DETERMINISTIC SKIP LISTS FOR K-DIMENSIONAL RANGE SEARCH

by

Michael G. Lamoureux
Bradford G. Nickerson

TR95-098, November 1995
Revision 1

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

Deterministic Skip Lists for k-dimensional Range Search

Faculty of Computer Science Technical Report TR95-098 Revision 1

Michael G. Lamoureux
Bradford G. Nickerson

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

E-mail: y320@unb.ca (Michael G. Lamoureux)
E-mail: bgn@unb.ca (Bradford G. Nickerson)

November 1995

Abstract

This report presents a new data structure for multidimensional data which is based on an extension of the deterministic skip list data structure *projected* into k-dimensions. The data structure is labeled the k-d Range DSL and it allows for efficient multidimensional range search on point data. This structure supports fast insertions, deletions, and search and the k-d Range DSL is shown to be optimal (i.e. $O(\lg^k n + t)$ to find t points in range) for k-d range search on n points in a dynamic environment with a pointer based memory model where we impose restrictions on worst-case search time, dynamic operations, and storage.

The structure is extendible and it does permit for multidimensional semi-infinite range queries in an equivalent time. The k-d Range DSL, by nature of the skip list structure, is well suited for range search in a parallel processing environment. Algorithms for building, insertion, deletion, and range search are provided for the data structure along with proofs of validity and worst-case complexity for these operations. Also, complete working code in (Borland's Turbo) Pascal is provided in the appendices.

Key words: dynamic data structures, algorithms, range search, orthogonal range query, multidimensional data, skip lists, searching, worst-case complexity

TABLE OF CONTENTS

1 Introduction

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

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

Although range search is commonly provided by relational algebra operations on index structures based on the B-Tree [Come79] and dynamic variants of the range tree of Bentley [Bent80a] due to Willard and Lueker ([Luek78], [Luek82], [Will85], and [Will85b]), these structures are either inefficient, difficult to implement, or both.

In this paper we use an extension of the skip list structure, as introduced by [Pugh90]), based on the deterministic skip list of [Munr92] *projected* into k-dimensions to provide a basic structure that allows for a more direct and more efficient range search capability while maintaining the conceptual simplicity and ease of implementation provided by the 1-d skip list. This work can be viewed as an extension of [Nick94] and, in fact, one can see that the Search Skip list structure is *embedded* in the Range DSL structure and use this fact to shave off a logarithmic factor from the complexity of certain multidimensional semi-infinite range queries.

Furthering the work of [Papa93], [Lamo95a] have shown the relationship of the DSL to the B-tree and we find that the DSL is equivalent (structurally and functionally) to an extension of the B-Tree known as the Bd^+-tree (which is similar in many respects to the B-link tree used by [John91]), providing a solid foundation for the use of skip lists as data (index) structures for multidimensional range queries.

The basic questions that are considered in this paper are: (following [Nick94])

1) Can skip lists be used for multidimensional data? If so, how?

2) What kind of time and space performance are achieved for range search when a skip list is used for multidimensional data?

3) What other operations can such structures support?

2 Range Search

Using the definition of [Knut73] we define a *range query* as a query that asks for records (in a File F containing n records) whose attributes fall within a specific range of values (e.g. height > 6'2" or \$23,000 <= annual income <= \$65,000). We will call these

limits L for low and H for high. Boolean combinations of range queries over different attributes are called *orthogonal range queries*. When the conjunction of range queries is required, we can view each separate attribute as one dimension of a k-dimensional space, and the orthogonal range query corresponds to asking for all records (points) that lie within a k-dimensional (hyper) rectangular box. A range search is performed to retrieve all records which satisfy the query. When specifying the limits of an orthogonal range query, we will use the limit vectors **L**, for lower limits, and **H**, for upper limits, and use the notation $\mathbf{L}_i$ to correspond to the ith lower limit (and, similarly, $\mathbf{H}_i$ corresponds to the ith upper limit).

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

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

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

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

Following the example of [Will85] we will use $U(n,k)$ to refer to the update cost function when the cost function for insertion, $I(n,k)$, is of the same order of complexity as the cost function for deletion, $D(n,k)$. (This is often the case when the insertion and deletion times are dependent upon the search time.) Note that we imply a Big-Oh analysis unless otherwise stated.

As indicated in the introduction, there are upper bounds that we want to place on the worst case search time, update operations, and storage. We must have storage $S(n,k) < O(n^2)$ in order to be able to feasibly store the structure for use. If range search time, $Q(n,k)$, is not less then $O(n)$ then there are no advantages to using such a structure. The same holds true for the time required by updates; if we don't have $U(n,k) < O(n)$ then there are no real advantages to using such a structure (unless range search was indeed optimal and storage minimal) as we would essentially have to rebuild the structure for every update. Note that, since we are working with a dynamic data structure, $U(n,k) < O(n)$ implies that we should have $P(n,k) < O(n^2)$.

Many approaches have been devised for the efficient handling of range searches. [Bent75] devised a multidimensional binary search tree, known as the k-d tree, to handle multidimensional point data. Assuming a balanced structure, it is known that the k-d tree has

$$P(n,k) = O(n \lg n),\ S(n,k) = O(kn), \text{ and } Q(n,k) = O(t + kn^{1-1/k}),$$

where t is the number of records found in the search.

The point quadtree can also be used for multidimensional range search. The reader is referred to [Same90] for a good overview of both the point quadtree and k-d tree algorithms. Under assumptions of relatively evenly spaced data, the point quadtree has $Q(n,k) = O(kn^{1-1/k})$.

The range tree of [Bent80a] consists of a base binary search tree with nodes discriminating on one of the k-dimensions. Each node in k-dimensional space contains a pointer to another range tree allowing a search in a (k-1)-dimensional space. This recursively defined structure has been shown to have

$P(n,k) = O(n \lg^{k-1} n)$, $S(n,k) = O(n \lg^{k-1} n)$, $Q(n,k) = O(t + \lg^{k} n)$

and, with the modifications of Lueker and Willard ([Luek78], [Luek82], [Will85, and [Will85b]),

$I(n,k) = D(n,k) = O(\lg^{k-1} n)$

using an amortized worst case analysis. Notice that Q(n,k) is substantially faster than for the k-d tree at the expense of requiring more space. It is advantageous to note that, using an amortized worst case analysis, this structure is *optimal* when we consider the overall balance of the structure as defined by the five cost functions (see [Lamo95b]).

Edelsbrunner ([Edel81]) introduced the RT-tree (also called the range priority tree by H. Samet [Same90]) which makes use of the range tree and the priority search tree of McCreight ([McCr85]) to arrive at the following costs:

$S(n,k) = P(n,k) = O(n \lg^{k-1} n)$, $Q(n,k) = O(t + \lg^{k-1} n)$, and $I(n,k) = D(n,k) = O(\lg^{k} n)$

which improves by one factor of lg n on the range search time. (Note that, due to the similarity of the RT-tree to the range tree, we can use the methods of Willard and Lueker to bring $I(n,k)$ and $D(n,k)$ down to $O(\lg^{k-1} n)$ if we desire.)

The structure that we will examine in this paper supports fast insertions, deletions, and efficient k-d range search on n points in a dynamic environment with a pointer based memory model where we impose restrictions on worst-case search time, dynamic operations, and storage. (For faster range queries using an array based model, see [Bent80b] and [Lamo95c] but note that these methods only work for static data sets and require significantly more storage. In this paper we assume a dynamic environment.)

3 Skip Lists

3.1 1-d Skip Lists

The probabilistic skip list was introduced by [Pugh90] as an alternative to balanced tree structures. Probabilistic skip lists support range search in one dimension, and it is known that the **expected** time for a simple query (i.e. search for only one record in the file) is O(lg n), and that the same time is required for insertion and deletion. Algorithm 1, given below, can be used for range search in one dimension.

```
1-d_Range_Search(S, L, H, A)
{Range search on skip list S to find all keys between L and H and place the resultant
records in A}

Begin

 A:= Empty set;
 determine node b with smallest key K in S ≥L;
 while K ≤ H do
  Begin
    add record at node b to A;
    b:= next node in S;
    K:= key of node b;
  End

End;
```

Algorithm 1. Range search in a 1-d skip list.

Due to the nature of a skip list having a complete set of pointers at level 0, the expected running time of this algorithm equals the time needed to find the first key K ≥ the low range key L in the skip list plus the time spent in the while loop to follow the level 0 pointers until the high range key H is exceeded. For the probabilistic skip list we have the **expected** worst case times of

$Q(n,1) = O(t + \lg n)$, and $I(n,1) = D(n,1) = O(\lg n)$.

However, the probabilistic skip list worst case time is still $O(n)$ for search, insert, and delete, although there is a very low probability (for large n) that the worst case will occur. With the probabilistic skip list in mind, [Munr92] have introduced the deterministic skip list (DSL) which is similar in structure to the probabilistic version while using no notions of probability at all. The deterministic version has a worst case search time of $\Theta(\lg n)$, along with similar costs for insertion and deletion. Their analysis also shows that the DSL space requirements are $O(n)$. We can use Algorithm 1 (above) for range search and find that the worst case costs are

$S(n,1) = O(n)$, $Q(n,1) = O(t + \lg n)$, and $I(n,1) = D(n,1) = O(\lg n)$

which are optimal in the 1-d case.

Although skip lists are often represented in the *standard* form of [Pugh90] as in Figure 1 (below), we will use the linked list representation introduced in [Munr92] and found in [Nick94], as shown in Figure 2. Note that pointer arrows (which always point down or to the right) are not shown in Figure 1 or 2 or any of the figures that follow.

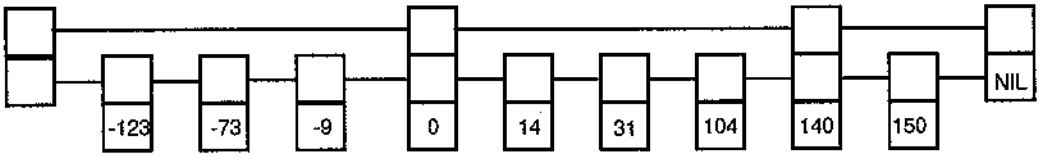


Figure 1. A skip list in *standard* form.

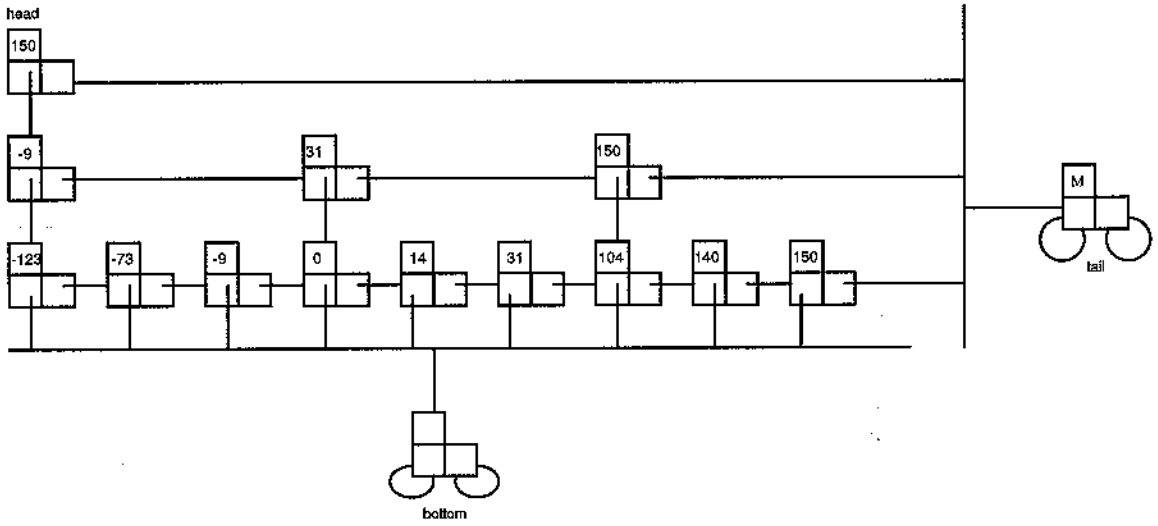


Figure 2. A linked list representation of the skip list of Figure 1.

Before we continue, a few extra definitions are in order. The down sub-tree of a node consists of all nodes that can be reached by traversing the down pointer of the current node such that those nodes are not reachable by traversing the down pointer of the node pointed to by the current node's right pointer. Thus, the nodes (-123, -73, and -9) at level 0 (leaf level) are in the down sub-tree of the node (-9) at level 1; and the node (0) at level 0 is not in the down sub-tree of the node (-9) at level 1 as it is in the down sub-tree of the node (31) at level 1 which is pointed to by the right pointer of node (-9) at level 1.

Also, the immediate down sub-tree of a node is all nodes at level (i-1) reachable by traversing the down pointer of the current node at level i. Thus, the nodes (-9, 31, and 150) at level 1 are in the immediate down sub-tree of the node (150) at level 2 (which, by definition, contains only the head node) and the nodes (-123, -73, and -9) at level 0 are in the immediate down sub-tree of the node (-9) at level 1.

The direct descendent in the immediate down sub-tree is that node which is the first node in a node's immediate down sub-tree. Note that we define gap size as the number of nodes in the immediate down sub-tree when we ignore the direct descendent.

3.2 The Range Tree

Before introducing the k-d Range DSL, we will briefly review the range tree of [Bent80a]. The range tree, like binary search trees, stores datapoints and is designed to detect all points that lie in a given range (i.e. it was designed with range queries in mind).

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

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

A range search for ($[L_x:H_x]$,$[L_y:H_y]$) is performed by procedure 2D_Search (below). It makes use of the procedure 1D_Search (not given). Procedure 2D_Search starts by searching T for the smallest key that is $\geq L_x$, say L_x', and the largest value that is $\leq H_x$, say H_x'. It then finds the common ancestor of L_x' and H_x', Q, that is closest to them and assigns $\{PL_i\}$ and $\{PH_i\}$ to be the sequences of nodes, excluding Q, that form the paths in T from Q to L_x' and H_x', respectively.

Let LEFT(P) and RIGHT(P) denote the left and right children, respectively, of non-leaf node P and RANGE_TREE(P) be the one-dimensional range tree for y that is stored at non-leaf node P. Then, for each P that is an element of $\{PL_i\}$ such that LEFT(P) is also in $\{PL_i\}$, 2D_SEARCH performs a one-dimensional range search for $[L_y:H_y]$ in the one-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 $\{PH_i\}$, 2D_SEARCH performs a one-dimensional range search for $[L_y:H_y]$ in the one-dimensional range tree for y rooted at node LEFT(P). 2D_SEARCH also checks to see if L_x' and H_x' are in the two dimensional range.

```
Procedure 2D_Search(Lx, Hx, Ly, Hy, T);

{Perform a range search for the two-dimensional range query ([Lx:Hx], [Ly:Hy]) in the two
dimensional range tree rooted at T}

Var PATH_TO_LXP, PATH_TO_HXP: set;
        LXP, HXP, P, Q: node;

Begin

 LXP:= closest node ≥ Lx;
 HXP:= closest node ≤ Hx;
 IF LXP in range then report LXP;
 IF HXP in range then report HXP;
 Q:= nearest common ancestor of LXP and HXP;
 PATH_TO_LXP:= {non-leaf nodes on the path from Q to LXP}
 PATH_TO_HXP:= {non-leaf nodes on the path from Q to HXP}
 For each P in PATH_TO_LXP Do
  Begin
   IF LEFT(P) ∈ PATH_TO_LXP Then
    1D_SEARCH(Ly, Hy, RANGE_TREE(RIGHT(P)));
  End;
 For each P in PATH_TO_HXP Then
  Begin
   IF RIGHT(P) ∈ PATH_TO_HXP Then
    1D_SEARCH(Ly, Hy, RANGE_TREE(LEFT(P)));
  End;

End;
```

Algorithm 2. 2-D range search in a 2-d range tree from [Same90].

The range tree can be extended to handle k-dimensional data. The extension is analogous to the extension of the 1-d case to the 2-d case. This recursively defined structure is known to have $P(n,k) = O(n \lg^k n)$, $S(n,k) = O(n \lg^{k-1} n)$, $Q(n,k) = (t + \lg^k n)$ and, with the modifications of Lueker and Willard ([Luek78], [Luek82], [Will85], and[Will85b]), dynamic updates may be performed with complexity $I(n,k) = D(n,k) = O(\lg^{k-1} n)$ where an amortized worst case analysis is used.

3.3 The k-d Range DSL

A new structure based on the linked-list representation of the 1-d skip list *projected* into k-dimensions is now introduced. It uses a generalization of the idea inherent in the range tree (which is based on the paradigm of multidimensional divide and conquer of [Bent80a]) as described above. The structure is based on the linked-list version of the 1-3 DSL of [Papa93] and is always balanced in the sense of B-Trees (in fact, it can be equivocated to a modification of the generalized Bd^+-tree for multi-attribute data, see [Lamo95a]); i.e. the leaves are all at the same depth of c lg *n* where $1/2 \leq c \leq 1$. It should be noted that the structure can be generalized 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 sub-tree) but for the present exposition we assume that a 1-3 DSL is used as the base structure.

The properties of this structure are (where k is the number of dimensions):

1. Each node contains a datapoint which is either
   (i) an actual datapoint if the node is a leaf node (K_1, K_2, ..., K_k)
   (ii) a pre-defined sentinel value (s.v.) if the node is not a leaf node
   (s.v. = (0,0, ..., 0) in the code and is not permitted as a valid datapoint)
2. Each node contains two values. These are
   (i) minki = minimum K_i value in the down sub-tree of the current node
   (ii) maxki = maximum K_i value in the down sub-tree 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 k-d Range DSL of the points in the down sub-tree ordered on the K_{i+1} coordinate values, analogous to the pointers in the non-leaf nodes of the Range Tree that point to a Range Tree ordered on the K_{i+1} coordinate values.
4. Four special nodes called head, tail, bottom, and lastdim are used to indicate the start and end of structure conditions. The nodes head, tail, and bottom are identical to the head, tail, and bottom nodes in the 1-d DSL and the node lastdim lets us know that we have reached the last dimension of the structure in our search. (Lastdim is also used to indicate that we have reached the leaf level when searching.) (The head node and lastdim nodes are defined to be dimension 0 and the first dimension of our structure is ordered on the K_1 coordinate values.)
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. Gap size is defined as the number of children a node has in it's immediate down sub-tree.
The basic 1-d structure is somewhat similar to a 2-3-4 tree [Papa93] and the resultant structure is similar to a 2-3-4 tree of 2-3-4 trees. Figure 3 illustrates the composition of a single node and Figure 4 illustrates a 2-d Range DSL for the data of Table 1.

Table 1. Example 2-d point set.

J	K_1	K_2
New York	-73	42
London	0	52
Naples	14	41
Vancouver	-123	48
Tokyo	140	37
Sydney	150	-34
Durban	31	-30
Lisbon	-9	39
Singapore	104	2

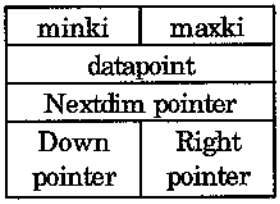


Figure 3. Composition of a single node in the 2-d Range DSL.

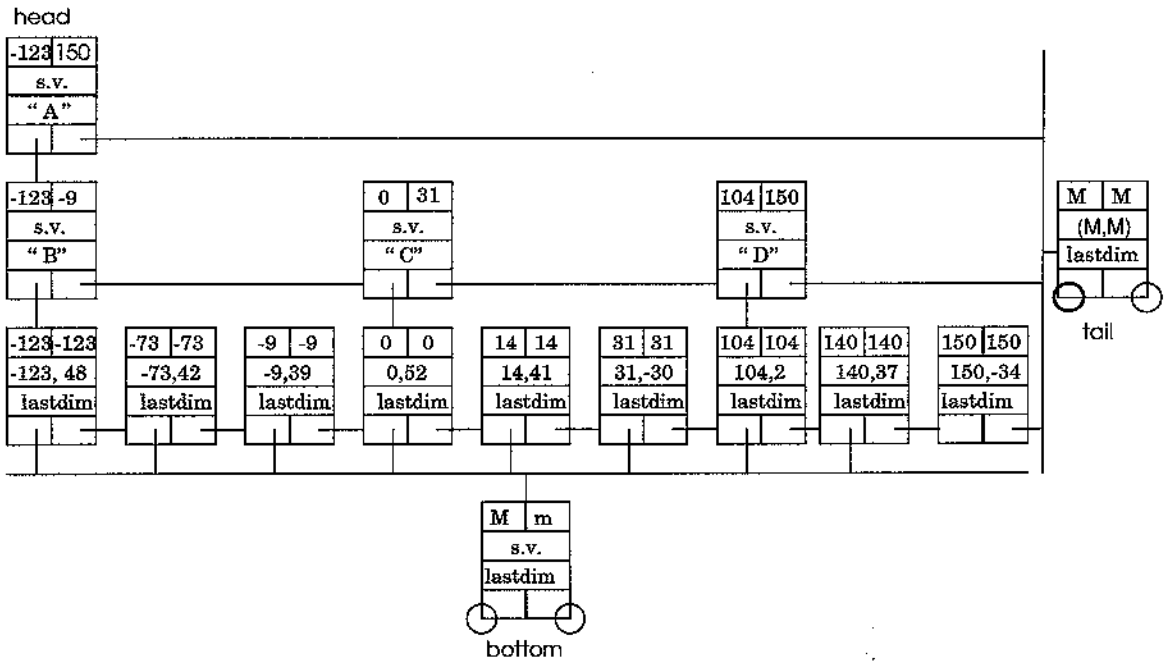


Figure 4a). The DSL structure in dimension 1 (ordered on K_1) for the 2-d Range DSL of the data in Table 1.

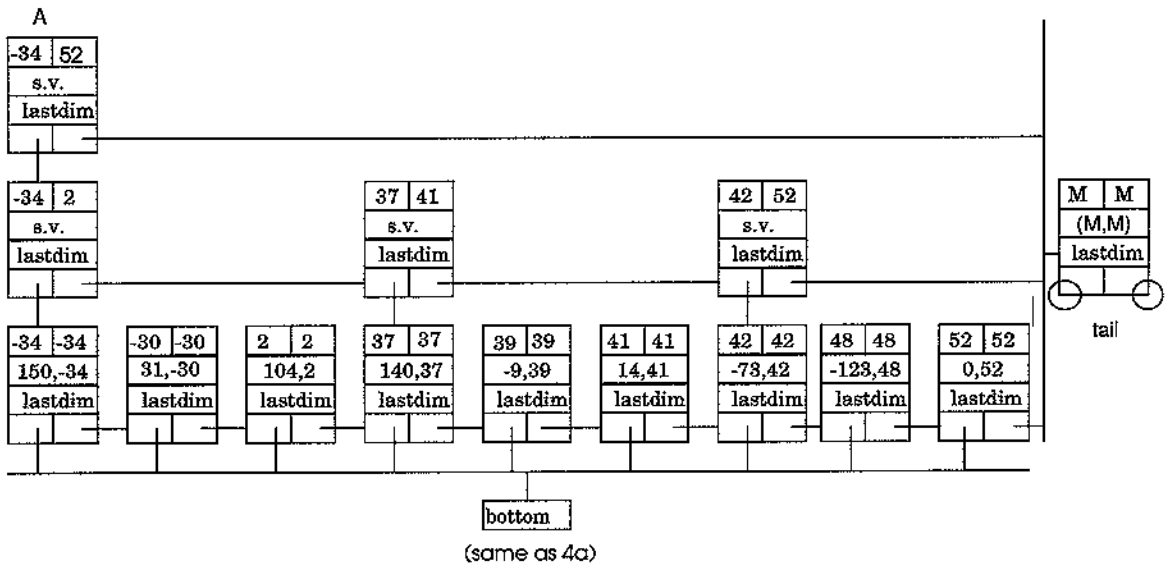


Figure 4b) The DSL Structure "A" in dimension 2 (ordered on K_2) for the 2-d Range DSL of the data in Table 1.

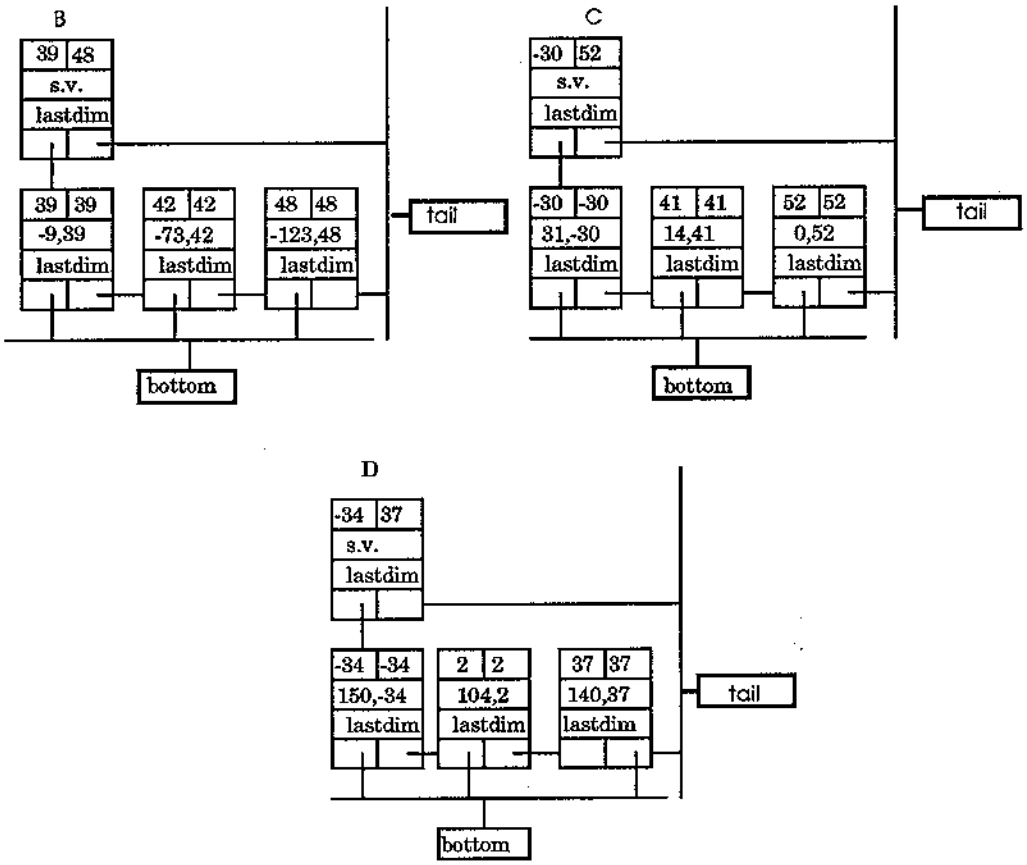


Figure 4c). The DSL structures "B", "C", & "D" (ordered on K_2) for the 2-d Range DSL of the data of Table 1. (The bottom and tail nodes have the same structure as Figure 4b).

The k-d Range DSL, like the range tree, is an *optimally balanced* structure using an amortized worst case analysis (it has an optimal balance cost, see [Lamo95b]). However, unlike the range tree, we find that the k-d Range DSL has an advantage that is not present in the range tree of [Bent80a] or in the modifications by Willard and Lueker ([Luek78], [Luek82], [Will85], and [Will85b]). We will soon see that, unlike the range tree, a dynamic implementation is both simple to conceptualize and to implement. (For an exposition on the structure and implementation of a k-d dynamic range tree, the reader is referred to [Lamo95d].) Also, rebuilding of a substructure in the next dimension of a k-d Range DSL can be done in $O(n \lg^{k-2} n)$ time, paralleling the rebuilding time of a substructure in the range tree. (We need to do rebuilding of a substructure in a k-d Range DSL when an insertion causes a split or when a deletion causes a merge. In a range tree, we need to do rebuilding of a substructure when the tree structure becomes unbalanced and we have to do a rotation to restore the balance needed for optimal k-d range search.)

We find that, like the range tree and its variations, the work involved in a single insertion or deletion is dependent not only on the number of datapoints in the structure, but also on a sequence of previous insertions and deletions (see [Fred81], [Luek78], and

[Will85b]). It is well known that any sequence of n update operations must take at least $O(n \lg n)$ time in 1-d and, from [Fred81], we know that any sequence of n update operations in k-d must take at least $O(n \lg^k n)$. This tells us that, using an amortized worst case analysis, an update operation is at least $O(\lg n)$ in 1-d and at least $O(\lg^k n)$ in k-d and this implies that, since the k-d Range DSL has $U(n,k) = O(\lg^k n)$ (see sections 3.3.3 and 3.3.4), again using an amortized worst case analysis, our structure has an optimal balance cost.

3.3.1 Building a k-d Range DSL

Constructing a k-d Range DSL requires inserting each point into the structure sequentially, starting with an empty Range DSL. Algorithm 3 can be used for this.

```
Procedure CreateEmptyRDSL;

Begin {CreateEmptyRDSL}
   new(head);      new(tail);
   new(bottom);    new(lastdim);
   head^.minki:= maxkey;                                  head^.maxki:= minkey;
   head^.datapoint:= sv;                                  head^.nextdim:= lastdim;
   head^.down:= bottom;                                   head^.right:= tail;
   tail^.minki:= maxkey;                                  tail^.maxki:= maxkey;
   tail^.datapoint:= (maxkey, maxkey, ..., maxkey);       tail^.nextdim:= lastdim;
   tail^.down:= tail;                                     tail^.right:= tail;
   bottom^.minki:= maxkey;                                bottom^.maxki:= minkey;
   bottom^.datapoint:= sv;                                bottom^.nextdim:= lastdim;
   bottom^.down:= bottom;                                 bottom^.right:= bottom;
   lastdim^.minki:= maxkey;                               lastdim^.maxki:= minkey;
   lastdim^.datapoint:= sv;                               lastdim^.nextdim:= lastdim;
   lastdim^.down:= lastdim;                               lastdim^.right:= lastdim;
 End;  {CreateEmptyRDSL}
```

Algorithm 3. Construction of a k-d Range DSL.

```
Procedure Construct_RDSL;

Const     maxkey = 2147483647;  {maxkey = (2^31) - 1}
          minkey = -2147483647;  {minkey = (2^31) + 1}     {assume 32 bit integers}

Type coordinates =     record  {stores a datapoint}
                         K1, K2, ..., Kn: integer;
                       end;
        nodeptr = ^node;
        node = record
                 minki, maxki: integer;
                 nextdim:  nodeptr;
                 datapoint: coordinates;
                 down, right: nodeptr;
                 end;

Const sv: coordinates = (0,0, ..., 0);

Var     head, tail, bottom, lastdim: nodeptr;  {sentinel nodes of the RDSL}
        dpoints:  array[1..numpoints] of coordinates;  {The datapoints we wish to insert}
        i:  integer;  {Loop variable}
        numpoints: integer;  {How many datapoints do we have?}
Begin  {Construct_RDSL}

 CreateEmptyRDSL;  {Create an empty Range DSL}
 GetDatapoints(dpoints, numpoints);  {Get the datapoints to be inserted into the DSL}
 For i:= 1 to numpoints Do  {For each of the datapoints Do}
  IF Search_RDSL(head, dpoints[i]) = bottom Then {If the datapoint isn't in the DSL}
   Insert_RDSL(head, dpoints[i]);                    {Insert it using the insertion alg.}

End;  {Construct_RDSL}
```

Algorithm 3. Construction of a k-d Range DSL (continued).

The procedure CreateEmptyDSL creates an empty k-d Range DSL which is then used by Procedure Construct_RDSL to create a k-d Range DSL of the datapoints, stored in the array dpoints, by way of the Insert_RDSL algorithm which we will see shortly. The Procedure Search_RDSL searches a k-d Range DSL for a given datapoint and returns a pointer to bottom if the point is not found in the k-d Range DSL; otherwise a pointer to the node containing the datapoint is returned and it is not inserted again by procedure Construct_RDSL. As the structure is constructed by inserting one point in a time, we shall discuss the space and time requirements for building a k-d Range DSL in the section on insertion.

3.3.2 Searching a k-d Range DSL

Range searching, though fairly simple, is in many ways different than that of the range tree for a number of reasons. First of all, when we know that we must search the down sub-tree we don't always know which node in the immediate down sub-tree is the one at which we are to continue our search. (Note that this does not affect the overall complexity of our search algorithm as we have to check at most 4 (1-3 DSL is the underlying structure) nodes to determine which one to continue our search at.) Secondly, as we are storing ranges in our nodes and not datapoints, the comparison operations are more involved as we must determine whether or not the range of the node overlaps the search range in that dimension; i.e. we must compare intervals. Algorithm 4 can be used to perform a range search in a k-d Range DSL. We will give the algorithm, verify it's validity, and then perform a worst-case analysis of the algorithm. Note that we can also use Algorithm 4 for a member query where we have the special case of our search interval being a datapoint.

The following procedure performs a range search on the k-d Range DSL rooted at current where the query ranges are given in the global variables **L** and **H**.

Take note that:
current is set to head on the first call
prev points to the previous node in our search
(and the current node if we are just beginning our search).
next is set to false on the first invocation of the procedure
(it is used to find the node to continue our search at in the immediate down sub-tree).
dim is the dimension we are currently searching (it corresponds to i in section 3.3).

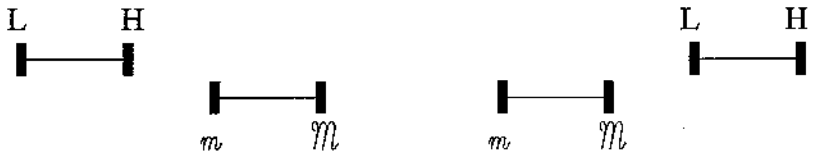
```
Procedure Search_RDSL(current: nodeptr; prev: nodeptr; next: boolean; dim: integer);
Var snode: nodeptr;  {the node after the last node in the immediate down sub-tree}
Begin  {Search_RDSL}
   IF current = prev Then  snode:= current^.right
   ELSE snode:= prev^.right^.down;
   IF next Then
     Begin While L[dim] > current^.maxki Do current:= current^.right;  next:= false;  End;
   With current^ Do
     IF NotLeafLevel(current) Then
      IF not ((L[dim] > maxki) or (H[dim] < minki)) Then
        IF ((L[dim] <= minki) and (maxki <= H[dim])) Then
          Begin
2)          IF (current^.nextdim <> lastdim) Then
              Search_RDSL(current^.nextdim, current^.nextdim, next, (dim+1))
            Else
              ReportAllPoints(current) {All points in sub-DSL are in range: report them}
            IF current^.right <> snode Then
                Search_RDSL(current^.right, prev, next, dim);
          End
        Else
            IF (H[dim] <= maxki) Then
              IF (L[dim] <= minki) Then
3)              Search_RDSL(current^.down, current, next, dim)
              Else
4)              Search_RDSL(current^.down, current, next:= true, dim)
            Else  {H[dim] > maxki and minki <= L[dim] <= maxki}
              Begin
5)              Search_RDSL(current^.down, current, next:= true, dim);
                IF current^.right <> snode Then
                  Search_RDSL(current^.right, prev, next, dim);
              End
      Else
1)      Return  {No points are in range in the down subtree of the current node}
    Else
      Begin  {we are at leaf level}
        While ((current <> snode) and (H[dim] >= current^.maxki)) Do
          Begin
            IF InRange(current) Then Report(current^.datapoint);
            current:= current^.right;
          End;
      End;  {we are at leaf level}
End;  {Search_RDSL}
```

Algorithm 4 Range Search in a k-d Range DSL.

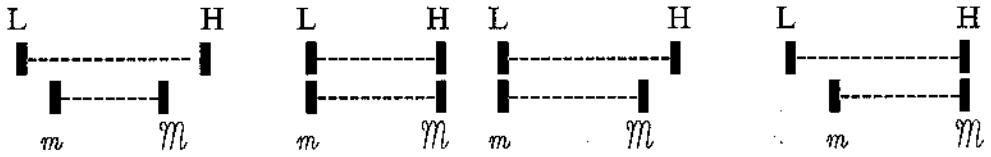
We will now show the validity of the search algorithm.

To show that the algorithm is correct, we must consider all of the possible cases where the search region overlaps the range of our dataset and show that the search algorithm not only accounts for all of them but handles them correctly. There are 13 possible overlap cases to consider and these are grouped into categories 1, 2, 3, 4, and 5 to simplify our proof (see Figure 5). We define minki as m and maxki as $\mathcal{M}$.

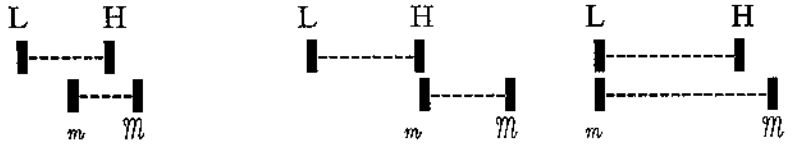
1) No overlap



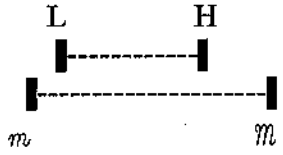
2) The range of the DSL is contained within the search range



3) Partial overlap where the search range includes m but not M



4) Partial overlap where m and M are not in the search range



5) Partial overlap where m is not in the search range and M is

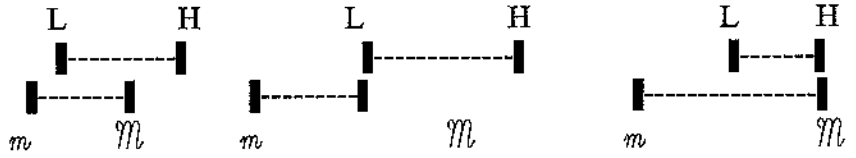


Figure 5: Overlap cases that must be considered to validate our search algorithm

The overlap cases are equivalent to the following inequalities:

(we assume that $L < H$ and $m < \mathcal{M}$)

1) $H < m$ or $\mathcal{M} < L$

2) $L < m$ & $\mathcal{M} < H$ or $L = m$ & $\mathcal{M} = H$,
$L = m$ & $\mathcal{M} < H$ or $L < m$ & $\mathcal{M} = H$

3) $L < m$ & $H < \mathcal{M}$ or $m = H$ or $L = m$ & $H < \mathcal{M}$

4) $m < L$ & $H < \mathcal{M}$

5) $m < L$ & $\mathcal{M} < H$ or $\mathcal{M} = L$ or $m < L$ and $H = \mathcal{M}$

Which are succinctly expressed as:

1) $L > \mathcal{M}$ or $H < m$

2) $L <= m$ and $\mathcal{M} <= H$

3) $H < \mathcal{M}$ and $L <= m <= H$

4) $H < \mathcal{M}$ and $L > m$

5) $L <= \mathcal{M} <= H$ and $m < L$

These are all accounted for in the search algorithm. We now show that they are handled correctly and in the right order. First we must determine whether or not there is any overlap. If there is none, as in case 1, then we terminate our search (as in the algorithm). If there is, we must then determine whether we search in this dimension or the next dimension. If the entire down sub-tree of the node is in range (case 2), then we should go to the next dimension, and we do. Otherwise we must proceed further in this dimension, pruning off any points not in range with respect to the current dimension before heading to the next dimension (as in the range tree).

If we proceed further, we must determine if we search the down sub-tree, the right sub-tree, or both. The fact that we did not terminate our search implies that there is some overlap. The fact that there is some overlap implies $L <= \mathcal{M}$ which implies that we must search the down sub-tree, and we do in all remaining cases. We need only search the right sub-tree when $H > \mathcal{M}$ (case 5), and this is the only case where we go to the right sub-tree. We must also insure that when we proceed to the down sub-tree that we continue our

search at a valid node. The while loop, which is invoked when we go to a down sub-tree where $L > m$, insures that we do continue our search at a valid node. The search algorithm is thus valid as we account for all cases of overlap and handle such appropriately.

We find that we only report data points when we reach the leaf level (and we insure we never leave the immediate down sub-tree by never going to the node snode, which is the first node in the down sub-tree of the right sibling of prev) and we check for the case that all points are in range in the last dimension, which is the only case where we discover the need to report points in range when not at the leaf level. In this case, we immediately proceed to the leaf level for reporting.

Theorem 1: The time, $Q(n,k)$, required to perform a range search on a k-d Range DSL is $O(\lg^k n + t)$.

Proof:

We will first show the 1-d case and then extend it to higher dimensions. From [Nick94] we know that the path length from the head node to the bottom level is $O(\lg n)$. Since range searching in a DSL is equivalent to range searching in a range tree; we find the first node in range and follow the links until we find a node out of range, it is easy to see that in the 1-d case range search is $O(\lg n + t)$.

To show that range search is $O(\lg^2 n + t)$ in the 2-d case we must determine what the worst case is. The worst case is when we have the worst case DSL (see Figure 6) in the first dimension (all gaps are of size one and the height of the DSL is $(\lg n + 1)$). In this instance, we have $2^{\lg h - \ell}$ nodes per level, where we take the head node as level $\ell = h$ (and define $\lg 0 = 0$), and our structure is very similar to the range tree. We can see that (as we have h+1 levels) the worst case search will imply that we search a k-d Range DSL in the next dimension for a node at every level from $\ell = h$ downto 1 (see Figure 6). We thus have to search a total of $(\lg n - 1)$ k-d Range DSLs in the next dimension, at a search time of at most $(\lg n)$ each, so our overall search time is at most $O(\lg^2 n + t)$, as desired.

The proof that $Q(n,k) = O(\lg^k n + t)$ is quite similar to the 2-d case for a given value of k and we will now give an inductive proof that $Q(n,k) = O(\lg^k n + t)$. We have our base case of $k = 1$ and we have shown that $Q(n,1) = O(\lg n)$. Assume that $Q(n,k) = O(\lg^k n + t)$. We must show that $Q(n,k+1) = O(\lg^{k+1} n + t)$ to complete our proof.

Although there may be a large number of structures in the kth dimension (up to $2^x \sum_{x=0}^{\lg n - 1} (x + k - 2)C(k - 2)$ structures, as we shall find out in Lemma 1), we are able to see that we need only search at most $O(\lg^{k-1} n)$ of them. This follows from the correspondence between the worst case k-d Range DSL and the range tree of Bentley (and from the fact that $Q(n,k) = O(\lg^k n + t)$). In the worst case (which corresponds to a cyclic repetition of overlap cases 3 and 2 as we move down the structure), we need to search one (1) structure in the next dimension at each of the levels of the k-d RDSLs that we are currently searching except for the head level and the leaf level. For instance, in Figure 6, we search nodes **e** and **f** in the next dimension and then conclude our search at the node **g**

which (being at the leaf level) contains a single datapoint. (If we search in the next dimension at the head level, then we don't move down the structure and there is no associated structure in the next dimension at the leaf level.) As there are ($\lg n + 1$) levels in the worst case k-d RDSL, in the worst case we must search structures at ($\lg n - 1$) levels in the next dimension in each of the k-d RDSLs that we are currently searching in dimension k. Thus, for each structure, we may need to search up to $O(\lg n)$ structures in dimension (k+1) at a search time of at most $O(\lg n)$ each. By induction we are searching $O(\lg^{k-1} n)$ structures in dimension k. This gives us an overall search time of $O(\lg^{k+1} n + t)$ as desired (we must search $O(\lg n)$ structures in dimension (k + 1) for each of the $O(\lg^{k-1} n)$ structures in dimension k at a search time of $O(\lg n)$) and we have completed our inductive proof. ■

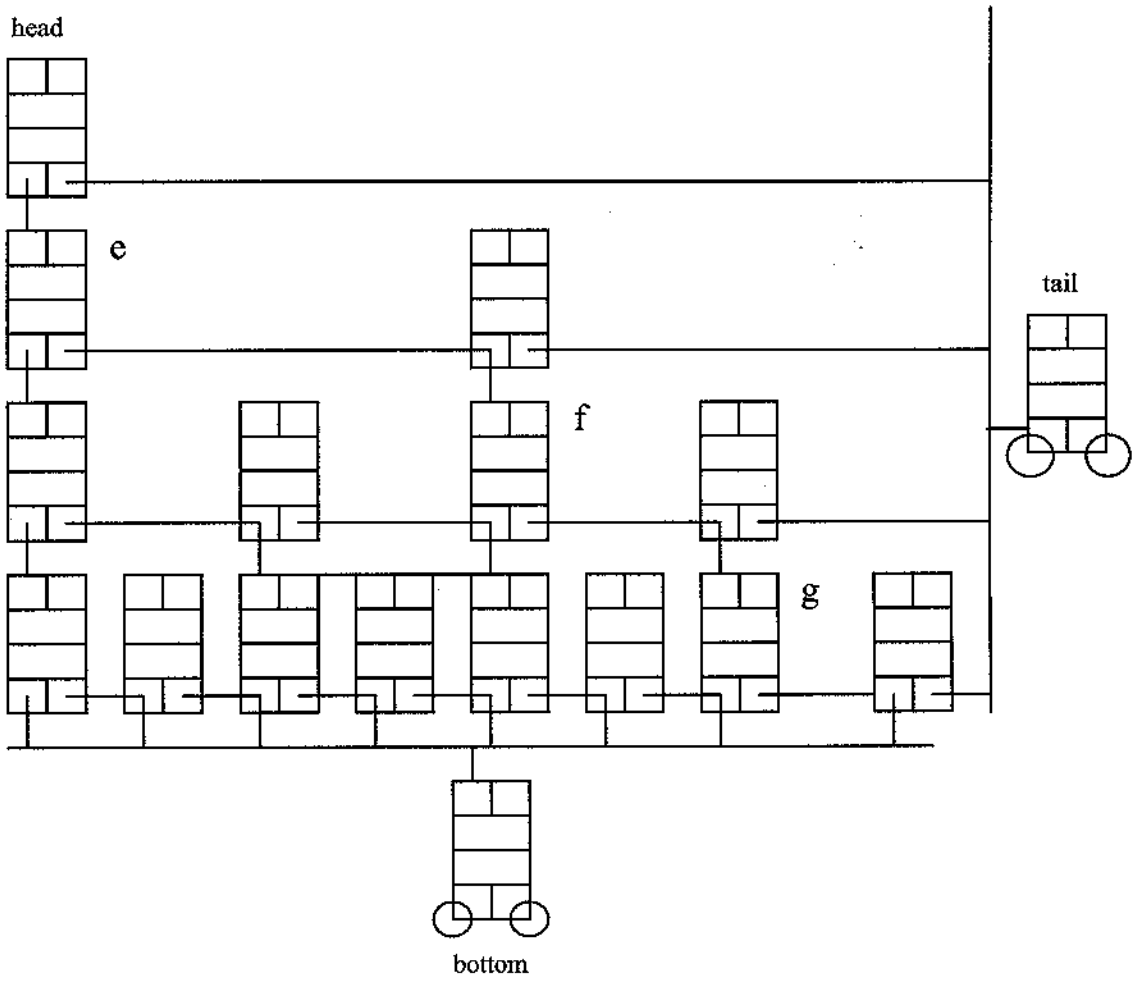


Figure 6. A worst-case DSL structure.

3.3.3.1 Simple Insertion in a k-d Range DSL

Insertion proceeds similar to that of the 1-d 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 DSLs in the next dimension. Note that the algorithm relies on the global variables head, tail, bottom, and lastdim as previously defined and the concepts of immediate down sub-trees and gap size. In addition, we assume (for simplicity) that all of the K_i coordinate values are unique and that the element isn't already in the structure. These assumptions are valid ones as 1) we can perturb datapoints (and use real numbers) if necessary and 2) we can insure an element isn't already in the structure by performing a search (and then use bottom up insertion to insert the new element if we keep track of the path or alter the structure to contain left and up pointers).

The (top down) insertion algorithm, in summary, is as follows:
(Let D represent the node being inserted.)

1. Start at the head node of the RDSL. (We use RDSL to abbreviate k-d Range DSL)
2. If the gap we are going to drop into is of size 1 or 2, determine where to drop, and if the node D is to be dropped into the first location (last location) of the immediate down sub-tree then alter the minki (maxki) value of the dropping node to be the K_i value of the new node D.
3. If the gap is of size 3, we first raise the middle element of the gap (where element is as defined in [Nick94]) by allocating a new node C to create two gaps of size 1, and then drop. The new node C created in the middle determines its minki and maxki value from the nodes in its immediate down sub-tree (called direct descendants by [Nick94]) and, if the node D is to be dropped in the down subtree of this newly created node C, from this node (D) as well. If raising an element implies that we have two elements at level h, then we must increase the height of the skip list by first moving the header node to height h+1 and then creating two new nodes at height h, thereby adding 1 to our height.
4. If we are not in the last dimension, we must insert the new node (D) into the RDSL rooted at the nextdim field of the last node visited at each level.
5. When we reach the bottom level, we insert a new element of height 1. This new element has a minki and maxki value = K_i.

This algorithm allows only gaps of sizes 1 and 2 on the path being traced down to the leaf level and therefore the resulting skip list with a newly inserted element is indeed a valid 1-3 skip list (i.e. it has gap sizes of only 1, 2, or 3). In addition, since the minki and maxki values are always determined with respect to the new point's K_i value, the minki and maxki values for a node will always be correct.

As an example of insertion, Figure 7 shows the 2-d Range DSL of Figure 4 after inserting the following three points into the structure:

J	K_1	K_2
Rio de Janeiro	-43	-23
Maracaibo	-72	11
Halifax	-65	44

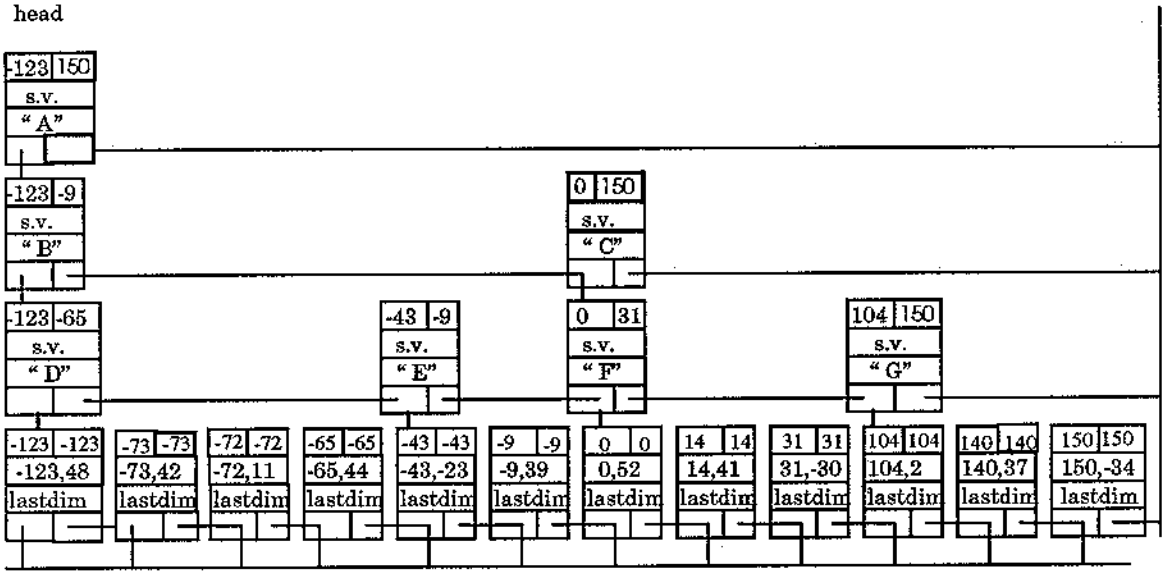


Figure 7a) DSL structure in dimension 1 (ordered on K_1 coordinates) of the 2-d Range DSL after inserting Rio de Janeiro, Maracaibo, and Halifax. The tail node and bottom node have been omitted to save space.

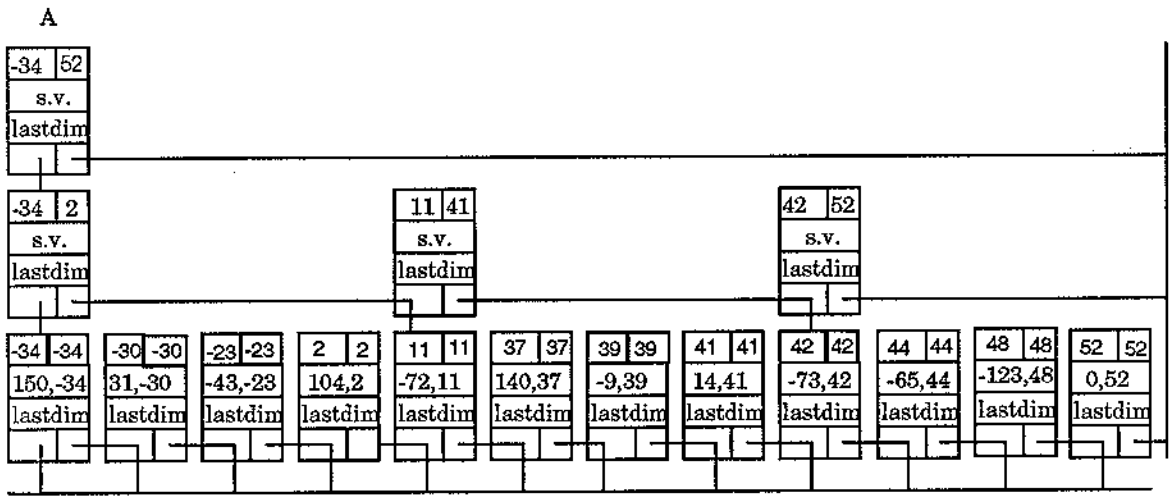


Figure 7b) Structure "A" (ordered on K_2) in dimension 2 for the 2-d Range DSL after inserting Rio de Janeiro, Maracaibo, and Halifax. The tail node and bottom node have been omitted to save space.

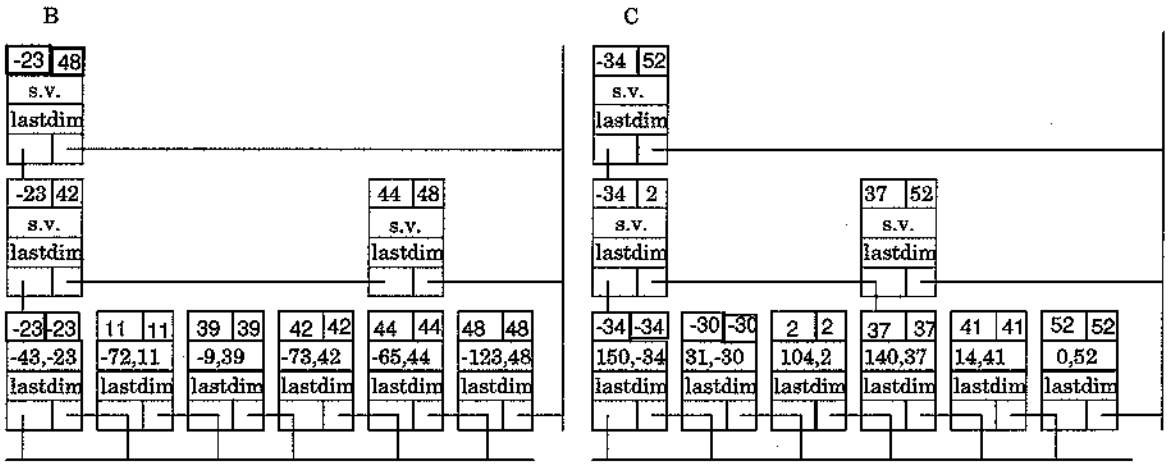


Figure 7c) Structures "B" and "C" (ordered on K_2) in dimension 2 for the 2-d Range DSL after inserting Rio de Janeiro, Maracaibo, and Halifax. The tail nodes and bottom nodes have been omitted to save space.

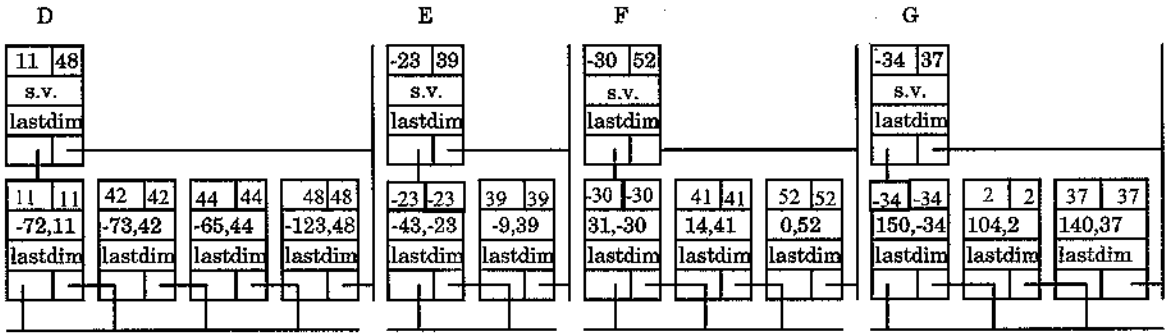


Figure 7d) Structures "D", "E", "F", and "G" (ordered on K_2) in dimension 2 for the 2-d Range DSL after inserting Rio de Janeiro, Maracaibo, and Halifax. The tail nodes and bottom nodes have been omitted to save space.

We will define a simple insertion as an insertion where we don't have to rebuild a k-d Range DSL in the next dimension (i.e. an insertion where we don't do a split (raise an element a level)).

Theorem 2.1: The time $I_S(n,k)$ required for a simple insertion in a k-d Range DSL is $O(\lg^k n)$.

Proof:

The proof depends on the observation that the time required for insertion of a node into a 1-d Range DSL depends only 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 pointers and values must be changed). It is true that we may have to change the minki or maxki value of each node visited, but this operation takes constant time as well and the number of nodes visited is accounted for

by the search time to find the new location. Thus, insertion of a new point into a 1-d Range DSL takes O(lg n) time.

The only difference in inserting a point into a 2-d Range DSL as compared to inserting a point into a 1-d Range DSL is that we must also insert the new point into the 1-d Range DSLs pointed to by the nextdim pointer in the last node visited at each level on the way to the bottom of the Range DSL in the first dimension. Thus we have to insert the new point into an additional O(lg n) structures which each have an insertion time of O(lg n) giving as an insertion time of $O(\lg^2 n)$.

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

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

Proof:

This is easy to see for k = 1. Assuming a worst case DSL, we have the maximum (lg n + 1) levels which gives us the maximum of $2n$ -1 nodes which tells us there are O(n) nodes in the structure, as desired.

In the 2-d case, assuming a worst case RDSL, in dimension 2 we have 1 structure of n points, 2 structures of $n/2$ points, 4 structures of $n/4$ points, 8 structures of $n/8$ points, ... x structures of $n/2^x$ points, ..., and $n/2$ structures of 2 points in dimension 2. This tells us that, for each level of the RDSL in dimension 1, we have n points or O(n) nodes in dimension 2. Thus, we have O(n lg n) nodes in dimension 2 as desired.

The k-d case turns out to be more difficult, but upon investigation (see Appendix 2) we see that we have ${}_{(x+k-2)}C_{(k-2)} \times 2^x$ structures of $n/2^x$ points or

$$n \sum_{x=0}^{\lg n - 1} (x + k - 2) C (k - 2) \qquad (1)$$

points in dimension k.

As we are interested in a Big-Oh analysis, we will perform an asymptotic approximation on Eqn. (1) to obtain our desired result. We can do this because we have at most twice as many nodes as we have points and constants are ignored in a Big- Oh analysis. We are dealing with a summation from 0 to lg n-1 so we can approximate the order of the summation by finding the order of the largest term and multiplying it by lg n.

Each term of the summation is of the form $\frac{(x+k-2)!}{(k-2)!x!}$ which tells us the largest term is of the form $\frac{(\lg n + k - 3)!}{(k-2)!(\lg n - 1)!}$. This can be asymptotically approximated by ([Grah94]):

$$\frac{\alpha}{\beta\gamma} \qquad (2)$$

where $\alpha = (\lg n + k - 3)!$
$\beta = (k-2)!$
$\gamma = (\lg n - 1)!$

Then

$$\alpha = \sqrt{2\pi(\lg n + k - 3)}\frac{(\lg n+k-3)^{\lg n+k-3}}{e^{\lg n+k-3}}(1 + \frac{1}{12(\lg n + k - 3)} + \frac{1}{288(\lg n+k-3)^2} + O(\frac{1}{(\lg n+k-3)^3}))$$

$$\beta = \sqrt{2\pi(k-2)}\frac{(k-2)^{k-2}}{e^{k-2}}(1 + \frac{1}{12(k-2)} + O(\frac{1}{(k-2)^2}))$$

$$\gamma = \sqrt{2\pi(\lg n - 1)}\frac{(\lg n-1)^{\lg n-1}}{e^{\lg n-1}}(1 + \frac{1}{12(\lg n - 1)} + O(\frac{1}{(\lg n-1)^2}))$$

and, since we assume $k << n$ (and therefore $k < \lg n$), (2) can be approximated by

$$\frac{\sqrt{2\pi \lg n}\frac{\lg n^{\lg n+k-3}}{e^{\lg n+k-3}}(1 + O(1))}{\sqrt{2\pi k}\frac{k^{k-2}}{e^{k-2}}(1 + O(1))\sqrt{2\pi \lg n}\frac{\lg n^{\lg n-1}}{e^{\lg n-1}}(1 + O(1))} = O(\lg n)^{k-2}.$$

Thus, we see that we can approximate the summation with $O(\lg^{k-1} n)$, since the largest term in the summation is of $O(\lg^{k-2} n)$ and there are $O(\lg n)$ terms, which gives us $O(n \lg^{k-1} n)$ nodes in dimension k, as we set out to prove. ■

Theorem 3: The space $S(n,k)$ required for storage of a k-d Range DSL is $O(n \lg^{k-1} n)$.

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 structure to determine the space required. To determine the maximum number of nodes we look at the worst-case DSL structure as presented in Figure 6. We see that, in the worst case, a 1-d RDSL structure contains $2n$ - 1 nodes and thus requires $O(n)$ space.

When we move from the 1-d Range DSL to the 2-d Range DSL we see that the worst case DSL structure allows for lg n levels, where each level points to Range DSLs in the next dimension that require a total of $O(n)$ space as the Range DSL is designed such that, like the range tree, the nodes at each level divide the data set into a number of

disjoint subsets. Thus the 2-d Range DSL requires $O(\lg n) \times O(n)$ or $O(n \lg n)$ space for storage.

The proof that $S(n,k) = O(n \lg^{k-1} n)$ follows from Lemma 1. We know that in dimension k there are at most $O(n \lg^{k-1} n)$ nodes. This tells us, since we are using a big-oh analysis, that our structure requires at most $O(n \lg^{k-1} n)$ storage space as the storage for the last dimension is the dominant term in our expression for storage (which is $\sum_{i=1}^{k} n \lg^{i-1} n$) as

$n \lg^{k-1} n >> n \lg^{k-2} n >> \bullet\bullet\bullet >> n$. ■

Theorem 4.1: The time, $P_D(n,k)$, required to dynamically build a k-d Range DSL is of $O(n \lg^k n)$.

Proof:

As a k-d Range DSL is constructed dynamically by inserting one point at a time into the structure, we easily see that the construction time is $O(n \lg^k n)$ as each insertion is bounded by $\lg^k n$ and there are n of them. ■

Theorem 4.2: The time $P_S(n,k)$ required to construct a k-d Range DSL from a sorted static data set is $O(n \lg^{k-1} n)$.

Proof:

A 1-d Range DSL contains at most $2n$ - 1 nodes (see Figure 6). Like a static range tree, a 1-d Range DSL can be built from the bottom up with a static data set. With a sorted data set it takes $O(n)$ time to build the leaf level as we need only connect each node to it's right sibling and the bottom node (which requires $2n$ pointer changes of O(1) time each). Once the leaf level is built, it takes at most $O(n)$ time to complete the structure This is because, assume we are building a worst case 1-d Range DSL (see Figure 6), we have to place at most $(n-1)$ nodes above the leaf level and the placement of a node takes only O(1) time as we know exactly where each node must go once the leaf level is built and placing a node requires changing only two pointers to connect it to it's right sibling and it's direct descendent in it's immediate down sub-tree. Thus it takes at most $O(n)$ time to build a 1-d Range DSL.

Moving from a 1-d Range DSL to a 2-d Range DSL seems to complicate our analysis and increase our rebuilding time as we must also build structures in the next dimension at each non-leaf node in the first dimension and we only have a sorted list for the first dimension, but we will soon see that we can maintain our $O(n \lg^{k-1} n)$ construction time by rebuilding the structures in the next dimension at the lower levels of the current structure first and working our way up the structure.

Let our structure be of height h (then the head node is at level h). We start by building the structures in the next dimension at level 1. There are $n/2$ structures and each structure contains only 2 datapoints in it's down sub-tree. Each of these structures can be built in O(1) time, as we can sort a list of 2 points with only 1 comparison and a 1-d Range DSL of 2 points has only 2 levels and therefore only 2+1 = 3 nodes. Therefore, we can see that it takes only $O(n)$ time to build the structures in the next dimension at level (1) as we are only building $n/2$ structures with a construction time of O(1) each.

It is a valid assumption to consider only the worst case RDSL. The best case structure which has the minimum number of levels (see Figure 8) has $n/4$ structures with 4 datapoints in the down sub-tree at level (1) and we can sort a list of 4 datapoints with a maximum of 5 comparisons. This tells us that it will never take more than $O(n)$ time to build the structures in the next dimension at level (1) in the best case.

The time to rebuild structures in the next dimension on any other level ℓ is also seen to also be $O(n)$ as we can build a sorted list for the structures by merging the sorted lists found in the structures in the next dimension which are rooted at the nodes in the immediate down sub-tree of the current node. We can merge two disjoint sorted lists in O(X) time , where X is the number of nodes in the two sorted lists, and thereby build the structures at level ℓ in $O(n)$ time (as the total number of points in the structures in the next dimension at level ℓ is n). We must rebuild structures at $\lg n$ levels so we see that building the structures in the next dimension takes $O(n \lg n)$ time. Adding this time to the time taken to build the first dimension gives us a total construction time of $O(n \lg n)$ for a 2-d Range DSL.

Proceeding from the second dimension to the third dimension would be tricky if it wasn't for the fact that the construction time for dimension k depends only on the number of nodes in dimension k. By Lemma 1 we know that there are at most $O(n \lg^{k-1} n)$ nodes in dimension k and this tells us that the time to build the structures in dimension k is at most $O(n \lg^{k-1} n)$. We see that the term for dimension k is dominant and so the time, $P_S(n,k)$, it takes to construct a k-d Range DSL is calculated to be $O(n \lg^{k-1} n)$ as we set out to show.■

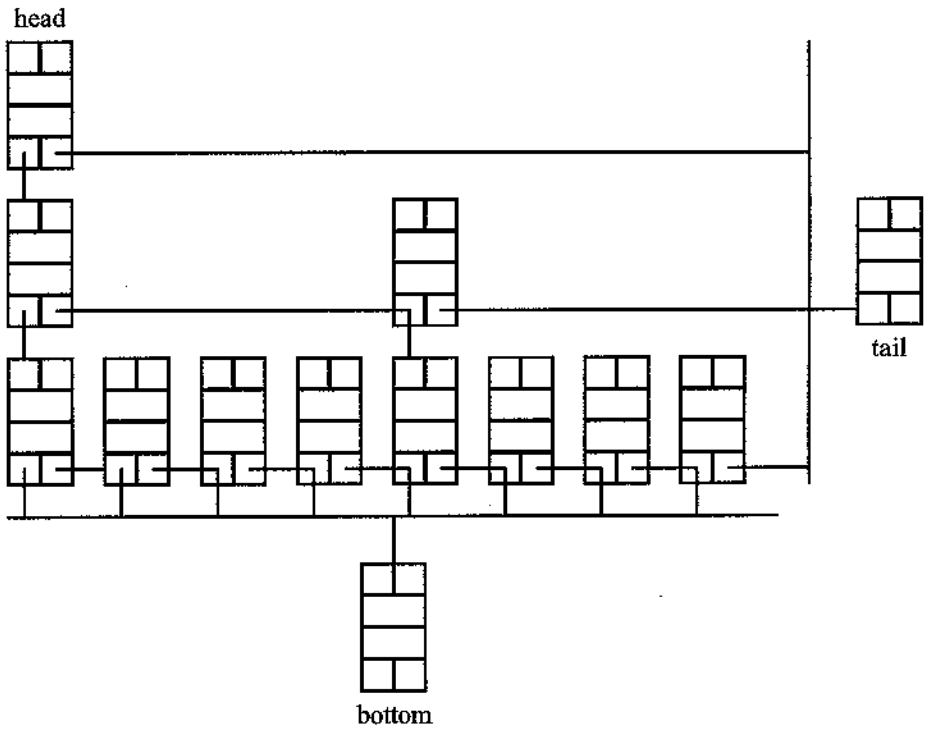


Figure 8: A best-case DSL structure.

3.3.3.2 General Insertion in a k-d Range DSL

When we define a general insertion as an insertion where we may have to rebuild a structure in the next dimension (an insertion in which a split may occur) we find that the worst case insert time increases to $O(n \lg^{k-1} n)$ for a single insertion, but, as with the range tree, the worst case can only occur upon inserting a minimum number of points, dependent on n, into the structure (see Figure 9).

For example, in Figure 9, after the worst-case insertion of (5,215), we see that we have to rebuild the tree B in dimension 2 as three (3) datapoints move from the left sub-tree of the root (B in 9a) to the right sub-tree of the root (C in 9a). The tree B contains roughly $n/2$ points and thus takes $O(n)$ time to rebuild, as predicted (remember that we are only dealing with a 2-d range tree and rebuilding in dimension 2 only). If we were dealing with a 3-d range tree for the data of Table 3 (which is table 2 where a coordinate for dimension 3 has been added to each datapoint), then, for each interior node in B, we would have to rebuild a range tree in the third dimension. It would take $O(n \lg n)$ time to rebuild the trees in dimension 3 attached to the tree B so the total rebuilding time would be $O(n \lg n)$ (as we are rebuilding only in dimensions 2 and 3). The reader is referred to [Lamo95d] for an exposition of the implementation of a k-d dynamic range tree.

In the case of the dynamic range tree, the worst case insertion (which gives us a rebuilding time of $O(n \lg^{k-2} n)$) is guaranteed to occur in a sequence of $O(n)$ insertions (when we start with n datapoints in the structure, see [Fred81] and [Will85]) and we find the same is true for the k-d Range DSL. For example, in Figure 10, after the worst case insertion of (5, 215), we see that we have to build the Range DSL structures F & G attached to the new interior nodes at level 2. (Note that we assume the top down insertion algorithm. With a bottom up insertion algorithm, we wouldn't achieve the worst case insertion until after we had inserted (5,215) and (35, 185).) The number of nodes in the two trees combined is n and so it takes $O(n)$ time to rebuild the structures. If we were dealing with a 3-d Range DSL for the data of Table 3 (which is table 2 where a coordinate for dimension 3 has been added to each datapoint), then, for each non-leaf node in the structures F and G, we would have to rebuild a DSL structure in the third dimension. It would take $O(n \lg n)$ time to rebuild the DSLs in dimension 3 attached to the DSLs F and G and so the total rebuilding time would be $O(n \lg n)$ (as we are rebuilding only in dimensions 2 and 3).

However, whereas we need at most n insertions to raise the height of the range tree, we may need as many as $2n$ insertions to raise the height of the k-d Range DSL when we start with n datapoints and a worst-case DSL as in Figure 6. In this case, a split occurs which raises the height of the k-d Range DSL in the first dimension and we need to rebuild the two structures in the next dimension at level h-1 (remember that we propagate the head node up a level to avoid unnecessary rebuilding). These two structures contain $n/2$ points each and we know that, from Theorem 4.2, we require a time of $O(n \lg^{k-2} n)$ to rebuild these structures as we are rebuilding two (k-1)-d structures.

Also, as with the dynamic range tree, the 2nd worst insertion (which is when we rebuild a structure in the next dimension at level h-2) can only occur after $O(n/2)$ insertions (and so only two can occur in a sequence of $O(n)$ insertions), the 3rd worst insertion (which is when we rebuild a structure in the next dimension at level h-3) can only

occur after $O(n/4)$ insertions (and so only four can occur in a sequence of $O(n)$ insertions), ..., and the insertion that requires the minimum rebuilding (which is when we rebuild a structure in the next dimension at level 1) can only occur every $n/2^{\lg n-1}$ (2) insertions. We see that we need to rebuild structures accounting for at most $O(n)$ points at each level after a sequence of $O(n)$ insertions so we can rebuild each level at a cost of $O(n \lg^{k-2} n)$ and thus complete all rebuilding in $O(n \lg^{k-1} n)$ time as there are only $O(\lg n)$ levels in which rebuilding is needed.

Thus, we can divide our total rebuilding time of $O(n \lg^{k-1} n)$ after a worst case sequence of insertions by the number of insertions, which is $O(n)$, to get an amortized worst case analysis of $O(\lg^{k-1} n)$ for necessary rebuilding in a general insertion.

Theorem 2.2 The time $I(n,k)$ required for an insertion in a k-d Range DSL is $O(\lg^k n)$.

Proof:

It takes $O(\lg^k n)$ work to perform a simple insertion and $O(\lg^{k-1} n)$ work to do any necessary rebuilding using an amortized worst case analysis (see argument above). Thus an insertion takes $O(\lg^k n)$ time. ■

Table 2: Datapoints for the 2-d Range Tree of Fig. 9 & the 2-d Range DSL of Fig. 10.

K_1	K_2
65	90
125	110
180	25
55	155
140	125
25	195
95	60
45	131
15	175
75	205

Table 3 Datapoints for a 3-d Range Tree (not shown) and a 3-d Range DSL (not shown).

K_1	K_2	K_3
65	90	100
125	110	130
180	25	30
55	155	160
140	125	40
25	195	80
95	60	140
45	131	20
15	175	50
75	205	70

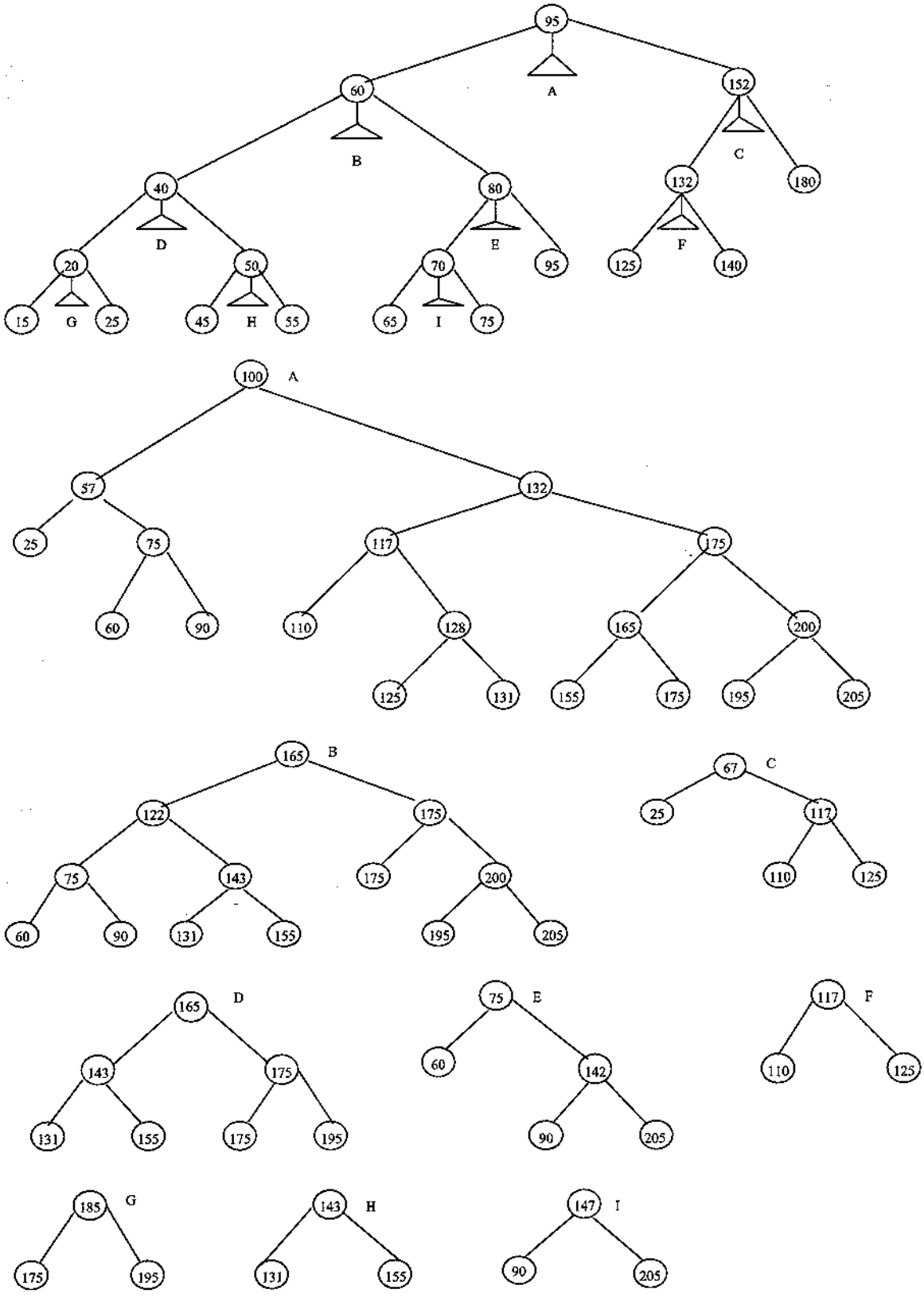


Figure 9a) 2-d Range DSL before the worst case insertion of (5,215).

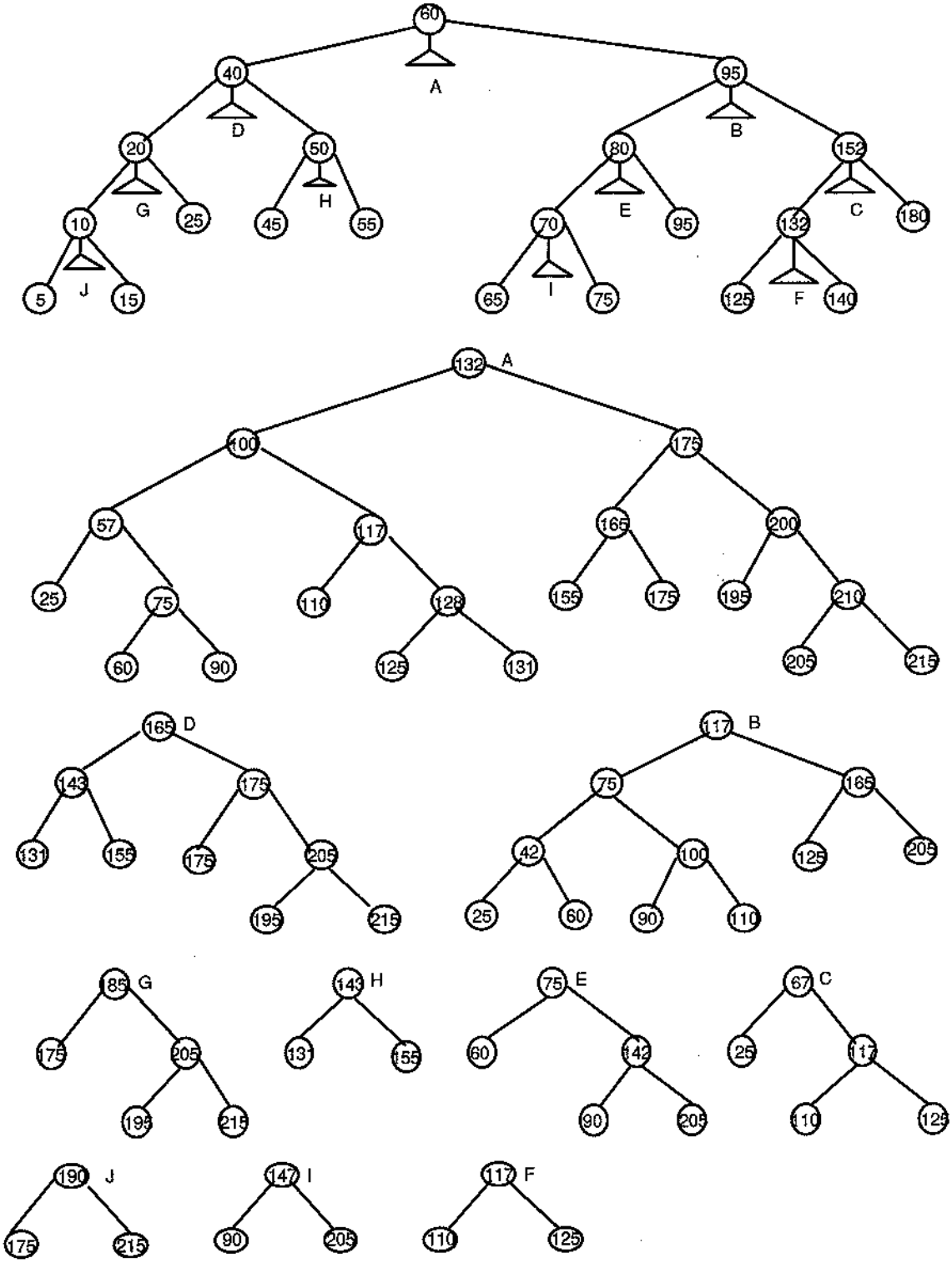


Figure 9b) 2-d Range Tree after the worst case insertion of (5,215).

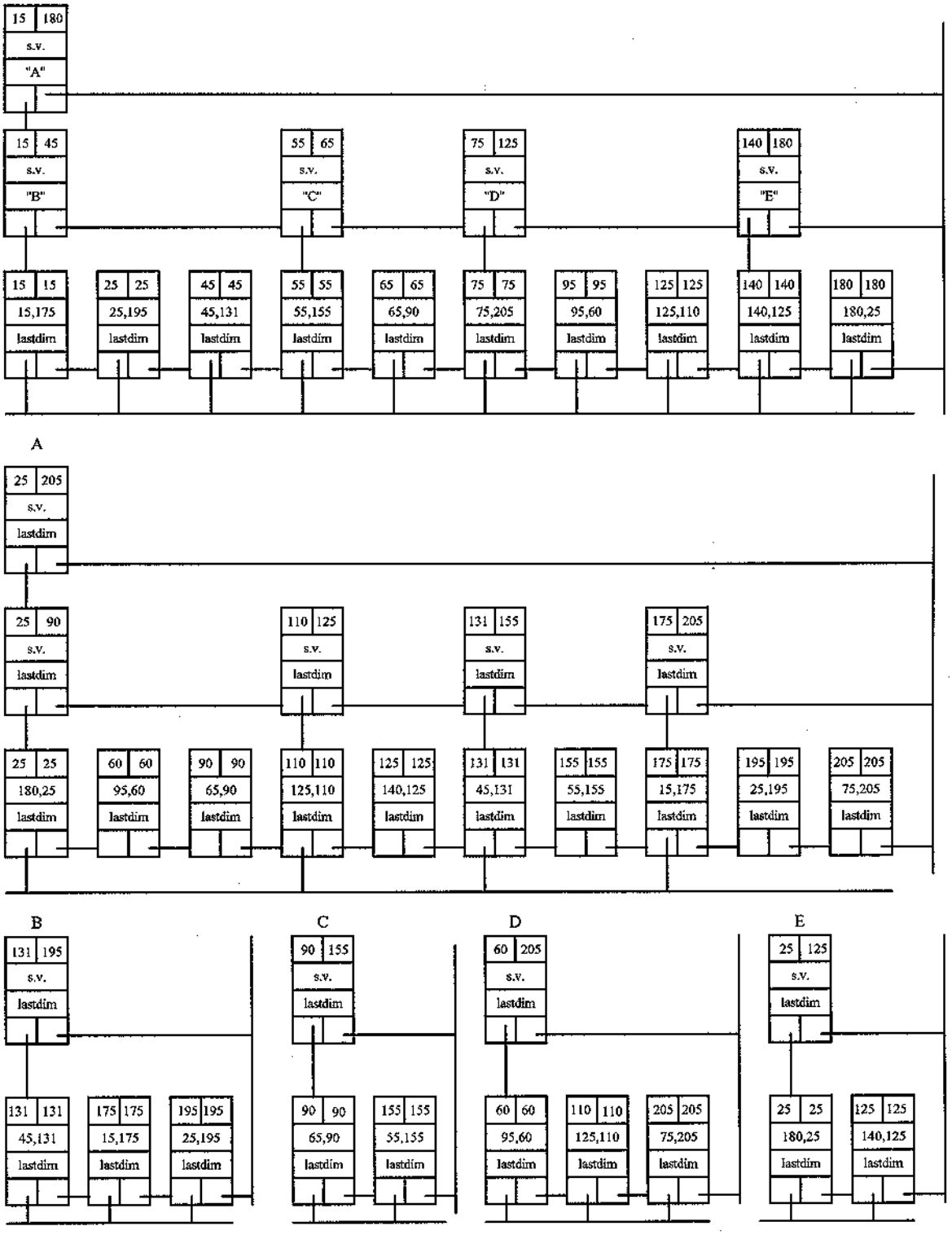


Figure 10a) 2-d Range DSL before the worst case insertion of (5,215). As per usual, the tail nodes and bottom nodes have been omitted to save space.

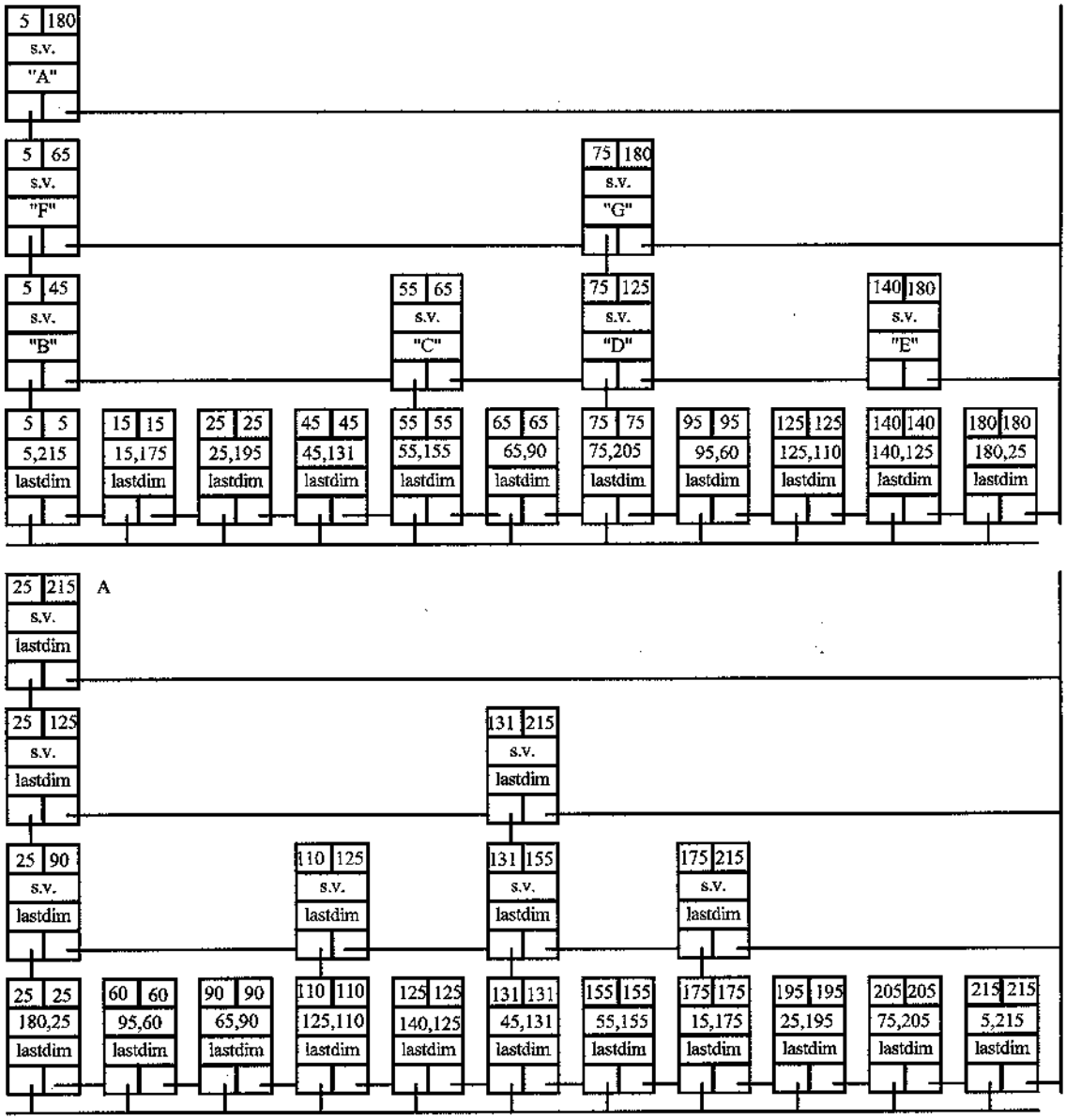


Figure 10b) The 2-d Range DSL after the worst case insertion of the node (5,215). As per usual, the tail nodes and bottom nodes have been omitted to save space.

Figure 10b) The 2-d Range DSL after the worst case insertion of the node (5,215). As per usual, the tail and bottom nodes have been omitted to save space (continued).

3.3.4.1 Simple Deletion in a k-d Range DSL

Deletion proceeds similar to that of the 1-d 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 DSLs in the next dimension. Again, note that the algorithm relies on the global variables head, tail, bottom, and lastdim as previously defined and the concepts of immediate down sub-trees and gap size. In addition, we assume (for simplicity) that all of the K_i coordinate values are unique and that the element is to be found in the structure. These assumptions are valid ones as 1) we can perturb datapoints (and use real numbers) if necessary and 2) we can insure that an element is located in the structure by first performing a search for the element (which only takes O(lg n) time as, if it is in the structure then, by definition of the structure, it must be in the first dimension).

We should also note that, like the 2-d search skip list of [Nick94], we must keep track of the path followed to the bottom on the deletion path as we must trace back up the structure after removing the datapoint (and associated node) as we must 1) change minki and maxki values as necessary and 2) remove the datapoint (and associated node) from the appropriate k-d Range DSLs in the next dimension. (We can remove the datapoint from the k-d Range DSLs in the next dimension on the way down if we so choose.)

The deletion algorithm, in summary, is as follows:
(Let D be the node being deleted.)

1) Start at the head node of the k-d Range DSL.
2) Record the node in a path list P (a simple linked list is sufficient).
3) If the gap G we are going to drop into is of size 2 or 3 then drop. (Removing an element from a gap of size 2(3) gives us a gap size of 1(2) which is valid.).
4) If the gap G 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'
     Else
       remove an element from G' and insert it into G
5) Continue until we reach the bottom level, where we remove the element of height 1. If the node to be deleted is not of height 1, swap with the predecessor(successor).
6) Follow the pointers in the list P (top down or bottom up), reestablishing the minki and maxki values and removing the node D from the associated k-d Range DSLs (if not in the last dimension) in the next dimension for all nodes on the deletion path.

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 inserted element is indeed a valid 1-d skip list. In addition, the minki and maxki values of the nodes on the path are adjusted as necessary with respect to the deleted node's minki and maxki values so the minki and maxki values for a node will always be correct.

We will define a simple deletion analogous to the definition of a simple insertion as a deletion where we don't have to rebuild a k-d Range DSL in the next dimension (i.e. a deletion where we don't do a merge (drop an element one level)).

Theorem 5.1: The time $D_S(n,k)$ to perform a simple deletion of a point in a k-d Range DSL is $O(\lg^k n)$.

Proof:

The proof depends on the observation that the time required for deletion of a node in a 1-d Range DSL depends on the time required to find the node being deleted as the operations of removing a node and borrowing a node (removing a node from one sub-tree and inserting it into another) require constant time (as only a constant number of pointers and values need be changed). It is also true that we may have to change the minki or maxki value of each node visited, but this operation takes constant time as well and the number of nodes visited is accounted for by the search time to locate the node being deleted. Thus, deletion of a point from a 1-d Range DSL takes $O(\lg n)$ time.

The only difference in deleting a point from a 2-d Range DSL as compared to deleting a point from a 1-d Range DSL is that we must also remove the point from the 1-d Range DSL pointed to by the nextdim pointer in the last node visited at each level on the way down to the bottom of the 1-d Range DSL in the first dimension. Thus, we have to delete the point from an additional $O(\lg n)$ structures which each have a deletion time of $O(\lg n)$, giving us an overall deletion time of $O(\lg^2 n)$.

The proof that $D_S(n,k) = O(\lg^k n)$ follows from the proof that $Q(n,k) = O(\lg^k n + t)$. In dimension k we must delete the point from a total of $O(\lg^{k-1} n)$ structures associated with the structures that we deleted the point from in dimension k-1. As it takes $O(\lg n)$ time to delete the point from a single structure, we see that it takes $O(\lg^k n)$ time to delete the point from a k-d structure. ■

3.3.4.2 General Deletion in a k-d Range DSL

When we define a general deletion as a deletion where we may have to rebuild a structure in the next dimension (a deletion where a merge may occur) we find that the worst case delete time increases to $O(n \lg^{k-1} n)$ for a single deletion, but, as with the range tree, the worst case can only occur upon deleting a minimum number of points, dependent on n, from the structure.

In the case of the dynamic range tree, the worst case deletion (which gives us a rebuilding time of $O(n \lg^{k-1} n)$) occurs after a sequence of $O(n)$ deletions (where we start with $O(n)$ datapoints in the structure) and we find that the same is true for the k-d Range

DSL. (However, whereas we need at most n deletions to lower the height of the range tree, we may need as many as $2n$ deletions to lower the height of a k-d Range DSL where we start with $3n$ datapoints and a worst case DSL as in Figure 6.) In this case a merge occurs at level h-1 and we must rebuild a structure which contains roughly $n/2$ datapoints. This is known to take $O(n \lg^{k-2} n)$ time by Theorem 4.2.

As the worst case deletion causes merges all the way up the structure (just as the worst case insertion causes splits all the way down the structure) we must rebuild structures at all lg n levels in a time bounded at $O(n \lg^{k-2} n)$ per level to get our worst case deletion time of $O(n \lg^{k-1} n)$.

As the sequence of deletions, like the sequence of insertions, implies that the 2nd worst deletion (which is when we perform a merge at level h-2 and must rebuild a structure in the next dimension that contains roughly $n/4$ datapoints) can only occur after $O(n/2)$ deletions, that the 3rd worst deletion (which is when we perform a merge at level h-3 and must rebuild a structure in the next dimension that contains roughly $n/8$ datapoints) can only occur after $O(n/4)$ deletions, ..., and the deletion requiring the minimum rebuilding time (which is when we perform a merge at level 1 and must rebuild a structure in the next dimension that contains roughly 2 datapoints) can only occur after 2 deletions, we see that we need to rebuild structures at each level which account for at most $O(n)$ points and that we can thus complete all rebuilding for a sequence of $O(n)$ deletions in $O(n \lg^{k-1} n)$ time as there are only lg n levels in which rebuilding is needed.

Thus, we can divide our total rebuilding time of $O(n \lg^{k-1} n)$ after a worst case sequence of deletions by the number of deletions, which is $O(n)$, to get an amortized worst case analysis of $O(\lg^{k-1} n)$ for necessary rebuilding in the case of a general deletion.

Theorem 5.2: The time $D(n,k)$ required for a deletion in a k-d Range DSL is $O(\lg^k n)$.

Proof:

It takes $O(\lg^k n)$ work to perform a simple deletion and $O(\lg^{k-1} n)$ work to do any necessary rebuilding using an amortized worst case analysis (see argument above). Thus, a deletion takes $O(\lg^k n)$ time. ■

Theorem 6: The k-d Range DSL is an optimally balanced structure.

Proof:

The balance cost is $P(n,k) * S(n,k) * I(n,k) * D(n,k) * Q(n,k) =$

$O(n \lg^k n) * O(n \lg^{k-1} n) * O(\lg^k n) * O(\lg^k n) * O(\lg^k n) =$

$O(n \lg^k n * n \lg^{k-1} n * \lg^k n * \lg^k n * \lg^k n) = O(n^2 \lg^{5k-1} n) = O(n^2 \lg^k n)$

which we know is optimal by [Lamo95b]. ■

4 CONCLUSIONS

The k-d Range DSL is a very practical way to carry out range search in optimal time in a completely dynamic environment. It's similarity to the B-tree should make it of practical use in indexing large datasets and databases. The algorithms (shown in their complete functional (Borland Turbo) Pascal code implementations in the appendices) are concise, easy to understand and implement, and inherit the simplicity of the skip list algorithms in general. Deletion and range search, in particular, take advantage of the extra horizontal (i.e. right) pointers not normally incorporated into B-trees to simplify the top down algorithms.

This structure is a very powerful one as it seems well suited to a parallel processing environment (a search in one dimension is independent of a search in the previous dimension and when a path splits in two down the DSL, each path can be processed separately) and it can efficiently handle arbitrary k-d semi-infinite range queries as it incorporates the search skip list structure of [Nick94].

Our conclusions are that skip list data structures provide efficient alternatives to balanced tree structures and that the ease in which they can be modified and extended should provide many applications programmers with a choice of data structures to use to achieve their goals.

5 REFERENCES

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

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

[Bent80a] Bently, J.L. "Multidimensional Divide-and-Conquer", *Communications of the ACM*, Vol. 23, No. 4, (April) 1980, pp. 214 - 229.

[Bent80b] Bently, J.L., and Maurer, H.A. "Efficient Worst-Case Data Structures for Range Searching", *Acta Informatica*, Vol. 13, 1980, pp. 155-168.

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

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

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

[Grah94] Graham, R.L., Knuth, D.E., and Patashnik, O. *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1991. Second Edition, 1994.

[John91] Johnson, Theodore "A Highly Concurrent Priority Queue Based on the B-link Tree" University of Florida, Department of CIS, Electronic Technical Report # 007-91.

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

[Lamo95a] Lamoureux, Michael G., and Nickerson, Bradford G. "On the equivalence of B-trees and deterministic skip lists", [submitted for publication] November 1995.

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

[Lamo95c] Lamoureux, Michael G. "On The Number of Orthogonal Range Queries in k-space", University of New Brunswick Technical Report, [in preparation].

[Lamo95d] Lamoureux, Michael G. "An implementation of a multidimensional dynamic range tree using an AVL tree", University of New Brunswick Technical Report, [in preparation].

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

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

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

[Munr92] Munro, J.I., Papadakis, T., and Sedgewick, R. "Deterministic skip lists", Proc. of the ACM-SIAM Third Annual Symposium on Discrete Algorithms, Orlando, Florida, Jan 27-29, 1992, pp. 367 - 375.

[Nick94] Nickerson, Bradford G. "Skip List Data Structures for Multidimensional Data", Computer Science Technical Report CS TR-3262 UMIACS-TR-94-52, April 1994; available via anonymous ftp site: "ftp.cs.umd.edu" in file "/pub/papers/TRs/3262.ps".

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

[Pugh90] Pugh, W. "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, Vol. 33, No. 6, (June) 1990, pp. 668 - 676.

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

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

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

APPENDIX 1

A (Borland Turbo) Pascal Implementation of the k-d Range DSL

As the implementation of the k-d Range DSL requires a substantial amount of code (about 1500 lines), the procedures have been placed in a number of different units in a manner such that code pertaining to a specific function (or functions) is (are) grouped together. This appendix contains the complete working code for the k-d Range DSL and as such contains the code for one driver program and 12 associated units. The code has been placed into the following units on the following pages and each unit contains a description of its function and the procedures found within.

RANGEDSL.PAS {Main Program}

```
Program RANGEDSL;

{$M 65520, 65536, 655360}
{use maximum stack and allow for maximum heap in real mode compilation}

{This is the code for the main program.  It calls the unit RDSLDECS which contains all
the constant and type declarations and necessary global variables; and the units
RDSLSTST, RDSLITST, and RDSLDTST which contain the driver code for testing the
search, insert, and delete procedures}

{Notes on compilation:  compile the units (to disk) in the following order:
  RDSLDECS -        declarations unit
  RDSLOUT -         report unit for searches
  RDSLSRCH -        search unit
  RDSLSTST -        driver unit for search testing
  RDSLBLD -         building/initialization unit
  RDSLRBLD -        initialization/rebuilding unit
  RDSLDMEM -        disposal unit
  RDSLDBUG -        "visual" debugging unit
  RDSLINS -         dynamic insertion unit
  RDSLDEL -         dynamic deletion unit
  RDSLITST -        driver unit for insertion testing
  RDSLDTST -        driver unit for deletion testing
  RangeDSL -        main program                          }

Uses RDSLDECS, RDSLSTST, RDSLITST, RDSLDTST;

Begin {Main}

  BuildRDSL(head, tail, bottom, lastdim);  {build the RDSL structure}

  Search(head);  {Search the structure}

  DestroyRDSL(head);  {dispose of the RDSL structure}

End.  {Main}
```

RDSLDECS.PAS {Declarations Unit}

Unit RDSLDECS;
{This unit contains the constant and type declarations for the RDSL structure along with the required global variables. It is referenced and required by all of the other units.

The main units are:

RDSLDECS	*this unit; the declarations unit*
RDSLSRCH	*this unit contains the Range Search & Member Query procedures*
RDSLOUT	*this unit contains the code to output points found by range search*
RDSLBLD	*this unit contains the code to initialize the RDSL structure*
RDSLRBLD	*this unit contains the procedures to build an RDSL from static data {used when we need to rebuild (in nextdim) due to a split/merge}*
RDSLDMEM	*this unit contains procedures to return unneeded memory (this program uses Dynamic Memory Allocation)*
RDSLDBUG	*contains procedures for "viewing" the RDSL in a datafile*
RDSLINS	*contains the code for inserting a datapoint into the RDSL*
RDSLDEL	*contains the code for deleting a datapoint from the RDSL*

The secondary units are:

RDSLSTST	*contains the driver code to test the range search procedure*
RDSLITST	*contains the driver code to test the insert procedure*
RDSLDTST	*contains the driver code to test the delete procedure*

The main program is:

RangeDSL.pas	*calls the driver units to build, test, and destroy the RDSL*

Interface

{We assume unique keys AND unique coordinate values for simplicity}
{Also, the code assumes an underlying 1-3 DSL structure by default. It could be modified, with some work, to allow for an order m DSL, but, for simplicity of coding and of illustration, a 1-3 DSL is used.}

```
Const maxdim = 3;      {maximum number of dimensions that we will have in our RDSL}
                       {this constant is needed as we store a datapoint in an array}
    dpointsfile = 'dpoints.dat'; {the datafile that contains the datapoints}
                       {must contain at least numpoints datapoints}
    dimensions = 3;    {the number of dimensions of the data}
                       {must be less than or equal to maxdim and at least 1}
                       {i.e. we must change this constant and recompile code to work with
                       a different number of dimensions in this simple implementation -
                        this can be changed by making this a variable and then we can
                        use any number of dimensions up to maxdim}
    mxnpl = 3;         {when rebuilding from an essentially "static" data
                       (2 <= mxnpl <=4)  set, we will use a gap size of mxnpl - 1}
                       {# of nodes placed in immediate down subtree when rebuilding}
    order = 3;         {order of the DSL - CANNOT BE CHANGED!!!}

    minkey = -32767;          {specify integer range; all coordinates must come}
    maxkey =  32767;          {between (excluding) these values}
    mx     = maxkey;          {abbreviation for maxkey}
    mn     = minkey;          {abbreviation for minkey}
        {minkey and maxkey could change if the integer datatype was changed}
    searchfile = 'sres.dat';  {datapoints found by search procedure are stored here}

Type  coordinate = array [1..maxdim] of integer; {stores a datapoint}

    dfiletype  = file of coordinate;    {the input file type}

    nodeptr = ^node;                    {points to a node}
    node    = record
              minki, maxki: integer;    {min and max values}
              datapoint: coordinate;    {the datapoint}
              nextdim: nodeptr;         {a pointer to a RDSL in the nextdim}
              down, right: nodeptr;     {standard down and right pointers}
             end;

{if we wish to store auxiliary information with the datapoint, we add the additional field
   auxdata: ^auxinfo_type;
which contains a pointer that points to the auxiliary information associated with the
datapoint  (and auxdata points to nil when the datapoint field is set to the sentinel value
at a nonleaf node)}
```

```
    listptr = ^list;                    {used to keep a list of nodes}
    list    = record
                RDSLnode: nodeptr;      {a node in the structure}
                next: listptr;          {the next node in the list}
                prev: listptr;          {the previous node in the list}
              end;

Const sv: coordinate = (0,0,0{,0,0,0,0,0,0,0});
    {sentinel value: non-leaf level}
    AMax: coordinate = (mx,mx,mx{,mx,mx,mx,mx,mx,mx,mx});
      {used to initialize a node}
    AMin: coordinate = (mn,mn,mn{,mn,mn,mn,mn,mn,mn,mn});
      {used to initialize a node}

Var head, tail, bottom, lastdim: nodeptr;  {sentinel nodes of the RDSL}
  sfile: dfiletype;  {datafile variable points to the file used to store range search results}
  L, R: coordinate;  {specify the search range of the range search}
  numpoints = integer; {the number of points we will be inserting}
                       {used by RDSLITST in building an RDSL during testing}
                       {used by RDSLDTST in destroying an RDSL during testing}

Implementation

End. {Implementation - RDSLDECS}
```

RDSLBLD.PAS {Construction Unit}

```
Unit RDSLBLD;

{This unit contains the procedures for initilizing and creating a new RDSL structure.}

Interface

Uses RDSLDECS;

Procedure CreateEmptyRDSL(Var head, tail, bottom, lastdim: nodeptr);
{This procedure is called to construct and initialize a new RDSL structure.}

Procedure NewEmptyRDSL(Var nhead: nodeptr);
{This procedure is called when we wish to create a new RDSL structure in the next
dimension which is going to be attached to an existing structure.}

Procedure CopyNode(current: nodeptr; Var cc: nodeptr; dim: integer);
{When we are inserting a datapoint into the next dimension, we must get a new node and
store the datapoint in that new node.  This procedure gets a new node and initializes it
with the relevant data.}

Procedure FCopyNode(current: nodeptr; Var cc: nodeptr);
{When we are rebuiling a structure in the next dimension and using already existing
structures in the next dimension as the basis for the rebuilding, we need simply make
copies of already existing nodes rather than initialize new nodes so we call this
procedure instead of the procedure CopyNode (above).}

Procedure InitHeadNode(head: nodeptr);
{This procedure initializes the head node in an empty RDSL.  It is used and usually
called by procedures CreateEmptyRDSL and NewEmptyRDSL, but it is also called by the
Delete procedure to reinitialize the RDSL structure to empty when the last node has been
deleted.}

Implementation
```

```
Procedure InitHeadNode;
{Initialize the head node in an empty RDSL structure.}

Begin {InitHeadNode}

 head^.minki:= maxkey;      head^.maxki:= minkey;
 head^.datapoint:= sv;      head^.nextdim:= lastdim;
 head^.down:= bottom;       head^.right:= tail;

End; {InitHeadNode}

Procedure InitTailNode(tail: nodeptr);
{Initialize the tail node in an empty RDSL structure.}

Begin {InitTailNode}

 tail^.minki:= maxkey;      tail^.maxki:= maxkey;
 tail^.datapoint:= AMax;    tail^.nextdim:= lastdim;
 tail^.down:= tail;         tail^.right:= tail;

End; {InitTailNode}

Procedure InitBottomNode(bottom: nodeptr);
{Initialize the bottom node in an empty RDSL structure.}

Begin {InitBottomNode}

 bottom^.minki:= minkey;    bottom^.maxki:= minkey;
 bottom^.datapoint:= Amin;  bottom^.nextdim:= lastdim;
 bottom^.down:= bottom;     bottom^.right:= bottom;

End; {InitBottomNode}

Procedure InitLastdimNode(lastdim: nodeptr);
{Initialize the lastdim node in an empty RDSL structure.}

Begin {InitLastdimNode}

 lastdim^.minki:= maxkey;   lastdim^.maxki:= minkey;
 lastdim^.datapoint:= sv;   lastdim^.nextdim:= lastdim;
 lastdim^.down:= lastdim;   lastdim^.right:= lastdim;

End; {InitLastdimNode}
```

```
Procedure CreateEmptyRDSL;
{This procedure creates a new empty RDSL and initializes the sentinel nodes}

Begin {CreateEmptyRDSL}

 {Allocate the nodes}
 new(head);   new(tail);
 new(bottom); new(lastdim);

 {Initialize the nodes}
 InitHeadNode(head);          {Initialize head node}
 InitTailNode(tail);          {Initialize tail node}
 InitBottomNode(bottom);      {Initialize bottom node}
 InitLastdimNode(lastdim);    {Initialize lastdim node}

End; {CreateEmptyRDSL}

Procedure NewEmptyRDSL;
{This procedure creates a new empty RDSL structure for the next dimension}

Begin {NewEmptyRDSL}

 new(nhead);
 InitHeadNode(nhead);

End; {NewEmptyRDSL}

Procedure CNChangePointers(current: nodeptr; Var cc: nodeptr);
{This procedure initializes the nextdim and down pointers in a node that is to be inserted
into the RDSL.  As all insertions happen at the leaf level, we can automatically set the
nextdim pointer to point to the lastdim node and the down pointer to point to the bottom
node.}

Begin {CNChangePointers}

 cc^.nextdim:= current^.nextdim;    {cc^.nextdim:= lastdim}
 cc^.down:= current^.down;          {cc^.down:= bottom}
 {cc^.right is set as the structure is built}

End; {CNChangePointers}
```

```
Procedure NewDpoint(current: nodeptr; Var cc: nodeptr; dim: integer);
{This procedure initializes the datapoint, minki, and maxki fields in a node that is to be
inserted into the RDSL.  The datapoint field simply gets the datapoint that is to be
inserted and the minki and maxki simply get the coordinate of the datapoint in the
current dimension (dim).}

Begin {NewDpoint}

 cc^.minki:= current^.datapoint[dim];   cc^.maxki:= current^.datapoint[dim];
 cc^.datapoint:= current^.datapoint;

End;  {NewDpoint}

Procedure Copynode;
{Used to copy a node for a structure in the next dimension}

Begin {CopyNode}

 new(cc);
 NewDpoint(current, cc, dim);
 CNChangePointers(current, cc);

End;  {CopyNode}

Procedure FCopyNode;
{Used to copy a node for a structure in the same dimension}

Begin {FCopyNode}

 new(cc);
 cc^.datapoint:= current^.datapoint;
 cc^.minki:= current^.minki;  cc^.maxki:= current^.maxki;
 CNChangePointers(current, cc);

End; {FCopyNode}

End. {Implementation - RDSLBLD}
```

RDSLSRCH.PAS {Search Unit}

```
Unit RDSLSRCH;

Interface

Uses RDSLDECS;

Function DPointInRDSL(dp: coordinate;  head: nodeptr): boolean;
{This procedure is used to provide member query on the RDSL structure.}

Procedure KDRSEARCH(head: nodeptr; prev: nodeptr; next: boolean; dim: integer);
{This procedure is used to provide range search on the RDSL structure.}

Implementation

USES RDSLOUT;

Procedure KDRSEARCH;
{This procedure is used to provide range search on the RDSL structure.}

Var snode: nodeptr;    {use a "sentinel node" to insure we don't leave the down subtree}

 Procedure REPORT(head, snode: nodeptr);
 {We have determined all points in a down subtree to be in range.  Report them.}

 Begin {Report}

   While head^.down <> bottom Do
    Begin {go down, not yet at leaf level}
     head:= head^.down;
     snode:= snode^.down; {head and snode should be at same level}
    End;  {go down, not yet at leaf level}

   While head <> snode Do
    Begin {go right, not yet at last node in down subtree}
     ReportPoint(head^.datapoint);
     head:= head^.right;
    End;  {go right, not yet at last node in down subtree}

 End;  {Report}
```

```
Procedure CheckPoint(head: nodeptr; dim: integer);
{This procedure is used to check if a datapoint is in range.}

Var i: integer;                {loop control variable to cycle through coordinates}
  stillinrange: boolean;       {temporary boolean variable}

Begin {CheckPoint}

 i:= dim;      {we need only check coordinates in current and succeeding dimensions}
 stillinrange:= true; {assume point in range until it is shown otherwise}

 With head^ Do
  While ((i <= dimensions) and (stillinrange)) Do
   {While datapoint still in range and not all coordinates checked}
   IF ((L[i] <= datapoint[i]) and (datapoint[i] <= R[i])) Then
    {current coordinate is in range, check next}
    i:= i + 1
   Else
    {datapoint has been determined not to be in range}
    stillinrange:= false;

  IF stillinrange Then
   ReportPoint(head^.datapoint);  {report point found in range}

End;  {CheckPoint}

Begin {KDRSEARCH}

 IF ((head<>nil) and (head<>bottom) and (head<>tail) and (head<>lastdim)) Then
  Begin {we haven't finished searching}
   {we can't go beyond the last node in the down subtree; snode is the
    first node in the next down subtree and the node that we can't go to.}
   IF head = prev Then
    snode:= head^.right
   Else
    snode:= prev^.right^.down;

   IF next Then
    Begin {find the right node in the down subtree to continue search}
     While (L[dim] > head^.maxki) Do
      head:= head^.right;
     next:= false;
    End;  {find the right node in the down subtree to continue search}
```

```
      IF (head^.down <> bottom) Then
       Begin {IF not at leaf level}
        IF not ((L[dim] > head^.maxki) or (R[dim] < head^.minki)) Then
         IF ((L[dim] <= head^.minki) and (head^.maxki <= R[dim])) Then
           Begin {all points in down subtree in range}
            IF dim <> dimensions Then
             KDRSearch(head^.nextdim, head^.nextdim, next, (dim+1))
            Else
             Report(head, head^.right);
            {insure that we check all of the immediate down subtree}
            IF head^.right <> snode Then
             KDRSearch(head^.right, prev, next, dim);
           End   {all points in down subtree in range}
          Else
           IF R[dim] <= head^.maxki Then
            IF L[dim] <= head^.minki Then
             KDRSearch(head^.down, head, next, dim)
            Else
             KDRSearch(head^.down, head, (true), dim)
           Else  {R[dim] > maxki and minki <= L[dim] <= maxki}
            Begin {we need to continue in down and right subtrees}
             KDRSearch(head^.down, head, (true), dim);
             IF head^.right <> snode Then
              KDRSearch(head^.right, prev, next, dim);
            End   {we need to continue in down and right subtrees}
        End   {IF not at leaf level}
       Else {we are at the leaf level}
        Begin {Search leaf level}
         While ((head <> snode) and (R[dim] >= head^.maxki)) Do
          Begin {not yet checked all nodes in down subtree}
           CheckPoint(head, dim);   {Is datapoint in node in range?}
           head:= head^.right;      {go to next node that may be in range}
          End; {not yet checked all nodes in down subtree}
        End;  {Search leaf level}
    End; {we haven't finished searching}

End; {KDRSEARCH}

Function DPointInRDSL;
{This procedure is used to provide member query on the RDSL structure.}
```

```
 Function MemQuery(head: nodeptr; dp: coordinate): boolean;
 {This procedure is the procedure that makes the proper initializations
  and calls procedure KDRSEARCH to perform the member query.}

 Begin {MemQuery}

   L:= dp; {init Left = Right = datapoint to search for point}
   R:= dp;

   KDRSEARCH(head, head, false, 1); {perform the search}

   reset (sfile);

   {report the results of our search}
   IF not eof(sfile) Then
     MemQuery:= true
   Else
     MemQuery:= false;

 End; {MemQuery}

Begin {DPointInRDSL}

assign (sfile, searchfile);
rewrite(sfile);
  DPointInRDSL:= MemQuery(head, dp);     {perform the member query}
close  (sfile);

End; {DPointInRDSL}

End. {Implementation - RDSLSRCH}
```

RDSLINS.PAS {Insertion Unit}

```
Unit RDSLINS;

Interface

Uses RDSLDECS;

Procedure Insert_RDSL(Var head: nodeptr; current: nodeptr; dim: integer);
{This procedure is used to dynamically insert a point into the RDSL.}

Procedure MustRebuildRDSL(head: nodeptr; Var nhead: nodeptr; dim: integer);
{This procedure is also used by the insert and delete procedures when rebuilding needs to
take place in the next dimension. It is called by procedure Insert_RDSL when a split
occurs and a new node is created which needs to have a structure in the next dimension
of the points in it's down subtree created and attached to it. It is called by procedure
Del_RDSL when a merge occurs and the structure in the next dimension of the points in
the down subtree has to be recreated.}

Procedure DoubleRebuild(head1,head2: nodeptr; Var nhead1,nhead2: nodeptr; dim:
integer);
{This procedure is also called by procedure Del_RDSL. When a borrow occurs,
structures need to be rebuilt in the next dimension at two nodes. Given these nodes, this
procedure calls MustRebuildRDSL twice.}

Implementation

Uses RDSLBLD, RDSLRBLD, RDSLDMEM;

Procedure GRRDSL(head: nodeptr; Var nhead: nodeptr; dim: integer);
{This procedure is called by procedure Insert_RDSL when a split occurs and the newly
created node needs a structure in the next dimension of the points in its down subtree
built for it.}

Begin {GRRDSL}

 NewEmptyRDSL(nhead);
 head^.nextdim:= nhead;
 RebuildRDSL(nhead, head, (dim+1));

End; {GRRDSL}
```

```
Procedure MustRebuildRDSL;
{This procedure rebuilds an RDSL in the next dimension when a split or a merge occurs
and the structure in the next dimension attached to a given node needs to be rebuilt.}

Begin {MustRebuildRDSL}

 Disposeof(head^.nextdim);
 GRRDSL(head, nhead, dim);

End; {MustRebuildRDSL}

Procedure DoubleRebuild;
{This procedure is used to rebuild structures in the next dimension when a borrow occurs
and two nodes need their structures in the next dimension rebuilt. It calls procedure
MustRebuildRDSL twice to accomplish it's task}

Begin {DoubleRebuild}

 MustRebuildRDSL(head1, nhead1, dim);
 MustRebuildRDSL(head2, nhead2, dim);

End; {DoubleRebuild}

Procedure Insert_RDSL;
{This procedure is used to dynamically insert a new datapoint into the RDSL.}

Var cc: nodeptr;        {we need a copy of the new node to insert in the nextdim}
  nhead1,
  nhead2: nodeptr;      {when we need to create a new RDSL in the
                          nextdim, we need a new header node}
  tracer: nodeptr;      {when inserting, use tracer to trace through the RDSL structure}
  raisen: nodeptr;      {we are about to insert into an immediate down subtree with 3
elements,
                         we must raise the middle element and create a new node: raisen}
  tdpoint: coordinate; {used as a temporary variable in a swap}
```

```
Procedure InsertInND(Var nhead, cc: nodeptr; current: nodeptr; dim: integer);
{We must propogate the insertion of the new datapoint throughout the dimensions.}

Begin {InsertInND}

 {create a node for the datapoint in the next dimension and initialize it}
 CopyNode(current, cc, (dim+1));
 {actually insert the node in the next dimension through a recursive call}
 Insert_RDSL(nhead, cc, (dim+1));

End; {InsertInND}

Procedure CNewNode(Var raisen: nodeptr);
{When a split occurs, we must create a new node. This procedure gets that new node}

Begin {CNewNode}

 new(raisen);
 raisen^.datapoint:= sv;
 raisen^.nextdim:= lastdim;

End; {CNewNode}

Begin {InsertRDSL}

 IF ((head^.minki = maxkey) and (head^.maxki = minkey)) Then
  Begin {We have an empty RDSL}
   {First update the head node}
    {update head node's minki and maxki values}
   head^.minki:= current^.datapoint[dim];
   head^.maxki:= current^.datapoint[dim];

   IF dim <> dimensions Then
    Begin {not in the last dimension - insert in next dimension}
     {create a new empty RDSL and attach it to the nextdim pointer}
     NewEmptyRDSL(nhead1); head^.nextdim:= nhead1;
     {make a copy of the current node and insert it in the nextdim}
     InsertInND(nhead1, cc, current, dim);
    End; {not in the last dimension - insert in next dimension}

   {Now insert the new node}
    head^.down:= current; {change the head node's downpointer}
    current^.right:= tail; {update current}
  End {We have an empty RDSL}
```

```
ELSE
 Begin {The RDSL is not empty}
  {start our search at the head node}
  tracer:= head;
  While (tracer^.down <> bottom) Do
   Begin {While not at leaf level}
    While ((current^.datapoint[dim] > tracer^.maxki) and (tracer^.right <> tail)) Do
     tracer:= tracer^.right;
    IF (tracer^.maxki > tracer^.down^.right^.right^.maxki) Then
     Begin {gap of size 3 in immediate down subtree, raise middle}
      {create the new node}
      CNewNode(raisen);
      {alter the proper pointers}
      raisen^.down:= tracer^.down^.right^.right;
      raisen^.right:= tracer^.right;
      tracer^.right:= raisen;
      {alter the proper minki and maxki values}
      tracer^.maxki:= tracer^.down^.right^.maxki;
      raisen^.minki:= raisen^.down^.minki;
      raisen^.maxki:= raisen^.down^.right^.maxki;

      IF dim <> dimensions Then
       Begin {not in last dimension - alter next dimension}
        MustRebuildRDSL(tracer, nhead1, dim);
        GRRDSL(raisen, nhead2, dim);
       End; {not in last dimension - alter next dimension}

      IF current^.datapoint[dim] > tracer^.maxki Then
       tracer:= tracer^.right; {tracer:= raisen;}

      IF head^.right <> tail Then
       Begin {we must raise the height of the RDSL}
        {create the new node}
        CNewNode(raisen);
        {alter the proper pointers}
        raisen^.down:= head;
        raisen^.right:= tail;
        {alter the proper minki and maxki values}
        raisen^.minki:= head^.minki;
        raisen^.maxki:= head^.right^.maxki;
        IF dim <> dimensions Then
          GRRDSL(raisen, nhead1, dim);
        head:= raisen; tracer:= head;
       End; {we must raise the height of the RDSL}
     End; {gap of size 3 in immediate down subtree, raise middle}
```

```
        {if necessary, alter minki/maxki values of current node}
        IF tracer^.minki > current^.minki Then
          tracer^.minki:= current^.minki;
        IF tracer^.maxki < current^.maxki Then
          tracer^.maxki:= current^.maxki;

        {the insertion must be propogated through all the dimensions}
        IF dim <> dimensions Then
          InsertInND(tracer^.nextdim, cc, current, dim);

        tracer:= tracer^.down;  {Go down!}
      End;  {While not at leaf level}

    {we have reached the leaf level}
    IF tracer^.datapoint[dim] > current^.datapoint[dim] Then
      Begin {we are inserting into the position tracer occupies}
        {insert current after tracer}
        current^.right:= tracer^.right;
        tracer^.right:= current;
        {now "swap" datapoints}
        tdpoint:= current^.datapoint;
        current^.datapoint:= tracer^.datapoint;
        tracer^.datapoint:= tdpoint;
        {and change minki and maxki values}
        tracer^.minki:= tracer^.datapoint[dim];
        tracer^.maxki:= tracer^.datapoint[dim];
        current^.minki:= current^.datapoint[dim];
        current^.maxki:= current^.datapoint[dim];
      End   {we are inserting into the position tracer occupies}
    Else
      Begin {find where to insert current and insert}
        While (current^.datapoint[dim] > tracer^.right^.datapoint[dim]) Do
          tracer:= tracer^.right;
        current^.right:= tracer^.right;
        tracer^.right:= current;
      End;  {find where to insert current and insert}
  End;  {The RDSL is not empty}

End;  {InsertRDSL}

End. {Implementation - RDSLINS}
```

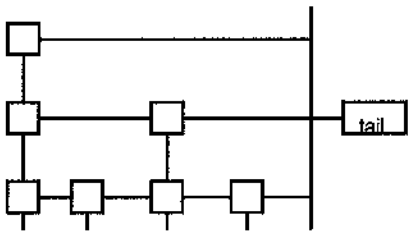
RDSLDEL.PAS {Deletion Unit}

```
Unit RDSLDEL;

Interface

Uses RDSLDECS;

Procedure Delete_RDSL(Var head: nodeptr; datapoint: coordinate; dim: integer);
{This procedure deletes a datapoint from the RDSL structure.}

Implementation

Uses RDSLDMEM, RDSLINS, RDSLSRCH, RDSLBLD;

Function Equal(dp1, dp2: coordinate): boolean;
{This function determines whether or not the two datapoints are equal.}

Var ss: boolean;        {temporarily variable, assigned to Equal, used in While}
  i: integer;           {used to cycle through coordinates of the datapoints}

Begin {Equal}

 ss:= true;     {assume datapoints equal until we determine otherwise}
 i:= 1;         {first dimension is dimension 1}

 While ((i <= dimensions) and (ss)) Do
  {While datapoints are equal and all coordinates are not checked Do}
  IF dp1[i] = dp2[i] Then
  {IF coordinates are equal, we need check next coordinate}
   i:= i + 1
  Else
   ss:= false; {datapoints are not equal}

 Equal:= ss; {return our findings}

End; {Equal}

Procedure Delete_RDSL;
{Datapoint is the datapoint we wish to remove and dim is the dimension we are currently
removing it from. The procedure first insures that the RDSL structure is not empty and
that the datapoint is in the RDSL before attempting to remove the datapoint from the
RDSL structure.}
```

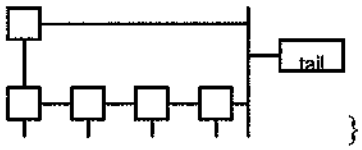
```
Procedure Del_RDSL(Var head: nodeptr; datapoint: coordinate; dim: integer);
{This procedure removes a node from the RDSL structure given the conditions:
 1) the datapoint we wish to remove is in the RDSL
 2) the datapoint that we are removing is not the last datapoint in the structure}

 Var tracer: nodeptr;    {the node we are currently working with}
   nhead1,
   nhead2: nodeptr; {when we borrow, two RDSL's in the nextdim change}
                    {when we merge, one changes; we use nhead1 only}
                    {used to point to the new structure(s) when we rebuild}
   pred,succ: nodeptr;  {used when merging/borrowing to index predecessor/successor}
   temp, prev: nodeptr; {does height need to be decreased? / prev node at prev level}
   pathnodes: listptr;  {keep track of nodes that may need minki or maxki value altered}
   cpn, pn: listptr;    {used to work with the pathnodes list}

   cv: boolean;         {when cv = true, we must change maxki values only}
   count: integer;      {used to count nodes in down subtree}

 Begin {Del_RDSL}
  new(pathnodes);  {dummy header node in pathnodes list}
  pathnodes^.RDSLNODE:= nil;
  pathnodes^.next:= nil;
  cpn:= pathnodes;

  tracer:= head; {start our search at the head node}
  prev:= tracer; {initialize previous to the head node as well}
  While (tracer^.down <> bottom) Do
{B} Begin {While not at leaf level}
     {find the node where we continue our search in the immediate down subtree}
     While (datapoint[dim] > tracer^.maxki) Do
       tracer:= tracer^.right;
     {IF minki and maxki coordinates must be altered, record the node}
     IF ((datapoint[dim] = tracer^.minki) or (datapoint[dim] = tracer^.maxki)) Then
       Begin {record node in pathnodes list}
         new(pn);
         pn^.RDSLNODE:= tracer;
         pn^.next:= nil;
         cpn^.next:= pn;
         cpn:= cpn^.next;
       End  {record node in pathnodes list}
     Else
       pn:= nil;  {minki/maxki coordinates don't have to be altered}
```

```
IF tracer^.down^.right^.right = tracer^.right^.down Then
 Begin {We are about to drop into a gap of size 1.  Fix it!}
  IF ((tracer^.right = tail) or (prev^.right^.down = tracer^.right)) Then
   Begin {we are at the last gap}
    {find predecessor}
    IF prev <> tracer Then
     pred:= prev^.down
    Else
     pred:= prev;

    IF pred <> tracer Then
     Begin {if there is a preceeding gap}
     While pred^.right <> tracer Do
      pred:= pred^.right;
     IF pred^.down^.right^.right = tracer^.down Then
      Begin {the preceeding gap is of size 1 - merge}
       IF pn <> nil Then
        pn^.RDSLnode:= pred;  {Fix up pathnodes list if needed}
       pred^.right:= tracer^.right;  {pred^.right:= tail}
       IF tracer^.nextdim <> lastdim Then
        DisposeOf(tracer^.nextdim);
       dispose  (tracer);
       tracer:= pred;
       tracer^.maxki:= tracer^.down^.right^.right^.right^.maxki;
       IF tracer^.nextdim <> lastdim Then
        MustRebuildRDSL(tracer, nhead1, dim);
      End   {the preceeding gap is of size 1 - merge}
     Else
      Begin {the preceeding gap is of size 2 or 3 - borrow}
       IF pred^.down^.right^.right^.right = tracer^.down Then
        Begin {preceeding gap is of size 2}
         {move tracer back 1!}
         tracer^.down:= pred^.down^.right^.right;
         tracer^.minki:= tracer^.down^.minki;
         pred^.maxki:= pred^.down^.right^.maxki;
         IF tracer^.nextdim <> lastdim Then
            DoubleRebuild(tracer, pred, nhead1, nhead2, dim);
        End   {preceeding gap is of size 2}
       Else
```

```
      Begin {preceeding gap is of size 3}
        {move tracer back 1!}
        tracer^.down:= pred^.down^.right^.right^.right;
        tracer^.minki:= tracer^.down^.minki;
        pred^.maxki:= pred^.down^.right^.right^.maxki;
        IF tracer^.nextdim <> lastdim Then
          DoubleRebuild(tracer, pred, nhead1, nhead2, dim);
      End;  {preceeding gap is of size 3}
    End   {the preceeding gap is of size 2 or 3 - borrow}
  End {if there is a preceeding gap}
Else
 Begin {no preceeding gap -> we may need to decrease height}
   IF tracer^.down^.down <> bottom Then
     Begin {check to see if we need to decrease height}
      temp:= tracer^.down^.down;
      count:= 0;
      While ((temp^.right <> tail) and (count <= order)) Do
        Begin {determine number of nodes at temp's level}
          count:= count + 1;
          temp:= temp^.right;
        End;  {determine number of nodes at temp's level}
      IF count <= order  Then
        Begin {we need to decrease height}
          {we need to decrease height when we have the following situation:
```



the structure we end up with is:




```
          }
        temp:= tracer^.down;
        tracer^.down:= temp^.down;
        IF temp^.nextdim <> lastdim Then
          Begin {dispose of structures in nextdim}
           Disposeof(temp^.nextdim);   Disposeof(temp^.right^.nextdim);
          End;  {dispose of structures in nextdim}
```

```
            dispose(temp^.right); dispose(temp); {dispose of the unneeded level}
           End  {we need to decrease height}
          {Else we need not decrease height.  just move down}
         End   {check to see if we need to decrease height}
        {Else there are only two nodes left.  Move down and delete}
       End   {no preceeding gap -> we may need to decrease height}
     End   {we are at the last gap}
    Else
     Begin {we are not at the last gap}
      succ:= tracer^.right;
      IF succ^.down^.right^.right = succ^.right^.down Then
       Begin {successor gap is of size 1 - merge}
        tracer^.right:= succ^.right;
        tracer^.maxki:= succ^.down^.right^.maxki;
        IF succ^.nextdim <> lastdim Then
         DisposeOf(succ^.nextdim);
        dispose(succ);
        IF tracer^.nextdim <> lastdim Then
         MustRebuildRDSL(tracer, nhead1, dim);
       End   {successor gap is of size 1 - merge}
      Else
       Begin  {successor gap is of size 2 or 3}
        IF succ^.down^.right^.right^.right = succ^.right^.down Then
         Begin {successor gap is of size 2 - borrow}
          succ^.down:= succ^.down^.right;
          succ^.minki:= succ^.down^.minki;
          tracer^.maxki:= tracer^.down^.right^.right^.maxki;
          IF tracer^.nextdim <> lastdim Then
           DoubleRebuild(tracer, succ, nhead1, nhead2, dim);
         End   {successor gap is of size 2 - borrow}
        Else
         Begin {successor gap is of size 3 - borrow}
          succ^.down:= succ^.down^.right;
          succ^.minki:= succ^.down^.minki;
          tracer^.maxki:= tracer^.down^.right^.right^.maxki;
          IF tracer^.nextdim <> lastdim Then
           DoubleRebuild(tracer, succ, nhead1, nhead2, dim);
         End   {successor gap is of size 3 - borrow}
       End   {successor gap is of size 2 or 3}
     End   {we are not at the last gap}
  End; {We are about to drop into a gap of size 1.  Fix it!}

 {now remove the node from the next dimension recursively}
 IF tracer^.nextdim <> lastdim Then
  Del_RDSL(tracer^.nextdim, datapoint, (dim+1));
```

```
      {Update our previous and current node in our search}
      prev:= tracer;
      tracer:= tracer^.down;
{B} End;  {While not at leaf level}

  cv:= true; {assume that if one of minki/maxki has to change, it is maxki}
  IF Equal(tracer^.datapoint, datapoint) Then
   Begin {we are removing the first node in the immediate down subtree}
    tracer^.datapoint:= tracer^.right^.datapoint;
    tracer^.minki:= tracer^.right^.minki;
    tracer^.maxki:= tracer^.right^.maxki;
    temp:= tracer^.right;
    tracer^.right:= temp^.right;
    dispose(temp);
    cv:= false;  {if either of minki/maxki changes, it must be minki}
   End   {we are removing the first node in the immediate down subtree}
  Else
   Begin {we are not removing the first node in the immediate down subtree}
    While (not Equal(tracer^.right^.datapoint, datapoint)) Do
     tracer:= tracer^.right;
    temp:= tracer^.right;
    tracer^.right:= temp^.right;
    dispose(temp);
   End;  {we are not removing the first node in the immediate down subtree}

  IF ((tracer^.right = prev^.right^.down) and (cv)) Then
   Begin {may need to change maxki values of nodes in pathnodes list}
    cpn:= pathnodes^.next;
    While cpn <> nil Do
     Begin {if necessary, change maxki value of cpn and dispose of pathnodes}
      dispose(pathnodes);
      temp:= cpn^.RDSLNODE;
      IF datapoint[dim] = temp^.maxki Then
       temp^.maxki:= tracer^.maxki;
      pathnodes:= cpn;
      cpn:= cpn^.next;
     End;  {if necessary, change maxki value of cpn and dispose of pathnodes}
   End   {may need to change maxki values of nodes in pathnodes list}
  Else
   Begin {must check minki values of nodes in pathnodes list}
    cpn:= pathnodes^.next;
```

```
      While cpn <> nil Do
       Begin {if necessary, change minki value of cpn and dispose of pathnodes}
        dispose(pathnodes);
        temp:= cpn^.RDSLNODE;
        IF datapoint[dim] = temp^.minki Then
         temp^.minki:= prev^.down^.minki;
        pathnodes:= cpn;
        cpn:= cpn^.next;
       End; {if necessary, change maxki value of cpn and dispose of pathnodes}
    end; {must check minki values of nodes in pathnodes list}
 End; {Del_RDSL}

Begin {Delete_RDSL}

 IF ((head^.minki = maxkey) and (head^.maxki = minkey)) Then
  writeln('Error: we are attempting to delete from an empty RDSL!')
 Else {We are deleting from a non-empty DSL}
  IF DPointInRDSL(datapoint, head) Then {we are performing a valid deletion}
   IF (head^.down^.right = tail) Then
    Begin {We are removing the last node from the RDSL}
     Disposeof(head^.nextdim);
     dispose(head^.down);
     InitHeadNode(head); {treat bottom, tail, and lastdim as globals}
    End  {We are removing the last node from the RDSL}
   Else
    Del_RDSL(head, datapoint, dim) {perform a normal deletion}
  Else
   writeln('Error: we are attempting to delete a node not in the RDSL!');

End; {Delete_RDSL}

End. {Implementation - RDSLDEL}
```

RDSLRBLD.PAS {Rebuilding Unit}

```
Unit RDSLRBLD;

Interface

Uses RDSLDECS;

Procedure RebuildRDSL(head, snode: nodeptr; dim: integer);
{This procedure is used to build an RDSL from a static data set.  It is used to accomplish
needed rebuiling that arises from an insertion (split) into or deletion (merge/borrow)
from the RDSL.}

Implementation

Uses RDSLBLD;

Procedure Rebuild(start: nodeptr; dim: integer);
{When we rebuild in a dimension that is not the last dimension, we must also rebuild
structures at all internal nodes in the following dimension.  We use this procedures to
recurse through the current dimension and rebuild the associated structures in the next
dimension.}

{The recursion, which is quite similar to the recursion used by DisposeOf to dispose of
an RDSL and return available memory to the system, is quite difficult and is broken up
over three procedures: the main procedure Rebuild and the two subprocedure
RecurseRight and RecurseDown.}

 Procedure RBuild(start: nodeptr; dim: integer);
 {Called by procedure Rebuild when we find a node at which we need to create/rebuild a
  structure in the next dimension}

 Var nhead: nodeptr;

 Begin {Rbuild}

   NewEmptyRDSL(nhead);
   start^.nextdim:= nhead;
   RebuildRDSL(nhead, start, (dim+1));

 End; {Rbuild}
```

```
Procedure RecurseRight(start: nodeptr; dim: integer);
{This procedure is used to recurse through a level in the RDSL structure.  We go all the
 way to the right on the current level, and as the recursion  unfolds we rebuild necessary
 structures on  the way back to the left.}

Begin {RecurseRight}

  IF start^.right <> tail Then
    RecurseRight(start^.right, dim);
  Rbuild(start, dim);

End;  {RecurseRight}

Procedure RecurseDown(start: nodeptr; dim: integer);
{This procedure is used to traverse the levels in the RDSL structure.  We go all the way
 down to the first level above the leaf level, and  as the recursion unfolds we rebuild
 necessary structures on the way back to the top.}

Begin {RecurseDown}

  IF start^.down^.down <> bottom Then
    RecurseDown(start^.down, dim);
  RecurseRight(start, dim);

End;  {RecurseDown}

Begin {Rebuild}

  IF start^.down^.down <> bottom Then
    RecurseDown(start, dim)
  Else
    IF start^.right <> tail Then
      RecurseRight(start, dim)
    Else
      Rbuild(start, dim);

End;  {Rebuild}

Procedure Disposelist(Var list: listptr);
{We create a linked list of the nodes in sorted order that we will use to create the RDSL
from when building from a static data set.  Once the RDSL is built, we no longer need the
list of nodes so we dispose of it.}

Var temp: listptr;
```

```
Begin {Disposelist}

 While list <> nil Do
  Begin {dispose of next unneeded pointer in list}
   temp:= list^.next;
   dispose(list);
   list:= temp;
  End;  {dispose of next unneeded pointer in list}

End;  {Disposelist}

Procedure GetList(start: nodeptr; Var list: listptr);
{When we are about to rebuild a structure in the next dimension and the nodes we are
going to rebuild the structure with are located in another structure in that dimension, we
must obtain a list of them to work with.  This procedure gets that list.}

Var current, nl: listptr;
```

```
Begin {GetList}

 IF start <> lastdim Then
  Begin {we are not in the last dimension, there is a list to get.}
   {initialize the list and use a dummy header node}
   new(list);
   list^.RDSLNODE:= nil;

   {find the first node to be inserted into the list}
   While start^.down <> bottom Do
    start:= start^.down;

   current:= list;
   While start <> tail Do
    Begin {continue creating the list}
     {get the next node to go in the list}
     new(nl);
     FCopyNode(start, nl^.RDSLNODE);
     {insert it into the list}
     current^.next:= nl;
     current:= nl;
     {go to the next node}
     start:= start^.right;
    End; {continue creating the list}
   current^.next:= nil;
  End {we are not in the last dimension, there is a list to get.}
 Else
  writeln('Error: Getting a list in the last dim.  Procedure GetList');

End; {GetList}

Procedure MergeLists(list1, list2: listptr; Var list3: listptr; dim: integer);
{Before we can create a structure, we must merge the lists of nodes that we have obtained
to build our structure with into one list.  This procedure is used to merge two sorted lists
into one list (in O(n) time).}

Var c1, c2, c3, nl: listptr;
```

```
Begin {MergeLists}

 {initialize the list with dummy header node}
 new(list3);  list3^.RDSLNODE:= nil;

 {initialize our pointers}
 c3:= list3;
 c1:= list1^.next;
 c2:= list2^.next;

 While ((c1 <> nil) and (c2 <> nil)) Do
 {While we have not reached the end of one list we must continue to make comparisons}
  IF c1^.RDSLnode^.datapoint[dim] < c2^.RDSLnode^.datapoint[dim] Then
    Begin {c1 goes next}
      c3^.next:= c1;
      c1:= c1^.next;
      c3:= c3^.next;
    End   {c1 goes next}
  Else {c2 < c1}
    Begin {c2 goes next}
      c3^.next:= c2;
      c2:= c2^.next;
      c3:= c3^.next;
    End;  {c2 goes next}
 {Once we have reached the end of one list, append rest of the other list to sorted list.}
 While (c1 <> nil) Do
  Begin {append the remainder of list 1}
    c3^.next:= c1;
    c1:= c1^.next;
    c3:= c3^.next;
  End;  {append the remainder of list 1}
 While (c2 <> nil) Do
  Begin {append the remainder of list 2}
    c3^.next:= c2;
    c2:= c2^.next;
    c3:= c3^.next;
  End;  {append the remainder of list 2}

c3^.next:= nil;  {mark our list as being completed}

 {dispose of no longer necessary pointers}
 dispose(list1);  dispose(list2);

End;  {MergeLists}
```

```
Procedure GetSortedList(start: nodeptr; Var list3: listptr; dim: integer);
{When rebuild structures in the next dimension at the first level above the leaf node, we
must first sort the data points at the leaf level to get our sorted list.  This procedure
accomplishes that sort to get our sorted list.}

Var snode: nodeptr;    {"sentinel node"; indicates we have just inserted the last node}
  cur, bst, t: listptr;  {used for building and tracing through binary search tree which is
                          built to accomplish sort / t is used to get a copy of a needed node}

{*****************************************************************}
 {We are dealing with a 1-3 DSL so we never have to sort more than 4 nodes.  Therefore,
  sorting by building a BST won't take long. The following procedures build, traverse, &
  destroy a BST to accomplish the sort.}

 Procedure InsertBST(t: listptr; Var bst: listptr);

 Begin {InsertBST}

  IF bst <> nil Then
   IF t^.RDSLnode^.datapoint[dim] < bst^.RDSLnode^.datapoint[dim] Then
    InsertBST(t, bst^.prev)
   Else
    InsertBST(t, bst^.next)
  Else
   Begin {not inserting first node into bst}
    new(bst);
    bst:= t;
    bst^.prev:= nil;
    bst^.next:= nil;
   End; {not inserting first node into bst}

 End;  {InsertBST}
```

```
Procedure InorderBST(bst: listptr; Var cur: listptr; dim: integer);

Var t: listptr;

Begin {InorderBST}

 IF bst <> nil Then
  Begin {if there is a node, continue}
   InorderBST(bst^.prev, cur, dim);
    new(t);
    t^.RDSLNODE:= bst^.RDSLNODE;
    cur^.next:= t;
    cur:= t;
   InorderBST(bst^.next, cur, dim);
  End; {if there is a node, continue}

End; {InorderBST}

Procedure PostOrderBST(Var bst: listptr);

Begin {PostOrderBST}

 IF bst <> nil Then
  Begin {continue to dispose of no longer needed tree}
   PostOrderBST(bst^.prev);
   PostOrderBST(bst^.next);
   dispose(bst); bst:= nil;
  End;  {continue to dispose of no longer needed tree}

End; {PostOrderBST}

{We are dealing with a 1-3 DSL so we won't have to sort more than 4 nodes. Therefore,
 sorting by building a BST won't take long.  The above procedures build, traverse, and
 destroy a BST to  accomplish the sort.}

{*************************************************************************}
```

```
Begin {GetSortedList}

 {initialize the list with a dummy header node}
 new(list3);  list3^.RDSLnode:= nil;

 snode:= start^.right^.down;
 start:= start^.down;

 {perform the sort by building the BST and performing an inorder traversal}
 bst:= nil;
 While start <> snode Do
  Begin {must get another node}
    new(t);
    copyNode(start, t^.RDSLNODE, dim);
    InsertBST(t, bst);
    start:= start^.right;
  End;  {must get another node}

 {obtain the sorted list}
 cur:= list3;
 InorderBST(bst, cur, dim);
 cur^.next:= nil;

 {dispose of the tree}
 PostOrderBST(bst);

End;  {GetSortedList}

Procedure LBuildRDSL(head: nodeptr; list: listptr;  dim: integer);
{This is the main procedure called by RebuildRDSL.  It is the procedure that actually
builds the RDSL from the sorted static dataset, provided by list.}

 Procedure GetNLNode(Var t: nodeptr; n1, n2: nodeptr);
 {This procedure is used by LBuildRDSL to create a non-leaf node.}

 Begin {GetNLNode}

   new(t);
   t^.datapoint:= sv;
   t^.nextdim:= lastdim;
   t^.minki:= n1^.minki;
   t^.maxki:= n2^.maxki;

 End;  {GetNLNode}
```

```
Procedure GetPcur(Var pcur: nodeptr);
{Procedure is used by LBuildRDSL to find the last node in an immediate down subtree.}

Var i: byte;

Begin {GetPcur}

 Case mxnpl of
 2,3: Begin
      i:= 0;
      While ((i < (mxnpl - 1)) and (pcur^.right^.right <> tail)) Do
       Begin {go right}
        i:= i + 1;
        pcur:= pcur^.right;
       End;  {go right}
      IF pcur^.right^.right = tail Then
       pcur:= pcur^.right;
     End;
  4: Begin
      i:= 0;
      While ((i < (mxnpl - 1)) and (pcur^.right^.right^.right <> tail)) Do
       Begin {go right}
        i:= i + 1;
        pcur:= pcur^.right;
       End;  {go right}
     End;
 End; {Case mxnpl}

 End;  {GetPcur}

Var ph,lh,pcur,cur,t: nodeptr; {pointers to first node on prev level & level, current node
on
                               previous level & level, and node we are inserting}
   npl,pnpl: integer;          {number of nodes on current level and on previous level}
   cl: listptr;                {next node to be inserted into bottom level}
   sstop: boolean;             {short-stop - insures that we don't have a head node
                               with only one node in its immediate down subtree}
   morelevels: boolean;        {do we build another level before we add head node?}

Begin {LBuildRDSL}

 {build the RDSL}

 npl:= 0;              {no nodes on bottom level yet}
 cl:= list^.next;      {we must first get first node in list}
```

```
{build the bottom first}
npl:= npl + 1;          {as we add a node to the bottom, increment our counter}
t:= cl^.RDSLnode;       {add the node to the bottom}
cur:= t;                {keep track of the last node added}
cl:= cl^.next;          {what is the next node to be added?}
lh:= t;                 {the first node on the bottom level}

While (cl <> nil) Do  {while more nodes to get}
 Begin {get next node}
  npl:= npl + 1;        {increment the counter of the number of nodes}
  t:= cl^.RDSLnode; {add the next node to the bottom}
  cur^.right:= t;       {make the proper pointer change}
  cur:= t;              {keep track of the last node added}
  cl:= cl^.next;        {what is the next node to be added?}
 End;  {get next node}

cur^.right:= tail;    {the last node points to the tail}

{now build the remaining levels}
sstop:= false;
morelevels:= false;

IF ((mxnpl = 2) or (mxnpl = 3)) Then
 Begin {are there more levels?}
  IF npl > mxnpl + 1 Then
   morelevels:= true
 End    {are there more levels?}
Else
 IF npl > mxnpl Then
  morelevels:= true;

IF morelevels Then
 Repeat
  pnpl:= npl;
  ph:= lh;
  pcur:= ph;

  npl:= 0;
  GetPcur(pcur);
```

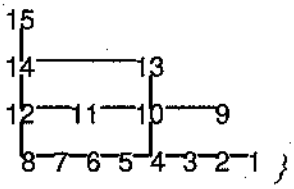
```
    GetNLnode(t, ph, pcur);
    t^.down:= ph;
    npl:= npl + 1;
    cur:= t;
    pcur:= pcur^.right;
    lh:= t;

    While (pcur <> tail) Do
     Begin {continue adding nodes to current level}
      ph:= pcur;
      GetPcur(pcur);

      GetNLnode(t,ph,pcur);
      t^.down:= ph;
      npl:= npl + 1;
      cur^.right:= t;
      cur:= t;
      pcur:= pcur^.right;
     End; {continue adding nodes to current level}

    cur^.right:= tail; {we have just added the last node of the level}

    IF ((mxnpl = 2) or (mxnpl = 3)) Then
     IF npl = mxnpl + 1 Then
      sstop:= true;
      {Otherwise we have a head node with only one immediate descendent}
   Until ((npl <= mxnpl) or (sstop));

  {now stick the head on}
  head^.down:= lh;
  head^.minki:= head^.down^.minki;
   cur:= head^.down;
   While (cur^.right <> tail) Do
    cur:= cur^.right;
  head^.maxki:= cur^.maxki;

  {if not in the last dimension, then we must rebuild RDSL's in the nextdim}
  IF dim <> dimensions Then
   Rebuild(head, dim); {we know the DSL that we just built isn't empty}

End; {LBuildRDSL}
```

```
Procedure RebuildRDSL;
{This procedure builds an RDSL from a static data set. It is called by procedure
GRRDSL in the insert unit, which is called by Insert_RDSL and MustRebuildRDSL
(called by Insert_RDSL and DoubleRebuild [called by Del_RDSL]).}

Var list1, list2, list3, list4, list5, list6, list7: listptr;     {used to build list of nodes}

Begin {RebuildRDSL}

IF snode^.down^.down <> bottom Then
 Begin {we can rebuild by merging the two (nextdim)RDSL's at the next level}
  {Get the nodes that will create the new RDSL stored in two sorted lists}
  GetList(snode^.down^.nextdim, list1);  GetList(snode^.down^.right^.nextdim, list2);
  {merge the lists into one sorted list}
  MergeLists(list1, list2, list3, dim);
  IF snode^.down^.right^.right <> snode^.right^.down Then
   Begin
    {IF s^.d^.r^.r = tail then snode^.r^.d = tail so list4 is not empty}
    GetList(snode^.down^.right^.right^.nextdim, list4);
    IF snode^.down^.right^.right^.right <> snode^.right^.down Then
     {if s^.d^.r^.r^.r = tail --> snode^.r^.d = tail --> list5 not empty}
      GetList(snode^.down^.right^.right^.right^.nextdim, list5)
    Else
     Begin {list5 is empty}
      new(list5);  list5^.RDSLNODE:= nil;  list5^.next:= nil;
     End;  {list5 is empty}
    Mergelists(list4, list5, list6, dim); Mergelists(list3, list6, list7, dim);
    list3:= list7;
   End;
  {build the RDSL from the bottom up with the essentially "static" dataset}
  LBuildRDSL(head, list3, dim);
 End   {we can rebuild by merging the two (nextdim)RDSL's at the next level}
Else
 Begin {we are one level above the leaf level and must rebuild from scratch}
  {get the nodes we will use and put them into a sorted list}
  GetSortedList(snode, list3, dim);
  {build the RDSL from the bottom up with the essentially "static" dataset}
  LBuildRDSL(head, list3, dim);
 End;  {we are one level above the leaf level and must rebuild from scratch}
Disposelist(list3); {Dispose of a no longer needed list3}

End;  {RebuildRDSL}

End. {Implementation - RDSLRBLD}
```

RDSLDMEM.PAS {Disposal Unit}

```
Unit RDSLDMEM;

Interface

Uses RDSLDECS;

Procedure DisposeOf(Var start: nodeptr);
{This procedure disposes of an RDSL structure in the next dimension before we rebuild it
to return all available memory to the system.}

Implementation

Procedure DisposeOf;
{This procedure is used to dispose of an RDSL in the next dimension when we find that
rebuilding is necessary.}

{We find that the recursion is quite complicated and that we need to break it up into three
procedures; one main procedure and two sub. procs.}

Procedure Axe(Var start, prev: nodeptr);
{This procedure is used to take us across a level.  We go to the end of the level and then
as the recursion unfolds we dispose of the level right to left.}

Begin {Axe}

 IF start^.right <> tail Then
   Axe(start^.right, start);       {continue to go right}
 IF start^.nextdim <> lastdim Then
   DisposeOf(start^.nextdim); {dispose of next dimension before current dimension}
 dispose(start);
 prev^.right:= tail;

End; {Axe}
```

```
Procedure Remove(Var start, prev: nodeptr);
{This procedure is used to take us down the structure.  We go to the bottom of the
structure and then dispose of the structure bottom up; leaf level back up to head node.}

Begin {Remove}

 IF start^.down <> bottom Then
  Remove(start^.down, start)          {continue to go down}
 Else
  Begin {removing last (bottom) level}
   IF start^.right <> tail Then
    Axe(start^.right, start);  {remove level, last node to first node}
   IF start^.nextdim <> lastdim Then
    DisposeOf(start^.nextdim);       {dispose of a structure in next dimension first}
   dispose(start);
   prev^.down:= bottom;
  End;  {removing last (bottom) level}

End;  {Remove}

Begin {DisposeOf}
{We check for special cases and call the procedures Axe and Remove as necessary}

{The following diagram illustrates the order in which the nodes in a simple
 RDSL structure would be disposed of through the given recursion:
```




```
}
```

```
IF ((start = tail) or (start = bottom)) Then
  writeln('Serious Error')
Else
 IF start^.down <> bottom Then
  IF start^.down^.down <> bottom Then
   Remove(start^.down, start)        {we can recurse down}
  Else
   Begin
    IF start^.down^.right <> tail Then
     Axe(start^.down^.right, start^.down); {we can recurse right}
    IF start^.down^.nextdim <> lastdim Then
     DisposeOf(start^.down^.nextdim);     {dispose of next dimension before current
one}
    dispose(start^.down);
    IF start^.nextdim <> lastdim Then
     DisposeOf(start^.nextdim);           {dispose of next dimension before current
one}
    dispose(start);
   End
 Else
  Begin {only one node to remove}
   IF start^.nextdim <> lastdim Then
    DisposeOf(start^.nextdim);
   dispose(start);
  End; {only one node to remove}

End; {DisposeOf}

End. {Implementation - RDSLDMEM}
```

RDSLDBUG.PAS {Debugging Unit}

```
Unit RDSLDBUG;
{This unit contains code that  used for debugging purposes.  It contains procedures
 which output the RDSL structure to a file so one may keep track of what the structure
 "looks" like.}

Interface

Uses RDSLDECS;

Procedure DisplayRDSL(head: nodeptr);
{Outputs the RDSL structure to a datafile.  It works but the format of the datafile
produced doesn't make the structure easily visualizable.  It is useful for determining the
order in which the nodes of the structure are visited when the obvious recursion is used.}

Procedure DisplayRDSLII(head: nodeptr);
{Outputs the RDSL structure to a datafile.  The format of the datafile is quite clearer
than that produced by the DisplayRDSL procedure and makes the structure much more
visualizable.}

Implementation

{*************************************************************************}
{The following procedures used to output the RDSL.  They are for debugging purposes.}

Procedure DisplayRDSL;

Var ofile: text;

 Procedure DRDSL(head: nodeptr);

 Begin {DRDSL}

  IF head^.nextdim <> lastdim Then
   Begin      {output next dimension before this one}
    writeln(ofile, 'nextdim');
    DRDSL(head^.nextdim);
   End;       {output next dimension before this one}

  IF head^.right <> tail Then
   Begin      {go right}
    writeln(ofile, 'right');
    DRDSL(head^.right);
   End;       {go right}
```

```
  IF head^.down <> bottom Then
   Begin       {then go down}
    writeln(ofile, 'down');
    DRDSL(head^.down);
   End;        {then go down}

  With head^ Do
   Begin       {then output node}
    write  (ofile, 'M vals: ', minki:4, maxki: 4, 'Datapoint: ');
    writeln(ofile, datapoint[1]:4, datapoint[2]:4, datapoint[3]:4);
   End;        {then output node}

 End; {DRDSL}

Begin {DisplayRDSL}

 assign (ofile, 'RDSL.dat');
 rewrite(ofile);
  DRDSL(head);
 close  (ofile);

End; {DisplayRDSL}

Procedure DisplayRDSLII;

Var ofile2: text;

 Procedure DRDSLII(head: nodeptr);

 Var lh, current: nodeptr;      {used to trace through the structure in an "orderly" fashion}

 Begin {DRDSL2}

  lh:= head;

  While lh <> bottom Do
   Begin {go down}
    writeln(ofile2, 'Newlevel');
    current:= lh;
    While current <> tail Do
```

```
      Begin {go right}
       With current^ Do
         Begin {output node's data}
           write(ofile2, 'M vals: ', minki:4, maxki: 4);
           writeln(ofile2, 'Datapoint: ', datapoint[1]:4, datapoint[2]:4, datapoint[3]:4);
         End;  {output node's data}
       current:= current^.right;
      End;  {go right}
    lh:= lh^.down;
   End;  {go down}

  lh:= head;

  IF head^.nextdim <> lastdim Then
   Begin {output structure in nextdim}
    While lh^.down <> bottom Do
     Begin {go down}
      current:= lh;
      While current <> tail Do
       Begin  {go right}
        writeln(ofile2, 'Nextdim');
        DRDSLII(current^.nextdim);
        current:= current^.right;
       End;  {go right}
      lh:= lh^.down;
     End; {go down}
   End;  {output structure in nextdim}

 End;  {DRDSL2}

Begin {DisplayRDSLII}

 assign (ofile2, 'RDSLII.dat');
 rewrite(ofile2);
  DRDSLII(head);
 close  (ofile2);

End;  {DisplayRDSLII}

{ The above procedures are used to output the RDSL.  They are for debugging purposes.}
{*****************************************************************************}

End.  {Implementation - RDSLDBUG}
```

RDSLOUT.PAS {Reporting Unit}

```
Unit RDSLOUT;

Interface

Uses RDSLDECS;

Procedure ReportPoint(datapoint: coordinate);
{This procedure is used to report a point found by the range search procedure
KDRSEARCH.}

Implementation

Procedure ReportPoint;
{Report the point found by KDRSEARCH (unit RDSLSRCH)}

Begin {ReportPoint}

 write(sfile, datapoint);

End; {ReportPoint}

End. {Implementation - RDSLOUT}
```

RDSLSTST.PAS {Drivers To Test Range Search }

```
Unit RDSLSTST;

Interface

Uses RDSLDECS;

Procedure Search(head: nodeptr);
{This procedure contains the driver code that tests the search procedure, KDRSEARCH,
which is located in unit RDSLSRCH.}

Implementation

Uses RDSLSRCH;

Procedure Search;
{This procedure contains the driver code that tests the search procedure,  KDRSEARCH,
which is located  in unit RDSLSRCH.}

Var next: boolean;

 Procedure GetLAR;
 {this procedure initializes L and R for our search}

 Const LRF = 'lr.dat';

 Var LRfile: dfiletype;

 Begin {GetL&R}

assign (LRfile, LRF);
reset  (LRfile);
  read(LRfile, L);
  read(LRfile, R);
close  (LRfile);

 End;  {GetL&R}
```

```
Begin {Search}

 GetLAR;
 next:= false;

 assign (sfile, searchfile);
 rewrite(sfile);
   KDRSearch(head, head, next, 1);
 close  (sfile);

End;  {Search}

End.  {Implementation - RDSLSTST}
```

RDSLITST.PAS {Drivers To Test Insertion}

```
Unit RDSLITST;

Interface

Uses RDSLDECS;

Procedure BuildRDSL(Var head, tail, bottom, lastdim: nodeptr);
{This is a driver procedure to test the insertion procedure.  It builds a RDSL structure by
inserting one datapoint at a time until we have inserted numpoints datapoints.}

Implementation

Uses RDSLBLD, RDSLINS, RDSLDBUG;

Procedure BuildRDSL;

Var dpoint: coordinate; {the datapoint we are about to insert}
   current: nodeptr;    {the new node we are about to insert}
   i: integer;          {loop control variable}
   dim: integer;        {current dimension that we are working in}
   dfile: dfiletype;    {the input datafile}
```

```
Begin {BuildRDSL}

 CreateEmptyRDSL(head, tail, bottom, lastdim);

 assign(dfile, dpointsfile);
 reset (dfile);

 numpoints:= -1;
 dim:= 1;  numpoints:= numpoints + 1;
 While not eof(dfile) Do
  Begin {get and insert next datapoint}
   numpoints:= numpoints + 1;
   read(dfile, dpoint);
   new(current);
    current^.datapoint:= dpoint;
    current^.minki:= current^.datapoint[dim];
    current^.maxki:= current^.datapoint[dim];
    current^.nextdim:= lastdim;
    current^.down:= bottom;
   Insert_RDSL(head, current, dim);
   writeln('Insertion: ', i);
   DisplayRDSLII(head);  {debugging}
  End; {get and insert next datapoint}

 close (dfile);

End;  {BuildRDSL}

End.  {Implementation - RDSLITST}
```

RDSLDTST.PAS {Drivers To Test Deletion}

```
Unit RDSLDTST;

Interface

Uses RDSLDECS;

Procedure DestroyRDSL(Var head: nodeptr);
{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.}

Implementation

Uses RDSLDEL, RDSLDBUG;

Procedure DestroyRDSL;

Var dpoint: coordinate;
   i: integer;          {loop control variable}
   dfile: dfiletype;    {the input datafile - contains datapoints to be removed from the
RDSL}

Begin {DestroyRDSL}

 assign(dfile, dpointsfile);
 reset (dfile);

 For i:= 1 to numpoints Do
  Begin {obtain and delete the next datapoint}
   read(dfile, dpoint);
   Delete_RDSL(head, dpoint, 1);
   writeln('Deletion: ', i);
   DisplayRDSLII(head);  {output the structure for debugging purposes}
  End;  {obtain and delete the next datapoint}

 close  (dfile);

End;  {DestroyRDSL}

End.  {Implementation - RDSLDTST}
```

APPENDIX 2

On The Number of Nodes in Dimension k

In section 3.3.3.1 on simple insertion in a k-d Range DSL we stated in Lemma 1 that the number of nodes, and therefore the number of points, in dimension k is $O(n \lg^{k-1} n)$ and that, upon careful investigation, we find that the total number of points (in a worst case RDSL) in dimension k is precisely given by:

$$[\sum_{x=0}^{\lg n-1}(x + k - 2)C(k - 2)] \times n$$

(3)

We shall now illustrate why this is so. Consider Figure 11. We see that (with a worst case DSL structure in dimension 1) we have 1 structure of n points in dimension 2 (structure A), 2 structures of $n/2$ points in dimension 2 (structures B & C), 4 structures of $n/4$ points in dimension 2 (structures D, E, F, & G), 8 structures of $n/8$ points in dimension 2 (structures H through O), etc., and that the number of structures of $n/2^x$ points is given by 2^x. Since a DSL structure of n points has at most $2n$ - 1 nodes, it matters little whether we work with points or nodes as the Big-Oh analysis will be the same. For simplicity we will work with points.

Now we will consider the number of points in dimension 3. For structure A in dimension 2 which consists of n points we have 1 structure of n points (A1) in dimension 3, 2 structures of $n/2$ points (A2 and A3), 4 structures of $n/4$ points (A4 through A7), 8 structures of $n/8$ points (A8 through A15), etc.; for the 2 structures (B & C) in dimension 2 of $n/2$ points we have 1 structure of $n/2$ points (B1/C1) in dimension 3, 2 structures of $n/4$ points (B2/C2 & B3/C3), 4 structures of $n/8$ points (B4/C4 through B7/C7), etc., for each of the structures; for the 4 structures (D, E, F, &G) of $n/4$ points in dimension 2 we have 1 structure of $n/4$ points (D1, E1, F1, & G1) in dimension 3, 2 structures of $n/8$ points (D2 & D3, E2 & E3, F2 & F3, G2 & G3), etc., for each of the structures; and for the 8 structures (H,I,J,K,L,M,N,O) of $n/8$ points in dimension 2 we have 1 structure of $n/8$ points each (H1 through O1), etc., in dimension 3. This gives us 1 structure of n points, 4 structures of $n/2$ points, 12 structures of $n/4$ points, and 32 structures of $n/8$ points, etc., for dimension 3.

Moving on to dimension 4 we find that, after looking at all the structures attached to
A1 - A15, B1/C1 - B7/C7, D1/E1/F1/G1 - D3/E3/F3/G3, and
H1/I1/J1/K1/L1/M1/N1/O1, we have 1 structure of n points, 6 structures of $n/2$ points, 24 structures of $n/4$ points, and 80 structures of $n/8$ points, etc.

The data can be summed up nicely in the following table:

Dimension	n/1 points	n/2 points	n/4 points	n/8 points	n/16 points
1	1				
2	1	2	4	8	16
3	1	4	12	32	80
4	1	6	24	80	240
5	1	8	40	160	560

and restated in the following format

Dimension	n/1 points	n/2 points	n/4 points	n/8 points	n/16 points
1	1				
2	1	2	4	8	16
3	1*1	2*2	3*4	4*8	5*16
4	1*1	3*2	6*4	10*8	15*16
5	1*1	4*2	10*4	20*8	35*16

which makes it clear that for dimension k, k > 2, we have ${}_{(x+k-2)}C_{(k-2)} \times 2^x$ structures of $n/2^x$ points.

Since x can vary from 1 to lg n - 1 (we must have at least 2 datapoints in a structure in a dimension other than 1 and we can have at most n datapoints in a structure in any dimension), this gives us

$$2n \times \sum_{x=0}^{\lg n - 1} (x + k - 2) C (k - 2)$$

nodes in dimension k (since we have at most $2n$ - 1 nodes for n datapoints). Since 2 is a constant factor, we ignore it for our Big Oh analysis.

We saw in section 3.3.3.1 in the proof of Lemma 1 (from [Grah94]) that this expression is approximated by $n \lg^{k-1} n$ which gives the number of nodes in dimension k as $O(n \lg^{k-1} n)$ as desired.

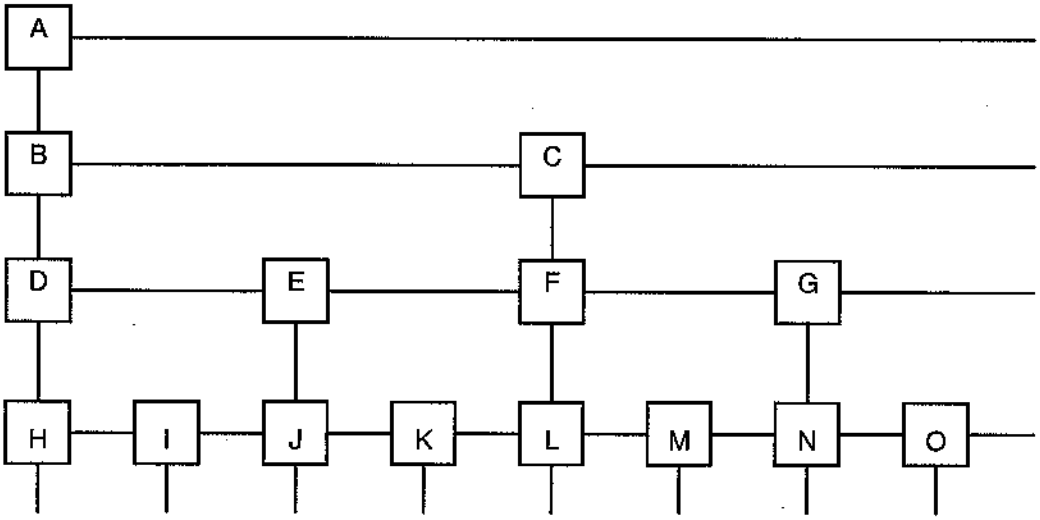


Figure 10 a) Dimension 1 of a k-d Range DSL

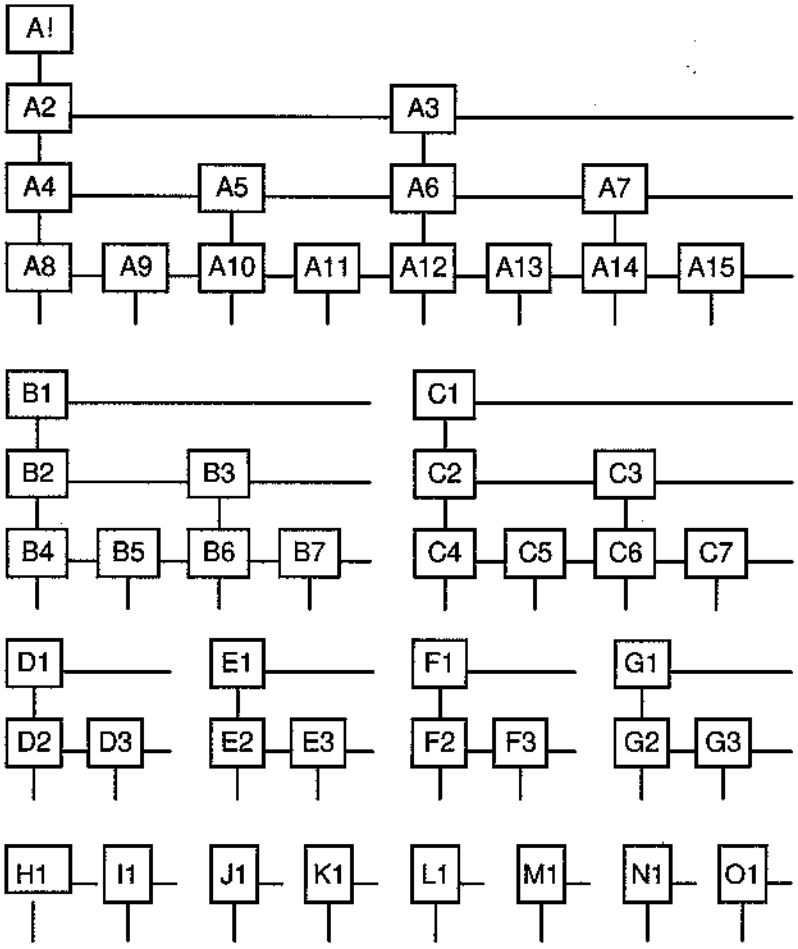


Figure 10b) Dimension 2 of a k-d Range DSL.

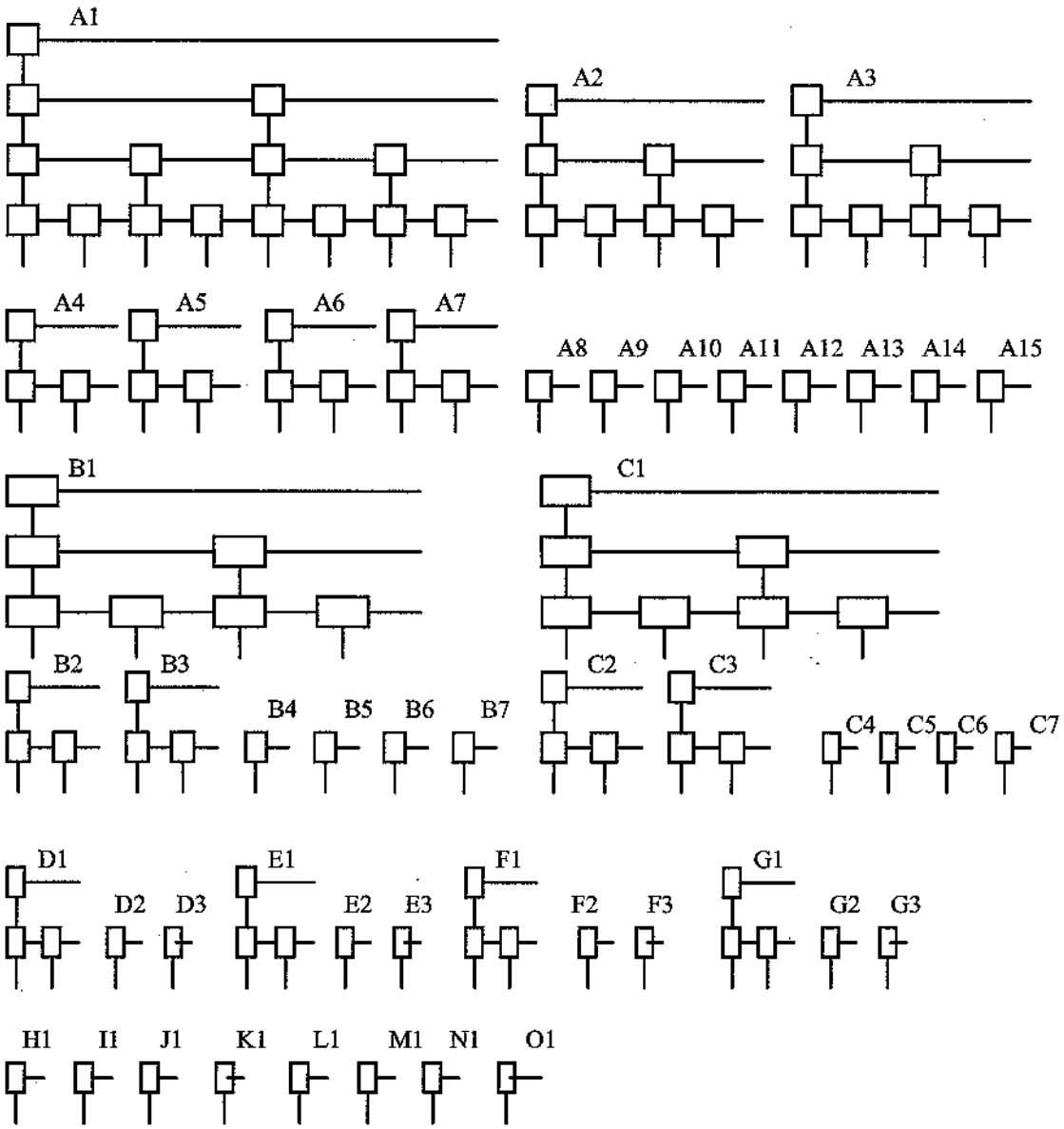


Figure 10c) Dimension 3 of a k-d Range DSL.