Skip List Data Structures for Multidimensional Data

University of Maryland Institute for Advanced Computer Studies (UMIACS)
Technical Report‡ UMIACS-TR-94-52, CS-TR-3262

Bradford G. Nickerson†
University of New Brunswick
Faculty of Computer Science
P.O. Box 4400
Fredericton, N.B. E3B 5A3
Canada
E-mail: bgn@unb.ca

April 1994

Abstract

This report presents four new data structures for multidimensional data. All of these data structures are based on the deterministic skip list. Explanations are provided for the 2-d search skip list and three different versions of the *k*-d skip list. These structures support fast insertion and deletion. The third version of the *k*-d skip list and the 2-d search skip list require only O(*n*) space. The 2-d search skip list allows semi-infinite range searches of type ($[L_1:H_1],[L_2:\infty]$), or of type ($[L_1:H_1],[-\infty:H_2]$) in time O(t + log *n*). The third version of the *k*-d skip list seems well-suited for range search using parallel processing. Algorithms for building, insertion, deletion and range search for all four data structures are given, along with proofs of worst case complexity for these operations. Complete C code for range search, insertion and deletion in the 2-d search skip list is also presented.

Key words: dynamic data structures, range search, multidimensional data, skip lists, searching.

ACM classification codes: E.1, F.2.2

1 Introduction

k-d range search has long been a problem of interest to computer science. Efficient access to large volumes of multi-attribute data is a fundamental requirement of most large-scale computer information systems. This is precisely where the *k*-d range search problem is applicable. Range search is commonly provided by relational algebra operations on index structures based on the B-tree [Comer, 1979].

‡ This report is also available via anonymous ftp site "`ftp.cs.umd.edu`" file "`/pub/papers/TRs/3262.ps`".

† This work was supported in part by a research study grant from the University of New Brunswick, and was performed at UMIACS while the author was on sabbatical leave.

Here, I consider the possibility of using the skip list (as introduced by Pugh [1990]) to provide the basic structure allowing a more direct and more efficient range search capability. Papadakis [1993] has shown the close relationship of the "deterministic" skip list and multiway search trees like the B-tree. The basic questions considered here are

1. Can the skip list be used for multi-dimensional data? If so, how?

Assuming that skip lists can be used, then the subsequent questions are

2. What kind of space and time performances are achieved for range search when a skip list is used for multi-dimensional data?
3. What other kinds of operations can this data structure support?

2 Range search

As defined in Knuth [1973], a *range query* asks for records (in a file F containing n records) whose attributes fall within a specific range of values (e.g. height > 6'2" or \$23,000 ≤ annual income ≤ \$65,000). We will call these limits L for low and H for high. Boolean combinations of range queries over different attributes are called *orthogonal range queries*. When the conjunction of range queries is required, each separate attribute can be viewed as one dimension of a k-dimensional space, and the orthogonal range query corresponds to asking for all records falling inside a k-dimensional box. A *range search* is performed to retrieve the records specified in the query.

Three functions normally portray the cost for range search on a specific data structure G that supports range search on F (e.g. Bentley and Friedman [1979]). 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.

In addition, the time required to insert or delete a point into or from G is considered here. The cost of these dynamic operations are represented as

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

Many approaches have been devised for efficiently handling range searches. Bentley [1975] devised a multidimensional binary search tree known as the k-d tree. Assuming a balanced k-d tree, it is known that the k-d tree has

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

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

The point quadtree can also be used for multidimensional range search. Samet [1990] contains a good overview of both point quadtree and k-d tree algorithms. Under assumptions of relatively evenly spaced data, the point quadtree has $Q(n, k) = O(kn^{1-1/k})$.

The range tree [Bentley, 1979; Lueker, 1978; Willard, 1985; Willard and Lueker, 1985] consists of a base binary search tree with nodes discriminating on one of the k dimensions. Each node contains another range tree allowing search in a k-1 dimensional space. This recursively defined structure has been shown to have

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

Notice that Q for the range tree is substantially faster than for the k-d tree at the expense of requiring more space.

Edelsbrunner [1981] introduced the RT-tree (also called the range priority tree by Samet [1990]), which makes use of the range tree and the priority search tree of McCreight [1985] to arrive at the following costs:

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

which improves by one factor of log n on the range search time.

3 Skip lists

Pugh [1990] introduced the probabilistic skip list as an alternative to balanced trees. Probabilistic skip lists support range search in one dimension, and it is shown that the expected worst case search time for a simple query (i.e. search for only one record in the file) is $O(\log n)$, and the same time is required for insertion and deletion. Papadakis et al [1992] have shown exact values for these expected costs. Algorithm 1, shown below, can be used for range search in one dimension.

```
1-d_Range_Search(S,L,H,A)
{ Perform a range search on skip list S to find all keys between L and H, placing the
  resultant records in set A. }
  A := empty set;
  determine node b with smallest key K in S ≥ L;
  while K ≤ H do
      begin
          add record at node b to A;
          b := next node in S;
          K := key of node b;
      end
```

Algorithm 1. Range search for a *1*-d skip list.

Due to the nature of a skip list having a complete set of pointers at level 1, the expected running time of this algorithm equals the time needed to find the low range key L in the skip list, plus the time spent in the while loop to follow the level 1 pointers until the high range key H is exceeded. For the probabilistic skip list, then, we have expected worst case times of

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

The probabilistic skip list worst case time is still $O(n)$ for search, insert and delete, although there is a very low probability (for large n) that the worst case will occur. Munro et al [1992], have introduced the deterministic skip list. This deterministic skip list (DSL) is similar in structure to the probabilistic version and uses no notions of probability at all. Munro et al [1992] show that it has a worst case search cost of $\Theta(\log n)$, along with similar costs for insertion and deletion. Their analysis also shows that DSL space requirements are $O(n)$. Using Algorithm 1 above for range search, with S a deterministic skip list, gives worst cases of

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

4 The *k*-d skip list, version 1

It is assumed that we have a set F of n points in k-space that we can preprocess into a data structure. In addition, it is assumed that each point has some other information associated with it that can be ordered. Thus, each element f of F is composed of $(k+1)$-tuples denoted as $(K_1, ..., K_k, J)$, where J is the information associated with this point, and K_i is the key for this point in dimension i. We will build a data structure G to support range search operations on F. Range search is defined as answering a query on G to retrieve all elements of F which simultaneously fall within the specified ranges. Let L and H be sets of size k specifying the ranges. The range query is then defined as determining the set A, where

$$A = \{f \mid f \in F \text{ and } L_i \leq K_i \leq H_i \ \forall\, i,\ 1 \leq i \leq k\}$$

The k-d skip list decomposes the keys for each point into sets of keys, one set for each dimension. Thus k sets of keys, each of size n, are used for the k-d skip list. Each dimension has one *1*-d (deterministic) skip list associated with it. Consider the *2*-d example shown in Table 1 and Figure 1. Figure 2 shows the *2*-d list data structure to represent this data.

Table 1. Example *2*-d point set.

J	K_1	K_2
New York	-73	42
London	0	52
Naples	14	41
Vancouver	-123	48
Tokyo	140	37
Sydney	150	-34
Durban	31	-30
Lisbon	-9	37
Singapore	104	2
Marseilles	5	44
Rotterdam	5	52

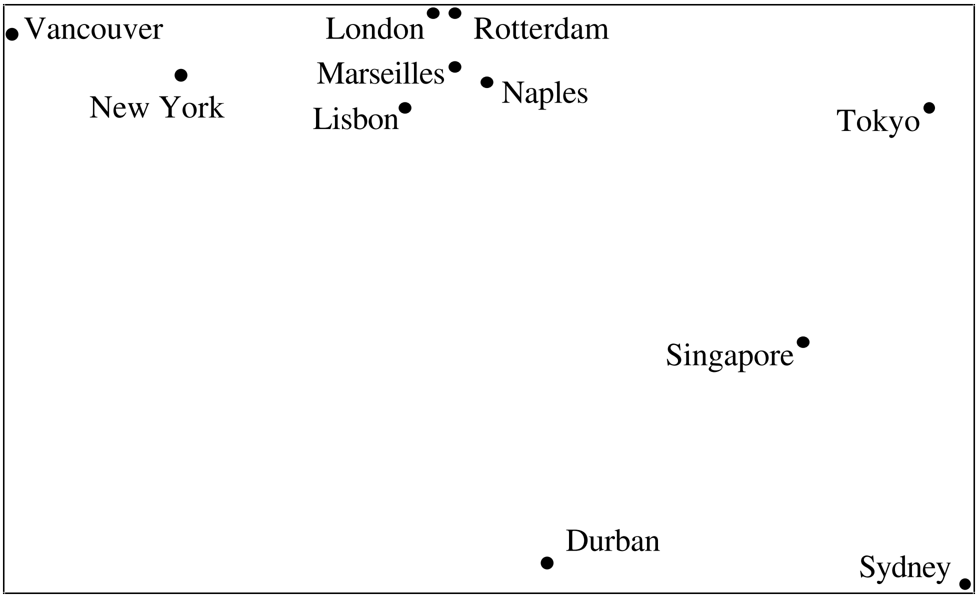

Figure 1. An example *2*-d point set, with keys interpreted as (x, y) coordinates on the Cartesian plane.

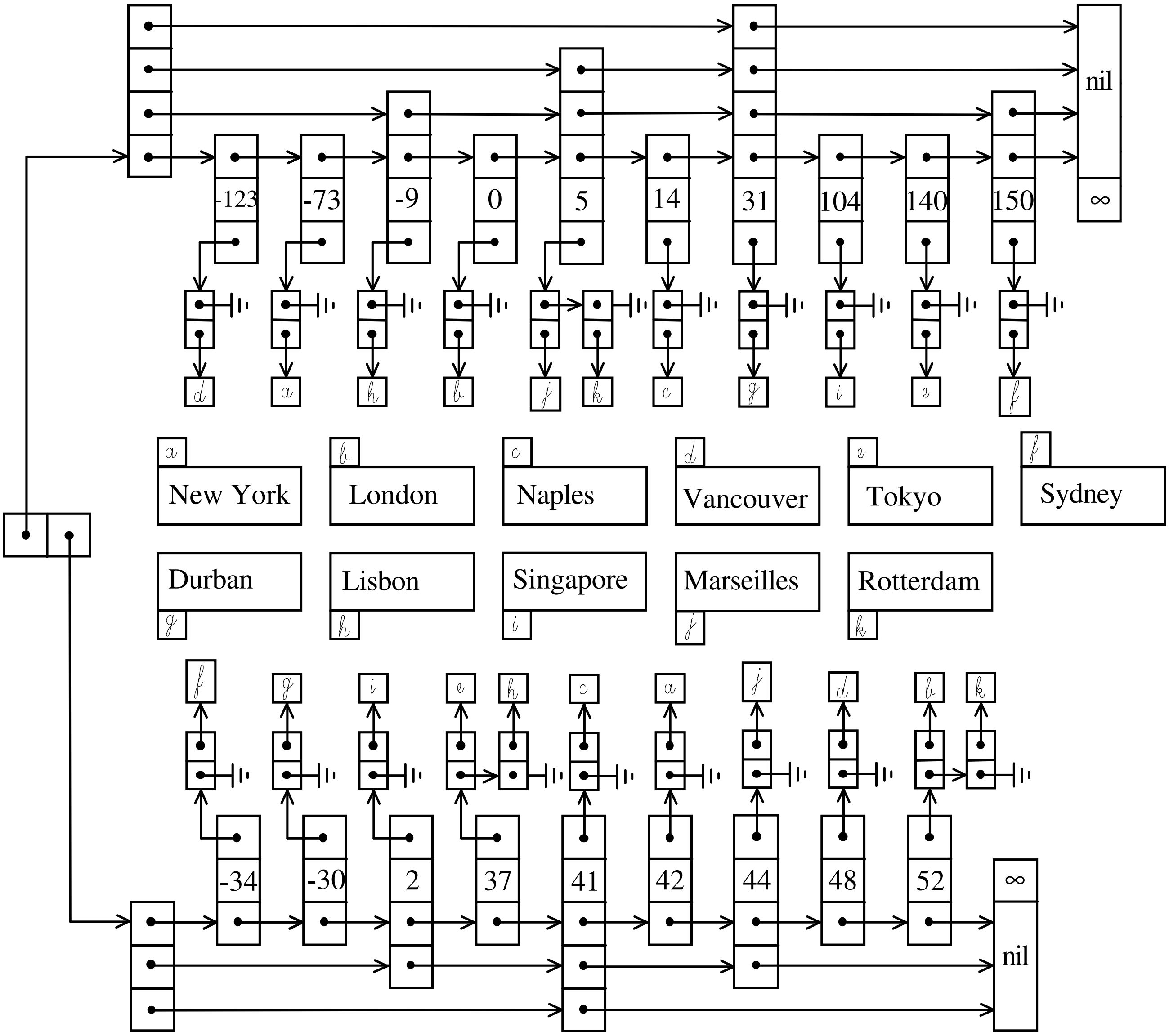

Figure 2. Example 2-d skip list, version 1, data structure.

Notice that there are 2 skip lists indexing the two dimensions. Each node of the skip list points to one (or more) points in the dataset through a pointer (here labelled a through k). Points with the same key have more than one pointer stored at the skip list node pointing to the points with the same key (e.g. key 5 for the K_1 skip list, keys 37 and 52 for the K_2 skip list).

Decomposition of the data F such that each dimension is kept in a sorted sequence (in this case as a skip list) is called the projection method [Bentley and Friedman, 1979]. As noted in Friedman et al [1975], the average time required for range search in nearest neighbour search problem is

$$Q(n, k) = O(n^{1-1/k})$$

where the search returns only a small number of records. In addition, insertion and deletion are O(n) operations. The power of this projection method for range search is improved when combined with the skip list. This is explored further below.

4.1 Building, version 1

To build the k-d skip list, assume we start with a simple list of points in any order. The construction proceeds as shown in Algorithm 2.

```
Make_k-d_skip_v1(F,G)
Make k empty skip lists S_1, ..., S_k;
Make structure G point to these;
for i := 1 to n do
  begin
      Allocate J_i to storage and get its location ℓ;
      for d := 1 to k do
          Insert K_d for point i in S_d, storing pointer to ℓ;
  end
```

Algorithm 2. Construction of a k-d skip list, version 1.

Theorem 1: The time P(n, k) to construct a k-d skip list, version 1, is O($k\ n \log n$), and it requires O($k\ n$) space.

Proof

From Munro et al [1992] we know that inserting a node in a skip list requires $\Theta(\log n)$ time. For each point in set F, this insertion is performed k times, giving the required result for P. The space required is $\Theta(n)$ for the points, plus the storage for the k skip lists. Using the array implementation of the 1-2 deterministic skip list, Munro et al [1992] show it never requires more than 2.282n pointers. With the storage required for the keys, and the extra pointer to storage locations, the storage is O($n + k\ n$) = O($k\ n$). ■

4.2 *k*-d range search, version 1

To perform a range search, we start with sets L and H of size k defining the lower and upper bounds of the search in each dimension. We then use the following algorithm.

```
k-d_Range_Search_v1(G,L,H,A)
1   A := empty set;
2   for i := 1 to k do
3       begin
4           initialize Bi to be an empty set;
5           determine location bi of the smallest key in Si ≥ Li;
6       end
7   done := false;
8   repeat
9       for i := 1 to k do
10          begin
11              add point(s) at location bi to Bi in order of J;
12              bi := next location in Si;
13              if Ki > Hi then done := true;
14          end
15  until done;
16  for i := 1 to k do
17      A := Intersection(A, Bi);
```

Algorithm 3. Range search on the *k*-d skip list, version 1.

Theorem 2: The time $Q(n, k)$ required to perform a range query on a *k*-d skip list, version 1, is $O(k \log n + (n + kt)(1 + \log (n/k + t))) = O(n \log (n/k + t))$, where t is the number of points in the range.

Proof

The proof analyses the three separate parts of Algorithm 3, i.e. (a) for loop at line 2, (b) repeat loop at line 9, and (c) the for loop at line 16, and then sums the three parts together.

(a) Most of the work in the first for loop arises from step 5, which searches for a key in the skip list for dimension i nearest to the lower bound L_i of range i. From Munro et al [1992], it is known that this search requires $\Theta(\log n)$ time. Since this loop is performed k times, the time for the for loop at line 2 is $O(k \log n)$.

(b) The repeat loop at step 8 steps through each dimension one key at a time (see for loop at step 9), comparing keys with the upper bound of the range until one key of one dimension goes out of range. This immediately halts the search. This means that the search stops as soon as the range with the minimum number of keys is found. Call this minimum number s. s is at most n, if all points are in range (giving $t = n$), and can be as small as 0 if no points are in one range (giving $t = 0$). What is the largest value of s when $t < s$? The worst case for 2-d, when $t = 0$, is illustrated in Figure 3.

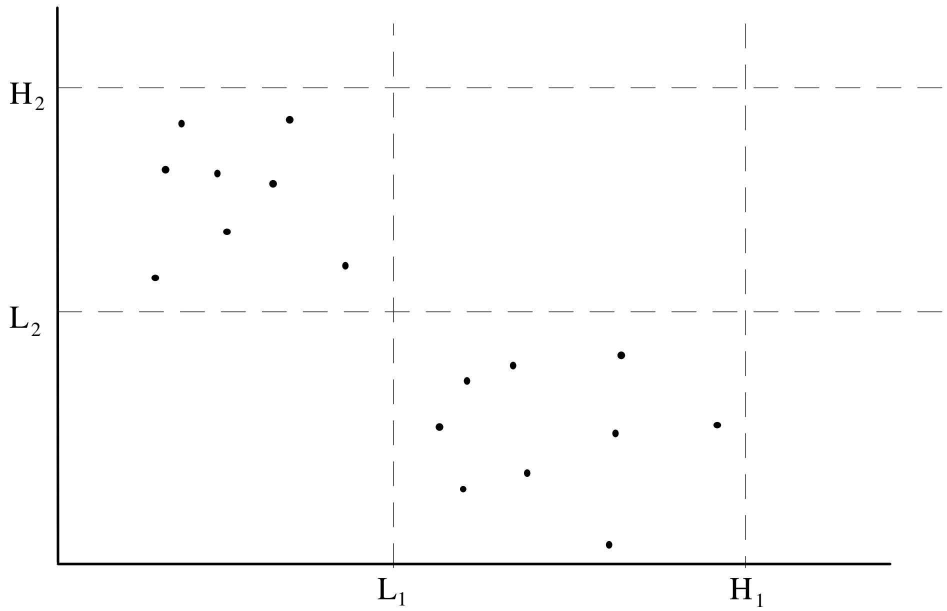

Figure 3. Illustration of worst 2-d case for k-d_Range_Search, version 1.

This illustrates that the largest difference s - t is $n/2$, for the 2-d case. Any value of t > 0 is also counted in s, giving the inequality $0 \leq (s - t) \leq n/2$. The general case reduces to $0 \leq (s - t) \leq n/k$, which can be rewritten as $t \leq s \leq n/k + t$. If more than one point per key exists in the range with the minimum number of keys, then fewer keys are compared.

Statement 11 requires an insertion into the set B_i. Using a balanced binary search tree (or a skip list) for set B_i requires O(log s) time for insertion, assuming the set structure already has s elements in it. The worst case is for $s = n/k + t$, resulting in the worst case for insertion as O(log $(n/k + t)$). Insertions are done for each dimension up to $s = n/k + t$ times, in the worst case, giving the resultant time for executing the for loop at step 9 as $O(k(n/k + t)\log(n/k + t)) = O((n + kt)\log(n/k + t))$.

(c) The third main part of the algorithm starts at step 16. This loop performs a series of intersection operations on an initially empty set A with each of the sets B_i containing the points found within range in each dimension. Each set B_i has s elements, and they are non-disjoint sets. This intersection operation can be performed in O(s) time since the sets are in sorted order. Loop 16 requires k of these unions, giving O(k s) time required for the loop 16 operations. Substituting the worst case of $s = n/k + t$ gives the worst case time required for loop 16 as $O(n + kt)$.

Adding the three parts up gives the result of $O(k \log n) + O((n + kt)\log(n/k + t)) + O(n + kt) = O(k \log n + (n + kt)(1 + \log(n/k + t))) = O(n \log(n/k + t))$. ■

Discussion

Note that the worst case presented above is unlikely to occur with relatively evenly distributed data. Data is rarely partitioned so that each n/k partition is independent of all other partitions of the data. In addition, if there is one dimension where very few points fall into the range, then the algorithm is very fast as it immediately halts when the range with minimum points is found. For s (and thus t) small relative to n, the k-d range search will be very fast. With t small relative to n, and for relatively well distributed data, the expected range search time would approach $O(k \log n + kt(1 + \log t)) = O(k(\log n + t \log t))$.

4.3 Insertion and deletion, version 1

Algorithm 4 shows how a new point can be inserted into a (version 1) k-d skip list. It follows in a straight-forward manner from the construction algorithm.

```
Insert_k-d_v1(G,f)
Allocate J for point f to storage and get its location ℓ;
for d := 1 to k do
        Insert K_d for point f in S_d, storing pointer to ℓ;
```

Algorithm 4. Insertion of a new point into a k-d skip list, version 1.

Theorem 3: The time $I(n, k)$ required to insert a point into a k-d skip list, version 1, is $O(k \log n)$.

Proof

From Munro et al [1992], we know that inserting a node in a skip list requires $\Theta(\log n)$ time. For a point f, this insertion is performed k times, giving the required result for I.■

Algorithm 5 shows the required sequence of steps for deletion of a point from a k-d skip list, version 1.

```
Delete_k-d_v1(G,f)
for d := 1 to k do
        begin
            Find f in S_d by performing a search for K_d;
            Delete K_d for point f in S_d;
        end
```

Algorithm 5. Deletion of a point from a k-d skip list, version 1.

Theorem 4: The time $D(n, k)$ required to delete a point from a k-d skip list, version 1, is $O(k\,n)$.

Proof

The worst case here arises when all points have the same key value in one (or more) of the dimensions. In this case, one (or more) of the skip lists S_i will have only one node, with pointers to the n storage locations in a linked list attached to this node. Finding point f in this list is an $O(n)$ operation, and the time for performing k of them gives the time for deletion as $O(k\,n)$. ■

Theorem 5: Assuming that all keys are unique within each dimension, the time $D(n, k)$ required to delete a point from a k-d skip list, version 1, is $O(k \log n)$.

Proof

With unique keys, the time required to find key K_d in skip list S_d (i.e. step 3 of the algorithm) is O(log *n*). Step 4 requires the deletion of K_d for point f, which also requires O(log *n*) time. This is done *k* times, once for each skip list, giving the required result. ■

5 The *k*-d skip list, version 2

This second version of the *k*-d skip list again decomposes the keys for each point into sets of keys, one set for each dimension. Thus *k* skip lists, each of size *n*, are used for the *k*-d skip list, version 2. Each dimension has a structure similar to a *1*-d (deterministic) skip list associated with it. Consider the *2*-d example shown in Table 2 and Figure 4. Figure 5 shows the *2*-d list data structure representing this data.

Table 2. Example *2*-d point set, version 2.

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

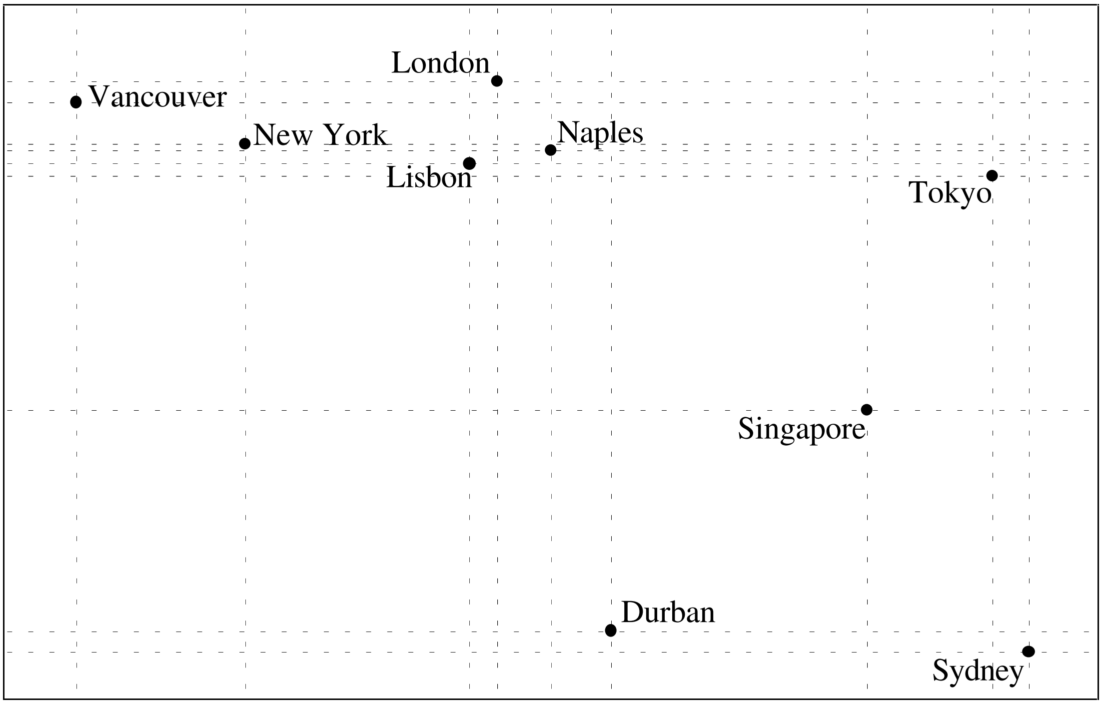

Figure 4. Second example *2*-d point set, with keys interpreted as (x, y) coordinates on the Cartesian plane.

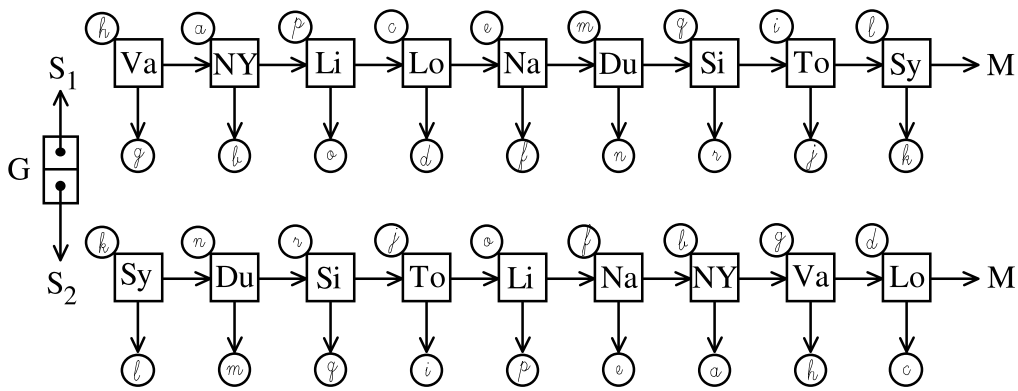

Figure 5. Example *2*-d skip list data structure, version 2.

There are two skip lists indexing the two dimensions. Skip list S_1 is ordered by keys of dimension 1 (plotted as x in Figure 4), and skip list S_2 is ordered by keys of dimension 2 (plotted as y in Figure 4). Figure 5 shows only the nodes; it does not show the *level pointers* at higher levels in the skip list. The script letters (*a* through *r*) inside the circles at the upper left of each node represent the *location* of these nodes in memory.

The *2*-d skip list, version 2, is built such that all points to the right and above any other point can be found by following the two outgoing pointers from each node. Following the right pointer takes you to the next point in this dimension; following the down pointer takes you to this point in the other dimension. This construction leads to the following properties of the *k*-d skip list, version 2:

1. Each point in F has *k* nodes, one in each of the *k* skip lists.
2. Each skip list is ordered by the keys for one dimension.
3. Each node in a skip list has *k dimension pointers*, one for its own dimension, and the rest pointing to the node for this point in the *k*-1 other dimensions.

5.1 Building, version 2

To build the k-d skip list, version 2, assume we start with a simple list of points in any order. The construction proceeds as shown in Algorithm 6.

```
Make_k-d_skip_v2(F,G)
1  Make k empty skip lists S_1, ..., S_k;
2  Make structure G point to these;
3  for i := 1 to n do
4    begin
5       for d := 1 to k do
6          begin
7             Allocate a node for J_i and K_d, and store its location in loc[d];
8             Insert node for point i in S_d ordered by K_d;
9          end
10      for d1 := 1 to k do      {Store pointers to all other dimensions}
11         for j := 1 to k-1 do
12            begin
13               d2 := mod(d1 + j - 1, k) + 1;        {Assumes base 1 index for vector loc}
14               At node location loc[d1], pointer to dimension d2 := loc[d2];
15            end
16   end
```

Algorithm 6. Construction of a *k*-d skip list, version 2.

Theorem 6: The time $P(n, k)$ to construct a *k*-d skip list, version 2, is $O(k\, n(k + \log n))$, and it requires $O(k^2\, n)$ space.

Proof

The for loop at line 5 requires the *k* insertions of a node for point i, one for each of the *k* skip lists. From Munro et al [1992] we know that inserting a node in a skip list requires $\Theta(\log n)$ time. The for loop at line 5 thus requires $O(k \log n)$ time for each point. The for loops at lines 10 and 11 store the pointers to the other dimensions. This double loop requires $k(k\text{-}1)$ operations for each point, or $O(k^2)$. Adding these contributions together gives the worst case time to construct a *k*-d skip list as $O(n(k^2 + k \log n))$, or $O(k\, n(k + \log n))$.

Each node requires *k* dimension pointers, and each point has *k* nodes, resulting in a space requirement of nk^2 dimension pointers. Using the array implementation of the deterministic skip list, Munro et al [1992] show it never requires more than $2.282n$ level pointers (in one dimension). Adding these contributions together gives the total space requirement as $O(nk^2 + 2.282nk)$, or $O(k^2 n)$. ■

5.2 *k*-d range search, version 2

To perform a range search, we start with sets L and H of size *k* defining the lower and upper bounds of the search in each dimension. We then use the following algorithm.

```
k-d_Range_Search_v2(G,L,H,A)
A := empty set;
for i := 1 to k do
    determine location b_i of the smallest key in S_i ≥ L_i;
d := 1;
repeat
    if In_Range(b_d, d, k, L, H, G) then
        begin
            add point at location b_d to A;
            b_d := next location in S_d
        end
until K_d > H_d;

In_Range(var b, var d, k, L, H, G) : boolean;
{Determine if a point in a k-d skip list is completely inside a k-d range}
startd := d; startb := b;
repeat
    begin
        K_d := key at location b;
        if K_d < L_d or K_d > H_d then
            begin
                In_Range := false;
                b := next location in S_startd after startb;
                return;
            end
        d := mod(d,k) + 1;
        b := b's dimension pointer to dimension d;
    end
until d = startd;
In_Range := true;
```

Algorithm 7. Range search, version 2, on the *k*-d skip list.

Theorem 7: The time $Q(n, k)$ required to perform a range search on a *k*-d skip list, version 2, is $O(n)$.

Proof

The for loop at line 2 of Algorithm 7 is similar to that of the version 1 range search given in Algorithm 3. Each iteration of the for loop searches for the smallest key in the skip list for dimension i that is larger (or equal to) the lower bound L_i of range i. From Munro et al [1992], it is known that this search requires $\Theta(\log n)$ time. Since this loop is performed k times, the time for the for loop at line 2 is $O(k \log n)$.

The second part of the search uses a repeat loop, and, similar to version 1 of the range search, it halts when a key of one dimension goes out of range. Also similar to version 1, a vector b of size k is used to keep track of the current node being looked at in one dimension's skip list.

Instead of considering only one dimension's range at a time, the range search uses a function "In_Range". This function uses the dimension pointers to connect to this point in the skip lists of the other dimensions. One can then determine in $O(k)$ time (by following the dimension pointers) if the current point is in all k ranges. The points added to set A are thus guaranteed to be in the query range.

For the case of Figure 3 (where no points are in the range, and the set of points is evenly divided among the dimensions), the search starts in dimension 1, discards this point as soon as it finds it is outside range 2, and proceeds to the next point in dimension 2. This point falls outside range 1, and the search proceeds again to dimension 1. This generalizes to k dimensions. The search to see if a point is in range in one dimension takes $O(1)$ time (in the case of Figure 3), and is done once for each point. Thus, the overall time for the case of Figure 3 is $O(n)$. ■

It is interesting to note the case when all points are in the query range. The range search for one dimension finds all of the points, and adds them to set A. The algorithm then halts as the next key for this dimension will be out of range. Thus, $O(kt)$ time is required, when $t = n$.

Another interesting case is when the algorithm finds a point f inside the query range, and the next point is outside; the search proceeds to the next dimension. This same point f will also be found to be inside the query range in the next dimension, and it is added to A even though it is already there based on the search from another dimension. This is due to the fact that the algorithm has no way to tell if a point was already placed in A previously. This leads to the worst case of points being alternately inside and outside the query range as one proceeds from one dimension to the next. Points inside the query range are confirmed to be there k times, giving $O(kt)$ time for discovering them when some points are outside the query range. In addition, the remaining points have to be checked to confirm they are outside at least one dimension's range, requiring $O(n - t)$ time.

Figure 6 illustrates this, with the number of arrows going into a point indicating the number of times it is checked for being inside the query range. Points 3, 6 and 9 are visited twice. Figure 6 also illustrates the order in which the points are visited in the k-d_Range_Search_v2 algorithm; it starts at point 1.

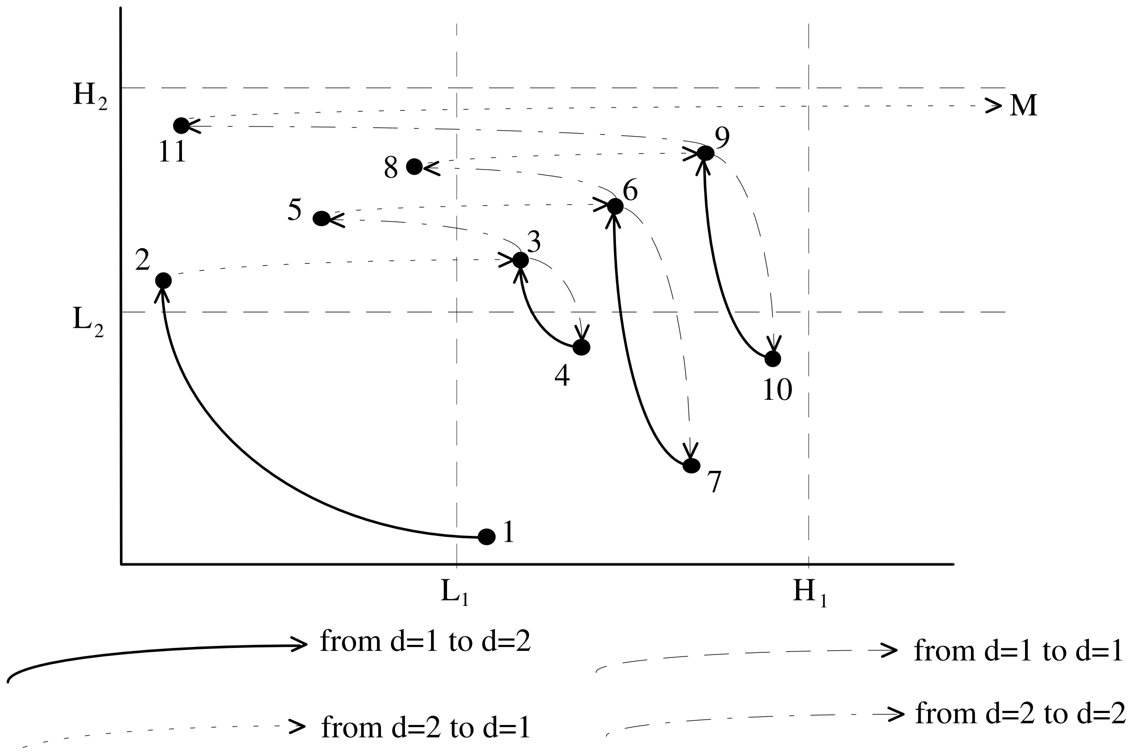

Figure 6. Illustration of *2*-d points alternately inside and outside the query range.

5.3 Insertion and deletion, version 2

Algorithm 8 shows how a new point can be inserted into a version 2 *k*-d skip list. It follows in a straight-forward manner from the construction algorithm.

```
Insert_k-d_v2(G,f)
for d := 1 to k do
    begin
        Allocate a node for J and K_d, and store its location in loc[d];
        Insert node for point f in S_d ordered by K_d;
    end
for d1 := 1 to k do           {Store pointers to all other dimensions}
    for j := 1 to k-1 do
        begin
            d2 := mod(d1 + j - 1, k) + 1;   {Assumes base 1 index for vector loc}
            At node location loc[d1], pointer to dimension d2 := loc[d2];
        end
```

Algorithm 8. Insertion of a new point into a *k*-d skip list, version 2.

Theorem 8: The time I(n, k) required to insert a point into a k-d skip list, version 2, is O($k^2 + k \log n$).

Proof

From Munro et al [1992], we know that inserting a node in a skip list requires $\Theta(\log n)$ time. For a point f, this insertion is performed k times, giving the time for part one of the algorithm (i.e. the for loop at line 1) as O($k \log n$). The second part of the algorithm (double loop at line 6) assigns the dimension pointers for all other dimensions at the new node in all k skip lists. There are k dimension pointers for each of the k skip lists, so this loop requires O(k^2) time. Adding the two contributions together gives the worst case time for insertion I. ■

Algorithm 9 shows the required sequence of steps for deletion of a point from a k-d skip list, version 2.

```
Delete_k-d_v2(G,f)
for d := 1 to k do
    begin
        Find f in S_d by performing a search for K_d;
        Delete the node containing K_d for point f in S_d;
    end
```

Algorithm 9. Deletion of a point from a k-d skip list, version 2.

Theorem 9: The time D(n, k) required to delete a point from a k-d skip list, version 2, is O(k n).

Theorem 10: Assuming that all keys are unique within each dimension, the time D(n, k) required to delete a point from a k-d skip list, version 2, is O($k \log n$).

The proofs of these theorems are identical to the proofs of theorems 4 and 5.

6 The *k*-d skip list, version 3

This version is based on the observation made after the proof of theorem 7 in version 2. Instead of storing each point in *k* nodes (one in each of the *k* skip lists), a point is stored *once* in *one and only one* of the *k* skip lists. The actual list used doesn't matter, as long as the points are equally distributed among the *k* skip lists. In addition, we drop the dimension pointers to further simplify the data structures.

Figure 7 illustrates a *2*-d skip list, version 3, for the example points given in Table 2 and shown in Figure 4.

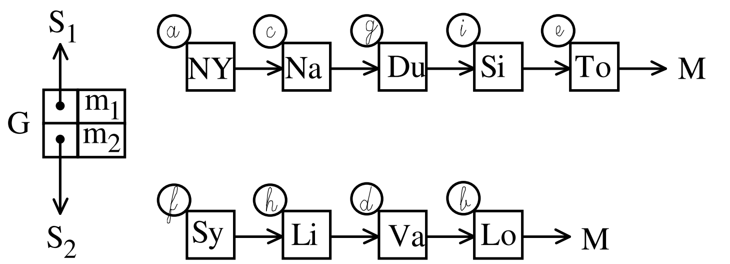

Figure 7. Example *2*-d skip list data structure, version 3.

6.1 Building, version 3

Again, the starting point is a simple list of points in any order. Algorithm 10 can be used to construct this version of the *k*-d skip list.

```
Make_k-d_skip_v3(F,G)
1  Make k empty skip lists S1, ..., Sk;
2  Initialize node counters m1, ..., mk to zero;
3  Make structure G point to these skip lists and node counters;
4  for i := 1 to n do
5    begin
6          Allocate a node for Ji;
7          Insert node for point i in Sd ordered by Kd, for d = mod(i-1, k) + 1;
8          md := md + 1;
9    end
```

Algorithm 10. Construction of a *k*-d skip list, version 3.

Theorem 11: The time P(*n*, *k*) to construct a *k*-d skip list, version 3, is O(*n* log *n*), and it requires O(*n*) space.

Proof

The for loop at line 5 requires one insertion of a node for point i. There are *n* insertions, each requiring Θ(log *n*) time [Munro et al, 1992], giving the worst case time for constructing a *k*-d skip list, version 3, as O(*n* log *n*).

Each point is inserted in one skip list. Knowing that a deterministic skip list of size n requires $2.282n$ level pointers (in one dimension), and that the points are evenly split among all k dimensions, gives the total space requirement for the k-d skip list, version 3, as $O(k(2.282(n/k)) = O(n)$. ■

6.2 *k*-d range search, version 3

To perform a range search, we start with sets L and H of size k defining the lower and upper bounds of the search in each dimension. We then use the following algorithm.

```
k-d_Range_Search_v3(G,L,H,A)
1    A := empty set;
2    for d := 1 to k do
3         begin
4              determine location b of the smallest key Kd in Sd ≥ Ld;
5              while Kd > Hd do
6                   begin
7                        if point at location b is completely inside the query range, add to A;
8                        b := next location in Sd; Kd := key at location b;
9                   end
10             end while
11        end
```

Algorithm 11. Range search for a k-d skip list, version 3.

Theorem 12: The time $Q(n, k)$ required to perform a range search on a k-d skip list, version 3, is $O(n)$.

Proof

The proof here relies on the fact that for any point to be in the query range, all of its keys must be inside the query range. This means that *any* key must also be in the query range. We have used only one key for each point to order it within one skip list. Given the previous argument, this one key must be inside the query range if the point is inside the query range.

The for loop at line 2 iterates over all k dimensions. The while loop at line 5 finds keys until one is out of range. The number of times this executes depends on the query range relationship to the layout of the points.

Figure 3 still illustrates the worst case. Assuming the worst case (i.e. that all points below L_2 are stored in S_1, and that all points to the left of L_1 are stored in S_2), the repeat loop executes n/k times for each dimension. Multiplying this by the k iterations over the k dimensions gives the query time $Q(n, k)$ as $O(n)$. Note that this assumes that the check to see if a point is completely inside a query range (step 7) can be done in $O(1)$ time. ■

It is interesting to note that the best case would be if, fortuitously, all points below L_2 are stored in S_2, and all points to the left of L_1 are stored in S_1. In this case, location b is immediately found to be M, and no points are investigated. Thus, the search would require O(k) time, where the determination of location M is assumed to require O(1) time.

6.3 Insertion and deletion, version 3

Insertion proceeds with determining the skip list with the minimum number of nodes, and inserting the new point into this skip list. Assuming an equal number of nodes per skip list, this requires O(*k* log *n/k*) time.

Deletion requires that the point to be deleted first be found. Since it could be in any one of the *k* skip lists, this requires, in the worst case, O(*k* log *n/k*) time, where O(log *n/k*) time is required to search each skip list.

7 The priority search tree

A priority search tree holds *2*-d data. The radix PST is constructed using the following (pseudo-Pascal) definitions [from McCreight, 1985]:

```
CONST
    k = 30000;
    FirstKey = 0;
    LastKey = k - 1;
FirstNonKey = LastKey + 1;

TYPE
    KeyRange = FirstKey..LastKey;
    KeyBound = FirstKey..FirstNonKey;
    Pair = RECORD x,y:KeyRange END;
    RPSTPtr = ->RPST;
    RPST = RECORD
        p: Pair;
        left, right: RPSTPtr
     END;
```

The radix PST has the following properties:

1. Each data pair (x,y) appears in the p field of exactly one node of the tree.
2. Priority queue invariant on y values:
    For any node t, if t.left is not NIL, then t.p.y <= t.left->.p.y;
    if t.right is not NIL, then t.p.y <= t.right->.p.y
3. Each node t has an associated x interval [lower..upper) within which t.p.x lies. The root of the tree is the interval KeyBound. The x-interval for node t.left-> is [lower..$\lfloor$(lower+upper)/2$\rfloor$). The x-interval for node t.right-> is [$\lfloor$(lower+upper)/2$\rfloor$..upper)

Figure 8 shows a radix PST for the *2*-d data of Table 2.

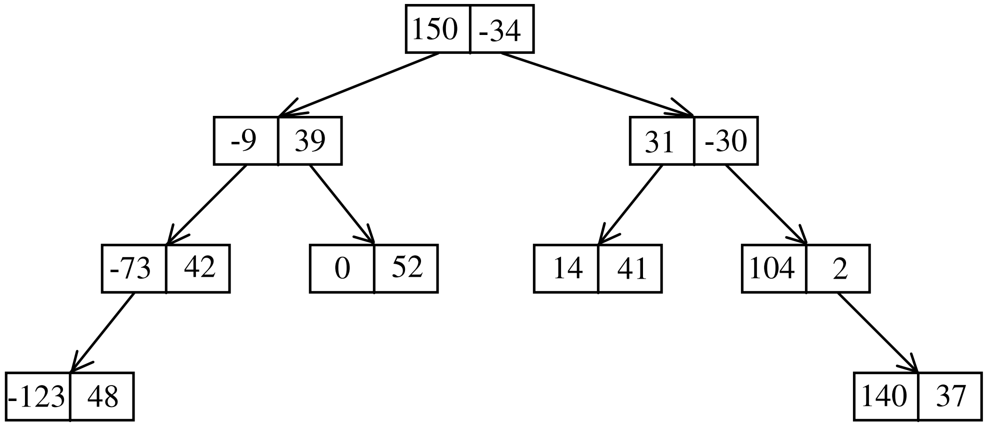

Figure 8. A radix priority search tree for the *2*-d data of Table 2.

Notice that the y value at each node is the minimum of all y values in the left and right subtrees attached to it. This follows from property 2. With this data structure, a semi-infinite range search of the form ([x0:x1], [-∞,y1]) can be performed. Algorithm 12 can be used for this purpose.

```
1   PST1_Range_Search(t:RPSTPtr; x0, x1, y1:KeyRange; lowerx:KeyRange;
upperx:KeyBound);
2   var  middlex : KeyRange;
3   begin
4       if t <> NIL then
5           if t->.p.y <= y1 then
6               begin
7                   if (x0 <= t->.p.x) and (t->.p.x <= x1) then report(t->.p);
8                   middlex := (lowerx + upperx) DIV 2;
9                   if x0 < middlex then
10                      PST1_Range_Search(t->.left, x0, x1, y1, lowerx, middlex);
11                  if middlex <= x1 then
12                      PST1_Range_Search(t->.right, x0, x1, y1, middlex, upperx);
13              end;
14  end;
```

Algorithm 12. Semi-infinite range search using a radix priority search tree (adapted from McCreight [1985]).

As each node is encountered, its y value is compared (at line 5) against y1, the upper limit of the y range. If the y value at a node is greater than y1, then the complete subtree attached to this node can be pruned as all y values in the subtree are greater than this node’s y value. Property 3 of the radix PST is then used to prune the search based on the x range.

Based on property 1, the storage required for the PST is $O(n)$. McCreight [1985] shows that the following properties also hold for the PST:

$P(n, 1.5) = O(n \log n)$, $Q(n, 1.5) = O(t + \log n)$ and $I(n, k) = D(n, k) = O(\log n)$

where I have taken the liberty of replacing *k* by *1.5* since only semi-infinite range queries are supported.

McCreight [1985] also develops the balanced priority search tree that does not depend on the radix structure for ordering the keys. The *2*-d search skip list below uses the actual keys and is a balanced structure.

8 A *2*-d search skip list

A new structure based on a skip list is now introduced. It uses the priority search tree idea described above. The structure 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*. The properties of this structure are

1. Each node contains four values. These are
   (i) (K_1, K_2) = keys for this node,
   (ii) mink2 = minimum K_2 value for all nodes in the left subtree of this node, and
   (iii) maxk2 = maximum K_2 value for all nodes in the left subtree of this node.
2. 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 left subtree is defined as all the descendant nodes and their siblings which have K_1 key values ≤ the K_1 key value of this node.
3. Three special nodes called head, tail and bottom are used to indicate the start and end of list condition. The head node is always at a height one greater than the height of the skip list. The head node also has mink2 = minimum K_2 value for all nodes in the search skip list, and maxk2 = maximum K_2 value for all nodes in the search skip list.
4. All data points appear at the leaf or bottom level, and some points also appear at nodes above the bottom level.
5. Every gap size in the skip list is of size 1, 2 or 3. Gap size is defined as the number of elements of height h-1 that exist between two linked elements in the skip list. Links are between two elements of height h and h or higher.
6. The skip list is ordered by the K_1 key values.

An element (used in describing property 5) is a vertically stacked set of nodes linked directly by down pointers. As described in Papadakis [1993], this structure is somewhat similar to a 2-3-4 tree. Figure 9 illustrates the composition of a single node, and Figure 10 shows a *2*-d search skip list for the data of Table 2. Figure 11 shows the same skip list with K_1 values only, organized as skip lists are portrayed in e.g., Pugh [1990].

K_1value	K_2value
minimum K_2value in left subtree	maximum K_2value in left subtree
down pointer	right pointer

Figure 9. Composition of a single node in a *2*-d search skip list.

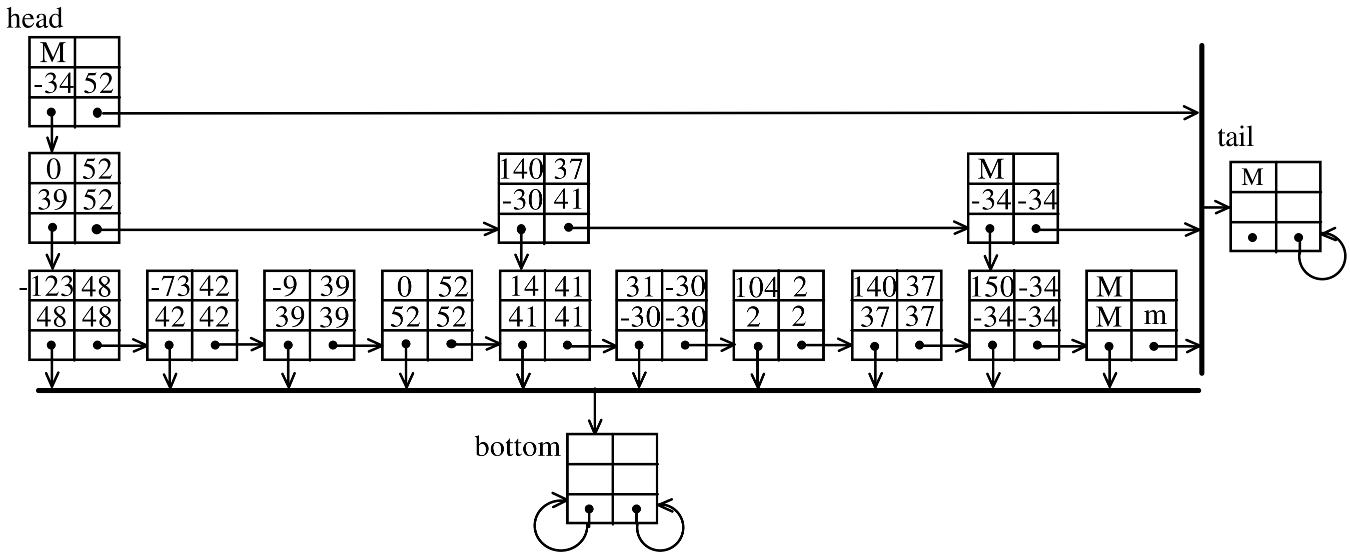

Figure 10. A 2-d search skip list for the data of Table 2.

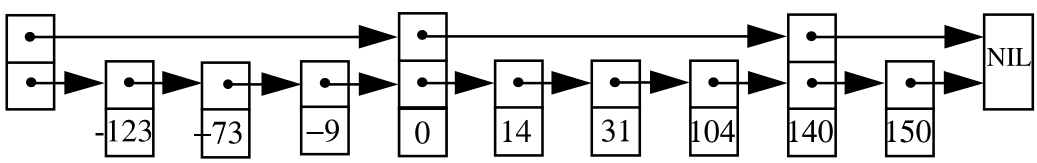

Figure 11. The skip list of Figure 9 shown in "standard" skip list fashion.

The 2-d search skip list has the keys pulled back into nodes on the upper levels. Note that M stands for the maximum key value, and m stands for the minimum key value. The left subtree for node (140, 37, -30, 41) at level 1 in the 2-d search skip list of Figure 10 includes the level 0 nodes with K_1 values 14, 31, 104 and 140. An element of height 2 consists of the two vertically stacked nodes (140, 37, -30, 41) and (14, 41, 41, 41).

8.1 Building a 2-d search skip list

Constructing a search skip list requires inserting each point into the structure sequentially, starting with an empty search skip list. Algorithm 13 can be used for this.

```
#define maxkey 2147483647   /*  maxkey := (2^31)-1  */
#define minkey -2147483647  /*  minkey = -(2^31) + 1    */
#define maxht  64           /*  for deletion stack  */
#define sampleSize 11
#define newnode (nodePtr)malloc(sizeof(struct node))

#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

typedef struct node * nodePtr;

struct node{
   int      k1, k2, mink2, maxk2;
   nodePtr  r, d;
};

nodePtr  head, bottom, tail;

void Empty_ssl_2d()
{
   head = newnode;
   bottom = newnode;
   tail = newnode;
   head->key = maxkey;
   head->d = bottom;
   head->r = tail;
   head->miny = maxkey;
   head->maxy = minkey;
   bottom->r = bottom;
   bottom->d = bottom;
   tail->key = maxkey;
   tail->r = tail;
}

main()
{
   int            i, keys1[sampleSize],
                     keys2[sampleSize];
.
. /* Define the keys for all the input 2-d points. */
.
   Empty_ssl_2d();
   for (i=0; i<sampleSize; i++)
      if (Search_ssl_2d(keys1[i]) == bottom)
         Insert_ssl_2d(keys1[i],keys2[i]);
}
```

Algorithm 13. Construction of a 2-d search skip list.

The notation `keys1[i]` and `keys2[i]` refer to keys of point i for dimensions 1 and 2, respectively. `newnode` allocates storage for a new node, and returns a pointer to it. `Search_ssl_2d` searches a 2-d search skip list for a point with this K_1 value; if a point in the 2-d search skip list already has a K_1 key = `keys1[i]`, a value of 1 is returned, otherwise 0. The `Insert_ssl_2d` algorithm is given below under the section on insertion. Theorems about the space and time requirements for building a 2-d search skip list are also discussed there.

8.2 Insertion for a *2*-d search skip list

Algorithm 14 can be used for insertion of points into a *2*-d search skip list. Note that the algorithm relies on the global variables head, tail and bottom as defined above in Algorithm 13. In addition, the algorithm assumes that all of the K_1 keys are unique. In summary form, the algorithm steps are as follows:

1. Start at the head node, and at one level higher than the height of the skip list.
2. If the gap we are going to drop into is of size 1 or 2, determine where to drop, and then check the new point's K_2 value. If less than the dropping node's minimum K_2 value, or greater than the dropping node's maximum K_2 value, change the minimum K_2 value and/or maximum K_2 value to be the new point's K_2 value.
3. If the gap is of size 3, we first raise the middle element in this gap, creating two gaps of size 1, and then we drop. The new node created in the middle determines its minimum and maximum K_2 value from its two direct descendants and, if the new K_1 key goes in the left subtree for this new node, the new point's K_2 value.
4. When we reach the bottom level, we insert a new element of height 1. This new element has a minimum and maximum K_2 value = K_2.

This algorithm allows only gaps of sizes 1 and 2, and the resulting skip list with a newly inserted element is a valid 1-3 skip list (i.e. has gap sizes of only 1, 2, or 3). In addition, since the minimum and maximum K_2 values are always determined with respect to the new point's K_2 value, the mink2 and maxk2 values for a node will always be correct.

The algorithm as shown in Algorithm 14 below corresponds to the actual C code used for insertion.

```
int Insert_ssl_2d(c1, c2)
/* c1 = k1 key value,  c2 = k2 key value.
   inx = boolean; true if c1 goes in x's subtree.
   add1 = boolean; true if a node was added at this level.
*/
register int c1, c2;
{
   nodePtr  t, x;
   register int      success;
   register int      inx, add1;

   x = head;
   bottom->k1 = c1; bottom->k2 = c2;
   bottom->mink2 = c2; bottom->maxk2 = c2;
   success = 1;
   while (x != bottom)  {      /* do at each level */
      while (c1 > x->k1)   /* find where you drop */
         x = x->r;
      add1 = 0;
      if  (x->k1 > x->d->r->r->k1) { /* if 3 elms at level below, or at */
         t = newnode;                /*   at bottom level & must insert.*/
                                     /*   ==> raise middle elm          */
      /*  Add a node.  Keep track of mink2 and maxk2 as node is added.  */
            t->r = x->r;
            t->d = x->d->r->r;
            x->r = t;
            inx = x->d->r->k1 > c1;
            t->k1 = x->k1; t->k2 = x->k2;
            t->mink2 = min(t->d->mink2, t->d->r->mink2);
            t->maxk2 = max(t->d->maxk2, t->d->r->maxk2);
            if (!inx) {
               t->mink2 = min(t->mink2, c2); t->maxk2 = max(t->maxk2, c2);
             }
            if (t->d == bottom) {   /* leaf level */
               t->mink2 = x->mink2; t->maxk2 = x->maxk2;
             }
            x->k1 = x->d->r->k1; x->k2 = x->d->r->k2;
            x->mink2 = min(x->d->r->mink2, x->d->mink2);
            x->maxk2 = max(x->d->r->maxk2, x->d->maxk2);
            if (inx) {
               x->mink2 = min(x->mink2, c2); x->maxk2 = max(x->maxk2, c2);
             }
         add1 = 1;
      }
      else  if (x->d == bottom)  /* if insert_Key already in DSL */
         success = 0;
      if(x->d != bottom && !add1) {
         x->mink2 = min(x->mink2, c2); x->maxk2 = max(x->maxk2, c2);
       }
      x = x->d;
   }
   if (head->r != tail)  {   /* raise height of DSL if necessary */
      t = newnode;
      t->d = head; t->r = tail;
      t->k1 = maxkey;
      t->mink2 = min(t->d->mink2, t->d->r->mink2);
      t->maxk2 = max(t->d->maxk2, t->d->r->maxk2);
      head = t;
   }
   return(success);
}
```

Algorithm 14. Insertion of a point into a 2-d search skip list (adapted from Papadakis [1993]).

As an example of insertion, Figure 12 shows the *2*-d search skip list of Figure 9 after inserting the following two points into it:

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

Notice that the left subtree for node (0, 52, -23, 52) at level 2 includes two nodes at level 1 (i.e. (-73, 42, 42, 48) and (0, 52, -23, 52)), along with their left subtrees.

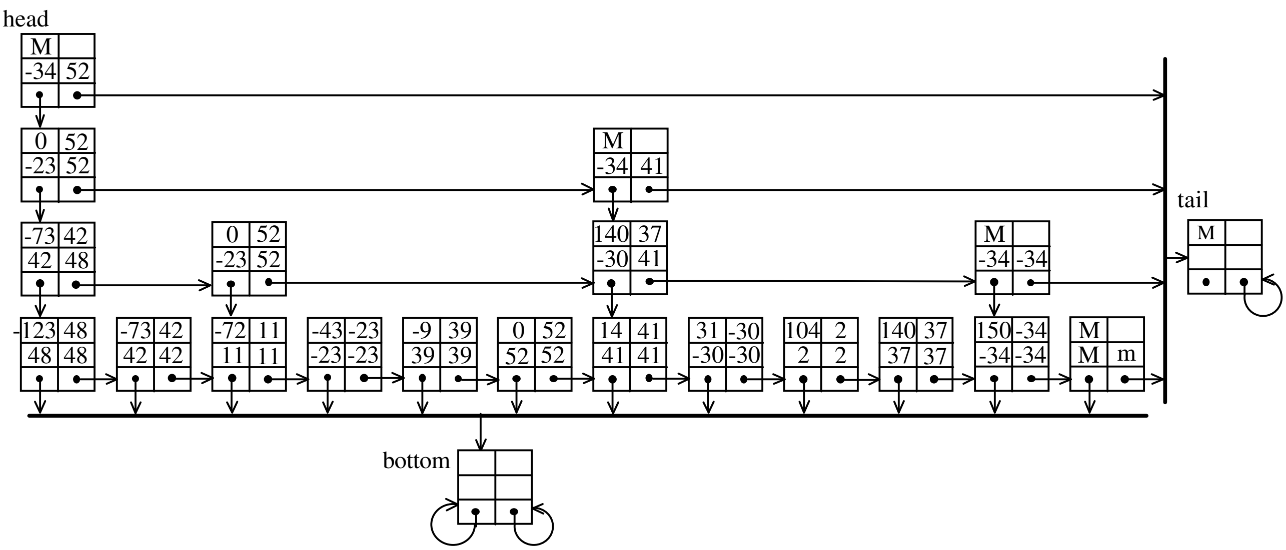

Figure 12. Example *2*-d search skip list after inserting two more points.

Define path length as the number of nodes encountered when traversing a *2*-d search skip list.

Lemma 1: The path length from the head node to the bottom level in a *2*-d search skip list is O(log *n*).

Proof

The longest path length for the *2*-d search skip list occurs when all gaps are of size 1. This means there will be $n + 1$ nodes at level 0, $\left\lfloor \frac{n}{2} \right\rfloor + 1$ nodes at level 1, $\left\lfloor \frac{n}{4} \right\rfloor + 1$ nodes at level 2, and, in general, $\left\lfloor \frac{n}{2^\ell} \right\rfloor + 1$ nodes at level ℓ. Solving $\left\lfloor \frac{n}{2^\ell} \right\rfloor + 1 = 2$ for ℓ gives $\ell = \lfloor \log_2 n \rfloor$ for the level below the head node. The maximum height of the *2*-d search skip list is thus $\lfloor \log_2 n \rfloor + 1$, where a search skip list with nodes only at the leaf level has a height of one.

The longest path length is followed by traveling from the head node to the node with the largest K_1 value. The number of down pointers followed is exactly the maximum height of the 1-*2*-d search skip list plus one (for the down pointer from the head node), i.e. $\lfloor \log_2 n \rfloor + 2$. The number of right pointers followed is one less than this; i.e. $\lfloor \log_2 n \rfloor + 1$. The total path length in the worst case is therefore $2\lfloor \log_2 n \rfloor + 3$. Thus, the longest path from the head node to the bottom level is O(log *n*). ■

Theorem 13: The time I(*n*, *2*) required to insert a new point into a *2*-d search skip list is O(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 *2*-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 *2*-d search skip list, which is (from the proof of Lemma 1) $\lfloor \log_2 n \rfloor + 2$ (as a new head node may be required if the height increases by one). This gives the time I(*n*, *2*) to insert a new point into a *2*-d search skip list as O(log *n*). ■

Theorem 14: The space S(*n*, *2*) required for storage of a *2*-d search skip list is O(*n*).

Proof

Each node in the *2*-d search skip list has two pointers and four data values. From Lemma 1, it is known that the number of nodes at each level ℓ is $\left\lfloor \frac{n}{2^\ell} \right\rfloor + 1$, and that the maximum height of the *2*-d search skip list is $\lfloor \log_2 n \rfloor$. The total number of nodes in the worst case is thus $\sum_{\ell=0}^{\lfloor \log_2 n \rfloor} \left(\left\lfloor \frac{n}{2^\ell} \right\rfloor + 1 \right)$. Evaluating this sum results in O(log *n*) + O(2*n*). Assuming that each pointer and data value occupy one storage space, the worst case storage requirement for a *2*-d search skip list is 6(O(log *n*) + O(2*n*)) = O(*n*). ■

Theorem 15: Using Algorithm 13, the time P(*n*, *2*) list required to construct a *2*-d search skip list is O(*n* log *n*).

Proof

Each point to be inserted into the *2*-d search skip list is first checked (at line 5) 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 *2*-d search skip list. From Theorem 13, we know that this requires O(log *n*) time. These operations are carried out a total of *n* times, once for each point inserted into the *2*-d search skip list, resulting in a total time of 2*nO*(log *n*) = O(*n* log *n*). ■

8.3 Range search for a *2*-d search skip list

Algorithm 15 can be used for a range search of a *2*-d search skip list. This is the actual C code used for the range search.

```
   void RSearch_ssl_2d(t, L1, H1, L2, H2, maxk1)

   register  nodePtr t;
   register  int     L1, H1, L2, H2, maxk1;
   {
1     if (t != bottom && t != tail) {
2        if (t->d != bottom && t->mink2 <= H2 && t->maxk2 >= L2) {
3           if (L1 <= t->k1)RSearch_ssl_2d(t->d, L1, H1, L2, H2, t->k1);
4           if (t->k1 < H1 && t->k1 < maxk1)
5              RSearch_ssl_2d(t->r, L1, H1, L2, H2, maxk1);
6         }
7        else if (t->d == bottom) {
8           if (t->k2 <= H2 && t->k2 >= L2 &&
9                 L1 <= t->k1 && t->k1 <= H1)
10             printf("  Point (%d, %d) - inside.\n", t->k1, t->k2);
11          if (t->k1 < H1 && t->k1 < maxk1)
12             RSearch_ssl_2d(t->r, L1, H1, L2, H2, maxk1);
          }
      }
   }
```

Algorithm 15. Range search for a *2*-d search skip list.

Theorem 16: The time Q(n, *2*) list required for range search in a *2*-d search skip list is O(n).

Proof

The worst case for *2*-d range search with a *2*-d search skip list is illustrated in Figure 13. In this case, there are no points in the range, but the `RSearch_ssl_2d` 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 key values. This means that the left subtrees of all nodes in the upper levels have the same minimum and maximum K_2 key values. Thus, the test at line 2 of the `RSearch_ssl_2d` algorithm never prunes any subtrees as the `mink2` value is always less than `H2`, and the `maxk2` value is always greater than `L2`, until the bottom level is reached. This means that O(n) nodes are checked to determine that no points are in the desired range. ■

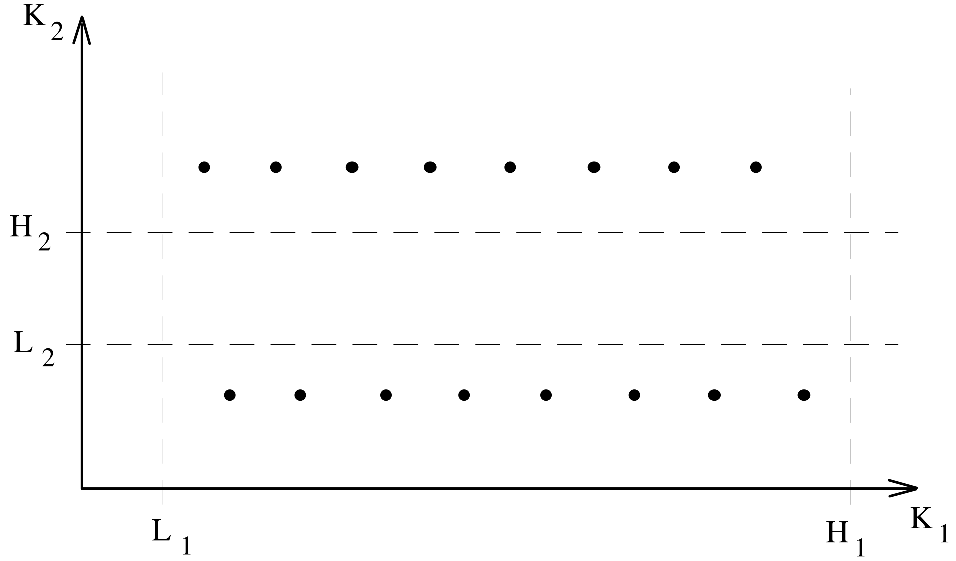

Figure 13. Illustration of worst case for *2*-d range search with a *2*-d search skip list.

Interestingly, the *2*-d search skip list can be used for semi-infinite range search, in either the $([L_1:H_1],[L_2:\infty])$, or the $([L_1:H_1],[-\infty:H_2])$ mode. We call these the "Up" mode and "Down" mode, respectively. Algorithm 16 can be used for the Up semi-infinite range search. The primary difference between this algorithm and the general *2*-d range search in Algorithm 15 is at line 2, where the subtree pruning only occurs if the maximum K_2 value `t->maxk2` is less than the lower bound on the semi-infinite query for the K_2 keys (i.e. all K_2 keys are now out of range).

```
   void RSearchUp_ssl_2d(t, L1, H1, L2, maxk1)

   register  nodePtr t;
   register  int     L1, H1, L2, H2, maxk1;
   {
1     if (t != bottom && t != tail) {
2        if (t->d != bottom && t->maxk2 >= L2) {
3           if (L1 <= t->k1)RSearchUp_ssl_2d(t->d, L1, H1, L2, t->k1);
4           if (t->k1 < H1 && t->k1 < maxk1)
5              RSearchUp_ssl_2d(t->r, L1, H1, L2, maxk1);
6         }
7        else if (t->d == bottom) {
8           if (t->k2 >= L2 &&
9                 L1 <= t->k1 && t->k1 <= H1)
10             printf("  Point (%d, %d) - inside.\n", t->k1, t->k2);
11          if (t->k1 < H1 && t->k1 < maxk1)
12             RSearch_ssl_2d(t->r, L1, H1, L2, H2, maxk1);
          }
      }
   }
```

Algorithm 16. Semi-infinite range search for a *2*-d search skip list.

Theorem 17: The time $Q(n, 2)$ list required for semi-infinite range search in a 2-d search skip list is $O(t + \log n)$, for t = number of points found in the range.

Proof

Each of the six cases shown in Figure 14 must be considered for range search. K_{min} and K_{max} correspond to the minimum and maximum values of the keys, and L and H refer to the range search limits (lower and higher) for this dimension's keys.

We only need to consider the K_1 key overlaps, since the only additional information provided by the maximum K_2 key values is to prune the search space. In other words, line 2 of the `RSearchUp_ssl_2d` algorithm can only reduce the amount of time required for the search.

Case (a). Here, we have $H_1 < K_{min}$ so no right pointers will ever be followed (test at line 4). L_1 is always less than `t->k1`, so the down pointers are followed until the bottom or leaf level is reached. After finding that the first point is not within the range, the test at line 11 finds that `t->k1` > H_1 and the search stops. The time required for this is thus equal to the maximum height of the 2-d search skip list, which, from Lemma 1, is known to be $O(\log n)$.

Case (b). Here, both down and right pointers are followed. Following right pointers is halted as soon as a node is encountered with `t->k1` > H_1. This means that only the part of the search skip list containing K_1 keys $\le H_1$ is searched (i.e. O(2t) nodes), plus the extra nodes tested to determine that `t->k1` > H_1. These extra nodes number a maximum of the height of the list, i.e. $O(\log n)$. The total number of nodes searched is thus $O(2t) + O(\log n) = O(t + \log n)$.

Case (c). Again, both down and right pointers are followed. Following down pointers is halted as soon as L_1 > `t->k1`; i.e. the left subtree is pruned. Following right pointers is halted when `t->k1` > H_1. Thus, only the part of the search skip list containing K_1 keys between L_1 and H_1 is searched (i.e. O(2t) nodes), plus the extra nodes tested to determine that L_1 > `t->k1` and that `t->k1` > H_1. As for case (b), these extra nodes number a maximum of the height of the list for each side, so the total number of nodes searched is thus $O(2t) + O(\log n) = O(t + \log n)$.

Case (d). This case uses the down pointer argument of case (c), resulting in a maximum number of nodes searched equal to $O(t + \log n)$.

Case (e). The down pointers are never followed in this case as L_1 > `t->k1`, halting the downward search at line 5, except for nodes that have the maximum key value as their K_1 key. The number of these nodes equals the height of the 2-d search skip list, which we know is $O(\log n)$. The same number of right pointers are followed from nodes directly below these nodes with maximum keys. The resultant total number of nodes searched is $2O(\log n) = O(\log n)$.

Case (f). Here, all nodes in the 2-d search skip list are visited to determine that all n of them are in the range. The number of nodes in the search skip list is known from Theorem 14 to be $O(\log n) + O(2n)$. In this case, $t = n$, and thus the number of nodes searched is $O(t + \log n)$.

Looking at the results of all cases, we see that the time requirement for any of them is never worse than $O(t + \log n)$, which proves the theorem. ■

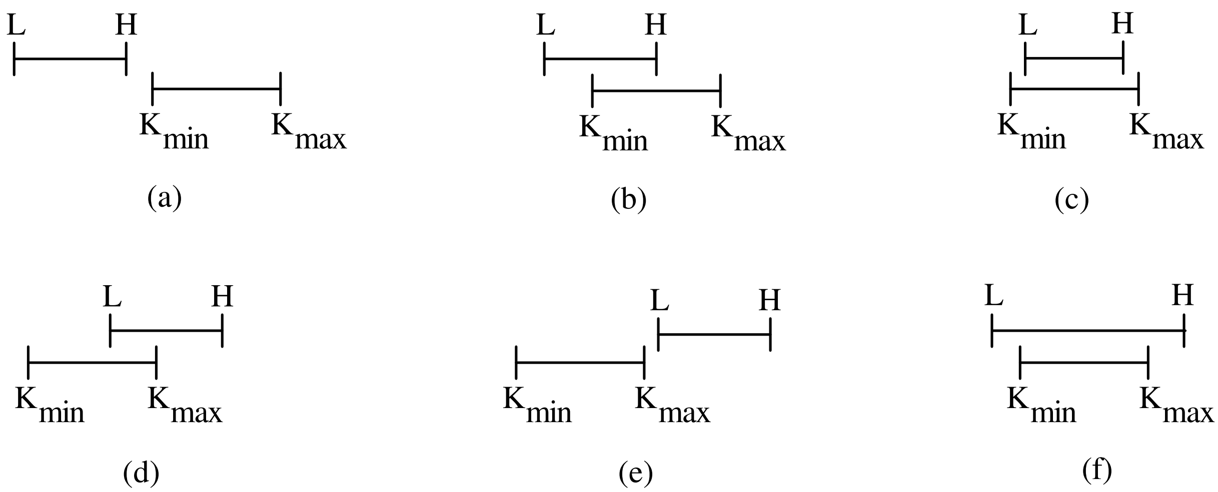

Figure 14. Possible overlap cases for the proof of Theorem 17.

8.4 Deletion in a *2*-d search skip list

Deletion for a *2*-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 one. Borrowing from the next (or previous) gap causes a node to have a different left subtree. This, in turn, means that the `mink2` and `maxk2` 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 17.

1. Start search at the header, at a level = height of skip list.
2. The current node has a gap G of size 1, 2 or 3; record the current node in an array P.
3. If gap G we are going to drop into is of size 2 or 3, drop.
4. If the gap G is of size 1, then:
5. If G is **not** the last gap in the current level, then
6. If the following gap G' is of size 1, merge G and G' (by lowering the element separating G and G').
7. Else if G' is of size 2 or 3, 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 1, merge G and G' (by lowering the element separating G and G').
10. Else if G' is of size 2 or 3, 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 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 key to be deleted does not have height 1, swap with 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 values for all nodes on the deletion path.

Algorithm 17. Summary of deletion steps in a *2*-d search skip list.

Algorithm 18 can be used for deletion of a point from a *2*-d search skip list. Again, Algorithm 18 corresponds exactly to the C code used for deletion in several experiments.

```
int Delete_ssl_2d(c1)
int c1;
{
   register  nodePtr  x, px, nx, t, pla[maxht];
   register  int      pred, lastAbove, success, predy, npla, i;

   x = head->d;          /* x = current elm in search path */
   success = (x != bottom);
   bottom->k1 = c1;
   lastAbove = head->k1;  /* rightmost key on search path at level above */
   npla = 0; pla[npla] = head;
   while  (x != bottom)  {   /* do at every level */
      while  (c1 > x->k1) {   /* find where you drop */
         px = x;              /* keeping track of the previous gap */
         x = x->r;
      }
      nx = x->d;   /* mark where to drop at level below */
      if  (x->k1 == nx->r->k1)  /* if {only 1 elm in gap to drop} or... */
                                 /* ...  {at bottom level + must delete} */
         if  (x->k1 != lastAbove) { /* if DOESN'T drop in last gap */
            t = x->r;
            if  ( (t->k1 == t->d->r->k1) || (nx == bottom) )  {
            /*if only 1 elm in next gap or at bottom level ...*/
               x->r = t->r;    /* ... lower separator of current+next gap */
               x->k1 = t->k1;
               x->k2 = t->k2; x->mink2 = t->mink2; x->maxk2 = t->maxk2;
               x->name = t->name;
               free(t);
            }
            else {  /* if >=2 elms in next gap */
               x->k1 = t->d->k1;  /* raise 1st elm in next gap & lower... */
               t->d = t->d->r;      /* ... separator of current+next gap */
               newMaxMin(t);
            }
         }
         else                       /** if DOES drop in last gap **/
            if (px->k1 <= px->d->r->k1) { /* if only 1 elm in prev. gap */
               if  (nx == bottom) { /* if del_Key is in elm of height>1 */
                  pred = px->k1; /* predecessor of del_key at bottom level*/
                  predk2 = px->k2;
                }
               px->r = x->r;     /* lower separator of previous+current gap */
               px->k1 = x->k1;
               if (nx != bottom) {
                  px->k2 = x->k2; px->maxk2 = x->maxk2; px->mink2 = x->mink2;
                }
               free(x);
               x = px;
            }
            else  {                     /* if >=2 elms in previous gap */
               /*  t = last elm in previous gap */
               t = (px->k1 == px->d->r->r->k1 ?  px->d->r : px->d->r->r);
               px->k1 = t->k1; /* raise last elm in previous gap &     */
               x->d = t->r;   /* lower separator of previous+current gap */
               newMaxMin(px);
            }
      else  if (nx == bottom)  /* if del_Key not in DSL */
         success = 0;
      lastAbove = x->k1;
      npla++; pla[npla] = x;
      x = nx;
   }
```

Algorithm 18 (part 1 of 2). Deletion from a 2-d search skip list (adapted from Papadakis [1993]).

```
   /* Do a 2nd pass, for the case that del_key was in elm of height>1 */
   x = head->d;
   while  (x != bottom) {
      while  (c1 > x->k1)
         x = x->r;
      if  (c1 == x->k1) {
         x->k1 = pred;
         x->k2 = predk2;
       }
      x = x->d;
   }
   for (i=npla; i>=0; i--) /* Check and reset mink2 and maxk2 for all nodes */
      newMaxMin(pla[i]);   /*  on path to deleted key.                    */
   if  (head->d->r == tail)  {   /* lower header of DSL, if necessary */
      x = head;
      head = x->d;
      free(x);
   }
   return(success);
}

void newMaxMin(x)
nodePtr x;
{
   nodePtr  t;
   int      repeat;
   t = x->d;
   if (t != bottom) {
      repeat = 1;
      x->maxk2 = t->maxk2; x->mink2 = t->mink2;
      while (t->k1 <= x->k1 && repeat) {
         if (t->k1 == maxkey) repeat = 0;
         if (t->maxk2 > x->maxk2)x->maxk2 = t->maxk2;
         if (t->mink2 < x->mink2)x->mink2 = t->mink2;
         t = t->r;
       }
    }
 }
```

Algorithm 18 (part 2 of 2). Deletion from a 2-d search skip list (adapted from Papadakis [1993]).

Theorem 18: The time $D(n, 2)$ required to delete a point from a 2-d search skip list is $O(\log n)$.

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 values. The borrowing requires the changing of one pointer. Resetting minimum and maximum K_2 values is done by a call to the `newMaxMin` function, 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 2-d search skip list is given by 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 values, which requires $O(\log n)$ time. The total time required is thus $3O(\log n) = O(\log n)$ time. ■

9 Conclusions

Four new dynamic data structures for multidimensional data have been presented. Three versions of the k-d skip list were presented, along with a 2-d search skip list. All of these structures are based on the deterministic skip list. Table 3 summarizes the results.

Table 3. Summary of multidimensional skip list data structure performance.

Data Structure	$P(n, k)$	$S(n, k)$	$Q(n, k)$	$I(n, k)$	$D(n, k)$
k-d skip list, version 1	$O(k\,n \log n)$	$O(k\,n)$	$O(n \log (n/k + t))$, $O(k(\log n + t \log t))$ for well-distributed data and t small relative to n	$O(k \log n)$	$O(k\,n)$ for non-unique keys, $O(k \log n)$ for unique keys
k-d skip list, version 2	$O(k\,n(k + \log n))$	$O(k^2\,n)$	$O(k\,n)$, $O(k\,t)$ for all points in query range	$O(k^2 + k \log n)$	$O(k\,n)$ for non-unique keys, $O(k \log n)$ for unique keys
k-d skip list, version 3	$O(n \log n)$	$O(n)$	$O(n)$, $O(t)$ for best case distribution	$O(k \log (n/k))$	$O(k \log (n/k))$
2-d search skip list	$O(n \log n)$	$O(n)$	$O(n)$, $O(t + \log n)$ for $k = 1.5$	$O(\log n)$	$O(\log n)$

All of the k-d skip lists show worst case performance for k-d range search of $O(n)$ compared to the query time of $O(t + \log^{k-1} n)$ for the RT-tree of Edelsbrunner (where t is the number of points reported). This worst case is due to the projection method employed, and is not expected to arise in practice. For a small number of reported points (relative to n), the expected search time for version 1 of the k-d skip list is linear in k. The worst case deletion time of $O(kn)$ compares poorly with deletion times of $O(\log^k n)$ for the RT-tree, but makes no assumption about uniqueness of the keys. If the keys can be assumed to be unique, then $D(n, k)$ requires $O(k \log n)$ time (for versions 1 and 2 of the k-d skip list), which is better than for the RT-tree. Version 3 of the k-d skip list has even better worst case times for insertion and deletion than versions 1 and 2.

Version 3 of the k-d skip list has the interesting property of its storage space requirement being independent of k. Points are only inserted once in the structure. This version of the k-d skip list is of a particularly simple construction, and seems to be a good fit for multiple processor computers. Each of the k skip lists are independent of all the others, and parallel processing algorithms should be able to improve $Q(n,k)$ to $O(n/k)$ by assigning each skip list to one processor. For large *k*, this might be an attractive option compared to the RT-tree, which has a worst case range search time exponential in *k*, and has substantially more complex operations for dynamic maintenance of the structure.

The *2*-d search skip list is a very practical way to carry out semi-infinite range search in optimal time. The algorithms for insertion and deletion are shown here in their complete C code implementation, and they inherit the simplicity of the skip list algorithms in general. Deletion, in particular, takes advantage of the extra horizontal (i.e. right) pointers (which are not normally incorporated in B-tree structures) to simplify the top-down deletion algorithm.

All of the structures introduced here suffer from a worst case *k*-d range search time of $O(n)$ (for $k \geq 2$). A careful consideration of why this is so, compared to the RT-tree and range tree, offers some insight as to where future progress can be made in improving worst case time bounds for *k*-d range search. For the *2*-d case, both the RT-tree and range tree have structures that are ordered on *both* of the key values (i.e. both K_1 and K_2 ordered structures). The RT-tree has a primary structure ordered on K_2, and the secondary structures are the priority search trees (both Up and Down versions) ordered on K_1 values. The range tree has a primary structure ordered on K_1 values, and a secondary structure attached to each node of the primary structure ordered on K_2 values. It seems clear that structures must exist which order all *k* of the dimension's keys to avoid worst case queries like that illustrated in Figure 13 for the *2*-d search skip list. The *2*-d search skip list has an ordering of only the K_1 values, and does not have any structure that orders the K_2 values.

Another observation is equally important. The *k*-d skip lists do have structures that order all *k* of the dimension's keys, but they also suffer from $O(n)$ worst case range search times. The reason for this is the way in which the orderings are related to one another. In the *k*-d skip lists, each ordering is essentially independent of the other. Pruning the search in one dimension has no effect on the other dimensions. In the RT-tree and range tree, the orderings of one dimension are highly dependent on the orderings of the others. For example, establishing that the range query has the root node as common ancestor for the RT-tree means that only the priority search trees for the left and right child nodes of the root node have to be searched. These priority search trees have precisely half of the points each, where the halfway dividing point is established by the primary structure that is ordered on the K_2 values. This interdependence makes the structures more time-consuming for insertion and deletion, but necessary to achieve their worst-case range search times.

This research leads to other questions, namely:

(1) What happens when the *2*-d search skip list is extended to a *k*-d search skip list, such that the minimum and maximum of $(k - 1)$ dimensions are kept in the nodes (i.e. minimum and maximum K_2, K_3, ..., K_k values)? It appears that this will allow a very fast semi-infinite range search, with $+\infty$ or $-\infty$ being part of all *k* ranges in the query except for dimension 1, where an exact query can be given.

(2) Can the interval skip list [Hanson and Johnson, 1992] be incorporated to give a rectangle search capability for multiple dimensions?

(3) Can the worst case time for *k*-d range search be improved by reorganizing the *2*-d (or *k*-d) search skip list somehow to give better pruning of points as the search progresses? One possible method for this is suggested by the *k*-ranges of Bentley and Maurer [1980]. Another

would be a reorganization of the 2-d search skip list in a fashion similar to the RT-tree of Edelsbrunner [1981].

(4) Given some assumptions about the distribution of the data (e.g. randomly distributed points in the *k*-d space), what are the expected worst case time requirements for *k*-d range search for the new structures introduced here?

Acknowledgments

I would like to thank the University of Maryland Institute for Advanced Computer Studies (UMIACS) for their generous support during my three month visit. UMIACS provided an office, a UNIX computer, technical and administrative assistance, and access to printing and copying facilities. I would also like to thank the University of New Brunswick for providing some financial assistance to make this visit possible. Dr. Hanan Samet is kindly acknowledged for providing constructive discussion during the course of this work, for providing me access to his substantial library, and for the clear discussions of range search data structures and algorithms in his text [Samet, 1990]. Finally, I would like to thank Dr. Thomas Papadakis of York University in Toronto, Ontario, Canada, for providing me with his source code for deterministic skip lists.

References

Bentley, J.L. “Multidimensional binary search trees used for associative searching”, *Communications of the ACM*, **18**(9), 1975, pp.509-517.

Bentley, J.L. “Decomposable searching problems”, *Information Processing Letters*, **8**(5), June, 1979, pp.244-251.

Bentley, J.L and Friedman, J.H. “Data structures for range searching”, *Computing Surveys*, **11**(4), Dec. 1979, pp.397-409.

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

Chazelle, B. “Filtering search: a new approach to query-answering”, *SIAM Journal on Computing*, **15**(3), 1986, pp.703-724.

Comer, D. “The ubiquitous B-tree”, *Computing Surveys*, **11**(2), 1979, pp.121-137.

Edelsbrunner, H. “A note on dynamic range searching”, Bulletin of the European Association for Theoretical Computer Science (EATCS), No. 15, Oct. 1981, pp.34-40.

Friedman, J.H., Baskett, F. and Shustek, L.J. “An algorithm for finding nearest neighbors”, *IEEE Transactions on Computers*, **24**(10), 1975, pp.1000-1006.

Hanson, E.N. and Johnson, T. “The interval skip list: a data structure for finding all intervals that overlap a point”, Technical report UF-CIS-92-016, University of Florida, Computing and Information Sciences Department, June 16, 1992. (available from anonymous ftp site “`ftp.cis.ufl.edu`” in file “`cis/tech-reports/tr92/tr92-016.ps.Z`”).

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

McCreight, E.M. “Priority search trees”, *SIAM Journal of Computing*, 14(2), May 1985, pp.257-276.

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

Lueker, G.S. “A data structure for orthogonal range queries”, 19th Annual IEEE Symposium on Foundations of Computer Science, Ann Arbor, Michigan, Oct. 16-18, 1978, pp.28-34.

Overmars, M.H. “Geometric data structures for computer graphics: an overview”, in Theoretical Foundations of Computer Graphics and CAD, edited by R.A. Earnshaw, Springer-Verlag, Berlin, Vol. 40 of Series F: Computer and Systems Sciences of the NATO Advanced Science Institutes Series, 1988, pp.21-49.

Papadakis, T. “Skip Lists and Probabilistic Analysis of Algorithms”, Ph.D. thesis, University of Waterloo, 1993. (available from anonymous ftp site “`cs-archive.uwaterloo.ca`”, in file “`cs-archive/CS-93-28`”).

Papadakis, T., Munro, J.I. and Poblete, P.V. “Average search and update costs in skip lists”, *BIT*, **32**(2), 1992, pp.316-332.

Pugh, W. "Skip lists: a probabilistic alternative to balanced trees", *Comm. of the ACM*, **33**(6), June 1990, pp.668-676.

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

Willard, D.E. “New data structures for orthogonal range queries”, *SIAM Journal on Computing*, **14**(1), 1985, pp.232-253.

Willard, D.E. and Lueker, G.S. “Adding range restriction capability to dynamic data structures”, *Journal of the Association for Computing Machinery*, **32**(3), July 1985, pp.597-617.