gerard Lamoureux

Place and Date of Birth: Charlottetown, Prince Edward Island, Canada
July 1, 1975

Permanent Address: 24 Dawson Court
Charlottetown, Prince Edward Island, Canada
C1A 8T1

School Attended: Colonel Gray Senior High
Charlottetown, P.E.I., Canada, 1989 - 1991

University Attended: University of Prince Edward Island
Charlottetown, P.E.I., 1991 - 1994
B.Sc. Mathematics with Computer Science

Publications:

Lamoureux, Michael G., and Nickerson, Bradford G., "On the equivalence of B-trees and deterministic skip lists", University of New Brunswick Technical Report, TR96-102, January 1996.

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.

Lamoureux, Michael G., "An Implementation of a Multidimensional Dynamic Range Tree Based On An AVL Tree", University of New Brunswick Technical Report, TR95-100, November 1995.

Lamoureux, Michael G., and Nickerson, Bradford G., "Deterministic Skip List Data Structures: Efficient Alternatives to Balanced Search Trees", Proceedings of the 19th Annual Mathematics and Computing Science Days (APICS'95), Sydney, Cape Breton (Nova Scotia, Canada), Oct. 20-21, 1995.