# INVESTIGATION OF A DYNAMIC K-D SEARCH SKIP LIST REQUIRING $\Theta(kn)$ SPACE

by

**Yunlan Pan**

**TR97-116, September 1997**

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

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

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

# INVESTIGATION OF A DYNAMIC K-D SEARCH SKIP LIST REQUIRING Θ(kn) SPACE

by

Yunlan Pan

Bsc(Math) Zhejing Normal University, China, 1989

A Thesis Submitted in Partial Fulfilment of
The Requirements for the Degree of

**Master of Computer Science**
in the Graduate Academic Unit of Computer Science

| | |
|---|---|
| Supervisor: | Nickerson, B. G., BScE, MscE (UNB), PhD(RPI), CS |
| Examining Board: | Horton, J. D., Bsc(Manit), MA(York), PhD(Wat), CS, chair |
| | Spencer, E. B., Bsc(Dal), Mmath, PhD(Wat), CS |
| External Reader: | Small, R. D., BASc(Toronto, Msc, PhD(CIT), Math |

This Thesis is Accepted

_____
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September 1997

# ABSTRACT

A new dynamic k-d data structure (supporting insertion and deletion) labeled the k-d Search Skip List is developed and analyzed. The time for k-d semi-infinite range search on n k-d data points without space restriction is shown to be $\Omega(k\log n)$ and $O(kn + kt)$, and there is a space-time trade-off represented as
$(\log T' + \log\log n) = \Omega(n(\log n)^{k-\theta})$, where $\theta = 1$ for $k = 2$, $\theta = 2$ for $k \geq 3$, and $T'$ is the scaled query time. The dynamic update time for the k-d Search Skip List is $O(k\log n)$.

We consider the problem of reporting all points in a positive half-space $h^*$ (bounded by a query hyper plane h) from a set F of n points in k-dimensional space $E^k$; that is, find and report $S \cap h^*$. Previous dynamic solutions (supporting insertion and deletion of points) are simplified, and the previously best known dynamic update time is improved. For $t = |S \cap h^*|$, we show that with $O(kn\log n)$ preprocessing time and $\Theta(kn)$ space, the k-d half-space searching problem can be solved in $O(kn)$ time with dynamic update time of $O(k\log n)$.

Experimentally for a random data set of 100,000 points, an average of 1048 milliseconds was required to report 50,000 points in range for a 10-d orthogonal range search; about 800 milliseconds was required to report 50,000 points in range for a 10-d semi-infinite range search; and 106 milliseconds was required to report 50,000 in range for a 10-d half-space range search.

# ACKNOWLEDGMENTS

First I would like to express my sincere thanks and appreciation to my supervisor, Dr. Bradford G. Nickerson, for his suggestion of this specific topic, invaluable guidance and encouragement concerning my academic and nonacademic endeavours. Through weekly discussions, his efforts made it possible to form a solid basis for my thesis, and for this I am grateful.

I would like to thank the readers, Dr. Joseph Horton, Dr. B. Spencer, and Dr. D.Small, whose valuable comments improved the overall presentation of the work. I thank Michael G. Lamoureux for his valuable comments and insightful advice.

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

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# Chapter 1
# INTRODUCTION

Multi-attribute data can be referred to as multi-dimensional (or k-d) data as each attribute can be viewed as a separate dimension of a k-dimensional space. The capability to store, retrieve and search for k-d data is a fundamental component of many computer information systems. The problem of k-d range search is important in itself (having applications in such areas as computer graphics, database management systems, computational geometry, pattern recognition, statistics, design automation, and geographic information systems). In addition, k-d range search serves as a representative of the entire class of multi-attribute searching problems.

The problem is not as obvious as it may first seem. The collection of records may be preprocessed to achieve a balance between the storage utilized and the time required to answer a query. There is an extensive literature (e.g. Bentley and Fredman [1979]) on algorithms for range query, and the space and time requirements have traditionally been used as performance measures for such algorithms. We also know there is a (storage) space - (retrieval) time tradeoff for orthogonal range search on a static database [Vaidya, 1989]. Today's applications require a dynamically updatable database and thus any index structure for the database must also allow fast dynamic updates to ensure feasibility. These interdependent requirements imply that the design of a dynamic data structure for efficient k-d range search is difficult and often unintuitive.

Upper bounds on the worst case search time, update operations, and storage

requirement are important in order to know that a structure is feasible for use. A storage requirement of $O(n^2)$ is excessive for large n, where n is the number of data points in the structure, or there may not be enough storage space for the structure when the data set is large. The k-d range search time must be O(kn), otherwise the structure provides no improvement over an unstructured brute force search. The same holds true for the time required for dynamic updates; otherwise the structure would have to be rebuilt with each insertion and deletion and such a structure is not considered dynamic.

Implementable structures do exist which provide for efficient k-d range search in a dynamic environment. These include dynamizations of the k-d range tree of Bentley[1980a] and the k-d Range Deterministic Skip List (k-d Range DSL) of Lamoureux[1996]. Although efficient, they do not meet the lower bound of $\Omega(k \log n + t)$ for range search provided by the decision tree model.

## 1.1 Definitions

*Range searching* is an old-age problem in computer science. We can phrase *range searching* in geometric terms; i.e. given a set F of n points in k-d space, we preprocess these points into a data structure. After we have preprocessed the points we must answer *queries* which ask for all points x of F such that the first coordinate of $x(x_1)$ is in some range $[L_1: H_1]$, the second coordinate $x_2 \in [L_2: H_2]$, ..., and $x_k \in [L_k: H_k]$. We also can phrase this problem in the terminology of data bases; i.e. given a file F of n records, each having k keys, process this file into a database that enables storage, retrieval and range search on the

records. We must then answer queries (normally phrased using Structured Query Language (SQL)) asking for all records such that the first key is in some specified range, the second key in a second range, etc. Boolean combinations of range queries over different attributes are called orthogonal range queries. A range search is performed to retrieve the records specified in the query. The intersection of the query ranges is a k-dimensional hyperrectangle in the space (that is, a "box"), and a range query calls for finding all points lying inside this hyperrectangle.

To motivate this problem, the classical scenario is to regard the points of S as the employees of a certain company. The coordinates of the points are important attributes of the emplyees, such as seniority, age, salary, traveling assignments, and legal status. Crucial to the proper functioning of the company is the ability to provide fast answers to questions of the form: *"Which senior-level employees are less than x years of age, make more than y dollars a year, have been abroad in the last z months, and have never been convicted of a Criminal Code offense?"*. We will often cast range searching in this geometric framework as an aid to intuition.

*Semi-infinite range search* asks for all records with key values each within either the ($[L_1: H_1]$, $[L_2: \infty)$, ..., $[L_k: \infty)$ ), or the ($[L_1: H_1]$, $( -\infty: L_2]$, ..., $(- \infty: L_k]$ ) mode. We call these the "Up semi-infinite range search" and "Down semi-infinite range search", respectively. When the conjunction of k semi-infinite range queries on different attributes is required, we can view each separate attribute as one dimension of a k-dimensional space, and the semi-infinite range search corresponds to asking for all records (data points) that lie within a k-dimensional (hyper) rectangular box with (k-1) line segments as boundaries of the data

space.

For obvious reasons, this is called range searching in report-mode, or more simply, *range reporting.* To put this problem in perspective, *range reporting* refers to the task of reporting all the points that fall in the searching region. Evaluating the number of the points falling into the search region is called *range counting*. Unless noted otherwise, *range reporting* is assumed. This implies that many of our $\Omega$, O, and $\Theta$ analyses of the range search cost functions will have an extra term of t which captures the time needed to report the t records in range.

Five functions normally portray the cost for range searching on a specific data structure G that supports range search and dynamic operations on F. They 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,

as defined in Bentley [1979]. In addition, we also 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 by

I(n, k) = time required to insert a new record in G,

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

For $k \geq 1$, let $E^k$ denote k-dimensional *Euclidean space*. We use h* to denote one of the two open halfspaces bounded by a hyperplane h in $E^k$. An open halfspace h* is called *positive* if either h is vertical or h* is the open half-space above h, i.e., h* intersects the positive vertical axis in a half-line [Haussler, 1987]. A halfspace range search is defined as:

given a set F of n points , find and report the points belonging to the form F∩R, where R is an open halfspace. For example, given a set F of n points in $E^2$ and a query hyperplane h, the halfspace range search problem asks for the retrieval of all points of F on a chosen side of h (see Figure 1.1).

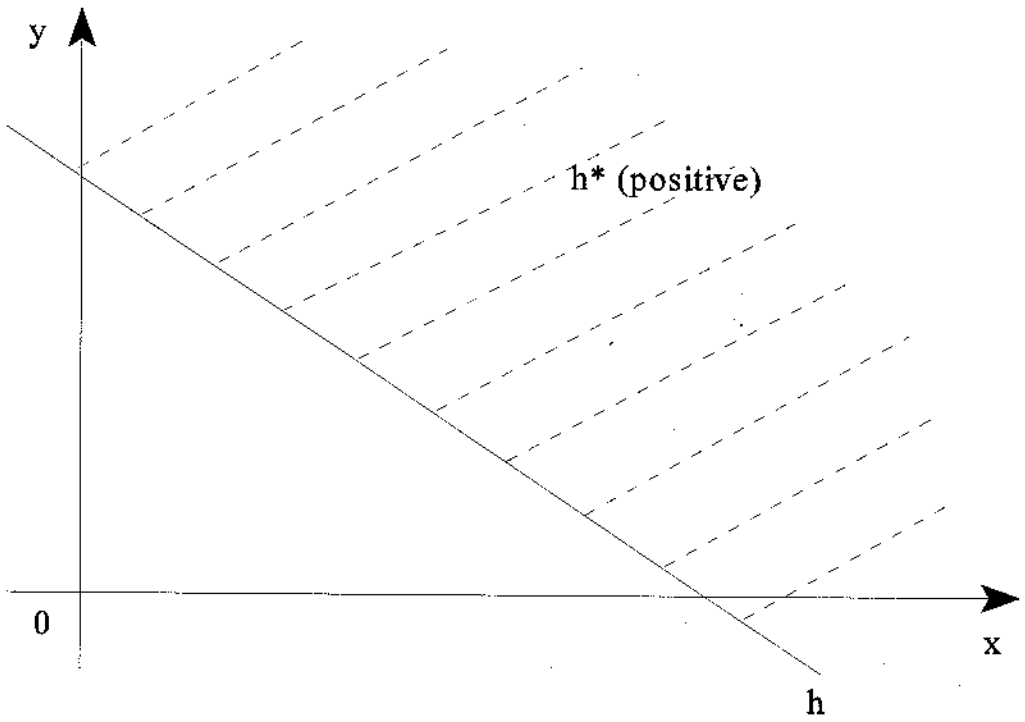


Figure 1.1 In $E^2$, the line h and the positive halfspace h*.

## 1.2 Objectives

The objectives of this thesis are stated as follows:

For a set F of n points in k dimensions,

1. review and analyze data structures that permit k-dimensional range search;

2. develop an algorithm to establish an efficient data structure for k-d range search and determine the asymptotic running times in the worst case;
3. determine the worst case for the semi-infinite range search using a k-d Search Skip List;
4. determine the lower bound for semi-infinite range search of k-d data points, either with or without restriction on the space required for the data structure;
5. determine if the k-d Search Skip List can be adapted to half-space range search;
6. perform an experimental verification of the results with large (e.g. $n = 10^5$) randomly generated data sets.

In the pages that follow, each of the objectives is addressed and an attempt is made to summarize and unite much of the relevant work in this area. The asymptotic running times in the worst case for the k-d Search Skip List are determined, and it is shown that the k-d Search Skip List meets the lower bounds for semi-infinite range search. Also, the k-d Search Skip List is adapted to half-space range search. An experimental analysis of the k-d Search Skip List was performed and the results are illustrated and compared in Chapter 6.

# Chapter 2

# DATA STRUCTURES FOR POINT DATA RANGE SEARCH

This thesis focuses on data structures for orthogonal range queries as defined by Knuth [1973], semi-infinite range search as defined by McCreight [1985] and half-space range search as defined by Chazelle and Preparata [1986]. Data structures and algorithms that support other types of range queries can be found in the excellent survey paper of Matousek [1994] which illustrates many problems which are of great importance in the field of computational geometry.

## 2.1 1-Dimensional Data Structure

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

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

One may also use a multi-way height balanced search tree known as the B-tree, of which a good overview may be found in Comer [1979], which allows for a worst case range search in time $\Theta(\lg_m n + t)$ (where m is the order of the B-tree), or a deterministic skip list

(DSL), as introduced by Munro et al [1992], which allows range search in the same time complexity.

## 2.2 k-Dimensional Data Structures

Although there are many data structures which facilitate rapid average query time in k-dimensions, we find that there are relatively few which exhibit good worst case behavior. In comparison to the 1-dimensional case, there are simple, inefficient structures that one may use to execute a k-d range search. These include the unsorted list, which requires $\Theta(kn)$ time to perform a range search, inverted tables, which are composed of k sorted lists on each of the coordinates and which require $\Theta(kn)$ time to perform a range search in the worst case (but often require less time in the average case), and cells, which divide the space up into a number of "boxes" or "blocks" which are searched separately if they are partially (or totally) within the desired search range. Cells are no better than inverted tables for range search in the worst case, but are often better in the average case. The reader is referred to Bentley [1979] for a more detailed overview of these structures.

Another simple, inefficient approach is that of the multidimensional B-tree of Guting and Kriegel [1980]. Designed specifically for exact match (member) queries, it is no better than inverted tables for range search in the worst case but may exhibit good average case performance for uniformly distributed random data sets.

A more sophisticated approach is that of the k-d tree of Bentley [1975] which is the multidimensional equivalent of the binary search tree. Although no better than inverted

tables in the worst case scenario, the worst case is extremely rare and is $O(kn^{1-1/k} + t)$ which is guaranteed to be the worst case search time if the given tree structure is height balanced (and therefore of minimal depth). The advantages of the structure are its low storage requirement, being essentially that of a binary search tree (i.e. $\Theta(kn)$), and the fact that dynamic operations are almost as simple as those of the regular 1-d binary search tree. In the average case, which is the worst case for a height balanced structure, $P(n, k) = O(n \lg n)$ and $U(n, k) = O(\lg n)$. The drawback of the structure is that a direct implementation does not support dynamic maintaince of height balance Samet [1990].

An approach that is similar in structure, complexity, and functionality to that of the k-d tree of Bentley [1975] is that of the K-D-B-Tree of Robinson [1981] which combines the k-d tree of Bentley with the B-tree (as defined in Comer [1979]). Although a detailed analysis, to the author' s knowledge, does not exist in the literature, experimental results indicate that its efficiency parallels that of the k-d tree.

A good overview of data structures for range searching can be found in Bentley et al [1979]. The point quadtree of Samet [1990] may also be used for multidimensional range search and, under the strong assumption of relatively evenly spaced data, has a worst case range search time of $O(kn^{1-1/k} + t)$ which is comparable to that of the k-d tree.

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

### 2.2.1 The Range Tree

The range tree of Bentley and Maurer [1980] is a modified height-balanced binary search tree which is designed to detect all points that lie in a given range. We briefly review the range tree data structure and refer the reader to Samet [1990] for a more comprehensive overview.

The 1-dimensional range tree (see Figure 2.1 (a)) is a height balanced binary search tree where the data points are stored in the leaf nodes which are linked in sorted order by a doubly linked list (the leaf nodes are threaded). A range search for [L:H] is performed by searching the tree for the node with the smallest key $\geq$ L and then following the links until reaching a leaf node with a key that is greater than or equal to H. For n points, we see that this procedure takes $\Theta(\lg n + t)$ time and uses $\Theta(n)$ storage.

A 2-dimensional range tree (see Figure 2.1 (b)) is simply a range tree of range trees. We build a 2-dimensional range tree as follows: we first sort all of the points along one of the attributes, x, and then store them in a balanced 1-dimensional range tree, T. We then append to each non-leaf node, I, of the range tree T a range tree $T_I$ of the points in the sub-tree rooted at I where these points are now sorted along the other attribute, y. In Figure 2.1 (b), the darkened links connect the range tree $T_I$ (which has a *primed* node as root) in dimension 2 to the (*un-primed*) non-leaf node it is rooted at in dimension 1.

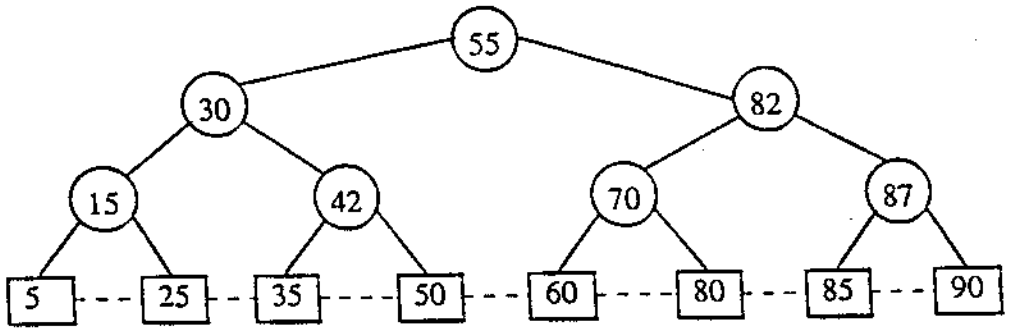


Figure 2.1 (a) A 1-d range tree (from Samet [1990]).

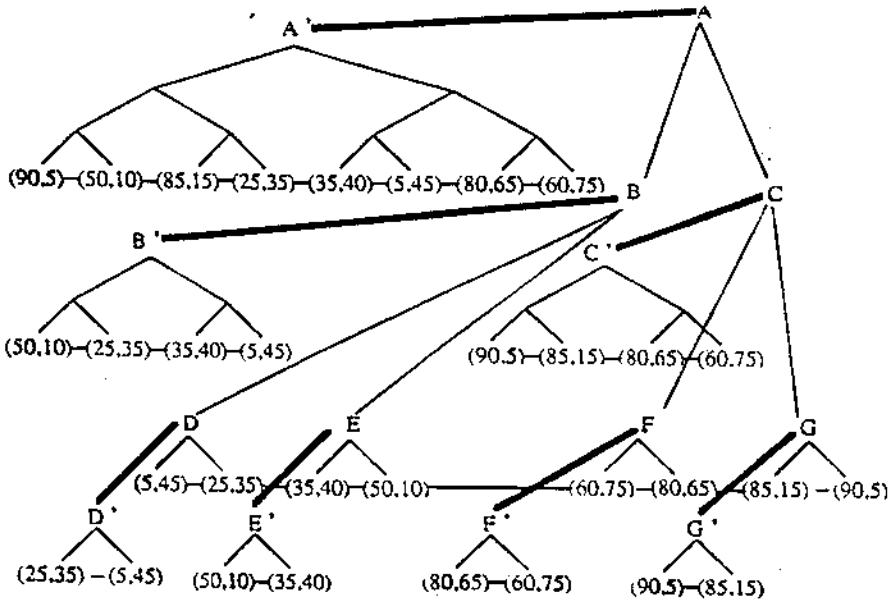


Figure 2.1 (b) A 2-d range tree (from Samet [1990]).

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

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

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

The recursively defined k-dimensional range tree has $P(n, k) = \Theta(n \lg^{k-1} n)$ and $S(n, k) = \Theta(n \lg^{k-1} n)$ in the static case and has $P(n, k) = \Theta(n \lg^k n)$ , $S(n, k) = \Theta(n \lg^{k-1} n)$, and $U(n, k) = \Theta(\lg^k n)$ in the dynamic case where an amortized worst case analysis is used. Dynamizations of the k-d range tree can be found in Lueker [1978], Willard and Lueker, [1985] and, more recently, in Lamoureux [1996].

The range tree is important for a number of reasons. Not only is it a prime example of the paradigm of multi-dimensional divide and conquer, introduced by Bentley [1980], but it is also a prime example of the effective use of a class of transformations which can be used to add range restriction to an existing data structure for a decomposable searching problem such as range search [Bentley, 1979]. Thus, we can build it bottom up starting with a 1-d range tree structure on the first coordinate or build it top down starting with a k-d data set and applying the paradigm of multi-dimensional divide and conquer until we reach a 1-d range tree structure.

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

A given interior node in the range tree is the root of a subtree of m nodes. It has two children which are the root nodes of range trees that each have approximately m/2 nodes and it has a pointer to a (k-1)-dimensional tree of m nodes. Another example of the paradigm can be found in Bentley [1979].

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

The range tree is a balanced data structure and is optimal for the execution of range queries in the class of balanced data structures. Also, the static range tree can be derived from non-overlapping k-ranges, and array-based data structures.

### 2.2.2 The Priority Search Tree

The priority search tree of McCreight [1985] is designed for solving semi-infinite range queries of the form ($[L_x: H_x]$, $[L_y: \infty]$) in optimal time. For each non-leaf node I of priority search tree S, we associate the point in the subtree rooted at I with the maximum

value for its y coordinate that has not already been stored at a shallower depth in the tree. It is assumed that no two data points have the same x coordinate; the reader is again referred to Samet [1990] for the details on the construction of the priority search tree. An example of a priority search tree is found in Figure 2.2.

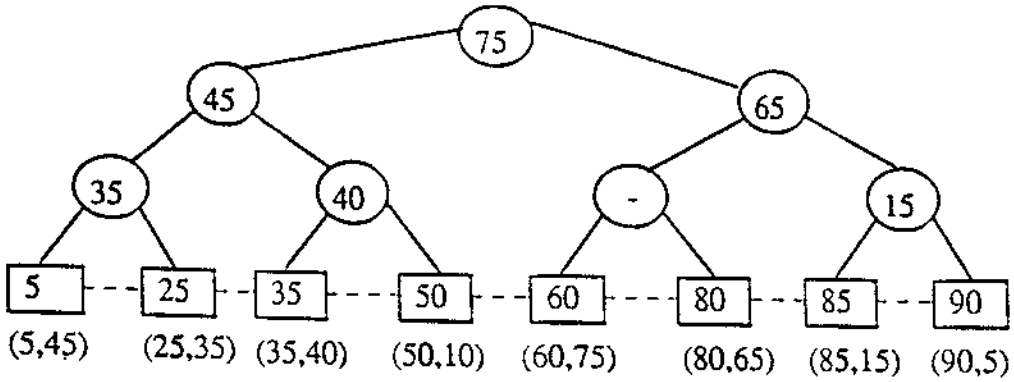


Figure 2.2 A priority search tree (from Samet [1990]).

The priority search tree requires minimal storage space, having $S(n, 2) = \Theta(n)$ and is optimal for semi-infinite range search in two dimensions, i.e. $Q(n, 1.5) = \Theta(\lg n + t)$. The 1.5 refers to the fact that only a lower or upper query limit can be specified in the second dimension, but not both. Dynamic updates are optimal as well, requiring only $\Theta(\lg n)$ time per update. However, a normal 2-d range query, $Q(n, 2)$, requires $\Theta(n)$ time in the worst case.

The importance of the structure is that it is the basis and inspiration for the Range Priority (RT) tree of Edelsbrunner [1981] and the 2-d Search Skip List of Nickerson [1994]. The Range Priority (RT) tree is important as it shaves off a logarithmic factor from the k-d range query time of the range tree.

### 2.2.3 The Range Priority Tree

The range priority tree, called the RT-tree, by Edelsbrunner [1981] is a k-d data structure designed specifically for k-d range search and it is similar to the range tree of Bentley. The structure is optimal for range search in two dimensions, i.e. $Q(n, 2) = \Theta(\lg n + t)$, and takes advantage of the priority search tree of McCreight [1985] to achieve its optimality. An example of the range priority tree can be found in Figure 2.3.

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

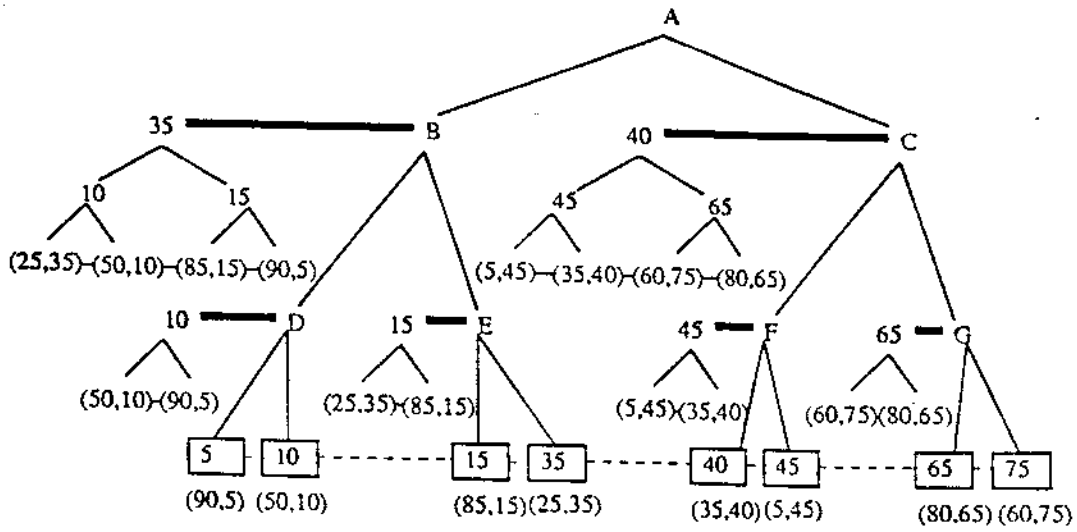


Figure 2.3 A 2-d range priority tree (from Samet [1990]).

A range priority tree is a balanced binary search tree, T, where all data points are stored in the leaf nodes and are sorted by their y coordinate values. With each non-leaf node I of T which is a right child of its parent we store an inverse priority search tree of the points

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

### 2.2.4 k-d Skip Lists

Nickerson [1994] has defined a number of k-d skip list data structures for k-d range search which are built from 1-d deterministic skip lists and similar both in structure and range query time to inverted tables. As they are not very efficient for k-d range search in the worst case, we do no more than mention them and instead focus on Nickerson's 2-d search skip list as it provides an efficient alternative to the priority search tree and is the special case for the k-d search skip list.

A deterministic skip list is a tree-like structure which has a node structure that contains (at least) a down pointer, a right pointer, and a data value. We use the following terms when describing the structure and the associated algorithms. The *down subtree* of a node N consists of all nodes that can be reached by traversing the down pointer of N, except those nodes reachable by traversing the down pointer of right sibling of N. The *left subtree* of a node N at depth i is composed of all nodes at depth (i +1) in the down subtree of N. The *gap size* is defined to be the number of nodes in the *left subtree* where we exclude the direct descendent.

For example, in Figure 2.4, note that M stands for the maximum key value, and $\phi$ stands for the non-value key. The down subtree of the node (140, 37, -30, 41) at level 1 consists of the nodes (14, 41, 41, 41), (31, -30, -30, -30), (104, 2, 2, 2), and (140, 37, 37, 37) at level 0. The left subtree of the node (M, $\phi$, -34, 52) at level 2 consists of the nodes (0, 52, 39, 52), (140, 37, -30, 41), and (M, $\phi$, -34, -34) at level 1. The direct descendent of (0, 52, 39, 52) at level 1 is (-123, 48, 48, 48) at level 0, and the gap size of (M, $\phi$, -34, -34) is 1.

#### 2.2.4.1 2-d Search Skip List

The 2-d search skip list of Nickerson [1994] uses the idea inherent in the priority search tree and is based on the linked-list version of the 1-3 deterministic skip list (DSL) of Munro et al [1992]. It is always balanced (in the sense of B-trees) as the leaves are all located at the same depth c lg n, $\frac{1}{2} \leq c \leq 1$. The properties of the structure are

1. Each node contains four values. These are
    i) $(K_1, K_2)$ = keys of the node (the two top fields in Figure 2.4)
    ii) mink2 = minimum $K_2$ value for all nodes in the left subtree
    iii) maxk2 = maximum $K_2$ value for all nodes in the left subtree
2. Each node has two pointers; a down pointer and a right pointer.
3. Special nodes head, tail, and bottom indicate the start and end of list conditions.
4. All data points appear at the leaf level and some may appear at higher levels.
5. Every gap size in the skip list is of size 1, 2, or 3.
6. The skip list is ordered on the $K_1$ key values.

The cost functions are the same as those for the priority search tree and the structure is a prime example of the fact that deterministic skip lists provide efficient alternatives to height balanced search trees as illustrated in Lamoureux [1996].

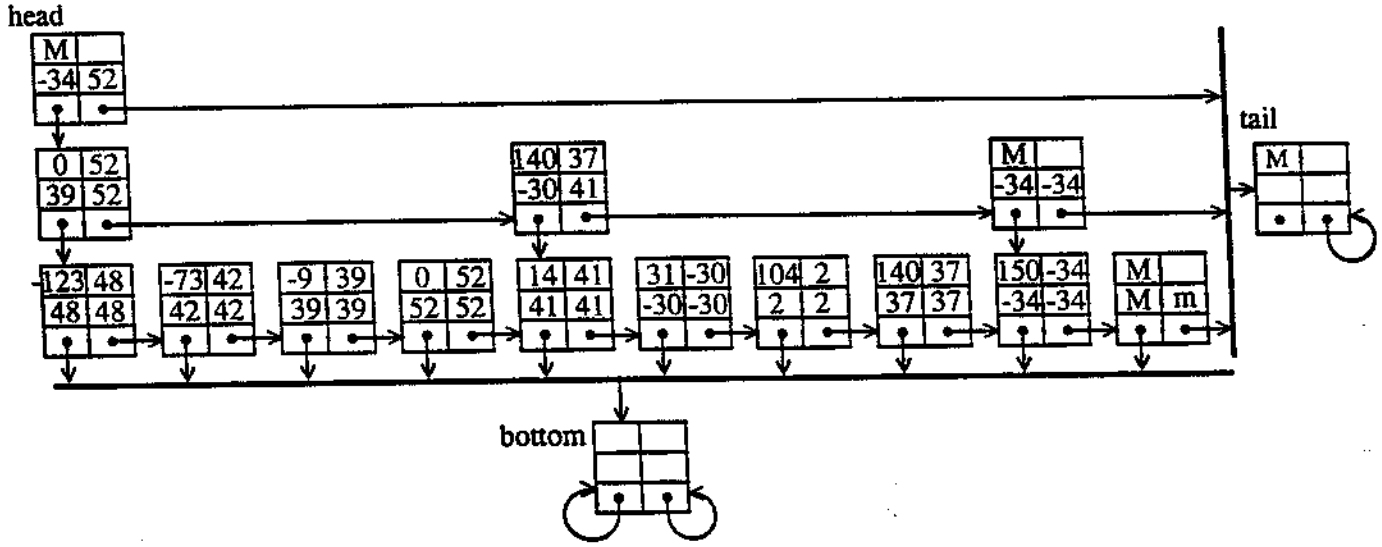


Figure 2.4 A 2-d search skip list (from Nickerson [1994]).

### 2.2.5 k-d Range Deterministic Skip List

The k-d Range Deterministic Skip List (or k-d Range DSL) is a data structure for multi-dimensional point data based on an extension of the 1-3 Deterministic Skip List projected into k-dimensions.

The properties of the structure are:

1. Each node contains a data point which is either

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

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

2. Each node contains two values. These are

   i) minki = minimum $K_i$ value in the left subtree of the current node, and

ii) maxki = maximum $K_i$ value in the down subtree of the current node.

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

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

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

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

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

The k-d Range DSL supports fast insertions and deletions which require $\Theta(\log^k n)$ amortized time, and it requires $Q(n, k) = \Theta(\log^k n + t)$ time to perform a worst case k-d range search. It is dynamically balanced and optimal for k-d range search in the class of dynamically balanced data structures. An example of a 2-d range DSL can be found in Figure 2.5.

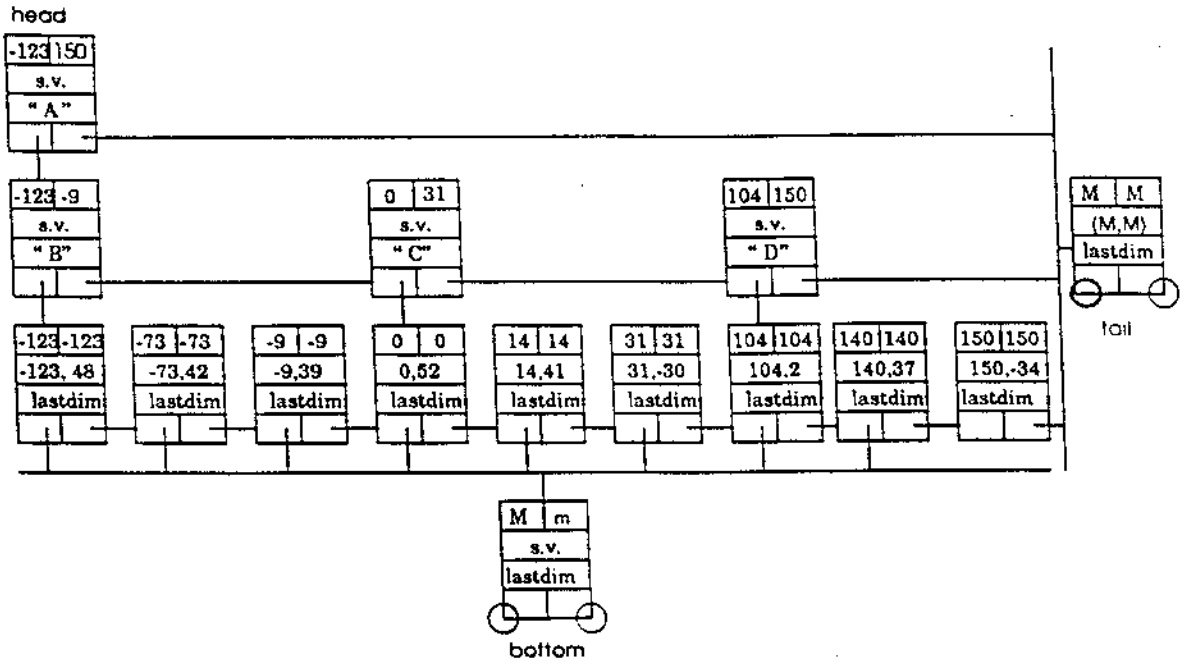


Figure 2.5 (a). The Range DSL structure in dimension 1 (ordered on $K_1$)

for a 2-d range DSL (from Lamoureux [1996]).

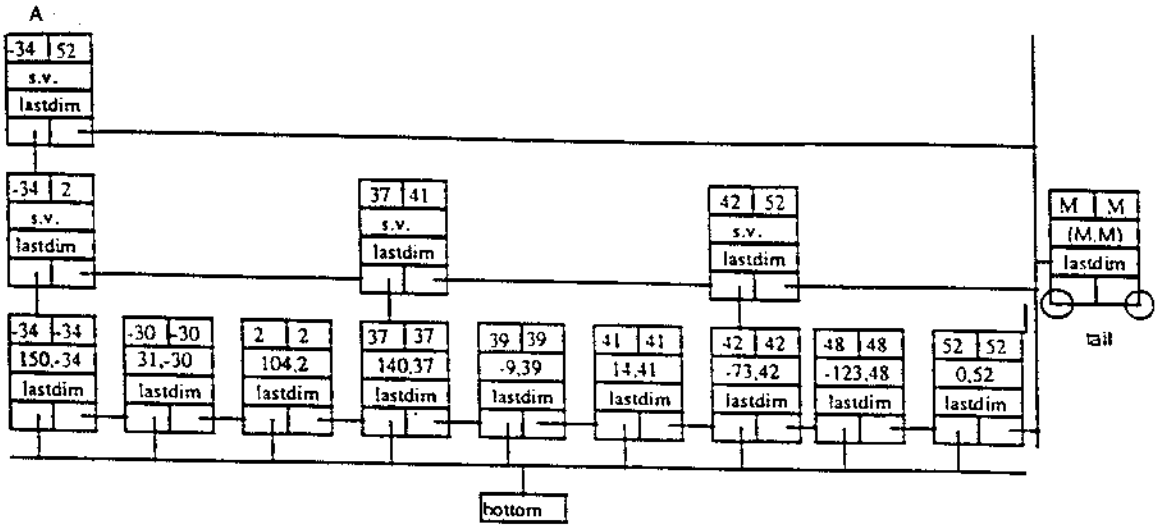


Figure 2.5 (b) The Range DSL structure "A" (ordered on $K_2$)

for a 2-d range DSL (from Lamoureux [1996]).

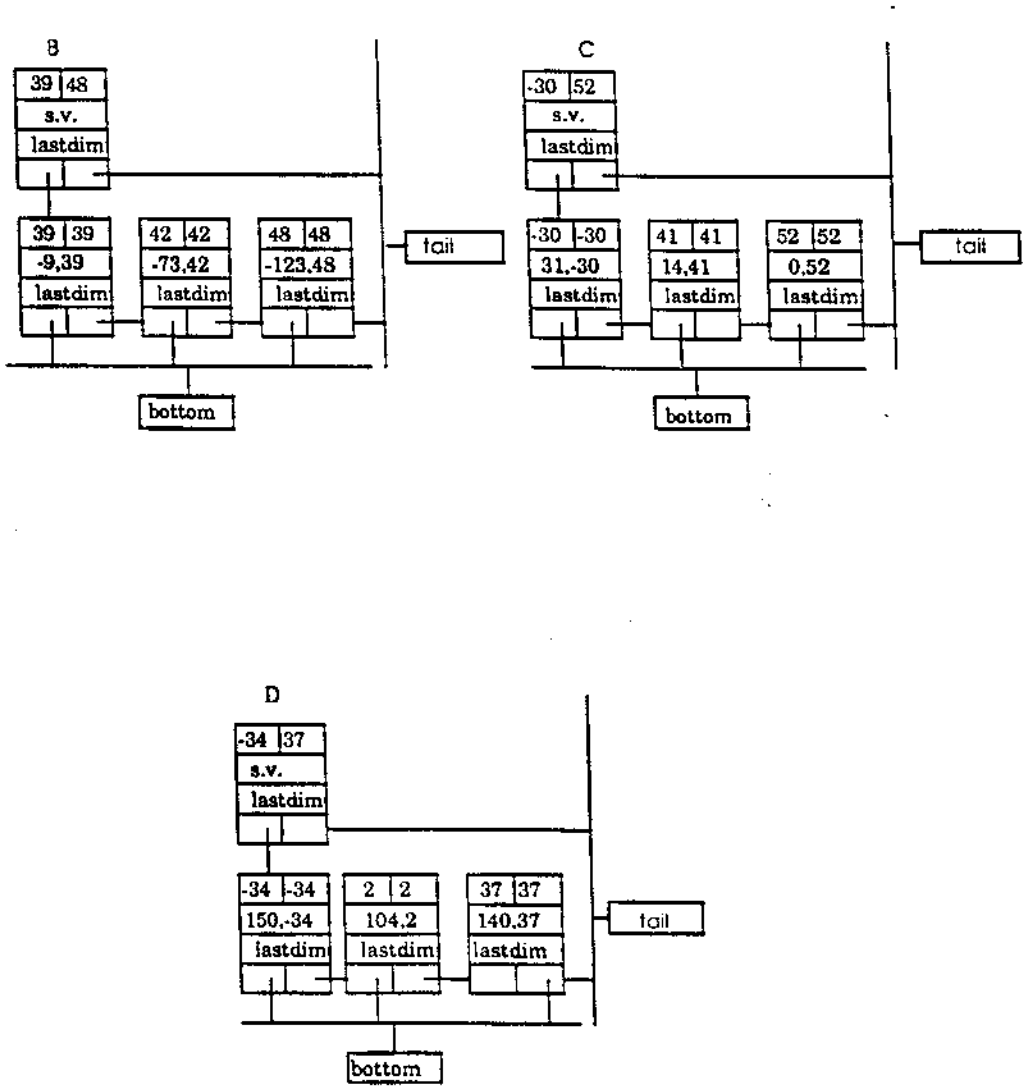


Figure 2.5 (c) The Range DSL structures "B", "C", & "D" (ordered on $K_2$) for a 2-d range DSL (from Lamoureux [1996]).

# Chapter 3

# THE K-D SEARCH SKIP LIST

The k-d Search Skip List is a new data structure for multi-dimensional point data based on a 2-d Search Skip List Nickerson [1994]. It uses the priority search tree idea. The k-d Search Skip List of order m is based on the linked-list version of the 1-3 Deterministic Skip List (DSL) Papadakis [1993]. This means that it is always balanced in the sense of B-trees; i.e. the leaves are all at the same depth of log *n*.

## 3.1 Definition of a k-d Search Skip List

A k-d Search Skip List of order m has the following properties:

1. Each node contains a data point which is an actual data point ($K_1$, $K_2$, ..., $K_k$ )
2. Each node contains (2k-2) values. These are
   i) minki = minimum $K_i$ value for all nodes in the left subtree of the current node, where $2 \le i \le k$
   ii) maxki = maximum $K_i$ value for all nodes in the left subtree of the current node, where $2 \le i \le k$
3. Each node has two pointers; a down pointer and a right pointer. The down pointer points to the left subtree for this node. The right pointer points to the right subtree for this node. A right subtree is defined as the right sibling node of this node. A left subtree is defined as all the descendant nodes and

their siblings which have a $K_1$ value ≤ the $K_1$ value of this node.

4. Three special nodes called head, tail and bottom are used to indicate the start and end of list condition. The nodes head, tail and bottom are identical to the head, tail, and bottom nodes in the 1-3 DSL.

5. All data points appear at the leaf or bottom level, and some points also appear at nodes above the bottom level.

6. Every gap size in the skip list is between $\lfloor m/2 \rfloor$ and m. Gap size is defined as the number of elements of height h-1 that exist between two linked elements at height h, excluding direct descendants.

7. The skip list is ordered by the $K_1$ key values.

An element (used in describing property 6) is a vertically stacked set of nodes linked directly by down pointers. As described in Papadakis [1993], this structure of order 3 is somewhat similar to a 2-3-4 tree. Figure 3.1 illustrates the composition of a single node, and Figure 3.2 shows a 2-d Search Skip List of order 3.

| datapoint | |
|---|---|
| min. $K_2$ value in the left subtree | max. $K_2$ value in the left subtree |
| . | . |
| . | . |
| . | . |
| min. $K_k$ value in the left subtree | max. $K_k$ value in the left subtree |
| down pointer | right pointer |

Figure 3.1 Composition of a single node in a k-d search skip list.

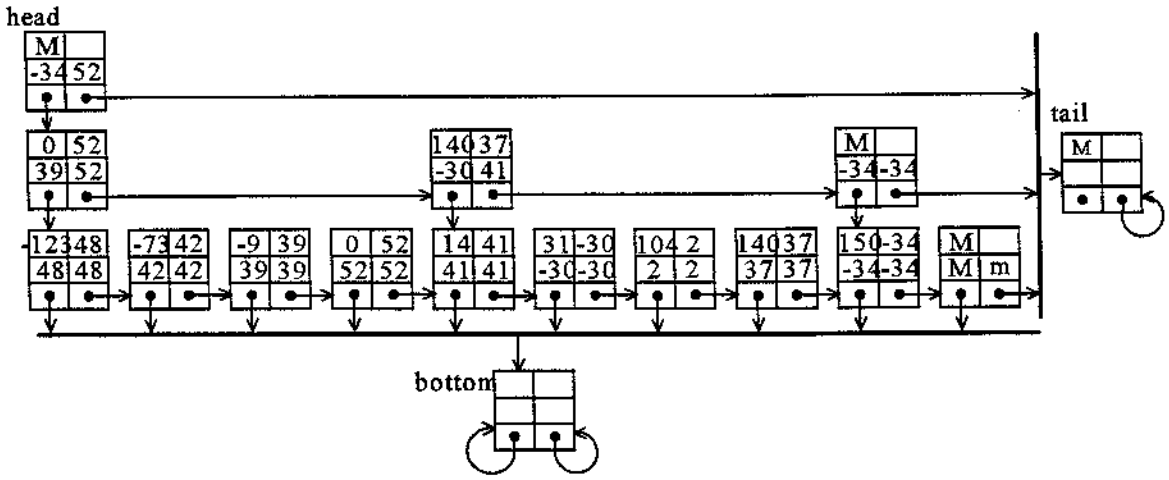


Figure 3.2 A *2*-d search skip list (order on $K_1$).

The main advantage of the k-d Search Skip List lies in the fact that it is much simpler to conceptualize and implement than a dynamic range tree structure. It has the additional advantage that the algorithms are highly iterative in nature. This implies that some implementations could outperform some implementations of a range tree as recursion involves extra overhead, and therefore extra time. Also, as we usually will not have a worst-case structure, the depth of the structure is usually less than that of a corresponding range tree structure and thus range searches may be faster than they would be in a range tree.

## 3.2 Description of Implementation Design

First we look at the aggregation diagram (see Figure 3.3) of class KDSSkipNode and class KDSSkipList.

To help understand the diagram, I would like to introduce several OMT concepts Rumbaugh et al [1991]. A line with a small diamond is the OMT symbol for "aggregation". Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. The small diamond indicates the assembly end of the relationship. A solid ball is the OMT symbol for "many", meaning zero or more. That is, many instances of one class may relate to a single instance of another class. Therefore our diagram indicates that one instance of KDSSkipNode class contains zero (empty list) or many instances of the KDSSkipNode class.

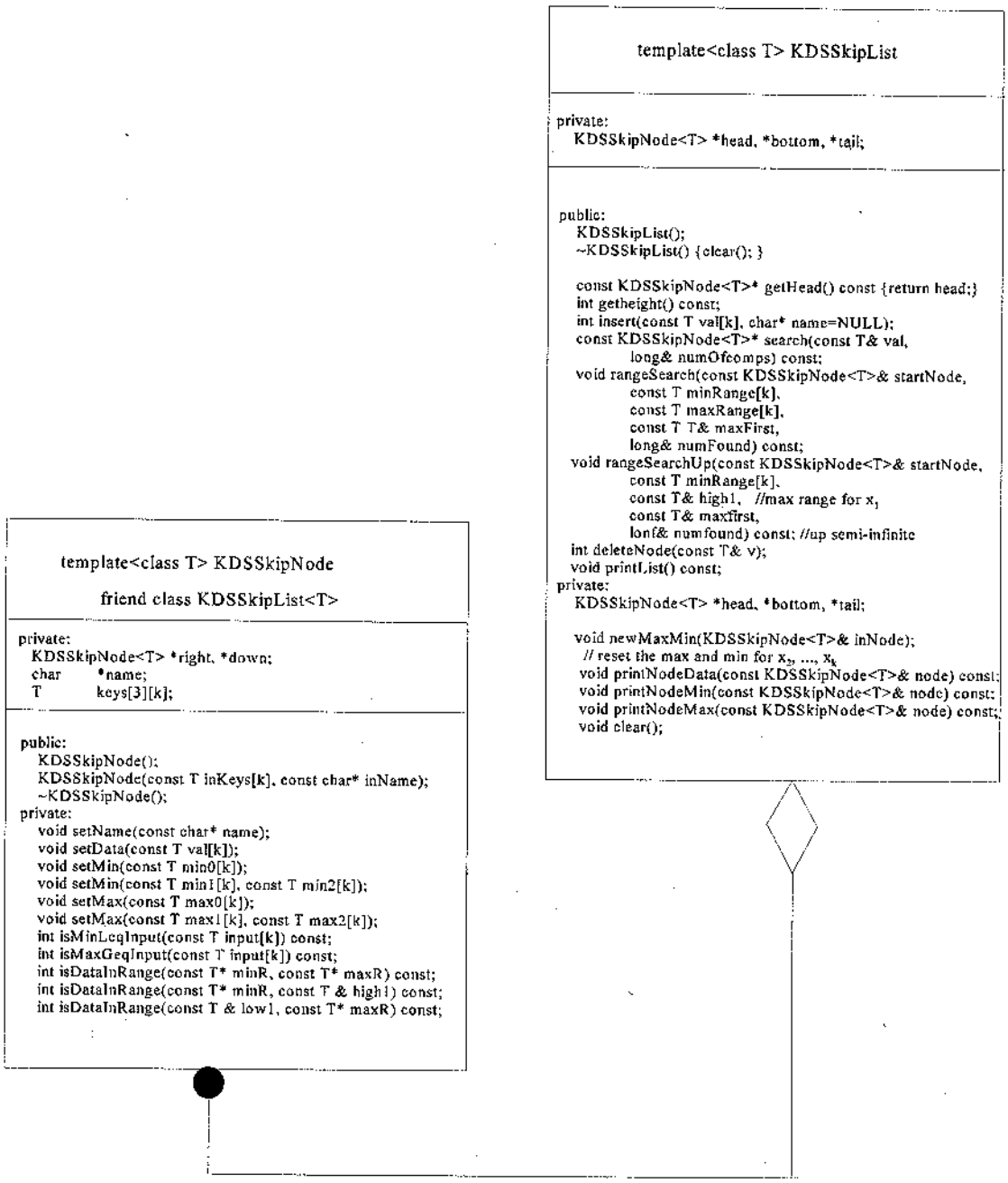


Figure 3.3 Aggregation between class KDSSkipNode and class KDSSkipList.

For class KDSSkipNode, it's data members are:

```
KDSSkipNode<T> *right;    //the right pointer.
KDSSkipNode<T> *down;     //the down pointer.
char    *name;            //each node may have a name.
T       keys[3][k];       //keys[0][]: x_1, x_2, ..., x_k;
                          //keys[1][]: minx_1, minx_2, ..., minx_k;
                          //keys[2][]: maxx_1, maxx_2, ..., maxx_k.
```

Figure 3.4 (a) The data members of the class KDSSkipNode.

The member functions are (see Figure 3.4 (b)):

```
public:
   KDSSkipNode();          //default constructor of KDSSkipNode.
   KDSSkipNode(const T inKeys[k], const char* inName);
                                 //constructor of KDSSkipNode.
   ~KDSSkipNode();      //destructor of KDSSkipNode.
private:
   void setName(const char* name);   //set the name of the node.
   void setData(const T val[k]);        //set the k key values for the node
   void setMin(const T min0[k]);
                 //set the minimum values of x2, ..., xk as the input
   void setMin(const T min1[k], const T min2[k]);
                //set the minimum values of x2, ..., xk as the minimum
                //of min1[i] and min2[i].
   void setMax(const T max0[k]);
                //set the maximum values of x2, ..., xk as the input.
   void setMax(const T max1[k], const T max2[k]);
                //set the maximum of x2, ..., xk as the maximum
                //of max1[i] and max2[i].
   int isMinLeqInput(const T input[k]) const;
                //check whether all the mins are less than or equal to the input.
   int isMaxGeqInput(const T input[k]) const;
                //check whether all the maxs are greater than or equal to the input.
   int isDataInRange(const T* minR, const T* maxR) const;
                //check whether all the data are inside the given range.
   int isDataInRange(const T* minR, const T & high1) const;
                //check whether all the data are inside the given range.
   int isDataInRange(const T & low1, const T* maxR) const;
                //check whether all the data are inside the given range.
```

Figure 3.4(b) The description of the member functions of class KDSSkipNode.

For class KDSSkipList, it's data members (see Figure 3.5 (a)):

```
KDSSkipNode<T> *head;    //the head.
KDSSkipNode<T> *bottom;  //the bottom node.
KDSSkipNode<T> *tail;       //the tail node.
```

Figure 3.5 (a) The data member of the class KDSSkipList.

The member functions are (see Figure 3.5 (b)):

```
public:
  KDSSkipList();              //constructructor of KDSSkipList<T>.
  ~KDSSkipList() {clear(); }        //destructor of KDSSkipList<T>.

  const KDSSkipNode<T>* getHead() const {return head;}
  int getheight() const;        //returns the height of this k-d Search Skip List.
  int insert(const T val[k], char* name=NULL);
            //insert the data val[k] and name into the k-d Search Skip List.
  const KDSSkipNode<T>* search(const T& val,
        long& numOfcomps) const;
            //returns the node whose keys[0][0] matches the input.
  void rangeSearch(const KDSSkipNode<T>& startNode,
        const T minRange[k], const T maxRange[k],
        const T T& maxFirst, long& numFound) const;
            //perform rangesearch on k-d priority search tree constructed using
            //a deterministic skip list.
  void rangeSearchUp(const KDSSkipNode<T>& startNode,
        const T minRange[k],
        const T& high1,  //max range for x1
        const T& maxfirst,
        lonf& numfound) const; //up semi-infinite
            //perform up semi-infinite range search on k-d priority dearch tree
            //constructed using a deterministic skip list.
  int deleteNode(const T& v);       //delete a node matching the input key.
  void printList() const;           //print the whole skip list.

private:
   void newMaxMin(KDSSkipNode<T>& inNode);
    // reset the max and min for x2, ..., xk for inNode.
    void printNodeData(const KDSSkipNode<T>& node) const;
            //print the node (without the minimum and maximum).
    void printNodeMin(const KDSSkipNode<T>& node) const;
            //print the node's minimum values.
    void printNodeMax(const KDSSkipNode<T>& node) const;
            //print the node's maximum values.
    void clear();       //clear the whole list.
```

Figure 3.5 (b) The member functions of the class KDSSkipList.

## 3.3 Building a k-d Search skip List

Constructing a k-d Search Skip List requires inserting each point into the k-d Search Skip List sequentially, starting with an empty k-d Search Skip List. Algorithm 3.1 can be used for this. As the k-d Search Skip List is constructed by inserting one point at a time, we discuss the time and space requirements for building the structure in the section on insertion. Note that minkey and maxkey are special constants which delineate the range that all data points must fall in.

## 3.4 Insertion in a k-d Search Skip List

Insertion proceeds similarly to that of the 1-3 DSL and the 2-d Search Skip List Nickerson [1994] with the only difference being that we must check the minkey and maxkey for each dimension except for dimension 1. Note that the algorithm relies on the private data members head, tail and bottom of the instantiation of the KDSSkipList <T> class along with the concepts of left subtrees and gap size. In addition, we assume that all of the K coordinates are unique and that the data point is not already in the k-d Search Skip List.

The (top down) insertion algorithm, in summary, is as follows:

1. Start at the head node of the Search Skip List.

2. If the gap we are going to drop into is between $\lfloor m/2 \rfloor$ and m-1, determine where to drop, and alter the minkey or maxkey of the dropping node as appropriate.

3. If the gap is of size m, we first raise the middle element of the gap by allocating a new node C to create two gaps of size $\lfloor m/2 \rfloor$ and $m - \lfloor m/2 \rfloor$. We then drop as appropriate and alter the minkeys and maxkeys of the dropping node and the node C as necessary. If raising an element implies the existence of two elements at depth 0, we must raise the height of the skip list by creating a new header node.

4. When we reach the bottom level, we insert a new element of height 1. This new element has a minimum and maximum value of $K_i$, where i = 2, 3, ..., k.

Algorithm 3.1 Summary of insertion steps in a k-d Search Skip List.

Algorithm 3.2 can be used for insertion of a point from a k-d skip list. Algorithm 3.2 corresponds to the C++ pseudocode used for insertion.

```
1   template<class T>
2   void KDSSkipList<T>::insert(const T val [k], char* name)
3   {
4       int add1 = 0; success = 1;
5       bottom->keys[0], keys[1], keys[2] = val; bottom->name = name;
6       tmpNode = head;
7       while (tmpNode != bottom)       //at each level
8       {
9               find the node in the right direction such that val < key,
                    rename the node as tmpNode;
10              if (tmpNode->keys[0][0]
                                > tmpNode->down->right->right->keys[0][0])
11              {
12                  insert a newNode betwee tmpNode and tmpNode->right;
13                  newNode->down = tmpNode->down->right->right;
                            //middle one
14                  newNode->keys[0] = tmpNode->keys[0];
15                  newNode->name = tmpNode->name;
16                  adjust the min's and max's for newNode (considering val)
17                  tmpNode->keys[0] = tmpNode->down->right->keys[0];
18                  tmpNode->name = tmpNode->down->right->name;
19                  adjust the min's and max's for newNode (considering val)
20                  add1 = 1;
21              }
22              else if (tmpNode->down == bottom)
                        //insert key already in the list
23                  success = 0;
24              if ( tmpNode->down != bottom && !add1)
25              {
26                  min of tmpNode = min (min of tmpNode, val);
27                  max of tmpNode = max (max of tmpNode, val);
28              }
29              tmpNode = tmpNode->down;
30      }
31      if (head->right != tail)
32              raise height; (i.e. create a new head)
33      return (success);
34  }
```

Algorithm 3.2 Insertion of data val[k] and name into a k-d Search Skip List.

This algorithm allows only gaps between $\lfloor m/2 \rfloor$ and m-1 on the path being traced down to the leaf level and the resulting skip list with a newly inserted element is a valid order m skip list (i.e. has gap sizes between $\lfloor m/2 \rfloor$ and m). Since the minimum and maximum value of $K_i$, where i = 2, 3, ..., k are always determined with respect to the new point's $K_i$ coordinate value, the minimum and maximum value of $K_i$, where i = 2, 3, ..., k for a node are always correct.

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

Define path length as the number of nodes encountered when traversing a k-d Search Skip List.

**Lemma 3.1** The path length from the head node to the bottom level in a k-d Search Skip List is O(log *n*).

We omit the proof of Lemma 1, since it closely follows the method achieving a similar result for the 2-d Search Skip List in Nickerson [1994].

**Theorem 3.1** The time I(n, k) required to insert a new point into a k-d Search Skip List is O(*k*log *n*).

Proof

The proof depends on the longest path length encountered on insertion. Insertion always starts at the head node, and proceeds via a single path to the bottom level, whereupon the new point is inserted. Along the way, new nodes may be created to increase the height of middle elements in the skip list. The time to create a new node is considered constant as

it mainly requires the copying of pointers and checking of minimum and maximum values.

The longest path length encountered on insertion for the k-d Search Skip List is given by Lemma 1 as O(log $n$). The maximum number of new nodes inserted on the way to the bottom level is equal to the maximum height of the k-d Search Skip List, which is (from the proof of Lemma 1) $\lfloor \log n \rfloor + 2$ (as a new head node may be required if the height increases by one), in order to get the position where you want to insert the new node in the k-d Search Skip List, there are $2k$ comparisons. This gives the time I(n, k) to insert a new point into a k-d Search Skip List as O($k$log $n$). **QED.**

**Theorem 3.2** The space S(n, k) required for storage of a k-d search skip list is O($kn$).

Proof

Each node in the k-d Search Skip List has two pointers and (3k-2) data values. From Lemma 1, it is known that the number of nodes at each level $l$ is $\left\lfloor \frac{n}{2^l} \right\rfloor + 1$, and that the maximum height of the k-d search skip list is $\lfloor \log n \rfloor$. The total number of nodes in the worst case is thus $\sum_{l=0}^{\lfloor \log_2 n \rfloor} \left( \left\lfloor \frac{n}{2^l} \right\rfloor + 1 \right)$. Evaluating this sum results in O(log $n$) + O($2n$). Assuming that each pointer and data value occupy one storage space, the worst case storage requirement for a $k$-d search skip list is $3k$(O(log $n$) + O($2n$)) = O($kn$). **QED.**

**Theorem 3.3** Using Algorithm 3.2, the time P(n, k) required to construct a k-d Search Skip List is O($kn$ log $n$).

Proof

Each point to be inserted into the k-d Search Skip List is first checked to see if its $K_1$ value is already there. This requires O(log $n$) time as, in the worst case, the longest path (Lemma 1) will have to be followed to determine if the point is already there. If the $K_1$ value is not already there, the point is then inserted into the k-d search skip list. From Theorem 3.1, we know that this requires O($k$log $n$) time. These operations are carried out a total of $n$ times, once for each point inserted into the k-d search skip list, resulting in a total time of $n$(O($k$log $n$) + $O$(log $n$)) = O($kn$ log $n$). **QED.**

## 3.5 Range Search for a k-d Search Skip List

Algorithm 3.3 can be used to perform a k-d orthogonal range search on a k-d Search Skip List. Note that it is in some ways more involved than the search algorithm for the range tree as we do not always know at which node in the immediate down subtree we are to continue our search and must allow for this situation. This procedure may also be used for a member query which is the special case where our search interval is a data point.

The algorithm assumes that the Search Skip List data structure we are currently searching is built in a top down fashion and that the query ranges are given in the variables minRange and maxRange.

Algorithm 3.3 can be used for orthogonal range search on a k-d Search Skip List. Again, Algorithm 3.3 corresponds exactly to the C++ code used for range search in the experiments.

```
// Perform range search on k-d priority search tree constructed using a
// deterministic skip list. i.e. report all points lying inside the range
// ([min1:max1], [min2:max2], ..., [mink:maxk]).

1     template<class T>
2     void KDSSkipList<T>::rangeSearch(const KDSSkipNode<T>& startNode,
            const T minRange[k],
            const T maxRange[k],
            const T& maxFirst,
            long& numFound) const
3     {
4         if ((&startNode != bottom) && (&startNode != tail))
5         {
6             if ((startNode.down != bottom) &&
                 (startNode.isMinLeqInput(maxRange)) &&
                 (startNode.isMaxGeqInput(minRange)))
7             {     // Not leaf level
                  // Check the first data to see which branch it goes
8                 if (*minRange <= startNode.keys[0][0])
9                     rangeSearch(*startNode.down, minRange,
                      maxRange, startNode.keys[0][0], numFound);
10                if ((startNode.keys[0][0] < *maxRange) &&
                     (startNode.keys[0][0] < maxFirst))
11                    rangeSearch(*startNode.right, minRange,
                      maxRange, maxFirst, numFound);
12            }
13            else if (startNode.down == bottom)
14            {     // leaf level, report points if in range
15                if (startNode.isDataInRange(minRange, maxRange))
16                {
                      //printNodeData(startNode);
17                    numFound++;
18                }
                  // Keep check the right points
19                if ( (startNode.keys[0][0] < *maxRange) &&
                      (startNode.keys[0][0] < maxFirst) )
20                    rangeSearch(*startNode.right, minRange,
                      maxRange, maxFirst, numFound);
21            }
22        }
23    }
```

Algorithm 3.3 Range search algorithm for a k-d Search Skip List

Note that the method ".isMinLeqInput(maxRange)" checks whether all the mins are less than or equal to the maxRange (which is the upper right of the searching region); the method ".isMaxGeqInput(minRange)" checks whether all the maxs are greater than or equal to the minRange (which is the lower left coordinate of the searching region); and the method ".isDataInRange(minRange, maxRange) checks whether all the data are inside the given range.

**Theorem 3.4** The time Q(n, k) list required for range search in a k-d Search Skip List is O(*kn*).

Proof

The worst case for k-d range search with a k-d Search Skip List is illustrated in Figure 3.4. In this case, there are no points in the range, but the rangeSearch algorithm must search *n* nodes to make this determination. The points are organized in the $K_1$ keys such that they alternate between the minimum and maximum $K_2$ , ... , $K_k$ key values. This means that the left subtrees of all nodes in the upper levels have the same minimum and maximum $K_2$ , ... , $K_k$ key values. Thus, the test at line 6 of the rangeSearch algorithm never prunes any subtrees as the minimum $K_2$ , ... , $K_k$ values are always less than maxRange (i.e. the upper bound of the query), and the maximum $K_2$ , ... , $K_k$ values are always greater than minRange (i.e. the lower bound of the query), until the bottom level is reached. This means that O(*kn*) nodes are checked to determine that no points are in the desired range. **QED.**

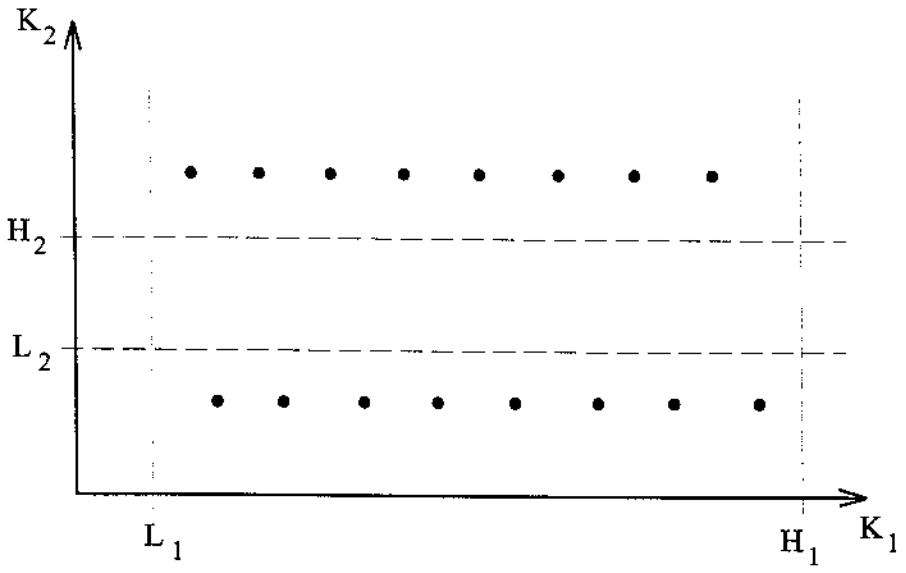


Figure 3.6 Illustration of worst case for 2-d range search with a 2-d search skip list.

## 3.6 Semi-infinite Range Search in a k-d Search Skip List

Interestingly, the k-d search skip list can be used for semi-infinite range search, in either the ($[L_1: H_1]$, $[L_2: \infty]$, $[L_3: \infty]$, ... $[L_k, \infty]$), or the ($[L_1: H_1]$, $[-\infty: H_2]$, ..., $[-\infty: H_k]$) mode. We call these the "Up" mode and "Down" mode, respectively. Algorithm 3.4 can be used for the Up semi-infinite range search; the Down semi-infinite range search algorithm is in the source code. The primary difference between this algorithm and the general k-d range search in Algorithm 3.3 is at line 6, where the subtree pruning only occurs if the maximum $K_2$ , ..., $K_k$ values are less than the lower bound on the semi-infinite query for the keys (i.e. all $K_2$ , ..., $K_k$ keys are now out of range).

```
// Perform up semi-infinite range search on k-d priority search tree
//constructed using a deterministic skip list. i.e., report all points
//lying inside the range ([min1:max1], [min2:infinity),
//..., [mink:infinity)).

1     template<class T>
2     void KDSSkipList<T>::rangeSearchUp(const KDSSkipNode<T>& startNode,
                    const T minRange[k],
                    const T& high1,
                    const T& maxFirst,
                    long& numFound) const
3     {
4         if ((&startNode != bottom) && (&startNode != tail))
5         {
6             if ((startNode.down != bottom) &&
                  (startNode.isMaxGeqInput(minRange)))
7             {     // Not leaf level
                  // Check the first data to see which branch it goes
8             if (*minRange <= startNode.keys[0][0])
9                       rangeSearchUp(*startNode.down, minRange,
                            high1, startNode.keys[0][0], numFound);
10            if ((startNode.keys[0][0] < high1) &&
                        (startNode.keys[0][0] < maxFirst))
11                      rangeSearchUp(*startNode.right, minRange,
                                high1, maxFirst, numFound);
12            }
13            else if (startNode.down == bottom)
14            {     // leaf level, report points if in range
15                  if (startNode.isDataInRange(minRange, high1))
16                  {    //printNodeData(startNode);
17                  numFound++;
18                  }
                    // Keep check the right points
19                  if ( (startNode.keys[0][0] < high1) &&
                              (startNode.keys[0][0] < maxFirst) )
20                            rangeSearchUp(*startNode.right, minRange,
                              high1, maxFirst, numFound);
21            }
22        }
23    }
```

Algorithm 3.4 Up semi-infinite range search for a k-d Search Skip List.

The algorithm 3.4 can be stated in the pseudo-code form as the follow:

1. Start at head node.

2. If not bottom and tail node, then check

If not leaf level and $K_{imax} \geq L_i$, i = 2, ..., k,

If $L_1 \leq K_1$ search down;

If $K_1 < H_1$ and $K_1 < M$, search right;

If leaf level, check

If $L_i \leq K_i \leq H_i$, i = 1, ..., k, report;

If $K_1 < H_1$ and $K_1 < M$, search right;

Algorithm 3.5 Up semi-infinite range search pseudo-code form for a k-d Search Skip List.

**Theorem 3.5** The time Q(n, k) required for semi-infinite range search in a k-d Search Skip List is $O(kt + kn)$, for t = number of points found in the range.

Proof

We consider the following cases (Figure 3.7 and Figure 3.8).

In Figure 3.7, there are no points in the range, the test at line 6 of the rangeSearchUp algorithm will prune all the left subtrees as the maximum $K_2$ , ..., $K_k$ values are less than the lower bound on the semi-infinite. This means that O(1) nodes are checked to determine that no points are in the desired range. This is not the worst case semi-infinite range search.

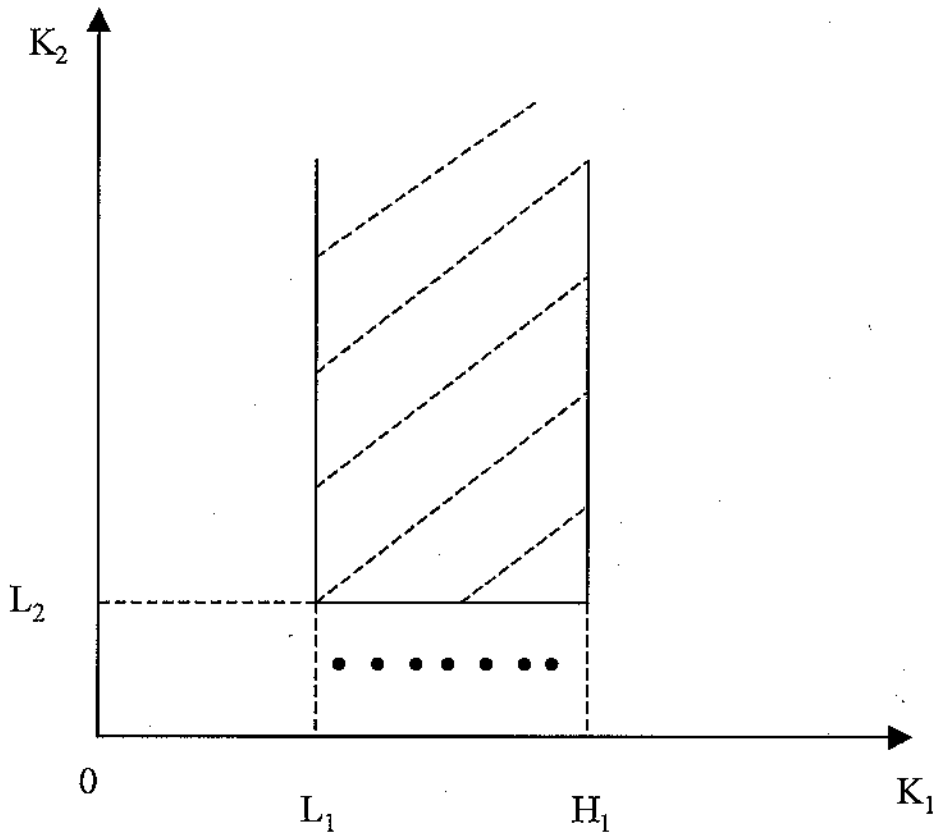


Figure 3.7 Illustration of no data point in the 2-d up semi-infinite search region.

The worst case for k-d semi-infinite range search with a k-d Search Skip List of order m is illustrated in Figure 3.8. There are some points in range, but the rangeSearchUp algorithm must search n nodes to make this determination. The points are organized in the $K_1$ keys such that alternately, there is one point less than the minimum $K_2$ , ..., $K_k$ values and one point great than the minimum $K_2$ , ..., $K_k$ values. Thus, the test at line 6 of the rangeSearchUp algorithm never prunes any subtrees as the maximum $K_2$, ..., $K_k$ values are always greater than minRange (i.e. the lower bound of the query), until the bottom level is reached. This means that

all nodes in the k-d Search Skip List are visited to determine that t points are in range. Thus the time requirement for this worst case is O($kn + kt$). **QED.**

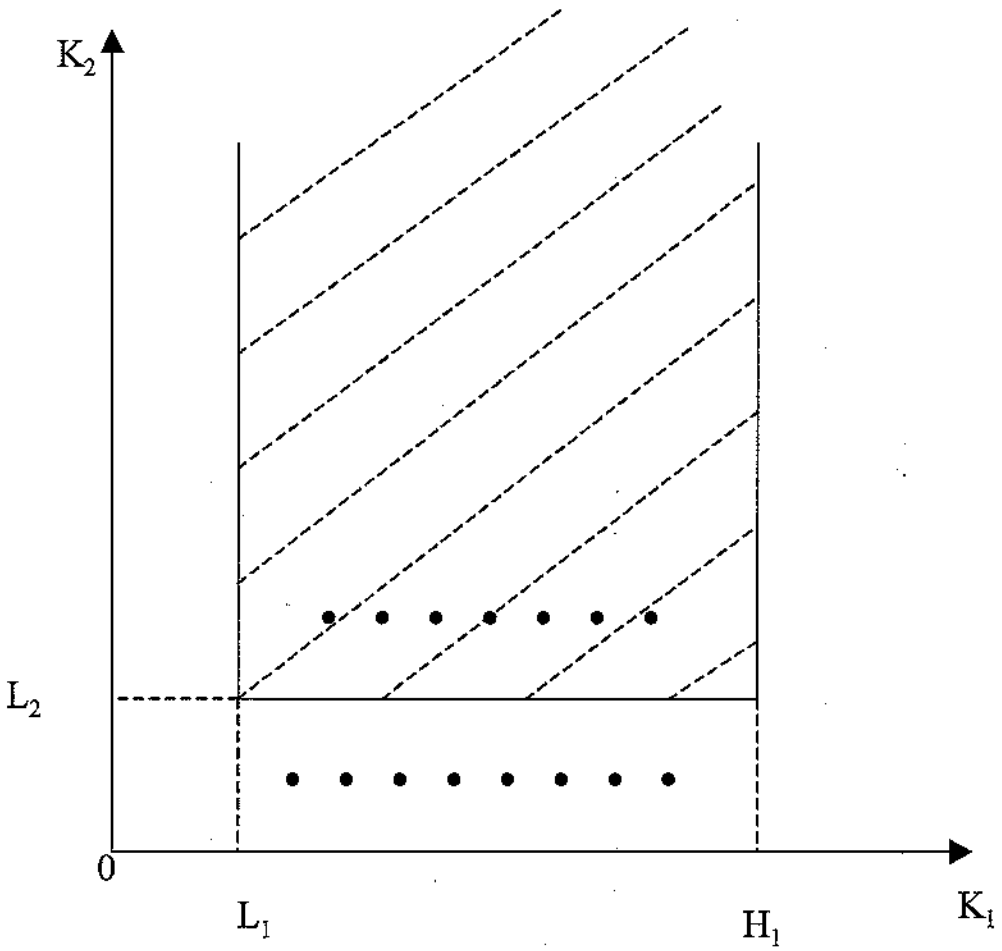


Figure 3.8 Illustration of worst case for 2-d up semi-infinite range search with 2-d Search Skip List of order 3.

## 3.7 Deletion in a k-d Search Skip List

Deletion for a k-d search skip list requires keeping track of the path followed to the bottom on the deletion path, and then reestablishing the minimum and maximum $K_2$ values in a bottom up fashion after the node is deleted. There is a complication if the gap is of size $\lfloor m/2 \rfloor$. Borrowing from the next (or previous) gap causes a node to have a different left subtree. This, in turn, means that the mink2, ..., minkk and maxk2, ..., maxkk value for this changed node must be reevaluated before proceeding to the next level. A summary of the deletion steps is given below in Algorithm 3.5.

1. Start search at the header, at a level = height of skip list.
2. The current node has a gap G of size $\lfloor m/2 \rfloor$, ..., m-1 or m; record the current node in an array P.
3. If gap G we are going to drop into is of size $\lfloor m/2 \rfloor + 1$, ..., m-1 or m, drop.
4. If the gap G is of size $\lfloor m/2 \rfloor$, then:
5. If G is not the last gap in the current level, then
6. If the following gap G' is of size $\lfloor m/2 \rfloor$, merge G and G' (by lowering the element separating G and G').
7. Else if G' is of size $\lfloor m/2 \rfloor + 1$, ..., m-1 or m, we borrow from it (by lowering the element separating G and G', and raising the first element in G'). The minimum and maximum $K_2$ values of the top node of the first element in G' are reset to correspond to its new left subtree nodes.
8. Else if G is the last gap, then
9. If the preceding gap G' is of size $\lfloor m/2 \rfloor$, merge G and G' (by lowering the element separating G and G').
10. Else if G' is of size $\lfloor m/2 \rfloor + 1$, ..., m - 1 or m, we borrow from it (by lowering the element separating G and G', and raising the last element in G'). The minimum and maximum $K_2$ , ..., $K_k$ values of the top node of the last element in G' are reset to correspond to its new left subtree nodes.
11. Continue until we reach the bottom level, where we remove the element of height 1 (if the key to be deleted does not have height 1, swap with its predecessor of height 1 and remove its predecessor).
12. Follow the pointers in array P from the bottom up, reestablishing the minimum and maximum $K_2$ , ..., $K_k$ values for all nodes on the deletion path.

Algorithm 3.5 Summary of deletion steps in a k-d search skip list

Algorithm 3.6 can be used for deletion of a point from a k-d Search Skip List.

Again, Algorithm 3.6 corresponds the C++ pseudocode used for deletion in several

experiments.

```
//delete a node matching the input key.

1   template<class T>
2   void KDSSkipList<T>::deleteNode(const T& v)
3   {
4       push head to a stack
5       bottom->kkeys[0][0] = v; x = head->down;
6       while (x != bottom)   //for every level
7       {   go to the right to find the first x such that x->keys[0][0] > v;
8           let lastAbove = key of lastAbove node of x;
9           if x has gap size ⌊m/2⌋, then
10          {   if (x->keys[0][0] != lastAbove)  //x is not the last gap
11              {   t = x->right;
13                  if ((t has gap size ⌊m/2⌋) or (x is leaf level))
14                      merge x and t->right by delete t;
15                  else  //has gap size great than ⌊m/2⌋
16                  {   x->keys[0] = t->down->keys[0];
17                      t->down = t->down->right;
18                      reset min's and max's for t; }
19              }
20              else if previous gap of x has gap size ⌊m/2⌋;
21              {   merge px and x->right by deleting x;
22                  rename px as x;   }
23              else   //px has gap size great than ⌊m/2⌋
24              {   px->kkeys[0] = keys[0] of the last second node of the gap px;
25                  x->down = t->right;
26                  reset min's and max's for px;  }
27          }
28          push x stack, x = x->down;
29      }
30      do a second pass from head->down, remove the element of height 1. If deleted
        key in element of height > 1. Swap with its predecessor of height 1 and remove
        its predecessor; reset min's and max's for all nodes in the stack; lower the head
        if necessary;
31      return (success);
32  }
```

Algorithm 3.6 Deletion of a node from a k-d Search Skip List.

**Theorem 3.6** The time D(n, k) required to delete a point from a k-d search skip list is O($k$log $n$).

Proof

As for insertion, the proof depends on the longest path length encountered on deletion. Deletion always starts at the head node, and proceeds via a single path to the bottom level, whereupon the existing point is deleted. Along the way, gap heights are possibly changed by borrowing from a next or preceding gap, and then resetting minimum and maximum $K_2$ , ..., $K_k$ values. The borrowing requires the changing of one pointer. Resetting minimum and maximum $K_2$ , ..., $K_k$ values is done by a call to the function *getmin, getmax*, which compares the values of up to two other nodes, and resets if appropriate. The time for these operations is considered a constant.

The longest path length encountered on deletion for the k-d search skip list is given in Lemma 1 as O(log $n$). Following this path to the bottom, and deleting the node thus requires O(log $n$) time. In addition, the deletion path is followed again to check for deleted $K_1$ keys in nodes higher than the bottom. This also requires O(log $n$) time. Finally, the deletion path is followed from the bottom up to reestablish the correct minimum and maximum $K_2$ , ..., $K_k$ values, which requires O($k$log $n$) time. The total time required is thus 2O(log $n$) + $k$ O(log $n$)= O($k$log $n$) time. **QED.**

# Chapter 4
# LOWER BOUNDS

In order to formulate a lower bound, we have to define an appropriate model of computation, the so-called *tree model*. In this model, a data structure is a rooted tree, and a condition is associated with each edge in the tree. Given a query, the query answering algorithm starts with the root, and visits a vertex v if and only if the given query satisfies the conjunction of the conditions on the path from the root to v. The output corresponding to a given query is a function of the data associated with the visited vertices. Several standard data structures, such as linked lists, and range trees are based on the *tree model*. In the *tree model*, we investigate the semi-infinite reporting problem where the response to a query is a list of all the records in the data base whose keys are located in the query box.

The lower bound for the worst case time to process (on-line) a sequence of n intermixed insertions, deletions, and range queries is $\Omega(n(\log n)^k)$ Fredman [1981].

It is convenient to transform the problem of semi-infinite range searching in a k-dimensional set F of n points with arbitrary real coordinates into the problem of range searching in a k-dimensional set F′ of n points with integer coordinates between 1 and n.

Such "normalization" can be carried out as follows. Let $F_i = \{x_i \mid x \in F\}$ $(1 \le i \le k)$ be the set of n numbers occuring as the i-th coordinate. For each point $x = (x_1, x_2, ..., x_k)$ of F take a point $x' = (x_1', x_2', ..., x_k')$ into F′, where $x_i'$ is the "rank" of $x_i$ in $F_i$. (If the numbers of $F_i$ are sorted into ascending order and duplicates are removed, then the position of $x_i$ in this sequence is its rank). Note that such normalization can be accomplished in time

O(knlog $n$) and space O(n). A range query $[L_1, H_1]$, $[L_2, \infty]$, ..., $[L, \infty]$ in F can be "normalized" into a range query $[L_1', H_1']$, $[L_2', \infty]$, ..., $[L_k', \infty]$ in F′ in a similar fashion in (k + 1) log $n$ comparisons.

For the above reasons, we can assume in the following that F is a set of n points in k dimensions with all coordinates being integers between 1 and n, and that each range query $[L_1, H_1]$, $[L_2, \infty]$, ..., $[L_k, \infty]$ consists of integers only with $1 \le L_i \le n$, $1 \le H_i \le n$, for i = 1, 2, ..., k.

## 4.1 No Space Restriction

For an arbitrary point set F, let R(F) be the number of different semi-infinite range queries possible for F (we assume that two range queries are different iff the results of a range search based on them are different).

Let $R(n, k) = \max\{R(F) | F \text{ is a set of n points in k dimensions}\}$.

It is easy to see that $R(n, 1) = \binom{n+1}{2} + 1 = (n+1)n/2 + 1$, for the answer to a range query is either empty (one such answer) or can be defined by two of n+1 interpoint locations. It is also easy to see that $R(n, 2) = R(n, 3) = \ldots = n + 1$.

The exact value of R(n, k) for general n and k seems more difficult to calculate. We can immediately observe that $R(n, k) \le [(n+1)n/2 + 1][n + 1]^{(k-1)}$.

**Lemma 4.1** $R(n, k) \ge (n/(2k))^{(k+1)}$.

Proof

To avoid complications assume n is a multiple of 2k. Let S be the set of all points

with a single nonzero integer coordinate in the closed interval [-n/(2k), n/(2k)]. Consider the set of all range queries $[L_1, H_1]$, $[L_2, \infty]$, ..., $[L_k, \infty]$ with $-n/(2k) \le L_i \le -1$ and $1 \le H_1 \le n/(2k)$, for i=1, ..., k. The $(n/(2k))^{(k+1)}$ range queries obtained in this way clearly determine different sets of points, and the result follows. **QED.**

**Theorem 4.1** For range queries on n points in k dimensions, the lower bound for semi-infinite range search time is $\Omega(k \log n)$.

Proof

By the Lemma 4.1, $R(n, k) \ge (n/(2k))^{(k+1)}$. Hence any algorithm for semi-infinite range search based on decision trees in the worst case requires at least

$$\lg (n/(2k))^{(k+1)} = (k+1) \log n - (k+1)\log k - (k+1)\log 2$$

comparisons. Hence the lower bound for semi-infinite query time is $\Omega(k \log n)$. **QED.**

## 4.2 With Space Restriction

Space and time requirements have traditionally been used as performance measures for the algorithms of range search. We investigate the question of (storage) space-(retrieval) time trade-off for semi-infinite range queries on a static data base.

Let k be a positive integer. Let $N = \{1, 2, ..., n\}$ and let $N^k$ denote the set of all k-tuples of positive integers less than or equal to n. Let $(K_1, K_2, ..., K_k) \in F$. A semi-infinite range query is specified by a (k + 1) - tuple $(L_1, H_1, L_2, \infty, L_3, \infty, ..., L_k, \infty)$ of positive integers satisfying $L_1 < H_1$. Alternately, the query region for a semi-infinite range search is a parallelepiped (box) b, defined by the product [L , H ] × [ L

of k intervals. A key k is said to be located in a box $b = [L_1, H_1] \times [L_2, \infty) \times [L_3, \infty) \times \ldots \times [L_k, \infty)$ ($b = [L_1, H_1] \times [L_2, maxK_2) \times [L_3, maxK_3) \times \ldots \times [L_k, maxK_k)$) if and only if $L_1 \le K_1 \le H_1$, $L_i \le K_i \le \infty$, $2 \le i \le k$. We consider such a query where the output is a list of all the records whose keys lie in the query parallelepiped b.

We study the tree model for data structures. In this model, the data structure is assumed to be a rooted tree.

We shall only consider sets of records where no two records in a set have the same key. Then the set F of keys completely specifies the set of records, and the semi-infinite reporting problem is to produce a list of all the keys in F that lie in the given query box. Note that considering this special case does not cause any loss of generality as the lower bounds obtained in the special case trivially extend to the general case. Since the output size is query dependent, the time required to answer a query is not the correct measure of the overhead involved in producing the desired response to the query. So we define a scaled query time T′ that measures the overhead for producing one unit of output. With respect to a fixed set F of keys, and a fixed tree for F, we define a scaled query time T′ as follows:

$$T' = \max_{1 \le |b \cap F| \le \log n} \frac{T(b)}{|b \cap F|} \qquad (4.1)$$

**Theorem 4.2** In the tree model, for the semi-infinite reporting problem on a static data base with n records, there is a space-time trade-off (where S = space, T = time)

$(\log T' + \log\log n)^{k-1}T'S = \Omega(n(\log n)^{k-\theta})$, where $\theta = 1$ for $k = 2$,

and $\theta = 2$ for $k \geq 3$.

Proof

From Vaidya [1989], the orthogonal reporting problem on a static data base with n records, the space-time trade-off is $(\log T' + \log\log n)^{k-1}T'S = \Omega(n(\log n)^{k-\theta})$, where $\theta = 1$ for $k = 2$, and $\theta = 2$ for $k \geq 3$. Again, the lower bound of T'S for Up semi-infinite range search is the same as that for Down semi-infinite range search; the lower bound of T'S of the Up semi-infinite range search, the Down semi-infinite range search, and the intersection between the Up semi-infinite range search and Down semi-infinite range search would be that for the orthogonal range search; and the lower bound of T'S of Down semi-infinite range search is the same as that in the orthogonal case. Therefore, the space-time trade-off for the semi-infinite reporting problem is $(\log T' + \log\log n)^{k-1}T'S = \Omega(n(\log n)k-\theta)$, where $\theta = 1$ for $k = 2$, and $\theta = 2$ for $k \geq 3$. **QED.**

This type of lower-bound argument, when used to prove a worst-case result, is sometimes known as an *information-theoretic* lower bound. The general theorem says that if there are P different possible cases to distinguish, and the questions are of the form YES/NO, then $\lceil \log P \rceil$ questions are always required in some case by any algorithm to solve the problem Weiss [1992].

The most obvious open problem left by this work is that of further tightening the bounds. We suspect that the above lower bound for semi-infinite range search without space restriction (Theorem 4.1) is exact up to second-order terms.

# Chapter 5

# HALF-SPACE RANGE SEARCH

When we discuss half-space range search, if we don't specifically mention it, we search the points locate in the positive side of a hyper plane h. The hyper plane h has the equation: $a_1x_1 + a_2x_2 + ... + a_kx_k = m$, where we always assume $a_1 \neq 0$.

To perform a half-space range search, we modify the composition of a single node in a k-d Search Skip List as follows:

| datapoint | |
|---|---|
| min. $K_1$ value in the left subtree | max. $K_1$ value in the left subtree |
| min. $K_2$ value in the left subtree | max. $K_2$ value in the left subtree |
| . . . | . . . |
| min. $K_k$ value in the left subtree | max. $K_k$ value in the left subtree |
| down pointer | right pointer |

Figure 5.1 Modified composition of a single node in a k-d Search Skip List to allow half-space range search.

The difference between the composition of a single node in a k-d Search Skip List and the modified one is that we add the minimum and maximum of all k coordinate values in the left subtree into the node representation.

A sample of a modified k-d Search Skip List (see Figure 5.2).

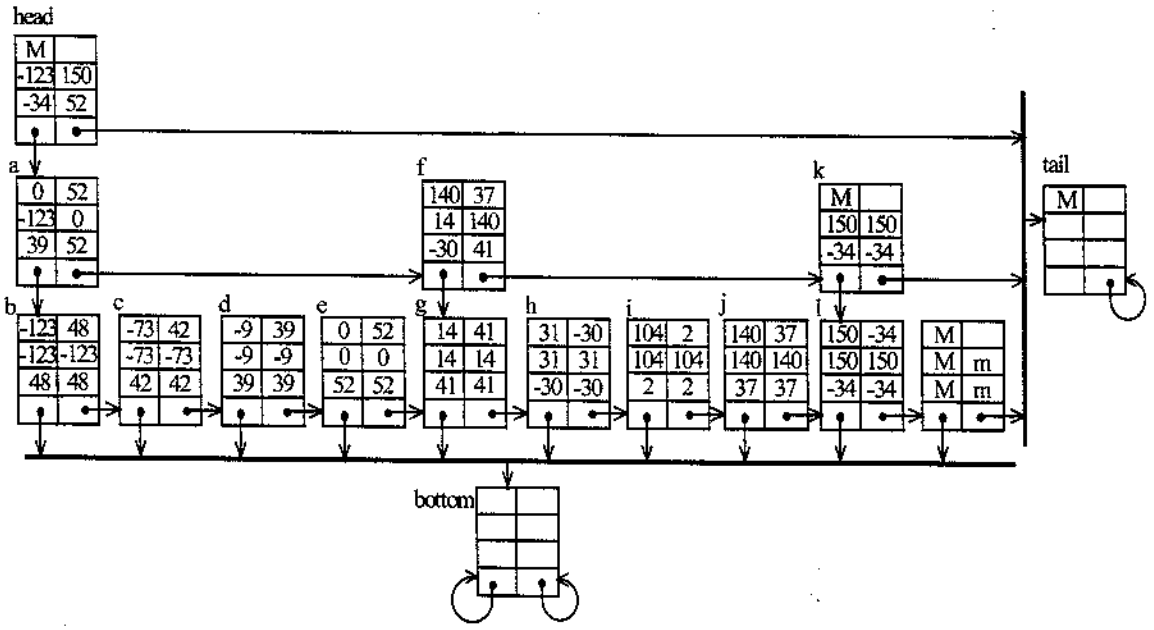


Figure 5.2 Sample modified k-d Search Skip List.

Note that M stands for the maximum coordinate value and m stands for the minimum coordinate value. The down and right pointers point to the left and right subtrees, respectively. The algorithms for insertion of points into and deletion of points from the modified k-d Search Skip List are not given here as they are similar to the Search Skip List algorithms. Special nodes head, tail and bottom are used in these algorithms to mark the beginning and end of data. Each leaf node contains one datapoint; these are maintained in order of 1st coordinate values. Interior nodes have a datapoint equal to that of the rightmost leaf level data point in the left subtree of this interior node.

## 5.1 Half-Space Range Search in The Planar Case

### 5.1.1 The Half-Space Range Search Algorithm

Let h: $ax + by - c = 0$ ($a \neq 0$) be a hyper plane in $E^2$, F be a subset of $E^2$.

For $b \neq 0$, let $y' = (-ax_{min} + c)/b$, $y'' = (-ax_{max} + c)/b$ (see Figure 5.3). Similarly, let $x' = (-by_{min} + c)/a$, $x'' = (-by_{max} + c)/a$.

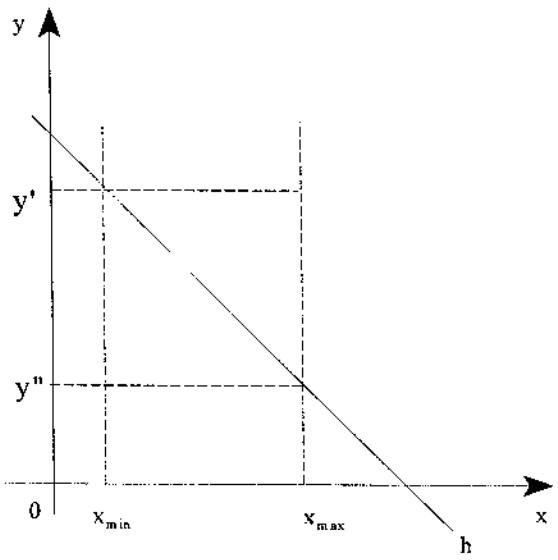


Figure 5.3 Half-space range search for the 2-dimensional Search Skip List.

We assume that there is no point lying on the query line h.

**Lemma 5.1** For any point $(x, y) \in F$, we have

(i) if $b > 0$, then $y < \min(y', y'')$ implies $ax + by - c < 0$, and

$y > \max(y', y'')$ implies $ax + by - c > 0$.

(ii) if $b < 0$, then $y < \min(y', y'')$ implies $ax + by - c > 0$, and

$y > \max(y', y'')$ implies $ax + by - c < 0$.

(iii) if $a > 0$, then $x < \min(x', x'')$ implies $ax + by - c < 0$, and

$x > \max(x', x'')$ implies $ax + by - c > 0$.

(vi) if $a < 0$, then $x < \min(x', x'')$ implies $ax + by - c > 0$, and

$x > \max(x', x'')$ implies $ax + by - c < 0$.

Proof

If $b \neq 0$, for any point $(x, y) \in F$, let $y_0 = (-ax + c)/b$, $y_1 = \min(y', y'')$, $y_2 = \max(y', y'')$. By Figure 5.3, we have $y_1 \leq y_0 \leq y_2$.

(i) Let $b > 0$. If $y < y_1$, then

$$y < y_1 \leq y_0 = (-ax + c)/b,$$

or

$$by < -ax + c,$$

that is

$$ax + by - c < 0.$$

If $y > y_2$, then

$$y > y_2 \geq y_0 = (-ax + c)/b,$$

or

$$by > -ax + c,$$

that is

$$ax + by - c > 0.$$

(ii) Let $b < 0$. If $y < y_1$, then

$$y < y_1 \le y_0 = (-ax + c)/b,$$

or

$$by > -ax + c,$$

that is

$$ax + by - c > 0.$$

If $y > y_2$, then

$$y > y_2 \ge y_0 = (-ax + c)/b,$$

or

$$by < -ax + c,$$

that is

$$ax + by - c < 0.$$

(iii), (vi) are Similar to case (i), (ii). **QED.**

From Lemma 5.1, we can get the 2-d half-space range search (top down) algorithm for our k-d Search Skip List:

1. Start at the head node of the Search Skip List.

2. Calculate x' and x".

   If ($a > 0$ and $x_{max} < \min(x', x")$) or ($a < 0$ and $x_{min} > \max(x', x")$)

      we prune this left subtree, exit.

   Else if ($a > 0$ and $x_{min} > \max(x', x")$) or ($a < 0$ and $x_{max} < \min(x', x")$)

      we report this left subtree, exit.

   Else (check b)

      If ($b \neq 0$), calculate y' and y".

If ($b > 0$ and $y_{max} < \min(y', y'')$) or

($b < 0$ and $y_{min} > \max(y', y'')$)

we prune this left subtree, exit.

Else if ($b > 0$ and $y_{min} > \max(y', y'')$) or

($b < 0$ and $y_{max} < \min(y', y'')$)

we report this left subtree, exit.

Else go to Step 3.

Else go to Step 3.

3. Following the down pointer or the right pointer, recursively go to Step 2.
4. When we reach the bottom level, we check whether the remaining points fall in the given half-space.

### 5.1.2 Analysis of The Half-Space Range Search for The 2-d Search Skip List

**Theorem 5.1** Given a set of n points in $E^2$, one can preprocess it for half-space range reporting in time $O(n \log n)$, storing the data structure in space $O(n)$.

**Theorem 5.2** Given a set of n points in $E^2$, the worst case half-space range search time is $O(n)$.

We omit the proof of these theorems, since it follows the proof for the k-d Search Skip List in Section 5.2.

## 5.2 Half-Space Range Search in $E^k$, When k > 2

### 5.2.1 The Half-Space Range Search Algorithm

In last section, we discussed half-space range search in $E^2$. The purpose of this section is to generalize the results for $E^2$ case to the general case $E^k$, where k is any positive integer greater than 1.

Although our arguments are developed in $E^k$, for ease of presentation, all Figures will represent 2-dimensional analogs. Given a hyper plane h of equation $AX^T = m$, where vector $A = (a_1, a_2, ..., a_k)$, $X = (x_1, x_2, ..., x_k)$, and m is a constant. Given a set of points F in $E^k$, we want to find which points in F are in the positive half-space of h. We assume that no point lies on the query plane h.

We would like to introduce several definitions first.

**Definition 5.1** We say $p = (x_1, x_2, ..., x_k)$ belongs to the positive half-space of h, denoted $p \in h^+$, if $A (x_1, x_2, ..., x_k)^T - m > 0$. We define the negative half-space of h, $h^-$, in a similar fashion by reversing the inequality in above.

**Definition 5.2** For a given set F in $E^k$, and any integer i, $1 \le i \le k$, $x_{imin}$ is defined as the minimum value of the $x_i$ coordinates of all the points in F. Similarly, $x_{imax}$ is defined as the maximum value of the $x_i$ coordinates of all the points in F.

**Definition 5.3** For a given set F in $E^k$, we define two k-element arrays, mins and maxs: For any integer i, $1 \le i \le k$,

$$\min s[i] = \begin{cases} -a_i x_{i\min}, & if \quad a_i < 0 \\ 0, & if \quad a_i = 0 \\ -a_i x_{i\max}, & if \quad a_i > 0 \end{cases}$$

$$\max s[i] = \begin{cases} -a_i x_{i\max}, & if \quad a_i < 0 \\ 0, & if \quad a_i = 0 \\ -a_i x_{i\min}, & if \quad a_i > 0 \end{cases}$$

**Definition 5.4** For a given set F in $E^k$, we define two discrete functions, $f_{mins}$ and $f_{maxs}$, defined on the set {1, 2, ..., k}: For any integer i, 1≤ i≤ k,

$$f_{\min s}(i) = \sum_{j \neq i} \min s[j],$$

$$f_{\max s}(i) = \sum_{j \neq i} \max s[j].$$

**Lemma 5.2** Let F be a subset of $E^k$. Then for any point $p = (x_1, x_2, ..., x_k) \in F$, we have

$mins[i] \le -a_i x_i \le maxs[i]$, $i = 1, 2, ..., k$.

Therefore

$f_{mins}(i) \le f_{maxs}(i)$, $i = 1, 2, ..., k$.

Proof

Since for $1 \le i \le k$, $x_{imin} \le x_i \le x_{imax}$, if $a_i < 0$, we have

$$-a_i x_{imin} \le -a_i x_i \le -a_i x_{imax}.$$

That is $mins[i] \le -a_i x_i \le maxs[i]$.

Similarly, if $a_i \ge 0$, we have

$$-a_i x_{imin} \ge -a_i x_i \ge -a_i x_{imax}.$$

That is

$$mins[i] \le -a_i x_i \le maxs[i].$$

Similarly, if $a_i \ge 0$, we have

$$-a_i x_{imin} \ge -a_i x_i \ge -a_i x_{imax}.$$

That is

$$maxs[i] \ge -a_i x_i \ge mins[i].$$

By above and Definition 5.4, it is not difficult to get

$$f_{mins}(i) \le f_{maxs}(i),\ i = 1, 2, ..., k. \quad \textbf{QED.}$$

We assume that there is no point lying on the query plane h.

**Lemma 5.3** Let h: $AX^T = m$ be a hyper plane in $E^k$, F be a subset of $E^k$ and i be an integer between 1 and k. If $a_i \ne 0$, let

$$x'_i = (f_{mins}(i) + m)/a_i,\ x''_i = (f_{maxs}(i) + m)/a_i,$$

then for any point $p = (x_1, x_2, ... x_k) \in F$,

(1) if $a_i > 0$, then $x_i < x'_i$ implies $p \in h^{\#}$,

and $x_i > x''_i$ implies $p \in h^*$,

(2) if $a_i < 0$, then $x_i < x''_i$ implies $p \in h^*$,

and $x_i > x'_i$ implies $p \in h^{\#}$,

Proof

Given $1 \le i \le k$, if $a_i \ne 0$, by Lemma 5.2, we have $a_i x'_i \le a_i x''_i$.

For any point $p = (x_1, x_2, ... x_k) \in F$,

let

$$x_{i0} = (\sum_{j \ne i} (-a_j * x_j) + m) / a_i,$$

again by Lemma 5.2, we have

$x'_i \le x_{i0} \le x''_i$.

(1) Let $a_i > 0$, if $x_i < x'_i$, then

$$x_i < x'_i \le x_{i0} = (\sum_{j \neq i}(-a_j * x_j) + m) / a_i,$$

or

$$a_i * x_i < \sum_{j \neq i}(-a_j * x_j) + m,$$

or

$$A(x_1\ x_2\ \ldots\ x_k)^T - m < 0.$$

Hence

$$p \in h^{\#}.$$

If $x_i > x''_i$, then

$$x_i > x''_i \ge x_{i0} = (\sum_{j \neq i}(-a_j * x_j) + m) / a_i,$$

or

$$a_i * x_i > \sum_{j \neq i}(-a_j * x_j) + m,$$

or

$$A(x_1\ x_2\ \ldots\ x_k)^T - m > 0.$$

Hence

$$p \in h^*.$$

(2) Let $a_i < 0$, if $x_i < x''_i$, then

$$x_i < x''_i \le x_{i0} = (\sum_{j \neq i}(-a_j * x_j) + m) / a_i,$$

or

$$a_i * x_i > \sum_{j \neq i} (-a_j * x_j) + m,$$

or

$A(x_1\ x_2\ \ldots\ x_k)^T - m > 0.$

Hence

$p \in h^*.$

If $x_i > x'_i$, then

$$x_i > x'_i \geq x_{i0} = (\sum_{j \neq i} (-a_j * x_j) + m) / a_i,$$

or

$$a_i * x_i < \sum_{j \neq i} (-a_j * x_j) + m,$$

or

$A(x_1\ x_2\ \ldots\ x_k)^T - m < 0.$

Hence

$p \in h^{\#}$. **QED.**

From Lemma 5.3, we can get the k-d half-space range search (top down) algorithm for our k-d Search Skip List:

1. Start at the head node of the Search Skip List.

2. Calculate two arrays: mins and maxs.

Let

$$h\ \mathrm{min}\ s = m + \sum_{i=1}^{k} \mathrm{min}\ s[i],$$

$$h\max s = m + \sum_{i=1}^{k} \max s[i].$$

For each $a_i \neq 0$, where $i \in \{1, 2, ..., k\}$, Calculate $x'_i$ and $x''_i$:

$x'_i = (\text{hmins} + \text{mins}[i])/a_i$, $x''_i = (\text{hmaxs} + \text{maxs}[i])/a_i$.

If ($a_i > 0$ and $x_{imax} < x'_i$) or ($a_i < 0$ and $x_{imin} > x'_i$)

we prune this left subtree, break the loop.

Else if ($a_i > 0$ and $x_{imin} > x''_i$) or ($a_i < 0$ and $x_{imax} < x''_i$)

we report this left subtree, break the loop.

Else

If it is the last loop $a_k$ case,

go to Step 3.

Else increase i to discuss next $a_i$ case.

3. Following the down pointer or the right pointer, recursively go to Step 2

4. When we reach the bottom level, we check whether the remaining points fall in the given half-space.

```
// Perform half-space (positive side)range search on k-d search tree
//constructed using a deterministic skip list. i.e. report all points
// lying inside the positive side of a hyper plane

1     template<class T>
2     void KDSSkipList<T>::halfSpSearch(const KDSSkipNode<T>& startNode,
            const T halfSp[k+1], const T& maxFirst, long& numFound) const
3     {
4         if ((startNode.keys[0][0] == maxkey) && (startNode.down== bottom)) return;
5         else if ((startNode.keys[0][0] == maxkey) &&
                  (startNode.down != bottom) && (&startNode != tail))
6         {
7             HalfSpSearch(*startNode.down, halfSp, startNode.keys[0][0],
                                                                  numFound);
8         }
9         else if ((&startNode != bottom) && (&startNode != tail))
10        {
11            if (startNode.down == bottom)
12            {
13               if this point in the given side, report it;
14               if (startNode.keys[0][0] < maxFirst)
15                  HalfSpSearch (*startNode.right, halfSp, maxFirst, numFound);
16            }
17            else   //not leaf level
18            {
19               check to report or prune down;
20               if report
21                  report this left subtree;
22               else if (!prune)
23                   halfSpSearch(*startNode.down, halfSp, startNode.keys[0][0],
                                 numFound);
24               if (startNode.keys[0][0] < maxFirst)
25                  halfSpSearch(*startNode.right, halfSp, maxFirst, numFound);
26            }
27        }
28    }
```

Algorithm 5.1 C++ pseudo-code algorithm for k-d half-space range search

### 5.2.2 Analysis of The Half-Space Range Search for The k-d Search Skip List

**Theorem 5.3** Given a set of n points in $E^k$, one can preprocess it for half-space range reporting in time $O(kn \log n)$, store the data structure in space $O(kn)$.

We omit the proof of this theorem, since it closely follows the method achieving a similar result for the k-d Search Skip List with orthogonal range search in Chapter 3.

**Theorem 5.4** Given a set of n points in $E^k$, the worst case half-space range search time is $O(kn)$.

Proof

The worst case for k-d half-space range search is illustrated in Figure 5.6. In this case, there are no points in range, but the half-space range search algorithm must search $n$ nodes to make this determination. The points are organized in the $x_{1min} < x_1 < x_{1max}$, and $x_j' < x_j < x_j''$, where j = 2, 3, ..., k. Thus, the algorithm never prune as $x_j$ always in the range $(x_j', x_j'')$, until the bottom level is reached. This means that O(n) nodes are checked to determine that no points are in the desired side, and each node requires O(k) operations. Therefore, the search time for half-space range search is $O(kn)$. **QED.**

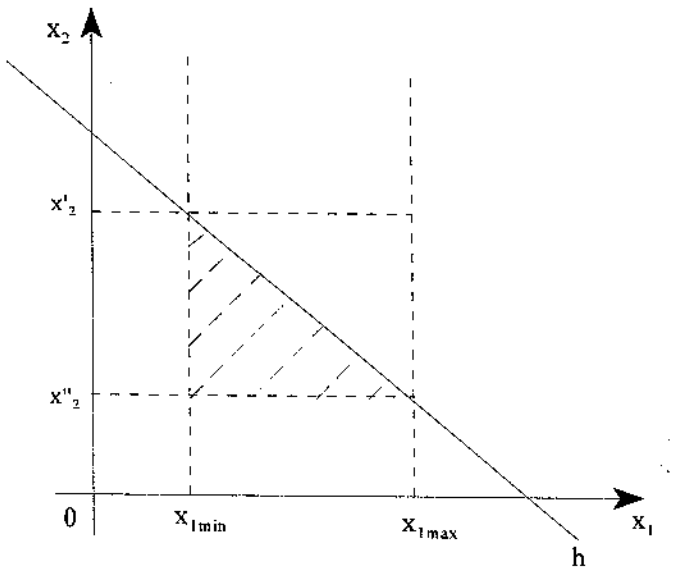


Figure 5.4 Illustration of all data points are in the cross-hatched area for 2-d half-space with a k-d Search Skip List.

# Chapter 6
# EXPERIMENTAL RESULTS

## 6.1 k-d Range Search and Semi-Infinite k-d Range Search

The k-d Search Skip List was experimentally analyzed using the methodical approach described below, in an attempt to facilitate an accurate comparison of their actual run times. We are interested in the average insertion time, the average deletion time, and the average search time as the worst-case is not only rare for large structures but difficult to specify. In this section we outline the general approach that is used in the experimental analysis of the k-d Search Skip List in Section 6.2.

The structure was tested on randomly generated data sets which were uniformly distributed throughout a k-dimensional space. The random data was generated by using the built in function lrand48(). Since we instantiate template T as *long*, lrand48() can only generate the random numbers between 0 and $2^{31}$ - 1. We chose sample sizes n of 1,000, 10,000, and 100,000. The range for the random numbers are shifted such that they are between $-(2^{31} - 1)/2$ and $(2^{31} - 1)/2$. The structure was analyzed based on their construction times, the destruction times, and range search times. It is the construction times and destruction times that allow us to approximate the "hidden" constants of our big-oh analysis of the insertion algorithms that we performed in Chapter 3. Our analysis of the update times is based on these constants as they allow us to make a direct comparison of the structures.

The construction time is defined as the time to build a structure by inserting $n$ points

sequentially into an initially empty structure. The destruction time is defined as the time to dispose of a structure by deleting one data point at a time until we are left with an empty structure. The range search time is simply the time to execute a k-d range search for a given k-d orthogonal range query. The semi-infinite range search time is the time to execute a k-d range search for a given k-d semi-infinite range search query.

The construction and destruction times were recorded by running the programs (kdskiplist.cc and kdtest.cc) 270 times. Each time the application reported the unsuccessful insertions which was caused by inserting a node whose first dimensional key was the same as that of an existing node in the k-d skip list.

The range search time is recorded for 6 different query windows that cover 0%, 10%, 30%, 50%, 70%, 100% of the search space, respectively. Given a query window area ratio, which is the ratio of the area of the query window to that of the data window, we can get the side length ratio which is (area ratio)$^{1/k}$, then call the function *getQueryWin* by using this side length ratio to get the required query window. The position of this query window is randomly determined. The query windows are restricted such that they are completely contained within the search space inclusively. Algorithm 6.1 was used to generate the random query windows.

```
void getQueryWin(int qwMinR[k], int qwMaxR[k],  int wMinR[k], int wMaxR[k],
                 int keys[SAMPLESIZE][k], float qwRatio[k])
// qwRatio[i]: the ratio of the i-th side of the query window and
//   its corresponding side of the whole window.
{
    unsigned i=0, j=0;     // loop index
    int tempMin, tempMax, temp;
    // Get the whole window
    for (i=0; i<k; i++)
    {
        tempMin = tempMax = keys[0][i];
        for (j=1; j<SAMPLESIZE; j++)
        {
            temp = keys[j][i];
            if (tempMin > temp)
                tempMin = temp;
            if (tempMax < temp)
                tempMax = temp;
        }
        wMinR[i] = tempMin;
        wMaxR[i] = tempMax;
    }
    // Get the portions randomly
    int wSideLen;   // length of side of the whole window
    int qwSideLen;  // length of side of the query window
    for (i=0; i<k; i++)
    {
        // Get the length of each side in integer
        wSideLen = wMaxR[i]-wMinR[i];
        qwSideLen = wSideLen * qwRatio[i];
        if (wSideLen != qwSideLen)
        {
            qwMinR[i] = (rand() % (wSideLen-qwSideLen)) + wMinR[i];
            qwMaxR[i] = qwMinR[i] + qwSideLen;
        }
        else
        {
            qwMinR[i] = wMinR[i];
            qwMaxR[i] = wMaxR[i];
        }
    }
}
```

Algorithm 6.1 Random query window generation.

For semi-infinite range search, it is impossible to specify the search area ratio in advance if you want the position of every side of the query window to be generated randomly. In our test, we only give the side length ratio for the first dimension. We then create our semi-infinite query window (see Figure 6.1), whose first dimension side position is randomly chosen but whose length (e.g. *l* in Figure 6.1) corresponds to the calculated side length ratio. Each remaining side has only one endpoint's position (e.g. a and b in Figure 6.1) which is also randomly created. After we get the query window, we then calculate the area ratio and report it.

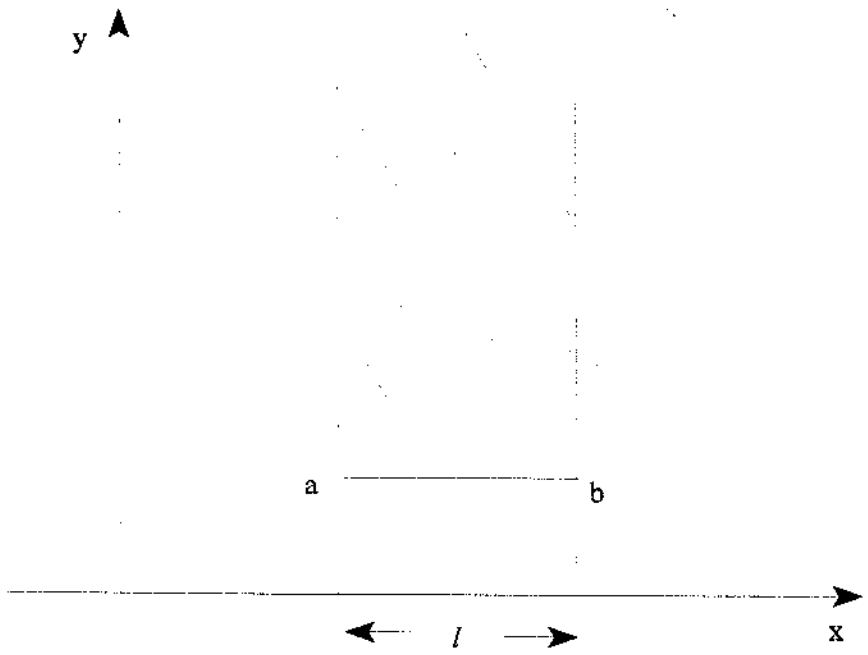


Figure 6.1 The query window of 2-d semi-infinite range search.

To allow for an accurate analysis, each structure is constructed, searched for the six different query windows, and destroyed a total of five times. Also, for each structure that is built, each range search is performed 270 times in order to bring the search time into the high millisecond range so that accurate timings may be obtained. A single range query is extremely fast, requiring only a few milliseconds for the largest structures we build.

Five test runs were performed on 2, 5, and 10 dimensional data for data sets that ranged in size from 1,000 to 100,000 data points. A single test run consisted of 5 passes over 9 data structures. Each run tests constructing, destroying, and searching a single data structure.

Times were obtained by using the built-in function *clock*, accurate to at least $10^{-6}$ of a second, which keeps track of the amount of system time used. All of our times are given in milliseconds except the search times are in microseconds when the query window area ratio is 0 or very small (see Table 2). Due to the size of $nk\log n$ as compared to the actual run times, the constant for building the data structures are calculated (see Table 1) according to the following equation:

$$c = \frac{average\_run\_time}{1milli\,second} * \frac{100{,}000}{nk\log n} = \frac{average\_run\_time}{nk\log n / 100{,}000} \qquad (6.1)$$

Table 1 The value of ($nk\log n$)/100,000 for different tests.

| k | Number of data points in the randomly generated datasets | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 0.20 | 2.66 | 33.22 |
| 5 | 0.50 | 6.64 | 83.05 |
| 10 | 1.00 | 13.29 | 166.10 |

Approximating the constant for the range search cost function, which is $ck\log n + t$, is difficult. The range search function is highly dependent on the time needed to report the t data points in range which is difficult to separate from the time required to search the skip list.

The method we use for analyzing the search times is to determine whether or not the search times increase appropriately with an increase in the size of the query window. For example, when the query window increases from 10% to 30% coverage, the search time should approximately triple if the search is performed on the same structure under the assumption that t dominates the search time (which our results seem to indicate).

Our methodology is to average the construction, and search times for each structure built (where one structure corresponds to one data file) and determine our constants and approximate run times from these averages. The results are summarized in the section that follows. Appendix B, D, and E contain the complete testing details.

The testing was performed on *sol*, a Sun SPARC 1000 workstation, under SUN OS 5.3. *Sol* has 640 MB of RAM and 820 MB of hard disk space available for use as virtual memory. *Sol* is located in the computing Services Department of the University of New

Brunswick and is a two-processor multi-user system which runs at 50 MHZ. *Sol* is heavily used throughout the campus and testing took place on May 20, 1997, when usage was usually below normal.

The C++ pseudo-code for the test runs is shown in Algorithm 6.2. It contains the pseudo-code for the main driver used in testing the k-d Search Skip List along with pseudo-code for the driver functions to accomplish the construction, orthogonal range searching, semi-infinite range search and destruction of the data structures. The complete source code for the k-d Search Skip List can be found in the appendix H.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <sys/time.h>    // library routines
#include "kdskiplist.h"

#define SAMPLESIZE     100000L
#define QWAREARATIO    0.8
#define MILLISEC_PER_SEC    1000

main()
{
     long   startT, endT;
     long    i, keysA[SAMPLESIZE][k];
     int    j;
     long*   dataIter = NULL;  // pointer iterator of the array keysA
     const long maxTest = 536870912L;

     for  (i=0L; i<SAMPLESIZE; i++)
     {
         for (j=0; j<k; j++)
               randomly generate SAMPLESIZE k-dimensional data points;
     }

     KDSSkipList<long>      kdlist;         // Create an empty skiplist

     BUILD the structure and output the construction time;
     ORTHOGONAL RANGE SEARCH the structure for the given query windows and
          report the data points in range and the search time;
     SEMI-INFINITE RANGE SEARCH the structure for the one given side of query
          windows and report the data points in range and the search time;
     DESTROY the structure and output the destruction time;
}
```

Algorithm 6.2 (a) Main driver routine for testing the k-d Search Skip List.

Sub-driver BUILD

```
long unsucNum = 0L;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
dataIter = keysA[0];
for (i = 0L; i < SAMPLESIZE; i++)
{
   if (0 == kdlist.insert(dataIter))
      UnsucNum++;
   DataIter += k;
}
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output unsucNum and endT-startT;
```

Sub-driver DESTROY

```
      long numFailDel = 0L;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
dataIter = &keysA[0][0];
for (i=0L; i<SAMPLESIZE; i++)
{
    if (kdlist.deleteNode(*dataIter) == 0)
       NumFailDel++;
   DataIter += k;
}
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output numFailDel and endT-startT;
```

Sub-driver ORTHOGONAL RANGE SEARCH

```
long   qwMinR[k], qwMaxR[k];
long   wMinR[k], wMaxR[k];
float  quRatio[k];
long   numfound = 0L;
call getQueryWin(...);
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
search the data structure for the
    given query window;
endT = clock();
output numFound and endT-startT;
```

Sub-driver SEMI-INFINITE RANGE SEARCH

```
numFound = 0L;
calculate the query window area ratio
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
search the data structure for calculated
  semi-infinite query window;
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output numFound and endT-startT;
```

Algorithm 6.2 (b) Sub-driver routines for testing the k-d Search Skip List.

## 6.2 An Analysis of the k-d Search Skip List

In this section we experimentally analyze the k-d Search Skip List of chapter 3. The construction and destruction times for all five passes of all six runs for randomly generated datasets can be found in Appendix A. Their average construction and destruction time can be found in Appendix B. Table 2 and Table 3 below summarize the construction and destruction times for the test runs.

The constants, as computed by equation (6.1), corresponding to the construction and destruction times of Appendix A can be found in Appendix C and the constants corresponding to the average construction and destruction times of Appendix B can be found in Appendix D and are summarized in Tables 4 and 5.

Table 2
Construct time averages for all test runs the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 44 | 542 | 8788 |
| 5 | 78 | 946 | 14051 |
| 10 | 127 | 1760 | 23071 |

Table 3
Destruct time averages for all test runs of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 57 | 774 | 13669 |
| 5 | 84 | 1373 | 20807 |
| 10 | 137 | 2324 | 32774 |

Table 4
Average insertion constants for all test runs of the k-d Search Skip List

| k | Constant c for insertion averages | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.02 | 8.98 | 14.25 |
| 5 | 5.17 | 6.11 | 9.34 |
| 10 | 4.24 | 5.89 | 7.43 |

Table 5
Average deletion constants for all test runs of the k-d Search Skip List

| k | Constant c for deletion averages | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.29 | 12.86 | 23.48 |
| 5 | 5.80 | 9.14 | 14.01 |
| 10 | 4.55 | 7.25 | 11.09 |

From Tables 4 and 5, it would appear that the constant for the insertion procedure is reasonable at around 8 and that the constant for deletion is moderately high at around 11.

Although the procedures should require the same amount of time on average as they both require the same amount of rebuilding in the worst case, it was expected that the deletion procedure might be slightly slower than the insertion procedure as it involves extra overhead and requires backtracking along the deletion path when a maximal or minimal value in a down subtree is removed. As with the k-d Search Skip List, we expect the constants to be slightly higher as larger structures require more overhead which could considerably slow down the construction times, especially when the constant paging and swapping of the process that is bound to happen in a multi-user system is taken into account.

We now turn to analyze the search times (orthogonal range search and semi-infinite range search). The average orthogonal range search times for the tests are given in Table 6. Table 6 shows the average range search time when the dimension is 2, 5, or 10. It appears as the area ratios increase, the increment ratios of range search times increase almost the same as that of the area ratios. We can conclude that the range search procedure is very efficient on average, behaving as predicted in our analysis.

Table 6

The average orthogonal range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 0% | 10% | 30% | 50% | 80% | 100% |
|---|---|---|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | | | | |
| 2 | $10^3$ | 0 | 2 | 2 | 2 | 4 | 4 |
| | $10^4$ | 0 | 8 | 22 | 38 | 38 | 50 |
| | $10^5$ | 0 | 108 | 364 | 460 | 494 | 578 |
| 5 | $10^3$ | 0 | 4 | 0 | 4 | 0 | 2 |
| | $10^4$ | 0 | 30 | 42 | 62 | 72 | 82 |
| | $10^5$ | 0 | 364 | 626 | 660 | 788 | 820 |
| 10 | $10^3$ | 0 | 2 | 4 | 6 | 10 | 10 |
| | $10^4$ | 0 | 88 | 84 | 104 | 126 | 108 |
| | $10^5$ | 0 | 696 | 994 | 1048 | 1168 | 1318 |

The classified semi-infinite range search times for the tests are given in Table 7. Table 7 (Appendix E), below, provides the average semi-infinite range search times obtained from our tests by classifying the percentage of points in range as 10% or less, 40 to 60%, and 90% or more. Examining the average search times of Table 7 leads us to conclude that the semi-infinite range search procedure is very efficient, behaving as predicted in our analysis (i.e. O($k$log $n$ + t) time on average).

Table 7

The average semi-infinite range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 10% or less | 40 to 60% | 90% or more |
|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | |
| 2 | $10^3$ | 0 | 0 | 4 |
| | $10^4$ | 4 | 16 | 38 |
| | $10^5$ | 38 | 172 | 500 |
| 5 | $10^3$ | 1 | 4 | 10 |
| | $10^4$ | 20 | 45 | 60 |
| | $10^5$ | 100 | 500 | 762 |
| 10 | $10^3$ | 4 | 10 | 10 |
| | $10^4$ | 38 | 70 | 90 |
| | $10^5$ | 180 | 800 | 960 |

From the above tables, we can see that the range search ( and semi-infinite range search) times increase as the dimension and query window area ratio increase. With the increments in query window size from 10-30%, 30-50%, 50-80%, and 80-100%, the increments of search times should be roughly 3.0, 1.6, 1.6, and 1.2 if t dominates the search time as predicted. Unfortunately, we didn't get such results because it is impossible to predict the time for reporting t. For given dimensionality k and query window area ratio, the results show that the semi-infinite range search time is lower than the range search time, behaving as predicted in our analysis.

For the orthogonal range search (semi-infinite range search) time, when the query window is very small, the search time is too little to report, even if microseconds are used

as the time unit.

## 6.3 k-d Half-Space Range Search

In this section we outline the general approach that is used in the experimental analysis of the average half-space range search times based on the modified k-d Search Skip List.

The structure was tested on randomly generated data sets which was uniformly distributed throughout a k-dimensional space. The random data was generated by using the built in function lrand48(). Since we instantiate template T as *long*, lrand48() can only generate the random number s between 0 and $2^{31}$ - 1. We chose sample sizes of 10,000, 50,000, and 100,000. The range for the random numbers are shifted such that they are between $-(2^{31} - 1)/2$ and $(2^{31} - 1)/2$. The structure was analyzed based on their range search times.

The half-space range search time is recorded for 100 different randomly generated hyper planes (using *srand48(first.tv_sec)* to get the seed of the random function *lrand48()*), values of k = 5, 7 and 10, and values of n = 10,000, 50,000, and 100,000. We run the programs (hskdlist.cc and kdtest.cc) 900 times. Each time the application reported the number of data points found in the positive side of the hyper plane.

The C++ pseudo-code for the test runs can be found in Algorithm 6.3. It contains pseudo-code for the main driver used in testing the modified k-d Search Skip List along with pseudo-code for the driver functions to accomplish the half-space range searching of the data

structures. The complete source code for the modified k-d Search Skip List can be found in Appendix H.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <sys/time.h>    // library routines
#include "hskdlist.h"

#define SAMPLESIZE     100000L
#define QWAREARATIO    0.8
#define MILLISEC_PER_SEC    1000

main()
{
     long   startT, endT;
     long    i, keysA[SAMPLESIZE][k];
     int    j;
     long*   dataIter = NULL;  // pointer iterator of the array keysA
     const long maxTest = 536870912L;

     for  (i=0L; i<SAMPLESIZE; i++)
     {
         for (j=0; j<k; j++)
               randomly generate SAMPLESIZE k-dimensional data points;
     }

     KDSSkipList<long>      kdlist;         // Create an empty skiplist

     BUILD the structure and output the construction time;
     HALF-SPACE RANGE SEARCH the structure for the given hyper plane and
          report the data points in the given side;
    DESTROY the structure and output the destruction time;
}
```

Algorithm 6.3 (a) Main driver routine for testing half-space range search algorithm.

Sub-driver BUILD

```
long unsucNum = 0L;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
dataIter = keysA[0];
for (i = 0L; i < SAMPLESIZE; i++)
{
   if (0 == kdlist.insert(dataIter))
      UnsucNum++;
   DataIter += k;
}
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output unsucNum and endT-startT;
```

Sub-driver DESTROY

```
long numFailDel = 0L;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
dataIter = &keysA[0][0];
for (i=0L; i<SAMPLESIZE; i++)
{
    if (kdlist.deleteNode(*dataIter) == 0)
       NumFailDel++;
   DataIter += k;
}
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output numFailDel and endT-startT;
```

Sub-driver HALF-SPACE RANGE SEARCH

```
long   numfound = 0L;
get the hyper plane;
print the hyper plane;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;
search the data structure for the
    given hyper plane;
endT = clock();
endT = clock()/MILLISEC_PER_SEC;
output numFound and endT-startT;
```

Algorithm 6.3 (b) Sub-driver routines for testing half-space range search algorithm.

We now analyze the half-space range search times. The average half-space range search times for the testing done here are given in Table 8. These times correspond to the increases in search time for an increase in dimension k and sample size n.

Table 8 (Appendix G), below, provides the average half-space range search times

obtained from our tests by classifying the percentage of points in range as 10% or less, 40 to 60%, and 90% or more. Examining the average search time of Table 8 leads us to conclude that the half-space range search procedure is correct and very efficient, behaving as predicted in our analysis.

Table 10
The average half-space range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 10% or less | 40 to 60% | 90% or more |
|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | |
| 5 | $10^4$ | 0 | 5.4 | 17 |
| | $5*10^4$ | 0.8 | 45 | 90 |
| | $10^5$ | 2 | 110 | 186 |
| 7 | $10^4$ | 0 | 10 | 19 |
| | $5*10^4$ | 1 | 55 | 105.4 |
| | $10^5$ | 2.6 | 130 | 213 |
| 10 | $10^4$ | 3 | 9.7 | 22 |
| | $5*10^4$ | 4.4 | 55.6 | 99.4 |
| | $10^5$ | 9 | 106.6 | 195.5 |

Appendix F and G give additional data and results on the testing which was performed on *sol*.

# Chapter 7
# CONCLUSIONS

## 7.1 Summary

We conclude that there is a k-dimensional data structure allowing dynamic updates and range search (orthogonal range search, semi-infinite range search and half-space range search) which maintains a linear storage requirement and logarithmic preprocessing cost. We label this k-d data structure as the k-d Search Skip List.

We have developed a simple, efficient algorithm and data structure for half-space range search based on the k-d Search Skip List. The dynamic update time of $O(k\log n)$ is the fastest we currently know of for a data structure supporting dynamic k-d half-space range search. In addition, this approach has an advantage of conceptual and programming simplicity compared to other implementations.

The algorithms presented here can search and report data points in a given query window or in a given side of query hyper plane. The analysis of these algorithms in the worst case has been studied. For our algorithm, the k-d orthogonal range search and k-d half-space range reporting time is $O(kn)$.

The lower bound on time for k-d semi-infinite range search on k-d data points has also been studied, both with and without restriction on the space requirement. This lower bound without space restriction was shown to be $\Omega(k\log n)$, and

there is a space-time trade-off

$(\log T' + \log \log n) = \Omega(n(\log n)^{k-\theta})$, where $\theta = 1$ for $k = 2$, $\theta = 2$ for $k \geq 3$, and $T'$ is the scaled query time defined in Chapter 4.

The algorithms and data structures reported here are likely to be useful in real applications. Our implementations of these data structures are extremely fast on average for k-d orthogonal range search, semi-infinite range search and half-space range search, even for relatively large k and n. In addition, our algorithms provide very good support for highly dynamic applications due to the guaranteed upper bound of $O(k \log n)$ time on any single insertion or deletion.

Dynamic implementations of the k-d Search Skip List are feasible and have been implemented and tested here. On average, only 1048 milliseconds was required to report 50% of the points in range for a 10-d orthogonal range search for n = 100,000; about 800 milliseconds was required to report 50% of the points in range for a 10-d semi-infinite range search for n = 100,000; and 106 milliseconds was required to report 50% of the points in range for a 10-d half-space range search for n = 100,000. The constants in the update procedures are reasonable, given the linear space requirement.

## 7.2 Future Work

In this thesis we have made a number of assumptions about our data set, it's size, including that of $k << n$ and no points falling on the hyper plane. The question is: what happens to our analysis if points do fall on the hyper plane?

The implementations of the k-d Search Skip List are extremely fast on average for k-d orthogonal range search, semi-infinite range search and half-space range search. What would be the upper bound of the average case search time?

There is a trade-off with the worst case range reporting time which now becomes O(kn). As suggested in Chazelle [1990], this leads one to ask if there is some lower bound on P(n, k)S(n, k)Q(n, k)U(n, k), for P = preprocessing time, S = storage space, Q = time to perform range searching for a given hyper plane, and U = time to perform a single update (insert or delete) operation. In our case, this product is $O(k^4n^3\log^2 n)$ (ignoring the time t for reporting). Other space/time trade-off studies (e.g. Bronnimann et al [1993]) have been carried out, and prove that if m units of storage are available, then the worst-case query time (for k-d half-space range searching) must be on the order of $(n/\log n)^{1-(k-1)/k(k-1)}/m^{1/k}$. What trade-offs exist if dynamic operations are considered?

Other questions concern that of extending the domain from k-d data points to k-d line segments. How are the range searches (orthogonal range search, semi-infinite range search and half-space range search) done for line segments? What lower bounds exist for k-d line segment range searching? Can a structure similar to the k-d Search Skip List be used for line segments?

In the section on lower bounds (Chapter 4.2), is there an algorithm to tighten the lower bound of the product of time and space in the tree model?

# REFERENCES


Agarwal, P. K. and Matousek, J., "Dynamic Half-Space Range Reporting and Its Applications". *Algorithmica*, vol. 13, 1995, pp. 325 - 345.

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

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

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

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

Bentley, J. L. and Maurer, H. A., " Efficient worst-case data structures for range searching". *Acta Informatica*, 13(2), 1980, pp. 155-168.

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

Bronnimann, H., Chazelle, B., and Pach, J., "How Hard IS Half-space Range Search?". *Discrete Comput. Geom. 10*, 1993, pp. 143-155.

Chazelle, B., "Lower bounds for orthogonal range searching, I: The reporting case". *J. ACM 37*, 1990a, pp. 200-212.

Chazelle, B. " Lower bounds for orthogonal range searching, II: The arithmetic model". *J. ACM 37*, 1990b, pp. 439-463.

Chazelle, B., and Edelsbrunner, H., "Linear Space data Structures for Two Types of Range Search". *Discrete. Comput. Geom. 2*, 1987, pp. 113 - 126.

Chazelle, B., Guibas, L., and Lee, D. T., "The power of geometric duality". *BIT*, 25(1), 1985, pp.76-90.

Chazelle, B. and Preparata, F. P., "Halfspace range search: An algorithmic application of k - sets". *Discrete Comput. Geom.* 1, 1986, 83-93.

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

Ewald, G., Larman, D. G., and Rogers, C. A., "The Directions of the Line Segments and of the r-Dimensional Balls on the boundary of a Convex Body in Euclidean Space". *Mathematika*, Vol. 17, Part 1. June, 1970, pp. 1-20.

Fredman M. L., "A Near Optimal Data Structure for a Type of Range Query Problem". *ACM*, 1979, pp. 62 - 66.

Fredman, M., "A lower bound on the complexity of orthogonal range queries". *J. Assoc. Comput. Mach.*, Vol. 28, No.4, Oct., 1981, pp.696-705.

Fredman, M., "Lower bounds on the complexity of some optimal data structures". *SICOMP* 10, 1981, pp. 1-10.

Guting, H. and Kriegel, H. P., "Multidimensional B-Tree: An Efficient Dynamic file

Structure for Exact Match Queries", proceedings of the 10th TI Annual Conference Informatik Fachberichte, Springer Verlag, 1980, pp. 375- 388.

Haussler, D. and Welzl, E., "ε-nets and simplex range queries". *Discrete Comput. Geom.* 2, 1987, pp. 127-151.

Lamoureux, Michael G., "A Dynamic Data Structure for Multi-Dimensional Range Searching", Technical Report, TR96-105, Faculty of Computer Science, Msc. Thesis, University of New Brunswick, 1996.

Laszlo, M. J., *"Computational Geometry and Computer Graphics in C++"*, Prentice hall, 1996, pp. 226 - 255.

Lueker, George S., " A Data Structure for Orthogonal Range Queries", proceedings of the 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, Oct 16 - 18, 1978, pp. 28 - 34.

Matousek, J., "Construction of ε-Nets". *Discrete Comput. Geom.* 5, 1990, pp. 427 - 448.

Matousek, J., "Reporting points in halfspaces". Comput. Geom. Theor. Appl. 2, 3, 1992, pp. 169 - 186.

Matousek J., "Geometric range searching". *ACM computing Surveys.* Vol.26, No. 4, Dec 1994, pp. 421-461.

McCreight, E. M., "Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles". Tech. Rep. CSL-80-9, Xerox Corp., Palo Alto Research Center, June 1980.

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

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

Nickerson, Bradford G., "Skip List Data Structure for Multidimensional Data", Computer Science Technical Report Series, CS TR-3263 UMIACS CS-TR-94-52, April 1994.

O'Rourke, J., "Computational Geometry Column 29". *SIGACT News*, Vol. 27, No. 3, Sept. 1997, pp. 55- 59.

Paterson, M. S., and Yao, F. F., "Optimal Binary Space Partitions for Orthogonal Objects". *Journal of algorithms*, Vol. 13, 1992, pp. 99 -113.

Rumbaugh, J., M Blaha, Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, New Jersey, 1991 P. 27-38.

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

Saxe, J. B., "On the number of range queries in k-space", *Discrete Appl. Math.* V. 1-2,1979, pp.217-225.

Vaidya, M. P., "Space-time tradeoffs for orthogonal range queries (Extended Abstract)". Proc. Of the 17th Annual ACM Symp. On Theor. Of Comput., 1985, pp. 169 - 174.

Vaidya, M. P., "Space-time tradeoffs for orthogonal range queries". *SIAMJ. Comput.* Vol. 18, No. 4, August 1989, pp. 748 - 758.

Van leeuwen, Jan, and Wood, D., "Dynamization of Decomposable Searching Problems",

*Information Processing Letters*, Volume 10, Number 2 (March 1980), pp. 51-56.

Weiss, M. A., "*Data Structures and Algorithm Analysis*". Second Edition. The Benjamin/Cummings Publishing Company, Inc., 1992, pp. 241 - 244.

Willard, D. E., "New Data Structures for Orthogonal Range Queries", SIAM J. *Comput.* Vol. 14, No. 1, Feb. 1985, pp. 232 - 253.

Willard, D. E., and Lueker, G. S., "Adding Range Restriction Capability to Dynamic Data Structures", *Journal of the ACM*, Vol. 32, No. 3, July 1985, pp. 597 - 617.

Yao, A. C., "Space-Time tradeoff for answering range Queries", Proc, 14th annual Symp. Theory of Comput., 1980, pp. 128 - 136.

# Appendix A

# Raw Construction and Destruction Time for Testing the k-d Search Skip List

Table A1 (a)
Raw construct time for pass 1 of run 1 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 490 | 8580 |
| 5 | 80 | 1040 | 14590 |
| 10 | 170 | 2620 | 21780 |

Table A1 (b)
Raw destruct time for pass 1 of run 1 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 640 | 12770 |
| 5 | 90 | 1440 | 22080 |
| 10 | 180 | 2080 | 33940 |

Table A2 (a)
Raw construct time for pass 2 of run 1 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 480 | 8480 |
| 5 | 70 | 1400 | 15730 |
| 10 | 120 | 1950 | 22780 |

Table A2 (b)
Raw destruct time for pass 2 of run 1 for testing k-d Search Skip List

| k | Raw destruct time(milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 620 | 13120 |
| 5 | 120 | 1720 | 21550 |
| 10 | 130 | 2590 | 31780 |

Table A3 (a)
Raw construct time for pass 3 of run 1 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 80 | 600 | 9060 |
| 5 | 50 | 960 | 14100 |
| 10 | 180 | 1540 | 25410 |

Table A3 (b)
Raw destruct time for pass 3 of run 1 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 1010 | 15320 |
| 5 | 80 | 1500 | 22260 |
| 10 | 190 | 2030 | 34190 |

Table A4 (a)
Raw construct time for pass 1 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 520 | 9890 |
| 5 | 70 | 1070 | 16540 |
| 10 | 100 | 1630 | 23140 |

Table A4 (b)
Raw destruct time for pass 4 of run 1 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 830 | 15240 |
| 5 | 80 | 1700 | 22170 |
| 10 | 150 | 2030 | 34370 |

Table A5 (a)
Raw construct time for pass 5 of run 1 for testing the k-d Search Skip List

| | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 100 | 480 | 9130 |
| 5 | 70 | 960 | 13170 |
| 10 | 130 | 1520 | 23140 |

Table A5 (b)
Raw destruct time for pass 5 of run 1 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 90 | 740 | 14290 |
| 5 | 100 | 1500 | 20420 |
| 10 | 130 | 2090 | 34850 |

Table A6 (a)
Raw construct time for pass 1 of run 2 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 110 | 540 | 8630 |
| 5 | 60 | 1090 | 13630 |
| 10 | 160 | 2230 | 22020 |

Table A6 (b)
Raw destruct time for pass 1 of run 2 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 140 | 670 | 13670 |
| 5 | 80 | 3380 | 19860 |
| 10 | 130 | 2300 | 32200 |

Table A7 (a)
Raw construct time for pass 2 of run 2 of initial testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 520 | 8670 |
| 5 | 80 | 890 | 14140 |
| 10 | 110 | 1620 | 22700 |

Table A7 (b)
Raw destruct time for pass 2 of run 2 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 660 | 14500 |
| 5 | 80 | 1200 | 19840 |
| 10 | 140 | 2100 | 31950 |

Table A8 (a)
Raw construct time for pass 3 of run 2 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 760 | 8990 |
| 5 | 110 | 860 | 14300 |
| 10 | 120 | 1540 | 22550 |

Table A8 (b)
Raw destruct time for pass 3 of run 2 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 1030 | 13590 |
| 5 | 120 | 1670 | 20580 |
| 10 | 120 | 2460 | 32090 |

Table A9 (a)
Raw construct time for pass 4 of run 2 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 480 | 8730 |
| 5 | 60 | 840 | 14320 |
| 10 | 110 | 1880 | 21450 |

Table A9 (b)
Raw destruct time for pass 4 of run 2 of initial testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 980 | 14190 |
| 5 | 50 | 1420 | 20630 |
| 10 | 120 | 2480 | 33370 |

Table A10 (a)
Raw construct time for pass 5 of run 2 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 520 | 9060 |
| 5 | 70 | 880 | 13210 |
| 10 | 120 | 1970 | 22080 |

Table A10 (b)
Raw destruct time for pass 5 of run 2 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 670 | 14150 |
| 5 | 80 | 1530 | 20550 |
| 10 | 120 | 2660 | 30520 |

Table A11 (a)
Raw construct time for pass 1 of run 3 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 620 | 8840 |
| 5 | 70 | 830 | 13080 |
| 10 | 130 | 1600 | 25760 |

Table A11 (b)
Raw destruct time for pass 1 of run 3 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 630 | 14930 |
| 5 | 70 | 1320 | 21210 |
| 10 | 140 | 2230 | 31610 |

Table A12 (a)
Raw construct time for pass 2 of run 3 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 680 | 9070 |
| 5 | 100 | 860 | 14060 |
| 10 | 110 | 1960 | 23560 |

Table A12 (b)
Raw destruct time for pass 2 of run 3 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 790 | 14020 |
| 5 | 90 | 1180 | 23490 |
| 10 | 140 | 2660 | 32330 |

Table A13 (a)
Raw construct time for pass 3 of run 3 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 540 | 8740 |
| 5 | 60 | 880 | 14100 |
| 10 | 130 | 1540 | 22460 |

Table A13 (b)
Raw destruct time for pass 3 of run 3 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 800 | 13240 |
| 5 | 70 | 1300 | 20950 |
| 10 | 140 | 2060 | 34090 |

Table A14 (a)
Raw construct time for pass 4 of run 3 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 590 | 9590 |
| 5 | 100 | 890 | 13220 |
| 10 | 110 | 2170 | 22500 |

Table A14 (b)
Raw destruct time for pass 4 of run 3 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 800 | 13940 |
| 5 | 70 | 1200 | 20710 |
| 10 | 100 | 2710 | 31240 |

Table A15 (a)
Raw construct time for pass 5 of run 3 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 520 | 8650 |
| 5 | 70 | 960 | 14990 |
| 10 | 110 | 1540 | 22110 |

Table A15 (b)
Raw destruct time for pass 5 of run 3 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 630 | 15610 |
| 5 | 60 | 1230 | 20680 |
| 10 | 140 | 2120 | 31140 |

Table A16 (a)
Raw construct time for pass 1 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 470 | 9430 |
| 5 | 90 | 870 | 15190 |
| 10 | 130 | 1940 | 22170 |

Table A16 (b)
Raw destruct time for pass 1 of run 4 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 600 | 14810 |
| 5 | 120 | 1240 | 19070 |
| 10 | 140 | 2660 | 34940 |

Table A17 (a)
Raw construct time for pass 2 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 500 | 8540 |
| 5 | 110 | 1140 | 13660 |
| 10 | 120 | 1660 | 23000 |

Table A17 (b)
Raw destruct time for pass 2 of run 4 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 650 | 13980 |
| 5 | 100 | 1370 | 21170 |
| 10 | 130 | 2082 | 31210 |

Table A18 (a)
Raw construct time for pass 3 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 460 | 8850 |
| 5 | 70 | 1080 | 13800 |
| 10 | 170 | 1530 | 22510 |

Table A18 (b)
Raw destruct time for pass 3 of run 4 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 620 | 14270 |
| 5 | 110 | 1520 | 21150 |
| 10 | 190 | 2150 | 31120 |

Table A19 (a)
Raw construct time for pass 4 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 530 | 8320 |
| 5 | 70 | 890 | 13150 |
| 10 | 120 | 1650 | 24000 |

Table A19 (b)
Raw destruct time for pass 4 of run 4 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 1050 | 13000 |
| 5 | 70 | 1200 | 19650 |
| 10 | 110 | 2260 | 32320 |

Table A20 (a)
Raw construct time for pass 5 of run 4 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 740 | 8960 |
| 5 | 70 | 880 | 12800 |
| 10 | 190 | 1620 | 23200 |

Table A20 (b)
Raw destruct time for pass 5 of run 4 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 900 | 8960 |
| 5 | 70 | 1160 | 20840 |
| 10 | 80 | 2170 | 35170 |

Table A21 (a)
Raw construct time for pass 1 of run 5 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 530 | 8530 |
| 5 | 80 | 840 | 14730 |
| 10 | 100 | 1770 | 23710 |

Table A21 (b)
Raw destruct time for pass 1 of run 5 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 1010 | 13640 |
| 5 | 100 | 1370 | 19380 |
| 10 | 140 | 2230 | 33440 |

Table A22 (a)
Raw construct time for pass 2 of run 5 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 510 | 8530 |
| 5 | 80 | 860 | 13660 |
| 10 | 130 | 2240 | 22820 |

Table A22 (b)
Raw destruct time for pass 2 of run 5 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 1040 | 14230 |
| 5 | 80 | 1210 | 19540 |
| 10 | 130 | 2940 | 43400 |

Table A23 (a)
Raw construct time for pass 3 of run 5 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 520 | 8020 |
| 5 | 60 | 980 | 13210 |
| 10 | 90 | 1600 | 24500 |

Table A23 (b)
Raw destruct time for pass 3 of run 5 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 640 | 14480 |
| 5 | 70 | 1630 | 21660 |
| 10 | 80 | 2080 | 32920 |

Table A24 (a)
Raw construct time for pass 4 of run 5 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 500 | 8530 |
| 5 | 80 | 860 | 13480 |
| 10 | 160 | 1670 | 22640 |

Table A24 (b)
Raw destruct time for pass 4 of run 5 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 640 | 13700 |
| 5 | 100 | 1170 | 19800 |
| 10 | 160 | 2390 | 33580 |

Table A25 (a)
Raw construct time for pass 5 of run 5 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 460 | 8190 |
| 5 | 70 | 850 | 13170 |
| 10 | 110 | 1600 | 23290 |

Table A25 (b)
Raw destruct time for pass 5 of run 5 testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 60 | 680 | 14270 |
| 5 | 70 | 1160 | 20180 |
| 10 | 140 | 2090 | 34160 |

Table A26 (a)
Raw construct time for pass 1 of run 6 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 560 | 8240 |
| 5 | 60 | 1070 | 15400 |
| 10 | 100 | 1580 | 23030 |

Table A26 (b)
Raw destruct time for pass 1 of run 6 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 780 | 13570 |
| 5 | 60 | 1580 | 29840 |
| 10 | 130 | 2750 | 31580 |

Table A27 (a)
Raw construct time for pass 2 of run 6 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 550 | 9220 |
| 5 | 70 | 970 | 14840 |
| 10 | 120 | 1650 | 23340 |

Table A27 (b)
Raw destruct time for pass 2 of run 6 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 80 | 800 | 15270 |
| 5 | 70 | 1380 | 22330 |
| 10 | 130 | 2100 | 32430 |

Table A28 (a)
Raw construct time for pass 3 of run 6 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 540 | 8450 |
| 5 | 70 | 900 | 14060 |
| 10 | 110 | 1670 | 23300 |

Table A28 (b)
Raw destruct time for pass 3 of run 6 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 70 | 690 | 13290 |
| 5 | 110 | 1140 | 19960 |
| 10 | 130 | 2640 | 31690 |

Table A29 (a)
Raw construct time for pass 4 of run 6 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 30 | 530 | 9150 |
| 5 | 70 | 920 | 13700 |
| 10 | 110 | 2100 | 22840 |

Table A29 (b)
Raw destruct time for pass 4 of run 6 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 850 | 14340 |
| 5 | 70 | 1220 | 19600 |
| 10 | 130 | 2540 | 31610 |

Table A30 (a)
Raw construct time for pass 5 of run 6 for testing the k-d Search Skip List

| k | Raw construct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 510 | 8560 |
| 5 | 90 | 860 | 13520 |
| 10 | 150 | 1670 | 24350 |

Table A30 (b)
Raw destruct time for pass 5 of run 6 for testing the k-d Search Skip List

| k | Raw destruct time (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 80 | 780 | 13770 |
| 5 | 120 | 1480 | 21940 |
| 10 | 180 | 2050 | 33860 |

# Appendix B

## Average Construction and Destruction Time for Each Test Run of the k-d Search Skip List

Table B1 (a)
Construct time average for run 1 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 78 | 514 | 9028 |
| 5 | 74 | 1086 | 14826 |
| 10 | 140 | 1852 | 23250 |

Table B1 (b)
Destruct time average for run 1 of testing of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 66 | 768 | 14148 |
| 5 | 94 | 1572 | 21696 |
| 10 | 156 | 2164 | 33826 |

Table B2 (a)
Construct time average for run 2 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 48 | 564 | 8816 |
| 5 | 86 | 912 | 13920 |
| 10 | 124 | 1848 | 22160 |

Table B2 (b)
Destruct time average for run 2 of testing of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 58 | 802 | 14020 |
| 5 | 94 | 1455 | 20292 |
| 10 | 126 | 2400 | 32026 |

Table B3 (a)
Construct time average for run 3 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 40 | 590 | 8978 |
| 5 | 80 | 884 | 13890 |
| 10 | 118 | 1762 | 23278 |

Table B3 (b)
Destruct time average for run 3 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 44 | 730 | 14348 |
| 5 | 72 | 1246 | 21408 |
| 10 | 132 | 2354 | 32082 |

Table B4 (a)
Construct time average for run 4 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 34 | 540 | 8820 |
| 5 | 82 | 972 | 13720 |
| 10 | 146 | 1680 | 22976 |

Table B4 (b)
Destruct time average for run 4 of testing of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 52 | 764 | 11384 |
| 5 | 74 | 1298 | 20376 |
| 10 | 140 | 2264 | 32952 |

Table B5 (a)
Construct time average for run 5 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 46 | 504 | 8360 |
| 5 | 74 | 878 | 13650 |
| 10 | 118 | 1776 | 23392 |

Table B5 (b)
Destruct time average for run 5 of testing of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 56 | 802 | 14064 |
| 5 | 84 | 1308 | 20112 |
| 10 | 130 | 2346 | 33525 |

Table B6 (a)
Construct time average for run 6 of testing of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 50 | 538 | 8724 |
| 5 | 72 | 944 | 14304 |
| 10 | 118 | 1642 | 23372 |

Table B6 (b)
Destruct time average for run 6 of testing of the k-d Search Skip List

| k | Destruct t time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 66 | 780 | 14048 |
| 5 | 86 | 1360 | 20957 |
| 10 | 140 | 2416 | 32234 |

Table B7 (a)
Construct time average for all test runs of of the k-d Search Skip List

| k | Construct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 44 | 542 | 8788 |
| 5 | 78 | 946 | 14051 |
| 10 | 127 | 1760 | 23071 |

Table B7 (b)
Destruct time average for all test runs of of the k-d Search Skip List

| k | Destruct time average (milliseconds) | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 57 | 774 | 13669 |
| 5 | 84 | 1373 | 20807 |
| 10 | 137 | 2324 | 32774 |

# Appendix C

# Raw Insertion and Deletion Constant for Testing the k-d Search Skip List

Table C1 (a)
Insertion constant for pass 1 of run 1 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 8.14 | 14.25 |
| 5 | 5.32 | 6.91 | 9.69 |
| 10 | 5.65 | 8.70 | 7.24 |

Table C1 (b)
Deletion constant for pass 1 of run 1 for testing the k-d Search Skip List

| k | Deletion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.63 | 21.21 |
| 5 | 5.98 | 9.57 | 14.67 |
| 10 | 5.97 | 6.48 | 7.57 |

Table C2 (a)
Insertion constant for pass 2 of run 1 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 7.97 | 14.09 |
| 5 | 4.65 | 9.30 | 10.45 |
| 10 | 3.99 | 6.48 | 7.57 |

Table C2 (b)
Destruction constant for pass 2 of run 1 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.30 | 21.80 |
| 5 | 7.97 | 11.43 | 14.32 |
| 10 | 4.32 | 8.60 | 10.56 |

Table C3 (a)
Insertion constant for pass 3 of run 1 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 13.28 | 9.97 | 15.05 |
| 5 | 5.30 | 6.38 | 9.37 |
| 10 | 5.98 | 5.12 | 8.44 |

Table C3 (b)
Destruction constant for pass 3 of run 1 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 16.78 | 25.45 |
| 5 | 5.32 | 9.97 | 14.79 |
| 10 | 6.31 | 6.74 | 11.36 |

Table C4 (a)
Insertion constant for pass 4 of run 1 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 8.64 | 16.43 |
| 5 | 4.65 | 7.11 | 10.99 |
| 10 | 3.32 | 5.41 | 7.69 |

Table C4 (b)
Destruction constant for pass 4 of run 1 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 13.79 | 25.31 |
| 5 | 5.32 | 11.29 | 14.73 |
| 10 | 4.98 | 6.74 | 11.42 |

Table C5 (a)
Insertion constant for pass 5 of run 1 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 16.61 | 7.97 | 15.16 |
| 5 | 4.65 | 6.38 | 8.75 |
| 10 | 4.32 | 5.05 | 7.69 |

Table C5 (b)
Destruction constant for pass 5 of run 1 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 14.95 | 12.29 | 23.74 |
| 5 | 6.64 | 9.97 | 13.57 |
| 10 | 4.32 | 6.94 | 11.58 |

Table C6 (a)
Insertion constant for pass 1 of run 2 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 8.97 | 14.33 |
| 5 | 7.31 | 7.24 | 9.05 |
| 10 | 5.32 | 7.41 | 7.31 |

Table C6 (b)
Destruction constant for pass 1 of run 2 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 13.29 | 11.13 | 22.71 |
| 5 | 9.31 | 10.46 | 13.19 |
| 10 | 4.32 | 7.64 | 10.70 |

Table C7 (a)
Insertion constant for pass 2 of run 2 of initial testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.64 | 14.40 |
| 5 | 5.32 | 5.91 | 9.39 |
| 10 | 3.65 | 5.38 | 7.54 |

Table C7 (b)
Destruction constant for pass 2 of run 2 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.96 | 24.08 |
| 5 | 5.31 | 7.97 | 13.18 |
| 10 | 4.65 | 6.98 | 10.61 |

Table C8 (a)
Insertion constant for pass 3 of run 2 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 12.62 | 14.93 |
| 5 | 7.31 | 5.71 | 9.50 |
| 10 | 3.98 | 5.12 | 7.49 |

Table C8 (b)
Destruction constant for pass 3 of run 2 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 17.11 | 22.57 |
| 5 | 7.97 | 11.09 | 13.67 |
| 10 | 3.98 | 8.17 | 10.66 |

Table C9 (a)
Insertion constant for pass 4 of run 2 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 7.97 | 14.50 |
| 5 | 3.98 | 5.58 | 9.51 |
| 10 | 3.65 | 6.25 | 7.13 |

Table C9 (b)
Destruction constant for pass 4 of run 2 of initial testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 16.28 | 23.57 |
| 5 | 3.32 | 9.43 | 13.70 |
| 10 | 3.99 | 8.24 | 11.09 |

Table C10 (a)
Insertion constant for pass 5 of run 2 testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.64 | 15.05 |
| 5 | 4.65 | 5.85 | 8.77 |
| 10 | 3.98 | 6.54 | 7.33 |

Table C10 (b)
Destruction constant for pass 5 of run 2 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 11.13 | 23.50 |
| 5 | 5.32 | 10.17 | 13.65 |
| 10 | 3.98 | 8.84 | 10.14 |

Table C11 (a)
Insertion constant for pass 1 of run 3 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.30 | 14.69 |
| 5 | 4.65 | 5.51 | 8.69 |
| 10 | 4.32 | 5.32 | 8.56 |

Table C11 (b)
Destruction constant for pass 1 of run 3 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.46 | 24.80 |
| 5 | 4.65 | 8.77 | 14.09 |
| 10 | 4.65 | 7.41 | 10.51 |

Table C12 (a)
Insertion constant for pass 2 of run 3 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 11.29 | 15.06 |
| 5 | 6.64 | 5.71 | 9.34 |
| 10 | 3.65 | 6.51 | 7.83 |

Table C12 (b)
Destruction constant for pass 2 of run 3 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 13.12 | 23.28 |
| 5 | 5.98 | 7.84 | 15.61 |
| 10 | 4.65 | 8.84 | 10.74 |

Table C13 (a)
Insertion constant for pass 3 of run 3 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 8.97 | 14.52 |
| 5 | 3.98 | 5.85 | 9.37 |
| 10 | 4.32 | 5.12 | 7.46 |

Table C13 (b)
Destruction constant for pass 3 of run 3 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 13.28 | 21.99 |
| 5 | 4.65 | 8.64 | 13.92 |
| 10 | 4.65 | 6.84 | 11.32 |

Table C14 (a)
Insertion constant for pass 4 of run 3 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 9.80 | 15.93 |
| 5 | 6.64 | 5.91 | 8.78 |
| 10 | 4.32 | 7.21 | 7.47 |

Table C14 (b)
Destruction constant for pass 4 of run 3 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 13.29 | 23.15 |
| 5 | 4.65 | 7.97 | 13.76 |
| 10 | 3.32 | 9.00 | 10.38 |

Table C15 (a)
Insertion constant for pass 5 of run 3 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.64 | 14.37 |
| 5 | 4.65 | 6.38 | 9.96 |
| 10 | 3.65 | 5.12 | 7.34 |

Table C15 (b)
Destruction constant for pass 5 of run 3 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 10.46 | 25.93 |
| 5 | 3.98 | 8.17 | 13.74 |
| 10 | 4.65 | 7.04 | 10.34 |

Table C16 (a)
Insertion constant for pass 1 of run 4 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 7.81 | 15.66 |
| 5 | 5.98 | 5.78 | 10.09 |
| 10 | 4.32 | 6.44 | 7.36 |

Table C16 (b)
Destruction constant for pass 1 of run 4 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 9.97 | 24.60 |
| 5 | 7.97 | 8.24 | 12.67 |
| 10 | 4.65 | 8.84 | 11.61 |

Table C17 (a)
Insertion constant for pass 2 of run 4 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 8.30 | 14.18 |
| 5 | 7.30 | 7.57 | 9.08 |
| 10 | 3.99 | 5.51 | 7.64 |

Table C17 (b)
Destruction constant for pass 2 of run 4 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 10.80 | 23.22 |
| 5 | 6.64 | 9.10 | 14.07 |
| 10 | 4.32 | 6.92 | 10.37 |

Table C18 (a)
Insertion constant for pass 3 of run 4 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 7.64 | 14.70 |
| 5 | 4.65 | 7.18 | 9.17 |
| 10 | 5.65 | 5.08 | 7.48 |

Table C18 (b)
Destruction constant for pass 3 of run 4 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 10.30 | 23.70 |
| 5 | 7.31 | 10.09 | 14.05 |
| 10 | 6.31 | 7.14 | 10.34 |

Table C19 (a)
Insertion constant for pass 4 of run 4 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 8.80 | 13.82 |
| 5 | 4.65 | 5.91 | 8.74 |
| 10 | 3.99 | 5.48 | 7.97 |

Table C19 (b)
Destruction constant for pass 4 of run 4 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 17.44 | 21.59 |
| 5 | 4.65 | 7.97 | 13.06 |
| 10 | 3.65 | 7.51 | 10.74 |

Table C20 (a)
Insertion constant for pass 5 of run 4 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 12.29 | 14.88 |
| 5 | 4.65 | 5.85 | 8.50 |
| 10 | 6.31 | 5.38 | 7.71 |

Table C20 (b)
Destruction constant for pass 5 of run 4 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 14.95 | 24.88 |
| 5 | 4.65 | 7.71 | 13.85 |
| 10 | 2.66 | 7.21 | 11.68 |

Table C21 (a)
Insertion constant for pass 1 of run 5 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.80 | 14.17 |
| 5 | 5.24 | 5.58 | 9.79 |
| 10 | 3.32 | 5.88 | 7.87 |

Table C21 (b)
Destruction constant for pass 1 of run 5 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 16.78 | 22.66 |
| 5 | 6.64 | 9.10 | 12.88 |
| 10 | 4.65 | 7.41 | 11.11 |

Table C22 (a)
Insertion constant for pass 2 of run 5 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 8.47 | 14.17 |
| 5 | 5.32 | 5.71 | 9.08 |
| 10 | 4.32 | 7.44 | 7.58 |

Table C22 (b)
Destruction constant for pass 2 of run 5 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 17.27 | 23.64 |
| 5 | 5.32 | 8.04 | 12.98 |
| 10 | 4.32 | 9.77 | 14.42 |

Table C23 (a)
Insertion constant for pass 3 of run 5 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 8.64 | 13.32 |
| 5 | 3.99 | 6.52 | 8.77 |
| 10 | 2.99 | 5.31 | 8.14 |

Table C23 (b)
Destruction constant for pass 3 of run 5 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 10.63 | 24.05 |
| 5 | 4.65 | 10.83 | 14.39 |
| 10 | 2.66 | 6.91 | 10.94 |

Table C24 (a)
Insertion constant for pass 4 of run 5 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.30 | 14.17 |
| 5 | 5.32 | 5.71 | 8.96 |
| 10 | 5.32 | 5.55 | 7.52 |

Table C24 (b)
Destruction constant for pass 4 of run 5 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 10.63 | 22.76 |
| 5 | 6.64 | 7.77 | 13.15 |
| 10 | 5.32 | 7.97 | 11.16 |

Table C25 (a)
Insertion constants for pass 5 of run 5 for testing of the k-d Search Skip List

| k | Constant c for insertion averages | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 7.64 | 13.60 |
| 5 | 4.65 | 5.65 | 8.75 |
| 10 | 3.65 | 5.32 | 7.74 |

Table C25 (b)
Destruction constant for pass 5 of run 5 testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.97 | 11.29 | 23.70 |
| 5 | 4.65 | 7.71 | 13.40 |
| 10 | 4.65 | 6.95 | 11.33 |

Table C26 (a)
Insertion constant for pass 1 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 9.30 | 13.69 |
| 5 | 3.99 | 7.11 | 10.23 |
| 10 | 3.32 | 5.25 | 7.65 |

Table C26 (b)
Destruction constant for pass 1 of run 6 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 12.96 | 22.54 |
| 5 | 3.99 | 10.50 | 19.83 |
| 10 | 4.32 | 9.14 | 10.49 |

Table C27 (a)
Insertion constant for pass 2 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 9.14 | 15.31 |
| 5 | 4.65 | 6.44 | 9.86 |
| 10 | 3.99 | 5.48 | 7.75 |

Table C27 (b)
Destruction constant for pass 2 of run 6 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.28 | 13.29 | 25.36 |
| 5 | 4.65 | 9.17 | 14.84 |
| 10 | 4.32 | 6.98 | 10.77 |

Table C28 (a)
Insertion constant for pass 3 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 8.97 | 14.04 |
| 5 | 4.65 | 5.98 | 9.34 |
| 10 | 3.65 | 5.55 | 7.74 |

Table C28 (b)
Destruction constant for pass 3 of run 6 for testing the k-d Search Skip List

| k | Destruction constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 11.63 | 11.46 | 22.07 |
| 5 | 7.31 | 7.57 | 13.26 |
| 10 | 4.32 | 8.77 | 10.53 |

Table C29 (a)
Insertion constant for pass 4 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 4.98 | 8.80 | 15.19 |
| 5 | 4.65 | 6.11 | 9.10 |
| 10 | 3.65 | 6.98 | 7.59 |

Table C29 (b)
Insertion constant for pass 4 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.30 | 14.17 | 23.82 |
| 5 | 4.65 | 8.11 | 13.02 |
| 10 | 4.32 | 8.44 | 10.50 |

Table C30 (a)
Insertion constant for pass 5 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 8.47 | 14.22 |
| 5 | 5.98 | 5.71 | 8.98 |
| 10 | 4.98 | 5.54 | 8.08 |

Table C30 (b)
Insertion constant for pass 5 of run 6 for testing the k-d Search Skip List

| k | Insertion constant | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | | | |
| 5 | | | |
| 10 | | | |

# Appendix D

# Insertion and Deletion Constant for Each Test Run of the k-d Search Skip List

Table D1 (a)
Constant for insertion average for run 1 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 12.96 | 8.54 | 14.99 |
| 5 | 4.91 | 7.22 | 9.85 |
| 10 | 4.65 | 6.15 | 7.72 |

Table D1 (b)
Constant for deletion average for run 1 of testing the k-d Search Skip List

| k | Constant for deletion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 10.96 | 12.75 | 23.50 |
| 5 | 6.25 | 10.44 | 14.41 |
| 10 | 5.18 | 7.18 | 11.23 |

Table D2 (a)
Constant for insertion average for run 2 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 7.97 | 9.36 | 14.64 |
| 5 | 5.71 | 6.06 | 9.24 |
| 10 | 4.11 | 6.14 | 7.36 |

Table D2 (b)
Constant for deletion average for run 2 of testing the k-d Search Skip List

| k | Constant for deletion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.63 | 13.32 | 23.28 |
| 5 | 6.24 | 9.82 | 13.48 |
| 10 | 4.18 | 7.97 | 10.63 |

Table D3 (a)
Constant for insertion average for run 3 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 6.64 | 9.79 | 14.91 |
| 5 | 5.31 | 5.87 | 9.22 |
| 10 | 4.05 | 5.85 | 6.33 |

Table D3 (b)
Constant for deletion average for run 3 of testing the k-d Search Skip List

| k | Constant for deletion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 7.30 | 12.12 | 23.81 |
| 5 | 4.78 | 8.28 | 14.15 |
| 10 | 4.65 | 6.21 | 11.28 |

Table D4 (a)
Constant for insertion average for run 4 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 5.64 | 8.97 | 14.64 |
| 5 | 5.44 | 6.46 | 9.11 |
| 10 | 4.85 | 5.58 | 7.61 |

Table D4 (b)
Constant for deletion average for run 4 of testing the k-d Search Skip List

| k | Constant for deletion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.63 | 12.70 | 23.58 |
| 5 | 6.24 | 8.61 | 13.53 |
| 10 | 4.31 | 7.52 | 10.94 |

Table D5 (a)
Constant for insertion average for run 5 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 7.63 | 8.36 | 11.88 |
| 5 | 4.89 | 5.83 | 9.17 |
| 10 | 3.91 | 5.89 | 7.77 |

Table D5 (b)
Constant for deletion average for run 5 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.29 | 13.32 | 23.36 |
| 5 | 5.58 | 8.68 | 13.36 |
| 10 | 4.32 | 6.60 | 11.78 |

Table D6 (a)
Constant for insertion average for run 6 of testing the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 7.30 | 8.90 | 14.46 |
| 5 | 4.78 | 5.27 | 9.50 |
| 10 | 3.91 | 5.76 | 7.76 |

Table D6 (b)
Constant for deletion average for run 6 of testing the k-d Search Skip List

| k | Constant for deletion | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.96 | 12.96 | 23.33 |
| 5 | 5.71 | 9.03 | 15.10 |
| 10 | 4.65 | 8.02 | 10.70 |

Table D7 (a)
Constant for insertion average for all test runs of the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 8.02 | 8.98 | 14.25 |
| 5 | 5.17 | 6.11 | 9.34 |
| 10 | 4.24 | 5.89 | 7.43 |

Table D7 (b)
Constant for deletion average for all test runs of the k-d Search Skip List

| k | Constant c for insertion average | | |
|---|---|---|---|
| | n = 1,000 | n = 10,000 | n = 100,000 |
| 2 | 9.29 | 12.86 | 23.48 |
| 5 | 5.80 | 9.14 | 14.01 |
| 10 | 4.55 | 7.25 | 11.09 |

# Appendix E

# Average Search Time for Each Test Run of the k-d Search Skip List

Table E1 (a)

The orthogonal range search time when dimension k= 2, size n = $10^3$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 67 | 0 | 0 | 0 | 10 | 0 | 2 |
| 30% | 304 | 0 | 0 | 10 | 0 | 0 | 2 |
| 50% | 497 | 10 | 0 | 0 | 0 | 0 | 2 |
| 80% | 803 | 0 | 10 | 0 | 10 | 0 | 4 |
| 100% | 997 | 0 | 0 | 10 | 0 | 10 | 4 |

Table E1 (b)

The semi-infinite range search time when dimension k = 2, size n = $10^3$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 39.85% | 96 | 0 | 0 | 0 | 0 | 0 | 0 |
| 83.89% | 481 | 0 | 0 | 0 | 10 | 0 | 2 |
| 96.89% | 702 | 10 | 0 | 0 | 0 | 0 | 2 |
| 91.25% | 841 | 10 | 0 | 10 | 0 | 0 | 4 |
| 100% | 997 | 10 | 10 | 10 | 0 | 0 | 6 |

Table E2 (a)

The orthogonal range search time when dimension k= 2, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 798 | 0 | 10 | 10 | 10 | 10 | 8 |
| 30% | 2648 | 20 | 20 | 20 | 30 | 20 | 22 |
| 50% | 4844 | 40 | 30 | 30 | 50 | 40 | 38 |
| 80% | 7951 | 60 | 40 | 30 | 30 | 30 | 38 |
| 100% | 9997 | 40 | 50 | 50 | 50 | 60 | 50 |

Table E2 (b)

The semi-infinite range search time when dimension k = 2, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 32.36% | 1252 | 10 | 10 | 20 | 20 | 10 | 14 |
| 62.95% | 5015 | 30 | 10 | 20 | 10 | 20 | 18 |
| 74.76% | 5302 | 30 | 30 | 20 | 30 | 40 | 30 |
| 89.45% | 8012 | 30 | 50 | 40 | 30 | 30 | 36 |
| 100% | 9997 | 40 | 40 | 40 | 40 | 40 | 40 |

Table E3 (a)

The orthogonal range search time when dimension k= 2, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 7738 | 100 | 100 | 110 | 130 | 100 | 108 |
| 30% | 27837 | 450 | 420 | 580 | 420 | 430 | 460 |
| 50% | 48808 | 360 | 380 | 370 | 360 | 550 | 364 |
| 80% | 79517 | 500 | 500 | 480 | 490 | 500 | 494 |
| 100% | 99997 | 560 | 690 | 550 | 530 | 560 | 578 |

Table E3 (b)

The semi-infinite range search time when dimension k = 2, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 64.74% | 28413 | 160 | 170 | 180 | 160 | 190 | 172 |
| 87.75% | 49140 | 480 | 340 | 340 | 340 | 340 | 368 |
| 85.40% | 63626 | 370 | 400 | 350 | 400 | 530 | 410 |
| 82.32% | 80548 | 430 | 460 | 480 | 460 | 470 | 460 |
| 100% | 99996 | 550 | 710 | 510 | 510 | 510 | 510 |

Table E4 (a)

The orthogonal range search time when dimension k= 5, size n = $10^3$

| area rat | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 61 | 0 | 10 | 0 | 10 | 0 | 4 |
| 30% | 264 | 0 | 0 | 0 | 0 | 0 | 0 |
| 50% | 462 | 0 | 0 | 0 | 10 | 10 | 4 |
| 80% | 795 | 0 | 0 | 0 | 0 | 0 | 0 |
| 100% | 994 | 0 | 10 | 0 | 0 | 0 | 2 |

Table E4 (b)

The semi-infinite range search time when dimension k = 5, size n = $10^3$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 36.01% | 198 | 0 | 10 | 0 | 10 | 0 | 4 |
| 38.71% | 347 | 10 | 10 | 10 | 10 | 10 | 10 |
| 74.27% | 621 | 10 | 10 | 0 | 10 | 0 | 4 |
| 91.43% | 872 | 0 | 0 | 10 | 10 | 0 | 4 |
| 100% | 995 | 10 | 10 | 0 | 0 | 0 | 4 |

Table E5 (a)

The orthogonal range search time when dimension k= 5, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 851 | 30 | 30 | 30 | 30 | 30 | 30 |
| 30% | 2844 | 50 | 40 | 50 | 40 | 30 | 42 |
| 50% | 4821 | 70 | 90 | 40 | 50 | 60 | 62 |
| 80% | 7964 | 70 | 60 | 100 | 70 | 60 | 72 |
| 100% | 9992 | 100 | 80 | 80 | 70 | 80 | 82 |

Table E5 (b)

The semi-infinite range search time when dimension k = 5, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24.22% | 2141 | 60 | 50 | 60 | 60 | 70 | 60 |
| 57.56% | 5170 | 40 | 40 | 50 | 50 | 50 | 42 |
| 70.77% | 6750 | 50 | 70 | 50 | 40 | 50 | 52 |
| 91.74% | 8851 | 50 | 50 | 80 | 60 | 60 | 60 |
| 100% | 9994 | 70 | 60 | 50 | 60 | 50 | 56 |

Table E6 (a)

The orthogonal range search time when dimension k= 5, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 8555 | 340 | 440 | 330 | 340 | 370 | 364 |
| 30% | 28931 | 520 | 610 | 730 | 560 | 710 | 626 |
| 50% | 49138 | 640 | 680 | 720 | 650 | 610 | 660 |
| 80% | 79713 | 730 | 920 | 760 | 710 | 820 | 788 |
| 100% | 99988 | 820 | 840 | 820 | 830 | 790 | 820 |

Table E6 (b)

The semi-infinite range search time when dimension k = 5, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 42.44% | 37270 | 450 | 480 | 420 | 420 | 410 | 436 |
| 57.69% | 49066 | 500 | 540 | 730 | 550 | 630 | 590 |
| 74.13% | 67253 | 640 | 680 | 720 | 650 | 610 | 628 |
| 91.25% | 87435 | 730 | 920 | 760 | 710 | 820 | 788 |
| 100% | 99989 | 700 | 750 | 710 | 750 | 690 | 720 |

Table E7 (a)

The orthogonal range search time when dimension k= 10, size n = $10^3$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 93 | 10 | 0 | 0 | 0 | 0 | 2 |
| 30% | 282 | 10 | 0 | 0 | 0 | 10 | 4 |
| 50% | 452 | 10 | 10 | 10 | 10 | 10 | 10 |
| 80% | 773 | 10 | 10 | 10 | 0 | 0 | 6 |
| 100% | 986 | 10 | 10 | 0 | 10 | 20 | 10 |

Table E7 (b)

The semi-infinite range search time when dimension k = 10, size n = $10^3$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 32.41% | 261 | 0 | 10 | 0 | 10 | 0 | 4 |
| 56.08% | 500 | 10 | 0 | 20 | 10 | 10 | 10 |
| 64.99% | 601 | 10 | 10 | 10 | 10 | 10 | 10 |
| 87.92% | 832 | 0 | 0 | 0 | 10 | 0 | 2 |
| 100% | 990 | 10 | 0 | 0 | 0 | 0 | 2 |

Table E8 (a)

The orthogonal range search time when dimension k= 10, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 884 | 100 | 70 | 70 | 80 | 120 | 88 |
| 30% | 2897 | 70 | 90 | 80 | 80 | 100 | 84 |
| 50% | 4950 | 150 | 90 | 100 | 90 | 90 | 104 |
| 80% | 7999 | 120 | 110 | 120 | 180 | 100 | 126 |
| 100% | 9997 | 90 | 110 | 120 | 90 | 130 | 108 |

Table E8 (b)

The semi-infinite range search time when dimension k = 10, size n = $10^4$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 27.10% | 2078 | 110 | 70 | 70 | 90 | 85 | 85 |
| 68.26% | 6706 | 80 | 90 | 70 | 80 | 80 | 80 |
| 75.23% | 7314 | 90 | 80 | 90 | 40 | 70 | 74 |
| 87.17% | 8572 | 90 | 110 | 90 | 100 | 90 | 96 |
| 100% | 9998 | 90 | 90 | 100 | 80 | 90 | 90 |

Table E9 (a)

The orthogonal range search time when dimension k= 10, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10% | 9408 | 680 | 680 | 690 | 710 | 720 | 696 |
| 30% | 28954 | 910 | 960 | 1300 | 880 | 920 | 994 |
| 50% | 49212 | 1100 | 1060 | 990 | 1040 | 1050 | 1048 |
| 80% | 79819 | 1240 | 1170 | 1130 | 1120 | 1180 | 1168 |
| 100% | 99996 | 1520 | 1190 | 1200 | 1220 | 1460 | 1318 |

Table E9 (b)

The semi-infinite range search time when dimension k = 10, size n = $10^5$

| area ratio | # nodes found | the search time in 5 passes (millisecond) | | | | | average time |
|---|---|---|---|---|---|---|---|
| 0% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 43.06% | 40522 | 740 | 800 | 780 | 810 | 710 | 768 |
| 56.39% | 51163 | 830 | 770 | 990 | 810 | 850 | 850 |
| 74.07% | 71921 | 1130 | 850 | 880 | 900 | 870 | 926 |
| 91.65% | 90030 | 1240 | 950 | 920 | 920 | 910 | 988 |
| 100% | 99995 | 910 | 900 | 930 | 940 | 1000 | 936 |

Table E10

The average orthogonal range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 0% | 10% | 30% | 50% | 80% | 100% |
|---|---|---|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | | | | |
| 2 | $10^3$ | 0 | 2 | 2 | 2 | 4 | 4 |
| | $10^4$ | 0 | 8 | 22 | 38 | 38 | 50 |
| | $10^5$ | 0 | 108 | 364 | 460 | 494 | 578 |
| 5 | $10^3$ | 0 | 4 | 0 | 4 | 0 | 2 |
| | $10^4$ | 0 | 30 | 42 | 62 | 72 | 82 |
| | $10^5$ | 0 | 364 | 626 | 660 | 788 | 820 |
| 10 | $10^3$ | 0 | 2 | 4 | 6 | 10 | 10 |
| | $10^4$ | 0 | 88 | 84 | 104 | 126 | 108 |
| | $10^5$ | 0 | 696 | 994 | 1048 | 1168 | 1318 |

Table E11

The average semi-infinite range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 10% or less | 40 to 60% | 90% or more |
|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | |
| 2 | $10^3$ | 0 | 0 | 4 |
| | $10^4$ | 4 | 16 | 38 |
| | $10^5$ | 38 | 172 | 500 |
| 5 | $10^3$ | 1 | 4 | 10 |
| | $10^4$ | 20 | 45 | 60 |
| | $10^5$ | 100 | 500 | 762 |
| 10 | $10^3$ | 4 | 10 | 10 |
| | $10^4$ | 38 | 70 | 90 |
| | $10^5$ | 180 | 800 | 960 |

# Appendix F
# Half-Space Range Search Time for Each Test Run

Table F1

The half-space range search time (milliseconds) when dimension k = 5, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $x_1 + x_2 + 10x_3 + 15x_4 + 4x_5 = 16$ | 9982 | 20 | 99.8% |
| $18x_1 + x_2 + 13x_4 = 9$ | 4768 | 10 | 47.7% |
| $7x_1 - 2x_2 + 10x_4 - x_5 = 6$ | 4766 | 10 | 47.7% |
| $7x_1 + 4x_3 - 2x_4 - x_5 = 0$ | 4767 | 10 | 47.7% |
| $9x_1 + 17x_2 + 3x_3 + 15x_4 + 18x_5 = -17$ | 10000 | 10 | 100% |
| $x_1 - x_2 + 7x_3 + 2x_4 + 17x_5 = 10$ | 161 | 10 | 1.6% |
| $14x_1 + 12x_3 + 11x_4 + 17x_5 = 2$ | 9466 | 10 | 94.7% |
| $5x_1 + 3x_2 + 16x_3 - x_4 + 16x_5 = 16$ | 4767 | 10 | 47.7% |
| $16x_1 + 12x_2 + 5x_3 + 18x_4 + 15x_5 = 4$ | 8536 | 10 | 85.4% |
| $13x_1 + 14x_2 + 10x_3 + 5x_4 + 15x_5 = 18$ | 9998 | 10 | 99.9% |
| $5x_1 + 17x_2 + 14x_3 + 14x_4 + 14x_5 = 10$ | 9999 | 20 | 99.9% |
| $18x_1 - x_2 - x_3 + x_4 + 13x_5 = 3$ | 4767 | 0 | 47.7% |
| $10x_1 + 2x_3 + 9x_4 + 12x_5 = 17$ | 9998 | 20 | 99.9% |
| $x_1 + 2x_2 + 7x_3 + 18x_4 + 12x_5 = 9$ | 4471 | 10 | 44.7% |
| $14x_1 + 5x_2 + 12x_3 + 5x_4 + 11x_5 = 2$ | 9998 | 20 | 99.9% |
| $6x_1 + 7x_2 + 16x_3 + 13x_4 + 10x_5 = 16$ | 9217 | 20 | 92.2% |
| $16x_1 + 3x_2 + 15x_3 + 10x_4 + 9x_5 = 13$ | 9998 | 20 | 99.9% |
| $8x_1 + 6x_2 - x_3 - 2x_4 + 8x_5 = 5$ | 4768 | 10 | 47.7% |
| $-2x_1 + 2x_2 - 2x_3 + 16x_4 + 7x_5 = 2$ | 0 | 0 | 0% |
| $5x_1 + 13x_2 + 12x_3 + x_4 + 6x_5 = 4$ | 4766 | 0 | 47.7% |
| $11x_1 + x_2 + 3x_3 + 2x_5 = 18$ | 9998 | 10 | 100% |
| $8x_1 + 2x_2 + 17x_3 + 7x_4 = 0$ | 8206 | 20 | 82.0% |
| $-2x_1 + 16x_3 + 4x_4 = 17$ | 0 | 10 | 0% |
| $10x_1 + x_2 + 15x_4 + 12x_5 = 10$ | 9998 | 10 | 100% |
| $2x_1 + 3x_2 + 4x_3 - x_5 = 2$ | 1251 | 0 | 12.5% |
| $15x_1 + 6x_2 + 9x_3 + 8x_4 - 2x_5 = 16$ | 4768 | 0 | 47.7% |
| $9x_1 + 15x_2 - x_3 + 6x_4 + 18x_5 = 4$ | 4769 | 0 | 47.7% |
| $x_1 + 17x_2 + 2x_3 + 15x_4 + 17x_5 = 18$ | 9233 | 10 | 92.3% |
| $14x_1 - x_2 + 7x_3 + 2x_4 + 17x_5 = 10$ | 4767 | 0 | 47.7% |
| $6x_1 + x_2 + 11x_3 + 10x_4 + 16x_5 = 2$ | 9998 | 10 | 100% |
| $-2x_1 + 9x_2 + 9x_4 + 15x_5 = 12$ | 7358 | 10 | 73.6% |
| $13x_1 + 12x_2 + 5x_3 + 17x_4 + 14x_5 = 4$ | 9998 | 10 | 100% |
| $2x_1 + 8x_2 + 4x_3 + 14x_4 + 13x_5 = 1$ | 6681 | 10 | 66.8% |
| $15x_1 + 10x_2 + 9x_3 + x_4 + 13x_5 = 15$ | 9999 | 10 | 100% |
| $7x_1 + 13x_2 + 13x_3 + 10x_4 + 12x_5 = 8$ | 9998 | 20 | 100% |
| $18x_1 + 9x_2 + 12x_3 + 7x_4 + 11x_5 = 5$ | 9999 | 20 | 100% |
| $6x_1 + 5x_2 + 11x_3 + 4x_4 + 10x_5 = 2$ | 9218 | 20 | 92.2% |
| $x_1 + 14x_2 + 2x_4 + 9x_5 = 11$ | 9997 | 20 | 100% |
| $11x_1 + 10x_2 + 8x_5 = 8$ | 9998 | 0 | 100% |
| $3x_1 + 12x_2 + 4x_3 + 8x_4 + 8x_5 = 1$ | 8119 | 40 | 81.2% |
| $16x_1 + 15x_2 + 8x_3 + 17x_4 + 7x_5 = 14$ | 8906 | 20 | 89.1% |
| $11x_1 + 2x_2 - 2x_3 + 15x_4 + 6x_5 = 2$ | 9999 | 10 | 100% |
| $4x_1 - x_2 + 17x_3 + 12x_4 + 5x_5 = 0$ | 4767 | 10 | 47.7% |
| $15x_1 + 7x_2 + 6x_3 + 11x_4 + 5x_5 = 9$ | 8040 | 20 | 80.0% |
| $7x_1 + 9x_2 + 11x_3 - x_4 + 4x_5 = 1$ | 7569 | 10 | 75.6% |
| $2x_1 + 18x_2 + 18x_4 + 4x_5 = 10$ | 3263 | 0 | 32.6% |
| $15x_1 + 4x_3 + 5x_4 + 3x_5 = 3$ | 8531 | 20 | 85.3% |

Table F2

The half-space range search time (milliseconds) when dimension k = 5, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $12x_1 + 16x_2 + 15x_3 - 2x_5 = 14$ | 8797 | 10 | 88.0% |
| $x_1 + 15x_2 - 2x_3 + 5x_4 + 17x_5 = 4$ | 4768 | 0 | 47.7% |
| $15x_1 + 17x_2 + 2x_3 + 14x_4 + 16x_5 = 18$ | 9999 | 10 | 100% |
| $2x_1 - x_2 - 2x_3 + 11x_5 = 3$ | 4768 | 10 | 47.7% |
| $-x_1 + 2x_2 + 8x_3 + 12x_4 + x_5 = 17$ | 0 | 0 | 0% |
| $12x_1 + 4x_2 + 13x_3 = 9$ | 9999 | 10 | 100% |
| $3x_1 + 6x_2 + 18x_3 + 8x_4 = 2$ | 6438 | 30 | 64.4% |
| $16x_1 + 9x_2 + x_3 + 17x_4 = 16$ | 9998 | 10 | 100% |
| $3x_1 + 16x_3 + 2x_4 - x_5 = 17$ | 4766 | 10 | 47.7% |
| $7x_1 + 4x_2 + 3x_3 - x_4 + 18x_5 = 2$ | 4768 | 0 | 47.7% |
| $18x_1 + 3x_3 + 16x_4 + 17x_5 = 0$ | 9274 | 10 | 92.75 |
| $12x_1 + 8x_2 + 13x_3 + 15x_4 + 16x_5 = 9$ | 9998 | 10 | 100% |
| $4x_1 + 11x_2 + 17x_3 + 2x_4 + 15x_5 = 1$ | 7571 | 10 | 75.75 |
| $-x_1 - x_2 + 6x_3 + 15x_5 = 10$ | 0 | 0 | 0% |
| $14x_1 + 7x_2 + 16x_3 + 14x_5 = -1$ | 8653 | 10 | 86.5% |
| $6x_1 + 9x_2 + 7x_4 + 14x_5 = 12$ | 9216 | 10 | 92.2% |
| $x_1 + 18x_2 + 10x_3 + 16x_4 + 13x_5 = 0$ | 12 | 0 | 0.1% |
| $13x_1 + 14x_3 + 14x_4 + 12x_5 = 14$ | 9597 | 20 | 96.0% |
| $5x_1 + 2x_2 - x_3 + x_4 + 12x_5 = 6$ | 4767 | 0 | 47.7% |
| $-2x_1 + 4x_2 + 2x_3 + 10x_4 + 11x_5 = 0$ | 4606 | 10 | 46.1% |
| $2x_1 + 9x_2 + 11x_3 + 5x_4 + 9x_5 = 5$ | 9998 | 10 | 100% |
| $18x_1 + 18x_2 + 4x_4 + 9x_5 = 14$ | 8797 | 10 | 88.0% |
| $3x_1 + 18x_2 + 11x_3 + 14x_4 - x_5 = 3$ | 8155 | 20 | 81.5% |
| $16x_1 + 15x_3 + x_4 + 18x_5 = 17$ | 7569 | 10 | 75.7% |
| $16x_1 + 4x_2 + 3x_3 - 2x_4 + 17x_5 = 2$ | 4768 | 0 | 47.7% |
| $10x_1 + 2x_3 + 15x_4 + 16x_5 = 0$ | 9998 | 10 | 100% |
| $17x_1 + 11x_2 + 17x_3 + x_4 + 14x_5 = 1$ | 9999 | 20 | 100% |
| $9x_1 + 13x_2 + 10x_4 + 14x_5 = 15$ | 9999 | 20 | 100% |
| $17x_1 + 3x_2 + 15x_3 + 17x_4 + 12x_5 = 17$ | 9998 | 10 | 100% |
| $8x_1 + 6x_2 - x_3 + 4x_4 + 12x_5 = 9$ | 4769 | 0 | 47.7% |
| $-2x_1 + 2x_2 - 2x_3 + x_4 + 11x_5 = 6$ | 0 | 0 | 0% |
| $10x_1 + 4x_2 + 2x_3 + 9x_4 + 10x_5 = 0$ | 9998 | 10 | 100% |
| $2x_1 + x_3 + 6x_4 + 9x_5 = 17$ | 8749 | 10 | 87.5% |
| $10x_1 + x_2 - 2x_4 + 2x_5 = 14$ | 9998 | 10 | 100% |
| $2x_1 + 3x_2 + 4x_3 + 5x_4 + x_5 = 6$ | 8618 | 10 | 86.2% |
| $15x_1 + 6x_2 + 9x_3 + 14x_4 + x_5 = -1$ | 9999 | 20 | 100% |
| $-x_1 + 11x_2 + 18x_3 + 10x_4 = 5$ | 4321 | 10 | 43.2% |
| $12x_1 + x13_2 + x_3 + 18x_4 - x_5 = -2$ | 7569 | 10 | 75.7% |
| $3x_1 + 15x_2 + 6x_3 + 5x_4 - 2x_5 = 11$ | 4766 | 10 | 47.7% |
| $16x_1 + 18x_2 + 10x_3 + 14x_4 + 18x_5 = 4$ | 9015 | 20 | 90.2% |
| $5x_1 + 14x_2 + 9x_3 + 11x_4 + 17x_5 = 1$ | 9997 | 20 | 100% |
| $-2x_1 + 16x_2 + 14x_3 - x_4 + 16x_5 = 15$ | 0 | 0 | 0% |
| $10x_1 - 2x_2 - 2x_3 + 6x_4 + 16x_5 = 7$ | 9999 | 10 | 100% |
| $2x_1 + 2x_3 + 15x_4 + 15x_5 = 0$ | 160 | 0 | 1.6% |
| $18x_1 + 9x_2 + 12x_3 + 13x_4 + 14x_5 = 9$ | 8799 | 10 | 88.0% |
| $9x_1 + 11x_2 + 17x_3 + 14x_5 = 1$ | 9596 | 10 | 96.0% |
| $4x_1 - x_2 + 5x_3 + 13x_5 = 10$ | 4746 | 0 | 47.5% |

Table F3

The half-space range search time (milliseconds) when dimension k = 5, size n = $5*10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $11x_1 + 4x_3 + 8x_5 = 8$ | 42153 | 100 | 84.3% |
| $6x_1 + 13x_2 + 13x_3 + 12x_4 + 2x_5 = 17$ | 49997 | 70 | 100% |
| $10x_1 + 7x_2 - 2x_3 + 11x_4 + 12x_5 = 11$ | 49996 | 100 | 100% |
| $8x_1 - 2x_2 + 10x_3 + 9x_4 + 4x_5 = 9$ | 49997 | 80 | 100% |
| $14x_1 + 14x_2 + 4x_4 - x_5 = 0$ | 37763 | 70 | 75.5% |
| $6x_1 + 18x_2 + 13x_3 + 5x_4 + 16x_5 = 16$ | 39261 | 70 | 78.5% |
| $14x_1 + 14x_2 + 10x_3 + 15x_4 + 9x_5 = 10$ | 49996 | 80 | 100% |
| $8x_1 + 16x_2 + 5x_3 + 3x_5 = 17$ | 43662 | 100 | 87.3% |
| $6x_1 - 2x_2 - 2x_4 + 13x_5 = -1$ | 49997 | 90 | 100% |
| $10x_1 - 2x_2 + 16x_3 + 18x_4 + 13x_5 = 0$ | 43854 | 90 | 87.7% |
| $6x_1 + 14x_2 + 14x_3 + 9x_4 + 9x_5 = 1$ | 49998 | 90 | 100% |
| $9x_1 + x_2 + 14x_3 + 5x_4 + 4x_5 = 3$ | 44213 | 80 | 88.4% |
| $7x_1 + 5x_2 + 15x_3 + 15x_4 + 9x_5 = 10$ | 37762 | 80 | 75.5% |
| $5x_1 + 9x_2 + 14x_3 + 6x_4 + x_5 = 9$ | 49080 | 80 | 98.2% |
| $11x_1 + 12x_2 + 18x_3 + 14x_4 = 2$ | 46082 | 90 | 92.2% |
| $x_1 - 2x_3 + 10x_4 + 16x_5 = 4$ | 49996 | 90 | 100% |
| $7x_1 + 16x_2 + 8x_3 + 6x_4 + 10x_5 = 16$ | 43247 | 100 | 86.5% |
| $16x_1 + 18x_2 + 3x_3 + 12x_4 + 5x_5 = 1$ | 45344 | 100 | 90.7% |
| $6x_1 + x_2 + 2x_3 + 18x_4 + 14x_5 = 18$ | 44276 | 80 | 88.5% |
| $x_1 + 14x_2 + 12x_3 + 10x_4 + 8x_5 = 5$ | 49990 | 90 | 100% |
| $7x_1 + 7x_2 + 15x_4 + 16x_5 = 10$ | 38496 | 80 | 77.05 |
| $-x_1 - x_2 + 5x_4 + 5x_5 = 12$ | 0 | 0 | 0% |
| $4x_1 + 11x_2 + 8x_3 + 10x_4 + 14x_5 = 17$ | 49997 | 90 | 1005 |
| $7x_1 + 14x_2 + 11x_3 + 9x_4 + 6x_5 = 6$ | 49998 | 100 | 100% |
| $17x_1 + 7x_2 + 8x_3 + 15x_4 - 2x_5 = 18$ | 24124 | 40 | 48.2% |
| $14x_1 + 15x_2 + 11x_3 + 8x_4 + 5x_5 = 6$ | 43019 | 100 | 86.4% |
| $6x_1 + x_2 + 2x_3 + 3x_4 + 17x_5 = 1$ | 46673 | 90 | 93.3% |
| $17x_1 + 15x_2 - x_3 + 9x_4 + 9x_5 = 13$ | 43855 | 80 | 87.7% |
| $-2x_1 + 10x_2 + 3x_4 = 17$ | 25874 | 50 | 51.7% |
| $8x_1 + 11x_2 - x_3 + 15x_4 + 14x_5 = 14$ | 43857 | 100 | 87.7% |
| $18x_1 + 9x_2 + 7x_3 + 7x_4 - x_5 = 16$ | 24143 | 40 | 48.3% |
| $3x_1 + 15x_2 + 3x_3 + 2x_4 - x_5 = 9$ | 10842 | 20 | 21.7% |
| $7x_1 + 17x_2 + 6x_3 + 5x_4 + 4x_5 = 14$ | 37596 | 70 | 75.2% |
| $11x_1 + 5x_3 - x_4 - x_5 = 13$ | 24124 | 50 | 48.2% |
| $2x_1 - 2x_2 + x_3 + 3x_4 + 7x_5 = 7$ | 10846 | 20 | 21.75 |
| $11x_1 - x_2 - x_4 + x_5 = 8$ | 24122 | 40 | 48.2% |
| $3x_1 + 15x_2 + 4x_4 + 6x_5 = 9$ | 10842 | 20 | 21.75 |
| $18x_1 + 4x_2 + 17x_3 + 15x_4 + x_5 = 0$ | 24124 | 60 | 48.2% |
| $x_1 + 15x_2 - x_3 + 2x_4 + 3x_5 = 9$ | 16673 | 70 | 33.3% |
| $3x_1 + 12x_2 + 18x_3 - x_4 + 2x_5 = 6$ | 24123 | 30 | 48/2% |
| $-x_1 - x_2 + 12x_4 + 9x_5 = 16$ | 30538 | 50 | 61.1% |
| $13x_1 + 12x_2 + 5x_3 - x_4 + 4x_5 = 13$ | 24123 | 50 | 48.25 |
| $6x_1 + 7x_3 + 3x_4 + 5x_5 = -1$ | 41899 | 100 | 83.8% |
| $14x_1 + x_2 + 2x_3 + 10x_4 = 5$ | 49998 | 90 | 1005 |
| $6x_1 + 18x_2 + 13x_3 + 5x_4 + 16x_5 = 16$ | 39261 | 70 | 78.5% |
| $10x_1 + 18x_2 + 11x_3 + 18x_4 + 10x_5 = 13$ | 43850 | 80 | 87.7% |
| $12x_1 + 5x_2 + 12x_3 + 14x_4 + 5x_5 = 15$ | 46060 | 120 | 92.1% |

Table F4

The half-space range search time (milliseconds) when dimension k = 5, size n =5* $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $5x_1 + 16x_2 - 2x_3 + 18x_4 + 9x_5 = 14$ | 49998 | 90 | 100% |
| $8x_1 + 7x_2 + 10x_3 + 14x_4 = 13$ | 46213 | 110 | 92.4% |
| $15x_1 + 16x_2 + 9x_3 + 8x_5 = 12$ | 49997 | 100 | 100% |
| $2x_1 - x_2 + 9x_3 + x_4 + 6x_5 = 15$ | 10802 | 20 | 21.6% |
| $16x_1 + 8x_2 + 17x_3 + 11x_4 - x_5 = 0$ | 24124 | 40 | 48.2% |
| $8x_1 + 10x_2 + 12x_3 + 18x_4 + 14x_5 = 6$ | 49998 | 90 | 100% |
| $6x_1 + 18x_2 + 13x_3 + 14x_4 + 10x_5 = 8$ | 49998 | 80 | 100% |
| $9x_1 + 4x_2 + 5x_3 + 18x_4 + 6x_5 = 7$ | 34664 | 70 | 69.3% |
| $12x_1 + 11x_2 + 8x_3 + 10x_4 + 14x_5 = 17$ | 49998 | 70 | 100% |
| $8x_1 + 15x_2 - x_3 + 2x_4 + 4x_5 = 9$ | 24125 | 30 | 48.2% |
| $x_1 - 2x_2 + 14x_4 + 11x_5 = 16$ | 49998 | 90 | 100% |
| $-x_1 + 17x_2 + 16x_3 + 18x_4 + 5x_5 = -1$ | 0 | 0 | 0% |
| $4x_1 - x_2 - x_3 + 4x_4 + 4x_5 = 12$ | 4975 | 10 | 9.5% |
| $9x_1 + 2x_2 + 10x_3 + 17x_4 + 15x_5 = 4$ | 49997 | 90 | 100% |
| $-2x_1 + 14x_2 + 2x_3 + 15 x_4 + 7x_5 = 2$ | 0 | 0 | 0% |
| $5x_1 + 14x_3 + 10x_4 - x_5 = -1$ | 49997 | 80 | 100% |
| $11x_1 + 15x_2 + 17x_4 + 14x_5 = 17$ | 44375 | 80 | 88.8% |
| $2x_1 + 4x_2 + 6x_5 = 14$ | 12639 | 30 | 25.3% |
| $9x_1 + 2x_2 + 5x_3 + 16x_5 = 11$ | 49998 | 90 | 100% |
| $5x_1 + 11x_2 + 2x_3 - x_4 + 18x_5 = 6$ | 24125 | 40 | 48.2% |
| $8x_1 + 12x_2 + 10x_4 + 10x_5 = 2$ | 49997 | 90 | 100% |
| $11x_1 + x_2 + 16x_3 + 9x_4 + 17x_5 = 12$ | 49997 | 100 | 100% |
| $10x_1 + 16x_2 + 16x_4 + 11x_5 = 10$ | 43855 | 80 | 87.7% |
| $2x_1 + 14x_2 + 15x_3 - 2x_4 + 4x_5 = 14$ | 24109 | 50 | 48.2% |
| $14x_1 - x_2 + 11x_3 + 15x_4 + 5x_5 = 6$ | 41352 | 60 | 82.7% |
| $11x_1 + 18x_3 - 2x_4 + 16x_5 = 6$ | 37763 | 80 | 75.5% |
| $11x_1 + 15x_2 + 11x_3 - 2x_4 + 8x_5 = 7$ | 49998 | 90 | 100% |
| $7x_1 - x_2 + 11x_3 + 5x_4 + 11x_5 = 14$ | 24135 | 40 | 48.23% |
| $12x_1 + 16x_2 + 8x_3 + 13x_4 + 2x_5 = 8$ | 42243 | 90 | 84.5% |
| $10x_1 + 16x_2 + 7x_3 + 11x_4 = 8$ | 49997 | 90 | 100% |
| $17x_1 + 15x_2 + 10x_3 + 15x_4 + 17x_5 = -1$ | 43853 | 60 | 87.7% |
| $11x_1 + 8x_2 + 16x_3 + 2x_4 + 13x_5 = 17$ | 42150 | 90 | 84.3% |
| $16x_1 + 15x_2 - x_3 + 3x_4 + 5x_5 = 9$ | 24125 | 50 | 48.3% |
| $12x_1 + 11x_2 + 8x_3 + 10x_4 + 14x_5 = 17$ | 49998 | 90 | 100% |
| $10x_1 + 14x_2 + 2x_3 + 8x_4 + 2x_5 = -1$ | 41587 | 100 | 83.2% |
| $-x_1 + 12x_2 + 16x_3 + 2x_4 + 11x_5 = 0$ | 49922 | 90 | 100% |
| $14x_1 + 5x_3 + 7x_4 + 14x_5 = 12$ | 42478 | 100 | 84.9% |
| $3x_1 + 11x_2 + 18x_3 + 4x_4 + 5x_5 = 10$ | 31085 | 80 | 62.2% |
| $17x_1 - x_2 + 7x_3 + 11x_4 + 10x_5 = 2$ | 49997 | 80 | 100% |
| $17x_1 + 16x_2 - x_4 + 2x_5 = -2$ | 24125 | 50 | 48.2% |
| $7x_1 - 2x_2 + 15x_3 + 16x_4 + 10x_5 = 1$ | 49996 | 100 | 100% |
| $8x_1 + 10x_3 + 4x_5 = 8$ | 45009 | 90 | 90% |
| $10x_1 - 2x_3 + 6x_5 = 15$ | 49997 | 80 | 100% |
| $18x_1 + 6x_2 + 4x_3 + 7x_4 - 2x_5 = 18$ | 24123 | 50 | 48.2% |
| $13x_1 + x_3 + 13x_4 + 9x_5 = 9$ | 24124 | 50 | 48.2% |
| $11x_1 + 12x_2 + 9x_3 - 2x_5 = 18$ | 49998 | 100 | |

Table F5

The half-space range search time (milliseconds) when dimension k = 5, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $2x_1 + x_2 + 4x_3 + 18x_4 + 4x_5 = 1$ | 16868 | 100 | 16.9% |
| $16x_1 + 3x_2 - x_3 + 2x_4 - x_5 = 8$ | 48173 | 80 | 48.25 |
| $-2x_1 + x_2 + 4x_3 + 17x_4 + 16x_5 = 13$ | 0 | 0 | 0% |
| $2x_1 + 14x_2 + 2x_3 + 10x_4 + 13x_5 = 8$ | 99994 | 220 | 100% |
| $12x_1 + 6x_2 + x_3 - x_4 = 10$ | 48173 | 90 | 48.2% |
| $10x_1 + 8x_2 + 16x_3 + 17x_4 + 9x_5 = 13$ | 99994 | 200 | 100% |
| $18x_1 + 17x_2 + 4x_3 + 15x_4 + 8x_5 = 1$ | 86438 | 170 | 86.4% |
| $2x_1 + 4x_2 + 16x_3 + 7x_4 + 18x_5 = 18$ | 27990 | 100 | 28.0% |
| $8x_1 + 18x_2 + 3x_3 + 12x_4 + 4x_5 = 1$ | 84995 | 140 | 85.0% |
| $17x_1 + 11x_2 + 8x_3 + 10x_4 + 14x_5 = 17$ | 99995 | 170 | 100% |
| $17x_1 + 10x_2 + 13x_3 + 3x_4 - 2x_5 = 10$ | 99994 | 150 | 100% |
| $16x_1 + 3x_2 + 8x_3 + 3x_5 = 18$ | 90228 | 270 | 90.2% |
| $11x_1 + 5x_2 + 14x_3 + 17x_4 + 8x_5 = 14$ | 90557 | 270 | 90.6% |
| $10x_1 - x_2 + 9x_3 + 14x_4 + 14x_5 = 1$ | 92130 | 170 | 92.1% |
| $2x_1 + 9x_2 + 3x_3 + 8x_4 - 2x_5 = 7$ | 48028 | 100 | 48.0% |
| $13x_1 + 11x_2 - x_3 + 14x_4 + 13x_5 = 14$ | 87647 | 160 | 87.7% |
| $10x_1 + 11x_2 + 7x_3 + 15x_4 + 15x_5 = 0$ | 75410 | 140 | 75.4% |
| $16x_1 + 12x_2 + 5x_3 + 12x_4 + 12x_5 = 0$ | 99994 | 230 | 100% |
| $8x_1 + 8x_2 + 3x_3 + 7x_4 + 8x_5 = -2$ | 88150 | 200 | 88.2% |
| $10x_1 + 8x_2 + 4x_3 + 18x_5 = 5$ | 81767 | 230 | 81.8% |
| $7x_1 + 8x_2 + 13x_3 + x_4 = 12$ | 93714 | 240 | 93.7% |
| $9x_1 - x_2 + 7x_3 + 16x_4 + 4x_5 = 18$ | 48173 | 110 | 48.2% |
| $12x_1 + 5x_2 + 10x_3 + 9x_4 + 12x_5 = 6$ | 99994 | 180 | 100% |
| $-x_1 + 5x_2 - x_4 = 17$ | 0 | 0 | 0% |
| $15x_1 + 6x_2 + 9x_3 + 4x_5 = 3$ | 81261 | 130 | 81.3% |
| $-x_1 + 14x_3 - x_4 + 14x_5 = 18$ | 0 | 0 | 0% |
| $10x_1 + 12x_2 + 14x_3 + 10x_4 + 3x_5 = -1$ | 99994 | 180 | 100% |
| $-x_1 - 2x_2 + 9x_3 + 10x_4 + 14x_5 = 4$ | 99838 | 160 | 100% |
| $6x_1 + 5x_2 + 5x_4 + 5x_5 = 0$ | 82228 | 310 | 82.2% |
| $18x_1 + 10x_2 + 14x_3 + x_4 + 7x_5 = 15$ | 98328 | 190 | 98.3% |
| $x_1 + 13x_2 + 7x_3 - 2x_4 + 15x_5 = 18$ | 100 | 0 | 0.1% |
| $11x_1 + 18x_3 + 4x_4 - x_5 = 10$ | 48173 | 110 | 48.2% |
| $13x_1 + 8x_3 + 14x_5 = 3$ | 95960 | 250 | 96.0% |
| $14x_1 - x_2 + x_4 + 4x_5 = 7$ | 75413 | 130 | 75.4% |
| $18x_1 - x_2 + 9x_3 + 2x_4 + 8x_5 = 15$ | 75411 | 120 | 75.4% |
| $8x_1 + 5x_3 + 17x_4 + 9x_5 = 4$ | 92526 | 180 | 92.5% |
| $6x_1 + x_2 + 2x_3 + 18x_4 + 14x_5 = 18$ | 88397 | 180 | 88.4% |
| $11x_1 + 7x_2 + 8x_3 + 17x_4 = 17$ | 87475 | 220 | 87.5% |
| $14x_1 + 7x_2 + 9x_3 + 4x_4 + 5x_5 = 0$ | 81085 | 170 | 81.1% |
| $12x_1 + x_2 + x_3 + 17x_4 + 12x_5 = -2$ | 99995 | 170 | 100% |
| $9x_1 + 17x_2 + 14x_3 + 8x_4 - x_5 = 16$ | 72757 | 140 | 72.8% |
| $18x_1 + 16x_2 + 7x_3 + 3x_4 + 7x_5 = 16$ | 84761 | 160 | 84.8% |
| $10x_1 + 12x_2 + 7x_3 + 8x_4 + 12x_5 = 17$ | 92455 | 230 | 92.5% |
| $18x_1 + 2x_2 - x_3 - 2x_4 - 2x_5 = 11$ | 75410 | 140 | 75.4% |
| $13x_1 + 13x_2 + 3x_3 + x_4 + 12x_5 = 6$ | 99994 | 190 | 100% |
| $4x_1 + 17x_3 + 13x_4 + 15x_5 = 9$ | 75410 | 150 | 75.4% |

Table F6

The half-space range search time (milliseconds) when dimension k = 5, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $16x_1 + x_2 + 3x_3 + 15x_4 + 14x_5 = 14$ | 95930 | 200 | 95.9% |
| $2x_1 + 14x_2 + 2x_3 + 16x_4 + 16x_5 = 12$ | 87603 | 140 | 87.6% |
| $9x_1 + 14x_2 + 12x_3 + 5x_4 + 5x_5 = 1$ | 99994 | 140 | 100% |
| $11x_1 + 6x_2 + 11x_3 + 7x_4 + 9x_5 = 0$ | 99994 | 190 | 100% |
| $13x_1 + 4x_2 + 15x_3 - x_4 + 11x_5 = 14$ | 75410 | 120 | 75.4% |
| $11x_1 + 7x_2 + 8x_3 + 17x_4 = 17$ | 87475 | 170 | 87.5% |
| $17x_1 + 13x_2 + 15x_3 + 15x_4 + 5x_5 = 16$ | 75411 | 160 | 75.4% |
| $14x_1 + 5x_3 + x_4 + 11x_5 = 8$ | 99995 | 180 | 100% |
| $-x_1 + 6x_2 + 11x_3 + 17x_5 = 7$ | 0 | 0 | 0% |
| $9x_1 + 11x_2 + 10x_3 + 5x_4 + 4x_5 = 3$ | 99995 | 170 | 100% |
| $12x_1 + 18x_2 + 13x_3 - x_4 + 12x_5 = 12$ | 84907 | 150 | 84.9% |
| $x_1 + 7x_2 + 7x_3 + 14x_4 + 17x_5 = 18$ | 52105 | 120 | 52.1% |
| $13x_1 + 13x_2 + 3x_3 + x_4 + 12x_5 = 6$ | 99994 | 210 | 100% |
| $3x_1 + 5x_2 + 13x_3 + x_4 + 11x_5 = -2$ | 95933 | 150 | 96.0% |
| $14x_1 + 5x_2 + 12x_3 - x_4 + 7x_5 = -1$ | 48174 | 80 | 48.2% |
| $12x_1 + 5x_2 + 11x_3 + 18x_4 + 5x_5 = -1$ | 91732 | 190 | 91.7% |
| $x_1 + 4x_2 + 14x_3 + 9x_4 + 7x_5 = 13$ | 59935 | 120 | 59.9% |
| $6x_1 + 7x_2 + 16x_3 - x_4 + 13x_5 = -1$ | 66899 | 120 | 66.9% |
| $-x_1 + 3x_2 + 16x_3 + 4x_4 - 2x_5 = 0$ | 0 | 0 | 0% |
| $2x_1 + 10x_2 + 9x_3 + 8x_4 + 17x_5 = -1$ | 99994 | 210 | 100% |
| $12x_1 + 13x_2 + x_3 + 18x_4 - x_5 = -2$ | 75411 | 130 | 75.4% |
| $17x_1 + 9x_2 + x_3 + x_4 + 2x_5 = -1$ | 99994 | 130 | 100% |
| $3x_1 + 13x_2 + 11x_3 + 5x_4 + 7x_5 = 8$ | 72053 | 130 | 72.1% |
| $10x_1 + 2x_3 - x_5 = 3$ | 75411 | 140 | 75.4% |
| $8x_1 + 15x_3 + 7x_4 = 0$ | 77883 | 170 | 77.9% |
| $-x_1 + 11x_3 + x_4 = 11$ | 61624 | 110 | 61.6% |
| $13x_1 - x_2 + 17x_3 + 3x_4 + 11x_5 = 8$ | 99995 | 180 | 100% |
| $7x_1 + 17x_2 + x_3 + 4x_4 + x_5 = 4$ | 95554 | 280 | 95.6% |
| $x_1 + x_2 + 14x_3 + 6x_4 + 14x_5 = 12$ | 88393 | 240 | 88.4% |
| $15x_1 + 15x_2 + 18x_3 + 11x_4 - x_5 = 8$ | 81262 | 120 | 81.3% |
| $4x_1 + 4x_2 + 13x_3 + 5x_4 + 3x_5 = 13$ | 73145 | 240 | 73.1% |
| $5x_1 + 9x_2 + 10x_3 + 2x_4 + 5x_5 = 5$ | 99994 | 210 | 100% |
| $17x_1 + 7x_2 + 16x_3 + 4x_4 + 16x_5 = 2$ | 81770 | 160 | 81.8% |
| $9x_1 + 12x_3 + 10x_4 + 7x_5 = 14$ | 99995 | 170 | 100% |
| $12x_1 + 3x_2 + 16x_3 + 9x_4 = 3$ | 99995 | 190 | 100% |
| $10x_1 + 5x_2 + 9x_3 + 7x_4 + 9x_5 = 6$ | 99994 | 160 | 100% |
| $9x_1 + 15x_2 + 5x_3 + 10x_4 = 15$ | 99995 | 170 | 100% |
| $10x_1 - x_2 + 2x_3 + 13x_4 + x_5 = 0$ | 75411 | 160 | 75.4% |
| $-2x_1 + 9x_2 + 6x_3 + 17x_4 + 15x_5 = 15$ | 20286 | 50 | 20.3% |
| $14x_1 + 13x_2 + 7x_3 + 18x_4 + 14x_5 = -2$ | 84137 | 200 | 84.1% |
| $2x_1 + 9x_2 + 16x_3 + 10x_4 + 5x_5 = 9$ | 99994 | 190 | 100% |
| $8x_1 + 12x_2 + 17x_3 - x_4 + 10x_5 = 16$ | 99994 | 180 | 100% |
| $3x_1 + 15x_2 - 2x_4 + 3x_5 = 5$ | 75410 | 140 | 75.4% |
| $9x_1 + 8x_2 + 7x_3 + 2x_4 + 11x_5 = 10$ | 81028 | 220 | 81.0% |
| $8x_1 + 17x_2 + 6x_3 + 9x_4 + 18x_5 = 8$ | 84136 | 230 | 84.1% |
| $18x_1 + 6x_2 + 10x_3 + 5x_4 + 6x_5 = 0$ | 82301 | 150 | 82.3% |
| $3x_1 + 17x_2 + 5x_3 + 8x_4 + 17x_5 = 9$ | | | |

Table F7

The half-space range search time (milliseconds) when dimension k = 7, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $5x_1 + 2x_2 + 14x_3 + 3x_4 - x_5 + 6x_6 + 13x_7 = 16$ | 4095 | 10 | 41.0% |
| $12x_1 + 5x_2 + 6x_3 + 5x_4 + 16x_5 + 2x_6 + x_7 = 17$ | 5985 | 10 | 59.9% |
| $14x_1 + 4x_2 + 10x_3 + 10x_4 + 14x_5 + 13x_6 + 11x_7 = 4$ | 10000 | 20 | 100% |
| $3x_1 + 15x_2 + 3x_3 + 17x_4 + 13x_5 + 15x_6 = 6$ | 5984 | 10 | 59.8% |
| $11x_1 + 15x_2 + 2x_3 + 8x_4 + 6x_5 + 11x_6 + 7x_7 = 1$ | 7376 | 20 | 73.8% |
| $16x_1 + 11x_3 + 3x_4 + 5x_5 + 18x_6 + 7x_7 = 17$ | 8545 | 10 | 85.5% |
| $7x_1 + 4x_2 + 3x_5 + 2x_6 + 6x_7 = 11$ | 6936 | 20 | 69.4% |
| $16x_1 + 3x_2 + 10x_3 - 2x_4 + 17x_6 + 15x_7 = 2$ | 9999 | 20 | 100% |
| $16x_1 + 17x_2 + 9x_3 + 13x_4 + 18x_5 + 11x_6 + 15x_7 = 1$ | 8670 | 20 | 86.7% |
| $4x_1 + 5x_2 + 6x_3 + 4x_4 + 15x_5 + 2x_6 + 13x_7 = 11$ | 2564 | 10 | 25.6% |
| $9x_1 + 10x_2 + 15x_3 + 14x_5 + 9x_6 + 13x_7 = 5$ | 9999 | 20 | 100% |
| $8x_1 + 2x_2 + 13x_3 + 15x_4 + 12x_5 + 3x_6 + 12x_7 = 5$ | 9669 | 10 | 96.7% |
| $-2x_1 - x_2 + 12x_3 + 12x_4 + 11x_5 + 12x_7 = 15$ | 8392 | 20 | 83.9% |
| $2x_1 + 3x_2 + 7x_4 + 9x_5 + 6x_6 + 11x_7 = 9$ | 5966 | 10 | 59.7% |
| $x_1 + 17x_2 - x_3 + 2x_4 + 7x_5 + 10x_7 = 9$ | 4080 | 10 | 40.8% |
| $6x_1 + 7x_3 - 2x_4 + 6x_5 + 7x_6 + 10x_7 = 3$ | 5884 | 10 | 58.8% |
| $17x_1 + 9x_2 + 17x_3 + 17x_4 + 5x_5 + 16x_6 + 9x_7 = 8$ | 9999 | 20 | 100% |
| $11x_1 + 5x_2 + 16x_3 + 14x_4 + 4x_5 + 13x_6 + 9x_7 = -2$ | 9999 | 30 | 100% |
| $16x_1 + 10x_2 + 4x_3 + 10x_4 + 3x_5 - x_6 + 8x_7 = 13$ | 5984 | 10 | 59.8% |
| $18x_1 + 9x_2 + 7x_3 + 15x_4 + x_5 + 9x_6 - 2x_7 = 0$ | 10000 | 20 | 100% |
| $9x_1 + 11x_2 + 12x_3 + 2x_4 + x_6 + 7x_7 = 18$ | 9999 | 30 | 100% |
| $4x_1 + 17x_2 + 10x_4 + 13x_5 + 4x_6 + 15x_7 = 7$ | 9999 | 20 | 100% |
| $17x_1 - x_2 + 3x_3 - 2x_4 + 12x_5 + 18x_6 + 4x_7 = 4$ | 5985 | 10 | 59.9% |
| $7x_1 + 6x_2 + 4x_3 + 15x_4 + 8x_5 + 13x_7 = 0$ | 6232 | 10 | 62.3% |
| $14x_1 + 9x_2 + 17x_3 + 16x_4 + 4x_5 + 16x_6 + x_7 = 2$ | 10000 | 20 | 100% |
| $-2x_1 + 14x_2 + 4x_3 + 12x_4 + 3x_5 + x_6 = 18$ | 4015 | 10 | 40.2% |
| $7x_1 + 13x_2 + 8x_3 + 17x_4 + x_5 + 12x_6 + 10x_7 = 4$ | 9939 | 10 | 99.4% |
| $7x_1 + 2x_2 + 2x_3 + 3x_4 + 14x_6 = 6$ | 8437 | 10 | 84.4% |
| $14x_1 + 14x_2 + 16x_3 + 10x_4 - x_5 + 15x_6 + 9x_7 = 8$ | 9151 | 20 | 91.5% |
| $9x_1 + x_2 + 5x_3 + 9x_4 - x_5 + 3x_6 + 9x_7 = 14$ | 9999 | 10 | 100% |
| $11x_1 + 9x_3 + 14x_4 + 18x_5 + 14x_6 - x_7 = 0$ | 10000 | 20 | 100% |
| $-x_1 + 2x_2 + 4x_3 + 13x_5 + 17x_7 = 9$ | 0 | 0 | 0% |
| $14x_1 + 7x_3 + 5x_4 + 11x_5 + 10x_6 + 6x_7 = 17$ | 8365 | 20 | 83.7% |
| $16x_1 + 9x_2 + 18x_3 + 3x_4 + 10x_5 - x_6 + 5x_7 = 0$ | 10000 | 20 | 100% |
| $3x_1 + 11x_3 + 10x_4 + 9x_5 + 16x_7 = 3$ | 9999 | 20 | 100% |
| $13x_1 + 16x_2 + 10x_3 + 7x_4 + 8x_5 - 2x_6 + 15x_7 = 13$ | 5985 | 10 | 59.9% |
| $18x_1 - x_3 + 3x_4 + 6x_5 + 3x_6 + 15x_7 = 7$ | 5984 | 10 | 59.9% |
| $4x_1 + 11x_2 + 13x_3 + 10x_4 + 5x_5 + 5x_6 + 3x_7 = 10$ | 9891 | 20 | 98.9% |
| $9x_1 + 16x_2 + x_3 + 6x_4 + 4x_5 + 11x_6 + 3x_7 = 4$ | 7223 | 10 | 72.2% |
| $3x_1 + 3x_2 + 11x_3 + 4x_4 + 3x_5 + 3x_7 = 9$ | 8571 | 40 | 85.7% |
| $16x_1 + 5x_2 + 15x_3 + 13x_4 + 2x_5 + 13x_6 + 13x_7 = 6$ | 9999 | 20 | 100% |
| $2x_1 + 17x_2 + 9x_3 - x_4 + x_5 + 15x_6 + 2x_7 = 8$ | 9989 | 20 | 99.9% |
| $18x_1 + 18x_2 + 17x_3 + 12x_4 - 2x_6 + x_7 = 13$ | 9999 | 20 | 100% |
| $12x_1 + 5x_2 + 6x_3 + 11x_4 - x_5 + 6x_6 + x_7 = -2$ | 5985 | 10 | 59.9% |
| $9x_1 + 12x_2 - x_3 + 15x_4 + 17x_5 + 5x_6 + 10x_7 = 10$ | 10000 | 20 | 100% |
| $5x_1 - x_2 + 12x_3 - x_4 + 15x_5 + 4x_6 = 1$ | 4092 | 10 | 40.9% |
| $13x_1 + 6x_2 + 3x_3 + 14x_4 + 6x_5 + 17x_7 = 9$ | 8295 | 20 | 83.0% |

Table F8

The half-space range search time (milliseconds) when dimension k = 7, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $-x_1 + 11x_2 + 11x_3 + 18x_5 + 2x_6 + 16x_7 = 14$ | 8392 | 20 | 83.9% |
| $9x_1 + 8x_2 + 10x_3 + 18x_4 + 17x_5 + 15x_7 = 3$ | 9999 | 20 | 100% |
| $x_1 + 10x_2 + 14x_3 + 5x_4 + 16x_5 + 13x_6 + 4x_7 = 0$ | 9953 | 20 | 99.5% |
| $14x_1 + 12x_2 - 2x_3 + 13x_4 + 16x_5 + 5x_6 + 15x_7 = -2$ | 10000 | 10 | 100% |
| $6x_1 + 15x_2 + 2x_3 + 15x_5 - x_6 + 4x_7 = 16$ | 5985 | 10 | 59.9% |
| $-x_1 + 17x_2 + 7x_3 + 9x_4 + 14x_5 + 12x_6 + 14x_7 = 13$ | 9928 | 10 | 99.3% |
| $8x_1 + 13x_2 + 6x_3 + 6x_4 + 13x_5 + 9x_6 + 14x_7 = 2$ | 5985 | 10 | 59.9% |
| $x_1 + 16x_2 + 10x_3 + 15x_4 + 12x_5 + x_6 + 3x_7 = 0$ | 9999 | 10 | 100% |
| $13x_1 - 2x_2 + 15x_3 + 2x_4 + 11x_5 + 15x_6 + 13x_7 = 17$ | 5985 | 10 | 59.9% |
| $4x_1 - x_3 + 10x_4 + 11x_5 + 7x_6 + 2x_7 = 15$ | 5984 | 20 | 59.9% |
| $2x_1 + 8x_2 + 8x_3 + 9x_4 + 10x_5 + 17x_6 + 2x_7 = -1$ | 9078 | 30 | 90.8% |
| $9x_1 + 4x_2 + 7x_3 + 6x_4 + 9x_5 + 14x_6 + x_7 = 9$ | 7111 | 20 | 71.1% |
| $4x_1 + 13x_2 + 18x_3 + 4x_4 + 9x_5 + 2x_6 + x_7 = 14$ | 5984 | 10 | 59.9% |
| $14x_1 + 9x_2 + 17x_3 + x_4 + 8x_5 + x_7 = 3$ | 9933 | 10 | 99.3% |
| $3x_1 + 5x_2 + 16x_3 - x_4 + 7x_5 + 17x_6 = 13$ | 430 | 0 | 4.3% |
| $16x_1 + 8x_2 + 7x_4 + 6x_5 + 10x_6 + 11x_7 = 10$ | 8473 | 20 | 84.7% |
| $5x_1 + 4x_2 - x_3 + 4x_4 + 5x_5 + 7x_6 + 10x_7 = 0$ | 2564 | 0 | 25.6% |
| $5x_1 + 13x_2 + 8x_3 + 2x_4 + 4x_5 + x16_6 + 10x_7 = 5$ | 8428 | 20 | 84.3% |
| $10x_1 + 9x_2 + 7x_3 + 3x_5 + 13x_6 + 10x_7 = 15$ | 10000 | 20 | 100% |
| $11x_1 + 5x_2 + 16x_3 + 8x_5 + 17x_6 + 9x_7 = -1$ | 9280 | 20 | 92.8% |
| $9x_1 + 11x_2 + 12x_3 + 9x_4 + 3x_5 + 5x_6 + 7x_7 = -2$ | 10000 | 20 | 100% |
| $-x_1 + 7x_2 + 11x_3 + 6x_4 + 2x_5 + 2x_6 + 7x_7 = 7$ | 0 | 0 | 0% |
| $14x_1 + 16x_2 + 4x_4 + 2x_5 + 12x_6 + 7x_7 = 13$ | 8365 | 20 | 83.7% |
| $6x_1 - 2x_2 + 4x_3 + 13x_4 + x_5 + 4x_6 + 17x_7 = 10$ | 5984 | 10 | 59.8% |
| $16x_1 + 14x_2 + 3x_3 + 10x_4 + x_6 + 17x_7 = 0$ | 10000 | 20 | 100% |
| $8x_1 + 17x_2 + 8x_3 + 18x_4 + 15x_6 + 6x_7 = 17$ | 8155 | 20 | 81.6% |
| $-x_1 - x_2 + 13x_3 + 5x_4 + 8x_6 + 16x_7 = 14$ | 0 | 0 | 0% |
| $13x_1 + 17x_3 + 14x_4 - x_5 + 5x_7 = 12$ | 9999 | 20 | 100% |
| $5x_1 + 3x_2 + x_4 - 2x_5 + 14x_6 + 16x_7 = 9$ | 5985 | 10 | 59.9% |
| $6x_1 + 16x_2 + 3x_4 + x_5 + 12x_6 - x_7 = 6$ | 5985 | 10 | 59.9% |
| $14x_1 + 6x_2 + 14x_3 + 10x_4 + 14x_6 + 8x_7 = 9$ | 9631 | 20 | 96.3% |
| $7x_1 - 2x_4 + 17x_5 + 11x_6 + 7x_7 = 13$ | 5985 | 10 | 59.9% |
| $17x_1 + 16x_2 - x_3 + 16x_4 + 16x_5 + 8x_6 + 6x_7 = 2$ | 5984 | 10 | 59.9% |
| $16x_1 + x_2 + 16x_3 + 4x_4 + 12x_5 + 18x_6 + 5x_7 = 1$ | 8809 | 20 | 88.1% |
| $10x_1 + 10x_2 + 4x_3 + 2x_4 + 11x_5 + 6x_6 + 4x_7 = 6$ | 10000 | 20 | 100% |
| $2x_1 + 12x_2 + 9x_3 + 11x_4 + 10x_5 - x_6 + 15x_7 = 3$ | 9990 | 20 | 99.9% |
| $15x_1 + 15x_2 + 14x_3 - x_4 + 10x_5 + 12x_6 + 4x_7 = 0$ | 9999 | 20 | 100% |
| $14x_1 + 7x_2 + 12x_3 + 13x_4 + 8x_5 + 6x_6 + 3x_7 = 0$ | 8646 | 20 | 86.5% |
| $17x_1 + 5x_2 + 15x_3 - x_4 + 6x_5 + 17x_6 + 13x_7 = 7$ | 5984 | 10 | 59.85 |
| $8x_1 + 8x_2 - x_3 + 6x_4 + 5x_5 + 10x_6 + 2x_7 = 4$ | 4092 | 10 | 40.9% |
| $-2x_1 + 4x_2 - 2x_3 + 3x_4 + 4x_5 + 7x_6 + 2x_7 = 15$ | 0 | 0 | 0% |
| $13x_1 + 13x_2 + 8x_3 + 2x_4 + 4x_5 + 16x_6 + x_7 = -1$ | 5984 | 10 | 59.8% |
| $-x_1 + 16x_2 + 3x_4 + 12x_6 + 11x_7 = 0$ | 9560 | 20 | 95.6% |
| $16x_1 + 2x_2 + 13x_3 + 7x_4 - x_5 + 11x_6 = 13$ | 8013 | 20 | 80.1% |
| $13x_1 + 9x_2 + 5x_3 + 11x_4 + 17x_5 + 10x_6 + 9x_7 = 4$ | 9999 | 20 | 100% |
| $7x_1 + 10x_2 + 14x_3 + 3x_4 + 15x_5 + 13x_6 + 8x_7 = 9$ | 9962 | 20 | 99.6% |
| $x_1 - x_2 + 2x_3 + 2x_4 + 14x_5 + x_6 + 8x_7 = 14$ | 81 | 0 | 0.8% |

Table F9

The half-space range search time (milliseconds) when dimension k = 7, size n = $5*10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $12x_1 + 4x_2 + 8x_3 + 8x_4 + 12x_5 + 13x_6 + 6x_7 = 6$ | 41437 | 130 | 82.9% |
| $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 = -7$ | 26711 | 160 | 53.4% |
| $x_1 + 4x_2 + 7x_3 + 5x_5 + 10x_6 + 14x_7 = 2$ | 49992 | 90 | 100% |
| $2x_1 + 17x_2 + 9x_3 - x_4 + x_5 + 15x_6 + 2x_7 = 8$ | 49999 | 100 | 100% |
| $9x_1 + 12x_2 - x_3 + 15x_4 + 17x_5 + 5x_6 + 10x_7 = 10$ | 49998 | 90 | 100% |
| $13x_1 + 9x_2 + 5x_3 + 5x_4 + 14x_5 + 5x_6 + 9x_7 = 3$ | 0 | 0 | 0% |
| $-2x_1 + 5x_2 + 16x_3 + 8x_5 + 17x_6 + 18x_7 = 4$ | 15314 | 30 | 30.6% |
| $x_1 + 13x_2 + 17x_3 + 17x_4 + 3x_5 - x_6 + 18x_7 = 16$ | 49999 | 90 | 100% |
| $13x_1 + 17x_3 + 14x_4 - x_5 + 5x_7 = 12$ | 49999 | 70 | 100% |
| $9x_1 + 8x_3 + 12x_4 + 15x_5 + 14x_6 + 14x_7 = 2$ | 49998 | 100 | 100% |
| $11x_1 + 16x_2 + 9x_3 + 5x_4 + 5x_5 - 2x_6 + 11x_7 = 16$ | 49998 | 110 | 100% |
| $9x_1 + x_2 + 5x_3 + 15x_4 + x_5 + 7x_6 + 9x_7 = 15$ | 12921 | 30 | 25.8% |
| $2x_1 + 15x_2 + 11x_3 + 2x_4 + 18x_5 + 5x_6 + 7x_7 = -2$ | 49999 | 90 | 100% |
| $17x_1 + 9x_2 + 18x_3 + 10x_4 + 14x_5 + 2x_6 + 5x_7 = 1$ | 45665 | 160 | 91.3% |
| $12x_1 + 8x_2 + 8x_3 + 8x_4 + 9x_5 + 17x_6 + 15x_7 = 13$ | 40651 | 140 | 81.3% |
| $7x_1 + 18x_3 + 3x_5 + 4x_6 + x_7 = 4$ | 31195 | 60 | 62.3% |
| $-2x_1 + 17x_3 + 12x_4 + 18x_5 + 9x_7 = 0$ | 46535 | 110 | 93.1% |
| $8x_1 + x_2 + 15x_3 + 3x_4 + 11x_5 + 18x_6 + 17x_7 = 16$ | 40559 | 130 | 81.1% |
| $9x_1 + 5x_2 + 15x_3 + 18x_4 + 5x_5 + 17x_6 + 4x_7 = 1$ | 30181 | 50 | 60.3% |
| $13x_1 + 12x_2 + 10x_3 + 17x_5 + 2x_6 + 7x_7 = 8$ | 49999 | 100 | 100% |
| $11x_1 + 17x_2 + 6x_3 + 8x_4 + 13x_5 + 12x_6 + 5x_7 = 6$ | 43480 | 110 | 86.9% |
| $13x_1 - x_2 + x_3 + 15x_4 + 8x_5 - 2x_6 + 3x_7 = 16$ | 30181 | 60 | 60.3% |
| $16x_1 + 2x_2 + x_3 + 9x_4 + 2x_5 + 18x_6 + 12x_7 = 1$ | 49998 | 80 | 100% |
| $3x_1 + 13x_2 + 6x_3 + 14x_4 + 18x_5 + 12x_6 + 10x_7 = 15$ | 49999 | 120 | 100% |
| $14x_1 + 14x_2 + 5x_3 + 5x_4 + 11x_5 + 9x_6 + 18x_7 = 11$ | 49998 | 100 | 100% |
| $9x_1 + 5x_2 + 15x_3 + 18x_4 + 5x_5 + 17x_6 + 4x_7 = 1$ | 40559 | 90 | 81.1% |
| $15x_1 + x_2 + 4x_3 + 13x_4 + 7x_6 + 13x_7 = 2$ | 30181 | 60 | 60.3% |
| $4x_1 + 15x_2 + x_3 - x_4 + 12x_5 - x_6 = 18$ | 15313 | 50 | 30.6% |
| $2x_1 - 2x_3 + 8x_4 + 8x_5 + 8x_6 - 2x_7 = 17$ | 70 | 0 | 0.1% |
| $3x_1 + 12x_2 + 8x_4 + 4x_5 + 13x_6 + 7x_7 = 3$ | 35683 | 120 | 71.3% |
| $18x_1 + 5x_2 + 5x_3 + 16x_4 + 10x_6 + 5x_7 = 7$ | 43959 | 70 | 87.9% |
| $11x_1 - x_2 + 11x_3 + 2x_4 + 17x_5 + 8x_6 + 3x_7 = 11$ | 30180 | 50 | 60.3% |
| $17x_1 + 15x_2 + x_3 - x_4 + 11x_5 - x_6 + 12x_7 = 12$ | 39127 | 70 | 78.2% |
| $-2x_1 + 6x_2 + 2x_3 - 2x_4 + 8x_5 + 3x_6 = -2$ | 0 | 0 | 0% |
| $17x_1 + 12x_2 - x_3 + 7x_4 + 4x_5 + 13x_6 - x_7 = 18$ | 30180 | 50 | 60.3% |
| $2x_1 + 8x_2 + 9x_3 + 2x_4 - x_5 + 3x_6 + 7x_7 = -2$ | 12 | 0 | 0.02% |
| $11x_1 + 9x_2 + 4x_3 + 9x_4 + 15x_5 + 10x_6 + 5x_7 = 7$ | 49998 | 130 | 100% |
| $-x_1 + 11x_2 + 15x_4 + 10x_5 + 17x_6 + 3x_7 = 16$ | 49928 | 90 | 100% |
| $15x_1 + 11x_2 + 11x_3 + 13x_4 + 5x_5 + 10x_6 + 12x_7 = 7$ | 49999 | 120 | 100% |
| $14x_1 + 7x_2 + 8x_4 + 12x_7 = 2$ | 43684 | 110 | 87.3% |
| $18x_1 + 8x_2 + 7x_3 + 13x_4 + 11x_5 - 2x_7 = 2$ | 30180 | 60 | 60.3% |
| $6x_1 + 10x_2 + 2x_3 - x_4 + 6x_5 + 6x_6 + 17x_7 = 11$ | 15313 | 30 | 30.6% |
| $-x_1 + 17x_2 + 6x_3 - x_6 + 14x_7 = 15$ | 0 | 0 | 0% |
| $2x_1 + 14x_2 + 14x_3 + 11x_4 + 17x_5 + 13x_7 = 8$ | 49967 | 100 | 100% |
| $14x_1 + x_2 + 14x_3 + 8x_4 + 13x_5 + x_6 = 4$ | 49260 | 120 | 98.4% |
| $x_1 + 18x_2 + 4x_3 + 3x_4 + 7x_5 + 12x_6 + 9x_7 = 5$ | 31146 | 90 | 62.3% |
| $2x_1 + 7x_2 + 9x_3 + 8x_4 + 2x_5 + 7x_6 + 7x_7 = -1$ | 17856 | 400 | 35.7% |

Table F10

The half-space range search time (milliseconds) when dimension k = 7, size n = $5*10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $17x_1 + x_2 + 5x_3 + x_4 + 6x_5 + 11x_6 + 18x_7 = 0$ | 42714 | 100 | 83.4% |
| $16x_1 + 6x_2 + x_3 + 10x_4 + 2x_5 + 16x_7 = 0$ | 43812 | 90 | 87.6% |
| $16x_1 + 3x_2 - x_4 + 15x_5 + 14x_6 + 2x_7 = 5$ | 30181 | 60 | 60.3% |
| $6x_1 + 11x_2 + 16x_4 + 10x_5 + 17x_6 + 11x_7 = 1$ | 49999 | 90 | 100% |
| $3x_1 + 5x_2 + 6x_3 + 3x_4 + 6x_5 + 14x_6 + 9x_7 = 5$ | 27585 | 100 | 55.1% |
| $-x_1 + 10x_2 + 2x_3 + 12x_4 + 2x_5 + 2x_6 + 8x_7 = 4$ | 0 | 0 | 0% |
| $12x_1 + 4x_2 + 8x_3 - 2x_5 + 6x_7 = 8$ | 30181 | 40 | 60.3% |
| $18x_1 - x_3 + 16x_4 + 13x_5 + 12x_6 + 15x_7 = 9$ | 43162 | 80 | 86.3% |
| $14x_1 + 10x_2 + 3x_3 + 8x_5 + 6x_6 + 13x_7 = 2$ | 43911 | 90 | 87.8% |
| $15x_1 + 18x_2 + 16x_3 + 4x_4 + 14x_5 + 15x_6 + 18x_7 = 15$ | 39128 | 100 | 78.2% |
| $16x_1 + x_2 + 16x_3 . x_4 + 8x_5 + 14x_6 + 5x_7 = 0$ | 30181 | 50 | 60.3% |
| $14x_1 + 7x_2 + 12x_3 + 7x_4 + 4x_5 + 2x_6 + 3x_7 = 0$ | 40146 | 110 | 80.2% |
| $4x_1 + 7x_2 + 10x_3 - x_4 - 2x_5 + 11x_7 = 15$ | 30180 | 60 | 60.3% |
| $5x_1 - x_2 + 12x_3 - x_4 + 15x_5 + 4x_6 = 1$ | 15314 | 20 | 30.6% |
| $6x_1 + 3x_2 + 12x_3 + 13x_4 + 9x_5 + 3x_6 + 7x_7 = 7$ | 47689 | 90 | 95.3% |
| $9x_1 + 3x_2 + 10x_3 + 3x_4 + x_5 + 7x_7 = 18$ | 47512 | 180 | 95.0% |
| $7x_1 + 9x_2 + 6x_3 + 12x_4 - 2x_5 + 9x_6 + 5x_7 = 17$ | 49999 | 90 | 100% |
| $13x_1 + 5x_2 + 17x_3 + 8x_4 + 14x_5 + 14x_7 = 18$ | 46530 | 120 | 93.0% |
| $18x_1 + 6x_2 + 2x_3 + 12x_4 + 4x_5 = 18$ | 40938 | 70 | 81.8% |
| $6x_1 - x_2 + 12x_3 + 10x_4 + 12x_6 = 3$ | 49998 | 90 | 100% |
| $4x_1 + 4x_2 + 8x_3 - x_4 + 18x_5 - 2x_7 = 2$ | 7724 | 10 | 15.4% |
| $-2x_1 - 2x_2 + 14x_3 + 6x_4 + 13x_5 - x_6 + 17x_7 = 6$ | 0 | 0 | 0% |
| $-x_1 + 9x_2 + 16x_3 + 6x_4 + 9x_5 + 3x_6 + 5x_7 = 13$ | 6836 | 20 | 13.7% |
| $-2x_1 + 15x_2 + 12x_3 + 16x_4 + 5x_5 + 13x_6 + 3x_7 = 12$ | 6836 | 20 | 13.7% |
| $14x_1 + 14x_2 + 2x_3 + 14x_4 + x_5 + 6x_6 + 12x_7 = 2$ | 49998 | 90 | 100% |
| $13x_1 - x_3 + 2x_4 - 2x_5 + 15x_6 + 11x_7 = 1$ | 49998 | 80 | 100% |
| $13x_1 + 3x_2 - x_3 + 17x_4 + 13x_5 + 15x_6 - x_7 = 8$ | 30181 | 60 | 60.3% |
| $9x_1 + 3x_2 + 10x_3 + 15x_4 + 8x_5 + 8x_6 + 7x_7 = -1$ | 45101 | 200 | 90.2% |
| $13x_1 + 17x_3 + 5x_4 + 5x_5 + 8x_6 + 5x_7 = 13$ | 43653 | 170 | 87.3% |
| $-x_1 + 8x_2 - 2x_3 + 18x_4 - x_5 + 6x_6 + 10x_7 = 0$ | 0 | 0 | 0% |
| $2x_1 + 11x_2 + 18x_4 + 12x_5 + 16x_6 + 7x_7 = 14$ | 43131 | 80 | 86.2% |
| $15x_1 + 4x_2 + 7x_3 + 4x_4 + 7x_5 + 14x_6 + 5x_7 = 18$ | 38450 | 100 | 76.9% |
| $11x_1 + 4x_2 - 2x_3 + 2x_4 + 3x_5 + 7x_6 + 14x_7 = 8$ | 30181 | 40 | 60.3% |
| $-x_1 + 6x_2 + 14x_3 + 9x_4 - x_5 + 14x_6 + 12x_7 = 18$ | 19818 | 40 | 39.6% |
| $-x_1 + 2x_2 + 11x_3 + 18x_4 + 12x_5 + 7x_6 = 2$ | 0 | 0 | 0% |
| $12x_1 + 9x_2 + 16x_3 + 5x_5 + 18x_7 = 5$ | 49998 | 110 | 100% |
| $2x_1 + 18x_2 + 16x_3 + 17x_4 + x_6 + 5x_7 = 1$ | 46664 | 150 | 93.3% |
| $-2x_1 + 9x_2 + 5x_3 + 9x_4 + 15x_5 + 10x_6 + 13x_7 = 13$ | 49717 | 90 | 99.4% |
| $-x_1 + 14x_2 + 4x_3 + 3x_4 + 10x_5 + 9x_6 = -1$ | 49929 | 100 | 100% |
| $15x_1 + 13x_2 + 16x_3 + x_4 + 5x_5 + 2x_6 + 9x_7 = 10$ | 49999 | 90 | 100% |
| $2x_1 + 15x_2 + 11x_3 + 8x_4 + 9x_6 + 7x_7 = -1$ | 36407 | 60 | 72.8% |
| $17x_1 + 8x_2 + 17x_3 + 16x_4 + 17x_5 + 6x_6 + 6x_7 = 2$ | 42541 | 110 | 85.1% |
| $15x_1 + 7x_2 + 12x_3 - x_4 + 11x_5 + 10x_6 + 3x_7 = 0$ | 49999 | 90 | 100% |
| $8x_1 + 18x_3 + 6x_4 + 6x_5 + 8x_6 + x_7 = 4$ | 26152 | 70 | 52.3% |
| $14x_1 + 17x_2 + 7x_3 + 2x_4 + x_5 - x_6 + 10x_7 = 6$ | 30180 | 60 | 60.3% |
| $x_1 - 2x_2 + 2x_3 + 8x_4 + 18x_5 + 5x_6 + 8x_7 = 15$ | 2 | 0 | 0% |

Table F11

The half-space range search time (milliseconds) when dimension k = 7, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $16x_1 + 12x_2 + 13x_3 + 9x_5 + 6x_6 + 16x_7 = 0$ | 99997 | 200 | 100% |
| $18x_1 + 18x_2 + 7x_3 + 18x_5 + 13x_6 + 14x_7 = 8$ | 82855 | 150 | 82.8% |
| $11x_1 + 5x_2 + 9x_3 + 4x_4 - x_5 - 2x_6 + 6x_7 = 3$ | 60153 | 120 | 60.2% |
| $8x_1 + x_3 + 17x_4 + 5x_5 + 16x_6 + 11x_7 = 7$ | 94543 | 210 | 94.5% |
| $3x_1 + x_2 + 6x_3 + 13x_4 + 11x_5 + 12x_6 + 5x_7 = 2$ | 86432 | 180 | 86.4% |
| $14x_1 + 6x_2 + 5x_3 - 2x_4 - x_5 + 8x_6 = 4$ | 60151 | 110 | 60.2% |
| $11x_1 + 10x_2 + 16x_3 + 10x_4 + 9x_5 + 17x_7 = 1$ | 99998 | 210 | 100% |
| $17x_1 + 10x_2 + 5x_3 - x_5 + 11x_6 + 13x_7 = 9$ | 99998 | 210 | 100% |
| $16x_1 + 13x_2 + 18x_4 + 8x_5 + 14x_6 - 2x_7 = 13$ | 60152 | 170 | 60.2% |
| $17x_1 - x_3 + 6x_4 + 18x_5 + 13x_6 + 14x_7 = 5$ | 60152 | 110 | 60.2% |
| $8x_1 + 7x_2 + x_3 + 4x_5 + x_6 - x_7 = 3$ | 30388 | 80 | 30.4% |
| $11x_1 + x_3 + 10x_4 + 15x_5 + 2x_6 + 16x_7 = 6$ | 99998 | 210 | 100% |
| $-2x_1 + 5x_2 + 18x_3 + 11x_4 + 4x_5 + 8x_6 + x_7 = 0$ | 0 | 0 | 0% |
| $14x_1 + 3x_2 + 2x_3 + 13x_4 + 15x_5 + 5x_6 + 7x_7 = 11$ | 80813 | 180 | 80.8% |
| $10x_1 + 12x_3 + 3x_5 + 13x_6 + 3x_7 = 7$ | 99997 | 190 | 100% |
| $16x_1 + 13x_2 + 3x_4 + 11x_5 + 18x_6 - x_7 = 14$ | 84302 | 210 | 84.3% |
| $9x_1 + 2x_2 + 3x_3 + 9x_6 + 3x_7 = 3$ | 99997 | 180 | 100% |
| $15x_1 + 16x_2 + 11x_3 + 3x_4 + 7x_5 + 14x_6 - x_7 = 10$ | 60153 | 180 | 60.2% |
| $10x_1 + 4x_2 - 2x_3 + 5x_4 + 14x_5 + 16x_6 + 14x_7 = 6$ | 99999 | 240 | 100% |
| $5x_1 + 10x_2 + 17x_3 + 3x_4 + 18x_5 + 8x_6 + 13x_7 = 15$ | 96017 | 340 | 96.0% |
| $4x_1 + 12x_2 + 10x_3 + 6x_5 + 11x_6 - x_7 = 0$ | 93269 | 210 | 93.3% |
| $2x_1 + 7x_2 + 2x_3 + 13x_4 + 14x_5 + 9x_6 + 3x_7 = 3$ | 86432 | 180 | 86.4% |
| $11x_1 + 13x_2 + 18x_3 + 14x_4 + 3x_5 + 15x_6 + 10x_7 = -2$ | 99998 | 190 | 100% |
| $5x_1 + 11x_2 + 3x_3 + 16x_4 + 14x_5 + 12x_6 + 16x_7 = 7$ | 99998 | 230 | 100% |
| $17x_1 + 2x_2 + 3x_3 + 6x_4 + 3x_5 + 13x_6 + 12x_7 = 10$ | 85368 | 160 | 85.4% |
| $16x_1 + 5x_2 + 18x_3 + 3x_4 + 13x_5 + 16x_6 - 2x_7 = 14$ | 60151 | 180 | 60.2% |
| $2x_1 + 17x_2 + 9x_3 + 13x_4 - x_5 + 11x_6 + 2x_7 = 8$ | 99999 | 190 | 100% |
| $7x_1 - 2x_2 + 16x_3 + 18x_4 + 10x_5 + 11x_6 + 9x_7 = 8$ | 99998 | 190 | 100% |
| $-x_1 - x_2 + 16x_3 + 14x_4 + 17x_5 + 16x_6 = -1$ | 0 | 0 | 0% |
| $10x_1 + 4x_2 + 12x_3 + x_4 + 12x_5 + 4x_6 + 11x_7 = 12$ | 99998 | 260 | 100% |
| $18x_1 + 8x_2 + 5x_4 + 16x_5 + 14x_6 + 15x_7 = 4$ | 87165 | 190 | 87.2% |
| $17x_1 + 15x_2 + 15x_3 + 18x_4 - x_5 + 17x_6 + 8x_7 = 15$ | 60153 | 120 | 60.2% |
| $x_1 + 7x_2 + 2x_3 + x_4 + 7x_5 + 3x_7 = 1$ | 67083 | 950 | 67.1% |
| $8x_1 + 7x_2 + 13x_3 + 11x_4 + 17x_5 + 11x_6 = 8$ | 86430 | 170 | 86.4% |
| $10x_1 + 11x_2 + 15x_3 + x_4 + 3x_5 + 18x_6 + 4x_7 = 18$ | 99998 | 200 | 100% |
| $12x_1 + 2x_2 + 15x_3 + 13x_4 + 13x_5 - x_6 = 0$ | 60152 | 120 | 60.2% |
| $16x_1 + 9x_2 + 18x_3 + 6x_4 + 8x_6 + 5x_7 = -1$ | 83006 | 200 | 83.0% |
| $15x_1 - x_2 + 14x_3 + 9x_4 + 11x_5 + 17x_6 + 12x_7 = 3$ | 99997 | 210 | 100% |
| $16x_1 + 7x_2 + 13x_3 + 18x_4 + 15x_6 + 7x_7 = 16$ | 78144 | 160 | 78.1% |
| $2x_1 + 8x_2 + 16x_4 + x_6 - x_7 = 15$ | 129 | 0 | 0.1% |
| $14x_1 + 7x_4 + 11x_5 + 3x_6 + 15x_7 = 17$ | 83673 | 160 | 83.7% |
| $4x_1 + 18x_2 + 17x_3 + 7x_4 + 17x_5 + 14x_6 + 9x_7 = -2$ | 95871 | 220 | 95.9% |
| $-x_1 - x_2 + 2x_3 + 2x_4 + x_5 + 10x_6 + 3x_7 = 14$ | 0 | 0 | 0% |
| $15x_1 + 16x_2 + 11x_3 + 10x_4 + 10x_5 + 18x_6 - x_7 = 11$ | 99998 | 180 | 100% |
| $12x_1 + 11x_2 + 18x_4 + 14x_5 + 10x_6 = 17$ | 91434 | 180 | 91.4% |
| $13x_1 - x_2 + 5x_4 + 2x_5 + 9x_6 + 16x_7 = 8$ | 83528 | 150 | 83.5% |
| $4x_1 + 12x_2 - 2x_3 + x_5 + 3x_6 + 2x_7 = 14$ | 60152 | 120 | 60.2% |

Table F12

The half-space range search time (milliseconds) when dimension k = 7, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $18x_1 + 13x_2 + 7x_3 + 3x_4 + 13x_5 + 4x_6 + 5x_7 = 5$ | 81063 | 160 | 81.1% |
| $-x_1 + x_2 + 7x_3 + 12x_4 + x_5 + 3x_6 + x_7 = 17$ | 0 | 0 | 0% |
| $13x_1 + 9x_2 + 6x_3 + 11x_5 + x_6 + 18x_7 = 9$ | 99999 | 200 | 100% |
| $12x_1 + 5x_2 + 6x_3 + 5x_4 + 16x_5 + 2x_6 + x_7 = 17$ | 60151 | 120 | 60.2% |
| $10x_1 + 13x_2 + 13x_4 + 6x_5 + 10x_6 + 15x_7 = 3$ | 99997 | 190 | 100% |
| $16x_1 + 15x_2 + 15x_3 + 10x_4 + 15x_5 + 13x_6 + 12x_7 = 2$ | 99997 | 220 | 100% |
| $2x_1 + 12x_3 + 7x_4 + 17x_5 + 5x_6 + 15x_7 = 0$ | 87661 | 340 | 87.7% |
| $15x_1 + 16x_2 + 12x_3 + 12x_4 + 6x_6 - x_7 = 8$ | 85848 | 170 | 85.8% |
| $4x_1 + 14x_2 + 8x_3 + 12x_4 + 7x_5 + 17x_6 + 14x_7 = 9$ | 92992 | 250 | 93.0% |
| $3x_1 + 10x_2 + 17x_3 - x_4 + 16x_5 + 4x_6 + 9x_7 = 6$ | 99998 | 310 | 100% |
| $9x_1 + 16x_2 + 12x_3 + 5x_5 + 10x_6 + 16x_7 = 0$ | 99997 | 280 | 100% |
| $12x_1 + 2x_2 + 15x_3 + 13x_4 + 13x_5 - x_6 = 0$ | 60152 | 120 | 60.2% |
| $5x_1 + 13x_2 - x_3 + 9x_4 + x_5 + 10x_6 + 6x_7 = 9$ | 60151 | 110 | 60.2% |
| $4x_1 + 2x_2 + 15x_3 + 12x_4 + 12x_5 - x_6 + 12x_7 = 14$ | 99982 | 210 | 100% |
| $11x_1 + 2x_2 + 4x_3 + x_4 + x_5 + 9x_6 + 8x_7 = 0$ | 91699 | 500 | 91.7% |
| $7x_1 - x_2 + 13x_3 + 9x_4 + 11x_5 + 17x_6 + 3x_7 = 18$ | 99999 | 230 | 100% |
| $-x_1 + 11x_2 + 14x_3 + 18x_6 = 0$ | 23550 | 150 | 23.6% |
| $10x_1 + 2x_2 + 14x_3 + 11x_4 + 11x_5 - x_6 + 17x_7 = 1$ | 99997 | 190 | 100% |
| $7x_1 + 6x_2 + 4x_3 + 3x_4 + x_5 + 12x_6 + 13x_7 = -1$ | 73335 | 160 | 73.3% |
| $4x_1 + 13x_2 + 8x_3 - 2x_4 + 10x_5 + 14x_7 = 10$ | 99999 | 140 | 100% |
| $11x_1 + 13x_2 - 2x_3 + 7x_4 + 11x_6 + 10x_7 = 18$ | 99997 | 190 | 100% |
| $12x_1 + x_2 + 18x_3 + 16x_4 + 9x_5 + 9x_6 + 5x_7 = 9$ | 99998 | 140 | 100% |
| $2x_1 + 6x_2 + 16x_3 + x_4 + 18x_5 + 5x_6 = 11$ | 9749 | 80 | 97.5% |
| $9x_1 + 6x_2 + 5x_3 + 11x_4 + 7x_5 + 16x_6 + 17x_7 = 18$ | 78488 | 230 | 78.5% |
| $10x_1 + 15x_2 + 4x_3 + 16x_5 + 14x_6 + 13x_7 = 10$ | 99999 | 190 | 100% |
| $16x_1 + 7x_2 + 13x_3 + 3x_4 + 3x_5 - x_6 + 8x_7 = 17$ | 60153 | 110 | 60.2% |
| $9x_1 + 18x_2 + 17x_3 + 12x_5 + 10x_6 + 13x_7 = 5$ | 99999 | 200 | 100% |
| $8x_1 + 11x_3 + 18x_4 + 14x_6 - x_7 = 10$ | 60151 | 120 | 60.2% |
| $x_1 + 10x_2 + 15x_3 + 14x_4 + 9x_5 + 5x_6 + 4x_7 = -1$ | 99924 | 220 | 100% |
| $17x_1 + x_3 + 15x_4 + 17x_5 + 7x_6 - x_7 = 16$ | 60152 | 200 | 60.2% |
| $18x_1 - x_3 + 18x_4 + 3x_5 + 15x_7 = 6$ | 60153 | 130 | 60.2% |
| $16x_1 + 2x_2 + 13x_3 + 16x_4 + 13x_5 + x3_6 = 11$ | 99999 | 190 | 100% |
| $-x_1 + 12x_2 + 11x_3 + 11x_4 + x_5 + 3x_6 + 15x_7 = 9$ | 69716 | 130 | 69.7% |
| $-x_1 + 6x_2 + 7x_3 + 8x_4 + 7x_5 + 11x_6 + 9x_7 = 0$ | 0 | 0 | 0% |
| $17x_1 - x_3 + 14x_5 + 9x_6 + 14x_7 = 4$ | 99998 | 180 | 100% |
| $5x_1 + 11x_2 + 14x_3 + 15x_4 - 2x_5 + 14x_6 + 8x_7 = 4$ | 60152 | 110 | 60.2% |
| $8x_1 + 11x_2 + 15x_3 + 2x_4 + 3x_5 + 18x_6 + 13x_7 = 3$ | 78144 | 150 | 78.1% |
| $2x_1 + 8x_2 + 4x_4 + 14x_5 + 15x_6 - x_7 = 13$ | 308 | 0 | 0.3% |
| $-2x_1 + 4x_2 + 9x_3 + 11x_4 + 2x_5 + x_6 + 14x_7 = 10$ | 69609 | 150 | 69.6% |
| $12x_1 + 16x_2 + 13x_3 + 7x_4 + 11x_5 + 14x_6 = -1$ | 99998 | 200 | 100% |
| $5x_1 + 5x_2 + 17x_3 + 3x_4 + 5x_6 + 5x_7 = 8$ | 86090 | 420 | 86.1% |
| $7x_1 + 5x_2 + 18x_3 + 12x_4 + 5x_5 + 8x_6 + 9x_7 = 6$ | 86502 | 290 | 86.5% |
| $11x_1 + x_3 + 10x_4 + 15x_5 + 2x_6 + 16x_7 = 6$ | 99998 | 280 | 100% |
| $-2x_1 + 3x_2 + 12x_3 + 14x_4 - x_5 + 12x_6 - x_7 = -1$ | 0 | 0 | 0% |
| $9x_1 + 8x_2 + 10x_3 + 7x_5 + 8x_6 + 15x_7 = 0$ | 89455 | 370 | 89.5% |
| $x_1 + 12x_2 + 12x_3 + 10x_4 + 14x_5 + 15x_6 = 10$ | 99989 | 180 | 100% |
| $14x_1 + 7x_2 + 12x_3 + 16x_4 - 2x_5 + 16x_6 + 3x_7 = 18$ | 60152 | 120 | 60.2% |

Table F13

The half-space range search time (milliseconds) when dimension k = 10, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $17x_1+13x_2+9x_3+17x_4+14x_5+9x_7+10x_8+10x_9-x_{10}=10$ | 9999 | 20 | 100% |
| $3x_1+16x_2+18x_4+11x_5+17x_6-2x_7+11x_8+18x_9=5$ | 10000 | 30 | 100% |
| $x_1+x_2+18x_3+6x_4+7x_5+5x_6+17x_7+10x_8-2x_9-x_{10}=15$ | 16 | 0 | 0.2% |
| $6x_1-x_2+3x_3+17x_4+3x_5+6x_6+16x_7+4x_8+3x_9+13x_{10}=8$ | 2600 | 0 | 26.0% |
| $10x_1+17x_2+11x_3+7x_4+6x_6+14x_7-2x_8+9x_9+6x_{10}=1$ | 10000 | 20 | 100% |
| $5x_1+16x_2+x_3+5x_4+17x_5+2x_7+10x_8+7x_9-2x_{10}=10$ | 5387 | 10 | 53.9% |
| $12x_1-x_2+14x_3+6x_4+14x_5+16x_6+11x_7+11x_8+15x_9=5$ | 10000 | 20 | 100% |
| $-x_1+x_2+5x_3+7x_4+10x_5+12x_6+13x_8+2x_9=0$ | 0 | 0 | 0% |
| $11x_1+8x_2+10x_3+9x_4+4x_5+4x_6+18x_7+17x_8-x_9+3x_{10}=11$ | 5387 | 0 | 53.9% |
| $-2x_1+12x_2+x_3+10x_4+6x_7-2x_8+6x_9+5x_{10}=6$ | 0 | 0 | 0% |
| $4x_1+15x_2+14x_3+11x_4-2x_5+17x_6+16x_7+15x_9+7x_{10}=1$ | 5388 | 10 | 53.9% |
| $2x_1+10x_3+15x_5+6x_6+14x_7-2x_8+15x_9+5x_{10}=12$ | 5387 | 10 | 53.9% |
| $9x_1+3x_2+x_3+x_4+11x_5+x_6+2x_7+2x_9+7x_{10}=7$ | 9553 | 40 | 95.5% |
| $11x_1+15x_2+3x_3+7x_5+7x_6+11x_7+6x_8+6x_9+14x_{10}=-2$ | 7007 | 10 | 70.1% |
| $17x_1-2x_2+16x_3+x_4+4x_5+2x_6+8x_8+14x_9+15x_{10}=14$ | 5387 | 10 | 53.9% |
| $12x_1+3x_2+2x_3+9x_4+17x_5+5x_6+7x_7-x_8+13x_9+5x_{10}=11$ | 5386 | 10 | 53.9% |
| $5x_1+12x_2+14x_5+18x_6+6x_7+7x_8+2x_9+14x_{10}=8$ | 7373 | 10 | 73.7% |
| $-x_1+18x_2+17x_3+10x_4+10x_5+6x_6+6x_8+3x_9+13x_{10}=-2$ | 0 | 0 | 0% |
| $12x_1+4x_2+12x_4+4x_5-x_6+x_7+10x_8-x_9+16x_{10}=9$ | 5388 | 0 | 53.9% |
| $-x_1+7x_2+12x_3+13x_4+15x_6+11x_7+12x_8+7x_9+18x_{10}=4$ | 3969 | 0 | 39.7% |
| $13x_1-x_2+14x_3+13x_4+18x_5-2x_8+10x_9+3x_{10}=16$ | 9998 | 10 | 100% |
| $6x_1+x_2+6x_3+14x_4+15x_5+16x_6+8x_7-2x_9+5x_{10}=11$ | 5388 | 10 | 53.9% |
| $13x_1+5x_2-2x_3+15x_4+11x_5+12x_6+18x_7+x_8+6x_9+7x_{10}=6$ | 5388 | 10 | 53.9% |
| $6x_1-x_2+3x_3+2x_4+7x_5+10x_6+16x_7+5x_8+x_9+10x_{10}=12$ | 5386 | 10 | 53.9% |
| $17x_1+6x_2+4x_3-x_4+2x_5+12x_6+3x_7+x_8+15x_9+6x_{10}=17$ | 5387 | 10 | 53.9% |
| $-x_1+16x_2+10x_4-x_5+3x_6+15x_7+4x_8+8x_9+9x_{10}=6$ | 0 | 0 | 0% |
| $9x_1+3x_2+x_3+7x_4+15x_5+5x_6+2x_7+4x_{10}=10$ | 5386 | 10 | 53.9% |
| $16x_1+11x_2+2x_3+4x_4+10x_5+8x_6+11x_7+18x_8+14x_9=15$ | 10000 | 30 | 100% |
| $17x_1+11x_2+14x_3+2x_4+6x_5+x_6+8x_8+12x_9+12x_{10}=1$ | 9666 | 30 | 96.7% |
| $15x_1+17x_2+10x_3+11x_4+2x_5+2x_7+7x_8+13x_9+11x_{10}=12$ | 9998 | 20 | 100% |
| $9x_1+3x_2+x_3+16x_4+8x_5-2x_6+2x_7-x_8+5x_9+9x_{10}=3$ | 5387 | 10 | 53.9% |
| $4x_1+13x_2+9x_3+17x_4+15x_5+18x_7+16x_8+7x_9+5x_{10}=18$ | 7297 | 20 | 73.0% |
| $5x_1+17x_2+9x_3+11x_4+9x_5+5x_7+x_8+10x_9+11x_{10}=14$ | 10000 | 20 | 100% |
| $12x_1+4x_2+6x_4+15x_6+x_7+9x_8+x_9-2x_{10}=5$ | 9998 | 20 | 100% |
| $10x_1+6x_2+16x_3+13x_4+17x_5+x_6+18x_8+12x_9+6x_{10}=7$ | 5386 | 10 | 53.9% |
| $11x_1+14x_2+17x_3+9x_4+12x_5+3x_6+4x_8+5x_9+2x_{10}=11$ | 9999 | 20 | 100% |
| $5x_1+15x_2+4x_3+2x_4+9x_5+6x_6+7x_7-2x_8+7x_{10}=12$ | 2599 | 10 | 26.0% |
| $17x_1+10x_2+6x_3+6x_5+14x_6+17x_7+15x_8+13x_9+3x_{10}=11$ | 8101 | 20 | 81.0% |
| $10x_1+3x_2+13x_32x_5+12x_6+15x_7-x_8+9x_9+7x_{10}=18$ | 9999 | 30 | 100% |
| $4x_1+4x_2+5x_4+15x_6+14x_7+3x_8+4x_9+12x_{10}=-2$ | 7732 | 20 | 77.3% |
| $16x_1+12x_2+2x_4+16x_5+18x_6+x_7+18x_9+7x_{10}=2$ | 6771 | 20 | 67.7% |
| $7x_1+7x_2+3x_3+4x_4+13x_5+4x_6+11x_7+7x_8+10x_9+4x_{10}=1$ | 7812 | 20 | 78.1% |
| $11x_1+4x_2+10x_3+15x_4+5x_6+10x_7+10x_8+16x_9-2x_{10}=15$ | 9999 | 20 | 100% |
| $5x_1+4x_2+9x_3+6x_4+3x_5+x_6+18x_7+6x_8+9x_9+14x_{10}=3$ | 5388 | 10 | 53.9% |
| $13x_1+11x_3+9x_4+9x_6+6x_7+2x_8+x_9+10x_{10}=2$ | 9998 | 10 | 100% |
| $17x_1-2x_3-x_4+18x_5+10x_6+5x_7+17x_8+7x_9+4x_{10}=17$ | 5386 | 10 | 53.9% |
| $18x_1+8x_2-x_4+14x_5+15x_6+14x_7+3x_8+10x_9+11x_{10}=7$ | 10000 | 20 | 100% |

Table F14

The half-space range search time (milliseconds) when dimension k = 10, size n = $10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $x_1+x_3+6x_4+16x_5+2x_6+6x_7+8x_8+2x_9+15x_{10}=3$ | 143 | 0 | 1.4% |
| $10x_1+8x_2-x_3-2x_4+13x_5+15x_6+5x_7+18x_8+13x_9+3x_{10}=0$ | 5387 | 10 | 53.9% |
| $12x_1-2x_4+9x_5+14x_7+3x_8+16x_9+10x_{10}=12$ | 10000 | 20 | 100% |
| $7x_1+12x_2+10x_3+11x_4+3x_5+7x_6+x_7+15x_8+15x_9=4$ | 9999 | 20 | 100% |
| $8x_1+3x_2+11x_3+10x_4+12x_6+10x_7+x_8+18x_9+7x_{10}=17$ | 9999 | 10 | 100% |
| $15x_1+7x_2+3x_3+11x_4+17x_5+8x_6-x_7+2x_8+5x_9+9x_{10}=12$ | 5387 | 20 | 53.9% |
| $11x_1+6x_2+15x_3+10x_4+13x_5+x_6+7x_7+14x_8+3x_9=0$ | 9998 | 30 | 100% |
| $4x_1+7x_2+2x_3+2x_4+11x_5+5x_6+6x_7-x_8-x_9+4x_{10}=0$ | 33 | 0 | 0.3% |
| $-2x_1+8x_3+10x_4+6x_5+2x_6+5x_7+2x_8+14x_9+7x_{10}=6$ | 0 | 0 | 0% |
| $12x_1+15x_2+14x_3-2x_4+x_5+3x_7+6x_8+10x_9+11x_{10}=12$ | 9998 | 20 | 100% |
| $9x_1-x_2+x_3+13x_4+6x_6+15x_7+15x_8+18x_9-x_{10}=14$ | 5387 | 10 | 53.9% |
| $11x_1+x_2+5x_3+13x_4+13x_5+16x_6+12x_7+8x_8+3x_9+12x_{10}=18$ | 9999 | 20 | 100% |
| $7x_1+x_2+17x_3+11x_4+9x_5+9x_6-x_8+x_9+2x_{10}=4$ | 9999 | 30 | 100% |
| $17x_1+16x_2+x_3-2x_4+4x_5+7x_6-x_7+3x_8+18x_9+6x_{10}=11$ | 5387 | 10 | 53.9% |
| $10x_1+4x_2-x_3+10x_4+x_5-x_6+18x_7+12x_8+7x_9+16x_{10}=8$ | 5387 | 10 | 53.9% |
| $14x_1+x_2+6x_3-2x_5+17x_7+6x_8+13x_9+10x_{10}=1$ | 5387 | 10 | 53.9% |
| $15x_1+13x_2+7x_3+15x_5+4x_6+5x_7+13x_8+16x_9+16x_{10}=13$ | 8101 | 10 | 81.0% |
| $16x_1+18x_2+7x_3+14x_4+9x_5+3x_6+13x_7-x_8-x_9+x_{10}=9$ | 5387 | 10 | 53.9% |
| $14x_1+15x_2+14x_3+4x_4+6x_5+4x_6+12x_7+13x_8+4x_9+16x_{10}=2$ | 6773 | 10 | 67.7% |
| $17x_1+5x_2+6x_3+2x_4-x_5+2x_6+9x_7+11x_8+5x_9+13x_{10}=1$ | 5388 | 10 | 53.9% |
| $x_1+2x_2+14x_3+13x_4+16x_5+2x_6+7x_7+4x_8+11x_9+7x_{10}=16$ | 7 | 0 | 0.1% |
| $14x_1+17x_2-x_3+11x_5+5x_7+8x_8+6x_9+11x_{10}=1$ | 5386 | 10 | 53.9% |
| $18x_1+11x_3+x_4+8x_5+17x_6+15x_7+10x_8+15x_9+12x_{10}=17$ | 9999 | 20 | 100% |
| $11x_1+10x_3+13x_4+x_5+14x_6+x_7+6x_8+8x_9+8x_{10}=5$ | 9709 | 40 | 97.1% |
| $6x_1+12x_2-x_3+5x_4+16x_5+x_6+10x_7+17x_8+6x_9-2x_{10}=-2$ | 5388 | 20 | 53.9% |
| $13x_1+16x_2+11x_3+6x_4+13x_5+18x_6-x_7-x_8+14x_9=14$ | 5386 | 10 | 53.9% |
| $17x_1+13x_2-2x_3+17x_4+9x_5-2x_6+18x_7+13x_8+14x_{10}=7$ | 9097 | 20 | 90.9% |
| $6x_1+10x_2+4x_3+7x_4+6x_5-x_6+16x_7+6x_8+5x_9+8x_{10}=0$ | 1176 | 10 | 11.8% |
| $-x_1+16x_2+17x_4+2x_5+7x_6+15x_7+5x_8+5x_9+6x_{10}=10$ | 0 | 10 | 0% |
| $12x_1+9x_2+7x_3+3x_4-2x_5+5x_6+13x_7+9x_8+10x_{10}=16$ | 10000 | 20 | 100% |
| $17x_1+6x_2+14x_3+14x_4+15x_5+5x_6+12x_7+3x_8+6x_9+4x_{10}=9$ | 8273 | 20 | 82.7% |
| $15x_1+12x_2+10x_3+3x_4+11x_5+15x_6+10x_7+2x_8+7x_9+2x_{10}=1$ | 10000 | 20 | 100% |
| $6x_1+7x_2+13x_3+5x_4+8x_5+2x_6-x_7-x_8=-2$ | 1231 | 10 | 12.3% |
| $18x_1+15x_2+13x_3+2x_4+3x_5+4x_6+7x_7+16x_8+13x_9+16x_{10}=1$ | 5387 | 10 | 53.9% |
| $8x_1+9x_2-2x_3+4x_4+4x_5+7x_6-2x_8+5x_9+8x_{10}=13$ | 5388 | 20 | 53.9% |
| $11x_1+16x_2-2x_4+12x_5+17x_6+6x_7+18x_8+15x_9+7x_{10}=13$ | 5387 | 10 | 53.9% |
| $9x_1+x_2+18x_3+7x_4+8x_5+5x_6+4x_7+16x_8+16x_9+5x_{10}=1$ | 7680 | 30 | 76.8% |
| $17x_1+9x_2+9x_3+x_4+x_7+3x_8+6x_9+13x_{10}=14$ | 9999 | 20 | 100% |
| $13x_1+8x_2+16x_5+15x_6+10x_7+15x_8+4x_9+4x_{10}=1$ | 8100 | 20 | 81.0% |
| $x_1-x_2+x_3+13x_4-x_5+6x_6+6x_7+9x_8+13x_{10}=7$ | 19 | 10 | 0.2% |
| $5x_1+17x_2+8x_3+2x_4+16x_5+7x_6+5x_7+3x_8+6x_9+6x_{10}=0$ | 7048 | 60 | 70.5% |
| $3x_1+15x_2+2x_3+6x_4+10x_5+11x_6+3x_7+x_8+6x_9+4x_{10}=15$ | 2600 | 10 | 26% |
| $7x_1+12x_2+10x_3+17x_4+7x_5+11x_6+x_7+16x_8+12x_9-x_{10}=8$ | 9999 | 30 | 100% |
| $11x_1+9x_2+17x_3+7x_4+3x_5+12x_6+10x_8+18x_9+13x_{10}=1$ | 10000 | 10 | 100% |
| $4x_1+2x_2+2x_3+15x_4-x_5+9x_6-x_7+14x_8+13x_9+16x_{10}=8$ | 5360 | 10 | 53.6% |
| $5x_1+15x_2+3x_3+14x_4+16x_5+14x_6+7x_7+16x_9+2x_{10}=-1$ | 8289 | 20 | 82.9% |
| $12x_1+18x_2+16x_3+16x_4+13x_5+10x_6+17x_7+x_8+4x_9+3x_{10}=15$ | 10000 | 20 | 100% |

Table F15

The half-space range search time (milliseconds) when dimension k = 10, size n = $5*10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $3x_1+14x_2+5x_3+16x_4+9x_5+5x_6+8x_7+12x_8+16x_9+9x_{10}=4$ | 49994 | 80 | 100% |
| $5x_1+13x_2+6x_3+3x_4+15x_5+8x_6+13x_7+10x_8+5x_9+7x_{10}=9$ | 37075 | 130 | 74.1% |
| $11x_1+11x_2+12x_3+4x_4+5x_5+5x_6-x_7+3x_9+17x_{10}=11$ | 26833 | 50 | 53.7% |
| $18x_1+9x_2+6x_3+8x_4+9x_6+17x_7-2x_8+4x_9+15x_{10}=5$ | 26833 | 60 | 53.7% |
| $3x_1+5x_2+17x_3+3x_4+15x_5+4x_7+13x_9+16x_{10}=5$ | 34853 | 70 | 79.7% |
| $-x_1+x_2+5x_3+11x_4+3x_5+7x_6+17x_8+15x_9+12x_{10}=15$ | 0 | 0 | 0% |
| $6x_1+9x_2+17x_3+6x_4+15x_5+2x_6+18x_7+3x_8+5x_9-x_{10}=6$ | 13033 | 40 | 26.1% |
| $3x_1+9x_3+3x_4+7x_5+14x_7+x_8+6x_9+17x_{10}=5$ | 43986 | 200 | 87.9% |
| $3x_1+17x_2+7x_3+13x_4+15x_6+x_7+7x_8+10x_9+x_{10}=7$ | 48844 | 150 | 97.7% |
| $3x_1+14x_2+5x_3+13x_5+9x_6+8x_7+13x_8+14x_9+6x_{10}=8$ | 38972 | 240 | 77.9% |
| $17x_1+3x_2+9x_3+18x_4+14x_7+x_8+13x_9+16x_{10}=16$ | 49999 | 80 | 100% |
| $14x_1+15x_2+x_3+16x_4+14x_5-x_6+11x_7+14x_9+13x_{10}=15$ | 49999 | 90 | 100% |
| $4x_1+16x_2+6x_4+7x_5+16x_6-x_7+16x_8+7x_9+8x_{10}=3$ | 66 | 0 | 1.3% |
| $-x_1+7x_3+14x_4-x_5-2x_6+5x_7+5x_8+5x_9-2x_{10}=0$ | 0 | 0 | 0% |
| $2x_1+15x_2+12x_3+13x_4+8x_5+12x_6+11x_7+5x_8+15x_9+8x_{10}=1$ | 49999 | 80 | 100% |
| $16x_1+17x_3+15x_4+2x_5+4x_6+9x_7+8x_8+10x_9=6$ | 46645 | 100 | 93.2% |
| $9x_1+9x_2+4x_3+5x_4+16x_5+10x_6+17x_7+9x_8-x_9=11$ | 49999 | 100 | 100% |
| $16x_1+15x_2+12x_3+12x_4+8x_5+12x_6+3x_7+18x_9+11x_{10}=8$ | 49999 | 80 | 100% |
| $11x_1+3x_2+2x_4+18x_6+10x_7+5x_9+11x_{10}=14$ | 50000 | 90 | 100% |
| $9x_1+2x_2+11x_3+14x_4+13x_5+7x_6+11x_7+4x_8+8x_9-2x_{10}=11$ | 50000 | 100 | 100% |
| $-x_1+17x_2+8x_3+4x_5-x_6+18x_7+2x_9+13x_{10}=4$ | 0 | 0 | 0% |
| $9x_1+17x_2+6x_3+11x_4-2x_5+15x_6+5x_7+16x_8+16x_9+8x_{10}=13$ | 26833 | 50 | 53.7% |
| $12x_1+8x_2+7x_3+2x_4+8x_5+17x_6+18x_8+12x_9+10x_{10}=14$ | 44392 | 160 | 88.7% |
| $8x_1+3x_2+2x_4+x_5+18x_6-x_7+5x_8+2x_9+18x_{10}=0$ | 13032 | 30 | 26.1% |
| $5x_1+14x_2+13x_3+14x_5+16x_6+16x_7+3x_8+3x_9+16x_{10}=0$ | 48868 | 100 | 97.7% |
| $13x_1+8x_2+6x_3+x_4+7x_5+17x_6+13x_7+12x_8+15x_9+3x_{10}=6$ | 45614 | 140 | 91.2% |
| $18x_1+10x_2+13x_3+6x_4-x_5+17x_6+12x_8+3x_9+2x_{10}=17$ | 50000 | 140 | 100% |
| $18x_1+4x_2+6x_3+6x_4+11x_5+18x_6+9x_7+14x_8+18x_9+3x_{10}=17$ | 26834 | 50 | 53.6% |
| $-2x_1+5x_2+3x_3+9x_4-x_5+11x_6+4x_7+5x_8+3x_9+15x_{10}=14$ | 0 | 0 | 0% |
| $14x_1+5x_3+2x_4+10x_5+15x_6-2x_8+10x_9+7x_{10}=1$ | 50000 | 150 | 100% |
| $2x_1+3x_2+8x_3+x_4+2x_5+4x_6+18x_7+11x_8+16x_9=5$ | 29075 | 220 | 58.1% |
| $6x_1-2x_2+13x_3+12x_5-x_6+3x_7+10x_8+5x_9-x_{10}=0$ | 26833 | 30 | 53.7% |
| $8x_1+14x_2+14x_3+15x_4+2x_5+2x_6+2x_8+11x_9+11x_{10}=7$ | 49999 | 90 | 100% |
| $5x_1+4x_2+6x_3+13x_4+16x_5+18x_7+12x_9+8x_{10}=7$ | 32017 | 70 | 64.0% |
| $16x_1+9x_2+14x_3+7x_5+3x_6+4x_7+11x_8+11x_9-2x_{10}=4$ | 50000 | 100 | 100% |
| $11x_1+x_2+14x_3+11x_4+18x_5+5x_6+14x_8+7x_9=4$ | 46060 | 100 | 92.1% |
| $13x_1+6x_2+2x_3+17x_6+15x_7+7x_8+9x_9+3x_{10}=-2$ | 40510 | 120 | 81.0% |
| $6x_1+18x_2+6x_3+18x_4+9x_5+8x_6+17x_8+8x_9+13x_{10}=5$ | 38651 | 140 | 77.7% |
| $14x_1+12x_2-2x_4+2x_5+9x_6-2x_7+4x_8-x_9=12$ | 49999 | 90 | 100% |
| $2x_1+18x_2+3x_3-x_4+8x_5+12x_6+12x_7+x_8+2x_9+10x_{10}=7$ | 26811 | 50 | 53.7% |
| $9x_1+4x_2+16x_3+15x_4+8x_6+9x_7+9x_8+14x_9+18x_{10}=-1$ | 50000 | 130 | 100% |
| $12x_1+7x_2-x_3+12x_4+14x_5+18x_6+6x_7+2x_8+10x_{10}=2$ | 46720 | 80 | 53.5% |
| $3x_1+6x_2+10x_4+4x_5+4x_6+2x_7+5x_8+17x_9+12x_{10}=18$ | 31542 | 110 | 63.1% |
| $2x_1+9x_2+15x_3+8x_4+13x_5+7x_6+8x_7+9x_8+17x_{10}=8$ | 43766 | 160 | 87.5% |
| $-x_1+10x_2+14x_3+x_4-2x_5+12x_6+3x_7+8x_8+18x_9+5x_{10}=14$ | 0 | 0 | 0% |
| $16x_1-2x_2+14x_3+10x_4+7x_5+11x_6+3x_9+17x_{10}=6$ | 49999 | 90 | 100% |
| $9x_1+9x_3+17x_4+2x_5+18x_6-2x_7+9x_8+15x_9+5x_{10}=8$ | 50000 | 90 | 100% |

Table F16

The half-space range search time (milliseconds) when dimension k = 10, size n = $5*10^4$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $3x_1+4x_2+16x_3+16x_4+7x_6+5x_7+8x_9+11x_{10}=13$ | 27779 | 150 | 55.5% |
| $14x_1-2x_2+13x_3+x_4+13x_5-x_6+12x_7+17x_8+11x_9+5x_{10}=6$ | 26833 | 60 | 53.7% |
| $-x_1+6x_2+x_3+12x_4+5x_5+4x_6-x_7+17x_8+11x_9+5x_{10}=12$ | 0 | 0 | 0% |
| $17x_1+x_2+14x_3+3x_4+13x_5+x_6+3x_7+15x_9+9x_{10}=7$ | 50000 | 90 | 100% |
| $16x_1+11x_2+10x_3+6x_4+3x_5+10x_6+10x_7+5x_8-x_9+14x_{10}=13$ | 50000 | 90 | 100% |
| $18x_1+6x_2+11x_3+15x_5+14x_6+7x_7-2x_8+4x_9+5x_{10}=0$ | 26834 | 50 | 53.7% |
| $14x_1+10x_2+x_3+13x_4+5x_5+7x_6+3x_7+16x_8+6x_9+2x_{10}=5$ | 42077 | 150 | 84.1% |
| $18x_1+18x_2+4x_3+6x_4+13x_5+16x_6+8x_7+15x_8+16x_9=4$ | 38401 | 70 | 76.8% |
| $6x_1+3x_2+10x_3+4x_5+6x_7+5x_8+x_9+2x_{10}=2$ | 39334 | 190 | 78.7% |
| $2x_1-x_3+7x_4+13x_5+7x_6+x_7+x_8+2x_9-x_{10}=12$ | 6 | 0 | 0% |
| $15x_1-x_2+x_3+3x_4+4x_5+15x_6-x_7+4x_8-x_9=7$ | 26833 | 50 | 53.7% |
| $15x_1+3x_2+18x_4-x_5+14x_6+6x_7+11x_8+2x_9+6x_{10}=3$ | 26833 | 50 | 53.7% |
| $-2x_1+13x_2+5x_3+2x_4+14x_5+8x_6+4x_7+4x_8+8x_9=1$ | 0 | 0 | 0% |
| $8x_1+14x_2+4x_3+14x_4+7x_5+5x_6+12x_7+x_9+16x_{10}=11$ | 43940 | 100 | 87.9% |
| $3x_1-x_2+12x_3+8x_6-x_7+10x_8+5x_{10}=8$ | 6148 | 20 | 12.3% |
| $5x_1+14x_2+13x_3+15x_4+11x_5+12x_6+16x_7+2x_8+6x_9+5x_{10}=6$ | 26833 | 40 | 53.7% |
| $10x_1+2x_2+x_3+4x_4+4x_5+18x_6+2x_7+3x_8+15x_9-2x_{10}=0$ | 26834 | 50 | 53.7% |
| $10x_1-x_3+14x_4+17x_5+11x_6+10x_7+9x_8-2x_9+2x_{10}=2$ | 50000 | 90 | 100% |
| $10x_1-2x_3+4x_4+10x_5+7x_6+18x_7+4x_8+11x_9-x_{10}=11$ | 26834 | 50 | 53.7% |
| $12x_1+10x_2+14x_3+x_4+18x_5+13x_6+16x_7+2x_8-x_{10}=6$ | 50000 | 130 | 100% |
| $18x_1+2x_2+x_3+5x_4+4x_5+18x_6+11x_7+9x_8+12x_9+4x_{10}=8$ | 26834 | 50 | 53.7% |
| $15x_1+14x_2+15x_3+3x_4+18x_5+16x_6+8x_7+7x_8+13x_9+2x_{10}=7$ | 41366 | 90 | 82.7% |
| $7x_1+13x_2+17x_3+9x_5+2x_6+5x_7+10x_8+9x_9+4x_{10}=2$ | 49999 | 110 | 100% |
| $17x_1+6x_2+14x_3+6x_{45}+14x_6+12x_7+5x_8+2x_9=17$ | 40902 | 60 | 81.8% |
| $7x_1+6x_2+12x_3+18x_4+15x_5+10x_6-x_7+16x_9+15x_{10}=5$ | 26833 | 50 | 53.7% |
| $5x_1+9x_2+6x_3+15x_4+3x_5+13x_6+4x_7+5x_8-x_{10}=16$ | 13033 | 40 | 26.1% |
| $x_1+14x_2+14x_3+2x_4+16x_5+16x_6+12x_7+16x_8-2x_9+9x_{10}=14$ | 49976 | 90 | 100% |
| $13x_1+18x_3+4x_4+9x_5+8x_6+9x_7-x_8+4x_9+12x_{10}=4$ | 49999 | 90 | 100% |
| $8x_1+13x_2+7x_3+17x_4+3x_5+16x_6+18x_7+9x_8+13x_9+2x_{10}=17$ | 41486 | 70 | 82.9% |
| $-x_1+13x_2+5x_3+8x_4+18x_5+13x_6+4x_7+5x_8+5x_9+18x_{10}=5$ | 0 | 0 | 0% |
| $x_1+8x_3+x_4+4x_5+x_6+10x_7+4x_8+15x_9+17x_{10}=5$ | 45684 | 440 | 91.3% |
| $x_1+17x_2+6x_3+10x_4+18x_5+16x_6+17x_7+10x_8-x_9+x_{10}=6$ | 49993 | 90 | 100% |
| $x_1+14x_2+4x_3-x_4+10x_5+9x_6+4x_7+15x_8+x_9+6x_{10}=7$ | 5695 | 10 | 11.4% |
| $3x_1+17x_2+7x_3-2x_4+2x_5-x_6+x_7+8x_8+8x_9-x_{10}=10$ | 12 | 0 | 0% |
| $16x_1+6x_2+2x_3+13x_5+7x_6+7x_8+15x_9+17x_{10}=10$ | 33535 | 70 | 67.1% |
| $15x_1+16x_2-x_3+3x_4+3x_5+16x_6+6x_7+12x_8-2x_9+x_{10}=16$ | 26834 | 50 | 53.7% |
| $14x_1+5x_2+15x_3+7x_4+15x_5+4x_6+13x_7+17x_8+x_9+6x_{10}=1$ | 26834 | 60 | 53.7% |
| $9x_1+10x_2+2x_3+15x_4+7x_5+6x_6+7x_8+16x_{10}=-1$ | 37105 | 70 | 74.2% |
| $-x_1+3x_2-x_3-x_5-2x_6+6x_7+2x_8+14x_9+11x_{10}=13$ | 0 | 0 | 0% |
| $7x_1+16x_2-x_3+12x_4+17x_5+8x_6-2x_7+4x_8+5x_9=1$ | 13032 | 20 | 26.1% |
| $12x_1+18x_2+5x_3+16x_4+8x_5+8x_6+4x_7+4x_8+14x_9-x_{10}=11$ | 13032 | 30 | 26.1% |
| $-x_1+12x_2-x_3+17x_4+x_5+9x_6+x_7+13x_8+4x_9+7x_{10}=-2$ | 0 | 0 | 0% |
| $11x_1+11x_2+13x_4+13x_5+16x_6-x_7+16x_8+9x_{10}=14$ | 49999 | 80 | 100% |
| $4x_1+x_2+4x_3+9x_4+x_5+7x_6+4x_7+4x_8-2x_{10}=0$ | 13033 | 40 | 26.1% |
| $x_1+13x_2+18x_3+7x_4+14x_5+5x_6+x_7+2x_8+16x_{10}=0$ | 19930 | 130 | 39.9% |
| $12x_1+13x_2+16x_3-x_4+7x_5+2x_6+9x_7-2x_8+15x_9+11x_{10}=9$ | 50000 | 140 | 100% |
| $4x_1+12x_2-2x_3+16x_4+9x_6+5x_7+10x_9+14x_{10}=4$ | 49993 | 120 | 100% |

Table F17

The half-space range search time (milliseconds) when dimension k = 10, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $13x_1+2x_2+13x_3+17x_4-x_5+18x_6+17x_7+5x_9+8x_{10}=8$ | 53582 | 100 | 53.6% |
| $12x_1+4x_2+9x_3+11x_4+7x_6+8x_8+5x_9+15x_{10}=15$ | 80143 | 150 | 80.1% |
| $6x_1+2x_2+15x_3+12x_4+10x_5+4x_6+5x_7+18x_8+4x_9+3x_{10}=17$ | 77238 | 140 | 77.2% |
| $17x_1+7x_2+13x_3-2x_4-2x_5+4x_{10}=1$ | 53583 | 90 | 53.6% |
| $2x_1+7x_2+2x_3+7x_4+8x_5+10x_6+17x_7+6x_8+12x_9+12x_{10}=-1$ | 66724 | 710 | 66.7% |
| $-x_1+11x_2+3x_3+9x_4+7x_5+13x_6+5x_8+7x_9+8x_{10}=12$ | 0 | 0 | 0% |
| $10x_1+2x_2+3x_3+18x_5+15x_6+18x_7+7x_8+4x_9+10x_{10}=13$ | 77330 | 150 | 77.3% |
| $6x_1-x_2+12x_3+7x_4+6x_5+x_6+13x_7+4x_8+5x_9+6x_{10}=1$ | 25998 | 70 | 26.0% |
| $18x_1+11x_2+13x_3-2x_4+16x_5+3x_6+9x_7+6x_8+x_9+8x_{10}=1$ | 53582 | 110 | 53.6% |
| $17x_1+8x_3+7x_5+12x_6+16x_7+11x_8+5x_9+13x_{10}=8$ | 86583 | 160 | 86.6% |
| $16x_1+11x_2+11x_3+7x_4+11x_6+4x_7+10x_8+6x_9+3x_{10}=8$ | 99996 | 250 | 100% |
| $9x_1-2x_2+13x_3+13x_4+x_5+17x_6+18x_7+5x_8+8x_9+18x_{10}=17$ | 53582 | 100 | 53.6% |
| $4x_1+5x_2+16x_3+5x_4+8x_5+6x_6+15x_7-2x_8+9x_{10}=9$ | 25999 | 50 | 26.0% |
| $x_1+8x_2+6x_3+18x_4-x_5-x_6+11x_7+16x_8+x_9+6x_{10}=13$ | 9 | 0 | 0% |
| $5x_1+10x_2+13x_3+x_4+10x_5-x_6+18x_7+16x_8+10x_9+5x_{10}=2$ | 99997 | 170 | 100% |
| $11x_1+2x_2+6x_4+18x_5+3x_6+13x_7+x_8+2x_9+12x_{10}=4$ | 53581 | 80 | 53.6% |
| $5x_1+6x_3+8x_4+8x_5-x_7+12x_8=6$ | 12180 | 20 | 12.2% |
| $9x_1-2x_2+12x_3+10x_4-2x_5+18x_6+5x_7+x_8+10x_{10}=9$ | 99997 | 180 | 100% |
| $13x_1+5x_2+13x_3+15x_4-x_5+2x_6+18x_7+17x_8+2x_9+4x_{10}=18$ | 53581 | 100 | 53.7% |
| $13x_1+15x_2+16x_4+5x_5+4x_6+4x_7+6x_8+7x_9+13x_{10}=3$ | 66018 | 100 | 66.2% |
| $15x_1+x_2+14x_3+7x_4+8x_5+7x_6-2x_7+17x_8+14x_9+2x_{10}=11$ | 53581 | 120 | 53.6% |
| $4x_1-2x_3+13x_5+12x_7+2x_8+6x_9+8x_{10}=2$ | 53581 | 110 | 53.6% |
| $5x_1+9x_2+5x_3+x_4-x_5+2x_6+6x_7-x_8+8x_9+3x_{10}=16$ | 5928 | 10 | 5.9% |
| $3x_1+3x_2+7x_3+3x_4+16x_5+10x_6+7x_7+9x_8+3x_9+14x_{10}=8$ | 88243 | 360 | 88.2% |
| $7x_1+15x_2+9x_3+11x_4-2x_5-x_6+15x_8+17x_9+18x_{10}=4$ | 53581 | 100 | 53.6% |
| $5x_1+18x_2+3x_3+8x_4+7x_5+x_6+6x_7-x_8+x_{10}=15$ | 1200 | 10 | 1.2% |
| $-2x_1+18x_2+3x_3+7x_4+6x_5+x_6-2x_7+13x_8+2x_9+15x_{10}=8$ | 0 | 0 | 0% |
| $4x_1+18x_2+13x_3+18x_4+16x_5+12x_6+14x_7+15x_9+x_{10}=4$ | 99996 | 270 | 100% |
| $10x_1+10x_2+x_4+3x_5+17x_6+9x_7+6x_8+6x_9+7x_{10}=6$ | 87905 | 290 | 87.9% |
| $-x_1+5x_2+12x_3+4x_4+3x_5+11x_6+x_7+6x_8+3x_9+6x_{10}=10$ | 0 | 0 | 0% |
| $12x_1-2x_2+11x_3+7x_4+7x_5+8x_6+5x_7+4x_8+14x_9+3x_{10}=0$ | 53581 | 120 | 53.6% |
| $x_1+8x_2+x_3+11x_4+12x_5+6x_6+x_7+17x_8+18x_9+16x_{10}=1$ | 94462 | 440 | 94.5% |
| $2x_1+14x_2-x_3+8x_4+14x_5-x_6+5x_7+14x_8+7x_9+12x_{10}=16$ | 112 | 0 | 0.1% |
| $14x_1-2x_2+18x_3+14x_4+2x_5+15x_6+16x_8+4x_9+14x_{10}=0$ | 99997 | 200 | 100% |
| $-x_1+7x_2+3x_3+6x_4+2x_5+14x_6+2x_7+8x_8-x_9+4x_{10}=3$ | 0 | 0 | 0% |
| $15x_1+17x_2+10x_3+8x_4+9x_5+16x_6+18x_7+4x_8=18$ | 95525 | 210 | 95.5% |
| $4x_1+14x_2+7x_3+8x_4+16x_5+5x_6+12x_7+5x_8-2x_9=7$ | 25975 | 50 | 26.0% |
| $17x_1+10x_2+6x_3+13x_4+6x_6+17x_7+14x_8+18x_9+8x_{10}=4$ | 84021 | 170 | 84.2% |
| $17x_1+11x_2+4x_3+16x_4+7x_5+11x_7+4x_8+4x_9-x_{10}=1$ | 53581 | 120 | 53.6% |
| $3x_1+8x_2+12x_3+14x_4+10x_5+3x_6+15x_7+18x_8-x_9+11x_{10}=0$ | 99998 | 180 | 100% |
| $16x_1+11x_2+14x_3+4x_4+17x_5+10x_6-x_7+6x_8-x_9-x_{10}=11$ | 53582 | 70 | 53.6% |
| $12x_1+17x_2+3x_4+2x_5+9x_6+14x_7+12x_8+10x_9+4x_{10}=18$ | 99997 | 210 | 100% |
| $11x_1+17x_2+12x_3+11x_4+15x_6-2x_7+14x_8+7x_9+14x_{10}=2$ | 99996 | 160 | 100% |
| $12x_1+4x_2+11x_3+2x_5+14x_6+14x_7+5x_8+14x_9+5x_{10}=4$ | 85538 | 150 | 85.5% |
| $14x_1+17x_2+10x_3+x_4+6x_5+12x_6+18x_7+3x_8+3x_9+2x_{10}=14$ | 96349 | 220 | 96.3% |
| $8x_1+12x_2+14x_3+12x_4+9x_5+2x_6+11x_7-2x_8+5x_9+18x_{10}=18$ | 99996 | 190 | 100% |
| $12x_1+10x_2-2x_3+5x_4+13x_5+17x_6+4x_7+3x_8+18x_9+x_{10}=8$ | 53582 | 110 | 53.6% |

Table F18

The half-space range search time (milliseconds) when dimension k = 10, size n = $10^5$

| equation of the hyper plane | #pts found | search time | nodes % |
|---|---|---|---|
| $16x_1+18x_2-2x_3+7x_4+11x_5+2x_6+4x_7+15x_8+12x_9+9x_{10}=9$ | 99997 | 240 | 100% |
| $14x_1+13x_2+10x_3-2x_5+10x_7-x_8+16x_9+12x_{10}=4$ | 99997 | 180 | 100% |
| $2x_1+11x_2+3x_3+15x_4+6x_5+15x_7+13x_8+10x_9+5x_{10}=8$ | 210 | 0 | 0.2% |
| $15x_1+4x_2+6x_4+3x_5+5x_6+16x_7+x_8+9x_9+x_{10}=15$ | 87461 | 290 | 87.5% |
| $8x_1+15x_2+4x_3+2x_4+12x_5+17x_6+11x_8+8x_9+10x_{10}=1$ | 95643 | 200 | 95.6% |
| $11x_1-x_2+5x_3+4x_4+12x_5+14x_7+16x_8+14x_{10}=2$ | 99998 | 190 | 100% |
| $18x_1-x_2+15x_3+14x_4+x_5+10x_6+9x_7+2x_8+12x_9=-1$ | 99998 | 170 | 100% |
| $8x_1+4x_2+11x_3+2x_4+18x_5-x_6+16x_8+16x_9+13x_{10}=1$ | 53582 | 80 | 53.6% |
| $12x_1+7x_2+13x_3+13x_4+3x_5+5x_6+4x_7+4x_8+15x_9=14$ | 70512 | 130 | 70.5% |
| $12x_1+9x_2+9x_3+7x_4+4x_5+16x_6+7x_7+11x_8+15x_9+7x_{10}=0$ | 80316 | 150 | 80.3% |
| $6x_1+17x_4+11x_5+10x_6+12x_7+5x_8+9x_9=8$ | 99996 | 170 | 100% |
| $-2x_1+12x_3+8x_4+16x_5+11x_6+4x_7+16x_8+15x_9+6x_{10}=3$ | 0 | 0 | 0% |
| $15x_1+17x_2-x_3-2x_4+12x_5+11x_6+15x_7+3x_8+2x_9+14x_{10}=10$ | 99997 | 190 | 100% |
| $17x_1+12x_2+12x_4+2x_5+16x_6+12x_7+17x_8+9x_9+6x_{10}=-2$ | 99996 | 190 | 100% |
| $8x_1+4x_2+2x_4+13x_5+17x_6+8x_7-x_8+5x_9+8x_{10}=-1$ | 25998 | 60 | 26.0% |
| $4x_1+8x_2+11x_3+16x_4+3x_5+10x_6+4x_7+16x_8+7x_9+4x_{10}=2$ | 70466 | 270 | 70.5% |
| $6x_1+3x_2+12x_3+9x_4+14x_5+14x_6+8x_8+13x_9+17x_{10}=11$ | 99997 | 170 | 100% |
| $15x_1-x_2+3x_3+13x_4+15x_5+8x_6+13x_7+8x_8+11x_9+16x_{10}=16$ | 53583 | 80 | 53.6% |
| $-2x_1+8x_2+8x_3+9x_4+16x_5+11x_6+11x_7+2x_8+12x_9+4x_{10}=0$ | 46415 | 90 | 46.4% |
| $9x_1+x_2+16x_3+12x_4+x_5+16x_6+10x_7-2x_8+9x_9+17x_{10}=8$ | 99998 | 180 | 100% |
| $9x_1+2x_2+13x_3+15x_4+9x_5+8x_6+5x_7+9x_8+16x_9+8x_{10}=5$ | 99996 | 170 | 100% |
| $17x_1+11x_2+x_3+13x_4+8x_5+18x_6+17x_7+8x_8+14x_9+5x_{10}=15$ | 92503 | 270 | 92.5% |
| $17x_1+x_2+16x_3-x_4+6x_5-x_6-2x_7+4x_8+4x_9+x_{10}=-1$ | 53582 | 100 | 53.6% |
| $4x_1+9x_3+14x_4+14x_5+3x_7-x_8-x_9+15x_{10}=1$ | 53564 | 100 | 53.6% |
| $9x_1+12x_2+9x_3+3x_4+3x_5+12x_7+15x_8+18x_9+10x_{10}=16$ | 99997 | 170 | 100% |
| $12x_1+6x_2+13x_3+2x_4+13x_5+16x_6+18x_7+5x_8+6x_9+9x_{10}=10$ | 53581 | 100 | 53.6% |
| $2x_1+8x_3+11x_4+12x_5-x_7+7x_9+15x_{10}=0$ | 28355 | 50 | 28.4% |
| $14x_1+4x_2+x_3+17x_4+17x_5+7x_6+6x_7+5x_8+18x_9+3x_{10}=-1$ | 84803 | 320 | 84.8% |
| $7x_1+15x_2+5x_3+13x_4+5x_5-x_6+12x_7+14x_8+17x_9+12x_{10}=5$ | 53582 | 100 | 53.6% |
| $5x_1+18x_2+10x_4+14x_5+x_6+18x_7-x_8+16x_{10}=17$ | 53583 | 140 | 53.6% |
| $-2x_1+x_3+8x_6+x_7+7x_8+3x_{10}=8$ | 0 | 0 | 0% |
| $8x_1+18x_2-2x_3+7x_5-x_6+17x_7+8x_8+17x_9+4x_{10}=-1$ | 25998 | 30 | 26.0% |
| $16x_1+9x_2+9x_3+10x_4+13x_5+13x_6+x_7+x_8+11x_9+18x_{10}=6$ | 99998 | 200 | 100% |
| $13x_1+10x_2-2x_3+11x_4+17x_5+4x_7+4x_8+16x_9=12$ | 99997 | 190 | 100% |
| $4x_1+9x_3+2x_5+15x_6+9x_7-2x_8+10x_9+13x_{10}=0$ | 99997 | 130 | 100% |
| $18x_1+9x_2+7x_4+7x_5+7x_6+14x_7+x_8+15x_9+16x_{10}=0$ | 78557 | 140 | 78.6% |
| $2x_1+x_2+7x_3+11x_4+15x_5+12x_6+9x_7+8x_8+6x_9+x_{10}=1$ | 93019 | 380 | 93.0% |
| $3x_1+2x_2+5x_3+14x_4+2x_5+4x_6+3x_7+13x_9+12x_{10}=0$ | 72509 | 140 | 72.5% |
| $-x_1+12x_2+14x_3+5x_4+5x_5-x_6+2x_7+12x_8+10x_9+13x_{10}=7$ | 46415 | 90 | 46.4% |
| $-2x_1+3x_2+3x_3+6x_4+4x_5+10x_6+15x_7-2x_9=3$ | 0 | 0 | 0% |
| $12x_1-2x_2+7x_3+17x_4+8x_5+x_6+8x_7+17x_8+16x_{10}=7$ | 53582 | 110 | 53.6% |
| $2x_1+16x_2+15x_3+15x_4+10x_5+5x_6+11x_7+9x_8+16x_9+7x_{10}=5$ | 73995 | 190 | 74.0% |
| $15x_1+16x_2+3x_3+17x_4+13x_5+12x_6+15x_7+11x_8+9x_{10}=10$ | 99997 | 190 | 100% |
| $2x_1+7x_2+15x_3+5x_4-x_5+6x_6-x_7+5x_8+15x_9+2x_{10}=-2$ | 9 | 0 | 0% |
| $17x_1+x_2-2x_3+16x_4+2x_5+18x_6+13x_7+17x_9+17x_{10}=1$ | 53583 | 90 | 53.6% |
| $3x_1+13x_2+9x_3+5x_4+8x_5+13x_6+18x_7+14x_8+11x_9+10x_{10}=10$ | 99997 | 210 | 100% |
| $17x_1+17x_2+11x_3+16x_4+15x_5-x_6+x_7+2x_8+11x_9-2x_{10}=1$ | 53583 | 110 | 53.6% |

# Appendix G
# Average Search Time for Half-Space Range Search

Table G1

The average half-space range search time (milliseconds) for the k-d Search Skip List

| % of points in range | | 10% or less | 40 to 60% | 90% or more |
|---|---|---|---|---|
| k | # points | Average search time in milliseconds | | |
| 5 | $10^4$ | 0 | 5.4 | 17 |
| | $5*10^4$ | 0.8 | 45 | 90 |
| | $10^5$ | 2 | 110 | 186 |
| 7 | $10^4$ | 0 | 10 | 19 |
| | $5*10^4$ | 1 | 55 | 105.4 |
| | $10^5$ | 2.6 | 130 | 213 |
| 10 | $10^4$ | 3 | 9.7 | 22 |
| | $5*10^4$ | 4.4 | 55.6 | 99.4 |
| | $10^5$ | 9 | 106.6 | 195.5 |

# Appendix H
# Complete Source Code for the Test Drive Routines for the k-d Search Skip List and the Half-Space Range Search

# H1 Makefile for the k-d Search Skip List & Half-Space Range Search

```
testpro: kdtest.o kdskiplist.o
        CC -o $@ kdtest.o kdskiplist.o -lm

testhss: kdtesths.o hskdlist.o
        CC -o $@ kdtesths.o hskdlist.o -lm

kdtest.o: kdskiplist.h kdtest.cc
        CC -g -c kdtest.cc

kdtesths.o: hskdlist.h kdtest.cc
        CC -DHALFSPACE -g -c -o $@ kdtest.cc

kdskiplist.o: kdskiplist.h kdskiplist.cc
        CC -g -c kdskiplist.cc

hskdlist.o: hskdlist.h hskdlist.cc
        CC -g -c hskdlist.cc
```

## H2 The File "kdskiplist.h"

```
//*************************************************************************
// File name:       kdskiplist.h
// Description: Declarations of class templates
//                  KDSSkipNode<T>
//                  KDSSkipList<T>
// Written by:  Yunlan Pan
// Date:       April 17, 1997
//
// Details: KDSSkipList<T>: implements a k-d search skip list
//          that has the following methods:
//            - insert data
//            - delete data
//            - semi-infinity range search
//            - range search
//            - search a node for data
//            - traverse entire list
//
//*************************************************************************

#ifndef KDSSKIPLIST_H
#define KDSSKIPLIST_H

#include <stdlib.h>
#include <iostream.h>

#define k 5                     // Suppose our dimension is 10
#define maxht  64          // for deletion stack
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
// Replace maxkey and minkey by the max and min of T (the template)
#define maxkey 2147483647   // maxkey := (2^31)-1
#define minkey -2147483648  // minkey = -(2^31) - 1

template<class T> class KDSSkipList;                    // predeclaration

//*************************************************************************
//       class KDSSkipNode
//         T: non pointer! e.g, int, float, object of string class, etc.
//         Such class T must have copy constructor, overloaded assignment
//         operator and overloaded operators: ==, >, <, etc.
//*************************************************************************

template<class T>
class KDSSkipNode
{
        friend class KDSSkipList<T>;

//************************ PUBLIC INTERFACE ************************
public:
```

```
    KDSSkipNode();
    KDSSkipNode(const T inKeys[k], const char* inName);
    ~KDSSkipNode();

//************************ PRIVATE INTERFACE ************************
private:
    KDSSkipNode<T>    *right, *down;
    char              *name;
    T          keys[3][k]; // keys[0][]: x_1, x_2, ..., x_k
                           // keys[1][]: 0, minx_2, ..., minx_k
                           // keys[1][]: 0, maxx_2, ..., maxx_k

        void setName(const char* name);
        void setData(const T val[k]);
        void setMin(const T min1[k]);
        void setMin(const T min1[k], const T min2[k]);
        void setMax(const T max1[k]);
        void setMax(const T max1[k], const T max2[k]);
        int isMinLeqInput(const T input[k]) const;
        int isMaxGeqInput(const T input[k]) const;
        int isDataInRange(const T* minR, const T* maxR) const;
        int isDataInRange(const T* minR, const T& high1) const;
        int isDataInRange(const T& low1, const T* maxR) const;
};


//*****************************************************************************
//              class KDSSkipList
// Assumption: different nodes have different first key, i.e. key[0].
//*****************************************************************************
template<class T>
class KDSSkipList
{
//************************ PUBLIC INTERFACE ************************
public:
    KDSSkipList();
    ~KDSSkipList(){clear(); }

    const KDSSkipNode<T>* getHead() const {return head;}
    int getHeight() const;
    int insert(const T val[k], char* name=NULL);
    const KDSSkipNode<T>* search(const T& val, long& numOfComps) const;
    void rangeSearch(const KDSSkipNode<T>& startNode,
                     const T minRange[k],
                           const T maxRange[k],
                           const T& maxFirst,
                           long& numFound) const;
    void rangeSearchUp(const KDSSkipNode<T>& startNode,
                             const T minRange[k],
                             const T& high1,          // max range for x_1
                             const T& maxFirst,
                             long& numFound) const;  // up semi-infinite
```

```
    void rangeSearchDown(const KDSSkipNode<T>& startNode,
                         const T maxRange[k],
                         const T& low1,    // min range for x_1
                         const T& maxFirst,
                         long& numFound) const;   // down semi-infinite
    int deleteNode(const T& v);
    void printList() const;

//************************* PRIVATE INTERFACE ************************
private:
        KDSSkipNode<T>  *head, *bottom, *tail;

        void newMaxMin(KDSSkipNode<T>& inNode); // reset the max and min for
                                                // x_2, ..., x_k
        void printNodeData(const KDSSkipNode<T>& node) const;
        void printNodeMin(const KDSSkipNode<T>& node) const;
        void printNodeMax(const KDSSkipNode<T>& node) const;
        void clear();
};

#endif
```

## H3 The File "kdskiplist.cc"

```
//************************************************************************
// Module:        kdskiplist.cc
// Description:   Implementation of KDSSkipNode and KDSSkipList classes
// Written by:    Yunlan Pan
// Date:  May 17, 1997
//
//************************************************************************

#include <iostream.h>
#include <string.h>

#include "kdskiplist.h"

//********************* Implementation of KDSSkipNode ********************


//************************************************************************
// Description: Default constructor of KDSSkipNode.
//************************************************************************
template<class T>
KDSSkipNode<T>::KDSSkipNode()
        : right(NULL), down(NULL), name(NULL)//, seq(0)
{
        // Do useless initialization
        for (int i=0; i<k; i++)
        {
                keys[0][i]=0;
                keys[1][i]=0;
                keys[2][i]=0;
        }
}


//************************************************************************
// Description: Constructor of KDSSkipNode.
//************************************************************************
template<class T>
KDSSkipNode<T>::KDSSkipNode(const T inKeys[k], const char* inName)
        : right(NULL), down(NULL), name(NULL)//, seq(0)
{

        this->setData(inKeys);
        this->setName(inName);
}


//************************************************************************
// Description: Destructor of KDSSkipNode.
//************************************************************************
```

```
template<class T>
KDSSkipNode<T>::~KDSSkipNode()
{
	if (name != NULL)
		delete[] name;
}


//*************************************************************************
// Description: Set the name of the node.
//*************************************************************************
template<class T>
void KDSSkipNode<T>::setName(const char* inName)
{
	// Clear old name first
	if (this->name)
	{
		delete[] name;
		this->name = NULL;
	}

	// Set the new name
	if (inName != NULL)
	{
		name = new char[strlen(inName)+1];
		strcpy(name, inName);
	}
}


//*************************************************************************
// Description: Set the k key values and the name for the node.
//*************************************************************************
template<class T>
void KDSSkipNode<T>::setData(const T inKeys[k])
{
	for (int i=0; i<k; i++)
	{
		keys[0][i] = inKeys[i];
	}
}


//*************************************************************************
// Description: Set the minimum values of x_2, ..., x_k as the minimum
//		of min1[i] and min2[i].
//*************************************************************************
template<class T>
void KDSSkipNode<T>::setMin(const T min1[k], const T min2[k])
{
	for (int i=1; i<k; i++)
	{
```

```
                keys[1][i] = min(min1[i], min2[i]);
        }
}


//*********************************************************************
// Description: Set the minimum values of x_2, ..., x_k as the input
//*********************************************************************
template<class T>
void KDSSkipNode<T>::setMin(const T min0[k])
{
        for (int i=1; i<k; i++)
        {
                keys[1][i] = min0[i];
        }
}


//*********************************************************************
// Description: Set the maximum values of x_2, ..., x_k as the maximum
//              of max1[i] and max2[i].
//*********************************************************************
template<class T>
void KDSSkipNode<T>::setMax(const T max1[k], const T max2[k])
{
        for (int i=1; i<k; i++)
        {
                keys[2][i] = max(max1[i], max2[i]);
        }
}


//*********************************************************************
// Description: Set the maximum values of x_2, ..., x_k as the input.
//*********************************************************************
template<class T>
void KDSSkipNode<T>::setMax(const T max0[k])
{
        for (int i=1; i<k; i++)
        {
                keys[2][i] = max0[i];
        }
}


//*********************************************************************
// Description: Check whether all the mins are less than or equal to the
//              input. Returns 1 if true, 0 otherwise.
//*********************************************************************
template<class T>
int KDSSkipNode<T>::isMinLeqInput(const T input[k]) const
{
```

```
	// We have only k-1 minimums
	for (int i=1; i<k; i++)
	{
		if (keys[1][i] > input[i])
			return 0;
	}

	return 1;
}


//*************************************************************************
// Description: Check whether all the maxs are greater than or equal to the
//		input. Returns 1 if true, 0 otherwise.
//*************************************************************************
template<class T>
int KDSSkipNode<T>::isMaxGeqInput(const T input[k]) const
{
	// We have only k-1 maximums
	for (int i=1; i<k; i++)
	{
		if (keys[2][i] < input[i])
			return 0;
	}

	return 1;
}


//*************************************************************************
// Description: Check whether all the data are inside the given range.
//		If minR == NULL, then we check whether all the data do
//		not exceed maxR; If maxR == NULL, check all the data
//		not less then minR. True, returns 1, false, returns 0.
//		minR and maxR are arrays with k elements.
//*************************************************************************
template<class T>
int KDSSkipNode<T>::isDataInRange(const T* minR, const T* maxR) const
{
	int i;

	// We have k values to check
	if (minR == NULL)
	{
		for (i=0; i<k; i++)
			if (keys[0][i] > maxR[i])
				return 0;
	}
	else if (maxR == NULL)
	{
		for (i=0; i<k; i++)
			if (keys[0][i] < minR[i])
```

```
                return 0;
        }
        else
        {
                for (i=0; i<k; i++)
                {
                        if ((keys[0][i] < minR[i]) || (keys[0][i] > maxR[i]))
                                return 0;
                }
        }

        return 1;
}


//*************************************************************************
// Method:      isDataInRange()
// Description: Check whether all the data are inside the given range.
//              {[minR[0], high1], [minR[1], infinity), ...
//               [minR[k-1], infinity)}. True, returns 1, false, returns 0.
//*************************************************************************
template<class T>
int KDSSkipNode<T>::isDataInRange(const T* minR, const T& high1) const
{
        // We have k values to check
        if (keys[0][0] > high1)
                return 0;
        else
        {
                for (int i=0; i<k; i++)
                {
                        if (keys[0][i] < minR[i])
                                return 0;
                }
        }

        return 1;
}


//*************************************************************************
// Method:      isDataInRange()
// Description: Check whether all the data are inside the given range.
//              {[low1, maxR[0]], (-infinity, maxR[1]], ...
//               (-infinity, maxR[k-1]]}. True, returns 1, false, returns 0.
//*************************************************************************
template<class T>
int KDSSkipNode<T>::isDataInRange(const T& low1, const T* maxR) const
{
        // We have k values to check
        if (keys[0][0] < low1)
                return 0;
```

```
        else
        {
                for (int i=0; i<k; i++)
                {
                        if (keys[0][i] > maxR[i])
                                return 0;
                }
        }

        return 1;
}


//****************** Implementation of KDSSkipList **********************

//*************************************************************************
// Description: Constructor of KDSSkipList<T>.
//*************************************************************************
template<class T>
KDSSkipList<T>::KDSSkipList()
{
        int i=0;

        head = new KDSSkipNode<T>();
        tail = new KDSSkipNode<T>();
        bottom = new KDSSkipNode<T>();

        head->keys[0][0] = maxkey;
        for (i=1; i<k; i++)
        {
                head->keys[1][i] = maxkey;
                head->keys[2][i] = minkey;
        }
        head->down = bottom;
        head->right = tail;

        bottom->right = bottom;
        bottom->down = bottom;

        tail->keys[0][0] = maxkey;
        tail->right = tail;
//      sequence = 0;
}


//*************************************************************************
// Description: returns the height of this k-d search skip list.
//                      head and bottom are not contributed to height.
//*************************************************************************
template<class T>
int KDSSkipList<T>::getHeight() const
{
```

```
        register int    h;

        KDSSkipNode<T>* x = head->down;
        h = 0;
        while (x!=bottom)
        {
                x = x->down;
                h++;
        }
        return(h);
}


//*********************************************************************
// Description: Returns the node whose keys[0][0] matches the input.
//              If there is no node matching it, bottom node will
//              be returned.
//*********************************************************************
template<class T>
const KDSSkipNode<T>* KDSSkipList<T>::search(
                                const T& val, long& numOfComps) const
{
        KDSSkipNode<T>* tmpNode;
        numOfComps = 0;

        tmpNode = head;
        while  (tmpNode != bottom)
        {
                while (val > tmpNode->keys[0][0])
                {
                tmpNode = tmpNode->right;
                        numOfComps++;
                }

                if  (tmpNode->down == bottom)
                {
                numOfComps++;
                return( (val == tmpNode->keys[0][0]) ?
                                tmpNode : bottom );
                }
                tmpNode = tmpNode->down;
        }

        return bottom;
}


//*********************************************************************
// Description: Insert the data val[k] and name into the
//              k-d search skiplist. If the data is already in
//              the list, it returns 0, otherwise it returns 1,
//              which means that the data was inserted successfully.
```

```
//*********************************************************************
template<class T>
int KDSSkipList<T>::insert(const T val[k], char* name)
{
        KDSSkipNode<T>* tmpNode = head;
        KDSSkipNode<T>* newNode = NULL;

        register int inSub = 0; // 1 if val[0][0] goes in tmpNode's subtree.
        register int add1 = 0;          // 1 if a node was added at this level.
        register int success = 1;

        // Set the bottom just for the comparisom consistency
        bottom->setData(val);
        bottom->setName(name);
        bottom->setMin(val);
        bottom->setMax(val);

        // Do at each level
        while (tmpNode != bottom)
        {
                // find where you drop
                while (*val > tmpNode->keys[0][0])
                tmpNode = tmpNode->right;

                add1 = 0;
                if (tmpNode->keys[0][0] >
                   tmpNode->down->right->right->keys[0][0])
                {       // if 3 elms at level below, or ...
                        // ... at bottom level + must insert...
            // ... ==> raise middle elm
                // Add a node.  Include keeping track of minimum
                //   and maximum y as the node is added.
                newNode = new KDSSkipNode<T>;
                // sequence ++;
                newNode->right = tmpNode->right;
                newNode->down = tmpNode->down->right->right;
                tmpNode->right = newNode;
                inSub = (tmpNode->down->right->keys[0][0] > *val);
                newNode->setData(tmpNode->keys[0]);
                        newNode->setName(tmpNode->name);
                newNode->setMin(newNode->down->keys[1],
                                        newNode->down->right->keys[1]);
                newNode->setMax(newNode->down->keys[2],
                                        newNode->down->right->keys[2]);

                        if (!inSub)
                        {
                                newNode->setMin(newNode->keys[1], val);
                                newNode->setMax(newNode->keys[2], val);
                }

                if (newNode->down == bottom)
```

```
                {       // leaf level
                        newNode->setMin(tmpNode->keys[1]);
                        newNode->setMax(tmpNode->keys[2]);
                        newNode->setName(tmpNode->name);
                }
                tmpNode->setData(tmpNode->down->right->keys[0]);
                        tmpNode->setName(tmpNode->down->right->name);
                tmpNode->setMin(tmpNode->down->right->keys[1],
                                        tmpNode->down->keys[1]);
                tmpNode->setMax(tmpNode->down->right->keys[2],
                                        tmpNode->down->keys[2]);
                if (inSub)
                        {
                                tmpNode->setMin(tmpNode->keys[1], val);
                                tmpNode->setMax(tmpNode->keys[2], val);
                }

                add1 = 1;
                }
                else  if (tmpNode->down == bottom)
                        // if insert_Key already in KDSkipList
                success = 0;

                if(tmpNode->down != bottom && !add1)
                {
                        tmpNode->setMin(tmpNode->keys[1], val);
                        tmpNode->setMax(tmpNode->keys[2], val);
                }
                tmpNode = tmpNode->down;
        } //while

        if (head->right != tail)
        {       // raise height of KDSSkiplist if necessary
        newNode = new KDSSkipNode<T>;
                newNode->down = head;
                newNode->right = tail;
                newNode->keys[0][0] = maxkey;
                //sequence ++;
                //newNode->seq = sequence;
                newNode->setMin(newNode->down->keys[1],
                                newNode->down->right->keys[1]);
                newNode->setMax(newNode->down->keys[2],
                                newNode->down->right->keys[2]);
                head = newNode;
        }

        return(success);
}


//*********************************************************************
// Description: Perform range search on k-d priority search tree
```

```
//                  constructed using a deterministic skip list. i.e., report
//                  all points lying inside the range ([min1:max1],
//                  [min2:max2], ..., [mink:maxk]).
// Argument:
//                  maxFirst: maximum first dimensional value in the subtree
//                              rooted startNode.
//*********************************************************************
template<class T>
void KDSSkipList<T>::rangeSearch(const KDSSkipNode<T>& startNode,
                const T minRange[k],
                const T maxRange[k],
                const T& maxFirst,
                            long& numFound) const
{
        if ((&startNode != bottom) && (&startNode != tail))
        {
                if ((startNode.down != bottom) &&
                   (startNode.isMinLeqInput(maxRange)) &&
                   (startNode.isMaxGeqInput(minRange)))
                {       // Not leaf level
                        // Check the first data to see which branch it goes
                if (*minRange <= startNode.keys[0][0])
                                rangeSearch(*startNode.down, minRange,
                                  maxRange, startNode.keys[0][0], numFound);
                if ((startNode.keys[0][0] < *maxRange) &&
                                (startNode.keys[0][0] < maxFirst))
                                rangeSearch(*startNode.right, minRange,
                                                maxRange, maxFirst, numFound);
                }
                else if (startNode.down == bottom)
                {       // leaf level, report points if in range
                        if (startNode.isDataInRange(minRange, maxRange))
                        {
                                //printNodeData(startNode);
                                numFound++;
                        }

                        // Keep check the right points
                if ( (startNode.keys[0][0] < *maxRange) &&
                                (startNode.keys[0][0] < maxFirst) )
                                rangeSearch(*startNode.right, minRange,
                                                maxRange, maxFirst, numFound);
                }
        }
}


//*********************************************************************
// Method:      rangeSearchUp()
// Description: Perform up semi-infinite range search on k-d priority search
//                  tree constructed using a deterministic skip list. i.e.,
//                  report all points lying inside the range ([min1:max1],
```

```
//                  [min2:infinity), ..., [mink:infinity)).
// Argument:
//                  maxFirst: maximum first dimensional value in the subtree
//                          rooted startNode.
//                  high1: max range for x_1.
//*************************************************************************
template<class T>
void KDSSkipList<T>::rangeSearchUp(const KDSSkipNode<T>& startNode,
                const T minRange[k],
                const T& high1,
                const T& maxFirst,
                        long& numFound) const
{
        if ((&startNode != bottom) && (&startNode != tail))
        {
                if ((startNode.down != bottom) &&
                  (startNode.isMaxGeqInput(minRange)))
                {       // Not leaf level
                        // Check the first data to see which branch it goes
                if (*minRange <= startNode.keys[0][0])
                                rangeSearchUp(*startNode.down, minRange,
                                   high1, startNode.keys[0][0], numFound);
                if ((startNode.keys[0][0] < high1) &&
                                (startNode.keys[0][0] < maxFirst))
                        rangeSearchUp(*startNode.right, minRange,
                                        high1, maxFirst, numFound);
                }
                else if (startNode.down == bottom)
                {       // leaf level, report points if in range
                        if (startNode.isDataInRange(minRange, high1))
                        {
                                //printNodeData(startNode);
                                numFound++;
                        }

                        // Keep check the right points
                if ( (startNode.keys[0][0] < high1) &&
                                (startNode.keys[0][0] < maxFirst) )
                        rangeSearchUp(*startNode.right, minRange,
                                        high1, maxFirst, numFound);
                }
        }
}


//*************************************************************************
// Method:      rangeSearchDown()
// Description: Perform up semi-infinite range search on k-d priority search
//                  tree constructed using a deterministic skip list. i.e.,
//                  report all points lying inside the range ([min1:max1],
//                  (-infinity, max2], ..., (-infinity, maxk]).
// Argument:
```

```
//                  maxFirst: maximum first dimensional value in the subtree
//                          rooted startNode.
//                  high1: min range for x_1.
//*************************************************************************
template<class T>
void KDSSkipList<T>::rangeSearchDown(const KDSSkipNode<T>& startNode,
                const T maxRange[k],
                const T& low1,
                const T& maxFirst,
                        long& numFound) const
{
        if ((&startNode != bottom) && (&startNode != tail))
        {
                if ((startNode.down != bottom) &&
                  (startNode.isMinLeqInput(maxRange)))
                {       // Not leaf level
                        // Check the first data to see which branch it goes
                if (low1 <= startNode.keys[0][0])
                                rangeSearchDown(*startNode.down, maxRange,
                                  low1, startNode.keys[0][0], numFound);
                if ((startNode.keys[0][0] < *maxRange) &&
                                  (startNode.keys[0][0] < maxFirst))
                                rangeSearchDown(*startNode.right, maxRange,
                                                low1, maxFirst, numFound);
                }
                else if (startNode.down == bottom)
                {       // leaf level, report points if in range
                        if (startNode.isDataInRange(low1, maxRange))
                        {
                                //printNodeData(startNode);
                                numFound++;
                        }

                        // Keep check the right points
                if ( (startNode.keys[0][0] < *maxRange) &&
                                  (startNode.keys[0][0] < maxFirst) )
                                rangeSearchDown(*startNode.right, maxRange,
                                                low1, maxFirst, numFound);
                }
        }
}


//*************************************************************************
// Method:        printNodeData()
// Description: Print the node (without the minimum and maximum).
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeData(const KDSSkipNode<T>& node) const
{
        cout << endl;
        cout << "\t Data of the Node: (";
        for (int i=0; i<k-1; i++)
```

```
                cout << node.keys[0][i] << ", ";
        cout << node.keys[0][k-1];

        if (node.name)
                cout << ", " << node.name;

        cout << ");" << endl;
}


//*************************************************************************
// Method:          printNodeMin()
// Description: Print the node's minimum values.
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeMin(const KDSSkipNode<T>& node) const
{
        cout << "\t Min of the Node: (";
        for (int i=1; i<k-1; i++)
                cout << node.keys[1][i] << ", ";
        cout << node.keys[1][k-1] << ");" << endl;
}


//*************************************************************************
// Method:          printNodeMax()
// Description: Print the node's maximum values.
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeMax(const KDSSkipNode<T>& node) const
{
        cout << "\t Max of the Node: (";
        for (int i=1; i<k-1; i++)
                cout << node.keys[2][i] << ", ";
        cout << node.keys[2][k-1] << ");" << endl;
}


//*************************************************************************
// Method:          newMaxMin()
// Description: reset the max and min for  x_2, ..., x_k for inNode.
//                  The method is called in the deletion method.
//*************************************************************************
template<class T>
void KDSSkipList<T>::newMaxMin(KDSSkipNode<T>& inNode)
{
        KDSSkipNode<T>*         tmp;
        int         repeat;
        int                 i=0;                //loop index

        tmp = inNode.down;
        if (tmp != bottom)
        {
```

```
                repeat = 1;
                inNode.setMin(tmp->keys[1]);
                inNode.setMax(tmp->keys[2]);
                while ((tmp->keys[0][0] <= inNode.keys[0][0]) && repeat)
                {
                if (tmp->keys[0][0] == maxkey)
                                repeat = 0;
                        // Update mins and maxs for inNode
                        for (i=1; i<k; i++)
                        {
                        if (tmp->keys[1][i] < inNode.keys[1][i])
                                        inNode.keys[1][i] = tmp->keys[1][i];
                        if (tmp->keys[2][i] > inNode.keys[2][i])
                                        inNode.keys[2][i] = tmp->keys[2][i];
                        }

                tmp = tmp->right;
                }
        }
}


//*************************************************************************
// Method:      deleteNode()
// Description: delete a node matching the input key.
//              If the deletion succeeds, it returns 1. Otherwise if there
//              is no such node in the list, it returns 0.
//              Implemetation notes:
//              If drop in last gap merge/borrow with/from previous gap
//              Second pass (always): -- after you reach bottom level
//                  -- from top of DSL
//*************************************************************************
template<class T>
int KDSSkipList<T>::deleteNode(const T& v)
{
        KDSSkipNode<T>          *x, *px, *nx, *t, *pla[maxht];
                                // px: previous x
                                // nx: next x
                                // pla: stack
        int    success, botlast, npla;
        char*  predname;  // name of predecessor del_key at bottom level
        T      pred[k];   // data of predecessor del_key at bottom level
        T      lastAbove; // righmost key on search path at level above

        x = head->down;         // x = current elm in search path
        success = (x != bottom);
        bottom->keys[0][0] = v;
        lastAbove = head->keys[0][0]; // righmost keys on search path
                                      // at level above
        npla = 0;
        pla[npla] = head;
        while (x != bottom)
        {  // do at every level
```

```
    while  (v > x->keys[0][0])
    {   // find where you drop
    px = x;      // keeping track of the previous gap
    x = x->right;
    }
    nx = x->down;   // mark where to drop at level below
    if  (x->keys[0][0] == nx->right->keys[0][0])
    {
            // if {only 1 elm in gap to drop} or
// {at bottom level + must delete}
    if  (x->keys[0][0] != lastAbove)
            { // if DOESN'T drop in last gap
            t = x->right;
            if ((t->keys[0][0] == t->down->right->keys[0][0])
                            ||
                (nx == bottom) )
                    {
                    // if only 1 elm in next gap or at
                            // bottom level
                            // lower separator of current+next gap
                            x->right = t->right;
                            x->setData(t->keys[0]);
                            x->setMin(t->keys[1]);
                            x->setMax(t->keys[2]);
                            x->setName(t->name);
                            delete t;
            }
            else
                    {  // if >=2 elms in next gap
                            // raise 1st elm in next gap
                            x->setData(t->down->keys[0]);
                            // lower separator of current+next gap
                            t->down = t->down->right;
                            newMaxMin(*t);
            }
    }
    else    // if DOES drop in last gap
    if (px->keys[0][0] <= px->down->right->keys[0][0])
            {       // if only 1 elm in previous gap
                    botlast = 0;
                    if  (nx == bottom)
                    { // if del_Key is in elm of height>1
                    botlast = 1;
                    // predecessor of del_key at bottom level
                            for(int j=0; j<k; j++)
                            pred[j] = px->keys[0][j];
                    predname = px->name;
            }
                    // lower separator of previous+current gap
                    px->right = x->right;
                    px->keys[0][0] = x->keys[0][0];
                    if (!botlast)
                    {
```

```
                                px->setData(x->keys[0]);
                                px->setMin(x->keys[1]);
                                px->setMax(x->keys[2]);
                        px->setName(x->name);
                        }
                        delete x;
                        x = px;
        }
        else
                {       // if >=2 elms in previous gap
                        // t = last elm in previous gap
                        KDSSkipNode<T>* tmp = px->down->right->right;
                        t = (px->keys[0][0] == tmp->keys[0][0] ?
                                        px->down->right : tmp);
                        // raise last elm in previous gap & lower
                        //separator of previous+current gap
                        px->setData(t->keys[0]);
                        x->down = t->right;
                        newMaxMin(*px);
        }
        } // end of if
        else  if (nx == bottom)  // if del_Key not in KDSSL
        success = 0;

        lastAbove = x->keys[0][0];
        npla++;
        pla[npla] = x;
        x = nx;
} // end of while

// Do a 2nd pass, for the case that del_key was in elm of height>1
x = head->down;
while  (x != bottom)
{
        while  (v > x->keys[0][0])
        x = x->right;

        if  (v == x->keys[0][0])
        {
                x->setData(pred);
        x->setName(predname);
        }
        x = x->down;
}

// Check and reset mins and maxs for all nodes
for (int i=npla; i>=0; i--)
        newMaxMin(*pla[i]);    //on path to deleted key.

if  (head->down->right == tail)
{  // lower header of DSL, if necessary
        x = head;
        head = x->down;
```

```
                delete x;
        }

        return(success);
}


//*************************************************************************
// Method:      printList()
// Description: Print the whole skiplist (head is printed, bottom and tail
//              are not printed.
//*************************************************************************
template<class T>
void KDSSkipList<T>::printList() const
{
        KDSSkipNode<T> *xx, *x;

        cout << " The List Contains the Following Points:\n" << endl;

        xx = head;
        while  (xx!= bottom)
        {
                x = xx;
                while  (x->keys[0][0] != maxkey)
                {
                printNodeData(*x);
                printNodeMin(*x);
                printNodeMax(*x);
                x = x->right;
                }
                cout << endl;
                cout << "MaxKey:" << x->keys[0][0] << endl;
        printNodeMin(*x);
        printNodeMax(*x);

                xx = xx->down;
        }
}

//*************************************************************************
// Method:      clear()
// Description: Clear the whole list. All the memory allocated will be
//              released. It is used in the destructor.
//*************************************************************************
template<class T>
void KDSSkipList<T>::clear()
{
        KDSSkipNode<T> *x, *tmp, *down;

        x = head;
        while  (x != bottom)
        {
                down = x->down; // hold down since xx will be deleted
```

```
                // Clean the level where x lies
                while (x->keys[0][0] != maxkey)
                {
                        tmp = x;
                x = x->right;
                        delete tmp;
                }
                delete x;  // delet the node with maxKey

                // Starts the next level
                x = down;
        }

        delete bottom;
        delete tail;
}
```

# H4 The File "hskdlist.h"

```
//*********************************************************************
// File name:       hskdlist.h
// Description:     Declarations of class templates
//                  KDSSkipNode<T>
//                  KDSSkipList<T>
// Written by:  Yunlan Pan
//Date:             June 29, 1997, Modified the node such that it
//                  contains the minimum and maximum values for the
//                  first dimension too. Such modification will be used
//                  by the added half-space range search.
//
// Details:         KDSSkipList<T>: implements a k-d search skip list
//                  that has the following methods:
//                  - insert data
//                  - delete data
//                  - semi-infinity range search
//                  - range search
//                          - half-space range search (Added June 29, 97)
//                  - search a node for data
//                  - traverse entire list
//
//*********************************************************************

#ifndef KDSSKIPLIST_H
#define KDSSKIPLIST_H

#include <stdlib.h>
#include <iostream.h>


#define k 7                     // The given dimension
#define maxht  64          // for deletion stack
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
// Replace maxkey and minkey by the max and min of T (the template)
#define maxkey 2147483647   // maxkey := (2^31)-1
#define minkey -2147483648  // minkey = -(2^31) - 1


template<class T> class KDSSkipList;                    // predeclaration

//*********************************************************************
//              class KDSSkipNode
//          T: non pointer! e.g, int, float, object of string class, etc.
//          Such class T must have copy constructor, overloaded assignment
//          operator and overloaded operators: ==, >, <, etc.
//
//          Midification on June 29, 97: For half-space range search, only
```

```
//          the number types (e.g., int, long, float, etc.) are valid.
//*************************************************************************

template<class T>
class KDSSkipNode
{
        friend class KDSSkipList<T>;

//************************* PUBLIC INTERFACE ************************
public:
    KDSSkipNode();
        KDSSkipNode(const T inKeys[k], const char* inName);
    ~KDSSkipNode();

//************************* PRIVATE INTERFACE ************************
private:
    KDSSkipNode<T>    *right, *down;
        char              *name;
    T           keys[3][k]; // keys[0][]: x_1, x_2, ..., x_k
                                    // keys[1][]: minx_1, ..., minx_k
                                    // keys[1][]: maxx_1, ..., maxx_k

        void setName(const char* name);
        void setData(const T val[k]);
        void setMin(const T min1[k]);
        void setMin(const T min1[k], const T min2[k]);
        void setMax(const T max1[k]);
        void setMax(const T max1[k], const T max2[k]);
        int isDataInHalfSp(const T* halfSp) const;
        void getLimits(const T* halfSp, double low[k], double up[k]) const;
        void checkPosition(const T* halfSp, const double* low,
                const double* up, int& in, int& prune) const;
};


//*************************************************************************
//              class KDSSkipList
// Assumption: different nodes have different first key, i.e. key[0].
//*************************************************************************
template<class T>
class KDSSkipList
{
//************************* PUBLIC INTERFACE ************************
public:
    KDSSkipList();
    ~KDSSkipList() {clear(); }

        const KDSSkipNode<T>* getHead() const {return head;}
        int getHeight() const;
        int insert(const T val[k], char* name=NULL);
        void halfSpSearch(const KDSSkipNode<T>& startNode,
                const T halfSp[k+1],
                const T& maxFirst,
```

```
            long& numFound) const;
        int deleteNode(const T& v);
        void printList() const;

//************************* PRIVATE INTERFACE ************************
private:
        KDSSkipNode<T>  *head, *bottom, *tail;

        void newMaxMin(KDSSkipNode<T>& inNode); // reset the max and min for
                                                // x_2, ..., x_k
        void printNodeData(const KDSSkipNode<T>& node) const;
        void printNodeMin(const KDSSkipNode<T>& node) const;
        void printNodeMax(const KDSSkipNode<T>& node) const;
        void repSubList(const KDSSkipNode<T>& bNode, long& numReport) const;
        void clear();
};

#endif
```

# H5 The File "hskdlist.cc"

```
//*********************************************************************
// Module:        hskdlist.cc
// Description:   Implementation of KDSSkipNode and KDSSkipList classes
// Written by:    Yunlan Pan
// Date:  June 29, 1997, Modified the node such that it
//        contains the minimum and maximum values for the
//        first dimension too. Such modification will be used
//        by the added half-space range search.
//
//*********************************************************************


#include <iostream.h>
#include <string.h>

#include "hskdlist.h"

//********************* Implementation of KDSSkipNode ********************


//*********************************************************************
// Description: Default constructor of KDSSkipNode.
//*********************************************************************
template<class T>
KDSSkipNode<T>::KDSSkipNode()
        : right(NULL), down(NULL), name(NULL)//, seq(0)
{
        // Do useless initialization
        for (int i=0; i<k; i++)
        {
                keys[0][i]=0;
                keys[1][i]=0;
                keys[2][i]=0;
        }
}


//*********************************************************************
// Description: Constructor of KDSSkipNode.
//*********************************************************************
template<class T>
KDSSkipNode<T>::KDSSkipNode(const T inKeys[k], const char* inName)
        : right(NULL), down(NULL), name(NULL)//, seq(0)
{

        this->setData(inKeys);
        this->setMin(inKeys);
        this->setMax(inKeys);
        this->setName(inName);
```

```
}

//*************************************************************************
// Description: Destructor of KDSSkipNode.
//*************************************************************************
template<class T>
KDSSkipNode<T>::~KDSSkipNode()
{
        if (name != NULL)
                delete[] name;
}


//*************************************************************************
// Description: Set the name of the node.
//*************************************************************************
template<class T>
void KDSSkipNode<T>::setName(const char* inName)
{
        // Clear old name first
        if (this->name)
        {
                delete[] name;
                this->name = NULL;
        }

        // Set the new name
        if (inName != NULL)
        {
                name = new char[strlen(inName)+1];
                strcpy(name, inName);
        }
}


//*************************************************************************
// Description: Set the k key values and the name for the node.
//*************************************************************************
template<class T>
void KDSSkipNode<T>::setData(const T inKeys[k])
{
        for (int i=0; i<k; i++)
        {
                keys[0][i] = inKeys[i];
        }
}


//*************************************************************************
// Description: Set the minimum values of x_1, ..., x_k as the minimum
//              of min1[i] and min2[i].
//*************************************************************************
```

```
template<class T>
void KDSSkipNode<T>::setMin(const T min1[k], const T min2[k])
{
	for (int i=0; i<k; i++)
	{
		keys[1][i] = min(min1[i], min2[i]);
	}
}


//************************************************************************
// Description: Set the minimum values of x_1, ..., x_k as the input
//************************************************************************
template<class T>
void KDSSkipNode<T>::setMin(const T min0[k])
{
	for (int i=0; i<k; i++)
	{
		keys[1][i] = min0[i];
	}
}


//************************************************************************
// Description: Set the maximum values of x_1, ..., x_k as the maximum
//		of max1[i] and max2[i].
//************************************************************************
template<class T>
void KDSSkipNode<T>::setMax(const T max1[k], const T max2[k])
{
	for (int i=0; i<k; i++)
	{
		keys[2][i] = max(max1[i], max2[i]);
	}
}


//************************************************************************
// Description: Set the maximum values of x_1, ..., x_k as the input.
//************************************************************************
template<class T>
void KDSSkipNode<T>::setMax(const T max0[k])
{
	for (int i=0; i<k; i++)
	{
		keys[2][i] = max0[i];
	}
}


//************************************************************************
// Method:	isDataInHalfSp()
// Description: Used only by half-space range search.
```

```
//                  Check whether this node is in the positive half space
//                  of the input hyperplane. Return 1 if it is, 0 otherwise.
//                  For the node previous tail in the leaf level, it returns 0.
//*************************************************************************
template<class T>
int KDSSkipNode<T>::isDataInHalfSp(const T* halfSp) const
{
        if (keys[0][0] == maxkey)
        {
                // No data in such node, like head
                return 0;
        }

        T lside = 0;
        for (int i=0; i<k; i++)
        {
                lside += halfSp[i]*keys[0][i];
        }

        if (lside > halfSp[k])
                return 1;

        return 0;
}


//*************************************************************************
// Method:      getLimits()
// Description: This method is used by half-space range search.
//              Get the lower and upper limits to help determine whether
//              we should report a subtree or prune a subtree.
// Arguments:
//      halfSp: input, k+1 array of coefficients of the hyperplane.
//      low:    output, x'_i in the thesis.
//      up:     output, x"_i in the thesis.
//*************************************************************************
template<class T>
void KDSSkipNode<T>::getLimits(const T* halfSp,
                               double low[k], double up[k]) const
{
        T mins[k], maxs[k];
        int i;  // Loop index

        for (i=0; i<k; i++)
        {
                if (halfSp[i] > 0)
                {
                        mins[i] = -halfSp[i]*keys[2][i];
                        maxs[i] = -halfSp[i]*keys[1][i];
                }
                else if (halfSp[i] < 0)
                {
                        mins[i] = -halfSp[i]*keys[1][i];
```

```
                        maxs[i] = -halfSp[i]*keys[2][i];
                }
                else
                        mins[i] = maxs[i] = 0;
        }

        // Get the low and up
        double hmins = mins[0];
        double hmaxs = maxs[0];
        for (i=1; i<k; i++)
        {
                hmins += mins[i];
                hmaxs += maxs[i];
        }
        hmins += halfSp[k];
        hmaxs += halfSp[k];

        for (i=0; i<k; i++)
        {
                if (halfSp[i] != 0)
                {
                        low[i] = (hmins - mins[i])/halfSp[i];
                        up[i] = (hmaxs - maxs[i])/halfSp[i];
                }
                else
                {
                        low[i] = (double)minkey;
                        up[i] = (double)maxkey;
                }
        }
}


//*********************************************************************
// Method:      checkPosition()
// Description: This method is used by half-space range search.
//              Check whether the subtree starts at this node completely
//              resides in the positive side of the hyperplane or should
//              be pruned or should be checked more.
// Arguments:
//      halfSp: input, k+1 array of coefficients of the hyperplane.
//      low:    input, x'_i in the thesis.
//      up:     input, x"_i in the thesis.
//      in:     output, 1 means the subtree resides in the half-space.
//      prune:  output, 0 means the subtree should not be pruned.
//*********************************************************************
template<class T>
void KDSSkipNode<T>::checkPosition(const T* halfSp, const double* low,
                        const double* up, int& in, int& prune) const
{
        // Initialize the outputs
        in = prune = 0;
```

```
        for (int i=0; i<k; i++)
        {
                if (((halfSp[i] > 0) && (keys[2][i] < low[i])) ||
                   ((halfSp[i] < 0) && (keys[1][i] > up[i])))
                {
                        prune = 1; // The subtree should be pruned.
                        return;
                }
                else if (((halfSp[i] > 0) && (keys[1][i] > up[i])) ||
                   ((halfSp[i] < 0) && (keys[2][i] < low[i])))
                {
                        in = 1; // The subtree resides in the halp-space.
                        return;
                }
        }
}


//****************** Implementation of KDSSkipList **********************

//*************************************************************************
// Description: Constructor of KDSSkipList<T>.
//*************************************************************************
template<class T>
KDSSkipList<T>::KDSSkipList()
{
        int i=0;

        head = new KDSSkipNode<T>();
        tail = new KDSSkipNode<T>();
        bottom = new KDSSkipNode<T>();

        head->keys[0][0] = maxkey;
        for (i=0; i<k; i++)
        {
                head->keys[1][i] = maxkey;
                head->keys[2][i] = minkey;
        }
        head->down = bottom;
        head->right = tail;

        bottom->right = bottom;
        bottom->down = bottom;

        tail->keys[0][0] = maxkey;
        tail->right = tail;
//  sequence = 0;
}


//*************************************************************************
// Description: returns the height of this k-d search skip list.
```

```
//                  head and bottom are not contributed to height.
//*************************************************************************
template<class T>
int KDSSkipList<T>::getHeight() const
{
        register int    h;

        KDSSkipNode<T>* x = head->down;
        h = 0;
        while (x!=bottom)
        {
                x = x->down;
                h++;
        }
        return(h);
}


//*************************************************************************
// Description: Insert the data val[k] and name into the
//                  k-d search skiplist. If the data is already in
//                  the list, it returns 0, otherwise it returns 1,
//                  which means that the data was inserted successfully.
//*************************************************************************
template<class T>
int KDSSkipList<T>::insert(const T val[k], char* name)
{
        KDSSkipNode<T>* tmpNode = head;
        KDSSkipNode<T>* newNode = NULL;

        register int inSub = 0; // 1 if val[0][0] goes in tmpNode's subtree.
        register int add1 = 0;          // 1 if a node was added at this level.
        register int success = 1;

        // Set the bottom just for the comparisom consistency
        bottom->setData(val);
        bottom->setName(name);
        bottom->setMin(val);
        bottom->setMax(val);

        // Do at each level
        while (tmpNode != bottom)
        {
                // find where you drop
                while (*val > tmpNode->keys[0][0])
                tmpNode = tmpNode->right;

                add1 = 0;
                if (tmpNode->keys[0][0] >
                   tmpNode->down->right->right->keys[0][0])
                {       // if 3 elms at level below, or ...
                        // ... at bottom level + must insert...
```

```
// ... ==> raise middle elm
    // Add a node.  Include keeping track of minimum
    //   and maximum y as the node is added.
    newNode = new KDSSkipNode<T>;
    // sequence ++;
    newNode->right = tmpNode->right;
    newNode->down = tmpNode->down->right->right;
    tmpNode->right = newNode;
    inSub = (tmpNode->down->right->keys[0][0] > *val);
    newNode->setData(tmpNode->keys[0]);
            newNode->setName(tmpNode->name);
    newNode->setMin(newNode->down->keys[1],
                                newNode->down->right->keys[1]);
    newNode->setMax(newNode->down->keys[2],
                                newNode->down->right->keys[2]);
            //newNode->seq = sequence;

            if (!inSub)
            {
                    newNode->setMin(newNode->keys[1], val);
                    newNode->setMax(newNode->keys[2], val);
    }

    if (newNode->down == bottom)
            {       // leaf level
                    newNode->setMin(tmpNode->keys[1]);
                    newNode->setMax(tmpNode->keys[2]);
                    newNode->setName(tmpNode->name);
    }
    tmpNode->setData(tmpNode->down->right->keys[0]);
            tmpNode->setName(tmpNode->down->right->name);
    tmpNode->setMin(tmpNode->down->right->keys[1],
                                tmpNode->down->keys[1]);
    tmpNode->setMax(tmpNode->down->right->keys[2],
                                tmpNode->down->keys[2]);
    if (inSub)
            {
                    tmpNode->setMin(tmpNode->keys[1], val);
                    tmpNode->setMax(tmpNode->keys[2], val);
    }

    add1 = 1;
    }
    else  if (tmpNode->down == bottom)
            // if insert_Key already in KDSkipList
    success = 0;

    if(tmpNode->down != bottom && !add1)
    {
            tmpNode->setMin(tmpNode->keys[1], val);
            tmpNode->setMax(tmpNode->keys[2], val);
    }
```

```
            tmpNode = tmpNode->down;
        } //while

        if (head->right != tail)
        {       // raise height of KDSSkiplist if necessary
        newNode = new KDSSkipNode<T>;
            newNode->down = head;
            newNode->right = tail;
            newNode->keys[0][0] = maxkey;
            //sequence ++;
            //newNode->seq = sequence;
            newNode->setMin(newNode->down->keys[1],
                        newNode->down->right->keys[1]);
            newNode->setMax(newNode->down->keys[2],
                        newNode->down->right->keys[2]);
            head = newNode;
        }

        return(success);
}


//*********************************************************************
// Method:      halfSpSearch()
// Description: Perform half-space (positive side) search on k-d search
//              tree constructed using a deterministic skip list. i.e.,
//              report all points lying inside the positive side of a
//              hyperspace.
// Argument:
//              maxFirst: maximum first dimensional value in the subtree
//                      rooted startNode.
//              halfSp: coefficients of halfplan a_1x_1 + ... + a_kx_k
//              = m where a_1 != 0.
//*********************************************************************
template<class T>
void KDSSkipList<T>::halfSpSearch(const KDSSkipNode<T>& startNode,
            const T halfSp[k+1],
            const T& maxFirst,
                    long& numFound) const
{
        if ( (startNode.keys[0][0] == maxkey) &&
           (startNode.down == bottom) )
        {
            // Last node in the leaf level (the node previous tail)
            return;
        }
        else if ( (startNode.keys[0][0] == maxkey) &&
           (startNode.down != bottom) && (&startNode != tail) )
        {
            // A previous node of tail, just go down
            halfSpSearch(*startNode.down, halfSp,
                    startNode.keys[0][0], numFound);
```

```
    }
    else if ((&startNode != bottom) && (&startNode != tail))
    {
        // Get the bounds
        double low[k], up[k];
        startNode.getLimits(halfSp, low, up);

        if (startNode.down == bottom)
        {   // Leaf level, report points if in half space.
            if (startNode.isDataInHalfSp(halfSp))
            {
                //printNodeData(startNode);
                numFound++;
            }

            // Keep check the right points
            if (startNode.keys[0][0] < maxFirst)
            {
            if ((halfSp[0] > 0) ||
           (startNode.keys[0][0] < up[0]))
                        halfSpSearch(*startNode.right,
                            halfSp, maxFirst, numFound);
            }
        }
        else //startNode.down != bottom) i.e. not leaf level
        {
            // Check to report or prune down.
            int in=0, prune=0;
            startNode.checkPosition(halfSp, low, up, in, prune);

            if (in) //Report the subtree
                this->repSubList(startNode, numFound);
            else if (!prune) // Check downword
                halfSpSearch(*startNode.down, halfSp,
                    startNode.keys[0][0], numFound);

            // Check to branch the right
        if ( (startNode.keys[0][0] < maxFirst) &&
                ( (halfSp[0] > 0) ||
         (startNode.keys[0][0] < up[0])
        )
                )
                halfSpSearch(*startNode.right,
                        halfSp, maxFirst, numFound);
        }
    }
}


//*************************************************************************
// Method:      printNodeData()
// Description: Print the node (without the minimum and maximum).
```

```
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeData(const KDSSkipNode<T>& node) const
{
        cout << endl;
        cout << "\t Data of the Node: (";
        for (int i=0; i<k-1; i++)
                cout << node.keys[0][i] << ", ";
        cout << node.keys[0][k-1];

        if (node.name)
                cout << ", " << node.name;

        cout << ");" << endl;
}


//*************************************************************************
// Method:        printNodeMin()
// Description: Print the node's minimum values.
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeMin(const KDSSkipNode<T>& node) const
{
        cout << "\t Min of the Node: (";
        for (int i=0; i<k-1; i++)
                cout << node.keys[1][i] << ", ";
        cout << node.keys[1][k-1] << ");" << endl;
}


//*************************************************************************
// Method:        printNodeMax()
// Description: Print the node's maximum values.
//*************************************************************************
template<class T>
void KDSSkipList<T>::printNodeMax(const KDSSkipNode<T>& node) const
{
        cout << "\t Max of the Node: (";
        for (int i=0; i<k-1; i++)
                cout << node.keys[2][i] << ", ";
        cout << node.keys[2][k-1] << ");" << endl;
}


//*************************************************************************
// Method:        newMaxMin()
// Description: reset the max and min for  x_1, ..., x_k for inNode.
//                The method is called in the deletion method.
//*************************************************************************
template<class T>
void KDSSkipList<T>::newMaxMin(KDSSkipNode<T>& inNode)
```

```
{
        KDSSkipNode<T>*         tmp;
        int         repeat;
        int                 i=0;                //loop index

        tmp = inNode.down;
        if (tmp != bottom)
        {
                repeat = 1;
                inNode.setMin(tmp->keys[1]);
                inNode.setMax(tmp->keys[2]);
                while ((tmp->keys[0][0] <= inNode.keys[0][0]) && repeat)
                {
                if (tmp->keys[0][0] == maxkey)
                                repeat = 0;
                        // Update mins and maxs for inNode
                        for (i=0; i<k; i++)
                        {
                        if (tmp->keys[1][i] < inNode.keys[1][i])
                                        inNode.keys[1][i] = tmp->keys[1][i];
                        if (tmp->keys[2][i] > inNode.keys[2][i])
                                        inNode.keys[2][i] = tmp->keys[2][i];
                        }

                tmp = tmp->right;
                }
        }
}


//************************************************************************
// Method:          deleteNode()
// Description: delete a node matching the input key.
//                  If the deletion succeeds, it returns 1. Otherwise if there
//                  is no such node in the list, it returns 0.
//                  Implemetation notes:
//                  If drop in last gap merge/borrow with/from previous gap
//                  Second pass (always): -- after you reach bottom level
//                      -- from top of DSL
//************************************************************************
template<class T>
int KDSSkipList<T>::deleteNode(const T& v)
{
        KDSSkipNode<T>              *x, *px, *nx, *t, *pla[maxht];
                                // px: previous x
                                // nx: next x
                                // pla: stack
        int    success, botlast, npla;
        char*  predname;  // name of predecessor del_key at bottom level
        T          pred[k];   // data of predecessor del_key at bottom level
        T          lastAbove; // righmost key on search path at level above
```

```
x = head->down;        // x = current elm in search path
success = (x != bottom);
bottom->keys[0][0] = v;
lastAbove = head->keys[0][0]; // rightmost keys on search path
                              // at level above
npla = 0;
pla[npla] = head;
while (x != bottom)
{  // do at every level
        while  (v > x->keys[0][0])
        {  // find where you drop
        px = x;      // keeping track of the previous gap
        x = x->right;
        }
        nx = x->down;   // mark where to drop at level below
        if  (x->keys[0][0] == nx->right->keys[0][0])
        {
                // if {only 1 elm in gap to drop} or
    // {at bottom level + must delete}
        if  (x->keys[0][0] != lastAbove)
                { // if DOESN'T drop in last gap
                t = x->right;
                if ((t->keys[0][0] == t->down->right->keys[0][0])
                                ||
                        (nx == bottom) )
                        {
                        // if only 1 elm in next gap or at
                                // bottom level
                                // lower separator of current+next gap
                                x->right = t->right;
                                x->setData(t->keys[0]);
                                x->setMin(t->keys[1]);
                                x->setMax(t->keys[2]);
                                x->setName(t->name);
                                delete t;
                }
                else
                        {  // if >=2 elms in next gap
                                // raise 1st elm in next gap
                                x->setData(t->down->keys[0]);
                                // lower separator of current+next gap
                                t->down = t->down->right;
                                newMaxMin(*t);
                }
        }
        else   // if DOES drop in last gap
        if (px->keys[0][0] <= px->down->right->keys[0][0])
                {       // if only 1 elm in previous gap
                        botlast = 0;
                        if  (nx == bottom)
                        { // if del_Key is in elm of height>1
                        botlast = 1;
```

```
                        // predecessor of del_key at bottom level
                                for(int j=0; j<k; j++)
                                pred[j] = px->keys[0][j];
                        predname = px->name;
                }
                        // lower separator of previous+current gap
                        px->right = x->right;
                        px->keys[0][0] = x->keys[0][0];
                        if (!botlast)
                        {
                                px->setData(x->keys[0]);
                                px->setMin(x->keys[1]);
                                px->setMax(x->keys[2]);
                        px->setName(x->name);
                        }
                        delete x;
                        x = px;
        }
        else
                {    // if >=2 elms in previous gap
                        // t = last elm in previous gap
                        KDSSkipNode<T>* tmp = px->down->right->right;
                        t = (px->keys[0][0] == tmp->keys[0][0] ?
                                        px->down->right : tmp);
                        // raise last elm in previous gap & lower
                        // separator of previous+current gap
                        px->setData(t->keys[0]);
                        x->down = t->right;
                        newMaxMin(*px);
        }
        } // end of if
        else  if (nx == bottom)  // if del_Key not in KDSSL
        success = 0;

        lastAbove = x->keys[0][0];
        npla++;
        pla[npla] = x;
        x = nx;
} // end of while

// Do a 2nd pass, for the case that del_key was in elm of height>1
x = head->down;
while  (x != bottom)
{
        while  (v > x->keys[0][0])
        x = x->right;

        if  (v == x->keys[0][0])
        {
                x->setData(pred);
        x->setName(predname);
        }
```

```
                x = x->down;
        }

        // Check and reset mins and maxs for all nodes
        for (int i=npla; i>=0; i--)
                newMaxMin(*pla[i]);    //on path to deleted key.

        if  (head->down->right == tail)
        {   // lower header of DSL, if necessary
                x = head;
                head = x->down;
                delete x;
        }

        return(success);
}


//************************************************************************
// Method:          printList()
// Description: Print the whole skiplist (head is printed, bottom and tail
//                  are not printed.
//************************************************************************
template<class T>
void KDSSkipList<T>::printList() const
{
        KDSSkipNode<T> *xx, *x;

        cout << " The List Contains the Following Points:\n" << endl;

        xx = head;
        while  (xx!= bottom)
        {
                x = xx;
                while  (x->keys[0][0] != maxkey)
                {
                printNodeData(*x);
                printNodeMin(*x);
                printNodeMax(*x);
                x = x->right;
                }
                cout << endl;
                cout << "MaxKey:" << x->keys[0][0] << endl;
        printNodeMin(*x);
        printNodeMax(*x);

                xx = xx->down;
        }
}


//************************************************************************
```

```
// Method:          repSubList()
// Description: Report all the leave nodes contained in the sublist starts
//                  with bNode.
// Argument:
//                  numReport: number of nodes reported, the function does not
//                  do initialization for it, it just add the number of nodes
//                  reported here to its previous value.
//
//*********************************************************************
template<class T>
void KDSSkipList<T>::repSubList(const KDSSkipNode<T>& bNode,
                                long& numReport) const
{
        const KDSSkipNode<T>* temp = &bNode;

        // Get to the leaf level
        while  (temp->down != bottom)
                temp = temp->down;

        // Report all the leaf nodes in the sublist.
        while ( (temp->keys[0][0] < maxkey) &&
                (temp->keys[0][0] <= bNode.keys[0][0]) )
        {
                //printNodeData(*temp);
                //printNodeMin(*temp);
                //printNodeMax(*temp);
                numReport++;
                temp = temp->right;
        }

}


//*********************************************************************
// Method:          clear()
// Description: Clear the whole list. All the memory allocated will be
//                  released. It is used in the destructor.
//*********************************************************************
template<class T>
void KDSSkipList<T>::clear()
{
        KDSSkipNode<T> *x, *tmp, *down;

        x = head;
        while  (x != bottom)
        {
                down = x->down; // hold down since xx will be deleted

                // Clean the level where x lies
                while (x->keys[0][0] != maxkey)
                {
                        tmp = x;
```

```
                x = x->right;
                        delete tmp;
                }
                delete x;  // delet the node with maxKey

                // Starts the next level
                x = down;
        }

        delete bottom;
        delete tail;
}
```

# H6 The File "kdtest.cc"

```
//*************************************************************************
// Module:     kdtest.cc
// Description: Implementation of k-d search skip list
// Written by:  Yunlan Pan
// Date:       April 17, 1997
// Log:    June 29, 1997, Added half-space range search.
//
//*************************************************************************

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <sys/time.h>    // library routines

#ifdef HALFSPACE
#include "hskdlist.h"
#else
#include "kdskiplist.h"
#endif

#define SAMPLESIZE     10000L
#define QWAREARATIO            0.8
#define MILLISEC_PER_SEC    1000

void getQueryWin(long*, long*, long*, long*, long keys[SAMPLESIZE][k], float*);

main()
{
        long      startT, endT;
        long    i, keysA[SAMPLESIZE][k];
        int       j;
        long*     dataIter = NULL;  // pointer iterator of the array keysA
        //char*   names[SAMPLESIZE];

        const long maxTest = 536870912L; //2^29

        // Randomly get the sample data.
        for  (i=0L; i<SAMPLESIZE; i++)
        {
                for (j=0; j<k; j++)
                        keysA[i][j] = lrand48() - maxTest/2 + 0.5;

        }

        // Create an empty skiplist
        KDSSkipList<long>         kdlist;
```

```
cout << "\n ==========================================" << endl;
cout << "k = " << k << endl;


//========================= PART 1 INSERT ====================

long unsucNum = 0L; // Number of points failed to insert

startT = clock();
startT = clock()/MILLISEC_PER_SEC;

// We use pointer to accelerate the access of data to the array
dataIter = keysA[0];

for  (i=0L; i<SAMPLESIZE; i++)
{
        // Only insert if x not already there.
        if (0 == kdlist.insert(dataIter))
        {
                unsucNum++;
                //cout << "key: " <<  keysA[i][0] << " is already in";
                //cout << " the kdskiplist." << endl;
        }

        dataIter += k;
}
cout << "\n Total " << unsucNum;
cout << " nodes are already in the list" << endl;

endT = clock();
endT = clock()/MILLISEC_PER_SEC;

//      kdlist.printList();


cout << "Insert times in milliseconds (total " ;
cout << SAMPLESIZE << " nodes):" << endl;
cout << "\t Total time used for insertion is " << endT-startT << endl;


/*
//================= PART 2 rectangular range Search ================

long    qwMinR[k], qwMaxR[k];
long    wMinR[k], wMaxR[k];
float   qwRatio[k];
long    numFound = 0L;

// Initialize the query window
for (j=0; j<k; j++)
{
        if (QWAREARATIO == 0)
```

```
                qwRatio[j] = 0;
        else
                qwRatio[j] = (float) pow(QWAREARATIO, 1.0/k);
}
getQueryWin(qwMinR, qwMaxR, wMinR, wMaxR, keysA, qwRatio);

cout << "\n ----------Range Search----------\n" << endl;
cout << "\n The following nodes are in the given range:\n" << endl;

startT = clock();
startT = clock()/MILLISEC_PER_SEC;

numFound = 0L;
kdlist.rangeSearch(*kdlist.getHead(),
                qwMinR, qwMaxR, maxkey, numFound);

endT = clock();
endT = clock()/MILLISEC_PER_SEC;
cout << "Range search times in milliseconds (total " ;
cout << numFound << " nodes are found):" << endl;
cout << "\t Total time used for this search is " << endT-startT << endl;

//================ PART 3 Range Search Up ================

cout << "\n ----------rangeSearchUp----------\n" << endl;
// Calculate the query window area ratio
float area = ((float)(wMaxR[0]-qwMinR[0]))/(wMaxR[0]-wMinR[0]);
for (j=1; j<k; j++)
{
        area *= ((float)(wMaxR[j]-qwMinR[j]))/(wMaxR[j]-wMinR[j]);
}
cout << "\n The semi-infinite window area ratio is " << area << endl;
cout << " The following nodes are in the given range:\n" << endl;

startT = clock();
startT = clock()/MILLISEC_PER_SEC;

numFound = 0L;
kdlist.rangeSearchUp(*kdlist.getHead(), qwMinR,
                        qwMaxR[0], maxkey, numFound);

endT = clock();
endT = clock()/MILLISEC_PER_SEC;

cout << "Semi-infinite range search (up) times in milliseconds (total " ;

cout << "\n---------- DELETE ----------\n\n" << endl;
cout << "DELETING the whole list \n" << endl;
startT = clock();
startT = clock()/MILLISEC_PER_SEC;

long numFailDel = 0L; // The times of deletion failure.
```

```
                                // It should be equal to the times of
                                // insertion failure (caused by the
                                // duplicate keys

        // We use pointer to accelerate the access of data to the array
        dataIter = &keysA[0][0];

        for  (i=0L; i<SAMPLESIZE; i++)
        {
                if (kdlist.deleteNode(*dataIter) == 0)
                {
                        //cout << keysA[i][0] << " is not in the list!" << endl;
                        numFailDel++;
                }

                // Next key
                dataIter += k;
        }
        endT = clock();
        endT = clock()/MILLISEC_PER_SEC;

        cout << "milliseconds (total " ;
        cout << numFailDel << " nodes are not found):" << endl;
        cout << "\t Total times (in milliseconds) used for the ";
        cout << "Destruction is " << endT-startT << endl;
*/

#ifdef HALFSPACE

        //================== PART 5 HALF-SPACE SEARCH ====================

        cout << "\n----------  HALF-SPACE SEARCH  ----------\n\n" << endl;

        // Get the hyperplane
    long hyplane[k+1];
    struct timeval      first;
    struct timezone     second;
    gettimeofday(&first, &second);
    srand48(first.tv_sec);
    for (i=0; i<=k; i++)
        hyplane[i] = (long)((lrand48() - maxTest/2 + 0.5)/100000000);
    // Make sure hyplane[0] != 0
    while (hyplane[0] == 0)
        hyplane[0] = (long)((lrand48() - maxTest/2 + 0.5)/100000000);

        // Get the hyperplane
        long hyplane[k+1];

        for (i=0; i<=k; i++)
        {
                hyplane[i] = lrand48() - maxTest/2 + 0.5;
        }
```

```
        // Make sure hyplane[0] != 0
        while (hyplane[0] == 0)
                hyplane[0] = lrand48() - maxTest/2 + 0.5;

        // Print the hyperplane
        cout << "The hyperplane is: \n" << endl;
        for (i=0; i<=k; i++)
                cout << "\t" << hyplane[i] << " " << endl;
        cout << endl << endl;

        startT = clock();
        startT = clock()/MILLISEC_PER_SEC;

        long numInHsp = 0L; // The number of node in the half-space.

        kdlist.halfSpSearch(*kdlist.getHead(), hyplane, maxkey, numInHsp);

        endT = clock();
        endT = clock()/MILLISEC_PER_SEC;

        cout << "There are (total " ;
        cout << numInHsp << " nodes are in the half-space):" << endl;
        cout << "\t Total times (in milliseconds) used for the " << endl;
        cout << "Half-Space range search is " << endT-startT << endl;
#endif

}


//=============================================================
//Function:       getQueryWin
//Description:    Randomly get a query window which covers a given ratio
//                of the area of all the points reside.
//Argument;       [qwMinR : qwMaxR] is the query window;
//                [wMinR : wMaxR] is the whole data window;
//=============================================================
void getQueryWin(long qwMinR[k], long qwMaxR[k],
                 long wMinR[k], long wMaxR[k],
                 long keys[SAMPLESIZE][k], float qwRatio[k])
// qwRatio[i]: the ratio of the i-th side of the query window and
//                 its corresponding side of the whole window.
{
        unsigned i=0, j=0;          // loop index
        long tempMin, tempMax, temp;

        // Get the whole window
        for (i=0; i<k; i++)
        {
                tempMin = tempMax = keys[0][i];

                for (j=1; j<SAMPLESIZE; j++)
                {
```

```
                    temp = keys[j][i];
                    if (tempMin > temp)
                        tempMin = temp;
                    if (tempMax < temp)
                        tempMax = temp;
                }

                wMinR[i] = tempMin;
                wMaxR[i] = tempMax;
        }

        // Get the portions randomly
        long wSideLen;   // length of side of the whole window
        long qwSideLen;  // length of side of the query window
        for (i=0; i<k; i++)
        {
                // Get the length of each side in integer
                wSideLen = wMaxR[i]-wMinR[i];
                qwSideLen = wSideLen * qwRatio[i];
                if (wSideLen != qwSideLen)
                {
                        qwMinR[i] = (lrand48() % (wSideLen-qwSideLen)) + wMinR[i];
                        qwMaxR[i] = qwMinR[i] + qwSideLen;
                }
                else
                {
                        qwMinR[i] = wMinR[i];
                        qwMaxR[i] = wMaxR[i];
                }
        }
}
```

# VITA

Candidate's Full Name:

Yunlan Pan

Place and Data of Birth:

Zhejiang, P. R. China, June 3, 1997

Permanent Address:

N/A

Universities Attended:

Bachelor of Science, September 1985 - July 1989,

Dept. Of Math., Zhejiang Normal Univ., Jihua, P. R. C.

Master of Science, September 1994 - August 1995,

Dept. Of Math. and Stat., University of New Brunswick, Fredericton, Canada

Master of Computer Science, September 1996 - August 1997,

Faculty of CS, University of New Brunswick, Fredericton, Canada